很多开源应用服务器都是集成tomcat作为web container的，而且对于tomcat的servlet container这部分代码很少改动。这样，这些应用服务器的性能基本上就取决于Tomcat处理HTTP请求的connector模块的性能。本文首先从应用层次分析了tomcat所有的connector种类及用法，接着从架构上分析了connector模块在整个tomcat中所处的位置，最后对connector做了详细的源代码分析。并且我们以Http11NioProtocol为例详细说明了tomcat是如何通过实现ProtocolHandler接口而构建connector的。

通过本文的学习，应该可以轻松做到将tomcat做为web container集成到第三方系统，并且自定义任何你想要的高性能的HTTP连接器。

**1 Connector介绍**

**1.1 Connector的种类**

Tomcat源码中与connector相关的类位于org.apache.coyote包中，Connector分为以下几类：

* Http Connector, 基于HTTP协议，负责建立HTTP连接。它又分为BIO Http Connector与NIO Http Connector两种，后者提供非阻塞IO与长连接Comet支持。
* AJP Connector, 基于AJP协议，AJP是专门设计用来为tomcat与http服务器之间通信专门定制的协议，能提供较高的通信速度和效率。如与Apache服务器集成时，采用这个协议。
* APR HTTP Connector, 用C实现，通过JNI调用的。主要提升对静态资源（如HTML、图片、CSS、JS等）的访问性能。现在这个库已独立出来可用在任何项目中。Tomcat在配置APR之后性能非常强劲。

**1.2 Connector的配置**

对Connector的配置位于conf/server.xml文件中。

**1.2.1 BIO HTTP/1.1 Connector配置**

一个典型的配置如下：

<Connector port=”8080” protocol=”**HTTP/1.1**” maxThreads=”150”

connectionTimeout=”20000” redirectPort=”8443”

其它一些重要属性如下：

* acceptCount : 接受连接request的最大连接数目，默认值是10
* address : 绑定IP地址，如果不绑定，默认将绑定任何IP地址
* allowTrace : 如果是true,将允许TRACE HTTP方法
* compressibleMimeTypes : 各个mimeType, 以逗号分隔，如text/html,text/xml
* compression : 如果带宽有限的话，可以用GZIP压缩
* connectionTimeout : 超时时间，默认为60000ms (60s)
* maxKeepAliveRequest : 默认值是100
* maxThreads : 处理请求的Connector的线程数目，默认值为200

如果是SSL配置，如下：

<Connector port="8181" protocol="HTTP/1.1" SSLEnabled="true"

maxThreads="150" scheme="https" secure="true"

clientAuth="false" sslProtocol = "TLS"

address="0.0.0.0"

**keystoreFile="E:/java/jonas-full-5.1.0-RC3/conf/keystore.jks"**

**keystorePass="changeit"** />

其中，keystoreFile为证书位置，keystorePass为证书密码

**1.2.2 NIO HTTP/1.1 Connector配置**

<Connector port=”8080” protocol=”**org.apache.coyote.http11.Http11NioProtocol**”

maxThreads=”150” connectionTimeout=”20000” redirectPort=”8443”

**1.2.3 Native APR Connector配置**

1. ARP是用C/C++写的，对静态资源（HTML，图片等）进行了优化。所以要下载本地库

tcnative-1.dll与openssl.exe，将其放在%tomcat%\bin目录下。

下载地址是：<http://tomcat.heanet.ie/native/1.1.10/binaries/win32/>

1. 在server.xml中要配置一个Listener,如下图。这个配置tomcat是默认配好的。
2. <!--APR library loader. Documentation at /docs/apr.html -->

<Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />

1. 配置使用APR connector
2. <Connector port=”8080” protocol=”**org.apache.coyote.http11.Http11AprProtocol**”

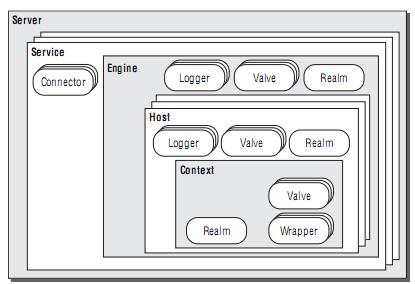
maxThreads=”150” connectionTimeout=”20000” redirectPort=”8443”

