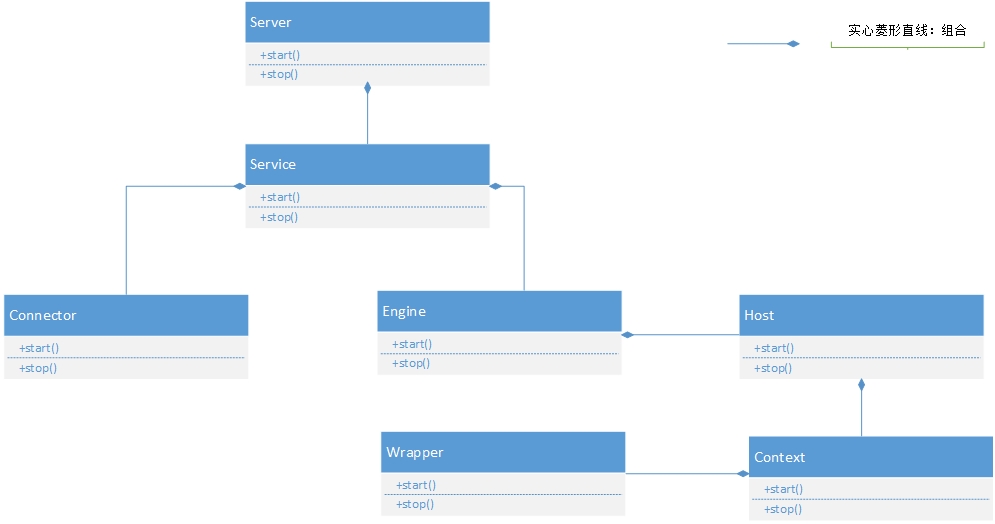
1. 总体架构



1. Server

从最基本的功能来讲，我们可以将服务器描述为这样一个应用：它接受其他计算机（客户端）发来的请求数据并进行解析，完成相关业务处理，然后把处理结果作为响应返回给请求计算机（客户端）。我们用Server类（接口）表示一个服务器，通过start()方法启动服务器，打开Socket链接，监听服务器端口，并负责在接收到客户端请求时进行处理并返回响应，同时提供一个stop()方法停止服务器并释放网络资源。

1. Service、Container、Connector

一个Server包含多个Service（它们相互独立，只是共享一个JVM以及系统类库），一个Service负责维护多个Connector和一个Container，这样来自Connector的请求只能由它所属Service维护的Container处理。Connector负责开启Socket并监听客户端请求、返回响应数据，不同Connector对应不同协议（AJP、HTTP、HTTP/2），绑定协议对应端口。Container负责具体的请求处理。

Service、Container、Connector分别拥有自己的start()和stop()方法来加载和释放自己维护的资源。

在Tomcat中，Container是一个更加通用的概念。为了与Tomcat中的组件命名一致，我们将Container重新命名为Engine，用以表示整个Servlet引擎。

