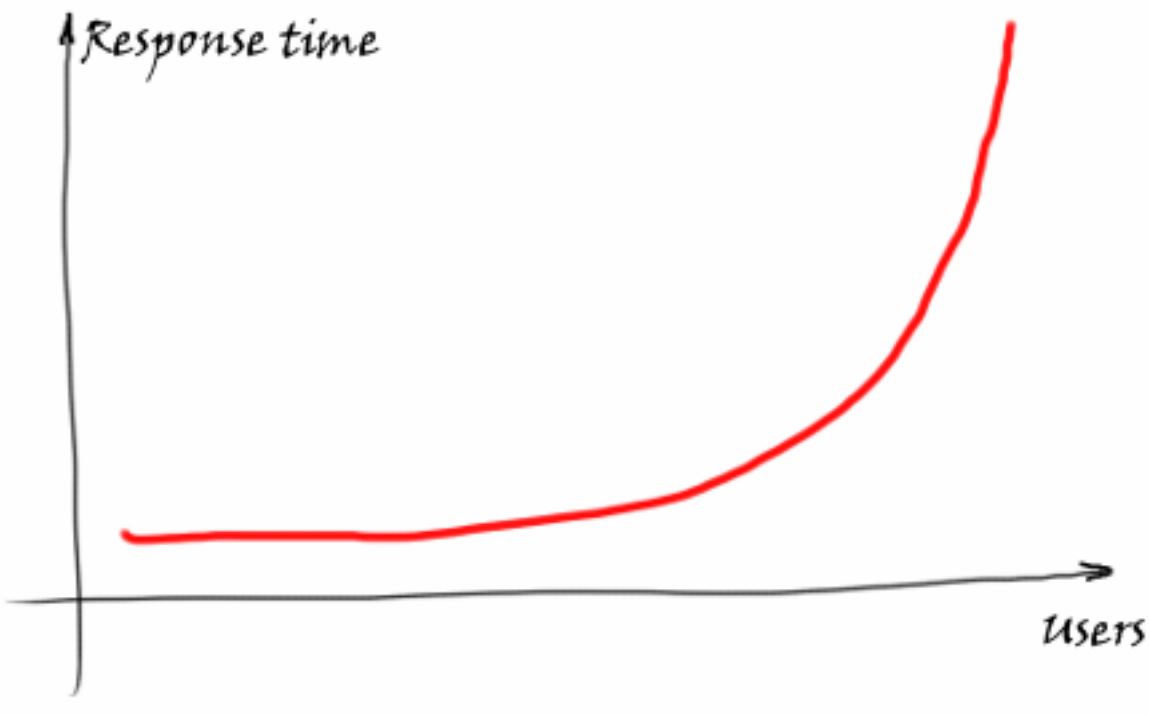
MySQL线程池问题个人整理

张秀云 | 腾讯数据库研发与运营高级工程师

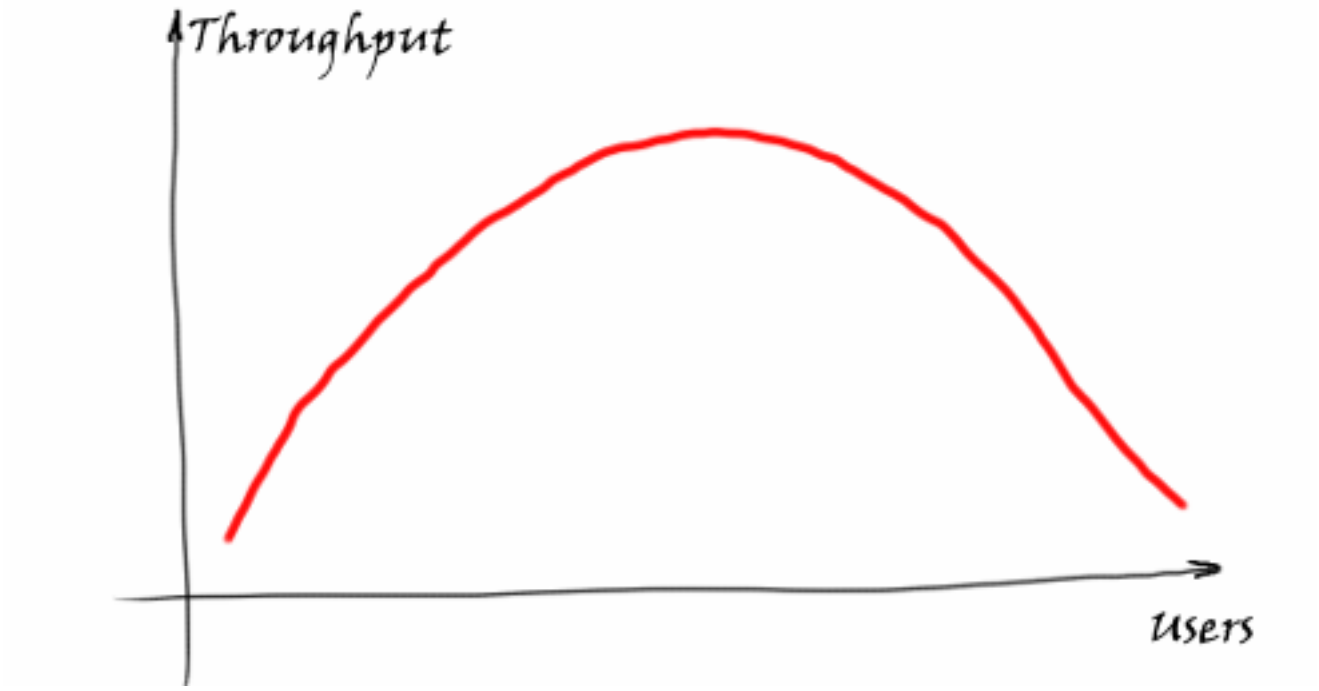
最近出现多次由于上层组件异常导致DB雪崩的情况，将部分监控DB启用了线程池功能。在使用线程池的过程中不断的深入学习，期间也遇到了不少问题。本文就来详细讲述一下MySQL线程池相关的知识，以帮助广大DBA快速了解MySQL的线程池机制，快速配置MySQL的线程池以及了解里面存在的一些坑。其实，我想说的是，了解和使用MySQL线程池，看这篇文章就够了。

