PostgreSQL 数据库执行 SQL 语句过程分析

目录

1　简单查询协议：exec_simple_query()函数

函数在文件 src/backend/tcop/postgres.c 中

1.1 环境准备

将本书的资源文件 university.sql 上传到 PostgreSQL 服务器的目录/home/postgres 下。

在#1 终端：

```
[postgres@dbsvr ~]$   cd /opt/db
[postgres@dbsvr db]$  rm -rf userdb
[postgres@dbsvr db]$  tar xf userdb.tar
[postgres@dbsvr db]$  cd
[postgres@dbsvr ~]$   pg_ctl start
（省略了一些输出）
server started
[postgres@dbsvr ~]$  psql -d postgres -U postgres -c "CREATE DATABASE universitydb;" -q
[postgres@dbsvr ~]$  psql -d universitydb -U postgres -f university.sql -q
[postgres@dbsvr ~]$
```

在#2 终端：

```
[postgres@dbsvr ~]$ psql -d universitydb -U postgres
psql (14.11)
Type "help" for help.

universitydb=#
```

在#1 终端：

```
[postgres@dbsvr ~]$  pgps
postgres   19798      1  0 23:25 ?        00:00:00 /opt/db/pg14/bin/postgres
postgres   19799  19798  0 23:25 ?        00:00:00 postgres: logger
postgres   19801  19798  0 23:25 ?        00:00:00 postgres: checkpointer
postgres   19802  19798  0 23:25 ?        00:00:00 postgres: background writer
postgres   19803  19798  0 23:25 ?        00:00:00 postgres: walwriter
postgres   19804  19798  0 23:25 ?        00:00:00 postgres: autovacuum launcher
postgres   19805  19798  0 23:25 ?        00:00:00 postgres: archiver
postgres   19806  19798  0 23:25 ?        00:00:00 postgres: stats collector
postgres   19807  19798  0 23:25 ?        00:00:00 postgres: logical replication launcher
```

```
    postgres   19831   19798  0 23:26 ?        00:00:00 postgres: postgres universitydb [local]
idle

    [postgres@dbsvr ~]$  gdb -p 19831 -q
    Attaching to process 19831
    Reading symbols from /opt/db/pg14/bin/postgres...
    Reading symbols from /usr/lib64/libm.so.6...
    Reading symbols from /usr/lib/debug//usr/lib64/libm.so.6-2.34-70.oe2203.x86_64.debug...
    Reading symbols from /usr/lib64/libc.so.6...
    Reading symbols from /usr/lib/debug//usr/lib64/libc.so.6-2.34-70.oe2203.x86_64.debug...
    Reading symbols from /lib64/ld-linux-x86-64.so.2...
    Reading symbols from /usr/lib/debug//lib64/ld-linux-x86-64.so.2-2.34-
70.oe2203.x86_64.debug...
    [Thread debugging using libthread_db enabled]
    Using host libthread_db library "/usr/lib64/libthread_db.so.1".
    0x00007fe1132b190a in epoll_wait (epfd=4, events=0x1cdf688, maxevents=1, timeout=-1)
at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
    30        return SYSCALL_CANCEL (epoll_wait, epfd, events, maxevents, timeout);
    (gdb) break exec_simple_query
    Breakpoint 1 at 0x9740a7: file postgres.c, line 957.
    (gdb) c
    Continuing.
```

在#2 终端：

```
    universitydb=# SELECT * FROM instructor;
    （执行中）
```

在#1 终端：

```
    Breakpoint 1, exec_simple_query (query_string=0x1ce4180 "SELECT * FROM instructor;") at
postgres.c:957
    957             CommandDest dest = whereToSendOutput;
    (gdb) list 949,970
    949     /*
    950      * exec_simple_query
    951      *
    952      * Execute a "simple Query" protocol message.
    953      */
    954     static void
    955     exec_simple_query(const char *query_string)
    956     {
    957             CommandDest dest = whereToSendOutput;
    958             MemoryContext oldcontext;
    959             List       *parsetree_list;
    960             ListCell   *parsetree_item;
    961             bool            save_log_statement_stats = log_statement_stats;
    962             bool            was_logged = false;
```

```
963             bool            use_implicit_block;
964             char            msec_str[32];
965
966             /*
967              * Report query to various monitoring facilities.
968              */
969             debug_query_string = query_string;
970
(gdb)
```

1.2 exec_simple_query()函数的注释、定义

```
(gdb) list 949,955
949     /*
950      * exec_simple_query
951      *
952      * Execute a "simple Query" protocol message.
953      */
954     static void
955     exec_simple_query(const char *query_string)
(gdb)
```

949-953: 这部分是一个多行注释，描述了 exec_simple_query 函数的主要功能。它指出该函数的目的是执行一个“简单查询”协议消息。在 PostgreSQL 中，“简单查询”是指客户端向服务器发送的一个单一的 SQL 语句，服务器执行这个语句并返回结果。这种类型的查询是通过 PostgreSQL 的前端/后端协议中定义的“简单查询”消息来处理的。

954-955: 定义了 exec_simple_query 函数，它是一个静态 void 函数，接受一个 const char *query_string 参数。这个参数 query_string 是指向包含 SQL 查询文本的字符串的指针。

static 关键字：表示 exec_simple_query 函数只能在定义它的源文件内部被调用，这是一种封装，防止函数在其他文件中被错误地使用或调用。

void 返回类型：意味着这个函数不返回任何值。

参数 const char *query_string：指明这个函数接受一个字符串参数，这个字符串包含要执行的 SQL 查询。const 关键字表示指向的字符串内容不会被这个函数修改，保证了输入查

询字符串的不可变性。

这个函数是 PostgreSQL 服务器响应客户端简单查询请求的核心部分之一，处理从客户端接收的 SQL 查询，执行这个查询，并负责生成并发送查询结果回客户端。这个过程包括解析查询、执行查询计划、收集和发送结果等多个步骤。

1.3 定义局部变量并初始化

```
(gdb) list 956,965
956     {
957             CommandDest dest = whereToSendOutput;
958             MemoryContext oldcontext;
959             List       *parsetree_list;
960             ListCell   *parsetree_item;
961             bool            save_log_statement_stats = log_statement_stats;
962             bool            was_logged = false;
963             bool            use_implicit_block;
964             char            msec_str[32];
965
(gdb)
```

957: 定义一个 CommandDest 类型的变量 dest，它代表了查询结果应该发送到哪里。whereToSendOutput 是一个全局变量，用于指示输出目的地（例如，客户端或日志）。

958: 定义一个 MemoryContext 类型的变量 oldcontext，用于保存当前内存上下文。在 PostgreSQL 中，内存上下文用于管理内存分配，确保在查询执行完成后可以正确清理内存。