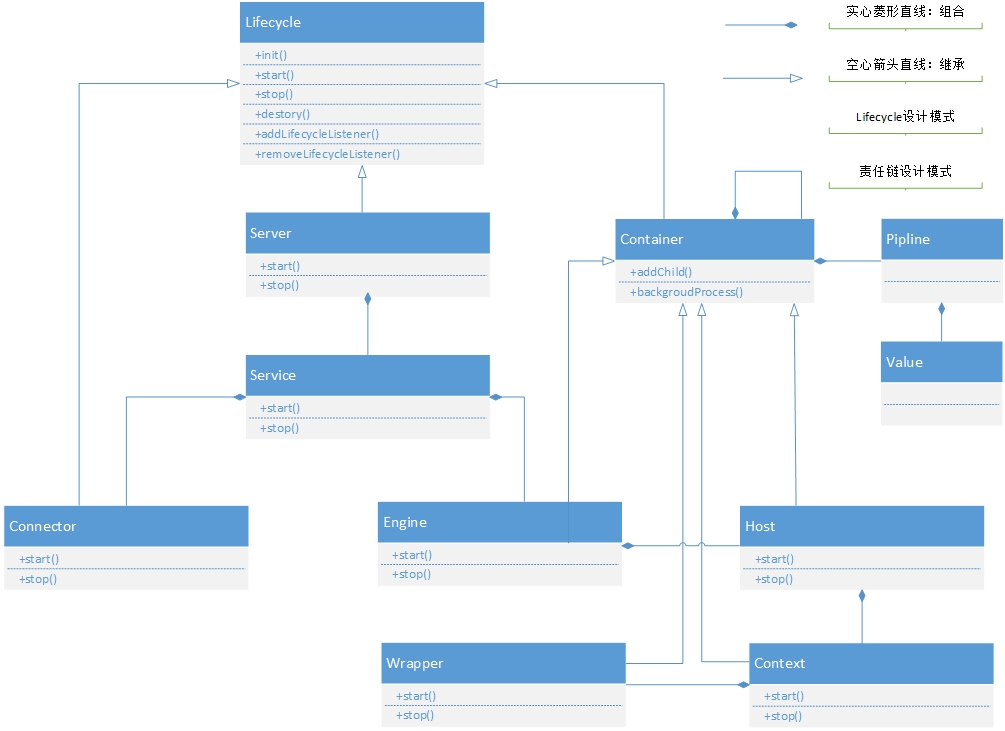
1. Container设计

* Container表示能够执行客户端请求并返回响应的一类对象。在Tomcat中存在不同级别的容器：Engine、Host、Context、Wrapper。

我们使用Context来表示一个Web应用（webApps目录下的应用），并且一个Engine可以包含多个Context。

我们用Host表示虚拟主机的概念，一个Engine可以包含多个Host。如果我们想在一个服务器实例上提供多个域名的服务，那么就可以将每个域名视为一个虚拟主机。一个Host可以包含多个Context，在每个虚拟主机下包含多个Web应用。

Wrapper作为一类容器，用于表示Web应用中定义的Servlet。



* Lifecycle设计模式

我们很容易发现，所有组件均存在启动、停止等生命周期方法，拥有生命周期管理的特征。因此，我们可以基于生命周期管理进行一次接口抽象，我们针对所有拥有生命周期管理特征的组件抽象了一个Lifecycle通用接口，该接口定义了生命周期管理的核心方法。

* init() : 初始化组件。
* start() : 启动组件。
* stop() : 停止组件。
* destroy() : 销毁组件。
* 责任链设计模式

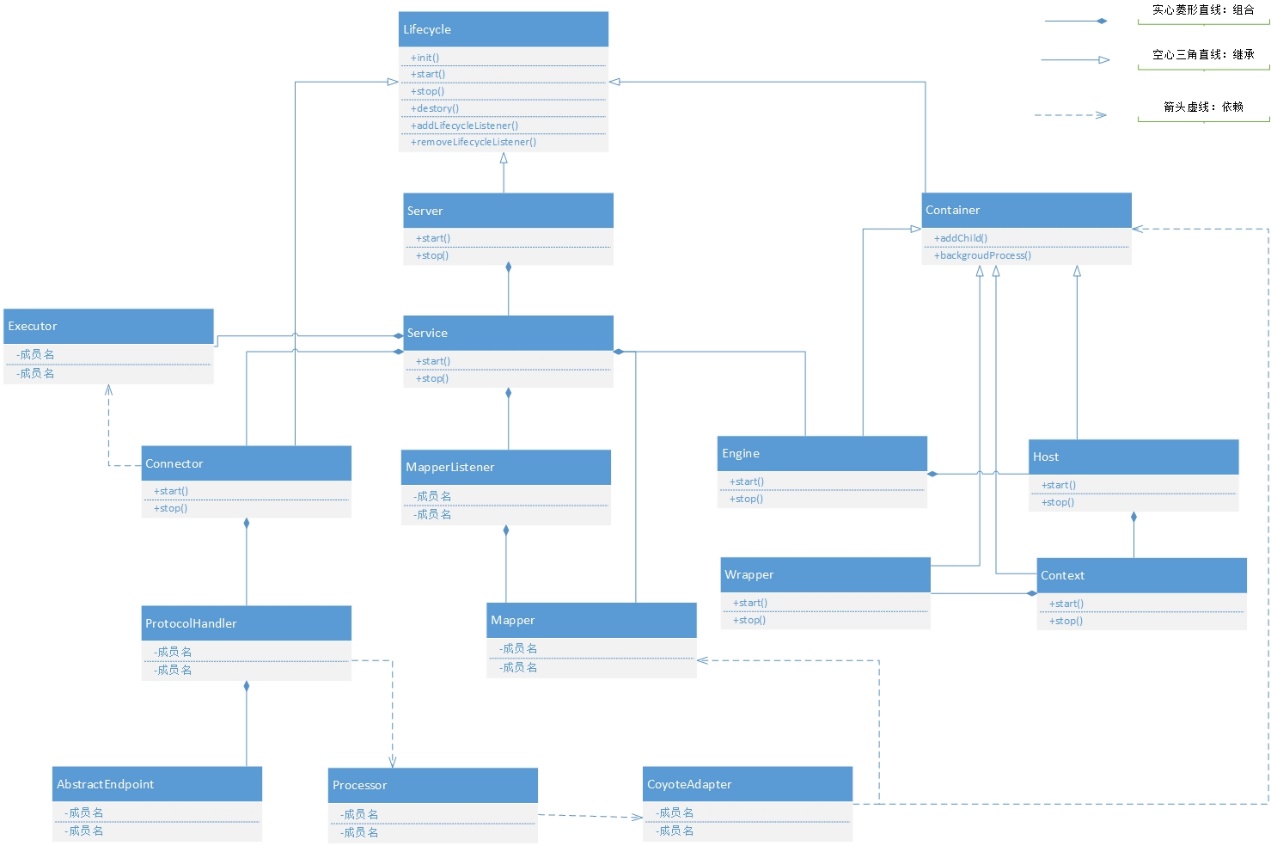
Tomcat定义了Pipeline（管道）和Value（阀）两个接口。前者用于构造职责链，后者代表职责链上的每个处理器。当然，我们还可以从字面意义来解释这两个接口所扮演的角色——来自客户端的请求就像是流经管道的水一般，经过每个阀进行处理。

