# 概述

参考：

SHUTDOWN过程：<http://mysql.taobao.org/monthly/2017/08/09/>

InnoDB 异步IO工作流程：<http://mysql.taobao.org/monthly/2017/07/10/>

代码路径：https://github.com/mysql

入口函数：mysql-server-8.0/sql/mysqld.cc::mysqld\_main

## 初始化内容

**参数配置解析：**

分别从/etc/my.conf, /etc/mysql/my.conf, SYSCONFDIR/my.cnf, "$MYSQL\_HOME"/my.conf, defaults-extra-file, ~/my.conf, DATADIR/mysqld-auto.cnf，命令行加载配置文件中的配置项。

SYSCONFDIR/my.cnf是在编译时指定的路径，DATADIR是指定的数据目录，如果不指定特定的配置的，将按照上面逐步查找配置。

**performance schema interface（PSI）初始化**：

PSI主要用于监控/诊断MySQL服务运行状态，可以阅读官方文档：https://dev.mysql.com/doc/refman/8.0/en/performance-schema.html 和（PFS）

**初始化初始化定时器timer；**

**时间相关的基础的锁结构；**

**资源池，通知消息等初始化；**

**字符集/collation**

**元数据**

**系统库表，information\_schema**

**DDL/DQL/DML/ACL支持的命令**

**plugin，存储引擎**

**binlog**

**优化器optimizer代价**

**恢复持久化中的事务数据，**

**ACL相关的初始化，授权相关的初始化**

**SSL**

**信号处理**

**审计**

**相关初始化**

**各类管理器**

**MySQL中的线程**

## 线程

MySQL中主要会启动以下线程处理各种任务：

连接监听主线程，用于监听处理客户端的连接请求，然后将连接请求分发到单独的线程，并处理该连接上的所有请求；

处理线程，是一个线程池；

slave/relay线程，连接到主服务的线程，并且从主服务器中读取binlog更新，在本机上应用更新；

信号处理线程，用来处理信号，具体的信号包括了：SIGTERM, SIGQUIT, SIGHUP, SIGUSR1 and SIGUSR2

event\_scheduler\_thread，事件调度线程，用来调度执行每个表上定义事件，即Event。

compress\_gtid\_table\_thread，开启一个线程用来压缩GTID\_Table，具体的压缩逻辑在mysql-server-8.0/sql/http://rpl\_gtid\_persist.cc中，估计后续会进一步阅读。

helper\_thread，辅助线程来分发计时器timer到期通知消息，具体逻辑在timer\_notify\_thread\_func。

admin\_socket\_thread，如果开启了该线程，并且以端口监听，则可以用handle\_admin\_socket处理请求的逻辑，具体的连接也会用一个新的线程处理。

## mysqld\_safe

参考：

mysqld\_safe代码考古：<http://mysql.taobao.org/monthly/2022/04/03/>

# 函数调用栈

substitute\_progpath：将命令行的程序名替换成全路径的程序名，可能是“全路径”，home路径解释，环境变量PATH解释。

sysd::notify\_connect：在环境变量（“NOTIFY\_SOCKET“）中查找socket的文件名，如果存在的话则连接该Socket

sysd::notify:向Notify Socket发送正在启动的消息。

my\_init：初始化my\_sys函数，资源和变量

my\_thread\_global\_init：初始化线程的环境（初始化一堆资源锁）

PSI\_mutex\_key 结构

my\_thread\_init：申请mysys线程的内存，主要用于debug

load\_defaults -> my\_load\_defaults：从配置文件中读取配置项

init\_default\_directories：获取配置的目录

my\_search\_option\_files：处理在缺省目录下的配置文件，即上一步结果

get\_defaults\_options：从命令行中获取配置选项；

init\_variable\_default\_paths：初始化配置文件的作用域（enum\_variable\_source）

search\_default\_file：如果是绝对路径，直接读取；

search\_default\_file\_with\_ext：从输入配置的文件中读取；

my\_default\_get\_login\_file:从login集群中读取配置。

my\_search\_option\_files

set\_args\_separator：分隔符

persisted\_variables\_cache：持久化的参数配置缓存，采用JSON进行存储，

Init：初始化，从命令行或者是环境变量“MYSQL\_DATADIR“中获取文件

my\_handle\_options：主要处理参数的逻辑。

load\_persist\_file：加载持久化文件的配置，并且转成JSON格式，分析出K/V结构；

append\_read\_only\_variables：将上一步分解出的只读配置参数追加到命令行参数中。

init\_variable\_default\_paths：初始化配置文件的作用域（enum\_variable\_source）

init\_pfs\_instrument\_array: PSI: performance schema interface; PFS:performance storage

handle\_early\_options:处理早期的选项，包括：performance schema；与help或者启动相关的参数

sys\_var\_add\_options：增加系统参数；

增加命令行中的早期选项；

add\_terminator：增加终结。

handle\_options->my\_handle\_options：处理参数

init\_sql\_statement\_names: 初始化与sql语句相关的名字（com\_status\_vars），包括各种dml/ddl/acl/dql等

sys\_var\_init：初始化系统变量(mysql-server-8.0/sql/sys\_vars.cc中定义静态变量)

mysql\_add\_sys\_var\_chain

init\_error\_log:初始化错误日志

adjust\_related\_options：调整相关的参数，

adjust\_open\_files\_limit

adjust\_max\_connections

adjust\_table\_cache\_size

adjust\_table\_def\_size

initialize\_performance\_schema：初始化performance\_schema

pfs\_automated\_sizing：将ps相关的参数都置为0；

init\_timers：根据系统进行初始化performance\_schema的计数器，并且获得当前值；可以从performance\_timers表获取

my\_timer\_init

my\_timer\_cycles

my\_timer\_nanoseconds

my\_timer\_milliseconds

初始化histograms（表的统计量）的定时器，

init\_event\_name\_sizing：初始化全局表I/O，表锁/元数据等的事件

register\_global\_classes：初始化全局的仪表，包括类型有：表I/O，表锁，Idle类，元数据类，错误类，事物类。

接下来是申请一堆仪表的内存

并且从参数里头获取配置

init\_pfs\_plugin\_table：初始化pfs内置的table

PFS\_dynamic\_table\_shares::init\_mutex

init\_pfs\_plugin\_table

LO\_init：初始化Lock Order，主要用Lock Order Graph描述锁依赖关系：https://dev.mysql.com/doc/refman/8.0/en/lock-order-tool.html

