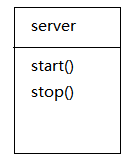
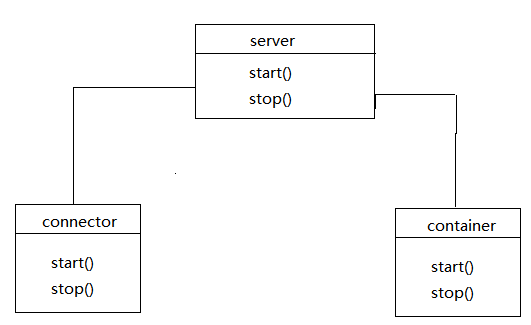
Server

通常情况下，我们通过使用socket监听服务器指定端口来实现该功能，即接收客户端请求并处理，然后返回相应的数据。

最简单的实现如下：



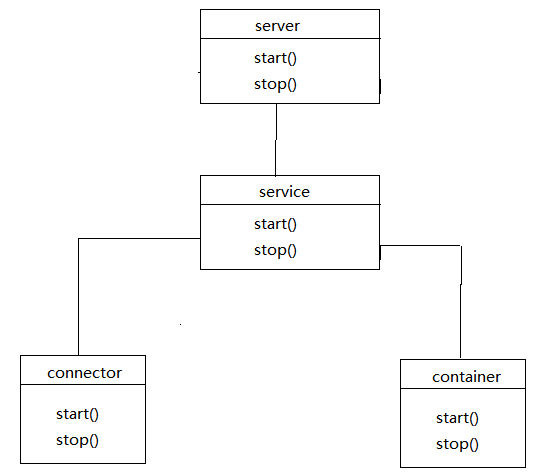
但是，我们发现，将请求监听和请求处理放在一起扩展性会很差（至少可以使用桥接模式将两个不同维度的组件单独出来，独立扩展）。于是出现了如下的设计：



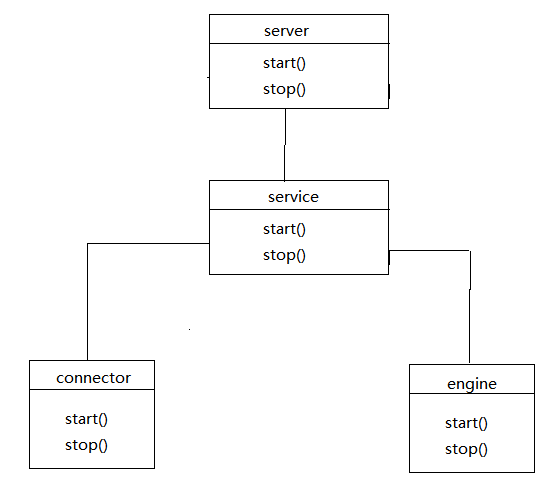
将服务端组件分为Connector和container，Connector负责开启socket并监听客户端请求、返回响应的数据；Container负责具体的请求处理

这样设计也很好地兼容了不同网络协议的处理问题，我们只需要使用connector来接收网络的请求，不同的connector可能接收不同的网络协议的请求，但是它们面对的处理是一个，即container是一个。

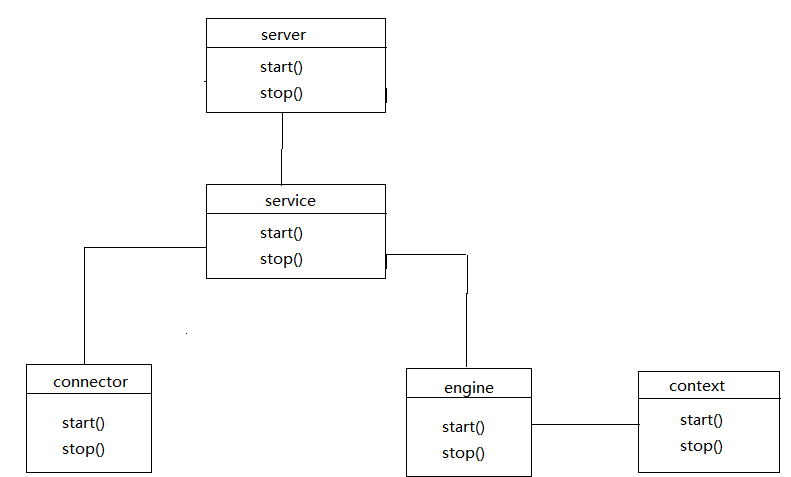
为了维护connector接受的请求和container的处理之间的映射关系，我们在connector和container上加上一层service，以便更好的管理请求和响应。一个server包含多个service（它们相互独立，只是共享一个jvm以及系统类库），一个service负责维护多个connector和一个container。



