Servlet工作原理解析

servlet容器的启动过程

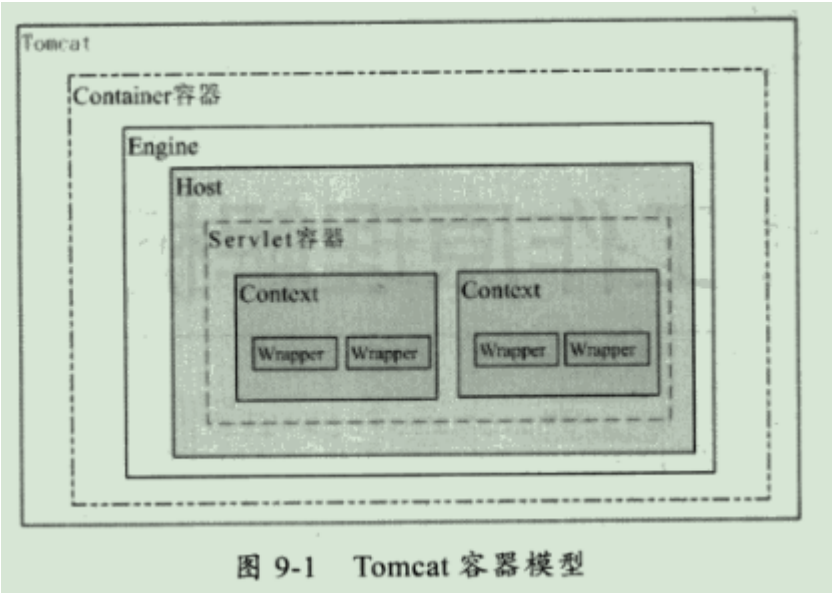

图 9-1　Tomcat 容器模型

从图中可以看出tomcat的容器分为四个等级，真正管理Servlet的容器是Context容器，一个Context对应一个web工程，在tomcat的配置文件中可以很容易的发现这一点

```
<Context path="/projectOne" docBase="D:\projects\projectOne"
reloadable="true"/>
```

关于tomcat中增加一个web应用，然后启动tomcat并调用其中的一个HelloWorldExample Servlet。

```
Tomcat tomcat getTomcatInstance();
File appDir = new File(getBuildDirectory(),"webapps/example");
tomcat.addWebapp(null,"/examples",appDir.getAbsolutePath());
tomcat.start();
ByteChunk res =
getUrl("http://localhost:"+getPort()+"/examples/servlets/servlet/HelloWorldE
xample");
assertTrue(res.toString().indexOf("<h1>hello World</h1>") > 0);
```

Tomcat的addWebapp的代码如下：

```
```

```
public Context addWebapp(Host host,String url,String path){
   silence(url);
   Context ctx = new StandardContext();
   ctx.setPath(url);
   ctx.setDocBase(path);
   if(defaultRealm == null){
      initSimpleAuth();
   }
   ctx.setRealm(defaultRealm);
   ctx.addLifecycleListener(new DefaultWebXmlListener());
   ContextConfig ctxCfg = new ContextConfig();
   ctx.addLifecycleListenner(ctxCfg);

 ctxCfg.setDefaultWebXml("org/apache/catalin/startup/NO_DEFAULT_XML");
   if(host == null){
      getHost().addChild(ctx);
   }else{
      host.addChild(ctx);
   }
   return ctx;
}
```

一个web应用对应一个Context容器，也就是Servlet运行时的Servlet的容器。添加一个Web应用时会创建一个StandardContext容器，并且给这个Context容器设置必要的参数，url和path分别代表这个应用在tomcat中访问路径和这个应用实际的物理路径，这两个参数与tomcat配置中的两个参数是一致的。其中最重要的配置时ContextConfig，这个类将会负责整个web应用配置的解析工作。最后将这个Context容器加到父容器host中。

添加examples应用所对应的StandaedContext容器的启动过程：

当Context容器初始化状态设置为init时，添加到Contex容器的Listenner将会被调用。ContextConfig继承了LifecycleListenner接口，他是在调用Tamcat.addWebapp被加入到StandardContext的容器中的。ContextConfig类将会负责整个Web应用中的配置文件的解析工作。

