# 概述

当有大量请求并发访问时，一定伴随着资源的不断申请和释放，导致资源的利用率低，降低服务质量。线程池通过预先创建一定数量的线程，当有请求到达时，线程池分配一个线程提供服务，请求结束后，该线程又去服务其他请求，通过这种方式，避免了线程和内存对象的频繁申请和释放，降低了服务端的并发度，减少了上下文切换和资源的竞争，提高资源利用效率。

所有服务的线程池本质都是为了提高资源利用效率，并且实现方式也大致相同。

在MySQL5.6之前，MySQL处理连接的方式是One-Connection-Per-Thread，即对于每一个数据库连接，MySQL-Server都会创建一个独立的线程服务，请求结束后，销毁线程。再来一个连接请求，则再创建一个连接，结束后再进行销毁。这种方式在高并发的情况下，会导致线程的频繁创建和释放。当然，可以通过thread-cache将线程缓存起来，以供下次使用，避免频繁地创建和释放的问题，但是无法解决高连接数的问题。One-Connection-Per-Thread方式随着连接数暴增，导致需要创建同样多的服务线程，高并发线程意味着高内存消耗，更多的上下文切换（CPU cache命中率降低）以及更多的资源竞争，导致服务出现抖动。相对于One-Connection-Per-Thread方式，一个线程对应一个连接，Thread-Pool实现方式中，线程处理的最小单元不再是连接而是statement（语句），一个线程可以处理多个连接的请求。这样，在保证充分利用硬件资源情况下（合理设置线程池大小），可以避免瞬间连接数暴增导致的服务器抖动。

# MQ与线程池

线程池多用于单机，MQ可以跨主机，用于分布式环境。

# 调度方式

MySQL-Server同时支持3种连接管理方式，包括No-Thread，One-Thread-Per-Connection和Pool-Threads。No-Threads表示处理连接使用主线程处理，不额外创建线程，这种方式主要用于调试；One-Thread-Per-Connection是线程池出现以前最常用的方式，为每一个连接创建一个线程服务；Pool-Thread是线程池方式。

MySQL-Server通过一组函数指针来同时支持3中连接管理方式，对于特定的方式，将函数指针设置成特定的回调函数，连接管理方式通过thread\_handing参数控制，代码如下：

if (thread\_handling <= SCHEDULER\_ONE\_THREAD\_PER\_CONNECTION)

one\_thread\_per\_connection\_scheduler(thread\_scheduler,

&max\_connections,

&connection\_count);

else if (thread\_handling == SCHEDULER\_NO\_THREADS)

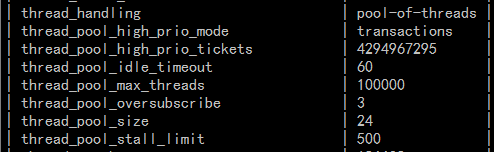
one\_thread\_scheduler(thread\_scheduler);

else

pool\_of\_threads\_scheduler(thread\_scheduler, &max\_connections,&connection\_count);

# 线程池参数

关于线程池参数使用show variables like 'thread%'可以看到如下图的参数：



## thread\_handling

thread\_handling:表示线程池模型，默认情况是one-thread-per-connection，也就是不启用线程池。将该参数设置为pool-of-threads即启用了线程池。

## thread\_pool\_size

thread\_pool\_size:表示线程池的group个数，一般设置为当前CPU核心数目。理想情况下，一个group一个活跃的工作线程，达到充分利用CPU的目的。

## thread\_pool\_stall\_limit

thread\_pool\_stall\_limit:用于timer线程定期检查group是否“停滞”，参数表示检测的间隔，默认为500ms。

## 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\_idle\_timeout

thread\_pool\_idle\_timeout:当一个worker空闲一段时间后会自动退出，保证线程池中的工作线程在满足请求的情况下，保持比较低的水平。

worker线程最大空闲时间，默认为60秒，超过限制后会退出。

## thread\_pool\_oversubscribe

thread\_pool\_oversubscribe: 该参数设置group中的最大线程数（控制CPU核心上“超频”的线程数），每个group的最大线程数为thread\_pool\_oversubscribe+1，注意这个参数设置值不含listen线程计数。