Container就是engine，是servlet引擎。Engine表示整个servlet引擎而非servlet容器，表示整个servlet容器的是server。引擎只负责请求的处理，并不需要考虑请求的连接、协议等的处理。为了与tomcat中的组件命名一致，我们的架构变为如下：

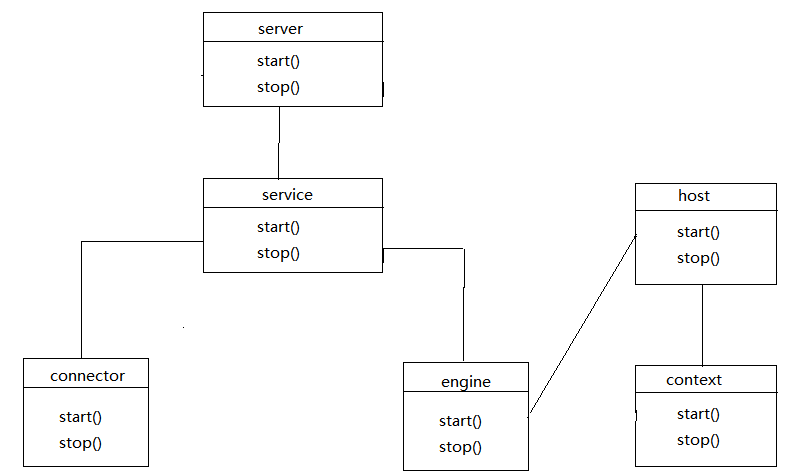


上面我们已经解决了网络协议和容器的解耦，但是应用服务器时用来部署并运行web环境的，而不是一个简单的接收connector转发的请求然后处理的业务处理系统，engine需要找到web应用来处理这些请求，进而架构变为如下这样：



这里我们使用Context来表示一个web一个应用，并且一个engine可以包含多个context.

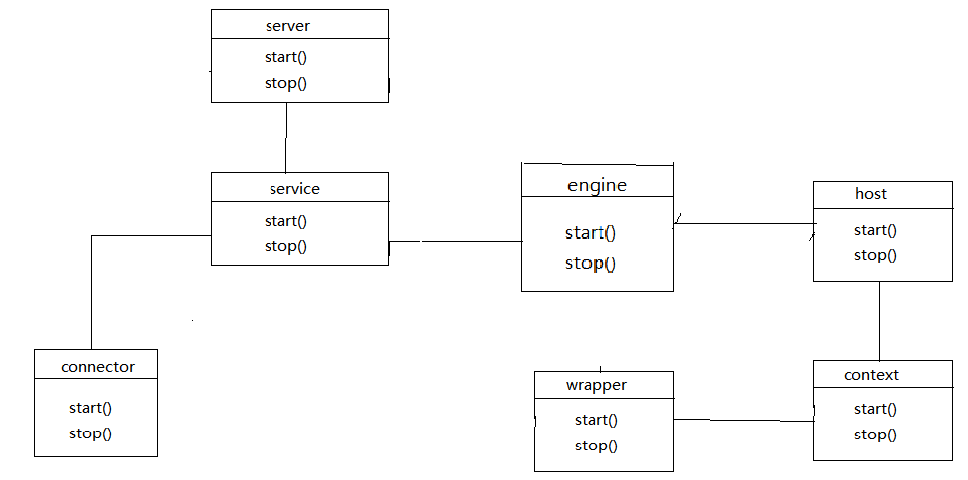
我们在一个主机上运行一个服务器示例即sever，但是我们希望接收对多个域名的请求服务，我们就可以将每个域名视为一个虚拟主机，在每个虚拟主机下包含多个web应用，于是架构有演变为如下这样：



