参考：<http://blog.csdn.net/qing_2012/article/details/8276789>

**Jetty 的工作原理以及与 Tomcat 的比较**

许令波，developerWorks 中国网站最佳作者，现就职于淘宝网，是一名 Java 开发工程师。对大型互联网架构设计颇感兴趣，喜欢钻研开源框架的设计原理。有时间将学到的知识整理成文章，也喜欢记录下工作和生活中的一些思考。个人网站是：http://xulingbo.net。

**简介：** Jetty 应该是目前最活跃也是很有前景的一个 Servlet 引擎。本文将介绍 Jetty 基本**[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "_blank)**与基本的工作原理：您将了解到 Jetty 的基本体系结构；Jetty 的启动过程；Jetty 如何接受和处理用户的请求。你还将了解到 AJP 的一些细节：Jetty 如何基于 AJP 工作；以及 Jetty 如何集成到 Jboss；最后我们将比较一下两个 Servlet 引擎：Tomcat 和 Jetty 的优缺点。

**发布日期：** 2011 年 9 月 21 日   
**级别：** 高级   
**访问情况 ：** 32028 次浏览   
**评论：** 10 ([查看](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#icomments) | [添加评论](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/) - 登录)

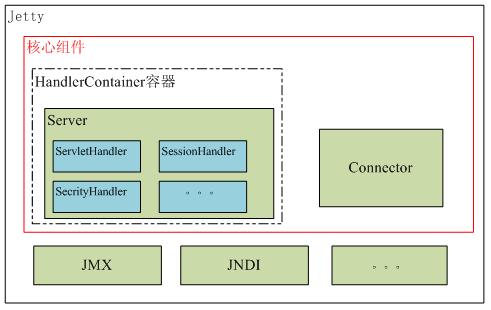
平均分 5 星 共 124 个评分 平均分 (124个评分)  
[为本文评分](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#iratings)

Jetty 的基本架构

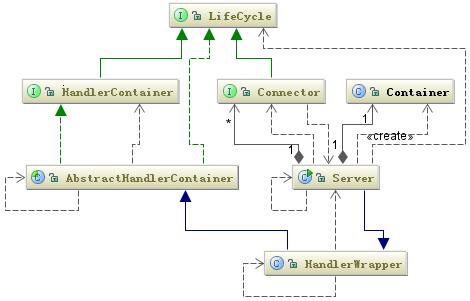
Jetty 目前的是一个比较被看好的 Servlet 引擎，它的架构比较简单，也是一个可扩展性和非常灵活的应用服务器，它有一个基本数据模型，这个数据模型就是 Handler，所有可以被扩展的组件都可以作为一个 Handler，添加到 Server 中，Jetty 就是帮你管理这些 Handler。

Jetty 的基本架构

下图是 Jetty 的基本架构图，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，它类似于 Tomcat 的 [**Container**](http://lib.csdn.net/base/docker) 容器（me：tomcat的容器是catalina），Jetty 与 Tomcat 的比较在后面详细介绍。Jetty 中另外一个必不可少的组件是 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（me：而tomcat也包含connector，只是tomcat的connector会将请求转发给engine来处理）去执行。

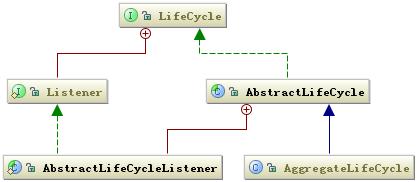
**图 1. Jetty 的基本架构**  


Jetty 中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。

**图 2. Jetty 的主要组件的类图**  


从上图可以看出整个 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。

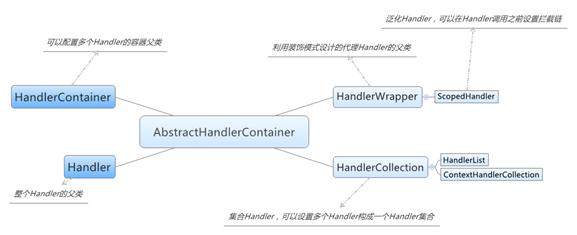
整个 Jetty 的所有组件的生命周期管理是基于观察者模板设计，它和 Tomcat 的管理是类似的。下面是 LifeCycle 的类关系图



