GBASE®

GBase 8s ESQL/C 编程指南

www.gbase.cn

GBase 8s ESQL/C 编程指南，南大通用数据技术股份有限公司

GBase 版权所有©2004-2030，保留所有权利

版权声明

本文档所涉及的软件著作权及其他知识产权已依法进行了相关注册、登记，由南大通用数据技术股份有限公司合法拥有，受《中华人民共和国著作权法》、《计算机软件保护条例》、《知识产权保护条例》和相关国际版权条约、法律、法规以及其它知识产权法律和条约的保护。未经授权许可，不得非法使用。

免责声明

本文档包含的南大通用数据技术股份有限公司的版权信息由南大通用数据技术股份有限公司合法拥有，受法律的保护，南大通用数据技术股份有限公司对本文档可能涉及到的非南大通用数据技术股份有限公司的信息不承担任何责任。在法律允许的范围内，您可以查阅，并仅能够在《中华人民共和国著作权法》规定的合法范围内复制和打印本文档。任何单位和个人未经南大通用数据技术股份有限公司书面授权许可，不得使用、修改、再发布本文档的任何部分和内容，否则将视为侵权，南大通用数据技术股份有限公司具有依法追究其责任的权利。

本文档中包含的信息如有更新，恕不另行通知。您对本文档的任何问题，可直接向南大通用数据技术股份有限公司告知或查询。

通讯方式

南大通用数据技术股份有限公司

天津市高新区开华道22号普天创新产业园东塔20-23层

电话：400-013-9696　　邮箱：info@gbase.cn

商标声明

GBASE®是南大通用数据技术股份有限公司向中华人民共和国国家商标局申请注册的注册商标，注册商标专用权由南大通用数据技术股份有限公司合法拥有，受法律保护。未经南大通用数据技术股份有限公司书面许可，任何单位及个人不得以任何方式或理由对该商标的任何部分进行使用、复制、修改、传播、抄录或与其它产品捆绑使用销售。凡侵犯南大通用数据技术股份有限公司商标权的，南大通用数据技术股份有限公司将依法追究其法律责任。

目　录

1 简介 1
1.1 ESQL/C 指南 1
1.2 符合行业标准 1
1.3 演示数据库 1
1.4 如何阅读语法图 2
1.5 示例代码约定 3
2 GBase 8s ESQL/C 是什么？ 4
2.1 GBase 8s ESQL/C 编程 4
2.1.1 GBase 8s ESQL/C 是什么? 4
2.1.2 嵌入式 SQL 语句 6
2.1.3 声明和使用主变量 10
2.1.4 ESQL/C 头文件 26
2.1.5 ESQL/C 预处理器伪指令 29
2.1.6 在 Windows™ 环境中设置和检索环境变量 32
2.1.7 Windows 环境中的全局 ESQL/C 变量 38
2.1.8 GBase 8s ESQL/C 程序的样本 39
2.2 程序指令 42
2.2.1 编译 ESQL/C 程序 42
2.2.2 esql 命令 44
2.2.3 编译和链接 esql 命令的选项 63
2.2.4 在 Windows(TM) 环境中访问 ESQL 客户端接口 71
2.3 GBase 8s ESQL/C 数据类型 72
2.3.1 为主机变量选择数据类型 73
2.3.2 数据转换 81
2.3.3 数据类型对齐库函数 87
2.4 字符和字符串数据类型 88
2.4.1 字符数据类型 88
2.4.2 获取并插入字符数据类型 95
2.4.3 字符和字符串库函数 101

2.5 Numeric 数据类型.....103
2.5.1 整型数据类型.....103
2.5.2 BOOLEAN 数据类型.....107
2.5.3 Decimal 数据类型.....108
2.5.4 浮点数据类型.....112
2.5.5 格式化数值字符串.....114
2.6 时间数据类型.....121
2.6.1 SQL DATE 数据类型.....121
2.6.2 DATE 库函数.....122
2.6.3 SQL DATETIME 和 INTERVAL 数据类型.....122
2.6.4 支持非 ANSI DATETIME 格式.....130
2.6.5 DATETIME 和 INTERVAL 库函数.....130
2.7 简单大对象.....132
2.7.1 选择大对象数据类型.....132
2.7.2 使用简单大对象编程.....133
2.7.3 在内存中找到简单大对象.....138
2.7.4 定位文件中的简单大对象.....146
2.7.5 用户定义的简单大对象位置.....158
2.7.6 dispcat_pic 程序.....165
2.8 智能大对象.....185
2.8.1 智能大对象数据结构.....185
2.8.2 创建智能大对象.....196
2.8.3 访问智能大对象.....196
2.8.4 获取智能大对象的状态.....202
2.8.5 更改智能大对象列.....204
2.8.6 迁移简单大对象.....205
2.8.7 智能大对象的 ESQL/C API.....205
2.9 复杂数据类型.....209
2.9.1 访问集合.....209
2.9.2 访问行类型.....237

2.10 不透明数据类型.....254
2.10.1 SQL 不透明数据类型.....254
2.10.2 访问不透明类型的外部格式.....257
2.10.3 访问不透明类型的内部格式.....266
2.10.4 lvarchar 指针和 var binary 库函数.....279
2.10.5 访问预定义的不透明数据类型.....280
3 数据库服务器通信.....280
3.1 异常处理.....280
3.1.1 获取 SOL 语句执行后的诊断信息.....281
3.1.2 使用 SQLSTATE 进行异常处理.....282
3.1.3 使用 sqlca 结构进行异常处理.....296
3.1.4 选择异常处理策略.....306
3.1.5 检索错误消息的库函数.....310
3.1.6 使用异常处理的程序.....311
3.2 使用数据库服务器.....323
3.2.1 ESQL/C 应用程序的客户端服务器架构.....323
3.2.2 客户端-服务器连接.....324
3.2.3 与数据库服务器交互.....339
3.2.4 优化消息传输.....350
3.2.5 数据库服务器控制函数.....354
3.2.6 超时程序.....354
3.2.7 Windows(TM) 环境中的 ESQL/C 连接库函数.....379
3.3 GBase 8s 库.....381
3.3.1 选择 GBase 8s 通用库的版本.....381
3.3.2 之前存在的 ESQL/C 应用程序与当前库版本的兼容性.....385
3.3.3 创建线程安全 ESQL/C 应用程序.....388
3.3.4 ESQL/C 线程安全 decimal 函数.....397
3.3.5 上下文线程优化.....397
3.3.6 线程安全程序示例.....398
3.3.7 在 UNIX™ 操作系统上创建动态线程库.....404

4 Dynamic SQL 414
4.1 使用动态 SQL 414
4.1.1 执行动态 SQL 415
4.1.2 数据库游标 422
4.1.3 collect.ec 程序 439
4.1.4 优化 OPEN、FETCH 和 CLOSE 442
4.1.5 一起使用 OPTOFC 与 延迟的 PREPARE 444
4.1.6 在编译时刻知道的 SQL 语句 445
4.1.7 在编译时刻不知道的 SQL 语句 462
4.2 确定 SQL 语句 463
4.2.1 动态管理结构 463
4.2.2 DESCRIBE 语句 470
4.2.3 在运行时刻确定语句信息 478
4.2.4 访存数组 487
4.3 系统描述符区域 509
4.3.1 管理系统描述符区域 509
4.3.2 使用系统描述符区域 516
4.3.3 处理未知的选择列表 517
4.3.4 处理未知的返回值 524
4.3.5 处理未知的列列表 530
4.3.6 处理参数化的 SELECT 语句 536
4.3.7 处理参数化的用户定义的例程 543
4.3.8 处理参数化的 UPDATE 或 DELETE 语句 544
4.3.9 dyn_sql 程序 544
4.4 sqlda 结构 564
4.4.1 管理 sqlda 结构 565
4.4.2 执行 SQL 语句的 sqlda 结构 575
4.4.3 处理未知的选择列表 575
4.4.4 处理未知的返回值 583
4.4.5 处理未知的列列表 585

4.4.6 处理参数化的 SELECT 语句.....586
4.4.7 处理参数化的用户定义的例程.....594
4.4.8 处理参数化的 UPDATE 或 DELETE 语句.....595
5 附录.....595
5.1 ESQL/C 示例程序.....595
5.2 ESQL/C 函数库.....595
5.2.1 GBase 8s ESQL/C库函数.....595
5.2.2 bigintcvasc() 函数.....602
5.2.3 bigintcvdbl() 函数.....602
5.2.4 bigintcvdec() 函数.....603
5.2.5 bigintcvflt() 函数.....603
5.2.6 bigintcvifx_int8() 函数.....604
5.2.7 bigintcvint2() 函数.....604
5.2.8 bigintcvint4() 函数.....605
5.2.9 biginttoasc() 函数.....606
5.2.10 biginttodbl() 函数.....606
5.2.11 biginttodec() 函数.....607
5.2.12 biginttoflt() 函数.....607
5.2.13 biginttoifx_int8() 函数.....608
5.2.14 biginttoint2() 函数.....608
5.2.15 biginttoint4() 函数.....609
5.2.16 bycmpr() 函数.....609
5.2.17 bycopy() 函数.....612
5.2.18 byfill() 函数.....614
5.2.19 byleng() 函数.....616
5.2.20 decadd() 函数.....618
5.2.21 deccmp() 函数.....621
5.2.22 deccopy() 函数.....624
5.2.23 deccvasc() 函数.....626
5.2.24 deccvdbl() 函数.....629

5.2.25 deccvflt() 函数 632
5.2.26 deccvint() 函数 634
5.2.27 deccvlong() 函数 636
5.2.28 decdiv() 函数 638
5.2.29 dececvt() 和 decfcvt() 函数 641
5.2.30 decmul() 函数 648
5.2.31 decround() 函数 651
5.2.32 decsub() 函数 653
5.2.33 dectoasc() 函数 656
5.2.34 dectodbl() 函数 659
5.2.35 dectoint() 函数 662
5.2.36 dectolong() 函数 665
5.2.37 dectrunc() 函数 667
5.2.38 dtaddinv() 函数 669
5.2.39 dtcurrent() 函数 672
5.2.40 dtcvasc() 函数 674
5.2.41 dtcvfmtasc() 函数 677
5.2.42 dtextend() 函数 681
5.2.43 dtsub() 函数 683
5.2.44 dtsubinv() 函数 687
5.2.45 dttoasc() 函数 689
5.2.46 dttofmtasc() 函数 692
5.2.47 GetConnect() 函数（Windows(TM)） 695
5.2.48 ifx_cl_card() 函数 697
5.2.49 ifx_dececvt() 和 ifx_decfcvt() 函数 699
5.2.50 ifx_defmtdate() 函数 701
5.2.51 ifx_dtcvasc() 函数 703
5.2.52 ifx_dtcvfmtasc() 函数 705
5.2.53 ifx_dttofmtasc() 函数 707
5.2.54 ifx_getenv() 函数 709

5.2.55 ifx_getcur_conn_name() 函数 710
5.2.56 ifx_getserial8() 函数 711
5.2.57 ifx_int8add() 函数 712
5.2.58 ifx_int8cmp() 函数 715
5.2.59 ifx_int8copy() 函数 718
5.2.60 ifx_int8cvasc() 函数 720
5.2.61 ifx_int8cvdbl() 函数 724
5.2.62 ifx_int8cvdec() 函数 726
5.2.63 ifx_int8cvflt() 函数 729
5.2.64 ifx_int8cvint() 函数 732
5.2.65 ifx_int8cvlong() 函数 735
5.2.66 ifx_int8div() 函数 737
5.2.67 ifx_int8mul() 函数 740
5.2.68 ifx_int8tub() 函数 742
5.2.69 ifx_int8toasc() 函数 745
5.2.70 ifx_int8todbl() 函数 749
5.2.71 ifx_int8todec() 函数 753
5.2.72 ifx_int8toflt() 函数 758
5.2.73 ifx_int8toint() 函数 761
5.2.74 ifx_int8tolong() 函数 765
5.2.75 ifx_lo_alter() 函数 769
5.2.76 ifx_lo_close() 函数 770
5.2.77 ifx_lo_col_info() 函数 771
5.2.78 ifx_lo_copy_to_file() 函数 772
5.2.79 ifx_lo_copy_to_lo() 函数 773
5.2.80 ifx_lo_create() 函数 774
5.2.81 ifx_lo_def_create_spec() 函数 776
5.2.82 ifx_lo_filename() 函数 776
5.2.83 ifx_lo_from_buffer() 函数 777
5.2.84 ifx_lo_lock() 函数 778

5.2.85 ifx_lo_open() 函数.....780
5.2.86 ifx_lo_read() 函数.....782
5.2.87 ifx_lo_readwithseek() 函数.....782
5.2.88 ifx_lo_release() 函数.....784
5.2.89 ifx_lo_seek() 函数.....785
5.2.90 ifx_lo_spec_free() 函数.....786
5.2.91 ifx_lo_specget_def_open_flags() 函数.....787
5.2.92 ifx_lo_specget_estbytes() 函数.....788
5.2.93 ifx_lo_specget_extsz() 函数.....788
5.2.94 ifx_lo_specget_flags() 函数.....789
5.2.95 ifx_lo_specget_maxbytes() 函数.....790
5.2.96 ifx_lo_specget_sbspace() 函数.....791
5.2.97 ifx_lo_specset_def_open_flags() 函数.....792
5.2.98 ifx_lo_specset_estbytes() 函数.....793
5.2.99 ifx_lo_specset_extsz() 函数.....794
5.2.100 ifx_lo_specset_flags() 函数.....794
5.2.101 ifx_lo_specset_maxbytes() 函数.....795
5.2.102 ifx_lo_specset_sbspace() 函数.....796
5.2.103 ifx_lo_stat() 函数.....797
5.2.104 ifx_lo_stat_atime() 函数.....798
5.2.105 ifx_lo_stat_cspec() 函数.....798
5.2.106 ifx_lo_stat_ctime() 函数.....799
5.2.107 ifx_lo_stat_free() 函数.....800
5.2.108 ifx_lo_stat_mtime_sec() 函数.....800
5.2.109 ifx_lo_stat_refcnt() 函数.....801
5.2.110 ifx_lo_stat_size() 函数.....802
5.2.111 ifx_lo_tell() 函数.....803
5.2.112 ifx_lo_to_buffer() 函数.....803
5.2.113 ifx_lo_truncate() 函数.....804
5.2.114 ifx_lo_unlock() 函数.....805

5.2.115 ifx_lo_write() 函数 806
5.2.116 ifx_lo_writewithseek() 函数 807
5.2.117 ifx_lvar_alloc() 函数 809
5.2.118 ifx_putenv() 函数 810
5.2.119 ifx_strdate() 函数 811
5.2.120 ifx_var_alloc() 函数 812
5.2.121 ifx_var_dealloc() 函数 813
5.2.122 ifx_var_flag() 函数 814
5.2.123 ifx_var_freevar() 函数 816
5.2.124 ifx_var_getdata() 函数 816
5.2.125 ifx_var_getlen() 函数 817
5.2.126 ifx_var_isnull() 函数 818
5.2.127 ifx_var_setdata() 函数 819
5.2.128 ifx_var_setlen() 函数 820
5.2.129 ifx_var_setnull() 函数 821
5.2.130 incvasc() 函数 822
5.2.131 incvfmtasc() 函数 825
5.2.132 intoasc() 函数 829
5.2.133 intofmtasc() 函数 831
5.2.134 invdivdbl() 函数 835
5.2.135 invdivinv() 函数 838
5.2.136 invextend() 函数 840
5.2.137 invmuldbl() 函数 843
5.2.138 ldchar() 函数 846
5.2.139 rdatestr() 函数 847
5.2.140 rdayofweek() 函数 849
5.2.141 rdefmtdate() 函数 852
5.2.142 rdownshift() 函数 856
5.2.143 ReleaseConnect() 函数（Windows(TM)） 858
5.2.144 rfmtdate() 函数 859

5.2.145 rfmtdec() 函数 863
5.2.146 rfmtdouble() 函数 868
5.2.147 rfmtlong() 函数 873
5.2.148 rgetlmsg() 函数 877
5.2.149 rgetmsg() 函数 880
5.2.150 risnull() 函数 882
5.2.151 rjulmdy() 函数 885
5.2.152 rleapyear() 函数 888
5.2.153 rmdyjul() 函数 890
5.2.154 rsetnull() 函数 892
5.2.155 rstod() 函数 897
5.2.156 rstoi() 函数 899
5.2.157 rstol() 函数 902
5.2.158 rstrdate() 函数 904
5.2.159 rtoday() 函数 907
5.2.160 rtypalign() 函数 908
5.2.161 rtypmsize() 函数 912
5.2.162 rtypname() 函数 916
5.2.163 rtypsize() 函数 920
5.2.164 rtypwidth() 函数 921
5.2.165 rupshift() 函数 925
5.2.166 SetConnect() 函数（Windows(TM)） 927
5.2.167 sqgetdbs() 函数 928
5.2.168 sqlbreak() 函数 932
5.2.169 sqlbreakcallback() 函数 934
5.2.170 sqldetach() 函数 936
5.2.171 sqldone() 函数 945
5.2.172 sqlexit() 函数 946
5.2.173 sqlsignal() 函数 946
5.2.174 sqlstart() 函数 948

5.2.175 stcat() 函数.....949
5.2.176 stchar() 函数.....951
5.2.177 stcmpr() 函数.....953
5.2.178 stcopy() 函数.....955
5.2.179 stleng() 函数.....956
5.3 智能大对象函数的示例.....957
5.3.1 前提条件.....958
5.3.2 create_clob.ec 程序.....959
5.3.3 upd_lo_descr.ec 程序.....966

1 简介

1.1 ESQL/C 指南

本主题介绍任何使用 GBase 8s ESQL/C（ C 的嵌入式结构化查询语言（SQL） GBase 8s 实现 （ESQL/C））来创建具有数据库管理功能的客户端应用程序。

这些主题可以作为 GBase 8s ESQL/C 的功能的完整性指南，使您能够在程序中与数据库服务器进行交互、访问数据库、操纵数据，以及检测错误。但是，某些操作系统不支持每个记录的 ESQL/C 功能。检测您的操作系统的 GBase 8s Client Software Development Kit (Client SDK) 机器注意事项，以确定哪些功能在您的环境中无法运行。

这些主题主要从一般主题进展到更高级的编程技术和示例。

本手册主要针对希望在其程序中嵌入 SQL 以访问 GBase 8s 数据库的 C 程序员。

本手册假定您具备以下背景：

对于计算机、操作系统和操作系统提供的实用程序的工作知识

使用关系数据库经验或熟悉数据库概念

一些计算机编程经验

以下用户可能会对这些主题有兴趣：

数据库服务器管理员

性能工程师

有关软件兼容性的信息，请参阅 GBase 8s Client SDK 的发版说明。

这些主题来自 GBase 8s ESQL/C 程序员手册。

1.2 符合行业标准

GBase 8s 产品符合各种标准。

基于 SQL 的 GBase 8s 产品完全兼容 SQL-92 入门级（发布为 ANSI X3.135-1992），这与 ISO 9075:1992 完全相同。另外， GBase 8s 数据库服务器的许多功能都遵守 SQL-92 中级和完全级别以及 X/Open SQL 公共应用程序环境 (CAE) 标准。

1.3 演示数据库

DB-Access 实用程序随 GBase 8s 数据库服务器产品一起提供，它包括一个或多个以下演示数据库：

stores_demo 数据库以一家虚构的体育用品批发商的有关信息举例说明了关系模式。GBase 8s 出版物中的许多示例均基于 **stores_demo** 数据库。

superstores_demo 数据库举例说明了对象关系模式。**superstores_demo** 数据库包含扩展数据类型、类型和表继承以及用户定义的例程的示例。

有关如何创建和填充演示数据库的信息，请参阅《GBase 8s DB-Access 用户指南》。有关数据库及其内容的描述，请参阅《GBase 8s SQL 指南：参考》。

用于安装演示数据库的脚本位于 UNIX™ 平台上的 $GBASEDBTDIR/bin 目录和 Windows™ 环境中的 %GBASEDBTDIR%\bin 目录中。

1.4 如何阅读语法图

语法图使用特殊组件描述语句和命令的语法。

从左到右，从上到下跟随线的路径阅读语法图。

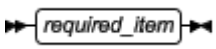
此右侧双箭头加直线符号 ▸▸—— 表示语句开始。

右侧箭头符号 ——▸ 表示语句延续到下一行。

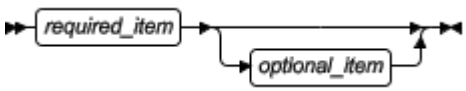
右箭头加直线符号 ▸—— 表示语句继续上一行的内容。

直线、右箭头加左箭头符号 ——▸◂ 表示语句结束。

必需项出现在水平线（主路径）中。

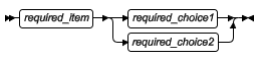


可选项出现在主路径下方。

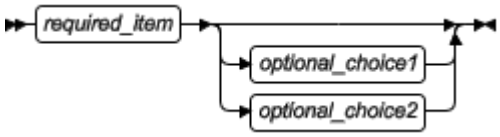


如果可以从两个或多个项中选择，那么它们以堆栈的方式表示。

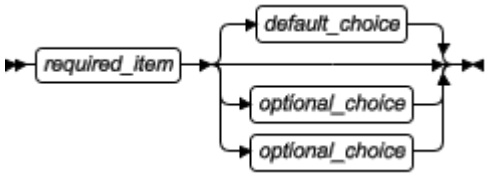
如果必须选择其中一项，那么堆栈中的一项出现在主路径上。



如果从中选择的项是可选的，那么整个堆栈出现在主路径下方。

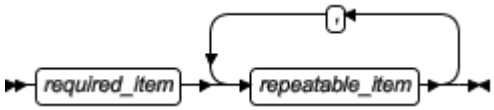


如果缺省其中一项，则它会在主路径上方显示，剩余的选项将会显示在下方。



返回左侧的箭头，在主线之上，表示该项可重复。在此情况下，重复项必须用一个或多个空格隔开。

如果重复的箭头包含一个逗号，那么您必须使用逗号分隔重复的项。

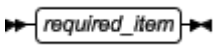


堆栈上方重复的箭头表示可以从堆栈的项目中进行多个选择或者重复一个选择。

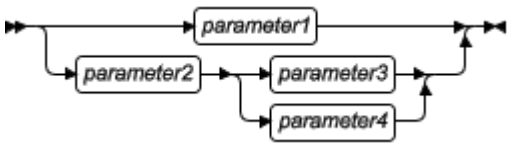
SQL 关键字以大写字母出现 （例如：FROM）。它们必须严格按照所显示的拼写。变量以小写字母出现（例如：column-name）。它们表示用户在语句中提供的名称或值。

如果出现了标点符号、括号、算术运算符或其它这样的符号，那么必须将它们作为语法的一部分输入。

某些时候，一个变量表示一个语句段。例如：在以下语法图中，变量 parameter-block 表示已标记为 parameter-block 的语句段：



parameter-block：



1.5 示例代码约定

SQL 代码的示例在整个出版物中出现。除非另有说明，代码不特定于任何单个的 GBase 8s 应用程序开发工具。

如果示例中仅列出 SQL 语句，那么它们将不用分号定界。例如：您可能看到以下示例中的代码：

```
CONNECT TO stores_demo
...
```

```
    DELETE FROM customer
        WHERE customer_num = 121
    ...

    COMMIT WORK
    DISCONNECT CURRENT
```

要将此 SQL 代码用于特定产品，必须应用该产品的语法规则。例如，如果使用的是 SQL API，那么必须在每条语句的开头使用 EXEC SQL，并在每条语句的结尾使用分号（或其他合适的定界符）。 如果使用的是 DB–Access，那么必须用分号将多条语句隔开。

提示：代码示例中的省略点表示在整个应用程序中将添加更多的代码，但是不必显示它以描述正在讨论的概念。

有关使用特定应用程序开发工具或 SQL API 的 SQL 语句的详细指导，请参阅您的产品文档。

2 GBase 8s ESQL/C 是什么？

2.1 GBase 8s ESQL/C 编程

本手册的最后一章 GBase 8s ESQL/C 程序的样本，显示被注释的 demo1 样本程序。demo1 程序说明了本手册介绍的 GBase 8s ESQL/C 编程的基本概念。

2. 1. 1 GBase 8s ESQL/C 是什么?

GBase 8s ESQL/C 是 SQL 应用程序编程接口（API），它使您可以直接在 C 程序中嵌入结构化查询语言（SQL）语句。

GBase 8s ESQL/C 的预处理器 esql 将每个 SQL 语句和特定于 GBase 8s 的所有代码转换为 C 语言源代码，并启动 C 编译器进行编译它。

ESQL/C 组件

GBase 8s ESQL/C 由以下软件组件构成：

C 函数的 GBase 8s ESQL/C 库，用于访问数据库服务器。

GBase 8s ESQL/C 头文件，提供数据结构、常量以及对 GBase 8s ESQL/C 程序有用的宏定义。

esql 命令，它处理 GBase 8s ESQL/C 源代码以创建一个 C 源文件，并将它传递给 C 编译器。

finderr 实用程序（在 UNIX 系统上）和 GBase 8s Error Messages 实用程序（Windows 系统上），用于获得特定于 GBase 8s 的错误消息的信息。

GBase 8s GLS 语言环境和代码集转换文件，此文件提供了特定语言环境的信息。

有关这些文件的更多信息，请参阅 GBase 8s GLS 用户指南。

用于 Windows 的 ESQL/C 文件

对于 Windows™ 环境，GBase 8s ESQL/C 产品包含下列附加的可执行文件：

Setnet32 实用程序是基于 Windows 的实用程序，用于设置配置文件。

有关更多信息，请参阅 *GBase 8s 客户端产品安装指南*。

ILOGIN 实用程序是打开带有连接参数字段的对话框的示范程序，它用于测试到数据库服务器的连接（使用 **stores7** 数据库）。

有关更多信息，请参阅 *GBase 8s 客户端产品安装指南*。

ESQLMF.EXE 多字节过滤程序，它将多字节字符串中的转义字符更改为十六进制字符。

这些 GBase 8s ESQL/C 可执行文件位于 %GBASEDBTDIR%\bin 、%GBASEDBTDIR%\lib 和 %GBASEDBTDIR%\demo 目录下。%GBASEDBTDIR% 变量代表 GBASEDBTDIR 环境变量的值。

ESQL/C 库函数

GBase 8s ESQL/C 库包含一组可用于应用程序的 C 函数。

这些函数可以归为以下几类：

- 数据类型对齐库函数为不同数据类型的计算机独立大小和对齐信息提供支持，并协助处理空数据库值。
- 字符和字符串库函数提供基于字符的操作，例如比较和复制。
- DECIMAL 库函数支持通过十进制结构访问 DECIMAL 值。
- 格式化函数使您指定不同数据类型的显示格式。
- DATE 库函数支持存取 DATE 值。
- DATETIME 和 INTERVAL 库函数支持存取通过 **datetime** 和 **interval** 结构的值。
- 错误消息函数为获取和格式化特定于 GBase 8s 的错误消息文本提供支持。
- 数据库服务器控制函数使您的应用程序实现例如撤销查询和终止连接的功能。
- INT8 库函数使您能够通过 **int8** 结构存取 INT8 值。
- 智能大对象库函数提供类似于 BLOB 和 CLOB 数据类型的文件。

创建 ESQL/C 程序

创建 GBase 8s ESQL/C 查询：

要在 C 语言源程序中嵌入 GBase 8s ESQL/C 语句，请执行以下步骤：

定义主变量以存储传输于 GBase 8s ESQL/C 和数据库服务器之间的数据。

通过 SQL 语句访问数据库服务器。

提供 GBase 8s ESQL/C 预处理器和 C 编译器的的伪指令。

使用 esql 命令预处理 GBase 8s ESQL/C 源文件以创建 C 语言源文件和启动 C 编译器。

必要时，改正预处理器和编译器报告的错误，并重复步骤 2。

使用 esql 命令，将编译好的对象代码链接到一个或多个可执行文件。

GBase 8s ESQL/C 源文件包含下列类型的语句：

预处理伪指令

GBase 8s ESQL/C 预处理伪指令创建简单的宏定义，包括 GBase 8s ESQL/C 文件，并执行附条件的 GBase 8s ESQL/C 编译。

C 预处理伪指令创建宏定义，包括系统和 C 源文件，并执行附条件的 C 编译。

语言语句

GBase 8s ESQL/C 主变量定义存储传输于 GBase 8s ESQL/C 和数据库服务器之间的数据。

嵌入 SQL 语句以便于数据库服务器通信。

C 语言语句提供程序逻辑。

有关 C 预处理伪指令和 C 语言语句的信息，请参阅 C 编程文本。

GBase 8s ESQL/C 源文件名称可以具有下格式：

esqlc_source.*ec*
esqlc_source.*ecp*

GBase 8s ESQL/C 源文件的特定后缀决定了该源文件由 esql 命令编译的默认顺序。.ec 后缀是缺省的后缀。有关 .ecp 后缀和非缺省编译顺序，请参阅 在运行 ESQL/C 预处理前运行 C 预处理器。

2.1.2 嵌入式 SQL 语句

GBase 8s ESQL/C 程序可以使用 SQL 语句与数据库服务器通信。该程序可以使用*静态*和*动态*SQL 语句。

静态 SQL 语句是指编译程序时所有组件都已知的 SQL 语句。动态 SQL 语句是您在编译时不指定所有组件的语句；该程序在运行时接收全部或部分语句。

可以使用以下两种格式之一将 C 语言嵌入到 C 函数中：

- EXEC SQL 关键字：

```
EXEC SQL SQL_statement;
```

使用 EXEC SQL 关键字是嵌入 SQL 语句的 ANSI 兼容方法。

- 美元符号 ($) ：

```
$SQL_statement;
```

在任一格式中，使用有效语句的完整文本代替 *SQL_statement*。GBase 8s ESQL/C 语句可以在大多数可以使用常量的地方包含主机变量。

嵌入式 SQL 语句中的区分大小写

下表描述了 GBase 8s ESQL/C 预处理程序如何对待大写和小写字母。

表 1. ESQL/C 文件中区分大小写

ESQL/C 标识符	区分大小写	示例

主机变量	是	GBase 8s ESQL/C 将变量 **fname** 和 **Fname** 视为不同的变量：
		`EXEC SQL BEGIN DECLARE SECTION;`
		`char fname[16], lname[16];`
		`char Fname[16];`
		`EXEC SQL END DECLARE SECTION;`
		该样本不会从预处理器生成警告。

变量类型	是	GBase 8s ESQL/C 和 C 将数据类型的名称都视为区分大小写。以下示例中的 CHAR 类型与 **char** 数据类型不同，它会生成错误：
		`EXEC SQL BEGIN DECLARE SECTION;`
		`char fname[16], lname[16];`
		`CHAR Fname[16];`
		`EXEC SQL END DECLARE SECTION;`
		然而，如果为它提供一个 typedef 语句 CHAR 类型不会生成错误。在以下示例中，**CHAR** 类型不会生成错误：
		`typedef char CHAR;`
		`EXEC SQL BEGIN DECLARE SECTION;`
		`char fname[16], lname[16];`
		`CHAR Fname[16];`
		`EXEC SQL END DECLARE SECTION;`

SQL 关键	否	两个 CONNECT 语句是建立与 **stores7** 演示数据库的连接的有效方法：

字		EXEC SQL CONNECT TO 'stores7'; or EXEC SQL connect to 'stores7'; 在本出版物中给出的示例中，SQL 关键字都以小写字母显示。
语句标识符 游标名称	否	以下示例显示语句 ID 和游标名称的创建： EXEC SQL prepare st from 'select * from tab1'; /* duplicate */ EXEC SQL prepare ST from 'insert into tab2 values (1,2)'; EXEC SQL declare curname cursor for st; /* duplicate */ EXEC SQL declare CURNAME cursor for ST; 该代码产生错误，因为语句 ID **st** 和 **ST** 是重复的，游标名称 **curname** 和 CURNAME 也是相同的。

引号符号和转义字符

转义字符指示 GBase 8s ESQL/C 预处理器应将字符打印为文字字符而不是对其进行解释。可以使用带有解释字符的转义字符来使编译器转义或忽略解释的含义。

在 ANSI SQL 中，反斜杠字符（\）是转义字符。要搜索以字符串 \abc 开始的数据，WHERE 子句必须如下使用转义字符：

```
... where col1 = '\\abc';
```

但是，ANSI 标准指定使用反斜杠字符（\）转义单引号（''）或双引号（""）是无效的。例如，以下查找引号的尝试不符合 ANSI 标准：

```
... where col1 = '\'';
```

在非嵌入式工具（例如 DB-Access）中，可以使用以下方法转义引号符号：

可以使用与转义字符相同的引号，如下所示：

```
... where col1 = '''';
```

可以使用另一种引号。例如，要查找双引号，可以将此双引号用单引号括起，如下所示：

```
... where col1 = ' "';
```

下图显示一个带有 WHERE 子句的 SELECT 语句，其中包含引号括起来的双引号。

图: 具有无效 WHERE 子句的 SELECT 语句

```
EXEC SQL select col1 from tab1 where col1 = ' "';
```

对于表 1 中的 WHERE 子句，GBase 8s ESQL/C 预处理器不会处理双引号；它将此

双引号传递给 C 编译器。当 C 编译器接收带字符串 '"' （引号括起来的双引号），它将第一个引号解释为字符串的开始，双引号解释为字符串的结束。编译器不会匹配剩下的引号因此产生错误。

要使 C 编译器将双引号解释为一个字符，请在双引号之前带有 C 转义字符——反斜杠（\）。以下示例显示表 1中查询的正确语法：

```
EXEC SQL select col1 from tab1 where col1 = '\"';
```

因为 C 和 ANSI SQL 都使用反斜杠字符作为转义字符，当您在嵌入式查询中查找字符反斜杠时，请留意。以下查询显示查找字符串 ″\″ 的正确语法（双引号作为字符串的一部分）：

```
EXEC SQL select col1 from tab1 where col1 = '\"\\\\\"';
```

此字符串需要五个反斜杠以获取正确的释义。三个反斜杠是转义字符，一个用于双引号，一个用于反斜杠。下表显示了它通过每个处理步骤处理后的字符串。

表 1. 转义查询字符串处理后

处理器	处理后
ESQL/C 预处理器	'\"\\\\\"'
C 编译器	"\"\\\""
符合 ANSI 的数据库服务器	"\"\""

GBase 8s ESQL/C 支持单引号（'*string*'）或双引号（"*string*"）。但是，C 语言字符串仅支持双引号中的字符串。因此，GBase 8s ESQL/C 预处理器将 GBase 8s ESQL/C源文件中的每一条语句转换为双引号字符串。

引号字符串中的换行符

GBase 8s ESQL/C不支持引号字符串中的换行符（0x0A）。但是，如果您指定您希望允许它，数据库服务器不允许在引号字符串中使用换行符。因此，您可以将引用的字符串包含在动态的准备好的 SQL 语句的一部分中，因为数据库服务器而不是 GBase 8s ESQL/C 处理准备好的语句。

可以指定您希望数据库服务器以引号字符串的形式允许每个会话或全部会话的换行符。会话是客户端连接到数据库服务器的持续时间。

要允许或禁止特定会话的引号字符串中的换行符，您必须执行用户定义的例程 ifx_allow_newline(boolean)。以下示例说明了如何启动 ifx_allow_newline() 用户定义的例程以在引用字符串中允许换行符：

```
EXEC SQL execute procedure ifx_allow_newline('t');
```

要在引号字符串中禁用换行符，将参数更改为 f，如下所示：

```
EXEC SQL execute procedure ifx_allow_newline('f');
```

要在所有会话的引号字符串中的允许或禁用换行符，设置 onconfig 文件中的

ALLOW_NEWLINE 参数。值 0 表示禁止引用换行符。

向 ESQL/C 程序添加注释

要在 GBase 8s ESQL/C 程序中添加注释，可以使用以下任一格式：

可以在任意 GBase 8s ESQL/C 语句上使用双重破折号(--)。该语句比较以 EXEC SQL 或 $ 开头并以分号结尾。该注释继续到行的末尾。

例如，以下第一行的注释说明 GBase 8s ESQL/C 语句打开 **stores7** 演示数据库：

```
EXEC SQL database stores7;     -- stores7 database is open now!
                    printf("\nDatabase opened\n");   /* This is not an ESQL/C */
                    /* line so it needs a       */
                    /* regular C notation       */
                    /* for a comment */
```

可以在 GBase 8s ESQL/C 行使用标准的 C 注释，如下所示：

```
EXEC SQL begin work;   /* You can also use a C comment here */
```

主机变量

主机变量是在嵌入式 SQL 语句中使用的 GBase 8s ESQL/C 或 C 变量，用于在数据库列和 GBase 8s ESQL/C 程序之间传输数据。

当在 SQL 语句中使用主机变量时，必须先在其名称前面加上一个符号，以将其标识为主变量。您可以使用以下符号之一：

冒号（:）

例如：将名为 **hostvar** 的主机变量指定为连接名称，使用以下语法：

```
EXEC SQL connect to :hostvar;
```

使用冒号（:）作为主机变量的前缀，符合 ANSI SQL 标准。

美元符号（$）

例如：将名为 **hostvar** 的主机变量指定为连接名称，使用以下语法：

```
EXEC SQL connect to $hostvar;
```

当在一条 SQL 语句中列出多个主机变量是，使用逗号（,）将主变量分隔。例如：esql 命令将下行解释为两个主变量 **host1** 和 **host2**：

```
EXEC SQL select fname, lname into :host1, :host2 from customer;
```

如果省略逗号，esql 会将第二个变量解释为第一个主变量的指示变量。esql 命令将下行解释为一个主变量 **host1** 和一个 **host1** 主变量的指示变量 **host2** ：

```
EXEC SQL select fname, lname into :host1 :host2 from customer;
```

在 SQL 语句之外，像常规的 C 变量一样处理主机变量。

2.1.3 声明和使用主变量

在 GBase 8s ESQL/C 应用程序中，SQL 语句可以引用*主变量*的内容。主变量是 GBase 8s ESQL/C 程序变量，它用于在 GBase 8s ESQL/C 程序和数据库服务器之间传输信

息。

可以使用 GBase 8s ESQL/C 表达式中的主变量，方法与使用文字值相同，但不能使用它们：

在已准备好的语句中

在存储过程中

在检查约束中

在视图中

在触发器中

作为字符串连接操作的一部分

要在 SQL 语句中使用主变量：

在 C 程序中声明主变量。

给主变量指定值。

在嵌入式 SQL 语句中指定主变量。

声明主机变量

在 GBase 8s ESQL/C 程序中可以使用该变量之前，必须定义主机变量所需的数据存储。要为变量指定一个标识符并将其与数据类型相关联，然后再声明此变量。

在 GBase 8s ESQL/C 程序中将主变量声明为 C 变量，使用与 C 变量相同的基本语法。

要标识该变量为主机变量，必须使用以下方式之一声明它：

将声明放置在 ESQL 声明段中：

```
EXEC SQL BEGIN DECLARE SECTION;
        -- 将主机变量声明放置于此处
        EXEC SQL END DECLARE SECTION;
```

确定使用分号结束 EXEC SQL BEGIN DECLARE SECTION 和 EXEC SQL END DECLARE SECTION 。

使用 EXEC SQL BEGIN DECLARE SECTION 和 EXEC SQL END DECLARE SECTION 关键字符合 ANSI 标准。

使用美元符号（$）表示每个声明。

在声明本身中，您必须指定以下信息：

主机变量的名称

主机变量的数据类型

主机变量的初始值（可选）

主机变量的范围（由程序中放置的声明决定）

主机变量名称

主机变量的名称必须符合 C 语言的命名规范。此外，必须遵循 C 编译器施加的任何限制。

通常情况下，C 变量必须以一个字母或下划线开头，并且可以包含字母和数字以及下划线。

重要： GBase 8s ESQL/C 产品中使用的许多变量名称以下划线开始。要避免与内部 GBase 8s ESQL/C 变量名称的冲突，请避免变量名称的第一个字符使用下划线。

C 变量名称是区分大小写的，因此变量 **hostvar** 和 **HostVar** 是不同的。

如果您的客户端语言环境支持非 ASCII 字符，那么可以在 GBase 8s ESQL/C 中使用非 ASCII （非英语）字符。

提示： 好的编程习惯需要您为主机变量名称创建一个命名规范。

主机变量数据类型

因为主机变量是 C 变量，您必须在声明它时给它指定一个 C 数据类型。否则，当您在 SQL 语句中使用主机变量时，必须将它与 SQL 数据类型关联起来。

可以声明主机变量为许多更复杂的 C 数据类型，例如指针、结构、**typedef** 表达式和函数参数。

初始主机变量值

可以使用 GBase 8s ESQL/C 声明具有正常 C 初始化表达式的主机变量。

以下示例显示有效的 C 初始值：

```
EXEC SQL BEGIN DECLARE SECTION;
    int varname = 12;
    long cust_nos[8] = {0,0,0,0,0,0,0,9999};
    char descr[100] = "Steel eyelets; Nylon cording.";
    EXEC SQL END DECLARE SECTION;
```

GBase 8s ESQL/C 预处理器不会检查有效 C 语法的初始化表达式；它将其复制到 C 源文件中。C 编译器诊断任何错误。

主机变量的范围

主机变量的引用的范围或范围是可以访问主变量的程序的一部分。

GBase 8s ESQL/C 声明语句的放置位置决定了变量的范围如下：

如果声明语句在程序块中，则该变量是程序块的*本地变量*。

只有在此程序块中的语句才能访问此变量。

如果声明语句在程序块之外，那么该变量是模块化的。

在声明之后发生的所有程序都可以访问此变量。

在代码内声明的主变量是该块的本地变量。可以使用一对花括号（{ }）定义一个代码块。

例如：下图中的主变量 **blk_int** 仅在大括号之间的代码块中有效，而 **p_int** 在块内部和外部都有效。

图: 在代码块内部和外部声明主变量

```
EXEC SQL BEGIN DECLARE SECTION;
        int p_int;
        EXEC SQL END DECLARE SECTION;
        ⋮
        EXEC SQL select customer_num into :p_int from customer
        where lname = "Miller";
        ⋮
        {
        EXEC SQL BEGIN DECLARE SECTION;
        int blk_int;
        EXEC SQL END DECLARE SECTION;

        blk_int = p_int;

        ⋮
        EXEC SQL select customer_num into :blk_int from customer
        where lname = "Miller";
        ⋮
        }
```

可以嵌套块至 16 层。全局层是第一层。

以下 C 规则也适用于 GBase 8s ESQL/C 主变量的范围：

主变量是一个自动变量，除非您将其明确定义为**外部**或**静态**变量或者在任何函数外部定义它。

函数声明的主变量是此函数的局部变量，并在函数外面屏蔽具有相同名称的定义。

不能在同一代码块中多次定义主变量。

主机变量声明示例

下图显示了如何使用 EXEC SQL 语法声明主机变量的一个示例：

图: 使用 EXEC SQL 语法声明主机变量

```
EXEC SQL BEGIN DECLARE SECTION;
          char     *hostvar;        /* pointer to a character */
          int      hostint;         /* integer */
          double hostdbl;           /* double */
          char     hostarr[80];    /* character array */
          struct {
          int svar1;
          int svar2;
```

```
        ⋮

        } hoststruct;        /* structure */
        EXEC SQL END DECLARE SECTION;
```

下图显示了如何使用美元符号（$）声明主机变量：

图: 使用美元符号（$）声明主机变量

```
$char   *hostvar;
        $int      hostint;
        $double hostdbl;
        $char     hostarr[80];

        $struct {
        int svar1;
        int svar2;

        ⋮

        } hoststruct;
```

主机变量信息

可以使用主变量包含下列种类的信息：

SQL 标识符

SQL 标识符包括部分数据库的名称例如：表、列、索引和视图。

数据

数据是数据库服务器从数据库中获取或存储的信息。该信息可以包括空值。空值表示列或变量的值是未知的。

主变量可以在语法允许的情况下显示在 SQL 语句中。但是，必须在主变量名称前面加上一个符号以区分它和常规 C 变量。

SQL 标识符

SQL 标识符是数据库对象的名称。

以下对象是 SQL 标识符的示例：

数据库模式的一部分，例如：表、列、视图、索引、同义词以及存储过程名称。

动态 GBase 8s ESQL/C 结构，例如游标和语句 ID。

根据语法允许，可以在嵌入式 SQL 语句中使用主机变量来保存 SQL 标识符的名称。

长标识符

GBase 8s 允许长度为 128 个字符的标识符，长度可达到 32 个字符的用户名。

数据库服务器使用以下两个标准来确定客户端程序是否可以接收长标识符：

客户端程序的内部版本号

IFX_LONGID 环境变量的设置

如果将 IFX_LONGID 环境变量设置为 0 （零），那么数据库服务器将此客户端视为它不能处理长标识符。如果 **IFX_LONGID** 设置为 1（一）并且客户端版本是最新版本，那么数据库服务器将客户端视为能够接收长标识符。如果未设置 IFX_LONGID，则将其视为设置为 1 。

重要： 如果在客户端的环境中设置了 **IFX_LONGID**，那么它仅对此客户端生效。如果在数据库服务器的环境中设置了 **IFX_LONGID** 环境变量，那么它对所有的客户端生效。

满足以下条件的客户端程序可以使用长标识符和长用户名，而无需重新编译：

它使用高于 9.20 版本的 ESQL/C 编译

它使用共享库（即，程序编译时没有 -static 选项）

如果数据库服务器删除长标识符或长用户名，那么将 SQLSTATE 变量设置为 '01004'，并将 SQL 通信区域中（**sqlca**）**sqlwarn1** 标记设置为 'W'。

分隔标识符

如果标识符的名称不符合命名规范，那么您必须使用*分隔标识符*。分隔标识符是使用双引号（""）括起的 SQL 标识符。

当使用双引号分隔标识符时，您符合 ANSI 标准；单引号（''）分隔字符串。

当程序必须指定一些语法上无效的标识符名称时，使用分隔标识符。可能无效的标识符包括：

与 SQL 保留字相同的标识符。

有关 SQL 保留字的列表，请参 *GBase 8s SQL 指南：语法*中标识符的描述。

不含有字母字符的标识符。

要使用分隔标识符，您必须编译并运行 GBase 8s ESQL/C 程序，并设置 DELIMIDENT 。您可以在以下任一阶段设置 DELIMIDENT ：

在编译时，GBase 8s ESQL/C 预处理器允许在标识符有效的 SQL 语法的区域中引用字符串。

在运行时，数据库服务器接受在标识符有效的动态 SQL 语句中引用字符串。

数据库实用程序（例如 **dbexport**）和 DB-Access 也支持分隔标识符。

重要： 当使用 DELIMIDENT 环境变量时，您可以不再使用双引号来分隔字符串。如果您要指示引用字符串，那么使用单引号 （''）括起此文本。

分隔标识符区分大小写。所有您放置在引号中的数据库对象名称会保留其大小写。请

牢记 GBase 8s ESQL/C 限制标识符名称的最大长度为 128 个字符。

以下限制应用于分隔标识符T：

不能对数据库名称使用分隔标识符。

不能对存储标识符、实例以及 dbspace 的名称使用分隔标识符。

DELIMIDENT 环境变量仅适用于数据库标识符。

分隔标识符的示例

下图显示了一个分隔标识符，用于指定游标名称和语句 ID 中不具有字母的字符。

图: 对游标名称使用分隔标识符

```
EXEC SQL BEGIN DECLARE SECTION;
        char curname1[10];
        char stmtname[10];
        EXEC SQL END DECLARE SECTION;

        stcopy("%#!", curname1);
        stcopy("(_=", stmtname);
        EXEC SQL prepare :stmtname from
        'select customer_num from customer';
        EXEC SQL declare :curname1 cursor for $stmtname;
        EXEC SQL open :curname;
```

在上图中，还可以直接在 SQL 语句中列出游标名称或语句 ID。例如，以下 PREPARE 语句也有效（设置了 DELIMIDENT ）：

```
EXEC SQL prepare "%#!" from
        'select customer_num from customer';
```

如果设置了 DELIMIDENT，那么 PREPARE 语句之前的 SELECT 再出发必须用引号括起，以便预处理器将其视为字符串。如果将此语句用双引号括起，那么预处理器将其视为标识符。

要声明包含双引号的游标名称，您必须在分隔标识符字符串中使用转义字符。例如：要使用字符串 ″abc″ 作为游标名称，必须转义游标名称中第一个引号：

```
EXEC SQL BEGIN DECLARE SECTION;
        char curname2[10];
        char stmtname[10];
        EXEC SQL END DECLARE SECTION;

        stcopy("\"abc\"", curname2);
        EXEC SQL declare :curname2 cursor for :stmtname;
```

在前面的示例中，游标名称需要以下几个转义字符：

反斜杠（\）是 C 转义字符。您需要它转义双引号。

若没有转义字符，那么 C 编译器将此双引号解释为字符串的末尾。

游标名称必须包含两个双引号。

第一个双引号转义双引号，第二个双引号是字符双引号。ANSI 标准表明您不能使用反斜杠转义引号。必须在游标名称中使用另一个引号转义此双引号。

下表显示了包含已经处理过的游标名称的字符串。

表 1. 处理后的转义游标名称字符串

处理器	处理后
ESQL/C 预处理器	\"\"abc
C 编译器	""abc
符合 ANSI 的数据库服务器	"abc

主机变量中的空值

空值表示未知或不可用的值。该值区别于任何给出的数据类型的合法值。

空值的表示取决于计算机和数据类型。通常，该表示法不对应与 C 数据类型的合法值。不要尝试对包含空值的主机变量执行计算或其它操作。

因此，程序必须具有识别空值的一些方法。要处理空值，GBase 8s ESQL/C 提供了以下功能：

risnull() 和 rsetnull() 库函数用于测试主变量是否包含空值，是否将主变量设置为空值。

指示变量是特殊的 GBase 8s ESQL/C 变量，它可以容纳允许为空值的数据库列值的主机变量。

指示变量的值显示关联的主机变量是否包含空值。

在符合 ANSI 的数据库中，在 INSERT 语句或任何 SQL 语句的 WHERE 子句中的主机变量必须为空终止。

数据结构中的主机变量

GBase 8s ESQL/C 支持在下列数据结构中使用主机变量：

数组

C 结构（**struct**）

C **typedef** 语句

指针

函数参数

主机变量的数组

GBase 8s ESQL/C 支持主机变量的数组的声明。当声明此数组时必须提供整数值作为数组的大小。主机变量的数组可以是一维或二维。

可以在 GBase 8s ESQL/C 语句中使用数组的元素。例如：如果您提供以下声明：

```
EXEC SQL BEGIN DECLARE SECTION;
        long customer_nos[10];
        EXEC SQL END DECLARE SECTION;
```

您可以使用以下语法：

```
for (i=0; i<10; i++)
        {
        EXEC SQL fetch customer_cursor into :customer_nos[i];

        ⋮
        }
```

如果数组是 CHAR 类型，那么也可以在某些 SQL 语句中单独使用数组名称。

将 C 结构作为主机变量

GBase 8s ESQL/C 支持将 C 结构（**struct**）声明为主机变量。可以使用 GBase 8s ESQL/C 语句中的结构组件。

以下 **cust_rec** 变量的定义，用于 **stores7** 数据库中 **customer** 表的前三列的主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        struct customer_t
        {
        int     c_no;
        char    fname[32];
        char    lname[32];
        } cust_rec;
        EXEC SQL END DECLARE SECTION;
```

以下 INSERT 语句在其 VALUES 子句中指 **cust_rec** 主变量的组件：

```
EXEC SQL insert into customer (customer_num, fname, lname)
        values (:cust_rec.c_no, :cust_rec.fname,
        :cust_rec.lname);
```

如果 SQL 语句需要单个主机变量，您必须使用结构名称来指定主机变量。 GBase 8s 在 UPDATE 语句的 SET 子句中需要组件名称。

在允许主机变量的 SQL 语句中，可以指定 C 结构的名称并且 GBase 8s ESQL/C 将结构变量的名称扩展到每个组件元素。可以将语法与 SQL 语句配合使用（例如：带有 INTO 子句的 FETCH 语句或带有 VALUES 子句的 INSERT 语句）。

下面的 INSERT 语句在其 VALUES 子句中指定整个 **cust_rec** 结构：

```
EXEC SQL insert into customer (customer_num, fname, lname)
            values (:cust_rec);
```

该插入操作执行与指定 **cust_rec** 结构的各个组件名称的插入操作相同。

使用 C typedef 语句作为主机变量

GBase 8s ESQL/C 支持 C **typedef** 语句并允许在主机变量的类型声明中使用

typedef 名称。

例如，以下代码创建 **smallint** 类型为短整型，**serial** 类型为长整型。然后声明 **row_nums** 变量为 **serial** 变量的数组，变量 **counter** 为 **smallint** 。

```
EXEC SQL BEGIN DECLARE SECTION;
        typedef short smallint;
        typedef long serial;
        serial row_nums [MAXROWS];
        smallint counter;
EXEC SQL END DECLARE SECTION;
```

不能使用将多维数组或联合或函数指针命名的 **typedef** 语句作为主变量的类型。

使用指针作为主机变量

如果程序使用指针将数据输入到 SQL 语句，那么您可以使用指针作为主机变量。

例如，下图显示了如何将游标和语句相关联并将值插入到表中。

图: 声明字符指针以输入数据

```
EXEC SQL BEGIN DECLARE SECTION;
          char *s;
          char *i;
          EXEC SQL END DECLARE SECTION;

          /* Code to allocate space for two pointers not shown */

          s = "select * from cust_calls";
          i = "NS";

          ⋮

EXEC SQL prepare x from :s;
EXEC SQL insert into state values (:i, 'New State');
```

下图显示了如何使用整数指针将数据输入到 INSERT 语句中。

图: 声明整数指针以输入数据

```
EXEC SQL BEGIN DECLARE SECTION;
        short *i;
        int *o;
        short *s;
        EXEC SQL END DECLARE SECTION;

        short i_num = 3;
        int o_num = 1002;
        short s_num = 8;

        i = &i_num;
        o = &o_num;
        s = &s_num;
```

```
EXEC SQL connect to 'stores7';
EXEC SQL insert into items values (:*i, :*o, :*s, 'ANZ', 5, 125.00);
EXEC SQL disconnect current;
```

如果使用的主机变量是指向 **char** 的指针，以接收来自 SELECT 语句的数据，那么您会接收到一个编译警告并且结可能会截断结果。

函数参数作为主机变量

可以使用主机变量作为函数的参数。您必须在主机变量的名称的前面加上 **parameter** 关键字以声明它为一个函数参数。

例如，下图显示了一个具有 Kernighan 和 Ritchie-Style 原型声明的代码片段，有三个参数其中两个是主机变量。

图: 使用 EXEC SQL 将主机变量声明为 Kernighan 和 Ritchie-Style 函数声明的参数

```
f(s, id, s_size)
EXEC SQL BEGIN DECLARE SECTION;
        PARAMETER char s[20];
        PARAMETER int id;
EXEC SQL END DECLARE SECTION;
        int s_size;
        {
        select fname into :s from customer
        where customer_num = :id;
        ⋮

        }
```

还可以使用美元符号（$）声明参数主机变量。例如：下图显示了图 1中的函数头，它使用美元符号（$）。

图: 使用美元符号（$）将主机变量声明为函数的参数

```
f(s, id, s_size)
$parameter char s[20];
$parameter int id;
int s_size;
```

可以将 ANSI 样式的原型函数声明中的参数声明为主变量。还可以将所有参数放入 EXEC SQL 声明部分中的原型函数声明，即使某些参数不能用作主机变量。下图显示了函数指针 **f** 可以包含在 EXEC SQL 声明部分中，即使它不是有效的主机变量并且不能作为主机变量使用。

图: 使用 EXEC SQL 将主机变量声明为 ANSI 样式函数声明的参数

```
int * foo(
        EXEC SQL BEGIN DECLARE SECTION;
        PARAMETER char s[20],
        PARAMETER int id,
        PARAMETER int (*f) (double)
```

```
EXEC SQL END DECLARE SECTION;
)
{
select fname into :s from customer
where customer_num = :id;
⋮

}
```

允许在 EXEC SQL 声明段中包含函数参数的功能符合与在 EXEC SQL 声明段中允许使用任何有效的 C 声明语法的要求，来为 C 和 GBase 8s ESQL/C 源文件使用注释头文件。

重要： 如果您想要定义 ANSI 样式参数的 GBase 8s ESQL/C 主机变量，那么必须使用 EXEC SQL BEGIN DECLARE SECTION 和 EXEC SQL END DECLARE SECTION 语法。不能使用 $BEGIN DECLARE 和 $END DECLARE 语法。因为以美元符号（$）开头的 SQL 语句必须以分号结尾。但是，ANSI 语法要求参数列表中的每个参数不能以分号结束符结束，但是可以用逗号（,）分隔符。

下列限制应用于使用主机变量作为函数参数：

不能在 C 代码块中声明参数变量。

不能在不是函数头部分的主机变量的声明中使用 **parameter** 关键字。如果这样做，您将收到不可预知的结果。

Windows 环境中的主机变量

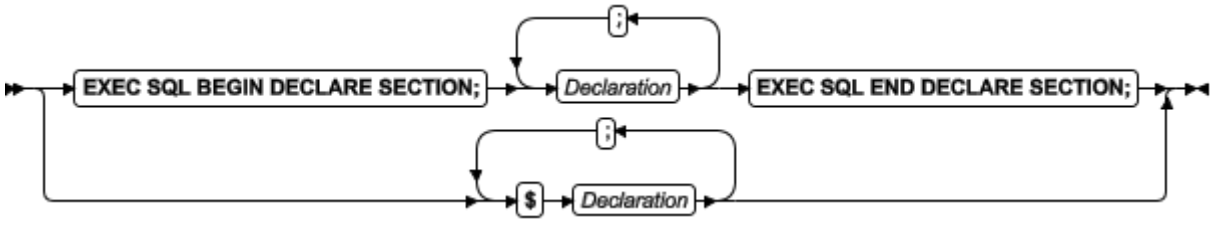
本节描述了有关 Windows™ 环境特有的 GBase 8s ESQL/C 主机变量：

如何使用非 ANSI 存储类修饰符声明主变量

如何声明全局 GBase 8s ESQL/C 变量

使用非 ANSI 存储类修饰符声明主变量

ANSI C 标准为变量声明定义了一组存储类修饰符。Windows™ 环境中的 C 编译器通常支持非 ANSI 存储类修饰符。为了在 GBase 8s ESQL/C 主机变量声明中为这些非 ANSI 存储类说明符提供支持，GBase 8s ESQL/C 预处理器支持 ANSI 语法的形式，如图所示。



Declaration

元素	意义	限制	语法
modifier name	要传递给 C 编译进行翻译的的文本。 该文本通常是存储类修饰符的名称。	该修饰符必须对 C 编译器有效或者是您的程序中定义的名称。	请参阅 C 编译器文档。
variable name	ESQL/C 主机变量的标识符名称	无	请参阅声明主机变量。
variable type	ESQL/C 主机变量的数据类型	该类型必须是有效的 C 或 ESQL/C 数据类型。	请参阅声明主机变量。

例如：Microsoft™ Visual C++ 编译器支持 **declspec** 编译器伪指令以使您能能声明扩展的存储类属性。该编译器伪指令具有以下语法：

```
__declspec(attribute) var_type var_name;
```

在此示例中，*attribute* 是受支持的关键字（例如 **thread** 、**dllimport** 或 **dllexport**），*var_type* 是变量的数据类型，*var_name* 是变量名称。

为了使您能够将 GBase 8s ESQL/C 主机变量声明为扩展存储类变量，GBase 8s ESQL/C 预处理器支持具有以下语法的 **declspec** 伪指令：

```
@("__declspec(attribute)") var_type var_name;
```

在此示例中，*attribute* 、*var_type* 和 *var_name* 与上一个示例的中相同。您可能会发现为 **declspec** 语法声明宏定义很方便。以下示例将 **threadCount** 声明为线程扩展存储类的实例特定整数变量：

```
#define DLLTHREAD __declspec(thread)
        ⋮

    EXEC SQL BEGIN DECLARE SECTION;
        @("DLLTHREAD") int threadCount;
    EXEC SQL END DECLARE SECTION;
```

本示例创建了 DLLTHREAD 宏以简化线程扩展存储类属性的声明。您可以声明类似的宏来简化要导出（导入）动态链接库（DLL）的变量的声明，如下所示：

```
#define DLLEXPORT __declspec(dllexport);
        ⋮

EXEC SQL BEGIN DECLARE SECTION;
        @("DLLEXPORT") int winHdl;
EXEC SQL END DECLARE SECTION;
```

指示变量

当 SQL 语句返回值时，它将返回指定列的主机变量。在某些情况下，可以将指示变

量与主机变量相关联，以获取有关返回值的其它信息。如果指定指示变量，GBase 8s ESQL/C 将其返回给主机变量。

指示变量为以下情况提供了附加信息：

- 如果主机变量与数据库列相关联或者与允许空值的聚集函数相关联，则指示变量可以指定该值是否为空。
- 如果主机变量是字符数组并且列值被传输截断，那么指示变量可以指定返回值的大小。

以下主题描述如何声明指示变量并将它与主机变量关联，以及 GBase 8s ESQL/C 如何设置指示变量以指定前述的两个条件。

声明指示变量

可以与主机变量相同的方式声明指示变量，如下所示，在 BEGIN DECLARE SECTION 和 END DECLARE SECTION 语句之间：

```
EXEC SQL BEGIN DECLARE SECTION;
        -- put indicator variable declarations here
EXEC SQL END DECLARE SECTION;
```

指示变量可以是除 DATETIME 或 INTERVAL 外的任何数据类型。通常，将指示变量声明为整数。例如：假如您的程序声明了名为 **name** 的主机变量。可以声明名为 **nameind** 的 **short** 整型指示变量 ，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
    char    name[16];
    short nameind;
    EXEC SQL END DECLARE SECTION;
```

如果客户机语言环境支持非 ASCII 字符，那么您可以在 GBase 8s ESQL/C 指示变量名称中使用非 ASCII（非英语）字符。

将指示变量与主机变量相关联

可以使用以下两种方式之一将指示变量与其主变量相关联：

在指示变量之间加上冒号（:）前缀，将关键字 INDICATOR 放置在主变量名和指示变量名之间，如下所示：

```
:hostvar INDICATOR :indvar
```

在主变量名和指示变量之间放置一个分隔符号。下列分隔符号有效：

冒号（:）

```
:hostvar:indvar
```

美元符号（$）

```
$hostvar$indvar
```

可以使用美元符号（$） 代替冒号（:） ，但是冒号使代码更容易阅读。

主变量和指示变量之间可以有一个或多个空白字符。例如：以下两种格式都可以在

hostvar 主机变量上指定指示变量 **hostvarind**：

```
$hostvar:hostvarind
$hostvar :hostvarind
```

指示空值

当 GBase 8s ESQL/C 语句对主机变量返回空值时，该值可能不是有意义的 C 值。您的程序可以采取以下操作之一：

- 如果您已经为此主变量定义了指示变量，那么 GBase 8s ESQL/C 将此指示变量设置为 -1。

  您的程序可以检查到此指示变量的值为 -1。

- 如果未定义指示变量，那么 GBase 8s ESQL/C 运行时的行为取决于编译程序的方式：

  - 如果使用 **-icheck** 预处理器选项编译查程序，那么当数据库服务器返回空值时 GBase 8s ESQL/C 生成错误并将 **sqlca.sqlcode** 设置为负数。
  - 如果没有使用 **-icheck** 选项编译程序，那么当数据库服务器返回空值时 GBase 8s ESQL/C 不会产生错误。在这种情况下，可以使用 **risnull()** 函数测试空值的主变量。

如果返回主变量的值不为空，那么 GBase 8s ESQL/C 将指示变量设置为 0 。如果 SQL 操作失败，那么指示变量的值无意义。因此，请在检查主机变量的空值之前检查 SQL 语句的输出结果。

INSERT 语句的 NULL 关键字允许您向表的行插入空值。作为 INSERT 语句的替代方法，可以使用带有主机变量的负指示变量。

如果您想要在指示符设置为空（-1）时插入变量，那么指示值优先于变量值。在这种情况下，插入的值将为 NULL，而不是主机变量的值。

当您将聚集函数的值的返回到主机变量中时，请记住，当数据库服务器在空表上执行聚集函数时，聚集操作的结果为空值。此规则的唯一例外是 **COUNT(*)** 聚集函数，它在此情况下返回 0 。

重要： 如果您激活数据库服务器的 DATASKIP 功能，那么如果所有分片离线或者所有联机的分片是空时，聚集函数也返回空。

指示截断值

当 SQL 语句将空值返回到主机变量字符数组中时，它可能截断该值以适应该变量。如果您为此主机变量定义了指示变量，那么 GBase 8s ESQL/C：

将 SQLSTATE 变量设置为 "01004" 以表示截断的发生。

GBase 8s ESQL/C 还将 **sqlca.sqlwarn** 结构的 **sqlwarn1** 设置为 W。

将关联的指示变量设置为等于截断前 SQL 主机变量的大小（以字节为单位）。

如果未定义指示变量，那么 GBase 8s ESQL/C 仍会设置 SQLSTATE 和 **sqlca.sqlwarn** 表示截断。但是，您的程序无法确定多少数据被截断。

如果数据库服务器返回未截断的值或空值，那么 GBase 8s ESQL/C 将指示变量设置为 0 。

使用指示变量的示例

图 1 和 图 2中的代码段显示了如何指示变量和主机变量的示例。这两个示例都使用指示变量执行以下任务：

确定字符数组中是否发生截断

如果在 **customer** 表中定义长度超过 15 个字符的 **lname**，那么 **nameind** 包含 **lname** 列的实际长度。**name** 主机变量包含 **lname** 值的前 15 个字符。（字符串 **name** 必须以空字符结尾。）如果具有 **customer_num = 105** 代表的公司的姓氏短于 15 个字符，那么 GBase 8s ESQL/C 仅截断尾随的空格。

检查空值

如果 **company** 对同一客户具有空值，那么 **compind** 具有一个负值。不能预测该字符数组 **comp** 的内容。

0下图显示了 SQL 语句使用 EXEC SQL 语法的 GBase 8s ESQL/C 程序。

图: 使用带有 EXEC SQL 和冒号（:）的指示变量

```
EXEC SQL BEGIN DECLARE SECTION;
        char    name[16];
        char    comp[20];
        short   nameind;
        short   compind;
EXEC SQL END DECLARE SECTION;
        ⋮

EXEC SQL select lname, company
        into :name INDICATOR :nameind, :comp INDICATOR :compind
        from customer
        where customer_num = 105;
```

图 1使用 INDICATOR 关键字将主变量和指示变量关联。此方法使用 ANSI 标准编译。

下图显示了使用美元符号（$）格式的 SQL 语句的 GBase 8s ESQL/C 程序示例。

图: 使用带美元符号（$）的指示变量

```
$char    name[16];
$char    comp[20];
$short   nameind;
$short   compind;
```

```
        ⋮
$select lname, company
into $name$nameind, $comp$compind
from customer
where customer_num = 105;
```

2.1.4 ESQL/C 头文件

当安装 GBase 8s ESQL/C 时，安装脚本将头文件存储在 $GBASEDBTDIR/incl/esql 目录（在 UNIX™ 操作系统上）和 %GBASEDBTDIR%\incl\esql 目录中（在 Windows™ 环境中）。

下表显示了 GBase 8s ESQL/C 产品提供的头文件。

表 1. ESQL/C 头文件

头文件	内容	其它信息
datetime.h	GBase 8s ESQL/C **datetime** 和 **interval** 结构的定义，它们是 DATETIME 和 INTERVAL 列的主变量	时间数据类型
decimal.h	GBase 8s ESQL/C **decimal** 数据类型的定义，是 DECIMAL 和 MONEY 数据类型的主变量	Numeric 数据类型
gls.h	用于 GLS 功能的函数原型和数据结构	*GBase 8s GLS 用户指南*
ifxtypes.h	精确映射 GBase 8s 数据类型 **int1** 、**int2** 、**int4** 、**mint** 、**mlong** 、**MSHORT** 和 **MCHAR** ，用于 32 位和 64 位平台	整数主机变量类型
locator.h	GBase 8s ESQL/C 定位器结构（**ifx_loc_t** 或 **loc_t**）的定义，它是字节和文本列的主变量	简单大对象
sqlca.h	ESQL/C 用来存储错误状态代码的结构的定义 当 esql 预处理器预处理程序时，自动包含此文件。	异常处理
sqlda.h	值指针和结构定义以及动态定义变量的描述	确定 SQL 语句
sqlhdr.h	该文件包含 sqlda.h 头文件、其它头文件以及函数原型。 预处理器预处理程序时自动包含该文件。	在程序中包含头文件

sqliapi.h	内部库 API 函数原型 仅供内部 GBase 8s ESQL/C 使用。	无
sqlstype.h	SQL 语句常量的定义 DESCRIBE 语句使用这些常量描述动态准备的 SQL 语句	确定 SQL 语句
sqltypes.h	定义与 ESQL/C 和 SQL 数据类型相对应的常量 当程序包含 DESCRIBE 语句时，ESQL/C 使用这些常量。	数据类型常量
sqlxtype.h	当在 X/Open 模式时定义与 GBase 8s ESQL/C 和 SQL 数据类型相对应的常量 当程序包含 DESCRIBE 语句时，ESQL/C 使用这些常量。	X/Open 数据类型常量
value.h	GBase 8s ESQL/C 使用的值结构 仅供内部 GBase 8s ESQL/C 使用。	无
varchar.h	可以使用 VARCHAR 数据类型的宏	字符和字符串数据类型

下图显示了特定于 GBase 8s 的 GBase 8s ESQL/C 头文件。

表 2. GBase 8s 的 ESQL/C 头文件

头文件	内容	其它信息
collct.h	GBase 8s ESQL/C 中复杂类型的数据结构的定义	复杂数据类型
ifxgls.h	用于 GLS 应用程序编程接口的函数原型 仅供内部 GBase 8s ESQL/C 使用。	无
int8.h	存储 INT8 数据类型结构的定义	int8 数据类型

下表显示了特定于 Windows 环境的 GBase 8s ESQL/C 头文件。

表 3. Windows 环境中的 ESQL/C 头文件

头文件	内容	其它信息

sqlproto.h	所有 ESQL/C 库函数的函数原型，用于不完全符合 ANSI C 标准的源	声明函数原型
infxcexp.c	包含用于导出 ESQL 客户端接口 DLL 使用的所有 C 运行时例程的地址的 C 代码	相同运行例程的版本独立性
login.h	**InetLogin** 和 **HostInfoStruct** 结构的定义，为应用程序定制配置参数 因为此文件不包含 ESQL 语句，所以不需要使用 ESQL **include** 伪指令包含它。而是用 C **#include** 预处理器伪指令。	InetLogin 结构的字段

声明函数原型

GBase 8s ESQL/C 提供 sqlproto.h 头文件为所有 GBase 8s ESQL/C 库函数声明函数原型。您使用 ANSI C 编译器的 GBase 8s ESQL/C 源文件需要这些函数原。缺省情况下，esql 命令处理器不包含函数原型声明。使处理器包含符合 GBase 8s ESQL/C 函数的 ANSI 兼容的函数原型可以防止 ANSI C 编译器生成警告。

限制： 尽管可以使用 ANSI C 编译器，GBase 8s ESQL/C 预处理器并不全支持 ANSI C，因此您可能不能处理遵循 ANSI C 标准的全部程序。

因为 sqlproto.h 文件不包含任何 GBase 8s ESQL/C 语句，使用可用使用以下方法包含此文件：

使用 GBase 8s ESQL/C **include** 预处理器伪指令：

```
EXEC SQL include sqlproto;
```

使用 C **#include** 预处理器伪指令：

```
#include "sqlproto.h";
```

在程序中包含头文件

GBase 8s ESQL/C 预处理器在程序中自动包含以下 GBase 8s ESQL/C 头文件：

sqlhdr.h 文件为您的 GBase 8s ESQL/C 程序提供和游标相关的结构。

该头文件自动包含 sqlda.h 和 ifxtypes.h 头文件。

sqlca.h 文件，允许您的查询检查带有 SQLSTATE 或 SQLCODE 变量的 GBase 8s ESQL/C 语句的成功或失败。

限制： 尽管现在使用 ANSI C 编译器，GBase 8s ESQL/C 预处理器也不完全支持 ANSI C，因此可能不能处理所有遵循 ANSI C 标准的程序。

要在您的 GBase 8s ESQL/C 程序中包含任何其它头文件，必须使用 **include** 预处理器伪指令。但是，如果您的程序引用头文件定义的结构或定义，则必须包含 GBase 8s ESQL/C 头文件。例如：如果您的程序访问 datetime 数据，那么必须包含 datetime.h 头文

件，如下所示：

```
EXEC SQL include datetime.h;
```

确保使用分号结束代码行。一些其它示例如下：

```
EXEC SQL include varchar.h;
EXEC SQL include sqlda;
        $include sqlstype;
```

提示： 不需要对 GBase 8s ESQL/C 头文件输入 .h 文件扩展名；esql 预处理器假定 .h 扩展。

2.1.5 ESQL/C 预处理器伪指令

当编写 GBase 8s ESQL/C 代码时可以使用下列 GBase 8s ESQL/C 预处理器的功能：

include 伪指令展开程序中的 GBase 8s ESQL/C include 文件。

define 和 **undef** 伪指令创建编译时的定义。

ifdef 、**ifndef** 、**else** 、**elif** 和 **endif** 伪指令指定条件编译。

与嵌入式 SQL 语句一样，您可以使用以下 GBase 8s ESQL/C 预处理器指令的两种格式之一 ：

EXEC SQL 关键字：

```
EXEC SQL preprocessor_directive;
```

EXEC SQL 关键字符号 ANSI 标准。

美元符号（$）：

```
$preprocessor_directive;
```

在任一格式中，使用下列章节描述的有效的预处理器伪指令替换 *preprocessor_directive* 。必须用分号（;）终止这些指令。

GBase 8s ESQL/C 预处理器分两个阶段工作。在第 1 阶段，它作为 GBase 8s ESQL/C 代码的预处理器。在第 2 阶段，它将所有嵌入式 SQL 代码转换为 C 代码。

在第 1 阶段，GBase 8s ESQL/C 预处理查询通过处理所有 **include** 伪指令（**$include** 和 EXEC SQL **include** 语句）在源文件中并入其它文件。同样在第 1 阶段，GBase 8s ESQL/C 通过处理 **define**（**$define** 和 EXEC SQL **define**）和 **undef**（**$undef** 和 EXEC SQL **undef**）伪指令创建或移除编译时的定义。

本节的其余部分将详细介绍 GBase 8s ESQL/C 预处理程序伪指令。

include 指令

include 伪指令运行您指定在 GBase 8s ESQL/C 程序中包含的文件。

GBase 8s ESQL/C 预处理器程序将指定文件的内容放到 GBase 8s ESQL/C 源文件中。GBase 8s ESQL/C 预处理程序的第一阶段将 *filename* 的内容读入 **include** 指令位置的当前文件中。

可以使用以下两种格式之一的 **include** 预处理器指令：

EXEC SQL include filename;

$include filename;

将 *filename* 替换为要包含在程序中的文件的名称。您可以指定带有或不带引号的文件名。但是，如果使用完整路径名称，则必须将路径名括在引号中。

以下示例显示了如何在 Windows™ 环境中使用完整路径名。

```
EXEC SQL include decimal.h;

        EXEC SQL include "C:\apps\finances\credits.h";
```

提示： 如果指定完整路径名，当文件位置变化后您必须重新编译此程序。更好的编程实践使用 **esql -I** 选项指定搜索位置，并使用 include 指定文件名。

如果忽略了文件名周围的引号，GBase 8s ESQL/C 将文件名更改为小写字母。如果省略了路径名，GBase 8s ESQL/C 预处理程序检查该文件的预处理程序的搜索路径。

可以对以下类型文件使用 **include** ：

GBase 8s ESQL/C 头文件

不必对 GBase 8s ESQL/C 头文件使用 .h 文件扩展名；编译器假定您的程序引用了扩展名为 .h 的文件。以下示例显示了包含 GBase 8s ESQL/C 头文件的有效语句：

```
EXEC SQL include varchar.h;
        $include sqlda;
        EXEC SQL include sqlstype;
```

其它用户定义的文件

必须指定您想要包含文件的确切名称。当您包含的不是 GBase 8s ESQL/C 头文件的头文件时，编译器不会假定 .h 扩展名。

以下示例显示在 UNIX 环境下包含 constant_defs 和 typedefs.h 文件的有效语句：

```
EXEC SQL include constant_defs;
EXEC SQL include "constant_defs";
 $include typedefs.h;
EXEC SQL include "typedefs.h";
```

如果指定的文件包含嵌入式 SQL 语句或者其它 GBase 8s ESQL/C 语句，您必须使用 GBase 8s ESQL/C **include** 伪指令。

使用标准 C **#include** 伪指令包含系统头文件。C 的 **#include** 在 GBase 8s ESQL/C 预处理后包含一个文件。

限制： 在声明部分不支持嵌入式 INCLUDE 语句，并可能产生误导性错误。

define 和 undef 指令

GBase 8s ESQL/C 预处理程序允许您创建只对 GBase 8s ESQL/C 预处理程序可用的简单变量。 GBase 8s 调用这些变量*定义*。GBase 8s ESQL/C 预处理程序使用两个指令管

理这些定义：

define

创建名称标志定义。该定义的范围是从您将其定义到 GBase 8s ESQL/C 源文件的末尾。

undef

移除 EXEC SQL **define** 或 **$define** 创建的名称标志。

GBase 8s ESQL/C 预处理程序而不是 C 预处理查询（处理 **#define** 和 **#undef**）处理这些指令。 GBase 8s ESQL/C 预处理程序创建（**define**）或移除（**undef**）预处理阶段 1 中的这些定义。

GBase 8s ESQL/C **define** 指令创建具有以下格式的定义：

Boolean 符号的格式是 define symbolname;

以下示例显示如何定义 Boolean 符号 TRANS 的两种方法：

```
EXEC SQL define TRANS;
        $define TRANS;
```

整数常量的格式是 define symbolname value;

以下示例显示了两个整数常量的格式，MAXROWS （具有值 25 ）和 USETRANSACTIONS （具有值 1）：

```
EXEC SQL define MAXROWS 25;
            $define MAXROWS 25;

            EXEC SQL define USETRANSACTIONS 1;
            $define USETRANSACTIONS 1;
```

重要： 不同于 C **#define** 定义，**define** 指令不支持字符串常量或接收运行时的值的语句的宏。

可以使用 esql 命令行选项 **-ED** 和 **-EU** 覆盖源程序中的 **define** 和 **undef** 语句。

ifdef 、ifndef 、elif 、else 和 endif 指令

GBase 8s ESQL/C 预处理程序支持条件编译的以下指令：

ifdef

如果定义已创建名称，则测试名称并执行后续的语句。

ifndef

如果定义未创建名称，则测试名称并执行后续的语句。

elif

开始 **ifdef** 或 **ifndef** 条件的替代部分并检查另一个 **define** 的存在。它是 “else if define ” 的缩写。

else

开始 **ifdef** 或 **ifndef** 条件的替代部分。

endif

结束 **ifdef** 或 **ifndef** 条件。

在以下示例中，如果之前使用 **define** 指令定义 USETRANSACTIONS 名称，则进行编译 BEGIN WORK 语句：

```
EXEC SQL ifdef USETRANSACTIONS;
        EXEC SQL begin work;
        EXEC SQL endif;
```

下列示例说明 **elif** 语句的使用。该样本将会输出 “USETRANSACTIONS defined”。

```
EXEC SQL define USETRANSACTIONS;

        ⋮

        EXEC SQL ifndef USETRANSACTIONS;
        printf(“USETRANSACTIONS not defined”);
        EXEC SQL elif USETRANSACTIONS;
        printf(“USETRANSACTIONS defined”);
        EXEC SQL endif;
```

GBase 8s ESQL/C 预处理查询不支持一般的 **if** 指令，它仅支持 **ifdef** 和 **ifndef** 语句来测试名称是否使用 **define** 定义。

GBase 8s ESQL/C 预处理程序处理预处理的阶段 1 中的条件编译定义。

2.1.6 在 Windows™ 环境中设置和检索环境变量

可以更改环境变量的设置或创建新的变量来增加应用程序的灵活性。您可以在运行时定义环境，而不是假定特定的环境配置。

该选项可以通过以下方式使您的应用程序受益：

应用程序变得较少以来预定义的环境。

用户可以在应用程序中输入他们的用户名和密码。

用户可以在同一计算机上运行的不同网络参数的两个应用程序。

相同的应用程序可以使用不同配置在客户机上运行。

以下 GBase 8s ESQL/C 库函数可用于设置和检索环境变量。

ifx_putenv()

修改或移除现有的环境变量或创建变量。

ifx_getenv()

检索环境变量的值。

重要： ifx_putenv() 函数设置 **InetLogin** 结构中的环境变量的值，ifx_getenv() 函数检索来自 **InetLogin** 环境变量的值。建议您使用这些函数设置和检索 **InetLogin** 字段值。

这些函数仅影响当前进程的本地变量。ifx_putenv() 函数不会更改命令级别环境。这些函数操作仅对 GBase 8s ESQL/C 运行时库可访问的数据结构进行操作，而不是操作系统为进程创建的环境段。当当前进程终止时，环境恢复到调用进程的级别（在大多数情况下，操作系统级别）。

该过程不能直接将修改的环境传递给 **_spawn()** 、**_exec()** 或 **system()** 创建的任何新进程。这些新进程不接收 ifx_putenv() 添加的任何新的变量。然而，可以使用以下方法将环境变量传递给新的进程。

当前进程使用 GBase 8s ESQL/C ifx_putenv() 函数创建环境变量。

当前进程使用 C putenv() 函数将环境变量放入操作系统环境中。

当前进程开始新的进程。

新进程使用 C **getenv()** 函数检索来自系统环境变量段的环境变量。

新进程使用 GBase 8s ESQL/C ifx_getenv() 函数将变量检索到运行时的环境段中。

环境变量指南

对于环境变量™，请遵循以下守则：

如果计划使用 ifx_putenv() 设置任何 GBase 8s 环境变量，则在调用任何其他 GBase 8s ESQL/C 库例程（包括 ifx_getenv()）或任何 SQL 语句之前，将应用程序设置为所有这些变量。首次调用其他 GBase 8s ESQL/C 库函数或 SQL 语句需要初始化 GLS 语言环境。此初始化加载并冻结 CLIENT_LOCALE 、DB_LOCALE 和 DATE 、TIME和 DATETIME 格式值的值。

如果 Setnet32 将注册表中 GBase 8s 环境变量设置为非空值，那么 ifx_putenv() 函数不会将此变量的在值更改为空字符串。

如果为 ifx_putenv() 函数调用中的环境变量指定一个空字符串，GBase 8s ESQL/C 将从运行时环境段清除环境变量的任何值集。然后为此环境变量注册可用于应用程序的值。

不要在命名行中使用 setenv 或使用 C putenv() 函数更改环境变量，因为在应用程序执行开始后，对操作系统环境段的更改对 ESQL 客户端接口 DLL 没有影响。

而是使用 ifx_putenv() 更改运行时环境段中的环境变量。

要在不影响环境表的情况下更改 ifx_getenv() 的返回值，使用 _strdup() 或 strcpy() 创建此字符串的副本。

限制： 不要将指针释放到 ifx_getenv() 返回的环境条目中。另外，不要将 ifx_putenv() 指针指向局部变量。然后退出声明变量的函数。

InetLogin 结构

Windows™ 环境中的 GBase 8s ESQL/C 客户端应用程序可以使用 **InetLogin** 结构

动态设置应用程序所需的配置信息。

重要： GBase 8s 仅支持与早期版本兼容的 **InetLogin** 结构。对于新开发，建议您改用 ifx_getenv() 和 ifx_putenv() 函数。

InetLogin 结构的字段

InetLogin 结构是 login.h 头文件声明的全局 C 结构。

要在 GBase 8s ESQL/C 程序中使用此结果，必须在您的源文件（.ec）中包含 login.h。

提示： 因为 login.h 不包含 GBase 8s ESQL/C 语句，可以使用 C **#include** 或 GBase 8s ESQL/C **include** 指令包含此文件。

下表在 **InetLogin** 结构中定义此字段。

表 1. InetLogin 结构的字段

Inetlogin 字段	数据类型	意义
InfxServer	char[19]	指定 GBASEDBTSERVER 环境变量的值（缺省数据库服务器）
DbPath	char[129]	指定 DBPATH 环境变量的值
DbDate	char[6]	指定 DBDATE 环境变量的值
DbMoney	char[19]	指定 DBMONEY 环境变量的值
DbTime	char[81]	指定 DBTIME 环境变量的值
DbTemp	char[81]	指定 DBTEMP 环境变量的值
DbLang	char[19]	指定 DBLANG 环境变量的值
DbAnsiWarn	char[1]	指定 DBANSIWARN 环境变量的值
GBase 8s **Dir**	char[255]	指定 GBASEDBTDIR 环境变量的值
Client_Loc	char *	指定 CLIENT_LOCALE 环境变量的值
DB_Loc	char *	指定 DB_LOCALE 环境变量的值
CollChar	char[3]	指定 COLLCHAR 环境变量的值
Lang	char[81]	指定 LANG 环境变量的值 for the database locale

Lc_Collate	char[81]	指定 LC_COLLATE 环境变量的值 for the database locale
Lc_CType	char[81]	为数据库本地环境指定 LC_CTYPE 环境变量的值
Lc_Monetary	char[81]	为数据库本地环境指定 **LC_MONETARY** 环境变量的值
Lc_Numeric	char[81]	为数据库本地环境指定 LC_NUMERIC 环境变量的值
Lc_Time	char[81]	为数据库本地环境指定 LC_TIME 环境变量的值
ConRetry	char[4]	指定环境变量 GBASEDBTCONRETRY 的值
ConTime	char[4]	指定环境变量 GBASEDBTCONTIME 的值
DelimIdent	char[4]	指定环境变量 DELIMIDENT 的值
Host	char[19]	指定 HOST 网络参数的值
User	char[19]	指定 USER 网络参数的值
Pass	char[19]	指定 PASSWORD 网络参数的值
AskPassAtConnect	char[2]	指示 sqlauth() 在连接时是否请求密码；包含是或否的值。如果首字符是 Y 或 y 那么请设置 **AskPassAtConnect**
Service	char[19]	指定 SERVICE 网络参数的值
Protocol	char[19]	指定 PROTOCOL 网络参数的值
Options	char[20]	保留备用
GBase 8s **SqlHosts**	char[255]	指定 GBASEDBTSQLHOSTS 环境变量的值
FetBuffSize	char[6]	指定 FET_BUF_SIZE 环境变量的值
CC8BitLevel	char[2]	指定 CC8BITLEVEL 环境变量的值

EsqlMF	char[2]	指定 ESQLMF 环境变量的值
GlDate	char[129]	指定 GL_DATE 环境变量的值
GlDateTime	char[129]	指定 GL_DATETIME 环境变量的值
DbAlsBc	char[2]	指定 DBALSBC 环境变量的值
DbApiCode	char[24]	指定 DBAPICODE 环境变量的值
DbAsciiBc	char[2]	指定 DBASCIIBC 环境变量的值
DbCentury	char[2]	指定 DBCENTURY 环境变量的值
DbCodeset	char[24]	指定 DBCODESET 环境变量的值
DbConnect	char[2]	指定 DBCONNECT 环境变量的值
DbCsConv	char[9]	指定 DBCSCONV 环境变量的值
DbCsOverride	char[2]	指定 DBCSOVERRIDE 环境变量的值
DbCsWidth	char[12]	指定 DBCSWIDTH 环境变量的值
DbFltMsk	char[4]	指定 DBFLTMASK 环境变量的值
DbMoneyScale	char[6]	指定 **DBMONEYSCALE** 环境变量的值
DbSS2	char[5]	指定 DBSS2 环境变量的值
DbSS3	char[5]	指定 DBSS3 环境变量的值
OptoFC	char[2]	不使用
OptMSG	char[2]	不使用

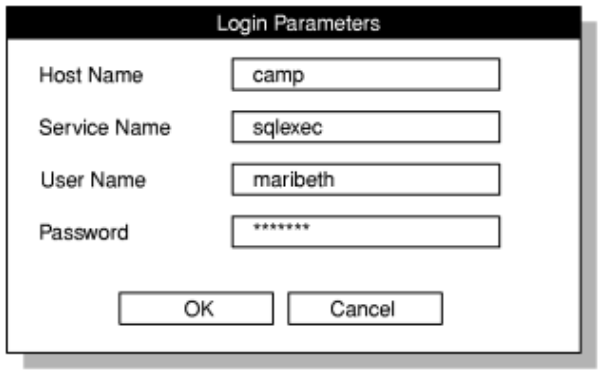
InetLogin 结构中的所有字段，除了 **DbAnsiWarn** 、**Client_Loc** 和 **DB_Loc** ，都是 **char** 数据类型，并且都是空终止字符串。**Client_Loc** 和 **DB_Loc** 字段是字符指针，GBase 8s ESQL/C 程序必须分配数据空间。

InetLogin 字段值

当应用程序执行 SQL 语句或需要配置信息的 GBase 8s ESQL/C 库函数之前必须设置 **InetLogin** 值。建议您使用 ifx_putenv() 和 ifx_getenv() 函数通过环境变量设置并检索 **InetLogin** 字段值，但是可以直接设置 **InetLogin** 字段值。

下图显示了客户端应用程序可能用于从最终用户获取网络参数的对话框。此应用程序将使用用户输入的账户信息，并在 **InetLogin** 结构中设置适当的网络值。

图: 用户输入登录参数的对话框



下图显示在 **InetLogin** 结构中设置登录值的代码段。应用程序通过对话框（同 图 1 中一样）从最终用户获取这些值。

图: 提示用户输入 InetLogin 值的代码

```
strcpy(InetLogin.InfxServer, "mainsrvr");

                ⋮

                case IDOK:
                *szDlgString = '\0';
                GetDlgItemText (hdlg, IDC_HOST, szDlgString, cbSzDlgMax);
                strcpy(InetLogin.Host, szDlgString);

                *szDlgString = '\0';
                GetDlgItemText (hdlg, IDC_USER, szDlgString, cbSzDlgMax);
                strcpy(InetLogin.User, szDlgString);
```

在上图中，如果用户输入主机信息，则该代码将 **mainsrvr** 数据库服务器的 **InetLogin.Host** 和 **InetLogin.User** 字段设置用户指定的主机名和用户名的名称。如果没有输入主机信息，那么 GBase 8s ESQL/C 使用 **mainsrvr** 数据库服务器的子项中的 HOST 和 USER 注册表值。

提示： 如何设置 InetLogin 字段的另一示例，请参阅 %GBASEDBTDIR%\demo\ilogin 目录下的 ILOGIN 演示程序。

配置值的优先级

当 Windows™ 环境中的客户端应用程序需要配置参数时，GBase 8s ESQL/C 从以下位置获得这些参数：

InetLogin 结构

如果应用程序使用 **InetLogin** 结构，GBase 8s ESQL/C 首先检查此结构中的配置信息。（要为此应用程序设置环境变量的值，ifx_putenv() 函数更改 **InetLogin** 字段的值。）

注册表的 GBASEDBT 子项

如果应用程序尚未在 **InetLogin** 中设置所需的配置信息，那么 GBase 8s ESQL/C 会在其注册表信息的副本中检查此选项。

不必在 **InetLogin** 结构中定义所有值。应用程序使用在注册表中使用在 **InetLogin** 中找不到的任何值的配置信息。如果未设置相应的注册表值，应用程序将使用其缺省值。

重要： 应用程序首次需要配置信息时，GBase 8s ESQL/C 从注册表中读取此信息，并将其存储在内存中。有关后续对注册表信息的引用，GBase 8s ESQL/C 访问此内存中的副本，并且不会重新读取注册表。

如果您希望应用程序在运行时用户提供用户名和密码，或者如果应用程序具有与注册表中一般值不同的配置信息，则配置信息的这种层级结构很有价值。例如，假设应用程序将 **InetLogin** 的 **ConRetry** 字段设置为 2 ，但是没有设置 **ConTime** 字段，如下列代码段所示：

```
strcpy(InetLogin.ConRetry, "2");
EXEC SQL connect to 'accnts';
```

当 GBase 8s ESQL/C 建立到 **accnts** 数据库的连接时，它会尝试建立两次（而不是缺省值一次），但是它仍然使用 15 秒的连接时间（内存中副本的默认值注册表信息）。如果 Setnet32 已更改连接值，那么GBase 8s ESQL/C 使用以更改的注册表值而非缺省值。

提示： 使用 Setnet32 实用程序定义注册表中的配置信息。

2.1.7 Windows 环境中的全局 ESQL/C 变量

GBase 8s ESQL/C 提供支持不同功能的全局变量。下表描述了这些全局变量。

表 1. 全局 ESQL/C 变量

Global 变量	描述
SQLSTATE	符合 ANSI 标准的状态代码，具有五个字母字符串（加上空终止符）
SQLCODE **sqlca.sqlcode**	特定于 GBase 8s 的状态代码，为整数值
sqlca 结构	特定于 GBase 8s 的诊断信息
FetBufSize 和 **BigFetBufSize**	获取缓冲区的大小 **BigFetBufSize** 与 **FetBufSize** 相同，但是游标缓冲区上限值较高
InetLogin 结构	用户客户端 ESQL/C 应用程序的环境信息

在此环境中，GBase 8s ESQL/C 将 表 1 中的全局变量作为 sqlhdr.h 文件定义的函数实现。这些函数返回与全局变量对应的数据类型相同的值。因此，这种实现方式的改变不需要修改现有的 GBase 8s ESQL/C 代码。您仍然可以在与全局变量相同的上下文中使用这些函数。

2.1.8 GBase 8s ESQL/C 程序的样本

demo1.ec 程序演示了本节呈现的大多数概念，例如：include 文件、标识符、主机变量以及嵌入式 SQL 语句。它演示如何使用头文件、声明以及使用主机变量和嵌入式 SQL 语句。

重要： 如果使用 UNIX™，可以在 $GBASEDBTDIR/demo/esqlc 目录下找到本版的联机以及其它演示程序。如果使用 Windows™，可以在 %GBASEDBTDIR%\demo\esqldemo 目录下找到此程序。

编译 demo1 程序

使用下列命令编译 **demo1** 程序：esql demo1.ec

在 UNIX™ 中，可执行程序的名称缺省为 **a.out**。

在 Windows™ 环境中，可执行程序的名称缺省为 **demo.exe**。

可以使用 **-o** 选项对可执行程序指定不同的名称。

demo1.ec 文件指南

GBase 8s ESQL/C 程序样本 demo1.ec 使用静态 SELECT 语句。这意味着在编译时程序可以获得它需要运行 SELECT 语句的所有信息。

```
1. #include <stdio.h>
2. EXEC SQL define FNAME_LEN 15;
3. EXEC SQL define LNAME_LEN 15;
4. main()
5. {
6. EXEC SQL BEGIN DECLARE SECTION;
7. char fname[ FNAME_LEN + 1 ];
8. char lname[ LNAME_LEN + 1 ];
9. EXEC SQL END DECLARE SECTION;
```

第 1 行

#include 语句告诉 C 预处理程序包含来自 /usr/include 目录下的 stdio.h 系统头文。stdio.h 文件使 **demo1** 使用标准 C 语言 I/O 库。

第 2 - 3 行

GBase 8s ESQL/C 处理预处理阶段 1 中的 **define** 指令。该指令定义定义 FNAME_LEN 和 LNAME_LEN 常量，程序在主机变量的声明中使用它们。

第 4 - 9 行

第 4 行开始 main() 函数，该程序的入口点。EXEC SQL 块声明 main() 函数本地的主机变量，它从 **customer** 表的 **fname** 和 **lname** 列接收数据。每个数组的长度大于其接收

数据的字符列的长度的 1 个字符。额外的字节存储空终止符。

```
10. printf( "DEMO1 Sample ESQL Program running.\n\n");
11. EXEC SQL WHENEVER ERROR STOP;
12. EXEC SQL connect to 'stores7';
13. EXEC SQL DECLARE democursor cursor for
14. select fname, lname
15. into :fname, :lname
16. from customer
17. where lname < 'C';
18. EXEC SQL open democursor;
```

第 10 - 12 行

printf() 函数显示用于标识的文本，并在程序开始执行时通知用户。WHENEVER 语句实现最少的错误处理，导致程序显示错误编号，并在数据库服务器处理 SQL 语句后返回错误时终止。CONNECT 语句发起与缺省数据库服务器的连接，并打开 **stores7** 演示数据库。您可以在 GBASEDBTSERVER 环境变量中指定缺省数据库服务器，必须在应用程序连接任何数据库服务器之前设置。

第 13 - 17 行

DECLARE 语句创建名为 **democursor** 的游标，来管理数据库服务器从 **customer** 表读取的行。DECLARE 语句中的 SELECT 语句确定数据库服务器从该表读取的数据的类型。此 SELECT 语句读取姓氏（**lname**）以小于 ’C’ 的字母开头的那些客户的第一个和家族姓名。

第 18 行

OPEN 语句打开 **democursor** 游标并开始执行 SELECT 语句。

```
19. for (;;)
20. {
21. EXEC SQL fetch democursor;
22. if (strncmp(SQLSTATE, "00", 2) != 0)
23. break;
24. printf("%s %s\n",fname, lname);
25. }
26. if (strncmp(SQLSTATE, "02", 2) != 0)
27. printf("SQLSTATE after fetch is %s\n", SQLSTATE);
28. EXEC SQL close democursor;
29. EXEC SQL free democursor;
```

第 19 - 25 行

本节代码在循环中执行 FETCH 语句并重复，直到 SQLSTATE 不等于 "00"。该条件表示数据结束条件或发生运行错误。在此循环中每个迭代中， FETCH 语句使用游标 **democursor** 检索 SELECT 语句返回的下一行并将选择的行放到主机变量 **fname** 和 **lname**中。数据库服务器每次成功获取行后都将 SQLSTATE 的状态变量设置为 "00" 。如果数据结束的条件发生，则数据库服务器将 SQLSTATE 设置为 "02"；如果发生错误，

则将 SQLSTATE 设置为比 "02" 大的值。

第 26 - 27 行

如果 SQLSTATE 中的类代码是除了 "02" 的任何值，则此用户的 SQLSTATE 值通过 printf() 显示。该输出在运行错误的事件中很有用。

第 28 - 29 行

CLOSE 和 FREE 语句释放为数据库服务器此游标分配的资源。该游标将不再使用。

```
30. EXEC SQL disconnect current;
31. printf("\nDEMO1 Sample Program over.\n\n");
32. }
```

第 30 - 32 行

DISCONNECT CURRENT 语句关闭数据库并终止当前对数据库服务器的连接。最后的 printf() 告诉用户程序结束。32 行的右括号（}）标识程序 main() 函数的结束。

OPEN 语句打开 **democursor** 游标并开始执行 SELECT 语句。

```
19. for (;;)
20. {
21. EXEC SQL fetch democursor;
22. if (strncmp(SQLSTATE, "00", 2) != 0)
23. break;
24. printf("%s %s\n",fname, lname);
25. }
26. if (strncmp(SQLSTATE, "02", 2) != 0)
27. printf("SQLSTATE after fetch is %s\n", SQLSTATE);
28. EXEC SQL close democursor;
29. EXEC SQL free democursor;
```

第 19 - 25 行

本节代码在循环中执行 FETCH 语句并重复，直到 SQLSTATE 不等于 "00"。该条件表示数据结束条件或发生运行错误。在此循环中每个迭代中， FETCH 语句使用游标 **democursor** 检索 SELECT 语句返回的下一行并将选择的行放到主机变量 **fname** 和 **lname**中。数据库服务器每次成功获取行后都将 SQLSTATE 的状态变量设置为 "00" 。如果数据结束的条件发生，则数据库服务器将 SQLSTATE 设置为 "02"；如果发生错误，则将 SQLSTATE 设置为比 "02" 大的值。

第 26 - 27 行

如果 SQLSTATE 中的类代码是除了 "02" 的任何值，则此用户的 SQLSTATE 值通过 printf() 显示。该输出在运行错误的事件中很有用。

第 28 - 29 行

CLOSE 和 FREE 语句释放为数据库服务器此游标分配的资源。该游标将不再使用。

```
30. EXEC SQL disconnect current;
31. printf("\nDEMO1 Sample Program over.\n\n");
32. }
```

第 30 - 32 行

DISCONNECT CURRENT 语句关闭数据库并终止当前对数据库服务器的连接。最后的 printf() 告诉用户程序结束。32 行的右括号（}）标识程序 main() 函数的结束。

2.2 程序指令

2.2.1 编译 ESQL/C 程序

esql 将 GBase 8s ESQL/C 源文件传递给 GBase 8s ESQL/C 预处理器和 C 编译器。它通过 GBase 8s ESQL/C 预处理器和 C 编译器特有的选项来预处理、编译和链接 GBase 8s ESQL/C 程序。

ESQL/C 预处理器

要处理、编译链接包含 GBase 8s ESQL/C 语句的程序，必需通过 GBase 8s ESQL/C 预处理器传递它。可以使用 esql 命令 GBase 8s ESQL/C 源文件上的预处理器并创建可执行文件。esql 命令按照以下步骤执行转换：

在第一阶段，GBase 8s ESQL/C 预处理执行以下步骤：

当头文件处理所有 **include** 指令（**$include** 和 EXEC SQL **include** 语句）时，将头文件合并到源文件中

在处理所有 **define**（**$define** 和 EXEC SQL **define**|）和 **undef**（**$undef** 和 EXEC SQL **undef**）指令时创建或删除编译时定义

在第二阶段，GBase 8s ESQL/C 预处理器处理任何条件编译指令（ifdef 、ifndef 、else 、elif 、endif）并将嵌入式 SQL 语句转换为 GBase 8s ESQL/C 函数调用和特殊的 0 数据类型。

阶段 1 和阶段 2 反映了 C 编译器的预处理器和编译器阶段。成功完成预处理步骤将产生一个 C 源文件（.c 扩展名）。

esql 命令预处理器作为 GBase 8s ESQL/C 产品的一部分安装。在使用 esql 之前，请确保：

GBase 8s ESQL/C 源文件的文件名称具有 .ec 或 .ecp。

GBASEDBTDIR 和 PATH 环境变量已正确设置。

如果在命令窗口或 Windows™ 注册表中未设置 GBASEDBTDIR 环境变量，则将其内部设置为 GBase 8s Client SDK 动态链接库的位置。

如果在 UNIX™ 中未设置 GBASEDBTDIR 环境，则当编译任何 GBase 8s Client Software Development Kit (Client SDK) 应用程序时会发生错误。

重要：始终将 esql 程序与程序链接。 使用的库列表可以在版本之间更改。与 esql 链接确保程序与库正确链接。

预处理器生成的 C 代码可能随着产品的版本而更改。因此，不要设置程序，这些程序取决于 如何在 预处理器生成的 C 代码中实现产品的功能。

C 预处理器和编译器

esql 不会自动编译和链接 GBase 8s ESQL/C 程序。esql 命令将 GBase 8s ESQL/C 代码转换为 C 代码，然后调用 C 编译器编译并链接到 C 代码。C 预处理器预处理 C 语言预处理指令。C 编译器执行此编译，然后它还调用链接到 C 对象文件的链接编辑器。

GBase 8s ESQL/C 源文件包含 C 预处理器的命令（组成 *#directive* 的指令）。当您使用编译的缺省顺序时，这些 C 指令对 GBase 8s ESQL/C 语句没有影响，但是在 C 编译器处理源文件时以常规方式生效。

如果在 GBase 8s ESQL/C 预处理之前选择在 GBase 8s ESQL/C 源文件上运行 C 预处理器，可以使用 C 语言 **#define** 和 **typedef** 指令定义 GBase 8s ESQL/C 主变量。

C 编译器采取以下操作：

将 C 语言语句编译为对象代码

链接到 GBase 8s ESQL/C 库或您指定的库或其它文件

创建可执行文件

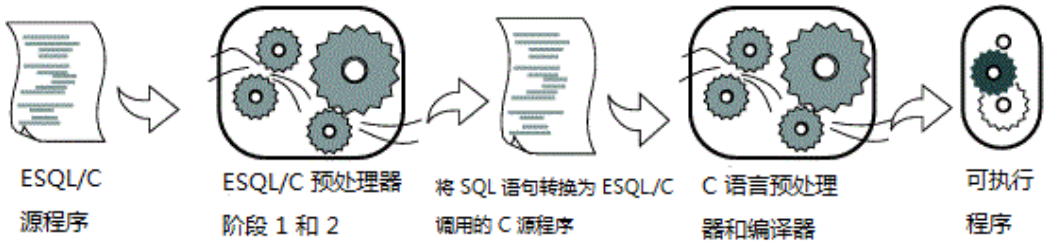
如果使用 C 编译器而不是通过将 GBASEDBTC 环境变量设置为非缺省值的本地 C 编译器，那么您可能需要覆盖此编译器的缺省选项。

缺省编译顺序

在您创建了 GBase 8s ESQL/C 程序文件之后，可以对此文件运行 esql 命令。缺省情况下，GBase 8s ESQL/C 预处理器首先运行，并将程序中的嵌入式 SQL 语句转换为与数据库服务器通信的 GBase 8s ESQL/C 函数调用。GBase 8s ESQL/C 预处理器生成一个 C 源文件并调用 C 编译器。然后 C 编译器预处理并编译您的源文件并链接到其它 C 源文件、对象文件或与其它 C 程序相同的库文件。如果 esql 在这些步骤中没有遇到错误，那么它会生成一个可执行文件。可以像任何 C 程序一样运行编译的 GBase 8s ESQL/C 程序。当程序运行时，它调用 GBase 8s ESQL/C 库程序；库程序域数据库服务器建立通信，以执行 SQL 操作。

下图说明了 GBase 8s ESQL/C 程序成为可执行程序的过程。

图: GBase 8s ESQL/C 和 C 之间的关系



重要： 请记住，使用缺省的编译顺序，在调用 C 编译器之前，esql 处理 GBase 8s ESQL/C 预处理器指令。因此，GBase 8s ESQL/C 指令在 C 编译器执行任何预处理之前生效。不能在 C 预处理器指令中访问 GBase 8s ESQL/C 定义，也不能使用 C 预处理器执行 GBase 8s ESQL/C 语句的条件编译。

先运行 C 预处理程序

使用 GBase 8s ESQL/C，您可以在编译 GBase 8s ESQL/C 程序时更改处理的缺省顺序。GBase 8s ESQL/C 允许您首先在 GBase 8s ESQL/C 源文件上运行 C 预处理器，然后将该文件传递到 GBase 8s ESQL/C 预处理器。此功能使您的 GBase 8s ESQL/C 程序访问由 C 预处理程序指令提供的变量。

2.2.2 esql 命令

要从 GBase 8s ESQL/C 源文件创建可执行 C 程序，请使用 esql 命令。GBase 8s 安装脚本将 esql 命令作为 GBase 8s ESQL/C 产品的一部分进行安装。本节描述 esql 命令的作用以及如何使用它。

esql 命令按以下步骤执行：

将嵌入式 SQL 语句转换为 C 语言代码。

将 ESQL/C 源文件转换为 C 语言源文件。

编译由 C 编译器生成的文件以创建对象文件。

创建资源编译器并链接您在 Windows™ 的 esql 命令行上指定的任何资源文件。

将对象文件与 GBase 8s ESQL/C 库和您自己的库链接。

使用 esql 命令的要求

在使用 esql 之前，请确保：

GBase 8s ESQL/C 源文件的文件名称具有 .ec 扩展名。如果希望在 GBase 8s ESQL/C 预处理器执行之前运行 C 预处理器，可以使用 .ecp 扩展名。

已正确设置 GBASEDBTDIR 环境变量和包含 GBASEDBTDIR 目录的 BIN 目录的路径（在 UNIX™ 操作系统中是$GBASEDBTDIR/bin ，在 Windows™ 环境中是 %GBASEDBTDIR%\bin ） PATH 环境变量。

esql 目录的语法

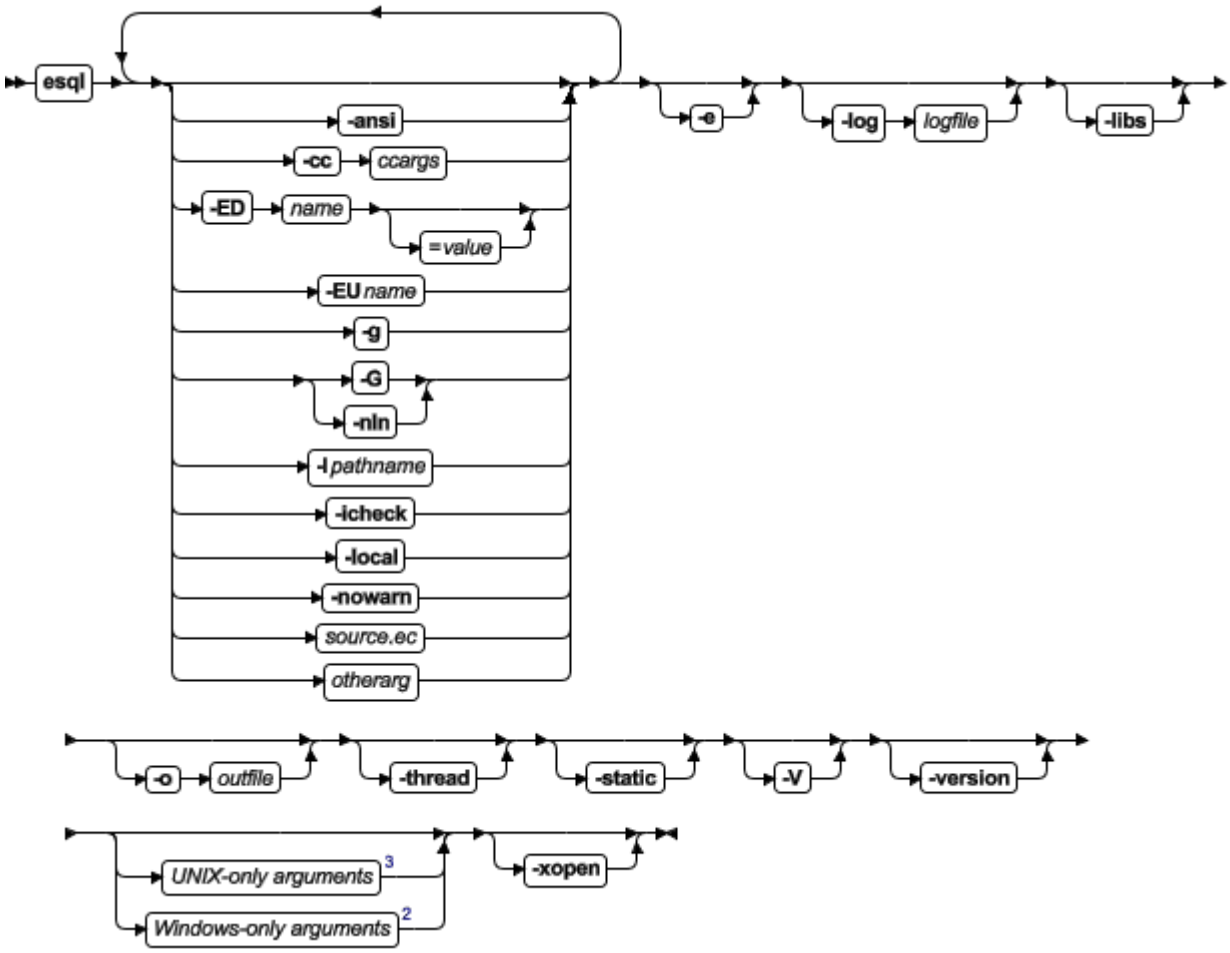
下列主题描述了 esql 命令的语法。

本节安装影响处理的阶段组织命令行选项：

预处理选项确定 esql 如何转换嵌入式 SQL 语句。

当 C 编译器将 C 源转换为对象文件时，编译选项影响编译阶段。

当 C 编译器链接对象代码以生成可执行文件时，链接选项影响链接阶段。



-ansi

如果源文件对 ANSI 标准 SQL 语法使用 GBase 8s 扩展，则导致 esql 发出警告。该参数仅影响命令行右侧的源文件。

-cc *ccargs*

将 *ccargs* 传递给 C 编译器，而无需解释或更改它们。变量 *ccargs* 表示 **-cc** 和下一个出现在任何这些参数之间的所有参数。

-l（仅限于 Windows™）

-r（仅限于 Windows）

-f（仅限于 Windows）

任何文件名称（除了为此选项的参数的文件名称）

-e

仅进行预处理，不编译或链接。ESQL/C 预处理器生成一个具有 .c 扩展名的 C 源文件。-ED*name*

创建 *name* 的定义。其效果与源文件中包含 *name* 的 ESQL/C **define** 指令相同。如果包含 *value* = ，那么此定义设置为 *value*。

-EU*name*

取消定义名为 *name* 的 GBase 8s ESQL/C 定义。就像源文件为此名称包含 GBase 8s ESQL/C **undef** 指令。

-g

将源文件的最后的 **-G** 选项的作用转换到命令行上该选项的右侧。

-G

通常 **#line** 指令将添加到 C 源代码中，以便 C 编译器在 C 文件中检测到错误时，可以将它引导到 GBase 8s ESQL/C 文件中的正确行。**-G** 选项会关闭在命令行中跟随它的 GBase 8s ESQL/C 源文件的此功能。使用 **-g** 参数重启此功能。**-nln** 参数是 **-G** 的同义词。

-I*pathname*

将 *pathname* 添加到 GBase 8s ESQL/C 和 C include 文件的搜索路径。搜索路径用于搜索和 **include** 和 **#include** 指令中命名的文件。

-icheck

告诉 esql 添加代码，如果将空值返回没有与其相关联的指示变量的主机变量，则会生成错误。此参数仅影响命令行右侧的源文件。

-local

指定在源文件中声明的静态游标名称和语句 ID 是文件的本地名称。如果不使用 **-local** 选项，游标名称和语句 ID 缺省情况下为全局实体。此参数只影响命令行右侧的源文件。

-log *logfile*

将 GBase 8s ESQL/C 预处理器生成的错误和警告消息发送给指定的文件而不是标准输出。该选项仅影响预处理器错误和警告。

-libs

防止所有编译和链接，并显示所有基于其他选项的链接的库的名称。

-nln

-G 的同义词

-nowarn

抑制来自预处理器的警告消息。仍会发出错误消息。此参数只影响命令行右侧的源文件的预处理。

-o *outfile*

指定由编译器创建的输出文件的名称。

otherarg

esql 无法直接识别或处理的参数传递给 C 编译器。此过程允许您在命令行中包含库、资源文件、C 编译器选项和类似的参数。如果传递给 C 编译器的参数与 esql 参数之一冲突，则使用 **-cc** 选项将其从 esql 中保护。

source.ec

具有缺省后缀 .ec 的 GBase 8s ESQL/C 源文件。

-thread

告诉 GBase 8s ESQL/C 预处理器创建线程安全代码。

-static

链接 GBase 8s 静态库而不是缺省 GBase 8s 标准库。

-V

打印您的 GBase 8s ESQL/C 预处理器的版本信息然后退出。如果指定此参数将忽略其它参数。

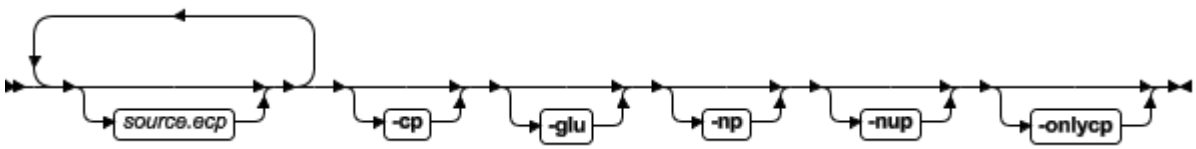
-version

打印您的 GBase 8s ESQL/C 预处理器的版本和安装信息然后退出。如果指定此参数将忽略其它参数。

-xopen

为使用 GBase 8s 扩展到 X/Open 标准的 SQL 语句生成警告消息。它还表明动态 SQL 语句使用数据类型的 X/Open 代码集（使用 GET DESCRIPTOR 和 SET DESCRIPTOR 语句或 **sqlda** 结构时）。

仅限于 UNIX 的参数



-cp

在处理 *source.ec* 文件时导致 esql 在 GBase 8s ESQL/C 预处理器之前运行 C 预处理器。文件中的 SQL 关键字被保护以防止被 C 预处理器解析，此保护在 C 预处理器运行完毕后撤销。此参数只影响命令行右侧的源文件。

-glu

编译使您的应用程序可以使用 GLU（对于 Unicode 为 GLS）。

-np

在 GBase 8s ESQL/C 预处理器处理之前，防止由 C 预处理器处理的源文件中的 SQL 语句的保护。此参数仅影响命令行右侧的源文件。

-nup

无取消保护模式。SQL 关键字包含在 C 预处理器运行完后不会被删除。该编译在 C 预处理器之后停止，结果放置扩展名为 **.icp** 的文件中。

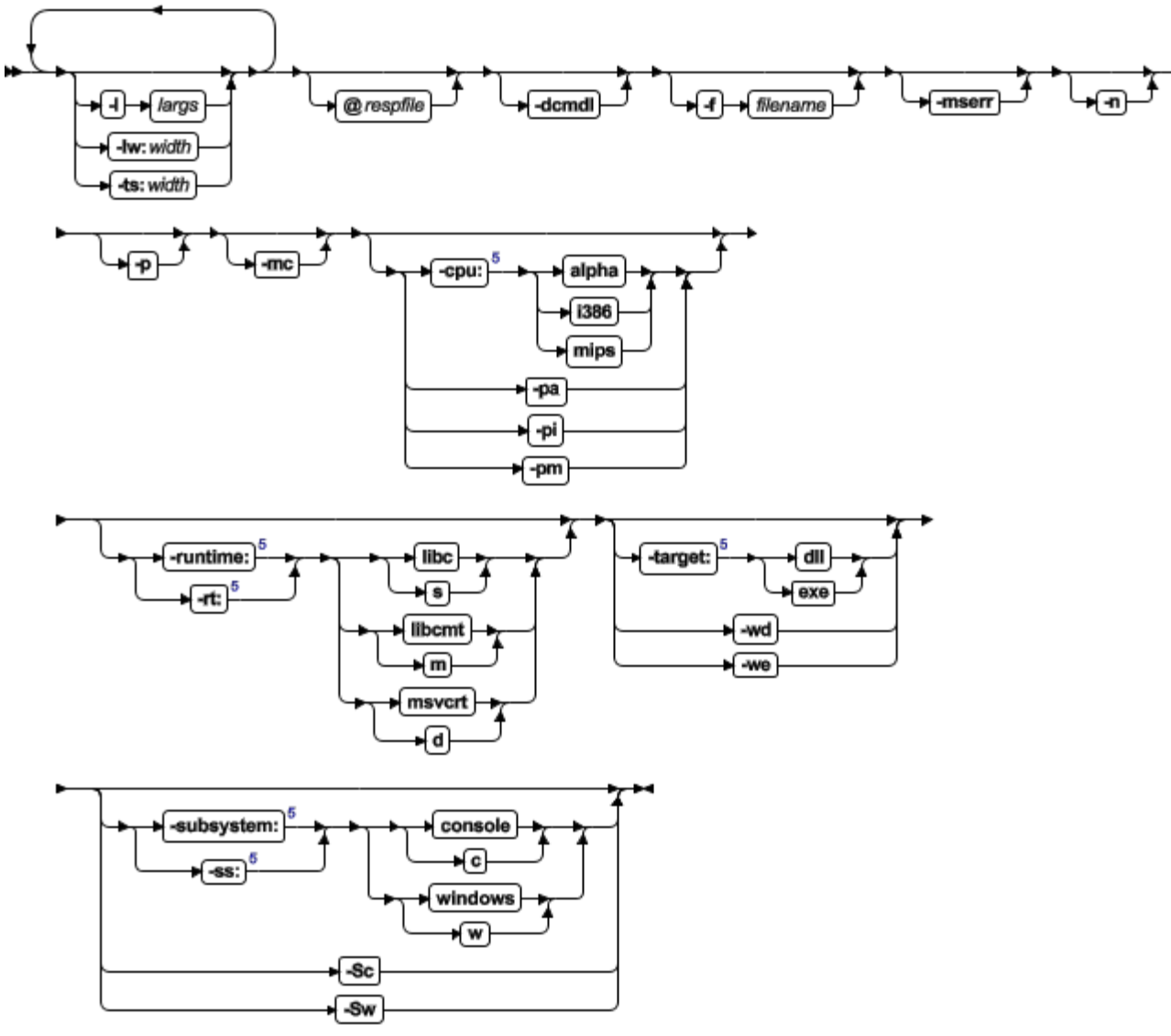
-onlycp

这种模式类似于 **-cp** 模式。它强制 C 预处理器在 GBase 8s ESQL/C 预处理器之前首先运行。但是，在 C 预处理器运行之后停止处理，将结果留在 **.icp** 文件中。

source.ecp

带有特殊后缀 **.ecp** 的 GBase 8s ESQL/C 源文件。它被视为正在使用 **-cp** 选项的 GBase 8s ESQL/C 文件。

仅限于 Windows 的参数



@ respfile

指定包含其它选项的文件。

-bc

告诉预处理器使用 Borland C 编译器而不是 Microsoft™ Visual C++ 编译器。

-cpu:

如果正在使用 Borland C 编译则此参数无效。该参数告诉 esql 什么类型的处理器希望被可执行程序优化。有三个可能值：

alpha

对于与 Alpha 架构兼容的处理器。

i386

适用于与 Intel386 架构兼容的处理器。这是缺省值。

mips

适用于使用 MIPS32 或 MIPS64 指令集架构（ISA）的处理器。

-dcmdl

显示用于启动 C 编译器的命令行。这使您可以直观地验证所使用的选项。

-f *filename*

指定包含其它 GBase 8s ESQL/C 源文件名称的文件的名称。

-l *largs*

将 *largs* 传递给链接，而不需解释或更改它们。*largs* 是 **-cc** 和 **-r** 选项或行末尾之间的所有参数。

-lw:*width*

当 GBase 8s ESQL/C 源文件转换为 C 因为我时，该参数导致 C 源文件中的行被包装在 *width* 指示的列的位置。此参数仅影响命令行右侧的源文件。

-mc

告诉预处理器使用 Microsoft Visual C++ 编译器编译和链接。

-mserr

提供 Microsoft 样式消息和警告。

-n

防止 esql 在运行时输出版本消息。

-p

-e 的同义词。

-pa

-cpu:alpha 的同义词。

-pi

-cpu:i386 的同义词。

-pm

-cpu:mips 的同义词。

-rt:

-runtime: 的同义词。

-runtime:

确定 C 运行时与可执行文件链接的库的类型。库类型的指示符必须遵循此选项并且两者之间不能有空格。该类型必须是以下之一：

d

链接多线程共享库。如果未指定 **-runtime:** 则它是缺省库。还可以使用库名称的代替 **d**。如果使用 Microsoft Visual C++ 编译，则库的名称是 **msvcrt**。如果使用 Borland C，则是 **cw32mti**。

m

链接静态多线程共享库。还可使用库名称代替 **m**。如果使用 Microsoft Visual C++ 编译，则库的名称是 **libcmt**。如果使用 Borland C，则是 **cw32mt**。不能与 **-static** 选项一起使用。

s

链接静态单线程库。还可以使用使用库名称代替 **s**。如果使用 Microsoft Visual C++ 编译，则库的名称是 **libc**。如果使用 Borland C，则是 **cw32**。不能与 **-static** 选项一起使用。

t

仅在使用 Borland C 时才能使用此选项。它链接静态多线程库。还可以使用使用库名称 **cw32i** 代替 **t**。不能与 **-static** 选项一起使用。

-Sc

-subsystem:console 的同义词。

-ss:

-subsystem: 的同义词。

-Sw

-subsystem:windows 的同义词。

-subsystem:

确定要链接到可执行文件中的子系统。子系统类型的指示符必须遵循此选项，之间不

能有空格。该类型必须是以下之一：

console

缺省类型。它创建控制台应用程序。该指示符可以缩写为 **c**.

windows

创建 Windows 应用程序。该指示符可以缩写为 **w**。

-subsystem: 选项可以缩写为 **-ss:**。

-target:

确定创建文件的类型。目标类型的指示符必须遵循此选项，并且两端不能具有空格。该指示符必须是以下一种：

dll

创建动态加载库（DLL）文件。

exe

缺省类型。会创建常规的可执行文件。

-ts:*width*

在创建 C 源文件时，告诉预处理器在每个 *width* 列上定义制表位。默认情况下，预处理器集选项卡每八列停止一次。

-v

-V 的同义词。

-wd

-target:dll 的同义词。

-we

-target:exe 的同义词

影响预处理的选项

GBase 8s ESQL/C 程序必须在 C 编译器能编译它之前进行预处理。GBase 8s ESQL/C 预处理器将嵌入式 SQL 语句转换为 C 语言代码。

可以使用以下主题描述的所有预处理器选项。仅用于预处理，或用于预处理、编译和链接。

检查版本号

GBase 8s ESQL/C 程序必须在 C 编译器能编译它之前进行预处理。GBase 8s ESQL/C 预处理器将嵌入式 SQL 语句转换为 C 语言代码。

可以使用以下主题描述的所有预处理器选项。仅用于预处理，或用于预处理、编译和链接。

将选项与文件相关联

许多预处理器仅影响命令行选项右侧显示的文件。例如此命令行：

```
esql -G source1.ec -ansi source2.ec
```

-G 选项影响 source1.ec 文件，**-ansi** 和 **-G** 选项影响 source2.ec 文件。

预处理，无需编译和链接

缺省情况下，esql 命令使 GBase 8s ESQL/C 程序被预处理、编译和链接。esql 命令的输出时一个可执行文件。可以指定 **-e** 选项抑制您的 GBase 8s ESQL/C 程序编译和链接。使用此选项，esql 只预处理处理此代码。该命令的输出是 C 源文件（.c 扩展名）。

例如，要处理 demo1.ec 文件中的查询，使用以下命令：

```
esql -e demo1.ec
```

上面的命令会生成名为 demo1.c 的 C 源文件。下面的 esql 命令预处理 demo1.ec，检查 ANSI 标准语法的 GBase 8s 扩展，并且不使用行编号：

```
esql -e -ansi -G demo1.ec
```

生成线程安全代码

可以使用 **-thread** 选项指示预处理器生成线程安全代码。

必须与THREADLIB 环境变量一起使用此选项在编译应用程序时指定使用哪个线程包。

对于 Windows™ 环境，GBase 8s 通用库（**libgen** 、**libos** 、**libgls** 、**libafs** 和 **libsql**）是共享的、线程安全的 DLL。因此 esql 命令自动链接是共享的、线程安全的 DLL。当在 Windows 环境中编译多线程应用程序时不要设置 THREADLIB 环境变量。

检查 ANSI 标准 SQL 语法

当编译 GBase 8s ESQL/C 程序是，可以指示预处理器按照以下两种方法之一检查 ANSI 标准 SQL 语法的 GBase 8s 扩展：

设置 DBANSIWARN 环境变量。

设置 DBANSIWARN 环境变量之后，每次编译或运行 GBase 8s ESQL/C 程序时，GBase 8s ESQL/C 检查是否符合 ANSI。

可以在编译时指定 **-ansi** 选项以检查 ANSI 兼容性。

-ansi 选项不会使 GBase 8s ESQL/C 在运行时检查 ANSI 兼容性。

使用 **-ansi** 选项，GBase 8s ESQL/C 预处理器在遇到 ANSI SQL 语法的 GBase 8s 扩展时生成警告消息。下面的 esql 命令预处理、编译并链接 demo1.ec 程序并验证它不包含 ANSI 标准语法的 GBase 8s 扩展：

```
esql -ansi demo1.ec
```

如果使用 **-ansi** 和 **-xopen** 选项编译程序，则 GBase 8s ESQL/C 预处理器对 ANSI 和 X/Open SQL 语法的 GBase 8s 扩展生成警告消息。

在预处理时定义和取消定义

可以使用 **-ED** 和 **-EU** 选项在 GBase 8s ESQL/C 预处理时创建或删除定义。

创建全局定义，使用以下任一格式的 **-ED** 选项：

使用 **-ED***name* 语法定义 Boolean 符号，如下所示：

```
esql -EDENABLE_CODE define_ex.ec
```

使用 -EDname=value 语法定义整数常量，如下所示：

```
esql -EDMAXLENGTH=10 demo1.ec
```

-ED*name* 等价于在您的 GBase 8s ESQL/C 程序上方带有 *name* 的 **define** 预处理程序指令（**$define** 或 EXEC SQL **define**）。

要删除或取消定义，使用以下 **-EU** 选项的语法：

```
-EUname
```

-EU 选项对整个文件具有全局影响，无论 *name* 其它的 **define** 指令如何。

限制： 在 **ED** 或 **EU** 和符号名称之间不要有空格。

与 **define** 和 **undef** 语句一样，GBase 8s ESQL/C 预处理器在预处理的第 1 阶段处理 -ED 和 -EU 选项（在它预处理源文件中的代码之前）。

下图显示使用条件编译的代码段（**ifdef** 和 **ifndef** 指令）。

图: 使用 ifdef 、ifndef 和 endif 的 ESQL/C 摘录

```
/* define_ex.ec */
        #include <stdio.h>
        EXEC SQL include sqlca;
        EXEC SQL define ENABLE_CODE;

        main()
        {
        ⋮

        EXEC SQL ifdef ENABLE_CODE;
        printf("First block enabled");
        EXEC SQL endif ENABLE_CODE;
        ⋮

        EXEC SQL ifndef ENABLE_CODE;
        EXEC SQL define ENABLE_CODE;
        EXEC SQL endif ENABLE_CODE;
        ⋮

        EXEC SQL ifdef ENABLE_CODE;
        printf("Second block enabled");
        EXEC SQL endif ENABLE_CODE;
        }
```

对于图 1 中显示的代码段，以下 esql 命令行不会生成代码，因为命令行删除了整个源文件的 ENABLE_CODE 定义：

```
esql -EUENABLE_CODE define_ex.ec
```

检查缺失的指示变量

如果包含 **-icheck** 选项，GBase 8s ESQL/C 预处理器会在程序中生成代码，如果 SQL 语句将 null 值返回给没有关联指示变量的主变量，则会返回运行时错误的代码。例如，以下命令告诉预处理器将检查空值的代码插入到 demo1.ec 程序中：

```
esql -icheck demo1.ec
```

如果未使用 **-icheck** 选项，则当数据库服务器将 null 值返回给不带有指示变量的主变量时，GBase 8s ESQL/C 不会生成错误。

命名 include 文件的位置

-I 预处理器选项允许您为预处理器搜索的 GBase 8s ESQL/C 和 C include 文件的目录命名。

该选项对 GBase 8s ESQL/C 和 C 预处理都有效，如下所示：

GBase 8s ESQL/C 预处理程序（esql）只处理 GBase 8s ESQL/C include 文件。

可以使用 **include** 预处理器指令 **$include** 或 EXEC SQL **include** 指定这些文件。

C 预处理程序（cc）仅处理 C include 文件。

可以使用 **#include** 预处理器程序语句指定这些文件。因为 C 预处理在 GBase 8s ESQL/C 编译完成后开始，GBase 8s ESQL/C include 文件处理后再处理 C include 文件。

预处理程序在预处理 C include 文件（**#include** 预处理程序语句指定的文件）之前将 **-I** 选项传递给 C。**-I** 选项的语法如下：

```
esql -Idirectory esqlcprogram.ec
```

如果标准 C 库函数 fopen() 、fread() 和 fclose() 可以访问它们，则该 *directory* 可以安装在远程文件系统上。

以下 esql 命令命名 UNIX™ 目录 /usr/johnd/incls ，以在 **demo1** 程序中搜索 GBase 8s ESQL/C 和 C include 文件：

```
esql -I/usr/johnd/incls demo1.ec
```

一个 **-I** 选项列出一个目录。要列出多个目录，必须在命令行列出多个 **-I** 选项。

要在 Windows™ 环境中的 C:\dorrie\incl 和 C:\johnd\incls 目录中搜索，必须发出以下命令：

```
esql -IC:dorrie\incl -IC:\johnd\incls demo1.ec
```

当预处理器收到 **include** 指令时，它会通过搜索路径查看要包含的文件。它按照以下顺序搜索目录：

当前目录

-I 预处理程序制定的目录 （按照在命令行中指定的顺序）

UNIX 操作系统上的 $GBASEDBTDIR/incl/esql 目录以及 Windows 环境中的 %GBASEDBTDIR%\incl\esql 目录（$GBASEDBTDIR 和 %GBASEDBTDIR% 表示该名称环境变量的内容）

/usr/include 目录

行号

缺省情况下，GBase 8s ESQL/C 预处理器将 **#line** 目录放在 .c 文件中，以便如果 C 编译器检测到错误，则会引导您在 GBase 8s ESQL/C 源文件中生成问题 C 代码的行。如果您想要定向到 C 文件中的问题行，可以使用 **-G** 选项关闭行号。**-G** 选项可以防止在命令行中的源代码文件中生成 **#line** 指令。要重新使用行号，请使用 **-g** 选项，**-g** 选项后面的文件将生成 **#line** 指令。

游标名称和语句 ID

缺省情况下，GBase 8s ESQL/C 将游标名称和语句 ID 定义为全局实体。如果使用 **-local** 选项，您在文件中声明的静态游标名称和静态语句 ID 是该文件的局部变量。

要创建本地名称，GBase 8s ESQL/C 将向 GBase 8s ESQL/C 程序中的游标名称和语句 ID 添加一个唯一的标记（长度为 2 到 9 个字符）。如果游标名称（或语句 ID）和唯一标记的长度超过 128 个字符，则会收到一条警告消息。

-local 选项主要用于与以前版本的 GBase 8s ESQL/C 创建的应用程序的兼容性。当编译新的应用程序时不要使用此选项。不要将此消息与 **-local** 标志混淆。如果混淆它们，您可能接收到不可预知的结果。

如果使用 **-local** 选项，必须在每次重命名源文件时重新编译它们。

重定向错误和警告

缺省情况下，esql 会将其生成的错误和警告消息指向标准输出。如果要将错误和警告放入文件中，请使用 **-log** 选项和文件名。

例如，下面的 esql 命令编译程序 demo1.ec 并将错误发送到 err.out 文件：

```
esql -log err.out -o demorun demo1.ec
```

此选项仅影响 GBase 8s ESQL/C 预处理程序生成的错误警告。编译和链接阶段的警告仍然转到标准错误输出，例如 UNIX™ 上的 **stderr**。

抑制警告

缺省情况下，预处理器在处理 GBase 8s ESQL/C 文件时生成警告。要抑制这些警告消息，使用 **-nowarn** 选项。该选项对错误消息没有影响。

启用 GLS（Unicode）GLU 功能

GLS Unicode（GLU）是一种功能，允许您的应用程序在处理 Unicode 是使用 Unicode 的国际标准组件库而不是一般的 GLS 库。

使用 ICU 库的主要优势是，在对 Unicode 字符进行整理时，它们会考虑区域设置，GLS 库不会。

启用 GLU：

使用 esql 命令的 **-glu** 选项编译应用程序。

在客户端和数据库服务器的环境中将 GL_USEGLU 设置为 1。GL_USEGLU 环境变量必须在服务器启动前和数据库创建前设置为 1。

选择使用 Unicode 或 GB18030-2000 作为代码集的区域设置。

X/Open 标准

-xopen 选项告诉 GBase 8s ESQL/C 预处理器您的程序使用 X/Open 标准。

当指定此选项时，预处理器执行以下任务：

它检查 X/Open 标准语法的 GBase 8s 扩展。

如果在您的代码中包含 X/Open 标准语法的 GBase 8s 扩展，则预处理器生成警告消息。

使用用于 SQL 数据类型的 X/Open 代码集。

GBase 8s ESQL/C 在动态管理结构（系统描述符区间或 **sqlda** 结构）中使用这些代码指示列数据类型。 GBase 8s 在 sqlxtype.h 头文件中定义这些代码。

如果在 GBase 8s ESQL/C 程序中使用 X/Open SQL，您必须使用 **-xopen** 选项编译同一应用程序中的其它源文件。

如果使用 **-xopen** 和 **-ansi** 选项编译程序，GBase 8s ESQL/C 预处理器为 GBase 8s 扩展生成 X/Open 和 ANSI SQL 语法的警告消息。

在运行 ESQL/C 预处理前运行 C 预处理器

GBase 8s ESQL/C 源文件的编译可以遵循缺省顺序。GBase 8s ESQL/C 预处理首先在源文件上运行，或者可以允许 C 预处理器在 GBase 8s 源文件之前运行 GBase 8s ESQL/C 预处理。

GBase 8s ESQL/C 源文件的缺省编译顺序如下：

GBase 8s ESQL/C 预处理器执行以下步骤来从 GBase 8s ESQL/C 源文件创建 .c 文件：

在处理所有 include 指令时（$include 和 EXEC SQL include 语句）时，将 GBase 8s ESQL/C 源文件合并到源文件中。

当处理所有 define（$define 和 EXEC SQL define）和 undef（$undef 和 EXEC SQL undef）指令时，创建或删除编译时定义。

处理任何条件编译指令（ifdef 、ifndef 、else 、elif 、endif）并将嵌入式 SQL 语句转换为 GBase 8s ESQL/C 函数调用和特定的数据结构。

C 预处理器执行以下操作：

在处理所有 include 指令（#include）时，将 C 头文件合并到源文件中。

当它处理所有的 C 语言指令 define（#define）和 undef（#undef）时，创建或删除编译时定义。

处理 C 条件编译指令（#ifdef 、#ifndef 、#else 、 #elif 、 #endif）

C 编译器、汇编器和链接器以通常的方式运行，就像 C 源文件一样，将 C 代码文件转换为可执行文件。

编译的缺省顺序是有限制的，因为无法使用 C 系统文件或自定义 C 头文件中定义的 **#defines** 或 **typedefs** 来定义 GBase 8s ESQL/C 主机变量和常量，也不能将它们用于 GBase 8s ESQL/C 代码的条件编译。使用缺省编译顺序，直到 GBase 8s ESQL/C 预处理器运行之后，C 头文件才被包含在 GBase 8s ESQL/C 源代码中，使这些定义对 GBase 8s ESQL/C 预处理器不可用。

首先运行 C 预处理器的选项

可以首先在 GBase 8s ESQL/C 源文件上运行 C 预处理器，从而在 GBase 8s ESQL/C 源文件之前扩展 GBase 8s ESQL/C 源文件中的任何依赖于 C 的 **typedefs** 或 **#defines**，然后才能在该源文件上运行 此源文件。可以通过以下任何方式执行此操作：

将 **-cp** 或 **-onlycp** 选项传递给 esql 命令。

两者强制首先运行 C 预处理器。但是在 **-cponly** 选项的情况下，编译在 C 预处理器运行后停止，并将结果放入 **.icp** 文件中。

创建带有 .ecp 扩展的 GBase 8s ESQL/C 源文件。

该处理自动触发文件上 **-cp** 选项。

将 CPFIRST 环境变量设置为 TRUE（使用大写字母）。

使用 eprotect.exe 实用程序。

CPFIRST 环境变量（UNIX(TM)）

CPFIRST 环境变量指定 C 预处理器在所有的 GBase 8s ESQL/C 文件上在 GBase 8s ESQL/C 预处理之前运行。

将此环境变量设置为 TRUE（仅限大写）以在所有的 GBase 8s ESQL/C 源文件上运行 C 预处理器，而不管 **-cp** 选项是否传递给 esql 命令，无论源文件是否具有 .ec 或 .ecp 扩展名。

使用 eprotect.exe 应用程序（Windows(TM)）

Windows™ 用户可使用 eprotect.exe 实用程序在 GBase 8s ESQL/C 源文件上运行预处理器。

eprotect.exe 实用程序保护所有的 SQL 关键字不被 C 预处理器解析。eprotect.exe -u 选项移除 SQL 关键字的保护。

要更改 Windows 上 GBase 8s ESQL/C 源文件的预处理器顺序：

运行以下命令：

```
%GBASEDBTDIR%\lib\eprotect.exe filename.ec filename.c
```

该命令保护所有的 SQL 关键字不被 C 预处理器解析并将结果写到 *filename*.c 文件中。

运行以下命令：

```
cl /E filename.c    > filename2.c
```

该命令在源文件 *filename*.c 上运行 C 预处理器，并将结果写到 *filename2*.c 文件中。

运行以下命令：

```
%GBASEDBTDIR%\lib\eprotect.exe -u filename2.c filename.ec
```

该命令移除 SQL 关键字保护并将结果存储到 *filename*.ec 中。

在源文件上运行 esql 并编译它

C 预处理器首先运行时的编译顺序

当用户选择在 GBase 8s ESQL/C 预处理器之前在源文件上运行 C 预处理器时，该文件将经历一些编译顺序。

在源文件上运行 eprotect 实用程序以保护所有的 SQL 关键字不被 C 预处理器解析。

在源文件上运行 C 预处理器。

eprotect 实用程序使用 **-u** 模式在 C 预处理器的输出上运行，以删除 SQL 关键字的保护。

GBase 8s ESQL/C 预处理器在 C 预处理器的输出上运行，它不再具有任何 SQL 关键字保护。

GBase 8s ESQL/C 预处理程序的输出由 C 编译器和链接器进行编译和链接，以生成可执行文件。

定义基于 C #defines 和 typedefs 的主机变量

当在源文件上运行 C 预处理器，它会扩展源文件中包含的所有 C 头文件的内容。此扩展使得可以根据 C 头文件中的类型和 **#define** 和 **typedef** 语句在 GBase 8s ESQL/C 源文件中使用主机变量。这里给出的示例表示共享头文件的一些优点。在以下示例中，GBase 8s ESQL/C 和 C 源文件使用同一 C 头文件。

图: 共享 C 头文件的 ESQL/C 和 C 摘录

```
/*header file i.h*/
                #define LEN 15
                typedef struct customer_t{
                int       c_no;
                char      fname[LEN];
```

```
char     lname[LEN];
} CUST_REC;
⋮

/*cust.ec*/
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
#include "i.h"
EXEC SQL END DECLARE SECTION;

int main()
{

EXEC SQL BEGIN DECLARE SECTION;
CUST_REC cust;
EXEC SQL END DECLARE SECTION;
⋮

}


/*name.c*/
#include “i.h”
int main ()
{...
CUST_REC cust;
⋮

}
```

在以下示例中，GBase 8s ESQL/C 源文件包含基于在 time.h 系统头文件中定义的类型的主变量。

图: 使用 C 系统头文件中定义的类型的主机变量的 ESQL/C 摘录

```
/*time.ec*/

#include <time.h>

main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
time_t time_hostvar;
EXEC SQL END DECLARE SECTION;
⋮

}
```

C 头文件可以包含在 GBase 8s ESQL/C 源文件中的任何地方。但是，要基于 C 头文件中定义的 **#defines** 和 **typedefs** 的 GBase 8s ESQL/C 文件中定义主机变量，必须在

EXEC SQL 声明部分包含 C 头文件。

对比图 3中的示例，导致错误 -33051: Syntax error on identifier or symbol ‘name_hostvar 其中图 1 中的示例不会。唯一的区别是，在图 1的示例中，具有 **#define** 的 C 头文件和在 EXEC SQL 声明部分中使用的 **typedef** 包含在该声明部分中。

图: 根据包含在声明部分之外的 C 头文件定义主机变量的 ESQL/C 摘录

```
/*header file i.h*/
        #define LEN 15
        typedef struct customer_t{
        int       c_no;
        char      fname[LEN];
        char      lname[LEN];
        } CUST_REC;
        ⋮

        /*cust.ec*/
        #include "i.h"

        int main()
        {

        EXEC SQL BEGIN DECLARE SECTION;
        CUST_REC cust;
        ⋮

        }
        ⋮

        ...Leads to error -33051...
```

在 EXEC SQL 声明部分中允许所有有效的 C 声明语法

当 GBase 8s ESQL/C 预处理器在文件上运行时，它会扩展 GBase 8s ESQL/C 源文件中头文件的所有内容，其中头文件包含在源文件中。因此，在 EXEC SQL 声明部分中包含 C 头文件的一个后果是，在它们通过 C 预处理器之后，所有类型的 C 声明语法都包含在 EXEC SQL 声明部分中。现在，您可以在 EXEC SQL 声明部分的 EXEC SQL 声明部分中包含所有有效的 C 声明语法。但是，您只能根据主机变量数据类型中描述的某些类型声明主机变量。

移除 C 头文件中的语句

如果 GBase 8s ESQL/C 预处理器在 C 头文件中存在某些语句的问题，可以从 GBase 8s ESQL/C 预处理器执行的预处理中排除特定行。如下所示：

图: 使用通用 C 头文件的 ESQL/C 和 C 摘录

```
/*header file i.h*/
        #ifndef ESBDS /*define empty macros, if included by a C\
        source*/
```

```
#define ESBDS
#define ESEDS
#endif
⋮

ESEDS
statement that you do not want ESQL/C    preprocessor to see
ESBDS


/*name.ec*/
#define ESBDS “EXEC SQL BEGIN DECLARE SECTION;”
#define ESEDS “EXEC SQL END DECLARE SECTION”
main ()
{...
EXEC SQL BEGIN DECLARE SECTION;
#include “i.h”
EXEC SQL END DECLARE SECTION;
⋮

}
```

SQL 关键字保护

如果 GBase 8s ESQL/C 文件中的代码在传递给 GBase 8s ESQL/C 预处理器之前未保护地传递给 C 预处理器，则 C 预处理器可能解析某些 SQL 关键字。在以下示例中，SQL 关键字 NULL 由值为零的 C 预处理替代，该预处理器将创建了一个有效的 SQL 语句，但是将一个值插入到 **orders** 表中，而不是程序员预期的值：

```
EXEC SQL insert into orders (shipcharge) values (NULL);
```

当用户提供在 GBase 8s ESQL/C 预处理器之前运行 C 预处理器的选项时，实用程序 **eprotect** 在 GBase 8s ESQL/C 源文件上运行 C 预处理器之前运行。eprotect 实用程序向前缀为 **SQLKEYWORD_** 的 SQL 语句中发生的任何 SQL 关键字添加前缀。这个前缀附在 SQL 语句中所有 SQL 关键字的上，以 EXEC SQL 指令开头，并以分号为结尾。当包含前面提到的 select 语句 GBase 8s ESQL/C 源文件传递给 C 预处理器时，SELECT 语句具有以下格式：

```
EXEC SQL SQLKEYWORD_insert SQLKEYWORD_into orders (order_num)
        SQLKEYWORD_values (SQLKEYWORD_NULL);
```

C 预处理器在 GBase 8s ESQL/C 源文件中运行后，esql 命令使用 **-u** 模式运行 eprotect 实用程序，在它在 C 预处理器的输出上运行 GBase 8s ESQL/C 预处理器之前删除所有的 **SQLKEYWORD_** 前缀。

SQL 关键字保护和美元符号（$）

GBase 8s ESQL/C 源文件中的所有的 SQL 语句可以以 EXEC SQL 关键词开头或者使用 $ 前缀。以下所有对语句都是等效的：

```
EXEC SQL BEGIN DECLARE SECTION;
$BEGIN DECLARE SECTION;
```

```
EXEC SQL connect to ‘database9’;
$connect to ‘database9’;

EXEC SQL select fname into :hostvar1 from table1;
$ select fname into :hostvar1 from table1;
```

但是，$ 符号还可以出现在 **typedef** 定义的开头，如下所示：

```
$int *ip = NULL;
```

在诸如前面的 **typedef** 示例的情况中，程序逻辑可能需要 C 预处理器将值替换为关键字 NULL 值。在这种情况下，不允许 C 预处理器进行值替换将导致错误。因此，eprotect 实用程序不会在以美元符号（$）开头的 SQL 语句中显示的 SQL 关键字上的 **SQLKEYWORD_** 前缀添加前缀。

重要： 如果要在 GBase 8s ESQL/C 预处理器之前在 GBase 8s ESQL/C 源文件上运行 C 预处理器，并且如果不希望 C 预处理器在源文件中发生的 SQL 语句中替换 SQL 关键字的值。您必须使用关键字EXEC SQL 开始的每个语句，而不是使用美元符号（$）。

特定于 Windows(TM) 环境的预处理器选项

如果在 Windows™ 环境中使用 GBase 8s ESQL/C，则可以使用以下其它预处理器选项。

自动换行

GBase 8s ESQL/C 预处理器将一个嵌入式 SQL 语句转换为一个 C 行。长行可能导致以下调试和编辑的问题。可以使用 **-lw** 选项告诉预处理器在特定列位置自动换行。例如，以下 esql 命令告诉预处理器在 75 列自动换行：

```
esql -lw:75 demo.ec
```

如果省略 **-lw** 选项，则预处理器不会执行自动换行。

更改错误和警告显示

缺省情况下，GBase 8s ESQL/C 预处理器当它处理 GBase 8s ESQL/C 文件时生成错误和警告消息。它在控制台窗口显示这些错误和警告。您可以使用以下命令行选项更改错误和警告消息显示。

使用 **-nowarn** 选项抑制警告消息。该选项对错误消息没有影响。

使用 **-mserr** 选项显示错误和警告消息（以 Microsoft™ 错误消息的格式）。以下文本编辑可以识别此格式并使用它来转到引起错误或警告的 GBase 8s ESQL/C 源文件中的行。

设置制表位

缺省情况下，GBase 8s ESQL/C 预处理器在每八列的位置使用制表符格式化 C 源文件。可以使用 **-ts** 选项设置不同的制表位。例如，以下的 esql 命令告诉预处理器每四个字符设置制表位：

```
esql -ts:4 demo.ec
```

2.2.3 编译和链接 esql 命令的选项

命名可执行文件

可以使用 **-o** 选项显式指定可执行文件的名称。

例如， 下面的 esql 命令生成名为 **inpt** 的可执行文件：

```
esql -o inpt custinpt.ec ordinpt.ec
```

如果 esql 在 Windows™ 操作系统上运行，目标文件的名称缺省为 esql 命令行上第一个 GBase 8s ESQL/C 源文件的名称。该扩展名根据生成的目标类型更改为 .exe 或 .dll。

如果 esql 在 UNIX™ 操作系统上运行，目标文件的名称缺省为您的编译器缺省的文件，通常为 a.out。

设置已创建的可执行文件的类型（Windows(TM)）

esql 命令可用于编译常规可执行文件和动态链接库（DLL）。使用 **-target:** 选项告诉 esql 想要输出的类型。**-target:** 选项仅告诉 esql 如何编译您的应用程序。如果编译为 DLL，那么源代码必须以 DLL 编写，否则编译失败。

将选项传递给 C 编译器

esql 命令处理器将任何命令行中未识别的参数传递给 C 编译器。

例如，因为 esql 不将 **/Zi** 视为选项，下面的 esql 命令将 **/Zi** 选项传递给 C 编译器：

```
esql /Zi demo1.ec
```

如果您想要给 C 编译器传递与 GBase 8s ESQL/C 处理器选项相同的选项，在它们前面放置 **-cc** 选项。esql 命令忽略 **-cc** 和下一次出现以下参数之间的选项：

-l （仅限于 Windows™ ）

-r （仅限于 Windows ）

-f（仅限于 Windows ）

GBase 8s ESQL/C 源文件。

指定特定的 C 编译器（Windows(TM)）

Windows™ 环境中的 ESQL/C 支持以下 C 编译器：

Microsoft™ Visual C++，Version 2.x 或之后的版本

Borland C++，Version 5

Microsoft C 编译器或 Borland C 编译器必须在您可以编译 GBase 8s ESQL/C 程序时已在计算机上安装。缺省的为 C 编译器选项 **-mc**，它启动 Microsoft 编译器。要选择 Borland 编译器，使用 **-bc** 。

编译而无需链接

缺省情况下，GBase 8s ESQL/C 命令处理器预处理、编译和链接 GBase 8s ESQL/C 程

序并创建可执行文件或 DLL 。要抑制 GBase 8s ESQL/C 程序链接，指定 **-c** 选项。使用此选项，esql 仅预处理和编译代码。该命令的输出是 C 对象文件（.obj 扩展名）对于C 源文件（.c）或 GBase 8s ESQL/C 源文件（.ec）。

例如：要预处理和编译 GBase 8s ESQL/C 源文件 demo1.ec，使用下面的命令：

```
esql -c demo1.ec
```

上面的命令生成名为 demo1.obj 的 C 对象文件。下面的 esql 命令预处理 demo1.ec，检查 X/Open 标准语法的 GBase 8s 扩展，并抑制警告消息：

```
esql -c -xopen -nowarn demo1.ec
```

重要： 如果指定冲突的选项 **-c** 和 **-o**，预处理器忽略 **-o** 选项处理 **-c** 选项。预处理器在错误消息中报告此冲突。

特定于Windows(TM) 环境的编译选项

如果在 Windows™ 环境中运行 GBase 8s ESQL/C，则可以指定下列附加选项。

指定项目文件的名称

-f 选项使您在 esql 命令行中指定项目文件的名称。

-f 后面的 *filename* 是包含要编译的 GBase 8s ESQL/C 源文件（.ec）的名称。

例如，假设项目文件 project.txt 包含下列行：

```
first.ec
second.ec
third.ec
```

在此示例中 first.ec 、second.ec 和 third.ec 表示您要编译的GBase 8s ESQL/C 源文件的名称。

下面的 esql 命令使用 **project.txt** 项目文件指定要编译和链接的三个源文件：

```
esql -f project.txt
```

前面的 esql 命令等价于下面的 esql 命令：

```
esql first.ec second.ec third.ec
```

可以使用响应文件完成相同的任务。

创建响应文件

可以在*响应文件*中为 GBase 8s ESQL/C 命令处理器指定命令行参数，并指定 GBase 8s ESQL/C 处理器的文件名。

GBase 8s ESQL/C 响应文件是具有 GBase 8s ESQL/C 命令行选项好文件名的文本文件，以空格分隔、新行、回车符、换行符或这些字符组成。

下面的示例显示了指定名为 resp.esq 的响应文件的语法：

```
esql @resp.esq
```

响应文件 resp.esq 可能包含以下行：

```
-we
```

```
first.ec
second.ec
third.ec
-r foo.rc
```

该响应文件的用途等价于下面的 esql 命令：

```
esql -we -f project.txt -r foo.rc
```

在此示例中 project.txt 是一个项目文件，它包含文件名 first.ec 、second.ec 和 third.ec。

您可能出于以下原因使用响应文件：

命令行限制为 1,024 个字符。如果 esql 选项超出此长度，则必须使用响应文件。

如果经常使用一组或多组 esql 选项，那么可以通过将每个集合放在不同的响应文件中来避免大量打字。您可以 esql 命令中列出相应的响应文件，而不是键入选项。

在 Windows(TM) 环境中 esql 预处理器调用的隐式选项

GBase 8s ESQL/C 命令处理器将编译器和链接器标志隐含地传递给支持的 C 编译器。下表列出了使用指定的 esql 选项时，esql 选项传递的隐式选项。如果您选择创建自己的构建文件，请根据应用程序使用指示的标志。

重要： esql 命令不会将任何选项隐式传递给资源编译器。

表 1. 隐式传递编译器选项

第一列显示了应用于右边列的编译器。编译器为版本 2.x 及之后版本的 Microsoft™ Visual C++ 或 Borland C++, Version 5。接下来的两列显示了模块类型和 esql 选项。第四列和第五列显示了隐式选项：编译器和链接器。

编译器	块类型	esql 选项	隐式选项	
			编译器	链接器
Microsoft Visual C++, Version 2.x or later	执行文件	-target:exe -we	-c -I%GBASEDBTDIR%\incl\esql/D_*systype*/D_*proctype*/*threadtype*/DWIN32	-DEF:deffile -OUT:target -MAP -SUBSYSTEM:systype %GBASEDBTDIR%\lib\isqlt09a.lib %GBASEDBTDIR%\lib\igl4g303.lib %GBASEDBTDIR%\lib\iglxg303.lib %GBASEDBT

				DIR%\lib\igo4g303.lib *libset*
	ll	-target:dll -wd	-c -I%GBASEDBTDIR%\incl\esql/D_*systype*/D_*proctype*/*threadtype*/DWIN32	-DLL -DEF:deffile -OUT:*target* -MAP -SUBSYSTEM:*systype* %GBASEDBTDIR%\lib\isqlt09a.lib %GBASEDBTDIR%\lib\igl4g303.lib %GBASEDBTDIR%\lib\iglxg303.lib %GBASEDBTDIR%\lib\igo4g303.lib *libset*
Borland C++, Version 5	执行文件	-target:exe -we	-c -I%GBASEDBTDIR%\incl\esql-*etarget*-*subtype*-*libtlog*-*libtlg*	-c -Tpe -M -DEF:*deffile-subsystem* %GBASEDBTDIR%\lib\igl4b303.lib %GBASEDBTDIR%\lib\iglxb303.lib %GBASEDBTDIR%\lib\igo4b303.lib c0*t*32.obj

				libset
	(ll	-target:dll -wd	-c -I%GBASEDBTDIR%\incl\esql-*etarget-subtype-libtlog-libtlg*	-c -Tpd -M -DEF:*deffile-subsystem* %GBASEDBTDIR%\lib\igl4b303.lib %GBASEDBTDIR%\lib\iglxb303.lib %GBASEDBTDIR%\lib\igo4b303.lib c0d32.obj *libset*

编译器和链接器中斜体字表示以下定义。

deffile

.def 文件的名称（只要您在命令行指定 .def 选项就执行 -DEF 选项）

libset

库设置（取决于应用程序是 WINDOWS 或 CONSOLE）

libtlg

-D_RTLDLL 表示动态链接库或者 "" 表示共享库

libtlog

-WM 表示多线程库或 "" 表示单线程库

proctype

处理器（X86）的类型

subsystem

ap 表示控制台子系统或 aa 表示 Windows™ 子系统

subtype

可执行控制台的 WC 、Windows 可执行文件的 W、控制台 DLL 的 WCD 、或 Windows DLL 的 WD

systype

子系统的类型（WINDOWS 或 CONSOLE）

t

X 为控制台子系统或 W 为 Windows 子系统

target

可执行文件的名称（第一个 .ec 文件的名称或由 **-o** 命令行选项指定的名称）

threadtype

线程选项的类型（ML 、MT 、MD 取决于 **-runtime** 命令行选项的值）

链接器使用的库设置取决于是创建 Windows 还是控制台应用程序。下表列出了 esql 选项使用的库集。

表 2. 链接器使用的库集

第一列显示应用于右边列的编译器。编译器是版本为 2 或更高版本的 Microsoft Visual C++ 、或版本 5 的 Borland C++。接下来的两列显示了 esql 的信息和链接器使用的库集。

编译器	esql 的选项	链接器使用的库集
Microsoft Visual C++, Version 2.x or later	-subsystem:windows -Sw -ss:w	advapi32.lib wsock32.lib user32.lib winmm.lib gdi32.lib comdlg32.lib winspool.lib
	-subsystem:console -Sc -ss:c	netapi32.lib wsock32.lib user32.lib winmm.lib
Borland C++, Version 5	-subsystem:windows -Sw -ss:w	cw32mti.lib import32.lib
	-subsystem:console -Sc	cw32mti.lib import32.lib

	-ss:c	

链接选项

C 编译器链接执行 编译的链接阶段。

一般链接选项

下列链接选项对 UNIX™ 和 Windows™ 环境都有影响：

链接其它 C 源和对象文件

指定 GBase 8s 通用库的版本

链接其它 C 源和对象文件

可以在 esql 命令行上列出以下类型的文件，以指示您希望链接编辑器链接到生成的对象文件：

C 源文件的格式为 otherCsrc.c

如果列出具有 .c 扩展名的文件，则 esql 通过 C 编译器传递它们，该编译器将它们编译到对象文件（.o 扩展）并链接这些对象文件。

C 对象文件的格式是 otherCobj.o （ UNIX™ 操作系统上）或 otherCobj.obj （ Windows™ 环境中）

如果列出具有 .o 或 .obj 扩展名的文件，则 esql 通过 C 编译器传递它们，该编译器链接这些文件。该链接编辑器将 C 对象文件与适当的 GBase 8s ESQL/C 库函数链接。

库文件，您自己的库文件或与链接器兼容的系统库

模块定义（.def）

资源文件，已编译（.res）或未编译（.rc）

提示： 如果指定未编译的资源文件，则 esql 将它们传递给资源编译器并将生成的 .res 文件链接到 GBase 8s ESQL/C 应用程序。

GBase 8s ESQL/C 命令预处理器直接将这些文件传递给链接器。它还链接需要的库支持 GBase 8s ESQL/C 函数库。可以使用 **-libs** 选项确定 esql 自动链接的库，如下所示：

```
esql -libs
```

指定 ESQL/C 通用库的版本

缺省情况下，esql 命令链接 GBase 8s 通用库的共享库：**libgen** 、**libos** 、**libgls** 、**libafs** 和 **libsql**。要使用共享库，您的计算机必须支持共享内存。

可以使用以下命令行选项更改预处理器链接到您的程序的 GBase 8s 通用库的版本：

-thread 选项告诉预处理器链接 GBase 8s 共享库的线程安全版本。

-static 选项告诉预处理器在 UNIX™ 环境中链接 GBase 8s 通用库的静态库。如果使

用 -static 选项，则不能设置 IFX_LONGID 环境变量。您必须使用 libos.a 重新编译、

可以组合这些选项来告诉预处理器在 GBase 8s 静态库的线程安全版本中进行链接。

特定于 Windows(TM) 的链接选项

下节介绍了只能在 Windows™ 环境中使用的链接选项。

传递参数给链接器

在 esql 命令行中，可以通过使用 -l 处理选项先列出链接器参数。

esql 命令处理器将 -l 选项之后的所有参数传递给链接器，最多可以先到达以下任一项：

-r 选项指定资源编译器选项

命令行的结尾

向资源编译器传递参数

在 esql 命令行中，可以通过使用 **-r** 处理选项先列出资源编译器参数。

GBase 8s ESQL/C 命令处理器将 **-r** 选项之后的所有参数直到命令行的结尾传递给链接器。然后处理器运行资源编译器创建 .res 文件，然后将它传递给链接器。如果指定 **-r** 选项但是不指定其关联 *resfile*.rc，则 esql 使用目标名称并附加 .rc 扩展名。

ESQL/C 动态链接库

对于 Windows™ 环境，GBase 8s ESQL/C 产品包括以下动态链接库（DLL）：

ESQL 客户端接口 DLL（isqlt09a.dll）包含 GBase 8s ESQL/C 库函数，GBase 8s ESQL/C 预处理器需要在运行时转换嵌入式 SQL 语句和其它内部函数。

esqlauth.dll DLL 通过客户端应用程序发送到数据库服务器的连接信息的运行时验证。当您的应用程序请求连接是，GBase 8s ESQL/C 调用 sqlauth() 函数（由 esqlauth.dll 定义）。

注册表 DLL iregt07b.dll 由 Setnet32 实用程序和 GBase 8s Connect 库用于设置和访问注册表中的配置信息。

igl4b304.dll 、igo4g303.dll 和 iglxg303.dll DLL 是全球语言支持（GLS）所必需的。

GBase 8s DLL 位于 %GBASEDBTDIR%\bin 目录中。%GBASEDBTDIR% 是环境变量 GBASEDBTDIR 的值。

相同运行例程的版本独立性

如果您的应用程序是使用早于 4.x 的 Microsoft™ Visual C++ 版本编译的，则必须将 C Runtime 库导出到 ESQL 客户端接口 DLL（isqlt09a.dll）。ESQL 客户端接口 DLL 使用您运行例程确保您的应用程序的所有部分都使用相同的运行版本进行编译。链接到您应用程序并调用 GBase 8s ESQL/C 库例程或 SQL 语句的任何应用程序也必须使用 C Runtime 库。

要导出 C Runtime 库，请在首次调用 GBase 8s ESQL/C 库例程或 SQL 语句前，在代码中包含以下语句：

```
#include "infxcexp.c";
```

infxcexp.c 文件包含用于导出 ESQL 客户端接口 DLL 使用的所有 C Runtime 例程代码的地址的 C 代码。该文件在 %GBASEDBTDIR%\incl\esql 目录下，esql 命令处理器在编译时自动搜索。如果您不使用 esql 命令，请在编译之前
将 %GBASEDBTDIR%\incl\esql 目录添加到编译器搜索路径。

在程序中的第一个 GBase 8s ESQL/C 库调用或 SQL 语句之前，必须在 main() 例程（每个进程一次）中包含一次 infxcexp.c 文件。此文件中的代码将您的 runtime 库导出到 ESQL runtime DLL（isqlt09a.dll），以便它们使用相同的 C runtime 代码。导出 runtime 例程使 ESQL runtime 例程分配内存（malloc()），将指针返回给 C 程序，并让程序是否内存（free()）。它还使 C 程序能够打开文件并将句柄（或文件指针）传递到 ESQL runtime 例程以进行读/写访问。

2.2.4 在 Windows(TM) 环境中访问 ESQL 客户端接口

动态链接库（DLL）是可由应用程序共享的资源和函数的集合。它类似于 runtime 库，因为它存储许多应用程序需要的函数。然而，它与 runtime 库不同之处在于它链接到调用的应用程序。

在编译时链接的库是*静态链接库*。诸如 **libc** 和 **libcmt** 之类的库（与 Microsoft™ Visual C++ Version 2.x 一起使用)是静态链接库。 无论何时将这些 Microsoft Visual C++（版本 2.x）之一链接到应用程序，链接器将代码从适当的静态链接库复制到应用程序的可执行文件中（.exe）。相比之下，当自动链接时，不会将代码复制到应用程序的可执行文件中。相反，您的函数在运行时链接。

静态链接库在不需要多任务的环境中是有效的。但是，当多个应用程序调用相同的功能时，它们变得无效。例如：如果在 Windows™ 环境中同时运行的两个应用程序调用相同的静态链接函数，则该函数的两个副本位于内存中。这种情况是低效率的。

但是如果一个函数是动态链接的，则 Windows 系统首先检查内容，看看函数的副本是否已经存在。如果存在副本，则 Windows 系统使用此副本而不是另一个创建另一个副本。如果内存中该函数还尚未存在，则 Windows 系统将该函数从 DLL 中链接或复制到内存中。

GBase 8s ESQL/C 库函数和其它内部函数包含在 ESQL 客户端接口 DLL 中。要在您的 GBase 8s ESQL/C 函数中使用这些函数，必须执行以下操作：

访问 ESQL 客户端接口 DLL 的导入库

定位 ESQL 客户端接口 DLL

访问导入库

使用 DLL 的导入库以使您的 GBase 8s ESQL/C 应用程序访问 ESQL 客户端接口 DLL 。

链接器使用导入库定位 DLL 中包含的函数。它包含使应用程序中使用的函数名称与包含该函数的库模块协调的引用。

当将静态库链接到应用程序时，链接器将程序代码以静态链接库复制到可执行文件。但是，如果将导入库链接到应用程序，链接器在链接可执行文件时不会复制程序代码。链接器存储在 DLL 中定位函数所需的信息。运行应用程序是，此位置信息充当 DLL 的动态链接。

ESQL 客户端接口库提供 GBase 8s ESQL/C 函数调用的位置信息。当您使用它编译并链接您的 GBase 8s ESQL/C 程序时，esql 命令处理器将自动链接 DLL 的导入和 Windows™ 库。

定位 DLL

在程序开发期间，GBase 8s ESQL/C 软件（如 esql 命令处理器）必须能访问对象库并导入库。但是，在运行应用程序时 DLL 必须是可访问的。也就是说，Windows™ 必须能够在您的硬盘上找到它们。

按照以下顺序搜寻 DLL 的目录：

加载应用程序的目录

Windows 环境系统目录 **SYSTEM**

当前目录（可执行文件所在的位置或图标的程序项目属性值指定的工作目录）

PATH 环境变量列出的目录

有关特定 Windows 操作系统的最新信息，请参阅 http://www.microsoft.com 上的动态链接库搜寻顺序文档。

构建应用程序 DLL

可以告诉 GBase 8s ESQL/C 处理器使用 **-target**（或 **-wd**）命令行选项将 GBase 8s ESQL/C 程序构建为 DLL（.dll 文件）ile)。这种 GBase 8s ESQL/C 程序被称为应用程序 DLL。

要将 GBase 8s ESQL/C 程序构建为 DLL，请遵循通用 DLL 的直到。有关更多信息，请参阅您的系统文档。使用 -target:dll（或 **-wd**）编译 GBase 8s ESQL/C 源文件以创建应用程序 DLL。

有关如何构建应用程序 DLL 的信息，请参阅 %GBASEDBTDIR%\demo\wdemo 目录中的 WDEMO 演示程序。示例应用程序 DLL 的 GBase 8s ESQL/C 源文件称为 wdll.ec。要编译此 DLL，使用下面的 esql 命令：

```
esql -subsystem:windows -target:dll wdll.ec
```

WDEMO 可执行文件的源代码在 wdemo.exe 文件中。

2.3 GBase 8s ESQL/C 数据类型

这些主题包含有关 SQL 和 C 数据类型之间的对应关系以及如何处理 GBase 8s

ESQL/C 程序中数据类型的信息。

本章包括下列信息：

为主机变量选择适当的数据类型

从一种数据类型转换为另一种数据类型

使用 NULL 和不同数据类型的函数

2.3.1 为主机变量选择数据类型

当您访问 GBase 8s ESQL/C 程序中的数据库列时，必须声明适当的 C 或 GBase 8s ESQL/C 数据类型的主机变量以保存该数据。表 1 列出了 GBase 8s 的 SQL 数据类型以及您可以为主机变量声明的对应的 GBase 8s ESQL/C 数据类型。表 2 列出了 GBase 8s 可用的其它 SQL 数据类型和 GBase 8s ESQL/C 数据类型，可用作这些类型的列的主机变量。这两个表都包括对本书中的章节或章节的引用，您可以在其中获取有关主机变量数据类型之间的更多信息。

表 1. 对应 SQL 的主机变量数据类型

SQL 数据类型	ESQL/C 预定义的数据类型	C 语言类型	请参阅
BIGINT	**BIGINT**	8 字节整数	Numeric 数据类型
BIGSERIAL	**BIGINT**	8 字节整数	Numeric 数据类型
BOOLEAN	**boolean**		表 2
BYTE	**ifx_loc_t** or **loc_t**		简单大对象
CHAR(*n*) CHARACTER(*n*)	**fixchar [*n*]** or **string [*n*+1]**	char [*n* + 1] 或 **char ***	字符和字符串数据类型
DATE	**date**	4 字节整数	时间数据类型
DATETIME	**datetime** or **dtime_t**		时间数据类型

DECIMAL DEC NUMERIC MONEY	**decimal** or **dec_t**		Numeric 数据类型
FLOAT DOUBLE PRECISION		**double**	时间数据类型
INT8	**int8** or **ifx_int8_t**		Numeric 数据类型
INTEGER INT		4 字节整数	Numeric 数据类型
INTERVAL	**interval** or **intrvl_t**		时间数据类型
LVARCHAR	**lvarchar**	char [*n* + 1] 或 **char ***	字符和字符串数据类型
NCHAR(*n*)	**fixchar [*n*]** or **string [*n*+1]**	char [*n* + 1]或 **char ***	字符和字符串数据类型
NVARCHAR(*m*)	**varchar[*m*+1]** or **string [*m*+1]**	char [*m*+1]	字符和字符串数据类型
SERIAL		4 字节整数	Numeric 数据类型
SERIAL8	**int8** or **ifx_int8_t**		Numeric 数据类型
SMALLFLOAT REAL		**float**	Numeric 数据类型
SMALLINT		2 字节整数	Numeric 数据类型

TEXT	**loc_t**		简单大对象
VARCHAR(*m,x*)	**varchar[*m*+1]** or **string [*m*+1]**	char d[*m*+1]	字符和字符串数据类型

表 2. 对应于 GBase 8s 特定的 SQL 和主机变量数据类型

SQL 数据类型	ESQL/C 预定义类型	请参阅
BLOB	**ifx_lo_t**	智能大对象
CLOB	**ifx_lo_t**	智能大对象
LIST(*e*)	**collection**	智能大对象
MULTISET(*e*)	**collection**	复杂数据类型
Opaque data type	**lvarchar**, **fixed binary**, or **var binary**	不透明数据类型
ROW(...)	**row**	复杂数据类型
SET(*e*)	**collection**	复杂数据类型

数据类型常量

GBase 8s ESQL/C sqltypes.h 头文件常量包含 SQL 和 GBase 8s ESQL/C 数据类型。一些 GBase 8s ESQL/C 库函数要求数据类型常量作为参数。还可以在动态 SQL 程序中比较这些数据类型常量，以确定描述 DESCRIBE 语句的列的类型。下图中的 GBase 8s ESQL/C 代码摘录将 **sqlvar** 的 **sqltype** 元素与一系列 SQL 数据类型常量进行比较，以确定 DESCRIBE 语句返回的列的类型。

图: 带有 SQL 数据类型常量的代码摘录

```
for (col = udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
                {
                switch(col->sqltype)
                {
                case SQLSMFLOAT:
                col->sqltype = CFLOATTYPE;
```

```
                break;

                case SQLFLOAT:
                col->sqltype = CDOUBLETYPE;
                break;

                case SQLMONEY:
                case SQLDECIMAL:
                col->sqltype = CDECIMALTYPE;
                break;

                case SQLCHAR:
                col->sqltype = CCHARTYPE;
                break;

                default:
                /*   The program does not handle INTEGER,
                *   SMALL INTEGER, DATE, SERIAL or other
                *   data types.   Do nothing if we see
                *   an unsupported type.
                */
                return;
                }
```

SQL 数据类型常量

表 1 显示了 GBase 8s 的 SQL 数据类型常量。表 2 显示了其它可用于 GBase 8s 的数据类型的 SQL 数据类型常量。

表 1. GBase 8s SQL 类数据类型的常量

SQL 数据类型	定义的常量	整数值
CHAR	SQLCHAR	0
SMALLINT	SQLSMINT	1
INTEGER	SQLINT	2
FLOAT	SQLFLOAT	3
SMALLFLOAT	SQLSMFLOAT	4
DECIMAL	SQLDECIMAL	5
SERIAL	SQLSERIAL	6
DATE	SQLDATE	7

MONEY	SQLMONEY	8
DATETIME	SQLDTIME	10
BYTE	SQLBYTES	11
TEXT	SQLTEXT	12
VARCHAR	SQLVCHAR	13
INTERVAL	SQLINTERVAL	14
NCHAR	SQLNCHAR	15
NVARCHAR	SQLNVCHAR	16
INT8	SQLINT8	17
BIGSERIAL	SQLBIGSERIAL	53
LVARCHAR	SQLLVARCHAR	43
BOOLEAN	SQLBOOL	45
BIGINT	SQLINFXBIGINT	52
BIGSERIAL	SQLBIGSERIAL	53

表 2. 特定于 GBase 8s 的 GBase 8s SQL 列数据类型的常量

SQL 数据类型	定义的常量	整数值
SET	SQLSET	19
MULTISET	SQLMULTISET	20
LIST	SQLLIST	21
ROW	SQLROW	22
可变长度的不透明类型	SQLUDTVAR	40
固定长度的不透明类型	SQLUDTFIXED	41

SENDRECV（仅限于客户端）	SQLSENDRECV	44

重要： SENDRECV 数据类型具有 SQL 常量但是只能在 GBase 8s ESQL/C 程序中使用。不能将数据库列定义为类型 SENDRECV。

ESQL/C 数据类型常量

在 GBase 8s ESQL/C 中给主机变量指定 GBase 8s ESQL/C 数据类型。下表显示了这些常量。

表 1. ESQL/C 主机变量数据类型的常量

ESQL/C 数据类型	常量	整数值
char	CCHARTYPE	100
short int	CSHORTTYPE	101
int4	CINTTYPE	102
long	CLONGTYPE	103
float	CFLOATTYPE	104
double	CDOUBLETYPE	105
dec_t 或 **decimal**	CDECIMALTYPE	107
fixchar	CFIXCHARTYPE	108
string	CSTRINGTYPE	109
date	CDATETYPE	110
dec_t 或 **decimal**	CMONEYTYPE	111
datetime 或 **dtime_t**	CDTIMETYPE	112
ifx_loc_t 或 **loc_t**	CLOCATORTYPE	113
varchar	CVCHARTYPE	114
intrvl_t 或 **interval**	CINVTYPE	115

char	CFILETYPE	116
int8	CINT8tYPE	117
collection	CCOLTYPE	118
lvarchar	CLVCHARTYPE	119
fixed binary	CFIXBINTYPE	120
var binary	CVARBINTYPE	121
boolean	CBOOLTYPE	122
row	CROWTYPE	123

可以使用这些 GBase 8s ESQL/C 数据类型作为 GBase 8s ESQL/C 库中某些函数的参数的数据类型。例如：rtypalign() 和 rtypmsize() 函数都要求数据类型值作为参数。

X/Open 数据类型常量

如果您的程序符合 X/Open 标准（使用 **-xopen** 选项编译），那么必须使用下表显示的数据类型值。GBase 8s 为 sqlxtype.h 头文件中的这些值定义常量。

表 1. X/Open 环境中 GBase 8s SQL 列数据类型的常量

SQL 数据类型	定义的常量	X/Open 整数值
CHAR	XSQLCHAR	1
DECIMAL	XSQLDECIMAL	3
INTEGER	XSQLINT	4
SMALLINT	XSQLSMINT	5
FLOAT	XSQLFLOAT	6

头文件的数据类型

要使用 SQL 数据类型，您的程序必须包括适当的 GBase 8s ESQL/C 头文件。表 1 显示了使用数据库服务器主机变量和 GBase 8s ESQL/C 头文件之间的关系。表 2显示了主机变量数据类型和特定于 GBase 8s Universal Data Option 的 GBase 8s ESQL/C 头文件之间的关系。

表 1. SQL 数据类型和 ESQL/C 头文件

SQL 数据类型	ESQL/C 或 C 数据类型	ESQL/C 头文件
BLOB	**ifx_lo_t**	locator.h
BOOLEAN	**boolean**	自动定义
BYTE	**ifx_loc_t** 或 **loc_t**	locator.h
CHAR(*n*) CHARACTER(*n*)	**fixchar array[*n*]** 或 **string array[*n*+1]**	自动定义
DATE	**date**	自动定义
DATETIME	**datetime** 或 **dtime_t**	datetime.h
DECIMAL DEC NUMERIC MONEY	**decimal** 或 **dec_t**	decimal.h
FLOAT DOUBLE PRECISION	**double**	自动定义
INT8	**int8**	int8.h
INTEGER INT	4 字节整数	自动定义
INTERVAL	**interval** or **intrvl_t**	datetime.h
LVARCHAR	**lvarchar array[*n* + 1]**（ *n* 是可能存储在 LVARCHAR 自动中的长字符串的长度）	自动定义
MULTISET(e)	**collection**	自动定义
NCHAR(*n*)	**fixchar array[*n*]** or **string array[*n*+1]**	自动定义
NVARCHAR(*m*)	**varchar[*m*+1]** or **string**	自动定义

	array[*m*+1]	
SERIAL	4 字节整数	自动定义
SERIAL8	**int8**	int8.h
BIGINT	BIGINT	自动定义
BIGSERIAL	BIGINT	自动定义
SMALLFLOAT REAL	**float**	自动定义
SMALLINT	**short int**	自动定义
TEXT	**loc_t**	locator.h
VARCHAR(*m*,*x*)	**varchar[*m*+1]** 或 **string array[*m*+1]**	自动定义

表 2. 特定于 GBase 8s 的 SQL 数据类型和 ESQL/C 头文件

SQL 数据类型	ESQL/C 或 C 数据类型	ESQL/C 头文件
BLOB	**ifx_lo_t**	locator.h
CLOB	**ifx_lo_t**	locator.h
LIST(*e*)	**collection**	自动定义
Opaque 数据类型	**lvarchar** 或 **fixed binary** 或 **var binary**	包含内部结构或不透明类型定义的常量的用户定义的头文件
ROW(...)	**row**	自动定义
SET(e)	**collection**	自动定义

2.3.2 数据转换

当两个值的数据类型之间存在差异时，GBase 8s ESQL/C 将尝试转换其中一种数据类型。将值从一种数据类型转换为另一种数据类型的过程称为数据转换。

以下列表列出了可能发生数据转换的几种常见情况：

比较

如果您使用比较两种不同类型值的条件，例如将 zip 代码列的内容与整数值进行比较，则可能发生数据转换。

例如：为了比较 CHAR 值和数字值，GBase 8s ESQL/C在执行此比较之前将 CHAR 值转换为数字值。

抓取和插入

如果您使用不同数据类型的主机变量和数据库列来获取或插入值，则可能会发生数据转换。

算术运算

如果一种数据类型的数值对不同数据类型的值进行操作，则可能会发生数据转换。

使用主机变量获取和插入

如果您尝试从数据库列中将值从根据表 1中所示的对应关系声明的主机变量中获取，则 GBase 8s ESQL/C 将尝试转换数据类型。同样，如果您尝试将主机变量的值插入到数据库列中，如果主机变量和数据库列不使用表 1中的对应关系，则 GBase 8s ESQL/C 可能需要转换数据类型。GBase 8s ESQL/C 将转换只有转换有意义的数据类型。

转换数字和字符串

在 GBase 8s ESQL/C 将从一种数据类型的值转换为另一种时，它必须确定该转换是否有意义。

下表显示了数字数据类型和字符数据类型之间可能的转换。在此表中，N 代表具有数字数据类型（例如 DECIMAL 、FLOAT 或 SMALLINT）的值，C 代表具有字符数据类型（例如 CHAR 或 VARCHAR）的值。