959: 定义一个 List 类型的变量 parsetree_list，它将用于存储解析后的查询树。在 PostgreSQL 中，查询首先被解析成一种内部结构（解析树），然后这个树被进一步处理和执行。

960: 定义一个 ListCell 类型的指针 parsetree_item，用于遍历 parsetree_list。

961: 定义一个布尔类型的变量 save_log_statement_stats，并将其设置为全局变量 log_statement_stats 的值。这个变量用于指示是否记录查询的统计信息，如执行时间等。

962: 定义一个布尔类型的变量 was_logged，用于跟踪查询是否已经被记录到日志中。

963: 定义一个布尔类型的变量 use_implicit_block，在后面的代码中将用于决定是否对查询隐式地使用事务块。

964: 定义一个字符数组 msec_str，长度为 32。这个数组将用于格式化并存储表示查询

执行时间的毫秒值的字符串。

1.4 将查询报告给多种监控设施

```
(gdb) list 966,974
966             /*
967              * Report query to various monitoring facilities.
968              */
969             debug_query_string = query_string;
970
971             pgstat_report_activity(STATE_RUNNING, query_string);
972
973             TRACE_POSTGRESQL_QUERY_START(query_string);
974
(gdb)
```

966-967: 这是一个注释，解释了接下来几行代码的目的是将查询报告给多种监控设施。

969: 将全局变量 debug_query_string 设置为当前执行的查询字符串 query_string。这个变量在调试时很有用，因为它允许系统的其他部分访问当前正在执行的查询字符串。

971: 调用 pgstat_report_activity 函数，报告当前的活动状态为 STATE_RUNNING，并提供正在执行的查询字符串。这是 PostgreSQL 统计信息收集器的一部分，用于监控数据库活动，例如当前执行的查询。

973: 调用 TRACE_POSTGRESQL_QUERY_START 宏，这是 PostgreSQL 的跟踪工具，用于记录查询开始的时间。这在性能分析中非常有用，特别是当需要确定系统的哪一部分导致延迟时。

1.5 重置各种性能计数器，为当前查询的性能统计收集准备环境

```
(gdb) list 975,981
975             /*
976              * We use save_log_statement_stats so ShowUsage doesn't report incorrect
977              * results because ResetUsage wasn't called.
```

```
978             */
979            if (save_log_statement_stats)
980                    ResetUsage();
981
(gdb)
```

975-977: 这是一个注释，解释了接下来的代码使用 save_log_statement_stats 变量的原因。目的是确保 ShowUsage 函数不会因为 ResetUsage 没有被调用而报告不正确的结果。ShowUsage 和 ResetUsage 是 PostgreSQL 用来收集和重置性能统计信息的函数。

979-980: 如果 save_log_statement_stats 为真（意味着开始时 log_statement_stats 全局变量被设置为记录查询统计信息），则调用 ResetUsage 函数。这个函数重置各种性能计数器，为当前查询的性能统计收集准备环境。这样做是为了确保当前查询的统计信息是准确的，不会受到之前查询遗留数据的影响。

这一部分代码主要关注于在查询开始时对监控和统计信息收集机制的交互。通过这样的机制，PostgreSQL 能够提供关于数据库活动的详细信息，这对于数据库的性能调优和问题诊断至关重要。

1.6 开始一个事务命令

```
(gdb) list 982,990
982            /*
983             * Start up a transaction command.  All queries generated by the
984             * query_string will be in this same command block, *unless* we find a
985             * BEGIN/COMMIT/ABORT statement; we have to force a new xact command after
986             * one of those, else bad things will happen in xact.c. (Note that this
987             * will normally change current memory context.)
988             */
989            start_xact_command();
990
(gdb)
```

982-987: 介绍了开启一个事务命令的过程。所有由 query_string 生成的查询都将在相同的事务命令块中执行，除非遇到 BEGIN、COMMIT、ABORT 等事务控制语句。在这些语句

之后，必须强制开始一个新的事务命令，否则可能会在事务管理模块 xact.c 中发生错误。这通常会改变当前内存上下文。

989: 调用 start_xact_command 函数来开始一个事务命令。

1.7 删除之前的未命名语句

```
(gdb) list 991,998
991             /*
992              * Zap any pre-existing unnamed statement.  (While not strictly necessary,
993              * it seems best to define simple-Query mode as if it used the unnamed
994              * statement and portal; this ensures we recover any storage used by prior
995              * unnamed operations.)
996              */
997             drop_unnamed_stmt();
998
(gdb)
```

991-995: 清除任何已存在的未命名语句。为了定义简单查询模式，假设它使用了未命名的语句和门户，这样做可以回收之前未命名操作使用的任何存储空间。

997: 调用 drop_unnamed_stmt 函数来删除之前的未命名语句。

1.8 切换到适当的内存上下文以构建解析树

```
(gdb) list 999,1003
999             /*
1000             * Switch to appropriate context for constructing parsetrees.
1001             */
1002            oldcontext = MemoryContextSwitchTo(MessageContext);
1003
(gdb)
```

999-1002: 切换到适当的内存上下文以构建解析树。MessageContext 是用于构造解析树的内存上下文，oldcontext 用于保存切换前的上下文。

1.9 pg_parse_query()：解析查询字符串，返回解析树列表

```
(gdb) list 1004,1009
1004            /*
1005             * Do basic parsing of the query or queries (this should be safe even if
1006             * we are in aborted transaction state!)
1007             */
1008            parsetree_list = pg_parse_query(query_string);
1009
(gdb)
```

1004-1007: 执行基本的查询解析。即使在事务被中止的状态下，这一步骤也应该是安全的。

1008: 调用 pg_parse_query 函数解析查询字符串，返回解析树列表。

1.10 根据解析树列表记录查询（如果在参数文件配置了 log_statement）

```
(gdb) list 1010,1019
1010            /* Log immediately if dictated by log_statement */
1011            if (check_log_statement(parsetree_list))
1012            {
1013                    ereport(LOG,
1014                                    (errmsg("statement: %s", query_string),
1015                                     errhidestmt(true),
1016                                     errdetail_execute(parsetree_list)));
1017                    was_logged = true;
1018            }
1019
(gdb)
```

1010-1018: 如果 log_statement 配置指示立即记录，那么就记录这个查询。check_log_statement 函数检查是否应该根据解析树列表记录查询，如果是，则使用 ereport 记录查询字符串。

1.11 将内存上下文切换回执行查询之前的状态

```
(gdb) list 1020,1024
1020            /*
1021             * Switch back to transaction context to enter the loop.
1022             */
1023            MemoryContextSwitchTo(oldcontext);
1024
(gdb)
```

