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附加功能

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. 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

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
bin	/	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
conf	/	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。
lib	/	Tomcat 服务器的依赖包
logs	/	Tomcat 默认的日志存放目录
webapps	/	Tomcat 默认的Web应用部署目录
work	/	Web 应用JSP代码生成和编译的临时目录

1.6 Tomcat 启动停止

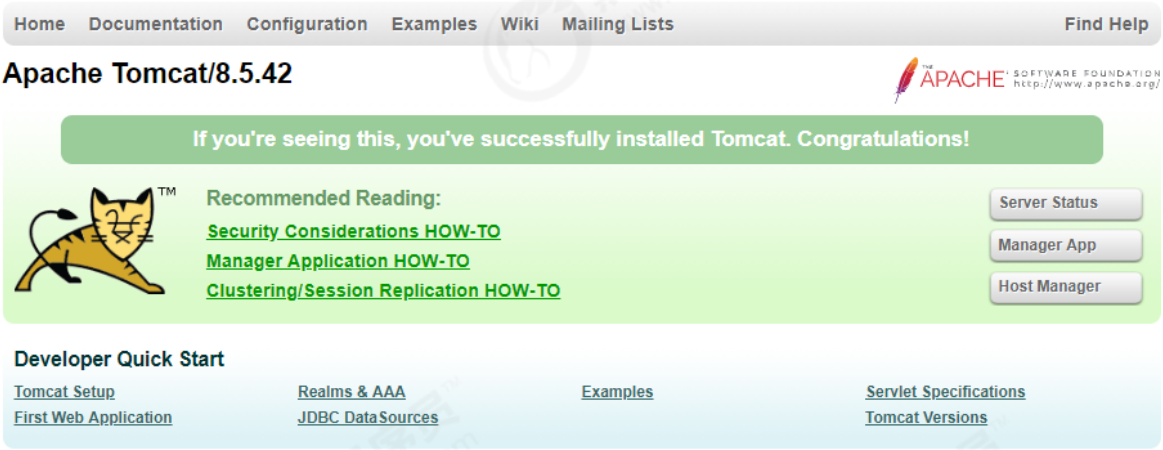
```
双击 bin/startup.bat 文件 ；
```

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

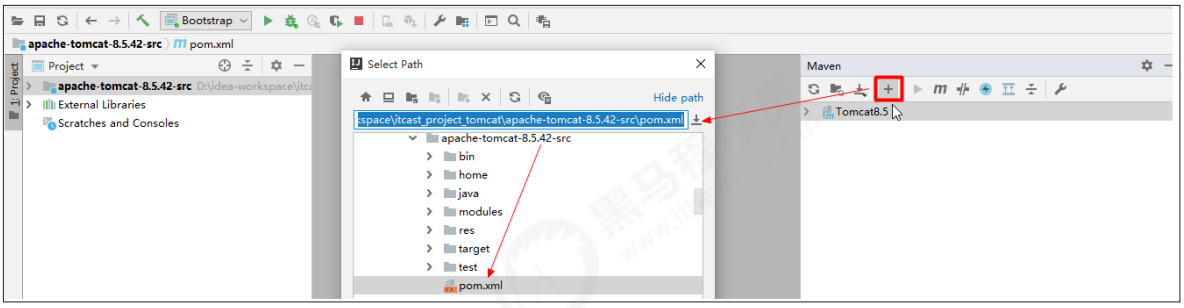
3 ）在当前目录下创建一个 pom.xml 文件，引入tomcat的依赖包

```
<?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>
```

```
        <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>
```

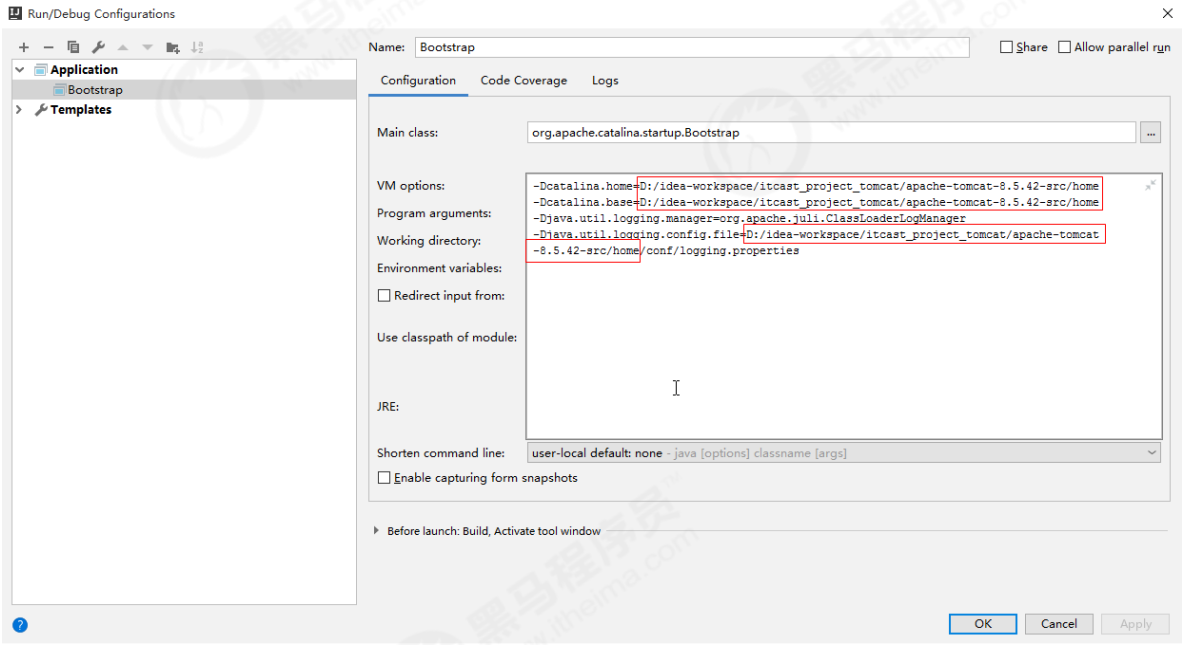
```
    </dependencies>
</project>
```

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 。

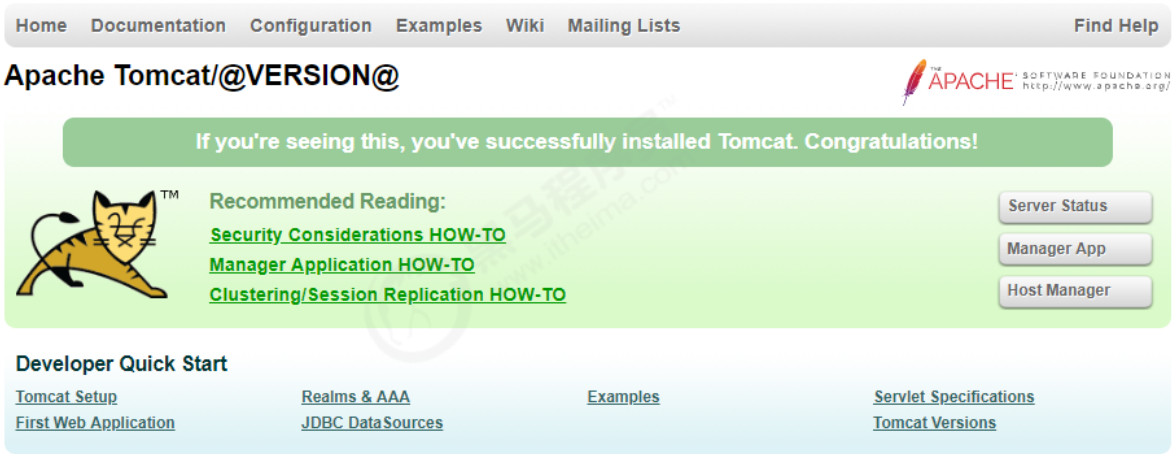
```
Type Exception Report

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();
    }
```

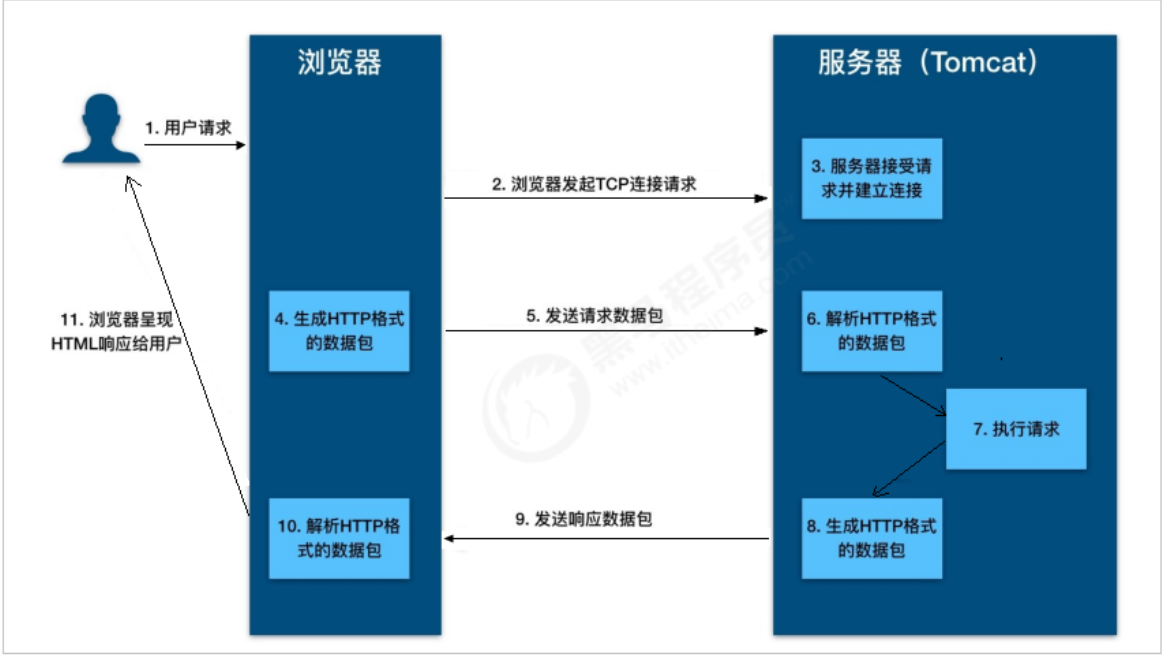
7）重启tomcat就可以正常访问了。



2.Tomcat 架构

2.1 Http工作原理

和服务器之间的通信格式。



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。

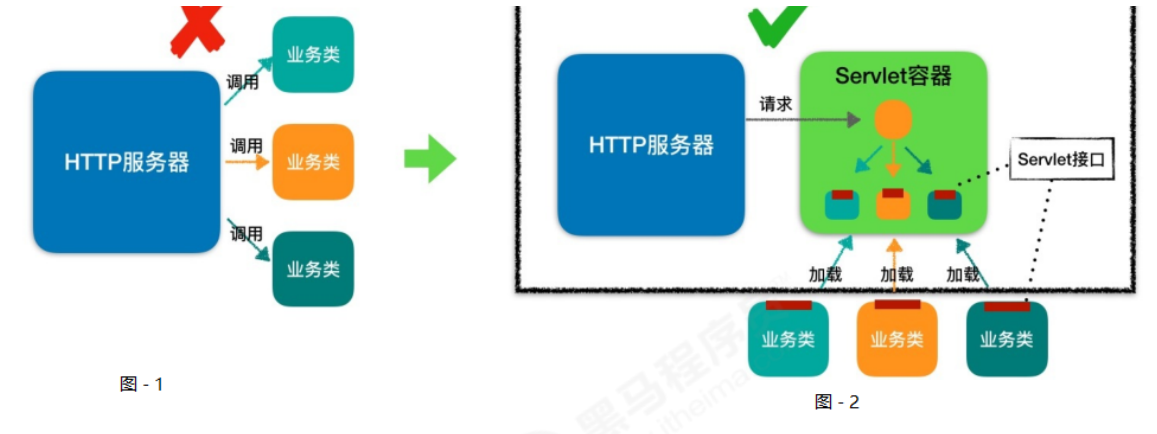
11）浏览器将HTML文件展示在页面上。

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。



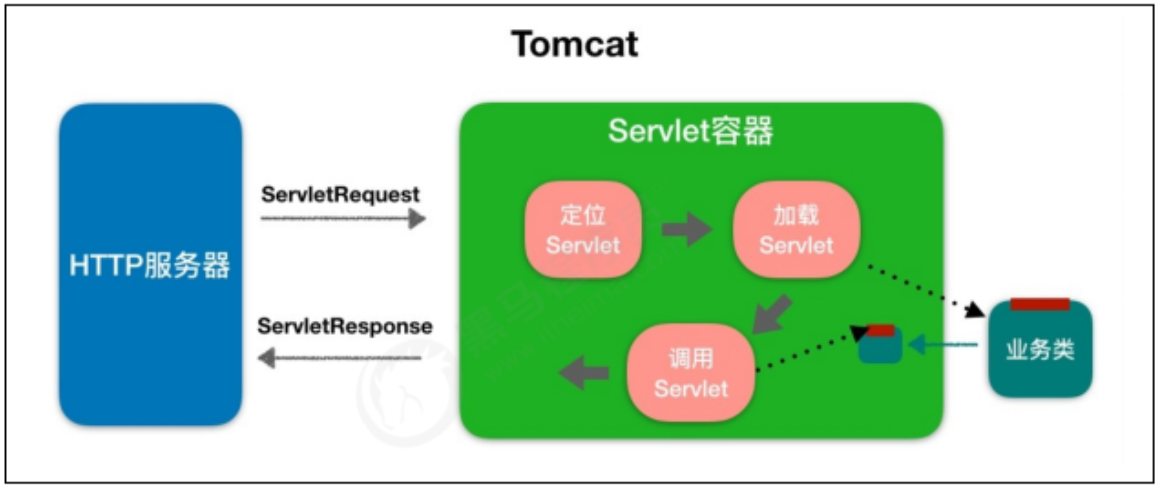
1）图1 ，表示HTTP服务器直接调用具体业务类，它们是紧耦合的。

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？

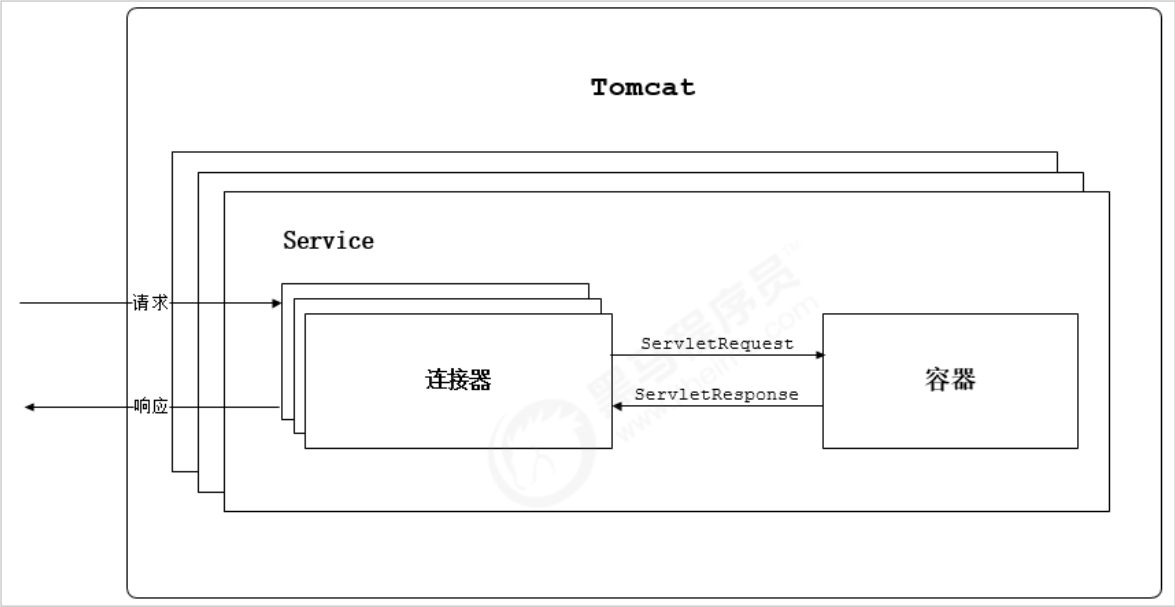
当客户请求某个资源时，HTTP服务器会用一个ServletRequest对象把客户的请求信息封装起来，然后调用Servlet容器的service方法，Servlet容器拿到请求后，根据请求的URL和Servlet的映射关系，找到相应的Servlet，如果Servlet还没有被加载，就用反射机制创建这个Servlet，并调用Servlet的init方法来完成初始化，接着调用Servlet的service方法来处理请求，把ServletResponse对象返回给HTTP服务器，HTTP服务器会把响应发送给客户端。