一、为什么要使用MySQL线程池

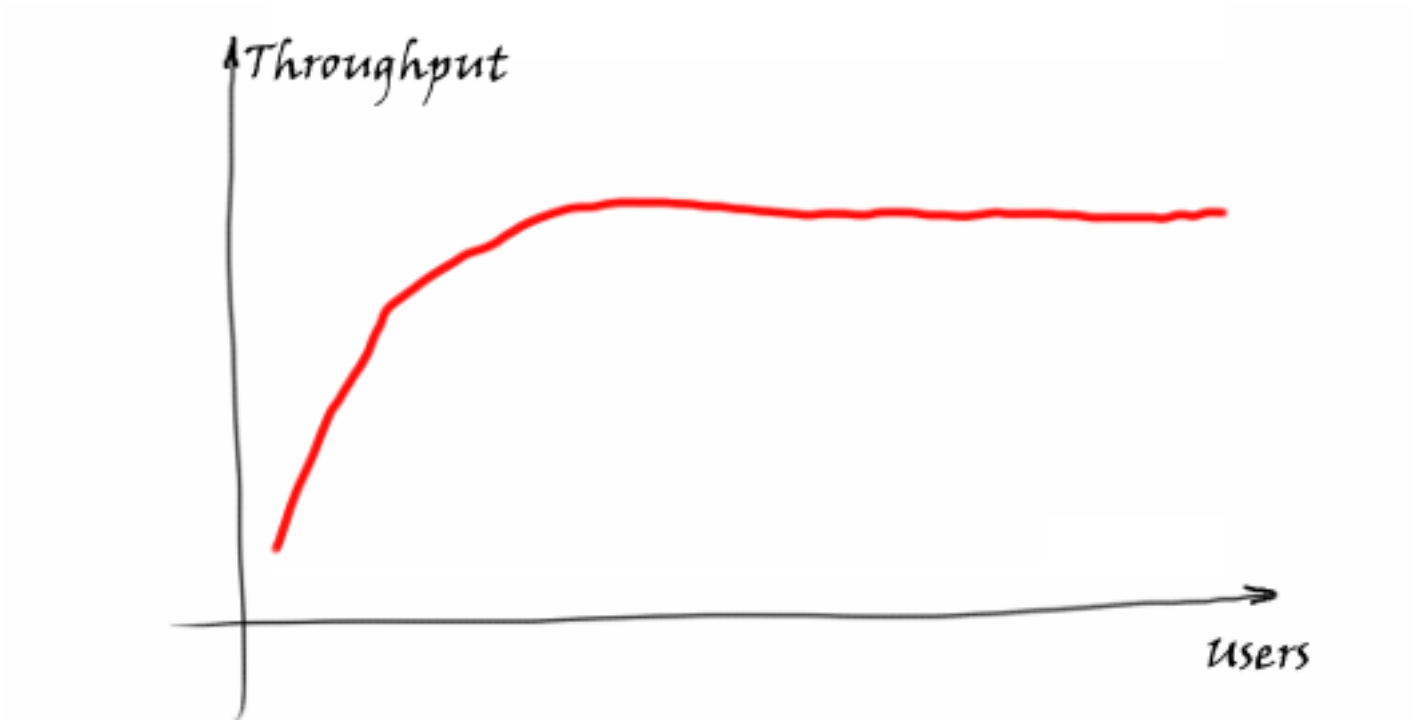
在介绍为什么要使用线程池之前，我们都知道，随着DB访问量越来越大，DB的响应时间也会随之越来越大，如下图：



而DB的访问大到一定程度的时候，DB的吞吐量也会出现下降，并且会越来越差，如下图所示：



那么是否有什么方式，能让对着DB的访问量越来越大，DB始终能表现出最佳的性能？类似下图的表现：



答案就是今天要重点介绍的线程池功能，总结一下，使用线程池的理由有如下两个：

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的时候，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，使用连接池是否也能达到类似的效果?

可能有的DBA会把线程池和连接池混淆，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；另外连接池可以取到避免了连接频繁创建和销毁，但是无法取到控制MySQL活动线程数的目标，在高并发场景下，无法取到保护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。

