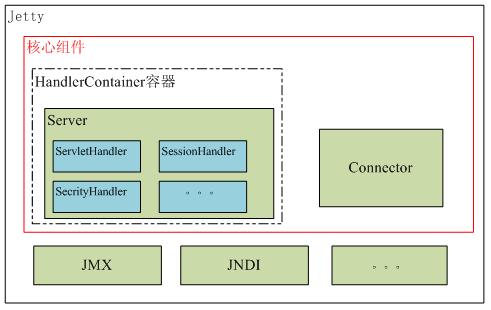
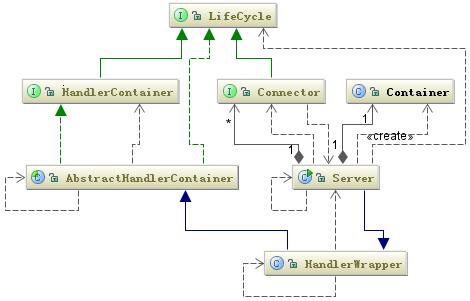
1. **Jetty基本架构图**



整个 Jetty 的核心组件由 Server 和 Connector 两个组件构成，整个 Server 组件是基于 Handler 容器工作的，它类似与 Tomcat 的 Container 容器。Jetty 中另外一个比不可少的组件是 Connector，它负责接受客户端的连接请求，并将请求分配给一个处理队列去执行。

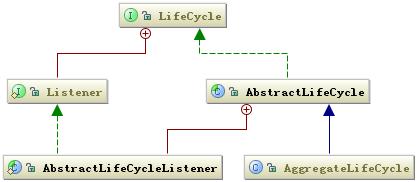
还有一些可有可无的组件，我们可以在它上做扩展。如 JMX，我们可以定义一些 Mbean 把它加到 Server 中，当 Server 启动的时候，这些 Bean 就会一起工作。

1. **Jetty 的主要组件的类图**  
   

整个 Jetty 的核心是围绕着 Server 类来构建，Server 类继承了 Handler，关联了 Connector 和 Container。Container 是管理 Mbean 的容器。Jetty 的 Server 的扩展主要是实现一个个 Handler 并将 Handler 加到 Server 中，Server 中提供了调用这些 Handler 的访问规则。

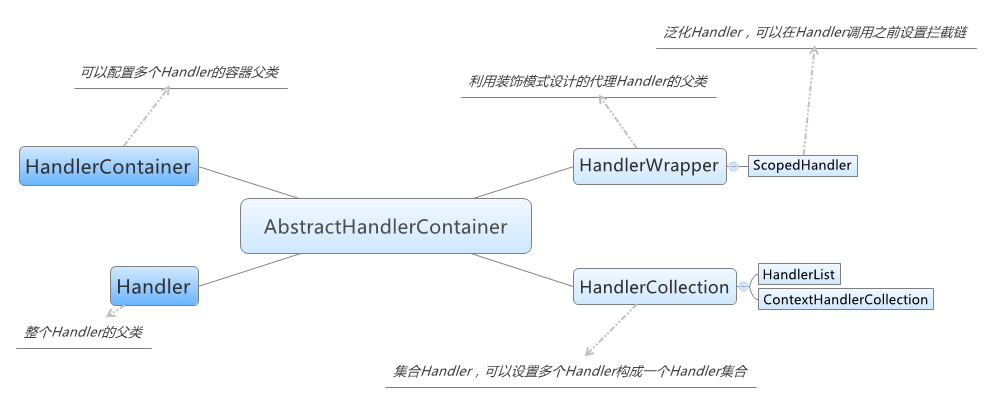
整个 Jetty 的所有组件的生命周期管理是基于观察者模板设计，它和 Tomcat 的管理是类似的。

1. **LifeCycle 的类关系图**



每个组件都会持有一个观察者（在这里是 Listener 类，这个类通常对应到观察者模式中常用的 Observer 角色）集合，当 start、fail 或 stop 等事件触发时，这些 Listener 将会被调用，这是最简单的一种设计方式，相比 Tomcat 的 LifeCycle 要简单的多。