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，关于观察者模式可以参考 [《Tomcat系统架构与设计模式，第2部分:设计模式分析》](http://www.ibm.com/developerworks/cn/java/j-lo-tomcat2/index.html)一文中关于观察者模式的讲解）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。

Handler 的体系结构

前面所述 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。下面总结了一下 Handler 的种类及作用：

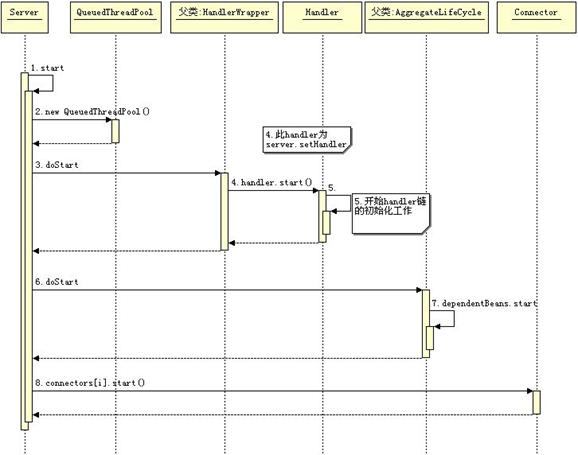
**图 3. Handler 的体系结构（[查看大图](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/image008.png" \t "_blank)）**  


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。

[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

Jetty 的启动过程

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。下面是 Jetty 的启动时序图：

**图 4. Jetty 的启动流程**  


因为 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。

[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

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。下面分别介绍 Jetty 如何基于这两种协议工作，并且它们如何建立连接和接受请求的。

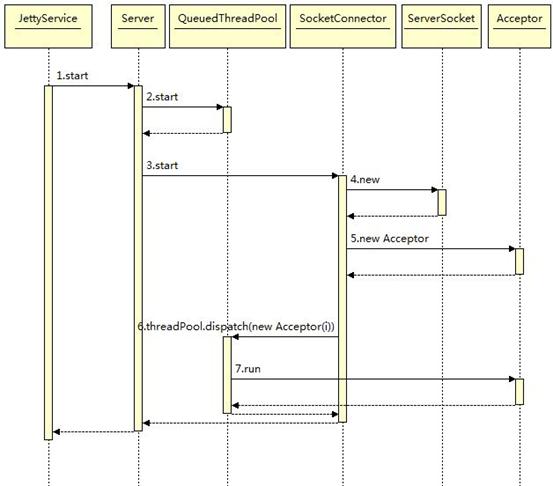
基于 HTTP 协议工作

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，有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。

下图是 Jetty 启动创建建立连接的时序图：

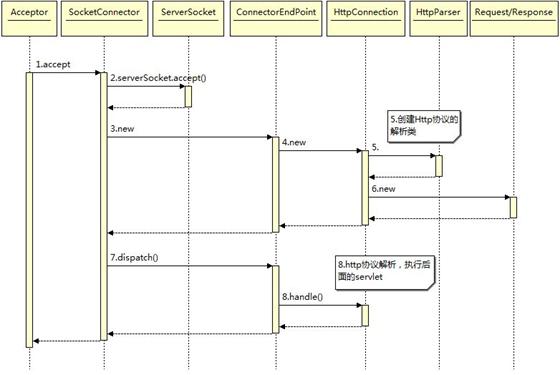
**图 5. 建立连接的时序图**  


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：

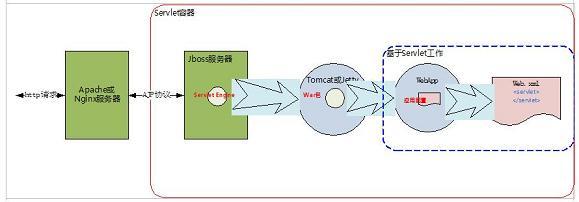
**图 6. 处理连接时序图**  


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。

基于 AJP 工作

通常一个 web 服务站点的后端服务器不是将 [**Java**](http://lib.csdn.net/base/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。

