**实力讲解Tomcat NIO模式如何最大化压榨CPU**

2018-03-28 21:23

<http://www.sohu.com/a/226624448_411876>

**作者介绍**

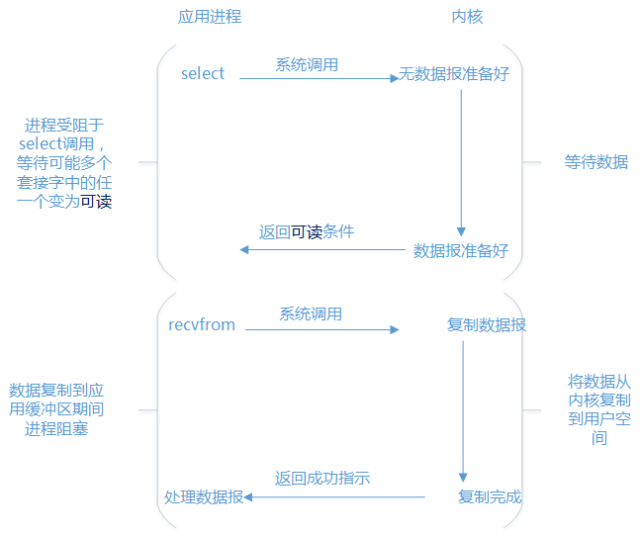
**王新栋**，从事京东京麦平台的架构设计与开发工作。熟悉各种开源软件架构，在web开发，架构优化上有较丰富的实战经历。有多年在NIO领域的设计、开发经验，对HTTP、TCP长连接技术有深入研究与领悟，目前主要致力于移动与PC平台网关技术的优化与实现。

个人公众号：程序架道(xindongbook17)，本文经作者同意授权转载。

**一、I/O复用模型解读**

Tomcat的NIO是基于I/O复用来实现的。对这点一定要清楚，不然我们的讨论就不在一个逻辑线上。

下面这张图学习过I/O模型知识的一般都见过，出自《UNIX网络编程》，I/O模型一共有阻塞式I/O、非阻塞式I/O、I/O复用(select/poll/epoll)、信号驱动式I/O和异步I/O。**这篇文章讲的是I/O复用。**

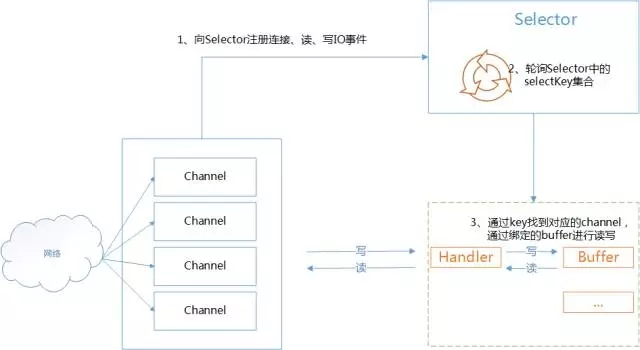


图源：《UNIX网络编程》

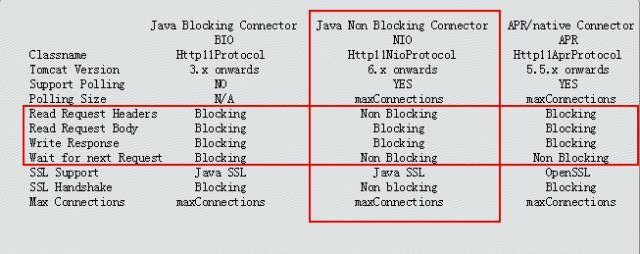
这里先来说下用户态和内核态。直白来讲，如果线程执行的是用户代码，当前线程处在用户态，如果线程执行的是内核里面的代码，当前线程处在内核态。更深层来讲，操作系统为代码所处的特权级别分了4个级别。不过现代操作系统只用到了0和3两个级别。0和3的切换就是用户态和内核态的切换。更详细的可参照书籍《深入理解计算机操作系统》。

**I/O复用模型，是同步非阻塞，这里的非阻塞是指I/O读写，对应的是recvfrom操作，**因为数据报文已经准备好，无需阻塞。说它是同步，是因为这个执行是在一个线程里面执行的。有时还会说它又是阻塞的，实际上是指阻塞在select上面，必须等到读就绪、写就绪等网络事件。有时我们又说I/O复用是多路复用，这里的多路是指N个连接，每一个连接对应一个channel，或者说多路就是多个channel。复用，是指多个连接复用了一个线程或者少量线程(在Tomcat中是Math.min(2,Runtime.getRuntime().availableProcessors()))。

