第二章 MySQL 服务器骨架分析

为了看到 MySQL 的整体程序架构，我们先分析其 main 函数，从这个函数，可以看到 MySQL 的启动流程，以及启动后的状态。

MySQL 的源码目录中，核心为 sql 目录下的源代码，该目录为 MySQL 的核心部分。main 函数也在该层实现，位于文件 main.cc 中。函数非常简单，只是简单的调用 mysqld_main 函数。因此分析 mysqld_main 函数即可。

首先通过分析 mysqld_main 函数来了解 MySQL 的程序骨架。该函数位于 mysqld.cc 文件中。

int mysqld_main（int argc, char **argv）

{

//一系列的初始化，最为重要的是各种基础资源的初始化，如线程资源的相关初始化，配置文件的装载，sql 语句名字的初始化，一些全局变量的初始化，插件的初始化，日志组件的初始化，线程栈保护区大小的设置，ddl 日志的恢复(因为 MySQL 启动的时候从 main 函数开始执行，因此可能是崩溃后的重启，因此会进行相关的恢复)，网络的初始化（主要是对套接字进行监听，简单来说，就是 linux 网络编程服务器端的那几步，只不过加了一些错误处理而已)。做好这些初始化工作之后，就可以处理来自客户端的网络连接了。这时进入作为一个服务器最为核心的骨架代码了：handle_connections_sockets（）； 该函数就是 MySQL 主程序的核心。后边就是 MySQL 退出时做的一些事情了。下边详细分析函数 handle_connections_sockets（)。

}

handle_connections_sockets 函数也位于 mysqld.cc 中

handle_connections_sockets()的程序骨架就是一个大的 while 循环，在这里你能够看到典型的服务端程序骨架：一个主线程负责处理监听套接字上的新连接，然后接受链接，并创建链接相关的数据结构 THD，然后创建一个新的线程来处理客户端的链接，主线程继续监听套接字上的新连接。这就是 MySQL 的核心主程序。在这里我们只是列出了其最为核心的部分，略去了一些错误处理。

```
void handle_connections_sockets ()
{
  ........
  while  (!abort_loop)
  {
    retval= select ( (int)  max_used_connection,&readFDs,0,0,0) ;
    sock = ip_sock;  //sock为监听套接字
     flags= ip_flags;
    //循环尝试接受新的连接，只要连接成功返回，则结束循环，MAX_ACCEPT_RETRY被定义为10.最多尝试10次。
    for  (uint retry=0; retry < MAX_ACCEPT_RETRY; retry++)
    {
      size_socket length= sizeof (struct sockaddr_storage) ;
      new_sock= mysql_socket_accept (key_socket_client_connection, sock,
                                     (struct sockaddr *)  (&cAddr) , &length) ;
      if  (mysql_socket_getfd (new_sock)  != INVALID_SOCKET ||
           (socket_errno != SOCKET_EINTR && socket_errno != SOCKET_EAGAIN) )
        break;
    }
    //创建一个针对该链接socket的THD，同时使用各种结构初始化THD的网络部分，
    if  (! (thd= new THD) )
    {//如果创建THD失败，则做些错误处理（关闭socket等）继续进行循环处理，等待下一个连接到来 }
```

```
    bool is_unix_sock=  (mysql_socket_getfd (sock)  == mysql_socket_getfd (unix_sock) ) ;
    enum_vio_type vio_type=  (is_unix_sock ? VIO_TYPE_SOCKET : VIO_TYPE_TCPIP) ;
     uint vio_flags=  (is_unix_sock ? VIO_LOCALHOST : 0) ;
    vio_tmp= mysql_socket_vio_new (new_sock, vio_type, vio_flags) ;
    if  (!vio_tmp || my_net_init (&thd->net, vio_tmp) )
    {//也是一样，如果初始化网络部分失败了，则做些资源清理，然后继续下一次循环，等待链接到来。
      continue;
    }
    init_net_server_extension (thd) ;
    create_new_thread (thd) ;
  }
}
```