1020-1023: 切换回事务上下文以进入循环。这意味着将内存上下文切换回执行查询之前的状态。

1.12 如果列表中有多于一个的语句，则启用隐式事务块

```
(gdb) list 1025,1034
1025            /*
1026             * For historical reasons, if multiple SQL statements are given in a
1027             * single "simple Query" message, we execute them as a single transaction,
1028             * unless explicit transaction control commands are included to make
1029             * portions of the list be separate transactions.  To represent this
1030             * behavior properly in the transaction machinery, we use an "implicit"
1031             * transaction block.
1032             */
1033            use_implicit_block = (list_length(parsetree_list) > 1);
1034
(gdb)
```

1025-1031: 为了兼容历史原因，如果在一个简单查询消息中给出了多个 SQL 语句，它们将作为一个单一的事务执行，除非包含了显式的事务控制命令（如 BEGIN、COMMIT 或 ABORT），这些命令将使得列表中的部分内容被视为独立的事务。为了在事务机制中正确表示这种行为，使用了所谓的“隐式”事务块。

1032-1033: 定义了一个布尔变量 use_implicit_block，其值根据解析后的查询列表 parsetree_list 的长度来设定。如果列表中有多于一个的语句，则启用隐式事务块。这反映了上述注释中描述的行为：多条语句将被视为一个事务执行，除非通过事务控制语句明确指定了分割。

1.13 遍历原始解析树列表并处理每一个

```
(gdb) list 1035,1283
1035            /*
1036             * Run through the raw parsetree(s) and process each one.
1037             */
1038            foreach(parsetree_item, parsetree_list)
1039            {
```

1035-1037: 注释指出，接下来的代码将遍历原始解析树列表并处理每一个。

1038-1039: 使用 foreach 循环遍历 parsetree_list，这是一个包含一个或多个解析后的 SQL 语句的列表。

1.13.1 获取当前项的解析树

```
1040                    RawStmt    *parsetree = lfirst_node(RawStmt, parsetree_item);
```

1040: 获取当前项的解析树（RawStmt 类型），准备进行进一步处理。

1.13.2 声明变量

```
1041                    bool            snapshot_set = false;
1042                    CommandTag      commandTag;
1043                    QueryCompletion qc;
1044                    MemoryContext per_parsetree_context = NULL;
1045                    List       *querytree_list,
1046                                  *plantree_list;
1047                    Portal          portal;
1048                    DestReceiver *receiver;
```

```
1049                int16           format;
1050
```

1041: 声明一个布尔变量 snapshot_set，用于跟踪是否为解析/计划阶段设置了快照。

1042-1043: 声明 commandTag 和 QueryCompletion 类型的变量，分别用于存储命令标签和查询完成状态。

1044: 声明一个 MemoryContext 类型的变量 per_parsetree_context，用于每个解析树的内存上下文。

1045-1046: 声明两个列表变量 querytree_list 和 plantree_list，分别用于存储查询树和计划树。

1047-1049: 声明 portal 和 receiver 变量，用于处理查询结果的输出。format 用于指定结果的格式。

1.13.3 当前查询没有关联的统计信息 ID

```
1051                pgstat_report_query_id(0, true);
1052
```

1051: 通过 pgstat_report_query_id 报告查询 ID 为 0，这意味着当前查询没有关联的统计信息 ID。

1.13.4 根据解析树的语句类型生成命令标签

```
1053                /*
1054                 * Get the command name for use in status display (it also becomes the
1055                 * default completion tag, down inside PortalRun).  Set ps_status and
1056                 * do any special start-of-SQL-command processing needed by the
1057                 * destination.
1058                 */
1059                commandTag = CreateCommandTag(parsetree->stmt);
1060
1061                set_ps_display(GetCommandTagName(commandTag));
1062
```

1059: 通过调用 CreateCommandTag 根据解析树的语句类型生成命令标签。

1061: 设置进程状态显示为当前命令标签的名称。

1.13.5 开始命令执行

```
1063                    BeginCommand(commandTag, dest);
1064
```

1063: 调用 BeginCommand 以开始命令执行，向客户端或其他目的地发送适当的开始执行信号。

1.13.6 如果当前处于中止事务的状态，并且当前语句不是事务结束语句（COMMIT/ABORT），则报错

```
1065                    /*
1066                     * If we are in an aborted transaction, reject all commands except
1067                     * COMMIT/ABORT.  It is important that this test occur before we try
1068                     * to do parse analysis, rewrite, or planning, since all those phases
1069                     * try to do database accesses, which may fail in abort state. (It
1070                     * might be safe to allow some additional utility commands in this
1071                     * state, but not many...)
1072                     */
1073                    if (IsAbortedTransactionBlockState() &&
1074                            !IsTransactionExitStmt(parsetree->stmt))
1075                            ereport(ERROR,
1076                                            (errcode(ERRCODE_IN_FAILED_SQL_TRANSACTION),
1077                                             errmsg("current transaction is aborted, "
1078                                                            "commands ignored until end of
transaction block"),
1079                                             errdetail_abort()));
1080
```

1073-1079: 如果当前处于中止事务的状态，并且当前语句不是事务结束语句（COMMIT/ABORT），则报错，因为在已中止的事务中不允许执行其他命令。

1.13.7　确保我们处于事务命令中

```
1081                    /* Make sure we are in a transaction command */
1082                    start_xact_command();
1083
```

1081: 确保我们处于事务命令中。

1.13.8　如果使用隐式事务块并且当前不在事务块中，则开始一个隐式事务块

```
1084                    /*
1085                     * If using an implicit transaction block, and we're not already in a
1086                     * transaction block, start an implicit block to force this statement
--Type <RET> for more, q to quit, c to continue without paging--c
1087                     * to be grouped together with any following ones.  (We must do this
1088                     * each time through the loop; otherwise, a COMMIT/ROLLBACK in the
1089                     * list would cause later statements to not be grouped.)
1090                     */
1091                    if (use_implicit_block)
1092                            BeginImplicitTransactionBlock();
1093
```

1091-1092: 如果使用隐式事务块并且当前不在事务块中，则开始一个隐式事务块。

1.13.9　检查是否有中断信号

```
1094                    /* If we got a cancel signal in parsing or prior command, quit */
1095                    CHECK_FOR_INTERRUPTS();
```

```
1096
```

1095: 检查是否有中断信号，如取消请求。

1.13.10 如果解析/计划需要快照，则推送一个活动快照

```
1097                    /*
1098                     * Set up a snapshot if parse analysis/planning will need one.
1099                     */
1100                    if (analyze_requires_snapshot(parsetree))
1101                    {
1102                            PushActiveSnapshot(GetTransactionSnapshot());
1103                            snapshot_set = true;
1104                    }
1105
```

