# 概述

参考：

<https://mp.weixin.qq.com/s/lBt5u3BiJueXsShJpxG5Yw>

<https://mp.weixin.qq.com/s/h12D0HS97Rjqeq8K1o5Lng>

<https://mp.weixin.qq.com/s/2JpPMauLq5aPkJTEMm6NuA>

<https://mp.weixin.qq.com/s/VCbJFFcJgPBDXdRTdFNtUg>

MySQL默认的连接控制方式采用的是每个连接使用一个**线程**执行客户端的请求。

MySQL的线程池是包含在企业版里面的服务器插件。使用线程池的目的是为了改善大量并发连接所带来的性能下降。在大量并发连接的工作负载下，使用线程池可以解决无法利用CPU缓存、上下文切换开销过大以及资源争用等问题。

注：**开源版本没有线程池，只有企业版才有线程池**。

使用线程池的理由有两个：

1、减少线程重复创建与销毁部分的开销，提高性能

线程池技术通过预先创建一定数量的线程，在监听到有新的请求时，线程池直接从现有的线程中分配一个线程来提供服务，服务结束后这个线程不会直接销毁，而是又去处理其他的请求。这样就**避免了线程和内存对象频繁创建和销毁，减少了上下文切换，提高了资源利用率，从而在一定程度上提高了系统的性能和稳定性**。

2、对系统起到保护作用

线程池技术限制了并发线程数，相当于限制了MySQL的runing线程数，无论系统目前有多少连接或者请求，超过最大设置的线程数的都需要排队，让系统保持高性能水平，从而防止DB出现雪崩，对底层DB起到保护作用。

可能有人会问，使用连接池能否也达到类似的效果？

也许有的DBA会把线程池和连接池混淆，但其实两者是有很大区别的：**连接池一般在客户端设置，而线程池是在DB服务器上配置；另外连接池可以起到避免了连接频繁创建和销毁，但是无法控制MySQL活动线程数的目标，在高并发场景下，无法起到保护DB的作用**。比较好的方式是将连接池和线程池结合起来使用。

# 模块

## THD

## Protocol

参考：<http://hutaow.com/blog/2013/11/06/mysql-protocol-analysis/>

### 交互过程

MySQL客户端与服务器的交互主要分为两个阶段：握手认证阶段和命令执行阶段。

#### 握手认证阶段

握手认证阶段为客户端与服务器建立连接后进行，交互过程如下：

服务器 -> 客户端：握手初始化消息

客户端 -> 服务器：登陆认证消息

服务器 -> 客户端：认证结果消息

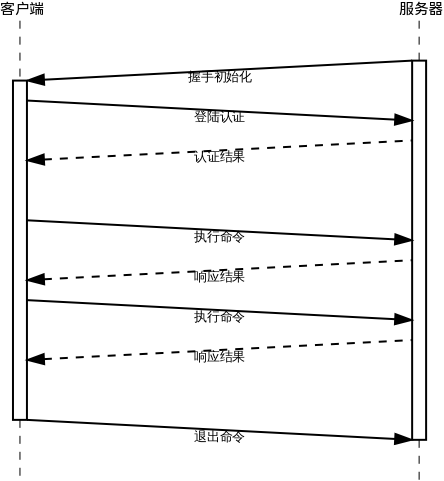
#### 命令执行阶段

客户端认证成功后，会进入命令执行阶段，交互过程如下：

客户端 -> 服务器：执行命令消息

服务器 -> 客户端：命令执行结果

MySQL客户端与服务器的完整交互过程如下：



MySQL客户端与服务器交互示意图

### 基本类型

### 报文结构

### 报文类型

## NET

## VIO

Vio封装了所有对socket的操作。根据不同的连接类型（TCP/IP，Socket，Name Pipe，SSL，SHARED MEMORY），相关函数指针在vio\_init函数中定义。