接着就是创建一个新的线程来处理这个链接了，create_new_thread（thd）;函数位于文件 mysqld.cc 中。该函数的主要逻辑为：检查当前连接数是否已经达到最大，如果是，则关闭链接，否则，允许链接，并进行下一步的处理。最核心的代码为：

```
static void create_new_thread (THD *thd)
{
  mysql_mutex_lock (&LOCK_connection_count) ;
  if  (connection_count >= max_connections + 1 || abort_loop)
  {
    //关闭链接，做些统计。
  }
  ++connection_count;
  if  (connection_count > max_used_connections)
    max_used_connections= connection_count;
  mysql_mutex_unlock (&LOCK_connection_count) ;
  /* Start a new thread to handle connection. */
  mysql_mutex_lock (&LOCK_thread_count) ;
  thd->thread_id= thd->variables.pseudo_thread_id= thread_id++;
  MYSQL_CALLBACK (thread_scheduler, add_connection,  (thd) ) ;
}
```

可见，最核心的是；MYSQL_CALLBACK（thread_scheduler, add_connection, （thd））。这一句是什么意思呢？MYSQL_CALLBACK 是个宏定义，该宏有三个参数，分别是：obj，func，params，即对象实例，对象实例中的函数指针成员，函数参数。因此，该语句就是调用 thread_scheduler 的 add_connection 函数，参数为 thd 即：thread_scheduler->add_connection（thd）。接着要分析函数 add_connection。

thread_scheduler 的定义如下:（位于 sql/scheduler.cc）

```
scheduler_functions *thread_scheduler= NULL;
```

scheduler_functions 的定义如下：（位于 sql/scheduler.h）

```
/* Functions used when manipulating threads */
struct scheduler_functions
{
  uint max_threads;
  bool  (*init)  (void) ;
  bool  (*init_new_connection_thread)  (void) ;
  void  (*add_connection)  (THD *thd) ;
  void  (*thd_wait_begin)  (THD *thd, int wait_type) ;
  void  (*thd_wait_end)  (THD *thd) ;
  void  (*post_kill_notification)  (THD *thd) ;
  bool  (*end_thread)  (THD *thd, bool cache_thread) ;
  void  (*end)  (void) ;
};
```

该结构体的成员主要是各种为了支持多线程的函数指针。该结构体的初始化在 sql/scheduler.cc 中，如下：

```
static scheduler_functions one_thread_per_connection_scheduler_functions=
{
  0,                                    // max_threads
  NULL,                                 // init
```

```
  init_new_connection_handler_thread,     // init_new_connection_thread
  create_thread_to_handle_connection,     // add_connection
  NULL,                                   // thd_wait_begin
  NULL,                                   // thd_wait_end
  NULL,                                   // post_kill_notification
  one_thread_per_connection_end,          // end_thread
  NULL,                                   // end
};
```

因此对 add_connection 的调用将调用到函数：create_thread_to_handle_connection，该函数位于 myqld.cc 中。该函数如下：

```
void create_thread_to_handle_connection (THD *thd)
{
  mysql_mutex_assert_owner (&LOCK_thread_count) ;
  if  (blocked_pthread_count >  wake_pthread)
  {
    /* 唤醒空闲的缓存的线程 */
    waiting_thd_list->push_back (thd) ;
    wake_pthread++;
    mysql_cond_signal (&COND_thread_cache) ;
  }
  else
  {
    char error_message_buff[MYSQL_ERRMSG_SIZE];
    /* Create new thread to handle connection */
    int error;
    inc_thread_created () ;
    thd->prior_thr_create_utime= thd->start_utime= my_micro_time () ;
    if  ( (error= mysql_thread_create (key_thread_one_connection,
                                       &thd->real_id, &connection_attrib,
                                       handle_one_connection,
                                        (void*)  thd) ) )
    {
      /* 创建线程失败了，则进行一系列的错误处理，然后返回*/
      return;
    }
    add_global_thread (thd) ;
  }
}
```

