DBProxy 连接池设计文档

变更说明

日期	版本	变更位置	变更说明	作者
2013-03-21	0.0.1		创建初始文档	侯金轩

本文档和最新代码不一定是严格匹配，仅仅作为参考

目录

1 目标

更改 MysqlProxy 的源代码，为其增加连接池功能，提供连接的复用功能。
同时，提供连接池的动态配置，便于 DBA 管理运维。

2 连接池概要设计

2.1 配置管理模块

主要完成与连接池配置、用户名密码配置、用户连接限制配置相关的配置文件读取和更新操作。这几类配置应该放在一个不同于 DBProxy 自己的启动配置文件里面，这样更容易管理设置。（也可以放在一块，就看如何组织文件结构了）

编码参考：可能的实现方案：(采用 xml？可以参考 Glib 库中的 Simple XML Subset Parser 或者 libxml。简单点自己定好格式就行了，可以一个连接池对应一行、一个用户对应一行、一个限制对应一行等？自定义格式最好与分节的结合使用，分解的配置文件操作可以参考：http://gtk-doc-cn.googlecode.com/svn/docs/glib/glib-Key-value-file-parser.html#GKeyFile)

2.2 认证管理模块

该功能模块主要实现用户名、密码的存取访问，通过用户的相关信息包括 username、passwd、ip 来实现。具体是实现握手阶段的握手包的生成和发送、并在 auth 认证阶段完成对用户名、密码及 IP 地址的校验。保证只有合法的用户才能够登陆 proxy 客户端进行后续的操作。

2.3 管理命令解析执行模块

实现对管理端口传送的类 os 带长参数命令的解析及参数合法性的验证。确保命令的合法性，同时调用其他模块的对外接口实现对用户信息的配置、连接限制的配置及连接池的配置、及信息查看和功能开闭等控制功能。

编码需要考虑的问题：如何对命令行选项做统一的处理？（getopt、getlongopt，参考 http://www.ibm.com/developerworks/cn/aix/library/au-unix-getopt.html#download http://www.docin.com/p-549292887.html，但是 GOptions 可能会更容易一些
）

2.4 连接池管理模块

该模块对外提供连接池操作管理接口，主要实现连接的创建、释放，连接池的初始化、获取特定连接池的空闲连接、连接池添加空闲连接、删除连接（主要是空闲时间过长的连接）、获取指定连接池中所有的连接及连接数、判断是否有空闲连接。为了较好支持连接池的连接大小控制需要对连接池中各用户对应的连接数的维护？

连接池需要记录的属性信息：每个 backend 对应一个大的连接池，然而只有同一个 username 对应的连接之间才能够复用，需要记录基于用户的连接池的限制配置信息、每个 backend 对应于某个用户的连接池的使用情况，及连接池的使用状态。

2.5 连接管理模块

该模块实现对 socket 的封装，提供数据的缓冲功能。提供连接的初始化、连接的释放、socket 的读写功能的封装。同时能够通过两个 socket 连接实现逻辑意义上的 client 与 server 间的连接。

为了便于用户行为信息的统计，需要能够记录对应客户端 sql 语句执行的条数（为了计算 sql 语句的执行速率需要涉及一种采样方式）。能够在记录每条语句的执行时间，便于在得知后端的响应时间的分布。

同时为了做到连接的复用，需要在连接中记录相应的上下文信息。

2.6 连接限制模块

该模块维护一个现有用户已经建立的与 Proxy 的连接数目，在新连接请求到来时，通过对比已建立连接数和在用连接数的关系，对用户连接做一定的控制。防止过多的连接打在 proxy 上面。

2.7 词法分析模块

该模块基于 flex 技术，对特定的 sql 语句进行解析，负责将 sql 语句分解成一串 tokens，便于后续根据 sql 语句关键字构造连接的上下文进而影响 sql 语句分发策略及连接复用的策略。可以复用 mysql-proxy 或 spock-proxy 现有的大部分代码？

2.8 性能统计模块

维护全局的信息统计变量，包括 sql 语句的执行时间、查询语句执行状况打印的接口(类似于慢查询功能，执行完毕后查看标志变量日志打印是否开启，若开启就会调用该模块接口将执行时间超过某个时间的查询语句打印到日志信息;或者是在特殊语句跟踪开启的情况下能够打印 drop、alter 语句的执行记录)。

简单起见维护一个 int[1001]数组？分别对应 0-1ms，1-2ms，2-3ms…..>1000ms？

2.9 Sql 语句过滤模块

需要对注册的已经 sql 语句进行过滤，如敏感语句的日志记录，危险语句的阻挡等。

3 模块详细设计

3.1 配置管理模块详细设计

3.1.1 数据结构设计

因为会有 ip 段的存在，若是通过字符串匹配来判断某个 ip 是否属于指定的 ip 端，实现比较麻烦。为了将方便的处理，我们将单个 ip 和 ip 段都转换成一个区间，通过整数的比较来判断 ip 是否属于哪个 ip 段。(这种处理方式是简化只支持 A.%,A.B.%,A.B.C.%这几种类型的 ip 段，也可以是掩码表示的，但需要时连续的)

a. Ip 段数据结构

结构定义：

struct ip_range{

```
    GString ip;
    glong min_ip;
    glong max_ip;
}
```

字段说明：

字段名	类型	说明
ip	GString	Ip 地址或地址段
min_ip	glong	ip 区间的对应的最小整数
max_ip	glong	ip 区间的对应的最大整数

b. 用户信息结构体

结构定义：

```
struct user_info{
    GString username;
    GString passwd;
    GPtrarray *ips; // array of struct ip_range *
}
```

字段说明：

字段名	类型	说明
username	GString	用户名
passwd	GString	密码(同一用户对应的密码相同)
ips	GPtrArray	允许访问的 ip 范围列表

c. 连接限制结构体

结构定义：

```
  struct conn_limit{
    GString username；
    struct ip_range ip;
    gint max_conn;
  }
```

字段说明：

字段名	类型	说明
username	GString	
ip	struct ip_range	限制访问的 ip 段
max_conn	gint	限制最大的连接数

d. 连接池限制结构体

结构定义:

```
  struct pool_limit{
    gint min_pool_conn;
    gint max_pool_conn;
    gint max_idle_interval;
  }
```

字段说明：

字段名	类型	说明
min_pool_conn	gint	连接数最小值（包括在用的和空闲的）
max_pool_conn	gint	连接数最小值（包括在用的和空闲的）
max_idle_interval	gint	连接的最大空闲时间（采用独立线程做连接空闲超时的处理）

e. 全局连接限制字典

采用 GHashTable<GString username,GPtrArray >,该变量主要是用户连接请求时来查询用户连接限制的。

说明：

对该字典的操作主要是：获取指定 username@ip 的连接限制数，并能快速查询到对应用户已经存在的连接数（这个在后续连接限制里面会使用到）。（已存在用户的连接数目保存同样需要通过 username、ip 快速定位对应的项）

f. 全局连接池配置字典

采用 GHashTable<GString username, struct pool_limit *>，该变量主要是在维护后端连接池时查询。当然设置是会更新对应的配置项。

g. 全局用户名密码配置字典

采用 GHashTable<GString username, struct user_info *>,该变量主要是存储的 proxy 上面注册的用户的用户名、密码及对应的访问 ip 列表。由配置管理模块维护，主要是在用户认证阶段使用。