VIO包含了多种网络连接类型，例如：

NO\_VIO\_TYPE：未知的网络类型

VIO\_TYPE\_TCPIP：TCP/IP连接

VIO\_TYPE\_SOCKET：Unix系统的Socket连接

VIO\_TYPE\_NAMEPIPE：Windows系统的命名管道

VIO\_TYPE\_SSL：SSL连接

VIO\_TYPE\_SHARED\_MEMORY：Windows下的共享内存

VIO\_TYPE\_PLUGIN：通过插件实现的网络类型，不属于上述任何一种

针对以上的不同类型的网络连接，MySQL提供了针对于不同平台，不同连接类型的网络操作方法。

VIO对着这些方法进行了封装，向上提供统一的接口。VIO只是一个通用类，具体的网络方法实现在以下几个文件中：

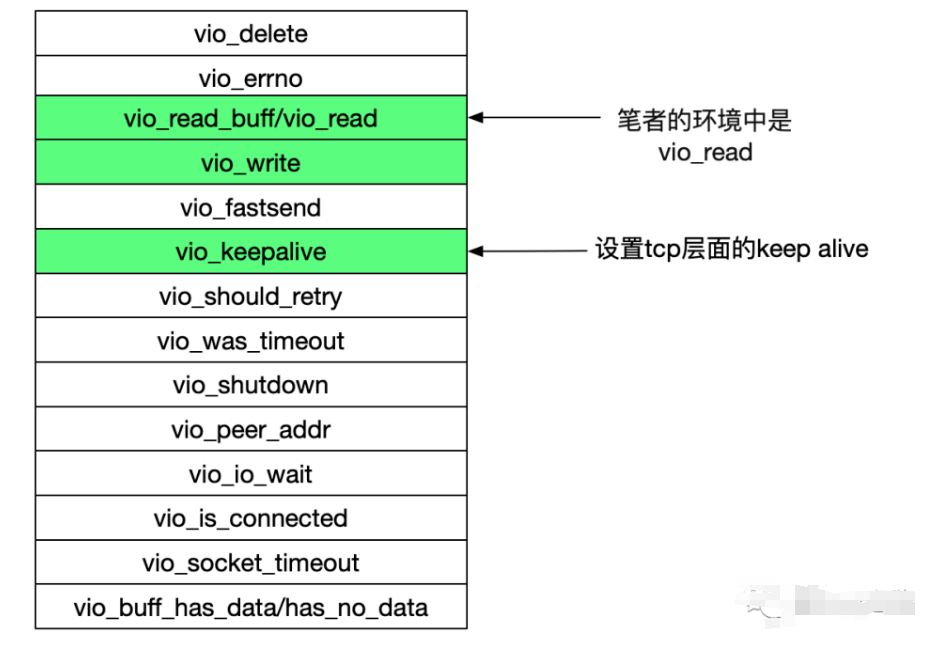
viosocket.c：提供了SOCKET相关的操作方法。

viossl.c：提供了SSL相关的操作方法。

viopipe.c：提供了Windows平台下命名管道的操作方法。

vioshm.c：提供了Windows平台下共享内存的操作方法。

如上图代码中，每新建一个连接，都随之新建一个vio(mysql\_socket\_vio\_new->vio\_init)，在vio\_init的过程中，初始化了一堆回掉函数,如下图所示：



我们关注点在vio\_read和vio\_write上，如上面代码所示，在笔者所处机器的环境下将MySQL连接的socket设置成了非阻塞模式(O\_NONBLOCK)模式。所以在vio的代码里面采用了nonblock代码的编写模式，如下面源码所示：

### Socket

#### vio\_delete && vio\_shutdown

以上函数为删除和关闭网络连接， vio\_delete 与 vio\_shutdown 的区别为， vio\_delete在vio\_shutdown 的基础上，对vio->read\_buffer和vio对象进行了释放。shutdown动作首先调用了系统函数shutdown套接字，然后再调用closesocket关闭套接字。

这些对系统函数的调用封装在mysql\_socket.h的inline函数中，并且对 PSI\_SOCKET\_INTERFACE进行了特殊处理。

#### vio\_errno && vio\_should\_retry

vio\_errno简单的返回了socket的全局错误码变量errno； vio\_should\_retry返回上次的errno错误码是否是EINTR。

#### vio\_socket\_io\_wait

vio\_socket\_io\_wait是等待套接字可读。网络数据的读取有阻塞模式和非阻塞模式，如果当前的套接字设置的是阻塞模式，那么读取函数会阻塞在mysql\_socket\_recv函数处，直到读取到了数据或者收到了一个错误。如果套接字设置的是非阻塞模式，并且给出了一个超时时间，那么调用mysq\_socket\_recv的时候，如果没有可读的数据，那么就会立即返回错误。这时，需要等待套接字可读，然后再次读取数据。该函数使用poll等待套接字上的事件，等到事件发生或者超时后返回。

#### vio\_read && vio\_read\_buffer

vio\_read为读取数据的方法，该方法根据flags会有不同的处理方式。如果用户开启了BUFFER\_READ，则会调用vio\_read\_buff。如果没有开启，则调用vio\_read。

vio\_read使用mysql\_socket\_recv函数读取数据，该函数封装mysql\_socket.h 的inline函数中，并对PSI\_SOCKET\_INTERFACE进行了特殊处理。如果用户设置了非阻塞模式，会先使用vio\_socket\_io\_wait等待fd可读，然后重新读取数据。