ContextConfig的init方法主要完成以下工作：

1. 创建用于解析XML配置文件的contextDigester对象。
2. 读取默认context.xml配置文件，如果存在解析他。
3. 读取默认Host配置文件，如果存在解析他。
4. 读取默认Context自身的配置文件，如果存在解析他。
5. 设置Context的DocBase。

ContextConfig的init方法完成之后，Context容器就会执行startInternal方法，这个方法的启动逻辑比较复杂，主要包括以下几个部分：

1. 创建读取资源文件的对象。
2. 创建ClassLoader对象
3. 设置应用的工作目录。

4. 启动相关辅助类，如Logger、realm、resources等。
5. 修改启动状态，通知感兴趣的观察者（web应用的配置）
6. 子容器的初始化
7. 获取ServletContext并设置必要的参数
8. 初始化“load on startup”的Servlet

web应用的初始化工作

web应用的初始化工作是在ContextConfig的configureStart方法中实现的，应用的初始化主要是解析web.xml文件，这个文件描述了一个web应用的关键信息，也是一个web应用的入口。

Tomcat首先会找到globalWebXml，这个文件的搜索路径是engine的工作目录下的org/apache/catalin/startup/NO_DEFAULT_XML或conf/web.xml.接着会找hostWebXml,这个文件可能会在System.getProperty("catalina.base")/conf/$EngineName$/{HostName}/web.xml.default中，接着寻找应用的配置文件examples/WEB-INF/web.xml.web.xml文件中的各个配置项将会被解析成相应的属性保存在WebXml对象中。如果当前应用支持Servlet3.0，解析还将完成额外9项工作，这额外的9项工作主要是Servlet3.0新增的特性（包括jar包中META-INF/web-fragment.xml）的解析及对annotations的支持。

接下来会将webXml对象中的属性设置到Context容器中，这里包括创建Servlet对象、filter、Listenner等、这段代码在WebXml的configureContext方法中。下面是解析Servlet的代码片段：

```
for(ServletDef servlet:servlets.values()){
   Wrapper wrapper = context.createWrapper();
   String jspFile = servlet.getJspFile();
   if(jspFile != null){
      wrapper.setJspFile(jspFile);
   }
   if(servlet.getLoadOnStartup()!= null){
wrapper.setLoadOnStartup(servlet.getLoadOnStartup().intValue());
}
   if(servlet.getEnabled() != null){
      wrapper.setEnabled(servlet.getEnabled().booleanValue());
   }
   wrapper.setName(servlet.getServletName());
   Map<String,String> params = servlet.getParameterMap();
   for(Entry<String,String> entry:params.entrySet)
   {
      wrapper.addInitParameter(entry.getKey(),entry.getValue());
   }
   wrapper.setRunAs(servlet.getRunAs());
   Set<SecurityRoleRef> roleRefs = servlet.getSecurityRoleRefs();
   for(SecurityRoleRef roleRef:roleRefs){
      wrapper.addSecurityReference(

      roleRef.getName(),roleRef.getLink());
```

```
        roleRef.getName(),roleRef.getLink());
    }
    wrapper.setServletClass(servlet.getServletClass);
    MultipartDef multipartdef = servlet.getMultipartDef();j
    if(multipartdef != null){
      if(multipartdef.getMaxFileSize() != null &&
      nultipartdef.getMaxRequestSize() != null &&
      multipartdef.getFileSizeThreshold() != null){
        wrapper.setMultipartConfigElement(new MultipartConfigElement(
        multipartdef.getLocation(),
        Long.parseLong(multipartdef.getMaxFileSize()),
       Long.parseLong(multipartdef.getMaxRequestSize()),
       Integet.parseInt(multipartdef.getFileSizeThreshold())));
      }else{
        wrapper.setMultipartConfigElement(new
MultipartConfigElement(multipartdef.getLoacation()));
      }
    }
    if(servlet.getAsyncSupported() != null){

 wrapper.setAsyncSupported(servlet.getAsyncSupported().booleanValue());
    }
    context.addChild(wrapper);
}
```

