[IMG_256](https://www.cnblogs.com/diegodu/)**Tomcat 架构探索**

## **前言**

花了一个礼拜的时间阅读了 how tomcat works，本文基于此书，整理了一下Tomcat 5的基本架构，其实也没什么多复杂的东西，无非是解析Http请求，然后调用相应的Servlet。另推荐看CSAPP的网络编程那一章

## **基本架构**

Tomcat由两个模块协同合作

* connector
* container

connector 负责解析处理HTTP请求，比如说请求头,查询字符串,请求参数之类的。生成HttpRequest和HttpResponse  
之后交给container，由它负责调用相应的Servlet。

## **Connector**

Tomcat默认的Connector为HttpConnector。作为Connector必须要实现Connector这个接口。

Tomcat启动以后会开启一个线程，做一个死循环，通过ServerSocket来等待请求。一旦得到请求，生成Socket，注意这里HttpConnector并不会自己处理Socket，而是把它交给HttpProcessor。详细看下面代码，这里我只保留了关键代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public void run() {  // Loop until we receive a shutdown command  while (!stopped) {  Socket socket = null;  try {  socket = serverSocket.accept(); //等待链接  } catch (AccessControlException ace) {  log("socket accept security exception", ace);  continue;  }  // Hand this socket off to an appropriate processor  HttpProcessor processor = createProcessor();  processor.assign(socket); //这里是立刻返回的  // The processor will recycle itself when it finishes  }  } |

注意一点，上面的processor.assign(socket);是立刻返回的，并不会阻塞在那里等待。因为Tomcat不可能一次只能处理一个请求，所以是异步的，每个processor处理都是一个单独的线程。

### **HttpProcessor**

上面的代码并没有显示调用HttpProcessor的process方法，那这个方法是怎么调用的呢？我们来看一下HttpProcessor的run方法。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public void run() {  // Process requests until we receive a shutdown signal  while (!stopped) {  // Wait for the next socket to be assigned  Socket socket = await();  if (socket == null)  continue;  // Process the request from this socket  try {  process(socket);  } catch (Throwable t) {  log("process.invoke", t);  }  // Finish up this request  connector.recycle(this);  }  } |

我们发现他是调用await方法来阻塞等待获得socket方法。而之前Connector是调用assign分配的，这是什么原因？  
下面仔细看await和assign方法。这两个方法协同合作，当assign获取socket时会通知await然后返回socket。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | synchronized void assign(Socket socket) {  // Wait for the Processor to get the previous Socket  while (available) {  try {  wait();  } catch (InterruptedException e) {  }  }  // Store the newly available Socket and notify our thread  this.socket = socket;  available = true;  notifyAll();  }  private synchronized Socket await() {  // Wait for the Connector to provide a new Socket  while (!available) {  try {  wait();  } catch (InterruptedException e) {  }  }  // Notify the Connector that we have received this Socket  Socket socket = this.socket;  available = false;  notifyAll();  return (socket);  } |

默认available为false。

接下来就是剩下的事情就是解析请求，填充HttpRequest和HttpResponse对象，然后交给container负责。  
这里我不过多赘述如何解析

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | private void process(Socket socket) {  //parse  ....  connector.getContainer().invoke(request, response);  ....  } |

## **Container**

A Container is an object that can execute requests received from a client, and return responses based on those requests

Container是一个接口，实现了这个接口的类的实例，可以处理接收的请求，调用对应的Servlet。

总共有四类Container，这四个Container之间并不是平行关系，而是父子关系

* Engine - 最顶层的容器，可以包含多个Host
* Host - 代表一个虚拟主机，可以包含多个Context
* Context - 代表一个web应用，也就是ServletContext，可以包含多个Wrappers
* Wrapper - 代表一个Servlet,不能包含别的容器了，这是最底层

### **Container的调用**

容器好比是一个加工厂，加工接受的request，加工方式和流水线也很像，但又有点区别。这里会用到一个叫做Pipeline的 东西，中文翻译为管道，request就放在管道里顺序加工，进行加工的工具叫做Valve，好比手术刀，Pipeline可添加多个Valve,最后加工的工具称为BaseValve

上面可能讲的比较抽象，接下来我们来看代码。Engine是顶层容器，所以上面invoke，执行的就是Engine的方法。StandardEngine是Engine的默认实现，注意它也同时实现了Pipeline接口，且包含了Pipeline。

它的构造方法同时指定了baseValve,也就是管道最后一个调用的Valve

|  |  |
| --- | --- |
| 1  2  3  4 | public StandardEngine() {  super();  pipeline.setBasic(new StandardEngineValve());  } |

