# 主要模块

源码结构（MySQL-5.5.0-m2）

BUILD: 内含在各个平台、各种编译器下进行编译的脚本。如compile-pentium-debug表示在pentium架构上进行编译的脚本。

client: 客户端工具，如mysql, mysqladmin之类。

cmd-line-utils: readline, libedit工具。

config: 给aclocal使用的配置文件。

dbug: 提供一些调试用的宏定义。

extra: 提供innochecksum，resolveip等额外的小工具。

include: 包含的头文件。

libmysql: 库文件，生产libmysqlclient.so。

libmysql\_r: 线程安全的库文件，生成libmysqlclient\_r.so。

libservices: 5.5.0中新加的目录，实现了打印功能。

man: 手册页。

mysql-test: mysqld的测试工具一套。

mysys: 为跨平台计，MySQL自己实现了一套常用的数据结构和算法，如string, hash等。

netware: 在netware平台上进行编译时需要的工具和库。

plugin: mysql以插件形式实现的部分功能。

pstack: 异步栈追踪工具。

regex: 正则表达式工具。

Scripts: 提供脚本工具，如mysql\_install\_db等

sql: mysql主要代码，将会生成mysqld文件。

sql-bench: 一些评测代码。

sql-common: 存放部分服务器端和客户端都会用到的代码。

storage: 存储引擎所在目录，如myisam, innodb, ndb等。

这个目录包含了所谓的MySQL存储引擎（storage engine）。存储引擎是数据库系统的核心，封装了数据库文件的操作，是数据库系统是否强大最重要的因素。MySQL实现了一个抽象接口层，叫做handler（sql/handler.h），其中定义了接口函数，比如：ha\_open，ha\_index\_end，ha\_create等等，存储引擎需要实现这些接口才能被系统使用。对于具体每种引擎的特点，推荐大家去看mysql的在线文档: <http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html>。

Strings: string库。

support-files: my.cnf示例配置文件。

tests: 测试文件所在目录。

unittest: 单元测试。

vio: virtual io系统，是对network io的封装。

win: 给windows平台提供的编译环境。

zip: zip库工具

# 主要数据结构

## THD

代码位置：sql/sql\_class.h

包含处理用户请求时需要的相关数据，每个连接会有一个线程来处理，在一些高层函数中，此数据结构常被当作第一个参数传递。

THD类继承了三个父类：

MDL\_context\_owner：作为Metadata Lock 拥有者相关的接口，其中元数据边界控制（进/退），元数据信息通知等，它是一个虚类。

Query\_arena：管理语句元素的类，主要针对指语句（包括store procedure）解释成抽象语法树后的节点

Open\_tables\_state：管理线程打开和加锁的表的状态，保持了一个临时表信息和锁信息，子类有Open\_tables\_backup

除这些外，还包括与锁相关的数据结构如MDL\_context，接口语法语义相关如LEX，日志和事务以及执行数据和优化参数等。

NET net; // 客户连接描述符

Protocol \*protocol; // 当前的协议

Protocol\_text protocol\_text; // 普通协议

Protocol\_binary protocol\_binary; // 二进制协议

HASH user\_vars; //用户变量的hash值

String packet; // 网络IO时所用的缓存

String convert\_buffer; // 字符集转换所用的缓存

struct sockaddr\_in remote; //客户端socket地址

THR\_LOCK\_INFO lock\_info; // 当前线程的锁信息

THR\_LOCK\_OWNER main\_lock\_id; // 在旧版的查询中使用

THR\_LOCK\_OWNER \*lock\_id; //若非main\_lock\_id, 指向游标的lock\_id

pthread\_mutex\_t LOCK\_thd\_data;

//thd的mutex锁，保护THD数据（thd->query, thd->query\_length）不会被其余线程访问到

Statement\_map stmt\_map;

//prepared statements和stored routines 会被重复利用

int insert(THD \*thd, Statement \*statement); // statement的hash容器

class Statement::

LEX\_STRING name;

LEX \*lex; //语法树描述符

bool set\_db(const char \*new\_db, size\_t new\_db\_len)

void set\_query(char \*query\_arg, uint32 query\_length\_arg);

{

pthread\_mutex\_lock(&LOCK\_thd\_data);

set\_query\_inner(query\_arg, query\_length\_arg);

pthread\_mutex\_unlock(&LOCK\_thd\_data);

