三、Tomcat中BIO和NIO底层原理实现对比

在Tomcat7中，默认为BIO，可以通过如下配置改为NIO

```
<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"
connectionTimeout="20000" redirectPort="8443" />
```

BIO

BIO的模型比较简单。

1. JioEndpoint中的Acceptor线程负责循环**阻塞接收socket连接**
2. 每接收到一个socket连接就包装成SocketProcessor扔进线程池Executor中，SocketProcessor是一个Runnable
3. SocketProcessor负责从socket中**阻塞读取**数据，并且向socket中**阻塞写入**数据

Acceptor线程的数量默认为1个，可以通过acceptorThreadCount参数进行配置
线程池Executor是可以配置的，比如：

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
        maxThreads="150" minSpareThreads="4"/>

<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"
        connectionTimeout="20000"
        redirectPort="8443" executor="tomcatThreadPool"/>
```

从上面的配置可以看到，每个Connector可以对应一个线程池，默认情况下，Tomcat中每个Connector都会创建一个自己的线程池，并且该线程池的默认值为：

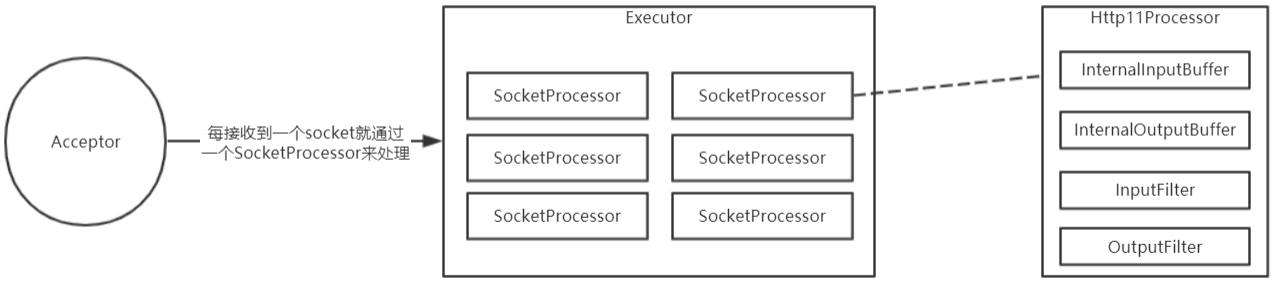
1. 最小线程数量为10
2. 最大线程数量为200

如果两个Connector配置的executor是一样的话，就表示这两个Connector公用一个线程池。

使用BIO来处理请求时，我们可以总结一下：

1. 当请求数量比较大时，可以提高Acceptor线程的数量，提高接收请求的速率
2. 当请求比较耗时是，可以提高线程池Executor的最大线程数量

当然，增加线程的目的都是为了提高Tomcat的性能，但是一台机器的线程数量并不是越多越好，需要利用压测来最终确定一个更加符合当前业务场景的线程数量。



NIO

NIO最大的特性就是非阻塞，非阻塞接收socket连接，非阻塞从socket中读取数据，非阻塞从将数据写到socket中。

但是在Tomcat7中，只有在从socket中读取请求行，请求头数据时是非阻塞的，在读取请求体是阻塞的，响应数据时也是阻塞的。

为什么不全是非阻塞的呢？因为Tomcat7对应Servlet3.0，Servlet3.0规范中没有考虑NIO，比如我们读取请求体的代码得这么写：

```
ServletInputStream inputStream = req.getInputStream();
byte[] bytes = new byte[1024];
int n;
while ((n = inputStream.read(bytes)) > 0) {
    System.out.println(new String(bytes, 0, n));
}
```

inputStream.read()方法的含义就是阻塞读取数据，当读取请求体时，如果操作系统中还没有准备好，那么read方法就得阻塞。

而NIO则不一样，NIO中是一旦操作系统中的数据准备好了，那么则会**通知**Java程序可以读取数据了，这里的**通知**很重要，这决定了我们的Java代码到底如何实现，如果在Servlet中想利用NIO去读取数据，那么在Servlet中肯定就要去监听是否有**通知**过来，比如在Servlet3.1中则增加了NIO相关的定义，如下面代码：

