# 概述

MySQL Proxy是一种在网络上**使用MySQL网络协议进行通信**的应用，提供了一或多个MySQL server与一或多个MySQL client之间的通信功能。由于MySQL Proxy使用的是MySQL网络协议，故其可以在不做任何修改的情况下，配合任何符合该协议的且与MySQL兼容的客户端一起使用。这其中也包括MySQL的命令行客户端，任何使用了MySQL客户端库的客户端，以及任何支持MySQL网络协议的连接器(connector)。

在最基本的配置下，MySQL Proxy仅简单的将自身至于服务器和客户端之间，负责将query从客户端传递到MySQL服务器，再将来自MySQL服务器的应答返回给适当的客户端。在更加高级的配置下，MySQL Proxy可以用来监视和改变客户端和服务器之间的通信。查询注入(query interception)功能允许你增加性能分析命令(profiling)，且可以通过 Lua 脚本语言对注入的命令进行脚本化控制。

通过拦截来自客户端的query，proxy能够插入额外的query到要发送给服务器的query列表中，然后在收到来自服务器的结果后再将额外信息移除。通过这个功能，你可以返回原始query对应的结果给客户端，同时还能为每一个query添加额外的信息性语句，例如，可以监控这些语句的执行时间和进度，或者分别对返回结果进行日志记录。

proxy允许你对query执行额外的监控、过滤，或者操纵，而不需要你对客户端做任何修改，甚至可以做到令客户端不感知其正在于proxy通信而不是与真正的MySQL服务器通信。

# 架构

MySQL Proxy的定位是存在于mysql client和mysql server之间的一个简单的程序，能够对从其上通过的数据进行检查、转换和直接进行相应操作。

**应用范围包括：**

负载均衡（load balancing）

故障处理（fail over）

查询追踪（query tracking）

查询分析（query analysis）

(...)

基于这个core能够将protocol的各个phase暴露给plugin。

connect -> auth;

auth -> command;

command -> disconnect;

command -> command;

connect -> disconnect;

auth -> disconnect;

上述生命周期中的每一个阶段都包含了多种协议状态变迁。以鉴权过程为例，其包含了至少 3 个包：

life-cycle中涉及的实体：Client, Proxy, Server;

Client -> Proxy [ label = "accept()" ];

Proxy -> Proxy [ label = "script: connect\_server()" ];

Proxy -> Server [ label = "connect()" ];

...;

Server -> Proxy [ label = "recv(auth-challenge)" ];

Proxy -> Proxy [ label = "script: read\_handshake()" ];

Proxy -> Client [ label = "send(auth-challenge)" ];

Client -> Proxy [ label = "recv(auth-response)" ];

Proxy -> Proxy [ label = "script: read\_auth()" ];

Server -> Proxy [ label = "send(auth-response)" ];

Server -> Proxy [ label = "recv(auth-result)" ];

Proxy -> Proxy [ label = "script: read\_auth\_result()" ];

Proxy -> Client [ label = "send(auth-result)" ];

...;

当核心层core面对大量connection的时候是具备自由伸缩特性时，plugin/scripting层就能够轻易对end-users隐藏细节，并简化客户端的实现。

MySQL Proxy是通过各种库（Chassis, libraries and Plugins）构建出来的协议栈：

其中chassis提供了命令行和作为daemon应用所需的所有通用函数实现，包括：

命令行和配置文件（commandline and configfiles）

日志（logging）

支持daemon/service方式运行（daemon/service support）

plugin加载（plugin loading）

同时MySQL Procotol的libraries还实现了实现了各种编码解码功能：

客户端协议（client protocol）

binlog协议（binlog protocol）

myisam文件处理（myisam files）

frm文件处理（frm files）

masterinfo文件处理（masterinfo files）

# 原理

## 核心core

Network Core构建于socket处理实现的基础之上，将client connection和server connection关联到一起。

### Connection Life Cycle

connection可处于下面4种协议基本phase之一：

connect

authentification

query

disconnect

**通过对plugin功能的定制实现，可以改变network core的默认工作方式**，进而获得如下三种基本plugin功能中的一个：

1、plugin-admin只实现了listen方面的功能

2、client plugins只实现了connection方面的功能

3、plugin-proxy实现了上述两方面的功能

### Scripting

源码中所提供的plugin都实现了相应的callback功能函数，并将其暴露给了scripting层。就目前而言，scripting层的功能是由Lua语言提供的——这是一种简单、快速和便于嵌入的脚本语言。

