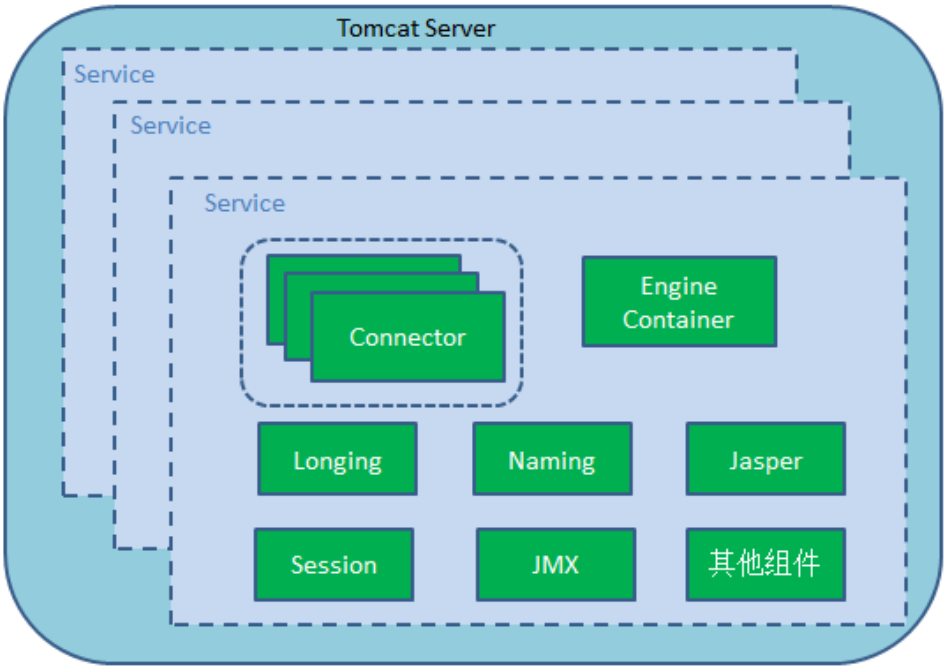
Tomcat 架构和源码简析

作者：淋雨

目录

一、Tomcat 的架构图



二、Tomcat 各个组件说明

1. Server 组件

它代表整个容器,是 Tomcat 实例的顶层元素.由 org.apache.catalina.Server 接口来定义.它用来控制 Tomcat 实例的启动与停止。在 Tomcat 启动阶段，通过 server 来控制的各个 service 的启动。启动完成后它启动独立的端口监听，用来接收 Tomcat 的停止命令。当接收到停止命令后它会触发 service 的停与注销。并提供一个接口让其它程序能够访问到这个 Service 集合、同时要维护它所包含的所有 Service 的生命周期，包括服务初始化、服务终结、Service 定位与查找。

Server 的类结构图如下：

Server 接口的标准实现类是 StandardServer,它实现了 server 接口的所有方法。

StandardServer 最重要的有方法 addService，主要是将各个 service 组件加入到 server 中来，进行统一管理。他的实现如下：

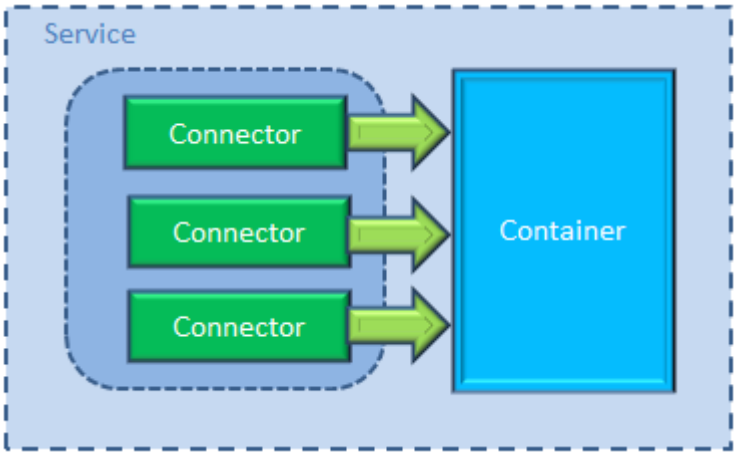
```
public void addService(Service service) {
    service.setServer(this);
    synchronized (services) {
        Service results[] = new Service[services.length + 1];
        System.arraycopy(services, 0, results, 0, services.length);
        results[services.length] = service;
        services = results;
        if (initialized) {
            try {
                service.initialize();
            } catch (LifecycleException e) {
                e.printStackTrace(System.err);
            }
        }
        if (started && (service instanceof Lifecycle)) {
            try {
                ((Lifecycle) service).start();
            } catch (LifecycleException e) {
                ;
            }
        }
        support.firePropertyChange("service", null, service);
    }
}
```

在这个方法中除了将 service 组件注册到 Server 中，同时还完成了 Service 组件的初始化与启动。

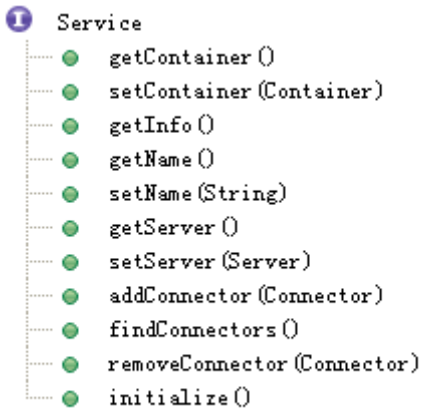
2. Service 组件

Service 是一个抽象的服务封装组件，是 web 容器的一个独立的运行单元。它具备了一个 web 容器所需要的监听连接，servlet 查找与调用，Session 创建与管理，日志服务等常用功能。Service 只是在 Connector 和 Container 外面多包一层，把它们组装在一起，向外面提供服务，一个 Service 可以设置多个 Connector，但是只能有一个 Engine Container 容器，将多个 Connector 收到的连接发送到同一个 Engine Container 进行处理。

