

29 | 比较：Jetty如何实现具有上下文信息的责任链？

2019-07-16 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 09:59 大小 9.15M

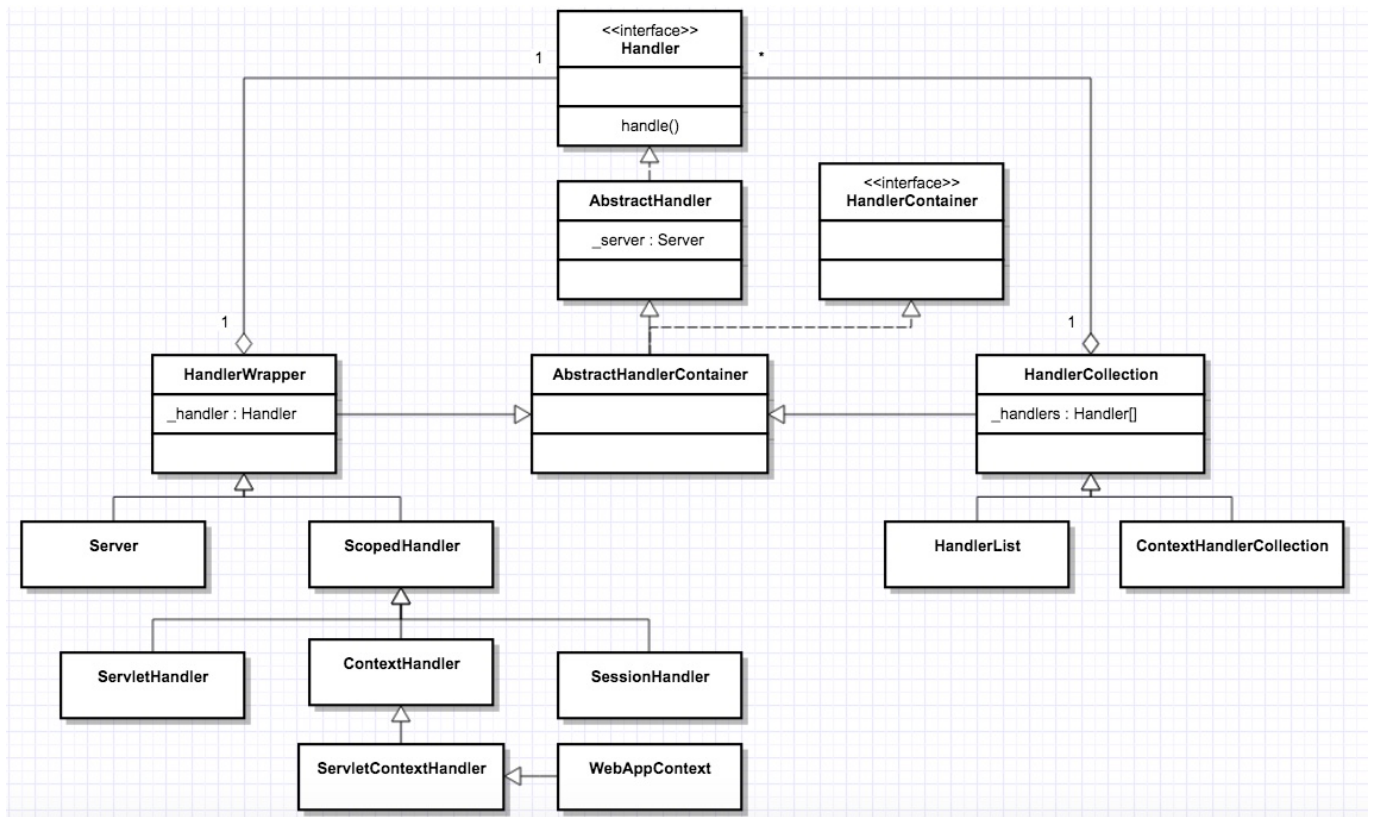


我们知道 Tomcat 和 Jetty 的核心功能是处理请求，并且请求的处理者不止一个，因此 Tomcat 和 Jetty 都实现了责任链模式，其中 Tomcat 是通过 Pipeline-Valve 来实现的，而 Jetty 是通过 HandlerWrapper 来实现的。HandlerWrapper 中保存了下一个 Handler 的引用，将各 Handler 组成一个链表，像下面这样：