如果转换不可能，可能因为它无意义或者目标变量太小以致于无法接收转换的值，GBase 8s ESQL/C 返回下表中**结果**列返回的值。

表 1. 数据转换的问题和结果

转换	问题	结果
C C	不合适	GBase 8s ESQL/C 截断字符串，设置警告（将 **sqlca.sqlwarn.sqlwarn1** 设置为 W ，SQLSTATE 设置为 01004），并且将任何指示变量设置为原始字符串的大小。
N C	无	GBase 8s ESQL/C 为数字值创建字符串；它使用大或小的指数格式。

N C	不合适	GBase 8s ESQL/C 用星号填充字符串，设置警告 （将 **sqlca.sqlwarn.sqlwarn1** 设置为 W ，SQLSTATE 设置为 01004），并将任何指示变量设置为正整数。 当数字的小数部分不适合字符变量时，GBase 8s ESQL/C 舍入数字。仅当整数部分不合适时才会显示星号。
C N	无	GBase 8s ESQL/C 根据字符值的格式的确定数字数据类型；如果字符包含小数点，则 GBase 8s ESQL/C 将该值转换为 DECIMAL 值。
C N	不是数字	数字未定义；GBase 8s ESQL/C 设置 **sqlca.sqlcode** 和 SQLSTATE 来指示运行时的错误。
C N	溢出	数字未定义；GBase 8s ESQL/C 设置 **sqlca.sqlcode** 和 SQLSTATE 来指示运行时的错误。
N N	不合适	GBase 8s ESQL/C 尝试将此数字转换为新的数据类型。
N N	溢出	数字未定义；GBase 8s ESQL/C 设置 **sqlca.sqlcode** 和 SQLSTATE 来指示运行时的错误。

表 1中，*不合适*自动表示源变量或列的数据的大小或超出了目标变量或列的大小。

将浮点数转换为字符串

GBase 8s ESQL/C 可以自动转换数据库列和字符类型为 **char** 、**varchar** 、**string** 或 **fixchar** 的主机变量之间的浮点列值（DECIMAL(*n*) 、FLOAT 或 SMALLFLOAT 数据类型）。当 GBase 8s ESQL/C 将浮点值转换为缓冲区不够大以容纳完整精度的字符串时，GBase 8s ESQL/C 将该值舍入到字符缓冲区中。

将 BOOLEAN 值转换为字符

数据库服务器可以自动转换数据库列和 **fixchar** 数据类型的主机变量之间的 BOOLEAN 值。

以下列出了 BOOLEAN 值的表示字符：

'\01'

'T'

'\00'

'F'

转换 DATETIME 和 INTERVAL 值

GBase 8s ESQL/C 可以自动转换数据库列和字符类型为 **char** 、**string** 或 **fixchar**

的主机变量之间的 DATETIME 和 INTERVAL 值。GBase 8s ESQL/C 将 DATETIME 或 INTERVAL 值转换为字符字符串并将它存储在主机变量中。

可以使用 GBase 8s ESQL/C 库函数显式转换 DATE 和 DATETIME 值。

在 VARCHAR 和字符数据类型之间进行转换

GBase 8s ESQL/C 可以自动转换数据库列和字符类型为 **char** 、**string** 或 **fixchar** 的主机变量之间的 VARCHAR 值。

执行计算操作

当 GBase 8s ESQL/C 在两个值之间执行算术操作时，如果两个值不具有匹配的数据类型，则它可能需要转换数据类型。

本节提供了用于算术操作的数据转换的以下信息：

GBase 8s ESQL/C 如何转换数字值

GBase 8s ESQL/C 处理调用浮点值的操作

将数字转换为数字

如果两个不同的数值数据类型的值彼此操作，则 GBase 8s ESQL/C 将值转换为下表所指示的数据类型，然后执行该操作。

表 1. ESQL/C 执行数字操作的数据类型

操作数	DEC	FLOAT	INT	SERIAL	SMALLFLOAT	SMALLINT
DEC	DEC	DEC	DEC	DEC	DEC	DEC
FLOAT	DEC	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
INT	DEC	FLOAT	INT	INT	FLOAT	INT
SERIAL	DEC	FLOAT	INT	INT	FLOAT	INT
SMALLFLOAT	DEC	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
SMALLINT	DEC	FLOAT	INT	INT	FLOAT	INT

表 1显示，如果 GBase 8s ESQL/C 在数据类型为 FLOAT 的操作数与数据类型为 DECIMAL（DEC）的第二个操作数之间执行操作，GBase 8s ESQL/C 将生成具有 DECIMAL

数据类型的结果。

调用小数值的操作

下表显示了数值数据类型。数据库列使用 SQL 数据类型，GBase 8s ESQL/C 主机变量使用对应的 GBase 8s ESQL/C 数据类型。

SQL 数据类型	ESQL/C 数据类型
INTEGER	4 字节整数
SMALLINT	short integer
DECIMAL	十进制
MONEY	十进制
FLOAT	double
SMALLFLOAT	float

当 GBase 8s ESQL/C 对具有数字数据类型的操作执行算术运算，并且其中一个操作数具有十进制值（ SQL 数据类型为 DECIMAL 或 GBase 8s ESQL/C 数据类型为 **decimal**），GBase 8s ESQL/C 将每个操作数和结果转换为十进制值。

SQL DECIMAL 数据类型具有格式 DECIMAL(*p*,*s*)，其中 *p* 和 *s* 表示以下参数：

p 参数是精度，它是实数中有效数字的总数。

例如：1237.354 的精度为 7。

s 参数是小数位，它是表示实数小数部分的位数。

例如：1237.354 的小数位是 3。如果 DECIMAL 数据类型包含小数位参数（(DECIMAL(*p*,*s*)）,则它保存定点小数。如果 DECIMAL 数据类型忽略了小数位参数（DECIMAL(*p*)），则它保存浮点小数。

GBase 8s ESQL/C **decimal** 数据类型跟踪 SQL DECIMAL 数据类型的精度和小数位。为了简单起见，本节使用 SQL DECIMAL 数据类型的格式描述 GBase 8s ESQL/C 如何对涉及小数的算术运算执行数据转换。但是，相同的数据转换信息适用于涉及 GBase 8s ESQL/C **decimal** 主机变量的算术运算。

转换非十进制数值操作

在执行算术运算之前，GBase 8s ESQL/C 将所有尚未 DECIMAL（或 **decimal**）的操作数转换为 DECIMAL 。

以下列表显示 GBase 8s ESQL/C 用于非 DECIMAL 操作数的精度和小数位。

操作数类型

转换为

FLOAT

DECIMAL(17)

SMALLFLOAT

DECIMAL(9)

INTEGER

DECIMAL(10,0)

SMALLINT

DECIMAL(5,0)

GBase 8s ESQL/C 不考虑前导或尾随零作为有效数字。前导或尾随零无助于确定精度和小数位。如果操作时进行加法或减法，则 GBase 8s ESQL/C 会将较少小数位的尾随零添加到操作数，直至小数位相等。

获得算术结果的 DECIMAL 数据类型

算术结果的精度和小数位取决于操作数的精度和小数位，以及操作数之一是否为浮点小数，如下所示：

当其中之一的操作数位浮点小数，则算术结果为浮点小数。

例如：对于在定点小数 DECIMAL(8,3) 和 FLOAT 值之间的算术，GBase 8s ESQL/C 将 FLOAT 值转换为浮点小数 DECIMAL(17)。算术结果具有 DECIMAL(17) 数据类型。

当这两个操作数都是定点小数，则算术结果也是定点小数。

下表总结了具有刻度（定点小数）的操作数的算术运算规则。在下表中，*p*1 和 *s*1 是第一个操作数的精度和小数位，*p*2 和 *s*2 是第二个操作数的精度和小数位。

表 1. 定点算术结果的精度和小数位

操作	结果的精度和小数位	
加法和减法	精度： 小数位：	MIN(32, MAX(*p*1 - *s*1, *p*2 - *s*2) + MAX(*s*1, *s*2) + 1) MAX(*s*1, *s*2)
乘法	精度： 小数位：	MIN(32, *p*1 + *p*2) *s*1 + *s*2; 如果 (*s*1 + *s*2) > 精度，则该结果是一个浮点小数（不具有小数位）。
除法	精度： 小数位：	32

		结果为浮点小数。 和：32 - *p*1 + *s*1 - *s*2 不能为负数。

如果算术操作的结果的数据类型需要有效数字的丢失，则 GBase 8s ESQL/C 将报告错误。

2.3.3 数据类型对齐库函数

以下 GBase 8s ESQL/C 库函数为不同的数据类型提供与机器无关的大小和对齐信息，并帮助您处理空数据库值。

函数名称	描述	请参阅
risnull()	检查 C 变量是否为 null	risnull() 函数
rsetnull()	将 C 变量设置为 null	rsetnull() 函数
rtypalign()	对准正确类型边界上的数据	rtypalign() 函数
rtypmsize()	给出 SQL 数据类型的字节的大小	rtypmsize() 函数
rtypname()	返回指定的 SQL 数据类型的名称	rtypname() 函数
rtypwidth()	返回字符数据类型需要避免截断的最小字符数	rtypwidth() 函数

当使用 esql 命令编译 GBase 8s ESQL/C 程序时，esql 调用链接器或链接这些函数到您的程序。

2.4 字符和字符串数据类型

这些主题介绍了如何在 GBase 8s ESQL/C 程序中使用字符数据类型。

2.4.1 字符数据类型

GBase 8s ESQL/C 支持五种数据类型，可以容纳从数据库检索和发送到数据库的字符数据。

如果对数据库列使用字符数据类型（例如 SQL 数据类型 CHAR 和 VARCHAR），那么您可以为您的主机变量选择以下之一的数据类型：

C 字符数据类型：char

其中之一的 GBase 8s ESQL/C 预定义的数据类型：fixchar 、string 、varchar

lvarchar 数据类型

如果您使用区域设置敏感字符数据类型（NCHAR 或 NVARCHAR），那么可以选择相关主机变量的字符数据类型。

以下两个条件决定要使用的字符数据类型：

是否要使 GBase 8s ESQL/C 使用空字符终止字符数据

是否需要 GBase 8s ESQL/C 来填充带有尾部空格的字符数据

下表总结了每个字符数据类型的属性。

表 1. ESQL/C 字符数据类型

ESQL/C 字符数据类型	Null 终止	包含尾部空格
char	Y	Y
fixchar		Y
string	Y	只当列包含空字符串时才返回尾部空格
varchar	Y	Y
lvarchar	Y	

char 数据类型

char 数据类型是持有字符数据的 C 数据类型。

当应用程序将 CHAR 列的值读取为 **char** 类型的主机变量时，GBase 8s ESQL/C 将此值附加到最大大小为主变量的尾部空格。它经终止主机数组的空字符留下一个位置。如果应用程序从 VARCHAR （或 NVARCHAR）列读取一个值到 **char** 数据类型的主机变量，则做出同样的操作。

声明长度为 [*n* + 1]（其中 *n* 是您要读取的值的列的大小）的 **char** 数据类型，以允许空终止符。使用以下语法声明 **char** 数据类型的主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        char ch_name[n + 1];
        EXEC SQL END DECLARE SECTION;
```

fixchar 数据类型

fixchar 数据类型是持有不附加空终止符的字符数据的 GBase 8s ESQL/C 数据类型。

当应用程序将 CHAR 列的值读取为 **fixchar** 类型的主机变量时，GBase 8s ESQL/C 将此值附加到最大大小为主变量的尾部空格。GBase 8s ESQL/C 不会附加任何空字符。如果应用程序从 VARCHAR （或 NVARCHAR）列读取一个值到 **fixchar** 数据类型的主机变量，则做出同样的操作。

限制： 不要使用 VARCHAR 或 NVARCHAR 数据的 **fixchar** 数据类型。使用 **fixchar**，即使数据的长度小于 **fixchar** 的大小，数据库服务器也会存储 **fixchar** 的所有 n 个字符，包括字符串末尾的任何空格。除非空字符有意义，否则存储它们将使 VARCHAR 数据类型提供的空间节省失效。

将 **fixchar** 主机变量声明为具有 *n* 个组件的数组（其中 *n* 是您要读取的值的列的大小）。使用以下语法声明 **fixchar** 数据类型的主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        fixchar fch_name[n];
        EXEC SQL END DECLARE SECTION;
```

重要： 如果空字符可用于 null 字符，则可以将 null 值终止的 C 字符串复制到 **fixchar** 变量中。但是，这不是很好的做法。当数据库服务器将此值插入列时，它还会插入空终止符。因此，稍后搜索表可能无法找到该值。

string 数据类型

string 数据类型是持有 null 终止符但不包含尾部空格的 GBase 8s ESQL/C 数据类型。

但是，如果一个空格（即 ‘ ’）的字符串存储在数据库字段中并被选择到 **string** 数据类型的主机变量中，那么结果将是一个空白字符。

当应用程序将 CHAR 列的值读取为 **string** 类型的主机变量时，它将剥离任何尾部空格的值，并附加一个 null 终止符。如果应用程序从 VARCHAR （或 NVARCHAR）列读取一个值到 **fixchar** 数据类型的主机变量，则做出同样的操作。

此规则的一个例外是如果 BLANK_STRINGS_NOT_NULL 环境变量设置为 1 或其它值（比如 0 或 2），那么字符串主机变量将空字符串存储为单个空格，后跟一个 null 终止符。如果未设置此环境变量，则字符串主机变量将空字符串存储为 null 字符串。

```
EXEC SQL BEGIN DECLARE SECTION;
        string buffer[16];
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL select lname into :buffer from customer
        where customer_num = 102;
```

声明长度为 [*n* + 1]（其中 *n* 是您要读取的值的列的大小）的 **string** 数据类型，以允许 null 终止符。 在上面的代码中，**customer** 表的 **lname** 列是 15 字节，因此 **buffer** 主机变量声明为 16 字节。使用以下语法声明 **string** 数据类型的主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        string str_name[n + 1];
        EXEC SQL END DECLARE SECTION;
```

varchar 数据类型

varchar 数据类型是持有可变长度的字符数据的 GBase 8s ESQL/C 数据类型。

当应用程序将 CHAR 列的值读取为 **varchar** 类型的主机变量时，GBase 8s ESQL/C 将保留任何尾部空白并以 NULL 字符终止此数组。如果应用程序从 VARCHAR 列读取一个值到 **varchar** 数据类型的主机变量，则做出同样的操作。

声明长度为 [*n* + 1]（其中 *n* 是您要读取的值的列的大小）的 **varchar** 数据类型，以允许 null 终止符。使用以下语法声明 **varchar** 数据类型的主机变量：

EXEC SQL BEGIN DECLARE SECTION;

varchar *varc_name[n + 1]*;

EXEC SQL END DECLARE SECTION;

VARCHAR 大小宏

GBase 8s 包含具有 GBase 8s ESQL/C u库的 varchar.h 头文件、该文件定义下表中显示的宏函数和名称。

表 2. VARCHAR 大小宏

宏的名称	描述
MAXVCLEN	您可以在 VARCHAR 列存储的最大字符数。该值为 255。
VCLENGTH(s)	要声明的主机变量的长度
VCMIN(s)	您可以在 VARCHAR 列存储的最小字符数。范围从 1 到 255 字节，但必须小于 VARCHAR 的最大大小
VCMAX(s)	您可以在 VARCHAR 列存储的最大字符数。范围可以是1 到 255 字节。
VCSIZ(min, max)	基于 *min* 和 *max* 的编码大小值，用于 VARCHAR 列。

当使用动态 SQL 时，这些宏很有用。在 DESCRIBE 语句之后，宏可以处理数据库服务器存储在系统描述符区域(或 **sqlda** 的 **sqllen** 字段)的 LENGTH 字段中的大小信息。数据库服务器在 **syscolumns** 系统目录表中存储 VARCHAR 列的大小信息、

varchar.ec 演示程序

varchar.ec 演示程序从 syscolumns 系统目录表中获取 cat_advert 列（stores7 数据库）的 collength 。然后它使用 varchar.h 中的宏来显示关于列的大小信息。该示例程序位于 demo 目录中的 varchar.ec 文件中。下图显示了 varchar.ec 演示程序的 main() 函数。

图 1. varchar.ec 演示程序

```
/*
        * varchar.ec *

        The following program illustrates the use of VARCHAR macros to
```

```
obtain size information.
*/

EXEC SQL include varchar;

char errmsg[512];

main()
{
mint vc_code;
mint max, min;
mint hv_length;

EXEC SQL BEGIN DECLARE SECTION;
mint vc_size;
EXEC SQL END DECLARE SECTION;

printf("VARCHAR Sample ESQL Program running.\n\n");

EXEC SQL connect to 'stores7';
chk_sqlcode("CONNECT");

printf("VARCHAR field 'cat_advert':\n");
EXEC SQL select collength into $vc_size from syscolumns
where colname = "cat_advert";
chk_sqlcode("SELECT");
printf("\tEncoded size of VARCHAR (from syscolumns.collength) = %d\n",
vc_size);

max = VCMAX(vc_size);
printf("\tMaximum number of characters = %d\n", max);

min = VCMIN(vc_size);
```

```
        printf("\tMinimum number of characters = %d\n", min);

        hv_length = VCLENGTH(vc_size);
        printf("\tLength to declare host variable = char(%d)\n", hv_length);

        vc_code = VCSIZ(max, min);
        printf("\tEncoded size of VARCHAR (from VCSIZ macro) = %d\n",
vc_code);

        printf("\nVARCHAR Sample Program over.\n\n");
        }
```

当 IFX_PAD_VARCHAR 环境变量设置为 1 时，客户端发送具有附加的尾部空格的 VARCHAR 数据类型、当未设置此环境变量（缺省），则客户端范式不带尾部空格的 VARCHAR 数据类型。

lvarchar 数据类型

lvarchar 数据类型是持有可变长度的字符数据的 GBase 8s ESQL/C 数据类型。

lvarchar 数据类型它实现类似于 **varchar** 数据类型的可变长度用户定义类型，但它可以支持大于 256 个字节的字符串，并具有以下两个用途：

保存数据库中 LVARCHAR 列的值。

当应用程序从 LVARCHAR 列读取值到 **lvarchar** 数据类型的主机变量时，GBase 8s ESQL/C 将保留任何尾随空格，并以 null 终止符终止数组。如果应用程序从 VARCHAR 列读取一个值到 **lvarchar** 数据类型的主机变量，则做出同样的操作。

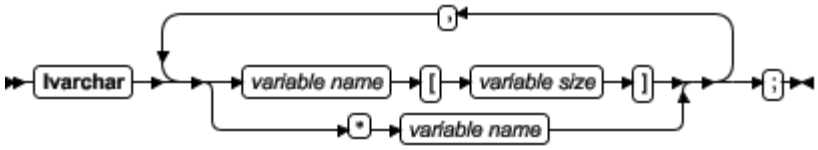
表示不透明数据类型的字符串或外部格式。

重要： 您无法从 **lvarchar** 主机变量检索或存储智能大对象（CLOB 或 BLOB 数据类型）。

lvarchar 关键字语法

要为字符列（CHAR 、VARCHAR 或 LVARCHAR）声明 lvarchar 主机变量。请使用 lvarchar 关键字作为变量数据类型。

以下语法显示了 **lvarchar** 关键字作为变量数据类型。



元素	意义	限制
variable name	指定大小的 **lvarchar** 变量的名称	无
variable size	为指定大小的 **lvarchar** 变量分配的字节数	可以是 1 - 32,768（32 KB）之间的整数值。
*variable name	未指定长度的 **lvarchar** 指针变量的名称	不等同于 C char 指针（char *）。指向此类型的内部 ESQL/C 表示法。必须使用 ifx_var() 函数操纵数据。

下图显示持有 LVARCHAR 列的值的三个 lvarchar 变量的声明。图 2. lvarchar 主机变量示例

```
EXEC SQL BEGIN DECLARE SECTION;
        lvarchar *a_polygon;
        lvarchar circle1[CIRCLESZ],        circle2[CIRCLESZ];
        EXEC SQL END DECLARE SECTION;
```

重要： 要声明不透明数据类型的外部格式 lvarchar 主机变量，使用声明 lvarchar 主机变量中描述的语法。

固定大小的 lvarchar 主机变量

如果未指定 **lvarchar** 主机变量的大小，那么其大小等于一种 C 语言 **char** 数据类型。如果指定了大小，则 **lvarchar** 主机变量等价于具有此大小的 C 语言 **char** 数据类型。当您指定固定大小的 **lvarchar** 主机变量时，获取该列时，任何超出指定大小的数据都会被截断。使用指示变量检查此截断。

因为已知大小的 **lvarchar** 主机变量等价于 C 语言 **char** 数据类型，所以可以使用 C 语言字符字符串操作操纵数据。

lvarchar 指针主机变量

当 **lvarchar** 主机变量是一个指针时，指针引用的数据的大小可以扩展到 2。**lvarchar** 指针主机变量旨在插入或选择可以以字符串格式表示的用户定义或不透明类型。

必须使用 ifx_var() 函数操纵 **lvarchar** 指针主机变量。

2.4.2 获取并插入字符数据类型

您可以使用以下任一操作在 CHAR 和 VARCHAR 列以及字符（**char** 、**string** 、**fixchar** 、**varchar** 或 **lvarchar**）主机变量之间传输字符数据：

提取操作将字符数据从 CHAR 或 VARCHAR 列传输到字符主机变量。

插入或更新操作将字符主机变量传输到 CHAR 、VARCHAR 或 LVARCHAR 列。

如果使用区域设置敏感字符数据类型（NCHAR 或 NVARCHAR），则还可以在 NCHAR 或 NVARCHAR 列和字符主机变量之间传输字符数据。

获取并插入 CHAR 数据

当应用程序使用字符主机变量插入或获取 CHAR 值时，GBase 8s ESQL/C 必须保证字符值适合主机变量或数据库列。

获取 CHAR 数据

应用程序可以从 CHAR 或 VARCHAR 类型的数据库列获取数据到字符（**char** 、**string** 、**fixchar** 、**varchar** 或 **lvarchar**）主机变量中。如果列数据不符合字符主机变量，则 GBase 8s ESQL/C 会截断此数据。要通知用户截断，GBase 8s ESQL/C 执行下列操作：

将 sqlca.sqlwarn.sqlwarn1 警告标记设置为 W 并将 SQLSTATE 变量设置为 01004。

它将与字符主机变量关联的任何指示变量设置为列中的字符数据的大小。

插入 CHAR 数据

应用程序可以将字符主机变量（**char** 、**string** 、**fixchar** 、**varchar** 或 **lvarchar**）中的数据插入到 CHAR 类型的数据库列。如果该值小于数据库列的大小，则数据库服务器将使用空格填充到该列大小的值。

如果值大于列的大小，则数据库服务器截断值（如果数据库为非 ANSI）。发生截断时不会产生警告。如果数据库是 ANSI 并且值大于列的长度，则插入失败并返回此错误：

-1279: Value exceeds string column length.

尽管 **char** 、**varchar** 、**lvarchar** 和 **string** 主机变量包含空终止符，GBase 8s ESQL/C 不会将这些字符插入到数据库列。（**fixchar** 类型的主机变量绝对不能包含空终止符。）

如果使用区域设置敏感的数据类型（ NCHAR），则可以将字符主机变量中的值插入 NCHAR 列。插入 NCHAR 列的方式与插入到 CHAR 列的行为相同。

不要将 **fixchar** 数据类型用于将字符数据插入到符合 ANSI 的数据库中的主机变量。

获取并插入 VARCHAR 数据

当应用程序使用字符主机变量插入或获取 VARCHAR 值时，GBase 8s ESQL/C 必须确保字符值适合主机变量或数据库列。当 GBase 8s ESQL/C 计算源条目的长度时，它不计算尾部的空格。以下部分将介绍 GBase 8s ESQL/C 如何执行将 VARCHAR 数据转换为 **char** 、**fixchar** 和 **string** 字符数据类型。

这些转换也适用于 NVARCHAR 数据。

获取 VARCHAR 数据

下表显示了当应用程序将其读取为 **char** 、**fixchar** 、**lvarchar** 和 **string** 和字符串数据类型的主机变量时，VARCHAR 数据的转换。

表 3. 将 VARCHAR 数据类型转换为 ESQL/C 字符数据类型

源类型	目标类型	结果
VARCHAR	char	如果源类型比较长，则截断并使用 null 终止，并设置任何指示变量。如果目标类型更长，则用尾部空格填充该值，并将其终止。
VARCHAR	fixchar	如果源类型比较长，则截断该值并设置任何指示变量。如果目标类型更长，则用尾部空格填充该值。
VARCHAR	string	如果源类型比较长，则截断该值并使用 null 终止，并设置任何指示变量。如果目标类型更长，则使用 null 终止该值。
VARCHAR	lvarchar	如果源类型比较长，则截断该值并设置任何指示变量。如果目标类型更长，则 null 终止该值。

下面的表显示了在获取期间 GBase 8s ESQL/C 可能执行的从 VARCHAR 列数据到字符主机变量的转换示例。在该表中，加号（+）表示空格字符，**长度**列中的值包括任何 null 终止符。

表 4. 获取期间的 VARCHAR 转换示例

源类型	内容	长度	目标类型	内容	指示符
VARCHAR(9)	Fairfield	9	char(5)	Fair\0	9
VARCHAR(9)	Fairfield	9	char(12)	Fairfield++\0	0
VARCHAR(	Fairfield+	12	char(10)	Fairfield\0	12

源类型	内容	长度	目标类型	内容	指示符
12)	++				
VARCHAR(10)	Fairfield+	10	char(4)	Fai\0	10
VARCHAR(11)	Fairfield++	11	char(14)	Fairfield++++\0	0
VARCHAR(9)	Fairfield	9	fixchar(5)	Fairf	9
VARCHAR(9)	Fairfield	9	fixchar(10)	Fairfield+	0
VARCHAR(10)	Fairfield+	10	fixchar(9)	Fairfield	10
VARCHAR(10)	Fairfield+	10	fixchar(6)	Fairfi	10
VARCHAR(10)	Fairfield+	10	fixchar(11)	Fairfield++	0
VARCHAR(9)	Fairfield	9	string(4)	Fai\0	9
VARCHAR(9)	Fairfield	9	string(12)	Fairfield\0	0
VARCHAR(12)	Fairfield++	12	string(10)	Fairfield\0	12
VARCHAR(11)	Fairfield++	11	string(6)	Fairf\0	11
VARCHAR(10)	Fairfield++	10	string(11)	Fairfield\0	0
VARCHAR(10)	Fairfield+	10	lvarchar(11)	Fairfield+	0
VARCHAR(9)	Fairfield	9	lvarchar(5)	Fair\0	9

插入 VARCHAR 数据

当应用程序将 **char** 、**varchar** 、**lvarchar** 和 **string** 主机变量的值插入到 VARCHAR 列时，GBase 8s ESQL/C 还会插入任何尾随空格。但是，GBase 8s ESQL/C 则不会添加尾随空格。

如果该值大于数据库列的最大大小，则数据库服务器截断该值（如果数据库为非

ANSI）。发生截断时不会产生警告。如果数据库是 ANSI 并且值大于列的长度，则插入失败并返回此错误：

-1279: Value exceeds string column length.

尽管 **char** 、**varchar** 、**lvarchar** 和 **string** 主机变量包含空终止符，GBase 8s ESQL/C 不会将这些字符插入到数据库列。（**fixchar** 类型的主机变量绝对不能包含空终止符。）如果应用程序将 **char** 、**varchar** 、**lvarchar** 和 **string** 值插入到 VARCHAR 列，则数据库服务器将内部跟踪该值的末尾。

下表显示了当应用程序将其从 **char** 、**varchar** 、**lvarchar** 和 **string** 字符数据类型的主机变量插入时， VARCHAR 数据的转换。

表 5. 将 ESQL/C 字符数据类型转换为 VARCHAR 数据类型

源类型	目标类型	结果
char	VARCHAR	如果源类型大于 VARCHAR 的最大值，则截断该值并设置指示变量。如果 VARCHAR 的最大值大于原类型，则目标类型的长度等于源类型的长度（不包含源类型的空终止符）。
fixchar	VARCHAR	如果源类型大于 VARCHAR 的最大值，则截断该值并设置指示变量。如果 VARCHAR 的最大值大于原类型，则目标类型的长度等于源类型的长度。
string	VARCHAR	如果源类型大于 VARCHAR 的最大值，则截断该值并设置指示变量。如果 VARCHAR 的最大值大于原类型，则目标类型的长度等于源类型的长度（不包含源类型的空终止符）。
lvarchar	VARCHAR	如果源类型大于 VARCHAR 的最大值，则截断该值并设置指示变量。如果 VARCHAR 的最大值大于原类型，则目标类型的长度等于源类型的长度。

如果使用区域设置敏感的数据类型（ NVARCHAR），则可以将字符主机变量中的值插入 NVARCHAR 列。插入 NVARCHAR 列的方式与插入到 VARCHAR 列的行为相同。

下表显示了在插入期间 GBase 8s ESQL/C 可能执行的从 VARCHAR 列数据到字符主机变量的转换示例。在该表中，加号（+）表示空格字符。

表 6. 插入期间 VARCHAR 转换的示例

源类型	内容	长度	目标类型	内容	长度
char(10)	Fairfield\0	10	VARCHAR(	Fair	4

源类型	内容	长度	目标类型	内容	长度
			4)		
char(10)	Fairfield\0	10	VARCHAR(11)	Fairfield	9
char(12)	Fairfield++\0	12	VARCHAR(9)	Fairfield	9
char(13)	Fairfield+++\0	13	VARCHAR(6)	Fairfi	6
char(11)	Fairfield+\0	11	VARCHAR(11)	Fairfield+	10
fixchar(9)	Fairfield	9	VARCHAR(3)	Fai	3
fixchar(9)	Fairfield	9	VARCHAR(11)	Fairfield	9
fixchar(11)	Fairfield++	11	VARCHAR(9)	Fairfield	9
fixchar(13)	Fairfield++++	13	VARCHAR(7)	Fairfie	7
fixchar(10)	Fairfield+	10	VARCHAR(12)	Fairfield+	10
string(9)	Fairfield\0	9	VARCHAR(4)	Fair	4
string(9)	Fairfield\0	9	VARCHAR(11)	Fairfield	9

获取和插入 lvarchar 数据

当应用程序使用 lvarchar 主机变量获取或插入数据值时，GBase 8s ESQL/C 必须确保该值适合主机变量或数据库列。

获取 lvarchar 数据

应用程序可以从 LVARCHAR 类型的数据库列获取数据到字符（**char** 、**string** 、**fixchar** 、**varchar** 或 **lvarchar**）主机变量中。如果列数据不符合字符主机变量，则 GBase 8s ESQL/C 会截断此数据。要通知用户截断，GBase 8s ESQL/C 执行下列操作：

将 sqlca.sqlwarn.sqlwarn1 警告标记设置为 W 并将 SQLSTATE 变量设置为 01004。

它将与字符主机变量关联的任何指示变量设置为列中的字符数据的大小。

插入 lvarchar 数据

应用程序可以将字符主机变量（**char** 、**string** 、**fixchar** 、**varchar** 或 **lvarchar**）中

的数据插入到 LVARCHAR 类型的数据库列。

如果该值大于数据库列的大小，则数据库服务器截断该值（如果数据库为非 ANSI）。发生截断时不会产生警告。如果数据库是 ANSI 并且值大于列的长度，则插入失败并返回此错误：

-1279: Value exceeds string column length.

如果您使用的插入的主机变量是 **char** 或 **varchar**免责数据库服务器强制转型为 **lvarchar** 类型。

当向 LVARCHAR 列写入数据时，数据库服务器在列上施加了 32 KB 的限制。如果主机变量 **lvarchar** 数据类型。并且数据超过 32 KB，则数据库服务器返回错误。如果列具有输入支持功能，则它必须使用超过 32 KB 的任何数据，以防止数据库服务器返回错误。

在符合 ANSI 的数据库中获取或插入

对于符合 ANSI 的数据库，当在 INSERT 语句或 SQL 语句（SELECT 、UPDATE 或 DELETE）的 WHERE 子句中使用字符主机变量时，主变量中的字符必须为空终止符。因此，对于字符主机变量，使用以下数据类型：

char 、string 或 varchar

lvarchar

例如，下面的插入时有效的，因为第一个和**最后一个**主机变量时 **char** 类型，是空终止符：

```
EXEC SQL BEGIN DECLARE SECTION;
        char first[16], last[16];
        EXEC SQL END DECLARE SECTION;
        ⋮

        stcopy("Dexter", first);
        stcopy("Haven", last);
        EXEC SQL insert into customer (fname, lname)
        values (:first, :last);
```

stcopy() 函数将空终止符复制到主机变量中，**char** 数据类型保留空终止符。

不要对主机变量使用 **fixchar** 数据类型，因为它不能在字符串中包含空终止符。对于符合 ANSI 的数据库。数据库服务器会在以下任一情况下生成错误：

如果尝试插入不是空终止符的字符串

如在 WHERE 子句中使用不是空终止符的字符串

2.4.3 字符和字符串库函数

GBase 8s ESQL/C 库包含以下字符操纵函数。可以在 C 程序中使用这些函数操纵单个字符或字符的字节的字符串，包括以下数据类型的可变长度的表达式 ：

varchar

固定大小 lvarchar

lvarchar 指针数据类型引用的内部结构与固定大小的 lvarchar 变量的字符表示不同。必须使用 ifx_var() 函数操纵 lvarchar 指针变量。

名称以 **by** 开头的函数作用于返回固定长度的字符串。函数名称以 **rst** 和 **st**（除了 **stchar**）开头的函数运行并返回以 null 结尾的字符串。rdownshift() 和 rupshift() 函数也以 null 结尾的字符串运行，但不返回值。 当使用 esql 预处理程序编译 GBase 8s ESQL/C 程序，它调用链接程序将这些函数链接到您的程序。下表提供了字符和字符串库函数的简要说明，并将其引用到给出每个函数的详细信息的页面。

函数名称	描述	请参阅
bycmpr()	比较两组连续的子节	bycmpr() 函数
bycopy()	将字节从一个区域复制到另一区域	bycopy() 函数
byfill()	使用字符填充您指定的区域	byfill() 函数
byleng()	计算字符串的字节数	byleng() 函数
ldchar()	将固定长度的字符串复制到空终止符字符串	ldchar() 函数
rdownshift()	将所有的字母转换为小写字母	rdownshift() 函数
rstod()	将 string 转换为 double 值	rstod() 函数
rstoi()	将 string 转换为 short integer 值	rstoi()

函数名称	描述	请参阅
		函数
rstol()	将 **string** 转换为 4 字节整数值	rstol() 函数
rupshift()	将所有的字母转换为大写字母	rtypmsize() 函数
stcat()	将一个字符串连接到另一个字符串	stcat() 函数
stchar()	将空终止符字符串复制到固定长度的字符串中	stchar() 函数
stcmpr()	比较两个字符串	stcmpr() 函数
stcopy()	将一个字符串复制到另一个字符串中	stcopy() 函数
stleng()	计算字符串中字节的长度	stleng() 函数

2.5 Numeric 数据类型

GBase 8s 数据库服务器支持以下 Numeric 数据类型：

Integer 数据类型：SMALLINT 、INTEGER 、INT8 、SERIAL 、SERIAL8

Boolean 数据类型

定点数据类型：DECIMAL 和 MONEY

浮点数据类型：SMALLFLOAT 和 FLOAT

这些主题包含如何有关使用 Numeric 数据类型的信息：

GBase 8s ESQL/C 数据类型用作 SQL 数字数据类型的主机变量

GBase 8s ESQL/C 数字数据类型的特征

格式化掩码，可用于格式化数字数据类型

GBase 8s ESQL/C 库函数，可用于操作数字数据类型

2.5.1 整型数据类型

数据库服务器支持整数值的以下数据类型：

SQL 整数数据类型	字节数	值的范围
SMALLINT	2	-32767 到 32767
INTEGER 、INT 、SERIAL	4	-2,147,483,647 到 2,147,483,647
INT8 、BIGINT 、SERIAL8 、BIGSERIAL	8	-9,223,372,036,854,775,807 到 9,223,372,036,854,775,807

C 语言支持整数值的 **short int** 和 **long int** 数据类型。

C **short int** 数据类型的存储大小取决于您使用的计算机的硬件和操作系统。

在 ESQL/C 中，无论平台和硬件如何，C 的 **long int** 数据类型总是被视为为 4 字节。这使得 **long int** 对于存储 GBase 8s 的 SMALLINT 、INTEGER 、INT 和 SERIAL 数据类型十分有用。

重要：

但是，不要尝试使用 **long int** 数据类型存储 8 字节 GBase 8s 整数数据类型 INT8 、BIGINT 、SERIAL8 或 BIGSERIAL。例如：当您的查询尝试在 -2,147,483,647 到 + 2,147,483,647 之间的 8 字节 BIGSERIAL 值中选择数据类型为 **long int** 的整数 C 变

量时，数据库服务器会发出此错误：

-1215 Value too large to fit in an INTEGER.

当声明整数主机变量时，必须确保该主机变量足够大，以便与该变量关联的 SQL 整数数据类型的所有值相匹配。对于 8 字节整数，使用 C 数据类型 **bigint** 或 **int8** 的主机变量。有关如何在系统上实现整数数据类型的更多信息，请咨询您的系统管理员或参阅 C 文档。

整数主机变量类型

提供以下数据类型用于指定特定长度的整数主机变量。

数据类型

长度

int1

1 个字节的整数

int2

2 个字节的整数

int4

4 个字节的整数

mint

机器本机的整数数据类型

mlong

机器本机的 **long** 整数数据类型，其大小等于机器指针的大小。**mlong** d数据类型映射到 Windows(TM) 32 位和 UNIX(TM) 和 Linux(TM) 32 位和 64 位平台上的 **long** 数据类型。它被映射到 Windows(TM) 64 位的 **__int64** 数据类型。

MSHORT

机器本机的 **short** 整数数据类型

MCHAR

机器本机的 **char** 数据类型

限制：前面的整数数据类型保留。您的程序不能使用 typedef 或 $typedef 语句定义这些数据类型。

在 ifxtypes.h 文件中定义整数主机变量数据类型，当您使用 esql 脚本编译它时，它

会字段包含在您的程序中。

重要： 许多 GBase 8s ESQL/C 库函数已经更改为声明 GBase 8s 整数数据类型，而不是特定于机器的类型，如 **int** 、**short** 和 **long**。当您调用 GBase 8s ESQL/C 库函数时，建议您使用 GBase 8s 整数类型。

INT8 和 SERIAL8 SQL 数据类型

GBase 8s ESQL/C 支持具有 int8 数据类型的 SQL INT8 和 SERIAL8 数据类型。int8 数据类型是一个独立于机器的方法，表示数字范围 -(263 -1) 到 263-1。

本节描述了如何操作 GBase 8s ESQL/C 数据类型 **int8**。

int8 数据类型

使用 GBase 8s ESQL/Cint8 数据类型为 INT8 和 SERIAL8 类型的数据库值声明主机变量。

下表显示结构 **ifx_int8_t** 的字段，表示 INT8 或 SERIAL8 值。

表 1. ifx_int8_t 结构的字段

字段名称	字段类型	意义
data	*无符号 4 字节整数*[INT8tIZE]	构成 8 字节整数值的整数值数组。当 INT8tIZE 常量定义为 2，此数组包含两个无符号的 4 字节整数。无符号 4 字节整数的实际数据类型可以是机器特定的。
sign	short integer	一个短整数，用于保存 8 字节整数的符号（无 、负或正）。2 字节整数的实际数据类型可以是机器特定的。

int8.h 头文件包含 **ifx_int8** 结构和名为 **ifx_int8_t** 的 **typedef**。将此文件包含在使用任何 **int8** 主机变量的所有 C 源文件中，如下所示：

```
EXEC SQL include int8;
```

可以使用以其中之一的方法声明 **int8** 主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        int8 int8_var1;
        ifx_int8_t int8_var2;
        EXEC SQL BEGIN DECLARE SECTION;
```

int8 库函数

您必须通过用于 **int8** 数据类型的 GBase 8s ESQL/C 库函数对 **int8** 类型数字执行所有操作。任何其它操作，修改或分析都可能产生不可预知的结果。GBase 8s ESQL/C 库函数允许您操作 **int8** 数并将 **int8** 类型数字转换为其它数据类型的函数。下表说明了这些函数。

表 2. 操纵函数

函数名称	描述	请参阅
ifx_getserial8()	返回插入的 SERIAL8 值	ifx_int8add() 函数
ifx_int8add()	添加两个 int8 数字	ifx_int8cmp() 函数
ifx_int8cmp()	比较两个 int8 数字	ifx_int8copy() 函数
ifx_int8copy()	复制一个 int8 数字	ifx_int8cvasc() 函数
ifx_int8div()	除以两个 int8 数字	ifx_int8div() 函数
ifx_int8mul()	乘以两个 int8 数字	ifx_int8mul() 函数
ifx_int8tub()	减去两个 int8 数字	ifx_int8tub() 函数

表 3. 类型转换函数

函数名称	描述	请参阅
ifx_int8cvasc()	将 C char 类型值转换为 **int8** 类型值	ifx_int8cvdbl() 函数
ifx_int8cvdbl()	将 C double 类型值转换为 **int8** 类型值	ifx_int8cvdbl() 函数
ifx_int8cvdec()	将 C decimal 类型值转换为 **int8** 类型值	ifx_int8cvdec() 函数
ifx_int8cvflt()	将 C **float** 类型值转换为 **int8** 类型值	ifx_int8cvflt() 函数
ifx_int8cvint()	将 C int 类型值转换为 **int8** 类型值	ifx_int8cvint() 函数
ifx_int8cvlong()	将 C 4 字节整数类型值转换为 **int8** 类型值	ifx_int8cvlong() 函数
ifx_int8toasc()	将 **int8** 类型值转换为文本字符串	ifx_int8toasc() 函数
ifx_int8todbl()	将 **int8** 类型值转换为 C **double** 类型值	ifx_int8todbl() 函数

函数名称	描述	请参阅
ifx_int8todec()	将 **int8** 类型值转换为 C **decimal** 类型值	ifx_int8todec() 函数
ifx_int8toflt()	将 **int8** 类型值转换为 C **float** 类型值	ifx_int8toflt() 函数
ifx_int8toint()	将 **int8** 类型值转换为 C **int** 类型值	ifx_int8toint() 函数
ifx_int8tolong()	将 **int8** 类型值转换为 C 4 字节整数类型值	ifx_int8tolong() 函数

2.5.2 BOOLEAN 数据类型

GBase 8s ESQL/C 使用 boolean 数据类型支持 SQL BOOLEAN 数据类型。

本节介绍了如何使用 GBase 8s ESQL/C**boolean** 数据类型。

可按以下示例声明 **boolean** 主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
    boolean flag;
EXEC SQL BEGIN DECLARE SECTION;
```

在 GBase 8s ESQL/C 程序中，下列值仅在您指定给 **boolean** 主机变量时有效：

TRUE

'\1'

FALSE

'\0'

NULL

使用具有 CBOOLTYPE 为第一个参数的 rsetnull() 函数

如果您要给 BOOLEAN 列指定 'T' 或 'F' 字符表示，必须声明一个 **fixchar** 主机变量并将它初始化为希望的字符值。在诸如 INSERT 或 UPDATE 的 SQL 语句中使用此主变量。数据库服务器将 **fixchar** 值转换为适当的 BOOLEAN 值。

下面的代码段将两个值插入到表 **table2** 中名为 **bool_col** 的 BOOLEAN 列中：

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    boolean flag;
    fixchar my_boolflag;
    int id;
EXEC SQL END DECLARE SECTION;

id = 1;
flag = '\0';    /* valid boolean assignment to FALSE */
EXEC SQL insert into table2 values (:id, :flag); /* inserts FALSE */

id = 2;
rsetnull(CBOOLTYPE, (char *) &flag); /* valid BOOLEAN assignment
                                        * to NULL */
EXEC SQL insert into table2 values (:id, :flag); /* inserts NULL */

id = 3;
my_boolflag = 'T' /* valid character assignment to TRUE */
EXEC SQL insert into table2 values (:id, :my_boolflag); /* inserts TRUE
                                                           */
```

2.5.3 Decimal 数据类型

GBase 8s ESQL/C 支持使用 decimal 数据类型的 SQL DECIMAL 和 MONEY 数据类型。decimal 数据类型是一种独立于机器的方法，代表最多 32 位有效数字，有效值范围为 10-129 - 10+125。

DECIMAL 数据类型可以使用以下两种格式：

DECIMAL(p) 浮点

当使用 DECIMAL(*p*) 数据类型定义列时，它总共有 *p* (< = 32) 个有效位。DECIMAL(*p*) 的绝对值范围为 10^{-130} - 10^{124}。

DECIMAL(p,s) 定点

当使用 DECIMAL(*p*,*s*) 数据类型定义列时，它总共有 *p* (< = 32) 个有效数字（精度）以及 *s* (< = *p*) 位小数部分的总位数（小数位）。

decimal 结构

使用 decimal 数据类型为 DECIMA 类型的数据库值声明主机变量。

decimal 类型的结构表示 **decimal** 主机变量中的一个值，如下所示：

```
#define DECSIZE 16

struct decimal
{
short dec_exp;
short dec_pos;
short dec_ndgts;
char    dec_dgts[DECSIZE];
};

typedef struct decimal dec_t;
```

decimal.h 头文件包含 **decimal** 结构和 **typedefdec_t**。将此文件包含在使用任何 **decimal** 主机变量的所有 C 源文件中，其中包含以下 **include** 指令：

```
EXEC SQL include decimal;
```

decimal 结构以数字对的形式存储。每对都是一个范围在 00 到 99 之间的数字（因此，可以将一对看作一个 100 位的数字）。下表显示了 **decimal** 结构的四个部分。

表 4. decimal 结构中的字段

字段	描述
dec_exp	*规范化***decimal** 类型数字的指数。该数字的标准化形式在最左边数字的左侧有小数点。该指数表示从左侧计数到位置小数点的数字对的位置（或对于 100 个数字的数字，为 100 的幂数）。
dec_pos	**decimal** 类型数字的*符号*。**dec_pos** 字段可以假定以下三个值之一：1：当数字大于等于零 0：当数字小于零 –1：当值为空
dec_ndgts	**decimal** 类型数中*数字对对数*（基数为 100 的有效数字的数）。该值也是 **dec_dgts** 数组中实体数。
dec_dgts[]	字符数组，保存标准化 decimal 类型数字的有效数字，假定 **dec_dgts[0] ! = 0**。 数字中的每个字节包含 **decimal** 类型数字中的下一个有效的基数的

字段	描述
	100 位，从 **dec_dgts[0]** 到 **dec_dgts[dec_ndgts]**。

下表显示了一些样本 **decimal** 值。

表 5. decimal 字段值的结构示例

值	dec_exp	dec_pos	dec_ndgts	dec_dgts[]
-12345.6789	3	0	5	dec_dgts[0] = 01 dec_dgts[1] = 23 dec_dgts[2] = 45 dec_dgts[3] = 67 dec_dgts[4] = 89
1234.567	2	1	4	dec_dgts[0] = 12 dec_dgts[1] = 34 dec_dgts[2] = 56 dec_dgts[3] = 70
-123.456	2	0	4	dec_dgts[0] = 01 dec_dgts[1] = 23 dec_dgts[2] = 45 dec_dgts[3] = 60
480	2	1	2	dec_dgts[0] = 04 dec_dgts[1] = 80
.152	0	1	2	dec_dgts[0] = 15

值	dec_exp	dec_pos	dec_ndgts	dec_dgts[]
				dec_dgts[1] = 20
-6	1	0	1	dec_dgts[0] = 06

可以使用 **deccvasc** 演示程序展示 GBase 8s ESQL/C 是如何 **decimal** 数字的。

decimal 库函数

您必须通过用于 **decimal** 数据类型的 GBase 8s ESQL/C 库函数对 **decimal** 类型数字执行所有操作。任何其它操作，修改或分析都可能产生不可预知的结果。

表 6. 操作函数

函数名称	描述	请参阅
decadd()	添加两个 decimal 数字	decadd() 函数
deccmp()	比较两个 decimal 数字	deccmp() 函数
deccopy()	复制一个 decimal 数字	deccopy() 函数
decdiv()	除以两个 decimal 数字	decdiv() 函数
decmul()	乘以两个 decimal 数字	decmul() 函数
decround()	舍入一个 decimal 数字	decround() 函数
decsub()	减去两个 decimal 数字	decsub() 函数
dectrunc()	截断一个 decimal 数字	dectrunc() 函数

表 7. 类型转换函数

函数名称	描述	请参阅
deccvasc()	将 C char 类型值转换为 **decimal** 类型值	deccvasc() 函数
deccvdbl()	将 C double 类型值转换为 **decimal** 类型值	deccvdbl() 函数
deccvint()	将 C int 类型值转换为 **decimal** 类型值	deccvint() 函数
deccvlong()	将 C 4 字节整数类型值转换为 **decimal** 类型值	deccvlong() 函数
dececvt()	将 decimal 值转换为 ASCII 字符串	dececvt() 和

函数名称	描述	请参阅
		decfcvt() 函数
decfcvt()	将 decimal 值转换为 ASCII 字符串	dececvt() 和 decfcvt() 函数
dectoasc()	将 **decimal** 类型值转换为 ASCII 字符串	dectoasc() 函数
dectodbl()	将 **decimal** 类型值转换为 C **double** 类型值	dectodbl() 函数
dectoint()	将 **decimal** 类型值转换为 C **int** 类型值	dectoint() 函数
dectolong()	将 **decimal** 类型值转换为 C 4 字节整数类型值	dectolong() 函数

2.5.4 浮点数据类型

数据库服务器支持浮点值的以下数据类型。

SQL 浮点数据类型	ESQL/C 或 C 语言类型	值的范围
SMALLFLOAT, REAL	float	单精度值可达到 9 位有效数字
FLOAT, DOUBLE PRECISION	double	双精度值可达到 17 位有效数字
DECIMAL(*p*)	decimal	绝对值的范围 10^{-130} - 10^{124}

声明浮点主机变量

当使用 C **float** 数据类型时（对于 SMALLFLOAT 值），请注意大多数 C 编译程序将 **float** 作为 **double** 数据类型传递给函数。如果您声明函数参数为 **float**，可能接收一个不正确的结果。例如：在以下摘录中，**:hostvar** 可能在 **tab1** 中产生一个不正确的值，这取决于当您的程序将它作为参数传递时，C 编译程序如何处理 **float** 数据类型。

```
main()
        {
        double dbl_val;

        EXEC SQL connect to 'mydb';
        ins_tab(dbl_val);
        ⋮
```

```
}

ins_tab(hostvar)
EXEC SQL BEGIN DECLARE SECTION;
PARAMETER double hostvar;
EXEC SQL END DECLARE SECTION;
{
EXEC SQL insert into tab1 values (:hostvar, ...);
}
```

隐式数据转换

当 GBase 8s ESQL/C 程序将浮点列值提取到字符主机变量（**char** 、**fixchar** 、**varchar** 或 **string**）中时，它仅包含可以适合字符缓冲区的 decimal 数字数。如果主机变量对于浮点数的完整精度来说太小，则 GBase 8s ESQL/C 将该数字舍入到主机变量可以容纳的精度。

在以下的代码段中，GBase 8s ESQL/C 程序从名为 **principal** 的 FLOAT 列中将值 1234.8763512 检索到 **prncpl_strng** 字符主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        char prncpl_strng[15]; /* character host variable */
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL select principal into :prncpl_strng from loan
        where customer_id = 1098;
        printf("Value of principal=%s\n", prncpl_strng);
```

因为 **prncpl_strng** 主机变量时一个 15 字节的缓冲区，所以 GBase 8s ESQL/C 可以将所有的 decimal 数字放置到主机变量中，此代码段产生以下输出：

```
Value of principal=1234.876351200
```

但是，如果前面的代码段声明 **prncpl_strng** 主机变量为一个 10 字节的缓冲区，则

GBase 8s ESQL/C 舍入 FLOAT 值以填充到 **prncpl_strng**，此代码段产生以下输出：

```
Value of principal=1234.8764
```

GBase 8s ESQL/C 为 FLOAT 或 SMALLFLOAT 值假定 17 十进制位的精度。对于 DECIMAL(*n*,*m*)，GBase 8s ESQL/C 假定 *m* 个数位。

2.5.5 格式化数值字符串

数字格式化掩码指定要应用于某些数值的格式。

此掩码是以下格式字符的组合：

*

此字符在显示字段中以空白填充任何位置的星号。

&

此字符将填充任何空白的显示字段中的任何位置。

#

该字符将前面位置的零更改为空白。使用此字符指定字段的最大值的左侧范围。

<

该字符左对齐显示字段中的数字。它将前导零更改为空字符串。

,

该字符指示在值的全数部分中分隔三位数组（从单位位置向左计数）的符号。缺省情况下，此符号是逗号。您可以使用 DBMONEY 环境变量设置符号。在格式化的数字中，只有当该值的全数部分具有四位或更多位数时，才会出现该符号。

.

该字符表示将货币值的全数部分与小数部分分开的符号。缺省情况下，此符号是一个句点。您可以使用 DBMONEY 环境变量设置此符号。格式字符串中只能有一个句点。

-

这个字符是一个文字。当 *expr1* 小于零时，它显示为负号。当您连续排列几个减号，单个减号会浮动到可占据的最右侧位置，它不会干扰数字及其货币符号。

+

该符号是一个文字。当 *expr1* 大于等于零时，它显示为加号。当 *expr1* 小于零时，它显示为负号。当您连续组合几个加号时，单个加号或减号会浮动到可占据的最右侧位置，它不会干扰数字及其货币符号。

(

该符号是一个文字。它显示负数左侧的左括号。它是一对会计括号中的一个，用于替换负数的括号。当您连续组合几个时，单个左括号将浮动到可占据的最右侧位置，它不会干扰数字及其货币符号。

)

这是用于替换负值的负号的一对会计括号之一。

$

该字符显示出现在数值前面的货币符号。缺省情况下，货币符号是美元符号（$）。可以使用 DBMONEY 环境变量设置此符号。当您连续组合几个美元符号时，单个货币符号浮动到可以占据的最右侧位置，它不会干扰这个数字。

格式化掩码中的任何其它字符在结果中字面上复制。

当您在格式化掩码中使用以下字符时，字符为 *float*；即，掩码中图案左侧的多个字符出现在格式化数字中尽可能向右的单个字符（不删除有效数字）：

-

+

(

)

$

例如：如果对数字 1234.56 应用掩码 $$$,$$$.## ，则该结果为 $1,234.56。

当您使用 rfmtdec() 、rfmtdouble() 或 rfmtlong() 格式化 MONEY 值时，函数使用 **DBMONEY** 环境变量指定的货币符号。如果未设置此环境变量，则数字格式化函数使用客户端语言环境定义的的货币符号。缺省情况使用美国英语定义货币符号，就像将 DBMONEY 设置为 “$,.”。

数字表达式的示例格式字符串

下表显示了数字表达式的示例格式字符串。字母 b 代表空格或空白。

表 8. 样本格式模式及结果

格式化掩码	数值	格式化结果
"#####"	0	

格式化掩码	数值	格式化结果
"&&&&&"	0	bbbbb 00000 bbbb$ *****
"$$$$$"	0	（空字符串）
"*****"	0	
"<<<<<"	0	
“##,###”	12345	
"##,###"	1234	12,345 b1,234 bbb123
"##,###"	123	bbbb12 bbbbb1 bbbbb1 bbbbbb
"##,###"	12	
"##,###"	1	
"##,###"	-1	
"##,###"	0	
"&&,&&&"	12345	12,345
"&&,&&&"	1234	01,234
"&&,&&&"	123	000123
"&&,&&&"	12	000012
"&&,&&&"	1	000001
"&&,&&&"	-1	000001
"&&,&&&"	0	000000
"$$,$$$"	12345	
"$$,$$$"	1234	*****（溢出）
"$$,$$$"	123	
"$$,$$$"	12	$1,234 bb$123 bbb$12
"$$,$$$"	1	bbbb$1 bbbb$1
"$$,$$$"	-1	bbbbb$ DM1,234
"$$,$$$"	0	
"$$,$$$" （DBMONEY 设置为 DM）	1234	
"**,***"	12345	12,345
"**,***"	1234	*1,234
"**,***"	123	***123
"**,***"	12	****12

格式化掩码	数值	格式化结果
"**,***"	1	*****1
"**,***"	0	******
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbbb.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbb1.00
"&&,&&&.&&"	.67	000000.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"$$,$$$.$$"	12345.67	
"$$,$$$.$$"	1234.56	*********（溢出）
"$$,$$$.##"	0.00	
"$$,$$$.##"	1234.00	$1,234.56 bbbbb$.00
"$$,$$$.&&"	0.00	$1,234.00 bbbbb$.00 $1,234.00
"$$,$$$.&&"	1234.00	
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	b-bbb12.34
"---,###.##"	-12.34	bb-bb12.34
"---,-##.##"	-12.34	bbbb-12.34
"---,--#.##"	-12.34	bbbb-12.34
"--#,###.##"	-1.00	b-bbbb1.00
"---,--#.##"	-1.00	bbbbb-1.00
"-##,###.##"	12345.67	b12,345.67
"-##,###.##"	1234.56	bb1,234.56

格式化掩码	数值	格式化结果
"-##,###.##"	123.45	bbbb123.45
"-##,###.##"	12.34	bbbbb12.34
"--#,###.##"	12.34	bbbbb12.34
"---,###.##"	12.34	bbbbb12.34
"---,-##.##"	12.34	bbbbb12.34
"---,---.##"	1.00	bbbbbb1.00
"---,---.--"	-.01	bbbbbb-.01
"---,---.&&"	-.01	bbbbbb-.01
"-$$$,$$$.&&"	-12345.67	-$12,345.67
"-$$$,$$$.&&"	-1234.56	-b$1,234.56
"-$$$,$$$.&&"	-123.45	-bbb$123.45
"--$$,$$$.&&"	-12345.67	-$12,345.67
"--$$,$$$.&&"	-1234.56	b-$1,234.56
"--$$,$$$.&&"	-123.45	b-bb$123.45
"--$$,$$$.&&"	-12.34	b-bbb$12.34
"--$$,$$$.&&"	-1.23	b-bbbb$1.23
"----,--$.&&"	-12345.67	-$12,345.67
"----,--$.&&"	-1234.56	b-$1,234.56
"----,--$.&&"	-123.45	bbb-$123.45
"----,--$.&&"	-12.34	bbbb-$12.34
"----,--$.&&"	-1.23	bbbbb-$1.23
"----,--$.&&"	-.12	bbbbbb-$.12
"$***,***.&&"	12345.67	$*12,345.67
"$***,***.&&"	1234.56	$**1,234.56
"$***,***.&&"	123.45	$****123.45
"$***,***.&&"	12.34	$*****12.34
"$***,***.&&"	1.23	$******1.23
"$***,***.&&"	0.12	$*******.12
"($$$,$$$.&&)"	-12345.67	($12,345.67)
"($$$,$$$.&&)"	-1234.56	(b$1,234.56)
"($$$,$$$.&&)"	-123.45	(bbb$123.45)
"(($$,$$$.&&)"	-12345.67	($12,345.67)

格式化掩码	数值	格式化结果
"(($$,$$$.&&)"	-1234.56	b($1,234.56)
"(($$,$$$.&&)"	-123.45	b(bb$123.45)
"(($$,$$$.&&)"	-12.34	b(bbb$12.34)
"(($$,$$$.&&)"	-1.23	b(bbbb$1.23)
"((((,(($.&&)"	-12345.67	($12,345.67)
"((((,(($.&&)"	-1234.56	b($1,234.56)
"((((,(($.&&)"	-123.45	bbb($123.45)
"((((,(($.&&)"	-12.34	bbbb($12.34)
"((((,(($.&&)"	-1.23	bbbbb($1.23)
"((((,(($.&&)"	-.12	bbbbbb($.12)
"($$$,$$$.&&)"	12345.67	b$12,345.67
"($$$,$$$.&&)"	1234.56	bb$1,234.56
"($$$,$$$.&&)"	123.45	bbbb$123.45
“(($$,$$$.&&)”	12345.67	b$12,345.67
"(($$,$$$.&&)"	1234.56	bb$1,234.56
"(($$,$$$.&&)"	123.45	bbbb$123.45
"(($$,$$$.&&)"	12.34	bbbbb$12.34
"(($$,$$$.&&)"	1.23	bbbbbb$1.23
"((((,(($.&&)"	12345.67	b$12,345.67
"((((,(($.&&)"	1234.56	bb$1,234.56
"((((,(($.&&)"	123.45	bbbb$123.45
"((((,(($.&&)"	12.34	bbbbb$12.34
"((((,(($.&&)"	1.23	bbbbbb$1.23
"((((,(($.&&)"	0.12	bbbbbbb$.12
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12

数字格式化函数

提供了特定函数，允许您格式化数字表达式进行显示。

这些格式化函数将给定的格式化掩码应用于数值，以便您可以对小数点进行排列，向

右或向左对齐数字，在圆括号中括起一个负数以及其它格式化功能。GBase 8s ESQL/C 库包含下列支持数值格式化掩码的函数。

函数名称	描述	请参阅
rfmtdec()	将 decimal 值转换为字符串	rfmtdec() 函数
rfmtdouble()	将 double 值转换为字符串	rfmtdouble() 函数
rfmtlong()	将 4 字节整数值转换为字符串	rfmtlong() 函数

2.6 时间数据类型

这些主题介绍在 GBase 8s ESQL/C 程序中如何使用 date 、datetime 和 interval 数据类型。

本节包含以下信息：

GBase 8s ESQL/Cdate 数据类型概述

可用于操纵 date 数据类型的 GBase 8s ESQL/C 库函数的语法

GBase 8s ESQL/Cdatetime 和 interval 数据类型的概述及如何使用它们

可用于操纵 datetime 和 interval 数据类型的 GBase 8s ESQL/C 库函数的语法

2.6.1 SQL DATE 数据类型

GBase 8s ESQL/C 支持具有主机变量的 GBase 8s ESQL/C**date** 数据类型的 SQL DATE 数据类型。**date** 数据类型存储内部 DATE 值。它被实现为一个 4 字节整数，其值是自 1899 年 12 月 31 日以来的天数。1899 年 12 月 31 日之前的日期是负数，1899 年 12 月 31 日之后的日期是正数。

格式化日期字符串

日期格式化指定应用于某些日期值的格式。

该掩码是以下格式的组合。

dd

当前的日期为两位数（01 - 31）

ddd

星期几位三个字母的缩写（Sun - Sat）

mm

月份为两位数（01 - 12）

mmm

月份为三个字母的缩写（Jan - Dec）

yy

年份为两位数（00 - 99）

yyyy

年份为四位数（0001 - 9999）

ww

星期几位两位数（星期日 00 ，星期一 01 ，星期二 02 … 星期六 06）

格式化掩码中的任何其它字符在结果中字面上复制。

当使用 rfmtdate() 或 rdefmtdate() 格式化 DATE 值时，函数使用 GLDATE 或 DBDATE 环境变量指定的日期最终用户格式。如果设置了其中一个环境变量，则这些日期格式化函数为语言环境使用日期最终用户格式。缺省的语言环境是美国英语，使用格式 mm/dd/yyyy。

2.6.2 DATE 库函数

下表为 GBase 8s ESQL/C 库中的日期操纵函数。它们在字符串格式和内部 DATE 格式之间转换日期。

函数名称	描述	请参阅
rdatestr()	将内部 DATE 转换为字符字符串格式	rdatestr() 函数
rdayofweek()	以内部格式返回日期的星期几	rdayofweek() 函数
rdefmtdate()	将指定的字符串转换为内部 DATE	rdefmtdate() 函数
rfmtdate()	将内部 DATE 转换为指定的字符串格式	rfmtdate() 函数
rjulmdy()	返回指定 DATE 的年、月、日	rjulmdy() 函数
rleapyear()	确定年份是否为闰年	rleapyear() 函数
rmdyjul()	从月份、日和年份返回 DATE 格式	rmdyjul() 函数
rstrdate()	将字符字符串格式转换为内部 DATE	rstrdate() 函数
rtoday()	返回系统日期最为内部 DATE	rtoday() 函数

当使用 esql 命令编译 GBase 8s ESQL/C 程序时，esql 自动来将这些函数链接到您的程序中。

2.6.3 SQL DATETIME 和 INTERVAL 数据类型

GBase 8s ESQL/C 支持可以保存时间值消息的这两种数据类型：

datetime 数据类型，以时间作为日历日期和时间进行编码。

interval 数据类型，编码一段时间。

下表总了这两种时间数据类型。

表 1. ESQL/C 时间数据类型

SQL 数据类型	ESQL/C 数据类型	C typedef 名称	样本声明
DATETIME	datetime	dtime_t	EXEC SQL BEGIN DECLARE SECTION; datetime 年销售; EXEC SQL END DECLARE SECTION;
INTERVAL	interval	intrvl_t	EXEC SQL BEGIN DECLARE SECTION; 间隔小时到第二个 test_num; EXEC SQL END DECLARE SECTION;

头文件 datetime.h 包含 **dtime_t** 和 **intrvl_t** 结构，以及可用于组合限定符值的多个宏定义。将此文件包含在使用任何 **datetime** 或 **interval** 主机变量的 C 源文件中：

```
EXEC SQL include datetime;
```

decimal.h 头文件定义类型 **dec_t**，它是 **dtime_t** 和 **intrvl_t** 结构的一个组件。

由于这些数据类型的多字性质，不可能声明一个名为 **year** 、**month** 、**day** 、**hour** 、**minute** 、**second** 或 **fraction** 的未初始化的 **datetime** 或 **interval** 主机变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        datetime year;                  /* will cause an error */
        datetime year to day year, today;               /* ambiguous */
        EXEC SQL END DECLARE SECTION;
```

datetime 或 **interval** 数据类型存储为十进制数，缩放因子为零，精度等于其限定符隐含的位数。当您了解精度和尺度时，可以知道存储格式。例如：如果将表列定义为 DATETIME YEAR TO DAY，则它包含四位数年份、两位数月份以及两位数日期，一共有八位数。因此它被存储为 **decimal(8,0)**。

如果基准十进制值的缺省精度不合适，则可以指定不同的精度。例如，如果您有一个类型为 **interval** 的主机变量，具有限定符 **day to day**，则底层的十进制数的缺省精度为两位。如果有一百天以上的时间间隔，这个精度是不够的。可以指定三位数的精度如下：

interval day(3) to day;

datetime 数据类型

使用 datetime 数据类型为 DATETIME 类型的数据库值声明主机变量。使用限定符指定 datetime 数据类型的准确性。

例如：以下声明中的限定符是 **year to day**：

datetime year to day sale;

作为一个主机变量 **dtime_t**. 结构表示一个 **datetime** 值：

```
typedef struct dtime {
        short dt_qual;
        dec_t dt_dec;
        } dtime_t;
```

dtime 结构和 **dtime_t** typedef 具有两个部分。下表列出了这些部分。

表 2. dtime 结构中的字段

字段	描述
dt_qual	datetime 值的限定符
dt_dec	datetime 值的字段位数，此字段是 **decimal** 值。

声明一个 DATETIME 列的主机变量，其中 **datetime** 数据类型后面是可选的限定符，如下所示：

```
EXEC SQL include datetime;
        ⋮

        EXEC SQL BEGIN DECLARE SECTION;
        datetime year to day holidays[10];
        datetime hour to second wins, places, shows;
```

```
        datetime column6;
        EXEC SQL END DECLARE SECTION;
```

如果您省略上一个示例中 datetime 主机变量中的限定符，则您的程序必须使用表 1中显示的宏显式初始化此限定符。

interval 数据类型

使用 interval 数据类型声明 INTERVAL 类型的数据库列的主机变量。

可以使用限定符指定 **interval** 数据类型的准确性。以下声明中的限定符为 **hour to second**：

```
interval hour to second test_run;
```

作为一个主机变量 **intrvl_t**，它表示一个 **interval** 值：

```
typedef struct intrvl {
        short in_qual;
        dec_t in_dec;
        } intrvl_t;
```

intrvl 结构和 **intrvl_t** typedef 具有两个部分。下表列出了这些部分。

表 3. intrvl 结构中的字段

字段	描述
in_qual	interval 值的限定符
in_dec	interval 值的字段位数，此字段是 **decimal** 值

声明一个 INTERVAL 列的主机变量，其中 **interval** 数据类型后面是可选的限定符，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        interval day(3) to day accrued_leave, leave_taken;
        interval hour to second race_length;
        interval scheduled;
        EXEC SQL END DECLARE SECTION;
```

如果您省略上一个示例中 **interval** 主机变量中的限定符，则您的程序必须使用以下章节中描述的宏显式初始化此限定符。

datetime 和 interval 数据类型的宏

除了 **datetime** 和 **interval** 数据结构，datetime.h 文件定义了下表中显示的宏函数，以直接使用二进制形式的限定符。

表 4. datetime 和 interval 数据类型的限定符宏

宏的名称	描述
TU_YEAR	YEAR 限定符字段的实际单位
TU_MONTH	MONTH 限定符字段的实际单位
TU_DAY	DAY 限定符字段的实际单位
TU_HOUR	HOUR 限定符字段的实际单位
TU_MINUTE	MINUTE 限定符字段的实际单位
TU_SECOND	SECOND 限定符字段的实际单位
TU_FRAC	FRACTION 的前导限定符字段的实际单位
TU_F*n*	名称为 **datetime** 结束字段的 FRACTION(*n*)，*n* 为 1 - 5
TU_START(*q*)	从限定符 *q* 返回前导字段数
TU_END(*q*)	从限定符 *q* 返回末尾字段数
TU_LEN(*q*)	返回限定符 *q* 中位数的长度
TU_FLEN(*f*)	返回 **interval** 限定符的第一个字段的位数长度
TU_ENCODE(*p,f,t*)	从第一个字段数 *f* 创建一个限定符，其精度为 *p* ，尾部字段数为 *t*
TU_DTENCODE(*f,t*)	从第一个字段数 *f* 和尾随的字段数 *t* 创建一个 **datetime** 限定符
TU_IENCODE(*p,f,t*)	从第一个字段数 *f* 和尾随的字段数 *t* 创建一个 **interval** 限定符，其精度为 *p* ，尾部字段为 *t*

例如：如果您的程序不在主机变量的声明声明中提供 **interval** 限定符，您需要使用 **interval** 限定符宏初始化并设置 **interval** 主机变量。在以下示例中，**interval** 变量获得一个 **day to second** 限定符，限定符中的最长字段的精度 **day** 设置为 2：

```
/* declare a host variable without a qualifier */
        EXEC SQL BEGIN DECLARE SECTION;
        interval inv1;
```

```
EXEC SQL END DECLARE SECTION;
⋮

/* set the interval qualifier for the host variable */
inv1.in_qual = TU_IENCODE(2, TU_DAY, TU_SECOND);
⋮

/* assign values to the host variable */
incvasc ("5 2:10:02", &inv1);
```

获取和插入 DATETIME 和 INTERVAL 值

当应用程序获取或插入 DATETIME 或 INTERVAL 值时，GBase 8s ESQL/C 必须确保主机变量的限定符字段是有效的：

当应用程序将 DATETIME 值提取到 datetime 主机变量或从 datetime 主机变量插入 DATETIME 值时，必须确保 dtime_t 结构的 dt_qual 字段有效。

当应用程序将 INTERVAL 值提取到 interval 主机变量或从 interval 主机变量插入 INTERVAL 值时，必须确保 intrvl_t 结构的 in_qual 字段有效。

获取和插入到 datetime 主机变量

当应用程序使用 **datetime** 主机变量获取或插入 DATETIME 值时，GBase 8s ESQL/C 必须在 **datetime** 主机变量中找到一个有效的限定符。GBase 8s ESQL/C 根据与主变量相关联的 **dtime_t** 结构中的 **dt_qual** 字段的值，执行以下操作之一：

当 dt_qual 字段包含有效限定符时，GBase 8s ESQL/C 扩展列值以符合 dt_qual 限定符。

扩展时添加或删除 DATETIME 值的字段以使其与给定的限定符匹配的操作。您可以使用 SQL EXTEND 函数和 GBase 8s ESQL/C**dtextend()** 函数显式扩展 DATETIME 值。

当 dt_qual 字段不包含一个有效限定符时，GBase 8s ESQL/C 对获取和插入采取不同的操作：

对于获取，GBase 8s ESQL/C 使用 DATETIME 列值和其限定符初始化 datetime 主机变量。

零（0）是无效的限定符。因此，如果将 **dt_qual** 字段设置为零，则可以确保 GBase 8s ESQL/C 使用 DATETIME 列的限定符。

对于插入，GBase 8s ESQL/C 不能执行插入或修改操作。

GBase 8s ESQL/C 将 SQLSTATE 状态变量设置为一个错误类代码（SQLCODE 设置为负值），并且 DATETIME 列上的修改和插入操作失败。

获取和插入到 interval 主机变量

当应用程序使用 **interval** 主机变量获取或插入 INTERVAL 值时，GBase 8s ESQL/C 必须在 **interval** 主机变量中找到一个有效的限定符。GBase 8s ESQL/C 根据与主变量相关联的 **intrvl_t** 结构中的 **in_qual** 字段的值，执行以下操作之一：

当 in_qual 字段包含有效限定符时，GBase 8s ESQL/C 检查它与来自 INTERVAL 列值的限定符的一致性。

如果它们属于同一 interval 类 **year to month** 或 **day to fraction**，则这两个限定符是符合的。如果限定符不符合，则 GBase 8s ESQL/C 将 SQLSTATE 状态变量设置为错误类代码（以及 SQLCODE 设置为负值），并且选择、修改或插入操作失败。

如果限定符符合但不一样，则 GBase 8s ESQL/C 扩展列的值以符合 **in_qual** 限定符。扩展时添加或删除 INTERVAL 的 interval 类值的字段以使其与给定的限定符匹配的操作，可以使用 GBase 8s ESQL/C**invextend()** 函数显式扩展 INTERVAL 值。

当 in_qual 字段不包含有效的限定符时，GBase 8s ESQL/C 对获取或插入采取不同的操作：

对于获取，如果 in_qual 字段包含零或是一个无效的限定符，则 GBase 8s ESQL/C 使用 INTERVAL 列值及其限定符初始化 interval 主机变量。

对于插入，如果 in_qual 字段与 INTERVAL 列不一致或如果它没有包含有效值，则 GBase 8s ESQL/C 不会执行插入或修改操作。

GBase 8s ESQL/C 将 SQLSTATE 状态变量设置为一个错误类代码（SQLCODE 设置为负值），并且 INTERVAL 列上的修改和插入操作失败。

隐式数据转换

可以将 DATETIME 或 INTERVAL 列值获取到字符（**char** 、**string** 或 **fixchar**）主机变量。 GBase 8s ESQL/C 在它将其存储在字符主机变量之前将 DATETIME 或 INTERVAL 列值主或为字符字符串。该字符字符串符合 DATETIME 和 INTERVAL 值的 ANSI SQL 标准。如果主机变量太短，则 GBase 8s ESQL/C 将 **sqlca.sqlwarn.sqlwarn1** 设置为 W，使用星号（*）填充主机变量，并将任何指示变量设置到不需截断字符字符串的长度。

还可以从字符（**char** 、**string** 、 **fixchar** 或 **varchar**）主机变量插入 DATETIME 或 INTERVAL 列值。GBase 8s ESQL/C 使用列值的数据类型和限定符将字符值转换为 DATETIME 或 INTERVAL 值。它希望字符串包含符合 ANSI SQL 标准的 DATETIME or INTERVAL 值。

如果转换失败，则 GBase 8s ESQL/C 将 SQLSTATE 状态变量设置为一个错误类代码（SQLCODE 设置为负值），并且修改和插入操作失败。

重要： GBase 8s 产品不支持从 DATETIME 和 INTERVAL 列值到数字（**double** 、**int** 等）主变量的自动数据转换。GBase 8s 产品也不支持从数字（**double** 、**int** 等）或 **date** 主机变量到 DATETIME 和 INTERVAL 列值的自动数据转换。

DATETIME 和 INTERVAL 值的 ANSI SQL 标准

ANSI SQL 标准指定 DATETIME 和 INTERVAL 值的字符表示法的限定符和格式。DATETIME 值的标准限定符是 YEAR TO SECOND，标准格式如下：

YYYY-MM-DD HH:MM:SS

INTERVAL 值的标准指定下列两种间隔类：

YEAR TO MONTH 类具有格式：YYYY-MM

该格式的子集也有效：例如，只有月间隔。

DAY TO FRACTION 类具有格式：DD HH:MM:SS.F

连续字段的任何子集也是有效的：例如，MINUTE TO FRACTION。

转换 datetime 值的数据

可以使用 GBase 8s ESQL/C 库函数 dtcvasc() 、dtcvfmtasc() 、dttoasc() 和 dttofmtasc() 显式在 DATETIME 列值和字符串之间转换。

例如，可以使用 GBase 8s ESQL/C 库函数和中间字符串在 DATETIME 和 DATE 日期类型之间执行转换。

将 DATETIME 值转换为 DATE 值：

使用 dtextend() 函数将 DATETIME 限定符调整为 year to day。

应用 dttoasc() 函数创建 yyyy-mm-dd 格式的字符字符串。

使用模式参数 yyyy-mm-dd 的 rdefmtdate() 函数将字符串转换为 DATE 值。

转换 interval 值的数据

可以使用 GBase 8s ESQL/C 库函数 incvasc() 、incvfmtasc() 、intoasc() 和 intofmtasc() 显式在 INTERVAL 列值和字符字符串之间转换。

例如，可以使用 GBase 8s ESQL/C 库函数和中间字符串在 DATETIME 和 DATE 日期类型之间执行转换。

将 DATE 值转换为 DATETIME 值：

声明具有限定符 year to day 的主机变量（或使用 TU_DTENCODE(TU_YEAR,TU_DAY) 宏返回的值初始化限定符）。

使用具有 yyyy-mm-dd 模式的 rfmtdate() 函数将 DATE 值转换为字符字符串。

使用 dtcvasc() 函数将字符字符串转换为之前预备好的 DATETIME 变量中的值。

如有必要，使用 dtextend() 函数调整 DATETIME 限定符。

2.6.4 支持非 ANSI DATETIME 格式

GBase 8s ESQL/C 支持从非 ANSI 格式的数据时间字符串到 DATETIME 数据类型的转换。此转换可以更轻松地从亚洲语言支持（ALS）客户端/服务器升级到全球语言支持（GLS）客户端/服务器产品。

USE_DTENV 环境变量

GBase 8s ESQL/C 使用 USE_DTENV 环境变量支持非 ANSI 日期时间格式。

当启用 USE_DTENV 环境变量是，使用以下顺序或优先级：

DBTIME

GL_DATETIME

CLIENT_LOCALE

LC_TIME

LANG （如果未设置 LC_TIME）

ANSI 格式

启用时，USE_DTENV 环境变量从 ESQL/C 程序传递到数据库服务器。为数据库服务器启用它不起作用。您必须将其设置为 ESQL/C 客户端应用程序，然后将其传递给数据库服务器。

如果数据库服务器不支持非 ANSI 日期-时间格式，那么不要为 ESQL/C 客户端程序设 USE_DTENV 环境变量。

必须设置此环境变量以在使用非缺省语言环境的数据库中正确显示本地化的 DATETIME 值，并且 GL_DATETIME 环境变量具有非缺省值。

2.6.5 DATETIME 和 INTERVAL 库函数

必须对 **datetime** 和 **interval** 数据类型使用以下 GBase 8s ESQL/C 库函数才能对这些类型的值执行所有操作，GBase 8s ESQL/C 中提供以下 C 函数来处理 **datetime** 和

interval 主机变量。

函数名称	描述	请参阅
dtaddinv()	将 interval 值添加到 **datetime** 值	dtaddinv() 函数
dtcurrent()	获得当前日期和时间	dtcurrent() 函数
dtcvasc()	将符合 ANSI 的字符字符串转换为 **datetime** 值	dtcvasc() 函数
dtcvfmtasc()	将具有特定格式的字符字符串转换为 **datetime** 值	dtcvfmtasc() 函数
dtextend()	更改 datetime 值的限定符	dtextend() 函数
dtsub()	从另一个 datetime 值减去一个 datetime 值	dtsub() 函数
dtsubinv()	从 **datetime** 值减去间隔值	dtsubinv() 函数
dttoasc()	将 datetime 值转换为符合 ANSI 的字符字符串	dttoasc() 函数
dttofmtasc()	将 datetime 值转换为具有特定格式的字符字符串	dttofmtasc() 函数
incvasc()	将符合 ANSI 的字符字符串转换为 **interval** 值	incvasc() 函数
incvfmtasc()	将具有特定格式的字符字符串转换为 **interval** 值	incvfmtasc() 函数
intoasc()	将 interval 值转换为符合 ANSI 的字符字符串	intoasc() 函数
intofmtasc()	将 interval 值转换为具有特定格式的字符字符串	intoasc() 函数
invdivdbl()	将 interval 值除以数值	intofmtasc() 函数
invdivinv()	将 interval 值除以另一个 **interval** 值	invdivdbl() 函数
invextend()	将 interval 值扩展到不同的 **interval** 限定符	invdivinv() 函数
invmuldbl()	将 interval 值乘以数值	invextend() 函数

2.7 简单大对象

简单大对象是存储在磁盘上的 blobspace 中的大对象，它是不可恢复的。

简单大对象包括 TEXT 和 BYTE 数据类型。TEXT 数据类型存储任何文本数据。BYTE 数据类型可以存储任何不同字节流的二进制数据。

这些主题介绍了有关简单大对象的下列信息：

在 GBase 8s ESQL/C 应用程序中选择使用简单对象还是智能大对象

使用简单大对象编程，包括如何声明主机变量以及如何使用定位结构

定位内存中的简单大对象

定位文件中的简单大对象，包括打开的文件和已命名的文件

在用户定义的位置找到简单大对象

本节的结尾介绍了一个名为 **dispcat_pic** 的注释示例程序。**dispcat_pic** 示例程序演示如何从 **stores7** 演示数据库的 **catalog** 表中读取和显示 **cat_descr** 和 **cat_picture** 简单大对象列。

2.7.1 选择大对象数据类型

如果使用 GBase 8s 作为您的数据库服务器，则可以在使用简单大对象或智能大对象之间进行选择。

当您编写新的需要访问大对象的应用程序时，使用智能大对象保存字符（CLOB）和二进制（BLOB）数据。

下表总结了智能大对象对于简单大对象的优势：

大对象功能	简单大对象	智能大对象
数据的最大大小	2 GB	4 TB
数据的可访问性	不能随机访问数据	随机访问数据
读取大对象	数据库服务器根据全部方式或不基于方式读取一个简单大对象	库函数提供类似于访问操作系统文件的访问。您可以访问智能大对象的指定部分。
写入大对象	数据库服务器以全部基础或无基础更改一个简单大对象	数据库服务器只能重写一部分智能大对象
数据日志记录	总是数据日志记录	可以打开或关闭数据日志记

大对象功能	简单大对象	智能大对象
		录

2.7.2 使用简单大对象编程

GBase 8s ESQL/C 支持 SQL 简单大对象和使用 loc_t 数据类型的数据类型 TEXT 和 BYTE 。

提示： 不能在 INSERT 或 UPDATE 语句中使用文字值将简单大对象放入 TEXT 或 BYTE 列。要将值插入到一个简单大对象中，可以使用来自 GBase 8s ESQL/C 客户端应用程序的 DB-Access 的 LOAD 语句或 **loc_t**主机变量。

由于简单大对象的潜在的巨大大小，则 GBase 8s ESQL/C 程序不会直接在 **loc_t** 主机变量中存储数据。相反，**loc_t** 结构是定位程序结构。它不包含实际数据，它包含简单大对象的大小和位置信息。您选择是否将数据存储在内存中、操作系统文件中或用户定义的位置。

要在 GBase 8s ESQL/C 程序中使用简单大对象，请采取以下操作：

声明具有 loc_t 数据类型的主机变量

访问 loc_t 定位器结构的字段

为简单大对象声明主机变量

使用 **loc_t** 数据类型为 TEXT 或 BYTE 类型的数据库值声明主机变量。为数据类型 **loc_t** 的简单大对象列声明一个主机变量，如下所示：

```
EXEC SQL include locator;
        :

        EXEC SQL BEGIN DECLARE SECTION;
        loc_t text_lob;
        loc_t byte_lob;
        EXEC SQL END DECLARE SECTION;
```

具有 TEXT 数据类型的定位器变量具有定位器结的 **loc_type** 字段设置为 SQLTEXT。对于 BYTE 变量，**loc_type** 是 SQLBYTE。

提示： sqltypes.h 头文件定义 SQLTEXT 和 SQLBYTE。因此，请确保在使用这些常

量之前包含了 sqltypes.h 。

从 GBase 8s ESQL/C 程序中，可以选择并将简单大对象数据插入到 **loc_t** 主变量中。您也可以在简单大对象列名称上仅选择具有下标的简单大对象变量的部分。这些下标可以编码到语句中，如下所示：

```
EXEC SQL declare catcurs cursor for
        select catalog_num, cat_descr[1,10]
        from catalog
        where manu_code = 'HSK';
        EXEC SQL open catcurs;
        while (1)
        {
        EXEC SQL fetch catcurs into :cat_num, :cat_descr;

        ⋮

        }
```

下标也可以作为输入参数传递，如以下代码片段所示：

```
EXEC SQL prepare slct_id from
        'select catalog_num, cat_descr[?,?] from catalog \
        where catalog_num = ?'
        EXEC SQL execute slct_id into :cat_num, :cat_descr
        using :n, :x, :cat_num;
```

访问定位器结构

在 GBase 8s ESQL/C 程序中，使用定位器结构访问简单大对象值。

当它们存储在数据库中或从数据库中检索时，定位器结构是 TEXT 和 BYTE 列的主机变量。此结构描述了以下两个数据库操作的简单大对象的位置：

当程序向数据库插入简单对象时，定位器结构标识要插入的简单大对象的源。

建议在使用数据结构之前进行初始化，如下所示：

```
byfill(&blob1, sizeof(loc_t), 0);
            where blob1 is declared as --
```

```
EXEC SQL BEGIN DECLARE SECTION;
loc_t blob1;
EXEC SQL END DECLARE SECTION;
```

这样可以确保数据结构的所有变量都初始化，并避免不一致。

当程序从数据库中选择简单大对象时，定位器结构标识简单大对象的目的地。

locator.h 头文件定义名为 loc_t 的定位器结构。下图显示了来自 locator.h 文件中 loc_t 定位器结构的声明。

图 1. locator.h 头文件中 loc_t 的声明

```
typedef struct tag_loc_t
        {
        int2 loc_loctype;           /* USER: type of locator - see below      */
        union                       /* variant on 'loc'
*/
        {
        struct                      /* case LOCMEMORY
*/
        {
        int4    lc_bufsize;     /* USER: buffer size                          */
        char *lc_buffer;        /* USER: memory buffer to use                 */
        char *lc_currdata_p;    /* INTERNAL: current memory buffer            */
        mint    lc_mflags;      /* USER/INTERNAL: memory flags
*/
        /*                          (see below)        */
        } lc_mem;

        struct                      /* cases L0CFNAME & LOCFILE
*/
        {
        char *lc_fname;         /* USER: file name                            */
        mint    lc_mode;        /* USER: perm. bits used if creating          */
        mint    lc_fd;          /* USER: os file descriptior                  */
```

```
                int4    lc_position;        /* INTERNAL: seek position                    */
                } lc_file;
                } lc_union;

                int4    loc_indicator;          /* USER/SYSTEM: indicator
*/
                int4    loc_type;               /* SYSTEM: type of blob
*/
                int4    loc_size;               /* USER/SYSTEM: num bytes in blob or -1
*/
                mint    loc_status;             /* SYSTEM: status return of locator ops
*/
                char *loc_user_env;             /* USER: for the user's PRIVATE use
*/
                int4    loc_xfercount;          /* INTERNAL/SYSTEM: Transfer count
*/
                /* USER: open function                                  */
                mint (*loc_open)(struct tag_loc_t *loc, mint flag, mint bsize);
                ;               /* USER: close function                         */
                mint (*loc_close)(struct tag_loc_t *loc)
                ;               /* USER: read function                          */
                mint (*loc_read)(struct tag_loc_t *loc, char *buffer, mint buflen)
                ;               /* USER: write function                         */
                mint (*loc_write)(struct tag_loc_t *loc, char *buffer, mint buflen)
                /* USER/INTERNAL: see flag definitions below */
                mint    loc_oflags;
                } loc_t;
```

在图 1中，locator.h 文件中的以下示例说明了如何在定位器结构中使用字段。

USER

GBase 8s ESQL/C 程序设置此字段，GBase 8s ESQL/C 库检查此字段。

SYSTEM

GBase 8s ESQL/C 库设置此字段，GBase 8s ESQL/C 程序检查此字段。

INTERNAL

此字段是 GBase 8s ESQL/C 库的工作区，GBase 8s ESQL/C 程序不需要检查此字段。

GBase 8s ESQL/C 不会在 GBase 8s ESQL/C 程序中自动包含 locator.h 头文件。您必须在定义简单大对象变量的任何 GBase 8s ESQL/C 程序中包含 locator.h 头文件。

EXEC SQL include locator;

定位器结构的字段

定位器结构具有以下部分：

loc_loctype 字段标识简单大对象的位置。它还标识 lc_union 结构的可变类型。

lc_union 结构是 union （重叠的可变结构）结构。

使用的变量取决于 GBase 8s ESQL/C 可以在运行时找打简单大对象的位置。
所有类型的简单大对象变量都有几个字段。

列出所有简单大对象位置中常见的定位器结构中的字段。

表 1. 简单大对象位置中常见的定位器结构中的字段

字段	数据类型	描述
loc_indicator	4 字节整数	**loc_indicator** 字段的值为 -1 时表示空简单大对象。GBase 8s ESQL/C 程序可以将此字段设置为指示插入空值；GBase 8s ESQL/C 库将其设置为 select 或 fetch。 为了在各种平台上保持一致的行为，建议将指标值设置为 0 或 -1。如果指示符未设置，您可能会遇到不一致的行为。设置时，指示字段中设置的值优先。 还可以使用 **loc_indicator** 字段来指示程序在内存中选择的错误。如果要检索的简单大对象不适合提供的空间，则 **loc_indicator** 字段包含简单大对象的实际大小。
loc_size	4 字节整数	包含简单大对象的大小（以字节为单位）。 该字段指示 GBase 8s ESQL/C 库读取或写入的简单大对象的数据量。GBase 8s ESQL/C 程序在数据库中插入一个简单大对象时，设置 GBase 8s ESQL/C 库在选择或获取一个简单大对象后设置了 **loc_size**。
loc_status	mint	指示最后一个定位器操作的状态。当定位器操作成功时，GBase 8s ESQL/C 库将 **loc_status** 设置为零，当发生错误时设置为负数。 SQLCODE 变量还包含此状态值。

字段	数据类型	描述
loc_type	4 字节整数	指定变量的数据类型是 TEXT（SQLTEXT）还是 BYTE（SQLBYTES）。sqltypes.h 头文件定义 SQLTEXT 和 SQLBYTES。

简单大对象数据的位置

在 GBase 8s ESQL/C 程序访问简单大对象列之前，它必须确定简单大对象的位置。

要指定简单大对象是位于内存或文件中，请指定定位器结构的 **loc_loctype** 字段的内容。下表显示了简单大对象数据的可能位置。

表 2. 简单大对象数据的可能位置

loc_loctype 字段的值	简单大对象数据的位置	请参阅
LOCMEMORY	在内存中	在内存中找到简单大对象
LOCFILE	在打开的文件中	在打开文件中定位简单大对象
LOCFNAME	在已命名的文件中	在已命名的文件中定位简单大对象
LOCUSER	在用户定义的位置	用户定义的简单大对象位置

在声明定位器变量之后，在此声明的变量接收到简单大对象值之前，请设置 **loc_loctype**。

locator.h 头文件定义 LOCMEMORY 、LOCFILE 、LOCFNAME 和 LOCUSER 位置常量。在您的 GBase 8s ESQL/C 程序中，当将值分配给 **loc_loctype** 时，请使用这些常量名称而不是他们的常量值。

在客户端服务器环境中，GBase 8s ESQL/C 在客户端计算机（运行应用程序的计算机）上找到简单大对象。

2.7.3 在内存中找到简单大对象

要使 GBase 8s ESQL/C 在内存中定位 TEXT 或 BYTE 数据，请将定位器结构的字段 **loc_loctype** 设置为 LOCMEMORY ，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        loc_t my_simple_lo;
        EXEC SQL END DECLARE SECTION;
        ⋮

        my_simole_lo.loc_loctype = LOCMEMORY;
```

当使用内存作为简单大对象的位置，则定位器结构使用 **lc_union** 结构的 **lc_mem** 结构。下表介绍了 **lc_union.lc_mem** 字段。

表 3. 用于内存中的简单大对象的 lc_union.lc_mem 结构中字段

字段	数据类型	描述
lc_bufsize	4 字节整数	**lc_buffer** 字段指向的缓冲区的大小（以字节为单位）
lc_buffer	char *	保存简单大对象值缓冲区的地址。GBase 8s ESQL/C 程序必须为此缓冲区分配空间并将它的地址存储在 **lc_buffer** 中。
lc_currdata_p	char *	系统缓冲区的地址。这是内部字段并且不能被 GBase 8s ESQL/C 程序修改。
lc_mflags	mint	当分配内存时使用的标志。

当您访问 **lc_union.lc_mem** 中的字段时，locator.h 文件提供以下宏结：

```
#define loc_bufsize          lc_union.lc_mem.lc_bufsize
        #define loc_buffer           lc_union.lc_mem.lc_buffer
        #define loc_currdata_p       lc_union.lc_mem.lc_currdata_p
        #define loc_mflags           lc_union.lc_mem.lc_mflags
```

提示： 建议在访问定位器结构时使用这些快捷方式名称。快捷方式名称提供了代码的可读性，并减少了编码错误。该引用在引用 **lc_union.lc_mem** 结构的 **lc_bufsize** 、**lc_buffer** 、**lc_currdata_p** 和 **lc_mflags** 字段时使用这些快捷方式名称。

demo 命令包含以下两个示例 GBase 8s ESQL/C 程序，它们演示了如何处理位于内存中的简单大对象数据：

getcd_me.ec 程序在内存中选择一个简单大对象。

updcd_me.ec 程序从内存中插入一个简单大对象。

这些程序假设 **stores7** 数据库作为简单大对象数据的缺省数据库。用户可以指定另一个数据库（在缺省的数据库服务器上）作为命令行参数。

getcd_me mystores

分配内存缓冲区

当查询将简单大对象选择到内存中，则 GBase 8s ESQL/C 使用内存缓冲区。

在程序获取 TEXT 或 BYTE 数据之前，必须将 **loc_bufsize**（**lc_union.lc_mem.lc_bufsize**）字段设置如下，以指示 GBase 8s ESQL/C 如何分配此内存缓冲区：

如果将 loc_bufsize 设置为 -1，则 GBase 8s ESQL/C 分配内存缓冲区保存简单大对象数据。

如果将 loc_bufsize 设置为不是 -1 的值，则 GBase 8s ESQL/C 假定程序处理内存缓冲区分配和撤销分配。

重要： 当在内存中找到简单大对象时，必须始终将 loc_mflags（lc_union.lc_mem.lc_mflags）和 loc_oflags 设置为 0 。

ESQL/C 库分配的内存缓冲区

当将 **loc_bufsize** 设置为 -1 时，GBase 8s ESQL/C 对获取和选择分配内存和缓冲区。GBase 8s ESQL/C使用 malloc() 系统调用内存缓冲区保存简单大对象（如果它不能分配缓冲区，GBase 8s ESQL/C 将 **loc_status** 字段设置为 -465 以指示错误）当选择（或第一次获取）完成时，GBase 8s ESQL/C 将 **loc_buffer** 设置为缓冲区的地址。将 **loc_bufsize** 和 **loc_size** 设置为获取的简单大对象的大小以更改定位器结构。

要获取具有较大或较小数据的连续简单大对象，**loc_mflags** 设置为 LOC_ALLOC 常量（locator.h 定义），以请求 GBase 8s ESQL/C 重新分配新的内存缓冲区。将 **loc_bufsize** 设置为当前已分配缓冲区的大小。

如果在初始提取时未设置 **loc_mflags** 为 LOC_ALLOC ，则 GBase 8s ESQL/C 不会释放其为 **loc_buffer** 缓冲区分配的内存。相反，它为后续的提取分配一个新的缓冲区。这种情况可能会导致每个提取的程序大小增加，除非明确释放分配给每个 **loc_buffer** 缓冲区的内存。如果您的应用程序在 Windows(TM) 操作系统上运行并使用多线程库，则使用 SqlFreeMem()GBase 8s ESQL/C 函数将其释放。否则使用 free() 系统调用。

当将 **loc_mflags** 设置为 LOC_ALLOC 时，GBase 8s ESQL/C 如下处理内存分配：

如果简单大对象数据大小增加，则 GBase 8s ESQL/C 释放现有的缓冲区并分配重要的内存。

如果发生这种重新分配，则 GBase 8s ESQL/C 改变存储简单大对象数据的存储器地址。因此，如果您在程序中引用地址，则程序逻辑必须考虑地址更改。GBase 8s ESQL/C 还将 **loc_bufsize** 和 **loc_size** 字段更改为已获取的简单大对象的大小。

如果数据的大小减少，则 GBase 8s ESQL/C 不需要重新分配缓冲区。

获取之后，**loc_size** 字段指示所获取的简单大对象的大小，而 **loc_bufsize** 字段仍然包含已分配的缓冲区的大小。

当 GBase 8s ESQL/C 获取下一个简单大对象值时，它会释放已分配的内存。因此，GBase 8s ESQL/C 不会显式释放所获取的最后一个简单大对象值，直到您的程序从数据库服务器断开连接。

程序分配的内存缓冲区

如果您想要为简单大对象处理自己的内存分配，请使用 malloc() 系统调用分配内存然后在定位器结构中设置一下字段：

在获取或选择 TEXT 或 BYTE 列之前，将 loc_buffer 字段设置为已分配内存缓冲区的地址，并将 loc_bufsize 字段设置为内存缓冲区的大小。

在插入 TEXT 或 BYTE 列之前，为选择和获取设置相同的字段。此外，将 loc_size 设置为要插入到数据库中数据的大小。

如果获取的数据不适合已分配的缓冲区，则 GBase 8s ESQL/C 库将 **loc_status**（和 SQLCODE）设置为负值（-451）并将数据的实际大小放在 **loc_indicator** 中。如果已获取的数据不适合，则 GBase 8s ESQL/C 将 **loc_size** 设置为已获取数据的大小。

重要：当分配您自己的内存缓冲区时， 也可以在选择或插入简单大对象完成后释放内存缓冲区。GBase 8s ESQL/C 不会释放此内存，因为它无法确定何时完成内存。由于您已经使用 malloc() 分配内存，可以使用 free() 系统调用以释放内存。

选择简单大对象到内存中

demo 目录中的 getcd_me 示例程序显示了如何从数据库中的选择一个简单大对象到内存中。

下图显示了将 catalog 表中的 cat_descr TEXT 列选择到内存中的代码片段，然后显示它。

图 2. getcd_me 示例程序代码片段

```
cat_descr.loc_loctype = LOCMEMORY;          /* set loctype for in memory */
```

```
cat_descr.loc_bufsize = -1;                    /* let db get buffer */
cat_descr.loc_oflags = 0;                      /* clear loc_oflags */
cat_descr.loc_mflags = 0;                      /* set loc_mflags to 0 */
EXEC SQL select catalog_num, cat_descr    /* look up catalog number */
into :cat_num, :cat_descr from catalog
where catalog_num = :cat_num;
if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100)         /* if not found
*/
{
printf("\nCatalog number %ld not found in catalog table\n",
cat_num);
if(!more_to_do())                                  /* More to do? */
break;                                  /* no, terminate loop */
else
continue;                               /* yes */
}
if(ret < 0)
{
printf("\nSelect for catalog number %ld failed\n", cat_num);
EXEC SQL disconnect current;
printf("GETCD_ME Sample Program over.\n\n");
exit(1);
}
prdesc();                                      /* if found, print cat_descr */
```

该程序将 **cat_descr** 定位器结构字段设置如下：

loc_loctype 字段设置为 LOCMEMORY ，以便 GBase 8s ESQL/C 返回内存缓冲区中 cat_descr 文本。

loc_bufsize 字段设置为 -1 ，以使 GBase 8s ESQL/C 为此缓冲区分配内存。

loc_oflags 字段设置为 0 ，因为程序不会对简单大对象使用文件。

当您在内存中定位简单大对象时，必须始终将 loc_mflags 字段设置为 0。

SELECT 或 FETCH 语句之后，定位器结构包含下列信息：

loc_buffer 字段包含内存缓冲区的地址。

loc_bufsize 字段包含 loc_buffer 缓冲区的大小。它是为简单大对象分配内存的总量。

loc_size 字段包含 loc_buffer 中简单大对象数据的字节数。

如果选择的简单大对象为空，则 loc_indicator 字段包含 -1。

loc_status 字段包含操作的状态：0 表示成功，负数表示发生失败。

如果程序选择了第二个简单大对象，则它需要在第二条 SELECT 语句阻止内存泄露之前将 loc_mflags 设置为 LOC_ALLOC 常量。

该代码片段显示了用户输出的目录号的 cat_descr 列。下图显示了 stores7 演示数据库中 cat_descr 列的用户输入和输出。

图 3. getcd_me 示例程序的样本输出

```
GETCD_ME Sample ESQL Program running.

        Connected to stores7

        This program requires you to enter a catalog number from the catalog
        table. For example: '10001'. It then displays the content of the
        cat_descr column for that catalog row. The cat_descr value is stored
        in memory.

        Enter a catalog number: 10004
        Description for 10004:

        Jackie Robinson signature glove. Highest professional quality,
        used by National League.

        **** More? (y/n) …
```

从内存插入简单大对象

demo 目录中的 updcd_me 示例程序显示了如何将内存中的简单大对象插入到数据库中。

程序从包含用户输入的文本的内存缓冲区更改 catalog 表的 cat_descr TEXT 列。下图

显示了用户更新 stores7 数据库的 cat_descr 列的示例输出。

图 4. updcd_me 示例程序的示例输出

```
Enter catalog number: 10004
        Description for 10004:

        Jackie Robinson signature ball. Highest professional quality,
        used by National League.

        Update this description? (y/n) …y

        Enter description (max 255 chars):and press RETURN
        Jackie Robinson home run ball, signed, 1955.

        *** Update complete.
        **** More?(y/n).... n
```

下图显示了一个代码摘录，说明了 updcd_me 程序如何使用定位器结构从存储在内存中的文本更改 cat_descr 列。

图 5. updcd_me 示例程序的代码摘录

```
/* Update? */
        ans[0] = ' ';
        while((ans[0] = LCASE(ans[0])) != 'y' && ans[0] != 'n')
        {
        printf("\nUpdate this description? (y/n) …");
        getans(ans, 1);
        }
        if(ans[0] == 'y')                       /* if yes */
        {
        printf("Enter description (max of %d chars) and press RETURN\n",
```

```
BUFFSZ - 1);
/* Enter description */
getans(ans, BUFFSZ - 1);
cat_descr.loc_loctype = LOCMEMORY;    /* set loctype for in memory */
cat_descr.loc_buffer = ans;                /* set buffer addr */
cat_descr.loc_bufsize = BUFFSZ;          /* set buffer size */
/* set size of data *
cat_descr.loc_size = strlen(ans);
/* Update */
EXEC SQL update catalog
set cat_descr =:cat_descr
where catalog_num = :cat_num;

⋮

}
```

该程序将 **cat_descr** 定位器结构设置如下：

loc_loctype 字段设置为 LOCMEMORY 以便 GBase 8s ESQL/C 从缓冲区读取 cat_descr 文本。

loc_buffer 字段设置为 ans，保存要插入的简单大对象值的内存缓冲区的地址。

loc_bufsize 字段设置为 BUFFSZ，分配 ans 内存缓冲区的大小。

loc_size 字段设置为 strlen(ans) + 1，当前保存的简单大对象的内存缓冲区中的字节数。

如果插入空简单大对象值，您的程序也需要将 **loc_indicator** 字段设置为 -1。

下图显示了一段代码摘录，说明了如何在 INSERT 语句中使用定位器结构。

图 6. 来自主内存的 INSERT 操作示例

```
char photo_buf[BUFFSZ];

EXEC SQL BEGIN DECLARE SECTION;
```

```
char name[20];
loc_t photo;
EXEC SQL END DECLARE SECTION;

photo.loc_loctype = LOCMEMORY; /* Photo resides in memory */
photo.loc_buffer = photo_buf; /* pointer to where it is */
photo.loc_size = BUFFSZ - 1;                    /* length of image*/

EXEC SQL insert into employee (name, badge_pic)
values (:name, :photo);
```

UPDATE 或 INSERT 语句之后，GBase 8s ESQL/C 从内存缓冲区读取的字节数更改 loc_size 字段，并将它发送给数据库服务器。它还设置 loc_status 字段以显示操作的状态：0 表示成功，如果发生错误则为负值。

2.7.4 定位文件中的简单大对象

可以定位打开或已命名类型的文件中定位简单大对象。

打开的文件，是一个在程序访问简单大对象数据之前已经打开的文件。程序提供文件描述符作为简单大对象数据的位置。

已命名的文件是程序还未打开的文件。程序提供文件名称作为简单大对象数据的位置。

当使用文件作为简单大对象位置时，定位器结构为 **lc_union** 结构使用 **lc_file** 结构。下表介绍了 **lc_union.lc_file** 字段。

表 4. 用于位于文件中的简单大对象 lc_union.lc_file 结构中的字段

字段	数据类型	描述
lc_fname	char *	包含简单大对象数据文件的路径名地址。当程序使用已命名的文件作为简单大对象位置，则设置此字段。
lc_mode	int	用于创建文件的权限位。该值是传递给系统 **open()** 函数的第三个参数。有关 **lc_mode** 的有效值，请参阅您的系统文档。
lc_fd	int	包含简单大对象数据文件的文件描述符。程序在使用打开文件时设置此字段。
lc_position	4 字节整数	当前在打开的文件中查找位置。这是一个内部字段，不能由 ESQL/C 程序修改。

当您访问文件中的简单大对象时，locator.h 文件提供以下宏定义快捷方式以供使用：

```
#define loc_fname          lc_union.lc_file.lc_fname
#define loc_fd             lc_union.lc_file.lc_fd
#define loc_position       lc_union.lc_file.lc_position
```

提示： 建议在访问定位器结构时使用这些快捷方式名称。快捷方式名称提供了代码的可读性，并减少了编码错误。该引用在引用 **lc_union.lc_file** 结构的 **lc_fname** 、**lc_fd** 和 **lc_position** 字段时使用这些快捷方式名称。

文件打开模式标志

当对简单大对象使用文件时，还将设置定位器结构的 **loc_oflags** 字段。**loc_oflags** 字段是 **integer** 类型并且它包含主系统文件打开模式标志。

这些标志确定文件打开后如何被访问：

LOC_RONLY 是只读模式的掩码。当向文件中插入简单大对象时使用此值。

LOC_WONLY 是只写模式的掩码。在文件中选择一个简单大对象并且希望每个选定的简单大对象都可以写入任何现有数据时使用此值。

LOC_APPEND 是写入模式的掩码。在文件中选择一个简单大对象并且想要将值附加到文件的末尾时使用此值。

在 loc_status 中返回错误

当 GBase 8s ESQL/C 打开文件时，其中一个标志将传递给 loc_open() 函数。GBase 8s ESQL/C 读取数据并将其写入文件中的当前位置（**loc_position** 字段指示）。如果 GBase 8s ESQL/C 无法读取或写入文件，它将定位器结构的 **loc_status** 字段设置为 -463 或 -464。如果 GBase 8s ESQL/C 无法关闭文件，它将 **loc_status** 设置为 -462。GBase 8s ESQL/C 使用相同的值更改 SQLCODE 变量。

在打开文件中定位简单大对象

要使 GBase 8s ESQL/C 在打开文件中定位 TEXT 或 BYTE 数据，将定位器结构的 **loc_loctype** 字段设置为 LOCFILE。

```
EXEC SQL BEGIN DECLARE SECTION;
        loc_t my_simple_lo;
        EXEC SQL END DECLARE SECTION;
        ⋮

        my_simple_lo.loc_loctype = LOCFILE;
```

要使用打开文件作为简单大对象的位置，您的 GBase 8s ESQL/C 程序必须在它访问

简单大对象数据之前打开希望的文件。然后将它的文件描述符存储在定位器结构的 **loc_fd** 字段中以指定此文件作为简单大对象位置。**loc_oflags** 字段还应包含文件打开模式标志来告诉 GBase 8s ESQL/C 在打开文件时如何访问文件。

demo 目录包含以下两个示例 GBase 8s ESQL/C 程序，以演示如何处理打开文件中的简单大对象：

getcd_of.ec 程序将简单大对象选择到打开文件中。

updcd_of.ec 程序从打开文件中插入简单大对象。

这些程序假设 **stores7** 数据库作为简单大对象数据的缺省数据库。用户可以指定另一个数据库（在缺省的数据库服务器上）作为命令行参数：

getcd_of mystores

将简单大对象选择到打开文件中

demo 目录下的 getcd_of 示例程序显示了如何将数据库中的简单大对象选择到打开的文件中。下图显示了代码片段，将 cat_descr 列选择到用户指定的文件中。

图 7. getcd_of 示例程序的代码片段

```
EXEC SQL BEGIN DECLARE SECTION;
        char db_name[30];
        mlong cat_num;
        loc_t cat_descr;
        EXEC SQL END DECLARE SECTION;
        ⋮

        if((fd = open(descfl, O_WRONLY)) < 0)
        {
        printf("\nCan't open file: %s, errno: %d\n", descfl, errno);
        EXEC SQL disconnect current;
        printf("GETCD_OF Sample Program over.\n\n"):
        exit(1);
        }
        /*
        * Prepare locator structure for select of cat_descr
```

```
                    */
                    cat_descr.loc_loctype = LOCFILE;                /* set loctype for open file
*/
                    cat_descr.loc_fd = fd;                          /* load the file descriptor */
                    cat_descr.loc_oflags = LOC_APPEND;              /* set loc_oflags to
append */
                    EXEC SQL select catalog_num, cat_descr          /* verify catalog number */
                    into :cat_num, :cat_descr from catalog
                    where catalog_num = :cat_num;
                    if(exp_chk2("SELECT", WARNNOTIFY) != 100)       /* if not found */
                    printf("\nCatalog number %ld not found in catalog table\n",
                    cat_num);
                    else
                    {
                    if(ret < 0)
                    {

                    ⋮

                    exit(1);
                    }
                    }
```

要为 SELECT 语句准备定位器结构，**getcd_of** 程序设置 **cat_descr** 定位器结构如下：

loc_loctype 字段设置为 LOCFILE 以指示 GBase 8s ESQL/C 将 cat_descr 列的文本放在打开的文件中。

loc_fd 字段设置为 fd 文件描述符以标识打开文件。

loc_oflags 字段设置为 LOC_APPEND 以指定将数据附加到文件中存在的任何数据。

要访问定位器结构的文件描述符（**loc_fd**）字段，**getcd_of** 程序使用名称 **cat_descr.loc_fd**。但是，定位器结构中该字段的实际名称如下所示：

cat_descr.lc_union.lc_file.lc_fd

loc_fd 的快捷方式名称定义为 locator.h 文件中的宏。

在 GBase 8s ESQL/C 将数据写入到打开文件中后，将设置定位器结构的以下字段：

loc_size 字段包含写入打开文件中的字节数。

如果选择的简单大对象为空，则 loc_indicator 字段包含 -1。

loc_status 字段包含操作的状态：0 表示成功，如果发生失败则为负值。

从打开文件插入简单对象

demo 目录中的 **updcd_of** 示例程序显示如何将来自打开文件中的简单大对象插入到数据库中。该程序更新包含一系列记录打开文件的 **catalog** 表的 **cat_descr** TEXT 列；每个都包含目录号和修改相应 **cat_descr** ；列的文件。以下程序假定输入文件具有以下格式：

\10001\

Dark brown leather first baseman's mitt. Specify right-handed or left-handed.

\10002\

Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right- or left-handed.

?

下图显示了一段代码摘录，说明了如何使用定位器结构更新打开文件的 catalog 表的 cat_descr 列。

图 8. updcd_of 示例程序的代码片段

```
EXEC SQL BEGIN DECLARE SECTION;
    mlong cat_num;
    loc_t cat_descr;
    EXEC SQL END DECLARE SECTION;
    ⋮

    if ((fd = open(descfl, O_RDONLY)) < 0) /* open input file */
    {

    ⋮
```

```
}
while(getcat_num(fd, line, sizeof(line)))      /* get cat_num line from file */
{

⋮

printf("\nReading catalog number %ld from file...\n", cat_num);
flpos = lseek(fd, 0L, 1);
length = getdesc_len(fd);
flpos = lseek(fd, flpos, 0);

/* lookup cat_num in catalog table */
EXEC SQL select catalog_num
into :cat_num from catalog
where catalog_num = :cat_num;
if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100) /* if not found */
{
printf("\nCatalog number %ld not found in catalog table.",
cat_num);

⋮

}
/*if found */
cat_descr.loc_loctype = LOCFILE;              /* update from open file */
cat_descr.loc_fd = fd;                        /* load file descriptor */
cat_descr.loc_oflags = LOC_RONLY;             /* set file-open mode (read)
*/
cat_descr.loc_size = length;                  /* set size of simple large obj */

/* update cat_descr column of catalog table */
EXEC SQL update catalog set cat_descr = :cat_descr
```

```
where catalog_num = :cat_num;
if(exp_chk2("UPDATE", WARNNOTIFY) < 0)
{
EXEC SQL disconnect current;
printf("UPDCD_OF Sample Program over.\n\n");
exit(1);
}
printf("Update complete.\n");
}
```

updcd_of 程序打开用户响应提示指定的输入文件（**descfl**），调用 getcat_num() 函数从文件中读取目录号，然后调用 getdesc_len() 函数来确定文本的长度，用于更新 **cat_descr** 列。该程序执行 SELECT 语句来验证 **catalog** 表中是否存在目录号。

如果此号码存在，则 **updcd_of** 程序将按照以下步骤准备定位器结构，以从打开的文件中的文件更新 **cat_descr** ：

loc_loctype 字段设置为 LOCFILE 来告诉 GBase 8s ESQL/C 将从打开的文件更新 cat_descr 列。

loc_fd 字段设置为 fd，即打开输入文件的文件描述符。

loc_oflags 字段设置为 LOC_RONLY，只读模式的打开文件模式标志。

loc_size 字段设置为 length，即 cat_descr 的传入文本的长度。

如果插入空简单大对象值，您的程序还需要将 **loc_indicator** 字段设置为 -1。

updcd_of 程序然后能够执行数据库更新。在 GBase 8s ESQL/C 从打开文件读取数据后，并将它传递给数据库服务器，GBase 8s ESQL/C 将从打开的文件读取的字节数更新 **loc_size** 字段并发送到数据库服务器。 GBase 8s ESQL/C 还会将 **loc_status** 字段设置为指示操作的状态：成功为 0，如果发生失败，则为负值。

在已命名的文件中定位简单大对象

要使 GBase 8s ESQL/C 在已命名的文件中定位 TEXT 或 BYTE 数据，请将定位器结构的 **loc_loctype** 字段设置为 LOCFNAME，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        loc_t my_simple_lo;
        EXEC SQL END DECLARE SECTION;
```

```
        :

        my_simple_lo.loc_loctype = LOCFNAME;
```

要使用已命名文件作为简单大对象位置，GBase 8s ESQL/C 程序必须指定一个指针指向定位器结构的 **loc_fname** 中的文件名称。还必须使用文件打开标志设置 **loc_oflags** 字段来告知 GBase 8s ESQL/C 如何在打开它时访问它。

要打开已命名的文件，GBase 8s ESQL/C 使用 **loc_oflags** 字段指定的模式标志打开 **loc_fname** 字段中命名的文件。如果该文件不存在，则 GBase 8s ESQL/C 创建它。GBase 8s ESQL/C 然后将此打开文件的文件描述符放入 **loc_fd** 字段中，就像您的程序打开此文件。如果 GBase 8s ESQL/C 不能打开此文件，则将 **loc_status** 字段（和 SQLCODE）设置为 -461。当传输完成后， GBase 8s ESQL/C 关闭此文件，释放 **loc_fd** 字段中的文件描述符。

demo 目录包含下列两个示例 GBase 8s ESQL/C 程序，来演示如何处理已命名文件中的简单大对象：

getcd_nf.ec 程序将简单大对象选择到已命名的文件中。

updcd_nf.ec 程序从已命名的文件插入简单大对象。

这些程序假设 **stores7** 数据库作为简单大对象数据的缺省数据库。用户可以指定另一个数据库（在缺省的数据库服务器上）作为命令行参数。

getcd_of mystores

将简单大对象选择到已命名文件中

demo 目录的 **getcd_nf** 示例程序显示了如何将数据库中的简单大对象选择到已命名的文件中。以下代码摘录提示用户输入 **catalog** 表的目录号和程序写入该行的 **cat_descr** 列的内容的文件的名称。该程序将文件的名称存储到 **descfl** 数组中。然后它执行 SELECT 语句从 **catalog** 表中读取 **cat_descr** TEXT 列，并将其写入用户响应提示时指定的文件。

下图显示了来自 getcd_nf 示例程序的代码摘录。

图 9. getcd_nf 示例程序的代码片段

```
EXEC SQL BEGIN DECLARE SECTION;
        char db_name[30];
        mlong cat_num;
```

```
loc_t cat_descr;
EXEC SQL END DECLARE SECTION;
⋮

printf("\nEnter a catalog number: ");       /* prompt for catalog number */
getans(ans, 6);
if(rstol(ans, &cat_num))                     /* cat_num string too long */
{
printf("\tCannot convert catalog number '%s' to integer\n", ans);
continue;
}
while(1)
{
printf("Enter the name of the file to receive the description: ");
if(!getans(ans, 15))
continue;
break;
}
strcpy(descfl, ans);
break;
}

/*
*    Prepare locator structure for select of cat_descr
*/
cat_descr.loc_loctype = LOCFNAME;                /* set loctype for in
memory */
cat_descr.loc_fname = descfl;                    /* load the addr of file name
*/
cat_descr.loc_oflags = LOC_APPEND;               /* set loc_oflags to
append */
EXEC SQL select catalog_num, cat_descr           /* verify catalog number
*/
into :cat_num, :cat_descr from catalog
```

```
where catalog_num = :cat_num;
if(exp_chk2("SELECT", WARNNOTIFY)  !=  0 )    /* if error, display
and quit */
printf("\nSelect for catalog number %ld failed\n", cat_num);

EXEC SQL disconnect current;
printf("\nGETCD_NF Sample Program over.\n\n");
}
```

该程序按以下步骤设置 **cat_descr** 定位器结构：

loc_loctype 字段包含 LOCFNAME 以告知 GBase 8s ESQL/C 将为已命名文件中的 cat_descr 列放置文本。

loc_fname 字段是 descfl 数组的地址以告知 GBase 8s ESQL/C 要将 cat_descr 列的内容写入到 descfl 中的命名的文件。

loc_oflags 字段，文件打开模式标识，将设置为 LOC_APPEND 以告知 GBase 8s ESQL/C 要将选择的数据附加到现有的文件中。

getcd_nf 程序然后执行 SELECT 语句检索行。在 GBase 8s ESQL/C 将数据写入已命名的文件后，它将设置定位器结构的以下字段：

loc_size 字段包含写入到文件中的字节数。如果 GBase 8s ESQL/C 程序将空简单大对象列获取到存在的已命名的文件中，则会截断该文件。

如果选择的简单大对象值为空，则 loc_indicator 字段包含 -1。

loc_status 字段包含操作的状态：成功为 0 ，如果发生错误则为负值。

从已命名的文件插入简单大对象

demo 目录中的 **updcd_nf** 示例程序显示如何将来自打开文件中的简单大对象插入到数据库中。该程序更新包含一系列记录打开文件的 **catalog** 表的 **cat_descr** TEXT 列；每个都包含目录号和修改相应 **cat_descr** ；列的文件。以下程序假定输入文件具有以下格式：

```
Babe Ruth signature glove. Black leather. Infield/outfield
        style. Specify right- or left-handed.
```

下图显示了一段代码摘录，说明了如何使用定位器结构更新打开文件的 catalog 表的 cat_descr 列。

图 10. updcd_nf 示例程序的代码摘录

```
    EXEC SQL BEGIN DECLARE SECTION;
            mlong cat_num;
            loc_t cat_descr;
            EXEC SQL END DECLARE SECTION;
            ⋮

            cat_descr.loc_loctype = LOCMEMORY;          /* set loctype for in
memory */
            cat_descr.loc_bufsize = -1;                 /* let server get memory */
            EXEC SQL select catalog_num, cat_descr      /* verify catalog number */
            into :cat_num, :cat_descr from catalog
            where catalog_num = :cat_num;

            /* if error,display and quit */
            if ((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100)
            {
            printf("\nCatalog number %ld not found in catalog table\n",
            cat_num);
            EXEC SQL disconnect current;
            printf("UPDCD_NF Sample Program over.\n\n");
            exit(1);
            }
            if(ret<0)
            {
            EXEC SQL disconnect current;
            printf("UPDCD_NF Sample Program over.\n\n");
            exit(1);
            }
            prdesc();                                   /* print current cat_descr
*/

            /* Update? */
            ans[0] = ' ';
```

```
while((ans[0] = LCASE(ans[0]))    != 'y' && ans[0]    != 'n')
{
printf("Update this description? (y/n) …");
scanf("%1s", ans);
}
if(ans[0] == 'y')
{
cat_descr.loc_loctype = LOCFNAME;            /* set type to named file */
cat_descr.loc_fname = descfl;                /* supply file name */
cat_descr.loc_oflags = LOC_RONLY;            /* set file-open mode (read) */
cat_descr.loc_size = -1;                     /* set size to size of file */
EXEC SQL update catalog
set cat_descr = :cat_descr                   /* update cat_descr column */
where catalog_num = :cat_num;
if(exp_chk2("UPDATE", WARNNOTIFY) < 0)       /* check status */
{
EXEC SQL disconnect current;
printf("UPDCD_NF Sample Program over.\n\n");
exit(1);
}
printf("Update complete.\n");
}
```

图 1中的 updcd_nf 程序首先在 catalog 上执行一个用户响应提示输入的目录号的 SELECT 语句。SELECT 语句返回 catalog_num 和 cat_descr 列。prdesc() 函数（prdesc.c 文件指南）显示 cat_descr 的当前内容。

然后程序提问用户是否要更改此描述符。如果用户回答是（ans[0] = = 'y'）则 **updcd_nf** 程序按以下步骤准备定位器结构来更改用户指定的文件文本中的 **cat_descr** 列：

cat_descr.loc_loctype 字段设置为 LOCFNAME 以指示更改的文本的源是已命名的文件。

cat_descr.loc_fname 字段设置为 descfl，文件的名称包含简单大对象数据。

cat_descr.loc_oflags 字段设置为 LOC_RONLY 以告知 GBase 8s ESQL/C 以只读模式打开此文件。

cat_descr.loc_size 字段设置为 -1 以告知 GBase 8s ESQL/C 一次性传输简单对象，而不是以较小的部分传输，一次传输一片。还可以将 loc_oflags 字段设置为 LOC_USEALL 掩码来执行此操作。

如果插入空简单大对象值，您的程序还需要将 **loc_indicator** 字段设置为 -1。

GBase 8s ESQL/C 从打开文件读取数据后，会将它传递给数据库服务器，GBase 8s ESQL/C 将从已命名的文件读取的字节数更新 loc_size 字段并发送到数据库服务器。 GBase 8s ESQL/C 还会将 loc_status 字段设置为指示操作的状态：成功为 0，如果发生失败，则为负值。

2.7.5 用户定义的简单大对象位置

可以创建您自己版本的 loc_open() 、loc_read() 、loc_write() 和 loc_close() 函数来为简单大对象定义的位置。

用户定义IDE位置函数的典型用途是在应用程序使用数据之前需要以某种方式解释数据。例如：如果数据被压缩，则在将数据发送到数据库之前，应用程序必须先解压缩。应用程序甚至可以在运行时选择多种不同的解释函数，它只需将适当的函数指针设置为所需的解释函数。

要使 GBase 8s ESQL/C 使用您子句的 C 函数定义 TEXT 或 BYTE 数据位置，将定位器结构的 **loc_loctype** 字段设置为 LOCUSER ，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        ifx_loc_t my_simple_lo;
        EXEC SQL END DECLARE SECTION;
        ⋮

        my_simple_lo.loc_loctype = LOCUSER;
```

使用用户定义的简单大对象位置，定位器结构使用下表总结的字段。

表 5. 用于创建用户定义的位置函数的定位器结构中的字段

字段	数据类型	描述
loc_open	mint (*)()	指向返回整数值的用户定义的打开函数的指针
loc_read	mint (*)()	指向返回整数值的用户定义的读取函数的指针

字段	数据类型	描述
loc_write	mint (*)()	指向返回整数值的用户定义的写入函数的指针
loc_close	mint (*)()	指向返回整数值的用户定义的关闭函数的指针
loc_user_env	char *	保存用户定义的位置函数所需的数据的缓冲区的地址。例如：可以将 loc_user_env 设置为公共工作区的地址
loc_xfercount	4 字节整数	传输简单大对象的最后一次传输操作的字节数

使用用户定义的简单大对象位置，定位器结构使用 lc_union 结构的 lc_mem 结构或 lc_file 结构。表 1和表 1介绍了 lc_union.lc_mem 结构和 lc_union.lc_file 结构的字段。

将简单大对象选择到用户定义的位置

当您的程序选择简单大对象值时，GBase 8s ESQL/C 库必须从数据库服务器接收数据并将它传输给 GBase 8s ESQL/C 程序。要这样做，GBase 8s ESQL/C 执行以下步骤：

在传输之前，GBase 8s ESQL/C 调用用户定义的打开函数以指示用户定义的位置。此打开函数的 *oflags* 参数设置为 LOC_WONLY。

GBase 8s ESQL/C 接收来自数据库服务器的简单大对象值，并将它们放入程序缓冲区。

GBase 8s ESQL/C 调用用户定义的写入函数将程序缓冲区的简单大对象数据传输到用户定义的位置。

GBase 8s ESQL/C 多次重复步骤 2和 3以便将全部的简单大对象值传输到用户定义的位置。

传输之后，GBase 8s ESQL/C 执行清除操作，清除用户定义的关闭函数指定的操作。

要将简单大对象选择到用户定义的位置，将 **loc_loctype** 设置为 LOCUSER 将 **loc_open** 、**loc_write** 和 **loc_close** 字段设置为包含用户定义的打开、写入和关闭函数的地址。

将简单大对象插入到用户定义的位置

当您的程序插入一个简单大对象值时，GBase 8s ESQL/C 库必须将来自 GBase 8s ESQL/C 程序的数据传输到数据库服务器。要做此，GBase 8s ESQL/C 执行以下步骤：

传输之前，GBase 8s ESQL/C 调用用户定义的打开函数初始化用户定义的位置。此打开的函数的 *oflags* 参数设置为 LOC_RONLY。

GBase 8s ESQL/C 定义用户定义的读取函数将简单大对象从用户定义的位置传输到程序缓冲区。

GBase 8s ESQL/C 将程序缓冲区中的值发送到数据库服务器。

GBase 8s ESQL/C 根据需要重复步骤 2和 3，将整个简单大对象值从用户定义的位置传输到数据库服务器。

传输后，GBase 8s ESQL/C 执行在用户定义关闭函数中指定的清除操作。

要插入存储在用户定义位置的简单大对象，请将 **loc_loctype** 设置为 LOCUSER ，并设置 **loc_open** 、**loc_read** 和 **loc_close** 字段，以便它们包含适当的用户定义的打开、读取和关闭函数的地址。如果要插入的简单大对象为空，请将 **loc_indicator** 字段设置为 -1。

将 **loc_size** 字段设置为插入的简单大对象数据的长度。**loc_size** 值 -1 表示 GBase 8s ESQL/C 在单个操作中发送整个用户定义的简单大对象数据。如果程序将 **loc_size** 设置为 -1，则数据库服务器将读取数据，直到读取函数返回文件结尾（EOF）信号。当计数不等于请求的字节数时，数据库服务器假定为 EOF 信号。

用户定义的简单大对象函数

GBase 8s ESQL/C 提供四个传输函数，它们可以重新定义以处理用户定义的简单大对象位置。

loc_open 、**loc_read** 、**loc_write** 和 **loc_close** 字段包含指向这些用户定义的位置函数的指针。每个函数接收 **ifx_loc_t** 结构的地址作为第一个（或唯一）参数。可以使用 **loc_user_env** 字段保存用户定义的位置函数所需的数据。此外，**loc_xfercount** 以及 **lc_union** 子结构的字段都可用于这些函数。

用户定义的打开函数

要定义如何为传送操作（读或写）准备用户定义的位置，可以创建名为用户定义的打开函数的 C 函数。

在从数据库服务器传送简单大对象数据或传输简单大对象数到数据库服务器之前，GBase 8s ESQL/C 调用定位器结构的 **loc_open** 字段中的打开函数。

此用户定义的打开函数必须接收下列两个参数：

定位器结构的地址 ifx_loc_t *loc_struc，其中 loc_struc 是您的用户定义的打开函数声明的定位器结构的名称

打开模式标志 intoflags，其中 oflags 是包含打开模式标志的变量

如果 GBase 8s ESQL/C 调用打开函数将简单大对象发送给数据库，则该标志包含 LOC_RONLY。如果 GBase 8s ESQL/C 调用函数接受来自数据库的数据，则包含 LOC_WONLY 。

用户定义的打开函数必须为打开操作返回成功代码，如下所示：

0

初始化成功。

-1

初始化失败。该返回码生成 **loc_status** （和 SQLCODE）-452 错误。

下图显示了用户定义的打开函数的骨架功能。

图 11. 用户定义的打开函数示例

```
open_simple_lo(adloc, oflags)
        ifx_loc_t *adloc;
        int oflags;
        {
        adloc->loc_status = 0;
        adloc->loc_xfercount = 0L;
        if (0 == (oflags & adloc->loc_oflags))
        return(-1);
        if (oflags & LOC_RONLY)
        /*** prepare for store to db ***/
        else
        /*** prepare for fetch to program ***/
        return(0);
        }
```

用户定义的读取函数

要定义如何读取用户定义的位置，可以创建名为用户定义的读取函数的 C 函数。

当 GBase 8s ESQL/C 将数据发送给数据库服务器时，它从字符缓冲区读取该数据。要将来自用户定位位置的数据传输到数据库服务器，它从字符缓冲区读取该数据。要将用户定义位置的数据传输到缓冲区，GBase 8s ESQL/C 调用用户定义的读取函数。您的 GBase 8s ESQL/C 程序必须在定位器结构的 **loc_read** 字段中提供用户定义的读取函数的地址。

该用户定义读取函数必须接收以下三个参数：

定位器结构的地址 ifx_loc_t *loc_struc，其中 loc_struc 是您的用户定义的读取函数使用的定位器结构

将数据发送到数据库服务器的缓冲区的地址 char *buffer，其中 buffer 是您程序分配的缓冲区

要从用户定义位置读取的字节数 intnread，其中 nread 是包含字节数的变量

该函数必须将来自用户定义的位置的数据传输给 *buffer* 指示的字符缓冲区。GBase 8s ESQL/C 可能会多次调用该函数从用户定义的位置读取单个简单大对象值。每个调用接收一段数据的地址和长度。跟踪用户定义的读取函数中用户定义位置的当前查找位置。您可能需要使用 **loc_position** 或 **loc_currdata_p** 字段用于此目的。还可以使用 **loc_xfercount** 字段跟踪读取的数据量。

用户定义的读取函数必须返回读取操作的成功代码，如下所示：

>0

读取操作成功。返回值指示实际从定位器结构读取的字节数。

-1

读取操作失败。该返回码生成 -454 的 **loc_status** （和 SQLCODE）错误。

下图显示了用户定义的读取函数的骨架功能。

图 12. 用户定义的读取函数示例

```
read_simple_lo(adloc, bufp, ntoread)
        ifx_loc_t *adloc;
        char *bufp;
        int ntoread;
        {
        int ntoxfer;

        ntoxfer = ntoread;
        if (adloc->loc_size != -1)
```

```
ntoxfer = min(ntoread,
adloc->loc_size - adloc->loc_xfercount);

/*** transfer "ntoread" bytes to *bufp ***/

adloc->loc_xfercount += ntoxfer;
return(ntoxfer);
}
```

用户定义的写入函数

要定义如何写入用户定义的位置，您将创建一个名为用户定义的写入函数的 C 函数。

当 GBase 8s ESQL/C 接收来自数据库服务器的数据时，它将此数据存储在字符缓冲区。要将来自缓冲区的数据传输到用户定义的位置，GBase 8s ESQL/C 调用用户定义的写入函数。您的 GBase 8s ESQL/C 程序必须在定位器结构的 **loc_write** 字段中提供用户定义的写入函数的地址。

该用户定义写入函数必须接收以下三个参数：

定位器结构的地址 ifx_loc_t *loc_struc，其中 loc_struc 是您的用户定义的写入函数使用的定位器结构

接收来自数据库服务器的数据的缓冲区的地址 char *buffer，其中 buffer 是您的程序分配的缓冲区

写入用户定义的位置的字节数 intnwrite，其中 nwrite 是包含字节数的变量

用户定义的写入函数必须传输来自字符缓冲区的数据，*buffer* 指示到用户定义的位置。GBase 8s ESQL/C 可能多次调用函数以将简单大对象值写入到用户定义的位置。每个调用接收一段数据的地址和长度。跟踪用户定义的写入函数中用户定义位置的当前查找位置。您可能需要使用 **loc_position** 或 **loc_currdata_p** 字段用于此目的。还可以使用 **loc_xfercount** 字段跟踪写入的数据量。

用户定义的写入函数必须返回写入操作的成功代码，如下所示：

>0

写入操作成功。返回值指示写入用户定义位置的字节数。

-1

写入操作失败。该返回码生成 -455 的 **loc_status** （和 SQLCODE）错误。

下图显示了用户定义的写入函数的骨架功能。

图 13. 用户定义的写入函数的示例

```
write_simple_lo(adloc, bufp, ntowrite)
        ifx_loc_t *adloc;
        char *bufp;
        int ntowrite;
        {
        int xtoxfer;

        ntoxfer = ntowrite;
        if (adloc->loc_size != -1)
        ntoxfer = min(ntowrite,
        (adloc->loc_size) - (adloc->loc_xfercount));

        /*** transfer "ntowrite" bytes from *bufp ***/

        adloc->loc_xfercount += ntoxfer;
        return(ntoxfer);
        }
```

用户定义的关闭函数

要定义如何执行清除用户定义的位置，您将创建一个名为用户定义的关闭函数的 C 函数。

当传输到数据库服务器或从数据库服务器传输完成时，GBase 8s ESQL/C 调用由定位器结构的 **loc_close** 字段提供的关闭函数，清除任务包括关闭文件或释放用户定义的位置使用的内存。

该函数必须接收一个参数：定位器结构的地址 **ifx_loc_t** **loc_struc* ，其中 *loc_struc* 是您的用户定义的关闭函数使用的定位器结构。用户定义的关闭函数为此关闭操作返回成功码如下所示：

0

清除操作成功。

-1

清除操作失败。该返回码生成 -455 的 **loc_status** （和 SQLCODE）错误。

下图显示了用户定义的关闭函数的骨架功能。

图 14. 用户定义的关闭函数示例

```
close_simple_lo (adloc)
        ifx_loc_t       *adloc;
        {
        adloc->loc_status = 0;
        if (adloc->loc_oflags & LOC_WONLY) /* if fetching */
        {
        adloc->loc_indicator = 0; /* clear indicator */
        adloc->loc_size = adloc->loc_xfercount;
        }
        return(0);
        }
```

2.7.6 dispcat_pic 程序

dispcat_pic 程序使用 GBase 8s ESQL/Cifx_loc_t 定位器结构接收两个简单大对象列。程序从 stores7 演示数据库的检索 catalog 表检索 cat_descr TEXT 简单大对象列和 cat_picture BYTE 列。

dispcat_pic 程序允许您从命令行中选择一个数据库，以防您以不同的名称创建 **stores7**数据库。如果没有给出数据库名称，则 **dispcat_pic** 打开 **stores7** 数据库。例如：以下命令运行 **dispcat_pic** 可执行文件并指定 **mystores** 数据库：

```
dispcat_pic mystores
```

该程序提示用户输入 **catalog_num** 值，并执行一个 SELECT 语句读取 **stock** 表的 **description** 列，并从 **catalog** 表中读取 **catalog_num** 、**cat_descr** 和 **cat_picture** 列。如

果数据库服务器发现目录号和 **cat_picture** 列不为空，它将 **cat_picture** 列写入 .gif 文件。

如果 SELECT 语句成功，程序显示 **catalog_num** 、**cat_descr** 和 **description** 列。由于这些列存储文件，因此可以显示在任何 GBase 8s ESQL/C 平台上。该程序还允许用户输入另一个 **catalog_num** 值或终止程序。

准备运行 dispcat_pic 程序

准备行 dispcat_pic 程序：

使用 blobload 实用程序将简单大对象图像加载到 **catalog** 表。

将 dispcat_pic.ec 文件编译到可执行程序中。

加载简单大对象图像

当 **catalog** 表作为 **stores7** 演示数据库的一部分创建时，所有行的 **cat_picture** 列都将设置为空。GBase 8s ESQL/C 演示目录提供五个图形图像。使用 blobload 应用程序将简单大对象图像加载到 **catalog** 表的 **cat_picture** 列。

要显示这些来自 **dispcat_pic** 程序的简单大对象图像，必须将图像加载到 **catalog** 表。

选择图像文件

在图形交换格式文件中提供五个 cat_picture 图像，具有 .gif 文件扩展名。

GBase 8s ESQL/C 提供 .gif 文件中的图像，以标准格式提供，可以在所有平台上显示，也可以将其与其它供应商提供的过滤程序一起翻译成其它格式。下表的右列显示了简单大对象图像的 .gif 文件的名称。

表 6. 简单大对象示例的图像文件

图像	图形交换格式（.gif 文件）
棒球手套	cn_10001.gif
自行车曲柄	cn_10027.gif
自行车头盔	cn_10031.gif
高尔夫球	cn_10046.gif
跑步鞋	cn_10049.gif

图像文件名的数字部分是要更新图像的 **catalog** 表的行的 **catalog_num** 值。例如，cn_10027.gif 应该更新到行的 **cat_picture** 列，其中 10027 是 **catalog_num** 的值。

使用 blobload 实用程序加载简单大对象图像

blobload 实用程序是作为 GBase 8s ESQL/C 演示文件一部分提供的 GBase 8s ESQL/C 程序。它使用命令行语法将字节映像加载到数据库的指定表和列。

要使用 blobload 加载简单大对象图像：

使用以下命令编译 blobload.ec 程序：

esql -o blobload blobload.ec

在 UNIX(TM) 命令行上输入不带任何参数的 blobload。

下图显示了此命令的输出，该命令描述了 blobload 期望的命令行参数。

图 15. blobload 实用程序的输出示例

Sorry, you left out a required parameter.

Usage: blobload {-i | -u} -- choose insert or update
-f filename -- file containing the blob data
-d database_name -- database to open
-t table_name -- table to modify
-b blob_column -- name of target column
-k key_column key_value -- name of key column and a value
-v -- verbose documentary output

All parameters except -v are required.

Parameters may be given in any order.

As many as 8 -k parameter pairs may be specified.

运行 blobload 程序将每个图像加载到它适合的 cat_picture 列。

blobload 的 **-u** 选项更新带有简单对象图像的指定列。要标识更新哪列，您还必须使

用 blobload 的 **-f** 、**-d** 、**-t** 、**-b** 和 **-k** 选项。

您必须为要更新的每个图像文件运行一次 blobload 程序。例如，以下命令将 **cn_10027.gif** 文件的内容加载到 **catalog_num**10027 的行的 **cat_picture** 列中。**catalog_num** 列是 **catalog** 表中的关键列。

```
blobload -u -f cn_10027.gif -d stores7 -t catalog -b cat_picture -k
        catalog_num 10027
```

使用相同的命令更新剩下的四个图像文件中的每一个。替换要加载的图像文件的文件名（**-f** 选项）和相应的 **catalog_num** 值（**-k** 选项）。

dispcat_pic.ec 文件指南

```
=============================================================================
======
1. /*
2.       * dispcat_pic.ec *
3.The following program prompts the user for a catalog number,
4. selects the cat_picture column, if it is not null, from the
5. catalog table of the demonstration database and saves the
6. image into a .gif file.
7. */
8. #include <stdio.h>
9. #include <ctype.h>
10. EXEC SQL include sqltypes;
11. EXEC SQL include locator;
12. #define WARNNOTIFY          1
13.#define NOWARNNOTIFY       0
14. #define LCASE(c) (isupper(c) ? tolower(c) : (c))
15. #define BUFFSZ 256
16. extern errno;
17. EXEC SQL BEGIN DECLARE SECTION;
18.         mlong cat_num;
19.         ifx_loc_t cat_descr;
20.         ifx_loc_t cat_picture;
```

21. EXEC SQL END DECLARE SECTION;

22. char cpfl[18]; /* file to which the .gif will be copied */

==========

第 8 - 11 行

#include <stdio.h> 语句包含 stdio.h 头文件（来自 UNIX(TM) 的 /usr/include 目录和 Windows(TM) 上的 Microsoft(TM) Visual C++ 的 **include** 子目录）。stdio.h 文件使 **dispcat_pic** 能够使用标准 C I/O 库。程序还包含 GBase 8s ESQL/C 头文件 sqltypes.h 和 locator.h（第 10 行和 11 行）。locator.h 文件包含定位器结构的定义以及您需要使用此结构的常量。

第 12 - 16 行

与 exp_chk2() 异常处理函数一起使用 WARNNOTIFY 和 NOWARNNOTIFY 常量（第 12 和 13 行）。调用 exp_chk2() 指定其中一个常量作为第二个参数，来指示是否显示 SQLSTATE 和 SQLCODE 的警告信息（WARNNOTIFY 或 NOWARNNOTIFY）。有关 exp_chk2() 函数的更多信息，请参阅 171 - 177 行。

程序使用 BUFFSZ（第 15 行）指定存储来自用户输入的数组的大小，第 16 行定义 **errno**，一个外部整数，用于系统调用存储错误号。

第 17 - 21 行

这些行定义程序需要的全局主机变量。**cat_num** 标量保存 **catalog** 表的 **catalog_num** 列值。第 19 和 20 行指定定位器结构作为主机变量的数据类型，用来接收 **catalog** 表的 **cat_descr** 和 **cat_picture** 简单大对象列。定位器结构是从数据库检索或存储到数据库的简单大对象列的主机变量。定位器结构具有 **ifx_loc_t** typedef。程序使用定位器结构指定简单大对象大小和位置。

第 22 行

第 22 行定义一个单独全局 C 变量。**cpfl** 字符组存储文件的名称。已命名的文件是数据库服务器写入 **cat_picture** 的简单大对象 .gif 图像的位置。

```
==========================================================================
======
23. main(argc, argv)
24. mint argc;
25. char *argv[];
26. {
27.          char ans[BUFFSZ];
28.          int4 ret, exp_chk2();
29.          char db_msg[ BUFFSZ + 1 ];
30.       EXEC SQL BEGIN DECLARE SECTION;
31.             char db_name[20];
32.             char description[16];
33.       EXEC SQL END DECLARE SECTION;
==========================================================================
======
```

第 23 - 26 行

main() 函数是程序执行开始的点。第一个参数 **argc** 是一个整数，它给出了在命令行中提交的参数数。第二个参数 **argv[]** 是一个指向包含命令行参数的字符数组的指针。**dispcat_pic** 程序期望只有 **argv[1]** 参数（可选）指定要访问的数据库的名称。如果 **argv[1]** 不存在，程序将打开 **stores7** 数据库。

第 27 - 29 行

第 27 - 29 行定义了 main() 函数的范围内的局部变量。**ans[BUFFSZ]** 数组是接收来自用户输入的缓冲区，即相关联的 **cat_picture** 列的目录号。第 28 行为 exp_chk2() 返回的值定义 4 字节整数（**ret**）并声明 exp_chk2() 为返回 **long** 的函数。**db_msg[BUFFSZ +**

1] 字符数组保存 CONNECT 语句的格式以打开数据库。如果当 CONNECT 执行时发生错误。则 **db_msg** 中的字符串传递到 exp_chk2() 函数来标识错误的原因。

第 30 - 33 行

第 30 - 33 行定义了 main() 函数本地的 GBase 8s ESQL/C 主机变量。主机变量接收从表中提取的数据，并提供写入表的数据。如果在命令行用户指定了一个，则 **db_name[20]** 字符数组是存储数据库名的主机变量 **description** 变量保存用户输入的值，它存储在 **stock** 表的列中。

```
==========================================================================
======
34.             printf("DISPCAT_PIC Sample ESQL Program running.\n\n");
35.         if (argc > 2)                   /* correct no. of args? */
36.             {
37.             printf("\nUsage: %s [database]\nIncorrect no. of
        argument(s)\n",
38.                     argv[0]);
39.             printf("DISPCAT_PIC Sample Program over.\n\n");
40.             exit(1);
41.             }
42.         strcpy(db_name, "stores7");
43.         if(argc == 2)
44.             strcpy(db_name, argv[1]);
45.         EXEC SQL connect to :db_name;
46.         sprintf(db_msg,"CONNECT TO %s",db_name);
47.         if(exp_chk2(db_msg, NOWARNNOTIFY) < 0)
48.             {
49.             printf("DISPCAT_PIC Sample Program over.\n\n");
50.             exit(1);
51.             }
52.         if(sqlca.sqlwarn.sqlwarn3 != 'W')
```

```
53.             {
54.             printf("\nThis program does not work with GBase 8s SE. ");
55.             EXEC SQL disconnect current;
56.             printf("\nDISPCAT_PIC Sample Program over.\n\n");
57.             exit(1);
58.             }
59.       printf("Connected to %s\n", db_name);
60.      ++argv;
```

==========

第 34 - 51 行

这些行解释命令行参数并打开数据库。第 35 行检查在命令行输入的参数是否多于两个。如果多于，**dispcat_pic** 显示它期望的参数后结束。第 42 行将 **stores7** 的缺省数据库名分配给 **db_name** 主机变量。如果用户没有输入命令行参数，则该程序打开此数据库。

该程序然后测试命令行参数的数量是否等于 2。如果等于，则 **dispcat_pic** 假定第二个参数 **argv[1]** 是用户希望打开的数据库名称。第 44 行使用 **strcpy()** 函数将 **argv[1]** 命令行中数据库名称复制到 **db_name** 主机变量。然后程序 CONNECT 语句（第 45 行）建立到缺省数据库服务器的连接，打开指定的数据库（在 **db_name** 中）。

该程序在 **db_msg[]** 数组（第 46 行）中再现 CONNECT 语句。这样做是为了在第 47 行上的 exp_chk2() 调用。它将参数作为语句的名称。第 47 行调用 exp_chk2() 函数俩检查结果。此调用 exp_chk2() 指定 NOWARNNOTIFY 参数以防止显示 CONNECT 生成的警告。

第 52 - 60 行

CONNECT 成功打开数据库后，它将有关服务器的信息存储在 **sqlca.sqlwarn** 数组中。因为 **dispcat_pic** 程序处理旧版本服务器不支持的简单大对象数据类型，所以 52 行检查数据库服务器的类型。如果 **sqlca.sqlwarn** 的 **sqlwarn3** 元素设置为 W，则数据库服务器将继续执行。否则，程序通知用户它无法继续并退出。程序已经建立了数据库服务器的有效性，现在显示打开的数据库的名称（第 59 行）。

```
    ================================================================
======
61.  while(1)
62.          {
63.          printf("\nEnter catalog number: "); /* prompt for cat.
                                                        * number */
64.          if(!getans(ans, 6))
65.                  continue;
66.          printf("\n");
67.          if(rstol(ans, &cat_num))        /* cat_num string to long */
68.                  {
69.                  printf("** Cannot convert catalog number '%s' to long
       integer\n",
                        ans);
70.                  EXEC SQL disconnect current;
71.                  printf("\nDISPCAT_PIC Sample Program over.\n\n");
72.                  exit(1);
73.                  }
74.          ret=sprintf(cpfl, "pic_%s.gif", ans);
75.          /*
76.            *    Prepare locator structure for select of cat_descr
77.            */
78.          cat_descr.loc_loctype = LOCMEMORY;    /* set for 'in memory' */
79.         cat_descr.loc_bufsize = -1;                /* let db get buffer */
80.          cat_descr.loc_mflags = 0;      /* clear memory-deallocation
                                              * feature */
81.          cat_descr.loc_oflags = 0;                  /* clear loc_oflags */
82.          /*
83.            *    Prepare locator structure for select of cat_picture
84.            */
85.          cat_picture.loc_loctype = LOCFNAME;      /* type = named file */
```

```
86.         cat_picture.loc_fname = cpfl;            /* supply file name */
87.         cat_picture.loc_oflags = LOC_WONLY; /* file-open mode = write
                                                    */
88.         cat_picture.loc_size = -1;   /* size = size of file */
```

==========

第 61 - 74 行

第 61 行上的 **while(1)** 开始 **dispcat_pic** 中的主处理循环。第 63 行提示用户输入用户希望看到的 **cat_picture** 列的目录号。第 64 行调用 getans() 接收用户输入的目录号。getans() 的参数是存储输入的地址 **ans[]**，和预期输入的最大长度，包括空终止符。如果输入不可接受，getans() 返回 0，第 65 行控制返回到第 61 行循环顶部的 **while** ，这样可以再次显示目录号的提示。第 67 行调用 GBase 8s ESQL/C 库函数 rstol() 将字符输入字符串转换为 **long** 数据类型以匹配 **catalog_num** 列的数据类型。如果 rstol() 返回非零值，则转换失败并且 69 – 72 行对用户显示消息，关闭连接并退出。第 74 行创建程序写入简单大对象图像的 .gif 文件名称。文件名由常量 **pic_**、用户输入的目录号和扩展 .gif 组成。该文件是在程序运行的目录中创建的。

第 75 - 81 行

这些行定义 **catalog** 表的 **cat_descr** 列的简单大对象位置。如下所示：

第 78 行将 **cat_descr** 定位器结构中的 **loc_loctype** 设置为 LOCMEMORY 以告知 GBase 8s ESQL/C 将 **cat_descr** 的数据选择为内存。

第 79 行将 **loc_bufsize** 设置为 -1，以便 GBase 8s ESQL/C 分配一个内存缓冲区来接收 **cat_descr** 的数据。

第 80 行将 **loc_mflags** 设置为 0 来禁用 GBase 8s ESQL/C 的内存释放功能（请参阅 149 行）。

如果选择成功，则 GBase 8s ESQL/C 返回 **loc_buffer** 中分配的缓冲区的地址。第 81 行将 **loc_oflags** 文件打开模式标志设置为 0，因为该程序将简单大对象信息检索到内存中而不是文件中。

第 82 - 88 行

这些行准备定位器结构检索 **catalog** 表的 **cat_picture** BYTE 列。第 85 行将 LOCFNAME 移动到 **loc_loctype** 以告知 GBase 8s ESQL/C 在已命名文件中定位 **cat_descr** 数据。第 86 行将 **cpfl** 文件名的地址移动到 **loc_fname**。第 87 行将 LOC_WONLY 值移动到 **loc_oflags** 文件打开模式标志来告知 GBase 8s ESQL/C 以只写的模式打开文件。最终，第 88 行将 **loc_size** 设置为 -1 来告知 GBase 8s ESQL/C 在单个传输中发送 BYTE 数据而不是将值分解成较小的部分并使用多个传输。

```
================================================================
======
89.     /* Look up catalog number */
90.     EXEC SQL select description, catalog_num, cat_descr, cat_picture
91.         into :description, :cat_num, :cat_descr, :cat_picture
92.         from stock, catalog
93.         where catalog_num = :cat_num and
94.         catalog.stock_num = stock.stock_num and
95.         catalog.manu_code = stock.manu_code;
96.     if((ret = exp_chk2("SELECT", WARNNOTIFY)) == 100) /* if not
                                                          * found */
97.         {
98.         printf("** Catalog number %ld not found in ", cat_num);
99.         printf("catalog table.\n");
100.          printf("\t OR item not found in stock table.\n");
101.          if(!more_to_do())
102.              break;
103.          continue;
104.          }
105.      if (ret < 0)
106.          {
107.          EXEC SQL disconnect current;
108.          printf("\nDISPCAT_PIC Sample Program over.\n\n");
109.          exit(1);
```

```
110.            }
111.        if(cat_picture.loc_indicator == -1)
112.            printf("\tNo picture available for catalog number %ld\n\n",
113.                    cat_num);
114.        else
115.            {
116.            printf("Stock Item for %ld: %s\n", cat_num, description);
117.            printf("\nThe cat_picture column has been written to the
        file:
118.                %s\n",      cpfl);
119.            printf("Use an image display tool or a Web browser ");
120.            printf("to open %s for viewing.\n\n", cpfl);
121           }
122.        prdesc();           /* display catalog.cat_descr */
```

==========

第 89 - 95 行

这些行定义了一个 SELECT 语句来检索用户输入的目录号的 **catalog** 表的 **catalog_num** 、**cat_descr** 和 **cat_picture** 列和 **stock**表的 **description** 列。SELECT 语句的 INTO 子句标识包含选择值的主机变量。这两个 **ifx_loc_t** 主机变量 **cat_descr** 和 **cat_picture**，在子句中列出了 TEXT 和 BYTE 值。

第 96 - 104 行

exp_chk2() 函数检查 SELECT 语句是否能够在 **catalog** 表和**stock** 表中找到所选行的 **stock_num** 和 **manu_code**。**catalog** 表不会包含在 **stock** 表中没有对应行的行。行 98 - 103 处理 NOT FOUND 条件。如果 exp_chk2() 函数返回 100，则未找到行；行 98 - 100 显示产生的消息。more_to_do() 函数（第 101 行）询问用户是否希望继续。如果用户回答否（n），则 **break** 终止主处理循环，并控制传输到 131 行以在程序终止之前关闭数据库。

第 105 - 110 行

如果在选择期间发生了运行错误，则程序关闭当前连接，通知用户并以 1 的状态退出。

第 111 - 113 行

如果 **cat_picture.loc_indicator** 包含 -1（第 111 行），则 **cat_picture** 列包含空，并且程序会通知用户（112 行）。然后继续到 113 行执行，以显示其它返回的列值。

第 114 - 122 行

这些行显示 SELECT 语句返回的其它列。第 116 行显示正在处理的目录号以及 stock 表的 description 列。第 122 行调用 prdesc() 显示 cat_descr 列。

```
==========================================================================
123.    if(!more_to_do())     /* More to do? */
124.        break;          /* no, terminate loop */
125.         /* If user chooses to display more catalog rows, enable the
126.            * memory-deallocation feature so that ESQL/C deallocates old
127.            * cat_desc buffer before it allocates a new one.
128.            */
129.        cat_descr.loc_mflags = 0;     /* clear memory-deallocation feature
                                          */
130.        }
131.        EXEC SQL disconnect current;
132.        printf("\nDISPCAT_PIC Sample Program over.\n\n");
133.        } /* end main */
134.        /* prdesc() prints cat_desc for a row in the catalog table */
135.        #include "prdesc.c"
==========================================================================
```

======

第 123 - 130 行

more_to_do() 函数然后询问用户是否要输入多个目录号。如果不，more_to_do() 返回 0 并且程序执行 **break** 来终止主处理循环，关闭数据库并终止程序。

第 130 行的结束括号终止了主处理循环，它从第 61 行的 **while(1)** 开始。如果用户希望输入另一个目录号，则控制返回到 61 行。

第 131 - 133 行

当 **break** 语句（124 行）终止 **while(1)** 在 61 行开始的主处理循环时，控制传输到 131 行，关闭数据库以及到缺省数据库服务器的连接。133 行的结束括号终止 23 行的 main() 函数并终止此程序。

第 134 和 135 行

一些 GBase 8s ESQL/C 简单大对象演示程序调用 prdesc() 函数。要避免在每个程序中具有此函数，该函数放在它自己的源文件汇总。调用 prdesc() 的每个函数都包含此 prdesc.c 源文件。自从 prdesc() 不会包含任何 GBase 8s ESQL/C 语句,该程序可以使用 C **#include** 预处理器语句包含它(而不是 GBase 8s ESQL/C**include** 伪指令）。

==========

```
136. /*
137.  * The inpfuncs.c file contains the following functions used in this
138.  * program:
139.  *      more_to_do() - asks the user to enter 'y' or 'n' to indicate
140.  *                     whether to run the main program loop again.
141.  *
142.  *      getans(ans, len) - accepts user input, up to 'len' number of
```

```
143.  *                    characters and puts it in 'ans'
144.  */
145. #include "inpfuncs.c"
146. /*
147.  * The exp_chk.ec file contains the exception handling functions to
148.  * check the SQLSTATE status variable to see if an error has
        occurred
149.  * following an SQL statement. If a warning or an error has
150.  * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
151.  * displays the detail for each exception that is returned.
152.  */
153. EXEC SQL include exp_chk.ec;
```

==========

第 136 和 145 行

一些 GBase 8s ESQL/C 演示程序还会调用 more_to_do() 和 getans() 函数。这些函数也被分解成一个单独 C 源文件，并包含在适当的演示程序中。这些函数都不包含 GBase 8s ESQL/C，因此程序可以使用 C **#include** 预处理器语句包含此文件。

第 146 - 153 行

exp_chk2() 函数检查 SQLSTATE 状态变量以确定 SQL 语句的输出。因为许多演示程序是异常检查 exp_chk2() 函数，以及它支持的函数都被分解到单独的 exp_chk.ec 源文件中。**dispcat_pic** 程序必须使用 GBase 8s ESQL/C**include** 伪指令包含此文件因为异常数量函数使用 GBase 8s ESQL/C 语句。

提示： 在生产环境中，诸如 prdesc() 、more_to_do() 、getans() 和 exp_chk2() 函数会被编入 C 库，并在编译过程包含在 GBase 8s ESQL/C 程序的命令行中。

prdesc.c 文件指南

prdesc.c 文件包含 prdesc() 函数。该函数将指针 **p** 设置为定位器结构的 **loc_buffer** 字段中提供的访问简单大对象的地址。该函数随后从缓冲区中读取 80 字节的文本，直到 **loc_size** 中指定的大小。此函数用于几个简单大对象演示程序，因此它位于单独的文件中，并包含在适当的源文件中。

```
==========================================================================
======
1. /* prdesc() prints cat_desc for a row in the catalog table */
2. prdesc()
3. {
4.          int4 size;
5.          char shdesc[81], *p;
6.          size = cat_descr.loc_size;       /* get size of data */
7.          printf("Description for %ld:\n", cat_num);
8.          p = cat_descr.loc_buffer;       /* set p to buffer addr */
9.       /* print buffer 80 characters at a time */
10.          while(size >= 80)
11.              {
12.              ldchar(p, 80, shdesc);          /* mv from buffer to shdesc */
13.              printf("\n%80s", shdesc);     /* display it    */
14.              size -= 80;                        /* decrement length */
15.              p += 80;                           /* bump p by 80 */
16.              }
17.           strncpy(shdesc, p, size);
18.          shdesc[size] = '\0';
19.          printf("%-s\n", shdesc);          /* display last segment */
20. }
```

```
==========================================================================
==
```

第 1 - 20 行

第 2 - 20 行构成了 main() 函数，它显示 **catalog** 表的 **cat_descr** 列。第 4 行定义 **size**，main() 用 **cat_descr.loc_size** 中的值初始化一个长整数。第 5 行定义 **shdesc[81]**，一个数组，main() 临时移动 **cat_descr** 文本的 80 字节块输出。第 5 行还定义了 ***p**，一个指针，用于标记缓冲区当前位置。

在 **loc_size** 中，数据库服务器返回其为简单大对象分配的缓冲区的大小。第 6 行将 **cat_descr.loc_size** 移动到 **size**。第 7 行显示字符串 "Description for:" 作为 **cat_descr** 文本的头文件。第 8 行将 **p** 指针设置为数据库服务器在 **cat_descr.loc_size** 中的返回的缓冲区地址。

第 10 行开始循环，向用户显示 **cat_descr** 文本。while() 重复循环，直到 **size** 小于 80 行。第 11 行开始循环的内容。GBase 8s ESQL/Cldchar() 库函数从缓冲区中的当前位置 **p** 地址复制 80 个字节到 **shdesc[]** ，并移除任何尾部空白。第 13 行打印 **shdesc[]** 的内容。第 14 行从 **size** 中减去 80 以考虑打印的缓冲区的部分。第 15 行，循环中的最后一行，将 80 添加到 **p** 以将其移动到显示的缓冲区的部分。

一次显示 **cat_descr.loc_size** 80 个字节的过程将持续到少于 80 个字符被显示（**size** < 80）。第 17 行将剩余的缓冲区复制到 **shdesc[]** 中的长度。第 18 行将一个空值添加到 **shdesc[size]** 以标记数组的末尾，第 19 行显示 **shdesc[]**。

inpfuncs.c 文件指南

inpfuncs.c 文件包含 getans() 和 more_to_do() 函数。

因为这些函数能在几个 GBase 8s ESQL/C 演示程序中使用，因此它们位于单独的文件中，并包含在适当的演示源文件中。

```
==============================================================================
```

```
1. /* The inpfuncs.c file contains functions useful in character-based
2.       input for a C program.
3.   */
4. #include <ctype.h>
5. #ifndef LCASE
6. #define LCASE(c) (isupper(c) ? tolower(c) : (c))
7. #endif
8. /*
9.         Accepts user input, up to 'len' number of characters and returns
10         it in 'ans'
11.  */
```

```
12. #define BUFSIZE 512
13. getans(ans, len)
14. char *ans;
15. mint len;
16. {
17.         char buf[BUFSIZE + 1];
18.         mint c, n = 0;
19.         while((c = getchar()) != ';' && n < BUFSIZE)
20.             buf[n++] = c;
21.       buf[n] = '\0';
22.       if(n > 1 && n >= len)
23.             {
24.             printf("Input exceeds maximum length");
25.             return 0;
26.             }
27.         if(len <= 1)
28.             *ans = buf[0];
29.         else
30.             strnpy(ans, buf, len);
31.         return 1;
32. }
```

==========

第 1 - 7 行

第 4 行包括 UNIX(TM)ctype.h 头文件。该头文件提供 LCASE() 宏定义中使用的 islower() 和 tolower() 宏的定义（在第 6 行中定义）。如果尚未在程序中定义则程序仅定义 LCASE 宏。

第 8 - 32 行

BUFSIZE 常量（第 12 行）定义在 getans() 函数中使用的字符缓冲区的大小。第 13 - 32 行继续 getans() 函数。getans() 函数使用 getchar() 标准库函数接收来自用户的输出。第 14 和 15 行定义 getans() 的参数，其中复制输入的缓冲区（**ans**）的地址以及调用函数期望的最大字符数（**len**）。第 17 行定义 **buf[]**，一个输入缓冲区数组。**int** 变量 **c**（第 18 行）接收 getchar() 返回的字符。第二个整数在第 18 行定义（**n**），它用于下标 **buf[]** 输入缓冲区。

第 19 行调用 getchar() 接收来自用户的输入，直到遇到 \n 换行符或者接收到最大输出值；即，**n** 不小于 BUFFSZ。第 20 行将输入字符 **c** 移动到 **buf[]** 中的当前行。第 21 行将空终止符放置到输出的末尾 **buf[n]**。

第 22 - 26 行检查接收到的字符数 **n** 是否小于预期的字符数 **len**。如果没有，第 24 行向用户显示消息，并且行 25 向调用函数返回 0 以指示错误。第 27 行检查是否输入了一个或多个字符。如果预期的 **len** 小于或等于 1，则第 28 行只将一个字符移动到 **ans** 调用函数给出的地址。如果预期只有一个字符，则 getans() 不在输入端附加一个空终止符。如果输入的长度大于 1，则第 30 行将用户的输入复制到调用函数（**ans**）。第 31 行返回 1 以调用函数指示成功完成。

```
==========================================================================
33. /*
34.  *    Ask user if there is more to do
35.  */
36. more_to_do()
37. {
38.       char ans;
39.            do
40.                 {
41.                 printf("\n**** More? (y/n) …");
42.                 getans(&ans, 1);
43.                 } while((ans = LCASE(ans)) != 'y' && ans != 'n');
44.            return (ans == 'n') ? 0 : 1;
45. }
```

==========
==

第 33 - 45 行

more_to_do() 函数显示 "More? (y/n)..." 来询问用户是否希望继续执行程序。more_to_do() 函数没有任何输入参数。行 38 定义了一个字符字段 **ans** 来接收来自用户的应答。在第 43 行表示的条件导致问题被再次显示，直到用户回答 y（yes）或 n（no）。LCASE 宏将用户的回答转换为小写字母进行比较。第 42 行调用 getans() 接受来自用户的输入。在用户回答是或否后，控制转到第 44 行，其返回 1，是 0 则返回给调用函数。

2.8 智能大对象

智能大对象是存储大型非关系数据对象（例如图像、声音片段、文档、图形、地图和其它大型对象）的数据类型，并允许您对这些对象执行读取、写入和查找操作。

智能大对象由 CLOB （字符大对象）和 BLOB（二进制大对象）数据类型组成。CLOB 数据类型存储大量的文本数据对象。BLOB 数据类型存储未分化字节流中的二进制数据的大对象。智能大对象存储在名为 sbspace 的特定类型的数据库空间中。

本节的结尾介绍了一个名为 **create_clob** 的示例程序。**create_clob** 示例程序演示如何从 GBase 8s ESQL/C 程序创建新的智能大对象，插入数据到 **stores7** 的 CLOB 列，然后从此列选择智能大对象数据。

仅当您使用 GBase 8s 作为您的数据库服务器时，这些主题中的信息才适用。

这些主题描述了有关使用智能大对象进行编程的以下信息：

智能大对象的数据结构

创建智能大对象

访问智能大对象

获取智能大对象的状态

变更智能大对象列

智能大对象的 GBase 8s ESQL/C API

2.8.1 智能大对象数据结构

GBase 8s ESQL/C 支持具有 **ifx_lo_t** 数据类型的 SQL 数据类型 CLOB 和 BLOB 。由于智能大对象数据的潜在的巨大规模，GBase 8s ESQL/C 程序不会直接将数据存储在主机变量中。相反，客户端应用程序以类似文件的结构访问数据。要在 GBase 8s ESQL/C 程序中使用智能大对象，请采取以下操作：

使用 ifx_lo_t 数据类型声明主机变量。

使用以下三个数据结构的组合访问智能大对象：

LO 特定的结构 ifx_lo_create_spec_t

LO 指针结构 ifx_lo_t

整数 LO 文件描述符

重要：ESQL/C 用于访问智能大对象的数据结构以 LO 前缀开头。这个前缀是大对象的缩写。目前，数据库服务器使用**大对象**来引用智能大对象和简单大对象。然而，为了遗留目的，保留在访问智能大对象的 ESQL/C 结构中使用此前缀。

声明主机变量

声明用于具有 **ifx_lo_t** 结构（名为 **ifx_lo_t** 数据类型）的**固定二进制**主变量的类型为 CLOB 或 BLOB 的数据库列的 GBase 8s ESQL/C 主机变量，如下所示：

```
EXEC SQL include locator;
        ⋮
        EXEC SQL BEGIN DECLARE SECTION;
        fixed binary 'clob' ifx_lo_t clob_loptr;
        fixed binary 'blob' ifx_lo_t blob_loptr;
        EXEC SQL END DECLARE SECTION;
        ⋮
        EXEC SQL select blobcol into :blob_loptr from tab1;
```

要访问智能大对象，您必须在您的 GBase 8s ESQL/C 程序中包含 locator.h 头文件。该头文件包含数据结构的定义和程序需要使用智能大对象的常量。

LO 特定结构

在创建新的智能大对象之前，必须使用 ifx_lo_def_create_spec() 函数分配 LO 特定结构。

ifx_lo_def_create_spec() 函数执行以下任务：

它分配新的 LO 特定结构，它的指针作为参数提供。

它初始化 LO 特定结构的所有字段：磁盘存储选项和创建时间标志到适当的空值。

ifx_lo_create_spec_t 结构

LO 特定结构 ifx_lo_create_spec_t，存储 GBase 8s ESQL/C 程序中智能大对象的存储特征。

locator.h 头文件定义 LO 特定结构，因此您必须在访问此结构的 GBase 8s ESQL/C

程序中包含 locator.h 文件。

重要： LO 特定结构 **ifx_lo_create_spec_t** 对 GBase 8s ESQL/C 程序来说是不透明数据类型。不要直接访问它的内容结构。**ifx_lo_create_spec_t** 的内部结构可能在之后的编译过程中发生改变。因此，要创建便捷式代码，请始终使用 GBase 8s ESQL/C 访问函数来获取和存储 LO 特定结构中的值。

LO 特定结构存储智能大对象的以下结构特征：

磁盘存储信息

创建时间标志

磁盘存储信息

LO 特定结构存储磁盘存储信息，它有助于数据库服务器确定如何在磁盘上最有效地存储智能大对象。

下表显示了相应的 GBase 8s ESQL/C 访问函数的磁盘存储信息。

表 1. LO 特定结构中的磁盘存储信息

磁盘存储信息	描述	ESQL/C 访问程序函数
估计字节数	估计智能大对象的最终大小（以字节为单位）。数据库服务器使用此值来确定要存储智能大对象的区域。此值提供优化信息，如果值严重不正确，则不会导致错误的行为。但是，这意味着数据库服务器可能不一定为智能大对象选择最佳扩展大小。	ifx_lo_specget_estbytes() ifx_lo_specset_estbytes()
最大字节数	智能大对象的最大字节数（以字节为单位）。数据库服务器不允许智能大对象超过此大小。	ifx_lo_specget_maxbytes() ifx_lo_specset_maxbytes()
分配 extent 大小	分配的 extent 大小以千字节为单位。最佳情况下，分配范围是容纳智能大对象的所有数据的 chunk 中的单个 extent 。 数据库服务器以分配的 extent 大小为增量为智能大对象执行存储分配。它尝	ifx_lo_sepcget_extsz(), ifx_lo_specset_extsz()

磁盘存储信息	描述	ESQL/C 访问程序函数
	试将分配范围为一个 chunk 中的单个 extent。但是，如果单个 extent 不足够大，则数据库服务器必须根据需要使用多个 extent 才能满足该请求。	
sbspace 的名称	包含智能大对象的 sbspace 名称。sbspace 名称最多可以为 18 字节长。该名称必须以空终止。	ifx_lo_specget_sbspace() ifx_lo_specset_sbspace()

对于大多数应用程序，建议您使用数据库服务器确定的磁盘存储信息。

创建时间标志

LO 特定结构存储创建时间标志，它告诉数据库服务器选择哪项分配给智能大对象。

下表显示了创建时间标志以及相应的 GBase 8s ESQL/C 存取函数。

表 2. LO 特定结构中的创建时间标志

指示符类型	创建时间标志	描述
日志记录	LO_LOG	告知数据库服务器将更改记录到系统日志文件中的智能大对象。 仔细考虑是否使用 LO_LOG 标志值。数据库服务器使用相当大的开销来记录智能大对象。还必须确保系统日志文件足够大以容纳智能大对象的值。
	LO_NOLOG	告知数据库服务器关闭涉及智能大对象的所有操作的日志记录。
最后一次访问时间	LO_KEEP_LASTACCESS_TIME	告知数据库服务器保存智能大对象的最后访问时间。此访问时间是上次读取或写入操作的时间。 是否考虑使用 LO_KEEP_LASTACCESS_TIME 标志值。数据库服务器使用相当大的开销来维护智能大对象的最后访问时间。

指示符类型	创建时间标志	描述
	LO_NOKEEP_LASTACCESS_TIME	告知数据库服务器不要维护智能大对象的最后访问时间。

locator.h 头文件定义 LO_LOG 、LO_NOLOG 、LO_KEEP_LASTACCESS_TIME 和 LO_NOKEEP_LASTACCESS_TIME 创建时间常量。两组创建时间标志，记录指示符和最后的访问时间指示符作为单个标志值存储在 LO 特定结构中。要设置每个组的标志，请使用 C 语言 OR 运算符一起屏蔽两个标志值。但是屏蔽互斥标志会导致错误。

ifx_lo_specset_flags() 函数将创建时间标志为新值。ifx_lo_specget_flags() 函数检索创建时间标志的当前值。

如果没有为其中一个标志组指定值，则数据库服务器将使用继承层次结构来确定此信息。

使用 LO 特定结构的 ESQL/C 函数

下表显示访问 LO 特定结构的 GBase 8s ESQL/C 库函数。

ESQL/C 库函数	意义	请参阅
ifx_lo_col_info()	使用列级别存储特征更新 LO 特定结构	ifx_lo_col_info() 函数
ifx_lo_create()	读取 LO 特定结构，以获取其超级的新智能大对象的存储特性	ifx_lo_create() 函数
ifx_lo_def_create_spec()	分配并初始化 LO 特定结构	ifx_lo_def_create_spec() 函数
ifx_lo_spec_free()	释放 LO 特定结构的资源	ifx_lo_spec_free() 函数
ifx_lo_specget_estbytes()	从 LO 特定结构获取估计的字节数	ifx_lo_specget_estbytes() 函数
ifx_lo_specget_extsz()	从 LO 特定结构获取分配 extent 大小	ifx_lo_specget_extsz() 函数
ifx_lo_specget_flags()	从 LO 特定结构获取创建时间标志	ifx_lo_specget_flags() 函数
ifx_lo_specget_maxbytes()	从 LO 特定结构获取最大字节数	ifx_lo_specget_maxbytes() 函数

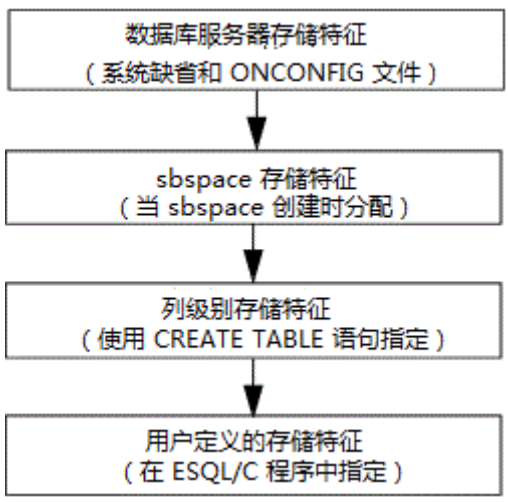
ESQL/C 库函数	意义	请参阅
ifx_lo_specget_sbspace()	从 LO 特定结构获取 sbspace 名称	ifx_lo_specget_sbspace() 函数
ifx_lo_specset_estbytes()	设置 LO 特定结构的字节估计数	ifx_lo_specset_estbytes() 函数
ifx_lo_specset_extsz()	设置 LO 特定结构中的分配 extent 大小	ifx_lo_specset_extsz() 函数
ifx_lo_specset_flags()	设置 LO 特定结构中的创建时间标志	ifx_lo_specset_flags() 函数
ifx_lo_specset_maxbytes()	设置 LO 特定结构中的最大字节数	ifx_lo_specset_maxbytes() 函数
ifx_lo_specset_sbspace()	设置 LO 特定结构中的 sbspace 名称	ifx_lo_specset_sbspace() 函数
ifx_lo_stat_cspec()	将存储特性返回给指定智能大对象的 LO 特定结构	ifx_lo_stat_cspec() 函数

获得存储特征

在您使用 ifx_lo_def_create_spec() 函数分配 LO 特定结构后， 您必须保证在创建智能大对象时此结构包含适当的存储特征。

GBase 8s 使用继承层次结构获取存储特征。下图显示了智能大对象存储特征的继承层次结构。

图 1. 存储特征的继承层次结构



系统指定的存储特征

GBase 8s 使用以下任一存储特征的设置作为系统指定的存储特征：

如果存储智能大对象的 sbspace 为特定存储特征指定了值，则数据库服务器使用 sbspace 值作为系统指定的特征。

数据库管理器（DBA）使用 gspaces 实用程序定义 sbspace 的存储特征。

如果存储智能大对象的 sbspace 没有为特定的存储特征指定值，则数据库服务器使用系统缺省值作为系统指定的存储特征。

数据库服务器内部定义存储特征的系统缺省值。然而，您可以使用 onconfig 文件的 SBSPACENAME 配置参数指定缺省 sbspace 名。并且，应用程序可以调用 ifx_lo_col_info() 或 ifx_lo_specset_sbspace() 提供 LO 特定结构中的目标 sbspace。

重要： 如果您未指定 sbspacename 配置参数并且 LO 特定结构不包含目标 sbspace 的名称，则会发生错误。

建议您使用系统指定的磁盘存储信息的值。大多数应用程序不需要更改这些系统指定的存储特征。

要对新的智能大对象使用系统指定的存储特征，请按以下步骤操作：

使用 ifx_lo_def_create_spec() 函数分配 LO 特定结构并初始化此结构为空值。

将 LO 特定结构传递给 **ifx_lo_create_function** 函数来创建智能大对象的实例。

ifx_lo_create() 函数创建一个智能大对象，其具有作为参数接收的 LO 特定结构中的存储特征。因为之前对 ifx_lo_def_create_spec() 的调用在此结构中存储空值，所以数据库服务器会将系统指定的特征分配给智能大对象的新实例。

列级别存储特征

数据库管理员（DBA）使用 CREATE TABLE 语句分配列级别的存储特征。CREATE TABLE 的 PUT 子句为智能大对象列（CLOB 或 BLOB）指定存储特征。**syscolattribs** 系统目录表存储列级别存储特征。

ifx_lo_col_info() 函数为智能大对象列获取列级别的存储特征。要对新的智能大对象示例使用列级别的存储特征，请按以下步骤操作：

使用 ifx_lo_def_create_spec() 函数分配 IO 特定结构并初始化此结构为空值。

将此 LO 特定结构传递给 ifx_lo_col_info() 函数并指定期望的列和表名作为参数。

该函数将列级别存储特征存储到指定的 LO 特定结构中。

将相同的 LO 特定结构传递给 ifx_lo_create() 函数来创建智能大对象的实例。

当 ifx_lo_create() 函数接收 LO 特定结构作为参数时，此结构包含之前调用 ifx_lo_col_info() 的列级存储特征。因此，数据库服务器将这些列级别特征分配给智能大对象的新实例。

当您使用列级别存储特征时，通常不需要为智能大对象提供 sbspace 的名称。sbspace 名称在 CREATE TABLE 语句的 PUT 子句指定中或由 ONCONFIG 文件中的 SBSPACENAME 参数指定。

用户定义的存储特征

GBase 8s ESQL/C 应用程序可以为新的智能大对象定义一组独特的存储特征，如下所示：

对于存储在列中的智能大对象，当创建智能大对象的实例时，可以覆盖列的某些存储特征。

如果应用程序不覆盖某些或所有这些特征，则智能大对象使用列级别存储特征。

可以为每个智能大对象指定更广泛的特征集，因为智能大对象不受表列属性约束。

如果应用程序程序员不覆盖某些或所有的这些特征，则智能大对象继承系统指定的存储特征。

要指定用户定义的存储特征，请为 LO 特定结构使用适当的 GBase 8s ESQL/C 访问程序函数。

释放 LO 特定结构

完成 LO 特定结构指定后，使用 ifx_lo_spec_free() 函数释放分配给它的资源。当资源被释放时，它们可以重新分配给您的程序所需的其它结构。

LO 指针结构

要对读取和写入操作打开智能大对象，GBase 8s ESQL/C 程序必须具有用于此智能大对象的 LO 指针结构。此结构包含磁盘地址和智能大对象的一个唯一十六进制标识符。

要为新的智能大对象创建 LO 指针结构，请使用 ifx_lo_copy_to_file() 函数。ifx_lo_copy_to_file() 函数执行以下操作：

它为新的智能大对象初始化 LO 指针结构，其指针作为参数。

新的智能大对象具有您提供的 LO 特定结构指定的存储特征。

它以指定的访问模式打开新的智能大对象，并返回在智能大对象上后续操作所需的 LO 文件描述符。

在调用 ifx_lo_create() 函数之前，必须调用 ifx_lo_def_create_spec() 创建新的智能大对象。

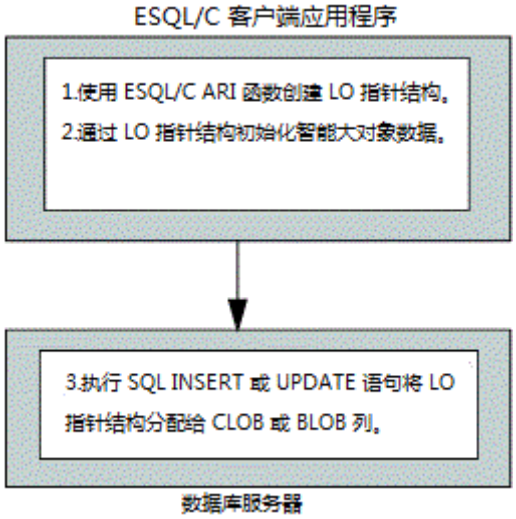
存储智能大对象

GBase 8s ESQL/C 程序通过 LO 指针结构访问智能大对象。使用 LO 指针结构的 ESQL/C 函数表中的 GBase 8s ESQL/C 库函数接受 LO 指针结构为参数。通过 LO 指针结构，这些函数允许您创建并控制智能大对象，而不用将它绑定到数据库行。