Service、Connector 和 Engine Container 关系如下：

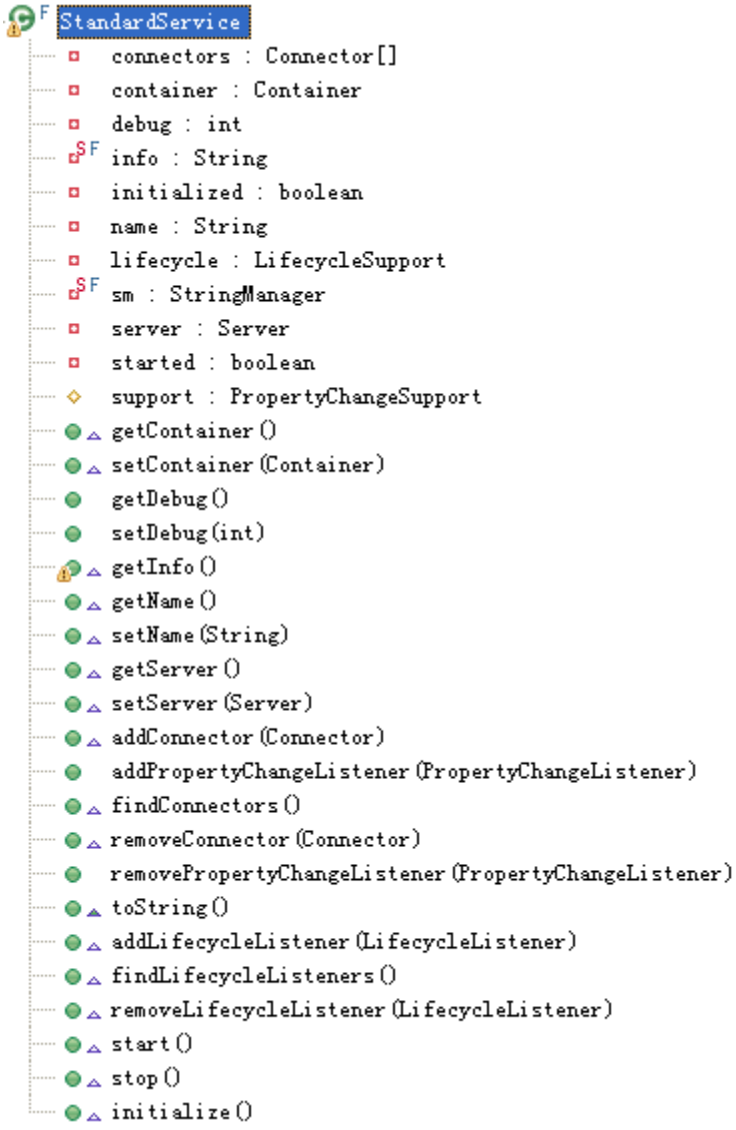


Service 接口如下:



从 Service 接口中定义的方法中可以看出，它主要是为了关联 Connector 和 Container，同时会初始化它下面的其它组件，注意接口中它并没有规定一定要控制它下面的组件的生命周期。所有组件的生命周期在一个 Lifecycle 的接口中控制的。

Tomcat 中 Service 接口的默认实现类是 StandardServic 它不仅实现了 Service 接口同时还实现了 Lifecycle 接口，这样它就可以控制它下面的组件的生命周期了。StandardService 类结构图如下：



下面看一下 StandardService 中主要的几个方法实现的代码，下面是 setContainer 和 addConnector 方法的源码。setContainer 的源码如下：

```
public void setContainer(Container container) {
    Container oldContainer = this.container;
    if ((oldContainer != null) && (oldContainer instanceof Engine)){//清除旧容器的service信息
        ((Engine) oldContainer).setService(null);
    }
    this.container = container;
    if ((this.container != null) && (this.container instanceof Engine)){//将新的容器关联上service信息
        ((Engine) this.container).setService(this);
    }
    if (started && (this.container != null) && (this.container instanceof Lifecycle)) {
        try {
            ((Lifecycle) this.container).start();//启动容器
        } catch (LifecycleException e) {
            ;
        }
    }
    synchronized (connectors) {
        for (int i = 0; i < connectors.length; i++)
            connectors[i].setContainer(this.container);//将每个connector设置新的容器信息
    }
    if (started && (oldContainer != null) &&
        (oldContainer instanceof Lifecycle)) {
        try {
            ((Lifecycle) oldContainer).stop();//对旧容器进行停职
        } catch (LifecycleException e) {
            ;
        }
    }
    support.firePropertyChange("container", oldContainer, this.container);

}
```

它先判断当前的这个 Service 有没有已经关联了 Engine Container，如果已经关联了，那么去掉这个关联关系。如果这个老的 Container 已经被启动了，结束它的生命周期。然后再替换新的关联、再初始化并开始这个新的 Container 的生命周期。最后将所有的 Connector 的容器修改给当前新的容器。注意 Service 和 Container 是双向关联，Connector 和 Container 的关联关系式是单向关联。

addConnector 方法的源码如下：

```
public void addConnector(Connector connector) {
    synchronized (connectors) {
        connector.setContainer(this.container);//设置connector的对应的当前容器
        connector.setService(this);
        Connector results[] = new Connector[connectors.length + 1];//进行数组扩容，并将新的连接器放入到数组的尾部
        System.arraycopy(connectors, 0, results, 0, connectors.length);
        results[connectors.length] = connector;
        connectors = results;
        if (initialized) {
            try {
                connector.initialize();//开始初始化连接器
            } catch (LifecycleException e) {
            }
        }
        if (started && (connector instanceof Lifecycle)) {
            try {
                ((Lifecycle) connector).start();//启动连接器，连接器会根据具体协议实现来启动端口监听和socket处理
            } catch (LifecycleException e) {
            }
        }
        support.firePropertyChange("connector", null, connector);
    }

}
```

上面是 addConnector 方法，这个方法也很简单，首先是设置关联关系，然后是初始化 Connector，然后启动 Connector 进行监听。

3. Connector 组件

Connector 组件是 Tomcat 容器最重要的组件之一，主要负责服务器的端口监听、接收浏览器的发过来的 TCP 连接请求,将收到的 HTTP 请求信息转换成 Request 对象，并将 Request 提交 Engine 组件去执行，将执行完成后所得到的 Response 对象转换 HTTP 的响应信息返回给浏览器。