WebApplicationContext -> SessionHandler -> SecurityHandler -> ServletHandler

这样链中的 Handler 从头到尾能被依次调用，除此之外，Jetty 还实现了“回溯”的链式调用，那就是从头到尾依次链式调用 Handler 的**方法 A**，完成后再回到头节点，再进行一次链式调用，只不过这一次调用另一个**方法 B**。你可能会问，一次链式调用不就够了吗，为什么还要回过头再调一次呢？这是因为一次请求到达时，Jetty 需要先调用各 Handler 的初始化方法，之后再调用各 Handler 的请求处理方法，并且初始化必须在请求处理之前完成。

而 Jetty 是通过 ScopedHandler 来做到这一点的，那 ScopedHandler 跟 HandlerWrapper 有什么关系呢？ScopedHandler 是 HandlerWrapper 的子类，我们还是通过一张图来回顾一下各种 Handler 的继承关系：



从图上我们看到，ScopedHandler 是 Jetty 非常核心的一个 Handler，跟 Servlet 规范相关的 Handler，比如 ContextHandler、SessionHandler、ServletHandler、WebappContext 等都直接或间接地继承了 ScopedHandler。

今天我就分析一下 ScopedHandler 是如何实现“回溯”的链式调用的。

HandlerWrapper

为了方便理解，我们先来回顾一下 HandlerWrapper 的源码：

[复制代码](#)

```
1 public class HandlerWrapper extends AbstractHandlerContainer
2 {
3     protected Handler _handler;
4
5     @Override
6     public void handle(String target,
7                       Request baseRequest,
8                       HttpServletRequest request,
```

```


9             HttpServletResponse response)
10             throws IOException, ServletException
11     {
12         Handler handler=_handler;
13         if (handler!=null)
14             handler.handle(target,baseRequest, request, response);
15     }
16 }

```

从代码可以看到它持有下一个 Handler 的引用，并且会在 handle 方法里调用下一个 Handler。

ScopedHandler

ScopedHandler 的父类是 HandlerWrapper，ScopedHandler 重写了 handle 方法，在 HandlerWrapper 的 handle 方法的基础上引入了 doScope 方法。


 复制代码

```

1 public final void handle(String target,
2                         Request baseRequest,
3                         HttpServletRequest request,
4                         HttpServletResponse response)
5                         throws IOException, ServletException
6 {
7     if (isStarted())
8     {
9         if (_outerScope==null)
10             doScope(target,baseRequest,request, response);
11         else
12             doHandle(target,baseRequest,request, response);
13     }
14 }

```

上面的代码中是根据 `_outerScope` 是否为 null 来判断是使用 `doScope` 还是 `doHandle` 方法。那 `_outerScope` 又是什么呢？`_outerScope` 是 ScopedHandler 引入的一个辅助变量，此外还有一个 `_nextScope` 变量。

 复制代码

```


1 protected ScopedHandler _outerScope;

```

```
2 protected ScopedHandler _nextScope;
3
4 private static final ThreadLocal<ScopedHandler> __outerScope= new ThreadLocal<ScopedHan
```

