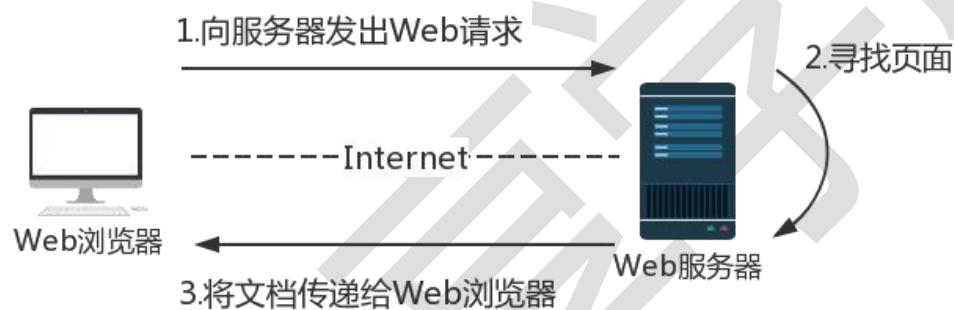


第一节 Tomcat 体系架构

什么是 Web 服务器？

Web 服务器的定义

其实并没有标准定义。一般认为，Web 服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的程序，可以向浏览器等 Web 客户端提供文档，也可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。。



WEB 服务器的介绍

Web 服务器的特点。

- 1、服务器是一种被动程序：只有当 Internet 上运行其他计算机中的浏览器发出的请求时，服务器才会响应。
- 2、服务器一般使用 HTTP（超文本传输协议）与客户机浏览器进行信息交流，这就是人们常把它们称为 HTTP 服务器的原因。
- 3、Web 服务器不仅能够存储信息，还能在用户通过 Web 浏览器提供的信息的基础上运行脚本和程序。

什么是 Tomcat

Tomcat 是一款开源轻量级 Web 应用服务器，是一款优秀的 Servlet 容器实现。

Servlet (Server Applet) 是 Java Servlet 的简称，称为小服务程序或服务连接器，用 Java 编写的服务器端程序，具有独立于平台和协议的特性，主要功能在于交互式地浏览和生成数据，生成动态 Web 内容。

Servlet 严格来讲是指 Java 语言实现的一个接口，一般情况下我们说的 Servlet 是指任何实现了这个 Servlet 接口的类。

- 实例化并调用 init() 方法初始化该 Servlet，一般 Servlet 只初始化一次(只有一个对象)
- service() (根据请求方法不同调用 doGet() 或者 doPost()，此外还有 doHead()、doPut()、doTrace()、doDelete()、doOptions()、destroy())。
- 当 Server 不再需要 Servlet 时 (一般当 Server 关闭时)，Server 调用 Servlet 的 destroy() 方法。

典型的 Servlet 的处理流程

1. 第一个到达服务器的 HTTP 请求被委派到 Servlet 容器。
2. Servlet 容器在调用 service() 方法之前加载 Servlet。
3. 然后 Servlet 容器处理由多个线程产生的多个请求，每个线程执行一个单一的 Servlet 实例的 service() 方法。

Tomcat 版本介绍

Tomcat版本对照

	6.X	7.X	8.X	8.5X	9.X
JDK	>=1.5	>=1.6	>=1.7	>=1.7	>=1.8
Servlet	2.5	3.0	3.1	3.1	4.0
JSP	2.1	2.2	2.3	2.3	2.3
WebSocket	N	1.1	1.1	1.1	1.1

Servlet2.X: 项目目录结构必须要有 WEB-INF, web.xml 等文件夹和文件, 在 web.xml 中配置 servlet,filter,listener, 以 web.xml 为 java web 项目的统一入口

servlet 3.x 规范: 项目中可以不需要 WEB-INF, web.xml 等文件夹和文件, 在没有 web.xml 文件的情况下, 通过注解实现 servlet, filter, listener 的声明, 当使用注解时, 容器自动进行扫描。

同时 Tomcat8.5 进行了大量的代码重构, 对比与 7.0 的版本, 也符合 Tomcat 未来的代码架构体系。但是 Tomcat 的核心和主体架构还是一直保持这样的。

8.5 版本特点

支持 Servlet3.1

默认采用 NIO, 移除 BIO

支持 NIO2(AIO)

支持 HTTP/2 协议

默认采用异步日志处理

为什么要讲 8.5 的版本，首先这个版本比较新，因为太老的版本比如 6.0 的版本 Servlet 不支持 3 所以会导致部署 SpringBoot 等项目有问题，同时这个版本是在 9.0 出现以后发布的一个中间版本，主体架构延续 8.0，同时又实现了部分 9.0 的新特性。

Tomcat 启动

Tomcat 下载地址:

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

Binary Distributions

- Core:
 - [zip \(pgp, sha512\)](#)
 - [tar.gz \(pgp, sha512\)](#)
 - [32-bit Windows zip \(pgp, sha512\)](#)
 - [64-bit Windows zip \(pgp, sha512\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)
- Full documentation:
 - [tar.gz \(pgp, sha512\)](#)
- Deployer:
 - [zip \(pgp, sha512\)](#)
 - [tar.gz \(pgp, sha512\)](#)
- Extras:
 - [JMX Remote jar \(pgp, sha512\)](#)
 - [Web services jar \(pgp, sha512\)](#)
- Embedded:
 - [tar.gz \(pgp, sha512\)](#)
 - [zip \(pgp, sha512\)](#)

符合linux的压缩格式

windows的，一般我们下zip压缩包即可

源码

Source Code Distributions

- [tar.gz \(pgp, sha512\)](#)
- [zip \(pgp, sha512\)](#)

一般启动

startup.bat/sh

启动成功的日志信息如下：

```
Tomcat - □ ×  
tractHandlerMethodMapping.java:543) - Mapped "{[/pushnews]}, produces=[text/html;charset=UTF-8]}" onto public java.lang.String com.xiangxue.servlet3.PushNewsController.news()  
10:16:41:801] [INFO] - org.springframework.web.servlet.handler.AbstractHandlerMethodMapping$MappingRegistry.register(AbstractHandlerMethodMapping.java:543) - Mapped "{[/needPrice]}, produces=[text/event-stream;charset=UTF-8]}" onto public java.lang.String com.xiangxue.sse.NobleMetalController.push()  
10:16:41:801] [INFO] - org.springframework.web.servlet.handler.AbstractHandlerMethodMapping$MappingRegistry.register(AbstractHandlerMethodMapping.java:543) - Mapped "{[/nobleMetal]}" onto public java.lang.String com.xiangxue.sse.NobleMetalController.stock()  
10:16:41:801] [INFO] - org.springframework.web.servlet.handler.AbstractHandlerMethodMapping$MappingRegistry.register(AbstractHandlerMethodMapping.java:543) - Mapped "{[/index]}" onto public java.lang.String com.xiangxue.web.IndexController.hello()  
10:16:41:832] [INFO] - org.springframework.web.servlet.handler.AbstractUrlHandlerMapping.registerHandler(AbstractUrlHandlerMapping.java:362) - Mapped URL path [/assets/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]  
10:16:42:050] [INFO] - org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.initControllerAdviceCache(RequestMappingHandlerAdapter.java:534) - Looking for @ControllerAdvice: WebApplicationContext for namespace 'dispatcher-servlet': startup date [Thu Jul 18 10:16:41 CST 2019]; root of context hierarchy  
10:16:42:174] [INFO] - org.springframework.web.servlet.FrameworkServlet.initServletBean(FrameworkServlet.java:508) - FrameworkServlet 'dispatcher' : initialization completed in 957 ms  
8-Jul-2019 10:16:42.174 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [D:\tools\apache-tomcat-8.5.34\webapps\ref-comet] has finished in [2,001] ms  
8-Jul-2019 10:16:42.174 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deploying web application directory [D:\tools\apache-tomcat-8.5.34\webapps\ROOT]  
8-Jul-2019 10:16:42.202 信息 [localhost-startStop-1] org.apache.catalina.startup.HostConfig.deployDirectory Deployment of web application directory [D:\tools\apache-tomcat-8.5.34\webapps\ROOT] has finished in [28] ms  
8-Jul-2019 10:16:42.208 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]  
8-Jul-2019 10:16:42.215 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]  
8-Jul-2019 10:16:42.218 信息 [main] org.apache.catalina.startup.Catalina.start Server startup in 7259 ms
```

启动成功可以访问到的 Tomcat 首页

<http://localhost:8080/>

Apache Tomcat/8.5.34



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](#)

Managing Tomcat

For security, access to the [manager](#) webapp is restricted. Users are defined in:

\$CATALINA_HOME/conf/tomcat-users.xml

In Tomcat 8.5 access to the manager application is split between different users.

Documentation

[Tomcat 8.5 Documentation](#)

[Tomcat 8.5 Configuration](#)

[Tomcat Wiki](#)

Find additional important configuration information in:

Getting Help

[FAQ and Mailing Lists](#)

The following mailing lists are available:

[tomcat-announce](#)

Important announcements, releases, security vulnerability notifications. (Low volume).

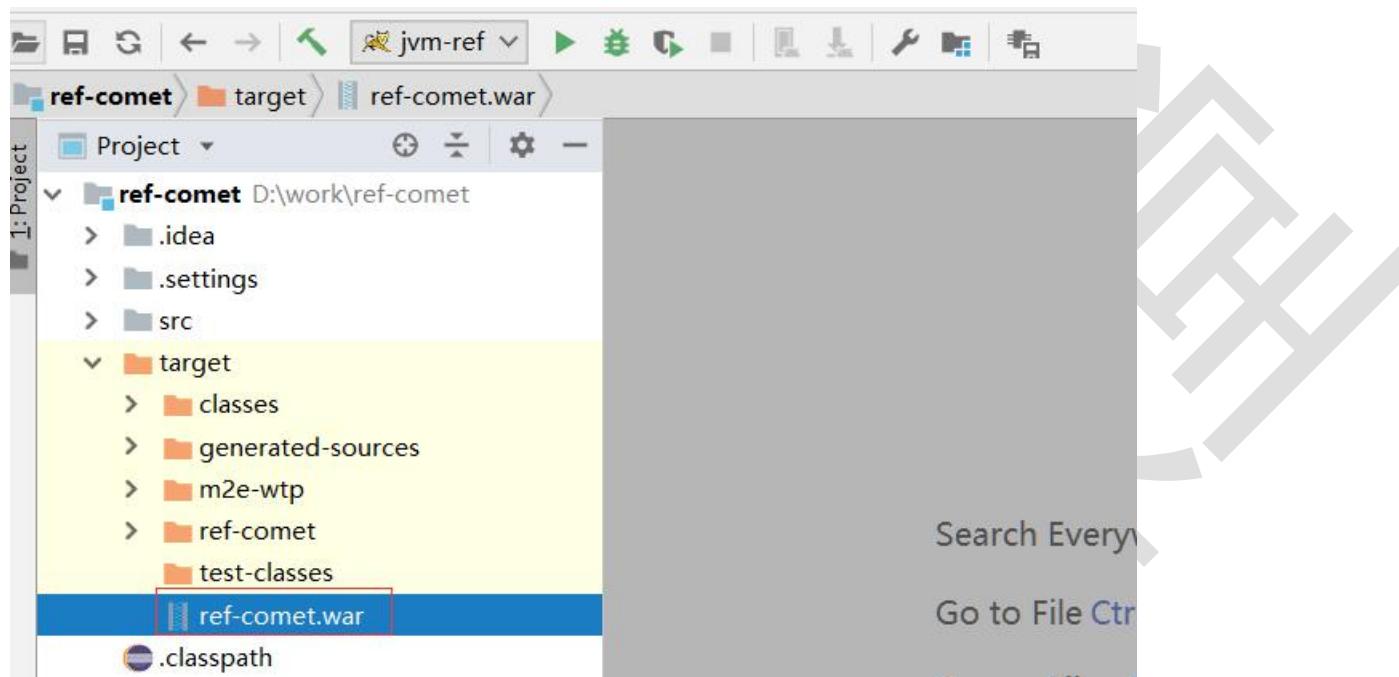
为什么有这个默认首页：Tomcat “买一送一”的思想。默认 Tomcat 启动后加载 webapps 中的项目