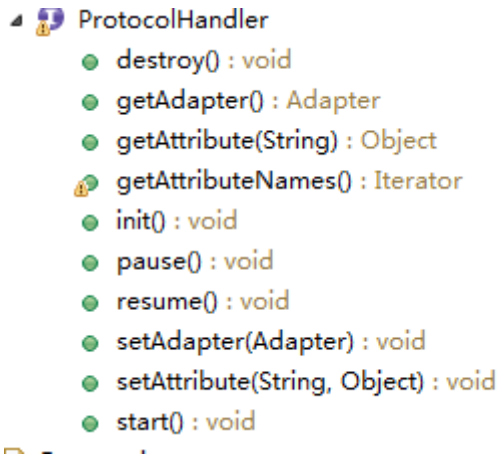
Connector 类主要有 Initialize 和 Start 两个方法来控制整个 connector 的初始化和启动。Initialize 方法源码如下：

```
public void initialize() throws LifecycleException
{
    if (initialized) {//当已经初始化了直接返回
       return;
    }
    this.initialized = true;
    if( oname == null && (container instanceof StandardEngine)) {//设置connector的名字
        try {
            StandardEngine cb=(StandardEngine)container;
            oname = createObjectName(cb.getName(), "Connector");
            Registry.getRegistry(null, null).registerComponent(this, oname, null);
            controller=oname;
        } catch (Exception e) {
            log.error( "Error registering connector ", e);
        }
    }
    adapter = new CoyoteAdapter(this);//初始化适配器
    protocolHandler.setAdapter(adapter);//设置适配器
    if( null == parseBodyMethodsSet ) setParseBodyMethods(getParseBodyMethods());//设置支持HTTP请求的方法 POST,GET等
    IntrospectionUtils.setProperty(protocolHandler, "jkHome", System.getProperty("catalina.base"));
    try {
        protocolHandler.init();//协议处理器初始化 这个非常重要
    } catch (Exception e) {
        throw new LifecycleException
            (sm.getString
             ("coyoteConnector.protocolHandlerInitializationFailed", e));
    }
}
```

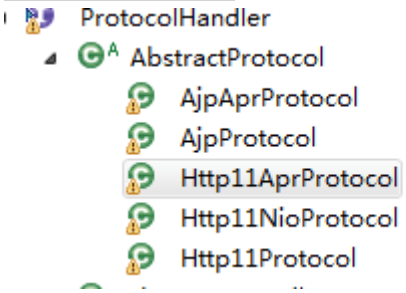
Start 方法源码如下：

```
public void start() throws LifecycleException {
    if( !initialized )
        initialize();
    if (started ) { //当连接器启动过了直接返回
        return;
    }
    lifecycle.fireLifecycleEvent(START_EVENT, null);
    started = true;
    if ( this.oname != null ) {//注册新的协议处理器
        try {
            Registry.getRegistry(null, null).registerComponent
            (protocolHandler, createObjectName(this.domain,"ProtocolHandler")
        } catch (Exception ex) {
        }
    } else {
    }
    try {
        protocolHandler.start();//协议处理器启动 这个很重要
    } catch (Exception e) {
    }
}
```

通过上面的代码我们能明显的观察到 Connector 并没有直接启动 socket 的监听功能，而是委托给 `ProtocolHandler` 协议处理器来做的。下面我们来看下 `ProtocolHandler` 接口的类图：



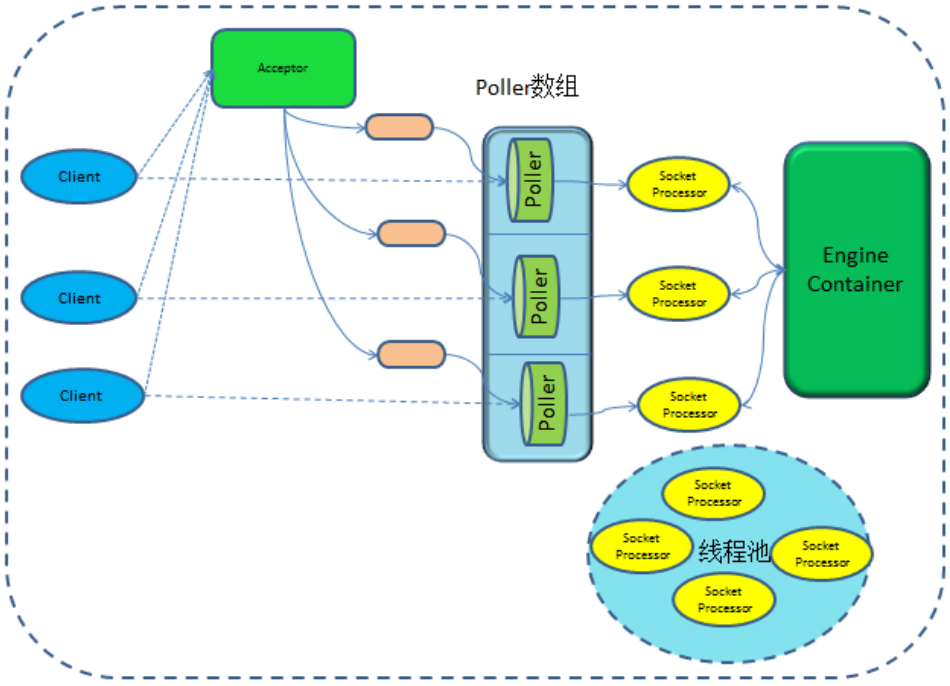
`ProtocolHandler` 本身是个接口，他有四个具体的实现类来实现它的功能：