Tomcat容器组件的灵活之处在于，每个层级的容器（Engine、Host、Context、Wrapper），均有对应的基础Value实现，同时维护了一个Pipeline实例。也就是说，我们可以在任何层级的容器上针对请求处理进行扩展。

由于Tomcat每个层级容器均通过Pipeline和Value进行请求处理，那么，我们很容易将一些通用的Value实现根据需要添加到任何层级的容器上。

基础Value负责获取启动下一级容器。

1. Connector设计



Connector与Container配合实现一个完整的服务器功能，完成如下几项功能。

* 监听服务器端口，读取来自客户端的请求。
* 将请求数据按照指定协议进行解析。
* 根据请求地址匹配正确的容器进行处理。
* 将响应返回给客户端。

Tomcat支持多协议：AJP 、HTTP、HTTP/2，支持BIO、NIO、NIO2、APR等I/O方式。

在Tomcat中，Protocol Handler表示一个协议处理器，针对不同协议和I/O方式，提供了不同的实现，如Http11NioProtocol表示基于NIO的HTTP协议处理器。ProtocolHandler包含一个Endpoint用于启动Socket监听，该接口按照I/O方式进行分类实现，如Nio2Endpoint表示非阻塞式Socket I/O。还包含一个Processor用于按照指定协议读取数据，并将请求交由容器处理，如Http11NioProcessor表示在NIO的方式下HTTP请求的处理类。

当Processor读取客户端请求后，需要按照请求地址映射到具体的容器进行处理，这个过程即为请求映射。Tomcat通过Mapper和MapperListener两个类实现上述功能。前者用于维护容器映射信息，同时按照映射规则（Servlet规范定义）查找容器。后者实现了ContainerListener和LifecycleListener，用于在容器组件状态发生变化时，注册或者取消对应的容器映射信息。