除此之外，MySQL还提供了buffer\_read方式，该方法针对许多的短数据请求做了优化，能够减少系统调用次数。具体的方法如下：

1、VIO初始化的时候会申请一段空间，叫做read\_buffer。

2、当上层程序从VIO读取数据的时候，会先看read\_buffer中是否还存在没有读取的数据，如果有，则先将缓存的数据读取出来。（如果缓存中的数据长度少于请求的数据长度，也不会重新从socket中读取数据）

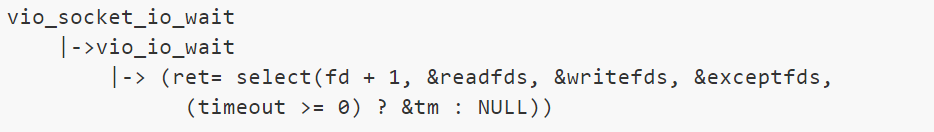
3、如果缓存中没有数据，并且请求的数据长度小于2048。则会将socket中的所有可读的数据全部读取出来（很有可能读取的长度大于请求的数据长度，这样可以将多次读取降低为一次读取，减少系统调用次数），并保存到read\_buffer中。

4、以上都不匹配的情况下，直接从socket中读取数据。

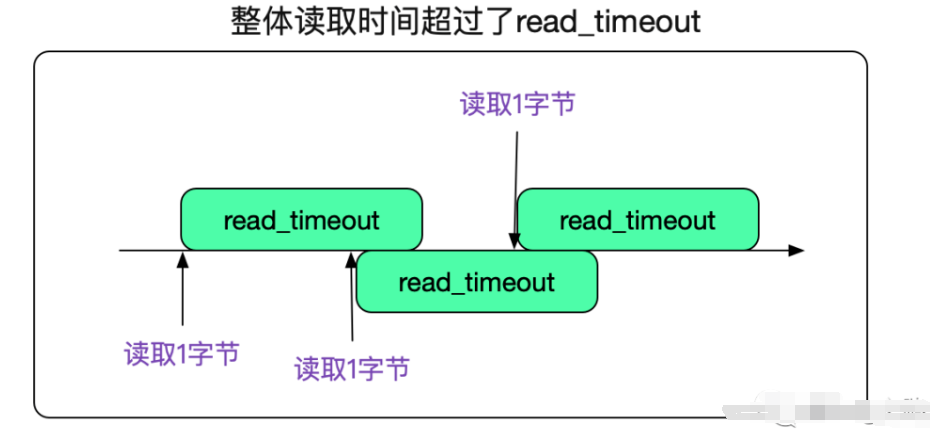
具体分析vio\_read：



即通过while循环去读取socket中的数据，如果读取为空，则通过vio\_socket\_io\_wait去等待(借助于select的超时机制)，其源码如下所示：



在jdk源码中看到java的connection time out也是通过这,select(…wait\_time)的方式去实现连接超时的。  
 由上述源码可以看出，这个mysql的read\_timeout是针对每次socket recv(而不是整个packet的)，所以可能出现超过read\_timeout MySQL仍旧不会报错的情况，如下图所示：



#### vio\_write

与vio\_read类似，通过调用mysql\_sicket\_send函数来发送数据，如果是非阻塞模式情况下遇到了发送失败，则会使用vio\_socket\_io\_waite等到fd可写后再次写入。

vio\_write实现模式和vio\_read一致，也是通过select来实现超时时间的判定,如下面源码所示：



#### vio\_fastsent && vio\_keepalive

通过setsocketopt来对套接字进行设置TCP\_NODELAY和SO\_KEEPALIVE来达到目的。

#### vio\_peer\_addr

vio\_peer\_addr提供了查询client的ip和端口号的功能。IPV6功能没有开启，返回IPV4的IP，if the client socket is associated with an IPv4-compatible or IPv4-mapped IPv6 address的情况下叶返回IPV4的IP，其他情况返回IPV6的IP。

该功能使用gerpeername系统函数读取addr结构，用来获取对端的连接地址和端口。

#### vio\_is\_connected

vio\_is\_connect用来判断当前的连接是否处于连接状态。

首先通过检查当前连接是否有数据可读。

其次读取数据，看是否读出了一个EOF或者其他错误。

#### vio\_buffer\_has\_data && has\_no\_data

检查当前raed\_buffer中是否存在未读数据。

如果read\_buffer启用，则会使用vio\_buffer\_has\_data检查read\_buffer中是否存在未读数据。

如果read\_buffer未启用，则会使用has\_no\_data直接返回FALSE。

### SSL

与普通的Socket连接相比，使用SSL的函数处理大致相同（viossl.c），除了一下几个函数：

#### vio\_reset && vio\_end

VIO使用SSL功能需要通过vio\_reset来添加ssl\_arg参数。（为什么要用reset来添加SSL上下文？）

当使用SSL的情况下进行vio\_end操作时，需要clean ssl相关资源。

#### vio\_ssl\_read && vio\_ssl\_write

这两个方法与vio\_read，vio\_write类似，在socket中是直接从套接字中读取和写入数据，当使用了SSL后，则使用SSL提供的read和write方法，进行数据的读取和写入。