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。



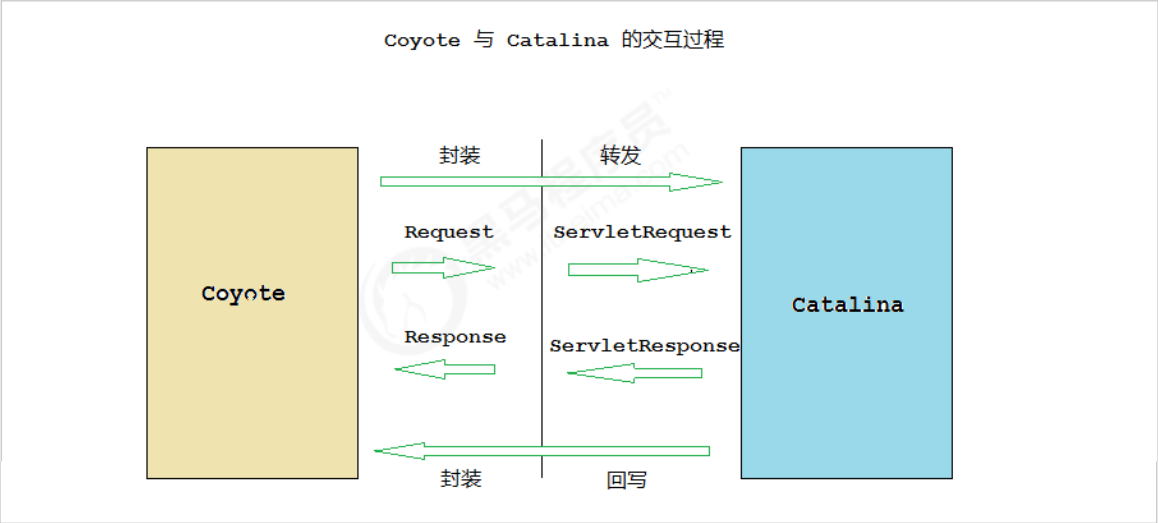
2.3 连接器 - Coyote

2.3.1 架构介绍

Coyote 是Tomcat的连接器框架的名称，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

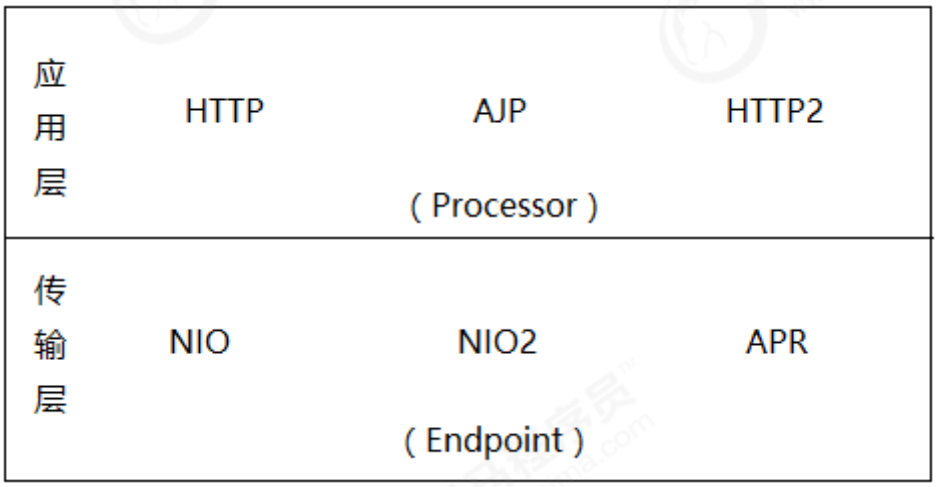
Tomcat 支持的IO模型（自8.5/9.0 版本起，Tomcat 移除了 对 BIO 的支持）：

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 ：

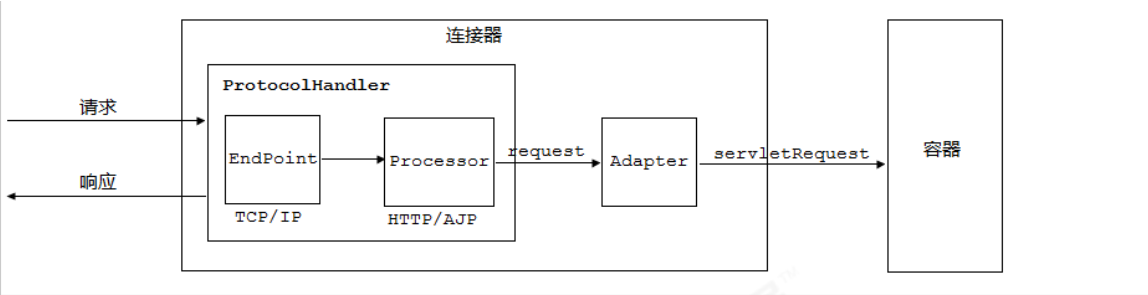
应用层协议	描述
HTTP/1.1	这是大部分Web应用采用的访问协议。
AJP	用于和Web服务器集成（如Apache），以实现对静态资源的优化以及集群部署，当前支持AJP/1.3。
HTTP/2	HTTP 2.0大幅度的提升了Web性能。下一代HTTP协议 ， 自8.5以及9.0版本之后支持。

协议分层 ：



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。但是单独的连接器或者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。



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Sevice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

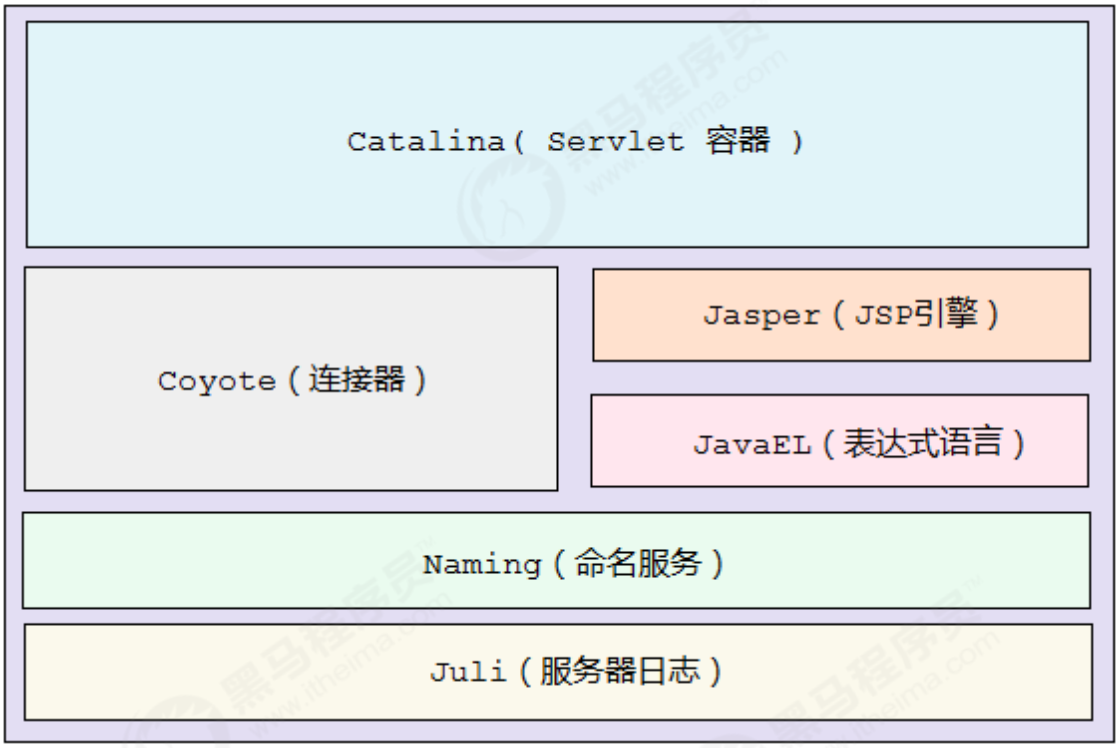
Tomcat是一个由一系列可配置的组件构成的Web容器，而Catalina是Tomcat的servlet容器。

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：

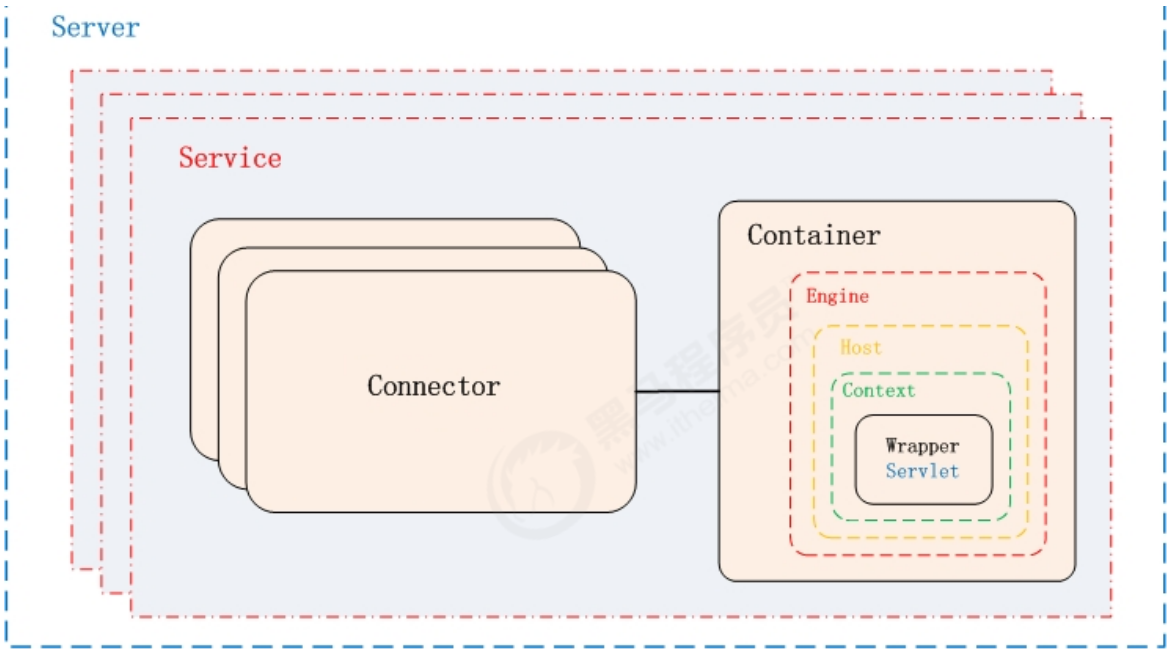
Tomcat 模块分层示意图



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：



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。

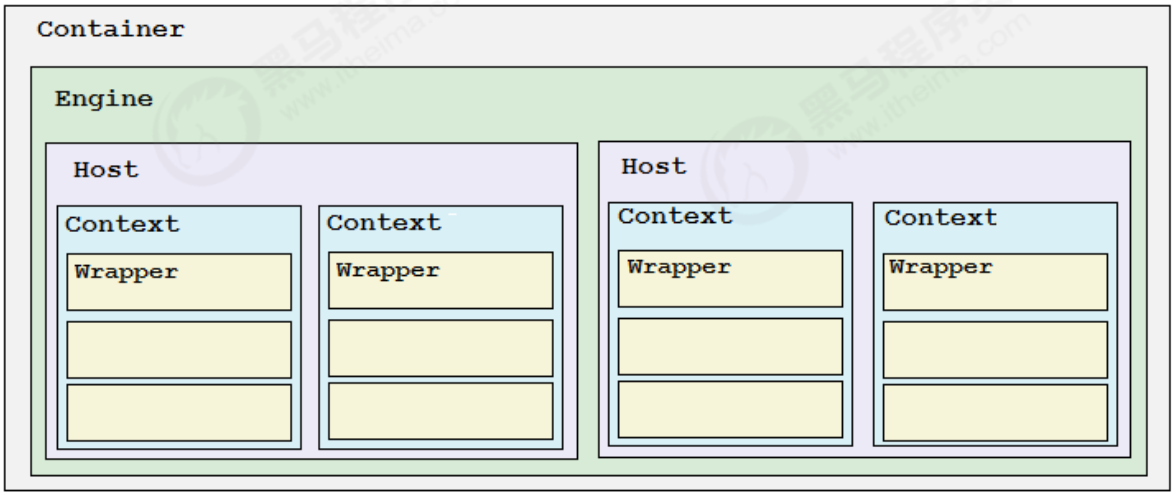
Catalina 各个组件的职责：

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

```
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。



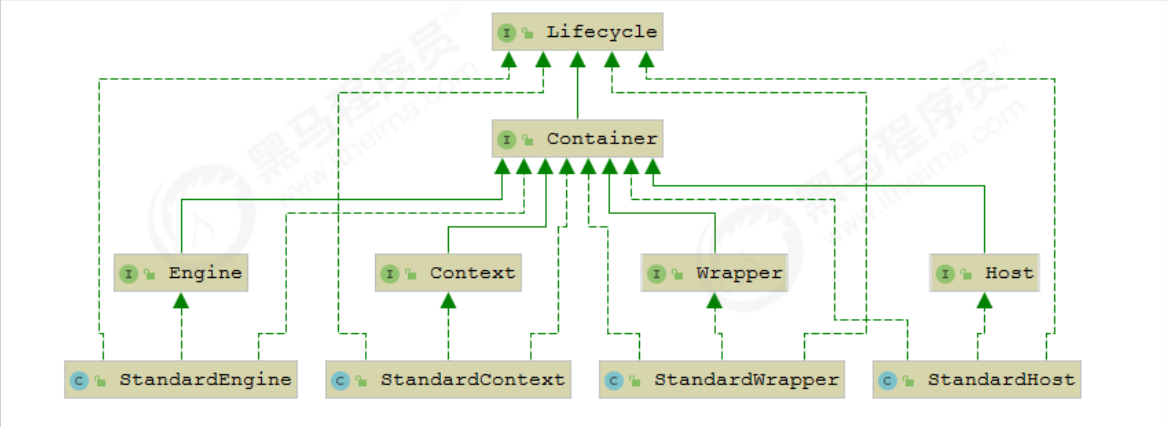
各个组件的含义 ：

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

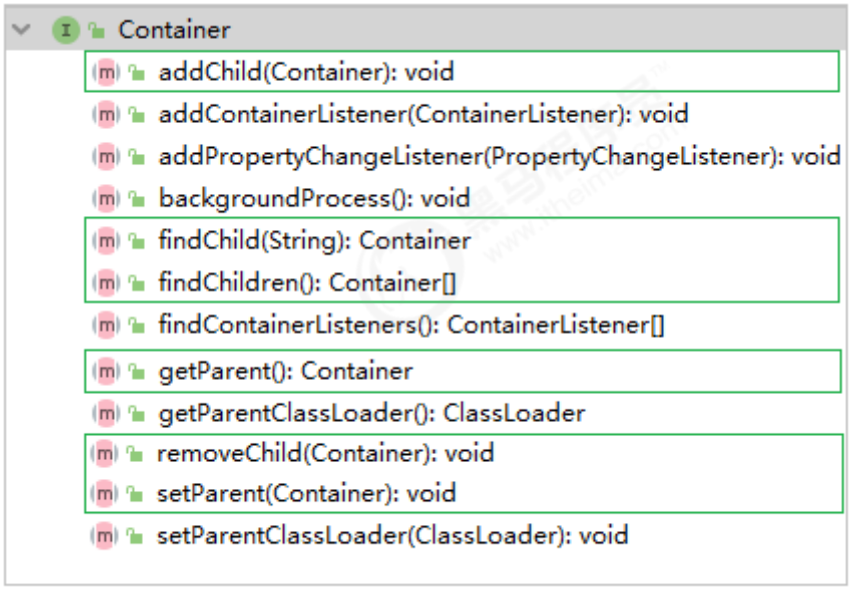
顶层容器中。