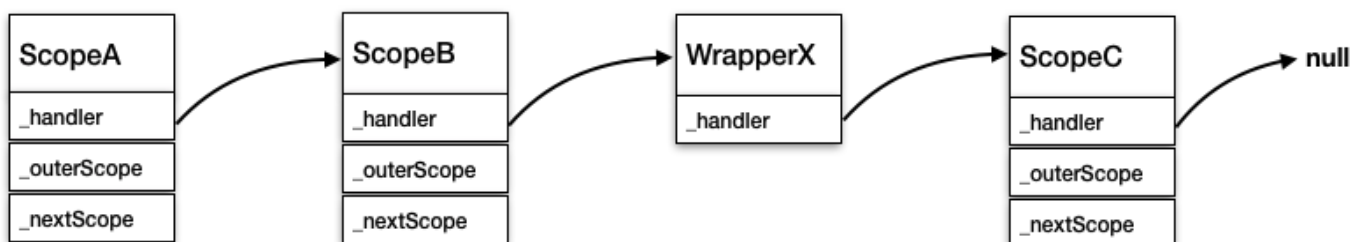
我们看到__outerScope是一个 ThreadLocal 变量，ThreadLocal 表示线程的私有数据，跟特定线程绑定。需要注意的是__outerScope实际上保存了一个 ScopedHandler。

下面通过我通过一个例子来说明_outScope和_nextScope的含义。我们知道 ScopedHandler 继承自 HandlerWrapper，所以也是可以形成 Handler 链的，Jetty 的源码注释中给出了下面这样一个例子：

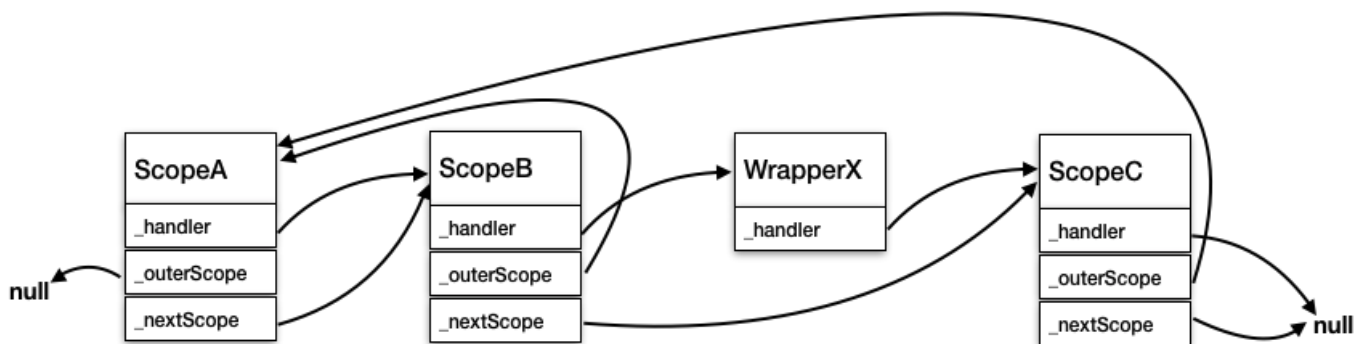
 复制代码

```
1 ScopedHandler scopedA;
2 ScopedHandler scopedB;
3 HandlerWrapper wrapperX;
4 ScopedHandler scopedC;
5
6 scopedA.setHandler(scopedB);
7 scopedB.setHandler(wrapperX);
8 wrapperX.setHandler(scopedC)
```

经过上面的设置之后，形成的 Handler 链是这样的：



上面的过程只是设置了_handler变量，那_outScope和_nextScope需要设置成什么样呢？为了方便你理解，我们先来看最后的效果图：



从上图我们看到：scopedA 的 `_nextScope`=scopedB，scopedB 的 `_nextScope`=scopedC，为什么 scopedB 的 `_nextScope` 不是 WrapperX 呢，因为 WrapperX 不是一个 ScopedHandler。scopedC 的 `_nextScope` 是 null（因为它是链尾，没有下一个节点）。因此我们得出一个结论：`_nextScope` 指向下一个 Scoped 节点的引用，由于 WrapperX 不是 Scoped 节点，它没有 `_outerScope` 和 `_nextScope` 变量。

注意到 scopedA 的 `_outerScope` 是 null，scopedB 和 scopedC 的 `_outerScope` 都是指向 scopedA，即 `_outerScope` 指向的是当前 Handler 链的头节点，头节点本身 `_outerScope` 为 null。