INSERT 或 UPDATE 语句不会执行智能大对象的实际输入。然而，它确实为应用程序提供了识别要与列相关联的哪个智能大对象数据的方法。数据库表中的 CLOB 或 BLOB 列存储智能大对象的 LO 指针。因此。当您存储 CLOB 或 BLOB 列时，可以为 INSERT 或 UPDATE 语句的 **ifx_lo_t** 主机变量中的列提供 LO 指针结构。因此，您将 CLOB 和 BLOB 值的主变量声明为 LO 指针结构。

下图显示了 GBase 8s ESQL/C 客户端应用程序如果将智能大对象的数据传输给数据库服务器。

图 2. 将来自客户端应用程序的智能大对象的数据传输到数据库服务器



只要 LO 指针结构存在， LO 指针结构标识的智能大对象就会存在。当您在数据库中存储 LO 指针结构时，数据库服务器可以确保在适当时将智能大对象释放。

当您检索一行，然后更新该行包含的智能大对象时，数据库服务器将专门绑定该行以更新智能大对象。此外，如果智能大对象需要很长时间才能更新或创建智能大对象（无论日志记录是否启用，以及它们是否与表行相关联），都会为长事务创建长期更新。

ifx_lo_t 结构

LO 指针结构 ifx_lo_t 用作智能大对象的引用。它提供与安全相关的信息，并保存关于智能大对象的实际磁盘位置的信息。

locator.h 头文件定义 LO 指针结构，因此您必须在访问此结构的 GBase 8s ESQL/C 程序中包含 locator.h 文件。

重要： LO 指针结构 **ifx_lo_t** 对 GBase 8s ESQL/C 程序来说是一个不透明结构。您不能直接访问它的内部结构。**ifx_lo_t** 的内部结构可能会发生改变。因此，要创建便捷式代码，使用正确的 GBase 8s ESQL/C 库函数来使用此结构。

LO 指针结构，并非 CLOB 或 BLOB 数据，存储在数据库的 CLOB 或 BLOB 列中。因此，诸如 INSERT 和 SELECT 的 SQL 语句接受 LO 指针结构作为智能大对象列的列值。声明 GBase 8s ESQL/C 主机变量保存智能大对象的值作为 **ifx_lo_t** 结构。

使用 LO 指针结构的 ESQL/C 函数

下表显示了访问 LO 指针结构的 GBase 8s ESQL/C 库函数，以及如何访问。

ESQL/C 库函数	意义	请参阅
ifx_lo_copy_to_file()	将 LO 指针结构标识的智能大对象复制到操作系统文件	ifx_lo_copy_to_file() 函数
ifx_lo_create()	为其创建的新智能大对象初始化 LO 指针结构，并返回此智能大对象的 LO 文件描述符。	ifx_lo_create() 函数
ifx_lo_filename()	返回文件的名称，其中 ifx_lo_copy_to_file() 函数将存储 LO 指针标识的智能大对象	ifx_lo_filename() 函数
ifx_lo_from_buffer()	将指定数量的字节从用户定义的缓冲区复制到 LO 指针结构引用的智能大对象中	ifx_lo_from_buffer() 函数
ifx_lo_release()	告知数据库服务器释放与 LO 指针结构引用的临时智能大对象相关联的资源	ifx_lo_from_buffer() 函数
ifx_lo_to_buffer()	从 LO 指针结构引用的智能大对象中将指定数量的字节复制到用户定义的缓冲区中	ifx_lo_to_buffer() 函数

LO 文件描述符

LO 文件描述符是一个整数值，以标识开放的智能大对象。

LO 文件描述符类似于操作系统文件的文件描述符。它用于 I/O 处理服务器中的智能大对象的数据。LO 文件描述符以搜索位置 0 开始。在接受 LO 文件描述符的 GBase 8s ESQL/C 库函数中使用 LO 文件描述符。

使用 LO 文件描述符的 ESQL/C 库函数

下表显示了访问 LO 文件描述符的 GBase 8s ESQL/C 库函数。

ESQL/C 库函数	意义	请参阅
ifx_lo_close()	关闭 LO 文件描述符标识的智能大对象，并释放 LO 文件描述符	ifx_lo_close() 函数
ifx_lo_copy_to_lo()	将操作系统文件复制到 LO 文件描述符标识的打开智能大对象中	ifx_lo_copy_to_lo() 函数
ifx_lo_create()	创建并打开新的智能大对象，	ifx_lo_create() 函数

ESQL/C 库函数	意义	请参阅
	并返回 LO 文件描述符	
ifx_lo_open()	打开智能大对象，并返回 LO 文件描述符	ifx_lo_open() 函数
ifx_lo_read()	读取来自 LO 文件描述符标识的打开智能大对象的数据	ifx_lo_read() 函数
ifx_lo_readwithseek()	在 LO 文件描述符标识的打开智能大对象中搜索指定的文件位置，然后从此位置读取	ifx_lo_readwithseek() 函数
ifx_lo_seek()	移动 LO 文件描述符标识的打开智能大对象的文件位置	ifx_lo_seek() 函数
ifx_lo_stat()	获取 LO 文件描述符标识的打开智能大对象的状态信息	ifx_lo_stat() 函数
ifx_lo_tell()	确定 LO 文件描述符标识的打开智能大对象中当前文件位置	ifx_lo_tell() 函数
ifx_lo_truncate()	以指定的偏移量截断 LO 文件描述符标识的开放智能大对象	ifx_lo_truncate() 函数
ifx_lo_write()	将数据写入到 LO 文件描述符标识的打开智能大对象中	ifx_lo_write() 函数
ifx_lo_writewithseek()	在 LO 文件描述符标识的打开智能大对象中搜索指定的文件位置，让将数据写入到此位置	ifx_lo_writewithseek() 函数

2.8.2 创建智能大对象

请按照以下步骤创建智能大对象：

使用 ifx_lo_def_create_spec() 函数分配 LO 特定结构。

确保 LO 特定结构包含新的智能大对象期望的存储特征。

创建新的智能大对象的 LO 指针结构，然后使用 ifx_lo_create() 函数打开智能大对象。

将新智能大对象的数据写入到打开的智能大对象中，使用 ifx_lo_write() 或 ifx_lo_writewithseek() 函数。

保存数据库列中的新的智能大对象。

使用 ifx_lo_spec_free() 函数释放 LO 特定结构。

2.8.3 访问智能大对象

要访问智能大对象，请按以下步骤操作：

使用 SELECT 语句将数据库中的智能大对象选择到 **ifx_lo_t** 主机变量中。

使用 ifx_lo_open() 函数打开智能大对象。

执行适当的读取或写入操作来更新智能大对象的数据。

使用 ifx_lo_close() 函数关闭智能大对象。

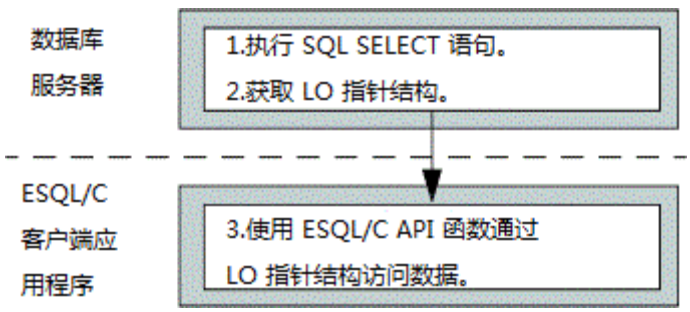
选择智能大对象

SELECT 语句不会执行智能大对象数据的实际输出。但是，它建立一种方法，使应用程序标识智能大对象，以致于它可以发出 GBase 8s ESQL/C 库函数可以在智能大对象上打开、读取、写入或执行其它操作。

数据库表中的 CLOB 或 BLOB 列包含智能大对象的 LO 指针结构。因此，当您将 CLOB 或 BLOB 列选择到 **ifx_lo_t** 主机变量时，SELECT 语句返回 LO 指针结构。出于此原因，声明 CLOB 和 BLOB 值的主机变量为 LO 指针结构。

下图显示了数据库服务器如何将智能大对象的数据传输到 GBase 8s ESQL/C 客户端应用程序中。

图 3. 将智能大对象的数据从数据库传输到 GBase 8s ESQL/C 客户端应用程序中



打开智能大对象

当您打开智能大对象时，获得智能大对象的 LO 文件描述符。通过 LO 文件描述符，您可以访问智能大对象的数据，就像它在操作系统文件中一样。

访问模式

当您打开智能大对象时，请为数据指定访问模式。

访问方式决定开放智能大对象上的哪种读取和写入操作有效。可以使用 locator.h 文

件定义的访问模式常量指定访问模式。

下表显示了访问模式以及它们对应的定义 ifx_lo_open() 和 ifx_lo_create() 函数支持的常量。

表 3. 智能大对象访问模式标志

访问模式	意义	访问模式常量
只读模式	只能对数据进行读取操作。	LO_RDONLY
脏读模式	仅对于 ifx_open() ，允许您读取智能大对象未提交的页。在将模式设置为 LO_DIRTY_READ 后，无法写入智能大对象。当您设置此标志时，将为当前的事务隔离模式重置为智能大对象的脏读取。 不要对从脏读取模式中的智能大对象获取的数据进行修改。	LO_DIRTY_READ
只写模式	只能对数据进行写入操作。	LO_WRONLY
附加模式	适用于 LO_WRONLY 或 LO_RDWR。在每次写入之前将位置指针设置为对象的末尾。将您写入的任何数据附加到智能大对象的末尾。如果单独使用 LO_APPEND ，则只打开该对象进行阅读。	LO_APPEND
读/写 模式	对数据的读取和写入操作都是有效的。	LO_RDWR
缓冲区访问	使用标准数据库服务器缓冲池。	LO_BUFFER
轻量 I/O	从数据库服务器的会话池中使用专用缓冲区。	LO_NOBUFFER
锁定全部	指定整个智能大对象将发生锁定。	LO_LOCKALL
锁定字节范围	指定锁定将发生在一定范围内，这将通过 ifx_lo_lock() 函数在放置锁定时指定。	LO_LOCKRANGE

提示： 智能大对象的这些访问模式标志在 UNIX(TM) 系统 V 访问模式之后被视图化。

设置脏读取访问模式

要对智能大对象设置脏读取访问模式，使用 SET ISOLATION 语句为此事务设置，或者在打开智能大对象时设置 LO_DIRTY_READ 访问模式。当您打开智能大对象时设置

LO_DIRTY_READ 访问模式仅影响智能大对象的读取模式，而不影响整个事务。换句话说，如果您的事务在提交读取模式下执行，则可以使用 LO_DIRTY_READ 访问模式以脏读取模式打开智能大对象，而不需更改事务的隔离模式。

LO_APPEND 标志

当仅使用 LO_APPEND 打开智能大对象时，智能大对象以只读的方式打开。搜索查找会移动文件指针但是智能大对象的写操作失败，并且文件指针在写入前不会进行位置移动。读取操作发生在文件指针所在的位置，然后移动文件指针。

可以使用其它访问模式屏蔽 LO_APPEND 标志。在任何这些 OR 组合中，搜索操作保持不受影响。下表显示每个 OR 组合对读写操作的影响。

OR 操作	读操作	写操作
LO_RDONLY | LO_APPEND	在文件位置发生，然后将文件位置移动到读取数据的末尾	失败并且不会移动文件位置
LO_WRONLY | LO_APPEND	失败，并且不会移动文件扩展名	将文件位置移动到智能大对象的末尾，然后写数据；文件位置在写入数据结束后
LO_RDWR | LO_APPEND	在文件位置发生，然后将文件位置移动到读取数据的末尾	将文件位置移动到智能大对象的末尾，然后写数据；文件位置在写入数据结束后

轻量级 I/O

当该数据库服务器访问智能大对象时，它使用来自缓冲池的缓冲区用于缓存访问。未缓存的访问称为 轻量级 I/O。轻量级 I/O 使用专用缓冲区而不是缓冲池来保存智能大对象。这些专用缓冲区被分配出数据库服务器会话池。

轻量级 I/O 允许您绕过数据库服务器用于管理缓冲池的最近最少使用（LRU）队列的开销。

当使用 ifx_lo_create() 函数创建智能大对象或者当使用 ifx_lo_open() 函数打开特定的智能大对象时，可以通过将标识参数设置为 LO_NOBUFFER 来指定轻量级 I/O 。要指定缓冲访问（缺省值），使用 LO_BUFFER 标志。

重要： 当使用轻量级 I/O 时，请注意以下问题：

使用 ifx_lo_close() 关闭智能大对象，当您完成它们以释放分配给专用缓冲区的内存。

所有使用轻量级 I/O 为指定的智能大对象的打开都共享相同的专用缓冲区。因此，一

个操作可能导致缓冲区中的页面被刷新，而其他操作对象存在于缓冲区中。

数据库服务器对从轻量级 I/O 到缓冲 I/O 的切换加了以下限制：

如果智能大对象未打开，可以使用 ifx_lo_alter() 函数将智能大对象从轻量级 I//O（LO_NOBUFFER）切换到缓冲 I/O（LO_BUFFER）。但是，如果您尝试将使用缓冲 I/O 的智能大对象切换为使用轻量级 I/O 则 ifx_lo_alter() 生成错误。

除非您首先使用 ifx_lo_alter() 将访问模式更改为缓冲访问（LO_BUFFER），否则只能使用 LO_NOBUFFER 访问模式标志，创建轻量级 I/O。如果一个打开指定 LO_BUFFER，则数据库服务器忽略此标志。

只有在以只读方式打开对象时，才能打开受缓冲访问（LO_BUFFER）创建的智能大对象，并使用 LO_NOBUFFER 标志。如果您尝试写入此对象，则数据库服务器返回错误。要写入智能大对象，必须关闭它然后使用 LO_BUFFER 标志和允许写操作的访问标志重新打开它。

可以使用数据库服务器实用程序 gspaces 为 sbspace 中的所有智能大对象指定轻量级 I/O。

智能大对象锁定

当打开智能大对象时，数据库服务器将锁定整个智能大对象或指定的一系列字节。以防止同时访问智能大对象数据。智能大对象上的锁与行锁不同。如果从行检索智能大对象，则数据库服务器可能持有行锁和智能大对象锁。数据库服务器锁定智能大对象数据是因为许多列包含相同的智能大对象数据。使用访问模式标志 LO_RDONLY 、LO_DIRTY_READ 、LO_APPEND 、LO_WRONLY 、LO_RDWR 和 LO_TRUNC 指定智能大对象锁的模式。您将这些标志传递给 ifx_lo_open() 和 ifx_lo_create() 函数。当指定 LO_RDONLY 时，数据库服务器在智能大对象上放置一个共享锁。当指定 LO_DIRTY_READ 时，数据库服务器不会在智能大对象删放置锁。如果指定任何其它访问模式标志，则数据库服务器获得一个更新锁，它会在首次写入或其它操作时提升为互斥锁。

共享和更新锁（只读模式或更改发生前的写入模式）会一直保持直到您的程序执行以下操作之一：

关闭智能大对象

提交事务或回滚它

即使您关闭智能大对象互斥锁也将一直保持直到事务结束。

重要： 即使智能大对象保持打开，您在事务结束时也会失去锁定。当数据库服务器检测到智能大对象没有活动锁定时，当智能大对象发生第一次访问时，它会自动获得新的锁定。它获得的锁是基于智能大对象的原始打开模式。

锁的范围

在智能大对象上放置锁定时，可以锁定整个智能大对象，或者锁定字节范围。字节范围锁定允许您只锁定将在智能大对象中影响的字节范围。

两个访问模式标志 LO_LoCKALL 和 LO_LOCKRANGE，使您能够知道将用于智能大对象的缺省锁类型。可以使用 ifx_lo_specset_flags() 设置它们，使用 ifx_specget_flags() 检索。LO_LOCKALL 标志指定整个智能大对象被锁定；LO_LOCKRANGE 标志指定您对智能大对象使用字节范围锁定。

可以使用 ifx_lo_alter() 函数将缺省范围从一种类型更改为另一种类型。可以通过在访问模式标志 ifx_lo_open() 的访问模式标志中设置 LO_LOCKALL 或 LO_LOCKRANGE 标志来覆盖缺省范围。

ifx_lo_lock() 函数允许您锁定要为智能大对象访问的字节范围，并且 ifx_lo_unlock() 函数允许您在完成时解锁字节。

智能大对象打开的持续时间

当您使用 ifx_lo_create() 函数或 ifx_lo_open() 函数打开智能大对象后，它仍保持打开状态，直到发生以下事件之一：

ifx_lo_close() 函数关闭智能大对象。

会话结束。

当前事务的结束不会关闭智能大对象。但是，它会释放智能大对象上的锁。使用您的应用程序在它们完成后立即关闭智能大对象。开启智能大对象不必要地消耗系统内存。关闭足够数量的智能大对象打开可以最终产生超出内存的条件。

删除智能大对象

如果应用程序打开了智能大对象，则智能大对象不会被删除直到当前会话提交并且智能大对象被关闭。

修改智能大对象

按照以下步骤修改智能大对象的数据：

在开放的智能大对象中读取和写入数据，直到数据库准备好保存。

使用 UPDATE 或 INSERT 语句将智能大对象的 LO 指针存储到数据库中。

从智能大对象读取

ifx_lo_read() 和 ifx_lo_readwithseek()GBase 8s ESQL/C 库函数从智能大对象读取数据。

它们都从打开的智能大对象中读取指定数量的字节到用户定义的字符缓冲区。ifx_lo_read() 函数在当前文件位置开始读取操作。可以使用 ifx_lo_seek() 函数指定读取的起始位置文件，您可以使用 ifx_lo_tell() 函数调用执行查找和读取操作。ifx_lo_readwithseek() 函数通过单个函数调用执行查找和读取操作。

ifx_lo_read() 和 ifx_lo_readwithseek() 函数需要一个有效的 LO 文件描述符标识要读取的智能大对象。您可以使用 ifx_lo_open() 或 ifx_lo_create() 函数获取 LO 文件描述符。

写入数据到智能大对象

ifx_lo_write() 和 ifx_lo_writewithseek()GBase 8s ESQL/C 库函数将数据写入到打开的智能大对象中。它们都将来自用户定义的字符缓冲区的特定编号写入到打开的智能大对象中。ifx_lo_write() 函数在当前文件位置开始写入操作。可以使用 ifx_lo_seek() 函数指定写入的起始文件位置，使用 ifx_lo_tell() 函数获取当前文件位置。ifx_lo_writewithseek() 函数使用一个函数调用执行搜索和写入操作。

ifx_lo_write() 和 ifx_lo_writewithseek() 函数需要有效的 LO 文件描述符来标识要写的智能大对象。使用 ifx_lo_open() 或 ifx_lo_create() 函数获取 LO 文件描述符。

关闭智能大对象

在完成对智能大对象的读取和写入操作后，使用 ifx_lo_close() 函数释放分配给它的资源。当资源被释放时，它们可以重新分配给您需要的其它结构。此外，LO 文件描述符可以重新分配给其它智能大对象。

2.8.4 获取智能大对象的状态

要获得智能大对象的状态信息，请按以下步骤操作：

获取有效的 LO 指针结构到您想要状态的智能大对象。

使用 ifx_lo_stat() 函数分配并填充 LO 状态结构。

使用适当的 GBase 8s ESQL/C 访问程序函数获取您需要的状态信息。

释放 LO 状态结构。

获取有效的 LO 指针结构

可以获取任何具有有效 LO 指针结构的智能大对象的状态信息。可以执行以下任一步骤来获取 LO 指针结构：

从数据库表选择 CLOB 或 BLOB 列。

创建新的智能大对象。

分配和访问LO 状态结构

LO 状态结构存储智能大对象的状态信息。本节描述如何分配和访问 LO 状态结构。

分配 LO 状态结构

ifx_lo_stat() 函数执行以下任务：

它分配一个新的 LO 状态结构，将您提供的指针作为参数。

它使用所提供的 LO 文件描述符的智能大对象的所有状态信息初始化 IO 状态结构。

访问 LO 状态结构

LO 状态结构 **ifx_lo_stat_t** 存储 GBase 8s ESQL/C 程序中智能大对象的状态信息。locator.h 头文件定义 LO 状态结构，所以您必须在访问此结构的 GBase 8s ESQL/C 程序中包含 locator.h 文件。

重要： LO 状态结构 **ifx_lo_stat_t** 对于 GBase 8s ESQL/C 程序是不透明的。不要直接访问它的内部结构。**ifx_lo_stat_t** 的内部结构可能在以后的版本中有所变更。因此，要创建便捷式代码，一般对此结构使用 GBase 8s ESQL/C 访问程序函数来获得和存储 LO 状态结构的值。

下表显示了对应于 GBase 8s ESQL/C 访问函数的状态信息。

表 4. LO 状态结构中的状态信息

磁盘存储信息	描述	ESQL/C 访问程序函数
上次访问时间	上一次访问智能大对象的时间（以秒为单位）。 仅当此智能大对象设置了 LO_KEEP_LASTACCESS_TIME 标识时，此值才可用	ifx_lo_stat_atime()
存储特征	智能大对象的存储特征。 这些特征都存储在 LO 特定结构中对 LO 特定结构使用 GBase 8s ESQL/C 存储程序函数（请参阅 表 1和 表 1）来获得此信息。	ifx_lo_stat_cspec()

磁盘存储信息	描述	ESQL/C 访问程序函数
最后一次更改状态	智能大对象最后一次更改状态的时间（以秒为单位）。 状态的更改包括更新、所有权变更以及引用次数的更改。	ifx_lo_stat_ctime()
上一次修改时间（秒）	上一次修改智能大对象的时间（以秒为单位）。	ifx_lo_stat_mtime_sec()
引用次数	引用智能大对象的次数。	ifx_lo_stat_refcnt()
大小	智能大对象的大小（以字节为单位）。	ifx_lo_stat_size()

时间值（例如上次访问时间和上次更改时间）可能与系统时间略有不同。这种差异是由于数据库服务器用来从操作系统获取时间的算法。

释放 LO 状态结构

在使用完毕 LO 状态结构后，使用 ifx_lo_stat_free() 函数释放它的资源。当资源被释放，它们可以被重新分配给程序需要的其它结构。

2.8.5 更改智能大对象列

可以使用 ALTER TABLE 语句的 PUT 子句更改 CLOB 或 BLOB 列的存储特征和存储位置。可以更改存储列的 sbspace ，并且还会执行循环分段，这会导致 CLOB 或 BLOB 列中的智能大对象在一系列指定的 sbspace 之间分配。例如，以下示例中的 ALTER TABLE 语句将 **advert.picture** 列的初始位置从 **s9_sbspc** 更改为 **s10_sbspc** 和 **s11_sbspc**。ALTER TABLE 语句还更改该列的特征：

```
advert          ROW (picture BLOB, caption VARCHAR(255, 65)),
                :

                PUT advert IN (s9_sbspc)
                (EXTENT SIZE 100)

                ALTER TABLE catalog
                PUT advert IN (s10_sbspc, s11_sbspc)
                (extent size 50, NO KEEP ACCESS TIME);
```

当您更改智能大对象列的存储位置或存储特征时，该更改仅适用于实例创建的新列。

不会影响列的现有的智能大对象的存储特征和存储位置。

2.8.6 迁移简单大对象

要将简单大对象迁移到智能大对象，请将 TEXT 数据转型为 CLOB 数据，BYTE 数据转型为 BLOB 数据。可以使用转型语法（例如：bytecolblobcol）将简单大对象迁移到智能大对象。以下示例将 BYTE 列 **cat_picture** 从 **stores7** 数据库中的 **catalog** 表迁移到智能大对象函数的示例中所述的备用 **catalog** 表中的 **advert** 行类型中的 BLOB 字段 **picture**：

```
update catalog set advert = ROW ((SELECT cat_picture::blob
        FROM stores7:catalog WHERE catalog_num = 10027), pwd
        advert.caption)
        WHERE catalog_num = 10027
```

还可以使用 ALTER TABLE 语句的 MODIFY 子句将 TEXT 或 BYTE 列更改为 CLOB 或 BLOB 列。当使用 ALTER TABLE 语句的 MODIFY 子句时，数据库服务器隐式地将旧的数据类型转型为新的数据类型以创建 ClOB 或 BLOB 列。

例如：如果您想将 **stores7** 数据库 **catalog** 表中的 **cat_descr** 列更改为 TEXT 列或 BYTE 列，那么可以使用类似于以下语句的构造：

```
ALTER TABLE catalog modify cat_descr CLOB,
        PUT cat_descr in (sbspc);
```

2.8.7 智能大对象的 ESQL/C API

智能大对象的 GBase 8s ESQL/C API 允许应用程序就像访问操作系统文件一样访问智能大对象。

不适合内存的智能大对象不必被读入文件，然后从文件中访问；它可以一次访问一个。GBase 8s ESQL/C 应用程序通过下表中的 GBase 8s ESQL/C 库函数访问智能大对象。

ESQL/C 函数	描述	请参阅
ifx_lo_alter()	更改现有智能大对象的存储特征	ifx_lo_alter() 函数
ifx_lo_close()	关闭打开的智能大对象	ifx_lo_close() 函数
ifx_lo_col_info()	检索 LO 特定结构中的列级别存储特征	ifx_lo_col_info() 函数
ifx_lo_copy_to_file()	将智能大对象复制到操作系统文件中	ifx_lo_copy_to_file() 函数

ESQL/C 函数	描述	请参阅
ifx_lo_copy_to_lo()	将操作系统文件复制到打开的智能大对象中	ifx_lo_copy_to_lo() 函数
ifx_lo_create()	创建智能大对象的 IO 指针结构	ifx_lo_create() 函数
ifx_lo_def_create_spec()	分配 LO 特定结构并将它的字段初始化为空值	ifx_lo_def_create_spec() 函数
ifx_lo_filename()	返回生成的文件名,给定一个 LO 指针结构和一个文件规范	ifx_lo_filename() 函数
ifx_lo_from_buffer()	将指定数量的字节从用户定义的缓冲区复制到智能大对象中	ifx_lo_from_buffer() 函数
ifx_lo_open()	打开现有的智能大对象	ifx_lo_open() 函数
ifx_lo_read()	从打开的智能大对象中读取指定字节数	ifx_lo_read() 函数
ifx_lo_readwithseek()	在打开的智能大对象中寻找指定位置，并读取指定的字节数	ifx_lo_readwithseek() 函数
ifx_lo_release()	释放提交到临时智能大对象的资源	ifx_lo_release() 函数
ifx_lo_seek()	在打开的智能大对象中设置下一个读或写的搜索位置	ifx_lo_seek() 函数
ifx_lo_spec_free()	释放分配到 LO 特定结构的资源	ifx_lo_spec_free() 函数
ifx_lo_specget_estbytes()	获取智能大对象的建立大小（以字节为单位）	ifx_lo_specget_estbytes() 函数
ifx_lo_specget_extsz()	获取智能大对象的分配 extent 大小	ifx_lo_specget_extsz() 函数
ifx_lo_specget_flags()	获取智能大对象的创建时间标志	ifx_lo_specget_flags() 函数
ifx_lo_specget_maxbytes()	获取智能大对象的最大大小	ifx_lo_specget_maxbytes() 函数
ifx_lo_specset_sbspace()	获取智能大对象的	ifx_lo_specget_sbspace(

ESQL/C 函数	描述	请参阅
	sbspace 名称t	) 函数
ifx_lo_specset_estbytes()	设置智能大对象的估计大小（以字节为单位）	ifx_lo_specset_estbytes() 函数
ifx_lo_specset_extsz()	设置智能大对象的分配 extent 大小	ifx_lo_specset_extsz() 函数
ifx_lo_specset_flags()	设置智能大对象的创建时间标志	ifx_lo_specset_flags() 函数
ifx_lo_specset_maxbytes()	设置智能大对象的最大大小	ifx_lo_specset_maxbytes() 函数
ifx_lo_specset_sbspace()	设置智能大对象的 sbspace 名称t	ifx_lo_specset_sbspace() 函数
ifx_lo_stat()	获取打开的智能大对象的状态信息	ifx_lo_stat() 函数
ifx_lo_stat_atime()	返回智能大对象的上一次访问时间	ifx_lo_stat_atime() 函数
ifx_lo_stat_cspec()	返回智能大对象的存储特征	ifx_lo_stat_cspec() 函数
ifx_lo_stat_ctime()	返回智能大对象的状态上一次更改的时间	ifx_lo_stat_ctime() 函数
ifx_lo_stat_free()	释放分配到 LO 状态结构的资源	ifx_lo_stat_free() 函数
ifx_lo_stat_mtime_sec()	返回最后一次修改智能大对象的时间（以秒为单位）	ifx_lo_stat_mtime_sec() 函数
ifx_lo_stat_refcnt()	返回智能大对象的引用次数	ifx_lo_stat_refcnt() 函数
ifx_lo_stat_size()	返回智能大对象的大小	ifx_lo_stat_size() 函数
ifx_lo_tell()	返回打开智能大对象当前搜索位置	ifx_lo_tell() 函数
ifx_lo_to_buffer()	将指定数量的字节从智能大对象复制到用户定义的缓冲区中	ifx_lo_to_buffer() 函数
ifx_lo_truncate()	将智能大对象截断到特定的偏移量	ifx_lo_truncate() 函数

ESQL/C 函数	描述	请参阅
ifx_lo_write()	将一个指定数量的字节写入到一个打开的智能大对象中	ifx_lo_write() 函数
ifx_lo_writewithseek()	在打开的智能大对象中寻找指定的位置并写入指定的字节数	ifx_lo_writewithseek() 函数

2.9 复杂数据类型

这些主题描述如何在 GBase 8s ESQL/C 程序中使用 **collection** 和 **row** 数据类型。

这些主题的信息仅在您使用 GBase 8s 作为您的数据库服务器才适用。

这些 GBase 8s ESQL/C 数据类型访问复杂数据类型，如下表所示：

数据类型	ESQL/C 主机变量
集合类型：LIST 、MULTISET、SET	已归类的**集合**主机变量 未归类的**集合**主机变量
行类型：已命名和未命名	已归类的**行**主机变量 未归类的**行**主机变量

2.9.1 访问集合

GBase 8s 支持以下种类的集合：

SET 数据类型，存储唯一值并且没有顺序位置的元素的集合。

MULTISET 数据类型，存储可以是重复值并且没有顺序位置的元素的集合。

LIST 数据类型，存储可以是重复值并具有有序位置的元素的集合。

SQL 和 GBase 8s ESQL/C 使您能够使用 SQL *集合派生表*子句访问集合的元素，就像它们是表中的行。在 GBase 8s ESQL/C 中，集合派生表采用*集合变量的*形式。集合变量是您检索集合的主机变量。将集合检索到集合变量后，可以对其进行带有限制的选择、更新和删除操作。

重要： 当 SQL 语句引用集合变量时，GBase 8s ESQL/C 而不是数据库服务器处理此语句。

SQL 允许您通过将集合派生表实现为虚拟表来对集合执行只读（SELECT）操作。

访问集合派生表

当集合的 SELECT 语句不引用 GBase 8s ESQL/C 集合变量时，数据库服务器执行此查询。

例如，考虑以下模式：

```
create row type person(name char(255), id int);
        create table parents(name char(255), id int,
        children list(person not null));
```

可以使用以下 SELECT 语句从表 **parent** 选择孩子的名称和 ID：

```
select name, id from table(select children from parents
```

```
                where parents.id = 1001) c_table(name, id);
```

要执行此查询，数据库服务器创建一个虚拟表（**c_table**），它来自 **parents** 表中 **parents.id** 等于 1001的行的 **children** 列表。

集合派生表的优势

将集合查询为虚拟表而不是通过集合变量进行查询的优点在于虚拟表提供了更高效的访问。

相比之下，如果您要使用集合变量，可能需要分配多个变量和多个游标。例如，考虑以下模式：

```
EXEC SQL create row type parent_type(name char(255), id int,
        children list(person not null));
        EXEC SQL create grade12_parents(class_id int,
        parents set(parent_type not null));
```

可以如下 SELECT 语句所示的那样将集合派生表查询为虚拟表：

```
EXEC SQL select name into :host_var1
        from table((select children from table((select parents
        from grade12_parents where class_id = 1))
        p_table where p_table.id = 1001)) c_table
        where c_table.name like 'Mer%';
```

要使用集合变量执行相同的查询，您需要执行以下语句：

```
EXEC SQL client collection hv1;
        EXEC SQL client collection hv2;
        EXEC SQL int parent_id;

        EXEC SQL char host_var1[256];
        :

        EXEC SQL allocate collection hv1;
        EXEC SQL allocate collection hv2;

        EXEC SQL select parents into :hv1 from grade12_parents
        where class_id = 1;
        EXEC SQL declare cur1 cursor for select id, children
        from table(:hv1);
        EXEC SQL open cur1;
        for(;;)
        {
        EXEC SQL fetch cur1 into :parent_id, :hv2;
        if(parent_id = 1001)
        break;
        }
        EXEC SQL declare cur2 cursor for select name from
        table(:hv2));
        EXEC SQL open cur2;
        for(;;)
```

```
{
EXEC SQL fetch cur2 into :host_var1;
/* user needs to implement 'like' function */
if(like_function(host_var1, "Mer%"))
break;
}
```

集合派生表的限制

以下限制适用于查询作为虚拟表的集合派生表：

它不能是 INSERT 、DELETE 或 UPDATE 语句的目标。

它不能是可以更新的任何游标或视图的基础表。

它不支持序数。例如，它不支持以下语句：

```
select name, order_in_list from table(select children
                from parents where parents.id = 1001)
                with ordinality(order_in_list);
```

如果集合派生表的底层集合表达式求值为空值，则会出现错误。

它不能引用在同一 FROM 子句中引用的表的列。例如：它不支持以下语句，因为集合派生表I table(parents.children) 引用了 FROM 子句中的引用的表 parents：

```
select count(distinct c_id) from parents,
                        table(parents.children) c_table(c_name, c_id)
                        where parents.id = 1001
```

数据库服务器必须能够静态地确定底层集合表达式的类型。例如：数据库服务器不支持：TABLE(?)。

数据库服务器不支持对主机变量的引用，而不将其转换为已知的集合类型。例如：不必指定 TABLE(:hostvar)，您必须转换主机变量：

```
TABLE(CAST(:hostvar AS type))
                                TABLE(CAST(? AS type))
```

如果底层集合是列表，它将不会保留列表中的行的顺序。

声明集合变量

要访问具有集合类型（LIST 、MULTISET 或 SET）的列的元素作为其数据类型，请执行以下步骤：

删除 **collection** 主机变量，已归类和未归类的都删除。

给 **collection** 主机变量分配内存。

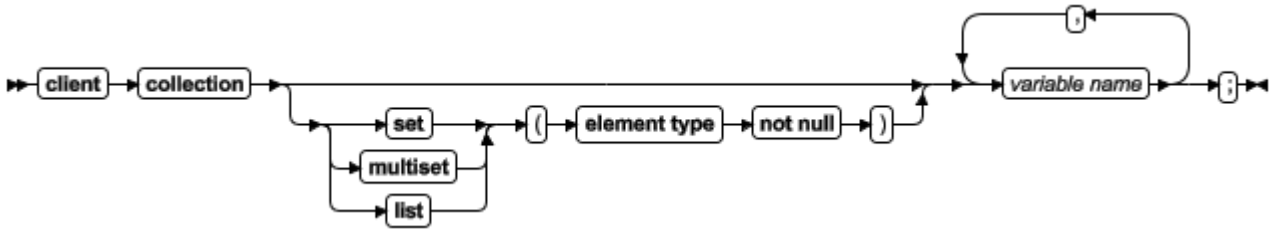
对 **collection** 主机变量执行任何 select 、insert 、update 或 delete 操作。

将 **collection** 主机变量的内容保存到集合列中。

集合数据类型的语法

使用**集合**数据类型来声明集合数据类型（SET 、MULTISET 或 LIST）列的主机变量。

如以下语法所示，必须使用 **collection** 关键字作为 **collection** 主机变量的数据类型。



元素	意义	限制	SQL 语法
element type	**collection** 变量中的元素的数据类型	可以是除 SERIAL 、SERIAL8 、BIGSERIAL 、TEXT 或 BYTE 之外的任何数据类型。	*GBase 8s SQL 指南：语法* 中的数据类型段
variable name	声明为 **collection** 变量的 GBase 8s ESQL/C 变量的名称		名称必须符合变量名称的语言规范规则。

集合变量可以是任何 SQL 集合类型：LIST 、MULTISET 或 SET。

重要： 当声明**集合**变量时，必须指定 **client** 关键字。

类型和无类型集合变量

GBase 8s ESQL/C 支持以下两种**集合**变量：

类型集合变量指定集合中的元素和集合本身的数据类型。

无类型集合变量不指定集合类型或元素类型。

类型集合变量

类型**集合**变量提供集合的提供了集合的准确描述。该声明指定集合的数据类型（SET 、MULTISET 或 LIST）以及**集合**变量的元素类型。

下图显示了三个类型**集合**变量的声明：

图: 类型集合变量示例

```
EXEC SQL BEGIN DECLARE SECTION;
                client collection list(smallint not null)
                list1;
                client collection set(row(
                x char(20),
                y set(integer not null),
                z decimal(10,2)) not null) row_set;
                client collection multiset(set(smallint
                not null)
```

```
not null) collection3;
EXEC SQL END DECLARE SECTION;
```

类型**集合**变量可以包含具有以下数据类型的元素：

任何内置数据类型（例如 INTEGER 、CHAR 、BOOLEAN 和 FLOAT）除 BYTE 、TEXT 、SERIAL 或 SERIAL8 之外。

集合数据类型，例如 SET 和 LIST，以创建嵌套集合

未命名的行类型（已命名行类型不可用）

不透明数据类型

当指定集合变量的元素类型时，请使用 SQL 数据类型而不是 GBase 8s ESQL/C 数据类型。例如： 图 1中显示的 **list1** 变量的声明，使用 SQL SMALLINT 数据类型而不是 GBase 8s ESQL/C **short** 数据类型来声明 LIST 变量，其元素是小整型。同样，对 CHAR 列使用 SQL 语法声明 SET 变量，其元素都是字符字符串，如下所示：

```
client collection set(char(20) not null) set_var;
```

重要： 必须在集合变量的元素类型上指定非空约束。

已命名的行类型不能作为**集合**变量的元素类型。但是，您可以指定未命名行类型的元素类型，其字段符合已命名行类型的那些字段。

例如：假定您的数据库具有已命名行类型 **myrow**，数据库表 **mytable** ，它们都如下定义：

```
CREATE ROW TYPE myrow
(
a int,
b float
);
CREATE TABLE mytable
(
col1 int8,
col2 set(myrow not null)
);
```

可以为 **mytable** 的 **col2** 列定义**集合**变量，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
client collection set(row(a int, b float) not null)
my_collection;
EXEC SQL END DECLARE SECTION;
```

只要两种数据类型兼容，就可以声明一个类型与集合列的元素类型不同的集合变量。如果数据库服务器能够在这两个元素类型之间转换，则在返回获取的集合时，它将自动执行此转换。

假设创建如下的 **tab1** 表：

```
CREATE TABLE tab1 (col1 SET(INTEGER NOT NULL))
```

可以声明类型集合变量，其元素类型符合（**set_int**）或其中之一的元素类型是兼容的

（**set_float**），如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
                client collection set(float not null) set_float;
                client collection set(integer not null) set_int;
                EXEC SQL END DECLARE SECTION;

                EXEC SQL declare cur1 cursor for select * from tab1;
                EXEC SQL open cur1;
                EXEC SQL fetch cur1 into:set_float;
                EXEC SQL fetch cur1 into :set_int;
```

当它执行第一个 FETCH 语句时，GBase 8s ESQL/C 客户端程序自动将列中的整数元素转换为 **set_float** 主机变量中的浮点值。如果在第一次获取后更改主机变量，则 GBase 8s ESQL/C 程序只能生成类型不兼容错误。在之前的代码段中，第二个 FETCH 语句生成类型不符错误，因为首次获取已经定义了元素类型为浮点。

在以下情况中使用类型**集合**变量：

当插入到派生表时（GBase 8s ESQL/C 需要知道是哪种类型）

当更改派生表中的元素时（GBase 8s ESQL/C 需要知道是哪种类型）

当需要服务器执行转型时。（GBase 8s ESQL/C 客户端将类型选项发送到数据库服务器，它尝试执行请求的转型操作。如果不可能，数据库服务器返回错误。）

将类型集合变量的声明与集合列的数据类型完全匹配。然后可以在 SQL 语句（INSERT 、DELETE 或 UPDATE）中或在集合派生表子句中直接使用此**集合**变量。

提示： 如果不知道您要访问的集合列的准确的数据类型，请使用无类型**集合**变量。

在单个声明行中，可以为同一类型的集合声明多个**集合**变量，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
                client collection multiset(integer not null) mset1, mset2;
                EXEC SQL END DECLARE SECTION;
```

不能在一个声明行中为不同的集合类型声明**集合**变量。

无类型集合变量

无类型**集合**变量提供了集合的一般描述。此声明仅包含**集合**关键字和变量名称。

下行声明了三个无类型**集合**变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection collection1, collection2;
        client collection grades;
        EXEC SQL END DECLARE SECTION;
```

无类型**集合**主机变量的优点是它在集合定义上提供更多的灵活性。对于无类型**集合**变量，在编译时不需要知道集合列的定义。相反，您可以在运行时会通过 SELECT 语句获得集合列中的集合描述。

提示： 如果知道您要访问的集合列的准确的数据类型，请使用类型**集合**变量。

要获得集合列的描述，执行 SELECT 语句将该列检索到无类型**集合**变量。数据库服务器返回具有列数据的列（集合类型和元素类型）描述。GBase 8s ESQL/C 将集合列的定义分配给无类型**集合**变量。

例如：假定将 **a_coll** 主机变量声明为无类型**集合**变量。如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection a_coll;
        EXEC SQL END DECLARE SECTION;
```

以下代码段在 INSERT 语句中使用集合**变量**之前，使用 SELECT 语句初始化具有 **list_col** 集合列（图 1 中定义）的定义的 **a_coll** 变量。

```
EXEC SQL allocate collection :a_coll;

        /* select LIST column into the untyped collection variable
        * to obtain the data-type information */
        EXEC SQL select list_col into :a_coll from tab_list;

        /* Insert an element at the end of the LIST in the untyped
        * collection variable */
        EXEC SQL insert into table(:a_coll) values (7);
```

要获得集合列的描述，您的应用程序必须验证集合列在选择列之前它里有数据。如果表中没有行，则 SELECT 语句无法返回列数据或列描述，并且 GBase 8s ESQL/C 不会将列描述分配给无类型**集合**变量。

可以使用无类型**集合**变量存储具有不同列定义的集合，只有您在 SQL 语句中使用变量之前，将关联的集合列描述选择到**集合**变量。

重要： 在 SQL 语句中使用变量*之前*，必须获得无类型**集合**变量的集合列的定义。在**集合**变量可以保存任何值之前，必须使用 SELECT 语句从数据库中的集合列获取集合数据类型的描述。因此，您无法直接将值插入或选择到无类型**集合**变量。

客户端集合

GBase 8s ESQL/C 应用程序声明**集合**变量名称，使用 ALLOCATE COLLECTION 语句为其分配内存，然后在**集合**数据上执行操作。

要访问集合变量的元素，在 SELECT 、INSERT 、UPDATE 或 DELETE 语句的集合派生表子句章指定变量。 GBase 8s ESQL/C 执行 SELECT 、INSERT 、UPDATE 或 DELETE 操作。当在集合派生表子句中包含客户端集合变量时，GBase 8s ESQL/C 不会将这些语句发送到数据库服务器。

例如：GBase 8s ESQL/C 对 **a_multiset** 集合变量上执行以下 INSERT 操作：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection multiset(integer not null) a_multiset;
        EXEC SQL END DECLARE SECTION;
        EXEC SQL insert into table(:a_multiset) values (6);
```

当 SQL 语句包含**集合**变量时，它具有以下语法限制：

只能使用集合派生表子句和 SELECT 、INSERT 、UPDATE 或 DELETE 语句访问客户端集合的元素。

INSERT 语句的 VALUES 子句中不能具有 SELECT 、EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句。

不能包含 WHERE 子句

不能包含表达式

不能使用滚动游标

集合的内存管理

GBase 8s ESQL/C 不会对集合变量自动分配或释放内存。您必须显式管理分配给集合变量的内存。

使用以下 SQL 语句管理类型和无类型集合主机变量的内存：

ALLOCATE COLLECTION 为指定的集合变量分配内存。

集合变量可以是类型的或无类型集合的。ALLOCATE COLLECTION 语句将 **SQLCODE**（**sqlca.sqlcode**）设置为零（如果内存分配成功）和负数（如果内存分配失败）。

DEALLOCATE COLLECTION 语句为指定的集合变量释放内存。

在使用 DEALLOCATE COLLECTION 语句释放集合变量的内存后，可以重新使用集合变量。

重要： 必须显式释放分配给集合变量的内存。使用 DEALLOCATE COLLECTION 语句释放内存。

以下代码段声明 **a_set** 主机变量作为类型集合，并为此变量分配内存，然后释放此变量的内存。

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL allocate collection :a_set;
        ⋮

        EXEC SQL deallocate collection :a_set;
```

ALLOCATE COLLECTION 语句为集合变量和集合数据分配内存。

当 DEALLOCATE COLLECTION 语句失败时，是因为集合上的游标仍是打开的，会返回错误消息，该错误不会被跟踪。

集合变量上的操作

GBase 8s 支持通过 SELECT 、UPDATE 、INSERT 和 DELETE 语句访问集合列。例如：SELECT 语句可以检索集合的所有元素，UPDATE 语句可以将集合中所有的元素更改为单个值。

提示： GBase 8s 直接使用 SELECT 语句的 WHERE 子句中的 IN 谓词只能访问集合列的内容。IN 谓词只在简单集合（集合的元素是不复杂的类型）中有用。

SELECT 、INSERT 、UPDATE 和 DELETE 语句不能访问表中集合列的元素。要访问集合列中的元素，GBase 8s ESQL/C 应用程序在集合变量中建立子表，称为集合派生表。为了组成集合派生表，GBase 8s ESQL/C 应用程序访问集合变量的元素作为表的行。

本节讨论以下主题，如何在 GBase 8s ESQL/C 应用程序中使用集合派生表访问集合列：

使用 SQL 语句中的集合派生表访问集合主机变量

具有集合列的集合主机变量

插入元素到集合主机变量中

从集合主机变量中选择元素

更新集合主机变量中的元素

指定集合主机变量的元素值

删除集合主机变量中的元素

访问具有集合主机变量的嵌套集合

集合上的集合派生表子句

集合派生表子句允许您指定**集合**主机变量作为表名。

该子句具有以下语法：

```
TABLE(:coll_var)
```

在此示例中，*coll_var* 是**集合**主变量。它可以是类型或无类型**集合**主机变量，但是在它出现在集合派生表子句之前，必须声明并已在 GBase 8s ESQL/C 应用程序中分配内存。

访问集合变量

在 SQL 语句中，GBase 8s ESQL/C 应用程序在表名的位置指定集合派生表，以在**集合**主机变量上执行下列操作：

可以使用 INSERT 的 INTO 关键字或 PUT 语句后的集合派生表子句将元素插入到集合变量中。

可以使用 SELECT 语句的 FROM 子句中的集合派生表子句从集合主机变量中选择一个元素。

可以在 UPDATE 语句的 UPDATE 关键字之后使用集合派生表子句（而不是表名）更新集合主机变量中的所有或某些值。

可以在 DELETE 语句的 FROM 关键字之后，使用集合派生表子句从集合主机变量删除所有或某些元素。

提示： 如果您仅需要插入或更改具有文字值的集合列，则不必使用**集合**主机变量。相反，可以在 INSERT 语句的 INTO 子句或 UPDATE 语句的 SET 子句中显式列出文字集合值。

集合主机变量包含有效元素后，可以使用主机变量的更新集合列。

区分列和集合变量

当在 SQL 语句（如 SELECT 、INSERT 或 UPDAT）中使用集合派生表子句与集合主机变量时，该语句不会发送到数据库服务器进行处理。相反，GBase 8s ESQL/C 处理该语句。因此，数据库服务器执行的某些语法检查在包含集合派生表子句的 SQL 语句上未完成。

尤其是，GBase 8s ESQL/C 预处理程序无法区分列名和主机变量。因此，在 UPDATE 或 INSERT 语句中使用集合派生表子句时，您必须子在以下子句中使用有效的主机变量：

UPDATE 语句的 SET 子句

INSERT 语句的 VALUES 子句

初始化集合变量

您必须始终通过在其中选择集合列来初始化无类型**集合**变量。无论要对无类型**集合**变量执行什么操作，都必须执行 SELECT 语句。

重要： 将集合列选择为无类型**集合**变量，为 GBase 8s ESQL/C 提供了集合声明的描述。

可以通过将集合列选择到**集合**变量中来初始化**集合**变量，构建 SELECT 语句如下：

在选择列表中指定集合列的名称。

在 INTO 子句中指定集合主机变量。

在 FROM 子句中指定表或视图名（而不是集合派生表子句）

可以通过执行使用集合派生表语法的 INSERT 语句来初始化类型集合变量。不需要在 INSERT 或 UPDATE 前初始化类型集合变量，因为 GBase 8s ESQL/C 具有集合变量的描述。

例如：假定您使用下图中的语句创建 **tab_list** 和 **tab_set** 表：

图: 具有集合列的示例表

```
EXEC SQL create table tab_list
                (list_col list(smallint not null));
                EXEC SQL create table tab_set
                (
                id_col integer,
                set_col set(integer not null)
```

```
        );
```

以下代码段使用名为 **a_set** 的类型**集合**变量访问 **set_col** 列：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate collection :a_set;
        EXEC SQL select set_col into :a_set from tab_set
        where id_col = 1234;
```

当使用类型**集合**主机变量时，集合列的描述（集合类型和元素类型）与相应的类型**集合**主机变量的描述匹配。如果数据类型不匹配，则数据库服务器可以执行一个转换。由于 **a_set** 主机变量具有与 **set_col** 列相同的集合类型（SET）元素类型（INTEGER），索引前面的代码段中的 SELECT 语句可以成功地检索 **set_col** 列。

以下 SELECT 语句成功，因为数据库服务器将 **list_col** 列转型为 **a_set** 主机变量中的集合，并丢弃任何重复值：

```
/* This SELECT generates an error */
        EXEC SQL select list_col into :a_set from tab_list;
```

可以将任何类型的集合选择到无类型**集合**主变量中。以下代码段，使用无类型**集合**主机变量访问图 1中定义的 **list_col** 和 **set_col** 列：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection a_collection;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate collection :a_collection;
        EXEC SQL select set_col into :a_collection
        from tab_set
        where id_col = 1234;
        :

        EXEC SQL select list_col into :a_collection
        from tab_list
        where list{6} in (list_col);
```

这段代码中的两个 SELECT 语句都可以成功地将集合列检索到 **a_collection** 主机变量中。

初始化**集合**主机变量后，可以使用集合派生表子句 select 、update 或 delete 集合中的现有元素，或者向集合中插入其它元素。

插入元素到集合变量中

要向**集合**变量中插入一个或多个元素，请在 INTO 关键字之后使用具有集合派生表子句的 INSERT 语句。集合派生表子句标识要插入元素的**集合**变量。将 INSERT 语句和集合派生表子句与游标相关联，以将多个元素插入到集合变量中。

限制： 不能在 VALUES 子句中使用表达式，也不能使用 WHERE 子句。

插入一个元素

INSERT 语句和集合派生表子句允许您往集合中插入一个元素。

GBase 8s ESQL/C 将 VALUES 子句指定的值插入到集合派生表子句指定的**集合**变量中。

提示： 将元素插入到客户端**集合**变量时，不能在 INSERT 的 VALUES 子句中指定 SELECT 、EXECUTE FUNCTION 、或 EXECUTE PROCEDURE 语句。

向 SET 和 MULTISET 集合中插入元素

对于 SET 和 MULTISET 集合，新元素的位置时未定义的，因为，这些集合的元素不具有顺序位置。假定表 **readings** 具有以下声明：

```
CREATE TABLE readings
(
    dataset_id              INT8,
    time_dataset MULTISET(INT8 NOT NULL)
);
```

要访问 **time_dataset** 列， 类型 GBase 8s ESQL/C 主机变量 **time_vals** 具有以下声明：

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection multiset(int8 not null) time_vals;
    ifx_int8_t an_int8;
EXEC SQL END DECLARE SECTION;
```

以下 INSERT 语句插入新 MULTISET 元素 1,423,231 到 **time_vals** 中：

```
EXEC SQL allocate collection :time_vals;
EXEC SQL select time_dataset into :time_vals
    from readings
    where dataset_id = 1356;
ifx_int8cvint(1423231, &an_int8);
EXEC SQL insert into table(:time_vals) values (:an_int8);
```

向 LIST 集合中插入元素

LIST 集合的元素具有有序位置。如果集合是 LIST 类型，则可以使用 INSERT 语句的 AT 子句来指定要添加新元素的列表中的位置。假设表 **rankings** 具有以下声明：

```
CREATE TABLE rankings
            (
            item_id             INT8,
            item_rankings               LIST(INTEGER NOT NULL)
            );
```

要访问 **item_rankings** 列，类型 GBase 8s ESQL/C 主机变量 **rankings** 具有以下声明：

```
EXEC SQL BEGIN DECLARE SECTION;
            client collection list(integer not null) rankings;
            int an_int;
            EXEC SQL END DECLARE SECTION;
```

以下 INSERT 语句将新元素 9 添加为 **rankings** 的第三个元素：

```
EXEC SQL allocate collection :rankings;
        EXEC SQL select rank_col into :rankings from results;
        an_int = 9;
        EXEC SQL insert at 3 into table(:rankings) values (:an_int);
```

假设在插入之前，**rankings** 包含元素 {1, 8, 4, 5, 2}。那么插入之后，该变量包含元素 {1, 8, 9, 4, 5, 2}。

如果未指定 AT 子句，那么 INSERT 在 LIST 集合的末尾添加新元素。

插入多个元素

包含具有集合派生表子句的 INSERT 语句的插入游标允许您向集合变量中插入许多元素。

要插入元素，请按以下步骤操作：

在 GBase 8s ESQL/C 程序中创建一个客户端集合变量。

使用 DECLARE 语给集合变量声明插入游标，并使用 OPEN 语句打开游标。

使用 PUT 语句和 FROM 子句将一个或多个元素插入集合变量。

使用 CLOSE 语句关闭插入游标，如果不再需要游标，则使用 FREE 语句释放它。

在集合变量包含所有元素后，可以在表名上使用 UPDATE 语句或 INSERT 语句来包含集合列（SET 、MULTISET 或 LIST）中集合变量的内容。

提示： 可以使用 INSERT 语句而不是插入游标将元素一次性插入到集合变量中。但是，插入游标对于大型数据的插入更有效。

为集合变量声明插入游标

插入游标允许您在集合中插入一个或多个元素。

要为集合变量声明插入游标，请在与游标相关联的 INSERT 语句中包含集合派生表子句。集合变量的插入游标具有以下限制：

它必须是顺序游标，DECLARE 语句不能指定 SCROLL 关键字。

不能是保持游标；DECLARE 语句不能指定 WITH HOLD 游标的特性。

如果需要使用输入参数，必须准备 INSERT 语句并在 DECLARE 语句中指定准备好的语句标识符。

可以使用输入参数指定 INSERT 语句的 VALUES 子句中的值。

以下 DECLARE 语句为 **a_list** 变量声明 **list_curs** 插入游标：

```
EXEC SQL prepare ins_stmt from
        'insert into table values';
        EXEC SQL declare list_curs cursor for ins_stmt;
        EXEC SQL open list_curs using :a_list;
```

然后使用 PUT 子句指定要插入的值。有关包含此语句的代码段，请参阅图 1。

重要：无论何时在集合派生表中的集合主机变量的 PREPARE 语句中使用问号(?)，如果执行 DESCRIBE 语句，则必须在 OPEN 语句之后执行。在 OPEN 语句之前，GBase 8s ESQL/C 不知道集合行的内容。

集合派生表中集合变量的名称

以下 DECLARE 语句为 **a_list** 变量声明 **list_curs2** 插入游标：

```
EXEC SQL prepare ins_stmt2 from
                    'insert into table values';
                    EXEC SQL declare list_curs2 cursor for ins_stmt2;
                    EXEC SQL open list_curs2 using :a_list;
                    while (1)
                    {
                    EXEC SQL put list_curs2 from :an_element;

                    ⋮

                    }
```

OPEN 语句的 USING 子句指定**集合**变量的名称。然后可以使用 PUT 语句指定要插入的值。

声明插入游标之后，可以使用 OPEN 语句打开它。一旦相关联的插入游标打开，则可以将元素插入到集合变量中。

将元素放入到插入游标中

要一次性将元素放入到插入游标中，使用 PUT 语句和 FROM 子句。

PUT 语句标识与**集合**变量相关联的插入游标。FROM 子句标识要插入到游标的元素值。FROM 子句中任何主机变量的数据类型必须与集合的元素类型相符合。

要指示以后由 PUT 语句的 FROM 子句提供集合元素，请在 INSERT 语句的 VALUES 子句中使用输入参数。您可以使用 PUT 语句与静态 DECLARE 语句或 PREPARE 语句之后插入游标。以下示例在静态 DECLARE 语句之后使用 PUT 。

```
EXEC SQL DECLARE list_curs cursor FOR INSERT INTO table
            (:alist);
            EXEC SQL open list_curs;
            EXEC SQL PUT list_curs from :asmint;
```

DECLARE 语句中不能显示输入参数。

下图包含一个代码片段，演示如何将元素插入到集合变量 **a_list** 中，然后使用此新集合更改 **tab_list** 表（图 1中定义）的 **list_col** 列。

图: 将许多元素插入到集合变量中

```
EXEC SQL BEGIN DECLARE SECTION;
            client collection list(smallint not null) a_list;
            int a_smint;
```

```
EXEC SQL END DECLARE SECTION;
⋮

EXEC SQL allocate collection :a_list;

/* Step 1: declare the insert cursor on the collection variable */
EXEC SQL prepare ins_stmt from
'insert into table values';
EXEC SQL declare list_curs cursor for ins_stmt;
EXEC SQL open list_curs using :a_list;

/* Step 2: put the LIST elements into the insert cursor */
for (a_smint=0; a_smint<10; a_smint++)
{
EXEC SQL put list_curs from :a_smint;
};
/* Step 3: save the insert cursor into the collection variable
EXEC SQL close list_curs;

/* Step 4: save the collection variable into the LIST column */
EXEC SQL insert into tab_list values (:a_list);

/* Step 5: clean up */
EXEC SQL deallocate collection :a_list;
EXEC SQL free ins_stmt;
EXEC SQL free list_curs;
```

在图 1中，访问 **a_list** 变量的第一个语句是 OPEN 语句。因此，在代码中，GBase 8s ESQL/C 必须可以从变量声明中确定 **a_list** 变量的数据类型。因为 **a_list** 主机变量是类型集合变量，所以 GBase 8s ESQL/C 可以从变量声明中变量的数据类型。但是，如果在无类型**集合**变量中声明 **a_list**，则在执行 DECLARE 之前需一个 SELECT 语句来返回关联集合列的定义。

当使用 PUT 语句将其插入到游标中时，GBase 8s ESQL/C 会自动将插入游标的内容保存在**集合**变量中。

释放游标资源

CLOSE 语句显式释放分配给插入游标的资源。但是，游标 ID 仍存在，因此可以使用 OPEN 语句重新打开它。FREE 语句显式释放游标 ID。要重新使用游标，必须使用 DECLARE 语句再次声明该游标。

FLUSH 语句不会影响与集合变量相关联的插入游标。

从集合变量选择数据

使用集合派生表子句的 SELECT 语句允许您从**集合**变量中选择元素。

集合派生表子句标识要从中选择元素的**集合**变量。客户端**集合**变量（具有集合派生表子句）的 SELECT 语句具有以下限制：

SELECT 的选择列表不能包含表达式。

选择列表必须是一个星号（*）。

选择列表中的列名必须是个单个列名称。

这些列不能使用 *database@server:table.column* 语法。

以下 SELECT 子句和选项是不允许的：GROUP BY 、HAVING 、INTO TEMP 、ORDER BY 、WHERE 、WITH REOPTIMIZATION。

FROM 子句没有规定进行连接。

SELECT 语句和集合派生表子句允许您在**集合**变量上执行以下操作：

从集合选择一个元素

使用具有集合派生表子句的 SELECT 语句

从集合选择一行元素

使用具有集合派生表子句和行变量的 SELECT 语句Use the SELECT

从集合选择一个或多个元素

将 SELECT 语句和集合派生表子句与一个游标相关联，以为集合变量声明一个选择游标。

选择一个元素

SELECT 语句和集合派生表子句允许您将一个元素选择到集合中。

INTO 子句标识存储从集合变量中选择的元素值的变量。INTO 子句中的主机变量的数据类型必须与集合的元素类型兼容。

以下代码片段仅使用名为 **a_set** 的**集合**主机变量从 **set_col** 列（参见图 1）中仅选择一个元素：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        int an_element, set_size;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate collection :a_set;
        EXEC SQL select set_col, cardinality(set_col)
        into :a_set, :set_size from tab_set
        where id_col = 3;
        if (set_size == 1)
        EXEC SQL select * into :an_element from table(:a_set);
```

重要：当您确定 SELECT 语句只返回一个元素时，请使用此形式的 SELECT 语句。如果 SELECT 语句返回多个元素，则 GBase 8s ESQL/C 返回错误。如果不指定集合中的元素数或者您知道集合包含多个元素，请使用选择游标访问元素。

如果集合的元素本身是一个复杂类型（集合或行类型），则集合是一个嵌套集合。

选择一行元素

可以将集合中的一整行元素选择到行类型主机变量中。

INTO 子句标识一个行变量，用于存储从集合变量中选择的行元素。

以下代码片段从 **set_col** 列中选择一行到行类型主机变量 **a_row** 中：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(row(a integer) not null) a_set;
        row (a integer) a_row;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL select set_col into :a_set from tab1
        where id_col = 17;
        EXEC SQL select * into :a_row from table(:a_set);
```

选择多个元素

包含具有集合派生表子句的 SELECT 语句的选择游标允许您从集合变量中选择许多元素。

要选择元素，请执行这些步骤：

在 GBase 8s ESQL/C 程序中创建客户端集合变量。

使用 DECLARE 语句为集合变量声明选择游标，并使用 OPEN 语句打开此游标。

使用 FETCH 语句和 INTO 子句从集合变量中获取元素。

如果必要，对获得的数据执行更改或删除，并将已修改的集合变量保存在集合列中。

使用 CLOSE 语句关闭选择游标，如果不再需要此游标，使用 FREE 语句释放它。

为集合变量声明选择游标

要为集合变量声明选择游标，使用与游标相关联的 SELECT 语句包含集合派生表子句。此选择游标的 DECLARE 具有以下限制：

选择游标是更新游标。

DECLARE 语句不能包含指定只读游标模式的 FOR READ ONLY 子句。

选择游标必须是顺序游标。

DECLARE 语句不能指定 SCROLL 或 WITH HOLD 游标特征。

当为集合变量声明选择游标时，SELECT 语句的集合派生表子句必须包含集合变量名称。例如：以下 DECLARE 语句对集合变量声明选择游标 **a_set**：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        EXEC SQL END DECLARE SECTION;
        ⋮
```

```
EXEC SQL declare set_curs cursor for
select * from table(:a_set);
```

要从**集合**变量中选择一个或多个元素，使用带 INTO 子句的 FETCH 语句。

如果想要修改**集合**变量的元素，使用 FOR UPDATE 关键字将选择游标声明为更新游标。然后使用 DELETE 和 UPDATE 的 WHERE CURRENT OF 子句删除或更改集合的元素。

从选择游标获取元素

要一次性从**集合**变量中一次性获取元素，请使用 FETCH 语句和 INTO 子句。

FETCH 语句标识与**集合**变量相关联的选择游标。INTO 子句标识从**集合**变量中获取的元素值的主变量。INTO 子句的主机变量的数据类型必须与集合的元素类型相匹配。

下图包含一个代码片段，演示如何将 **set_col** 列（参见图 1）中的所有元素选择到名为 **a_set** 的类型**集合**主机变量，然后一次性从 **a_set** **集合**变量获取这些元素。

图: 从集合主机变量中选择许多元素

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection set(integer not null) a_set;
    int an_element, set_size;
EXEC SQL END DECLARE SECTION;
int an_int
⋮

EXEC SQL allocate collection :a_set;
EXEC SQL select set_col, cardinality(set_col)
    into :a_set from tab_set
    from tab_set where id_col = 3;

/* Step 1: declare the select cursor on the host variable */
EXEC SQL declare set_curs cursor for
    select * from table(:a_set);
EXEC SQL open set_curs;

/* Step 2: fetch the SET elements from the select cursor */
for (an_int=0; an_int<set_size; an_int++)
{
    EXEC SQL fetch set_curs into :an_element;

⋮

};
EXEC SQL close set_curs;

/* Step 3: update the SET column with the host variable */
EXEC SQL update tab_list SET set_col = :a_set
    where id_col = 3
```

```
EXEC SQL deallocate collection :a_set;
EXEC SQL free set_curs;
```

更改集合变量

在使用集合列初始化集合主机变量后，可以使用具有集合派生表子句的 UPDATE 语句更改集合中的元素。集合派生表子句标识要更改元素的集合变量。

UPDATE 语句和集合派生表子句允许您对集合变量执行以下操作：

将集合中的所有元素更改为同一值。

使用 UPDATE 语句（不带 WHERE CURRENT OF 子句）在 SET 子句中指定派生列名。

更改集合中的特定元素。

必须为集合变量声明更新游标，并使用带有 WHERE CURRENT OF 子句的 UPDATE 。

这两种形式的 UPDATE 语句都不能包含 WHERE 子句。

更改所有元素

不能在具有集合派生表子句的 UPDATE 语句上包含 WHERE 子句。因此，集合变量上的 UPDATE 语句将集合中的所有元素设置为您在 SET 子句中指定的值。不需要更新游标来更改集合中的所有元素。

例如，以下 UPDATE 语句将 **a_list** GBase 8s ESQL/C 集合变量中的所有元素更改为值 16：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection list(smallint not null) a_list;
        int an_int;
        EXEC SQL END DECLARE SECTION;
      ⋮

      EXEC SQL update table(:a_list) (list_elmt)
      set list_elmt = 16;
```

在此示例中，派生列 **list_elmt** 提供别名以标识 SET 子句中集合的元素。

更改一个元素

要更改集合中的一个特定元素，请为集合主机变量声明更新游标。

集合变量的更新游标是用 FOR UPDATE 关键字声明的选择游标。该更新游标允许您顺序划过集合中的元素并使用 UPDATE...WHERE CURRENT OF 语句更新当前的元素。

要更改元素，请执行这些步骤：

在 GBase 8s ESQL/C 程序中创建客户端集合变量。

使用 DECLARE 语句和 FOR UPDATE 子句为集合变量声明更新游标，并使用 OPEN 语句打开此游标。

缺省情况下，集合变量上的选择游标支持更新。

使用 FETCH 语句和 INTO 子句从集合变量中获取一个或多个元素。

使用 UPDATE 语句和 WHERE CURRENT OF 子句更改获取的数据。

保存集合列中已修改的集合变量。

使用 CLOSE 语句关闭选择游标，如果不再需要此游标，使用 FREE 语句释放它。

应用程序必须将更改游标放到要更改的元素上，然后使用 UPDATE...WHERE CURRENT OF 更改此值。

下图中的 GBase 8s ESQL/C 程序使用更新游标更改集合变量 **a_set** 中的一个元素，然后更改 **tab_set** 表（请参阅图 1）的 **set_col** 列。

图: 更改集合主机变量中的一个元素

```
EXEC SQL BEGIN DECLARE SECTION;
        int an_element;
        client collection set(integer not null) a_set;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate collection :a_set;
        EXEC SQL select set_col into :a_set from tab_set
        where id_col = 6;

        EXEC SQL declare set_curs cursor for
        select * from table(:a_set)
        for update;

        EXEC SQL open set_curs;
        while (SQLCODE != SQLNOTFOUND)
        {
        EXEC SQL fetch set_curs into :an_element;
        if (an_element = 4)
        {
        EXEC SQL update table(:a_set)(x)
        set x = 10
        where current of set_curs;
        break;
        }
        }

        EXEC SQL close set_curs;

        EXEC SQL update tab_set set set_col = :a_set
        where id_col = 6;
```

```
EXEC SQL deallocate collection :a_set;
EXEC SQL free set_curs;
```

指定元素值

可以指定以下任何值作为在集合变量中的元素：

文字值

可以直接为集合列指定文字值，而不是首先使用**集合**变量。

GBase 8s ESQL/C 主机变量

主机变量必须包含数据类型与集合的元素类型相符的值。

不能包含复杂表达式来直接指定值。

文字值作为元素

可以使用文字值指定集合变量的元素。文字值必须具有与集合元素类型相符的数据类型。

例如，以下 INSERT 语句向 SET(INTEGER NOT NULL) 主机变量 **a_set** 插入文字整数：

```
EXEC SQL insert into table(:a_set) values (6);
```

以下 UPDATE 语句使用派生列名（**an_element**）将**集合**变量 **a_set** 中的所有元素更改为 19：

```
EXEC SQL update table(:a_set) (an_element)
        set an_element = 19;
```

以下 INSERT 语句向名为 **a_set2** 的 LIST(CHAR(5)) 主机变量中插入带引号的字符串：

```
EXEC SQL insert into table(:a_set2) values ('abcde');
```

下列 INSERT 语句向名为 **nested_coll** 的 SET(LIST(INTEGER NOT NULL) 主机变量中插入文字集合：

```
EXEC SQL insert into table(:nested_coll)
        values (list{1,2,3});
```

提示： **集合**变量的文字集合的语法与集合列的文字集合语法不同。**集合**变量不需要带引号。

以下 UPDATE 语句将 **nested_coll** 集合变量更改为新的文字集合值：

```
EXEC SQL update table(:nested_coll) (a_list)
        set a_list = list{1,2,3};
```

提示：如果您只需要插入或更改集合列为文字值，则不需要使用**集合**主机变量。相反，可以在 INSERT 语句的 INTO 子句或 UPDATE 语句的 SET 子句中显式地列出作为文字集合的文字值。

ESQL/C 主机变量作为元素

可以使用 GBase 8s ESQL/C 主机变量指定集合变量的一个元素。

该主机变量必须使用与集合的元素类型相匹配的数据类型声明，并且必须包含与其符合的值。例如：以下 INSERT 语句使用主机变量将一个值插入到前面示例中的 **a_set** 变量中：

```
an_int = 6;
        EXEC SQL insert into table(:a_set) values (:an_int);
```

要插入多个值到集合变量，可以对每个值使用 INSERT 语句货值可以声明插入游标并使用 PUT 语句。

以下 UPDATE 语句使用主机变量将 **a_set**t 集合中的所有元素更改为值 4：

```
an_int = 4;
        EXEC SQL update table(:a_set) (an_element)
        set an_element = :an_int;
```

要将多个值更改到集合变量，可以声明更新游标并使用 UPDATE 语句的 WHERE CURRENT OF 子句。

删除集合变量中的元素

使用集合列初始化的集合主机变量后，可以使用 DELETE 语句和集合派生表子句删除集合变量中的元素。集合派生表子句标识要删除元素的集合变量。

DELETE 语句和集合派生表子句允许您对集合变量执行以下操作：

删除集合中的所有元素。

使用 DELETE 语句（不带 WHERE CURRENT OF 子句）。

删除集合中的特定元素。

必须为集合变量声明更新游标，并使用带 WHERE CURRENT OF 子句的 DELETE。

这两种形式的 DELETE 语句都不能包含 WHERE 子句。

删除所有元素

不能在具有集合派生表子句的 DELETE 语句中包含 WHERE 语句。因此，集合变量的 DELETE 语句删除集合中的所有元素。 删除集合的所有元素不需要更新游标。

例如，以下 DELETE 移除 **a_list** GBase 8s ESQL/C 集合变量中的所有元素：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection list(smallint not null) a_list;
        EXEC SQL END DECLARE SECTION;
        :

        EXEC SQL delete from table(:a_list);
```

删除一个元素

要删除集合中的一个特定元素，请对集合主机变量声明更新游标。集合变量的更新

游标是用 FOR UPDATE 关键字声明的选择游标。该更新游标允许您顺序划过集合中的元素并使用 DELETE...WHERE CURRENT OF 语句删除当前的元素。

要删除特定的元素，请按照更新特定元素的步骤执行。在这些步骤中，将 UPDATE...WHERE CURRENT OF 语句替换为 DELETE...WHERE CURRENT OF 语句。

应用程序必须将更改游标放到要删除的元素上，然后使用 DELETE...WHERE CURRENT OF 删除此值。

下图中的 GBase 8s ESQL/C 程序使用更新游标和具有 WHERE CURRENT OF 子句的 DELETE 语句删除 **tab_set** 表（请参阅图 1）的 **set_col** 列的元素。

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        int an_int, set_size;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL allocate collection :a_set;
        EXEC SQL select set_col, cardinality(set_col)
        into :a_set, :set_size
        from tab_set
        where id_col = 6;

        EXEC SQL declare set_curs cursor for
        select * from table(:a_set)
        for update;

        EXEC SQL open set_curs;
        while (i < set_size)
        {
        EXEC SQL fetch set_curs into :an_int;
        if (an_int == 4)
        {
        EXEC SQL delete from table(:a_set)
        where current of set_curs;
        break;
        }
        i++;
        }
        EXEC SQL close set_curs;
        EXEC SQL free set_curs;

        EXEC SQL update tab_set set set_col = :a_set
        where id_col = 6;

        EXEC SQL deallocate collection :a_set;
```

假定行中的 **id_col** 值为 6，则 **set_col** 列在该代码执行之前包含值 {1,8,4,5,2}。在 DELETE...WHERE CURRENT OF 语句执行之后，此集合变量包含元素 {1,8,5,2}。代码末

尾的 UPDATE 语句将更改的集合保存到数据库的 **set_col** 列中。若没有此 UPDATE 语句，则集合列将不会删除元素 4。

访问嵌套集合

GBase 8s 支持将嵌套集合作为列类型。嵌套集合是元素类型是另一个集合的**集合**列。例如：下图中的代码段创建表 **tab_setlist** ，其列是嵌套集合。

图: 具有嵌套集合的示例列

```
EXEC SQL create table tab_setlist
            ( setlist_col set(list(integer not null));
```

setlist_col 列是一个集合，每个元素是一个列表。该嵌套集合类似于具有集合元素的 Y 轴和列表元素的 X 轴的二维数组。

从嵌套集合选择值

要从嵌套集合选择值，您必须为每个级别的集合声明集合变量和选择游标。

以下代码段使用嵌套**集合**变量 **nested_coll** 和**集合**变量 **list_coll** 来选择嵌套集合列 **setlist_col** 中的最低级别元素。

```
EXEC SQL BEGIN DECLARE SECTION;
    client collection set(list(integer not null) not null) nested_coll;
    client collection list(integer not null) list_coll;
    int an_element;
EXEC SQL END DECLARE SECTION;
int num_elements = 1;
int an_int;
int keep_fetching = 1;
:

EXEC SQL allocate collection :nested_coll;
EXEC SQL allocate collection :list_coll;

/* Step 1: declare the select cursor on the SET collection variable */
EXEC SQL declare set_curs2 cursor for
    select * from table(:nested_coll);

/* Step 2: declare the select cursor on the LIST collection variable */
EXEC SQL declare list_curs2 cursor for
    select * from table(:list_coll);

/* Step 3: open the SET cursor */
EXEC SQL open set_curs2;

while (keep_fetching)
    {

/* Step 4: fetch the SET elements into the SET insert cursor */
    EXEC SQL fetch set_curs2 into :list_coll;
```

```
/* Open the LIST cursor */
    EXEC SQL open list_curs2;

/* Step 5: put the LIST elements into the LIST insert cursor */
    for (an_int=0; an_int<10; an_int++)
        {
        EXEC SQL fetch list_curs2 into :an_element;

⋮

        };
    EXEC SQL close list_curs2;
    num_elements++;

    if (done_fetching(num_elements))
        {
        EXEC SQL close set_curs2;
        keep_fetching = 0;
        }
    };
EXEC SQL free set_curs2;
EXEC SQL free list_curs2;

EXEC SQL deallocate collection :nested_coll;
EXEC SQL deallocate collection :list_coll;:
```

向嵌套集合中插入值

要将文字值插入到集合列的**集合**变量中，请为此元素类型指定文字集合。

不需要对实际的集合类型指定构建关键字。以下类型**集合**主机变量可以访问 **tab_setlist** 表的 **setlist_col** 列：

```
EXEC SQL BEGIN DECLARE SECTION;
            client collection set(list(integer not null) not null)
            nested_coll;
            EXEC SQL END DECLARE SECTION;

            EXEC SQL allocate collection nested_coll;
```

以下代码片段，将文字值插入到 **nested_coll** 集合变量中，然后更改 **setlist_col** 列（在图 1 中定义）：

```
EXEC SQL insert into table(:nested_coll)
            values (list{1,2,3,4});
            EXEC SQL insert into tab_setlist values (:nested_coll);
```

要向嵌套集合中插入非文字值，必须为集合的每一级声明**集合**变量和插入游标。例如：以下代码片段使用嵌套**集合**变量 **nested_coll**，向嵌套集合列 **setlist_col** 插入新元素：

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
    client collection set(list(integer not null) not null) nested_coll;
    client collection list(integer not null) list_coll;
    int an_element;
EXEC SQL END DECLARE SECTION;
int num_elements = 1;
int keep_adding = 1;
int an_int;
:

EXEC SQL allocate collection :nested_coll;
EXEC SQL allocate collection :list_coll;

/* Step 1: declare the insert cursor on the SET collection variable */
EXEC SQL declare set_curs cursor for
    insert into table(:nested_coll) values;

/* Step 2: declare the insert cursor on the LIST collection variable */
EXEC SQL declare list_curs cursor for
    insert into table(:list_coll) values;

/* Step 3: open the SET cursor */
EXEC SQL open set_curs;

while (keep_adding)
    {

/* Step 4: open the LIST cursor */
      SQL open list_curs;

/* Step 5: put the LIST elements into the LIST insert cursor */
    for (an_int=0; an_int<10; an_int++)
        {
        an_element = an_int * num_elements;
        EXEC SQL put list_curs from :an_element;

:

        };
    EXEC SQL close list_curs;
    num_elements++;

/* Step 6: put the SET elements into the SET insert cursor */
    EXEC SQL put set_curs from :list_coll;
    if (done_adding(num_elements)
        {
        EXEC SQL close set_curs;
        keep_adding = 0;
        }
    };
```

```
EXEC SQL free set_curs;
EXEC SQL free list_curs;

/* Step 7: insert the nested SET column with the host variable */
EXEC SQL insert into tab_setlist values (:nested_coll);

EXEC SQL deallocate collection :nested_coll;
EXEC SQL deallocate collection :list_coll;
```

集合列上的操作

集合变量存储集合的元素。但是，它与数据库列没有固定的连接。必须使用 INSERT 或 UPDATE 语句显式将集合变量的元素保存到集合列中。

可以使用 SELECT 、UPDATE 、INSERT 和 DELETE 语句访问集合列（SET 、MULTISET 或 LIST），如下所示：

SELECT 语句从集合列获取所有元素。

INSERT 语句将新集合插入到集合列中。

对表或视图名使用 INSERT 语句，在 VALUES 子句中指定**集合**变量。

UPDATE 语句使用新值更新集合列中整个集合。

对表或视图名使用 UPDATE ，在 SET 子句中指定**集合**变量。

从集合列选择

要选择集合列中的所有元素，在 SELECT 语句的选择列表中指定集合列。如果将**集合**主机变量放到 SELECT 语句的 INTO 子句中，则可以从 GBase 8s ESQL/C 应用程序访问这些元素。

插入和更改集合列

INSERT 和 UPDATE 语句支持集合列，如下所示：

要将集合的元素插入到集合列，在 INSERT 语句的 VALUES 子句中指定新的元素。

要更改集合列的整个集合，请在 UPDATE 语句的 SET 子句中指定新元素。UPDATE 语句必须指定派生列名来为此元素创建标识符。然后在 SET 子句中使用派生列名标识要分配新元素值的位置。

在 INSERT 语句的 VALUES 子句或 UPDATE 语句的 SET 子句中，元素值可以是以下任一一种格式：

GBase 8s ESQL/C 集合主机变量

文字集合变量

要为集合列表示文字值，请指定文字集合值。您创建一个文字集合值，使用 SET 、MULTISET 或 LIST 关键字引入该值，并以包含大括号的逗号分隔列表提供字段值。您可以使用引导（单引号或双引号）来包围整个文字集合值。以 INSERT 语句将集合 SET {7, 12,

59, 4} 插入到 **tab_set** 表的 **set_col** 列（图 1中定义）：

```
EXEC SQL insert into tab_set values
        (
        5, 'set{7, 12, 59, 4}'
        );
```

下图中的 UPDATE 语句重写前面 INSERT 添加到 **tab_set** 表的 SET 值。

图: 更改集合列

```
EXEC SQL update tab_set
          set set_col = ("list{1,2,3,4}")
          where id_col = 5;
```

重要： 如果省略 WHERE 子句，则 图 1 中的 UPDATE 语句将更改表 **tab_set** 中的所有行的 **set_col** 列。

如果此文字集合值中出现任何字符值，它与必须用引号括起来。此条件创建嵌套引号。例如，对于 SET(CHAR(5) 类型列 **col1**，文字值可以表示如下：

```
'SET{"abcde"}'
```

要在 GBase 8s ESQL/C 程序的 SQL 语句中指定嵌套的字符串，则必须在引号中转义每个双引号。以下 INSERT 语句形式如何对内双引号使用转义字符：

```
EXEC SQL insert into (col1) tab1
        values ('SET{\"abcde\"}');
```

当将双引号字符串嵌套到另一个双引号中时，您不需要转义最内部的引号，如以下 INSERT 语句所示：

```
EXEC SQL insert into tabx
        values (1, "set{""row(12345)""}");
```

如果集合或行类型是嵌套的，即，如果它包含另一个集合或行类型作为成员，内部集合或行不需要用引号括起。例如，对于数据类型是 LIST(ROW(a INTEGER, b SMALLINT) NOT NULL) 的列 **col2**，可以使用以下方式表达文字值：

```
'LIST{ROW(80, 3)}'
```

删除整个集合

要删除集合列中的整个集合，可以使用 UPDATE 语句将集合设置为空。

以下示例中的 UPDATE 语句有效地删除了 **tab_set** 表的 **set_col** 列中 **id_col** 等于 5 的行的整个集合。

```
EXEC SQL create table tab_set
        (
        id_col integer,
        set_col set(integer not null)
        );
        EXEC SQL update tab_set set set_col = set{}
        where id_col = 5;
```

以下示例显示了不带 WHERE 子句的同一 UPDATE 语句，将 **tab_set** 表所有行的 **set_col** 列设置为空。

```
EXEC SQL update tab_set set set_col = set{};
```

2.9.2 访问行类型

GBase 8s ESQL/C 支持具有 GBase 8s ESQL/C 行类型主机变量的 SQL 行类型。行类型是一个复杂数据类型，它包含一个或多个名为字段的成员。每个字段具有一个名称和与其相关联的数据类型。

GBase 8s 支持以下两种类型的行类型：

已命名行类型具有唯一名，以标识一组字段。

已命名行类型是行定义的模板。使用 CREATE ROW TYPE 语句创建命名行类型。然后可以使用命名行类型，如下所示：

在 CREATE TABLE 语句的列定义中，为数据库中的列分配数据类型

在 CREATE TABLE 语句的 OF TYPE 子句中，创建类型表

未命名行类型使用 ROW 构造函数来定义字段。

可以使用特定的未命名行类型作为数据库中一列的数据类型。您可以在 CREATE TABLE 语句的列定义中使用 ROW 构造函数创建未命名的行类型。

要访问具有行类型作为其数据类型的表中的列，请执行以下步骤：

声明行主机变量。

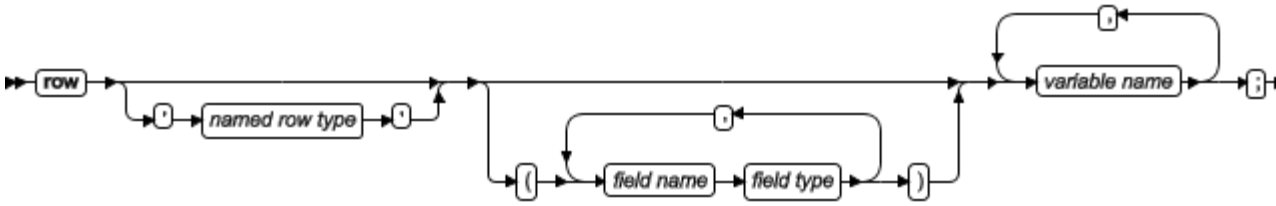
使用 ALLOCATE ROW 语句为行主机变量分配内存。

在行主机变量上执行任何 select 或 update 操作。

将行主机变量的内容保存在行类型中。

声明行变量

要声明 行主机变量，使用以下语法：



元素	意义	限制	SQL 语法
字段名称	行变量中的字段名称	必须与任何相关联的行类型列中对应的字段名称相匹配	*GBase 8s SQL 指南：语法* 中的 Identifier 段
字段类	行变量中的字段的名	可以是除 SERIAL 、SERIAL8 、BIGSERIAL 、	*GBase 8s SQL 指南：语*

型	称的数据类型	TEXT 或 BYTE 之外的任何数据类型	*法*中的数据类型段
已命名行类型	分配给**行**变量的已命名行类型的名称	已命名的行类型必须在数据库中定义	*GBase 8s SQL 指南：语法*中的 Identifier 段
变量名称	声明为**行**变量的 ESQL/C 变量的名称		名称必须符合变量名的特定语言规则

类型和无类型行变量

GBase 8s ESQL/C 支持下列两种**行**变量：

类型行变量指定行中字段的名称和数据类型。

无类型行变量不会为行指定字段名称或字段类型。

GBase 8s ESQL/C 将**行**变量当作客户端**集合**变量处理。

类型行变量

类型**行**变量指定字段列表，包含行中每个字段的名称和数据类型。

下图显示了三个类型**行**变量的声明。

图: 类型行变量示例

```
EXEC SQL BEGIN DECLARE SECTION;
        row (circle_vals circle_t, circle_id integer) mycircle;
        row (a char(20),
        b set(integer not null),
        c decimal(10,2)) row2;
        row (x integer,
        y integer,
        length integer,
        width integer) myrect;
        EXEC SQL END DECLARE SECTION;
```

类型行变量可以包含具有以下数据类型的变量：

任何除 BYTE 、TEXT 、SERIAL 或 SERIAL8 之外的内置数据类型（例如 INTEGER 、CHAR 、BOOLEAN 、FLOAT） 。

集合数据类型，例如 SET 和 LIST

行类型，已命名或未命名

不透明数据类型

当指定**行**变量中的字段的类型时，使用 SQL 数据类型，而不是 GBase 8s ESQL/C 数据类型。例如：要声明具有保存小整数的字段的**行**变量，使用 SQL SMALLINT 数据类型，而不是 GBase 8s ESQL/C **int** 数据类型。同样，要声明值为字符字符串的字段，对 CHAR 列

使用 SQL 语法，而不是 **char** 变量的 C 语法。例如：以下 **row_var** 主机变量的声明包含小整数指定和字符字段：

```
row (
        smint_fld smallint,
        char_fld char(20)
        ) row_var;
```

当知道存储在行变量中的行类型列的准确的数据类型时，使用类型行变量。将集合行变量的声明与行类型列的数据类型完全匹配。可以在诸如 INSERT 、DELETE 或 UPDATE 的 SQL 语句中直接使用此行变量。还可以在集合派生表子句中使用它。

可以在单个声明行声明多个行变量。但是，所有的变量必须具有相同的字段类型，如以下声明所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (x integer, y integer) typed_row1, typed_row2;
        EXEC SQL END DECLARE SECTION;
```

如果不知道要访问的类型列的准确数据类型，则使用无类型行变量。

无类型行变量

无类型行变量的定义仅指定行关键字和名称。下行声明了三个行变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        row row1, row2;
        row rectangle1;
        EXEC SQL END DECLARE SECTION;
```

无类型行主机变量的优点是它在行定义中提供更多的灵活性。对于无类型行变量，您在编译时不需要知道行类型列的定义。相反，在运行时，从行类型列获得行的描述。

要在运行时获取此描述，请执行一个行类型列检索到无类型行变量的 SELECT 语句。当数据库服务器执行此 SELECT 语时，它会将行类型列的数据类型信息（行中的字段类型）返回给客户端应用程序。

例如，假定 **a_row** 主机变量声明为无类型行变量，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
        row a_row;
        EXEC SQL END DECLARE SECTION;
```

以下代码段使用 SELECT 语句在使用 UPDATE 语句中的行变量之前，先使用数据类型信息初始化 **a_row** 变量。

```
EXEC SQL allocate row :a_row;

        /* obtain the data-type information */
        EXEC SQL select row_col into :a_row from tab_row;

        /* update row values in the untyped row variable */
        EXEC SQL update table(:a_row) set fld1 = 3;
```

字段名 **fld1**，引用来自 **tab_row** 表中行列的定义的 **:a_row** 的字段。

可以使用相同的无类型行变量来存储不同的行类型，但是必须将相关联的行类型列选择到每个新行类型的行变量中。

已命名行类型

已命名行类型将名称与行结构相关联。对于数据库，使用 CREATE ROW TYPE 语句创建已命名行类型。

如果数据库包含具有相同结构不同名称的多个行类型，则当数据库服务器比较命名行类型时，数据库服务器无法正确强制执行行结构等价。要解决这个歧义，在行变量的声明中指定行类型名称。

已命名 GBase 8s ESQL/C 行变量可以是类型或无类型的。

GBase 8s ESQL/C 预处理程序不会检查行类型名称的有效性，GBase 8s ESQL/C 不会在运行时使用此名称。GBase 8s ESQL/C 只是将该名称发送到数据库服务器，以提供类型解析的信息。因此，GBase 8s ESQL/C 将以下声明中的 **a_row** 变量视为无类型行变量，即使指定了行类型名称：

```
EXEC SQL BEGIN DECLARE SECTION;
        row 'address_t' a_row;
        EXEC SQL END DECLARE SECTION;
```

如果在声明（类型命名行变量）中指定行类型名称和行结构，则行类型名会覆盖此结构。例如：假设数据库包含以下定义的 **address_t** 命名行类型：

```
CREATE ROW TYPE address_t
        (
        line1       char(20),
        line2       char(20),
        city        char(20),
        state       char(2),
        zipcode     integer
        );
```

在以下声明中，**another_row** 主机变量具有 CHAR(20) 类型（来自 **address_t** 行类型）的 **line1** 和 **line2** 字段，而不是此声明指定的 CHAR(10) 。

```
EXEC SQL BEGIN DECLARE SECTION;
        row 'address_t' (line1 char(10), line2 char(10),
        city char(20), state char(2), zipcode integer) another_row;
        EXEC SQL END DECLARE SECTION;
```

集合派生表中

您不能指定已命名行类型来声明在集合派生表中使用的行变量。GBase 8s ESQL/C 不具有有关命名行类型的信息，只有数据库服务器具有。例如：假设数据库具有已命名行类型 **r1** 和表 **tab1**，它们具有以下定义：

```
CREATE ROW TYPE r1 (i integer);

            CREATE TABLE tab1
```

```
(
nt_col INTEGER,
row_col r1
);
```

要访问此列，假设您声明了两个行变量，如下所示：

```
EXEC SQL BEGIN DECLARE SECTION;
row (i integer) row1;
row (j r1) row2;
EXEC SQL END DECLARE SECTION;
```

有了此声明，以下语句成功执行，因为 GBase 8s ESQL/C 具有 **row1** 结构的信息：

```
EXEC SQL update table(:row1) set i = 31;
checksql("UPDATE Collection Derived Table 1");
```

但是，以下语句失败；因为 GBase 8s ESQL/C 不具有确定 **row2** 行变量精确的存储结构的必要信息。

```
EXEC SQL update table(:row2) set j = :row1;
checksql("UPDATE Collection Derived Table 2");
```

同样，以下语句也失败。在此情况中，GBase 8s ESQL/C 将 **r1** 视为用户定义的类型而不是已命名行类型。

```
EXEC SQL insert into tab1 values (:row2);
checksql("INSERT row variable");
```

可以使用以下这两种方法解决此限制：

在行变量声明中使用实际数据类型，如以下示例所示：

```
EXEC SQL BEGIN DECLARE SECTION;
  row (i integer) row1;
  row (j row(i integer)) row2;
  EXEC SQL END DECLARE SECTION;
```

声明无类型行变量并执行 select ，以致于 GBase 8s ESQL/C 可以从数据库服务器获取此数据类型信息。

```
EXEC SQL BEGIN DECLARE SECTION;
        row (i integer) row1;
        row row2_untyped;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL select row_col into :row2_untyped from tab1;
```

要想此方法起作用，**tab1** 表中至少有一行。

现在 UPDATE 语句在它的集合派生表子句中使用 **row2** 或 **row2_untyped** 行变量都会执行成功。

客户端行

行变量有时称为客户端行。当声明行变量时，必须声明行变量名称，分配内存和对行执行操作。

要访问行变量的元素，请在 SELECT 或 UPDATE 语句的集合派生表子句中指定变量。当其中任一语句包含集合派生表子句时，GBase 8s ESQL/C 在行变量上执行 select 或 update 操作；它包含将这些语句发送到数据库服务器执行。例如：GBase 8s ESQL/C 在行变量 **a_row** 上执行以下 UPDATE 语句指定的更新操作：

```
EXEC SQL update table(:a_row) set fld1 = 6;
```

要访问行类型的字段，必须使用具有集合派生表子句的 SELECT 或 UPDATE 语句。

管理行的内存

声明行变量后，GBase 8s ESQL/C 识别变量名。对于已命名行变量， GBase 8s ESQL/C 还会识别关联的数据类型。但是 GBase 8s ESQL/C 不会自动为行变量分配或释放内存。必须显式管理分配给行变量的内存。要管理类型和无类型行主机变量的内存，请使用以下 SQL 语句：

ALLOCATE ROW 语句为特定的行变量分配内存。

行变量可以是类型或无类型行。如果内存分配成功，则 ALLOCATE ROW 语句将 **SQLCODE**（**sqlca.sqlcode**）设置为零，如果分配失败则为负的错误代码。

DEALLOCATE ROW 语句为特定的行变量是否内存。

当使用 DEALLOCATE ROW 语句释放行变量后，可以重新使用行变量但是必须重新为它分配内存。例如，您可能使用无类型行变量存储不同的行类型。

重要： GBase 8s ESQL/C 不会隐式释放您用 ALLOCATE ROW 语句分配的内存。必须使用 DEALLOCATE ROW 语句显式执行内存释放。

以下代码段声明 **a_name** 主机变量为类型行，并为此变量分配内存，然后释放此变量的内存：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (
        fname char(15),
        mi char(2)
        lname char(15)
        ) a_name;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL allocate row :a_name;
        ⋮

        EXEC SQL deallocate row :a_name;
```

行变量的操作

SELECT 和 UPDATE 语句允许您作为一个整体访问行类型列。

GBase 8s ESQL/C 客户端应用程序可以访问以下各个字段：

只要这些操作涉及文字值，就可以使用 SQL 语句和点符号直接选择、更新或删除数

据库的行类型列中的字段。

与集合列不同，SELECT 语句可以成功访问行类型列的各个成员。因此，GBase 8s ESQL/C 客户端应用程序可以直接选择或更改数据库的行类型列中的字段。

使用行主机变量来对整个行或单个字段上的行执行操作。

限制： 不能在 SELECT 语句中使用点符号来访问**行**变量中嵌套行的字段。

使用**行**主机变量，可以*作为集合派生表*访问行类型列。集合派生表包含一列，每列都是一个字段。集合派生表允许您将行分解为其字段，然后单独访问字段。

应用程序首先通过行主机变量对字段执行操作。修改完成后，应用程序可以将行变量的内容保存到数据库的行类型列中。

本节描述有关如何在 GBase 8s ESQL/C 应用程序中使用集合派生表访问行类型列的以下主题：

如何在 SQL 语句中使用集合派生表子句来访问行主机变量

如何使用行类型列初始化行主机变量

如何从行主机变量中选择字段

如果更改行主机变量中的字段值

行类型上的集合派生表子句

集合派生表子句允许您从行类型列创建集合派生表。

本子句具有以下语法

```
TABLE(:row_var)
```

变量 *row_var* 是 **行**主机变量。它可以使用类型或无类型行主机变量，但是您必须在之前声明它。

访问行变量

可以使用集合派生表子句在**行**主机变量上执行以下操作：

可以使用 SELECT 语句的 FROM 子句中的集合派生表子句从行主机变量中选择一个或多个字段。

您可以在 UPDATE 语句中的 UPDATE 关键字之后更新行主机变量集合派生表子句中的所有或某些字段。

行变量不支持插入和删除操作。

提示： 如果只需要使用文字值插入或更新行类型列，则不需要使用行主机变量。相反，您可以在 INSERT 语句的 INTO 子句或 UPDATE 语句的 SET 子句中显式列出 literal-row 值。

当**行**主机变量包含您需要的值时，使用主机变量的内容更新行类型列。

区分列和行变量

当您使用带有 SELECT 或 UPDATE 语句的集合派生表子句时，GBase 8s ESQL/C 处理此语句。它不会将它发送到数据库服务器。因此，数据库服务器执行的某些语法检查子句在包含集合派生表子句的 SQL 语句上未完成。

特别的是，GBase 8s ESQL/C 预处理程序无法区别列名和主机变量。因此，当使用带有 UPDATE 语句的集合派生表子句更改行主机变量时，预处理程序不会检查您是否正确指定了主机变量。您必须确保使用有效的主机变量语法。

如果省略主机变量符合（冒号（:）或美元符号（$）），则预处理程序假设此名称是一个列名。例如，以下 UPDATE 语句在 SET 子句中忽略 **clob_ins** 主机变量的冒号：

```
EXEC SQL update table(:named_row1)
        set (int_fld, clob_fld, dollar_fld) =
        (10000000, clob_ins, 110.02);
```

初始化行变量

要对行类型列中的现有字段执行操作，必须首先使用字段值初始化行变量。

要执行此初始化，使用 SELECT 语句将行类型列的现有字段选择到行变量中，如下所示：

在 SELECT 语句的选择列表中指定行列名称。

在 SELECT 语句的 INTO 子句中指定行主机变量。

在 SELECT 语句的 FROM 子句中指定表或视图名，而不是集合派生表子句。

假设使用下图中的语句创建 **tab_unmrow** 和 **tab_nmrow** 表：

图: 具有行类型列的示例表

```
EXEC SQL create table tab_unmrow
        (
        area integer,
        rectangle row(
        x integer,
        y integer,
        length integer,
        width integer)
        );

        EXEC SQL create row type full_name
        (
        fname char(15),
        mi char(2),
        lname char(15)
        );
        EXEC SQL create table tab_nmrow
        (
        emp_num integer,
        emp_name full_name
```

```
        );
```

以下代码段初始化一个名为 **a_rect** 的类型行主机变量，其行列中的 **rectangle** 列的内容 **area** 列为 1234：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (
        x integer,
        y integer,
        length integer,
        width integer
        ) a_rect;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate row :a_rect;
        EXEC SQL select rectangle into :a_rect from tab_unmrow
        where area = 1234;
```

当使用类型行变量时，行类型列（字段类型）的数据类型必须与对应的类型行主机变量的数据类型相匹配。之前代码段中的 SELECT 语句成功检索 **rectangle** 列，因为 **a_rect** 主机变量具有与 **rectangle** 列相同的字段类型。

下面的 SELECT 语句失败，因为 **emp_name** 列中的字段的数据类型与 **a_rect** 主机变量不匹配：

```
/* This SELECT generates an error */
        EXEC SQL select emp_name into :a_rect from tab_nmrow;
```

您可以选择任何含管道无类型行主机变量中。以下代码片段使用无类型行主机变量来访问图 1中定义 **emp_name** 和 **rectangle** 列：

```
EXEC SQL BEGIN DECLARE SECTION;
        row an_untyped_row;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate row :an_untyped_row;
        EXEC SQL select rectangle into :an_untyped row
        from tab_unmrow
        where area = 64;
        :

        EXEC SQL select emp_name into :an_untyped_row
        from tab_nmrow
        where row{'Tashi'} in (emp_name.fname);
```

此代码段中的所有 SELECT 语句都能成功将行类型列检索到 **an_untyped_row** 主机变量中。但是，GBase 8s ESQL/C 不会对无类型行主机变量执行类型检查，因为它的元素不具有预定义的数据类型。

在初始化行主机变量后，则可以使用集合派生表子句选择或更改行中的现有字段。

插入到行变量

不能使用 INSERT 语句插入行变量。行变量代表集合派生表形式的单个表行。行类

型中的每个字段就像是虚拟表中的一个列。如果您尝试插入到行变量，则 GBase 8s ESQL/C 返回错误。

但是，可以使用 UPDATE 语句将新字段值插入到行变量中。

从行变量选择

SELECT 语句和集合派生表子句允许您在行变量中选择一个特定的字段或一组字段。

INTO 子句标识保存从行类型变量选择的字段值的主机变量。INTP 子句中的主机变量的数据类型必须与字段类型相匹配。

例如：下图中的代码段，将 **width** 字段（行变量 **myrect** 中）的值放到 **rect_width** 主机变量中。

图: 从行变量选择

```
EXEC SQL BEGIN DECLARE SECTION;
        row (x int, y int, length float, width float) myrect;
        double rect_width;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL select rect into :myrect from rectangles
        where area = 200;
        EXEC SQL select width into :rect_width
        from table(:myrect);
```

行变量上的 SELECT 语句（包含集合派生表子句）具有以下限制：

选择列表中不允许有表达式。

如果包含不透明、重复或内置数据类型的元素，则选择列表必须是星号（*）。

选择列表中的列名必须是简单列名。

这些列不能使用 *database@server:table.column* 语法。

选择列表不能使用点符号访问行的字段。

不允许以下 SELECT 子句l：GROUP BY 、HAVING 、INTO TEMP 、ORDER BY 和 WHERE。

FROM 子句没有规定进行连接。

不能在 WHERE 子句中的比较条件中指定行类型列。

如果行变量是一个嵌套行，则 SELECT 语句不是使用点符号访问内部行的字段。相反，您必须为每一个行类型声明行变量。下图中的代码段显示如何访问 **nested_row** 主机变量中的内部行字段。

图: 示例嵌套行变量

```
EXEC SQL BEGIN DECLARE SECTION;
            row (a int, b row(x int, y int)) nested_row;
```

```
        row (x int, y int) inner_row;
        integer x_var, y_var;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL select row_col into :nested_row from tab_row
        where a = 7;
        EXEC SQL select b into :inner_row
        from table(:nested_row);
        EXEC SQL select x, y into :x_var, :y_var
        from table(:inner_row);
```

以下访问 **nested_row** 变量 **x** 和 **y** 字段的 SELECT 语句是无效的，因为它使用点符号：

```
EXEC SQL select row_col into :nested_row from tab_row
        EXEC SQL select b.x, b.y       /* invalid syntax */
        into :x_var, :y_var from table(:nested_row);
```

当 SELECT 语句访问数据库列时，GBase 8s ESQL/C 应用程序可以使用点符号访问嵌套行的字段。

更新行变量

UPDATE 语句和集合派生表子句允许您更改行变量章的特定字段或一组字段。

在 SET 子句中指定新的字段值。行变量中字段的 UPDATE 不能包含 WHERE 子句。

例如：以下 UPDATE 更改 **myrect** GBase 8s ESQL/C 行变量中的 **x** 和 **y** 字段：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (x int, y int, length float, width float) myrect;
        int new_y;
        EXEC SQL END DECLARE SECTION;
        :

        new_y = 4;
        EXEC SQL update table(:myrect)
        set x=3, y=:new_y;
```

不能在 INSERT 语句的集合派生表子句中使用行变量。但是，可以使用 UPDATE 语句和集合派生表子句将新的字段值插入到行主机变量中，只要您对行中的每一个字段指定了值。例如：以下代码段将新的字段值插入到行变量 **myrect**，然后将此行变量插入到数据库中。

```
EXEC SQL update table(:myrect)
        set x=3, y=4, length=12, width=6;
        EXEC SQL insert into rectangles
        values (72, :myrect);
```

从行变量删除

删除操作不适用于行变量，因为一个删除通常从表移除一行。行变量将行类型值视为集合派生表中的单个表行。行类型中的每个字段是该表的一列。不能从集合派生表移除单个表行。因此， DELETE 语句在集合派生表子句中不支持行变量。如果尝试对行变量执行

DELETE 操作，则 GBase 8s ESQL/C 返回错误。

但是，可以使用 UPDATE 语句行变量中的现有字段值。

指定字段名称

对于行变量的字段名称，GBase 8s ESQL/C 不区分大小写。在 SELECT 或 UPDATE 语句中，GBase 8s ESQL/C 总是将行变量的字段名称解释为小写。例如，在以下 SELECT 语句中，GBase 8s ESQL/C 将字段解释为 **x** 和 **y**，尽管 SELECT 语句以大写指定它们：

```
EXEC SQL select X, Y from table(:myrect);
```

此行为与数据库服务器如何处理 SQL 语句中的标识符名称一致。要保持字符段名称的大小写，请指定字段名称作为分隔符。也就是说，在双引号之间加上字段名称，并在编译程序之前启用 DELIMIDENT 环境变量。

GBase 8s ESQL/C 将以下 SELECT 语句中的字段解释为 **X** 和 **Y**（大写）（假设启用了 DELIMIDENT 环境变量）：

```
EXEC SQL select "X", "Y" from table(:myrect);
```

主机变量字段名称

如果行列的字段名称和行变量的字段名不同，则必须指定行主机变量的字段名。例如，如果以下示例中的最后一条 SELECT 语句引用字段名 **x** 和 **y** 而不是 **a_row** 的字段名，它会生成运行错误。

```
EXEC SQL BEGIN DECLARE SECTION;
        row (a integer, b float) a_row;
        int i;
        double f;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL create table tab (row_fld(x integer, y float));
        EXEC SQL insert into tab values ('row(9, 3.34e7)');
        EXEC SQL select * into a_row from tab;
        EXEC SQL select a, b into :i, :f from table(:a_row);
```

指定字段值

可以为行变量中的字段指定以下任何值：

文字值

可以直接为行变量指定文字值，而不是首先使用行变量。

构造的行

不能直接包含复杂表达式来指定字段值。但是，构造行将表达式作为字段值提供支持。

GBase 8s ESQL/C 主机变量

文字值作为字段值

可以使用文字值来为行变量指定字段值。文字值必须具有与字段类型相匹配的数据类

型。

例如：以下 UPDATE 语句将字面整数指定为 **myrectt** 变量的 **length** 字段的字段值。

```
EXEC SQL update table(:myrect) set length = 6;
```

以下 UPDATE 语句更改 **myrect** 变量的 x 和 y 的坐标字段：

```
EXEC SQL update table(:myrect)
        set (x = 14, y = 6);
```

以下 UPDATE 语句使用带引号的字符串更改名为 **a_row2** 的 ROW(a INTEGER, b CHAR(5)) 主机变量：

```
EXEC SQL update table(:a_row2) set b = 'abcde';
```

以下 UPDATE 语句使用文字行更改 **nested_row** 主机变量（图 2中定义）：

```
EXEC SQL insert into table(:nested_row)
        values (1, row(2,3));
```

重要： 行变量的文字行的语法与行类型列的文字行的语法不同，行变量不需要带引号的字符串。

如果仅需要使用文字值插入或更改行类型列，则可以在 INSERT 语句的 INTO 子句或 UPDATE 语句的 SET 子句中列出文字值。

构建行

可以使用构建的行将表达式指定为行变量的字段值。构建的表达式必须使用行构造函数，并评估与字段的字段类型兼容的数据类型。

假定您具有以下声明的嵌套行变量：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (fld1 integer, fld2 row(x smallint, y char(5))) a_nested_row;
        EXEC SQL END DECLARE SECTION;
```

以下 UPDATE 语句使用 ROW 构造函数在 ***a_nested_row*** 变量的 ***fld2*** 字段的值中指定表达式：

```
EXEC SQL update table(:a_nested_row)
        set fld2 = row(:an_int, a_func(:a_strng));
```

ESQL/C 主机变量作为字段值

可以使用 GBase 8s ESQL/C 主机变量为行变量指定字段值。

主机变量必须使用与字段的数据类型兼容的数据类型声明，并且包含的值也必须相符合。例如：以下 UPDATE 语句使用主机变量更改 **a_row** 变量的一个值。

```
an_int = 6;
        EXEC SQL update table(:a_row) set fld1 = :an_int;
```

要向行变量中插入多个值时，可以对每个值使用 UPDATE 语句或可以在 UPDATAE 语句中指定所有的字段值。

```
one_fld = 469;
        second_fld = 'dog';
```

```
        EXEC SQL update table(:a_row)
        set fld1 = :one_fld, fld2 = :second_fld;
```

以下 UPDATE 语句的变形执行与之前 UPDATE 语句相同的任务：

```
EXEC SQL update table(:a_row) set (fld1, fld2) =
        (:one_fld, :second_fld);
```

以下 UPDATE 语句使用文字字段值和主机变量更改 **nested_row** 变量：

```
EXEC SQL update table(:nested_row)
        set b = row(7, :i);
```

访问类型表

可以使用行变量访问*类型表*的列。类型表时使用 CREATE TABLE 语句的 OF TYPE 子句创建的表。该表包含来自已命名行类型的列的信息。

假定您创建名为 **names** 的类型表，它来自图 1定义的 **full_name** 已命名行类型：

```
EXEC SQL create table names of type full_name;
```

可以使用行变量访问 **names** 类型表的行。以下代码片段将 **a_name** 声明为类型行变量，并将 **names** 表的一行选择到此行变量中：

```
EXEC SQL BEGIN DECLARE SECTION;
        row (
        fname char(15),
        mi char(2)
        lname char(15)
        ) a_name;
        char last_name[16];
        EXEC SQL END DECLARE SECTION;
        :

        EXEC SQL allocate row :a_name;
        EXEC SQL select name_row into :a_name
        from names name_row
        where lname = 'Haven'
        and fname = 'C. K.'
        and mi = 'D';
        EXEC SQL select lname into :last_name from table(:a_name);
```

最后一条 SELECT 语句访问 ***:a_name*** 行变量的 ***lname*** 字段值。

以下示例说明如何使用无类型行变量访问无类型表的行：

```
EXEC SQL BEGIN DECLARE SECTION;
        row untyped_row;
        int i;
        char s[21];
        EXEC SQL END DECLARE SECTION;

        EXEC SQL create table tab_untyped(a integer, b char(20));
        EXEC SQL insert into tab_untyped(1, "junk");
        EXEC SQL select tab_untyped into :untyped_row
```

```
        from tab_untyped;
        EXEC SQL select a, b into :i, :s from table(:untyped);
```

行类型列上的操作

行变量存储行类型的字段。但是，行变量与数据库列没有固有的连接。必须使用 INSERT 或 UPDATE 语句显式将变量的内容保存到行类型列中。

可以使用 SELECT 、UPDATE 、INSERT 和 DELETE 语句访问行类型列（已命名或未命名），如下所示：

SELECT 语句从类型列获取所有的字段或特定的字段。

INSERT 语句将新行插入到行类型列中。

UPDATE 语句使用新值更改行类型列中的整个行。

在表或视图名称上使用 UPDATE 语句，并在 Values 子句中指定行名称。

DELETE 语句从表删除包含行类型列的行，即删除此行类型列的所有字段值。

从行类型列选择

SELECT 语句允许您以以下方式访问行类型列：

选择行类型列中的所有字段

选择行类型列中的特定字段

选择整个行类型列

要选择行类型列中的所有字段，请在 SELECT 语句中的选择列表中指定行类型列。要从 GBase 8s ESQL/C 应用程序访问这些字段，在 SELECT 语句的 INTO 子句中的指定行主机变量。

选择一行列的字段

您可以使用点表示法访问行类型列中的单个字段。点符号允许您使用另一个 SQL 标识符限定 SQL 标识符。使用句点符号（.）分隔标识符。以下 SELECT 语句执行与图 1 中两个 SELECT 语句相同的任务：

```
    EXEC SQL select rect.width into :rect_width from rectangles;
```

插入或更改行类型列

INSERT 和 UPDATE 语句支持行类型列：

要往行类型列中插入新行，在 INSERT 语句的 VALUES 子句中指定新的值。

要更改整个行类型列，在 UPDATE 语句的 SET 子句中指定新的字段值。

在 INSERT 语句的 VALUES 子句中或 UPDATE 语句的 SET 子句中，该字段值可以是以下任何格式：

GBase 8s ESQL/C 行主机变量

构建行

文字行值

要为行类型列表示文字值，请指定一个文字行值。您创建一个文字行值或一个命名或未命名含行类型，使用 ROW 关键字引入该值，并以逗号分隔的列表提供括号中的字段值。您可以使用引号（双或单）来包围整个文字行值。以下 INSERT 语句将 ROW(0, 0, 4, 5) 的文字行插入到 **tab_unmrow** 表（图 1中定义）的 **rectangle** 列中；

```
EXEC SQL insert into tab_unmrow values
                (
                20, "row(0, 0, 4, 5)"
                );
```

下图中的 UPDATE 语句将覆盖先前 INSERT 添加到 **tab_unmrow** 表的 SET 值。

图: 更改行类型列

```
EXEC SQL update tab_unmrow
                   set rectangle = ("row(1, 3, 4, 5)")
                   where area = 20;
```

重要： 如果省略 WHERE 子句，之前的 UPDATE 语句更改 **tab_unmrow** 表中所有行的 **rectangle** 列。

如果任何字符值出现在此文字行中，它必须包含在引号中；此条件创建嵌套引号。例如：ROW(id INTEGER 行类型的 **row1** 列的文字 name CHAR(5) 可以是：

```
'ROW(6, "dexter")'
```

要在 GBase 8s ESQL/C 程序中的 SQL 语句中指定嵌套引号，当双引号出现在引号字符串中时，必须转义每个双引号。以下两个 INSERT 语句显示如何使用内部引号的转义字符：

```
EXEC SQL insert into (row1) tab1
                values ('ROW(6, \"dexter\")');

                EXEC SQL insert into (row2) tab1
                values ('ROW(1, \"SET{80, 81, 82, 83}\")');
```

当将双引号字符串嵌入另一个带双引号的字符串中时，您不需要转义最里面的引号：

```
EXEC SQL insert into tabx
                values (1, "row(""row(12345)"")");
```

如果行类型包含行类型或集合作为成员，则内部行不需要引号。例如：对于数据类型是 ROW(a INTEGER, b SET (INTEGER)) 的列 **col2**，文字值可以是：

```
'ROW(1, SET{80, 81, 82, 83})'
```

删除整个行类型

要删除行类型列中的所有字段，在 DELETE 语句之后的 FROM 关键字之后指定表、视图或同义词名称，使用 WHERE 子句标识您要删除的表的行。

以下的 DELETE 语句删除 **tab_unmrow** 表中包含图 1中 UPDATE 一节所保存的行类型的行：

```
EXEC SQL delete from tab_unmrow
```

```
                where area = 20;
```

2.10 不透明数据类型

这些主题介绍了如何使用 lvarchar 、fixed binary 和 var binary 数据类型从 GBase 8s ESQL/C 程序访问不透明数据类型。使用这些 GBase 8s ESQL/C 数据类型来表示不透明数据类型，它们将传输到 GBase 8s 。

这些主题中的信息仅适用于您使用 GBase 8s 作为您的数据库服务器。

2.10.1 SQL 不透明数据类型

不透明数据类型是用户定义的数据类型，可以与 GBase 8s 内置数据类型相同的方式使用。不透明数据类型允许您为数据库应用程序定义新的数据类型。

不透明数据类型是完全封装的；数据库服务器不了解不透明数据类型的内部格式。因此，数据库服务器无法对如何访问具有不透明数据类型的列进行假设。数据库开发人员定义了一个数据结构。它保存了不透明类型的信息和支持功能，告诉数据库服务器如何访问此数据结构。

可以通过以下两种方式之一从 GBase 8s ESQL/C 应用程序访问不透明数据类型的值：

在外部格式中，作为字符字符串

数据库服务器通过不透明数据类型的输入和输出支持功能，在客户端应用程序和数据库服务器之间传输外部格式。

在内部格式中，作为外部编程语言（C）中的数据结构

数据库服务器通过不透明数据类型的结束和发送支持功能，在客户端应用程序和数据库服务器之间传输内部格式。

下表显示了可用于访问不透明数据类型的 GBase 8s ESQL/C 数据类型。

GBase 8s 数据类型

ESQL/C 主机变量

不透明数据类型的外部格式

lvarchar 主机变量

不透明数据类型的内部格式

fixed binary 主机变量

var binary 主机变量

本节使用名为 circle 的不透明数据类型演示 GBase 8s ESQL/Clvarchar 和 fixed binary 主机变量如何访问不透明数据类型。此数据类型包括 x,y 坐标，表示圆的中心半径值。下图显示了 circle 数据类型的内部数据结构。

图 1. 圆不透明数据类型的内部数据结构

```
typedef struct
        {
        double          x;
        double          y;
        } point_t;

        typedef struct
        {
        point_t             center;
        double              radius;
        } circle_t;
```

下图显示了数据库中注册 circle 数据类型及其输入、输出、发送和接收支持函数的 SQL 语句。

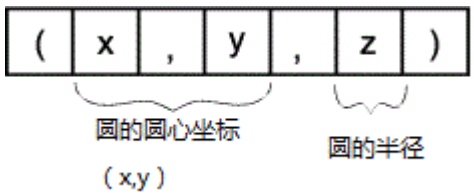
图 2. 注册 circle 不透明数据类型

```
CREATE OPAQUE TYPE circle (INTERNALLENGTH = 24,
        ALIGNMENT = 4);

        CREATE FUNCTION circle_in(c_in lvarchar) RETURNS circle
        EXTERNAL NAME '/usr/lib/circle.so(circle_input)'
        LANGUAGE C;
```

```
CREATE IMPLICIT CAST (lvarchar AS circle WITH circle_in);

CREATE FUNCTION circle_out(c_out circle) RETURNS lvarchar
EXTERNAL NAME '/usr/lib/circle.so(circle_output)'
LANGUAGE C;
CREATE IMPLICIT CAST (circle AS lvarchar WITH circle_out);

CREATE FUNCTION circle_rcv(c_rcv sendrcv) RETURNS circle
EXTERNAL NAME '/usr/lib/circle.so(circle_receive)'
LANGUAGE C;
CREATE IMPLICIT CAST (sendrcv AS circle WITH circle_rcv);

CREATE FUNCTION circle_snd(c_snd circle) RETURNS sendrcv
EXTERNAL NAME '/usr/lib/circle.so(circle_send)'
LANGUAGE C;
CREATE IMPLICIT CAST (circle AS sendrcv WITH circle_snd);

CREATE FUNCTION radius(circle) RETURNS FLOAT
EXTERNAL NAME '/usr/lib/circle.so'
LANGUAGE C;
```

假设 circle 数据类型的输入和输出函数定义了下图所示的外部格式。

图 3. circle Opaque 数据类型的外部格式



下图显示了创建和插入多行到 circle_tab 表的 SQL 语句，它具有 circle 类型的列。

图 4. 创建具有 circle 不透明数据类型的列

```
CREATE TABLE circle_tab (circle_col    circle);
        INSERT INTO circle_tab VALUES ('(12.00, 16.00, 13.00)');
        INSERT INTO circle_tab VALUES ('(6.5, 8.0, 9.0)');
```

2.10.2 访问不透明类型的外部格式

将 lvarchar 数据类型用于具有外部表示字符串的不透明类型列的操作。

要在 SQL 语句中使用不透明类型的外部格式，不透明数据类型必须具有定义的输入和输出支持函数。当客户端应用程序使用 **lvarchar** 主机变量将数据传输到不透明类型类或从不透明类型列传输数据时，数据库服务器调用以下不透明数据类型的支持函数：

输入支持函数描述如何将不透明数据类型的数据从 lvarchar 主机变量传输到不透明类型的列中。

数据库服务器调用此输入支持函数，用于将数据库服务器发送不透明类型的外部格式的操作（ INSERT 和 UPDATE 语句）。

输出支持函数描述如何将透明数据类型的数据从不透明类型的列传输到 lvarchar 主机变量中。

数据库服务器调用此输出函数，用于将诸如 SELECT 和 FETCH 语句的操作发送到客户端应用程序的不透明类型的外部格式。

重要： 如果 CREATE OPAQUE TYPE 语句指定了最大长度限制，则该值是数据库服务器存储此列的最大长度（忽略了客户端应用程序发送的数据的大小）。如果数据的长度大于此最大大小，则数据库服务器截断此数据并通知应用程序。

按照以下步骤在数据库服务器和 GBase 8s ESQL/C 应用程序之间传输不透明类型列的外部格式：

声明一个 **lvarchar** 主机变量；

在 SQL 语句中使用 **lvarchar** 主机变量在不透明类型列的外部格式上执行 select 、insert 、update 或 delete 操作。

声明 lvarchar 主机变量

使用 lvarchar 数据类型对不透明数据类型的外部格式声明主机变量。

下图描述了声明 **lvarchar** 主机变量的语法。要声明，使用 **lvarchar** 关键字作为变量数据类型，如下所示： GBase 8s 扩展lvarchar'*opaque type*',*variable name*[*variable size*]**variable name*;

元素	意义	限制	SQL 语法
opaque type	外部格式要存储到 **lvarchar** 变量中的不透明数据类型的名称	必须已经在数据库中定义	GBase 8s SQL 指南：语法中的标识符段
variable name	要声明为 **lvarchar** 变量的 GBase 8s ESQL/C 变量名		名称必须符合变量名的特定语言规则
*variable name	未指定长度的数据的 **lvarchar** 指针变量的名称	不等同于 C 字符指针（char *）。指向此类型的内部 ESQL/C 表示。必须使用 ifx_var() 函数操纵数据。	名称必须符合变量名的特定语言规则
variable size	分配给 **lvarchar** 变量的字节数	可以是 1 - 32,000 字节（32 KB）中的整数	

提示： 要对 LVARCHAR 列声明 lvarchar 主机变量，请使用 lvarchar 数据类型中显示的语法。

下图显示了四个 lvarchar 变量的声明，它们保存了不透明类型列的外部格式。

图 5. 不透明数据类型 lvarchar 主机变量示例

```
#define CIRCLESZ 20

    EXEC SQL BEGIN DECLARE SECTION;
    lvarchar 'shape' a_polygon[100];
    lvarchar 'circle' circle1[CIRCLESZ],
    circle2[CIRCLESZ];
    lvarchar 'circle' *a_crcl_ptr;
    EXEC SQL END DECLARE SECTION;
```

可以在当声明行中声明多个 lvarchar 变量。但是，所有变量必须是相同的不透明类型，如 图 1中显示的 circle1 和 circle2 的声明那样。图 1还显示了 a_crcl_ptr 主机变量的 lvarchar 指针声明。

固定大小的 lvarchar 主机变量

如果未指定 **lvarchar** 主机变量的大小，则该大小等价于一个 C 语言 **char** 数据类型。如果指定了大小，则 **lvarchar** 主机变量等价于具有此大小的 C 语言 **char** 数据类型。如果您指定固定大小的 **lvarchar** 主机变量，则当获取此列时，任何超出此指定大小的数据都会被截断。使用指示变量检查此截断。

因为固定大小的 **lvarchar** 主机变量等价于 C 语言 **char** 数据类型，所以可以使用 C 语言字符字符串操作来操纵它们。

lvarchar 指针主机变量

lvarchar 指针主机变量设计用于插入或选择可以以字符串格式表示的用户定义或不透明类型。

不透明列的字符串表示形式的大小可能因行而异。因此数据的大小是未知的，直到列被提取到主变量中为止。**lvarchar** 指针主机变量引用的数据大小最多可达 2 GB。

lvarchar 指针类型不是 C 语言 char 指针。GBase 8s ESQL/C 维护自己的 lvarchar 指针类型的内部表示。该表示与 var binary 主机变量的表示相同，不同之处在于它支持 ASCII 数据而不是二进制数据。必须使用 ifx_var() 函数操纵 lvarchar 指针主机变量。ifx_var() 函数只能用于声明为指针的 lvarchar 变量和 var binary 变量，而不能用于固定大小的 lvarchar 变量。

因为不透明类型列中的数据大小可能会从表中的第一行变化到另一行，因此您无法知道数据库服务器将返回的数据的最大大小。当您使用 **lvarchar** 指针主机变量时，可以根据来自数据库服务器的数据大小，让 GBase 8s ESQL/C 分配内存来保存数据，也可以自行分配内存。使用 ifx_var_flag() 函数指定您将要使用的函数。在这两种情况下，您必须通过使用 ifx_var_dealloc() 函数显式释放内存。

自行分配内存

要指定您正在分配内存以存储 **lvarchar** 主机变量，必须首先调用 ifx_var_flag()，给出 **lvarchar** 指针的地址，将标志值设置为零（0），如下所示：

```
ifx_var_flag(&mypoly, 0);
```

然后您必须将数据获取到 sqlda 或相同描述符结构中。之后使用 ifx_var_getllen() 函

数获取数据的长度，使用 ifx_var_alloc() 函数为此大小分配内存。

```
#include <stdio.h>
exec sql include “polyvar.h” /* includes udt - polygon_type */

main()
{
    exec sql begin declare section;
        lvarchar ‘polygon_type’ *mypoly1
        char *buffer;
        int size, p_id, len;
    exec sql end declare section;

    ifx_var_flag(&mypoly1, 0); /* specifies that appl. will allocate
memory */
    exec sql select poly into :mypoly1 from polygon_tab where p_id = 1;
    if ( ifx_var_getlen(&mypoly1) > 0 ) { /* If select returns valid data
                                                    */
    buffer = (char *)ifx_var_getdata(&mypoly1); /*Access data in
                                                        * mypoly1*/
    printf(“Length of data : %ld \n”, (int)ifx_var_getlen(&mypoly) );
    ifx_var_dealloc(&mypoly1); /* Always users responsibility to free */
}
```

不透明类型名称

此不透明类型名称是可选的；其存在影响声明如下：

当您从 lvarchar 声明中省略不透明类型时，数据库服务器将尝试识别在 lvarchar 和不透明数据类型之间转换时使用的适当的支持和转换函数。

可以使用 **lvarchar** 主机变量保存多个不同的不透明类型的数据。（只要数据库服务器可以找到合适的支持函数）

当在 lvarchar 声明中指定不透明类型时，数据库服务器会精确地知道在 lvarchar 和不透明数据类型之间转换时要使用的支持和转换声明。

使用不透明类型可以使数据转换更有效率。但是，在这种情况下，**lvarchar** 主机变量只能保存特定的不透明类型的数据。

在 **lvarchar** 主机变量的声明中，不透明类型的名称必须是带引号的字符串。

重要： 单引号（'）和双引号（"）都是 **lvarchar** 声明中的有效的引用字符串。但是，起始的引号和结束的引号必须匹配。

lvarchar 主机变量

您的 GBase 8s ESQL/C 程序必须操纵 **lvarchar** 主机变量的外部数据。如果来自不透明类型列的数据的长度不变，或者如何您指定不透明类型列中数据的最大长度，则可以使用固定大小的 **lvarchar**主机变量。但是，如果数据的大小从一个表行变成另一个表行，则使用 **lvarchar** 指针变量并使用 ifx_var() 函数操纵数据。

固定大小 lvarchar 主机变量

下图显示了如何使用固定大小的 lvarchar 主机变量向 circle_tab 表（参见图 4）中的 circle_col 列插入或选择数据。

图 6. 访问 circle 不透明数据类型的外部格式

```
EXEC SQL BEGIN DECLARE SECTION;
    lvarchar 'circle' lv_circle[30];
    char *x_coord;
EXEC SQL END DECLARE SECTION;

/* Insert a new circle_tab row with a literal opaque
 * value */
EXEC SQL insert into circle_tab
    values ('(3.00, 2.00, 11.5)');

/* Insert data into column circle of table circle_tab using an lvarchar host
 * variable */
strcpy(lv_circle, "(1.00, 17.00, 15.25)");
```

```
EXEC SQL insert into circle_tab values (:lv_circle);


/* Select column circle in circle_tab from into an lvarchar host variable
 */
EXEC SQL select circle_col into :lv_circle
    from circle_tab
    where radius(circle_col) = 15.25;
```

从固定大小的 lvarchar 主机变量插入

要将来自固定大小的 lvarchar 主机变量的数据插入到不透明类型列，请按照以下步骤操作，它们在图 1中有所描述：

定义固定大小的 **lvarchar** 主机变量。

此示例显式对 **lv_circle** 主机变量保留 30 字节。

将与不透明类型外部格式相对应的字符串放到 lvarchar 主机变量中。

当将数据放到 lvarchar 主机变量中时，必须知道不透明类型的外部格式。要使 INSERT 语句成功，lvarchar 主机变量 lv_circle 中的数据必须符合不透明数据类型的外部格式（图 3中显示）。

将 lvarchar 主机变量包含的数据插入到不透明类型列。

当数据库服务器执行 INSERT 语句时，它为 **circle** 数据类型(**circle_in**）调用输入支持函数，以将 GBase 8s ESQL/C 客户端应用程序发送的内部数据的外部格式解释为存储在磁盘上的内部格式。

图 1还显示将文字值插入到 circle_col 列的 INSERT 语句。INSERT （或 UPDATE）语句中的文字值必须还符合不透明数据类型的外部格式。

可以按照以下步骤使用固定大小的 **lvarchar** 主机变量将空值插入到不透明列中：

将 **lvarchar** 主机变量设置为空字符串。

将 **lvarchar** 主机变量的指示变量设置为 -1。

以下代码段使用 **lv_circle** 主机变量将空值插入到 **circle_col** 列：

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
lvarchar lv_circle[30];
int circle_ind;
EXEC SQL END DECLARE SECTION;
⋮

strcpy(lv_circle, "");
circle_ind = -1;
EXEC SQL insert into circle_tab
values (:lv_circle:circle_ind)l;
```

将数据选择到固定大小的 lvarchar 主机变量中

要将不透明数据类型的数据选择到固定大小的 lvarchar 主机变量，图 1 中的代码片段采用以下步骤：

将 **circle_col** 不透明类型列中包含的数据选择到 **lv_circle** 主机变量中。

当数据库服务器执行 SELECT 语句时，它为它为circle 数据类型（circle_in）调用输出支持函数，以将从磁盘检索到的内部格式翻译为 GBase 8s ESQL/C 应用程序所需的外部格式。此SELECT 语句还使用名为 radius （参见图 2）的用户定义的函数来校准来自不透明列的半径值。此函数必须在数据库服务器中注册，才能使 SELECT 语句成功执行。

访问来自 lvarchar 主机变量的 circle 数据。

SELECT 语句之后，**lv_circle** 主机变量包含的数据库是 **circle**数据类型的外部格式。

当将不透明类型列的空值选择到 **lvarchar** 主机变量中时，GBase 8s ESQL/C 将任何附加的指示变量设置为 -1。

lvarchar 指针变量

以下章节描述如何使用 **lvarchar** 指针主机变量插入和从不透明类型列中进行选择。示例使用的不透明类型列的结构表示被称为 **polygon_type**，并且具有如下定义：

```
struct {
        int no_of_edges; /* No of sides in the polygon */
        int length[100]; /* Maximum number of edges in this polygon
        is 100 */
        int center_x; /* Center x co-ordinate of the polygon */
```

```
        int center_y; /* Center y co-ordinate of the polygon */
        }
```

下行描述了此列的字符串表示法：

```
"no_of_edges, length_of_edge 1, . . . length_of_edge n, -1, center_x,
        center_y”
```

从 lvarchar 指针主变量插入

以下示例代码描述了将 **lvarchar** 指针主机变量的数据插入到不透明类型列的步骤。为了简化此示例，此代码不会检查错误的返回的值。

```
#include <stdio.h>
exec sql include “polyvar.h” /* includes udt - polygon_type */

main()
{
    exec sql begin declare section;
        lvarchar ‘polygon_type’ *mypoly1
        char *buffer;
        int size, p_id, len;
    exec sql end declare section;

    exec sql create table polygon_tab (p_id int, poly polygon_type);
    ifx_var_flag(&mypoly1, 0); /* User does allocation */
    buffer = malloc(50);

    /* String representation of mypoly1 copied into buffer*/
    strcpy(buffer, “5, 10, 20 15, 10, 5, -1, 0, 0”);
    size = strlen(buffer);
    ifx_var_alloc(&myploy1, size+1); /* Allocate memory for data in
                                        * mypoly1 */
    ifx_var_setlen(&myploy1, size); /* Set length of data bufferin
                                      * mypoly1 */
    ifx_var_setdata(&mypoly1, buffer, size); /* Store data inside mypoly1
```

```
                                                    */
    exec sql insert into polygon_tab values (1, :mypoly1);
    ifx_var_setnull(&mypoly1, 1); /* Set data buffer in mypoly1 to NULL
                                                    */
    ifx_var_dealloc(&mypoly1); /* Deallocate the data buffer in mypoly1
                                                    */
    free (buffer);
}
```

该示例代码执行以下步骤：

声明 lvarchar 指针主机变量 *mypoly1。

创建表，它包含整数 ID 列 **p_id** 和 polygon 列 **polygon_type**。

调用 ifx_var_flag() 函数指定它要为数据缓冲区分配的内存（**flag** 等于 0）。

创建缓冲区，复制 polygon 的字符串表示法，将 **size** 变量设置为缓冲区的大小。

调用 ifx_var_alloc() 、ifx_var_setlen() 和 ifx_var_setdata() 来分配数据传输缓冲区，将应用程序缓冲区的数据复制到数据传输缓冲区。

将 ID 值 1 和 **mypoly1** 插入到 **polygon_tab** 表。

将数据选择到 lvarchar 指针主机变量

以下示例代码描述将不透明类型列的数据选择到 **lvarchar** 指针主机变量中。为了简化示例。此代码不会检查错误的返回的值。

```
#include <stdio.h>
exec sql include “polyvar.h” /* includes udt - polygon_type */

main()
{
    exec sql begin declare section;
        lvarchar ‘polygon_type’ *mypoly1
        char *buffer;
        int size, p_id, len;
    exec sql end declare section;

    ifx_var_flag(&mypoly1, 1); /* ESQL/C run time will do the allocation
```

```
                                                    */

        exec sql select poly into :mypoly1 from polygon_tab where p_id = 1;
        if ( ifx_var_getlen(&mypoly1) > 0 ) { /* If select returns valid data
                                                    */
        buffer = (char *)ifx_var_getdata(&mypoly1); /*Access data in
                                                    * mypoly1*/
        printf("Length of data : %ld \n", (int)ifx_var_getlen(&mypoly) );
        printf("Data: %s \n", buffer);
        ifx_var_dealloc(&mypoly1); /* Always users responsibility to free */
    }
```

该示例代码执行以下步骤：

声明 lvarchar 指针主机变量 *mypoly1。

调用 ifx_var_flag() 函数指定它将让 GBase 8s ESQL/C 分配给数据缓冲区的内存（**flag** 等于 1)）。如果为调用 ifx_var_flag() ，则GBase 8s ESQL/C 缺省分配内存。

将列 **poly** 选择到 ***mypoly** 主机变量中。

调用 ifx_var_getdata() 获取数据缓冲区的地址，将返回的值强制转型为 **char *** 并将地址存储在 **buffer** 中。

调用 ifx_var_getlen() 显示如何获取检索到的数据的长度。

释放 GBase 8s ESQL/C 分配给 ***mypoly1** 的内存。

2.10.3 访问不透明类型的内部格式

可以使用 GBase 8s ESQL/C 主机变量以两种方式访问不透明数据类型的内部或二进制格式：

使用 fixed binary 数据类型访问固定长度的不透明数据类型，您可以为其具有表示不透明数据类型的 C 语言数据结构。

固定长度的不透明数据类型对其数据具有预定义的大小。该大小等于不透明数据类型的内部数据结构的大小。

使用 var binary 数据类型访问不同长度的不透明数据类型或访问不具有 C 语言的数据类型的固定长度不透明数据类型。

不同长度数据类型保存的数据的大小可能因行到行或实例而而异。

fixed binary 和 **var binary** 数据类型都在它们的声明和不透明数据类型的内部结构之间具有一对一的映射。当应用程序传输 **fixed binary** 或 **var binary** 主机变量中的数据时，数据库服务器调用以下不透明类型的支持函数：

接收支持函数描述如何将数据从 fixed binary 或 var binary 主机变量传输到不透明类型列。

数据库服务器对诸如将不透明类型的内部格式发送到服务器的 INSERT 和 UPDATE 语句的操作调用接收函数。

发送支持函数描述如何将不透明类型列的不透明类型数据传输到 fixed binary 或 var binary 主机变量。

数据库服务器对将不透明类型的内部格式发送到客户端应用程序的 SELECT 和 FETCH 语句调用发送支持函数。

访问固定长度不透明类型

fixed binary 数据类型允许您以它的内部格式访问固定长度的不透明数据类型。

请按照以下步骤，在数据库服务器和 GBase 8s ESQL/C 应用程序之间传输固定长度不透明类型列的内部格式：

声明 fixed binary 主机变量。

在 SQL 语句中使用 **fixed binary** 主机变量在固定长度的不透明类型列的外部格式上执行 select 、insert 、update 或 delete 操作。

声明固定二进制主机变量

使用 fixed binary 数据类型声明访问固定长度不透明数据类型的内部格式的主机变量。

要声明 **fixed binary** 主机变量，使用以下语法： GBase 8s 扩展fixed binary'*opaque type*'*structure name*,*variable name*;

元素	意义	限制	SQL 语法
opaque type	固定长度不透明数据类型的名称，其内部格式存储在	必须已经在数据库中定	GBase 8s SQL 指南：语法中

元素	意义	限制	SQL 语法
	fixed binary 变量中	义。	的标识符段
structure name	代表不透明数据类型的内部格式的 C 数据结构的名称	必须在源文件包含的头文件（.h）中定义。还必须符合数据库服务器用于表示不透明类型的内部格式的数据结构。	名称必须符合特定于变量名的语言规则。
variable name	声明为 **fixed binary** 变量 ESQL/C 变量的名称		名称必须符合特定于变量名的语言规则。

重要： **fixed binary** 主机变量仅对固定长度不透明数据类型的列有效。如果不透明数据类型是可变长度，请使用 **var binary** 主机变量。如果不知道固定长度不透明数据类型的内部格式，仍必须使用 **var binary** 主机变量访问它。

要使用 **fixed binary** 主机变量，必须引用映射不透明数据类型的内部数据结构的 C 数据结构。指定此 C 数据结构作为 **fixed binary** 声明中相同的结构名。

建议您为定义固定长度的不透明数据类型的 C 数据结构创建 C 头文件（.h 文件）。然后在每个 GBase 8s ESQL/C 源文件中包含此头文件，来使用 **fixed binary** 主机变量访问不透明数据类型。

例如，以下代码片段声明 **circle** 不透明数据类型的名为 **my_circle** 的 **fixed binary** 主机变量：

```
#include <circle.h>      /* contains definition of circle_t */

        EXEC SQL BEGIN DECLARE SECTION;
        fixed binary 'circle' circle_t my_circle;
        EXEC SQL END DECLARE SECTION;
```

在此示例中，circle.h 头文件包含 circle_t 结构（参见图 1）的声明，它是 circle 不透明数据类型的内部数据结构。my_circle 主机变量的声明指定不透明数据类型 circle 的名称和 circle_t 内部数据结构的名称。

不透明类型

当声明 **fixed binary** 主机变量时，必须指定不透明类型为引用字符串。

重要： 单引号（'）和双引号（"）都是有效的引号字符。但是，起始引号和结束引号字符必须相匹配。

不透明类型名称是可选的；它的存在对声明具有以下影响：

当在 fixed binary 声明中省略不透明类型时，数据库服务器尝试标识合适的支持函数，以在它将主机变量发送到数据库服务器来存储不透明类型列时使用。

可以使用 **fixed binary** 主机变量保存多个不同的不透明类型的数据（只要数据库服务器能够找到合适的支持函数。）

当在 fixed binary 声明中指定不透明类型时，数据库服务器精确知道使用哪种函数读取和写入不透明类型列。

使用不透明数据类型可以使数据转换更有效。但是，在这个情况中，**fixed binary** 主机变量仅保存特定的不透明数据类型的数据。

可以在单个声明中声明多个 **fixed binary** 变量。但是，所有的变量必须具有相同的不透明类型，如下所示：

```
#include <shape.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        fixed binary 'shape' shape_t square1, square2;
        EXEC SQL END DECLARE SECTION;
```

固定二进制主机变量

GBase 8s ESQL/C 程序必须处理 **fixed binary** 主机变量的内部数据结构的所有操作，它必须显式分配内存和指定字段值。

下图显示了如何使用 fixed binary 主机变量在 circle_tab 表（参见图 4）circle_col 的列中插入和选择数据。

图 7. 使用固定的二进制主机变量访问 circle 不透明数据类型的内部结构

```
/* Include declaration for circle_t structure */
```

```
#include <circle.h>;

EXEC SQL BEGIN DECLARE SECTION;
    fixed binary 'circle' circle_t fbin_circle;
EXEC SQL END DECLARE SECTION;

/* Assign data to members of internal data structure */
fbin_circle.center.x = 1.00;
fbin_circle.center.y = 17.00;
fbin_circle.radius = 15.25;

/* Insert a new circle_tab row with a fixed binary host
 * variable */
EXEC SQL insert into circle_tab values (:fbin_circle);

/* Select a circle_tab row from into a fixed binary
 * host variable */
EXEC SQL select circle_col into :fbin_circle
    from circle_tab
    where radius(circle_col) = 15.25;
if ((fbin_circle.center.x == 1.00) &&
        (fbin_circle.center.y == 17.00))
    printf("coordinates = (%d, %d)\n",
        fbin_circle.center.x,    fbin_circle.center.y);
```

从固定二进制主机变量插入

要将 **fixed binary** 主机变量包含的数据插入到不透明类型列，图 1中的代码片段执行以下步骤：

包含 **circle** 不透明数据类型的内部结构的定义。

circle_t 内部数据结构的定义（图 1显示）必须可用于您的 GBase 8s ESQL/C 程序。因此，代码段包含 **circle.h** 头文件，它包含 **circle_t** 结构的定义。

将 fixed binary 主机变量中的数据存储到内部数据结构 circle_t。

使用 **fbin_circle** 主机变量将 **fixed binary** 主变量的声明使用与 **circle_t** 内部数据结构相关联。此代码片段将值分配给 **circle_t** 数据结构的每个成员。

将 fbin_circle 主机变量包含的数据插入到 circle_col 不透明类型列中。

当数据库服务器执行此 INSERT 语句时，它调用 **circle** 数据类型（**circle_rcv**）接收支持函数，以便在 GBase 8s ESQL/C 客户端应用程序发送的数据的内部格式（**circle_t**）和磁盘上的 **circle** 数据类型的内部格式。

使用 **fixed binary** 主机变量将空值插入到不透明类型列，将指示变量设置为 -1。以下代码使用使用 **fbin circle** 主机变量将空值插入到 **circle_col** 列：

```
#include <circle.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        fixed binary 'circle' circle_t fbin_circle;
        int circle_ind;
        EXEC SQL END DECLARE SECTION;
        ⋮

        circle_ind = -1;
        EXEC SQL insert into circle_tab
        values (:fbin_circle:circle_ind);
```

将数据选择到固定二进制主机变量

要将不透明类型列包含的数据选择到 fixed binary 主机变量，图 1中的代码采用以下步骤：

将 **circle_col** 不透明列包含的数据选择到 **fbin_circle** 主机变量中。

> 当数据库服务器执行此 SELECT 语句时，它会调用 circle（circle_snd）发送支持函数，以便来在从磁盘检索到的内部格式与 GBase 8s ESQL/C 应用程序使用的内部格式之间进行必要的翻译。此 SELECT 语句还使用名为 radius（参见图 2）的用户定义函数精确知道不透明类型列的半径值。

从 fixed binary 主机变量访问 circle 数据。

SELECT 语句执行完成后，**fbin_circle** 主机变量包含 **circle** 数据类型内部格式的数据。此代码段包含来自 **circle_t** 数据结构成员的 (x,y) 坐标值。

当将不透明类型列中的空值选择到 **fixed binary** 主机变量时，GBase 8s ESQL/C 将任何附加指示变量设置为 -1。

访问不固定长度的不透明类型

var binary 数据类型允许您访问以下之一的透明数据类型的内部格式：

固定长度不透明类型列，不需要访问内部格式的 C 结构

不固定长度不透明类型列

请按照以下步骤，在数据库服务和 GBase 8s ESQL/C 应用程序之间传输这些不透明类型列的内部格式：

声明 var binary 主机变量

在 SQL 语句中使用 **var binary** 主机变量对不透明类型列的内部格式执行 select 、insert 、update 或 delete 操作。

声明 var binary 主机变量

要声明 **var binary** 主机变量，请使用以下语法。 GBase 8s 扩展var binary'*opaque type*'*structure name*,*variable name*;

元素	意义	限制	SQL 语法
opaque type	不透明数据类型的名称，其内部格式存储在 **var binary** 变量中	必须已经在数据库中定义	GBase 8s SQL 指南：语法中的标识符段
variable name	声明为 **var binary** 变量的 ESQL/C 变量名		名称必须符合特定于变量名的语言规则

下图显示了三个 var binary 变量。图 8. var binary 主机变量示例

```
#include <shape.h>;
        #include <image.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        var binary polygon1;
        var binary 'shape' polygon2, a_circle;
        var binary 'image' an_image;
        EXEC SQL END DECLARE SECTION;
```

在 **var binary** 主机变量的声明中，不透明类型的名称必须是一个引用字符串。

重要： 单引号（'）和双引号（"）都是有效的引号字符。但是，起始引号和结束引号字符必须相匹配。

不透明类型名称是可选的；它的存在对声明具有以下影响：

当在 **var binary** 声明中省略不透明类型时，当应用程序接收来自不透明列的内部数据结构时，数据库服务器尝试标识合适的支持函数。

省略不透明类型的优势是您可以使用 **var binary** 主机变量保存从多个不同不透明类型选择的数据（只要数据库服务器能找到合适的支持函数） 。

省略不透明类型的劣势是此方法声明的主机变量不能作为用户定义的例程（UDR）的参数使用。

当在 var binary 声明中指定不透明类型时，数据库服务器将内部数据结构发送到数据库服务器来存储在不透明类型列中时，它会精确显示要使用的支持函数。

不透明类型名称提供的歧义的丧失可以使数据转换更有效率。但是，在此情况中，**var binary** 主机变量可以保存特定不透明类型数据类型的数据。

可以在单个声明行中声明多个 var binary 变量。但是所有的变量必须具有相同的不透明类型，如 图 1中所示。

var binary 主机变量

在 GBase 8s ESQL/C 程序中，可变长度 C 结构 **ifx_varlena_t** 存储 **var binary** 主机变量的二进制值。此数据结构允许您在不指定此不透明数据类型的精确结构时传输二进制数据。它提供一个数据缓冲区保存与 **var binary** 主机变量相关联的数据。

重要： **ifx_varlena_t** 结构对 GBase 8s ESQL/C 程序来说是一个不透明结构。即，您不能直接访问它的内部结构。**ifx_varlena_t** 的内部结构可能在之后的版本更改。因此，要创建便捷的代码，总是对此结构使用 GBase 8s ESQL/C 访问程序函数获取并将值存储在 **ifx_varlena_t** 结构中。

本节使用可变长度的不透明数据类型 image 演示 GBase 8s ESQL/Cvar binary 主机变量如何访问不透明数据类型。此图像数据类型封装了 JPEG 、GIF 或 PPM 文件等图像。如果图像小于 2 千字节，则此数据类型的数据结构直接存储此图像。但是，如果图像大于 2 千字节，则数据结构存储包含此图像数据的智能大对象的引用（LO 指针结构）、下图显示了数据库中 image 数据类型的内部数据结构。

图 9. 图像不透明数据类型的内部数据结构

```
typedef struct
        {
        int         img_len;
        int         img_thresh;
        int         img_flags;
        union
        {
        ifx_lop_t          img_lobhandle;
        char          img_data[4];
        }

        } image_t;

        typedef struct
        {
        point_t              center;
        double               radius;
        } circle_t;
```

下图显示了 CREATE TABLE 语句，它创建了具有 image 类型的列和图像标识符的表 image_tab 。

图 10. 创建一个图像不透明数据类型的列

```
CREATE TABLE image_tab
        (
        image_id              integer not null primary key),
        image_col              image
        );
```

下图显示了如何使用 var binary 主机变量在 image_tab 表（参见图 2）image_col 列中进行插入和选择。

图 11. 使用 var binary 主机变量访问图像不透明数据类型的内部格式

```
#include <image.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        var binary 'image' vbin_image;
        EXEC SQL END DECLARE SECTION;

        struct image_t user_image, *image_ptr;
        int imgsz;

        /* Load data into members of internal data structure
        load_image(&user_image);
        imgsz = getsize(&user_image);

        /* Allocate memory for var binary data buffer */
        ifx_var_flag(&vbin_image, 0);
        ifx_var_alloc(&vbin_image, imgsz);

        /* Assign data to data buffer of var binary host
        * variable */
        ifx_var_setdata(&vbin_image, &user_image, imgsz);

        /* Insert a new image_tab row with a var binary host
        * variable */
        EXEC SQL insert into image_tab values (1, :vbin_image);

        /* Deallocate image data buffer */
        ifx_var_dealloc(&vbin_image);
```

```
/* Select an image_tab row from into a var binary
* host variable */
ifx_var_flag(&vbin_image, 1);
EXEC SQL select image_col into :vbin_image
from image_tab
where image_id = 1;
image_ptr = (image_t *)ifx_var_getdata(&vbin_image);
unload_image(&user_image);
ifx_var_dealloc(&vbin_image);
```

从 var binary 主机变量插入

将 var binary 主机变量包含的数据插入到不透明类型列，图 3中的代码采用以下步骤：

将图像数据从外部 JPEG 、GIF 或 PPM 文件加载到 **image_t** 内部数据结构中。

load_image() C 例程从外部文件加载 user_image 结构。图 1所示的 image_t 内部结构的定义必须可用于 GBase 8s ESQL/C 程序。因此，代码片段包含定义 image_t 结构的 image.h 头文件。

getsize() C 函数作为 GBase 8s ESQL/C 支持 **image** 不透明数据类型的一部分模块提供，它返回 **image_t** 结构的大小。

为 var binary 主机变量 vbin_image 的数据缓冲区分配内存。

标志值为 0 的 ifx_var_flag() 函数通知 GBase 8s ESQL/C 应用程序将为 **vbin_image** 主机变量分配内存。ifx_var_alloc() 函数然后为数据缓冲区分配图像数据所需的字节数（**imgsz**）。

将 image_t 结构存储在 vbin_image 主机变量的数据缓冲区。

ifx_var_setdata() 函数将 **user_image** 结构包含的数据保存到 **vbin_image** 数据缓冲区。此函数还要求数据缓冲区的大小，getsize() 函数返回。

将 vbin_image 数据缓冲区包含的数据插入到 image_col 不透明类型列。

当数据库服务器执行此 INSERT 语句时，它调用 **image** 数据类型的接收支持函数，以便在 GBase 8s ESQL/C 客户端应用程序发送的数据的内部格式（**image_t**）和内部格式磁盘上的 **image** 数据类型。

释放 vbin_image 主机变量的数据缓冲区。

ifx_var_dealloc() 函数释放 vbin_image 数据缓冲区。

要使用 **var binary** 主机变量将空值插入到不透明类型列中，可以使用以下两种方法：

将与 var binary 主机变量相关联的指示变量设置为 -1。

以下代码使用 **image_ind** 指示变量和 **vbin_image** 主机变量将空值插入到 **circle_col** 列：

```
#include <image.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        var binary 'image' vbin_image;
        int image_ind;
        EXEC SQL END DECLARE SECTION;

        image_ind = -1;
        EXEC SQL insert into image_tab
        values (:vbin_image:image_ind);
```

使用 ifx_var_setnull() 函数将 var binary 主机变量的数据缓冲区设置为空值。

对于相同的 **vbin_image** 主机变量，下行使用 ifx_var_setnull() 函数将空值插入到 **circle_col** 列中：

```
ifx_var_setnull(&vbin_image, 1);
                EXEC SQL insert into image_tab values (:vbin_image);
```

选择到 var binary 主机变量

将不透明类型列包含的数据选择到 var binary 主机变量中，图 3中的代码采取以下步骤：

给 **var binary** 主机变量 **vbin_image** 的数据缓冲区分配内存。

具有标志值 1 的 ifx_var_flag() 函数通知 GBase 8s ESQL/C 它给 **vbin_image** 主机变量分配新的数据缓冲区。（此数据缓冲区在 INSERT 语句完成后被释放。）当 GBase 8s ESQL/C 接收到 SELECT 语句的数据时，执行此分配。

将 image_col 不透明类型列包含的数据选择到 vbin_image 数据缓冲区。

当数据库服务器执行此 SELECT 语句时，它调用 **image** 的发送支持函数执行在磁盘上的 **image** 数据类型的内部格式和已发送（**image_t**）的 GBase 8s ESQL/C 客户端应用程序的内部格式之间的必须翻译。

将 vbin_image 主机变量包含的数据缓冲区中的数据存储到 image_t 结构中。

此 SELECT 语句之后，**vbin_image** 主机变量的数据缓冲区包含 **image** 数据类型内部格式的数据。ifx_var_getdata() 函数将此数据缓冲区的内容返回到 **image_t** 数据结构中。因为 ifx_var_getdata() 函数将数据缓冲区作为一个 "**void ***" 值返回，所以这个对 ifx_var_getdata() 的调用在将它分配给 **image_ptr** 变量之前，将此返回值作为一个指向 **image_t** 结构的指针。

将 image_t 内部数据结构的图像数据卸载为一个外部 JPEG 、GIF 或 PPM 文件。

unload_image() 例程将 **user_image** 结构卸载为一个外部文件。

释放 vbin_image 主机变量的数据缓冲区。

ifx_var_dealloc() 函数释放 **vbin_image** 数据缓冲区。必须显式释放此数据缓冲区即使是 GBase 8s ESQL/C 分配的。

要检查具有 **var binary** 主机变量的不透明类型列是否包含空值，可以使用以下两种方法：

检查与 **var binary** 主机变量的相关联的指示变量的值是否为 -1。

以下代码使用 image_ind 指示变量和 vbin_image 主机变量检查 circle_col 列的空值：

```
#include <image.h>;

        EXEC SQL BEGIN DECLARE SECTION;
        var binary 'image' vbin_image;
        int image_ind;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL select image_col into :vbin_image:image_ind
        from image_tab
        where image_id = 1;
```

```
if (image_ind == -1)
?
```

使用 ifx_var_isnull() 函数检查 **var binary** 主机变量的数据缓冲区是否为空值。

对于相同的 **vbin_image** 主机变量，以下行使用 ifx_var_isnull() 函数检查 **image_col** 列中的空值：

```
EXEC SQL select image_col into :vbin_image
        from image_tab
        where image_id = 1;
        if (ifx_var_isnull(&vbin_image) == 1)
        ?
```

2.10.4 lvarchar 指针和 var binary 库函数

下列库函数在 GBase 8s ESQL/C 中可用与范围 **lvarchar** 指针或 **var binary** 主机变量的数据缓冲区。

函数名	意义	请参阅
ifx_var_alloc()	为数据缓冲区分配内存。	ifx_var_alloc() 函数
ifx_var_dealloc()	释放数据缓冲区的内存。	ifx_var_dealloc() 函数
ifx_var_flag()	确定是 ESQL/C 还是应用程序处理数据缓冲区的内存分配。	ifx_var_flag() 函数
ifx_var_getdata()	返回数据缓冲区的内容。	ifx_var_getdata() 函数
ifx_var_getlen()	返回数据缓冲区的长度。	ifx_var_getlen() 函数
ifx_var_isnull()	检查数据缓冲区中的数据是否为空。	ifx_var_isnull() 函数
ifx_var_setdata()	设置数据缓冲区的数据。	ifx_var_setdata() 函数
ifx_var_setlen()	设置数据缓冲区的长度。	ifx_var_setlen() 函数
ifx_var_setnull()	将数据缓冲区中的数据设置为空值。	ifx_var_setnull() 函数

这些 **lvarchar** 指针和 **var binary** 还是在 sqlhdr.h 头文件中定义，因此您不需要在您的使用它们的 GBase 8s ESQL/C 程序中包含特定的头文件。

2.10.5 访问预定义的不透明数据类型

GBase 8s 将几个内置数据类型作为预定义不透明数据类型实现。这些数据类型是支持函数的不透明和数据库定义提供的数据类型。例如，智能大对象数据类型 CLOB 和 BLOB，作为不透明数据类型 **clob** 和 **blob** 实现。GBase 8s ESQL/C 使用 **ifx_lo_t** 结构（称为 LO 指针）访问智能大对象。此结构在 locator.h 头文件中定义。

因此，您将 CLOB 或 BLOB 类型的数据库列的 GBase 8s ESQL/C 主机变量声明为 **fixed binary** 主机变量，如下所示：

```
EXEC SQL include locator;
⋮

EXEC SQL BEGIN DECLARE SECTION;
    fixed binary 'clob' ifx_lo_t clob_loptr;
    fixed binary 'blob' ifx_lo_t blob_loptr;
EXEC SQL END DECLARE SECTION;
⋮

EXEC SQL select blobcol into :blob_loptr from tab1;
```

3 数据库服务器通信

3.1 异常处理

正确的数据库管理要求您了解数据库服务器是否按您的意图成功处理 SQL 语句。如果查询失败，您不知道，可能会向用户显示无意义的教程。更严重的后果可能是您更新客户账户以显示 $100 的付款，如果您不知道，则更新失败。该账户现在不正确。

要处理这些错误情景，您的 GBase 8s ESQL/C 程序必须检查每个 SQL 语句是否按照您的目的而执行。这些主题介绍了以下异常处理信息：

如何解释数据库服务器在执行 SQL 语句后呈现的诊断信息。

如何使用 SQLSTATE 变量和 GET DIAGNOSTICS 语句检查运行时错误和 GBase 8s ESQL/C 程序可能生成的警告。

如何使用 SQLCODE 变量和 SQL 通信区域（sqlca）检查运行时错误和 GBase 8s ESQL/C 程序可能生成的警告。

如何选择异常处理策略来处理 GBase 8s ESQL/C 程序中的警告和错误。

如何使用 rgetlmsg() 和 rgetmsg() 库函数检索与给出的 GBase 8s 错误号相关联的信息。

在这些主题的末尾呈现了一个名为 **getdiag** 的注释示例程序。**getdiag** 样本程序演示如何使用 **SQLSTATE** 变量和 GET DIAGNOSTICS 语句处理异常。

3.1.1 获取 SQL 语句执行后的诊断信息

当您的 GBase 8s ESQL/C 程序执行 SQL 语句之后，数据库服务器返回有关此语句的成功信息。本节总结了以下信息：

对 GBase 8s ESQL/C 程序可用的诊断信息的类型

GBase 8s ESQL/C 程序可以用于获取诊断信息的两种方法

诊断信息的种类

数据库服务器可以返回以下类型的诊断信息：

数据库异常是数据库服务器返回以描述 SQL 语句执行成功的统计。

描述性信息，例如 DESCRIBE 和 GET DIAGNOSTICS 语句可以提供某些 SQL 语句。

数据库异常的类型

当数据库服务器执行 SQL 语句时，它对应用程序返回以下四种数据库异常：

成功

SQL 语句执行成功。当语句可能将数据返回到主机变量的语句执行时，成功条件意味着语句已返回数据，并且程序可以通过主机变量访问它。

成功，但是警告生成

警告是一个不能阻止成功执行的 SQL 语句的统计。然而，声明的效果是有限的，声明可能不会产生预期的结果。警告也可以提供有关已执行语句的其它信息。

成功，但是未找到行

SQL 语句执行时没有错误，但有以下例外：

没有行符合搜索条件（NOT FOUND 条件）。

该语句没有对一行进行操作（ END OF DATA 条件）。

错误

SQL 语句执行失败，并且不会更改数据库。运行时错误可能发生在以下级别：

硬件错误，包括控制失败、磁盘坏扇区等。

内核错误，包括文件表溢出、不足的信号量等。

访问方法错误，包括重复的索引键、 插入非空列的 SQL 空等等。

解析程序错误，包括语法、未知对象、无效语句等等。

应用程序错误，包括用户或锁定表溢出等等。

描述性信息

以下 SQL 语句可以返回有关 SQL 语句的信息：

DESCRIBE 语句返回已准备的 SQL 语句的信息。此消息在您执行动态 SQL 时有用。

GET DIAGNOSTICS 语句，当在建立连接到数据库环境后调用此语句时，可以返回数据库服务器和连接的名称。

状态变量的类型

以下方法获取有关 SQL 语句结果的诊断信息：

访问 SQLSTATE 变量，一个包含符合 ANSI 和 X/Open 标准的状态值的五个字符的字符串

访问 SQLCODE 变量，该变量包含特定于 GBase 8s 的状态值的 int4 整数

当创建必须符合 ANSI 或 X/Open 标准的应用程序时，请使用 **SQLSTATE** 变量作为主异常处理方法。

3.1.2 使用 SQLSTATE 进行异常处理

建议您使用 **SQLSTATE** 变量和 GET DIAGNOSTICS 语句获取 SQL 语句的诊断信息。

重要： **SQLSTATE** 是一种比 **SQLCODE** 变量更有效的检测和处理错误消息的方法，因为 **SQLSTATE** 支持多个异常。**SQLSTATE** 也更加便捷，因为它符合 ANSI 和 X/Open 标准。GBase 8s ESQL/C 支持 **sqlca** 结构和 **SQLCODE** ，以兼容特定于 GBase 8s 的异常。

在数据库服务器执行 SQL 语句之后，它使用指示语句成功或失败的值来设置 **SQLSTATE**。从这个值，您的程序可以确定是否需要执行进一步的诊断测试。如果 **SQLSTATE** 表明有问题，您可以使用 GET DIAGNOSTICS 语句获取更多的信息。

本节介绍了如何使用 **SQLSTATE** 变量和 GET DIAGNOSTICS 语句执行异常处理，它描述了以下主题：

使用 GET DIAGNOSTICS 语句访问诊断区域的字段

了解 **SQLSTATE** 值的格式

使用 **SQLSTATE** 检查异常的不同类型

GET DIAGNOSTICS 语句

本节简单描述了如何在 GBase 8s ESQL/C 程序中使用 GET DIAGNOSTICS 语句。

GET DIAGNOSTICS 语句返回保存在诊断区域的信息。诊断区域是数据库服务器在执行 SQL 语句后进行更新的内部结构。每个应用都有一个诊断区域。虽然 GET DIAGNOSTICS 访问此诊断区域，但它不会更改此区域的内容。

要访问此诊断区域中的字段，提供一个主机变量来保存值和字段关键字以指定要访问的字段：

:*host_var* = *FIELD_NAME*

请确保主机变量的数据类型和诊断字段相符合。

诊断区域的字段分为以下两类：

语句信息描述 SQL 语句的总体结果，特别是它已修改的行数以及导致的异常数。

异常信息描述由 SQL 语句产生的各种异常。

语句信息

GET DIAGNOSTICS 语句返回最近一期执行的 SQL 语句的信息。

此格式的 GET DIAGNOSTICS 语句具有以下语法：

EXEC SQL get diagnostics *statement_fields*;

下表总结了诊断区域的 *statement_fields* 。

表 1. GET DIAGNOSTICS 语句中的语句信息

字段名称关键字	ESQL/C 数据类型	描述
NUMBER	**mint**	此字段包含诊断区域为最近执行的 SQL 语句所包含的异常数NUMBER 的范围是 1 到 35,000。即使 SQL 语句成功，此诊断区域仍会包含一个异常。
MORE	**char[2]**	此字段的值为 N 或 Y （加上一个空终止符）。N 字符表示诊断区域包含所有可用的异常信息。Y 字符表示数据库服务器已经检测到比存储在诊断区域中的异常更多的异常。现在，数据库服务器总是返回一个 N ，因为数据库服务器可用存储所有异常。
ROW_COUNT	**mint**	当此 SQL 语句是 INSERT 、UPDATE 或 DELETE 时，此字段保存一个数值，该值指定语句插入、更新或删除的行数。ROW_COUNT 的范围是 0 到 999,999,999。 有关其它 SQL 语句，ROW_COUNT 的值是未定义的。

下图显示了 GET DIAGNOSTICS 语句，它将 CREATE TABLE 语句的语句信息检索到主机变量 **:exception_count** 和 **:overflow** 中。

图: 使用 GET DIAGNOSTICS 返回语句信息

```
EXEC SQL BEGIN DECLARE SECTION;
        mint exception_count;
        char overflow[2];
        EXEC SQL END DECLARE SECTION;
        :

        EXEC SQL create database db;

        EXEC SQL create table tab1 (col1 integer);
        EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
```

使用此语句信息确定最近执行的 SQL 语句生成的异常的数量。

异常信息

GET DIAGNOSTICS 语句还返回有关最近执行的 SQL 语句生成的异常信息。每个异常都有一个异常号。要获取有关特定异常的信息，请使用 GET DIAGNOSTICS 语句的 EXCEPTION 子句，如下所示：

```
EXEC SQL get diagnostics exception except_num exception_fields;
```

except_num 参数可以是文字值或主机变量。一（1）的 *except_num* 对应于最近执行的 SQL 语句设置的 **SQLSTATE** 值。在第一个异常以后，数据库服务器用异常值填充诊断区域的顺序不是预先确定的。

下表总了的诊断区域的 *exception_fields* 信息。

表 1. GET DIAGNOSTICS 语句的异常信息

字段名称关键字	ESQL/C 数据类型	描述
RETURNED_SQLSTATE	**char[6]**	该字段包含 **SQLSTATE** 值，以描述当前异常。
GBASEDBT_SQLCODE	**int4**	此字段包含特定于 GBase 8s 的状态代码。此代码还在全局 **SQLCODE** 变量中可用。
CLASS_ORIGIN	**char[255]**	此字段包含一个可变长度的字符串。用于定义 **SQLSTATE** 类部分的源代码。如果 GBase 8s 定义了类代码，则该值为 "IX000"。如果国际标准组织（ISO）定义的此类代码，则 CLASS_ORIGIN 的值

		为 "ISO 9075"。如果用户定义例程定义了一次的消息文本，则 CLASS_ORIGIN 的值为 "U0001"。
SUBCLASS_ORIGIN	**char[255]**	此字段包含可变长度字符字符串，其中包含 **SQLSTATE** 子类部分的源。如果 ISO 定义的子类，则 SUBCLASS_ORIGIN 的值为 "ISO 9075"。如果 GBase 8s 定义了此子类，则值为 "IX000"。如果用户定义例程定义了异常的消息文本，则值为 "U0001"。
MESSAGE_TEXT	**char[8191]**	此字段包含可变长度字符串，其中包含描述异常的消息文本。该字段还可以包含任何 ISAM 异常的消息文本或来自用户定义例程的用户定义消息。
MESSAGE_LENGTH	**mint**	此字段包含 MESSAGE_TEXT 字符串的文本的字符数。
SERVER_NAME	**char[255]**	此字段包含变量长度字符串，其中包含与 CONNECT 或 DATABASE 语句的操作相关联的数据库服务器的名称。若当前不存在连接，此字段为空。
CONNECTION_NAME	**char[255]**	此字段包含变量长度字符串，其保存与 CONNECT 或 SET CONNECTION 语句的操作相关联的连接的名称。若没有当前连接或不存在显式连接。则此字段为空。否则，它包含最后成功建立连接的名称。

使用异常信息保存异常的详细信息。下表中的代码段检索 CREATE TABLE 语句的第

一个异常的异常信息。

图: 使用 GET DIAGNOSTICS 返回异常信息的示例

```
EXEC SQL BEGIN DECLARE SECTION;
        char class_origin_val[255];
        char subclass_origin_val[255];
        char message_text_val[8191];
        mint messlength_val;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL create database db;

        EXEC SQL create table tab1 (col1 integer);
        EXEC SQL get diagnostics exception 1
        :class_origin_val = CLASS_ORIGIN,
        :subclass_origin_val = SUBCLASS_ORIGIN,
        :message_text_val = MESSAGE_TEXT,
        :messlength_val = MESSAGE_LENGTH;
```

SQLSTATE 变量

SQLSTATE 变量是一个包含五个字符的字符串，在数据库服务器执行每条 SQL 语句后设置它。

GBase 8s ESQL/C 头文件 sqlca.h 将 **SQLSTATE** 声明为全局变量。由于 GBase 8s ESQL/C 预处理器自动在 GBase 8s ESQL/C 程序中包含 sqlca.h，所以您不需要声明 **SQLSTATE**。

在数据库服务器执行 SQL 语句之后，数据库服务器会自动更新 **SQLSTATE** 变量，如下所示：

数据库服务器在诊断区域的 RETURNED_SQLSTATE 字段存储异常值。

GBase 8s ESQL/C 将 RETURNED_SQLSTATE 字段的值复制到全局 SQLSTATE 变量。

这些到 **SQLSTATE** 变量的更新等价于在 SQL 语句之后立即执行以下 GET DIAGNOSTICS 语句：

```
EXEC SQL get diagnostics exception 1 :SQLSTATE = RETURNED_SQLSTATE;
```

提示： 运行时，GBase 8s ESQL/C 自动将 RETURNED_SQLSTATE 字段的值复制到全局 **SQLSTATE** 变量。因此，您不需要直接访问 RETURNED_SQLSTATE 字段。

SQLSTATE 的值是执行 GET DIAGNOSTICS 语句之前最近执行的 SQL 语句的状态。如果数据库服务器在执行 GET DIAGNOSTICS 语句时遇到错误，则将 **SQLSTATE** 设置为 "IX001"，将 **SQLCODE**（和 **sqlca.sqlcode**）设置为与错误对应的错误号的值，诊断区域的内容未定义。

SQLSTATE 变量包含异常的 ANSI 定义的值。每个 **SQLSTATE** 值具有特定于 GBase 8s 的关联状态代码。可以从以下之一的项目中获取此状态代码的值，其特定于

GBase 8s ：

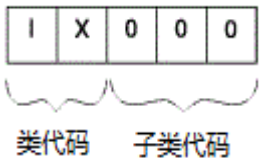
诊断区域的 GBASEDBT_SQLCODE 字段

SQLCODE 变量

类和子类代码

要确定 SQL 语句是否成功，您的 GBase 8s ESQL/C 程序必须解释 **SQLSTATE** 变量中的值。**SQLSTATE** 由两字符类代码和三字符子类代码组成。下图中，IX是子类代码，000 是子类代码。值 ″IX000″ 表示特定于 GBase 8s 的错误。

图: 具有值 IX000 的 SQLSTATE 代码的结构



SQLSTATE 只能包含数字和大写字母。类代码是唯一的，但子类代码不是。子类代码的含义取决于相关的类代码。类代码的初始字符表示异常代码的来源。 如下表所示。

表 1. 初始 SQLSTATE 类代码值

初始类代码值	异常代码的源	注意事项
0 - 4 A - H	X/Open 和 ANSI/ISO	相关的子类代码也在 0 - 4 或 A - H 的范围内开始。
5 - 9	有实施界定	子类代码也由实施界定。
I - Z	GBase 8s ESQL/C	GBase 8s 特有的任何错误消息（X/Open 或 ANSI/ISO 保留范围不支持的）的 **SQLSTATE** 值为 "IX000"。 如果用户定义例程返回的错误消息由该例程定义，则 **SQLSTATE** 值为 "U0001"。

SQLSTATE 类代码列表

下表列出了有效的 **SQLSTATE** 类和子类值。下图以粗体显示每个类代码的第一个条目。

表 1. SQLSTATE 的类和子类代码

类	子类	含义
00	000	**成功**
01	000	**带有警告的成功**
01	002	连接断开失败—事务回滚
01	003	在设置函数中消除空值
01	004	字符串数据，右截断
01	005	项目描述符区域不满足
01	006	特权未撤销
01	007	未授予特权
01	I01	**数据库具有事务**
01	I03	选择符合 ANSI 的数据库
01	I04	数据库服务器是最先版本的 GBase 8s
01	I05	使用浮点数转换为十进制数
01	I06	GBase 8s 扩展为符合 ANSI 标准语法
01	I07	在 DESCRIBE 之后，预备的 UPDATE/DELETE 语句没有 WHERE 子句
01	I08	使用 ANSI 关键字作为游标名称
01	I09	选择列表中的条目数不等于 INTO 列表中的条目数
01	I10	数据库服务器以辅助模式运行
01	I11	DATASKIP 功能打开
01	U01	用户定义的例程返回的用户定义的警告
02	000	**未找到数据或数据结束**
07	000	**动态 SQL 错误**
07	001	USING 子句不符合动态参数
07	002	USING 子句不符合目标规格
07	003	不能执行游标规范
07	004	动态参数需要 USING 子句
07	005	Prepared 语句不是游标规范
07	006	

07	008	受限数据类型属性违规
07	009	描述符计数无效
		描述符索引无效
08	000	**连接异常**
08	001	数据库服务器拒绝连接
08	002	连接名正在使用中
08	003	连接不存在
08	004	客户端无法建立连接
08	006	事务回滚
08	007	事务状态未知
08	S01	连接失败
0A	000	**不支持此功能**
0A	001	多个数据库服务器事务
21	000	**基数违规**
21	S01	插入值列表与列列表不匹配
21	S02	派生表的度不符合列表
22	000	**数据异常**
22	001	字符串数据，右截断
22	002	空值，没有指示符参数
22	003	数字值超出范围
22	005	分配中错误
22	012	除以零
22	019	转义字符无效
22	024	未终止的字符串
22	025	无效的转义序列
22	027	数据异常修整错误
23	000	**完整性约束违规**
24	000	**无效的游标状态**

25	000	**无效的事务状态**
2B	000	**依赖特权描述符依然存在**
2D	000	**无效的事务终止**
26	000	**无效的 SQL 语句标识符**
2E	000	**无效的连接名**
28	000	**无效的用户验证规范**
33	000	**无效的 SQL 描述符名称**
34	000	**无效的游标名称**
35	000	**无效的异常号**
37	000	**PREPARE 或 EXECUTE IMMEDIATE 中语法错误或访问冲突**
3C	000	**重复的游标名称**
40 40	000 003	**事务回滚** 声明未完成未知
42	000	**语法错误或访问冲突**
S0 S0 S0 S0 S0	000 001 002 011 021	**无效的名称** 基本表或视图表存在 未发现基表 索引存在 列存在
S1	001	**内存分配错误消息**
IX	000	GBase 8s **保留错误消息**
IX	001	**GET DIAGNOSTICS 语句失败**
U0	001	**由用户定义例程返回的用户定义的错误**

ANSI 或 X/Open 标准定义了除以下之外的所有 **SQLSTATE** 值：

"IX000" 运行错误表示这是一个特定于 GBase 8s 的错误消息。

"IX001" 例程称为表示 GET DIAGNOSTiCS 语句失败。

"U0001" 例程称为表示用户定义的错误消息。

"01Ixx" 警告表示特定于 GBase 8s 的警告。

"01U01" 警告表示用户定义的警告消息。

使用 SQLSTATE 检查异常

在 SQL 语句执行后，**SQLSTATE** 值可以指示下表中显示的四个条件之一。

表 1. SQLSTATE 返回的异常

异常条件	SQLSTATE 值
成功	"00000"
成功，但是没有行返回	"02000"
成功，但是警告生成	类代码 = "01" 子类代码 = "000" 到 "006" （ ANSI 和 X/Open 警告） 子类代码 = "I01" 到 "I11" （特定于 GBase 8s 的警告） 子类代码 = "U01" （用户定义的警告）
失败，运行时错误生成	类代码 > "02" （ ANSI 和 X/Open 错误） 类代码 = "IX" （特定于 GBase 8s 的警告） 类代码 = "U0" （用户定义的错误）

确定 SQLSTATE 中异常的原因

要确定 **SQLSTATE** 中异常的原因，请使用 GET DIAGNOSTICS 语句。

要确定 **SQLSTATE** 中异常的原因：

使用 GET DIAGNOSTICS 获取语句信息，如数据库服务器已经生成的异常数。

对于每个异常，使用 GET DIAGNOSTICS 的 EXCEPTION 子句获取该异常的详细信息。

SQLSTATE 中成功

当数据库服务器成功执行一个 SQL 语句时，它将 **SQLSTATE** 设置为 ″00000″（类 = ″00″，子类 = ″000″）。要检查执行是否成功，您的代码仅需要验证 **SQLSTATE** 的前两个字符。

提示： 在 CONNECT 、SET CONNECTION 、DATABASE 、CREATE DATABASE 或 START DATABASE 语句之后，**SQLSTATE** 变量具有类值 "01" 和特定于 GBase 8s 的子类值，用于提供有关数据库和连接的信息。

getdiag.ec 文件指南中的 **getdiag** 样本程序使用 sqlstate_err() 函数将 **SQLSTATE** 的前两个字符与字符串 ″00″ 做比较，以检查 SQL 语句的成功执行。图 2 中所示的 sqlstate_exception() 函数使用系统 strncmp() 函数检查 **SQLSTATE** 中的成功。

在 SQLSTATE 中 NOT FOUND

当 SELECT 或 FETCH 语句遇到 NOT FOUND （或 END OF DATA）时，数据库服务器将 **SQLSTATE** 设置为 ″02000″（类 =″02″）。下表列出了导致 SQL 语句发生 NOT FOUND 的条件。

表 1. SQL 语句未返回任何行时设置的 SQLSTATE 的值

SQL 语句生成指示的 SQLSTATE 结果	符合 ANSI 的数据库的结果	符合 ANSI 的数据库的结果
FETCH 语句：最后一个排序行已经被返回（达到数据的末尾）。	"02000"	"02000"
SELECT 语句:没有行符合 SELECT 条件。	"02000"	"02000"
DELETE 和 DELETE...WHERE 语句(不是多语句的 PREPARE 的部分）：没有行符合 DELETE 条件。	"02000"	"00000"
INSERT INTO *tablename* SELECT 语句（不是多语句的 PREPARE 的一部分）：没有行符合 SELECT 条件。	"02000"	"00000"
SELECT... INTO TEMP 语句（不是多语句的 PREPARE 的一部分）：没有行符合 SELECT 条件。	"02000"	"00000"
UPDATE 和 UPDATE...WHERE 替换（不是多语句的 PREPARE 的一部分）：没有行符合 UPDATE 条件。	"02000"	"00000"

表 1显示，在某些情况下，NOT FOUND 条件生成的值取决于数据库是否符合 ANSI 标准。

要检查 NOT FOUND 条件，您的代码只需要验证 **SQLSTATE** 的类代码。子类代码始终为 ″000″。getdiag.ec 文件指南中的 **getdiag** 示例程序使用 sqlstate_err() 函数执

行异常处理。要检查 SQL 语句中的警告，sqlstate_err() 将 **SQLSTATE** 的前两个字符与字符串 ″02″ 相比较。

SQLSTATE 中的警告

当数据库服务器成功执行 SQL 语句时，但遇到了警告条件，将 **SQLSTATE** 的类代码设置为 ″01″。子类代码指示警告的原因。此警告可以是以下类型：

ANSI 或 X/Open 警告消息具有范围为 "000" 到 "006" 的子类代码。

诊断区域的 **CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段具有值 ″ISO 9075″ 来指示 ANSI 或 X/Open 为警告的源。

GBase 8s 特定的警告消息具有范围为 "I01" 到 "I11" 的子类代码。(请参阅 表 1)。

诊断区域的 **CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段具有值 ″IX000″ 来指示特定于 GBase 8s 的异常，并将其作为警告的源。

来自用户定义例程的用户定义的消息具有子类代码 "U01"。

诊断区域的 **CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段具有值 ″U0001″ ，以指示用户定义例程作为警告的源。

下表列出了特定于 GBase 8s 的警告消息，以及 SQL 语句和警告产生的条件。

表 1. 对给出的条件设置 GBase 8s 特定的警告的 SQL 语句

警告值	SQL 语句	警告条件
"01I01"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	您的应用程序打开了一个使用事务的数据库。
"01I03"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	您的应用程序打开了一个符合 ANSI 的数据库。
"01I04"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	您的应用程序打开了 GBase 8s 管理的数据库。
"01I05"	CONNECT	您的应用程序打开了一个主

	CREATE DATABASE DATABASE SET CONNECTION	机数据库服务器上的数据库，该数据库需要 FLOAT 列的浮点转换（或 SMALLFLOAT 列的小型转换为十进制数）。.
"01I06"	所有的语句	执行语句包含 GBase 8s 对 SQL 的扩展（将当设置了 DBANSIWARN 环境变量）。
"01I07"	PREPARE DESCRIBE	预备好的不带 WHERE 子句的 UPDATE 或 DELETE 语句。该操作影响表的所有行。
"01I09"	FETCH SELECT...INTO EXECUTE...INTO	选择列表中的项数不等于 INTO 子句中的主机变量数。
"01I10"	CONNECT CREATE DATABASE DATABASE SET CONNECTION	数据库服务器当前正以辅助模式运行。数据库服务器是复制对中的辅助服务器，因此，数据库服务器可用于读取操作。
"01I11"	其它语句（当您的应用程序激活 DATASKIP 功能）	在查询处理期间跳过的数据段（dbspace）。

要检查警告，您的代码仅需要验证 **SQLSTATE** 的前两个字符。但是，要标识特定的结构，您想要检查子类代码。您可能还想使用 GET DIAGNOSTICS 语句获取 **MESSAGE_TEXT** 字段的警告消息。