二、MySQL线程池介绍

（一）、MySQL线程池简介

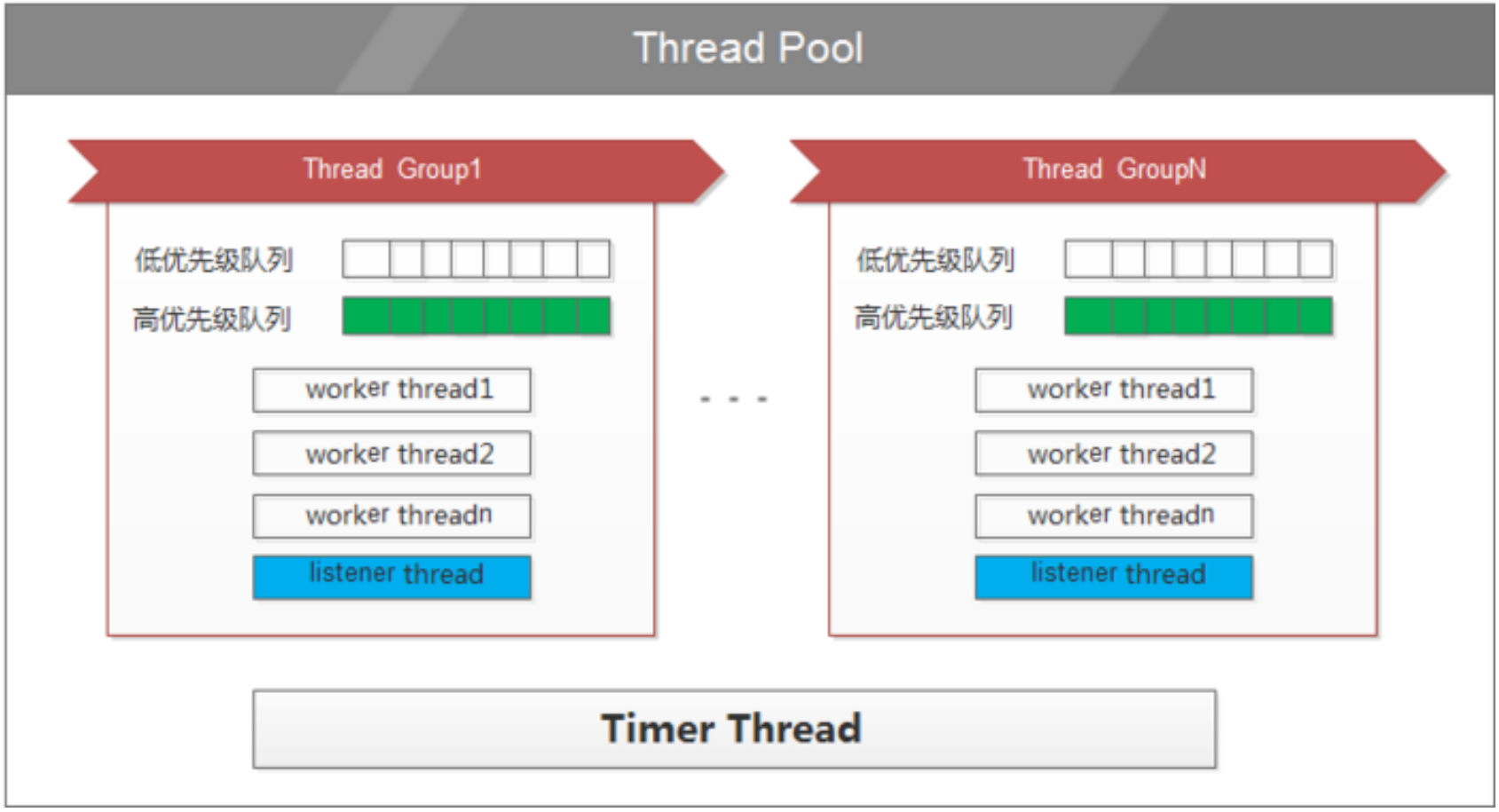
为了解决one-thread-per-connection(每个连接一个线程)存在的频繁创建和销毁大量线程以及高并发情况下DB雪崩的问题，实现DB在高并发环境依然能保持较高的性能。Oracle和MariaDB都推出了ThreadPool方案，目前Oracle的threadpool实现为plugin方式，并且只添加到在Enterprise版本中，Percona移植了MariaDB的threadpool功能，并做了进一步的优化。本文的环境是基于Percona MySQL 5.7版本。

（二）、MySQL线程池架构

MySQL的threadpool（线程池）被划分为多个group（组），每个组又有对应的工作线程，整体的工作逻辑还是比较复杂，下面我试图通过简单的方式来介绍MySQL线程池的工作原理。

1、架构图

首先来看看threadpool的架构图



2、Thread Pool的组成

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：

a、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。

b、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。

c、worker线程

worker线程是真正干活的线程

d、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。

3、Thread Pool的是如何运作的?