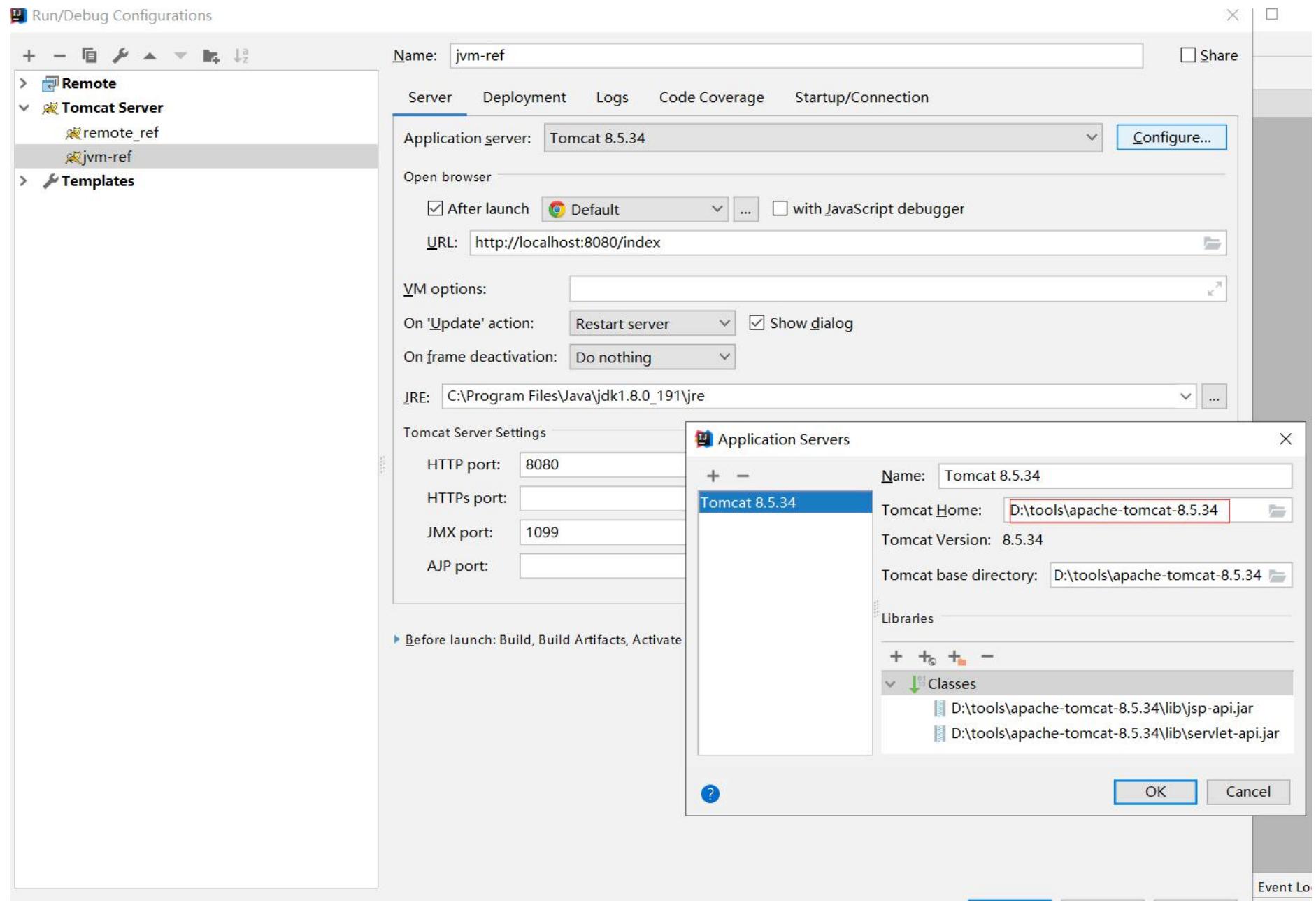
『 > 软件 (D:) > tools > apache-tomcat-8.5.34 > webapps

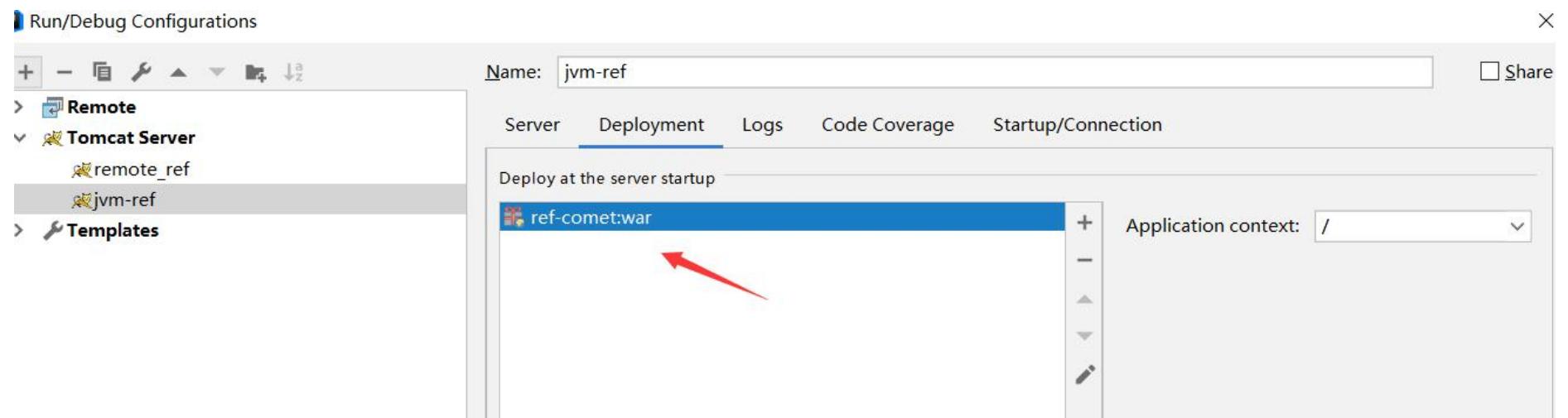
名称	修改日期	类型
comet	2019/7/5 17:08	文件夹
docs	2018/9/4 23:29	文件夹
examples	2018/9/4 23:29	文件夹
host-manager	2018/9/4 23:29	文件夹
manager	2018/9/4 23:29	文件夹
ref-comet	2019/7/16 20:23	文件夹
ROOT	2018/9/4 23:29	文件夹

IDE 中启动

一个 Sprei 项目配置和打包







IDE 中启动成功后的只有项目的访问，没有首页的显示的

Run: jvm-ref ×

Server Tomcat Localhost Log × Tomcat Catalina Log × warnings Q

```
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.base=C:\Use
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.home=D:\toc
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.io.tmpdir=D:\tc
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Loaded APR based Apache Tomcat Native
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR capabilities: IPv6 [true], sendfil
18-Jul-2019 10:25:28.331 信息 [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR/OpenSSL configuration: useAprConne
18-Jul-2019 10:25:29.105 信息 [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL successfully initialized [OpenS
18-Jul-2019 10:25:29.168 信息 [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["http-nio-8080"]
18-Jul-2019 10:25:29.199 信息 [main] org.apache.tomcat.util.net.NioSelectorPool.getSharedSelector Using a shared selector for servlet wr
18-Jul-2019 10:25:29.199 信息 [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["ajp-nio-8009"]
18-Jul-2019 10:25:29.199 信息 [main] org.apache.tomcat.util.net.NioSelectorPool.getSharedSelector Using a shared selector for servlet wr
18-Jul-2019 10:25:29.199 信息 [main] org.apache.catalina.startup.Catalina.load Initialization processed in 1094 ms
18-Jul-2019 10:25:29.230 信息 [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
18-Jul-2019 10:25:29.230 信息 [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet Engine: Apache Tomcat/8.5.34
18-Jul-2019 10:25:29.230 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
18-Jul-2019 10:25:29.251 信息 [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["ajp-nio-8009"]
18-Jul-2019 10:25:29.255 信息 [main] org.apache.catalina.startup.Catalina.start Server startup in 41 ms
```



服务器推送技术演示

- [查看服务器时间](#)
- [Servlet异步-推送实时新闻](#)
- [SSE-贵金属期货价格实时查询](#)
- [在线支付](#)



HTTP Status 404 – Not Found

Type Status Report

Description The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

Apache Tomcat/8.5.34

嵌入式启动

SpringBoot 中一个 main 方法嵌入式启动 Tomcat

The screenshot shows the IntelliJ IDEA interface with the following details:

Project Structure: The project is named "mall". The `MySpringBootApplication.java` file is open in the editor. The code defines a `MySpringBootApplication` class with a `@SpringBootApplication` annotation and a `main` method. Red arrows point to the `@SpringBootApplication` annotation and the closing brace of the `main` method.

```
package com.enjoy;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MySpringBootApplication {
    //相当于启动了Tomcat,端口默认为8080
    public static void main(String[] args) {
        SpringApplication.run(MySpringBootApplication.class, args);
    }
}
```

Run Tab: The "Run" tab is active, showing the output of the application. The console output includes logs from the Spring Boot framework and the application itself. A red box highlights the line "Tomcat started on port(s): 8080 (http)".

```
-07-10 13:49:10.929 INFO 10152 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: 
-07-10 13:49:10.929 INFO 10152 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: /* 
-07-10 13:49:11.188 INFO 10152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping 
-07-10 13:49:11.232 INFO 10152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[/killProduct]" onto public java.lang.String com.enjoy.controller.Controller.killProduct(org.springframework.ui.Model) 
-07-10 13:49:11.236 INFO 10152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[/error]" onto public org.springframework.http.ResponseEntity<org.springframework.web.bind.support.ErrorAttributeSource> com.enjoy.controller.Controller.error() 
-07-10 13:49:11.236 INFO 10152 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[[error],produces=[text/html]]" onto public org.springframework.web.servlet.ModelAndView com.enjoy.controller.Controller.error(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) 
-07-10 13:49:11.261 INFO 10152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [org.springframework.web.servlet.handler.SimpleUrlHandlerMapping] 
-07-10 13:49:11.261 INFO 10152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [com.enjoy.controller.Controller] 
-07-10 13:49:11.289 INFO 10152 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [com.enjoy.controller.Controller] 
-07-10 13:49:11.490 WARN 10152 --- [           main] arterDeprecationWarningAutoConfiguration : spring-boot-starter-redis is deprecated as of 2.2.0.M1 
-07-10 13:49:11.529 INFO 10152 --- [           main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup 
-07-10 13:49:11.571 INFO 10152 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http) 
-07-10 13:49:11.575 INFO 10152 --- [           main] com.enjoy.MySpringBootApplication : Started MySpringBootApplication in 2.599 seconds
```

Debug 启动

在项目发布后，我们有时候需要对基于生产环境部署的应用进行调试，以解决在开发环境无法重现的 BUG。这时我们就需要用到应用服务器的远程调试功能，这个主要是基于 JDK 提供的 JPDA（Java Platform Debugger Architecture,Java 平台调试体系结构）。不过一般情况下用不到，这里简单讲一讲。

使用 DeBug 启动可以对基于生产环境部署的应用进行调试，以解决在开发环境无法重现的 BUG。

使用 IDEA 远程部署 tomcat 和调试

1. 在 catalina.sh 文件中加入以下的配置

```
CATALINA_OPTS="-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.port=1099  
-Dcom.sun.management.jmxremote.ssl=false  
-Dcom.sun.management.jmxremote.authenticate=false  
-Djava.rmi.server.hostname=192.168.19.200  
-agentlib:jdwp=transport=dt_socket,address=15833,suspend=n,server=y" export CATALINA_OPTS
```