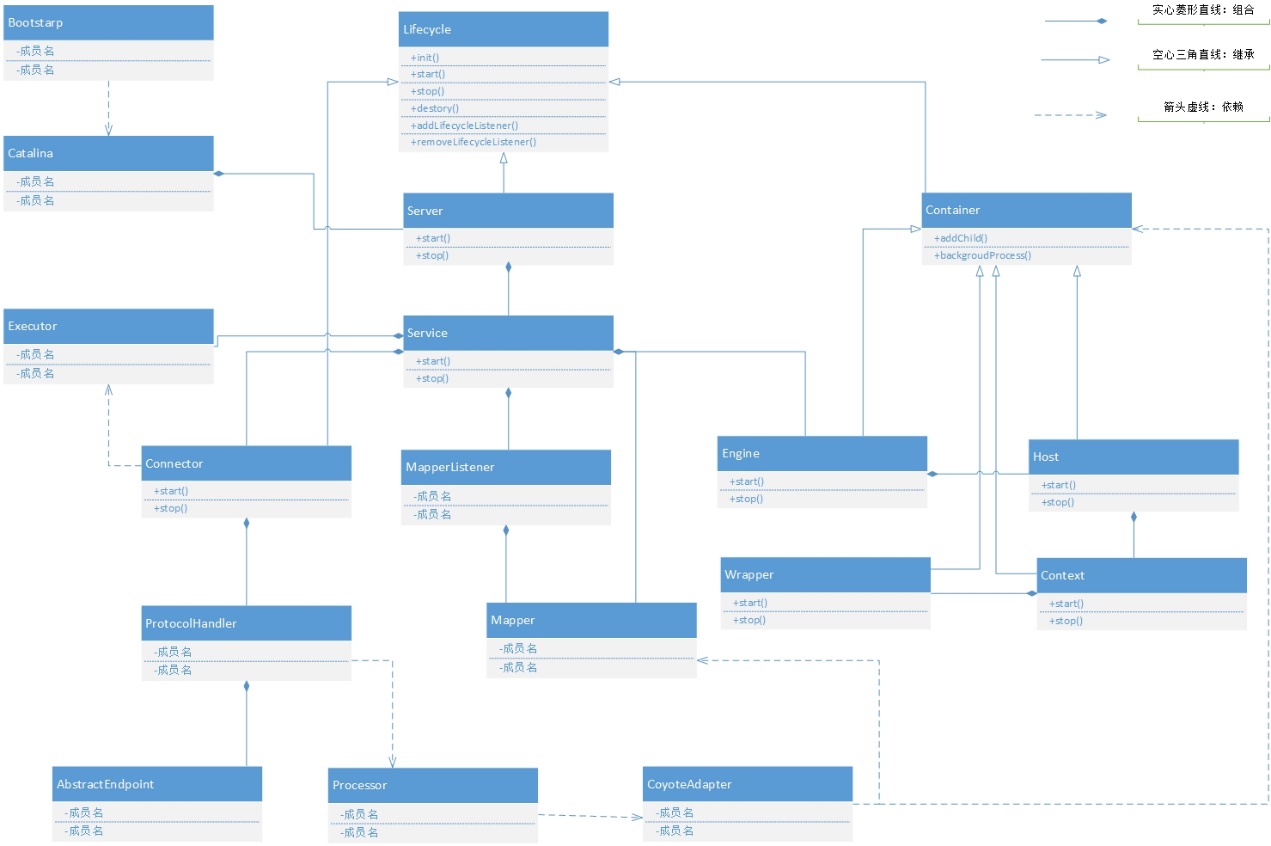
Tomcat通过适配器模式（Adapter）实现了连接器（Connector）与Mapper、容器（Container）的解耦。Tomcat默认的Connector实现对应的适配器为CoyoteAdapter。

1. Executor（并发）

Tomcat提供了Executor接口来表示一个可以在组件间共享的线程池，该接口同样继承自Lifecycle接口，可按照通用的组件进行管理。

Tomcat中Executor由Service维护，因此同一个Service中的组件可以共享一个线程池。比如Endpoint会依赖线程池启动线程监听Socket端口，当接收到客户端请求后，会创建请求处理对象，并交由线程池处理，由此支持并发处理客户端请求。