Host表示虚拟主机的概念，一个host可以包含多个Context

在tomcat的设计中，engine既可以包含host，又可以包含context，具体还要看engine的实现，默认实现StandardEngine只能包含host

Servlet规范中，在一个web应用中可以包含多个servlet实例以常理来自不同链接的请求，因此我们需要一个组件概念来表示servlet定义，在tomcat中使用wrapper来表示servlet概念。于是架构又做了如下改变：



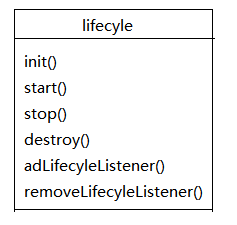
容器代表了一类组件，这类组件的作用就是处理接收字客户端的请求并返回响应数据。

Service持有的是engine接口，8.5版本之前为container接口

Tomcat的container可以表示不同的概念级别：servlet引擎，虚拟主机host，web引用和servlet。此外，tomcat的container还有一个很重要的功能，就是后台处理，在很多情况下，我们的container需要执行一些异步处理，而且是定期执行，如每隔30秒执行一次，tomcat对于web应用文件的扫描就是通过该机制实现的。

在tomcat中，backgroundProcess为 异步后台处理类

既然所有的组件都有start和stop方法，我们就可以让这些组件都实现同一个lifecyle管理的接口，该接口的设计如下：



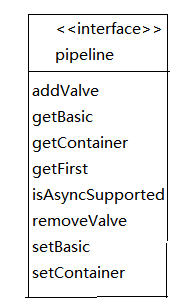
在tomcat中，核心组件都默认继承自LifecyleMBeanBase抽象类。该类不但负责组件各个状态的转换和事件处理，还将组件自身注册为MBean，以便通过Tomcat管理工具进行动态维护。

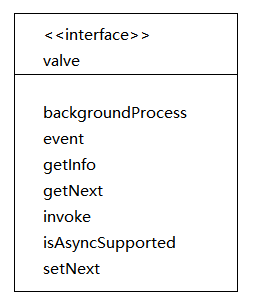
每个状态的转换都可能触发生命周期事件，但这不是必须的。

pipeline和valve

在增强组件的灵活性和可扩展性方面，职责链模式是一种比较好的选择，tomcat即采用该模式来实现客户端请求的处理——请求处理也是职责链模式典型的应用场景之一。在tomcat中，每个组件通过执行一个职责链来完成具体的请求处理。

Tomcat定义了pipelline和valve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。从字面意义来理解，来自客户端的请求就像流经管道的水一般，经过每个阀门的处理。





Pipeline中维护了一个基础的valve，它始终位于pileline的末端（即最后执行），封装了具体的请求处理和输出响应的过程，然后通过addValve方法添加其他的valve，后添加的valve位于基础valve之前，并按照添加顺序执行，pipeline通过获得首个valve来启动整个链条的执行。

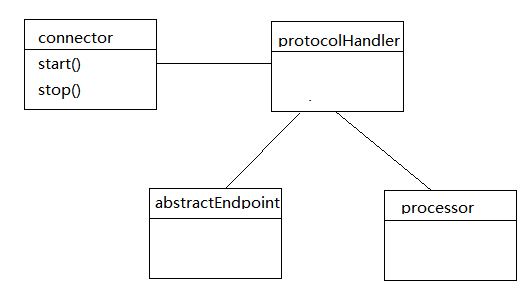
Tomcat的每个层级的容器（Engine、host、context、wrapper）均通过pipeline和valve进行请求的处理，这样的设计使得tomcat工作起来非常的灵活。这也使得我们可以在任何的层级上对请求处理进行扩展。

Connector设计

connector至少要完成以下职能：

1. 监听服务器端口，读取来自客户端的请求
2. 将请求数据按照指定的协议进行解析
3. 根据请求的url地址匹配正确的容器进行处理
4. 将响应返回客户端

Tomcat支持很多协议，默认支持HTTP和AJP，同时tomcat还支持多种I/O方式，包括BIO、NIO、APR。connector的设计如下：



在tomcat中，protocolhandler表示一个协议处理器，针对不同的协议和I/O方式，提供了不同的实现，protocolhandler包含一个endpoint用于启动socket监听，该接口按照I/O分类方式进行分类实现；processor用于按照指定协议读取数据，并将请求交由容器/engine处理。