例如，以下代码块确定了 CONNECT 语句打开的是哪个数据库。

```
    int trans_db, ansi_db, online_db = 0;
                                                    :

                                                    msg = "CONNECT stmt";
                                                    EXEC SQL connect to 'stores7';
                                                    if(!strncmp(SQLSTATE, "02", 2)) /* <
0 is an error */
                                                    err_chk(msg);
                                                    if (!strncmp(SQLSTATE, "01", 2))
                                                    {
                                                    if (!strncmp(SQLSTATE[2], "I01", 3))
                                                    trans_db = 1;
                                                    if (!strncmp(SQLSTATE[2], "I03", 3))
```

```
                                        ansi_db = 1;
                                        if (!strncmp(SQLSTATE[2], "I04", 3))
                                        online_db = 1;
                                        }
```

之前的代码段使用系统 strncmp() 函数检查 **SQLSTATE**。**getdiag** 示例程序（getdiag.ec 文件指南）使用 sqlstate_err() 函数通过将 **SQLSTATE** 的前两个字符与字符串 "01" 比较来检查 SQL 语句的成功。

SQLSTATE 中的运行错误

当 SQL 语句在运行时产生错误时，数据库服务器在 **SQLSTATE** 中存储一个值，其类代码大于 "02"。实际类和子类代码标识特定的错误。表 1列出了 **SQLSTATE** 的类和类代码。要检索者想错误消息文本，请使用 GET DIAGNOSTICS 语句的 **MESSAGE_TEXT** 字段。**CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段的值为 "ISO 9075"，表示错误的来源。

如果 SQL 语句生成 ANSI 或 X/Open 标准不支持的错误，则 **SQLSTATE** 可能包含以下之一值：

SQLSTATE 值 "IX000"，表示特定于 GBase 8s 的错误。

SQLCODE 变量包含错误代码，**MESSAGE_TEXT** 字段包含错误消息文本和任何 ISAM 消息文本。**CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段的值为 "IX000"，指示错误的来源。

SQLSTATE 值 "U0001"，表示来自用户定义例程的用户定义的错误消息。

MESSAGE_TEXT 字段包含错误消息文本。**CLASS_ORIGIN** 和 **SUBCLASS_ORIGIN** 异常字段的值为 "U0001"，指示错误的来源。

GET DIAGNOSTICS 失败

如果 GET Diagnostics 语句失败，则 SQLState 包含一个 ix001 值。其它的失败不会返回此错误。sqlcode 指示了导致失败的特殊错误。

多个异常

数据库服务器会对一个 SQL 语句生成多个异常。GET DIAGNOSTICS 语句的显著优点是它能够报告多个异常情况。

要了解数据库服务器为 SQL 语句报告的异常数量，请从诊断区域的语句信息中检索 **NUMBER** 字段的值。以下 GET DIAGNOSTICS 语句检索数据库服务器生成的异常数，并将数字存储在 **:exception_num** 主机变量中。

```
EXEC SQL get diagnostics :exception_num = NUMBER;
```

当您知道了发生的异常数目，可以启动一个循环来报告它们。在此循环中执行 GET DIAGNOSTICS ，并使用异常数来控制循环。以下代码说明了在 SQL 语句之后检索和报告多个异常情况的一种方法。

```
EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
        printf("NUMBER: %d\n", exception_count);
        printf("MORE : %s\n", overflow);
        for (i = 1; i <= exception_count; i++)
        {
        EXEC SQL get diagnostics   exception :i
        :sqlstate = RETURNED_SQLSTATE,
        :class = CLASS_ORIGIN, :subclass = SUBCLASS_ORIGIN,
        :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;

        printf("SQLSTATE: %s\n",sqlstate);
        printf("CLASS ORIGIN: %s\n",class);
        printf("SUBCLASS ORIGIN: %s\n",subclass);
        message[messlen] ='\0'; /* terminate the string. */
        printf("TEXT: %s\n",message);
        printf("MESSAGE LENGTH: %d\n",messlen);
        }
```

不要将 **RETURNED_SQLSTATE** 值与 **SQLSTATE** 全局变量混淆。**SQLSTATE** 变量提供最近执行的 SQL 语句的一般状态值。**RETURNED_SQLSTATE** 值与数据库服务器遇到的一个特殊异常相关联。对于第一个异常，**SQLSTATE** 和 **RETURNED_SQLSTATE** 有相同的值。但是，对于多个异常，必须访问每个异常的**RETURNED_SQLSTATE**。

要在应用程序中定义接收 **RETURNED_SQLSTATE** 值的数据变量，您必须将其定义为长度为六的字符数组（字段中为五，加上一个用于空终止符）。您可以为所需的任何名称分配此变量。

下列语句定义了一个如此的主机变量并将它命名为 **sql_state**：

```
EXEC SQL BEGIN DECLARE SECTION;
        char sql_state[6];
        EXEC SQL END DECLARE SECTION;
```

符合 X/Open 标准的数据库系统在报告任何特定于 GBase 8s 的错误或警告之前必须报告任何 X/Open 异常。但是，除此之外，数据库服务器不以任何特定顺序报告异常。**getdiag** 样本程序（getdiag.ec 文件指南）包括 disp_sqlstate_err() 函数以显示多个异常。

3.1.3 使用 sqlca 结构进行异常处理

另一种获取诊断信息的方法是通过 SQL 通信区域。当执行 SQL 语句时，数据库服务器字段返回名为 **sqlca** 的 C 结构中语句的成功或失败。

要获取异常信息，您的 GBase 8s ESQL/C 程序可以访问 **sqlca** 结构或 **SQLCODE** 变量，如下所示：

sqlca 结构。可以使用 C 结构获取附加的异常信息。还可以获取与性能相关的信息或处理的数据的性质。对于某些语句，sqlca 结构包含警告。

SQLCODE 变量。 可以获取最近执行 SQL 语句的状态代码。SQLCODE 包含特定

于 GBase 8s 的错误代码，其从 sqlca.sqlcode 字段复制。

重要： GBase 8s ESQL/C 支持之前版本的 **sqlca** 结构。但是，建议新的应用程序使用具有 GET DIAGNOSTICS 语句的 **SQLSTATE** 变量执行异常检查。该方法符合 X/Open 和 ANSI SQL 标准，并支持多个异常。

之后的三个章节介绍了如何使用 **SQLCODE** 变量和 **sqlca** 结构执行异常处理。本节包含以下主题：

了解 sqlca 结构

使用 SQLCODE 变量获取错误代码

使用 sqlca 结构检查不同种类的异常

sqlca 结构的字段

sqlca 结构在 sqlca.h 头文件中定义。GBase 8s ESQL/C 预处理器自动在 GBase 8s ESQL/C 程序中包含 sqlca.h 头文件。

下表描述了 **sqlca** 结构的字段。

表 1. sqlca 结构的字段

字段	类型	值	值描述
sqlcode	int4	0	表示成功。
		>=0, < 100	DESCRIBE 语句之后，表示所描述的 SQL 的类型
		100	成功查询之后不返回任何行，表示 NOT FOUND 条件。在 INSERT INTO/SELECT 、UPDATE 、DELETE 或 SELECT... INTO TEMP 语句无法访问任何行之后，NOT FOUND 也可能发生早符合 ANSI 的数据库中。
		<0	错误代码。
sqlerrm	character (72) 或 character (600)		使用有关 GBase 8s 数据库服务器时，此字段长度为 72 个字符，并包含错误消息参数。此参数用于替换实际错误消息中的 **%s** 令牌。如果错误消息

			不需要参数，则此字段为空。
sqlerrp	character (8)		仅限内部使用。
sqlerrd	6 int4s 的数组	[0]	在 SELECT 、UPDATE 、INSERT 或 DELETE 语句成功的 PREPARE 语句之后，或在打开选择游标之后，此字段包含估计的受影响的行数。
		[1]	当 **SQLCODE** 包含错误代码时，此字段包含零或其它错误代码，称为 ISAM 错误代码，用于说明主错误的原因。 在成功插入一行后，此字段不合适为此行生成的任何 SERIAL 值。
		[2]	成功插入、更改或删除多行后，此字段包含已处理的行数。 在以错误结束插入、更改或删除多行后，此字段包含在检测到错误之前已成功处理的行数。
		[3]	SELECT 、UPDATE 、INSERT 或 DELETE 语句的 PREPARE 语句成功之后，或者打开选择游标之后，此字段包含磁盘访问和处理的总行的估计加权和。
		[4]	PREPARE 、EXECUTE IMMEDIATE 、DECLARE 或静态 SQL 语句中出现错误之后，此字段包含检测到错误的语句文本中的偏移量。
		[5]	在成功获取选择的行后，或 insert 、update 或 delete 操

			作成功后，此字段包含处理的最后一行的 rowid（物理地址）。该 rowid 值是否对应于数据库服务器返回给用户的行，取决于数据库服务器如何处理查询，特别是 SELECT 语句。

表 2. 当打开数据库时 sqlca 结构的字段

字段	类型	值	值描述
sqlwarn	8 字符的数组	sqlwarn0	当其它警告字段设置为 W 时，此字段设置为 W 。如果为空；不需要检查其它字段。
		sqlwarn1	当现在打开的数据库使用事务日志时，设置为 W。
		sqlwarn2	当现在打开的数据库是符合 ANSI 的时，设置为 W 。
		sqlwarn3	设置为 W。
		sqlwarn4	当数据库以 DECIMAL 格式存储 FLOAT 数据类型时，则设置为 W（当主机系统缺少对 FLOAT 数据类型的支持时）。
		sqlwarn5	保留。
		sqlwarn6	当应用程序连接到以辅助模式运行的数据库服务器时，设置为 。数据库服务器是数据复制对中的辅助服务器（数据库服务器仅对读操作可用）。
		sqlwarn7	Set to W 当客户端 **DB_LOCALE** 与数据库语言环境不匹配时，设置为 W。

表 3. 所有其它操作的 sqlca 结构的字段：

字段	类型	值	值描述
sqlwarn	8 字符的数组	sqlwarn0	当其它警告字段设置为 W 时，设置为 W 。如果为空，则不需检查 **sqlwarn** 中的其它字段。
		sqlwarn1	如果使用在 FETCH 或 SELECT...INTO 语句将列值获取到主机变量时，列值被截断，则设置为 W。在 REVOKE ALL 语句中，，当所有七个表级别特权都未被撤销时，设置为 W。
		sqlwarn2	当 FETCH 或 SELECT 语句返回聚集函数（SUM 、AVG 、MIN 、MAX）的值的为空时，设置为 W。
		sqlwarn3	在 SELECT...INTO 、FETCH...INTO 或 EXECUTE...INTO 语句中，当选择列表中的项数与接收它们的 INTO 子句中的给出的主机变量的数不一致时，设置为 W 。在 GRANT ALL 语句中，当所有七个表级别特权都未被撤销时，设置为 W。
		sqlwarn4	如果 DESCRIBE 语句之后的预备的语句包含不带 WHERE 子句的 DELETE 语句或 UPDATE 语句，则设置为 W。
		sqlwarn5	在执行不使用 ANSI 标准 SQL 语法的语句（设置了 DBANSIWARN 环境变量）后设置为 W。
		sqlwarn6	在查询处理期间（DATASKIP 功能启用）跳过数据段（dbspace）时，设置为 W 。

		sqlwarn7	保留。

SQLCODE 变量

SQLCODE 变量是 **int4**，指示 SQL 语句成功还是失败。

GBase 8s ESQL/C 头文件 sqlca.h 将 **SQLCODE** 声明为全局变量。因为 GBase 8s ESQL/C 预处理器在 GBase 8s ESQL/C 程序中自动包含 sqlca.h，所以您不需要声明 **SQLCODE**。

当数据库服务器执行 SQL 语句时，数据库服务器字段更新 **SQLCODE** 变量，如下所示：

数据库服务器在 **sqlca** 结构的 **sqlcode** 字段中存储异常值。

GBase 8s ESQL/C 将 **sqlca.sqlcode** 的值复制到全局 **SQLCODE** 变量。

提示： 为了可读性和简洁性。请在 GBase 8s ESQL/C 程序中使用 **SQLCODE** 代替 **sqlca.sqlcode**。

SQLCODE 值可以表示以下类型的异常：

SQLCODE = 0

成功

SQLCODE = 100

NOT FOUND 条件

SQLCODE < 0

运行错误

纯 C 模块中的 SQLCODE

要返回与 GBase 8s ESQL/C 模块中返回的 **SQLCODE** 状态变量相同的值，可以在链接到 GBase 8s ESQL/C 程序的纯 C 模块（具有 .c 扩展名的模块）中使用 **SQLCODE**。要在纯 C 模块中使用 **SQLCODE**，请将 **SQLCODE** 声明为外部变量，如下所示：

```
extern int4 SQLCODE;
```

SQLCODE 和 exit() 调用

要将错误代码返回给父进程，不要尝试使用 **SQLCODE** 值作为 exit() 系统调用的参数。当 GBase 8s ESQL/C 将 exit() 的参数传递给父代时，它只会传递值的低八位。因为 **SQLCODE** 是一个四字节（s a f**long**）整数，所以 GBase 8s ESQL/C 返回到父进程的值可能不是您期望的。

要在进程之间传递错误消息，请使用退出值作为发生某种联系的错误的指示。要获取有关时间错误的信息，请使用临时文件，数据库表或某种形式的进程间通信。

DESCRIBE 语句之后的 SQLCODE

DESCRIBE 语句返回语句执行的准备好的语句的信息。它使用 PREPARE 语句以前

分配给动态 SQL 语句的语句 ID 进行操作。

成功执行 DESCRIBE 语句之后，数据库服务器将 **SQLCODE**（和 **sqlca.sqlcode**）设置为非负数，它表示 DESCRIBE 已检查的 SQL 语句的类型。sqlstype.h 头文件为每个返回值声明常量名称。

因为 DESCRIBE 语句使用的 **SQLCODE** 字段与其它任何语句不同，所以您可能需要修改异常处理流程以适应这种差异。

检查 sqlca 异常

SQL 语句执行后，**sqlca** 结构可以指示下表所示的四个可能的条件之一。

表 1. sqlca 结构返回的异常

异常条件	sqlca 值
成功	**SQLCODE**（和 **sqlca.sqlcode**）= 0
成功，但未找到行	**SQLCODE**（和 **sqlca.sqlcode**）= 100
成功，但生成警告	**sqlca.sqlwarn.sqlwarn0** = 'W' 要指示特定警告： **sqlwarn1** 至 **sqlwarn7** 其中之一在 **sqlca.sqlwarn** 结构中也被设置为 W
失败，生成运行错误	**SQLCODE**（和 **sqlca.sqlcode**）< 0

sqlca 中成功

当数据库服务器成功执行 SQL 语句时，它将 **SQLCODE**（**sqlca.sqlcode**）设置为 0。数据库服务器可能还会在成功执行 SQL 语句之后在 **sqlca** 中设置以下一个或多个选项字段：

在 SELECT 、INSERT 、DELETE 或 UPDATE 的 PREPARE 之后：

sqlca.sqlerrd[0] 表示估计受影响的行数。

sqlca.sqlerrd[3] 包含磁盘访问和处理的总行的估计加权和。

INSERT 之后，sqlca.sqlerrd[1] 包含数据库服务器为 SERIAL 列生成的值。

SELECT 、INSERT 、DELETE 或 UPDATE 之后：

sqlca.sqlerrd[2] 包含数据库服务器已处理的行数。

sqlca.sqlerrd[5] 包含最后处理的行的 rowid（物理地址）。该 rowid 值是否对应于数据库服务器返回给用户的行，取决于数据库服务器如何处理查询。特别是对于 SELECT 语句。

CONNECT 、SET CONNECTION 、DATABASE 、CREATE DATABASE 或 START DATABASE 之后，sqlca.sqlwarn.sqlwarn0 字段设置为 W，sqlca.sqlwarn 的其它字段提供有关数据库和连接的信息。

SQLCODE 中 NOT FOUND

当 SELECT 或 FETCH 语句遇到 NOT FOUND （或 END OF DATA）时，数据库服务器将 **SQLCODE**（**sqlca.sqlcode**）设置为 100。下表列出了导致 SQL 语句发生 NOT FOUND 的条件。

表 1. 当 SQL 语句未返回任何行时设置的 SQLCODE 值

SQL 语句生成指示的 SQLCODE 结果	符合 ANSI 的数据库的结果	不符合 ANSI 的数据库的结果
FETCH 语句：最后一个排序行已经被返回（达到数据的末尾）。	100	100
SELECT 语句：没有行符合 SELECT 条件。	100	100
DELETE 和 DELETE...WHERE 语句（不是多语句的 PREPARE 的部分）：没有行符合 DELETE 条件。	100	0
INSERT INTO *tablename* SELECT 语句（不是多语句的 PREPARE 的一部分）：没有行符合 SELECT 条件。	100	0
SELECT... INTO TEMP 语句（不是多语句的 PREPARE 的一部分）：没有行符合 SELECT 条件。	100	0
UPDATE 和 UPDATE...WHERE 替换（不是多语句的 PREPARE 的一部分）：没有行符合 UPDATE 条件。	100	0

表 1显示，在某些情况下，NOT FOUND 条件生成的值取决于数据库是否符合 ANSI 标准。

在下列示例中，INSERT 语句将订单数量大于 10,000 的库存项插入到 **hot_items** 表中。如果没有任何项目的订单数量很大，则语句的 SELECT 部分无法插入任何行。数据库服务器在符合 ANSI 的数据库中返回 100 ，在不符合 ANSI 的数据库中返回 0。

```
EXEC SQL insert into hot_items
        select distinct stock.stock_num,
        stock.manu_code,description
```

```
        from items, stock
        where stock.stock_num = items.stock_num
        and stock.manu_code = items.manu_code
        and quantity > 10000;
```

为了可读性，请使用 END OF DATA 值为 100 的常量 SQLNOTFOUND 值。sqlca.h 头文件定义 SQLNOTFOUND 常量。以下比较检查 NOT FOUND 和 END OF DATA 条件：

```
if(SQLCODE == SQLNOTFOUND)
```

sqlca.sqlwarn 中的警告

当数据库服务器成功执行 SQL 语句，但遇到警告条件时，它将更新 **sqlca.sqlwarn** 结构中的以下两个字段：

将 sqlca.sqlwarn.sqlwarn0 字段设置为字母 W。

将 sqlwarn 结构（sqlwarn1 到 sqlwarn7）中的其中一个字段设置为字母 W，来指示特定的警告条件。

这些警告都特定于 GBase 8s。表 1包含可在 **sqlca.sqlwarn** 结构的字段中发生的两组警告条件。表 1中显示的第一组警告发生在数据库服务器打开数据库或建立连接之后。第二组警告是因为其它 SQL 语句可能发生的情况。

要测试警告，请检查第一个警告字段（**sqlwarn0**）是否设置为 W。在您确定数据库服务器生成警告后，可以检查 **sqlca.sqlwarn** 中其它字段的值以识别特定条件。例如，如果要查找 CONNECT 语句打开的数据库，可以使用下图显示的代码：

图: CONNECT 语句后检查警告的代码片段

```
int trans_db, ansi_db, us_db = 0;
        :

        msg = "CONNECT stmt";
        EXEC SQL connect to 'stores7';
        if(SQLCODE < 0)        /* < 0 is an error */
        err_chk(msg);
        if (sqlca.sqlwarn.sqlwarn0 == 'W')
        {
        if (sqlca.sqlwarn.sqlwarn1 == 'W' )
        trans_db = 1;
        if (sqlca.sqlwarn.sqlwarn2 == 'W' )
        ansi_db = 1;
        if (sqlca.sqlwarn.sqlwarn3 == 'W' )
        us_db = 1;
        }
```

SQLCODE 中的运行错误

当 SQL 语句产生运行错误。则数据库服务器将 **SQLCODE**（和 **sqlca.sqlcode**）设置为负值。实际数标识特定错误。错误消息文档列出了 GBase 8s 特定的错误代码和它们的修改操作。

从 GBase 8s ESQL/C 程序中，您可以使用 rgetlmsg() 或 rgetmsg() 库函数来检索与负 **SQLCODE**（**sqlca.sqlcode**）值相关联的错误消息文本。

当数据库服务器遇到运行错误时，它还会设置 **sqlca** 结构中的以下其它字段：

sqlca.sqlerrd[1] 保存附加的 ISAM 错误返回代码。还可以使用 rgetlmsg() 和 rgetmsg() 库函数获取 ISAM 错误消息文本。

sqlca.sqlerrd[2] 指示在多行 INSERT 、UPDATE 或 DELETE 语句中发生错误之前处理的行数。

根据数据库服务器正在使用的类型，sqlca.sqlerrm 的使用方式有所不同。

如果服务器是 GBase 8s 数据库服务器，则此值设置为错误消息参数。该值用于替换错误消息中的 **%s** 令牌。

例如，在以下错误消息中，（**sam.xyz**）表的名称保存在 **sqlca.sqlerrm** 中：

```
310: Table (sam.xyz) already exists in database.
```

如果服务器是 DB2 数据库服务器，则将此字段设置为完整的错误消息。

在执行 PREPARE 、EXECUTE IMMEDIATE 或 DECLARE 语句遇到错误后设置 sqlca.sqlerrd[4]。

提示： 还可以使用 WHENEVER SQLERROR 语句测试错误。

PREPARE 语句执行之后的错误

当数据库服务器返回 PREPARE 语句的错误时，该错误通常是因为准备的文本中的语法错误。当发生此错误时，数据库服务器返回以下信息：

SQLCODE 变量指示错误的原因。

sqlca.sqlerrd[4] 字段包含发生错误的准备语句文本中的偏移量。您的程序可以使用 sqlca.sqlerrd[4] 中的值来指示动态准备文本的语法在哪里不正确。

如果使用一个 PREPARE 语句准备多个语句，则数据库服务器将在文本中的第一个错误上返回错误状态，即使它遇到多个错误。

重要： 由于 GBase 8s ESQL/C 预处理器将嵌入式 SQL 语句转换为主机语言格式，所以将错误偏移到 SQL 语句中的 **sqlerrd[4]** 字段可能并不总是正确的。这样做，预处理器可能会改变嵌入语句中元素的相对位置。

例如，考虑以下语句，其包含无效的 WHERE 子句：

```
EXEC SQL INSERT INTO tab VALUES (:x, :y, :z)
            WHERE i = 2;
```

预处理器将此语句转换为类似于以下字符串的字符串：

```
" insert into tab values ( ? , ? , ? ) where i = 2 "
```

该字符串不具有 EXEC SQL 关键字。另外，字符 ?，?，? 已经替换 :x，:y，:z（五个字符而不是八个字符）。GBase 8s ESQL/C 预处理器也在左括号（")"）和 WHERE 关

键字之间删除了一个换行符。因此，数据库服务器看到的 SQL 语句中的错误偏移量与嵌入式 SQL 语句中错误的偏移量不同。

sqlca.sqlerrd[4] 字段还会报告 EXECUTE IMMEDIATE 和 DECLARE 语句中的错误的语句偏移量。

执行 EXECUTE 语句后的 SQLCODE

执行 EXECUTE 语句后，数据库服务器设置 **SQLCODE** 以指示准备语句的成功，如下所示：

如果数据库服务器不能成功执行准备的语句，则它将 SQLCODE 设置为小于 0 的值。SQLCODE 变量保存数据库服务器从失败的语句返回的错误。

如果数据库服务器可以成功执行块中准备的语句，则将 SQLCODE 设置为 0；如果预备的块包括多个语句，则所有的语句成功。

显示错误文本（Windows）

GBase 8s ESQL/C 应用程序使用 GBase 8s ERRMESS.HLP 文件显示描述错误及其修正确操作的文本。

可以调用具有以下 WinHelp 参数的 Windows™ API WinHelp()。

WinHelp 参数

事件

HELP_CONTEXT

SQLCODE 或 **sqlca.sqlcode** 的错误号

HELP_CONTEXTPOPUP

SQLCODE 或 **sqlca.sqlcode** 的错误号

HELP_KEY

指向包含 SQLCODE 或 **sqlca.sqlcode** 中的错误编号的字符串，并使用 sprintf() 或 wsprintf() 转换为 ASCII

HELP_PARTIALKEY

指向包含 SQLCODE 或 **sqlca.sqlcode** 中的错误编号的字符串，并使用 sprintf() 或 wsprintf() 转换为 ASCII

3.1.4 选择异常处理策略

缺省情况下，GBase 8s ESQL/C 应用程序不会执行任何 SQL 语句的异常处理。因此，除非您显式提供这些代码，否则在执行时继续执行。虽然这种行为对于成功执行、警告和 NOT FOUND 条件可能不太严重，但是在运行错误的情况下可能产生严重后果。

运行错误可能会停止程序执行。除非您在应用程序代码中检查并梳理这些错误，否则这种行为可能会导致用户混淆和困惑。它还使应用程序处于不一致的状态。

在 GBase 8s ESQL/C 应用程序中，请为异常处理选择一致的策略。可以选择以下一种异常处理策略：

可以在每条 SQL 语句执行后检查，这意味着您在每个 SQL 语句之后包含测试 SQLSTATE （或 SQLCODE）值的代码。

可以使用 WHENEVER 语句将响应关联到每次发生特定类型的异常的时候。

重要： 请在开发前考虑如何在应用程序中执行异常处理，以致于可以采取一致和可维护的方法。

在执行每条 SQL 语句后检查

要检查异常，可以包含代码来显式测试 **SQLSTATE**（或 **SQLCODE**）的值。

提示： 请确定使用 **SQLSTATE** （和诊断区域）或 **SQLCODE**（和 **sqlca** 结构）来确定异常值。选择的异常处理变量需一致。如果混淆这两个变量，则您创建的代码很难维护。请记住在这两个选项中 **SQLSTATE** 更灵活更便捷。

例如，如果要使用 **SQLSTATE** 检查 CREATE DATABASE 语句是否按期望执行，则可以使用下图显示的代码。

图: 使用 SQLSTATE 测试错误是否是在 SQL 语句执行期间执行statement

```
EXEC SQL create database personnel with log;
                if(strncmp(SQLSTATE, "02", 2) > 0) /* > 02 is an error */
                {
                EXEC SQL get diagnostics exception 1
                :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
                message[messlen] ='\0'; /* terminate the string. */

                printf("SQLSTATE: %s, %s\n", SQLSTATE, message);
                exit(1);
                }
```

作为备选方案，可以编写处理任何异常的异常处理函数。您的程序然后可以在每个 SQL 语句之后调用一个异常处理函数。

下图显示的 sqlstate_exception() 函数是异常处理函数的示例，它使用 **SQLSTATE** 变量和诊断区域检查警告、NOT FOUND 条件和运行错误。它在每条 SQL 语句之后调用。

图: 使用 SQLSTATE 的异常处理函数的示例

```
EXEC SQL select * from customer where fname not like "%y";
                sqlstate_exception("select");
                :

                int4 sqlstate_exception(s)
                char *s;
                {
                int err = 0;

                if(!strncmp(SQLSTATE, "00", 2) ||
```

```
!strncmp(SQLSTATE,"02",2))
return(SQLSTATE[1]);

if(!strncmp(SQLSTATE, "01", 2))
printf("\n********Warning encountered in %s********\n",
statement);
else /* SQLSTATE class > "02" */
{
printf("\n********Error encountered in %s********\n",
statement);
err = 1;
}

disp_sqlstate_err(); /* See the getdiag sample program */
if(err)
{
printf("********Program terminated*******\n\n");
exit(1);
}

/*
* Return the SQLCODE
*/
return(SQLCODE);
}
```

图 2中显示的 sqlstate_exception() 函数处理下列异常：

如果语句成功，则 sqlstate_exception() 返回零。

如果在 SELECT 或 FETCH 语句发生 NOT FOUND 条件，则 sqlstate_exception() 返回值 2。

如果发生警告或运行错误—即，如果 **SQLSTATE** 的前两个字节是 ″01″（警告）或大于 ″02″（错误）— sqlstate_exception() 函数调用 disp_sqlstate_err() 函数显示异常信息。0

如果 **SQLSTATE** 指示错误，则 sqlstate_exception() 函数使用 exit() 系统调用退出程序。如果不调用 exit()，则会在发生错误的一个语句后的下一个 SQL 语句继续执行。

要处理错误，sqlstate_exception() 函数可以省略 exit() 调用，并允许执行继续执行。在此情况中，函数必须返回 **SQLSTATE** 或 **SQLCODE**（特定于 GBase 8s 的错误）值，而此调用的程序可以确定对此运行错误采取哪种操作。

WHENEVER 语句

可以使用 WHENEVER 语句追踪 SQL 语句执行期间的发生异常。

WHENEVER 语句提供下列信息：

检查哪些条件：

SQLERROR 检查 SQL 语句是否失败。当数据库服务器将 **SQLCODE**（**sqlca.sqlcode**）设置为负值，并将 **SQLSTATE** 的类代码设置为大于 ″02″ 的值时，应用程序执行特定的操作。

NOT FOUND 检查是否找到特定的数据。当数据库服务器将 **SQLCODE**（**sqlca.sqlcode**）设置为 SQLNOTFOUND ，并将 **SQLSTATE** 的类代码设置为 ″02″ 时，应用程序执行特定的操作。

SQLWARNING 检查 SQL 语句是否生成警告。当数据库服务器将 **sqlca.sqlwarn.sqlwarn0**（某些 **sqlca.sqlwarn** 的其它字段）设置为 W ，并将 **SQLSTATE** 的类代码设置为 ″01″ 时，应用程序执行特定的操作。

在 Windows™ 环境中，不能在要编译为 DLL 的 GBase 8s ESQL/C 程序中使用 WHENEVER ERROR STOP 约束。.

当特定条件发生时，采用什么操作：

CONTINUE 忽略此异常并在 SQL 语句之后的下一个语句中继续执行。

GO TO *label* 键异常传输到指定 *label* 介绍的代码节。

STOP 立即停止程序执行。

CALL *函数名称*将执行传输到指定的*函数名称*。

如果不存在 WHENEVER 语句符合给出的条件，则 GBase 8s ESQL/C 预处理器使用 CONTINUE 作为缺省的操作。要在每次错误发生时执行 sqlstate_exception() 函数（如图 2 所示示），可以使用 WHENEVER SQLERROR 语句的 GOTO 操作，如果指定了 WHENEVER 的 SQLERROR 条件，则可以获得与每个 SQL 语句之后检查 **SQLCODE** 或 **SQLSTATE** 变量的错误相同的行为。

GOTO 操作的 WHENEVER 语句可以采取以下两种形式：

ANSI 标准形式使用关键字 GOTO （一个词），并使用冒号（:）引入标签名称：

```
EXEC SQL whenever goto :error_label;
```

GBase 8s 扩展使用关键字 GO TO（两个词）并仅指定 label 名称：

```
EXEC SQL whenever go to error_label;
```

使用 GOTO 操作，当 SQL 语句生成异常时，程序会自动将控件传输到 **error_label** 标签。当使用 WHENEVER 语句的 GOTO *label* 操作时，您的代码必须包含标签和适当逻辑来处理错误条件。在以下示例中， *label* 中的逻辑只是调用 sqlstate_exception() 函数：

```
error_label:
        sqlstate_exception (msg);
```

您必须在包含 SQL 语句的每个程序块中定义此 **error_label** 标签。如果您的程序包含多个函数，则可能需要在每个函数中包含 **error_label** 标签和代码。否则，预处理器到达不包含 **error_label** 的函数时会产生错误。它尝试插入 WHENEVER...GOTO 语句请求的代码，但该函数尚未定义 **error_label** 标签。

要删除预处理器错误，您可以在每个函数中放置带有相同标签名称的带标签的语句，您可以为 WHENEVER 语句发出另一个操作来重置错误条件，也可以使用 CALL 操作替换 GOTO 操作，以调用分离函数。

当错误发生时，还可以在 WHENEVER 语句中使用 CALL 关键字调用 sqlstate_exception() 函数。（CALL 选项是 ANSI 标准的 GBase 8s 扩展名。）

如果您要在程序中每次发生 SQL 错误时调用 sqlstate_exception() 函数，请执行以下步骤：

修改 sqlstate_exception() 函数，使其不需要任何参数。 CALL 操作指定的函数不能接受参数。要传递信息，请改用全局变量。

在任何 SQL 语句之前，将以下 WHENEVER 语句放置程序的前面部分：

```
EXEC SQL whenever sqlerror call sqlstate_exception;
```

提示： 在之前的代码段中，不能在 sqlstate_exception() 函数之后包含括号。

但是，请确保，WHENEVER...CALL 影响的所有函数都可以找到 sqlstate_exception() 函数的声明。

3.1.5 检索错误消息的库函数

每个 **SQLCODE** 值具有相关联消息。$GBASEDBTDIR/msg 目录中的错误消息文件存储消息编号和它的文本。

当使用 **SQLCODE** 和 **sqlca** 结构时，可以使用 rgetlmsg() 或 rgetmsg() 函数检索错误消息。这些函数都将 **SQLCODE** 错误代码作为输入并返回相关联的错误消息。

提示： 当使用 **SQLSTATE** 和 GET DIAGNOSTICS 语句时，可以访问诊断区域 **MESSAGE_TEXT** 的字段中的信息，来检索与异常相关联的消息文本。

重要： 在您写的任何新的 GBase 8s ESQL/C 代码中使用 rgetlmsg()。GBase 8s ESQL/C 提供 rgetmsg() 函数主要为了兼容较早的版本。

在 Windows 环境中的显示错误消息

GBase 8s ESQL/C 应用程序可以使用 GBase 8s ERRMESS.HLP 文件显示描述错误及其修正操作的文本。

可以使用 WinHelp 参数调用 Windows™ API WinHelp()。

WinHelp 参数

数据
HELP_CONTEXT
SQLCODE 或 **sqlca.sqlcode** 的错误号
HELP_CONTEXTPOPUP
SQLCODE 或 **sqlca.sqlcode** 的错误号

HELP_KEY

指向包含 SQLCODE 或 **sqlca.sqlcode** 中的错误编号的字符串，并使用 sprintf() 或 wsprintf() 转换为 ASCII

HELP_PARTIALKEY

指向包含 SQLCODE 或 **sqlca.sqlcode** 中的错误编号的字符串，并使用 sprintf() 或 wsprintf() 转换为 ASCII

3.1.6 使用异常处理的程序

getdiag.ec 程序包含其执行的每个 SQL 语句的异常处理。本程序是 demo1.ec 程序的改版。本节列出的版本和描述使用以下异常处理方法：

SQLSTATE 变量和 GET DIAGNOSTICS 语句获取异常信息。

WHENEVER 语句的 SQLWARNING 和 SQLERROR 关键字为警告和错误调用 whenexp_chk() 函数。

whenexp_chk() 函数显示错误号和伴随的 ISAM 错误，如果存在。exp_chk.ec 源文件包含此函数及其异常处理函数。getdiag.ec 源文件包含 exp_chk.ec 文件。

编译程序

使用以下命令编译 **getdiag** 程序：

```
esql -o getdiag getdiag.ec
```

-o getdiag 选项告知 esql 命名可执行程序 **getdiag**。若不使用 -o 选项，可选择程序的名称缺省为 a.out。

getdiag.ec 文件指南

本节的注释主要描述异常处理语句。

```
================================================================
=========
1. #include <stdio.h>
2. EXEC SQL define FNAME_LEN 15;
3. EXEC SQL define LNAME_LEN 15;
4. int4 sqlstate_err();
5. extern char statement[20];
6. main()
7. {
8.          EXEC SQL BEGIN DECLARE SECTION;
9.          char fname[ FNAME_LEN + 1 ];
10.          char lname[ LNAME_LEN + 1 ];
11.          EXEC SQL END DECLARE SECTION;
12.          EXEC SQL whenever sqlerror CALL whenexp_chk;
13.          EXEC SQL whenever sqlwarning CALL whenexp_chk;
14.          printf("GETDIAG Sample ESQL program running.\n\n");
15.          strcpy (statement, "CONNECT stmt");
16.          EXEC SQL connect to 'stores7';
```

```
17.         strcpy (statement, "DECLARE stmt");
18.         EXEC SQL declare democursor cursor for
19.             select fname, lname
20.             into :fname, :lname;
21.             from customer
22.             where lname < 'C';
23.         strcpy (statement, "OPEN stmt");
24.         EXEC SQL open democursor;
25.         strcpy (statement, "FETCH stmt");
26.         for (;;)
27.         {
28.             EXEC SQL fetch democursor;
29.             if(sqlstate_err() == 100)
30.                 break;
31.             printf("%s %s\n", fname, lname);
32.         }
33.         strcpy (statement, "CLOSE stmt");
34.         EXEC SQL close democursor;
```

第 4 行

第 4 行声明外部区间变量来保存最近执行 SQL 语句的名称。异常处理函数使用此信息（请参阅 第 169 - 213 行）。

第 12 和 13 行

WHENEVER SQLERROR 语句告知 GBase 8s ESQL/C 预处理器向程序中添加代码，以在 SQL 语句生成错误时调用 whenexp_chk() 函数。WHENEVER SQLWARNING 语句告知 GBase 8s ESQL/C 预处理器向程序添加代码，以在 SQL 语句产生警告时调用 whenexp_chk() 函数。whenexp_chk() 函数在 exp_chk.ec 文件中，第 40 行包含。

第 15 行

strcpy() 函数将字符串 "CONNECT stmt" 复制到全局 **statement** 变量。如果错误发生，则 whenexp_chk() 函数使用此变量打印导致失败的语句的名称。

第 17 、23 、25 和 33 行

这些行在 DECLARE 、OPEN 、FETCH 和 CLOSE 语句执行前，将当前 SQL 语句的名称复制到 **statement** 变量中。如果发生错误，则此操作启用 whenexp_chk() 函数标识错误的语句。

```
36.         strcpy (statement, "FREE stmt");
37.         EXEC SQL free democursor;
38.         strcpy (statement, "DISCONNECT stmt");
39.         EXEC SQL disconnect current;
```

```
40.             printf("\nGETDIAG Sample Program Over.\n");
41. }               /* End of main routine */
42. EXEC SQL include exp_chk.ec;
```

```
================================================================
=====
```

第 35 和 37 行

这些行在 FREE 和 DISCONNECT 语句执行之前，将当前 SQL 语句的名称复制到 **statement** 变量中。如果发生错误，whenexp_chk() 函数使用 **statement** 变量标识失败的语句。

第 41 行

whenexp_chk() 函数检查 **SQLSTATE** 状态变量来确定 SQL 语句的输出。因为几个演示程序使用具有 WHENEVER 语句的 whenexp_chk() 函数来处理异常， whenexp_chk() 函数及其支持函数存放在各自的源文件中 exp_chk.ec 中， **getdiag** 程序必须包括具有 GBase 8s ESQL/C **include** 伪指令的文件，因为，异常处理函数使用 GBase 8s ESQL/C 语句。

提示： 当编译 GBase 8s ESQL/C 程序时，请考虑将诸如 whenexp_chk() 的函数放到库中，并在命令行中包含此库。

exp_chk.ec 文件指南

exp_chk.ec 文件包含 GBase 8s ESQL/C 演示程序的异常处理函数。

这些函数支持以下两种种类的异常处理：

WHENEVER SQLERROR CALL 语句指定执行的异常处理函数。

支持此种异常处理的函数包括 whenexp_chk() 、sqlstate_err() 和 disp_sqlstate_err()。本章节的 **getdiag** 示例程序使用此形式的异常处理。

GBase 8s ESQL/C 程序在每条 SQL 语句执行之后显式执行异常处理的函数。

支持此种异常处理的函数包括 exp_chk() 、exp_chk2() 、sqlstate_err() 、**disp_sqlstate_err()** 和 disp_exception()。**dispcat_pic** 示例程序（简单大对象）使用 exp_chk2()，**dyn_sql** 示例程序（系统描述符区域）使用 exp_chk() 执行异常处理。

要获得异常信息，上述函数使用 **SQLSTATE** 变量和 GET DIAGNOSTICS 语句。它们只有在需要特定 GBase 8s 的信息时使用 **SQLCODE**。

```
    ================================================================
=========
    1. EXEC SQL define SUCCESS 0;
    2. EXEC SQL define WARNING 1;
    3. EXEC SQL define NODATA   100;
    4. EXEC SQL define RTERROR -1;
    5. char statement[80];
    6. /*
```

```
7.  * The sqlstate_err() function checks the SQLSTATE status variable
     * to see
8.  * if an error or warning has occurred following an SQL statement.
9.  */
10. int4 sqlstate_err()
11. {
12.     int4 err_code = RTERROR;
13.     if(SQLSTATE[0] == '0') /* trap '00', '01', '02' */
14.         {
15.         switch(SQLSTATE[1])
16.             {
17.             case '0': /* success - return 0 */
18.                 err_code = SUCCESS;
19.                 break;
20.             case '1': /* warning - return 1 */
21.                 err_code = WARNING;
22.                 break;
23.             case '2': /* end of data - return 100 */
24.                 err_code = NODATA;
25.                 break;
26.             default: /* error - return -1*/
27.                 break;
28.             }
29.         }
30.     return(err_code);
31. }
================================================================
=========
```

第 1 - 4 行

这些 GBase 8s ESQL/C **定义**指令创建成功、警告、NOT FOUND 和运行错误异常的定义。此文件中的函数使用这些定义而不是常量来确定对给出类型的异常采取的操作。

第 5 行

statement 变量是一个全局变量，它调用程序（声明为 **extern**）设置最近执行 SQL 语句的名称。

whenexp_chk() 函数显示 SQL 语句作为错误消息的一部分（请参阅行 85 和 92）。

第 6 - 31 行

sqlstate_err() 函数返回 0 、1 、100 或 −1 的状态，来指示当前 **SQLSTATE** 中的异常是成功、警告、NOT FOUND 还是运行错误。sqlstate_err() 函数检查 **SQLSTATE** 全局变量的前两个字母。因为 GBase 8s ESQL/C 自动声明 **SQLSTATE** 变量，所以此函数不需要声明它。

第 13 行检查 **SQLSTATE** 全局变量的第一个字母。此字符定义最近执行的 SQL 语句释放生成一个无错误条件。无错误条件包括 NOT FOUND 条件（或 END OF DATA）、成功和警告。第 15 行检查 **SQLSTATE**（**SQLSTATE[1]**）变量的第二个字母，来确

定生成的无错误条件的类型。

sqlstate_err() 函数设置 **err_code** 来显示以下异常状态：

第 17 - 19 行：如果 SQLSTATE 具有类代码 "00"，则最近执行的 SQL 语句成功。sqlstate_err() 函数返回 0（第 1 行定义为 SUCCESS）。

第 20 - 22 行：如果 SQLSTATE 具有类代码 "01"，则最近执行的 SQL 语句生成一个警告。sqlstate_err() 函数返回 1（第 2 行定义为 WARNING）。

第 23 - 25 行：如果 SQLSTATE 具有类代码 "02"，则最近执行的 SQL 语句生成 NOT FOUND（或 END OF DATA）条件。sqlstate_err() 函数返回 100（第 3 行定义为 NODATA）

如果 **SQLSTATE[1]** 包含其它字符而不是 '0'、'1' 或 '2'，则最近执行的 SQL 语句生成一个运行错误。 如果 **SQLSTATE[1]** 包含某些字符而不是 '0' 则 **SQLSTATE** 表示一个运行错误。在这两种情况中，第 30 行返回（-1）（第 4 行定义为 RTERROR）。

```
    ================================================================
=========
32. /*
33.   * The disp_sqlstate_err() function executes the GET DIAGNOSTICS
34.   * statement and prints the detail for each exception that is
35.   * returned.
36.   */
37. void disp_sqlstate_err()
38. {
39. mint j;
40. EXEC SQL BEGIN DECLARE SECTION;
41.         mint exception_count;
42.         char overflow[2];
43.         int exception_num=1;
44.         char class_id[255];
45.         char subclass_id[255];
46.         char message[8191];
47.         mint messlen;
48.         char sqlstate_code[6];
49.         mint i;
50. EXEC SQL END DECLARE SECTION;
51.         printf("--------------------------------");
52.         printf("-------------------------\n");
53.         printf("SQLSTATE: %s\n",SQLSTATE);
54.         printf("SQLCODE: %d\n", SQLCODE);
55.         printf("\n");
56.       EXEC SQL get diagnostics :exception_count = NUMBER,
57.               :overflow = MORE;
58.         printf("EXCEPTIONS:    Number=%d\t", exception_count);
59.         printf("More? %s\n", overflow);
```

```
60.       for (i = 1; i <= exception_count; i++)
61.          {
62.          EXEC SQL get diagnostics    exception :i
63.              :sqlstate_code = RETURNED_SQLSTATE,
64.              :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
65.              :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
66.          printf("- - - - - - - - - - - - - - - - - - -\n");
67.          printf("EXCEPTION %d: SQLSTATE=%s\n", i,
68.              sqlstate_code);
69.          message[messlen-1] = '\0';
70.          printf("MESSAGE TEXT: %s\n", message);
71.          j = byleng(class_id, stleng(class_id));
72.          class_id[j] = '\0';
73.          printf("CLASS ORIGIN: %s\n",class_id);
74.          j = byleng(subclass_id, stleng(subclass_id));
75.          subclass_id[j] = '\0';
76.          printf("SUBCLASS ORIGIN: %s\n",subclass_id);
77.          }
78.      printf("---------------------------------");
79.      printf("-------------------------\n");
80. }
    ================================================================
=========
```

第 32 - 80 行

disp_sqlstate_err() 函数使用 GET DIAGNOSTICS 语句获取有关最近执行的 SQL 语句的诊断信息。

第 40 - 50 行声明接收诊断信息的主机变量。GET DIAGNOSTICS 语句将诊断区域的信息复制到这些主机变量中。第 48 行包括 **SQLSTATE** 值的声明名称（**sqlstate_code**），因为 disp_sqlstate_err() 函数处理多个异常。**sqlstate_code** 变量保存每个异常的 **SQLSTATE** 值。

第 53 - 55 行显示 **SQLSTATE** 和 **SQLCODE** 变量的值。如果 **SQLSTATE** 包含 ″IX000″（GBase 8s 特定的错误），则 **SQLCODE** 包含特定于 GBase 8s 的错误代码。

第一个 GET DIAGNOSTICS 语句（第 56 和 57 行）存储 **:exception_count** 和 **:overflow** 主机变量中的语句信息。第 58 和 59 行显示这些信息。

第 60 - 77 行为最近执行的 SQL 语句生成的每个异常执行 **for** 循环。**:exception_count** 主机变量，保存异常号，确定此循环循环的迭代次数。

第二条 GET DIAGNOSTICS 语句（第 62 - 65 行）获取单个异常的异常信息。 第 67 - 70 行打印 **SQLSTATE** 值（**sqlstate_code**）及其对应的消息文本。此外，SQL 错误消息 disp_sqlstate_err() 可以显示 ISAM 错误消息，因为诊断区域的 MESSAGE_TEXT 字段还包含这些消息。该函数使用 MESSAGE_LENGTH 值确定空终止符放置在消息字符串中的位置。此操作仅导致输出包含文本的消息变量的一部分（而不是所有的 255-字符缓冲

区）。

将类和子类起始的主机变量声明为大小为 255 的字符缓冲区。但是，这些变量的文本通常仅填充缓冲区的一部分，而不显示整个缓冲区，第 71 - 73 行使用 GBase 8s ESQL/C byleng() 和 stleng() 库函数只显示包含文本的 **:class_id** 的一部分；第 74 - 76 行对 **:subclass_id** 执行相同的操作。

```
    ================================================================
=========
    81. void disp_error(stmt)
    82. char *stmt;
    83. {
    84.     printf("\n********Error encountered in %s********\n",
    85.         stmt);
    86.     disp_sqlstate_err();
    87. }
    88. void disp_warning(stmt)
    89. char *stmt;
    90. {
    91.     printf("\n********Warning encountered in %s********\n",
    92.         stmt);
    93.     disp_sqlstate_err();
    94. }
    95. void disp_exception(stmt, sqlerr_code, warn_flg)
    96. char *stmt;
    97. int4 sqlerr_code;
    98. mint warn_flg;
    99. {
    100.    switch(sqlerr_code)
    101.        {
    102.        case SUCCESS:
    103.        case NODATA:
    104.            break;
    105.        case WARNING:
    106.            if(warn_flg)
    107.                disp_warning(stmt);
    108.            break;
    109.        case RTERROR:
    110.            disp_error(stmt);
    111.            break;
    112.        default:
    113.            printf("\n********INVALID EXCEPTION STATE for
        %s********\n",
    114.                stmt);
    115. /*          break;
    116.        }
    117. }
    ================================================================
=========
```

第 81 - 87 行

disp_error() 函数通知用户运行错误。它调用 disp_sqlstate_err() 函数（行 86）显示诊断信息。

第 88 - 94 行

disp_warning() 函数通知用户警告。它调用 disp_sqlstate_err() 函数（行 93）显示诊断信息。

第 95 - 117 行

disp_exception() 函数处理异常信息的显示。它期望以下三个参数：

stmt

最近执行 SQL 语句的名称。

sqlerr_code

sqlstate_err() 返回的代码，以显示遇到的异常类型。

warn_flg

指示是否显示警告的诊断信息的标志。

第 102 - 104 行处理 SUCCESS 和 NOData 条件。对于这些情况，函数不显示诊断信息。第 105 - 108 行通知用户发生警告。此函数检查 **warn_flg** 参数来确定是否调用 disp_warning() 函数显示最近执行的 SQL 语句（第 137 - 142 行）的警告信息。第 109 - 111 行通知用户发生运行错误。disp_err() 函数实际上处理诊断信息的显示。

```
    ================================================================
=========
118.   * The exp_chk() function calls sqlstate_err() to check the SQLSTATE
119.   * status variable to see if an error or warning has occurred
         * following
120.   * an SQL statement. If either condition has occurred, exp_chk()
121.   * calls disp_sqlstate_err() to print the detailed error
         * information.
122.   *
123.   * This function handles exceptions as follows:
124.   *       runtime errors - call exit()
125.   *       warnings - continue execution, returning "1"
126.   *       success - continue execution, returning "0"
127.   *       Not Found - continue execution, returning "100"
128.   */
129. long exp_chk(stmt, warn_flg)
130. char *stmt;
131. int warn_flg;
132.   {
133.       int4 sqlerr_code = SUCCESS;
134.       sqlerr_code = sqlstate_err();
135.       disp_exception(stmt, sqlerr_code, warn_flg);
```

```
136.      if(sqlerr_code == RTERROR)       /* Exception is a runtime error */
137.      {
138.          /* Exit the program after examining the error */
139.          printf("********Program terminated********\n\n");
140.          exit(1);
141.      }
142. /*    else                            /* Exception is "success", "Not Found", */
143 .         return(sqlerr_code);     /* or "warning" */
144. }
```

```
==========
```

第 118 - 144 行

exp_chk() 函数是处理异常的三个包装函数只有。它解析 **SQLSTATE** 值来确定最近执行的 SQL 语句的成功或失败。每个 SQL 语句之后都会显式调用该函数。此设计需要以下函数：

exp_chk() 函数，作为参数传递生成异常的 SQL 语句的名称。

因为 WHENEVER 语句不会调用此函数，所以此函数不限制于使用全局变量。

在成功执行 SQL 语句（0）、NOT FOUND 条件或警告（1）的情况下，exp_chk() 函数返回一个值。

exp_chk() 函数使用标志参数（warn_flg）指示是否显示警告信息给用户。

因为警告可以指示非严重错误，并且在 CONNECT 之后可以提供信息，所以显示警告消息对于用户来说可能是分心和不必要的。**warn_flg** 参数允许调用程序确定是否显示 SQL 语句可能生成的警告消息。

sqlstate_err() 函数（行 134）确定 **SQLSTATE** 包含的异常类型。 函数然后调用 disp_exception()（行 135），并传递将 **warn_flg** 参数以指示是否显示警告信息。要处理运行错误，sqlstate_err() 函数调用 exit() 系统函数（第 136 - 141 行）来终止程序。该行为与 whenexp_chk() 函数（参见行 170 - 214）为运行的错误提供的相同。

dyn_sql 示例程序还使用 exp_chk() 处理异常。

```
==========
145.  * The exp_chk2() function calls sqlstate_err() to check the
          * SQLSTATE
146.  * status variable to see if an error or warning has occurred
          * following
147.  * an SQL statement. If either condition has occurred, exp_chk2()
148.  * calls disp_sqlstate_err() to print the detailed error
          * information.
149.  *
150.  * This function handles exceptions as follows:
151.  *       runtime errors - continue execution, returning SQLCODE (<0)
152.  *       warnings - continue execution, returning one (1)
```

```
153.  *       success - continue execution, returning zero (0)
154.  *       Not Found - continue execution, returning 100
155.  */
156. int4 exp_chk2(stmt, warn_flg)
157. char *stmt;
158. mint warn_flg;
159.  {
160.      int4 sqlerr_code = SUCCESS;
161.      int4 sqlcode;
162.      sqlcode = SQLCODE;       /* save SQLCODE in case of error */
163.      sqlerr_code = sqlstate_err();
164.      disp_exception(stmt, sqlerr_code, warn_flg);
165.      if(sqlerr_code == RTERROR)
166. /*          sqlerr_code = sqlcode;
167.      return(sqlerr_code);
168. }
```

```
=========
```

第 145 - 168 行

exp_chk2() 函数是 exp_chk.ec 文件中三个异常处理包装函数中的第二个。它执行与 exp_chk() 函数相同的基本任务。在每个 SQL 语句之后调用这两个函数，并返回一个状态代码。两者之间的唯一区别就是它们对运行的错误的响应。exp_chk() 函数调用 exit() 终止程序（行 140），而 exp_chk2() 函数将 **SQLCODE** 值返回给调用程序（第 165 - 166 行）。

exp_chk2() 函数返回 **SQLCODE** 而不是 **SQLSTATE** 来允许程序检查特定于 GBase 8s 的特定错误代码。可能的增强可能是返回 **SQLSTATE** 和 **SQLCODE** 值。

dyn_sql 示例程序还使用 exp_chk2() 处理程序。

```
=========
169.      *
170. * The whenexp_chk() function calls sqlstate_err() to check the
       * SQLSTATE
171.  * status variable to see if an error or warning has occurred
       * following
172.  * an SQL statement. If either condition has occurred, whenerr_chk()
173.  * calls disp_sqlstate_err() to print the detailed error
       * information.
174.  *
175.  * This function is expected to be used with the WHENEVER SQLERROR
176.  * statement: it executes an exit(1) when it encounters a negative
177.  * error code. It also assumes the presence of the "statement"
       * global
178.  * variable, set by the calling program to the name of the statement
179.  * encountering the error.
180. */
181. whenexp_chk()
```

```
182. {
183.     int4 sqlerr_code = SUCCESS;
184.     mint disp = 0;
185.     sqlerr_code = sqlstate_err();
186.     if(sqlerr_code == WARNING)
187.         {
188.             disp = 1;
189.             printf("\n********Warning encountered in %s********\n",
190.                 statement);
191.         }
192.     else
193.             if(sqlerr_code == RTERROR)
194.             {
195.                 printf("\n********Error encountered in %s********\n",
196.                     statement);
197.                 disp = 1;
198.             }
199.     if(disp)
200.             disp_sqlstate_err();
201.     if(sqlerr_code == RTERROR)
202.         {
203.             /* Exit the program after examining the error */
204.             printf("********Program terminated*******\n\n");
205.             exit(1);
206.         }
207.     else
208.         {
209.       if(sqlerr_code == WARNING)
210.             printf("\n********Program execution
      continues********\n\n");
211.       return(sqlerr_code);
212.         }
213. }
    ============================================================
=========
```

第 169 - 213 行

whenexp_chk() 是 exp_chk.ec 文件中三个异常处理包装函数中的第三个函数。它分析 **SQLSTATE** 值，并对异常处理使用 GET DIAGNOSTICS 语句。但是，该函数与下列 WHENEVER 语句一起调用：

```
    EXEC SQL whenever sqlerror call whenexp_chk;
            EXEC SQL whenever sqlwarning call whenexp_chk;
```

WHENEVER 语句对 whenexp_chk() 函数的设计施加以下限制：

whenexp_chk()语句不能接收参数；因此，函数使用全局变量 statement 来标识生成异常的 SQL 语句（第 190 和 196 行）。

要在 whenexp_chk() 函数中使用参数，可以使用 WHENEVER 语句的 GOTO 子句。

```
EXEC SQL whenever sqlerror goto :excpt_hndlng;
```

其标签 **:excpt_hndlng** 可能具有以下代码：

```
:excpt_hndlng
                whenexp_chk(statement);
```

whenexp_chk() 函数不返回任何值；因此，它不会将此特定的异常返回给主程序。

出于此原因，whenexp_chk() 处理运行错误而不是主程序；当它遇到运行错误时，whenexp_chk() 调用 **exit()** 函数。要用主程序访问错误代码。可以修改 whenexp_chk() 来设置全局变量。

本节描述的 **getdiag** 示例程序，使用 whenexp_chk() 处理异常。参见 getdiag.ec 文件指南中的 getdiag.ec 文件的第 11 和 12 行。

sqlstate_err() 函数（行 185）返回整数来指示最近执行的 SQL 语句的成功。此返回值基于 **SQLSTATE** 值。

第 186 - 198 行显示特殊的行，来将注意力引到生成的异常信息的。**disp** 变量是一个标志，它指示是否显示异常信息。该函数显示警告（WARNING）和运行错误（RTERROR）的异常信息，但不显示其它异常条件的信息。调用 printf() 函数（第 189 和 195 行）显示生成警告或错误的 SQL 语句的名称。全局变量（名为 **statement**）必须存储语句名称，因为函数不能将它作为参数接收。

只要 **SQLSTATE** 指示警告或运行错误（**disp** = 1），则 disp_sqlstate_err() 函数（第 199 和 200 行）显示诊断区域包含的信息。

第 201 - 206 行处理运行错误。它们通知用户程序终止。然后使用 exit() 系统调用（行 205）来终止此程序。调用的 disp_sqlstate_err() 函数（行 200）已经显示了有关运行错误的原因的信息。

3.2 使用数据库服务器

这些主题描述了 GBase 8s ESQL/C 程序如何与数据库服务器交互。

它包含以下信息：

GBase 8s ESQL/C 应用程序的客户端访问架构描述

GBase 8s ESQL/C 程序与数据库服务器交互方式的概述

控制数据库服务器的 GBase 8s ESQL/C 库函数的语法

这些主题的最后呈现了一个名为 **timeout** 的注释示例程序。**timeout** 示例程序演示如何中断 SQL 请求。

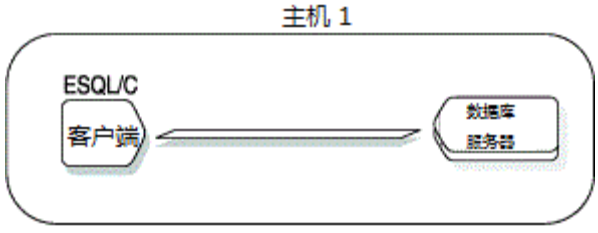
3. 2. 1 ESQL/C 应用程序的客户端服务器架构

当 GBase 8s ESQL/C 程序执行 SQL 语句时，它有效地将语句传递给数据库服务器。数据库服务器从数据库应用程序接收 SQL 语句，传递它们，优化数据检索路径，接收来自数据库的数据，并将数据和状态信息返回给应用程序。

GBase 8s ESQL/C 程序和数据库服务器通过进程间通信机制相互通信。 GBase 8s ESQL/C 程序是对话中的客户端进程，因为它从数据库服务器请求信息。数据服务器是服务器进程，因为它提供了响应客户端请求的信息。客户端和服务器进程之间的分工在网络中是有利的，其中数据可能不在与需要它的客户端程序相同的计算机上。

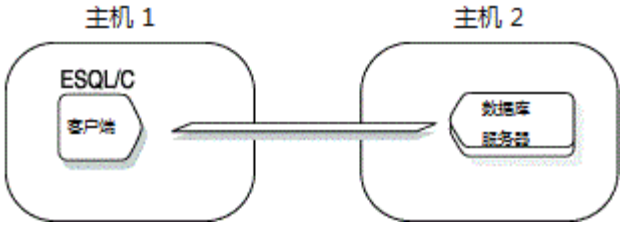
当编译 GBase 8s ESQL/C 程序时，它将自动配置为与同一台计算机（本地）或其他计算机（远程）上的网络上的数据库服务器进行通信。下图显示了 GBase 8s ESQL/C 应用程序和本地数据库服务器之间的连接。

图 1. 连接到本地数据库服务器的 ESQL/C 应用程序



下图显示了跨网络的 GBase 8s ESQL/C 应用程序与远程数据库服务器的连接。

图 2. 连接到远程数据库服务器的 ESQL/C 应用程序



要建立与数据库服务器的连接，您的应用程序必须执行以下操作：

标识已经为应用程序的客户端服务器环境定义的数据库服务器连接

执行 SQL 语句以连接到数据库服务器

3.2.2 客户端-服务器连接

GBase 8s ESQL/C 应用程序建立与任何有效的数据库环境的连接。数据库环境可以是数据库、数据库服务器或数据库和数据库服务器。

每个数据库必须具有数据库服务器来管理它的信息。要建立连接，客户端应用程序必须能够定位有关可用的数据库服务器的信息。该信息在 sqlhosts 文件或注册表中。运行时，应用程序必须能够访问与连接相关的环境变量的的信息。可以访问下列环境变量：

GBASEDBTCONTIME

定义客户端尝试连接的最少指定时间

GBASEDBTCONRETRY

定义由 GBASEDBTCONTIME 指定的时间内客户端尝试的连接数

GBASEDBTSQLHOSTS

定义在哪里找到 sqlhosts 信息。sqlhosts 信息包含客户端能连接的有效数据库服务器的列表、要使用的连接类型以及每个数据库服务器所在的服务器机器名称。在 UNIX(TM) 操作系统中，它是到文件的路径。在 Windows(TM) 环境中，它是网络中包含客户端应用程序可访问的中央注册表的计算机的名称。

GBASEDBTSERVER

指定客户端连接到缺省数据库服务器的名称。此值标识用于建立数据库连接的 sqlhosts 文件或在注册表中的哪个条目。

重要： 当应用程序未明确指定连接的数据库服务器时，客户端应用程序将连接到缺省的数据库服务器。即使应用程序未建立缺省的数据库服务器的连接，也必须设置 GBASEDBTSERVER 环境变量。

客户端还发送环境变量，以便数据库服务器确定服务器处理区域设置。

当数据库服务器处理应用程序请求时，它使用适当的环境信息。它忽略任何不相关的信息。例如，如果应用程序具有亚洲语言支持（ALS）的数据库的环境变量，但是它连接到非 ALS 数据库，则数据库服务器忽略 ALS 信息。

UNIX(TM) 操作系统的连接信息的源

sqlhosts 文件，包含网络环境中所有有效数据库服务器的定义。

GBASEDBTSERVER 环境变量，为应用程序指定缺省数据库服务器。

访问 sqlhosts 文件

要建立到数据库服务器的连接，应用程序进程必须能够在 sqlhosts 文件中找到数据库服务器的条目。sqlhosts 文件定义对客户端-服务器环境有效的数据库服务器连接。对于每个数据库服务器，此文件定义以下信息：

数据库服务器的名称。

在客户端应用程序和数据库服务器之间连接的连接类型。

数据库服务器所在的主机计算机的名称。

用于建立连接的系统文件或程序名。

应用程序期望在 $GBASEDBTDIR/etc 目录找到 sqlhosts 文件。但是，您可以使用 GBASEDBTSQLHOSTS 环境变量更改此位置或文件名。如果数据库服务器不在运行客户端程序的计算机上，则 sqlhosts 文件必须位于 GBase 8s ESQL/C 客户端程序和数据库服务器的主机上。

客户端应用程序可以连接到 sqlhosts 文件定义的任何数据库服务器。如果您的应用程序需要连接 sqlhosts 未定义的数据库服务器，则可能需要数据库管理器（DBA）的帮助才能在此文件中创建必要的条目。除了 sqlhosts 文件之外，您还可能需要配置系统网络文件以支持连接。您的 *GBase 8s 管理员指南* 介绍如何在 sqlhosts 文件中创建数据库服务器项目。

如果您启用单点登录（SSO），则还需要配置 ESQL/C 和 ODBC 驱动程序的 SSO。

指定缺省数据库服务器

为了您的 GBase 8s ESQL/C 应用程序连接任何数据库服务器，必须设置 GBASEDBTSERVER 环境变量来指定缺省数据库服务器的名称。因此，缺省数据库服务器的名称必须在 sqlhosts 文件中存在，并且 sqlhosts 必须在运行应用程序的计算机上存在。

Windows(TM) 环境中连接信息的源

要建立与数据库服务器的连接，Windows(TM) 环境中的 GBase 8s ESQL/C 应用程序执行下列操作：

提供有关与注册表、ifx_putenv() 函数或 InetLogin 结构的信息

使用中央注册表连接信息

为应用程序用户执行连接身份的信息

在 Windows(TM) 环境中，GBase 8s ESQL/C 获取来自 **InetLogin** 结构或注册表的内存中复制获取的配置信息。

如果应用程序已经初始化 **InetLogin** 中的一个字段，则 GBase 8s ESQL/C 将此值发送到数据库服务器。对应用程序未在 **InetLogin** 结构中设置任何字段，GBase 8s ESQL/C 使用注册表的 GBase 8s 子键的对应的信息。

重要： 因为应用程序需要配置信息建立连接，所以必须在执行建立连接的 SQL 语句之前设置任何 **InetLogin** 配置值。

注册表包含以下配置信息：

GBase 8s 环境变量的值

连接信息

当客户端 GBase 8s ESQL/C 应用程序建立到数据库服务器的连接时，它将配置信息发送到数据服务器。

为 Windows(TM) 环境中的连接设置环境变量

注册表提供大多数环境变量的缺省值。

Windows(TM) 环境中的 sqlhosts 信息

注册表包含下列连接信息：

sqlhosts 信息定义与建立的数据库服务器的连接。

此选项包括主机计算机的名称、要使用的协议类型以及连接的名称。注册表将 sqlhosts 信息存储在 GBase 8s 密钥的 **SqlHosts** 子项中。要在注册表中存储 sqlhosts 信息，请使用 Setnet32 实用程序的 **服务器信息** 选项卡。

.netrc 信息定义远程连接的有效用户。

在 UNIX(TM) 操作系统中，此文件在用户的主目录中，并指定了用户账户的名称和密码。在 Windows(TM) 环境中，注册表中的 GBase 8s 密钥的 **NETRC** 子项存储相同的账户信息。要在注册表中存储 .netrc 信息，请使用 Setnet32 实用程的 **Host Information** 选项卡。

客户端发送网络参数以建立与数据库服务器的连接。建立连接的第一步是登录到正确的主机。协议软件使用当前数据库服务器的网络参数。客户端以下列方式之一查找当前数据库服务器的网络参数：

如果请求连接的 SQL 语句（如 CONNECT 或 DATABASE）指定数据库服务器的名称。客户端将发送此指定数据库服务器的网络参数。

如果 **InetLogin** 的 **InfxServer** 字段包含指定的数据库服务器的名称。则客户端检查网络参数的 **InetLogin** 。否则，客户端从注册表的内存章复制获取该数据库服务器的网络参数。

如果 SQL 语句未指定数据库服务器，则客户端发送缺省数据库服务器的网络参数。

如果 **InetLogin** 的 **InfxServer** 字段包含数据库服务器的名称。则客户端检查 **InetLogin** 的网络参数。否则，客户端将从注册表的内存中的 GBASEDBTSERVER 值中确定缺省数据库服务器。然后从该数据库服务器的注册表发送网络参数值。

GBase 8s ESQL/C 检查应用程序当前设置的任何这些网络参数的 InetLogin 的网络参数字段。对于未设置的任何指定（ 包括缺省数据库服务器的名称），GBase 8s ESQL/C 从注册表的内存中复制获取值。

例如，以下代码段使用 **mainsrvr** 数据库服务器的信息来初始化 **InetLogin** 结构；**mainsrvr** 是缺省的数据库服务器：

```
void *cnctHndl;
        ⋮

        strcpy(InetLogin.InfxServer, "mainsrvr");
        strcpy(InetLogin.User, "finance");
        strcpy(InetLogin.Password, "in2money");
        EXEC SQL connect to 'accounts';
        ⋮
```

```
QL connect to 'custhist@bcksrvr';
```

当执行到达上述代码片段中的第一个 CONNECT 语句时，客户端应用程序请求与 **mainsrvr** 数据库服务器上的 **accounts** 数据库的连接。 CONNECT 语句不会知道数据库服务器，因此客户端会为缺省数据库服务器发送以下网络参数：

缺省数据库服务器是 mainsrvr，因为 InfxServer 在 InetLogin 中设置。

User 和 Password 值为 finance 和 in2money，因为应用程序在 InetLogin 中设置了它们。

Host 、Service 、Protocol 和 AskPassAtConnect 值来自注册表值的 mainsrvr 子项蚂蚁王应用程序没有在 InetLogin 中设置它们。

上述代码片段中的第二条 CONNECT 语句请求与数据库服务器 **bcksrvr** 上的求 **custhist** 数据库的连接。对于此连接，客户端发送指定数据库服务器 **bcksrvr** 的网络参数。因为 **InetLogin** 结构目前包含 **mainsrvr** 的网络参数，客户端必须从注册表的内存中副本获取*所有*这些参数。因此，应用程序不会使用 **finance** 用户账户进行第二次连接（除非注册表对 **bcksrvr** 数据库指定 finance 和 in2money 的 **User** 和 **Password** 值）。

如果您启用单点登录（SSO），则进程不同。有关配置的详细信息和其它步骤，请参阅为 SSO 配置 ESQL/C 和 ODBC 驱动程序。

中央注册表

可以在以下任一位置指定 sqlhosts 信息：

本地注册表是在与 GBase 8s ESQL/C 应用程序相同的 Windows(TM) 计算机上的注册表。

中央注册表是两个或多个 GBase 8s ESQL/C 应用程序可以访问以获取 sqlhosts 信息的注册表。

中央注册表可以在域服务器上或 Microsoft 网络上的任何 Windows(TM) 工作站中。它可能是一个应用程序的本地，并且远程到其它所有应用程序。中央注册表使您能够维护 sqlhosts 信息的单个副本，供 Windows(TM) 环境中的所有 GBase 8s ESQL/C 应用程序使用。

要使用中央注册表，必须在您的计算机上设置 GBASEDBTSQLHOSTS 环境变量。此环境变量指定中央注册表所在的计算机的名称。要设置此环境变量，可以使用 Setnet32 和 ifx_putenv() 函数（在 Windows 环境中设置和检索环境变量）或 InetLogin 结构（InetLogin 结构）。

在 Windows(TM) 环境中，GBase 8s ESQL/C 应用程序在请求连接时使用以下优先级来查找 sqlhosts 信息：

中央注册表中的 sqlhosts 信息，在 GBASEDBTSQLHOSTS 环境变量指示的计算机上（如果设置了 GBASEDBTSQLHOSTS ）

本地注册表中的 sqlhosts 信息

Windows(TM) 环境中的连接身份验证功能

GBase 8s ESQL/C 应用程序获取有关连接的信息（从注册表或 **InetLogin** 结）之后，ESQL 客户端-接口 DLL 执行以下步骤：

1. 它将来自 InetLogin 结构（或从未定义的 InetLogin 字段的注册表）的连接信息复制到 HostInfoStruct 结构中（参见 表 1）。

它将指向 **HostInfoStruct** 的指针传递给 esqlauth.dll 中的 sqlauth() 函数以验证连接身份。

如果 sqlauth() 返回 TRUE，则该连接有效并且用户可以访问服务器计算机。但是，如果 sqlauth() 返回 FALSE，则连接被拒绝，并且拒绝访问。缺省情况下，sqlauth() 函数返回 TRUE 值。

传递给 sqlauth() 的参数是一个指向 **HostInfoStruct** 结构的指针，它是 login.h 头文件定义的。此结构包含下表所示的 **InetLogin** 字段的子集。

表 1. HostInfoStruct 结构的字段

HostInfoStruct 字段	数据类型	意义
InfxServer	char[19]	指定 GBASEDBTSERVER 网络参数的值
Host	char[19]	指定 HOST 网络参数的值
User	char[19]	指定传递到 sqlauth() 函数的 USER 网络参数的值
Pass	char[19]	指定传递到 sqlauth() 函数的 PASSWORD 网络参数的值
AskPassAtConnect	char[2]	指示 sqlauth() 是否请求在连接时传入 sqlauth() 函数
Service	char[19]	指定传递到 sqlauth() 函数的 SERVICE 网络参数的值
Protocol	char[19]	指定传递到 sqlauth() 函数的 PROTOCOL 网络参数的值

HostInfoStruct 字段	数据类型	意义
Options	char[20]	保留备用

在 sqlauth() 中，可以使用 **pHostInfo** 指针访问 **HostInfoStruct** 的字段：

if (pHostInfo->AskPassAtConnect)

可以编辑所有的 **HostInfoStruct** 字段值。但是，ESQL/C 仅检查 **HostInfoStruct** 的 **User** 和 **Pass**字段。

以下代码片段显示了 esqlauth.c 文件包含的缺省的 sqlauth() 函。

```
BOOL __declspec( dllexport ) sqlauth ( HostInfoStruct *pHostInfo )
        {
        return TRUE;
        }
```

sqlauth() 的这种缺省操作意味着 GBase 8s ESQL/C 在建立连接时不进行身份验证。要提供验证，您可以自定义 sqlauth() 函数。您可能需要自定义 sqlauth() 来执行以下验证任务之一：

验证用户名称

该函数可以经当前用户名与有效或无效的用户名列表进行比较。

提示输入密码

当该字段设置为 Y 或 y 时，函数可以检查 **HostInfoStruct** 结构中的 **AskPassAtConnect** 字段的值。您可以编写 sqlauth() 来显示一个提示用户输入密码的窗口。

以下步骤描述如何创建自定义的 sqlauth() 函数：

在您的系统编辑器中打开 esqlauth.c 源文件。该文件位于 %GBASEDBTDIR%\demo\esqlauth 目录。

1. 将 sqlauth() 函数的主体添加到执行所需连接验证的代码中。在表 1中的字段中，sqlauth() 函数只能修改 User 和 Pass 字段。确保 sqlauth() 返回 TRUE 或 FALSE 以指示是否继续连接请求。不要修改此文件中的其他代码。

通过编译 esqlauth.c 文件并指定 esql 命令处理器的 -target:dll（或 -wd）命令行选项

来创建 esqlauth.dll 的版本。有关如何定义 sqlauth() 函数的示例，请参阅 %GBASEDBTDIR%\demo\esqlauth 目录中的 esqlauth.c 文件。

连接数据库服务器

当 GBase 8s ESQL/C 应用程序开始执行时，它与任何数据库服务器没有连接。但是，对于要执行的 SQL 语句，此连接必须存在。要建立与数据库服务器的连接，GBase 8s ESQL/C 程序必须采取以下操作：

使用 SQL 语句建立与数据库服务器的连接

在 SQL 语句中指定要连接的数据库服务器的名称

建立连接

以下两组 SQL 语句可以建立与数据库环境的连接：

SQL 连接语句为 CONNECT 、SET CONNECTION 和 DISCONNECT。这些语句符合用于创建连接的 ANSI SQL 和 X/Open 标准。

SQL 数据库语句包括 DATABASE 、CREATE DATABASE 、CLOSE DATABASE 和 START DATABASE。这些语句是建立特定于 GBase 8s 的连接的一种方式。

重要： 建议您对新的应用程序使用 CONNECT 、DISCONNECT 和 SET CONNECTION 连接语句。

应用程序建立的连接的类型取决于在应用程序中首先执行的语句的类型：

如果第一个 SQL 语句是连接语句（CONNECT 、SET CONNECT）语句，则应用程序建立显式连接。

如果第一个语句是 SQL 数据库语句（DATABASE 、CREATE DATABASE 、START DATABASE），则应用程序建立隐式连接。

显式连接

当使用 CONNECT 语句连接到数据库环境时，您建立一个显式连接。

应用程序直接连接到您指定的数据库服务器。如果未在 CONNECT 语句中指定数据库服务器的名称，则应用程序建立与缺省数据库服务器（GBASEDBTSERVER 环境变量 标识）的显式连接。

显式连接使应用程序能够支持到一个或多个数据库环境的多个连接。虽然应用程序在执行期间可以连接到多个数据库环境，但一次只能有一个连接。休眠连接是应用程序已建立但当前未使用的连接。应用程序必须具有当前连接才能执行 SQL 语句。以下 SQL 连接语句建立和管理显示连接。

以下 SQL 连接语句建立并管理显式连接：

CONNECT 语句建立数据库环境和应用程序之间的显示连接。

SET CONNECTION 语句在显式连接之间切换。它使当前连接处于休眠状态。

DISCONNECT 语句终止到数据库环境的连接。

这些连接语句提供以下好处，允许您创建更多便捷的应用程序：

符合 ANSI 和 X/Open 数据库连接标准

用于在分布式客户端-服务器的本地和远程数据访问的统一语法

支持单个应用程序中的多个连接

因为 CONNECT 、DISCONNECT 和 SET CONNECTION 语句包括 ANSI 标准语法的 GBase 8s 扩展，这些语句在以下时间会生成 ANSI 扩展警告消息：

运行时，如果设置了 DBANSIWARN 环境变量

编译时，如果已经使用 -ansi 预处理选项编译了 GBase 8s ESQL/C 源文件

GBase 8s ESQL/C 应用程序而不是数据库服务器处理这些连接语句。因此，应用程序不能在 PREPARE 或 EXECUTE IMMEDIATE 语句中使用它们。

重要： DATABASE 、CREATE DATABASE 、START DATABASE 、CLOSE DATABASE 和 DROP DATABASE 语句在显式连接时仍然有效。但是，在这种情况下，仅引用这些语句中当前连接本地的数据库；不要使用 **@server** 或 **//server** 语法。

隐式连接

当以下任一 SQL 语句是应用程序执行第一条 SQL 语句时，该语句建立隐式连接：

DATABASE 语句创建于数据库环境的隐式连接并打开指定的数据库。

CREATE DATABASE 用户创建隐式连接并创建一个数据库。

DROP DATABASE 语句创建隐式连接并删除（移除）指定的数据库。

上述语句的一个 PREPARE 语句也会建立隐式连接。

当您执行上述之一的语句时，应用程序首先连接缺省数据库服务器

（GBASEDBTSERVER 环境变量指示）。缺省数据库服务器解析数据库语句。如果语句指定了数据库服务器的名称，则应用程序连接到指定数据库服务器。要建立与指定数据库服务器的隐式连接，则应用程序必须连接两个数据库服务器。显式连接仅要求连接一个数据库服务器，因此它调用更少的资源。

如果隐式连接存在，则这些数据库语句在它们建立新的连接之前关闭它。新的隐式连接在 SQL 语句执行后仍保持打开状态。此行为与显式连接不同，显式连接允许与相同或不同的数据库环境建立多个连接。

CLOSE DATABASE 语句关闭数据库，并关闭与数据库的隐式连接。如果在这些语句之前使用 CONNECT 语句，每个语句也可以在当前显式连接的上下文中运行。

隐式连接为较早的应用程序提供了一个平滑的迁移路径，用于支持 CONNECT 、DISCONNECT 和 SET CONNECTION 语句的面向对象的连接。

连接类型的总结

下表总结了 GBase 8s ESQL/C 支持连接数据库服务器的方法。

表 2. 启动数据库服务器的语句和函数

SQL 语句或 ESQL/C 函数	隐式	显式	建立连接	打开数据库
如果程序中的第一条 SQL 语句是：				
DATABASE	是		是	是
CREATE DATABASE	是		是	是
START DATABASE	是		是	是
DROP DATABASE	是		是	
sqlstart()	是		是	
CONNECT TO DEFAULT		是	是	
CONNECT TO '*@servername*'		是	是	
CONNECT TO '*dbname*'		是	是	是
CONNECT TO 'dbname@servername'		是	是	是

在 Windows(TM) 环境中建立显式连接

使用隐式连接，每个 GBase 8s ESQL/C 模块都可以存在与数据库服务器的一个连接，并且无法共享此连接。显式连接允许客户端应用程序内的多个连接。您可能需要设计需要

执行多个连接的应用程序，原因如下：

当您想要多个 GBase 8s ESQL/C 模块（.exe 或 .dll）使用相同的连接来操作数据库数据时

图 1显示了多个应用程序使用与数据库服务器相同连接的方案。

当您想要一个 GBase 8s ESQL/C 命令创建两个或多个连接到一个或多个数据库时，其中包括在两个 C 应用程序之间共享一个 ESQL DLL

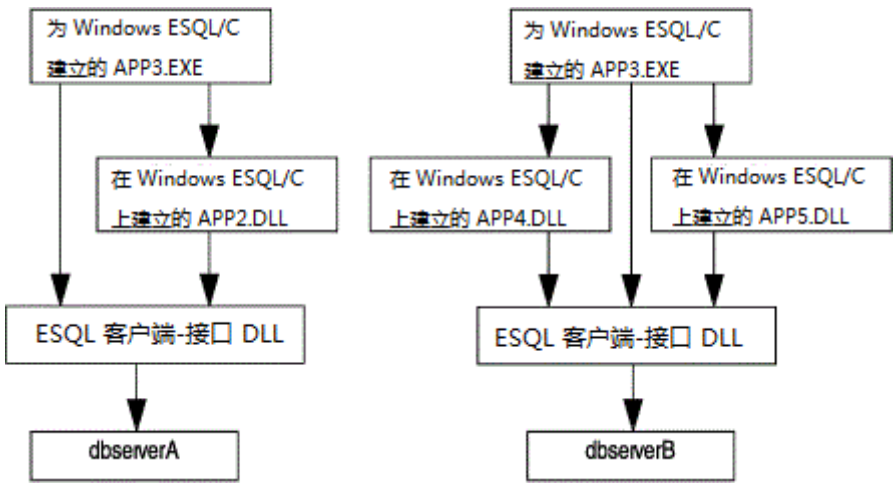
图 2显示了建立与多个数据库服务器的连接的单个应用程序。

图 1显示了以下两种场景，其中多个应用程序共享与数据库服务器的单一连接：

左侧的创建要求 APP1.EXE 建立与 dbserverA 数据库服务器的显式连接。此连接建立后，APP1 可以传递在 APP2 DLL 中设置连接所属的连接信息。

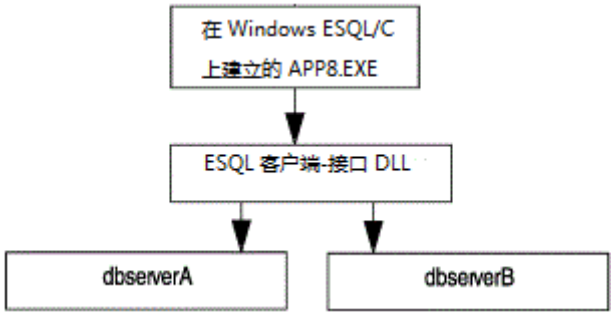
右侧的场景要求 APP3.EXE 建立与 dbserverB 数据库服务器的显式连接。当 APP3 通过适当的连接信息时，APP4 和 APP5 DLL 可以共享此连接。

图 3. 多个应用程序使用与数据库服务器的单一连接的两种场景



如果您想要一个应用程序在同一时间与两个不同的数据库服务器建立连接，则还可以使用显式连接，如下所示：

图 4. 一个应用程序在同一时间连接多个数据库服务器



密码加密

当客户端应用程序向数据库服务器发生密码进行身份验证时，密码不加密。除非您通过简单的密码通信支持模块（SPWDCSM）请求密码加密。您可以通过在 sqlhosts 信息中的数据库服务器激活密码加密，请在条目 的选项字段中指定以下值：

您可以通过在 sqlhosts 信息中的数据库服务器激活密码加密，请在 sqlhosts 条目的 Options 字段中指定以下值：

csm=(SPWDCSM)

客户端或数据库服务器使用字符串 "SPWDCSM" 作为密钥来查找在 CSS/CSM 配置文件中描述 CSM 的条目。

当激活时，SPWDCSM 要求密码，才能有效地覆盖任何受信任的主机机制。如果信任的主机策略到位，则指定密码加密机制是矛盾的。

可插拔认证模块（PAM）

要使用可插拔认证模块（PAM）服务进行客户端-服务器认证，您必须重启客户端应用程序，以便它注册一个回调函数。回调函数必须支持您打算使用 PAM 服务的任何挑战-响应机制。

演示程序 pamdemo.ec 作为回调函数示例。

LDAP 认证

可以使用 GBase 8s ESQL/C在 Windows(TM) 上使用轻量级命令访问协议（LDAP）身份验证。

当您要使用 LDAP 服务器验证系统用户时，请使用 LDAP 认证支持模块。该模块包含您可以根据具体情况参数修改的源代码。

多路复用连接

多路复用连接使 GBase 8s ESQL/C 应用程序可以通过使用最少量的通信资源来建立与同一数据库服务器上不同数据库的多个连接。当您自动多路复用连接时，数据库服务器将使用与客户端的单个连接进行多个 SQL 连接（CONNECT 语句）。没有复用，每个 SQL 连接创建一个数据库-服务器连接。

客户端执行的必要条件

要实现多路复用连接，请在客户端 sqlhosts 文件或注册表中，在要连接的数据库服务器的 **dbservername** 参数上设置多路复用选项。要指定多路复用。请将 m 选项设置为 1。以下 **dbservername** 参数指定与 **personnel** 数据库服务器的多路连接。

服务器名	Nettype	主机名	服务名	选项
personnel	onsoctcp	corp	prsnl_ol	m=1

将多路复用选项设置为零（m = 0）（缺省值），禁用指定数据库服务器的多路复用。

在 Windows(TM) 平台上，除了在 sqlhosts 注册表设置多路复用的选项之外，您还必须定义 **ifx_session_mux** 环境变量。如果未定义 **ifx_session_mux** 环境变量，则数据库服务器会忽略多路复用选项并不复用连接。

限制： 在 Windows(TM) 中，多线程应用程序不能使用多路复用连接功能。如果多线程应用程序启用了 sqlhosts 注册表中多路复用选项，并定义了 IFX_SESSION_MUX 环境变量，则可能会导致灾难性的后果，包括崩溃和数据损坏。

如果多线程应用程序和单线程应用程序在同一 Windows(TM) 计算机上运行，则单线程应用程序可以使用以下两种方法使用多路复用连接：

使用不同的 sqlhosts 信息。

在 sqlhosts 文件中使用不指定多路复用选项的 dbserver 别名。例如，可以使用以下配置：

服务器名	Nettype	主机名	服务名	选项
personnel	onsoctcp	corp	prsnl_01	m=1
personnel_nomux	onsoctcp	corp	prsnl_02	

连接到 *personnel* 服务器的任何多线程应用程序使用服务器名称 *personnel_nomux* ，

而单线程应用程序可以继续使用服务器名称 *personnel*。

多用复用连接的限制

GBase 8s ESQL/C 对多路复用连接实施以下限制：

不支持共享内存连接。

不支持多路复用应用程序。

数据库服务器忽略复用连接上的 sqlbreak() 函数，如果调用它，数据库服务器不会中断连接，并且不会返回错误。

标识数据库服务器

要连接数据库环境（例如，使用 CONNECT 语句），GBase 8s ESQL/C 应用程序可以以下两种方法标识数据库服务器：

应用程序可以在 SQL 语句中指定数据库服务器的名称。这类数据库服务器是特定的数据库服务器。

应用程序可以在 SQL 语句中省略数据库服务器的名称。这类数据库服务器是缺省数据库服务器。 GBASEDBTSERVER 环境变量执行缺省数据库服务器的名称。

特定的数据库服务器

GBase 8s ESQL/C 应用程序可以建立与指定数据库服务器的连接，只要它在 SQL 语句中列出数据库服务器名称和可选的数据库名，如下所示：

CONNECT 语句建立一个与数据库服务器的显示连接。

下列 CONNECT 语句建立与名为 **valley** 数据库服务器的显示连接：

```
EXEC SQL connect to 'stores7@valley';
        EXEC SQL connect to '@valley';
```

当应用程序的第一条语句为 SQL 数据库语句（如 DATABASE 或 START DATABASE）时，它可以建立一个隐式连接。

以下每一条 SQL 语句与名为 **valley** 特定数据库服务器中的 **stores7** 数据库的建立隐式连接：

```
EXEC SQL database '//valley/stores7';
        EXEC SQL database stores7@valley;
```

对于 UNIX(TM) 操作系统，使用下列语句：

EXEC SQL database '/usr/dbapps/stores7@valley';

对于 Windows(TM) 环境，使用下列语句：

EXEC SQL database 'C:\usr\dbapps\stores@valley';

缺省数据库服务器

GBase 8s ESQL/C 应用程序可以建立与缺省数据库服务器的连接，当它在 SQL 语句中省略来自数据库环境的数据库服务器名称。如下所示：

CONNECT 语句可以使用关键字 DEFAULT 或它忽略数据库服务器名称。

以下每条 CONNECT 语句都可以建立显式缺省连接：

EXEC SQL connect to 'stores7';

EXEC SQL connect to default;

在 UNIX(TM) 操作系统中，使用下列语句：

EXEC SQL connect to '/usr/dbapps/stores7';

在 Windows(TM) 环境中，使用下列语句：

EXEC SQL connect to 'C:\usr\dbapps\stores7';

当其中一个 SQL数据库语句（如 DATABASE 或 START DATABASE）是应用程序的第一个 SQL 语句时，它可以建立一个隐式的默认连接。

以下每条 SQL 语句在缺省数据库服务器上与名为 **stores7** 的数据库建立隐式连接：

EXEC SQL database stores7;

EXEC SQL start database stores7 with no log;

GBASEDBTSERVER 环境变量确定数据库服务器的名称。

重要： 即使应用程序未建立缺省连接，仍然需要设置 GBASEDBTSERVER 环境变量。

还可以使用 DBPATH 环境变量指定要作为缺省数据库服务器的数据库服务器名称的列表。应用程序在搜索 GBASEDBTSERVER 指定的数据库服务器后搜索这些数据库服务器。

3.2.3 与数据库服务器交互

在 GBase 8s ESQL/C 程序中，可以用以下方式与数据库服务器交互：

启动新的数据库服务器进程。当应用程序开始执行时该进程不存在。

在多个连接之间进行切换。应用程序可以建立多个连接。

标识显式连接。应用程序可以获得数据库服务器和连接的名称。

确定当前连接的数据库服务器可以访问的数据库。

检查数据库服务器检查的状态。对于某些操作，数据库服务器必须正忙，对于其它操作，数据库服务器必须空闲。

脱离当前连接。应用程序必须将子进程与当前进程连接。

中断数据库服务器进程。如果 SQL 请求执行了很长时间，则应用程序可以中断它。

终止数据库服务器进程。应用程序可以关闭未使用的连接以释放资源。

GBase 8s ESQL/C 支持安全套接字层（SSL）连接，有关 SSL 协议的信息，请参阅安全套接字层协议。

确定数据库服务器的功能

可以在执行以下 SQL 语句之后检查数据库服务器的功能。

CONNECT

CREATE DATABASE

DATABASE

SET CONNECTION

当数据库服务器使用这些语句建立连接时，它可以获取有关数据库服务器的下列信息：

是长标识符还是长用户名称被截断？

打开的数据库是否使用事务日志？

打开的数据库是否符合 ANSI ？

数据库服务器的名称为？

数据库是否以 DECIMAL 格式存储 FLOAT 数据类型（当主机系统缺少对 FLOAT 类型的支持时）？

数据库服务器是否处于辅助模式？（如果数据库服务器处于辅助模式，则它是数据复

制对中的辅助服务器，仅可用于读取操作）？

客户端应用程序设置的 DB_LOCALE 环境变量的值是否符合打开的数据库的数据库语言环境的值？下表总结了 **SQLSTATE** 变量和 **sqlca** 结构用于指示这些条件的值。

数据库功能	SQLSTATE 值	sqlca 值
长标识符或长用户名被截断	"01004"	sqlca.sqlwarn.sqlwarn1 是 'W'
数据库具有事务	"01I01"	sqlca.sqlwarn.sqlwarn1 是 'W'
数据库符合 ANSI	"01I03"	sqlca.sqlwarn.sqlwarn2 是 'W'
数据库服务器不是独立的产品	"01I04"	sqlca.sqlwarn.sqlwarn3 是 'W'
FLOAT 作为 DECIMAL 表示	"01I05"	sqlca.sqlwarn.sqlwarn4 是 'W'
数据库服务器在辅助模式	"01I06"	sqlca.sqlwarn.sqlwarn6 是 'W'
M 是符合的数据库语言环境	未定义	sqlca.sqlwarn.sqlwarn7 是 'W'

这些语句执行之后，SQLSTATE 变量可能返回多路复用异常。

在多个数据库连接中切换

GBase 8s ESQL/C 应用程序可以使用 CONNECT 语句进行多个并发的数据库连接。这些连接可以是多个数据库环境，也可以是同一数据库环境的多个连接。要在连接之间切换，GBase 8s ESQL/C 应用程序必须遵循以下步骤：

使用 CONNECT STATEMENT 建立连接

处理任何活动的事务

如果当前连接具有活动事务，则只有在 WITH CONCURRENT TRANSACTION 子句的 CONNECT 语句建立当前连接时，才能切换连接。

使用 SET CONNECTION 或 CONNECT 语句建立连接

建立当前连接

当存在多个连接时，应用程序一次只能与一个连接进行通信。此连接是当前连接。所有其他建立的连接都是休眠的。您的应用程序可以使其它连接与以下任一连接语句更新：

CONNECT 语句建立新的连接并使其成为当前连接。

SET CONNECTION 语句切换到休眠的连接并使其为当前连接。

当连接休眠然后再次连接时，您将执行类似于断开连接后重新连接到数据库环境的操作。但是，如果使用连接休眠，您通常可以避免数据库服务器再次执行身份验证，从而节省与连接相关联的资源的成本和使用。

提示： 线程安全 GBase 8s ESQL/C 应用程序具有多个当前连接。每个线程一个当前连接。但是，一次只有一个当前连接处于活动状态。

处理事务

如果具有 WITH CONCURRENT TRANSACTION 子句的 CONNECT 语句已经建立连接，则应用程序可以切换到另一个连接，即使当前连接包含活动的事务。

对于不是使用 CONNECT...WITH CONCURRENT TRANSACTION 语句建立的连接，应用程序必须在它切换到另一个连接之前结束活动的事务。任何在事务处于活动状态时切换的尝试都会导致 CONNECT 或 SET CONNECTION 语句失败（错误号 -1801）。当前连接中的事务仍处于活动状态。

要维持数据库信息的完整性，请通过以下方式之一显式终止活动事务：

使用 COMMIT WORK 语句提交事务来确保数据库服务器保存在事务中对数据库所做的任何更改。

使用 ROLLBACK WORK 语句回滚事务来确保数据库服务器备份在事务中对数据库所做的任何更改。

COMMIT WORK 或 ROLLBACK WORK 语句仅适用当前连接中的事务，而不适用于处于任何休眠连接的事务。

标识显式连接

在 GBase 8s ESQL/C 应用程序中，您可以使用 GET DIAGNOSTICS 语句获得数据库服务器的名称以及显式连接的名称。

当在 SQL 连接语句（CONNECT 、SET CONNECTION 和 DISCONNECT）之后使用 GET DIAGNOSTICS 时，GET DIAGNOSTICS 键数据库服务器的信息放置到 SERVER_NAME 和 CONNECTION_NAME 字段的诊断区域。

以下代码段保存 **srvrname** 和 **cnctname** 主机变量中的连接信息。

```
EXEC SQL connect to :dbname;
```

```
if(!strncmp(SQLSTATE, "00", 2)
{
EXEC SQL get diagnostics exception 1
:srvrname = SERVER_NAME, :cnctname = CONNECTION_NAME;
printf("The name of the server is '%s'\n", srvrname);
}
```

在 GBase 8s ESQL/C 应用程序中。可以使用 ifx_getcur_conn_name() 函数获得当前连接的名称。该函数将当前连接的名称返回到用户定义的字符缓冲区。该函数可用于在具有多个线程的 GBase 8s ESQL/C 应用程序中的一组活动的连接中确定当前连接。

例如，下列代码由一个 callback 函数 cb() 组成，它在两个不同的线程招工难使用两个 sqlbreakcallback() 调用：

```
void
cb(mint status)
{
mint res;
char *curr_conn = ifx_getcur_conn_name();

if (curr_conn && strcmp(curr_conn, "con2") == 0)
{
res = sqlbreak();
printf("Return status of sqlbreak(): %d\n", res);
}
}

void
thread_1()
{
EXEC SQL BEGIN DECLARE SECTION;
mint res;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL connect to 'db' as 'con1' ;
sqlbreakcallback(100, cb);
EXEC SQL SELECT count(*) INTO :res FROM x, y;
if (sqlca.sqlcode == -213)
printf("Connection con1 fired an sqlbreak().\n");
printf("con1: Result of count(*) = %d\n", res);
EXEC SQL set connection 'con1' dormant ;
}

void
thread_2()
{
EXEC SQL BEGIN DECLARE SECTION;
mint res;
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to 'db' as 'con2' ;
sqlbreakcallback(100, cb);
EXEC SQL SELECT count(*) INTO :res FROM x, y;
if (sqlca.sqlcode == -213)
printf("Connection con2 fired an sqlbreak().\n");
printf("con2: Result of count(*) = %d\n", res);
EXEC SQL set connection 'con2' dormant ;
}
```

cb() callback 函数使用 ifx_getcur_conn_name() 检查哪个是当前连接。

获取可用的数据库

从 GBase 8s ESQL/C 应用程序中，可以使用 sqgetdbs() 函数从指定的数据库服务器获取可用的数据库名称。该函数返回当前连接的数据库服务器中的可用的数据库的名称。

检查数据库服务器的状态

除非数据库服务器空闲，否则数据库服务器中的某些交互不会执行。其它操作假定数据库服务器正在忙于处理请求。可用使用 sqldone() 函数检查数据库服务器是否正在处理 SQL 请求。如果数据库服务器空闲，则该函数返回 0；如果数据库服务器忙， 则返回一

个负值。

为子进程建立一个单独的数据库连接

当您的应用程序推进一个进程时，子进程继承父进程的数据库连接。如果您将这些连接保持打开状态，则父和子进程都会使用同一连接与同一数据库服务器通信。 因此，子进程需要建立单独的数据库连接。

要为子进程建立一个单独的数据库连接：

调用 sqldetach() 将子进程从父进程中的数据库服务器连接脱离。

在子进程中建立新的连接（如果需要）。

中断 SQL 请求

要中断数据库服务器，可以使用 sqlbreak() 库函数。

有时您可能需要取消 SQL 语句。例如，如果不小心为长查询提供了错误的搜索条件。您希望取消 SELECT 语句而不是等待不需要的时间。当数据库服务器执行 SQL 请求时，GBase 8s ESQL/C 应用程序被阻断。要重新控制，应用程序必须中断 SQL 请求。

您可能出于以下原因中断 SQL 请求：

应用程序用户想要中断当前 SQL 请求。

当前 SQL 请求超出时间间隔。

重要： 应用程序必须在中断 SQL 请求后，处理任何打开的事务、游标和数据库。

中断 SQL 语句

您无法取消所有的 SQL 语句。某些类型的数据库操作不会被中断，其它的在特定的点也不会中断。GBase 8s ESQL/C 应用程序可以中断以下 SQL 语句：

ALTER INDEX

ALTER TABLE

CREATE INDEX

CREATE TABLE

EXECUTE FUNCTION

EXECUTE PROCEDURE

DELETE

INSERT

OPEN

SELECT

UPDATE

除了上述语句，您还可以取消循环的操作，如果它在 SPL 例程中执行。

GBase 8s ESQL/C 应用程序以及数据库服务器同*消息请求*通信。消息请求是发起 SQL 任务的消息的完整往返。它可以包括应用程序发送到数据库服务器以及数据库服务器回复的消息。或者，消息请求可以由数据库服务器向应用发送的信息以及应用程序回复的消息组成。

大多数 SQL 语句仅需要一个消息请求来执行。应用程序将此 SQL 语句发送到数据库服务器，数据服务器执行它。但是，传输大量数据（SELECT 、INSERT 或 PUT）SQL 语句可能需要多个消息请求执行，如下所示：

在第一个消息请求中，应用程序将 SQL 语句发送到数据库服务器执行。

在后续的消息请求中，数据库服务器使用数据填充缓冲区，然后将此数据发送到应用程序。缓冲区的大小决定了数据库服务器在单个消息请求中发送的数据量。

此外，OPEN 语句需要两个消息请求。

数据库服务器决定何时检查中断请求。因此，数据库服务器不可能立即终止 SQL 语句的执行，您的应用程序不可能就在它发送中断请求时就重新控制。

允许用户中断

当数据库服务器处理大数据的查询时，您可能想要用户使用中断键（通常为 CTRL-C）中断查询请求。

为此，必须设置一个*信号处理函数*。信号处理函数是用户定义的函数，当应用程序接收到特定的信号时应用程序进程会调用它。

要允许用户中断 SQL 请求，您要为此 SIGINT 信号定义信号处理函数。此函数必须具有以下声明：

```
void sigfunc_ptr();
```

用户定义的信号处理函数可以包含 GBase 8s ESQL/C 控制函数 sqlbreak() 和 sqldone()。如果在数据库服务器正在处理时使用任何其它 GBase 8s ESQL/C 控制函数或者信号处理程序中的任何 SQL 语句，GBase 8s ESQL/C 将生成一个错误（-439）。

GBase 8s ESQL/C 应用程序必须确定在信号处理完成后如何继续执行。一种可能的方法是使用 setjmp() 和 longjmp() 系统函数来设置非本地化。这些函数一起工作来支持低级中断，如下所示：

setjmp() 函数保存当前的执行环境，然后在 longjmp() 调用之后建立一个执行返回点。

longjmp() 调用在信号处理函数。只有在 sqldone() 返回 0（数据库服务器空闲）时，才能在信号处理函数中使用 longjmp()。

有关 setjmp() 和 longjmp() 系统函数的更多信息，请参阅 UNIX(TM) 操作系统文档。

要将用户定义的信号处理程序域系统信号相关联，请使用 signal() 系统函数，如下所示：

signal(SIGINT, *sigfunc_ptr*);

当 GBase 8s ESQL/C 应用程序接收了 SIGINT 信号，它调用 *sigfunc_ptr* 指示的函数。有关 signal() 系统函数的更多信息，请参阅您的 UNIX(TM) 操作系统文件。

要从信号取消信号处理函数的关联，调用使用 SIG_DFL 作为函数指针的 signal()，如下所示：

signal(SIGINT, SIG_DFL);

SIG_DFL 是缺省的信号处理操作。对于 SIGINT 信号，缺省操作时停止进程并生成核心转储。您可能需要指定 SIG_IGN 操作以使应用程序忽略该信号。

重要： 在大多数系统中，信号处理程序在应用程序捕获信号后保持有效。在这些系统上，如果您不想在下一次捕获系统的信号时执行，则需要将信号处理程序明确解除关联。

然而，在几个（大多数较旧的）系统上，当信号处理器捕获信号时，系统会将 SIG_DFL 操作恢复为处理机制。在这些系统上，如果您希望在下一次捕获信号时处理相同的信号，则信号处理程序将自动恢复。有关系统如何处理信号的信息，请查看系统文档。

设置超时间隔

当数据库服务器处理大量数据的查询时，您可能想要间隔地提示用户是否继续该请求。

要设置时间间隔，可以使用 sqlbreakcallback() 函数提供以下信息：

时间间隔是应用程序重新控制之前 SQL 请求等待执行的时间。

callback 函数是用户定义的函数，每次超时间隔过去都要调用。

限制： 如果您的 GBase 8s ESQL/C 应用程序使用共享内存（onipcshm）作为与数据库服务器实例色连接中的 NETTYPE，则不要使用 sqlbreakcallback()函数。共享内存不是真正的网络协议，并且不处理支持回调函数所需的非阻塞 I/O 。当您使用 sqlbreakcallback()与共享内存时，函数调用似乎成功注册回调函数 （它返回零），但是在 SQL 请求期间，应用程序永远不会调用回调函数。

超时间隔

使用 sqlbreakcallback() 函数，可以指定超时间隔。

超时间隔是数据库服务器在应用程序重新获得控制之前处理 SQL 请求的时间量（以毫秒为单位）。应用程序然后调用您指定回调函数并执行完成。

回调函数完成后，应用程序将恢复其等待，直到发生以下其中一个操作：

数据库服务器将以下条件一直的控制权返回给应用程序：

它已经完成了 SQL 请求。数据库服务器返回 SQLCODE 和 SQLSTATE 变量中请求的状态。

它语句停止处理 SQL 请求，因为它已经从回调函数中的 sqlbreak() 函数接收到中断请求。

超过下一个超时间隔。当应用程序恢复执行时，它再次调用回调函数。

每次超时时间间隔都会调用回调函数，直到数据库服务器完成请求或中断。

回调函数

使用 sqlbreakcallback() 函数，您还可以指定在执行 SQL 请求时在多个点调用回调函数。

回调函数是用户定义的 GBase 8s ESQL/C 函数，它在 SQL 语句执行期间指定要采取的操作。该函数必须具有以下声明：

```
void callbackfunc(status)
        mint status;
```

integer*status* 变量标识在调用回调函数的 SQL 请求执行的时间点。在回调函数中，您可以检查此状态变量，以确定该函数在哪个是假被调用。下表总结了有效的 *status* 值。

表 3. 回调函数的状态值

调用回调的时间点	回调参数值
在数据库服务器语句完成 SQL 请求之后	0
紧随着应用程序将 SQL 请求发送到数据库服务器后	1
当数据库服务器正在处理 SQL 请求时，超时间隔已过	2

在此调用函数中，您可能想要检查 *status* 参数的值来确定函数要采取的操作。

提示： 当您使用 sqlbreakcallback() 注册回调函数时，应用程序每次调用回调函数发送消息请求。因此，需要多个消息请求的 SQL 语句会导致应用程序调用回调函数多次。

回调函数，以及它的子例程，只能获得以下 GBase 8s ESQL/C 控制函数：

sqldone() 库函数确定数据库服务器是否仍在忙。

如果 sqldone() 返回错误 -439，则数据库服务器仍在忙并且您可以继续执行中断。

sqlbreakcallback() 库函数将回调函数从超时间隔中解除关联。

使用以下参数调用 sqlbreakcallback()：

```
sqlbreakcallback(-1L, (void *)NULL);
```

如果您希望在当前连接期间保留回调函数，则不需要此步骤。关闭当前连接时，还可以取消关联回调函数。

sqlbreak() 库函数中断数据库服务器的执行。

如果使用任何 GBase 8s ESQL/C 控制函数而不是上述列表中的函数，或者如果您使用任何数据库服务器正在处理的 SQL 语句，则 GBase 8s ESQL/C 生成错误（-439）。

如果应用程序调用回调函数，因为超时间隔已过，此函数可以提示用户是否继续或取消 SQL 请求，如下所示：

要继续执行 SQL 请求，则回调函数略过对 sqlbreak() 的调用。

当回调函数执行时，数据库服务器继续忽略它的 SQL 请求。在回调函数执行完毕后，应用程序在再次调用回调函数之前等待另一个超时间隔。在此期间，数据库服务器继续执行 SQL 请求。

要取消此 SQL 请求，回调函数调用 sqlbreak() 函数，它将中断请求发送给数据库服务器。

回调函数在 sqlbreak() 发送请求后立即继续执行。该应用程序不会等待数据库服务器响应，指定它完成回调函数的执行。

当数据库服务器接收到中断请求信号时，它确定是否中断当前 SQL 请求。如果中断，数据库服务器不会继续处理并将控制返回给应用程序。此应用程序负责优化程序的终止；它必须释放资源并回滚当前的事务。

使用 sqlbreakcallback() 函数设置超时间隔（以毫秒为单位），然后注册回调函数，如下所示：

```
sqlbreakcallback(timeout, callbackfunc_ptr);
```

callbackfunc_ptr 必须指向您已经定义的回调函数。在调用程序中，您必须声明此函数，如下所示：

```
void callbackfunc_ptr();
```

重要： 在建立连接后，执行嵌入的您想要取消的 SQL 语句之前，必须注册回调函数。在您关闭函数后，回调函数不再注册。

数据传输期间的错误检查

IFX_LOB_XFERSIZE 环境变量用于指定在检查是否发生错误之前，要从客户端应用程序传输到数据库服务器的 CLOB 或 BLOB 中的千字节数。每次传输指定千字节的数据时都会发生错误。如果发生错误，不会发送剩余数据并报告错误。如果没有发送错误，会继续传输文件直到结束。

IFX_LOB_XFERSIZE 的有效服务为 1 - 9223372036854775808 KB。IFX_LOB_XFERSIZE 环境变量在客户端上设置。

终止连接

GBase 8s ESQL/C 程序可以是以以下语句和函数关闭连接：

CLOSE DATABASE 语句关闭数据库。 CLOSE DATABASE 语句执行后，连接仍然保持打开状态。

sqlexit() 库函数关闭所有当前连接，显式和隐式。当任何数据库仍旧打开时，如果调用 sqlexit() ，则该函数导致任何打开的事务回滚。

sqldetach() 库函数关闭子进程的数据库连接。它不会影响父进程的数据库服务器连接。

DISCONNECT 语句关闭指定的连接。如果数据库是打开的，则 DISCONNECT 语句在它关闭连接前关闭数据库。如果事务是打开的，则 DISCONNECT 语句失败。

3.2.4 优化消息传输

GBase 8s ESQL/C 提供名为优化消息传输的功能，它可以让大多数 GBase 8s ESQL/C 语句使数据库服务器的信息传输最小化。

GBase 8s ESQL/C 通过将消息链接在一起，甚至消除一些小的消息数据包来实现优化的消息传输。当优化的消息传输功能被启用时，GBase 8s ESQL/C 期望 SQL 语句将会成功。因此，GBase 8s ESQL/C 在某些情况下消除了数据库服务器的确认消息。

当启用 OPTMSG 功能时，您的 GBase 8s ESQL/C 应用程序无法对任何链接的语句执行错误处理。如果您不确定特定语句是否可能生成错误，请包括错误处理代码，并且不要为该语句启用消息链接。

优化消息传输的限制

GBase 8s ESQL/C 不会链接以下 SQL 语句，即使您启用了优化消息传输：

COMMIT WORK

DESCRIBE

EXECUTE

FETCH

FLUSH

PREPARE

PUT

ROLLBACK WORK

SELECT INTO (singleton SELECT)

当 GBase 8s ESQL/C 到达以上任一语句时，它将消息刷新到数据库服务器。GBase 8s ESQL/C 然后继续后续的 SQL 语句的消息链接。只有需要网络流量的 SQL 语句才能使 GBase 8s ESQL/C 刷新消息。

不需要网络流量的 SQL 语句（如 DECLARE 语句）不会导致 GBase 8s ESQL/C 将消息队列发送到数据库服务器。

启用优化消息传输

要启用优化消息传输，或消息链接，您必须在客户端环境中设置以下变量：

在运行时设置 OPTMSG 环境变量，以使所有的合格 SQL 语句启用优化消息传输。

在 GBase 8s ESQL/C 应用程序中设置 **OptMsg** 全局变量，以控制哪些 SQL 语句使

用消息链接。

设置 OPTMSG 环境变量

OPTMSG 环境变量为应用程序中的所有 SQL 语句启用优化消息传输。

可以将以下值分配给 OPTMSG 环境变量：

1

该值启用优化消息传输， 实现后续任何连接的功能。

0

该值禁用优化消息传输。（缺省）

OPTMSG 环境变量的缺省值为 0。将 OPTMSG 设置为 0 来将禁用显示消息。您可能需要禁用针对需要即时回复或用于调试目的的语句的优化消息传输。

要启用优化消息传输，您必须在启动 GBase 8s ESQL/C 应用程序前设置 OPTMSG 。

在 UNIX(TM) 操作系统中，可以使用 putenv() 系统调用在应用程序中设置 OPTMSG（只要您的系统支持 putenv() 函数）。例如，以下对 putenv() 的调用，启用优化消息传输：

```
putenv("OPTMSG=1");
```

在 Windows(TM) 环境中，可以使用 ifx_putenv() 函数在应用程序中设置 OPTMSG。例如，以下 ifx_putenv()调用，启用优化消息传输：

```
ifx_putenv("OPTMSG=1");
```

当您在应用程序中设置 OPTMSG 时，可以激活或停用每个连接或每个线程中优化的消息传输。要启用优化消息传输，必须在建立连接前设置 OPTMSG。

设置 OptMsg 全局变量

OptMsg 全局变量在 GBase 8s ESQL/Csqlhdr.h 头文件中定义。

当将 OPTMSG 环境变量设置为 1 后，您必须设置 **OptMsg** 全局变量来指定消息链接是否对每个后续 SQL 语句生效。您可以将以下值分配给 **OptMsg**：

1

该值对每一条后续 SQL 语句启用消息链接。

0

该值对每一条后续 SQL 语句禁用消息链接。

将 OPTMSG 环境变量设置为 1，您仍然必须将 **OptMsg** 全局变量设置为 1 来启用消息链接。如果您从程序中忽略以下语句，则 GBase 8s ESQL/C 不会执行消息链接：

OptMsg = 1;

当您已经将 OPTMSG 环境变量设置为 1 时，您可能会出于以下原因禁用链接：

某些 SQL 语句需要立即回复。

限制 SQL 语句完成后重新启用 OPTMSG 功能。

出于调试目的

当您尝试确定每个 SQL 语句的响应方式时，可以禁用 OPTMSG 功能。

在程序中的最后一个 SQL 语句执行之前，确保数据库服务器在应用程序退出之前处理所有消息。如果启用 OPTMSG，则该消息将排队等待数据库服务器，但不会发送进行处理。

要避免意外链接，请在需要的 SQL 语句之后立即重置 **OptMsg** 全局变量。以下代码片段为 DELETE 语句启用消息链接：

```
OptMsg = 1;
        EXEC SQL delete from customer;
        OptMsg = 0;
        EXEC SQL create index ix1 on customer (zipcode);
```

此示例启用消息链接因为 DELETE 语句的执行不可能失效。因此，它可以安全地链接到下一个 SQL 语句。GBase 8s ESQL/C 延迟发送 DELETE 语句的信息。该示例在 DELETE 语句之后禁用消息链接，以使 GBase 8s ESQL/C 刷新下一个 SQL 语句执行时已排队的所有消息。通过禁用 DELETE 之后的消息链接，代码片段避免了意外的消息链接。当发生意外的链接时，可能很难确定哪些链接的语句失败。

在 CREATE INDEX 语句中，GBase 8s ESQL/C 将 DELETE 和 CREATE INDEX 语句发送给数据库服务器。

使用优化消息传输处理错误

当启用 OPTMSG 功能时，您的 GBase 8s ESQL/C 应用程序无法对任何链接的语句执行错误处理。如果您不确定特定语句是否可能生成错误，请包括错误处理代码，并且不要为该语句启用消息链接。

当在链接的语句中发生错误时，数据库服务器停止执行。此错误之后的任何 SQL 语句都不会执行。例如，以下代码旨在链接五个 INSERT 语句（该片段假设 OPTMSG 环境变量设置为 1）：

```
EXEC SQL create table tab1 (col1 INTEGER);

        /* enable message chaining */
        OptMsg = 1;

        /* these two INSERT statements execute successfully */
        EXEC SQL insert into tab1 values (1);
        EXEC SQL insert into tab1 values (2);

        /* this INSERT statement generates an error because the data
        * in the VALUES clause is not compatible with the column type */
        EXEC SQL insert into tab1 values ('a');

        /* these two INSERT statements never execute */
        EXEC SQL insert into tab1 values (3);
        EXEC SQL insert into tab1 values (4);

        /* disable message chaining */
        OptMsg = 0;

        /* update one of the tab1 rows */
        EXEC SQL update tab1 set col1 = 5 where col1 = 2;
        if ( SQLCODE < 0 )
        ?
```

在此代码片段中，GBase 8s ESQL/C 在到达 UPDATE 语句时刷新消息队列。将五个 INSERT 语句和 UPDATE 语句发送到数据库服务器执行。因为第三个 INSERT 语句发生错误，所以数据库服务器不执行剩余的 INSERT 语句和 UPDATE 语句。UPDATE 语句（它是链接语句中的最后一个语句）从失败的 INSERT 语句返回错误。**tab1** 表包含 **col1** 值为 1 和 2 的行。

3.2.5 数据库服务器控制函数

下表描述了您可以用于控制数据库服务器会话的 GBase 8s ESQL/C 库函数。

函数名称	描述	请参阅
ifx_getcur_conn_name()	返回当前连接的名称。	ifx_getcur_conn_name() 函数
sqgetdbs()	返回数据库服务器可以访问的数据库名称。	sqgetdbs() 函数
sqlbreak()	向数据库服务器发送一个停止处理的请求。	sqlbreak() 函数
sqlbreakcallback()	建立超时间隔和回调函数来中断 SQL 请求。	sqlbreak() 函数
sqldetach()	将子进程从数据库服务器连接脱离。	sqldetach() 函数
sqldone()	确定数据库服务器当前是否正在处理 SQL 请求。	sqldone() 函数
sqlexit()	终止数据库服务器连接。	sqlexit() 函数
sqlsignal()	执行信号处理并清除子进程。	sqlsignal() 函数
sqlstart()	启动数据库服务器连接。	sqlstart() 函数

3.2.6 超时程序

timeout 程序演示如何设置超时间隔。

此程序使用 sqlbreakcallback() 函数执行以下操作：

指定执行 SQL 请求的超时间隔为 200 毫秒 。

要注册在 SQL 请求开始和结束时以及当超时间隔过去时要调用的 on_timeout() 回调函数。

如果 SQL 请求的执行超出了超时间隔，则回调函数使用 sqldone() 函数确保数据库

服务器仍然在忙，提示用户确认中断，然后使用 sqlbreak() 函数发送中断请求到数据库服务器。

编译程序

使用以下命令编译 **timeout** 程序：

esql -o timeout timeout.ec

-o timeout 选项将可执行程序命名为 **timeout**。不用 **-o** 选项，可执行程序的名称缺省为 **a.out**。

timeout.ec 文件指南

```
================================================================================
1. /*
2.  * timeout.ec *
3.  */
4. #include <stdio.h>
5. #include <string.h>
6. #include <ctype.h>
7. #include <decimal.h>
8. #include <errno.h>
9. EXEC SQL include sqltypes;
10. #define LCASE(c) (isupper(c) ? tolower(c) : (c))
11. /* Defines for callback mechanism */
12. #define DB_TIMEOUT        200     /* number of milliseconds in timeout */
13. #define SQL_INTERRUPT -213     /* SQLCODE value for interrupted stmt
      */
14. /* These constants are used for the canceltst table, created by
15.  * this program.
16.  */
17. #define MAX_ROWS            10000   /* number of rows added to table */
18. EXEC SQL define CHARFLDSIZE   20; /* size of character columns in
                                        * table */
19. /* Define for sqldone() return values */
20. #define SERVER_BUSY     -439
```

```
21. /* These constants used by the exp_chk2() function to determine
22.    * whether to display warnings.
23.    */
24. #define WARNNOTIFY              1
25. #define NOWARNNOTIFY            0
26. int4 dspquery();
27. extern int4 exp_chk2();
28. void on_timeout();
29. main()
30. {
31.         char ques[80], prompt_ans();
32.         int4 ret;
33.         mint create_tbl(), drop_tbl();
34.         printf("TIMEOUT Sample ESQL Program running.\n\n");
35.         /*
36.          * Establish an explicit connection to the stores7 database
37.          * on the default database server.
38.          */
39.         EXEC SQL connect to 'stores7';
```

==========

第 4 - 9 行

行 4 - 8 包括来自 /usr/include 目录的 UNIX(TM) 头文件。GBase 8s ESQL/Csqltypes.h 头文件（第 9 行）定义标识 SQL 和 C 的数据类型的整数值。

第 10 - 20 行

第 10 行定义 LCASE，是一个将大写字符转换为小写字符的宏。DB_TIMEOUT（第 12 行）常量定义超时间隔中的毫秒数。 SQL_INTERRUPT 常量（第 13 行）定义数据库服务器中断 SQL 语句时返回的 SQLCODE 值。

第 17 和 18 行定义 create_tbl() 函数用于创建 **canceltst** 表的常量。此表保存大查询所需测试数据（第 125 - 132）。MAX_ROWS 时 create_tbl() 插入到 **canceltst** 中的行数。如果发现查询运行时间不足以阻止该查询，您可以更改此数。CHARFLDSIZE 是 **canceltst** 的字符字段（**char_fld1** 和 **char_fld2**）中的字符数。

第 20 行定义 SERVER_BUSY 常量保存 sqldone() 返回值，来指示数据库服务器是否正在忙于处理 SQL 请求。使用此常量使代码更可读，并且移除代码显式返回值。

第 24 和 25 行

exp_chk2() 异常处理函数使用 WARNNOTIFY 和 NOWARNNOTIFY 常量（第 24 和 25 行）。调用 exp_chk2() 指定其中的一个作为第二个参数，以指示该函数是否显示警告信（WARNNOTIFY）的 SQLSTATE 和 SQLCODE 信息，或者不显示警告信息（NOWARNNOTIFY）。

第 29 - 33 行

main() 程序从第 29 行开始。第 31 - 33 行声明 main() 程序块的本地变量。

```
40.     if (exp_chk2("CONNECT to stores7", NOWARNNOTIFY) < 0)
41.         exit(1);
42.     printf("Connected to 'stores7' on default server\n");
43.     /*
44.      * Create the canceltst table to hold MAX_ROWS (10,000) rows.
45.      */
46.     if (!create_tbl())
47.         {
48.         printf("\nTIMEOUT Sample Program over.\n\n");
49.         exit(1);
50.         }
```

```
51.     while(1)
52.         {
53.         /*
54.           * Establish on_timeout() as callback function. The callback
55.           * function is called with an argument value of 2 when the
56.           * database server has executed a single SQL request for number
57.           * of milliseconds specified by the DB_TIMEOUT constant
58.           * (0.00333333 minutes by default). Call to    sqlbreakcallback()
59.           * must come after server connection is established and before
60.           * the first SQL statement that can be interrupted.
61.           */
62.         if (sqlbreakcallback(DB_TIMEOUT, on_timeout))
63.             {
64.             printf("\nUnable to establish callback function.\n");
65.             printf("TIMEOUT Sample Program over.\n\n");
66.             exit(1);
67.               }
68.         /*
69.           * Notify end user of timeout interval.
70.           */
71.         printf("Timeout interval for SQL requests is: ");
72.        printf("%0.8f minutes\n", DB_TIMEOUT/60000.00);
73.           stcopy("Are you ready to begin execution of the query?",
74.             ques);
75.         if (prompt_ans(ques) == 'n')
76.             {
77.             /*
78.               * Unregister callback function so table cleanup will not
79.               * be interrupted.
80.               */
81.             sqlbreakcallback(-1L, (void *)NULL);
82.             break;
83.             }
```

==========

第 43 - 50 行

create_tbl() 函数在数据库 **stores7** 中创建 **canceltst** 表。它插入 MAX_ROWS 数行。如果 create_tbl() 在创建 **canceltst** 时遇到错误，则 **timeout** 程序不能继续执行。该程序具有状态值 1（第49 行）。

第 51 行

while 循环（在第 97 行结束），控制 **canceltst** 表上的查询。它允许用户多次运行此查询以测试多种中断场景。

第 53 - 67 行

while 循环的第一要务是使用 sqlbreakcallback() 指定DB_TIMEOUT (200) 的超时间隔毫秒，并将 on_timeout() 注册为回调函数。如果 sqlbreakcallback() 调用失败，则程序具有状态值 1。要测试不同的超时间隔，可以更改 DB_TIMEOUT 处理并重新编译 timeout.ec 源文件。

第 68 - 72 行

这些 printf() 函数通知用户超时间隔。请注意，消息以分钟显示此间隔，而不是毫秒。它将 DB_TIMEOUT 值除以 60,000 （一分钟的毫秒数）。

第 73 - 83 行

prompt_ans() 函数要求用户指示何时开始执行 canceltst 查询执行。如果用户输入 n （否），则程序调用 sqlbreakcallback() 函数取消注册回调函数。

此调用阻止 drop_tbl() 函数（第 322-329 行）中的 SQL 语句启动调用函数。有关 prompt_ans() 函数的描述，请参阅第 337 - 347 行。

```
==========================================================================
84.          /*
85.           * Start display of query output
86.           */
87.          printf("\nBeginning execution of query...\n\n");
88.          if ((ret = dspquery()) == 0)
89.              {
90.              if (prompt_ans("Try another run?") == 'y')
91.                  continue;
92.              else
93.                  break;
94.              }
95.          else    /* dspquery() encountered an error */
96.              exit(1);
97.          } /* end while */
98.      /*
99.       * Drop the table created for this program
100.      */
101.     drop_tbl();
102.     EXEC SQL disconnect current;
103.     if (exp_chk2("DISCONNECT for stores7", WARNNOTIFY) != 0)
104.         exit(1);
105.     printf("\nDisconnected stores7 connection\n");
106.     printf("\nTIMEOUT Sample Program over.\n\n");
107. }
108. /* This function performs the query on the canceltst table. */
109. int4 dspquery()
110. {
111.     mint cnt = 0;
```

```
112.    int4 ret = 0;
113.    int4 sqlcode = 0;
114.    int4 sqlerr_code, sqlstate_err();
115.    void disp_exception(), disp_error(), disp_warning();
116.    EXEC SQL BEGIN DECLARE SECTION;
117.        char fld1_val[ CHARFLDSIZE + 1 ];
118.        char fld2_val[ CHARFLDSIZE + 1 ];
119.        int4 int_val;
120.    EXEC SQL END DECLARE SECTION;
121.    /* This query contains an artificially complex WHERE clause to
122.     * keep the database server busy long enough for an interrupt
123.     * to occur.
124.     */
125.    EXEC SQL declare cancel_curs cursor for
126.        select sum(int_fld), char_fld1, char_fld2
127.        from canceltst
128.        where char_fld1 matches "*f*"
129.            or char_fld1 matches "*h*"
130.            or char_fld2 matches "*w*"
131.            or char_fld2 matches "*l*"
132.        group by char_fld1, char_fld2;
```

==========

第 84 - 97 行

如果用户选择继续查询，则此程序调用 dspquery() 函数（第 88 行）运行 **canceltst** 查询，prompt_ans() 函数显示提示，以便用户可以决定是否再次运行该程序。

第 98 - 101 行

drop_tbl() 函数从 **stores7** 数据库删除 **canceltst** 表，以便程序清理。

第 108 - 120 行

dspquery() 函数返回 **canceltst** 表的查询并显示结果。它返回零（成功）或者 SQLCODE 负值（失败）来指示 **canceltst** 查询的结果。

第 121 - 132 行

第 125 行为此游标声明 **cancel_curs** 游标。实际 SELECT （第 126 - 132 行）包含 **int_fld** 列和两个字符列（**char_fld1** 和 **char_fld2**）的值的总和。WHERE 子句使用 MATCHES 运算符来指定匹配的行，如下所示： follows:

包含 f 或 h 的所有的 **char_fld1** 列都包含以下条件：

char_fld1 matches "*f*"

or char_fld1 matches "*h*"

这些条件与 Gbasedbt 或"4100 Bohannon Dr." 的 **char_fld1** 值匹配。

包含 w 或 l 的所有的 char_fld2 列都包含以下条件：

char_fl2 matches "*w*"

or char_fld2 matches "*l*"

这些条件与 Software 或 "Menlo Park, CA" 的 **char_fld2** 值相匹配。

此 SELECT 是人为复制的，以确保查询需要很长时间才能执行。没有相当复杂的查询，数据库服务器在用户有机会中断之前完成执行。在生产应用程序中，只能使用 sqlbreakcallback() 功能与需要很长时间执行的查询。

```
============================================================
======
    EXEC SQL open cancel_curs;
    sqlcode = SQLCODE;
    sqlerr_code = sqlstate_err(); /* check SQLSTATE for exception */
```

```
        if (sqlerr_code != 0)              /* if exception found */
            {
            if (sqlerr_code == -1)         /* runtime error encountered */
                {
                if (sqlcode == SQL_INTERRUPT)    /* user interrupt */
                    {
                    /* This is where you would clean up resources */
                    printf("\n            TIMEOUT INTERRUPT PROCESSED\n\n");
                    sqlcode = 0;
                    }
                else                                /* serious runtime error */
                    disp_error("OPEN cancel_curs");
                EXEC SQL close cancel_curs;
                EXEC SQL free cancel_curs;
                return(sqlcode);
                }
            else if (sqlerr_code == 1)          /* warning encountered */
                disp_warning("OPEN cancel_curs");
             }
    ================================================================
======
```

第 133 行

OPEN 函数使数据库服务器执行与 **cancel_curs** 游标关联的 SELECT 。因为数据库服务器执行 **canceltst** 查询，此 OPEN 是用户最优可能中断的语句。当 FETCH 执行时，数据库服务器仅发送匹配的行到应用程序，这种操作通常并不耗时。

第 134 - 154 行

此代码检查 OPEN 的成功。因为 OPEN 可以被中断，所以此异常检查必须包含中断值为 -213 的显式检查。数据库服务器在中断一个 SQL 请求时将 SQLCODE 设置为 -213。在第 140 行，程序为 SQLCODE 值使用 SQL_INTERRUPT 定义常量（第 13 行定义）。

sqlstate_err() 函数（第 135 行）使用 GET DIAGNOSTICS 语句分析 SQLSTATE 变量的值。如果此函数返回非零值，则 SQLSTATE 指示警告，运行错误或 NOT FOUND 条件。在调用 sqlstate_err() 之前，第 134 行保存 SQLCODE 值，以便任何其它执行的 SQL 语句（如 sqlstate_err() 中的 GET DIAGNOSTICS）不会重写它。如果 OPEN 语句遇到运行错误该函数返回 SQLCODE 的值（第 150 行）

如果 OPEN 遇到任何种类的异常（sqlstate_err() 返回非零值），则第一个 **if** 语句（136 行）检查。如果 OPEN 已经生成了运行错误（返回值为 -1），则第二个 **if**（138 行）检查。但是，如果数据库服务器打断了 OPEN，则 sqlstate_err() 也会返回 -1。因为 GBase 8s ESQL/C 不会将打断的 SQL 语句作为运行错误处理。所以第三个 **if** 显式检查 SQL_INTERRUPT 值（140 行）。如果 OPEN 被打断，则 143 行通知用户打断的请求成功然后还是重新将保存的 SQLCODE 值（**sqlcode** 中）设置为零，以指示 OPEN 没有生成运行错误。

第 146 和 147 行只在 OPEN 生成运行错误时执行，而不是在 SQL_INTERRUPT（-213）时。disp_error() 函数显式诊断区域中的异常信息和 SQLCODE 值。第 148 - 150 行在 OPEN 后清除。它们关闭并释放 **cancel_curs** 游标，然后返回 SQLCODE 值。如果 OPEN 被中断，则 dspquery() 函数不会使用 FETCH （158 行）继续。

如果 sqlstate_err() 返回一（1），则 OPEN 生成警告。第 152 和 153 行调用 disp_warning() 函数显示来自诊断区域的警告信息。有关 disp_error() 和 disp_warning() 函数的更多信息，请参阅 第 348 - 355 行。

```
==============================================================================
155.    printf("Displaying data...\n");
156.    while(1)
157.        {
158.        EXEC SQL fetch cancel_curs into :int_val, :fld1_val,
        :fld2_val;
159.        if ((ret = exp_chk2("FETCH from cancel_curs", NOWARNNOTIFY))
        == 0)
160.            {
161.            printf("    sum(int_fld) = %d\n", int_val);
```

```
162.                printf("      char_fld1 = %s\n", fld1_val);
163.                printf("      char_fld2 = %s\n\n", fld2_val);
164.                }
165.           /*
166.             * Will display warning messages (WARNNOTIFY) but continue
167.             * execution when they occur (exp_chk2() == 1)
168.             */
169.           else
170.               {
171.               if (ret==100)                    /* NOT FOUND condition */
172.                   {
173.                   printf("\nNumber of rows found: %d\n\n", cnt);
174.                   break;
175.                   }
176.               if (ret < 0)                     /* Runtime error */
177.                   {
178.                   EXEC SQL close cancel_curs;
179.                   EXEC SQL free cancel_curs;
180.                   return(ret);
181 .                 }
182.               }
183.           cnt++;
184.           } /* end while */
185.       EXEC SQL close cancel_curs;
186.       EXEC SQL free cancel_curs;
187.       return(0);
188. }
189. /*
190.    *    The on_timeout() function is the callback function. If the user
191.    *    confirms the cancellation, this function uses sqlbreak() to
192.    *    send an interrupt request to the database server.
193.    */
194.   void on_timeout(when_called)
```

```
195.    mint when_called;
196.    {
197.    mint ret;
198.    static intr_sent;
```

==========

第 155 - 182 行

while 循环对 **cancel_curs** 游标包含的每一行执行。 FETCH 语句（第 158 行） 从 **cancel_curs** 游标检索一行。如果 FETCH 生成错误，则函数释放游标资源并返回 SQLCODE 错误值（第 176 - 181 行）。否则，此函数向用户显示检索到的数据。在最后一行（**ret** = 100），该函数显示它检索到的行数（173 行）。

第 185 - 187 行

在 FETCH 从游标检索到最后一行后，该函数将释放分配给 **cancel_curs** 游标的资源，并返回成功值为零。

第 190 - 198 行

on_timeout() 函数是 **timeout** 程序的回调函数。62 行调用 sqlbreakcallback() 注册此函数并建立一个 200 毫秒的超时间隔。每次在数据库服务器开始或结束 SQL 请求时调用此函数。对于长时间运行请求，应用程序在每次超时时间间隔调用调用 on_timeout()。

==========

```
199.        /* Determine when callback function has been called.    */
200.        switch(when_called)
201.            {
202.            case 0: /* Request to server completed */
```

```
203.            printf("+------SQL Request ends");
204.           printf("-----------------------------+\n\n");
205.            /*
206.             * Unregister callback function so no further SQL statements
207.             * can be interrupted.
208.             */
209.            if (intr_sent)
210.                sqlbreakcallback(-1L, (void *)NULL);
211.            break;
212.        case 1: /* Request to server begins */
213.            printf("+------SQL Request begins");
214.            printf("-----------------------------+\n");
215.            printf("|                                    ");
216.            printf("                                  |\n");
217.            intr_sent = 0;
218.            break;
219.        case 2: /* Timeout interval has expired */
220.            /*
221.             * Is the database server still processing the request?
222.            */
223.             if (sqldone() == SERVER_BUSY)
224.                if (!intr_sent)      /* has interrupt already been sent? */
225.                    {
226.                    printf("|      An interrupt has been received ");
227.                    printf("by the application.|\n");
228.                    printf("|                                       ");
229.                    printf("                                     |\n");
230.                    /*
231.                     * Ask user to confirm interrupt
232.                     */
233.                    if (cancel_request())
234.                        {
235.                        printf("|           TIMEOUT INTERRUPT ");
```

```
236.                    printf("REQUESTED                    |\n");
237.                /*
238.                 * Call sqlbreak() to issue an interrupt request for
239.                 * current SQL request to be cancelled.
240.                 */
241.                    sqlbreak();
242.                    }
243.                intr_sent = 1;
244.                }
245.            break;
```

==========

第 199 - 249 行

switch 语句使用回调函数参数 **when_called** 确定回调函数的操作，如下所示：

第 202 - 211 行：如果 when_called 是 0，则在数据库服务器服务器结束 SQL 请求后调用回调函数。消息请求框底部显示此消息，显示 SQL 语句的结束，如下所示：

```
+------SQL Request ends-----------------------------+
```

第 212 - 218 行：如果 when_called 是 1，则在数据库服务器开始 SQL 请求时调用此回调函数。消息请求框顶部的显示指示此条件：

```
+------SQL Request begins-----------------------------+
                    |                                                    |
```

有关这些消息请求框的更多信息，请参阅 第 21 - 30 行。此函数还指示 intr_sent 标记为 0，因为用户尚未发送此 SQL 请求的中断。

第 219 - 245 行：如果 when_called 是 2，则调用此回调函数，因为超时间隔已过。

为了处理已过的超时间隔，回调函数首先调用 GBase 8s ESQL/Csqldone() 函数（第 223 行）来确定数据库服务器是否仍在处理 SQL 请求。如果数据库服务器空闲，则应用程序不需要发送中断。如果 sqldone() 返回 SERVER_BUSY (-439)，则数据库服务器仍在忙。

224 行检查用户是否已尝试中断正在执行的 SQL 请求。如果发送了中断，

则 intr_sent 是 1，程序不需要发送另一个请求。如果中断请求还尚未发送，则回调函数通知用户超时间隔已过（第 226 - 229 行）。它然后使用 cancel_request() 函数（第 233 行）允许用户确认此中断。有关 cancel_request() 的更多信息， 请参阅第 251 - 261 行。

```
==========================================================================
======
246.    default:
247.        printf("Invalid status value in callback: %d\n", when_called);
248.        break;
249.      }
250.      }
251.  /* This function prompts the user to confirm the sending of an
252.   * interrupt request for the current SQL request.
253.   */254.   mint cancel_request()
255.  {
256.    char prompt_ans();
257.    if (prompt_ans("Do you want to confirm this interrupt?") == 'n')
258.        return(0);      /* don't interrupt SQL request */
259.    else
260.        return(1);      /* interrupt SQL request */
261. }
262.  /* This function creates a new table in the current database. It
263.   * populates this table with MAX_ROWS rows of data. */
264.   mint create_tbl()
265.  {
266.    char st_msg[15];
267.    int ret = 1;
268.    EXEC SQL BEGIN DECLARE SECTION;
269.        mint cnt;
270.        mint pa;
271.        mint i;
272.        char fld1[ CHARFLDSIZE + 1 ], fld2[ CHARFLDSIZE + 1 ];
```

```
273.      EXEC SQL END DECLARE SECTION;
274.   /*
275.    * Create canceltst table in current database
276.    */
277.      EXEC SQL create table canceltst (char_fld1 char(20),
278.           char_fld2 char(20), int_fld integer);
279.      if (exp_chk2("CREATE TABLE", WARNNOTIFY) < 0)
280.           return(0);
281.      printf("Created table 'canceltst'\n");
282.      /*
283.       * Insert MAX_ROWS of data into canceltst
284.       */
285.     printf("Inserting rows into 'canceltst'...\n");
286.     for (i = 0; i < MAX_ROWS; i++)
187.          {
```

==========

第 199 - 249 行（接上行）

如果用户确认中断，则回调函数调用 sqlbreak() 函数发送中断请求到数据库服务器。回调函数不等待数据库服务器响应中断请求。执行继续到行 243，并将 **intr_sent** 标志设置为 1，来显示中断请求已发送。如果使用无效参数的值（0 、1 或 2 以外的值）调用回调函数，该函数将显示错误消息（第 247 行）。

第 251 - 261 行

cancel_request() 函数询问用户确认中断请求。它显示以下提示：

Do you want to confirm this interrupt?

如果用户回应 y （是）则 cancel_request() 返回 0。如果用户回应 n（否），则 cancel_request() 返回 1。

第 262 - 281 行

create_tbl() 函数创建 **canceltst** 表，并将测试数据插入到此表中。CREATE TABLE 语句（第 277 和 278 行）创建具有三列 **int_fld** 、**char_fld1** 和 **char_fld2** 的 **canceltst**表。如果 CREATE TABLE 遇到错误，则 exp_chk2() 函数（第 279 行）显示诊断区域信息，create_tbl() 返回 0 来显示错误语句发生。

第 282 - 287 行

此 **for** 循环控制 **canceltst** 行的插入。MAX_ROWS 处理为此循环确定迭代次数，因此确定函数插入到表中的行数。如果由于执行速度太快而无法中断 **canceltst** 查询（第 126 - 132 行），请增加 MAX_ROWS 的值并重新编译 timeout.ec 文件。

```
==============================================================================
288.            if (i%2 == 1)    /* odd-numbered rows */
289.                {
290.                stcopy("4100 Bohannan Dr", fld1);
291               stcopy("Menlo Park, CA", fld2);
292.                }
293.            else                    /* even-numbered rows */
294.                {
295.                stcopy("Gbasedbt", fld1);
296.                stcopy("Software", fld2);
297.                }
298.            EXEC SQL insert into canceltst
299.                values (:fld1, :fld2, :i);
300.            if ( (i+1)%1000 == 0 ) /* every 1000 rows */
301.                printf("      Inserted %d rows\n", i+1);
302.            sprintf(st_msg, "INSERT #%d", i);
```

```
303.            if (exp_chk2(st_msg, WARNNOTIFY) < 0)
304.                {
305.                ret = 0;
306.                break;
307.                }
308.            }
309.        printf("Inserted %d rows into 'canceltst'.\n", MAX_ROWS);
310.    /*
311.     * Verify that MAX_ROWS rows have added to canceltst
312.     */
313.        printf("Counting number of rows in 'canceltst' table...\n");
314.        EXEC SQL select count(*) into :cnt from canceltst;
315.        if (exp_chk2("SELECT count(*)", WARNNOTIFY) < 0)
316.            return(0);
317.        printf("Number of rows = %d\n\n", cnt);
318.        return (ret);
319. }
320.    /* This function drops the 'canceltst' table */
321.    mint drop_tbl()
322.    {
323.        printf("\nCleaning up...\n");
324.        EXEC SQL drop table canceltst;
325.        if (exp_chk2("DROP TABLE", WARNNOTIFY) < 0)
326.            return(0);
327.        printf("Dropped table 'canceltst'\n");
328.        return(1);
329.    }
```

==========

第 288 - 292 行

此 **if** 语句生成 **canceltst** 表的 **char_fld1** 和 **char_fld2** 列的值。第 290 行和 291

行执行奇数行。它们将字符串 "4100 Bohannon Dr" 和 "Menlo Park, CA" 存储在 **fld1** 和 **fld2** 变量中。

第 293 - 297 行

第 295 和 296 行对偶数行执行。它们将字符 Gbasedbt 和 Software 存储在 **fld1** 和 **fld2** 变量中。

第 298 - 307 行

INSERT 语句将行插入到 **canceltst** 表中。它从 **:i** 主机变量（行号）采用 **int_fld** 列的值，采用 **:fld1** 主变量 **char_fld1** 和 **char_fld2** 列的值。该函数在每插入 1000 行后，通知用户（第 300 和 301 行）。如果 INSERT 遇到错误，则 exp_chk2() 函数（第 303 行）显示诊断区域信息，create_tbl() 返回零来指示发生错误。

第 300 - 317 行

这些行验证程序已将行添加到 **canceltst** 表中，并且可以访问它们，程序在新创建的 **canceltst** 表上执行 SELECT ，并返回找到的行数。该程序检查该号码是否与函数添加的号码相匹配，该行显示的是哪一行。如果遇到错误，则 exp_chk2() 函数（第 315 行）显示诊断区域信息，并且 create_tbl() 返回 0 以指示发生错误。

第 320 - 329 行

drop_tbl() 函数从当前数据库删除 **canceltst** 表。如果 DROP TABLE 语句（第 324 行）遇到错误，则 exp_chk2() 函数显示诊断区域信息，drop_tbl() 返回 0 以指示发生错误。

==========

```
330.  /*
331.   * The inpfuncs.c file contains the following functions used in
       this
332.   * program:
333.   *       getans(ans, len) - accepts user input, up to 'len' number of
334.   *                          characters and puts it in 'ans'
335.   */
336.  #include "inpfuncs.c"
337.  char prompt_ans(question)
338.  char * question;
339.  {
340.     char ans = ' ';
341.     while(ans != 'y' && ans != 'n')
342.          {
343.          printf("\n*** %s (y/n): ", question);
344.          getans(&ans,1);
345.          }
346.     return ans;
347.  }
348.  /*
349.  * The exp_chk() file contains the exception handling functions to
350.   * check the SQLSTATE status variable to see if an error has
          * occurred
351.   * following an SQL statement. If a warning or an error has
352.   * occurred, exp_chk2() executes the GET DIAGNOSTICS statement and
353.   * displays the detail for each exception that is returned.
354.   */
355.  EXEC SQL include exp_chk.ec;
```

==========

第 330 - 336 行

某些 GBase 8s ESQL/C 演示程序还会调用 getans() 函数。因此，该函数被分解为一个独立的 C 源文件并在适当的演示程序中包含。因为此函数不包含 GBase 8s ESQL/C，所以查询可以使用 C **#include** 预处理器语句包含此文件。

第 337 - 347 行

prompt_ans() 函数显示 **question** 参数中的字符串并等待用户输入 y（是）或 n（否）作为回应。它返回单个字符响应。

第 348 - 355 行

timeout 程序使用 exp_chk2() 、sqlstate_err() 、disp_error() 和 disp_warning() 函数执行它的异常处理。因为几个演示程序使用这些函数，所以 exp_chk2() 函数及其支持的函数放在单独的 exp_chk.ec 源文件中。timeout 程序必须包含具有 GBase 8s ESQL/Cinclude 伪指令的程序，因为异常处理函数使用GBase 8s ESQL/C 语句。

提示： 在生产环境中，将 getans() 、exp_chk2() 、sqlstate_err() 、disp_error() 和 disp_warning() 函数放到库中并在 GBase 8s ESQL/C 编译程序的命令行包含此库。

示例输出

本节包含 **timeout** 演示程序的示例输出。

该程序执行两次 **canceltst** 查询，如下所示：

第 20 - 43 行：一旦出现确认提示，第一次运行就会确认中断请求。（用户输入 y）

第 44 - 75 行：第二次运行不会确认中断请求（用户输入 n）

以下输出中出现的数字仅供参考，它们不会出现在实际的程序输出中。

```
==============================================================
======
```

1. TIMEOUT Sample ESQL Program running.
2. Connected to 'stores7' on default server
3. Created table 'canceltst'

```
4.  Inserting rows into 'canceltst'...
5.      Inserted 1000 rows
6.      Inserted 2000 rows
7.      Inserted 3000 rows
8.      Inserted 4000 rows
9.      Inserted 5000 rows
10.     Inserted 6000 rows
11.     Inserted 7000 rows
12.     Inserted 8000 rows
13.     Inserted 9000 rows
14.     Inserted 10000 rows
15. Inserted 10000 rows into 'canceltst'.
16. Counting number of rows in 'canceltst' table...
17. Number of rows = 10000
18. Timeout interval for SQL requests is: 0.00333333 minutes
19. *** Are you ready to begin execution of the query? (y/n): y
20. Beginning execution of query...
21. +------SQL Request begins----------------------------+
22. |                                                     |
23. +------SQL Request ends-----------------------------+
24. +------SQL Request begins----------------------------+
25. |                                                     |
26. |     An interrupt has been received by the application.|
27. |                                                     |
28. *** Do you want to confirm this interrupt? (y/n): y
29. |          TIMEOUT INTERRUPT REQUESTED                |
30. +------SQL Request ends-----------------------------+
```

==========

第 3 - 17 行

create_tbl() 函数生成这些行。它们指示函数已经成功创建 canceltst 表，插入了 MAX_ROWS 数行（1,000），并确认 SELECT 语句可以访问行。有关 create_tbl() 函数的描述，请参阅第 262 - 281 行开头的注释。

第 18 - 19 行

第 18 行显示超时间隔来指示 sqlbreakcallback() 成功注册了回调函数，并建立了 200 毫秒（0.00333333 分钟）的超时间隔。第 19 行询问用户以指示开始查询。此提示将为用户提供确认提示（第 28 行和 43 行）。当数据库服务器仍在执行查询时，必须快速回复该请求以发送中断。

第 20 行

该行指示 dspquery() 函数开始执行，数据库服务器开始执行 **canceltst** 查询。

第 21 - 30 行

程序输出使用消息请求框来指示客户端-服务器通信：

```
+------SQL Request begins----------------------------+
       |                                               |
       +------SQL Request ends-----------------------------+
```

每个框代表在客户端和服务器之间发送的单个消息请求。回调函数显示消息请求框的文本。（有关函数的哪些部分描述显示文件，请参见第 199 - 249 行。）要执行 OPEN 语句，客户端和服务器交换了两个消息请求。输出中的两个消息请求框指示。有关消息请求的更多信息，请参阅中断 SQL 语句。

第一个消息请求框（第 21 - 23 行）表示第一个消息请求在超时间隔过去之前完成。第二个消息请求框（第 29 - 30 行）表示此消息请求的执行超过了超时间隔，并调用状态值为 2 的回调函数。回调函数提示用户确认中断请求（第 28 行）。

第 29 行指示 sqlbreak() 函数已经请求中断。然后该消息请求完成（第 30 行）。

```
================================================================
======
```

```
31.  TIMEOUT INTERRUPT PROCESSED
32.  *** Try another run? (y/n): y
33.  Timeout interval for SQL requests is: 0.00333333 minutes
34.  *** Are you ready to begin execution of the query? (y/n): y
35.  Beginning execution of query...
36.  +------SQL Request begins----------------------------+
37.  |                                                    |
38.  +------SQL Request ends-----------------------------+
39.  +------SQL Request begins----------------------------+
40.  |                                                    |
41.  |     An interrupt has been received by the application.|
42.  |                                                    |
43.  *** Do you want to confirm this interrupt? (y/n): n
44.  +------SQL Request ends-----------------------------+
45.  Displaying data...
46.     sum(int_fld) = 25000000
47.     char_fld1 = 4100 Bohannan Dr
48.     char_fld2 = Menlo Park, CA
49.     sum(int_fld) = 24995000
50.     char_fld1 = Gbasedbt
51.     char_fld2 = Software
52.  Number of rows found: 2
53.  *** Try another run? (y/n): n
54.  Cleaning up...
55.  Dropped table 'canceltst'
56.  Disconnected stores7 connection
57.  TIMEOUT Sample Program over.
```

```
================================================================
==
```

第 31 行

当数据库服务器实际处理中断请求时，它将 SQLCODE 设置为 -213。第 31 行指示应用程序已经响应此状态。

第 32 行

此提示表示 **canceltst** 查询的第一次运行结束。用户第二次响应 y 到提示运行查询。

第 36 - 41 行

消息请求框指示第一个消息请求在超时间隔过去之前完成。第二个消息请求框（第 39 - 44 行）指示该消息请求的执行在才超过时间间隔，并调用回调函数（**when_called** = 2）。回调函数提示用户确认中断请求（第 43 行）。这次用户回答 n。

第 45 - 52 行

因为用户没有中断 **canceltst** 查询，使用程序显示查询返回的行信息。

第 54 和 55 行

drop_tbl() 函数生成这些行。它们指示函数成功从数据库删除 canceltst 比哦啊。有关 drop_tbl() 函数的描述，请参阅第 320 - 329 行开头的注释。

3.2.7 Windows(TM) 环境中的 ESQL/C 连接库函数

要建立显式连接（有时称为直接连接）， GBase 8s ESQL/C 支持 SQL 连接语句。GBase 8s ESQL/C 还支持下表中列出的连接库函数，以在 Windows(TM) 环境中建立显式连接。

表 4. ESQL/C 连接库函数及其等价的 SQL 语句

Windows(TM) 库函数的 ESQL/C	描述	等价 SQL 语句	请参阅
GetConnect()	请求显式连接并返回指向连接信息的指针	CONNECT TO '@dbservername' WITH CONCURRENT TRANSACTION	GetConnect() 函数（Windows）
SetConnect()	将连接切换为已建立（休眠状态的）的显式连接	SET CONNECT TO （不带 DEFAULT 选项）	SetConnect() 函数（Windows）
ReleaseConnect()	关闭已建立的显式连接	DISCONNECT （不带 DEFAULT 、CURRENT 或 ALL 选项）	ReleaseConnect() 函数（Windows）

重要： GBase 8s ESQL/C 支持库函数，以兼容 Windows(TM) 应用程序 5.01 版本的 GBase 8s ESQL/C。当在为 Windows(TM) 环境编写新的 GBase 8s ESQL/C 应用程序时，使用 SQL 连接语句（CONNECT 、DISCONNECT 和 SET CONNECTION）而不是 GBase 8s ESQL/C 连接库函数。

GBase 8s ESQL/C 使用包含连接语句柄以及其它连接信息。GBase 8s ESQL/C 连接库函数使用连接语句柄以及内部结构中的信息将应用程序的连接信息传递给应用程序。应用程序可以使用连接语句柄来标识显式连接。

如果使用这些连接函数建立显式连接，请注意以下限制：

如果在一个模块中打开（如果共享 DLL），然后使用显式连接来使用另一个模块中的游标时，在声明游标时必须使用主机变量作为游标的名称。

确保您的应用程序始终使用正确的游标语句柄。

重要： 如果应用程序使用错误的连接句柄，则应用程序可以更改错误的数据库而不需通知用户。

当您编译 GBase 8s ESQL/C 程序时，esql 命令处理器自动将 GBase 8s ESQL/C 连接函数链接到您的程序。

3.3 GBase 8s 库

这些主题介绍如何使用 GBase 8s ESQL/C 应用程序链接静态的、共享的以及线程安全的 GBase 8s 通用库。

GBase 8s 产品使用 GBase 8s 通用库来进行客户端 SQL 应用程序接口（API）产品（GBase 8s ESQL/C 和 GBase 8s ESQL/COBOL）以及数据库服务器。可以在以下类型的 GBase 8s 通用库中选择链接到您的 GBase 8s ESQL/C 应用程序的库：

静态 GBase 8s 通用库

要链接静态库，链接器将函数复制到您的 GBase 8s ESQL/C 程序的可执行文件中。静态 GBase 8s 通用库允许不支持共享的计算机上的 GBase 8s ESQL/C 程序访问 GBase 8s 通用库函数。

共享 GBase 8s 通用库

要链接共享库，链接器将有关库的位置复制到您的 GBase 8s ESQL/C 程序的可执行文件中。共享 GBase 8s 库允许多个应用程序共享这些库的单个副本，操作系统将一次加载到共享内存中。

静态和共享 GBase 8s 通用库的线程安全版

线程安全版本的 GBase 8s 通用库允许具有多个线程的 GBase 8s ESQL/C 应用程序同时调用这些库函数。线程安全版本的 GBase 8s 库可用作静态库和共享库。

从 GBase 8s Client Software Development Kit 3.0 版本开始， GBase 8s 通用库的静态版本在 Windows™ 和 UNIX™ 操作系统上都可用。下表显示了可用的选项。

表 1. 可用于 UNIX ™和 Windows 的不同版本的 ESQL/C 通用库

链接选项	线程安全	缺省
静态	静态、线程安全通用库	静态，缺省通用库
共享	共享、线程安全通用库	共享，缺省通用库

3. 3. 1 选择 GBase 8s 通用库的版本

本节提供有关以下主题的信息：

GBase 8s 通用库是什么？

esql 命令的哪些命令行选项与 GBase 8s ESQL/C 程序链接 GBase 8s 通用库的版本？

如何使用 GBase 8s ESQL/C 程序链接可用于 UNIX™ 和 Windows™ 操作系统的静态 GBase 8s 通用库？

如何使用 GBase 8s ESQL/C 程序链接共享 GBase 8s 通用库？

您需要考虑哪些因素来确定要使用哪种类型的 GBase 8s 通用库？

GBase 8s 通用库

以下是GBase 8s ESQL/C 在 UNIX™ 操作系统上的 GBase 8s 通用库列表。

libgen

包含一般任务的函数。

libos

包含操作系统需要的任务的函数。

libsql

包含在客户端应用程序和数据库服务器之间发送 SQL 语句的函数。

libgls

包含为 GBase 8s 产品通过全球语言支持（GLS）的函数。

libasf

包含处理客户端应用程序和数据库服务器之间通信协议的函数。

GBase 8s 通用库位于 UNIX 操作系统的 $GBASEDBTDIR/lib/esql 和 $GBASEDBTDIR/lib 目录 中。

GBase 8s ESQL/C For Windows™ 的 GBase 8s 通用库只是一个名为的 isqlt09a.dll的 DLL。该文件在 %GBASEDBTDIR%\lib 目录中。

GBase 8s ESQL/C For Windows 的静态库名为 isqlt09s.lib。该文件在 $GBASEDBTDIR/lib 目录中。

在许多平台上有一个名为 libgen.a 的系统库。要避免编译错误，建议您不要使用 libgen.a GBase 8s 库。而是使用 libifgen.a GBase 8s 库，它包含一个到 libgen.a 的符号链接。

esql 命令

要确定您 GBase 8s ESQL/C 应用程序的要链接的 GBase 8s 通用库的类型，esql 命令支持下表的命令行选项。

表 1. GBase 8s 通用库的 esql 命令行选项

要链接的 GBase 8s 库版本	esql 命令行选项	请参阅
共享库	没有选项(缺省)	链接共享 GBase 8s 通用库
静态库	**-static**	链接静态 GBase 8s 通用库.

线程安全共享库	**-thread**	UNIX 系统上将线程安全 GBase 8s 通用库链接到 ESQL/C 模块和将线程安全的 GBase 8s 通用库链接到 Windows 环境中的 ESQL/C 模块
线程安全静态库	**-thread -static**	在 UNIX 操作系统上创建动态线程库

链接静态 GBase 8s 通用库

从 GBase 8s Client Software Development Kit 版本 3.0 开始，Windows™ 和 UNIX™ 操作系统上提供了静态版本的 GBase 8s 通用库。

静态 GBase 8s 通用库保留其 Version 7.2 的名称，静态库名称具有以下格式：

非程序安全的静态 GBase 8s 通用库的名称 libxxx.a。

线程安全的静态 GBase 8s 通用库的名称为 libthxxx.a。

在这些静态库名称中，*xxx* 标识了特定的静态 GBase 8s 通用库。对于 Version 7.2 和更高版本，静态和线程安全的静态 GBase 8s 通用库将使用此格式的名称作为其实际名称。以下示例输出显示 **libos** 静态（libos.a）和线程安全静态（libthos.a）库：

```
% cd $GBASEDBTDIR/lib/esql
        % ls -l lib*os.a
        -rw-r--r--  1 gbasedbt     145424 Nov 8 01:40 libos.a
        -rw-r--r--  1 gbasedbt     168422 Nov 8 01:40 libthos.a
```

esql 命令将与静态 GBase 8s 通用库的实际名称相关联的代码链接到 GBase 8s ESQL/C 应用程序。运行时，您的 GBase 8s ESQL/C 程序可以直接从它的可执行文件访问这些 GBase 8s 通用库函数。

将静态 GBase 8s 通用库链接到 ESQL/C 模块中

要使用 GBase 8s ESQL/C 模块链接 GBase 8s 通用库，请使用 **-static** 命令行选项编译您的程序。

以下命令使用 file.exe 可执行文件链接非线程安全 GBase 8s 库：

```
esql -static file.ec -o file.exe
```

esql 命令还可以将线程安全的 GBase 8s 通用库的代码与 GBase 8s ESQL/C 应用程序链接。

提示： Version 7.2 之前的 GBase 8s ESQL/C esql 命令链接了 GBase 8s 通用库的静态版本。因为 esql 命令缺省链接这些库的共享版本。您必须指定 **-static** 选项将静态版本与 GBase 8s ESQL/C 应用程序链接。

链接共享 GBase 8s 通用库

GBase 8s ESQL/C 可以自动链接共享库，它将此库放置到共享内存中。当共享库在共享内存中时，其它 GBase 8s ESQL/C 应用程序也会使用它。共享库在多用户环境中最为有用，其中所有应用程序只需要一个库副本。

重要： 要在 GBase 8s ESQL/C 应用程序中使用共享库，您的操作系统必须支持共享

库。支持共享库的操作系统包括 Sun 和 HP 版本的 UNIX™ 和 Windows™。您应该熟悉创建共享库以及 C 编程器需要构建它们的编译选项。

链接共享库（UNIX）的符号名称

当 esql 命令使用 GBase 8s ESQL/C 应用程序链接共享或线程安全共享 GBase 8s 库时，它使用这些函数的 *symbolic* 名称。

非线程安全共享 GBase 8s 通用库具有 libxxx.yyy. 格式的符号名称。

线程安全共享 GBase 8s 通用库具有 libthxxx.yyy 格式的符号名称。

在这些静态库名称中，*xxx* 标识特定的库，*yyy* 是一个特定于平台的文件扩展名，用于标识共享库文件。

提示： 要引用指定的共享库文件，此版本经常使用 Sun UNIX™ 操作系统的文件扩展名，.so 文件扩展名。对于 UNIX 操作系统使用的共享库文件扩展名，请参阅 UNIX 操作系统文件。

当您安装 GBase 8s ESQL/C 产品时，安装脚本将实际共享成品库名称与符号名称的文件进行符号链接。下图显示了 GBase 8s 库的共享和线程安全共享版本的实际名称格式。

图: GBase 8s 共享库名称的格式

以下示例输出显示了 **libos.a** 静态库和 **libos.so** 共享库（在 Sun 平台上）的符号以及实际名称：

```
%ls -l $GBASEDBTDIR/esql/libos*
        -rw-r--r--   1 gbasedbt     145424 Nov 8 01:40 libos.a
        lrwxrwxrwx   1 root             11 Nov 8 01:40 libos.so -> iosls07a.so*
```

esql 命令使用 GBase 8s ESQL/C 应用程序链接符号共享库名称。运行期间，当程序需要 GBase 8s 通用库函数时，GBase 8s ESQL/C 动态链接共享 GBase 8s 通用库的代码。

将共享 GBase 8s 通用库链接到 ESQL/C 模块中

要将共享 GBase 8s 通用库链接到 ESQL/C 模块中：

在运行时设置指定库搜索路径的环境变量，以便它在 UNIX™ 操作系统上包含 $GBASEDBTDIR/lib 和 $GBASEDBTDIR/lib/esql 路径；在 Windows™ 环境中包含 %GBASEDBTDIR%lib。

在许多 UNIX 操作系统上，LD_LIBRARY_PATH 环境变量指定库搜索路径。以下命令在 C shell 中设置 LD_LIBRARY_PATH：

```
setenv LD_LIBRARY_PATH $GBASEDBTDIR/lib:$GBASEDBTDIR/
```

```
lib/esql:/usr/lib
```

在 Windows 环境中，使用以下命令：

```
set LIB = %GBASEDBTDIR%\lib\;%LIB%
```

使用 esql 命令编译程序。

要使用 GBase 8s ESQL/C 模块链接 GBase 8s 通用库，您不需要指定命令行选项。GBase 8s ESQL/C 缺省链接共享库。以下命令使用共享 GBase 8s 库编译 file.ec 源文件：

```
esql file.ec -o file.exe
```

当 esql 命令使用 GBase 8s ESQL/C 应用程序链接线程安全共享 GBase 8s 通用库时，还会使用信号名称。

在共享和静态库版本之间选择

从 GBase 8s Client Software Development Kit 版本 3.0 开始，GBase 8s 通用库的静态版本适用于 Windows™ 和 UNIX™ 操作系统。

共享库在多用户环境中十分有用，其中所有应用程序只需要一个库副本。共享库为您的 GBase 8s ESQL/C 应用程序带来以下好处：

共享库减少可执行文件的大小，因为这些库函数是在需要的基础上动态链接的。

运行时，共享库的单个副本会链接到多个程序，这导致更少的内存使用。

GBase 8s ESQL/C 可执行文件中共享库的效果对用户来说是透明的。

尽管共享库保存磁盘和内存空间，但是当 GBase 8s ESQL/C 应用程序使用它们时，它必须执行以下所有任务：

首次将共享库动态加载到内存中

执行链接编辑操作

执行库位置无关代码

这些开销任务可能会导致运行时迟缓，并且在使用静态库时不是必需的。但是，一旦操作系统语句加载并映射了 GBase 8s 共享库，输入/输出（I/O）访问时间就可以抵消这些成本。

重要： 当第一次执行应用程序加载共享库时，可能会对应用程序的客户端的性能造成一次性负面影响。有关更多信息，请参阅操作系统文档。

由于操作系统需要加载程序及其库的真实 I/O 时间，通常不会超过保存的 I/O 时间，所以使用共享库的查询的性能明显与使用静态库的程序一样好或更好。但是，如果应用程序不进行共享，或者如何您的处理器在应用程序调用共享库例程时饱和，则可能无法实现这些节省。

还可以使用 GBase 8s ESQL/C 应用程序链接静态和共享 GBase 8s 通用库的线程安全版本。

3.3.2 之前存在的 ESQL/C 应用程序与当前库版本的兼容性

可以指定 esql 命令行选项（在表 1中）来告知 esql 命令要与 GBase 8s ESQL/C 应用程序链接的 GBase 8s 库的版本。在 esql 命令成功编译并链接您的应用程序后， GBase 8s 通用库的版本就是固定的。当安装新版本的 GBase 8s ESQL/C 时，会接收到 GBase 8s 通用库的新副本，是否需要重新编译并重新链接现有的 GBase 8s ESQL/C 应用程序以运行这些新副本取决于下表所述的因素。

变更为 GBase 8s 通用库	GBase 8s 通用库的版本	需要重新编译或重新链接?
GBase 8s 通用库的新版本	静态 线程安全静态	只有应用程序需要在新版本中使用新功能
新版本中 GBase 8s 通用库具有一个新的主版号	共享 线程安全共享	只有应用程序需要在新版本中使用新功能
新版本中 GBase 8s 通用库具有一个新 API 版号	共享 线程安全共享	必须重新编译并重新链接

在 UNIX™ 上，您可以使用 ifx_getversion 实用程序确定安装在您的系统上的 GBase 8s 库的版本。

在 Windows™ 环境中，使用以下 find 命令在 isqlt09A.dll 中找到包含版本号字符串的出处。该命令需要从 %GBASEDBTDIR%\bin 命令发出。

```
C: cd %GBASEDBTDIR%\bin
C: find "GBASEDBT-SQL" isqlt09a.dll
```

find 命令的输出如下所示：

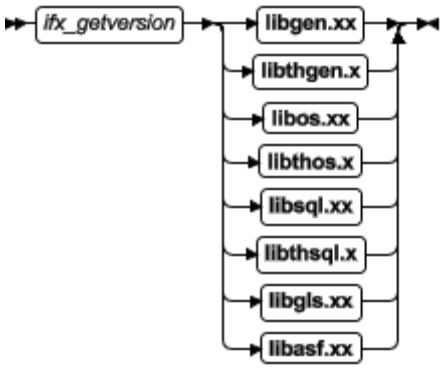
```
---------- ISQLT09A.DLL
           GBASEDBT-SQL Version
```

ifx_getversion 实用程序（UNIX™）

要获得 GBase 8s 库的完整的版本名称，请使用 ifx_getversion 实用程序。

在运行 ifx_getversion 之前，将 GBASEDBTDIR 环境变量设置为您的 GBase 8s 产品的安装目录。

ifx_getversion 实用程序具有以下语法。



元素	意义	关键注意事项
xx	对于静态库，*xx* 指定 .a 文件扩展名；对于共享库，*xx* 指定特定于平台的文件扩展名。	对于共享库，Sun 平台使用 .so 文件扩展名，Hewlett-Packard （HP）平台使用 .sl 文件扩展名。

以下示例显示 ifx_getversion 实用程序为 **libgen** GBase 8s 库生成的输出：

```
GBase 8s-Client SDK Version 3.00.UN191
        GBase 8s LIBGEN LIBRARY Version 3.00.UN191
        Copyright (C) 1991-2007 GBase
```

ifx_getversion 的输出取决于安装在您的系统上的 GBase 8s ESQL/C 软件的版本。

创建库的 API 版本

当调用链接到共享 GBase 8s 通用库的 GBase 8s ESQL/C 应用程序时，这些共享库的版本号必须与 $GBASEDBTDIR/lib 目录中的共享库的版本号相符合。

在 Windows™ 环境中，开发者可以很容易地验证共享库 DLL 的名称，例如 isqlt*nnx*.dll 其 *nn* 表示版本号，*x* 表示 API 版本号。

但是，对于 UNIX™ 上的 GBase 8s ESQL/C 应用程序，鉴于链接库获取符号名称，找到链接库的版本并不容易。因此，GBase 8s ESQL/C 会对您进行检查。GBase 8s ESQL/C 在应用程序使用的库的 API 版本之间执行检查。图 1显示了 API 版本的在共享库中的位置。

GBase 8s ESQL/C 使用名为 checkapi() 的 GBase 8s 函数执行此检查。checkapi() 函数在 checkapi.o 对象文件中，它包含在 $GBASEDBTDIR/lib/esql 命令中。esql 命令自动将 checkapi.o 对象文件与它创建的每一个可执行文件相链接。

要确定应用程序使用的库的 API 版本，GBase 8s ESQL/C 将检查可执行文件中特殊宏定义的值。当 GBase 8s ESQL/C 预处理器处理源文件时，它将 sqlhdr.h 头文件中的宏定义复制到生成的 C 源文件（.c）中。以下示例显示了这些宏的示例值：

```
#define CLIENT_GEN_VER          710
#define CLIENT_OS_VER           710
#define CLIENT_SQLI_VER         710
#define CLIENT_GLS_VER          710
```

提示： GBase 8s ESQL/C 预处理器自动在其生成的所有 GBase 8s ESQL/C 可执行文件中包含 sqlhdr.h 文件。

如果可执行文件中的库的 API 版本不兼容，则 GBase 8s ESQL/C 返回运行错误，来指示哪个库不兼容。必须重新编译 GBase 8s ESQL/C 应用程序来链接新版本的共享库。 .