1100-1104: 如果解析/计划需要快照，则推送一个活动快照。

1.13.11 根据是否还有更多的解析树项来切换内存上下文

```
1106                    /*
1107                     * OK to analyze, rewrite, and plan this query.
1108                     *
1109                     * Switch to appropriate context for constructing query and plan trees
1110                     * (these can't be in the transaction context, as that will get reset
1111                     * when the command is COMMIT/ROLLBACK).  If we have multiple
1112                     * parsetrees, we use a separate context for each one, so that we can
1113                     * free that memory before moving on to the next one.  But for the
1114                     * last (or only) parsetree, just use MessageContext, which will be
1115                     * reset shortly after completion anyway.  In event of an error, the
1116                     * per_parsetree_context will be deleted when MessageContext is reset.
1117                     */
1118                    if (lnext(parsetree_list, parsetree_item) != NULL)
1119                    {
1120                            per_parsetree_context =
1121                                    AllocSetContextCreate(MessageContext,
```

```
1122                                                    "per-parsetree message context",
1123                                                    ALLOCSET_DEFAULT_SIZES);
1124                    oldcontext = MemoryContextSwitchTo(per_parsetree_context);
1125            }
1126            else
1127                    oldcontext = MemoryContextSwitchTo(MessageContext);
1128
```

1118-1127: 根据是否还有更多的解析树项来切换内存上下文，以优化内存使用。

1.13.12 pg_analyze_and_rewrite()：分析并重写查询

```
1129            querytree_list = pg_analyze_and_rewrite(parsetree, query_string,
1130                                                    NULL, 0, NULL);
1131
```

1129-1131: 调用 pg_analyze_and_rewrite 分析并重写查询

1.13.13 pg_plan_queries ()：生成执行计划

```
1132            plantree_list = pg_plan_queries(querytree_list, query_string,
1133                                            CURSOR_OPT_PARALLEL_OK, NULL);
1134
```

1132-1134: 调用 pg_plan_queries 生成执行计划。

1.13.14 如果设置了快照，则弹出活动快照。

```
1135            /*
1136             * Done with the snapshot used for parsing/planning.
1137             *
1138             * While it looks promising to reuse the same snapshot for query
```

```
1139                 * execution (at least for simple protocol), unfortunately it causes
1140                 * execution to use a snapshot that has been acquired before locking
1141                 * any of the tables mentioned in the query.  This creates user-
1142                 * visible anomalies, so refrain.  Refer to
1143                 * https://postgr.es/m/flat/5075D8DF.6050500@fuzzy.cz for details.
1144                 */
1145                if (snapshot_set)
1146                        PopActiveSnapshot();
1147
```

1145-1146: 完成快照使用后，如果设置了快照，则弹出活动快照。

1.13.15 如果再分析或计划生成过程中收到取消信号，则退出

```
1148                /* If we got a cancel signal in analysis or planning, quit */
1149                CHECK_FOR_INTERRUPTS();
1150
```

1.13.16 CreatePortal()：创建一个无名 Portal，用于执行查询

```
1151                /*
1152                 * Create unnamed portal to run the query or queries in. If there
1153                 * already is one, silently drop it.
1154                 */
1155                portal = CreatePortal("", true, true);
1156                /* Don't display the portal in pg_cursors */
1157                portal->visible = false;
1158
```

1155-1158: 创建一个未命名的 Portal，用于执行查询。设置门户不在 pg_cursors 中显示。

门户是 PostgreSQL 用于执行 SQL 查询的一种结构，它不仅包含了查询本身，还包含了查询执行所需的所有上下文信息。

1.13.17　PortalDefineQuery()：定义一个查询门户（Portal）

```
1159                /*
1160                 * We don't have to copy anything into the portal, because everything
1161                 * we are passing here is in MessageContext or the
1162                 * per_parsetree_context, and so will outlive the portal anyway.
1163                 */
1164                PortalDefineQuery(portal,
1165                                                    NULL,
1166                                                    query_string,
1167                                                    commandTag,
1168                                                    plantree_list,
1169                                                    NULL);
1170
```

这段代码位于 PostgreSQL 的 exec_simple_query 函数中，负责定义（设置或配置）一个查询门户（Portal）。门户是 PostgreSQL 用于执行 SQL 查询的一种结构，它不仅包含了查询本身，还包含了查询执行所需的所有上下文信息。下面是对这段代码的逐行解释：

1159-1162: 注释解释了为什么在定义门户时不需要复制任何内容。这是因为所有传递给门户的内容都位于 MessageContext 或 per_parsetree_context 中。这两个内存上下文的生命周期都超过了门户本身，因此，门户中使用的数据将会在门户存在期间一直有效，不需要额外的复制操作。

1164-1169: 调用 PortalDefineQuery 函数为当前门户定义一个查询。这个函数的参数包括：

portal：当前操作的门户对象。

第一个 NULL：指定命名空间（在这种情况下不适用，因为是未命名门户）。

query_string：执行的 SQL 查询字符串。

commandTag：该查询的命令标签，它是一个枚举值，表示 SQL 命令的类型（如 SELECT、UPDATE 等）。

plantree_list：该查询的计划树列表，包含了解析和优化后的查询计划，这是执行查询的实际指令。

第二个 NULL：参数列表，用于参数化查询（在这个场景中不适用，因此为 NULL）。

通过 PortalDefineQuery，门户被赋予了执行一个特定查询所需的所有信息，包括查询字符串、查询的类型以及如何执行该查询的计划。

这个步骤是 PostgreSQL 查询处理流程中设置执行环境的关键环节之一，确保了查询能够按照预期被正确执行。

1.13.18 PortalStart()：启动 Portal

```
1171                    /*
1172                     * Start the portal.  No parameters here.
1173                     */
1174                    PortalStart(portal, NULL, 0, InvalidSnapshot);
1175
```

1174: 使用 PortalStart 启动门户，准备执行。

1.13.19 设置结果格式

```
1176                    /*
1177                     * Select the appropriate output format: text unless we are doing a
1178                     * FETCH from a binary cursor.  (Pretty grotty to have to do this
here
1179                     * --- but it avoids grottiness in other places.  Ah, the joys of
1180                     * backward compatibility...)
1181                     */
1182                    format = 0;                             /* TEXT is default */
1183                    if (IsA(parsetree->stmt, FetchStmt))
1184                    {
1185                            FetchStmt  *stmt = (FetchStmt *) parsetree->stmt;
1186
1187                            if (!stmt->ismove)
1188                            {
1189                                    Portal          fportal =
GetPortalByName(stmt->portalname);
1190
1191                                    if (PortalIsValid(fportal) &&
1192                                            (fportal->cursorOptions & CURSOR_OPT_BINARY))
1193                                            format = 1; /* BINARY */
1194                            }
1195                    }
1196                    PortalSetResultFormat(portal, 1, &format);
1197
```

