**网络层(tomcat)：**

**NioEndpoint：**

ServerSocketChannelImpl

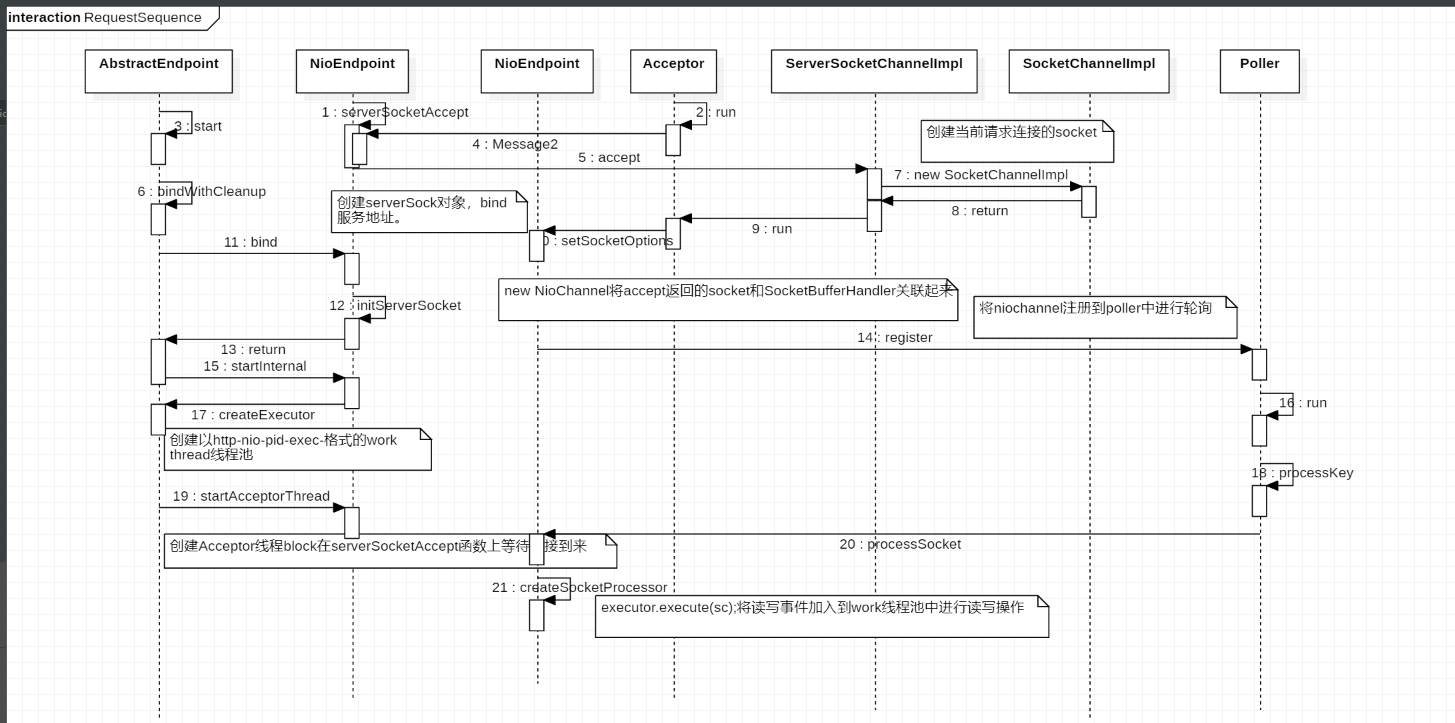
Bind

Listen

Accept

Poller

Workpoolthread



1.createExecutor()函数：

使用ThreadPoolExecutor和LinkedBlockingQueue创建工作线程池，默认最大线程池数量为200。

2.ClientPoller线程：

SynchronizedQueue<PollerEvent> **events**

java.nio.channels.Selector;

线程在**events异步队列中等待**PollerEvent的offer。

3.startAcceptorThread

启动accept线程循环做以下事情：

1.检查**endpoint**是否是运行状态而且不为暂停状态。

2.检查连接的数量是否超过最大连接数。

默认**maxConnections** = 10000

*//if we have reached max connections, wait* **endpoint**.countUpOrAwaitConnection();