以上端口可以随意改动，但是必要的是后续的设置必须保持一致，并且务必保证端口没有被占用，这些设置的端口在防火墙中是开放状态；

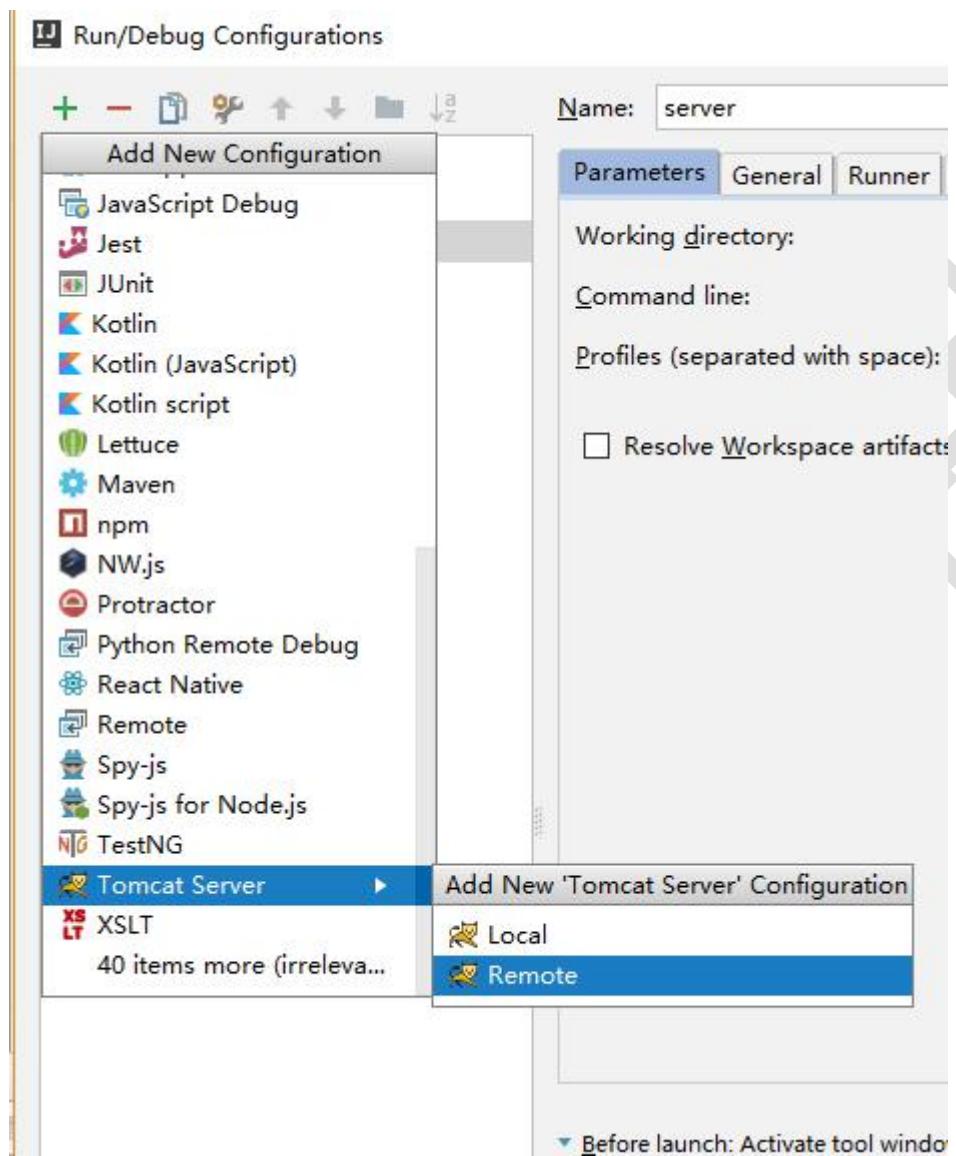
其中 1099 的是 tomcat 远程部署连接端口；

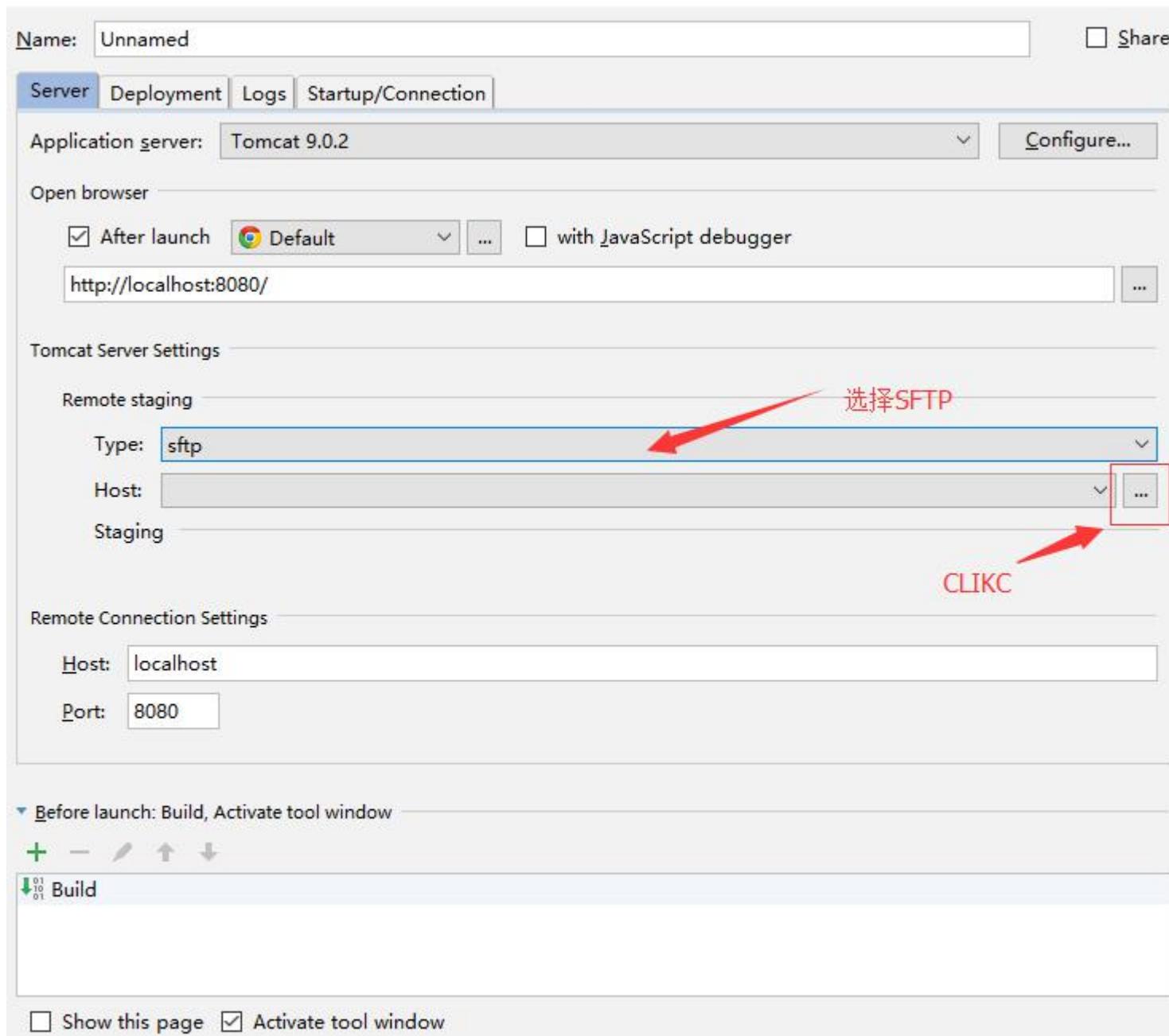
15833 是远程调试的端口；

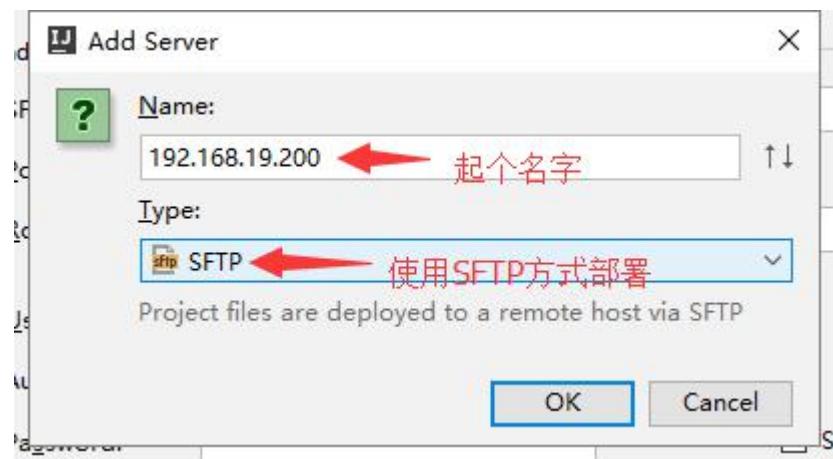
192.168.19.200 是远程的服务器的 Ip。

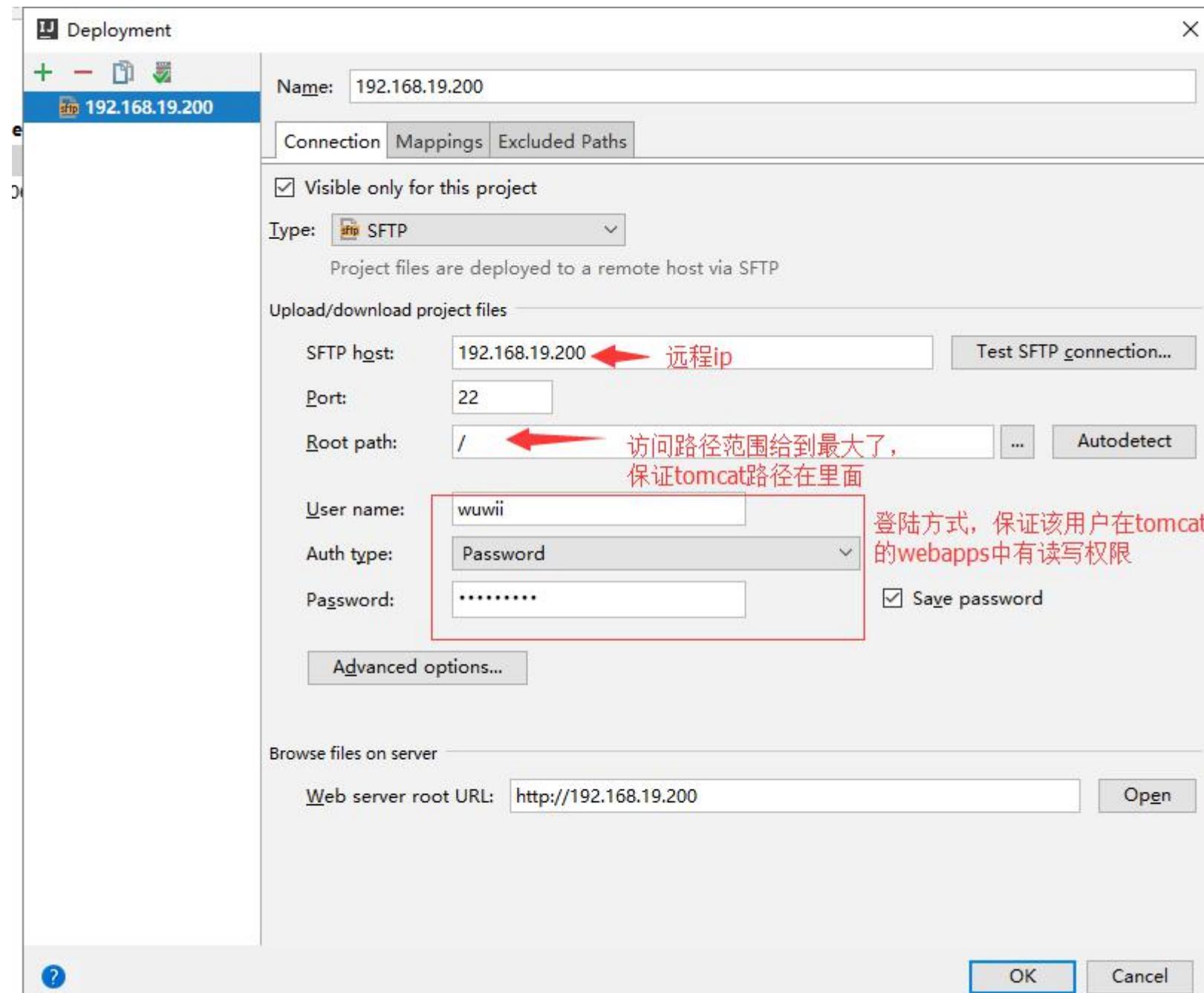
2. 在 Linux 上启动 tomcat, 使用命令启动

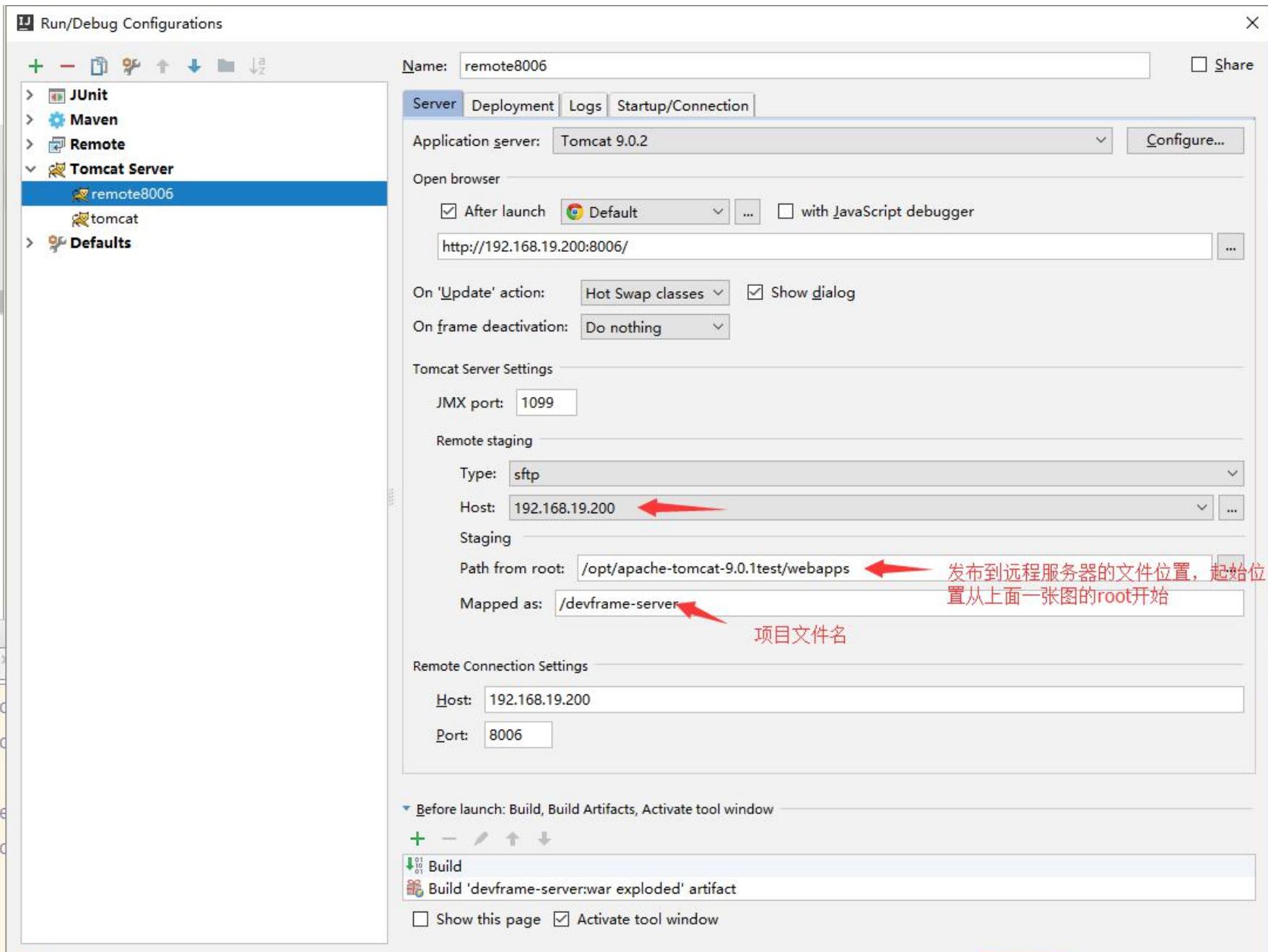
```
./bin/catalina.sh run &
```

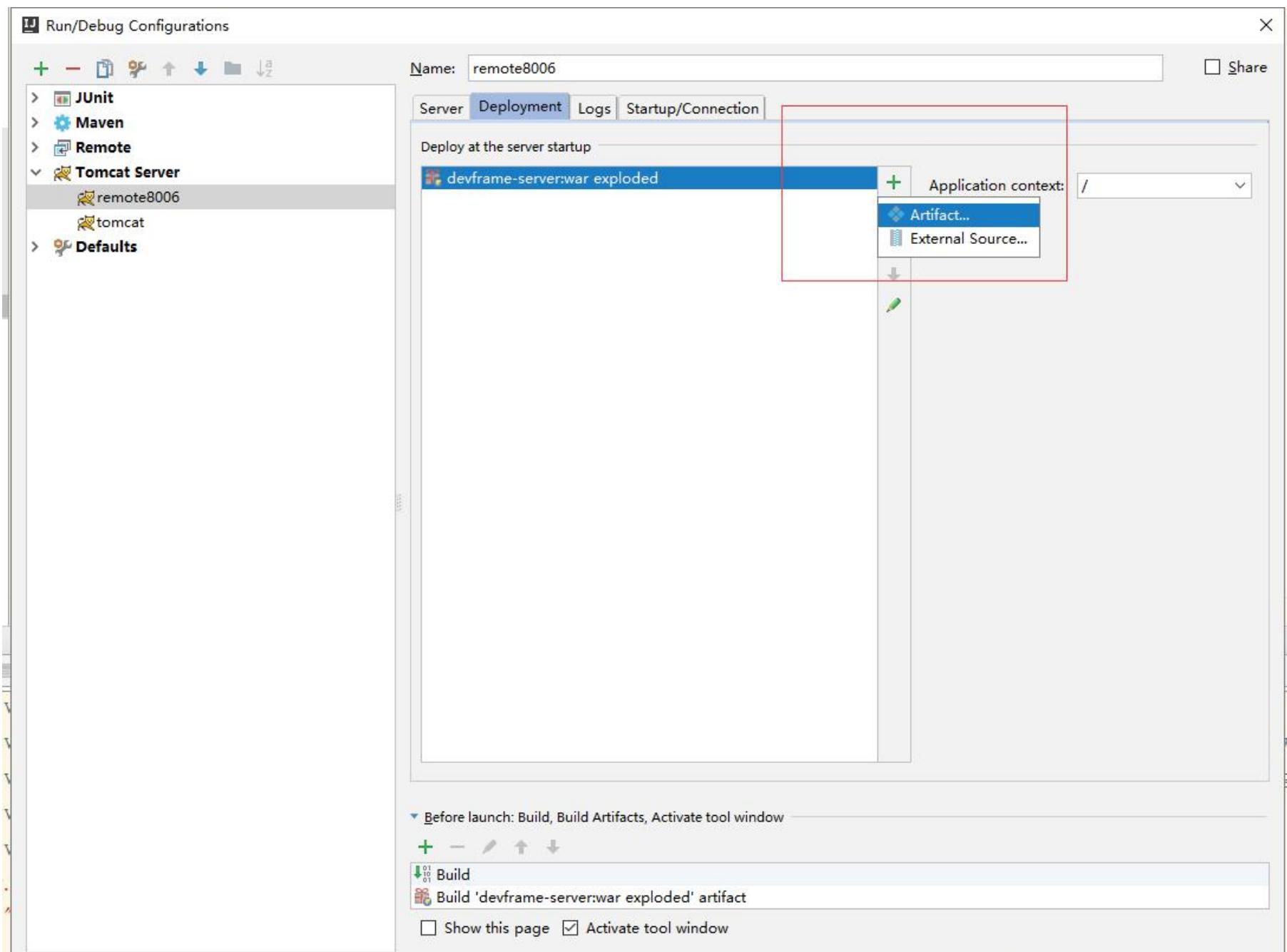


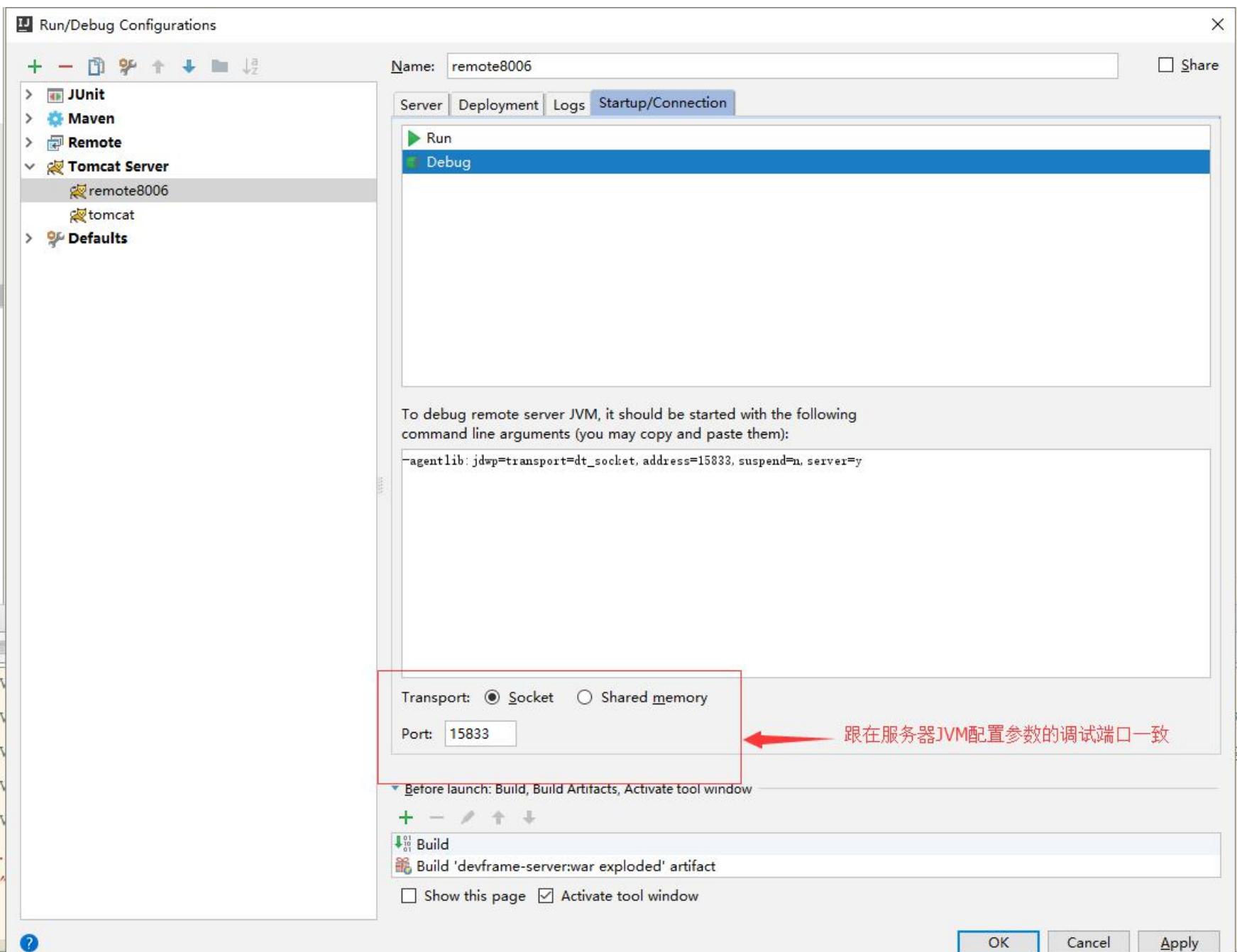












debug 启动测试

连接:

```
[2017-12-23 08:47:03,592] Artifact devframe-server:war exploded: Artifact is not deployed. Press 'Deploy' to start deployment  
[2017-12-23 08:47:03,650] Artifact devframe-server:war exploded: Artifact is being deployed, please wait...  