当connector启动时，endpoint会启动线程来监听服务器端口，并在接收到请求后调用processor进行数据读取。

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两个类来实现上述功能。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LifecyleListener，用于在容器组件状态变更时注册或者取消对应容器的映射信息。

Tomcat通过适配器模式实现了connector与Mapper、Conainer的解耦。

Tomcat的默认connector实现为Coyote

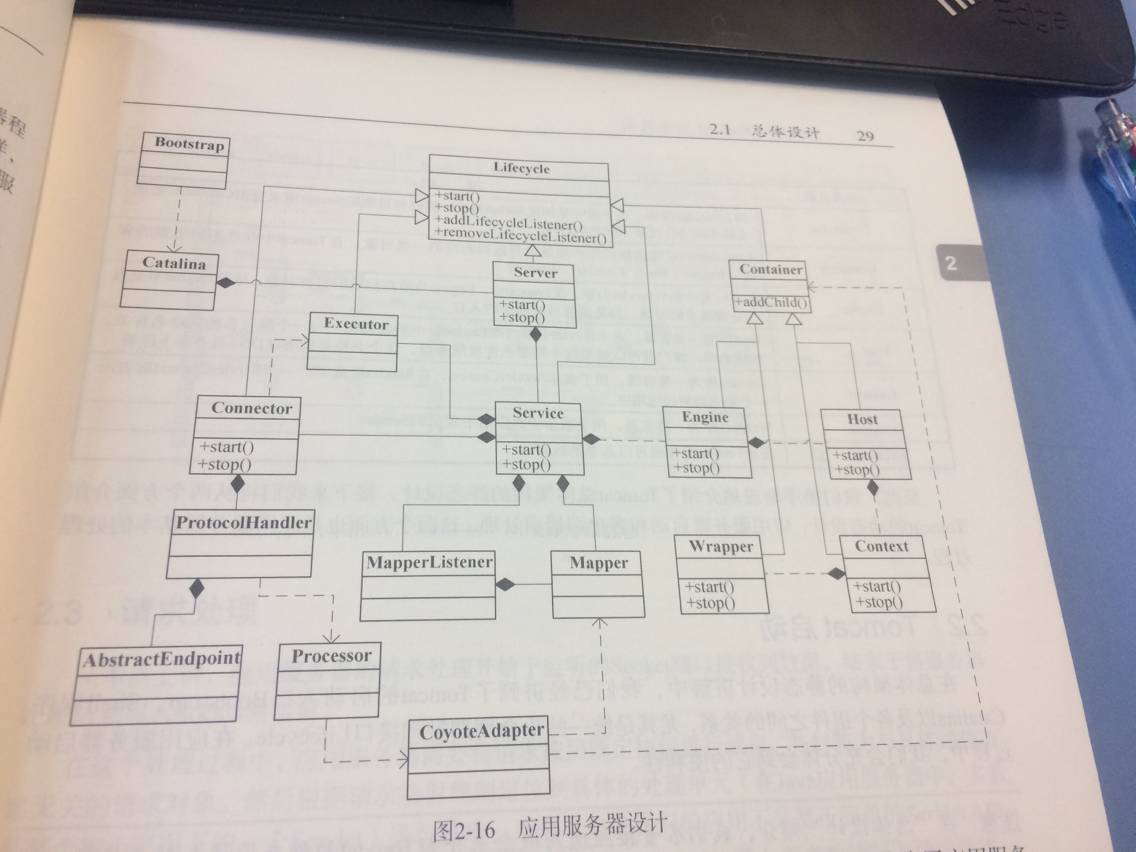
Executor

Tomcat提供了Executor接口来表示一个可以在组件间共享的线程池。该组件同样继承自Lifecyle，以实现可插拔的组件管理。在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。如果没有在Service中定义线程池，相关组件（如Endpoint）会自动创建线程池，此时线程池不在共享。

在tomcat中，Endpoint会启动一组线程来监听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。

Tomcat通过类Catalina提供了一个shell程序，用于解析server.xml创建各个组件，同时负责启动、停止应用服务器（只需要启动tomcat顶层组件server即可）。Tomcat使用Digester解析XML文件，tomcat提供了Bootstrap作为应用服务器启动入口。Bootstrap负责创建Catalina实例。

