Tomcat

Tomcat Wrapper 组件

3
1 年前
由 **diecui1202** 发布 在 **Tomcat**

一些基本概念

1、ServletContext：

作用：表示一个 web 应用的上下文；可以想象成一个 Web 应用程序的共享数据区域，该区域保存该 Web 应用程序的共享数据；

生命周期：每个 Web 应用程序都对应一个 ServletContext，保存在 Context 中，在 Context 初始化时创建，Context 撤销时销毁；

2、servlet-mapping：

作用：按 url 将请求匹配到 servlet；

匹配过程：1) url 按最长 context path 匹配当前 Host 下的 Context；

2) 余下的将用于匹配该 Context 下的 Servlet；

匹配规则，顺序如下：

1) 精确匹配：如/user/login；必须以/开头，不能以/*结尾；

2) 通配符匹配：如/customers/*，以最长串优先依次匹配；

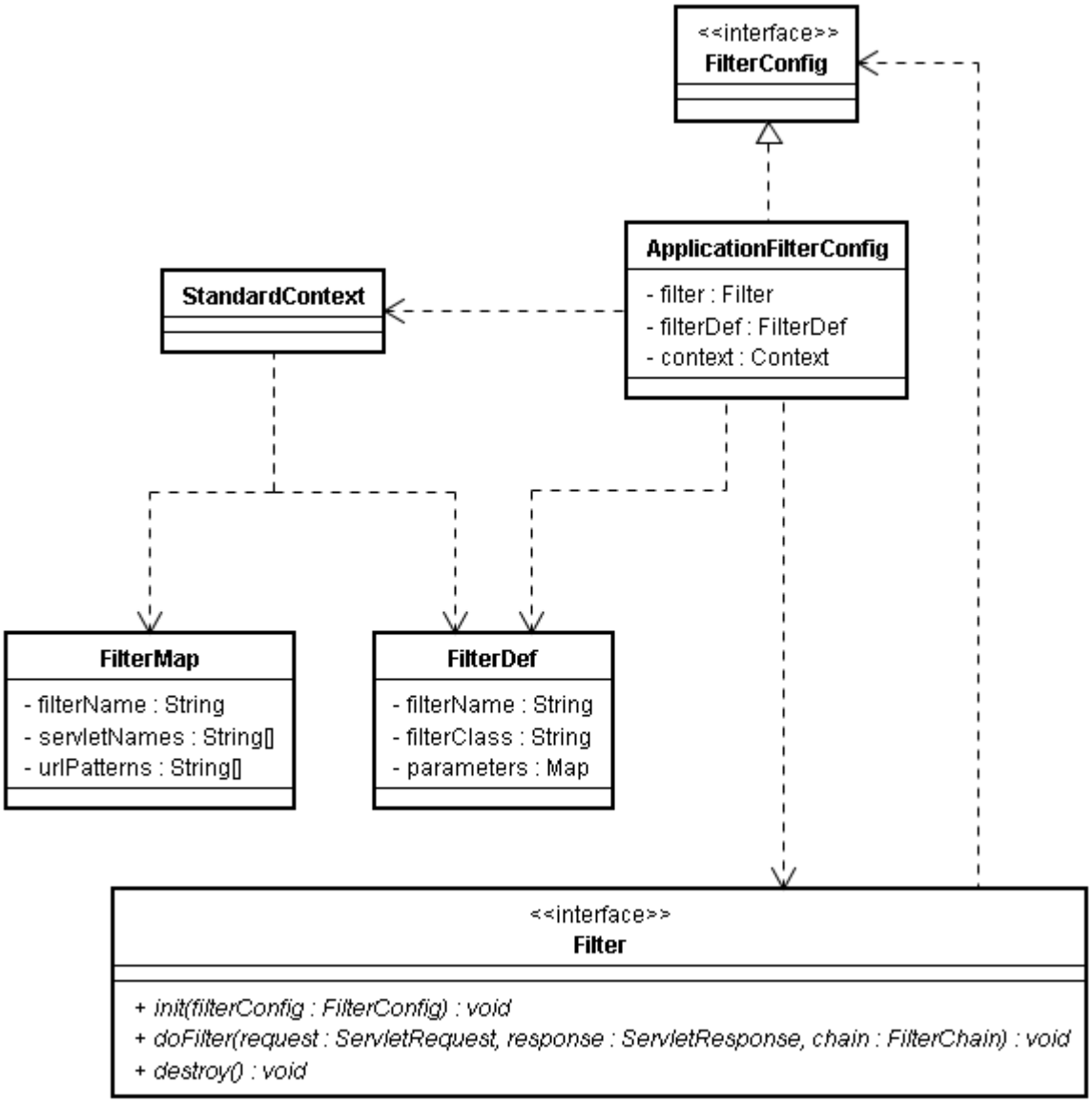
3) 扩展名匹配：如/user/register.jsp；任何以*开头，以 jsp 结尾的 mapping 都会匹配；

4) 默认匹配：上述 3 种匹配失败后，将匹配到该 mapping 上；

3、Filter：

作用：Filter 使用户可以改变一个 request 和修改一个 response，它不是一个 servlet，也不能产生 response，它能够在一个 request 到达 servlet 之前预处理 request，也可以在 response 离开 servlet 时处理 response。

类图：如下



其中：

1) FilterDef 用于保存从 web.xml 解析出来的 filter 定义，包含 filterName, filterClass, init-param 等；

2) FilterMap 用于保存从 web.xml 解析出来的 filter-mapping 配置；

3) 当初始化完所有的 FilterDef 以及 FilterMap 后，容器会初始化 ApplicationFilterConfig，用于 Filter 内部访问 Tomcat 容器，如 Filter 的 init-param, ServletContext 等；

