OkHttp3源码分析[复用连接池]

BlackSwift (/u/b99b0edd4e77) + 关注
2016.01.16 00:15* 字数 2688 阅读 20568 评论 39 喜欢 96 赞赏 6
(/u/b99b0edd4e77)

OkHttp系列文章如下

- OkHttp3源码分析[综述] (https://www.jianshu.com/p/aad5aacd79bf)
- OkHttp3源码分析[复用连接池] (https://www.jianshu.com/p/92a61357164b)
- OkHttp3源码分析[缓存策略] (https://www.jianshu.com/p/9cebbbd0eeab)
- OkHttp3源码分析[DiskLruCache] (https://www.jianshu.com/p/23b8aa490a6b)
- OkHttp3源码分析[任务队列] (https://www.jianshu.com/p/6637369d02e7)

---

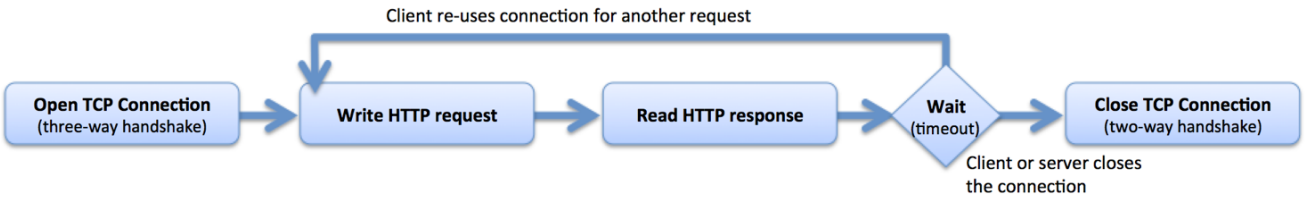
1. 概述

HTTP中的 keepalive连接 在网络性能优化中，对于延迟降低与速度提升的有非常重要的作用。

通常我们进行http连接时，首先进行tcp握手，然后传输数据，最后释放

这种方法的确简单，但是在复杂的网络内容中就不够用了，创建socket需要进行3次握手，而释放socket需要2次握手(或者是4次)。重复的连接与释放tcp连接就像每次仅仅挤1mm的牙膏就合上牙膏盖子接着再打开接着挤一样。而每次连接大概是TTL一次的时间(也就是ping一次)，在TLS环境下消耗的时间就更多了。很明显，当访问复杂网络时，延时（而不是带宽）将成为非常重要的因素。

当然，上面的问题早已经解决了，在http中有一种叫做 keepalive connections 的机制，它可以在传输数据后仍然保持连接，当客户端需要再次获取数据时，直接使用刚刚空闲下来的连接而不需要再次握手



在现代浏览器中，一般同时开启6～8个 keepalive connections 的socket连接，并保持一定的链路生命，当不需要时再关闭；而在服务器中，一般是由软件根据负载情况(比如FD最大值、Socket内存、超时时间、栈内存、栈数量等)决定是否主动关闭。

> Okhttp支持5个并发KeepAlive，默认链路生命为5分钟(链路空闲后，保持存活的时间)

当然keepalive也有缺点，在提高了单个客户端性能的同时，复用却阻碍了其他客户端的链路速度，具体来说如下

1. 根据TCP的拥塞机制，当总水管大小固定时，如果存在大量空闲的 `keepalive connections` （我们可以称作 `僵尸连接` 或者 `泄漏连接` ），其它客户端们的正常连接速度也会受到影响，这也是运营商为何限制P2P连接数的道理
2. 服务器/防火墙上有并发限制，比如 `apache` 服务器对每个请求都开线程，导致只支持150个并发连接（数据来源于nginx官网），不过这个瓶颈随着高并发server软硬件的发展（golang/分布式/IO多路复用）将会越来越少
3. 大量的DDOS产生的僵尸连接可能被用于恶意攻击服务器，耗尽资源

好了，以上科普完毕，本文主要是写客户端的，服务端不再介绍。

下文假设服务器是经过专业的运维配置好的，它默认开启了 `keep-alive` ，并不主动关闭连接

2. 连接池的使用与分析

首先先说下源码中关键的对象：

- `Call`：对http的请求封装，属于程序员能够接触的上层高级代码
- `Connection`：对jdk的socket物理连接的包装，它内部有 `List<WeakReference<StreamAllocation>>` 的引用
- `StreamAllocation`：表示 `Connection` 被上层高级代码的引用次数
- `ConnectionPool`：Socket连接池，对连接缓存进行回收与管理，与CommonPool有类似的设计
- `Deque`：Deque也就是双端队列，双端队列同时具有队列和栈性质，经常在缓存中被使用，这个是java基础

在okhttp中，连接池对用户，甚至开发者都是透明的。它自动创建连接池，自动进行泄漏连接回收，自动帮你管理线程池，提供了put/get/clear的接口，甚至内部调用都帮你写好了。