1. 如果配置成功，启动tomcat,会看到如下信息：

org.apache.coyote.http11.Http11AprProtocol init

**2 Connector在Tomcat中所处的位置**

**2.1 Tomcat架构**



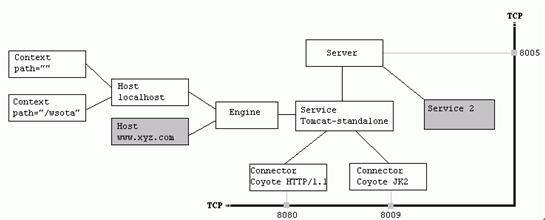
**图2-1 Tomcat架构**

* Server(服务器)是Tomcat构成的顶级构成元素，所有一切均包含在Server中，Server的实现类StandardServer可以包含一个到多个Services;
* 次顶级元素Service的实现类为StandardService调用了容器(Container)接口，其实是调用了Servlet Engine(引擎)，而且StandardService类中也指明了该Service归属的Server；
* 接下来次级的构成元素就是容器(Container)，主机(Host)、上下文(Context)和引擎(Engine)均继承自Container接口，所以它们都是容器。但是，它们是有父子关系的，在主机(Host)、上下文(Context)和引擎(Engine)这三类容器中，引擎是顶级容器，直接包含是主机容器，而主机容器又包含上下文容器，所以引擎、主机和上下文从大小上来说又构成父子关系，虽然它们都继承自Container接口。
* 连接器(Connector)将Service和Container连接起来，首先它需要注册到一个Service，它的作用就是把来自客户端的请求转发到Container(容器)，这就是它为什么称作连接器的原因。

故我们从功能的角度将Tomcat源代码分成5个子模块，它们分别是：

1. Jsper子模块：这个子模块负责jsp页面的解析、jsp属性的验证，同时也负责将jsp页面动态转换为java代码并编译成class文件。在Tomcat源代码中，凡是属于org.apache.jasper包及其子包中的源代码都属于这个子模块；
2. Servlet和Jsp规范的实现模块：这个子模块的源代码属于javax.servlet包及其子包，如我们非常熟悉的javax.servlet.Servlet接口、javax.servet.http.HttpServlet类及javax.servlet.jsp.HttpJspPage就位于这个子模块中；
3. Catalina子模块：这个子模块包含了所有以org.apache.catalina开头的java源代码。该子模块的任务是规范了Tomcat的总体架构，定义了Server、Service、Host、Connector、Context、Session及Cluster等关键组件及这些组件的实现，这个子模块大量运用了Composite设计模式。同时也规范了Catalina的启动及停止等事件的执行流程。从代码阅读的角度看，这个子模块应该是我们阅读和学习的重点。
4. Connectors子模块：如果说上面三个子模块实现了Tomcat应用服务器的话，那么这个子模块就是Web服务器的实现。所谓连接器(Connector)就是一个连接客户和应用服务器的桥梁，它接收用户的请求，并把用户请求包装成标准的Http请求(包含协议名称，请求头Head，请求方法是Get还是Post等等)。同时，这个子模块还按照标准的Http协议，负责给客户端发送响应页面，比如在请求页面未发现时，connector就会给客户端浏览器发送标准的Http 404错误响应页面。
5. Resource子模块：这个子模块包含一些资源文件，如Server.xml及Web.xml配置文件。严格说来，这个子模块不包含java源代码，但是它还是Tomcat编译运行所必需的。