Connected to server  
Connected to the target VM, address: '192.168.19.200:15833', transport: 'socket'  
[2017-12-23 08:47:11,434] Artifact devframe-server:war exploded: Error during artifact deployment. See serverlog for details.
```

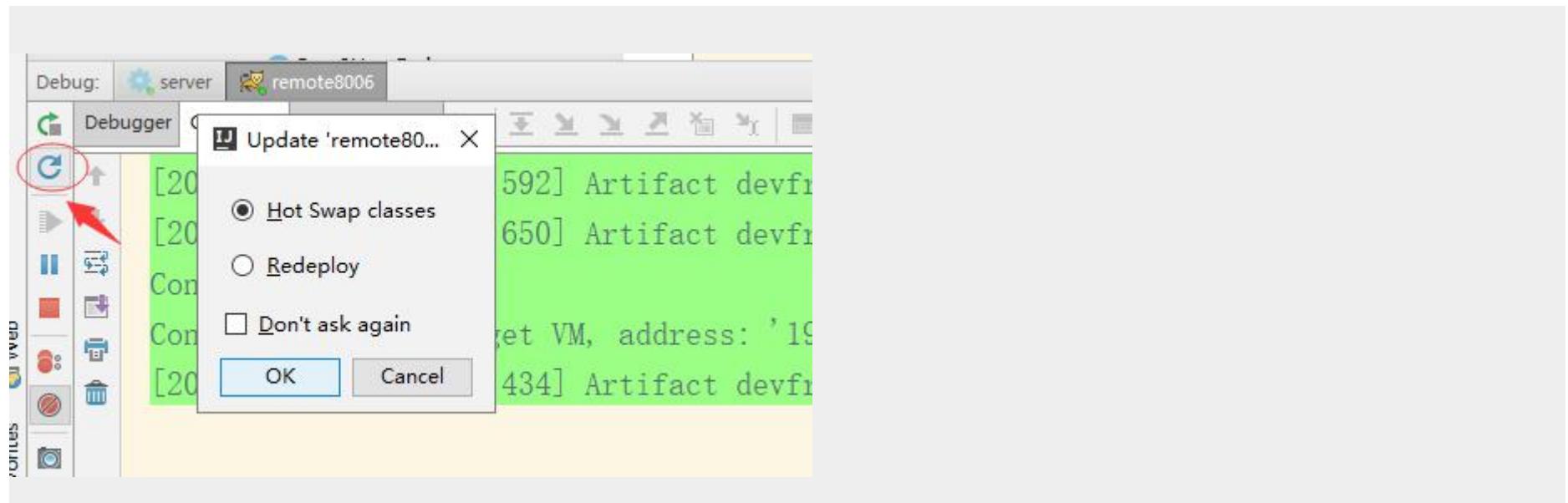
文件传输:

```
[2017/12/23 20:47] Uploading to 192.168.19.200 completed in less than a minute:357 files transferred (8 Mbit/s)
```

这样就能够成功远程部署并且调试了。

使用的技巧:





容易出现的问题

如果远程没有连接上，两个端口被占用或者防火墙屏蔽。除了 JMX server 指定的监听端口号外，JMXserver 还会监听一到两个随机端口号，这个如果防火墙关闭了的话就不用考虑，如果使用了防火墙，还需要查看它监听的端口。

- 账号的相应的读写权限一定要有；

Tomcat 项目部署及目录结构

项目部署

隐式部署

直接丢文件夹、war、jar 到 webapps 目录，tomcat 会根据文件夹名称自动生成虚拟路径，简单，但是需要重启 Tomcat 服务器，包括要修改端口和访问路径的也需要重启。