//START与psi相关的服务

设置psi相关的服务：thread\_service，mutex\_service，rwlock\_service，cond\_service，file\_service，socket\_service，table\_service，mdl\_service，idle\_service，stage\_service，statement\_service，transaction\_service，memory\_service，error\_service，data\_lock\_service，system\_service

init\_server\_psi\_keys:注册psi key的函数，包括psi服务的mutex，rwlock，conds，threads，files，stage，socket；内存的key，sql语句的信息（init\_sql\_statement\_names中提到的），psi\_api，与store procedures相关的key，与调度（scheduler）相关的（主要是evengt），与client公用的key，vio的key（与通信相关的）

my\_thread\_global\_reinit：重新初始化一些比较开始与锁相关的，因为需要考虑一些仪表

// 与psi相关的服务 END

component\_infrastructure\_init

mysql\_services\_bootstrap：启动服务注册，动态加载器，和基础的服务

registry\_init：创建注册服务套件，内部的结构mysql\_registry

把加载器的实现写入到服务注册表

dynamic\_loader\_init：初始化动态加载器，其实就是初始化锁

dynamic\_loader\_scheme\_file\_init：初始化scheme文件

pfs\_init\_services: ps是performance storage

register\_pfs\_notification\_service

register\_pfs\_resource\_group\_service

Resource\_group\_mgr::init：初始化资源组管理器，

与资源相关的服务，包括： registry service，mysql\_service\_pfs\_resource\_group\_v3\_t,mysql\_service\_pfs\_notification\_v3\_t

注册线程创建/断开连接的回调

创建用户级/系统级的资源组对象

调用psi\_thread\_service相关函数：new\_thread，set\_thread\_os\_id，set\_thread

mysql\_audit\_initialize:初始化与audit相关的变量；

Srv\_session::module\_init：初始化srv\_session模块

query\_logger.init：查询日志初始化

init\_common\_variables：

init\_thread\_environment：初始化与线程相关的锁；

mysql\_init\_variables：初始化全局变量为缺省值

mysql\_bin\_log.set\_psi\_keys完成bin log的初始化，让仪表相关的对bin log可见

mysql\_bin\_log.init\_pthread\_objects：初始化bin log与线程相关的锁

get\_options，剩余的可选项，binlog，压缩算法，页的大小，线程缓存大小，back\_log

Connection\_handler\_manager::init,初始化连接池

Per\_thread\_connection\_handler::init，初始化Per\_thread\_connection\_handler，初始化锁

初始化mysql client的plugin

选字符集/collation

日志配置

创建数据目录

表的大小写等

my\_init\_signals：初始化信号处理

线程栈大小检查

Migrate\_keyring::init:初始化Migrate\_keyring（一种mysql数据搬运模式）相关的，如压缩方式，并且连接到数据源主机

Migrate\_keyring::execute

fetch\_and\_store\_keys

// 如果启动方式为Migrate\_keyring，则数据搬完后退出

set\_ports:确定tcp端口

init\_server\_components

mdl\_init：metadata locking subsystem元数据锁子系统，

MDL\_map::init， MDL\_key，一个重要的数据结构：元数据锁

partitioning\_init

table\_def\_init：表定义缓存（Table\_definition\_cache）

Table\_cache\_manager::init:表缓存初始化

Table\_cache::init

hostname\_cache\_init：初始化（client）主机名缓存，Host\_entry

my\_timer\_initialize：初始化内部组建，fd，

start\_helper\_thread：时间通知线程，timer\_notify\_thread\_func

init\_slave\_list：初始化从机列表

transaction\_cache\_init：初始化事务缓存的psi键

MDL\_context\_backup\_manager::init，MDL\_context\_backup\_manager

：维护XA事务中元数据锁

delegates\_init：初始化外部observer的委托，包括事务，bin log，服务状态，bin log传输，relay IO

与bin/relay log相关的初始化

process\_key\_caches：用ha\_init\_key\_cache函数初始化所有的key缓存

ha\_init\_errors：初始化错误信息

gtid\_server\_init：GTID server初始化

udf\_init\_globals：初始化udf结构体

init\_ssl：配置SSL包

ssl\_start：

SSL\_library\_init

OpenSSL\_add\_all\_algorithms

init\_ssl\_locks：初始化ssl锁，

init\_lock\_callback\_functions：锁回调函数

plugin\_register\_early\_plugins

plugin\_init\_internals：初始化内部plugin，资源/锁

plugin\_load\_list

plugin\_dl\_add

plugin\_add

plugin\_init\_initialize\_and\_reap:重试失败的plugin

plugin\_register\_builtin\_and\_init\_core\_se,初始化内置plugin，包含MyIsam，CSV，InnoDB

init\_sql\_command\_flags:初始化sql命令flag

dd::init：

cache::Shared\_dictionary\_cache::init，包括字符/collation，抽象表，事件，routine，schema，column，tablespace，资源组等字典初始化

System\_tables::instance()->add\_inert\_dd\_tables()

System\_views::instance()->init() 系统视图，主要是information\_schema下的

Dictionary\_impl::init

plugin\_register\_dynamic\_and\_init\_all，注册动态的plugin和初始化所有的plugin

确定线程池组织方式：SCHEDULER\_ONE\_THREAD\_PER\_CONNECTION/SCHEDULER\_NO\_THREADS

从5.7升级到8.0相关的处理

dd::performance\_schema::init\_pfs\_tables：处理好pfs相关的表

upgrade\_system\_schemas：升级系统库表

Resource\_group\_mgr::post\_init从磁盘读取资源组数据

handle\_options：处理剩余的参数可选项

ha\_init：初始化

query\_logger相关的处理

initialize\_storage\_engine：初始化缺省（或临时）的存储引擎

Recovered\_xa\_transactions::init：恢复XA事务初始化

Recovered\_xa\_transactions::recover\_prepared\_xa\_transactions恢复预处理的XA事务

init\_server\_auto\_options：初始化服务自动项，

MYSQL\_BIN\_LOG::open\_binlog，打开一个bin log文件，分析主备bin log，也会purge一些日志

init\_optimizer\_cost\_module：初始化优化器（应该是物理优化器）代价的模块

init\_cache\_tmp\_engine\_properties

Cache\_temp\_engine\_properties::init，临时的引擎特性，HEAP/TempTable/INNODB

ft\_init\_stopwords：停用词

init\_max\_user\_conn：client连接

Gtid\_state::init，将server\_uuid放至sid\_map中

MYSQL\_BIN\_LOG::init\_gtid\_sets，从gtid\_executed表和binlog文件中初始化GLOBAL.GTID\_EXECUTED 和 GLOBAL.GTID\_PURGED 的值，分别应该是已经执行的GTID和回收的GTID

init\_ssl\_communication：

set\_fips\_mode：设置openssl包的fips

SslAcceptorContext::singleton\_init，初始化ssl接收器

do\_auto\_cert\_generation，生成证书

init\_rsa\_keys，从磁盘中加载RSA键值对，sha256

network\_init，初始化服务的listener，三种，Mysqld\_socket\_listener/Named\_pipe\_listener/Shared\_mem\_listener，后两者仅在windows下会启动