3.1.2 接口设计

3.1.2.1 用户信息操作相关

a. 添加用户

函数说明：

boolean addUser(const char *username, const char *passwd, const char *ip):添加用户信息。若添加成功，返回 true，反之、返回 false；

参数说明：

参数名称	参数类型	说明
username	const char *	添加的用户名
passwd	const char *	对应的密码
ip	const char *	对应的 ip 段

b. 更新用户密码

函数说明：

gboolean modifyUserPwd(const char *username，const char *oldpwd， const char *newpwd):修改原先的密码，需要通过原密码的验证;改正成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	需修改密码的用户名
oldpwd	const char *	旧的密码
newpwd	const char *	新的密码

c. 删除用户

函数说明：

gboolean deleteUser(const char *username, const char *ip):删除指定的用户，成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	需删除的用户名
ip	const char *	需删除的用户名对应的 ip 段

d. 获取用户密码

函数说明：

GString getUserPwd(char *username):获取指定用户的密码。

参数说明：

参数名称	参数类型	说明
username	const char *	需查询密码的用户名

e. 判定用户是否具有访问权限

函数说明：

gboolean isAuthed(const char *username, const char *ip,):判定指定用户是否已经存在，存在返回 true，反之 false。

参数说明：

参数名称	参数类型	说明
username	const char *	需查询密码的用户名
ip	const char *	用户的 ip 地址

3.1.2.2 连接限制操作相关

a. 添加对某个用户的连接限制

函数说明：

gboolean addConnLimit(const char *username, const char *ip, gint conn_limit)：添加某个用户的连接限制，成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	添加的用户名
ip	const char *	对应的 ip 段
conn_limit	gint	限制值

b. 更新对某个用户的连接限制

函数说明：

gboolean modifyUserConnLimit(const char *username, const char *ip, gint conn_limit)：修改某个用户的连接限制，成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	用户名
ip	const char *	对应的 ip 段
conn_limit	gint	新的限制值

c. 查询指定用户的连接限制值

函数说明：

gint getConnLimit(const char *username, const char *ip)：获取某个用户的连接限制值，0 为没有限制，没有设置取默认的设置，反之返回真实的限制值。

参数说明：

参数名称	参数类型	说明
username	const char *	用户名
ip	const char *	对应的 ip 段

3.1.2.3 连接池操作相关

a. 添加对某个用户的连接池配置

函数说明：

gboolean addPoolConf(const char *username, struct pool_limit* pool_conf)：添加某个用户的连接池配置，成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	用户名
pool_conf	struct pool_limit	连接池配置

b. 更新某个用户的连接池配置

函数说明：

gboolean modifyUserConnLimit(const char *username, struct pool_limit* pool_conf)：修改某个用户的连接池配置，成功返回 true，反之返回 false。

参数说明：

参数名称	参数类型	说明
username	const char *	用户名
pool_conf	struct pool_limit	新的连接池配置

c. 查询指定用户的连接池配置信息

函数说明：

struct pool_limit * getConnLimit(const char *username)：获取某个用户的连接配置，没有设置返回默认的设置，反之返回真实的配置。

参数说明：

参数名称	参数类型	说明
username	const char *	用户名

3.1.2.4 配置文件操作相关

gboolean addUserToFile(const char *username, const char *passwd, const char *ip):在配置文件中添加用户信息
gboolean modifyUserPWDToFile(const char *username， const char *oldpwd， const char *newpwd):在配置文件中修改用户密码
gboolean addConnLimitToFile(const char *username, const char *ip, gint conn_limit):在配置文件中添加用户连接限制信息
gboolean modifyConnLimitToFile(const char *username, const char *ip, gint conn_limit):更新用户连接限制信息
gboolean addPoolConfToFile(const char *username, struct pool_limit* pool_conf):在用户配置文件中添加连接池配置信息
gboolean modifyPoolConfToFile(const char *username, struct pool_limit* pool_conf):更新配置文件中连接池配置信息

3.2 连接池管理模块详细设计

3.2.1 数据结构设计

a. 连接池数据结构
结构定义：

```
typedef struct {
    GHashTable *users; /*GHashTable<GString, GQueue<network_connection_pool_entry>> */
    /** Number of connections being used by the clients, they do not appear in users hash table*/
    gint num_conns_being_used;/** 因为用户的连接池互异，这个数值只是代表该 backend 上面在用的用户连接数*/
    gint num_conns_total;// 便于查询连接池的状态（相同 backend 的连接池组织在一起）
    /** 这里因为是各个用户的连接池互异，因而需要记录每个用户对应的连接的使用数目。因而添加变量类型如后续所示的变量 GHashTable conn_num_in_use<GString username, gint num_conns_being_used>*/
    GHashTable conn_num_in_use<GString username, gint num_conns_being_used>; /**记录每个 backend 不同用户的连接池的使用情况*/
    guint max_idle_connections;// 默认的最大连接数（每个用户来说都会有自己的配置）
    guint min_idle_connections; // 默认的最小连接数（每个用户来说都会有自己的配置）
    gint max_idle_interval; // 默认的最大空闲时间（每个用户来说都会有自己的配置）
} network_connection_pool;
```

字段说明：

字段名	类型	说明
users	GHashTable	对应与 username 的连接池
num_conns_being_used	gint	backend 上面连接的客户端连接数
num_conns_total	gint	backend 上面所有的连接(包括连接池中未用连接和在用连接)

conn_num_in_use	GHashTable	记录对应每个用户的连接池使用数
max_idle_connections	gint	默认的连接池最大连接数
min_idle_connections	gint	默认的连接池最小连接数
max_idle_interval	gint	默认的连接最大空闲时间

说明：
单独用户的连接池的配置应该集中放置还是在单独的进行：
各有优缺点：
集中放置
优点：动态修改的时候都会比较方便，只需要修改一处。不会出现数据的不一致。
不足：每次对连接池进行维护的时候都需要查询相同的结构体，访问的时候连接的层次会有点深。
各自存放
优点：每次都是访问自己的限制变量。
不足：更新的时候或是新增加配置用户的时候，各个连接池里面的限制变量都需要做相应的修改会造成暂时不一致。一个很大的锁??
我们的场景：配置的修改不太频繁。后续可能会做到连接池用户和具体连接的同步？
实现的难易程度来看，集中会相对方便一些。
用户自身的连接池配置结构体为：

```
struct pool_config{
    gint max_idle_connections;
    gint min_idle_connections;
    //gint initial_connections;   这里就不用了，我直接初始化为最小链接就可以了
    gint max_idle_interval;
}
```

这个在 3.1.1 中已经提到。

b. 连接实体数据结构

结构定义：

```
typedef struct {
    network_socket *sock;                /** the idling socket */
    network_connection_pool *pool; /** a pointer back to the pool */
    GTimeVal added_ts;        /** added at ... we want to make sure we don't hit wait_timeout */
} network_connection_pool_entry;
```

字段说明：

字段名	类型	说明
sock	network_socket *	连接实体对应的后端的 socket
pool	network_connection_pool *	连接实体到连接池的回指指针
added_ts	GTimeVal	添加到连接池时的时间，记录连接的空闲时间

可能需要查询连接池内连接的信息，例如显示：ip：user：total：inuse 需要有个地方存放，每个用户对应单个 backend 的最大连接数和最小连接数，实际连接数可以通过 Gqueue->len 获取.