我们通过将大部分内部数据暴露给scripting层的方式，令相应模块函数可以对传进scripting层的数据进行操作。

### Network Core Layer

MySQL Proxy的网络引擎的设计目标是同时处理数以千计的connection，并打算将其用于load-balancing和fail-over处理，所以其必须能够在同时存在很多MySQL backend server的情况下，对connection进行优雅地处理。目标锁定在了5k到10k的connection数量上。

一直到MySQL Proxy 0.7版本，MySQL Proxy使用都是纯粹的事件驱动、非阻塞网络模型。

基于事件驱动的设计决定了MySQL Proxy对idling connection仅会保存少量必要信息：即仅仅保存connection的状态，然后让其等待事件的到来。

### Threaded Scripting

通常脚本都是精巧的，并只被用于做一些简单的决策处理，而把大部分工作交由网络层处理。

在未来的0.9版本中，将会利用脚本层中所支持的多线程模型，从而达到以线程池形式呈现的多脚本线程同时运行的效果。

如此，脚本层就可以调用阻塞或者慢函数，而不需要打断其他connection的执行。

对全局plugin mutex的lift，意味着我们将必须以不同的方式对全局结构进行访问。由于大多数的访问出现在connection level上（也就是event-threading的那层），故只有对类似"proxy.global.\*"的全局结构进行访问时才需要保持同步。基于这方面的需求，我们将研究如何在各个独立的Lua-states之间相互发送数据。

## 底层实现篇（chassis）

### Configfile and Commandline Options

glib2提供了config-file解析和command-line option解析功能。其提供了将option以相同方式暴露给调用者的方法，以及从Configfile和Commandline获取option的功能。

所有option的解析过程都可以分为三步：

1、提取command-line上的basic option

--help

--version

--defaults-file

2、处理defaults-file文件

3、处理其余command-line option并覆盖defaults-file文件中的相同内容

### Plugin Interface

chassis为plugin接口调用提供了基础结构。值得注意的是，其不是专门用于MySQL的，而是可以用于任何符合其接口要求的plugin 。提供的功能包括：

1、解析plugin所在路径

2、对plugin的加载

3、对plugin进行版本检查

4、提供init和shutdown函数

5、向plugin暴露配置选项

6、基于线程的i/o

由于chassis不是仅针对于MySQL设计的，所以其可以用于加载任何种类的plugin，只要该plugin提供了符合chassis要求的init和shutdown函数。

就MySQL Proxy本身而言，一般情况下加载的plugin为：

plugin-proxy

plugin-admin

### Threaded IO

从MySQL Proxy 0.8版本开始，已经添加了基于线程的network-io以使proxy能够按照可用CPU和网卡的数量进行线性扩展。

使能network-threading功能只需要在启动proxy时加入下面的参数：

--event-threads={2 \* no-of-cores} (default: 0)

每一个event-thread都通过"event\_base\_dispatch()"进行loop，并针对network-event或者time-event执行相关函数。这些线程只具有两种状态：执行函数状态和idle状态。如果其处于idle状态，则其能够从event-queue中获取要进行等待的新event ，然后将其添加到自身的等待列表中。

connection是可以在多个event-thread之间“跳跃”的：因为只要是idle状态的event-thread就能够获取到wait-for-event request，即具体的事件，并进行等待，触发后执行相关代码。无论何时，只要当前connection需要重新等待事件（也就是之前事件所对应的操作已经完成），其就会将自身从所在线程中unregister，之后重新向全局event-queue发送wait-for-event request以获取新事件。

一直到MySQL Proxy 0.8版本，脚本代码的执行都是单线程方式：通过一个全局mutex来保护plugin的接口操作。因为connection或者是处于发送包的状态，或者是处于调用plugin函数的状态，所以网络事件将会按照并行方式被处理，仅在多个connection需要调用同一个plugin函数的时候才会无法并行。

chassis\_event\_thread\_loop()函数就是event-thread的主循环实体（其中调用event\_base\_dispatch()函数），而函数chassis\_event\_threads\_init\_thread()用于设置要监听的事件和对应的回调。

下面的描述的是一种典型控制流（不包含连接池的情况）

涉及到的实体：EventRequestQueue, MainThread, WorkerThread1, WorkerThread2;