#### SSL的模块化加载

MySQL社区版使用了yassl组件来提供SSL功能。该组件在extra/yassl下。

yassl通过prefix\_ssl.h和 prefix\_crypto.h 这两个头文件来定义提供的所有SSL操作方法，这两个头文件中全部是宏定义（#define对外提供的函数名 内部的函数名）。yassl提供了一个脚本（generate\_prefix\_files.pl）来维护这两个头文件，以适应组件升级（内部函数名的变化）。（为何MySQL使用这些宏的c文件中并没有引入这几个头文件也可以使用？）

MySQL对SSL的实现有多种方式：

1、不使用SSL，则不会使用内置的yassl

2、使用内置的SSL ，使用内置的yassl，静态连接yassl库

3、用户指定的SSL组件，仅支持openssl，cmake -DWITH\_SSL=

4、使用系统中的SSL组件，MySQL会从系统库中查找SSL，cmake -

DWITH\_SSL=”system”

# 流程

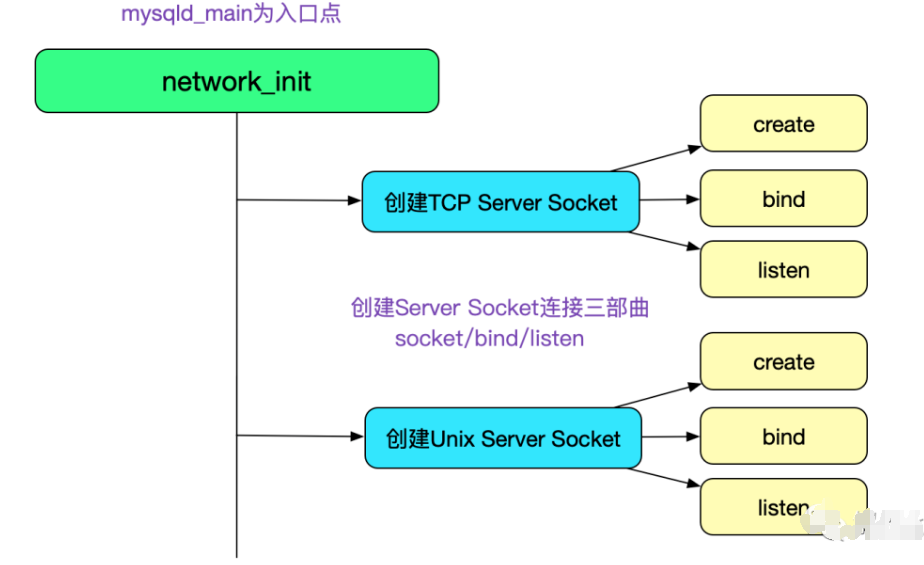
## 创建会话

在MySQL5.7中对会话协议层的代码进行了大量的重构以优化性能，并使得代码更加可读。以下这幅图大概展示了几个相关的类关系（未包含诸如windows平台的相关类）：

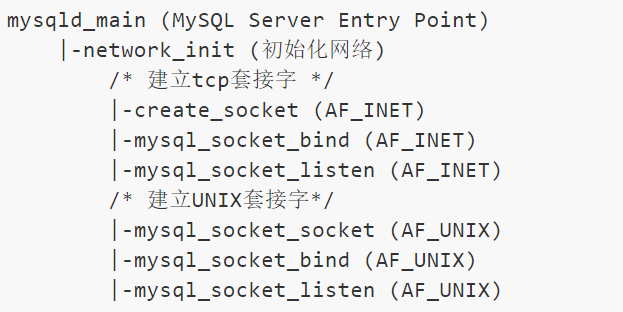
图示

中度可信度描述已自动生成

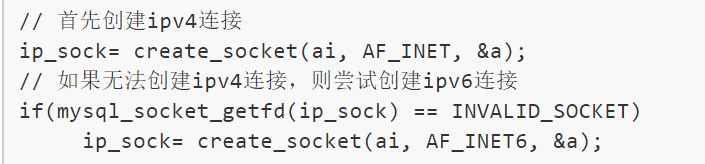
首先就需要找到其入口点，mysqld的入口点为mysqld\_main,跳过了各种配置文件的加载之后，来到了network\_init初始化网络环节，如下图所示：