4) 解析出来的 FilterDef、FilterMap 以及 ApplicationFilterConfig 均保存在 StandardContext 中；

4、Listener：

作用：通过在 web.xml 中配置<listener>元素，可以在适当的时候收到 Tomcat 容器的事件通知，从而可以做出相应的响应；

类型：

1) 事件类型：当某事件发生时触发；

a) ServletContextAttributeListener：当 ServletContext 的属性发生变化（新增、修改、删除）时触发；

b) ServletRequestAttributeListener：当 request 的属性发生变化（新增、修改、删除）时响应；

c) ServletRequestListener：当请求在经 Filter 链处理前后时响应；

d) HttpSessionAttributeListener：当 session 中的属性发生变化（新增、修改、删除）时响应；

2) 生命周期类型：在生命周期中的某个点触发；

a) ServletContextListener：在 ServletContext 初始化完成和销毁时触发；

b) HttpSessionListener：在 session 创建和销毁时触发；

重要的数据结构

1、Mapper：

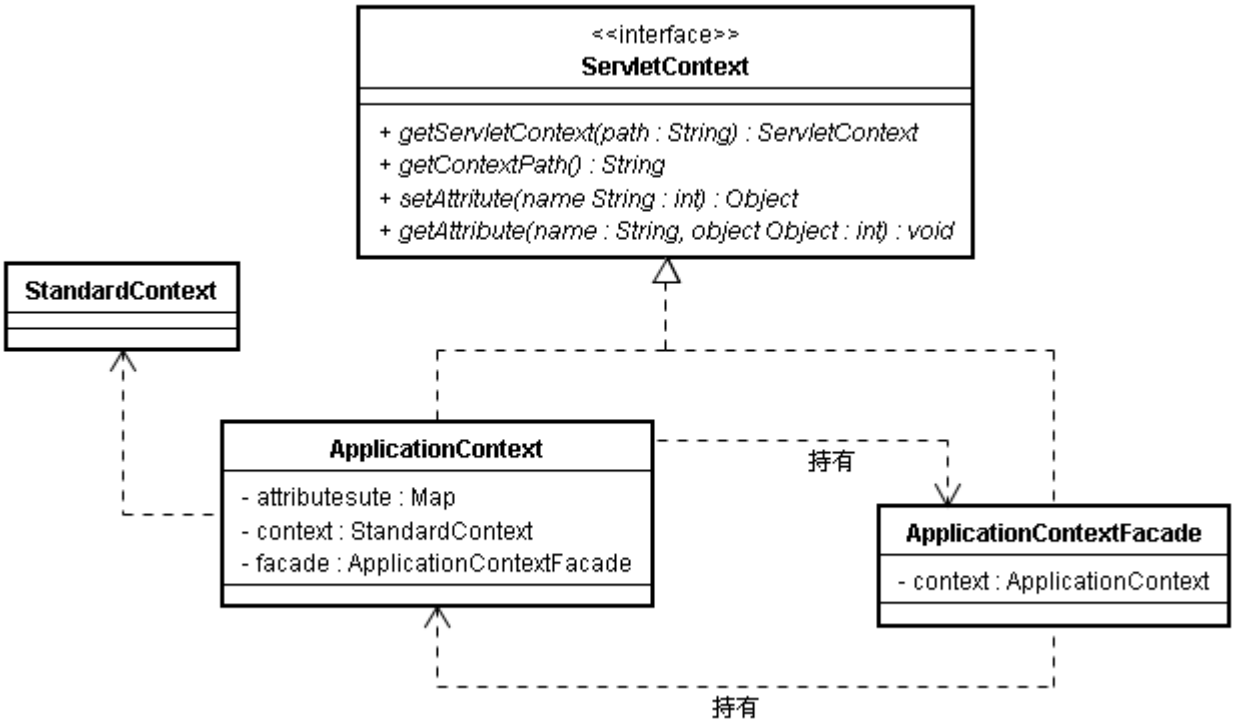
保存 Tomcat 容器中所有 Host, Context, Wrapper 及 servlet-mapping 的映射关系；

2、MappingData：

当前请求经过 Mapper 后，即决定将该请求交给哪个 Host, 哪个 Context, 哪个 Wrapper 来处理；这些数据组成 MappingData；

创建 ServletContext

1、ServletContext 类图



这里采用了门面模式，ApplicationContext 仅供 tomcat 内部使用，在 Servlet 中取得的 servletContext 实际上是 ApplicationContextFacade 的实例，防止外部修改 tomcat 容器中重要的数据结构；

2、实例化 ServletContext

实例化 ServletContext 是通过 StandardContext.getServletContext()方法来完成的：

```
1   public ServletContext getServletContext() {
2       if (context == null) {
3           context = new ApplicationContext(getBasePath(), this);
4           if (altDDName != null)
5               context.setAttribute(Globals.ALT_DD_ATTR,altDDName);
6       }
7       return (context.getFacade());
```

```
8    }
```

解析部署描述符 web.xml

1、初始化 Wrapper

当 StandardContext 初始化时，会解析 web.xml 文件（参考 WebRuleSet）；当解析到 web-app/servlet 标签时会调用 StandardContext.createWrapper()方法，从而构造 Wrapper 组件，并将其作为子节点添加到当前 Context 下；

初始化 Wrapper 后，会设置与该 servlet 相关的配置，如 servlet-name, servlet-class, init-param, jsp-file, load-on-startup 等；这些配置值都是刚才构建出来的 Wrapper 组件的属性；

2、初始化 servlet-mapping

由于 servlet-mapping 是针对 Context 的，处理 servlet-mapping 的工作由 Context 来完成，主要工作如下：

1) 根据 servert-name 找到该 Context 的子节点 Wrapper，然后将该 mapping 添加到 Wrapper 中；

2) 将该 Wrapper 添加到当前 Context 的 Mapper 中，供 ServletContext 使用；

3、初始化 filter 及 filter-mapping

1) 解析 filter：调用 StandardContext.addFilterDef(FilterDef filterDef)，如下：

```
1    public void addFilterDef(FilterDef filterDef) {
2        synchronized (filterDefs) {
3            filterDefs.put(filterDef.getFilterName(), filterDef);
4        }
5        fireContainerEvent("addFilterDef", filterDef);
6    }
```

即：将 filterDef 以 filterName 为 key 保存在 Map 中；

2) 解析 filter-mapping：调用 StandardContext.addFilterMap(FilterMap filterMap)，如下：

```
1    public void addFilterMap(FilterMap filterMap) {
2        // 先会做一些校验
3        synchronized (filterMaps) {
4            FilterMap results[] =new FilterMap[filterMaps.length + 1];
5            System.arraycopy(filterMaps, 0, results, 0, filterMaps.length);
6            results[filterMaps.length] = filterMap;
7            filterMaps = results;
8        }
9        fireContainerEvent("addFilterMap", filterMap);
10   }
```

3) 初始化 ApplicationFilterConfig：当 StandardContext 启动时，会通过 filterStart()方法来初始化 ApplicationFilterConfig：将 1)中保存的每个 filterDef 和当前 StandardContext 包装成一个 ApplicationFilterConfig，供后面的创建 filter 链时使用；

4、初始化 listener

当解析到 web.xml 中的<listener>元素时，会调用 Standard.addApplicationListener(String listener)将 listener 的 class 信息保存起来，然后在 StandardContext 启动时，通过 listenerStart()方法来初始化 listener：实例化先前保存起来的 listener，并按事件/生命周期分类保存其实例；此时 StandardContext 初始化也快完成了，因此会触发 ServletContextListener.contextInitialized()方法；

Servlet 的 loadOnStartup

在配置 Servlet 时，可以指定其 loadOnStartup 属性，该属性值决定了 Servlet 在容器启动后是否加载及加载顺序，默认为-1，表示不加载；

StandardContext 在加载完所有 Servlet 配置后，会检查当前 Context 下的所有 Wrapper 的

loadOnStartup 属性值，当值大于 0 时，通过 TreeMap 按 loadOnStartup 从小到大排序后，分别调用每个 Wrapper.load()方法来完成 Servlet 的初始化：