3.2.2 接口设计

a. 获取某个用户所有的空闲连接

函数说明：

GQueue network_connection_pool_get(network_connection_pool *pool,const char * username): 获取后端某个 backend 上面指定 username 对应的空闲连接列表,若存在空闲的列表返回链表地址，反之返回空；

参数说明：

参数名称	参数类型	说明
pool	network_connection_pool *	backend 对应的连接池的指针
username	const char *	对应的用户名

b. 向指定的连接池中添加新的连接

函数说明：

network_connection_pool_entry * network_connection_pool_add(network_connection_pool * pool, network_socket * network_socket)：将连接添加到后端的连接池中，返回包含指定连接的连接实体。

参数说明：

参数名称	参数类型	说明
pool	network_connection_pool *	backend 对应的连接池的指针
network_socket	network_socket *	将要添加到连接池的连接

还需要如下功能接口函数：

往连接池中添加连接：network_connection_pool_add(pool,network_socket)

删除指定的连接体：network_connection_pool_remove(pool,network_socket)

删除指定的连接：network_connection_del_by_conn(pool,network_socket)

获取连接池中的所有空闲连接：network_connectio_pool_get_conns(pool,username)

获取空闲连接数：get_connections_poo_get_conn_num(pool,username)

连接池初始化：network_connection_pool_new(void)

连接池释放：network_connection_pool_free(pool)

获取某个用户对应的连接限制数 min：（已经放在配置管理模块中）

获取某个用户对应的连接限制数 max；

3.3 连接管理模块详细设计

3.3.1 数据结构设计

a. 连接结构体(socket 的封装结构体)

结构定义：

该结构比较复杂，但是总的来说结构中已经该记录如下几类信息：

1. 对应的 socket fd;// 数据读写使用，及相应的事件结构
2. 用于数据传输的中间结构，recv_queue 及 send_queue;

3. Client 端或 server 端的一些属性信息，如：
   a. 握手数据包信息，用于 auth 阶段使用;
   b. Client 段认证的用户名、密码信息;
   c. Client 端及 server 端的 db 信息，autocommit 等属性的值;
4. 等一些其他信息，会随着支持的语句的增加而更加丰富；
5. 所属的认证阶段的 ip 段。

实际结构定义如下：

```
typedef struct {
    int fd;                     /**< socket-fd */
    struct event event; /**< events for this fd */
    network_address *src; /**< getsockname()  对应连接的 client 端*/
    network_address *dst; /**< getpeername() */
    int socket_type; /**< SOCK_STREAM or SOCK_DGRAM for now */
    guint8      last_packet_id; /**< internal tracking of the packet_id's the automaticly set the next good packet-id */
    gboolean packet_id_is_reset; /**< internal tracking of the packet_id sequencing */
    network_queue *recv_queue;
    network_queue *recv_queue_raw;
    network_queue *send_queue;
    off_t header_read;
    off_t to_read;
    /**
     * data extracted from the handshake
     */
    network_mysqld_auth_challenge *challenge;/** server 端发送的握手包信息：对于 client 端的连接即是 proxy 端生成的数据包；对于 server 端的连接对应的是连接池建立时 mysql server 发送过来的认证数据包信息 **/
    network_mysqld_auth_auth *auth;/** client 端发送过来的，auth 数据包。包括用户名，加密的密码及可以定义如下：
typedef struct {
        guint32 client_flags;
        guint32 max_packet_size;
        guint8    charset_number;
        gchar * user;
        gchar * scramble_buf;
        gchar * db_name;
    } mysql_packet_auth;
//    加入这一段的意义是为了在 con 保存认证用的用户名、加密密码、及相应的数据库名**/
    network_mysqld_auth_response    *response; /** server 端发送的认证结果包信息：对于 client 端的连接即是 proxy 端生成的 auth_resp 包；对于 server 端的连接对应的是连接池建立时 mysql server 发送过来的认证结果数据包信息 **/
    gboolean is_authed;                 /** did a client already authed this connection */
    ip_range ip_rang; //  记录 client 端所在的 ip 地址段
    /**
```

```
* store the default-db of the socket
*
* the client might have a different default-db than the server-side due to
* statement balancing
*/
```

GString *default_db; /** 记录 client 端需要连接的后端 dbname 或者是 server 端 session 现在对应的 dbname*/

} network_socket;

重要字段说明：

字段名	类型	说明
fd	int	对应的 socket 连接的结构体
event	struct event	连接对应的事件结构体
src	network_address *	Socket 对应的 client 端，socket 为 client 与 proxy 间的连接指 client；当为 proxy 与 db 间的连接对应 proxy
dst	network_address *	Socket 对应的 client 端，socket 为 client 与 proxy 间的连接时指 proxy；当为 proxy 与 db 间的连接对应 mysql server 端
recv_queue	network_queue *	接受数据包存放的缓存队列
send_queue	network_queue *	发送数据包存放的缓存队列
challenge	network_mysqld_auth_challenge *	主要保存认证阶段的随机串
auth	network_mysqld_auth_auth *	这里存放的是 client 发送的用户名、密码、db
response	network_mysqld_auth_response *	应用端返回的认证信息
ip_ran	struct ip_range *	Client 端 ip 地址所属的认证 ip 的 ip 段
default_db	GString	Client 端要连接的对比信息

b. client 到 mysql server 的连接对应的结构体

结构说明：

该结构对应 client 到 mysql server 端的连接，整个连接包括上述连接结构体的两个变量。在支持连接池的情况下，server 端的连接时可以复用的。该结构体主要包括如下部分的变量：

1. client 端和 server 端的连接；
2. 连接对应的 plugin 及回调函数，以此表示不同端口上面的连接，不同端口上面的连接会做不同的处理；
3. 标示数据包信息的变量：包括包头信息里面的语句类型、词法分析后的 tokens 列表；
4. 标示数据传输阶段的标志变量；
5. 事关连接复用的连接属性信息，如：创建但未关闭的 Prepare Statement 的 id 列表、以普通查询发送的 statement 的名称及是否在事务中的标示变量信息；（我们认为即使是 autocommit 为 0 的情况下，前一个事务结束后即 rollback、commit 执行结束后后端连接仍然是可以复用的。）
6. 查询需要发往哪些 backend 的信息、缓存需要继续使用的 backend 连接信息；
7. 各个 backend 返回结果集情况的信息；
8. 在执行的 sql 语句信息，及开始的执行时间；