显式部署

添加 context 元素

server.xml 中的 Host 加入一个 Context（指定路径和文件地址），例如：

```
<Host name="localhost">  
<Context path="/comet" docBase="D:\work_tomcat\ref-comet.war" />
```

即/comet 这个虚拟路径映射到了 D:\work_tomcat\ref-comet 目录下(war 会解压成文件)，修改完 server.xml 需要重启 tomcat 服务器。

创建 xml 文件

在 conf/Catalina/localhost 中创建 xml 文件，访问路径为文件名，例如：

在 localhost 目录下新建 demo.xml，内容为：

```
<Context docBase="D:\work_tomcat\ref-comet" />
```

不需要写 path，虚拟目录就是文件名 demo，path 默认为/demo，添加 demo.xml 不需要重启 tomcat 服务器。

三种方式比较：

隐式部署：可以很快部署，需要人手动移动 Web 应用到 webapps 下，在实际操作中不是很人性化

添加 context 元素：配置速度快，需要配置两个路径，如果 path 为空字符串，则为缺省配置，每次修改 server.xml 文件后都要重新启动 Tomcat 服务器，重新部署。

创建 xml 文件：服务器后台会自动部署，修改一次后台部署一次，不用重复启动 Tomcat 服务器，该方式显得更为智能化。

Bin 执行脚本目录

startup 文件，主要是检查 catalina.bat/sh 执行所需环境，并调用 catalina.bat 批处理文件。启动 tomcat。

catalina 文件，真正启动 Tomcat 文件，可以在里面设置 jvm 参数。后面性能调优会重点讲

shutdown 文件，关闭 Tomcat

脚本 version.sh、startup.sh、shutdown.sh、configtest.sh 都是对 catalina.sh 的包装，内容大同小异，差异在于功能介绍和调用 catalina.sh 时的参数不同。

Version：查看当前 tomcat 的版本号，

Configtest：校验 tomcat 配置文件 server.xml 的格式、内容等是否合法、正确。

Service：安装 tomcat 服务，可用 net start tomcat 启动

web.xml

Tomcat 中所有应用默认的部署描述文件，主要定义了基础的 Servlet 和 MIME 映射(mime-mapping 文件类型，其实就是 Tomcat 处理的文件类型)，如果部署的应用中不包含 Web.xml，那么 Tomcat 将使用此文件初始化部署描述，反之，Tomcat 会在启动时将默认描述与定义描述进行合并。

加载一些 tomcat 内置的 servlet

DefaultServlet 默认的，加载静态文件 html,js,jpg 等静态文件。

JspServlet 专门处理 jsp。

context.xml

用于自定义所有 Web 应用均需要加载的 Context 配置，如果 Web 应用指定了自己的 context.xml，那么该文件的配置将被覆盖。

context.xml 与 server.xml 中配置 context 的区别

server.xml 是不可动态重加载的资源，服务器一旦启动了以后，要修改这个文件，就得重启服务器才能重新加载。而 context.xml 文件则不然，tomcat 服务器会定时去扫描这个文件。一旦发现文件被修改（时间戳改变了），就会自动重新加载这个文件，而不需要重启服务器。

catalina.policy

权限相关 Permission ， Tomcat 是跑在 jvm 上的，所以有些默认的权限

tomcat-users.xml

配置 Tomcat 的 server 的 manager 信息

```
<?xml version="1.0" encoding="UTF-8"?>

<tomcat-users xmlns="http://tomcat.apache.org/xml"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
               version="1.0">

    <role rolename="manager-gui"/>

    <user username="manager" password="manager" roles="manager-gui"/>

</tomcat-users>
```

logging.properties

设置 tomcat 日志

控制输出不输出内容到文件，不能阻止生成文件，阻止声文件可用注释掉

webapps 目录

存放 web 项目的目录，其中每个文件夹都是一个项目；如果这个目录下已经存在了目录，那么都是 tomcat 自带的项目。其中 ROOT 是一个特殊的项目，在地址栏中没有给出项目目录时，对应的就是 ROOT 项目。<http://localhost:8080/examples>，进入示例项目。其中 examples 就是项目名，即文件夹的名字。

lib 目录

Tomcat 的类库，里面是一大堆 jar 文件。如果需要添加 Tomcat 依赖的 jar 文件，可以把它放到这个目录中，当然也可以把应用依赖的 jar 文件放到这个目录中，这个目录中的 jar 所有项目都可以共享之，但这样你的应用放到其他 Tomcat 下时就不能再共享这个目录下的 Jar 包了，所以建议只把 Tomcat 需要的 Jar 包放到这个目录下；

work 目录

运行时生成的文件，最终运行的文件都在这里。通过 webapps 中的项目生成的！可以把这个目录下的内容删除，再次运行时会生再次生成 work 目录。当客户端用户访问一个 JSP 文件时，Tomcat 会通过 JSP 生成 Java 文件，然后再编译 Java 文件生成 class 文件，生成的 java 和 class 文件都会存放到这个目录下。

temp 目录

存放 Tomcat 的临时文件，这个目录下的东西可以在停止 Tomcat 后删除！

logs 目录

这个目录中都是日志文件，记录了 Tomcat 启动和关闭的信息，如果启动 Tomcat 时有错误，那么异常也会记录在日志文件中

localhost-xxx.log Web 应用的内部程序日志，建议保留

catalina-xxx.log 控制台日志

host-manager.xxx.log Tomcat 管理页面中的 host-manager 的操作日志，建议关闭

localhost_access_log_xxx.log 用户请求 Tomcat 的访问日志（这个文件在 conf/server.xml 里配置），建议关闭

Conf 配置文件目录

server.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Server 代表一个 Tomcat 实例。可以包含一个或多个 Services，其中每个 Service 都有自己的 Engines 和 Connectors。-->

<Server port="8005" shutdown="SHUTDOWN">
    <!-- 监听器 -->
    <Listener className="org.apache.catalina.startup.VersionLoggerListener" />
    <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
    <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
    <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
    <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />
    <!-- 全局命名资源，定义了 UserDatabase 的一个 JNDI(java 命名和目录接口)，通过 pathname 的文件得到一个用户授权的内存数据库 -->
```

```
<GlobalNamingResources>

    <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>

<!-- Service 它包含一个<Engine>元素, 以及一个或多个<Connector>, 这些 Connector 元素共享用同一个 Engine 元素 -->

<Service name="Catalina">

    <!--

        每个 Service 可以有一个或多个连接器<Connector>元素,
        第一个 Connector 元素定义了一个 HTTP Connector, 它通过 8080 端口接收 HTTP 请求; 第二个 Connector 元素定
        义了一个 JD Connector, 它通过 8009 端口接收由其它服务器转发过来的请求.

    -->

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

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

    <!-- 每个 Service 只能有一个<Engine>元素 -->
```

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

    <Realm className="org.apache.catalina.realm.LockOutRealm">
        <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
              resourceName="UserDatabase"/>
    </Realm>

    <!-- 默认 host 配置，有几个域名就配置几个 Host，但是这种只能是同一个端口号 -->

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

        <!-- Tomcat 的访问日志，默认可以关闭掉它，它会在 logs 文件里生成 localhost_access_log 的访问日志 -->

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

    </Host>

    <Host name="www.hzg.com"  appBase="webapps"
          unpackWARs="true" autoDeploy="true">

        <Context path="" docBase="/myweb1" />

        <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
              prefix="hzg_access_log" suffix=".txt"
              pattern="%h %l %u %t \"%r\" %s %b" />
    
```

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

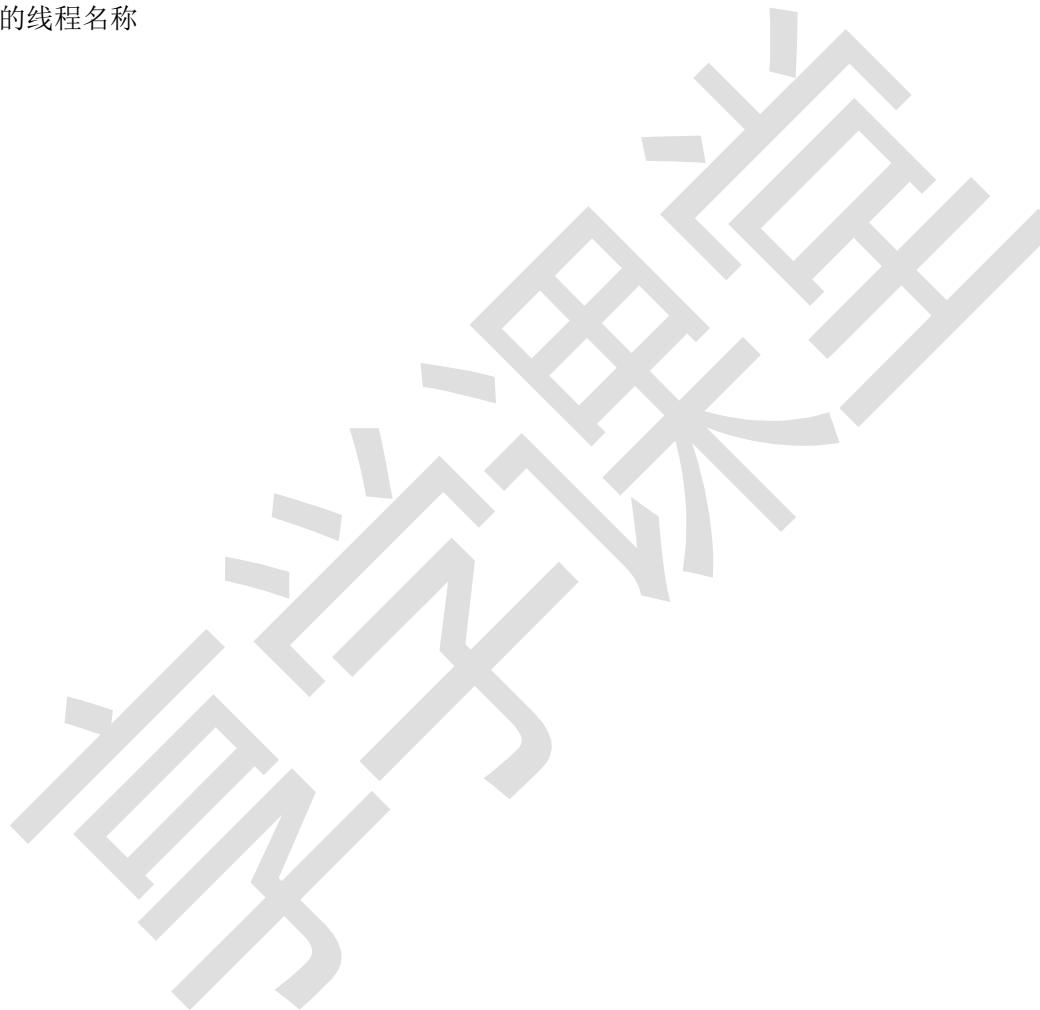
server.xml 中日志的 patter 解释

有效的日志格式模式可以参见下面内容，如下字符串，其对应的信息由指定的响应内容取代：

- %a - 远程 IP 地址
- %A - 本地 IP 地址
- %b - 发送的字节数，不包括 HTTP 头，或“-”如果没有发送字节
- %B - 发送的字节数，不包括 HTTP 头
- %h - 远程主机名
- %H - 请求协议
- %l(小写的 l)- 远程逻辑从 identd 的用户名（总是返回'-'）
- %m - 请求方法
- %p - 本地端口
- %q - 查询字符串（在前面加上一个“?”如果它存在，否则是一个空字符串
- %r - 第一行的要求
- %s - 响应的 HTTP 状态代码
- %S - 用户会话 ID
- %t - 日期和时间，在通用日志格式
- %u - 远程用户身份验证
- %U - 请求的 URL 路径
- %v - 本地服务器名
- %D - 处理请求的时间（以毫秒为单位）