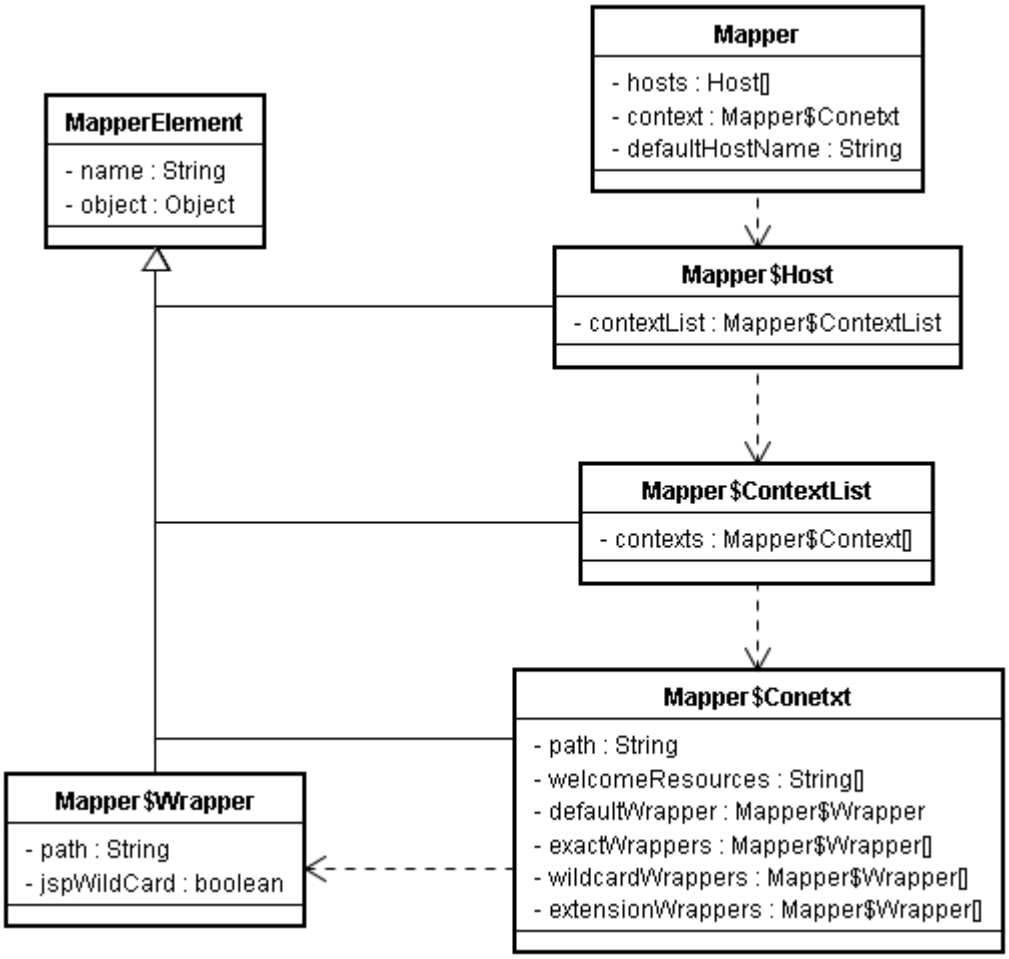
1) 根据 servletClass 是否是 Tomcat 内部 servlet，决定是用 webappClassloader 还是用 serverClassloader 来装载 servletClass；

2) 实例化 servletClass；

3) 如果 servlet 是单线程（即一个 servlet 实例同时只能处理一个请求），则初始化一个 servlet pool，用于存放空闲的 servlet 实例；后面在处理请求时分配 servlet 则会用到该 servlet pool；

Mapper 相关

1、先看看 Mapper 相关的类图



请不要把上图中的 Host, Context, Wrapper 和 Tomcat 中的 Host 组件、Context 组件、Wrapper 组件相混淆了，它们仅仅代表这三个组件的映射关系，真正所对应的组件实例引用存放 在 MapperElement 抽象类中的 object 属性上；

2、初始化 Mapper 结构

我们再来回顾一下 Tomcat 容器的初始化过程：

1) 解析 server.xml（由 Catalina 类来完成），同时调用 StandardServer.start()方法来初始化 Server 组件；

2) Server 组件初始化时，调用 StandardService.start()方法来初始化 Service 组件；

3) 在 StandardService.start()中，先完成所有内嵌组件的初始化（Engine, Host, Context, Wrapper 等），然后调用 Connect.start()方法来初始化 Connector 组件；

4) 由于在 Connector 初始化时，所有的内嵌组件都已初始化完毕，此时便可以完成 Mapper 的构造了；

初始化 Mapper 主要是由 MapperListener 来完成（在 Connector.start()中调用 MapperListener.init()方法）：

1) 注册 Engine：找到 defaultHost，并设置 Mapper 的 defaultHostName 属性；

2) 注册所有的 Host：针对每个 Host 组件，都构造一个 Mapper$Host 实例，name 属性为该 Host 组件的 name，object 属性即该 Host 组件；另外，针对 Host 组件的别名，也会产生新的 Mapper$Host 实例，不过其 object 和 Mapper$contextList 的引用相同；

3) 注册所有的 Context：针对每个 Context 组件，根据其 hostName，将其加到对应 Mapper$Host 实例的 Mapper$contextList 中去；

4) 注册所有的 Wrapper：和注册 Context 类似，将每个 Wrapper 组件根据其 servlet-mapping 保存到 Mapper$Context 的相应属性上；

3、MappingData 的构造

当请求到达后，在交给 Engine 处理前，会构造 MappingData（见 CoyoteAdapter.postParseRequest()方法）：

1) 根据请求的 host 属性，从 Mapper 中找到相应的 Host 组件及 Mapper$Context 数组；此时如果没有指定 host 属性的 Host 组件，则取 defaultHostName 所对应的 Host 组件；

2) 根据请求的 uri，找到 Context 组件；

3) 根据请求的 uri，按 servlet-mapping 规则，依次查找，直接找到对应的 Wrapper 组件；此时如果找不到，则依次查找 welcomeFile 所对应 Wrapper 组件；

4) 如果最后还是没有找到，则返回 defaultWrapper 组件；

5) 将匹配到的 Host, Context, Wrapper 保存在 request 中；

Wrapper 组件对请求的处理

当 MappingData 构造完成后，CoyoteAdapter 便将请求交给 Engine、Host、Context、Wrapper 组件进行处理；在 Context 组件中，StandardContextValve 直接从 request 中取出 Wrapper，然后交给它处理；下面看看 Wrapper 组件是如何处理的？

1、StandardWrapperValve

Wrapper 组件对请求的处理，是通过 StandardWrapperValve 来完成的：

1) 从 Wrapper 中分配一个 servlet 实例；

2) 根据当前 servlet 和 requestPath 将匹配到的 filter 加到 filter 链中；

3) 调用 filter chain 处理请求和响应；当所有 filter 执行完，最后才执行 servlet.service()方法；

Tomcat Context 组件

0
1 年前
由 **diecui1202** 发布 在 **Tomcat**

Context 代表一个 Web 应用，它运行在某个指定的虚拟主机（Host）上；每个 Web 应用都是一个 WAR 文件，或是一个包含 WAR 解压后的文件的目录；

Connector 组件接收到 http 请求后，通过将请求 URI 的最长可能前缀与每个 Context 的 path 进行匹配，然后选择相应的 Web 应用来处理这个 http 请求。

之后，Context 会根据 web application deployment descriptor 文件中定义的 servlet 映射，会选择一个正确的 Servlet 来处理请求。

Servlet 映射必须定义在该 Web 应用目录层次结构中的/WEB-INF/web.xml 中。

一、几个重要的概念

1、context frgament file

Context 片断文件，即描述 Context 配置的 xml 文件；它有三个层级：

a) Engine Scoped：适用于 Engine 下的所有 Web 应用程序，位于 $CATALINA_BASE/conf/context.xml；

b) Host Scoped：适用于指定的 Host 组件下的应用程序，位于 $CATALINA_BASE/[Engine]/[Host]/context.xml.default；

c) Web Application Scoped：代表一个独立的应用程序，它可以位于 $CATALINA_BASE/[Engine]/[Host]/[ContextPath].xml 或应用程序中的 META-INF/context.xml；