上面提到的网络事件有连接就绪、接收就绪、读就绪、写就绪四个网络事件。**I/O复用主要是通过Selector复用器来实现的，**可以结合下面这个图理解上面的叙述。



二、Tomcat对IO模型的支持



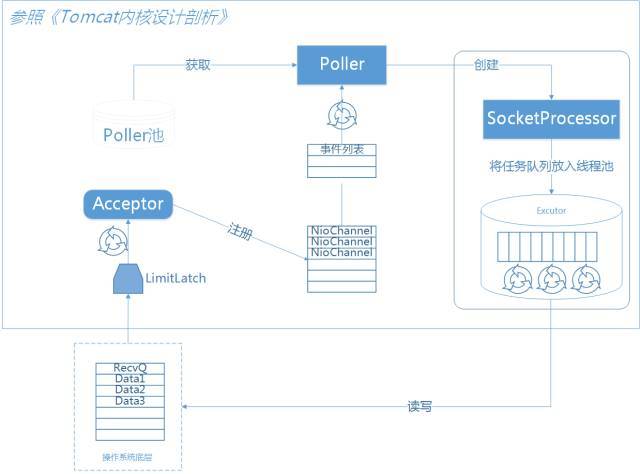
Tomcat从6以后开始支持NIO模型，实现是基于JDK的java.nio包。这里可以看到对read body 和response body是Blocking的。关于这点在下文第6.3节源代码阅读重点介绍。

三、Tomcat中NIO的配置与使用

在Connector节点配置protocol="org.apache.coyote.http11.Http11NioProtocol"，Http11NioProtocol协议下默认最大连接数是10000，也可以重新修改maxConnections的值，同时我们可以设置最大线程数maxThreads，这里设置的最大线程数就是Excutor的线程池的大小。

在BIO模式下实际上是没有maxConnections，即使配置也不会生效，BIO模式下的maxConnections是保持跟maxThreads大小一致，因为它是一请求一线程模式。

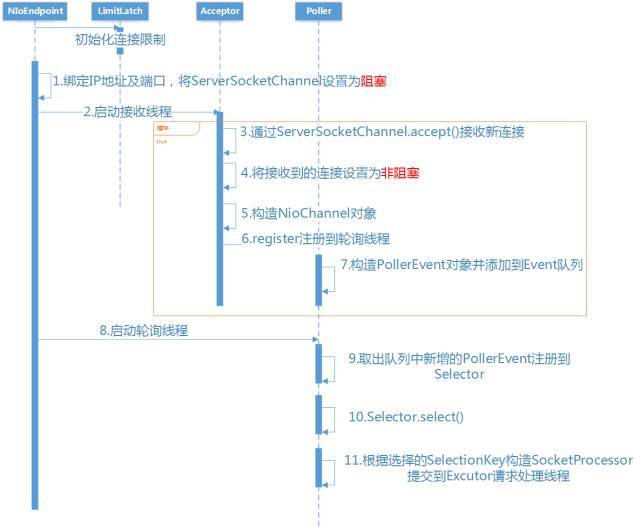
四、NioEndpoint组件关系图解读



我们要理解Tomcat的NIO最主要就是对NioEndpoint的理解。它一共包含LimitLatch、Acceptor、Poller、SocketProcessor、Excutor5个部分。LimitLatch是连接控制器，它负责维护连接数的计算，nio模式下默认是10000，达到这个阈值后，就会拒绝连接请求。Acceptor负责接收连接，默认是1个线程来执行，将请求的事件注册到事件列表。有Poller来负责轮询，Poller线程数量是cpu的核数Math.min(2,Runtime.getRuntime().availableProcessors())。由Poller将就绪的事件生成SocketProcessor同时交给Excutor去执行。Excutor线程池的大小就是我们在Connector节点配置的maxThreads的值。在Excutor的线程中，会完成从socket中读取http request，解析成HttpServletRequest对象，分派到相应的servlet并完成逻辑，然后将response通过socket发回client。

在从socket中读数据和往socket中写数据的过程，并没有像典型的非阻塞的NIO的那样，注册OP\_READ或OP\_WRITE事件到主Selector，而是直接通过socket完成读写，这时是阻塞完成的，但在timeout控制上使用了NIO的Selector机制，但这个Selector并不是Poller线程维护的主Selector，而是BlockPoller线程中维护的Selector，称之为辅Selector。详细源代码可以参照下面第6.3节。