下图是通常的 web 服务端的架构图：

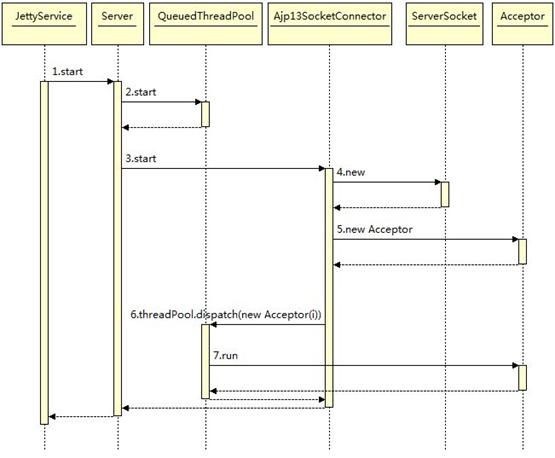
**图 7. Web 服务端架构（[查看大图](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/image017-large.jpg" \t "_blank)）**  


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。

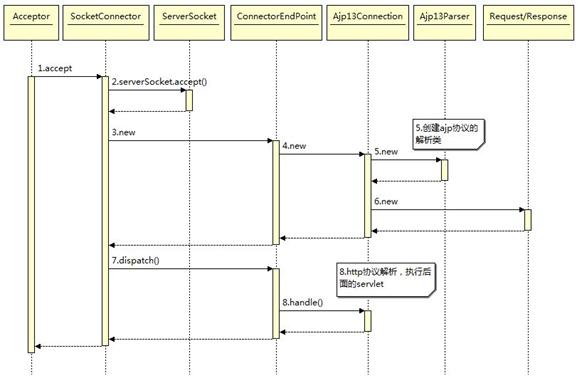
对比 HTTP 协议的时序图可以发现，它们的逻辑几乎是相同的，不同的是替换了一个类 Ajp13Parserer 而不是 HttpParser，它定义了如何处理 AJP 协议以及需要哪些类来配合。

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。如下图表示的是 Jetty 创建连接环境时序图：



与 HTTP 方式唯一不同的地方的就是将 SocketConnector 类替换成了 Ajp13SocketConnector。改成 Ajp13SocketConnector 的目的就是可以创建 Ajp13Connection 类，表示当前这个连接使用的是 AJP 协议，所以需要用 Ajp13Parser 类解析 AJP 协议，处理连接的逻辑都是一样的。如下时序图所示：



基于 NIO 方式工作

前面所描述的 Jetty 建立客户端连接到处理客户端的连接都是基于 BIO 的方式，它也支持另外一种 NIO 的处理方式，其中 Jetty 的默认 connector 就是 NIO 方式。

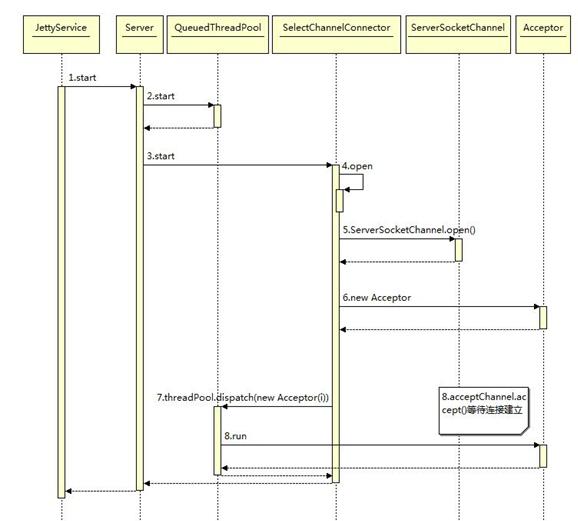
关于 NIO 的工作原理可以参考 developerworks 上关于 NIO 的文章，通常 NIO 的工作原型如下：

|  |
| --- |
| 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](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "_blank)**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：