struct network_mysqld_con {

```
/**
 * The current/next state of this connection.(在状态机中使用)
 * @see network_mysqld_con_handle
 */
network_mysqld_con_state_t state;
 // this only holds the host names,
/**
 * The server and client side of the connection as it pertains to the low-level network implementation.
 */
network_socket *server;
network_socket *client;
/**
 * Function pointers to the plugin's callbacks.
 * @see chassis_plugin_config
 */
network_mysqld_hooks  plugins;// 一堆函数指针，连接到不同的端口的连接会对应不同的 plugin，对应的 hook 函数不一样。但是接口函数，不同的 plugin 是一样的。
/**
 * A pointer to a plugin-private struct describing configuration parameters.
 *
 * @note The actual struct definition used is private to each plugin.
 */
chassis_plugin_config *config;// 这里每个 plugin 里面分别定义自己的 config，结构体很不一样
/**
 * A pointer back to the global, singleton chassis structure.
 */
chassis *srv; /* our srv object */
/**
 * A boolean flag indicating that this connection should only be used to accept incoming connections.
 *
 * It does not follow the MySQL protocol by itself and its client network_socket will always be NULL.
 */
int is_listen_socket;
/**
 * An integer indicating the result received from a server after sending an authentication request.
 *
 * This is used to differentiate between the old, pre-4.1 authentication and the new, 4.1+ one based on the response.
 */
```

```
guint8 auth_result_state;
/* track the auth-method-switch state */
GString *auth_switch_to_method;
GString *auth_switch_to_data;
guint32    auth_switch_to_round;
gboolean auth_next_packet_is_from_server;
/** Flag indicating if we the plugin doesn't need the resultset itself.
 *
 * If set to TRUE, the plugin needs to see the entire resultset and we will buffer it.
 * If set to FALSE, the plugin is not interested in the content of the resultset and we'll
 * try to forward the packets to the client directly, even before the full resultset is parsed.
 */
gboolean resultset_is_needed;
/**
 * Flag indicating whether we have seen all parts belonging to one resultset.
 */
gboolean resultset_is_finished;
/**
 * Flag indicating that we have received a COM_QUIT command.
 *
 * This is mainly used to differentiate between the case where the server closed the connection because of some error
 * or if the client asked it to close its side of the connection.
 * MySQL Proxy would report spurious errors for the latter case, if we failed to track this command.
 */
gboolean com_quit_seen;
/**
 * Flag indicating whether we have received all data from load data infile.
 */
gboolean local_file_data_is_finished;
/**
 * Contains the parsed packet.
 */
struct network_mysqld_con_parse parse;
/**
 * An opaque pointer to a structure describing extra connection state needed by the plugin.
 *
 * The content and meaning is completely up to each plugin and the chassis will not access this in any way.
 *
 * @note In practice, all current plugins and the chassis assume this to be network_mysqld_con_lua_t.
 */
```

```
void *plugin_con_state;
/**
 * track the timestamps of the processing of the connection
 *
 */
chassis_timestamps_t *timestamps;
/* connection specific timeouts */
struct timeval connect_timeout; //  连接的超时时间，是 plugin 级别的？
struct timeval read_timeout;
struct timeval write_timeout;

// elements new added
// the next one 'servers' has the real socket connection
GPtrArray *backends;//  在确定 sql 发送到哪个 backend 时，用到
GPtrArray *    servers;          // array of MULTIPART_DATA servers
GPtrArray *    cache_servers;        // cash array of MULTIPART_DATA servers，缓存的连接
guint8         keep_srv_con;         // keep server connections
GPtrArray *sql_tokens; // sql 语句的 tokens  序列
struct {
    guint32 len;
    enum enum_server_command command;
    /**
     * each state can add their local parsing information
     *
     * auth_result is used to track the state
     * - OK    is fine
     * - ERR will close the connection
     * - EOF asks for old-password
     */
    union {
        struct {
            int want_eofs;
            int first_packet;
        } prepare;

        query_type query;
        struct {
            char state; /** OK, EOF, ERR */
        } auth_result;
        /** track the db in the COM_INIT_DB */
        struct {
            GString *db_name;
        } init_db;
    } state;
```

```
} parse2;

GHashTable stmtids; //  标示已经创建的 prepare statement 的 id
GHashTable stmtname;//标示已经创建的 prepare statement 的名字，主要是针对
int tx_flag; //  标示是否在事务中
GPtrArray *pending_conn_server; //结果还没有全部返回的 server 集
};
```

重要字段说明(主要考虑我们新添加的变量)：

字段名称	类型	说明
to_backends	GPtrArray *	需要发送 sql 的 backends(若是不考虑分片，可以去掉该变量)
servers	GPtrArray *	已经分配的连接信息
cache_servers	GPtrArray *	用于缓存连接信息(若是不考虑分片，可以去掉该变量)
sql_tokens	GPtrArray *	Sql 对应的 token 列表
parse2	内嵌 struct	用于标示认证、数据传输阶段的变量
stmtids	GHashTable	Prepare 语句的 id 列表
stmtname	GHashTable	Prepare 语句的名称列表
tx_flag	gint	是否在事务中的标志变量
sql_execute	GString	con 在执行的 sql 语句变量
tx_begin	chassis_timestamp_t	事务开始时间点
pre_begin	chassis_timestamp_t	Prepare 语句开始时间点
exe_begin	chassis_timestamp_t	语句开始执行时间

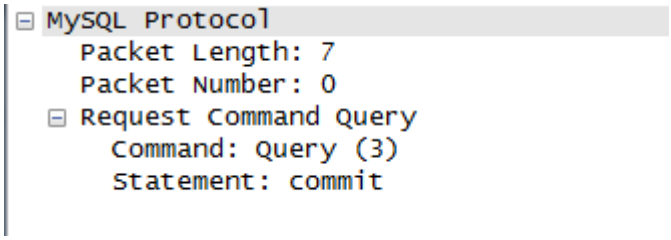
分配连接的中间结构：(参考 spockproxy，这个主要是考虑以后分片的需要)

```
typedef struct multipart_data
{
    int             backend_ndx;// index to the backend - used by LUA script
    guint32         read_wait_count;
    network_socket  *server;   // pointer to existing network_socket object
    GString         *sql;      // new sql if required, else use clients
    network_mysqld_con *con;   // pointer to the connection_state object
} MULTIPART_DATA, *PMULTIPART_DATA;
```

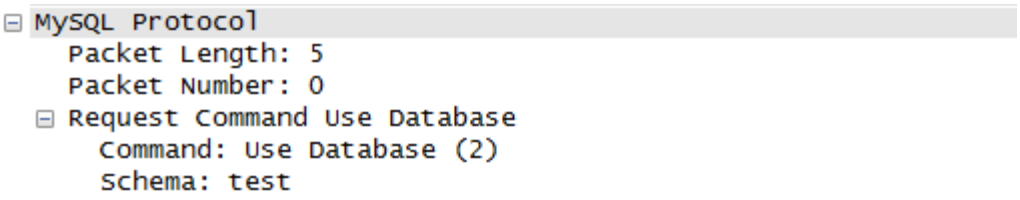
c. 数据包结构体

说明我们在程序中常用的数据包有 ok packet、error pack、eof packet、init db packet 及 commit packet。Ok、error 和 eof 对应的数据包在 mysql-proxy 中都有实现。因为我们需要在多个 db 之间支持连接的复用，且 server 端返回包事务标志位会存在错误，需要在隐式提交 set autocommit = 1 时，需要主动 commit 一次。

Commit 是普通的数据包，如下图：



Init db 的数据包如下图：



因而定义 init db 的数据包和 commit 数据包如下：

struct command_packet {

Gstring head; // 数据包包头

GString command: //命令

GString data; // 对应 use database 的 Schema，Commit 命令的 statement

}

3.3.2 接口设计

a. 获取某个用户所有的空闲连接

函数说明：

network_socket * network_mysqld_con_new():获取后端某个 backend 上面指定 username 对应的空闲连接列表,若存在空闲的列表返回链表地址，反之返回空；