对于应用程序中内嵌的 context.xml 文件，有两个局限性：

1) 当 server.xml 文件中的 Host 节点指定 deployXML 属性为 false 时，则会忽略 Web 应用程序内嵌的 context.xml；

2) 内嵌的 context.xml 不能指定 context path；

2、appBase、docBase&context path

a) appBase：Host 组件中需要部署的应用的目录，如 webapps；也可以指定绝对路径，将 Web 应用程序放在 tomcat 之外；

b) docBase：应用程序资源所有的路径；应用程序的资源包括 HTML、CSS、JS、jar、class 文件等，其中静态资源直接放在该 docBase 或其子目录下，jar 文件放在 WEB-INF/lib 目录下，class 文件则放在 WEB-INF/classes 目录下；

c) context path：唯一代表一个应用；

二、如何配置 Context？

配置 context 有多种方式，如下：

1、将应用文件夹或 war 文件直接 copy 到 tomcat 的 webapps 目录下，这样 tomcat 启动的时候会将 webapps 目录下的文件夹或 war 文件的内容当成应用部署。这种方式最简单且无须书写任何配置文件。

2、在 tomcat 的 server.xml 配置文件中的 Host 节点下增加 Context 子节点，如：

```
1   <Context path="/test" docBase="D:\private\tomcat\test.war" />
```

其中，path 即 context path；docBase 指向应用所在的文件夹或 war 文件，可以是绝对路径，也可以是相对路径（相对该 Context 所在的 Host 的 appBase 属性值）；

3、在 tomcat 的 conf/[Engine]/[Host]目录下新建 xml 文件，文件名为 context path，内容如下：

```
1  <Context docBase="D:\private\tomcat\test.war"
2   privileged="true" antiResourceLocking="false" antiJARLocking="false">
3   <!-- Link to the user database we will get roles from -->
```

```
4    <ResourceLink name="users" global="UserDatabase"
5    type="org.apache.catalina.UserDatabase"/>
6   </Context>
```

其中，docBase 与第二种方式中的含义一样；

当 Host 的 autoDeploy 属性值为 true 时，以上三种配置 Context 的方式中，只有第 1、3 两种方式配置署的应用不需要重启 tomcat 即可完成部署；第二种方式需要重启 tomcat；另外，第 1 种方式不能指定特定的 context path；

三、启动 Context

在《Tomcat Host 组件》这篇文档中提到，应用的部署是由 HostConfig 完成的；Host 组件在初始化过程中会扫描三种不同类型的应用：context.xml 描述文件、war 包、文件夹；对于直接通过修改 server.xml 文件来配置的应用（上一节中的第 2 种场景），则是在启动 tomcat 解析 server.xml 文件时根据 Context 子节点来构造相应的应用；

不管是哪种方式配置 Context（context.xml 描述文件、war 包、文件夹、配置 server.xml 的 Context 节点），最终都是构造出一个 StandardContext 实例，并将其作为子容器添加到 Host 组件中；与 Host 组件类似，在构造 StandardContext 实例时，会为每个 StandardContext 对象构造一个 ContextConfig 实例，该 ContextConfig 实例实现了 LifecycleListener 接口，它会监听 StandardContext 实例的各个生命周期事件，并针对不同的事件做出不同的响应；

当 StandardContext 实例以子容器被添加到 Host 组件中时，会触发 StandardContext 实例的 start() 方法，至此 Context 工作由此展开：

1、Context 初始化和启动

Context 初始化时，主要会触发 init 事件，并通过 ContextConfig 来完成初始化工作；

Context 启动时主要完成如下事情：

a) 触发 before_start 事件来处理文件锁的问题（见下一小节 before_start 事件）；

b) 根据 docBase 是 war 包还是文件夹，初始化应用程序资源的访问对象；

c) 初始化 context 特定的 classloader：WebappLoader；

d) 计算 work directory 并初始化 ServletContext：tomcat 会为每个 Context 生成一个 work

directory，位于$CATALINA_BASE/work/[Engine]/[Host]目录下，work directory 目录名根据 context path 而来（将"/"转换为"_"）；work directory 目录用于存放由 jsp 生成的 servlet 的 class 文件；

e) 启动内嵌组件，如 Loader 等；

f) 触发 start 事件来解析 web 应用描述符文件：如 Servlet 配置、Filter 配置、Listener 配置、session-timeout、welcome-file 以及 servlet 参数等；

g) 启动 Listerer；

h) 根据 backgroundProcessorDelay 决定是否需要开启后台线程：该后台线程可以用于管理 session 超时、监听类的变化来决定是否需要重启 Context；当 Context 的 reloadable 属性为 true 时，如果应用程序的资源发生变化时，会通过 backgroundProcess()来完成应用的重启；

i) 根据 Servlet 配置的 loadOnStartup 参数及大小顺序决定是否需要初始化 Servlet 及其初始化顺序；

2、ContextConfig 事件处理

a) init 事件

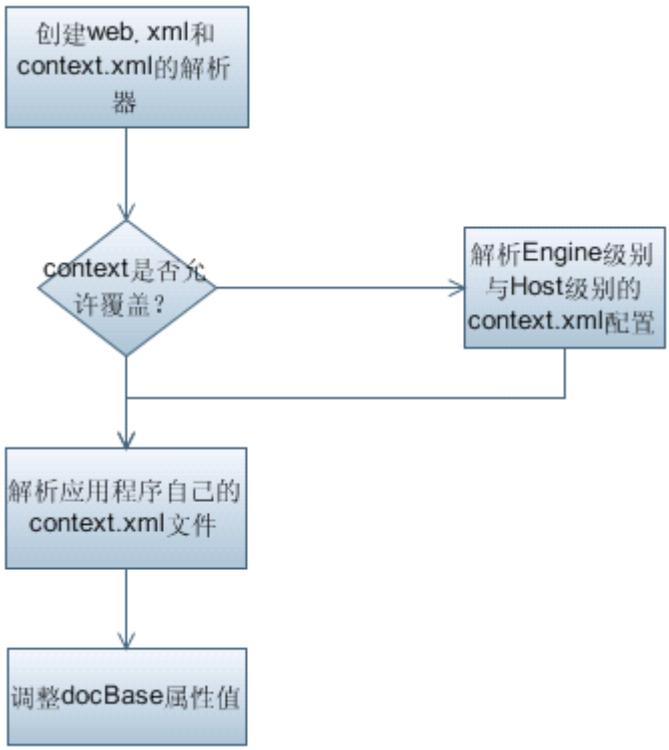
ContextConfig 在处理 init 事件时，主要工作是：

1) 实例化了两个 Digester 类，分别为 webDigester 和 contextDigester；webDigester 用于解析 web.xml 配置文件，contextDigester 用于解析前面提到的三个层级的 context fragment file；在解析 context fragment file 时，主要的工作是解析内嵌元素，如 Listener, Loader, Manager, Parameter, Resources, Valve 以及 WatchedResource，并将它们设置到该 Context 对应的属性上；；

2) 调整 docBase 属性值（即应用程序资源所在的位置）：

- 在配置 Context 时如果没有指定 docBase，则根据 context path 计算得出（"/"会替换成"#"，空会替换成 ROOT），并转化为绝对路径；
- 根据 Host 中配置的 unpackWARs 属性，决定是否将配置在 tomcat 之外的 war 包解压到 appBase 下，并以 context path 为目录名（"/"会替换成"#"）；此时 docBase 指向解压后的文件夹；如果的 unpackWARs 为 false，则不会自动解压，此时 docBase 不变；