Connection\_acceptor::init\_connection\_acceptor

Mysqld\_socket\_listener::setup\_listener，如果是Mysqld\_socket\_listener，并且需要开通管理员socket

spawn\_admin\_thread，启动admin的listener线程。

admin\_socket\_thread

handle\_admin\_socket,开始监听管理请求

create\_pid\_file，创建pid文件

reload\_optimizer\_cost\_constants，重新加载优化器的计算成本常量

Cost\_constant\_cache::reload

mysql\_component\_infrastructure\_init，通过初始化动态加载器来初始化mysql服务组建

persistent\_dynamic\_loader\_init，恢复持久化文件中的数据

mysql\_persistent\_dynamic\_loader\_imp::init

open\_component\_table

读取组件table中的数据

mysql\_dynamic\_loader\_imp::load

dd::sdi\_file::load

trans\_rollback\_stmt：回滚单语句的事务

ha\_rollback\_trans

trans\_rollback：回滚当前的事务

server\_component\_init，除第一个函数外，其他均为dummy空函数，

mysql\_comp\_sys\_var\_services\_init

mysql\_string\_services\_init();

mysql\_comp\_status\_var\_services\_init();

mysql\_comp\_system\_variable\_source\_init();

mysql\_backup\_lock\_service\_init();

clone\_protocol\_service\_init();

page\_track\_service\_init();

mysql\_security\_context\_init();

mysql\_server\_ongoing\_transactions\_query\_init();

host\_application\_signal\_imp\_init();

mysql\_audit\_api\_service\_init();

mysql\_current\_thread\_reader\_imp\_init();

mysql\_keyring\_iterator\_service\_init();

mysql\_comp\_udf\_extension\_init();

mysql\_connection\_attributes\_iterator\_imp\_init();

trans\_commit\_stmt，提交语句事务

trans\_commit，提交当前事务

mysql\_rm\_tmp\_tables，删除之前服务遗留的临时文件

acl\_init

init\_acl\_cache

check\_engine\_type\_for\_acl\_table，

check\_acl\_tables\_intact

notify\_flush\_event

init\_acl\_memory，出事acl的内存

my\_tz\_init：初始化时区

open\_trans\_system\_tables\_for\_read

open\_tables

open\_tables\_check\_upgradable\_mdl

find\_table\_for\_mdl\_upgrade

lock\_table\_names

lock\_tables

mysql\_lock\_tables

lock\_tables\_check

get\_lock\_data：获得表的锁结构

check\_lock\_and\_start\_stmt

闰秒相关的初始化逻辑

grant\_init：

grant\_reload：加载授权

open\_and\_lock\_tables

grant\_load：从授权表中加载表/列的权限检查信息

grant\_reload\_procs\_priv：从procs\_priv表中读取权限信息

commit\_and\_close\_mysql\_tables：

dynamic\_privilege\_init：

mysql\_plugin\_registry\_acquire

初始化管理相关的权限

servers\_init：从mysql库中初始化结构数据

servers\_reload:初始化mysql.servers表的数据

udf\_read\_functions\_table：从mysql.func中加载数据

init\_status\_vars：

init\_slave：初始化slave线程，并开启

Rpl\_info\_factory::create\_slave\_info\_objects

start\_slave\_threads,对多个channel开启

start\_slave\_thread：：handle\_slave\_sql处理relay日志

initialize\_performance\_schema\_acl

ACL\_internal\_schema\_registry::register\_schema

Events::init：初始化event结构

load\_events\_from\_db

Event\_scheduler::start

pre\_init\_event\_thread

event\_scheduler\_thread

start\_signal\_handler,启动单独的线程signal\_hand来处理信号，如SIGTERM, SIGQUIT, SIGHUP, SIGUSR1 and SIGUSR2

process\_bootstrap：

server\_components\_initialized

run\_bootstrap\_thread：启动单独处理init\_file 中的sql语句

handle\_bootstrap

handle\_bootstrap\_impl

process\_iterator

dd::init

mysql\_audit\_notify：

mysql\_audit\_acquire\_plugins，获取plugins

event\_class\_dispatch

plugins\_dispatch,通过调用plugin的event\_notify函数分发时间

start\_handle\_manager，开启handler管理线程

handle\_manager，线程函数，监听COND\_manager信号等

create\_compress\_gtid\_table\_thread，开启一个线程调用以下函数

compress\_gtid\_table，根据压缩COND\_compress\_gtid\_table信号对gtid\_executed表进行压缩

Gtid\_state::compress

Gtid\_table\_persistor::compress，更多详细的逻辑在mysql-server-8.0/sql/rpl\_gtid\_persist.cc中

Connection\_acceptor::connection\_event\_loop：开启监听服务。 window会进入setup\_conn\_event\_handler\_threads函数，并启动三个单独线程处理不同的连接方式，即Mysqld\_socket\_listener/Named\_pipe\_listener/Shared\_mem\_listener

// 以下为结束阶段的代码

待收到结束信号后，

terminate\_compress\_gtid\_table\_thread，终止compress\_gtid\_table\_thread

save\_gtids\_of\_last\_binlog\_into\_table，保存gtid的最新值到表中

发送信号给监听线程

close\_connections：关闭连接，SIGQUIT信号

Per\_thread\_connection\_handler::kill\_blocked\_pthreads

Mysqld\_socket\_listener/Shared\_mem\_listener/Named\_pipe\_listener::close\_listener

mysql\_socket\_shutdown

mysql\_socket\_close

Connection\_acceptor::close\_listener

Events::stop，停止Event调度器

Set\_kill\_conn,对连接设置KILL\_CONNECTION flag

Events::deinit，清理调度器的资源，

关闭所有连接，Call\_close\_conn::close\_connection

Connection\_handler\_manager::wait\_till\_no\_connection，等待所有连接结束

delete\_slave\_info\_objects，释放slave线程占用的资源

终止仪表线程

# 总体流程

## 概述

MySQL 的启动流程主要包括以下几个步骤：

1. 读取配置文件：MySQL启动时会读取配置文件（通常是my.cnf），该文件包含了MySQL的配置参数和选项。MySQL根据配置文件中的设置进行初始化和运行时的参数配置。