参数说明：

参数名称	参数类型	说明
pool	network_connection_pool *	backend 对应的连接池的指针
username	const char *	对应的用户名

b. Initdb 数据包的构建

c. Initdb 数据包的发送

d. Commit 数据包的构建

e. Commit 数据包的发送

连接的创建：network_mysqld_con_new()

连接的释放：network_mysqld_con_free()

一些特殊包的处理：

// 除了现在 mysql-proxy 已经实现的 ok 和 error packet 还增加 init_db 的数据包

发送 init_db 数据包：

network_mysqld_con_send_initdb(network_socket *con);// ok 和 error 都是发往 client 端的。这个数据包是发送到 server 端的。

Socket 创建释放及 socket 数据处理函数（读取数据入 resv_queue，将 send_queue 中的数据写入到 sock 中）

3.4 认证管理模块详细设计

3.4.1 认证管理数据结构

a. 用户信息结构体

结构定义：

```
struct user_info{
    GString username;
    GString passwd;
    GPtrarray *ips; // array of struct ip_range *
}
```

字段说明：

字段名	类型	说明
username	GString	用户名
passwd	GString	密码(同一用户对应的密码相同)
ips	GPtrArray	允许访问的 ip 范围列表

b. 全局用户信息变量

结构定义：

采用 GHashTable<GString username, struct user_info users >,用户认证阶段通过用户名查询得到相应的密码，并确保 ip 地址在允许的范围内。

字段说明：

字段名	类型	说明
username	GString	用户名
users	struct user_info	密码(同一用户对应的密码相同)

3.4.2 接口设计

添加新用户的接口：addUser(GString *username, GString *ip, GString *password)

addUserWithDB(GString *username, GString *ip, GString *password,GString *DBname)

方便性考虑(暂时只支持单 IP 地址，或%表示的 ip 段)

删除用户的接口：deleteUser(GString *username.GString *ip)

更新用户信息的接口：只能通过 username 更新密码信息

updatePassword(GString *username, GString *password)

updateIplist(Gstring *username, Gstring *ipold,Gstring *ipnew)// 已经在配置管理中实现

获取密码信息 GString getPassword(const char *user)
判断 ip 字段是否在某个用户允许的 ip 端 gboolean isIPAuthed(const char *user,const char *ip)
验证密码正确性 checkPassword(GString *username, GString *randstr, Gstring *scramble)

3.5 连接限制模块详细设计

3.5.1 连接限制模块数据结构

需要维护一个全局的每个用户的已经与 proxy 建立的连接的数目。因为这个限制时基于 username@ip 地址的，因而为了便于更新与查询需要采用用户名加 ip 地址作为 key，因而需要快速定位到 ip 所在的 ip 段。因为，在认证阶段我们存储的是<username, userinfo>，需要查询该变量获取 ip 段(这个值需要记录下来)，然后查询连接数记录。连接释放时也可以快速的定位更新连接数。其对应的结构体如下：

GHashTable<GString username_ip，gint cons_in_uses >:对应于每条连接限制列表信息，记录每个 username@ip 的在用的连接数。

3.5.2 接口设计

查看用户连接是否已经查过连接限制数：gboolean isAvailable(const char * username, const char *ip_range)
将用户连接数加 1： gint incUserConns(const char * username, const char *ip_range)
将用户连接数减 1： gint decUserConns(const char * username, const char *ip_range)

3.6 性能统计模块详细设计

3.6.1 性能统计模块数据结构

需要在全局的 chassis 变量中，添加相应的开关参数。如下：
gboolean printed_slowlog: 标示是否开启 sql 语句执行记录的打印功能；
gboolean record_drop: 标示是否统计特殊语句
gint perf[1001]: perf[i] 记录这段时间内响应时间在[i, i+1）ms 内执行的语句的条数,perf[1000]指执行时间大于 1000ms 的语句的条数。

3.6.2 接口设计

开启/关闭 sql 语句记录功能
开启关闭特殊语句统计功能

打印出性能统计信息

3.7 Sql 语句过滤模块详细设计

需要记录 admin 注册的危险的或敏感的语句的特征。可以采用数组等数据结构。但是更高效的方式是在代码逻辑中实现。代码逻辑中实现扩展性查，但是数据结构存储性能会降低较多。
代码中实现时：需要对不同的语句添加不同的处理分支，比较麻烦。
存储结构存储：记录敏感的语句类或危险的语句类。对于特定类型的语句判断是否是指定的语句，做出相应的动作。

4 重要逻辑流程

4.1 重要逻辑分析

4.1.1 配置管理相关逻辑

4.1.1.1 功能描述

与此相关的逻辑包括：1. 启动阶段用户信息、连接池信息、连接控制信息、连接池启动与否信息的初始化；2. 用户信息、连接池信息、连接控制信息的添加或修改；3. 用户信息、连接池信息、连接控制信息的持久化。

4.1.1.2 具体流程分析

1. 配置信息初始化
   1.1 初始化流程
   a. 启动阶段需要从配置文件读取加密的用户信息,解密后保存在内存变量中,即实现变量 GHashTable<UserName, struct user> 的初始化；
   b. 读取用户连接池相关的配置信息，保存在内存中，即初始化连接控制变量 GHashTable<UserName, struct pool_config *>；// 这里采用统一的配置管理，因为每个 backend 都有自己的 pool，而这个 pool 里面的连接在以用户划分。
   c. 读取默认的连接池配置，没有设置程序中指定相应的默认配置；// 默认配置不可动态修改
   d. 读取用户的连接控制相关配置信息，并初始化到内存变量 GHashTable<GString Username, GHashTable<GString ip, gint num>>; //注意这里的 ip 可能是 ip 段，需要判定 ip 是否在范围内？（最好能有个比较高效的方式实现）
   e. 读取默认的 user@单个 ip 的默认连接限制，没有设置的程序会指定相应的设置；
   f. 根据配置信息设置连接池开关参数，没有采用默认配置。默认连接池开启。

1.2 需要注意点

a. 连接限制没有指定时，使用默认的连接限制；
b. 连接限制为 0 时，连接是没有限制的；
c. 用户已经使用的连接数的记录；（认证成功时++，连接释放时--）

2. 信息修改更新及持久化

2.1 信息修改流程

a. 用户添加流程

首先查询相关用户是否已经存在，若 username 存在则将相应的 ip 端添加到 struct 具体根据 user 对应的 ip 地址段内（需判重）；若不存在，则在 hash 列表中添加 username 对应的用户信息。若指定持久化需要将信息持久化到磁盘。

b. 用户名密码信息更新

内存中根据用户名，查找找到对应的用户信息，更新即可。注意需要加密后持久化到配置文件。

注意：实现时，1 需要提供初始的密码信息，校对验证；2 更新不存在的用户，返回失败。

连接池、连接限制对应配置信息的添加，修改与用户信息的添加修改类似。操作流程基本类似。

2.2 需要注意点

a. 这部分特别是连接控制或连接池部分的设置，查询是比较频繁的，需要考虑采用比较高效的数据结构。设计中暂时采用的 glib 的 hash 表实现。后续编码中，根据需要须做相应的修改。

