链滴

阿里 memcached 客户端 socket 连接池源码分析

作者：changming

原文链接：https://ld246.com/article/1471966775322

来源网站：链滴

在<a href="http://www.yangchangming.com/articles/2016/08/23/1471966196362.html"上篇文章中，我们对阿里memcached客户端主要源码进行了分析，其中socket连接池部分是系关键部分，涉及到整个客户端运行是否稳定，与服务端连接是否高效，本篇就来分析socket连接池实的这部分源码。
连接池源码位于com.alisoft.xplatform.asf.cache.memcached.client下的SockIOPool类，该类包括2个内部类，分别是MaintThread和SockIO。SockIOPool主要功能是用于维护与memcached务端的持久化连接；并提供初始化连接池，获取、释放连接，以及设置服务器权重，连接池维护等功。
<blockquote>
初始化连接池
</blockquote>
在系统启动时，会首先读取缓存服务设置的相关配置文件，读取成功后即开始实例化一个SockIOool实例，再将pool的相关设置参数赋给该实例，如
<pre class="brush: java">//所有memcahed服务端ip地址，以及对应的权重
private String[] servers;
private Integer[] weights;</pre>
然后调用初始化逻辑，即initialize方法；该方法中，用一个支持并发的concurrenthashap初始化一个容器socketpool，容器大小由配置的memcachd服务端和初始化连接数决定。
<pre class="brush: java">socketPool = new ConcurrentHashMap<String, ConcurrentMap<SockIO, Integer>>(servers.length*initConn);</pre>
该容器中存放的是每个server地址，以及对应socket的一个包装内部类SockIO（在该内类中完成所有的socket相关操作），同时该socket的状态也被记录下来，下面看看初始化代码：</spn>
<pre class="brush: java">public void initialize()
{
     // check to see if already initialized
     if (initialized && (buckets != null || consistentBuckets != null)&& (socktPool != null))
    {
       log.error("++++ trying to initialize an already initialized pool");
       return;
    }
       //加锁，防止多线程并发问题
    initDeadLock.lock();
    try
    {
      // check to see if already initialized
      if (initialized && (buckets != null || consistentBuckets != null)&& (socetPool != null))
      {
       log.error("++++ trying to initialize an already initialized pool");
       return;
      }

   // pools
   socketPool = new ConcurrentHashMap<String, ConcurrentMap<SockIO, Integer>&t;(
               servers.length * initConn);