用流程图来表述如下：



b) before_start 事件

在该过程中，主要是针对反文件锁做了一定的处理；在 windows 下，当 ClassLoader 加载类时，会打开相应的 jar 包，此时该 jar 包会被锁定，是不能被删除的；在 tomcat 中，当 Context 的 antiResourceLocking 设置为 true 时，如果删掉应用程序的 war 包时，对应的文件夹也会被删除，这是怎么做到的呢？

很简单，把应用程序的资源拷贝到一个临时的目录，然后将 docBase 指向该临时目录，这样锁定的都是副本；

c) start 事件

当 start 事件发生时，ContextConfig 会针对该 Context 完成如下工作：

1) 解析应用程序描述符：

a) $CATALINA_BASE/conf/web.xml：作用于所有的 Web 应用程序；

b) $CATALINA_BASE/conf/[Engine]/[Host]]/web.xml.default：作用于指定 Host 下的所有 Web 应用程序；

c) WEB-INF/web.xml：仅作用于当前 Web 应用程序；

2) 新版本中增加了对 Listener、Filter、Servlet 注解的支持；

3) 对 web.xml 里设置的一些 roles 进行一些简单的校验,，然后全部放到 context 持有的字符串

数组 securityRoles 中；（这块没有仔细研究过，熟悉的同学可以解释一下）；

4) 根据 web.xml 里配置的 login-config 等权限安全访问来确定是否为本 context 配置一个用于确定访问权限的 Valve；如果需要，则通过 addValve 添加到 pipeline 中；

d) destroy 事件

在 Context 需要 reload 时，会触发 destory 事件，此时则会将该 Context 的 work directory 目录删除；

e) stop 事件

此时会清除所有与 Context 相关的配置，如子容器、从 web.xml 读取的配置信息（如 Listener、Filter、Servlet 等）；

四、Context 对请求的处理

Context 在初始化时，会创建 StandardContextValve 的实例并加入 pipeline 中；Host 组件在接收到请求后，会调用以下方法将请求转交给 Context 来处理：

```
1    context.getPipeline().getFirst().invoke(request, response);
```

也就是会调用 StandardContextValve 的 invoke 方法，其具体流程如下：

1、先检查所请求的资源是否是保护资源，如果是，则直接返回客户端“资源不存在”（404）；

2、如果应用在 reload，则 sleep 1s；此处是通过忙等待实现；同时用新的 WebappLoader 替换当前的 ClassLoader；

3、如果找不到该请求对应的 Wrapper（Servlet），则同样返回 404；

4、触发 ServletRequestListener 的 requestInitialized()方法；

5、将请求交由 wrapper 进行处理；

```
1    wrapper.getPipeline().getFirst().invoke(request, response);
```

6、wrapper 处理完后，再触发发 ServletRequestListener 的 requestDestroyed()方法；

Context

Tomcat Host 组件

0

1 年前
由 khotyn 发布 在 Tomcat

Tomcat Host 组件在 Tomcat 中代表一个“Virtual Host”，使 Tomcat 可以在单个 Tomcat 实例中支持多个“Virtual Host”，这样，我们也就可以知道一个 Engine 可以包含多个 Host 组件。Host 组件包含两个主要的 Valve，一个 Valve 决定请求由哪一个 Context 处理，另一个 Valve 负责处理在 Context 中未被捕获的异常。除了两个 Valve 以外，Host 组件还包含了一个 Configurator，它的作用是负责应用的部署，加载等事情。

一：Virtual Host

在了解 Virtual Host 之前，我们先来了解下什么是 Host。我们知道，在我们访问一个网站时，我们需要在浏览器的地址栏输入一个网页地址，浏览器会试图将域名解析成 IP，这个 IP 代表了连接到互联网的一台主机(Host)。在浏览器向主机发送的 HTTP 请求中，也包含了请求的 Host 信息：

在最简单的情况下，一台主机只需要对应一个 IP，提供一个 web 服务即可，这种情况下一个 IP 就对应一台物理主机(Physical Host)。然而，在多数情况下，一台主机不会只提供一个 web 服务，因而一台物理主机就需要虚拟出多台主机来，这就是 Virtual Host。Virtual Host 根据实现技术的不同可以分为基于名称的 Virtual Host 和基于 IP 的 Virtual Host。

基于名称的 Virtual Host

基于名称的 Virtual Host，对于基于名称的 Virtual Host 来说，每一个 Virtual Host 对应一个域名，这些域名都解析到同一个 IP 下去，这样，这些 Virtual Host 就共享了这个 IP 对应的物理主机的资源，在 Tomcat 中，主要配置以下信息就配置了一个基于名称的 Virtual Host：

```
1    <Host name="localhosts"  appBase="webapps"
2        unpackWARs="true" autoDeploy="true"
3        xmlValidation="false" xmlNamespaceAware="false">
4    </Host>
```

当然，在 Tomcat 中，如果多个 Virtual Host 仅仅名称是不一样的，其他都是一样的，那么就可以使用别名来配置，如下图：

```
1    <Host name="localhosts"  appBase="webapps"
2            unpackWARs="true" autoDeploy="true"
3            xmlValidation="false" xmlNamespaceAware="false">
4        <Alias>khotyn.org</Alias>
5    </Host>
```

基于 IP 的 Virtual Host

不同于基于名称的 Virtual Host，在基于 IP 的 Virtual Host 中，可以将多个 IP 地址绑定到同一台物理主机上去，这个是怎么做到的呢？一个方式在物理主机上配置多块网卡，另一个就是通 过 Virtual Network Interfaces 来实现，在 Tomcat 中配置基于 IP 的 Virtual Host，可以参考下图中的配置：

```
1    <Connector port="8080"
     protocol="org.apache.coyote.http11.Http11NioProtocol"
2    connectionTimeout="20000"
```

```
            redirectPort="8443" executor="tomcatThreadPool"
    address="127.0.0.2" useIPVHosts="true"/>
```

注意，需要将 Connector 的 useIPVHosts 设置成 true，默认情况为 false，才能够使用基于 IP 的 Virtual Host。

二：StandardHost 和 HostConfig

StandardHost

Tomcat 对 Host 的默认实现是 StandardHost，StandardHost 是一个非常简单地类，其属性和方法基本上和 server.xml 中对应的 host 元素的属性一一对应，但是值得注意的是，StandardHost 在初始化的时候，会初始化一个 HostConfig，并把它注册成一个 Lifecycle Listener，用于负责应用的部署。StandardHost 另外一个特别的地方就是在初始化的时候不仅仅加入它的 Basic Valve: StandardHostValve 到 Pipeline 中，而且加入了一个 ErrorReportValve 到 Pipeline 中去。

HostConfig

在 Host 调用其 start 方法的时候，它也生成了一个 HostConfig 实例，并让它去监听 Host 的启动，关闭以及定时事件。

在 Host 中每一个应用由一个 Context 和一个 DeployedApplication 来表示，你可以在 server.xml 或者 CATALINA_BASE/conf/[engineName]/[hostName]下或者在应用的 META-INF 目录下配置 context。