Tomcat的整体架构：



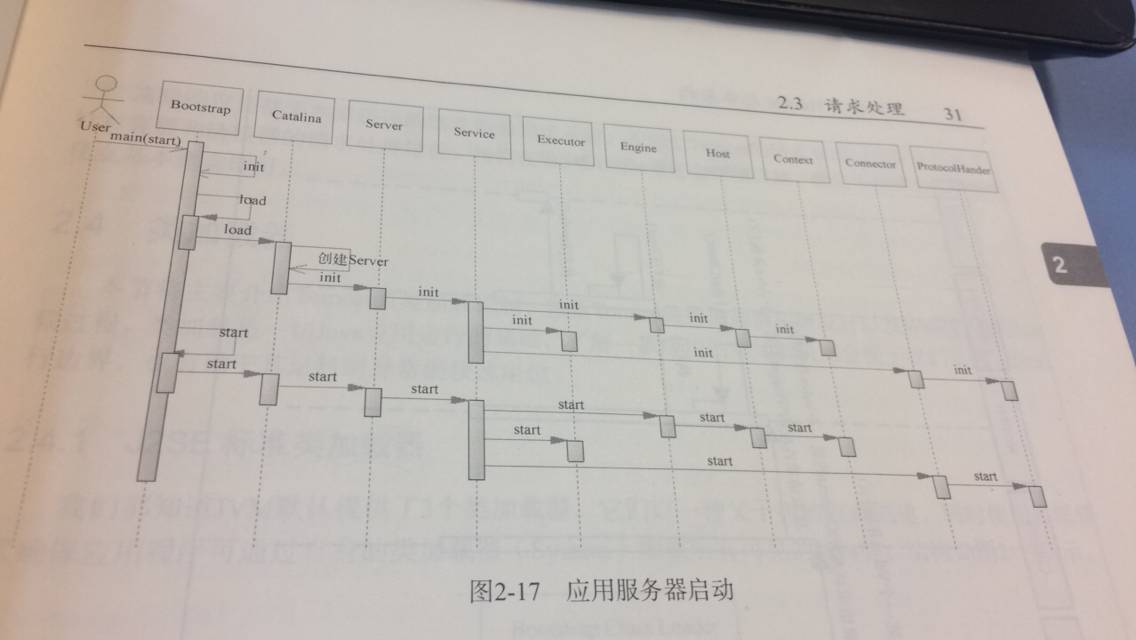
Tomcat组件说明

|  |  |
| --- | --- |
| 组件名称 | 说明 |
| Server | 表示整个Servlet容器，因此Tomcat运行环境中只有唯一一个Server实例 |
| Service | Service表示一个或者多个Connector的集合，这些Connector共享同一个Container来处理其请求，在同一个Tomcat实例内可以包含任意多个Service实例，它们彼此独立 |
| Connector | 即tomcat链接器，用于监听并转化Socket请求，同时将读取的Socket请求交由Container处理，支持不同协议以及不同的IO方式 |
| Container | 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 |
| Engine | Engine表示整个Servlet引擎，在Tomcat中，engine为最高层级的容器对象，尽管engine不是直接处理请求的容器，确实获取目标容器的入口 |
| Host | Host作为一类容器，表示Servlet引擎(即Engine)中的虚拟机。与一个服务器的网络名有关，如域名等。客户端可以使用这个网络名连接服务器，这个名称必须要在DNS服务器上注册。 |
| Context | Context作为一类容器，用于表示ServletContext，在Servlet规范中，一个ServletContext即表示一个独立的Web应用 |
| Wrapper | Wrapper作为一类容器，用于表示web应用定义的Servlet |
| Executor | 表示tomcat组件间可以共享的线程池 |

