

# Tomcat 专题

## 课程内容

序号	第一天	第二天
1	Tomcat 基础	Web 应用配置
2	Tomcat 架构	Tomcat管理配置
3	Jasper	JVM配置
4	Tomcat 服务器配置	Tomcat集群
5		Tomcat安全
6		Tomcat性能调优
7		Tomcat 附件功能

## 1.Tomcat 基础

### 1.1 web 概念

### 1) . 软件架构

1. C/S: 客户端/服务器端 -----> QQ , 360 ....

2. B/S: 浏览器/服务器端 -----> 京东, 网易, 淘宝, 传智播客  
官网

### 2) . 资源分类

1. 静态资源: 所有用户访问后, 得到的结果都是一样的, 称为静态资源。静态资源可以直接被浏览器解析。

\* 如: html,css,JavaScript, jpg

2. 动态资源: 每个用户访问相同资源后, 得到的结果可能不一样, 称为动态资源。动态资源被访问后, 需要先转换为静态资源, 再返回给浏览器, 通过浏览器进行解析。

\* 如: servlet/jsp,php,asp....

### 3) . 网络通信三要素

1. IP: 电子设备(计算机)在网络中的唯一标识。

2. 端口: 应用程序在计算机中的唯一标识。 0~65536

3. 传输协议: 规定了数据传输的规则

1. 基础协议:

1. tcp : 安全协议, 三次握手。 速度稍慢

2. udp: 不安全协议。 速度快

## 1.2 常见的web服务器

### 1.2.1 概念

- 1) . 服务器：安装了服务器软件的计算机
- 2) . 服务器软件：接收用户的请求，处理请求，做出响应
- 3) . web服务器软件：接收用户的请求，处理请求，做出响应。

在web服务器软件中，可以部署web项目，让用户通过浏览器来访问这些项目

## 1.2.2 常见web服务器软件

- 1). webLogic: oracle公司，大型的JavaEE服务器，支持所有的JavaEE规范，收费的。
- 2). webSphere: IBM公司，大型的JavaEE服务器，支持所有的JavaEE规范，收费的。
- 3). JBOSS: JBOSS公司的，大型的JavaEE服务器，支持所有的JavaEE规范，收费的。
- 4). Tomcat: Apache基金组织，中小型的JavaEE服务器，仅仅支持少量的JavaEE规范 servlet/jsp。开源的，免费的。

## 1.3 Tomcat 历史

1) Tomcat 最初由Sun公司的软件架构师 James Duncan Davidson 开发，名称为“JavaWebServer”。

2) 1999年，在 Davidson 的帮助下，该项目于1999年于apache 软件基金会旗下的JServ 项目合并，并发布第一个版本（3.x），即是现在的Tomcat，该版本实现了Servlet2.2 和 JSP 1.1 规范。


3) 2001年，Tomcat 发布了4.0版本，作为里程碑式的版本，Tomcat 完全重新设计了其架构，并实现了 Servlet 2.3 和 JSP1.2规范。

目前 Tomcat 已经更新到 9.0.x版本，但是目前企业中的Tomcat服务器，主流版本还是7.x 和 8.x，所以本课程是基于 8.5 版本进行讲解。

## 1.4 Tomcat 安装

### 1.4.1 下载

<https://tomcat.apache.org/download-80.cgi>

 apache-tomcat-8.5.42-windows-x64.zip

## 1.4.2 安装

将下载的 .zip 压缩包，解压到系统的目录（建议是没有中文不带空格的目录）下即可。

## 1.5 Tomcat 目录结构

Tomcat 的主要目录文件如下：

目录	目录下文件	说明
<b>bin</b>	/	存放Tomcat的启动、停止等批处理脚本文件
	startup.bat , startup.sh	用于在windows和linux下的启动脚本
	shutdown.bat , shutdown.sh	用于在windows和linux下的停止脚本
<b>conf</b>	/	用于存放Tomcat的相关配置文件
	Catalina	用于存储针对每个虚拟机的Context配置
	context.xml	用于定义所有web应用均需加载的Context配置，如果web应用指定了自己的context.xml，该文件将被覆盖
	catalina.properties	Tomcat 的环境变量配置
	catalina.policy	Tomcat 运行的安全策略配置
	logging.properties	Tomcat 的日志配置文件，可以通过该文件修改Tomcat 的日志级别及日志路径等
	server.xml	Tomcat 服务器的核心配置文件
	tomcat-users.xml	定义Tomcat默认的用户及角色映射信息配置
	web.xml	Tomcat 中所有应用默认的部署描述文件，主要定义了基础Servlet和MIME映射。
<b>lib</b>	/	Tomcat 服务器的依赖包
<b>logs</b>	/	Tomcat 默认的日志存放目录
<b>webapps</b>	/	Tomcat 默认的Web应用部署目录
<b>work</b>	/	Web 应用JSP代码生成和编译的临时目录

## 1.6 Tomcat 启动停止

### 启动

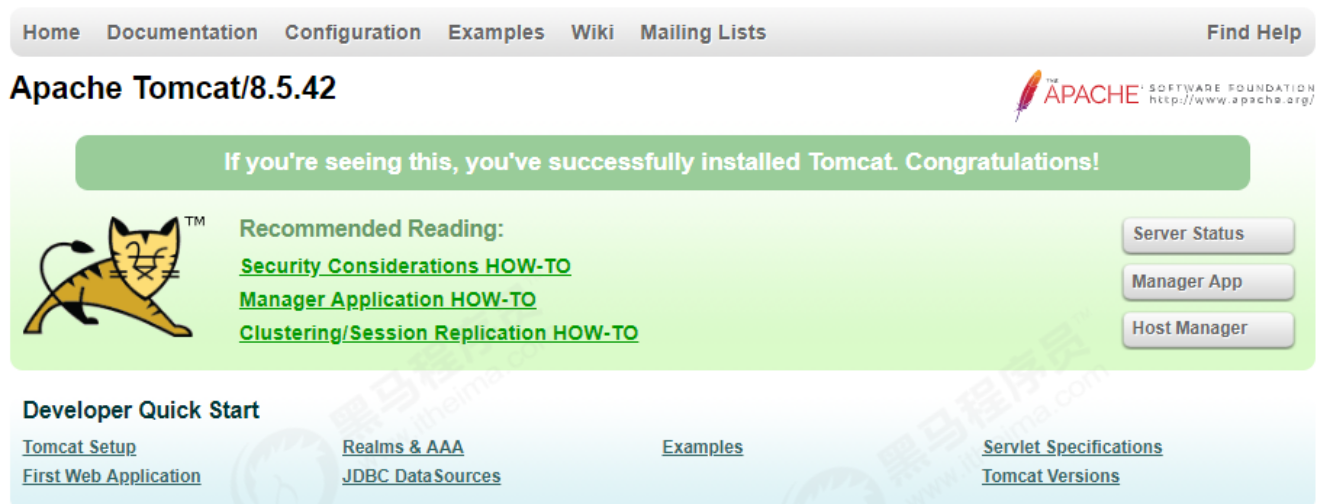
双击 bin/startup.bat 文件；

停止

双击 bin/shutdown.bat 文件；

访问


http://localhost:8080



## 1.7 Tomcat源码

### 1.7.1 下载

地址：<https://tomcat.apache.org/download-80.cgi>

 apache-tomcat-8.5.42-src.zip

### 1.7.2 运行

- 1) 解压zip压缩包
- 2) 进入解压目录，并创建一个目录，命名为home，并将conf、webapps目录移入home目录中
- 3) 在当前目录下创建一个pom.xml文件，引入tomcat的依赖包



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

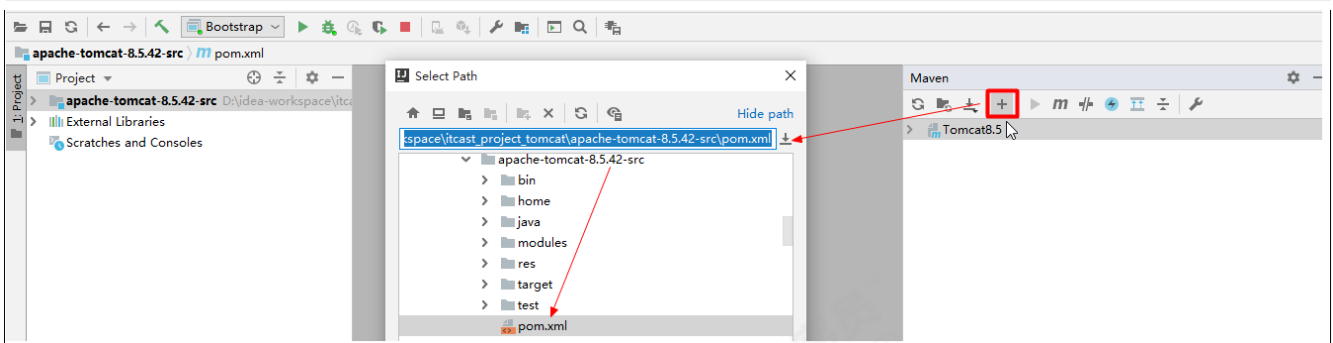
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>apache-tomcat-8.5.42-src</artifactId>
  <name>Tomcat8.5</name>
  <version>8.5</version>

  <build>
    <finalName>Tomcat8.5</finalName>
    <sourceDirectory>java</sourceDirectory>
    <!-- <testSourceDirectory>test</testSourceDirectory>-->
    <resources>
      <resource>
        <directory>java</directory>
      </resource>
    </resources>
    <!-- <testResources>
      <testResource>
        <directory>test</directory>
      </testResource>
    </testResources>-->
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3</version>
        <configuration>
          <encoding>UTF-8</encoding>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.4</version>
  </dependency>
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant</artifactId>
    <version>1.7.0</version>
  </dependency>
  <dependency>
    <groupId>wsdl4j</groupId>
    <artifactId>wsdl4j</artifactId>
    <version>1.6.2</version>
  </dependency>
  <dependency>
    <groupId>javax.xml</groupId>
    <artifactId>jaxrpc</artifactId>
    <version>1.1</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.jdt.core.compiler</groupId>
    <artifactId>ecj</artifactId>
    <version>4.5.1</version>
  </dependency>
</dependencies>
</project>
```

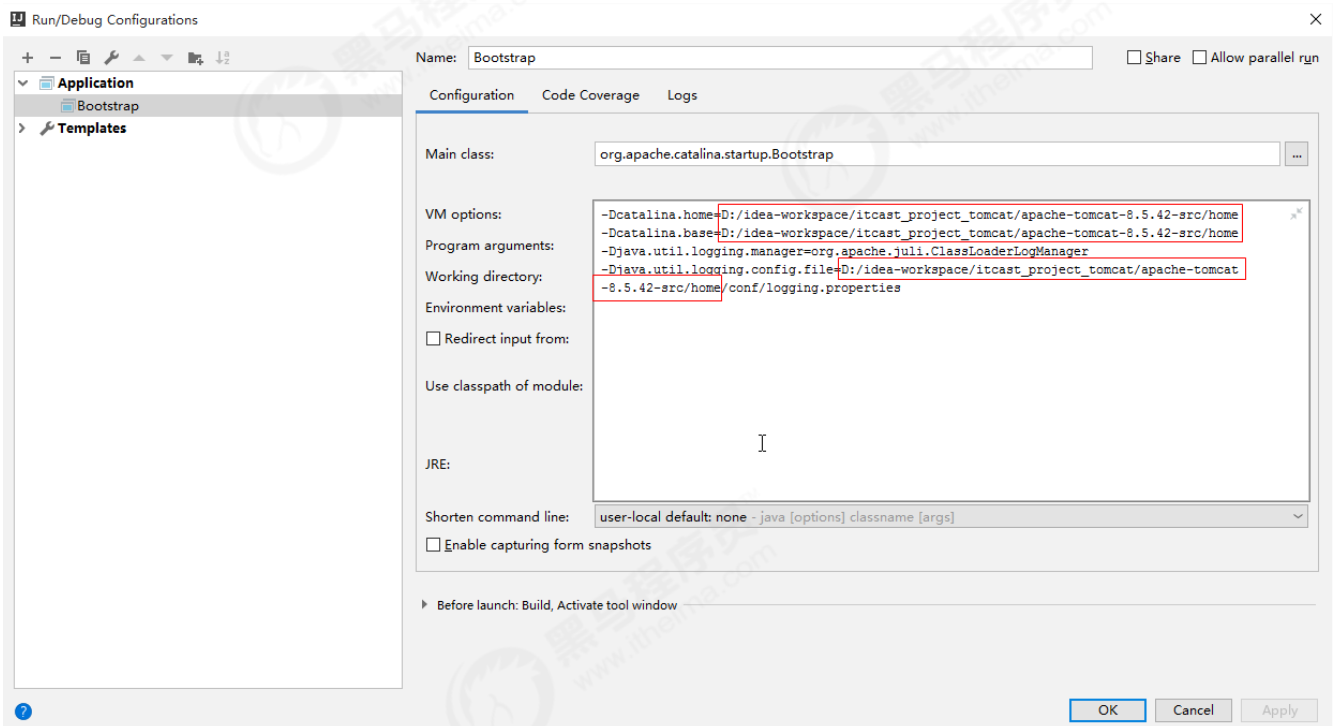
4) 在idea中， 导入该工程。





5) 配置idea的启动类， 配置 MainClass ， 并配置 VM 参数。