该函数逻辑也很清晰，如果存在已经空闲的线程，则唤醒空闲的线程来处理这个链接，否则创建一个新的线程来处理这个链接，至此，主线程逻辑已经结束，它又回去监听新的网络链接去了。

新线程将执行函数 handle_one_connection。该函数位于文件 sql/sql_connect.cc 中，什么也不做，只是简单的调用函数 do_handle_one_connection。do_handle_one_connection 位于文件 sql/sql_connect.cc 中。该函数做如下几件事情：

初始化线程；

初始化 THD 结构（这是一个非常重要的结构，一个链接、对应一个线程，对应一个 THD）；

用户授权；

处理所有的在这个链接上的查询；

结束线程（会将线程缓存下来用于下一个链接的处理）。

接下来结合代码具体分析，该函数代码如下：为了体现核心的逻辑，我们略去了一些代码

```
void do_handle_one_connection (THD *thd_arg)
{
  THD *thd= thd_arg;
  //初始化线程，主要是将线程pthread_detach，同时进行一些其他的初始化。
  MYSQL_CALLBACK_ELSE (thread_scheduler, init_new_connection_thread,  () , 0) ;
```

```
  //外层循环为线程的主体逻辑，以后可用于放在线程缓存中。
  for  (;;)
  {
  bool rc;
    NET *net= &thd->net;
     mysql_socket_set_thread_owner (net->vio->mysql_socket) ;
     //这个函数会进行用户授权，通过这个函数的授权，用户才能接着继续执行。他会调用以下函数;
     thd_prepare_connection ()
         |-login_connection ()
           |-check_connection ()
             |-acl_authenticate ()
    rc= thd_prepare_connection (thd) ;
    if  (rc)
       goto end_thread;
    //这个循环是该线程处理该客户端连接的核心。就是循环调用函数do_command
    while  (thd_is_connection_alive (thd) )
    {
      mysql_audit_release (thd) ;
      if  (do_command (thd) )
         break;
    }
    end_connection (thd) ;
end_thread:
    close_connection (thd) ;
    if  (MYSQL_CALLBACK_ELSE (thread_scheduler, end_thread,  (thd, 1) , 0) )
      return;                                   // Probably no-threads
    /*
      If end_thread ()  returns, we are either running with
      thread-handler=no-threads or this thread has been schedule to
      handle the next connection.
    */
    thd= current_thd;
    thd->thread_stack=  (char*)  &thd;
  }
}
```

右上边的分析，可以看到，链接建立之后，核心的就是循环调用函数 do_command 了。该函数位于文件 sql/sql_parse.cc 中，该函数读取 TCP 连接上的数据包，根据 MySQL 客户端和服务器之间的通信协议解析出数据包的内容，执行里边的命令。它的核心代码如下：

```
bool do_command (THD *thd)
{
  thd->lex->current_select= 0;
  my_net_set_read_timeout (net, thd->variables.net_wait_timeout) ;
  thd->clear_error () ;       // Clear error message
  thd->get_stmt_da () ->reset_diagnostics_area () ;
  net_new_transaction (net) ;
  thd->m_server_idle= true;
  packet_length= my_net_read (net) ;
  thd->m_server_idle= false;
  if  (packet_length == packet_error)
  {
    //一系列的错误处理，然后返回。篇幅原因，这里省略了。
    return_value= FALSE;
    goto out;
  }
  packet=  (char*)  net->read_pos;
  packet[packet_length]= '\0';                  /* safety */
  command=  (enum enum_server_command)  (uchar)  packet[0];
  if  (command >= COM_END)
    command= COM_END;       // Wrong command
  /* Restore read timeout value */
  my_net_set_read_timeout (net, thd->variables.net_read_timeout) ;
```

```
  return_value= dispatch_command (command, thd, packet+1,  (uint)   (packet_length-1) ) ;
out:
  /* The statement instrumentation must be closed in all cases. */
  DBUG_ASSERT (thd->m_digest == NULL) ;
  DBUG_ASSERT (thd->m_statement_psi == NULL) ;
  DBUG_RETURN (return_value) ;
}
```