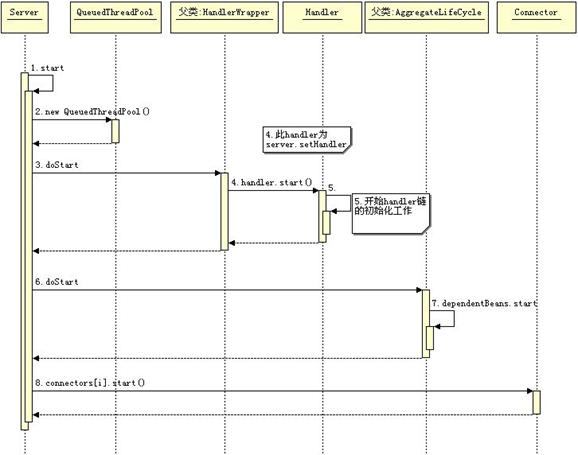
1. **Handler 的体系结构**



Jetty 主要是基于 Handler 来设计的，Handler 的体系结构影响着整个 Jetty 的方方面面。

Jetty 主要提供了两种 Handler 类型，一种是 HandlerWrapper，它可以将一个 Handler 委托给另外一个类去执行，如我们要将一个 Handler 加到 Jetty 中，那么就必须将这个 Handler 委托给 Server 去调用。配合 ScopeHandler 类我们可以拦截 Handler 的执行，在调用 Handler 之前或之后，可以做一些另外的事情，类似于 Tomcat 中的 Valve；另外一个 Handler 类型是 HandlerCollection，这个 Handler 类可以将多个 Handler 组装在一起，构成一个 Handler 链，方便我们做扩展。

1. **Jetty 的启动流程**



Jetty 的入口是 Server 类，Server 类启动完成了，就代表 Jetty 能为你提供服务了。它到底能提供哪些服务，就要看 Server 类启动时都调用了其它组件的 start 方法。从 Jetty 的配置文件我们可以发现，配置 Jetty 的过程就是将那些类配置到 Server 的过程。

Jetty 中所有的组件都会继承 LifeCycle，所以 Server 的 start 方法调用就会调用所有已经注册到 Server 的组件，Server 启动其它组件的顺序是：首先启动设置到 Server 的 Handler，通常这个 Handler 会有很多子 Handler，这些 Handler 将组成一个 Handler 链。Server 会依次启动这个链上的所有 Handler。接着会启动注册在 Server 上 JMX 的 Mbean，让 Mbean 也一起工作起来，最后会启动 Connector，打开端口，接受客户端请求，启动逻辑非常简单。

1. **接受请求**

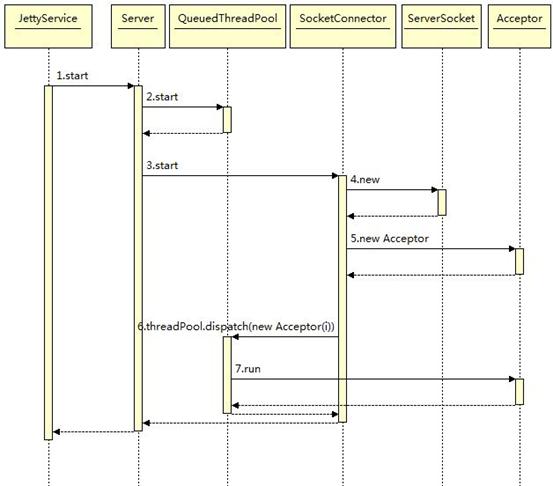
Jetty 作为一个独立的 Servlet 引擎可以独立提供 Web 服务，但是它也可以与其他 Web 应用服务器集成，所以它可以提供基于两种协议工作，一个是 HTTP，一个是 AJP 协议。如果将 Jetty 集成到 Jboss 或者 Apache，那么就可以让 Jetty 基于 AJP 模式工作。

* 1. **基于 HTTP 协议工作**

如果前端没有其它 web 服务器，那么 Jetty 应该是基于 HTTP 协议工作。也就是当 Jetty 接收到一个请求时，必须要按照 HTTP 协议解析请求和封装返回的数据。那么 Jetty 是如何接受一个连接又如何处理这个连接呢？

我们设置 Jetty 的 Connector 实现类为 org.eclipse.jetty.server.bi.SocketConnector 让 Jetty 以 BIO 的方式工作，Jetty 在启动时将会创建 BIO 的工作环境，它会创建 HttpConnection 类用来解析和封装 HTTP1.1 的协议，ConnectorEndPoint 类是以 BIO 的处理方式处理连接请求，ServerSocket 是建立 socket 连接接受和传送数据，Executor 是处理连接的线程池，它负责处理每一个请求队列中任务。acceptorThread 是监听连接请求，一有 socket 连接，它将进入下面的处理流程。

当 socket 被真正执行时，HttpConnection 将被调用，这里定义了如何将请求传递到 servlet 容器里，又如何将请求最终路由到目的 servlet，关于这个细节可以参考《 servlet 工作原理解析》一文。