1182-1196: 设置结果格式，默认为文本。

如果是从二进制游标中提取（FETCH），并且游标选项为二进制，则改为二进制格式。

1.13.20 创建目的地接收器（DestReceiver）

```
1198                    /*
1199                     * Now we can create the destination receiver object.
1200                     */
1201                    receiver = CreateDestReceiver(dest);
1202                    if (dest == DestRemote)
1203                            SetRemoteDestReceiverParams(receiver, portal);
1204
```

1201-1203: 创建目的地接收器（DestReceiver），用于处理查询结果的输出。

1.13.21 执行前，切回到事务上下文

```
1205                    /*
1206                     * Switch back to transaction context for execution.
1207                     */
1208                    MemoryContextSwitchTo(oldcontext);
1209
```

1208: 在执行前，切回到事务上下文。

1.13.22 PortalRun()

```
1210                /*
1211                 * Run the portal to completion, and then drop it (and the receiver).
1212                 */
1213                (void) PortalRun(portal,
1214                                                    FETCH_ALL,
1215                                                    true,  /* always top level */
1216                                                    true,
1217                                                    receiver,
1218                                                    receiver,
1219                                                    &qc);
1220
```

1213-1219: 使用 PortalRun 运行门户直至完成，处理查询结果。

1.13.23 销毁接收器

```
1221                receiver->rDestroy(receiver);
1222
```

1221-1222: 销毁接收器。

1.13.24 PortalDrop()：丢弃门户

```
1223                PortalDrop(portal, false);
1224
```

1223: 丢弃门户。

1.13.25　如果当前处理的解析树是查询字符串中的最后一个

```
1225                if (lnext(parsetree_list, parsetree_item) == NULL)
1226                {
1227                        /*
1228                         * If this is the last parsetree of the query string, close down
1229                         * transaction statement before reporting command-complete.  This
1230                         * is so that any end-of-transaction errors are reported before
1231                         * the command-complete message is issued, to avoid confusing
1232                         * clients who will expect either a command-complete message or an
1233                         * error, not one and then the other.  Also, if we're using an
1234                         * implicit transaction block, we must close that out first.
1235                         */
1236                        if (use_implicit_block)
1237                                EndImplicitTransactionBlock();
1238                        finish_xact_command();
1239                }
```

1225-1239: **如果当前处理的解析树是查询字符串中的最后一个**：

如果使用了隐式事务块，则：

首先关闭隐式事务块，

然后调用 finish_xact_command 完成事务命令。

这是为了确保任何事务结束时的错误在命令完成消息之前报告，避免客户端混淆。

1.13.26　如果当前解析的语句是事务控制语句

```
1240                else if (IsA(parsetree->stmt, TransactionStmt))
1241                {
1242                        /*
1243                         * If this was a transaction control statement, commit it. We will
1244                         * start a new xact command for the next command.
1245                         */
1246                        finish_xact_command();
1247                }
```

1240-1247: **如果当前解析的语句是事务控制语句**（如 BEGIN, COMMIT, ROLLBACK），

则同样调用 finish_xact_command 来完成事务命令。这意味着，对于事务控制语句，即使它们是在一个查询字符串中一起提交的，也会被立即执行并完成。

1.13.27 如果当前解析的语句是非事务控制语句

```
1248                else
1249                {
1250                        /*
1251                         * We had better not see XACT_FLAGS_NEEDIMMEDIATECOMMIT set if
1252                         * we're not calling finish_xact_command().  (The implicit
1253                         * transaction block should have prevented it from getting set.)
1254                         */
1255                        Assert(!(MyXactFlags & XACT_FLAGS_NEEDIMMEDIATECOMMIT));
1256
1257                        /*
1258                         * We need a CommandCounterIncrement after every query, except
1259                         * those that start or end a transaction block.
1260                         */
1261                        CommandCounterIncrement();
1262
1263                        /*
1264                         * Disable statement timeout between queries of a multi-query
1265                         * string, so that the timeout applies separately to each query.
1266                         * (Our next loop iteration will start a fresh timeout.)
1267                         */
1268                        disable_statement_timeout();
1269                }
1270
```

1248-1268: 对于**非事务控制语句**，进行以下操作：

- 断言不应当设置 XACT_FLAGS_NEEDIMMEDIATECOMMIT 标志，因为没有调用 finish_xact_command（隐式事务块应该防止了它的设置）。
- 调用 CommandCounterIncrement 来增加命令计数器，这在每个查询后都是必需的，除非那个查询开始或结束了一个事务块。这确保了数据库状态的更新对后续的查询可见。
- 调用 disable_statement_timeout 禁用语句超时，这样多查询字符串中的每个查询都有独立的超时设置。在下一个循环迭代中将开始一个新的超时。

1.13.28　向客户端报告当前查询已完成

```
    1271                    /*
    1272                     * Tell client that we're done with this query.  Note we emit exactly
    1273                     * one EndCommand report for each raw parsetree, thus one for each
SQL
    1274                     * command the client sent, regardless of rewriting. (But a command
    1275                     * aborted by error will not send an EndCommand report at all.)
    1276                     */
    1277                    EndCommand(&qc, dest, false);
    1278
```

1271-1277: 向客户端报告当前查询已完成。对于每个原始解析树（即客户端发送的每个 SQL 命令），发出一个 EndCommand 报告，无论是否进行了重写。如果命令因错误而中止，则不会发送 EndCommand 报告。

1.13.29　如果为当前解析树创建了一个 per-parsetree 上下文，则删除它

```
    1279                    /* Now we may drop the per-parsetree context, if one was created. */
    1280                    if (per_parsetree_context)
    1281                            MemoryContextDelete(per_parsetree_context);
```

1279-1281: 如果为当前解析树创建了一个 per-parsetree 上下文，则删除它。这是查询处理的最后一步，目的是在处理下一个查询之前释放任何占用的内存。

1.13.30　结束遍历解析树列表的循环

```
    1282            }                                               /* end loop over parsetrees */
    1283
    (gdb)
```

1282-1283: 结束遍历解析树列表的循环。

1.14 如果打开了一个事务语句（例如，通过 BEGIN 命令），则尝试关闭它。

```
(gdb) list 1284,1323
1284            /*
1285             * Close down transaction statement, if one is open.  (This will only do
1286             * something if the parsetree list was empty; otherwise the last loop
1287             * iteration already did it.)
1288             */
1289            finish_xact_command();
1290
```

1284-1289: 如果打开了一个事务语句（例如，通过 BEGIN 命令），则尝试关闭它。如果解析树列表为空，即没有执行任何 SQL 命令，这将会是必要的操作，因为在这种情况下之前的循环（处理每个解析树）不会执行 finish_xact_command。

1.15 如果 parsetree_list 为空，给客户端返回 EmptyQueryResponse 消息

```
1291            /*
1292             * If there were no parsetrees, return EmptyQueryResponse message.
1293             */
1294            if (!parsetree_list)
1295                    NullCommand(dest);
1296
```