## thread\_pool\_high\_prio\_mode

threadpool\_high\_prio\_mode:表示优先队列的模式。

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。

# 线程池实现

## 架构图



每一个绿色的方框代表一个group，group数目由thread\_pool\_size参数决定。每个group包含一个优先队列和普通队列，包含一个listener线程和若干个工作线程，listener线程和worker线程可以动态转换，worker线程数目由工作负载决定，同时受到thread\_pool\_oversubscribe设置影响。此外，整个线程池有一个timer线程监控group，防止group“停滞”。

## 组成

从架构图中可以看到Thread Pool由一个Timer线程和多个Thread Group组成，而每个Thread Group又由两个队列、一个listener线程和多个worker线程构成。下面分别来介绍每个各个部分的作用：

### 队列

（高优先级队列和低优先级队列）

用来存放待执行的IO任务，分为高优先级队列和低优先级队列，高优先级队列的任务会优先被处理。    什么任务会放在高优先级队列呢？    事务中的语句会放到高优先级队列中，比如一个事务中有两个update的SQL，有1个已经执行，那么另外一个update的任务就会放在高优先级中。这里需要注意，如果是非事务引擎，或者开启了autocommit的事务引擎，都会放到低优先级队列中。    还有一种情况会将任务放到高优先级队列中，如果语句在低优先级队列停留太久，该语句也会移到高优先级队列中，防止饿死的问题。Xxxxxxxxx

### listener线程

listener线程监听该线程group的语句，并确定是自己转变成worker线程立即执行对应的语句还是放到队列中，判断的标准是看队列中是否有待执行的语句。如果队列中待执行的语句数量为0，而listener线程转换成worker线程，并立即执行对应的语句。如果队列中待执行的语句数量不为0，则认为任务比较多，将语句放入队列中，让其他的线程来处理。这里的机制是为了减少线程的创建，因为一般SQL执行都非常快。

### worker线程

worker线程是真正干活的线程

### Timer线程

Timer线程是用来周期性检查group是否处于处于阻塞状态，当出现阻塞的时候，会通过唤醒线程或者新建线程来解决。具体的检测方法为，通过queue\_event\_count的值和IO任务队列是否为空来判断线程组是否为阻塞状态。每次worker线程检查队列中任务的时候，queue\_event\_count会+1，每次Timer检查完group是否阻塞的时候会将queue\_event\_count清0，如果检查的时候任务队列不为空，而queue\_event\_count为0，则说明任务队列没有被正常处理，此时该group出现了阻塞，Timer线程会唤醒worker线程或者新建一个wokrer线程来处理队列中的任务，防止group长时间被阻塞。

## 连接管理流程

1、通过poll监听mysql端口的连接请求；

2、收到连接后，调用accept接口，创建通信socket；

3、初始化thd实例，vio对象等；

4、根据thread\_handling方式设置，初始化thd实例的scheduler函数指针；

5、调用scheduler特定的add\_connection函数新建连接。

## 关键接口

### tp\_add\_connection

[处理新连接]

1) 创建一个connection对象

2) 根据thread\_id%group\_count确定connection分配到哪个group

3) 将connection放进对应group的队列

4) 如果当前活跃线程数为0，则创建一个工作线程

### worker\_main

[工作线程]

1) 调用get\_event获取请求

2) 如果存在请求，则调用handle\_event进行处理

3) 否则，表示队列中已经没有请求，退出结束。

### get\_event

[获取请求]

1) 获取一个连接请求

2) 如果存在，则立即返回，结束

3) 若此时group内没有listener，则线程转换为listener线程，阻塞等待

4) 若存在listener，则将线程加入等待队列头部

5) 线程休眠指定的时间(thread\_pool\_idle\_timeout)