下面描述极简的Thread Pool运作，只是简单描述，省略了大量的复杂逻辑，请不要挑刺，~~

a、请求连接到MySQL，根据threadid%thread_pool_size确定落在哪个group

b、group中的listener线程监听到所在的group有新的请求以后，检查队列中是否有请求还未处理，如果没有，则自己转换为worker线程立即处理该请求，如果队列中还有未处理的请求，则将对应请求放到队列中，让其他的线程处理。

c、group中的thread线程检查队列的请求，如果队列中有请求，则进行处理，如果没有请求，则休眠，一直没有被唤醒，超过thread_pool_idle_timeout后就自动退出。线程结束。当然，获取请求之前会先检查group中的running线程数是否超过thread_pool_oversubscribe+1，如果

超过也会休眠。

d、timer线程定期检查各个group是否有阻塞，如果有，就对wokrer线程进行唤醒或者创建一个新的worker线程。

4、Thread Pool的分配机制

线程池会根据参数thread_pool_size的大小分成若干的group，每个group各自维护客户端发起的连接，当客户端发起连接到MySQL的时候，MySQL会跟进连接的线程id（thread_id）对thread_pool_size进行取模，从而落到对应的group。thread_pool_oversubscribe参数控制每个group的最大并发线程数，每个group的最大并发线程数为thread_pool_oversubscribe+1个，若对应的group达到了最大的并发线程数，则对应的连接就需要等待。这个分配机制在某个group中有多个慢SQL的场景下会导致普通的SQL运行时间很长，这个问题在后面的会做详细描述。

（三）、MySQL线程池参数说明

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	pool-of-threads
thread_pool_high_prio_mode	transactions
thread_pool_high_prio_tickets	4294967295
thread_pool_idle_timeout	60
thread_pool_max_threads	100000
thread_pool_oversubscribe	3
thread_pool_size	24
thread_pool_stall_limit	500

> thread_handling

该参数是配置线程模型，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

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

> 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

该参数设置timer线程的检测group是否异常的时间间隔，默认为500ms

三、MySQL线程池的使用

线程池的使用比较简单，只需要添加配置后重启实例即可

具体配置如下：

```
#thread pool thread_handling=pool-of-threads
thread_pool_oversubscribe=3 thread_pool_size=24
performance_schema=off #extra connection extra_max_connections = 8
extra_port = 33333
```

备注：其他参数默认即可

以上具体的参数在前面已经有做详细说明。下面是配置中需要注意的两个点：

1、只所以添加performance_schema=off，是由于测试过程中发现Thread pool和PS同时开启的时候会出现内存泄漏，在后面遇到的问题部分有详细描述；

2、添加extra connection是防止线程池满的情况下无法登录MySQL，因此特意已用管理端口，以备紧急的情况下使用；

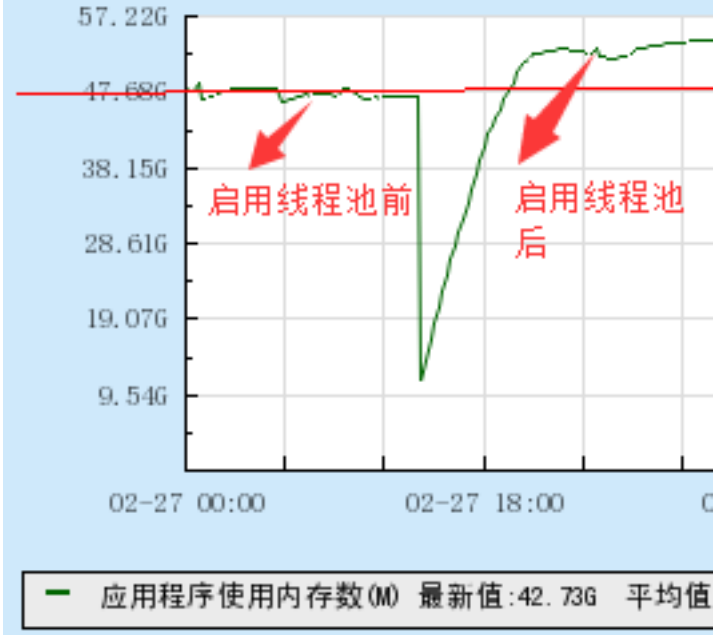
重启实例后，可以通过show variables like '%thread%';来查看配置的参数是否生效。