这段代码清楚的描述了如何将Servlet包装成Context容器中的StandardWrapper,因为Servlet是一个独立的web开发标准，而不是Tomcar中的一部分，所以需要将Servlet包装成一个StandardWrapper。

除了将Servlet包装成StandardWrapper并作为子容器添加到Context中，其他的所有web.xml属性都被解析到Context中，所以说Context容器才真正运行Servlet的Servlet容器。一个web应用对应一个Context容器，容器的配置属性由应用的web.xml指定，这样我们就能理解web.xml到底起什么作用了。

创建Servlet对象

如果Servlet的load-on-startup配置项大于0，那么在Context容器启动的时候就会被实例化，前面提到在解析配置文件时会读取默认的globalWebXml，在conf下的web.xml文件中定义一些默认的配置项，其中定义了两个Servlet，分别是org.apache.catalina.servlets.DefaultServlet和org.apache.jasper.servlet.JspServet.他们的load-on-startup分别是1和3.

创建Servlet实例的方法是从Wrapper.loadServlet开始的。loadServlet方法要完成的就是获取ServletClass，然后把他交给InstanceManager去创建一个基于ServletClass.class的对象。如果这个对象配置了jsp-file，那么这个ServletClass就是conf/web.xml中定义的org.apache.jasper.servlet.jspServlet了。

创建Servlet对象的相关类结构：

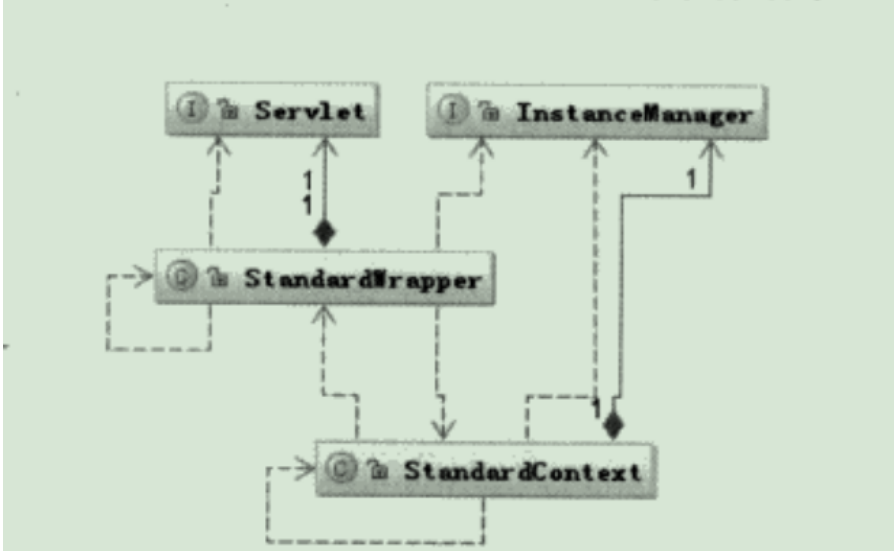

图 9-3　创建 Servlet 对象的相关类结构

初始化Servlet

初始化Servlet在StandardWrapper的initServlet方法中，这个方法很简单，就是调用Servelet的init()方法，同时把包装了StandardWrapper对象的StandardWrapperFacade作为ServletConfig传给Servlet。

如果该Servlet关联的是一个JSP文件，那么前面初始化的就是JspServlet，接下去会模拟一次简单的请求，请求调用这个jsp文件，以便编译这个jsp文件为类，并初始化这个类。