前一种在解析 server.xml 的时候就会被解析成 Context，后两种会在 HostConfig 部署应用的时候被解析并生成对应的 Context。

DeployedApplication 是 HostConfig 的一个内部类，是用来监视应用的文件的修改的，它有两个重要的属性：

- redeployResources：这里面的文件被删除或者被修改会造成应用的重新部署。
- reloadResources：这里面的文件被删除或者被修改会造成应用的重新加载。具体哪些文件需要被监视可以在 Context 的 WatchedResource 里面配置。

在这里需要区分应用的重新部署和重新加载是不一样的行为：重新部署一个 Context 只需要调用 Context 的 stop 方法，然后在调用 start 方法就可以了，但是重新加载一个 Context 则需要将 Context 完全从 Host 中移除，然后重新生成一个 Context，并加入到 Host 中去。

HostConfig 的一个重要的作用就是负责应用的部署，在 HostConfig 中，"应用"被分成 3 种不用的类型：context.xml 描述文件，war 包，文件夹，其部署方式都是类似的，下面仅以 war 包的部署为例来讲一讲在 HostConfig 中大概的一个应用部署过程：

1. 判断是否已经部署了该 war，如果已经部署就直接返回。
2. 看下 war 包里面有没有 META-INF/context.xml 文件，有就将其复制一份到 CATALINA_BASE/conf/[engineName]/[hostName]目录下并解析成一个 Context
3. 生成一个 DeloyedApplication 实例并将需要监视的文件放入其中。
4. 设置 2 中解析到的 Context（如果 2 中没有 context.xml 文件，就实例化一个 Context）的一些属性，比如 DocBase 等。
5. 将 Context 加入到 host 中去。
6. 如果 war 包是需要被解开来被部署的，那么就需要更新 Context 的 Docbase 属性（具体的解包过程在步骤 5 中由 Context 的 start 方法去完成）。

三：StandardHostValve 和 ErrorReportValve

对于每一个组件，我们可以通过观察其包含的 Valve 的实现来了解其功能，在 Host 组件中，有两个重要的 Valve：StandardHostValve 和 ErrorReportValve；

StandardHostValve

StandardHostValve 的工作一个是决定请求由 Host 内的哪一个 Context 去处理，另一个处理应用没有捕获的异常，把处理未捕获异常的工作放在 Host 中是因为 Host 包含的子组件就是 Context 了，对应一个应用，理所当然，这个工作就放在了应用的上一层组件 Host 中处理。前一个决定哪一个 Context 来处理请求的代码相当简单，无非就是从 Request 中取出 Context，并交给它处理：

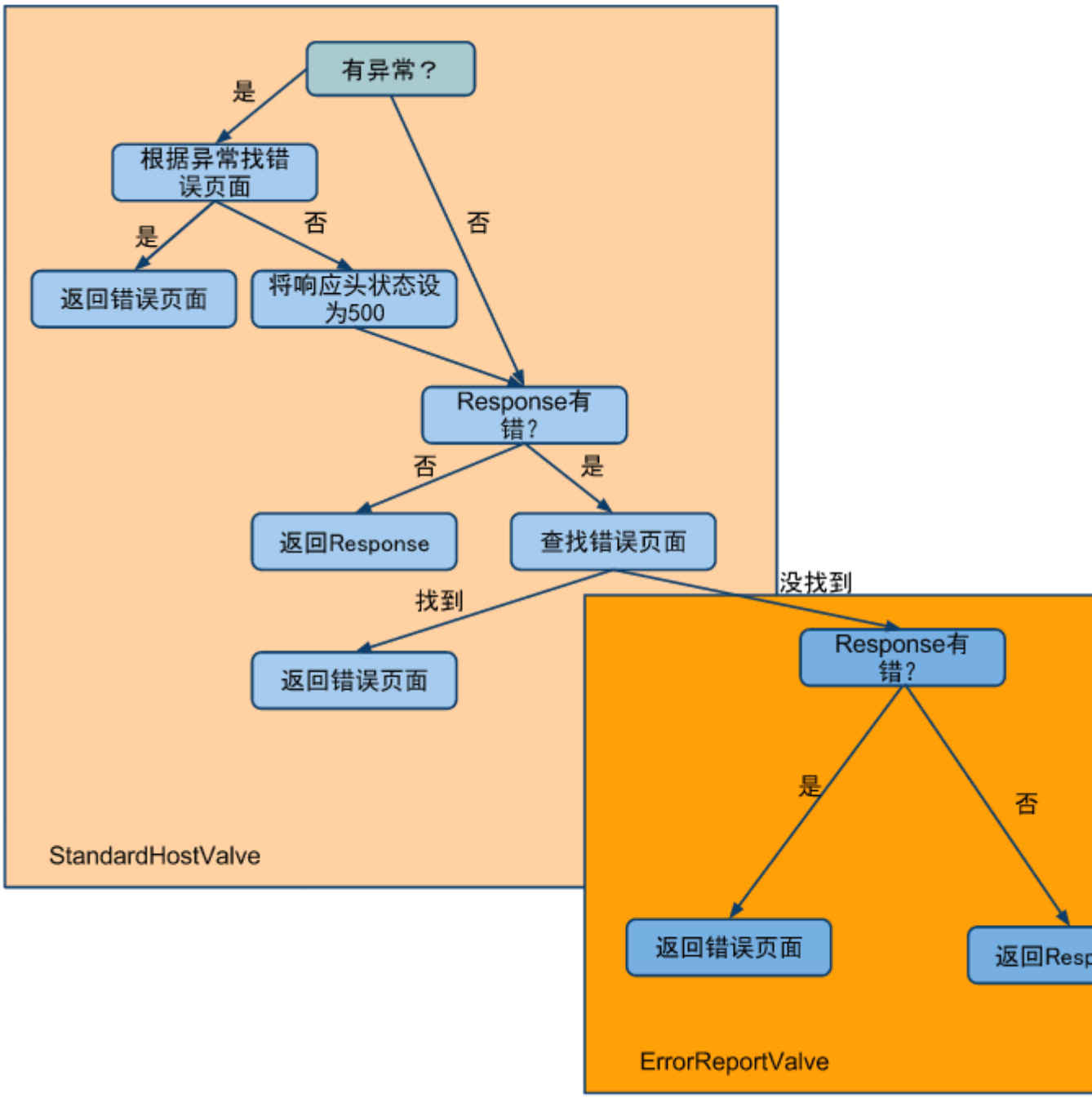
```
1    context.getPipeline().getFirst().invoke(request, response);
```

我们重点来看一下 StandardHostValve 对异常的处理，但是在这个之前，我们还是了解下 Servlet 规范(JSR154)对 error page 的描述，因为 StandardHostValve 对异常的处理就是按照 JSR154 来的：

> *为了让开发者能够在 servlet 产生错误的时候返回给客户端一些自定义的信息，我们可以在部署描述符(web.xml)中定义 error page。这个配置可以让容器在 servlet、filter 调用 response 的 sendError 方法或者 servlet 产生错误、异常传播到容器的时候，向客户端返回一个自定义的错误页面。*

也就是说 Servlet 容器需要能够根据 web.xml 中定义的 ErrorPage 的配置来处理由应用进入容器的错误。

下面通过一幅流程图来描述 StandardHostValve 对错误的处理过程：



ErrorReportValve

这个 Valve 的作用更像是给 StandardHostValve 擦屁股的，如果请求没有被提交，并且包含了 javax.servlet.error.exception，ErrorReportValve 就调用 response.sendError，然后产生一个默认的错误页面给客户端。