[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

处理请求

下面看一下 Jetty 是如何处理一个 HTTP 请求的。

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：

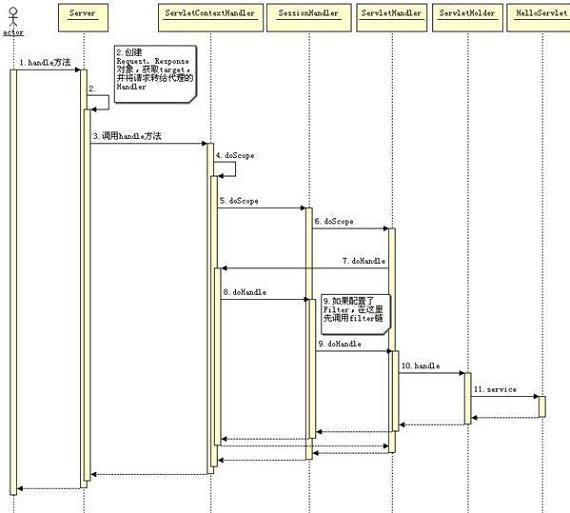
|  |
| --- |
| Server server = new Server(8080);  ContextHandler context = new ContextHandler();  context.setContextPath("/");  context.setResourceBase(".");  context.setClassLoader(Thread.currentThread().getContextClassLoader());  server.setHandler(context);  context.setHandler(new HelloHandler());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

**图 8. Jetty 处理请求的时序图**  


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。

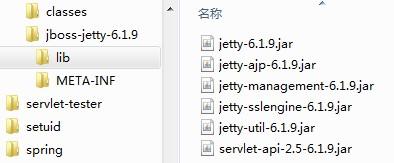
[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

与 Jboss 集成

前面介绍了 Jetty 可以基于 AJP 协议工作，在正常的企业级应用中，Jetty 作为一个 Servlet 引擎都是基于 AJP 协议工作的，所以它前面必然有一个服务器，通常情况下与 Jboss 集成的可能性非常大，这里介绍一下如何与 Jboss 集成。

Jboss 是基于 JMX 的架构，那么只要符合 JMX 规范的系统或框架都可以作为一个组件加到 Jboss 中，扩展 Jboss 的功能。Jetty 作为主要的 Servlet 引擎当然支持与 Jboss 集成。具体集成方式如下：

Jetty 作为一个独立的 Servlet 引擎集成到 Jboss 需要继承 Jboss 的 AbstractWebContainer 类，这个类实现的是模板模式，其中有一个抽象方法需要子类去实现，它是 getDeployer，可以指定创建 web 服务的 Deployer。Jetty 工程中有个 jetty-jboss 模块，编译这个模块就会产生一个 SAR 包，或者可以直接从官网下载一个 SAR 包。解压后如下图：

**图 9. jboss-jetty 目录**  


在 jboss-jetty-6.1.9 目录下有一个 webdefault.xml 配置文件，这个文件是 Jetty 的默认 web.xml 配置，在 META-INF 目录发下发现 jboss-service.xml 文件，这个文件配置了 MBean，如下：

|  |
| --- |
| <mbean code="org.jboss.jetty.JettyService"  name="jboss.web:service=WebServer" xmbean-dd="META-INF/webserver-xmbean.xml"> |

同样这个 org.jboss.jetty.JettyService 类也是继成 org.jboss.web.AbstractWebContainer 类，覆盖了父类的 startService 方法，这个方法直接调用 jetty.start 启动 Jetty。