--- [ label = "Accepting new connection "];

MainThread -> MainThread [ label = "network\_mysqld\_con\_accept()" ];

MainThread -> MainThread [ label = "network\_mysqld\_con\_handle()" ];

MainThread -> EventRequestQueue [ label = "Add wait-for-event request" ];

WorkerThread1 <- EventRequestQueue [ label = "Retrieve Event request" ];

WorkerThread1 -> WorkerThread1 [ label = "event\_base\_dispatch()" ];

...;

WorkerThread1 -> WorkerThread1 [ label = "network\_mysqld\_con\_handle()" ];

WorkerThread1 -> EventRequestQueue [ label = "Add wait-for-event request" ];

WorkerThread2 <- EventRequestQueue [ label = "Retrieve Event request" ];

WorkerThread2 -> WorkerThread2 [ label = "event\_base\_dispatch()" ];

...;

WorkerThread2 -> WorkerThread2 [ label = "network\_mysqld\_con\_handle()" ];

WorkerThread2 -> EventRequestQueue [ label = "Add wait-for-event request" ];

...;

在上面的例子中，存在两个用于处理event的工作线程（设置--event-threads=2），每个线程都有自己的event\_base。以Proxy plugin为例，首先将network\_mysqld\_con\_accept()函数设置为被监听socket的回调，当有新连接发生时被触发。该回调函数是注册在主线程的event\_base上的（同时也是全局chassis的event\_base）。在设置了连接相关结构network\_mysqld\_con后，程序将进入到状态机处理函数network\_mysqld\_con\_handle()中，此时仍然处于主线程中。

状态机将进行入起始状态：CON\_STATE\_INIT ，在当前代码实现中该状态是主线程所必进入的第一个状态。接下来MySQL Proxy要做的事，要么是和client交互，要么是和server进行交互（即或者等待socket可读，或者主动向backend server建立连接），而状态机函数network\_mysqld\_con\_handle()将设置等待处理事件（对应结构体为chassis\_event\_op\_t）。简单来说就是将event结构添加到异步队列中，具体讲，就是通过向之前创建的wakeup-pipe的写文件描述符写入一个字节，以产生一个文件描述符事件。这样就可以向所有线程通知有新事件请求需要处理。

该pipe的实现是libevent对应实现的一个翻版，其将各种事件与基于文件描述符的event-handler建立了对应关系，采用的轮询方式进行处理：

1、工作线程中的event\_base\_dispatch()函数在其监听的fd被触发前处于阻塞监听状态（在具体实现中是有定时唤醒机制的）。

2、定时器事件，信号事件等都不能直接中断event\_base\_dispatch()的运行。

3、上述事件均是通过write(pipe\_fd, ".", 1); 来触发fd-event的可读，从而通过回调来进行处理。

在文件chassis-event-thread.c中可以看到，通过pipe实现了向工作线程通知：在全局event-queue中有东东需要处理。从函数chassis\_event\_handle()可以看出，所有处于idle状态的线程都有平等机会进行事件处理，所以这些线程就能够“并行的”从全局事件队列中拉取event ，并将其添加到自身的监听事件列表中。

通过调用chassis\_event\_add()或者chassis\_event\_add\_local()函数可以将 event 添加到event-queue中。一般情况下，所有事件都由全局event\_base负责处理。只有在使用connection pool的情况下，才会强制将与特定server connection对应的events投递到特定线程，即将当前connection加入到connection pool中的那个线程。

如果event被投递到全局event\_base中，那么不同的线程都可以获取这个事件，并可以对无保护的connection pool数据结构进行修改，可能会导致竞争冒险和崩溃。令这个内部数据结构成为具有线程安全性质是0.9 release版本的工作，当前只提供了最小限度的线程安全性。

典型情况是，某个线程会从event queue中获取request信息（理论上，发送wait request的线程很可能也是处理这个request的线程），并将其添加到自身以thread-local-store方式保存的event\_base中，并在对应fd有事件触发时获得通知。

该处理过程将一直持续到当前connection被client或者server关闭，或者发生了导致的socket关闭的网络错误。此后将无法处理任何新的request。

单独一个线程就足以处理任何添加到其thread-local的event\_base上面的event。只有在一个新的blocking I/O操作发生时（一般来说也就是重新进入event\_base\_dispatch()阻塞时），event才会在不同线程间被“跳跃着”处理，除此外没有其他例外。所以理论上讲，可能会出现一个线程处理了所有活跃的socket事件，而另一个线程一直处于idle状态。