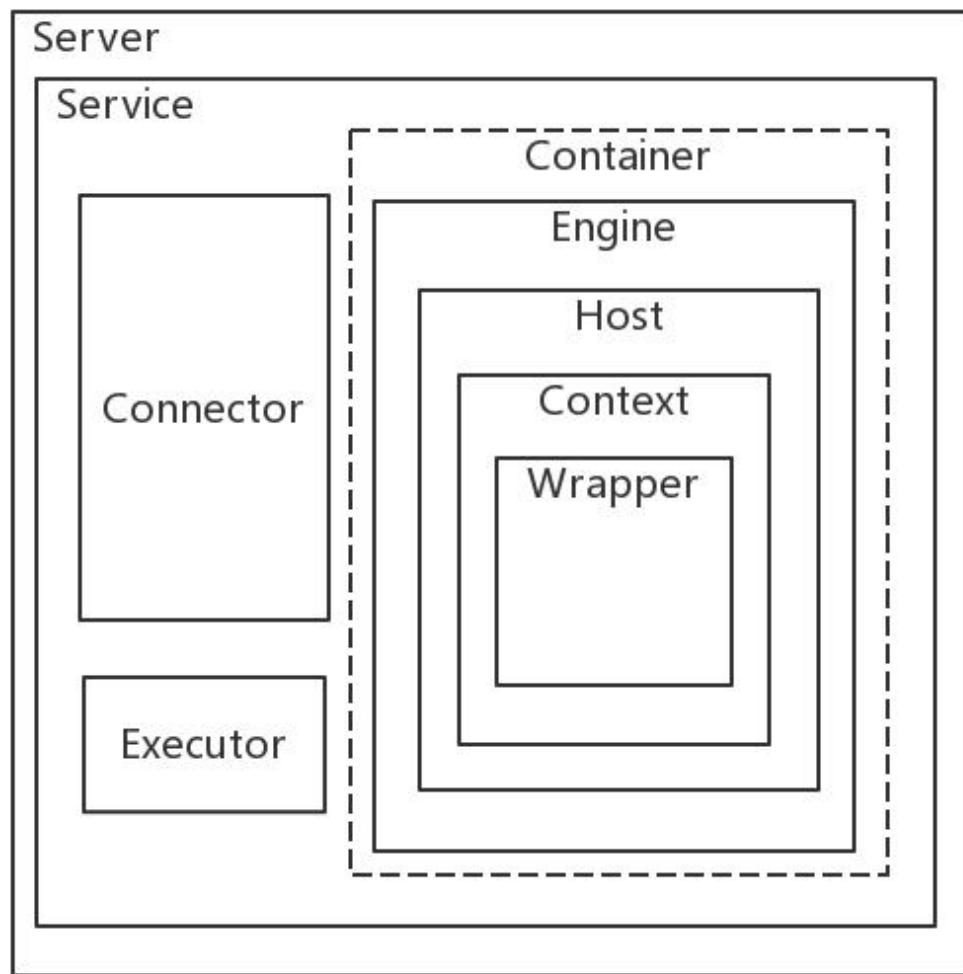
%T - 处理请求的时间（以秒为单位）

%I (大写的 i) - 当前请求的线程名称



Tomcat 组件及架构





Server

Server 是最顶级的组件，它代表 Tomcat 的运行实例，它掌管着整个 Tomcat 的生死大权；

- 提供了监听器机制，用于在 Tomcat 整个生命周期中对不同时间进行处理
- 提供 Tomcat 容器全局的命名资源实现，JNDI
- 监听某个端口以接受 SHUTDOWN 命令，用于关闭 Tomcat

Service

一个概念，一个 Service 维护多个 Connector 和一个 Container

Connector 组件

链接器：监听转换 Socket 请求，将请求交给 Container 处理，支持不同协议以及不同的 I/O 方式

Container

表示能够执行客户端请求并返回响应的一类对象，其中有不同级别的容器：Engine、Host、Context、Wrapper

Engine

整个 Server 引擎，最高级的容器对象

Host

表示 Servlet 引擎中的虚拟机，主要与域名有关，一个服务器有多个域名是可以使用多个 Host

Context

用于表示 ServletContext,一个 ServletContext 表示一个独立的 Web 应用

Wrapper

用于表示 Web 应用中定义的 Servlet

Executor

Tomcat 组件间可以共享的线程池

Tomcat 的核心组件

解耦：网络协议与容器的解耦。

Connector 链接器封装了底层的网络请求(Socket 请求及相应处理),提供了统一的接口,使 Container 容器与具体的请求协议以及 I/O 方式解耦。

Connector 将 Socket 输入转换成 Request 对象,交给 Container 容器进行处理,处理请求后,Container 通过 Connector 提供的 Response 对象将结果写入输出流。

因为无论是 Request 对象还是 Response 对象都没有实现 Servlet 规范对应的接口,Container 会将它们进一步分装成 ServletRequest 和 ServletResponse.

Tomcat 的连接器

AJP 主要是用于 Web 服务器与 Tomcat 服务器集成, AJP 采用二进制传输可读性文本, 使用保持持久性的 TCP 链接,使得 AJP 占用更少的带宽,并且链接开销要小得多,但是由于 AJP 采用持久化链接,因此有效的连接数较 HTTP 要更多。

HTTP2.0 目前市场不成熟,这个技术点后续我们的三期、四期如果市面上协议很普遍了会考虑加入。

对于 I/O 选择,要根据业务场景来定,一般高并发场景下,APR 和 NIO2 的性能要优于 NIO 和 BIO,(linux 操作系统支持的 NIO2 由于是一个假的,并没有真正实现 AIO,所以一般 linux 上推荐使用 NIO,如果是 APR 的话,需要安装 APR 库,而 Windows 上默认安装了),所以在 8.5 的版本中默认是 NIO。

第二节 Tomcat 源码分析（从启动流程到请求处理）

Tomcat 8.5 下载地址

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

Tomcat 启动流程

Tomcat 源码目录

catalina 目录

catalina 包含所有的 Servlet 容器实现，以及涉及到安全、会话、集群、部署管理 Servlet 容器的各个方面，同时，它还包含了启动入口。

coyote 目录

coyote 是 Tomcat 链接器框架的名称，是 Tomcat 服务器提供的客户端访问的外部接口，客户端通过 Coyote 与服务器建立链接、发送请求并接收响应。

El 目录，提供 java 表达式语言

Jasper 模块提供 JSP 引擎

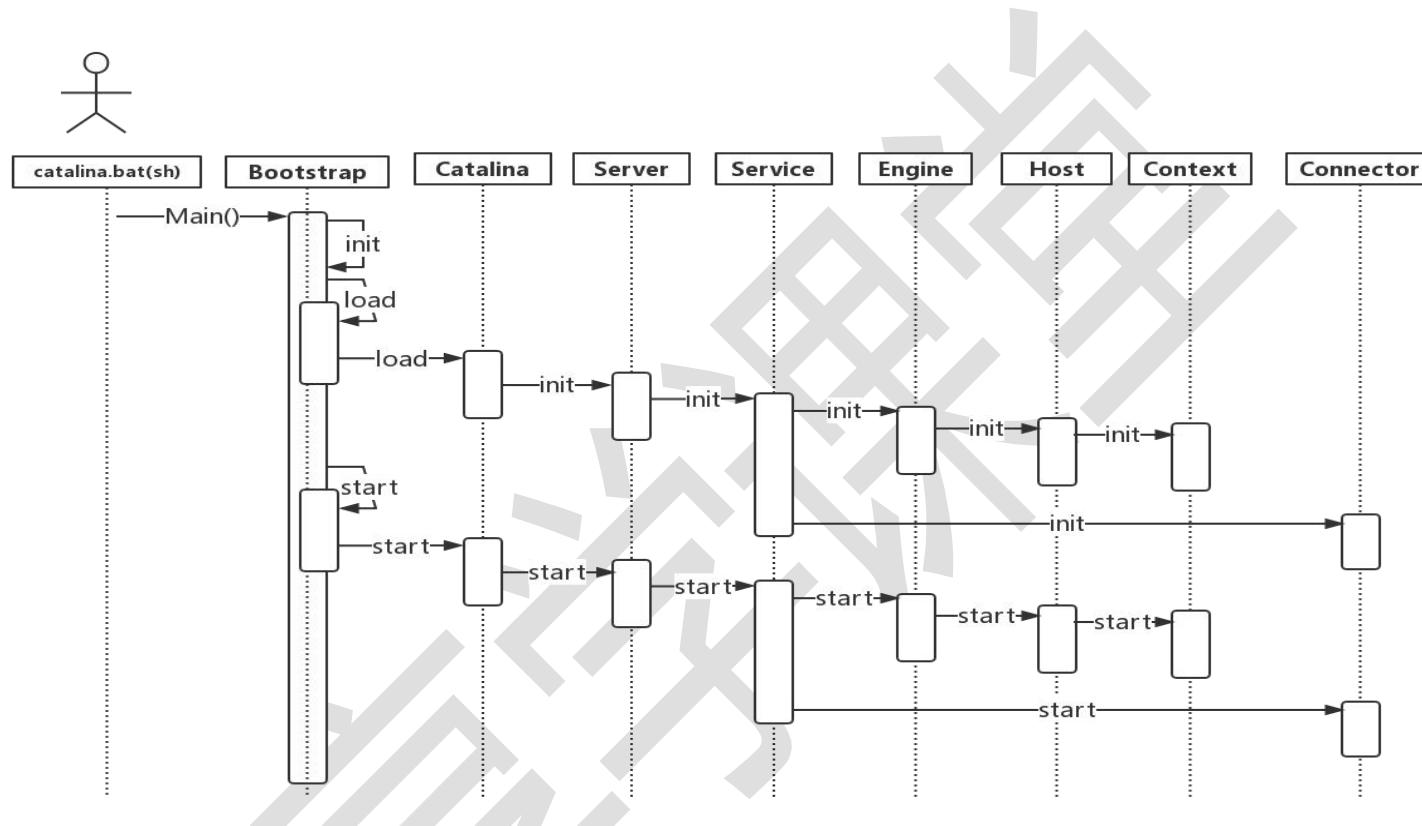
Naming 模块提供 JNDI 的服务

Juli 提供服务器日志的服务

tomcat 提供外部调用的接口 api

Tomcat 启动流程分析

1. 启动流程解析：注意是标准的启动，也就是从 bin 目录下的启动文件中启动 Tomcat



我们可以看到这个流程非常的清晰，同时注意到，Tomcat 的启动非常的标准，除去 Bootstrap 和 Catalin，我们可以对照一下 Server.xml 的配置文件。Server,service 等等这些组件都是一一对照，同时又有先后顺序。

基本的顺序是先 init 方法，然后再 start 方法。

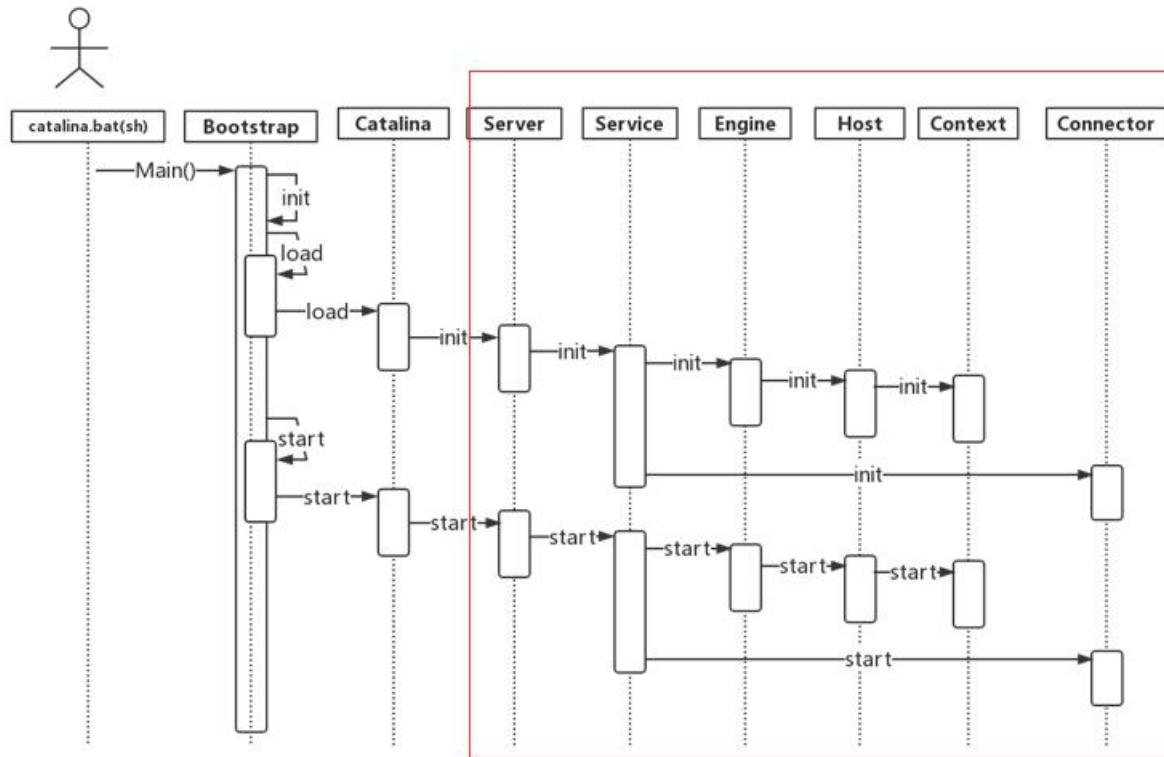
2. 加入调试信息(): 注意是标准的启动，也就是从 bin 目录下的启动文件中启动 Tomcat

```
Bootsrap--init()
19-Jul-2019 15:20:08.248 警告 [main] org.apache.catalina.startup.
19-Jul-2019 15:20:08.253 警告 [main] org.apache.catalina.startup.
19-Jul-2019 15:20:08.254 警告 [main] org.apache.catalina.startup.
19-Jul-2019 15:20:08.254 警告 [main] org.apache.catalina.startup.
Bootsrap--load()
19-Jul-2019 15:20:08.270 严重 [main] org.apache.catalina.startup.
Catalina--load()
class org.apache.catalina.core.StandardServer--init()
19-Jul-2019 15:20:08.652 信息 [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent The APR based Apache Tomcat
class org.apache.catalina.core.StandardService--init()
class org.apache.catalina.core.StandardEngine--init()
class org.apache.catalina.connector.Connector--init()
19-Jul-2019 15:20:08.751 信息 [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["http-nio-8080"]
19-Jul-2019 15:20:09.267 信息 [main] org.apache.tomcat.util.net.NioSelectorPool.getSharedSelector Using a shared selector
class org.apache.catalina.connector.Connector--init()
Bootsrap--start()
Catalina--start()
class org.apache.catalina.core.StandardServer--start()
19-Jul-2019 15:20:09.278 信息 [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["ajp-nio-8009"]
19-Jul-2019 15:20:09.281 信息 [main] org.apache.tomcat.util.net.NioSelectorPool.getSharedSelector Using a shared selector
19-Jul-2019 15:20:09.281 信息 [main] org.apache.catalina.startup.Catalina.load Initialization processed in 1011 ms
19-Jul-2019 15:20:09.308 信息 [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
19-Jul-2019 15:20:09.308 信息 [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet Engine: Apache
class org.apache.catalina.core.StandardService--start()
class org.apache.catalina.core.StandardEngine--start()
```