弄清楚了 `_outerScope` 和 `_nextScope` 的含义，下一个问题就是对于一个 ScopedHandler 对象如何设置这两个值以及在何时设置这两个值。答案是在组件启动的时候，下面是 ScopedHandler 中的 `doStart` 方法源码：

[复制代码](#)

```

1  @Override
2  protected void doStart() throws Exception
3  {
4      try
5      {
6          // 请注意 _outerScope 是一个实例变量，而 __outerScope 是一个全局变量。先读取全局的线程
7          _outerScope=__outerScope.get();
8
9          // 如果全局的 __outerScope 还没有被赋值，说明执行 doStart 方法的是头节点
10         if (_outerScope==null)
11             //handler 链的头节点将自己的引用填充到 __outerScope
12             __outerScope.set(this);
13
14         // 调用父类 HandlerWrapper 的 doStart 方法
15         super.doStart();
16         // 各 Handler 将自己的 _nextScope 指向下一个 ScopedHandler
17         _nextScope= getChildHandlerByClass(ScopedHandler.class);
18     }
19     finally

```

```
20     {
21         if (_outerScope==null)
22             __outerScope.set(null);
23     }
24 }
```

你可能会问，为什么要设计这样一个全局的__outerScope，这是因为这个变量不能通过方法参数在 Handler 链中进行传递，但是在形成链的过程中又需要用到它。

你可以想象，当 scopedA 调用 start 方法时，会把自己填充到__scopeHandler中，接着 scopedA 调用super.doStart。由于 scopedA 是一个 HandlerWrapper 类型，并且它持有的_handler引用指向的是 scopedB，所以super.doStart实际上会调用 scopedB 的 start 方法。


这个方法里同样会执行 scopedB 的 doStart 方法，不过这次__outerScope.get方法返回的不是 null 而是 scopedA 的引用，所以 scopedB 的_outScope被设置为 scopedA。

接着super.dostart会进入到 scopedC，也会将 scopedC 的_outScope指向 scopedA。到了 scopedC 执行 doStart 方法时，它的_handler属性为 null（因为它是 Handler 链的最后一个），所以它的super.doStart会直接返回。接着继续执行 scopedC 的 doStart 方法的下一行代码：

 复制代码

```
1 _nextScope=(ScopedHandler)getChildHandlerByClass(ScopedHandler.class)
```


对于 HandlerWrapper 来说 getChildHandlerByClass 返回的就是其包装的_handler对象，这里返回的就是 null。所以 scopedC 的_nextScope为 null，这段方法结束返回后继续执行 scopedB 中的 doStart 中，同样执行这句代码：

 复制代码

```
1 _nextScope=(ScopedHandler)getChildHandlerByClass(ScopedHandler.class)
```

因为 `scopedB` 的 `_handler` 引用指向的是 `scopedC`，所以 `getChildHandlerByClass` 返回的结果就是 `scopedC` 的引用，即 `scopedB` 的 `_nextScope` 指向 `scopedC`。


同理 `scopedA` 的 `_nextScope` 会指向 `scopedB`。`scopedA` 的 `doStart` 方法返回之后，其 `_outScope` 为 `null`。请注意执行到这里只有 `scopedA` 的 `_outScope` 为 `null`，所以 `doStart` 中 `finally` 部分的逻辑被触发，这个线程的 `ThreadLocal` 变量又被设置为 `null`。

 复制代码

```
1 finally
2 {
3     if (_outerScope==null)
4         __outerScope.set(null);
5 }
```

你可能会问，费这么大劲设置 `_outScope` 和 `_nextScope` 的值到底有什么用？如果你觉得上面的过程比较复杂，可以跳过这个过程，直接通过图来理解 `_outScope` 和 `_nextScope` 的值，而这样设置的目的是用来控制 `doScope` 方法和 `doHandle` 方法的调用顺序。