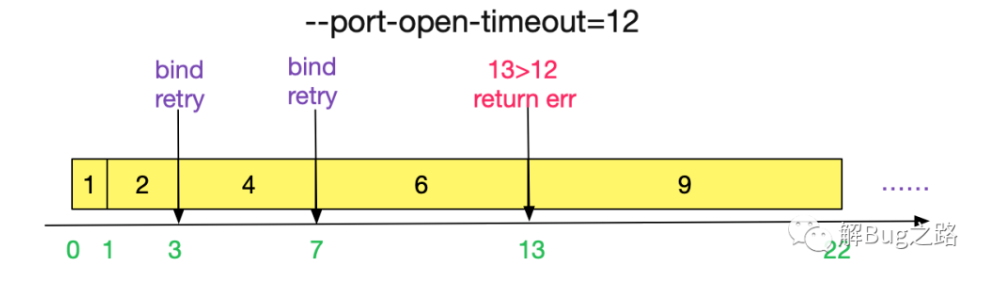
下面是其调用栈：



值得注意的是，在tcp socket的初始化过程中，考虑到了ipv4/v6的两种情况:



如果我们以很快的速度stop/start mysql，会出现上一个mysql的listen port没有被release导致无法当前mysql的socket无法bind的情况（这也是我们平时遇到的关闭后立即重启，然后连接mysql会报错稍等片刻可以连接成功的原因），在此种情况下mysql会循环等待，其每次等待时间为当前重试次数retry \* retry/3 +1秒，一直到设置的—port-open-timeout(默认为0)为止，如下图所示：



创建用户线程堆栈是从主线程开始的，监听客户端请求并创建处理线程：

mysqld\_main

|-->connection\_event\_loop

|-->listen\_for\_connection\_event

//根据不同的监听模式，去监听新请求, 当获取到一个新的监听请求时，会创建一个Channel\_info类，用 来存储用户的socket信息

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

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

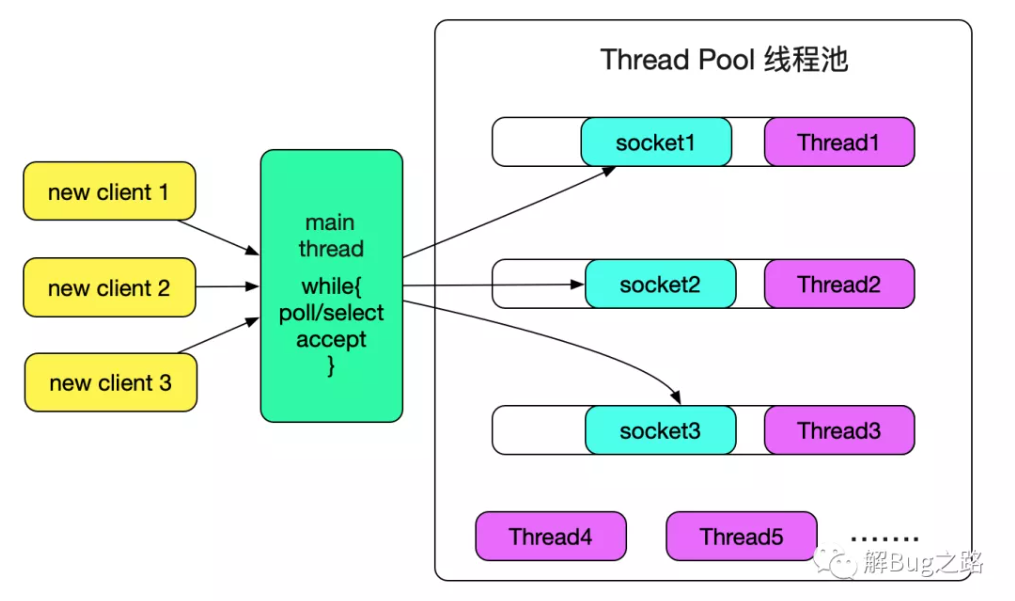
//我们通常用的one thread one connection对应的类为Per\_thread\_connection\_handler

|-->创建用户线程，线程函数为handle\_connection

在MySQL5.7里一个重大的优化，如上所述，就是关于用户会话的thd, net, vio等信息的初始化都不是在主线程进行的，而是创建用户线程后，由用户线程自己来完成。通过这种方式，主线程可以更高效的接受新的连接请求，从而优化了在短连接场景下的性能。

