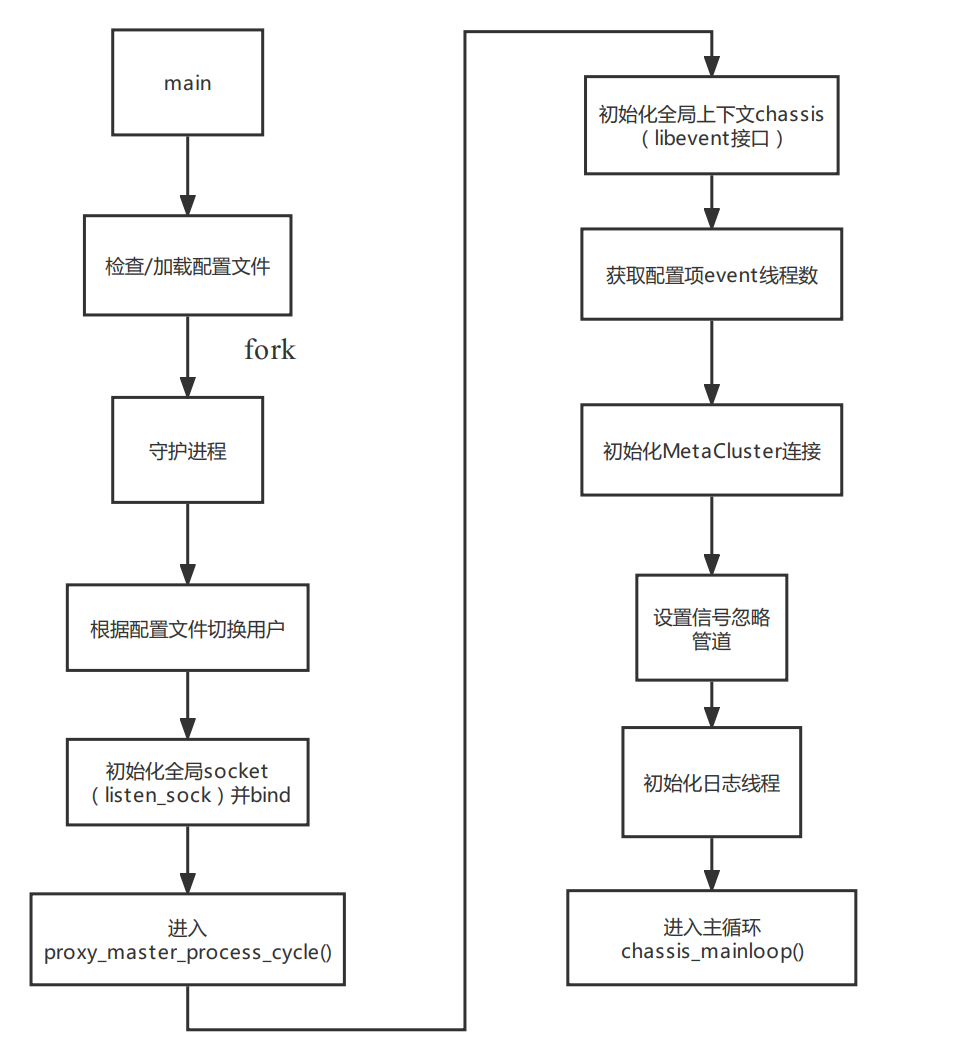
# 事件处理

整个main函数的大致流程：



流程分析：

1. 检查配置文件并加载配置项；
2. 根据配置文件中的用户切换用户；
3. 调用chassis\_unix\_daemonize（fork）创建守护进程；
4. 调用init\_listen\_sock初始化全局socket（listen\_socke）并执行bind操作；
5. 进入工作线程控制流程proxy\_master\_process\_cycle，执行工作线程的创建，kill，重启操作；
6. 进入proxy\_worker\_process\_cycle执行工作线程的主要动作：
7. 调用chassis\_new（libevent接口初始化event\_base）初始化全局上下文变量chassis；
8. 获取配置文件中event\_threads配置项；
9. 如果开启全局一致性读，需要初始化MetaCluster连接；
10. 设置信号忽略管道（后面会使用管道进行通信）；
11. 调用log\_init初始化日志清理线程；
12. 进入主循环chassis\_mainloop执行事件处理线程的创建、启动、监听、回收。

## add\_inherited\_sockets

## chassis\_unix\_daemonize

fork创建守护进程。

## init\_listen\_sock

初始化全局socket（listen\_socke）并执行bind操作。

### network\_socket\_new

### network\_address\_set\_address

### network\_socket\_bind

## proxy\_master\_process\_cycle

启动工作线程，并且不断循环判断进程状态并重启。

### proxy\_start\_worker\_processes

#### proxy\_worker\_process\_cycle

1. 调用chassis\_new初始化event\_base
2. 读取配置文件中事件线程数，初始化
3. 如果开启全局一致性读，则初始化MetaCluster的连接
4. 调用主循环函数chassis\_mainloop
5. 调用chassis\_free清理event\_base

##### GatewayConf::init

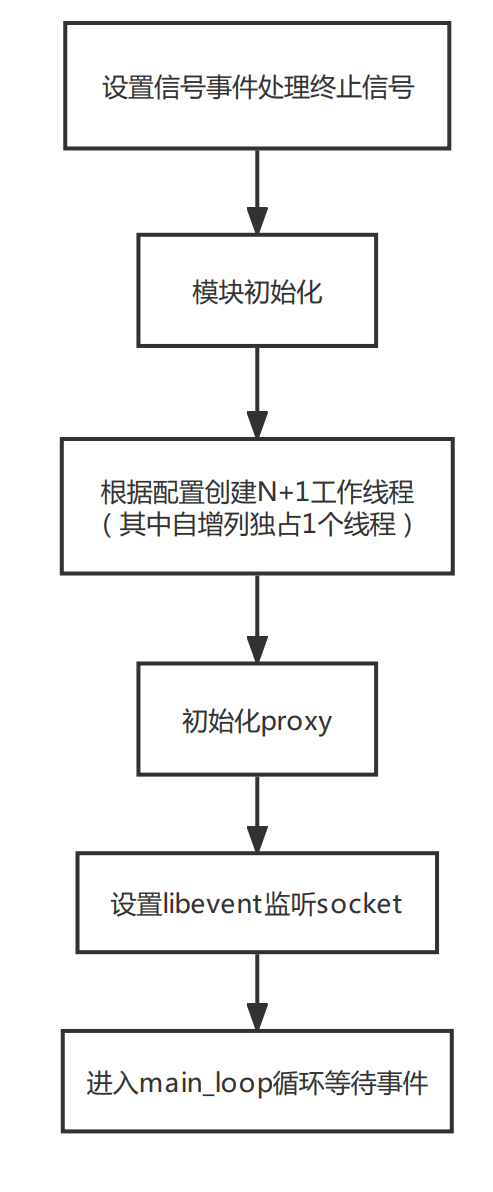
初始化日志清理线程log-clean、慢日志上报线程slow-report、配置加载线程conf-reload。

##### chassis\_new

初始化全局chassis对象指针（event\_base\_new）

##### item\_create\_init

##### chassis\_mainloop



这里是创建全局的event上下文之后执行监听的主要流程：

1. 调用signal\_set设置信号事件处理终止信号；
2. 调用module\_init\_t::initialize\_all初始化组件；
3. 根据前面配置项中设置的event\_threads执行：
4. 调用chassis\_event\_thread\_new初始化event\_base；
5. 调用chassis\_event\_threads\_init\_thread进行事件线程的初始化设置；
6. 调用chassis\_event\_threads\_add将event\_thread添加到chas->threads中组成线程池。
7. 调用chassis\_event\_threads\_start启动所有线程；
8. 调用proxy\_init初始化proxy；
9. 调用init\_global\_part\_rebalance初始化重分布；
10. 调用add\_listen\_sock设置全局上下文chassis对应的event；
11. 调用main\_loop进入事件处理的主循环。