**2.2 Tomcat运行流程**



**图2-2 tomcat运行流程**

假设来自客户的请求为：<http://localhost:8080/test/index.jsp>

1. 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得
2. Connector把该请求交给它所在的Service的Engine来处理，并等待Engine的回应
3. Engine获得请求localhost:8080/test/index.jsp，匹配它所有虚拟主机Host
4. Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）
5. localhost Host获得请求/test/index.jsp，匹配它所拥有的所有Context
6. Host匹配到路径为/test的Context（如果匹配不到就把该请求交给路径名为""的Context去处理）
7. path="/test"的Context获得请求/index.jsp，在它的mapping table中寻找对应的servlet
8. Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类
9. 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法
10. Context把执行完了之后的HttpServletResponse对象返回给Host
11. Host把HttpServletResponse对象返回给Engine
12. Engine把HttpServletResponse对象返回给Connector
13. Connector把HttpServletResponse对象返回给客户browser

**3 Connector源码分析**

**3.1 Tomcat的启动分析与集成设想**

我们知道，启动tomcat有两种方式：

* 双击bin/startup.bat
* 运行bin/catalina.bat run

它们对应于Bootstrap与Catalina两个类，我们现在只关心Catalina这个类，这个类使用Apache Digester解析conf/server.xml文件生成tomcat组件，然后再调用Embedded类的start方法启动tomcat。

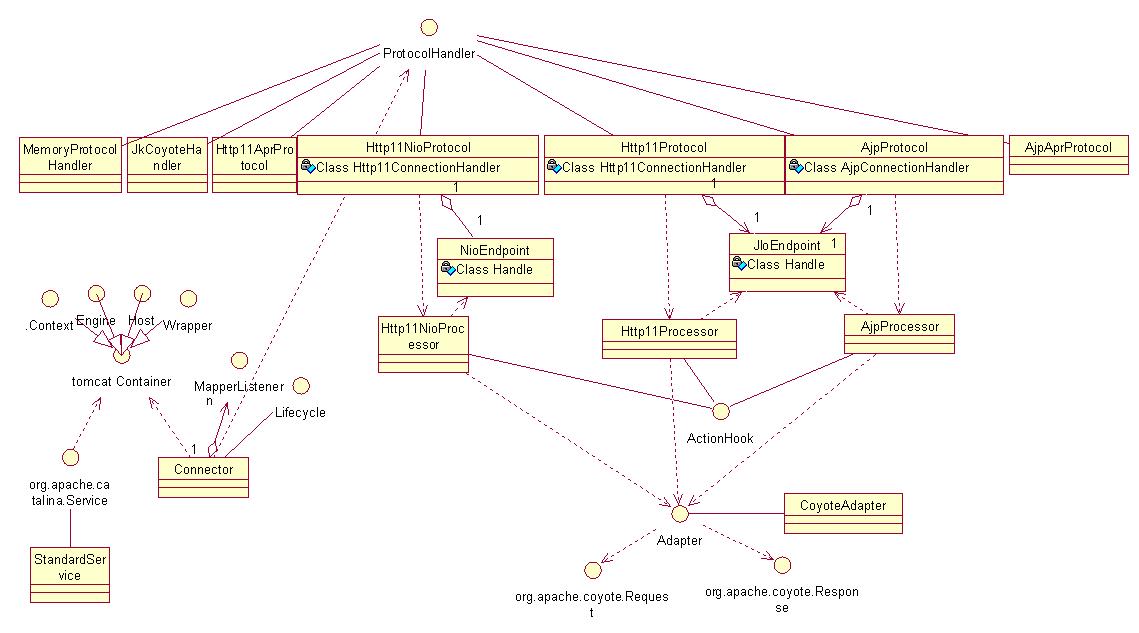
所以，集成Tomcat的方式就有以下两种了：

* 沿用tomcat自身的server.xml
* 自己定义一个xml格式来配置tocmat的各参数，自己再写解析这段xml，然后使用tomcat提供的API根据这些xml来生成Tomcat组件，最后调用Embedded类的start方法启动tomcat