```
<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。



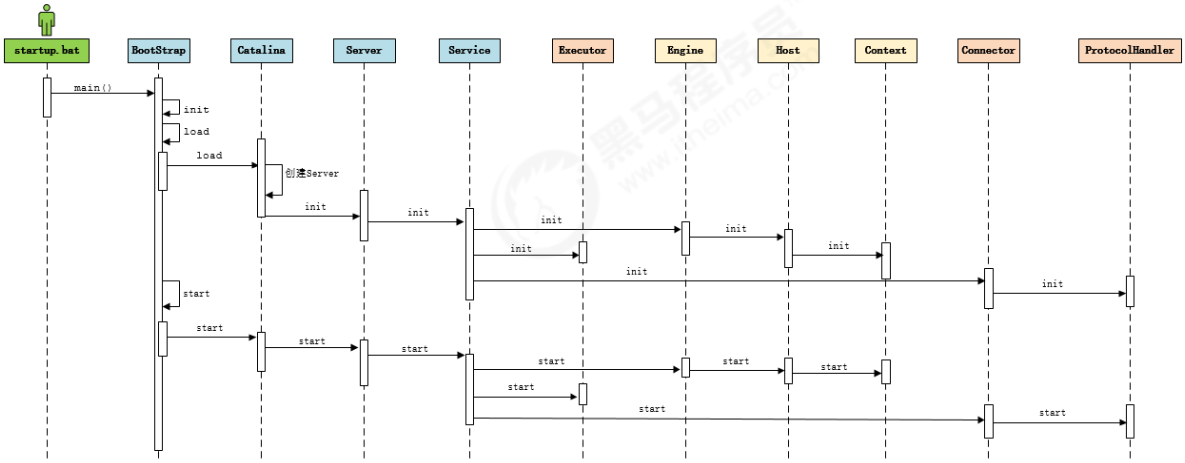
Container 接口中提供了以下方法（截图中知识一部分方法）：



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

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

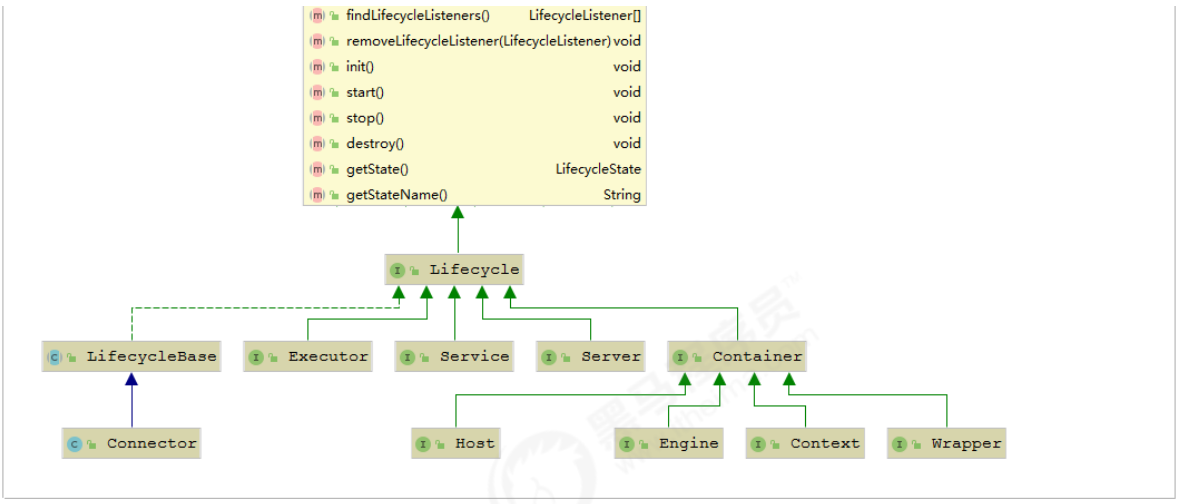
由于所有的组件均存在初始化、启动、停止等生命周期方法，拥有生命周期管理的特性， 所以Tomcat 在设计的时候， 基于生命周期管理抽象成了一个接口 Lifecycle ，而组件 Server、Service、Container、Executor、Connector 组件 ， 都实现了一个生命周期的接口，从而具有了以下生命周期中的核心方法：

1 ） init ( ) ：初始化组件

2 ） start ( ) ：启动组件

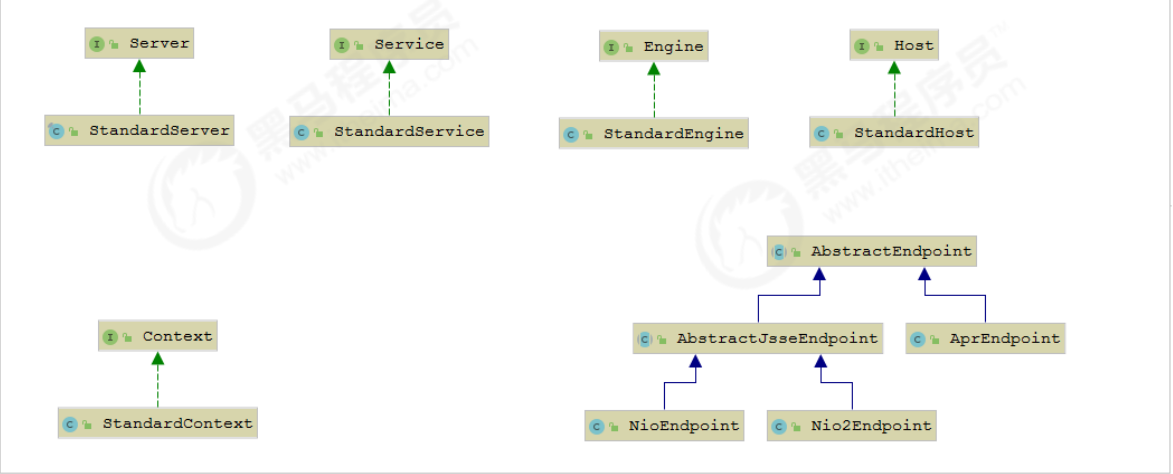
3 ） stop ( ) ：停止组件

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。

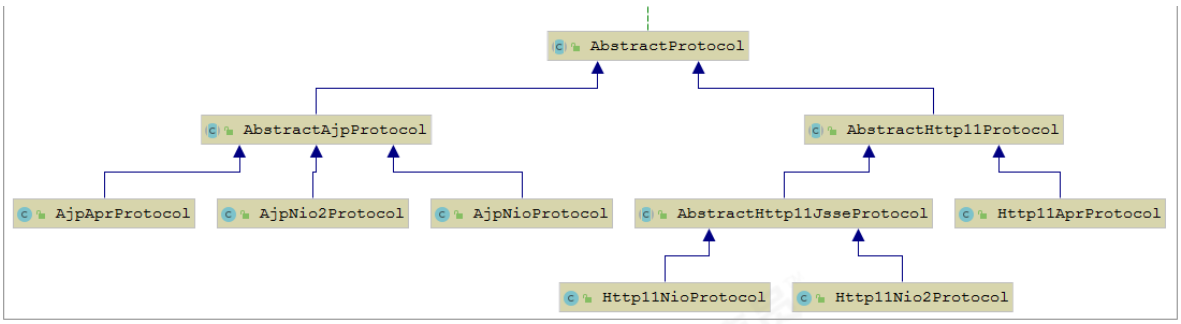
3 ）AjpAprProtocol ：采用APR的IO模型，需要依赖于APR库。

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)
```

```java
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。

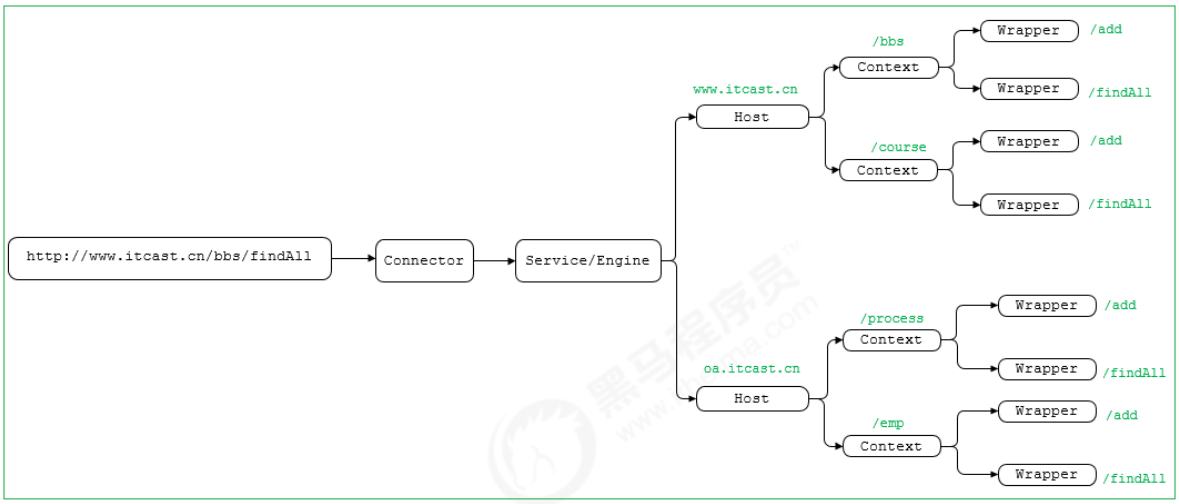
2.6 Tomcat 请求处理流程

2.6.1 请求流程

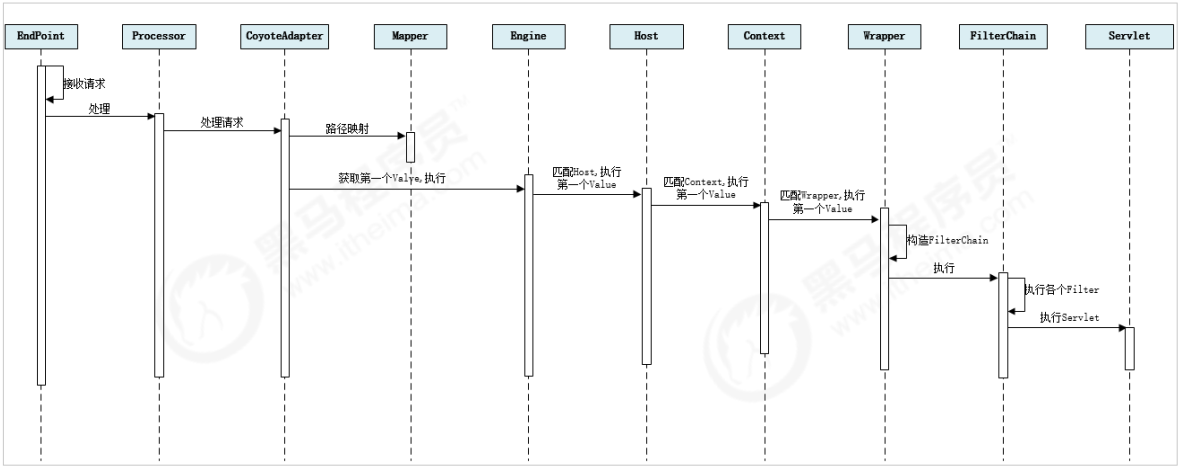
设计了这么多层次的容器，Tomcat是怎么确定每一个请求应该由哪个Wrapper容器里的Servlet来处理的呢？答案是，Tomcat是用Mapper组件来完成这个任务的。

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。



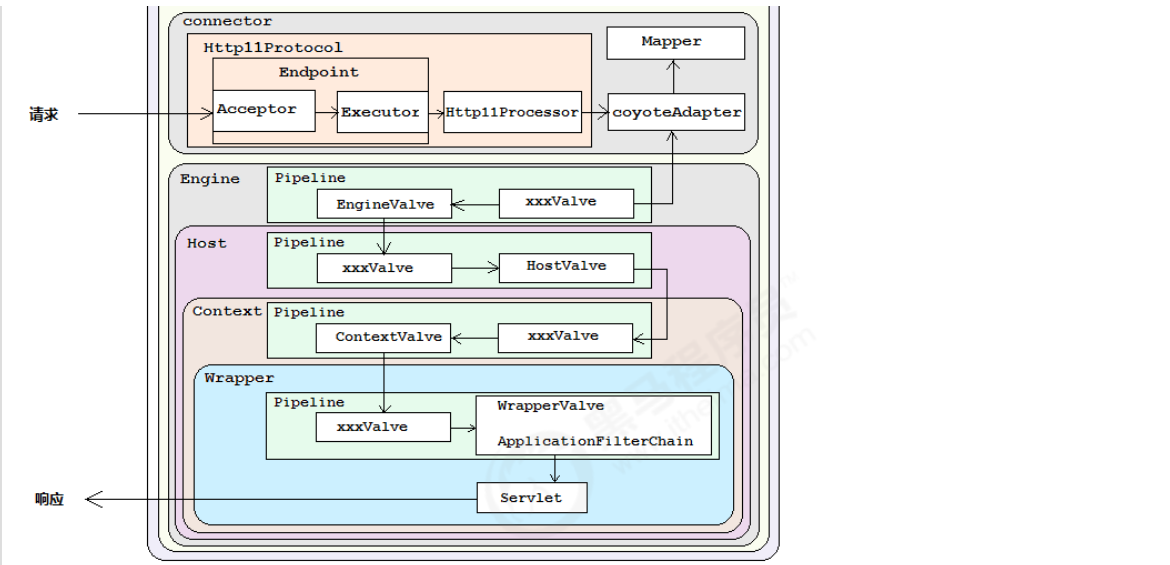
那上面这幅图只是描述了根据请求的URL如何查找到需要执行的Servlet ，那么下面我们再来解析一下，从Tomcat的设计架构层面来分析Tomcat的请求处理。



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。Pipiline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：

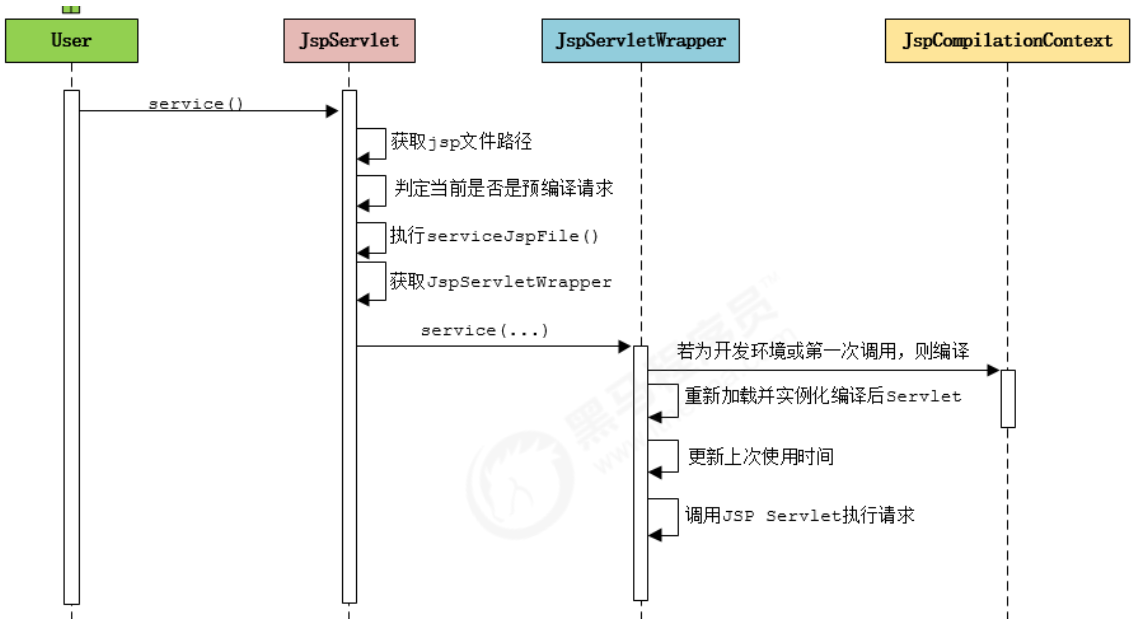
```
<%@ page import="java.util.Date" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
  <head>
    <title>$Title$</title>
  </head>
  <body>
    <%
      DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd 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，默认的目录为 ：work/Catalina/localhost/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\.IntelliJIdea2019.1\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

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

  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) {
```

```
tExpressionFactory();
        }
      }
    }
    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(getServletCo
nfig());
        }
      }
    }
    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();
```

```
      _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("\n");
      out.write("    <br/>\n");
      out.write("\n");
      out.write("    ");
      out.print( format );
      out.write("\n");
      out.write("\n");
      out.write("  </body>\n");
      out.write("</html>\n");
    } catch (java.lang.Throwable t) {
      if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          try {
            if (response.isCommitted()) {
              out.flush();
            } else {
              out.clearBuffer();
            }
          } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

由编译后的源码解读，可以分析出以下几点 ：

1 ）其类名为 index_jsp ，继承自 org.apache.jasper.runtime.HttpJspBase ，该类是HttpServlet 的子类 ，所以jsp 本质就是一个Servlet 。

资源的上次修改时间）。

3）通过属性 _jspx_imports_packages 存放导入的 java 包，默认导入 javax.servlet ，javax.servlet.http, javax.servlet.jsp 。

4）通过属性 _jspx_imports_classes 存放导入的类，通过import 指令导入的 DateFormat 、SimpleDateFormat 、Date 都会包含在该集合中。 _jspx_imports_packages 和 _jspx_imports_classes 属性主要用于配置 EL 引擎上下文 。

5）请求处理由方法 _jspService 完成 ，而在父类 HttpJspBase 中的service 方法通过模板方法模式 ，调用了子类的 _jspService 方法。