```
-Dcatalina.home=D:/idea-workspace/itcast_project_tomcat/apache-tomcat-8.5.42-src/home
-Dcatalina.base=D:/idea-workspace/itcast_project_tomcat/apache-tomcat-8.5.42-src/home
-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Djava.util.logging.config.file=D:/idea-workspace/itcast_project_tomcat/apache-tomcat-8.5.42-src/home/conf/logging.properties
```



6) 启动主方法， 运行Tomcat ， 访问Tomcat 。



## HTTP Status 500 – Internal Server Error

### Type Exception Report

Message java.lang.NullPointerException

Description The server encountered an unexpected condition that prevented it from fulfilling the request.

### Exception

```
org.apache.jasper.JasperException: java.lang.NullPointerException
    org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:598)
    org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:514)
    org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:386)
    org.apache.jasper.servlet.JspServlet.service(JspServlet.java:330)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
```

### Root Cause

```
java.lang.NullPointerException
    org.apache.jsp.index_jsp._jspService(index_jsp.java:424)
    org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:70)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
    org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:476)
    org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:386)
    org.apache.jasper.servlet.JspServlet.service(JspServlet.java:330)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:741)
```

Note The full stack trace of the root cause is available in the server logs.

出现上述异常的原因，是我们直接启动org.apache.catalina.startup.Bootstrap的时候没有加载JasperInitializer，从而无法编译JSP。解决办法是在tomcat的源码ContextConfig中的configureStart函数中手动将JSP解析器初始化：

```
context.addServletContainerInitializer(new JasperInitializer(), null);
```

```
if (log.isDebugEnabled()) {
    log.debug(sm.getString( key: "contextConfig.xmlSettings",
        context.getName(),
        Boolean.valueOf(context.getXmlValidation()),
        Boolean.valueOf(context.getXmlNamespaceAware()) ));
}

webConfig();
context.addServletContainerInitializer(new JasperInitializer(), classes: null);