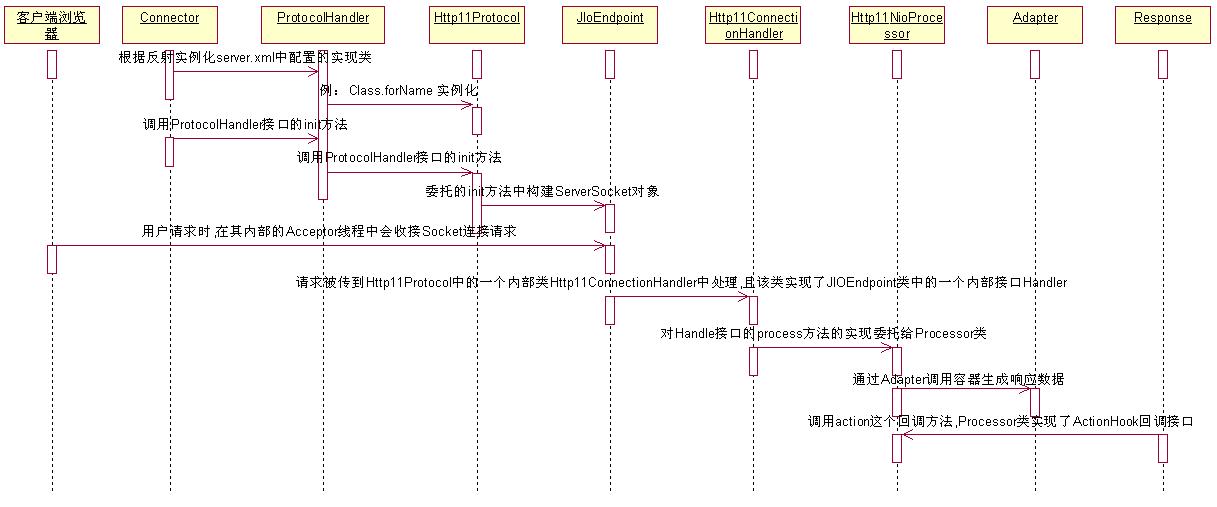
个人觉得第一种方式要优越，给开发者比较好的用户体验，如果使用这种，直接模仿Catalina类的方法即可实现集成。

目前，JOnAS就使用了这种集成方式，JBoss、GlassFish使用的第二种自定义XML的方式。

**3.2 Connector类图与顺序图**



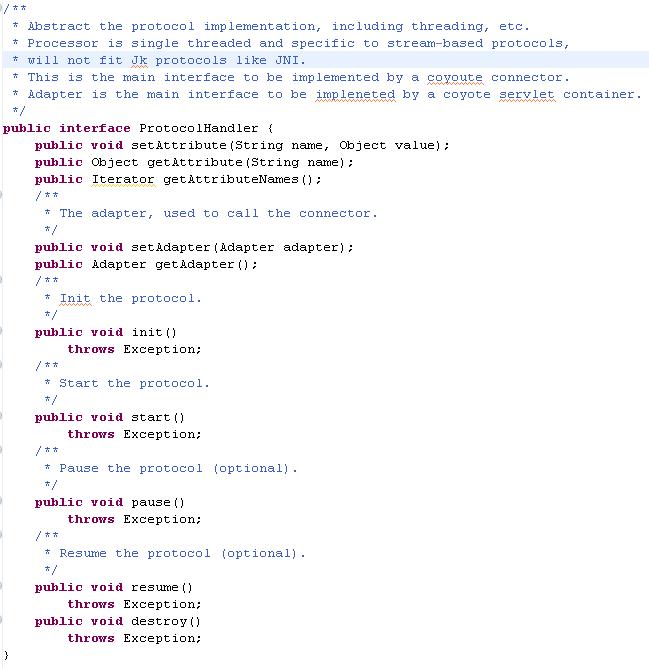
**图3-1 Connector相关类图**



**图3-2 Connector工作流程顺序图**

从上面二图中我们可以得到如下信息：

1. Tomcat中有四种容器(Context、Engine、Host、Wrapper)，前三者常见，第四个不常见但它也是实现了Container接口的容器
2. 如果要**自定义一个Connector的话，只需要实现ProtocolHander接口**,该接口定义如下：