**protected void** countUpOrAwaitConnection() **throws** InterruptedException {  
 **if** (**maxConnections**==-1) **return**;  
 LimitLatch latch = **connectionLimitLatch**;  
 **if** (latch!=**null**) latch.countUpOrAwait();  
}

NioEndpoint 中的connectionLimitLatch记录当前的连接数目。

3.Accept ServerSocketChannel是否有请求:

socket = **endpoint**.serverSocketAccept();

@Override  
**protected** SocketChannel serverSocketAccept() **throws** Exception {  
 **return serverSock**.accept();  
}

4.有client请求过来：

**if** (**endpoint**.isRunning() && !**endpoint**.isPaused()) {  
 *// setSocketOptions() will hand the socket off to  
 // an appropriate processor if successful* **if** (!**endpoint**.setSocketOptions(socket)) {  
 **endpoint**.closeSocket(socket);  
 }  
} **else** {  
 **endpoint**.destroySocket(socket);  
}

调用**endpoint**.setSocketOptions的函数对socket进行处理。

setSocketOptions主要做以下事情：

创建NioChannel对象：

NioChannel channel = **null**;  
**if** (**nioChannels** != **null**) {  
 channel = **nioChannels**.pop();  
}  
**if** (channel == **null**) {  
 SocketBufferHandler bufhandler = **new** SocketBufferHandler(  
 **socketProperties**.getAppReadBufSize(),  
 **socketProperties**.getAppWriteBufSize(),  
 **socketProperties**.getDirectBuffer());  
 **if** (isSSLEnabled()) {  
 channel = **new** SecureNioChannel(bufhandler, **selectorPool**, **this**);  
 } **else** {  
 channel = **new** NioChannel(bufhandler);  
 }  
}

为socket创建读写的ByteBuffer，默认大小为8KB,分装成SocketBufferHandler

对象和NioChannel对象，最终将NioChannel封装到NioSocketWrapper中。

NioSocketWrapper socketWrapper = **new** NioSocketWrapper(channel, **this**);  
channel.reset(socket, socketWrapper);  
socketWrapper.setReadTimeout(getConnectionTimeout());  
socketWrapper.setWriteTimeout(getConnectionTimeout());  
socketWrapper.setKeepAliveLeft(NioEndpoint.**this**.getMaxKeepAliveRequests());  
socketWrapper.setSecure(isSSLEnabled());  
**poller**.register(channel, socketWrapper);

**poller**.register主要动作是：

1.创建***OP\_REGISTER***类型的PollerEvent。

2.将PollerEvent添加到SynchronizedQueue，同时唤醒client poller线程阻塞在select

上的轮询选择器(设置1秒超时)，对SynchronizedQueue队列中的PollerEvent进行处理，主要是将socket channel注册到ClientPoller线程中，进行读的轮询操作。

**public void** register(**final** NioChannel socket, **final** NioSocketWrapper socketWrapper) {

**if** (r == **null**) {  
 r = **new** PollerEvent(socket, ***OP\_REGISTER***);  
 } **else** {  
 r.reset(socket, ***OP\_REGISTER***);  
 }  
 addEvent(r);  
}

client poller线程主要作用：

1.将新的连接注册到轮询selector中。

2.为selectkey添加ops。

3.轮询所有连接上产生的事件。

**//**PollerEvent

@Override  
**public void** run() {

**if** (**interestOps** == ***OP\_REGISTER***) {  
 **try** {  
**socket**.getIOChannel().register(**socket**.getSocketWrapper().getPoller().getSelector(), **SelectionKey.*OP\_READ***, **socket**.getSocketWrapper());  
 } **catch** (Exception x) {  
 ***log***.error(***sm***.getString(**"endpoint.nio.registerFail"**), x);  
 }  
}