```
@Override
public final void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    _jspService(request, response);
}
```

6）_jspService 方法中定义了几个重要的局部变量 ： pageContext 、Session、application、config、out、page。由于整个页面的输出有 _jspService 方法完成，因此这些变量和参数会对整个JSP页面生效。 这也是我们为什么可以在JSP页面使用这些变量的原因。

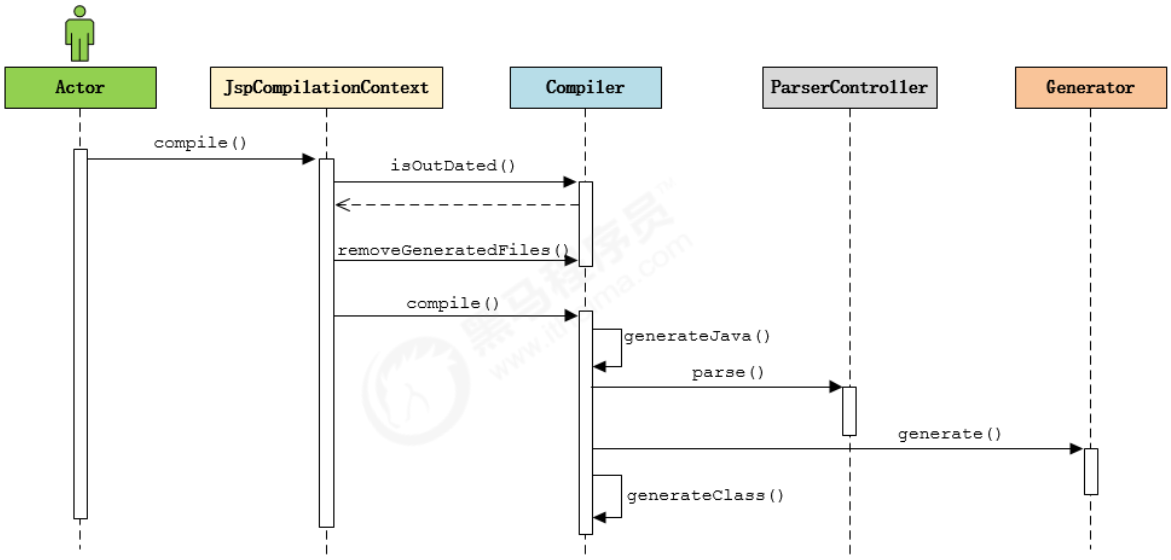
7）指定文档类型的指令（page）最终转换为 response.setContentType() 方法调用。

8）对于每一行的静态内容（HTML），调用 out.write 输出。

9）对于 <% ... %> 中的java 代码 ，将直接转换为 Servlet 类中的代码。 如果在 Java 代码中嵌入了静态文件，则同样调用 out.write 输出。

3.3.2 编译流程

JSP 编译过程如下：



Compiler 编译工作主要包含代码生成 和 编译两部分 ：

代码生成

2 ）调用ParserController 解析指令节点， 验证其是否合法，同时将配置信息保存到PageInfo 中， 用于控制代码生成。

3 ）调用ParserController 解析整个页面， 由于 JSP 是逐行解析， 所以对于每一行会创建一个具体的 Node 对象。如 静态文本（TemplateText ）、Java代码（Scriptlet ）、定制标签（CustomTag ）、Include指令（IncludeDirective ）。

4 ）验证除指令外其他所有节点的合法性， 如 脚本、定制标签、EL表达式等。

5 ）收集除指令外其他节点的页面配置信息。

6 ）编译并加载当前 JSP 页面依赖的标签

7 ）对于JSP页面的EL表达式，生成对应的映射函数。

8 ）生成JSP页面对应的Servlet 类源代码

编译

代码生成完成后， Compiler 还会生成 SMAP 信息。 如果配置生成 SMAP 信息，Compiler 则会在编译阶段将SMAP 信息写到class 文件中 。

在编译阶段， Compiler 的两个实现 AntCompiler 和 JDTCompiler 分别调用先关框架的API 进行源代码编译。

对于 AntCompiler 来说， 构造一个 Ant 的javac 的任务完成编译。

对于 JDTCompiler 来说， 调用 org.eclipse.jdt.internal.compiler.Compiler 完成编译。

4.Tomcat 服务器配置

Tomcat 服务器的配置主要集中于 tomcat/conf 下的 catalina.policy、catalina.properties、context.xml、server.xml、tomcat-users.xml、web.xml 文件。

4.1 server.xml

server.xml 是tomcat 服务器的核心配置文件，包含了Tomcat的 Servlet 容器（Catalina ）的所有配置。由于配置的属性特别多，我们在这里主要讲解其中的一部分重要配置。

4.1.1 Server

Server是server.xml的根元素，用于创建一个Server实例，默认使用的实现类是org.apache.catalina.core.StandardServer。

```
<Server port="8005" shutdown="SHUTDOWN">
    ...
</Server>
```

port : Tomcat 监听的关闭服务器的端口。

shutdown： 关闭服务器的指令字符串。

默认配置的5个Listener 的含义：

```
<!-- 用于以日志形式输出服务器 、操作系统、JVM的版本信息 -->
<Listener className="org.apache.catalina.startup.VersionLoggerListener" />

<!-- 用于加载（服务器启动） 和 销毁 （服务器停止） APR。 如果找不到APR库， 则会输出日志， 并不影响Tomcat启动 -->
<Listener className="org.apache.catalina.core.AprLifecycleListener"
SSLEngine="on" />

<!-- 用于避免JRE内存泄漏问题 -->
<Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener"
/>

<!-- 用户加载（服务器启动） 和 销毁（服务器停止） 全局命名服务 -->
<Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />

<!-- 用于在Context停止时重建Executor 池中的线程， 以避免ThreadLocal 相关的内存泄漏 -->
<Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener"
/>
```

GlobalNamingResources 中定义了全局命名服务：

```
<!-- Global JNDI resources
     Documentation at /docs/jndi-resources-howto.html
-->
<GlobalNamingResources>
    <!-- Editable user database that can also be used by
         UserDatabaseRealm to authenticate users
    -->
    <Resource name="UserDatabase" auth="Container"
              type="org.apache.catalina.UserDatabase"
              description="User database that can be updated and saved"
              factory="org.apache.catalina.users.MemoryUserDatabaseFactory"
              pathname="conf/tomcat-users.xml" />
</GlobalNamingResources>
```

4.1.2 Service

该元素用于创建 Service 实例，默认使用 org.apache.catalina.core.StandardService。 默认情况下，Tomcat 仅指定了Service 的名称， 值为 "Catalina"。Service 可以内嵌的元素为 ： Listener、Executor、Connector、Engine，其中 ： Listener 用于为Service添加生命周期监听器， Executor 用于配置Service 共享线程池，Connector 用于配置Service 包含的链接器， Engine 用于配置Service中链接器对应的Servlet 容器引擎。

```
<Service name="Catalina">
    ...
</Service>
```

4.1.3 Executor

默认情况下，Service 并未添加共享线程池配置。 如果我们想添加一个线程池， 可以在 下添加如下配置：

```
<Executor name="tomcatThreadPool"
    namePrefix="catalina-exec-"
    maxThreads="200"
    minSpareThreads="100"
    maxIdleTime="60000"
    maxQueueSize="Integer.MAX_VALUE"
    prestartminSpareThreads="false"
    threadPriority="5"
    className="org.apache.catalina.core.StandardThreadExecutor"/>
```

属性说明：

属性	含义
name	线程池名称，用于 Connector中指定。
namePrefix	所创建的每个线程的名称前缀，一个单独的线程名称为 namePrefix+threadNumber。
maxThreads	池中最大线程数。
minSpareThreads	活跃线程数，也就是核心池线程数，这些线程不会被销毁，会一直存在。
maxIdleTime	线程空闲时间，超过该时间后，空闲线程会被销毁，默认值为 6000（1分钟），单位毫秒。
maxQueueSize	在被执行前最大线程排队数目，默认为Int的最大值，也就是广义的无限。除非特殊情况，这个值不需要更改，否则会有请求不会被处理的情况发生。

prestartminSpareThreads	启动线程池时是否启动 minSpareThreads部分线程。默认值为false，即不启动。
threadPriority	线程池中线程优先级，默认值为5，值从1到10。
className	线程池实现类，未指定情况下，默认实现类为org.apache.catalina.core.StandardThreadExecutor。如果想使用自定义线程池首先需要实现 org.apache.catalina.Executor接口。

如果不配置共享线程池，那么Catalina 各组件在用到线程池时会独立创建。

4.1.4 Connector

Connector 用于创建链接器实例。默认情况下，server.xml 配置了两个链接器，一个支持HTTP协议，一个支持AJP协议。因此大多数情况下，我们并不需要新增链接器配置，只是根据需要对已有链接器进行优化。

```
<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000"
redirectPort="8443" />

<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />
```

属性说明：

1）port：端口号，Connector 用于创建服务端Socket 并进行监听，以等待客户端请求链接。如果该属性设置为0，Tomcat将会随机选择一个可用的端口号给当前Connector 使用。

2）protocol ：当前Connector 支持的访问协议。 默认为 HTTP/1.1 ，并采用自动切换机制选择一个基于 JAVA NIO 的链接器或者基于本地APR的链接器（根据本地是否含有Tomcat的本地库判定）。

如果不希望采用上述自动切换的机制，而是明确指定协议，可以使用以下值。

Http协议：

```
org.apache.coyote.http11.Http11AprProtocol ，  APR 链接器
```

AJP协议 ：

```
org.apache.coyote.ajp.AjpNioProtocol ，  非阻塞式 Java NIO 链接器
org.apache.coyote.ajp.AjpNio2Protocol ，非阻塞式 JAVA NIO2 链接器
org.apache.coyote.ajp.AjpAprProtocol ，  APR 链接器
```

3 ） connectionTimeOut : Connector 接收链接后的等待超时时间， 单位为 毫秒。 -1 表示不超时。

4 ） redirectPort：当前Connector 不支持SSL请求， 接收到了一个请求， 并且也符合security-constraint 约束， 需要SSL传输，Catalina自动将请求重定向到指定的端口。

5 ） executor ： 指定共享线程池的名称， 也可以通过maxThreads、minSpareThreads 等属性配置内部线程池。

6 ） URIEncoding : 用于指定编码URI的字符编码， Tomcat8.x版本默认的编码为 UTF-8 , Tomcat7.x版本默认为ISO-8859-1。

完整的配置如下：

```
<Connector port="8080"
          protocol="HTTP/1.1"
          executor="tomcatThreadPool"
          maxThreads="1000"
          minSpareThreads="100"
          acceptCount="1000"
          maxConnections="1000"
          connectionTimeout="20000"
          compression="on"
          compressionMinSize="2048"
          disableUploadTimeout="true"
          redirectPort="8443"
          URIEncoding="UTF-8" />
```

4.1.5 Engine

Engine 作为Servlet 引擎的顶级元素，内部可以嵌入： Cluster、Listener、Realm、Valve和Host。

```
<Engine name="Catalina" defaultHost="localhost">
    ...
</Engine>
```

属性说明：

1 ） name： 用于指定Engine 的名称， 默认为Catalina 。该名称会影响一部分Tomcat的存储路径（如临时文件）。

2 ） defaultHost ： 默认使用的虚拟主机名称， 当客户端请求指向的主机无效时， 将交由默认的虚拟主机处理， 默认为localhost。

Host 元素用于配置一个虚拟主机， 它支持以下嵌入元素：Alias、Cluster、Listener、Valve、Realm、Context。如果在Engine下配置Realm， 那么此配置将在当前Engine下的所有Host中共享。 同样，如果在Host中配置Realm ， 则在当前Host下的所有Context中共享。Context中的Realm优先级 > Host的Realm优先级 > Engine中的Realm优先级。

```
<Host name="localhost"  appBase="webapps" unpackWARs="true" autoDeploy="true">
    ...
</Host>
```

属性说明：

1 ） name: 当前Host通用的网络名称， 必须与DNS服务器上的注册信息一致。 Engine中包含的Host必须存在一个名称与Engine的defaultHost设置一致。

2 ） appBase： 当前Host的应用基础目录， 当前Host上部署的Web应用均在该目录下（可以是绝对目录，相对路径）。默认为webapps。

3 ） unpackWARs： 设置为true， Host在启动时会将appBase目录下war包解压为目录。设置为false， Host将直接从war文件启动。

4 ） autoDeploy： 控制tomcat是否在运行时定期检测并自动部署新增或变更的web应用。

通过给Host添加别名，我们可以实现同一个Host拥有多个网络名称，配置如下：

```
<Host name="www.web1.com"  appBase="webapps" unpackWARs="true"
autoDeploy="true">
    <Alias>www.web2.com</Alias>
</Host>
```

这个时候，我们就可以通过两个域名访问当前Host下的应用（需要确保DNS或hosts中添加了域名的映射配置）。

4.1.7 Context

Context 用于配置一个Web应用，默认的配置如下：

```
<Context docBase="myApp" path="/myApp">
    ....
</Context>
```

属性描述：

1 ） docBase：Web应用目录或者War包的部署路径。可以是绝对路径，也可以是相对于 Host appBase的相对路径。

2 ） path：Web应用的Context 路径。如果我们Host名为localhost， 则该web应用访问的根路径为：http://localhost:8080/myApp。

它支持的内嵌元素为：CookieProcessor， Loader， Manager，Realm，Resources，WatchedResource，JarScanner，Valve。

```
autoDeploy="true">

    <Context docBase="D:\servlet_project03" path="/myApp"></Context>

    <Valve className="org.apache.catalina.valves.AccessLogValve"
directory="logs"
           prefix="localhost_access_log" suffix=".txt"
           pattern="%h %l %u %t &quot;%r&quot; %s %b" />

</Host>
```

4.2 tomcat-users.xml

该配置文件中，主要配置的是Tomcat的用户，角色等信息，用来控制Tomcat中manager， host-manager的访问权限。

5.Web 应用配置

web.xml 是web应用的描述文件， 它支持的元素及属性来自于Servlet 规范定义 。 在Tomcat 中，Web 应用的描述信息包括 tomcat/conf/web.xml 中默认配置 以及 Web 应用 WEB-INF/web.xml 下的定制配置。

```
servlet                          http://xmlns.jcp.org/xml/ns/javaee
filter-mapping                   http://xmlns.jcp.org/xml/ns/javaee
filter                           http://xmlns.jcp.org/xml/ns/javaee
listener                         http://xmlns.jcp.org/xml/ns/javaee
context-param                    http://xmlns.jcp.org/xml/ns/javaee
absolute-ordering                http://xmlns.jcp.org/xml/ns/javaee
administered-object              http://xmlns.jcp.org/xml/ns/javaee
connection-factory               http://xmlns.jcp.org/xml/ns/javaee
data-source                      http://xmlns.jcp.org/xml/ns/javaee
deny-uncovered-http-methods      http://xmlns.jcp.org/xml/ns/javaee
description                      http://xmlns.jcp.org/xml/ns/javaee
display-name                     http://xmlns.jcp.org/xml/ns/javaee
distributable                    http://xmlns.jcp.org/xml/ns/javaee
ejb-local-ref                    http://xmlns.jcp.org/xml/ns/javaee
ejb-ref                          http://xmlns.jcp.org/xml/ns/javaee
env-entry                        http://xmlns.jcp.org/xml/ns/javaee
error-page                       http://xmlns.jcp.org/xml/ns/javaee
icon                             http://xmlns.jcp.org/xml/ns/javaee
jms-connection-factory           http://xmlns.jcp.org/xml/ns/javaee
jms-destination                  http://xmlns.jcp.org/xml/ns/javaee
jsp-config                       http://xmlns.jcp.org/xml/ns/javaee
locale-encoding-mapping-list     http://xmlns.jcp.org/xml/ns/javaee
login-config                     http://xmlns.jcp.org/xml/ns/javaee
mail-session                     http://xmlns.jcp.org/xml/ns/javaee
message-destination              http://xmlns.jcp.org/xml/ns/javaee
message-destination-ref          http://xmlns.jcp.org/xml/ns/javaee
mime-mapping                     http://xmlns.jcp.org/xml/ns/javaee
module-name                      http://xmlns.jcp.org/xml/ns/javaee
persistence-context-ref          http://xmlns.jcp.org/xml/ns/javaee
persistence-unit-ref             http://xmlns.jcp.org/xml/ns/javaee
post-construct                   http://xmlns.jcp.org/xml/ns/javaee
pre-destroy                      http://xmlns.jcp.org/xml/ns/javaee
resource-env-ref                 http://xmlns.jcp.org/xml/ns/javaee
resource-ref                     http://xmlns.jcp.org/xml/ns/javaee
security-constraint              http://xmlns.jcp.org/xml/ns/javaee
security-role                    http://xmlns.jcp.org/xml/ns/javaee
service-ref                      http://xmlns.jcp.org/xml/ns/javaee
session-config                   http://xmlns.jcp.org/xml/ns/javaee
```

5.1 ServletContext 初始化参数

我们可以通过 添加ServletContext 初始化参数，它配置了一个键值对，这样我们可以在应用程序中使用 javax.servlet.ServletContext.getInitParameter()方法获取参数。

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext-*.xml</param-value>
    <description>Spring Config File Location</description>
</context-param>
```

5.2 会话配置

用于配置Web应用会话，包括 超时时间、Cookie配置以及会话追踪模式。它将覆盖server.xml 和 context.xml 中的配置。

```
<session-config>
  <session-timeout>30</session-timeout>
  <cookie-config>
    <name>JESSIONID</name>
    <domain>www.itcast.cn</domain>
    <path>/</path>
    <comment>Session Cookie</comment>
    <http-only>true</http-only>
    <secure>false</secure>
    <max-age>3600</max-age>
  </cookie-config>
  <tracking-mode>COOKIE</tracking-mode>
</session-config>
```