五、NioEndpoint执行序列图

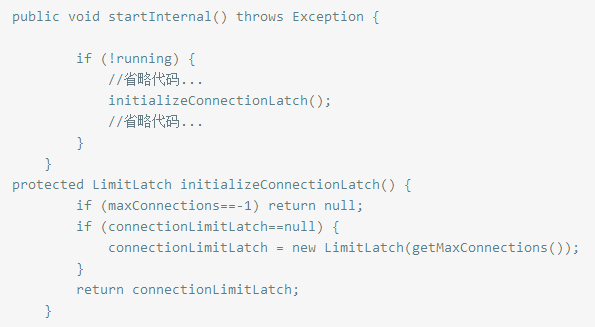


在下一小节NioEndpoint源码解读中我们将对步骤1-步骤11依次找到对应的代码来说明。

六、NioEndpoint源码解读

1、初始化

无论是BIO还是NIO，开始都会初始化连接限制，不可能无限增大，NIO模式下默认是10000。



2、步骤解读

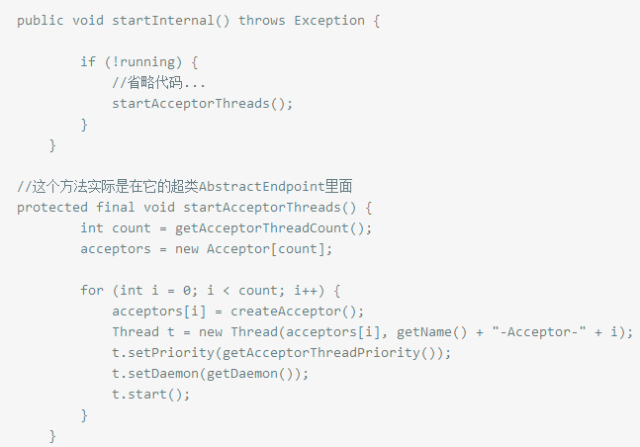
下面我们着重叙述跟NIO相关的流程，共分为11个步骤，分别对应上面序列图中的步骤。

步骤1：绑定IP地址及端口，将ServerSocketChannel设置为阻塞。

这里为什么要设置成阻塞呢，我们一直都在说非阻塞。Tomcat的设计初衷主要是为了操作方便。这样这里就跟BIO模式下一样了。只不过在BIO下这里返回的是Socket，NIO下这里返回的是SocketChannel。