4.1.2 认证相关逻辑

4.1.2.1 功能描述

实现用户的登陆认证，在用户请求连接的初始阶段，主要是对应 mysql 协议的三次握手协议。负责生成认证的随机串，保存在连接属性中，组装生成 handshake 数据包，发送到 client。读取 client 端的 auth 数据包，初始化连接的属性信息（username，dbname，scrambed passwd 等信息），根据 client 端的 username，ip 地址，加密密码及连接使用情况对实现用户的验证。（连接限制应该在验证前还是在验证后），若认证成功则用户在用的使用数加 1；反之，不变向客户端发送认证失败数据包（包括连接超过限制和认证密码错误两种情况，当然在连接数过多时，可以发送一个连接数过多的数据包 client 应该会自动放弃连接，这样都会触发一个 socket 的 EV_READ 事件）。认证完成后，连接的状态进入 CON_STATE_READ_QUERY 状态。后续状态机中会对认证失败的情况进行处理，若失败连接的状态最终进入 CON_STATE_CLOSE_CLIENT 状态，然后释放资源。

4.1.2.2 具体流程分析

1. 登陆认证流程

用户登录认证的流程具体描述如下：

a. 有用户请求到来时，会触发 network_mysqld_con_accept，会构造一个 network_socket 变量 client，其 fd 设置为该监听端口上面执行 accept 函数获得的 fd 变量。之后构造一个 network_socket_con 变量其 client 指针指向上面构造的 client 变量，然后执行 network_mysqld_con_handle 函数；
b. 根据连接的端口如 admin 或 proxy，设置相应的回调函数；（为了避免后续 lua 的调用可以参考 spock-proxy 的做法，至少 proxy 端口上面的连接应该这么做。Admin 访问量比较少，可以沿用 mysql-proxy 的做法）；
c. 连接进入 CON_STATE_INIT 状态，proxy 自己需要产生 handshake 数据包（spock 中将自己伪装为5.1.0的mysqld，mysql-proxy admi-plugin将自己伪装成5.0.99 mysqld），将 handshake 数据包 append 到 client 端的 send queue 中，然后进入 CON_STATE_SEND_HANDSHAKE 状态；
d. 调用 network_mysqld_write 向 client 发送认证数据包，连接进入 CON_STATE_READ_AUTH 状态；
e. 调用 network_mysqld_read 从 client 端读取 auth 数据包，放在 client 的 recv_queue 中；(mysql-proxy 中读取了两次，首先会读取数据到 raw_quque 中，然后再读取到)
f. 对照分析 client 的数据包，初始化整个连接属性中的 username、scrambled passwd、dbname；
g. 查询连接控制列表，查询该 username@ip 对应的连接是否达到限制，若连接达到最大连接数，则向 client 返回错误包，错误信息为 too many connections，连接状态修改为 CON_STATE_ERROR；
h. 若该用户对应的连接数，没有达到最大连接数，对用户名、密码及 ip 地址进行验证，验证成功则返回成功数据包，相应的用户的连接数增加 1，反之返回错误包，错误信息是'Authentication failed for user@ip with password'；
i. 连接状态转变为 CON_STATE_READ_QUERY。

2. 需要注意点

a. 用户连接超过限制的情况；
b. 用户认证不成功的情况，包括用户名不存在，ip 不在指定范围内，密码不正确等。

4.1.3 命令解析模块相关逻辑

该部分逻辑比较简单。

需要注意点：

参数的判断，争取能高效支持尽可能多的命令。

4.1.4 连接限制模块相关逻辑

用户请求连接时，通过对比该用户已有连接和限制连接判定是否让用户连接到 proxy。是在认证过程中，密码验证之前进行的。过程比较简单。

4.1.5 连接池管理控制相关逻辑

连接池的管理控制与连接的状态关系比较大，其实现过程几乎涉及到用户查询请求的各个步骤，加上相对 mysql-proxy，DBProxy 的状态机会有较大的改变。接下来我们首先会对 DBProxy 的状态机做介绍，然后再对 DBProxy 为支持连接池流程上面的一些修改。此外，该模块需要能够对数据库连接池做到初始化、连接的添加及删除、连接的获取及连接的归还、连接的池的维护等。

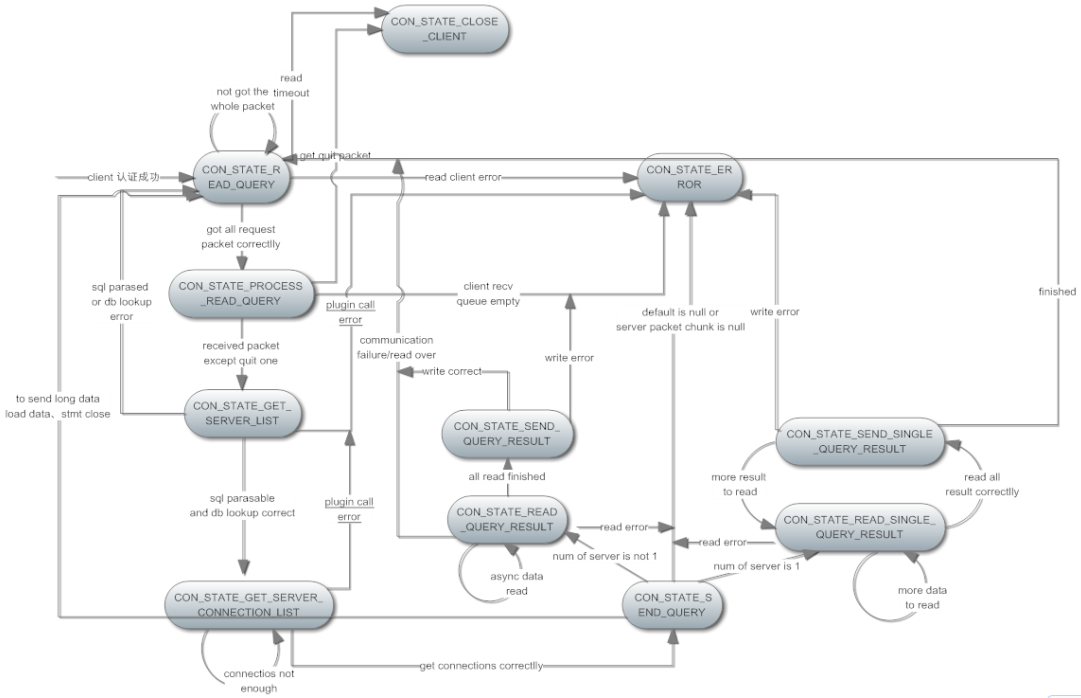
4.1.5.1 功能描述

依据 client 查询语句，确定后端 server 的连接是否可以归还到连接池中，主要考虑用户连接是否在事务中、是否在 prepare 语句中。进而在连接释放的阶段，来确定是否将连接释放到连接池当中。同时，需要记录一些连接的上下文信息比如 db 信息，以便在连接分配时对连接的部分属性进行初始化。

4.1.5.2 DBProxy 请求处理状态机

状态机类似于 spock-proxy 的状态机如下。由于第一期的目标是替换 lvs，没有分片的需求。实现过程只是对应从后端的单个 mysql server 读取结果的情形，对应下图中的 CON_STATE_READ_QUERY_RESULT 及 CON_STATE_SEND_QUERY_RESULT 这一路径。