###### sigterm\_handler

设置信号事件处理终止函数

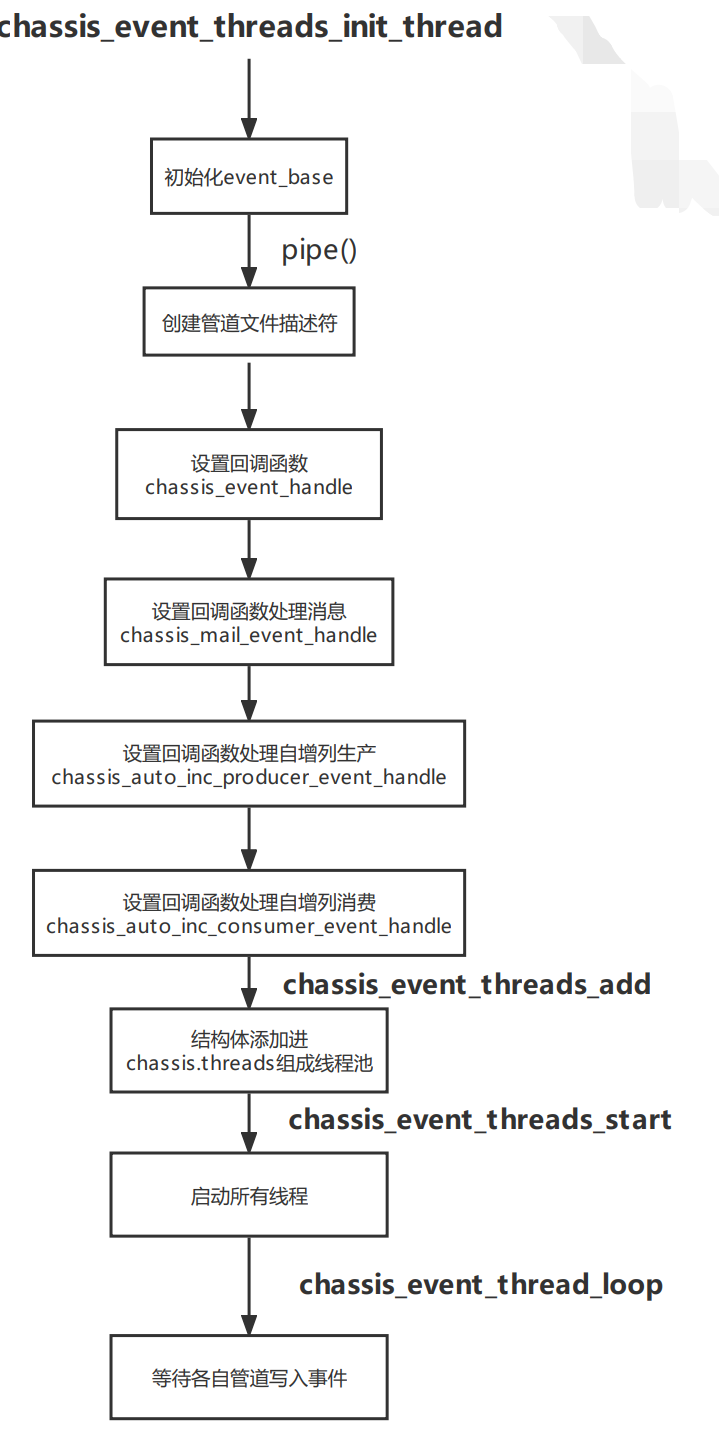
###### module\_init\_t::initialize\_all

init\_mysql\_library，初始化本地mysql，用于后续分布式事务记录和join的处理

###### chassis\_event\_thread\_new

构造一个新的libevent线程。

###### chassis\_event\_threads\_init\_thread



event\_base\_new

初始化event\_base

pipe

创建管道文件描述符

event\_set

chassis\_event\_handle

设置回调函数chassis\_event\_handle

event\_base\_set

event\_add

chassis\_mail\_event\_handle

chassis\_auto\_inc\_producer\_event\_handle

chassis\_auto\_inc\_consumer\_event\_handle

###### chassis\_event\_threads\_add

结构体添加进threads->event\_threads组成线程池。

###### chassis\_event\_threads\_start

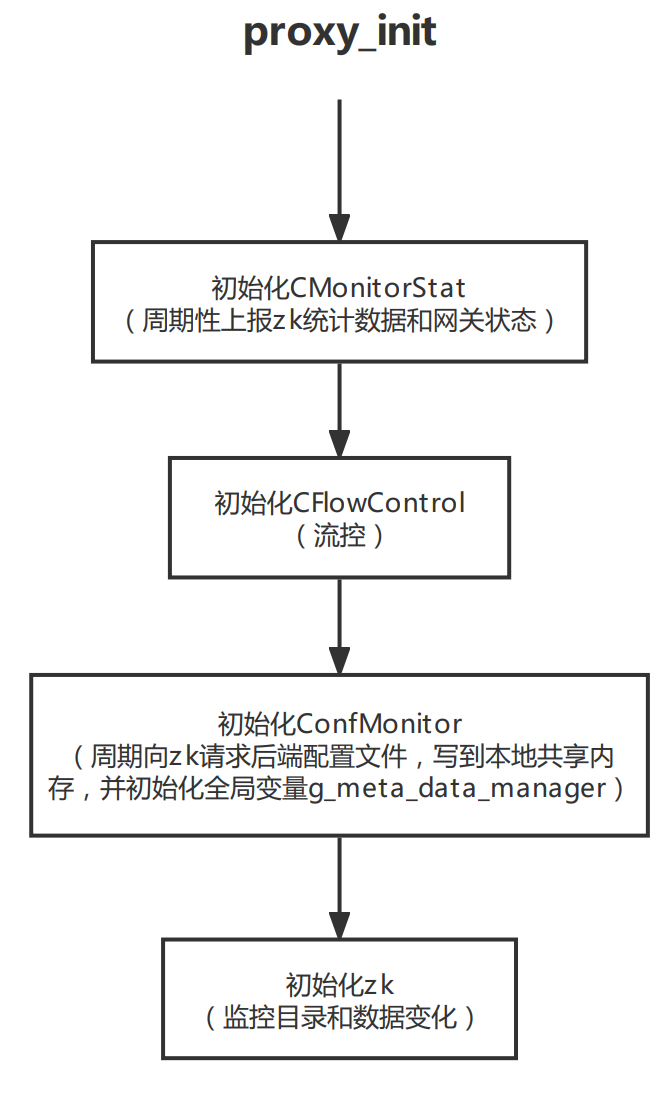
启动所有线程

chassis\_event\_thread\_loop

等待各自的管道写入事件

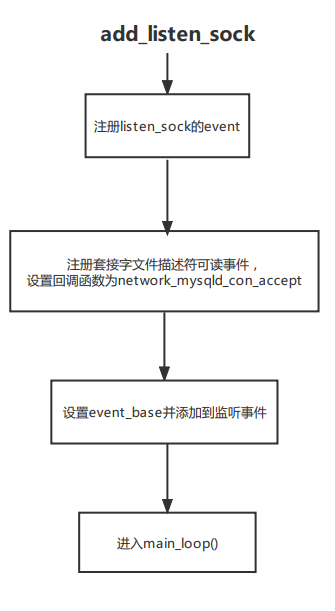
chassis\_event\_thread\_loop\_auto

###### proxy\_init



###### init\_global\_part\_rebalance

###### add\_listen\_sock



event\_set

network\_mysqld\_con\_accept

注册套接字文件描述符可读事件，设置回调函数为network\_mysqld\_con\_accept

event\_base\_set

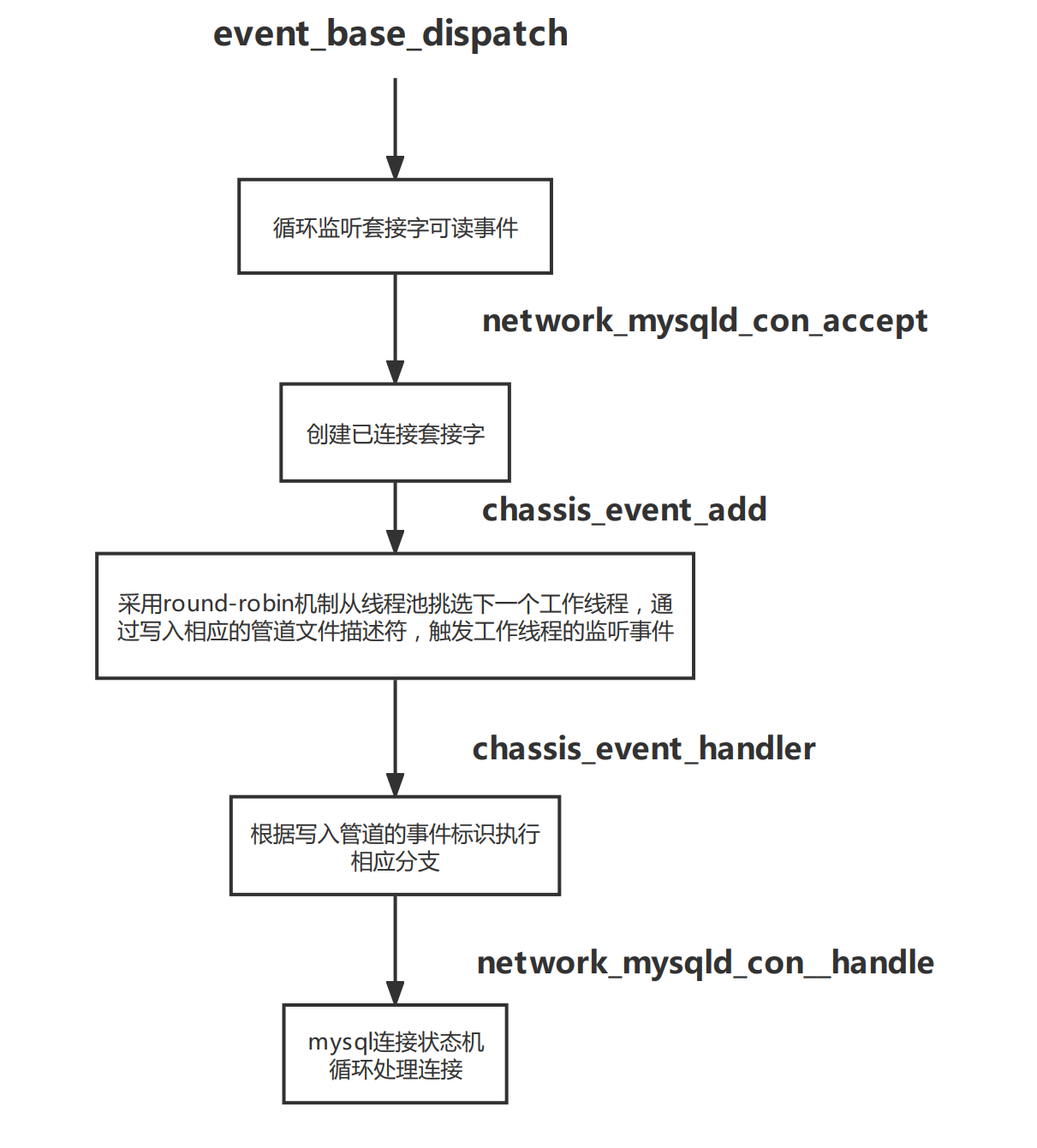
event\_add

###### main\_loop

chassis\_is\_shutdown

event\_base\_loopexit

event\_base\_dispatch



chassis\_event\_add

采用roud-robin机制从线程池挑选下一个工作线程，通过写入相应管道文件描述符，触发工作线程的监听事件

##### chassis\_set\_shutdown\_location

##### chassis\_free

##### item\_create\_cleanup

# MySQL通信协议

## client<->proxy

函数：

proto/client\_proxy.cpp：定义client->proxy的连接初始化（SSL）、握手信号等

proto/protocol.cpp：设置proxy发送client协议包

proto/viossl.cpp：解析包

## proxy<->server

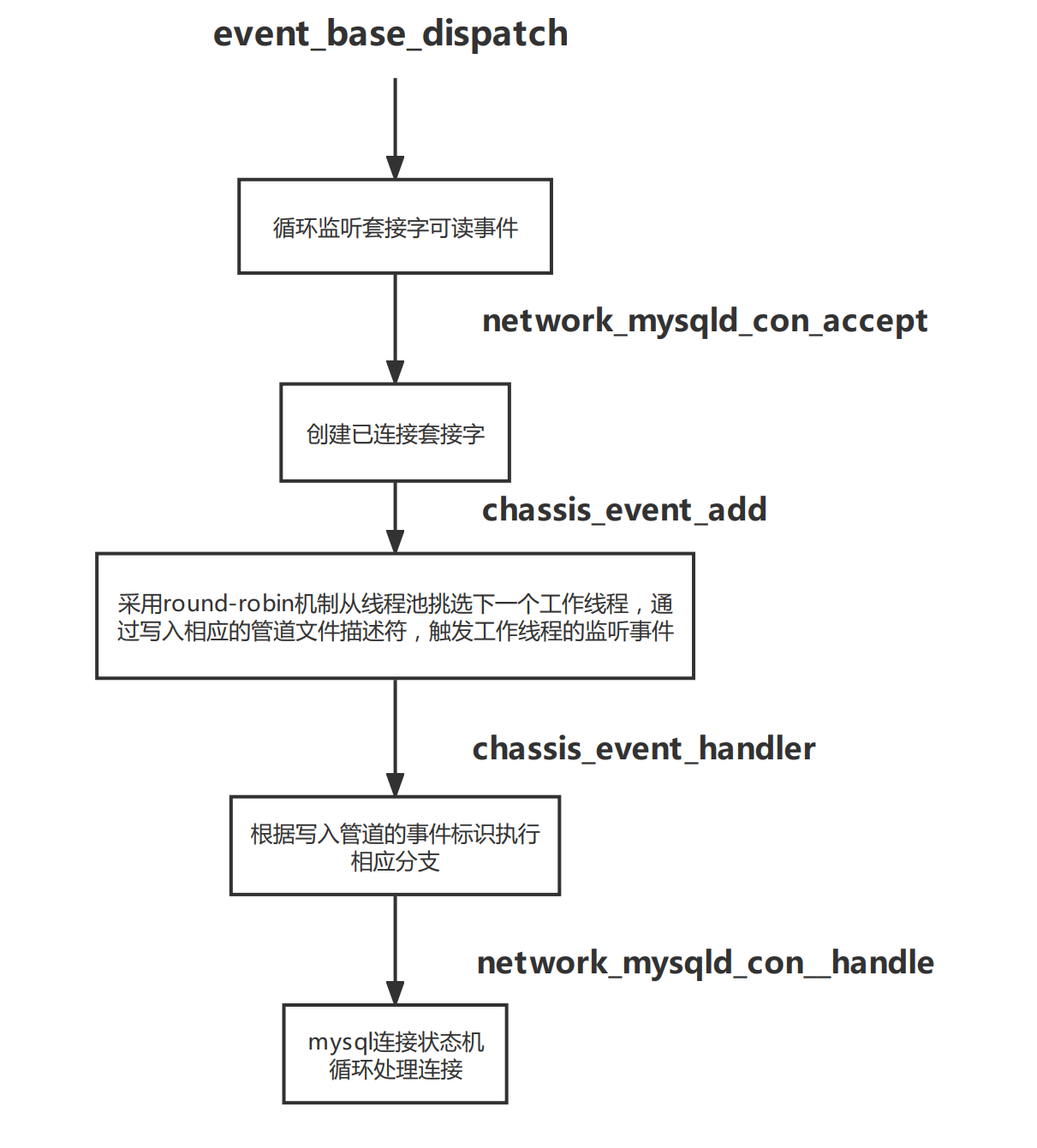
函数：

proto/client.cpp：解析server返回数据包

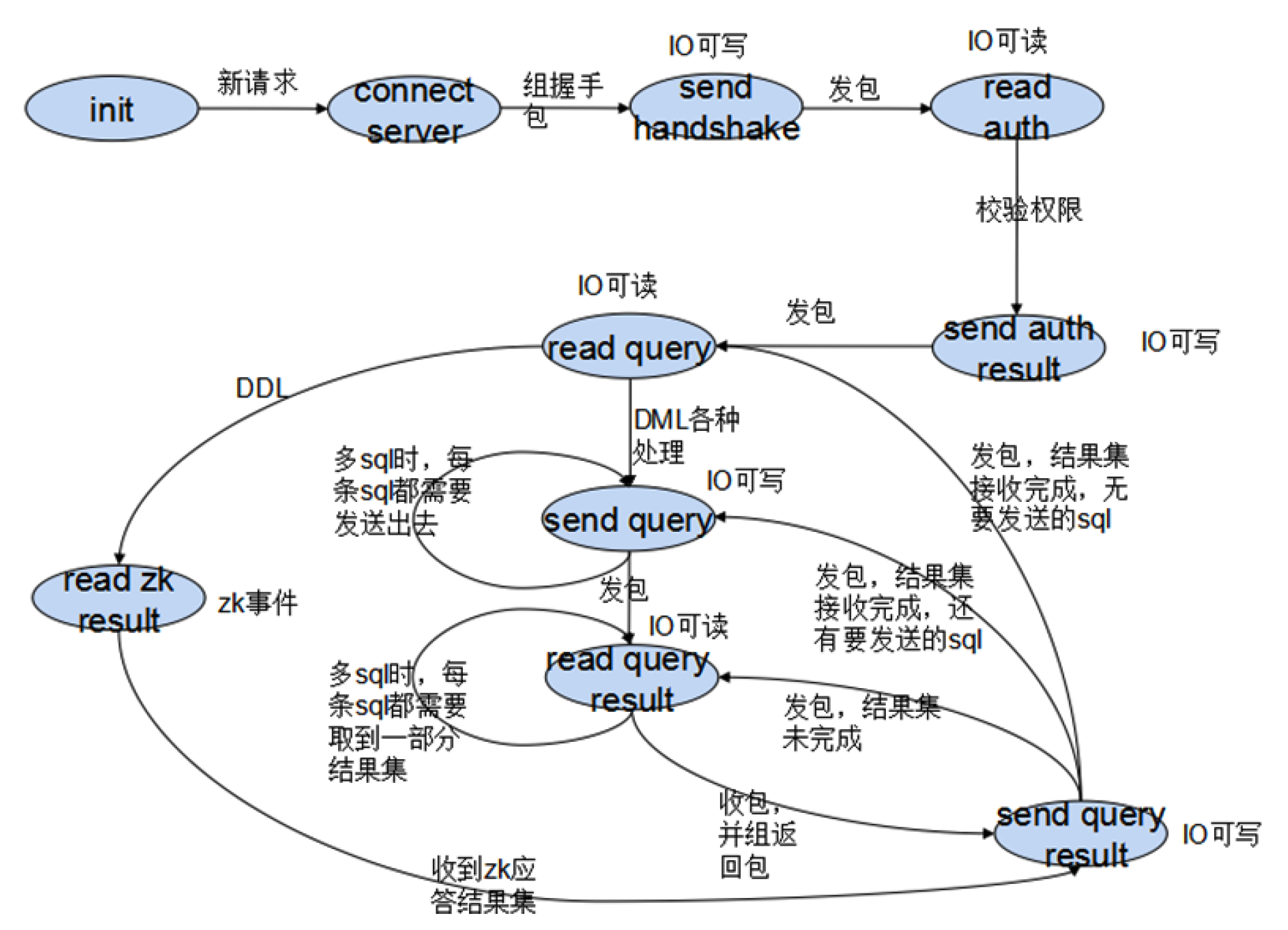
proto/proxy\_server.cpp：proxy发送给server的数据