fastPool = new HashMap<String,SockIO>();</pre>

```
hostDeadDur = new ConcurrentHashMap&lt;String, Long&gt;();
hostDead = new ConcurrentHashMap&lt;String, Date&gt;();
maxCreate = (poolMultiplier &gt; minConn) ? minConn : minConn
        / poolMultiplier; // only create up to maxCreate
            // connections at once

.......

// if servers is not set, or it empty, then
// throw a runtime exception
if (servers == null || servers.length &lt;= 0)
{
    log.error("++++ trying to initialize with no servers");
    throw new IllegalStateException("++++ trying to initialize with no servers");
}

// 初始化hash环结构的同时，创建每个server的socket，具体初始化hash算法可以参见上篇文章
if (this.hashingAlg == CONSISTENT_HASH)  //如果是hash一致性算法
    populateConsistentBuckets();
else
    populateBuckets();

// mark pool as initialized
this.initialized = true;

// 开始执行维护线程，该新线程会根据用户设定的时间间隔(maintsleep)进行连接池的维护工作
if (this.maintSleep &gt; 0)
this.startMaintThread();

} finally
{
   initDeadLock.unlock();
}

}</pre>

<blockquote>
<p>创建sokcet连接</p>
</blockquote>
<p>由代码中我们可以看出，初始化hash环结构时，即为每个server初始化了socket连接，创建连接辑由createSocket方法负责。</p>
<p>如果给定的server发生故障，或者其他原因，无法创建socket的话，策略就是将其加入故障服务队列<strong>hostDead</strong>，并且设置故障过期时间，下次再有需要对该server创建socket，会先检测hostDead中是否包含该server，以及过期时间是否已经过了，如果包含并且未过过期时的话，直接返回null，不在创建socket。无论创建是否成功，都会调用addSocketToPool方法将socke放入容器<strong>socketPool</strong>中。</p>
<p><strong><em>注意：同一个server，有可能被创建多个socket。</em></strong></p>
<pre class="brush: java">protected &lt;T&gt; boolean addSocketToPool(ConcurrentMap&lt;String, ConcurrentMap&lt;SockIO, T&gt;&gt; pool, String host,SockIO socket, T oldValue,T newValue, boolean needReplace)
{
    .....
    if (!pool.containsKey(host))
    {
```

```
    sockets = new ConcurrentHashMap&lt;SockIO, T&gt;();
    pool.putIfAbsent(host, sockets);
    }
    sockets = pool.get(host);
    if (sockets != null)
    {
        if (needReplace)
        {       //对于同一个host，有可能创建多个socket
            sockets.put(socket, newValue);
            result = true;
        }
        else{
            return sockets.replace(socket, oldValue, newValue);
        }
    }
    return result;
}</pre>
<p><span>创建socket的逻辑就是构建一个SockIO对象，默认使用NIO建立socket，部分代码如下
</span></p>
<pre class="brush: java">public SockIO(SockIOPool pool, String host, int timeout,
            int connectTimeout, boolean noDelay) throws IOException,UnknownHostException
{
    .......
    // 创建真正的sokcet对象，默认使用NIO
 sock = getSocket(host.substring(0,index), Integer.parseInt(host.substring(index+1)), connectT
meout);


if (timeout &gt;= 0)
this.sock.setSoTimeout(timeout);
// testing only
sock.setTcpNoDelay(noDelay);
// 包装输入输出流
in = new DataInputStream(sock.getInputStream());
out = new BufferedOutputStream(sock.getOutputStream());

this.host = host;

}</pre>

<p><span>从SocketChannel中获取一个socket连接。</span></p>
<pre class="brush: java">protected static Socket getSocket(String host, int port, int timeout)
hrows IOException
{
    SocketChannel sock = SocketChannel.open();
    sock.socket().connect(new InetSocketAddress(host, port), timeout);
    return sock.socket();
}</pre>
<blockquote>
<p>如何获取socket连接</p>
</blockquote>
<p>连接池初始化成功，socket也创建完毕，那么下面看看如何获取一个指定server的socket连接。<
p>
<pre class="brush: java">public SockIO getConnection(String host)
```

```
{
    ........
    if (socketPool != null &amp;amp;&amp;amp; !socketPool.isEmpty())
    {
        //该host对应的map中可能包含多个socket对象
        Map&lt;SockIO, Integer&gt; aSockets = socketPool.get(host);


    //fast check
    SockIO socket = fastPool.get(host);
    if (socket != null)
    {
        if (isFreeSocket(socket,aSockets))
        return socket;
    }

    if (aSockets != null &amp;amp;&amp;amp; !aSockets.isEmpty())
    {       //aSockets中可能会包含一个host的多个socket，随机指定一个
        int start = (random.nextInt() % aSockets.size());
        if (start &lt; 0)  start*= -1;
        int count = 0;
        //下面2个for循环，是对整个host对应的所有生成的socket连接进行遍历，随机遍历
        for (Iterator&lt;SockIO&gt; i = aSockets.keySet().iterator(); i.hasNext();)
        {
            if (count &lt; start){
                i.next();count++;continue;
            }
            socket = i.next(); //从比起始位置start大的第一个socket开始，判断连接是否可用
            if (isFreeSocket(socket,aSockets))
                return socket;
        }
        //如果没有结果，就从第一个socket开始，逐渐到比start小的第一个socket结束，判断是否可用
        for (Iterator&lt;SockIO&gt; i = aSockets.keySet().iterator();i.hasNext();)
        {
            if (count &gt; 0)
            {
                socket = i.next();
                if (isFreeSocket(socket,aSockets))
                    return socket;
                    count--;
            }else break;
        }
    }
}
// create one socket -- let the maint thread take care of creating more
SockIO socket = createSocket(host);
if (socket != null)
{
   addSocketToPool(socketPool, host, socket,SOCKET_STATUS_BUSY,SOCKET_STATUS_BUSY, t
ue);
}
return socket;

}</pre>
```

获取一个连接，希望通过高效的方式获取到一个合适的socket对象，所有采用了取余的一种算法
<blockquote>
线程池的维护MaintThread
</blockquote>
前面说过，初始化连接后，即开始执行维护线程MaintThread，用于在指定时间间隔内维护socke pool，其内部实现其实也是调用SockIOPool的内部方法selfMaint，该方法维护socket连接池的步骤下：

在socketpool中找出需要建立socket的host，并且计算需要建立几个socket，其实就是根据配参数minConn进行计算
为每个host创建需要增加的socket实例，创建成功后放入socketpool中
计算所有的空闲状态的活动socket，并且计算每个host的多余的空闲socket实例个数，即大于maConn的，同时将这些socket状态置为SOCKET_STATUS_DEAD
清理socketpool中所有状态为SOCKET_STATUS_DEAD的socket，从socketpool中删除，当然先关闭socket再删除

<blockquote>
总结
</blockquote>
在基于NIO的基础上，该客户端实现了socket的灵活管理，使用多线程做连接池的定期维护，使连接池始终保持在高校可用的状态；使用合理的包装，以符合分布式memcached缓存的实现需求。