Host Tomcat

Tomcat Engine 组件

6
1 年前
由 khotyn 发布 在 Tomcat

Tomcat Engine 组件是一个职责相当简单的组件，他的主要作用就是决定从 Connecter 过来的请求应该交给哪一个 Host 来处理。在本文中，我们将会简单介绍下 Tomcat 的 Engine 组件的功能，backgroundProcess()方法，以及和 Engine 组件相关的几个 Valve。

一、Engine 组件的主要功能

在之前的学习中，我们已经了解到，Tomcat 的各个组件都是通过 Pipeline 连在一起的，而各个组件的功能都封装在了各个 Pipeline 的 BaseValve 中，对于 Engine 组件，它的 BaseValve 就是 StandardEngineValve，我们看一下它的 invoke 方法，便可以了解它的功能了：

```
public final void invoke(Request request, Response response
    throws IOException, ServletException {

    // Select the Host to be used for this Request
    Host host = request.getHost();
    if (host == null) {
        response.sendError
            (HttpServletResponse.SC_BAD_REQUEST,
            sm.getString("standardEngine.noHost",
                         request.getServerName()));
        return;
    }

    // Ask this Host to process this request
    host.getPipeline().getFirst().invoke(request, response);

}
```

从上面的代码片段我们看到，StandardEngineValve 只是从 request 中拿到 Host，然后调用 Host 组件的 Pipeline 就完事了，所以 Engine 组件的功能基本上就是决定请求由哪一个 Host 来处理，功能相当简单。

二、BackgroundProcessorDelay

Engine 组件的默认实现继承了 ContainerBase 类，这个类是所有的 Tomcat 组件的基类，它实现了 Container 接口，这个接口里面有一个很重要的方法：backgroundProcess，从它的名字来看就是做一个后台的处理。事实上，Tomcat 的组件可以和一个线程相关联，这个线程会定时地

去调 backgroundProcess 方法，这个方法可以用来实现应用的重新加载或者其他任何需要定时触发的功能。

在 ContainerBase 的 start()方法中，调用了一个 threadStart 方法，这个方法就是起一个线程去定时的调用 backgroundProcess 方法：

```
protected void threadStart() {

    if (thread != null)
        return;
    if (backgroundProcessorDelay <= 0)
        return;

    threadDone = false;
    String threadName = "ContainerBackgroundProcessor[" + toString() + "
    thread = new Thread(new ContainerBackgroundProcessor(), threadNa
    thread.setDaemon(true);
    thread.start();

}
```

至于 backgroundProcess 方法的调用间隔可以在 server.xml 中的各个组件配置中配置相应的 backgroundProcessDelay 来指定。

三、Engine 组件中其他一些可用 Valve

默认配置下，Engine 组件只包含了一个 BaseValve，也就是 StandardEngineValve，当然，你也可以根据自己的需要来增加 Valve，Tomcat 本身已经提供了三个 Valve 供 Engine 组件使用，你可以在 server.xml 中配置它们来使用，下面就来介绍下这几个 Valve 的功能：

1. REQUEST DUMPER VALVE

这个 Valve 的作用就是将请求头信息和相应头信息通过 log 打印出来，当然，读者们可能已经想到，通过 log 将请求的详细信息打印出来必然会对性能产生影响，但是如果仅仅是用来追踪问题，配置上这个 Valve 还是有点作用的。

2. REQUEST FILTER VALVE

RequestFilterValve 是用来请求的过滤的，通过配置 allows 和 denies 列表，可以允许一些请求的访问或阻止一些请求的访问，RequestFilterValve 是一个抽象类，它有两个默认的实现类 RemoteAddrValve 和 RemoteHostValve，这两个 Valve 分别可以根据请求 ip 地址和请求的 host 来过滤请求。当然，你也可以根据自己的需要来扩展 RequestFilterValve，这里需要注意的一点

是最好不要把业务相关的逻辑用 RequestFilterValve 来控制，这样势必会造成应用对 Tomcat 的依赖。

3. ACCESSLOGVALVE

AccessLogValve 的作用是将请求的详细信息打印出来，注意，它和 RequestDumperValve 是不同的，RequestDumperValve 打印的是请求头信息和相应头信息，而 AcessLogValve 的作用则更像 Apache Httpd 的 cookieLog，通过配置一定的 log 格式，来将你需要的请求信息打印出来，具体的格式配置大家可以参考 Tomcat 的官方文档

engine

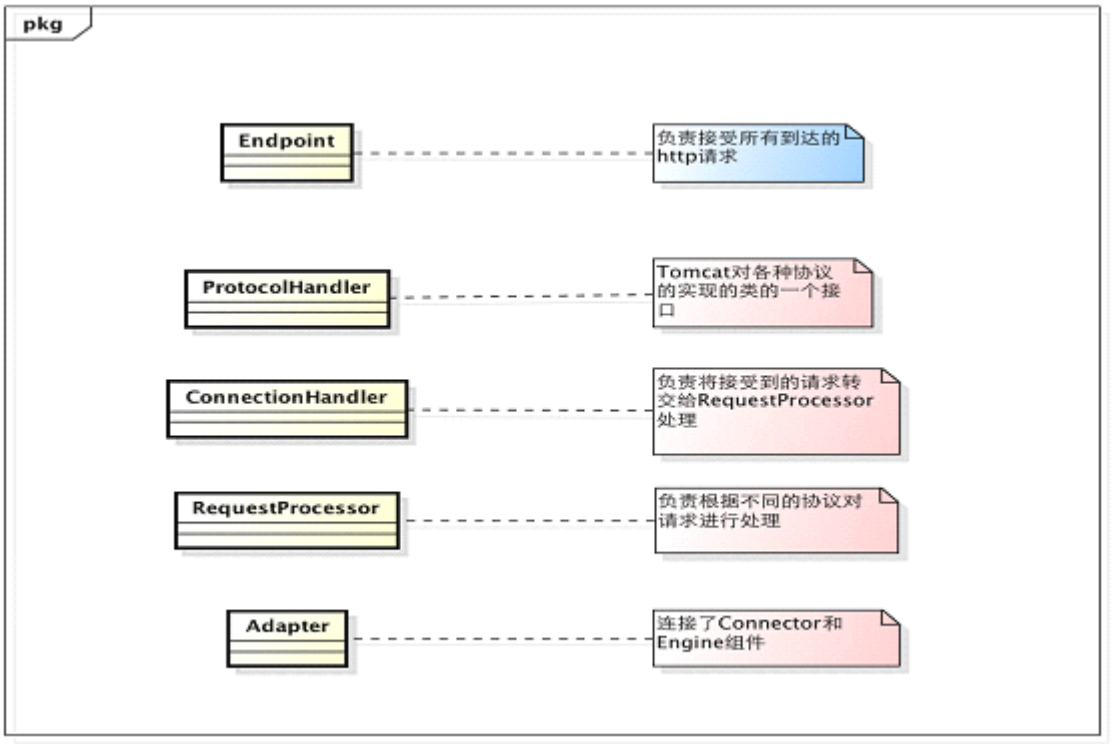
Tomcat Connector 组件

2
1 年前
由 **khotyn** 发布 在 **Tomcat**

Tomcat Connector 是 Tomcat 中的一个重要的组件，它负责监听到达 Tomcat 的请求，并将这些 请求转换成 Servlet 规范中所定义的 Request，然后将转换后的请求交给 Engine 组件去处理，最后将 Engine 返回的 Response 返回给客户端。