## 循环处理新建连接

通过handle\_connections\_sockets处理MySQL的新建连接循环，根据操作系统的配置通过poll/select处理循环(非epoll，这样可移植性较高，且mysql瓶颈不在网络上)。  
 MySQL通过线程池的模式处理连接(一个连接对应一个线程，连接关闭后将线程归还到池中)，如下图所示：



对应的调用栈如下所示：

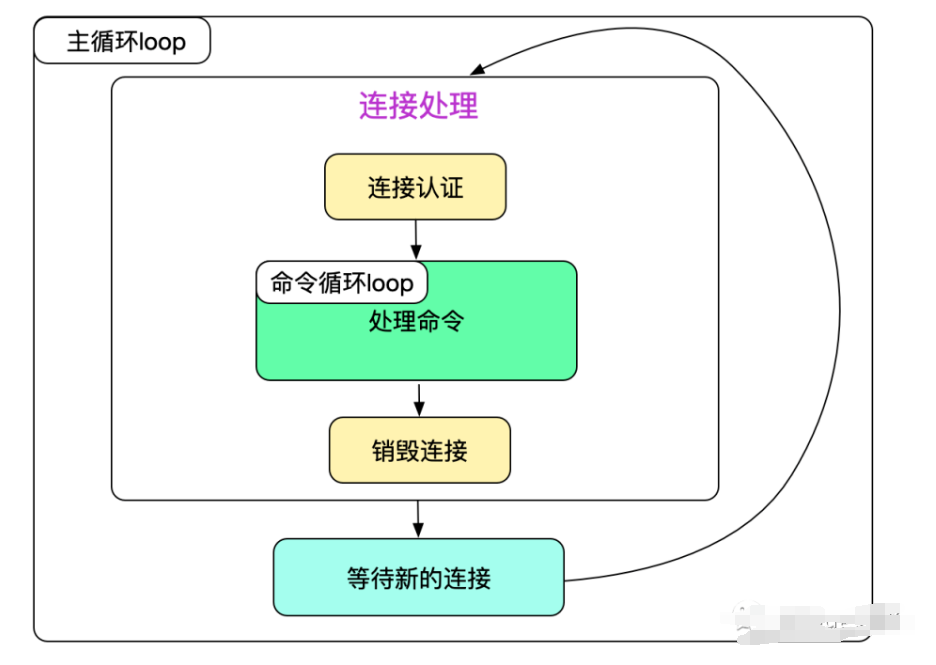


## 连接处理线程

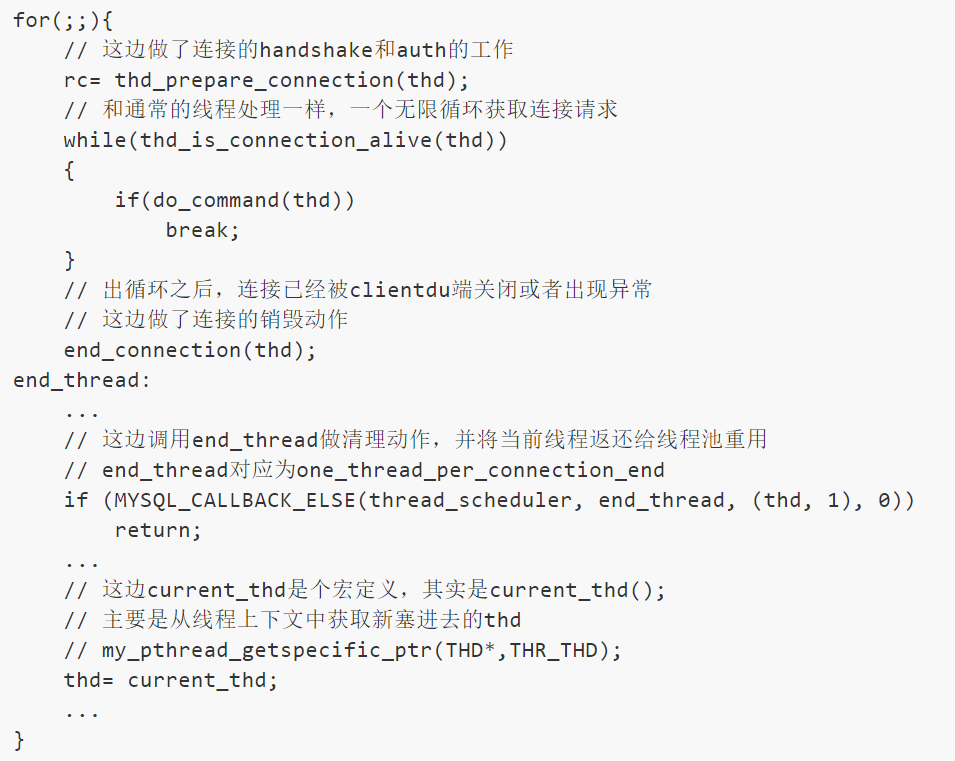
从上面的代码：



可以发现，MySQL每个线程的处理函数为handle\_one\_connection，其过程如下图所示：