proto/sql\_prepare.cpp：解析server返回的包

# MySQL网络状态机



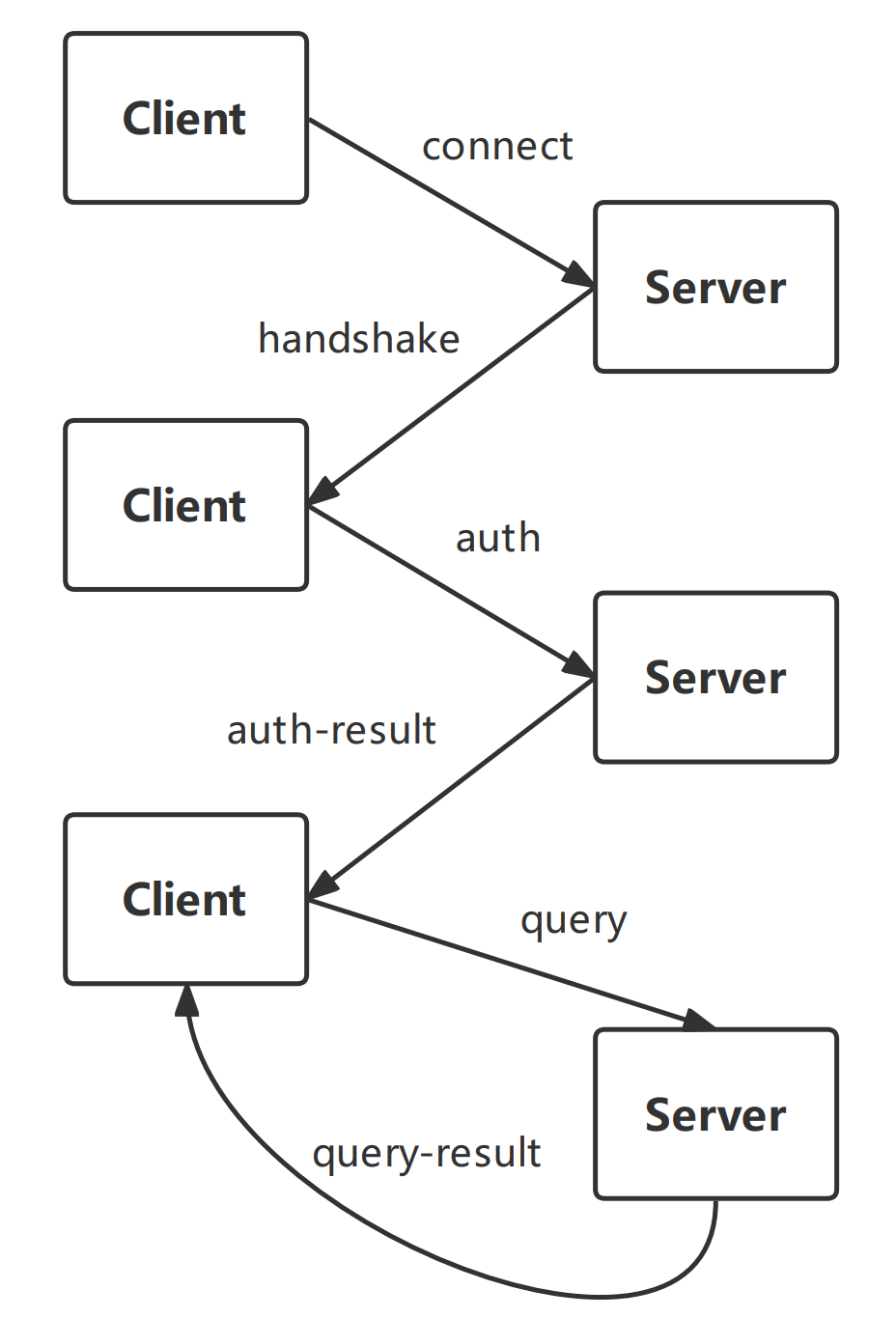
## 连接状态机



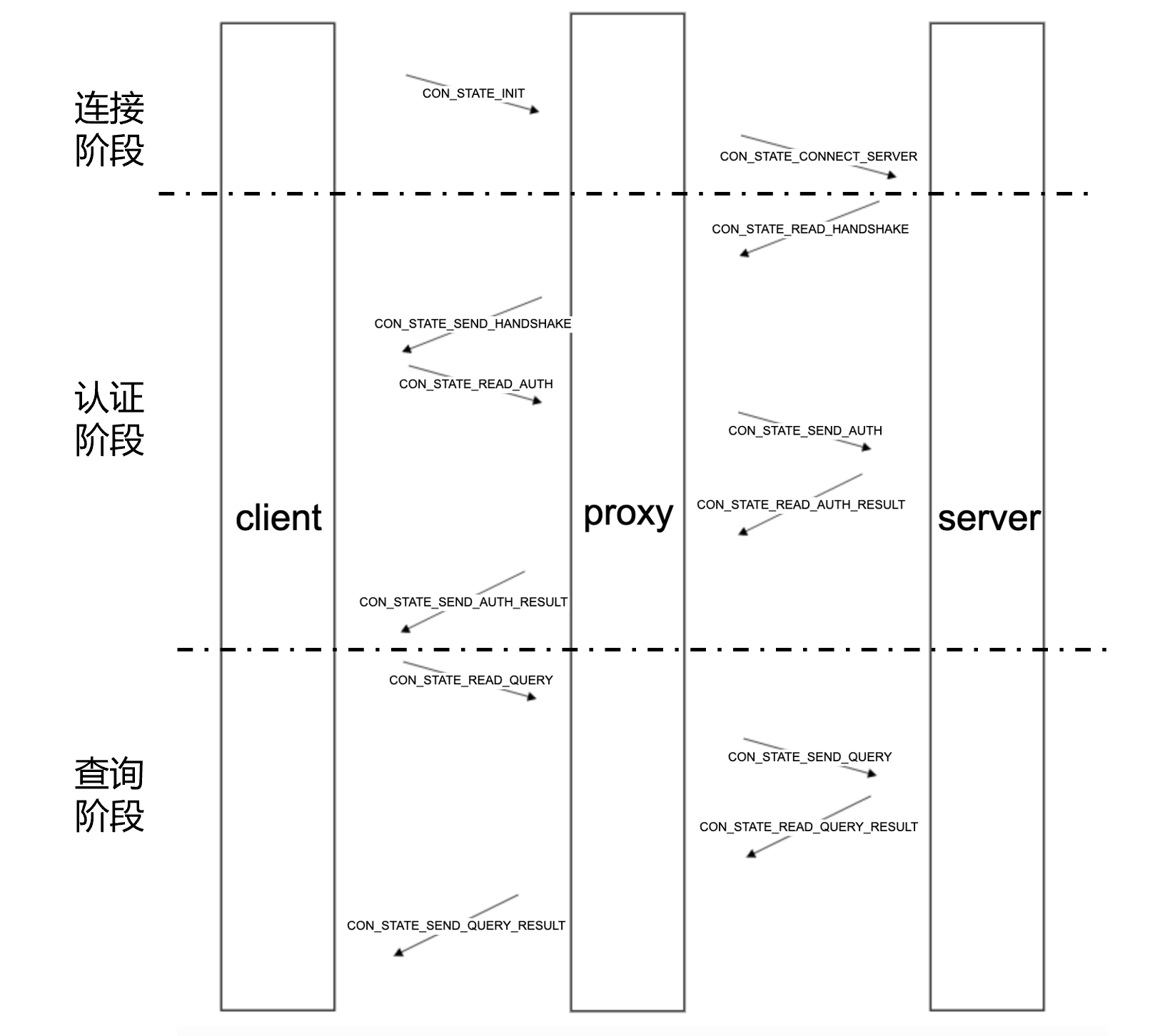
## C-S交互图

Client-Server交互大致流程：

1. 连接阶段
2. 认证阶段
3. 查询阶段

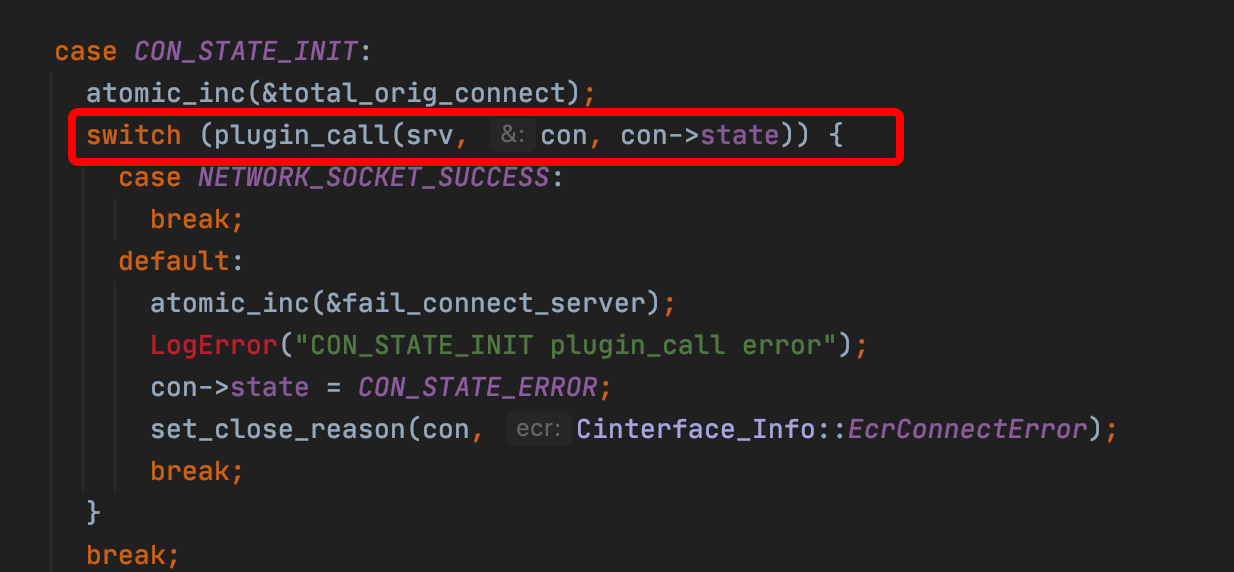


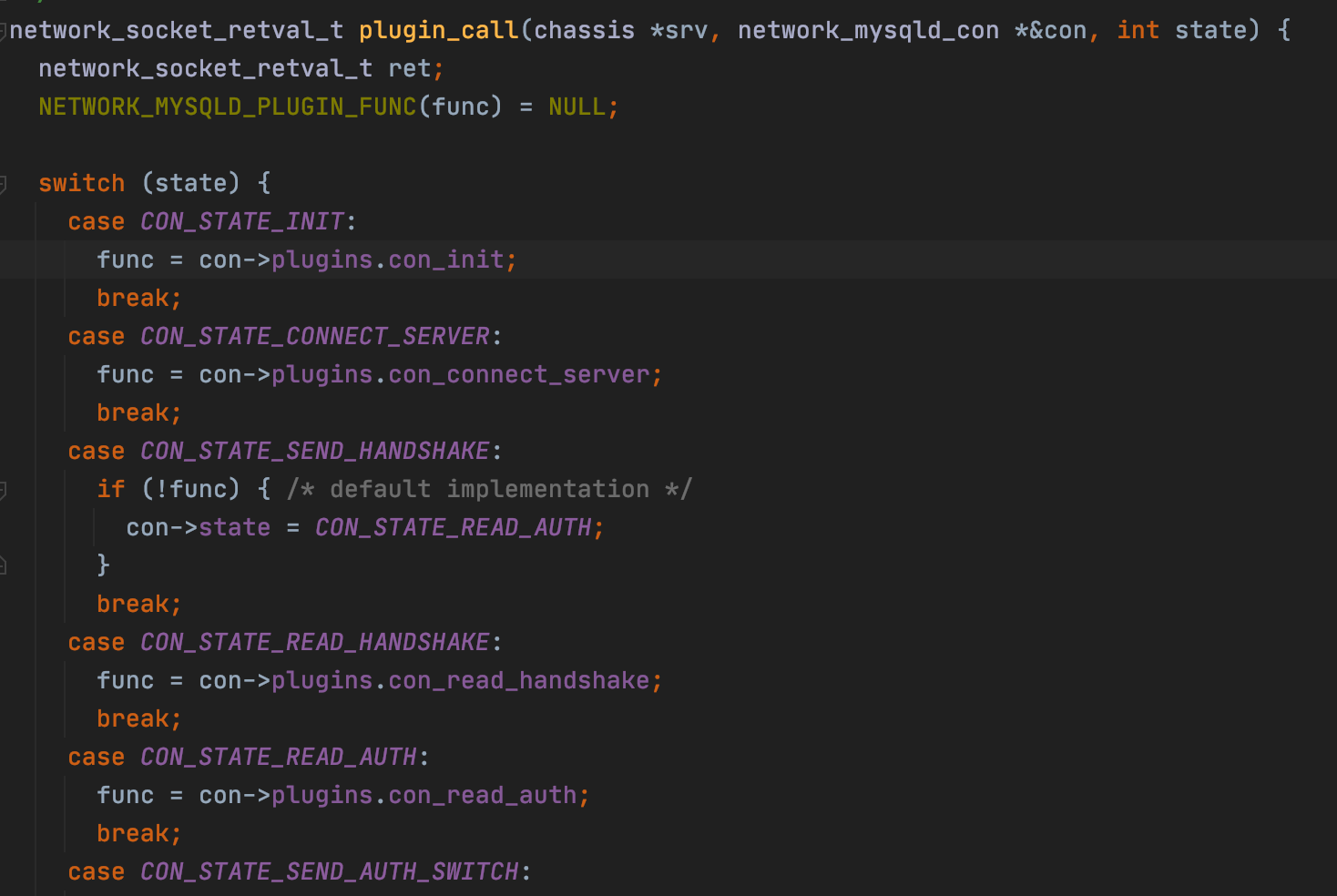
具体阶段以及状态：



## 插件函数

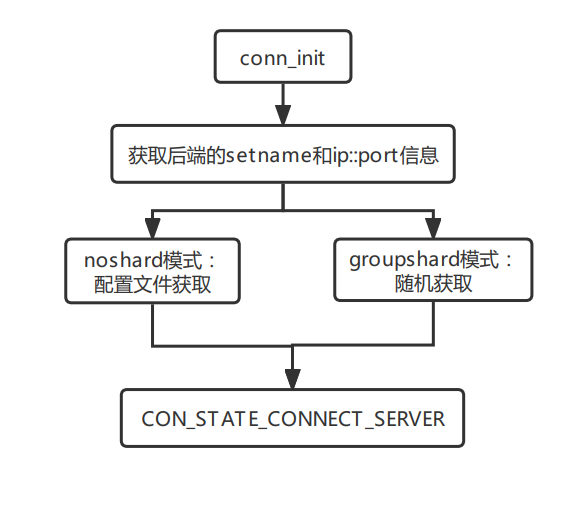
状态循环中的插件函数均通过plugin\_call() 间接调用，在主线程加载共享库时确定。





## 连接阶段

### CON\_STATE\_INIT



所有新建立的客户端连接进入的第一个状态，调用plugin\_call ()插件加载函数，通过con\_init()初始化连接信息。

在conn\_init中主要是获取与后端DB建链所需要的setname和ip::port信息，这里需要根据配置（是否用备机做鉴权），从共享内存中（从zk同步过来）获取set状态以及随机选择主备机。

具体流程：

1. 调用get\_ipport获取连接信息（set，ip，port）：
2. 读取配置项gateway.mode.slave.connect判断是否使用备机做用户鉴权，设置connect\_slave=true；
3. 读取配置项判断实例类型：如果是非分布式实例，直接从配置文件获取set和ip信息，并且存储到共享内存中
4. 如果是分布式实例：

3.1）读取共享内存中的信息，判断group的状态，如果状态异常则获取配置文件默认set，返回；

3.2）如果是初始化状态（即con\_init流程）且配置项中设置允许使用备机做用户鉴权（connect\_slave==true），则调用get\_slave\_random\_set从共享内存中获取随机备机set信息（减轻主机的负载压力）；

3.3）其他情况，获取随机主机set信息（get\_master\_random\_set这里初始化是随机连接一个DB，后面的连接set会变动）；

3.4）如果是初始化状态且配置用备机做用户鉴权，调用get\_one\_slave\_address获取对应set备机的ip:port连接信息，如果获取备机的ip::port信息失败，调用get\_set\_master\_address获取master节点的ip:port；

3.5）其他情况，调用get\_set\_master\_address获取master节点的ip:port；

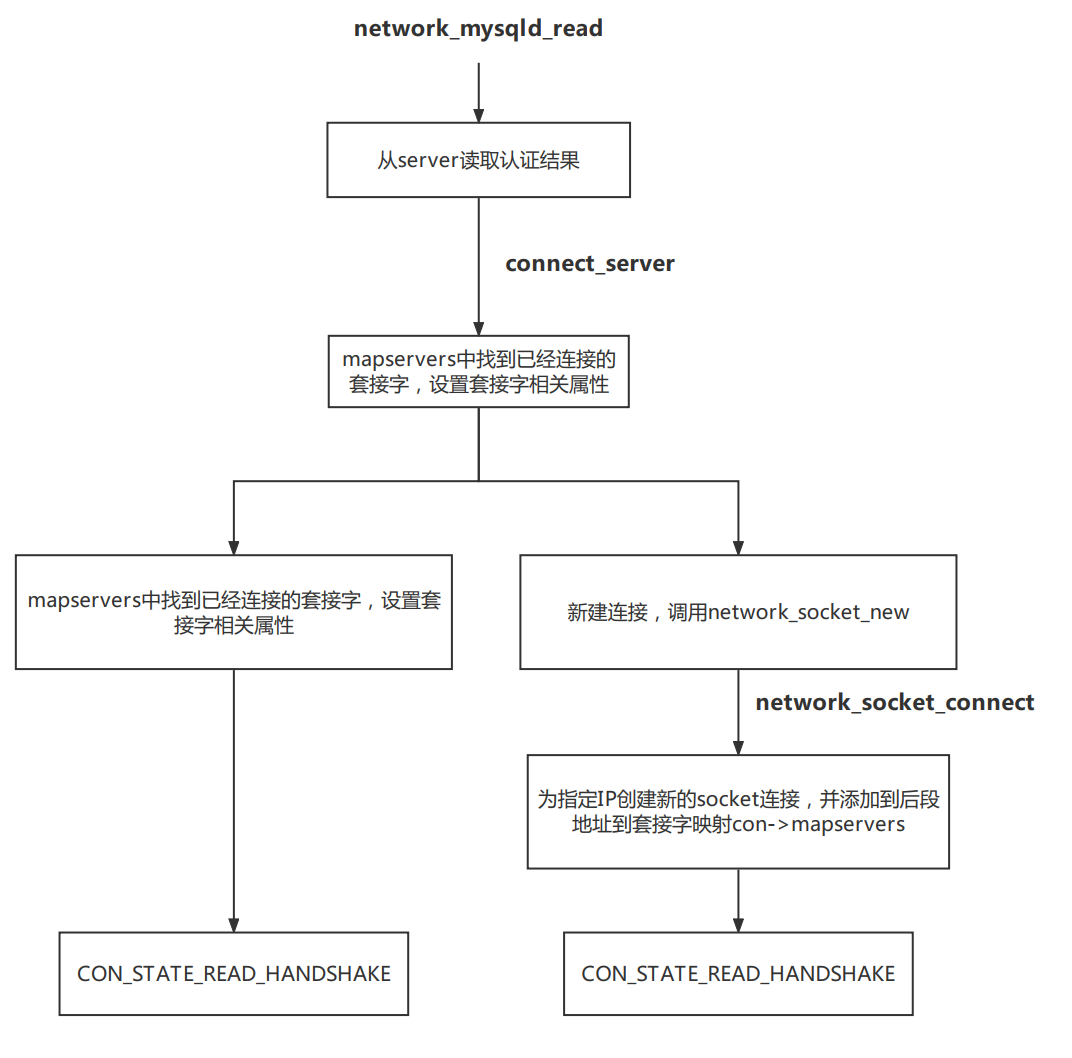
3.6）获取set信息和ip，返回pair<setname,ip\_port>。

1. 如果获取后端连接信息失败：

连接状态转移到NETWORK\_SOCKET\_ERROR；

1. 如果获取连接信息成功：
2. 更新当前connection的新建连接数m\_new\_connect\_num（初始化和与后端DB建立socket连接的时候都会递增，记录到日志中）；
3. 状态转移CON\_STATE\_CONNECT\_SERVER。

### CON\_STATE\_CONNECT\_SERVER



在调用连接初始化con\_init成功后，连接状态设置为CON\_STATE\_CONNECT\_SERVER，调用插件函数connect\_server()。这里是通过前面获取的setname和ip::port信息与后端DB建立socket连接，设置socket的基本属性，然后状态转移到CON\_STATE\_READ\_HANDSHAKE。

主要流程：

1、在连接的con->mapservers（ip:port->socket）中查找是否有当前连接地址con->address，如果未查到则表名是新连接：

该情况由CON\_STATE\_INIT跳转而来，基本流程：

1）调用network\_socket\_new创建新的socket；

2）将创建的socket添加进con->mapservers[con->address]，实现后端地址到套接字的映射；

3）调用network\_address\_set\_address设置地址（字符串->网络地址结构体）；

4）调用network\_socket\_connect()连接server（设置为非阻塞IO：O\_NONBLOCK，TCP属性：TCP\_NODELAY，TCP\_QUICKACK）：