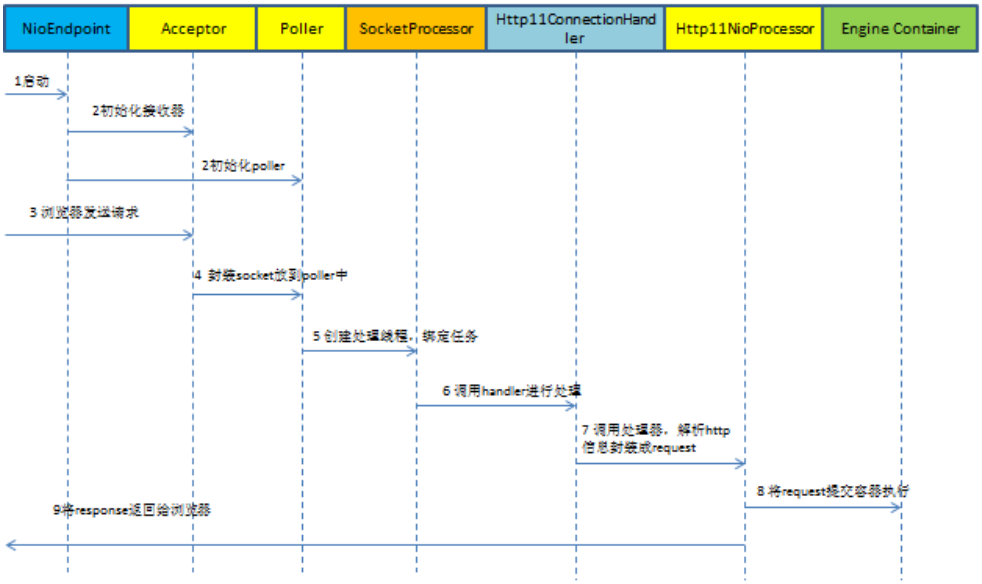
也就是我们常用的四中 Connector。

下面我们就以 `Http11NioProtocol` 为例详细了解下 Connector 的实现细节。

整个流程图如下：



整个时序图如下：



1 Http11NioProtocol 调用 NioEndpoint 的 init 方法初始化服务端 Socket Server 的基础信息

2 Http11NioProtocol 调用 NioEndpoint 的 start 方法启动端口监听，并初始化好 Acceptor，Executer 和 Poller。让这些线程开始工作。至此服务端启动完成。

3 客户端发起 socket 连接，Acceptor 将 socket 连接封装成 NioChannel 对象注册到 Poller 中。

4 Poller 进一步将 NioChannel 封装成 PollerEvent 对象，并放到 event 的事件队列中。

5 Poller 在 run 方法中将 event 队列中的 PollerEvent，注册到当前的 selector 中，注册 SelectionKey.OP_READ 事件。

6 当 selector 中获取到读的事件后，获取到 SocketProcessor 处理线程，将 NioChannel 绑定到 SocketProcessor，交由 SocketProcessor 处理。

7 当线程 SocketProcessor 开始工作后，先检查 NioChannel 的状态，然后调用 Http11ConnectionHandler 进行解析 socket 数据信息。

8 Http11ConnectionHandler 主要用就是掉 Http11NioProcessor 用来对 socket 进行处理，当处理完成了之后将结果反馈给客户端，并对资源进行回收。

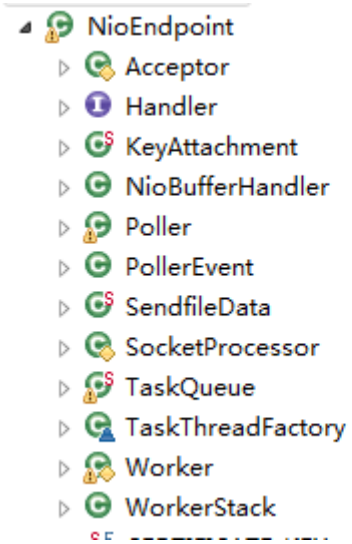
Init 方法源码如下：

```
/** Start the protocol
 */
public void init() throws Exception {
    //设置服务端点的名称
    ep.setName(getName());
    //设置协议处理器
    ep.setHandler(cHandler);

    //设置接收和发送缓冲区的大小
    ep.getSocketProperties().setRxBufSize(Math.max(ep.getSocketProperties().getRxBufSize(),getMaxHttpHeaderSize()));
    ep.getSocketProperties().setTxBufSize(Math.max(ep.getSocketProperties().getTxBufSize(),getMaxHttpHeaderSize()));

    try {
        ep.init();//初始化端点
        sslImplementation = new JSSEImplementation();
    } catch (Exception ex) {
        log.error(sm.getString("http11protocol.endpoint.initerror"), ex);
        throw ex;
    }
    if(log.isInfoEnabled())
        log.info(sm.getString("http11protocol.init", getName()));

}
```

初始化了服务端点的信息，并为端点设置了协议处理器，和数据缓冲区大小
Start 方法源码如下：

```
public void start() throws Exception {
    try {
        ep.start();//启动端点
    } catch (Exception ex) {
        log.error(sm.getString("http11protocol.endpoint.starterror"), ex);
        throw ex;
    }
    if(log.isInfoEnabled())
        log.info(sm.getString("http11protocol.start", getName()));
}
```

上面代码说明 Http11NioProtocol 是将服务端点 NioEndpoint 和协议处理器 Http11ConnectionHandler 来进行绑定的。NioEndpoint 负责收到 socket 请求，而协议处理器 Http11ConnectionHandler 负责处理请求。
NioEndpoint 的类图如下：



通过上面的类图我们发现 NioEndpoint 有很多的内部类，下面我们就看下里面重要的内部类，分别都是用来干什么的。看内部类之前我们还是看下 NioEndpoint 的 init 和 start 方法分别做了哪些工作。
Init 方法源码如下：

```
public void init()
    throws Exception {
    if (initialized)
        return;
    serverSock = ServerSocketChannel.open();//打开socket通道
    serverSock.socket().setPerformancePreferences(socketProperties.getPerformanceConnectionTime(),
                                                  socketProperties.getPerformanceLatency(),
                                                  socketProperties.getPerformanceBandwidth());//设置socket性能参数
    InetSocketAddress addr = (address!=null?new InetSocketAddress(address,port):new InetSocketAddress(port));
    serverSock.socket().bind(addr,backlog); //绑定端口地址,设置为NIO模型
    serverSock.configureBlocking(true); //mimic APR behavior
    serverSock.socket().setSoTimeout(getSocketProperties().getSoTimeout());//设置超时
     //设置连接器的线程数，默认是1
    if (acceptorThreadCount == 0) {
        // FIXME: Doesn't seem to work that well with multiple accept threads
        acceptorThreadCount = 1;
    }
    if (pollerThreadCount <= 0) {
        //minimum one poller thread
        pollerThreadCount = 1;
    }
    stopLatch = new CountDownLatch(pollerThreadCount);//设置线程锁
    if (oomParachute>0) reclaimParachute(true);
    selectorPool.open();
    initialized = true;
}
```