如果不使用 esql 将其中一个共享 GBase 8s 通用库与 GBase 8s ESQL/C 应用程序链接，则您必须显式地将 checkapi.o 文件与您的应用程序相链接。否则，GBase 8s ESQL/C 可能会在链接时生成错误。

```
undefined ifx_checkAPI()
```

3.3.3 创建线程安全 ESQL/C 应用程序

GBase 8s ESQL/C 在 UNIX™ 和 Windows™ 操作系统上提供共享的和静态线程安全以及共享和静态的缺省版本的 GBase 8s 通用库。在 Windows 操作系统中，GBase 8s ESQL/C 提供名为 isqlt09a.dll 的 DLL 以及名为 isqlt09s.lib 的静态线程安全库。

线程安全 GBase 8s ESQL/C 应用程序的每个线程可以有一个活动连接。每个应用程序可以有多个线程。线程安全库包含线程安全（或可重入）函数。线程安全函数是当多个线程同时调用它时正常运行的函数。

对于 UNIX™ 操作系统上的 GBase 8s ESQL/C，线程安全 GBase 8s 通用库使用来自分布式计算环境（DCE）线程包的函数。DCE 线程库（Open Software Foundation （OSF）开发）为线程安全应用程序创建标准接口。

如果 DCE 线程库在您的操作系统上不可用，则 ESQL/C 可以使用 POSIX 线程库或 Sun Solaris 线程库。

如果您的操作系统支持 DCE 、POSIX 或 Solaris 线程包，则必须将它们安装在与 ESQL/C 相同的客户端计算机上。

在 Windows 环境中， GBase 8s 通用库使用 Windows API 来确保它们是线程安全。

使用程序安全 GBase 8s 通用库，可以开发线程安全 GBase 8s ESQL/C 应用程序。线程安全应用程序可以控制许多线程。它将一个进程分隔成多个可执行线程，每个线程独立运行。当非线程的 GBase 8s ESQL/C 应用程序可以建立与一个或多个数据库的连接时，但它一次只能有一个活动连接。活动连接时准备好处理 SQL 请求的连接。线程安全的 GBase 8s ESQL/C 应用程序灭个线程可以有一个连接，每个应用程序可以有多个线程。

当指定 -thread 命令行选项时，esql 命令将此选项发送给 GBase 8s ESQL/C 预处理器 esqlc。使用 **-thread** 选项，GBase 8s ESQL/C 预处理器生成线程安全的代码， 不同的线程可以并发执行。

线程安全 ESQL/C 代码的特征

线程安全 GBase 8s ESQL/C 代码具有不同于非线程按代码的以下特征：

线程安全代码无法定义任何静态数据结构。

例如：GBase 8s ESQL/C 自动分配 **sqlda** 结构，并在运行时将主机变量与这些 **sqlda** 结构绑定。

线程安全代码动态地声明游标块而不是静态地声明它们。

线程安全代码使用状态变量的宏定义（SQLCODE 、SQLSTATE 和 sqlca 结构）。

由于上述的不同，GBase 8s ESQL/C 预处理器生成的线程安全 C 源文件（.c） 与非线程安全的 C 源文件不同。因此，不能将用 **-thread** 选项编译的 GBase 8s ESQL/C 应用程序与用 **-thread** 选项编译的应用程序相链接。要链接这些应用程序，您必须使用 **-thread** 选项编译这两个应用程序。

编写线程安全 ESQL/C 应用程序

本节介绍如何创建线程安全 GBase 8s ESQL/C 应用程序。

并发活动连接

在线程安全 GBase 8s ESQL/C 应用程序中，数据库服务器连接可以是以下之一的状态：

活动的数据库服务器连接已经准备好处理 SQL 请求。

线程安全 GBase 8s ESQL/C 应用程序的主要优点是每个线程都可以有一个到数据库服务器的活动连接。使用 CONNECT 语句建立连接并激活它。使用 SET CONNECTION 语句（没有 DORMANT 子句）来激活休眠的连接。

已建立休眠的数据库服务器连接 ，当前不与线程相关联。

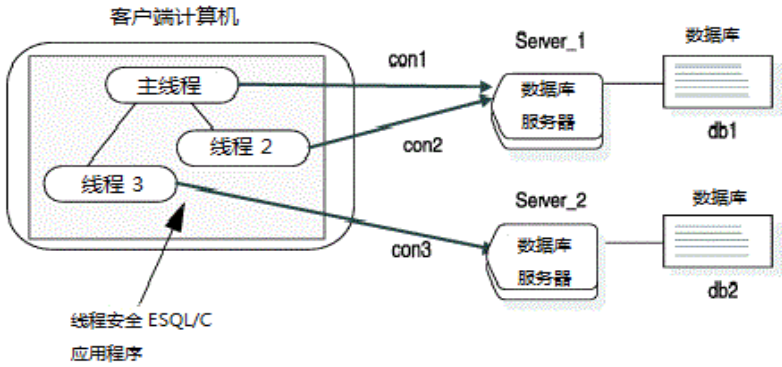
当线程使活动连接处于休眠状态时，则该连接将可用于其它线程。相反，当线程使休眠连接处于活动状态时，该连接将变得不可用于其它线程。使用 SET CONNECTION...DORMANT 语句显式地将连接置于休眠状态。

当前连接是当前向数据库服务器发送 SQL 请求并可能从数据库服务器的活动数据库服务器连接。单线程应用程序一次只有一个当前（或活动的）连接。在多线程应用程序中，每个线程都可以具有当前连接。因此，多线程应用程序可以同时具有多个活动连接。

当您使用 SET CONNECTION 语句（没有 DORMANT 子句）语句切换连接时，SET CONNECTION 隐式地将当前连接置于休眠状态。在休眠状态时，连接可用于其它线程。任何线程可以访问任何休眠连接。但是，一个线程一次只能具有一个活动连接。

下图显示了线程安全 GBase 8s ESQL/C 应用程序建立了三个并发连接，每个连接处于活动状态。

图: 线程安全 ESQL/C 应用程序中的并发连接



在上图中，GBase 8s ESQL/C 应用程序由以下三个线程构成：

主线程（主函数）启动连接 con1 到 Server_1 上的数据库 db1。

主线程创建线程 2。线程 2 建立到 Server_1 上的数据库 db1 的连接 con2。

主线程创建线程 3。线程 3 建立到 Server_2 上的数据库 db2 的连接 con3。

图 1中的连接都是并发活动的，都可以执行 SQL 语句。以下代码片段建立了图 1显示的连接。它没有显示 DCE 相关的调用，以及 start_threads() 函数的代码。

```
main()
        {
        EXEC SQL connect to 'db1@Server_1' as 'con1';
        start_threads(); /* start 2 threads */
        EXEC SQL select a into :a from t1; /* table t1 resides in db1 */

        ⋮

        }
        thread_1()
        {
        EXEC SQL connect to 'db1@Server_1' as 'con2';
        EXEC SQL insert into table t2 values (10); /* table t2 is in db1 */
        EXEC SQL disconnect 'con2';
        }
        thread_2()
        {
        EXEC SQL connect to 'db2@Server_2' as 'con3';
        EXEC SQL insert into table t1 values(10); /* table t1 resides in db2
        */
        EXEC SQL disconnect 'con3';
        }
```

可以使用 ifx_getcur_conn_name() 函数获得当前连接的名称。

跨线程连接

如果您的应用程序包含需要使用相同连接的线程，则一个线程可能在另一个线程需要访问时使用该连接。违例避免这种争用，您的 GBase 8s ESQL/C 应用程序必须管理对连接的访问。

管理几个线程必须使用连接的最简单的方法是将 SET CONNECTION 语句放到循环中。以下代码片段显示了 SET CONNECTION 循环示例。

```
/* wait for connection: error -1802 indicates that the connection
        is in use
        */
        do {
        EXEC SQL set connection   :con_name;
        } while (SQLCODE == -1802);
```

上述算法等待主机变量 **:con_name** 名称变为可用的连接。然而，这种方法的缺点是它消耗了处理器周期。

以下代码片段使用 CONNECT 语句创建连接，使用 SET CONNECTION 语句激活线程中的休眠的连接。它还使用 SET CONNECTION...DORMANT 激活休眠的连接。此代码片段建立图 1中所示的连接。它无法显示 DCE 相关的调用，以及 start_threads() 函数的代码。

```
main()
        {    EXEC SQL BEGIN DECLARE SECTION;
        int a;
        EXEC SQL END DECLARE SECTION;

        start_threads(); /* start 2 threads */
        wait for the threads to finish work.

        /* Use con1 to update table t1; Con1 is dormant at this point.*/
        EXEC SQL set connection 'con1';
        EXEC SQL update table t1 set a = 40 where a = 10;

        /* disconnect all connections */
        EXEC SQL disconnect all;
        }
        thread_1()
        {
        EXEC SQL connect to 'db1' as 'con1';
        EXEC SQL insert into table t1 values (10); /* table t1 is in db1*/

        /* make con1 available to other threads */
        EXEC SQL set connection 'con1' dormant;

        /* Wait for con2 to become available and then update t2 */
        do {
        EXEC SQL set connection 'con2';
        } while ((SQLCODE == -1802) );
        if (SQLCODE != 0)
        return;
        EXEC SQL update t2 set a = 12 where a = 10; /* table t2 is in db1 */
        EXEC SQL set connection 'con2' dormant;
        }
```

```
thread_2()
{     /* Make con2 an active connection */
EXEC SQL connect to 'db2' as 'con2';
EXEC SQL insert into table t2 values(10); /* table t2 is in db2*/
/* Make con2 available to other threads */
EXEC SQL set connection'con2' dormant;
}
```

在此代码中，**thread_1()** 使用 SET CONNECTION 语句循环（参见图 1）来等待 **con2** 变成可用状态。当 **thread_2()** 使 **con2** 休眠，其它线程可以使用此连接。同时，**thread_1()** 中的 SET CONNECTION 语句成功，并且 **thread_1()** 可用使用 **con2** 连接更新表 **t2**。

DISCONNECT ALL 语句

DISCONNECT ALL 语句按顺序断开应用程序中的所有连接。

在线程安全 GBase 8s ESQL/C 应用程序中，只有发出 DISCONNECT ALL 语句的线程可以处理 SQL 语句（在这情况中，是 DISCONNECT ALL 语句）。如果任何其它语句正在执行 SQL 语句，则当 DISCONNECT ALL 语句尝试断开此连接失败。此故障可能会使应用程序处于不一致的状态。

例如，假设 DISCONNECT ALL 语句成功断开连接 A 和连接 B 但是无法断开连接 C ，因为此连接正在处理一个 SQL 语句。DISCONNECT ALL 语句失败，连接 A 和 B 断开但是连接 C 打开。建议您在所用的线程完成它们的工作后在应用程序的主函数中发出 DISCONNECT ALL 语句。

当 DISCONNECT ALL 语句按顺序断开应用程序连接时，GBase 8s ESQL/C 阻止其它连接请求。如果别的线程在 DISCONNECT ALL 语句执行时请求连接，则此连接必须等待直到 DISCONNECT ALL 语句完成，然后 GBase 8s ESQL/C 可以发送此新的连接请求到数据库服务器。

跨线程的预备语句

PREPARE 语句的作用域在连接级别。即，它们与连接相关联。当线程激活连接，它可以访问与此连接相关联的任何准备好的语句。如果您的线程安全的 GBase 8s ESQL/C 应用程序使用准备好的语句，则您可能需要隔离 PREPARE 语句的编译，以便在程序中仅编译一次。

构建应用程序的一种可能方法是执行将连接上下文初始化为一组的语句。连接上下文包括当前用户的名称和数据库环境与此名称相关联的信息（包括准备好的语句）。

对于每个连接，应用程序将执行以下步骤：

使用 CONNECT 语句建立线程所需的连接。

使用 PREPARE 语句编译与连接相关联的任何 SQL 语句。

使用 SET CONNECTION...DORMANT 语句将连接置于休眠状态。

当连接是休眠的，任何线程可以通过 SET CONNECTION 语句访问休眠的连接。当线程激活连接时，它可以发送对应的预备的语句到数据库服务器执行。

在下图中，代码片段在连接初始化时准备 SQL 语句，然后在程序中执行它们。

图: 跨线程使用准备好的 SQL 语句

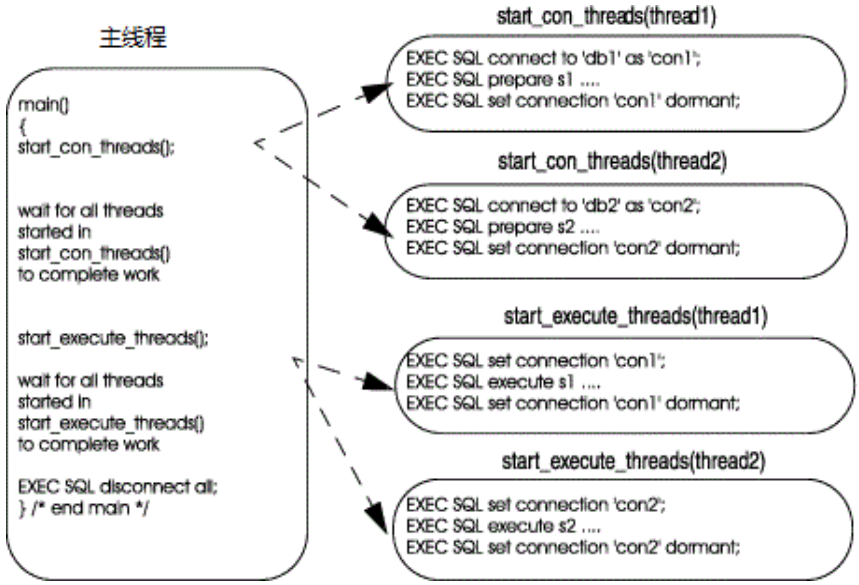


图 1中的代码执行以下操作：

主线程调用 start_con_threads()，它调用 start_con_thread() 来启动两个线程：

对于线程 1，start_con_thread() 函数建立连接 con1，准备名为 s1 的语句，使连接 con1 休眠。

对于线程 2，start_con_thread() 函数建立连接 con2，准备名为 s2 的语句，使连接 con2 休眠。

主线程调用 start_execute_threads()，它调用 **start_execute_thread()** 来为每个线程执行准备好的语句：

对于线程 1，start_execute_thread() 函数激活连接 con1，执行与 con1 相关联的 s1 准备语句，并使 con1 休眠。

对于线程 2，start_execute_thread() 函数激活连接 con2，执行与 con2 相关联的 s2 准备语句，并使 con2 休眠。

主线程断开所有连接。

跨线程游标

如同准备的语句一样，游标在连接级别被限定。也就是说，它们与连接相关联。当线程使连接处于活动状态时，它可以访问此连接声明的任何数据库游标。如果线程安全的 GBase 8s ESQL/C 应用程序使用数据库游标，则可能需要以与隔离预备语句相同的方式隔离游标声明。以下代码片段显示了 start_con_thread() 函数的修改版本（在图 1中）。本版

本准备 SQL 语句，并为此语句声明游标：

```
EXEC SQL connect to 'db1' as 'con1';
        EXEC SQL prepare s1 ....
        EXEC SQL declare cursor cursor1 for s1;
        EXEC SQL set connection 'con1' dormant;
```

跨线程环境变量

在 GBase 8s ESQL/C线程安全应用程序中，环境变量不是线程作用域的。即，如果线程更改特定环境变量的值，则此更改也可在所有其它线程中显示。

消息文件描述符

缺省情况下，ESQL/C 会在关闭消息文件时释放消息文件的所有的文件描述符。但是，作为性能优化，您可以设置环境变量 IFX_FREE_FD ，以使如果正在关闭的消息文件对另一线程打开，GBase 8s ESQL/C 不会释放文件描述符。如果将 IFX_FREE_FD 设置为 1，则 GBase 8s ESQL/C 仅为关闭文件的线程释放消息文件描述符。

Decimal 函数

GBase 8s ESQL/C 库的 dececvt() 和 decfcvt() 函数返回一个字符串，如果两个线程同时调用这些函数则可以覆盖此字符串。出于此原因，请使用这些函数的线程安全版本 ifx_dececvt() 和 ifx_decfcvt()。

DCE 限制（UNIX™）

线程安全的 GBase 8s ESQL/C 代码也适用于 DCE 线程包实现的所有限制。DCE 要求所有使用 DCE 线程库的的应用程序符合 ANSI 标准。本节列出了一些限制，当创建 GBase 8s ESQL/C 应用程序请记住这些限制。

操作系统调用

必须将线程安全的 GBase 8s ESQL/C 应用程序中的所有操作系统调用替换为 DCE 线程套件程序。线程套接字符采用系统调用的名称，但是在调用事件系统服务之前，它们将调用 DCE pthread_lock_global_np() 函数来锁定全局互斥锁（mutex）。（ Mutexes 通过确保一次只有一个线程执行代码的关键部分来按顺序执行线程。DCE 包含文件 pthread.h 定义了系统调用的套接版本。

fork() 操作系统调用

在 DCE 环境中，限制使用 fork() 操作系统调用。一般来说，在调用 fork() 之前终止所有线程。此规则的一个例外是当 fork() 系统调用紧跟在 fork() 调用之后。如果您的应用程序使用 fork()，则建议子进程在执行任何 GBase 8s ESQL/C 语句之前调用 sqldetach()。

资源分配

建议您将紧急循环中的 DCE pthread_yield() 调用包含在内，以体现调度程序更具需要分配资源。对 pthread_yield() 的调用指示 DCE 调用程序尝试统一分配资源。如果一个线程陷入紧密循环，等待连接（从而阻止其他线程进行）。以下代码片段显示 pthread_yield() 例程的调用：

```
/* loop until the connection is available*/
do
    {
    EXEC SQL set connection   :con_name;
    pthread_yield();
    } while (SQLCODE == -1802);
```

链接线程安全库

当指定 **-thread** 命令行选项时，esql 命令链接静态或共享的线程安全版本的 GBase 8s 通用库。

UNIX 系统上将线程安全 GBase 8s 通用库链接到 ESQL/C 模块

在 UNIX™ 操作系统上，执行以下步骤来将线程安全 GBase 8s 通用库链接到 GBase 8s ESQL/C 模块：

将 DCE 线程包安装到 GBase 8s ESQL/C 产品的同一客户端的计算机上。

如果 DCE 不可用于您的平台，则 ESQL/C 可以使用 POSIX 线程库或 Sun Solaris 线程库。

设置 THREADLIB 环境变量以指示当编译程序时使用哪个线程包。

以下 C-shell 命令将 THREADLIB 设置为 DCE 线程包：

```
setenv THREADLIB DCE
```

SOL 和 POSIX 也是 THREADLIB 环境变量的有效选项。

重要： 此版本的 GBase 8s ESQL/C 仅支持 DCE 线程包。

使用 esql 命令编译您的程序，并指定 -thread 命令行选项。

-thread 命令行选项告诉 esql 生成线程安全代码并在线程安全库中链接。以下命令将线程安全共享库与 file.exe 可执行文件相链接：

```
esql -thread file.ec -o file.exe
```

-thread 命令行选项说明 esql 命令以执行以下步骤：

将 **-thread** 选项发送给 GBase 8s ESQL/C 预处理器来生成线程安全代码。

使用 -DIFX_THREAD 命令行选项调用 C 编译器。

将合适的线程库（共享或静态）链接到可执行文件。

提示： 比较在使用带有 **-thread** 命令行选项的 esql 命令前设置 THREADLIB 环境变量。

如果指定 **-thread** 选项但不设置 THREADLIB，或者如果将 THREADLIB 设置为某些不支持的线程包，则 esql 命令发出下列消息：

```
esql: When using -thread, the THREADLIB environment variable
            must be set to a supported thread library. Currently
            supporting: DCE, POSIX(Solaris 2.5 and higher only) and
            SOL (Solaris Kernel Threads)
```

定义线程安全变量（UNIX™）

当指定 esql 的 **-thread** 命令行选项时，GBase 8s ESQL/C 预处理器将 IFX_THREAD 定义传递给 C 编译器。IFX_THREAD 定义告诉 C 编译器为非线程安全的 GBase 8s ESQL/C 代码创建线程范围变量。

例如，当 C 编译器包含设置了 IFX_THREAD 的 sqlca.h 文件时，它为 GBase 8s ESQL/C 状态变量定义线程范围变量：SQLCODE 、SQLSTATE 和 **sqlca** 结构。线程范围版本的状态变量是将全局状态变量映射到线程安全函数调用的宏。可以获取特定于线程的状态信息。

下图显示了具有 GBase 8s ESQL/C 状态变量的线程范围定义的 sqlca.h 文件的摘录。

图: 线程范围状态变量的声明

```
⋮

        extern struct sqlca_s sqlca;

        extern int4 SQLCODE;

        extern char SQLSTATE[];
        #else /* IFX_THREAD */
        extern int4 * ifx_sqlcode();
        extern struct sqlca_s * ifx_sqlca();
        #define SQLCODE (*(ifx_sqlcode()))
        #define SQLSTATE ((char *)(ifx_sqlstate()))
        #define sqlca (*(ifx_sqlca()))
        #endif /* IFX_THREAD */
```

链接共享或静态版本

要告诉 esql 命令将 GBase 8s 库的线程安全版本链接到应用程序中，请使用 esql 的 **-thread** 命令行选项。如下所示：

线程安全共享库仅需要 -thread 命令行选项。

线程安全静态库需要 -thread 和 -static 命令行选项。

将线程安全的 GBase 8s 通用库链接到 Windows 环境中的 ESQL/C 模块

要创建线程安全的 GBase 8s ESQL/C 应用程序，您必须执行以下步骤：

在您的 GBase 8s ESQL/C 源文件中，包括适当的线程函数和 Windows™ API 的变量。

当编译 GBase 8s ESQL/C 源文件时，指定 esql 命令的 -thread 命令行选项。

-thread 选项告诉 GBase 8s ESQL/C 预处理器当它翻译 SQL 和 GBase 8s ESQL/C 语句时，生成线程安全 C 代码。此线程安全代码包含 GBase 8s DLL 中线程安全函数的调用。

如果您没有创建具有线程的 GBase 8s ESQL/C 应用程序，则忽略 **-thread** 选项。尽管 GBase 8s DLL 是线程安全的，但是当您忽略 **-thread** 时，您的非线程安全应用程序不

能使用线程安全功能。

3.3.4 ESQL/C 线程安全 decimal 函数

GBase 8s ESQL/C 库的 dececvt() 和 decfcvt() 函数返回一个字符串，如果两个线程同时调用这些函数，该字符串可以被覆盖。因此，请使用以下两个函数的线程安全版本。

函数名称	描述	请参阅
ifx_dececvt()	dececvt() ESQL/C 函数的线程安全版本	ifx_dececvt() 和 ifx_decfcvt() 函数
ifx_decfcvt()	decfcvt() ESQL/C 函数的线程安全版本	ifx_dececvt() 和 ifx_decfcvt() 函数

这些函数都可以将十进制值转换为 ASCII 字符串，并在用户定义的缓冲区内返回它。

当使用 esql 命令的 **-thread** 选项编译您的 GBase 8s ESQL/C 程序时，esql 字段将这些函数链接到您的程序。

3.3.5 上下文线程优化

GBase 8s ESQL/C 运行开发者指定用于一组语句的运行时上下文。运行上下文保存 GBase 8s ESQL/C 必须维护的所有特定于线程的数据，包括当前状态和游标。

此功能允许 GBase 8s ESQL/C 程序员能够提高其 MESQL/C 应用程序的性能。通过使用 SQLCONTEXT 定义和指令，减少了线程特定数据块查找的数量。

以下嵌入式 SQL 语句支持运行上下文的定义和用法：

```
SQLCONTEXT context_var;
                PARAMETER SQLCONTEXT param_context_var;
                BEGIN SQLCONTEXT OPTIMIZATION;
                END SQLCONTEXT OPTIMIZATION;
```

仅在使用 esql-thread 选项时才能识别 SQLCONTEXT 定义和语句。如果未指定 **-thread** 选项，则忽略此语句。

SQLCONTEXT 语句的用法导致 ESQL/C 预处理器在 .c 文件中生成的代码与不存在 SQLCONTEXT 语句时生成的代码的不同。

下列 SQLCONTEXT 定义生成代码来定义并将 SQLCONTEXT 的值设置为运行上下文的句柄：

```
SQLCONTEXT context_var;
```

以下 SQLCONTEXT 用于生成代码来定义包含运行上下文句柄的参数：

```
PARAMETER SQLCONTEXT param_context_var;
```

以下 BEGIN SQLCONTEXT 指令使源文件中的所有语句都位于其中，以使用指定的运行时上下文，直到看到 END CONTEXT 指令：

```
BEGIN SQLCONTEXT OPTIMIZATION;
                    ...
                    END SQLCONTEXT OPTIMIZATION;
```

END SQLCONTEXT 指令出现在当前使用的 SQLTCONTEXT 定义的范围之前，或者发生“undefined symbol sql_context_var”错误编译时。

3.3.6 线程安全程序示例

以下示例程序 **thread_safe** 显示如何跨线程使用游标。此程序的示例输出遵循此源列表。

源列表

主线程开始名为 **con1** 的临界，并在表 **t** 上声明一个游标。然后打开游标并使 **con1** 连接休眠。主线程然后启动六个线程（threads_all() 函数的六个实例），调用 pthread_join() DCE 等待线程完成它们的工作。

每个线程使用连接 **con1**，打开的游标来执行获取操作。在获取操作之后，程序使连接休眠。线程以顺序方式使用连接 **con1**，因为一次只能有一个线程可以使用函数，每个线程从 **t** 表读取下一个记录。

```
/* ************************************************************
* Program Name: thread_safe()
*
* purpose       : If a server connection is initiated with the WITH
*                 CONCURRENT TRANSACTION clause, an ongoing transaction
*                 can be shared across threads that subsequently
*                 connect to that server.
*                 In addition, if an open cursor is associated with such
*                 connection, the cursor will remain open when the
*                 connection
*                  is made dormant. Therefore, multiple threads can share a
*                 cursor.
*
* Methods       : - Create database db_con221 and table t1.
*                 - Insert 6 rows into table t1, i.e. values 1 through 6.
*                 - Connect to db_con221 as con1 with CONCURRENT
*                     TRANSACTION.
*                       The CONCURRENT TRANSACTION option is required
since
*                       all
*                       threads use the cursor throughout the same
*                       connection.
*                 - Declare c1 cursor for "select a from t1 order by a".
*                 - Open the cursor.
*                 - Start 6 threads. Use DCE pthread_join() to determine if
```

```
*                       all threads complete & all threads do same thing as
*                       follows.
*                       For thread_1, thread_2, ..., thread_6:
*                               o SET CONNECTION con1
*                               o FETCH a record and display it.
*                               o SET CONNECTION con1 DORMANT
*                   - Disconnect all connections.
****************************************************************
*/

#include <pthread.h>
#include <dce/dce_error.h>

/* global definitions */
#define num_threads        6

/* Function definitions */
static   void thread_all();
static   long   dr_dbs();
static int checksql(char *str, long expected_err, char *con_name);
static void dce_err();

/* Host variables declaration */
EXEC SQL BEGIN DECLARE SECTION;
    char     con1[] = "con1";
EXEC SQL END DECLARE SECTION;

/* ************************************************
* Main Thread
************************************************/
main()
{
/* create database */

EXEC SQL create database db_con221 with log;
if (! checksql("create database", 0, EMPTYSTR))
    {
    printf("MAIN:: create database returned status {%d}\n", SQLCODE);
    exit(1);
    }

EXEC SQL create table t1( sales int);
if (! checksql( "create_table", 0, EMPTYSTR))
    {
    dr_dbs("db_con221");
    printf("MAIN:: create table returned status {%d}\n", SQLCODE);
    exit(1);
    }
```

```
    if ( populate_tab() != FUNCSUCC)
        {
        dr_dbs("db_con221");
        printf("MAIN:: returned status {%d}\n", SQLCODE);
        exit(1);
        }

    EXEC SQL close database;
    checksql("[main] <close database>", 0, EMPTYSTR);

    /* Establish connection 'con1' */
    EXEC SQL connect to 'db_con221' as 'con1' WITH CONCURRENT
TRANSACTION;
    if (! checksql("MAIN:: <close database>", 0, EMPTYSTR))
         {
         dr_dbs("db_con221");
         exit(1);
         }

    /* Declare cursor c1 associated with the connection con1 */
    EXEC SQL prepare tabid from "select sales from t1 order by sales";
    checksql("MAIN:: <prepare>", 0, EMPTYSTR);

    EXEC SQL declare c1 cursor for tabid;
    checksql("MAIN:: <declare c1 cursor for>", 0, EMPTYSTR);

    /* Open cursor c1 and make the connection dormant */
    EXEC SQL open c1;
    checksql("MAIN:: <open c1>", 0, EMPTYSTR);
    EXEC SQL set connection :con1 dormant;
    checksql("MAIN:: <set connection con1 dormant>", 0, EMPTYSTR);

    /* Start threads    */
    start_threads();

    /* Close cursor and drop database */
    EXEC SQL set connection :con1;
    checksql("MAIN:: set connection", 0, EMPTYSTR);
    EXEC SQL close c1;
    checksql("MAIN:: <close cursor>", 0, EMPTYSTR);
    EXEC SQL free c1;
    checksql("MAIN:: <free cursor>", 0, EMPTYSTR);

    EXEC SQL disconnect all;
    checksql("MAIN:: disconnect all", 0, EMPTYSTR);
    dr_dbs("db_con221");
    } /* end of Main Thread */

    /*********************************************************************
```

```
* Function: thread_all()
 * Purpose : Uses connection con1 and fetches a row from table t1 using *
        cursor c1.
 * Returns : Nothing
*************************************************************************/
static void thread_all(thread_num)
int *thread_num;
{
EXEC SQL BEGIN DECLARE SECTION;
    int       val;
EXEC SQL END DECLARE SECTION;

/* Wait for the connection to become available */
do {
    EXEC SQL set connection :con1;
    } while (SQLCODE == -1802);

checksql("thread_all: set connection", 0, con1);

/* Fetch a row */
EXEC SQL fetch c1 into :val;
checksql("thread_all: fetch c1 into :val", 0, con1);

/* Free connection con1 */
EXEC SQL set connection :con1 dormant;
checksql("thread_all: set connection con1 dormant", 0, EMPTYSTR);
printf("Thread id %d fetched value %d from t1\n", *thread_num, val);
} /* thread_all() */

/************************************************************************
* Function: start_threads()
 * purpose : Create num_threads and passes a thread id number to each
 *                      thread
*************************************************************************/

start_threads()
{

    int                thread_num[num_threads];
    pthread_t                thread_id[num_threads];
    int              i, ret, return_value;

for(i=0; i< num_threads; i++)
    {
    thread_num[i] = i;
    if ((pthread_create(&thread_id[i], pthread_attr_default
            (pthread_startroutine_t ) thread_all, &thread_num[i])) == -1)
        {
        dce_err(__FILE__, "pthread_create failed", (unsigned long)-1);
```

```
            dr_dbs("db_con221");
            exit(1);
            }
        }

     /* Wait for all the threads to complete their work */
    for(i=0; i< num_threads; i++)
        {
        ret = pthread_join(thread_id[i], (pthread_addr_t *) &return_value);
        if(ret == -1)
            {
            dce_err(__FILE__, "pthread_join", (unsigned long)-1);
            dr_dbs("db_con221");
            exit(1);
            }
        }
    } /* start_threads() */

    /*****************************************************************************
    * Function: populate_tab()
     * Purpose : insert values in table t1.
     * Returns : FUNCSUCC on success and FUNCFAIL when it fails.
    *****************************************************************************/
    static int
    populate_tab()
    {
    EXEC SQL BEGIN DECLARE SECTION;
        int i;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL begin work;
    if (!checksql("begin work", 0,EMPTYSTR))
        return FUNCFAIL;
    for (i=1; i<=num_threads; i++)
        {
        EXEC SQL insert into t1 values (:i);
          if(!checksql("insert", 0,EMPTYSTR))
            return FUNCFAIL;
        }
    EXEC SQL commit work;
    if (!checksql("commit work", 0,EMPTYSTR))
            return FUNCFAIL;

    return FUNCSUCC;
    } /* populate_tab() */

    /*****************************************************************************
     * Function: dr_dbs()
     * Purpose : drops the database.
```

```
**************************************************************************/
long dr_dbs(db_name)
EXEC SQL BEGIN DECLARE SECTION;
    char     *db_name;
EXEC SQL END DECLARE SECTION;

{
EXEC SQL connect to DEFAULT;
checksql("dr_dbs: connect",    0, "DEFAULT");

EXEC SQL drop database :db_name;
checksql("dr_dbs: drop database", 0, EMPTYSTR);

EXEC SQL disconnect all;
checksql("dr_dbs: disconnect all", 0, EMPTYSTR);
} /*dr_dbs() */

/**************************************************************************
 * Function: checksql()
 * Purpose : To check the SQLCODE against the expected error
 *                          (or the expected SQLCODE) and issue a proper message.
 * Returns : FUNCSUCC on success & FUNCFAIL on FAILURE.
**************************************************************************/
int checksql(str, expected_err, con_name)
char *str;
long expected_err;
char *con_name;
{
if (SQLCODE != expected_err)
    {
      printf( "%s %s Returned {%d}, Expected {%d}\n", str, con_name,
SQLCODE,
          expected_err);
    return(FUNCFAIL);
    }
return (FUNCSUCC);
} /* checksql() */

/**************************************************************************
* Function: dce_err()
 * purpose : prints error from dce lib routines
 * return    : nothing
**************************************************************************/

void dce_err(program, routine, status)
char *program, *routine;
unsigned long status;
{
int dce_err_status;
```

```
char dce_err_string[dce_c_error_string_len+1];

if(status == (unsigned long)-1)
    {
    dce_err_status = 0;
    sprintf(dce_err_string, "returned FAILURE (errno is %d)", errno);
    }
else
    dce_error_inq_text(status, (unsigned char *)dce_err_string,
&dce_err_status);

if(!dce_err_status)
    {
     fprintf(stderr, "%s: error in %s:\n     ==> %s (%lu) <==\n",
     program, routine, dce_err_string, status);
     }
else
     fprintf(stderr, "%s: error in %s: %lu\n", program, routine, status);
} /* dce_err() */
```

输出

每次执行示例程序时，示例输出可能会出现不同，因为它取决于线程的执行顺序。

```
Thread id 0 fetched value 1 from t1
                Thread id 2 fetched value 2 from t1
                Thread id 3 fetched value 3 from t1
                Thread id 4 fetched value 4 from t1
                Thread id 5 fetched value 5 from t1
                Thread id 1 fetched value 6 from t1
```

在此输出中，线程 1 获取表中最后一条记录。

3.3.7 在 UNIX™ 操作系统上创建动态线程库

要创建动态线程库，您必须为 GBase 8s ESQL/C 执行的每个线程操作定义例程，并且使用 GBase 8s ESQL/C 注册这些函数。以下列表显示了多线程 GBase 8s ESQL/C 应用程序所需的所有函数，并描述了每个函数必须执行的操作。

mint ifxOS_th_once(ifxOS_th_once_t *pblock, ifxOS_th_initroutine_t pfn, int *init_data)

此例程执行初始化 pfn()。即使在同一线程中同时被多个线程同时调用多次，也只执行一次 pfn() 函数。 pfn() 例程相当于 DCE pthread_once() 或 POSIX pthread_once() 例程。

init_data 变量用于没有 pthread_once() 类型例程的线程包，例如 Solaris Kernel 线程。该例程可以通过使用 **init_data** 作为初始化为 0 的全局变量来模拟该例程。

```
if (!*init_data)
                    {
                    mutex_lock(pblock);
                    if (!*init_data)
                    {
                    (*pfn)();
```

```
                    *init_data = 1;
                    }
                    mutex_unlock(pblock);
                    }
                    return(0);
```

mint ifxOS_th_mutexattr_create(ifxOS_th_mutexattr_t *mutex_attr

该函数创建一个互斥属性对象，它们在创建互斥时指定互斥体的属性。互斥属性对象由用户实现定义的所有属性的缺省值初始化。该例程相当于 DCE pthread_mutexattr_create() 或 POSIX pthread_mutexattr_init() 例程。如果线程包不支持互斥属性对象，则互斥属性例程可以是 no-ops。

mint ifxOS_th_mutexattr_setkind_np(ifxOS_th_mutexattr_t *mutex_attr, int kind)

该例程在创建互斥对象时，使用的互斥类型属性。此互斥属性 mutex_attr 设置为类型 **kind**。对于 DCE，该例程是 pthread_mutexattr_setkind_np()。

mint ifxOS_th_mutexattr_delete(ifxOS_th_mutexattr_t *mutex_attr)

该例程删除互斥属性对象 mutex_attr。该例程具有与 DCE pthread_mutexattr_delete() 或 POSIX pthread_mutexattr_destroy() 例程相同的功能。

mint ifxOS_th_mutex_init(ifxOS_th_mutex_t *mutexp, ifxOS_th_mutexattr_t mutex_attr)

该例程创建互斥并将它初始化为未锁定的状态。此例程具有与 DCE pthread_mutex_init() 或 POSIX pthread_mutex_init() 例程相同的功能。

mint ifxOS_th_mutex_destroy(ifxOS_th_mutex_t *mutexp)

此例程删除互斥对象。此互斥对象在其删除前必须未锁定。此例程具有与 DCE pthread_mutex_destroy() 或 POSIX pthread_mutex_destroy() 例程相同的功能。

mint ifxOS_th_mutex_lock(ifxOS_th_mutex_t *mutexp)

该例程锁定为锁定的互斥对象。如果互斥对象已经被锁定，则此调用线程等待指定互斥对象变为未锁定状态。该例程具有与 DCE pthread_mutex_lock() 或 POSIX pthread_mutex_lock() 例程相同的功能。

mint ifxOS_th_mutex_trylock(ifxOS_th_mutex_t *mutexp)

如果互斥对象成功被锁定，则它返回值 1，如果互斥对象由另一个线程锁定，则它返回值 0。

该例程具有与 DCE pthread_mutex_trylock() 例程相同的功能。

mint ifxOS_th_mutex_unlock(ifxOS_th_mutex_t *mutexp)

该例程解锁互斥 mutexp。如果线程等锁定此互斥，则该实现定义哪个线程接收到互斥。该例程具有与 DCE pthread_mutex_unlock() 或 POSIX pthread_mutex_unlock() 例程相同的功能。

mint ifxOS_th_condattr_create(ifxOS_th_condattr_t *cond_attr)

该例程创建一个对象，用于指定条件变量创建时指定其属性。使用用户实现定义的所

有属性的缺省值初始化对象。该例程具有与 DCE pthread_condattr_create() 或 POSIX pthread_condattr_init() 例程相同的功能。

mint ifxOS_th_cond_init(ifxOS_th_cond_t *condp, ifxOS_th_condattr_t cond_attr)

该例程创建并初始化集合变量。该例程具有与 DCE pthread_cond_init() 或 POSIX pthread_cond_init() 例程相同的功能。

mint ifxOS_th_condattr_delete(ifxOS_th_condattr_t *cond_attr)

该例程删除条件变量属性对象 cond_attr。该例程具有与 DCE pthread_condattr_delete() 或 POSIX pthread_condattr_destroy() 例程相同的功能。

mint ifxOS_th_cond_destroy(ifxOS_th_cond_t *condp)

该例程删除条件变量 condp。该例程具有与 DCE pthread_cond_destroy() 或 POSIX pthread_cond_destroy() 例程相同的功能。

mint ifxOS_th_cond_timedwait(ifxOS_th_cond_t *sleep_cond, ifxOS_th_mutex_t *sleep_mutex, ifxOS_th_timespec_t *t)

该例程导致线程等待直到状态变量 sleep_cond 发出信号或广播，或者当前系统时钟时间变得大于或等于 **t** 中指定的时间。该例程具有与 DCE pthread_cond_timedwait() 或 POSIX pthread_cond_timedwait() 例程相同的功能。

mint ifxOS_th_keycreate(ifxOS_th_key_t *allkey, ifxOS_th_destructor_t AllDestructor)

该例程生成标识特定于线程的数据值的唯一值。该例程具有与 DCE pthread_keycreate() 或 POSIX pthread_key_create() 例程相同的功能。

mint ifxOS_th_getspecific(ifxOS_th_key_t key, ifxOS_th_addr_t *tcb)

该例程获取与键相关联的线程特定数据。该例程具有与 DCE pthread_getspecific() 或 POSIX pthread_getspecific() 例程相同的功能。

mint ifxOS_th_setspecific(ifxOS_th_key_t key, ifxOS_th_addr_t tcb)

该例程设置于当前线程的**键**相关联的 **tcb** 中的线程的特定数据。如果已经为当前线程中的**键**定义了一个值，则将替换现有值的新值。该例程具有与 DCE pthread_setspecific() 或 POSIX pthread_setspecific() 例程相同的功能。

数据类型

可以为上述函数中的数据类型为线程程序包中的等效数据类型创建 typedefs ，或者可以还有线程程序包而不是 **ifxOS_** 版本中的相应的数据类型。以下列表包括上述函数使用的所有数据类型：

ifxOS_th_mutex_t

此结构在 DCE 和 POSIX 中定义互斥对象：**pthread_mutex_t**。

ifxOS_th_mutexattr_t

该结构在 DCE 和 POSIX 中定义了一个名为 **pthread_mutexattr_t** 的互斥属性

对象。如果线程包（例如，Solaris 核心线程）中不支持互斥属性，则可以将它们指定为 **mint** 数据类型。

ifxOS_th_once_t

该结构允许客户端初始化操作保证对初始化例程的互斥访问，并保证每次初始化只执行一次。该例程与 DCE 和 POSIX 中的 **pthread_once_t** 结构具有相同的功能。

ifxOS_th_condattr_t

该结构定义了一个对象，它指定 DCE 和 POSIX 中一个条件变量 **pthread_condattr_t** 的属性。如果线程包（例如，Solaris 核心线程）中不支持此对象，则可以为其分配为 **mint** 数据类型。

ifxOS_th_cond_t

该结构定义了 DCE 和 POSIX 中的名为 **pthread_cond_t** 的条件变量。

ifxOS_th_timespec_t

该结构定义 ifxOS_th_cond_timedwait() 函数在条件变量未发出信号或广播时，超时的绝对时间。该结构是在 DCE 和 POSIX 中的 **timespec_t**。

ifxOS_th_key_t

该结构定义特定于线程键，在 ifxOS_th_keycreate() 、ifxOS_th_setspecific() 和 ifxOS_getspecific() 例程中使用。该结构是在 DCE 和 POSIX 中的 **pthread_key_t**。

ifxOS_th_addr_t

该结构定义包含要与 **ifxOS_th_key_t** 类型特定于线程的数据键相关联的数据的地址。**ifxOS_th_addr_t** 结构等同于 DCE 中的 **pthread_addr_t**。可以指定 **void *** 作为可用于不定义此类结构的线程包（如 POSIX）的替代方法。

以下示例使用 Solaris 核心线程演示如何设置动态线程库。第一个任务是定义共享和/或静态库所需的 17 个动态线程函数。在此示例中，该文件称为 dynthr.c：

```
/* Prototypes for the dynamic thread functions */

mint ifx_th_once(mutex_t *pblock, void (*pfn)(void), mint *init_data);
mint ifx_th_mutexattr_create(mint *mutex_attr);
mint ifx_th_mutexattr_setkind_np(mint *mutex_attr, mint kind);
mint ifx_th_mutexattr_delete(mint *mutex_attr);
mint ifx_th_mutex_init(mutex_t *mutexp, mint mutex_attr);
mint ifx_th_mutex_destroy(mutex_t *mutexp);
mint ifx_th_mutex_lock(mutex_t *mutexp);
mint ifx_th_mutex_trylock(mutex_t *mutexp);
mint ifx_th_mutex_unlock(mutex_t *mutexp);
mint ifx_th_condattr_create(mint *cond_attr);
mint ifx_th_cond_init(cond_t *condp, mint cond_attr);
mint ifx_th_condattr_delete(mint *cond_attr);
mint ifx_th_cond_destroy(cond_t *condp);
mint ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t *sleep_mutex,
```

```
    timestruc_t *t);
mint ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)
    (void *));
mint ifx_th_getspecific(thread_key_t key, void **tcb);
mint ifx_th_setspecific(thread_key_t key, void *tcb);

/*
 * The functions . . . *
 *                          */

mint ifx_th_once(mutex_t *pblock, void (*pfn)(void), mint *init_data)
{
    if (!*init_data)
    {
        mutex_lock(pblock);
        if (!*init_data)
        {
            (*pfn)();
            *init_data = 1;
        }
        mutex_unlock(pblock);
    }
    return(0);
}

/* Mutex attributes are not supported in solaris kernel threads *
* The functions must be defined anyway, to avoid accessing      *
* a NULL function pointer.                                       */

mint ifx_th_mutexattr_create(mint *mutex_attr)
{
    *mutex_attr = 0;
    return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
mint ifx_th_mutexattr_setkind_np(mint *mutex_attr, mint kind)
{
    *mutex_attr = 0;
    return(0);
}

/* Mutex attributes are not supported in solaris kernel threads */
mint ifx_th_mutexattr_delete(mint *mutex_attr)
{
    return(0);
}

mint ifx_th_mutex_init(mutex_t *mutexp, mint mutex_attr)
```

```
{
    return(mutex_init(mutexp, USYNC_THREAD, (void *)NULL));
}

mint ifx_th_mutex_destroy(mutex_t *mutexp)
{
    return(mutex_destroy(mutexp));
}

mint ifx_th_mutex_lock(mutex_t *mutexp)
{
    return(mutex_lock(mutexp));
}
/* Simulate mutex_trylock using mutex_lock */
mint ifx_th_mutex_trylock(mutex_t *mutexp)
{
    mint ret;

    ret = mutex_trylock(mutexp);
    if (ret == 0)
        return(1); /* as per the DCE call */
    if (ret == EBUSY)
        return(0); /* as per the DCE call */
    return(ret);
}

mint ifx_th_mutex_unlock(mutex_t *mutexp)
{
    return(mutex_unlock(mutexp));
}

/* Condition attributes are not supported in solaris kernel threads */
mint ifx_th_condattr_create(mint *cond_attr)
{
    *cond_attr = 0;
    return(0);
}

mint ifx_th_cond_init(cond_t *condp, mint cond_attr)
{
    return(cond_init(condp, USYNC_THREAD, (void *)NULL));
}

mint ifx_th_condattr_delete(int *cond_attr)
{
    return(0);
}

mint ifx_th_cond_destroy(cond_t *condp)
```

```
{
    return(cond_destroy(condp));
}

mint ifx_th_cond_timedwait(cond_t *sleep_cond, mutex_t
    *sleep_mutex,timestruc_t
    *t)
{
    return(cond_timedwait(sleep_cond, sleep_mutex, t));
}

mint ifx_th_keycreate(thread_key_t *allkey, void (*AllDestructor)
    (void *))
{
    return(thr_keycreate(allkey, AllDestructor));
}

mint ifx_th_getspecific(thread_key_t key, void **tcb)
{
    return(thr_getspecific(key, tcb));
}

mint ifx_th_setspecific(thread_key_t key, void *tcb)
{
    return(thr_setspecific(key, tcb));
```

注册动态线程函数

GBase 8s ESQL/C 应用程序必须使用 ifxOS_set_thrfunc() 函数注册 GBase 8s ESQL/C 动态线程函数f。

下列声明描述 ifxOS_set_thrfunc() 函数。

```
mint ifxOS_set_thrfunc(mint func, mulong (*funcptr)())
```

第一个参数 **func** 是 **mint**，索引被注册的函数。第二个参数是要注册的函数的名称。

对于在在 UNIX 操作系统上创建动态线程库中列出的 17 个 **ifxOS** 函数，必须调用 ifxOS_set_thrfunc() 一次。

如果 ifxOS_set_thrfunc() 函数成功注册函数，则返回 0，如果失败则返回 -1。例如，要将用户定义的函数 my_mutex_lock() 注册为 **ifxOS_th_mutex_lock** 例程，请使用以下调用：

```
if (ifxOS_set_thrfunc(TH_MUTEX_LOCK, (mulong (*)())my_mutex_lock)
            == -1)
```

TH_MUTEX_LOCK 在 sqlhdr.h 中定义，它告诉客户端在它需要锁定互斥时调用 **my_mutex_lock()** 。

以下列表显示了它们注册的索引及其函数。

索引

函数

TH_ONCE

ifxOS_th_once

TH_MUTEXATTR_CREATE

ifxOS_th_mutexattr_create()

TH_MUTEXATTR_SETKIND

ifxOS_th_mutexattr_setkind_np()

TH_MUTEXATTR_DELETE

ifxOS_th_mutexattr_delete()

TH_MUTEX_INIT

ifxOS_th_mutex_init()

TH_MUTEX_DESTROY

ifxOS_th_mutex_destroy()

TH_MUTEX_LOCK

ifxOS_th_mutex_lock()

TH_MUTEX_UNLOCK

ifxOS_th_mutex_unlock()

TH_MUTEX_TRYLOCK

ifxOS_th_mutex_trylock()

TH_CONDATTR_CREATE

ifxOS_th_condattr_create()

TH_CONDATTR_DELETE

ifxOS_th_condattr_delete()

TH_COND_INIT

ifxOS_th_cond_init()

TH_COND_DESTROY

ifxOS_th_cond_destroy()

TH_COND_TIMEDWAIT

ifxOS_th_cond_timedwait()

TH_KEYCREATE

ifxOS_th_keycreate()

TH_GETSPECIFIC

ifxOS_th_getspecific()

TH_SETSPECIFIC

ifxOS_th_setspecific()

下列函数 **dynthr_init()**，也在 **dynthr.c** 中定义，注册了在 UNIX 操作系统上创建动态线程库中定义的 17 个函数。FUNCFAIL 定义为 -1。

```
dynthr_init()
{
    if (ifxOS_set_thrfunc(TH_ONCE, (mulong (*)())ifx_th_once)
== FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_CREATE,
        (mulong (*)())ifx_th_mutexattr_create) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_SETKIND,
        (mulong (*)())ifx_th_mutexattr_setkind_np) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEXATTR_DELETE,
        (mulong (*)())ifx_th_mutexattr_delete) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_INIT,
        (mulong (*)())ifx_th_mutex_init) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_DESTROY,
        (mulong (*)()) ifx_th_mutex_destroy) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_LOCK,
        (mulong (*)()) ifx_th_mutex_lock) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_UNLOCK,
        (mulong (*)())ifx_th_mutex_unlock) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_MUTEX_TRYLOCK,
        (mulong (*)())ifx_th_mutex_trylock) == FUNCFAIL)
            return FUNCFAIL;
    if (ifxOS_set_thrfunc(TH_CONDATTR_CREATE,
        (mulong (*)())ifx_th_condattr_create) == FUNCFAIL)
            return FUNCFAIL;
```

```
    if (ifxOS_set_thrfunc(TH_CONDATTR_DELETE,
        (mulong (*)())ifx_th_condattr_delete) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_INIT,
        (mulong (*)())ifx_th_cond_init) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_COND_DESTROY,
        (mulong (*)())ifx_th_cond_destroy) == FUNCFAIL)
            return FUNCFAIL;
    if (ifxOS_set_thrfunc(TH_COND_TIMEDWAIT,
        (mulong (*)())ifx_th_cond_timedwait) == FUNCFAIL)
        return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_KEYCREATE,
        (mulong (*)())ifx_th_keycreate) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_GETSPECIFIC,
        (mulong (*)())ifx_th_getspecific) == FUNCFAIL)
            return FUNCFAIL;

    if (ifxOS_set_thrfunc(TH_SETSPECIFIC,
        (mulong (*)())ifx_th_setspecific) == FUNCFAIL)
            return FUNCFAIL;
    return 0;

}
```

设置 $THREADLIB 环境变量

以下 C-shell 命令设置 THREADLIB 环境变量以指定用户定义线程包：

```
setenv THREADLIB DYNAMIC
```

创建共享库

必须将 dynthr.c 编译到共享或静态库汇总。以下示例说明如何在运行 Solaris 操作系统的工作站上编译共享或静态库：

```
% cc -c -DIFX_THREAD -I$GBASEDBTDIR/incl/esql -D_REENTRANT -K pic
        dynthr.c
        % ld -G -o libdynthr.so dynthr.o
        % cp libdynthr.so /usr/lib              <== as root
```

还可以使用 $LD_LIBRARY_PATH 环境变量：

```
% cc -c -DIFX_THREAD -I$GBASEDBTDIR/incl/esql -D_REENTRANT -K pic
        dynthr.c
        % cp dynthr.so <some directory>
        % setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:<some
directory>
```

要将 dynthr.c 编译到静态库，请执行以下操作（在 Solaris 上）：

```
% cc -c -DIFX_THREAD -I$GBASEDBTDIR/incl/esql -D_REENTRANT dynthr.c

        % ar -cr dynthr.a dynthr.o
```

初次调用 dynthr_init() 例程，或线程函数都没有注册时，必须更改您的应用程序 test.ec 。

```
void main(argc , argv )
        int argc;
        char *argv[] ;
        {       /* begin main */

        /* First, set up the dynamic thread library */

        dynthr_init();

        /* Rest of program */

        EXEC SQL database stores7;

        :

        }
```

使用 -thread 和 -l 预处理器选项编译

必须使用 **-thread** 和 **-l** 预处理器选项编译应用程序。

-thread 选项指示您正在链接线程安全库而不是缺省的 GBase 8s 共享库。**-l** 选项允许您指定想要链接的系统库。最终，编译应用程序时，链接 libdynthr.so 并运行它，如以下示例所示：

```
% setenv THREADLIB "dynamic"
        % esql -thread -ldynthr test.ec -o test.exe
        % test.exe
```

4 Dynamic SQL

4.1 使用动态 SQL

静态 SQL 语句是在编译时刻其所有信息都已知的语句。例如，下列 SELECT 语句是静态 SQL 语句，因为在编译时刻对于它的执行所需的所有信息都存在。

```
EXEC SQL select company into :cmp_name from customer where customer_num
        = 101;
```

然而，在某些应用程序中，编程人员不知道该程序需要执行的 SQL 语句的这些内容，

或甚至可能不知道其类型。例如，程序可能提示用户输入选择语句，因此，当该程序运行时，编程人员不知道要访问哪些列。这样的应用程序需要动态 SQL。动态 SQL 允许 GBase 8s ESQL/C 程序在运行时刻构建 SQL 语句，因此，可由用户输入来确定该语句的内容。

这些主题描述下列动态 SQL 信息：

如何执行动态 SQL 语句，要使用的 SQL 语句，以及您可动态地执行的语句的类型

当您知道关于该语句在编译时刻的大多数信息时，如何执行 SQL 语句

4.1.1 执行动态 SQL

要执行 SQL 语句，数据库服务器必须有下列关于该语句的信息：

语句的类型，诸如 SELECT、DELETE、EXECUTE PROCEDURE 或 GRANT

任何数据库对象的名称，诸如表、列和索引的名称

任何 WHERE 子句条件，诸如列名称和相匹配的标准

将任何返回的值放在哪里，诸如来自 SELECT 语句的选择列表的列值

需要发送至数据库服务器的值，诸如 INSERT 语句的新行的列值

如果 SQL 语句中的信息随着应用程序中的某些条件而不同，则您的 GBase 8s ESQL/C 程序可使用动态 SQL 来在运行时刻构建该 SQL 语句。对于动态地执行的 SQL 语句的基本处理，由下列步骤组成：

在字符串变量中组装 SQL 语句的文本。

使用 PREPARE 语句来让数据库服务器检测该语句文本，并为执行准备它。

以 EXECUTE 或 OPEN 语句执行准备好的语句。

释放用于执行准备好的语句的动态资源。

组装和准备 SQL 语句

动态 SQL 允许您在字符串中组装 SQL 语句，作为用户与您的程序的交互。动态 SQL 语句像嵌入至程序内的任何其他 SQL 语句一样，只不过，该语句字符串不可包含任何主变量的名称。PREPARE 语句将 SQL 语句字符串的内容发送至数据库服务器，数据库服务器解析它并创建语句标识符结构（语句标识符）。

组装语句

请将 SQL 语句的文本指定为单个主变量，其出现在 PREPARE 语句中。动态地执行 SQL 语句的关键在于，将该语句的文本组装至字符串内。您可以下列两种方式组装此语句字符串：

作为固定的字符串，如果您在编译时刻知道所有信息的话

作为字符串操作的序列，如果您在编译时刻没有所有信息的话

如果您知道整个的语句结构，则可在 PREPARE 语句的 FROM 关键字之后罗列它。括起该语句的引号或双引号是有效的，虽然 ANSI SQL 标准指定引号。例如：

```
EXEC SQL prepare slct_id from
        'select company from customer where customer_num = 101';
```

提示： 虽然 GBase 8s ESQL/C 不允许在引号括起的字符串内有换行符，但您可在 PREPARE 语句的用引号括起的字符串中包括换行符。如果您指定其为应该的话，随同 PREPARE 语句，将引号括起的字符串传至数据库服务器，则数据库服务器允许引号括起的字符串中的换行符。因此，您可允许用户从命令行输入前面的 SQL 语句，如下：

```
select lname from customer
        where customer_num = 101
```

或者，您可将该语句复制至 **char** 变量内，如下列代码段所示。

```
stcopy("select company from customer where customer_num = 101", stmt_txt);
        EXEC SQL prepare slct_id from :stmt_txt;
```

这两种方法都有与静态 SQL 语句相同的限制。它们假设您在编译时刻知道整个的语句结构。与静态语句相比，这些动态形式的缺点在于，直到（通过 PREPARE 语句）运行时刻，才会发现在该语句中遇到的任何语法错误。如果您静态地执行该语句，则 GBase 8s ESQL/C 预处理器可在编译时刻发现句法错误（直到运行时刻，才可能诊断出语义错误）。当您动态地执行要多次执行的 SQL 语句时，可提升性能。仅解析该语句一次。

在前面的代码段中，**stmt_txt** 变量是主变量，因为它用在嵌入的 SQL 语句（PREPARE 语句）中。由于主变量不可出现在语句字符串中，因此，还移除了 SELECT 语句的 INTO 子句。反而，请您在 EXECUTE 或 FETCH 语句的 INTO 子句中指定主变量。像 DESCRIBE、EXECUTE 和 FREE 一样的其他 SQL 语句可访问准备好的语句，当它们指定 **slct_id** 语句标识符时。

重要： 在缺省情况下，语句标识符的作用域是全局的。如果您创建多文件应用程序，且您想要将语句标识符的作用域限制为单个文件，则以 **-local** 预处理器选项来预处理该文件。

如果您在编译时刻不知道关于该语句的所有信息，则您可使用下列特性来组装该语句字符串：

char 主变量可在 SQL 语句（列名称或表名称）中，或在像 WHERE 子句一样的语句的部分中保存标识符。它们还可包含语句的关键字。

如果您知道语句指定的是什么列值，则可声明主变量来提供在 WHERE 子句中需要的列值，或来保存由数据库服务器返回的列值。

在 WHERE 子句中的以问号（?）表示的输入参数占位符指示要提供的列值，通常在执行时刻的主变量中。以这种方式使用的主变量称为输入参数。

您可使用 GBase 8s ESQL/C 字符串库函数，比如 stcopy() 和 stcat()。

下列代码段展示更改了的前面代码段的 SELECT 语句，因此，它使用主变量来动态地确定客户编号。

```
stcopy("select company from customer where customer_num = ", stmt_txt);
        stcat(cust_num, stmt_txt);
        EXEC SQL prepare slct_id from :stmt_txt;
```

下列代码段展示您可如何使用输入参数来编写与此相同的 SELECT 语句的程序，以便用户可输入客户编号。

```
EXEC SQL prepare slct_id from
        'select company from customer where customer_num = ?';
```

您可动态地准备几乎任何 SQL 语句。您不可动态地准备的仅有的语句，是那些直接地涉及动态 SQL 和游标管理的语句（比如 FETCH 和 OPEN），以及 SQL 连接语句。

提示： 您可使用“延迟了的 PREPARE”特性来延迟执行准备好的 SELECT、INSERT 或 EXECUTE FUNCTION 语句，直到 OPEN 语句为止。

准备有集合变量的语句

请您使用带有 INSERT 或 SELECT 语句的 Collection Derived Table 子句来访问 GBase 8s ESQL/C**集合**变量。

当您准备操纵 GBase 8s ESQL/C**集合**变量的语句时，下列限制适用：

您必须执行该语句文本作为 PREPARE 语句中的以引号括起的字符串。

对于**集合**变量， GBase 8s ESQL/C 不支持存储在程序变量中的语句文本。

语句文本的以引号括起的字符串不可包含任何集合主变量。

要操纵**集合**变量，您必须使用问号（?）符来指示输入参数，然后，当您执行该语句时，提供**集合**变量。

如果语句包含集合变量，则您不可执行多语句准备。

例如，下列 GBase 8s ESQL/C 代码段在 **a_set** 客户机**集合**变量上准备 INSERT：

```
EXEC SQL BEGIN DECLARE SECTION;
        client collection set(integer not null) a_set;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL prepare coll_stmt from
        'insert into table values (1, 2, 3)';
        EXEC SQL execute coll_stmt using :a_set;
```

重要： 您必须声明一个 GBase 8s ESQL/C 集合变量作为客户机**集合**变量（存储在客户机计算机上的集合变量）。

检查准备好的语句

当 PREPARE 将语句字符串发送至数据库服务器时，数据库服务器解析它来分析它的错误。数据库服务器在 **sqlca** 结构中指示该解析的成功，如下：

如果语法是正确的，则数据库服务器设置下列 sqlca 字段：

sqlca.sqlcode 字段（SQLCODE）包含零。

sqlca.sqlerrd[0] 字段包含对受影响的行数的估算，如果被解析的语句为 SELECT、UPDATE、INSERT 或 DELETE 的话。

sqlca.sqlerrd[3] 字段包含对执行的成本的估算，如果被解析的语句为 SELECT、UPDATE、INSERT 或 DELETE 的话。此执行成本为加权的磁盘访问的合计，以及处理的总行数。

如果语句字符串包含语法错误，或如果在 PREPARE 期间遇到某其他错误，则数据库服务器设置下列 sqlca 字段：

将 sqlca.sqlcode 字段（SQLCODE）设置为负的数值（<0）。数据库服务器还将 SQLSTATE 变量设置为错误代码。

sqlca.sqlerrd[4] 字段包含检测到错误的语句文本内的偏移量。

执行 SQL 语句

在准备 SQL 语句之后，数据库服务器可执行它。执行准备好的语句的方式依赖于：

该 SQL 语句返回多少行（值的组）：

返回一行数据的语句包括单个的 SELECT 和 EXECUTE FUNCTION 语句。

可返回多行数据的语句需要游标来执行；它们包括非单个的 SELECT 和 EXECUTE FUNCTION 语句。

不返回数据行的所有其他 SQL 语句，包括 EXECUTE PROCEDURE。

该语句是否有输入参数

如果有，必须以 USING 子句执行该语句：

对于 SELECT 和 INSERT 语句，请使用 OPEN...USING 语句。

对于非 SELECT 语句，请使用 EXECUTE...USING 语句。

在编译时刻您是否知道语句列的数据类型：

当您在编译时刻知道该列的数目和数据类型时，您可使用主变量来保存列值。

当您在编译时刻不知道列的数目和数据类型时，您必须使用 DESCRIBE 语句来定义该列和动态管理结构来保存列值。

下表总结如何执行不同类型的准备好的 SQL 语句。

表 1. 执行不返回行的准备好的 SQL 语句（与游标相关联的 INSERT 除外）

SQL 语句的类型	输入参数	要执行的语句	请参阅
不带有输入参数	无	EXECUTE	执行非 SELECT 语句
当输入参数的数目和数据类型已知时	有	EXECUTE...USING	EXECUTE USING 语句
当输入参数的数目和数据类型未知时	有	EXECUTE...USING SQL DESCRIPTOR EXECUTE...USING DESCRIPTOR	处理参数化的 UPDATE 或 DELETE 语句 处理参数化的 UPDATE 或 DELETE 语句

表 2. 执行与游标相关联的 INSERT 语句

SQL 语句的类型	输入参数	要执行的语句	请参阅
不带有输入参数	无	OPEN	声明选择游标
当输出参数（插入列）的数目和数据类型已知时	有	OPEN...USING	OPEN USING 语句，处理未知的列列表
当输入参数的数目和数据类型未知时	有	OPEN...USING SQL DESCRIPTOR OPEN...USING DESCRIPTOR	处理未知的列列表 处理未知的列列表

表 3. 执行可返回多行的准备好的 SQL 语句：非单个的 SELECT、SPL 函数

SQL 语句的类型	输入参数	要执行的语句	请参阅
不带有输入参数	无	OPEN	声明选择游标
当选择列表列的数目和数据类型未知时	无	OPEN	执行返回多行的 SELECT， 执行返回多行的 SELECT
当返回值的数目和数据类型未知时	无	OPEN	执行游标函数， 执行游标函数
当输入参数的数目和数据类型已知时	有	OPEN...USING	OPEN USING 语句
当输入参数的数目和数据类型未知时	有	OPEN...USING SQL DESCRIPTOR OPEN...USING DESCRIPTOR	执行返回多行的参数化的 SELECT 执行返回多行的参数化的 SELECT

表 4. 执行仅返回一行的准备好的 SQL 语句：单个的 SELECT、任何外部函数，或仅返回一组值的 SPL 函数

SQL 语句的类型	输入参数	要执行的语句	请参阅
不带有输入参数	无	EXECUTE...INTO	PREPARE 和 EXECUTE INTO 语句
当返回的值的数目和数据类型未知时	无	EXECUTE...INTO DESCRIPTOR EXECUTE...INTO SQL	处理未知的选择列表 执行非游标函数

SQL 语句的类型	输入参数	要执行的语句	请参阅
		DESCRIPTOR	处理未知的选择列表 执行非游标函数
当输入参数的数目和数据类型已知时	有	EXECUTE...INTO ...USING	EXECUTE USING 语句
当输入参数的数目和数据类型未知时	有	EXECUTE...INTO ...USING SQL DESCRIPTOR EXECUTE...INTO ...USING DESCRIPTOR	执行参数化的单个 SELECT 语句 执行参数化的单个 SELECT 语句

释放资源

有时，您可忽略分配给准备好的语句和游标的资源的成本。然而，应用程序可创建的准备好的对象的数目是有限的。请释放 GBase 8s ESQL/C 用于执行准备好的语句的资源，如下：

如果该语句与游标相关联，则在访存（或插入）所有行之后，请使用 CLOSE 来关闭游标。

请使用 FREE 语句来释放为准备好的语句和任何相关联的游标分配的资源。在您已释放了准备好的语句之后，您可不再在您的程序中使用它，直到您重新准备或重新声明它为止。然而，一旦您声明游标，您即可释放相关联的语句标识符，但不影响该游标。

您可使用 AUTOFREE 特性来让数据库服务器为游标及其准备好的语句自动地释放资源。

如果您的程序在运行时刻使用动态管理结构来描述 SQL 语句，则一旦不再需要该结构，还请归还此结构的资源。

4.1.2 数据库游标

数据库游标是与一组行相关联的标识符。在某种意义上，它是指向缓冲区中当前行的指针。

在下列情况下，您必须使用游标：

从数据库服务器返回多行数据的语句：

SELECT 语句需要选择游标。

EXECUTE FUNCTION 语句需要函数游标。

将多行数据发送到数据库服务器的 INSERT 语句需要插入游标。

接收多行

返回一行数据的语句包括单个的 SELECT 和其用户定义的函数仅返回一行数据的 EXECUTE FUNCTION 语句。返回多行数据的语句包括：

非单个的 SELECT。

当 SELECT 语句返回多行时，请以 DECLARE 语句定义选择游标。

其用户定义的函数返回多行的 EXECUTE FUNCTION 语句。

当 EXECUTE FUNCTION 语句执行返回多行的用户定义的函数时，请以 DECLARE 语句定义函数游标。

对于选择或函数游标，您可使用顺序的、滚动、保存或更新游标。下表总结管理选择或函数游标的 SQL 语句。

表 5. 管理选择或函数游标的 SQL 语句

任务	选择游标	函数游标
声明游标标识符	与 SELECT 语句相关联的 DECLARE	与 EXECUTE FUNCTION 语句相关联的 DECLARE
执行该语句	OPEN	OPEN
从访存缓冲区访问单个行至程序内	FETCH	FETCH
关闭游标	CLOSE	CLOSE
释放游标资源	FREE	FREE

选择游标

选择游标使得您能够扫描 SELECT 语句返回的多行数据。DECLARE 语句将 SELECT 语句与选择游标相关联。

在 DECLARE 语句中，SELECT 可为下列格式之一：

DECLARE 语句中的文字 SELECT 语句

下列 DECLARE 语句将文字 SELECT 语句与 **slct1_curs** 游标相关联：

```
EXEC SQL declare slct1_curs cursor for select * from customer;
```

DECLARE 语句中准备好的 SELECT 语句

下列 DECLARE 语句将准备好的 SELECT 语句与 **slct2_curs** 游标相关联：

```
EXEC SQL prepare slct_stmt cursor from
        'select * from customer';
        EXEC SQL declare slct2_curs for slct_stmt;
```

如果 SELECT 返回仅一行，则称之为单个的 SELECT，且它不需要选择游标来执行。

函数游标

函数游标使得您能够扫描用户定义的函数返回的多行数据。

下列用户定义的函数可返回多行：

在其 RETURN 语句中包含 WITH RESUME 关键字的 SPL 函数

为迭代函数的外部函数

请您以 EXECUTE FUNCTION 语句执行用户定义的函数。 DECLARE 语句将 EXECUTE FUNCTION 与函数游标相关联。 在 DECLARE 语句中，EXECUTE FUNCTION 语句可为下列格式之一：

DECLARE 语句中的文字 EXECUTE FUNCTION 语句

下列 DECLARE 语句将文字 EXECUTE FUNCTION 语句与 **func1_curs** 游标相关联：

```
EXEC SQL declare func1_curs cursor for execute function
        func1();
```

DECLARE 语句中的准备好的 EXECUTE FUNCTION 语句

下列 DECLARE 语句将准备好的 EXECUTE FUNCTION 语句与 **func2_curs** 游标相关联：

```
EXEC SQL prepare func_stmt from
        'execute function func1()';
        EXEC SQL declare func2_curs cursor for func_stmt;
```

如果外部函数或 SPL 函数返回仅一行，则它不需要函数游标来执行。

发送多行

当您执行 INSERT 语句时，该语句发送一行数据至数据库服务器。当 INSERT 语句发送多行时，请以 DECLARE 语句定义一插入游标。插入游标使得您能够为一次插入缓冲多行数据。DECLARE 语句将 INSERT 语句与插入游标相关联。在 DECLARE 语句中，INSERT 语句可为下列格式之一：

DECLARE 语句中的文字 INSERT 语句

下列 DECLARE 语句将文字 INSERT 与 **ins1_curs** 游标相关联：

```
EXEC SQL declare ins1_curs cursor for
        insert into customer values;
```

DECLARE 语句中的准备好的 INSERT 语句

下列 DECLARE 语句将准备好的 INSERT 语句与 **ins2_curs** 游标相关联：

```
EXEC SQL prepare ins_stmt from
        'insert into customer values';
        EXEC SQL declare ins2_curs cursor for ins_stmt;
```

如果您使用插入游标，要比您一次插入一行效率高得多，因为应用程序处理不需要如常地将新行发送至数据库。对于插入游标，您可使用顺序游标或保存游标。下表总结管理插入游标的 SQL 语句。

表 6. 管理插入游标的 SQL 语句

任务	插入游标
声明游标 ID	与 INSERT 语句相关联的 DECLARE
执行该语句	OPEN
将来自程序的单个行发送至插入缓冲区内	PUT

任务	插入游标
清理插入缓冲区并将内容发送至数据库服务器	FLUSH
关闭该游标	CLOSE
释放游标资源	FREE

命名游标

在 GBase 8s ESQL/C 程序中，您可以任何下列项指定游标名称：

文字名称必须服从标识符名称的规则。

定界的标识符是包含不符合标识符命名规则的字符的标识符名称。

动态游标是包含游标的名称的字符主变量。此类游标规范意味着由主变量的值来动态地指定游标名称。您可在允许游标名称的任何 SQL 语句中使用动态游标，除了 DELETE 或 UPDATE 语句的 WHERE CURRENT OF 子句之外。

对于创建通用的函数来执行游标管理任务，动态游标是有用的。您可将该游标的名称作为参数传至函数。如果要在该函数内的 GBase 8s ESQL/C 语句中使用该游标名称，请确保您以 PARAMETER 关键字声明该参数作为主变量。下列代码段展示名为 remove_curs() 的通用游标释放函数。

```
void remove_curs(cursname)
                EXEC SQL BEGIN DECLARE SECTION;
                PARAMETER char *cursname;
                EXEC SQL END DECLARE SECTION;
                {
                EXEC SQL close :cursname;
                EXEC SQL free :cursname;
                }
```

优化游标执行

GBase 8s ESQL/C 支持下列特性，当 GBase 8s ESQL/C 应用程序从数据库服务器访存行时，允许您最小化网络流量：

更改访存和插入缓冲区的大小

自动地释放游标

延迟 PREPARE 语句，直到 OPEN 语句为止

确定游标缓冲区的大小

游标缓冲区是 GBase 8s ESQL/C 应用程序用于保存游标中的数据（除了简单大对象数据之外）的缓冲区。

GBase 8s ESQL/C 有对游标缓冲区的下列使用：

访存缓冲区保存来自选择游标或函数游标的数据

当数据库服务器从查询的活动集返回行时，GBase 8s ESQL/C 在访存缓冲区中存储这些行。

插入缓冲区为插入游标保存数据。

GBase 8s ESQL/C 在插入缓冲区中存储要插入的行，然后将此缓冲区作为一个整体发送至数据库服务器来插入。

使用访存缓冲区，客户机应用程序执行下列任务：

将缓冲区的大小发送至数据库服务器，并当它执行第一个 FETCH 语句时，请求行。

数据库服务器将可适合该访存缓冲区那么多的行发送至应用程序。

从数据库服务器检索行，并将它们置于访存缓冲区中。

从访存缓冲区中取出第一行，并将数据置于该用户已提供的主变量中。

对于后续的 FETCH 语句，该应用程序检查在访存缓冲区中是否存在更多的行。如果存在，则它从访存缓冲区取出下一行。如果访存缓冲区中没有更多的行，则应用程序从数据库服务器请求更多行，发送访存缓冲区大小。

客户机应用程序使用插入缓冲区来执行下列任务：

将来自第一个 PUT 语句的数据置于插入缓冲区内。

为后续的 PUT 语句检查在插入缓冲区中是否存在更多空间。

如何可装下更多行，则应用程序将下一行置于插入缓冲区内。如果插入缓冲区装不下更多行，则应用程序将插入缓冲区的内容发送至数据库服务器。

应用程序继续此过程，直到没有更多行置于插入缓冲区内为止。它将插入缓冲区的内容发送至数据库服务器，当

插入缓冲区已满

它对插入游标执行 FLUSH 语句

它对插入游标执行 CLOSE 语句

缺省的缓冲区大小

客户机应用程序将与该游标相关联的准备好的语句发送至数据库服务器，并请求关于该语句的 DESCRIBE 信息。如果该游标与相关联的准备好的语句，则当 PREPARE 语句执行时，GBase 8s ESQL/C 发出此请求。如果该游标没有相关联的语句，则当 DECLARE 语句执行时，GBase 8s ESQL/C 发出该请求。

当它接收此请求时，数据库服务器将关于该 projection 列表中每一列的 DESCRIBE 信息发送至应用程序。以此信息，GBase 8s ESQL/C 可确定一行数据的大小。在缺省情况下，GBase 8s ESQL/C 确定此游标缓冲区的大小来保存一行数据。它使用下列算法来确定游标缓冲区的缺省的大小：

如果 4096 字节缓冲区装下一行，则缺省的缓冲区大小为 4096 字节（4 KB）。

如果一行的大小超过 4096 字节，则缺省的缓冲区大小为那一行的大小。

一旦它有缓冲区大小，GBase 8s ESQL/C 就分配该游标缓冲区。

自动地释放游标

当 GBase 8s ESQL/C 应用程序使用游标时，它通常将 FREE 语句发送至数据库服务器来归还指定给选择游标的内存，一旦它不再需要那个游标的话。此语句的执行涉及在应用程序与数据库服务器之间消息请求的往返过程。“自动的 FREE”特性（AUTOFREE）将往返过程的数目减少一次。

当启用 AUTOFREE 特性时， GBase 8s ESQL/C 节省消息请求的一次往返过程， 因为它不需要执行该 FREE 语句。 当数据库服务器关闭选择游标时，它自动地释放它已为它分配的内存。 假设您为下列选择游标启用 AUTOFREE 特性：

```
/* 与 SELECT 语句相关联的选择游标 */
        EXEC SQL declare sel_curs cursor for
        select * from customer;
```

当数据库服务器关闭 **sel_curs** 游标时，它相当于自动地执行下列 FREE 语句：

```
FREE sel_curs
```

如果该游标有相关联的准备好的语句，则数据库服务器还释放分配给该准备好的语句的内存。 假设您为下列选择游标启用 AUTOFREE 特性：

```
/* 与准备好的语句相关联的选择游标 */
        EXEC SQL prepare sel_stmt 'select * from customer';
        EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

当数据库服务器关闭 **sel_curs2** 游标时，它相当于自动地执行下列 FREE 语句：

```
FREE sel_curs2;
        FREE sel_stmt
```

在您打开或重新打开游标之前，您必须启用 AUTOFREE 特性。

启用 AUTOFREE 特性

您可以下列方式之一来为 GBase 8s ESQL/C 应用程序启用 AUTOFREE 特性：

将 IFX_AUTOFREE 环境变量设置为 1。

当您使用 IFX_AUTOFREE 环境变量来启用 AUTOFREE 特性时，当关闭该程序的任何线程中的游标时，您自动地释放游标内存。

执行该 SQL 语句，SET AUTOFREE。

以 SET AUTOFREE 语句，您可为特定的游标启用 AUTOFREE 特性。您还可在特定的连接或线程中启用或禁用该特性。

重要： 当您在传统的 GBase 8s ESQL/C 应用程序中启用 AUTOFREE 特性是，请小心。如果传统应用程序使用同一游标两次，则当它试图第二次打开该又表示，它生成错误。当启用 AUTOFREE 特性时，数据库服务器自动地释放该游标，当它关闭它时。因此，当传统应用程序尝试第二次打开它时，该游标不存在，即使该应用程序未显式地执行 FREE 语句。

SET AUTOFREE 语句

您可使用 SQL 语句 SET AUTOFREE 来启用或禁用 AUTOFREE 特性。

SET AUTOFREE 语句允许您在 GBase 8s ESQL/C 程序中采取下列行动：

为所有游标启用 AUTOFREE 特性：

```
EXEC SQL set autofree;
        EXEC SQL set autofree enabled;
```

这些语句是等同的，因为 SET AUTOFREE 语句的缺省行为是启用所有游标。

为所有游标禁用 AUTOFREE 特性：

```
EXEC SQL set autofree disabled;
```

为指定的游标标识符或游标变量启用 AUTOFREE 特性：

```
EXEC SQL set autofree for cursor_id;
            EXEC SQL set autofree for :cursor_var;
```

SET AUTOFREE 语句覆盖 IFX_AUTOFREE 环境变量的任何值。

下列代码段使用 SET AUTOFREE 语句的 FOR 子句来仅为 **curs1** 游标启用 AUTOFREE 特性。在数据库服务器为 **curs1** 执行 CLOSE 语句之后，它自动地释放该游标和准备好的语句。不自动地释放 **curs2** 游标及其准备好的语句。

```
EXEC SQL BEGIN DECLARE SECTION;
        int a_value;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL create database tst_autofree;
        EXEC SQL connect to 'tst_autofree';
        EXEC SQL create table tab1 (a_col int);
        EXEC SQL insert into tab1 values (1);

        /* Declare the curs1 cursor for the slct1 prepared
        * statement */
        EXEC SQL prepare slct1 from 'select a_col from tab1';
        EXEC SQL declare curs1 cursor for slct1;

        /* Enable AUTOFREE for cursor curs1 */
```

```
EXEC SQL set autofree for curs1;

/* Open the curs1 cursor and fetch the contents */
EXEC SQL open curs1;
while (SQLCODE == 0)
{
EXEC SQL fetch curs1 into :a_value;
printf("Value is: %d\n", a_value);
}

/* Once the CLOSE completes, the curs1 cursor is freed and
* cannot be used again. */
EXEC SQL close curs1;

/* Declare the curs2 cursor for the slct2 prepared
* statement */
EXEC SQL prepare slct2 from 'select a_col from tab1';
EXEC SQL declare curs2 cursor for slct2;

/* Open the curs2 cursor and fetch the contents */
EXEC SQL open curs2;
while (SQLCODE == 0)
{
EXEC SQL fetch curs2 into :a_value;
printf("Value is: %d\n", a_value);
}

/* Once this CLOSE completes, the curs2 cursor is still
* available for use. It has not been automatically freed. */
EXEC SQL close curs2;

/* You must explicitly free the curs2 cursor and slct2
* prepared statement.    */
```

```
EXEC SQL free curs2;
EXEC SQL free slct2;
```

当您使用 AUTOFREE 特性时，请确保您未导致准备好的语句成为脱离的。如果您在同一准备好的语句上声明多个游标，则可发生此情况。准备好的语句被关联到或附加到在 DECLARE 语句中指定它的第一个游标。如果为此游标启用 AUTOFREE 特性，则当它对该游标执行 CLOSE 语句时，数据库服务器释放该游标及其相关联的准备好的语句。

当下列事件之一发生时，准备好的语句成为脱离的：

如果准备好的语句未与任何声明了的游标相关联

如果释放了带有准备好的语句的游标，但未释放该准备好的语句。

如果未为游标启用 AUTOFREE 特性，且您仅释放游标，未释放准备好的语句，则可发生第二种情况。准备好的语句成为脱离的。要重新附加该准备好的语句，请为准备好的语句声明新的游标。一旦释放了准备好的语句，就不可使用它来声明任何新的游标。

下列代码段对 **slct1** 准备好的语句声明下列游标：

curs1 游标，slct1 准备好的语句首先与其相关联

curs2 游标，其执行 slct1，但 slct1 未与其相关联

curs3 游标，slct1 与其相关联

下列代码段展示脱离的准备好的语句可如何发生：

```
/*****************************************************************
        * Declare curs1 and curs2. The slct1 prepared statement is      *
        * associated curs1 because curs1 is declared first.                */
        EXEC SQL prepare slct1 'select a_col from tab1';
        EXEC SQL declare curs1 cursor for slct1;
        EXEC SQL declare curs2 cursor for slct1;

/*****************************************************************
        * Enable the AUTOFREE feature for curs2                            */
        EXEC SQL set autofree for curs2;
```

```
/*****************************************************************
                * Open the curs1 cursor and fetch the contents                  */
                EXEC SQL open curs1;
                {
                EXEC SQL fetch curs1 into :a_value;
                printf("Value is: %d\n", a_value);
                }

                EXEC SQL close curs1;

                /* Because AUTOFREE is enabled only for the curs2 cursor, this     *
                * CLOSE statement frees neither the curs1 cursor nor the slct1    *
                * prepared statement. The curs1 cursor is still defined so the    *
                * slct1 prepared statement does not become detached.              *
*****************************************************************************/


/*****************************************************************
                * Open the curs2 cursor and fetch the contents                  */
                EXEC SQL open curs2;
                while (SQLCODE == 0)
                {
                EXEC SQL fetch curs2 into :a_value;
                printf("Value is: %d\n", a_value);
                }

                EXEC SQL close curs2;

                /* This CLOSE statement frees the curs2 cursor but does not free *
                * slct1 prepared statement because the prepared statement is not*
                * associated with curs2.                                        *
```

```
**********************************************************************/

/**********************************************************************
                * Reopen the curs1 cursor. This open is possible because the       *
                * AUTOFREE feature has not been enabled on curs1. Therefore, the*
                * database server did not automatically free curs1 when it closed it.*/
                EXEC SQL open curs1;
                while (SQLCODE == 0)
                {
                EXEC SQL fetch curs1 into :a_value;
                printf("Value is: %d\n", a_value);
                }

                EXEC SQL close curs1;
                EXEC SQL free curs1;

                /* Explicitly freeing the curs1 cursor, with which the slct1        *
                * statement is associated, causes slct1 to become detached. It    *
                * is no longer associated with a cursor.                                  *

**********************************************************************/

                /**********************************************************************
                * This DECLARE statement causes the slct1 prepared statement    *
                * to become reassociated with a cursor. Therefore, the slct1    *
                * statement is no longer detached.                                        */
                EXEC SQL declare curs3 cursor for slct1;
                EXEC SQL open curs3;

                /* Enable the AUTOFREE feature for curs                                 */
                EXEC SQL set autofree for curs3;
```

```
                /* Open the curs3 cursor and fetch the content                    */
                EXEC SQL open curs3;
                while (SQLCODE == 0)
                {
                EXEC SQL fetch curs3 into :a_value;
                printf("Value is: %d\n", a_value);
                }

                EXEC SQL close curs3;

                /* Because AUTOFREE is enabled for the curs3 cursor, this CLOSE*
                * statement frees the curs3 cursor and the slct1 PREPARE stmt.*
**************************************************************************/

                /*****************************************************************
                * This DECLARE statement would generate a run time error        *
                * because the slct1 prepared statement has been freed.          */
                EXEC SQL declare x4 cursor for slct1;

/*************************************************************************/
```

延迟 PREPARE 语句的执行

当 GBase 8s ESQL/C 应用程序使用 PREPARE/DECLARE/OPEN 语句块来执行游标时，每一语句都涉及在应用程序与数据库服务器之间的消息请求的往返过程。“延迟的 PREPARE”特性减少往返过程一次。当启用“延迟的 PREPARE”特性时，GBase 8s ESQL/C 节省消息请求的一次往返过程，因为它不需要发送分开的命令来执行 PREPARE 语句。反而，当数据库服务器接收 OPEN 语句时，它自动地执行 PREPARE 语句。

假设您为下列选择游标启用“延迟的 PREPARE”特性：

```
/* 与 SELECT 语句相关联的选择游标 */
        EXEC SQL prepare slct_stmt FOR
        'select * from customer';
        EXEC SQL declare sel_curs cursor for slct_stmt;
```

```
EXEC SQL open sel_curs;
```

当 GBase 8s ESQL/C 应用程序在 DECLARE 语句之前遇到 PREPARE 时，它不将 PREPARE 语句发送至数据库服务器。反而，当它执行 OPEN 语句时，它将 PREPARE 和 OPEN 发送至数据库服务器。

您可在 GBase 8s ESQL/C 应用程序中使用“延迟的 PREPARE”特性，该应用程序包含使用 PREPARE、DECLARE 和 OPEN 的语句块的动态 SQL 语句来执行下列语句：

SELECT 语句（选择游标）

EXECUTE FUNCTION 语句（函数游标）

INSERT 语句（插入游标）

例如，“延迟的 PREPARE”特性为下列选择游标减少网络往返过程：

```
/* 对于“延迟的 PREPARE”优化有效的选择游标 */
        EXEC SQL prepare sel_stmt 'select * from customer';
        EXEC SQL declare sel_curs cursor for sel_stmt;
        EXEC SQL open sel_curs;
```

对“延迟的 PREPARE”的限制

当您启用延迟的 PREPARE 特性时，当客户机应用程序遇到 PREPARE 语句时，它不将它们发送至数据库服务器。当数据库服务器执行 OPEN 语句时，它接收该准备好的语句的描述。

如果您在游标的第一个 OPEN 之前对准备好的语句执行 DESCRIBE 语句，则数据库服务器生成错误。发生该错误，是由于数据库服务器尚未执行 DESCRIBE 语句指定的 PREPARE 语句。当启用延迟的 PREPARE 特性时，您必须在游标的第一个 OPEN 之后执行 DESCRIBE 语句。

重要： 延迟的 PREPARE 特性消除 PREPARE 语句作为分开的步骤的执行。因此，应用程序不接收在准备好的语句中可能存在的任何错误条件，直到初始的 OPEN 之后为止。

启用延迟的 PREPARE 特性

您可以下列方式之一，为 GBase 8s ESQL/C 应用程序启用“延迟的 PREPARE”特性：

将 IFX_DEFERRED_PREPARE 环境变量设置为 1。

当您使用 IFX_DEFERRED_PREPARE 环境变量量启用“延迟的 PREPARE”特性时，您自动地延迟该 PREPARE 语句的执行，直到就在为该应用程序的任何线程中的每个 PREPARE 语句执行 OPEN 语句之前为止。

IFX_DEFERRED_PREPARE 环境变量的缺省值为 0。如果您从 shell 设置此环境变量，请确保在您启动 GBase 8s ESQL/C 应用程序之前设置它。

执行 SQL 语句 SET DEFERRED_PREPARE。

以 SET DEFERRED_PREPARE 语句，您可为特定的 PREPARE 语句启用“延迟的 PREPARE”特性。您还可在特定的连接或线程中启用或禁用该特性。

SET DEFERRED_PREPARE 语句

在 GBase 8s ESQL/C 应用程序中，您可使用 SQL 语句 SET DEFERRED_PREPARE 来启用或禁用“延迟的 PREPARE”特性。

SET DEFERRED_PREPARE 语句允许您在 GBase 8s ESQL/C 程序中采取下列行动：

启用“延迟的 PREPARE”特性：

EXEC SQL set deferred_prepare;

EXEC SQL set deferred_prepare enabled;

禁用“延迟的 PREPARE”特性：

EXEC SQL set deferred_prepare disabled;

该 SET DEFERRED_PREPARE 语句覆盖 IFX_DEFERRED_PREPARE 环境变量的任何值。

下列代码段展示如何为 **ins_curs** 插入游标启用“延迟的 PREPARE”特性：

```
EXEC SQL BEGIN DECLARE SECTION;
        int a;
        EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL create database test;
EXEC SQL create table table_x (col1 integer);

/*************************************
* Enable Deferred-Prepare feature
*************************************/
EXEC SQL set deferred_prepare enabled;

/*************************************
* Prepare an INSERT statement
*************************************/
EXEC SQL prepare ins_stmt from
'insert into table_x values(?)';

/*************************************
* Declare the insert cursor for the
* prepared INSERT.
*************************************/
EXEC SQL declare ins_curs cursor for ins_stmt;
/*************************************************************
* OPEN the insert cursor. Because the Deferred-PREPARE feature
* is enabled, the PREPARE is executed at this time
*************************************************************/
EXEC SQL open ins_curs;
a = 2;
while (a<100)
{
EXEC SQL put ins_curs from :a;
a++;
}
```

要对准备好的语句执行 DESCRIBE 语句，您必须在该游标的初始 OPEN 语句之后执

行该 DESCRIBE。在下列代码段中，第一个 DESCRIBE 语句失败，是因为它在该游标上的第一个 OPEN 语句之前执行。第二个 DESCRIBE 语句成功，是因为它跟在 OPEN 语句之后。

```
EXEC SQL BEGIN DECLARE SECTION;
        int a, a_type;
        EXEC SQL END DECLARE SECTION;
        EXEC SQL allocate descriptor 'desc';
        EXEC SQL create database test;
        EXEC SQL create table table_x (col1 integer);

        /*********************************************
        * Enable Deferred-Prepare feature
        *********************************************************/
        EXEC SQL set deferred_prepare enabled;

        /*********************************************
        * Prepare an INSERT statement
        *********************************************************/
        EXEC SQL prepare ins_stmt from 'insert into table_x values (?)';

/*************************************************************************
        * The DESCRIBE results in an error, because the description of the
        * statement is not determined until after the OPEN. The OPEN is what
        * actually sends the PREPARE statement to the database server and
        * requests a description for it.
*************************************************************************/
        EXEC SQL describe ins_stmt using sql descriptor 'desc'; /* fails */
        if (SQLCODE)
        printf("DESCRIBE : SQLCODE is %d\n", SQLCODE);

/*************************************************************************
```

```
                * Now DECLARE a cursor for the PREPARE statement and OPEN it.

*****************************************************************************/
                EXEC SQL declare ins_cursor cursor for ins_stmt;
                EXEC SQL open ins_cursor;


/*****************************************************************************
                * Now the DESCRIBE returns the information about the columns to the
                * system-descriptor area.

*****************************************************************************/
                EXEC SQL describe ins_stmt using sql descriptor 'desc'; /* succeeds */
                if (SQLCODE)
                printf("DESCRIBE : SQLCODE is %d\n", SQLCODE);
                a = 2;
                a_type = SQLINT;
                while (a<100)
                {
                EXEC SQL set descriptor 'desc' values 1
                type = :a_type, data = :a;
                EXEC SQL put ins_curs using sql descriptor 'desc';
                a++;
                }
```

4.1.3 collect.ec 程序

collect.ec 示例程序说明如何使用集合变量来访问 LIST、SET 和 MULTISET 列。该 SELECT 语句被认为是静态的，因为当编写程序时，确定它访问的列。

```
/*
                        **
                        ** Sample use of collections in ESQL/C.
                        **
                        ** Statically determined LIST, SET, and MULTISET collection
types.
                        */
```

```
#include <stdio.h>

static void print_collection(
const char *tag,
EXEC SQL BEGIN DECLARE SECTION;
parameter client collection c
EXEC SQL END DECLARE SECTION;
)
{
EXEC SQL BEGIN DECLARE SECTION;
int4 value;
EXEC SQL END DECLARE SECTION;
mint item = 0;

EXEC SQL WHENEVER ERROR STOP;
printf("COLLECTION: %s\n", tag);
EXEC SQL DECLARE c_collection CURSOR FOR
SELECT * FROM TABLE(:c);
EXEC SQL OPEN c_collection;
while (sqlca.sqlcode == 0)
{
EXEC SQL FETCH c_collection INTO :value;
if (sqlca.sqlcode != 0)
break;
printf("\tItem %d, value = %d\n", ++item, value);
}
EXEC SQL CLOSE c_collection;
EXEC SQL FREE c_collection;
}

mint main(int argc, char **argv)
{
```

```
EXEC SQL BEGIN DECLARE SECTION;
client collection list          (integer not null) lc1;
client collection set           (integer not null) sc1;
client collection multiset (integer not null) mc1;
char *dbase = "stores7";
mint seq;
char *stmt1 =
"INSERT INTO t_collections VALUES(0, "
"'LIST{-1,0,-2,3,0,0,32767,249}', 'SET{-1,0,-2,3}', "
"'MULTISET{-1,0,0,-2,3,0}') ";
EXEC SQL END DECLARE SECTION;

if (argc > 1)
dbase = argv[1];
EXEC SQL WHENEVER ERROR STOP;
printf("Connect to %s\n", dbase);
EXEC SQL connect to :dbase;

EXEC SQL CREATE TEMP TABLE t_collections
(
seq serial not null,
l1 list       (integer    not null),
s1 set         (integer    not null),
m1 multiset(integer    not null)
);
EXEC SQL EXECUTE IMMEDIATE :stmt1;

EXEC SQL ALLOCATE COLLECTION :lc1;
EXEC SQL ALLOCATE COLLECTION :mc1;
EXEC SQL ALLOCATE COLLECTION :sc1;

EXEC SQL DECLARE c_collect CURSOR FOR
SELECT seq, l1, s1, m1 FROM t_collections;
```

```
EXEC SQL OPEN c_collect;

EXEC SQL FETCH c_collect INTO :seq, :lc1, :sc1, :mc1;
EXEC SQL CLOSE c_collect;
EXEC SQL FREE c_collect;

print_collection("list/integer", lc1);
print_collection("set/integer", sc1);
print_collection("multiset/integer", mc1);

EXEC SQL DEALLOCATE COLLECTION :lc1;
EXEC SQL DEALLOCATE COLLECTION :mc1;
EXEC SQL DEALLOCATE COLLECTION :sc1;

puts("OK");
return 0;
}
```

4.1.4 优化 OPEN、FETCH 和 CLOSE

当 GBase 8s ESQL/C 应用程序使用 DECLARE 和 OPEN 语句来执行游标时，每一语句涉及在应用程序与数据库服务器之间的消息请求的往返过程。“优化 OPEN-FETCH-CLOSE”特性（OPTOFC）减少往返过程两次，如下：

GBase 8s ESQL/C 节省一次往返过程，因为它不作为分开的命令发送 OPEN 语句。

当 GBase 8s ESQL/C 执行 OPEN 语句时，它不打开该游标。反而，它保存在 OPEN 语句的 USING 子句中提供了的任何输入值。当 GBase 8s ESQL/C 执行初始的 FETCH 语句时，它与该 FETCH 语句一道发送此输入值。数据库服务器打开该游标，并返回在此游标中的第一个值。

GBase 8s ESQL/C 节省第二次往返过程，因为它不作为分开的命令发送 CLOSE 语句。

当数据库服务器达到打开游标的最后的值时，在它将最后的值发送至客户机应用程序之后，它自动地关闭该游标。因此，GBase 8s ESQL/C 不需要将 CLOSE 语句发送至数据库服务器。

重要： GBase 8s ESQL/C 不将 CLOSE 语句发送至数据库服务器。然而，如果您包括该 CLOSE 语句，则不生成错误。

对 OPTOFC 的限制

对于启用的 OPTOFC 特性，存在下列限制：

您仅可对准备了其 SELECT 语句的选择游标使用 OPTOFC 特性。例如，OPTOFC 特性为下列选择游标减少网络往返过程：

```
/* Valid select cursor for OPTOFC optimization */
        EXEC SQL prepare sel_stmt 'select * from customer';
        EXEC SQL declare sel_curs cursor for sel_stmt;
```

OPTOFC 特性消除作为分开的步骤的 OPEN 语句的执行。因此，不返回打开该游标可能生成的任何错误条件，直到初始的 FETCH 之后为止。

当关闭静态游标时，不释放游标。

随同启用的 OPTOFC 特性，当关闭静态游标或动态游标时，不释放它们。由于 GBase 8s ESQL/C 实际上不将 CLOSE 语句发送至数据库服务器，因此，不隐式地释放游标。对游标的后续的 OPEN 和 FETCH 实际上打开同一个游标。如果更改了表（如果删除了、修改了或重命名了它），则数据库服务器仅在此时会注意到，在此情况下，它生成错误（-710）。

随同禁用的 OPTOFC 特性，当关闭静态游标时，释放它。当 ESQL/C 达到静态游标的 CLOSE 语句，它实际上发送消息来关闭该游标，并释放与此游标相关联的内存。然而，当关闭动态游标时，不隐式地释放它们。

对于在至数据库服务器的途中延迟的 SQL 语句，GET DIAGNOSTICS 语句不奏效。例如，下列 SQL 语句序列中，GET DIAGNOSTICS 返回 0，指示成功，即使延迟该 OPEN，直到首次访存为止：

```
EXEC SQL declare curs1 …
        EXEC SQL open curs1
        EXEC SQL get diagnostic
        EXEC SQL fetch curs1
```

启用 OPTOFC 特性

OPTOFC 环境变量启用 OPTOFC 特性。

您可将下列值指定给 OPTOFC 环境变量。

1

此值启用 OPTOFC 特性。当您指定此值时，您为该应用程序的每个线程中的每个游标启用 OPTOFC 特性。

0

此值为该应用程序的所有线程禁用 OPTOFC 特性。

OPTOFC 环境变量的缺省值为 0。如果您从 shell 设置此环境变量，则请确保在您启用 ESQL/C 应用程序之前设置它。

在 UNIX(TM) 操作系统上，您可在应用程序中以 putenv() 系统调用来设置 OPTOFC（只要您的系统支持 putenv() 函数）。例如，下列对 putenv() 的调用启用 OPTOFC 特性：

```
putenv("OPTOFC=1");
```

在 Windows(TM) 环境中，您可使用 ifx_putenv() 函数。

以 putenv() 或 ifx_putenv()，您可为每一连接或在每一线程内激活或去激活 OPTOFC 特性。您必须在建立连接之前调用 putenv() 或 ifx_putenv() 函数。

重要： GBase 8s 实用程序不支持 IFX_AUTOFREE、OPTOFC 和 IFX_DEFERRED_PREPARE 环境变量。请仅以 GBase 8s ESQL/C 客户机应用程序来使用这些环境变量。

4.1.5 一起使用 OPTOFC 与 延迟的 PREPARE

要获得在客户机应用程序与数据库服务器之间消息的最优数目，请一起使用“优化 OPEN、FETCH、CLOSE”特性与“延迟的 PREPARE”特性。

然而，当您一起使用这两种优化特性时，请记住下列要求：

如果在语句文本中存在语法错误，直到数据库服务器执行 FETCH，它才返回该错误。

GBase 8s ESQL/C 不将 PREPARE、DECLARE 和 OPEN 语句发送至数据库服务器，直到它执行 FETCH 语句为止。因此，直到数据库服务器执行 FETCH 语句，才能得到任何这些语句生成的任何错误。

您必须使用 GET DESCRIPTOR 语句的特殊情况来获得准备好的语句的 DESCRIBE 信息。

DESCRIBE 语句的典型使用，是在 PREPARE 确定关于该准备好的语句的信息之后执行它。然而，随同同时启用的 OPTOFC 和“延迟的 PREPARE”特性，GBase 8s ESQL/C 不将 DESCRIBE 语句发送至数据库，直到它达到 FETCH 语句为止。要允许您获取关于准备好的语句的信息，GBase 8s ESQL/C 执行类似于 SET DESCRIPTOR 语句的语句来获得

准备好的语句的数据类型、长度和其他系统描述符字段。然后，您可在 FETCH 之后使用 GET DESCRIPTOR 语句来获取此信息。

而且，当数据类型为内建的数据类型时，GBase 8s ESQL/C 仅可在 GET DESCRIPTOR 语句中的主变量上执行数据转换。对于 opaque 数据类型和复合的数据类型（集合和 row 类型），数据库服务器总是以其原本的格式将数据返回至客户机应用程序。然后，您可在 GET DESCRIPTOR 语句之后对此数据执行数据转换。

例如，数据库服务器以其内部的（二进制）格式从 opaque 类型列返回数据。因此，您的 GBase 8s ESQL/C 程序必须将列数据放置到 **var binary**（或 **fixed binary**）主变量内，当它执行 GET DESCRIPTOR 语句时。**var binary** 和 **fixed binary** 数据类型以其内部的格式保存 opaque 类型数据。您不可使用 **lvarchar** 主变量来保存该数据，因为 GBase 8s ESQL/C 不可将 opaque 类型数据从其内部的（它从数据库服务器收到的）格式转换为其外部的（**lvarchar**）格式。

当同时启用“延迟的 PREPARE”与 OPTOFC 特性时，FetArrSize 特性不奏效。当启用这两个特性时，直到 FETCH 完成之后，GBase 8s ESQL/C 才知道行的大小。到此时，要以 FetArrSize 值来调整访存缓冲区为时已晚。

提示：要获得最大的优化，请一起使用 OPTOFC、“延迟的 PREPARE”与 AUTOFREE 特性。

4.1.6 在编译时刻知道的 SQL 语句

要执行的最简单的动态 SQL 的类型，是您知道其下列项：

要执行的 SQL 语句的结构，包括比如语句类型以及语句的语法这样的信息

在 GBase 8s ESQL/C 程序与数据库服务器之间传递的任何数据的数目及数据类型

执行非 SELECT 语句

术语非 SELECT 语句指的是可被准备的任何 SQL 语句，除了 SELECT 和 EXECUTE FUNCTION 之外。此术语包括 EXECUTE PROCEDURE 语句。

重要： INSERT 语句是对非 SELECT 语句的规则的一个例外。如果 INSERT 插入单个行，则请使用 PREPARE 和 EXECUTE 来执行它。然而，如果该 INSERT 与插入游标相关联，则您必须声明该插入游标。

您可以下列方式执行非 SELECT 语句：

如果多次执行该语句，则请使用 PREPARE 和 EXECUTE 语句。

如果仅执行该语句一次，则请使用 EXECUTE IMMEDIATE 语句。此语句对它可执行的语句没有限制。

PREPARE 和 EXECUTE 语句

PREPARE 和 EXECUTE 语句允许您将非 SELECT 语句的执行分为两个步骤：

PREPARE 将语句字符串发送至数据库服务器，其解析该语句，并分配给它语句标识符。

EXECUTE 执行由语句标识符指示的准备好的语句。

对于需要多次执行的语句，这两个步骤的处理是有用的。当您仅解析该语句一次时，请减少在客户机应用程序与数据库服务器之间的流量。

例如，您可编写通用目的的删除程序，对任何表操作。此程序会采取下列步骤：

提示用户表的名称和 WHERE 子句的文本，并将该信息放置在诸如 **tabname** 和 **search_condition** 这样的 C 变量内。**tabname** 和 **search_condition** 变量不需要作为主变量，因为它们在实际的 SQL 语句中出现。

通过连接下列四个组件来创建文本字符串：DELETE FROM、**tabname**、WHERE 和 **search_condition**。在此示例中，主变量中的字符串称为 **stmt_buf**：

```
sprintf(stmt_buf, "DELETE FROM %s WHERE %s",
                    tabname, search_condition);
```

准备整个语句。下列 PREPARE 语句对 stmt_buf 中的字符串操作，并创建称为 d_id 的语句标识符：

```
EXEC SQL prepare d_id from :stmt_buf;
```

执行该语句。下列 EXECUTE 语句执行 DELETE：

```
EXEC SQL execute d_id;
```

如果您不需要再次执行该语句，请释放该语句标识符结构使用的资源。此示例会使用下列 FREE 语句：

```
EXEC SQL free d_id;
```

如果非 SELECT 语句包含输入参数，则您必须使用 EXECUTE 语句的 USING 子句。

通常使用 EXECUTE 语句来执行非 SELECT 语句。对于 SELECT 或 EXECUTE FUNCTION 语句，您可使用带有 INTO 子句的 EXECUTE，只要这些语句仅返回一组值（一行）。然而，请不要使用 EXECUTE 语句，对于：

与插入游标相关联的 INSERT...VALUES 语句。

游标函数（返回多组值的用户定义的函数）的 EXECUTE FUNCTION 语句。

EXECUTE IMMEDIATE 语句

并非准备语句然后执行它，而是您可以 EXECUTE IMMEDIATE 语句在同一步骤中准备并执行该语句。EXECUTE IMMEDIATE 语句还在完成时释放语句标识符资源。

例如，对于在前一部分中使用的 DELETE 语句，您可以下列语句替代 PREPARE-EXECUTE 语句序列：

```
EXEC SQL execute immediate :stmt_buf;
```

如果语句字符串包含输入参数，则您不可使用 EXECUTE IMMEDIATE。SQL 还对您可以 EXECUTE IMMEDIATE 的执行有限制。

执行 SELECT 语句

您可以下列方式执行 SELECT 语句：

如果 SELECT 语句仅返回一行，则请使用 PREPARE 和 EXECUTE INTO。此类 SELECT 常常称为单个的 SELECT。

如果 SELECT 语句返回多行，则您必须使用游标管理语句。

PREPARE 和 EXECUTE INTO 语句

您可以 EXECUTE 语句执行的唯一的准备好的 SELECT 语句是单个 SELECT。您的 GBase 8s ESQL/C 程序必须采取下列行动：

声明主变量来接收数据库服务器返回的值。

对于准备好的 SELECT 语句，这些值为选择列表列。

组装并准备该语句。

准备好的 SELECT 语句可在 WHERE 子句中包含输入参数。

以 EXECUTE...INTO 语句执行准备好的选择，带有 INTO 关键字之后的主变量。

如果 SELECT 语句包含输入参数，则请包括 EXECUTE 的 USING 子句。

提示： 要执行单个的 SELECT，与使用 DECLARE、OPEN 和 FETCH 语句相比，EXECUTE...INTO 语句更高效。

以 EXECUTE 语句的 INTO 子句，您仍可使用下列特性：

您可将指示符变量与接收选择列表列值的主变量相关联。

请使用后跟指示符主变量的名称的 INDICATOR 关键字，如下：

```
EXEC SQL prepare sel1 from
        'select fname, lname from customer where customer_num = 123';
        EXEC SQL execute sel1 into :fname INDICATOR :fname_ind,
        :lname INDICATOR :lname_ind;
```

您可指定输入参数值。

请包括 EXECUTE 的 USING 子句，如下：

```
EXEC SQL prepare sel2 from
        'select fname, lname from customer where customer_num = ?';
        EXEC SQL execute sel2 into :fname, :lname using :cust_num;
```

重要： 当您使用 EXECUTE INTO 语句时，请确保该 SELECT 语句为单个的 SELECT。如果该 SELECT 返回多行，则您收到运行时刻错误。如果您尝试执行（以 DECLARE）声明了的准备好的语句，则也生成错误。

您不需要准备单个的 SELECT。如果您不需要准备好的语句的益处，您可直接将单个的 SELECT 语句嵌入在您的 GBase 8s ESQL/C 程序中，如下列示例所示：

```
EXEC SQL select order_date from orders where order_num = 1004;
```

下图展示如何执行 items_pct() SPL 函数（如 图 1所示）。由于此函数返回单个的**十进制**值，因此，EXECUTE...INTO 语句可执行它。

```
EXEC SQL prepare exfunc_id from
        'execute function items_pct(\"HSK\")';
        EXEC SQL execute exfunc_id into :manuf_dec;
```

您可使用例程参数的主变量，但不可使用例程名称。例如，如果 **manu_code** 变量保存值 "HSK"，则下列 EXECUTE 语句替代准备好的语句中的输入参数，来执行与前面的代码段中 EXECUTE 一样的任务。

```
EXEC SQL prepare exfunc_id from
        'execute function items_pict(?)';
        EXEC SQL execute exfunc_id into :manuf_dec using :manu_code;
```

如果您不知道选择列表列或函数返回值的数目或数据类型，则您必须使用动态管理结构，而不是随同 EXECUTE...INTO 语句的主变量。动态管理结构在运行时刻定义选择列表列。

声明选择游标

要执行返回多行的 SELECT 语句，您必须声明选择游标。选择游标使得 GBase 8s ESQL/C 应用程序能够处理查询返回的多行。

您的 GBase 8s ESQL/C 程序必须采取下列行动来使用选择游标：

声明主变量来接收数据库服务器返回的值。

对于准备好的 SELECT 语句，这些值为选择列表列。对于准备好的 EXECUTE FUNCTION 语句，这些值为用户定义的函数的返回值。

组装并准备该语句。

准备好的 SELECT 语句可在 WHERE 子句中包含输入参数。准备好的 EXECUTE FUNCTION 语句可包含输入参数作为函数参数。

声明选择游标。

DECLARE 语句将准备好的 SELECT 语句与选择游标相关联。

执行该查询。

OPEN 语句将它的 USING 子句指定的任何输入参数发送至数据库服务器，并告诉数据库服务器执行该 SELECT 语句。

从选择游标检索值的行。

FETCH 接收与查询标准相匹配的一行数据。

限制：请不要在与游标相关联的 SELECT 语句与从该游标检索数据的 FETCH 语句中同时使用 INTO 子句。GBase 8s ESQL/C 预处理器或可执行的程序不可生成此情况的错误。然而，同时在两个语句中使用 INTO 子句可生成不可预料的结果。

lvarptr.ec 程序

lvarptr.ec 示例程序使用 **lvarchar** 指针，如下

```
/*
**
** Sample use of LVARCHAR to fetch collections in ESQL/C.
**
** Statically determined collection types.
*/

#include <stdio.h>

static void print_lvarchar_ptr(
const char *tag,
EXEC SQL BEGIN DECLARE SECTION;
parameter lvarchar **lv
EXEC SQL END DECLARE SECTION;
)
{
char *data;

data = ifx_var_getdata(lv);
if (data == 0)
data = "<<NO DATA>>";
printf("%s: %s\n", tag, data);
}

static void process_stmt(char *stmt)
{
EXEC SQL BEGIN DECLARE SECTION;
lvarchar *lv1;
lvarchar *lv2;
lvarchar *lv3;
mint seq;
```

```
char *stmt1 = stmt;
EXEC SQL END DECLARE SECTION;

printf("SQL: %s\n", stmt);

EXEC SQL WHENEVER ERROR STOP;
EXEC SQL PREPARE p_collect FROM :stmt1;
EXEC SQL DECLARE c_collect CURSOR FOR p_collect;
EXEC SQL OPEN c_collect;

ifx_var_flag(&lv1, 1);
ifx_var_flag(&lv2, 1);
ifx_var_flag(&lv3, 1);

while (sqlca.sqlcode == 0)
{
EXEC SQL FETCH c_collect INTO :seq, :lv1, :lv2, :lv3;
if (sqlca.sqlcode == 0)
{
printf("Sequence: %d\n", seq);
print_lvarchar_ptr("LVARCHAR 1", &lv1);
print_lvarchar_ptr("LVARCHAR 2", &lv2);
print_lvarchar_ptr("LVARCHAR 3", &lv3);
ifx_var_dealloc(&lv1);
ifx_var_dealloc(&lv2);
ifx_var_dealloc(&lv3);
}
}

EXEC SQL CLOSE c_collect;
EXEC SQL FREE c_collect;
EXEC SQL FREE p_collect;
}
```

```
mint main(int argc, char **argv)
{
EXEC SQL BEGIN DECLARE SECTION;
char *dbase = "stores7";
char *stmt1 =
"INSERT INTO t_collections VALUES(0, "
"'LIST{-1,0,-2,3,0,0,32767,249}', 'SET{-1,0,-2,3}', "
"'MULTISET{-1,0,0,-2,3,0}') ";
char *data;
EXEC SQL END DECLARE SECTION;

if (argc > 1)
dbase = argv[1];
EXEC SQL WHENEVER ERROR STOP;
printf("Connect to %s\n", dbase);
EXEC SQL CONNECT TO :dbase;

EXEC SQL CREATE TEMP TABLE t_collections
(
seq serial not null,
l1 list       (integer    not null),
s1 set         (integer    not null),
m1 multiset(integer    not null)
);

EXEC SQL EXECUTE IMMEDIATE :stmt1;
EXEC SQL EXECUTE IMMEDIATE :stmt1;
EXEC SQL EXECUTE IMMEDIATE :stmt1;

process_stmt("SELECT seq, l1, s1, m1 FROM t_collections");

puts("OK");
```

```
return 0;
}
```

执行 GBase 8s 中的用户定义的例程

在 GBase 8s 中，用户定义的例程是执行用户定义的任务的语句的集合。过程是可接受参数但不返回任何值的例程。函数是可接受参数并返回值的例程。

下表总结用户定义的例程的 SQL 语句。

表 7. 用户定义的例程的 SQL 语句

任务	过程	函数
创建并注册例程	CREATE PROCEDURE	CREATE FUNCTION
执行例程	EXECUTE PROCEDURE	EXECUTE FUNCTION
删除例程	DROP PROCEDURE	DROP FUNCTION

对于用户定义的例程，GBase 8s 支持几种语言：

以诸如 C 这样的外部语言编写的外部例程。

在外部过程不返回值时，外部函数可返回一个值。

SPL 例程是以“存储过程语言”（SPL）编写的。

在 SPL 过程不返回任何值时，SPL 函数可返回一个或多个值。

提示： 在较早版本的 GBase 8s 产品中，使用术语存储过程同时表示 SPL 过程和 SPL 函数。即，存储过程可包括 RETURN 语句来返回值。为了与较早的产品相兼容，GBase 8s 继续支持以 EXECUTE PROCEDURE 语句来执行 SPL 函数。然而，对于新的 SPL 例程，推荐您对于过程仅使用 EXECUTE PROCEDURE，而对于函数仅使用 EXECUTE FUNCTION。

对于它的参数，用户定义的例程可使用输入函数。然而，对于它的例程名称，它不可使用输入参数。

用户定义的过程

如果您在编译时刻知道用户定义的过程（外部的或 SPL）的名称，则请以 EXECUTE PROCEDURE 语句执行用户定义的过程。下列 EXECUTE PROCEDURE 语句执行名为 revise_stats() 的用户定义的过程：

```
EXEC SQL execute procedure revise_stats("customer");
```

如果您直到运行时刻才知道用户定义的过程的名称，则您必须动态地执行该过程。要动态地执行用户定义的过程，您可使用：

PREPARE 和 EXECUTE 语句

EXECUTE IMMEDIATE 语句

用户定义的函数

如果您在编译时刻知道用户定义的函数的名称，则请以 EXECUTE FUNCTION 语句执行用户定义的（外部的或 SPL）函数。在 EXECUTE FUNCTION 的 INTO 子句中，请您罗列保存一个或多个返回值的主变量。下列 EXECUTE FUNCTION 语句执行名为 items_pct() 的用户定义的函数（图 1对其定义）：

```
EXEC SQL execute function items_pct(\"HSK\")
        into :manuf_percent;
```

如果您直到运行时刻才知道用户定义的函数的名称，则您必须动态地执行该函数。用户定义的函数的动态执行类似于 SELECT 语句的动态执行（处理未知的选择列表）。SELECT 和用户定义的函数都将值返回至 GBase 8s ESQL/C 程序。

请以 EXECUTE FUNCTION 语句执行用户定义的函数。您可以下列两种方式执行 EXECUTE FUNCTION 语句：

如果用户定义的函数仅返回一行，则请使用 PREPARE 和 EXECUTE INTO 来执行 EXECUTE FUNCTION 语句。此类用户定义的函数常常称为非游标函数。

如果用户定义的函数返回多行，则您必须声明函数游标来执行该 EXECUTE FUNCTION 语句。

此类用户定义的函数常常称为游标函数。以 SPL（和 SPL 函数）编写的游标函数在它的 RETURN 语句中有 WITH RESUME 子句。以诸如 C 这样的外部语言编写的游标函数是迭代函数。

提示： 如果您不知道返回值的数据类型，则您必须使用动态管理结构来保存该值。

非游标函数

您可使用 PREPARE 和 EXECUTE 语句来执行用户定义的非游标函数。非游标函数仅返回一行值。

您的 GBase 8s ESQL/C 程序必须采取下列行动：

声明主变量来接收数据库服务器返回的值。

对于准备好的 EXECUTE FUNCTION 语句，这些值是用户定义的函数的返回值。

组装并准备该语句。

准备好的 EXECUTE FUNCTION 语句可包含输入参数作为函数参数。

以 EXECUTE...INTO 语句执行准备好的用户定义的函数，随同 INTO 关键字之后的主变量。

如果 EXECUTE FUNCTION 包含数据参数，请包括 EXECUTE 的 USING 子句。

重要： 要执行非游标函数，与 DECLARE、OPEN 和 FETCH 语句相比，EXECUTE...INTO 更高效。然而，您常常不知道返回的行数。当您不使用游标来执行返回多行的游标函数时，GBase 8s ESQL/C 生成运行时刻错误。因此，总是将用户定义的函数与游标相关联是一个好做法。

大多数外部函数仅可返回一行数据和仅单个值。例如，下列代码段执行名为 stnd_dev() 的外部函数：

```
strcpy(func_name, "stnd_dev(ship_date)");
        sprintf(exfunc_stmt, "%s %s %s",
        "execute function",
        func_name);
        EXEC SQL prepare exfunc_id from :exfunc_stmt;
        EXEC SQL execute exfunc_id into :ret_val;
```

要返回多个值，外部函数必须返回复合的数据类型，比如集合或 row 类型。

SPL 函数可返回一个或多个值。如果 SPL 函数的 RETURN 语句不包含 WITH RESUME 关键字，则该函数仅返回一行。要动态地执行 SPL 函数，请准备该 EXECUTE FUNCTION，并以 EXECUTE...INTO 语句执行它。

函数游标

要执行其用户定义的函数返回多行的 EXECUTE FUNCTION 语句，您必须声明函数游标。函数游标使得 GBase 8s ESQL/C 应用程序能够处理用户定义的函数返回的多个行。

您的 GBase 8s ESQL/C 程序必须采取下列行动来使用函数游标：

声明主变量来接收用户定义的函数返回的值。

组装并准备该语句。

准备好的 EXECUTE FUNCTION 语句可包含输入参数作为函数参数。

声明函数游标。

DECLARE 语句将准备好的 EXECUTE FUNCTION 语句与函数游标相关联。

执行用户定义的函数。

OPEN 语句将它的 USING 子句指定的任何输入参数发送至数据库服务器，并告诉数据库服务器执行该 EXECUTE FUNCTION 语句。

从函数游标检索值的行。

FETCH 语句检索用户定义的函数返回的一行值。

仅为迭代函数的外部函数可返回多行数据。

如果 SPL 函数的 RETURN 语句包含 WITH RESUME 关键字，则该函数可返回多行。您必须将这样的 SPL 函数与函数游标相关联。要动态地执行该 SPL 函数，请将 EXECUTE FUNCTION 语句与游标相关联，使用 OPEN 语句来执行该函数，并使用 FETCH...INTO 语句来将行从游标检索至主变量内。

执行带有输入参数的语句

输入参数是 SQL 语句中的占位符，指示在运行时刻提供的实际的值。由于数据库服务器对应用程序中声明的变量一无所知，因此，您不可在动态 SQL 语句的文本中罗列主变量名称。反而，您可在表达式为有效的语句内的任何地方，以问号（?）指示输入参数，其作为占位符。您不可使用输入参数来代表诸如数据库名称、表名称或列名称这样的标识符。

包含输入参数的 SQL 语句称为参数化的语句。对于参数化的 SQL 语句，您的程序必须将下列关于其输入参数的信息提供给数据库服务器：

您的程序必须使用问号（?）作为该语句的文本中的占位符，来指示期望输入参数的位置。例如，下列 DELETE 语句包含两个输入参数：

```
EXEC SQL prepare dlt_stmt from
        'delete from orders where customer_num = ? \
        and order_date > ?';
```

为 **customer_num** 列的值定义第一个输入参数，为 **order_date** 列的值定义第二个。

当以 USING 子句执行该语句时，您的程序必须为输入参数指定该值。要执行前面步骤中的 DELETE 语句，您可使用下列语句：

```
EXEC SQL execute dlt_stmt using :cust_num, :ord_date;
```

在运行时刻您用于提供带有值的输入参数的语句，依赖于您执行的 SQL 语句的类型，如下：

对于带有输入参数的非 SELECT 语句（诸如 UPDATE、INSERT、DELETE 或 EXECUTE PROCEDURE），EXECUTE...USING 语句执行该语句，并提供输入参数值。

对于与游标相关联的 SELECT 语句，或对于游标函数（EXECUTE FUNCTION），OPEN...USING 语句执行该语句，并提供输入参数值。

对于单个 SELECT 语句，或对于非游标函数（EXECUTE FUNCTION），EXECUTE...INTO...USING 语句执行该语句，并提供输入参数值。

当该语句执行时，您可罗列主变量或文字值来替代 USING 子句中的每一输入参数。该值必须与相关联的输入参数在数目和数据类型上相兼容。主变量还必须大到足以保存该数据。

重要： 要以 USING 子句使用主变量，您必须知道 SQL 语句中参数的数目及其数据类型。在运行时刻，如果您不知道输入参数的数目和数据类型，则您必须以 USING 子句使用动态管理结构。

EXECUTE USING 语句

你可以 EXECUTE...USING 语句执行参数化的非 SELECT 语句（包含输入参数的非 SELECT）。

下列语句是参数化的非 SELECT 语句：

在 WHERE 子句中带有输入参数的 DELETE 或 UPDATE 语句

在 SET 子句中带有输入参数的 UPDATE 语句

在 VALUES 子句中带有输入参数的 INSERT 语句

带有其函数参数的输入参数的 EXECUTE PROCEDURE 语句

提示： 对于用户定义的过程，您不可使用输入参数作为过程名称。

例如，在其 WHERE 子句中，下列 UPDATE 语句需要两个参数：

```
EXEC SQL prepare upd_id from
        'update orders set paid_date = NULL \
        where order_date > ? and customer_num = ?';
```

USING 子句罗列保存参数数据的主变量的名称。如果在 **hvar1** 和 **hvar2** 中存储输入变量值，则您的程序可以下列语句执行此 UPDATE：

```
EXEC SQL execute upd_id using :hvar1, :hvar2;
```

当在编译时刻知道参数的类型和数目时，下列步骤描述如何处理参数化的 UPDATE 或 DELETE：

为在该准备好的语句中的每一输入参数声明主变量。

1. 为该语句组装该字符串，为每一输入参数带有问号（?）占位符。一旦您已组装了该字符串，就请准备它。要获取关于这些步骤的更多信息，

给与每一输入参数相关联的主变量指定一个值。（该应用程序可以交互地获取这些值。）

以 EXECUTE...USING 语句执行该 UPDATE 或 DELETE 语句。您必须罗列在 USING 子句中包含输入参数值的主变量。

可选地，使用 FREE 语句来释放随同准备好的语句分配了的资源。

重要： 在编译时刻，如果您不知道准备好的语句中的输入参数的数目和数据类型，请不要以 USING 子句使用主变量。反而，请使用动态管理结构来指定输入参数值。

OPEN USING 语句

您可以 OPEN...USING 语句执行下列语句：

返回一行或多行的参数化的 SELECT 语句（在其 WHERE 子句中包含输入参数的 SELECT 语句）

参数化的 EXECUTE FUNCTION 语句（对于其参数包含输入参数的游标函数）

提示： 对于用户定义的函数，您不可使用输入参数作为函数名称。

例如，下列 SELECT 语句是在其 WHERE 子句中需要两个参数的参数化的 SELECT：

```
EXEC SQL prepare slct_id from
        'select from orders where customer_num = ? and order_date > ?';
        EXEC SQL declare slct_cursor cursor for slct_id;
```

如果 **cust_num** 和 **ord_date** 主变量包含输入参数值，则下列 OPEN 语句已这些输入参数执行该 SELECT：

```
EXEC SQL open slct_id using :cust_num, :ord_date;
```

在编译时刻，仅当您知道 SELECT 语句的 WHERE 子句中的输入参数的类型和数目时，请使用 USING *host_var* 子句。

demo2.ec 样例程序

demo2.ec 样例程序展示如何处理在其 WHERE 子句中有输入参数的动态 SELECT 语句。

demo2.ec 程序使用主变量来保存 SELECT 语句的输入参数的值。它还使用主变量来保存从数据库返回的列值。

```
. #include <stdio.h>
        2. EXEC SQL define FNAME_LEN        15;
        3. EXEC SQL define LNAME_LEN        15;
        4. main()
        5. {
        6. EXEC SQL BEGIN DECLARE SECTION;
        7.      char demoquery[80];
        8.      char queryvalue[2];
        9.      char fname[ FNAME_LEN + 1 ];
        10.     char lname[ LNAME_LEN + 1 ];
        11. EXEC SQL END DECLARE SECTION;
        12.     printf("DEMO2 Sample ESQL program running.\n\n");
        13.     EXEC SQL connect to'stores7';
```

```
14. /* The next three lines have hard-wired the query. This
15.  * information could have been entered from the terminal
16.  * and placed into the demoquery string
17.  */
18.     sprintf(demoquery, "%s %s",
19.             "select fname, lname from customer",
20.             "where lname > ? ");
21.     EXEC SQL prepare demo2id from :demoquery;
```

行 9 和 10

这些行为 SELECT 语句的 WHERE 子句中的参数声明主变量（fname），并为 SELECT 语句返回的值声明主变量（fname 和 lname）。

行 14 - 21

这些行为该语句组装字符串（在 demoquery 中），并准备它作为 demo2id 语句标识符。问号（?）指示在该 WHERE 子句中的输入参数。

```
.     EXEC SQL declare demo2cursor cursor for demo2id;
23.     /* The next line has hard-wired the value for the parameter.
24.      * This information could also have been entered from the
* terminal
25.      * and placed into the queryvalue string.
26.      */
27.     sprintf(queryvalue, "C");
28.         EXEC SQL open demo2cursor using :queryvalue;
29.     for (;;)
30.         {
31.         EXEC SQL fetch demo2cursor into :fname, :lname;
32.         if (strncmp(SQLSTATE, "00", 2) != 0)
33.             break;
34.         /* Print out the returned values */
```

```
35.         printf("Column: fname\tValue: %s\n", fname);
36.         printf("Column: lname\tValue: %s\n", lname);
37.         printf("\n");
38.       }
```

行 22

此行为准备好的语句标识符 demo2id 声明 demo2cursor 游标。所有非单个 SELECT 语句都必须有声明了的游标。

行 23 - 27

queryvalue 主变量是 SELECT 语句的输入参数。它包含值 C。在交互的应用程序中，可能会从用户处获取此值。

行 28

当数据库服务器打开 demo2cursor 游标时，它执行该 SELECT 语句。因为 SELECT 语句的 WHERE 子句包含输入参数（行 20 和 21），因此，OPEN 语句包括 USING 子句来在 queryvalue 中指定输入参数值。

行 29 - 38

为从数据库访存的每一行执行此 for 循环。FETCH 语句（行 31）包括 INTO 子句来为列值指定 fname 和 lname 主变量。在此 FETCH 语句执行之后，在这些主变量中存储该列值。

```
.     if (strncmp(SQLSTATE, "02", 2) != 0)
40.         printf("SQLSTATE after fetch is %s\n", SQLSTATE);
41.     EXEC SQL close demo2cursor;
42.     EXEC SQL free demo2cursor;
43.     EXEC SQL free demo2id;
44.     EXEC SQL disconnect current;
```

```
45.     printf("\nProgram Over.\n");
46. }
```

行 39 和 40

在 **for** 循环外部，该程序再次测试 SQLSTATE 变量，以便它可在成功的执行、运行时刻错误或警告事件中通知用户（类代码不等于 "02"）。

行 41

在访存所有行之后，CLOSE 语句关闭 **demo2cursor** 游标。

行 42 和 43

这些 FREE 语句为准备好的语句（行 42）和数据库游标（行 43）释放分配的资源。一旦已释放了游标或准备好的语句，在该程序中就不可再次使用它。

4.1.7 在编译时刻不知道的 SQL 语句

在编译时刻不知道的 SQL 语句通常是用户在交互式应用程序中输入的语句。

当您编写像 DB-Access 一样的交互式数据库查询应用程序时，您提前不知道用户想要访问哪些数据库、表或列，或不知道用户可能在 WHERE 子句中应用什么条件。如果 GBase 8s ESQL/C 应用程序翻译并运行用户输入的 SQL 语句，则直到在运行时刻用户输入该语句之后，此应用程序才知道在主变量中要存储什么类型的信息。

例如，如果程序包含下列 DELETE 语句，则基于受影响的列，您知道接收的值的数目及数据类型：

```
DELETE FROM customer WHERE city = ? AND lname > ?
```

您可定义其数据类型与它们接收的数据类型相兼容的主变量。然而，假设您的程序为用户提供提示，诸如：

对于 stores7 数据库，请输入 DELETE 语句：

在此情况下，直到运行时刻，您才知道 DELETE 在其上发生的表名称，或在 WHERE

子句中罗列的列名称。因此，您不可声明必要的主变量。

您可动态地确定准备好的 SQL 语句以及关于它以 DESCRIBE 语句和动态管理结果访问的表和列的信息。

4.2 确定 SQL 语句

如果直到运行时刻，您才知道要执行什么 SQL 语句，则您可以 DESCRIBE 语句动态地确定该语句，并使用动态管理结构来保存该语句发送到数据库服务器的或从数据库服务器接收的任何值。

这些主题包含关于如何动态地确定 SQL 语句的下列信息：

存在什么动态管理结构，以及哪些 SQL 语句访问它们。

如何使用带有动态管理结构的 DESCRIBE 语句。

4.2.1 动态管理结构

如果您不知道发送至数据库服务器，或从数据库服务器接收的值的数目或数据类型，则请使用动态管理结构。动态管理结构允许您将数据的变长列表传至数据库服务器，或从它接收变长列表。

要执行带有位置的列的动态 SQL 语句，您可在您的 GBase 8s ESQL/C 程序中使用下列动态管理结构之一：

系统描述符区域是符合 X/Open 标准的与语言无关的数据结构。请您以 SQL 语句 ALLOCATE DESCRIPTOR、GET DESCRIPTOR、SET DESCRIPTOR 和 DEALLOCATE DESCRIPTOR 来分配和操纵它。

sqlda 结构是您以相同类型的 C 语言语句操作的 C 语言数据结构，您使用其来分配和操纵其他 C 结构（有 struct 数据类型的区域）。

由于此方法在 SQL 语句内使用 C 语言结构，因此它依赖于语言，且不符合 X/Open 标准。

对于给定的动态 SQL 语句，动态管理结构可保存任何下列信息：

该语句中未知列的数目

对于每一未知的值，数据类型和长度，数据的空间，以及关于任何相关联的变量的信息（它的数据类型、长度和数据）

然后，GBase 8s ESQL/C 程序可使用此信息来确定要保存该值的主变量适合的长度和

类型。

系统描述符区域

系统描述符区域是由 GBase 8s ESQL/C 声明来保存从准备好的语句返回的数据或由准备好的数据发送的数据的内存区域。它是符合 X/Open 标准的动态管理结构。

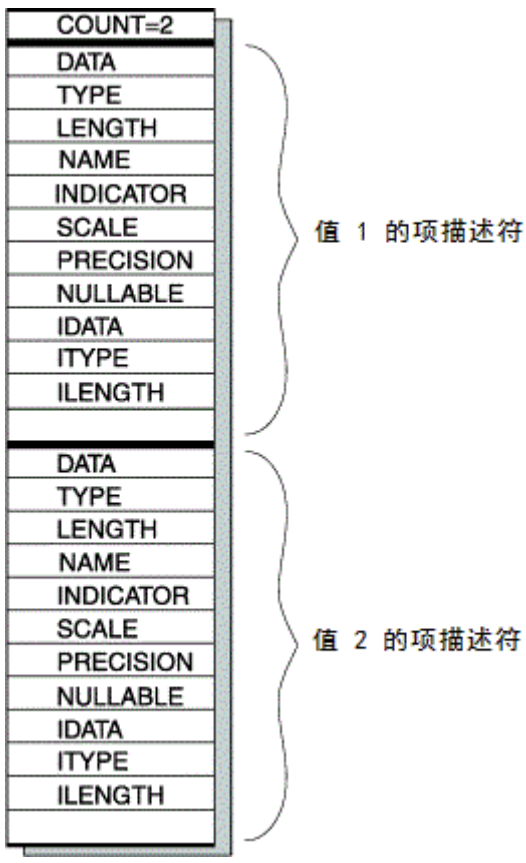
系统描述符区域有两个部分：

有 COUNT 字段构成的固定大小的部分。此字段包含在系统描述符区域中描述的列的数目。

对于系统描述符区域中每一值，包含项描述符的变长部分。每一项描述符是一个固定大小的结构。

对于两个值，下图展示系统描述符区域的样子。

图 1. 展示两个值的系统描述符区域的简图



定长部分

系统描述符区域的固定大小部分有单个字段构成，如下表所示。

表 8. 系统描述符区域的固定大小部分中的字段

字段	数据类型	描述
COUNT	short	系统描述符区域中列值的数目或发生率。这是项描述符的数目，每一列一个。DESCRIBE...USING SQL DESCRIPTOR 语句将 COUNT 设置为被描述的列的数目。在您将列值发送至数据库服务器之前，您必须使用 SET DESCRIPTOR 来初始化该字段。

项描述符

系统描述符区域中的每一项描述符保存关于可传送至数据库服务器或从数据库服务器收到的数据值的信息。

每一项描述符由下表总结的字段组成。

表 9. 系统描述符区域的每一项描述符中的字段

字段	数据类型	描述
DATA	char *	指向发送至数据库服务器或从数据库服务器接收的列数据的指针。
TYPE	short	表示正在转换的列的数据类型的整数。在 sqltypes.h 和 sqlxtype.h 头文件中定义这些值。
LENGTH	short	以字节计的 CHAR 类型数据的长度，DATETIME 或 INTERVAL 数据的编码的限定符，或 DECIMAL 或 MONEY 值的大小。
NAME	char *	指向包含列名称或正在转换的显示标签的字符数组的指针。
INDICATOR	short	可包含两个值之一的指示符变量： 0 需要 DATA 字段来包含非空数据。 -1 当未指定 DATE 字段值时，插入 NULL。
SCALE	short	包含在 DATA 字段中的列的范围；仅对于 DECIMAL 或 MONEY 数据类型定义。
PRECISION	short	包含在 DATA 字段中的列的精度；仅为 DECIMAL 或 MONEY 数据类型定义。

字段	数据类型	描述
NULLABLE	short	指定该列可否（在 DESCRIBE 语句之后）包含空值: 1 该列允许空值 0 该列不允许空值。 在执行 EXECUTE 语句或动态 OPEN 语句之前，必须将它设置为 1 来指示在 INDICATOR 字段中指定指示符值，而如果未指定它，则设置为 0。（当您执行动态 FETCH 语句时，忽略 NULLABLE 字段。
IDATA	char *	用户定义的指示符数据，或包含 DATA 字段的指示符语句的主变量名称。IDATA 字段不是标准 X/Open 字段。
ITYPE	short	用户定义的指示符变量的数据类型。在 sqltypes.h 和 sqlxtype.h 头文件中定义这些值。ITYPE 字段不是标准 X/Open 字段。
ILENGTH	short	以字节计的用户定义的指示符的长度。ILENGTH 字段不是标准 X/Open 字段。
EXTYPEID	int4	用户定义的(opaque 或 distinct)或复合的(集合或 row）数据类型的扩展的标识符。
EXTYPENAME	char *	用户定义的(opaque 或 distinct)或复合的(集合或 row）数据类型的名称。
EXTYPELENGTH	short	以字节计的 EXTYPENAME 字段中的字符串的长度。
EXTYPEOWNERNAME	char *	用户定义的(opaque 或 distinct)或复合的(集合或 row）数据类型的所有者的名称（对于 ANSI 数据库）。
EXTYPEOWNERLENGTH	short	以字节计算的 EXTYPEOWNERNAME 字段中的字符串的长度。
SOURCETYPE	short	对于 distinct 类型列，源数据类型的数据类型常量（来自 sqltypes.h）。
SOURCEID	int4	对于 distinct 类型列，源数据类型的扩展的标

字段	数据类型	描述
		识符。

sqlda 结构

sqlda 结构是保存从准备好的语句返回的数据的 C 结构（在 sqlda.h 头文件中定义）。

每一 **sqlda** 结构有三部分：

由 sqld 字段构成的定长部分，其包含在 sqlda 结构中描述的列的数目。

对于为每一列值，包含 sqlvar_struct 结构的变长部分。每一 sqlvar_struct 结构是定长的结构。

包括关于 sqlda 结构自身的描述性信息。

对于两个值，下图展示 sqlda 结构看上去的样子。

图 2. 展示两个值的 sqlda 结构的简图

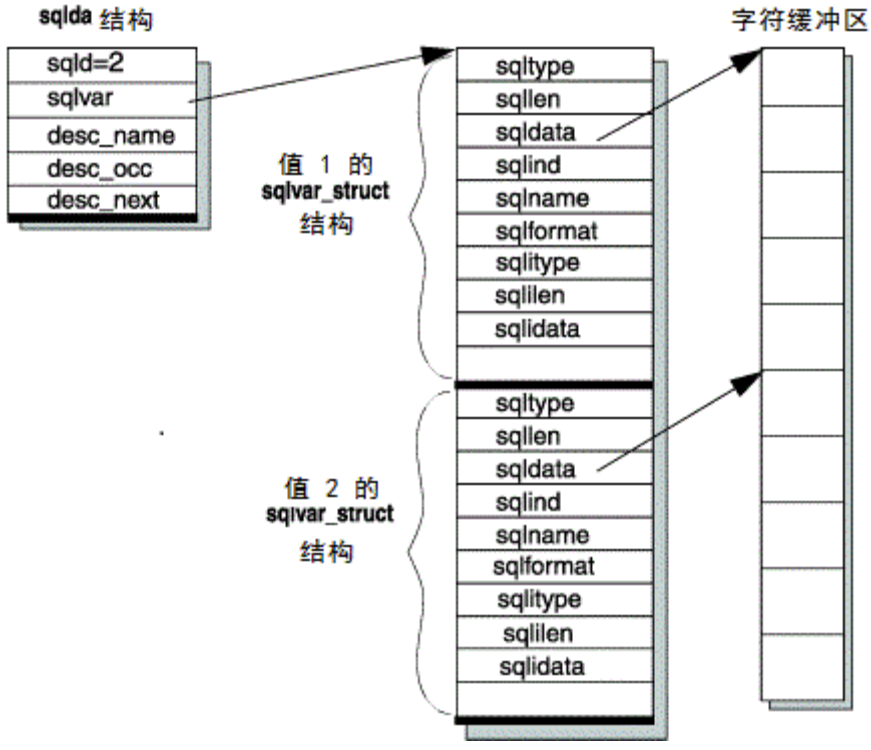


图 1展示单个数据缓冲区中 sqldata 字段中的列数据。可在分开的缓冲区中存储此数据。

定长部分

下表描述 **sqlda** 结构的定长部分，其由单个字段构成。

表 10. sqlda 结构的定长部分中的字段

字段	数据类型	描述
sqld	short	**sqlda** 中列值的数目或发生率。这是 **sqlvar_struct** 结构的数目，每一列一个。DESCRIBE...INTO 语句将 **sqld** 设置为被描述的列的数目。在您将列值发送至数据库服务器之前，您必须设置 **sqld** 来初始化该字段。

sqlvar_struct 结构

当完全地定义它的所有组件时，对于该集中的每一变量，sqlda 结构指向包含必要信息的 sqlvar_struct 结构的序列的初始地址。每一 sqlvar_struct 结构保存可发送至数据库服务器或从数据库服务器接收的数据值。您的程序通过 sqlda 的 sqlvar 字段访问这些 sqlvar_struct 结构。表 1和 表 2总结 sqlda 的变长结构。

表 11. 访问 sqlda 结构的变长部分的字段

字段	数据类型	描述
sqlvar	struct sqlvar_struct *	指向 sqlda 结构的变长部分的指针。从数据库服务器返回的或发送至数据库服务器的每一列值有一个 **sqlvar_struct**。**sqlvar** 字段指向第一个 **sqlvar_struct** 结构。

下表展示 **sqlvar_struct** 结构中的字段。

表 12. sqlvar_struct 结构中的字段

字段	数据类型	描述
sqltype	short	标识数据库服务器发送或接收的列的数据类型的整数。在 sqltypes.h 和 sqlxtype.h 头文件中定义这些值。
sqllen	short	以字节计的 CHAR 类型数据的产度，或 DATETIME 或 INTERVAL 值的编码的限定符。该长度的含义依赖于信息的类型，以及如何使用该 sqlda： 当您以 DESCRIBE 语句检索 sqlda 结构时，自动地将 **sqllen** 字段中的值设置为该数据在磁盘上占据的空间的长度。该值来自系统目录。 当您将数据访存至缓冲区内，或通过缓冲区发送数据时，您必须将 **sqllen** 字段中的值设置为用于该列的内存缓冲区的大小。

字段	数据类型	描述
sqldata	char *	指向数据库服务器发送或接收的列数据的指针。
sqlind	short *	对于可包含两个值之一的列，指向指示符变量指针： 0 **sqldata** 字段包含非空数据。 -1 **sqldata** 字段包含空数据。
sqlname	char *	指向包含列名称数据库服务器发送或接收的显示标签的字符数组的指针。
sqlformat	char *	为将来的使用保留。
sqlitype	short	指定用户定义的指示符变量的数据类型的整数。在 sqltypes.h 和 sqlxtype.h 头文件中定义这些值。
sqlilen	int4	以字节计的用户定义的指示符变量的长度。
sqlidata	char *	指向用户定义的指示符变量的数据的指针。
sqlxid	int4	用户定义的（opaque 或 distinct）或复合的（集合或 row）数据类型的扩展的标识符。
sqltypename	char *	用户定义的（opaque 或 distinct）或复合的（集合或 row）数据类型的名称。
sqltypelen	short	以字节计的 **sqltypename** 字段中的字符串的长度。
sqlownername	char *	用户定义的（opaque 或 distinct）或复合的（集合或 row）数据类型的所有者的名称（对于 ANSI 数据库）。
sqlownerlen	short	以字节计的 **sqlownername** 字段中的字符串的长度。
sqlsourcetype	short	对于 distinct 类型列，源数据类型的数据类型常量（来自 sqltypes.h）。
sqlsourceid	int4	对于 distinct 列，源数据类型的扩展的标识符。
sqlflags	int4	通常内部使用此字段。然而，如果由 DESCRIBE 语句已初始化了该 **sqlda** 结构，则您可通过使用在 sqltypes.h 中定义的 **ISCOLUMNULLABLE()** 宏来确定该列是否接受空。如果它返回 1，则该列接受空。 在 sqltypes.h 中定义 **ISCOLUMNULLABLE()** 宏。

描述性信息

下表总结描述 **sqlda** 结构自身的 **sqlda** 字段。

表 13. 在 sqlda 结构中的描述性字段

字段	数据类型	描述
desc_name	char[19]	描述符的名称；最大 18 字符
desc_occ	short	sqlda 结构的大小
desc_next	struct sqlda *	指向下一 sqlda 结构的指针

4.2.2 DESCRIBE 语句

DESCRIBE 语句获取关于准备好的语句中数据库列或表达式的信息。

DESCRIBE 语句可将此信息放置在下列动态管理结构之一中：

DESCRIBE...USING SQL DESCRIPTOR 在系统描述符区域中存储信息。

每一项描述符描述一列。将 COUNT 字段设置为项描述符的数目（在列列表中的列数）。你可以 GET DESCRIPTOR 语句访问此信息。要获取关于系统描述符区域的字段的更多信息，请参阅 图 1至 表 1。

DESCRIBE...INTO sqlda_ptr 在 sqlda 结构中存储信息，该结构的地址存储在 sqlda_ptr 中。

每一 sqlvar_struct 结构描述一列。将 sqld 字段设置为 sqlvar_struct 结构的数目（在该列列表中的列数）。您可通过 sqlvar_struct 结构中的字段访问此信息。要获取关于 sqlda 结构的字段的信息，请参阅 图 1至 表 1。

重要： 如果启用“延迟的 PREPARE”特性，则您不可在 OPEN 语句执行之前使用 DESCRIBE 语句。

如果 DESCRIBE 是成功的，则它获取关于准备好的语句的下列信息：

SQLCODE 值指示准备了的语句的类型。

动态管理结构包含关于 SELECT、INSERT 或 EXECUTE FUNCTION 语句的列列表中列的数目和数据类型的信息。

当 DESCRIBE 语句描述 DELETE 或 UPDATE 语句时，它可指示该语句是否包括 WHERE 子句。

确定语句类型

对于可准备的 SQL 语句，sqlstype.h 文件包含定义的整数常量。DESCRIBE 语句在

SQLCODE（**sqlca.sqlcode**）变量中返回这些值之一，来标识准备好的语句。即，SQLCODE 指示该语句是否为 INSERT、SELECT、CREATE TABLE 或任何其他 SQL 语句。

在使用动态 SQL 语句的 GBase 8s ESQL/C 程序内，您可使用下表展示的常量，来确定准备了哪个 SQL 语句。

表 14. sqlstype.h 文件定义的 SQL 语句类型的常量

SQL 语句	定义的 sqlstype.h 常量	值
SELECT（无 INTO TEMP 子句）	无	0
DATABASE	SQ_DATABASE	1
	仅限于内部使用	2
SELECT INTO TEMP	SQ_SELINTO	3
UPDATE...WHERE	SQ_UPDATE	4
DELETE...WHERE	SQ_DELETE	5
INSERT	SQ_INSERT	6
UPDATE WHERE CURRENT OF	SQ_UPDCURR	7
DELETE WHERE CURRENT OF	SQ_DELCURR	8
	仅限于内部使用	9
LOCK TABLE	SQ_LOCK	10
UNLOCK TABLE	SQ_UNLOCK	11
CREATE DATABASE	SQ_CREADB	12
DROP DATABASE	SQ_DROPDB	13
CREATE TABLE	SQ_CRETAB	14
DROP TABLE	SQ_DRPTAB	15
CREATE INDEX	SQ_CREIDX	16
DROP INDEX	SQ_DRPIDX	17
GRANT	SQ_GRANT	18
REVOKE	SQ_REVOKE	19
RENAME TABLE	SQ_RENTAB	20
RENAME COLUMN	SQ_RENCOL	21
CREATE AUDIT	SQ_CREAUD	22
	仅限于内部使用	23–28

SQL 语句	定义的 sqlstype.h 常量	值
ALTER TABLE	SQ_ALTER	29
UPDATE STATISTICS	SQ_STATS	30
CLOSE DATABASE	SQ_CLSDB	31
DELETE（无 WHERE 子句）	SQ_DELALL	32
UPDATE（无 WHERE 子句）	SQ_UPDALL	33
BEGIN WORK	SQ_BEGWORK	34
COMMIT WORK	SQ_COMMIT	35
ROLLBACK WORK	SQ_ROLLBACK	36
	仅限于内部使用	37–39
CREATE VIEW	SQ_CREVIEW	40
DROP VIEW	SQ_DROPVIEW	41
	仅限于内部使用	42
CREATE SYNONYM	SQ_CREASYN	43
DROP SYNONYM	SQ_DROPSYN	44
CREATE TEMP TABLE	SQ_CTEMP	45
SET LOCK MODE	SQ_WAITFOR	46
ALTER INDEX	SQ_ALTIDX	47
SET ISOLATION、SET TRANSACTION	SQ_ISOLATE	48
SET LOG	SQ_SETLOG	49
SET EXPLAIN	SQ_EXPLAIN	50
CREATE SCHEMA	SQ_SCHEMA	51
SET OPTIMIZATION	SQ_OPTIM	52
CREATE PROCEDURE	SQ_CREPROC	53
DROP PROCEDURE	SQ_DRPPROC	54
SET CONSTRAINTS	SQ_CONSTRMODE	55
EXECUTE PROCEDURE、EXECUTE FUNCTION	SQ_EXECPROC	56
SET DEBUG FILE TO	SQ_DBGFILE	57
SET MOUNTING TIMEOUT	SQ_OPTIMEOUT	63
对于过程 UPDATE STATS...	SQ_PROCSTATS	64
	仅为 Kanji 版本定义	65 和

SQL 语句	定义的 sqlstype.h 常量	值
		66
	保留的	67–69
CREATE TRIGGER	SQ_CRETRIG	70
DROP TRIGGER	SQ_DRPTRIG	71
	SQ_UNKNOWN	72
SET DATASKIP	SQ_SETDATASKIP	73
SET PDQPRIORITY	SQ_PDQPRIORITY	74
ALTER FRAGMENT	SQ_ALTFRAG	75
SET	SQ_SETOBJMODE	76
START VIOLATIONS TABLE	SQ_START	77
STOP VIOLATIONS TABLE	SQ_STOP	78
	仅限于内部使用	79
SET SESSION AUTHORIZATION	SQ_SETDAC	80
	仅限于内部使用	81-82
CREATE ROLE	SQ_CREATEROLE	83
DROP ROLE	SQ_DROPROLE	84
SET ROLE	SQ_SETROLE	85
	仅限于内部使用	86–;89
CREATE ROW TYPE	SQ_CREANRT	90
DROP ROW TYPE	SQ_DROPNRT	91
CREATE DISTINCT TYPE	SQ_CREADT	92
CREATE CAST	SQ_CREACT	93
DROP CAST	SQ_DROPCT	94
CREATE OPAQUE TYPE	SQ_CREABT	95
DROP TYPE	SQ_DROPTYPE	96
	保留的	97
CREATE ACCESS_METHOD	SQ_CREATEAM	98
DROP ACCESS_METHOD	SQ_DROPAM	99
	保留的	100
CREATE OPCLASS	SQ_CREATEOPC	101

SQL 语句	定义的 sqlstype.h 常量	值
DROP OPCLASS	SQ_DROPOPC	102
CREATE CONSTRUCTOR	SQ_CREACST	103
SET (MEMORY/NON)_RESIDENT	SQ_SETRES	104
CREATE AGGREGATE	SQ_CREAGG	105
DROP AGGREGATE	SQ_DRPAGG	106
onutil 检查索引命令	SQ_CHKIDX	108
设置日程	SQ_SCHEDULE	109
"set environment..."	SQ_SETENV	110
	保留的	111
	保留的	112
	保留的	113
	保留的	114
SET STMT_CACHE	SQ_STMT_CACHE	115
RENAME INDEX	SQ_RENIDX	116
CREATE SEQUENCE	SQ_CRESEQ	124
DROP SEQUENCE	SQ_DRPSEQ	125
ALTER SEQUENCE	SQ_ALTERSEQ	126
RENAME SEQUENCE	SQ_RENSEQ	127
SET COLLATION	SQ_COLLATION	129
SET NO COLLATION	SQ_NOCOLLATION	130
SET ROLE DEFAULT	SQ_SETDEFROLE	131
SET ENCRYPTION	SQ_ENCRYPTION	132
保存外部伪指令	SQ_EXTD	133
CREATE XAdatasource TYPE	SQ_CRXASRCTYPE	134
CREATE XAdatasource	SQ_CRXADTSRC	135
DROP XAdatasource TYPE	SQ_DROPXATYPE	136
DROP XAdatasource	SQ_DROPXADTSRC	137
截断表	SQ_TRUNCATE	138
CREATE SECURITY LABEL COMPONENT	SQ_CRESECCMP	139
ALTER SECURITY LABEL	SQ_ALTSECCMP	140

SQL 语句	定义的 sqlstype.h 常量	值
COMPONENT		
DROP SECURITY LABEL COMPONENT	SQ_DRPSECCMP	141
RENAME SECURITY LABEL COMPONENT	SQ_RENSECCMP	142
CREATE SECURITY POLICY	SQ_CRESECPOL	143
DROP SECURITY POLICY	SQ_DRPSECPOL	144
RENAME SECURITY POLICY	SQ_RENSECPOL	145
CREATE SECURITY LABEL	SQ_CRESECLAB	146
DROP SECURITY LABEL	SQ_DRPSECLAB	147
RENAME SECURITY LABEL	SQ_RENSECLAB	148
GRANT DBSECADM	SQ_GRTSECADM	149
REVOKE DBSECADM	SQ_RVKSECADM	150
GRANT EXEMPTIONS	SQ_GRTSECEXMP	151
REVOKE EXEMPTIONS	SQ_RVKSECEXMP	152
GRANT SECURITY LABEL	SQ_GRTSECLAB	153
REVOKE SECURITY LABEL	SQ_RVKSECLAB	154
GRANT SETSESSIONAUTH	SQ_GRTSESAUTH	155
REVOKE SETSESSIONAUTH	SQ_RVKSESAUTH	156

提示： 对于最新的 SQL 语句类型值的列表，请检查在您的系统上的 sqlstype.h 头文件。

要确定动态地准备了的 SQL 语句的类型，您的 GBase 8s ESQL/C 程序必须采取下列行动：

使用 include 伪指令来包括 sqlstype.h 头文件。

将 SQLCODE 变量中的值（sqlca.sqlcode）与在 sqlstype.h 文件中定义的常量相对比。

执行 SPL 函数的样例程序使用 SQ_EXECPROC 常量来验证已准备了 EXECUTE FUNCTION 语句。

确定列的数据类型

DESCRIBE 语句以整数值标识列的数据类型。

在 DESCRIBE 分析准备好的语句之后，它将此值存储在动态管理结构中，如下：

在系统描述符区域中，对于描述的每一列，在项描述符的 TYPE 字段中

在 sqlda 结构中，对于描述的每一列，在 sqlvar_struct 结构的 sqltype 字段中

GBase 8s ESQL/C 在下列两个头文件中为这些数据类型提供定义了的常量：

对于 GBase 8s 特定的 SQL 数据类型，sqltypes.h 头文件包含定义了的常量。这些值是 DESCRIBE 语句使用的缺省值。

对于 X/Open SQL 数据类型，sqlxtype.h 头文件包含定义了的常量。当您以预处理器的 -xopen 选项，编译您的 GBase 8s ESQL/C 源文件时，DESCRIBE 使用这些值。

请使用来自 sqltypes.h 或 sqlxtype.h 的 SQL 数据类型常量来分析 DESCRIBE 语句返回的信息，或在执行前设置列的数据类型。

提示： 当您在系统描述符区域中设置列的数据类型时，请您以 SET DESCRIPTOR 语句来将数据类型常量指定给项描述符的 TYPE 字段（可选地，以及 ITYPE 字段）。当您在 **sqlda** 结构中设置列的数据类型时，请您将数据类型常量指定给 **sqlvar** 结构的 **sqltype** 字段（可选地，以及 **sqlitype** 字段）。

特定于 GBase 8s 的 SQL 数据类型

在 GBase 8s 数据库中的列可用特定于 GBase 8s 的 SQL 数据类型。

当您编译您的 GBase 8s ESQL/C 程序时，如果您未包括 **-xopen** 选线，则 DESCRIBE 语句使用这些数据类型来指定列的数据类型或用户定义的函数的返回值。在 GBase 8s ESQL/Csqltypes.h 头文件中定义这些 GBase 8s SQL 数据类型的常量。

下图展示 sqltypes.h 中的一些 SQL 数据类型条目。

图 3. 一些 GBase 8s SQL 数据类型常量

```
#define SQLCHAR             0
#define SQLSMINT            1
#define SQLINT              2
#define SQLFLOAT            3
#define SQLSMFLOAT          4
```

```
#define SQLDECIMAL          5
#define SQLSERIAL           6
#define SQLDATE             7
#define SQLMONEY            8
?
```

要获取 SQL 数据类型的常量的完整列表，请参阅 表 1。在 图 1中的整数值是与语言无关的常量；在所有 GBase 8s 嵌入的产品中，它们都一样。

X/Open SQL 数据类型

X/Open 标准仅支持特定于 GBase 8s 的 SQL 数据类型的子集。要符合 X/Open 标准，您必须使用 X/Open SQL 数据类型常量。

当您以 **-xopen** 选项来编译您的 GBase 8s ESQL/C 程序时，DESCRIBE 语句使用这些常量来指定列（或返回值）的数据类型。

在 sqlxtype.h 头文件中定义 X/Open 数据类型常量。

ESQL/C 数据类型的常量

sqltypes.h 头文件包含 GBase 8s ESQL/C 数据类型的定义了的常量。

在 GBase 8s ESQL/C 程序中将 GBase 8s ESQL/C 数据类型指定给主变量。如果您的程序初始化列描述，则它总是从 GBase 8s ESQL/C 主变量获得该列值。要为此值设置列数据类型，该程序必须使用 GBase 8s ESQL/C 数据类型。

下列代码段仅展示 sqltypes.h 头文件中一部分 GBase 8s ESQL/C 数据类型。要获取 GBase 8s ESQL/C 数据类型的常量的完整列表，请参阅 表 1。

```
#define CCHARTYPE             100
        #define CSHORTTYPE            101
        #define CINTTYPE              102
        #define CLONGTYPE             103
        #define CFLOATTYPE            104
        #define CDOUBLETYPE           105
        ?
```

在使用动态 SQL 语句的 GBase 8s ESQL/C 程序内，您可使用前面的代码段中展示的那些常量来设置与主变量相关联的数据类型。请使用 GBase 8s

ESQL/C 数据类型来设置用作动态地定义了的 SQL 语句的输入参数的主变量的数据类型，或作为由数据库服务器返回的列值的存储。执行动态 INSERT 语句的样例程序将 TEXT 值存储至数据库表内。

确定输入参数

在执行它之前，您可使用 DESCRIBE 和 DESCRIBE INPUT 来为准备好的语句返回输入参数。

DESCRIBE INPUT 语句返回值的数目、数据类型和大小，以及该查询返回的列或表达式的名称 DESCRIBE INPUT 语句可为下列语句返回参数信息：

使用 WHERE 子句的 INSERT

使用 WHERE 子句的 UPDATE

带有 IN、BETWEEN、HAVING 和 ON 子句的 SELECT。

SELECT 子查询

SELECT INTO TEMP

DELECT

EXECUTE

检查 WHERE 子句

当 DESCRIBE 分析准备好的 DELETE 或 UPDATE 语句时，它指示该语句是否包括 WHERE 子句，如下：

如果该准备好的语句是不带有 WHERE 子句的 UPDATE 或 DELETE，则它将 sqlca.sqlwarn.sqlwarn0 和 sqlca.sqlwarn.sqlwarn4 字段设置为 W。

它将 SQLSTATE 变量设置为警告值 "01I07"，其特定于 GBase 8s。

您的程序可检查这些条件之一，来确定执行了的 DELETE 或 UPDATE 语句的类型。如果 DELETE 或 UPDATE 未包含 WHERE 子句，则数据库服务器删除或更新该表中的所有行。

4.2.3 在运行时刻确定语句信息

当您在下列条件之下执行 SQL 语句时，请考虑动态管理结构：

关于 SQL 语句的结构，有些情况是未知的：

要执行的语句的类型是未知的。

表名称是未知的，因此，要访问的列是未知的。

找不到 WHERE 子句。

关于在 GBase 8s ESQL/C 程序与数据库服务器之间传递的值的数目或类型，有些情

况是未知的：

在 SELECT 的选择列表中，或在 INSERT 的列列表中的列的数目和数据类型

在该语句中输入参数的数目和数据类型是未知的

（以 EXECUTE FUNCTION 语句执行的）用户定义的函数的返回值的数目和数据类型是未知的。

处理未知的选择列表

对于 SELECT 语句，选择列表中的列标识从数据库服务器接收的列值。在 demo1.ec 示例程序中描述和说明的 SELECT 语句中，将从该查询返回的值放置至主变量内，该主变量罗列在该 SELECT 语句的 INTO *host_var* 子句中。

然而，当您的程序在运行时刻创建 SELECT 语句时，您不可使用 INTO 子句，因为在编译时刻，您不知道需要什么主变量。在编译时刻，如果您的 GBase 8s ESQL/C 程序接收的值的类型和数目是未知的，则您的程序必须执行下列任务：

声明动态管理结构来作为选择列表列定义的存储。此结构可为系统描述符区域，也可为 **sqlda** 结构。

系统描述符区域的使用符合 X/Open 标准。

使用 DESCRIBE 语句来检测准备好的 SELECT 语句的选择列表，并描述这些列。

指定该动态管理结构作为从该数据库访存的数据的位置。该程序可将列值从该动态管理结构移至主变量内。

重要： 在编译时刻，仅当您不知道选择列表列的数目和数据类型时，请使用动态管理结构。

处理未知的列列表

对于 INSERT 语句，VALUES 子句中的值标识要插入至新行内的列值。在编译时刻，如果 GBase 8s ESQL/C 程序插入的值的数据类型和数目是未知的，则您不可简单地使用主变量来保存正被插入的数据。反而，您的程序必须执行下列任务：

定义动态管理结构来作为位置的列定义的存储。此结构可为系统描述符区域，也可为 sqlda 结构。

系统描述符区域的使用符合 X/Open 标准。

使用 DESCRIBE 语句来检测该准备好的 INSERT 语句的列列表，并描述这些列。

当 INSERT 语句执行时，指定动态管理结构作为要插入的数据的位置。

重要： 在编译时刻，仅当您不知道列列表列的数目和数据类型时，才使用动态管理结构。

确定未知的输入参数

如果您知道 SQL 语句的输入参数的数据类型和数目，则请使用 USING *host_var* 子句。然而，如果在编译时刻您不知道这些输入参数的数据类型和数目，则您不可使用主变量来提供该参数值；您没有关于该参数的充分的信息来声明主变量。

您也不可使用 DESCRIBE 语句来定义未知的参数，因为 DESCRIBE 不检测：

WHERE 子句（对于 SELECT、UPDATE 或 DELETE 语句）

用户定义的例程的参数（对于 EXECUTE FUNCTION 或 EXECUTE PROCEDURE 语句）

您的 GBase 8s ESQL/C 程序必须按照这些步骤来定义任何上述语句中的输入参数：

确定输入参数的数目和数据类型。除非您编写通用目的的、交互的翻译器，否则，您总是有此信息。如果您没有它，则您必须编写 C 分析该语句字符串的代码，并获得下列信息：

出现在该语句字符串的 WHERE 子句中的输入参数 [问号（?）]的数目，或作为用户定义的例程的参数

基于它与其对应的列（对于 WHERE 子句）或参数（对于参数）的每一输入参数的数据类型

存储定义和在动态管理结构中的输入参数的值。此结构可为系统描述符区域，也可为 sqlda 结构。

系统描述符区域的使用符合 X/Open 标准。

当该语句执行时，指定动态管理结构作为输入参数值的位置。

重要： 在编译时刻，仅当您不知道输入参数的数目和数据类型时，请使用动态管理结构。

动态地确定返回值

对于 EXECUTE FUNCTION 语句，在 INTO 子句中的值标识将用户定义的函数的返回值存储在哪里。在编译时刻，如果不知道该函数返回值的数据类型和数目，则您不可在 EXECUTE FUNCTION 的 INTO 子句中使用主变量来保存这些值。反而，您的程序必须执行下列任务：

定义动态管理结构来作为用户定义的函数返回的一个值或多个值的定义的存储。

您可使用系统描述符区域，也可使用 **sqlda** 结构来保存返回的一个值或多个值。

系统描述符区域的使用符合 X/Open 标准。

使用 DESCRIBE 语句来检测准备好的 EXECUTE FUNCTION 语句，并描述返回的一个值或多个值。

指定动态管理结构作为由用户定义的函数返回的数据的位置。

该程序可将返回值从动态管理结构移至主变量内。

重要： 在编译时刻，仅当您不知道用户定义的函数返回的返回值的数目和数据类型时，才使用动态管理结构。

处理包含用户定义的数据类型的语句

此部分提供如何执行包含带有下列用户定义的数据类型的列的动态 SQL 语句的信息：

Opaque 数据类型：用户可定义的封装了的数据类型

Distinct 数据类型：与其源类型有相同的内部存储表示的数据类型，但有不同的名称

带有 opaque 类型列的 SQL 语句

对于 opaque 类型列的动态执行，请记住下列项：

您必须确保动态管理结构（系统描述符区域或 sqlda 结构）的类型和长度与您插入至 opaque 类型列内的值的数据类型相匹配。

如果该主变量不大得足以保存该数据，则 GBase 8s ESQL/C 截断 opaque 类型数据为 32 KB。

插入 opaque 类型数据

当 DESCRIBE 语句描述准备好的 INSERT 语句时，它将动态管理结构的类型和长度字段设置为该列的数据类型。

下表展示动态管理结构的类型和长度字段。

表 15. 动态管理结构的类型和长度字段

动态管理结构	类型字段	长度字段
系统描述符区域	项描述符的 TYPE 字段	项描述符的 LENGTH 字段
sqlda 结构	sqlvar_struct 结构的 sqltype 字段	sqlvar_struct 结构的 sqllen 字段

如果 INSERT 语句包含其数据类型为 opaque 数据类型的列，则 DESCRIBE 以下列类型字段值之一来标识此列：

定长 opaque 类型的 SQLUDTFIXED 常量

变长 opaque 类型的 SQLUDTVAR 常量

这些数据类型常量表示采用其内部格式的 opaque 类型。

当您将 opaque 类型数据放置至动态管理结构内时，您必须确保类型字段和长度字段与您为 INSERT 提供的数据的数据类型相兼容，如下：

如果您提供采用内部格式的 opaque 类型数据，则 DESCRIBE 设置的类型和长度字段是正确的。

如果您以外部格式（或任何不同于内部格式的格式）提供该数据，则您必须更改类型和长度字段，DESCRIBE 已将这些字段设置为与该数据的数据类型相兼容的。

该 opaque 类型的输入和输出支持函数不在客户机计算机上。因此，客户机应用程序不可调用它们来将动态管理结构中的 opaque 类型数据由其外部格式转换为其内部格式。要以其外部表示提供 opaque 类型数据，请将类型字段值设置为字符数据类型。当数据库服务器接收字符数据（opaque 类型的外部表示）时，它调用输入支持函数来将 opaque 类型的外部表示转换为其内部表示。如果该数据是某其他类型，且存在有效的支持或强制转型函数，则数据库服务器可调用这些函数，而不是转换值。

例如，假设您使用系统描述符区域来保存插入值，且您想要将 opaque 类型数据以其外部表示发送至数据库服务器。在下列代码段中，SET DESCRIPTOR 语句将 TYPE 字段重置为 SQLCHAR，以便于 TYPE 字段与它指定给 DATA 字段的主变量（char）的数据类型相匹配：

```
EXEC SQL BEGIN DECLARE SECTION;
        char extrn_value[100];
        int extrn_lngth;
        int extrn_type;
        EXEC SQL END DECLARE SECTION;
        ⋮

        EXEC SQL allocate descriptor 'desc1' with max 100;
```

```
EXEC SQL prepare ins_stmt from
'insert into tab1 (opaque_col) values(?)';
EXEC SQL describe ins_stmt using sql descriptor 'desc1';

/* At this point the TYPE field of the item descriptor is
* SQLUDTFIXED
*/

stcopy("(1, 2, 3, 4)", extrn_value);
extrn_lngth = stleng(extrn_value);
dtype = SQLCHAR;

/* This SET DESCRIPTOR statement assigns the external
* representation of the data to the item descriptor and
* resets the TYPE field to SQLCHAR.
*/
EXEC SQL set descriptor 'desc1' value 1
data = :extrn_value, type = :extrn_type,
length = :extrn_lngth;
EXEC SQL execute ins_stmt using sql descriptor 'desc1';
```

opaque 类型数据的截断

如果您指定主变量，该主变量不够大到保存来自服务器的完全的返回值，则 GBase 8s ESQL/C 通常截断该数据以适应主变量，并将实际的长度放置在指示符变量中。

此指示符变量可为您显式地提供的，或对于动态 SQL，是动态管理结构的下列字段之一。

动态管理结构

指示符字段

系统描述符区域

项描述符的 INDICATOR 字段

sqlda 结构

sqlvar_struct 结构的 sqlind 字段

然而，这些指示符字段被定义作为 **short** 整数，因此，存储大小最大仅为 32 KB。

对指示符字段的此大小限制影响 GBase 8s ESQL/C 处理大于 32 KB 的 opaque 类型数据的截断方式。当 GBase 8s ESQL/C 收到大于 32 KB 的 opaque 类型数据，且该主变量不够大到足以保存该 opaque 类型数据时，GBase 8s ESQL/C 将该数据截断为 32 KB。GBase 8s ESQL/C 在 32 KB 处执行此截断，即使您编程主变量大于 32 KB（但仍然不够大到足以保存该数据）也不行。

带有 distinct 类型列的 SQL 语句

对于 distinct 类型列的动态执行，已经修改了动态管理结构来保存下列关于 distinct 类型的信息：

对于 distinct 类型列的源类型，数据类型常量（来自 sqltypes.h）

distinct 类型列的源类型的扩展的标识符

这些值位于动态管理结构的下列字段中。

动态管理结构	源类型字段	扩展的标识符字段
系统描述符区域	项描述符的 SOURCETYPE 字段	项描述符的 SOURCEID 字段
sqlda 结构	sqlvar_struct 结构的 sqlsourcetype 字段	sqlvar_struct 结构的 sqlsourceid 字段

当 DESCRIBE 语句描述准备好的语句时，它将关于该语句的列的信息存储在动态管理结构中。在 sqltypes.h 文件中没有特殊的常量来指示 distinct 数据类型。因此，动态管理结构的类型字段不可直接地指示 distinct 类型。（表 1展示动态管理结构的类型字段。）

反而，动态管理结构中的类型字段有一个特殊的值来指示为 distinct 类型列设置 distinct 位。该类型字段与 distinct 位相组合来指示该 distinct 数据的源类型。 sqltypes.h 头文件提供下列数据类型常量和宏，来为 distinct 列标识 distinct 位。

源类型	Distinct 位常量	Distinct 位宏
LVARCHAR	SQLDLVARCHAR	ISDISTINCTLVARCHAR(*type_id*)
BOOLEAN	SQLDBOOLEAN	ISDISTINCTBOOLEAN(*type_id*)

源类型	Distinct 位常量	Distinct 位宏
任何其他数据类型	SQLDISTINCT	ISDISTINCTTYPE(*type_id*)

使用下列算法来确定一列是否为 distinct 类型：

```
if (one of the distinct bits is set)
                                        {
                                        /* Have a distinct type, now find the
source type */
                                        if (ISDISTINCTLVARCHAR(sqltype))
                                        {
                                        /* Is a distinct of LVARCHAR:
                                        *     type field = SQLUDTVAR +
SQLDLVARCHAR
                                        *     source-type field = 0
                                        *     source-id field = extended identifier
of lvarchar
                                        */
                                        }
                                        else if
(ISDISTINCTBOOLEAN(sqltype))
                                        {
                                        /* Is a distinct of BOOLEAN
                                        *     type field = SQLUDTFIXED +
SQLDBOOLEAN
                                        *     source-type field = 0
                                        *     source-id field = extended id of
boolean
                                        */
                                        }
                                        else
                                        {
                                        /* SQLDISTINCT is set */
                                        if (ISUDTTYPE(sqltype))
                                        {
                                        /* Source type is either a built-in simple
```

```
type or an
                                        * opaque data type
                                        */
                                        if (source-id field > 0)
                                        /* Is a distinct of an opaque type.
                                        * Pick up the xtended identifier of the
source type
                                        * from the source-id field
                                        */
                                        else
                                        /* Is a distinct of a built-in simple type.
                                        * Pick up the type id of the source type
from the
                                        * source-type field
                                        */
                                        }
                                        else
                                        {
                                        /* Source type is a non-simple type, a
complex type.
                                        * Both the source-type and source-id
fields should be 0,
                                        * the source type is embedded in the type
field:
                                        *       type = source type +
SQLDISTINCT
                                        */
                                        }
                                        }
                                        }
```

下表总结前面算法的伪代码。

源类型	类型字段	源类型字段	扩展的标识符字段

源类型	类型字段	源类型字段	扩展的标识符字段
内建的数据类型	SQLUDTVAR + SQLDISTINCT	内建的数据类型的数据类型常量	0
LVARCHAR	SQLUDTVAR + SQLDLVARCHAR	0	LVARCHAR 的扩展标识符
BOOLEAN	SQLUDTFIXED + SQLDBOOLEAN	0	BOOLEAN 的扩展标识符
任何其他数据类型	源类型 + SQLDISTINCT	0	0

4.2.4 访存数组

访存数组使得您能够在您的程序中增加单个 FETCH 语句从访存缓冲区返回至 sqlda 结构的行数。当您访存简单大对象（TEXT 或 BYTE）数据时，访存数组特别有用。

对不带有访存数组的简单大对象的访存，需要与数据库服务器进行下列两项交换：

当 GBase 8s ESQL/C 访存 TEXT 或 BYTE 列时，数据库服务器为该列返回描述符。

然后，GBase 8s ESQL/C 请求数据库服务器获取该列数据。

当您使用访存数组时，GBase 8s ESQL/C 将一些列简单大对象描述符发送至数据库服务器，且数据库服务器一次返回所有对应的列数据。

使用访存数组

要使用访存数组：

声明 **sqlda** 结构来保存您想要访存的列。

> 您不可在 FETCH 语句中使用主变量或系统描述符区域来为列保存访存数组。您必须使用 sqlda 结构和 FETCH...USING DESCRIPTOR 语句。

使用 DESCRIBE...INTO 语句来初始化 sqlda 结构，并获得关于准备好的查询的信息。

DESCRIBE...INTO statement 语句为 **sqlda** 结构和 **sqlvar_struct** 结构分配内存。

对于 sqldata 字段，为每一列分配大到足以保存访存数组的缓冲区。

> 要为 sqldata 字段分配内存，您必须为相关的列将 FetArrSize 全局变量设置为访存设置的大小。

发出 FETCH...USING DESCRIPTOR 语句来将该列值检索至访存设置内。

FETCH 语句将检索到的行放置至 **sqlda** 中的 **sqlvar_struct** 结构的 **sqldata** 字段内。每一 FETCH 语句将由 **FetArrSize** 指定的值的数目返回至 **sqldata** 字段内。

从每一 sqlvar_struct 结构的访存数组获得列值。

在您执行下一 FETCH 语句之前，您必须从访存数组获得这些值。您可检查 sqlca.sqlerrd[2] 字段来确定 FETCH 已返回的有效行的数目。sqlerrd[2] 中的值等于或小于您在 FetArrSize 中设置的值。

重复步骤 4和 5，直到访存所有行为止。

释放 sqlda 结构使用的内存。

正如 sqlda 结构的其他使用一样，GBase 8s ESQL/C 不为此结构释放资源。当您的应用程序不再需要分配给 **sqlda** 结构的内存时，它必须释放它。

重要：当启用“延迟的 PREPARE” 和 OPTOFC 特性时，**FetArrSize** 特性不起作用。当启用这两个特性时，GBase 8s ESQL/C 不知道行的大小，直到 FETCH 语句完成之后为止。到此时，对于要以 **FetArrSize** 值来调整访存缓冲区而言，为时已晚。

样例访存数组程序

下列样例程序展示如何执行 使用访存数组中的步骤。它使用分开的函数来初始化、打印和释放 sqlda 结构。在下面的部分中描述这些函数。

```
#include <windows.h>
#include
#include

EXEC SQL include sqlda.h;
EXEC SQL include locator.h;
EXEC SQL include sqltypes.h;


#define BLOBSIZE 32275      /* using a predetermined length for blob */

EXEC SQL begin declare section;
    long blobsize;     /* finding the maximum blob size at runtime */
EXEC SQL end declare section;


/*****************************************************************
```

```
**
* Function: init_sqlda()
* Purpose: With the sqlda pointer that was returned from the DESCRIBE
* statement, function allocates memory for the fetch arrays
* in the sqldata fields of each column. The function uses
* FetArrSize to determine the size to allocate.
* Returns: < 0 for error
* > 0 error with messagesize
**********************************************************************
*/
int init_sqlda(struct sqlda *in_da, int print)
{
    int i, j,
   row_size=0,
   msglen=0,
   num_to_alloc;
    struct sqlvar_struct *col_ptr;
    ifx_loc_t *temp_loc;
    char *type;


    if (print)
   printf("columns: %d. \n", in_da->sqld);

    /* Step 1: determine row size */
    for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++, col_ptr++)
    {
   /* The msglen variable holds the sum of the column sizes in the
    * database; these are the sizes that DESCRIBE returns. This
    * sum is the amount of memory that ESQL/C needs to store
    * one row from the database. This value is <= row_size. */
   msglen += col_ptr->sqllen; /* get database sizes */


   /* calculate size for C data: string columns get extra byte added
    * to hold null terminator */
   col_ptr->sqllen = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);

   /* The row_size variable holds the sum of the column sizes in
        * the client application; these are the sizes that rtypmsize()
        * returns. This sum is amount of memory that the client
        * application needs to store one row. */
   row_size += col_ptr->sqllen;
   if(print)
```

```
    printf("Column %d size:  %d\n", i+1, col_ptr->sqllen);
 }

 if (print)
 {
printf("Total message size = %d\n", msglen);
printf("Total row size = %d\n", row_size);
 }

 EXEC SQL select max(length(cat_descr)) into :blobsize from catalog;

 /* Step 2: set FetArrSize global variable to number of elements
  * in fetch array; this function calculates the FetArrSize
  * value that can fit into the existing fetch buffer.
  * If FetBufSize is not set (equals zero), the code assigns a
  * default size of 4096 bytes (4 kilobytes). Alternatively, you
  * could set FetArrSize to the number elements you wanted to
  * have and let ESQL/C size the fetch buffer. See the text in
  * "Allocating Memory for the Fetch Arrays" for more information.*/
 if (FetArrSize <= 0) /* if FetArrSize not yet initialized */
 {
if (FetBufSize == 0) /* if FetBufSize not set */
   FetBufSize = 4096; /* default FetBufSize */
FetArrSize = FetBufSize/msglen;
 }
 num_to_alloc = (FetArrSize == 0)? 1: FetArrSize;
 if (print)
 {
printf("Fetch Buffer Size %d\n", FetBufSize);
printf("Fetch Array Size:  %d\n", FetArrSize);
 }

 /* set type in sqlvar_struct structure to corresponding C type */
 for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
 col_ptr++)
 {
switch(col_ptr->sqltype)
    {
   case SQLCHAR:
       type = "char ";
       col_ptr->sqltype = CCHARTYPE;
       break;
   case SQLINT:
    case SQLSERIAL:
```

```
            type = "int ";
            col_ptr->sqltype = CINTTYPE;
            break;
        case SQLBYTES:
        case SQLTEXT:
            if (col_ptr->sqltype == SQLBYTES)
           type = "blob ";
            else
           type = "text ";
            col_ptr->sqltype = CLOCATORTYPE;

            /* Step 3 (TEXT & BLOB only): allocate memory for sqldata
             * that contains ifx_loc_t structures for TEXT or BYTE column
*/
            temp_loc = (ifx_loc_t *)malloc(col_ptr->sqllen *
num_to_alloc);
            if (!temp_loc)
            {
           fprintf(stderr, "blob sqldata malloc failed\n");
           return(-1);
            }
            col_ptr->sqldata = (char *)temp_loc;

            /* Step 4 (TEXT & BLOB only): initialize ifx_loc_t structures
to
            hold blob values in a user-defined buffer in memory */
            byfill( (char *)temp_loc, col_ptr->sqllen*num_to_alloc ,0);
            for (j = 0; j< num_to_alloc; j++, temp_loc++)
            {
           /* blob data to go in memory */
           temp_loc->loc_loctype = LOCMEMORY;

               /* assume none of the blobs are larger than BLOBSIZE */
           temp_loc->loc_bufsize = blobsize;
           temp_loc->loc_buffer = (char *)malloc(blobsize+1);
           if (!temp_loc->loc_buffer)
           {
              fprintf(stderr, "loc_buffer malloc failed\n");
              return(-1);
               }
               temp_loc->loc_oflags = 0; /* clear flag */
            } /* end for */
            break;
        default: /* all other data types */
```

```
            fprintf(stderr, "not yet handled(%d)!\n", col_ptr->sqltype);
            return(-1);
        } /* switch */

    /* Step 5: allocate memory for the indicator variable */
    col_ptr->sqlind = (short *)malloc(sizeof(short) * num_to_alloc);
    if (!col_ptr->sqlind)
        {
        printf("indicator malloc failed\n");
        return -1;
        }

    /* Step 6 (other data types): allocate memory for sqldata. This
     * function
     * casts the pointer to this memory as a (char *). Subsequent
     * accesses to the data would need to cast it back to the data
     * type that corresponds to the column type. See the print_sqlda()
     * function for an example of this casting. */
    if (col_ptr->sqltype != CLOCATORTYPE)
    {
        col_ptr->sqldata = (char *) malloc(col_ptr->sqllen *
num_to_alloc);
        if (!col_ptr->sqldata)
        {
            printf("sqldata malloc failed\n");
            return -1;
        }
        if (print)
            printf("column %3d, type = %s(%3d), len=%d\n", i+1, type,
                  col_ptr->sqltype, col_ptr->sqllen);
        }
    }  /* end for */
    return msglen;
}

/*****************************************************************
***
* Function: print_sqlda
* Purpose: Prints contents of fetch arrays for each column that the
* sqlda structure contains. Current version only implements
* data types found in the blobtab table. Other data types
* would need to me implemented to make this function complete.
*****************************************************************
**/
```

```
void print_sqlda(struct sqlda *sqlda, int count)
{
    void *data;
    int i, j;
   ifx_loc_t *temp_loc;
    struct sqlvar_struct *col_ptr;
    char *type;
    char buffer[512];
    int ind;
    char i1, i2;

    /* print number of columns (sqld) and number of fetch-array elements
    */
    printf("\nsqld: %d, fetch-array elements: %d.\n", sqlda->sqld,
    count);

    /* Outer loop: loop through each element of a fetch array */
    for (j = 0; j < count; j ++)
    {
   if (count > 1)
       {
      printf("record[%4d]:\n", j);
      printf("col | type | id | len | ind | rin | data ");
      printf("| value\n");
      printf("-----------------------------------------");
      printf("-----------------\n");
       }

   /* Inner loop: loop through each of the sqlvar_struct structures */
   for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++, col_ptr++)
       {
      data = col_ptr->sqldata + (j*col_ptr->sqllen);
      switch (col_ptr->sqltype)
      {
          case CFIXCHARTYPE:
          case CCHARTYPE:
         type = "char";
         if (col_ptr->sqllen > 40)
            sprintf(buffer, " %39.39s<..", data);
         else
            sprintf(buffer, "%*.*s", col_ptr->sqllen,
                   col_ptr->sqllen, data);
         break;
          case CINTTYPE:
```

```
         type = "int";
         sprintf(buffer, " %d", *(int *) data);
         break;
          case CLOCATORTYPE:
          type = "byte";
         temp_loc = (ifx_loc_t *)(col_ptr->sqldata +
                   (j * sizeof(ifx_loc_t)));
         sprintf(buffer, " buf ptr: %p, buf sz: %d, blob sz: %d",
temp_loc->loc_buffer,
                temp_loc->loc_bufsize, temp_loc->loc_size);
         break;
          default:
         type = "??????";
         sprintf(buffer, " type not implemented: ",
                "can't print %d", col_ptr->sqltype);
         break;
      } /* end switch */

      i1 = (col_ptr->sqlind==NULL) ? 'X' :
          (((col_ptr->sqlind)[j] != 0) ? 'T' : 'F');
      i2 = (risnull(col_ptr->sqltype, data)) ? 'T' : 'F';

       printf("%3d | %-6.6s | %3d | %3d | %c | %c | ",
          i, type, col_ptr->sqltype, col_ptr->sqllen, i1, i2);
      printf("%8p |%s\n", data, buffer);
       } /* end for (i=0...) */
    } /* end for (j=0...) */
}

/*****************************************************************************
***
* Function: free_sqlda
* Purpose: Frees memory used by sqlda. This memory includes:
* o loc_buffer memory (used by TEXT & BYTE)
* o sqldata memory
* o sqlda structure
*****************************************************************************
**/
void free_sqlda(struct sqlda *sqlda)
{
    int i,j, num_to_dealloc;
    struct sqlvar_struct *col_ptr;
    ifx_loc_t *temp_loc;
```

```
    for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++,
    col_ptr++)
    {
   if ( col_ptr->sqltype == CLOCATORTYPE )
       {
      /* Free memory for blob buffer of each element in fetch array */
      num_to_dealloc = (FetArrSize == 0)? 1: FetArrSize;
      temp_loc = (ifx_loc_t *) col_ptr->sqldata;
      for (j = 0; j< num_to_dealloc; j++, temp_loc++)
       {
          free(temp_loc->loc_buffer);
       }
       }
   /* Free memory for sqldata (contains fetch array) */
   free(col_ptr->sqldata);
    }

    /* Free memory for sqlda structure */
    free(sqlda);
}

void main()
{
    int i = 0;
    int row_count, row_size;


    EXEC SQL begin declare section;
   char *db   = "stores7";
   char *uid  = "odbc";
   char *pwd  = "odbc";
    EXEC SQL end declare section;


/************************************************************************
*
     * Step 1: declare an sqlda structure to hold the retrieved column
     * values

************************************************************************
/
    struct sqlda *da_ptr;

    EXEC SQL connect to :db user :uid using :pwd;
```

```
    if ( SQLCODE < 0 )
    {
   printf("CONNECT failed: %d\n", SQLCODE);
   exit(0);
    }

    /* Prepare the SELECT */
    EXEC SQL prepare selct_id from 'select catalog_num, cat_descr from
    catalog';
    if ( SQLCODE < 0 )
    {
   printf("prepare failed: %d\n", SQLCODE);
   exit(0);
    }


/************************************************************************
*
     * Step 2: describe the prepared SELECT statement to allocate memory
     * for the sqlda structure and the sqlda.sqlvar structures
     * (DESCRIBE can allocate sqlda.sqlvar structures because
     * prepared statement is a SELECT)

************************************************************************
/
    EXEC SQL describe selct_id into da_ptr;
    if ( SQLCODE < 0 )
    {
   printf("describe failed: %d\n", SQLCODE);
   exit(0);
    }


/************************************************************************
*
     * Step 3: initialize the sqlda structure to hold fetch arrays for
     * columns

************************************************************************
/
    row_size = init_sqlda(da_ptr, 1);

    /* declare and open a cursor for the prepared SELECT */
    EXEC SQL declare curs cursor for selct_id;
```

```
    if ( SQLCODE < 0 )
    {
   printf("declare failed: %d\n", SQLCODE);
   exit(0);
    }
    EXEC SQL open curs;
    if ( SQLCODE < 0 )
    {
   printf("open failed: %d\n", SQLCODE);
   exit(0);
    }
    while (1)
    {

/*********************************************************************
*
    * Step 4: perform fetch to get "FetArrSize" array of rows from
    * the database server into the sqlda structure

*********************************************************************
/
   EXEC SQL fetch curs using descriptor da_ptr;

   /* Reached last set of matching rows? */
   if ( SQLCODE == SQLNOTFOUND )
      break;


/*********************************************************************
*
    * Step 5: obtain the values from the fetch arrays of the sqlda
    * structure; use sqlca.sqlerrd[2] to determine number
    * of array elements actually retrieved.