[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

与 Tomcat 的比较

Tomcat 和 Jetty 都是作为一个 Servlet 引擎应用的比较广泛，可以将它们比作为中国与美国的关系，虽然 Jetty 正常成长为一个优秀的 Servlet 引擎，但是目前的 Tomcat 的地位仍然难以撼动。相比较来看，它们都有各自的优点与缺点。

Tomcat 经过长时间的发展，它已经广泛的被市场接受和认可，相对 Jetty 来说 Tomcat 还是比较稳定和成熟，尤其在企业级应用方面，Tomcat 仍然是第一选择。但是随着 Jetty 的发展，Jetty 的市场份额也在不断提高，至于原因就要归功与 Jetty 的很多优点了，而这些优点也是因为 Jetty 在技术上的优势体现出来的。

架构比较

从架构上来说，显然 Jetty 比 Tomcat 更加简单，如果你对 Tomcat 的架构还不是很了解的话，建议你先看一下 [《Tomcat系统架构与设计模式》](http://www.ibm.com/developerworks/cn/java/j-lo-tomcat2/index.html)这篇文章。

Jetty 的架构从前面的分析可知，它的所有组件都是基于 Handler 来实现，当然它也支持 JMX。但是主要的功能扩展都可以用 Handler 来实现。可以说 Jetty 是面向 Handler 的架构，就像 [**spring**](http://lib.csdn.net/base/javaee) 是面向 Bean 的架构，iBATIS 是面向 statement 一样，而 Tomcat 是以多级容器构建起来的，它们的架构设计必然都有一个“元神”，所有以这个“元神“构建的其它组件都是肉身。

从设计模板角度来看 Handler 的设计实际上就是一个责任链模式，接口类 HandlerCollection 可以帮助开发者构建一个链，而另一个接口类 ScopeHandler 可以帮助你控制这个链的访问顺序。另外一个用到的设计模板就是观察者模式，用这个设计模式控制了整个 Jetty 的生命周期，只要继承了 LifeCycle 接口，你的对象就可以交给 Jetty 来统一管理了。所以扩展 Jetty 非常简单，也很容易让人理解，整体架构上的简单也带来了无比的好处，Jetty 可以很容易被扩展和裁剪。

相比之下，Tomcat 要臃肿很多，Tomcat 的整体设计上很复杂，前面说了 Tomcat 的核心是它的容器的设计，从 Server 到 Service 再到 engine 等 container 容器。作为一个应用服务器这样设计无口厚非，容器的分层设计也是为了更好的扩展，这是这种扩展的方式是将应用服务器的内部结构暴露给外部使用者，使得如果想扩展 Tomcat，开发人员必须要首先了解 Tomcat 的整体设计结构，然后才能知道如何按照它的规范来做扩展。这样无形就增加了对 Tomcat 的学习成本。不仅仅是容器，实际上 Tomcat 也有基于责任链的设计方式，像串联 Pipeline 的 Vavle 设计也是与 Jetty 的 Handler 类似的方式。要自己实现一个 Vavle 与写一个 Handler 的难度不相上下。表面上看，Tomcat 的功能要比 Jetty 强大，因为 Tomcat 已经帮你做了很多工作了，而 Jetty 只告诉，你能怎么做，如何做，有你去实现。

打个比方，就像小孩子学数学，Tomcat 告诉你 1+1=2，1+2=3，2+2=4 这个结果，然后你可以根据这个方式得出 1+1+2=4，你要计算其它数必须根据它给你的公式才能计算，而 Jetty 是告诉你加减乘除的**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)**规则，然后你就可以根据这个规则自己做运算了。所以你一旦掌握了 Jetty，Jetty 将变得异常强大。

性能比较

单纯比较 Tomcat 与 Jetty 的性能意义不是很大，只能说在某种使用场景下，它表现的各有差异。因为它们面向的使用场景不尽相同。从架构上来看 Tomcat 在处理少数非常繁忙的连接上更有优势，也就是说连接的生命周期如果短的话，Tomcat 的总体性能更高。