2. 启动服务进程：MySQL会创建一个服务进程，通常是mysqld进程。该进程是MySQL数据库的核心进程，负责处理客户端连接、执行查询语句、管理数据等。

3. 初始化内存和数据结构：mysqld进程在启动时会进行内存和数据结构的初始化工作。它会分配内存来存储数据缓冲区、连接信息、查询缓存、索引结构等。此外，还会加载并解析数据字典、系统表等数据库元数据。

4. 启动插件和存储引擎：MySQL可以加载和启动不同的插件和存储引擎，以扩展功能和支持不同的数据存储引擎。在启动过程中，MySQL会加载和初始化配置文件中指定的插件和存储引擎，以便后续的数据操作和查询。

5. 建立网络连接：MySQL会监听指定的网络端口，以便接收客户端的连接请求。它会建立与客户端的网络连接，并等待客户端发送查询请求。

6. 处理客户端请求：当客户端发送查询请求时，MySQL会接收并处理这些请求。它会解析查询语句，执行相应的操作，如查询、更新、插入、删除等。MySQL使用查询优化器来选择最优的执行计划，并利用存储引擎来访问和操作实际的数据。

7. 提供结果和关闭连接：MySQL在执行查询后，会将结果返回给客户端，并关闭与客户端的连接。如果客户端还有其他请求，MySQL将继续处理这些请求，直到客户端断开连接或服务器关闭。

需要注意的是，以上是一个简化的描述，MySQL的启动流程涉及到更多细节和步骤。具体的启动流程可能会受到操作系统、MySQL 版本和配置参数的影响。不同的启动选项和参数可以影响 MySQL 的行为和性能，因此在启动 MySQL 时，配置文件的设置和选项选择非常重要。

主要代码在sql/mysqld.cc中，精简后的代码如下：

int main(int argc, char \*\*argv) //标准入口函数

MY\_INIT(argv[0]); //调用mysys/My\_init.c->my\_init()，初始化mysql内部的系统库

logger.init\_base(); //初始化日志功能

init\_common\_variables(MYSQL\_CONFIG\_NAME,argc, argv, load\_default\_groups)

//调用load\_defaults(conf\_file\_name, groups, &argc, &argv)，读取配置信息

user\_info= check\_user(mysqld\_user);//检测启动时的用户选项

set\_user(mysqld\_user, user\_info);//设置以该用户运行

init\_server\_components();//初始化内部的一些组件，如table\_cache, query\_cache等

network\_init();//初始化网络模块，创建socket监听

start\_signal\_handler();//创建pid文件

mysql\_rm\_tmp\_tables() || acl\_init(opt\_noacl)//删除tmp\_table并初始化数据库级别的权限。

init\_status\_vars(); // 初始化mysql中的status变量

start\_handle\_manager();//创建manager线程

handle\_connections\_sockets();//主要处理函数，处理新的连接并创建新的线程处理之

## 读取配置文件

参考：

参数解析流程：<http://mysql.taobao.org/monthly/2021/08/04/>

## 启动服务进程

## 初始化内存和数据结构

## 启动插件和存储引擎

## 建立网络连接

在mysqld.cc/main函数中调用，通过独立的线程实现。

具体调用关系如下：

mysqld\_main

-> check\_and\_spawn\_admin\_connection\_handler\_thread

-> spawn\_admin\_thread

-> admin\_socket\_thread

-> handle\_admin\_socket

-> select

-> accept\_connection

-> Connection\_handler\_manager::process\_new\_connection

-> Per\_thread\_connection\_handler::add\_connection

-> handle\_connection

-> do\_command

### handle\_admin\_socket

主要代码在sql/mysqld.cc中，精简后的代码如下：

THD \*thd;

FD\_SET(ip\_sock,&clientFDs); //客户端socket

while (!abort\_loop)

readFDs=clientFDs;

if (select((int) max\_used\_connection,&readFDs,0,0,0) error && net->vio != 0 &&!(thd->killed == THD::KILL\_CONNECTION))

{

if(do\_command(thd)) //处理客户端发出的命令

break;

}

end\_connection(thd);

}

#### Select

#### accept\_connection

#### Connection\_handler\_manager::process\_new\_connection

##### Per\_thread\_connection\_handler::add\_connection

基本逻辑如下：

thread\_count++;//增加当前连接的线程

thread\_scheduler.add\_connection(thd);

for (;;) {

lex\_start(thd);

login\_connection(thd); // 认证

prepare\_new\_connection\_state(thd); //初始化thd描述符

while(!net->error && net->vio != 0 &&!(thd->killed == THD::KILL\_CONNECTION))

{

if(do\_command(thd)) //处理客户端发出的命令

break;

}

end\_connection(thd);

}

###### handle\_connection

###### do\_command

## 处理客户端请求

### do\_command

do\_command在sql/sql\_parse.cc中：读取客户端传递的命令并分发。

注：do\_command被do\_handle\_one\_connection()调用（基于MySQL5.6），如果线程池中没有可用的缓存线程，则会通过本方法创建线程来处理用户请求。

新版本中，do\_command被handle\_connection()调用（sql/conn\_handler/Connection\_handler\_per\_thread.cc），这个handler\_connection接口被类Per\_thread\_connection\_handler函数add\_connection调用，处理请求的主要实现接口。

1. 循环调用do\_command，从socket中读取网络包，并且解析执行；

2、当远程客户端发送关闭连接COMMAND（比如COM\_QUIT，COM\_SHUTDOWN）时，退出循环；

3、调用close\_connection关闭连接（thd->disconnect()）；

4、根据返回结果，确定退出工作线程还是继续循环执行命令。

do\_command基本流程：

NET \*net= &thd->net;

packet\_length= my\_net\_read(net);

packet= (char\*) net->read\_pos;

command= (enum enum\_server\_command) (uchar) packet[0];

//从net结构中获取命令

dispatch\_command(command, thd, packet+1, (uint) (packet\_length-1));