ClientPoller线程处理读写轮询：

当有读事件的时候返回SelectionKey对象进行处理，最终创建SocketProcessor对象,放入到work pool线程池进行处理：

processKey(sk, socketWrapper);

processSocket(socketWrapper, SocketEvent.***OPEN\_WRITE***, **true**)

Executor executor = getExecutor();

**public boolean** processSocket(SocketWrapperBase<S> socketWrapper,  
 SocketEvent event, **boolean** dispatch) {

**if** (sc == **null**) {  
 sc = createSocketProcessor(socketWrapper, event);  
} **else** {  
 sc.reset(socketWrapper, event);  
}

**if** (dispatch && executor != **null**) {  
 executor.execute(sc);  
} **else** {  
 sc.run();  
}

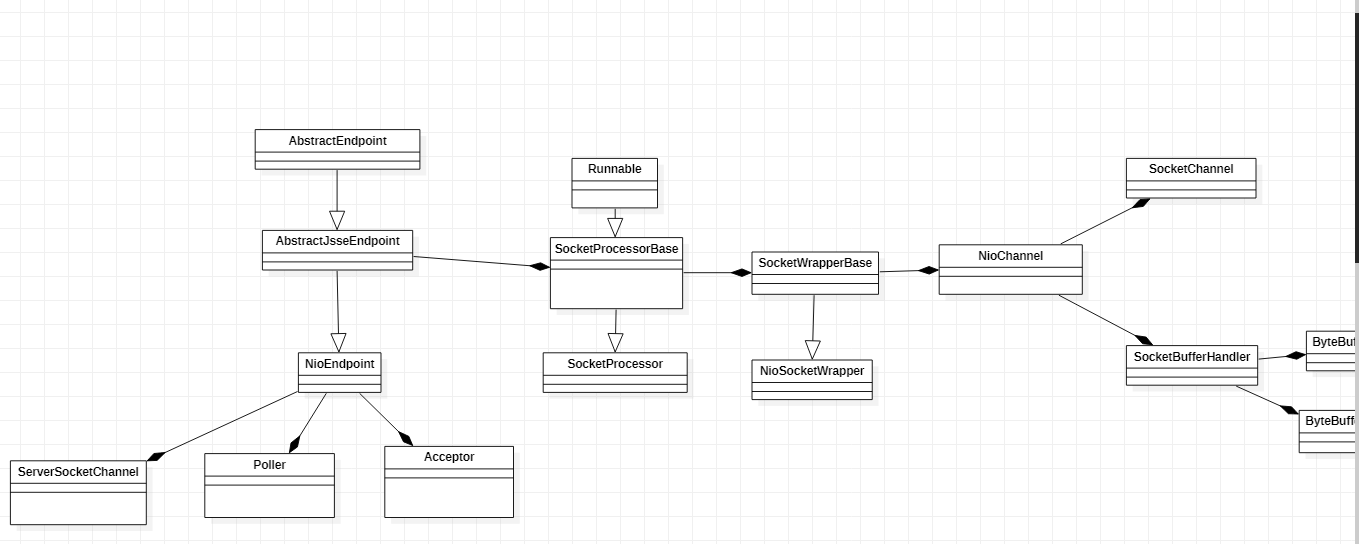
}

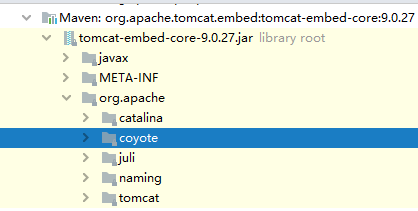
在workpool中运行SocketProcessor的doRun函数：

@Override  
**protected void** doRun() {

**if** (handshake == 0) {  
 SocketState state = SocketState.***OPEN***;  
 *// Process the request from this socket* **if** (**event** == **null**) {  
 state = getHandler().process(**socketWrapper**, SocketEvent.***OPEN\_READ***);  
 } **else** {  
 **state = getHandler().process(socketWrapper, event);** }  
 **if** (state == SocketState.***CLOSED***) {  
 poller.cancelledKey(key, **socketWrapper**);  
 }  
}