*********************************************************************
/
   printf("\n===============\n");
   printf("FETCH %d\n", i++);
   printf("===============");
   print_sqlda(da_ptr, ((FetArrSize == 0) ? 1 : sqlca.sqlerrd[2]));


/*********************************************************************
*
```

```
    * Step 6: repeat the FETCH until all rows have been fetched (SQLCODE
    * is SQLNOTFOUND

**********************************************************************
/
    }


/**********************************************************************
*
     * Step 7: Free resources:
     * o statement id, selct_id
     * o select cursor, curs
     * o sqlda structure (with free_sqlda() function)
     * o delete sample table and its rows from database

**********************************************************************
/

    EXEC SQL free selct_id;
    EXEC SQL close curs;
    EXEC SQL free curs;
    free_sqlda(da_ptr);
}
```

为访存数组分配内存

DESCRIBE...INTO 语句为 **sqlda** 结构及其 **sqlvar_struct** 结构分配内存。然而，它不为 **sqlvar_struct** 结构的 **sqldata** 字段分配内存。**sqldata** 字段为检索了的列保存访存数组。因此，您必须将足够的内存分配给每一 **sqldata** 字段来保存该访存数组的元素。

新的全局变量 **FetArrSize** 指示由 FETCH 语句返回的行数。定义此变量为 C 语言 short integer 数据类型。它的缺省值为零，其禁用访存数组特性。您可将 **FetArrSize** 设置为在下列范围内的任何整数值：

0 <= FetArrSize <= MAXSMINT

MAXSMINT 值是 GBase 8s ESQL/C 可检索的数据类型的最大数量。它的值为 32767 字节（32 KB）。如果该访存数组的大小大于 MAXSMINT，则 GBase 8s ESQL/C 自动地将它的大小减为 32 KB。

您可使用下列计算来确定该访存数组的适当的大小：

(fetch-array size) = (fetch-buffer size) / (row size)

前面的等式使用下列信息：

fetch-array size

访存数组的大小，由 **FetArrSize** 全局变量来指示

fetch-buffer size

访存缓冲区的大小，由 FetBufSize 和 BigFetBufSize 全局变量来指示。

row size

要访存的行的大小。要确定要访存的行的大小，对于该行的每一列，请调用 rtypmsize() 函数。此函数返回需要存储该数据类型的字节数。

然而，如果您设置 **FetArrSize** 以便于下列关系为真，

(FetArrSize * row size) > FetBufSize

则 GBase 8s ESQL/C 自动地调整该访存缓冲区的大小（**FetBufSize**）如下，来保存该访存数组的大小：

FetBufSize = FetArrSize * row size

如果该值大于 32 KB（MAXSMINT），则 GBase 8s ESQL/C 将 **FetBufSize** 设置为 32 KB 和 **FetArrSize** 如下：

FetArrSize = MAXSMINT / (row size)

重要： **FetArrSize** 全局变量可用在线程安全 GBase 8s ESQL/C 应用程序中。

为访存数组分配内存

要为访存数组分配内存：

确定您正从数据库检索的行的大小。

确定该访存数组的大小，并将 **FetArrSize** 全局变量设置为此值。

对于每一简单大对象列（TEXT 或 BYTE），分配一个 **ifx_loc_t** 结构的访存数组。

对于每一简单大对象列（TEXT 或 BYTE），初始化 **ifx_loc_t** 数据结构如下。

将 **loc_loctype** 字段设置为 LOCMEMORY。

a) 将 loc_buffer 字段设置为您在步骤 3中分配的缓冲区的地址。

将 **loc_bufsize** 字段设置为您分配了的缓冲区的大小。

或者，您可将 loc_bufsize 设置为 -1，来让 GBase 8s ESQL/C 自动地为简单大对象列分配内存。

为指示符变量分配内存。

对于所有其他列，分配保存那列的数据类型的访存数组。

对于下列准备好的查询，下列示例代码说明您会如何为访存数组分配内存：

SELECT * from blobtab;

该函数称为 init_sqlda()：

```
/************************************************************************
        * Function: init_sqlda()
        * Purpose: With the sqlda pointer that was returned from the DESCRIBE
        * statement, function allocates memory for the fetch arrays
        * in the sqldata fields of each column. The function uses
        * FetArrSize to determine the size to allocate.
        * Returns: < 0 for error
        * > 0 error with messagesize

************************************************************************/
        int init_sqlda(struct sqlda *in_da, int print)
        {
        int i, j,
        row_size=0,
        msglen=0,
        num_to_alloc;
        struct sqlvar_struct *col_ptr;
        ifx_loc_t *temp_loc;
        char *type;


        if (print)
        printf("columns: %d. \n", in_da->sqld);
```

```
/* Step 1: determine row size */
for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
col_ptr++)
{
/* The msglen variable holds the sum of the column sizes in the
* database; these are the sizes that DESCRIBE returns. This
* sum is the amount of memory that ESQL/C needs to store
* one row from the database. This value is <= row_size. */
msglen += col_ptr->sqllen; /* get database sizes */


/* calculate size for C data: string columns get extra byte added
* to hold null terminator */
col_ptr->sqllen = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);

/* The row_size variable holds the sum of the column sizes in
* the client application; these are the sizes that rtypmsize()
* returns. This sum is amount of memory that the client
* application needs to store one row. */
row_size += col_ptr->sqllen;
if(print)
printf("Column %d size:    %d\n", i+1, col_ptr->sqllen);
}

if (print)
{
printf("Total message size = %d\n", msglen);
printf("Total row size = %d\n", row_size);
}

EXEC SQL select max(length(cat_descr)) into :blobsize from catalog;

/* Step 2: set FetArrSize global variable to number of elements
```

```
* in fetch array; this function calculates the FetArrSize
* value that can fit into the existing fetch buffer.
* If FetBufSize is not set (equals zero), the code assigns a
* default size of 4096 bytes (4 kilobytes). Alternatively, you
* could set FetArrSize to the number elements you wanted to
* have and let ESQL/C size the fetch buffer. See the text in
* "Allocating Memory for the Fetch Arrays" for more information.*/
if (FetArrSize <= 0) /* if FetArrSize not yet initialized */
{
if (FetBufSize == 0) /* if FetBufSize not set */
FetBufSize = 4096; /* default FetBufSize */
FetArrSize = FetBufSize/msglen;
}
num_to_alloc = (FetArrSize == 0)? 1: FetArrSize;
if (print)
{
printf("Fetch Buffer Size %d\n", FetBufSize);
printf("Fetch Array Size:    %d\n", FetArrSize);
}

/* set type in sqlvar_struct structure to corresponding C type */
for (i = 0, col_ptr = in_da->sqlvar; i < in_da->sqld; i++,
col_ptr++)
{
switch(col_ptr->sqltype)
{
case SQLCHAR:
type = "char ";
col_ptr->sqltype = CCHARTYPE;
break;
case SQLINT:
case SQLSERIAL:
type = "int ";
```

```
col_ptr->sqltype = CINTTYPE;
break;
case SQLBYTES:
case SQLTEXT:
if (col_ptr->sqltype == SQLBYTES)
type = "blob ";
else
type = "text ";
col_ptr->sqltype = CLOCATORTYPE;

/* Step 3 (TEXT & BLOB only): allocate memory for sqldata
* that contains ifx_loc_t structures for TEXT or BYTE column */
temp_loc = (ifx_loc_t *)malloc(col_ptr->sqllen * num_to_alloc);
if (!temp_loc)
{
fprintf(stderr, "blob sqldata malloc failed\n");
return(-1);
}
col_ptr->sqldata = (char *)temp_loc;

/* Step 4 (TEXT & BLOB only): initialize ifx_loc_t structures to
hold blob values in a user-defined buffer in memory */
byfill( (char *)temp_loc, col_ptr->sqllen*num_to_alloc ,0);
for (j = 0; j< num_to_alloc; j++, temp_loc++)
{
/* blob data to go in memory */
temp_loc->loc_loctype = LOCMEMORY;

/* assume none of the blobs are larger than BLOBSIZE */
temp_loc->loc_bufsize = blobsize;
temp_loc->loc_buffer = (char *)malloc(blobsize+1);
if (!temp_loc->loc_buffer)
{
```

```
fprintf(stderr, "loc_buffer malloc failed\n");
return(-1);
}
temp_loc->loc_oflags = 0; /* clear flag */
} /* end for */
break;
default: /* all other data types */
fprintf(stderr, "not yet handled(%d)!\n", col_ptr->sqltype);
return(-1);
} /* switch */

/* Step 5: allocate memory for the indicator variable */
col_ptr->sqlind = (short *)malloc(sizeof(short) * num_to_alloc);
if (!col_ptr->sqlind)
{
printf("indicator malloc failed\n");
return -1;
}

/* Step 6 (other data types): allocate memory for sqldata. This function
* casts the pointer to this memory as a (char *). Subsequent
* accesses to the data would need to cast it back to the data
* type that corresponds to the column type. See the print_sqlda()
* function for an example of this casting. */
if (col_ptr->sqltype != CLOCATORTYPE)
{
col_ptr->sqldata = (char *) malloc(col_ptr->sqllen *
num_to_alloc);
if (!col_ptr->sqldata)
{
printf("sqldata malloc failed\n");
return -1;
}
```

```
if (print)
printf("column %3d, type = %s(%3d), len=%d\n", i+1, type,
col_ptr->sqltype, col_ptr->sqllen);
}
}   /* end for */
return msglen;
}
```

从访存数组获得值

每一 FETCH 尝试将值的 **FetArrSize** 数目返回至 **sqlda** 结构的 **sqlvar_struct** 结构的 **sqldata** 字段内。您可检查 **sqlca.sqlerrd[2]** 值来确定 FETCH 确实返回了的实际行数。

对于该查询的一列，每一访存数组保存该值。要获得值的行，您必须访问每一访存数组的同一索引处的元素。例如，要获得值的第一行，请访问每一访存数组的第一个元素。

样例程序调用 print_sqlda() 函数来为下列准备好的查询从访存数组获得值：

SELECT * from blobtab

```
/************************************************************************
* Function: print_sqlda
* Purpose: Prints contents of fetch arrays for each column that the
* sqlda structure contains. Current version only implements
* data types found in the blobtab table. Other data types
* would need to me implemented to make this function complete.
************************************************************************/
void print_sqlda(struct sqlda *sqlda, int count)
{
void *data;
int i, j;
ifx_loc_t *temp_loc;
struct sqlvar_struct *col_ptr;
char *type;
```

```
char buffer[512];
int ind;
char i1, i2;

/* print number of columns (sqld) and number of fetch-array elements
*/
printf("\nsqld: %d, fetch-array elements: %d.\n", sqlda->sqld,
count);

/* Outer loop: loop through each element of a fetch array */
for (j = 0; j < count; j ++)
{
if (count > 1)
{
printf("record[%4d]:\n", j);
printf("col | type | id | len | ind | rin | data ");
printf("| value\n");
printf("-------------------------------------------");
printf("------------------\n");
}

/* Inner loop: loop through each of the sqlvar_struct structures */
for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++, col_ptr++)
{
data = col_ptr->sqldata + (j*col_ptr->sqllen);
switch (col_ptr->sqltype)
{
case CFIXCHARTYPE:
case CCHARTYPE:
type = "char";
if (col_ptr->sqllen > 40)
sprintf(buffer, " %39.39s<..", data);
else
```

```
sprintf(buffer, "%*.*s", col_ptr->sqllen,
col_ptr->sqllen, data);
break;
case CINTTYPE:
type = "int";
sprintf(buffer, " %d", *(int *) data);
break;
case CLOCATORTYPE:
type = "byte";
temp_loc = (ifx_loc_t *)(col_ptr->sqldata +
(j * sizeof(ifx_loc_t)));
sprintf(buffer, " buf ptr: %p, buf sz: %d, blob sz: %d",
temp_loc->loc_buffer,
temp_loc->loc_bufsize, temp_loc->loc_size);
break;
default:
type = "??????";
sprintf(buffer, " type not implemented: ",
"can't print %d", col_ptr->sqltype);
break;
} /* end switch */

i1 = (col_ptr->sqlind==NULL) ? 'X' :
(((col_ptr->sqlind)[j] != 0) ? 'T' : 'F');
i2 = (risnull(col_ptr->sqltype, data)) ? 'T' : 'F';

printf("%3d | %-6.6s | %3d | %3d | %c | %c | ",
i, type, col_ptr->sqltype, col_ptr->sqllen, i1, i2);
printf("%8p |%s\n", data, buffer);
} /* end for (i=0...) */
} /* end for (j=0...) */
}
```

为访存数组释放内存

GBase 8s ESQL/C 不为 **sqlda** 结构释放资源。当您的应用程序不再需要 **sqlda** 结构时，它必须释放它使用的所有内存。

该样例程序调用 **free_sqlda()** 函数来释放 **sqlda** 结构使用的内存。

```
/************************************************************************
        * Function: free_sqlda
        * Purpose: Frees memory used by sqlda. This memory includes:
        * o loc_buffer memory (used by TEXT & BYTE)
        * o sqldata memory
        * o sqlda structure
************************************************************************/
        void free_sqlda(struct sqlda *sqlda)
        {
        int i,j, num_to_dealloc;
        struct sqlvar_struct *col_ptr;
        ifx_loc_t *temp_loc;

        for (i = 0, col_ptr = sqlda->sqlvar; i < sqlda->sqld; i++,
        col_ptr++)
        {
        if ( col_ptr->sqltype == CLOCATORTYPE )
        {
        /* Free memory for blob buffer of each element in fetch array */
        num_to_dealloc = (FetArrSize == 0)? 1: FetArrSize;
        temp_loc = (ifx_loc_t *) col_ptr->sqldata;
        for (j = 0; j< num_to_dealloc; j++, temp_loc++)
        {
        free(temp_loc->loc_buffer);
        }
        }
        /* Free memory for sqldata (contains fetch array) */
        free(col_ptr->sqldata);
```

```
}

/* Free memory for sqlda structure */
free(sqlda);
}
```

4.3 系统描述符区域

系统描述符区域是您可保存的准备好的语句从数据库服务器返回的或发送至数据库服务器的数据的动态管理结构。系统描述符区域符合 X/Open 标准。

这些主题包含下列关于如何使用系统描述符区域的信息：

管理动态 SQL 的系统描述符区域

使用动态描述符区域来处理动态 SQL 语句中的未知值

此部分的结尾呈现一称为 **dyn_sql** 的注释的示例程序，其使用系统描述符区域来处理在运行时刻输入的 SELECT 语句。

4.3.1 管理系统描述符区域

您的 GBase 8s ESQL/C 程序可操纵带有下表总结的 SQL 语句的系统描述符区域。

表 16. 可用于操纵系统描述符区域的 SQL 语句

SQL 语句	用途	请参阅
ALLOCATE DESCRIPTOR	为系统描述符区域分配内存	为系统描述符区域分配内存
DESCRIBE...USING SQL DESCRIPTOR	以关于列列表列的信息初始化系统描述符区域	初始化系统描述符区域
GET DESCRIPTOR	从系统描述符区域的字段获得信息	指定与从系统描述符区域获得值
SET DESCRIPTOR	对于要访问的数据库服务器，将信息放至系统描述符区域内	指定与从系统描述符区域获得值

表 17. 可用于操纵系统描述符区域的 SQL 语句：使用游标的 SELECT 和 EXECUTE FUNCTION 语句

SQL 语句	用途	请参阅
OPEN...USING SQL DESCRIPTOR FETCH...USING SQL DESCRIPTOR	从指定的系统描述符区域取任何输入参数 将该行的内容放至系统描述符区域内	指定输入参数值 将列值放至系统描述符区域内

表 18. 可用于操纵系统描述符区域的 SQL 语句：仅返回一行的 SELECT 和 EXECUTE FUNCTION 语句

SQL 语句	用途	请参阅
EXECUTE...INTO SQL DESCRIPTOR	将单个行的内容放至系统描述符区域内	将列值放至系统描述符区域内

表 19. 可用于操纵系统描述符区域的 SQL 语句：非 SELECT 语句：

SQL 语句	用途	请参阅
EXECUTE...USING SQL DESCRIPTOR	从指定的系统描述符区域取任何输入参数	指定输入参数值

表 20. 可用于操纵系统描述符区域的 SQL 语句：使用插入游标的 INSERT 语句：

SQL 语句	用途	请参阅
PUT...USING SQL DESCRIPTOR	将一行放至插入缓冲区内，从指定的系统描述符区域获得列值	处理未知的列列表
DEALLOCATE DESCRIPTOR	当以其结束您的程序时，释放为系统描述符区域分配的内存	释放分配给系统描述符区域的内存

为系统描述符区域分配内存

要为系统描述符区域分配内存，请使用 ALLOCATE DESCRIPTOR 语句。

ALLOCATE DESCRIPTOR 语句执行下列任务：

它指定指定的描述符名称来标识此内存区域。在罗列在 表 1中的所有 SQL 语句中，此名称是必须提供的标识符，来指定对其采取行动的系统描述符。

它分配项描述符。在缺省情况下，它在系统描述符区域中分配 100 个项描述符。您可以 WITH MAX 子句来更改此缺省值。

它将系统描述符区域中的 COUNT 字段初始化为分配的项描述符的数目。

重要： ALLOCATE DESCRIPTOR 不为列数据（DATA 字段）分配内存。由 DESCRIBE 语句根据需要来分配内存。

初始化系统描述符区域

DESCRIBE...USING SQL DESCRIPTOR 语句以关于准备好的语句的信息来初始化系统描述符区域。

DESCRIBE...USING SQL DESCRIPTOR 语句采取下列行动：

它设置 COUNT 字段，其包含以数据初始化的项描述符的数目。

此值是列列表（SELECT 和 INSERT）中的列和表达式的数目，或返回的值（EXECUTE FUNCTION）的数目。

它描述（不带有 INTO TEMP）的准备好的 SELECT 语句、EXECUTE FUNCTION 或 INSERT 语句中每一未知的列。

DESCRIBE 语句为每一列初始化项描述符的字段，如下：

它基于 TYPE 和 LENGTH 信息为 DATA 字段分配内存。

它初始化 TYPE、LENGTH、NAME、SCALE、PRECISION 和 NULLABLE 字段来从数据库提供关于列的信息。

它返回准备好的 SQL 语句的类型。

正如前面提到的，DESCRIBE 语句提供关于列列表的列的信息。因此，您通常在准备好了的（不带有 INTO TEMP 子句的）SELECT、INSERT 或 EXECUTE FUNCTION 语句之后使用此语句。

DESCRIBE 语句和输入参数

当您使用系统描述符区域来保存输入参数时，您不可使用 DESCRIBE 来初始化该系统描述符区域。您的代码必须以 SET DESCRIPTOR 语句来定义输入函数，来显式地设置系统描述符区域的恰当的字段。

DESCRIBE 语句与内存分配

当您使用系统描述符区域来保存准备好的 SQL 语句的列时，ALLOCATE

DESCRIPTOR 语句为每一列的项描述符分配内存，且 DESCRIBE...USING SQL DESCRIPTOR 语句为每一项描述符的 DATA 字段分配内存。

然而，当您描述一准备好的 SELECT 语句，而该语句将数据从一列访存至 **lvarchar** 类型的主变量内时，DESCRIBE...USING SQL DESCRIPTOR 语句不为系统描述符区域的 DATA 字段分配内存。

在您将 **lvarchar** 数据访存至系统描述符区域之前，您必须显式地将内存指定到 DATA 字段来保存该列值，如下：

声明一大小恰当的 **lvarchar** 主变量。

请确保此变量不只是一个指针，而有与它相关联的内存。

以 SET DESCRIPTOR 语句将此主变量指定给 DATA 字段。

此 SET DESCRIPTOR 语句发生在 DESCRIBE...USING SQL DESCRIPTOR 语句之后，但发生在 FETCH...USING SQL DESCRIPTOR 语句之前。

执行 FETCH...USING SQL DESCRIPTOR 语句来将列数据检索至系统描述符区域的 DATA 字段内。

下列代码段展示为 **table1** 表中的名为 **lvarch_col** 的 LVARCHAR 列分配内存的基本步骤：

```
EXEC SQL BEGIN DECLARE SECTION;
        lvarchar lvarch_val[50];
        int i;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL allocate descriptor 'desc';
        EXEC SQL prepare stmt1 from 'select opaque_col from table1';
        EXEC SQL describe stmt1 using sql descriptor 'desc';
        EXEC SQL declare cursor curs1 for stmt1;
        EXEC SQL open curs1;
        EXEC SQL set descriptor 'desc' value 1
        data = :lvarch_val, length = 50;
```

```
while (1)
{
EXEC SQL fetch curs1 using sql descriptor 'desc';
EXEC SQL get descriptor 'desc' value 1 :lvarch_val;
printf("Column value is %s\n", lvarch_val);

⋮

}
```

前面的代码段不执行异常处理。

指定与从系统描述符区域获得值

下列 SQL 语句允许您的程序访问系统描述符区域的字段：

SET DESCRIPTOR 语句将值指定给系统描述符区域的字段。

GET DESCRIPTOR 语句从系统描述符区域的字段获得值。

SET DESCRIPTOR 语句

要将值指定给系统描述符区域字段，请使用 SET DESCRIPTOR 语句。

您可使用 SET DESCRIPTOR 语句来：

设置 COUNT 字段，来匹配您在系统描述符区域中提供其描述的项的数目。此值通常为 WHERE 子句中输入参数的数目。

EXEC SQL set descriptor *sysdesc* COUNT=:*hostvar*;

为您为其提供描述的每一列值设置项描述符字段。

EXEC SQL set descriptor sysdesc VALUE :item_numDESCRIP_FIELD=:hostvar;

在此示例中，*item_num* 是对应于期望的列的项描述符的数目，*DESCRIP_FIELD* 是罗列在 表 1中的项描述符字段之一。

设置字段值来为 WHERE 子句中的输入参数提供值（指定输入参数值）或在您使用 DESCRIBE...USING SQL DESCRIPTOR 语句来填充系统描述符区域之后，来修改项描述符字段的内容（将列值放至系统描述符区域内）。

数据库服务器在 sqltypes.h 头文件中提供数据类型常量，来标识系统描述符区域的 TYPE 字段（及可选的 ITYPE 字段）中列的数据类型。然而，您不可在 SET DESCRIPTOR 语句中直接地指定数据类型常量。相反，请将该常量值指定给整数主变量，并在 SET DESCRIPTOR 语句中指定此变量，如下：

```
EXEC SQL BEGIN DECLARE SECTION;
        int i;
        ⋮
        EXEC SQL END DECLARE SECTION;
        ⋮
        i = SQLINT;
        EXEC SQL set descriptor 'desc1' VALUE 1
        TYPE = :i;
```

带有描述符的 lvarchar 指针主变量

如果您随同使用系统描述符区域的 FETCH 或 PUT 语句使用 **lvarchar** 指针主变量，则您必须显式地在 SET DESCRIPTOR 语句中将类型设置为 124（CLVCHARPTRTYPE 来自 incl/esql/sqltypes.h）。下列示例说明：

```
EXEC SQL BEGIN DECLARE SECTION;
        lvarchar *lv;
        EXEC SQL END DECLARE SECTION;
        /* where tab has lvarchar * column */
        EXEC SQL prepare stmt from "select col from tab";
        EXEC SQL allocate descriptor 'd';
        /* The following describe will return SQLLVARCHAR for the
        type of the column */
        EXEC SQL describe stmt using sql descriptor 'd';
        /* You must set type for *lv variable */
        EXEC SQL set descriptor 'd' value 1 DATA = :lv, TYPE = 124;
```

```
EXEC SQL declare c cursor for stmt;
EXEC SQL open c;
EXEC SQL fetch c using sql descriptor 'd';
```

GET DESCRIPTOR 语句

GET DESCRIPTOR 语句从系统描述符区域字段获得值。

您可使用 GET DESCRIPTOR 语句来：

获得 COUNT 字段来确定在系统描述符区域中描述了多少个值。

EXEC SQL get descriptor *sysdesc* :*hostvar*=COUNT;

获得每一被描述的列的项描述符字段。

EXEC SQL get descriptor *sysdesc* VALUE :*item_num*

:hostvar=DESCRIP_FIELD;

在此示例中，*item_num* 是对应于期望的列的项描述符的数目，*DESCRIP_FIELD* 是罗列在 表 1中的项描述符字段之一。

这些项描述符值通常是 SELECT、INSERT 或 EXECUTE FUNCTION 语句中列的描述。还在 FETCH...USING SQL DESCRIPTOR 之后使用 GET DESCRIPTOR，来将由数据库服务器返回的列值从系统描述符区域复制至主变量内(将列值放至系统描述符区域内）。

主变量的数据类型必须与相关联的系统描述符区域字段的类型相兼容。当您在 TYPE 字段中说明时，请确保您使用的数据类型值与您的环境相匹配。对于一些数据类型，X/Open 值不同于 GBase 8s 值。

指定输入参数值

由于 DESCRIBE...USING SQL DESCRIPTOR 语句不分析 WHERE 子句，因此，您的程序必须在系统描述符区域的字段中存储输入参数的数目、数据类型和值，来显式地描述这些参数。

当您执行参数化的语句时，您必须以 USING SQL DESCRIPTOR 子句指定系统描述符区域作为输入参数值的位置，如下：

对于 SELECT 的 WHERE 子句中的输入参数，请使用 OPEN...USING SQL DESCRIPTOR 语句。此语句处理顺序的、滚动的、保存或更新游标。如果您确定该 SELECT 仅返回一行，则您可使用 EXECUTE...INTO...USING SQL DESCRIPTOR 语句，而不使用游标。

对于非 SELECT 语句的 WHERE 子句中的输入参数，诸如 DELETE 或 UPDATE，

请使用 EXECUTE...USING SQL DESCRIPTOR 语句。

对于 INSERT 语句的 VALUES 子句中的输入参数，请使用 EXECUTE...USING SQL DESCRIPTOR 语句。如果该 INSERT 语句与插入游标相关联，则请改为使用 PUT...USING SQL DESCRIPTOR 语句。

将列值放至系统描述符区域内

当您动态地创建 SELECT 语句时，您不可使用 FETCH 的 INTO *host_var* 子句，因为您不可在准备好的语句中命名主变量。要将列值访存至系统描述符区域内，请使用 FETCH 的 USING SQL DESCRIPTOR 子句，而不是 INTO 子句。FETCH...USING SQL DESCRIPTOR 语句将每一列值放至它的项描述符的 DATA 字段内。

FETCH...USING SQL DESCRIPTOR 语句的使用假设存在与准备好的语句相关联的游标。您必须总是使用 SELECT 语句的游标以及游标函数（返回多行的 EXECUTE FUNCTION 语句）。然而，如果该 SELECT（或 EXECUTE FUNCTION）仅返回一行，则您可忽略该游标，而以 EXECUTE...INTO SQL DESCRIPTOR 语句，将列值检索至系统描述符区域内。

重要： 如果您执行返回多行的 SELECT 语句或用户定义的函数，且不将该语句与游标相关联，则您的程序生成运行时刻错误。当您将单个 SELECT（或 EXECUTE FUNCTION）语句与游标相关联，则 GBase 8s ESQL/C 不生成错误。因此，总是将动态 SELECT 或 EXECUTE FUNCTION 语句与游标相关联，且使用 FETCH...USING SQL DESCRIPTOR 语句来将列值从此游标检索至系统描述符区域内，是一个很好的做法。

当列值在系统描述符区域中时，您可使用 GET DESCRIPTOR 语句来将这些值从它们的 DATA 字段转移至恰当的主变量。在运行时刻，您必须使用 LENGTH 和 TYPE 字段来确定这些主变量的数据类型。您可能需要在 TYPE 字段中的 SQL 数据类型与保存该返回值的主变量所需要的 GBase 8s ESQL/C 数据类型之间，执行数据类型或长度转换。

释放分配给系统描述符区域的内存

DEALLOCATE DESCRIPTOR 语句释放指定的系统描述符区域使用的内存。被释放的内存包括由项描述符使用的（在 DATA 字段中）保存数据的内存。请确保仅在您不再需要它之后，才释放系统描述符区域。不可重新使用被释放了的系统描述符区域。

4.3.2 使用系统描述符区域

请使用系统描述符区域来执行包含未知值的 SQL 语句。

下表总结此章节的余下部分涵盖的动态语句的类型。

表 21. 使用系统描述符区域来执行动态 SQL 语句

系统描述符区域的用途	请参阅
保存由 SELECT 语句检索的选择列表列值	处理未知的选择列表
保存从用户定义的函数返回的值	处理未知的返回值
描述在 INSERT 语句中的未知列	处理未知的列列表
描述 SELECT 语句的 WHERE 子句中的输入参数	处理参数化的 SELECT 语句
描述在 SELECT 或 UPDATE 语句的 WHERE 子句中的输入参数	处理参数化的 UPDATE 或 DELETE 语句

4.3.3 处理未知的选择列表

此部分描述如何使用系统描述符区域来处理 SELECT 语句。

要使用系统描述符区域来处理未知的选择列表列：

1. 以 PREPARE 语句准备 SELECT 语句，来给它一个语句标识符。该 SELECT 语句不可包括 INTO TEMP 子句。
2. 以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域。
3. 以 DESCRIBE...USING SQL DESCRIPTOR 语句确定该选择列表列的数目和数据类型。DESCRIBE 为该选择列表中的每一列填充项描述符。

以 GET DESCRIPTOR 语句保存主变量中的选择列表列的数目，来获取 COUNT 字段的值。

4. 声明并打开一游标，然后使用 FETCH...USING SQL DESCRIPTOR 语句来将列值访存至分配了的系统描述符区域内，一次访存一行。
5. 以 GET DESCRIPTOR 语句将行数据从系统描述符区域检索至主变量，来访问 DATA 字段。
6. 以 DEALLOCATE DESCRIPTOR 语句释放系统描述符区域。

重要： 如果 SELECT 语句在 WHERE 子句中有未知的输入参数，则您的程序还必须以系统描述符区域来处理这些输入参数。

执行返回多行的 SELECT

demo4.ec 样例程序展示如何以下列条件执行动态 SELECT 语句：

SELECT 返回多行。

必须将 SELECT 与游标相关联，以 OPEN 语句执行，并以 FETCH...USING SQL DESCRIPTOR 语句检索它的返回值。

SELECT 或没有输入参数，或没有 WHERE 子句。

OPEN 语句不需要包括 USING 子句。

SELECT 在它的选择列表中有未知的列。

FETCH 语句包括 USING SQL DESCRIPTOR 子句来在 **sqlda** 结构中存储返回值。

demo4.ec 样例程序

此 demo4 程序是 demo3 样例程序的一个版本（demo3.ec 样例程序），该程序使用系统描述符区域来保存选择列表列。demo4 程序不包括异常处理。

```
================================================================
======
1. #include <stdio.h>
2. EXEC SQL define NAME_LEN             15;
3. main()
4. {
5. EXEC SQL BEGIN DECLARE SECTION;
6.      mint i;
7.      mint desc_count;
8.      char demoquery[80];
9.      char colname[19];
10.      char result[ NAME_LEN + 1 ];
11. EXEC SQL END DECLARE SECTION;
```

```
================================================================
==
```

5 - 11 行

这些行声明主变量来保存从用户获得的数据，以及从系统描述符区域检索的列值。

```
================================================================
======
```

```
12.     printf("DEMO4 Sample ESQL program running.\n\n");
13.     EXEC SQL connect to 'stores7';
14. /* These next three lines have hard-wired both the query and
15.      * the value for the parameter. This information could have been
16.      * been entered from the terminal and placed into the strings
17.      * demoquery and the query value string (queryvalue),
* respectively.
18.      */
19.     sprintf(demoquery, "%s %s",
20.          "select fname, lname from customer",
21.          "where lname < 'C' ");
22.     EXEC SQL prepare demo4id from :demoquery;
23.     EXEC SQL declare demo4cursor cursor for demo4id;
24.     EXEC SQL allocate descriptor 'demo4desc' with max 4;
25.     EXEC SQL open demo4cursor;
```

==========

14 - 22 行

这些行为（demoquery 中的）语句组装字符串，并准备它作为 demo4id 语句标识符。

23 行

这一行为准备好的语句标识符 **demo4id** 声明 **demo4cursor** 游标。所有非单个的 SELECT 语句都必须有一个声明了的游标。

24 行

要能够为选择列表列使用系统描述符区域，您必须首先分配它。此 ALLOCATE DESCRIPTOR 语句分配带有四个项描述符的 **demo4desc** 系统描述符区域。

25 行

数据库服务器执行 SELECT 语句，当它打开 demo4cursor 游标时。如果您的 SELECT 语句的 WHERE 子句包含输入参数，则您还需要指定 OPEN 语句的 USING SQL DESCRIPTOR 子句。

```
==============================================================================
```

```
26.     EXEC SQL describe demo4id using sql descriptor 'demo4desc';
27.     EXEC SQL get descriptor 'demo4desc' :desc_count = COUNT;
28.     printf("There are %d returned columns:\n", desc_count);
29.     /* Print out what DESCRIBE returns */
30.     for (i = 1; i <= desc_count; i++)
31.         prsysdesc(i);
32.     printf("\n\n");
```

```
==============================================================================
```

26 行

DESCRIBE 语句描述 **demo4id** 语句标识符中准备好的语句的选择列表列。为此，DESCRIBE 必须跟在 PREPARE 之后。此 DESCRIBE 包括 USING SQL DESCRIPTOR 子句来指定 **demo4desc** 系统描述符区域作为这些列描述的位置。

27 和 28 行

27 行使用 GET DESCRIPTOR 语句来获得由 DESCRIBE 发现的选择列表列的数目。从 **demo4desc** 系统描述符区域的 COUNT 字段读取此数目，并保存在 **desc_count** 主变量中。28 行将此信息显示给用户。

29 - 31 行

此 **for** 循环仔细检查选择列表的列的项描述符。它使用 **desc_count** 主变量来确定由 DESCRIBE 语句初始化的项描述符的数目。对于每一项描述符，**for** 循环调用 prsysdesc() 函数（31 行）来保存诸如主变量中列的数据类型、长度和名称之类的信息。

```
==========================================================================
33.     for (;;)
34.         {
35.         EXEC SQL fetch demo4cursor using sql descriptor 'demo4desc';
36.         if (strncmp(SQLSTATE, "00", 2) != 0)
37.             break;
38.         /* Print out the returned values */
39.         for (i = 1; i <= desc_count; i++)
40.             {
41.             EXEC SQL get descriptor 'demo4desc' VALUE :i
42.                 :colname=NAME, :result = DATA;
43.             printf("Column: %s\tValue:%s\n ", colname, result);
44.             }
45.         printf("\n");
46.         }
==========================================================================
```

33 - 46 行

对于从数据库访存的每一行，执行此内部 **for** 循环。FETCH 语句（35 行）包括 USING SQL DESCRIPTOR 子句来指定 **demo4desc** 系统描述符区域为列值的位置。在此 FETCH 执行之后，将列值存储在指定的系统描述符区域中。

if 语句（36 和 37 行）检测 SQLSTATE 变量的值来确定该 FETCH 是否成功了。如果 SQLSTATE 包含不同于 "00" 的类代码，则 FETCH 生成警告（"01"）、NOT FOUND 条件（"02"）或错误（> "02"）。在任何这些情况下，37 行结束该 **for** 循环。

对于选择列表中每一列，39 - 45 行访问项描述符的字段。在每一 FETCH 语句之后，GET DESCRIPTOR 语句（41 和 42 行）将 DATA 字段的内容加载至恰当类型和长度的主变量内。第二个 **for** 循环（39 - 44 行）确保为选择列表中的每一列调用 GET DESCRIPTOR。

重要： 在此 GET DESCRIPTOR 语句中，demo4 程序假设返回的列为 CHAR 数据类型。如果该程序未做此假设，则它会需要检查 TYPE 和 LENGTH 字段来确定要保存该 DATA 值的主变量的恰当的数据类型。

```
==========================================================================
47.     if(strncmp(SQLSTATE, "02", 2) != 0)
48.         printf("SQLSTATE after fetch is %s\n", SQLSTATE);
49.     EXEC SQL close demo4cursor;
50.     /* free resources for prepared statement and cursor*/
51.     EXEC SQL free demo4id;
52.     EXEC SQL free demo4cursor;
53.     /* free system-descriptor area */
54.     EXEC SQL deallocate descriptor 'demo4desc';
55.     EXEC SQL disconnect current;
56.     printf("\nDEMO4 Sample Program Over.\n\n");
57. }
==========================================================================
```

47 和 48 行

在该 **for** 循环之外，程序再次检测 SQLSTATE 变量，以便于它可通知用户执行成功、运行时刻错误，或警告（类代码不等于 "02"）。

49 行

在访存所有行之后，CLOSE 语句关闭 **demo4cursor** 游标。

50 - 54 行

这些 FREE 语句释放为准备好的语句（51 行）和数据库游标（52 行）分配的资源。

DEALLOCATE DESCRIPTOR 语句（54 行）释放分配给 **demo4desc** 系统描述符区域的内存。

==========

```
58. prsysdesc(index)
59. EXEC SQL BEGIN DECLARE SECTION;
60.     PARAMETER mint index;
61. EXEC SQL END DECLARE SECTION;
62. {
63.     EXEC SQL BEGIN DECLARE SECTION;
64.         mint type;
65.         mint len;
66.         mint nullable;
67.         char name[40];
68.     EXEC SQL END DECLARE SECTION;
69.     EXEC SQL get descriptor 'demo4desc' VALUE :index
70.       :type = TYPE,
71.         :len = LENGTH,
72.           :nullable = NULLABLE,
73.         :name = NAME;
74.     printf("      Column %d: type = %d, len = %d, nullable=%d, name =
%s\n",
75.         index, type, len, nullable, name);
76. }
```

==========
==

58 - 76 行

prsysdesc() 函数显示关于选择列表列的信息。它使用 GET DESCRIPTOR 语句来访问来自 **demo4desc** 系统描述符区域的一个项描述符。

GET DESCRIPTOR 语句（70 - 74 行）访问来自 **demo4desc** 中的项描述符的 TYPE、LENGTH、NULLABLE 和 NAME 字段，来提供关于一列的信息。它将此信息存储在恰当长度和数据类型的主变量中。VALUE 关键字指示要访问的项描述符的数目。

执行单个 SELECT

demo4 程序假设该 SELECT 语句返回多行，因此，该程序将该语句与游标相关联。如果您此刻知道，您编写的程序的动态 SELECT 总是只返回一行，则可省略该游标，并使用 EXECUTE...INTO SQL DESCRIPTOR 语句，而不用 FETCH...USING SQL DESCRIPTOR。您需要使用 DESCRIBE 语句来定义选择列表列。

4.3.4 处理未知的返回值

此部分描述如何使用系统描述符区域来保存动态地执行的用户定义的函数返回的值。

要使用系统描述符区域来处理未知的函数返回值：

组装并准备 EXECUTE FUNCTION 语句。

EXECUTE FUNCTION 语句不可包括 INTO 子句。

以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域。

以 DESCRIBE...USING SQL DESCRIPTOR 语句来确定返回的一个值（或多个值）的数目和数据类型。

DESCRIBE...USING SQL DESCRIPTOR 语句为用户定义的函数返回的每一值填充项描述符。

在 DESCRIBE 语句之后，对于 SQ_EXECPROC 定义了的常量，您可测试 SQLCODE 变量（sqlca.sqlcode），来检查准备好的 EXECUTE FUNCTION 语句。

在 sqlstype.h 头文件中定义此常量。

执行 EXECUTE FUNCTION 语句，并将返回值存储在系统描述符区域中。

您用来执行用户定义的函数的语句，依赖于该函数是非游标函数，还是游标函数。后面的部分讨论如何执行每一类函数。

以 DEALLOCATE DESCRIPTOR 语句释放系统描述符区域。

执行非游标函数

非游标函数仅将一行返回值返回给应用程序。请使用 EXECUTE...INTO SQL DESCRIPTOR 语句来执行该函数，并将返回的一个值或多个值保存在系统描述符区域中。

未显式地定义作为迭代函数的外部函数，仅返回单行数据。因此，您可使用 EXECUTE...INTO SQL DESCRIPTOR 来动态地执行最外部的函数。此单行数据仅由一个值组成，因为外部函数仅可返回单个值。系统描述符区域仅包含带有单个返回值的一个项描述符。

其 RETURN 语句不包括 WITH RESUME 关键字的 SPL 函数，仅返回单行数据。因此，您可使用 EXECUTE...INTO SQL DESCRIPTOR 来动态地执行大多数 SPL 函数。SPL 函数可一次返回一个或多个值，因此，系统描述符区域包含一个或多个项描述符。

重要： 由于您通常不知道用户定义的函数返回的行数，因此，不可保证仅返回一行。如果您不使用游标来执行游标函数，则 GBase 8s ESQL/C 生成运行时刻错误。因此，总是将用户定义的函数与函数游标相关联，是一种好的做法。

下列程序片段动态地执行名为 items_pct 的 SPL 函数。此 SPL 函数计算给定的生产商代表的项占 items 表中所有项的总价的百分比。对于选中的生产商，它接受一个参数 manu_code 值，且它返回该百分比作为十进制值。下图展示 items_pct SPL 函数。

图 4. items_pct SPL 函数的代码

```
create function items_pct(mac char(3)) returning decimal;
        define tp money;
        define mc_tot money;
        define pct decimal;
        let tp = (select sum(total_price) from items);
```

```
        let mc_tot = (select sum(total_price) from items
        where manu_code = mac);
        let pct = mc_tot / tp;
        return pct;
        end function;
```

执行 SPL 函数的样例程序

该样例程序片段使用系统描述符区域来动态地执行 SPL 函数，该函数返回多组返回值。

```
==========================================================================
======
```

```
1. #include <stdio.h>
2. #include <ctype.h>
3. EXEC SQL include sqltypes;
4. EXEC SQL include sqlstype;
5. EXEC SQL include decimal;
6. EXEC SQL include datetime;
7. extern char statement[80];
8. main()
9. {
10. EXEC SQL BEGIN DECLARE SECTION;
11.     int sp_cnt, desc_count;
12.     char dyn_stmnt[80], rout_name[30];
13. EXEC SQL END DECLARE SECTION;
14. int whenexp_chk();
15.     printf("Sample ESQL program to execute an SPL function
running.\n\n");
16.     EXEC SQL whenever sqlerror call whenexp_chk;
17.     EXEC SQL connect to 'stores7';
18.     printf("Connected to stores7 database.\n");
19.     /* These next five lines hard-wire the execute function
20.     * statement.  This information could have been entered
21.     * by the user and placed into the string dyn_stmnt.
```

```
22.     */
23.     stcopy("items_pct(\"HSK\")", rout_name);
24.     sprintf(dyn_stmnt, "%s %s",
25.             "execute function", rout_name);
```

==========

19 - 25 行

对 sprintf() 的调用（24 行）为 EXECUTE FUNCTION 语句组装字符串，该语句执行 items_pct() SPL 函数。

==========

```
26.     EXEC SQL prepare spid from :dyn_stmnt;
27.     EXEC SQL allocate descriptor 'spdesc';
28.     EXEC SQL describe spid using sql descriptor 'spdesc';
29.     if(SQLCODE != SQ_EXECPROC)
30.         {
31.         printf("\nPrepared statement is not EXECUTE FUNCTION.\n");
32.         exit();
33.         }
```

==========

26 行

然后，PREPARE 语句为 EXECUTE FUNCTION 语句创建 spid 语句标识符。

27 行

ALLOCATE DESCRIPTOR 语句分配 spdesc 系统描述符区域。

28 - 33 行

DESCRIBE 语句确定 **items_pct** SPL 函数返回的值的数目和数据类型。此 DESCRIBE 包括 USING SQL DESCRIPTOR 子句来指定 **spdesc** 系统描述符区域作为这些描述的位置。

在 28 行，该程序对照 sqlstype.h 文件中定义的常量值，检测 SQLCODE 变量（**sqlca.sqlcode**）的值，来验证是否准备好了 EXECUTE FUNCTION 语句。

```
================================================================
======
        EXEC SQL get descriptor 'spdesc' :sp_cnt = COUNT;
        if(sp_cnt == 0)
            {
            sprintf(dyn_stmnt, "%s %s", "execute procedure", rout_name);
            EXEC SQL prepare spid from :dyn_stmnt;
            EXEC SQL execute spid;
            }
        else
            {
            EXEC SQL declare sp_curs cursor for spid;
            EXEC SQL open sp_curs;
            while(getrow("spdesc"))
                disp_data(:sp_cnt, "spdesc");
            EXEC SQL close sp_curs;
           EXEC SQL free sp_curs;
            }
```

```
================================================================
==
```

34 - 40 行

要获得主变量中返回值的数目，GET DESCRIPTOR 语句将 COUNT 字段的值检索至主变量内。当您需要确定该 SPL 例程返回了多少个值时，此值是有用的。如果 SPL 例程未返回值，即，COUNT 的值为零，则该 SPL 例程为过程，不是函数。因此，该程序准备 EXECUTE PROCEDURE 语句（38 行），然后使用 EXECUTE 语句（39 行）来执行该过程。EXECUTE 语句不需要使用系统描述符区域，因为该 SPL 过程没有任何返回值。

41 - 49 行

如果 SPL 例程确实返回值，即，如果 COUNT 的值大于零，则程序为准备好的 SPL 函数声明并打开 **sp_curs** 游标。

对于由该 SPL 函数返回的每一组值，执行 **while** 循环（45 和 46 行）。此循环调用 getrow() 函数来将一组值访存至 **spdesc** 系统描述符区域。然后，它调用 disp_data() 函数来显示返回的值。

在返回了所有组的返回值之后，CLOSE 语句（47 行）关闭 **sp_curs** 游标，且 FREE 语句（48 行）释放分配给该游标的资源。

```
================================================================
======
        50.     EXEC SQL free spid;
        51.     EXEC SQL deallocate descriptor 'spdesc';
        52.     EXEC SQL disconnect current;
        53. }

================================================================
==
```

50 和 51 行

此 FREE 语句释放为准备好的语句分配的资源。DEALLOCATE DESCRIPTOR 语句释放分配给 spdesc 系统描述符区域的内存。

执行游标函数

游标函数可将一行或多行返回值返回给应用程序。要执行游标函数，您必须将 EXECUTE FUNCTION 语句与函数游标相关联，并使用 FETCH...INTO SQL DESCRIPTOR

语句来保存系统描述符区域中的一个或多个返回值。

要使用系统描述符区域来保存游标函数返回值：

为用户定义的函数声明函数游标。

使用 DECLARE 语句来将 EXECUTE FUNCTION 语句与函数游标相关联。

使用 OPEN 语句来执行该函数，并打开游标。

使用 FETCH...USING SQL DESCRIPTOR 语句来将返回值从游标检索至系统描述符区域内。

使用 GET DESCRIPTOR 语句来将返回值从系统描述符区域检索至主变量内。

每一项描述符的 DATA 字段都包含返回值。

以 DEALLOCATE DESCRIPTOR 语句释放系统描述符区域。

仅定义作为迭代函数的外部函数可返回多行数据。因此，您必须定义函数游标来动态地执行迭代函数。每一行数据仅由一个值组成，因为外部函数仅可返回单个值。对于每一行，系统描述符区域仅包含一个带有单个返回值的项描述符。

其 RETURN 语句包括 WITH RESUME 关键字的 SPL 函数可返回一行或多行数据。因此，您必须定义函数游标来动态地执行这些 SPL 函数。每一行数据都由一个或多个值组成，因为 SPL 函数一次可返回一个或多个值。对于每一行，系统描述符区域包含每一返回值的项描述符。

4.3.5 处理未知的列列表

本部分描述如何使用系统描述符区域来处理 INSERT...VALUES 语句。

要使用系统描述符区域来处理 INSERT 中的输入参数：

1. （以 PREPARE 语句）准备 INSERT 语句来给它一个语句标识符。
2. 以 ALLOCATE DESCRIPTOR 语句分配系统描述符区域。
3. 以 DESCRIBE...USING SQL DESCRIPTOR 语句来确定列的数目和数据类型。对于选择列表中的每一列，DESCRIBE 语句填充一个项描述符。

以 GET DESCRIPTOR 语句来在主变量中保存未知的列的数目，其包含 COUNT 字段的值。

4. 以 SET DESCRIPTOR 语句将列设置为它们的值，其设置恰当的 DATA 和 VALUE 字段。该列值必须与它们相关联的列的数据类型相兼容。如果您想要插入 NULL 值，则请将 INDICATOR 字段设置为 -1，并不在 SET DESCRIPTOR 语句中指定任何 DATA 字段。

执行 INSERT 语句来将值插入至数据库内。

后面的部分演示如何执行简单的 INSERT 语句，其仅插入一行，以及使用插入游标来从插入缓冲区插入几行的演示。

以 DEALLOCATE DESCRIPTOR 语句释放系统描述符区域。

执行简单的插入

下列步骤概括如何以系统描述符区域执行简单的 INSERT 语句：

（以 PREPARE 语句）准备 INSERT 语句，并给它一个语句标识符。

以 SET DESCRIPTOR 语句将该列设置为它们的值。

以 EXECUTE...USING SQL DESCRIPTOR 语句执行 INSERT 语句。

执行动态 INSERT 语句的样例程序

此样例程序展示如何执行动态 INSERT 语句。此 INSERT 语句不与插入游标相关联。

该程序将两个 TEXT 值插入至 **txt_a** 表内。它从名为 desc_ins.txt 的命名的文件读取该文本值。然后，该程序从此表选择列，并将 TEXT 值存储在两个命名的文件 txt_out1 和 txt_out2 中。该程序说明使用系统描述符区域来处理列列表中的列。

==========

```
1. EXEC SQL include locator;
2. EXEC SQL include sqltypes;
3. main()
4. {
5.      EXEC SQL BEGIN DECLARE SECTION;
6.           int ;
7.           int cnt;
8.           ifx_loc_t loc1;
9.           ifx_loc_t loc2;
10.     EXEC SQL END DECLARE SECTION;
11.     EXEC SQL create database txt_test;
```

```
12.     chkerr("CREATE DATABASE txt_test");
13.     EXEC SQL create table txt_a (t1 text not null, t2 text);
14.     chkerr("CREATE TABLE t1");
15.     /* The INSERT statement could have been created at runtime. */
16.     EXEC SQL prepare sid from 'insert into txt_a values (?, ?)';
17.     chkerr("PREPARE sid");
```

==========

5 - 10 行

这些行声明主变量来保存要插入的列值（从用户获得）。

15 - 17 行

这些行为该语句组装字符串，并准备它作为 **sid** 语句标识符。该输入参数指定 INSERT 语句的找不到的行。INSERT 语句在此为硬编码，但可在运行时刻创建它。

==========

```
18.     EXEC SQL allocate descriptor 'desc';
19.     chkerr("ALLOCATE DESCRIPTOR desc");
20.     EXEC SQL describe sid using sql descriptor 'desc';
21.     chkerr("DESCRIBE sid");
22.     EXEC SQL get descriptor 'desc' :cnt = COUNT;
23.     chkerr("GET DESCRIPTOR desc");
24.     for (i = 1; i <= cnt; i++)
25.         prsysdesc(i);
```

==========

18 和 19 行

要能够使用列的系统描述符区域，您必须首先分配该系统描述符区域。此 ALLOCATE DESCRIPTOR 语句分配名为 **desc** 的系统描述符区域。

20 和 21 行

DESCRIBE 语句为 **sid** 标识的准备好的 INSERT 描述列。此 DESCRIBE 语句包括 USING SQL DESCRIPTOR 子句来指定 **desc** 系统描述符区域作为这些列描述的位置。

22 和 23 行

GET DESCRIPTOR 语句获得由 DESCRIBE 发现的列的数目（COUNT 字段）。在 **cnt** 主变量中存储此数目。

24 和 25 行

对于 INSERT 语句的列，此 for 循环详细检查项描述符。它使用 cnt 变量来确定由 DESCRIBE 初始化的项描述符的数目。对于每一项描述符，prsysdesc() 函数在主变量中保存诸如数据类型、长度和名称这样的信息。

```
==============================================================================
```

```
26.     loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
27.     loc1.loc_fname = loc2.loc_fname = "desc_ins.txt";
28.     loc1.loc_size = loc2.loc_size = -1;
29.     loc1.loc_oflags = LOC_RONLY;
30.     i = CLOCATORTYPE;
31.     EXEC SQL set descriptor 'desc' VALUE 1
32.         TYPE = :i, DATA = :loc1;
33.     chkerr("SET DESCRIPTOR 1");
34.     EXEC SQL set descriptor 'desc' VALUE 2
```

```
35.         TYPE = :i, DATA = :loc2;
36.     chkerr("SET DESCRIPTOR 2");
37.     EXEC SQL execute sid using sql descriptor 'desc';
38.     chkerr("EXECUTE sid");
```

==========

26 - 29 行

要插入 TEXT 值，该程序必须首先以 GBase 8s ESQL/C 定位符结构分配该值。**loc1** 定位符结构为 **txt_a** 表的 **t1** 列存储 TEXT 值；**loc2** 是 **txt_a** 的 **t2** 列的定位符结构。(参见 13 行)。该程序包括 GBase 8s ESQL/Clocator.h 头文件(1 行)来定义 **ifx_loc_t** 结构。

两个 TEXT 值都位于名为 desc_ins.txt 的命名的文件（loc_loctype = LOCFNAME）中。当您将 loc_size 字段设置为 -1 时，该定位符结构告诉 GBase 8s ESQL/C 将 TEXT 值在单个操作中发送给数据库服务器。

30 - 36 行

第一个 SET DESCRIPTOR 语句设置 **t1** 列的项描述符中的 TYPE 和 DATA 字段（VALUE 1）。该数据类型为 CLOCATORTYPE（在 GBase 8s ESQL/Csqltypes.h 头文件中定义）来指示将列值存储在 GBase 8s ESQL/C 定位符结构中；将该数据设置为 **loc1** 定位符结构。第二个 SET DESCRIPTOR 语句对 **t2** 列值执行相同的任务；它将它的 DATA 字段设置为 **loc2** 定位符结构。

37 和 38 行

数据库服务器以 EXECUTE...USING SQL DESCRIPTOR 语句执行 INSERT 语句，来从 **desc** 系统描述符区域获得新的列值。

==========

```
39.     loc1.loc_loctype = loc2.loc_loctype = LOCFNAME;
```

```
40.       loc1.loc_fname = "txt_out1";
41.       loc2.loc_fname = "txt_out2";
42.       loc1.loc_oflags = loc2.loc_oflags = LOC_WONLY;
43.       EXEC SQL select * into :loc1, :loc2 from a;
44.       chkerr("SELECT");
45.       EXEC SQL free sid;
46.       chkerr("FREE sid");
47.       EXEC SQL deallocate descriptor 'desc';
48.       chkerr("DEALLOCATE DESCRIPTOR desc");
49.       EXEC SQL close database;
50.       chkerr("CLOSE DATABASE txt_test");
51.       EXEC SQL drop database txt_test;
52.       chkerr("DROP DATABASE txt_test");
53       EXEC SQL disconnect current;
54. }
55. chkerr(s)
56. char *s;
57. {
58. if (SQLCODE)
59.       printf("%s error %d\n", s, SQLCODE);
60. }
```

==========

39 - 44 行

该程序使用 **loc1** 和 **loc2** 定位符结构来选择插入的值。将这些 TEXT 值读取至命名的文件：**t1** 列（在 **loc1** 中）读取至 **txt_out1**，**t2** 列（在 **loc2** 中）读取至 **txt_out2**。LOC_WONLY 的 **loc_oflags** 值意味着此 TEXT 数据重写这些输出文件中任何现有的数据。

45 - 48 行

FREE 语句（45 行）释放为 **sid** 准备好的语句分配的资源。一旦释放了准备好的语句，在该程序中就不可再次使用它。DEALLOCATE DESCRIPTOR 语句（46 行）释放分配给 **desc** 系统描述符区域的内存。

55 - 60 行

chkerr() 函数是一个简单的异常处理例程。它为非零值检查全局的 SQLCODE 变量。由于零指示 SQL 语句的成功执行，因此，每当发生运行时刻错误时，都执行 printf()（58 行）。

执行与游标相关联的 INSERT

对于从插入缓冲区插入行的 INSERT 语句的列列表值，您的 GBase 8s ESQL/C 程序必须仍使用 DESCRIBE 和 SET DESCRIPTOR 语句来使用系统描述符区域。它还必须随同插入游标使用 PUT...USING SQL DESCRIPTOR 语句，如下：

准备 INSERT 语句，并以 DECLARE 语句将它与插入游标相关联。所有多行 INSERT 语句都*必须*有一声明了的插入游标。

以 OPEN 语句为 INSERT 语句创建该游标。

以 PUT 语句及其 USING SQL DESCRIPTOR 子句来将第一组列插入至插入缓冲区内。在此 PUT 语句之后，将存储在指定的系统描述符区域中的列值存储在插入缓冲区中。在循环内重复该 PUT 语句，直到没有更多的行要插入为止。

在插入所有行之后，退出该循环，并以 FLUSH 语句刷新该插入缓冲区。

以 CLOSE 语句关闭该插入游标。

您处理该插入游标的方式，与您处理与 SELECT 语句相关联的游标的方式相同。

4.3.6 处理参数化的 SELECT 语句

本部分描述如何处理带有系统描述符区域的参数化的 SELECT 语句。如果准备好的 SELECT 语句有带有未知数目和数据类型的输入参数的 WHERE 子句，则您的 GBase 8s ESQL/C 程序必须使用系统描述符区域来定义输入参数。

要使用系统描述符区域来定义 WHERE 子句的输入参数：

1. 确定 SELECT 语句的输入参数的数目和数据类型。
2. 分配系统描述符区域，并以 ALLOCATE DESCRIPTOR 语句为它指定名称。
3. 以 SET DESCRIPTOR 语句指示 WHERE 子句中输入参数的数目，其设置 COUNT 字段。

4. 以 SET DESCRIPTOR 语句存储该定义和每一输入参数的值，其设置在恰当的项描述符中的 DATA、TYPE 和 LENGTH 字段：
   - TYPE 字段必须使用在 sqltypes.h 头文件中定义的 GBase 8s ESQL/C 数据类型常量来表示输入参数的数据类型。

对于 CHAR 或 VARCHAR 值，LENGTH 是以字节计的字符数组的大小；对于 DATETIME 或 INTERVAL 值，LENGTH 字段存储编码的限定符。

重要： 如果您使用 X/Open 代码（且以 -xopen 标志编译），则对于 TYPE 和 ITYPE 字段，您必须使用 X/Open 数据类型值。

如果您使用指示符变量，则您还需要设置 INDICATOR 字段，可能以及 IDATA、ILENGTH 和 ITYPE 字段(仅限于非 X/Open 应用程序)。请使用 SET DESCRIPTOR 的 VALUE 关键字来标识项描述符。

5. 以 USING SQL DESCRIPTOR 子句将定义了的输入参数从系统描述符区域传至数据库服务器。

提供输入参数的语句依赖于该 SELECT 语句返回多少行。后面的部分讨论如何执行每一类 SELECT 语句。

6. 以 DEALLOCATE DESCRIPTOR 语句释放系统描述符区域。

重要： 如果 SELECT 语句在选择列表中有未知的列，则您的程序还必须处理这些带有系统描述符区域的列。

执行返回多行的参数化的 SELECT

下列样例程序展示如何使用带有下列条件的动态 SELECT 语句：

SELECT 返回多行。

该 SELECT 必须与游标相关联，以 OPEN 语句执行，并有以 FETCH...USING SQL DESCRIPTOR 语句检索的它的返回值。

该 SELECT 在它的 WHERE 子句中有输入参数。

OPEN 语句包括 USING SQL DESCRIPTOR 子句来提供系统描述符区域中的参数值。

该 SELECT 在选择列表中有未知的列。

FETCH 语句包括 USING SQL DESCRIPTOR 子句来在系统描述符区域中存储返回值。

使用动态 SELECT 语句的样例程序

该程序是 demo4.ec 样例程序的一个版本；demo4 使用选择列表列的系统描述符区域，而本 demo4 的修改版本对于选择列表列和 WHERE 子句的输入参数同时使用系统描述符

区域。

```
1. #include <stdio.h>
2. EXEC SQL include sqltypes;
3.
4. EXEC SQL define NAME_LEN 15;
5. EXEC SQL define MAX_IDESC 4;
6. main()
7. {
8. EXEC SQL BEGIN DECLARE SECTION;
9.       int i;
10.       int desc_count;
11.       char demoquery[80];
12.      char queryvalue[2];
13.         char result[ NAME_LEN + 1 ];
14. EXEC SQL END DECLARE SECTION;
15.       printf("Modified DEMO4 Sample ESQL program running.\n\n");
16.       EXEC SQL connect to 'stores7';
```

8 - 14 行

这些行声明主变量来保存从用户取得的数据，以及从系统描述符检索的列值。

```
17.       /* These next three lines have hard-wired both the query and
18.         * the value for the parameter. This information could have
19.         * been entered from the terminal and placed into the strings
```

```
20.      * demoquery and queryvalue, respectively.
21.      */
22.     sprintf(demoquery, "%s %s",
23.         "select fname, lname from customer",
24.         "where lname < ? ");
25.     EXEC SQL prepare demoid from :demoquery;
26.     EXEC SQL declare democursor cursor for demoid;
27.     EXEC SQL allocate descriptor 'demodesc' with max MAX_IDESC;
```

==========

17 - 25 行

这些行为（demoquery）中的语句组装字符串，并准备它作为 demoid 语句标识符。问号（?）指示 WHERE 子句中的输入参数。

26 行

此行为准备好的语句标识符 **demoid** 声明 **democursor** 游标。所有非单个的 SELECT 语句都必须有一个声明了的游标。

27 行

要能够使用输入参数的系统描述符区域，您必须首先分配该系统描述符区域。此 ALLOCATE DESCRIPTOR 语句分配 demodesc 系统描述符区域。

==========

```
28.     /*      This section of the program must evaluate :demoquery
29.      *      to count how many question marks are in the where
30.      *     clause and what kind of data type is expected for each
31.      *     question mark.
```

```
32.     *       For this example, there is one parameter of type
33.     *       char(15). It would then obtain the value for
34.     *       :queryvalue. The value of queryvalue is hard-wired in
35.     *       the next line.
36.     */
37.    sprintf(queryvalue, "C");
38.    desc_count = 1;
39.    if(desc_count > MAX_IDESC)
40.        {
41.        EXEC SQL deallocate descriptor 'demodesc';
42.        EXEC SQL allocate descriptor 'demodesc' with max :desc_count;
43.        }
44.    /* number of parameters to be held in descriptor is 1 */
45.    EXEC SQL set descriptor 'demodesc' COUNT = :desc_count;
```

==========

28 - 38 行

这些行模仿输入参数值的动态条目。虽然该参数值在此为硬编码（37 行），但该程序更可能会从用户输入取得该值。38 行模仿会确定在语句字符串中存在多少输入参数的代码。如果您不知道此值，则您会需要包括 C 代码来为问号（?）字符解析该语句字符串。

39 - 43 行

对于参数化的 SELECT 语句，此 **if** 语句确定 **demodesc** 系统描述符区域是否包含足够的项描述符。它将语句字符串中的输入参数的数目（**desc_count**）与当前实际分配的项描述符的数目（MAX_IDESC）相比较。如果该程序尚未分配足够的项描述符，则该程序释放现有的系统描述符区域（41 行），并分配新的（42 行）；它使用 WITH MAX 子句中输入参数的实际数目来指定要分配的项描述符的数目。

44 和 45 行

此 SET DESCRIPTOR 语句在 **demodesc** 系统描述符区域的 COUNT 字段中存储输入参数的数目。

```
46.     /* Put the value of the parameter into the descriptor */
47.     i = SQLCHAR;
48.     EXEC SQL set descriptor 'demodesc' VALUE 1
49.             TYPE = :i, LENGTH = 15, DATA = :queryvalue;
50.     /* Associate the cursor with the parameter value */
51.     EXEC SQL open democursor using sql descriptor :demodesc;
52.     /*Reuse the descriptor to determine the contents of the Select-
* list*/
53.     EXEC SQL describe qid using sql descriptor 'demodesc';
54.     EXEC SQL get descriptor 'demodesc' :desc_count = COUNT;
55.     printf("There are %d returned columns:\n", desc_count);
56.     /* Print out what DESCRIBE returns */
57.     for (i = 1; i <= desc_count; i++)
58.         prsysdesc(i);
59.     printf("\n\n");
```

47 - 49 行

此 SET DESCRIPTOR 语句为 WHERE 子句中的每一参数设置 TYPE、LENGTH（对于 CHAR 值）和 DATA 字段。该程序仅调用 SET DESCRIPTOR 一次，因为它假设该 SELECT 语句仅有一个输入参数。如果您在编译时刻不知道输入参数的数目，则请将 SET DESCRIPTOR 放至循环中，**desc_count** 主变量为其控制迭代的数目。

50 和 51 行

当数据库服务器打开 **democursor** 游标时，它执行 SELECT 语句。此 OPEN 语句包括 USING SQL DESCRIPTOR 子句来指定 **demodesc** 系统描述符区域作为输入参数值的位置。

52 - 59 行

该程序还使用 demodesc 系统描述符区域来保存由 SELECT 语句返回的列。DESCRIBE 语句（53 行）检测选择列表，来确定这些列的数目的数据类型。然后，GET DESCRIPTOR 语句（54 行）从 demodesc 的 COUNT 字段获得描述的列的数目。然后，55 - 58 行显示每一返回的列的列信息。

```
==========================================================================
======
```

```
60.     for (;;)
61.         {
62.         EXEC SQL fetch democursor using sql descriptor 'demodesc';
63.         if (sqlca.sqlcode != 0) break;
64.         for (i = 1; i <= desc_count; i++)
65.             {
66.             EXEC SQL get descriptor 'demodesc' VALUE :i :result = DATA;
67.             printf("%s ", result);
68.             }
69.         printf("\n");
70.         }
71.     if(strncmp(SQLSTATE, "02", 2) != 0)
72.         printf("SQLSTATE after fetch is %s\n", SQLSTATE);
73.     EXEC SQL close democursor;
74.     EXEC SQL free demoid; /* free resources for statement */
75.     EXEC SQL free democursor; /* free resources for cursor */
76.     /* free system-descriptor area */
```

```
77.     EXEC SQL deallocate descriptor 'demodesc';
78.     EXEC SQL disconnect current;
79.     printf("\nModified DEMO4 Program Over.\n\n");
80. }
```

==========

60 - 70 行

对于选择列表中的每一列，这些行访问项描述符的字段。在每一 FETCH 语句之后，GET DESCRIPTOR 语句将 DATA 字段的内容加载至 **result** 主变量内。

73 行

访存所有行之后，CLOSE 语句释放分配给 **democursor** 游标的活动集的资源。

74 - 77 行

在 75 行的 FREE 语句释放分配给 democursor 游标的资源时，74 行的 FREE 语句释放分配给 demoid 语句标识符的资源。DEALLOCATE DESCRIPTOR 语句释放分配给 demodesc 系统标识符区域的资源。

执行参数化的单个 SELECT 语句

前一部分中的说明，假设该参数化的 SELECT 语句返回多行，因此，与游标相关联。如果您在编写该程序的时候知道，该参数化的 SELECT 语句总是只返回一行，则您可省略该游标，并使用 EXECUTE...USING SQL DESCRIPTOR...INTO 语句，而不是 OPEN...USING SQL DESCRIPTOR 语句，来指定来自系统描述符区域的参数值。

4.3.7 处理参数化的用户定义的例程

本部分描述如何处理带有系统描述符区域的参数化的用户定义的例程。下列语句执行用户定义的例程：

EXECUTE FUNCTION 语句执行用户定义的函数（外部的和 SPL）。

EXECUTE PROCEDURE 语句执行用户定义的过程（外部的和 SPL）。

如果准备好的 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 有指定作为未知数目和水力学的输入参数的参数，则您的 GBase 8s ESQL/C 程序可使用系统描述符区域来定义输入参数。

执行参数化的函数

您处理用户定义的函数的输入参数的方式，与处理 SELECT 语句的 WHERE 子句中的输入参数的方式一样，如下：

执行非游标函数的方式，与执行单个 SELECT 语句的方式一样。

如果您在此时知道您编写的动态用户定义的函数的程序总是只返回一行，则您可使用 EXECUTE...USING SQL DESCRIPTOR...INTO 语句来提供来自系统描述符区域的参数值，并执行该函数。

执行游标函数的方式，与执行返回一行或多行的 SELECT 语句的方式一样。

如果您在此时不确定您编写的动态用户定义的函数的程序是否只返回一行，则请定义函数游标，并使用 OPEN...USING SQL DESCRIPTOR 语句来提供来自系统描述符区域的参数值。

执行这些 EXECUTE FUNCTION 语句与执行 SELECT 语句执行唯一的区别在于，您为非游标函数准备 EXECUTE FUNCTION 语句，而不是 SELECT 语句。

执行参数化的过程

要执行参数化的用户定义的过程，您可使用 EXECUTE...USING SQL DESCRIPTOR 语句来提供来自系统描述符区域的参数值，并执行该过程。您处理用户定义的过程输入参数的方式，与处理非游标函数中的输入参数的方式相同。执行 EXECUTE PROCEDURE 语句与执行 EXECUTE FUNCTION 语句（对于非游标函数）的唯一区别在于，您不需要为用户定义个过程指定 EXECUTE...USING SQL DESCRIPTOR 语句的 INTO 子句。

4.3.8 处理参数化的 UPDATE 或 DELETE 语句

您确定 DELETE 或 UPDATE 语句的 WHERE 子句中的输入参数的方式，与您确定 SELECT 语句的 WHERE 子句中的它们的方式类似。这两类动态的参数化的语句的主要区别如下：

您不需要使用游标来处理 DELETE 或 UPDATE 语句。因此，您以 EXECUTE 语句的 USING SQL DESCRIPTOR 子句来提供来自系统描述符区域的参数值，而不是 OPEN 语句。

您可使用 DESCRIBE...USING SQL DESCRIPTOR 语句来确定 DELETE 或 UPDATE 语句是否有 WHERE 子句。

4.3.9 dyn_sql 程序

dyn_sql.ec 程序是一个使用动态 SQL 的 GBase 8s ESQL/C 演示程序。该程序提示用户输入 **stores7** 演示数据库的 SELECT 语句，然后，使用系统描述符区域来动态地执行该 SELECT。

在缺省情况下，该程序打开 **stores7** 数据库。然而，如果给定了演示数据库的名称而不是 **stores7**，则您可在命令行上指定数据库名称。下列命令在 **mystores7** 数据库上运行可执行的 **dyn_sql**：

dyn_sql mystores7

编译该程序

请使用下列命令来编译 **dyn_sql** 程序：

esql -o dyn_sql dyn_sql.ec

-o dyn_sql 选项导致将可执行程序命名为 **dyn_sql**。不带有 **-o** 选项，则可执行程序的名称缺省为 a.out。

dyn_sql.ec 文件指南

```
================================================================================
1. /*
2.      This program prompts the user to enter a SELECT statement
3.      for the stores7 database. It processes the statement using
         dynamic sql
4.      and system descriptor areas and displays the rows returned by the
5.      database server.
6. */
7. #include <stdio.h>
8. #include <stdlib.h>
9. #include <ctype.h>
10. EXEC SQL include sqltypes;
11. EXEC SQL include locator;
12. EXEC SQL include datetime;
13. EXEC SQL include decimal;
14. #define WARNNOTIFY          1
15. #define NOWARNNOTIFY        0
16. #define LCASE(c) (isupper(c) ? tolower(c) : (c))
```

17. #define BUFFSZ 256

18. extern char statement[80];

==========

7 - 13 行

这些行指定要包括在该程序中的 C 和 GBase 8s ESQL/C 文件。stdio.h 文件启用 **dyn_sql** 来使用标准 C I/O 库。stdlib.h 文件包含从字符串至数值的转换函数、内存分配函数和各种各样的标准库函数。ctypes.h 文件包含检查字符的属性的宏。例如，确定字符为大写还是小写的宏。

sqltypes.h 头文件包含对应于在 GBase 8s 数据库中找到的数据类型的符号常量。该程序使用这些常量来确定动态 SELECT 语句返回的列的数据类型。

locator.h 文件包含定位器结构（**ifx_loc_t**）的定义，其为 TEXT 和 BYTE 列所需要的主变量的类型。datetime.h 文件包含 **datetime** 和 **interval** 结构的定义，其为 DATETIME 和 INTERVAL 列的主变量的数据类型。decimal.h 文件包含 **dec_t** 结构的定义，其为 DECIMAL 列所需要的主变量的类型。

14 - 17 行

exp_chk() 异常处理函数使用 WARNNOTIFY 和 NOWARNNOTIFY 常量（14 和 15 行）。exp_chk() 的第二个参数告诉该函数显示 SQLSTATE 中的信息，以及警告的 SQLCODE 变量（WARNNOTIFY），或不显示警告的信息（NOWARNNOTIFY）。exp_chk() 函数在 exp_chk.ec 源文件中。

16 行定义 LCASE，一个将大写字符转换为小写字符的宏。17 行将 BUFFSZ 定义为数值 256。该程序使用 BUFFSZ 来指定存储来自用户的输入的数组的大小。

18 行

18 行声明 **statement** 作为外部全局变量，来保存该程序请求数据库服务器执行的最

后 的 SQL 语句的名称。该异常处理函数使用此信息（请参阅 399 - 406 行。）

```
==========================================================================
======
19. EXEC SQL BEGIN DECLARE SECTION;
20.         ifx_loc_t lcat_descr;
21.         ifx_loc_t lcat_picture;
22. EXEC SQL END DECLARE SECTION;
23. mint whenexp_chk();
24. main(argc, argv)
25. mint argc;
26. char *argv[];
27. {
28.       int4 ret, getrow();
29.       short data_found = 0;
30.       EXEC SQL BEGIN DECLARE SECTION;
31.             char ans[BUFFSZ], db_name[30];
32.             char name[40];
33.             mint sel_cnt, i;
34.             short type;
35.       EXEC SQL END DECLARE SECTION;
36.        printf("DYN_SQL Sample ESQL Program running.\n\n");
37.        EXEC SQL whenever sqlerror call whenexp_chk;
38.        if (argc > 2)                        /* correct no. of args? */
39.           {
40.           printf("\nUsage: %s [database]\nIncorrect no. of
                argument(s)\n",
41.                argv[0]);
42.           printf("\nDYN_SQL Sample Program over.\n\n");
43.           exit(1);
44.           }
45.       strcpy(db_name, "stores7");
46.       if(argc == 2)
```

```
47.             strcpy(db_name, argv[1]);
48.         sprintf(statement,"CONNECT TO %s",db_name);
49.         EXEC SQL connect to :db_name;
50.         printf("Connected to %s\n", db_name);
51.         ++argv;
```

==========

19 - 23 行

19 - 23 行定义在 SQL 语句中使用的全局主变量。20 和 21 行定义定位器结构，其为 **catalog** 表的 **cat_descr** 和 **cat_picture** 列的主变量。23 行声明 whenexp_chk() 函数，当发生 SQL 语句错误时，该程序调用它。

24 - 27 行

main() 函数是该程序开始执行的始点。**argc** 函数给定当调用该程序时，来自命令行的参数的数目。**argv** 是一个执行命令行参数的指针的数组。此程序仅期望一个参数（要访问的数据库的名称），且它是可选的。

28 - 51 行

28 行定义 int4 数据类型（ret）来接收来自 getrow() 函数的返回值。28 行还声明 getrow 函数返回 int4 数据类型。30 – 35 行定义 main() 函数块的本地主变量。如果在 SQL 语句中发生任何错误，则 37 行执行 WHENEVER 语句来将控制转移至 whenexp_chk()。

38 - 51 行建立到数据库的连接。如果 **argc** 等于 2，则该程序假设用户在命令行上输入了数据库名称（按照惯例，第一个参数是该程序的名称），且该程序打开此数据库。如果用户未在命令行上输入数据库名称，则该程序打开 **stores7** 数据库（请参阅 45 行），这是缺省的。在这两种情况下，该程序连接到由 GBASEDBTSERVER 环境变量指定的缺省数据库服务器，因为未指定数据库服务器。

```
    ================================================================
======
52.     while(1)
53.         {
54.         /* prompt for SELECT statement */
55.         printf("\nEnter a SELECT statement for the %s database",
56.                     db_name);
57.         printf("\n\t(e.g.    select * from customer;)\n");
58.         printf("\tOR a ';' to terminate program:\n>> ");
59.         if(!getans(ans, BUFFSZ))
60.             continue;
61.         if (*ans == ';')
62.             {
63.             strcpy(statement, "DISCONNECT");
64.             EXEC SQL disconnect current;
65.             printf("\nDYN_SQL Sample Program over.\n\n");
66.             exit(1);
67.             }
68.         /* prepare statement id */
69.         printf("\nPreparing statement (%s)...\n", ans);
70.         strcpy(statement, "PREPARE sel_id");
71.         EXEC SQL prepare sel_id from :ans;
72.         /* declare cursor */
73.         printf("Declaring cursor 'sel_curs' for SELECT...\n");
74.         strcpy(statement, "DECLARE sel_curs");
75.         EXEC SQL declare sel_curs cursor for sel_id;
76.         /* allocate descriptor area */
77.         printf("Allocating system-descriptor area...\n");
78.         strcpy(statement, "ALLOCATE DESCRIPTOR selcat");
79.         EXEC SQL allocate descriptor 'selcat';
80.     /* Ask the database server to describe the statement */
81.         printf("Describing prepared SELECT...\n");
82.         strcpy(statement,
```

```
83.             "DESCRIBE sel_id USING SQL DESCRIPTOR selcat");
84.         EXEC SQL describe sel_id using sql descriptor 'selcat';
85.         if (SQLCODE != 0)
86.             {
87.             printf("** Statement is not a SELECT.\n");
88.             free_stuff();
89.             strcpy(statement, "DISCONNECT");
90.             EXEC SQL disconnect current;
91.             printf("\nDYN_SQL Sample Program over.\n\n");
92.             exit(1);
93.             }
```

==========

52 - 67 行

52 行上的 **while(1)** 开始一个持续到 main() 函数的结束的循环。55 - 58 行提示用户输入 SELECT 语句，或输入分号终止该程序。getans() 函数从用户接收该输入。如果第一个字符不是分号，则该程序继续处理输入。

68 - 75 行

PREPARE 语句准备来自数组 **ans[]** 的（用户输入的）SELECT 语句，并为它指定语句标识符 **sel_id**。PREPARE 语句使得数据库服务器能够解析、验证并为该语句生成执行计划。

DECLARE 语句（72 - 75 行）为 SELECT 语句返回的行集创建 **sel_curs** 游标，如果它返回多行的话。

76 - 79 行

ALLOCATE DESCRIPTOR 语句在内存中分配 **selcat** 系统描述符区域。该语句不包括 WITH MAX 子句，因此使用缺省的内存分配，其为 100 列分配。

80 - 93 行

DESCRIBE 语句从数据库服务器获得关于 **sel_id** 语句标识符中的语句的信息。数据库服务器返回 **selcat** 系统描述符区域中的信息，前面的 ALLOCATE DESCRIPTOR 语句创建该信息。DESCRIBE 放入系统描述符区域内的信息，包括选择列表中列的名称、数据类型和长度。

DESCRIBE 语句还将 SQLCODE 变量设置为指示被描述的语句的类型的数值。要检查该语句类型是否为 SELECT，85 行将 SQLCODE 的值与 0 相比较（对于不带有 INTO TEMP 子句的 SELECT 语句，在 sqlstypes.h 文件中定义该值）。如果该语句不是 SELECT，则 87 行显示相关意思的消息，该程序释放已分配了的游标和资源。然后，它关闭连接并退出。

```
=================================================================
======
94.      /* Determine the number of columns in the select list */
95.          printf("Getting number of described values from ");
96.          printf("system-descriptor area...\n");
97.          strcpy(statement, "GET DESCRIPTOR selcat: COUNT field");
98.          EXEC SQL get descriptor 'selcat' :sel_cnt = COUNT;
99.       /* open cursor; process select statement */
100.           printf("Opening cursor 'sel_curs'...\n");
101.           strcpy(statement, "OPEN sel_curs");
102.           EXEC SQL open sel_curs;
103.      /*
104.       * The following loop checks whether the cat_picture or
105.       * cat_descr columns are described in the system-descriptor area.
106.       * If so, it initializes a locator structure to read the simple
107.       * large-object data into memory and sets the address of the
108.       * locator structure in the system-descriptor area.
```

```
109.      */
110.           for(i = 1; i <= sel_cnt; i++)
111.              {
112.              strcpy(statement,
113.                    "GET DESCRIPTOR selcat: TYPE, NAME fields");
114.              EXEC SQL get descriptor 'selcat' VALUE :i
115.                  :type = TYPE,
116.                  :name = NAME;
117.              if (type == SQLTEXT && !strncmp(name, "cat_descr",
118.                       strlen("cat_descr")))
119.                  {
120.                  lcat_descr.loc_loctype = LOCMEMORY;
121.                  lcat_descr.loc_bufsize = -1;
122.                  lcat_descr.loc_oflags = 0;
123.                  strcpy(statement, "SET DESCRIPTOR selcat: DATA field");
124.                  EXEC SQL set descriptor 'selcat' VALUE :i
125.                       DATA = :lcat_descr;
126.                  }
127.              if (type == SQLBYTES && !strncmp(name, "cat_picture",
128.                       strlen("cat_picture")))
129.                  {
130.                  lcat_picture.loc_loctype = LOCMEMORY;
131.                  lcat_picture.loc_bufsize = -1;
132.                  lcat_picture.loc_oflags = 0;
133.                  strcpy(statement, "SET DESCRIPTOR selcat: DATA field");
134.                  EXEC SQL set descriptor 'selcat' VALUE :i
135.                       DATA = :lcat_picture;
136.                  }
137.              }
```

==========

94 - 98 行

GET DESCRIPTOR 语句从 **selcat** 系统描述符区域检索 COUNT 值。该 COUNT 值指示在该系统描述符区域中描述了多少列。

99 - 102 行

OPEN 语句开始执行动态 SELECT 语句，并为它返回的列集激活 **sel_curs** 游标。

114 - 137 行

本部分代码使用 GET DESCRIPTOR 语句来确定在选择列表中是否包括来自 **catalog** 表的简单大对象（**cat_descr** 和 **cat_picture**）。如果您动态地选择简单大对象列，则您必须将定位符结构的地址设置到该项描述符的 DATA 字段内，以便告诉数据库将定位器结构返回至哪里。

然而，首先，该程序初始化定位器结构，如下：

在内存缓冲区中返回的数据（**loc_loctype** = LOCMEMORY）。

数据库服务器分配该内存缓冲区（**loc_bufsize** = -1）。

然后，该程序使用 SET DESCRIPTOR 语句来将定位器结构的地址加载至描述符区域的 DATA 字段。

```
==============================================================================
138.        while(ret = getrow("selcat"))              /* fetch a row */
139.            {
140.            data_found = 1;
141.            if (ret < 0)
142.                {
143.                strcpy(statement, "DISCONNECT");
144.                EXEC SQL disconnect current;
```

```
145.                    printf("\nDYN_SQL Sample Program over.\n\n");
146.                    exit(1);
147.                    }
148.                disp_data(sel_cnt, "selcat");              /* display the data */
149.                }
150.            if (!data_found)
151.                printf("** No matching rows found.\n");
152.            free_stuff();
153.            if (!more_to_do())              /* More to do? */
154.                break;                  /* no, terminate loop */
155.            }
156. }
157. /* fetch the next row for selected items */
158. int4 getrow(sysdesc)
159. EXEC SQL BEGIN DECLARE SECTION;
160.        PARAMETER char *sysdesc;
161. EXEC SQL END DECLARE SECTION;
162. {
163.        int4 exp_chk();
164.        sprintf(statement, "FETCH %s", sysdesc);
165.        EXEC SQL fetch sel_curs using sql descriptor :sysdesc;
166.        return((exp_chk(statement)) == 100 ? 0 : 1);
167. }
```

```
=================================================================
======
```

138 - 149 行

getrow() 函数逐一地检索选择了的行。**while** 循环的每一迭代检索一行，然后，该程序以 disp_data() 函数（148 行）处理这行。当检索了所有行时，getrow() 返回 0（零），该 **while** 循环结束。

152 行

free_stuff() 函数释放当处理动态 SELECT 语句时分配了的那些资源。

153 - 156 行

当处理了所有选择的行时，该程序调用 more_to_do() 函数，其询问用户是否要处理更多的 SELECT 语句。如果回答为否，则 more_to_do() 返回 0，且 **break** 语句终止始于 52 行的 **while** 循环。如果回答为是，则程序开始 52 行上的 **while** 语句的下一迭代，来接收和处理另一 SELECT 语句。

157 - 167 行

getrow() 函数移动游标，然后访存由动态 SELECT 语句返回的行集中的下一行。它将该行值访存至系统描述符区域内，在 **sysdesc** 变量中指定它。如果没有更多的行要访存（exp_chk() 返回 100），则 getrow() 返回 0。如果 FETCH 遇到运行时刻错误，则 getrow() 返回 1。

```
==========================================================================
======
168. {/*
169.    * This function loads a column into a host variable of the correct
170.    * type and displays the name of the column and the value, unless
          * the
171.    * value is NULL.
172. disp_data(col_cnt, sysdesc)
173.    */
174. mint col_cnt;
175. EXEC SQL BEGIN DECLARE SECTION;
176.        PARAMETER char *sysdesc;
177. EXEC SQL END DECLARE SECTION;
178.        EXEC SQL BEGIN DECLARE SECTION;
179.             mint int_data, i;
180.             char *char_data;
```

```
181.        int4 date_data;
182.        datetime dt_data;
183.        interval intvl_data;
184.        decimal dec_data;
185.          short short_data;
186.        char name[40];
187.        short char_len, type, ind;
188.    EXEC SQL END DECLARE SECTION;
189.    int4 size;
190.    unsigned amount;
191.    mint x;
192.    char shdesc[81], str[40], *p;
193.    printf("\n\n");
194.    /* For each column described in the system descriptor area,
195.       * determine its data type. Then retrieve the column name and its
196.       * value, storing the value in a host variable defined for the
197.       * particular data type. If the column is not NULL, display the
198.       * name and value.
199.       */
200.    for(i = 1; i <= col_cnt; i++)
201.        {
202.        strcpy(statement, "GET DESCRIPTOR: TYPE field");
203.        EXEC SQL get descriptor :sysdesc VALUE :i
204.            :type = TYPE;
205.        switch(type)
206.            {
207.            case SQLSERIAL:
208.            case SQLINT:
209.                strcpy(statement,
210.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
211.                EXEC SQL get descriptor :sysdesc VALUE :i
212.                    :name = NAME,
213.                    ind = INDICATOR,
```

```
214.                    :int_data = DATA;
215.                if(ind == -1)
216.                    printf("\n%.20s: NULL", name);
217.                else
218.                    printf("\n%.20s: %d", name, int_data);
219.                break;
220.            case SQLSMINT:
221.                strcpy(statement,
222.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
223.                EXEC SQL get descriptor :sysdesc VALUE :i
224.                    :name = NAME,
225.                    :ind = INDICATOR,
226.                    :short_data = DATA;
227.                if(ind == -1)
228.                    printf("\n%.20s: NULL", name);
229.                else
230.                    printf("\n%.20s: %d", name, short_data);
231.                break;
232.            case SQLDECIMAL:
233.            case SQLMONEY:
234.                strcpy(statement,
235.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
236.                EXEC SQL get descriptor :sysdesc VALUE :i
237.                    :name = NAME,
238.                    :ind = INDICATOR,
239.                    :dec_data = DATA;
240.                if(ind == -1)
241.                    printf("\n%.20s: NULL", name);
242.                else
243.                    {
244.                    if(type == SQLDECIMAL)
245.                        rfmtdec(&dec_data, "###,###,###.##", str);
246.                    else
```

```
247.                    rfmtdec(&dec_data, "$$$,$$$,$$$.$$", str);
248.                printf("\n%.20s: %s", name, str);
249.                }
250.            break;
251.        case SQLDATE:
252.            strcpy(statement,
253.                "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
254.            EXEC SQL get descriptor :sysdesc VALUE :i
255.                :name = NAME,
256.                :ind = INDICATOR,
257.                :date_data = DATA;
258.            if(ind == -1)
259.                printf("\n%.20s: NULL", name);
260.            else
261.                {
262.                if((x = rfmtdate(date_data, "mmm. dd, yyyy",
263.                        str)) < 0)
264.                    printf("\ndisp_data() - DATE - fmt error");
265.                else
266.                    printf("\n%.20s: %s", name, str);
267.                }
268.            break;
269.        case SQLDTIME:
270.            strcpy(statement,
271.                "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
272.            EXEC SQL get descriptor :sysdesc VALUE :i
273.                :name = NAME,
274.                :ind = INDICATOR,
275.                :dt_data = DATA;
276.            if(ind == -1)
277.                printf("\n%.20s: NULL", name);
278.            else
279.                {
```

```
                x = dttofmtasc(&dt_data, str, sizeof(str), 0);
                printf("\n%.20s: %s", name, str);
                }
            break;
        case SQLINTERVAL:
            strcpy(statement,
                "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
            EXEC SQL get descriptor :sysdesc VALUE :i
                :name = NAME,
                :ind = INDICATOR,
                :intvl_data = DATA;
            if(ind == -1)
                printf("\n%.20s: NULL", name);
            else
                {
                if((x = intofmtasc(&intvl_data, str,
                        sizeof(str),
                        "%3d days, %2H hours, %2M minutes"))
                        < 0)
                    printf("\nINTRVL - fmt error %d", x);
                else
                    printf("\n%.20s: %s", name, str);
                }
            break;
        case SQLVCHAR:
        case SQLCHAR:
            strcpy(statement,
                "GET DESCRIPTOR: LENGTH, NAME fields");
            EXEC SQL get descriptor :sysdesc VALUE :i
                :char_len = LENGTH,
                :name = NAME;
            amount = char_len;
            if(char_data = (char *)(malloc(amount + 1)))
```

```
313.                    {
314.                    strcpy(statement,
315.                        "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
316.                    EXEC SQL get descriptor :sysdesc VALUE :i
317.                        :char_data = DATA,
318.                        :ind =    INDICATOR;
319.                    if(ind == -1)
320.                         printf("\n%.20s: NULL", name);
321.                    else
322.                        printf("\n%.20s: %s", name, char_data);
323.                    }
324.                else
325.                    {
326.                    printf("\n%.20s: ", name);
327.                    printf("Can't display: out of memory");
328.                    }
329.                break;
330.            case SQLTEXT:
331.                strcpy (statement,
332.                    "GET DESCRIPTOR: NAME, INDICATOR, DATA fields");
333.                EXEC SQL get descriptor :sysdesc VALUE :i
334.                    :name = NAME,
335.                    :ind = INDICATOR,
336.                    :lcat_descr = DATA;
337.                size = lcat_descr.loc_size;       /* get size of data */
338.                printf("\n%.20s: ", name);
339.                if(ind == -1)
340.                    {
341.                    printf("NULL");
342.                    break;
343.                    }
344.                p = lcat_descr.loc_buffer;       /* set p to buf addr */
345.             /* print buffer 80 characters at a time */
```

```
346.                while(size >= 80)
347.                    {
348.                    /* mv from buffer to shdesc */
349.                    ldchar(p, 80, shdesc);
350.                    printf("\n%80s", shdesc);       /* display it */
351.                    size -= 80;             /* decrement length */
352.                    p += 80;            /* bump p by 80 */
353.                    }
354.                strncpy(shdesc, p, size);
355.                shdesc[size] = '\0';
356.                printf("%-s\n", shdesc);       /* dsply last segment */
357.                break;
358.            case SQLBYTES:
359.                strcpy (statement,
360.                    "GET DESCRIPTOR: NAME, INDICATOR fields");
361.                EXEC SQL get descriptor :sysdesc VALUE :i
362.                    :name = NAME,
363.                    :ind = INDICATOR;
364.                if(ind == -1)
365.                    printf("%.20s: NULL", name);
366.                else
367.                    {
368.                    printf("%.20s: ", name);
369.                    printf("Can't display BYTE type value");
370.                    }
371.                break;
372.            default:
373.                printf("\nUnexpected data type: %d", type);
374.                EXEC SQL disconnect current;
375.                printf("\nDYN_SQL Sample Program over.\n\n");
376.                exit(1);
377.            }
378.        }
```

```
379.       printf("\n");
380.}
```

==========

168 - 380 行

disp_data() 函数显示该 SELECT 语句返回的每一行中存储的值。该函数必须能够接收和处理该动态 SELECT 语句的作用域内的任何数据类型（在此情况下，为 **stores7** 数据库内的任何列）。此函数接收两个参数：*col_cnt* 包含那些在系统描述符区域中包含的列的数目，*sysdesc* 包含包含该列信息的系统描述符区域的名称。必须以 PARAMETER 关键字来声明这第二个参数，因为在 FETCH 语句中使用该参数。

disp_data() 函数首先为在 **stores7** 数据库中找到的每一数据类型定义主变量（178 - 188 行），除了已经为 **catalog** 表的 **cat_descr** 和 **cat_picture** 列全局地定义了的定位符结构之外（19 - 22 行）。

对于系统描述符区域中描述的每一列，disp_data() 以 GET DESCRIPTOR 语句检索它的数据类型。接下来，disp_data() 对该数据类型执行 **switch**，对于每一类型（列），它执行另一 GET DESCRIPTOR 语句来检索该列的名称、指示符标志和数据。除非该列为空，否则 disp_data() 将列数据从系统描述符区域的 DATA 字段移至对应的主变量。然后，它显示列名称和主变量的内容。

disp_data() 使用在 sqltypes.h 中定义的符号常量来比较数据类型。它还使用 GBase 8s ESQL/C 库函数 rfmtdec()、rfmtdate()、dttofmtasc() 和 intofmtosc() 来为显示格式化 DECIMAL 和 MONEY、DATE、DATETIME 和 INTERVAL 数据类型。

对于 TEXT 和 BYTE 数据类型，由于数据库服务器返回定位器结构，而不是数据，因此，您可以下列两个阶段的处理，来检索该列的值：

GET DESCRIPTOR 语句（333 和 361 行）从系统描述符区域检索定位符结构，并将它移至 **ifx_loc_t** 主变量。

disp_data() 函数包含来自定位符结构的数据缓冲区的地址，在 **loc_buffer** 中，并从那里检索该数据。

关于 BYTE 数据类型，为了简短起见，disp_data() 检索定位符结构，但

不显示该数据。

```
    ================================================================
======
    381.  free_stuff()
    382.  {
    383.      EXEC SQL free sel_id;       /* free resources for statement */
    384.      EXEC SQL free sel_curs;       /* free resources for cursor */
    385.      /* free system descriptor area */
    386.      EXEC SQL deallocate descriptor 'selcat';
    387.  }
    388.  /*
    389.   * The inpfuncs.c file contains the following functions used in
           * this
    390.   * program:
    391.   *       more_to_do() - asks the user to enter 'y' or 'n' to indicate
    392.   *                              whether to run the main program loop again.
    393.   *
    394.   *       getans(ans, len) - accepts user input, up to 'len' number of
    395.   *                          characters and puts it in 'ans'
    396.   */
    397.  #include "inpfuncs.c"
    398.  /*
    399    * The exp_chk.ec file contains the exception handling functions to
    400.   * check the SQLSTATE status variable to see if an error has
           * occurred
    401.   * following an SQL statement. If a warning or an error has
    402.   * occurred, exp_chk() executes the GET DIAGNOSTICS statement and
    403.   * displays the detail for each exception that is returned.
    404.   */
    405.  EXEC SQL include exp_chk.ec;
    ================================================================
======
```

381 - 387 行

free_stuff() 函数释放为处理该动态语句而分配了的资源。383 行释放由该应用程序在准备动态 SELECT 语句时分配了的资源。384 行释放由数据库服务器处理 **sel_curs** 游标而分配了的资源。DEALLOCATE DESCRIPTOR 语句释放为 **selcat** 系统描述符区域及其相关联的数据区域分配了的内存。

388 - 397 行

几个 GBase 8s ESQL/C 演示程序还调用 more_to_do() 和 getans() 函数。因此，还将这些函数分至单独的 C 源文件内，包含在恰当的演示程序中。 这些函数都不包含 GBase 8s ESQL/C，因此， 该程序可使用 C **#include** 预处理器语句来包括该文件。

398 - 405 行

作为 37 行上 WHENEVER 语句的结果，如果在执行 SQL 语句期间发生错误，则调用 whenexp_chk() 函数。whenexp_chk() 函数检测 SQLSTATE 状态变量来确定 SQL 语句的结果。由于几个演示程序都以 WHENEVER 语句使用此函数来处理异常，因此，已将 whenexp_chk() 函数及其支持函数分至单独的 exp_chk.ec 源文件内。因为该异常处理函数使用 GBase 8s ESQL/C 语句，因此，**dyn_sql** 程序必须以 GBase 8s ESQL/C**include** 伪指令包括此文件。在 异常处理中描述该 exp_chk.ec 源文件。

提示： 在产品环境中，您会将诸如 more_to_do()、getans() 和 whenexp_chk() 这样的函数放至库中，当您编译 GBase 8s ESQL/C 程序时，在命令行上包括它们。

4.4 sqlda 结构

sqlda 结构是动态管理结构，可保存由数据库服务器返回的数据，或由准备好的语句发送至数据库服务器的数据。它是在 sqlda.h 头文件中定义的 C 结构。

重要： **sqlda** 结构不符合 X/Open 标准。它是 GBase 8s 对 GBase 8s ESQL/C 的扩展。

这些主题描述关于如何使用 **sqlda** 结构的下列信息：

使用 **sqlda** 结构来保存未知的值

管理 **sqlda** 结构

使用 **sqlda** 结构来处理动态 SQL 语句中的未知的值

4.4.1 管理 sqlda 结构

您的 GBase 8s ESQL/C 程序可以下表总结的 SQL 语句来操纵 **sqlda** 结构。

表 22. 可用于操纵 sqlda 结构的 SQL 语句

SQL 语句	用途	请参阅
DESCRIBE...INTO	分配 **sqlda** 结构，并以关于列列表列的信息初始化该结构	为 sqlda 结构分配内存 初始化 sqlda 结构

表 23. 可用于操纵 sqlda 结构的 SQL 语句：使用游标的 SELECT 和 EXECUTE FUNCTION 语句

SQL 语句	用途	请参阅
OPEN...USING DESCRIPTOR FETCH...USING DESCRIPTOR	从指定的 **sqlda** 结构取得任何输入参数 将该行的内容放至 **sqlda** 结构内	指定输入参数值 将列值放至 sqlda 结构内

表 24. 可用于操纵 sqlda 结构的 SQL 语句：仅返回一行的 SELECT 和 EXECUTE FUNCTION 语句

SQL 语句	用途	请参阅
EXECUTE...INTO DESCRIPTOR	将该单个行的内容放至 **sqlda** 结构内	将列值放至 sqlda 结构内

表 25. 可用于操纵 sqlda 结构的 SQL 语句：非 SELECT 语句

SQL 语句	用途	请参阅
EXECUTE...USING DESCRIPTOR	从指定的 **sqlda** 结构取得任何输入信息	指定输入参数值

表 26. 可用于操纵 sqlda 结构的 SQL 语句：使用插入游标的 INSERT 语句

SQL 语句	用途	请参阅
PUT...USING DESCRIPTOR	在它从指定的 **sqlda** 结构获得列值之后，将一行放至插入缓冲区内	处理未知的列列表

此外，您的 GBase 8s ESQL/C 程序可以下列方式管理 **sqlda** 结构：

声明一个指向**sqlda** 结构的变量指针。

将值指定给 **sqlda** 字段来以丢失的列信息提供数据库服务器。

从 **sqlda** 字段取得信息来访问从数据库服务器接收的列信息。

释放分配给 **sqlda** 结构的内存，当以它停止您的程序时。

定义 sqlda 结构

GBase 8s ESQL/Csqlda.h 头文件定义 sqlda 结构。

要定义 **sqlda** 结构，GBase 8s ESQL/C 程序必须采取下列行动：

包括 sqlda.h 头文件来在您的程序中为 sqlda 提供声明

GBase 8s ESQL/C 预处理器自动地包括 sqlhdr.h 文件，其包括 sqlda.h 头文件。

声明变量名称作为指向 sqlda 结构的指针

下列代码行声明 **da_ptr** 变量作为 **sqlda** 指针：

```
struct sqlda *da_ptr;
```

重要： 指向 **sqlda** 结构的指针不是 GBase 8s ESQL/C 主变量。因此，您不需要以关键字 EXEC SQL 或美元（$）符号先于该语句声明。此外，在该程序会中，您不以冒号（:）或美元（$）符号先于任何对指针的引用。

为 sqlda 结构分配内存

在您定义主变量作为指向 **sqlda** 结构的指针之后，您必须确保为此结构的所有部分分配内存，如下：

要为 **sqlda** 结构自身分配内存，请使用 DESCRIBE...INTO 语句。

下列 DESCRIBE 语句获得关于准备好的语句 **st_id** 的信息，为 **sqlda** 结构分配内存，并将 **sqlda** 结构的地址放至指针 **da_ptr** 中：

```
EXEC SQL describe st_id into da_ptr;
```

要为 sqlvar_struct 结构分配内存，请采取下列行动：

如果该准备好的语句是 SELECT（不带有 INTO TEMP 子句）、INSERT 或 EXECUTE FUNCTION 语句，则 DESCRIBE...INTO 语句可为 sqlvar_struct 结构分配空间。

如果准备了某些其他 SQL 语句，且您想要在数据库服务器中发送或接收列，则您的程序必须为 sqlvar_struct 结构分配空间。

要为 sqldata 字段的数据分配内存，请确保该数据类型与正确的词边界一致。

如果您使用 **sqlda** 结构来定义输入参数，则您不可使用 DESCRIBE 语句。因此，您的程序必须显式地同时为 **sqlda** 结构和 **sqlvar_struct** 结构分配内存。

初始化 sqlda 结构

要发送或接收数据库中的列值， 您的 GBase 8s ESQL/C 程序必须初始化 **sqlda** 结构，以便于它描述该准备好的语句的未知的列。

要初始化 **sqlda** 结构，您必须执行下列步骤：

将 sqlvar 字段设置为初始化的 sqlvar_struct 结构的地址。

设置 sqld 字段来指示未知的列数（以及相关联的 sqlvar_struct 结构）。

除了为 sqlda 结构分配内存，DESCRIBE...INTO 语句还以关于该准备好的语句的信息初始化此结构。DESCRIBE...INTO 可提供的信息，依赖于它已描述了哪个 SQL 语句。

如果准备好的语句是 SELECT（不带有 INTO TEMP 子句）、INSERT 或 EXECUTE FUNCTION 语句，则 DESCRIBE...INTO 语句可确定关于列列表中列的信息。因此，DESCRIBE...INTO 语句采取下列行动来初始化 **sqlda** 结构：

它为 sqlda 结构分配内存。

它设置 sqlda.sqld 字段，其包含以数据初始化了的 sqlvar_struct 结构的数目。此值为列列表中的列和表达式的数目（SELECT 和 INSERT），或返回的值的数目（EXECUTE FUNCTION）。

它为组件 sqlvar_struct 结构分配内存，对于列列表中的每一列或表达式（SELECT 和 INSERT），或对于每一返回的值（EXECUTE FUNCTION），对应一个 sqlvar_struct 结构。

它将 sqlda.sqlvar 字段设置为 DESCRIBE 为该 sqlvar_struct 结构分配的内存的初始地址。

它描述在准备好的 SELECT（不带有 INTO TEMP）、EXECUTE FUNCTION 或 INSERT 语句中每一未知的列。DESCRIBE...INTO 语句为每一列初始化 sqlvar_struct 结构的字段，如下：

它初始化 sqltype、sqllen 和 sqlname 字段（对于 CHAR 类型数据，或对于

DATETIME 或 INTERVAL 数据的限定符），来提供关于该列的来自数据库的信息。

对于大多数数据类型，**sqllen** 字段保存以字节计的该列的长度。如果该列为集合类型（SET、MULTISET 或 LIST）、row 类型（命名的或未命名的）或 opaque 类型，则 **sqllen** 字段为零。

它将 sqldata 和 sqlind 字段初始化为空。

重要： 不像系统描述符区域那样，带有 **sqlda** 指针的 DESCRIBE 不为列数据（**sqldata** 字段）分配内存。在您的程序从数据库服务器收到列值之前，它必须分配此数据空间。

DESCRIBE 语句提供关于列列表的列的信息。因此，您通常在准备好了 SELECT（不带有 INTO TEMP 子句）、INSERT 或 EXECUTE FUNCTION 语句*之后*，使用 DESCRIBE...INTO。DESCRIBE...INTO 语句不仅初始化 sqlda 结构，而且返回准备好的 SQL 语句的类型。

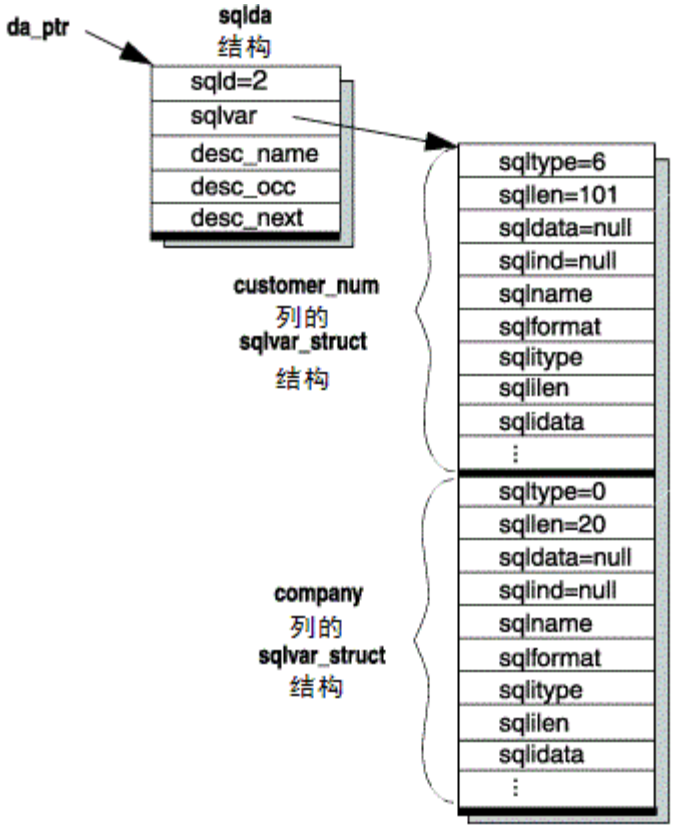
下列 DESCRIBE 语句还为一个 **sqlda** 结构和为两个 **sqlvar_struct** 数据结构（一个为 **customer_num** 列，另一个为 **company** 列）分配内存，然后，以分配给 **sqlvar_struct** 结构的内存的初始地址来初始化指针 **da_ptr->sqlvar**：

```
EXEC SQL prepare st_id
        'select customer_num, company from customer
        where customer_num = ?';
        EXEC SQL describe st_id into da_ptr;
```

前面的 DESCRIBE...INTO 语句返回 SQLCODE 值 0，来指示该准备好的语句为 SELECT 语句。

下图展示此 DESCRIBE...INTO 语句可以初始化的样例 sqlda 结构。

图 5. 两列的样例 sqlda 结构



如果准备好了某其他 SQL 语句，则该 DESCRIBE...INTO 语句不可初始化 **sqlda** 结构。要发送或接收该数据库中的列值，您的程序必须显式地执行此初始化，如下：

为组件 sqlvar_struct 结构分配内存，每一列一个 sqlvar_struct 结构。

您可使用诸如 malloc() 或 calloc() 这样的系统内存分配函数，并指定地址为 **sqlvar**，如下：

```
da_ptr->sqlvar = (struct sqlvar_struct *)
                    calloc(count, sizeof(struct sqlvar_struct));
```

执行下列任务来描述每一未知的列：

设置 sqlda.sqld 字段，其包含以数据初始化了的 sqlvar_struct 结构的数目。此值为准备好的语句中未知的列的数目。

初始化每一 sqlvar_struct 结构的字段。

设置 **sqltype**、**sqllen** 和 **sqlname** 字段（对于 CHAR 类型数据，或对于 DATETIME 或 INTERVAL 数据的限定符），来将关于列的信息提供给数据库服务器。

要提供列数据，您的程序还必须为此数据分配空间，并将每一

sqlvar_struct 结构的 sqldata 字段设置为此空间内的恰当位置。如果您将列数据发送给数据库服务器，则请务必正确地设置 sqlind 字段。

如果您使用 sqlda 结构来定义输入参数，则您不可使用 DESCRIBE 语句来初始化 sqlda 结构。您的代码必须显式地设置 sqlda 结构的恰当的字段，来定义输入参数。

为列数据分配内存

sqlda 结构为 **sqlvar_struct** 结构的 **sqldata** 字段中的每一列存储一个指向该数据的指针。不像 DESCRIBE...USING SQL DESCRIPTOR 语句那样，DESCRIBE...INTO 语句不为此数据分配内存。当 DESCRIBE...INTO 语句为 **sqlda** 指针分配内存时，它将每一 **sqlvar_struct** 结构的 **sqldata** 字段初始化为空。

要发送或接收数据库中的列数据，您的 GBase 8s ESQL/C 程序必须执行下列任务：

为该列数据分配内存。

将与该列相关联的 sqlvar_struct 结构的 sqldata 字段，设置为为该列数据分配的内存的地址。

要为 sqldata 字段分配内存，您可使用系统内存分配函数，比如 malloc() 或 calloc()。作为对 malloc() 系统内存分配函数的替代，您可程序可为该数据缓冲区声明静态的字符缓冲区。下图展示从名为 data_buff 的静态字符缓冲区分配列数据的代码片段。

图 6. 从静态的字符缓冲区分配列数据

```
static char data_buff[1024];
        struct sqlda *sql_descp;
        struct sqlvar_struct * col_ptr;
        short cnt, pos;
        int size;
        ⋮

        for(col_ptr=sql_descp->sqlvar, cnt=pos=0; cnt < sql_descp->sqld;
        cnt++, col_ptr++)
        {
        pos = (short)rtypalign(pos, col_ptr->sqltype);
        col_ptr->sqldata = &data_buf[pos];
```

```
size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
pos += size;
}
```

您可在 for 循环内以一系列系统内存分配调用来替代 图 1中的代码片段。然而，系统内存分配调用可能代价高昂，因此，更加高效的方式，通常是分配单个内存，然后将指针对准至该内存区域内。

当您分配列数据时，请务必为列数据类型格式化分配了的内存。此数据类型为 GBase 8s ESQL/C 或定义在 sqltypes.h 头文件中的 SQL 数据类型之一。请使得分配了的内存足够大，以容纳该列中数据的最大大小。

您还必须确保每一列的数据开始于内存中正确的词边界上。在许多硬件平台上，整数和其他数值数据类型必须开始于词边界上。C 语言内存分配例程分配与包括结构的任何数据类型恰当一致的内存，但该例程不对该结构的组成部件执行校准。

使用正确的词边界，确保数据类型与机器无关。为了在此任务中辅助您， GBase 8s ESQL/C 提供下列内存管理函数：

对于指定的数据类型，rtypalign() 函数返回下一个正确的词边界的位置。

此函数接受两个参数：在数据缓冲区中的当前位置，以及整数 GBase 8s ESQL/C 或您想要为其分配空间的 SQL 数据类型。

rtypmsize() 函数返回内存的字节数，您必须为指定的 GBase 8s ESQL/C 或 SQL 数据类型分配该内存。

此函数接受两个参数：整数 GBase 8s ESQL/C 或对于每一列值的 SQL 数据类型（在 **sqltype** 中）和长度（在 **sqllen** 中）。

当您为 DATETIME 或 INTERVAL 数据类型分配内存时，您可采取下列行动，来设置 **dtime_t** 和 **intrvl_t** 结构中的限定符：

请使用在相关联的 sqlda 的 sqllen 字段中的值。

以该值与 datatime.h 头文件定义的宏组成一个不同的限定符。

将数据类型限定符设置为 0，并使得数据库服务器在访存期间设置此限定符。对于 DATETIME 值，该数据类型限定符为 dtime_t 结构的 dt_qual 字段。对于 INTERVAL 值，该数据类型限定符为 intrvl_t 结构的 in_qual 字段。

要获取为 **sqldata** 字段分配内存的示例，请参阅 demo3.ec 和 unload.ec 演示程序，

以 GBase 8s ESQL/C 提供这些程序。

指定并从 sqlda 结构取得值

当您以动态 SQL 使用 sqlda 结构时，您必须以 C 语言语句将信息移入和移除它。

指定值

要将值指定给 **sqlda** 和 **sqlvar_struct** 结构中的字段，请使用常规的 C 语言赋值给恰当的结构的字段。例如：

```
da_ptr->sqld = 1;
        da_ptr->sqlvar[0].sqldata = compny_data;
        da_ptr->sqlvar[0].sqltype = SQLCHAR;     /* CHAR data type */
        da_ptr->sqlvar[0].sqllen = 21;                /* column is CHAR(20) */
```

设置 sqlda 字段来为 WHERE 子句中的输入参数提供值（指定输入参数值），或在您使用 DESCRIBE...INTO 语句来填充 sqlda 结构之后来修改字段的内容（为列数据分配内存）。

取得值

要从 **sqlda** 字段取得值，您还必须使用来自该结构的字段的常规的 C 语言赋值。例如：

```
count = da_ptr->sqld;
        /* Allow for the trailing null character in C character arrays */
        if (da_ptr->sqlvar[0].sqltype == SQLCHAR)
        a_ptr->sqlvar[0].sqllen += 1;
        /* Allocate a separate buffer per column */
        da_ptr->sqlvar[0].sqldata = malloc(col_ptr->sqllen);
```

通常，您取得 sqlda 字段值来检测 SELECT、INSERT 或 EXECUTE FUNCTION 语句中列的描述。您还可能需要访问这些列，来将由数据库服务器返回的列值从 sqlda 结构复制至主变量（将列值放至 sqlda 结构内）。在后者的情况下，您可能需要更改 sqllen 来说明正确的缓冲区长度。例如，您必须对 CHAR 数据类型增加 sqllen。如果您未增加 sqllen，则会截断访存到的数据的最后一个字符，因为 FETCH 会假设 sqllen 值为该缓冲区的大小，并会将缓冲区中的最后位置用作零来终止该字符串。

主变量的数据类型必须与 sqlda 结构中相关联的字段的类型相兼容。当您解释 sqltype 字段时，请确保您使用匹配您的环境的数据类型值。对于某些数据类型，X/Open 值不同于 GBase 8s 值。

指定输入参数值

由于 DESCRIBE...INTO 语句不分析 WHERE 子句，因此，您的程序必须显式地分配 sqlda 结构和 sqlvar_struct 结构。要描述输入参数，您必须确定输入参数的数目以及它们的数据类型，并将此信息存储在分配了的 sqlda 结构中。

当您执行参数化的语句时，您必须包括 USING DESCRIPTOR 子句来指定 **sqlda** 结构作为输入参数值的位置，如下：

对于 SELECT 语句的 WHERE 子句的输入参数，请使用 OPEN...USING DESCRIPTOR 语句。此语句处理顺序的、滚动的、保存或更新游标。如果您确定该 SELECT 仅返回一行，则可使用 EXECUTE...INTO...USING SQL DESCRIPTOR 语句，而不使用游标。

对于非 SELECT 语句的 WHERE 子句中的输入参数，诸如 DELETE 或 UPDATE，请使用 EXECUTE...USING DESCRIPTOR 语句。

对于 INSERT 语句的 VALUES 子句中的输入参数，请使用 EXECUTE...USING SQL DESCRIPTOR 语句。如果该 INSERT 与插入游标相关联，则请使用 PUT...USING DESCRIPTOR 语句。

将列值放至 sqlda 结构内

当您动态地创建 SELECT 语句时，您不可使用 FETCH 的 INTO *host_var* 子句，因为您不可在准备好的语句中命名主变量。要将列值访存至 **sqlda** 结构内，请使用 FETCH 的 USING DESCRIPTOR 子句，而不用 INTO 子句。FETCH...USING DESCRIPTOR 语句将每一列值放至它的 **sqlvar_struct** 结构的 **sqldata** 字段内。

使用 FETCH...USING DESCRIPTOR 语句，假设游标与准备好的语句相关联。您必须总是为 SELECT 语句和游标函数（返回多行的 EXECUTE FUNCTION 语句）使用游标。然而，如果这些语句之一仅返回一行，则您可省略游标，并以 EXECUTE...INTO DESCRIPTOR 语句将列值检索至 **sqlda** 结构内。

重要： 如果您执行返回多行的 SELECT 语句或用户定义的函数，且未将该语句与游标相关联，则您的程序生成运行时刻错误。当您将单个 SELECT（或 EXECUTE FUNCTION）语句与游标相关联时，GBase 8s ESQL/C 不生成错误。因此，总是将动态 SELECT 或 EXECUTE FUNCTION 语句与游标相关联，并使用 FETCH...USING DESCRIPTOR 语句来将列值从此游标检索至 **sqlda** 结构，是一种好的做法。

一旦列值在 **sqlda** 结构中，您就可将这些值从 **sqldata** 移至恰当的主变量。在运行时刻，您必须使用 **sqllen** 和 **sqltype** 字段来确定该主变量的数据类型。您可能需要在 **sqltype** 字段中的数据类型与保存返回的值的主变量的 GBase 8s ESQL/C 数据类型之间执行数据类型或长度转换。

释放分配给 sqlda 结构的内存

一旦您以 **sqlda** 结构结束，就请释放相关联的内存。如果您执行多个 DESCRIBE 语句，且忽略由这些语句分配的内存，则您的应用程序可能冲击内存限制，且数据库服务器可能退出。

如果在 Windows(TM) 操作系统上运行您的应用程序，且使用多线程库，则请使用 GBase 8s ESQL/C 函数 SqlFreeMem() 来释放 **sqlda** 结构所占的内存，如此示例所示：

SqlFreeMem(*sqlda_ptr*, SQLDA_FREE);

否则，请使用标准 C 库 free() 函数，如此示例所示：

free(sqlda_ptr);

对于同一准备好的语句，如果您的 GBase 8s ESQL/C 程序执行 DESCRIBE 语句多次，并为每一 DESCRIBE 分配单独的 sqlda 结构，则它必须显式地释放每一 sqlda 结构。下图展示一个示例。

图 7. 为同一准备好的语句释放多个 sqlda 结构

```
EXEC SQL prepare qid from 'select * from customer';
        EXEC SQL describe qid into sqldaptr1;
        EXEC SQL describe qid into sqldaptr2;
        EXEC SQL describe qid into sqldaptr3;
        ⋮
        1
        free(sqldaptr1);
        free(sqldaptr2);
        free(sqldaptr3);
```

如果你的程序为列数据分配了空间，则您还必须释放分配给 **sqldata** 字段的内存。

4.4.2 执行 SQL 语句的 sqlda 结构

使用 SQL 描述符区域（sqlda）结构来执行包含未知的值的 SQL 语句。

下表总结在本部分中涵盖的动态语句的类型。

表 27. 使用 sqlda 结构来执行动态 SQL 语句

sqlda 结构的用途	请参阅
保存由 SELECT 检索的选择类表列值	处理未知的选择列表
保存来自用户定义的函数的返回的值	处理未知的返回值
描述 INSERT 中的未知的列	处理未知的列列表
描述 SELECT 的 WHERE 子句中的输入参数	处理参数化的 SELECT 语句
描述 DELETE 或 UPDATE 的 WHERE 子句中的输入参数	处理参数化的 UPDATE 或 DELETE 语句

4.4.3 处理未知的选择列表

本部分描述如何使用 sqlda 结构来处理 SELECT 语句。

要使用 **sqlda** 结构来处理未知的选择列表列：

1. 声明变量来保存 sqlda 结构的地址。
2. （以 PREPARE 语句）准备 SELECT 语句，来给它一个语句标识符。该 SELECT 语句不可包括 INTO TEMP 子句。
3. 使用 DESCRIBE...INTO 语句来执行两项任务：
   a) 分配 sqlda 结构。将分配了的结构的地址存储在您声明的 sqlda 指针中。
   b) 确定选择列表列的数目和数据类型。DESCRIBE 语句为选择列表的每一列填充 sqlvar_struct 结构。
4. 对于每一选择列表列检测 sqlda 的 sqltype 和 sqllen 字段，来确定需要为该数目分配的内存量。
5. 保存存储在主变量中的 sqld 字段中的选择列表列的数目。
6. 声明并打开游标，然后，使用 FETCH...USING DESCRIPTOR 语句来将列值访存至分配了的 sqlda 结构内，一次一行。
7. 将行输入从 sqlda 结构检索至带有 C 语言语句的主变量内，其访问每一选择列表列的 sqldata 字段。

8. 释放分配给 sqldata 字段和 sqlda 结构的内存。

重要： 如果 SELECT 语句在 WHERE 子句中有未知数目和类型的输入参数，则您的程序还必须以 **sqlda** 结构来处理这些输入参数。

执行返回多行的 SELECT

demo3.ec 样例程序以下列条件执行动态 SELECT 语句：

该 SELECT 返回多行。

该 SELECT 必须与游标相关联，以 OPEN 语句执行，并以 FETCH...USING DESCRIPTOR 语句检索它的检索的值。

该 SELECT 或者没有输入参数，或者没有 WHERE 子句。

OPEN 语句不需要包括 USING 子句。

该 SELECT 在它的选择列表中有未知的列。

该 FETCH 语句包括 USING DESCRIPTOR 子句来将返回值存储在 **sqlda** 结构中。

demo3.ec 样例程序

demo4 样例程序（demo4.ec 样例程序）假设这些相同的条件，在 demo4 使用系统描述符区域来定义选择列表列时，demo3 使用 sqlda 结构。demo3 程序不执行异常处理。

```
================================================================================
```

```
1. #include <stdio.h>
2. EXEC SQL include sqlda;
3. EXEC SQL include sqltypes;
4. main()
5. {
6. struct sqlda *demo3_ptr;
7. struct sqlvar_struct *col_ptr;
8. static char data_buff[1024];
9. int pos, cnt, size;
10. EXEC SQL BEGIN DECLARE SECTION;
11.     int2 i, desc_count;
12.     char demoquery[80];
13. EXEC SQL END DECLARE SECTION;
```

```
14.     printf("DEMO3 Sample ESQL program running.\n\n");
15.     EXEC SQL connect to 'stores7';
```

==========

2 行

该程序必须包括 GBase 8s ESQL/Csqlda.h 头文件来提供对 **sqlda** 结构的定义。

6 - 13 行

6 和 7 行声明该程序需要的 **sqlda** 变量。**demo3_ptr** 变量指向将保存从数据库访存的数据的 **sqlda** 结构。**col_ptr** 变量指向 **sqlvar_struct** 结构，以便于该代码可逐步经过 **sqlda** 的变长部分中的每一 **sqlvar_struct** 结构。这些变量都不声明作为 GBase 8s ESQL/C 主变量。10 - 13 行声明主变量来保存从用户取得的数据，以及从 **sqlda** 结构检索的数据。

==========

```
16.     /* These next four lines have hard-wired both the query and
17.      * the value for the parameter. This information could have
18.      * been entered from the terminal and placed into the strings
19.      * demoquery and a query value string (queryvalue), respectively.
20.      */
21.     sprintf(demoquery, "%s %s",
22.         "select fname, lname from customer",
23.         "where lname < 'C' ");
24.     EXEC SQL prepare demo3id from :demoquery;
25.     EXEC SQL declare demo3cursor cursor for demo3id;
26.     EXEC SQL describe demo3id into demo3_ptr;
```

==========

16 - 24 行

这些行为 SELECT 语句组装字符串（在 **demoquery** 中），并准备它作为 **demo3id** 语句标识符。

25 行

此行为准备好的语句标识符 **demo3id** 声明 **demo3cursor**。

26 行

该 DESCRIBE 语句为 **demo3id** 语句标识符内的准备好的语句描述选择列表列。为此，您必须在使用 DESCRIBE 之前准备该语句。此 DESCRIBE 包括 INTO 子句来执行 **sqlda** 结构，**demo3_ptr** 指向其作为这些列描述的位置。DESCRIBE...INTO 语句还为 **sqlda** 结构分配内存，并将此结构的地址存储在 **demo3_ptr** 变量中。

demo3 程序假设在运行时刻组装下列 SELECT 语句，并存储在 **demoquery** 字符串中：

SELECT fname, lname FROM customer WHERE lname < 'C'

在 26 行中的 DESCRIBE 语句之后，**sqlda** 结构的组件包含如下：

sqlda 组件 demo3_ptr->sqld 有值 2，因为从 customer 表选择了两列。

组件 demo3_ptr->sqlvar[0]，包含关于 customer 表的 fname 列信息的 sqlvar_struct 结构。例如，demo3_ptr->sqlvar[0].sqlname 组件给出第一列的名称（fname）。

组件 demo3_ptr->sqlvar[1]，包含关于 customer 表的 lname 列的信息的 sqlvar_struct 结构。

==========

```
27.     desc_count = demo3_ptr->sqld;
28.     printf("There are %d returned columns:\n", desc_count);
29.     /* Print out what DESCRIBE returns */
30.     for (i = 1; i <= desc_count; i++)
31.         prsqlda(i, demo3_ptr->sqlvar[i-1]);
32.     printf("\n\n");
```

==========

27 和 28 行

27 行将由 DESCRIBE 语句找到的选择列表列的数目指定给 **desc_count** 主变量。28 行将此信息显示给用户。

29 - 32 行

此 **for** 循环仔细检查该选择列表的列的 **sqlvar_struct** 结构。它使用 **desc_count** 主变量来确定由 DESCRIBE 初始化的这些结构的数目。对于每一 **sqlvar_struct** 结构，prsqlda() 函数（31 行）显示诸如数据类型、长度和名称这样的信息。要了解 prsqlda() 的描述，请参阅 75 - 81 行的描述。

==========

```
33.     for(col_ptr=demo3_ptr->sqlvar, cnt=pos=0; cnt < desc_count;
34.         cnt++, col_ptr++)
35.         {
36.         /* Allow for the trailing null character in C
37.                 character arrays */
38.         if(col_ptr->sqltype==SQLCHAR)
39.             col_ptr->sqllen += 1;
40.         /* Get next word boundary for column data and
41.                 assign buffer position to sqldata */
42.         pos = (int)rtypalign(pos, col_ptr->sqltype);
43.         col_ptr->sqldata = &data_buff[pos];
44.         /* Determine size used by column data and increment
45.                 buffer position */
46.         size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
47.         pos += size;
48.         }
```

==========

33 - 48 行

这第二个 **for** 循环为 **sqldata** 字段分配内存，并将设置 **sqldata** 字段来指向此内存。

对于每一选择列表列，40 - 47 行检测 **sqlda** 的 **sqltype** 和 **sqllen** 字段，来确定您需要为该数据分配的内存的数量。该程序不使用 malloc() 来动态地分配空间。反而，它使用静态数据缓冲区（8 行上定义的 **data_buff** 变量）来保存该列数据。对于列数据类型，GBase 8s ESQL/Crtypalign() 函数（42 行）返回下一词边界的位置（在 **col_ptr->sqltype** 中）。然后，43 行将 **data_buff** 数据缓冲区内此位置的地址指定给 **sqldata** 字段（对于接收由该查询返回的值的列）。

GBase 8s ESQL/Crtypmsize() 函数（46 行）返回由 **sqltype** 和 **sqllen** 字段指定的 SQL 数据类型所需要的字节数。然后，47 行按该数据需要的大小增加该数据缓冲区指针（**pos**）。

==========

```
49.     EXEC SQL open demo3cursor;
50.     for (;;)
51.         {
52.         EXEC SQL fetch demo3cursor using descriptor demo3_ptr;
53.         if (strncmp(SQLSTATE, "00", 2) != 0)
54.             break;
55.         /* Print out the returned values */
56.         for (i=0; i<desc_count; i++)
57.             printf("Column: %s\tValue:%s\n", demo3_ptr-
                >sqlvar[i].sqlname,
58.                 demo3_ptr->sqlvar[i].sqldata);
59.         printf("\n");
60.         }
```

==========

49 行

当数据库服务器打开 demo3cursor 游标时，它执行 SELECT 语句。如果您的 SELECT 语句的 WHERE 子句包含输入参数，则您还需要指定 OPEN 的 USING DESCRIPTOR 子句。

50 - 60 行

对于从数据库访存的每一行，执行此内层的 **for** 循环。FETCH 语句（52 行）包括 USING DESCRIPTOR 子句来指定 **demo3_ptr** 指向的 **sqlda** 结构作为列值的位置。在此 FETCH 之后，将列值存储在指定的 **sqlda** 结构中。

if 语句（53 和 54 行）检测 SQLSTATE 变量的值来确定该 FETCH 是否成功。如果 SQLSTATE 指示任何不成功的状态，则执行 54 行并终止该 **for** 循环。对于选择列表的每一列，56 - 60 行显示 **sqlname** 和 **sqldata** 字段的内容。

重要： **demo3** 程序假设返回的列为字符数据类型。如果该程序未进行此假设，则它会需要检查 **sqltype** 和 **sqllen** 字段，来确定保存 **sqldata** 值的主变量的恰当的数据类型。

```
================================================================
======
61.     if (strncmp(SQLSTATE, "02", 2) != 0)
62.         printf("SQLSTATE after fetch is %s\n", SQLSTATE);
63.     EXEC SQL close demo3cursor;
================================================================
======
```

61 和 62 行

在该 **for** 循环之外，程序再次检测 SQLSTATE 变量，如果执行成功、发生运行时刻错误或警告（类代码不等于 "02"），以便于它可通知用户。

63 行

在访存所有行之后，CLOSE 语句关闭 **demo3cursor** 游标。

```
=================================================================
======
64.     EXEC SQL free demo3id;
65.     EXEC SQL free demo3cursor;
66.     /* No need to explicitly free data buffer in this case because
67.      * it wasn't allocated with malloc(). Instead, it is a static char
68.      * buffer
69.      */
70.     /* Free memory assigned to sqlda pointer. */
71.     free(demo3_ptr);
72.     EXEC SQL disconnect current;
73.     printf("\nDEMO3 Sample Program Over.\n\n");
74. }
75. prsqlda(index, sp)
76. int2 index;
77. register struct sqlvar_struct *sp;
78. {
79.     printf("      Column %d: type = %d, len = %d, data = %s\n",
80.         index, sp->sqltype,   sp->sqllen, sp->sqldata, sp->sqlname);
81. }
=================================================================
======
```

64 和 65 行

这些 FREE 语句释放为 **demo3id** 准备好的语句和 **demo3cursor** 数据库游标分配的资源。

66 - 71 行

在该程序的结尾，释放分配给 sqlda 结构的内存。由于此程序不使用 malloc() 来分配数据缓冲区，因此，它不使用 free() 系统调用来释放 sqldata 指针。虽然从静态缓冲区分配内存是简单的，但它有缺点：保持分配此缓冲区，直到

程序结束为止。

free() 系统调用（71 行）释放 **demo3_ptr** 指向的 **sqlda** 结构。

75 - 81 行

prsqlda() 函数显示关于选择列表列的信息。它从 **sqlvar_struct** 结构读取此信息，将其地址传至该函数内（**sp**）。

提示： GBase 8s ESQL/C 演示程序 unload.ec 和 dyn_sql.ec（在 dyn_sql 程序中描述）还使用 sqlda 来描述选择列表的列。

执行单个 SELECT

demo3 程序假设 SELECT 语句返回多行，因此，该程序与游标相关联。如果在此时您知道编写的是动态 SELECT 总是只返回一行的程序，则您可省略游标并使用 EXECUTE...INTO DESCRIPTOR 语句，而不用 FETCH...USING DESCRIPTOR。您必须仍使用 DESCRIBE 语句来定义选择列表列。

4.4.4 处理未知的返回值

您可使用 sqlda 结构来保存动态地执行的用户定义的函数返回的值。

要使用 **sqlda** 结构来处理位置的函数返回值：

声明变量来保存 **sqlda** 结构的地址。

组装并准备 EXECUTE FUNCTION 语句。

EXECUTE FUNCTION 语句不可包含 INTO 子句。

使用 DESCRIBE...INTO 语句来执行两项任务：

a) 分配 sqlda 结构。将分配的结构的地址存储在您声明的 sqlda 指针中。
b) 确定函数返回值的数目和数据类型。该 DESCRIBE 语句为每一返回值填充 sqlvar_struct 结构。

在 DESCRIBE 语句之后，您可为定义的常量 SQ_EXECPROC 测试 SQLCODE 变量（sqlca.sqlcode），来检查准备好的 EXECUTE FUNCTION 语句。

在 sqlstype.h 头文件中定义 SQ_EXECPROC 常量。

对于每一返回值，测试 sqlda 的 sqltype 和 sqllen 字段，来确定需要为该数据分配的内存的数量。

执行 EXECUTE FUNCTION 语句，并将返回值存储在 sqlda 结构中。

您用来执行用户定义的函数的语句，依赖于该函数是非游标函数，还是游标函数。

释放您分配给 sqlda 结构的内存。

执行非游标函数

非游标函数仅返回一行给应用程序。请使用 EXECUTE...INTO DESCRIPTOR 语句来执行该函数，并将返回的一个值或多个值保存在 **sqlda** 结构中。

未显式地定义作为迭代函数的外部函数仅返回单行数据。因此，您可使用 EXECUTE...INTO DESCRIPTOR 来动态地执行最外部的函数，并将它们的返回值保存至 **sqlda** 结构内。此单行数据仅由一个值组成，因为外部函数仅可返回单个值。**sqlda** 结构仅包含带有单个返回值的一个项描述符。

其 RETURN 语句不包括 WITH RESUME 关键字的 SPL 函数仅返回单行数据。因此，您可使用 EXECUTE...INTO DESCRIPTOR 来动态地执行大多数 SPL 函数，并将它们的返回值保存至 **sqlda** 结构内。SPL 函数一次可返回一个或多个值，因此，**sqlda** 结构包含一个或多个项描述符。

重要： 由于您通常不知道用户定义的函数返回的返回行数，因此，您不可保证仅返回一行。如果您不使用游标来执行游标函数，则 GBase 8s ESQL/C 生成运行时刻错误。因此，总是将用户定义的函数与函数游标相关联，是一种好的做法。

执行游标函数

游标函数可将一个或多个返回值的行返回给应用程序。要执行游标函数，您必须将 EXECUTE FUNCTION 语句与函数游标相关联，并使用 FETCH...INTO DESCRIPTOR 语句来将一个返回值或多个返回值保存在 sqlda 结构中。

要使用 **sqlda** 结构来保存游标函数返回值：

为用户定义的函数声明函数游标。

使用 DECLARE 语句来将 EXECUTE FUNCTION 语句与函数游标相关联。

使用 OPEN 语句来执行该函数并打开该游标。

使用 FETCH...USING DESCRIPTOR 语句来将返回值从游标检索至 **sqlda** 结构内。

将行数据从 **sqlda** 结构检索至带有 C 语言语句的主变量，其访问每一选择列表列的 **sqldata** 字段。

释放分配给 **sqldata** 字段和 **sqlda** 结构的内存。

仅定义作为迭代函数的外部函数可返回多行数据。因此，您必须定义函数游标来动态地执行迭代函数。每一行数据仅由一个值组成，因为外部函数仅可返回单个值。对于每一行，**sqlda** 结构仅包含带有单个返回值的一个 **sqlvar_struct** 结构。

其 RETURN 语句包括 WITH RESUME 关键字的 SPL 函数可返回一行或多行数据。因此，您必须定义函数游标来动态地执行这些 SPL 函数。每一行数据可由一个或多个值组成，因为 SPL 函数可一次返回一个或多个值。对于每一行，**sqlda** 结构包含每一返回值的 **sqlvar_struct** 结构。

4.4.5 处理未知的列列表

您可使用 sqlda 结构来处理 INSERT...VALUES 语句。

要使用 **sqlda** 结构来处理 INSERT 中的输入参数：

1. 声明变量来保存 sqlda 的地址。
2. 准备 INSERT 语句（以 PREPARE 语句），并给它一个语句标识符。

使用 DESCRIBE...INTO 语句来执行两项任务：

a) 分配 sqlda 结构。将分配了的结构的地址存储在您声明的sqlda 指针中。
b) 以 DESCRIBE...INTO 语句来确定表中列的数目和数据类型。DESCRIBE 语句为列列表的每一项填充 sqlvar_struct 结构。

3. 对于每一列，检测 sqlda 的 sqltype 和 sqllen 字段，来确定需要为该数据分配的内存的数量。

保存存储在主变量中的 **sqld** 字段中的列的数目。

以 C 语言语句将列设置为它们的值，其设置 **sqlda** 的 **sqlvar_struct** 结构中的恰当的 **sqldata** 字段。列值必须与它的相关联的列的数据类型相兼容。如果您插入空值，请务必将恰当的 **sqlind** 字段设置为包含 -1 的指示符变量的地址。

执行 INSERT 语句来将该值插入至数据库内。

4. 释放分配给 sqldata 字段和 sqlda 结构的内存。

执行简单的插入

下列步骤该如如何以 **sqlda** 结构执行简单的 INSERT 语句：

（以 PREPARE 语句）准备 INSERT 语句，并给它一个语句标识符。

以 C 语言语句将这些列设置为它们的值，其设置 **sqlda** 的 **sqlvar_struct** 结构中恰当的 **sqldata** 字段。

以 EXECUTE...USING DESCRIPTOR 语句执行 INSERT 语句。

这些步骤基本上与处理 SELECT 语句的未知的选择列表的那些步骤相同。由于该语句不是 SELECT 语句，因此，主要的差异在于 INSERT 不需要游标。

执行与游标相关联的 INSERT

您还可使用 **sqlda** 结构来处理与插入游标相关联的 INSERT。在此情况下，您不以

EXECUTE...USING DESCRIPTOR 语句来执行该语句。反而，您必须声明并打开插入游标，并以 PUT...USING DESCRIPTOR 语句执行该插入游标，如下：

准备 INSERT 语句，并以 DECLARE 语句将它与插入游标相关联。所有的多行 INSERT 语句都必须有声明了的插入游标。

以 OPEN 语句为 INSERT 语句创建该游标。

以 PUT 语句以及它的 USING DESCRIPTOR 子句将第一组列值插入至缓冲区内。在此 PUT 语句之后，存储在指定的 **sqlda** 结构中的列值被存储在插入缓冲区中。在循环内反复执行该 PUT 语句，直到没有更多的列要插入为止。

在插入所有行之后，退出该循环，并以 FLUSH 语句刷新插入缓冲区。

以 CLOSE 语句关闭插入游标。

您处理插入游标的方式，与您处理与 SELECT 语句相关联的游标的方式相同。

4.4.6 处理参数化的 SELECT 语句

您可以 sqlda 结构来处理参数化的 SELECT 语句。

如果准备好的 SELECT 语句有带有未知数目和数据类型的输入参数的 WHERE 子句，则您的 GBase 8s ESQL/C 程序必须使用 **sqlda** 结构来定义输入参数。

要使用 **sqlda** 结构来为 WHERE 子句定义输入参数：

1. 声明变量来保存 sqlda 结构的地址。
2. 确定 SELECT 语句的输入参数的数目和数据类型。
3. 以诸如 malloc() 这样的系统内存分配函数分配 sqlda 结构。

以 C 语言语句指示 WHERE 子句中输入参数的数目，其设置 **sqlda** 结构的 **sqld** 字段。

以 C 语言语句存储每一输入参数的定义和值，其在 **sqlda** 结构的恰当的 **sqlvar_struct** 中设置 **sqltype**、**sqllen** 和 **sqldata** 字段：

sqltype 字段使用 sqltypes.h 头文件定义的 GBase 8s ESQL/C 数据类型常量，来表示输入参数的数据类型。

对于 CHAR 或 VARCHAR 值，sqllen 是以字节计的字符数组的大小。对于 DATETIME 或 INTERVAL 值，此字段存储编码的限定符。

每一 sqlvar_struct 结构的 sqldata 字段包含为输入参数值分配的内存的地址。对于每一输入参数，您可能需要使用 sqltype 和 sqllen 字段来确定需要分配的内存的数量。

如果您使用指示符变量，则还要设置 sqlind 字段，可能还要设置 sqlidata、sqlilen 和 sqlitype 字段（仅限于非 X/Open 应用程序）。

请使用 sqlda.sqlvar 数组内的索引来标识 sqlvar_struct 结构。

以 USING DESCRIPTOR 子句，将定义了的输入参数从 **sqlda** 结构传至数据库服务器。

提供输入参数的语句，依赖于该 SELECT 返回多少行。

4. 以 free() 系统调用,释放为 sqlvar_struct 字段、sqldata 字段，以及为 sqlda 结构本身分配的内存。

重要： 如果 SELECT 语句在选择列表中有未知的列，则您的程序还必须以 sqlda 结构来处理这些列。

执行返回多行的参数化的 SELECT

下列描述的示例程序是 demo4.ec 示例程序的一个修改版本。它展示如何以下列条件使用动态 SELECT 语句：

该 SELECT 返回多行。

该 SELECT 必须与游标相关联，以 OPEN 语句来执行，并以 FETCH...USING DESCRIPTOR 语句检索它的返回值。

该 SELECT 在它的 WHERE 子句中有输入参数。

OPEN 语句包括 USING DESCRIPTOR 子句来提供 **sqlda** 结构中的参数值。

该 SELECT 在选择列表中有未知的列。

FETCH 语句包括 USING DESCRIPTOR 子句来存储 **sqlda** 结构中的返回值。

使用 sqlda 结构的样例程序

该程序说明如何使用 **sqlda** 结构来同时处理 WHERE 子句的数据参数和选择列表中的列。

```
==========================================================================
```

```
1. #include <stdio.h>
2. EXEC SQL include sqlda;
3. EXEC SQL include sqltypes;
4. #define FNAME 15
5. #define LNAME 15
```

```
6. #define PHONE 18
```

==========

2 行

该程序必须包括 GBase 8s ESQL/Csqlda.h 头文件来使用 **sqlda** 结构。

==========

```
7. main()
8. {
9.       char fname[ FNAME + 1 ];
10.      char lname[ LNAME + 1 ];
11.      char phone[ PHONE + 1 ];
12.      int count, customer_num, i;
13.      struct sqlvar_struct *pos;
14.      struct sqlda *sqlda_ptr;
15.      printf("Sample ESQL program running.\n\n");
16.      EXEC SQL connect to 'stores7';
17.      stcopy("Carole", fname);
18.      stcopy("Sadler", lname);
19.      EXEC SQL prepare sql_id from
20.           'select * from customer where fname=? and lname=?';
21.      EXEC SQL declare slct_cursor cursor for sql_id;
```

==========

9 - 14 行

9 - 11 行声明变量来保存从用户取得的数据。**sqlda_ptr** 变量（14 行）是指向 **sqlda** 结构的指针。**pos** 变量（13 行）指向 **sqlvar_struct** 结构，以便于该代码可全程处理 **sqlda** 的变长部分中的每一个 **sqlvar_struct** 结构。不将这些变量定义作为 GBase 8s ESQL/C 主变量。

17 - 20 行

这些行为 SELECT 语句组装字符串，并准备该 SELECT 字符串。此程序假设输入参数的数目和数据类型。因此，在运行时刻，无需 C 代码来确定此信息。问号（?）指示 WHERE 子句中的输入参数。

21 行

此行为准备好的语句标识符声明 **slct_cursor** 游标。

```
=================================================================
======
22.     count=2;
23.     whereClauseMem(&sqlda_ptr, count, fname, lname);
24.     EXEC SQL open slct_cursor using descriptor sqlda_ptr;
25.     free(sqlda_ptr->sqlvar);
26.     free(sqlda_ptr);
=================================================================
======
```

22 和 23 行

这些行以输入参数信息初始化 **sqlda** 结构。该程序假设两个输入参数（22 行）。如果输入参数的数目是未知的，则该程序需要解析 SELECT 字符串（不是准备好的版本），并对它包含的 “?” 字符的数目计数。

然后，该程序调用 whereClauseMem() 函数来分配并初始化 **sqlda** 结构。要获取更多信息，请参阅 69 - 77 行。

24 行

当数据库服务器执行 SELECT 时，它打开该游标。您必须包括 OPEN 的 USING DESCRIPTOR 子句，来指定该 **sqlda** 结构作为输入参数值的位置。

25 和 26 行

一旦以执行了 OPEN...USING DESCRIPTOR 语句，就已使用了这些输入参数值。由于不再需要它，因此，释放此 **sqlda** 结构，且它与包含该检索的值的 **sqlda** 不冲突。请记住，在可使用这第二个 **sqlda** 之前，必须为它分配内存。

```
=================================================================
======
27.     EXEC SQL describe sql_id into sqlda_ptr;
28.     selectListMem(sqlda_ptr);
29.     while(1)
30.         {
31.         EXEC SQL fetch slct_cursor using descriptor sqlda_ptr;
32.         if(SQLCODE != 0)
```

```
33.             {
34.             printf("fetch SQLCODE %d\n", SQLCODE);
35.             break;
36.              }
```

==========

27 行

出于演示的目的，此程序假设在编译时刻，选择列表列的数目和数据类型还是未知的。它使用 DESCRIBE...INTO 语句（27 行）来分配 **sqlda** 结构，并将关于该选择列表列的信息放至 **sqlda_ptr** 指向的结构内。

28 行

selectListMem() 函数为列值处理内存的分配。

29 - 31 行

为从数据库访存的每一行执行该 while 循环。FETCH 语句（31 行）包括 USING DESCRIPTOR 子句来指定 sqlda 结构作为返回的列值的位置。

32 - 36 行

这些行检测 SQLCODE 变量的值，来确定该 FETCH 是否成功了。如果 SQLCODE 包含非零值，则该 FETCH 生成 NOT FOUND 条件（100）或错误（< 0）。在任何这些情况下，34 行打印出 SQLCODE 值。要确定 FETCH 语句是否生成了警告，您需要检测 **sqlca.sqlwarn** 结构。

==========

```
37.         for(i=0; i<sqlda_ptr->sqld; i++)
38.             {
39.             printf("\ni=%d\n", i);
40.             prsqlda(sqlda_ptr->sqlvar[i]);
41.             switch (i)
42.                 {
43.                 case 0:
44.                     customer_num = *(int *)(sqlda_ptr->sqlvar[i].sqldata);
```

```
45.                     break;
46.                 case 1:
47.                     stcopy(sqlda_ptr->sqlvar[i].sqldata, fname);
48.                     break;
49.                 case 2:
50.                     stcopy(sqlda_ptr->sqlvar[i].sqldata, lname);
51.                     break;
52.                 case 9:
53.                    stcopy(sqlda_ptr->sqlvar[i].sqldata, phone);
54.                     break;
55.                 }
56.             }
57.         printf("%d ==> |%s|, |%s|, |%s|\n",
58.             customer_num, fname, lname, phone);
59.         }
60.     EXEC SQL close slct_cursor;
61.     EXEC SQL free slct_cursor;
62.     EXEC SQL free sql_id;
```

==========

37 - 59 行

对于选择列表中的每一列，这些行访问 **sqlvar_struct** 结构的字段。prsqlda() 函数（请参阅 75 - 81 行）显示列名称（来自 **sqlvar_struct.sqlname**）以及它的值（来自 **sqlvar_struct.sqldata** field）。**switch**（41 - 55 行）将列值从 **sqlda** 结构移至恰当的长度和数据类型的主变量内。

60 - 62 行

在访存所有行之后，这些行释放资源。60 行关闭 **slct_cursor** 游标，且 61 行释放它。62 行释放 **sql_id** 语句 ID。

==========

```
63.     free(sqlda_ptr->sqlvar);
```

```
64.     free(sqlda_ptr);
65.     EXEC SQL close database;
66.     EXEC SQL disconnect current;
67.     printf("\nProgram Over.\n");
68. }
69. whereClauseMem(descp, count, fname, lname)
70.     struct sqlda **descp;
71.     int count;
72.     char *fname, *lname;
73. {
74.     (*descp)=(struct sqlda *) malloc(sizeof(struct sqlda));
75.     (*descp)->sqld=count;
76.     (*descp)->sqlvar=(struct sqlvar_struct *)
77.         calloc(count, sizeof(struct sqlvar_struct));
```

==========

63 和 64 行

这些 free() 系统调用释放与 sqlda 结构相关联的内存。63 行释放分配给 sqlvar_struct 结构的内存。64 行释放为 sqlda 结构分配的内存。该程序不需要释放与 sqldata 字段相关联的内存，因为这些字段已经使用了在数据缓冲区中的空间。

69 - 77 行

whereClauseMem() 函数以输入参数定义来初始化 **sqlda** 结构。74 行为 **sqlda** 结构分配内存来保存 WHERE 子句中的输入参数。使用 DESCRIBE...INTO 来分配 **sqlda** 导致保存关于该 SELECT 的选择列表列的信息的 **sqlda**。由于您想要在 WHERE 子句中描述输入参数，因此，请不要在此使用 DESCRIBE。

75 行将 **sqlda** 结构的 **sqld** 字段设置为 **count** 的值（2），来指示 WHERE 子句中参数的数目。76 和 77 行使用 calloc() 系统函数来分配内存，以便于 WHERE 子句中的每一输入参数都有一个 **sqlvar_struct** 结构。然后，这些行设置 **sqlda** 结构的 **sqlvar** 字段，以便于它指向此 **sqlvar_struct** 内存。

==========

```
======
    78.     (*descp)->sqlvar[0].sqltype = CCHARTYPE;
    79.     (*descp)->sqlvar[0].sqllen = FNAME + 1;
    80.     (*descp)->sqlvar[0].sqldata = fname;
    81.     (*descp)->sqlvar[1].sqltype = CCHARTYPE;
    82.     (*descp)->sqlvar[1].sqllen = LNAME + 1;
    83.     (*descp)->sqlvar[1].sqldata = lname;
    84. }
    85. selectListMem(descp)
    86.     struct sqlda *descp;
    87. {
    88.     struct sqlvar_struct *col_ptr;
    89.     static char buf[1024];
    90.     int pos, cnt, size;
    91.     printf("\nWITHIN selectListMem: \n");
    92.     printf("number of parms: %d\n", descp->sqld);
    93.     for(col_ptr=descp->sqlvar, cnt=pos=0; cnt < descp->sqld;
    94.         cnt++, col_ptr++)
    95.         {
    96.         prsqlda(col_ptr);
    97.         pos = rtypalign(pos, col_ptr->sqltype);
    98.         col_ptr->sqldata = &buf[pos];
    99.         size = rtypmsize(col_ptr->sqltype, col_ptr->sqllen);
    100.          pos += size;
    101.          }
    102. }
    ================================================================
======
```

78 - 84 行

78 - 80 行设置 **sqlvar_struct** 结构的 **sqltype**、**sqllen** 和 **sqldata** 字段，来描述第一个输入参数：其数据存储在 **fname** 缓冲区中的长度为 16（FNAME + 1）的字符（CCHARTYPE）主变量。**fname** 缓冲区是在 main() 程序中声明的字符缓冲区，且作为参数传给 whereClauseMem()。

81 行设置 **sqlvar_struct** 结构的 **sqltype**、**sqllen** 和 **sqldata** 字段来描述第二个输入参数。此参数是针对 **lname** 列的。定义它的方式，与 **fname** 列一样（78 - 80 行），但它从 **lname** 缓冲区接收它的数据 [还被从 main() 传至 whereClauseMem()]。

85 - 102 行

selectListMem() 函数分配内存，并为参数化的 SELECT 语句的未知的选择列表列初始化 sqlda 结构。

执行参数化的单个 SELECT 语句

在前一主题中的说明假设参数化的 SELECT 语句返回多个值，因此，与游标相关联。如果您在此时知道您编写一个程序，参数化的 SELECT 语句总是只返回一行，则您可省略游标并使用 EXECUTE...USING DESCRIPTOR...INTO 语句，而不使用 OPEN...USING DESCRIPTOR 语句，来指定来自 **sqlda** 结构的参数值。

4.4.7 处理参数化的用户定义的例程

本部分描述如何以 sqlda 结构来处理参数化的用户定义的例程。下列语句执行用户定义的例程：

EXECUTE FUNCTION 语句执行（外部的和 SPL）用户定义的函数。

EXECUTE PROCEDURE 语句执行（外部的和 SPL）用户定义的过程。

如果准备好的 EXECUTE PROCEDURE 或 EXECUTE FUNCTION 语句有参数，指定这些参数作为未知数目和数据类型的输入参数，则您的 GBase 8s ESQL/C 程序可使用 **sqlda** 结构来定义该输入参数。

执行参数化的函数

您处理用户定义的函数的输入参数的方式，与您处理 SELECT 语句的 WHERE 子句中的输入参数的方式相同，如下：

执行非游标函数的方式，与单个 SELECT 语句的方式相同。

> 如果您在此时知道您编写的程序，其动态的用户定义的函数总是只返回一行，则您可使用 EXECUTE...USING DESCRIPTOR...INTO 语句来提供来自 sqlda 结构的参数值，并执行该函数。

执行游标函数的方式，与返回一行或多行的 SELECT 语句的方式相同。

如果您在此时不确定您编写的程序，其动态的用户定义的函数是否总是只返回一行，则请定义函数游标，并使用 OPEN...USING DESCRIPTOR 语句来提供来自 **sqlda** 结构的参数值。

执行这些 EXECUTE FUNCTION 与 SELECT 语句的唯一区别在于，您为非游标函数准备 EXECUTE FUNCTION 语句，而不是 SELECT 语句。

执行参数化的过程

要执行参数化的用户定义的过程，您可使用 EXECUTE...USING DESCRIPTOR 语句来提供来自 **sqlda** 结构的参数值，并执行该过程。您处理用户定义的过程的输入参数的方式，与您处理非游标函数的输入参数的方式相同。执行 EXECUTE PROCEDURE 语句与执行 EXECUTE FUNCTION 语句（对于非游标函数）的唯一区别，在于您不需要为用户定义的过程指定 EXECUTE...USING DESCRIPTOR 语句的 INTO 子句。

4.4.8 处理参数化的 UPDATE 或 DELETE 语句

确定 DELETE 或 UPDATE 语句的 WHERE 子句中的输入参数的方式，类似于确定 SELECT 语句的 WHERE 子句中的它们的方式。这两类动态的参数化的语句之间的主要区别，如下：

您不需要使用游标来处理 DELETE 或 UPDATE 语句。您以 EXECUTE 语句的 USING DESCRIPTOR 子句提供来自 sqlda 结构的参数值，而不是以 OPEN 语句。

您可使用 DESCRIBE...INTO 语句来确定该 DELETE 或 UPDATE 语句是否有 WHERE 子句。

5 附录

本部分包含附加的参考信息。

5.1 ESQL/C 示例程序

您的 GBase 8s 软件包括演示数据库。GBase 8s ESQL/C 在本出版物中，还包括许多演示程序和示例的源文件，其中有些访问该演示数据库。

在 Windows™ 环境中，您可在 %GBASEDBTDIR%\demo\esqldemo 目录中找到 GBase 8s ESQL/C 示例程序的源文件。

在 UNIX™ 操作系统上，您可在 $GBASEDBTDIR/demo/esqlc 目录中找到 GBase 8s ESQL/C 示例程序的源文件。包括 GBase 8s ESQL/C 的 esqldemo 脚本将源文件从 $GBASEDBTDIR/demo/esqlc 目录复制至当前目录内。

要获取关于创建演示数据库的信息，请参阅 GBase 8s DB-Access 用户指南。

5.2 ESQL/C 函数库

这些主题描述随同 GBase 8s ESQL/C 提供的所有库函数的语法和行为。

5.2.1 GBase 8s ESQL/C库函数

下表按字母顺序罗列 GBase 8s ESQL/C 库函数

函数名称	描述
bigintcvasc()	将 C char 类型值转换为 BIGINT 类型数值。
bigintcvdbl()	将 double 类型数值转换为 BIGINT 类型数值。

函数名称	描述
bigintcvdec()	将 decimal 类型数值转换为 BIGINT 类型数值。
bigintcvflt()	将 float 类型数值转换为 BIGINT 类型数值。
bigintcvifx_int8()	将 int8 类型数值转换为 BIGINT 类型数值。
bigintcvint2()	将 int2 类型数值转换为 BIGINT 类型数值。
bigintcvint4()	将 int4 类型数值转换为 BIGINT 类型数值。
biginttoasc()	将 BIGINT 类型值转换为 C char 类型值。
biginttodbl()	将 BIGINT 类型数值转换为 double 类型数值。
biginttodec()	将 BIGINT 类型数值转换为 decimal 类型数值。
biginttoflt()	将 BIGINT 类型数值转换为 float 类型数值。
biginttoifx_int8()	将 BIGINT 类型数值转换为 int8 类型数值。
biginttoint2()	将 BIGINT 类型数值转换为 int2 类型数值。
biginttoint4()	将 BIGINT 类型数值转换为 int4 类型数值。
bycmpr()	比较两组相邻的字节
bycopy()	将字节从一个区域复制到另一个
byfill()	以字符填充指定的区域
byleng()	对字符串中的字节数计数
decadd()	两个 **decimal** 数值相加
deccmp()	比较两个 **decimal** 数值
deccopy()	复制 decimal 数值
deccvasc()	将 C **char** 类型转换为 **decimal** 类型
deccvdbl()	将 C **double** 类型转换为 **decimal** 类型
deccvint()	将 C **int2** 类型转换为 **decimal** 类型
deccvlong()	将 C **int4** 类型转换为 **decimal** 类型
decdiv()	两个 **decimal** 数值相除
dececvt()	将 **decimal** 值转换为 ASCII 字符串
decfcvt()	将 **decimal** 值转换为 ASCII 字符串
decmul()	两个 **decimal** 数值相乘
decround()	四舍五入 **decimal** 数值

函数名称	描述
decsub()	两个 **decimal** 数值相减
dectoasc()	将 **decimal** 类型转换为 ASCII 字符串
dectodbl()	将 **decimal** 类型转换为 C **double** 类型
dectoint()	将 **decimal** 类型转换为 C **int** 类型
dectolong()	将 **decimal** 类型转换为 C **long** 类型
dectrunc()	截断 decimal 数值
dtaddinv()	将 interval 值加到 datetime 值
dtcurrent()	取得当前日期和时间
dtcvasc()	将符合 ANSI 的字符串转换为 **datetime**
dtcvfmtasc()	将字符串转换为 **datetime** 值
dtextend()	更改 **datetime** 的限定符
dtsub()	从另一 datetime 减去一个 datetime 值
dtsubinv()	从 datetime 值减去 interval 值
dttoasc()	将 **datetime** 值转换为符合 ANSI 的字符串
dttofmtasc()	将 **datetime** 值转换为字符串
GetConnect()	请求显式的连接，并返回指向该连接信息的指针
ifx_cl_card()	返回指定的集合类型主变量的基数
ifx_dececvt()	将 decimal 值转换为 ASCII 字符串（线程安全版本）
ifx_decfcvt()	将 decimal 值转换为 ASCII 字符串（线程安全版本）
ifx_getcur_conn_name()	返回当前连接的名称
ifx_getenv()	检索环境变量的值
ifx_getserial8()	返回插入了的 SERIAL8 值
ifx_int8add()	将两个 **int8** 数值相加
ifx_int8cmp()	比较两个 **int8** 数值
ifx_int8copy()	复制 **int8** 数值
ifx_int8cvasc()	将 C **char** 类型值转换为 **int8** 类型值
ifx_int8cvdbl()	将 C **double** 类型值转换为 **int8** 类型值

函数名称	描述
ifx_int8cvdec()	将 C **decimal** 类型值转换为 **int8** 类型值
ifx_int8cvflt()	将 C **float** 类型值转换为 **int8** 类型值
ifx_int8cvint()	将 C **int2** 类型值转换为 **int8** 类型值
ifx_int8cvlong()	将 C **int4** 类型值转换为 **int8** 类型值
ifx_int8div()	两个 **int8** 数值相除
ifx_int8mul()	两个 **int8** 数值相乘
ifx_int8tub()	两个 **int8** 数值相减
ifx_int8toasc()	将 **int8** 类型值转换为文本字符串
ifx_int8todbl()	将 **int8** 类型数值转换为 C **double** 类型值
ifx_int8todec()	将 **int8** 类型值转换为 **decimal** 类型值
ifx_int8toflt()	将 **int8** 类型值转换为 C **float** 类型值
ifx_int8toint()	将 **int8** 类型值转换为 C **int2** 类型值
ifx_int8tolong()	将 **int8** 类型值转换为 C **int4** 类型值
ifx_lo_alter()	修改现有的智能大对象的存储特征
ifx_lo_close()	关闭打开的智能大对象
ifx_lo_col_info()	将列级别存储特征取至 LO 规范结构内
ifx_lo_copy_to_file()	将智能大对象复制到操作系统文件
ifx_lo_copy_to_lo()	将操作系统文件复制到打开的智能大对象
ifx_lo_create()	为智能大对象创建 LO 描述符
ifx_lo_def_create_spec()	分配 LO 规范结构，并将它的字段初始化为空值
ifx_lo_filename()	返回生成的文件名称，给定 LO 描述符和文件规范
ifx_lo_from_buffer()	将字节从用户定义的缓冲区复制到智能大对象
ifx_lo_open()	打开现有的智能大对象
ifx_lo_read()	从打开的智能大对象读取指定的字节数
ifx_lo_readwithseek()	在打开的智能大对象中寻找指定的文职，并读取指定的字节数
ifx_lo_release()	释放与临时的智能大对象相关联的资源
ifx_lo_seek()	为了对打开的智能大对象的下一次读或写设置寻找位置

函数名称	描述
ifx_lo_spec_free()	释放分配给 LO 规范结构的资源
ifx_lo_specget_estbytes()	从 LO 规范结构取得估计的字节数
ifx_lo_specget_extsz()	从 LO 规范结构取得分配 extent 大小
ifx_lo_specget_flags()	从 LO 规范结构缺的创建时间标志
ifx_lo_specget_maxbytes()	从 LO 规范结构取得最大字节数
ifx_lo_specget_sbspace()	从 LO 规范结构取得 sbspace 的名称
ifx_lo_specset_estbytes()	从 LO 规范结构设置估计的字节数
ifx_lo_specset_extsz()	在 LO 规范结构中设置分配 extent 大小
ifx_lo_specset_flags()	在 LO 规范结构中设置创建时间标志
ifx_lo_specset_maxbytes()	在 LO 规范结构中设置最大字节数
ifx_lo_specset_sbspace()	在 LO 规范结构中设置 sbspace 的名称
ifx_lo_stat()	返回关于打开的智能大对象的状态信息
ifx_lo_stat_atime()	返回智能大对象的最后访问时间
ifx_lo_stat_cspec()	对于指定的智能大对象，将存储特征返回至 LO 规范结构内
ifx_lo_stat_ctime()	对于智能大对象，返回最后的状态更改时间
ifx_lo_stat_free()	释放分配给 LO 状态结构的资源
ifx_lo_stat_mtime_sec()	对于智能大对象，返回以秒计的最后修改时间
ifx_lo_stat_refcnt()	返回对于智能大对象的引用计数
ifx_lo_stat_size()	返回智能大对象的大小
ifx_lo_tell()	返回打开的智能大对象的当前寻找位置
ifx_lo_to_buffer()	将字节从智能大对象复制至用户定义的缓冲区内
ifx_lo_truncate()	将智能大对象截断到特定的偏移量
ifx_lo_write()	将指定的字节数写到打开的智能大对象
ifx_lo_writewithseek()	在打开的智能大对象中寻找指定的位置，并写指定的字节数
ifx_lvar_alloc()	当访存 **lvarchar** 数据时，指定是否分配内存
ifx_putenv()	修改或移除现有的环境变量，或创建一个新的
ifx_var_alloc()	为数据缓冲区分配内存

函数名称	描述
ifx_var_dealloc()	为数据缓冲区释放内存
ifx_var_flag()	确定是由 GBase 8s ESQL/C，还是由应用程序来为数据缓冲区处理内存分配
ifx_var_getdata()	返回数据缓冲区的内容
ifx_var_getlen()	返回数据缓冲区的长度
ifx_var_isnull()	检查数据缓冲区中的数据是否为空
ifx_var_setdata()	为数据缓冲区设置数据
ifx_var_setlen()	设置数据缓冲区的长度
ifx_var_setnull()	将数据缓冲区中的数据设置为空值
incvasc()	将符合 ANSI 的字符串转换为 **interval** 值
incvfmtasc()	将字符串转换为 **interval** 值
intoasc()	将 **interval** 值转换为符合 ANSI 的字符串
intofmtasc()	将 **interval** 值转换为字符串
invdivdbl()	将 **interval** 值除以一个数值值
invdivinv()	将 interval 值除以一个 interval 值
invextend()	在不同的限定符之下复制 **interval** 值
invmuldbl()	interval 值乘以数值值
ldchar()	将定长字符串复制到以空结尾的字符串
rdatestr()	将内部的格式转换为字符串
rdayofweek()	返回星期几
rdefmtdate()	将字符串转换为内部的格式
rdownshift()	将所有字母转换为小写
ReleaseConnect()	关闭建立了的显式连接
rfmtdate()	将内部的格式转换为字符串
rfmtdec()	将 **decimal** 类型转换为格式化的字符串
rfmtdouble()	将 **double** 类型转换为字符串
rfmtlong()	将 **int4** 转换为格式化的字符串
rgetlmsg()	对于大型错误编号，检索错误消息
rgetmsg()	对于错误编号，检索错误消息

函数名称	描述
risnull()	检查 C 变量是否为空
rjulmdy()	返回来自内部格式的月、日和年
rleapyear()	确定指定的年是否为闰年
rmdyjul()	从月、日和年返回内部的格式
rsetnull()	将 C 变量设置为空
rstod()	将字符串转换为 **double** 类型
rstoi()	将以空结尾的字符串转换为 **int2**
rstol()	将字符串转换为 **int4**
rstrdate()	将字符串转换为内部的格式
rtoday()	以内部的格式返回系统日期
rtypalign()	将数据对其恰当的类型边界
rtypmsize()	给定 SQL 数据类型的字节大小
rtypname()	返回指定的 SQL 类型的名称
rtypwidth()	给定最小转换字节大小
rupshift()	将所有字母转换为大写
SetConnect()	将连接切换到建立了的（休眠的）显式连接
sqgetdbs()	返回数据库服务器可连接的数据库的名称
sqlbreak()	发送请求给数据库服务器，来停止处理
sqlbreakcallback()	提供返回控制的方法给应用程序，在它等待数据库服务器处理 SQL 请求时
sqldetach()	将子进程从父进程脱离
sqldone()	确定数据库服务器当前是否正在处理 SQL 请求
sqlexit()	终止数据库服务器进程
sqlsignal()	执行信号处理和子进程清理
sqlstart()	启动数据库服务器进程
stcat()	将一个字符串连接到另一个
stchar()	将以空结尾的字符串复制到定长字符串
stcmpr()	比较两个字符串
stcopy()	将一个字符串复制到另一字符串

函数名称	描述
stleng()	对字符串中的字节数计数

5.2.2 bigintcvasc() 函数

bigintcvasc() 函数将 C char 类型值转换为 BIGINT 类型数值。

语法

```
mint bigintcvasc(strng_val, len, bigintp)
        const char *strng_val
        mint len
        bigint *bigintp
```

strng_val

指向字符串的指针。

len

strng_val 字符串的长度。

bigintp

指向包含转换结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.3 bigintcvdbl() 函数

bigintcvdbl() 函数将 double 类型数值转换为 BIGINT 类型数值。

语法

```
mint bigintcvdbl(dbl, bigintp)
        const double dbl
        bigint *bigintp
```

dbl

要转换为 bigint 的 double 值。

bigintp

指向包含转换的结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.4 bigintcvdec() 函数

bigintcvdec() 函数将 decimal 类型数值转换为 BIGINT 类型数值。

语法

```
mint bigintcvdec(decp, bigintp)
        const dec_t *decp
        bigint *bigintp
```

decp

指向包含要转换为 bigint 的该值的十进制结构的指针。

bigintp

指向包含转换的结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.5 bigintcvflt() 函数

bigintcvflt() 函数将 float 类型数值转换为 BIGINT 类型数值。

语法

mint bigintcvflt(dbl, bigintp)

const double dbl

bigint *bigintp

dbl

要转换为 bigint 的 float 值。

bigintp

指向包含转换的结果的 bigint 值的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.6 bigintcvifx_int8() 函数

bigintcvifx_int8() 函数将 int8 类型数值转换为 BIGINT 类型数值。

语法

mint bigintcvifx_int8(int8p, bigintp)

const ifx_int8_t *int8p

bigint *bigintp

int8p

要转换为 bigint 值的 int8 值。

bigintp

指向包含转换的结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.7 bigintcvint2() 函数

bigintcvint2() 函数将 int2 类型数值转换为 BIGINT 类型数值。

语法

mint bigintcvint2(int2v, bigintp)

const int2 int2v

bigint *bigintp

int2v

要转换为 bigint 值的 int2 值。

bigintp

指向包含转换的结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.8 bigintcvint4() 函数

bigintcvint4() 函数将 int4 类型数值转换为 BIGINT 类型数值。

语法

mint bigintcvint4(int4v, bigintp)

const int4 int4v

bigint *bigintp

int4v

要转换为 bigint 值的 int4 值。

bigintp

指向包含转换的结果的 bigint 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.9 biginttoasc() 函数

biginttoasc() 函数将 BIGINT 类型值转换为 C char 类型值。

语法

mint biginttoasc(bigintv, strng_val, len, base)

const bigint bigintv

char *strng_val

mint len

mint base

bigintv

要转换为文本字符串的 bigint 值。

strng_val

指向包含文本字符串的字符缓冲区的第一个字节的指针。

len

以字节计的 *strng_val* 的大小，对于空终止符为负 1。

base

输出的数值进制。支持 10 进制和 16 进制。其他值导致 10 进制。

返回代码

0

转换成功。

<0

转换失败。

5.2.10 biginttodbl() 函数

biginttodbl() 函数将 BIGINT 类型数值转换为 double 类型数值。

语法

mint biginttodbl(bigintv, dbl)

const bigint bigintv

double *dbl

bigintv

要转换为 double 的 bigint 值。

dbl

指向包含转换的结果的 double 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.11 biginttodec() 函数

biginttodec() 函数将 BIGINT 类型数值转换为 decimal 类型数值。

语法

```
mint biginttodec(bigintv, decp)
        const bigint bigintv
        dec_t *decp
```

bigintv

要转换为 decimal 的 bigint 值。

decp

指向包含转换的结果的 decimal 变量的指针。

返回结果

0

转换成功。

<0

转换失败。

5.2.12 biginttoflt() 函数

biginttoflt() 函数将 BIGINT 类型数值转换为 float 类型数值。

语法

```
mint biginttoflt(bigintv, fltp)
        const bigint bigintv
        float *fltp
```

bigintv

要转换为 float 的 bigint 值。

fltp

指向包含转换的结果的 float 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.13 biginttoifx_int8() 函数

biginttoifx_int8() 函数将 BIGINT 类型数值转换为 int8 类型数值。

语法

```
void biginttoifx_int8(bigintv, int8p)
        const bigint bigintv
        ifx_int8_t *int8p
```

bigintv

要转换为 int8 的 bigint 值。

int8p

指向包含转换的结果的 int8 结构的指针。

5.2.14 biginttoint2() 函数

biginttoint2() 函数将 BIGINT 类型数值转换为 int2 类型数值。

语法

```
mint biginttoint2(bigintv, int2p)
        const bigint bigintv
        int2 *int2p
```

bigintv

要转换为 int2 整数值的 bigint 值。

int2p

指向包含转换的结果的 int 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.15 biginttoint4() 函数

biginttoint4() 函数将 BIGINT 类型数值转换为 int4 类型数值。

语法

```
mint biginttoint4(bigintv, int4p)
        const bigint bigintv
        int4 *int4p
```

bigintv

要转换为 int4 整数值的 bigint 值。

int4p

指向包含转换的结果的 int4 变量的指针。

返回代码

0

转换成功。

<0

转换失败。

5.2.16 bycmpr() 函数

bycmpr() 函数比较两组给定长度的相邻字节。它返回比较的结果作为它的值。

语法

```
mint bycmpr(byte1, byte2, length)
        char *byte1;
        char *byte2;
        mint length;
```

byte1

指向第一组相邻字节起始位置的指针。

byte2

指向第二组相邻字节起始位置的指针。

length

您想要 bycmpr() 比较的字节数。

用法

bycmpr() 执行对两组相邻字节的逐字节比较，直到它发现不同为止，或直到它比较字节的 *length* 数目为止。bycmpr() 函数返回整数，其值（0、-1 或 +1）指示两组字节之间比较的结果。

bycmpr() 函数从 *byte1* 组的字节减去 *byte2* 组的字节来完成比较。

返回代码

0

两个组相同。

-1

byte1 组小于 byte2 组。

+1

byte1 组大于 byte2 组。

示例

此样例程序位于 demo 目录中的 bycmpr.ec 文件中。

```
/*
* bycmpr.ec *

The following program performs three different byte comparisons with bycmpr() and displays the results.
                */

#include <stdio.h>

main()
 {
mint x;
```

```
static char string1[] = "abcdef";
static char string2[] = "abcdeg";

static mint number1 = 12345;
static mint number2 = 12367;

static char string3[] = {0x00, 0x07, 0x45, 0x32, 0x00};
static char string4[] = {0x00, 0x07, 0x45, 0x31, 0x00};

printf("BYCMPR Sample ESQL Program running.\n\n");

/* strings */
printf("Comparing strings: String 1=%s\tString 2=%s\n", string1, string2);
printf("    Executing: bycmpr(string1, string2, sizeof(string1))\n");
x = bycmpr(string1, string2, sizeof(string1));
printf("    Result = %d\n", x);

/* ints */
printf("Comparing numbers: Number 1=%d\tNumber 2=%d\n", number1, number2);
printf("    Executing: bycmpr( (char *) &number1, (char *) &number2, ");
printf("sizeof(number1))\n");
x = bycmpr( (char *) &number1, (char *) &number2, sizeof(number1));
printf("    Result = %d\n", x);

/* non printable */
printf("Comparing strings with non-printable characters:\n");
printf("    Octal string 1=%o\tOctal string 2=%o\n", string3, string4);
printf("    Executing: bycmpr(string3, string4, sizeof(string3))\n");
x = bycmpr(string3, string4, sizeof(string3));
printf("    Result = %d\n", x);

/* bytes */
```

```
printf("Comparing bytes: Byte string 1=%c%c\tByte string 2=%c%c\n");
printf("    Executing: bycmpr(&string1[2], &string2[2], 2)\n");
x = bycmpr(&string1[2], &string2[2], 2);
printf("    Result = %d\n", x);

printf("\nBYCMPR Sample ESQL Program over.\n\n");
            }
```

输出

```
BYCMPR Sample ESQL Program running.

        Comparing strings: String1=abcdef          String 2=abcdeg
        Executing: bycmpr(string1, string2, sizeof(string1))
        Result = -1
        Comparing numbers: Number 1=12345          Number 2=12367
        Executing: bycmpr( (char *) &number1,    (char *) &number2, sizeof(number1)
        Result = -1
        Comparing strings with non-printable characters:
        Octal string 1=40300       Octal string 2=40310
        Executing: bycmpr(string3, string4, sizeof(string3))
        Result = 1
        Comparing bytes: Byte string 1=cd          Byte string 2=cd
        Executing: bycmpr(&string1[2], &string2[2], 2)
        Result = 0

        BYCMPR Sample ESQL Program over.
```

5.2.17 bycopy() 函数

bycopy() 函数将给定的字节数从一个位置复制到另一个。

语法

```
void bycopy(from, to, length)
        char *from;
        char *to;
```

mint length;

from

指向您想要 bycopy() 复制的该组字节的第一个字节的指针。

to

指向目的字节组的第一个字节的指针。*to* 指向的内存区域可与 *from* 参数指向的区域交叠。在此情况下，GBase 8s ESQL/C 不保留 *from* 指向的值。

length

您想要 bycopy() 复制的字节数。

重要： 请注意，不要重写目标区域相邻的内存区域。

示例

此示例程序位于 demo 目录中的 bycopy.ec 文件中。

```
/*
    * bycopy.ec *

The following program shows the results of bycopy() for three copy operations.
                */

                #include <stdio.h>

                char dest[20];

                main()
                {
                mint number1 = 12345;
                mint number2 = 0;
                static char string1[] = "abcdef";
                static char string2[] = "abcdefghijklmn";

                printf("BYCOPY Sample ESQL Program running.\n\n");

                printf("String 1=%s\tString 2=%s\n", string1, string2);
                printf("    Copying String 1 to destination string:\n");
```

```
bycopy(string1, dest, strlen(string1));
printf("    Result = %s\n\n", dest);

printf("    Copying String 2 to destination string:\n");
bycopy(string2, dest, strlen(string2));
printf("    Result = %s\n\n", dest);

printf("Number 1=%d\tNumber 2=%d\n", number1, number2);
printf("    Copying Number 1 to Number 2:\n");
bycopy( (char *) &number1, (char *) &number2, sizeof(int));
printf("    Result = number1(hex) %08x, number2(hex) %08x\n",
number1, number2);

printf("\nBYCOPY Sample Program over.\n\n");
}
```

输出

```
BYCOPY Sample ESQL Program running.

String 1=abcdef    String2=abcdefghijklmn
Copying String 1 to destination string:
Result    = abcdef

Copying String 2 to destination string:
Result = abcdefghijklmn

Number 1=12345    Number2=0
Copying Number 1 to Number 2:
Result = number1(hex) 00003039, number2(hex) 00003039

BYCOPY Sample Program over.
```

5.2.18 byfill() 函数

byfill() 函数以一个字符填充指定的区域。

语法

void byfill(*to*, *length*, *ch*)

char *to;

mint length;

char ch;

to

指向您想要 byfill() 填充的内存区域的第一个字节的指针。

length

您想要 byfill() 在该区域内重复该字符的次数。

ch

您想要 byfill() 用于填充该区域的字符。

重要： 请注意，不要重写您想要 byfill() 填充的区域相邻的内存区域。

示例

此样例程序位于 demo 目录中的 byfill.ec 文件中。

```
/*
        * byfill.ec *

The following program shows the results of three byfill() operations on an area that is initialized to x's.
        */

        #include <stdio.h>

        main()
        {
        static char area[20] = "xxxxxxxxxxxxxxxxxxx";

        printf("BYFILL Sample ESQL Program running.\n\n");

        printf("String = %s\n", area);
```

```
    printf("\nFilling string with five 's' characters:\n");
    byfill(area, 5, 's');
    printf("Result = %s\n", area);

    printf("\nFilling string with two 's' characters starting at ");
    printf("position 16:\n");
    byfill(&area[16], 2, 's');
    printf("Result = %s\n", area);

    printf("Filling entire string with 'b' characters:\n");
    byfill(area, sizeof(area)-1, 'b');
    printf("Result = %s\n", area);

    printf("\nBYFILL Sample Program over.\n\n");
    }
```

输出

```
BYFILL Sample ESQL Program running.

        String = xxxxxxxxxxxxxxxxxxx

        Filling string with five 's' characters:
        Result = sssssxxxxxxxxxxxxxx

        Filling string with two 's' characters starting at      position 16:
        Result = sssssxxxxxxxxxxxxssx

        Filling entire string with 'b' characters:
        Result = bbbbbbbbbbbbbbbbbbb

        BYFILL Sample Program over.
```

5.2.19 byleng() 函数

byleng() 函数返回字符串中有效字符的数目，结尾的空格符不计数。

语法

mint byleng(*from*, *count*)

char *from;

mint count;

from

指向定长字符串（不以空结尾）的指针。

count

定长字符串中的字节数。这不包括结尾的空格符。

示例

此样例程序位于 demo 目录中的 byleng.ec 文件中。

```
/*
        * byleng.ec *

The following program uses byleng() to count the significant characters in an area.
        */

        #include <stdio.h>

        main()
        {
        mint x;
        static char area[20] = "xxxxxxxxxx          ";


        printf("BYLENG Sample Program running.\n\n");

        /* initial length */
        printf("Initial string:\n");
```

```
    x = byleng(area, 15);
    printf("    Length =    %d, String = '%s'\n", x, area);

    /* after copy */
    printf("\nAfter copying two 's' characters starting ");
    printf("at position 16:\n");
    bycopy("ss", &area[16], 2);
    x = byleng(area, 19);
    printf("    Length = %d, String = '%s'\n", x, area);

    printf("\nBYLENG Sample Program over.\n\n");
    }
```

输出

```
BYLENG Sample Program running.

 Initial string:
Length = 10, String = 'xxxxxxxxxx                '
      After copying two 's' characters starting at position    16:
Length = 18, String =    'xxxxxxxxxx           ss '

 BYLENG Sample Program over.
```

5.2.20 decadd() 函数

decadd() 函数将两个 decimal 类型值相加。

语法

```
mint decadd(n1, n2, sum)
        dec_t *n1;
        dec_t *n2;
        dec_t *sum;
```

n1

指向第一个运算对象的 **decimal** 结构的指针。

n2

指向第二个运算对象的 **decimal** 结构的指针。

sum

指向包含总和（*n1 + n2*）的 **decimal** 结构的指针。

用法

sum 可与 *n1* 或 *n2* 相同。

返回代码

0

操作成功。

-1200

导致溢出的操作。

-1201

导致下溢的操作。

示例

demo 目录中的文件 decadd.ec 包含下列样式程序。

```
/*
* decadd.ec *

The following program obtains the sum of two DECIMAL numbers.
                */
#include <stdio.h>

EXEC SQL include decimal;
/* leading spaces will be ignored */
char string1[] = "    1000.6789";
char string2[] = "80";
char result[41];
```

```
 main()
{
mint x;
dec_t num1, num2, sum;

printf("DECADD Sample ESQL Program running.\n\n");

if (x = deccvasc(string1, strlen(string1), &num1))
{
printf("Error %d in converting string1 to DECIMAL\n", x);
     exit(1);
}
if (x = deccvasc(string2, strlen(string2), &num2))
{
printf("Error %d in converting string2 to DECIMAL\n", x);
exit(1);
}
if (x = decadd(&num1, &num2, &sum))
{
printf("Error %d in adding DECIMALs\n", x);
exit(1);
   }
       if (x = dectoasc(&sum, result, sizeof(result), -1))
   {
printf("Error %d in converting DECIMAL result to string\n", x);
exit(1);
   }
   result[40] = '\0';
 /* display result */
 printf("\t%s + %s = %s\n", string1, string2, result);
 printf("\nDECADD Sample Program over.\n\n");
 exit(0);
                }
```

输出

DECADD Sample ESQL Program running.

1000.6789 + 80 = 1080.6789

DECADD Sample Program over.

5.2.21 deccmp() 函数

deccmp() 函数比较两个 decimal 类型数值。

语法

mint deccmp(*n1*, *n2*)

dec_t *n1;

dec_t *n2;

n1

指向要比较的第一个数值的 **decimal** 结构的指针。

n2

指向要比较的第二个数值的 **decimal** 结构的指针。

返回代码

-1

第一个值小于第二个值。

0

两个值相同。

1

第一个值大于第二个值。

DECUNKNOWN

有一个值为空。

示例

demo 目录中的文件 deccmp.ec 包含下列样例程序。

/*

* deccmp.ec *

The following program compares DECIMAL numbers and displays the results.

```
*/

#include <stdio.h>

EXEC SQL include decimal;
/* leading spaces will be ignored */
char string1[] = "-12345.6789";
char string2[] = "12345.6789";
char string3[] = "-12345.6789";
char string4[] = "-12345.6780";

main()
{
mint x;
dec_t num1, num2, num3, num4;

 printf("DECCOPY Sample ESQL Program running.\n\n");

if (x = deccvasc(string1, strlen(string1), &num1))
 {
printf("Error %d in converting string1 to DECIMAL\n", x);
   exit(1);
            }
 if (x = deccvasc(string2, strlen(string2), &num2))
 {
   printf("Error %d in converting string2 to DECIMAL\n", x);
   exit(1);
            }
      if (x = deccvasc(string3, strlen(string3), &num3))
      {
      printf("Error %d in converting string3 to DECIMAL\n", x);
      exit(1);
            }
```

```
    if (x = deccvasc(string4, strlen(string4), &num4))
    {
    printf("Error %d in converting string4 to DECIMAL\n", x);
    exit(1);
            }

   printf("Number 1 = %s\tNumber 2 = %s\n", string1, string2);
   printf("Number 3 = %s\tNumber 4 = %s\n",string3, string4);
   printf("\nExecuting: deccmp(&num1, &num2)\n");
   printf("    Result = %d\n", deccmp(&num1, &num2));
   printf("Executing: deccmp(&num2, &num3)\n");
   printf("    Result = %d\n", deccmp(&num2, &num3));
   printf("Executing: deccmp(&num1, &num3)\n");
   printf("    Result = %d\n", deccmp(&num1, &num3));
   printf("Executing: deccmp(&num3, &num4)\n");
   printf("    Result = %d\n", deccmp(&num3, &num4));

   printf("\nDECCMP Sample Program over.\n\n");
   exit(0);
            }
```

输出

```
DECCMP Sample ESQL Program running.

        Number 1 = -12345.6789          Number 2 = 12345.6789
        Number 3 = -12345.6789          Number 4 = -12345.6780

        Executing: deccmp(&num1, &num2)
        Result = -1
        Executing: deccmp(&num2, &num3)
        Result = 1
        Executing: deccmp(&num1, &num3)
        Result = 0
```

Executing: deccmp(&num3, &num4)

Result = -1

DECCMP Sample Program over.

5.2.22 deccopy() 函数

deccopy() 函数将一个 decimal 结构复制到另一个。

语法

void deccopy(*source*, *target*)

dec_t *source;

dec_t *target;

source

指向保存在源 **decimal** 结构中的值的指针。

target

指向目标 **decimal** 结构的指针。

deccopy() 函数不返回状态值。要确定复制操作成功与否，请查看 *target* 指向的 **decimal** 结构的内容。

示例

demo 目录中的文件 deccopy.ec 包含下列样例程序。

```
/*
        * deccopy.ec *

  The following program copies one DECIMAL number to another.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string1[] = "12345.6789";
        char result[41];
```

```
main()
{
int x;
dec_t num1, num2;

printf("DECCOPY Sample ESQL Program running.\n\n");

printf("String = %s\n", string1);
if (x = deccvasc(string1, strlen(string1), &num1))
{
printf("Error %d in converting string1 to DECIMAL\n", x);
exit(1);
}
printf("Executing: deccopy(&num1, &num2)\n");
deccopy(&num1, &num2);
if (x = dectoasc(&num2, result, sizeof(result), -1))
{
printf("Error %d in converting num2 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("Destination = %s\n", result);

printf("\nDECCOPY Sample Program over.\n\n");
exit(0);
}
```

输出

```
DECCOPY Sample ESQL Program running.