Start 方法源码如下：

```
public void start()  throws Exception {
    if (!initialized) {//判断是否初始化过，没有初始化
        init();
    }
    if (!running) {
        running = true;
        paused = false;
        //设置任务队列以及线程池
        if (getUseExecutor()) {
            if ( executor == null ) {
                TaskQueue taskqueue = new TaskQueue();
                TaskThreadFactory tf = new TaskThreadFactory(getName() + "-exec-");
                executor = new ThreadPoolExecutor(getMinSpareThreads(), getMaxThreads(),
                        60, TimeUnit.SECONDS,taskqueue, tf);
                taskqueue.setParent( (ThreadPoolExecutor) executor, this);
            }
        } else if ( executor == null ) {//avoid two thread pools being created
            workers = new WorkerStack(maxThreads);
        }
        //设置socket事件监听池，长度默认为处理器的个数
        pollers = new Poller[getPollerThreadCount()];
        for (int i=0; i<pollers.length; i++) {//初始化化Poller
            pollers[i] = new Poller();
            Thread pollerThread = new Thread(pollers[i], getName() + "-ClientPoller-"+i);
            pollerThread.setPriority(threadPriority);
            pollerThread.setDaemon(true);
            pollerThread.start();
        }

          // 设置连接器
          for (int i = 0; i < acceptorThreadCount; i++) {
              Thread acceptorThread = new Thread(new Acceptor(), getName() + "-Acceptor-" + i);
              acceptorThread.setPriority(threadPriority);
              acceptorThread.setDaemon(daemon);
              acceptorThread.start();//启动连接器
          }
      }
  }
```

在 start 的方法中 初始化了任务队列和线程处理器，同时初始化了 socket nio 事件的监听数组，听初始化 socket 连接的接收器。

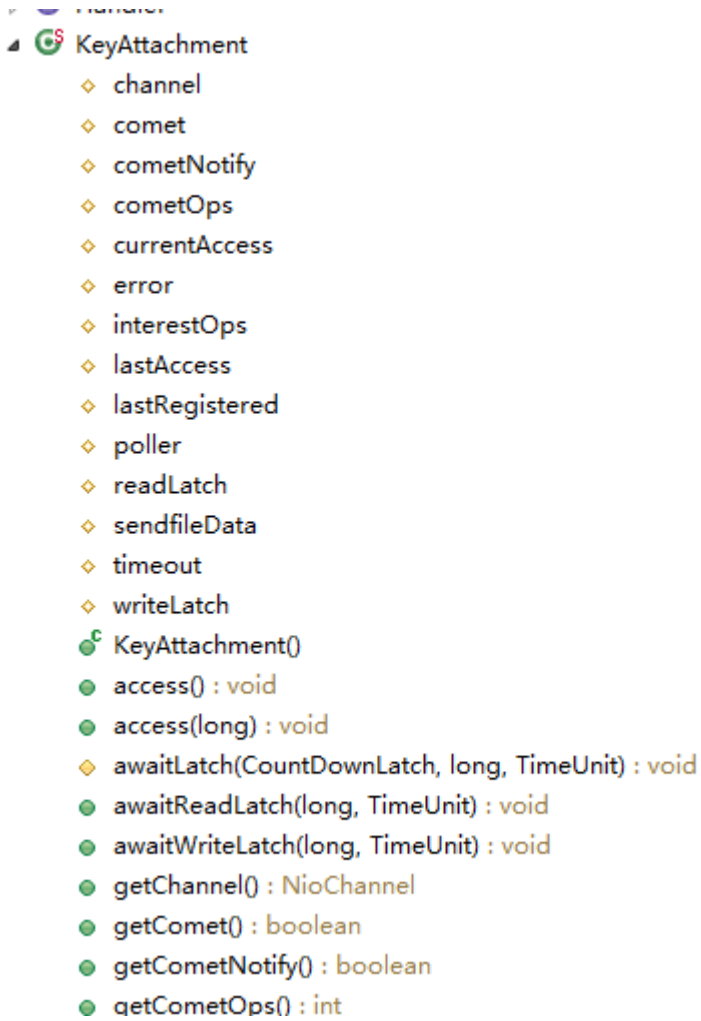
Acceptor 是一个用来接收 socket 的连接请求，对他进行封装，将 socket 封装成 NioChannel 对象，然后将 NioChannel 对象放入到 pollers 数组中。源码如下：

```
    protected class Acceptor implements Runnable {
        public void run() {
            while (running) {
                try {
                    SocketChannel socket = serverSock.accept();
                    if ( running && (!paused) && socket != null ) {
                        if (!setSocketOptions(socket)) {
                            try {
                                socket.socket().close();
                                socket.close();
                            } catch (IOException ix) {
                                    log.debug("", ix);
                            }
                        }
                    }
                }catch (SocketTimeoutException sx) {
                }catch ( IOException x ) {
                    if ( running ) log.error(sm.getString("endpoint.accept.fail"), x);
                } catch (OutOfMemoryError oom) {
                } catch (Throwable t) {
                    log.error(sm.getString("endpoint.accept.fail"), t);
                }
            }//while
        }//run
    }
```