//分发命令

#### my\_net\_read

#### dispatch\_command

在dispatch\_command函数中，根据命令的类型（用户请求信息的第一个字段表示该请求类型）进行分发。

基本流程：

1、先执行以下逻辑：

更新performance schema接口指标；

检查当前时间，如果超过了2038年，则停止该MySQL服务；

如果请求协议类型是PROTOCOL\_PLUGIN，但该命令不允许PROTOCOL\_PLUGIN，那么将断开该连接；

检查密码是否过期；

将该命令通知到相关的审计中audit；

String::shrink: Reclaim some memory

dispatch\_command

performance(schema) counter and profile

if past 2038, kill mysql server

如果请求协议类型是PROTOCOL\_PLUGIN，但该命令不允许PROTOCOL\_PLUGIN，那么将断开该连接

Enforce password expiration

mysql\_audit\_notify

mysql\_audit\_acquire\_plugins

check\_audit\_mask

acquire\_lookup\_mask，

acquire\_plugins：Acquire and lock any additional audit plugins, whose subscription mask overlaps with the lookup\_mask.

add\_audit\_mask

event\_class\_dispatch\_error

event\_class\_dispatch

plugins\_dispatch

st\_mysql\_audit::event\_notify

dispatch (switch ... case...):

2、最后遍进入dispatch，即执行switch ... case ... 这里列举了以下命令，我把这些命令分成三类：功能类、数据操作类、未实现。

thd->command=command;

switch(command) {

case COM\_INIT\_DB: ...;

case COM\_TABLE\_DUMP: ...;

case COM\_CHANGE\_USER: ...;

….

case COM\_QUERY: //如果是查询语句

{

alloc\_query(thd, packet, packet\_length)

//从网络数据包中读取Query并存入thd->query，即thd->set\_query(query, packet\_length);

mysql\_parse(thd, thd->query(), thd->query\_length(), &end\_of\_stmt);

// 解析查询语句

….

}

具体：

功能类：COM\_INIT\_DB，COM\_REGISTER\_SLAVE，COM\_RESET\_CONNECTION，COM\_CLONE，COM\_CHANGE\_USER，COM\_QUIT，COM\_BINLOG\_DUMP\_GTID，COM\_BINLOG\_DUMP，COM\_REFRESH，COM\_STATISTICS，COM\_PING，COM\_PROCESS\_INFO，COM\_SET\_OPTION，COM\_DEBUG，COM\_PROCESS\_KILL

数据操作类：COM\_STMT\_EXECUTE，COM\_STMT\_FETCH，COM\_STMT\_SEND\_LONG\_DATA，COM\_STMT\_PREPARE，COM\_STMT\_CLOSE，COM\_STMT\_RESET，COM\_QUERY，COM\_FIELD\_LIST

未实现：COM\_SLEEP，COM\_CONNECT，COM\_TIME，COM\_DELAYED\_INSERT，COM\_END

**1、COM\_INIT\_DB**

初始化当前连接的缺省数据库schema，即 `use XXX`命令。具体流程如下：

当新的schema name不合法时，返回失败，但当force\_switch打开时，将会switch到NULL\_CSTR中；

在合法情况下，做权限检查，是否有该schema的权限，如果没有也将返回错误；

获取该schema的charset和collation；

最后切到新schema中，在THD中修改，更新安全信息等；

1) mysql\_change\_db\_impl, switch to null-database; when db\_name is null/empty, and force\_switch is true;

2) is\_infoschema\_db && mysql\_change\_db\_impl, switch to INFORMATION SCHEMA

3) check\_and\_convert\_db\_name -> check\_table\_name, if error, mysql\_change\_db\_impl to NULL\_CSTR

4) ACL check: Security\_context::check\_access -> Security\_context::is\_access\_restricted\_on\_db -> Security\_context::filter\_access

acl\_get-> look up in cache, -> check access right for database and user: mysqld\_partial\_revokes -> save in cache.

master\_access, global privileges

check\_grant\_db: -> for each role, Security\_context::db\_acl(database check); has\_any\_table\_acl; has\_any\_routine\_acl (table check); column check; check\_grant\_db\_routine;

witch to null-database;

5) get\_default\_db\_collation

6) mysql\_change\_db\_impl, switch to new database

change new database in THD;

update security context;

Update db-charset environment variables

**2、COM\_REGISTER\_SLAVE**

在master节点上注册一个slave，注册完之后主将发送binlog到slave节点上，主要有三个步骤：

检查权限，是否有REPL\_SLAVE\_ACL权限；

清理该server\_id已经注册的slave信息；

将新的slave加入到slave\_list中；

1) check\_access(REPL\_SLAVE\_ACL)

2）clean old register: unregister\_slave

3) register struct SLAVE\_INFO

**3、COM\_RESET\_CONNECTION**

重置当前连接，不会重新创建连接和授权。

将状态变量复制到全局系统变量中；

重置系统的变量；

初始化当前连接对象THD，THD::init；

清空告警信息和profile information；

1) copy status\_var to global\_status\_var;

2) reset\_system\_status\_vars

3) THD::init

4) THD::cleanup: MDL\_context\_backup\_manager::create\_backup; transaction\_cache\_detach{create\_and\_insert\_new\_transaction}; trans\_rollback; transaction\_cache\_delete；mysql\_ha\_cleanup[clean up handler's table]; mysql\_ull\_cleanup[user level lock]; release\_all\_locking\_service\_locks; release\_backup\_lock; close\_temporary\_tables; sp\_cache\_clear{proc and func}

**4、COM\_CLONE**

SQLCOM\_CLONE命令，克隆远程的plugin到本地，并且保证一致性。

构造出Sql\_cmd\_clone对象；

通过plugin\_clone\_init初始化；

mysql\_declare\_plugin中的clone\_descriptor设置的Mysql\_clone对象调用plugin\_clone\_remote\_client发起远端clone请求；

1) Security\_context::has\_global\_grant, whether user has dynamic privileges(defined at runtime, https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html), User\_to\_dynamic\_privileges\_map

2) clone\_plugin\_lock,

will clone at the end.