四、使用MySQL线程池遇到的问题

在使用线程池的过程中，遇到了几个问题，也顺便总结一下：

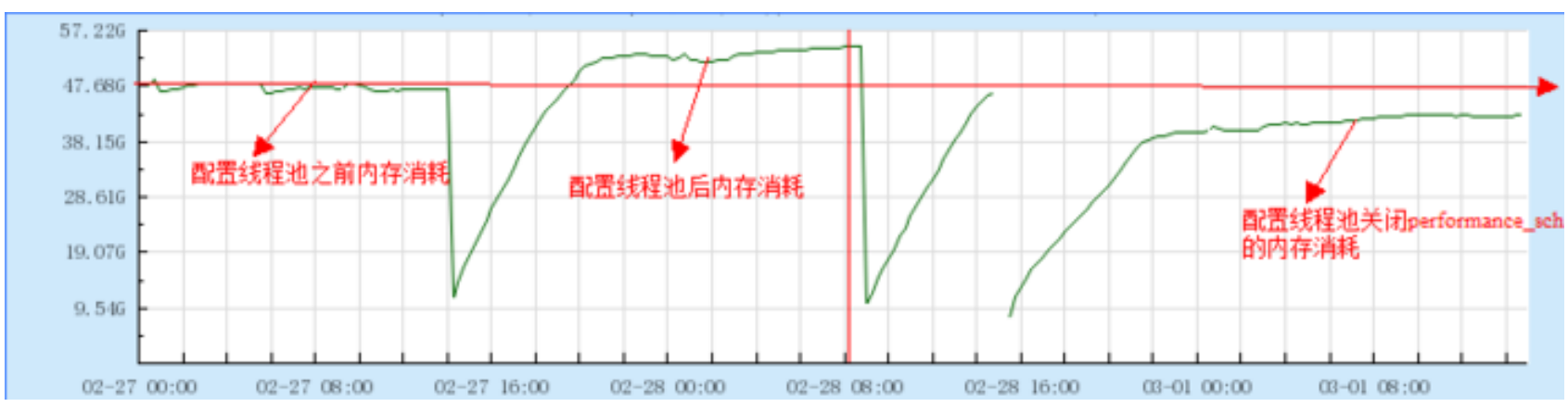
（一）内存泄漏问题

DB启用线程池后，内存飙升了8G左右，如下图：



不但启用线程池后内存飙升了8G左右，而且内存还在持续增长，很明显启用线程池后存在内存泄漏了。

这个问题在网上也有不少的人遇到，确认是percona的bug导致（https://jira.percona.com/browse/PS-3734），只有开启Performance_Schema和ThreadPool的时候才会出现，解决办法是关闭Performance_Schema即可，具体操作方法是在配置文件添加performance_schema=off，然后重启MySQL就OK。下面是关闭PS后的内存使用情况对比：



备注：

目前Percona server 5.7.21-20版本已经修复了线程池和PS同时打开内存泄漏的问题，从我测试的情况来看问题也得到了解决，大家也可以直接使用Percona server 5.7.21-20的版本，如下图

PS-1096: destroy PSI objects when removing thread from pool

Browse files

5.7 (#2184)

dutow committed 28 days ago 1 parent 2b3250b commit 5060e22e1dbd37b7fd3b370bebf3663467142ddd

Showing 1 changed file with 4 additions and 0 deletions. Unified Split

4 sql/threadpool_common.cc View

```
@@ -208,6 +208,10 @@ void threadpool_remove_connection(THD *thd)
208 208
209 209    thd->release_resources();
210 210
211     +#ifdef HAVE_PSI_THREAD_INTERFACE
212     +  PSI_THREAD_CALL(delete_thread)(thd->get_psi());
213     +#endif
214     +
211 215    Global_THD_manager::get_instance()->remove_thd(thd);
212 216    Connection_handler_manager::dec_connection_count(false);
213 217    delete thd;
```