各状态的功能具体如下：



1. CON_STATE_READ_QUERY：从 client 读取请求数据包，放置在 client 端的 recv_queue 中；
2. CON_STATE_PROCESS_READ_QUERY:对特定的 sql 语句进行词法分析，将 sql 语句转换成 token 列表；

3. CON_STATE_GET_SERVER_LIST：依据语句的特点查看主要是判断事务、Prepare 语句及 init_db 的语句，用户记录连接的上下文；接下来依据查询的特点确定将查询发送到后端的哪些 backend（若是只是替换 lvs 的这里只是做一个简单的负载均衡；若是读写分离还需要通过语句来确定是发向主库还是发向其中的一个备库；同样分表也是在这里做分发规则）；
4. CON_STATE_GET_SERVER_CONNECTION_LIST：是连接分配的阶段，依据步骤 3 的连接上下文，分配连接；连接选取的原则是：首先是从 cache 的连接中选取，若没有缓存的连接，再从连接池中选取可用连接，连接不够的情况下触发连接池创建的操作，重复三次，若任然没有足够的连接则释放已经分配的连接返回错误；（1.实现过程中，可以比较苛刻若连接在事务中、且查询不是事务或 Prepare 的第一个请求，则连接只能在已经缓存的连接中选取；2.需要考虑虽然是 set autocommit = 0 或 begin 语句但是语句可能执行不成功；3.需要在连接的上下文中添加记录重复次数的变量，每次连接全部分配完成时归零）
5. CON_STATE_SEND_QUERY：向后端的数据节点转发查询语句；
6. CON_STATE_READ_SINGLE_QUERY_RESULT:从后端读取 server 端的 sql 语句执行结果，这里需要对 set autocommit = 0，set autocommit = 1 ，rollback，commit，普通查询的 PREPARE、DEALLOCATE PREPARE，数据包头为 prepare 及 close 的语句的执行情况等。（注意需要记录连接的上下文信息 autocommit 的值、prepare statement 的 id、prepare statement 的名称以便连接复用时，连接的上下文初始化），读数据失败连接状态设置为 CON_STATE_ERROR，反之状态设置为 CON_STATE_SEND_SINGLE_QUERY_RESULT；
7. CON_STATE_SEND_SINGLE_QUERY_RESULT:将 server 端的 recv_queue 里面的数据 append 到 client 的 send_queue 中，将数据集 write 到 client 中。若写失败，将状态设置为 CON_STATE_ERROR。若写成功，但是结果没有读取完毕，将状态设置为 CON_STATE_READ_SINGLE_QUERY_RESULT，若写成功且结果处理完毕，将状态设置为 CON_STATE_READ_QUERY。

4.1.5.3 具体流程分析

请求详细流程分析

在连接分配时，需要查看是否开启连接池，若没有开启且后端 servers 不为空则不需要进行连接的再分配，若没有开启但是后端 servers 为空才进行连接的初始分配。读写分离是在连接池的接触上来完成的！

1. 普通 sql 语句执行流程

a. 用户发送 sql 语句即查询请求，会触发事件，调用 network_mysqld_con_handle,进入 CON_STATE_READ_QUERY 阶段处理流程。调用 network_mysqld_read 从 con->client fd 中读取数据至 client 端的 recv_queue。若读取数据成功则设置 con->state 为 CON_STATE_PROCESS_READ_QUERY;若只是读取了部分数据，则在该 fd 上面注册 EV_READ 及 EV_TIMEOUT 事件，保持 con->state 的状态；若读取失败，设置 con->state 为 CON_STATE_ERROR；

b. 处理 con->client->recv_queue 中的数据，分析协议头 TYPE = packet->str[NET_HEADER_SIZE + 0]。若 TYPE 为 COM_QUIT，则将状态设置为 CON_STATE_CLIENT_CLOSE；若 TYPE 为 COM_QUERY、COM_INIT_DB、COM_LIST_FIELDS、COM_CREATE_DB、COM_DROP_DB、COM_STMT_PREPARE、COM_STMT_EXECUTE、COM_STMT_CLOSE 做词法分析？将

con->state 设置为 CON_STATE_GET_SERVER_LIST；(在 sql 语句进行解析的时候，可根据是否开启连接复用、读写分离来选择是否对一些类型的 sql 语句做词法分析)

c. 在 get_server_list 阶段，来实现负载均衡？即后端 backend 的选取？读写分离选取主库或选取从库是在这里面选择？非事务中、非 Prepare 的 select 语句发送到主库，其余发送到从库。同时，根据 sql 语句类型设置相关的连接上下文的参数。若为 COM_INIT_DB，则更新连接的 dbname 属性（需要命令执行成功返回才能更新）；将 con->state 设置为：CON_STATE_GET_SERVER_CONNECTION_LIST;

d. 从步骤 c 指定的 backend 对应的连接池上面，获取空闲连接，将 socket 加入到 con->servers 内；若连接足够，则将 con->state 设置为 CON_STATE_SEND_QUERY；若连接不足，注册超时事件，con->state 不变等待 100us 在重试获取连接，但是最多重试三次；若连接获取的过程中出错，则将 con->state 设置为 CON_STATE_ERROR；

e. 接下来，将 query 语句发送到 server 端。若发送失败，则将 con->state 设置为 CON_STATE_ERROR，并向 server 端发送 COM_STMT_RESET 数据包（**这个是针对需要多个后端发送 query 的情况，对已经成功发送的 backend 做 reset**），向客户端发送 error 数据包。反之设置连接 con 在执行的语句的变量，并记录语句开始执行时间。（这里执行超时处理如何做？对应查询被 lock 情况），在不考虑分片的情况下，con->state 设置为 CON_STATE_READ_SINGLE_QUERY_RESULT。（网 server 端发送数据包是否考虑一步发送？）

f. 接下来从后端的 server 端读取数据读取数据，调用 network_mysqld_read(同时设置结果集是否读取完毕的变量(这个 mysql-proxy 在 src/network-mysqld-packet.c 中有专门处理的函数 network_mysqld_proto_get_query_result 可以参考))；若是接受成功将 state 设置为 CON_STATE_SEND_SINGLE_QUERY_RESULT；没有读取完毕数据包会再次注册监听事件，state 不改变；读取失败将 state 设置为 CON_STATE_ERROR，同时将 con->server->recv_queue 里面的数据包 append 到 con->client->send_queue 中；

g. 接着将 con->server->recv_queue 中的数据存放到 con->client->send_queue 中，接下来调用 network_mysqld_write 函数将数据发送到客户端。若后续还有返回结果则将 con->states 设置为 CON_STATE_READ_QUERY，反之设置为 CON_STATE_READ_SINGLE_QUERY_RESULT。若全部结果军返回，根据连接池是否开启，对连接做相应处理。若连接池开启，对普通语句，将连接释放到连接池中；若没有开启则保留 server 连接。保留链接。