**图3-3 自定义connector时需实现的ProtocolHandler接口**

Tomcat以HTTP（包括BIO与NIO）、AJP、APR、内存四种协议实现了该接口（它们分别是：AjpAprProtocol、AjpProtocol、Http11AprProtocol、Http11NioProtocol、Http11Protocal、JkCoyoteHandler、MemoryProtocolHandler），要使用哪种Connector就在conf/server.xml中配置，在Connector的构造函数中会通过反射实例化所配置的实现类：

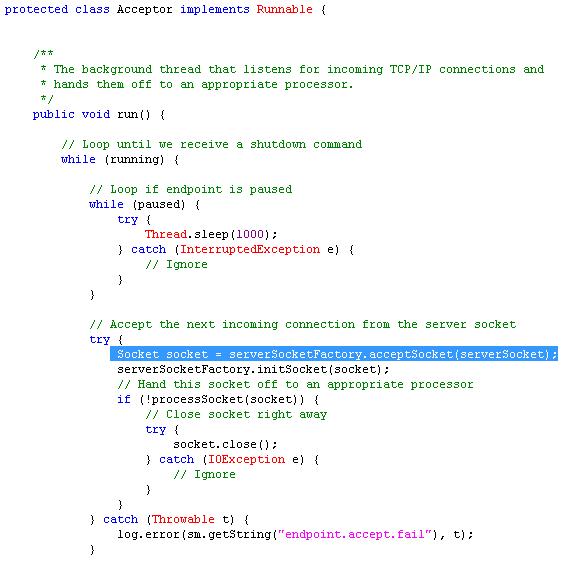
<Connector port="8181"

protocol="org.apache.coyote.http11.Http11AprProtocol " />

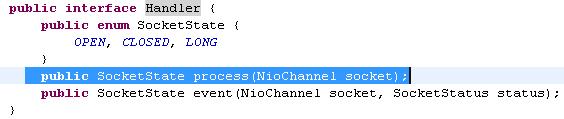
**3.3 Connector的工作流程**

下面我们以Http11AprProtocol为例说明Connector的工作流程。

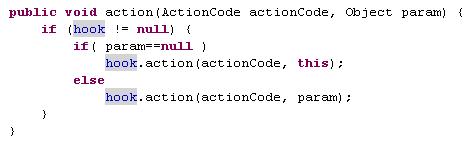
1. 它将工作委托给NioEndpoint类。在NioEndpoint类的init方法中构建一个SocketServer(当然，不同的实现类会有一些微小的变化，例如如果是NIO，它构建的就是SocketServerChannel)
2. 在NioEndpoint.Acceptor类中会接收一个客户端新的连接请求，如下图：



1. 在NioEndpoint类中，有一个内部接口Handle，该接口定义如下：



1. 在Http11NioProtocol类中实现了Handle这个内部接口，并调用Http11NioProcessor类(该类实现了ActionHook回调接口)。在Response类中会调用ActionHook实现类的相关方法的，Response类的action方法如下：



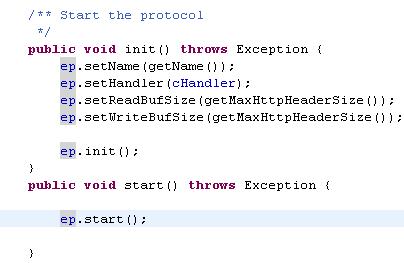
1. Http11NioProcessor的process实现方法中，会通过Adapter来调用Servler容器生成响应结果。

**4 如何实现Connector**

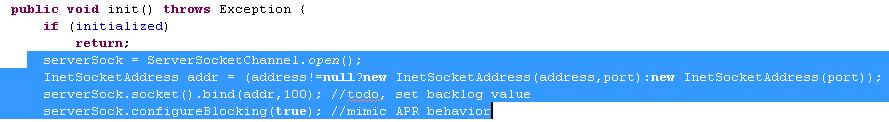
由上面的介绍我们可以知道，实现Connector就是实现ProtocolHander接口的过程。