实际上在 `ScopedHandler` 中对于 `doScope` 和 `doHandle` 方法是没有具体实现的，但是提供了 `nextHandle` 和 `nextScope` 两个方法，下面是它们的源码：

 复制代码


```
1 public void doScope(String target,
2                     Request baseRequest,
3                     HttpServletRequest request,
4                     HttpServletResponse response)
5     throws IOException, ServletException
6 {
7     nextScope(target, baseRequest, request, response);
8 }
9
10 public final void nextScope(String target,
11                             Request baseRequest,
12                             HttpServletRequest request,
13                             HttpServletResponse response)
14     throws IOException, ServletException
15 {
16     if (_nextScope!=null)
17         _nextScope.doScope(target, baseRequest, request, response);
18     else if (_outerScope!=null)
```

```

19         _outerScope.doHandle(target,baseRequest,request, response);
20     else
21         doHandle(target,baseRequest,request, response);
22 }
23
24 public abstract void doHandle(String target,
25                               Request baseRequest,
26                               HttpServletRequest request,
27                               HttpServletResponse response)
28     throws IOException, ServletException;
29
30
31 public final void nextHandle(String target,
32                               final Request baseRequest,
33                               HttpServletRequest request,
34                               HttpServletResponse response)
35     throws IOException, ServletException
36 {
37     if (_nextScope!=null && _nextScope==_handler)
38         _nextScope.doHandle(target,baseRequest,request, response);
39     else if (_handler!=null)
40         super.handle(target,baseRequest,request,response);
41 }

```

从 nextHandle 和 nextScope 方法大致上可以猜到 doScope 和 doHandle 的调用流程。我通过一个调用栈来帮助你理解：

 复制代码

```

1 A.handle(...)
2     A.doScope(...)
3         B.doScope(...)
4             C.doScope(...)
5                 A.doHandle(...)
6                     B.doHandle(...)
7                         X.handle(...)
8                             C.handle(...)
9                                 C.doHandle(...)

```

因此通过设置 `_outerScope` 和 `_nextScope` 的值，并且在代码中判断这些值并采取相应的动作，目的就是让 ScopedHandler 链上的 **doScope 方法在 doHandle、handle 方法之前执行**。并且不同 ScopedHandler 的 doScope 都是按照它在链上的先后顺序执行的，doHandle 和 handle 方法也是如此。

这样 ScopedHandler 帮我们把调用框架搭好了，它的子类只需要实现 doScope 和 doHandle 方法。比如在 doScope 方法里做一些初始化工作，在 doHandle 方法处理请求。


ContextHandler

接下来我们来看看 ScopedHandler 的子类 ContextHandler 是如何实现 doScope 和 doHandle 方法的。ContextHandler 可以理解为 Tomcat 中的 Context 组件，对应一个 Web 应用，它的功能是给 Servlet 的执行维护一个上下文环境，并且将请求转发到相应的 Servlet。那什么是 Servlet 执行的上下文？我们通过 ContextHandler 的构造函数来了解一下：

 复制代码

```
1 private ContextHandler(Context context, HandlerContainer parent, String contextPath)
2 {
3     //_scontext 就是 Servlet 规范中的 ServletContext
4     _scontext = context == null?new Context():context;
5
6     //Web 应用的初始化参数
7     _initParams = new HashMap<String, String>();
8     ...
9 }
```

我们看到 ContextHandler 维护了 ServletContext 和 Web 应用的初始化参数。那 ContextHandler 的 doScope 方法做了些什么呢？我们看看它的关键代码：