一、Connector 组件的主要请求处理类以及它们的功能

在了解 Connector 组件的大致功能之后，我们来看一下 Connector 组件里面的一些主要的类以及它们的职责，然后在后面的内容中，我们会讲述这些类是如何协同工作以完成 Connector 组件的功能的，下面是一张 Connector 组件中主要的类的类图（这里面并没有画出它们之间的关系，因为感觉没有太大的意义）：



在上面的这一张图中：

- EndPoint：从名字上可以看出，这个类代表的是一个 Socket 连接的一个端点，它主要负责接收所有到达的请求。
- ProtocalHandler：这个接口的实现类是 Tomcat 对各种协议的实现，包括 Http11Protocal 和 Http11NioProtocal 等等
- ConnectionHandler：这个类负责的主要工作是将接受到的请求交给 RequestProcessor 处理。
- RequestProcessor：这个类负责的是根据不同的协议来实现对请求的处理。
- Adapter：这个类看名字负责的就是适配器的工作，实际上它负责的工作是通过适配的方式来将 Connector 组件和 Engine 组件给连起来，说到这里，聪明的你一定有一个疑问，那么它到底将什么东西做了适配呢？这里我们先卖个关子，等到介绍到了 Connector 组件的再来说明这个问题。

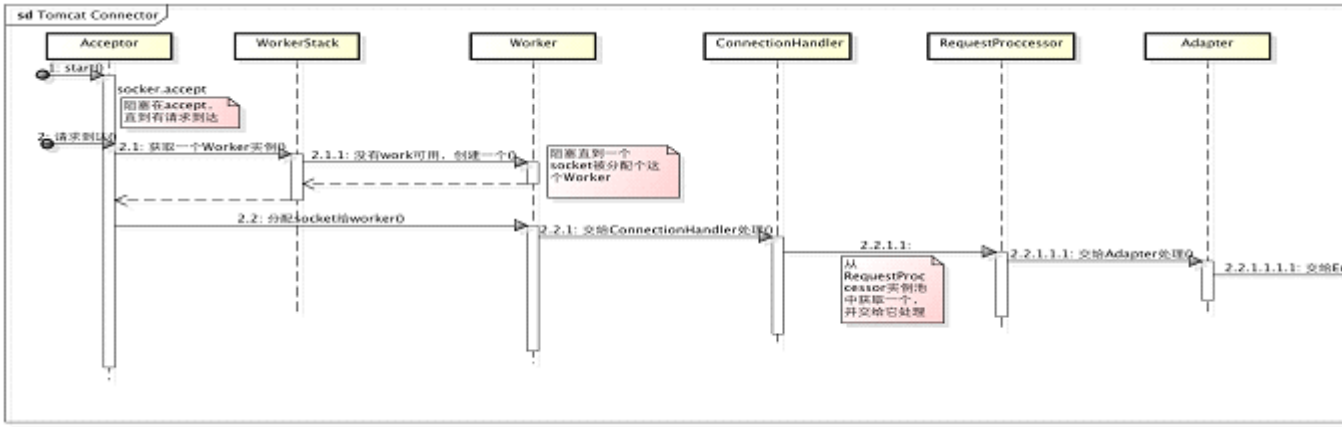
二、Connector 组件中 Request 和 Response 相关的类

在 Connector 中，对于 Request 和 Reponse，分别有三种：Coyote，Catalina，Facade，我们来看下它们之间有什么区别：

- Coyote：这一对 Request（org.apache.coyote.Request）和 Reponse（org.apache.coyote.Response）是专门用于 Tomcat 的内部表示的，和 Servlet 规范无关
- Catalina：这一对 Request（org.apache.catalina.connector.Request）和 Response（org.apache.catalina.connector.Response）实现了 Servlet 规范。到这里，聪明的读者一定已经想到，我们在上面提到的 Connector 的工作就是将 Coyote 的那对 Request 和 Response 适配成 Catalina 下的那对 Request 和 Response。
- Facade：这一对 Request（org.apache.catalina.connector.RequestFacade）和 Response（org.apache.catalina.connector.ResponseFacade）从名字上看一看出就是一个门面，它们的作用是做 Catalina 那对 Request 和 Response 的门面，从 Catalina 那对的 Request 和 Reponse 的源代码中我们可以看到，这一对 Request 和 Response 虽然实现了 Servlet 规范，但是另外还有一些其他一些 public 方法，这些方法如果直接暴露给 Web 应用，显然是不合适的，所以就加了这么一层门面，只暴露出去 Servlet 规范里面所定义的方法。

三、Connector 组件对请求的处理过程

在了解了 Connector 组件的一些主要的类以及它们的职责以后，我们以 Http11 为例来看写这些类是如何协同起来来对请求进行处理的，下面这张图简单的画了一下整个请求的处理过程：



[点击查看大图]

我们可以看到：

1. 在 Tomcat 启动的时候，JioEndpoint 的 Acceptor 阻塞在 Socket.accept()系统调用中，等待连接的进入。
2. 一旦接收到了一个连接，Acceptor 便从 WorkerStack 中取出一个 Worker 并将 Socket 交给 Worker 去处理
3. 而 Worker 拿到 Socket 以后将这个 Socket 交给了 ConnectionHandler 处理
4. ConnectionHandler 接着从 recycledProcessors 中取出一个 RequestProcessor，并把请求交给它去处理
5. 这个时候，这个 Socket 已经到了 RequestProcessor 手上了，RequestProcessor 就负责将 Socket 转换成 CoyoteRequest 和 CoyoteResponse，最后将请求交给了 Adapter 来处理。
6. 最后，请求到了 Adapter 这里，Adapter 接受到的参数仍然不是 Servlet 规范所定义的 Request 和 Response，Adapter 的工作就是将 CoyoteRequest 和 CoyoteResponse 转换成 Servlet 请求所定义的 Request 和 Response，并将这个 Request 和 Response 交给 Engine 去处理。

这是一个大致的 Connector 对请求的处理过程，当然，实际上的处理过程要比这个复杂的多，但是这么一个简化的过程对我们理解这些类的工作还是很有帮助的。

看完上面的处理过程后，读者可能会有一个疑问，实际上 Connector 在处理请求的时候维持了两份池，一份是 WorkStack，另一份是 RecycledProcessors，这两份池到底有什么用呢？先说下 WorkStack，它维护的是一份线程池，为了 Tomcat 能够多线程地处理请求；而 recycledProcessors 这个池维持的是一份对象池，至于为什么要维护这么一份对象池，我想是为了能够对对象进行重复的利用，大家知道，如果这些对象使用后直接释放必然最后会被虚拟机的垃圾回收器所回收，而垃圾回收必然会对 Tomcat 的性能造成一定程度的影响，为了尽可能高的提升 Tomcat 的性能，Tomcat 采用了这种对象回收利用的方法，我们在 Tomcat 的源代码中可以看到，有很多类都有一个 recycle 方法，用于把这些对象的状态归位，以便于用以处理后续的请求。

四、总结

这篇文章只是很简单地介绍了一下 Tomcat 的 Connector 组件，使大家对 Tomcat 的 Connector 组件有一个感性的认识，知道 Connector 组件大概是如何工作的。另外，Tomcat 的 Connector 组件里面还有许多值得我们深入学习的东西，比如 Nio 的处理，对象的重复利用等等等等，期待大家有深入的学习并且有精彩的分享！

Connector