该函数最核心的就是两步：my_net_read 和 dsipatch_command 函数。

前一个函数从客户端获取完整的数据包（如果包太大，客户端分开传送的，那么它会把包读取完整的），解压缩它，并去掉一些头部字段。一旦完成，我们得到字节数组，其中包含客户端发送的内容。第一个字节很重要，因为它标识了消息的类型。我们将把它和其余的数据包传递给 dispatch_command 函数。例如我们的一个查询:

select * from test1；

那么将会收到如下参数，当然了，不同的体系结构 thd，packet+1 会有所不同。如下：

dispatch_command （COM_QUERY, 0x33b5940, packet=0x338e9e1 "select * from test1", packet_length=19）

该函数位于 sql/sql_parse.cc 中，该函数有 700 多行，其主要功能是：对每个客户端传过来的命令进行处理。虽然很长，但是代码结构非常清晰，具体如下：

```
bool dispatch_command (enum enum_server_command command, THD *thd,
          char* packet, uint packet_length)
{
  NET *net= &thd->net;
  ..............
  thd->profiling.start_new_query () ;
  thd->set_command (command) ;
  ............
  thd->set_query_id (next_query_id () ) ;
  inc_thread_running () ;
  ..............
  switch  (command)  {
  case COM_INIT_DB:
  case COM_REGISTER_SLAVE:
  case COM_CHANGE_USER:
  case COM_STMT_EXECUTE:
  ..........
  case COM_QUERY:
  case COM_FIELD_LIST:        // This isn't actually needed
  case COM_QUIT:
  case COM_BINLOG_DUMP_GTID:
  case COM_BINLOG_DUMP:
  case COM_REFRESH:
  case COM_SHUTDOWN:
  case COM_STATISTICS:
  case COM_PING:
  case COM_PROCESS_INFO:
  case COM_PROCESS_KILL:
  case COM_SET_OPTION:
  {
    status_var_increment (thd->status_var.com_stat[SQLCOM_SET_OPTION]) ;
    uint opt_command= uint2korr (packet) ;
    switch  (opt_command)  {
    case  (int)  MYSQL_OPTION_MULTI_STATEMENTS_ON:
    case  (int)  MYSQL_OPTION_MULTI_STATEMENTS_OFF:
    default:
    }
    break;
  }
  case COM_DEBUG:
  case COM_SLEEP:
  case COM_CONNECT:       // Impossible here
  case COM_TIME:        // Impossible from client
```

```
  case COM_DELAYED_INSERT:
  case COM_END:
  default:
    break;
  }

done:
  thd->update_server_status () ;
  thd->protocol->end_statement () ;
  query_cache_end_of_result (thd) ;
  thd->set_command (COM_SLEEP) ;
  MYSQL_END_STATEMENT (thd->m_statement_psi, thd->get_stmt_da () ) ;
  ......................
  dec_thread_running () ;
  thd->packet.shrink (thd->variables.net_buffer_length) ; // Reclaim some memory
  free_root (thd->mem_root,MYF (MY_KEEP_PREALLOC) ) ;
  ......................
  thd->profiling.finish_current_query () ;
}
```

就是一系列的 case，一共 27 个，针对每一种情况执行对应的 case。例如：客户端执行 use test，则表示要改变当前的数据库，然后就命中 COM_INIT_DB 这个 case 了。如果客户端要执行预编译好的 SQL 语句，则命中 COM_STMT_EXECUTE 这个 case 了，再比如，客户端执行 select * from test1，则命中 COM_QUERY（通用查询，MySQL 中不是 select 才叫查询。MySQL 中很多操作，比如 update，insert 等都叫一次查询。）这个 case 了。我们知道，MySQL 是插件式存储引擎，那么接下来我们以一种场景为例来说明 MySQL Server 层如何与存储引擎层进行交互。显然，必须 MySQL Server 层定义好 API，然后存储引擎层去实现这些 API，那么 MySQL Server 层肯定需要将语句的执行计划对应到对 API 的调用上。这只是我们的分析，具体看看是不是这样呢？现在以插入一条数据为例进行分析：