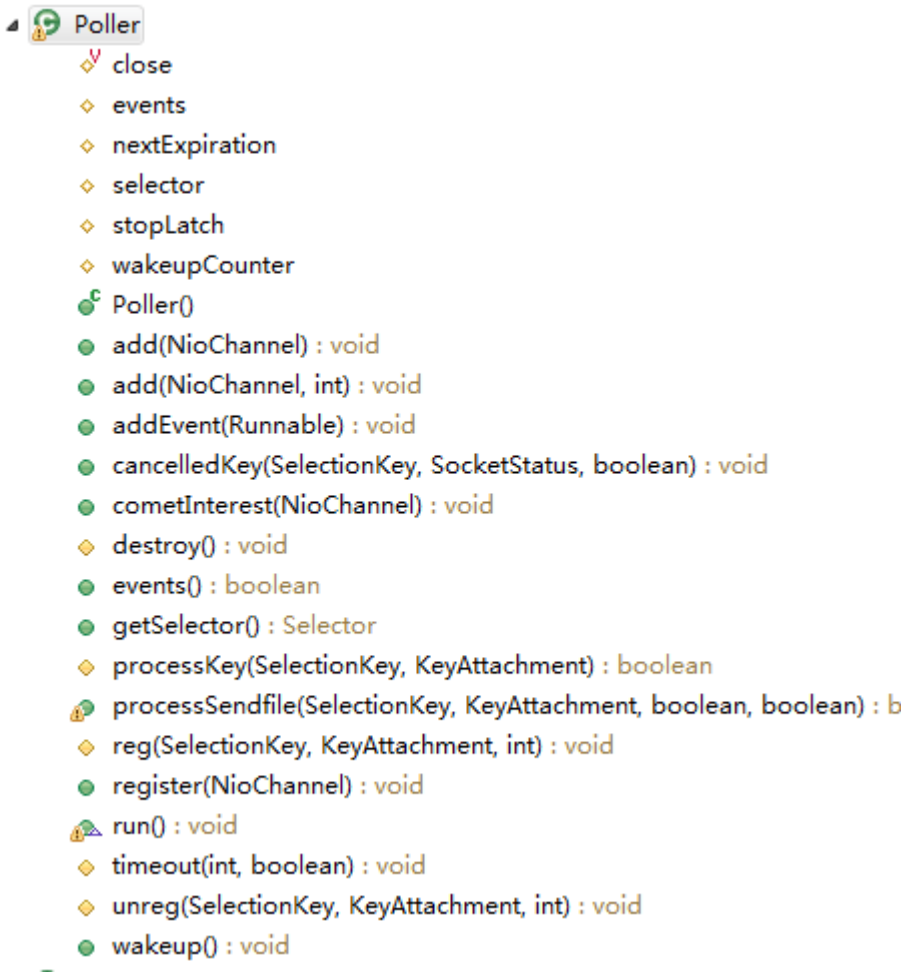
`setSocketOptions` 方法的源码如下：

```
protected boolean setSocketOptions(SocketChannel socket) {
    // Process the connection
    try {
        socket.configureBlocking(false);//设置socket为非阻塞的socket
        Socket sock = socket.socket();
        socketProperties.setProperties(sock);//设置socket属性
        NioChannel channel = nioChannels.poll(); //从通道对象池中获取一个通道，当通道为空的时候创建一个新的
        if ( channel == null ) {
                // normal tcp setup
                NioBufferHandler bufhandler = new NioBufferHandler(socketProperties.getAppReadBufSize(),
                                                                   socketProperties.getAppWriteBufSize(),
                                                                   socketProperties.getDirectBuffer());

                channel = new NioChannel(socket, bufhandler);
        } else {
            channel.setIOChannel(socket);//设置socket
            channel.reset();//重置对象信息
        }
        getPoller0().register(channel);//将NioChannel对象放置到Poller中
    } catch (Throwable t) {
        try {
            log.error("",t);
        }catch ( Throwable tt){}
        // Tell to close the socket
        return false;
    }
    return true;
}
```

`KeyAttachment` 将 `NioChannel` 和 `Poller` 组合成的一个钩子，用来在 `socket NIO` 的 `attache` 方法中使用



Poller 是用来做 socket nio 读写事件的 selector 持有容器。在前面讲到 Acceptor 将 socket 对象封装成一个 `NioChannel` 对象，然后注入到 `Poller` 中来。下面是 `Poller` 的类图：



他包含一个 `Selector` 是用来对具体每一个 `socket` 的读写事件进行注册的。
`Register` 的方法源代码如下：

```
public void register(final NioChannel socket)
{
    socket.setPoller(this);//将NioChannel 绑定一个poller
    //从连接池中获取到一个KeyAttachment，当不能获取到的时候就直接创建
    KeyAttachment key = keyCache.poll();
    final KeyAttachment ka = key!=null?key:new KeyAttachment();
    ka.reset(this,socket,getSocketProperties().getSoTimeout());
    ka.interestOps(SelectionKey.OP_READ);//注册一个可读的事件监听

    //创建一个PollerEvent,当没有的时候直接创建
    PollerEvent r = eventCache.poll();
    if (r==null) {
        r = new PollerEvent(socket,ka,OP_REGISTER);
    }
    else {
         r.reset(socket,ka,OP_REGISTER);
    }
    addEvent(r);
}
```

```
public void addEvent(Runnable event) {
    //内部定义了事件的队列
    events.offer(event);
    if ( wakeupCounter.incrementAndGet() == 0 ) {//当前需
        selector.wakeup();
    }
}
```

Poller 同时实现了 `Runnable` 接口，他又是一个线程类，当前面初始完成后它就变成一个处于运行状态的线程。

```
public void run() {
    while (running) {
        try {
            boolean hasEvents = false;
            hasEvents = (hasEvents | events());//PollerEvent内部的socket向当前的Poller的selector中注册监听读的事件
            int keyCount = 0;
            try {
                if ( !close ) {//获取到 select中读事件的个数
                    if (wakeupCounter.getAndSet(-1) > 0) {
                        keyCount = selector.selectNow();
                    } else {
                        keyCount = selector.select(selectorTimeout);
                    }
                    wakeupCounter.set(0);
                }
            } catch ( Exception x ) {
                continue;
            }
            //当当前没有注册事件的事件发生的时候，再触发一次事件注册
            if ( keyCount == 0 ) {
                hasEvents = (hasEvents | events());
            }

              //获取到注册的事件信息
              Iterator iterator = keyCount > 0 ? selector.selectedKeys().iterator() : null;
              while (iterator != null && iterator.hasNext()) {
                  SelectionKey sk = (SelectionKey) iterator.next();
                  KeyAttachment attachment = (KeyAttachment)sk.attachment();
                  if (attachment == null) {
                      iterator.remove();
                  } else {
                      attachment.access();//更新访问的时间
                      iterator.remove();
                      processKey(sk, attachment);//处理HTTP请求，将其转换成Request对象
                  }
              }
              timeout(keyCount,hasEvents);
              if ( oomParachute > 0 && oomParachuteData == null ) checkParachute();
          } catch (OutOfMemoryError oom) {
          }
      }//while
      synchronized (this) {
          this.notifyAll();
      }
      stopLatch.countDown();
  }
```