AjpAprProtocol、AjpProtocol、Http11AprProtocol、Http11Protocol、JkCoyoteHandler、MemoryProtocolHandler这些实现类的实现流程与Http11NioProtocol相同，下面我们以Http11NioProtocol为类重点说明tomcat中如何实现ProtocolHander接口的。

Http11NioProtocol实现了ProtocolHander接口，它将所有的操作委托给NioEndpoint类去做，如下图：



NioEndpoint类中的init方法中首先以普通阻塞方式启动了SocketServer：



NioEndpoint类的start方法是关键，如下：

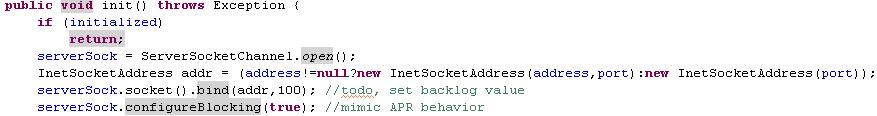


可以看出，在start方法中启动了两个线程和一个线程池：

* Acceptor线程，该线程以普通阻塞方式接收客户端请求（socket.accep()），将客户Socket交由线程池是处理，线程池要将该Socket配置成非阻塞模式（socket.configureBlocking(false)）,并且向Selector注册READ事件。该线程数目可配置，默认为1个。
* Poller线程，由于Acceptor委托线程为客户端Socket注册了READ事件，当READ准备好时，就会进入Poller线程的循环，Poller线程也是委托线程池去做，线程池将NioChannel加入到ConcurrentLinkedQueue<NioChannel>队列中。该线程数目可配置，默认为1个。
* 线程池，就是上面说的做Acceptor与Poller线程委托要做的事情。

**4.1 Init接口实现方法中阻塞方式启动ServerSocketChannel**

在Init接口实现方法中阻塞方式启动ServerSocketChannel。



**4.2 Start接口实现方法中启动所有线程**

Start方法中启动了线程池，acceptor线程与Poller线程。其中acceptor与poller线程一般数目为1，当然，数目也可配置。



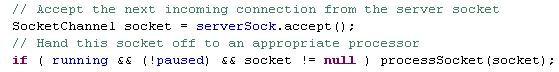
可以看出，线程池有两种实现方式：

* 普通queue + wait + notify方式，默认使用的方式，据说实际测试这种比下种效率高
* JDK1.5自带的线程池方式

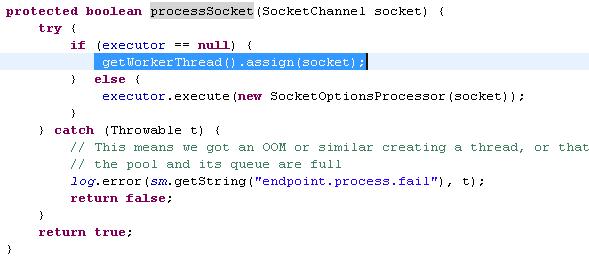
**4.3 Acceptor线程接收客户请求、注册READ事件**

在Acceptor线程中接收了客户请求，同时委托线程池注册READ事件。

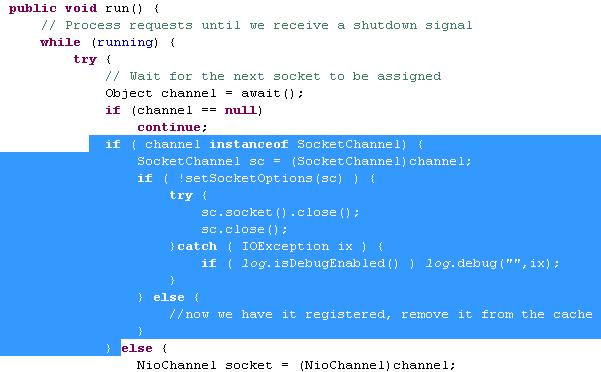
1. 在Acceptior线程中接收了客户请求（serverSock.accept()）