然而，由于等待网络事件的发生的状态是常态（意思就是实际处理的速度都很快），所以（从概率上讲）活跃connection在所有线程中的分布必定是很均匀的，也就会减轻单个线程处理活跃connection的压力。

值得注意的是，尽管在下面的说明中没有具体指出，主线程当前会在accept状态后参与到对后续event的处理中。这不是一个非常理想的实现方式，因为所有accept动作本身就需要在主线程中完成。但从另一方面讲，这个问题暂时也没成为实际工作中的瓶颈显现出来：

涉及到的实体：Plugin, MainThread, MainThreadEventBase, EventRequestQueue, WorkerThread1, WorkerThread1EventBase, WorkerThread2, WorkerThread2EventBase;

--- [ label = "Accepting new connection "];

Plugin -> MainThread [ label = "network\_mysqld\_con\_accept()" ];

MainThread -> MainThread [ label = "network\_mysqld\_con\_handle()" ];

MainThread -> EventRequestQueue [ label = "Add wait-for-event request" ];

WorkerThread1 <- EventRequestQueue [ label = "Retrieve Event request" ];

WorkerThread1 -> WorkerThread1EventBase [ label = "Wait for event on local event base" ];

...;

WorkerThread1EventBase >> WorkerThread1 [ label = "Process event" ];

WorkerThread1 -> EventRequestQueue [ label = "Add wait-for-event request" ];

WorkerThread2 <- EventRequestQueue [ label = "Retrieve Event request" ];

WorkerThread2 -> WorkerThread2EventBase [ label = "Wait for event on local event base" ];

...;

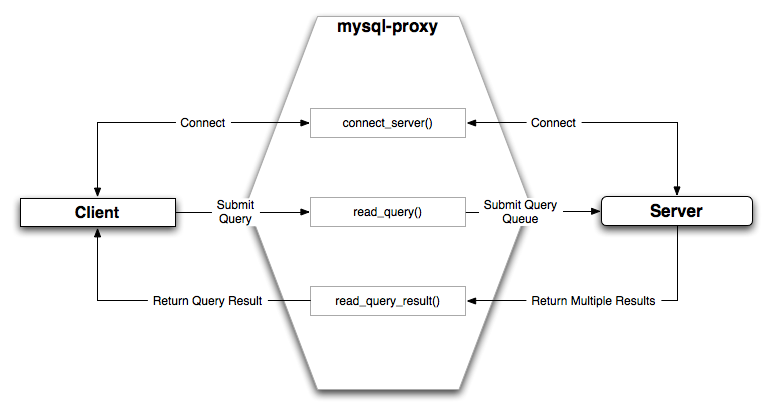
WorkerThread2EventBase >> WorkerThread2 [ label = "Process event" ];

WorkerThread2 -> EventRequestQueue [ label = "Add wait-for-event request" ];

...;

## query注入动作中的脚本序列

下图展示了一个如何使用proxy将客户端发送过来的query注入到query队列的例子。因为proxy位于客户端和MySQL服务器之间，所以经由proxy发送到服务器，以及由proxy最终返回给客户端的信息，不需要做到完全匹配或者关联。一旦客户端连接到了proxy，下图中展现的由客户端发送每一个单独的query引起的命令序列将会发生。



当客户端向proxy提交了一个query的时候，proxy内部的read\_query()函数将会被触发。该函数会添加query到query队列中。

一旦read\_query()函数执行完毕， query会按加入队列的顺序向MySQL服务器提交。

MySQL服务器会返回由每一个query所产生的结果集，每一个提交的query对应一个结果集。当结果集到达proxy后，read\_query\_result()函数会被触发，在每一次触发的时候都可以由我们自己决定哪个结果集需要发送回客户端。

例如，你可以向全局query队列中添加额外的query让服务器来进行处理。可以用于在原始query的前后添加获取统计信息的query语句的方式对原始query进行改变：

SELECT \* FROM City;

改变为一系列query：

SELECT NOW();

SELECT \* FROM City;

SELECT NOW();

你同样可以修改原始的语句，例如，向每一条要执行的语句中添加EXPLAIN以获取该语句如何被处理的信息，同样也是将原始query语句进行改变，变成下面一系列语句：

SELECT \* FROM City;