（二）拨测异常问题

启用线程池以后，相当于限制了MySQL的并发线程数，当达到最大线程数的时候，其他的线程需要等待，新连接也会卡在连接验证那一步，这时候会造成拨测程序连接MySQL超时，拨测返回错误如下：

```
[2018-03-18 01:11:32] check_mysql_health:MYSQL_OK=ERROR:returnCode=1:[mysql: [Warning] Using a password on the command
ERROR 2013 (HY000): Lost connection to MySQL server at 'waiting for initial communication packet', system error: 110]
```

拨测程序连接实例超时后，就会认为master已经出现问题，极端情况下，重试多次都有异常后，就启动自动切换的操作，将业务切换到从机。

这种情况有两种解决办法：

1、启用MySQL的旁路管理端口，监控和高可用相关直接使用MySQL的旁路管理端口

具体做法为是在my.cnf中添加如下配置后重启，就可以通过旁路端口登录MySQL了，不受线程池最大线程数的影响：

```
extra_max_connections = 8     extra_port = 33333
```

备注：建议启用线程池后，这个也添加上，方便紧急情况下进行故障处理。

2、修改高可用探测脚本，将达到线程池最大活动线程数返回的错误做异常

处理，类似于当作超过最大连接数的场景（备注：超过最大连接数只告警，不进行自动切换）

（三）慢SQL引入的问题

随着对拨测超时的问题的深入分析，线程池满只是监控拨测出现超时的其中一种情况，还有一种情况是线程池并没有满，线上的两个配置：

> thread_pool_oversubscribe=3 thread_pool_size=24

按照上面的两个配置来计算的话，总共能并发运行24x(3+1)=96，但是根据多次问题的追踪，发现有多次线程池并没有达到96，也就是说整体的线程池并没有满。那会是什么问题导致拨测失败呢？

鉴于线程池的结构和分配机制，通过前面线程池部分的描述，大家都知道了在内部是将线程池分成一个一个的group，我们线上配置了24个group，而线程池的分配机制是对Threadid进行取模，然后确定该线程是落在哪个group。出现超时的时候，有很多的load线程到导入数据。也就是说那个时候有部分线程比较慢的情况。那么会不会是某个group的线程满了，从而导致新分配的线程等待？