3) send response,

4) Sql\_cmd\_clone::execute\_server,

Clone\_handler::clone\_remote\_server -> Mysql\_clone::clone\_server -> plugin\_clone\_remote\_client

match\_valid\_donor\_address -> scan\_donor\_list to validate whether valid (config clone\_valid\_donor\_list variable)

mysql\_clone\_start\_statement(sql/server\_component/clone\_protocol\_service.cc)

Client::clone -> Client::connect\_remote

clone\_client new thread.

error handling

**5、COM\_CHANGE\_USER**

修改当前连接的用户，并且重置连接。

清空当前连接，同COM\_RESET\_CONNECTION；

保存旧的user\_conn结构，用于失败恢复；

对新用户进行授权；

解析请求包；

进行验证；

更新thd信息；

保存用户连接信息；

证书验证；

检查密码是否到期；

检查当前用户超过最大连接数，如果超过则不允许修改；

修改当前schema的属性；

通知事件；

如果失败，将旧连接恢复；

1) THD::cleanup\_connection，同COM\_RESET\_CONNECTION

2)保存旧 struct user\_conn;

3) acl\_authenticate, -> server\_mpvio\_initialize

parse\_com\_change\_user\_packet

copy data from mpvio

copy\_and\_convert，将username和db name转成utf8

find\_mpvio\_user，从user的db中查找acl用户信息；

将数据包保存到mpvio中

do\_auth\_once，重新进行认证

server\_mpvio\_update\_thd，认证成功，更新thd信息

check\_and\_update\_password\_lock\_state，更新密码/锁等信息，如有效期

assign\_priv\_user\_host，将用户名/客户主机存到Security\_context；

授权proxy user，activate\_all\_granted\_and\_mandatory\_roles（授权角色身份）

acl\_check\_ssl，前面校验通过，开始校验ssl，如X509，客户指定的证书

check\_password\_lifetime，检查密码是否到期，date\_add\_interval\_with\_warn

如果当前已经执行太多query了，则不允许修改用户，get\_or\_create\_user\_conn

检查是否已经超过用户的最大连接数

mysql\_change\_db，修改当前database和对应的属性（见COM\_INIT\_DB）

4)mysql\_audit\_notify, 触发该thd的audit通知事件，见上

5)错误处理并恢复旧连接或者清空旧连接（成功）

**6、COM\_QUIT**

清除数据，退出error = true退出循环；

**7、COM\_BINLOG\_DUMP\_GTID**

请求发送binlog数据流。会执行以下逻辑：

检查是否有REPL\_SLAVE\_ACL权限；

读取求情数据；

杀死可能存在的多个binlog dump线程，具体产生多个线程的原因见下；

发送binlog数据。会做binlog初始化，然后发送binlog数据；

从unregister\_slave，从slave list中删除该从；

1) check\_global\_access，校验是否有REPL\_SLAVE\_ACL权限

2)读取请求数据，Gtid\_set::add\_gtid\_encoding，并填充gtid数据下表

3) kill\_zombie\_dump\_threads，杀死可能存在多个 binlog\_dump多余的线程(当从停掉，旧的binlog dump线程仍会执行，从重现连上时，会创建新的线程）；

4) mysql\_binlog\_send，发送请求gtid的binlog

Binlog\_sender::run

Binlog\_sender::init

Binlog\_sender::init\_heartbeat\_period

Binlog\_sender::check\_start\_file， check the start file and position is valid.

Binlog\_sender::init\_checksum\_alg

open bin log and send binlog

Binlog\_sender::send\_binlog

Binlog\_sender::get\_binlog\_end\_pos

Binlog\_sender::send\_events

Binlog\_sender::cleanup

5) unregister\_slave, 从slave list中删除

**8、COM\_BINLOG\_DUMP**

com\_binlog\_dump，同COM\_BINLOG\_DUMP\_GTID，除第二步不指定；

**9、COM\_REFRESH**

该操作已过时，调用 REFRESH或FLUSH statements。刷新缓存/权限GRANT，日志，表，连接主机信息，状态表，线程，从库等数据。在做这些之前，将会将gtid等落盘，并将事务回滚或者提交。这里涉及到2pc XA事务。

在这之后，将检查是否有RELOAD\_ACL的权限，之后遍进行加载数据。

combination REFRESH\_GRANT/REFRESH\_LOG/REFRESH\_TABLES/REFRESH\_HOSTS/REFRESH\_STATUS/REFRESH\_THREADS/REFRESH\_SLAVE/REFRESH\_MASTER

1) lex\_start

THD::init\_cost\_model->Cost\_model\_server::init， 初始化优化模型

LEX::new\_top\_level\_query, LEX::new\_query, 创建一个query expression对象(select\_lex\_unit)和query block object(select\_lex)

dd::info\_schema::Table\_statistics::invalidate\_cache, invalidate cache.

dd::info\_schema::Tablespace\_statistics::invalidate\_cache,

2) trans\_commit\_implicit，隐式地提交当前事务，但是不释放锁；

ha\_commit\_trans，

commit\_owned\_gtids，提交自身的事务

ending\_trans，检查事务是否可提交；

Gtid\_state::save

Gtid\_table\_persistor::save，写入到Gtid\_table中，

Gtid\_table\_access\_context::init,

System\_table\_access::open\_table，打开Gtid\_table；

Rpl\_info\_table\_access::before\_open, call lex\_start and mysql\_reset\_thd\_for\_next\_command