}

## Lex

MySQL通过THD的父类字段Statement::lex保存解析出来的语法树信息。struct LEX继承于Query\_tables\_list。LEX中涉及的查询语法树的核心成员包括：

unit：语法树顶层节点；

select\_lex：顶层第一个select语句节点；

all\_selects\_list：通过link\_next和link\_prev构成所有select子句的双向链表；

query\_tables：通过next\_global和prev\_global构成所有talbe的双向链表；

query\_tables\_last：指向query\_tables中最后一个元素；

注：其他成员不作描述，在sql/sql\_lex.h源码文件中。

### select语法树

#### 核心数据结构

select语法树由SELECT\_LEX\_UNIT（即st\_select\_lex\_unit）节点和SELECT\_LEX（即st\_select\_lex）节点组成。两个class都继承自st\_select\_lex\_node。SELECT\_LEX的核心成员如下：

table\_list：通过next\_local串联起来的表结构，也就是from后面的表；

top\_join\_list：将from后面的表根据jion语法规则生成的jion树；

where：where语句后面的条件表示树；

group\_list：group分组条件；

having：having条件；

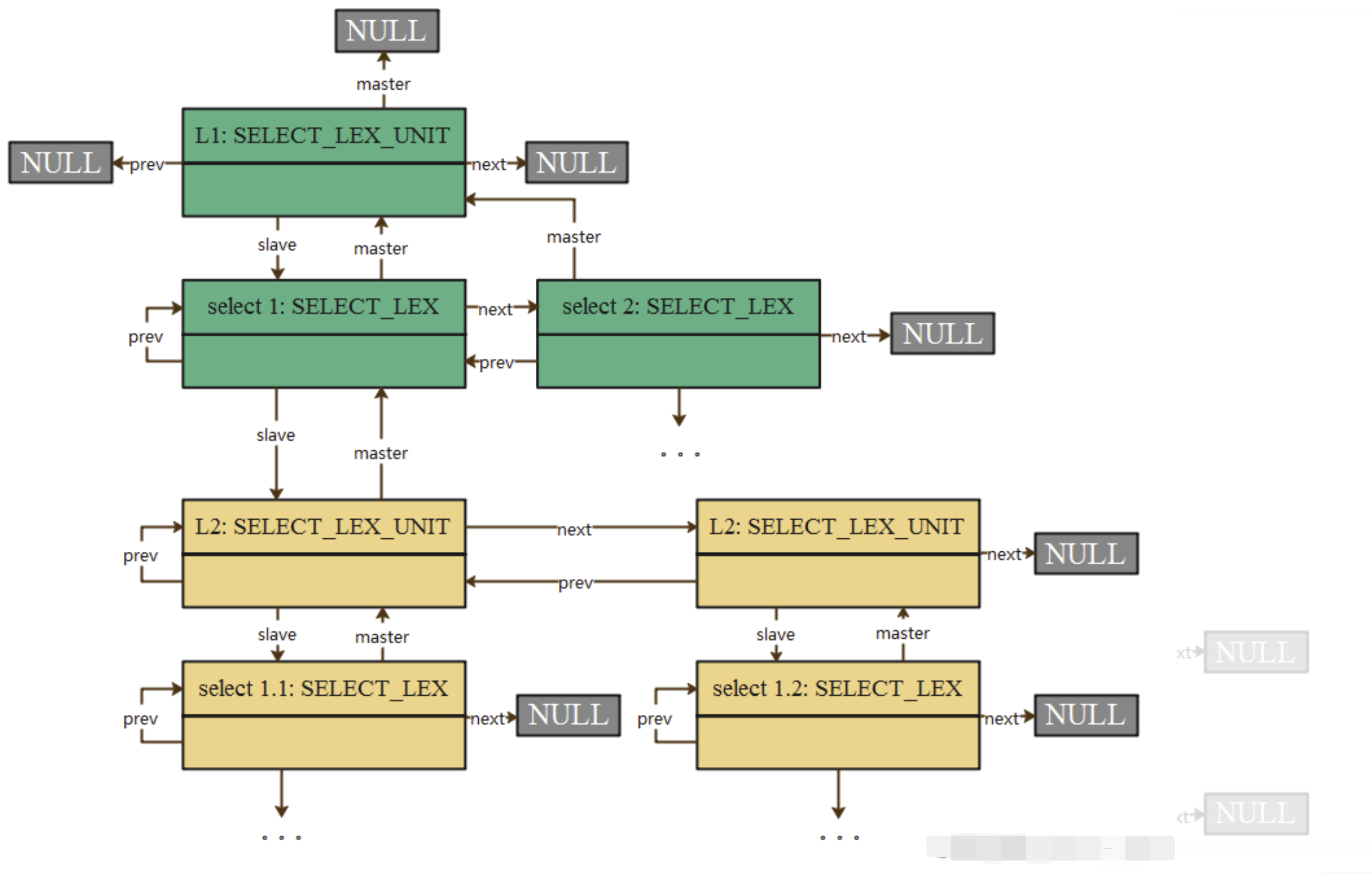
item\_list：select语句后面的字段列表；

order\_list：order排序方式；

select\_limit，offset\_limit：行限制条件；

#### 语法树结构

查询语句的语法树结构如下：



整个语法树由SELECT\_LEX\_UNIT作为顶层，下面挂unit的多个select语句；每个select子句的顶层SELECT\_LEX\_UNIT挂到所属的select主句上面。THD::lex->unit指向顶层L1；THD::lex->select\_lex指向顶层select 1。