if (!context.getIgnoreAnnotations()) {
    applicationAnnotationsConfig();
}

if (ok) {
    validateSecurityRoles();
}
```


7) 重启tomcat就可以正常访问了。

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/@VERSION@

THE APACHE SOFTWARE FOUNDATION  
http://www.apache.org/

If you're seeing this, you've successfully installed Tomcat. Congratulations!



Recommended Reading:  
[Security Considerations HOW-TO](#)  
[Manager Application HOW-TO](#)  
[Clustering/Session Replication HOW-TO](#)

Server Status  
Manager App  
Host Manager

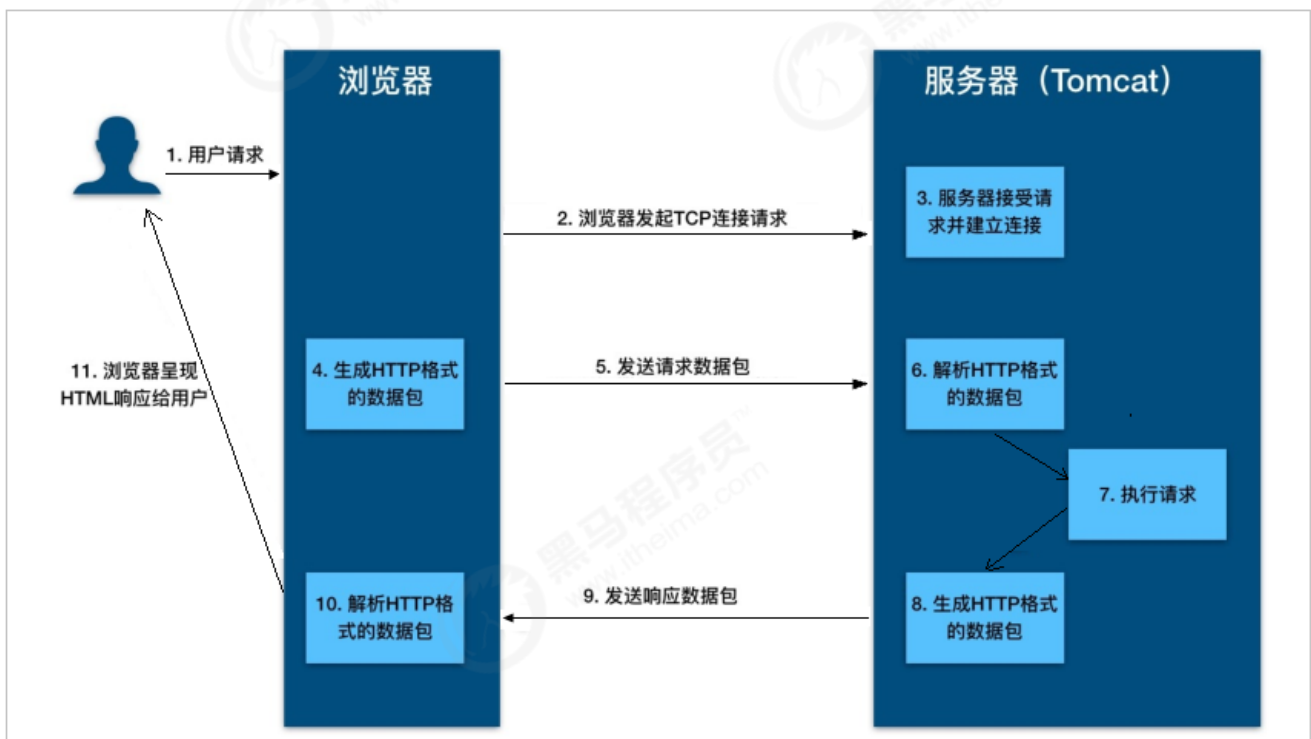
Developer Quick Start

[Tomcat Setup](#)  
[First Web Application](#)  
[Realms & AAA](#)  
[JDBC DataSources](#)  
[Examples](#)  
[Servlet Specifications](#)  
[Tomcat Versions](#)

## 2.Tomcat 架构

### 2.1 Http工作原理

HTTP协议是浏览器与服务器之间的数据传送协议。作为应用层协议，HTTP是基于TCP/IP协议来传递数据的（HTML文件、图片、查询结果等），HTTP协议不涉及数据包（Packet）传输，主要规定了客户端和服务端之间的通信格式。



从图上你可以看到，这个过程是：

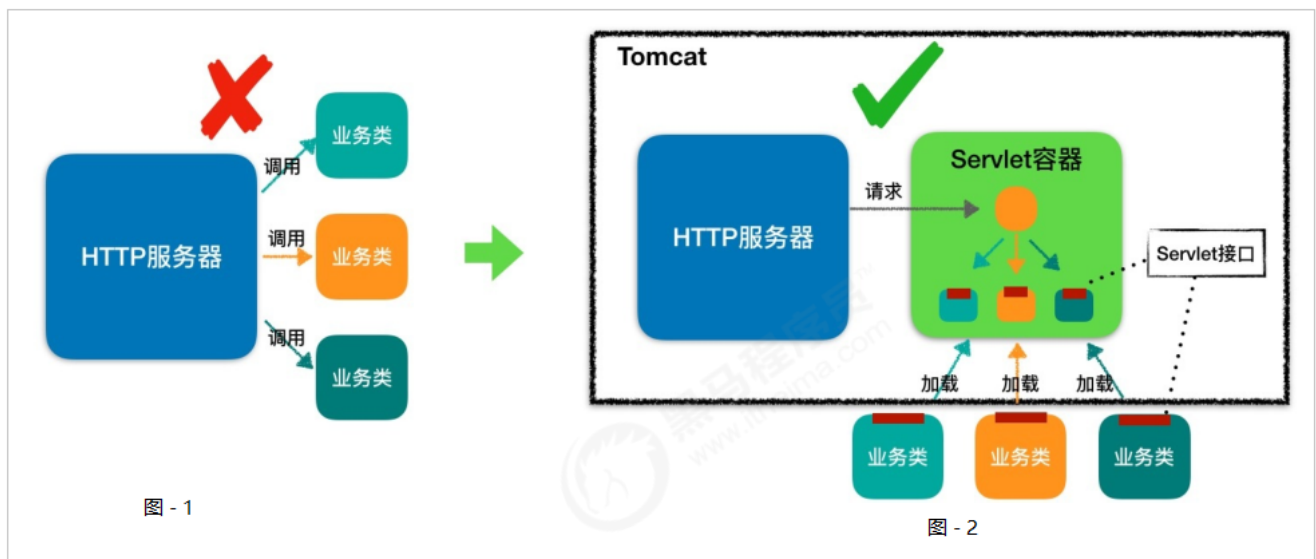
- 1) 用户通过浏览器进行了一个操作，比如输入网址并回车，或者是点击链接，接着浏览器获取了这个事件。
- 2) 浏览器向服务端发出TCP连接请求。
- 3) 服务程序接受浏览器的连接请求，并经过TCP三次握手建立连接。
- 4) 浏览器将请求数据打包成一个HTTP协议格式的数据包。
- 5) 浏览器将该数据包推入网络，数据包经过网络传输，最终达到端服务程序。
- 6) 服务端程序拿到这个数据包后，同样以HTTP协议格式解包，获取到客户端的意图。
- 7) 得知客户端意图后进行处理，比如提供静态文件或者调用服务端程序获得动态结果。
- 8) 服务器将响应结果（可能是HTML或者图片等）按照HTTP协议格式打包。
- 9) 服务器将响应数据包推入网络，数据包经过网络传输最终达到到浏览器。
- 10) 浏览器拿到数据包后，以HTTP协议的格式解包，然后解析数据，假设这里的数据是HTML。
- 11) 浏览器将HTML文件展示在页面上。

那我们想要探究的Tomcat作为一个HTTP服务器，在这个过程中都做了些什么事情呢？主要是接受连接、解析请求数据、处理请求和发送响应这几个步骤。

## 2.2 Tomcat整体架构

### 2.2.1 Http服务器请求处理

浏览器发给服务端的是一个HTTP格式的请求，HTTP服务器收到这个请求后，需要调用服务端程序来处理，所谓的服务端程序就是你写的Java类，一般来说不同的请求需要由不同的Java类来处理。



1) 图1，表示HTTP服务器直接调用具体业务类，它们是紧耦合的。

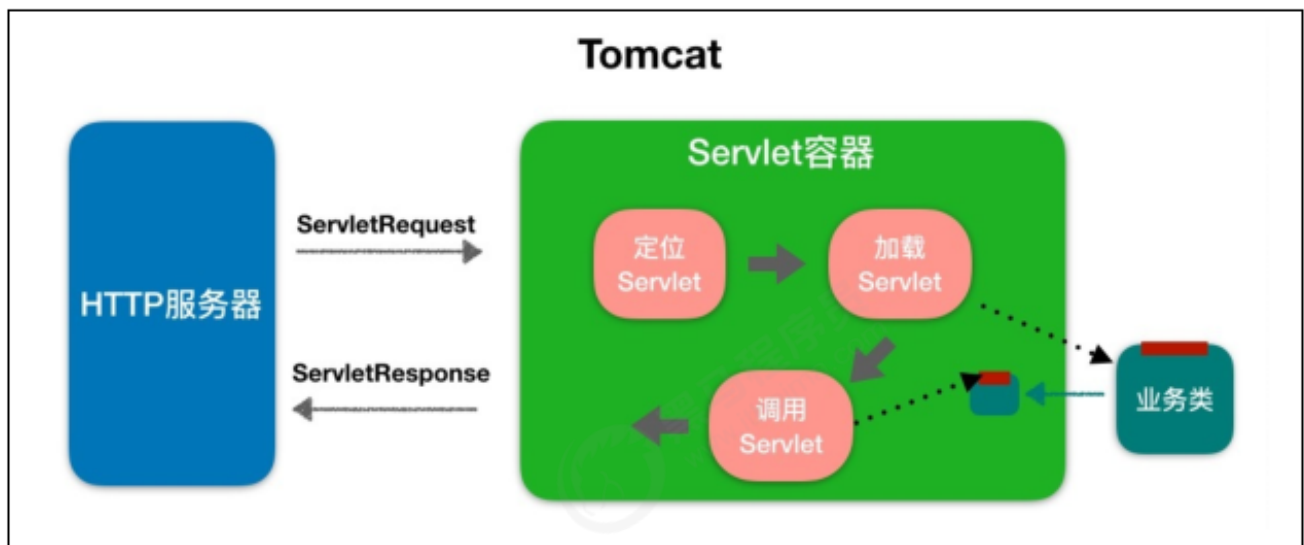
2) 图2，HTTP服务器不直接调用业务类，而是把请求交给容器来处理，容器通过Servlet接口调用业务类。因此Servlet接口和Servlet容器的出现，达到了HTTP服务器与业务类解耦的目的。而Servlet接口和Servlet容器这一整套规范叫作Servlet规范。

Tomcat按照Servlet规范的要求实现了Servlet容器，同时它们也具有HTTP服务器的功能。作为Java程序员，如果我们要实现新的业务功能，只需要实现一个Servlet，并把它注册到Tomcat（Servlet容器）中，剩下的事情就由Tomcat帮我们处理了。

## 2.2.2 Servlet容器工作流程

为了解耦，HTTP服务器不直接调用Servlet，而是把请求交给Servlet容器来处理，那Servlet容器又是怎么工作的呢？

当客户请求某个资源时，HTTP服务器会用一个ServletRequest对象把客户的请求信息封装起来，然后调用Servlet容器的service方法，Servlet容器拿到请求后，根据请求的URL和Servlet的映射关系，找到相应的Servlet，如果Servlet还没有被加载，就用反射机制创建这个Servlet，并调用Servlet的init方法来完成初始化，接着调用Servlet的service方法来处理请求，把ServletResponse对象返回给HTTP服务器，HTTP服务器会把响应发送给客户端。

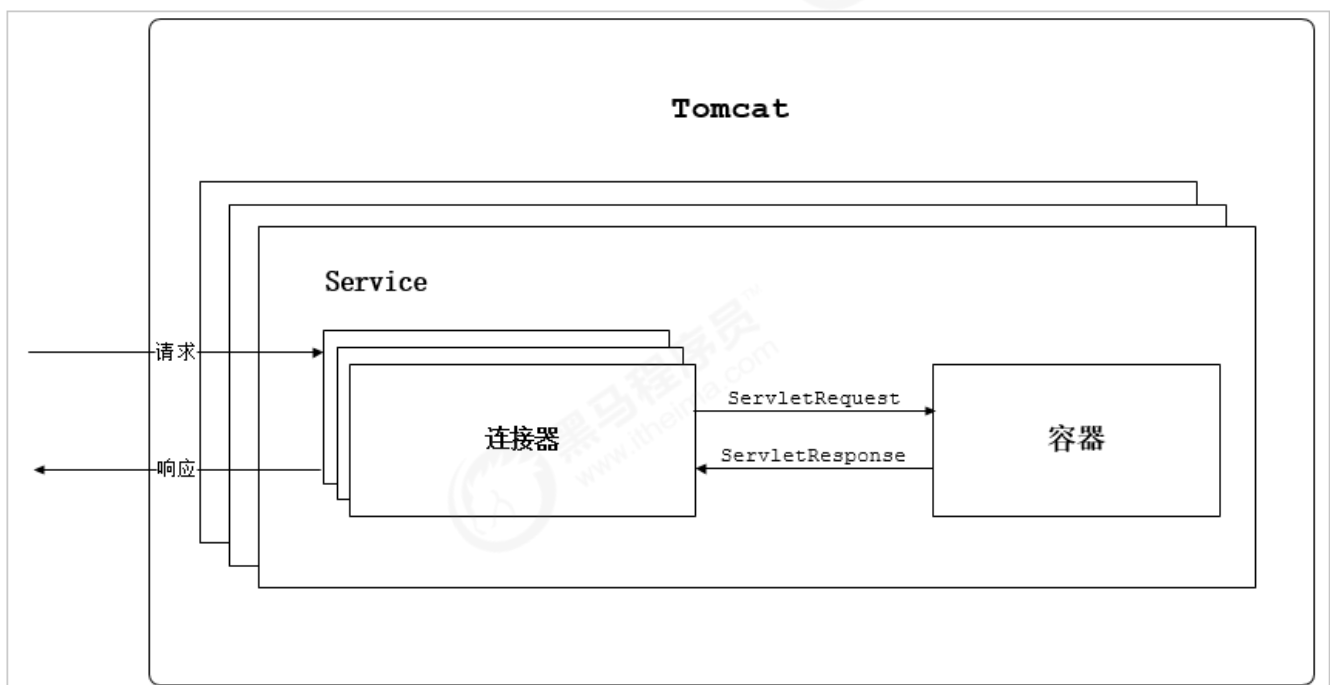


### 2.2.3 Tomcat整体架构

我们知道如果要设计一个系统，首先是要了解需求，我们已经了解了Tomcat要实现两个核心功能：