Servlet的体系结构

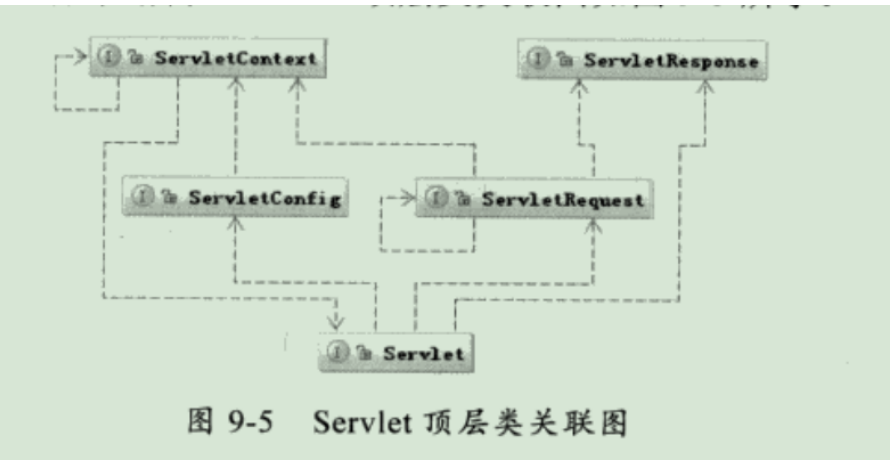

图 9-5　Servlet 顶层类关联图

与Servlet主动关联的是三个类，分别是ServletConfig、ServletRequest和ServletResponse。这三个类都是通过容器传递给Servlet的，其中ServletConfig在Servlet初始化时就传给了Servlet了。而后两个是在请求到达时调用Servlet传递过来的。

ServletConfig是在servlet init时由容器传过来的。

ServletConfig在容器中的类关联图

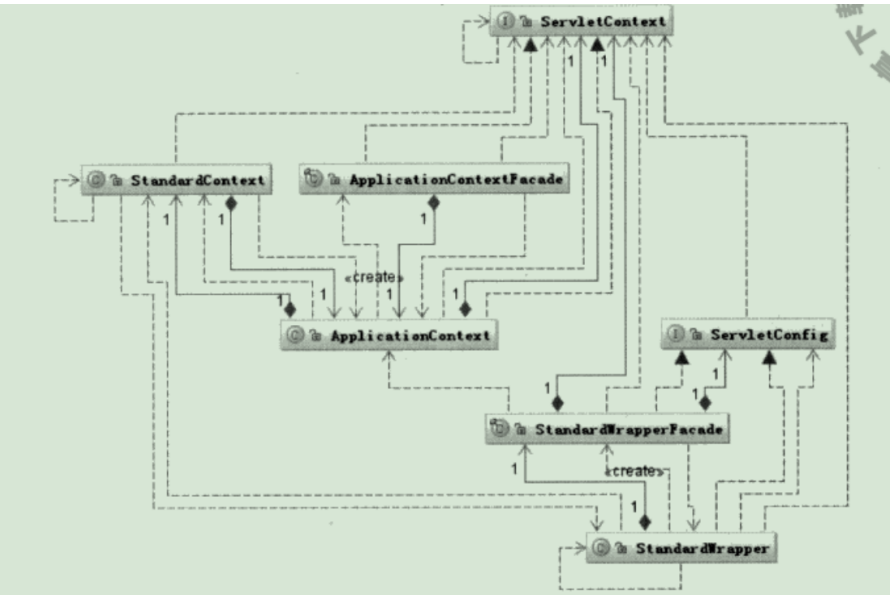

图 9-6　ServletConfig 在容器中的类关联图

可以看出，StandardWrapper和StandardWrapperFacade都实现了ServletConfig接口，而StandardWrapperFacade是StandardWrapper门面类。所以传给Servlet的是StandardWrapperFacade对象，这个类能够保证从StandardWrapper中拿到ServletConfig所规定的数据，而又不把ServletConfig不关心的数据暴露给Servlet。

同样ServletContext也与ServletConfig有类似的结构，Servlet中拿到的ServletContext的实际对象也是ApplicationContextFacade对象。APplicationContextFacade同样保证了ServletContext只能充容器中拿到他该拿的数据，他们都气到对数据的封装作用，他们使用的都是门面设计模式。

Tomcat接到请求首先将会创建org.apache.coyote.Request和org.apache.coyote.Response,这两个类是Tomcat内部使用的描述一次请求和相应的信息类，他们是一个轻量级的类，作用就是在服务器接受到请求之后，经过简单的解析将这个请求快速的分配给后续线程去处理，所以他们的对象很小，很容易被JVM回收。接下去当交给一个用户线程去处理这个请求时由创建了org.apache.catalina.connector.Request和org.apache.catalina.connector.Response对象。这两个对象一直贯穿了整个Servelet容器知道要传给Servlet，传给Servlet的Request和Response的门面类是RequestFacade和ResponseFacade。