配置解析：

```
1） session-timeout ： 会话超时时间，单位 分钟

2） cookie-config： 用于配置会话追踪Cookie

    name：Cookie的名称

    domain：Cookie的域名

    path：Cookie的路径

    comment：注释

    http-only：cookie只能通过HTTP方式进行访问，JS无法读取或修改，此项可以增加网站访问的安全性。

    secure：此cookie只能通过HTTPS连接传递到服务器，而HTTP 连接则不会传递该信息。注意是从浏览器传递到服务器，服务器端的Cookie对象不受此项影响。

    max-age：以秒为单位表示cookie的生存期，默认为-1表示是会话Cookie，浏览器关闭时就会消失。

3） tracking-mode ：用于配置会话追踪模式，Servlet3.0版本中支持的追踪模式：COOKIE、URL、SSL

    A. COOKIE ： 通过HTTP Cookie 追踪会话是最常用的会话追踪机制， 而且Servlet规范也要求所有的Servlet规范都需要支持Cookie追踪。

    B. URL ： URL重写是最基本的会话追踪机制。当客户端不支持Cookie时，可以采用URL重写的方式。当采用URL追踪模式时，请求路径需要包含会话标识信息，Servlet容器会根据路径中的会话标识设置请求的会话信息。如： http://www.myserver.com/user/index.html;jessionid=1234567890。

    C. SSL ： 对于SSL请求， 通过SSL会话标识确定请求会话标识。
```

Servlet 的配置主要是两部分， servlet 和 servlet-mapping ：

```
<servlet>
  <servlet-name>myServlet</servlet-name>
  <servlet-class>cn.itcast.web.MyServlet</servlet-class>
  <init-param>
    <param-name>fileName</param-name>
    <param-value>init.conf</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  <enabled>true</enabled>
</servlet>

<servlet-mapping>
  <servlet-name>myServlet</servlet-name>
  <url-pattern>*.do</url-pattern>
  <url-pattern>/myservet/*</url-pattern>
</servlet-mapping>
```

配置说明：

```
1） servlet-name ： 指定servlet的名称， 该属性在web.xml中唯一。

2） servlet-class ： 用于指定servlet类名

3） init-param： 用于指定servlet的初始化参数， 在应用中可以通过
HttpServlet.getInitParameter 获取。

4） load-on-startup： 用于控制在Web应用启动时，Servlet的加载顺序。 值小于0，web应用启动
时，不加载该servlet，第一次访问时加载。

5） enabled： true ， false 。 若为false ，表示Servlet不处理任何请求。

6） url-pattern： 用于指定URL表达式，一个 servlet-mapping可以同时配置多个 url-pattern。
```

Servlet 中文件上传配置：

```
<servlet>
    <servlet-name>uploadServlet</servlet-name>
    <servlet-class>cn.itcast.web.UploadServlet</servlet-class>
    <multipart-config>
      <location>C://path</location>
      <max-file-size>10485760</max-file-size>
      <max-request-size>10485760</max-request-size>
      <file-size-threshold>0</file-size-threshold>
    </multipart-config>
</servlet>
```

配置说明：

```
2） max-file-size：允许上传的文件最大值。 默认值为-1， 表示没有限制。

3） max-request-size：针对该 multi/form-data 请求的最大数量，默认值为-1， 表示无限制。

4） file-size-threshold：当数量量大于该值时， 内容会被写入文件。
```

5.4 Listener配置

Listener用于监听servlet中的事件，例如context、request、session对象的创建、修改、删除，并触发响应事件。Listener是观察者模式的实现，在servlet中主要用于对context、request、session对象的生命周期进行监控。在servlet2.5规范中共定义了8中Listener。在启动时，ServletContextListener的执行顺序与web.xml 中的配置顺序一致， 停止时执行顺序相反。

```
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

5.5 Filter配置

filter 用于配置web应用过滤器， 用来过滤资源请求及响应。 经常用于认证、日志、加密、数据转换等操作， 配置如下：

```
<filter>
    <filter-name>myFilter</filter-name>
    <filter-class>cn.itcast.web.MyFilter</filter-class>
    <async-supported>true</async-supported>
    <init-param>
      <param-name>language</param-name>
      <param-value>CN</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>myFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

配置说明：

```
2） filter-class ： 过滤器的全限定类名， 该类必须实现Filter接口。

3） async-supported： 该过滤器是否支持异步

4） init-param ：用于配置Filter的初始化参数， 可以配置多个， 可以通过
FilterConfig.getInitParameter获取

5） url-pattern： 指定该过滤器需要拦截的URL。
```

5.6 欢迎页面配置

welcome-file-list 用于指定web应用的欢迎文件列表。

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

尝试请求的顺序，从上到下。

5.7 错误页面配置

error-page 用于配置Web应用访问异常时定向到的页面，支持HTTP响应码和异常类两种形式。

```
<error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/500.html</location>
</error-page>
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
</error-page>
```

6.Tomcat 管理配置

从早期的Tomcat版本开始，就提供了Web版的管理控制台，他们是两个独立的Web应用，位于webapps目录下。Tomcat 提供的管理应用有用于管理的Host的host-manager和用于管理Web应用的manager。

6.1 host-manager

中配置）。所以要想访问该页面，需要在conf/tomcat-users.xml 中配置，并分配对应的角色：

1）admin-gui：用于控制页面访问权限

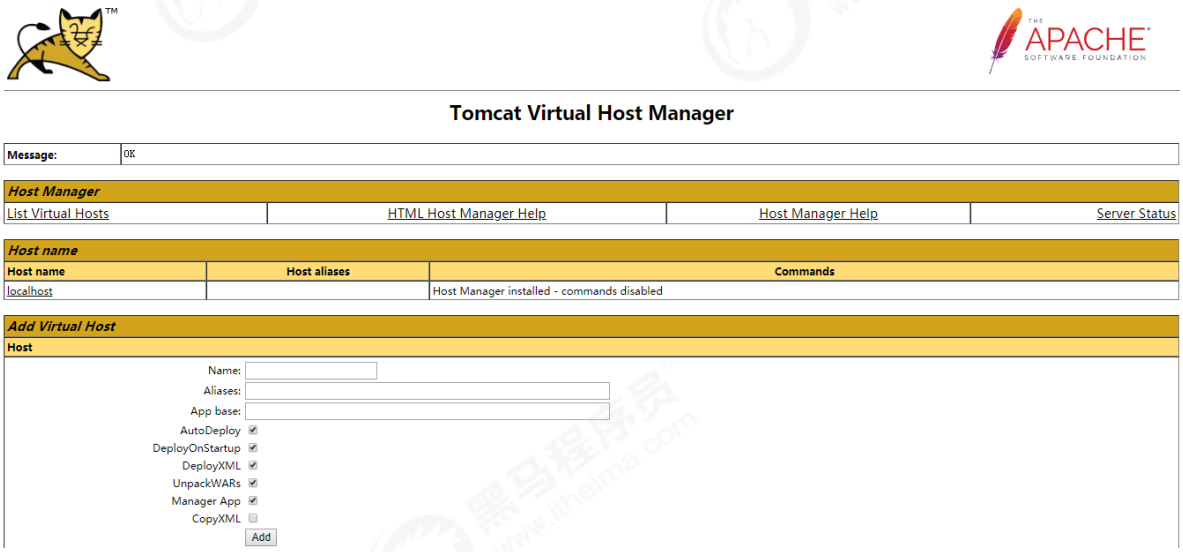
2）admin-script：用于控制以简单文本的形式进行访问

配置如下：

```
<role rolename="admin-gui"/>
<role rolename="admin-script"/>
<user username="itcast" password="itcast" roles="admin-script,admin-gui"/>
```

登录：

界面：

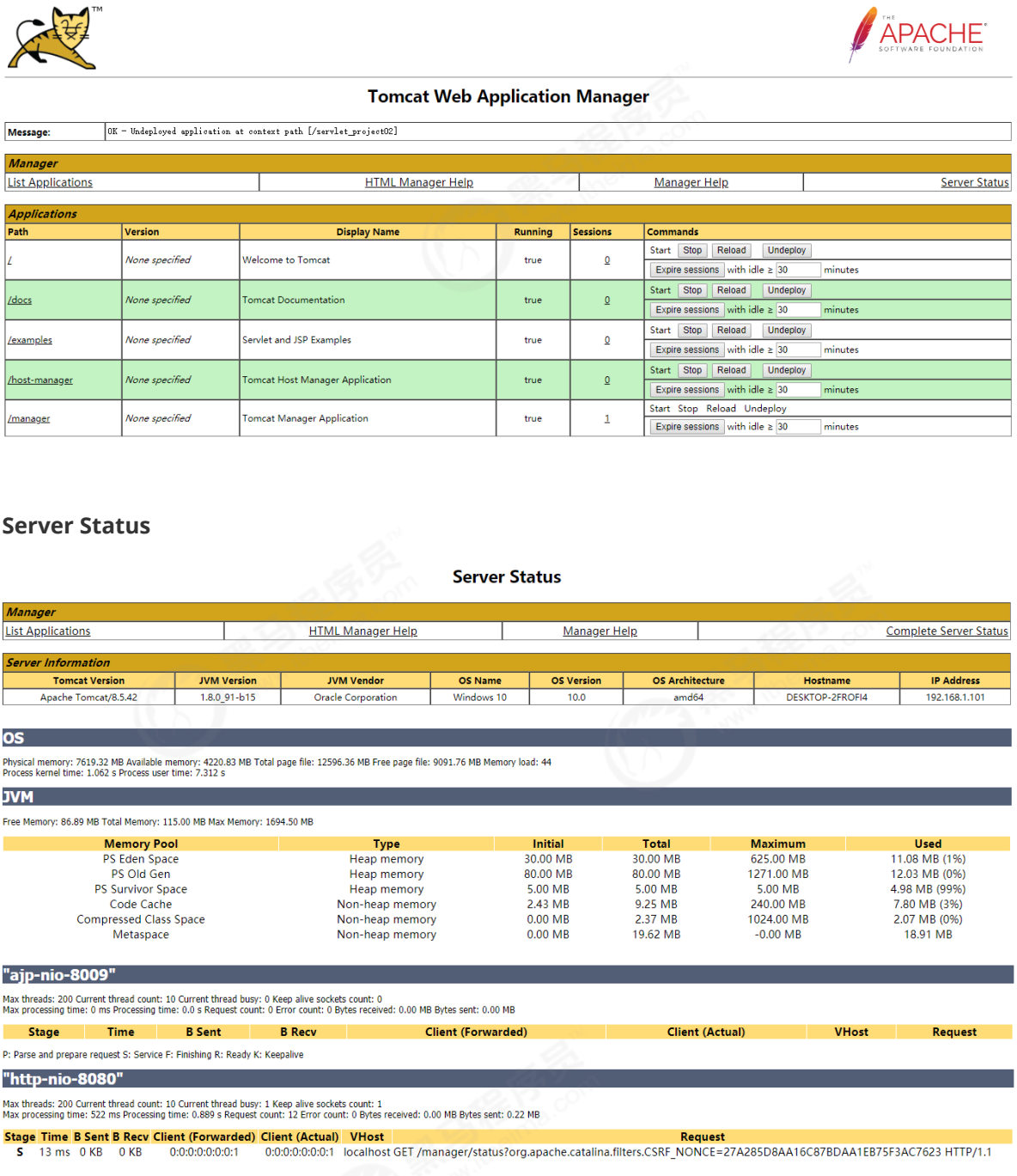


6.2 manager

manager的访问地址为 http://localhost:8080/manager，同样，manager也添加了页面访问控制，因此我们需要为登录用户分配角色为：

```
<role rolename="manager-script"/>
<user username="itcast" password="itcast" roles="admin-script,admin-gui,manager-gui,manager-script"/>
```

界面：

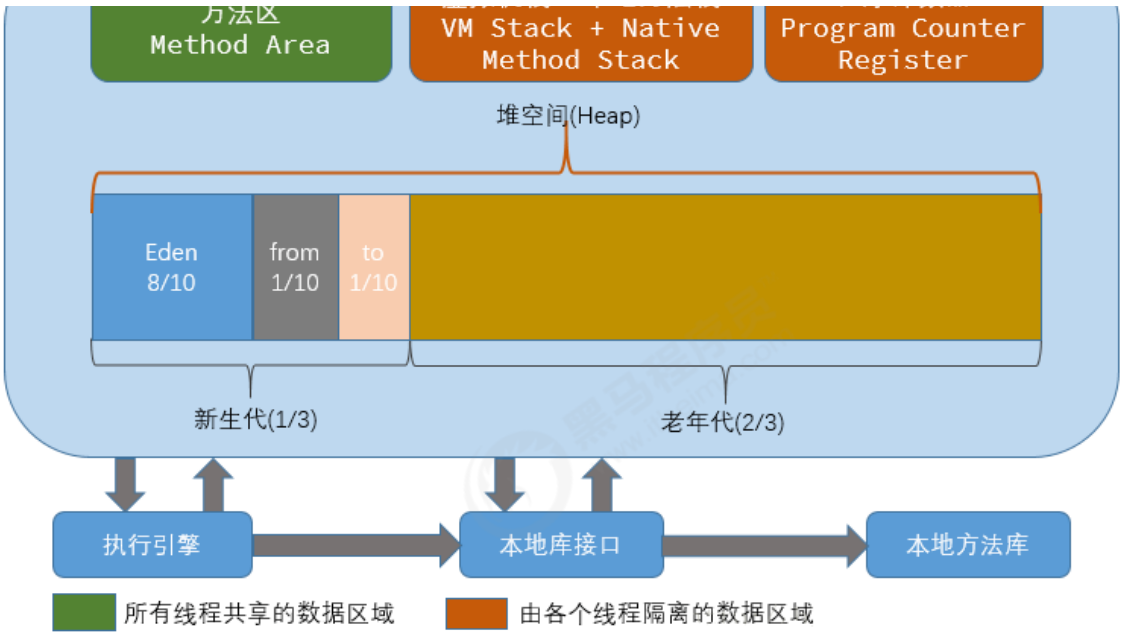


Server Status

7.JVM 配置

最常见的JVM配置当属内存分配，因为在绝大多数情况下，JVM默认分配的内存可能不能够满足我们的需求，特别是在生产环境，此时需要手动修改Tomcat启动时的内存参数分配。

7.1 JVM内存模型图



7.2 JVM配置选项

windows 平台(catalina.bat)：

```
set JAVA_OPTS=-server  -Xms2048m  -Xmx2048m  -XX:MetaspaceSize=256m -
XX:MaxMetaspaceSize=256m -XX:SurvivorRatio=8
```

linux 平台(catalina.sh)：

```
JAVA_OPTS="-server -Xms1024m -Xmx2048m -XX:MetaspaceSize=256m -
XX:MaxMetaspaceSize=512m  -XX:SurvivorRatio=8"
```

参数说明 ：

序号	参数	含义
1	-Xms	堆内存的初始大小
2	-Xmx	堆内存的最大大小
3	-Xmn	新生代的内存大小，官方建议是整个堆得3/8。
4	-XX:MetaspaceSize	元空间内存初始大小， 在JDK1.8版本之前配置为 -XX:PermSize（永久代）
5	-XX:MaxMetaspaceSize	元空间内存最大大小， 在JDK1.8版本之前配置为 -XX:MaxPermSize（永久代）
6	-XX:InitialCodeCacheSize -XX:ReservedCodeCacheSize	代码缓存区大小

7	-XX:NewRatio	XX:NewRatio=3 指定老年代 / 新生代为 3/1。 老年代占堆大小的 3/4，新生代占 1/4 。
8	-XX:SurvivorRatio	指定伊甸园区 (Eden) 与幸存区大小比例。如 -XX:SurvivorRatio=10 表示伊甸园区 (Eden) 是 幸存区 To 大小的 10 倍 (也是幸存区 From 的 10 倍)。 所以， 伊甸园区 (Eden) 占新生代大小的 10/12， 幸存区 From 和幸存区 To 每个占新生代的 1/12 。 注意， 两个幸存区永远是一样大的。

配置之后, 重新启动Tomcat ,访问 :

JVM

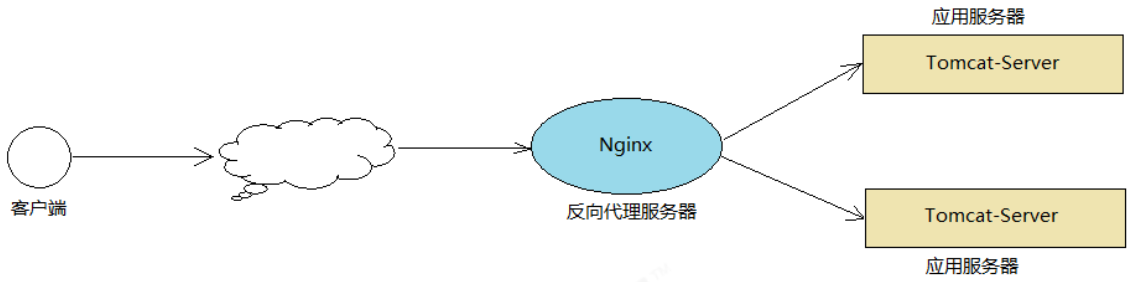
Free Memory: 1553.71 MB Total Memory: 1980.00 MB Max Memory: 1980.00 MB