1291-1295: 如果 parsetree_list 为空，即没有解析出任何 SQL 命令，这可能意味着客户端发送了一个空的查询请求，函数将返回一个 EmptyQueryResponse 消息给客户端。这是通过调用 NullCommand 函数完成的。

1.16　如果配置了日志持续时间（log_duration）、查询执行时间日志或者是基于执行时间的自动日志记录，根据 check_log_duration 函数的返回值，使用 ereport 记录相应的日志

```
1297        /*
1298         * Emit duration logging if appropriate.
1299         */
1300        switch (check_log_duration(msec_str, was_logged))
1301        {
1302                case 1:
1303                        ereport(LOG,
1304                                        (errmsg("duration: %s ms", msec_str),
1305                                         errhidestmt(true)));
1306                        break;
1307                case 2:
1308                        ereport(LOG,
1309                                        (errmsg("duration: %s ms  statement: %s",
1310                                                        msec_str, query_string),
1311                                         errhidestmt(true),
1312                                         errdetail_execute(parsetree_list)));
1313                        break;
1314        }
1315
```

1298-1313: 如果配置了日志持续时间（log_duration）、查询执行时间日志或者是基于执行时间的自动日志记录，根据 check_log_duration 函数的返回值（1 表示只记录持续时间，2 表示记录持续时间和查询语句），使用 ereport 记录相应的日志。msec_str 包含了查询执行时间的字符串表示，query_string 是执行的 SQL 语句文本。

1.17　显示记录查询统计信息

```
1316        if (save_log_statement_stats)
1317                ShowUsage("QUERY STATISTICS");
```

```
1318
```

1316-1317: 如果在开始时 save_log_statement_stats 被设置为 true，即配置了记录查询统计信息（log_statement_stats），则调用 ShowUsage 函数显示这些统计信息。这些信息包括时间、内存使用等统计数据，对于性能分析非常有用。

1.18　标记查询执行结束

```
1319            TRACE_POSTGRESQL_QUERY_DONE(query_string);
1320
```

1319: 调用 TRACE_POSTGRESQL_QUERY_DONE 宏，标记查询执行结束。这是 PostgreSQL 内部跟踪工具的一部分，用于记录查询的结束时间点。

1.19　清除变量 debug_query_string（存储当前执行的查询字符串）

```
1321            debug_query_string = NULL;
```

1321: 将 debug_query_string 设置为 NULL。debug_query_string 在查询执行过程中用于存储当前执行的查询字符串，执行结束后清除这个变量。

1.20　函数结束

```
1322    }
1323
(gdb)
```

2 扩展查询协议

扩展查询协议（Extended query protocol）将查询的处理流程分为若干步骤，以达到执行计划复用的目的。每一步都由单独的服务端消息进行确认（但是服务端消息可以连续发送，无需等待）。

扩展查询协议允许查询中带参数，如：select * from student where id =$var；不允许一个查询中包含多条 SQL 命令，如：select * from company;delete from company;

扩展查询协议通常包括的五个阶段：

- Parse
- Bind
- Describe
- Execute
- Sync

2.1 JDBC

要使用扩展查询协议访问 PostgreSQL，你通常需要一个支持该协议的客户端库。扩展查询协议允许客户端以两个阶段执行查询：首先是语句的准备（预编译），然后是执行。这样做的好处包括减少网络开销（对于重复执行的查询）、提高性能（通过预编译）以及减少 SQL 注入的风险。

2.1.1 方法 1：编译 PostgreSQL JDBC 驱动

```
[postgres@dbsvr ~]$ cd /opt/db/soft
[postgres@dbsvr soft]$ git clone https://github.com/pgjdbc/pgjdbc.git
Cloning into 'pgjdbc'...
remote: Enumerating objects: 47433, done.
remote: Counting objects: 100% (32/32), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 47433 (delta 7), reused 30 (delta 7), pack-reused 47401
```

```
Receiving objects: 100% (47433/47433), 43.43 MiB | 8.12 MiB/s, done.
Resolving deltas: 100% (29168/29168), done.
[postgres@dbsvr soft]$ cd pgjdbc/
[postgres@dbsvr pgjdbc]$ git tag
(省略了许多输出)
v42.3.0-rc5
[postgres@dbsvr soft]$
[postgres@dbsvr pgjdbc]$ git checkout tags/v42.3.0-rc5
Note: switching to 'tags/v42.3.0-rc5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at a7d6e00b travis: copr
[postgres@dbsvr pgjdbc]$ mvn clean package
```

mvn clean package
mvn clean package -DskipTests

[root@dbsvr target]# pwd
/opt/pgjdbc/pgjdbc/target
[root@dbsvr target]# ls -l
total 1728
drwxr-xr-x 4 root root 4096 Mar 30 04:39 classes
drwxr-xr-x 2 root root 4096 Mar 30 04:40 feature
drwxr-xr-x 3 root root 4096 Mar 30 04:39 generated-sources
drwxr-xr-x 3 root root 4096 Mar 30 04:39 generated-test-sources
drwxr-xr-x 3 root root 4096 Mar 30 04:39 gen-src
drwxr-xr-x 3 root root 4096 Mar 30 04:39 gen-test-src
drwxr-xr-x 3 root root 4096 Mar 30 04:39 maven-status
-rw-r--r-- 1 root root 799636 Apr 4 2024 original-postgresql-42.2.11-SNAPSHOT.jar
-rw-r--r-- 1 root root 930102 Mar 30 04:40 postgresql-42.2.11-SNAPSHOT.jar
drwxr-xr-x 3 root root 4096 Mar 30 04:39 test-classes
[root@dbsvr target]#

编译

[postgres@dbsvr ~]$

javac -cp .:postgresql-42.2.11-SNAPSHOT.jar JdbcPreparedStatementExample.java

运行

[postgres@dbsvr ~]$

java -cp .:postgresql-42.2.11-SNAPSHOT.jar JdbcPreparedStatementExample

2.1.2 方法 2：下载 PostgreSQL JDBC 驱动

```
[postgres@dbsvr ~]$ wget https://jdbc.postgresql.org/download/postgresql-42.7.3.jar
--2024-03-29 15:06:56--  https://jdbc.postgresql.org/download/postgresql-42.7.3.jar
Resolving jdbc.postgresql.org (jdbc.postgresql.org)... 72.32.157.228, 2001:4800:3e1:1::228
Connecting to jdbc.postgresql.org (jdbc.postgresql.org)|72.32.157.228|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1089312 (1.0M) [application/java-archive]
Saving to: ‘postgresql-42.7.3.jar’

postgresql-42.7.3.jar         100%[=========================>]   1.04M   837KB/s    in 1.3s

2024-03-29 15:06:58 (837 KB/s) - ‘postgresql-42.7.3.jar’ saved [1089312/1089312]

[postgres@dbsvr ~]$
```