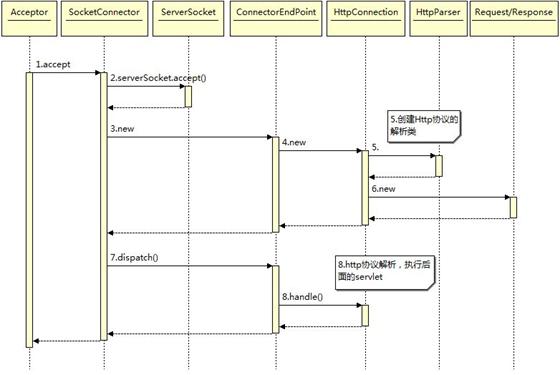


Jetty 创建接受连接环境需要三个步骤：

1. 创建一个队列线程池，用于处理每个建立连接产生的任务，这个线程池可以由用户来指定，这个和 Tomcat 是类似的。
2. 创建 ServerSocket，用于准备接受客户端的 socket 请求，以及客户端用来包装这个 socket 的一些辅助类。
3. 创建一个或多个监听线程，用来监听访问端口是否有连接进来。

相比 Tomcat 创建建立连接的环境，Jetty 的逻辑更加简单，牵涉到的类更少，执行的代码量也更少了。

当建立连接的环境已经准备好了，就可以接受 HTTP 请求了，当 Acceptor 接受到 socket 连接后将转入下图所示流程执行：

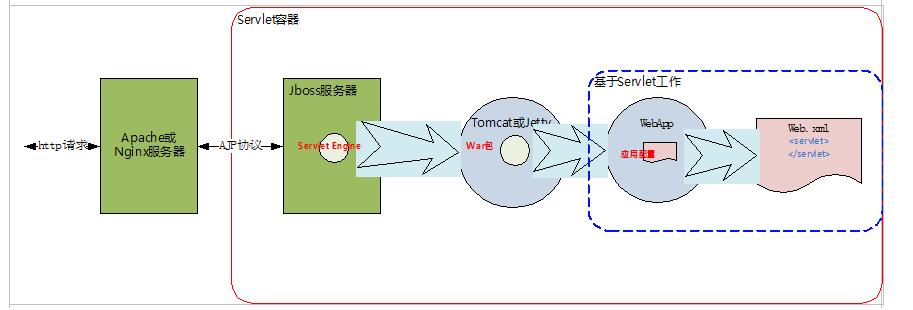


Accetptor 线程将会为这个请求创建 ConnectorEndPoint。HttpConnection 用来表示这个连接是一个 HTTP 协议的连接，它会创建 HttpParse 类解析 HTTP 协议，并且会创建符合 HTTP 协议的 Request 和 Response 对象。接下去就是将这个线程交给队列线程池去执行了。

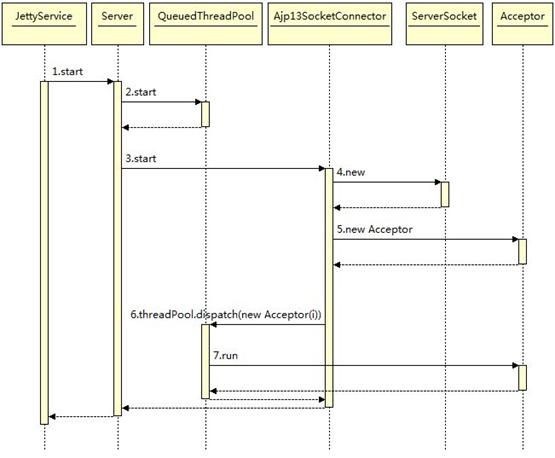
* 1. **基于 AJP 协议工作**

通常一个 web 服务站点的后端服务器不是将 Java 的应用服务器直接暴露给服务访问者，而是在应用服务器，如 Jboss 的前面在加一个 web 服务器，如 Apache 或者 nginx，我想这个原因大家应该很容易理解，如做日志分析、负载均衡、权限控制、防止恶意请求以及静态资源预加载等等。

Web服务端架构图：

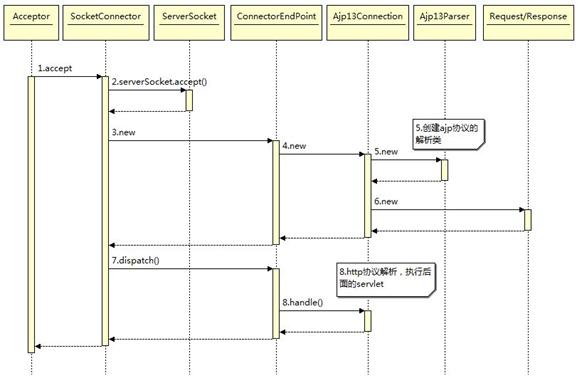


这种架构下 servlet 引擎就不需要解析和封装返回的 HTTP 协议，因为 HTTP 协议的解析工作已经在 Apache 或 Nginx 服务器上完成了，Jboss 只要基于更加简单的 AJP 协议工作就行了，这样能加快请求的响应速度。