Memory Pool	Type	Initial	Total	Maximum	Used
PS Eden Space	Heap memory	546.50 MB	546.50 MB	546.50 MB	426.28 MB (78%)
PS Old Gen	Heap memory	1365.50 MB	1365.50 MB	1365.50 MB	0.00 MB (0%)
PS Survivor Space	Heap memory	68.00 MB	68.00 MB	68.00 MB	0.00 MB (0%)
Code Cache	Non-heap memory	2.43 MB	8.25 MB	240.00 MB	8.23 MB (3%)
Compressed Class Space	Non-heap memory	0.00 MB	2.25 MB	1024.00 MB	1.94 MB (0%)
Metaspace	Non-heap memory	0.00 MB	18.25 MB	256.00 MB	17.59 MB (6%)

8. Tomcat 集群

8.1 简介

由于单台Tomcat的承载能力是有限的，当我们的业务系统用户量比较大，请求压力比较大时，单台Tomcat是扛不住的，这个时候，就需要搭建Tomcat的集群，而目前比较流程的做法就是通过Nginx来实现Tomcat集群的负载均衡。



8.2 环境准备

8.2.2 准备Tomcat

在服务器上, 安装两台tomcat, 然后分别改Tomcat服务器的端口号 :

```
8005 ---------> 8015 ---------> 8025

8080 ---------> 8888 ---------> 9999

8009 ---------> 8019 ---------> 8029
```

8.2.3 安装配置Nginx

```
upstream serverpool{
    server localhost:8888;
    server localhost:9999;
}

server {
    listen       99;
    server_name localhost;

    location / {
        proxy_pass http://serverpool/;
    }
}
```

8.3 负载均衡策略

1). 轮询

最基本的配置方法，它是upstream模块默认的负载均衡默认策略。每个请求会按时间顺序逐一分配到不同的后端服务器。

```
upstream serverpool{
    server localhost:8888;
    server localhost:9999;
}
```

参数说明:

参数	描述
fail_timeout	与max_fails结合使用
max_fails	设置在fail_timeout参数设置的时间内最大失败次数，如果在这个时间内，所有针对该服务器的请求都失败了，那么认为该服务器会被认为是停机了
fail_time	服务器会被认为停机的时间长度,默认为10s
backup	标记该服务器为备用服务器。当主服务器停止时，请求会被发送到它这里
down	标记服务器永久停机了

2). weight权重

权重方式，在轮询策略的基础上指定轮询的几率。

```
upstream serverpool{
    server localhost:8888 weight=3;
    server localhost:9999 weight=1;
}
```

此策略比较适合服务器的硬件配置差别比较大的情况。

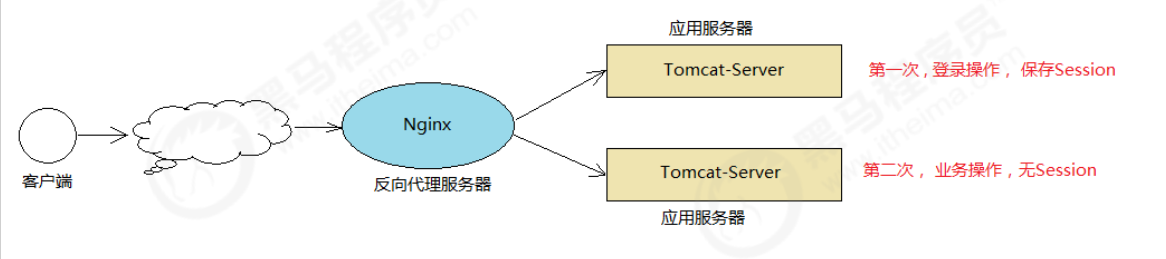
3). ip_hash

指定负载均衡器按照基于客户端IP的分配方式，这个方法确保了相同的客户端的请求一直发送到相同的服务器，以保证session会话。这样每个访客都固定访问一个后端服务器，可以解决session不能跨服务器的问题。

```
upstream serverpool{
    ip_hash;
    server 192.168.192.133:8080;
    server 192.168.192.137:8080;
}
```

8.4 Session共享方案

在Tomcat集群中，如果应用需要用户进行登录，那么这个时候，用于tomcat做了负载均衡，则用户登录并访问应用系统时，就会出现问题 。



解决上述问题， 有以下几种方案：

8.4.1 ip_hash 策略

一个用户发起的请求，只会请求到tomcat1上进行操作，另一个用户发起的请求只在tomcat2上进行操作 。那么这个时候，同一个用户发起的请求，都会通过nginx的ip_hash策略，将请求转发到其中的一台Tomcat上。

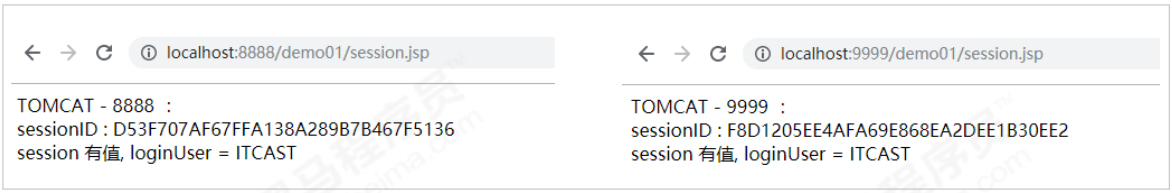
8.4.2 Session复制

在servlet_demo01 工程中，制作session.jsp页面，分别将工程存放在两台 tomcat 的 webapps/ 目录下：

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
```

```
    <br/>
    sessionID : <%= session.getId()%>

    <br/>

    <%
        Object loginUser = session.getAttribute("loginUser");
        if(loginUser != null && loginUser.toString().length()>0){
            out.println("session 有值, loginUser = " + loginUser);
        }else{
            session.setAttribute("loginUser","ITCAST");
            out.println("session 没有值");
        }
    %>

</body>
</html>
```

通过nginx访问 ， http://localhost:99/demo01/session.jsp ，访问到的两台Tomcat出现的sessionID是不一样的：



上述现象，则说明两台Tomcat的Session各是各的，并没有进行同步，这在集群环境下是存在问题的。

Session同步的配置如下：

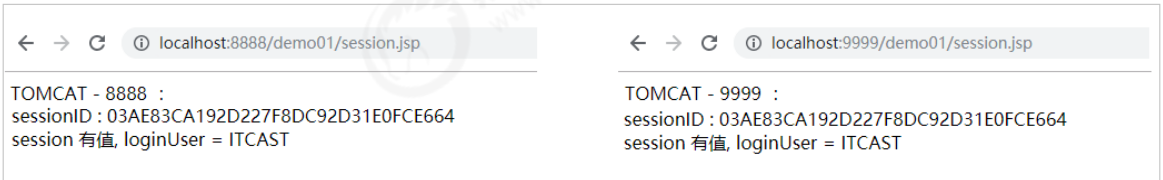
1 ）在Tomcat的conf/server.xml 配置如下:

```
<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/>
```

2 ）在Tomcat部署的应用程序 servlet_demo01 的web.xml 中加入如下配置 ：

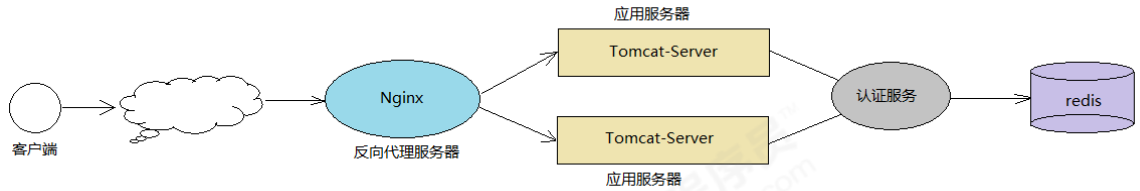
```
<distributable/>
```

3 ）配置完毕之后， 再次重启两个 Tomcat服务。



上述方案，适用于较小的集群环境（节点数不超过4个），如果集群的节点数比较多的话，通过这种广播的形式来完成Session的复制，会消耗大量的网络带宽，影响服务的性能。

单点登录（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统，也是用来解决集群环境Session共享的方案之一 。



9. Tomcat 安全

9.1 配置安全

1）删除webapps目录下的所有文件，禁用tomcat管理界面；

2）注释或删除tomcat-users.xml文件内的所有用户权限；

3）更改关闭tomcat指令或禁用；

tomcat的server.xml中定义了可以直接关闭 Tomcat 实例的管理端口（默认8005）。可以通过 telnet 连接上该端口之后，输入 SHUTDOWN （此为默认关闭指令）即可关闭 Tomcat 实例（注意，此时虽然实例关闭了，但是进程还是存在的）。由于默认关闭Tomcat 的端口和指令都很简单。默认端口为8005，指令为SHUTDOWN 。

方案一：

```
更改端口号和指令：
<Server port="8456" shutdown="itcast_shut">
```

方案二：

```
禁用8005端口：
<Server port="-1" shutdown="SHUTDOWN">
```

4）定义错误页面

在webapps/ROOT目录下定义错误页面 404.html，500.html；

然后在tomcat/conf/web.xml中进行配置 ， 配置错误页面：

```
<error-page>
    <error-code>404</error-code>
    <location>/404.html</location>
</error-page>

<error-page>
    <error-code>500</error-code>
    <location>/500.html</location>
</error-page>
```

9.2 应用安全

在大部分的Web应用中，特别是一些后台应用系统，都会实现自己的安全管理模块（权限模块），用于控制应用系统的安全访问，基本包含两个部分：认证（登录/单点登录）和授权（功能权限、数据权限）两个部分。对于当前的业务系统，可以自己做一套适用于自己业务系统的权限模块，也有很多的应用系统直接使用一些功能完善的安全框架，将其集成到我们的web应用中，如：SpringSecurity、Apache Shiro等。

9.3 传输安全

9.3.1 HTTPS介绍

HTTPS的全称是超文本传输安全协议（Hypertext Transfer Protocol Secure），是一种网络安全传输协议。在HTTP的基础上加入SSL/TLS来进行数据加密，保护交换数据不被泄露、窃取。

SSL 和 TLS 是用于网络通信安全的加密协议，它允许客户端和服务器之间通过安全链接通信。SSL 协议的3个特性：

1）保密：通过SSL链接传输的数据时加密的。

2）鉴别：通信双方的身份鉴别，通常是可选的，单至少有一方需要验证。

3）完整性：传输数据的完整性检查。

从性能角度考虑，加解密是一项计算昂贵的处理，因为尽量不要将整个Web应用采用SSL链接， 实际部署过程中， 选择有必要进行安全加密的页面（存在敏感信息传输的页面）采用SSL通信。

HTTPS和HTTP的区别主要为以下四点：

1）HTTPS协议需要到证书颁发机构CA申请SSL证书, 然后与域名进行绑定，HTTP不用申请证书；

2）HTTP是超文本传输协议，属于应用层信息传输，HTTPS 则是具有SSL加密传安全性传输协议，对数据的传输进行加密，相当于HTTP的升级版；

3）HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是8080，后者是8443。

4）HTTP的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP协议安全。

HTTPS协议优势：

1）提高网站排名，有利于SEO。谷歌已经公开声明两个网站在搜索结果方面相同，如果一个网站启用了SSL，它可能会获得略高于没有SSL网站的等级，而且百度也表明对安装了SSL的网站表示友好。因此，网站上的内容中启用SSL都有明显的SEO优势。

2）隐私信息加密，防止流量劫持。特别是涉及到隐私信息的网站，互联网大型的数据泄露的事件频发发生，网站进行信息加密势在必行。

9.3.2 Tomcat支持HTTPS

1）生成秘钥库文件。

```
keytool -genkey -alias tomcat -keyalg RSA -keystore tomcatkey.keystore
```

```
D:\temp\SSL>keytool -genkey -alias tomcat -keyalg RSA -keystore tomcatkey.keystore
输入密钥库口令:
再次输入新口令:
您的名字与姓氏是什么?
  [Unknown]:  itcast
您的组织单位名称是什么?
  [Unknown]:  itcast
您的组织名称是什么?
  [Unknown]:  itcast
您所在的城市或区域名称是什么?
  [Unknown]:  xian
您所在的省/市/自治区名称是什么?
  [Unknown]:  xian
该单位的双字母国家/地区代码是什么?
  [Unknown]:  CN
CN=itcast, OU=itcast, O=itcast, L=xian, ST=xian, C=CN是否正确?
  [否]:  Y

输入 <tomcat> 的密钥口令
        (如果和密钥库口令相同, 按回车):
再次输入新口令:
```

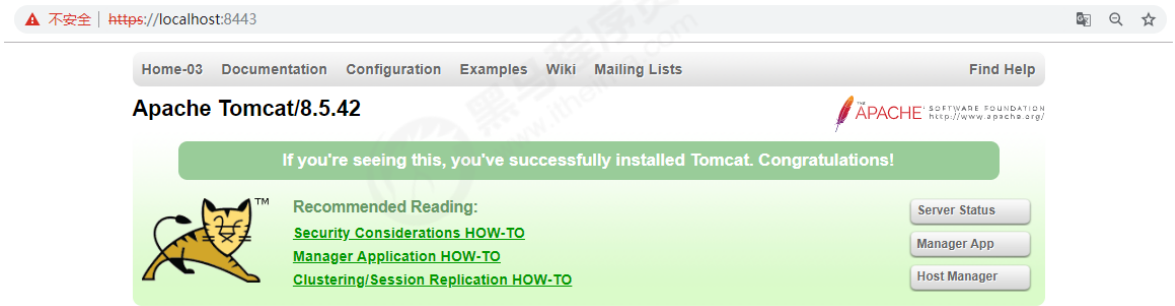
输入对应的密钥库密码， 秘钥密码等信息之后，会在当前文件夹中出现一个秘钥库文件：tomcatkey.keystore

2）将秘钥库文件 tomcatkey.keystore 复制到tomcat/conf 目录下。

3）配置tomcat/conf/server.xml

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
    maxThreads="150" schema="https" secure="true" SSLEnabled="true">
    <SSLHostConfig certificateVerification="false">
        <Certificate certificateKeystoreFile="D:/DevelopProgramFile/apache-
tomcat-8.5.42-windows-x64/apache-tomcat-8.5.42/conf/tomcatkey.keystore"
certificateKeystorePassword="itcast"  type="RSA" />
    </SSLHostConfig>
</Connector>
```

4）访问Tomcat ，使用https协议。



10. Tomcat 性能调优

对于系统性能，用户最直观的感受就是系统的加载和操作时间，即用户执行某项操作的耗时。从更为专业的角度上讲，性能测试可以从以下两个指标量化。

1). 响应时间：如上所述，为执行某个操作的耗时。大多数情况下，我们需要针对同一个操作测试多次，以获取操作的平均响应时间。

2). 吞吐量：即在给定的时间内，系统支持的事务数量，计算单位为 TPS。

通常情况下，我们需要借助于一些自动化工具来进行性能测试，因为手动模拟大量用户的并发访问几乎是不可行的，而且现在市面上也有很多的性能测试工具可以使用，如：ApacheBench、ApacheJMeter、WCAT、WebPolygraph、LoadRunner。

我们课程上主要介绍两款免费的工具：ApacheBench。

10.1.1 ApacheBench

ApacheBench（ab）是一款ApacheServer基准的测试工具，用户测试Apache Server的服务能力（每秒处理请求数），它不仅可以用户Apache的测试，还可以用于测试Tomcat、Nginx、lighthttp、IIS等服务器。

1）安装

```
yum install httpd-tools
```

2）查看版本号

```
ab -V
```

```
[root@localhost ~]# ab -V
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
```

3）部署war包，准备环境

```
A. 在Linux系统上安装Tomcat

    上传 ：  alt + p -------> put D:/apache-tomcat-8.5.42.tar.gz

    解压 ：  tar -zxvf apache-tomcat-8.5.42.tar.gz -C /usr/local

    修改端口号：8005 ，  8080 ，  8009

B. 将资料中的war包上传至Tomcat的webapps下

    上传：  alt + p ---------> put  D:/ROOT.war

    启动Tomcat解压

C. 导入SQL脚本 ，  准备环境
```

4）测试性能

```
http://localhost:9000/course/search.do?page=1&pageSize=10
```

参数说明

参数	含义描述
-n	在测试会话中所执行的请求个数，默认只执行一次请求
-c	一次产生的请求个数，默认一次一个
-p	包含了需要POST的数据文件
-t	测试所进行的最大秒数，默认没有时间限制
-T	POST数据所需要使用的Content-Type头信息
-v	设置显示信息的详细程度
-w	以HTML表的格式输出结果，默认是白色背景的两列宽度的一张表

结果说明

Server Software	服务器软件
Server Hostname	主机名
Server Port	端口号
Document Path	测试的页面
Document Length	测试的页面大小
Concurrency Level	并发数
Time taken for tests	整个测试持续的时间
Complete requests	完成的请求数量
Failed requests	失败的请求数量，这里的失败是指请求的连接服务器、发送数据、接收数据等环节发生异常，以及无响应后超时的情况。
Write errors	输出错误数量
Total transferred	整个场景中的网络传输量，表示所有请求的响应数据长度总和，包括每个http响应数据的头信息和正文数据的长度。
HTML transferred	整个场景中的HTML内容传输量，表示所有请求的响应数据中正文数据的总和
Requests per second	每秒钟平均处理的请求数（相当于 LR 中的 每秒事务数）这便是我们重点关注的吞吐率，它等于：Complete requests / Time taken for tests
Time per request	每个线程处理请求平均消耗时间（相当于 LR 中的 平均事务响应时间）用户平均请求等待时间
Transfer rate	平均每秒网络上的流量
Percentage of the requests served within a certain time (ms)	指定时间里，执行的请求百分比