好，接着我们看invoke,这个方法是继承自ContainerBase。只有一行，之间交给pipeline，进行加工。

|  |  |
| --- | --- |
| 1  2  3  4 | public void invoke(Request request, Response response)  throws IOException, ServletException {  pipeline.invoke(request, response);  } |

下面是StandardPipeline的invoke实现，也就是默认的pipeline实现。

|  |  |
| --- | --- |
| 1  2  3  4  5 | public void invoke(Request request, Response response)  throws IOException, ServletException {  // Invoke the first Valve in this pipeline for this request  (new StandardPipelineValveContext()).invokeNext(request, response);  } |

也只有一行！调用StandardPipelineValveContext的invokeNext方法，这是一个pipeline的内部类。让我们来看  
具体代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public void invokeNext(Request request, Response response)  throws IOException, ServletException {  int subscript = stage;  stage = stage + 1;  // Invoke the requested Valve for the current request thread  if (subscript < valves.length) {  valves[subscript].invoke(request, response, this); //加工  } else if ((subscript == valves.length) && (basic != null)) {  basic.invoke(request, response, this);  } else {  throw new ServletException  (sm.getString("standardPipeline.noValve"));  }  } |

它调用了pipeline所用的Valve来对request做加工，当Valve执行完，会调用BaseValve,也就是上面的StandardEngineValve，  
我们再来看看它的invoke方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | // Select the Host to be used for this Request  StandardEngine engine = (StandardEngine) getContainer();  Host host = (Host) engine.map(request, true);  if (host == null) {  ((HttpServletResponse) response.getResponse()).sendError  (HttpServletResponse.SC\_BAD\_REQUEST,  sm.getString("standardEngine.noHost",  request.getRequest().getServerName()));  return;  }  // Ask this Host to process this request  host.invoke(request, response); |

它通过(Host) engine.map(request, true);获取所对应的Host,然后进入到下一层容器中继续执行。后面的执行顺序  
和Engine相同，我不过多赘述

#### **执行顺序小结**

经过一长串的invoke终于讲完了第一层容器的执行顺序。估计你们看的有点晕，我这里小结一下。

Connector -> HttpProcessor.process() -> StandardEngine.invoke() -> StandardPipeline.invoke() ->  
StandardPipelineValveContext.invokeNext() -> valves.invoke() -> StandardEngineValve.invoke() ->  
StandardHost.invoke()

到这里位置Engine这一层结束。接下来进行Host，步骤完全一致

StandardHost.invoke() -> StandardPipeline.invoke() ->  
StandardPipelineValveContext.invokeNext() -> valves.invoke() -> StandardHostValve.invoke() ->  
StandardContext.invoke()

然后再进行Context这一层的处理，到最后选择对应的Wrapping执行。

## **Wrapper**

Wrapper相当于一个Servlet实例，StandardContext会更根据的request来选择对应的Wrapper调用。我们直接来看看  
Wrapper的basevalve是如果调用Servlet的service方法的。下面是StandardWrapperValve的invoke方法，我省略了很多，  
只看关键。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public void invoke(Request request, Response response,  ValveContext valveContext)  throws IOException, ServletException {  // Allocate a servlet instance to process this request  if (!unavailable) {  servlet = wrapper.allocate();  }  // Create the filter chain for this request  ApplicationFilterChain filterChain =  createFilterChain(request, servlet);  // Call the filter chain for this request  // NOTE: This also calls the servlet's service() method  String jspFile = wrapper.getJspFile(); //是否是jsp  if (jspFile != null)  sreq.setAttribute(Globals.JSP\_FILE\_ATTR, jspFile);  else  sreq.removeAttribute(Globals.JSP\_FILE\_ATTR);  if ((servlet != null) && (filterChain != null)) {  filterChain.doFilter(sreq, sres);  }  sreq.removeAttribute(Globals.JSP\_FILE\_ATTR);  } |

首先调用wrapper.allocate(),这个方法很关键，它会通过反射找到对应servlet的class文件，构造出实例返回给我们。然后创建一个FilterChain，熟悉j2ee的各位应该对这个不陌生把？这就是我们在开发web app时使用的filter。然后就执行doFilter方法了，它又会调用internalDoFilter，我们来看这个方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | private void internalDoFilter(ServletRequest request, ServletResponse response)  throws IOException, ServletException {  // Call the next filter if there is one  if (this.iterator.hasNext()) {  ApplicationFilterConfig filterConfig =  (ApplicationFilterConfig) iterator.next();  Filter filter = null;    filter = filterConfig.getFilter();  filter.doFilter(request, response, this);  return;  }  // We fell off the end of the chain -- call the servlet instance  if ((request instanceof HttpServletRequest) &&  (response instanceof HttpServletResponse)) {  servlet.service((HttpServletRequest) request,  (HttpServletResponse) response);  } else {  servlet.service(request, response);  }  } |