- 1) 处理Socket连接，负责网络字节流与Request和Response对象的转化。
- 2) 加载和管理Servlet，以及具体处理Request请求。

因此Tomcat设计了两个核心组件连接器（Connector）和容器（Container）来分别做这两件事情。连接器负责对外交流，容器负责内部处理。





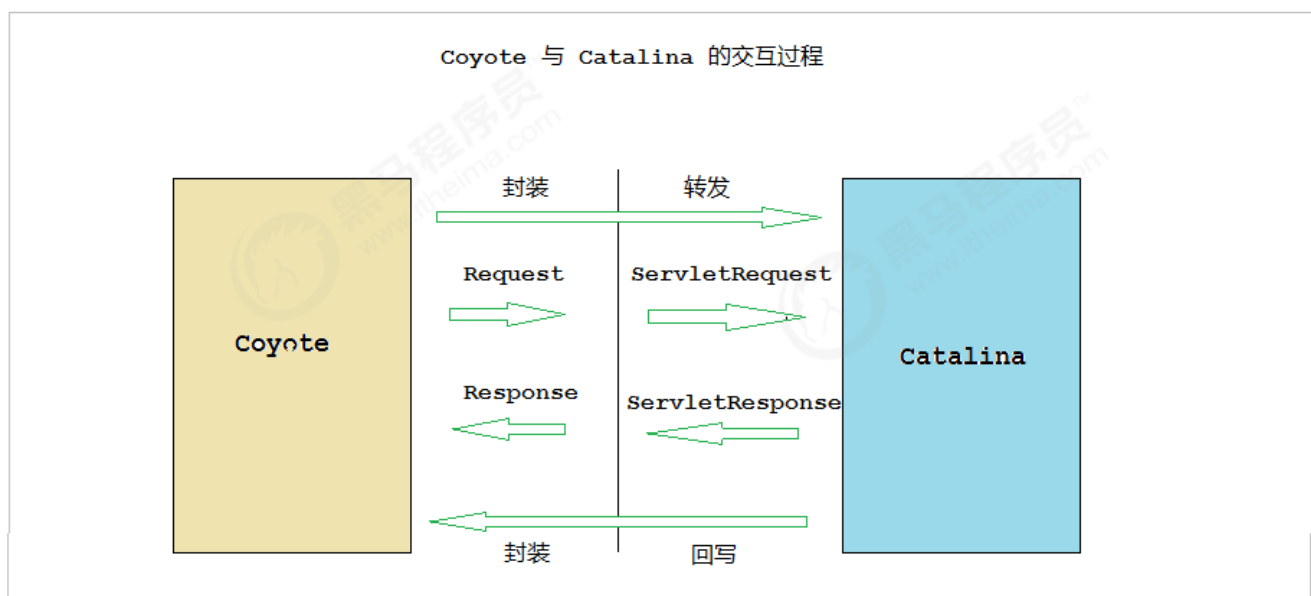
## 2.3 连接器 - Coyote

### 2.3.1 架构介绍

Coyote 是Tomcat的连接框架的名称，是Tomcat服务器提供的供客户端访问的外部接口。客户端通过Coyote与服务器建立连接、发送请求并接受响应。

Coyote 封装了底层的网络通信（Socket 请求及响应处理），为Catalina 容器提供了统一的接口，使Catalina 容器与具体的请求协议及IO操作方式完全解耦。Coyote 将Socket 输入转换封装为 Request 对象，交由Catalina 容器进行处理，处理请求完成后，Catalina 通过Coyote 提供的Response 对象将结果写入输出流。

Coyote 作为独立的模块，只负责具体协议和IO的相关操作，与Servlet 规范实现没有直接关系，因此即便是 Request 和 Response 对象也并未实现Servlet规范对应的接口，而是在Catalina 中将他们进一步封装为ServletRequest 和 ServletResponse。



### 2.3.2 IO模型与协议

在Coyote中，Tomcat支持的多种I/O模型和应用层协议，具体包含哪些IO模型和应用层协议，请看下表：

Tomcat 支持的IO模型（自8.5/9.0 版本起，Tomcat 移除了 对 BIO 的支持）：

IO模型	描述
NIO	非阻塞I/O，采用Java NIO类库实现。
NIO2	异步I/O，采用JDK 7最新的NIO2类库实现。
APR	采用Apache可移植运行库实现，是C/C++编写的本地库。如果选择该方案，需要单独安装APR库。

Tomcat 支持的应用层协议：

应用层协议	描述
HTTP/1.1	这是大部分Web应用采用的访问协议。
AJP	用于和Web服务器集成（如Apache），以实现静态资源的优化以及集群部署，当前支持AJP/1.3。
HTTP/2	HTTP 2.0大幅度的提升了Web性能。下一代HTTP协议，自8.5以及9.0版本之后支持。

协议分层：

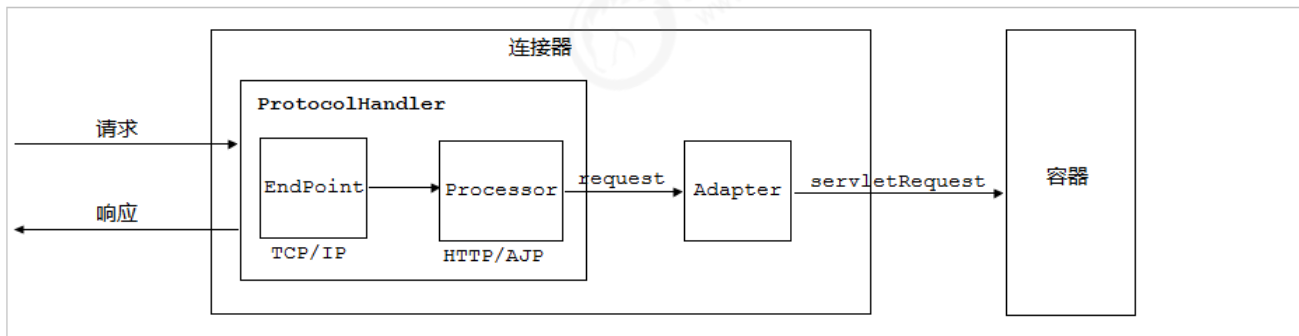
应用层	HTTP	AJP	HTTP2
	( Processor )		
传输层	NIO	NIO2	APR
	( Endpoint )		

在 8.0 之前，Tomcat 默认采用的I/O方式为 BIO，之后改为 NIO。无论 NIO、NIO2 还是 APR，在性能方面均优于以往的BIO。如果采用APR，甚至可以达到 Apache HTTP Server 的影响性能。



Tomcat为了实现支持多种I/O模型和应用层协议，一个容器可能对接多个连接器，就好比一个房间有多个门。但是单独的连接器和容器都不能对外提供服务，需要把它们组装起来才能工作，组装后这个整体叫作Service组件。这里请你注意，Service本身没有做什么重要的事情，只是在连接器和容器外面多包了一层，把它们组装在一起。Tomcat内可能有多个Service，这样的设计也是出于灵活性的考虑。通过在Tomcat中配置多个Service，可以实现通过不同的端口号来访问同一台机器上部署的不同应用。

### 2.3.3 连接器组件



连接器中的各个组件的作用如下：

#### EndPoint

1) EndPoint：Coyote 通信端点，即通信监听的接口，是具体Socket接收和发送处理器，是对传输层的抽象，因此EndPoint用来实现TCP/IP协议的。

2) Tomcat 并没有EndPoint 接口，而是提供了一个抽象类AbstractEndpoint，里面定义了两个内部类：Acceptor和SocketProcessor。Acceptor用于监听Socket连接请求。SocketProcessor用于处理接收到的Socket请求，它实现Runnable接口，在Run方法里调用协议处理组件Processor进行处理。为了提高处理能力，SocketProcessor被提交到线程池来执行。而这个线程池叫作执行器（Executor），我在后面的专栏会详细介绍Tomcat如何扩展原生的Java线程池。

#### Processor

Processor：Coyote 协议处理接口，如果说EndPoint是用来实现TCP/IP协议的，那么Processor用来实现HTTP协议，Processor接收来自EndPoint的Socket，读取字节流解析成Tomcat Request和Response对象，并通过Adapter将其提交到容器处理，Processor是对应用层协议的抽象。