2.1.3 JDBC 示例代码

[postgres@dbsvr ~]$ vi JdbcPreparedStatementExample.java

添加以下内容：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Scanner; // 导入 Scanner 类

public class JdbcPreparedStatementExample {
```

```
    public static void main(String[] args) {
        // 使用你提供的数据库连接参数
        String url = "jdbc:postgresql://localhost/universitydb"; // 根据你的设置，host 是 localhost，数据库名是 universitydb
        String user = "postgres";
        String password = "dba123";

        Scanner scanner = new Scanner(System.in); // 创建 Scanner 对象

        try {
            // 加载 PostgreSQL JDBC 驱动
            Class.forName("org.postgresql.Driver");

            // 创建连接
            Connection conn = DriverManager.getConnection(url, user, password);

            while (true) { // 开始循环
                // 创建预编译的 Statement 对象
                String query = "SELECT * FROM instructor WHERE id = ?";
                PreparedStatement pstmt = conn.prepareStatement(query);

                // 提示用户输入 id
                System.out.print("Enter instructor ID: ");
                String id = scanner.nextLine(); // 读取用户输入的 id

                // 设置参数
                pstmt.setString(1, id); // 使用用户输入的 id 作为查询参数

                // 执行查询
                ResultSet rs = pstmt.executeQuery();

                // 处理查询结果
                while (rs.next()) {
                    System.out.println("Name: " + rs.getString("name") + ", Department: " + rs.getString("dept_name") + ", Salary: " + rs.getDouble("salary"));
                }

                // 关闭结果集和语句
                rs.close();
                pstmt.close();

                // 提示用户是否继续
                System.out.print("Do you want to continue? (yes/no): ");
                String answer = scanner.nextLine();
```

```
                    if (!answer.equalsIgnoreCase("yes")) {
                        break; // 如果用户不输入 yes，则退出循环
                    }
                }

                // 关闭连接
                conn.close();
            } catch (ClassNotFoundException e) {
                System.out.println("PostgreSQL JDBC Driver not found.");
                e.printStackTrace();
            } catch (SQLException e) {
                System.out.println("Connection failure.");
                e.printStackTrace();
            } finally {
                scanner.close(); // 关闭 Scanner 对象
            }
        }
}
```

2.1.4 编译 JDBC 示例代码

```
[postgres@dbsvr ~]$ javac -cp .:postgresql-42.7.3.jar JdbcPreparedStatementExample.java
[postgres@dbsvr ~]$ ls -l JdbcPreparedStatementExample.class
-rw-r--r-- 1 postgres dba 2606 Mar 29 15:10 JdbcPreparedStatementExample.class
[postgres@dbsvr ~]$
```

2.1.5 运行 JDBC 示例程序

```
[postgres@dbsvr ~]$ java -cp .:postgresql-42.7.3.jar JdbcPreparedStatementExample
Enter instructor ID: 12121
Name: Wu, Department: Finance, Salary: 90000.0
Do you want to continue? (yes/no): yes
Enter instructor ID: 10101
Name: Srinivasan, Department: Comp. Sci., Salary: 65000.0
Do you want to continue? (yes/no): no
[postgres@dbsvr ~]$
```

2.1.6　跟踪调试

在#2 上执行下面的代码：

```
[postgres@dbsvr ~]$ java -cp .:postgresql-42.7.3.jar JdbcPreparedStatementExample
Enter instructor ID:
（等待输入教师 ID 值）
```

在#1 上执行下面的代码

```
[postgres@dbsvr ~]$ pgps
postgres  114626       1  0 Mar28 ?        00:00:01 /opt/db/pg14/bin/postgres
postgres  114627  114626  0 Mar28 ?        00:00:00 postgres: logger
postgres  114629  114626  0 Mar28 ?        00:00:00 postgres: checkpointer
postgres  114630  114626  0 Mar28 ?        00:00:00 postgres: background writer
postgres  114631  114626  0 Mar28 ?        00:00:00 postgres: walwriter
postgres  114632  114626  0 Mar28 ?        00:00:01 postgres: autovacuum launcher
postgres  114633  114626  0 Mar28 ?        00:00:00 postgres: archiver last was
000000010000000000000004
postgres  114634  114626  0 Mar28 ?        00:00:04 postgres: stats collector
postgres  114635  114626  0 Mar28 ?        00:00:00 postgres: logical replication launcher
postgres  201843  114626  0 15:13 ?        00:00:00 postgres: postgres universitydb
127.0.0.1(13718) idle
[postgres@dbsvr ~]$

[postgres@dbsvr ~]$ gdb -p 201843 -q
Attaching to process 201843
Reading symbols from /opt/db/pg14/bin/postgres...
Reading symbols from /usr/lib64/libssl.so.1.1...
Reading symbols from /usr/lib/debug//usr/lib64/libssl.so.1.1.1m-1.1.1m-
26.oe2203.x86_64.debug...
Reading symbols from /usr/lib64/libcrypto.so.1.1...
Reading symbols from /usr/lib/debug//usr/lib64/libcrypto.so.1.1.1m-1.1.1m-
26.oe2203.x86_64.debug...
Reading symbols from /usr/lib64/libm.so.6...
Reading symbols from /usr/lib/debug//usr/lib64/libm.so.6-2.34-70.oe2203.x86_64.debug...
Reading symbols from /usr/lib64/libc.so.6...
Reading symbols from /usr/lib/debug//usr/lib64/libc.so.6-2.34-70.oe2203.x86_64.debug...
Reading symbols from /usr/lib64/libz.so.1...
Reading symbols from /usr/lib/debug//usr/lib64/libz.so.1.2.11-1.2.11-
24.oe2203.x86_64.debug...
Reading symbols from /lib64/ld-linux-x86-64.so.2...
Reading symbols from /usr/lib/debug//lib64/ld-linux-x86-64.so.2-2.34-
70.oe2203.x86_64.debug...
```

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib64/libthread_db.so.1".
0x00007f67f5fcd90a in epoll_wait (epfd=4, events=0x29b7688, maxevents=1, timeout=-1)
at ../sysdeps/unix/sysv/linux/epoll_wait.c:30
30        return SYSCALL_CANCEL (epoll_wait, epfd, events, maxevents, timeout);
(gdb) break exec_parse_message
(gdb) break exec_bind_message
(gdb) break exec_describe_statement_message
(gdb) break exec_describe_portal_message
(gdb) break exec_execute_message
(gdb) break finish_xact_command
```