可以看到，在源码中加入调试的信息和流程图是一致的。

我们可以看到，除了 Bootstrap 和 catalina 类，其他的 Server,service 等等之类的都只是一个接口，实现类均为 StandardXXX 类。

我们来看下 StandardServer 类，



问题来了，我们发现 `StandardServer` 类中没有 `init` 方法，只有一个类似于 `init` 的 `initInternal` 方法，这个是为什么？

带着问题我们进入下面的内容。

分析 Tomcat 请求过程

解耦：网络协议与容器的解耦。

Connector 链接器封装了底层的网络请求(Socket 请求及相应处理),提供了统一的接口,使 Container 容器与具体的请求协议以及 I/O 方式解耦。

Connector 将 Socket 输入转换成 Request 对象,交给 Container 容器进行处理,处理请求后,Container 通过 Connector 提供的 Response 对象将结果写入输出流。

因为无论是 Request 对象还是 Response 对象都没有实现 Servlet 规范对应的接口,Container 会将它们进一步分装成 ServletRequest 和 ServletResponse.问题来了,在 Engine 容器中,有四个级别的容器,他们的标准实现分别是 StandardEngine、StandardHost、StandardContext、StandardWrapper。

组件的生命周期管理

各种组件如何统一管理

Tomcat 的架构设计是清晰的、模块化、它拥有很多组件,加入在启动 Tomcat 时一个一个组件启动,很容易遗漏组件,同时还会对后面的动态组件拓展带来麻烦。如果采用我们传统的方式的话,组件在启动过程中如果发生异常,会很难管理,比如你的下一个组件调用了 start 方法,但是如果它的上级组件还没有 start 甚至还没有 init 的话,Tomcat 的启动会非常难管理,因此, Tomcat 的设计者提出一个解决方案:用 Lifecycle 管理启动,停止、关闭。

生命周期统一接口——Lifecycle

Tomcat 内部架构中各个核心组件有包含与被包含关系,例如: Server 包含了 Service,Service 又包含了 Container 和 Connector,这个结构有一点像数据结构中的树,树的根结点没有父节点,其他节点有且仅有一个父节点,每一个父节点有 0 至多个子节点。所以,我们可以通过父容器启动它的子容器,这样只要启动根容器,就可以把其他所有的容器都启动,从而达到了统一的启动,停止、关闭的效果。

所有所有组件有一个统一的接口——Lifecycle,把所有的启动、停止、关闭、生命周期相关的方法都组织到一起,就可以很方便管理 Tomcat 各个容器组件的生命周期。

Lifecycle 其实就是定义了一些状态常量和几个方法,主要方法是 init,start,stop 三个方法。

例如: Tomcat 的 Server 组件的 init 负责遍历调用其包含所有的 Service 组件的 init 方法。

注意：Server 只是一个接口，实现类为 StandardServer,有意思的是，StandardServer 没有 init 方法，init 方法是在哪里，其实是在它的父类 LifecycleBase 中，这个类就是统一的生命周期管理。

```
/*
public final class StandardServer extends LifecycleMBeanBase implements Server {
```



```
public abstract class LifecycleMBeanBase extends LifecycleBase implements JmxEnabled {
```

LifecycleBase.java

```
89     ^ param data - data associated with event.
90     */
91     protected void fireLifecycleEvent(String type, Object data) {
92         LifecycleEvent event = new LifecycleEvent( lifecycle: this, type, data);
93         for (LifecycleListener listener : lifecycleListeners) {
94             listener.lifecycleEvent(event);
95         }
96     }
97
98
99     @Override
100    public final synchronized void init() throws LifecycleException {
101        if (!state.equals(LifecycleState.NEW)) {
102            invalidTransition(Lifecycle.BEFORE_INIT_EVENT);
103        }
104
105        try {
106            setStateInternal(LifecycleState.INITIALIZING, data: null, check: false);
107            initInternal();
108            setStateInternal(LifecycleState.INITIALIZED, data: null, check: false);
109        } catch (Throwable t) {
110            ExceptionUtils.handleThrowable(t);
111            setStateInternal(LifecycleState.FAILED, data: null, check: false);
112            throw new LifecycleException(
113                sm.getString(key: "lifecycleBase.initFail",toString()), t);
114        }
115    }
116
117    protected abstract void initInternal() throws LifecycleException;
```

所以 StandardServer 最终只会调用到 initInternal 方法，这个方法会初始化子容器 Service 的 init 方法



```
 45      //...
 46      if (getCatalina() != null) {
 47          ClassLoader cl = getCatalina().getParentClassLoader();
 48          //...
 49          while (cl != null && cl != ClassLoader.getSystemClassLoader()) {
 50              if (cl instanceof URLClassLoader) {
 51                  URL[] urls = ((URLClassLoader) cl).getURLs();
 52                  for (URL url : urls) {
 53                      if (url.getProtocol().equals("file")) {
 54                          try {
 55                              File f = new File (url.toURI());
 56                              if (f.isFile() &&
 57                                  f.getName().endsWith(".jar")) {
 58                                  ExtensionValidator.addSystemResource(f);
 59                              }
 60                          } catch (URISyntaxException e) {
 61                              // Ignore
 62                          } catch (IOException e) {
 63                              // Ignore
 64                          }
 65                      }
 66                  }
 67              }
 68          }
 69      }
 70      cl = cl.getParent();
 71  }
 72 }
 73 // Server组件中的init方法负责遍历所有的子容器service组件的init方法
 74 for (int i = 0; i < services.length; i++) {
 75     services[i].init();
 76 }
 77 }
```

为什么 LifecycleBase 这么玩，其实很多架构源码都是这么玩的，包括 JDK 的容器源码都是这么玩的，一个类，有一个接口，同时抽象一个抽象骨架类，把通用的实现放在抽象骨架类中，这样设计就方便组件的管理，使用 LifecycleBase 骨架抽象类，在抽象方法中就可以进行统一的处理，具体的内容见下面。

抽象类 LifecycleBase 统一管理组件生命周期

```
@Override  
public final synchronized void init() throws LifecycleException {  
    if (!state.equals(LifecycleState.NEW)) {  
        invalidTransition(Lifecycle.BEFORE_INIT_EVENT);  
    }  
  
    try {  
        //只打印核心组件  
        if(this.getClass().getName().startsWith("org.apache.catalina.core"))||this.getClass().  
            System.out.println(this.getClass()+"--init());  
    }  
    setStateInternal(LifecycleState.INITIALIZING, data: null, check: false);  
    initInternal();  
    setStateInternal(LifecycleState.INITIALIZED, data: null, check: false);  
} catch (Throwable t) {  
    ExceptionUtils.handleThrowable(t);  
    setStateInternal(LifecycleState.FAILED, data: null, check: false);  
    throw new LifecycleException(  
        sm.getString( key: "lifecycleBase.initFail",toString()), t);  
}  
}
```

初始化统一的状态验证

```
/*
@Override
public final synchronized void start() throws LifecycleException {

    if (LifecycleState.STARTING_PREP.equals(state) || LifecycleState.STARTING.equals(state) ||
        LifecycleState.STARTED.equals(state)) {

        if (log.isDebugEnabled()) {
            Exception e = new LifecycleException();
            log.debug(sm.getString( key: "lifecycleBase.alreadyStarted", toString()), e);
        } else if (log.isInfoEnabled()) {
            log.info(sm.getString( key: "lifecycleBase.alreadyStarted", toString()));
        }

        return;
    }

    if (state.equals(LifecycleState.NEW)) {
        init();
    } else if (state.equals(LifecycleState.FAILED)) {
        stop();
    } else if (!state.equals(LifecycleState.INITIALIZED) &&
               !state.equals(LifecycleState.STOPPED)) {
        invalidTransition(Lifecycle.BEFORE_START_EVENT);
    }
}

try {
    //只打印核心组件
    if(this.getClass().getName().startsWith("org.apache.catalina.core")||this.getClass().getName().startsWith("org.apache.catalina.connector")) {
        System.out.println(this.getClass()+"--start()");
    }
}
```

统一的生命周期管理，组件的状态验证

```
try {
    //只打印核心组件
    if(this.getClass().getName().startsWith("org.apache.catalina.core")||this.getClass().getName().
        System.out.println(this.getClass()+"--start()");
}

setStateInternal(LifecycleState.STARTING_PREP,  data: null,  check: false);
startInternal();

if (state.equals(LifecycleState.FAILED)) {
    // This is a 'controlled' failure. The component put itself into the
    // FAILED state so call stop() to complete the clean-up.
    stop();
} else if (!state.equals(LifecycleState.STARTING)) {
    // Shouldn't be necessary but acts as a check that sub-classes are
    // doing what they are supposed to.
    invalidTransition(Lifecycle.AFTER_START_EVENT);
} else {
    setStateInternal(LifecycleState.STARTED,  data: null,  check: false);
}
} catch (Throwable t) {
    // This is an 'uncontrolled' failure so put the component into the
    // FAILED state and throw an exception.
    ExceptionUtils.handleThrowable(t);
    setStateInternal(LifecycleState.FAILED,  data: null,  check: false);
    throw new LifecycleException(sm.getString( key: "lifecycleBase.startFail",  toString()),  t);
}
}
```

启动完以后状态验证

具体实现类 StandardXXX 类调用 initInternal 方法实现具体的业务处理。

```
.5.42-src | java | org | apache | catalina | util | LifecycleBase |
CardServer.java x | LifecycleBase.java x

    for (LifecycleListener listener : lifecycleListeners) {
        listener.lifecycleEvent(event);
    }
}

@Override
public final synchronized void init() throws LifecycleException {
    if (!state.equals(LifecycleState.NEW)) {
        invalidTransition(Lifecycle.BEFORE_INIT_EVENT);
    }

    try {
        //只打印核心组件
        if(this.getClass().getName().startsWith("org.apache.catalina.core"))||this.getClass().getNa
            System.out.println(this.getClass()+"--init()");
    }
    setStateInternal(LifecycleState.INIALIZING, data: null, check: false);
    initInternal();
    setStateInternal(LifecycleState.INITIALIZED, data: null, check: false);
} catch (Throwable t) {
    ExceptionUtils.handleThrowable(t);
    setStateInternal(LifecycleState.FAILED, data: null, check: false);
    throw new LifecycleException(
        sm.getString( key: "lifecycleBase.initFail",toString()), t);
}
}
```

分析 Tomcat 请求过程

Host 设计的目的

Tomcat 诞生时，服务器资源很贵，所以一般一台服务器其实可以有多个域名映射，满足了这种需求，比如，我的这台电脑，有一个 localhost 域名，同时在我的 hosts 文件中配置两个域名，一个 www.a.com 一个 localhost。

Context 设计的目的

container 从上一个组件 connector 手上接过解析好的内部 request，根据 request 来进行一系列的逻辑操作，直到调用到请求的 servlet，然后组装好 response，返回给 connector。

先来看看 container 的分类吧：

Engine

Host

Context

Wrapper

它们各自的实现类分别是 StandardEngine, StandardHost, StandardContext, and StandardWrapper，它们都在 tomcat 的 org.apache.catalina.core 包下。

它们之间的关系，可以查看 tomcat 的 server.xml 也能明白（根据节点父子关系），这么比喻吧：除了 Wrapper 最小，不能包含其他 container 外，Context 内可以有零或多个 Wrapper，Host 可以拥有零或多个 Host，Engine 可以有零到多个 Host。

Standard 的 container 都是直接继承抽象类：org.apache.catalina.core.ContainerBase：

```
/*
public abstract class ContainerBase extends LifecycleMBeanBase
    implements Container {

    private static final Log log = ...

    /**
     * Perform addChild with the path.
     * addChild can be called with
     * this allows the XML parser to
     * Tomcat.
     */
    protected class PrivilegedAddCh...
```