Tomcat的启动

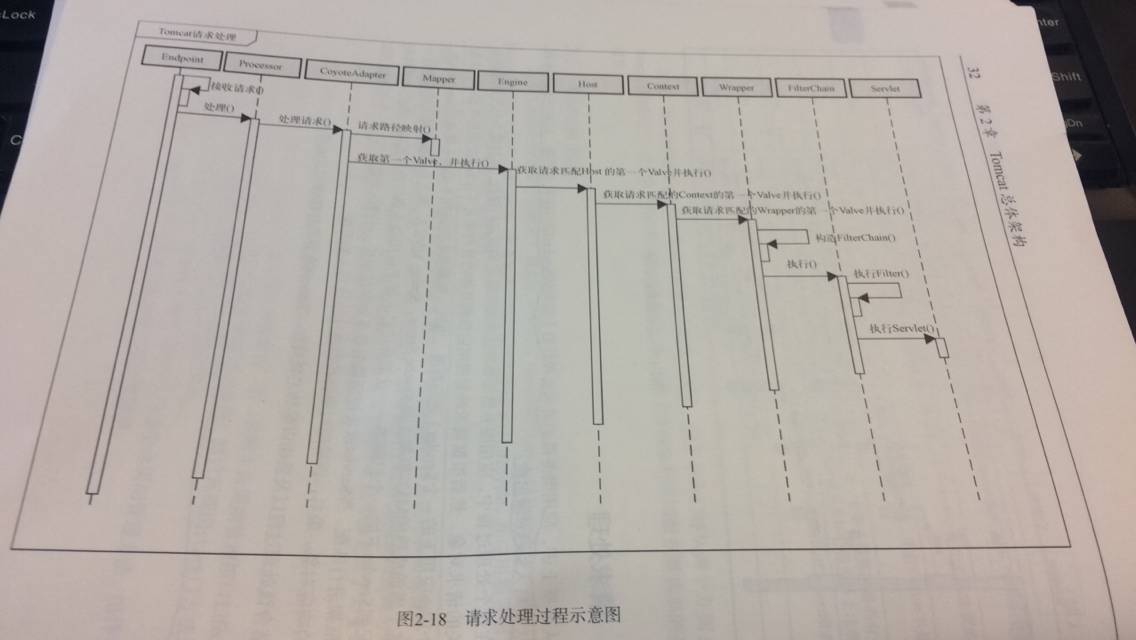
首先调用init方法进行组件的逐级初始化，然后在调用start方法进行启动，当然，每次调用均伴随着生命周期状态变更事件的触发。