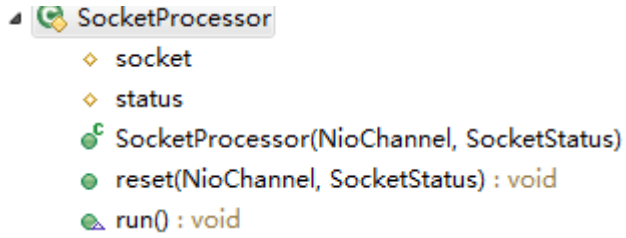
Poller 线程通过循环不停的来获取到当前 selector 中所有发生的读写事件，然后进行处理。通过上面的代码可以看出具体进行处理是在 `processKey` 方法里面进行的。下面我们来看下 `processKey` 的具体源码：

```
protected boolean processKey(SelectionKey sk, KeyAttachment attachment) {
    boolean result = true;
    try {
        if ( close ) {//当系统已经关闭的时候，停止注册事件
            cancelledKey(sk, SocketStatus.STOP, false);
        } else if ( sk.isValid() && attachment != null ) {
            attachment.access();//更新最后访问时间
            sk.attach(attachment);//将attachment和socket绑定
            NioChannel channel = attachment.getChannel();
            if (sk.isReadable() || sk.isWritable() ) {//判断是读写事件
                //当需要发送文件的时候发送文件
                if ( attachment.getSendfileData() != null ) {
                    processSendfile(sk,attachment,true, false);
                } else if ( attachment.getComet() ) {
                    if ( isWorkerAvailable() ) { //判断当前是否有可用的线程，对容器进行过载保护
                        reg(sk, attachment, 0);//注册读写事件
                        if (sk.isReadable()) {//当通道时可读的时候
                            if (!processSocket(channel, SocketStatus.OPEN)) {//处理niochannel
                                processSocket(channel, SocketStatus.DISCONNECT);
                            }
                        } else {
                            if (!processSocket(channel, SocketStatus.OPEN)){
                                processSocket(channel, SocketStatus.DISCONNECT);
                            }
                        }

                        } else {
                            result = false;
                        }
                    } else {
                        if ( isWorkerAvailable() ) {
                            unreg(sk, attachment,sk.readyOps());
                            boolean close = (!processSocket(channel));
                            if (close) {
                                cancelledKey(sk,SocketStatus.DISCONNECT,false);
                            }
                        } else {
                            result = false;
                        }
                    }
                }
            } else {
                cancelledKey(sk, SocketStatus.ERROR,false);
            }
        } catch ( CancelledKeyException ckx ) {
            cancelledKey(sk, SocketStatus.ERROR,false);
        } catch (Throwable t) {
            log.error("",t);
        }
        return result;
    }
```

通过上面我们看到对 `SelectionKey` 进行了处理，并将 `attachment` 对象绑定到 `SelectionKey` 上面。当判断 `SelectionKey` 是读写事件的时候，通过 `processSocket` 方法来读取 HTTP 请求信息，或者发送 HTTP 响应信息。
下面是 `processSocket` 的源码：

```
protected boolean processSocket(NioChannel socket, SocketStatus status, boolean dispatch) {
    try {
        KeyAttachment attachment = (KeyAttachment)socket.getAttachment(false);
        attachment.setCometNotify(false);
        //当没有线程池的时候，直接获取线程来处理
        if (executor == null) {
            getWorkerThread().assign(socket, status);
        } else {
            //获取处理器，没有创建一个
            SocketProcessor sc = processorCache.poll();
            if ( sc == null ) {
                sc = new SocketProcessor(socket,status);
            }
            else {
                sc.reset(socket,status);
            }
            if ( dispatch ){//提交到线程池中直接执行
                executor.execute(sc);
            }
            else {
                 sc.run();
            }
        }
    } catch (Throwable t) {
        return false;
    }
    return true;
}
```

它主要的功能是获取到用来处理 HTTP 请求信息的线程 `SocketProcessor`，然后启动线程进行执行。`SocketProcessor` 的类图结果如下：



`SocketProcessor` 实现了 `Runnable` 接口是个线程类，每个处理器就会启动一个线程。

```
public void run() {
    synchronized (socket) {
        NioEndpoint.this.activeSocketProcessors.addAndGet(1);//当前处理的线程数自增
        SelectionKey key = null;
        try {
            key = socket.getIOChannel().keyFor(socket.getPoller().getSelector());//获取到当前要处理socket的SelectionKey
            int handshake = -1;
            try {
                if (key!=null){//没有任何意义的代码
                    handshake = socket.handshake(key.isReadable(), key.isWritable());
                }
            }catch (Exception x ) {
                handshake = -1;
            }
            if ( handshake == 0 ) {
                //调用Http11ConnectionHandler对 socket进行真正的处理
                boolean closed = (status==null)?(handler.process(socket)==Handler.SocketState.CLOSED) :
                    (handler.event(socket,status)==Handler.SocketState.CLOSED);
```

```
                if (closed) {//当请求已经处理结束 有三种情况会是请求结束 1请求处理过程中出错 2HTTP1.0请求 一个socket只发送一个HTTP请求
                                //3 HTTP1.1中keeplive模式，最后一个请求处理完成了
                    try {
                        KeyAttachment ka = null;
                        if (key!=null) {//取消selectKey的任何注册事件
                            ka = (KeyAttachment) key.attachment();
                            if (ka!=null) {
                                ka.setComet(false);
                            }
                            socket.getPoller().cancelledKey(key, SocketStatus.ERROR, false);
                        }
                        if (socket!=null) {//将socket放到对象池中，供后面的请求调用
                            nioChannels.offer(socket);
                            socket = null;
                        }
                        if ( ka!=null ) {
                            keyCache.offer(ka);//将KeyAttachment放到对象池中，供后面的请求调用
                            ka = null;
                        }
                    }catch ( Exception x ) {
                        log.error("",x);
                    }
                }
```