#### ProtocolHandler

**ProtocolHandler:** Coyote 协议接口，通过Endpoint 和 Processor，实现针对具体协议的处理能力。Tomcat 按照协议和I/O 提供了6个实现类：AjpNioProtocol，AjpAprProtocol，AjpNio2Protocol，Http11NioProtocol，Http11Nio2Protocol，Http11AprProtocol。我们在配置tomcat/conf/server.xml 时，至少要指定具体的ProtocolHandler，当然也可以指定协议名称，如：HTTP/1.1，如果安装了APR，那么将使用Http11AprProtocol，否则使用 Http11NioProtocol。

## Adapter

由于协议不同，客户端发过来的请求信息也不尽相同，Tomcat定义了自己的Request类来“存放”这些请求信息。ProtocolHandler接口负责解析请求并生成Tomcat Request类。但是这个Request对象不是标准的ServletRequest，也就意味着，不能用Tomcat Request作为参数来调用容器。Tomcat设计者的解决方案是引入CoyoteAdapter，这是适配器模式的经典运用，连接器调用CoyoteAdapter的Service方法，传入的是Tomcat Request对象，CoyoteAdapter负责将Tomcat Request转成ServletRequest，再调用容器的Service方法。

### 2.3.4 源码解析

具体的源码解析，请参考2.5，2.6 章节讲解的Tomcat启动流程及请求处理流程

## 2.4 容器 - Catalina

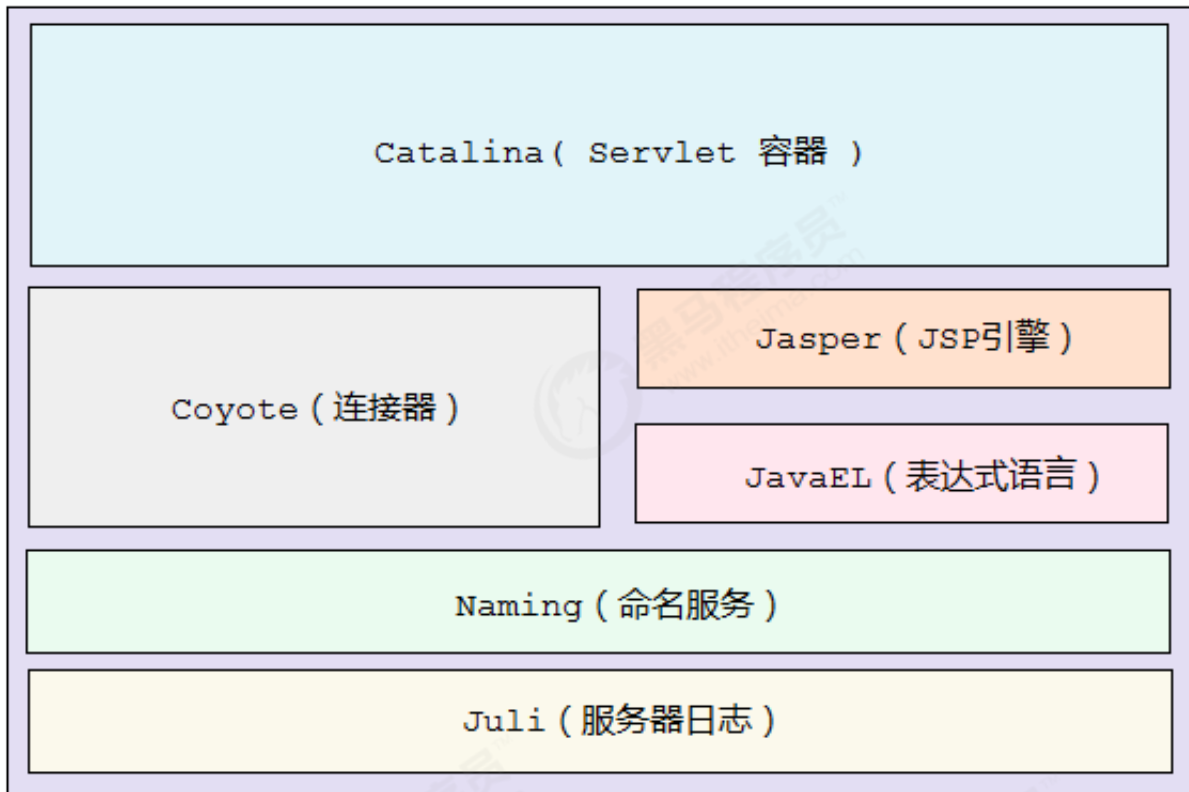
Tomcat是一个由一系列可配置的组件构成的Web容器，而Catalina是Tomcat的servlet容器。

Catalina 是Servlet 容器实现，包含了之前讲到的所有的容器组件，以及后续章节涉及到的安全、会话、集群、管理等Servlet 容器架构的各个方面。它通过松耦合的方式集成Coyote，以完成按照请求协议进行数据读写。同时，它还包括我们的启动入口、Shell程序等。

### 2.4.1 Catalina 地位

Tomcat 的模块分层结构图，如下：

Tomcat 模块分层示意图

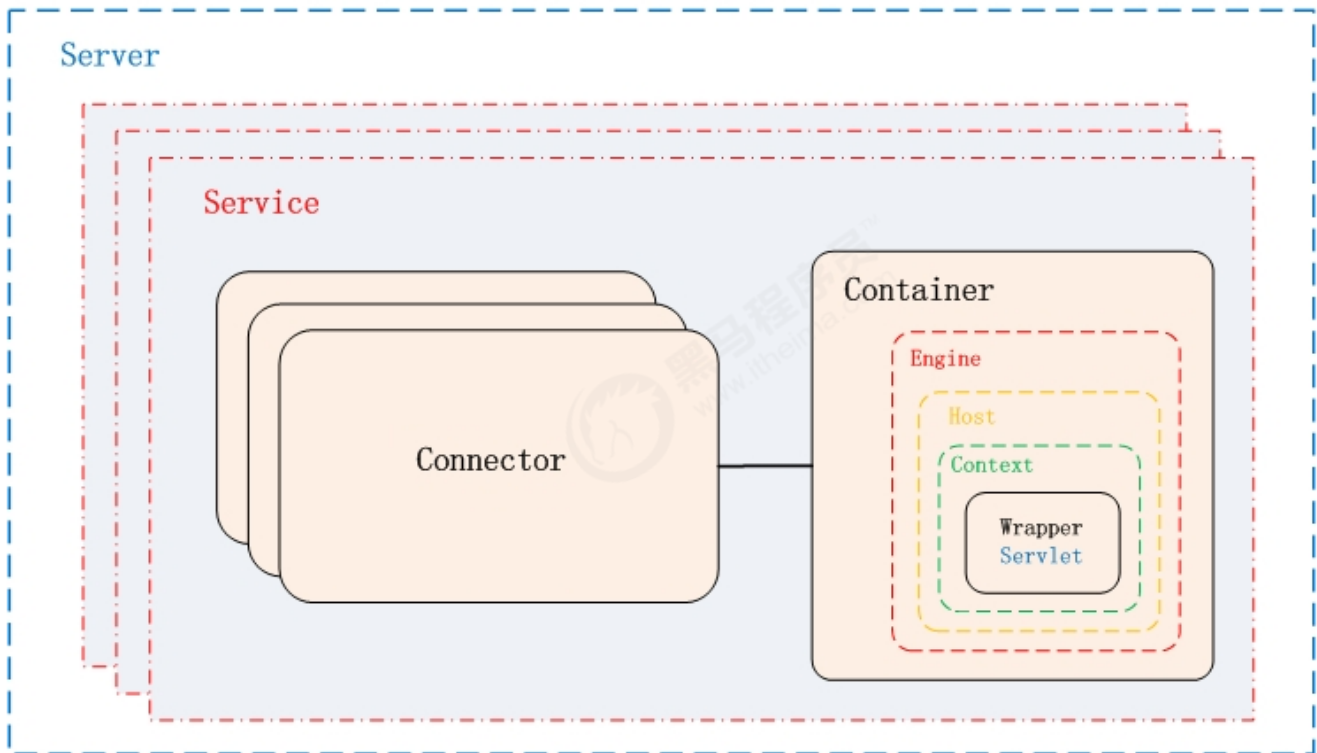


Tomcat 本质上就是一款 Servlet 容器，因此 Catalina 才是 Tomcat 的核心，其他模块都是为 Catalina 提供支撑的。比如：通过 Coyote 模块提供链接通信，Jasper 模块提供 JSP 引擎，Naming 提供 JNDI 服务，Juli 提供日志服务。

## 2.4.2 Catalina 结构

Catalina 的主要组件结构如下：

## Catalina



如上图所示，Catalina负责管理Server，而Server表示着整个服务器。Server下面有多个服务Service，每个服务都包含着多个连接器组件Connector（Coyote 实现）和一个容器组件Container。在Tomcat 启动的时候，会初始化一个Catalina的实例。

Catalina 各个组件的职责：

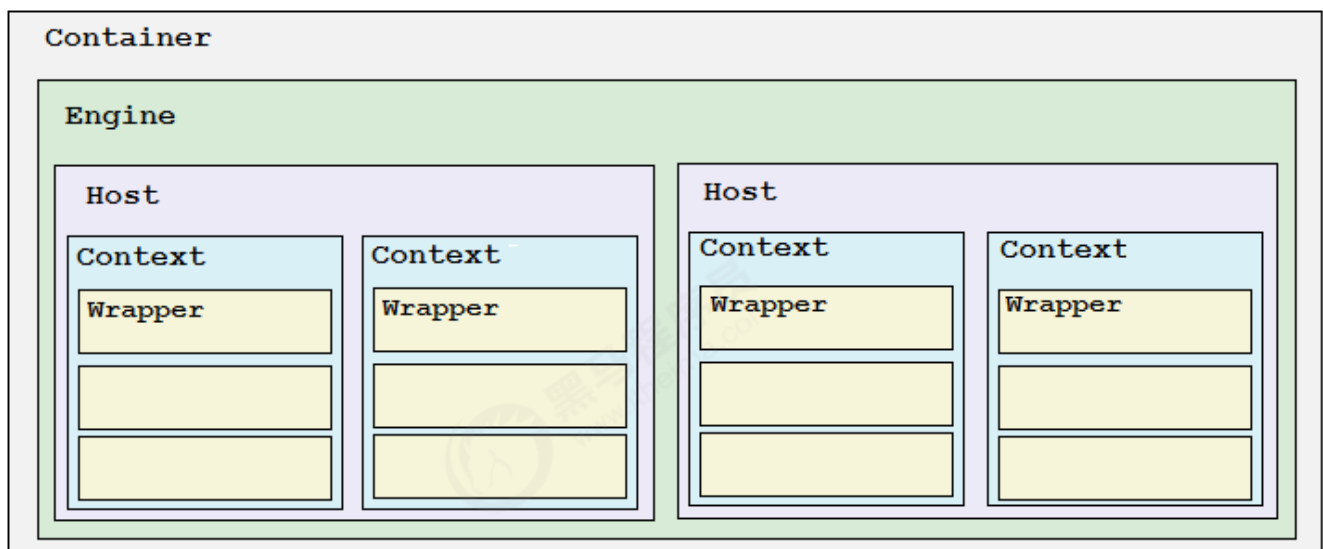
组件	职责
Catalina	负责解析Tomcat的配置文件，以此来创建服务器Server组件，并根据命令来对其进行管理
Server	服务器表示整个Catalina Servlet容器以及其它组件，负责组装并启动Servlet引擎,Tomcat连接器。Server通过实现Lifecycle接口，提供了一种优雅的启动和关闭整个系统的方式
Service	服务是Server内部的组件，一个Server包含多个Service。它将若干个Connector组件绑定到一个Container（Engine）上
Connector	连接器，处理与客户端的通信，它负责接收客户请求，然后转给相关的容器处理，最后向客户返回响应结果
Container	容器，负责处理用户的servlet请求，并返回对象给web用户的模块

Service

- addConnector(Connector): void
- addExecutor(Executor): void
- findConnectors(): Connector[]
- findExecutors(): Executor[]
- getContainer(): Engine
- getDomain(): String
- getExecutor(String): Executor
- getMapper(): Mapper
- getName(): String
- getParentClassLoader(): ClassLoader
- getServer(): Server
- removeConnector(Connector): void
- removeExecutor(Executor): void
- setContainer(Engine): void
- setName(String): void
- setParentClassLoader(ClassLoader): void
- setServer(Server): void

### 2.4.3 Container 结构

Tomcat设计了4种容器，分别是Engine、Host、Context和Wrapper。这4种容器不是平行关系，而是父子关系。，Tomcat通过一种分层的架构，使得Servlet容器具有很好的灵活性。



各个组件的含义：

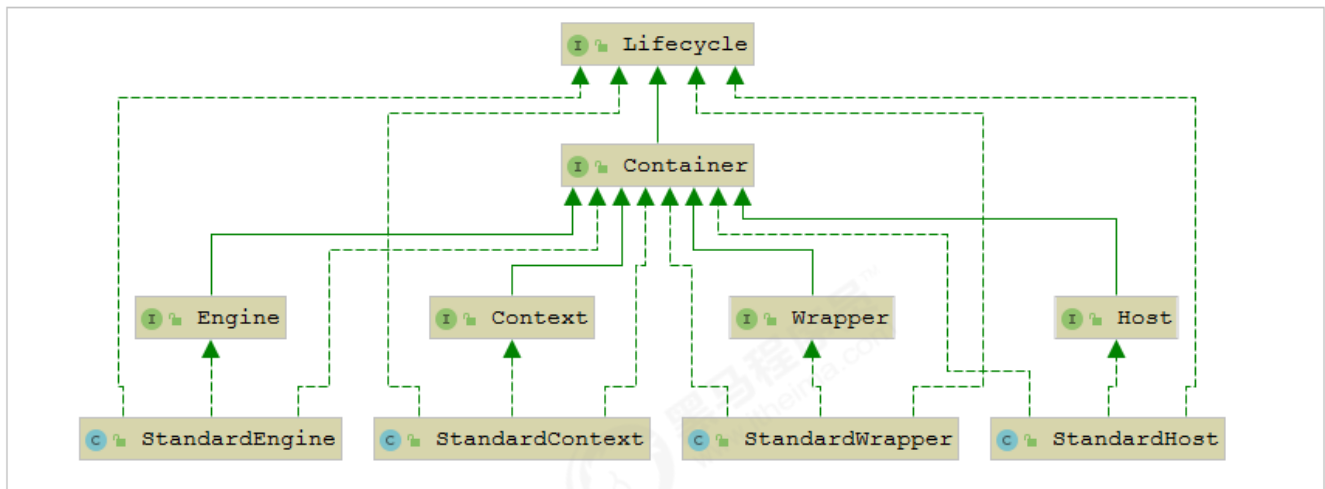
容器	描述
Engine	表示整个Catalina的Servlet引擎，用来管理多个虚拟站点，一个Service最多只能有一个Engine，但是一个引擎可包含多个Host
Host	代表一个虚拟主机，或者说一个站点，可以给Tomcat配置多个虚拟主机地址，而一个虚拟主机下可包含多个Context