EXPLAIN SELECT \* FROM City;

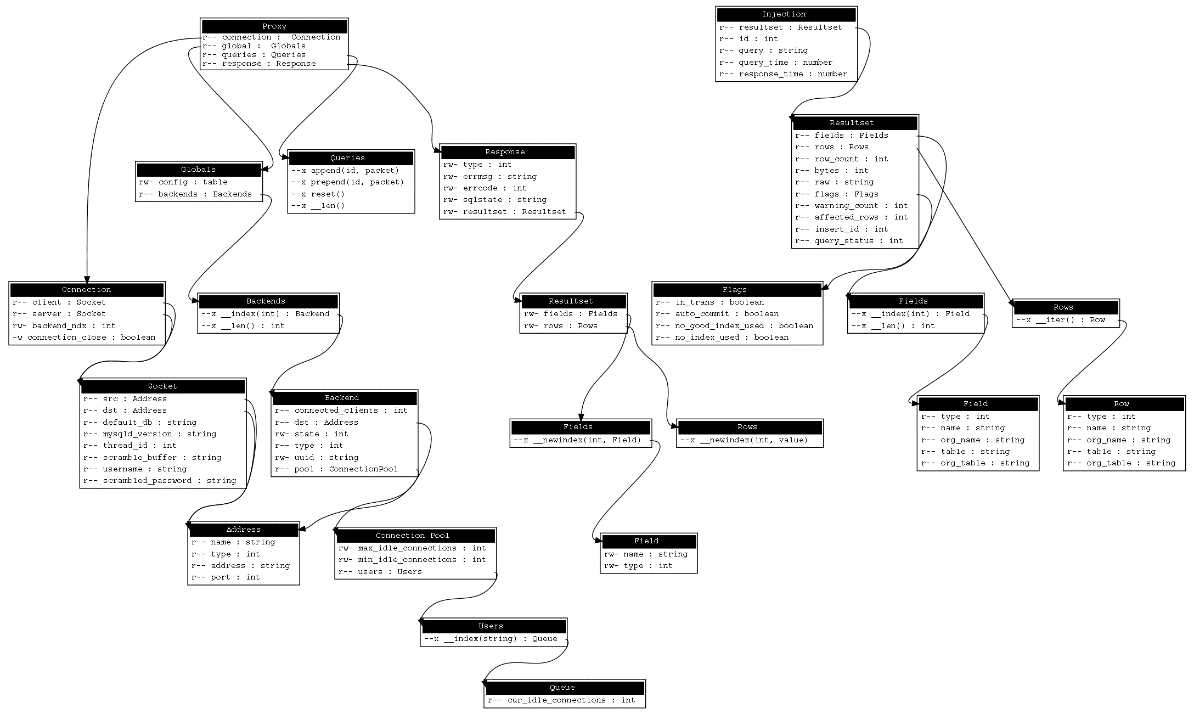
在上述两个例子中，客户端将会收到比预期更多的结果集。无论你如何控制进入的query以及返回的结果，由proxy返回的对应query数量的结果集必须要与由客户端原始发送的query数目相等。

你可以调整客户端代码以便处理由proxy返回的多个结果集，但是在大多数情况下，你会希望proxy的存在对于客户端来说是透明的。为了保证这一点，query的数量和结果集的数量应该保持一致，你可以使用MySQL Proxy的read\_query\_result()函数来提取额外的结果集信息，以便只返回客户端原始query对应的结果集。你可以通过在向query队列中添加query时，赋予每一个query一个唯一的ID的方式来保证这点，然后当read\_query\_result()函数在后续处理中收到结果集时通过匹配该ID的方式过滤掉额外的结果集。