4.1）如果连接状态为NETWORK\_SOCKET\_SUCCESS，则进入状态CON\_STATE\_READ\_HANDSHAKE；

4.2）若连接状态为NETWORK\_SOCKET\_ERROR\_RETRY，表示需要等待完成，则使用WATI\_FOR\_EVENT 添加监听套接字可写事件到当前工作线程，并返回退出。

2、如果不是新连接，则调用network\_socket\_connect\_finish判断连接状态：

1）成功，状态跳转为CON\_STATE\_READ\_HANDSHAKE；

2）失败，状态跳转为NETWORK\_SOCKET\_ERROR。

3、一切完成后，状态设置为CON\_STATE\_READ\_HANDSHAKE。

如果调用插件函数connect\_server()返回失败，表示对当前指定后端server建链失败，需要做如下处理：

1、如果插件函数socket状态为NETWORK\_SOCKET\_ERROR\_RETRY，则获取配置gateway.security.timeout.connect（连接DB超时时间），继续监听连接DB的事件，继续等待连接；

2、其他情况：

1）当con->server\_init为0时，连接状态设置为CON\_STATE\_ERROR，关闭连接；

2）当con->server\_init为1时，代表当前状态是由CON\_STATE\_GET\_SOCK转入，调用try\_another\_server()将当前建链失败的地址剔除，并与另外一个servere建链，连接状态设置为CON\_STATE\_GET\_SOCK。

**说明：**

根据con->server\_init不同的值（该字段默认为0，会在认证阶段成功结束后设置为1，表示该连接已完成身份校验并成功连接过后端。设置此字段的意义在于，当后续请求存在向新的主机发起连接的情况时，如读写分离时从备机读取，而当前连接还只存在和master的连接，那么和备机的连接过程应该对client透明，即握手和校验过程均在proxy中完成，client无法感知。因此该字段值对各个状态的行为均有影响）会产生不同的处理。当con->server\_init 为1时，代表当前状态由后续CON\_STATE\_GET\_SOCK转入。

### CON\_STATE\_CLOSE\_CLIENT

函数network\_mysqld\_con\_handle中设置，这里主要针对客户端没有数据发送的情况：

1. 如果客户端主动关闭连接，则关闭客户端，状态设置为CON\_STATE\_CLOSE\_CLIENT；
2. 如果客户端异常终止，则关闭server，状态设置为CON\_STATE\_CLOSE\_SERVER；
3. 如果是server关闭，则调用network\_mysqld\_con\_handle\_connection\_broken：

1）server未知原因关闭，状态设置为CON\_STATE\_ERROR；

2）server鉴权失败，状态设置为CON\_STATE\_CLOSE\_SERVER；

3）上报故障的连接

空闲server，忽略继续

人为关闭server，状态设置为CON\_STATE\_READ\_QUERY

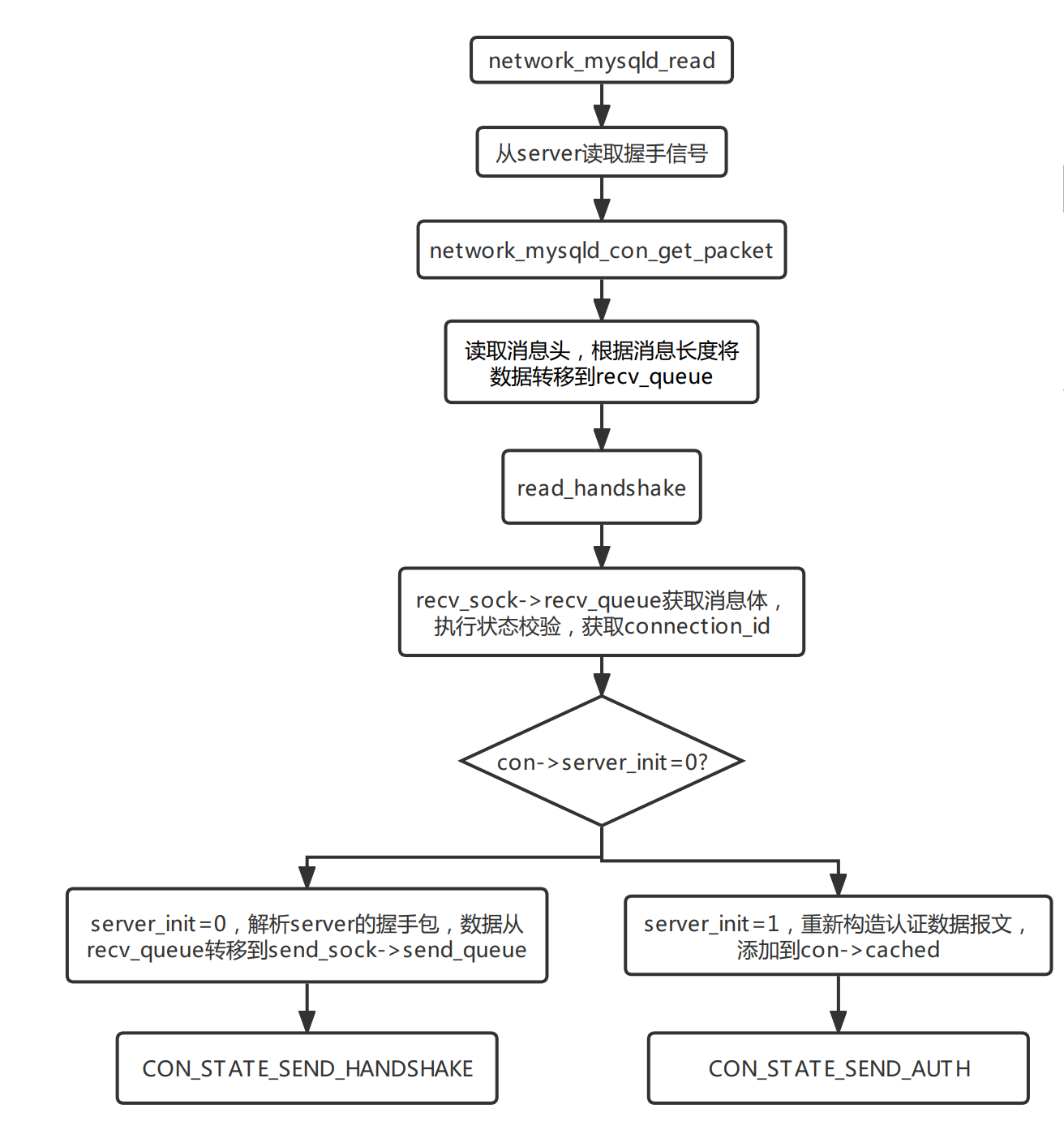
事务影响，状态设置为CON\_STATE\_ERROR

### CON\_STATE\_CLOSE\_SERVER

## 认证

从这个阶段起，后续认证相关的状态主要是对网络包进行解析。

### CON\_STATE\_READ\_HANDSHAKE



该状态从CON\_STATE\_CONNECT\_SERVER跳转过来，即与server完成连接之后，首先调用network\_mysqld\_read()读取server返回的握手信号，接着调用read\_handshake()处理握手信息，完成后状态转移到CON\_STATE\_SEND\_HANDSHAKE（发送给客户端握手信号），CON\_STATE\_SEND\_AUTH（中间状态认证使用）。

说明：这里只是建立socket的握手信号，不是执行鉴权，在完成socket建链后才执行鉴权操作，即状态CON\_STATE\_READ\_AUTH。

首先调用network\_mysqld\_read()读取网络包。该函数也会多次在后续状态被调用用于从任一端读取数据。

基本流程：

1. 调用network\_socket\_read\_optimize()，其中recv()将原生数据读入sock->recv\_queue\_raw。
2. 如果返回状态错误：

E\_NET\_CONNABORTED、E\_NET\_CONNRESET、E\_NET\_WOULDBLOCK、EAGAIN，则设置状态为NETWORK\_SOCKET\_WAIT\_FOR\_EVENT

1. 如果是对方关闭socket类型，则设置状态为NETWORK\_SOCKET\_WAIT\_FOR\_EVENT

这个过程需要等待，NETWORK\_SOCKET\_WAIT\_FOR\_EVENT会注册监听可读事件然后函数退出。

1. 数据读入完成后，调用network\_mysqld\_con\_get\_packet()，通过network\_queue\_get\_header()读取消息头，根据报文长度将数据从recv\_queue\_raw转移到recv\_queue并检查包序号，判断是否有序接收。

判断network\_mysqld\_read()的返回状态：

1. NETWORK\_SOCKET\_SUCCESS，成功
2. NETWORK\_SOCKET\_WAIT\_FOR\_EVENT，获取超时时间并等待事件；
3. NETWORK\_SOCKET\_ERROR\_RETRY/NETWORK\_SOCKET\_ERROR，关闭连接。

之后调用插件函数read\_handshake()：

1. 调用g\_queue\_peek\_tail从recv\_sock->recv\_queue ->chunks获取封装好的数据包，并校验状态是否有错误；
2. 调用get\_connection\_id获取包头指针后NET\_HEADER\_SIZE处的con->server\_init，根据值不同分别处理：
3. server\_init == 0

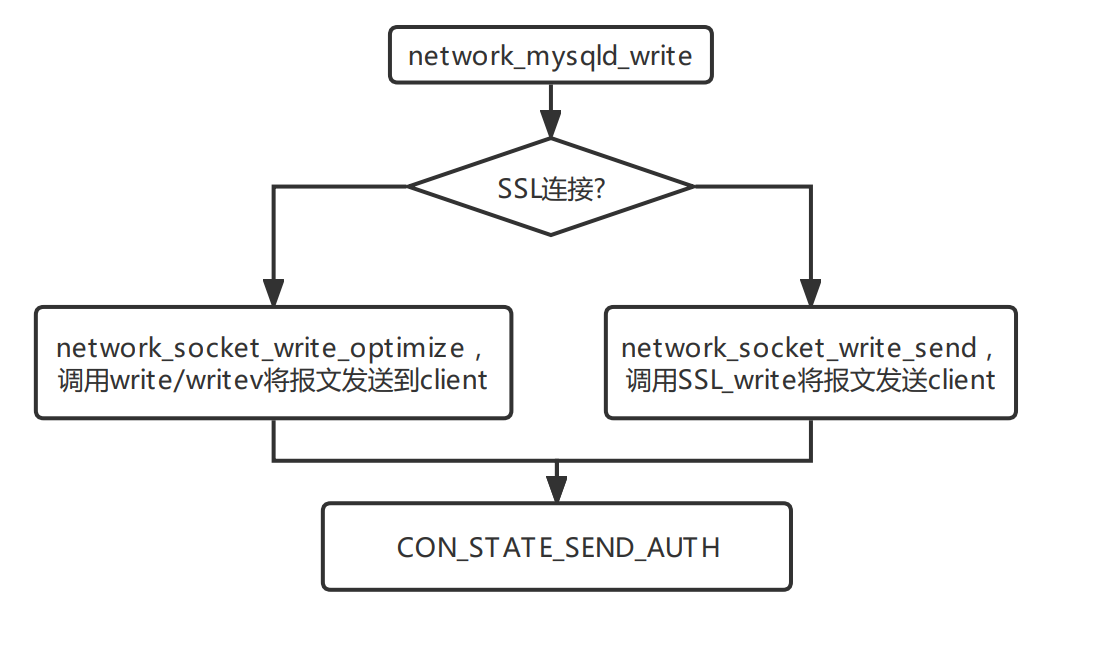
在首次连接的情况中，我们需要将server发来的握手包转发给client， parse\_server\_handshake\_packet ()函数就是用来解析包的内容并修改或保留相应字节。解析完成后，数据包从recv\_queue添加到send\_sock->send\_queue，为下一状态CON\_STATE\_SEND\_HANDSHAKE向client发送握手包做准备。

1. server\_init == 1

对于非首次连接的情况（其余状态转入），我们不应向client转发（对用户无感知）。根据连接中记录的信息，send\_client\_reply\_packet()组装出正确的客户端认证报文并添加进server套接字的发送队列，跳过CON\_STATE\_SEND\_HANDSHAKE和CON\_STATE\_READ\_AUTH两个与客户端交互的状态（同一个链路不需要再次执行重复动作），进入CON\_STATE\_SEND\_AUTH。

这个标志位server\_init意义在于，当后续请求向新的server发起建链请求的时候，比如开启读写分离时从备机读，而当前连接还只存在于主的连接，那么和备机的连接过程应该是对客户端无感知的，即这个时候proxy需要完成握手和校验等过程。

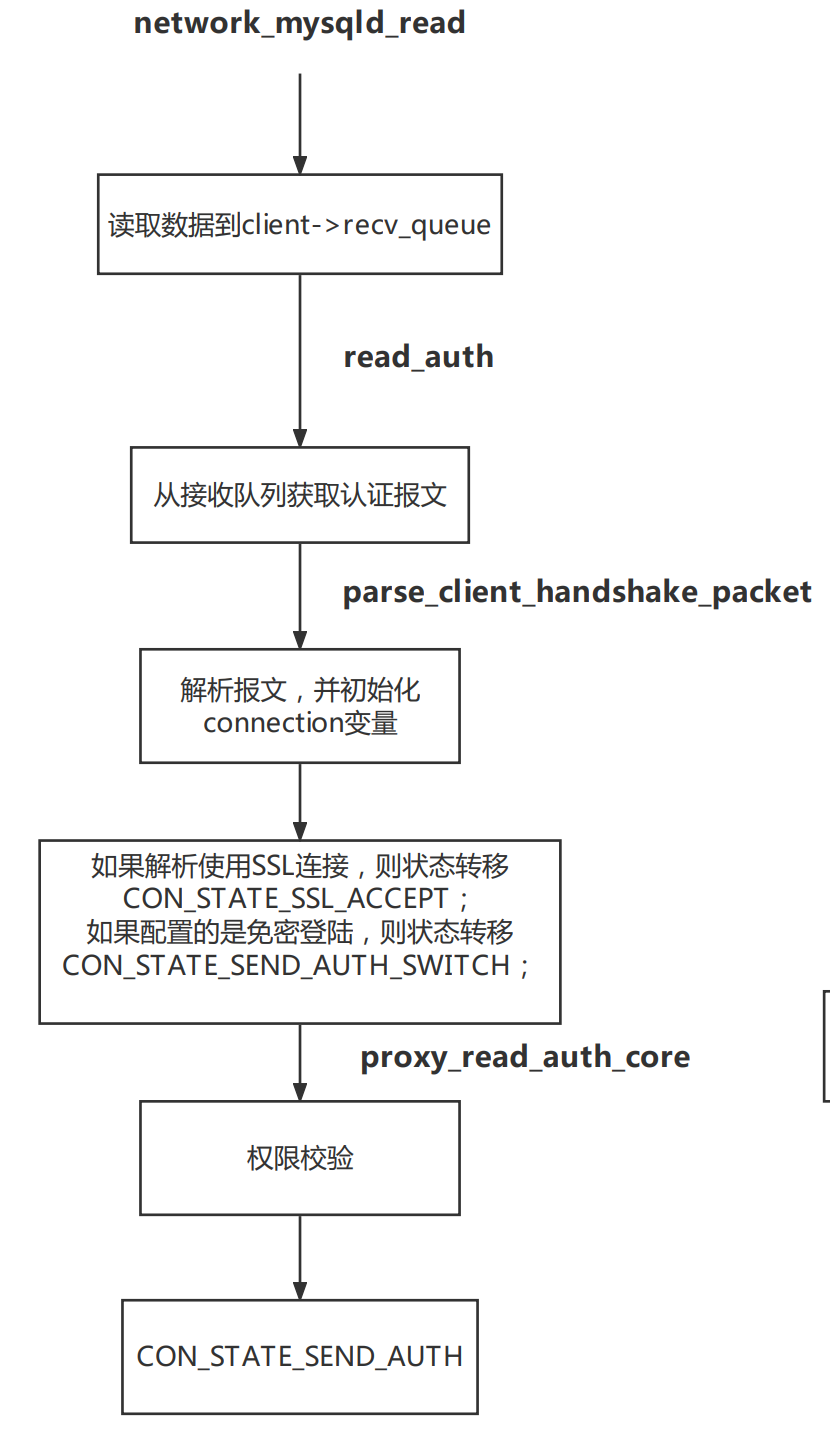
### CON\_STATE\_SEND\_HANDSHAKE



该状态从CON\_STATE\_SEND\_HANDSHAKE转移过来，首先调用network\_mysqld\_write ()将client套接字发送队列中在上个状态准备好的握手包发送给客户端（proxy->client），同样的，当写入需要等待，监听事件会被注册。

