# 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](#_附录一)）

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

版权所有，文章以学习和交流为主，切勿用于商业用途。

限于本人水平有限，欢迎大家随时指正，联系方式：

[xufubobo@gmail.com](mailto:xufubobo@gmail.com)

[xufubobo@163.com](mailto:xufubobo@163.com)

[1332841493@qq.com](mailto:1332841493@qq.com)