open\_n\_lock\_single\_table，打开并且锁定该文件

open\_and\_lock\_tables，常识打开文件，并且锁定文件，如果失败，则需要将已经开始的事务回滚，关闭打开的表（包括临时表），释放锁

如失败，则回滚事务，关闭表等资源

Gtid\_table\_persistor::write\_row，写入gtid信息，

Gtid\_table\_persistor::fill\_fields， SID/gno\_start/gno\_end

handler::ha\_write\_row，将gtid数据写入，

binlog\_log\_row

check\_table\_binlog\_row\_based，检查是否是基于行的binlog格式，如果需要，则声称binlog数据

add\_pke

generate\_mv\_hash\_pke，生成多个键值的hash值，

generate\_hash\_pke

generate\_hash\_pke，单个键值的hash值，

check\_foreign\_key，生成外键的hash值，会生成键名，每个字段对应的值；

Gtid\_table\_access\_context::deinit

System\_table\_access::close\_table，

ha\_rollback\_trans，如果失败，则回滚事务；

MYSQL\_BIN\_LOG::rollback，写入一个binlog的rollback行，

do\_binlog\_xa\_commit\_rollback，

ha\_rollback\_low，

write\_incident，如果中间出错

trans\_cannot\_safely\_rollback，

ha\_commit\_trans，如果成功，则提交事务；

commit\_owned\_gtids，

ha\_check\_and\_coalesce\_trx\_read\_only，

MYSQL\_BIN\_LOG::prepare，2pc开始，

ha\_prepare\_low

MYSQL\_BIN\_LOG::commit， 2pc commit，

binlog\_cache\_data::finalize

MYSQL\_BIN\_LOG::ordered\_commit

ha\_commit\_low

ha\_rollback\_low

close\_thread\_tables，关闭所有打开的表

mark\_temp\_tables\_as\_free\_for\_reuse

close\_thread\_table

mark\_used\_tables\_as\_free\_for\_reuse

mysql\_unlock\_tables，释放表的锁

close\_open\_tables

3) check\_global\_access， 检查是否有RELOAD\_ACL的权限；

4) handle\_reload\_request，重新加载权限/缓存的diff

reload\_acl\_caches，加载所有的权限到cache，

check\_engine\_type\_for\_acl\_table，

grant\_tables\_setup\_for\_open，列举表，

open\_and\_lock\_tables，打开上一步表，

check\_acl\_tables\_intact，

acl\_reload，

swap\_role\_cache

roles\_init，

roles\_graph\_init

default\_roles\_init

grant\_reload，表/列授权加载

5) close\_thread\_tables

6) MDL\_context::release\_transactional\_locks

release\_locks\_stored\_before，语句级别/事务级别的

7)返回处理结果

**10、COM\_STATISTICS**

获得内部统计的可读信息。

1) calc\_sum\_of\_all\_status

Global\_THD\_manager::do\_for\_all\_thd\_copy， copy all status of thd to global variable.

2)发送数据

**11、COM\_PING**

直接返回ok。

**12、COM\_PROCESS\_INFO**

获得进程处理信息。

1) check\_global\_access, 检验PROCESS\_ACL权限

2) mysqld\_list\_processes

THD::send\_result\_metadata

Protocol\_local::start\_result\_metadata -> Protocol\_local::start\_row

Protocol\_local::send\_field\_metadata

Protocol\_local::end\_result\_metadata -> Ed\_row

List\_process\_list, 收集所有进/线程信息

**13、COM\_SET\_OPTION**

设置当前连接的属性。

针对该客户连接设置的option

add\_client\_capability

remove\_client\_capability

**14、COM\_DEBUG**

在mysql服务上触发打印debug信息。包括了当前锁信息，表缓存信息，队列，等信息。

1) check\_global\_access, 检查是否有SUPER\_ACL权限；

2) mysql\_print\_status，打印状态，

thr\_print\_locks，写debug信息

print\_cached\_tables，

Table\_cache\_manager::print\_tables，打印所有表缓存实例，

process\_key\_caches， 打印key cache

print\_key\_cache\_status

calc\_sum\_of\_all\_status，

display\_table\_locks，打印锁信息，

Events::dump\_internal\_status，打印schedule和缓存队列的状态，

Event\_scheduler::dump\_internal\_status，

Event\_queue::dump\_internal\_status

3) Query\_logger::general\_log\_print，

mysql\_audit\_general\_log，调用GENERAL LOG 的审计类

Query\_logger::general\_log\_write

4) LO\_graph::dump\_txt，dump Lock order图到文本中

**15、COM\_PROCESS\_KILL**

终止一个连接。

sql\_kill:

kill\_one\_thread

get thd,

获取用户是否有要kill掉的thd权限，包括：SUPER\_ACL或是否有CONNECTION\_ADMIN权限或自己当前thd

THD::awake

shutdown\_clone\_vio，关闭vio

shutdown\_active\_vio, post\_kill\_notification

##### alloc\_query

##### mysql\_parse

mysql\_parse()是主要解析函数。

lex\_start(thd); //将thd->lex对象内容重新清理置为初始化状态

/\* lex：语法分析对象

注: thd是当前线程上下文信息类，后续与用户处理相关函数都会传入这个类，估计是c++没有像java的ThreadLocal那么方便的类，所以老是要这么麻烦地传来传去的。

\*/

//是否命中query cache，如有，则返回结果。如没有，则作如下动作

if(query\_cache\_send\_result\_to\_client(thd,rawbuf,length)<=0){

bool err = parse\_sql(thd,parser\_state,NULL); //解析sql

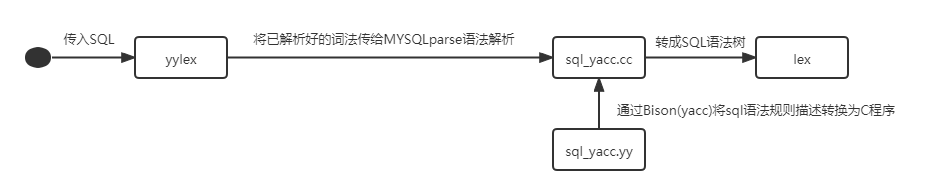
mysql\_execute\_command(thd); //执行

}

###### lex\_start

###### parse\_sql

MySQL解析过程如下：



mysql是使用了开始的bison（即yacc的开源版）作为sql语法解析器。

在lex词法解析阶段，会解析出select，from，where这几个token，接下来sql\_yacc.cc的MYSQLparse会根据上面的token解析出语法树，yacc是使用巴科斯范式(BNF)表达语法规则。

对于执行查询语句select \* from tb\_name where 1=0 and id=0，如下：

select\_from:

FROM join\_table\_list where\_clause group\_clause having\_clause

opt\_order\_clause opt\_limit\_clause procedure\_analyse\_clause

{

Select->context.table\_list=

Select->context.first\_name\_resolution\_table=

Select->table\_list.first;

}

| FROM DUAL\_SYM where\_clause opt\_limit\_clause

/\* oracle compatibility: oracle always requires FROM clause,

and DUAL is system table without fields.

Is "SELECT 1 FROM DUAL" any better than "SELECT 1" ?