Request相关类结构图

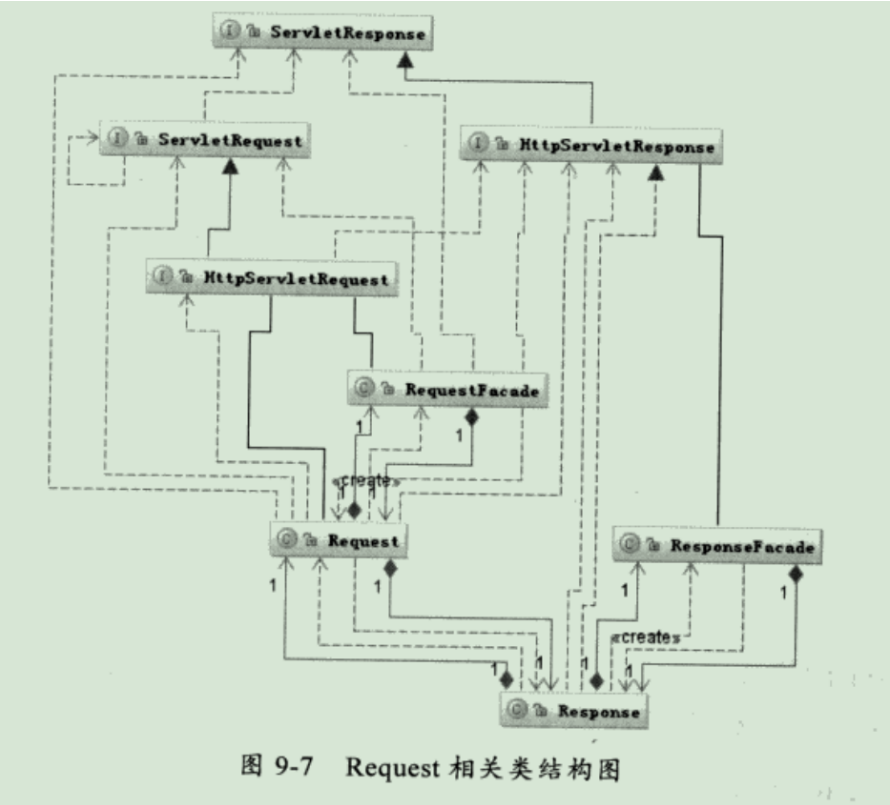

图 9-7　Request 相关类结构图

Request和Response的转变过程

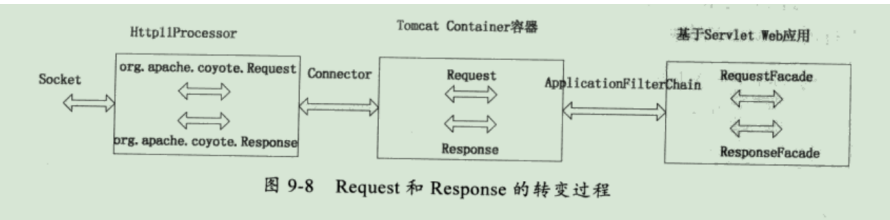

图 9-8　Request 和 Response 的转变过程

Servlet如何工作

服务器如何根据url来到达正确的Servlet容器中呢?

Tomcat7中这件事很容易解决，因为这种映射工作由专门一个类来完成，这个类就是org.apache.tomcat.util.http.mapper,这个类保存了Tomcat的Container容器中的所有子容器的信息，org.apache.catalina.connector.Request类在进入Container容器之前，Mapper将会根据这次请求的hostname和contextpath将host和context容器设置到Request的mappingData属性中。所以当Request进入Container容器之前，他要访问那个子容器就已经确定了。