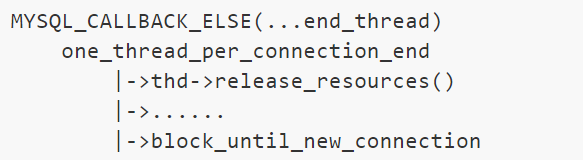
代码如下所示：



mysql的每个woker线程通过无限循环去处理请求。

## 线程的归还过程

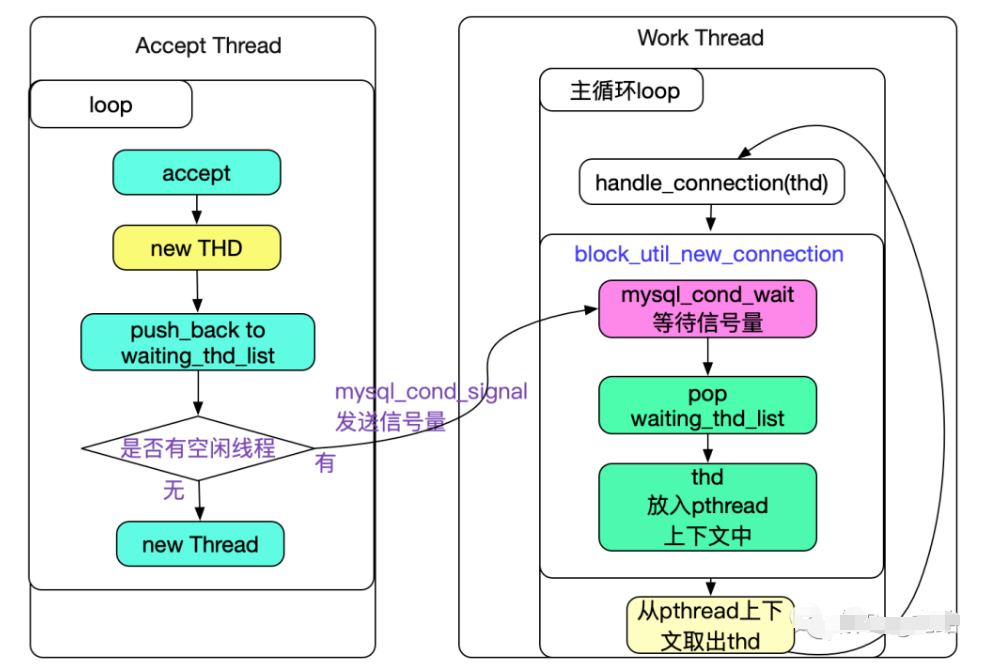
MySQL通过调用one\_thread\_per\_connection\_end(即上面的end\_thread)去归还连接。



线程在新连接尚未到来之前，等待在信号量上(下面代码是C/C++ mutex condition的标准使用模式)：



整个过程如下图所示：



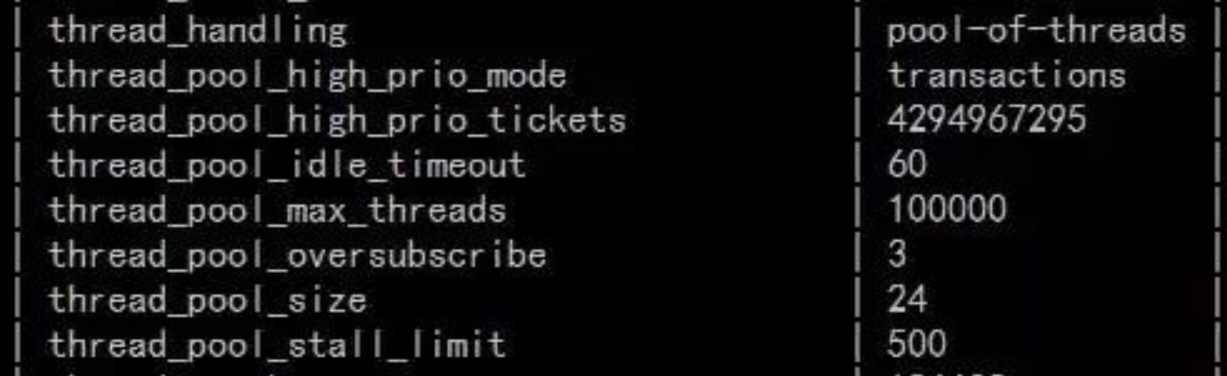
由于MySQL的调用栈比较深，所以将thd放入线程上下文中能够有效的在调用栈中减少传递参数的数量。