String = 12345.6789
Executing: deccopy(&num1, &num2)
Destination = 12345.6789
```

DECCOPY Sample Program over.

5.2.23 deccvasc() 函数

deccvasc() 函数将以 C **char** 类型中作为可打印字符保存的值转换为 **decimal** 类型数值。

语法

mint deccvasc(*strng_val*, *len*, *dec_val*)

```
char *strng_val;
mint len;
dec_t *dec_val;
```

strng_val

指向要将其值 deccvasc() 转换为 **decimal** 值的字符串的指针。

len

strng_val 字符串的长度。

dec_val

指向 deccvasc() 将转换的结果放置其中的 **decimal** 结构的指针。

用法

字符串 *strng_val* 可包含下列符号：

前置符号，或正号（+）或负号（-）

小数点，以及小数点游标的数字

以 e 或 E 开头的指数。您可在指数前添加符号，或正号（+）或负号（-）。

deccvasc() 函数忽略字符串中开头的空格。

返回代码

0

转换成功。

-1200

该数值太大，以至于不能放入 **decimal** 类型结构内（溢出）。

-1201

该数值太小，以至于不能放入 **decimal** 类型结构内（下溢）。

-1213

该字符串有非数值字符。

-1216

该字符串有不良指数。

示例

demo 目录中的 deccvasc.ec 文件包含下列样例程序。

```
/*
        * deccvasc.ec *

The following program converts two strings to DECIMAL numbers and displays the values
stored in each field of the decimal structures.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string1[] = "-12345.6789";
        char string2[] = "480";

        main()
        {
        mint x;
        dec_t num1, num2;

        printf("DECCVASC Sample ESQL Program running.\n\n");

        if (x = deccvasc(string1, strlen(string1), &num1))
        {
    printf("Error %d in converting string1 to DECIMAL\n", x);
```

```
        exit(1);
        }
        if (x = deccvasc(string2, strlen(string2), &num2))
        {
    printf("Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
        }
        /*
        *    Display the exponent, sign value and number of digits in num1.
        */
        printf("\tstring1 = %s\n", string1);
        disp_dec("num1", &num1);

        /*
*   Display the exponent, sign value and number of digits in num2.
        */
        printf("\tstring2 = %s\n", string2);
        disp_dec("num2", &num2);

        printf("\nDECCVASC Sample Program over.\n\n");
        exit(0);
        }



        disp_dec(s, num)
        char *s;
        dec_t *num;
        {
        mint n;

        printf("%s dec_t structure:\n", s);
      printf("\tdec_exp = %d, dec_pos = %d, dec_ndgts = %d, dec_dgts: ",
        num->dec_exp, num->dec_pos, num->dec_ndgts);
```

```
        n = 0;
        while(n < num->dec_ndgts)
        printf("%02d ", num->dec_dgts[n++]);
        printf("\n\n");
        }
```

输出

```
DECCVASC Sample ESQL Program running.

        string1 =    -12345.6789
        num1 dec_t structure:
            dec_exp = 3, dec_pos =    0, dec_ndgts = 5,
            dec_dgts: 01 23 45 67 89

        string2 =    480
        num2 dec_t structure:
        dec_exp = 2, dec_pos =    1, dec_ndgts = 2,
        dec_dgts: 04 80

        DECCVASC Sample Program over.
```

5.2.24 deccvdbl() 函数

deccvdbl() 函数将 C double 类型数值转换为 decimal 类型数值。

语法

```
mint deccvdbl(dbl_val, np)
        double dbl_val;
        dec_t *dec_val;
```

dbl_val

deccvdbl() 转换为 **decimal** 类型值的 **double** 值。

dec_val

指向 deccvdbl() 在其中放置转换结果的 **decimal** 结构的指针。

结果代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的 deccvdbl.ec 文件包含下列样例程序。

```
/*
        * deccvdbl.ec *
    The following program converts two double type numbers to DECIMAL numbers and
displays the results.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char result[41];


        main()
        {
        mint x;
        dec_t num;
        double d = 2147483647;

        printf("DECCVDBL Sample ESQL Program running.\n\n");

        printf("Number 1 (double) = 1234.5678901234\n");
        if (x = deccvdbl((double)1234.5678901234, &num))
        {
      printf("Error %d in converting double1 to DECIMAL\n", x);
```

```
        exit(1);
        }
        if (x = dectoasc(&num, result, sizeof(result), -1))
        {
    printf("Error %d in converting DECIMAL1 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("    String Value = %s\n", result);

        printf("Number 2 (double) = $.1f\n", d);
        if (x = deccvdbl(d, &num))
        {
  printf("Error %d in converting double2 to DECIMAL\n", x);
        exit(1);
        }
        if (x = dectoasc(&num, result, sizeof(result), -1))
        {
    printf("Error %d in converting DECIMAL2 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("    String Value = %s\n", result);

        printf("\nDECCVDBL Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECCVDBL Sample ESQL Program running.

    Number 1 (double) = 1234.5678901234
    String Value = 1234.5678901234
    Number 2 (double) = 2147483647.0
```

String Value = 2147483647.0

DECCVDBL Sample Program over.

5.2.25 deccvflt() 函数

deccvflt() 函数将 C float 类型数值转换为 ESQL/C decimal 类型数值。

语法

int deccvflt(flt_val, dec_val)

float flt_val;

dec_t *dec_val;

flt_val

deccvflt() 转换为 decimal 类型值的 float 值。

dec_val

指向 deccvflt() 放置转换结果的 decimal 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

下列示例程序将两个 float 类型数值转换为 DECIMAL 数值，并显示结果。

```
#include <stdio.h>
        EXEC SQL include decimal;
        char result[41];
        main()
        {
        int x;
        dec_t num;
        float f = 2147483674;
        printf(“DECCVFLT Sample ESQL Program Running.\n\n);
        if (x = deccvflt((float)1234.5678901234, &num))
```

```
        {
    printf(“Error %d in converting double1 to DECIMAL\n”, x);
        exit(1);
        }
        if (x = dectoasc(&num, result, sizeof(result), -1))
        {
    printf(“Error %d in converting DECIMAL1 to string\n”, x);
        exit(1);
        }
        result[40] = ‘\0’;
        printf(“ String Value = %s\n”, result);
        printf(“ Number 2 (float) = %.1f\n”, f);
        if (x = deccvflt(f, &num))
        {
    printf(“Error %d in converting float2 to DECIMAL\n”, x);
        exit(1);
        }
       if (x = dectoasc(&num, result, sizeof(result), -1))
        {
    printf(“Error %d in converting DECIMAL2 to string\n”, x);
        exit(1);
        }
        result[40] = ‘\0’;
        printf(“ String Value = %s\n”, result);
        printf(“\n DECCVFLT Sample Program Over.\n\n”);
        exit(0);
        }
```

输出

```
DECCVFLT Sample ESQL Program running.

    Number 1 (float) = 1234.5678901234
    String Value = 1234.56787
```

Number 2 (float) = 2147483647.0

String Value = 2147483647.0

DECCVFLT Sample Program over.

5.2.26 deccvint() 函数

deccvint() 函数将 C int 类型数值转换为 decimal 类型数值。

语法

mint deccvint(*int_val*, *dec_val*)

mint int_val;

dec_t *dec_val;

int_val

deccvint() 转换为 **decimal** 类型值的 **mint** 值。

dec_val

指向 deccvint() 放置转换结果的 **decimal** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的 deccvint.ec 文件包含下列样例程序。

```
/*
        * deccvint.ec *

    The following program converts two integers to DECIMAL numbers and displays the
results. */

        #include <stdio.h>
```

```
EXEC SQL include decimal;

char result[41];

main()
{
mint x;
dec_t num;

printf("DECCVINT Sample ESQL Program running.\n\n");

printf("Integer 1 = 129449233\n");
if (x = deccvint(129449233, &num))
{
printf("Error %d in converting int1 to DECIMAL\n", x);
exit(1);
}
if (x = dectoasc(&num, result, sizeof(result), -1))
{
printf("Error %d in converting DECIMAL to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    String for Decimal Value = %s\n", result);

printf("Integer 2 = 33\n");
if (x = deccvint(33, &num))
{
printf("Error %d in converting int2 to DECIMAL\n", x);
exit(1);
}
result[40] = '\0';
printf("    String for Decimal Value = %s\n", result);
```

```
        printf("\nDECCVINT Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECCVINT Sample ESQL Program running.

        Integer 1 = 129449233
        String for Decimal Value = 129449233.0
        Integer 2 = 33
        String for Decimal Value = 33.0

DECCVINT Sample Program over.
```

5.2.27 deccvlong() 函数

deccvlong() 函数将 C long 类型值转换为 decimal 类型值。

语法

```
mint deccvlong(lng_val, dec_val)
        int4 lng_val;
        dec_t *dec_val;
```

lng_val

deccvlong() 转换为 **decimal** 类型值的 **int4** 值。

dec_val

指向 deccvlong() 放置转换结果的 **decimal** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 deccvlong.ec 包含下列样例程序。

```
/*
        * deccvlong.ec *

    The following program converts two longs to DECIMAL numbers and displays the
results.  */

        #include <stdio.h>

        EXEC SQL include decimal;
      char result[41];
        main()
        {
        mint x;
        dec_t num;

        int4 n;

        printf("DECCVLONG Sample ESQL Program running.\n\n");

        printf("Long Integer 1 = 129449233\n");
        if (x = deccvlong(129449233L, &num))
        {
        printf("Error %d in converting long to DECIMAL\n", x);
        exit(1);
        }
        if (x = dectoasc(&num, result, sizeof(result), -1))
        {
    printf("Error %d in converting DECIMAL to string\n", x);
        exit(1);
        }
        result[40] = '\0';
```

```
    printf("    String for Decimal Value = %s\n", result);

    n = 2147483646;                                    /* set n */
    printf("Long Integer 2 = %d\n", n);
    if (x = deccvlong(n, &num))
    {
printf("Error %d in converting long to DECIMAL\n", x);
    exit(1);
    }
    if (x = dectoasc(&num, result, sizeof(result), -1))
    {
printf("Error %d in converting DECIMAL to string\n", x);
    exit(1);
    }
    result[40] = '\0';
    printf("    String for Decimal Value = %s\n", result);

    printf("\nDECCVLONG Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
DECCVLONG Sample ESQL Program running.

    Long Integer 1 = 129449233
    String for Decimal Value =    129449233.0
    Long Integer 2 = 2147483646
    String for Decimal Value = 2147483646.0
DECCVLONG Sample Program over.
```

5.2.28 decdiv() 函数

decdiv() 函数将两个 decimal 类型值相除。

语法

mint decdiv(*n1, n2, result*)　　/* result = n1 / n2 */

dec_t *n1;

dec_t *n2;

dec_t *result;

n1

指向第一个运算对象的 **decimal** 结构的指针。

n2

指向第二个运算对象的 **decimal** 结构的指针。

quotient

指向包含 *n1* 除以 *n2* 的商的 **decimal** 结构的指针。

语法

quotient 可与 n1 或 n2 相同。

返回代码

0

操作成功。

-1200

操作导致溢出。

-1201

操作导致下溢。

-1202

操作尝试除零。

示例

demo 目录中的文件 decdiv.ec 包含下列样例程序。

/*

* decdiv.ec *

The following program divides two DECIMAL numbers and displays the result.

```
*/

#include <stdio.h>

EXEC SQL include decimal;

char string1[] = "480";
char string2[] = "80";
char result[41];

main()
{
mint x;
dec_t num1, num2, dvd;

printf("DECDIV Sample ESQL Program running.\n\n");

if (x = deccvasc(string1, strlen(string1), &num1))
{
  printf("Error %d in converting string1 to DECIMAL\n", x);
exit(1);

}
if (x = deccvasc(string2, strlen(string2), &num2))
{
printf("Error %d in converting string2 to DECIMAL\n", x);
exit(1);
}
if (x = decdiv(&num1, &num2, &dvd))
{
printf("Error %d in converting divide num1 by num2\n", x);
exit(1);
}
```

```
    if (x = dectoasc(&dvd, result, sizeof(result), -1))
    {
    printf("Error %d in converting dividend to string\n", x);
    exit(1);
    }
    result[40] = '\0';
    printf("\t%s / %s = %s\n", string1, string2, result);

    printf("\nDECDIV Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
DECDIV Sample ESQL Program running.

        480 / 80 = 6.0

DECDIV Sample Program over.
```

5.2.29 dececvt() 和 decfcvt() 函数

dececvt() 和 decfcvt() 函数相似，对于“UNIX(TM) 程序员手册”的第三部分中 ECVT(3) 之下的子例程。dececvt() 函数的工作方式与 ecvt(3) 函数相同，decfcvt() 函数的工作方式与 fcvt(3) 函数相同。它们都将 decimal 类型数值转换为 C char 类型值。

语法

```
char *dececvt(dec_val, ndigit, decpt, sign)
        dec_t *dec_val;
        mint ndigit;
        mint *decpt;
        mint *sign;

        char *decfcvt(dec_val, ndigit, decpt, sign)
        dec_t *dec_val;
        mint ndigit;
        mint *decpt;
```

mint *sign;

dec_val

指向包含您想要这些函数来转换的 **decimal** 值的 **decimal** 结构的指针。

ndigit

dececvt() 的 ASCII 字符串的长度。它是 decfcvt() 的小数点右边的数字的数目。

decpt

指向一个整数的指针，该整数是小数点相对于字符串起点的位置。*decpt 的负值或零意味着在返回的数字的左边。

sign

指向结果的符号的指针。 如果结果的符号为负，则 *sign 非零；否则，*sign 为零。

用法

dececvt() 函数将 *np* 指向的 **decimal** 值转换为 *ndigit* ASCII 数字的以空结尾的字符串，并返回指向该字符串的指针。对此函数的后续调用重写该字符串。

dececvt() 函数对低位数字四舍五入。

decfcvt() 函数与 dececvt() 相同，除了 *ndigit* 指定小数点右边的数字的数目，而不是数字的总数目之外。

让 *dec_val* 指向 12345.67 的 **decimal** 值，并阻止除了 *ndigit* 之外的所有参数。对于四个不同的 *ndigit* 值，下表展示 dececvt() 函数返回的值。

ndigit 值	返回字符串	*decpt 值	*sign
4	"1235"	5	0
10	"1234567000"	5	0
1	"1"	5	0
3	"123"	5	0

要获取 *dec_val* 和 *ndigit* 值的更多示例，请参阅 dececvt() 的示例上 dececvt.ec 演示程序的样例输出。

重要： 当您编写线程安全 GBase 8s ESQL/C 应用程序时，请不要使用 dececvt()

或 decfcvt() 库函数。反而，请使用它们的线程安全等价的 ifx_dececvt() 和 ifx_decfcvt() 函数。

dececvt() 的示例

demo 目录中的文件 dececvt.ec 包含下列样例程序。

```
/*
    * dececvt.ec *

    The following program converts a series of DECIMAL numbers to fixedm    strings of 20
ASCII digits.    For each conversion it displays the resulting    string, the decimal position from
the beginning of the string and the sign value.
                */

                #include <stdio.h>

                EXEC SQL include decimal;



                char *strings[] =
                {
                "210203.204",
                "4894",
                "443.334899312",
                "-12344455",
                "12345.67",
                ".001234",
                0
                };

                char result[40];

                main()
                {
```

```
                mint x;
                mint i = 0, f, sign;
                dec_t num;
                char *dp, *dececvt();

                printf("DECECVT Sample ESQL Program running.\n\n");
                while(strings[i])
                {
               if (x = deccvasc(strings[i], strlen(strings[i]), &num))
                {
    printf("Error %d in converting string [%s] to DECIMAL\n",x,    strings[i]);
                break;
                }
               printf("\Input string[%d]: %s\n", i, strings[i]);

          /* to 20-char ASCII string */

          dp = dececvt(&num, 20, &f, &sign);
               printf(" Output of dececvt(&num, 20, ...): %c%s    decpt: %d    sign: %d\n",
(sign ? '-' : '+'), dp, f, sign);

               /* display result */

               /* to 10-char ASCII string */

             dp = dececvt(&num, 10, &f, &sign);
    printf(" Output of dececvt(&num, 10, ...): %c%s    decpt: %d    sign: %d\n",(sign ? '-' : '+'),
dp, f, sign);

    /* to 4-char ASCII string */

               dp = dececvt(&num, 4, &f, &sign);
```

```
                /* display result */
                   printf(" Output of dececvt(&num, 4, ...): %c%s    decpt: %d
sign: %d\n",(sign ? '-' : '+'), dp, f, sign);

                /* to 3-char ASCII string */
        dp = dececvt(&num, 3, &f, &sign);

                /* display result */
        printf(" Output of dececvt(&num, 3, ...): %c%s    decpt: %d    sign: %d\n", (sign ? '-' : '+'),
dp, f, sign);

                 /* to 1-char ASCII string */
                 dp = dececvt(&num, 1, &f, &sign);

                 /* display result */
          printf(" Output of dececvt(&num, 1, ...): %c%s decpt:
          %d    sign: %d\n",(sign ? '-' : '+'), dp, f, sign);

                   ++i;       /* next string */
                   }
                   printf("\nDECECVT Sample Program over.\n\n");
                   }
```

dececvt() 的输出

```
DECECVT Sample ESQL Program running.

                Input string[0]: 210203.204
                Output of dececvt: +2102    decpt: 6    sign: 0
                Output of dececvt: +2102032040    decpt: 6    sign: 0
                Output of dececvt: +2    decpt: 6    sign: 0
                Output of dececvt: +210    decpt: 6    sign: 0

                Input string[1]: 4894
                Output of dececvt: +4894    decpt: 4    sign: 0
```

```
Output of dececvt: +4894000000    decpt: 4    sign: 0
Output of dececvt: +5    decpt: 4    sign: 0
Output of dececvt: +489    decpt: 4    sign: 0

Input string[2]: 443.334899312
Output of dececvt: +4433    decpt: 3    sign: 0
Output of dececvt: +4433348993    decpt: 3    sign: 0
Output of dececvt: +4    decpt: 3    sign: 0
Output of dececvt: +443    decpt: 3    sign: 0

Input string[3]: -12344455
Output of dececvt: -1234    decpt: 8    sign: 1
Output of dececvt: -1234445500    decpt: 8    sign: 1
Output of dececvt: -1    decpt: 8    sign: 1
Output of dececvt: -123    decpt: 8    sign: 1

Input string[4]: 12345.67
Output of dececvt: +1235    decpt: 5    sign: 0
Output of dececvt: +1234567000    decpt: 5    sign: 0
Output of dececvt: +1    decpt: 5    sign: 0
Output of dececvt: +123    decpt: 5    sign: 0

Input string[5]: .001234
Output of dececvt: +1234    decpt: -2    sign: 0
Output of dececvt: +1234000000    decpt: -2    sign: 0
Output of dececvt: +1    decpt: -2    sign: 0
Output of dececvt: +123    decpt: -2    sign: 0
```

DECECVT Sample Program over.

decfcvt() 的示例

demo 目录中的文件 decfcvt.ec 包含下列示例程序。

```
/*
        * decfcvt.ec *
```

The following program converts a series of DECIMAL numbers to strings of ASCII digits with 3 digits to the right of the decimal point. For each conversion it displays the resulting string, the position of the decimal point from the beginning of the string and the sign value. */

```
#include <stdio.h>

EXEC SQL include decimal;

char *strings[] =
{
"210203.204",
"4894",
"443.334899312",
"-12344455",
0
};

main()
{
mint x;
dec_t num;
mint i = 0, f, sign;
char *dp, *decfcvt();

printf("DECFCVT Sample ESQL Program running.\n\n");

while(strings[i])
{
if (x = deccvasc(strings[i], strlen(strings[i]), &num))
{
printf("Error %d in converting string [%s] to DECIMAL\n",  x, strings[i]);
```

```
                break;
                }

    dp = decfcvt(&num, 3, &f, &sign);           /* to ASCII string */

                /* display result */
                printf("Input string[%d]: %s\n", i, strings[i]);
                printf("    Output of decfcvt: %c%*.*s.%s    decpt: %d    sign: %d\n\n", (sign ?
'-' : '+'), f, f, dp, dp+f, f, sign);
                ++i;                            /* next string */
                }
                printf("\nDECFCVT Sample Program over.\n\n");
                }
```

decfcvt() 的输出

```
DECFCVT Sample ESQL Program running.

        Input    string[0]: 210203.204
        Output of decfcvt: +210203.204    decpt: 6    sign: 0

        Input    string[1]: 4894
        Output of decfcvt: +4894.000    decpt: 4    sign: 0

        Input    string[2]: 443.334899312
        Output of decfcvt: +443.335    decpt: 3    sign: 0

        Input    string[3]: -12344455
        Output of decfcvt: -12344455.000    decpt: 8    sign: 1
    DECFCVT Sample Program over.
```

5.2.30 decmul() 函数

decmul() 函数将两个 decimal 类型值相乘。

语法

mint decmul(*n1*, *n2*, *product*)

```
    dec_t *n1;
    dec_t *n2;
    dec_t *product;
```

n1

指向第一个运算对象的 **decimal** 结构的指针。

n2

指向第二个运算对象的 **decimal** 结构的指针。

product

指向包含 *n1* 乘以 *n2* 的积的 **decimal** 结构的指针。

用法

product 可与 *n1* 或 *n2* 相同。

返回代码

0

操作成功。

-1200

操作导致溢出。

-1201

操作导致下溢。

示例

demo 目录中的 decmul.ec 文件包含下列示例程序。

```
/*
    * decmul.ec *

    This program multiplies two DECIMAL numbers and displays the result.
*/

#include <stdio.h>

EXEC SQL include decimal;
```

```
char string1[] = "80.2";
char string2[] = "6.0";
char result[41];

main()
{
        mint x;
        dec_t num1, num2, mpx;

        printf("DECMUL Sample ESQL Program running.\n\n");

        if (x = deccvasc(string1, strlen(string1), &num1))
            {
            printf("Error %d in converting string1 to DECIMAL\n", x);
            exit(1);
            }
        if (x = deccvasc(string2, strlen(string2), &num2))
            {
            printf("Error %d in converting string2 to DECIMAL\n", x);
            exit(1);
            }
      if (x = decmul(&num1, &num2, &mpx))
            {
            printf("Error %d in converting multiply\n", x);
            exit(1);
            }
        if (x = dectoasc(&mpx, result, sizeof(result), -1))
            {
      printf("Error %d in converting mpx to display string\n", x);
            exit(1);
            }
        result[40] = '\0';
```

```
        printf("\t%s * %s = %s\n", string1, string2, result);

        printf("\nDECMUL Sample Program over.\n\n");
        exit(0);
}
```

输出

```
DECMUL Sample ESQL Program running.

    80.2 * 6.0 = 481.2

DECMUL Sample Program over.
```

5.2.31 decround() 函数

decround() 函数将 decimal 类型数值四舍五入到小数数位。

语法

```
void decround(d, s)
        dec_t *d;
        mint s;
```

d

指向 decround() 函数四舍五入其值的 **decimal** 结构的指针。

s

decround() 对 *d* 四舍五入的小数数字的数目。请使用 *s* 参数的整数值。

用法

四舍五入因子为 $5x10^{-s-1}$。要四舍五入一个值，decround() 函数对正数值加上舍入因子，或从负数值减去此因子。然后，它截断至 *s* 位，如下表所示。

四舍五入之前的值	*s* 的值	四舍五入的值
1.4	0	1.0
1.5	0	2.0
1.684	2	1.68
1.685	2	1.69
1.685	1	1.7
1.685	0	2.0

示例

demo 目录中的文件 decround.ec 包含下列样例程序。

```
/*
        * decround.ec *

The following program rounds a DECIMAL type number six times and displays the result
of each operation.        */
        #include <stdio.h>

          EXEC SQL include decimal;

          char string[] = "-12345.038572";
          char result[41];

          main()
          {
          mint x;
          mint i = 6;    /* number of decimal places to start with */
          dec_t num1;

          printf("DECROUND Sample ESQL Program running.\n\n");

          printf("String = %s\n", string);
          while(i)
          {
          if (x = deccvasc(string, strlen(string), &num1))
          {
         printf("Error %d in converting string to DECIMAL\n", x);
          break;
          }
          decround(&num1, i);
    if (x = dectoasc(&num1, result, sizeof(result), -1))
```

```
        {
        printf("Error %d in converting result to string\n", x);
        break;
        }
        result[40] = '\0';
printf("    Rounded to %d Fractional Digits: %s\n", i--, result);
        }
        printf("\nDECROUND Sample Program over.\n\n");
        }
```

输出

```
DECROUND Sample ESQL Program running.

        String = -12345.038572
        Rounded to 6 Fractional Digits: -12345.038572
        Rounded to 5 Fractional Digits: -12345.03857
        Rounded to 4 Fractional Digits: -12345.0386
        Rounded to 3 Fractional Digits: -12345.039
        Rounded to 2 Fractional Digits: -12345.04
        Rounded to 1 Fractional Digits: -12345.

 DECROUND Sample Program over.
```

5.2.32 decsub() 函数

decsub() 函数将两个 decimal 类型值相减。

语法

```
mint decsub(n1, n2, difference)
        dec_t *n1;
        dec_t *n2;
        dec_t *difference;
```

n1

指向第一个运算对象的 **decimal** 结构的指针。

n2

指向第二个运算对象的 **decimal** 结构的指针。

difference

指向包含 *n1* 减去 *n2* 的差的 **decimal** 结构的指针。

用法

difference 可与 n1 或 n2 相同。

返回代码

0

操作成功。

-1200

操作导致溢出。

-1201

操作导致下溢。

示例

demo 目录中的文件 decsub.ec 包含下列示例程序。

```
/*
        * decsub.ec *

  The following program subtracts two DECIMAL numbers and displays the result.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string1[] = "1000.038782";
        char string2[] = "480";
        char result[41];

        main()
```

```
        {
        mint x;
        dec_t num1, num2, diff;

        printf("DECSUB Sample ESQL Program running.\n\n");

        if (x = deccvasc(string1, strlen(string1), &num1))
        {
    printf("Error %d in converting string1 to DECIMAL\n", x);
        exit(1);
        }
        if (x = deccvasc(string2, strlen(string2), &num2))
        {
    printf("Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
        }
        if (x = decsub(&num1, &num2, &diff))
        {
        printf("Error %d in subtracting decimals\n", x);
        exit(1);
        }
        if (x = dectoasc(&diff, result, sizeof(result), -1))
        {
        printf("Error %d in converting result to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("\t%s - %s = %s\n", string1, string2, result);

        printf("\nDECSUB Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECSUB Sample ESQL Program running.

        1000.038782 - 480 = 520.038782

    DECSUB Sample Program over.
```

5.2.33 dectoasc() 函数

dectoasc() 函数将 decimal 类型数值转换为 C char 类型值。

语法

```
mint dectoasc(dec_val, strng_val, len, right)
        dec_t *dec_val;
        char    *strng_val;
        mint     len;
        mint     right;
```

dec_val

指向 dectoasc() 将其值转换为文本字符串的 **decimal** 结构的指针。

strng_val

指向 dectoasc() 函数放置文本字符串所在的字符缓冲区的第一个字节的指针。

len

以字节计的 strng_val 的大小，对于空终止符，为负 1。

right

指示小数点右边小数位的数目的整数。

用法

如果 *right* = -1，*dec_val* 的小数确定小数位的数目。

如果 **decimal** 数目不适于长度 *len* 的字符串，则 **dectoasc()()** 将该数值转换为指数表示方法。如果该数目仍不适合，则 dectoasc() 以星号填充该字符串。如果该数目比字符串段，则 dectoasc() 左向调整该数值，并以空格补在右边。

由于 **dectoasc()()** 返回的字符串不是以空结尾的，因此，在您打印它之前，您的程序必须将空字符添加到该字符串。

返回代码

0

转换成功。

-1

转换失败。

示例

demo 目录中的文件 dectoasc.ec 包含下列样例程序。

```
/*
        * dectoasc.ec *

The following program converts DECIMAL numbers to strings of varying sizes.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        #define    END    sizeof(result)

        char string1[] = "-12345.038782";
        char string2[] = "480";
        char result[40];

        main()
        {
        mint x;
        dec_t num1, num2;

        printf("DECTOASC Sample ESQL Program running.\n\n");
```

```
printf("String Decimal Value 1 = %s\n", string1);
if (x = deccvasc(string1, strlen(string1), &num1))
{
printf("Error %d in converting string1 to DECIMAL\n", x);
exit(1);
}
printf("String Decimal Value 2 = %s\n", string2);
if (x = deccvasc(string2, strlen(string2), &num2))
{
printf("Error %d in converting string2 to DECIMAL\n", x);
exit(1);
}

printf("\nConverting Decimal back to ASCII\n");
printf("    Executing: dectoasc(&num1, result, 5, -1)\n");
if (x = dectoasc(&num1, result, 5, -1))
printf("\tError %d in converting DECIMAL1 to string\n", x);
else
{
result[5] = '\0';                       /* null terminate */
printf("\tResult ='%s'\n", result);
}

printf("Executing: dectoasc(&num1, result, 10, -1)\n");
if (x = dectoasc(&num1, result, 10, -1))
printf("Error %d in converting DECIMAL1 to string\n", x);
else
{
result[10] = '\0';                        /* null terminate */
printf("\tResult = '%s'\n", result);
}

printf("Executing: dectoasc(&num2, result, END, 3)\n");
```

```
    if (x = dectoasc(&num2, result, END, 3))
    printf("\tError %d in converting DECIMAL2 to string\n", x);
    else
    {
    result[END-1] = '\0';                    /* null terminate */
    printf("\tResult = '%s'\n", result);
    }

    printf("\nDECTOASC Sample Program over.\n\n")
    }
```

输出

```
DECTOASC Sample ESQL Program running.

        String Decimal Value 1 = -12345.038782
        String Decimal Value 2 = 480

        Converting Decimal back to ASCII
        Executing: dectoasc(&num1, result, 5, -1)
        Error -1 in converting decimal1 to string
        Executing: dectoasc(&num1, result, 10, -1)
        Result = '-12345.039'
        Executing: dectoasc(&num2, result, END, 3)
        Result = '480.000'
DECTOASC Sample Program over.
```

5.2.34 dectodbl() 函数

dectodbl() 函数将 decimal 类型数值转换为 C double 类型值。

语法

```
mint dectodbl(dec_val, dbl_val)
        dec_t *dec_val;
        double *dbl_val;
```

dec_val

指向 dectodbl() 将其值转换为 **double** 类型值的 **decimal** 结构的指针。

dbl_val

指向 dectodbl() 放置转换的结果处的 **double** 类型的指针。

用法

在 **decimal** 类型数值转换为 **double 类型数值** 过程中，主计算机的浮点格式可导致精度的损失。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的 dectodbl.ec 文件包含下列样例程序。

```
/*
        * dectodbl.ec *

The following program converts two DECIMAL numbers to doubles and displays the results.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string1[] = "2949.3829398204382";
        char string2[] = "3238299493";
        char result[40];

        main()
```

```
        {
        mint x;
        double d = 0;
        dec_t num;

        printf("DECTODBL Sample ESQL Program running.\n\n");

        if (x = deccvasc(string1, strlen(string1), &num))
        {
printf("Error %d in converting string1 to DECIMAL\n", x);
        exit(1);
        }
        if (x = dectodbl(&num, &d))
        {
 printf("Error %d in converting DECIMAL1 to double\n", x);
        exit(1);
        }
        printf("String 1 = %s\n", string1);
        printf("Double value = %.15f\n", d);

        if (x = deccvasc(string2, strlen(string2), &num))
        {
 printf("Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
        }
        if (x = dectodbl(&num, &d))
        {
 printf("Error %d in converting DECIMAL2 to double\n", x);
        exit(1);
        }
        printf("String 2 = %s\n", string2);
        printf("Double value = %f\n", d);
```

```
        printf("\nDECTODBL Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECTODBL Sample ESQL Program running.

        String 1 = 2949.3829398204382
        Double value = 2949.382939820438423

        String 2 = 3238299493
        Double value = 3238299493.000000

 DECTODBL Sample Program over.
```

5.2.35 dectoint() 函数

dectoint() 函数将 decimal 类型数值转换为 C int 类型数值。

语法

```
mint dectoint(dec_val, int_val)
        dec_t *dec_val;
        mint *int_val;
```

dec_val

指向 dectoint() 将其值转换为 **mint** 类型值的 **decimal** 结构的指针。

int_val

指向 dectoint() 放置转换的结果处的 **mint** 值的指针。

用法

dectoint() 库函数将 **decimal** 值转换为 C 整数。该 C 整数的大小依赖于您正在使用的计算机的硬件和操作系统。因此，**dectoint()** 函数将整数值等同于 SQL SMALLINT 数据类型。SMALLINT 有效范围在 32767 与 -32767 之间。要将较大的 **decimal** 转换为较大的整数，请使用 dectoint() 库函数。

返回代码

0

转换成功。

<0

转换失败。

-1200

decimal 类型数值的大小大于 32767。

示例

demo 目录中的文件 dectoint.ec 包含下列样例程序。

```
/*
        * dectoint.ec *

The following program converts two DECIMAL numbers to integers and displays the result
of each conversion.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string1 [] = "32767";
        char string2 [] = "32768";

        main()
        {
        mint x;
        mint n = 0;
        dec_t num;

        printf("DECTOINT Sample ESQL Program running.\n\n)";

        printf("String 1 = %s\n", string1);
```

```
        if (x = deccvasc(string1,strlen(string1), &num))
        {
    printf("   Error %d in converting string1 to decimal\n", x);
        exit(1);
        }
        if (x = dectoint(&num, &n))
        printf("    Error %d in converting decimal to int\n", x);
        else
        printf("    Result = %d\n", n);

        printf("\nString 2 = %s\n", string2);
        if (x = deccvasc(string2, strlen(string2), &num))
        {
  printf("    Error %d in converting string2 to decimal\n", x);
        exit(1);
        }
        if (x = dectoint(&num, &n))
        printf("    Error %d in converting decimal to int\n", x);
        else
        printf("    Result = %d\n", n);

        printf("\nDECTOINT Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECTOINT Sample ESQL Program running.

        String 1 = 32767
        Result = 32767

        String 2 = 32768
        Error -1200 in converting decimal to int
```

DECTOINT Sample Program over.

5.2.36 dectolong() 函数

dectolong() 函数将 decimal 类型数值转换为 int4 类型数值。

语法

mint dectolong(*dec_val*, *lng_val*)

```
dec_t *dec_val;
int4   *lng_val;
```

dec_val

指向 dectolong() 将其值转换为 **int4** 整数的 **decimal** 结构的指针。

lng_val

指向 dectolong() 放置转换的结果处的 **int4** 整数的指针。

返回代码

0

转换成功。

-1200

decimal 类型数值的大小大于 2,147,483,647。

示例

demo 目录中的文件 dectolong.ec 包含下列样例程序。

```
/*
        * dectolong.ec *
```

The following program converts two DECIMAL numbers to longs and displays the return value and the result for each conversion.

```
        */

        #include <stdio.h>

        EXEC SQL include decimal;
```

```
char string1[] = "2147483647";
char string2[] = "2147483648";

main()
{
int x;
long n = 0;
dec_t num;

printf("DECTOLONG Sample ESQL Program running.\n\n");

printf("String 1 = %s\n", string1);
if (x = deccvasc(string1, strlen(string1), &num))
{
printf("    Error %d in converting string1 to DECIMAL\n", x);
exit(1);
 }
 if (x = dectolong(&num, &n))
 printf("    Error %d in converting DECIMAL1 to long\n", x);
 else
 printf("    Result = %ld\n", n);

 printf("\nString 2 = %s\n", string2);
 if (x = deccvasc(string2, strlen(string2), &num))
 {
 printf(" Error %d in converting string2 to DECIMAL\n", x);
 exit(1);
 }
 if (x = dectolong(&num, &n))
 printf(" Error %d in converting DECIMAL2 to long\n", x);
 else
 printf("    Result = %ld\n", n);
```

```
        printf("\nDECTOLONG Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
DECTOLONG Sample ESQL Program running.

        String 1 = 2147483647
        Result = 2147483647

        String 2 = 2147483648
        Error -1200 in converting DECIMAL2 to long

 DECTOLONG Sample Program over.
```

5.2.37 dectrunc() 函数

dectrunc() 函数将四舍五入了的 decimal 类型数值截断到小数位。

语法

```
void dectrunc(d, s)
        dec_t *d;
        mint s;
```

d

指向 dectrunc() 截断其值的四舍五入了的数值的 **decimal** 结构的指针。

s

dectrunc() 将该值截断至的小数位的数目。对于此参数，请使用正数或零。

用法

下表展示以不同的输入，来自 dectrunc() 的样例输出。

截断之前的值	*s* 的值	截断了的值
1.4	0	1.0

截断之前的值	*s* 的值	截断了的值
1.5	0	1.0
1.684	2	1.68
1.685	2	1.68
1.685	1	1.6
1.685	0	1.0

示例

demo 目录中的文件 dectrunc.ec 包含下列样例程序。

```
/*
        * dectrunc.ec *

The following program truncates a DECIMAL number six times and displays each result.
        */

        #include <stdio.h>

        EXEC SQL include decimal;

        char string[] = "-12345.038572";
        char result[41];

        main()
        {
        mint x;
        mint i = 6;       /* number of decimal places to start with */
        dec_t num1;

        printf("DECTRUNC Sample ESQL Program running.\n\n");

        printf("String = %s\n", string);
        while(i)
```

```
    {
    if (x = deccvasc(string, strlen(string), &num1))
    {
   printf("Error %d in converting string to DECIMAL\n", x);
    break;
    }
    dectrunc(&num1, i);
    if (x = dectoasc(&num1, result, sizeof(result), -1))
    {
    printf("Error %d in converting result to string\n", x);
    break;
    }
    result[40] = '\0';
    printf("Truncated to %d Fractional Digits: %s\n", i--,          result);
    }

    printf("\nDECTRUNC Sample Program over.\n\n");
    }
```

输出

```
DECTRUNC Sample ESQL Program running.

        String = -12345.038572
        Truncated to 6 Fractional Digits: -12345.038572
        Truncated to 5 Fractional Digits: -12345.03857
        Truncated to 4 Fractional Digits: -12345.0385
        Truncated to 3 Fractional Digits: -12345.038
        Truncated to 2 Fractional Digits: -12345.03
        Truncated to 1 Fractional Digits: -12345.0

DECTRUNC Sample Program over.
```

5.2.38 dtaddinv() 函数

dtaddinv() 函数将 interval 值加到 datetime 值。结果为 datetime 值。

语法

mint dtaddinv(*dt, inv, res*)

dtime_t *dt;

intrvl_t *inv;

dtime_t *res;

dt

指向初始化的 **datetime** 主变量的指针。

inv

指向初始化的 **interval** 主变量的指针。

res

指向包含结果的 **datetime** 主变量的指针。

用法

dtaddinv() 函数将 *inv* 中的**interval** 值加至 *dt* 中的 **datetime** 值，并将 **datetime** 值存储在 *res* 中。此结果继承 *dt* 的限定符。

该 interval 值必须在 year to month 或 day to fraction(5) 范围中。

datetime 值必须包括在 **interval** 值中出现的所有字段。

如果您未初始化变量 *dt* 和 *inv*，则该函数可能返回不可预计的结果。

返回代码

0

加法成功。

<0

加法错误。

示例

demo 目录在 dtaddinv.ec 文件中包含此样例程序。

/*

```
* dtaddinv.ec *
```

The following program adds an INTERVAL value to a DATETIME value and displays the result.

```
    */

#include <stdio.h>

EXEC SQL include datetime;

main()
{
char out_str[16];

EXEC SQL BEGIN DECLARE SECTION;
datetime year to minute dt_var, result;
interval day to minute intvl;
EXEC SQL END DECLARE SECTION;

printf("DTADDINV Sample ESQL Program running.\n\n");

printf("datetime year to minute value=2006-11-28 11:40\n");
dtcvasc("2006-11-28 11:40", &dt_var);
printf("interval day to minute value = 50 10:20\n");
incvasc("50 10:20", &intvl);

dtaddinv(&dt_var, &intvl, &result);

/* Convert to ASCII for displaying */
   dttoasc(&result, out_str);
            printf("---------------------------------------\n");
   printf("Sum=%s\n", out_str);
```

```
        printf("\nDTADDINV Sample Program over.\n\n");
        }
```

输出

```
DTADDINV Sample ESQL Program running.

        datetime year to minute value=2006-11-28 11:40
        interval day to minute value =          50 10:20
        ---------------------------------------------
        Sum=2007-01-17 22:00
  DTADDINV Sample Program over.
```

5.2.39 dtcurrent() 函数

dtcurrent() 函数将当前的日期和时间赋给 datetime 变量。

语法

```
void dtcurrent(d)
        dtime_t *d;
```

d

指向初始化的 **datetime** 主变量的指针。

用法

当将变量限定符设置为零（或任何无效的限定符）时，dtcurrent() 函数以 **year to fraction(3)** 限定符来初始化它。

当该变量包含有效的限定符时，dtcurrent() 函数扩展当前的日期和时间来与该限定符保持一致。

示例调用

下列语句将主变量 **timewarp** 设置为当前的日期：

```
EXEC SQL BEGIN DECLARE SECTION;
        datetime year to day timewarp;
        EXEC SQL END DECLARE SECTION;
      dtcurrent(&timewarp);
```

下列语句将变量 **now** 设置为当前的时间，最接近于毫秒：

```
now.dt_qual = TU_DTENCODE(TU_HOUR,TU_F3);
        dtcurrent(&now);
```

示例

demo 目录在 dtcurrent.ec 文件中包含此样例程序。

```
/*
        * dtcurrent.ec *
```

The following program obtains the current date from the system, converts it to ASCII and prints it.

```
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        mint x;
        char out_str[20];

        EXEC SQL BEGIN DECLARE SECTION;
        datetime year to hour dt1;
        EXEC SQL END DECLARE SECTION;

        printf("DTCURRENT Sample ESQL Program running.\n\n");

        /* Get today's date */
        dtcurrent(&dt1);

        /* Convert to ASCII for displaying */
```

```
    dttoasc(&dt1, out_str);
    printf("\tToday's datetime (year to minute) value is %s\n", out_str);

    printf("\nDTCURRENT Sample Program over.\n\n");
    }
```

输出

```
DTCURRENT Sample ESQL Program running.
 Today's datetime (year to minute) value is 2007-09-16 14:49
DTCURRENT Sample Program over.
```

5.2.40 dtcvasc() 函数

dtcvasc() 函数将符合 ANSI SQL 标准的 DATETIME 值的字符串转换为 **datetime** 值。

要获取关于 ANSI SQL DATETIME 标准的信息，请参阅 DATETIME 和 INTERVAL 值的 ANSI SQL 标准。

语法

```
mint dtcvasc(inbuf, dtvalue)
        char *inbuf;
        dtime_t *dtvalue;
```

inbuf

指向包含 ANSI 标准 DATETIME 字符串的缓冲区的指针。

dtvalue

指向初始化的 **datetime** 变量的指针。

用法

您必须以您想要此变量拥有的限定符，来初始化 *dtvalue* 中的 **datetime** 变量。

inbuf 中的字符串必须有符合 **year to second** 限定符的值，该限定符的格式为 ANSI SQL。*inbuf* 字符串可有开头的和结尾的空格。然而，从第一个有效位至最后一个，*inbuf* 仅可包含符合 ANSI SQL 标准的 DATETIME 值的数字和定界符的字符。

如果您指定年份值作为一个或两个数字，则 dtcvasc() 函数假定该年份在当前世纪中。您可设置该 DBCENTURY 环境变量来确定，当您省略来自该日期的世纪时，dtcvasc() 使用哪个世纪。

如果该字符串为空字符串，则 dtcvasc() 函数将 *dtvalue* 指向的值设置为空。如果该字符串是可接受的，则函数设置 **datetime** 变量中的值，并返回零。否则，函数保持该变量不变，并返回负的错误代码。

返回代码

0

转换成功。

-1260

不可能在指定的类型之间转换。

-1261

在 **datetime** 或 **interval** 的第一个字段中数字太多。

-1262

在 datetime 或 interval 中的非数值字符。

-1263

在 **datetime** 或 **interval** 值中的字段超出范围或不正确。

-1264

在 **datetime** 或 **interval** 的末尾存在额外的字符。

-1265

在 datetime 或 interval 运算上发生溢出。

-1266

datetime 或 interval 值与该运算不兼容。

-1267

datetime 计算的结果超出范围。

-1268

参数包含无效的 **datetime** 限定符。

示例

demo 目录在 dtcvasc.ec 文件中包含此样例程序。

/*

```
* dtcvasc.ec *
```

The following program converts ASCII datetime strings in ANSI SQL format into datetime (dtime_t) structure.

```
*/

#include <stdio.h>

EXEC SQL include datetime;

main()
{
mint x;

EXEC SQL BEGIN DECLARE SECTION;
datetime year to second dt1;
EXEC SQL END DECLARE SECTION;

printf("DTCVASC Sample ESQL Program running.\n\n");

printf("Datetime string #1 = 2007-02-11 3:10:35\n");
if (x = dtcvasc("2007-02-11 3:10:35", &dt1))
printf("Result =    failed with conversion error: %d\n", x);
else
printf("Result = successful conversion\n");

/*
* Note that the following literal string has a 26 in the      hours place
*/
printf("\nDatetime string #2 = 2007-02-04 26:10:35\n");
if (x = dtcvasc("2007-02-04 26:10:35", &dt1))
printf("Result =    failed with conversion error: %d\n", x);
else
printf("Result = successful conversion\n");
```

```
        printf("\nDTCVASC Sample Program over.\n\n");
        }
```

输出

```
DTCVASC Sample ESQL Program running.

        Datetime string #1 = 2007-02-11 3:10:35
        Result =    successful conversion

        Datetime string #2 = 2007-02-04 26:10:35
        Result = failed with conversion error:-1263
DTCVASC Sample Program over.
```

5.2.41 dtcvfmtasc() 函数

dtcvfmtasc() 函数使用格式化掩码来将字符串转换为 datetime 值。

语法

```
mint dtcvfmtasc(inbuf, fmtstring, dtvalue)
        char *inbuf;
        char *fmtstring;
        dtime_t *dtvalue;
```

inbuf

指向包含要转换的字符串的缓冲区的指针。

fmtstring

指向要针对 *inbuf* 字符串使用其格式化掩码的缓冲区的指针。此时间格式化掩码包含 DBTIME 环境变量支持的相同的格式化伪指令。

dtvalue

指向初始化了的 **datetime** 变量的指针。

用法

您必须以您想要此变量拥有的限定符来初始化 *dtvalue* 中的 **datetime** 变量。**datetime** 变量不需要指定该格式化掩码说明的同一限定符。当 **datetime** 限定符与说明了的格式化掩码限定符不同时，dtcvfmtasc() 扩展 **datetime** 值（如同它以调用了 dtextend() 函数）。

inbuf 中的字符串中的所有限定符字段都必须是相邻的，换句话说，如果限定符为 **hour to second**，则您必须为该字符串中某处的 **hour**、**minute** 和 **second** 都指定值，否则，dtcvfmtasc() 函数返回错误。

inbuf 字符串可有开头的和结尾的空格。然而，从第一个有效位至最后一个，*inbuf* 仅可包含适于格式化掩码说明的限定符字段的数字和定界符。

如果格式化掩码 *fmtstring* 为空字符串，则 dtcvfmtasc() 函数返回错误。如果 *fmtstring* 是空指针，则当 dtcvfmtasc() 函数读取 *inbuf* 中的字符串时，它必须确定格式。当您使用缺省的语言环境时，该函数使用下列优先顺序：

DBTIME 环境变量指定的格式（如果设置 DBTIME 的话）。

GL_DATETIME 环境变量指定的格式（如果设置 GL_DATETIME 的话）。

符合标准 ANSI SQL 格式的缺省的日期格式：

%iY-%m-%d %H:%M:%S

ANSI SQL 格式为输出指定 **year to second** 的限定符。您可以四位（2007）或以两位（07）表示年份。当您在格式化掩码中使用两位年份（**%y**）时，dtcvfmtasc() 函数使用 DBCENTURY 环境变量的值来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 dtcvfmtasc() 为两位年份假定当前的世纪。

当您使用非缺省的语言环境（US English 之外的一种），且未设置 DBTIME 或 GL_DATETIME 环境变量时，dtcvfmtasc() 使用该语言环境定义的缺省的 DATETIME 格式。

当字符串和格式化掩码是可接受的时，dtcvfmtasc() 函数设置 *dtvalue* 中的 **datetime** 变量，并返回零。否则，它返回错误代码，且 **datetime** 变量包含不可预计的值。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录在文件 dtcvfmtasc.ec 中包含此样例程序。该代码将变量 **birthday** 初始化为虚构的生日。

```
/* *dtcvfmtasc.ec*

    The following program illustrates the conversion of several ascii strings into datetime values.

                */


                #include <stdio.h>


                EXEC SQL include datetime;


                main()
                {
                char out_str[17],    out_str2[17],    out_str3[17];
                mint x;

                EXEC SQL BEGIN DECLARE SECTION;
                datetime month to minute birthday;
                datetime year to minute birthday2;
                datetime year to minute birthday3;
                EXEC SQL END DECLARE SECTION;

               printf("DTCVFMTASC Sample ESQL Program running.\n\n");

/* Initialize birthday to "09-06 13:30" */
                printf("Birthday #1 = September 6 at 01:30 pm\n");
                x = dtcvfmtasc("September 6 at 01:30 pm", "%B %d at %I:%M %p",
                &birthday);

 /*Convert the internal format to ascii in ANSI format, for displaying. */
                x = dttoasc(&birthday, out_str);
printf("Datetime (month to minute) value = %s\n\n", out_str);
                /* Initialize birthday2 to "07-14-88 09:15" */
```

```
                printf("Birthday #2 = July 14, 1988. Time: 9:15 am\n");
                x = dtcvfmtasc("July 14, 1988. Time: 9:15am",
                "%B %d, %Y. Time: %I:38p",    &birthday2);

    /*Convert the internal format to ascii in ANSI format, for displaying. */
                x = dttoasc(&birthday2, out_str2);
                printf("Datetime (year to minute) value = %s\n\n", out_str2);
                /* Initialize birthday3 to "07-14-XX 09:15" where XX is current year.
                * Note that birthday3 is year to minute but this initialization only
                * provides month to minute. dtcvfmtasc provides current information for the
missing year.
                */
                printf("Birthday #3 = July 14. Time: 9:15 am\n");
                x = dtcvfmtasc("July 14. Time: 9:15am", "%B %d.
Time: %I:%M %p",&birthday3);

    /* Convert the internal format to ascii in ANSI format, for displaying. */
                x = dttoasc(&birthday3, out_str3);
    printf("Datetime (year to minute) value with current year = %s\n", out_str3);

                printf("\nDTCVFMTASC Sample Program over.\n\n");

                }
```

输出

```
DTCVFMTASC Sample ESQL Program running.

        Birthday #1 = September 6 at 01:30 pm
        Datetime (month to minute) value = 09-06 13:30

        Birthday #2 = July 14, 1988 Time: 9:15 am
        Datetime (year to minute) value =    2007-07-14 09:15

        Birthday #3 = July 14. Time: 9:15 am
```

```
Datetime    (year to minute) value with current year = 2007-07-14 09:15

DTCVFMTASC Sample Program over.
```

5.2.42 dtextend() 函数

dtextend() 函数将 datetime 值扩展为不同的限定符。扩展是指添加或删除 DATETIME 值的字段来使其与给定的限定符相匹配的操作。

语法

mint dtextend(*in_dt*, *out_dt*)

dtime_t *in_dt, *out_dt;

in_dt

指向要扩展的 **datetime** 变量的指针。

out_dt

指向带有要用于该扩展的有效限定符的 **datetime** 变量的指针。

用法

dtextend() 函数将 *in_dt***datetime** 变量的限定符字段数字复制到 *out_dt***datetime** 变量。*out_dt* 变量的限定符控制该复制。

该函数丢弃 *out_dt* 变量不包括的 *in_dt* 中的任何字段。该函数填写 *in_dt* 中未出现的 *out_dt* 中的任何字段，如下：

它从当前的时间和日期来填写 *in_dt* 中最高有效字段左边的字段。

它以多个零来填写 *in_dt* 中最低有效字段右边的字段。

在下列示例中，以始于 6 月 1 日的财政年度的第一天来设置变量 **fiscal_start**。dtextend() 函数生成当前的年份。

```
EXEC SQL BEGIN DECLARE SECTION;
        datetime work, fiscal_start;
        EXEC SQL END DECLARE SECTION;

        work.dt_qual = TU_DTENCODE(TU_MONTH,TU_DAY);
        dtcvasc("06-01",&work);
```

```
fiscal_start.dt_qual = TU_DTENCODE(TU_YEAR,TU_DAY);
dtextend(&work,&fiscal_start);
```

返回代码

0

操作成功。

-1268

参数包含无效的 **datetime** 限定符。

示例

demo 目录在文件 dtextend.ec 中包含此样例程序。

```
/*
        * dtextend.ec *

The following program illustrates the results of datetime extension.
The fields to the right are filled with zeros,and the fields to the left are filled in from current date and time.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        mint x;
        char year_str[20];

        EXEC SQL BEGIN DECLARE SECTION;
        datetime month to day month_dt;
        datetime year to minute year_min;
        EXEC SQL END DECLARE SECTION;

        printf("DTEXTEND Sample ESQL Program running.\n\n");
```

```
        /* Assign value to month_dt and extend */
        printf("Datetime (month to day) value = 12-07\n");
        if(x = dtcvasc("12-07", &month_dt))
        printf("Result = Error %d in dtcvasc()\n", x);
        else
        {
        if (x = dtextend(&month_dt, &year_min))
        printf("Result = Error %d in dtextend()\n", x);
        else
        {
        dttoasc(&year_min, year_str);
printf("Datetime (year to minute) extended value =%s\n", year_str);
        }
        }

        printf("\nDTEXTEND Sample Program over.\n\n");
        }
```

输出

```
DTEXTEND Sample ESQL Program running.

Datetime (month to day) value = 12-07
Datetime (year to minute) extended value = 2006-12-07 00:00
    DTEXTEND Sample Program over.
```

5.2.43 dtsub() 函数

dtsub() 函数从另一 datetime 值减去一个 datetime 值。结果为 interval 值。

语法

```
mint dtsub(d1, d2, inv)
        dtime_t *d1, *d2;
        intrvl_t *inv;
```

d1

指向初始化了的 **datetime** 主变量的指针。

d2

指向初始化了的 **datetime** 主变量的指针。

inv

指向包含结果的 **interval** 主变量的指针。

用法

dtsub() 函数从 *d1* 减去 **datetime** 值 *d2*，并将 **interval** 结果存储在 *inv* 中。该结果可为正值或负值。如果必要，在减法之前，该函数扩展 *d2* 来匹配 *d1* 的限定符。

以 year to month 或 day to fraction(5) 类中的一个值来初始化 *inv* 的限定符。当 *d1* 包含 day to fraction 类中的字段时，interval 限定符还必须在 day to fraction 类中。

返回代码

0

减法成功。

<0

在执行减法时，发生错误。

示例

demo 目录在文件 dtsub.ec 中包含此样例程序。该程序执行 **datetime** 减法，返回在 **year to month** 与 **month to month** 的范围中的等同的 **interval** 结果，并尝试返回在 **day to hour** 的范围中的 **interval** 结果。

```
/*
                * dtsub.ec *

The following program subtracts one DATETIME value from another and  displays  the
resulting INTERVAL value or an error message.
                */

                #include <stdio.h>

                EXEC SQL include datetime;
```

```
main()
{
mint x;
char out_str[16];

EXEC SQL BEGIN DECLARE SECTION;
datetime year to month dt_var1, dt_var2;
interval year to month i_ytm;
interval month to month i_mtm;
interval day to hour i_dth;
EXEC SQL END DECLARE SECTION;


printf("DTSUB Sample ESQL Program running.\n\n");

printf("Datetime (year to month) value #1 = 2007-10\n");
dtcvasc("2007-10", &dt_var1);
printf("Datetime (year to month) value #2 = 2001-08\n");
dtcvasc("2001-08", &dt_var2);

printf("-----------------------------------\n");

/* Determine year-to-month difference */
printf("Difference (year to month)       = ");
if(x = dtsub(&dt_var1, &dt_var2, &i_ytm))
printf("Error from dtsub(): %d\n", x);
else
{
/* Convert to ASCII for displaying */
intoasc(&i_ytm, out_str);
printf("%s\n", out_str);
}
```

```
/* Determine month-to-month difference */
printf("Difference (month to month)              = ");
if(x = dtsub(&dt_var1, &dt_var2, &i_mtm))
printf("Error from dtsub(): %d\n", x);
else
{
/* Convert to ASCII for displaying */
intoasc(&i_mtm, out_str);
printf("%s\n", out_str);
}

 /* Determine day-to-hour difference: Error - Can't convert year-to-month to
day-to-hour
*/
printf("Difference (day to hour)    = ");
if(x = dtsub(&dt_var1, &dt_var2, &i_dth))
printf("Error from dtsub(): %d\n", x);
else
{
/* Convert to ASCII for displaying */
intoasc(&i_dth, out_str);
printf("%s\n", out_str);
}

printf("\nDTSUB Sample Program over.\n\n");
}
```

输出

```
DTSUB Sample ESQL Program running.

        Datetime (year to month) value #1 = 2007-10
        Datetime (year to month) value #2 = 2001-08
```

```
----------------------------------------
Difference (year to month)    = 0006-02
Difference (month to month) = 86
Difference (day to hour)          = Error from dtsub(): -1266

DTSUB Sample Program over.
```

5.2.44 dtsubinv() 函数

dtsubinv() 函数从 **datetime** 之中减去 interval 值。结果为 datetime 值。

语法

```
mint dtsubinv(dt, inv, res)
        dtime_t *dt;
        intrvl_t *inv;
        dtime_t *res;
```

dt

指向初始化了的 **datetime** 主变量的指针。

inv

指向初始化了的 **interval** 主变量的指针。

res

指向包含结果的 **datetime** 主变量的指针。

用法

dtsubinv() 函数从 *dt* 中的 **datetime** 中减去 *inv* 中的 **interval** 值，并存储 *res* 中的 **datetime** 值。此结果继承 *dt* 的限定符。

datetime 值必须包括出现在 **interval** 值中的所有字段。当您为初始化 *inv* 中的变量 *dt* 时，该函数可能返回不可预料的结果。

返回代码

0

减法成功。

<0

在执行减法时，发生错误。

示例

demo 目录在文件 dtsubinv.ec 中包含此样例程序。

```
/*
        * dtsubinv.ec *

    The following program subtracts an INTERVAL value from a DATETIME value and
displays the result.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        char out_str[16];

        EXEC SQL BEGIN DECLARE SECTION;
        datetime year to minute dt_var, result;
        interval day to minute intvl;
        EXEC SQL END DECLARE SECTION;

        printf("DTSUBINV Sample ESQL Program running.\n\n");

        printf("Datetime (year to month) value = 2007-11-28\n");
        dtcvasc("2007-11-28 11:40", &dt_var);
        printf("Interval (day to minute) value      =          50 10:20\n");
        incvasc("50 10:20", &intvl);

        printf("-------------------------------------\n");
        dtsubinv(&dt_var, &intvl, &result);
```

```
    /* Convert to ASCII for displaying */
    dttoasc(&result, out_str);
    printf("Difference (year to hour)      = %s\n", out_str);

    printf("\nDTSUBINV Sample Program over.\n\n");
    }
```

输出

```
DTSUBINV Sample ESQL Program running.

    Datetime (year to month) value = 2007-11-28
    Interval (day to minute) value =      50 10:20
    ----------------------------------------------------
    Difference (year to hour)          = 2007-10-09    01:20

    DTSUBINV Sample Program over.
```

5.2.45 dttoasc() 函数

dttoasc() 函数将 datetime 变量的字段值转换为符合 ANSI SQL 标准的 ASCII 字符串。

语法

```
mint dttoasc(dtvalue, outbuf)
    dtime_t *dtvalue;
    char *outbuf;
```

dtvalue

指向要转换的初始化了的 **datetime** 变量的指针。

outbuf

指向为 *dtvalue* 中的值接收 ANSI 标准 DATETIME 字符串的缓冲区的指针。

用法

dttoasc() 函数将 **datetime** 变量中的字段的数字转换为它们等同的字符，并将它们复制至 *outbuf* 字符串，在它们之间带有定界符（连字符、空格、冒号或句号）。您必须以您想要该字符串拥有的限定符来初始化 *dtvalue* 中的 **datetime** 变量。

该字符串不包括 SQL 语句用于定界 DATETIME 文字的限定符或圆括号。 *outbuf* 字符串符合 ANSI SQL 标准。它包括针对每一定界符的一个字符，加上字段，其大小如下：

字段

字段大小

Year

4 位

DATETIME 的部分

如精度所指定

所有其他字段

2 位

带有 **year to fraction(5)** 限定符的 **datetime** 值产生最大的输出长度。等同的字符串包含 19 位、6 定界符以及空终止符，总计 26 字节：

YYYY-MM-DD HH:MM:SS.FFFFF

如果您未初始化 **datetime** 变量的限定符，则 dttoasc() 函数返回不可预料的值，但此值不超过 26 字节。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录在文件 dttoasc.ec 中包含此样例程序。

```
/*
        * dttoasc.ec *
  The following program illustrates the conversion of a datetime value into an ASCII string
in ANSI SQL format
        */
```

```
#include <stdio.h>

EXEC SQL include datetime;

main()
{
char out_str[16];

EXEC SQL BEGIN DECLARE SECTION;
datetime year to hour dt1;
EXEC SQL END DECLARE SECTION;

printf("DTTOASC Sample ESQL Program running.\n\n");

/* Initialize dt1 */
dtcurrent(&dt1);

/* Convert the internal format to ascii for displaying */
dttoasc(&dt1, out_str);

/* Print it out*/
printf("\tToday's datetime (year to hour)value    is %s\n", out_str);

printf("\nDTTOASC Sample Program over.\n\n");
}
```

输出

```
DTTOASC Sample ESQL Program running.

        Today's datetime (year to hour) value is 2007-09-19 08

 DTTOASC Sample Program over.
```

5.2.46 dttofmtasc() 函数

dttofmtasc() 使用格式化的掩码来将 datetime 变量转换为字符串。

语法

```
mint dttofmtasc(dtvalue, outbuf, buflen, fmtstring)
        dtime_t *dtvalue;
        char *outbuf;
        mint buflen;
        char *fmtstring;
```

dtvalue

指向要转换的初始化了的 **datetime** 变量的指针。

outbuf

对于 *dtvalue* 中的值，指向接收该字符串的缓冲区的指针。

buflen

outbuf 缓冲区的长度。

fmtstring

包含要为 *outbuf* 字符串使用的格式化掩码的缓冲区的指针。此时间格式化掩码包含 **DBTIME** 环境变量支持的同样的格式化伪指令。

用法

您必须以你想要该字符串拥有的限定符来初始化 *dtvalue* 中的 **datetime** 变量。如果您未初始化 **datetime** 变量，则该函数返回不可预料的值。*outbuf* 中的字符串*不*包括 SQL 语句用来定界 DATETIME 文字的限定符或圆括号。

格式化掩码 *fmtstring* 不需要说明与 **datetime** 变量相同的限定符。当说明了的格式化掩码限定符与 **datetime** 限定符不同时，dttofmtasc() 扩展 **datetime** 值（如同它调用了 dtextend() 函数那样）。

如果格式化掩码为空字符串，则该函数将字符串 *outbuf* 设置为空字符串。如果 *fmtstring* 是空指针，则 dttofmtasc() 必须确定为 *outbuf* 中的字符串使用的格式。当您使用缺省的语言环境时，该函数采用下列优先顺序：

DBTIME 环境变量指定的格式（如果设置 DBTIME 的话）。

GL_DATETIME 环境变量指定的格式（如果设置 GL_DATETIME 的话）。

符合标准 ANSI SQL 格式的的缺省的日期格式：

%iY-%m-%d %H:%M:%S

当您在格式化掩码中使用两位年份（**%y**）时，dttofmtasc() 函数使用 DBCENTURY 环境变量的值来确定使用哪一世纪。如果您未设置 DBCENTURY，则 dttofmtasc() 为两位年份假定当前的世纪。

当您使用非缺省的语言环境（US English 之外的一种），且未设置 DBTIME 或 GL_DATETIME 环境变量的话，则 dttofmtasc() 使用客户机语言环境定义的缺省的 DATETIME 格式。

返回代码

0

转换成功。

<0

转换失败。请检查错误消息的文本。

示例

demo 目录在文件 dttofmtasc.ec 中包含此样例程序。

```
/* *dttofmtasc.ec*
The following program illustrates the conversion of a datetime
value into strings of different formats.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        char out_str1[25];
        char out_str2[25];
        char out_str3[30];
```

```
        mint x;

        EXEC SQL BEGIN DECLARE SECTION;
        datetime month to minute birthday;
        EXEC SQL END DECLARE SECTION;

      printf("DTTOFMTASC Sample ESQL Program running.\n\n");

        /* Initialize birthday to "09-06 13:30" */
        printf("Birthday datetime (month to minute) value = ");
        printf("September 6 at 01:30 pm\n");
   x = dtcvfmtasc("September 6 at 01:30 pm","%B %d at %I:%M %p",
        &birthday);

        /* Convert the internal format to ascii for 3 given display formats.Note that the
second format does not include the minutes field and that the last format includes a year field
even though birthday was
        * not initialized as year to minute.
        */

        x = dttofmtasc(&birthday, out_str1, sizeof(out_str1),
        "%d %B at %H:%M");
        x = dttofmtasc(&birthday, out_str2, sizeof(out_str2),
        "%d %B at %H");
        x = dttofmtasc(&birthday, out_str3, sizeof(out_str3),
        "%d %B, %Y at%H:%M");      /* Print out the three forms of the same date */
   printf("\tFormatted value (%%d %%B at %%H:%%M) = %s\n",      out_str1);
   printf("\tFormatted value (%%d %%B at %%H) = %s\n", out_str2);
   printf("\tFormatted value (%%d %%B, %%Y at %%H:%%M) = %s\n",    out_str3);

   printf("\nDTTOFMTASC Sample Program over.\n\n");
        }
```

输出

DTTOFMTASC Sample ESQL Program running.

Birthday datetime (month to minute) value =

September 6 at 01:30 pm

Formatted value (%d %B at %H:%M) = 06 September at 13:30

Formatted value (%d %B at %H)) = 06 September at 13

Formatted value (%d %B, %Y at %H:%M)) =

06 September, 2007 at 13:30

DTTOFMTASC Sample Program over.

5.2.47 GetConnect() 函数（Windows(TM)）

GetConnect() 函数仅在 Windows(TM) 环境中可用，并建立与数据库服务器的新的显式的连接。

重要： 为了与 Windows(TM) 应用程序的版本 5.01 GBase 8s ESQL/C 相兼容，GBase 8s ESQL/C 支持 GetConnect() 连接库函数。对于 Windows(TM) 环境，当您编写新的 GBase 8s ESQL/C 应用程序时，请使用 SQL CONNECT 语句来建立显式的连接。

语法

void *GetConnect ( )

用法

GetConnect() 函数*自我*调用，等同于下列 SQL 语句：

EXEC SQL connect to '*@dbservername*' with concurrent transaction;

在此示例中，*dbservername* 是定义了的数据库服务器的名称。必须在*至少*下列位置之一中定义客户机应用程序指定的所有数据库服务器：

Registry 中的 GBASEDBTSERVER 环境变量包含*缺省的*数据库服务器的名称。Setnet32 实用程序设置该 Registry 值。

InetLogin 结构中的 **InfxServer** 字段包含*缺省的*数据库服务器或*指定的*数据库服务器的名称。客户机应用程序设置这些 **InetLogin** 字段。

例如，下列代码片段使用 GetConnect() 来建立与 **mainsrvr** 数据库服务器上 **stores7** 数据库的显式的连接：

void *cnctHndl;

⋮

```
strcpy(InetLogin.InfxServer, "mainsrvr");
⋮

cnctHndl = GetConnect();
EXEC SQL database stores7;
```

在前一示例中，如果您已省略了对 **InetLogin.InfxServer** 字段赋值，则 GBase 8s ESQL/C 会建立与缺省的数据库服务器（Registry 中 GBASEDBTSERVER 环境变量指示的数据库服务器）中 **stores7** 数据库的显示的连接。

在任何对 GetConnect() 的调用之后，请使用 SQL DATABASE 语句（或打开数据库的某其他 SQL 语句）来打开想要的数据库。在前面的代码片段中，GetConnect() 函数与 DATABASE 语句的组合等同于下列 CONNECT 语句：

```
EXEC SQL connect to 'stores7@mainsrvr' with concurrent transaction;
```

重要：由于 GetConnect() 函数映射到 CONNECT 语句，因此，它设置 SQLCODE 和 SQLSTATE 状态代码来指示该连接请求是成功还是失败。在 Windows(TM) 的版本 5.01 GBase 8s ESQL/C 中，GetConnect() 与此行为不同，其中，此函数不设置 SQLCODE 和 SQLSTATE 值。

下表展示使用 GetConnect() 函数与使用 SQL CONNECT 语句之间的差异。

情况	GetConnect() 库函数	SQL CONNECT 语句
连接名称	在内部生成，并保存在该连接的连接句柄结构中	在内部生成，除非 CONNECT 包括 AS 子句；因此，要切换至其他连接，当您创建该连接时，请指定 AS 子句。
打开数据库	仅建立到数据库服务器的显式的连接；因此，应用程序必须使用 DATABASE 语句（或某其他有效的 SQL 语句）来打开数据库。	可创建到数据库服务器的显式的连接,*并*当同时提供数据库服务器与数据库的名称时，打开数据库

重要： 由于 GetConnect() 函数映射到带有 WITH CONCURRENT TRANSACTION 子句的 CONNECT 语句，因此，它允许带有打开的事务的显式的连接成为休眠的。在您的 GBase 8s ESQL/C 应用程序调用 SetConnect() 函数来切换至另一显式的连接之前，它不需要确保提交了或回滚了当前的事务。

对于您以 GetConnect() 建立的每一连接，请调用 ReleaseConnect() 来关闭该连接，并释放资源。

返回代码

CnctHndl

调用 GetConnect() 成功，且函数已为新的连接返回了连接句柄。

空指针

调用 GetConnect() 不成功。

5.2.48 ifx_cl_card() 函数

ifx_cl_card() 函数返回指定的集合类型主变量的基数。

语法

mint ifx_cl_card(*collp*, isnull)

```
ifx_collection_t *collp;
mint *isnull;
```

collp

指向应用程序中 **collection** 主变量的名称的指针。

isnull

如果该集合为空，则设置为 1，否则，设置为 0

用法

ifx_cl_card() 函数使得您能够确定集合中元素的数目、是否为空集合，以及集合是否为空值。

返回代码

0

空集合。

>0

集合中元素的数目。

<0

发生错误。

示例

此样例程序在 demo 目录中的 ifx_cl_card.ec 文件中。

```
/*
 * Check the cardinality of the collection variable when
 * the data is returned from the server
 */

        main()
        {
        exec sql begin declare section;
        client collection myset;
        exec sql end declare section;
        mint numelems = 0;
        mint isnull = 0;

        exec sql allocate collection ::myset;
        exec sql create database newdb;
        exec sql create table tab (col set(int not null));
        exec sql insert into tab values ("set{}");
        exec sql select * into :myset from tab;
        if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
        printf("collection is empty\n");
 else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
        printf("collection is null\n");
        else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
        printf("number of elements is %d\n", numelems);
        else
        printf("error occurred\n");

        exec sql update tab set col = ’set{1,2,3}’;
        exec sql select * into :myset from tab;
        if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
        printf("collection is empty\n");
```

```
else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
        printf("collection is null\n");
        else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
        printf("number of elements is %d\n", numelems);
        else
        printf("error occurred\n");

        exec sql update tab set col = NULL;
        exec sql select * into :myset from tab;
        if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 0)
        printf("collection is empty\n");
    else if ((ifx_cl_card(myset, &isnull) == 0) && isnull == 1)
        printf("collection is null\n");
        else if ((numelems = ifx_cl_card(myset, &isnull))> 0)
        printf("number of elements is %d\n", numelems);
        else
        printf("error occurred\n");
        }
```

输出

```
collection is empty
        number of elements is 3
        collection is null
```

5.2.49 ifx_dececvt() 和 ifx_decfcvt() 函数

ifx_dececvt() 和 ifx_decfcvt() 函数是 dececvt() 和 decfcvt()GBase 8s ESQL/C 库函数的线程安全版本。

语法

```
mint ifx_dececvt(np, ndigit, decpt, sign, decstr, decstrlen)
        register dec_t *np;
        register mint ndigit;
        mint *decpt;
        mint *sign;
        char *decstr;
        mint decstrlen;
```

```
mint ifx_decfcvt(np, ndigit, decpt, sign, decstr, decstrlen)
register dec_t *np;
register mint ndigit;
mint *decpt;
mint *sign;
char *decstr;
mint decstrlen;
```

np

指向包含要被转换的 **decimal** 值的 **decimal** 结构的指针。

ndigit

ifx_dececvt() 的 ASCII 字符串的长度。它是 ifx_decfcvt() 的小数点的右边的位数。

decpt

指向整数的指针，该整数是小数点相对于该字符串的开头的位置。*decpt 的负值或零值意味着该位置位于返回的数字的左边。

sign

指向结果的符号的指针。如果该结果的符号为负的，则 *sign 非零；否则，它为零。

decstr

函数将转换了的 decimal 值返回到其中的用户定义的缓冲区。

decstrlen

用户定义的 decstr 缓冲区的长度，以字节计。

用法

ifx_dececvt() 函数是 dececvt() 函数的线程安全版本。ifx_decfcvt() 函数是 decfcvt() 函数的线程安全版本。当两个线程同时调用该函数时，每一函数返回一个不可被重写的字符串。

返回代码

0

转换成功。

<0

转换不成功。

-1273

输出缓冲区为空，或太小以至于不能保存结果。

5.2.50 ifx_defmtdate() 函数

ifx_defmtdate() 函数使用格式化掩码来将字符串转换为内部的 DATE 格式。

语法

```
mint ifx_defmtdate(jdate, fmtstring, instring, dbcentury)
        int4 *jdate;
        char *fmtstring;
        char *instring;
        char dbcentury;
```

jdate

指向 **int4** 整数值的指针，该值接收 *inbuf* 字符串的内部 DATE 值。

fmtstring

指向包含要用于 *inbuf* 字符串的格式化掩码的缓冲区的指针。

instring

指向包含要转换的日期字符串的缓冲区的指针。

dbcentury

可为下列字符之一，其确定适用于该日期的年份部分的世纪：

R

当前的。该函数使用当前年份的两个高数位来扩展该年份值。

P

过去的。该函数使用当前的和过去的世纪来扩展该年份值。它将这两个日期与当前日期对比，并使用在当前世纪之前的那个世纪。如果两个日期都在当前日期之前，则该函数使用距离当前日期最近的世纪。

F

未来的。该函数使用当前的和下一世纪来扩展该年份值。它将这些世纪与当前日期对比，并使用晚于当前日期的那个世纪。如果两个日期都晚于当前的日期，则该函数使用距离当前日期最近的日期。

C

最近的。该函数使用当前的、过去的和下一世纪来扩展该年份值。它选择距离当前日期最近的那个世纪。

用法

fmtstring 参数指向日期格式化掩码，其包含描述如何说明该日期字符串的格式。

input 字符串和 *fmtstring* 必须依照与月份、日子和年份相同的先后顺序。然而，它们不需要包含月份、日子和年份的相同的文字或相同的表示。

您可在 *fmtstring* 中包括 weekday 格式（ww），但数据库服务器忽略那种格式。没有来自 *inbuf* 的内容与该 weekday 格式相对应。

下列 *fmtstring* 与 *input* 的组合是有效的。

格式化掩码输入

mmddyy

Dec. 25th, 2007

mmddyyyy

Dec. 25th, 2007

mmm. dd. yyyy

dec 25 2007

mmm. dd. yyyy

DEC-25-2007

mmm. dd. yyyy

122507

mmm. dd. yyyy

12/25/07

yy/mm/dd

07/12/25

yy/mm/dd

2007, December 25

yy/mm/dd

In the year 2007, the month of December, it is the 25th day

dd-mm-yy

This 25th day of December 2007

如果存储在 *inbuf* 中的值是四位的年份，则 ifx_defmtdate() 函数使用那个值。如果存储在 *inbuf* 中的值是二位的年份，则 ifx_defmtdate() 函数使用 *dbcentury* 参数的值来确定要使用的那个世纪。如果您未设置 *dbcentury* 参数，则 ifx_defmtdate() 使用 DBCENTURY 环境变量来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 ifx_strdate() 假定两位年份的当前世纪。

返回代码

如果您使用无效的日期字符串格式，则 ifx_defmtdate() 返回错误代码，并将内部的 DATE 设置为当前日期。下列是可能的返回代码：

0

操作成功。

-1204

**input* 参数指定无效的年份。

-1205

**input* 参数指定无效的月份。

-1206

**input* 参数指定无效的日子。

-1209

由于 **input* 未包含在该年份、月份和日子之间的定界符，因此，**input* 的长度必须恰为 6 或 8 字节。

-1212

**fmtstring* 未指定年份、月份和日子。

5.2.51 ifx_dtcvasc() 函数

ifx_dtcvasc() 函数将 DATETIME 值的符合 ANSI SQL 标准的字符串转换为 datetime 值。

语法

```
mint dtcvasc(str, d, dbcentury)
        char *str;
        dtime_t *d;
        char dbcentury;
```

str

指向包含 ANSI 标准 DATETIME 字符串的缓冲区的指针。

d

指向初始化了的 **datetime** 变量的指针。

dbcentury

可为下列字符之一，其确定适用于该日期的年份部分的那个世纪：

R

当前的。该函数使用当前年份的两个高数位来扩展该年份值。

P

过去的。该函数使用过去的和当前的世纪来扩展该年份值。它将这两个日期与当前的日期对比，并使用在当前世纪之前的世纪。如果两个日期都在当前日期之前，则该函数使用距离当前日期最近的那个世纪。

F

将来的。该函数使用当前的和下一世纪来扩展该年份值。它将这些值与当前日期对比，并使用晚于当前日期的那个世纪。如果两个日期都晚于当前日期，则该函数使用距离当前日期最近的日期。

C

最近的。该函数使用过去的、当前的和下一世纪来扩展该年份值。它选择距离当前日期最近的那个世纪。

用法

您必须以您想要此变量拥有的限定符来初始化 *d* 中的 **datetime** 变量。

str 中的字符串必须有符合采用 ANSI SQL 格式的 **year to second** 限定符的值。*str* 字符串可有开头的和结尾的空格。然而，对于 DATETIME 值，从第一个有效位至最后一个，*str* 仅可包含符合 ANSI SQL 标准的数字和定界符字符。

如果您指定年份值为一位或两位，则 ifx_dtcvasc() 函数使用 *dbcentury* 参数的值来确定要使用哪个世纪。如果您未设置 *dbcentury* 参数，则 ifx_dtcvasc() 使用 DBCENTURY 环境变量来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 ifx_dtcvasc() 假定两位年份的当前世纪。

如果该字符串为空字符串，则 ifx_dtcvasc() 函数将 *d* 指向的值设置为空。如果字符串是可接受的，则该函数设置 **datetime** 变量中的值，并返回零。否则，该函数保持该变量不变，并返回负的错误代码。

返回代码

0

转换成功。

-1260

在指定的类型之间，不可能转换。

-1261

在 **datetime** 或 **interval** 的第一个字段中数字太多。

-1262

datetime 或 interval 中的非数值字符。

-1263

datetime 或 **interval** 值中的字符按超出范围或不正确。

-1264

在 **datetime** 或 **interval** 的结尾处存在额外的字符。

-1265

在 datetime 或 interval 操作上发生了溢出。

-1266

datetime 或 interval 值与该操作不相兼容。

-1267

datetime 计算的结果超出范围。

-1268

参数包含无效的 **datetime** 限定符。

5.2.52 ifx_dtcvfmtasc() 函数

ifx_dtcvfmtasc() 函数使用格式化掩码来将字符串转换为 datetime 值。

语法

```
mint ifx_dtcvfmtasc(input, fmtstring, d, dbcentury)
        char *input;
        char *fmtstring;
        dtime_t *d;

        char dbcentury;
```

input

指向包含要转换的字符串的缓冲区的指针。

fmtstring

指向包含要用于输入字符串的更实话掩码的缓冲区的指针。此时间格式化掩码包含 DBTIME 环境变量支持的相同的格式化伪指令。

d

指向初始化了的 **datetime** 变量的指针。

dbcentury

可为下列字符之一，其确定适用于该日期的年份部分的那个世纪：

R

当前的。该函数使用当前年份的两个高数位来扩展该年份值。

P

过去的。该函数使用过去的和当前的世纪来扩展该年份值。它将这两个日期与当前日期对比，并使用在当前世纪之前的世纪。如果两个日期都在当前日期之前，则该函数使用距离当前日期最近的世纪。

F

未来的。该函数使用当前的和下一世纪来扩展该年份值。它将这些世纪与当前日期对比，并使用晚于当前日期的世纪。如果两个日期都晚于当前日期，则该函数使用距离当前日期最近的世纪。

用法

您必须以您想要此变量拥有的限定符来初始化 *d* 中的 **datetime** 变量。**datetime** 变量不需要指定该格式化掩码说明的同一限定符。当 **datetime** 限定符与说明了的格式化掩码限定符不同时，ifx_dtcvfmtasc() 扩展 **datetime** 值（如同它已调用了 dtextend() 函数那样）。

input 中字符串中的所有限定符字段都必须是相邻的。换句话说，如果该限定符为 **hour to second**，则您必须在该字符串中的某处为 **hour**、**minute** 和 **second** 指定所有值，否则，ifx_dtcvfmtasc() 函数返回错误。

input 字符串可有开头的和结尾的空格。然而，对于该格式化掩码说明的限定符字段，从第一个有效位至最后一个，*input* 仅可包含恰当的数字和定界符。

如果格式化掩码 *fmtstring* 为空字符串，则 ifx_dtcvfmtasc() 函数返回错误。如果 *fmtstring* 为空指针，则当 ifx_dtcvfmtasc() 函数读取 *input* 中的字符串时，它必须确定要使用的格式。当您使用缺省的语言环境时，该函数采用下列优先顺序：

DBTIME 环境变量指定的格式（如果设置 DBTIME 的话）。

GL_DATETIME 环境变量指定的格式（如果设置 GL_DATETIME 的话）。

符合标准 ANSI SQL 格式的缺省日期格式：

%iY-%m-%d %H:%M:%S

ANSI SQL 格式为输出指定 **year to second** 的限定符。您可将年份表达为四位（2007）或为两位（07）。当您在格式化掩码中使用两位年份（**%y**）时，ifx_dtcvfmtasc() 函数使用 *dbcentury* 参数的值来确定要使用哪个世纪。如果您未设置 *dbcentury* 参数，则 ifx_dtcvfmtasc() 使用 DBCENTURY 环境变量来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 ifx_dtcvfmtasc() 为两位年份假定当前的世纪。

当您使用非缺省的语言环境（US English 之外的一种），且未设置 DBTIME 或 GL_DATETIME 环境变量时，ifx_dtcvfmtasc() 使用该语言环境定义的缺省 DATETIME 格式。

当该字符串和格式化掩码是可接受的时，ifx_dtcvfmtasc() 函数设置 *d* 中的 **datetime** 变量，并返回零。否则，它返回错误代码，且 **datetime** 变量包含不可预料的值。

返回代码

0

转换成功。

<0

转换失败。

5.2.53 ifx_dttofmtasc() 函数

ifx_dttofmtasc() 函数使用格式化掩码来将 datetime 变量转换为字符串。

语法

```
mint dttofmtasc(dtvalue, output, str_len, fmtstring, dbcentury)
        dtime_t *dtvalue;
        char *outbuf;
        mint buflen;
```

char *fmtstring;

d

指向要转换的初始化了的 **datetime** 变量的指针。

output

指向为 d 中的值接收该字符串的缓冲区的指针。

str_len

输出缓冲区的长度。

fmtstring

指向包含要用于输出字符串的格式化掩码的缓冲区的指针。此时间格式化掩码包含 DBTIME 环境变量支持的同一格式化伪指令。

dbcentury

可为下列字符之一，其确定适用于该日期的年份部分的那个世纪：

R

当前的。该函数使用当期年份的两个高数位来扩展该年份值。

P

过去的。该函数使用过去的和但前的世纪来扩展该年份值。它将这两个日期与当前日期相对比，并使用在当期世纪之前的世纪。如果两个日期都在当前日期之前，则该函数使用距离当期日期最近的世纪。

F

将来的。该函数使用当前的和下一世纪来扩展该年份值。它将这些世纪与当前日期相对比，并使用晚于当前日期的世纪。如果两个日期都晚于当前日期，则该函数使用距离当前日期最近的日期。

C

最近的。该函数使用过去的、当前的和下一世纪来扩展该年份值。它选择距离当前日期最近的世纪。

用法

您必须以您想要该字符串拥有的限定符来初始化 *dtvalue* 中的 **datetime** 变量。如果您未初始化 **datetime** 变量，则该函数返回不可预料的值。*outbuf* 中的字符串不包括 SQL 语句用来定界 DATETIME 文字的限定符或圆括号。

格式化掩码 *fmtstring* 不需要说明相同的限定符作为 **datetime** 变量。当说明了的格式化掩码不同于 **datetime** 限定符时，dttofmtasc() 扩展 **datetime** 值（如同它已调用了 dttofmtasc() 函数那样）。

如果该格式化掩码为空字符串，则该函数将字符串 *outbuf* 设置为空字符串。如果 *fmtstring* 为空指针，则 dttofmtasc() 函数必须为 *outbuf* 中的字符串确定要使用的格式。当您使用缺省的语言环境时，该函数采用下列优先顺序：

DBTIME 环境变量指定的格式（如果设置 DBTIME 的话）。

GL_DATETIME 环境变量指定的格式（如果设置 GL_DATETIME 的话）。

符合标准 ANSI SQL 格式的缺省的日期格式：

%iY-%m-%d %H:%M:%S

当您在格式化掩码中使用两位年份（**%y**）时，dttofmtasc() 函数使用 DBCENTURY 环境变量的值来确定要使用的世纪。如果您未设置 DBCENTURY，则 dttofmtasc() 为两位年份假定当前的世纪。

当您使用非缺省的语言环境（US English 之外的一种），且未设置 DBTIME 或 GL_DATETIME 环境变量时，dttofmtasc() 使用客户机语言环境定义的缺省的 DATETIME 格式。

返回代码

0

转换成功。

<0

转换失败。请检查错误消息的文本。

5.2.54 ifx_getenv() 函数

ifx_getenv() 函数检索当前环境变量的值。

语法

char *ifx_getenv(*varname*);

const char *varname;

varname

指向包含环境变量的名称的缓冲区的指针。

用法

ifx_getenv() 函数以下列顺序来搜索环境变量：

应用程序以 ifx_putenv() 函数或直接地修改了或定义了的 GBase 8s 环境变量的表（**InetLogin** 结构）

用户以 Setnet32 实用程序已在 Registry 中定义了的 GBase 8s 环境变量的表

从 C 运行时刻环境变量检索的非 GBase 8s 环境变量

定义了的 GBase 8s 环境变量的缺省值的表

ifx_getenv() 函数不区分大小写。您以大写或小写指定环境变量的名称。

ifx_getenv() 仅对 C 运行时刻库可访问的数据结构进行操作，而不对操作系统为进程创建的环境段进行操作。因此，使用 ifx_getenv() 的程序可能检索无效的信息。

ifx_putenv() 和 ifx_getenv() 函数使用全局变量 **_environ** 指向的环境的副本来访问该环境。

下列程序片段使用 ifx_getenv() 来检索 GBASEDBTDIR 环境变量的当前值：

```
char GbasedbtDirVal[100];


        /* Get current value of GBASEDBTDIR */
        GbasedbtDirVal = ifx_getenv( "gbasedbtdir" );
        /* Check if GBASEDBTDIR is set */
        If( GbasedbtDirVal != NULL )
printf( "Current GBASEDBTDIR value is %\n", GbasedbtDirVal );
```

返回代码

ifx_getenv() 函数返回指向包含 *varname* 的 GBase 8s 环境表条目的指针，或返回 NULL，如果该函数在表中未找到 *varname* 的话。

限制： 请不要使用返回的指针来修改环境变量的值。请改为使用 ifx_putenv() 函数。如果 ifx_getenv() 在 GBase 8s 环境表中未找到 "varname" ，则返回值为 NULL。

5.2.55 ifx_getcur_conn_name() 函数

ifx_getcur_conn_name() 函数返回当前连接的名称。

语法

```
char *ifx_getcur_conn_name(void);
```

用法

当前的连接是活动的数据库服务器连接，当前，其正在将 SQL 请求发送至数据库服务器，并可能正在从数据库服务器接收数据。在 callback 函数中，当前的连接是当随同对 sqlbreakcallback() 函数的调用注册了 callback 的那个时刻的当前连接。当前连接的名称是当前连接的显式的名称。如果建立连接的 CONNECT 语句未包括 AS 子句，则该连接没

有显式的名称。

返回代码

当前连接的名称

成功地取得当前连接名称

空指针

不能取得当前连接名称，或当前连接没有显式的名称

5.2.56 ifx_getserial8() 函数

ifx_getserial8() 函数返回插入至 int8 主变量内的最后一行的 SERIAL8 值。

语法

void ifx_getserial8(*serial8_val*)

ifx_int8_t *serial8_val;

serial8_val

指向 ifx_getserial8() 在其中放置新插入的 SERIAL8 值的 **int8** 结构的指针。

用法

在您插入包含 SERIAL8 列的行之后，请使用 ifx_getserial8() 函数。该函数返回您声明的 **int8** 变量、*serial8_val* 中新的 SERIAL8 值。如果 INSERT 语句生成了新的 SERIAL8 值，则 *serial8_val* 指向大于零的值。零或空的 SERIAL8 值指示无效的 INSERT；该 INSERT 可能以失败，或可能尚未执行。

示例

```
EXEC SQL BEGIN DECLARE SECTION;
        int8 order_num;
        int8 rep_num;
        char str[20];
        EXEC SQL END DECLARE SECTION;

        EXEC SQL create table order2
        (
        order_number SERIAL8(1001),
```

```
order_date DATE,
customer_num INTEGER,
backlog CHAR(1),
po_num CHAR(10),
paid_date DATE,
sales_rep INT8
);
EXEC SQL insert into order2 (order_number, sales_rep)
values (0, :rep_num);
if (SQLCODE == 0)
{
ifx_getserial8(order_num);
if (ifx_int8toasc(&order_num, str, 20) == 0)
printf("New order number is %s\n", str);
}
```

5.2.57 ifx_int8add() 函数

ifx_int8add() 函数将两个 int8 类型值相加。

语法

```
mint ifx_int8add(n1, n2, sum)
        ifx_int8_t *n1;
        ifx_int8_t *n2;
        ifx_int8_t *sum;
```

n1

指向包含第一个操作对象的 **int8** 结构的指针。

n2

指向包含第二个操作对象的 **int8** 结构的指针。

sum

指向包含 *n1 + n2* 的和的 **int8** 结构的指针。

用法

sum 可与 *n1* 或 *n2* 相同。

返回代码

0

运算成功。

-1284

运算导致溢出或下溢。

示例

demo 目录中的文件 int8add.ec 包含下列示例程序。

*int8add.ec *

The following program obtains the sum of two INT8 type values.

```
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "6";
char string2[] = "9,223,372,036,854,775";
char string3[] = "999,999,999,999,999,9995";
char result[41];

main()
{
mint x;
ifx_int8_t num1, num2, num3, sum;

printf("INT8 Sample ESQL Program running.\n\n");

  if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
{
printf("Error %d in converting string1 to INT8\n", x);
```

```
                exit(1);
                }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
                {
                printf("Error %d in converting string2 to INT8\n", x);
                exit(1);
                }
     if (x = ifx_int8add(&num1, &num2, &sum))        /* adding the first two INT8t */
                {
                printf("Error %d in adding INT8t\n", x);
                exit(1);
                }
                if (x = ifx_int8toasc(&sum, result, sizeof(result)))
                {
    printf("Error %d in converting INT8 result to string\n", x);
                exit(1);
                }
                result[40] = '\0';
                printf("\t%s + %s = %s\n", string1, string2, result); /* display result */

                /* attempt to convert to INT8 value that is too large*/

    if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
                {
                printf("Error %d in converting string3 to INT8\n", x);
                exit(1);
                }
                if (x = ifx_int8add(&num2, &num3, &sum))
                {
                printf("Error %d in adding INT8t\n", x);
                exit (1);
                }
                if (x = ifx_int8toasc(&sum, result, sizeof(result)))
```

```
{
printf("Error %d in converting INT8 result to string\n", x);
exit(1);
}
result[40] = '\0';
printf("\t%s + %s = %s\n", string2, string3, result); /* display result */

printf("\nINT8 Sample Program over.\n\n");
exit(0);
}
```

输出

```
INT8 Sample ESQL Program running.

    6 + 9,223,372,036,854,775 = 9223372036854781
    Error -1284 in converting string3 to INT8
INT8 Sample Program over.
```

5.2.58 ifx_int8cmp() 函数

ifx_int8cmp() 函数比较两个 int8 类型数值。

语法

```
mint ifx_int8cmp(n1, n2)
        ifx_int8_t *n1;
        ifx_int8_t *n2;
```

n1

指向包含要比较的第一个数值的 **int8** 结构的指针。

n2

指向包含要比较的第二个数值的 **int8** 结构的指针。

返回代码

-1

第一个值小于第二个值。

0

两个值相同。

1

第一个值大于第二个值。

INT8UNKNOWN

有的值为空。

示例

demo 目录中的文件 int8cmp.ec 包含下列示例程序。

```
/*
        * ifx_int8cmp.ec *
The following program compares INT8t types and displays
the results.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "-999,888,777,666";
        char string2[] = "-12,345,678,956,546";
        char string3[] = "123,456,780,555,224,456";
        char string4[] = "123,456,780,555,224,456";
        char string5[] = "";

        main()
        {
        mint x;
        ifx_int8_t num1, num2, num3, num4, num5;

    printf("IFX_INT8CMP Sample ESQL Program running.\n\n");

 if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
```

```
                printf("Error %d in converting string1 to int8\n", x);
                exit(1);
                }
  if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
                {
                printf("Error %d in converting string2 to int8\n", x);
                exit(1);
                }
  if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
                {
                printf("Error %d in converting string3 to int8\n", x);
                exit(1);
                }
if (x = ifx_int8cvasc(string4, strlen(string4), &num4))
                {
                printf("Error %d in converting string4 to int8\n", x);
                exit(1);
                }
if (x = ifx_int8cvasc(string5, strlen(string5), &num5))
                {
                printf("Error %d in converting string5 to int8\n", x);
                exit(1);
                }
                printf("num1 = %s\nnum2 = %s\n", string1, string2);
                printf("num3 = %s\nnum4 = %s\n", string3, string4);
                printf("num5 = %s\n", "NULL");
                printf("\nExecuting: ifx_int8cmp(&num1, &num2)\n");
                printf("    Result = %d\n", ifx_int8cmp(&num1, &num2));
                printf("Executing: ifx_int8cmp(&num2, &num3)\n");
                printf("    Result = %d\n", ifx_int8cmp(&num2, &num3));
                printf("Executing: ifx_int8cmp(&num1, &num3)\n");
                printf("    Result = %d\n", ifx_int8cmp(&num1, &num3));
                printf("Executing: ifx_int8cmp(&num3, &num4)\n");
```

```
    printf("    Result = %d\n", ifx_int8cmp(&num3, &num4));
    printf("Executing: ifx_int8cmp(&num1, &num5)\n");
    x = ifx_int8cmp(&num1, &num5);
    if(x == INT8UNKNOWN)
    printf("RESULT is INT8UNKNOWN.    One of the INT8 values in null.\n");
    else
    printf("    Result = %d\n", x);
    printf("\nIFX_INT8CMP Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
IFX_INT8CMP Sample ESQL Program running.

        Number 1 = -999,888,777,666
        Number 2 = -12,345,678,956,546
        Number 3 = 123,456,780,555,224,456
        Number 4 = 123,456,780,555,224,456
        Number 5 =

        Executing: ifx_int8cmp(&num1, &num2)
        Result = 1
        Executing: ifx_int8cmp(&num2, &num3)
        Result = -1
        Executing: ifx_int8cmp(&num1, &num3)
        Result = -1
        Executing: ifx_int8cmp(&num3, &num4)
        Result = 0
        Executing: ifx_int8cmp(&num1, &num5)
        RESULT is INT8UNKNOWN.    One of the INT8 values in null.
        IFX_INT8CMP Sample Program over.
```

5.2.59 ifx_int8copy() 函数

ifx_int8copy() 函数将一个 int8 结构复制至另一个。

语法

void ifx_int8copy(*source*, *target*)

ifx_int8_t *source;

ifx_int8_t *target;

source

指向包含要复制的源 **int8** 值的 **int8** 结构的指针。

target

指向目标 **int8** 结构的指针。

ifx_int8copy() 函数不返回状态值。要确定该复制操作是否成功，请查看 *target* 参数指向的 **int8** 结构的内容。

示例

demo 目录中的文件 int8copy.ec 包含下列样例程序。

```
/*
        * ifx_int8copy.ec *

The following program copies one INT8 number to another.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "-12,888,999,555,333";
        char result[41];

        main()
        {
        mint x;
        ifx_int8_t num1, num2;

    printf("IFX_INT8COPY Sample ESQL Program running.\n\n");
```

```
        printf("String = %s\n", string1);
    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
        printf("Executing: ifx_int8copy(&num1, &num2)\n");
        ifx_int8copy(&num1, &num2);
        if (x = ifx_int8toasc(&num2, result, sizeof(result)))
        {
        printf("Error %d in converting num2 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("Destination = %s\n", result);

        printf("\nIFX_INT8COPY Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
IFX_INT8COPY Sample ESQL Program running.

        String = -12,888,999,555,333
        Executing: ifx_int8copy(&num1, &num2)
        Destination = -12888999555333
IFX_INT8COPY Sample Program over.
```

5.2.60 ifx_int8cvasc() 函数

ifx_int8cvasc() 函数将 C char 类型中保存作为可打印的字符的值转换为 int8 类型数值。

语法

```
mint ifx_int8cvasc(strng_val, len, int8_val)
```

```
char *strng_val
mint len;
ifx_int8_t *int8_val;
```

strng_val

指向字符串的指针。

len

strng_val 字符串的长度。

int8_val

指向 ifx_int8cvasc() 放置转换的结果处的 **int8** 结构的指针。

用法

字符串 *strng_val* 可包含下列符号：

开头的符号，或为正（+）或为负（-）。

在 e 或 E 前面的指数。您可在该指数前面加符号，或为正（+）或为负（-）。

strng_val 字符串在小数点分隔符的右边不包含小数点分隔符或数字。ifx_int8cvasc() 函数截断小数点分隔符右边的小数点分隔符和任何数字。ifx_int8cvasc() 函数忽略该字符串中开头的空格。

当您使用非缺省的语言环境（US English 之外的一种）时，ifx_int8cvasc() 支持 *strng_val* 字符串中的非 ASCII 字符。

返回代码

0

转换成功。

-1213

字符串有非数值的字符。

-1284

操作导致溢出或下溢。

示例

demo 目录中的文件 int8cvasc.ec 包含下列样例程序。

```
/*
```

```
                * ifx_in8cvasc.ec *
The following program converts three strings to INT8
types and displays the values stored in each field of
 the INT8 structures.
                */

                #include <stdio.h>

                EXEC SQL include "int8.h";

                char string1[] = "-12,555,444,333,786,456";
                char string2[] = "480";
                char string3[] = "5.2";
                main()
                {
                mint x;
                ifx_int8_t num1, num2, num3;
                void nullterm(char *, mint);

 printf("IFX_INT8CVASC Sample ESQL Program running.\n\n");

if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
                {
                printf("Error %d in converting string1 to INT8\n", x);
                exit(1);
                }
 if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
                {
                printf("Error %d in converting string2 to INT8\n", x);
                exit(1);
                }
if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
                {
```

```
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

/*  Display the exponent, sign value and number of digits in num1. */

        ifx_int8toasc(&num1, string1, sizeof(string1));
        nullterm(string1, sizeof(string1));
    printf("The value of the first INT8 is = %s\n", string1);

/*  Display the exponent, sign value and number of digits in num2. */

        ifx_int8toasc(&num2, string2, sizeof(string2));
        nullterm(string2, sizeof(string2));
        printf("The value of the 2nd INT8 is = %s\n", string2);

/*  Display the exponent, sign value and number of digits in num3. */
 /*  Note that the decimal is truncated */

        ifx_int8toasc(&num3, string3, sizeof(string3));
        nullterm(string3, sizeof(string3));
        printf("The value of the 3rd INT8 is = %s\n", string3);

        printf("\nIFX_INT8CVASC Sample Program over.\n\n");
        exit(0);
        }
        void nullterm(char *str, mint size)
        {
        char *end;

        end = str + size;
        while(*str && *str > ' ' && str <= end)
        ++str;
```

```
        *str = '\0';
        }
```

输出

```
IFX_INT8CVASC Sample ESQL Program running.

        The value of the first INT8 is = -12555444333786456
        The value of the 2nd INT8 is = 480
        The value of the 3rd INT8 is = 5
```

5.2.61 ifx_int8cvdbl() 函数

ifx_int8cvdbl() 函数将 C double 类型数值转换为 int8 类型数值。

语法

```
mint ifx_int8cvdbl(dbl_val, int8_val)
        double dbl_val;
        ifx_int8_t *int8_val;
```

dbl_val

ifx_int8cvdbl() 将其转换为 **int8** 类型值的 **double** 值。

int8_val

指向 ifx_int8cvdbl() 放置转换结果处的 **int8** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8cvdbl.ec 包含下列样例程序。

```
/*
        * int8cvdbl.ec *
```

The following program converts two double type numbers to INT8 types and displays the results.

```
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
mint x;
ifx_int8_t num;
double d = 2147483647;

printf("IFX_INT8CVDBL Sample ESQL Program running.\n\n");

printf("Number 1 (double) = 1234.5678901234\n");
if (x = ifx_int8cvdbl((double)1234.5678901234, &num))
{
printf("Error %d in converting double1 to INT8\n", x);
exit(1);
}
if (x = ifx_int8toasc(&num, result, sizeof(result)))
{
printf("Error %d in converting INT8 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    String Value = %s\n", result);
/* notice that the ifx_int8cvdbl function truncates digits to the    right of a decimal separator.
*/
```

```
        printf("Number 2 (double) = %.1f\n", d);
        if (x = ifx_int8cvdbl(d, &num))
        {
        printf("Error %d in converting double2 to INT8\n", x);
        exit(1);
        }
        if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
    printf("Error %d in converting second INT8 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("    String Value = %s\n", result);

        printf("\nIFX_INT8CVDBL Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
IFX_INT8CVDBL Sample ESQL Program running.

        Number 1 (double) = 1234.5678901234
        String Value = 1234
        Number 2 (double) = 2147483647.0
        String Value = 2147483647
IFX_INT8CVDBL Sample Program over.\
```

5.2.62 ifx_int8cvdec() 函数

ifx_int8cvdec() 函数将 decimal 类型值转换为 int8 类型值。

语法

```
mint ifx_int8cvdec(dec_val, int8_val)
```

```
        dec_t *dec_val;
        ifx_int8_t *int8_val;
```

dec_val

指向 ifx_int8cvdec() 转换为 **int8** 类型值的 **decimal** 结构的指针。

int8_val

指向 ifx_int8cvdec() 放置转换的结果处的 **int8** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8cdec.ec 包含下列样例程序。

```
/*
        * ifx_int8cvdec.ec *

The following program converts two INT8t types to DECIMALS and displays the results.
        */

        #include <stdio.h>

        EXEC SQL include decimal;
        EXEC SQL include "int8.h";

        char string1[] = "2949.3829398204382";
        char string2[] = "3238299493";
        char result[41];

        main()
        {
```

```
        mint x;
        ifx_int8_t n;
        dec_t num;

    printf("IFX_INT8CVDEC Sample ESQL Program running.\n\n");

        if (x = deccvasc(string1, strlen(string1), &num))
        {
      printf("Error %d in converting string1 to DECIMAL\n", x);
        exit(1);
        }
        if (x = ifx_int8cvdec(&num, &n))
        {
        printf("Error %d in converting DECIMAL1 to INT8\n", x);
        exit(1);
        }

    /*  Convert the INT8 to ascii and display it. Note that the digits to the right of the decimal
are truncated in the conversion.
        */

        if (x = ifx_int8toasc(&n, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("String 1 Value = %s\n", string1);
        printf("    INT8 type value = %s\n", result);
        if (x = deccvasc(string2, strlen(string2), &num))
        {
    printf("Error %d in converting string2 to DECIMAL\n", x);
        exit(1);
```

```
}
if (x = ifx_int8cvdec(&num, &n))
{
printf("Error %d in converting DECIMAL2 to INT8\n", x);
exit(1);
}
printf("String 2 = %s\n", string2);

/* Convert the INT8 to ascii to display value. */

if (x = ifx_int8toasc(&n, result, sizeof(result)))
{
printf("Error %d in converting INT8 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    INT8 type value = %s\n", result);

printf("\nIFX_INT8CVDEC Sample Program over.\n\n");
exit(0);
}
```

输出

```
IFX_INT8CVDEC Sample ESQL Program running.

    String 1 Value = 2949.3829398204382
    INT8 type value = 2949
    String 2 = 3238299493
    INT8 type value = 3238299493
IFX_INT8CVDEC Sample Program over.
```

5.2.63 ifx_int8cvflt() 函数

ifx_int8cvflt() 函数将 C float 类型数值转换为 int8 类型数值。

语法

mint ifx_int8cvflt(*flt_val*, *int8_val*)

double flt_val;

ifx_int8_t *int8_val;

flt_val

ifx_int8cvflt() 要将其转换为 **int8** 类型值的 **float** 值。

int8_val

指向 ifx_int8cvflt() 放置转换的结果处的 **int8** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8cvflt.ec 包含下列样例程序。

```
/*
        * ifx_int8cvflt.ec *
The following program converts two floats to INT8 types and displays the results.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char result[41];

        main()
        {
        mint x;
        ifx_int8_t num;
```

```
        printf("IFX_INT8CVFLT Sample ESQL Program running.\n\n");

                printf("Float 1 = 12944.321\n");

        /* Note that in the following conversion, the digits to the right of the decimal are ignored.
*/

                if (x = ifx_int8cvflt(12944.321, &num))
                {
                printf("Error %d in converting float1 to INT8\n", x);
                exit(1);
                }

                /* Convert int8 to ascii to display value. */

                if (x = ifx_int8toasc(&num, result, sizeof(result)))
                {
                printf("Error %d in converting INT8 to string\n", x);
                exit(1);
                }
                result[40] = '\0';
                printf("    The INT8 type value is = %s\n", result);
                printf("Float 2 = -33.43\n");

        /* Note that in the following conversion, the digits to the right of the decimal are ignored.
*/

                if (x = ifx_int8cvflt(-33.43, &num))
                {
                printf("Error %d in converting float2 to INT8\n", x);
                exit(1);
                }
                if (x = ifx_int8toasc(&num, result, sizeof(result)))
```

```
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
 printf("    The second INT8 type value is = %s\n", result);

        printf("\nIFX_INT8CVFLT Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
IFX_INT8CVFLT Sample ESQL Program running.

        Float 1 = 12944.321
        The INT8 type value is = 12944
        Float 2 = -33.43
        The second INT8 type value is = -33
IFX_INT8CVFLT Sample Program over.
```

5.2.64 ifx_int8cvint() 函数

ifx_int8cvint() 函数将 C int 类型数值转换为 int8 类型数值。

语法

```
mint ifx_int8cvint(int_val, int8_val)
        mint int_val;
        ifx_int8_t *int8_val;
```

int_val

ifx_int8cvint() 将其转换为 **int8** 类型值的 **mint** 值。

int8_val

指向 ifx_int8cvint() 放置转换的结果处的 **int8** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8cvint.ec 包含下列样例程序。

```
/*
        * ifx_int8cvint.ec *

The following program converts two integers to INT8
types and displays the results.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char result[41];

        main()
        {
        mint x;
        ifx_int8_t num;

        printf("IFX_INT8CVINT Sample ESQL Program running.\n\n");

        printf("Integer 1 = 129449233\n");
        if (x = ifx_int8cvint(129449233, &num))
        {
        printf("Error %d in converting int1 to INT8\n", x);
        exit(1);
        }
```

```
/* Convert int8 to ascii to display value. */

if (x = ifx_int8toasc(&num, result, sizeof(result)))
{
printf("Error %d in converting INT8 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    The INT8 type value is = %s\n", result);

printf("Integer 2 = -33\n");
if (x = ifx_int8cvint(-33, &num))
{
printf("Error %d in converting int2 to INT8\n", x);
exit(1);
}

/* Convert int8 to ascii to display value. */

if (x = ifx_int8toasc(&num, result, sizeof(result)))
{
printf("Error %d in converting INT8 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    The second INT8 type value is = %s\n", result);

printf("\nIFX_INT8CVINT Sample Program over.\n\n");
exit(0);
}
```

输出

```
IFX_INT8CVINT Sample ESQL Program running.
```

Integer 1 = 129449233

The INT8 type value is = 129449233

Integer 2 = -33

The second INT8 type value is = -33

5.2.65 ifx_int8cvlong() 函数

ifx_int8cvlong() 函数将 C **long** 类型值转换为 **int8** 类型值。

语法

mint ifx_int8cvlong(*lng_val*, *int8_val*)

int4 lng_val;

ifx_int8_t *int8_val;

lng_val

ifx_int8cvlong() 将其转换为 **int8** 类型值的 **int4** 整数。

int8_val

指向 ifx_int8cvlong() 放置转换的结果处的 **int8** 结构的指针。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8cvlong.ec 包含下列样例程序。

```
/*
        * ifx_int8cvlong.ec *

        The following program converts two longs to INT8
        types and displays the results.
        */
```

```
#include <stdio.h>

EXEC SQL include "int8.h";

char result[41];

main()
{
mint x;
ifx_int8_t num;
int4 n;

printf("IFX_INT8CVLONG Sample ESQL Program running.\n\n");

printf("Long Integer 1 = 129449233\n");
if (x = ifx_int8cvlong(129449233L, &num))
{
printf("Error %d in converting long to INT8\n", x);
exit(1);
}
if (x = ifx_int8toasc(&num, result, sizeof(result)))
{
printf("Error %d in converting INT8 to string\n", x);
exit(1);
}
result[40] = '\0';
printf("    String for INT8 type value = %s\n", result);

n = 2147483646;                    /* set n */
printf("Long Integer 2 = %d\n", n);
if (x = ifx_int8cvlong(n, &num))
{
printf("Error %d in converting long to INT8\n", x);
```

```
        exit(1);
        }
        if (x = ifx_int8toasc(&num, result, sizeof(result)))
        {
        printf("Error %d in converting INT8 to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("      String for INT8 type value = %s\n", result);

        printf("\nIFX_INT8CVLONG Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
IFX_INT8CVLONG Sample ESQL Program running.

        Long Integer 1 = 129449233
        String for INT8 type value = 129449233
        Long Integer 2 = 2147483646
        String for INT8 type value = 2147483646
IFX_INT8CVLONG Sample Program over.
```

5.2.66 ifx_int8div() 函数

ifx_int8div() 函数将两个 int8 类型值相除。

语法

```
mint ifx_int8div(n1, n2, quotient)
        ifx_int8_t *n1;
        ifx_int8_t *n2;
        ifx_int8_t *quotient;
```

n1

指向包含被除数的 **int8** 结构的指针。

n2

指向包含除数的 **int8** 结构的指针。

quotient

指向包含 *n1*/*n2* 的商的 **int8** 结构的指针。

用法

quotient 可与 n1 或 n2 相同。

返回代码

0

运算成功。

-1202

运算尝试除以零。

示例

demo 目录中的文件 int8div.ec 包含下列样例程序。

```
/*
        * ifx_int8div.ec *
    The following program divides two INT8 numbers and displays the result.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "480,999,777,666,345,567";
        char string2[] = "80,765,456,765,456,654";
        char result[41];

        main()
        {
        mint x;
        ifx_int8_t num1, num2, dvd;
```

```
printf("IFX_INT8DIV Sample ESQL Program running.\n\n");

        if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
            {
            printf("Error %d in converting string1 to INT8\n", x);
            exit(1);
            }
        if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
            {
            printf("Error %d in converting string2 to INT8\n", x);
            exit(1);
            }
            if (x = ifx_int8div(&num1, &num2, &dvd))
            {
            printf("Error %d in dividing num1 by num2\n", x);
            exit(1);
            }
            if (x = ifx_int8toasc(&dvd, result, sizeof(result)))
            {
        printf("Error %d in converting dividend to string\n", x);
            exit(1);
            }
            result[40] = '\0';
            printf("\t%s / %s = %s\n", string1, string2, result);

            printf("\nIFX_INT8DIV Sample Program over.\n\n");
            exit(0);
            }
```

输出

```
IFX_INT8DIV Sample ESQL Program running.

        480,999,777,666,345,567 / 80,765,456,765,456,654 = 5
```

IFX_INT8DIV Sample Program over.

5.2.67 ifx_int8mul() 函数

ifx_int8mul() 函数将两个 int8 类型值相乘。

语法

mint ifx_int8mul(*n1*, *n2*, *product*)

ifx_int8_t *n1;

ifx_int8_t *n2;

ifx_int8_t *product;

n1

指向包含第一个操作对象的 **int8** 结构的指针。

n2

指向包含第二个操作对象的 **int8** 结构的指针。

product

指向包含 *n1* * *n2* 的乘积的 **int8** 结构的指针。

用法

product 可与 *n1* 或 *n2* 相同。

返回代码

0

运算成功。

-1284

运算导致溢出或下溢。

示例

demo 目录中的文件 int8mul.ec 包含下列样例程序。

```
/*
        * ifx_int8mul.ec *

        The following program multiplies two INT8 numbers and
```

```
displays the result.
*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "480,999,777,666,345";
char string2[] = "80";
char result[41];

main()
{
mint x;
ifx_int8_t num1, num2, prd;

printf("IFX_INT8MUL Sample ESQL Program running.\n\n");

    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
        if (x = ifx_int8mul(&num1, &num2, &prd))
        {
        printf("Error %d in multiplying num1 by num2\n", x);
        exit(1);
        }
```

```
    if (x = ifx_int8toasc(&prd, result, sizeof(result)))
    {
printf("Error %d in converting product to string\n", x);
    exit(1);
    }
    result[40] = '\0';
    printf("\t%s * %s = %s\n", string1, string2, result);

    printf("\nIFX_INT8MUL Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
IFX_INT8MUL Sample ESQL Program running.

        480,999,777,666,345 * 80 = 38479982213307600

IFX_INT8MUL Sample Program over.
```

5.2.68 ifx_int8tub() 函数

ifx_int8tub() 函数将两个 int8 类型值相减。

语法

```
mint ifx_int8tub(n1, n2, difference)
        ifx_int8_t *n1;
        ifx_int8_t *n2;
        ifx_int8_t *difference;
```

n1

指向包含第一个运算对象的 **int8** 结构的指针。

n2

指向包含第二个运算对象的 **int8** 结构的指针。

difference

指向包含 *n1* 与 *n2* 的差（*n1 - n2*）的 **int8** 结构的指针。

用法

difference 可与 n1 或 n2 相同。

返回代码

0

减法成功。

-1284

减法导致溢出或下溢。

示例

demo 目录中的文件 int8tub.ec 包含下列样例程序。

```
/*
        *int8tub.ec *

 The following program obtains the difference of two INT8    type values.
        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "6";
        char string2[] = "9,223,372,036,854,775";
        char string3[] = "999,999,999,999,999.5";
        char result[41];

        main()
        {
        mint x;
        ifx_int8_t num1, num2, num3, sum;

      printf("IFX_INT8tUB Sample ESQL Program running.\n\n");

       if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
```

```
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
    if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }

        /* subtract num2 from num1 */

        if (x = ifx_int8tub(&num1, &num2, &sum))
        {
        printf("Error %d in subtracting INT8t\n", x);
        exit(1);
        }
        if (x = ifx_int8toasc(&sum, result, sizeof(result)))
        {
printf("Error %d in converting INT8 result to string\n", x);
        exit(1);
        }
        result[40] = '\0';
        printf("\t%s - %s = %s\n", string1, string2, result);
        /* display result */

      if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

/* notice that digits right of the decimal are truncated. */
```

```
        if (x = ifx_int8tub(&num2, &num3, &sum))
        {
        printf("Error %d in subtracting INT8t\n", x);
        exit (1);
        }
        if (x = ifx_int8toasc(&sum, result, sizeof(result)))
        {
printf("Error %d in converting INT8 result to string\n", x);
        exit(1);
        }
        result[40] = '\0';
       printf("\t%s - %s = %s\n", string2, string3, result);
       /* display result */

        printf("\nIFX_INT8tUB Sample Program over.\n\n");
        exit(0);
        }
```

输出

```
IFX_INT8tUB Sample ESQL Program running.

        6 - 9,223,372,036,854,775 = -9223372036854769
        9,223,372,036,854,775 - 999,999,999,999,999.5
        = 8223372036854776
    IFX_INT8tUB Sample Program over.
```

5.2.69 ifx_int8toasc() 函数

ifx_int8toasc() 函数将 int8 类型数值转换为 C char 类型值。

语法

```
mint ifx_int8toasc(int8_val, strng_val, len)
        ifx_int8_t *int8_val;
```

```
char    *strng_val;
mint    len;
```

int8_val

指向 ifx_int8toasc() 将其值转换为文本字符串的 **int8** 结构的指针。

strng_val

指向 ifx_int8toasc() 放置文本字符串处的字符缓冲区的第一个字节的指针。

len

以字节计算的 **strng_val** 的大小，对于空终止符，为负 1。

用法

如果 **int8** 数值不适于放入长度 *len* 的字符串内，则 ifx_int8toasc() 将该数值转换为指数计数法。如果该值仍不适合，则 ifx_int8toasc() 以星号填充字符串。 如果该数值比字符串短，则 ifx_int8toasc() 向左对齐该数值，并用空格填充它的右边。

由于 ifx_int8toasc() 返回的字符串不是以空终止的，因此，在您打印它之前，您必须将空字符添加到该字符串。

当您使用非缺省的语言环境（US English 之外的一种）时，ifx_int8toasc() 支持 *strng_val* 字符串中的非 ASCII 字符。要获取更多信息，

返回代码

0

转换成功。

-1207

被转换的值不适于放入分配了的空间内。

示例

demo 目录中的文件 int8toasc.ec 包含下列样例程序。

```
/*
        * ifx_int8toasc.ec *

        The following program converts three string
        constants to INT8 types and then uses ifx_int8toasc()
        to convert the INT8 values to C char type values.
        */

        #include <stdio.h>
        #define END sizeof(result)
```

```
EXEC SQL include "int8.h";

char string1[] = "-12,555,444,333,786,456";
char string2[] = "480";
char string3[] = "5.2";
char result[40];

main()
{
mint x;
ifx_int8_t num1, num2, num3;

 printf("IFX_INT8tOASC Sample ESQL Program running.\n\n");

if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
{
printf("Error %d in converting string1 to INT8\n", x);
exit(1);
}
 if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
{
printf("Error %d in converting string2 to INT8\n", x);
exit(1);
}
 if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
{
printf("Error %d in converting string3 to INT8\n", x);
exit(1);
}
printf("\nConverting INT8 back to ASCII\n");
  printf(" Executing: ifx_int8toasc(&num1, result, END - 1)");
if (x = ifx_int8toasc(&num1, result, END - 1))
```

```
                printf("\tError %d in converting INT8 to string\n", x);
                else
                {
                result[END - 1] = '\0';                    /* null terminate */
    printf("\n The value of the first INT8 is = %s\n", result);
                }
                printf("\nConverting second INT8 back to ASCII\n");
  printf(" Executing: ifx_int8toasc(&num2, result, END - 1)");
                if (x= ifx_int8toasc(&num2, result, END - 1))
                printf("\tError %d in converting INT8 to string\n", x);
                else
                {
                result[END - 1] = '\0';                    /* null terminate */
                printf("\n The value of the 2nd INT8 is = %s\n", result);
                }

                printf("\nConverting third INT8 back to ASCII\n");
printf(" Executing: ifx_int8toasc(&num3, result, END - 1)");
                /* note that the decimal is truncated */

                if (x= ifx_int8toasc(&num3, result, END - 1))
                printf("\tError %d in converting INT8 to string\n", x);
                else
                {
                result[END - 1] = '\0';                /* null terminate */
                printf("\n The value of the 3rd INT8 is = %s\n", result);
                }
                printf("\nIFX_INT8tOASC Sample Program over.\n\n");
                exit(0);
                }
```

输出

```
IFX_INT8tOASC Sample ESQL Program running.
```

```
Converting INT8 back to ASCII
Executing: ifx_int8toasc(&num1, result, sizeof(result)-1)
The value of the first INT8 is = -12555444333786456

Converting second INT8 back to ASCII
Executing: ifx_int8toasc(&num2, result, sizeof(result)-1)
The value of the 2nd INT8 is = 480

Converting third INT8 back to ASCII
Executing: ifx_int8toasc(&num3, result, END)
The value of the 3rd INT8 is = 5
```

5.2.70 ifx_int8todbl() 函数

ifx_int8todbl() 函数将 int8 类型数值转换为 C double 类型数值。

语法

```
mint ifx_int8todbl(int8_val, dbl_val)
        ifx_int8_t *int8_val;
        double *dbl_val;
```

int8_val

指向 ifx_int8todbl() 将其值转换为 **double** 类型值的 **int8** 结构的指针。

dbl_val

指向 ifx_int8todbl() 放置转换的结果处的 **double** 值的指针。

用法

在将 **int8** 类型数值转换为 **double** 类型数值过程中，主计算机的浮点格式可导致精度的损失。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8todbl.ec 包含下列样例程序。

```
/*
        * ifx_int8todbl.ec *

        The following program converts three strings to INT8
        types and then to C double types and displays the
        results.

        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "-12,555,444,333,786,456";
        char string2[] = "480";
        char string3[] = "5.2";

        main()
        {
        mint x;
        double d =0;
        ifx_int8_t num1, num2, num3;

        printf("\nIFX_INT8tODBL Sample ESQL Program running.\n\n");

        if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
```

```
exit(1);
}
if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
{
printf("Error %d in converting string2 to INT8\n", x);
exit(1);
}
if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
{
printf("Error %d in converting string3 to INT8\n", x);
exit(1);
}
printf("\nConverting INT8 to double");
if (x= ifx_int8todbl(&num1, &d))
{
printf("\tError %d in converting INT8 to double\n", x);
exit(1);
}
else
{
printf("\nString 1= %s\n", string1);
printf("INT8 value is = %.10f\n", d);
}
printf("\nConverting second INT8 to double");
if (x= ifx_int8todbl(&num2, &d))
{
printf("\tError %d in converting INT8 to double\n", x);
exit(1);
}
else
{
printf("\nString2 = %s\n", string2);/*
printf("INT8 value is =    %.10f\n",d);
```

```
}
printf("\nConverting third INT8 to double");
/* Note that the decimal places will be truncated. */

if (x= ifx_int8todbl(&num3, &d))
{
printf("\tError %d in converting INT8 to double\n", x);
exit(1);
}
else
{
printf("\nString3 = %s\n", string3);
printf("INT8 value is =    %.10f\n",d);
}
printf("\nIFX_INT8tODBL Sample Program over.\n\n");
exit(0);
}
```

输出

```
IFX_INT8tODBL Sample ESQL Program running.

        Converting INT8 to double

        Executing: ifx_int8todbl(&num1,&d)
        String 1= -12,555,444,333,786,456

        The value of the first double is = -12555444333786456.0000000000

        Converting second INT8 to double

        Executing: ifx_int8todbl(&num2, &d)
        String2 = 480

        The value of the second double is =    480.0000000000
```

Converting third INT8 to double

Executing: ifx_int8todbl(&num3, &d)

String3 = 5.2

The value of the third double is = 5.0000000000

5.2.71 ifx_int8todec() 函数

ifx_int8todec() 函数将 int8 类型数值转换为 decimal 类型数值。

语法

mint ifx_int8todec(*int8_val*, *dec_val*)

ifx_int8_t *int8_val;

dec_t *dec_val;

int8_val

指向 ifx_int8todec() 将其值转换为 **decimal** 类型值的 **int8** 结构的指针。

dec_val

指向 ifx_int8todec() 在其中放置转换的结果的 **decimal** 结构的指针。

返回代码

0

转换成功。

<0

转换不成功。

示例

demo 目录中的文件 int8todec.ec 包含下列样例程序。

```
/*
        * ifx_int8todec.ec *

  The following program converts three strings to INT8 types and
```

```
converts the INT8 type values to decimal type values.
Then the values are displayed.

        */

        #include <stdio.h>

        EXEC SQL include "int8.h";
        #define END sizeof(result)

        char string1[] = "-12,555,444,333,786,456";
        char string2[] = "480";
        char string3[] = "5.2";
        char result [40];

        main()
        {
        mint x;
        dec_t d;
        ifx_int8_t num1, num2, num3;
printf("IFX_INT8tODEC Sample ESQL Program running.\n\n");

      if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
       {
       printf("Error %d in converting string1 to INT8\n", x);
       exit(1);
       }
      if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
       {
       printf("Error %d in converting string2 to INT8\n", x);
       exit(1);
       }
      if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
```

```
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }

        printf("\n***Converting INT8 to decimal\n");
        printf("\nString 1= %s\n", string1);
        printf(" \nExecuting: ifx_int8todec(&num1,&d)");
        if (x= ifx_int8todec(&num1, &d))
        {
    printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
        else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");
        if (x = dectoasc(&d, result, END, -1))
printf("\tError %d in converting DECIMAL1 to string\n", x);
        else
        {
        result[END - 1] = '\0';                 /* null terminate */
        printf("Result = %s\n", result);
        }
        }
        printf("\n***Converting second INT8 to decimal\n");
        printf("\nString2 = %s\n", string2);
        printf(" \nExecuting: ifx_int8todec(&num2, &d)");
        if (x= ifx_int8todec(&num2, &d))
        {
    printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
```

```
        else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");
        if (x = dectoasc(&d, result, END, -1))
    printf("\tError %d in converting DECIMAL2 to string\n", x);
        else
        {
        result[END - 1] = '\0';                 /* null terminate */
        printf("Result = %s\n", result);
        }
        }
        printf("\n***Converting third INT8 to decimal\n");
        printf("\nString3 = %s\n", string3);
        printf(" \nExecuting: ifx_int8todec(&num3, &d)");
        if (x= ifx_int8todec(&num3, &d))
        {
      printf("\tError %d in converting INT8 to decimal\n", x);
        exit(1);
        }
        else
        {
        printf("\nConverting Decimal to ASCII for display\n");
        printf("Executing: dectoasc(&d, result, END, -1)\n");

        /* note that the decimal is truncated */

        if (x = dectoasc(&d, result, END, -1))
printf("\tError %d in converting DECIMAL3 to string\n", x);
        else
        {
        result[END - 1] = '\0';                 /* null terminate */
        printf("Result = %s\n", result);
```

```
    }
    }
    printf("\nIFX_INT8tODEC Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
IFX_INT8tODEC Sample ESQL Program running.


    ***Converting INT8 to decimal

    String 1= -12,555,444,333,786,456

    Executing: ifx_int8todec(&num1,&d)
    Converting Decimal to ASCII for display
    Executing: dectoasc(&d, result, END, -1)
    Result = -12555444333786456.0

    ***Converting second INT8 to decimal

    String2 = 480

    Executing: ifx_int8todec(&num2, &d)
    Converting Decimal to ASCII for display
    Executing: dectoasc(&d, result, END, -1)
    Result = 480.0

    ***Converting third INT8 to decimal

    String3 = 5.2

    Executing: ifx_int8todec(&num3, &d)
    Converting Decimal to ASCII for display
```

Executing: dectoasc(&d, result, END, -1)

Result = 5.0

IFX_INT8tODEC Sample Program over.

5.2.72 ifx_int8toflt() 函数

ifx_int8toflt() 函数将 int8 类型数值转换为 **C float** 类型数值。

语法

mint ifx_int8toflt(*int8_val*, *flt_val*)

ifx_int8_t *int8_val;

float *flt_val;

int8_val

指向 ifx_int8toflt() 将其值转换为 **float** 类型值的 **int8** 结构的指针。

flt_val

指向 ifx_int8toflt() 放置转换的结果处的 **float** 值的指针。

用法

ifx_int8toflt() 库函数将 **int8** 值转换为 C float。C float 的大小依赖于您正在使用的计算机的硬件和操作系统。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8toflt.ec 包含下列样例程序。

```
/*
        * ifx_int8toflt.ec *

        The following program converts three strings to
        INT8 values and then to float values and
```

```
displays the results.

*/

#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,555.765";
char string2[] = "480.76";
char string3[] = "5.2";

main()
{
mint x;
float f =0.0;
ifx_int8_t num1, num2, num3;

printf("\nIFX_INT8tOFLT Sample ESQL Program running.\n\n");
    if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
      if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
      if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
```

```
}

printf("\nConverting INT8 to float\n");
if (x= ifx_int8toflt(&num1, &f))
{
printf("\tError %d in converting INT8 to float\n", x);
exit(1);
}
else
{
printf("String 1= %s\n", string1);
printf("INT8 value is = %f\n", f);
}
printf("\nConverting second INT8 to float\n");
if (x= ifx_int8toflt(&num2, &f))
{
printf("\tError %d in converting INT8 to float\n", x);
exit(1);
}
else
{
printf("String2 = %s\n", string2);
printf("INT8 value is =    %f\n", f);
}
printf("\nConverting third INT8 to integer\n");

/* Note that the decimal places will be truncated */

if (x= ifx_int8toflt(&num3, &f))
{
printf("\tError %d in converting INT8 to float\n", x);
exit(1);
}
```

```
else
{
printf("String3 = %s\n", string3);
printf("INT8 value is =    %f\n",f);
}
printf("\nIFX_INT8tOFLT Sample Program over.\n\n");
exit(0);
}
```

输出

```
IFX_INT8tOFLT Sample ESQL Program running.

        Converting INT8 to float

        Executing: ifx_int8toflt(&num1,&f)
        String 1= -12,555.765
        The value of the first float is = -12555.000000


        Converting second INT8 to float

        Executing: ifx_int8toflt(&num2, &f)
        String2 = 480.76
        The value of the second float is =    480.000000


        Converting third INT8 to integer

        Executing: ifx_int8toflt(&num3, &f)
        String3 = 5.2
        The value of the third float is =    5.000000
IFX_INT8tOFLT Sample Program over.
```

5.2.73 ifx_int8toint() 函数

ifx_int8toint() 函数将 int8 类型数值转换为 C int 类型数值。

语法

mint ifx_int8toint(*int8_val*, *int_val*)

ifx_int8_t *int8_val;

mint *int_val;

int8_val

指向 ifx_int8toint() 将其值转换为 **mint** 类型值的 **int8** 结构的指针。

int_val

指向 ifx_int8toint() 放置转换的结果处的 **mint** 值的指针。

用法

ifx_int8toint() 库函数将 **int8** 值转换为 C 整数。C 整数的大小依赖于您正在使用的计算机的硬件和操作系统。因此，ifx_int8toint() 函数将整数值与 SQL SMALLINT 数据类型同等看待。SMALLINT 的有效范围在 32767 与 -32767 之间。要将较大的 **int8** 值转换为较大的整数，请使用 ifx_int8tolong() 库函数。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录中的文件 int8toint.ec 包含下列样例程序。

```
/*
        * ifx_int8toint.ec *
The following program converts three strings to INT8 types and
converts the INT8 type values to C integer type values.
Then the values are displayed.

        */
```

```
#include <stdio.h>

EXEC SQL include "int8.h";

char string1[] = "-12,555";
char string2[] = "480";
char string3[] = "5.2";

main()
{
mint x;
mint i =0;
ifx_int8_t num1, num2, num3;

printf("IFX_INT8tOINT Sample ESQL Program running.\n\n");
        if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
        {
        printf("Error %d in converting string1 to INT8\n", x);
        exit(1);
        }
        if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
        {
        printf("Error %d in converting string2 to INT8\n", x);
        exit(1);
        }
        if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
        {
        printf("Error %d in converting string3 to INT8\n", x);
        exit(1);
        }
        printf("\nConverting INT8 to integer\n");
        if (x= ifx_int8toint(&num1, &i))
        {
```

```
                printf("\tError %d in converting INT8 to integer\n", x);
                 exit(1);
                 }
                 else
                 {
                 printf("String 1= %s\n", string1);
                 printf("INT8 value is = %d\n", i);
                 }
                 printf("\nConverting second INT8 to integer\n");
                 if (x= ifx_int8toint(&num2, &i))
                 {
                printf("\tError %d in converting INT8 to integer\n", x);
                 exit(1);
                 }
                 else
                 {
                 printf("String2 = %s\n", string2);
                 printf("INT8 value is =    %d\n", i);
                 }
                 printf("\nConverting third INT8 to integer\n");

                 /* note that the decimal will be truncated */

                 if (x= ifx_int8toint(&num3, &i))
                 {
    printf("\tError %d in converting INT8 to integer\n", x);
                 exit(1);
                 }
                 else
                 {
                 printf("String3 = %s\n", string3);
                 printf("INT8 value is =    %d\n",i);
                 }
```

```
    printf("\nIFX_INT8tOINT Sample Program over.\n\n");
    exit(0);
    }
```

输出

```
IFX_INT8tOINT Sample ESQL Program running.
        Converting INT8 to integer

        Executing: ifx_int8toint(&num1,&i)
        String 1= -12,555
        The value of the first integer is = -12555


        Converting second INT8 to integer

        Executing: ifx_int8toint(&num2, &i)
        String2 = 480
        The value of the second integer is =    480


        Converting third INT8 to integer

        Executing: ifx_int8toint(&num3, &i)
        String3 = 5.2
        The value of the third integer is =    5
IFX_INT8tOINT Sample Program over.
```

5.2.74 ifx_int8tolong() 函数

ifx_int8tolong() 函数将 int8 类型数值转换为 C long 类型数值。

语法

```
mint ifx_int8tolong(int8_val, lng_val)
        ifx_int8_t *int8_val;
        int4    *lng_val;
```

int8_val

指向 ifx_int8tolong() 将其值转换为 **int4** 整数类型值的 **int8** 结构的指针。

lng_val

指向 ifx_int8tolong() 放置转换的结果处的 **int4** 整数的指针。

返回代码

0

转换成功。

-1200

int8 类型的数值的大小大于 2,147,483,647。

示例

demo 目录中的文件 int8tolong.ec 包含下列样例程序。

```
/*
        * ifx_int8tolong.ec *
The following program converts three strings to INT8 types and converts the INT8 type values to C long type values.Then the values are displayed.

        */

        #include <stdio.h>

        EXEC SQL include "int8.h";

        char string1[] = "-1,555,345,698";
        char string2[] = "3,235,635";
        char string3[] = "553.24";

        main()
        {
        int x;
        long l =0;
        ifx_int8_t num1, num2, num3;
```

```
printf("IFX_INT8tOLONG Sample ESQL Program running.\n\n");

        if (x = ifx_int8cvasc(string1, strlen(string1), &num1))
          {
          printf("Error %d in converting string1 to INT8\n", x);
          exit(1);
          }
        if (x = ifx_int8cvasc(string2, strlen(string2), &num2))
          {
          printf("Error %d in converting string2 to INT8\n", x);
          exit(1);
          }
        if (x = ifx_int8cvasc(string3, strlen(string3), &num3))
          {
          printf("Error %d in converting string3 to INT8\n", x);
          exit(1);
          }
          printf("\nConverting INT8 to long\n");
          if (x= ifx_int8tolong(&num1, &l))
          {
          printf("\tError %d in converting INT8 to long\n", x);
          exit(1);
          }
          else
          {
          printf("String 1= %s\n", string1);
          printf("INT8 value is = %d\n", l);
          }

          printf("\nConverting second INT8 to long\n");
          if (x= ifx_int8tolong(&num2, &l))
          {
```

```
printf("\tError %d in converting INT8 to long\n", x);
exit(1);
}
else
{
printf("String2 = %s\n", string2);
printf("INT8 value is =    %d\n",l);
}
printf("\nConverting third INT8 to long\n");

/* Note that the decimal places will be truncated. */

if (x= ifx_int8tolong(&num3, &l))
{
printf("\tError %d in converting INT8 to long\n", x);
exit(1);
}
else
{
printf("String3 = %s\n", string3);
printf("INT8 value is =    %d\n",l);
}
printf("\nIFX_INT8tOLONG Sample Program over.\n\n");
exit(0);
}
```

输出

```
IFX_INT8tOLONG Sample ESQL Program running.

        Converting INT8 to long

        Executing: ifx_int8tolong(&num1,&l)
        String 1= -1,555,345,698
```

```
        The value of the first long is = -1555345698


        Converting second INT8 to long

        Executing: ifx_int8tolong(&num2, &l)
        String2 = 3,235,635
        The value of the second long is =    3235635


        Converting third INT8 to long

        Executing: ifx_int8tolong(&num3, &l)
        String3 = 553.24
        The value of the third long is =    553
IFX_INT8tOLONG Sample Program over.
```

5.2.75 ifx_lo_alter() 函数

ifx_lo_alter() 函数修改现有的智能大对象的存储特征。

语法

```
mint ifx_lo_alter(LO_ptr, LO_spec)
        ifx_lo_t *LO_ptr;
        ifx_lo_create_spec_t *LO_spec;
```

LO_ptr

指向标识其存储特征被更改的智能大对象的 LO-pointer 结构的指针。

LO_spec

指向包含存储特征的 LO-specification 结构的指针，ifx_lo_alter() 为 LO_*ptr* 指示的智能大对象保存该特征。

用法

ifx_lo_alter() 函数以 *LO_spec* 指向的 LO-specification 结构中的特征来更新现有的智能大对象的存储特征。使用 ifx_lo_alter()，您仅可更改下列存储特征：

日志记录特征

您可以 ifx_lo_specget_flags() 函数来设置 LO_LOG 或 LO_NOLOG 标志。

最后访问时间特征

您可以 ifx_lo_specset_flags() 函数来设置 LO_KEEP_LASTACCESS_TIME 或 LO_NOKEEP_LASTACCESS_TIME 标志。

Extent 大小

您可以 ifx_lo_specset_extsz() 函数来为分配 extent 大小存储新的整数值。在 ifx_lo_alter() 函数完成之后，新的 extent 大小仅适用于被写的 extent。

在它继续更新之前，该函数取得整个智能大对象的排他锁。它保持此锁，直到更新完成为止。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.76 ifx_lo_close() 函数

ifx_lo_close() 函数关闭打开的智能大对象。

语法

mint ifx_lo_close(*LO_fd*)

mint LO_fd;

LO_fd

要关闭的智能大对象的 LO 文件描述符。

用法

ifx_lo_close() 函数关闭与 LO 文件描述符 *LO_fd* 相关联的智能大对象。当 ifx_lo_open() 和 ifx_lo_create() 函数成功地打开智能大对象时，它们返回 LO 文件描述符。

当 ifx_lo_close() 函数关闭智能大对象时，数据库服务器尝试解锁智能大对象。在某些情况下，数据库服务器不允许释放锁，直到事务结束为止。（如果您未执行对 BEGIN WORK 事务块之内的智能大对象进行更新，则每次更新都是单独的事务。）如果隔离模式为 repeatable read，或如果保持的锁为排他锁，则可能发生此行为。

返回代码

0

函数成功。

<0

函数不成功，且返回代码指示失败的原因。

5.2.77 ifx_lo_col_info() 函数

ifx_lo_col_info() 函数设置 LO-specification 结构的字段为指定的数据库列的列级别存储特征。

语法

mint ifx_lo_col_info(*column_name*, *LO_spec*)

char *column_name;

ifx_lo_create_spec_t *LO_spec;

column_name

指向包含您想要使用其列级别存储特征的数据库列的名称的缓冲区的指针。

LO_spec

指向在其中存储 *column_name* 的列级别存储特征的 LO-specification 结构的指 针。

用法

ifx_lo_col_info() 函数将 *LO_spec* 指向的 LO-specification 结构的指端设置为 *column_name* 数据库列的存储特征。如果此指定了的列没有为其定义的列级别存储特征，则数据库服务器使用继承的存储特征。

column_name 缓冲区必须以下列格式指定列名称：

database@server_name:table.column

如果该列在符合 ANSI 的数据库中，则您还可包括 *owner_name*，如下：

database@server_name:owner.table.column

重要： 在您调用 ifx_lo_col_info() 之前，您必须调用 ifx_lo_def_create_spec() 函数。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.78 ifx_lo_copy_to_file() 函数

ifx_lo_copy_to_file() 函数将智能大对象的内容复制至操作系统文件内。

语法

mint ifx_lo_copy_to_file(*LO_ptr*, *fname*, *flags*, *result*)

ifx_lo_t *LO_ptr;

char *fname;

mint flags;

char *result;

LO_ptr

指向您提供来标识要复制的智能大对象的 LO-pointer 结构的指针。

fname

要保存该数据的目标文件的完全路径名称。

flags

指定 fname 文件的位置的整数。

result

指向包含 ifx_lo_copy_to_file() 生成的文件名称的缓冲区的指针。

用法

ifx_lo_copy_to_file() 函数可在服务器或客户机计算机上创建目标文件。*flags* 参数的标志值指示要复制的文件的位置。有效值包括下列 locator.h 头文件定义的常量。

File-location 常量用途**LO_CLIENT_FILE**

fname 文件在客户机计算机上。

LO_SERVER_FILE

fname 文件在服务器计算机上。

在缺省情况下，ifx_lo_copy_to_file() 函数生成的文件名称形如：

fname.hex_id

在此格式中，*fname* 是您指定作为 ifx_lo_copy_to_file() 的参数的文件名称，且 *hex_id* 为唯一的十六进制 smart-large-object 标识符。smart-large-object 标识符的位数最大值为 17；然而，大多数智能大对象的标识符位数较少。

例如，假定您指定 *pathname* 值为 '/tmp/resume'。

如果该 CLOB 列有标识符 **203b2**，则 ifx_lo_copy_to_file() 函数创建文件：/tmp/resume.203b2。

要更改此缺省的文件名称，您可在 *fname* 的文件名称部分指定下列通配符：

文件名称中的一个或多个相邻的问号（?）字符可生成唯一的文件名称。

ifx_lo_copy_to_file() 函数以来自 BLOB 或 CLOB 列的标识符的十六进制数字替代每一问号。例如，假定您指定 *pathname* 值为 '/tmp/resume??.txt'。

ifx_lo_copy_to_file() 函数将两位十六进制标识符放入该名称内。如果该 CLOB 列有标识符 **203b2**，则 ifx_lo_copy_to_file() 函数会创建文件 /tmp/resumeb2.txt。

如果您指定多于 17 个问号，则 ifx_lo_copy_to_file() 函数忽略它们。

在文件名称的末尾的感叹号（!）指示该文件名称不需要为唯一的。

例如，假定您指定路径名称值为 '/tmp/resume.txt!'。

ifx_lo_copy_to_file() 函数在该文件名称中未使用 smart-large-object 标识符，因此，它生成下列文件：ifx_lo_copy_to_file()

感叹号覆盖文件名称规范中的问号。

提示： 这些通配符在 ifx_lo_filename() 函数的 *fname* 参数中也有效。

您的应用程序必须确保有充足的空间来保存生成的文件。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.79 ifx_lo_copy_to_lo() 函数

ifx_lo_copy_to_lo() 函数将文件的内容复制至打开的智能大对象内。

语法

mint ifx_lo_copy_to_lo(*LO_fd, fname, flags*)

mint LO_fd;

```
char *fname;
mint flags;
```

LO_fd

要向其中写入文件内容的打开的智能大对象的 LO 文件描述符。

fname

包含要复制的数据的源文件的完全路径名称。

flags

指定 fname 文件的位置的整数。

用法

ifx_lo_copy_to_lo() 函数可复制服务器或客户机计算机上的源文件的内容。*flags* 参数的标志值指示要复制的文件的位置。locator.h 头文件定义的有效的值包括下列常量。

File-location 常量用途

LO_CLIENT_FILE

fname 文件在客户机计算机上。

LO_SERVER_FILE

fname 文件在服务器计算机上。

LO_APPEND

将 *fname* 中的数据追加到指定的智能大对象的末尾。可以前面的标志之一来屏蔽此标志。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.80 ifx_lo_create() 函数

ifx_lo_create() 函数创建新的智能大对象，并打开它用于在 GBase 8s ESQL/C 程序内的访问。

语法

mint ifx_lo_create(*LO_spec*, *flags*, *LO_ptr*, *error*)

ifx_lo_create_spec_t *LO_spec;

mint flags;

ifx_lo_t *LO_ptr;

mint *error;

LO_spec

指向包含新的智能大对象的存储特征的 LO-specification 结构的指针。

flags

指定打开新的智能大对象的模式的整数。

LO_ptr

指向新的智能大对象的 LO-pointer 结构的指针。

error

指向包含 ifx_lo_create() 设置的错误代码的整数的指针。

用法

ifx_lo_create() 函数执行下列步骤来创建新的智能大对象：

它创建 LO-pointer 结构，并将指向此结构的指针赋予 *LO_ptr* 参数。

它为来自 LO-specification 结构 *LO_spec* 的智能大对象指定存储特征。

如果 LO-specification 结构未包含存储特征（相关联的字段为空），则对于新的智能大对象，ifx_lo_create() 使用来自继承层级的存储特征。如果 *LO_spec* 指针为空，则 ifx_lo_create() 函数还使用系统指定的存储特征。

它以 *flags* 参数指定的访问模式打开新的智能大对象。

在 ifx_lo_create() 成功地完成之后，*flags* 参数的标志值指示该智能大对象的模式。有效值包括所有 access-mode 常量，如 表 1所示。它返回标识打开的智能大对象的 LO 文件描述符。

重要： 在您调用 ifx_lo_create() 函数之前，您必须调用 ifx_lo_def_create_spec() 函数来初始化 LO-specification 结构。

GBase 8s 使用调用 ifx_lo_create() 建立的缺省的参数，来确定后续的操作是否导致对应的智能大对象锁定和/或记录日志。

每一 ifx_lo_create() 调用都隐含地与当前连接相关联。当此连接关闭时，数据库服务器释放任何列都不引用的智能大对象（引用计数为零的那些）。

如果 ifx_lo_create() 函数成功，则它返回有效的 LO 文件描述符（*LO_fd*）。然后，您可使用 *LO_fd* 来标识在后续的函数调用中，哪个智能大对象来访问，诸如 ifx_lo_read() 和 ifx_lo_write()。然而，*LO_fd* 仅在当前的数据库连接内有效。

返回代码

有效的 LO 文件描述符

函数成功地创建并打开新的智能大对象。

-1

函数不成功；请检测该错误的详尽错误代码。

5.2.81 ifx_lo_def_create_spec() 函数

ifx_lo_def_create_spec() 函数分配并初始化 LO-specification 结构。

语法

```
mint ifx_lo_def_create_spec(LO_spec)
        ifx_lo_create_spec_t **LO_spec;
```

LO_spec

指向指向包含初始化的字段的新的 LO-specification 结构的指针的指针。

用法

ifx_lo_def_create_spec() 函数创建并初始化新的 LO-specification 结构 **ifx_lo_create_spec_t**。ifx_lo_def_create_spec() 函数以恰当的空值初始化新的 **ifx_lo_create_spec_t** 结构，并将它的地址放置在 *LO_spec* 指针中。在数据库服务器存储该大对象时，数据库服务器将空值翻译为应该用于存储特征的系统指定的缺省含义。

由于 ifx_lo_def_create_spec() 函数为 **ifx_lo_create_spec_t** 结构分配内存，因此，当您停止使用该结构时，您必须调用 ifx_lo_spec_free() 函数来释放内存。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.82 ifx_lo_filename() 函数

如果您将智能大对象复制至文件，则 ifx_lo_filename() 函数返回数据库服务器将

使用的路径名称。

语法

mint ifx_lo_filename(LO_ptr, fname, result, result_buf_nbytes)

ifx_lo_t *LO_ptr;

char *fname;

char *result;

mint result_buf_nbytes;

LO_ptr

指向标识要复制的智能大对象的 LO-pointer 结构的指针。

fname

要保存数据的目标文件的完全路径名称。

result

指向包含 ifx_lo_copy_to_file() 将生成的文件名称的缓冲区的指针。

result_len

以字节计的 *result* 字符缓冲区的大小。

用法

ifx_lo_filename() 生成来自您提供的 *fname* 参数的文件名称。请使用 ifx_lo_filename() 函数来确定 ifx_lo_filename() 函数将为它的 *fname* 参数创建的文件名称。

在缺省情况下，ifx_lo_copy_to_file() 函数生成的文件名称形如：

fname.hex_id

然而，您可在 *fname* 参数中指定通配符，这可更改此缺省的文件名称。您可使用 ifx_lo_filename() 的 *fname* 参数中的这些通配符来查看这些通配符生成了什么文件名称。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.83 ifx_lo_from_buffer() 函数

ifx_lo_from_buffer() 函数将指定字节数从用户定义的缓冲区复制至智能大对象内。

语法

```
mint ifx_lo_from_buffer(LO_ptr, size, buffer, error)
        ifx_lo_t *LO_ptr;
        mint size;
        char *buffer;
        mint *error;
```

LO_ptr

您想要将数据复制至其内的智能大对象的 LO-pointer 结构。

size

标识要复制至智能大对象的字节数的 **mint**。

buffer

指向您想要从其复制数据的用户定义的缓冲区的指针。

error

包含保存 ifx_lo_from_buffer() 设置的错误代码的 **mint** 的地址。

用法

ifx_lo_from_buffer() 复制多达 *size* 指定的大小的字节，从用户定义的缓冲复制至 *LO_ptr* 参数标识的智能大对象内。对智能大对象的写操作始于零字节偏移量处。要使用 ifx_lo_from_buffer() 函数，在您复制数据之前，该智能大对象必须在 sbspace 中存在。

返回代码

0

函数成功。

-1

函数不成功。

5.2.84 ifx_lo_lock() 函数

ifx_lo_lock() 函数允许您锁定智能大对象中显式范围的字节。

语法

```
mint ifx_lo_lock(LO_fd, offset, whence, range, lockmode)
        mint LO_fd;
        int8 *offset;
```

mint whence;

int8 *range;

mint lockmode;

LO_fd

要在其中锁定字节的范围的智能大对象的 LO 文件描述符。offset

指向指定偏移量的 8 字节整数（INT8）的指针，锁定始于智能大对象之内的该偏移量。

whence

指定从被计算的偏移量的哪个点的 **mint** 常量：智能大对象的开始、智能大对象内的当前位置或智能大对象的末尾。

range

指向指定要锁定的字节数的 8 字节整数（INT8）的指针。

lockmode

锁定指定的字节所采用的模式。对于排他锁，请设置为 LO_EXCLUSIVE_MODE，或对于共享锁，请设置为 LO_SHARED_MODE。

用法

对于 *LO_fd* 指定的智能大对象，ifx_lo_lock() 函数锁定由 *range* 指定的字节数，始于由 *offset* 和 *whence* 指定的位置。ifx_lo_lock() 函数放置 *lockmode* 指定的类型的锁。如果您指定 ISSLOCK，则 ifx_lo_lock() 在字节范围上放置共享锁。如果您指定 ISXLOCK，则 ifx_lo_lock() 在字节范围上放置排他锁。

在您调用 ifx_lo_lock() 之前，您必须取得有效的 LO 文件描述符，这需要通过调用 ifx_lo_create() 来创建新的智能大对象，或通过调用 ifx_lo_open() 来打开现有的智能大对象。

ifx_lo_lock() 函数使用 *whence* 和 *offset* 参数来确定搜寻位置，如下：

whence 值标识从哪里开始搜寻。

有效的值包括 locator.h 头文件定义的下列常量。

Whence 常量

启动搜寻位置

LO_SEEK_SET

智能大对象的启动

LO_SEEK_CUR

智能大对象中当前的搜寻位置

LO_SEEK_END

智能大对象的末尾

offset 参数标识以字节计的偏移量，从开始锁定的字节的（*whence* 参数指定的）起始搜寻位置。

除了锁定 *nbytes* 之外，您还可锁定从指定的偏移量至大对象的末尾的字节，在此，您可指定该大对象的当前末尾或最大末尾。您可使用两个整数常量（LO_CURRENT_END 和 LO_MAX_END）表示这些值。要使用这些值之一，首先将它转换为 **int8** 值，然后对于 *nbytes* 参数使用它。

返回代码

0

函数成功。

< 0

函数不成功。返回的值为 **sqlcode**，其为 GBase 8s 错误消息的数目。

5.2.85 ifx_lo_open() 函数

ifx_lo_open() 函数打开现有的智能大对象来访问。

语法

mint ifx_lo_open(*LO_ptr*, *flags*, *error*)

ifx_lo_t *LO_ptr;

mint flags;

mint *error;

LO_ptr

指向标识要打开的智能大对象的 LO-pointer 结构的指针。

flags

指定 *LO_ptr* 标识的智能大对象以何种方式打开的 **mint**。

error

指向包含 ifx_lo_open() 设置的错误代码的 **mint** 的指针。

用法

对于它需要访问的每一智能大对象的实例，您的 GBase 8s ESQL/C 程序必须调用 ifx_lo_open() 函数。

在 ifx_lo_open() 成功地完成之后，*flags* 参数的值指示智能大对象的模式。要了解 *flags* 参数的有效值的描述，请参阅 表 1。

GBase 8s 使用 ifx_lo_open()（或 ifx_lo_create()）建立的缺省参数，来确定后续的操作是否导致对于智能大对象的锁定或记录日志。

每一 ifx_lo_open() 调用都隐式地与当前连接相关联。当此连接关闭时，数据库服务器释放任何列都不引用的任何智能大对象（其引用计数为零的那些）。

如果 ifx_lo_open() 函数成功，则它返回有效的 LO 文件描述符（*LO_fd*）。然后，您可使用该文件描述符来标识在后续的函数调用中访问哪个智能大对象，诸如 ifx_lo_read() 和 ifx_lo_write()。*LO_fd* 仅在当前数据库连接之内有效。

在 ifx_lo_open() 已打开了智能大对象之后，它将放回的 LO 文件描述符中的搜寻位置设置为字节 0。如果为了锁定整个智能大对象而设置锁定的缺省范围，则 ifx_lo_open() 函数还可取得智能大对象上的锁，基于访问模式的下列设置：

对于 dirty-read 模式，数据库服务器不在智能大对象上放置锁。

对于 read-only 模式，数据库服务器在智能大对象上取得共享锁。

对于 write-only、write-append 或 read-write 模式，数据库服务器在智能大对象上取得更新锁。当发生对 ifx_lo_write() 或 ifx_lo_writewithseek() 函数的调用时，数据库服务器将该锁提升为排他锁。

当当前事务终止时，ifx_lo_open() 取得的锁丢失。然而，当需要锁的下一函数执行时，数据库服务器在此取得该锁。如果不需要此行为，请使用 BEGIN WORK 事务块，并在需要使用该锁的最后一条语句之后放置 COMMIT WORK 或 ROLLBACK WORK 语句。

返回代码

-1

函数不成功；请检测该错误的详尽错误代码。

有效的 LO 文件描述符

函数已成功地打开了智能大对象，并返回了有效的 LO 文件描述符。

5.2.86 ifx_lo_read() 函数

ifx_lo_read() 函数从打开的智能大对象读取指定数目的字节。

语法

mint ifx_lo_read(*LO_fd*, *buf*, *nbytes*, *error*)

```
mint LO_fd;
char *buf;
mint nbytes;
mint *error;
```

LO_fd

要从其读取的智能大对象的 LO 文件描述符。

buf

指向包含 ifx_lo_read() 从智能大对象读取的字符缓冲区的指针。

nbytes

以字节计的 buf 字符缓冲区的大小。此值不可超过 2 GB。

error

指向包含 ifx_lo_read() 设置的错误代码的 **mint** 的指针。

用法

ifx_lo_read() 函数从 *LO_fd* 文件描述符标识的智能大对象读取 *nbytes* 的数据。该读取始于 *LO_fd* 的当前搜寻位置。您可使用 ifx_lo_tell() 函数来取得当前的搜寻位置。

该函数将此数据读取至 *buf* 指向的用户定义的缓冲区内。*buf* 缓冲区的大小必须小于 2 GB。要读取大于 2 GB 的智能大对象，请在 2-GB chunk 中读取它们。

返回代码

>=0

该函数已从智能大对象读取至 *buf* 字符缓冲区内的字节数。

-1

函数不成功；请检测 *error* 的详尽错误代码。

5.2.87 ifx_lo_readwithseek() 函数

ifx_lo_readwithseek() 函数执行搜寻操作，然后从打开的智能大对象读取指定字节数的数据。

语法

mint ifx_lo_readwithseek(*LO_fd*, *buf*, *nbytes*, *offset*, *whence*, *error*)

```
char *buf;
mint nbytes;
ifx_int8_t *offset;
mint whence;
mint *error;
```

LO_fd

从其读取的智能大对象的 LO 文件描述符。

buf

指向包含 ifx_lo_readwithseek() 从智能大对象读取的数据的字符缓冲区的指针。

nbytes

以字节计的 buf 字符缓冲区的大小。此值不可超过 2 GB。

offset

指向从起始搜索位置的 8 字节整数（INT8）偏移量的指针。

whence

标识起始搜寻位置的 mint 值。

error

指向包含 ifx_lo_readwithseek() 设置的错误代码的 mint 的指针。

用法

ifx_lo_readwithseek() 函数从 *LO_fd* 文件描述符标识的打开的智能大对象读取 *nbytes* 的数据。

读取始于 *offset* 和 *whence* 参数指示的 *LO_fd* 的搜寻位置处，如下：

whence 参数标识从其开启搜寻的位置。

有效值包括 locator.h 头文件定义的下列常量。

Whence 常量

起始搜寻位置

LO_SEEK_SET

智能大对象的起始

LO_SEEK_CUR

智能大对象中的当前搜寻位置

LO_SEEK_END

智能大对象的末尾

offset 参数标识以字节计的偏移量，从（*whence* 参数指定的）起始搜寻位 置，至设置的搜寻位置。

该函数将此数据读取至 *buf* 指向的用户定义的缓冲区内。*buf* 缓冲区的大小必须小于 2 GB。要读取大于 2 GB 的智能大对象，请在 2-GB chunk 中读取它们。

返回代码

>=0

函数已从智能大对象读取至 *buf* 字符缓冲区内的字节数。

-1

函数不成功；请检测 *error* 的详尽错误代码。

5.2.88 ifx_lo_release() 函数

ifx_lo_release() 函数告诉数据库服务器释放与临时智能大对象相关联的资源。

语法

mint ifx_lo_release(*LO_ptr*)

ifx_lo_t *LO_ptr;

LO_ptr

您想要为其释放资源的智能大对象的 LO-pointer 结构。

用法

对于告诉数据库服务器何时释放与临时智能大对象相关联的资源为安全的，ifx_lo_release() 函数是有用的。*临时的*智能大对象是有一个或多个 LO 句柄的智能大对象之一，未将任何一个插入至表内。临时的智能大对象可以下列方式发生：

您以 ifx_lo_create() 创建智能大对象，但未将它的 LO 句柄插入至该数据库的列内。

您调用常见查询中的智能大对象的用户定义的例程，但从未将它的 LO 句柄赋予该数据库的列。

例如，下列查询为 **table1** 表中的每一行创建一个智能大对象，并将每一个发送至客户机应用程序：

SELECT filetoblob(...) FROM table1;

当该客户机应用程序结束处理每一这些智能大对象时，它可使用 ifx_lo_release() 函数来指示数据库服务器。在您调用在临时的智能大对象上的此函数之后，数据库服务器可在任何时刻释放该资源。对于 LO 句柄和任何相关联的 LO 文件描述符的应用，都不能保证奏效。

在非临时的智能大对象上使用此函数，不会导致任何不正确的行为。然而，该调用代价高，且不需要永久的智能大对象。

返回代码

0

函数成功。

< 0

函数不成功。

5.2.89 ifx_lo_seek() 函数

ifx_lo_seek() 函数为打开的智能大对象上的下一读或写操作设置文件位置。

语法

```
mint ifx_lo_seek(LO_fd, offset, whence, seek_pos)
        mint LO_fd;
        ifx_int8_t *offset;
        mint whence;
        ifx_int8_t *seek_pos;
```

LO_fd

您想要更改其搜寻位置的智能大对象的 LO 文件描述符。

offset

指向从起始搜寻位置的 8 字节整数偏移量的指针。

whence

标识起始的搜寻位置的 **mint** 值。

seek_pos

相对于该文件的起始，指向结果的 8 字节整数偏移量的指针，其对应于下一读/写操作的位置。

用法

该函数使用 *whence* 和 *offset* 参数来确定搜寻位置，如下：

whence 值标识从其起始搜寻的位置。

有效值包括 locator.h 头文件定义的下列常量。

Whence 常量起始的搜寻位置**LO_SEEK_SET**

智能大对象的起始

LO_SEEK_CUR

智能大对象中的当前搜寻位置

LO_SEEK_END

智能大对象的末尾

offset 参数标识以字节计的偏移量，从开始该搜寻位置处的（*whence* 参数指定的）起始的搜寻位置。

ifx_lo_tell() 函数为打开的智能大对象返回当前的搜寻位置。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.90 ifx_lo_spec_free() 函数

ifx_lo_spec_free() 函数释放 LO-specification 结构的资源。

语法

mint ifx_lo_spec_free(*LO_spec*)

ifx_lo_create_spec_t *LO_spec;

LO_spec

指向要释放的 LO-specification 结构的指针。

用法

ifx_lo_spec_free() 函数通过调用 ifx_lo_spec_free() 来释放分配了的 LO-specification 结构。*LO_spec* 指针指向要被释放的 **ifx_lo_create_spec_t** 结构。

GBase 8s ESQL/C 不执行 LO-specification 结构的内存管理。你必须为每一您以 ifx_lo_def_create_spec() 函数的调用来分配的 LO-specification 结构调用 ifx_lo_spec_free()。

重要： 请不要使用 ifx_lo_spec_free() 来释放您通过调用 ifx_lo_stat_cspec() 来访问的 **ifx_lo_create_spec_t** 结构。当您调用 ifx_lo_stat_free() 来释放 **ifx_lo_stat_t** 结构时，它还自动地释放 **ifx_lo_create_spec_t** 结构。请仅使用 ifx_lo_spec_free() 来释放您通过调用 ifx_lo_def_create_spec() 创建了的 **ifx_lo_create_spec_t** 结构。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.91 ifx_lo_specget_def_open_flags() 函数

ifx_lo_specget_def_open_flags() 函数从 LO-specification 结构取得智能大对象的缺省的打开标志。

语法

mint ifx_lo_specget_def_open_flags(*LO_spec*)

ifx_lo_create_spec_t *LO_spec;

LO_spec

指向从其取得缺省的打开标志的 LO-specification 结构的指针。

用法

可使用此函数从 LO-specification 结构取得缺省的打开标志。可以 ifx_lo_stat_cspec() 使用它来取得缺省的打开标志，当创建现有的智能大对象时，设置了该标志。

返回代码

>=0

函数成功。返回的整数存储缺省的打开标志的值。

-1

函数不成功。

5.2.92 ifx_lo_specget_estbytes() 函数

ifx_lo_specget_estbytes() 函数从 LO-specification 结构取得智能大对象的估计大小。

语法

mint ifx_lo_specget_estbytes(*LO_spec*, *estbytes*)

ifx_lo_create_spec_t *LO_spec;

ifx_int8_t *estbytes;

LO_spec

指向从其取得估计的大小的 LO-specification 结构的指针。

estbytes

指向 ifx_lo_specget_estbytes() 将智能大对象的估计字节数放置其内的 **ifx_int8_t** 结构的指针。

用法

estbytes 值是以字节计的智能大对象的估计的最终大小。此估计是智能大对象优化器的优化线索。

重要：在您调用 ifx_lo_specget_estbytes() 之前，您必须调用 ifx_lo_def_create_spec() 函数来初始化 LO-specification 结构。您可使用 ifx_lo_col_info() 函数来存储与 LO-specification 结构中的特定列相关联的存储特征。

返回代码

0

函数成功。

-1

函数不成功。

5.2.93 ifx_lo_specget_extsz() 函数

ifx_lo_specget_extsz() 函数从 LO-specification 结构取得智能大对象的分配 extent 大小。

语法

mint ifx_lo_specget_extsz(*LO_spec*)

ifx_lo_create_spec_t *LO_spec;

LO_spec

指向从其取得 extent 大小的 LO-specification 结构的指针。

用法

当数据库服务器写超出当前的 extent 的末尾时，extsz 值指定以字节计的为智能大对象分配的分配 extent 的大小。此值覆盖 GBase 8s 为 extent 的大小生成的估计。要获取关于分配 extent 的更多信息，请参阅 表 1。

重要： 在您调用 ifx_lo_specget_extsz() 之前，你必须调用 ifx_lo_def_create_spec() 函数来初始化 LO-specification 结构。您可使用 ifx_lo_col_info() 函数来存储与 LO-specification 结构中特定列相关联的存储特征。

返回代码

>=0

函数成功，且返回值指示 extent 大小。

-1

函数不成功。

5.2.94 ifx_lo_specget_flags() 函数

ifx_lo_specget_flags() 函数从 LO-specification 结构取得智能大对象的 create-time 标志。

语法

mint ifx_lo_specget_flags(*LO_spec*)

ifx_lo_create_spec_t *LO_spec;

LO_spec

指向从其取得该标志值的 LO-specification 结构的指针。

用法

create-time 标志提供下列关于智能大对象的信息：

是否在智能大对象上使用日志记录

是否存储对于智能大对象的最后访问的时间

将这两个指示符掩码在一起成为单个标志值。

重要： 在您调用 ifx_lo_specget_flags() 之前，您必须调用 ifx_lo_def_create_spec() 函数来初始化 LO-specification 结构。您可使用 ifx_lo_col_info() 函数来存储与 LO-specification 结构中特定列相关联的存储特征。

返回代码

>=0

函数成功，且返回值为 create-time 标志的值。

-1

函数不成功。

5.2.95 ifx_lo_specget_maxbytes() 函数

ifx_lo_specget_maxbytes() 函数从 LO-specification 结构取得智能大对象的最大大小。

语法

```
mint ifx_lo_specget_maxbytes(LO_spec, maxbytes)
        ifx_lo_create_spec_t *LO_spec;
        ifx_int8_t *maxbytes;
```

LO_spec

指向从其取得最大大小的 LO-specification 结构的指针。

maxbytes

指向 ifx_lo_specget_maxbytes() 将以字节计的智能大对象的最大大小放置其内的 **int8** 值的指针。如果此值为 -1，则该智能大对象没有大小限制。

用法

GBase 8s 不允许智能大对象的大小超过 *maxbytes* 值。

重要： 在您调用 ifx_lo_specget_maxbytes() 之前，您必须调用 ifx_lo_def_create_spec()

函数来初始化 LO-specification 结构。您可使用 ifx_lo_col_info() 函数来存储与 LO-specification 结构中特定列相关联的存储特征。

返回代码

0

函数成功。

-1

函数不成功。

5.2.96 ifx_lo_specget_sbspace() 函数

ifx_lo_specget_sbspace() 函数从 LO-specification 结构取得存储智能大对象处的 sbspace 的名称。

语法

mint ifx_lo_specget_sbspace(*LO_spec*, *sbspace_name*, *length*)

```
ifx_lo_create_spec_t *LO_spec;
char *sbspace_name;
mint length;
```

LO_spec

指向从其取得 sbspace 名称的 LO-specification 结构的指针。

sbspace_name

ifx_lo_specget_sbspace() 将智能大对象的 sbspace 的名称放置其内的字符缓冲区。

length

指定以字节计的 *sbspace_name* 缓冲区的大小的 mint 值。

用法

ifx_lo_specget_sbspace() 函数返回在其中存储智能大对象的 sbspace 的名称的当前设置。该函数将最多 *length*-1 字节复制至 *sbspace_name* 缓冲区内，并确保它是以空结尾的。

重要： 在您调用 ifx_lo_specget_sbspace() 之前，您必须调用 ifx_lo_def_create_spec() 函数来初始化 LO-specification 结构。您可使用 ifx_lo_col_info() 函数来存储与 LO-specification 结构中特定列相关联的存储特征。

sbspace 最长可为 18 字符。然而，您可能想要为 *sbspace_name* 缓冲区分配至少 129 字节来容纳未来 sbspace 名称长度的增长。

返回代码

0

函数成功。

-1

函数不成功。

5.2.97 ifx_lo_specset_def_open_flags() 函数

ifx_lo_specset_def_open_flags() 函数为智能大对象设置缺省的打开标志。

语法

mint ifx_lo_specset_def_open_flags(*LO_spec*, *flags*)

ifx_lo_create_spec_t *LO_spec;

mint flags;

LO_spec

指向在其中设置缺省的打开标志的 LO-specification 结构的指针。

flags

表示智能大对象的缺省的打开标志的 mint 值。

用法

此函数的最常用用法是，是指定总是通过使用未缓冲的 I/O 来打开智能大对象。此函数还可用于为智能大对象提供任何所需要的缺省的打开标志。在后来任何时刻打开该智能大对象时，都可使用所提供的标志，除非在打开时刻显式地重新了。

返回代码

0

函数成功。

-1

函数不成功。

示例

/* use unbuffered I/O on all opens for this LO */

```
ret = ifx_lo_specset_def_open_flags(lospec, LO_NOBUFFER);
```

5.2.98 ifx_lo_specset_estbytes() 函数

ifx_lo_specset_estbytes() 函数设置智能大对象的估计大小。

语法

mint ifx_lo_specset_estbytes(*LO_spec*, *estbytes*)

```
ifx_lo_create_spec_t *LO_spec;
ifx_int8_t *estbytes;
```

LO_spec

指向要在其中保存估计大小的 LO-specification 结构的指针。

estbytes

指向包含智能大对象的估计数值的 **ifx_int8_t** 结构的指针。

用法

estbytes 值是以字节计的智能大对象的估计的最终大小。此估计是 smart-large-object 优化器的优化线索。

当您创建新的智能大对象时，如果未指定 *estbytes* 值，则 GBase 8s 从存储特征的继承层级取得该值。

请不要更改此系统值，除非您知道该智能大对象的估计大小。如果您确需设置智能大对象的估计大小，则请不要指定比该智能大对象的最终大小高太多的值。否则，数据库服务器可能分配无用的存储。

返回代码

0

函数成功。

-1

函数不成功。

5.2.99 ifx_lo_specset_extsz() 函数

ifx_lo_specset_extsz() 函数为智能大对象设置分配 extent 大小。

语法

mint ifx_lo_specset_extsz(*LO_spec*, *extsz*)

ifx_lo_create_spec_t *LO_spec;

mint extsz;

LO_spec

指向将 extent 大小保存在其中的 LO-specification 结构的指针。

extsz

代表智能大对象的分配 extent 的大小的整数值。

用法

当数据库服务器写超出当前 extent 的末尾时，extsz 值指定要为智能大对象分配的分配 extent 的大小。对于 extent 要为多大，该值覆盖 GBase 8s 生成的估计。要获取关于分配 extent 的更多信息，请参阅 表 1。

当您创建智能大对象时，如果未指定 *extsz* 值，则 GBase 8s 尝试基于对该智能大对象的过去的操作，以及从存储特征的继承层级取得的其他存储特征（诸如最大字节），来优化该 extent 大小。

请不要更改此系统值，除非您知道该智能大对象的分配 extent 大小。仅遇到严重的存储分片的应用程序应总是设置分配 extent 大小。对于这样的应用程序，请确保您知道该智能大对象要扩展的确切字节数。

返回代码

0

函数成功。

-1

函数不成功。

5.2.100 ifx_lo_specset_flags() 函数

ifx_lo_specset_flags() 函数设置智能大对象的 create-time 标志。

语法

mint ifx_lo_specset_flags(*LO_spec*, *flags*)

ifx_lo_create_spec_t *LO_spec;

mint flags;

LO_spec

指向将标志值保存在其中的 LO-specification 结构的指针。

flags

表示智能大对象的 create-time 标志的整数值。

用法

create-time 标志提供关于智能大对象的下列信息：

是否对智能大对象使用日志记录

是否存储智能大对象的最后访问时间

将这两个指示符掩码在一起至单个标志值内。

当您创建新的智能大对象时，如果未指定 *flags* 值，则 GBase 8s 从存储特征的继承层级取得该值。

返回代码

0

函数成功。

-1

函数不成功。

5.2.101 ifx_lo_specset_maxbytes() 函数

ifx_lo_specset_maxbytes() 函数设置智能大对象的最大大小。

语法

mint ifx_lo_specset_maxbytes(*LO_spec*, *maxbytes*)

ifx_lo_create_spec_t *LO_spec;

ifx_int8_t *maxbytes;

LO_spec

指向将最大大小保存在其中的 LO-specification 结构的指针。

maxbytes

指向包含智能大对象的最大数目的 **ifx_int8_t** 结构的指针。如果此值为 -1，则该智能大对象没有大小限制。

用法

GBase 8s 不允许智能大对象的大小超过 *maxbytes* 值。数据库服务器不从存储特征的继承层级取得该值。要获取关于最大大小的更多信息，请参阅 表 1。

返回代码

0

函数成功。

-1

函数不成功。

5.2.102 ifx_lo_specset_sbspace() 函数

ifx_lo_specset_sbspace() 函数设置智能大对象的 sbspace 的名称。

语法

mint ifx_lo_specset_sbspace(*LO_spec*, *sbspace_name*)

ifx_lo_create_spec_t *LO_spec;

char *sbspace_name;

sbspace_name

指向包含在其中存储智能大对象的 sbspace 的名称的缓冲区的指针。

LO_spec

指向将 sbspace 名称保存在其中的 LO-specification 结构的指针。

用法

sbspace 的名称最长可为 18 字符。它还必须以空终止。

当您创建新的智能大对象时，如果未指定 *sbspace_name*，则 GBase 8s 从该列信息，或从 onconfig 文件的 SBSPACENAME 参数取得该 sbspace。

返回代码

0

函数成功。

-1

函数不成功。

5.2.103 ifx_lo_stat() 函数

ifx_lo_stat() 函数返回关于打开的智能大对象的状态的信息。

语法

mint ifx_lo_stat(*LO_fd, LO_stat*)

mint LO_fd;

ifx_lo_stat_t **LO_stat;

LO_fd

表示您想要取得其状态信息的打开的智能大对象的 LO 文件描述符。

LO_stat

指向指向 ifx_lo_stat() 分配并以状态信息完成的 LO-status 结构的指针的指针。

用法

ifx_lo_stat() 函数分配 LO-status 结构 **ifx_lo_stat_t**，并以 *LO_fd* 文件描述符标识的智能大对象的状态信息初始化。要访问该状态信息，对于 LO-status 结构，请使用 GBase 8s ESQL/C 访问器函数。

请使用 ifx_lo_stat_free() 函数来释放 LO-status 结构。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.104 ifx_lo_stat_atime() 函数

ifx_lo_stat_atime() 函数返回对智能大对象的最后访问时间。

语法

mint ifx_lo_stat_atime(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 ifx_lo_stat() 分配并以状态信息完成的 LO-status 结构的指针。

用法

如果智能大对象设置了 LO_KEEP_LASTACCESS_TIME 标志，则仅保证要维护最后访问的时间。如果您未设置此标志，则数据库服务器不将此访问时间值写到磁盘。ifx_lo_stat_atime() 函数返回的时间的分辨率为秒。

智能大对象的状态信息在 *LO_stat* 指向的 LO-status 结构中。ifx_lo_stat() 函数分配此结构，并以特定的智能大对象的状态信息填充它。因此，您必须先以对 ifx_lo_stat() 的调用来调用 ifx_lo_stat_atime()。

返回代码

>=0

LO_stat 标识的智能大对象的 last-access 时间。

-1

函数不成功。

5.2.105 ifx_lo_stat_cspec() 函数

ifx_lo_stat_cspec() 函数返回智能大对象的 LO-specification 结构。

语法

ifx_lo_create_spec_t *ifx_lo_stat_cspec(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 ifx_lo_stat() 分配并以状态信息完成的 LO-status 结构的指针。

用法

ifx_lo_stat_cspec() 函数返回指向 LO-specification 结构的指针 **ifx_lo_create_spec_t**，其包含指定的智能大对象的存储特征。您可使用此 LO-specification 结构来创建带有相同存储特征的另一智能大对象，或通过访问器（**ifx_specget_**）函数来访问存储特征。

您必须先以 ifx_lo_stat() 的调用来调用 ifx_lo_stat_cspec()。ifx_lo_stat() 函数为 **ifx_lo_create_spec_t** 结构分配内存，伴随着 **ifx_lo_stat_t** 结构，并以您指定了的智能大对象的状态信息来初始化它。当您调用 ifx_lo_stat_free() 函数来释放 **ifx_lo_stat_t** 结构时，它自动地释放 **ifx_lo_create_spec_t** 结构。

返回代码

指向 LO-specification 结构（ifx_lo_create_spec_t）的有效指针。

函数成功。

NULL

函数不成功。

5.2.106 ifx_lo_stat_ctime() 函数

ifx_lo_stat_ctime() 函数返回智能大对象最后状态更改的时间。

语法

mint ifx_lo_stat_ctime(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 ifx_lo_stat() 分配并以状态信息完成的 LO-status 结构的指针。

用法

状态的最后更改包括存储特征的修改，包括对智能大对象的引用和写的数目的更改。ifx_lo_stat_ctime() 函数返回的时间的分辨率为秒。

智能大对象的状态信息在 *LO_stat* 指向的 LO-status 结构中。ifx_lo_stat() 函数分配此结构并以特定的智能大对象的状态信息填充它。因此，您必须先以对 ifx_lo_stat() 的调

用来调用 ifx_lo_stat_ctime()。

返回代码

>=0

LO_stat 标识的智能大对象的 last-change 时间。

-1

函数成功。

5.2.107 ifx_lo_stat_free() 函数

ifx_lo_stat_free() 函数释放 LO-status 结构。

语法

mint ifx_lo_stat_free(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 ifx_lo_stat() 函数已分配了的 LO-status 结构的指针。

用法

ifx_lo_stat() 函数在 LO-status 结构中返回关于打开的智能大对象的状态信息。当您的应用程序不再需要此状态信息时，请使用 ifx_lo_stat_free() 函数来释放该 LO-status 结构。

返回代码

0

函数成功。

-1

函数不成功。

5.2.108 ifx_lo_stat_mtime_sec() 函数

ifx_lo_stat_mtime_sec() 函数返回智能大对象的最后修改的时间。

语法

mint ifx_lo_stat_mtime_sec(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 ifx_lo_stat() 分配并以状态信息完成的 LO-status 结构的指针。

用法

ifx_lo_stat_mtime_sec() 函数返回的时间的分辨率为秒。

智能大对象的状态信息在 *LO_stat* 指向的 LO-status 结构中。ifx_lo_stat() 函数分配此结构，并以特定的智能大对象的状态信息完成它。因此，您必须先以对 ifx_lo_stat() 的调用来调用 ifx_lo_stat_mtime_sec()。

返回代码

>=0

LO_stat 标识的智能大对象的最后修改时间。

-1

函数成功。

5.2.109 ifx_lo_stat_refcnt() 函数

ifx_lo_stat_refcnt() 函数返回对智能大对象的引用的数目。

语法

mint ifx_lo_stat_refcnt(*LO_stat*)

ifx_lo_stat_t *LO_stat;

LO_stat

指向 **ifx_lo_stat()** 分配并以状态信息填写的 LO-status 结构的指针。

用法

refcnt 参数时智能大对象的引用计数。对于智能大对象，此计数指示当前存在的永久地存储的 LO-pointer 结构（**ifx_lo_t**）的数目。数据库服务器假定它可安全地移除该智能大对象，并当引用计数为零且下列条件存在时，重新使用分配给它的任何资源：

引用计数在其中递减的事务提交。

连接终止，且在此连接期间创建该智能大对象，但它的引用计数不是递增的。

当数据库服务器为一行中的智能大对象存储 LO-pointer 结构时，它递增引用计数器。

智能大对象的状态信息在 *LO_stat* 指向的 LO-status 结构中。ifx_lo_stat() 函数分配此结构，并以特定的智能大对象的状态信息填充它。因此，您必须以对 ifx_lo_stat() 的调用来调用 ifx_lo_stat_refcnt()。

返回代码

>=0

LO_stat 标识的智能大对象的引用计数。

-1

函数不成功。

5.2.110 ifx_lo_stat_size() 函数

ifx_lo_stat_size() 函数返回以字节计的智能大对象的大小。

语法

```
mint ifx_lo_stat_size(LO_stat, size)
        ifx_lo_stat_t *LO_stat;
        ifx_int8_t *size;
```

LO_stat

指向 ifx_lo_stat() 分配并以状态信息填写的 LO-status 结构的指针。

size

指向 ifx_lo_stat_size() 以其大小字节填充智能大对象的 **ifx_int8_t** 结构的指针

用法

智能大对象的状态信息在 *LO_stat* 指向的 LO-status 结构中。ifx_lo_stat() 函数分配此结构，并以特定的智能大对象的状态信息填充它。因此，您必须先以对 ifx_lo_stat() 的调用来调用 ifx_lo_stat_size()。

返回代码

0

函数成功。

-1

函数不成功。

5.2.111 ifx_lo_tell() 函数

ifx_lo_tell() 函数返回打开的智能大对象的当前文件或搜寻位置。

语法

mint ifx_lo_tell(*LO_fd*, *seek_pos*)

mint LO_fd;

ifx_int8_t *seek_pos;

LO_fd

表示您想要确定其搜寻位置的智能大对象的 LO 文件描述符。

seek_pos

指向标识当前搜寻位置的 8 字节整数的指针。

用法

该查询位置是智能大对象上下一读或写操作的偏移量，该智能大对象与 LO 文件描述符 *LO_fd* 相关联。**ifx_lo_tell()** 函数在用户定义的 **int8** 变量 *seek_pos* 中返回此搜寻位置。

返回代码

0

返回成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.112 ifx_lo_to_buffer() 函数

ifx_lo_to_buffer() 函数将指定的字节数从智能大对象复制至用户定义的缓冲区内。

语法

mint ifx_lo_to_buffer(*LO_ptr*, *size*, *buf_ptr*)

ifx_lo_t *LO_ptr;

```
mint size;
char **buf_ptr;

mint error;
```

LO_ptr

您想要从其复制数据的智能大对象的 LO-pointer 结构。

size

标识要从智能大对象复制的字节数的 **mint**。

buf_ptr

指向您想要将数据复制到的用户定义的缓冲区的加倍间接指针。

error

包含保存 ifx_lo_to_buffer() 设置的错误代码的 **mint** 的地址。

用法

ifx_lo_to_buffer() 函数从 *LO_ptr* 参数标识的智能大对象复制字节，最多为 *size* 参数指定的大小。来自智能大对象的读操作起始于零字节偏移量。如果该智能大对象小于 *size* 值，则 ifx_lo_to_buffer() 仅复制智能大对象中的字节数。如果该智能大对象包含多于 *size* 的字节，则 ifx_lo_to_buffer() 函数最多仅复制 *size* 字节至用户定义的缓冲区内。

当 **buf_ptr** 为 NULL 时，ifx_lo_to_buffer() 为用户定义的缓冲区分配内存。否则，该函数假定您已分配了 *buf_ptr* 标识的内存。

返回代码

0

buf_ptr 标识的从智能大对象复制到用户定义的缓冲区的字节数。

-1

函数不成功。

5.2.113 ifx_lo_truncate() 函数

ifx_lo_truncate() 函数在指定的字节处截断智能大对象。

语法

mint ifx_lo_truncate(*LO_fd*, *offset*)

mint LO_fd;

ifx_int8_t *offset;

LO_fd

表示您想要截断其值的打开的智能大对象的 LO 文件描述符。

offset

指向标识从其开始截断智能大对象处的偏移量的 8 字节整数的指针。

用法

ifx_lo_truncate() 函数将智能大对象的最后有效字节设置为指定的 *offset* 值。如果此 *offset* 值超出当前的智能大对象的末尾，则您实际地扩展该智能大对象。如果此 *offset* 值小于该智能大对象的当前末尾，则数据库服务器取回所有存储，从 *offset* 指示的位置至该智能大对象的末尾。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示失败的原因。

5.2.114 ifx_lo_unlock() 函数

ifx_lo_unlock() 函数允许您解锁由 ifx_lo_lock() 锁定了的智能大对象中的一些字节。

语法

mint ifx_lo_unlock(lofd, offset, whence, range)

mint lofd;

int8 *offset;

mint whence;

int8 *range;

LO_fd

表示要在其中解锁一些字节的智能大对象的 LO 文件描述符。

offset

指向在智能大对象之内指定开始解锁处的偏移量的 8 字节整数（INT8）的指针。

whence

指定从哪一点计算偏移量的整数常量：智能大对象的开头、智能大对象之内的当前位置，或智能大对象的末尾。

range

指向指定要解锁的字节数的 8 位整数（INT8）的指针。

用法

对于由 *LO_fd* 指定的智能大对象，ifx_lo_unlock() 函数解锁由 *nbytes* 指定的字节数，从由 *offset* 和 *whence* 指定的偏移量处开始。在调用 ifx_lo_unlock() 之前，您必须通过调用 ifx_lo_create() 来创建新的智能大对象，或通过调用 ifx_lo_open() 来打开现有的智能大对象，来取得有效的 LO 文件描述符。

返回代码

0

函数成功。

< 0

函数不成功。返回值为 **sqlcode**，其为 GBase 8s 错误消息的数目。

5.2.115 ifx_lo_write() 函数

ifx_lo_write() 函数将指定的字节数写到打开的智能大对象。

语法

```
mint ifx_lo_write(LO_fd, buf, nbytes, error)
        mint LO_fd;
        char *buf;
        mint nbytes;
        mint *error;
```

LO_fd

表示要写到其的智能大对象的 LO 文件描述符。

buf

指向包含该函数写到智能大对象的数据的缓冲区的指针。

nbytes

写到智能大对象的字节数。此值的最小长度为 0，且必须小于 2 GB。

error

指向包含 ifx_lo_write() 设置的错误代码的 **mint** 的指针。

用法

ifx_lo_write() 函数将 *nbytes* 的数据写到 *LO_fd* 文件描述符标识的智能大对象。在 *LO_fd* 的当前搜寻位置开始写。您可使用 ifx_lo_tell() 函数来取得当前的搜寻位置。

该函数从 *buf* 指向的用户定义的缓冲区取得数据。*buf* 缓冲区的大小必须小于 2 GB。

如果数据库服务器将少于 *nbytes* 的数据写到智能大对象，则 ifx_lo_write() 函数返回它写了的字节数，并设置 *error* 值来指向指示未完成的写操作的原因的值。当 sbspace 用尽空间时，可发生此情况。

返回代码

>=0

函数已从 *buf* 字符缓冲区写到打开的智能大对象的字节数。

-1

函数不成功；请检测 *error* 的详尽错误代码。

5.2.116 ifx_lo_writewithseek() 函数

ifx_lo_writewithseek() 函数执行搜寻操作，然后将指定字节数的数据写到打开的智能大对象。

语法

mint ifx_lo_writewithseek(*LO_fd, buf, nbytes, offset, whence, error*)

```
mint LO_fd;
char *buf;
mint nbytes;
```

```
ifx_int8_t *offset;
mint whence;
mint *error;
```

LO_fd

表示要写到其的智能大对象的 LO 文件描述符。

buf

指向包含函数写到智能大对象的数据的缓冲区的指针。

nbytes

要写到智能大对象的字节数。此值不可超过 2 GB。

offset

指向从其起始的搜寻位置的 8 字节整数（INT8）偏移量的指针。

whence

标识起始的搜寻位置的 **mint** 值。

error

指向包含 ifx_lo_writewithseek() 设置的错误代码的 **mint** 的指针。

用法

ifx_lo_writewithseek() 函数将 *nbytes* 的数据写到 LO*_fd* 文件描述符标识的智能大对象。该函数从 *buf* 指向的用户定义的缓冲区取得要写的数据。该缓冲区的大小必须小于 2 GB。

在 *offset* 和 *whence* 参数指示的 *LO_fd* 的搜寻位置开始写，如下：

whence 参数标识从其开始搜寻的位置。

有效的值包括下列 locator.h 头文件定义的常量。

Whence 常量

起始的搜寻位置。

LO_SEEK_SET

智能大对象的起始

LO_SEEK_CUR

智能大对象中的当前搜寻位置

LO_SEEK_END

智能大对象的末尾

offset 参数标识以字节计的偏移量，从（*whence* 参数指定的）起始的搜寻位置，至应设置的搜寻位置。

如果数据库服务器将少于 *nbytes* 的数据写到智能大对象，则 ifx_lo_writewithseek() 函数返回它写了的字节数，并设置 *error* 值来指向表示未完成的写操作的原因的值。当 sbspace 用尽空间时，可发生此情况。

返回代码

>=0

该函数已从 *buf* 字符缓冲区写到智能大对象的字节数。

-1

函数不成功；要获取详尽的错误代码，请检测 *error*。

5.2.117 ifx_lvar_alloc() 函数

ifx_lvar_alloc() 函数指定当访存 lvarchar 数据时，是否分配内存。

语法

mint ifx_lvar_alloc(mint*alloc*)

mint alloc;

alloc

分配标志的值；为 **1** 或 **0**

用法

当设置标志为 1 时，ESQL/C 自动地执行此内存分配。当您不确定 SELECT 语句返回的数据量时，在 SELECT 语句前，您可使用标志值 1。当设置标志为 0 时，ESQL/C 不自动地执行此内存分配。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.118 ifx_putenv() 函数

ifx_putenv() 函数更改现有环境变量的值，创建环境变量，或从运行时刻环境移除变量。

语法

int ifx_putenv(*envstring*);

const char *envstring;

nvstring

指向形如 *varname=string* 的字符串的指针，其中 *varname* 是要添加或修改的环境变量的名称，*string* 是变量值。

用法

ifx_putenv() 函数在 **InetLogin** 结构中添加新的环境变量，或修改现有环境变量的值。这些变量定义进程在其中执行的环境。如果 *varname* 已是环境的一部分，则 ifx_putenv() 以 *string* 替代现有的值；否则，ifx_putenv() 以值 *string* 将 *varname* 添加至环境。

要从运行时刻环境移除变量，则请将 *varname* 指定为它的缺省值。如果缺省值为 NULL，则以 ifx_putenv() 将变量设置为空字符串，可有效地移除它。如果该变量的缺省值不为 NULL，则以 ifx_putenv() 将该变量设置为空，将变量重置为它的缺省值，但未将它从运行时刻环境移除。

ifx_putenv() 函数先设置 GBase 8s 变量，然后是其他变量。

下列对 ifx_putenv() 函数的调用更改 GBASEDBTDIR 环境变量的值：

ifx_putenv("gbasedbtdir=c:\gbasedbt");

此函数仅影响当前进程的环境变量。命令处理器的环境不更改。

返回代码

0

调用 ifx_putenv() 成功。

-1

调用 ifx_putenv() 不成功。

5.2.119 ifx_strdate() 函数

ifx_strdate() 函数将字符串转换为内部的 DATE。

语法

mint ifx_strdate(*str*, *jdate*, *dbcentury*)

```
char *str;
int4 *jdate;
char dbcentury;
```

str

指向包含要转换的日期的字符串的指针。

jdate

指向接收 *str* 字符串的内部 DATE 的 **int4** 整数的指针。

dbcentury

可为下列字符之一，其确定将哪个世纪应用于该日期的年份部分：

R

当前的。该函数使用当前年份的两个高位数字来扩展年份值。

P

过去的。该函数使用过去的和当前的世纪来扩展年份值。它将这两个日期与当前日期相对比，并使用在当前世纪之前的那个世纪。如果两个日期都在当前日期之前，则该函数使用最接近于当前日期的世纪。

F

将来的。该函数使用当前的和下一世纪来扩展年份值。它将这些世纪与当前的日期相对比，并使用晚于当前日期的世纪。如果两个日期都晚于当前的日期，则该函数使用最接近于当前日期的日期。

C

最近的。该函数使用过去的、当前的和下一世纪来扩展年份值。它选择最接近于当前日期的世纪。

用法

对于缺省的语言环境 US English，ifx_strdate() 函数以下列优先顺序确定如何格式化该字符串：

DBDATE 环境变量指定的格式（如果设置 DBDATE 的话）。

GL_DATE 环境变量指定的格式（如果设置 GL_DATE 的话）。

缺省的日期形式：mm/dd/yyyy。您可使用任何非数值的字符作为月份、日期与年份之间的分隔符。您可将年份表达为四位（2007）或为两位（07）。

当您使用非缺省的语言环境，且未设置 DBDATE 或 GL_DATE 环境变量时，ifx_strdate() 使用客户机语言环境定义的日期终端用户格式。

当您在日期字符串中使用两位年份时，ifx_strdate() 函数使用 *dbcentury* 参数的值来确定使用哪个世纪。如果您未设置 *dbcentury* 参数，则 ifx_strdate() 使用 DBCENTURY 环境变量来确定使用哪个世纪。如果您未设置 DBCENTURY，则 ifx_strdate() 假定两位年份的当前世纪。

返回代码

0

转换成功。

< 0

转换失败。

-1204

str 参数指定无效的年份。

-1205

str 参数指定无效的月份。

-1206

str 参数指定无效的日期。

-1212

数据转换格式必须包含月份、日期或年份组件。DBDATE 指定数据转换格式。

-1218

由 *str* 参数指定的日期不能恰当地表示日期。

5.2.120 ifx_var_alloc() 函数

ifx_var_alloc() 函数为 lvarchar 或 var binary 主变量的数据缓冲区分配内存。

语法

var binary

mint ifx_var_alloc(*var_bin*, *var_size*)

```
        var binary **var_bin
        int4 var_size;
```

```
lvarchar
mint ifx_var_alloc(lvar, var_size)
        lvarchar **lvar
        int4 var_size;
```

var_bin

分配其数据缓冲区的 **var binary** 指针主变量的地址。

lvar

分配其数据缓冲区的 **lvarchar pointer** 主变量的地址。

var_size

要分配的以字节计的数据缓冲区的大小。

用法

ifx_var_flag() 函数的分配标志通知用于该数据缓冲区的分配方式的 GBase 8s ESQL/C。如果您在 ifx_var_flag() 中设置分配标志为 0，则您必须以 ifx_var_alloc() 函数为 **var binary** 主变量的数据缓冲区显式地分配内存。

重要： 或者您分配内存，或者允许 GBase 8s ESQL/C 为您分配内存，您必须通过使用 ifx_var_dealloc() 函数来释放分配的内存。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.121 ifx_var_dealloc() 函数

ifx_var_dealloc() 函数释放为 var binary 主变量的数据缓冲区分配了的内存。

语法

var binary

mint ifx_var_dealloc(*var_bin*)

var binary **var_bin;

lvarchar

mint ifx_var_dealloc(*lvar*)

lvarchar **lvar;

var_bin

释放其数据缓冲区的 **var binary** 指针主变量的地址。

lvar

释放其数据缓冲区的 **lvarchar** 指针主变量的地址。

用法

ifx_var_flag() 函数的分配标志告诉 GBase 8s ESQL/C 用于该数据缓冲区的是哪种分配方式。不管是 GBase 8s ESQL/C（设置分配标志为 1），还是您的应用程序（设置分配标志为 0）分配该内存，您必须显式地释放分配给 **lvarchar** 或 **var binary** 主变量的数据缓冲区的内存。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.122 ifx_var_flag() 函数

ifx_var_flag() 函数确定如何为 lvarchar 或 var binary 主变量的数据缓冲区分配内存。

语法

var binary

mint ifx_var_flag(*var_bin, flag*)

var binary **var_bin;

int2 flag;

lvarchar

mint ifx_var_flag(*lvar, flag*)

lvarchar **lvar;

int2 flag;

flag

分配标志的 **int2** 值，或为 0，或为 1。

var_bin

var binary 主变量的地址。

lvar

lvarchar pointer 主变量的地址。

用法

flag 参数的值是分配标志。它确定由谁来为 *var_bin* 主变量的数据处理内存分配，如下：

当 *flag* 为 1 时，GBase 8s ESQL/C 自动地执行此内存分配。

当您不确定 SELECT 返回的数据量时，在 SELECT 语句之前，您可使用 *flag* 值 1。

当 *flag* 为 0 时，GBase 8s ESQL/C 不自动地执行此内存分配。

当您设置标志为 0 时，您必须以 ifx_var_alloc() 函数为 *lvar* 或 *var_bin* 变量的数据缓冲区分配内存。

如果您不为 **lvarchar** 或 **var binary** 主变量调用 ifx_var_flag() 函数，则 GBase 8s ESQL/C 为它的数据缓冲区分配内存。或者您为 **lvarchar** 或 **var binary** 变量分配内存，或者允许 GBase 8s ESQL/C 为您分配，您必须以 ifx_var_dealloc() 函数释放该内存。

返回代码

0

函数成功。

<0

函数*不*成功，且返回值指示错误的原因。

5.2.123 ifx_var_freevar() 函数

ifx_var_freevar() 函数释放已为 var binary 和 lvarchar pointer 主变量分配了的内存。

语法

int fix_var_freevar(*var_bin*)

var binary *var_bin;

var_bin

var binary 或 lvarchar pointer 主变量的地址。

用法

每当您有 **var binary** 或 **lvarchar pointer** 主变量时，如下列示例所示，您必须通过使用 ifx_var_freevar() 函数来显式地释放为它分配的内存。

EXEC SQL var binary ‘polygon’ poly;

EXEC SQL lvarchar *c;

下列示例说明 ifx_var_freevar() 的使用。您必须通过使用 ifx_var_freevar() 函数来显式地释放已为 **var binary** 和 **lvarchar pointer** 主变量分配了的内存。

ifx_var_freevar(&poly);

ifx_var_freevar(&c);

如果您未使用 ifx_var_dealloc() 来释放已为 **var binary** 主变量的数据缓冲区分配了的内存，则 ifx_var_freevar() 会这么做。然后，它释放 **var binary** 和 **lvarchar pointer** 主变量的内存。在前面的示例中，在调用了 ifx_var_freevar() 之后，会将 *poly* 和 *c* 设置为空。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.124 ifx_var_getdata() 函数

ifx_var_getdata() 函数返回来自 lvarchar 或 var binary 主变量的数据。

语法

var binary

void *ifx_var_getdata(*var_bin*)

var binary **var_bin;

lvarchar

void *ifx_var_getdata(*lvar*)

lvarchar **lvar;

var_bin

检索其数据的 **var binary** 主变量的地址。

lvar

检索其数据的 lvarchar pointer 主变量的地址。

用法

ifx_var_getdata() 函数返回数据作为 **void *** 指针。您的 GBase 8s ESQL/C 应用程序必须将此指针强制转型为正确的数据类型。当您对 **lvarchar pointer** 使用 ifx_var_getdata() 时，您必须将返回的（void）指针强制转型为 C 语言 **character pointer**（char *）。

返回代码

空指针

函数不成功。

指向数据缓冲区的有效指针

函数成功。

5.2.125 ifx_var_getlen() 函数

ifx_var_getlen() 函数返回 lvarchar pointer 或 var binary 主变量中数据的长度。

语法

var binary

mint ifx_var_getlen(*var_bin*)

var binary **var_bin;

lvarchar

mint ifx_var_getlen(*lvar*)

lvarchar **lvar;

var_bin

返回其长度的 var binary 主变量的地址。

lvar

返回其长度的 lvarchar pointer 主变量的地址。

用法

ifx_var_getlen() 函数返回的长度是已为 *lvar* 或 *var_bin* 主变量的数据缓冲区分配了的字节数。

如果您通过使用 GET DESCRIPTOR 语句的 DATA 子句从描述符区域取得 **lvarchar pointer** 或 **var binary**，则该值以空终止。如果对这样一个变量使用 ifx_var_getlen()，则返回的长度包括空终止符。要取得正确的长度，请使用 GET DESCRIPTOR 语句的 LENGTH 子句。

返回代码

>=0

var_bin 主变量的数据缓冲区的长度。

<0

函数不成功。

5.2.126 ifx_var_isnull() 函数

ifx_var_isnull() 函数检查 lvarchar 或 var binary 主变量是否包含空值。

语法

var binary

mint ifx_var_isnull(*var_bin*)

var binary **var_bin;

lvarchar

mint ifx_var_isnull(*lvar*)

lvarchar **lvar;

var_bin

var binary 主变量的地址。

lvar

lvarchar pointer 主变量的地址。

用法

ifx_var_isnull() 函数检查 **lvarchar** 或 **var binary** 主变量是否为空值。要确定任何其他数据类型的 GBase 8s ESQL/C 主变量是否包含空，请使用 risnull() 库函数。

返回代码

0

opaque 类型数据不是空值。

1

opaque 类型数据是空值。

5.2.127 ifx_var_setdata() 函数

ifx_var_setdata() 函数在 lvarchar 或 var binary 主变量中存储数据。

语法

var binary

mint ifx_var_setdata(*var_bin*, *buffer*, *buf_len*)

var binary **var_bin;

char *buffer;

int4 buf_len;

lvarchar

mint ifx_var_setdata(*lvar*, *buffer*, *buf_len*)

lvarchar **lvar;

char *buffer;

int4 buf_len;

buffer

包含要存储在 *lvar* 或 *var_bin* 主变量中的数据的字符缓冲区。

buf_len

以字节计的缓冲区的长度。

var_bin

var binary 主变量的地址。

lvar

lvarchar pointer 主变量的地址。

用法

ifx_var_setdata() 函数将数据存储在 *lvar* 或 *var_bin* 主变量的数据缓冲区中的 *buffer* 中。对于 **lvarcharpointer** 主变量，GBase 8s ESQL/C 期望 *buffer* 之中的数据为以空终止的 ASCII 数据。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.128 ifx_var_setlen() 函数

ifx_var_setlen() 函数存储 lvarchar 或 var binary 主变量的数据缓冲区的长度。

语法

var binary

mint ifx_var_setlen(*var_bin*, *length*)

var binary **var_bin;

int4 length

lvarchar

mint ifx_var_setlen(*lvar*, *length*)

lvarchar **lvar;

int4 length

length

以字节计的要为 **var binary** 数据分配的数据缓冲区的长度。

var_bin

var binary 主变量的地址。

lvar

lvarchar pointer 主变量的地址。

用法

ifx_var_setlen() 函数设置的 *length* 是为 *lvar* 或 *var_bin* 主变量的数据缓冲区分配的字节数。调用此函数来更改 ifx_var_alloc() 函数为 *lvar* 或 *var_bin* 主变量分配的数据缓冲区的大小。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.129 ifx_var_setnull() 函数

ifx_var_setnull() 函数将 lvarchar 或 var binary 主变量设置为空值。

语法

var binary

mint ifx_var_setnull(*var_bin*, *flag*)

var binary **var_bin;

mint flag

lvarchar

mint ifx_var_setnull(*var_bin*, *flag*)

var binary **var_bin;

mint flag;

var_bin

var binary 主变量的地址。

lvar

lvarcharpointer 主变量的地址。

flag

值 0 指示为非空值，值 1 指示为空值。

用法

ifx_var_setnull() 函数将类型 **lvarchar** 或 **var binary** 的主变量设置为空值。要将任何其他数据类型的 GBase 8s ESQL/C 主变量设置为空，请使用 rsetnull() 库函数。

返回代码

0

函数成功。

<0

函数不成功，且返回值指示错误的原因。

5.2.130 incvasc() 函数

incvasc() 函数将 INTERVAL 值的符合 ANSI SQL 标准的字符串转换为 interval 值。

语法

mint incvasc(*inbuf*, *invvalue*)

char *inbuf;

intrvl_t *invvalue;

inbuf

指向包含 ANSI 标准 INTERVAL 字符串的缓冲区的指针。

invvalue

指向初始化了的 **interval** 变量的指针。

用法

您必须以您想要此变量拥有的限定符来初始化 *invvalue* 中的 **interval** 变量。

inbuf 中的字符串可有开头的和结尾的空格。然而，从第一个有效数字到最后一个，*inbuf* 仅可包含字符，它们是适合于 **interval** 变量的限定符字段的数字和定界符。

如果该字符串为空字符串，则 incvasc() 函数将 *invvalue* 中的值设置为空。如果该字符串是可接受的，则该函数设置 **interval** 变量中的值，并返回零。否则，该函数将 **interval** 值中的值设置为空。

返回代码

0

转换成功。

-1260

不可能在指定的类型之间转换。

-1261

datetime 或 **interval** 的第一个字段中的数字太多。

-1262

datetimeinterval 中的非数值字符。

-1263

datetime 或 **interval** 值中的字段出界或不正确。

-1264

在 **datetime** 或 **interval** 值的末尾有额外的字符。

-1265

在 datetime 或 interval 操作上发生溢出。

-1266

datetime 或 interval 值与该操作不兼容。

-1267

datetime 计算的结果出界。

-1268

参数包含无效的 **datetime** 或 **interval** 限定符。

示例

demo 目录在文件 incvasc.ec 中包含此样例程序。

```
/*
        * incvasc.ec *
```

The following program converts ASCII strings into interval (intvl_t) structure. It also illustrates error conditions involving invalid qualifiers for interval values.