# 脚本

你可以通过使用嵌入式Lua脚本语言对MySQL Proxy的行为进行控制，以使得其能够对发送给MySQL服务器的query和response进行操纵。

下图展现了MySQL Proxy中使用的类的总览。



在MySQL Proxy和服务器之间的主要交互功能是通过Lua脚本定义的一个或者多个函数。根据客户端与一个或者多个backend MySQL servers之间通信序列中不同的事件和操作，定义如下一些关键函数:

connect\_server(): 该函数在每次有来自客户端到MySQL Proxy的连接发生时被调用。你可以在做负载均衡（load-balancing）功能时使用该函数以拦截原始连接，然后再通过策略决定客户端将要连接的服务器。如果你没有定义一个确定方案，默认情况下，一个简单的轮询（round-robin）类型的分发方式将被采用。

read\_handshake(): 该函数在初始握手信息被服务器返回时被调用。你可以捕获返回的握手信息，然后在授权交换（authorization exchange）发生前提供额外的检查。

read\_auth(): 该函数在授权包（用户名，密码，默认数据库）被客户端提交给服务器用于鉴权时被调用。

read\_auth\_result(): 该函数在收到服务器返回的将要发送给客户端表明是否授权成功的授权包时被调用。

read\_query(): 该函数在每次客户端向服务器发送query时被调用。可以使用该函数对原始的query进行编辑和操纵，包括在原始query的前面和后面添加新的query。同样可以使用该函数直接返回信息给客户端而不经过（bypassing）服务器，该功能在打算过滤掉不期望的query的时候或者query的数量超过了已有限制的情况下很有用。

read\_query\_result(): 该函数在每次有结果从服务器返回的时候被调用，前提是你手动注入了query到query队列中。如果你没有显示地在read\_query()中注入query，则该函数不会被触发。可以使用该函数对结果集进行编辑，或者移除、过滤结果集的内容，只要该内容是由于你在read\_query()中注入的额外的query产生的结果。

下表中列出了MySQL proxy和服务器的通信函数，提供的信息，以及当函数被触发时的信息流方向。

Function Supplied Information Direction

connect\_server() None Client to Server

read\_handshake() None Server to Client

read\_auth() None Client to Server

read\_auth\_result() None Server to Client

read\_query() Query Client to Server

read\_query\_result() Query result Server to Client

默认情况下，所有的函数都会返回结果以表明当前数据是否要传递给客户端或者服务器（取决于被传输的信息的方向）。返回值可以通过显示地返回一个常量的方式进行覆盖，以表明应该发送某个特定的应答。例如，一种可能的情况是，在read\_query()中手动创建结果集，然后直接返回给客户端，中间不再涉及发送原始query到服务器的动作。

除了上述这些函数，还有一些内置结构通过提供针对元素的简化了的接口，例如query的列表和结果集的分组，提供了对MySQL Proxy如何转发query以及返回结果集的控制。

# 使用

存在许多种使用MySQL Proxy的不同方法。采用最基本用法，你可以允许MySQL Proxy将来自客户端的query透传到后端的一个服务器。如果打算让MySQL Proxy在这种模式下工作，你只需要在命令行上指定proxy打算连接的 backend server 的信息：

shell> mysql-proxy --proxy-backend-addresses=sakila:3306

如果你指定了多个后端MySQL服务器，proxy将按照轮询的方式为客户端与后端的每一个服务器建立连接。假设你有两个MySQL服务器A和B。第一个要进行连接的客户端会被连接到服务器A上，第二个会连接到服务器B上，第三个又会连接到A上，例如：

shell> mysql-proxy \

--proxy-backend-addresses=narcissus:3306 \

--proxy-backend-addresses=nostromo:3306

当你以上述方式指定了多个后端服务器，proxy将会自动检测后端服务器的可用性，当出现不可用时会进行标记。新连接会被自动的分配到当前可用的服务器上面，并且会在mysql-proxy的标准输出上报告警告信息：

network-mysqld.c.367: connect(nostromo:3306) failed: Connection refused

network-mysqld-proxy.c.2405: connecting to backend (nostromo:3306) failed, marking it as down for ...

通过Lua脚本可以提供更细粒度的行为控制，包括对连接的控制、对分发的控制，以及对query和结果集的处理控制。当使用Lua脚本时，你必须在命令行上通过选项--proxy-lua-script指定脚本的名字：

shell> mysql-proxy --proxy-lua-script=mc.lua --proxy-backend-addresses=sakila:3306

当你指定了一个脚本，该脚本直到一个连接被建立时才会被读取。这意味着脚本自身的错误要到其真正被执行的时候才会被触发。脚本错误不会对向后端MySQL服务器分发query的行为产生影响。

因为脚本直到连接建立时才会被读取，你可以在proxy处于运行状态的情况下对Lua脚本的内容进行修改，而修改后的内容会在下一次新连接建立时自动被使用。这使得MySQL Proxy可以再无需重启的前提下使得行为发生改变。