## 总结

MySQL的网络IO模型采用了经典的线程池技术，虽然性能上不及reactor模型，但好在其瓶颈并不在网络IO上，采用这种方法无疑可以节省大量的精力去专注于处理sql等其它方面的优化。

# 参数

关于线程池参数不多，使用show variables like 'thread%'可以看到如下图的参数，下面就一个一个来解析：



## thread\_handling

该参数是配置线程模型，默认情况是one-thread-per-connection，即不启用线程池；将该参数设置为pool-of-threads即启用了线程池。

**thread\_pool\_size**

该参数是设置**线程池的Group的数量，默认为系统CPU的个数，充分利用CPU资源**。

线程池是由一定数量的**线程组**（默认为16个通过thread\_pool\_size进行配置）构成，每个线程组管理一组客户端连接，最大连接数为4096。连接创建之后会以**轮询**的方式分配给线程组。连接池打破了每个连接与线程一一对应的关系，这一点与MySQL默认的线程控制方式不同，默认方式将一个线程与一个连接相关联，以便给定的线程从其连接执行所有的语句。

默认情况下，线程池试图确保每个组中每次最多执行一个线程，但有时为了获得最佳性能，允许临时执行多个线程。**每组里面有一个监听线程，负责监听分配给该组的连接**。线程会选择立即执行或稍后执行连接里面的语句，如果语句是唯一接收到的，并且当前没有排队或正在执行的语句，该语句就会立即执行。其它情况则会选择稍后执行。当该语句被判断为立即执行时，监听线程负责执行该语句，如果能够快速完成执行，该线程会返回监听状态，如果执行语句时间过长产生停滞，线程组会开启一个新的监听线程。线程池插件使用一个后台线程监控线程组状态，以确保线程组不会因为停滞的语句阻塞线程组。

## thread\_pool\_oversubscribe

该参数设置group中的最大线程数，每个group的最大线程数为thread\_pool\_oversubscribe+1，注意listener线程不包含在内。

## thread\_pool\_high\_prio\_mode

高优先级队列的控制参数，有三个值（transactions/statements/none），默认是transactions，三个值的含义如下：

transactions：对于已经启动事务的语句放到高优先级队列中，不过还取决于后面的thread\_pool\_high\_prio\_tickets参数。

statements：这个模式所有的语句都会放到高优先级队列中，不会使用到低优先级队列。

none：这个模式不使用高优先级队列。

## thread\_pool\_high\_prio\_tickets

该参数控制每个连接最多语序多少次被放入高优先级队列中，默认为4294967295，注意这个参数只有在thread\_pool\_high\_prio\_mode为transactions的时候才有效果。

## thread\_pool\_idle\_timeout

worker线程最大空闲时间，默认为60秒，超过限制后会退出。

## thread\_pool\_max\_threads

该参数用来限制线程池最大的线程数，超过该限制后将无法再创建更多的线程，默认为100000。

## thread\_pool\_stall\_limit

可以通过thread\_pool\_stall\_limit 配置等待值时长，短等待值允许线程更快启动，也有助于避免死锁情况。长时间等待值对于长时间运行的工作负载非常有用，可以避免在当前语句执行时启动太多新语句。

## thread\_pool\_max\_active\_query\_threads

通过thread\_pool\_max\_active\_query\_threads设置运行的最大线程，如果该值不为0，则该数值为允许运行的最大线程数量，设置为0使用默认最大值。

线程池侧重于限制短时间运行语句的并发数量。在执行语句达到待值时长之前，它会阻止其他语句开始执行。如果语句执行超过了待值时长，允许其继续执行，但不再阻止其他语句启动。通过这种方式，线程池尝试确保每个线程组中永远不会有超过一个的短时间运行语句，但可能有多个长时间运行的语句。

如果遇到磁盘I/O操作或用户级锁(行锁或表锁)，语句就会被阻塞，将导致线程组无法使用。线程池的回调功能，可以确保线程池立即启动该组中的新线程来执行另一条语句。当一个被阻塞的线程返回时，线程池允许它立即重新启动。

线程池包含两个队列，高优先级队列和低优先级队列。当前正在执行的语句及该事务后续关联的语句将进入高优先级队列，其它语句进入低优先级队列。

此外，线程池重用活跃的线程，以更好地利用CPU缓存。这是一个对性能有很大影响的调整。

## max\_connections

理论上，可能出现的最大线程数是 max\_connections和thread\_pool\_size的总和。当所有连接都处于执行模式，并且每个组都创建了一个额外的线程来监听，可能会发生这种情况。

# 问题