写入完成后，状态设置为CON\_STATE\_READ\_AUTH。

### CON\_STATE\_READ\_AUTH



该状态需要从客户端读取认证报文用于后续服务端认证（client->proxy），首先调用network\_mysqld\_read读取客户端的数据包，调用read\_auth接收认证报文并进行鉴权，完成后状态转移到CON\_STATE\_SEND\_AUTH。

注：前面的CON\_STATE\_READ\_HANDSHAKE是完成socket建链操作，为后面额鉴权提供连接，并不会参与鉴权。

基本流程：

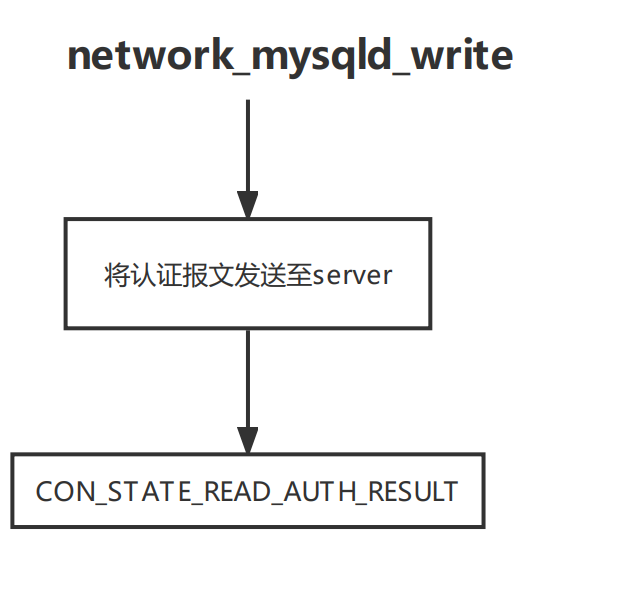
1. 首先调用network\_mysqld\_read()读取数据到client-> recv\_queue\_raw；
2. 调用插件函数read\_auth()从接受队列获取认证报文：
   1. 调用parse\_client\_handshake\_packet()解析报文内容（TLV格式解析=type[1]+length[3]+value），将用户名等信息写入连接结构体相关字段；

记录相关信息后调用proxy\_read\_auth\_core ()校验权限，包括IP限制，连接数上限，账户锁定和用户名密码检验等；

校验通过后，调用send\_client\_reply\_packet报文从con->response 添加到server套接字的发送队列；

* 1. 状态转移到CON\_STATE\_SEND\_AUTH。

### CON\_STATE\_SEND\_AUTH

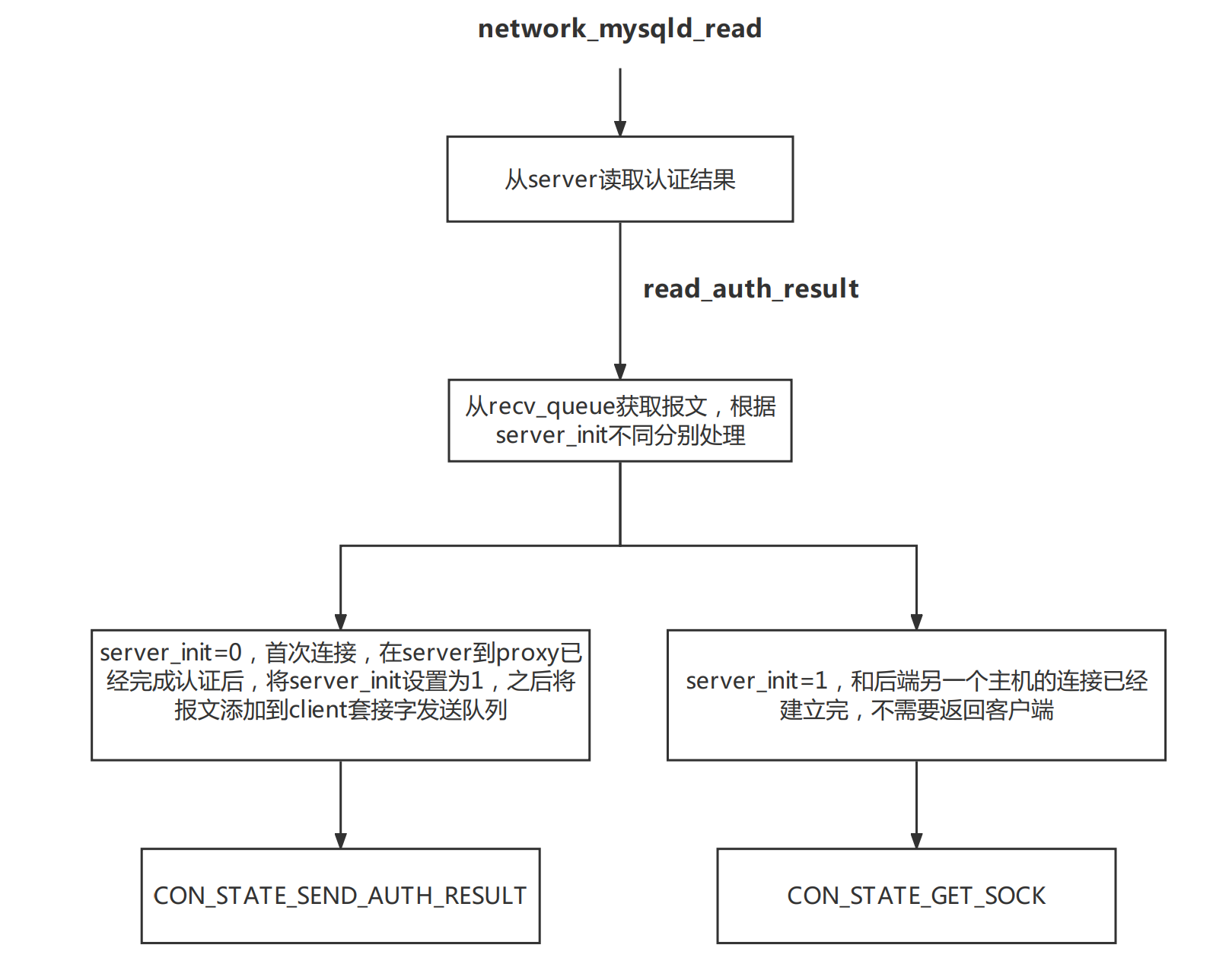


该状态是将之前client与proxy的认证响应发送给server（proxy->server）。

主要流程：

1. 调用network\_mysqld\_write()将认证报文发送到server；
2. 转移到CON\_STATE\_READ\_AUTH\_RESULT。

### CON\_STATE\_READ\_AUTH\_RESULT



此状态从CON\_STATE\_SEND\_AUTH转移过来，即完成鉴权发送后获取结果，主要是处理server返回的鉴权结果（server->proxy）。

主要流程：

1. 首先调用network\_mysqld\_read ()从server读取认证结果，如果失败同样调用try\_another\_server()尝试下一个地址；
2. 然后调用插件函数read\_auth\_result()。在插件函数中：
3. 首先从recv\_queue获取报文，之后根据server\_init做不同处理：
4. 根据server\_init值设置不同操作：

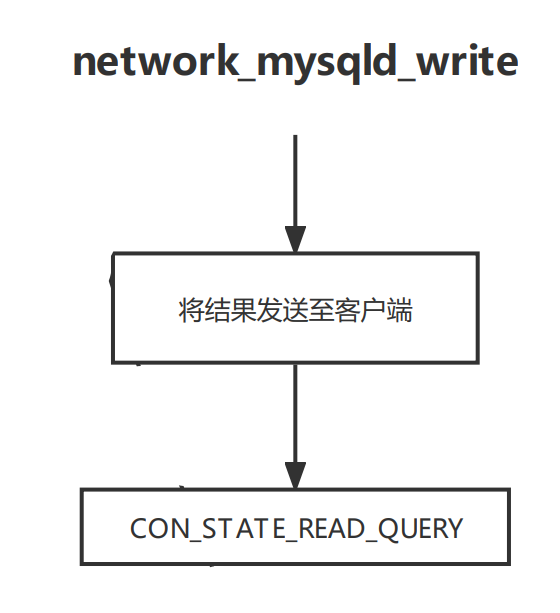
2.1）server\_init == 0

首次连接到此在server端和proxy内已经完成认证，将 server\_init 设置为1，之后将报文添加到client套接字的发送队列，转移到CON\_STATE\_SEND\_AUTH\_RESULT 状态，准备将结果发送到客户端。

2.2）server\_init == 1

和后端另一主机的连接已经建立完成，不需要返回客户端，状态转移到CON\_STATE\_GET\_SOCK。

### CON\_STATE\_SEND\_AUTH\_RESULT



该状态主要是将握手信号发送给客户端（proxy->client）。

主要流程：

1. 调用network\_mysqld\_write()将认证结果发送至客户端；
2. 状态转移到CON\_STATE\_READ\_QUERY（至此鉴权已经完成可以正常执行查询等操作）。

### CON\_STATE\_SEND\_AUTH\_SWITCH

在调用read\_auth()从客户端读取鉴权信息的时候，如果使用mysql\_native\_password，下一个状态转移CON\_STATE\_READ\_AUTH\_SWITCH\_RESULT。

### CON\_STATE\_READ\_AUTH\_SWITCH\_RESULT

### CON\_STATE\_SSL\_ACCEPT

在调用read\_auth()从客户端读取鉴权信息的时候，如果客户端使用SSL协议，则设置连接状态为CON\_STATE\_SSL\_ACCEPT。

调用插件函数con\_ssl\_accept()：

如果插件函数执行成功，状态转移CON\_STATE\_READ\_AUTH。

如果插件函数执行错误，状态转移CON\_STATE\_SEND\_ERROR。

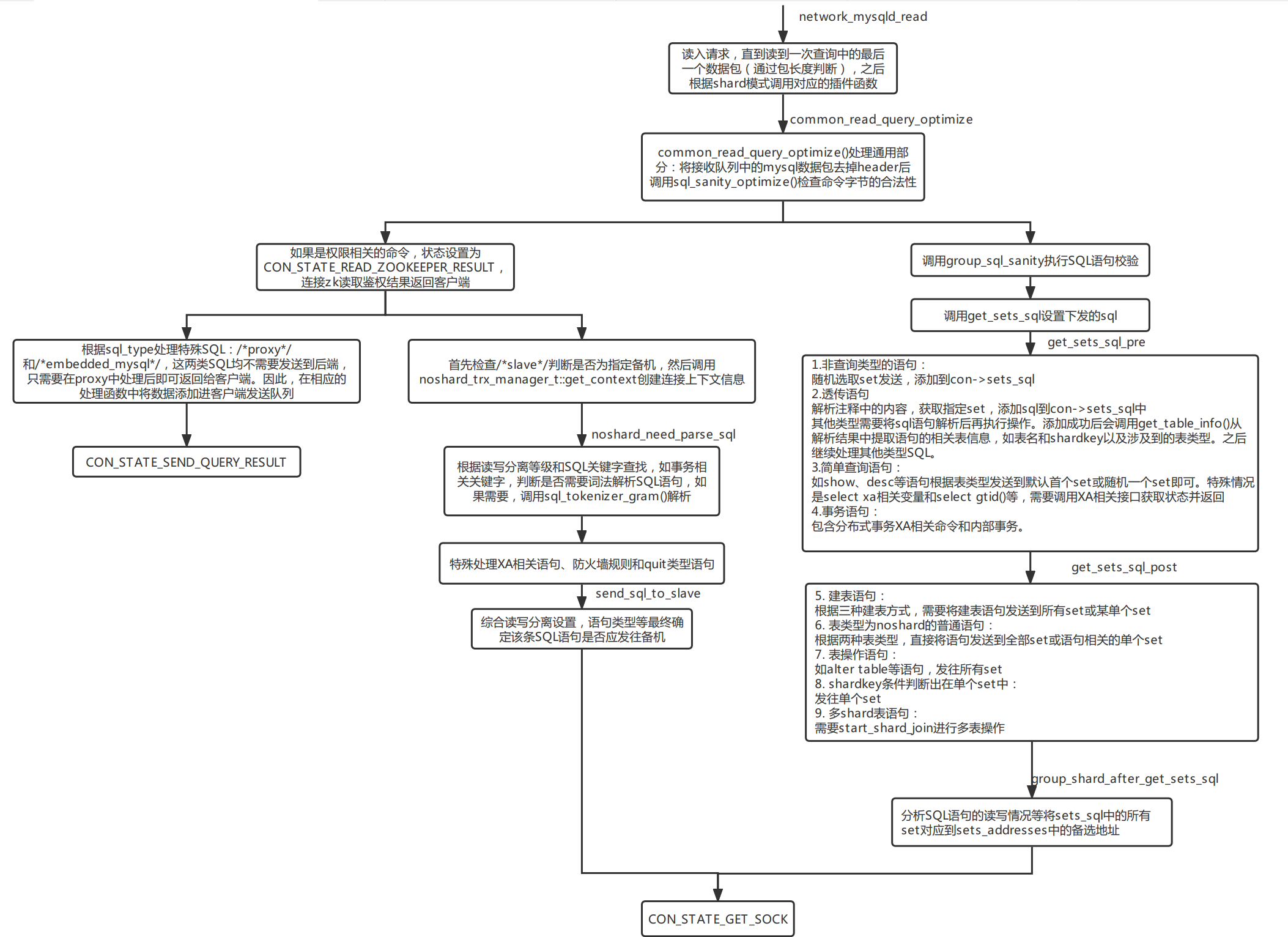
如果插件函数NETWORK\_SOCKET\_WAIT\_FOR\_EVENT，则继续等待。

## 查询

查询阶段处理连接建立后客户端查询和服务端查询结果的传递。处理过程中proxy可能会对数据包的内容做相应修改或拆分重组，尤其在group\_shard的模式下，部分类型的客户端命令可能会产生非常复杂的逻辑操作。

状态循环中的插件函数均通过plugin\_call()间接调用，查询阶段的各个状态对应的插件函数在no\_shard和group\_shard模式下会对应到不同的入口，在主线程加载共享库时确定。

### CON\_STATE\_READ\_QUERY



该状态负责在建立连接后从客户端读入命令。大多数连接都会等待（事件监听）在这个状态，直到有客户端输入。首先循环调network\_mysqld\_read ()读入请求，直到读到一次查询中的最后一个数据包（通过包长度判断）。之后根据shard模式调用相应的插件函数。

该状态负责在建立连接和认证后从客户端读入命令。大多数连接都会等待（事件监听）在这个状态，直到有客户端输入。

主要流程：

1. 循环调用network\_mysqld\_read ()读入请求；
2. 调用plugin\_call加载插件函数，根据shard模式（no shard/group shard）调用相应的插件函数：

2.1）no\_shard模式：no\_shard\_read\_query()

1）调用common\_read\_query\_optimize()处理通用部分：

a）将接收队列中的mysql数据包去掉header后添加进con->orig\_sql字段，调用sql\_sanity\_optimize检查命令字节的合法性

b）将接收sql数据包去掉NET\_HEADER\_SIZE后，调用reset\_sql执行语法解析；

c）如果是用户鉴权相关的命令，则状态设置为CON\_STATE\_READ\_ZOOKEEPER\_RESULT，调用deal\_priv\_command处理（发送zk->zk返回鉴权结果->返回客户端）；

d）根据con->sql\_type的不同处理两种类型SQL：

如果是/\*proxy\*/，调用deal\_proxy\_command。

如果是/\*embedded\_mysql\*/，调用do\_embedded\_mysql\_command。

这两类语句均不需要发送到后端，只需要在proxy中处理后即可返回给客户端。因此，在相应的处理函数中将数据添加进客户端发送队列后转移到CON\_STATE\_SEND\_QUERY\_RESULT。

2）检查con->sql\_type 是否为PROXY\_SQL\_SLAVE ，即是否/\*slave\*/指定备机，然后调用noshard\_trx\_manager\_t::get\_context ()创建连接上下文信息。

