# 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。

# 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);

# 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。

# 线程池参数

thread\_handling:表示线程池模型。

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:用于timer线程定期检查group是否“停滞”，参数表示检测的间隔。

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。

thread\_pool\_oversubscribe:该参数用于控制CPU核心上“超频”的线程数。这个参数设置值不含listen线程计数。

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

# 线程池实现

## 原理



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”。

## 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

# 线程池优化