```
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        mint x;

        EXEC SQL BEGIN DECLARE SECTION;
        interval day to second in1;
        EXEC SQL END DECLARE SECTION;

        printf("INCVASC Sample ESQL Program running.\n\n");

        printf("Interval string #1 = 20 3:10:35\n");
        if(x = incvasc("20 3:10:35", &in1))
    printf("Result = failed with conversion error:%d\n",x);
        else
        printf("Result = successful conversion\n");

        /*
 * Note that the following literal string has a 26 in the hours field
        */
        printf("\nInterval string #2 = 20 26:10:35\n");
        if(x = incvasc("20 26:10:35", &in1))
```

```
        printf("Result = failed with conversion error:%d\n",x);
            else
            printf("Result = successful conversion\n");

            /*
* Try to convert using an invalid qualifier (YEAR to SECOND)
            */
            printf("\nInterval string #3 = 2007-02-11 3:10:35\n");
            in1.in_qual = TU_IENCODE(4, TU_YEAR, TU_SECOND);
            if(x = incvasc("2007-02-11 3:10:35", &in1))
        printf("Result = failed with conversion error:%d\n",x);
            else
            printf("Result = successful conversion\n");

            printf("\nINCVASC Sample Program over.\n\n");
            }
```

输出

```
INCVASC Sample ESQL Program running.
        Interval string #1 = 20 3:10:35
        Result = successful conversion

        Interval string #2 = 20 26:10:35
        Result = failed with conversion error:-1263

        Interval string #3 = 2007-02-11 3:10:35
        Result = failed with conversion error:-1268
```

5.2.131 incvfmtasc() 函数

incvfmtasc() 函数使用格式化掩码来将字符串转换为 interval 值。

语法

```
mint incvfmtasc(inbuf, fmtstring, invvalue)
```

```
char *inbuf;
char *fmtstring;
intrvl_t *invvalue;
```

inbuf

指向包含要转换的字符串的缓冲区的指针。

fmtstring

指向包含要用于 inbuf 字符串的格式化掩码的缓冲区的指针。此时间格式化掩码包含与 DBTIME 环境变量支持的相同的格式化伪指令。

invvalue

指向初始化了的 **interval** 变量的指针。

用法

您必须以您想要此变量拥有的限定符来初始化 *invvalue* 中的 **interval** 变量。**interval** 变量不需要指定与该格式化掩码相同的限定符。当 **interval** 限定符不同于隐含的格式化掩码限定符时，必要时，incvfmtasc() 将结果转化为恰当的单位。然而，两个限定符都必须属于同一 interval 类：或为 **year to month** 类，或为 **day to fraction** 类。

inbuf 中字符串中的所有字段都必须是相邻的。换句话说，如果限定符为 **hour to second**，则您必须为该字符串中某处的 **hour**、**minute** 和 **second** 指定所有值，否则，incvfmtasc() 返回错误。

inbuf 字符串可有开头和结尾的空格。然而，从第一个有效数字至最后一个，*inbuf* 仅可包含适于该格式化掩码暗指的限定符字段的数字和定界符。

如果该字符串是可接受的，则 incvfmtasc() 函数设置 *invvalue* 中的 **interval** 值，并返回零。否则，该函数返回错误代码，且 **interval** 变量包含不可预料的值。

DBTIME 环境变量接收的格式化伪指令 **%B**、**%b** 和 **%p** 在 *fmtstring* 中不适用，因为 *month name* 和 *a.m./p.m.* 信息与时间的间隔无关。如果 **interval** 多于 99 年，则请使用 **%Y** 伪指令（**%y** 仅可处理两位数字）。对于小时，请使用 **%H**（而不是 **%I**，因为 **%I** 仅可表示 12 小时）。如果 *fmtstring* 为空字符串，则该函数返回错误。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录在文件 incvfmtasc.ec 中包含此样例程序。

```
/* *incvfmtasc.ec*
        The following program illustrates the conversion of two strings to three interval values.
                */

                #include <stdio.h>

                EXEC SQL include datetime;

                main()
                {
                char out_str[30];
                char out_str2[30];
                char out_str3[30];
                mint x;


                EXEC SQL BEGIN DECLARE SECTION;
                interval day to minute short_time;
                interval minute(5) to second moment;
                interval hour to second long_moment;
                EXEC SQL END DECLARE SECTION;

            printf("INCVFMTASC Sample ESQL Program running.\n\n");

                /* Initialize short_time */
```

```
    printf("Interval value #1 = 20 days, 3 hours, 40 minutes\n");
            x = incvfmtasc("20 days, 3 hours, 40 minutes",
            "%d days, %H hours, %M minutes", &short_time);

            /*Convert the internal format to ascii in ANSI format, for displaying. */
            x = intoasc(&short_time, out_str);
  printf("Interval value (day to minute) = %s\n", out_str);

            /* Initialize moment */
     printf("\nInterval value #2 = 428 minutes, 30 seconds\n");
            x = incvfmtasc("428 minutes, 30 seconds",
            "%M minutes, %S seconds", &moment);

            /* Convert the internal format to ascii in ANSI format, for displaying. */
            x = intoasc(&moment, out_str2);
  printf("Interval value (minute to second) = %s\n", out_str2);

            /* Initialize long_moment */
  printf("\nInterval value #3 = 428 minutes, 30 seconds\n");
            x = incvfmtasc("428 minutes, 30 seconds",
            "%M minutes, %S seconds", &long_moment);

            /*Convert the internal format to ascii in ANSI format, for displaying. */
            x = intoasc(&long_moment, out_str3);
printf("Interval value (hour to second) = %s\n", out_str3);

            printf("\nINCVFMTASC Sample Program over.\n\n");
            }
```

输出

```
INVCFMTASC Sample ESQL Program running.

        Interval value #1 = 20 days, 3 hours, 40 minutes
        Interval value (day to minute) = 20 03:40
```

Interval value #2 = 428 minutes, 30 seconds

Interval value (minute to second) = 428:30

Interval value #3 = 428 minute, 30 seconds

Interval value (hour to second) = 7:08:30

5.2.132 intoasc() 函数

intoasc() 函数将 interval 变量的字段值转换为符合 ANSI SQL 标准的 ASCII 字符串。

语法

mint intoasc(*invvalue*, *outbuf*)

intrvl_t *invvalue;

char *outbuf;

invvalue

指向要转换的初始化了的 **interval** 变量的指针。

outbuf

指向接收 *invvalue* 中的值的 ANSI 标准 INTERVAL 字符串的缓冲区的指针。

用法

intoasc() 函数将 **interval** 变量中字段的数字转换为它们的等同的字符，并将它们以它们之间的的定界符（连字符、空格、冒号或句号）复制到 *outbuf* 字符串。您必须以您想要该字符串拥有的限定符来初始化 *invvalue* 中的 **interval** 变量。

该字符串不包括 SQL 语句用来定界 INTERVAL 文字的限定符或圆括号。*outbuf* 字符串符合 ANSI SQL 标准。对于每一定界符（连字符、空格、冒号或句号），它包括一个字符外加带有下列大小的字段。

字段	字段大小
开头字段	由精度指定
小数	由精度指定

所有其他字段　　两位

带有 **day(5) to fraction(5)** 限定符的 **interval** 值产生输出的最大长度。该等同的字符串包含 16 数字、4 定界符以及空终止符，总计 21 字节：

DDDDD HH:MM:SS.FFFFF

如果您未初始化 **interval** 变量的限定符，则 intoasc() 函数返回不可预计的值，但此值不会超出 21 字节。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录在文件 intoasc.ec 中包含此样例程序。

```
/*
        * intoasc.ec *

 The following program illustrates the conversion of an interval (intvl_t) into an ASCII
string.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        mint x;
        char out_str[10];

        EXEC SQL BEGIN DECLARE SECTION;
        interval day(3) to day in1;
```

```
EXEC SQL END DECLARE SECTION;

printf("INTOASC Sample ESQL Program running.\n\n");

printf("Interval (day(3) to day) string is '3'\n");
if(x = incvasc("3", &in1))
printf("Initial conversion failed with error: %d\n",x);
else
{
/* Convert the internal format to ascii for displaying */
intoasc(&in1, out_str);
printf("\tInterval value after conversion    is '%s'\n", out_str);
}

printf("\nINTOASC Sample Program over.\n\n");
}
```

输出

```
INTOASC Sample ESQL Program running.

        Interval (day(3) to day) string is '3'
        Interval value afer conversion    is '      3'
```

5.2.133 intofmtasc() 函数

intofmtasc() 函数使用格式化掩码来将 interval 变量转换为字符串。

语法

```
mint intofmtasc(invvalue, outbuf, buflen, fmtstring)
        intrvl_t *invvalue;
        char *outbuf;
        mint buflen;
```

char *fmtstring;

invvalue

指向要转换的初始化了的 **interval** 变量的指针。

outbuf

指向为 invvalue 中的值接收字符串的缓冲区的指针。

buflen

outbuf 缓冲区的长度。

fmtstring

指向包含用于 outbuf 字符串的格式化掩码的缓冲区的指针。此时间格式化掩码包含与 DBTIME 环境变量支持的相同的格式化伪指令。

用法

您必须以您想要该字符串拥有的限定符来初始化 *invvalue* 中的 **interval** 变量。如果您未初始化 **interval** 变量，则该函数返回不可预料的值。*outbuf* 中的字符串不包括 SQL 语句用来定界 INTERVAL 文字的限定符或圆括号。

格式化掩码 *fmtstring* 不需要暗示与 **interval** 变量相同的限定符。当暗示的格式化掩码限定符不同于 **interval** 限定符时，在必要时，intofmtasc() 将该结果转换为恰当的单位（如同它调用了 invextend() 函数那样）。然而，两个限定符必须同时属于同一类：或者 **year to month** 类，或者 **day to fraction** 类。

如果 *fmtstring* 为空字符串，则 intofmtasc() 函数将 *outbuf* 设置为空字符串。

DBTIME 环境变量接受的格式化伪指令 **%B**、**%b** 和 **%p** 不适用于 *fmtstring*，因为 *month name* 和 *a.m./p.m.* 信息与时间的间隔无关。如果 **interval** 大于 99 年，则请使用 **%Y** 伪指令（**%y** 仅可处理两位数）。对于小时，请使用 **%H**（而不是 **%I**，因为 **%I** 仅可表示 12 小时）。如果 *fmtstring* 为空字符串，则该函数返回错误。

如果该字符串和格式化掩码是可接受的，则 intofmtasc() 函数设置 *invvalue* 中的 **interval** 值，并返回零。否则，该函数返回错误代码，且 **interval** 变量包含不可预料的值。

返回代码

0

转换成功。

<0

转换失败。

示例

demo 目录在文件 intofmtasc.ec 中包含此样例程序。

```
/*
        *intofmtasc.ec*
The following program illustrates the conversion of interval values to ASCII strings with the specified formats.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        char out_str[60];
        char out_str2[60];
        char out_str3[60];
        mint x;

        EXEC SQL BEGIN DECLARE SECTION;
        interval day to minute short_time;
        interval minute(5) to second moment;
        EXEC SQL END DECLARE SECTION;

 printf("INTOFMTASC    Sample ESQL Program running.\n\n");

* Initialize short_time (day to minute) interval value */
 printf("Interval string #1 = '20 days, 3 hours, 40 minutes'\n");
   x = incvfmtasc("20 days, 3 hours, 40 minutes",
     "%d days, %H hours, %M minutes", &short_time);
 /*Turn the interval into ascii string of a certain format. */
```

```
x = intofmtasc(&short_time, out_str, sizeof(out_str),
"%d days, %H hours, %M minutes to go!");
printf("\tFormatted value: %s\n", out_str);

/* Initialize moment (minute(5) to second interval value */
  printf("\nInterval string #2: '428 minutes, 30 seconds'\n");
    x = incvfmtasc("428 minutes, 30 seconds",
    "%M minutes, %S seconds", &moment);

                /* Turn each interval into ascii string of a certain format. Note
                * that the second and third calls to intofmtasc both use moment
                * as the input variable, but the output strings have different    formats.
                */
                x = intofmtasc(&moment, out_str2, sizeof(out_str2),
                "%M minutes and %S seconds left.");
                x = intofmtasc(&moment, out_str3, sizeof(out_str3),
                "%H hours, %M minutes, and %S seconds still left.");

                /* Print each resulting string */
                printf("\tFormatted value: %s\n", out_str2);
                printf("\tFormatted value: %s\n", out_str3);

                printf("\nINTOFMTASC Sample Program over.\n\n");
                }
```

输出

```
INTOFMTASC Sample ESQL Program running.

        Interval string #1: '20 days, 3 hours, 40 minutes'
        Formatted value: 20 days, 03 hours, 40 minutes to go!

        Interval string #2: '428 minutes, 30 seconds'
        Formatted value: 428 minutes and 30 seconds left.
        Formatted value: 07 hours, 08 minutes, and 30 seconds still left.
```

5.2.134 invdivdbl() 函数

invdivdbl() 函数将 interval 值除以一个数值的值。

语法

mint invdivdbl(*iv*, *num*, *ov*)

intrvl_t *iv;

double num;

intrvl_t *ov;

iv

指向要被除的 **interval** 变量的指针。

num

数值的除数值。

ov

指向带有有效的限定符的 **interval** 变量的指针。

用法

输入和输出限定符必须同时属于同一 **interval** 类：或者 **year to month** 类，或者 **day to fraction(5)** 类。如果 *ov* 的限定符不同于（同一类之内的）*iv* 的限定符，则 invdivdbl() 函数扩展该结果（如 invextend() 函数定义的那样）。

invdivdbl() 函数将 *iv* 中的 **interval** 值除以 *num*，并将结果存储在 *ov* 中。

num 中的值可为正值或负值。

返回代码

0

除法成功。

<0

除法失败。

-1200

数值的值（大小）太大。

-1201

数值的值（大小）太小。

-1202

num 参数为零（0）。

-1265

interval 操作发生溢出。

-1266

interval 值与该操作不兼容。

-1268

参数包含无效的 **interval** 限定符。

示例

demo 目录在文件 invdivdbl.ec 中包含此样例程序。

```
/*
        * indivdbl.ec *

    The following program divides an INTERVAL type variable by a numeric value and
stores the result in an INTERVAL variable. The operation is done twice, using INTERVALs with
different qualifiers to store the result.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        char out_str[16];

        EXEC SQL BEGIN DECLARE SECTION;
        interval day to second daytosec1;
        interval hour to minute hrtomin;
        interval day to second daytosec2;
```

```
    EXEC SQL END DECLARE SECTION;

    printf("INVDIVDBL Sample ESQL Program running.\n\n");

  /* Input is 3 days, 5 hours, 27 minutes, and 30 seconds */
   printf("Interval (day to second) string = '3 5:27:30'\n");
    incvasc("3 5:27:30", &daytosec1);

  /* Divide input value by 3.0, store in hour to min interval*/
    invdivdbl(&daytosec1, (double) 3.0, &hrtomin);

  /* Convert the internal format to ascii for displaying */
    intoasc(&hrtomin, out_str);
    printf("Divisor (double)     =    3.0    \n");
    printf("---------------------------------\n");
    printf("Quotient #1 (hour to minute)='%s'\n", out_str);

 /* Divide input value by 3.0, store in day to sec interval variable */
   invdivdbl(&hrtomin, (double) 3.0, &daytosec2);

 /* Convert the internal format to ascii for displaying */
    intoasc(&daytosec2, out_str);
    printf("Quotient #2 (day to second) = '%s'\n", out_str);
    printf("\nINVDIVDBL Sample Program over.\n\n");
     }
```

输出

```
INVDIVDBL Sample ESQL Program running.

    Interval (day to second) string = '3 5:27:30'
    Divisor (double)                 =          3.0
    -------------------------------------------
    Quotient #1 (hour to minute)     = '    25:49'
    Quotient #2 (day to second) =    '    1 01:49:10'
```

INVDIVDBL Sample Program over.

5.2.135 invdivinv() 函数

invdivinv() 函数将 interval 值除以另一 interval 值。

语法

mint invdivinv(*i1, i2, num*)

intrvl_t *i1, *i2;

double *num;

i1

指向被除的 **interval** 变量的指针。

i2

指向为除数的 **interval** 变量的指针。

num

指向为商的 **double** 值的指针。

用法

invdivinv() 函数将 *i1* 中的 **interval** 值除以 *i2*，并将结果存储在 *num* 中。该结果可为正的或负的。

输入和输出限定符必须同时属于同一 **interval** 类：或者 **year to month** 类，或者 **day to fraction(5)** 类。如果有必要，则在除法之前，invdivinv() 扩展 *i2* 中的 **interval** 值，来匹配 *i1* 的限定符。

返回代码

0

除法成功。

<0

除法失败。

-1200

数值的值（大小）太大。

-1201

数值的值（大小）太小。

-1266

interval 值与该操作不兼容。

-1268

参数包含无效的 **interval** 限定符。

示例

demo 目录在文件 invdivinv.ec 中包含此样例程序。

```
/*
        * invdivinv.ec *
  The following program divides one interval value by another and displays the resulting
numeric value.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        mint x;
        char out_str[16];

        EXEC SQL BEGIN DECLARE SECTION;
        interval hour to minute hrtomin1, hrtomin2;
        double res;
        EXEC SQL END DECLARE SECTION;

        printf("INVDIVINV Sample ESQL Program running.\n\n");

        printf("Interval #1 (hour to minute) = 75:27\n");
        incvasc("75:27", &hrtomin1);
        printf("Interval #2 (hour to minute) = 19:10\n");
```

```
    incvasc("19:10", &hrtomin2);

    printf("-----------------------------------------\n");
    invdivinv(&hrtomin1, &hrtomin2, &res);
    printf("Quotient (double)    =    %.1f\n", res);

    printf("\nINVDIVINV Sample Program over.\n\n");
    }
```

输出

```
INVDIVINV Sample ESQL Program running.

    Interval #1 (hour to minute) = 75.27
    Interval #2 (hour to minute) = 19:10
    -----------------------------------
    Quotient (double)            =    3.9
INVDIVINV Sample Program over.
```

5.2.136 invextend() 函数

invextend() 函数复制不同限定符之下的 interval 值。

扩展是添加或删除 INTERVAL 值的字段的操作，来使得它与给定的限定符相匹配。对于 INTERVAL 值，两个限定符都必须属于同一 **interval** 类：或者 **year to month** 类，或者 **day to fraction(5)** 类。

语法

```
mint invextend(in_inv, out_inv)
    intrvl_t *in_inv, *out_inv;
```

in_inv

指向要扩展的 **interval** 变量的指针。

out_inv

指向带有有效的限定符用于扩展的 **interval** 变量的指针。

用法

invextend() 函数将 *in_inv***interval** 变量的限定符字段数字复制至 *out_inv* **interval** 变量。*out_inv* 变量的限定符控制该复制。

该函数丢弃 *in_inv* 中最低有效字段右边的 *out_inv* 中的任何字段。该函数填写未出现在 *in_inv* 中的 *out_inv* 中的任何字段，如下：

它以零填充 *in_inv* 中最低有效字段右边的字段。

它将 *in_inv* 中最高有效字段左边的字段设置为有效的 **interval** 值。

返回代码

0

转换成功。

<0

转换失败。

-1266

interval 值与该操作不兼容。

-1268

参数包含无效的 **interval** 限定符。

示例

demo 目录在文件 invextend.ec 中包含此样例程序。该示例程序说明 **interval** 扩展。在第二个结果中，在 **seconds** 字段中，输出包含零，且已将 **days** 字段设置为 3。

```
/*
        * invextend.ec *

The following program illustrates INTERVAL extension. It extends an INTERVAL value
to another INTERVAL value with a different qualifier. Note that in the second example, the
output contains zeros in the seconds field and the days field has been set to 3.

        */
```

```
#include <stdio.h>
EXEC SQL include datetime;

main()
{
mint x;
char out_str[16];
;
EXEC SQL BEGIN DECLARE SECTION;
interval hour to minute hrtomin;
interval hour to hour hrtohr;
interval day to second daytosec;
EXEC SQL END DECLARE SECTION;

printf("INVEXTEND Sample ESQL Program running.\n\n");

 printf("Interval (hour to minute) value =          75.27\n");
incvasc("75:27", &hrtomin);

/* Extend to hour-to-hour and convert the internal format to
* ascii for displaying
*/
invextend(&hrtomin, &hrtohr);
intoasc(&hrtohr, out_str);
printf("Extended (hour to hour) value =      %s\n", out_str);

/* Extend to day-to-second and convert the internal format to    ascii for displaying
*/
invextend(&hrtomin, &daytosec);
intoasc(&daytosec, out_str);
printf("Extended (day to second) value =: %s\n", out_str);

printf("\nINVEXTEND Sample Program over.\n\n");
```

```
}
```

输出

```
INVEXTEND Sample ESQL Program running.


        Interval (hour to minute) value =       75:27
        Extended (hour to hour) value     =       75
        Extended (day to second) value    =    3 03:27:00
INVEXTEND Sample Program over.
```

5.2.137 invmuldbl() 函数

invmuldbl() 函数将 interval 值乘以数值的值。

语法

```
mint invmuldbl(iv, num, ov)
        intrvl_t *iv;
        double num;
        intrvl_t *ov;
```

iv

指向要相乘的 **interval** 变量的指针。

num

数值的 **double** 值。

ov

指向带有有效的限定符的 **interval** 变量的指针。

用法

invmuldbl() 函数将 *iv* 中的 **interval** 值乘以 *num*，并将结果存储在 *ov* 中。*num* 中的值可正可负。

输入和输出限定符都必须属于同一 **interval** 类；或者 **year to month** 类，或者 **day to fraction(5)** 类。如果 *ov* 的限定符不同于 *iv* 的限定符（但属于同于类），则 invmuldbl() 扩展该结果（如 invextend() 函数定义的那样）。

返回代码

0

乘法成功。

<0

乘法失败。

-1200

数值的值（大小）太大。

-1201

数值的值（大小）太小。

-1266

interval 值与该操作不兼容。

-1268

参数包含无效的 **interval** 限定符。

示例

demo 目录的文件 invmuldbl.ec 中包含此样例程序。该示例说明如何将 **interval** 值与数值的值相乘。第二个相乘说明当输入与输出限定符不同时 **interval** 乘法的结果。

```
/*
        * invmuldbl.ec *

The following program multiplies an INTERVAL type variable by a numeric value  and
stores the result in an INTERVAL variable. The operation is done twice, using INTERVALs with
different qualifiers to store the result.
        */

        #include <stdio.h>

        EXEC SQL include datetime;

        main()
        {
        char out_str[16];
```

```
    EXEC SQL BEGIN DECLARE SECTION;
    interval hour to minute hrtomin1;
    interval hour to minute hrtomin2;
    interval day to second daytosec;
    EXEC SQL END DECLARE SECTION;

    printf("INVMULDBL Sample ESQL Program running.\n\n");


    /* input is 25 hours, and 49 minutes */
    printf("Interval (hour to minute)    =    25:49\n");
    incvasc("25:49", &hrtomin1);
    printf("Multiplier (double)    =        3.0\n");
    printf("---------------------------------------\n");

  /* Convert the internal format to ascii for displaying */
    invmuldbl(&hrtomin1, (double) 3.0, &hrtomin2);
    intoasc(&hrtomin2, out_str);
   printf("Product #1 (hour to minute)= '%s'\n", out_str);

 /* Convert the internal format to ascii for displaying */
    invmuldbl(&hrtomin1, (double) 3.0, &daytosec);
    intoasc(&daytosec, out_str);
  printf("Product #2 (day to second)='%s'\n", out_str);

    printf("\nINVMULDBL Sample Program over.\n\n");
    }
```

输出

```
INVMULDBL Sample ESQL Program running.

        Interval (hour to minute) =          25:49
        Multiplier (double) =               3.0
        ----------------------------------------
```

Product #1 (hour to minute) = ' 77:27'

Product #2 (day to second) = ' 3 05:27:00'

INVMULDBL Sample Program over.

5.2.138 ldchar() 函数

ldchar() 函数将定长的字符串复制至以空终止的字符串内，并移除任何结尾的空格。

语法

void ldchar(*from, count, to*)

char *from;

mint count;

char *to;

from

指向定长的源字符串的指针。

count

定长的源字符串中的字节数。

to

指向以空为终止的目标字符串的第一个字节的指针。to 参数可指向与 *from* 参数相同的位置，或指向覆盖 *from* 参数的位置。如果如此，则 ldchar() 不保留 *from 指向的值*。

示例

demo 目录中 ldchar.ec 的文件中为此样例程序。

```
/*
        * ldchar.ec *

  The following program loads characters to specific locations in an array that is initialized
to z's.  It displays the result of each ldchar()  operation.
        */

        #include <stdio.h>

        main()
```

```
{
static char src1[] = "abcd      ";
static char src2[] = "abcd    g    ";
static char dest[40];

printf("LDCHAR Sample ESQL Program running.\n\n");

ldchar(src1, stleng(src1), dest);
printf("\tSource: [%s]\n\tDestination: [%s]\n\n", src1, dest);

ldchar(src2, stleng(src2), dest);
printf("\tSource: [%s]\n\tDestfination: [%s]\n", src2, dest);

printf("\nLDCHAR Sample Program over.\n\n");
}
```

输出

```
LDCHAR Sample ESQL Program running.

        Source: [abcd      ]
        Destination: [abcd]

        Source: [abcd    g    ]
        Destination: [abcd    g]
LDCHAR Sample Program over.
```

5.2.139 rdatestr() 函数

rdatestr() 函数将内部的 DATE 转换为字符串。

语法

```
mint rdatestr(jdate, outbuf)
        int4 jdate;
        char *outbuf;
jdate
```

要格式化的日期的内部表示。

outbuf

指向接收 jdate 值的字符串的缓冲区的指针。

用法

对于缺省的语言环境 US English，rdatestr() 获取确定如何以下列优先顺序解释该字符串的格式：

DBDATE 环境变量指定的格式（如果设置 DBDATE 的话）。

GL_DATE 环境变量指定的格式（如果设置 GL_DATE 的话）。

缺省的日期样式：mm/dd/yyyy。

当您使用非缺省的语言环境，且未设置 DBDATE or GL_DATE 环境变量，则 rdatestr() 使用客户机语言环境定义的日期终端用户格式。

返回代码

0

转换成功。

<0

转换失败。

-1210

不会将内部的日期转换为字符串格式。

-1212

数据转换格式必须包含月份、日期或年份组件。DBDATE 指定数据转换格式。

示例

demo 目录在 rtoday.ec 文件中包含此样例程序。

```
/*
        * rtoday.ec *

The following program obtains today's date from the system.
 It then converts it to ASCII for displaying the result.
```

```
*/

#include <stdio.h>

main()
{
mint errnum;
char today_date[20];
int4 i_date;

printf("RTODAY Sample ESQL Program running.\n\n");

/* Get today's date in the internal format */
rtoday(&i_date);

/* Convert date from internal format into a mm/dd/yyyy string */
if ((errnum = rdatestr(i_date, today_date)) == 0)
printf("\n\tToday's date is %s.\n", today_date);
else
printf("\n\tError %d in converting date to mm/dd/yyyy\n", errnum);

printf("\nRTODAY Sample Program over.\n\n");
}
```

输出

```
RTODAY Sample ESQL Program running.

        Today's date is 10/26/2007.
RTODAY Sample Program over.
```

5.2.140 rdayofweek() 函数

对于 内部 DATE，rdayofweek() 函数返回星期几作为整数值。

语法

mint rdayofweek(*jdate*)

int4 jdate;

jdate

日期的内部表示。

返回代码

0

星期日

1

星期一

2

星期二

3

星期三

4

星期四

5

星期五

6

星期六

示例

demo 目录在 rdayofweek.ec 文件中包含此样例程序。

```
/*
        * rdayofweek.ec *

 The following program accepts a date entered from the console.
        */
```

```
#include <stdio.h>

main()
{
mint errnum;
int4 i_date;
char *day_name;
char date[20];
int x;

static char fmtstr[9] = "mmddyyyy";

 printf("RDAYOFWEEK Sample ESQL Program running.\n\n");

/* Allow user to enter a date */
printf("Enter a date as a single string, month.day.year\n");
gets(date);

printf("\nThe date string is %s.\n", date);

/* Put entered date in internal format */
if (x = rdefmtdate(&i_date, fmtstr, date))
printf("Error %d on rdefmtdate conversion\n", x);
else
{
/* Figure out what day of the week i_date is */
switch (rdayofweek(i_date))
{
case 0:     day_name = "Sunday";
break;
case 1:     day_name = "Monday";
break;
```

```
        case 2:     day_name = "Tuesday";
        break;
        case 3:     day_name = "Wednesday";
        break;
        case 4:     day_name = "Thursday";
        break;
        case 5:     day_name = "Friday";
        break;
        case 6:     day_name = "Saturday";
        break;
        }
        printf("This date is a %s.\n", day_name);
        }

        printf("\nRDAYOFWEEK Sample Program over.\n\n");
        }
```

输出

```
RDAYOFWEEK Sample ESQL Program running.

        Enter a date as a single string, month.day.year
        10.13.07

        The date string is 10.13.07.
        This date is a Saturday.
RDAYOFWEEK Sample Program over.
```

5.2.141 rdefmtdate() 函数

rdefmtdate() 函数使用格式化掩码来将字符串转换为内部的 DATE 格式。

语法

```
mint rdefmtdate(jdate, fmtstring, inbuf)
        int4 *jdate;
```

```
char *fmtstring;
char *inbuf;
```

jdate

指向接收 *inbuf* 字符串的内部 DATE 值的 **int4** 整数值的指针。

fmtstring

指向包含要使用 *inbuf* 字符串的的格式化掩码的缓冲区的指针。

inbuf

指向包含要转换的日期字符串的缓冲区的指针。

用法

rdefmtdate() 函数的 *fmtstring* 参数指向日期格式化掩码，其包含描述如何解释该日期字符串的格式。

input 字符串和 *fmtstring* 必须以相同的月份、日期和年份的序列顺序。然而，它们不需要包含相同的文字，或月份、日期和年份的相同表示。

在 *fmtstring* 中，您可包括 weekday 格式（ww），但数据库服务器忽略该格式。*inbuf* 中没有与 weekday 格式对应的内容。

下列 *fmtstring* 与 *input* 的组合是有效的。

格式化掩码输入

mmddyy

Dec. 25th, 2007

mmddyyyy

Dec. 25th, 2007

mmm. dd. yyyy

dec 25 2007

mmm. dd. yyyy

DEC-25-2007

mmm. dd. yyyy

122507

mmm. dd. yyyy

12/25/07

yy/mm/dd

07/12/25

yy/mm/dd

2007, December 25

yy/mm/dd

In the year 2007, the month of December, it is the 25th day

dd-mm-yy

This 25th day of December 2007

如果存储在 *inbuf* 中的值是四位年份。则 rdefmtdate() 函数使用那个值。如果存储在 *inbuf* 中的值是两位年份，则 rdefmtdate() 函数使用 DBCENTURY 环境变量的值来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 GBase 8s ESQL/C 使用 20 世纪。

返回代码

如果您使用无效的日期字符串格式，则 rdefmtdate() 返回错误代码，并将该内部的 DATE 设置为当前日期。下列为可能的返回代码。

0

操作成功。

-1204

**input* 参数指定无效的年份。

-1205

**input* 参数指定无效的月份。

-1206

**input* 参数指定无效的日期。

-1209

由于 **input* 不包含年份、月份和日期之间的定界符，则 **input* 的长度必须恰好为 6 或 8 字节。

-1212

**fmtstring* 未指定年份、月份和日期。

示例

demo 目录在 rdefmtdate.ec 文件中包含此样例程序。

```
/*
                * rdefmtdate.ec *
```

The following program accepts a date entered from the console, converts it into the internal date format using rdefmtdate(). It checks the conversion by finding the day of the week.

```
*/

#include <stdio.h>

main()
{
mint x;
char date[20];
int4 i_date;
char *day_name;

static    char fmtstr[9] = "mmddyyyy";

printf("RDEFMTDATE Sample ESQL Program running.\n\n");

printf("Enter a date as a single string, month.day.year\n");
gets(date);

printf("\nThe date string is %s.\n", date);

if (x = rdefmtdate(&i_date, fmtstr, date))
printf("Error %d on rdefmtdate conversion\n", x);
else
{
/* Figure out what day of the week i_date is */
switch (rdayofweek(i_date))
{
case 0:     day_name = "Sunday";
break;
case 1:     day_name = "Monday";
```

```
        break;
        case 2:     day_name = "Tuesday";
        break;
        case 3:     day_name = "Wednesday";
        break;
        case 4:     day_name = "Thursday";
        break;
        case 5:     day_name = "Friday";
        break;
        case 6:     day_name = "Saturday";
        break;
        }
        printf("\nThe day of the week is %s.\n", day_name);
        }

        printf("\nRDEFMTDATE Sample Program over.\n\n");
        }
```

输出

```
RDEFMTDATE Sample ESQL Program running.

        Enter a date as a single string, month.day.year
        080894

        The date string is 080894
        The day of the week is Monday.
RDEFMTDATE Sample Program over.
```

5.2.142 rdownshift() 函数

rdownshift() 函数将以空终止的字符串内所有大写字符更改为小写字符。

语法

```
void rdownshift(s)
        char *s;
```

s

指向以空终止的字符串的指针。

用法

rdownshift() 函数引用当前的语言环境，来确定大写和小写字母。对于缺省的语言环境 US English，rdownshift() 使用 ASCII 小写字母（a-z）和大写字母（A-Z）。

如果您使用非缺省的语言环境，则 rdownshift() 使用该语言环境定义的小写和大写字母。

返回代码

此样例程序在 demo 目录中的 rdownshift.ec 文件中。

```
/*
        * rdownshift.ec *

    The following program uses rdownshift() on a string containing alphanumeric and
punctuation characters.
        */

        #include <stdio.h>

        main()
        {
        static char string[] = "123ABCDEFGHIJK'.;";

    printf("RDOWNSHIFT Sample ESQL Program running.\n\n");

        printf("\tInput string...: [%s]\n", string);
        rdownshift(string);
        printf("\tAfter downshift: [%s]\n", string);

        printf("\nRDOWNSHIFT Sample Program over.\n\n");
        }
```

输出

RDOWNSHIFT Sample ESQL Program running.

Input string...: [123ABCDEFGHIJK'.;]

After downshift: [123abcdefghijk'.;]

RDOWNSHIFT Sample Program over.

5.2.143 ReleaseConnect() 函数（Windows(TM)）

ReleaseConnect() 函数仅在 Windows(TM) 环境中可用。 它释放或终止显式的连接，并清除所有分配了的内存。

重要： 对于与 Version 5.01 GBase 8s ESQL/C 相兼容的 Windows(TM) 应用程序，GBase 8s ESQL/C 支持 ReleaseConnect() 连接库函数。当您编写新的 Windows(TM) 环境的 GBase 8s ESQL/C 应用程序时，请使用 SQL DISCONNECT 语句来终止建立了的显式的连接。

语法

void *ReleaseConnect (void **CnctHndl*)

CnctHndl

由前面的 GetConnect() 调用返回的连接句柄。

用法

ReleaseConnect() 函数映射到简单的 SQL DISCONNECT 语句（*没有* ALL、CURRENT 或 DEFAULT 选项的）。 ReleaseConnect() 调用*自身*等同于下列 SEL 语句：

EXEC SQL disconnect *db_connection*;

在此示例中，*db_connection* 是 is the name of an existing connection that the GetConnect() 函数已建立的现有连接的名称。 您将此 *db_connection* 名称作为参数传递至 ReleaseConnect()；它是所需要的连接的连接句柄。

例如，下列代码片段使用 ReleaseConnect() 来关闭缺省的数据库服务器上的至 **stores7** 数据库的显式的连接：

void *cnctHndl;

```
⋮

cnctHndl = GetConnect();
EXEC SQL database stores7;
⋮

EXEC SQL close database;
cnctHndl = ReleaseConnect( cnctHndl );
```

对于 GetConnect() 已建立了的每一连接，请调用 ReleaseConnect() 一次。在 ReleaseConnect() 函数终止当前的连接之前，它关闭任何打开的数据库。如果在当前连接中任何事务是打开的，则它失败。

在调用 ReleaseConnect() 之前，以 SQL CLOSE DATABASE 语句显式地关闭数据库是一种好的编程实践。

重要： 由于 ReleaseConnect() 函数映射至 DISCONNECT 语句，因此它设置 SQLCODE 和 SQLSTATE 状态代码，来指示该连接终止请求成功还是失败。此行为不同于 Windows(TM) 的 Version 5.01 GBase 8s ESQL/C 中的 ReleaseConnect()，此函数不在其中设置 SQLCODE 和 SQLSTATE 值。

ReleaseConnect() 函数与 DISCONNECT 语句的不同之处在于，它取得连接名称。ReleaseConnect() 使用存储在连接句柄中的内部生成的名称；您必须至此此句柄作为 ReleaseConnect() 调用的参数。仅对于不带有 AS 子句的 CONNECT 语句已建立了的连接，DISCONNECT 语句才使用内部生成的连接名称；如果该连接有（CONNECT 语句的 AS 子句指定的）用户定义的连接名称，则 DISCONNECT 使用此名称。

返回代码

CnctHndl

如果 ReleaseConnect() 已返回了传递给它的连接句柄相匹配的连接句柄，则对它的调用成功。

5.2.144 rfmtdate() 函数

rfmtdate() 函数使用格式化掩码来将内部的 DATE 格式转换为字符串。

语法

```
mint rfmtdate(jdate, fmtstring, outbuf)
        int4 jdate;
        char *fmtstring;
        char *outbuf;
```

jdate

要转换的日期的内部表示。

fmtstring

指向包含要使用该 jdate 值的格式化掩码的缓冲区的指针。

outbuf

指向接收该 jdate 值的格式化的字符串的缓冲区的指针。

用法

rfmtdate() 函数的 *fmtstring* 参数指向日期格式化掩码，其包含描述如何格式化该日期字符串的格式。

下列列表中的这些示例使用 *fmtstring* 中的格式化掩码将整数 *jdate*，其值对应于 2007 年 12 月 25 日，转换为格式化的字符串 *outbuf*。您必须指定一个或多个字段。

格式化掩码格式化的结果

"mmdd"

1225

"mmddyy"

122507

"ddmmyy"

251207

"yydd"

0725

"yymmdd"

071225

"dd"

25

"yy/mm/dd"

07/12/25

"yy mm dd"

07 12 25

"yy-mm-dd"

07-12-25

"mmm. dd, yyyy"

Dec. 25, 2007

"mmm dd yyyy"

Dec 25 2007

"yyyy dd mm"

2007 25 12

"mmm dd yyyy"

Dec 25 2007

"ddd, mmm. dd, yyyy"

Tue, Dec. 25, 2007

"ww mmm. dd, yyyy"

Tue Dec. 25, 2007

"(ddd) mmm. dd, yyyy"

(Tue) Dec. 25, 2007

"mmyyddmm"

25071225

""

不可预料的结果

返回代码

0

转换成功。

-1210

不可将内部的日期转换为 month-day-year 格式。

-1211

程序用尽内存（内存分配错误）。

-1212

格式字符串为 NULL 或无效。

示例

demo 目录在 rfmtdate.ec 文件中包含此样例程序。

```
/*
        * rfmtdate.ec *
```

The following program converts a date from internal format to a specified format using rfmtdate().

```
        */

        #include <stdio.h>

        main()
        {
        char the_date[15];
        int4 i_date;
        mint x;
        int errnum;
        static short mdy_array[3] = { 12, 10, 2007 };

        printf("RFMTDATE Sample ESQL Program running.\n\n");

        if ((errnum = rmdyjul(mdy_array, &i_date)) == 0)
        {

        /*
        * Convert date to "mm-dd-yyyy" format
        */
        if (x = rfmtdate(i_date, "mm-dd-yyyy", the_date))
    printf("First rfmtdate() call failed with error %d\n", x);
        else
      printf("\tConverted date (mm-dd-yyy): %s\n", the_date);

        /*
        * Convert date to "mm.dd.yy" format
```

```
        */
        if (x = rfmtdate(i_date, "mm.dd.yy", the_date))
    printf("Second rfmtdate() call failed with error %d\n",x);
        else
    printf("\tConverted date (mm.dd.yy): %s\n", the_date);

        /*
        * Convert date to "mmm ddth, yyyy" format
        */
        if (x = rfmtdate(i_date, "mmm ddth, yyyy", the_date))
  printf("Third rfmtdate() call failed with error %d\n", x);
        else
  printf("\tConverted date (mmm ddth, yyyy): %s\n", the_date);
        }

        printf("\nRFMTDATE Sample Program over.\n\n");
        }
```

输出

```
RFMTDATE Sample ESQL Program running.

        Converted date (mm-dd-yyy): 12-10-2007.
        Converted date (mm.dd.yy): 12.10.07.
        Converted date (mmm ddth, yyyy): Dec 10th, 2007

        RFMTDATE Sample Program over.
```

5.2.145 rfmtdec() 函数

rfmtdec() 函数使用格式化掩码来将 decimal 值转换为字符串。

语法

```
mint rfmtdec(dec_val, fmtstring, outbuf)
        dec_t *dec_val;
        char *fmtstring;
```

char *outbuf;

dec_val

指向要格式化的 **decimal** 值的指针。

fmtstring

指向包含要用于 dec_val 值的格式化掩码的字符缓冲区的指针。

outbuf

指向接收 dec_val 值的格式化的字符串的字符缓冲区的指针。

用法

rfmtdec() 函数的 *fmtstring* 参数指向数值格式化掩码，其包含描述如何格式化 decimal 值的字符。

当您使用 rfmtdec() 来格式化 MONEY 值时，该函数使用 DBMONEY 环境变量指定的货币符号。如果您未设置此环境变量，则 rfmtdec() 使用客户机语言环境定义的货币符号。缺省的语言环境 US English 定义货币符号，就如同您将 DBMONEY 设置为 “$,.” 一样。

当您使用有多字节代码集的非缺省的语言环境时，rfmtdec() 支持格式字符串中的多字节字符。

返回代码

0

转换成功。

-1211

程序用尽内存（内存分配错误）。

-1217

格式字符串太大。

示例

demo 目录在文件 rfmtdec.ec 中包含此样例程序。

```
/*
        * rfmtdec.ec *
  The following program applies a series of format specifications to each of a series of
DECIMAL numbers and displays each result.
        */
```

```
#include <stdio.h>

EXEC SQL include decimal;

char *strings[] =
{
"210203.204",
"4894",
"443.334899312",
"-12344455",
0
};

char *formats[] =
{
"**##########",
"$$$$$$$$$.##",
"(&&,&&&,&&&.)",
"<,<<<,<<<,<<<",
"$*********.**",
0
};

char result[41];

main()
{
mint x;
mint s = 0, f;
dec_t num;

printf("RFMTDEC Sample ESQL Program running.\n\n");
```

```
    while(strings[s])
    {
    /*
    *    Convert each string to DECIMAL
    */
    printf("String = %s\n", strings[s]);
  if (x = deccvasc(strings[s], strlen(strings[s]), &num))
    {
printf("Error %d in converting string [%s] to decimal\n",
    x, strings[s]);
    break;
    }
    f = 0;
    while(formats[f])
    {
    /*
    * Format DECIMAL num for each of formats[f]
    */
    rfmtdec(&num, formats[f], result);
    /*
    * Display result and bump to next format (f++)
    */
    result[40] = '\0';
    printf("    Format String = '%s'\t", formats[f++]);
    printf("\tResult = '%s'\n", result);
    }
    ++s;                /* bump to next string */
    printf("\n");           /* separate result groups */
    }

    printf("\nRFMTDEC Sample Program over.\n\n");
    }
```

输出

RFMTDEC Sample ESQL Program running.

```
String = 210203.204
Format String = '**###########'
Result = '**         210203'
Format String =    '$$$$$$$$$$.##'
Result = '      $210203.20'
Format String =    '(&&,&&&&,&&&.)'
Result = ' 000210,203. '
Format String =    '<,<<<,<<<,<<<'
Result = '210,203'
Format String = ' $*********.**'
Result = '$***210203.20'

String = 4894
Format String = '**###########'
Result = ' ** 4894'
Format String =    '$$$$$$$$$$.##'
Result = ' $4894.00'
Format String =    '(&&,&&&&,&&&.)'
Result = ' 000004,894. '
Format String =    '<,<<<,<<<,<<<'
Result = '4,894'
Format String = ' $*********.**'
Result = '$*****4894.00'

String = 443.334899312
Format String = '**###########'
Result = ' **              443'
Format String =    '$$$$$$$$$$.##'
Result = '           $443.33'
```

```
Format String =    '(&&,&&&&,&&&&.)'
Result = '    0000000443. '
Format String =    '<,<<<,<<<,<<<'
Result = ' 443'
Format String = ' $*********.**'
Result = '$******443.33'

String = -12344455
Format String = '**###########'
Result = ' **      12344455'
Format String =    '$$$$$$$$$$.##'
Result = '    $12344455.00'
Format String =    '(&&,&&&&,&&&&.)'
Result = '(12,344,455.)'
Format String =    '<,<<<,<<<,<<<'
Result = '12,344,455'
Format String = ' $*********.**'
Result = ' $*12344455.00'

RFMTDEC Sample Program over.
```

5.2.146 rfmtdouble() 函数

rfmtdouble() 函数使用格式化掩码来将 double 值转换为字符串。

语法

mint rfmtdouble(*dbl_val*, *fmtstring*, *outbuf*)

```
double dbl_val;
char *fmtstring;
char *outbuf;
```

dbl_val

要格式化的 **double** 数值。

fmtstring

指向包含 dbl_val 中的值的格式化掩码的字符缓冲区的指针。

outbuf

指向接收 dbl_val 中的值的格式化的字符串的字符缓冲区的指针。

用法

rfmtdouble() 函数的 *fmtstring* 参数指向数值格式化掩码，其包含描述如何格式化 double 值的字符。

当您使用 rfmtdouble() 来格式化 MONEY 值时，该函数使用 DBMONEY 环境变量指定的货币符号。如果您未指定此环境变量，则 rfmtdouble() 使用客户机语言环境定义的货币符号。缺省的语言环境 US English 定义货币符号如同您将 DBMONEY 设置为“$,.”一样。

当您使用有多字节代码集的非缺省的语言环境时，rfmtdouble() 支持格式字符串中的多字节字符。

返回代码

0

转换成功。

-1211

程序用尽内存（内存分配错误）。

-1217

格式字符串太大。

示例

demo 目录在文件 rfmtdouble.ec 中包含此样例程序。

```
/*
        * rfmtdouble.ec *
    The following program applies a series of format specifications to series of doubles
and displays the result of each format.
        */

        #include <stdio.h>
```

```
double dbls[] =
{
210203.204,
4894,
443.334899312,
-12344455,
0
};

char *formats[] =
{
"##############",
"<,<<<,<<<,<<<",
"$$$$$$$$$.##",
"(&&,&&&,&&&.)",
"$*********.**",
0
};

char result[41];

main()
{
mint x;
mint i = 0, f;

printf("RFMTDOUBLE Sample ESQL Program running.\n\n");

while(dbls[i])              /* for each number in dbls */
{
printf("Double Number = %g\n", dbls[i]);
f = 0;
while(formats[f])    /* format with each of formats[] */
```

```
{
if (x = rfmtdouble(dbls[i], formats[f], result))
{
printf("Error %d in formatting %g using %s\n",
x, dbls[i], formats[f]);
break;
}
/*
* Display each result and bump to next format (f++)
*/
result[40] = '\0';
printf("    Format String = '%s'\t", formats[f++]);
printf("\tResult = '%s'\n", result);
}
++i;                                    /* bump to next double */
printf("\n");                   /* separate result groups */
}
printf("\nRFMTDOUBLE Sample Program over.\n\n");
}
```

输出

```
RFMTDOUBLE Sample ESQL Program running.

        Double Number = 210203
        Format String = '##############'
        Result = '             210203'
        Format String = '<,<<<,<<<,<<<'
        Result = '210,203'
        Format String = '$$$$$$$$$$.##'
        Result = '      $210203.20'
        Format String = '(&&,&&&,&&&.)'
        Result = '    000210,203. '
        Format String = '$**********.**'
        Result = '$***210203.20'
```

```
  Double Number = 4894
  Format String = '#############'
  Result = '                4894'
  Format String = '<,<<<,<<<,<<<'
  Result = '4,894'
  Format String = '$$$$$$$$$$.##'
  Result = '         $4894.00'
  Format String = '(&&,&&&,&&&.)'
Result = ' 000004,894. '
  Format String = '$**********.**'
Result = '$*****4894.00'

  Double Number = 443.335
  Format String = '#############'
  Result = '                 443'
  Format String = '<,<<<,<<<,<<<'
  Result = '443'
  Format String = '$$$$$$$$$$.##'
  Result = '          $443.33'
  Format String = '(&&,&&&,&&&.)'
  Result = '    0000000443. '
  Format String = '$**********.**'
  Result = '$******443.33'

  Double Number = -1.23445e+07
  Format String = '#############'
  Result = '         12344455'
  Format String = '<,<<<,<<<,<<<'
  Result = '12,344,455'
  Format String = '$$$$$$$$$$.##'
  Result = '    $12344455.00'
  Format String = '(&&,&&&,&&&.)'
```

Result = '(12,344,455.)'

Format String = '$*********.**'

Result = '$*12344455.00'

RFMTDOUBLE Sample Program over.

5.2.147 rfmtlong() 函数

rfmtlong() 函数使用格式化掩码来将 C long 值转换为字符串。

语法

mint rfmtlong(lng_val, fmtstring, outbuf)

int4 lng_val;

char *fmtstring;

char *outbuf;

lng_val

rfmtlong() 将其转换为字符值的 **int4** 整数。

fmtstring

指向包含 lng_val 中的值的格式化掩码的字符缓冲区的指针。

outbuf

指向接收 lng_val 中的值的格式化的字符串的字符缓冲区的指针。

用法

rfmtlong() 函数的 *fmtstring* 参数指向数值的格式化掩码，其包含描述如何格式化 long integer 值的字符。

当您使用 rfmtlong() 来格式化 MONEY 值时，该函数使用 DBMONEY 环境变量指定的货币符号。如果您未设置此环境变量，则 rfmtlong() 使用客户机语言环境定义的货币符号。缺省的语言环境 US English 定义货币符号如同您将 DBMONEY 设置为 “$,.” 一样。

当您使用有多字节字符集的非缺省的语言环境时，rfmtlong() 支持格式字符串中的多字节字符。

返回代码

0

转换成功。

-1211

程序用尽内存（内存分配错误）。

-1217

格式字符串太大。

示例

demo 目录在文件 rfmtlong.ec 中包含此样例程序。

```
/*
        * rfmtlong.ec *
The following program applies a series of format specifications to a series of longs and
displays the result of each format.
        */

        #include <stdio.h>

        long lngs[] =
        {
        21020304,
        334899312,
        -334899312,
        -12344455,
        0
        };

        char *formats[] =
        {
        "################",
        "$$$$$$$$$$$$.##",
        "(&,&&&,&&&,&&&.)",
        "<<<<,<<<,<<<,<<<",
        "$**********.**",
```

```
0
};

char result[41];

main()
{
mint x;
mint s = 0, f;

printf("RFMTLONG Sample ESQL Program running.\n\n");

while(lngs[s])              /* for each long in lngs[] */
{
printf("Long Number = %d\n", lngs[s]);
f = 0;
while(formats[f]) /* format with each of formats[] */
{
if (x = rfmtlong(lngs[s], formats[f], result))
{
printf("Error %d in formatting %d using %s.\n",
x, lngs[s], formats[f]);
break;
}
/*
* Display result and bump to next format (f++)
*/
result[40] = '\0';
printf("    Format String = '%s'\t", formats[f++]);
printf("\tResult = '%s'\n",    result);
}
++s;                            /* bump to next long */
printf("\n");          /* separate display groups */
```

```
    }

    printf("\nRFMTLONG Sample Program over.\n\n");
    }
```

输出

```
RFMTLONG ESQL Sample Program running.

        Long Number = 21020304
        Format String = '################'
        Result = '              21020304'
        Format String = '$$$$$$$$$$$$$.##'
        Result = '       $21020304.00'
        Format String = '(&,&&&,&&&,&&&.)'
        Result = ' 00021,020,304. '
        Format String = '<<<<,<<<,<<<,<<<'
        Result = '21,020,304'
        Format String = '$**********.**'
        Result = '$****21020304.00'

        Long Number = 334899312
        Format String = '################'
        Result = '             334899312'
        Format String = '$$$$$$$$$$$$$.##'
         Result = '     $334899312.00'
        Format String = '(&,&&&,&&&,&&&.)'
        Result = ' 00334,899,312. '
        Format String = '<<<<,<<<,<<<,<<<'
        Result = '334,899,312'
        Format String = '$**********.**'
        Result = '$***334899312.00'

        Long Number = -334899312
        Format String = '################'
```

```
Result = '            334899312'
Format String = '$$$$$$$$$$$$$.##'
Result = '      $334899312.00'
Format String = '(&,&&&,&&&,&&&.)'
Result = '(00334,899,312.)'
Format String = '<<<<,<<<,<<<,<<<'
Result = '334,899,312'
Format String = '$**********.**'
Result = '$***334899312.00'

Long Number = -12344455
Format String = '################'
Result = '              12344455'
Format String = '$$$$$$$$$$$$$.##'
Result = '       $12344455.00'
Format String = '(&,&&&,&&&,&&&.)'
Result = '(00012,344,455.)'
Format String = '<<<<,<<<,<<<,<<<'
Result = '12,344,455'
Format String = '$**********.**'
Result = ' $****12344455.00'

RFMTLONG Sample Program over.
```

5.2.148 rgetlmsg() 函数

对于特定于 GBase 8s 的给定错误编号，rgetlmsg() 函数检索相应的错误消息。rgetlmsg() 函数允许在 long 整数范围内的错误编号。

语法

```
mint rgetlmsg(msgnum, msgstr, lenmsgstr, msglen)
        int4    msgnum;
        char *msgstr;
        mint lenmsgstr;
        mint *msglen;
```

msgnum

错误编号。该四字节参数提供特定于 GBase 8s 的错误编号的完整范围。

msgstr

指向接收消息字符串的缓冲区（输出缓冲区）的指针。

lenmsgstr

msgstr 输出缓冲区的大小。使得此值为您期望检索的最大消息的大小。

msglen

指向包含 rgetlmsg() 返回的消息的实际大小的 **mint** 的指针。

用法

msgnum 错误编号通常是 **SQLCODE**（或 **sqlca.sqlcode**）的值。您还可检索 ISAM 错误的消息文本（在 **sqlca.sqlerrd[1]** 中）。对于错误消息文本，rgetlmsg() 函数使用 GBase 8s 错误消息文件（在 $GBASEDBTDIR/msg 目录中）。

rgetlmsg() 函数在第四个参数 *msglen* 中返回您请求的消息的实际大小。您可使用此值来调整消息区域的大小，如果它太小的话。如果返回的消息比您提供的缓冲区还长，则该函数截断该消息。您还可使用 *msglen* 值来仅显示包含错误文本的 *msgstr* 消息缓冲区的一部分。

返回代码

0

转换成功。

-1227

未找到消息文件。

-1228

在消息文件中未找到消息编号。

-1231

在消息文件中不可搜寻。

-1232

消息缓冲区太小。

示例

此样例程序在 demo 目录中的 rgetlmsg.ec 文件中。

```
/*
        * rgetlmsg.ec *
        *
* The following program demonstrates the usage of rgetlmsg() function.
        * It displays an error message after trying to create a table that already exists.
        */
        EXEC SQL include sqlca; /* this include is optional */

        main()
        {
        mint msg_len;
        char errmsg[400];

        printf("\nRGETLMSG Sample ESQL Program running.\n");
        EXEC SQL connect to 'stores7';

        EXEC SQL create table customer (name char(20));

        if(SQLCODE != 0)
        {
        rgetlmsg(SQLCODE, errmsg, sizeof(errmsg), &msg_len);
        printf("\nError %d: ", SQLCODE);
        printf(errmsg, sqlca.sqlerrm);
        }
        printf("\nRGETLMSG Sample Program over.\n\n");
        }
```

此示例使用 **sqlca.sqlerrm** 中的错误消息参数来显示表的名称。**sqlca.sqlerrm** 的此种用法是有效的，因为该错误消息包含 printf() 认识的格式参数。如果错误消息不包含该格式参数，则不会导致错误。

输出

RGETLMSG Sample ESQL Program running.

Error -310: Table (gbasedbt.customer) already exists in database.

RGETLMSG Sample Program over.

5.2.149 rgetmsg() 函数

对于特定于 GBase 8s 的给定错误编号， rgetmsg() 函数检索错误消息文本。rgetmsg() 函数可处理 short 错误编号，因此，可仅处理 -32768 - +32767 范围内的错误编号。为此，请在所有新的 GBase 8s ESQL/C 代码中使用 rgetlmsg() 函数。

语法

```
mint rgetmsg(msgnum, msgstr, lenmsgstr)
        mint    msgnum;
        char *msgstr;
        mint lenmsgstr;
```

msgnum

错误编号。该二字节参数限制错误编号为 -32768 - +32767。

msgstr

指向接收消息字符串的缓冲区（输出缓冲区）的指针。

lenmsgstr

msgstr 输出缓冲区的大小。使得此值为您期望检索的最大消息的大小。

用法

通常，**SQLCODE**（**sqlca.sqlcode**）包含错误编号。您还可检索 ISAM 错误的消息文本（在 **sqlca.sqlerrd[1]** 中）。对于错误消息文本，rgetmsg() 函数使用 GBase 8s 错误消息文件（在 $GBASEDBTDIR/msg 目录中）。如果该消息比您提供的缓冲区的大小更长，则该函数截断消息来适应它。

重要： 为了与较早的版本相兼容，GBase 8s ESQL/C 支持 rgetmsg() 函数。有些 GBase 8s 错误编号当前超出 **short** 整数、**msgnum** 支持的范围。推荐支持 **long** 整数作为错误编号的 rgetlmsg() 函数，而不是 rgetmsg()。

如果您的程序直接地传递 **SQLCODE** 变量中的值（或 **sqlca.sqlcode**）作为 *msgnum*，则请强制转型 **SQLCODE** 值作为 **short** 数据类型。rgetmsg() 的 *msgnum* 参数有 **short** 数据类型，而 **SQLCODE** 值有 **long** 数据类型。

返回代码

0

转换成功。

-1227

未找到消息文件。

-1228

在消息文件中未找到消息编号。

-1231

在消息文件内不可搜寻。

-1232

消息缓冲区太小。

示例

此程序在 demo 目录中的 rgetmsg.ec 文件中。

```
/*
        * rgetmsg.ec *
        *
        * The following program demonstrates the usage of the rgetmsg() function.
        * It displays an error message after trying to create a table that already    exists.
        */
        EXEC SQL include sqlca; /* this include is optional */

        main()
        {
        char errmsg[400];

        printf("\nRGETMSG Sample ESQL Program running.\n\n");
        EXEC SQL connect to 'stores7';

        EXEC SQL create table customer (name char(20));
        if(SQLCODE != 0)
```

```
{
rgetmsg((short)SQLCODE, errmsg, sizeof(errmsg));
printf("\nError %d: ", SQLCODE);
printf(errmsg, sqlca.sqlerrm);
}
printf("\nRGETMSG Sample Program over.\n\n");
}
```

输出

```
RGETMSG Sample ESQL Program running.
    Error -310: Table (gbasedbt.customer) already exists in database.
    RGETMSG Sample Program over.
```

5.2.150 risnull() 函数

risnull() 函数检查 C 或 GBase 8s ESQL/C 变量是否包含空值。

语法

```
mint risnull(type; ptrvar)
        mint type;
        char *ptrvar;
```

type

对应于 C 或 GBase 8s ESQL/C 变量的整数。此 *type* 可为除了 **var binary** 或 **lvarchar** 指针变量之外的任何数据类型。

ptrvar

指向 C 或 GBase 8s ESQL/C 变量的指针。

用法

risnull() 函数确定除了 **var binary** 和 **lvarchar** 指针变量之外的所有数据类型的 GBase 8s ESQL/C 变量是否包含空值。要确定 **var binary** 或 **lvarchar** 指针主变量是否包含空，则请使用 ifx_var_isnull() 宏。

返回代码

1

变量确包含空值。

0

变量不包含空值。

示例

此样例程序在 demo 目录中的 risnull.ec 文件中。

```
/*
 * risnull.ec *

  This program checks the paid_date column of the orders table for NULL to determine whether an order has been paid.
                */

#include <stdio.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY              1
#define NOWARNNOTIFY    0

main()
{
char ans;
nt4 ret, exp_chk();

EXEC SQL BEGIN DECLARE SECTION;
int4 order_num;

mint order_date, ship_date, paid_date;
EXEC SQL END DECLARE SECTION;

printf("RISNULL Sample ESQL Program running.\n\n");
EXEC SQL connect to 'stores7';
/* open stores7 database*/
   exp_chk("CONNECT TO stores7", NOWARNNOTIFY)
```

```
  EXEC SQL declare c cursor for
select order_num, order_date, ship_date, paid_date from orders;
EXEC SQL open c;
if(exp_chk("OPEN c", WARNNOTIFY) == 1)    /* Found warnings */
exit(1);
printf("\n Order#\tPaid?\n");              /* print column hdgs */
while(1)
{
  EXEC SQL fetch c into :order_num,
  :order_date, :ship_date, :paid_date;
  if ((ret = exp_chk("FETCH c")) == 100)      /* if end of rows */
    break;                /* terminate loop */
  if(ret < 0)
    exit(1);
  printf("%5d\t", order_num);
  if (risnull(CDATETYPE, (char *)&paid_date))
  /* is price NULL ? */
  printf("NO\n");
  else
  printf("Yes\n");
  }
  printf("\nRISNULL Sample Program over.\n\n");
    }
```

/* The exp_chk() file contains the exception handling functions to check the SQLSTATE status variable to see if an error has occurred following an SQL statement. If a warning or an error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and

prints the detail for each exception that is returned.

*/

```
EXEC SQL include exp_chk.ec
```

输出

```
RISNULL Sample ESQL Program running.

Order#      Paid?
1001        Yes
1002        Yes
1003        Yes
1004        NO
1005        Yes
1006        NO
1007        NO
1008        Yes
1009        Yes
1010        Yes
1011        Yes
1012        NO
1013        Yes
1014        Yes
1015        Yes
1016        NO
1017        NO
1018        Yes
1019        Yes
1020        Yes
1021        Yes
1022        Yes
1023        Yes

RISNULL Sample Program over.
```

5.2.151 rjulmdy() 函数

rjulmdy() 函数创建一个三个 short 整数值的数组，其表示来自内部 DATE 值的月份、日子和年份。

语法

mint rjulmdy(*jdate*, *mdy*)

int4 jdate;

int2 mdy[3];

jdate

该日期的内部表示。

mdy

short 整数的数值，其中 *mdy[0]* 为月份（1 - 12），*mdy[1]* 为日子（1 - 31），*mdy[2]* 为年份（1 - 9999）。

返回代码

0

操作成功。

< 0

操作失败。

-1210

无法将内部的日期转换为字符串格式。

示例

demo 目录在 rjulmdy.ec 文件中包含此样例程序。

```
/*
        * rjulmdy.ec *

The following program accepts a date entered from the console and converts it to an array
of three short integers that contain the month, day, and year.
        */

        #include <stdio.h>

        main()
        {
        int4 i_date;
        short mdy_array[3];
```

```
        mint errnum;
        char date[20];
        mint x;

        static char fmtstr[9] = "mmddyyyy";

        printf("RJULMDY Sample ESQL Program running.\n\n");

        /* Allow user to enter a date */
printf("Enter a date as a single string, month.day.year\n");
        gets(date);

        printf("\nThe date string is %s.\n", date);

        /* Put entered date in internal format */
        if (x = rdefmtdate(&i_date, fmtstr, date))
        printf("Error %d on rdefmtdate conversion\n", x);
        else
        {

        /* Convert from internal format to MDY array */
        if ((errnum = rjulmdy(i_date, mdy_array)) == 0)
        {
printf("\tThe month component is: %d\n", mdy_array[0]);
printf("\tThe    day    component is: %d\n", mdy_array[1]);
printf("\tThe year    component is: %d\n", mdy_array[2]);
        }
        else
        printf("rjulmdy() call failed with error %d", errnum);
        }

        printf("\nRJULMDY Sample Program over.\n\n");
        }
```

输出

RJULMDY Sample ESQL Program running.

Enter a date as a single string, month.day.year

10.12.07

The date string is 10.12.07.

The month component is: 10

The day component is: 12

The year component is: 2007

RJULMDY Sample Program over.

5.2.152 rleapyear() 函数

当传递给 rleapyear() 函数的是闰年时，它返回 1（TRUE），当不是时，它返回 0（FALSE）。

语法

mint rleapyear(*year*)

mint year;

year

整数。

用法

参数 *year* 必须是日期的年份组件，而不是该日期本身。您必须必须以完整的形式（2007），而不是缩写的形式（07）来表达 *year*。

返回代码

1

该年份为闰年。

0

该年份不是闰年。

示例

demo 目录在 rleapyear.ec 文件中包含此样例程序。

```
/*
        * rleapyear.ec *
```

The following program accepts a date entered from the console and stores this date into an int4, which stores the date in an internal format. It then converts the internal format into an array of three short integers that contain the month, day, and year portions of the date. It then tests the year value to see if the year is a leap year.

```
        */

        #include <stdio.h>

        main()
        {
        int4 i_date;
        mint errnum;
        short mdy_array[3];
        char date[20];
        mint x;

        static char fmtstr[9] = "mmddyyyy";

        printf("RLEAPYEAR Sample Program running.\n\n");

        /* Allow user to enter a date */
    printf("Enter a date as a single string, month.day.year\n");
        gets(date);

        printf("\nThe date string is %s.\n", date);

        /* Put entered date in internal format */
        if (x = rdefmtdate(&i_date, fmtstr, date))
        printf("Error %d on rdefmtdate conversion\n", x);
        else
```

```
{

/* Convert internal format into a MDY array */
if ((errnum = rjulmdy(i_date, mdy_array)) == 0)
{
/* Check if it is a leap year */
if (rleapyear(mdy_array[2]))
printf("%d is a leap year\n", mdy_array[2]);
else
printf("%d is not a leap year\n", mdy_array[2]);
}
else
printf("rjulmdy() call failed with error %d", errnum);
}

printf("\nRLEAPYEAR Sample Program over.\n\n");
}
```

输出

```
RLEAPYEAR Sample ESQL Program running.

Enter a date as a single string, month.day.year
10.12.07

The date string is 10.12.07.
2007 is not a leap year

RLEAPYEAR Sample Program over.
```

5.2.153 rmdyjul() 函数

rmdyjul() 函数从表示月份、日子和年份的三个 short 整数值的数组创建内部的 DATE。

语法

mint rmdyjul(*mdy*, *jdate*)

```
int2 mdy[3];
int4 *jdate;
```

mdy

short 整数值的数组，其中 *mdy[0]* 为月份（1 - 12），*mdy[1]* 为日子（1 - 31），*mdy[2]* 为年份（1 - 9999）。

jdate

指向接收 *mdy* 数组的内部 DATE 值的 **long** 整数的指针。

用法

您可以完全的形式（2007）或缩写的形式（07）来表达年份。

返回代码

0

操作成功。

-1204

mdy[2] 变量包含无效的年份。

-1205

mdy[0] 变量包含无效的月份。

-1206

mdy[1] 变量包含无效的日子。

示例

demo 目录在 rmdyjul.ec 文件中包含此样例程序。

```
/*
        * rmdyjul.ec *

    This program converts an array of short integers containing values for month, day and
year into an integer that stores the date in    internal format.
        */

        #include <stdio.h>
```

```
main()
{
int4 i_date;
mint errnum;
static    short mdy_array[3] = { 12, 21, 2007 };
char str_date[15];

printf("RMDYJUL Sample ESQL Program running.\n\n");

/* Convert MDY array into internal format */
if ((errnum = rmdyjul(mdy_array, &i_date)) == 0)
{
rfmtdate(i_date, "mmm dd yyyy", str_date);
printf("Date '%s' converted to internal format\n", str_date);
}
else
printf("rmdyjul() call failed with errnum = %d\n", errnum);

printf("\nRMDYJUL Sample Program over.\n\n");
}
```

输出

```
RMDYJUL Sample ESQL Program running.

    Date 'Dec 21 2007' converted to internal format

    RMDYJUL Sample Program over.
```

5.2.154 rsetnull() 函数

rsetnull() 函数将 C 变量设置为对应于数据库空值的值。

语法

```
mint rsetnull(type, ptrvar)
    mint type;
```

char *ptrvar;

type

对应于 C 或 GBase 8s ESQL/C 变量的数据类型的 **mint**。此 *type* 可为除了 **var binary** 或 **lvarchar** 指针变量之外的任何数据类型。

ptrvar

指向 C 或 GBase 8s ESQL/C 变量的指针。

用法

rsetnull() 函数将除了 **var binary** 和 **lvarchar** 指针主变量之外的任何数据类型的 GBase 8s ESQL/C 变量设置为空。要将 **var binary** 或 **lvarchar** 指针主变量设置为空，请使用 ifx_var_setnull() 宏。

示例

此样例程序在 demo 目录中的 rsetnull.ec 文件中。

```
/*
        * rsetnull.ec *

 This program fetches rows from the stock table for a chosen manufacturer and allows the
user to set the unit_price to NULL.
        */

        #include <stdio.h>
        #include <ctype.h>
        EXEC SQL include decimal;
        EXEC SQL include sqltypes;

        #define WARNNOTIFY          1
        #define NOWARNNOTIFY        0

        #define LCASE(c) (isupper(c) ? tolower(c) : (c))

        char format[] = "($$,$$$,$$$.&&)";

        main()
```

```
        {
        char decdsply[20];
        char ans;
        int4 ret, exp_chk();

        EXEC SQL BEGIN DECLARE SECTION;
        short stock_num;
        char description[16];
        dec_t unit_price;
        char manu_code[4];
        EXEC SQL END DECLARE SECTION;

printf("RSETNULL Sample ESQL Program running.\n\n");
EXEC SQL connect to 'stores7';          /* connect to stores7 */
exp_chk("Connect to stores7", NOWARNNOTIFY);

 printf("This program selects all rows for a given manufacturer\n");
 printf("from the stock table and allows you to set the unit_price\n");
printf("\nTo begin, enter a manufacturer code - for example: 'HSK'\n");
printf("\nEnter Manufacturer code: ");
 /* prompt for mfr. code */
 gets(manu_code);                /* get mfr. code */
 EXEC SQL declare upcurs cursor for          /* declare cursor */
        select stock_num, description, unit_price from stock
        where manu_code = :manu_code
        for update of unit_price;
 rupshift(manu_code);            /* Make mfr code upper case */
 EXEC SQL open upcurs;    /* open select cursor */
        if(exp_chk("Open cursor", WARNNOTIFY) == 1)
        exit(1);

        /*
        * Display Column Headings
```

```
                */
                printf("\nStock # \tDescription \t\tUnit Price");
                while(1)
                {
                /* get a row */
    EXEC SQL fetch upcurs into :stock_num, :description, :unit_price;
                if ((ret = exp_chk("fetch", WARNNOTIFY)) == 100)
                /* if end of rows */
                break;
                if(ret == 1)
                exit(1);
                if(risnull(CDECIMALTYPE, (char *) &unit_price))
                  /* unit_price NULL? */
                continue;                       /* skip to next row */
                rfmtdec(&unit_price, format, decdsply);
                  /* format unit_price */
                /* display item */
printf("\n\t%d\t%15s\t%s", stock_num, description, decdsply);
                ans = ' ';
                /* Set unit_price to NULL? y(es) or n(o) */
                while((ans = LCASE(ans)) != 'y' && ans != 'n')
                {
                printf("\n. . . Set unit_price to NULL ? (y/n) ");
                scanf("%1s", &ans);
                }
                if (ans == 'y')        /* if yes, NULL to unit_price */
                {
                rsetnull(CDECIMALTYPE, (char *) &unit_price);
                EXEC SQL update stock set unit_price = :unit_price
                where current of upcurs; /* and update current row */
                if(exp_chk("UPDATE", WARNNOTIFY) == 1)
                exit(1);
                }
```

```
}
printf("\nRSETNULL Sample Program over.\n\n");
}

/*
*  The exp_chk() file contains the exception handling functions to check the
SQLSTATE status variable to see if an error has occurred  following an SQL statement. If a
warning or an error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and
prints the detail for each exception that is returned.
*/

EXEC SQL include exp_chk.ec
```

输出

```
RSETNULL Sample ESQL Program running.

This program selects all rows for a given manufacturer
from the stock table and allows you to set the unit_price
to NULL.

To begin, enter a manufacturer code - for example: 'HSK'

Enter Manufacturer code: HSK

Stock #          Description          Unit Price
1          baseball gloves          $800.00
. . . Set unit_price to NULL ? (y/n) n

3          baseball bat          $240.00
. . . Set unit_price to NULL ? (y/n) y

4          football          $960.00
. . . Set unit_price to NULL ? (y/n) n
```

```
        110             helmet                  $600.00
        . . . Set unit_price to NULL ? (y/n) y

        RSETNULL Sample Program over.
```

5.2.155 rstod() 函数

rstod() 函数将以空终止的字符串转换为 double 值。

语法

```
mint rstod(string, double_val)
        char *string;
        double *double_val;
```

string

指向以空终止的字符串的指针。

double_val

指向保存转换了的值的 **double** 值的指针。

用法

=0

转换成功。

!=0

转换失败。

示例

此样例程序在 demo 目录中的 rstod.ec 文件中。

```
/*
        * rstod.ec *

    The following program tries to convert three strings to doubles. It displays the result of each
attempt.
        */
```

```
#include <stdio.h>

main()
{
mint errnum;
char *string1 = "1234567887654321";
char *string2 = "12345678.87654321";
char *string3 = "zzzzzzzzzzzzzzz";
double d;

printf("RSTOD Sample ESQL Program running.\n\n");

printf("Converting String 1: %s\n", string1);
if ((errnum = rstod(string1, &d)) == 0)
printf("\tResult = %f\n\n", d);
else
  printf("\tError %d in conversion of string 1\n\n", errnum);

printf("Converting String 2: %s\n", string2);
if ((errnum = rstod(string2, &d)) == 0)
printf("\tResult = %.8f\n\n", d);
else
printf("\tError %d in conversion of string 2\n\n", errnum);

printf("Converting String 3: %s\n", string3);
if ((errnum = rstod(string3, &d)) == 0)
printf("\tResult = %.8f\n\n", d);
else
printf("\tError %d in conversion of string 3\n\n", errnum);

printf("\nRSTOD Sample Program over.\n\n");
}
```

输出

RSTOD Sample ESQL Program running.

Converting String 1: 123456788764321

Result = 1234567887654321.000000

Converting String 2: 12345678.87654321

Result = 12345678.87654321

Converting String 3: zzzzzzzzzzzzzzz

Error -1213 in conversion of string 3

RSTOD Sample Program over.

5.2.156 rstoi() 函数

rstoi() 函数将以空终止的字符串转换为 short integer 值。

语法

mint rstoi(*string*, *ival*)

char *string;

mint *ival;

string

指向以空终止的字符串的指针。

ival

指向保存转换了的值的 **mint** 值的指针。

用法

值的合法范围为 -32767 - 32767。值 -32768 不是有效的，因为此值为指示空的保留值。

如果 *string* 对应于空整数，则 *ival* 指向 SMALLINT 空的表示。要转换对应于 long integer 的字符串，请使用 rstol()。该操作的失败可导致错误的数据表示。

返回代码

=0

转换成功。

!=0

转换失败。

示例

此样例程序在 demo 目录中的 rstoi.ec 文件中。

```
/*
        * rstoi.ec *
    The following program tries to convert three strings to integers. It displays the result of
each conversion.
        */

        #include <stdio.h>

        EXEC SQL include sqltypes;

        main()
        {
        mint err;
        mint i;
        short s;

        printf("RSTOI Sample ESQL Program running.\n\n");

        i = 0;
        printf("Converting String 'abc':\n");
        if((err = rstoi("abc", &i)) == 0)
        printf("\tResult = %d\n\n", i);
        else
```

```
printf("\tError %d in conversion of string #1\n\n", err);

    i = 0;
    printf("Converting String '32766':\n");
    if((err = rstoi("32766", &i)) == 0)
    printf("\tResult = %d\n\n", i);
    else
  printf("\tError %d in conversion of string #2\n\n", err);

    i = 0;
    printf("Converting String ":\n");
    if((err = rstoi("", &i)) == 0)
    {
    s = i;        /* assign to a SHORT variable */
    if (risnull(CSHORTTYPE, (char *) &s))
   /* and then test for NULL */
    printf("\tResult = NULL\n\n");
    else
    printf("\tResult = %d\n\n", i);
    }
    else
  printf("\tError %d in conversion of string #3\n\n", err);

    printf("\nRSTOI Sample Program over.\n\n");
    }
```

输出

```
RSTOI Sample ESQL Program running.

    Converting String 'abc':
    Error -1213 in conversion of string #1

    Converting String '32766':
    Result = 32766
```

Converting String ":

Result = NULL

RSTOI Sample Program over.

5.2.157 rstol() 函数

rstol() 函数将以空终止的字符串转换为 long integer 值。

语法

mint rstol(*string*, *long_int*)

char *string;

mlong *long_int;

string

指向以空终止的字符串的指针。

long_int

指向保存转换了的值的 **mlong** 值的指针。

用法

值的合法范围为 -2,147,483,647 - 2,147,483,647。值 -2,147,483,648 不是有效的，因为此值为指示空的保留值。

返回代码

=0

转换成功。

!=0

转换失败。

示例

此样例程序在 demo 目录中的 rstol.ec 文件中。

/*

```
* rstol.ec *
```

The following program tries to convert three strings to longs. It displays the result of each attempt.

```
*/

#include <stdio.h>

EXEC SQL include sqltypes;

main()
{
mint err;
mlong l;

printf("RSTOL Sample ESQL Program running.\n\n");

l = 0;
printf("Converting String 'abc':\n");
if((err = rstol("abc", &l)) == 0)
printf("\tResult = %ld\n\n", l);
else
printf("\tError %d in conversion of string #1\n\n", err);

l = 0;
printf("Converting String '2147483646':\n");
if((err = rstol("2147483646", &l)) == 0)
printf("\tResult = %ld\n\n", l);
else
printf("\tError %d in conversion of string #2\n\n", err);

l = 0;
printf("Converting String ":\n");
```

```
    if((err = rstol("", &l)) == 0)
    {
    if(risnull(CLONGTYPE, (char *) &l))
    printf("\tResult = NULL\n\n", l);
    else
    printf("\tResult = %ld\n\n", l);
    }
    else
  printf("\tError %d in conversion of string #3\n\n", err);

    printf("\nRSTOL Sample Program over.\n\n");
    }
```

输出

```
RSTOL Sample ESQL Program running.

        Converting String 'abc':
        Error -1213 in conversion of string #1

        Converting String '2147483646':
        Result = 2147483646

        Converting String '':
        Result = NULL
         RSTOL Sample Program over.
```

5.2.158 rstrdate() 函数

rstrdate() 函数将字符串转换为内部的 DATE。

语法

```
mint rstrdate(inbuf, jdate)
        char *inbuf;
        int4 *jdate;
```

inbuf

指向包含要转换的日期的字符串的指针。

jdate

指向接收 *inbuf* 字符串的内部 DATE 值的 **int4** 整数的指针。

用法

对于缺省的语言环境 US English， rstrdate() 函数以下列优先顺序来确定如何格式化该字符串：

DBDATE 环境变量指定的格式（如果设置 DBDATE 的话）。

GL_DATE 环境变量指定的格式（如果设置 GL_DATE 的话）。

缺省的日期形式：mm/dd/yyyy。您可使用任何非数值的字符作为月份、日子与年份之间的分隔符。您可将年份表达为四位数字（2007），或为两位数值（07）。

当您使用非缺省的语言环境，且未设置 DBDATE 或 GL_DATE 环境变量时，rstrdate() 使用客户机语言环境定义的日期终端用户格式。

当您在日期字符串中使用两位数字年份时，rstrdate() 函数使用 DBCENTURY 环境变量的值来确定要使用哪个世纪。如果您未设置 DBCENTURY，则 **rstrdate()** 为两位年份假定第 20 世纪。

返回代码

0

转换成功。

< 0

转换失败。

-1204

inbuf 参数指定无效的年份。

-1205

inbuf 参数指定无效的月份。

-1206

inbuf 参数指定无效的日子。

-1212

数据转换格式必须包含月份、日子或年份组件。DBDATE 指定数据转换格式。

-1218

由 *inbuf* 参数指定的日期未正确地表示日期。

示例

demo 目录在 rstrdate.ec 文件中包含此样例程序。

```
/*
        * rstrdate.ec *
        The following program converts a character string
        in "mmddyyyy" format to an internal date format.
        */

        #include <stdio.h>

        main()
        {
        int4 i_date;
        mint errnum;
        char str_date[15];

        printf("RSTRDATE Sample ESQL Program running.\n\n");

        /* Convert Sept. 6th, 2007 into i_date */
        if ((errnum = rstrdate("9.6.2007", &i_date)) == 0)
        {

        rfmtdate(i_date, "mmm dd yyyy", str_date);
    printf("Date '%s' converted to internal format\n" str_date);
        }
        else
      printf("rstrdate() call failed with error %d\n", errnum);

        printf("\nRSTRDATE Sample Program over.\n\n");
        }
```

输出

RSTRDATE Sample ESQL Program running.

Date 'Sep 06 2007' converted to internal format

RSTRDATE Sample Program over.

5.2.159 rtoday() 函数

rtoday() 函数返回系统日期作为 long integer 值。

语法

void rtoday(*today*)

int4 *today;

today

指向接收内部 DATE 的 **int4** 值的指针。

用法

rtoday() 函数取得客户机计算机上的系统日期，而不是服务器计算机上的。

示例

demo 目录在 rtoday.ec 文件中包含此样例程序。

```
/*
        * rtoday.ec *
   The following program obtains today's date from the system,
  converts it to ASCII using rdatestr(), and displays the result.
        */

        #include <stdio.h>

        main()
        {
        mint errnum;
```

```
        char today_date[20];
        int4 i_date;

        printf("RTODAY Sample ESQL Program running.\n\n");

        /* Get today's date in the internal format */
        rtoday(&i_date);
/* Convert date from internal format into a mm/dd/yyyy string */
        if ((errnum = rdatestr(i_date, today_date)) == 0)
        printf("\n\tToday's date is %s.\n", today_date);
        else
            printf("\n\tError %d in converting date to mm/dd/yyyy\n", errnum);

        printf("\nRTODAY Sample Program over.\n\n");
        }
```

输出

```
RTODAY Sample ESQL Program running.

        Today's date is 09/16/2007.

        RTODAY Sample Program over.
```

5.2.160 rtypalign() 函数

rtypalign() 函数为指定类型的变量返回下一恰当的边界。

语法

32 位

```
mint rtypalign(pos, type)
    mint pos;
    mint type;
```

64 位

mlong rtypalign(*pos*, *type*)

mlong pos;

mint type;

pos

缓冲区中的当前位置。

type

对应于 C 或 GBase 8s ESQL/C 变量的数据类型的整数。此 *type* 可为除了下列之外的任何数据类型：

var binary

CFIXBINTYPE

CVARBINTYPE

SQLUDTVAR

SQLUDTFIXED

用法

当您使用 **sqlda** 结构来动态地将数据访存至缓冲区内时，rtypalign() 和 rtypmsize() 函数是有用的。在许多硬件平台上，整数及其他数值数据类型必须在工作边界上开始。C 语言内存分配例程为包括结构在内的任何数据类型分配恰当地对齐的内存。然而，对于该结构的组成组件，这些例程不执行对齐。程序员负责以诸如 rtypalign() 和 rtypmsize() 这样的函数来执行对齐。这些函数提供独立于机器地列数据存储。

在确定列信息的 DESCRIBE 语句之后，GBase 8s ESQL/C 在 **sqlda.sqlvar->sqltype** 中存储 *type* 的值。

在 unload.ec 演示程序中，您可看到 rtypalign() 函数的应用。

返回代码

>0

返回值是 *type* 数据类型的变量的下一恰当边界的偏移量。

示例

此样例程序在 demo 目录中的 rtypalign.ec 文件中。

```
/*
    * rtypalign.ec *

    The following program prepares a select on all columns of the orders table and then
calculates the proper alignment for each column in a buffer.
*/

#include <decimal.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY          1
#define NOWARNNOTIFY        0

main()
{
    mint i, pos;
    int4 ret, exp_chk();
    struct sqlda *sql_desc;
    struct sqlvar_struct *col;

    printf("RTYPALIGN Sample ESQL Program running.\n\n");

    EXEC SQL connect to 'stores7';    /* open stores7 database */
    exp_chk("Connect to", NOWARNNOTIFY);

    EXEC SQL prepare query_1 from "select * from orders";
    /* prepare select */
    if(exp_chk("Prepare", WARNNOTIFY) == 1)
       exit(1);

    EXEC SQL describe query_1 into sql_desc;
    /* initialize sqlda */
```

```
    if(exp_chk("Describe", WARNNOTIFY) == 1)
        exit(1);

    col = sql_desc->sqlvar;
     printf("\n\ttype\t\tlen\tnext\taligned\n");
    /* display column hdgs. */
    printf("\t\t\t\tposn\tposn\n\n");
    /*
    * For each column in the orders table
     */
    i = 0;
    pos = 0;
    while(i++ < sql_desc->sqld)
        {
        /* Modify sqllen if SQL type is DECIMAL or MONEY */
        if(col->sqltype == SQLDECIMAL || col->sqltype == SQLMONEY)
            {
            col->sqllen = sizeof(dec_t);
            }
        /*
* display name of SQL type, length and un-aligned buffer position
          */
        printf("\t%s\t\t%d\t%d", rtypname(col->sqltype), col->sqllen, pos);

        pos = rtypalign(pos, col->sqltype);
        /* align pos. for type */
        printf("\t%d\n", pos);

        pos += col->sqllen;             /* set next position */
        ++col;                          /* bump to next column */
    }
    printf("\nRTYPALIGN Sample Program over.\n\n");
}
```

```
/*
 *  The exp_chk() file contains the exception handling functions to check the SQLSTATE
status variable to see if an error has occurred following an SQL statement. If a warning or an
error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and prints the detail
for each exception that is returned.
 */

EXEC SQL include exp_chk.ec
```

输出

RTYPALIGN Sample ESQL Program running.

type	len	next posn	aligned posn
serial	4	0	0
date	4	4	4
integer	4	8	8
char	40	12	12
char	1	52	52
char	10	53	53
date	4	63	64
decimal	22	68	68
money	22	90	90
date	4	112	112

RTYPALIGN Sample Program over.

5.2.161 rtypmsize() 函数

对于指定的 GBase 8s ESQL/C 或 SQL 数据类型，rtypmsize() 函数返回您必须在内存中分配的字节数。

语法

mint rtypmsize(*sqltype*, *sqllen*)

```
        mint sqltype;
```

mint sqllen;

sqltype

GBase 8s ESQL/C 或 SQL 数据类型的整数代码。

sqllen

对于指定的数据类型，数据文件中的字节数。

用法

当您使用 **sqlda** 结构来动态地将数据访存至缓冲区时，rtypalign() 和 rtypmsize() 函数是有用的。这些函数为列数据提供独立于机器的存储。

随同 DESCRIBE 语句初始化的 **sqlda** 结构，提供 rtypmsize() 函数来使用。在 DESCRIBE 语句确定列信息之后，*sqltype* 和 *sqllen* 组件的值在每一 **sqlda.sqlvar** 结构中同名的组件中。

当 rtypmsize() 确定字符数据的大小时，请记住下列大小信息：

对于 CCHARTYPE（**char**）和 CSTRINGTYPE（**string**）， GBase 8s ESQL/C 为空终止符将字符的数目加一个字节。

对于 CFIXCHARTYPE（**fixchar**），GBase 8s ESQL/C 不添加空终止符。

在 unload.ec 演示程序中，您可看到 rtypmsize() 函数的应用。

返回代码

0

sqltype 不是有效的 SQL 类型。

>0

返回值是 *sqltype* 数据类型要求的字节数。

示例

此样例程序在 demo 目录中的 rtypmsize.ec 文件中。

```
/*
        * rtypmsize.ec *

This program prepares a select statement on all columns of the catalog table. Then it
```

displays the data type of each column and the number of bytes needed to store it in memory.

```
*/

#include <stdio.h>

EXEC SQL include sqltypes;

#define WARNNOTIFY          1
#define NOWARNNOTIFY        0

EXEC SQL BEGIN DECLARE SECTION;
char db_name[20];
EXEC SQL END DECLARE SECTION;

main(argc, argv)
int argc;
char *argv[];
{
mint i;
char db_stmnt[50];
int4 exp_chk();
struct sqlda *sql_desc;
struct sqlvar_struct *col;

printf("RTYPMSIZE Sample ESQL Program running.\n\n");

if (argc > 2)                   /* correct no. of args? */
{
printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
argv[0]);
exit(1);
}
strcpy(db_name, "stores7");
```

```
    if (argc == 2)
    strcpy(db_name, argv[1]);

    EXEC SQL connect to :db_name;
    sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
    exp_chk(db_stmnt, NOWARNNOTIFY);

    printf("Connected to '%s' database.", db_name);

    EXEC SQL prepare query_1 from 'select * from catalog';
    /* prepare select */
    if(exp_chk("Prepare", WARNNOTIFY) == 1)
    exit(1);
    EXEC SQL describe query_1 into sql_desc;
    /* setup sqlda */
    if(exp_chk("Describe", WARNNOTIFY) == 1)
    exit(1);
   /* column hdgs. */
    printf("\n\tColumn Type Size\n\n");
    /*
 * For each column in the catalog table display the column name and the number of bytes needed to store the column in memory.
    */
  for(i = 0, col = sql_desc->sqlvar; i < sql_desc->sqld; i++, col++)
    printf("\t%-20s%-8t%3d\n",
    col->sqlname, rtypname(col->sqltype),
    rtypmsize(col->sqltype, col->sqllen));

    printf("\nRTYPMSIZE Sample Program over.\n\n");
    }

    /*
    *    The exp_chk() file contains the exception handling functions to    check the
```

SQLSTATE status variable to see if an error has occurred following an SQL statement. If a warning or an error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and prints the detail for each exception that is returned.

```
*/
EXEC SQL include exp_chk.ec
```

输出

```
RTYPMSIZE Sample ESQL Program running.

    Connected to stores7 database.

    Column              Type            Size
    catalog_num         serial          4
    stock_num           smallint        s
    manu_code           char            4
    cat_descr           text            64
    cat_picture         byte            64
    cat_advert          varchar         256

    RTYPMSIZE Sample Program over.
```

5.2.162 rtypname() 函数

rtypname() 函数返回包含指定的 SQL 数据类型的名称的以空终止的字符串。

语法

```
char *rtypname(sqltype)
    mint sqltype;
```

sqltype

SQL 数据类型之一的整数代码。

rtypname() 函数将（sqltypes.h 定义的）GBase 8s SQL 数据类型的常量转换为字符串。

返回代码

rtypname() 函数返回指向包含 *sqltype* 指定的数据类型的名称的字符串的指针。如果 *sqltype* 是无效的值，则 rtypname() 返回空字符串（""）。

示例

此样例程序在 demo 目录中的 rtypname.ec 文件中。

```
/*
        * rtypname.ec *

This program displays the name and the data type of each column in the 'orders' table.
        */

        #include <stdio.h>

        EXEC SQL include sqltypes;

        #define WARNNOTIFY          1
        #define NOWARNNOTIFY        0

        main(argc, argv)
        int argc;
        char *argv[];
        {
        mint i;
        int4 err_chk();
        char db_stmnt[50];
        char *rtypname();
        struct sqlda *sql_desc;
        struct sqlvar_struct *col;

        EXEC SQL BEGIN DECLARE SECTION;
        char db_name[20];
        EXEC SQL END DECLARE SECTION;
```

```
        printf("RTYPNAME Sample ESQL Program running.\n\n");

        if (argc > 2)                   /* correct no. of args? */
        {
    printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",
        argv[0]);
        exit(1);
        }
        strcpy(db_name, "stores7");

        if (argc == 2)
        strcpy(db_name, argv[1]);

        EXEC SQL connect to :db_name;
        sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
        exp_chk(db_stmnt, NOWARNNOTIFY);

        printf("Connected to '%s' database.", db_name);
        EXEC SQL prepare query_1 from 'select * from orders';
          /* prepare select */
        if(exp_chk("Prepare", WARNNOTIFY) == 1)
        exit(1);
        /* initialize sqlda */
        EXEC SQL describe query_1 into sql_desc;
        if(exp_chk("Describe", WARNNOTIFY) == 1)
        exit(1);
        printf("\n\tColumn Name       \t\tSQL type\n\n");

        /*
        * For each column in the orders table display the column name and    the name
of the SQL data type
        */
    for (i = 0, col = sql_desc->sqlvar; i < sql_desc->sqld; i++, col++)
```

```
printf("\t%-15s\t\t%s\n",col->sqlname, rtypname(col->sqltype));

        printf("\nRTYPNAME Sample Program over.\n\n");
        }

        /*
 *  The exp_chk() file contains the exception handling functions  to check the
SQLSTATE status variable to see if an error has occurred following an SQL statement. If a
warning or an error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and
prints the detail for each exception that is returned.
        */

        EXEC SQL include exp_chk.ec
```

输出

```
RTYPNAME Sample ESQL Program running.

        Connected to stores7 database
        Column Name                SQL type

        order_num                  serial
        order_date                 date
        customer_num               integer
        ship_instruct              char
        backlog                    char
        po_num                     char
        ship_date                  date
        ship_weight                decimal
        ship_charge                money
        paid_date                  date

        RTYPNAME Sample Program over.
```

5.2.163 rtypsize() 函数

对于使用“更改数据捕获”API 的客户机，rtypsize() 函数返回数据的内部存储大小。

语法

mint rtypsize(*sqltype*, *sqllen*)

mint sqltype;

mint sqllen;

sqltype

GBase 8s ESQL/C 或 SQL 数据类型的整数代码。

sqllen

对于指定的数据类型，数据文件中的字节数。

用法

尽管类似于 rtypmsize() 函数，但 rtypsize() 函数返回内部的服务器存储长度，而不是 ESQL/C 数据长度。

随同 DESCRIBE 语句初始化的 **sqlda** 结构，提供 rtypsize() 函数来使用。在 DESCRIBE 语句确定列信息之后，*sqltype* 和 *sqllen* 组件的值在每一 **sqlda.sqlvar** 结构中同名的组件中。

在 demo 目录中的 cdcapi.ec 样例程序中，您可看到 rtypsize() 函数的应用。

返回代码

0

sqltype 不是有效的 SQL 类型。

>0

返回值是 *sqltype* 数据类型要求的字节数。

示例

下列代码片段来自 demo 目录中的 cdcapi.ec 文件，展示 rtypsize() 函数的使用。

```
sprintf(sql_stm, "select * from %s", tabname);
        $prepare select_id from $sql_stm;
        CHK_SQL_CODE(sql_stm);

        $describe select_id into sqlda;
        CHK_SQL_CODE("Describe");

        /*
        *    Save the description of the column descriptor for the table.
* We will use this later to process the insert/update/delete records
        *    for this table.
        */
        for (col = 0; col < sqlda->sqld; col++)
        {
        colsize = rtypsize(sqlda->sqlvar[col].sqltype,
        sqlda->sqlvar[col].sqllen); printStdoutAndFile("\tColumn %d is %s, type = %d,
size =%d\n", col,sqlda->sqlvar[col].sqlname,sqlda->sqlvar[col].sqltype, colsize);

  coldesc.colobj[col].coltype = sqlda->sqlvar[col].sqltype;
  coldesc.colobj[col].colsize = colsize;
  coldesc.colobj[col].colxid = sqlda->sqlvar[col].sqlxid;
  coldesc.colobj[col].colname =
  malloc(strlen(sqlda->sqlvar[col].sqlname)+1);
  strcpy(coldesc.colobj[col].colname,
    sqlda->sqlvar[col].sqlname);
        }
        coldesc.num_of_columns = col;
```

5.2.164 rtypwidth() 函数

当您将带有 SQL 数据类型的值转换为字符数据类型时，rtypwidth() 函数返回字符数据类型需要避免截断的最小字符数。

语法

mint rtypwidth(*sqltype*, *sqllen*)

mint sqltype;

mint sqllen;

sqltype

SQL 数据类型的整数代码。

sqllen

对于指定的 SQL 数据类型，在数据文件中的字节数。

用法

随同 DESCRIBE 语句初始化的 **sqlda**，提供 rtypwidth() 函数来使用。*sqltype* 和 *sqllen* 组件对应于每一 **sqlda.sqlvar** 结构中同名的组件。

返回代码

0

sqltype 不是有效的 SQL 数据类型。

>0

返回值是 *sqltype* 数据类型要求的最小字符数。

示例

此样例程序在 demo 目录中的 rtypwidth.ec 文件中。

```
/*
        * rtypwidth.ec *

  This program displays the name of each column in the 'orders' table and the number of
characters required to store the column when the data type is converted to characters.
        */

        #include <stdio.h>
```

```
#define WARNNOTIFY          1
#define NOWARNNOTIFY        0

main(argc, argv)
int argc;
char *argv[];
{
mint i, numchars;

int4 exp_chk();
char db_stmnt[50];
struct sqlda *sql_desc;
struct sqlvar_struct *col;

EXEC SQL BEGIN DECLARE SECTION;
char db_name[20];
EXEC SQL END DECLARE SECTION;

printf("RTYPWIDTH Sample ESQL Program running.\n\n");

if (argc > 2)                   /* correct no. of args? */
{
printf("\nUsage: %s [database]\nIncorrect no. of argument(s)\n",argv[0]);
        exit(1);
}
strcpy(db_name, "stores7");
if (argc == 2)
strcpy(db_name, argv[1]);

EXEC SQL connect to :db_name;
sprintf(db_stmnt, "CONNECT TO %s", argv[1]);
exp_chk(db_stmnt, NOWARNNOTIFY);
```

```
printf("Connected to %s\n", db_name);
EXEC SQL prepare query_1 from 'select * from orders';
/* prepare select */
if(exp_chk("Prepare", WARNNOTIFY) == 1)
exit(1);
EXEC SQL describe query_1 into sql_desc;
/* setup sqlda */
if(exp_chk("Describe", WARNNOTIFY) == 1)
exit(1);
printf("\n\tColumn Name        \t# chars\n");

/*
```

* For each column in orders print the column name and the minimum number of characters required to convert the SQL type to a character data type

```
*/
  for (i = 0, col = sql_desc->sqlvar; i < sql_desc->sqld; i++, col++)
{
numchars = rtypwidth(col->sqltype, col->sqllen);
printf("\t%-15s\t%d\n", col->sqlname, numchars);
}

printf("\nRTYPWIDTH Sample Program over.\n\n");
}

/*
```

The exp_chk() file contains the exception handling functions to

heck the SQLSTATE status variable to see if an error has occurred following an SQL statement. If a warning or an error has occurred, exp_chk() executes the GET DIAGNOSTICS statement and prints the detail for each exception that is returned.

```
*/
EXEC SQL include exp_chk.ec
```

输出

RTYPWIDTH Sample ESQL Program running.

```
Connected to stores7

Column Name             # chars
order_num                 11
order_date                10
customer_num              11
ship_instruct             40
backlog                    1
po_num                    10
ship_date                 10
ship_weight               10
ship_charge               9
paid_date                 10

RTYPWIDTH Sample Program over.
```

5.2.165 rupshift() 函数

rupshift() 函数将以空终止的字符串内的所有字符更改为大写字符。

语法

```
void rupshift(s)
        char *s;
```

s

指向以空终止的字符串的指针。

用法

rupshift() 函数应用当前的语言环境来确定大写和小写字母。对于缺省的语言环境 US English，rupshift() 使用 ASCII 小写字母（a-z）和大写字母（A-Z）。

如果您使用非缺省的语言环境，则 rupshift() 使用该语言环境定义的小写字母和大写字母。

示例

此示例程序在 demo 目录中的 rupshift.ec 文件中。

```
/*
        * rupshift.ec *

  The following program displays the result of rupshift() on a string of numbers, letters and punctuation.
        */

        #include <stdio.h>

        main()
        {
        static char string[] = "123abcdefghijkl;.";

        printf("RUPSHIFT Sample ESQL Program running.\n\n");

        printf("\tInput    string: %s\n", string);
        rupshift(string);
        printf("\tAfter upshift: %s\n", string);    /* Result */

        printf("\nRUPSHIFT Sample Program over.\n\n");
        }
```

输出

```
RUPSHIFT Sample ESQL Program running.

        Input    string: 123abcdefghijkl;.
        After upshift: 123ABCDEFGHIJKL;.

        RUPSHIFT Sample Program over.
```

5.2.166 SetConnect() 函数（Windows(TM)）

SetConnect() 函数仅在 Windows(TM) 环境中可用。它将该连接切换至指定的显式的连接。

重要： 对于与 Windows(TM) 应用程序的 Version 5.01 GBase 8s ESQL/C 的兼容性，GBase 8s ESQL/C 支持 SetConnect() 连接库函数。当您为 Windows(TM) 环境编写新的 GBase 8s ESQL/C 应用程序时，请使用 SQL SET CONNECTION 语句来切换至另一活动的连接。

语法

void *SetConnect (void **CnctHndl*)

CnctHndl

前面的 GetConnect() 调用已返回的连接句柄。

用法

SetConnect() 函数映射至简单的（不带有 DEFAULT 选项的）SQL SET CONNECTION 语句。SetConnect() 调用等同于下列 SQL 语句：

EXEC SQL set connection *db_connection*;

在此示例中，*db_connection* 是 GetConnect() 函数已建立的现有连接的名称。您将此 *db_connection* 名称作为参数传递给 SetConnect() 函数。它是您想要使其活动的连接的连接句柄。

如果您传递空句柄，则 SetConnect() 函数返回当前的连接句柄，且不更改当前的连接。如果当您传递空句柄时不存在当前的连接，则 SetConnect() 返回空。

例如，下列代码片段使用 SetConnect() 来将至 **acctsrvr** 数据库服务器上 **accounts** 数据库的连接（**cnctHndl2**）切换至 **mainsrvr** 数据库服务器上的 **customers** 数据库（**cnctHndl1**）：

```
void *cnctHndl1, *cnctHndl2, *prevHndl;
        ⋮
```

```
lish connection 'cnctHndl1' to customers@mainsrvr */
strcpy(InetLogin.InfxServer, "mainsrvr");
cnctHndl1 = GetConnect();
EXEC SQL database customers;
⋮
/* Establish connection 'cnctHndl2' to accounts@acctsrvr */
strcpy(InetLogin.InfxServer, "acctsrvr");
cnctHndl2 = GetConnect();
EXEC SQL database accounts;
⋮
prevHndl = SetConnect( cnctHndl1 );
```

重要： 由于 SetConnect() 函数映射至 SET CONNECTION 语句，因此，它设置 SQLCODE 和 SQLSTATE 状态代码，来指示该连接切换请求成功还是失败。此行为不同于 Windows(TM) 的 Version 5.01 GBase 8s ESQL/C 中的 SetConnect()，其中，此函数未设置 SQLCODE 和 SQLSTATE 值。

SetConnect() 函数取得连接名称的方式不同于 SET CONNECTION 语句。SetConnect() 使用存储在连接句柄中的内部生成的名称。您必须指定此句柄作为 SetConnect() 调用的参数。该 SET CONNECTION 语句使用 CONNECT 语句的 AS 子句指定的用户定义的连接名称。

重要： 由于 GetConnect() 函数映射至带有 WITH CONCURRENT TRANSACTION 子句的 CONNECT 语句，因此它允许带有打开的事务的显式的连接成为暂停的。在您的 GBase 8s ESQL/C 应用程序调用 SetConnect() 函数来切换至另一显式的连接之前，它不需要确保提交了或回滚了当前的事务。

返回代码

CnctHndl

如果该函数已返回了当前为暂停的连接的连接句柄，则 SetConnect() 调用成功。

空指针

SetConnect() 调用不成功，指示未建立显式的连接。

5.2.167 sqgetdbs() 函数

sqgetdbs() 函数返回数据库服务器可访问的数据库的名称。

语法

```
mint sqgetdbs(ret_fcnt, dbnarray, dbnsize, dbnbuffer, dbnbufsz)
        mint *ret_fcnt;
        char **dbnarray;
        mint dbnsize;
        char *dbnbuffer;
        mint dbnbufsz;
```

ret_fcnt

指向该函数返回的数据库名称的数目的指针。

dbnarray

用户定义的字符指针的数组。

dbnsize

dbnarray 用户定义的数组的大小。

dbnbuffer

指向包含该函数返回的数据库名称的用户定义的缓冲区的指针。

dbnbufsz

dbnbuffer 用户定义的缓冲区的大小。

用法

您必须将下列用户定义的数据结构提供给 sqgetdbs() 函数：

dbnbuffer 缓冲区保存 sqgetdbs() 返回的以空终止的数据库名称的名称。

dbnarray 数组保存指向该函数存储在 *dbnbuffer* 缓冲区中的数据库名称的指针。例如，*dbnarray*[0] 指向第一个数据库名称的第一个字符（在 *dbnbuffer* 中），*dbnarray*[1] 指向第二个数据库名称的第一个字符，依此类推。

如果该应用程序连接到数据库服务器，则对sqgetdbs() 函数的调用返回当前连接的数据库服务器中可用的数据库的名称。这包括用户定义的数据库和 **sysmaster** 数据库。然而，它返回（GBASEDBTSERVER 环境变量指示的）在缺省的数据库服务器中可用的数据库名称。如果您使用 DBPATH 环境变量来标识包含数据库的附加的数据库服务器，则

sqgetdbs() 也罗列在这些数据库服务器上可用的数据库。它首先罗列通过 DBPATH 可用的数据库，然后是通过 GBASEDBTSERVER 环境变量可用的数据库。

返回代码

0

成功地取得数据库名称

<0

未能取得数据库名称

示例

demo 目录中的 sqgetdbs.ec 文件中包含此样例程序。

```
/*
        * sqgetdbs.ec *

 This program lists the available databases in the database server of the current connection.
        */

        #include <stdio.h>

/* Defines used with exception-handling function: exp_chk() */
        #define WARNNOTIFY      1
        #define NOWARNNOTIFY    0

/* Defines used for user-defined data structures for sqgetdbs() */
        #define BUFFSZ          256
        #define NUM_DBNAMES     10

        main()
        {
    char db_buffer[ BUFFSZ ];   /* buffer for database names */     /* array of pointers to database names in 'db_buffer' */
        char *dbnames[ NUM_DBNAMES ];
```

```
    mint num_returned; /* number of database names returned */
                mint ret, i;

                printf("SQGETDBS Sample ESQL Program running.\n\n");

                EXEC SQL connect to default;
                exp_chk("CONNECT TO default server", NOWARNNOTIFY);
                printf("Connected to default server.\n");

                ret = sqgetdbs(&num_returned, dbnames, NUM_DBNAMES,
                db_buffer, BUFFSZ);
                if(ret < 0)
                {
                printf("Unable to obtain names of databases.\n");
                exit(1);
                }
    printf("\nNumber of database names returned = %d\n", num_returned);

                printf("Databases currently available:\n");
                for (i = 0; i < num_returned; i++)
                printf("\t%s\n", dbnames[i]);
                printf("\nSQGETDBS Sample Program over.\n\n");
                }

                /*
    * The exp_chk() file contains the exception handling functions to
    * check the SQLSTATE status variable to see if an error has occurred
    * following an SQL statement. If a warning or an error has
    * occurred, exp_chk() executes the GET DIAGNOSTICS statement and * displays the
detail for each exception that is returned.
                */
                EXEC SQL include exp_chk.ec;
```

输出

您从 **sqgetdbs** 样例程序看到的输出依赖于如何设置您的 **GBASEDBTSERVER** 和 **DBPATH** 环境变量。下列样例输出假定将 GBASEDBTSERVER 环境变量设置为 **mainserver**，且假定此数据库服务器包含称为 **stores7**、**sysmaster** 和 **tpc** 的三个数据库。此输出还假定未设置 **DBPATH** 环境变量。

```
SQGETDBS Sample ESQL Program running.

        Connected to default server.

        Number of database names returned = 3
        Databases currently available:
        stores7@mainserver
        sysmaster@mainserver
        tpc@mainserver

        SQGETDBS Sample Program over.
```

5.2.168 sqlbreak() 函数

sqlbreak() 函数发送给数据库服务器一个请求，来中断当前 SQL 请求的处理。通常，您调用此函数来中断长查询。

语法

```
mint sqlbreak();
```

用法

sqlbreak() 函数发送中断请求给当前连接的数据库服务器。当数据库服务器收到此请求时，它必须确定该 SQL 求实是否可中断。某些类型数据库操作不可中断，可在某些点中断其他的。您可中断下列 SQL 语句。

ALTER INDEX

ALTER TABLE

CREATE INDEX

CREATE TABLE

DELETE

EXECUTE PROCEDURE

INSERT

OPEN

SELECT

UPDATE

如果可中断该 SQL 请求，则数据库服务器采取下列行动：

停止当前 SQL 请求的执行

将 SQLCODE（**sqlca.sqlcode**）设置为负值（-213）

将控制返回至应用程序

当应用程序在中断了的 SQL 请求之后重新获得控制时，分配给该 SQL 语句的任何资源都保持被分配。任何打开的数据库、游标和事务都保持打开。任何系统描述符区域或 **sqlda** 结构保持被分配。应用程序负责恰当地终止该程序；它必须释放资源，并回滚当前的事务。

在数据库服务器执行 SQL 请求时，锁定该应用程序，等待来自数据库服务器的结果。要调用 sqlbreak()，您必须先建立某种解除该应用程序进程锁定的机制。两种可能的方法如下：

一旦它开始执行，就提供该应用程序用户以中断 SQL 请求的能力。

当用户按下 Interrupt 键时，解除该应用程序的锁定，并调用 SIGINT 信号句柄函数。此信号句柄函数包括对 sqlbreak() 的调用，来中断数据库服务器。

请以 sqlbreakcallback() 函数指定超时间隔。

经过了该超时间隔之后，解除应用程序的锁定，并调用 callback 函数。此 callback 函数包括对 sqlbreak() 的调用，来中断数据库服务器。

在您的程序调用 sqlbreak() 之前，请以 sqldone() 函数来验证数据库服务器当前正在处理 SQL 请求。

返回代码

0

sqlbreak() 调用成功。数据库服务器连接存在，且或者成功地发送了中断请求，或者数据库服务器空闲。

!=0

当您调用 sqlbreak() 时，无数据库服务器正在运行（不存在数据库连接）。

5.2.169 sqlbreakcallback() 函数

sqlbreakcallback() 函数允许您指定超时间隔，并注册 callback 函数。当数据库服务器正在处理 SQL 请求时，该 callback 函数为应用程序提供重新获得控制的方法。

限制： 如果您的 GBase 8s ESQL/C 应用程序使用共享内存（onipcshm）作为 **nettype** 来连接到 GBase 8s 数据库服务器，则请不要使用 sqlbreakcallback() 函数。共享内不能不是真正的网络协议，且不处理支持 callback 函数所需要的 nonblocking I/O。当您随同共享内存使用 sqlbreakcallback() 时，该调用看来似乎成功地注册 callback 函数（它返回零）；然而，在 SQL 请求期间，该应用程序从不调用 callback 函数。

语法

mint sqlbreakcallback(*timeout, callbackfunc_ptr*);

int4 timeout;

void (* callbackfunc_ptr)(int status);

timeout

在应用程序进程重新获得控制之前，SQL 请求等待执行的时间间隔。

此值可为如下：

-1

清除 timeout 值。

0

立即调用 callbackfunc_ptr 指示的函数。

>0

将超时间隔设置为在该应用程序调用 *callbackfunc_ptr* 指示的函数之前经过的毫秒数。

timeout 参数是一 4 字节变量。此参数依赖于操作系统：它可为 **int**、**long** 或 **short** 数据类型的变量。

callbackfunc_ptr

执行用户定义的 callback 函数的指针。

用法

在您以 sqlbreakcallback() 注册 callback 函数之后，应用程序在执行 SQL 请求的三个不同的点处调用此函数。callback 函数的 *status* 参数中的值指示应用程序在哪个点调用该函数。下表总结 *status* 值。

调用 callback 函数的时刻	status 参数的值
当数据库服务器开始处理 SQL 请求时	status = 1
在数据库服务器执行 SQL 请求时，当已经过了超时间隔时	status = 2
当数据库服务器完成 SQL 请求的处理时	status = 0

当您以 *status* 值 2 调用 callback 函数时，callback 函数可确定数据库服务器是否可以下列行动之一来继续处理：

它可调用 sqlbreak() 函数来取消该 SQL 请求。

它可忽略 sqlbreak() 调用，来继续该 SQL 请求。

该 callback 函数，以及任何它的子例程，仅可包含下列 GBase 8s ESQL/C 控制函数：sqldone()、sqlbreak() 和 sqlbreakcallback()。

如果您以 timeout 值零来调用 sqlbreakcallback()，则 callback 函数立即执行。callback 函数多次执行，直到它包含对 sqlbreakcallback() 的调用来以下列行动之一重新定义该 callback 函数为止：

它解除该 callback 函数的关联，来终止该 callback 函数的调用，如下：

```
sqlbreakcallback(-1L, (void *)NULL);
```

它定义某个其他的 callback 函数，或将 timeout 值重置为非零值，如下：

```
sqlbreakcallback(timeout, callbackfunc_ptr);
```

重要： 小 timeout 值可能反而影响您的应用程序的性能。

在您调用 sqlbreakcallback() 函数之前，您必须建立数据库服务器连接。在连接期间，callback 函数保持有效，或直到 sqlbreakcallback() 函数重新定义该 callback 函数为止。

返回代码

0

sqlbreakcallback() 调用成功。

<0

sqlbreakcallback() 调用不成功。

5.2.170 sqldetach() 函数

sqldetach() 函数从数据库服务器脱离进程。通常，当应用程序分叉一个新的进程来开始新的执行流时，您调用此函数。

语法

mint sqldetach();

用法

如果在应用程序开启至数据库服务器的连接之后，它创建一个或多个进程，则所有子进程 从父进程（产生了子进程的应用程序进程）继承数据库服务器连接。然而，数据库服务器仍假定此连接仅有一个进程。如果一个数据库服务器连接试图同时为父进程和子进程服务时，可导致问题。例如，如果两个进程都发送消息做某事，则数据库服务器无法知道哪个消息属于哪个进程。数据库服务器可能未按合理的顺序接收消息，因此可能生成错误（诸如错误 -408）。

自此情况下，请从子进程调用 sqldetach() 函数。sqldetach() 函数将子进程从父进程建立的连接（子进程继承的）脱离。此行动删除子进程中的所有数据库服务器连接。然后，子进程可建立自己的至数据库服务器的连接。

请随同 fork() 系统调用使用 sqldetach() 函数。当您以数据库服务器连接创建来自应用程序进程的子进程时，函数调用排序如下：

调用来自父进程的 fork() 来创建父进程的副本（子进程）。现在，父子同时分享至数据库服务器的相同的连接。

调用来自子进程的 sqldetach()，来将子进程从数据库服务器脱离。此调用关闭子进程中的连接。

限制：您不可在 vfork() 调用之后使用 sqldetach()，因为直到调用 exec() 函数为止，vfork() 才执行真正的进程分叉。在父进程使用 exec() 之后，请不要使用 sqldetach()；当

exec() 启动子进程时，该子进程不继承父进程建立了的连接。

对 sqldetach() 函数的调用不影响父进程的数据库服务器会话。因此，在子进程中执行 sqldetach() 之后，父进程保持任何打开的游标、事务或数据库，且子进程不拥有数据库服务器会话或数据库服务器连接。

当您从父进程调用 sqlexit() 函数时，该函数删除父进程中的连接，但不影响子进程中的连接。类似地，当您从子进程调用 sqlexit() 时，该函数仅删除子连接；它不影响父连接。在 sqlexit() 函数关闭连接之前，它回滚任何打开的事务。

如果您从子进程执行 DISCONNECT 语句，则您从数据库服务器断开该进程的连接，并终止对应于那些连接的数据库会话。如果任何事务是打开的，则 DISCONNECT 失败。

在子进程应用程序调用 sqldetach() 之前，如果它仅有隐式的连接，则执行下一 SQL 语句，或执行下一 sqlstart() 库函数，会重新建立至缺省的数据库服务器的隐式的连接。如果该应用程序已建立了一个或多个显式的连接，则在您执行任何其他 SQL 语句之前，必须发出 CONNECT 语句。

sqldetach 演示程序说明如何使用 sqldetach() 函数。

返回代码

0

sqldetach() 调用成功。

<0

sqldetach() 调用不成功。

示例

demo 目录中的 sqldetach.ec 文件包含此样例程序。

```
/*
    * sqldetach.ec *

    This program demonstrates how to detach a child process from a parent process using the
ESQL/C sqldetach() library function.
```

```
*/

main()
{
    EXEC SQL BEGIN DECLARE SECTION;
        mint pa;
    EXEC SQL END DECLARE SECTION;

    printf("SQLDETACH Sample ESQL Program running.\n\n");

    printf("Beginning execution of parent process.\n\n");
    printf("Connecting to default server...\n");
    EXEC SQL connect to default;
    chk("CONNECT");
    printf("\n");

    printf("Creating database 'aa'...\n");
    EXEC SQL create database aa;
    chk("CREATE DATABASE");
    printf("\n");

    printf("Creating table 'tab1'...\n");
    EXEC SQL create table tab1 (a integer);
    chk("CREATE TABLE");
    printf("\n");

    printf("Inserting 4 rows into 'tab1'...\n");
    EXEC SQL insert into tab1 values (1);
    chk("INSERT #1");
    EXEC SQL insert into tab1 values (2);
    chk("INSERT #2");
    EXEC SQL insert into tab1 values (3);
    chk("INSERT #3");
```

```
    EXEC SQL insert into tab1 values (4);
    chk("INSERT #4");
    printf("\n");

    printf("Selecting rows from 'tab1' table...\n");
    EXEC SQL declare c cursor for select * from tab1;
    chk("DECLARE");

   EXEC SQL open c;
    chk("OPEN");

    printf("\nForking child process...\n");
    fork_child();

    printf("\nFetching row from cursor 'c'...\n");
    EXEC SQL fetch c into $pa;
    chk("Parent FETCH");
    if (sqlca.sqlcode == 0)
        printf("Value selected from 'c' = %d.\n", pa);
    printf("\n");

    printf("Cleaning up...\n");
    EXEC SQL close database;
    chk("CLOSE DATABASE");
    EXEC SQL drop database aa;
    chk("DROP DATABASE");
    EXEC SQL disconnect all;
    chk("DISCONNECT");

    printf("\nEnding execution of parent process.\n");
    printf("\nSQLDETACH Sample Program over.\n\n");
}
```

```
fork_child()
{
        mint rc, status, pid;

        EXEC SQL BEGIN DECLARE SECTION;
                mint cnt, ca;
        EXEC SQL END DECLARE SECTION;

        pid = fork();
        if (pid < 0)
                printf("can't fork child.\n");

        else if (pid == 0)
       {
                printf("\n*********************************\n");
                printf("* Beginning execution of child process.\n");
                rc = sqldetach();
                printf("* sqldetach() call returns %d.\n", rc);

/* Verify that the child is not longer using the parent's connection and has not inherited the
parent's connection environment.
                  */
                printf("* Trying to fetch row from cursor 'c'...\n");
                EXEC SQL fetch c into $ca;
                chk("* Child FETCH");
                if (sqlca.sqlcode == 0)
                        printf("* Value from 'c' = %d.\n", ca);

                /* startup a connection for the child, since
                  * it doesn't have one.
                  */
printf("\n* Establish a connection, since child doesn't have one\n");
                printf("* Connecting to database 'aa'...\n");
```

```
            EXEC SQL connect to 'aa';
            chk("* CONNECT");
            printf("* \n");
            printf("* Determining number of rows in 'tab1'...\n");
            EXEC SQL select count(*) into $cnt from tab1;
            chk("* SELECT");
            if (sqlca.sqlcode == 0)
                printf("* Number of entries in 'tab1' =    %d.\n", cnt);
            printf("* \n");

            printf("* Disconnecting from 'aa' database...\n");
            EXEC SQL disconnect current;
            chk("* DISCONNECT");
            printf("* \n");
            printf("* Ending execution of child process.\n");
            printf("*********************************\n");

            exit();
        }

        /* wait for child process to finish */
        while ((rc = wait(&status)) != pid && rc != -1);

}

chk(s)
char *s;
{
        mint msglen;
        char buf1[200], buf2[200];

        if (SQLCODE == 0)
        {
```

```
            printf("%s was successful\n", s);
            return;
        }
         printf("\n%s:\n", s);
         if (SQLCODE)
        {
            printf("\tSQLCODE =    %6d: ", SQLCODE);
            rgetlmsg(SQLCODE, buf1, sizeof(buf1), &msglen);
            sprintf(buf2, buf1, sqlca.sqlerrm);
            printf(buf2);
            if (sqlca.sqlerrd[1])
           {
                printf("\tISAM Error =    %6hd: ", sqlca.sqlerrd[1]);
          rgetlmsg(sqlca.sqlerrd[1], buf1, sizeof(buf1), &msglen);
                sprintf(buf2, buf1, sqlca.sqlerrm);
                printf(buf2);
           }
        }
}
```

输出

```
SQLDETACH Sample ESQL Program running.
Beginning execution of parent process.

Connecting to default server...
CONNECT was successful

Creating database 'aa'...
CREATE DATABASE was successful

Creating table 'tab1'...
CREATE TABLE was successful
```

```
Inserting 4 rows into 'tab1'...
INSERT #1 was successful
INSERT #2 was successful
INSERT #3 was successful
INSERT #4 was successful

Selecting rows from 'tab1' table...
DECLARE was successful
OPEN was successful

Forking child process...

**********************************************
* Beginning execution of child process.
* sqldetach() call returns 0.
* Trying to fetch row from cursor 'c'...

* Child FETCH:
     SQLCODE =       -404: The cursor or statement is not available.

* Establish a connection, since child doesn't have one
* Connecting to database 'aa'...
* CONNECT was successful
*
* Determining number of rows in 'tab1'...
* SELECT was successful
* Number of entries in 'tab1' =    4.
*
* Disconnecting from 'aa' database...
* DISCONNECT was successful
*
* Ending execution of child process.
**********************************************
```

```
SQLDETACH Sample ESQL Program running.

Beginning execution of parent process.

Connecting to default server...
CONNECT was successful

Creating database 'aa'...
CREATE DATABASE was successful

Creating table 'tab1'...

CREATE TABLE was successful

Inserting 4 rows into 'tab1'...
INSERT #1 was successful
INSERT #2 was successful
INSERT #3 was successful
INSERT #4 was successful

Selecting rows from 'tab1' table...
DECLARE was successful
OPEN was successful

Forking child process...

Fetching row from cursor 'c'...
Parent FETCH was successful
Value selected from 'c' = 1.

Cleaning up...
CLOSE DATABASE was successful
DROP DATABASE was successful
```

DISCONNECT was successful

Ending execution of parent process.

SQLDETACH Sample Program over.

5.2.171 sqldone() 函数

sqldone() 函数确定数据库服务器当前是否正在处理 SQL 请求。

语法

mint sqldone();

用法

请在下列情况下使用 sqldone() 来检测数据库服务器的状态：

在调用 sqlbreak() 函数来确定数据库服务器是否正在处理 SQL 请求之前。

在信号句柄函数中，在调用 longjmp() 系统函数之前。如果 sqldone() 返回零（数据库服务器空闲），则仅使用信号句柄函数中的 longjmp()。

当 sqldone() 函数确定当前数据库服务器未正在处理 SQL 请求时，您可假定数据库服务器未开启任何其他处理，直到您的应用程序发出它的下一请求为止。

您可能想要为 -439 值创建定义的常量，来使得您的代码更加可读。例如，下列代码片段创建 SERVER_BUSY 常量，然后使用它来检测 sqldone() 返回状态：

```
#define SERVER_BUSY -439
.
.
.
 if (sqldone() == SERVER_BUSY)
```

返回代码

0

数据库服务器当前没有正在处理 SQL 请求：它是空闲的。

-439

数据库服务器当前正在处理 SQL 请求。

5.2.172 sqlexit() 函数

sqlexit() 函数终止所有数据库服务器连接，并释放资源。您可使用 sqlexit() 来减少程序中数据库开销，其仅短暂地引用数据库且在很长间隔之后，或仅在初始化期间访问数据库。

语法

mint sqlexit();

用法

当无数据库打开时，仅调用 sqlexit() 函数。如果打开的数据库使用事务，则在 sqlexit() 关闭数据库之前，它回滚任何打开的事务。此函数的行为类似于 DISCONNECT ALL 语句。然而，如果任何当前的事务退出，则 DISCONNECT ALL 语句*失败*。在您调用 sqlexit() 之前，请使用 CLOSE DATABASE 语句来关闭打开的数据库。

在应用程序调用 sqlexit() 之前，如果它仅有一个隐式的连接，则下一 SQL 语句的执行，或 sqlstart() 库函数的执行会重新建立至缺省的数据库服务器的隐式的连接。如果应用程序创建了一个或多个显式的连接，则在您执行任何其他 SQL 语句之前，您必须发出 CONNECT 语句。

返回代码

0

sqlexit() 调用成功。

<0

sqlexit() 调用不成功。

5.2.173 sqlsignal() 函数

sqlsignal() 函数启用或禁用 GBase 8s ESQL/C 库处理的信号的信号处理。

语法

void sqlsignal(sigvalue, sigfunc_ptr, mode)

```
mint sigvalue;
void (*sigfunc_ptr) (void);
int mode;
```

sigvalue

需要捕获的特定的信号的 **mint** 值（如 signal.h) 定义的那样）。

当前，此参数是未来设计功能的占位符。请将此参数初始化为 -1。

sigfunc_ptr

指向用户定义的函数的指针，其不带参数，来为 *sigvalue* 信号作为信号句柄调用。

当前，此参数是未来设计功能的占位符。请将此参数初始化为指向不接收参数的函数的空指针。

mode

可为三种可能的模式之一：

0

初始化信号处理。

1

禁用信号处理。

2

重新启用信号处理。

用法

sqlsignal() 函数当前仅为 SIGCHLD 信号提供处理。在有些情况下，在应用程序结束之后，defunct 子进程保持。如果应用程序不清理这些进程，则它们导致进程 ID 不必要的使用，并增加您用尽进程的风险。对于客户机-服务器通讯，当应用程序使用管道时（即，sqlhosts 文件的 **nettype** 字段为 **ipcpip**），此行为才显而易见。对于其他通讯机制，你无需调用 sqlsignal()（例如， **tlipcp** 的 **nettype**）。

sqlsignal() 的 *mode* 参数确定 sqlsignal() 执行的任务，如下：

设置 *mode* 为 0 来初始化信号处理。

```
sqlsignal(-1, (void (*)(void))0, 0);
```

当您以 sqlsignal() 来初始化信号处理时，GBase 8s ESQL/C 库捕获 SIGCHLD 信号来处理 defunct 子进程的清除。此初始的 sqlsignal() 调用必须发生在您的应用程序的开头，在程序中的第一个 SQL 语句之前。如果您省略此初始的调用，则在您的程序中，稍后不可开启信号处理能力。

启用或禁用信号处理。

如果您想要让 GBase 8s ESQL/C 库为程序的部分执行信息处理，且您自己的代码为其他部分执行信号处理，则您可采取下列行动：

要禁用信号处理，请以设置为 1 的 *mode* 调用 sqlsignal()，在您想要您的程序处理信号的点处：

```
sqlsignal(-1, (void (*)(void))0, 1);
```

要重新启用信号处理，请以设置为 2 的 mode 来调用 sqlsignal()，在您想要 GBase 8s ESQL 库处理信号的点处：

```
sqlsignal(-1, (void (*)(void))0, 2);
```

当您以 sqlsignal() 初始化 SIGCHLD 信号处理时，您允许 GBase 8s ESQL/C 库处理 SIGCHLD 清除。否则，如果 defunct 子进程是个问题，则您的应用程序必须执行对这些进程的清理。

5.2.174 sqlstart() 函数

sqlstart() 函数启动隐式的缺省连接。隐式的缺省连接可支持至缺省数据库服务器的一个连接。（GBASEDBTSERVER 环境变量指定的）。

提示： 在仅使用一个连接的 6.0 版本之前，限制应用程序使用 sqlstart()。为了与这些应用程序的较早版本相兼容，GBase 8s ESQL/C 继续支持此函数。对于 Version 6.0 和更晚的应用程序，请使用 CONNECT 语句来建立至缺省数据库服务器的显示的连接。

语法

```
mint sqlstart();
```

用法

GBase 8s ESQL/C 为仅支持单个连接的 Version 6.0 之前的应用程序提供 sqlstart() 函数。在此上下文中，可能的 sqlstart() 使用如下：

您仅需证实缺省数据库服务器可用，但您不打算打开数据库。如果 sqlstart() 调用失败，则您可检查返回状态来证实缺省数据库服务器不可用。

当应用程序在网络上运行时，您需要加速 DATABASE 语句的执行。当您将 sqlstart() 的调用放置在初始化例程中时，在用户开始与该应用程序交互之前，应用程序建立连接。然后，DATABASE 语句可打开指定的数据库。

您不知道要访问的实际数据库的名称，或您的应用程序计划创建数据库。调用 sqlstart() 可建立隐式的缺省连接，且稍后应用程序可确定要访问或创建的数据库的名称。

如果您有一个 6.0 版本之前的应用程序，处于任何其他原因，其需要隐式的缺省连接，则请使用 DATABASE 语句，而不是 sqlstart()。对于 6.0 版及其后来的应用程序，请使用 CONNECT 语句来建立数据库服务器连接。

当您调用 sqlstart() 函数时，请确保该应用程序尚未建立任何连接，隐式的或显式的。当应用程序已建立了显式的连接时，sqlstart() 返回错误 -1811。如果建立了隐式的连接，则 sqlstart() 返回错误 -1802。

在您建立显式的连接之前，您可多次调用此函数，只要在下一 sqlstart() 调用之前断开每一隐式的连接即可。

返回代码

0

sqlstart() 调用成功。

<0

sqlstart() 调用不成功。

5.2.175 stcat() 函数

stcat() 函数将一个以空终止的字符串列接到另一字符串的末尾。

语法

```
void stcat(s, dest)
        char *s, *dest;
```

s

指向 stcat() 放置在目的字符串的末尾处的字符串的开头的指针。

dest

指向以空终止的字符串的开头的指针。

示例

此样例程序在 demo 目录中的 stcat.ec 文件中。

```
/*
        * stcat.ec *
This program uses stcat() to append user input to a SELECT statement.
        */

        #include <stdio.h>

        /*
        * Declare a variable large enough to hold
        * the select statement + the value for customer_num entered from the terminal.
        */
        char selstmt[80] = "select fname, lname from customer where customer_num =
";

        main()
        {
        char custno[11];

        printf("STCAT Sample ESQL Program running.\n\n");

        printf("Initial SELECT string:\n    '%s'\n", selstmt);

        printf("\nEnter Customer #: ");
        gets(custno);


        /*
        * Add custno to "select statement"
        */
```

```
        printf("\nCalling stcat(custno, selstmt)\n");
        stcat(custno, selstmt);
        printf("SELECT string is:\n    '%s'\n", selstmt);

        printf("\nSTCAT Sample Program over.\n\n");
        }
```

输出

```
STCAT Sample ESQL Program running.

        Initial SELECT string:
        'select fname, lname from customer where customer_num = '

        Enter Customer #: 104

        Calling stcat(custno, selstmt)
        SELECT string is:
'select fname, lname from customer where customer_num = 104'

        STCAT Sample Program over.
```

5.2.176 stchar() 函数

stchar() 函数在定长的字符串中存储以空终止的字符串，如果有必要，则以空格填充末尾。

语法

```
void stchar(from, to, count)
        char *from;
        char *to;
        mint count;
```

from

指向以空终止的源字符串的第一个字节的指针。

to

指向定长目标字符串的指针。此参数可指向重叠 *from* 参数指向的位置的位置。在此情况下，GBase 8s ESQL/C 丢弃 *from* 指向的值。

count

定长目标字符串中的字节数。

示例

此样例程序在 demo 目录中的 stchar.ec 文件中。

```
/*
        * stchar.ec *

 The following program shows the blank padded result produced by stchar() function.
        */

        #include <stdio.h>

        main()
        {
        static char src[] = "start";
        static char dst[25] = "123abcdefghijkl;.";

        printf("STCHAR Sample ESQL Program running.\n\n");

        printf("Source string: [%s]\n", src);
printf("Destination string before stchar: [%s]\n", dst);

        stchar(src, dst, sizeof(dst) - 1);

 printf("Destination string    after stchar: [%s]\n", dst);

        printf("\nSTCHAR Sample Program over.\n\n");
        }
```

输出

STCHAR Sample ESQL Program running.

Source string: [start]
Destination string before stchar: [123abcdefghijkl;.]
Destination string after stchar: [start]

STCHAR Sample Program over.

5.2.177 stcmpr() 函数

stcmpr() 函数比较两个以空终止的字符串。

语法

mint stcmpr(*s1*, *s2*)
char *s1, *s2;

s1

指向第一个以空终止的字符串的指针。

s2

指向第二个以空终止的字符串的指针。

重要： 在 ASCII 排序序列中，当 *s1* 出现在 *s2* 之后时，*s1* 大于 *s2*。

返回代码

=0

两个字符串相同。

<0

第一个字符串小于第二个字符串。

>0

第一个字符串大于第二个字符串。

示例

此样例程序在 demo 目录中的 stcmpr.ec 文件中。

```
/*
        * stcmpr.ec *

The following program displays the results of three string comparisons using stcmpr().
        */

        #include <stdio.h>

        main()
        {
printf("STCMPR Sample ESQL Program running.\n\n");
printf("Executing: stcmpr(\"aaa\", \"aaa\")\n");
printf("    Result = %d", stcmpr("aaa", "aaa"));    /* equal */
printf("\nExecuting: stcmpr(\"aaa\", \"aaaa\")\n");
printf("    Result = %d", stcmpr("aaa", "aaaa")); /* less */
printf("\nExecuting: stcmpr(\"bbb\", \"aaaa\")\n");
printf("    Result = %d\n", stcmpr("bbb", "aaaa")); /* greater */
printf("\nSTCMPR Sample Program over.\n\n");
        }
```

输出

```
STCMPR Sample ESQL Program running.

        Executing: stcmpr("aaa", "aaa")
        Result = 0
        Executing: stcmpr("aaa", "aaaa")
        Result = -1
        Executing: stcmpr("bbb", "aaaa")
        Result = 1

        STCMPR Sample Program over.
```

5.2.178 stcopy() 函数

stcopy() 函数将以空终止的字符串从内存中一个位置复制至另一位置。

语法

void stcopy(*from*, *to*)

char *from, *to;

from

指向您想要 stcopy() 复制的以空终止的字符串的指针。

to

指向内存中 stcopy() 复制字符串的位置的指针。

示例

此样例程序在 demo 目录中的 stcopy.ec 文件中。

```
/*
        * stcopy.ec *

This program displays the result of copying a string using stcopy().
        */

        #include <stdio.h>

        main()
        {
static char string[] = "abcdefghijklmnopqrstuvwxyz";

printf("STCOPY Sample ESQL Program running.\n\n");

printf("Initial string:\n    [%s]\n", string);
/* display dest */
stcopy("John Doe", &string[15]);                    /* copy */
```

```
printf("After copy of 'John Doe' to position 15:\n    [%s]\n",
                string);

                printf("\nSTCOPY Sample Program over.\n\n");
                }
```

输出

```
STCOPY Sample ESQL Program running.

            Initial string:
            [abcdefghijklmnopqrstuvwxyz]
            After copy of 'John Doe' to position 15:
            [abcdefghijklmnoJohn Doe]

            STCOPY Sample Program over.
```

5.2.179 stleng() 函数

stleng() 函数返回您指定的以空终止的字符串的以字节计的长度。

语法

```
mint stleng(string)
char *string;
string
```

指向以空终止的字符串的指针。

用法

长度不包括空终止符。

示例

此样例程序在 demo 目录中的 stleng.ec 文件中。

```
/*
            * stleng.ec *

 This    program uses stleng to find strings that are greater than 35 characters in
length.
            */
```

```
#include <stdio.h>

char *strings[] =
{
"Your First Season's Baseball Glove",
"ProCycle Stem with Pearl Finish",
"Athletic Watch w/4-Lap Memory, Olympic model",
"High-Quality Kickboard",
"Team Logo Silicone Swim Cap - fits all head sizes",
};

main(argc, argv)
int argc;
char *argv[];
{
mint length, i;

printf("STLENG Sample ESQL Program running.\n\n");

printf("Strings with lengths greater than 35:\n");
i = 0;
while(strings[i])
{
if((length = stleng(strings[i])) > 35)
{
printf("    String[%d]: %s\n", i, strings[i]);
printf("    Length: %d\n\n", length);
}
++i;
}
printf("\nSTLENG Sample Program over.\n\n");
}
```

输出

```
STLENG Sample ESQL Program running.
Strings with lengths greater than 35:
String[2]: Athletic Watch w/4-Lap Memory, Olympic model
Length: 44
String[4]: Team Logo Silicone Swim Cap - fits all head sizes
Length: 49
STLENG Sample Program over.
```

5.3 智能大对象函数的示例

您的 GBase 8s 软件包括演示数据库。GBase 8s ESQL/C 在本出版物中，还包括许多演示程序和示例的源文件，其中有些访问该演示数据库。

在 Windows(TM) 环境中，您可在 %GBASEDBTDIR%\demo\esqldemo 目录中找到 GBase 8s ESQL/C 示例程序的源文件。

在 UNIX(TM) 操作系统上，您可在 $GBASEDBTDIR/demo/esqlc 目录中找到 GBase 8s ESQL/C 示例程序的源文件。包括 GBase 8s ESQL/C 的 **esqldemo** 脚本将源文件从 $GBASEDBTDIR/demo/esqlc 目录复制至当前目录内。

这些示例说明如何使用 GBase 8s ESQL/C 库函数来访问智能大对象。如果您正在使用 GBase 8s 作为您的数据库服务器，则仅这些示例适用。

5.3.1 前提条件

本部分中的示例依赖于下列 stores7 数据库的替代 catalog 表的存在。这些样例还依赖于 sbspace s9_sbspc 的存在，存储 BLOB 和 CLOB 列的内容，替代 catalog 表中的 picture 和 advert_descr。

```
-- create table that uses smart large objects (CLOB & BLOB) to
-- store the catalog advertisement data.
CREATE TABLE    catalog
(
    catalog_num        SERIAL8 (10001) primary key,
    stock_num          SMALLINT,
    manu_code           CHAR(3),
    unit                CHAR(4),
    advert              ROW (picture BLOB, caption VARCHAR(255, 65)),
    advert_descr        CLOB,
   FOREIGN KEY (stock_num, manu_code)
    REFERENCES stock constraint aa)
            PUT advert IN (s9_sbspc)
            (EXTENT SIZE 100),
            advert_descr IN (s9_sbspc)
            (EXTENT SIZE 20, KEEP ACCESS TIME)
```

下列示例说明创建 sbspace 的典型命令。特定的选项的值可不同。您必须以您为 sbspace 分配的文件的完整文件名称替换 PATH。

```
touch s9_sbspc
gspaces -c -S s9_sbspc -g 4 -p PATH -o 0 -s 2000
```

下列代码说明加载文件中条目的格式，您可能使用该文件来将数据库加载至替代 **catalog** 表内。加载文件包含 LOAD 语句加载至表内的数据。下图中的每一行加载表中的一行。下图仅展示您可用来加载 **catalog** 表的代码样例。

```
|1|HRO|case|ROW(/tmp/cn_1001.gif,"Your First Season's Baseball Glove")|0,62,
/tmp/catalog.des|
0|1|HSK|case|ROW(NULL,"All Leather, Hand Stitched, Deep Pockets, Sturdy
Webbing That Won't Let Go")||
```

```
0|1|SMT|case|ROW(NULL,"A Sturdy Catcher's Mitt With the Perfect Pocket")||
0|2|HRO|each|ROW(NULL,"Highest Quality Ball Available, from the
Hand-Stitching to the Robinson Signature")||
0|3|HSK|case|ROW(NULL,"High-Technology Design Expands the Sweet Spot")||
0|3|SHM|case|ROW(NULL,"Durable Aluminum for High School and Collegiate
Athletes")||
0|4|HSK|case|ROW(NULL,"Quality Pigskin with Norm Van Brocklin Signature")||
```

下列代码片段说明 catalog.des 文件中的信息的格式，前面的代码引用它。前面的代码中 **advert_descr** 的条目（0,62,/tmp/catalog.des）指定从其加载该描述的偏移量、长度和文件名称。偏移量和长度是十六进制值。

```
Brown leather. Specify first baseman's or infield/outfield style.
Specify right- or left-handed.

Double or triple crankset with choice of chainrings. For double  crankset, chainrings
from 38-54 teeth. For triple crankset, chainrings from 24-48 teeth.

No buckle so no plastic touches your chin. Meets both ANSI and Snell standards for
impact protection.7.5 oz. Lycra cover.

Fluorescent yellow.

 Super shock-absorbing gel pads disperse vertical energy into a horizontal plane for
extraordinary cushioned comfort. Great motion control.
 Mens only. Specify size
```

本部分包含下列示例程序。

程序	描述	请参阅
create_clob.ec	将包含 CLOB 列的行插入值替代目录表内。	create_clob.ec 程序
get_lo_info.ec	将来自 **stores7** 数据库的存货表的价格追加到替代 **catalog** 表的 **advert_descr** 列。	get_lo_info.ec 程序
upd_lo_descr.ec	取得 **advert_descr** 列非空的目录项的价格，并将该价格追加到描述。	upd_lo_descr.ec 程序

5.3.2 create_clob.ec 程序

create_clob 演示如何对智能大对象执行下列任务：

以用户定义的存储特征来创建智能大对象。

将新的智能大对象插入至数据库列内。

示例的存储特征

create_clob 程序创建 **advert_descr** 智能大对象，其有下列用户定义的存储特征：

开启日志记录：LO_LOG

保持最后的访问时间（缺省来自 **advert_descr** 列）：LO_KEEP_ACCESSTIME

完整性高

分配 extent 大小为 10 KB

```
EXEC SQL include int8;
EXEC SQL include locator;
EXEC SQL define BUFSZ 10;

                        extern char statement[80];

                        main()
                        {
                        EXEC SQL BEGIN DECLARE SECTION;
                        int8 catalog_num, estbytes, offset;
                        int error, numbytes, lofd, ic_num, buflen = 256;
                        char buf[256], srvr_name[256], col_name[300];
                        ifx_lo_create_spec_t *create_spec;
                        fixed binary 'clob' ifx_lo_t descr;
                        EXEC SQL END DECLARE SECTION;

                        void nullterm(char *);
                        void handle_lo_error(int);

                        EXEC SQL whenever sqlerror call whenexp_chk;
                        EXEC SQL whenever sqlwarning call whenexp_chk;

                        printf("CREATE_CLOB Sample ESQL program running.\n\n");
                        strcpy(statement, "CONNECT stmt");
                        EXEC SQL connect to 'stores7';
                        EXEC SQL get diagnostics exception 1
                        :srvr_name = server_name;
                        nullterm(srvr_name);

/* Allocate and initialize the LO-specification structure   */
                        error = ifx_lo_def_create_spec(&create_spec);
                        if (error < 0)
                        {
                        strcpy(statement, "ifx_lo_def_create_spec()");
                        handle_lo_error(error);
                        }

                        /* Get the column-level storage characteristics for the
                        * CLOB column, advert_descr.
                        */
                        sprintf(col_name, "stores7@%s:catalog.advert_descr",
                        srvr_name);
                        error = ifx_lo_col_info(col_name, create_spec);
                        if (error < 0)
```

```
                        {
                        strcpy(statement, "ifx_lo_col_info()");
                        handle_lo_error(error);
                        }

                        /* Override column-level storage characteristics for
                        * advert_desc with the following user-defined storage
                        * characteristics:
                        * no logging
                        * extent size = 10 kilobytes
                        */
                        ifx_lo_specset_flags(create_spec,LO_LOG);
                        ifx_int8cvint(BUFSZ, &estbytes);
                        ifx_lo_specset_estbytes(create_spec, &estbytes);

/* Create an LO-specification structure for the smart large object    */

                        if ((lofd = ifx_lo_create(create_spec, LO_RDWR,
                        &descr, &error)) == -1)
                        {
                        strcpy(statement, "ifx_lo_create()");
                        handle_lo_error(error);
                        }
                        /* Copy data into the character buffer 'buf' */

                        sprintf(buf, "%s %s",
                "Pro model infielder's glove. Highest quality leather and
                        stitching. "
                        "Long-fingered, deep pocket, generous web.");

                        /* Write contents of character buffer to the open smart
                        * large object that lofd points to.    */

                        ifx_int8cvint(0, &offset);
                        numbytes = ifx_lo_writewithseek(lofd, buf, buflen,
                        &offset, LO_SEEK_SET, &error);
                        if ( numbytes < buflen )
                        {
                        strcpy(statement, "ifx_lo_writewithseek()");
                        handle_lo_error(error);
                        }

      /* Insert the smart large object into the table */
                        strcpy(statement, "INSERT INTO catalog");
          EXEC SQL insert into catalog values (0, 1, 'HSK', 'case',          ROW(NULL,
NULL),:descr);

      /* Need code to find out what the catalog_num value was
         * assigned to new row */
      /* Close the LO file descriptor */
                        ifx_lo_close(lofd);

         /* Select back the newly inserted value. The SELECT
          * returns an LO-pointer structure, which you then use to
          * open a smart large object to get an LO file descriptor.
                        */
                        ifx_getserial8(&catalog_num);
```

```
        strcpy(statement, "SELECT FROM catalog");
        EXEC SQL select advert_descr into :descr from catalog
        where catalog_num = :catalog_num;

        /* Use the returned LO-pointer structure to open a smart
        * large object and get an LO file descriptor.
        */
        lofd = ifx_lo_open(&descr, LO_RDONLY, &error);
        if (error < 0)
        {
        strcpy(statement, "ifx_lo_open()");
        handle_lo_error(error);
        }
        /* Use the LO file descriptor to read the data in the
        * smart large object.
        */
        ifx_int8cvint(0, &offset);
        strcpy(buf, "");
        numbytes = ifx_lo_readwithseek(lofd, buf, buflen,
        &offset, LO_SEEK_CUR, &error);
        if (error || numbytes == 0)
        {
        strcpy(statement, "ifx_lo_readwithseek()");
        handle_lo_error(error);
        }
        if(ifx_int8toint(&catalog_num, &ic_num) !=    0)
printf("\nifx_int8toint failed to convert catalog_num to int");
printf("\nContents of column \'descr\' for catalog_num:
        %d \n\t%s\n",
        ic_num, buf);
        /* Close open smart large object */
        ifx_lo_close(lofd);
        /* Free LO-specification structure */
        ifx_lo_spec_free(create_spec);
        }

        void handle_lo_error(error_num)
        int error_num;
        {
        printf("%s generated error %d\n", statement, error_num);
        exit(1);
        }

        void nullterm(str)
        char *str;
        {
        char *end;

        end = str + 256;
        while(*str != ' ' && *str != '\0' && str < end)
        {
        ++str;
        }
        if(str >= end)
        printf("Error: end of str reached\n");
        if(*str == ' ')
        *str = '\0';
```

```
                        }

                        /* Include source code for whenexp_chk() exception-checking
                         * routine
                         */

                        EXEC SQL include exp_chk.ec;
```

get_lo_info.ec 程序

此程序检索关于存储在 BLOB 列中的智能大对象的信息。

```
#include <time.h>

EXEC SQL define BUFSZ 10;

extern char statement[80];

main()
{
        int error, ic_num, oflags, cflags, extsz, imsize, isize, iebytes;
        time_t time;
        struct tm *date_time;
        char col_name[300], sbspc[129];

        EXEC SQL BEGIN DECLARE SECTION;
        fixed binary 'blob' ifx_lo_t picture;
        char srvr_name[256];
        ifx_lo_create_spec_t *cspec;
        ifx_lo_stat_t *stats;
        ifx_int8_t size, c_num, estbytes, maxsize;
        int lofd;
        long atime, ctime, mtime, refcnt;
        EXEC SQL END DECLARE SECTION;

        void nullterm(char *);
        void handle_lo_error(int);

        imsize = isize = iebytes = 0;
        EXEC SQL whenever sqlerror call whenexp_chk;
        EXEC SQL whenever sqlwarning call whenexp_chk;

        printf("GET_LO_INFO Sample ESQL program running.\n\n");
        strcpy(statement, "CONNECT stmt");
        EXEC SQL connect to 'stores7';
        EXEC SQL get diagnostics exception 1
                :srvr_name = server_name;
        nullterm(srvr_name);

        EXEC SQL declare ifxcursor cursor for
                select catalog_num, advert.picture
                into :c_num, :picture
                from catalog
                where advert.picture is not null;

EXEC SQL open ifxcursor;
        while(1)
        {
                EXEC SQL fetch ifxcursor;
                if (strncmp(SQLSTATE, "00", 2) != 0)
```

```
        {
            if(strncmp(SQLSTATE, "02", 2) != 0)
                printf("SQLSTATE after fetch is %s\n", SQLSTATE);
            break;
        }
        /* Use the returned LO-pointer structure to open a smart
        * large object and get an LO file descriptor.
        */
        lofd = ifx_lo_open(&picture, LO_RDONLY, &error);
        if (error < 0)
        {
            strcpy(statement, "ifx_lo_open()");
            handle_lo_error(error);
        }
        if(ifx_lo_stat(lofd, &stats) < 0)
    {
            printf("\nifx_lo_stat() < 0");
            break;
        }
        if(ifx_int8toint(&c_num, &ic_num) !=   0)
            ic_num = 99999;
        if((ifx_lo_stat_size(stats, &size)) < 0)
            isize = 0;
        else
            if(ifx_int8toint(&size, &isize) !=   0)
            {
                printf("\nFailed to convert size");
                isize = 0;
            }
        if((refcnt = ifx_lo_stat_refcnt(stats)) < 0)
            refcnt = 0;
        printf("\n\nCatalog number %d", ic_num);
        printf("\nSize is %d, reference count is %d", isize, refcnt);

        if((atime = ifx_lo_stat_atime(stats)) < 0)
            printf("\nNo atime available");
        else
        {
            time = (time_t)atime;
            date_time = localtime(&time);
            printf("\nTime of last access: %s", asctime(date_time));
        }
        if((ctime = ifx_lo_stat_ctime(stats)) < 0)
            printf("\nNo ctime available");
        else
        {
            time = (time_t)ctime;
            date_time = localtime(&time);
            printf("Time of last change: %s", asctime(date_time));
        }

if((mtime = ifx_lo_stat_mtime_sec(stats)) < 0)
            printf("\nNo mtime available");
        else
        {
            time = (time_t)mtime;
            date_time = localtime(&time);
```

```
                printf("Time to the second of last modification: %s",
                    asctime(date_time));
            }
        if((cspec = ifx_lo_stat_cspec(stats)) == NULL)
            {
                printf("\nUnable to access ifx_lo_create_spec_t structure");
                break;
            }
            oflags = ifx_lo_specget_def_open_flags(cspec);
            printf("\nDefault open flags are: %d", oflags);
            if(ifx_lo_specget_estbytes(cspec, &estbytes) == -1)
            {
                printf("\nifx_lo_specget_estbytes() failed");
                break;
            }
            if(ifx_int8toint(&estbytes, &iebytes) !=    0)
            {
                printf("\nFailed to convert estimated bytes");
            }
            printf("\nEstimated size of smart LO is: %d", iebytes);
            if((extsz = ifx_lo_specget_extsz(cspec)) == -1)
        {
                printf("\nifx_lo_specget_extsz() failed");
                break;
            }
            printf("\nAllocation extent size of smart LO is: %d", extsz);
            if((cflags = ifx_lo_specget_flags(cspec)) == -1)
            {
                printf("\nifx_lo_specget_flags() failed");
                break;
            }
            printf("\nCreate-time flags of smart LO are: %d", cflags);
            if(ifx_lo_specget_maxbytes(cspec, &maxsize) == -1)
            {
                printf("\nifx_lo_specget_maxsize() failed");
                break;
            }
            if(ifx_int8toint(&maxsize, &imsize) !=    0)
            {
                printf("\nFailed to convert maximum size");
                break;
            }
            if(imsize == -1)
                printf("\nMaximum size of smart LO is: No limit");
            else
                printf("\nMaximum size of smart LO is: %d\n", imsize);
            if(ifx_lo_specget_sbspace(cspec, sbspc, sizeof(sbspc)) == -1)
                printf("\nFailed to obtain sbspace name");
            else
                printf("\nSbspace name is %s\n", sbspc);

    }
        /* Close smart large object */
        ifx_lo_close(lofd);
        ifx_lo_stat_free(stats);
      EXEC SQL close ifxcursor;
       EXEC SQL free ifxcursor;
```

```
}

void handle_lo_error(error_num)
int error_num;
{
        printf("%s generated error %d\n", statement, error_num);
        exit(1);
}

void nullterm(str)
char *str;
{
        char *end;

        end = str + 256;
        while(*str != ' ' && *str != '\0' && str < end)
        {
                ++str;
        }
        if(str >= end)
                printf("Error: end of str reached\n");
        if(*str == ' ')
                *str = '\0';
}
/* Include source code for whenexp_chk() exception-checking
* routine
*/

EXEC SQL include exp_chk.ec;
```

5.3.3 upd_lo_descr.ec 程序

此程序取得 **advert_descr** 列非空的目录项的价格，并将该价格追加到描述。

```
EXEC SQL include sqltypes;
EXEC SQL define BUFSZ 10;
extern char statement[80];
main()
{
        int error, isize;
        char format[] =    "<<<,<<<.&&";
        char decdsply[20], buf[50000], *end;

        EXEC SQL BEGIN DECLARE SECTION;
        dec_t price;
        fixed binary 'clob' ifx_lo_t descr;
        smallint stockno;
        char srvr_name[256], mancd[4], unit[5];
        ifx_lo_stat_t *stats;
        ifx_int8_t size, offset, pos;
        int lofd, ic_num;
        EXEC SQL END DECLARE SECTION;

        void nullterm(char *);
```

```
        void handle_lo_error(int);

        isize = 0;
        EXEC SQL whenever sqlerror call whenexp_chk;
        EXEC SQL whenever sqlwarning call whenexp_chk;

        printf("UPD_LO_DESCR Sample ESQL program running.\n\n");
        strcpy(statement, "CONNECT stmt");
        EXEC SQL connect to 'stores7';
        EXEC SQL get diagnostics exception 1
              :srvr_name = server_name;
        nullterm(srvr_name);

/* Selects each row where the advert.picure column is not null and
* displays
*    status information for the smart large object.
*/
        EXEC SQL declare ifxcursor cursor for
              select catalog_num, stock_num, manu_code, unit, advert_descr
              into :ic_num, :stockno, :mancd, :unit, :descr
              from catalog
              where advert_descr is not null;

EXEC SQL open ifxcursor;
        while(1)
        {
              EXEC SQL fetch ifxcursor;
              if (strncmp(SQLSTATE, "00", 2) != 0)
              {
                    if(strncmp(SQLSTATE, "02", 2) != 0)
                          printf("SQLSTATE after fetch is %s\n", SQLSTATE);
                    break;
             }
              EXEC SQL select unit_price into :price
                    from stock
                    where stock_num = :stockno
                    and manu_code = :mancd
                    and unit = :unit;
              if (strncmp(SQLSTATE, "00", 2) != 0)
              {
                    printf("SQLSTATE after select on stock: %s\n", SQLSTATE);
                    break;
              }
              if(risnull(CDECIMALTYPE, (char *) &price))    /* NULL? */
                    continue;            /* skip to next row */
              rfmtdec(&price, format, decdsply); /* format unit_price */
              /* Use the returned LO-pointer structure to open a smart
               * large object and get an LO file descriptor.
               */
```

```
            lofd = ifx_lo_open(&descr, LO_RDWR, &error);
            if (error < 0)
            {
                strcpy(statement, "ifx_lo_open()");
                handle_lo_error(error);
            }
            ifx_int8cvint(0, &offset);
            if(ifx_lo_seek(lofd, &offset, LO_SEEK_SET, &pos) < 0)
            {
                printf("\nifx_lo_seek() < 0\n");
                break;
            }
            if(ifx_lo_stat(lofd, &stats) < 0)
            {
                printf("\nifx_lo_stat() < 0");
                break;
            }
            if((ifx_lo_stat_size(stats, &size)) < 0)
            {
                printf("\nCan't get size, isize = 0");
                isize = 0;
            }
            else
                if(ifx_int8toint(&size, &isize) !=    0)
                {
                    printf("\nFailed to convert size");
                    isize = 0;
                }
            if(ifx_lo_read(lofd, buf, isize, &error) < 0)
            {
                printf("Read operation failed\n");
                break;
            }
end = buf + isize;
            strcpy(end++, "(");
            strcat(end, decdsply);
            end += strlen(decdsply);
            strcat(end++, ")");
            if(ifx_lo_writewithseek(lofd, buf, (end - buf), &offset,
            LO_SEEK_SET,
                &error) < 0)
            {
                printf("Write error on LO: %d", error);
                continue;
            }
          printf("\nNew description for catalog_num %d is: \n%s\n", ic_num,
          buf);
        }
        /* Close smart large object */
```

```
        ifx_lo_close(lofd);
        ifx_lo_stat_free(stats);
        /* Free LO-specification structure */
        EXEC SQL close ifxcursor;
        EXEC SQL free ifxcursor;
}

void handle_lo_error(error_num)
int error_num;
{
        printf("%s generated error %d\n", statement, error_num);
        exit(1);
}

void nullterm(str)
char *str;
{
        char *end;

        end = str + 256;
        while(*str != ' ' && *str != '\0' && str < end)
        {
                ++str;
        }
        if(str >= end)
                printf("Error: end of str reached\n");
        if(*str == ' ')
                *str = '\0';
}

/* Include source code for whenexp_chk() exception-checking
* routine
*/

EXEC SQL include exp_chk.ec;
```

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696