`Http11ConnectionHandler` 主要用就是掉 `Http11NioProcessor` 用来对 `socket` 进行处理，当处理完成了之后对资源进行回收。

```
        public SocketState process(NioChannel socket) {
            Http11NioProcessor processor = null;
            try {
                processor = connections.remove(socket);//从keeplive 的处理器池connections中获取当前NIO的处理器
                if (processor == null) {
                    processor = recycledProcessors.poll();
                }
                if (processor == null) {//的那个没有取到就创建一个
                    processor = createProcessor();
                }
                if (processor instanceof ActionHook) {
                    ((ActionHook) processor).action(ActionCode.ACTION_START, null);
                }
                SocketState state = processor.process(socket);//处理socket连接
                if (state == SocketState.LONG) {//当是keeplive模式的时候
                    connections.put(socket, processor);//将当前的处理器放入到处理器池中
                    socket.getPoller().add(socket);
                } else if (state == SocketState.OPEN) {
                    release(socket, processor);//释放处理器
                    socket.getPoller().add(socket);
                } else {
                    release(socket, processor);//释放处理器
                }
                return state;

            } catch (Exception e) {
                }
                release(socket, processor);
                return SocketState.CLOSED;
            }
```

`Http11NioProcessor` 主要的处理方法如下：

```
    public SocketState process(NioChannel socket) throws IOException {
        RequestInfo rp = request.getRequestProcessor();
        rp.setStage(org.apache.coyote.Constants.STAGE_PARSE);
        this.socket = socket;
        inputBuffer.setSocket(socket);//定义输入缓冲区
        outputBuffer.setSocket(socket);//定义输出缓冲区
        inputBuffer.setSelectorPool(endpoint.getSelectorPool());//设置Poller
        outputBuffer.setSelectorPool(endpoint.getSelectorPool());
        error = false;
        keepAlive = true;
        comet = false;
        int keepAliveLeft = maxKeepAliveRequests;
        long soTimeout = endpoint.getSoTimeout();
        boolean keptAlive = false;
        boolean openSocket = false;
        boolean recycle = true;
```

```
while (!error && keepAlive && !comet) {
    try {
        if( !disableUploadTimeout && keptAlive && soTimeout > 0 ) {//设置超时时间
            socket.getIOChannel().socket().setSoTimeout((int)soTimeout);
        }
        if (!inputBuffer.parseRequestLine(keptAlive)) {//读取一行信息
            openSocket = true;
            recycle = false;
            break;
        }
        keptAlive = true;
        request.getMimeHeaders().setLimit(endpoint.getMaxHeaderCount());//设置请求信息中头部参数的最大个数
        if ( !inputBuffer.parseHeaders() ) {//解析HTTP的头信息
            openSocket = true;
            recycle = false;
            break;
        }
        request.setStartTime(System.currentTimeMillis());//更新请求开始时间
        if (!disableUploadTimeout) { //only for body, not for request headers
            socket.getIOChannel().socket().setSoTimeout((int)timeout);
        }
    }
```

```
    catch (Throwable t) {
        response.setStatus(400);
        adapter.log(request, response, 0);
        error = true;
    }

    if (!error) {
        rp.setStage(org.apache.coyote.Constants.STAGE_PREPARE);//准备开始解析http request
        try {
            prepareRequest();//解析HTTP Request
        } catch (Throwable t) {
            response.setStatus(400);
            adapter.log(request, response, 0);
            error = true;
        }
    }

    if (maxKeepAliveRequests > 0 && --keepAliveLeft == 0){//判断是否keeplive模式下最后一个请求
        keepAlive = false;
    }
```

```
    catch (Throwable t) {
        response.setStatus(400);
        adapter.log(request, response, 0);
        error = true;
    }
    if (!error) {
        rp.setStage(org.apache.coyote.Constants.STAGE_PREPARE);//准备开始解析http request
        try {
            prepareRequest();//解析HTTP Request
        } catch (Throwable t) {
            response.setStatus(400);
            adapter.log(request, response, 0);
            error = true;
        }
    }
    if (maxKeepAliveRequests > 0 && --keepAliveLeft == 0){//判断是否keeplive模式下最后一个请求
        keepAlive = false;
    }
```

```
    if (!error) {
        try {
            rp.setStage(org.apache.coyote.Constants.STAGE_SERVICE);
            adapter.service(request, response);//将request和response中到service中进行处理
            if(keepAlive && !error) { // Avoid checking twice.
                error = response.getErrorException() != null ||statusDropsConnection(response.getStatus());
            }
            SelectionKey key = socket.getIOChannel().keyFor(socket.getPoller().getSelector());
            if (key != null) {
                NioEndpoint.KeyAttachment attach = (NioEndpoint.KeyAttachment) key.attachment();
                if (attach != null)  {
                    attach.setComet(comet);
                    if (comet) {
                        Integer comettimeout = (Integer) request.getAttribute("org.apache.tomcat.comet.timeout");
                        if (comettimeout != null) attach.setTimeout(comettimeout.longValue());
                    }
                }
            }
        } catch (Throwable t) {
            response.setStatus(500);
            adapter.log(request, response, 0);
            error = true;
        }
    }
```

```
            if (!comet) {
                if(error)
                    inputBuffer.setSwallowInput(false);
                endRequest();
            }
            if (error) {
                response.setStatus(500);
            }
            request.updateCounters();
            if (!comet) {
                inputBuffer.nextRequest();
                outputBuffer.nextRequest();
            }

            if (sendfileData != null && !error) {
                KeyAttachment ka = (KeyAttachment)socket.getAttachment(false);
                ka.setSendfileData(sendfileData);
                sendfileData.keepAlive = keepAlive;
                SelectionKey key = socket.getIOChannel().keyFor(socket.getPoller().getSelector());
                openSocket = socket.getPoller().processSendfile(key,ka,true,true);
                break;
            }
            rp.setStage(org.apache.coyote.Constants.STAGE_KEEPALIVE);
        }

        rp.setStage(org.apache.coyote.Constants.STAGE_ENDED);
        if (comet) {
            if (error) {
                recycle();
                return SocketState.CLOSED;
            } else {
                return SocketState.LONG;
            }
        } else {
            if ( recycle ) recycle();
            //return (openSocket) ? (SocketState.OPEN) : SocketState.CLOSED;
            return (openSocket) ? (recycle?SocketState.OPEN:SocketState.LONG) : SocketState.CLOSED;
        }

    }
```