6) 如果依然没有被唤醒，是超时，则线程结束，结束退出

7) 否则，表示队列里有连接请求到来，跳转1

备注：获取连接请求前，会判断当前的活跃线程数是否超过了thread\_pool\_oversubscribe+1，若超过了，则将线程进入休眠状态。

### handle\_event

[处理请求]

1) 判断连接是否进行登录验证，若没有，则进行登录验证

2) 关联thd实例信息

3) 获取网络数据包，分析请求

4) 调用do\_command函数循环处理请求

5) 获取thd实例的套接字句柄，判断句柄是否在epoll的监听列表中

6) 若没有，调用epoll\_ctl进行关联

7) 结束

### Listener

[监听线程]

1) 调用epoll\_wait进行对group关联的套接字监听，阻塞等待

2) 若请求到来，从阻塞中恢复

3) 根据连接的优先级别，确定是放入普通队列还是优先队列

4) 判断队列中任务是否为空

5) 若队列为空，则listener转换为worker线程

6) 若group内没有活跃线程，则唤醒一个线程

备注：这里epoll\_wait监听group内所有连接的套接字，然后将监听到的连接

请求push到队列，worker线程从队列中获取任务，然后执行。

### timer\_thread

[监控线程]

1) 若没有listener线程，并且最近没有io\_event事件

2) 则创建一个唤醒或创建一个工作线程

3) 若group最近一段时间没有处理请求，并且队列里面有请求，则

4) 表示group已经stall，则唤醒或创建线程

5）检查是否有连接超时

备注：timer线程通过调用check\_stall判断group是否处于stall状态，通过调用timeout\_check检查客户端连接是否超时。

### tp\_wait\_begin

[进入等待状态流程]

1) active\_thread\_count减1，waiting\_thread\_count加1

2）设置connection->waiting= true

3) 若活跃线程数为0，并且任务队列不为空，或者没有监听线程，则

4) 唤醒或创建一个线程

### tp\_wait\_end

[结束等待状态流程]

1) 设置connection的waiting状态为false

2) active\_thread\_count加1，waiting\_thread\_count减1

备注：

1) waiting\_threads这个list里面的线程是空闲线程，并非等待线程，所谓空闲线程是随时可以处理任务的线程，而等待线程则是因为等待锁，或等待io操作等无法处理任务的线程。

2) tp\_wait\_begin和tp\_wait\_end的主要作用是由于汇报状态，即使更新active\_thread\_count和waiting\_thread\_count的信息。

### tp\_init/tp\_end

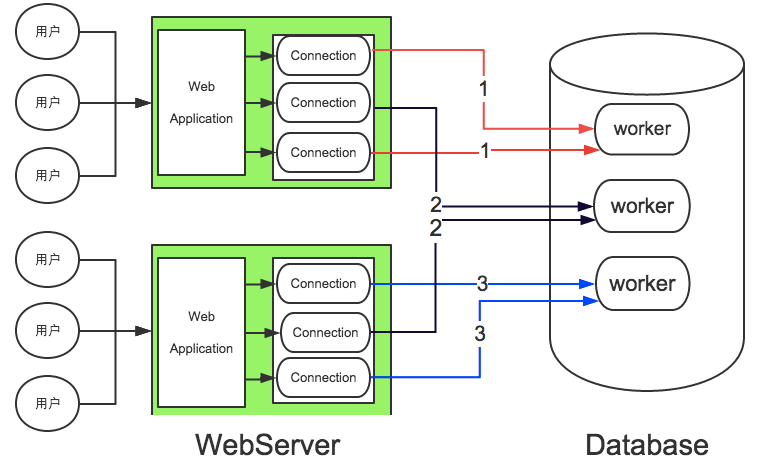
分别调用thread\_group\_init和thread\_group\_close来初始化和销毁线程池

# 线程池与连接池

连接池通常实现在Client端，是指应用(客户端)创建预先创建一定的连接，利用这些连接服务于客户端所有的DB请求。如果某一个时刻，空闲的连接数小于DB的请求数，则需要将请求排队，等待空闲连接处理。通过连接池可以复用连接，避免连接的频繁创建和释放，从而减少请求的平均响应时间，并且在请求繁忙时，通过请求排队，可以缓冲应用对DB的冲击。