### from子句

#### 解析原理

整个from表格解析规则如下：

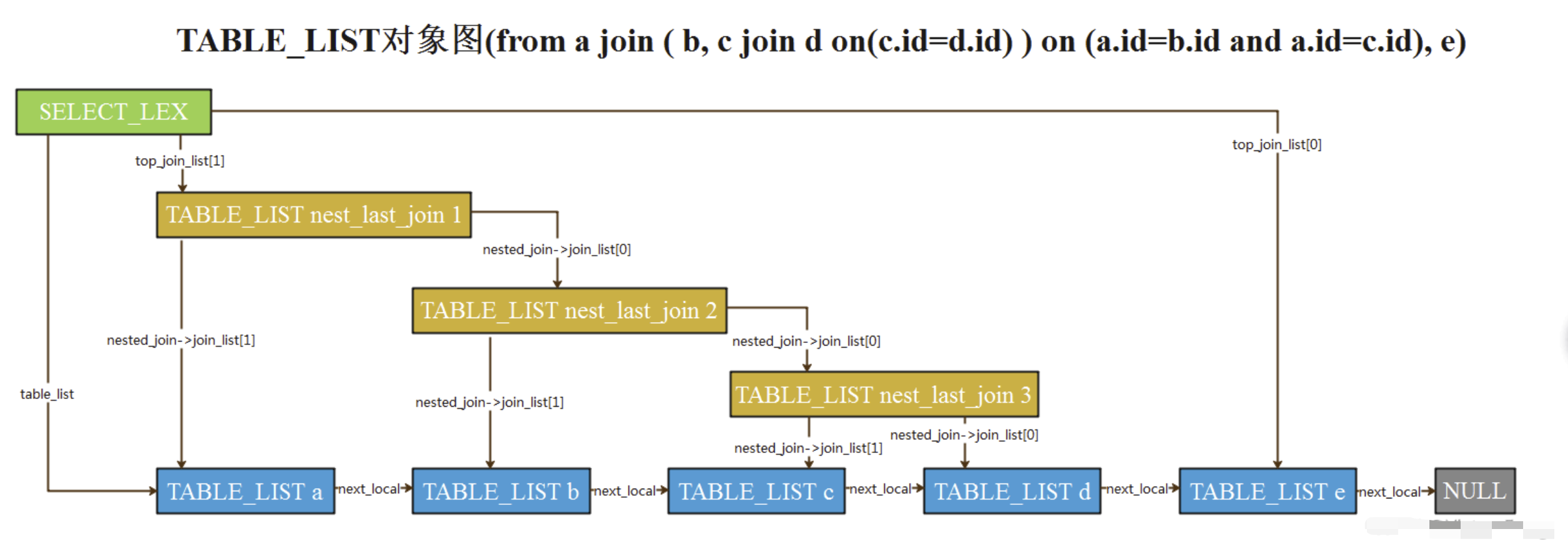
规则1：JOIN表达式：=（JOIN表达式 | JOIN嵌套）JOIN（JOIN表达式 | JOIN嵌套）

规则2：JOIN嵌套：=’(’ （表引用 | JOIN表达式）（，表引用 | ，JOIN表达式）\* ‘)’ | 表引用

也就是说一个jion表达式是由join表达式或join嵌套组成；一个join嵌套可以是一个表引用，或者多个表引用或join表达式通过“,”相隔、且首尾括号包围。

#### 语法树结构

from后面的表解析成TABLE\_LIST结构，所有的TABLE\_LIST按如下方式串联：



### where语句

where语句都是抽象类Item的子类一层一层组成语法树结构。

## NET

**代码位置：**sql/mysql\_com.h

网络连接描述符，对内部数据包进行了封装，是client和server之间的通信协议。

Vio \*vio; //底层的网络I/O socket描述符

unsigned char \*buff,\*buff\_end,\*write\_pos,\*read\_pos; //缓存相关

unsigned long remain\_in\_buf,length, buf\_length, where\_b;

unsigned long max\_packet,max\_packet\_size; //当前值;最大值

unsigned int pkt\_nr,compress\_pkt\_nr; //当前（未）压缩包的顺序值

my\_bool compress; //是否压缩

unsigned int write\_timeout, read\_timeout, retry\_count; //最大等待时间

unsigned int \*return\_status; //thd中的服务器状态

unsigned char reading\_or\_writing;

unsigned int last\_errno; //返回给客户端的错误号

unsigned char error;