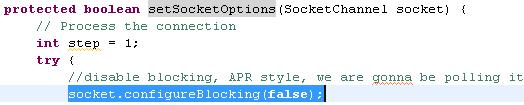
1. 委托线程池处理



1. 在线程池的Worker线程的run方法中有这么几句：



在setSocketOptions方法中，首先将socket配置成非阻塞模式：



在setSocketOptions方法中，最后调用getPoller0().register(channel);一句为SocketChannel注册READ事件，register方法代码如下(注意：这是Poller线程的方法)：

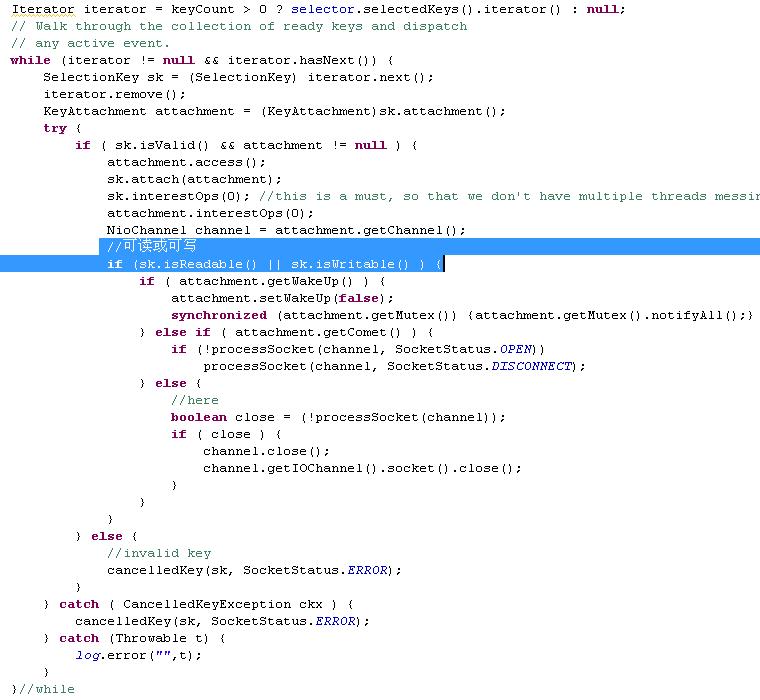


其中attachment的结构如下，它可以看做是一个共享的数据结构：

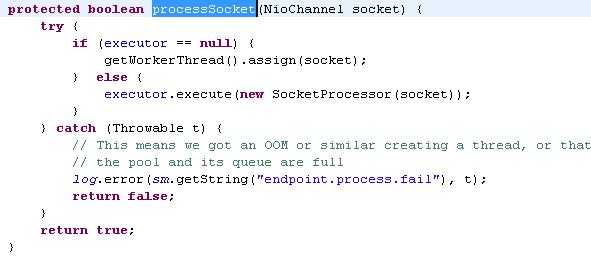


**4.4 Poller线程读请求、生成响应数据、注册WRITE事件**

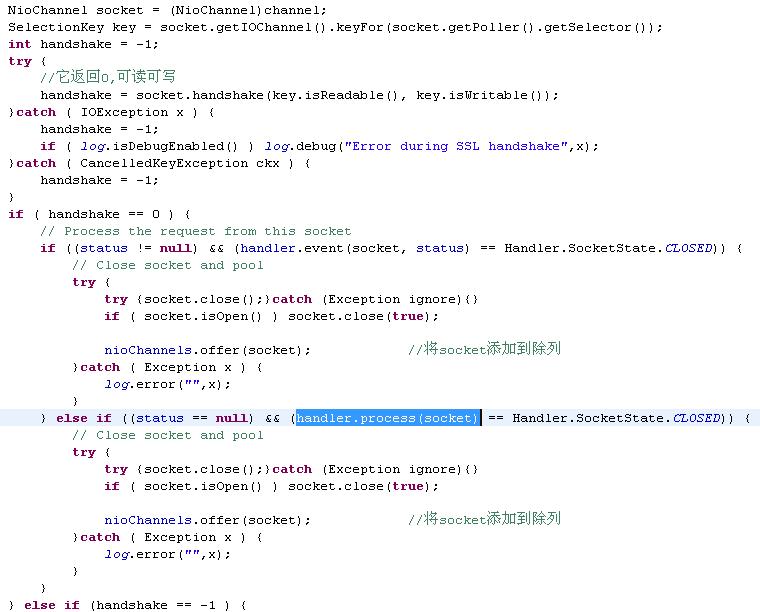
1. 在上面说的setSocketOptions方法中调用Poller线程的register方法注册读事件之后，当READ准备就绪之后，就开始读了。下面代码位于Poller线程的run方法之中：