步骤2：启动接收线程



步骤3：ServerSocketChannel.accept()接收新连接



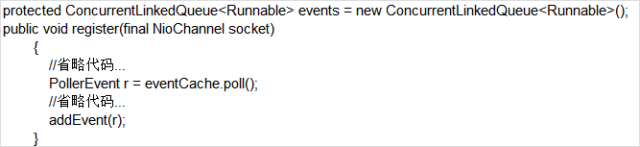
步骤4：将接收到的链接通道设置为非阻塞

步骤5：构造NioChannel对象

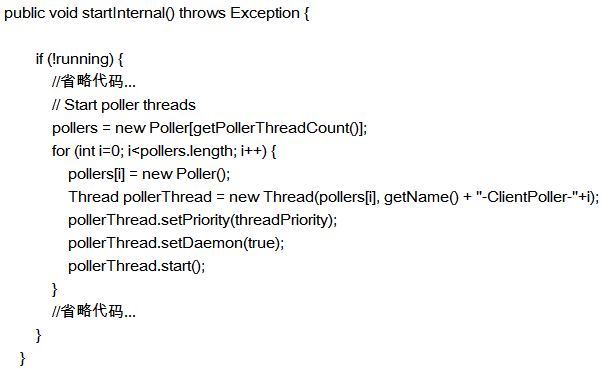
步骤6：register注册到轮询线程



步骤7：构造PollerEvent，并添加到事件队列



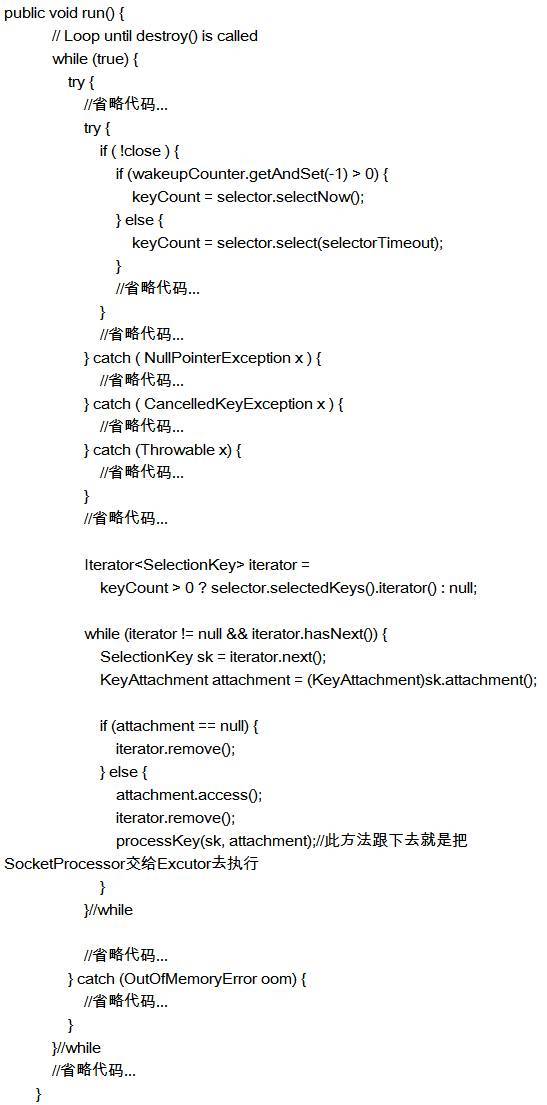
步骤8：启动轮询线程



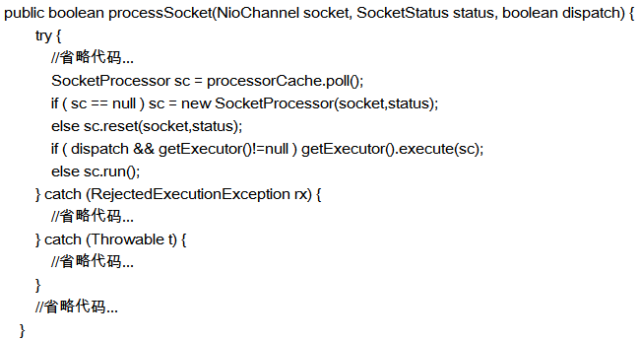
步骤9：取出队列中新增的PollerEvent并注册到Selector



步骤10：Selector.select()