Context	表示一个Web应用程序，一个Web应用可包含多个Wrapper
Wrapper	表示一个Servlet，Wrapper 作为容器中的最底层，不能包含子容器

我们也可以再通过Tomcat的server.xml配置文件来加深对Tomcat容器的理解。Tomcat采用了组件化的设计，它的构成组件都是可配置的，其中最外层的是Server，其他组件按照一定的格式要求配置在这个顶层容器中。

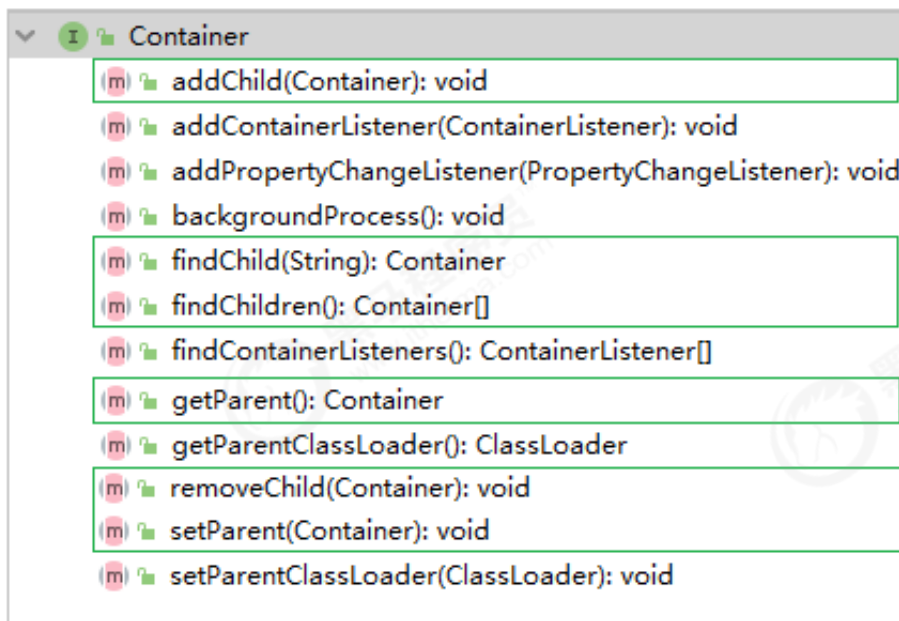
```
<Server>
  <Service>
    <Connector/>
    <Connector/>
    <Engine>
      <Host>
        <Context></Context>
      </Host>
    </Engine>
  </Service>
</Server>
```

那么，Tomcat是怎么管理这些容器的呢？你会发现这些容器具有父子关系，形成一个树形结构，你可能马上就想到了设计模式中的组合模式。没错，Tomcat就是用组合模式来管理这些容器的。具体实现方法是，所有容器组件都实现了Container接口，因此组合模式可以使得用户对单容器对象和组合容器对象的使用具有一致性。这里单容器对象指的是最底层的Wrapper，组合容器对象指的是上面的Context、Host或者Engine。





Container 接口中提供了以下方法（截图中知识一部分方法）：

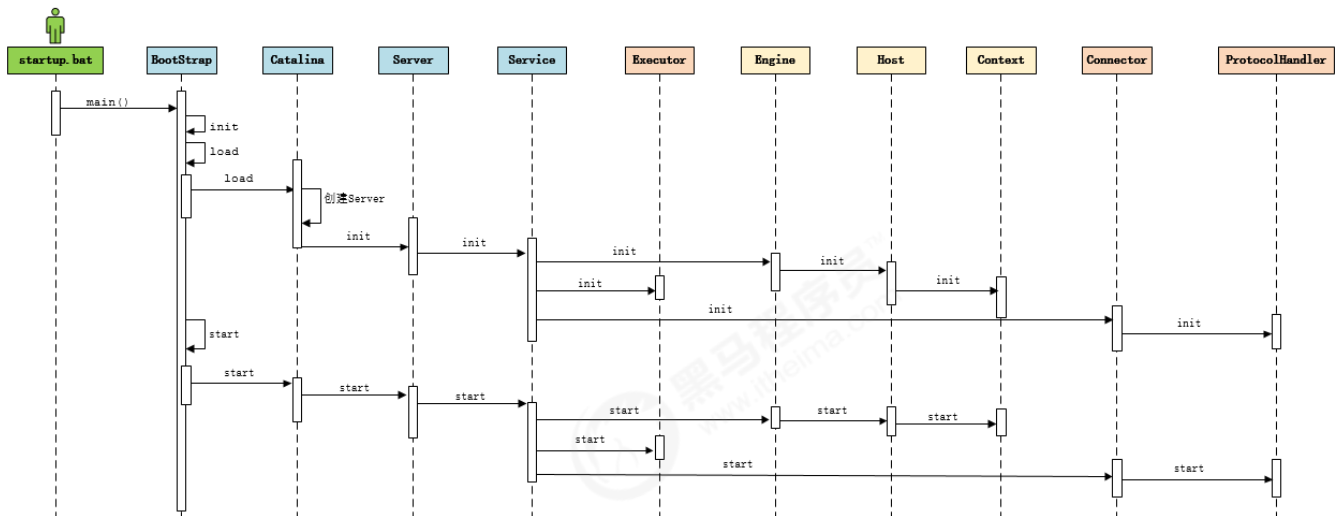


在上面的接口看到了getParent、SetParent、addChild和removeChild等方法。

Container接口扩展了Lifecycle接口，Lifecycle接口用来统一管理各组件的生命周期，后面我也用专门的篇幅去详细介绍。

## 2.5 Tomcat 启动流程

### 2.5.1 流程



步骤：

- 1) 启动tomcat，需要调用 bin/startup.bat (在linux 目录下，需要调用 bin/startup.sh)，在startup.bat 脚本中，调用了catalina.bat。
- 2) 在catalina.bat 脚本文件中，调用了Bootstrap 中的main方法。
- 3) 在Bootstrap 的main 方法中调用了init 方法，来创建Catalina 及 初始化类加载器。
- 4) 在Bootstrap 的main 方法中调用了load 方法，在其中又调用了Catalina的load方法。
- 5) 在Catalina 的load 方法中，需要进行一些初始化的工作，并需要构造Digester 对象，用于解析 XML。
- 6) 然后在调用后续组件的初始化操作。。。

加载Tomcat的配置文件，初始化容器组件，监听对应的端口号，准备接受客户端请求。

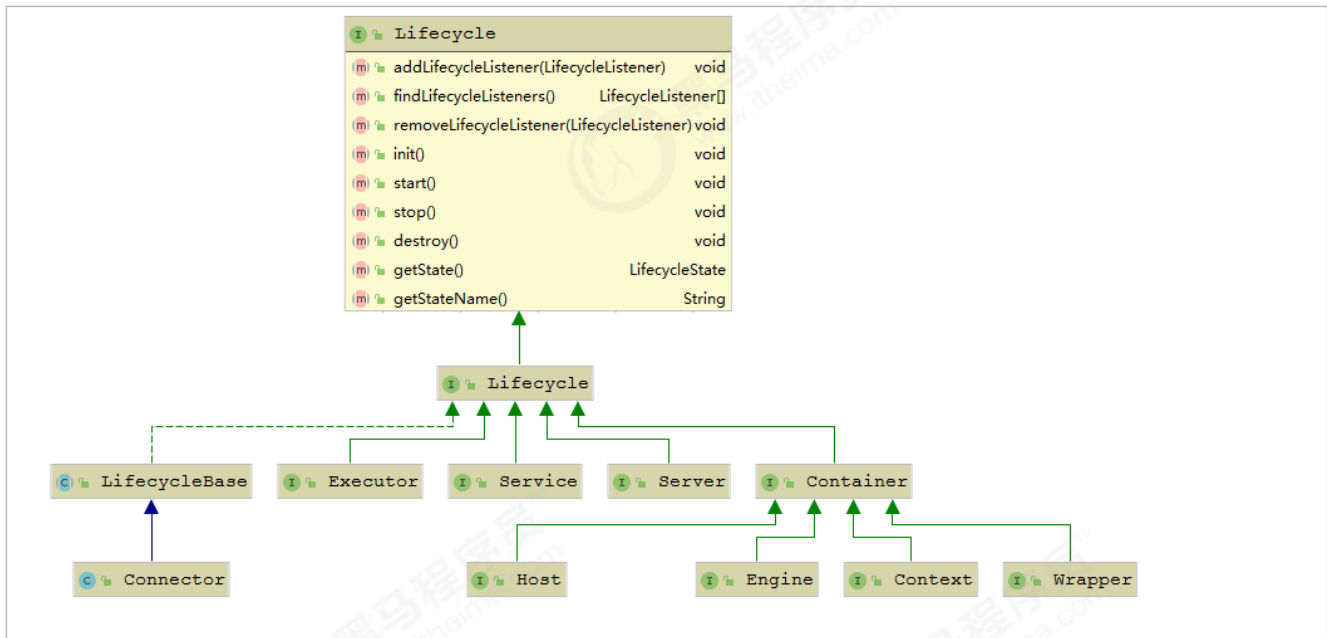
## 2.5.2 源码解析

### 2.5.2.1 Lifecycle

由于所有的组件均存在初始化、启动、停止等生命周期方法，拥有生命周期管理的特性，所以Tomcat在设计的时候，基于生命周期管理抽象成了一个接口 Lifecycle，而组件 Server、Service、Container、Executor、Connector 组件，都实现了一个生命周期的接口，从而具有了以下生命周期中的核心方法：

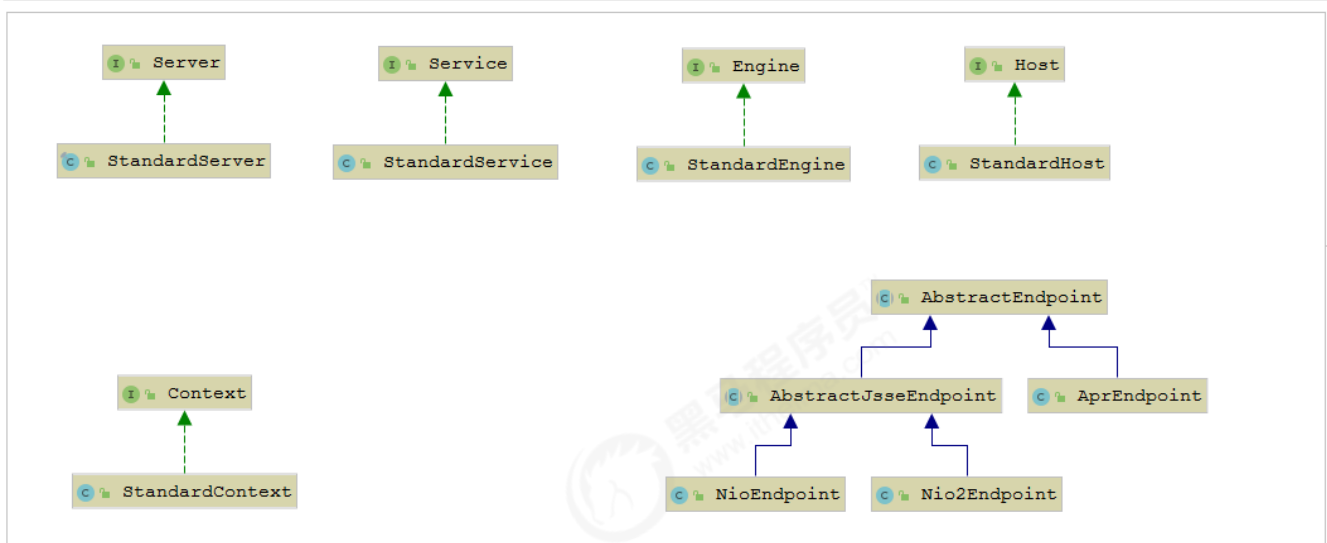


- 1) init () : 初始化组件
- 2) start () : 启动组件
- 3) stop () : 停止组件
- 4) destroy () : 销毁组件



### 2.5.2.2 各组件的默认实现

上面我们提到的Server、Service、Engine、Host、Context都是接口，下图中罗列了这些接口的默认实现类。当前对于Endpoint组件来说，在Tomcat中没有对应的Endpoint接口，但是有一个抽象类AbstractEndpoint，其下有三个实现类：NioEndpoint、Nio2Endpoint、AprEndpoint，这三个实现类，分别对应于前面讲解链接器Coyote时，提到的链接器支持的三种IO模型：NIO，NIO2，APR，Tomcat8.5版本中，默认采用的是NioEndpoint。



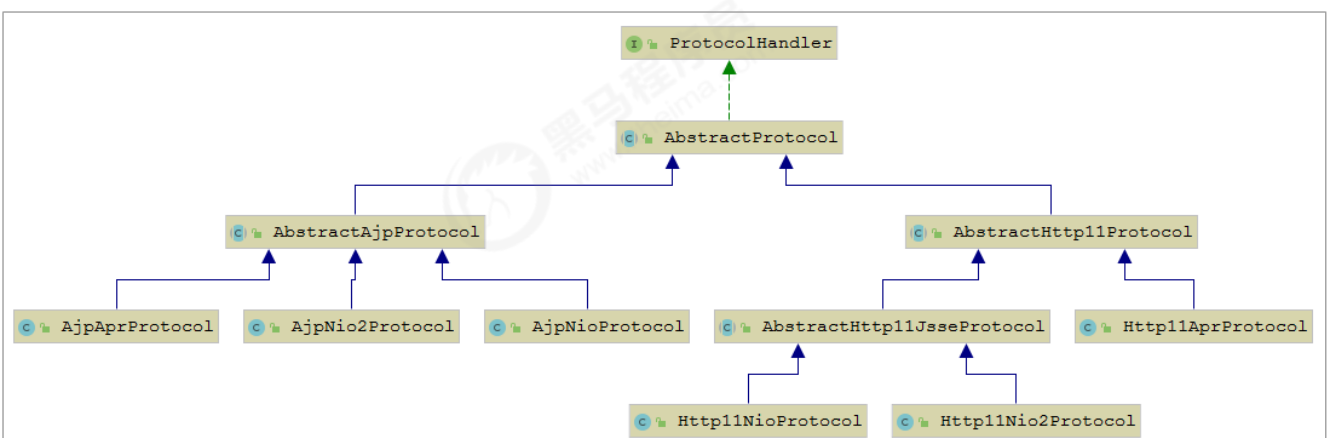
**ProtocolHandler**：Coyote协议接口，通过封装Endpoint和Processor，实现针对具体协议的处理功能。Tomcat按照协议和IO提供了6个实现类。

AJP协议：

- 1) AjpNioProtocol：采用NIO的IO模型。
- 2) AjpNio2Protocol：采用NIO2的IO模型。
- 3) AjpAprProtocol：采用APR的IO模型，需要依赖于APR库。

HTTP协议：

- 1) Http11NioProtocol：采用NIO的IO模型，默认使用的协议（如果服务器没有安装APR）。
- 2) Http11Nio2Protocol：采用NIO2的IO模型。
- 3) Http11AprProtocol：采用APR的IO模型，需要依赖于APR库。



### 2.5.2.3 源码入口

目录: org.apache.catalina.startup

MainClass: Bootstrap ----> main(String[] args)