终于，在这个方法里看到了service方法，现在你知道在使用filter的时候如果不执行doFilter，service就不会执行的原因了把。

## **小结**

Tomcat的重要过程应该都在这里了，还值得一提的是LifeCycle接口，这里所有类几乎都实现了LifeCycle，Tomcat通过它来统一管理容器的生命流程，大量运用观察者模式。有兴趣的同学可以自己看书

## 

## 

## **https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/index.html**

## 

## **Tomcat 总体结构**

Tomcat 的结构很复杂，但是 Tomcat 也非常的模块化，找到了 Tomcat 最核心的模块，您就抓住了 Tomcat 的“七寸”。下面是 Tomcat 的总体结构图：

##### **图 1.Tomcat 的总体结构**



从上图中可以看出 Tomcat 的心脏是两个组件：Connector 和 Container，关于这两个组件将在后面详细介绍。Connector 组件是可以被替换，这样可以提供给服务器设计者更多的选择，因为这个组件是如此重要，不仅跟服务器的设计的本身，而且和不同的应用场景也十分相关，所以一个 Container 可以选择对应多个 Connector。多个 Connector 和一个 Container 就形成了一个 Service，Service 的概念大家都很熟悉了，有了 Service 就可以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生命、掌握其生死大权，那就非 Server 莫属了。所以整个 Tomcat 的生命周期由 Server 控制。

### **以 Service 作为“婚姻”**

我们将 Tomcat 中 Connector、Container 作为一个整体比作一对情侣的话，Connector 主要负责对外交流，可以比作为 Boy，Container 主要处理 Connector 接受的请求，主要是处理内部事务，可以比作为 Girl。那么这个 Service 就是连接这对男女的结婚证了。是 Service 将它们连接在一起，共同组成一个家庭。当然要组成一个家庭还要很多其它的元素。

说白了，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 Container 容器。

### **以 Server 为“居”**

前面说一对情侣因为 Service 而成为一对夫妻，有了能够组成一个家庭的基本条件，但是它们还要有个实体的家，这是它们在社会上生存之本，有了家它们就可以安心的为人民服务了，一起为社会创造财富。

Server 要完成的任务很简单，就是要能够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，包括如何初始化、如何结束服务、如何找到别人要访问的 Service。还有其它的一些次要的任务，如您住在这个地方要向当地政府去登记啊、可能还有要配合当地公安机关日常的安全检查什么的。

## **Servlet 容器“Container”**

Container 是容器的父接口，所有子容器都必须实现这个接口，Container 容器的设计用的是典型的责任链的设计模式，它有四个子容器组件构成，分别是：Engine、Host、Context、Wrapper，这四个组件不是平行的，而是父子关系，Engine 包含 Host,Host 包含 Context，Context 包含 Wrapper。通常一个 Servlet class 对应一个 Wrapper，如果有多个 Servlet 就可以定义多个 Wrapper，如果有多个 Wrapper 就要定义一个更高的 Container 了，如 Context，Context 通常就是对应下面这个配置：

##### **清单 10. Server.xml**

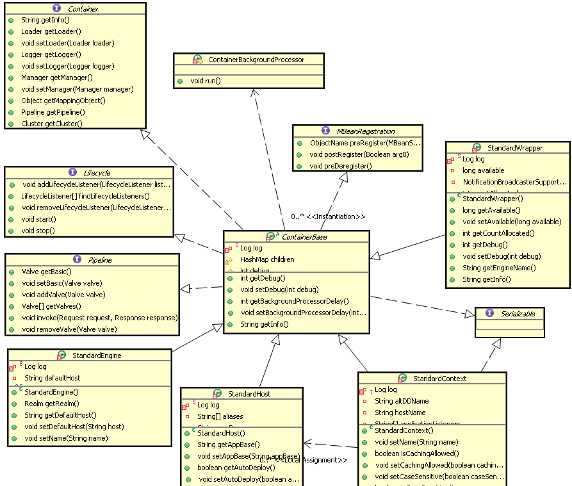
|  |  |
| --- | --- |
| 1  2  3  4  5 | <Context      path="/library"      docBase="D:\projects\library\deploy\target\library.war"      reloadable="true"  /> |

### **容器的总体设计**

Context 还可以定义在父容器 Host 中，Host 不是必须的，但是要运行 war 程序，就必须要 Host，因为 war 中必有 web.xml 文件，这个文件的解析就需要 Host 了，如果要有多个 Host 就要定义一个 top 容器 Engine 了。而 Engine 没有父容器了，一个 Engine 代表一个完整的 Servlet 引擎。