```
ServletInputStream inputStream = req.getInputStream();
inputStream.setReadListener(new ReadListener() {
  // 有数据可用时触发
  @Override
  public void onDataAvailable() throws IOException {

  }
  // 数据全部读完了
  @Override
  public void onAllDataRead() throws IOException {

  }
  // 出现异常了
  @Override
  public void onError(Throwable throwable) {

  }
});
```

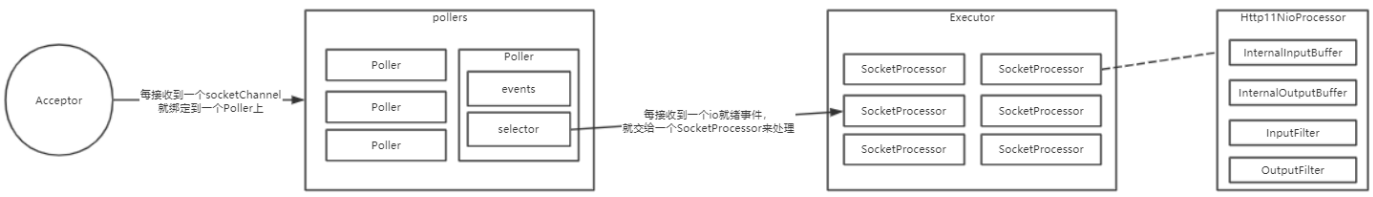
很明显可以看出来，这里有Listener，用来监听数据可读的通知，这才是真正的利用了NIO。

而我们这里暂时只分析Tomcat7中的NIO，Servlet3.1是在Tomcat8中实现的，所以敬请期待吧。

首先我们来看一下Tomcat7中使用NIO处理请求的基本流程：

1. 利用Acceptor来**阻塞**获取socket连接，NIO中叫socketChannel
2. 接收到socketChannel后，需要将socketChannel绑定到一个Selector中，并注册读事件，另外，基于NIO还需要一个线程来轮询Selector中是否存在就绪事件，如果存在就将就绪事件查出来，并处理该事件，在Tomcat中支持多个线程同时查询是否存在就绪事件，该线程对象为Poller，每个Poller中都包含一个Selector，这样每个Poller线程就负责轮询自己的Selector上就绪的事件，然后处理事件。

3. 当Acceptro接收到一个socketChannel后，就会将socketChannel注册到某一个Poller上，确定Polloer的逻辑非常简单，假设现在有3个Poller，编号为1,2,3，那么Tomcat接收到的第一个socketChannel注册到1号Poller上，第二个socketChannel注册到2号Poller上，第三个socketChannel注册到3号Poller上，第四个socketChannel注册到1号Poller上，依次循环。
4. 在某一个Poller中，除开有selector外，还有一个ConcurrentLinkedQueue队列events，events表示待执行事件，比如Tomcat要socketChannel注册到selector上，但是Tomcat并没有直接这么做，而是先自己生成一个PollerEvent，然后把PollerEvent加入到队列events中，然后这个队列中的事件会在Poller线程的循环过程中真正执行
5. 上面说了，Poller线程中需要循环查询selector中是否存在就绪事件，而Tomcat在真正查询之前会先看一下events队列中是否存在待执行事件，如果存在就会先执行，这些事件表示需要向selector上注册事件，比如注册socketChannel的读事件和写事件，所以在真正执行events队列中的事件时就会真正的向selector上注册事件。所以只有先执行events队列中的PollerEvent，Poller线程才能有机会从selector中查询到就绪事件
6. 每个Poller线程一旦查询到就绪事件，就会去处理这些事件，事件无非就是读事件和写事件
7. 处理的第一步就是获取当前就绪事件对应的socketChannel，因为我们要向socketChannel中读数据或写数据
8. 处理的第二步就是把socketChannel和当前要做的事情（读或写）封装为SocketProcessor对象
9. 处理的第三步就是把SocketProcessor扔进线程池进行处理
10. 在SocketProcessor线程运行时，就会从socketChannel读取数据（假设当前处理的是读事件），并且是非阻塞读