NioEndpoint网络层主要的对象：



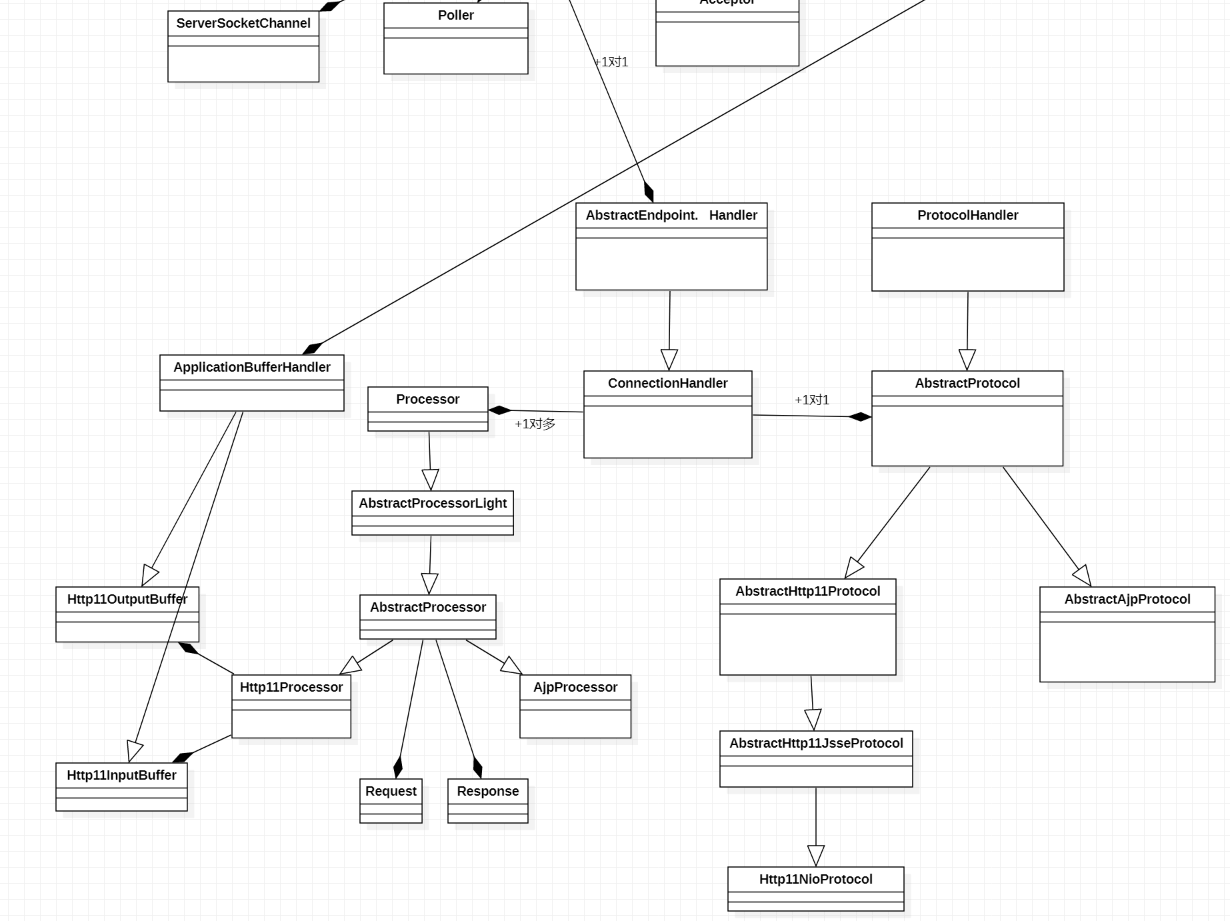


Tomcat 包主要包含的就是网络层的处理逻辑。

**Coyote协议层(handler):**

Coyote包里包含的协议层的逻辑。

主要的作用从niochanel中读取数据解析成http格式的数据。



AbstractProtocol

ConnectionHandler

Process()函数：

1.为请求的连接创建对应协议的Processor，保存在map容器中。

**protected static class** ConnectionHandler<S> **implements** AbstractEndpoint.Handler<S> {

@Override  
**public** SocketState process(SocketWrapperBase<S> wrapper, SocketEvent status) {