3）no shard模式下在如下几种情况会调用sql\_tokenizer\_gram()执行语法解析（sql\_yacc\_gw.yy，即proxy自定义的解析规则）：

a）开启读写分离（rw\_split==2），需要检查是read/write操作

b）set sql，需要获取变量和值

c）事务控制语句begin/commit，需要检查是否在事务中

d）开启防火墙，需要检查规则文件并执行过滤

4）如果开启防火墙过滤，则调用route\_query按照规则进行过滤；

5）如果当前连接的server关闭（即con->server\_closed==true），则需要结束事务，如果是XA事务，发送XA ROLLBACK xid到后端DB；

6）调用noshard\_trx\_manager\_t::send\_sql\_to\_slave()判断该条sql语句是否应发往备机；

7）调用get\_noshard\_setname获取set名称，存储到 con->setname；

8）调用get\_set\_address获取默认set在此次查询中应发送的实例地址，存储到con->sets\_address；

9）将con->address存储到con->current\_socks中，同时状态转移到CON\_STATE\_GET\_SOCK。

2.2）group\_shard模式：group\_shard\_read\_query()

1）调用increase\_query\_sequence递增查询序列号；

2）调用get\_group\_status检查group的状态；

3）调用common\_read\_query\_optimize：

a）调用sql\_sanity\_optimize判断合法性；

b）获取包的原始语句，调用reset\_sql执行解析；

c）根据con->sql\_type的不同处理两种类型SQL：

如果是/\*proxy\*/，调用deal\_proxy\_command。

如果是/\*embedded\_mysql\*/，调用do\_embedded\_mysql\_command。

4）将接收队列中的mysql数据包去掉header后添加进con->orig\_sql 字段；

5）如果是slave hint，则调用shard\_trx\_manager\_t::get\_context申请上下文；

6）处理XA，COM\_QUIT等无需发送后端DB的操作，成功返回；

7）调用get\_sets\_sql() 处理其他需要发送到后端的语句，该函数主要作用就是分析SQL，设置发送到每个set的具体语句（存储在con->sets\_sql），主要包括两个函数：

7.1）、get\_sets\_sql\_pre()预处理SQL：

1. 如果是prepare语句，调用DealPrepareStatementSql处理prepare语句；
2. 非查询类型的语句（sql[0] != COM\_QUERY），调用deal\_not\_comm\_query\_type\_sql：随机主set发送，下发的SQL添加到con->sets\_sql中
3. 调用deal\_sql\_route\_comment处理路由hint和注释，为透传SQL设置set信息；
4. 如果sql\_command== SQLCOM\_PREPARE，调用save\_prepare保存相关信息，当sql\_command== SQLCOM\_EXECUT的时候，调用combine\_execute\_sql根据前面保存的prepare信息处理，最后调用get\_sets\_sql获取具体发送set的sql；
5. 调用get\_table\_info从zk获取表的信息；
6. 处理create/delete sequence相关操作；
7. 处理kill connection操作；
8. 处理show processlist操作；
9. 处理视图相关的操作，将视图创建语句发给第一个set；
10. 调用query\_simple\_sql判断处理简单查询语句，包括xa\_deal\_simple\_select处理select gtid()事务相关的SQL；
11. 如果是事务相关的SQL，调用deal\_transaction\_sql

包括分布式事务xa相关命令和内部事务；

l）如果是DDL，通过handle\_ddl\_after\_commit控制事务提交，调用do\_after\_commit\_or\_rollback控制DDL的回滚。

7.2）、get\_sets\_sql\_post()处理发送的SQL：

a）如果是查询系统表information\_schema，发送第一个set，其余系统表，随机选择一个set发送；

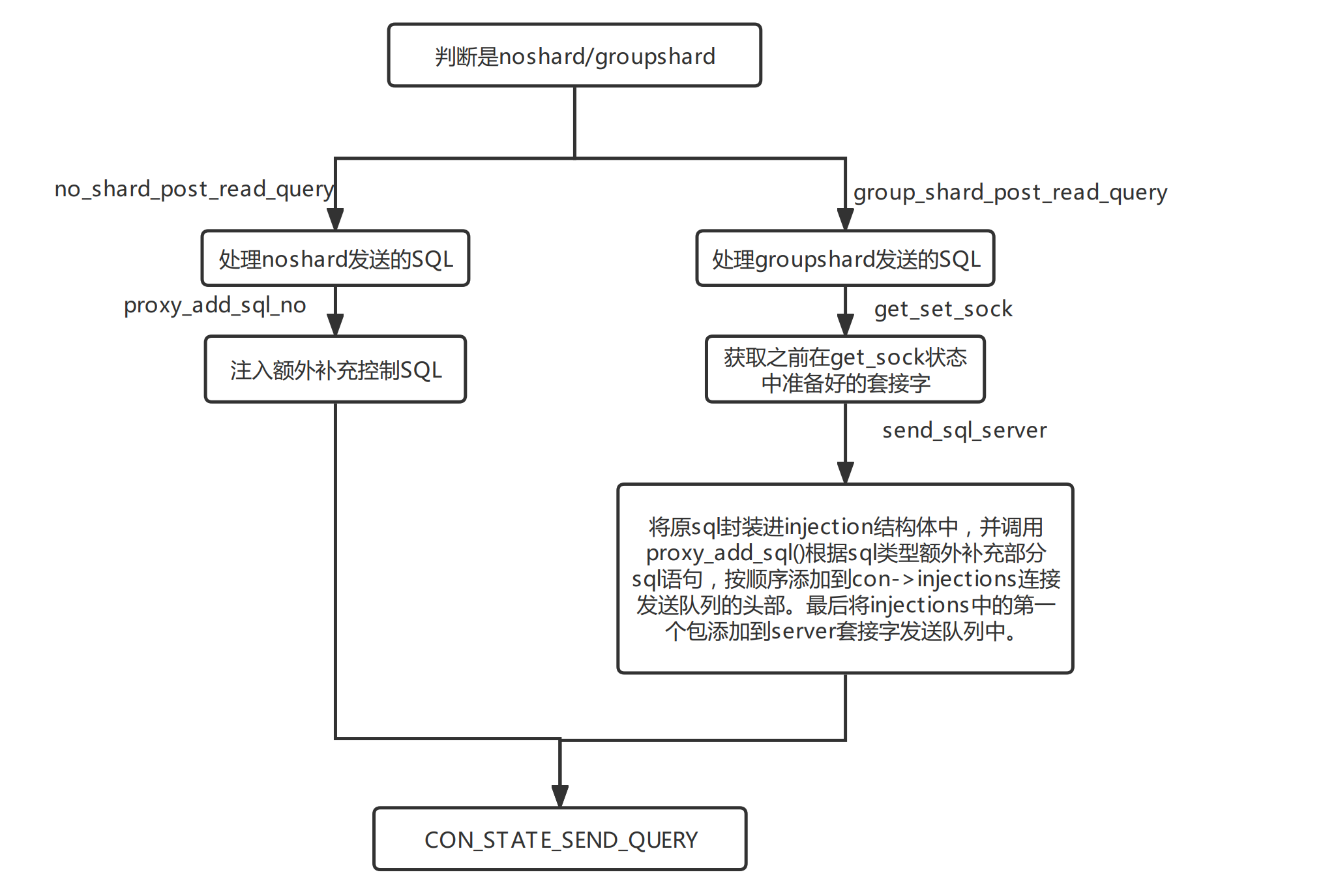
b）如果是create table，调用deal\_create\_table处理分区表、全局分区表、二级分区表、普通表、广播表；

c）如果是no shard相关语句（单表或广播表），调用deal\_noshard：不支持DDL，随机选择set；

d）如果是全局分区表和二级分区表的其他ddl语句，调用deal\_table\_query处理；  
e）如果是其他简单SQL，调用new\_sql\_parse构造发送到set的SQL；

f）如果是复杂语句，调用start\_shard\_join走嵌入式。

### CON\_STATE\_POST\_READ\_QUERY



在no\_shard\_get\_sock、no\_shard\_read\_query、HandleGtsMail中会设置CON\_STATE\_POST\_READ\_QUERY，调用插件函数no\_shard\_post\_read\_query/ group\_shard\_post\_read\_query处理就绪队列中的数据（proxy->server），并且在发射队列injections中增加一些额外的控制SQL（比如事务、切换用户等）。

主要流程：

获取套接字后，调用plugin\_call 加载插件函数，根据不同类型加载不同函数：

1、no\_shard：no\_shard\_post\_read\_query()

在send\_sql\_noshard中处理，基本流程：

1.1）解析包，调用parse\_com\_change\_user\_packet处理change user；

1.2）调用proxy\_add\_sql\_no()根据SQL类型注入SQL语句：如use db，set character，flow control，send begin，按顺序添加到con-> injections\_noshard，状态转移到CON\_STATE\_SEND\_QUERY。

2、group shard：group\_shard\_post\_read\_query()

2.1）判断是否为写语句，且涉及多个后端的set，是否为全局一致性读，如果是则设置is\_multi\_set\_modify；

2.2）调用xa\_start\_transaction\_if\_necessary()判断是否需要开启事务；

2.3）是否需要GTS，如果需要调用RequestMetaClusterGetGts向MC申请；

2.4）调用is\_sql\_allowed\_in\_transaction处理异常场景：server关闭，事务非活跃状态异常

2.5）如果是单条SQL，不需要改变结果：

a）当前连接不在事务中，调用send\_single\_set处理：

调用get\_set\_sock从链路缓存的连接中查找是否存在已建立的socket；

调用send\_sql\_server，在后端注入SQL（事务控制、用户切换等），状态设置为CON\_STATE\_SEND\_QUERY。

b）否则，调用deal\_multi\_simple（需要处理多个节点的分布式事务）：

增加后端注入SQL，设置状态为CON\_STATE\_SEND\_QUERY。

c）返回结果

2.6）检查SQL（事务控制相关）：

a）单条语句，调用check\_single\_set\_sql

b）否则，调用check\_multi\_sets\_sql

2.7）处理结果

a）如果con->flexible\_sel==true（拆分多个查询），调用post\_query，该函数会绑定高级功能CAdvance\_join的接口

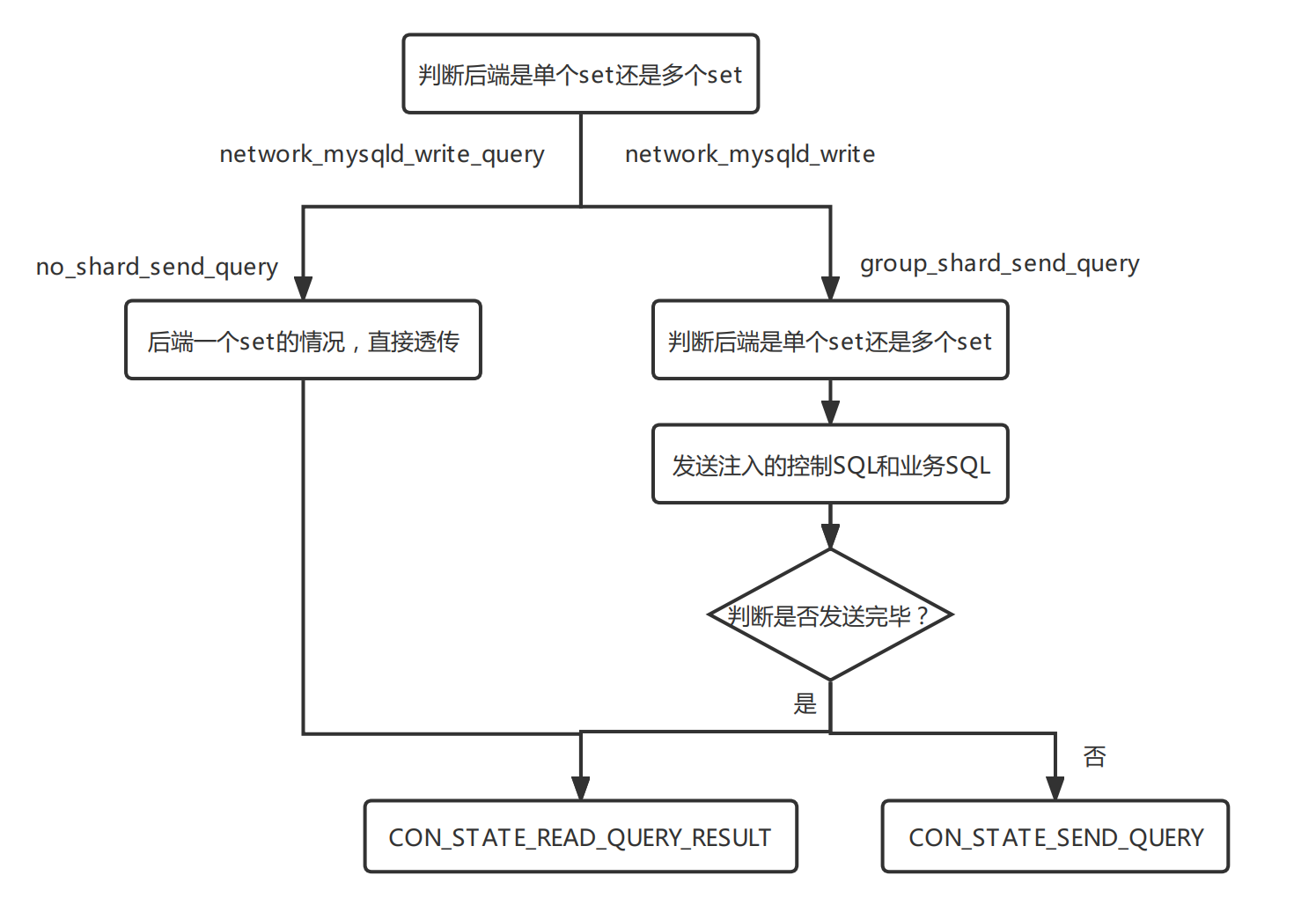
b）如果是select、需要下发多节点的explain或者需要修改结果集的SQL，调用高级功能接口deal\_advance\_select：

如果是简单SQL且仅1个SQL，则并发执行语句；

否则，注入后端控制SQL，状态设置为CON\_STATE\_SEND\_QUERY。

c）如果是其他SQL（insert/update/delete/replace/create），调用deal\_multi\_simple

### CON\_STATE\_SEND\_QUERY



该状态主要表示将发送队列的数据发送到后端（proxy->server），前面CON\_STATE\_POST\_READ\_QUERY中会注入额外的控制SQL，这里会取出来设置下发的set和对应的sql。

基本流程：

1. 不断循环直至所有来自发送队列send-queue的数据包全部flush完毕；
2. 如果是单节点，调用优化函数network\_mysqld\_write\_query
3. 如果非单节点，调用network\_mysqld\_write()将发送队列的数据发送到后端。

1）no\_shard：no\_shard\_send\_query()

不做任何处理，状态转移CON\_STATE\_READ\_QUERY\_RESULT。

2）group shard：group\_shard\_send\_query()