 复制代码


```
1 public void doScope(String target, Request baseRequest, HttpServletRequest request, Http
2 {
3     ...
4     //1. 修正请求的 URL，去掉多余的 '/', 或者加上 '/'
5     if (_compactPath)
6         target = URIUtil.compactPath(target);
7     if (!checkContext(target, baseRequest, response))
8         return;
9     if (target.length() > _contextPath.length())
10    {
11        if (_contextPath.length() > 1)
12            target = target.substring(_contextPath.length());
13        pathInfo = target;
14    }
```

```

15     else if (_contextPath.length() == 1)
16     {
17         target = URIUtil.SLASH;
18         pathInfo = URIUtil.SLASH;
19     }
20     else
21     {
22         target = URIUtil.SLASH;
23         pathInfo = null;
24     }
25
26     //2. 设置当前 Web 应用的类加载器
27     if (_classLoader != null)
28     {
29         current_thread = Thread.currentThread();
30         old_classloader = current_thread.getContextClassLoader();
31         current_thread.setContextClassLoader(_classLoader);
32     }
33
34     //3. 调用 nextScope
35     nextScope(target,baseRequest,request,response);
36
37     ...
38 }

```

从代码我们看到在 doScope 方法里主要是做了一些请求的修正、类加载器的设置，并调用 nextScope，请你注意 nextScope 调用是由父类 ScopedHandler 实现的。接着我们来 ContextHandler 的 doHandle 方法：

 复制代码

```

1 public void doHandle(String target, Request baseRequest, HttpServletRequest request, Http
2 {
3     final DispatcherType dispatch = baseRequest.getDispatcherType();
4     final boolean new_context = baseRequest.takeNewContext();
5     try
6     {
7         // 请求的初始化工作，主要是为请求添加 ServletRequestAttributeListener 监听器，并将
8         if (new_context)
9             requestInitialized(baseRequest,request);
10
11         ...
12
13         // 继续调用下一个 Handler，下一个 Handler 可能是 ServletHandler、SessionHandler ..
14         nextHandle(target,baseRequest,request,response);
15     }
16     finally

```

```
17     {
18         // 同样一个 Servlet 请求处理完毕，也要通知相应的监听器
19         if (new_context)
20             requestDestroyed(baseRequest,request);
21     }
22 }
```

从上面的代码我们看到 ContextHandler 在 doHandle 方法里分别完成了相应的请求处理工作。

本期精华

今天我们分析了 Jetty 中 ScopedHandler 的实现原理，剖析了如何实现链式调用的“回溯”。主要是确定了 doScope 和 doHandle 的调用顺序，doScope 依次调用完以后，再依次调用 doHandle，它的子类比如 ContextHandler 只需要实现 doScope 和 doHandle 方法，而不需要关心它们被调用的顺序。

这背后的原理是，ScopedHandler 通过递归的方式来设置 `_outScope` 和 `_nextScope` 两个变量，然后通过判断这些值来控制调用的顺序。递归是计算机编程的一个重要的概念，在各种面试题中也经常出现，如果你能读懂 Jetty 中的这部分代码，毫无疑问你已经掌握了递归的精髓。

另外我们进行层层递归调用中需要用到一些变量，比如 ScopedHandler 中的 `__outerScope`，它保存了 Handler 链中的头节点，但是它不是递归方法的参数，那参数怎么传递过去呢？一种可能的办法是设置一个全局变量，各 Handler 都能访问到这个变量。但这样会有线程安全的问题，因此 ScopedHandler 通过线程私有数据 ThreadLocal 来保存变量，这样既达到了传递变量的目的，又没有线程安全的问题。

课后思考

ScopedHandler 的 doStart 方法，最后一步是将线程私有变量 `__outerScope` 设置成 null，为什么需要这样做呢？

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。

深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | 新特性：Spring Boot如何使用内嵌式的Tomcat和Jetty？

下一篇 30 | 热点问题答疑（3）：Spring框架中的设计模式

精选留言 (3)

写留言



nightmare

2019-07-16

每一次请求的请求链互不影响

展开



2



despacito

2019-07-16

ScopedHandler 会有不同的实现类，而`_outerScope`是ScopedHandler里static的变量，如果不设置为null，那么不同的子类实例执行doStrat()方法的时候，会有问题



1



往事随风，顺其自然

2019-07-16

可以重新处理下一次请求

展开 ∨