## TABLE

**代码位置：**sql/table.h

数据库表描述符，分成TABLE和TABLE\_SHARE两部分。

handler \*file; //指向这张表在storage engine中的handler的指针

THD \*in\_use;

Field \*\*field;

uchar \*record[2];

uchar \*write\_row\_record;

uchar \*insert\_values;

key\_map covering\_keys;

key\_map quick\_keys, merge\_keys;

key\_map keys\_in\_use\_for\_query;

key\_map keys\_in\_use\_for\_group\_by;

key\_map keys\_in\_use\_for\_order\_by;

KEY \*key\_info;

HASH name\_hash; //数据域名字的hash值

MEM\_ROOT mem\_root; //内存块

LEX\_STRING db;

LEX\_STRING table\_name;

LEX\_STRING table\_cache\_key;

enum db\_type db\_type //当前表的storage engine类型

enum row\_type row\_type //当前记录是定长还是变长

uint primary\_key;

uint next\_number\_index; //自动增长key的值

bool is\_view ;

bool crashed;

## FIELD

**代码位置：**sql/field.h

域描述符，是各种字段的抽象基类。

uchar \*ptr; // 记录中数据域的位置

uchar \*null\_ptr; // 记录 null\_bit 位置的byte

TABLE \*table; // 指向表的指针

TABLE \*orig\_table; // 指向原表的指针

const char \*\*table\_name, \*field\_name;

LEX\_STRING comment;

key\_map key\_start, part\_of\_key, part\_of\_key\_not\_clustered;

key\_map part\_of\_sortkey;

enum utype { NONE,DATE,SHIELD,NOEMPTY,CASEUP,PNR,BGNR,PGNR,YES,NO,REL,

CHECK,EMPTY,UNKNOWN\_FIELD,CASEDN,NEXT\_NUMBER,

INTERVAL\_FIELD,BIT\_FIELD, TIMESTAMP\_OLD\_FIELD, CAPITALIZE, BLOB\_FIELD,

TIMESTAMP\_DN\_FIELD, TIMESTAMP\_UN\_FIELD, TIMESTAMP\_DNUN\_FIELD};

…..

virtual int store(const char \*to, uint length,CHARSET\_INFO \*cs)=0;

inline String \*val\_str(String \*str) { return val\_str(str, str); }

## Utility API Calls API

各种核心的工具，例如内存分配，字符串操作或文件管理。标准C库中的函数只使用了很少一部分，C++中的函数基本没用。

void \*my\_malloc(size\_t size, myf my\_flags)

//对malloc的封装

size\_t my\_write(File Filedes, const uchar \*Buffer, size\_t Count, myf MyFlags)

//对write的封装

## Preprocessor Macros处理器宏

Mysql中使用了大量的C预编译，随编译参数的不同最终代码也不同。

#define max(a, b) ((a) > (b) ? (a) : (b)) //得出两数中的大者

do \

{ \

char compile\_time\_assert[(X) ? 1 : -1] \

\_\_attribute\_\_ ((unused)); \

} while(0)

使用gcc的attribute属性指导编译器行为

• Global variables 全局变量

• configuration settings

• server status information

• various data structures shared among threads

主要包括一些全局的设置，服务器信息和部分线程间共享的数据结构。

struct system\_status\_var global\_status\_var; //全局的状态信息

struct system\_variables global\_system\_variables; //全局系统变量

# 主要调用流程

## MySQL启动

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

## 监听接收链接

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

### do\_command

### end\_connection

## 预处理连接

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);

}

### add\_connection

### lex\_start

### prepare\_new\_connection\_state

### do\_command

### end\_connection

## 处理

### 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

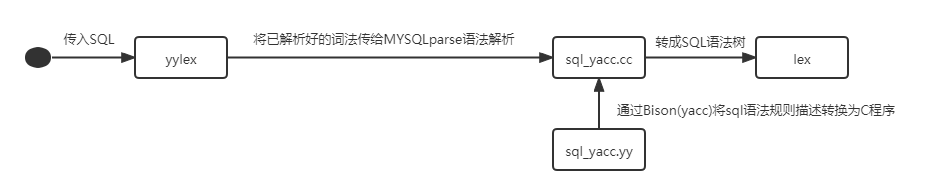
mysql\_execute\_command(thd); //执行

}

###### 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()

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()

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

## 总结