对比 HTTP 协议的时序图可以发现，它们的逻辑几乎是相同的，不同的是替换了一个类 Ajp13Parserer 而不是 HttpParser，它定义了如何处理 AJP 协议以及需要哪些类来配合。实际上在 AJP 处理请求相比较 HTTP 时唯一的不同就是在读取到 socket 数据包时，如何来转换这个数据包，是按照 HTTP 协议的包格式来解析就是 HttpParser，按照 AJP 协议来解析就是 Ajp13Parserer。封装返回的数据也是如此。

让 Jetty 工作在 AJP 协议下，需要配置 connector 的实现类为 Ajp13SocketConnector，这个类继承了 SocketConnector 类，覆盖了父类的 newConnection 方法，为的是创建 Ajp13Connection 对象而不是 HttpConnection。



Ajp13Parser 类解析 AJP 协议，时序图如上所示。

1. **基于NIO方式工作（？）**

Jetty 的默认 connector 是 NIO 方式。

Selector selector = Selector.open();

ServerSocketChannel ssc = ServerSocketChannel.open();

ssc.configureBlocking( false );

SelectionKey key = ssc.register( selector, SelectionKey.OP\_ACCEPT );

ServerSocketChannel ss = (ServerSocketChannel)key.channel();

SocketChannel sc = ss.accept();

sc.configureBlocking( false );

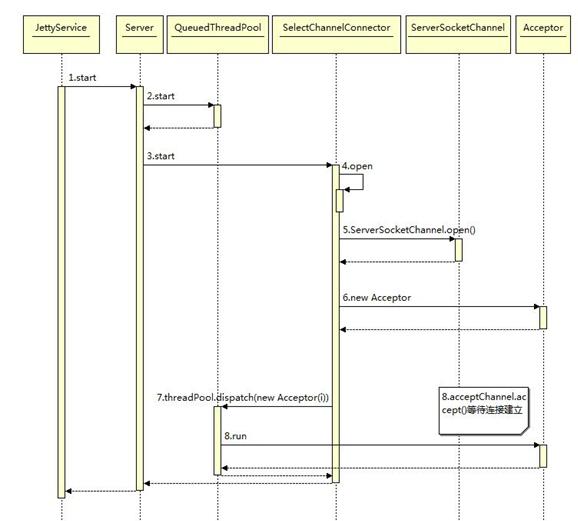
SelectionKey newKey = sc.register( selector, SelectionKey.OP\_READ );

Set selectedKeys = selector.selectedKeys();

创建一个 Selector 相当于一个观察者，打开一个 Server 端通道，把这个 server 通道注册到观察者上并且指定监听的事件。然后遍历这个观察者观察到事件，取出感兴趣的事件再处理。这里有个最核心的地方就是，我们不需要为每个被观察者创建一个线程来监控它随时发生的事件。而是把这些被观察者都注册一个地方统一管理，然后由它把触发的事件统一发送给感兴趣的程序模块。这里的核心是能够统一的管理每个被观察者的事件，所以我们就可以把服务端上每个建立的连接传送和接受数据作为一个事件统一管理，这样就不必要每个连接需要一个线程来维护了。

这里需要注意的地方时，很多人认为监听 SelectionKey.OP\_ACCEPT 事件就已经是非阻塞方式了，其实 Jetty 仍然是用一个线程来监听客户端的连接请求，当接受到请求后，把这个请求再注册到 Selector 上，然后才是非阻塞方式执行。这个地方还有一个容易引起误解的地方是：认为 Jetty 以 NIO 方式工作只会有一个线程来处理所有的请求，甚至会认为不同用户会在服务端共享一个线程从而会导致基于 ThreadLocal 的程序会出现问题，其实从 Jetty 的源码中能够发现，真正共享一个线程的处理只是在监听不同连接的数据传送事件上，比如有多个连接已经建立，传统方式是当没有数据传输时，线程是阻塞的也就是一直在等待下一个数据的到来，而 NIO 的处理方式是只有一个线程在等待所有连接的数据的到来，而当某个连接数据到来时 Jetty 会把它分配给这个连接对应的处理线程去处理，所以不同连接的处理线程仍然是独立的。