11. 既然是非阻塞读，大概的一个流程就是，某一个Poller中的selector查询到了一个读就绪事件，然后交给一个SocketProcessor线程进行处理，SocketProcessor线程读取数据之后，如果发现请求行和请求头的数据都已经读完了，并解析完了，那么该SocketProcessor线程就会继续把解析后的请求交给Servlet进行处理，Servlet中可能会读取请求体，可能会响应数据，而不管是读请求体还是响应数据都是阻塞的，直到Servlet中的逻辑都执行完后，SocketProcessor线程才会运行结束。假如SocketProcessor读到了数据之后，发现请求行或请求头的数据还没有读完，那么本次读事件处理完毕，需要Poller线程再次查询到就绪读事件才能继续读数据，以及解析数据
12. 实际上Tomcat7中的非阻塞读就只是在读取请求行和请求体数据时才是非阻塞的，至于请求体的数据，是在Servlet中通过inputstream.read()方法获取时才会真正的去获取请求体的数据，并且是阻塞的。



接下来我们来看下Tomcat7中是怎么阻塞的利用NIO来读取数据的。

当Servlet中通过inputstream.read()来读取请求体数据时，最终执行的是InternalNioInputBuffer.SocketInputBuffer.doRead()方法。

在这个方法中会调用fill(true,true)，第一个参数是timeout，第二个参数是block，block等于true，表示阻塞，fill方法中就会从操作系统读取数据填充到Tomcat的buf中。

在接下来的阻塞读取数据流程中，主要利用的还是Selector，为什么阻塞的时候还要利用Selector呢？这是因为，socketChannel一开始是非阻塞的，我们现在如果想把它改成阻塞的，在NIO里是有一个限制的，如果一个socketChannel被设置成了非阻塞的，然后注册了事件，然后又想把socketChannel设置成阻塞的，这时是会抛异常的。所以在Tomcat中是使用的另外的方式来达到阻塞效果的。

所以现在的目的是，仍然基于Selector的情况下达到阻塞效果，为了达到这个效果，原理也不难。

在我们需要读取请求体数据时，不能直接利用之前的主Selector了（主Selector就是用来注册新socketChannel的），所以我们需要一个辅助Selector，在读取请求体数据时，新生成一个辅助Selector，这个辅助Selector用来监听当前请求的读事件，当有数据就绪时，辅助Selector就会查询到此次就绪事件（注意：这个时候主Selector是监听不到的，因为在这之前主Selector已经取消了对当前socketChannel的事件）。

这是辅助Selector的主要作用，具体流程如下：

1. inputstream.read()
2. 向辅助Selector注册读事件
3. 加锁（目的是达到阻塞）
4. 与辅助Selector对应的有另外一个辅助Poller，辅助Poller负责轮询辅助Selector上发生的就绪事件，一旦轮询到就绪事件就会解锁，从而解阻塞
5. 从socketChannel中读数据
6. 返回，本次read结束

默认情况下，辅助Selector是NioBlockingSelector对象，每次read都使用同一个NioBlockingSelector对象，在NioBlockingSelector对象中存在一个BlockPoller线程，BlockPoller就是辅助Poller。

对于响应也是类似的思路，也是先注册写事件，阻塞，都有写就绪事件时就解阻塞，开始写入数据。

总结

在Tomcat7，虽然有NIO，但是不够彻底，相比如BIO，优点仅限于能利用较少的线程同时接收更多的请求，但是在真正处理请求时，想比如BIO并没有太多的优势，如果在处理一个请求时既不用读取请求，也不需要响应很多的数据那么NIO模式还是会拥有更大的吞吐量，所以如果要优化的话，将BIO改成NIO也是可以的。