2. 事务型语句处理流程

在没有启用连接池的情况下，事务型的语句处理和普通的 sql 语句的处理是一样的。在启用连接池的情况下，会在一下几个阶段有所区别：

a. 若执行语句是事务的第一条语句，包括：autocommit=1 时的 begin 语句、autocommit=0 时的 begin 语句 或者 set autocommit=0、cimmit 和 rollback 后的第一个语句（commit、rollback 除外）直接从返回包看也可以。（但是从返回结果看是否在事务中会存在一个问题）。现在倾向于，后者不需要单独考虑隐式提交。

```
⊟ Server Status: 0x0003
    .... .... .... ...1 = In transaction: Set
    .... .... .... ..1. = AUTO_COMMIT: Set
    .... .... .... .0.. = More results: Not set
    .... .... .... 0... = Multi query - more resultsets: Not set
    .... .... ...0 .... = Bad index used: Not set
    .... .... ..0. .... = No index used: Not set
    .... .... .0.. .... = Cursor exists: Not set
    .... .... 0... .... = Last row sebd: Not set
    .... ...0 .... .... = database dropped: Not set
    .... ..0. .... .... = No backslash escapes: Not set
  Warnings: 0
```

b. 在连接分配阶段，若是在事务中，不需要再重新做负载均衡及连接分配，直接使用之前的连接；

c. 连接释放时，查看返回包，若包头显示不在事务中将连接归还的连接池。

3. Prepare 语句处理流程

在没有启用连接池的情况下，Prepare 类型的语句处理和普通的 sql 语句的处理是一样的。在启用连接池的情况下，会在一下几个阶段有所区别：

a. 会定位 COM_STMT_PREPARE 语句及普通 query 语句但是有 PREPARE 关键字，若是对应的语句执行成功。记录下 prepare statement 的 id 至或 stmt 的名称至 GHashTable stmtids 或变量 GHashTable stmtname 中。当定位到 COM_STMT_CLOSE 语句或普通的 query 语句中有 DEALLOCATE PREPARE 关键字，且执行成功将相应的 id 或 prepare statement 的语句删除。当 stmtids 及 stmtname 都为空时，则表明连接不在 Prepare 中，反之在 prepare 中。是否在 Prepare 中，不能通过 server 端返回包获取，如下：

```
⊟ Server Status: 0x0003
    .... .... .... ...1 = In transaction: Set
    .... .... .... ..1. = AUTO_COMMIT: Set
    .... .... .... .0.. = More results: Not set
    .... .... .... 0... = Multi query - more resultsets: Not set
    .... .... ...0 .... = Bad index used: Not set
    .... .... ..0. .... = No index used: Not set
    .... .... .0.. .... = Cursor exists: Not set
    .... .... 0... .... = Last row sebd: Not set
    .... ...0 .... .... = database dropped: Not set
    .... ..0. .... .... = No backslash escapes: Not set
  Warnings: 0
```

b. 在连接分配阶段，若是在 prepare 中，不需要再重新做负载均衡及连接分配，直接使用之前的连接；

c. 连接释放时，查看变量 stmtids 及 stmtname 是否都为空，若都为空则释放连接，反之保有连接。

4. 连接分配和释放的流程

4.1 分配阶段

a. 若事务标志或 prepare 标志位真，即连接处于事务或 prepare 中，则直接使用 cached servers 中缓存的连接作为服务端；若 cached servers 为空说明执行过程中出现未知的错误，返回错误设置连接状态；

b. 若连接事务标志为假且 prepare 标志为假，单 100us 标志为真即连接处于 100us 缓冲阶段，则直接使用 cached servers 中缓存的连接作为服务端；若 cached servers 为空说明执行过程中出现未知的错误，返回错误设置连接状态；(也可以直接通过判断 cached server 是否有缓存的连接进行判断)

c. 若连接既不在事务又不在 prepare 中，并且 100us 标志为假，则说明没有缓存的连接可以使用，需要在 get_server_list 阶段做负载均衡在 get_server_connection_list 阶段做连接的重新分配；

4.2 连接释放阶段

a. 若连接事务标志或 prepare 标志为真，则将连接保持不释放；放入 cached server 内部缓存起来，并在上面注册空闲超时事件(超时时间为 t1)；

b. 若事务标志为假且 prepare 标志为假，连接不释放；维持 100us 一个可配置的时间，放入 cached server 里面并注册空闲超时事件(超时时间为 t2)；

5. 连接使用过程中异常事件的处理

在请求处理过程中，会出现连接中断、连接超时、执行超时等状况；需要对这些情况作相应的处理。

a. 若为连接异常中断，如 client 端被 kill 或者 server 端被 kill 等都会触发 EV_READ 事件，需要做特殊处理，这个在 spock 中已经考虑；

b. 对于超时事件，这里是指根据连接复用的需要在 DBProxy 注册的超时时间的处理；在实现过程中采取的方式为 kill 超时端的连接，具体实现过程如下：

1. Prepare 或事物中占用后端连接空闲时间超时，即该链接对应的 cached servers 中后端 connection 超时时间被触发，需要将事务 rollback、prepare stmt close 掉将 server 端连接关闭掉即可。按常理关闭 server 端的连接与至对应的事务及 prepare 会回滚或关闭，因而对 serve 端只需要关闭 socket 连接即可？对 client 端，需要对其返回错误信息，然后释放对应的资源并将 client 端对应的连接关闭。
2. 对于执行超时的连接，我们不能对连接执行 kill 操作。现阶段只能由 DBA 接入解决，不过 proxy 能够对外提供连接查询的信息。类似与 show processlist 的结果。不过会增加两个字段：a. 是否在事务中或 prepare 中的标志信息；b. 在 prepare 或在 prepare 中的时间；c. 执行的时间。

c. 对于客户端 sql 语句发送异常的处理：包括 rollback 多次、commit 多次这些都可以当做正常的执行过程来处理。但是对于 prepare 语句 close 多次需要特殊处理，处理结果如下：

1. 若是 cacheserver 为空：说明没有 prepare statement 需要 close，或事物需要提交直接返回应用端错误，查询不需要往 server 发送；
2. 若非空，对比要关闭的 id 或 statement 的名字是否在缓存中，没有即报错；
3. 若存在则需要发送请求至后端，返回执行结果。

对于 last_insert_id()等函数也是根据 prepare 的处理方式进行。

需要注意点

1. 连接如何复用？何种情况下可以复用？
2. 单 session 连接占用时间太长如何解决？
3. 需要对哪些上下文进行考虑？每个属性又该如何处理？
4. 对于连接池中的空闲连接，需要有空闲超时处理?
5. 对于分配出去的连接，需要处理执行时间过长的事务、执行时间过长的 prepare？另外对于执行时间过长的语句也要考虑？

对于情况 5，包括几种情况：

a. 连接分配出去但是没有使用的情况？可以通过注册超时时间处理。或等待 server 结果时间过长？也可以通过注册超时事件来处理。可以支持个性化的配置。但是超时了怎么办？回滚？提交？是个问题。做回滚处理

b. 一直占用连接，没有空闲。有查询或是有数据传输？如何处理？都会记录一个连接分配的时间，可以设定一个时间查过就坐相关操作？被动的触发，但是触发了该如何做是个问题。这个初步可以先不做处理，需要 DBA 接入处理，但是 show processlist 能查看在事务中吗？估计不行，需要提供在事务中的语句的查询功能？或者是打印到日志中（warning）

6. 正常的连接可以释放的条件？

   当连接处于非事务、非 prepare 中可以保留 100us，若在 100us 内没有请求查询，则将连接放回资源池。反之，对请求做处理。

5 参考文献

1. Mysql proxy 源代码：http://dev.mysql.com/downloads/mysql-proxy/#downloads
2. Spock proxy 源代码：http://sourceforge.net/projects/spockproxy/