1. 可以看到，可读之后调用processSocket方法，该方法将读处理操作委拖给线程池处理(注意此时加入到线程池的是NioChannel，不是SocketChannel)：

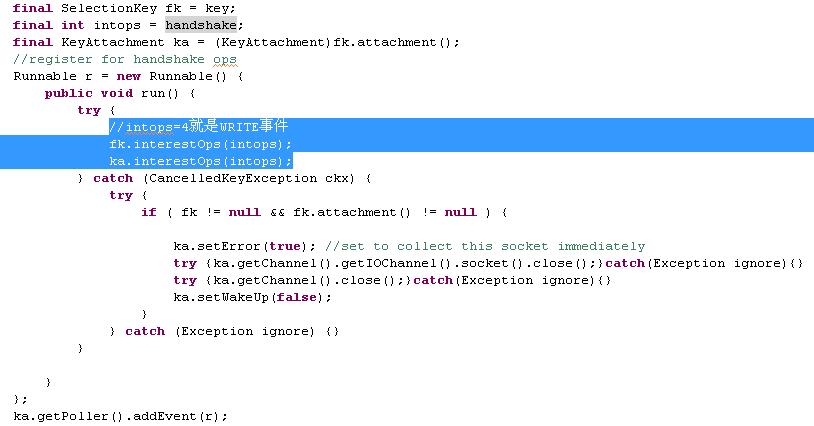


1. 线程池的Worker线程中的run方法中的部分代码如下（请注意handler.process(socket)这一句）：



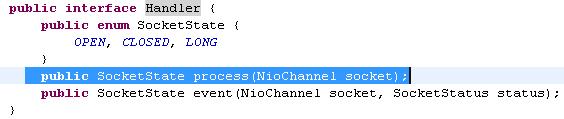
注意：

* + 调用了hanler.process(socket)来生成响应数据）
  + 数据生成完之后，注册WRITE事件的，代码如下：

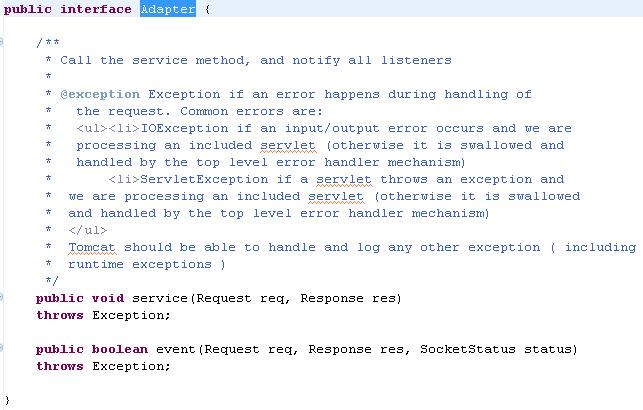


**4.5 Handle接口实现类通过Adpater调用Servlet容器生成响应数据**

NioEndpoint类中的Handler接口定义如下：



其中process方法通过Adapter来调用Servlet Container生成返回结果。Adapter接口定义如下：



**4.6 小结**

实现一个tomcat连接器Connector就是实现ProtocolHander接口的过程。Connector用来接收Socket Client端的请求，通过内置的线程池去调用Servlet Container生成响应结果，并将响应结果同步或异步的返回给Socket Client。在第三方应用集成tomcat作为Web容器时，一般不会动Servlet Container端的代码，那么connector的性能将是整个Web容器性能的关键。