在#2 上执行

```
Enter instructor ID: 10101
```

在#1 上执行

```
(gdb) c
Continuing.

Breakpoint 1, exec_parse_message (query_string=0x29bd131 "SELECT * FROM instructor WHERE id
= $1", stmt_name=0x29bd130 "", paramTypes=0x29bd998, numParams=1) at postgres.c:1335
1335            MemoryContext unnamed_stmt_context = NULL;
(gdb) c
Continuing.

Breakpoint 2, exec_bind_message (input_message=0x7fff7c76a840) at postgres.c:1592
1592            int16      *pformats = NULL;
(gdb) c
Continuing.

Breakpoint 4, exec_describe_portal_message (portal_name=0x29bd131 "") at postgres.c:2655
2655            start_xact_command();
(gdb) c
Continuing.

Breakpoint 5, exec_execute_message (portal_name=0x29bd130 "", max_rows=0) at postgres.c:2059
2059            bool            save_log_statement_stats = log_statement_stats;
(gdb) c
Continuing.

Breakpoint 6, finish_xact_command () at postgres.c:2730
2730            disable_statement_timeout();
(gdb) c
Continuing.
```

在#2 终端，执行如下：

```
Name: Srinivasan, Department: Comp. Sci., Salary: 65000.0
Do you want to continue? (yes/no): no
```

在#1 终端，执行如下：

```
[Inferior 1 (process 201843) exited normally]
(gdb) q
[postgres@dbsvr ~]$
```

2.2 ODBC

2.2.1 下载 PostgreSQL DDBC 驱动源代码

https://www.postgresql.org/download/

源代码

https://www.postgresql.org/ftp/odbc/versions/src/

https://www.postgresql.org/ftp/odbc/versions/src/

2.2.2 编译安装 PostgreSQL DDBC 驱动

[root@dbsvr postgres]# yum install -y unixODBC unixODBC-devel

[postgres@dbsvr ~]$ tar xfz psqlodbc-16.00.0000.tar.gz

[postgres@dbsvr ~]$ cd psqlodbc-16.00.0000/

[postgres@dbsvr psqlodbc-16.00.0000]$

[postgres@dbsvr psqlodbc-16.00.0000]$ ./configure --prefix=/usr/local

[postgres@dbsvr psqlodbc-16.00.0000]$ make

[postgres@dbsvr psqlodbc-16.00.0000]$ sudo make install

2.2.3 配置 ODBC 的安装信息

ODBC 驱动的安装信息

vi /etc/odbcinst.ini

[PostgreSQL]

Description=ODBC for PostgreSQL

Driver=/usr/local/lib/psqlodbcw.so

Setup=/usr/local/lib/psqlodbcw.so

2.2.4 配置 ODBC 数据源

vi /etc/odbc.ini

[UniversityDB]

Description=PostgreSQL Data Source

Driver=PostgreSQL

Database=universitydb

Servername=localhost

UserName=postgres

Password=dba123

Port=5432

在应用程序中，使用 DSN 名称为 UniversityDB

2.2.5 ODBC 示例代码

[postgres@dbsvr ~]$ vi odbc_example odbc_example.c

添加以下内容：

```
#include <stdio.h>
#include <sql.h>
#include <sqlext.h>

int main() {
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLRETURN ret;
    char id[6];   // ID 字段大小，假设最大长度为 5 加一个终止符
    char deptName[21];   // 部门名称字段大小，最大 20 个字符加一个终止符
    char name[21];   // 姓名字段大小，最大 20 个字符加一个终止符
    SQL_NUMERIC_STRUCT salary;   // 存储数值类型的变量，用于薪水
    SQLLEN salaryInd = 0; // 更改 salary 的指示器变量类型为 SQLLEN

    // 初始化环境
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

    // 分配连接句柄并连接到数据库
    SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    ret = SQLConnect(dbc, (SQLCHAR*)"UniversityDB", SQL_NTS, NULL, 0, NULL, 0);

    if (SQL_SUCCEEDED(ret)) {
        printf("Connected to the database.\n");

        // 分配语句句柄
        SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);

        // 执行 SQL 查询
        SQLExecDirect(stmt, (SQLCHAR*)"SELECT id, name, dept_name, salary FROM instructor",
SQL_NTS);

        // 绑定列
        SQLBindCol(stmt, 1, SQL_C_CHAR, id, sizeof(id), NULL);
```

```
        SQLBindCol(stmt, 2, SQL_C_CHAR, name, sizeof(name), NULL);
        SQLBindCol(stmt, 3, SQL_C_CHAR, deptName, sizeof(deptName), NULL);
        SQLBindCol(stmt, 4, SQL_C_NUMERIC, &salary, 0, &salaryInd);

        // 循环获取结果
        while (SQLFetch(stmt) == SQL_SUCCESS) {
            printf("ID: %s, Name: %s, Department: %s, Salary: %ld\n", id, name, deptName,
salary.val[0]); // 假设薪水可以存放在长整型中
        }

        // 清理
        SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    } else {
        // 获取并打印错误信息
        SQLCHAR message[1024];
        SQLCHAR sqlstate[SQL_SQLSTATE_SIZE+1];
        SQLINTEGER sqlcode;
        SQLSMALLINT length;

        SQLGetDiagRec(SQL_HANDLE_DBC, dbc, 1, sqlstate, &sqlcode, message, sizeof(message),
&length);
        fprintf(stderr, "连接失败, SQLSTATE = %s, SQLCODE = %d, MESSAGE = %s\n", sqlstate,
sqlcode, message);
    }

    SQLDisconnect(dbc);
    SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    SQLFreeHandle(SQL_HANDLE_ENV, env);

    return 0;
}
```

2.2.6 编译 ODBC 示例代码

[postgres@dbsvr ~]$ gcc -o odbc_example odbc_example.c -I/usr/local/include -L/usr/local/lib -lodbc

2.2.7 运行 ODBC 示例代码

```
[postgres@dbsvr ~]$ ./odbc_example
Connected to the database.
ID: 10101, Name: Srinivasan, Department: Comp. Sci., Salary: 160
ID: 12121, Name: Wu, Department: Finance, Salary: 64
ID: 15151, Name: Mozart, Department: Music, Salary: 0
ID: 22222, Name: Einstein, Department: Physics, Salary: 96
ID: 32343, Name: El Said, Department: History, Salary: 128
ID: 33456, Name: Gold, Department: Physics, Salary: 96
ID: 45565, Name: Katz, Department: Comp. Sci., Salary: 224
ID: 58583, Name: Califieri, Department: History, Salary: 192
ID: 76543, Name: Singh, Department: Finance, Salary: 0
ID: 76766, Name: Crick, Department: Biology, Salary: 0
ID: 83821, Name: Brandt, Department: Comp. Sci., Salary: 128
ID: 98345, Name: Kim, Department: Elec. Eng., Salary: 0
[postgres@dbsvr ~]$
```