应用服务器启动：



从本质上讲，应用服务器的请求处理开始于监听的Socket端口接收到数据，结束于将服务器处理结果写入Socket输出流。

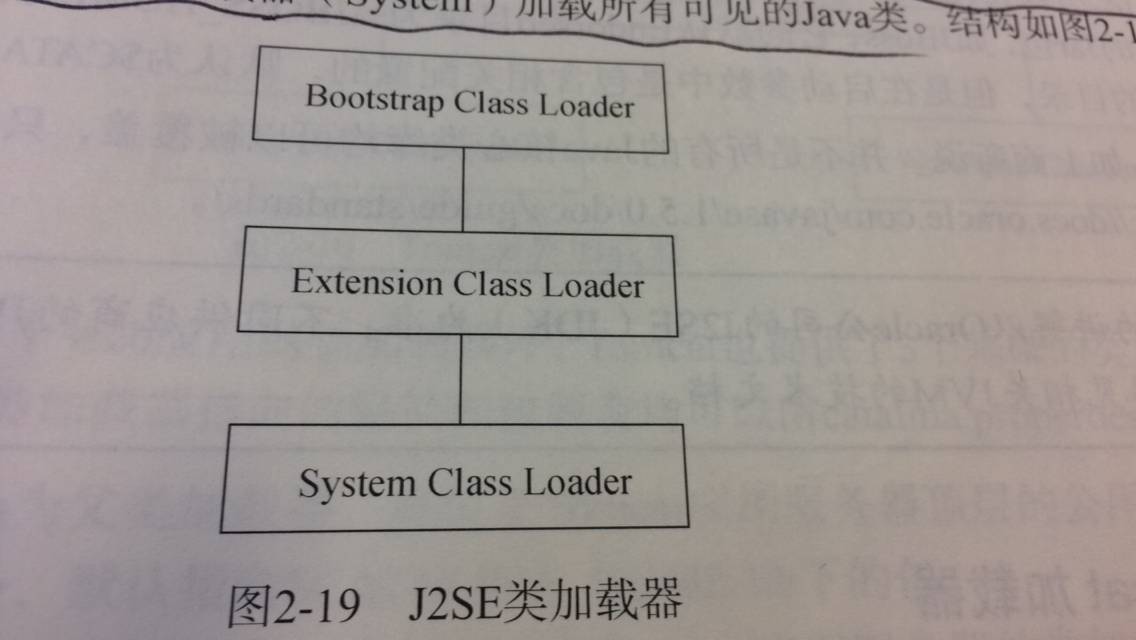
Tomcat请求处理示意图：



类加载器

J2SE标准类加载器

Jvm默认提供了3个类加载器，它们以一种父子树的方式创建，同时使用委派模式确保应用服务器可通过自身的类加载器（System）加载所有可见的java类。



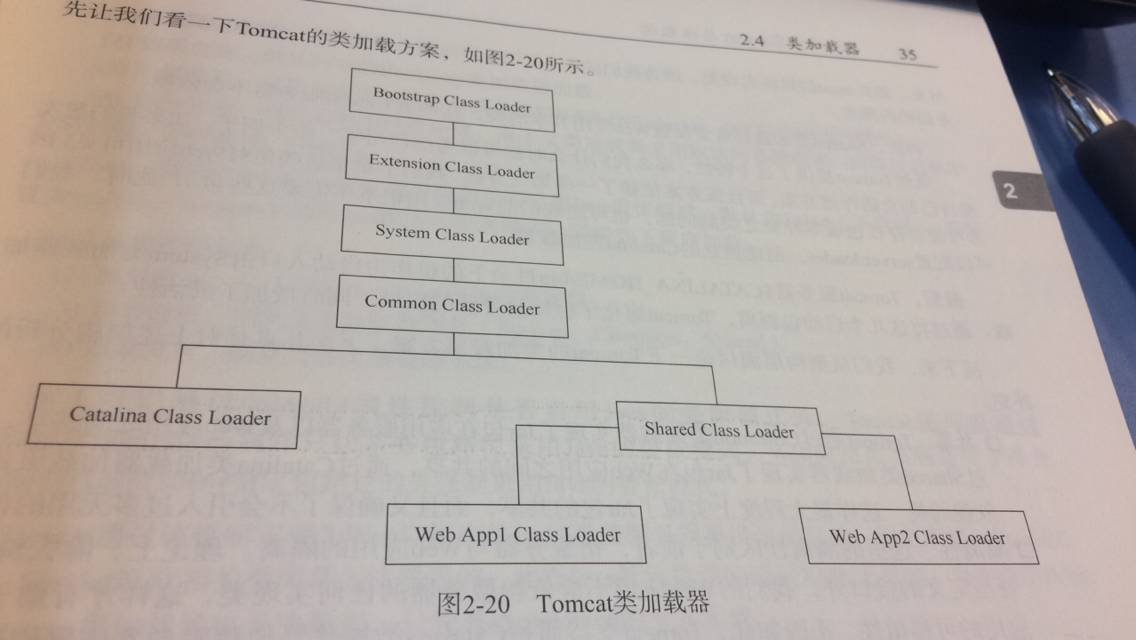
Bootstrap：用于加载jvm提供的基础运行类，即位于%JAVA\_HOME%/jre/lib目录下的核心类库

Extension:Java提供的一个标准的扩展机制用于加载除核心库外的jar包，即只要复制到指定的扩展目录（可以多个）下的jar，jvm会自动加载。默认的扩展目录是%JAVA\_HOME%/jre/lib/ext。不推荐将应用程序依赖的类库放置到扩展目录下，因为该目录下的类库对所有基于该JVM运行的应用程序可见。

System：通常用于加载应用程序jar包及其启动入口类（tomcat的bootstrap类即由System类加载器加载）。

应用程序在不自己构造加载器的情况下，使用System作为默认类加载器，如果应用程序自己构造类加载器，基本也是以system作为父类加载器。

Tomcat的类加载器：



Java默认的类加载机制是委派模式，委派的过程如下：

1. 从缓存中加载
2. 如果缓存中没有，则从父类加载器中加载
3. 如果父类加载器没有则从当前类加载器加载
4. 如果没有，则抛出异常