步骤11：根据选择的SelectionKey构造SocketProcessor提交到请求处理线程

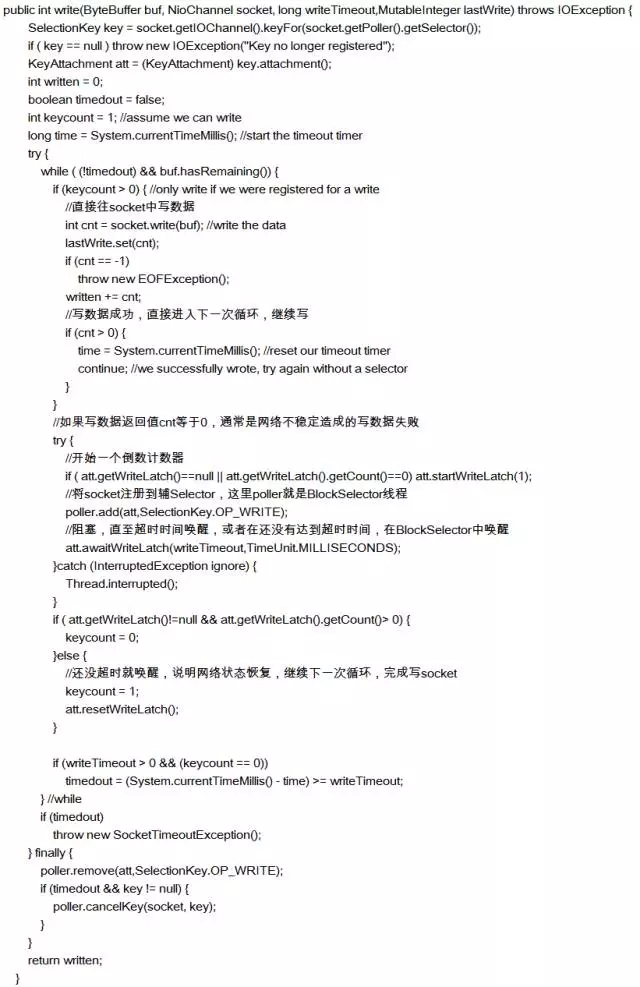


3、NioBlockingSelector和BlockPoller介绍

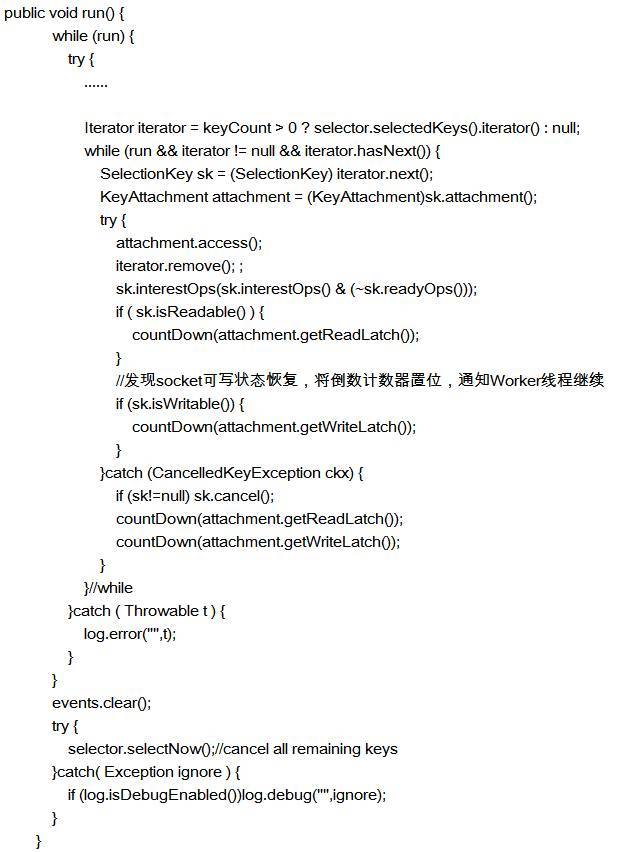
上面的序列图有个地方我没有描述，就是NioSelectorPool这个内部类，是因为在整体理解tomcat的NIO上面，在序列图里面不包括它会更好理解。

在有了上面的基础后，我们在来说下**NioSelectorPool**这个类，对更深层了解Tomcat的NIO一定要知道它的作用。

NioEndpoint对象中维护了一个NioSelecPool对象，这个NioSelectorPool中又维护了一个BlockPoller线程，这个线程就是基于辅Selector进行NIO的逻辑。以执行servlet后，得到response，往socket中写数据为例，最终写的过程调用NioBlockingSelector的write方法。代码如下：

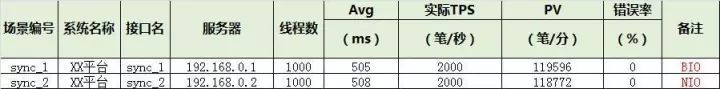


也就是说当socket.write()返回0时，说明网络状态不稳定，这时将socket注册OP\_WRITE事件到辅Selector，由BlockPoller线程不断轮询这个辅Selector，直到发现这个socket的写状态恢复了，通过那个倒数计数器，通知Worker线程继续写socket动作。看一下BlockSelector线程的代码逻辑：



使用这个辅Selector主要是减少线程间的切换，同时还可减轻主Selector的负担。

七、关于性能





这份报告是我们压测的一个结果，跟想象的是不是不太一样？几乎没有差别，实际上NIO优化的是I/O的读写，如果瓶颈不在这里的话，比如传输字节数很小的情况下，BIO和NIO实际上是没有差别的。**NIO的优势更在于用少量的线程hold住大量的连接。**

还有一点，我们在压测的过程中，遇到在NIO模式下刚开始的一小段时间内容，会有错误，这是因为一般的压测工具是基于一种长连接，也就是说比如模拟1000并发，那么同时建立1000个连接，下一时刻再发送请求就是基于先前的这1000个连接来发送，还有TOMCAT的NIO处理是有POLLER线程来接管的，它的线程数一般等于CPU的核数，如果一瞬间有大量并发过来，POLLER也会顿时处理不过来。

八、总结

NIO只是优化了网络IO的读写，如果系统的瓶颈不在这里，比如每次读取的字节说都是500b，那么BIO和NIO在性能上没有区别。**NIO模式是最大化压榨CPU，把时间片都更好利用起来。**对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系统的一些资源如内存，有关线程资源可参照这篇文章《一台Java服务器可以跑多少个线程》。

（http://www.jianshu.com/p/f1930596947d）

因此，使用的线程越少越好。而I/O复用模型正是利用少量的线程来管理大量的连接。在对于维护大量长连接的应用里面更适合用基于I/O复用模型NIO，比如web qq这样的应用。**所以我们要清楚系统的瓶颈是I/O还是CPU的计算。**

**参考资料**

http://tomcat.apache.org/tomcat-7.0-doc/config/http.html

http://gearever.iteye.com/blog/1844203

《Tomcat内核设计剖析》

《深入理解计算机操作系统》

《UNIX网络编程》卷1