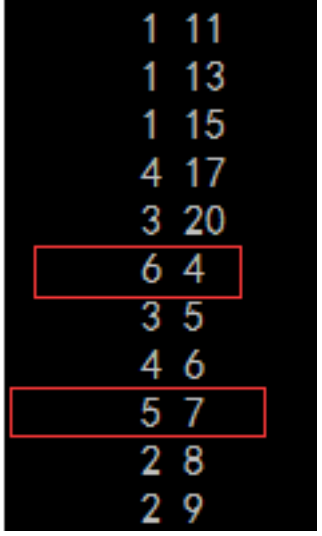
有了这个猜想以后，接下来就是来验证这个问题。验证分为两步：

1、抓取线上运行的processlist，然后对threadid取模，看看是否有多个load线程落在同一个group的情况

2、在测试环境模拟这种场景，看看是否符合预期

线上场景分析：

先来看线上的场景，通过抓取拨测超时时间点的processlist，找出当时正在load的线程，根据threadid进行去模，并进行汇总统计后，得出如下结果：



可以看出，当时第4和第7个group的请求个数都超过了4个，说明是单个group满导致的拨测异常，当然，也会导致部分运行很快的SQL变慢。

测试环境模拟场景分析：

为了构建快速重现环境，我将参数调整如下：

> thread_pool_oversubscribe=1 thread_pool_size=2

通过上面参数的调整，可以计算出最大并发线程为2x(1+1)=4，如下图，当活动线程数超过4个后，其他的线程就必须等待：

```
mysql> show processlist;
+-----+-----------------------+--------------------+------+---------+------+------------+---------------------+-----------+---------------+
| Id  | User                  | Host               | db   | Command | Time | State      | Info                | Rows_sent | Rows_examined |
+-----+-----------------------+--------------------+------+---------+------+------------+---------------------+-----------+---------------+
| 137 | root                  | localhost          | NULL | Query   |   25 | User sleep | select sleep(100)   |         0 |             0 |
| 139 | root                  | localhost          | NULL | Query   |   27 | User sleep | select sleep(100)   |         0 |             0 |
| 140 | root                  | localhost          | NULL | Query   |   29 | User sleep | select sleep(100)   |         0 |             0 |
| 236 | root                  | localhost          | NULL | Query   |   22 | User sleep | select sleep(100)   |         0 |             0 |
| 237 | root                  | 127.0.0.1:58635    | NULL | Query   |    0 | starting   | show processlist    |         0 |             0 |
| 244 | unauthenticated user  | connecting host    | NULL | Connect |   18 | login      | NULL                |         0 |             0 |
| 245 | unauthenticated user  | connecting host    | NULL | Connect |    2 | login      | NULL                |         0 |             0 |
+-----+-----------------------+--------------------+------+---------+------+------------+---------------------+-----------+---------------+
```

我模拟线上环境的方法为，开启1个线程的慢SQL，这个时候，测试环境的线程池情况如下：

按照之前的推测，这个时候Group1的处理能力相当于Group2的处理能力的50%，如果之前的推论是正确的，那么分配在Group1上的线程就会出现阻塞。比如此时来了20个线程请求，按照线程池的分配原则，此时Group1和

Group2都会分到10个线程请求。稼穑所有的线程请求耗时都是一样的，那么分配到Group1上的线程请求整体处理时间应该是分配到Group2上整体处理时间的2倍。

我使用脚本，并发发起12个线程请求，每个线程请求都运行select sleep(2)，那么在Group1和Group2都空闲的情况下，运行情况如下：

> 2018-03-18-20:23:53 2018-03-18-20:23:53 2018-03-18-20:23:53 2018-03-18-20:23:53 2018-03-18-20:23:55 2018-03-18-20:23:55 2018-03-18-20:23:55 2018-03-18-20:23:55 2018-03-18-20:23:57 2018-03-18-20:23:57 2018-03-18-20:23:57 2018-03-18-20:23:57

每次4个线程，总共运行了6秒

接下来在Group1被1个长时间运行的线程沾满以后，看看测试结果是怎么样的

> 2018-03-18-20:24:35 2018-03-18-20:24:35 2018-03-18-20:24:35 2018-03-18-20:24:37 2018-03-18-20:24:37 2018-03-18-20:24:37 2018-03-18-20:24:39 2018-03-18-20:24:39 2018-03-18-20:24:39 2018-03-18-20:24:41 2018-03-18-20:24:43 2018-03-18-20:24:45

从上面的结果中可以看出，在没有阻塞的时候，每次都是4个线程，而后面有1个线程长时间运行的时候，就会出现那个长时间线程对应的group出现排队的情况，最后虽然有3个空闲的线程，但是却是只有1个线程在处理（红色部分结果）。

解决把法有两个：

1、将thread_pool_oversubscribe适当调大，这个办法只能缓解类似问题，无法根治；

2、找到慢的SQL，解决慢的问题；

五、参考文献：

https://www.percona.com/doc/percona-

server/LATEST/performance/threadpool.html

https://www.percona.com/blog/2013/03/16/simcity-outages-traffic-control-and-thread-pool-for-mysql/

http://www.cnblogs.com/cchust/p/4510039.html

http://blog.jobbole.com/109695/

http://blog.csdn.net/u012662731/article/details/54375137