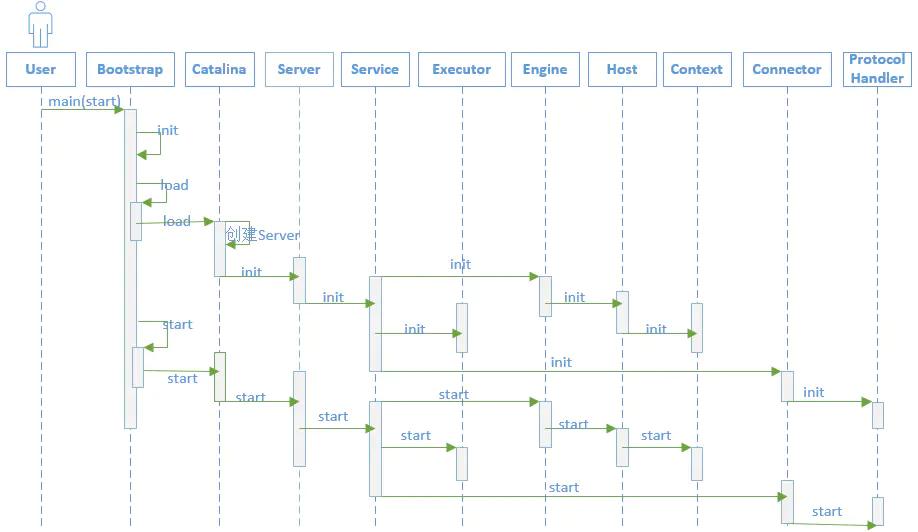
1. Bootstrap和Catalina



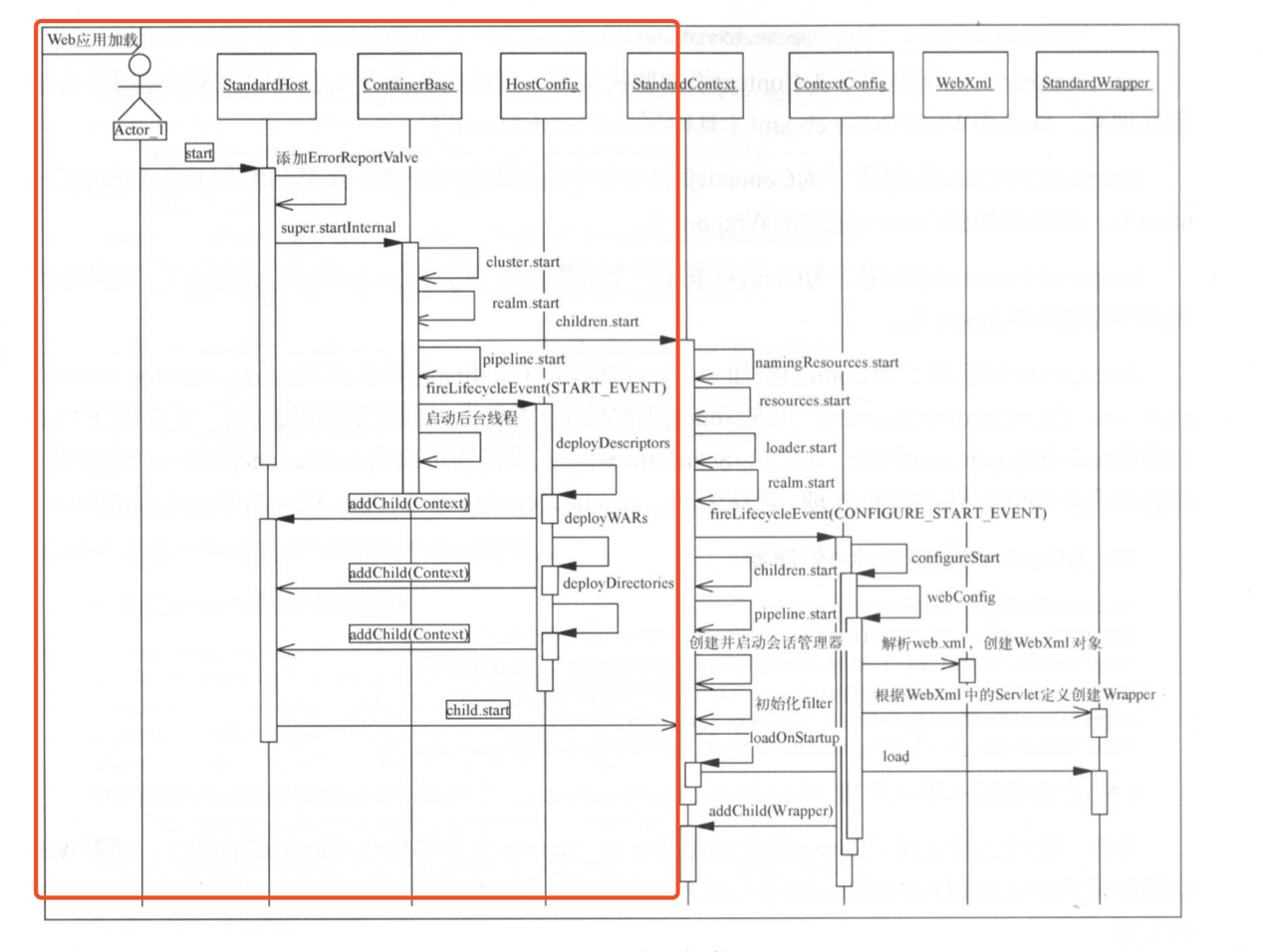
最后，Tomcat提供了Bootstrap作为应用服务器启动入口。Bootstrap负责创建Catalina实例，根据执行参数调用Catalina相关方法完成针对应用服务器的操作（启动、停止）。Bootstrap和Tomcat应用服务器完全松耦合，Bootstrap通过反射调用Catalina实例，实现了启动入口与核心环境的解耦。