重要指标

Requests per second	**吞吐率**:服务器并发处理能力的量化描述，单位是reqs/s，指的是在某个并发用户数下单位时间内处理的请求数。某个并发用户数下单位时间内能处理的最大请求数，称之为最大吞吐率。 这个数值表示当前机器的整体性能，值越大越好。
Time per request	**用户平均请求等待时间**：从用户角度看，完成一个请求所需要的时间
Time per request:across all concurrent requests	**服务器平均请求等待时间**：服务器完成一个请求的时间
Concurrency Level	并发用户数

10.2 Tomcat 性能优化

10.2.1 JVM参数调优

Tomcat是一款Java应用，那么JVM的配置便与其运行性能密切相关，而JVM优化的重点则集中在内存分配和GC策略的调整上，因为内存会直接影响服务的运行效率和吞吐量，JVM垃圾回收机制则会不同程度地导致程序运行中断。可以根据应用程序的特点，选择不同的垃圾回收策略，调整JVM垃圾回收策略，可以极大减少垃圾回收次数，提升垃圾回收效率，改善程序运行性能。

1）JVM内存参数

参数	参数作用	优化建议
-server	启动Server，以服务端模式运行	服务端模式建议开启
-Xms	最小堆内存	建议与-Xmx设置相同
-Xmx	最大堆内存	建议设置为可用内存的80%
-XX:MetaspaceSize	元空间初始值	
-XX:MaxMetaspaceSize	元空间最大内存	默认无限
-XX:MaxNewSize	新生代最大内存	默认16M
-XX:NewRatio	年轻代和老年代大小比值，取值为整数，默认为2	不建议修改
-XX:SurvivorRatio	Eden区与Survivor区大小的比值，取值为整数，默认为8	不建议修改

```
JAVA_OPTS="-server -Xms2048m -Xmx2048m -XX:MetaspaceSize=256m -XX:MaxMetaspaceSize=512m  -XX:SurvivorRatio=8"
```

2 ） GC策略

JVM垃圾回收性能有以下两个主要的指标:

- 吞吐量：工作时间（排除GC时间）占总时间的百分比， 工作时间并不仅是程序运行的时间，还包含内存分配时间。
- 暂停时间：测试时间段内，由垃圾回收导致的应用程序停止响应次数/时间。

在Sun公司推出的HotSpotJVM中， 包含以下几种不同类型的垃圾收集器：

垃圾收集器	含义说明
串行收集器（Serial Collector）	采用单线程执行所有的垃圾回收工作， 适用于单核CPU服务器，无法利用多核硬件的优势
并行收集器（Parallel Collector）	又称为吞吐量收集器， 以并行的方式执行年轻代的垃圾回收， 该方式可以显著降低垃圾回收的开销(指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态)。适用于多处理器或多线程硬件上运行的数据量较大的应用
并发收集器（Concurrent Collector）	以并发的方式执行大部分垃圾回收工作，以缩短垃圾回收的暂停时间。适用于那些响应时间优先于吞吐量的应用， 因为该收集器虽然最小化了暂停时间(指用户线程与垃圾收集线程同时执行,但不一定是并行的，可能会交替进行)， 但是会降低应用程序的性能
CMS收集器（Concurrent Mark Sweep Collector）	并发标记清除收集器， 适用于那些更愿意缩短垃圾回收暂停时间并且负担的起与垃圾回收共享处理器资源的应用
G1收集器（Garbage-First Garbage Collector）	适用于大容量内存的多核服务器， 可以在满足垃圾回收暂停时间目标的同时，以最大可能性实现高吞吐量(JDK1.7之后)

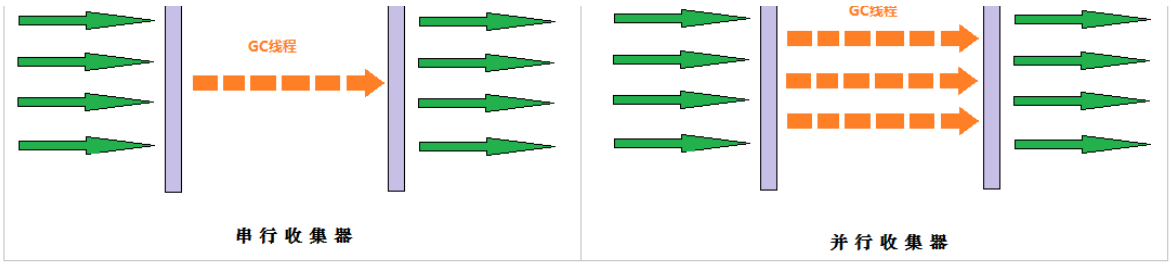
不同的应用程序， 对于垃圾回收会有不同的需求。 JVM 会根据运行的平台、服务器资源配置情况选择合适的垃圾收集器、堆内存大小及运行时编译器。如无法满足需求， 参考以下准则：

A. 程序数据量较小，选择串行收集器。

B. 应用运行在单核处理器上且没有暂停时间要求， 可交由JVM自行选择或选择串行收集器。

C. 如果考虑应用程序的峰值性能， 没有暂停时间要求， 可以选择并行收集器。

D. 如果应用程序的响应时间比整体吞吐量更重要， 可以选择并发收集器。



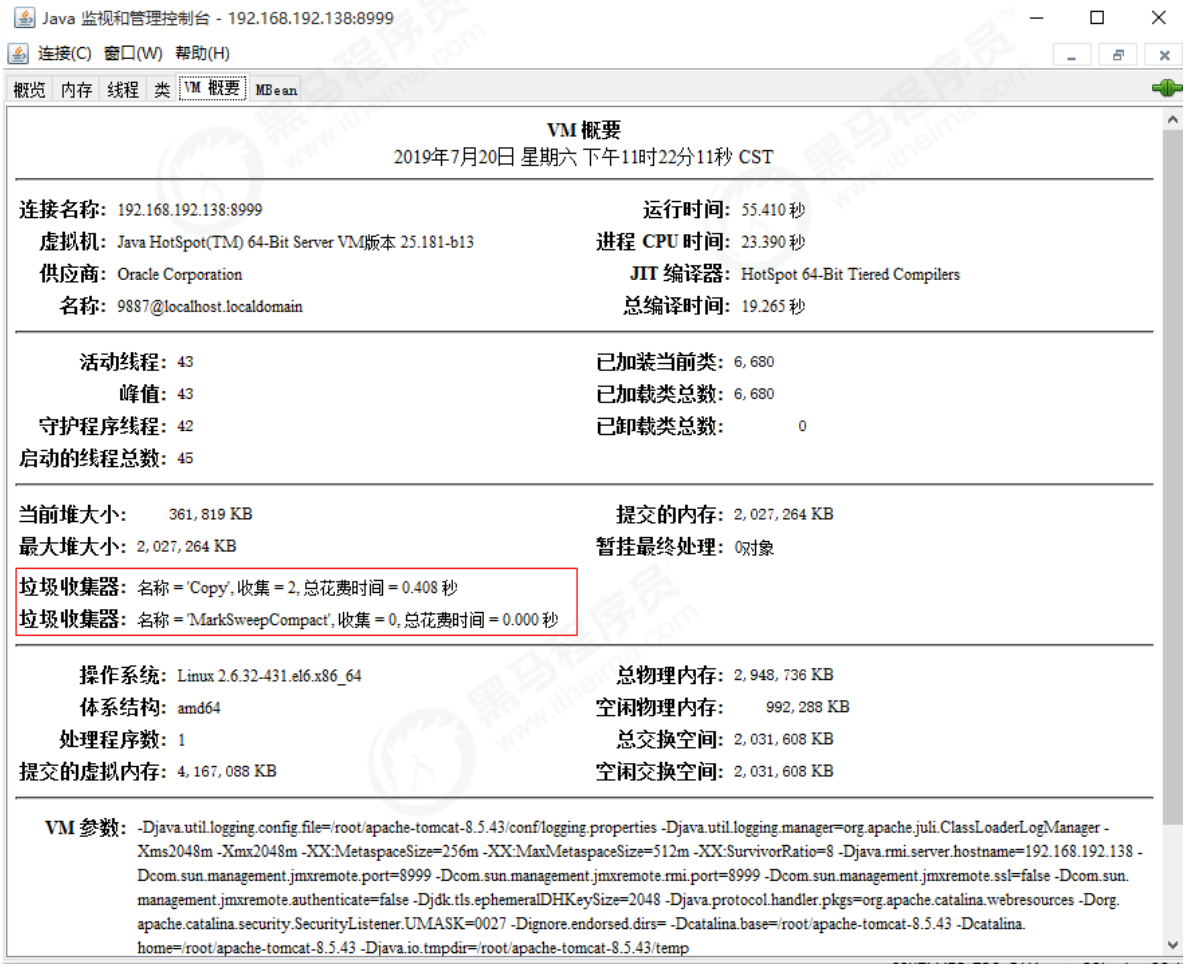
查看Tomcat中的默认的垃圾收集器:

1). 在tomcat/bin/catalina.sh的配置中, 加入如下配置

```
JAVA_OPTS=" -Djava.rmi.server.hostname=192.168.192.138 -
Dcom.sun.management.jmxremote.port=8999 -
Dcom.sun.management.jmxremote.rmi.port=8999 -
Dcom.sun.management.jmxremote.ssl=false -
Dcom.sun.management.jmxremote.authenticate=false"
```

2). 打开 jconsole , 查看远程的tomcat的概要信息

连接远程tomcat



GC参数：

参数	描述
-XX:+UseSerialGC	启用串行收集器
-XX:+UseParallelGC	启用并行垃圾收集器，配置了该选项，那么 -XX:+UseParallelOldGC默认启用
-XX:+UseParallelOldGC	FullGC 采用并行收集，默认禁用。如果设置了 -XX:+UseParallelGC则自动启用
-XX:+UseParNewGC	年轻代采用并行收集器，如果设置了 -XX:+UseConcMarkSweepGC选项，自动启用
-XX:ParallelGCThreads	年轻代及老年代垃圾回收使用的线程数。默认值依赖于JVM使用的CPU个数
-XX:+UseConcMarkSweepGC	对于老年代，启用CMS垃圾收集器。 当并行收集器无法满足应用的延迟需求是，推荐使用CMS或G1收集器。 启用该选项后， -XX:+UseParNewGC 自动启用。
-XX:+UseG1GC	启用G1收集器。 G1是服务器类型的收集器， 用于多核、大内存的机器。它在保持高吞吐量的情况下，高概率满足GC暂停时间的目标。

我们也可以在测试的时候，将JVM参数调整之后，将GC的信息打印出来，便于为我们进行参数调整提供依据，具体参数如下：

选项	描述
-XX:+PrintGC	打印每次GC的信息
-XX:+PrintGCApplicationConcurrentTime	打印最后一次暂停之后所经过的时间， 即响应并发执行的时间
-XX:+PrintGCApplicationStoppedTime	打印GC时应用暂停时间
-XX:+PrintGCDateStamps	打印每次GC的日期戳
-XX:+PrintGCDetails	打印每次GC的详细信息
-XX:+PrintGCTaskTimeStamps	打印每个GC工作线程任务的时间戳
-XX:+PrintGCTimeStamps	打印每次GC的时间戳

在bin/catalina.sh的脚本中，追加如下配置：

```
JAVA_OPTS="-XX:+UseConcMarkSweepGC  -XX:+PrintGCDetails"
```

10.2.2 Tomcat 配置调优

调整tomcat/conf/server.xml 中关于链接器的配置可以提升应用服务器的性能。

maxConnections	最大连接数，当到达该值后，服务器接收但不会处理更多的请求， 额外的请求将会阻塞直到连接数低于maxConnections 。可通过ulimit -a 查看服务器限制。对于CPU要求更高(计算型)时，建议不要配置过大; 对于CPU要求不是特别高时，建议配置在2000左右(受服务器性能影响)。 当然这个需要服务器硬件的支持
maxThreads	最大线程数,需要根据服务器的硬件情况，进行一个合理的设置
acceptCount	最大排队等待数,当服务器接收的请求数量到达maxConnections ，此时Tomcat会将后面的请求，存放在任务队列中进行排序， acceptCount指的就是任务队列中排队等待的请求数 。 一台Tomcat的最大的请求处理数量，是maxConnections+acceptCount。

11. Tomcat 附加功能

11.1 WebSocket

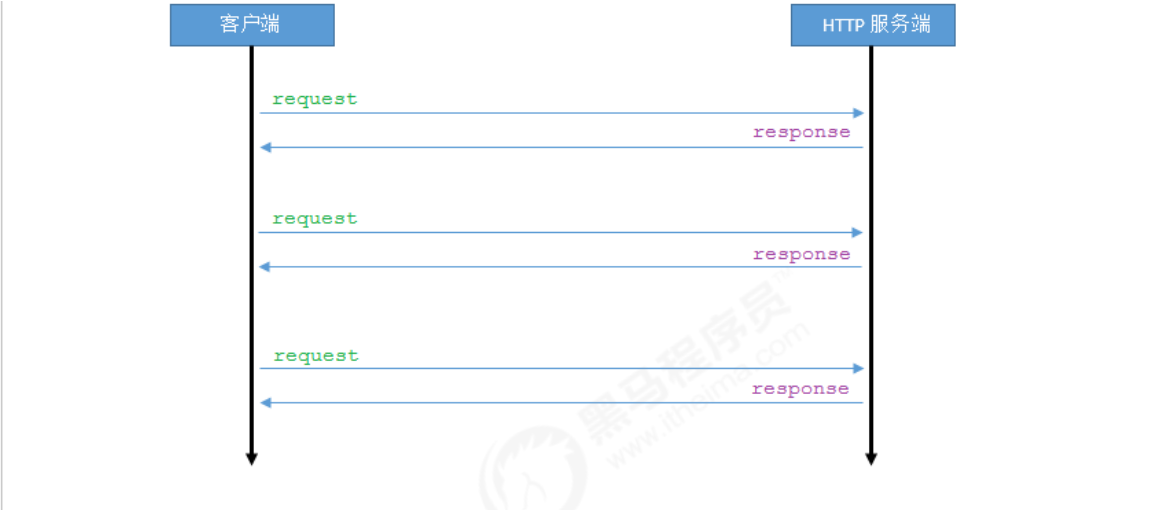
11.1.1 WebSocket介绍

WebSocket是HTML5新增的协议，它的目的是在浏览器和服务器之间建立一个不受限的双向通信的通道，比如说，服务器可以在任意时刻发送消息给浏览器。

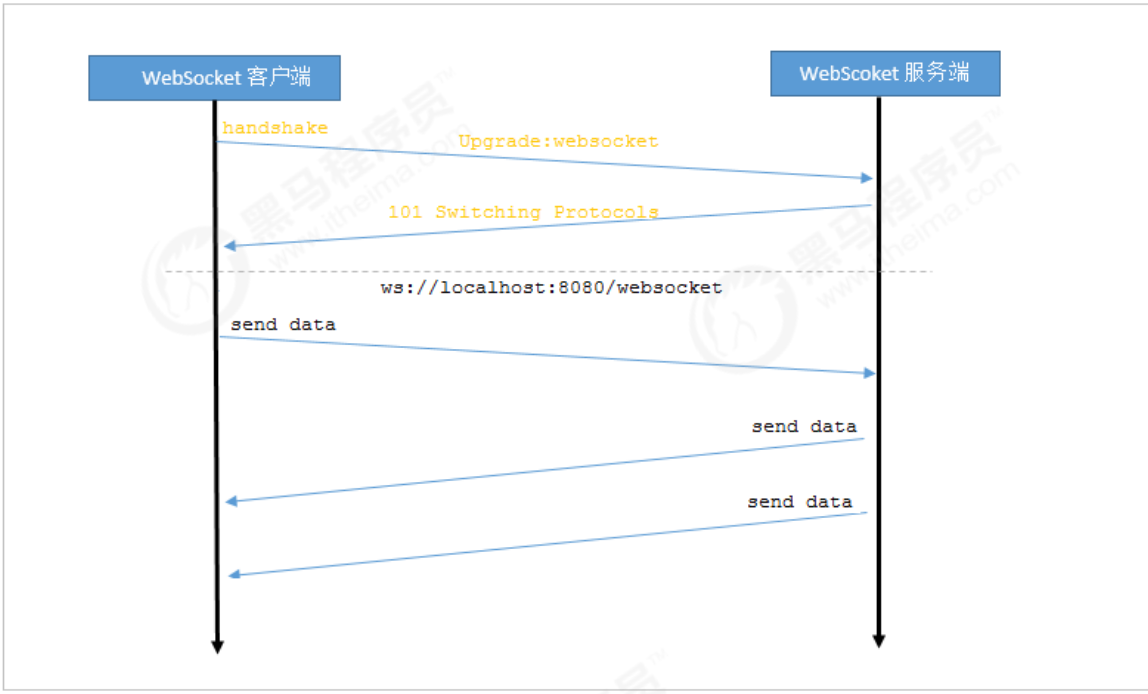
为什么传统的HTTP协议不能做到WebSocket实现的功能？这是因为HTTP协议是一个请求 - 响应协议，请求必须先由浏览器发给服务器，服务器才能响应这个请求，再把数据发送给浏览器。换句话说，浏览器不主动请求，服务器是没法主动发数据给浏览器的。