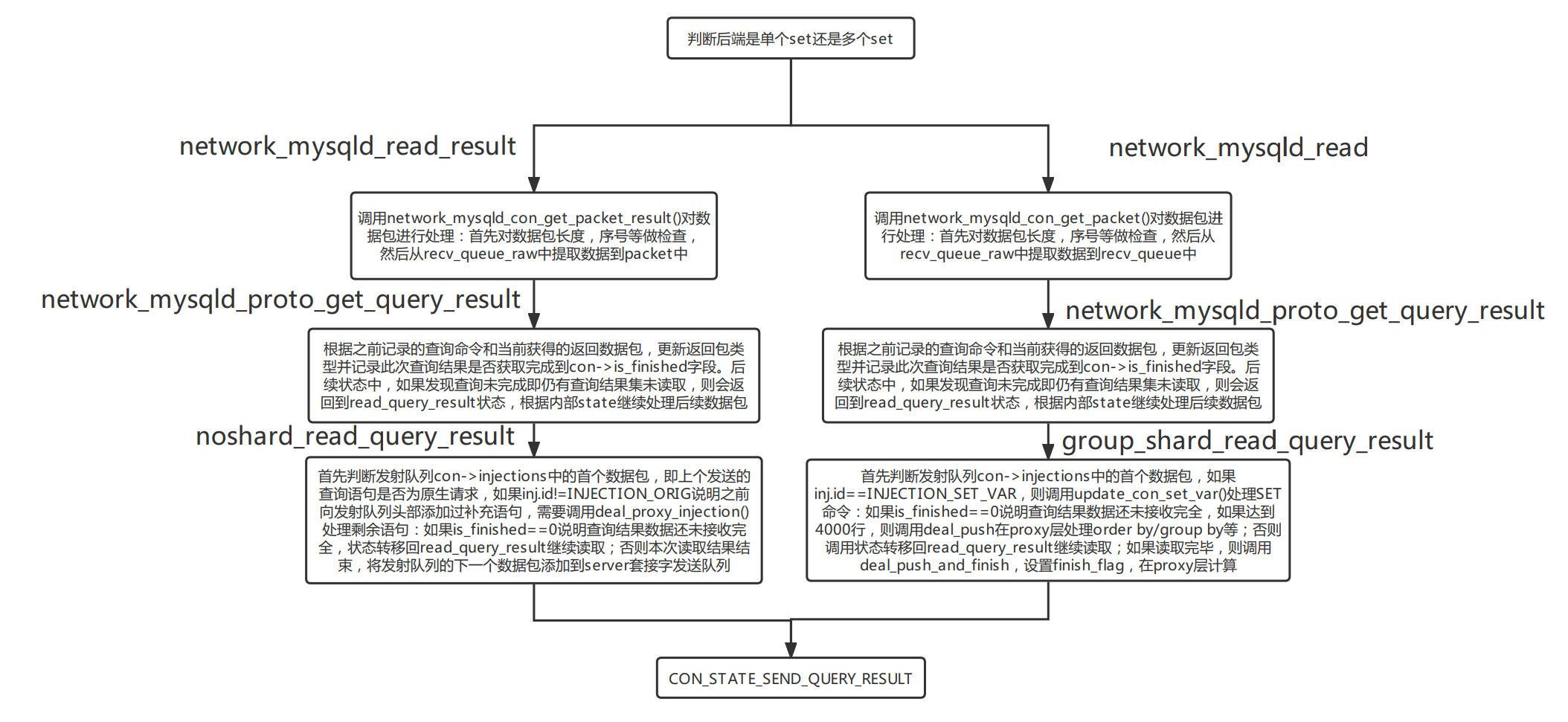
2.1）对于单connection，并行将注入的控制SQL和业务SQL发送到后端DB；

2.2）对于复杂查询，判断m\_backend\_index是否完成所有后端DB的SQL发送：

a）如果后端DB还未完全发送完毕，则替换后端连接为con->server，调用network\_mysqld\_queue\_append将injection查询追加到发送队列send\_sock->send\_queue，发送新的SQL，状态设置为CON\_STATE\_SEND\_QUERY；

b）如果已经发送完毕，则状态设置为CON\_STATE\_READ\_QUERY\_RESULT。

### CON\_STATE\_READ\_QUERY\_RESULT



Proxy读取server的结果集（proxy<-server）。

1. 根据con->state状态是否变化循环读取结果；
2. 如果是no shard，调用network\_mysqld\_read\_result()：

network\_mysqld\_read\_result ()从服务端读取数据。在该函数中，调用recv函数实现消息接收，然后会调用network\_mysqld\_con\_get\_packet\_result()对数据包进行处理：

2.1）对数据包长度，序号等做检查，如果错误则返回NETWORK\_SOCKET\_WAIT\_FOR\_EVENT；

2.2）调用network\_queue\_peek\_shift\_string从recv\_queue\_raw中提取数据到packet中；

2.3）调用network\_mysqld\_proto\_get\_query\_result()根据之前记录的查询命令和当前获得的返回数据包，更新返回包类型并记录此次查询结果是否获取完成到con->is\_finished字段。

1. 如果是group shard，调用network\_mysqld\_read()：

与前面的network\_mysqld\_read\_result()区别在于结果处理上，no shard会一次获取后端的结果并返回client，这里需要多次获取存储到接收队列con->recv\_queue。

1. 数据读取完成后，调用plugin\_call加载插件函数

4.1）、no\_shard：no\_shard\_read\_query\_result()

a）首先判断发射队列con->injections中的首个数据包，即上个发送的查询语句是否为raw请求，如果inj.id!=INJECTION\_ORIG说明之前向injection\_queue头部添加过补充语句，需要调用deal\_proxy\_injection()处理剩余语句：

如果 is\_finished==0说明查询结果数据还未接收完全，状态转移回CON\_STATE\_READ\_QUERY\_RESULT，调用read\_query\_result继续读取；

否则本次读取结果结束，将发射队列的下一个数据包添加到server套接字发送队列，状态转移到CON\_STATE\_SEND\_QUERY 准备继续发送查询到服务端。

b）根据is\_finished字段做出后续处理。最后，状态都会转移到CON\_STATE\_SEND\_QUERY\_RESULT向客户端发送已准备好的数据；

c）根据server的事务状态（如SERVER\_STATUS\_IN\_TRANS），设置事务上下文trx\_context。

4.2）group shard：group\_shard\_read\_query\_result()

a）如果inj->id == INJECTION\_SET\_VAR，则调用update\_con\_set\_var设置连接的变量；

b）调用network\_mysqld\_proto\_get\_query\_result()获取结果集，并标识标志位con->backend\_resultset\_is\_finished；

c）如果取结果完成且在事务中，会调用report\_server\_data\_changed()标识状态改变；

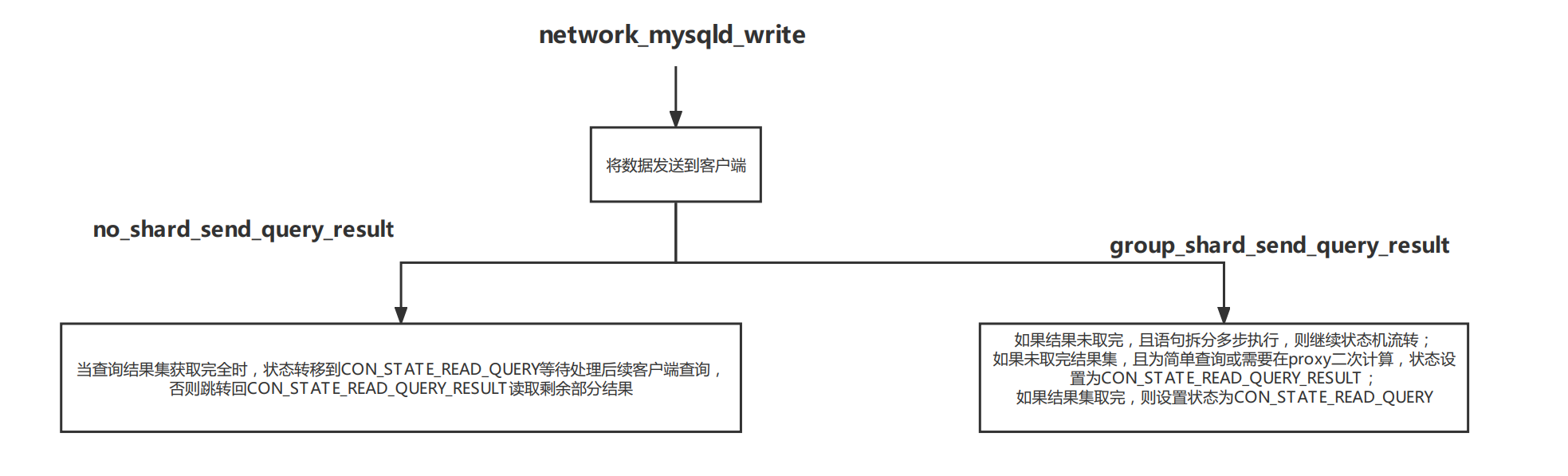
d）如果inj->id < INJECTION\_S\_CON\_MAX，则调用deal\_single\_con；

e）如果仅仅是一个OK包，则处理显示warning等信息，返回；

f）如果结果未取完且行数为1或者结果取完行数为0，调用proxy\_resultset\_new 构造result\_set，CAdvance\_base::deal\_fields 找到排序字段在列表中位置，调用merge\_one\_row 合并行数据，接着调用deal\_push()，通过子类的deal\_resultset()完成advance高级功能相关操作（group by/order by/distinct）；

g）结果取完，调用deal\_push\_and\_finish()，设置finish\_flag，调用deal\_push()。

### CON\_STATE\_SEND\_QUERY\_RESULT



该状态由CON\_STATE\_READ\_QUERY\_RESULT状态转移而来，主要是Proxy发送结果到客户端（proxy->client）。

基本流程：

1. 调用network\_mysqld\_write ()将数据发送到客户端；
2. 调用插件函数：

1）no\_shard：no\_shard\_send\_query\_result()

1.1）如果是dump binlog操作，在备机dump binlog的时候，如发生主备切换，则需要指向新的slave，状态设置为CON\_STATE\_READ\_QUERY\_RESULT继续获取结果集；

1.2）当con->resultset\_is\_finished==false，状态设置为CON\_STATE\_READ\_QUERY\_RESULT，读取剩余部分结果；

1.3）当查询结果集获取完全时，转移状态到开始的CON\_STATE\_READ\_QUERY状态等待，调用proxy\_read\_query()处理后续新的客户端查询。

2）group shard：group\_shard\_send\_query\_result()

2.1）如果结果没处理完（con->resultset\_is\_finishe==0），且拆分多步执行的情况，调用continue\_shard\_join()处理结果；

2.2）如果结果没有处理完且简单查询，设置状态为CON\_STATE\_READ\_QUERY\_RESULT，继续获取结果；

2.3）如果是advance处理逻辑，且执行状态未结束（con->advance\_base->m\_exec\_state != CAdvance\_base::SEND\_OVER），则状态设置为CON\_STATE\_READ\_QUERY\_RESULT，设置需要读取数据的后端链路（con->advance\_base->m\_read\_con），继续读取后端数据（需要将拆分的全部都处理完毕才可以发送给client）；

2.4）结果集处理完后继续，如果是advance流程，需要重新设置结果集行数。

## 其他中间状态

### CON\_STATE\_GET\_SOCK

GET\_SOCK状态属于由于后端分布式特性产生的中间状态。后续查询阶段中，经常需要向多个实例或多个set发送查询请求，如向备机查询或有分布式事务出现。GET\_SOCK状态负责为每个sql计划发送的地址集合con->sets\_addresses中的每个set产生至少一个socket，并添加con->sets\_address（注意区分两个变量），全部完成后转移到CON\_STATE\_POST\_READ\_QUERY。

CON\_STATE\_GET\_SOCK状态是从server获取一个新的套接字，用于各个状态轮转的中间状态存在，如向备机查询或有分布式事务。

调用插件函数no\_shard\_get\_sock/get\_sock()实现，主要流程：

1. 针对no shard，调用no\_shard\_get\_sock：
2. 如果已存在当前连接地址，则设置状态为（即为中间状态使用）CON\_STATE\_POST\_READ\_QUERY；
3. 如果是不存在的新连接地址，则增加新连接数m\_new\_connect\_num，设置状态为CON\_STATE\_CONNECT\_SERVER。
4. 针对group shard，调用get\_sock()。

### CON\_STATE\_GET\_AUTO\_ID

调用get\_auto\_id()插件函数处理自增列相关操作。

主要流程：

1. 对于首次进入，调用write，将事件写入自增处理线程（句柄thread->auto\_inc\_producer\_fd）；
2. 处理线程事件触发后，调用deal\_auto\_inc\_thread()处理自增列；
3. 如果是第二次进入，调用assign\_next\_value设置next\_value，然后调用get\_sets\_sql\_post设置下发set的SQL，同时状态设置为CON\_STATE\_READ\_QUERY。

## 断链状态

函数network\_mysqld\_con\_handle处理不同客户端连接，这里主要针对客户端没有数据发送的情况：

1. 如果客户端主动关闭连接，则关闭客户端，状态设置为CON\_STATE\_CLOSE\_CLIENT；
2. 如果客户端异常终止，则关闭server，状态设置为CON\_STATE\_CLOSE\_SERVER；
3. 如果是server关闭，则调用network\_mysqld\_con\_handle\_connection\_broken：

1）server未知原因关闭，状态设置为CON\_STATE\_ERROR；

2）server鉴权失败，状态设置为CON\_STATE\_CLOSE\_SERVER；

3）上报故障的连接

空闲server，忽略继续

人为关闭server，状态设置为CON\_STATE\_READ\_QUERY

事务影响，状态设置为CON\_STATE\_ERROR

### CON\_STATE\_ERROR

### CON\_STATE\_CLOSE\_CLIENT

### CON\_STATE\_CLOSE\_SERVER

## Proxy执行状态机

在joiner\_t::report\_done调用执行状态机，具体堆栈如下：

tdsql::joiner\_t::report\_done

tdsql::CSubquery::report\_done

tdsql::flexible\_sel\_t::\_\_report\_done

operator()

tdsql::CAdvance\_join::deal\_resultset

tdsql::CAdvance\_join::deal\_resultset

tdsql::CAdvance\_base::deal\_push

tdsql::group\_shard\_read\_query\_result

tdsql::plugin\_call

状态机的入口是joiner\_t::exec()，通过状态转移实现SQL控制：

enum enum\_join\_state {

PULLING\_TABLE\_META, /\*\* 从db拉取表的定义\*/

PREPARE\_SQL\_LEX, /\*\* 用mysqld的语法解析对sql进行解析 \*/

OPTIMIZE\_JOIN, /\*\* 从db拉取数据，对表进行填充 \*/

//NOTE:所谓拉数据,其实就是在proxy执行具体的优化

DO\_JOIN, /\*\* 执行join \*/

DO\_ASYNC\_JOIN, /\*\* 异步join \*/

EXPLAIN\_SELECT, /\*\* 向用户发送explain结果 \*/

RELEASE\_TRX, /\*\* 释放事务 \*/

SIMPLE\_JOIN, /\*\* 执行简单join \*/

SIMPLE\_UNION, /\*\* 简单的能够直接下推的union \*/

FINAL\_STREAM\_ACTION, /\*\* 等待最后一个ACTION的结果\*/

NORMAL\_ACTION,

ERROR

};

### ERROR

#### start\_trx\_in\_join

### PULLING\_TABLE\_META

这个是初始状态，主要是从DB拉取表的定义并进行处理（防止出现proxy本地缓存表结构与DB不一致的情况）。

根据不同执行阶段，涉及如下三个流程：

#### joiner\_t::collect\_table\_meta

调用collect\_table\_meta()执行从DB开始拉取表定义的操作。

1. 遍历所有select查询中涉及的表，如果是derived table跳过，如果是系统表（information\_schema）报错（即系统表结构不会变更）；
2. 如果是视图，则调用collect\_view\_reference()获取视图依赖的表；
3. 将最终的表信息存储到set结构participate\_table中；
4. 然后遍历上面过滤后的participate\_table，如果配置文件中enable\_metacache=0即没有设置网关缓存表定义，则需要将表添加到需要拉取的链表need\_to\_pull中并拉取元数据，避免元数据陈旧，导致查询出错；
5. 如果通过元数据接口FGetMeta已经获取到最新表元数据，则继续，不用插入到need\_to\_pull，否则还是需要将该表插入到need\_to\_pull中；
6. 如果participate\_table为空，说明join中没有表，则状态跳转到PREPARE\_SQL\_LEX执行语法解析；
7. 如果need\_to\_pull为空，即已经全部获取所有participated tables的元数据，则状态跳转到PREPARE\_SQL\_LEX；
8. 构造查询表定义的SQL并下发：
9. 判断是否为视图：

如果是视图，则show create view;

如果是表，则show create table；

1. 设置下发的set并执行SQL。
2. noshard、全局分区表、视图，下发到第一个set，其余的获取共享内存中set信息，下发对应的set；
3. 获取元数据的epoll。

#### joiner\_t::collect\_table\_meta\_report\_data

调用joiner\_t::collect\_table\_meta\_report\_data()处理从server返回的结果集。

1. 依次遍历结果集每一行数据，解析TLV结构的码流；
2. 对表定义进行修正；
3. 获取set name，并设置table define到set name的映射setmetamap。

#### joiner\_t::collect\_table\_meta\_report\_done

调用joiner\_t::collect\_table\_meta\_report\_done()处理最后上报。

1. setmetamap为空，则对应set没有元数据信息，报错；
2. 调用CMetaCahce::FaddMeta更新本地元数据（使用前面collect\_table\_meta中获取的epoll）；
3. 调用CMetaCahce::FGetMeta获取元数据，即网关更新缓存的元数据；
4. 将need\_to\_pull不断pop，知道为空。