```
public static void main(String args[]) {  
  
    if (daemon == null) {  
        // Don't set daemon until init() has completed  
        Bootstrap bootstrap = new Bootstrap();  
        try {  
            bootstrap.init();  
        } catch (Throwable t) {  
            handleThrowable(t);  
            t.printStackTrace();  
            return;  
        }  
        daemon = bootstrap;  
    } else {  

```

## 2.5.3 总结

从启动流程图中以及源码中，我们可以看出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操作，然后再调用start()方法进行启动。

每一级的组件除了完成自身的处理外，还要负责调用子组件响应的生命周期管理方法，组件与组件之间是松耦合的，因为我们可以很容易的通过配置文件进行修改和替换。

## 2.6 Tomcat 请求处理流程

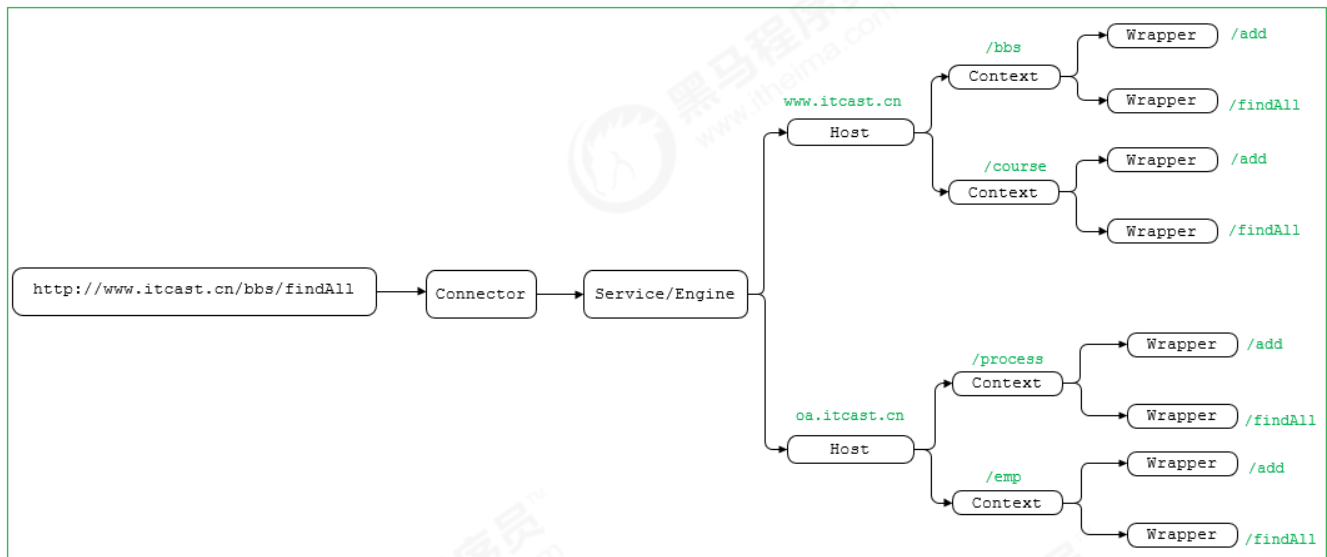
### 2.6.1 请求流程

设计了这么多层次的容器，Tomcat是怎么确定每一个请求应该由哪个Wrapper容器里的Servlet来处理呢？答案是，Tomcat是用Mapper组件来完成这个任务的。

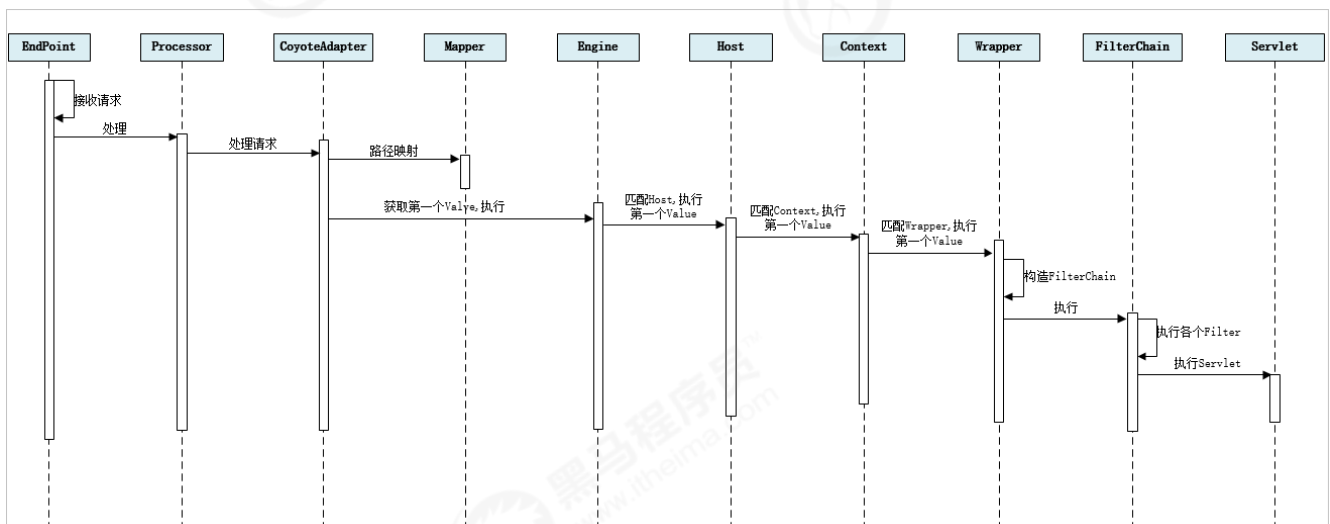
Mapper组件的功能就是将用户请求的URL定位到一个Servlet，它的工作原理是：Mapper组件里保存了Web应用的配置信息，其实就是容器组件与访问路径的映射关系，比如Host容器里配置的域名、Context容器里的Web应用路径，以及Wrapper容器里Servlet映射的路径，你可以想象这些配置信息就是一个多层次的Map。

当一个请求到来时，Mapper组件通过解析请求URL里的域名和路径，再到自己保存的Map里去查找，就能定位到一个Servlet。请你注意，一个请求URL最后只会定位到一个Wrapper容器，也就是一个Servlet。

下面的示意图中，就描述了当用户请求链接 `http://www.itcast.cn/bbs/findAll` 之后，是如何找到最终处理业务逻辑的servlet。



那上面这幅图只是描述了根据请求的URL如何查找到需要执行的Servlet，那么下面我们再解析一下，从Tomcat的设计架构层面来分析Tomcat的请求处理。



步骤如下：

- 1) Connector组件Endpoint中的Acceptor监听客户端套接字连接并接收Socket。
- 2) 将连接交给线程池Executor处理，开始执行请求响应任务。
- 3) Processor组件读取消息报文，解析请求行、请求体、请求头，封装成Request对象。
- 4) Mapper组件根据请求行的URL值和请求头的Host值匹配由哪个Host容器、Context容

器、Wrapper容器处理请求。

5) CoyoteAdaptor组件负责将Connector组件和Engine容器关联起来，把生成的Request对象和响应对象Response传递到Engine容器中，调用 Pipeline。

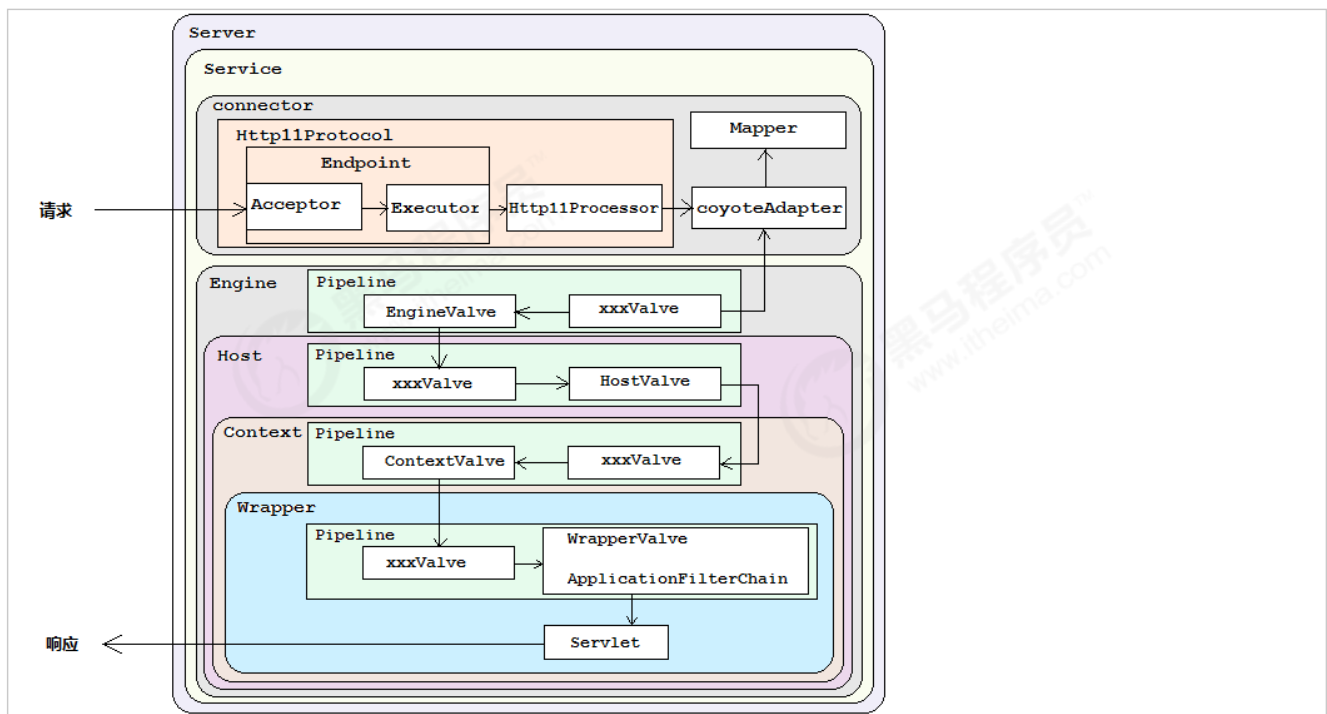
6) Engine容器的管道开始处理，管道中包含若干个Valve、每个Valve负责部分处理逻辑。执行完Valve后会执行基础的 Valve--StandardEngineValve，负责调用Host容器的Pipeline。

7) Host容器的管道开始处理，流程类似，最后执行 Context容器的Pipeline。

8) Context容器的管道开始处理，流程类似，最后执行 Wrapper容器的Pipeline。

9) Wrapper容器的管道开始处理，流程类似，最后执行 Wrapper容器对应的Servlet对象的 处理方法。

## 2.6.2 请求流程源码解析



在前面所讲解的Tomcat的整体架构中，我们发现Tomcat中的各个组件各司其职，组件之间松耦合，确保了整体架构的可伸缩性和可拓展性，那么在组件内部，如何增强组件的灵活性和拓展性呢？在Tomcat中，每个Container组件采用责任链模式来完成具体的请求处理。

在Tomcat中定义了Pipeline 和 Valve 两个接口，Pipeline 用于构建责任链，后者代表责任链上的每个处理器。Pipeline 中维护了一个基础的Valve，它始终位于Pipeline的末端（最后执行），封装了具体的请求处理和输出响应的过程。当然，我们也可以调用 addValve()方法，为Pipeline 添加其他的Valve，后添加的Valve 位于基础的Valve之前，并按照添加顺序执行。Pipeline通过获得首个Valve来启动整合链条的执行。

## 3.Jasper

### 3.1 Jasper 简介

对于基于JSP的web应用来说，我们可以直接在JSP页面中编写Java代码，添加第三方的标签库，以及使用EL表达式。但是无论经过何种形式的处理，最终输出到客户端的都是标准的HTML页面（包含js，css...），并不包含任何的java相关的语法。也就是说，我们可以把jsp看做是一种运行在服务端的脚本。那么服务器是如何将JSP页面转换为HTML页面的呢？

Jasper模块是Tomcat的JSP核心引擎，我们知道JSP本质上是一个Servlet。Tomcat使用Jasper对JSP语法进行解析，生成Servlet并生成Class字节码，用户在进行访问jsp时，会访问Servlet，最终将访问的结果直接响应在浏览器端。另外，在运行的时候，Jasper还会检测JSP文件是否修改，如果修改，则会重新编译JSP文件。

### 3.2 JSP 编译方式

#### 3.2.1 运行时编译

Tomcat并不会在启动Web应用的时候自动编译JSP文件，而是在客户端第一次请求时，才编译需要访问的JSP文件。

创建一个web项目，并编写JSP代码：