而 Jetty 刚好相反，Jetty 可以同时处理大量连接而且可以长时间保持这些连接。例如像一些 web 聊天应用非常适合用 Jetty 做服务器，像淘宝的 web 旺旺就是用 Jetty 作为 Servlet 引擎。

另外由于 Jetty 的架构非常简单，作为服务器它可以按需加载组件，这样不需要的组件可以去掉，这样无形可以减少服务器本身的内存开销，处理一次请求也是可以减少产生的临时对象，这样性能也会提高。另外 Jetty 默认使用的是 NIO 技术在处理 I/O 请求上更占优势，Tomcat 默认使用的是 BIO，在处理静态资源时，Tomcat 的性能不如 Jetty。

特性比较

作为一个标准的 Servlet 引擎，它们都支持标准的 Servlet 规范，还有 [**Java**](http://lib.csdn.net/base/java)EE 的规范也都支持，由于 Tomcat 的使用的更加广泛，它对这些支持的更加全面一些，有很多特性 Tomcat 都直接集成进来了。但是 Jetty 的应变更加快速，这一方面是因为 Jetty 的开发社区更加活跃，另一方面也是因为 Jetty 的修改更加简单，它只要把相应的组件替换就好了，而 Tomcat 的整体结构上要复杂很多，修改功能比较缓慢。所以 Tomcat 对最新的 Servlet 规范的支持总是要比人们预期的要晚。

[回页首](http://www.ibm.com/developerworks/cn/java/j-lo-jetty/#ibm-pcon)

总结

本文介绍了目前 Java 服务端中一个比较流行应用服务器 Jetty，介绍了它的基本架构和工作原理以及如何和 Jboss 工作，最后与 Tomcat 做了比较。在看这篇文章的时候最好是结合我前面写的两篇文章《 Tomcat 系统架构与设计模式》和《 Servlet 工作原理解析》以及这些系统的源代码，耐心的都看一下会让你对 Java 服务端有个总体的了解。

参考资料

**学习**

* 查看文章 [《 Tomcat 系统架构与设计模式》](http://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html)（developerWorks，2010 年 5 月）：了解 Tomcat 中容器的体系结构，基本的工作原理，以及 Tomcat 中使用的经典的设计模式介绍。
* [Servlet工作原理解析](http://www.ibm.com/developerworks/cn/java/j-lo-servlet/)，（developerWorks，2011 年 2 月）：以 Tomcat 为例了解 Servlet 容器是如何工作的？一个 Web 工程在 Servlet 容器中是如何启动的？ Servlet 容器如何解析你在 web.xml 中定义的 Servlet ？用户的请求是如何被分配给指定的 Servlet 的？ Servlet 容器如何管理 Servlet 生命周期？你还将了解到最新的 Servlet 的 API 的类层次结构，以及 Servlet 中一些难点问题的分析。
* [Tomcat vs Jetty](http://www.webtide.com/choose/jetty.jsp)，对 Tomcat 与 Jetty 的做了比较。
* [HTTP 协议](http://www.w3.org/Protocols/)，W3C 关于 HTTP 协议的详细描述。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：这里有数百篇关于 Java 编程各个方面的文章。

**讨论**

* 加入 [developerWorks 中文社区](http://www.ibm.com/developerworks/cn/community/" \t "_blank)。查看开发人员推动的博客、论坛、组和维基，并与其他 developerWorks 用户交流。

关于作者

许令波，developerWorks 中国网站最佳作者，现就职于淘宝网，是一名 Java 开发工程师。对大型互联网架构设计颇感兴趣，喜欢钻研开源框架的设计原理。有时间将学到的知识整理成文章，也喜欢记录下工作和生活中的一些思考。个人网站是：http://xulingbo.net。