### PREPARE\_SQL\_LEX

用mysql的语法解析对sql进行解析。

PULLING\_TABLE\_META状态跳转过来。

该状态涉及如下流程：

#### parse\_sql

调用parse\_sql()执行语法解析，主要流程为：

1. 获取当前连接的信息，设置嵌入式数据库字符集，默认db；
2. 查看hint：derived\_merge，如果存在后续会调用merge\_derived\_table优化虚表，重写字段，然后重新加载调用reset\_sql()执行语法解析；
3. 如果是视图：
4. 调用reset\_sql解析SQL，判断是否为update/delete/insert操作（不支持）；
5. 展开视图。
6. 非视图操作，调用reset\_sql解析SQL（MySQL的sql\_yacc.yy解析）；
7. 获取变量信息，对常量变量替换（select list/where/having/group by/order by）；
8. 调用setup\_user\_mysql\_env设置embedded\_mysqld的用户变量；
9. 获取锁信息（LockTypeWrite/ LockTypeRead/ LockTypeNone）；
10. 是否存在where条件语句（不带where不支持）；
11. 对于insert select场景：
12. 调用is\_insert\_select解析insert select语句：
    1. 暂存表的库表名，插入字段，ignore，replace等属性；
    2. 如果是ON DUPLICATE UPDATE操作，则获取对应的字段和值存储在action\_str中（出参）update\_str，后续Action中作为更新操作SQL使用。
13. 调用make\_insert\_select\_action设置对应的action：

2.1）如果insert\_columns为空，即没有指定目标列（insert into t1 select \* from t2），则调用get\_table\_field\_info()获取表定义；

2.2）如果指定了目标列，则必须包含分区键（否则无法计算set）；

2.3）根据前面解析的插入字段构建action输入数据描述信息，存储到数组input\_layout；

2.4）设置Action类型：insert/replace；

2.5）如果存在on duplicate update，则设置批量更新语句batch\_update\_action\_str\_；

2.6）将所有DML action都存到actions中；

2.7）为全局索引构造Action。

10、对于update select场景：

1. 调用is\_update\_select解析update select语句：
   1. 过滤多表update不支持的场景：update order，update limit，不支持多列更新（set t1.a=t2.b）；
   2. 设置目标表的锁类型：TL\_WRITE，如果是RR隔离级别，其他表加读锁：TL\_READ\_WITH\_SHARED\_LOCKS；
   3. 解析update语句，拼接update set …SQL存储到action\_str，后续构造action使用；
   4. 根据更新字段构建额外的查询条件，较少proxy的数据量和网络流量：

update t1 set a=b+1 where …

=>

select \* from t1 where … and (a!=b+1)

* 1. 根据update目标列构造select语句（判断set的左右两侧表）：

update t1, t2 set t1.a=t2.a where ...

=>

select t1.\*, t2.a from t1, t2 where ... group by t1.pk

1. 调用make\_update\_action设置对应action：

2.1）过滤一些不支持场景（二级分区表），更新表必须有主键字段，不允许更新主键/分区键；

2.2）根据前面解析的字段构造字段布局信息layout；

2.3）设置action类型；

2.4）将当前action存储在变量actions后续使用；

2.5）为每个更新的全局索引构造对应的更新语句。

11、对于delete select场景：

1）调用is\_delete\_select解析delete select语句：

1.1）设置待删除库表信息；

1.2）锁类型TL\_WRITE，在RR隔离级别下，对其他表加读锁TL\_READ\_WITH\_SHARED\_LOCKS；

1.3）根据待删除的列设置select的item\_list。

2）调用make\_delete\_action，将构造的actions存起来后续使用；

12、cutoff\_unused\_columns\_from\_tables ->get\_pruned\_tables()剔除原始表中不使用的列，简化临时表结构体；

13、遍历经过裁剪后的表，查询缓存，如果不存在则调用嵌入式数据接口执行create table，并建立原库表与嵌入式库表的对应关系rename\_map；

14、对裁剪后的表做执行优化：

1）rename\_parser\_dbtable对嵌入式表进行重命名（T1）；

2）调用reset\_sql执行语法解析；

3）调用merge\_derived\_table消除虚表。

15、调用make\_shardkey\_dict构建表-shardkey的映射关系；

16、根据配置文件是否优化简单查询，调用is\_simple\_join()判断是否为简单查询：

1）如果是join语句中存在union，非简单查询；

2）是否存在不能下推的用户自定义变量；

3）调用is\_simpe\_select()判断是否为简单select：

3.1）获取ON关联字段，调用my\_optimize\_cond执行优化；

3.2）非简单查询过滤：union，subselect包含子查询，子查询返回多行，子查询有聚合函数/group by/limit等；

3.3）如果是IN子查询操作，构造等值类，存储在equal\_classes；

3.4）设置表类型（shard，noshard，replicated table，global part）调用check\_duplicate\_output执行表级别的结果类型判断：

replicated [left] join replicated => replicated

shard [left] join replicated => shard

replicated left join shard => replicated

3.5）收集WHERE和ON中的等值链：gather\_equal\_fields()，gather\_equal\_fields\_from\_on\_expr()；

3.6）根据前面的等值关系，调用check\_relation\_pushdown()判断是否SQL能够下推。

执行完毕后：

如果是简单join且需要临时表，则设置状态为SIMPLE\_JOIN（状态机的另一个流程），设置enable\_stream\_action=false。DML大部分都是使用通用处理模型，只有当SQL能够直接下推DB且不在事务中的INSERT语句这种情况，才会使用流式处理模型；

17、调用is\_simple\_union()判断是否为简单UNION：

如果是简单union（子查询能下推；分组包含分区字段；不存在分组时，如果存在聚合函数或者having，则不能下推；不包含order by；不包含limit；如果包含distinct，或者为union语句，则结果字段必须要求包含分区字段），则设置状态为SIMPLE\_UNION，下推DB；

18、设置下一个状态为OPTIMIZE\_JOIN（状态机轮转）。

### OPTIMIZE\_JOIN

从db拉取数据，对表进行填充。

执行完语法解析后，会状态轮转到OPTIMIZE\_JOIN，主要是执行具体优化。

根据不同执行阶段，该状态主要涉及如下流程：

#### joiner\_t::optimize\_join

当后台异步写任务全部结束的时候，调用optimize\_join()执行优化。

其主要流程：

1. 初次进入该函数optimizer为空，需要申请，用于后续优化；
2. 调用build()执行优化

1）modify\_joins简化join，外连接消除（转inner join）；

2）获取条件查询，调用my\_optimize\_cond()进行谓词优化；

3）调用collect\_LHS\_equal\_item\_from\_filter收集等值链，存储到item\_equal\_map；

4）递归调用build\_current\_select构造当前查询优化：

4.1）第一次进入默认optimize\_easy\_subq==true，都需要调用optimize\_subquery()尝试优化，无法优化置optimize\_easy\_subq =false：

4.1.1）调用remove\_redundant\_subquery\_clauses处理IN/ALL/ANY/EXISTS中结果集为单行的场景，由子查询中转为select list；

4.1.2）EXISTS子查询：子查询不存在group by、having，聚合函数子查询为单行，非聚合函数且子查询非单行，将待优化子查询存储到subq\_opt\_list；

4.1.3）ANY/ALL子查询->MAX/MIN优化

4.2）对于无法优化的子查询，调用materialize\_cacheable\_subquery：

4.2.1）简单单表查询不需要物化；

4.2.2）调用create\_materialized\_table创建临时表；

4.2.3）设置需要被物化的子查询，存到current\_materialized\_subquery。

1. 事务控制

1）调用xa\_start\_transaction\_if\_necessary开启事务，申请m\_gtxn；

2）如果join在一个事务中，调用start\_silent\_transaction标识事务由proxy开启；

3）退出时检查事务是否提交或回滚。

1. 调用get\_next\_query继续：

1）获取待优化的子查询列表subq\_opt\_list，调用do\_replace优化，将子查询结果带入SQL中；

2）调用build处理子查询中常量替换；

3）如果需要物化子查询，设置类型为OPT\_TYPE\_MAT\_SQL。

1. 获取配置文件async\_join\_trigger配置项的值，并判断异步join结果集是否超过阈值：

判断参与连接的表的行数和列数乘积是否超过该值，超过则状态设置为DO\_ASYNC\_JOIN，使用后台线程执行查询；

不超过则状态设置为DO\_JOIN。

1. 子查询物化（OPT\_TYPE\_MAT\_SQL）：

1）调用create\_materialzed\_simplejoin，设置临时表库表信息（MATDB.tmp\_tb，注意与嵌入式临时表tmpdb区别），申请SimpleJoin对象；

2）get\_sets\_sql-> build\_scan\_sql获取下发set的SQL：

2.1）从group by/order by/having条件中提取访问的列；

2.2）判断是否所有聚合函数都有distinct，有则下推；

2.3）存在group by时判断group by的列是否是select list的子集，如果不是子集，则设置标志位tmptbl\_deduplicate，将结果写入到临时表去重；

2.4）不存在group by且无聚合函数时，设置select list为排序分组的字段，设置distinct下推的标志位distinct\_pushdown；

2.5）最终结果只有一行（single\_row\_result）的情况，无需去重和排序；

2.6）不使用临时表且不存在having且不存在聚合/聚合下推，则limit下推；

2.7）根据下推的优化，构造从后端数据库拉取数据的SQL；

2.8）构造创建临时表的语句；

1. 根据下推类型（EFirstSet、ERandomSet、else），设置下发set信息。

#### joiner\_t::optimize\_join\_report\_data

执行JOIN优化后，处理server返回的结果。

1. 如果子查询下推，则调用report\_subquery()暂存结果；
2. 如果是子查询物化，则调用simple\_join\_t::report\_data()，将数据写入buffer中，并处理聚合函数。
3. 其他情况，调用start\_filling\_table\_job()异步插入表数据。

#### joiner\_t::optimize\_join\_report\_done

最后处理结束后操作。

1. 如果是子查询物化，则调用simple\_join\_t::report\_done()执行最后聚合函数计算，利用临时表对结果最终排序，返回给客户端；
2. 如果是子查询下推，上报信息。

### DO\_JOIN

执行JOIN。

状态OPTIMIZE\_JOIN（optimize\_join）跳转过来。

#### do\_join

调用do\_join()执行同步获取结果并返回客户端的相关操作。

主要流程：

1. 如果还有job未结束，则调用嵌入式接口query()查询接口；
2. 调用嵌入式接口fetch\_next\_row()不断分批获取结果集；
3. 如果没有更多的数据可以插入目标表，则调用finish\_normal\_action()设置事务commit/rollback并返回结果；
4. 如果没有可执行action，则返回客户结果，如果还有数据需要继续插入目标表，则数据插入目标表。

### DO\_ASYNC\_JOIN

执行异步JOIN。

状态OPTIMIZE\_JOIN（optimize\_join）跳转过来。

#### do\_async\_join

调用do\_async\_join()完成后台异步查询。

主要流程：

1. 调用async\_join\_func增加异步执行的job（不需要等待获取结果）；
2. 如果没有更多的数据可以插入目标表，则调用finish\_normal\_action()设置事务commit/rollback并返回结果；
3. 如果没有可执行action，则返回客户结果，如果还有数据需要继续插入目标表，则数据插入目标表。

### EXPLAIN\_SELECT

#### joiner\_t::do\_explain

### RELEASE\_TRX

#### release\_silent\_transaction

调用release\_silent\_transaction释放分片的事务。

主要流程：

1. 对于非全局事务控制，直接报错；
2. 对于事务分支为空的情况，调用回调函数finish\_transaction结束事务；
3. 调用do\_normal\_commit\_or\_rollback执行事务的commti/rollback。

### SIMPLE\_JOIN

执行简单JOIN。

该状态从PREPARE\_SQL\_LEX状态（parse\_sql）中跳转过来。

#### joiner\_t::do\_simple\_join

调用joiner\_t::do\_simple\_join()实现对简单查询的处理。

具体流程：

1. 调用simple\_join:: get\_sets\_sql-> simple\_join\_t::build\_scan\_sql获取下发的SQL，处理一些非简单语句场景，需要重新设置状态；
2. 如果是直接下推的情况则不需要设置set的SQL；
3. 处理for update of；
4. 如果有待执行的action且为通用处理模式，则利用临时表（proxy\_tmpdb）将数据存储下来，调用嵌入式数据库query执行查询（make\_tmp\_table\_for\_join，make\_tmptbl\_for\_IS创建表）；
5. 设置回调函数：

simple\_join->send\_data —— sj\_argument\_t::\_\_send\_data

simple\_join-> send\_eof —— sj\_argument\_t:: \_\_send\_eof

simple\_join-> clean —— clean\_sj\_argument

1. 调用add\_sets\_sql设置下发set的SQL；
2. 调用init\_next\_query继续取set的SQL。

#### joiner\_t:: simple\_join\_report\_data

1. 调用report\_data
2. 判断advance下一个状态是否为CAdvance\_join::ENStateRead，暂时放弃当前work所有权。

#### joiner\_t::simple\_join\_report\_done

1. 调用report\_done
2. 如果还有未执行完的action，如果流式处理模式，则状态设置为FINAL\_STREAM\_ACTION；
3. 如果普通处理模式，则状态设置为NORMAL\_ACTION；
4. 如果action都处理完，设置join\_done、send\_done。

### SIMPLE\_UNION

执行简单的能够直接下推的UNION。

该状态从PREPARE\_SQL\_LEX状态（parse\_sql）中跳转过来。

#### joiner\_t::do\_simple\_union

调用do\_simple\_union处理简单的union。

#### joiner\_t::simple\_union\_report\_data

调用send\_client\_result发送客户端结果。

#### joiner\_t::simple\_union\_report\_done

### FINAL\_STREAM\_ACTION

joiner\_t::simple\_join\_report\_done设置流式/普通处理模式。

#### finish\_stream\_action

调用finish\_stream\_action完成流式处理。

主要流程：

1. 调用get\_current\_backend\_job等待所有child完成后才开始执行后续操作；
2. 判断事务事务提交（stream\_action\_committed），调用stream\_action\_need\_commit()判断是否流式Action需要提交，如果需要提交，则调用stream\_action\_commit()提交事务；
3. 计算流式action的影响行数、warning等信息；
4. 调用网络接口发送OK包；
5. 设置连接状态CON\_STATE\_SEND\_QUERY\_RESULT，标志位join\_done、send\_done等。

do\_action()中会处理Action：

1. 如果enable\_stream\_action==true，则采用流式处理模型：
2. 调用stream\_action\_prepare，构建child，以及与child通信的结构体CInternalChannel；
3. 调用stream\_action\_add\_sql：

2.1）parent将需要执行的SQL（sets\_sql）传递给前面申请的child，通过回调函数start\_action向后端DB发送SQL；

2.2）设置连接状态CON\_STATE\_IDLE；

2.3）调用CInternalChannel::NotifyChild()通知parent。

1. 递增后台运行job。
2. 如果enable\_stream\_action==false，则采用通用处理模型：
3. 申请结构体CSubquery（判断是否读备机）；
4. 遍历sets\_sql，为每个set设置下发SQL信息；
5. 根据action类型，设置查询的类型（insert select/update/delete/select）;
6. 调用init\_next\_query继续取结果。

joiner\_t::simple\_join\_report\_done()设置action处理过程中状态转移FINAL\_STREAM\_ACTION/NORMAL\_ACTION，调用状态机finish\_stream\_action/do\_normal\_action。

### NORMAL\_ACTION

#### do\_normal\_action

调用do\_normal\_action ()完成普通流程处理。

主要流程：

1. 判断后台是否还有运行的job未结束：等待所有child完成所有的insert/update操作；
2. 如果coquery\_builded==false，即没有构造用户态协程，则调用嵌入式接口build\_coquery构造用户态协程上下文（底层build\_coquery）；
3. 如果join\_done==false，即操作还未结束，则按照异步发送字节数调用嵌入式数据库接口插入数据：

3.1）如果仍然还有数据，则调用handle\_result()将结果插入到目标表；

3.2）否则，则递归调用do\_normal\_action()提交事务。

1. 如果join\_done==true，即完成操作，调用finish\_normal\_action()处理：

4.1）如果有未提交事务，利用回调函数结束事务；

4.2）发送给客户端，网络连接状态设置为CON\_STATE\_SEND\_QUERY\_RESULT。