线程池实现在server端，通过创建一定数量的线程服务DB请求，相对于one-conection-per-thread的一个线程服务一个连接的方式，线程池服务的最小单位是语句，即一个线程可以对应多个活跃的连接。通过线程池，可以将server端的服务线程数控制在一定的范围，减少了系统资源的竞争和线程上下文切换带来的消耗，同时也避免出现高连接数导致的高并发问题。

连接池和线程池相辅相成，通过连接池可以减少连接的创建和释放，提高请求的平均响应时间，并能很好地控制一个应用的DB连接数，但无法控制整个应用集群的连接数规模，从而导致高连接数，通过线程池则可以很好地应对高连接数，保证server端能提供稳定的服务。如图2所示，每个web-server端维护了3个连接的连接池，对于连接池的每个连接实际不是独占db-server的一个worker，而是可能与其他连接共享。这里假设db-server只有3个group，每个group只有一个worker，每个worker处理了2个连接的请求。



# 问题

## 内存泄漏问题

## 拨测异常问题

## 慢SQL引入的问题

# 线程池优化

## 调度死锁解决

引入线程池解决了多线程高并发的问题，但也带来一个隐患。假设，A，B两个事务被分配到不同的group中执行，A事务已经开始，并且持有锁，但由于A所在的group比较繁忙，导致A执行一条语句后，不能立即获得调度执行；而B事务依赖A事务释放锁资源，虽然B事务可以被调度起来，但由于无法获得锁资源，导致仍然需要等待，这就是所谓的调度死锁。由于一个group会同时处理多个连接，但多个连接不是对等的。比如，有的连接是第一次发送请求；而有的连接对应的事务已经开启，并且持有了部分锁资源。为了减少锁资源争用，后者显然应该比前者优先处理，以达到尽早释放锁资源的目的。因此在group里面，可以添加一个优先级队列，将已经持有锁的连接，或者已经开启的事务的连接发起的请求放入优先队列，工作线程首先从优先队列获取任务执行。

## 大查询处理

假设一种场景，某个group里面的连接都是大查询，那么group里面的工作线程数很快就会达到thread\_pool\_oversubscribe参数设置值，对于后续的连接请求，则会响应不及时(没有更多的连接来处理)，这时候group就发生了stall。

通过前面分析知道，timer线程会定期检查这种情况，并创建一个新的worker线程来处理请求。如果长查询来源于业务请求，则此时所有group都面临这种问题，此时主机可能会由于负载过大，导致hang住的情况。这种情况线程池本身无能为力，因为源头可能是烂SQL并发，或者SQL没有走对执行计划导致，通过其他方法，比如SQL高低水位限流或者SQL过滤手段可以应急处理。但是，还有另外一种情况，就是dump任务。很多下游依赖于数据库的原始数据，通常通过dump命令将数据拉到下游，而这种dump任务通常都是耗时比较长，所以也可以认为是大查询。如果dump任务集中在一个group内，并导致其他正常业务请求无法立即响应，这个是不能容忍的，因为此时数据库并没有压力，只是因为采用了线程池策略，才导致了请求响应不及时，为了解决这个问题，我们将group中处理dump任务的线程不计入thread\_pool\_oversubscribe累计值，避免上述问题。

# 适用场景

**适用于有大量短查询的业务场景**

在该场景下，每个连接一个线程，过多的连接数很容易达到连接数的最大值，同时，过多的活跃线程会导致频繁的上下文切换。此时，可使用线程池，因为是短查询，不会有某个连接长时间占用线程池中的线程，所以几乎不会影响客户端请求的响应时间，并且，随着连接数的增加，线程池中的线程数被控制都在一定范围内，减轻了系统的压力。

**在有大量长查询的业务场景下不适合使用线程池**

在该场景下，长查询可能会占据线程池的所有线程，导致线程池出现效率低效的情况，客户端设置不能进行连接。