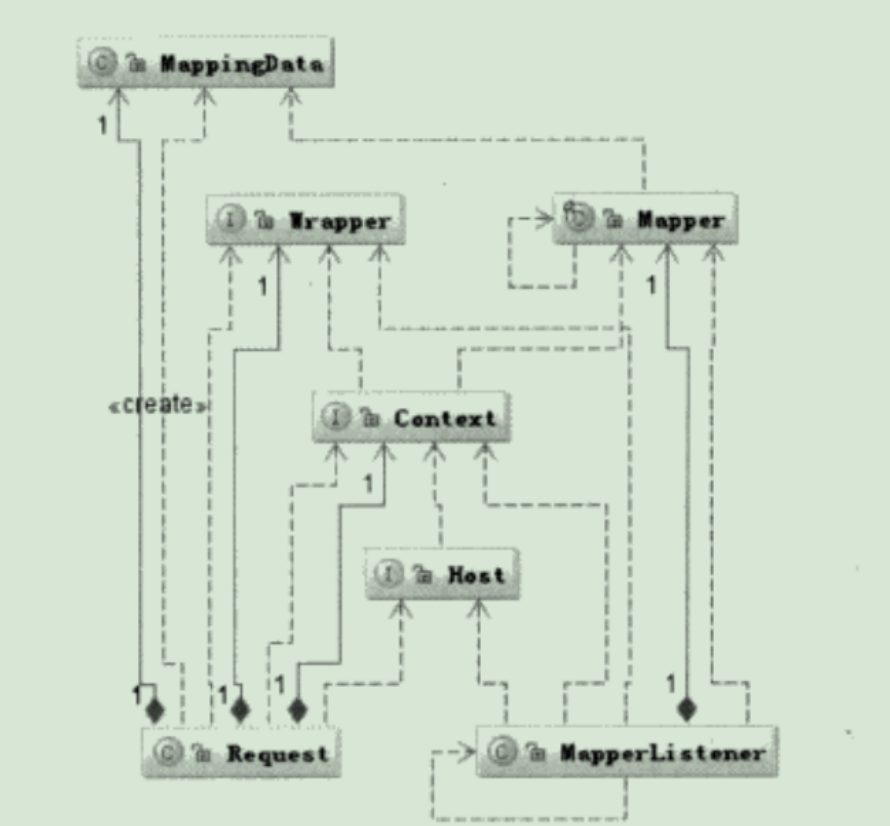

图 9-9　Request 的 Mapper 类关系图

Request在容器中的路由图：

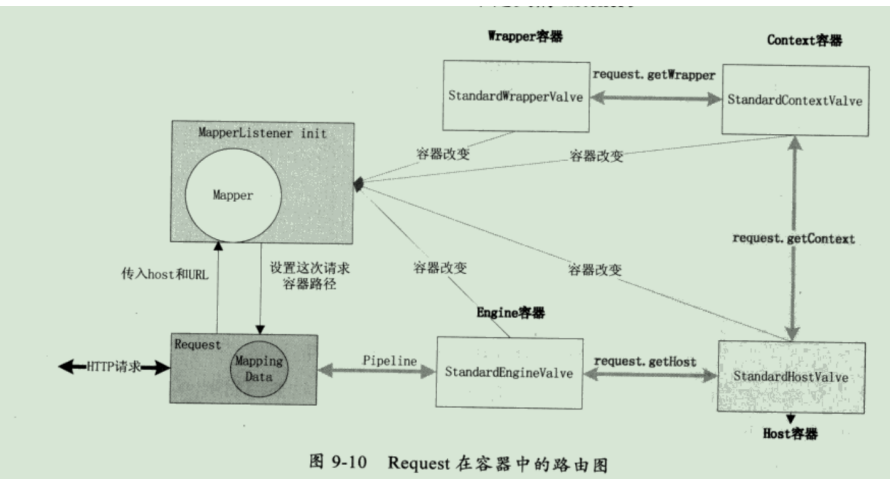

图 9-10　Request 在容器中的路由图

Servlet的确应已经能够帮我们完成所有工作了，但是现在的Web应用很少直接将交互全部页面都用Servlet来实现，而是采用更高效的MVC框架来实现。这些MVC框架的基本原理是将所有的请求都映射到一个Servlet，然后取实现Service方法，这个方法也就是MVC框架的入口。

当Servlet从Servlet容器总移除时，也就表明Servlet的生命周期结束了，这是Servlet的destory方法将被调用，做一些扫尾工作。