```
<%@ page import="java.text.DateFormat" %>
<%@ page import="java.text.SimpleDateFormat" %>
<%@ page import="java.util.Date" %>
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>$Title$</title>
  </head>
  <body>
    <%
      DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
      String format = dateFormat.format(new Date());
    %>
    Hello , Java Server Page . . . .

    <br/>

    <%= format %>

  </body>
</html>
```

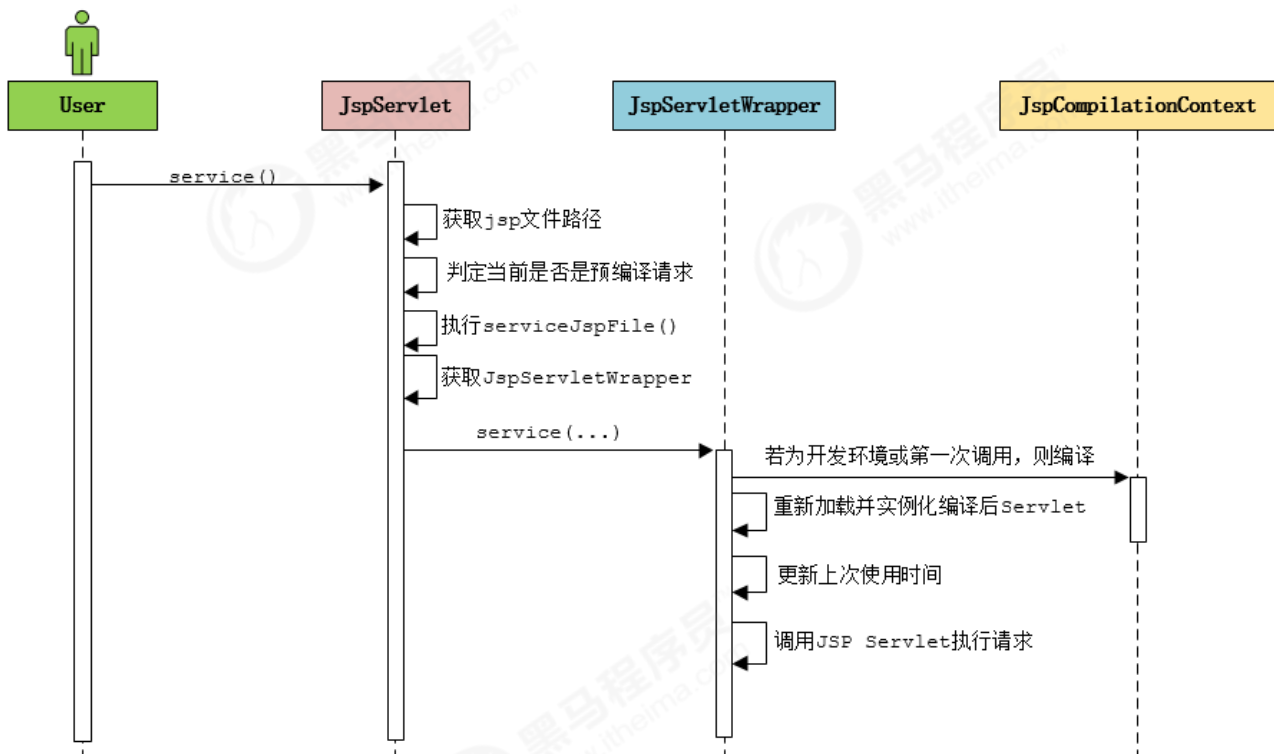
### 3.2.1.1 编译过程

Tomcat 在默认的web.xml 中配置了一个org.apache.jasper.servlet.JspServlet，用于处理所有的.jsp 或 .jspx 结尾的请求，该Servlet 实现即是运行时编译的入口。



```
<servlet>
  <servlet-name>jsp</servlet-name>
  <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
  <init-param>
    <param-name>fork</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>xpoweredBy</param-name>
    <param-value>>false</param-value>
  </init-param>
  <load-on-startup>3</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

JspServlet 处理流程图：



### 3.2.2 编译结果

1) 如果在 tomcat/conf/web.xml 中配置了参数 scratchdir，则jsp编译后的结果，就会存储在该目录下。



```
<init-param>
    <param-name>scratchdir</param-name>
    <param-value>D:/tmp/jsp/</param-value>
</init-param>
```

2) 如果没有配置该选项，则会将编译后的结果，存储在Tomcat安装目录下的work/Catalina(Engine名称)/localhost(Host名称)/Context名称。假设项目名称为jsp\_demo 01。

3) 如果使用的是 IDEA 开发工具集成Tomcat 访问web工程中的jsp，编译后的结果，存放在：

```
C:\Users\Administrator\.IntelliJ IDEA2019.1\system\tomcat\_project_tomcat\work\Catalina\localhost\jsp_demo_01_war_exploded\org\apache\jsp
```

### 3.2.2 预编译

除了运行时编译，我们还可以直接在Web应用启动时，一次性将Web应用中的所有的JSP页面一次性编译完成。在这种情况下，Web应用运行过程中，便可以不必再进行实时编译，而是直接调用JSP页面对应的Servlet 完成请求处理，从而提升系统性能。

Tomcat 提供了一个Shell程序jspC，用于支持JSP预编译，而且在Tomcat的安装目录下提供了一个 catalina-tasks.xml 文件声明了Tomcat 支持的Ant任务，因此，我们很容易使用 Ant 来执行JSP 预编译。（要想使用这种方式，必须得确保在此之前已经下载并安装了Apache Ant）。

## 3.3 JSP编译原理

### 3.3.1 代码分析

编译后的.class 字节码文件及源码：



```
public final class index_jsp extends
org.apache.jasper.runtime.HttpJspBase
    implements
org.apache.jasper.runtime.JspSourceDependent,org.apache.jasper.runtime.Js
pSourceImports {

    private static final javax.servlet.jsp.JspFactory _jspxFactory =
        javax.servlet.jsp.JspFactory.getDefaultFactory();

    private static java.util.Map<java.lang.String,java.lang.Long>
    _jspx_dependants;

    static {
        _jspx_dependants = new
java.util.HashMap<java.lang.String,java.lang.Long>(2);
        _jspx_dependants.put("jar:file:/D:/DevelopProgramFile/apache-tomcat-
8.5.42-windows-x64/apache-tomcat-8.5.42/webapps/jsp_demo_01/WEB-
INF/lib/standard.jar!/META-INF/c.tld", Long.valueOf(1098682290000L));
        _jspx_dependants.put("/WEB-INF/lib/standard.jar",
Long.valueOf(1490343635913L));
    }

    private static final java.util.Set<java.lang.String>
    _jspx_imports_packages;

    private static final java.util.Set<java.lang.String>
    _jspx_imports_classes;

    static {
        _jspx_imports_packages = new java.util.HashSet<>();
        _jspx_imports_packages.add("javax.servlet");
        _jspx_imports_packages.add("javax.servlet.http");
        _jspx_imports_packages.add("javax.servlet.jsp");
        _jspx_imports_classes = new java.util.HashSet<>();
        _jspx_imports_classes.add("java.util.Date");
        _jspx_imports_classes.add("java.text.SimpleDateFormat");
        _jspx_imports_classes.add("java.text.DateFormat");
    }

    private volatile javax.el.ExpressionFactory _el_expressionfactory;
```



```
private volatile org.apache.tomcat.InstanceManager
_jsp_instancemanager;

public java.util.Map<java.lang.String,java.lang.Long> getDependants() {
    return _jspx_dependants;
}

public java.util.Set<java.lang.String> getPackageImports() {
    return _jspx_imports_packages;
}

public java.util.Set<java.lang.String> getClassImports() {
    return _jspx_imports_classes;
}

public javax.el.ExpressionFactory _jsp_getExpressionFactory() {
    if (_el_expressionfactory == null) {
        synchronized (this) {
            if (_el_expressionfactory == null) {
                _el_expressionfactory =
                _jspxFactory.getJspApplicationContext(getServletConfig().getServletContext()).getExpressionFactory();
            }
        }
    }
    return _el_expressionfactory;
}

public org.apache.tomcat.InstanceManager _jsp_getInstanceManager() {
    if (_jsp_instancemanager == null) {
        synchronized (this) {
            if (_jsp_instancemanager == null) {
                _jsp_instancemanager =
                org.apache.jasper.runtime.InstanceManagerFactory.getInstanceManager(getServletConfig());
            }
        }
    }
    return _jsp_instancemanager;
}
```



```
public void _jspInit() {
}

public void _jspDestroy() {
}

public void _jspService(final javax.servlet.http.HttpServletRequest
request, final javax.servlet.http.HttpServletResponse response)
    throws java.io.IOException, javax.servlet.ServletException {

    final java.lang.String _jspx_method = request.getMethod();
    if (!"GET".equals(_jspx_method) && !"POST".equals(_jspx_method) &&
    !"HEAD".equals(_jspx_method) &&
    !javax.servlet.DispatcherType.ERROR.equals(request.getDispatcherType()))
    {
        response.sendError(HttpServletResponse.SC_METHOD_NOT_ALLOWED, "JSPs
only permit GET POST or HEAD");
        return;
    }

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;

    try {
        response.setContentType("text/html;charset=UTF-8");
        pageContext = _jspxFactory.getPageContext(this, request, response,
            null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;
```



```
out.write("\n");
out.write("\n");
out.write("\n");
out.write("\n");
out.write("\n");
out.write("<html>\n");
out.write("  <head>\n");
out.write("    <title>$Title$</title>\n");
out.write("  </head>\n");
out.write("  <body>\n");
out.write("    ");

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
String format = dateFormat.format(new Date());

out.write("\n");
out.write("    Hello , Java Server Page . . . . \n");
out.write(
```