在以前的内存泄露分析文章 (https://www.jianshu.com/p/c59c199ca9fa)中我写到，我们知道在socket连接中，也就是 `Connection` 中，本质是封装好的流操作，除非手动 `close` 掉连接，基本不会被GC掉，非常容易引发内存泄露。所以当涉及到并发socket编程时，我们就会非常紧张，往往写出来的代码都是 `try/catch/finally` 的迷之缩进，却又对这样的代码无可奈何。

在okhttp中，在高层代码的调用中，使用了类似于引用计数的方式跟踪Socket流的调用，这里的计数对象是 `StreamAllocation` ，它被反复执行 `aquire` (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L296-L296) 与 `release` (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L301-L301)操作(点击函数可以进入github查看)，这两个函数其实是在改变 `Connection` 中的 `List<WeakReference<StreamAllocation>>` 大小。 `List` 中Allocation的数量也就是物理socket被引用的计数（Refference Count），如果计数为0的话，说明此连接没有被使用，是空闲的，需要通过下文的算法实现回收；如果上层代码仍然引用，就不需要关闭连接。

> 引用计数法：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用。它不能处理循环引用的问题。

2.1. 实例化

在源码中，我们先找 `ConnectionPool` 实例化的位置，它是直接new出来的，而它的各种操作却在 `OkHttpClient` 的static区 (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/ee83d8d26afd92d27fbcd2a328e882f25a5090c6/okhttp/src/main/java/okhttp3/OkHttpClient.java#L65-L65)实现了 `Internal.instance` 接口作为 `ConnectionPool` 的包装。

> 至于为什么需要这么多此一举的分层包装，主要是为了让外部包的成员访问非 `public` 方法，详见这里注释 (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/21d63034188d90ca51d635be348d5deba4abeca3/okhttp/src/main/java/okhttp3/internal/Internal.java#L34-L34)

2.2. 构造

1. 连接池内部维护了一个叫做 `OkHttp ConnectionPool` 的 `ThreadPool` ，专门用来淘汰末位的socket，当满足以下条件时，就会进行末位淘汰，非常像GC

```
1. 并发socket空闲连接超过5个
2. 某个socket的keepalive时间大于5分钟
```

2. 维护着一个 `Deque<Connection>` ，提供get/put/remove等数据结构的功能

3. 维护着一个 `RouteDatabase` ，它用来记录连接失败的 `Route` 的黑名单，当连接失败的时候就会把失败的线路加进去（本文不讨论）

2.3 put/get操作

在连接池中，提供如下的操作，这里可以看成是对deque的一个简单的包装

```
//从连接池中获取
get
//放入连接池
put
//线程变成空闲，并调用清理线程池
connectionBecameIdle
//关闭所有连接
evictAll
```

随着上述操作被更高级的对象调用， `Connection` 中的 `StreamAllocation` 被不断的 `aquire` (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L296-L296) 与 `release` (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L301-L301)，也就是 `List<WeakReference<StreamAllocation>>` 的大小将**时刻变化**

2.4 Connection自动回收的实现

java内部有垃圾回收GC，okhttp有socket的回收；垃圾回收是根据对象的引用树实现的，而okhttp是根据 `RealConnection` 的虚引用 `StreamAllocation` 引用计数是否为0实现的。我们先看代码

cleanupRunnable:

当用户socket连接成功，向连接池中 `put` 新的socket时，回收函数会被主动调用，线程池就会执行 `cleanupRunnable` ，如下

```
//Socket清理的Runnable，每当put操作时，就会被主动调用
//注意put操作是在网络线程
//而Socket清理是在`OkHttp ConnectionPool`线程池中调用
while (true) {
  //执行清理并返回下场需要清理的时间
  long waitNanos = cleanup(System.nanoTime());
  if (waitNanos == -1) return;
  if (waitNanos > 0) {
    synchronized (ConnectionPool.this) {
      try {
        //在timeout内释放锁与时间片
        ConnectionPool.this.wait(TimeUnit.NANOSECONDS.toMillis(waitNanos));
      } catch (InterruptedException ignored) {
      }
    }
  }
}
```

这段死循环实际上是一个阻塞的清理任务，首先进行清理(clean)，并返回下次需要清理的间隔时间，然后调用 `wait(timeout)` 进行等待以释放锁与时间片，当等待时间到了后，再次进行清理，并返回下次要清理的间隔时间...

Cleanup:

cleanup (https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/ConnectionPool.java#L183-L183)使用了类似于GC的 标记-清除算法 ，也就是首先标记出最不活跃的连接(我们可以叫做 泄漏连接 ，或者 空闲连接)，接着进行清除，流程如下:

```
long cleanup(long now) {
  int inUseConnectionCount = 0;
  int idleConnectionCount = 0;
  RealConnection longestIdleConnection = null;
  long longestIdleDurationNs = Long.MIN_VALUE;

  //遍历`Deque`中所有的`RealConnection`，标记泄漏的连接
  synchronized (this) {
    for (RealConnection connection : connections) {
      // 查询此连接内部StreamAllocation的引用数量
      if (pruneAndGetAllocationCount(connection, now) > 0) {
        inUseConnectionCount++;
        continue;
      }

      idleConnectionCount++;

      //选择排序法，标记出空闲连接
      long idleDurationNs = now - connection.idleAtNanos;
      if (idleDurationNs > longestIdleDurationNs) {
        longestIdleDurationNs = idleDurationNs;
        longestIdleConnection = connection;
      }
    }

    if (longestIdleDurationNs >= this.keepAliveDurationNs
        || idleConnectionCount > this.maxIdleConnections) {
      //如果(`空闲socket连接超过5个`
      //且`keepalive时间大于5分钟`)
      //就将此泄漏连接从`Deque`中移除
      connections.remove(longestIdleConnection);
    } else if (idleConnectionCount > 0) {
      //返回此连接即将到期的时间，供下次清理
      //这里依据是在上文`connectionBecameIdle`中设定的计时
      return keepAliveDurationNs - longestIdleDurationNs;
    } else if (inUseConnectionCount > 0) {
      //全部都是活跃的连接，5分钟后再次清理
      return keepAliveDurationNs;
    } else {
      //没有任何连接，跳出循环
      cleanupRunning = false;
      return -1;
    }
  }

  //关闭连接，返回`0`，也就是立刻再次清理
  closeQuietly(longestIdleConnection.socket());
  return 0;
}
```

太长不想看的话，就是如下的流程：

1. 遍历 `Deque` 中所有的 `RealConnection` ，标记泄漏的连接
2. 如果被标记的连接满足(`空闲socket连接超过5个` && `keepalive时间大于5分钟`)，就将此连接从 `Deque` 中移除，并关闭连接，返回 `0` ，也就是将要执行 `wait(0)` ，提醒立刻再次扫描
3. 如果(`目前还可以塞得下5个连接，但是有可能泄漏的连接(即空闲时间即将达到5分钟)`)，就返回此连接即将到期的剩余时间，供下次清理
4. 如果(`全部都是活跃的连接`)，就返回默认的 `keep-alive` 时间，也就是5分钟后再执行清理
5. 如果(`没有任何连接`)，就返回 `-1` ,跳出清理的死循环

> 再次注意：这里的“并发”==(“空闲” + “活跃”)==5，而不是说并发连接就一定是活跃的连接

pruneAndGetAllocationCount:

如何标记并找到最不活跃的连接呢，这里使用了 `pruneAndGetAllocationCount` 的方法(https://link.jianshu.com?t=https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/ConnectionPool.java#L238-L238)，它主要依据弱引用是否为 `null` 而判断这个连接是否泄漏

```
//类似于引用计数法，如果引用全部为空，返回立刻清理
private int pruneAndGetAllocationCount(RealConnection connection, long now) {
  //虚引用列表
  List<Reference<StreamAllocation>> references = connection.allocations;
  //遍历弱引用列表
  for (int i = 0; i < references.size(); ) {
    Reference<StreamAllocation> reference = references.get(i);
    //如果正在被使用，跳过，接着循环
    //是否置空是在上文`connectionBecameIdle`的`release`控制的
    if (reference.get() != null) {
      //非常明显的引用计数
      i++;
      continue;
    }

    //否则移除引用
    references.remove(i);
    connection.noNewStreams = true;

    //如果所有分配的流均没了，标记为已经距离现在空闲了5分钟
    if (references.isEmpty()) {
      connection.idleAtNanos = now - keepAliveDurationNs;
      return 0;
    }
  }

  return references.size();
}
```

1. 遍历 `RealConnection` 连接中的 `StreamAllocationList` ，它维护着一个弱引用列表
2. 查看此 `StreamAllocation` 是否为空(它是在线程池的put/remove手动控制的)，如果为空，说明已经没有代码引用这个对象了，需要在List中删除
3. 遍历结束，如果List中维护的 `StreamAllocation` 删空了，就返回 `0` ，表示这个连接已经没有代码引用了，是 泄漏的连接 ;否则返回非0的值，表示这个仍然被引用，是活跃的连接。

上述实现的过于保守，实际上用filter就可以大致实现，伪代码如下

```
return references.stream().filter(reference -> {
    return !reference.get() == null;
}).count();
```

总结

通过上面的分析，我们可以总结，okhttp使用了类似于引用计数法与标记擦除法的混合使用，当连接空闲或者释放时， `StreamAllocation` 的数量会渐渐变成0，从而被线程池监测到并回收，这样就可以保持多个健康的keep-alive连接，Okhttp的黑科技就是这样实现的。

最后推荐一本《图解HTTP》，日本人写的，看起来很不错。

再推荐阅读开源Redis客户端Jedis的源码，可以看下它的 `JedisFactory` 的实现。

如果你期待更多高质量的文章，不妨关注我或者点赞吧！

Ref

1. https://www.nginx.com/blog/http-keepalives-and-web-performance/ (https://link.jianshu.com?t=https://www.nginx.com/blog/http-keepalives-and-web-performance/)

< 上一篇 (/p/aad5aacd79bf)	目录	下一篇 > (/p/9cebbbd0eeab)