Processor processor = **connections**.get(socket);

**if** (processor == **null**) {  
 processor = getProtocol().createProcessor();  
 register(processor);  
 }

*// Associate the processor with the connection*

**connections**.put(socket, processor);

}

}

2.调用AbstractProcessorLight父类的process函数进行处理读事件。

@Override  
**public** SocketState process(SocketWrapperBase<?> socketWrapper, SocketEvent status)**throws** IOException {

**do** {

} **else if** (status == SocketEvent.***OPEN\_READ***) {  
 state = service(socketWrapper);  
 }

} **while** (state == SocketState.***ASYNC\_END*** ||  
 dispatches != **null** && state != SocketState.***CLOSED***);

}

3.调用Http11Processor处理器的service函数处理读请求。

@Override  
**public** SocketState service(SocketWrapperBase<?> socketWrapper)  
 **throws** IOException {

**inputBuffer**.parseRequestLine

**inputBuffer**.parseHeaders()

*// Process the request in the adapter***if** (getErrorState().isIoAllowed()) {  
 **try** {  
 rp.setStage(org.apache.coyote.Constants.***STAGE\_SERVICE***);  
 getAdapter().service(**request**, **response**);

}

}

**Catalina servlet容器层：**

CoyoteAdapter(在Connector初始化的时候创建)：

Server 、Service、Connector、Container、Pipeline、Valve

CoyoteAdapter：

**connector**.getService().getContainer().getPipeline().getFirst().invoke(  
 request, response);

StandardEngineValve：

host.getPipeline().getFirst().invoke(request, response);

ErrorReportValve：

getNext().invoke(request, response);

StandardHostValve：

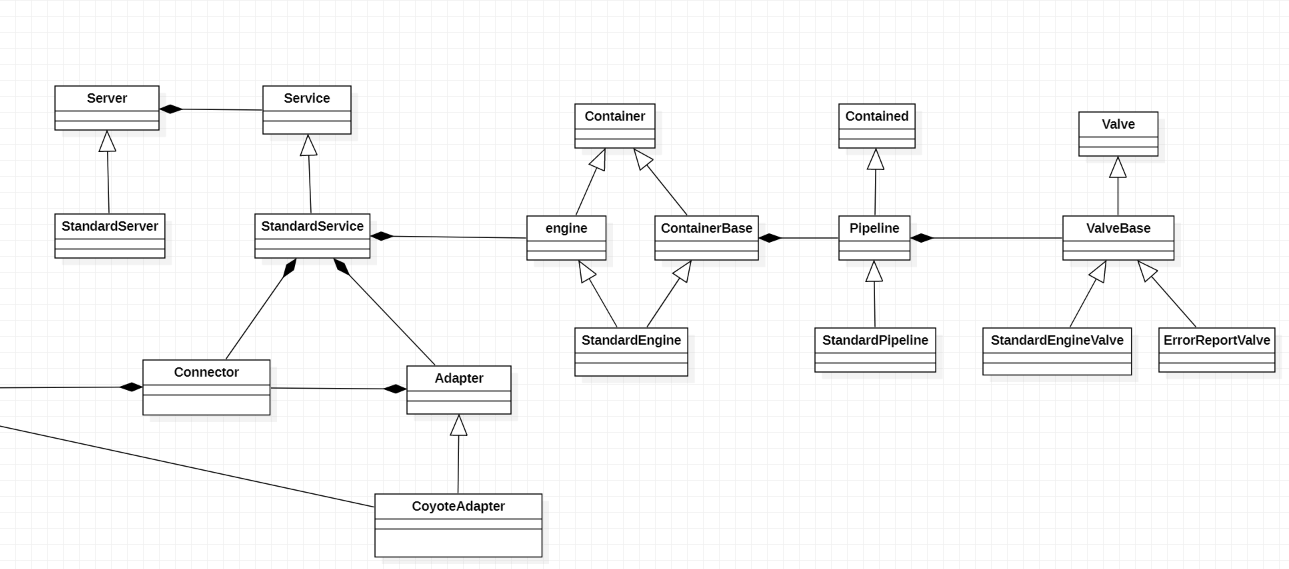
context.getPipeline().getFirst().invoke(request, response);