如上分析，也将命中 COM_QUERY 这个 case，这个 case 下，主要是做一些 sql 语句处理前的准备，然后调用函数 mysql_parse 来处理 sql 语句。其核心就是 mysql_parse。该函数位于 sql/sql_parse.cc 中，负责具体的 sql 处理，其核心逻辑有两个：因为 sql 的词法分析、语法分析、查询重写、查询优化、物理执行计划生成、执行物理计划需要很长时间，因此，数据库系统通常会对查询进行缓存，当一个查询（广义的查询）来了之后首先会查看缓存，如果命中，则直接返回，如果没有命中查询缓存，则进行下一步的处理即：sql 的分析、查询计划的生成、执行等。

下边针对没有命中查询缓存的情况进行分析：

首先调用 parse_sql，该函数只是简单的调用函数 MYSQLparse，MYSQLparse 其实会在预处理阶段被替换为 yyparse，yyparse 就是 gnu 的 Bison 生成的语法解析器的入口，该函数会生成语法树，MySQL 的词法分析是自己做的，该函数执行之后，SQL 语句的语法分析结果会被放在 TDH 的 LEX 这个对象中，该对象包含了 sql 语句的信息，例如：什么样的 sql 语句，查询的那些表，where 条件信息，选择那些列等等。

对 sql 分析完成之后，如果分析成功，则会进行查询重写，注意：查询缓存只处理 SELECT，我们也不会对 SELECT 进行查询重写。**而且一定要区分开查询重写与查询优化完全是两码事哈，查询优化还早着呢。**

经过 sql 分析和查询重写，然后就是对其进行执行了。这一步由函数 mysql_execute_command 来完成，位于 sql/sql_parse.cc 中，需要详细分析，该函数非常长，大约 2800 多行，虽然很长，但是逻辑也非常清晰，就是一堆 case，这堆 case 就是对应 sql 语法分析器分析出来的要执行的 sql 命令。由此可见，代码的处理结构还是非常的清晰的。例如：在语法分析中的一个代码片段：（词法分析、语法分析的基础知识参见附录 1）

```
    break;
  case 1748:
    {
              LEX *lex= Lex;
              lex->sql_command= SQLCOM_SHOW_TABLE_STATUS;
              lex->select_lex.db=  (yyvsp[ (3)  -  (4) ].simple_string) ;
              if  (prepare_schema_table (YYTHD, lex, 0, SCH_TABLES) )
                MYSQL_YYABORT;
     }
    break;
  case 1749:
    {
             LEX *lex= Lex;
             lex->sql_command= SQLCOM_SHOW_OPEN_TABLES;
             lex->select_lex.db=  (yyvsp[ (3)  -  (4) ].simple_string) ;
             if  (prepare_schema_table (YYTHD, lex, 0, SCH_OPEN_TABLES) )
               MYSQL_YYABORT;
     }
    break;
  case 1750:
```

如上：例如在语法分析中，语法分析器发现要执行的 SQL 语句是 SQLCOM_SHOW_OPEN_TABLES，那么在函数 mysql_execute_command 中也会有对应的 case 来处理这个命令的。例如在函数 mysql_execute_command 中就有对应的如下代码段：

```
  ..................
  case SQLCOM_SHOW_TABLE_STATUS:
  case SQLCOM_SHOW_OPEN_TABLES:
  .........
  case SQLCOM_SELECT:
  {
    thd->status_var.last_query_cost= 0.0;
    thd->status_var.last_query_partial_plans= 0;
    if  ( (res= select_precheck (thd, lex, all_tables, first_table) ) )
      break;
    res= execute_sqlcom_select (thd, all_tables) ;
    break;
  }
  。。。。。。。。。
```