Hmmm :) \*/

;

where\_clause:

/\* empty \*/ { Select->where= 0; }

| WHERE

{

Select->parsing\_place= IN\_WHERE;

}

expr

{

SELECT\_LEX \*select= Select;

select->where= $3;

select->parsing\_place= NO\_MATTER;

if ($3)

$3->top\_level\_item();

}

;

parse\_sql()方法执行完后,我们可以在gdb中查看语法树lex：

##查看select\_lex->where

(gdb) call print\_where(lex->select\_lex->where,"",QT\_WITHOUT\_INTRODUCERS)

WHERE:() 0x7fff98005e10 ((1 = 0) and (`id` = 0))

(gdb) p lex->select\_lex->table\_list->first

$9 = (TABLE\_LIST \*) 0x7fff98005260

##查看sql使用的database

(gdb) p $9->db

$10 = 0x7fff980057c0 "db\_name"

(gdb) p $9->table\_name

$11 = 0x7fff98005218 "tb\_name"

###### mysql\_execute\_command

在mysql\_execute\_command中，根据命令类型，转到相应的执行函数。基本流程：

switch (lex->sql\_command) {

LEX \*lex= thd->lex;//获取解析后的语法树

TABLE\_LIST \*all\_tables;

case SQLCOM\_SELECT:

check\_table\_access(thd, lex->exchange ? SELECT\_ACL | FILE\_ACL : SELECT\_ACL, all\_tables, UINT\_MAX, FALSE); //检查用户权限

execute\_sqlcom\_select(thd, all\_tables); //执行select命令

break;

case SQLCOM\_INSERT:

{

res= insert\_precheck(thd, all\_tables) //rights

mysql\_insert(thd, all\_tables, lex->field\_list, lex->many\_values,

lex->update\_list, lex->value\_list,

lex->duplicates, lex->ignore);

break;

execute\_sqlcom\_select

在execute\_sqlcom\_select函数中，基本流程：

res= open\_and\_lock\_tables(thd, all\_tables)//directly and indirectly

res= handle\_select(thd, lex, result, 0);

handle\_select

mysql\_select

handle\_select在sql\_select.cc中，调用mysql\_select，这是SQL真正执行入口。

在mysql\_select中，基本流程：

join->prepare();//预处理，Prepare of whole select (including sub queries in future).

join->optimize();//查询优化，global select optimisation.

join->exec();//执行

JOIN::prepare()/Sql\_cmd\_dml::prepare()

执行sql查询优化计划前的准备工作。  
其中setup\_wild()方法会把查询语句中的“\*”扩展为表上的所有列。

**1、setup\_wild()**

while (wild\_num && (item= it++))

{

if (item->type() == Item::FIELD\_ITEM &&

//如果field值为\*

((Item\_field\*) item)->field\_name[0] == '\*' &&

!((Item\_field\*) item)->field)

{

if (subsel &&

subsel->substype() == Item\_subselect::EXISTS\_SUBS)

{

...

}else if (insert\_fields(thd, ((Item\_field\*) item)->context,

((Item\_field\*) item)->db\_name,

((Item\_field\*) item)->table\_name, &it,

any\_privileges))

{

...

}

}

**1.1 insert\_field**

//字段迭代器

Field\_iterator\_table\_ref field\_iterator;

field\_iterator.set(tables);

for (; !field\_iterator.end\_of\_fields(); field\_iterator.next())

{

Item \*item;

item= field\_iterator.create\_item(thd);

if (!found)

{

found= TRUE;

it->replace(item);

}

else

it->after(item); /\*将当前sql语句的表的字段一一加到fields\_list中\*/

}

JOIN::optimize()

JOIN::optimize()函数主要功能是对sql各种优化，包括条件下推，关联索引列，计算最优查询优化执行计划。

以SQL：select \* from tb\_name where 1=0 and id=0为例，与本请求sql优化相关的是optimize\_cond()方法。

处理本sql时，optimize\_cond()方法最终会将select\_lex->cond\_value置为Item::COND\_FALSE，针对这个结果，后续处理如下:

if (select\_lex->cond\_value == Item::COND\_FALSE ||

select\_lex->having\_value == Item::COND\_FALSE ||

(!unit->select\_limit\_cnt && !(select\_options & OPTION\_FOUND\_ROWS)))

{ /\* Impossible cond \*/

zero\_result\_cause= select\_lex->having\_value == Item::COND\_FALSE ?

"Impossible HAVING" : "Impossible WHERE";

tables= 0;

primary\_tables= 0;

best\_rowcount= 0;

goto setup\_subq\_exit;

}

1. **optimize\_cond()**

该方法主要简化代码：

//等式合并

build\_equal\_items(thd,conds,NULL,true,join\_list,cond\_equal);

//常量求值

propagate\_cond\_constants(thd, (I\_List<COND\_CMP> \*) 0, conds, conds);

//条件去除

remove\_eq\_conds(thd, conds, cond\_value) ;

在刚进这个方法时，我们可以打印 conds对象的值：

(gdb) p call print\_where(conds,"",QT\_WITHOUT\_INTRODUCERS)

WHERE:() 0x7fff98005e10 ((1 = 0) and (`db\_name`.`tb\_name`.`id` = 0))

remove\_eq\_conds()方法会优化掉条件中1=0

**1.1 remove\_eq\_conds()**

本方法会调用: internal\_remove\_eq\_conds(thd, cond, cond\_value);

// Scan all the condition

**1.1.1 internal\_remove\_eq\_conds**

while ((item=li++))

{

/×这里会取当前条件组的第一个条件递归调用本方法

在递归的方法中会判断到item->const\_item()为true,

并对1=0进行求值:

\*tmp\_cond\_value= eval\_const\_cond(cond) ? Item::COND\_TRUE : Item::COND\_FALSE;

×/

Item \*new\_item=internal\_remove\_eq\_conds(thd, item, &tmp\_cond\_value);

switch (tmp\_cond\_value) {

case Item::COND\_OK: // Not TRUE or FALSE

if (and\_level || \*cond\_value == Item::COND\_FALSE)

\*cond\_value=tmp\_cond\_value;

break;

//当前1=0的条件会进入 Item::COND\_FALSE

case Item::COND\_FALSE:

if (and\_level)

{

\*cond\_value=tmp\_cond\_value;

return (Item\*) 0; // Always false

}

break;

}

这里在gdb中如果输入:

(gdb) call print\_where(conds,"",QT\_WITHOUT\_INTRODUCERS)

WHERE:() (nil)

JOIN::exec()/Sql\_cmd\_dml::execute\_inner

调用do\_select，接着调用handle\_select，不断处理结果集。

mysql\_insert

在mysql\_insert函数中，

open\_and\_lock\_tables(thd, table\_list)

mysql\_prepare\_insert(); //prepare item in INSERT statment

while ((values= its++))

write\_record(thd, table ,&info);//写入新的数据

write\_record

在write\_record函数中，

table->file->ha\_write\_row(table->record[0])

ha\_write\_row在Handler.cc中，只是一个接口

write\_row(buf); //调用表存储所用的引擎

## 处理结果和关闭连接

# 总结