1. Tomcat启动

* LifeCycle生命周期管理
* 事件监听设计模式

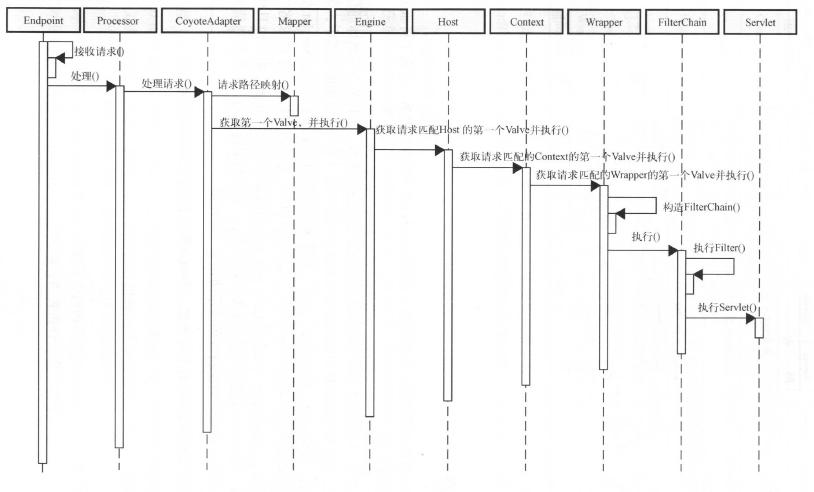


Tomcat的启动过程非常标准化，统一按照生命周期管理接口Lifecycle的定义进行启动。首先，调用init()方法进行组件的逐级初始化，然后再调用start()方法进行启动



Container启动采用事件监听设计模式，比如StrandHost启动时触发START\_EVENT事件(容器基础父类ContainerBase实现)，生命周期监听器HostConfig监听到事件执行部署war包。

1. 请求处理



官网：

<https://tomcat.apache.org/tomcat-8.5-doc/architecture/requestProcess/request-process.png>

前面我们曾讲到，Tomcat采用职责链模式来处理客户端请求，以提高Servlet容器的灵活性和可扩展性。Tomcat定义了Pipeline（管道）和Value（阀）两个接口，前者用于构造职责链，后者代表职责链上的每个处理器。由于Tomcat每一层Container均维护了一个Pipeline实例，因此我们可以在任何层级添加Value配置，以拦截客户端请求进行定制处理（如打印请求日志）。

Tomcat的每一级容器均提供了基础的Value实现已完成当前容器的请求处理过程（如StandardHost对应的基础Value实现为StandardHostValue），而且基础Value实现始终位于职责链的末尾，以确保最后执行。

我们看一下一个典型的Value实现：

*class* SampleValve *extends* ValveBase {  
 @Override  
 *public final void* invoke(Request request, Response response)  
 *throws* IOException, ServletException {

*//do something*  
 getNext().invoke(request, response);  
 }  
}

由上可知，只要我们得到Pipeline中的第一个Value即可以启动整个职责链的执行。每一级Container的基础Value在完成自身处理的情况下，同时还要确保启动下一级Container的Value链的执行。

在StandardWrapperValue中（由于Wrapper为最低一级的Container，且该Value处于职责链末端，因此它始终最后执行），Tomcat构造FilterChain实例完成javax.servlet.Filter责任链的执行，并执行Servlet.service()方法将请求交由应用程序进行分发处理（如果采用了如SpringMVC等Web框架的话，Servlet会进一步根据应用程序内部的配置将请求交由对应的控制器处理）。