为了逻辑清晰，在此总结一下 MySQL 服务器的骨架代码（后续还会分析如何与存储引擎的 API 进行交互）。

```
//MySQL服务器主线程
int main ()
{
  mysqld_main ()
  {
    my_init () ;
    udf_init () ;
    .....
    init_slave () ;
    network_init () ;
    .....
    handle_connections_sockets ()
    {
      while ()
      {//循环监听socket上的连接，对于每个连接，创建一个线程去处理该链接上的查询。
        select () ;
        accept () ;
        create_new_thread ()
        {
          create_thread_to_handle_connection ()
          {
            mysql_thread_create (,, ,handle_one_connection, (void*)  thd) ;
            //在此，主线程又进入对socket的监听，等待新连接的到来，而新线程处理该链接。
          }
```

```
        } //end of create_new_thread
      } //end of while
    } // end of handle_connections_sockets
  } //end of mysqld_main
}// end of main

//MySQL服务器处理链接的线程开始执行
//如上，MySQL开始处理新的链接，执行函数handle_one_connection
handle_one_connection ()
{
  do_handle_one_connection ()
  {
    ......
    thd_prepare_connection () ;  //检查用户权限，看看是否允许链接
    while  ()   //循环处理该连接上的各种命令（各种服务器管理命令、SQL命令等）
    {
      do_command (thd)
      {
        my_net_read () ;  //读取完整的客户端命令
        ...... //解析客户端的命令
        dispatch_command (enum enum_server_command) //命令分发
        {//注意：客户端和服务端之间的命令分为两层，该函数只识别第一层命令。
          switch  (command)  {
            ........
            case COM_STMT_EXECUTE:
            ..........
            case COM_QUERY:
            ..........
          }
        }
      }
    }
  }
}

/*
 * dispatch_command中处理第一层命令，即：mysql客户端和服务端通信协议中粒度比较粗的命令，
 * 即：enum_server_command中定义的命令，这些命令是针对整个MySQL服务器而言的，例如：
 * COM_REGISTER_SLAVE：注册从属服务器
 * COM_CHANGE_USER：改变当前的链接用户
 * COM_QUERY：让服务器处理一个查询（注意：MySQL中的查询是广义的，包含：DDL和DML等语句）
 * COM_SHUTDOWN：服务器的关闭
 * COM_PROCESS_KILL：kill的处理
 * 以上只是举了几个简单的例子，enum_server_command的完整定义在mysql_comm.h中。
 *
 * 对于COM_QUERY，MySQL会对各种SQL语句进行解析。解析出SQL相关的命令（enum_sql_command）
 * 来进行执行，这个是由函数mysql_execute_command来完成的。enum_sql_command枚举的定义
 * 在sql_cmd.h中。可见MySQL从代码上就是逐渐分层，sql的处理虽然重要，但只是一个MySQL服务
 * 器的一个模块。具体enum_server_command和enum_sql_command的定义可以查看对应的.h文件。
 */
 //接着分析链接线程中dispatch_command函数最为重要的一个case分支,case COM_QUERY:

case COM_QUERY:
{
  MYSQL_QUERY_START () ;
  mysql_parse ()
  {
    if  (query_cache_send_result_to_client (thd, rawbuf, length)  <= 0)
    {//没有命中查询缓存
      //对sql语句进行词法分析,语法分析,上边说的enum_sql_command就在这里解析出来的。
      parse_sql () ;
      mysql_rewrite_query () ;  //必要时进行查询重写,注意：select不会重写。
      .......
      mysql_execute_command (THD *thd)
      {
          switch  (lex->sql_command)
```

```
          {
            case SQLCOM_SHOW_STATUS_PROC:
            case SQLCOM_SHOW_PROFILE:
            ........
            case SQLCOM_SELECT:
            .........
          }
      }
    }
    else
    {
      //查询命中,做点处理,写日志
    }
  }
}
```

至此，服务器的大体骨架就清楚了，接下来研究 MySQL 服务器如何同存储引擎曾进行交互。

作者：许富博

限于本人水平有限，欢迎大家随时指正，联系方式：
xufubobo@gmail.com
xufubobo@163.com
1332841493@qq.com