Jetty 的 NIO 处理方式和 Tomcat 的几乎一样，唯一不同的地方是在如何把监听到事件分配给对应的连接的处理方式。从测试效果来看 Jetty 的 NIO 处理方式更加高效。下面是 Jetty 的 NIO 处理时序图：



1. **处理请求**

实际上 Jetty 的工作方式非常简单，当 Jetty 接受到一个请求时，Jetty 就把这个请求交给在 Server 中注册的代理 Handler 去执行，如何执行你注册的 Handler，同样由你去规定，Jetty 要做的就是调用你注册的第一个 Handler 的 handle(String target, Request baseRequest, HttpServletRequest request, HttpServletResponse response) 方法，接下去要怎么做，完全由你决定。

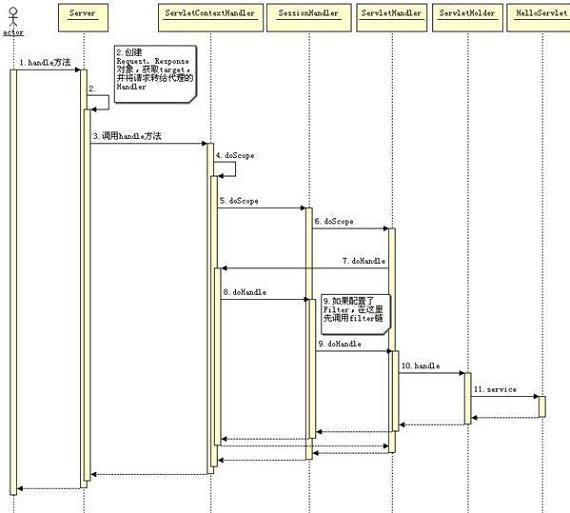
要能接受一个 web 请求访问，首先要创建一个 ContextHandler，如下代码所示：

|  |
| --- |
| Server server = new Server(8080);  ContextHandler context = new ContextHandler();  context.setContextPath("/");  context.setResourceBase(".");  context.setClassLoader(Thread.currentThread().getContextClassLoader());  server.setHandler(context);  context.setHandler(new HelloHandler());  server.start();  server.join(); |

当我们在浏览器里敲入 http://localhost:8080 时的请求将会代理到 Server 类的 handle 方法，Server 的 handle 的方法将请求代理给 ContextHandler 的 handle 方法，ContextHandler 又调用 HelloHandler 的 handle 方法。这个调用方式是不是和 Servlet 的工作方式类似，在启动之前初始化，然后创建对象后调用 Servlet 的 service 方法。在 Servlet 的 API 中我通常也只实现它的一个包装好的类，在 Jetty 中也是如此，虽然 ContextHandler 也只是一个 Handler，但是这个 Handler 通常是由 Jetty 帮你实现了，我们一般只要实现一些我们具体要做的业务逻辑有关的 Handler 就好了，而一些流程性的或某些规范的 Handler，我们直接用就好了，如下面的关于 Jetty 支持 Servlet 的规范的 Handler 就有多种实现，下面是一个简单的 HTTP 请求的流程。

访问一个 Servlet 的代码：

|  |
| --- |
| Server server = new Server();  Connector connector = new SelectChannelConnector();  connector.setPort(8080);  server.setConnectors(new Connector[]{ connector });  ServletContextHandler root = new  ServletContextHandler(null,"/",ServletContextHandler.SESSIONS);  server.setHandler(root);  root.addServlet(new ServletHolder(new  org.eclipse.jetty.embedded.HelloServlet("Hello")),"/");  server.start();  server.join(); |

创建一个 ServletContextHandler 并给这个 Handler 添加一个 Servlet，这里的 ServletHolder 是 Servlet 的一个装饰类，它十分类似于 Tomcat 中的 StandardWrapper。下面是请求这个 Servlet 的时序图：  


上图可以看出 Jetty 处理请求的过程就是 Handler 链上 handle 方法的执行过程，在这里需要解释的一点是 ScopeHandler 的处理规则，ServletContextHandler、SessionHandler 和 ServletHandler 都继承了 ScopeHandler，那么这三个类组成一个 Handler 链，它们的执行规则是：ServletContextHandler.handleServletContextHandler.doScope SessionHandler. doScopeServletHandler. doScopeServletContextHandler. doHandleSessionHandler. doHandleServletHandler. doHandle，它这种机制使得我们可以在 doScope 做一些额外工作。

额外参考资料：

[《 Tomcat 系统架构与设计模式》](http://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)

《[Servlet工作原理解析](http://www.ibm.com/developerworks/cn/java/j-lo-servlet/)》