那么这些容器是如何协同工作的呢？先看一下它们之间的关系图：

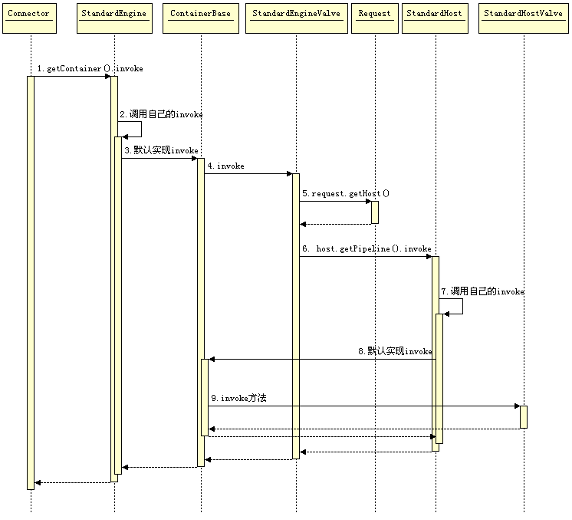
##### **图 8. 四个容器的关系图**



（[查看清晰大图](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/origin_image008.png" \t "https://www.cnblogs.com/diegodu/p/_blank)）

当 Connector 接受到一个连接请求时，将请求交给 Container，Container 是如何处理这个请求的？这四个组件是怎么分工的，怎么把请求传给特定的子容器的呢？又是如何将最终的请求交给 Servlet 处理。下面是这个过程的时序图：

##### **图 9. Engine 和 Host 处理请求的时序图**



（[查看清晰大图](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/origin_image009.png" \t "https://www.cnblogs.com/diegodu/p/_blank)）

这里看到了 Valve 是不是很熟悉，没错 Valve 的设计在其他框架中也有用的，同样 Pipeline 的原理也基本是相似的，它是一个管道，Engine 和 Host 都会执行这个 Pipeline，您可以在这个管道上增加任意的 Valve，Tomcat 会挨个执行这些 Valve，而且四个组件都会有自己的一套 Valve 集合。您怎么才能定义自己的 Valve 呢？在 server.xml 文件中可以添加，如给 Engine 和 Host 增加一个 Valve 如下：

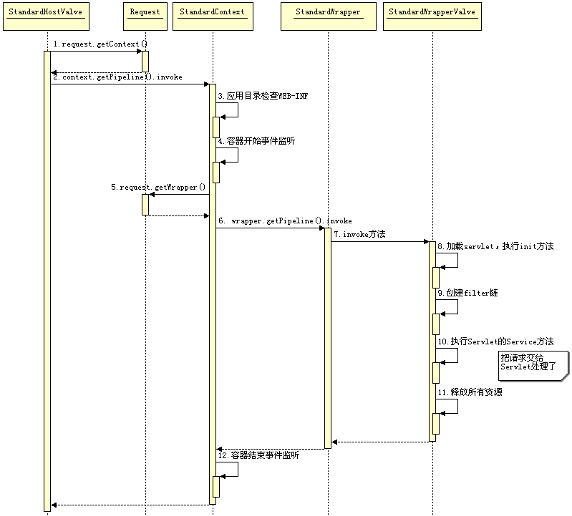
##### **清单 11. Server.xml**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <Engine defaultHost="localhost" name="Catalina">        <Valve className="org.apache.catalina.valves.RequestDumperValve"/>      ………      <Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true"          xmlNamespaceAware="false" xmlValidation="false">            <Valve className="org.apache.catalina.valves.FastCommonAccessLogValve"              directory="logs"  prefix="localhost\_access\_log." suffix=".txt"              pattern="common" resolveHosts="false"/>      …………      </Host>  </Engine> |

StandardEngineValve 和 StandardHostValve 是 Engine 和 Host 的默认的 Valve，它们是最后一个 Valve 负责将请求传给它们的子容器，以继续往下执行。

前面是 Engine 和 Host 容器的请求过程，下面看 Context 和 Wrapper 容器时如何处理请求的。下面是处理请求的时序图：

##### **图 10. Context 和 wrapper 的处理请求时序图**



（[查看清晰大图](https://www.ibm.com/developerworks/cn/java/j-lo-tomcat1/origin_image010.png" \t "https://www.cnblogs.com/diegodu/p/_blank)）

从 Tomcat5 开始，子容器的路由放在了 request 中，request 中保存了当前请求正在处理的 Host、Context 和 wrapper。