AuthenticatorBase：

getNext().invoke(request, response);

StandardContextValve：

wrapper.getPipeline().getFirst().invoke(request, response);



通过pipline和valve 进行调用，最终调用到StandardWrapperValve的invoke函数。

StandardWrapperValve:

@Override  
**public final void** invoke(Request request, Response response)  
 **throws** IOException, ServletException {

*// Create the filter chain for this request*ApplicationFilterChain filterChain =  
 ApplicationFilterFactory.*createFilterChain*(request, wrapper, servlet);

**if** (request.isAsyncDispatching()) {  
 request.getAsyncContextInternal().doInternalDispatch();  
 } **else** {  
 filterChain.doFilter  
 (request.getRequest(), response.getResponse());  
 }

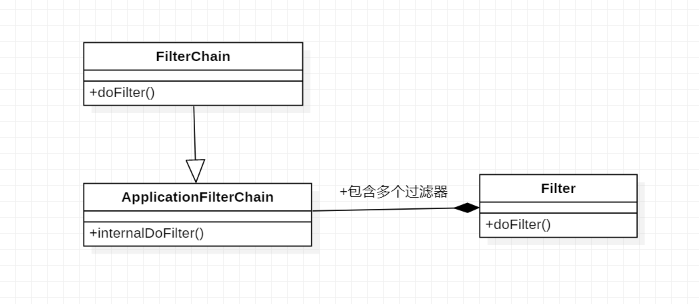
}

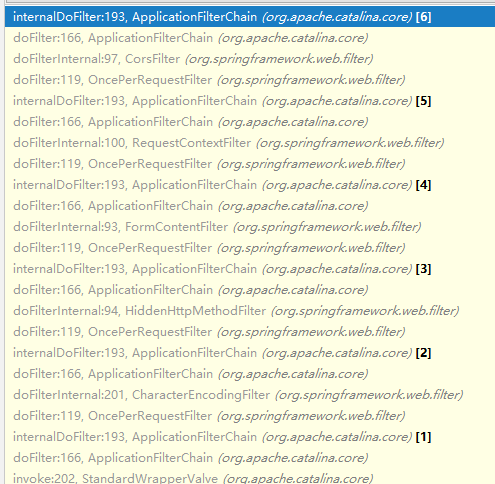
**doFilter请求过滤：**

# ApplicationFilterChain（责任链模式）

按照顺序调用过滤器的doFilter函数，在过滤器中默认有6个系统过滤器。

可以对请求和相应进行修改。





使用责任链模式形成了一个递归调用的效果完成对请求和响应参数的过滤。

ApplicationFilterChain:

**private void** internalDoFilter(ServletRequest request,  
 ServletResponse response)  
 **throws** IOException, ServletException {

**if** (**pos** < **n**) {

ApplicationFilterConfig filterConfig = **filters**[**pos**++];  
**try** {  
 Filter filter = filterConfig.getFilter();

filter.doFilter(request, response, **this**);

}  
 **servlet**.service(request, response);

**DispatcherServlet相应请求：**

**在**internalDoFilter函数递归调用完最后一个filter之后，调用DispatcherServlet的**servlet**.service(request, response)函数进行请求的处理。

处理完成后再最终完成filter责任链的递归调用。

**HandlerAdapter：**

RequestMappingHandlerAdapter

由HandlerMethod获取到对象的HandlerAdapter.

**HandlerMapping：**

RequestMappingHandlerMapping

HandlerExecutionChain

由request从HandlerMapping获取到到对应的HandlerExecutionChain.

**HandlerMethod：**

HandlerExecutionChain对应一个HandlerMethod

**Spring boot框架图:**