这样一来，要在浏览器中搞一个实时聊天，或者在线多人游戏的话就没法实现了，只能借助Flash这些插件。也有人说，HTTP协议其实也能实现啊，比如用轮询或者Comet。轮询是指浏览器通过JavaScript启动一个定时器，然后以固定的间隔给服务器发请求，询问服务器有没有新消息。这个机制的缺点一是实时性不够，二是频繁的请求会给服务器带来极大的压力。

Comet本质上也是轮询，但是在没有消息的情况下，服务器先拖一段时间，等到有消息了再回复。这个机制暂时地解决了实时性问题，但是它带来了新的问题：以多线程模式运行的服务器会让大部分线程大部分时间都处于挂起状态，极大地浪费服务器资源。另外，一个HTTP连接在长时间没有数据传输的情况下，链路上的任何一个网关都可能关闭这个连接，而网关是我们不可控的，这就要求Comet连接必须定期发一些ping数据表示连接“正常工作”。



以上两种机制都治标不治本，所以，HTML5推出了WebSocket标准，让浏览器和服务器之间可以建立无限制的全双工通信，任何一方都可以主动发消息给对方。WebSocket并不是全新的协议，而是利用了HTTP协议来建立连接。我们来看看WebSocket连接是如何创建的。



首先，WebSocket连接必须由浏览器发起，因为请求协议是一个标准的HTTP请求，格式如下：

```
Status Code: 101
Response Headers    view source
  Connection: upgrade
  Date: Sun, 21 Jul 2019 09:27:37 GMT
  Sec-WebSocket-Accept: 2geCfi8VX1YkfCbhTR/ZmOM49DY=
  Sec-WebSocket-Extensions: permessage-deflate;client_max_window_bits=15
  Upgrade: websocket
Request Headers    view source
  Accept-Encoding: gzip, deflate, br
  Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
  Cache-Control: no-cache
  Connection: Upgrade
  Cookie: JSESSIONID=9D9371F7858CA4459652B5C47540670B
  Host: localhost:8080
  Origin: http://localhost:8080
  Pragma: no-cache
  Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
  Sec-WebSocket-Key: hmC1XSpyKVPzEsAD64qXWw==
  Sec-WebSocket-Version: 13
  Upgrade: websocket
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.131 Safari/537.36
```

该请求和普通的HTTP请求有几点不同：

1. GET请求的地址不是类似 http://，而是以 ws:// 开头的地址；
2. 请求头 Connection: Upgrade 和 请求头 Upgrade: websocket 表示这个连接将要被转换为 WebSocket 连接；
3. Sec-WebSocket-Key 是用于标识这个连接， 是一个BASE64编码的密文, 要求服务端响应一个对应加密的Sec-WebSocket-Accept头信息作为应答；
4. Sec-WebSocket-Version 指定了WebSocket的协议版本;
5. HTTP101 状态码表明服务端已经识别并切换为WebSocket协议，Sec-WebSocket-Accept是服务端与客户端一致的秘钥计算出来的信息。

11.2.2 Tomcat的 Websocket

Tomcat的7.0.5 版本开始支持WebSocket,并且实现了Java WebSocket规范(JSR356), 而在7.0.5版本之前(7.0.2之后)则采用自定义API, 即WebSocketServlet实现。

Java WebSocket应用由一系列的WebSocketEndpoint组成。Endpoint 是一个java对象，代表WebSocket链接的一端，对于服务端，我们可以视为处理具体WebSocket消息的接口， 就像Servlet之与http请求一样。

我们可以通过两种方式定义Endpoint:

1). 第一种是编程式， 即继承类 javax.websocket.Endpoint并实现其方法。

2). 第二种是注解式, 即定义一个POJO, 并添加 @ServerEndpoint相关注解。

Endpoint实例在WebSocket握手时创建，并在客户端与服务端链接过程中有效，最后在链接关闭时结束。在Endpoint接口中明确定义了与其生命周期相关的方法， 规范实现者确保生命周期的各个阶段调用实例的相关方法。生命周期方法如下：

onClose	当会话关闭时调用。	@OnClose
onOpen	当开启一个新的会话时调用, 该方法是客户端与服务端握手成功后调用的方法。	@OnOpen
onError	当连接过程中异常时调用。	@OnError

通过为Session添加MessageHandler消息处理器来接收消息，当采用注解方式定义Endpoint时，我们还可以通过 @OnMessage 注解指定接收消息的方法。发送消息则由RemoteEndpoint 完成， 其实例由Session维护， 根据使用情况， 我们可以通过Session.getBasicRemote获取同步消息发送的实例 ， 然后调用其sendXxx()方法就可以发送消息， 可以通过Session.getAsyncRemote 获取异步消息发送实例。

11.2.3 WebSocket DEMO案例

11.2.3.1 需求

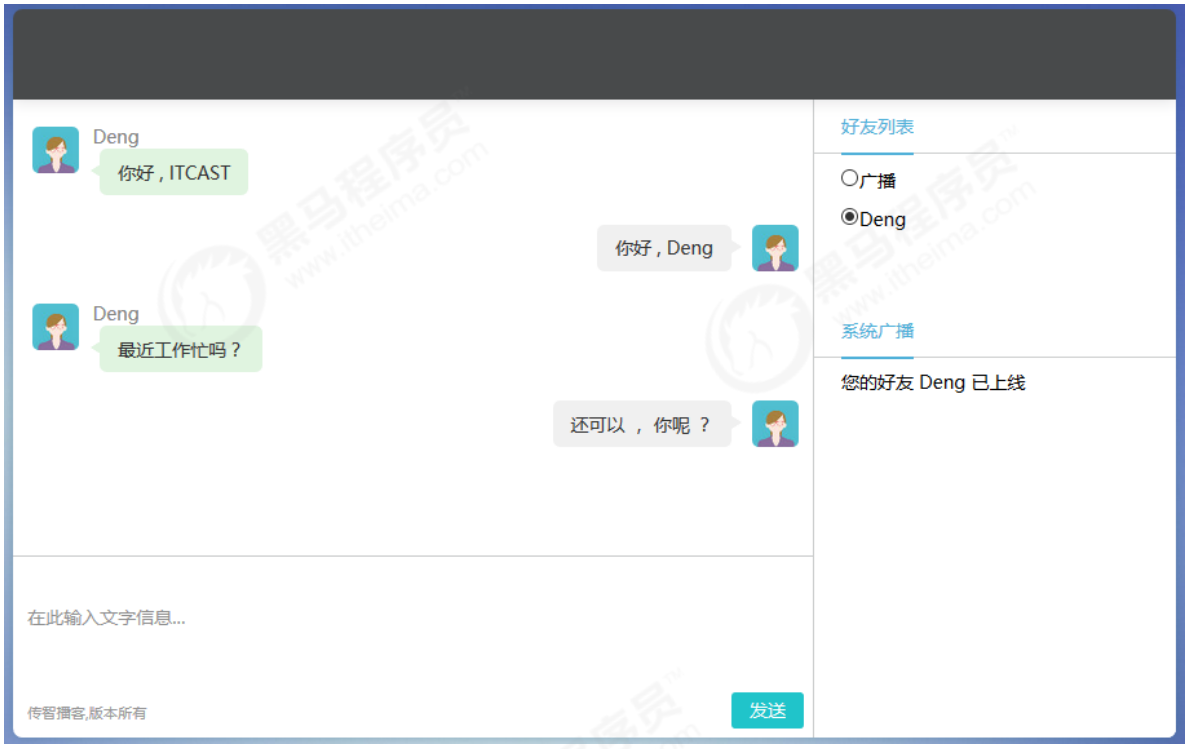
通过 websocket 实现一个简易的聊天室功能；

1). 登录聊天室

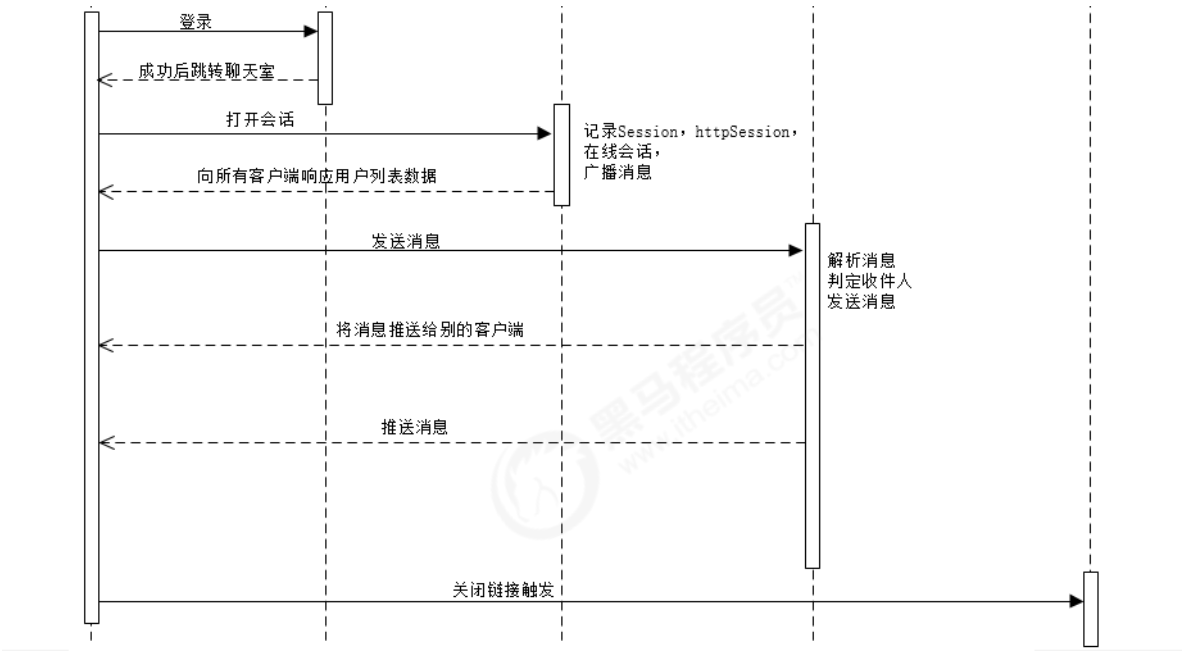
2) 登陆之后，进入聊天界面进行聊天

用户 Deng 的界面：

用户 ICAST 的界面：



11.2.3.2 实现流程



11.2.3.3 消息格式

客户端-->服务端 ： {"fromName":"Deng","toName":"HEIMA","content":"约会呀"}

服务端-->客户端 ：

①. 如果type为user ， 则说明返回的是用户列表

{"data":"HEIMA,Deng,ITCAST","toName":"","fromName":"","type":"user"}

②. 如果type为message ， 则说明返回的是消息内容

{"data":"你好","toName":"HEIMA","fromName":"Deng","type":"message"}

11.2.3.4 功能实现

1 ）创建项目， 导入项目依赖。

fastjson-1.2.5.jar	Executable Jar File	402 KB
tomcat-websocket.jar	Executable Jar File	222 KB
websocket-api.jar	Executable Jar File	38 KB

2) 引入静态资源文件。

css	文件夹	
img	文件夹	
js	文件夹	
chat.jsp	JSP 文件	5 KB
login.jsp	JSP 文件	3 KB

3 ）定义一个登陆的servlet

```
@WebServlet(name = "loginServlet",urlPatterns = {"/login"})
public class LoginServlet  extends HttpServlet {

    private static final String PASSWORD = "123456";

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        doPost(req,resp);
    }


    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.setCharacterEncoding("UTF-8");

        String username = req.getParameter("username");
        String password = req.getParameter("password");

        Map resultMap = new HashMap();
        if(password != null && password.equals(PASSWORD)){

            req.getSession().setAttribute("username",username);

            resultMap.put("success",true);
            resultMap.put("message","登录成功");
        }else{
            resultMap.put("success",false);
            resultMap.put("message","登录失败");
        }

        resp.getWriter().write(JSON.toJSONString(resultMap));
    }

}
```

4）定义配置类，便于在WS中获取HttpSession

```
/**
 * 获取HttpSession对象的配置类
 */
public class GetHttpSessionConfigurator extends
ServerEndpointConfig.Configurator {

    /**
     * 获取到HttpSession ，并将其存储在 ServerEndpointConfig对象中.
     * @param config
     * @param request
     * @param response
     */
    @Override
    public void modifyHandshake(ServerEndpointConfig config, HandshakeRequest
request, HandshakeResponse response) {
```

```
httpSession);

    }

}
```

5）定义WebSocket的服务端程序

```
@ServerEndpoint(value = "/websocket", configurator =
GetHttpSessionConfigurator.class)
public class ChatServlet {


    //该map集合用来存储所有在线用户的实例信息
    private static final Map<HttpSession, ChatServlet> onlineUsers = new
HashMap<HttpSession, ChatServlet>();

    //记录在线用户数
    private static int onlineCount = 0;

    //用户的HttpSession
    private HttpSession httpSession;

    //用户的WS的会话信息Session
    private Session session;

    /**
     *  @onOpen :当开启一个新的会话时调用， 该方法是客户端与服务端握手成功后调用的方法。
     *
     *  为当前Servlet中的Session赋值 ， 为HttpSession赋值， 将当前的会话信息， 记录在在线
用户集合中 ;
     *
     *  获取到当前所有在线的用户信息 ， 并且给所有的WS客户端推送消息， 在客户端更新好友列表。
     *  在线用户数增加1
     *
     */
    @OnOpen
    public void onOpen(Session session, EndpointConfig config) {

        this.session = session;
        this.httpSession = (HttpSession)
config.getUserProperties().get(HttpSession.class.getName());
        if (httpSession.getAttribute("username") != null) {
            onlineUsers.put(httpSession, this);
        }
        String names = getNames();
        String content = MessageUtil.getContent(MessageUtil.TYPE_USER, "", "",
names);

        System.out.println("服务端给客户端广播消息:  "+ content);
        broadcastAll(content);
        addOnlineCount();            //在线数加1
        System.out.println("有新连接加入!当前在线人数为" + onlineUsers.size());
```

```
    @OnClose
    public void onClose(Session session, CloseReason closeReason) {
        //onlineUsers.remove(this);  //从set中删除
        subOnlineCount();           //在线数减1
        System.out.println("有一连接关闭！当前在线人数为" + getOnlineCount());
    }


    /**
     *  接收客户端传递的消息，并且根据消息中的toName判定当前消息给那个客户端发送
     * @param message
     * @param session
     * @throws Exception
     */
    @OnMessage
    public void onMessage(String message, Session session) throws Exception {

        //获取客户端发送的消息,并解析
        Map<String, String> messageMap = JSON.parseObject(message, Map.class);
        String fromName = messageMap.get("fromName");    //消息来自人 的userId
        String toName = messageMap.get("toName");       //消息发往人的 userId
        String mapContent = messageMap.get("content");

        //判断是否有接收人
        if (toName == null || toName.isEmpty()) {
            return;
        }

        //如果接收人是 all ，则说明是广播消息
        if ("all".equals(toName)) {
            String content = MessageUtil.getContent(MessageUtil.TYPE_MESSAGE,
fromName, "all", mapContent);
            broadcastAll(content);
        } else {
            //如果不是all ，则给指定用户推送消息
            try {
                String content =
MessageUtil.getContent(MessageUtil.TYPE_MESSAGE, fromName, toName, mapContent);
                System.out.println("服务端给客户端推消息:  "+ content);
                singleChat(fromName, toName, content);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        System.out.println("来自客户端的消息:" + message);
        //broadcastAll(message);
    }


    private void singleChat(String fromName, String toName, String mapContent)
throws IOException {
        boolean isExit = false;
        //判定收件人是否存在
        for (HttpSession key : onlineUsers.keySet()) {
```

```
            }
        }
        //如果存在则，发送消息
        if (isExit) {
            for (HttpSession key : onlineUsers.keySet()) {
                if (key.getAttribute("username").equals(fromName) ||
key.getAttribute("username").equals(toName)) {

 onlineUsers.get(key).session.getBasicRemote().sendText(mapContent);
                }
            }
        }
    }


    //发送广播消息
    private void broadcastAll(String msg) {
        for (HttpSession key : onlineUsers.keySet()) {
            try {
                onlineUsers.get(key).session.getBasicRemote().sendText(msg);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    @OnError
    public void onError(Session session, Throwable error) {
        error.printStackTrace();
        System.out.println("发生错误");
    }

    //获取所有当前在线的用户
    private String getNames() {
        String names = "";
        for (HttpSession key : onlineUsers.keySet()) {
            String name = (String) key.getAttribute("username");
            names += name + ",";
        }
        String namesTemp = names.substring(0, names.length() - 1);
        return namesTemp;
    }


    public static synchronized int getOnlineCount() {
        return onlineCount;
    }

    public static synchronized void addOnlineCount() {
        ChatServlet.onlineCount++;
    }

    public static synchronized void subOnlineCount() {
        ChatServlet.onlineCount--;
    }
}
```