Choose Implementation of ContainerBase

- c CustomContext in TestApplicationSessionCookieConfig (org.apache.catalina.core)
- c ExistingStandardWrapper in Tomcat (org.apache.catalina.startup.Tomcat)
- c ReplicatedContext (org.apache.catalina.ha.context)
- c StandardContext (org.apache.catalina.core)
- c StandardEngine (org.apache.catalina.core)
- c StandardHost (org.apache.catalina.core)
- c StandardWrapper (org.apache.catalina.core)

c TesterContext in TestHostConfigAutomaticDeployment (org.apache.catalina.core)

Tomcat 处理一个 HTTP 请求的过程

用户点击网页内容，请求被发送到本机端口 8080，被在那里监听的 Coyote HTTP/1.1 Connector 获得。

Connector 把该请求交给它所在的 Service 的 Engine 来处理，并等待 Engine 的回应。

Engine 获得请求 localhost/test/index.jsp，匹配所有的虚拟主机 Host。

Engine 匹配到名为 localhost 的 Host（即使匹配不到也把请求交给该 Host 处理，因为该 Host 被定义为该 Engine 的默认主机），名为 localhost 的 Host 获得请求/test/index.jsp，匹配它所拥有的所有的 Context。Host 匹配到路径为/test 的 Context（如果匹配不到就把该请求交给路径名为“”的 Context 去处理）。

path="/test" 的 Context 获得请求/index.jsp，在它的 mapping table 中寻找出对应的 Servlet。Context 匹配到 URL PATTERN 为*.jsp 的 Servlet, 对应于 JspServlet 类。

构造 HttpServletRequest 对象和 HttpServletResponse 对象，作为参数调用 JspServlet 的 doGet () 或 doPost () .执行业务逻辑、数据存储等程序。

Context 把执行完之后的 HttpServletResponse 对象返回给 Host。

Host 把 HttpServletResponse 对象返回给 Engine。
Engine 把 HttpServletResponse 对象返回 Connector。
Connector 把 HttpServletResponse 对象返回给客户 Browser。

管道模式

管道与阀门

在一个比较复杂的大型系统中，如果一个对象或数据流需要进行繁杂的逻辑处理，我们可以选择在一个大的组件中直接处理这些繁杂的逻辑处理，这种方式虽然达到目的，但是拓展性和可重用性差。因为牵一发而动全身。

管道是就像一条管道把多个对象连接起来，整体看起来就像若干个阀门嵌套在管道中，而处理逻辑放在阀门上。

它的结构和实现是非常值得我们学习和借鉴的。

首先要了解的是每一种 container 都有一个自己的 StandardValve
上面四个 container 对应的四个是：

StandardEngineValve

StandardHostValve

StandardContextValve

StandardWrapperValve

Pipeline 就像一个工厂中的生产线，负责调配工人（valve）的位置，valve 则是生产线上负责不同操作的工人。

一个生产线的完成需要两步：

- 1, 把原料运到工人边上
- 2, 工人完成自己负责的部分

而 tomcat 的 Pipeline 实现是这样的：

- 1, 在生产线上的第一个工人拿到生产原料后，二话不说就人给下一个工人，下一个工人模仿第一个工人那样扔给下一个工人，直到最后一个工人，而最后一个工人被安排为上面提过的 StandardValve，他要完成的工作居然是把生产资料运给自己包含的 container 的 Pipeline 上去。
- 2, 四个 container 就相当于有四个生产线（Pipeline），四个 Pipeline 都这么干，直到最后的 StandardWrapperValve 拿到资源开始调用 servlet。完成后返回来，一步一步的 valve 按照刚才丢生产原料时的顺序的倒序一次执行。如此才完成了 tomcat 的 Pipeline 的机制。

手写管道模式实现

具体代码见工程代码，通过一个简单的 demo, 我们了解到了，在管道中连接一个或者多个阀门，每一个阀门负责一部分逻辑处理，数据按照规定的顺序往下流。此种模式分解了逻辑处理任务，可方便对某个任务单元进行安装、拆卸，提高流程的可拓展性，可重用性，机动性，灵活性。

源码分析

在 CoyoteAdapter 的 service 方法里，由下面这一句就进入 Container 的。

```
connector.getContainer().getPipeline().getFirst().invoke(request, response);
```

是的，这就是进入 container 迷宫的大门，欢迎来到 Container。

```
c CoyoteAdapter.java x
dHeader( name: "X-Powered-By", POWERED_BY);

327
328
329     boolean async = false;
330     boolean postParseSuccess = false;
331
332     req.getRequestProcessor().setWorkerThreadName(THREAD_NAME.get());
333
334     try {
335         // Parse and set Catalina and configuration specific
336         // request parameters
337         postParseSuccess = postParseRequest(req, request, res, response);
338         if (postParseSuccess) {
339             //检查valves是否我们支持异步机制
340             request.setAsyncSupported(
341                 connector.getService().getContainer().getPipeline().isAsyncSupported());
342             // 调用容器---责任链模式(pipeline) 的开始
343             connector.getService().getContainer().getPipeline().getFirst().invoke(
344                 request, response);
345         }
346     } catch (Exception e) {
347         log.error("Error processing request", e);
348     }
349 }
```

一个 StandardValve

来自 org.apache.catalina.core.StandardEngineValve 的 invoke 方法:

```
c StandardEngineValve.java x
63     * @param request Request to be processed
64     * @param response Response to be produced
65     *
66     * @exception IOException if an input/output error occurred
67     * @exception ServletException if a servlet error occurred
68     */
69 @Override
70 public final void invoke(Request request, Response response)
71     throws IOException, ServletException {
72
73     // 拿到自己包含的host
74     Host host = request.getHost();
75     if (host == null) {
76         response.sendError
77             (HttpServletResponse.SC_BAD_REQUEST,
78              sm.getString( key: "standardEngine.noHost",
79                          request.getServerName()));
80         return;
81     }
82     if (request.isAsyncSupported()) {
83         request.setAsyncSupported(host.getPipeline().isAsyncSupported());
84     }
85
86     // 最为自己pipeline上最后一位工人，负责把原料运给下一个pipeline
87     // 这是把所有pipeline串联起来的关键
88     host.getPipeline().getFirst().invoke(request, response);
89
90 }
```

其他的类似 StandardHostValve、StandardContextValve、StandardWrapperValve

Tomcat 中定制阀门

管道机制给我们带来了更好的拓展性，例如，你要添加一个额外的逻辑处理阀门是很容易的。

1. 自定义个阀门 PrintIPValve，只要继承 ValveBase 并重写 invoke 方法即可。注意在 invoke 方法中一定要执行调用下一个阀门的操作，否则会出现异常。

```
public class PrintIPValve extends ValveBase{  
    @Override  
    public void invoke(Request request, Response response) throws IOException, ServletException {  
        System.out.println("-----自定义阀门 PrintIPValve:"+request.getRemoteAddr());  
        getNext().invoke(request,response);  
    }  
}
```

2. 配置 Tomcat 的核心配置文件 server.xml, 这里把阀门配置到 Engine 容器下，作用范围就是整个引擎，也可以根据作用范围配置在 Host 或者是 Context 下

```
<Valve className="org.apache.catalina.valves.PrintIPValve"/>
```

3. 源码中是直接可以有效果，但是如果是运行版本，则可以将这个类导出成一个 Jar 包放入 Tomcat/lib 目录下，也可以直接将.class 文件打包进 catalina.jar 包中。

Tomcat 中提供常用的阀门

AccessLogValve，请求访问日志阀门，通过此阀门可以记录所有客户端的访问日志，包括远程主机 IP，远程主机名，请求方法，请求协议，会话 ID，请求时间，处理时长，数据包大小等。它提供任意参数化的配置，可以通过任意组合来定制访问日志的格式。

JDBCAccessLogValve，同样是记录访问日志的阀门，但是它有助于将访问日志通过 JDBC 持久化到数据库中。

ErrorReportValve，这是一个讲错误以 HTML 格式输出的阀门

PersistentValve，这是对每一个请求的会话实现持久化的阀门

RemoteAddrValve，访问控制阀门。可以通过配置决定哪些 IP 可以访问 WEB 应用

RemoteHostValve，访问控制阀门，通过配置决定哪些主机名可以访问 WEB 应用

RemoteIpValve，针对代理或者负载均衡处理的一个阀门，一般经过代理或者负载均衡转发的请求都将自己的 IP 添加到请求头”X-Forwarded-For”中，此时，通过阀门可以获取访问者真实的 IP。

SemaphoreValve，这个是一个控制容器并发访问的阀门，可以作用在不同容器上。

第三节 Tomcat 源码分析（类加载与类加载器）

Tomcat 的挑战

Tomcat 上可以部署多个项目

Tomcat 的一般部署，可以通过多种方式启动一个 Tomcat 部署多个项目，那么 Tomcat 在设计时会遇到什么挑战呢？

Tomcat 运行时需要加载哪些类

Tomcat 中的多个项目可能存在相同的类

Tomcat 中类加载的挑战

源码分析彻底弄懂 Tomcat 的类加载

类加载与类加载器

类加载

类加载：主要是将.class 文件中的二进制字节读入到 JVM 中

我们可以看到因为这个定义，所以并没有规定一定是要磁盘加载文件，可以通过网络，内存什么的加载。只要是二进制流字节数据，JVM 就认。

类加载过程：

- 1.通过类的全限定名获取该类的二进制字节流；
- 2.将字节流所代表的静态结构转化为方法区的运行时数据结构
- 3.在内存中生成一个该类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口

类加载器

定义：JVM 设计者把“类加载”这个动作放到 java 虚拟机外部去实现，以便让应用程序决定如何获取所需要的类。实现这个动作的代码模块成为“类加载器”

类与类加载器

对于任何一个类，都需要由加载它的类加载器和这个类来确定其在 JVM 中的唯一性。也就是说，两个类来源于同一个 Class 文件，并且被同一个类加载器加载，这两个类才相等。

注意：这里所谓的“相等”，一般使用 instanceof 关键字做判断。

类加载器与双亲委派模型

类加载器

启动类加载器：该加载器使用 C++ 语言实现，属于虚拟机自身的一部分。

启动类加载器（Bootstrap ClassLoader）：负责加载 JAVA_HOME\lib 目录中并且能被虚拟机识别的类库加载到 JVM 内存中，如果名称不符合的类库即使在 lib 目录中也不会被加载。该类加载器无法被 java 程序直接引用

拓展类加载器与应用程序类加载器：另一部分就是所有其它的类加载器，这些类加载器是由 Java 语言实现，独立于 JVM 外部，并且全部继承抽象类 java.lang.ClassLoader。

扩展类加载器(Extension ClassLoader):该加载器主要负责加载 JAVA_HOME\lib\ext 目录中的类库，开发者可以使用扩展加载器。

应用程序类加载器（Application ClassLoader）:该类加载器也称为系统加载器，它负责加载用户类路径(Classpath)上所指定的类库，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器

双亲委派模型

定义: 双亲委派模型的工作过程为: 如果一个类加载器收到了类请求, 它首先不会自己去尝试加载这个类, 而是把这个请求委派给父加载器去完成, 每一层都是如此, 因此所有类加载的请求都会传到启动类加载器, 只有当父加载器无法完成该请求时, 子加载器才去自己加载。

实现方式: 该模型要求除了顶层的启动类加载器外, 其余的类加载器都应当有自己的父类加载器。子类加载器不是以继承的关系来实现, 而是通过组合关系来复用父加载器的代码。

意义: 好处双亲委派模型的好处就是 java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如: Object, 无论那个类加载器去加载该类, 最终都是由启动类加载器进行加载的, 因此 Object 类在程序的各种类加载环境中都是一个类。如果不用改模型, 那么 java.lang.Object 类存放在 classpath 中, 那么系统中就会出现多个 Object 类, 程序变得很混乱。

双亲委派模型

从虚拟机的角度来说, 有两种不同的类加载器: 一种是启动类加载器 (Bootstrap ClassLoader), 该加载器使用 C++ 语言实现, 属于虚拟机自身的一部分。另一部分就是所有其它的类加载器, 这些类加载器是由 Java 语言实现, 独立于 JVM 外部, 并且全部继承抽象类 java.lang.ClassLoader.

从 java 开发人员的角度看, 大部分 java 程序会用到以下三种系统提供的类加载器:

1、启动类加载器 (Bootstrap ClassLoader): 负责加载 JAVA_HOME\lib 目录中并且能被虚拟机识别的类库加载到 JVM 内存中, 如果名称不符合的类库即使在 lib 目录中也不会被加载。该类加载器无法被 java 程序直接引用。

2、扩展类加载器(Extension ClassLoader): 该加载器主要负责加载 JAVA_HOME\lib\ext 目录中的类库, 开发者可以使用扩展加载器。

3、应用程序类加载器 (Application ClassLoader) : 该类加载器也称为系统加载器, 它负责加载用户类路径(Classpath)上所指定的类库, 开发者可以直接使用该类加载器, 如果应用程序中没有自定义过自己的类加载器, 一般情况下这个就是程序中默认的类加载器。

JVM 中的类加载器源码分析

AppClassLoader

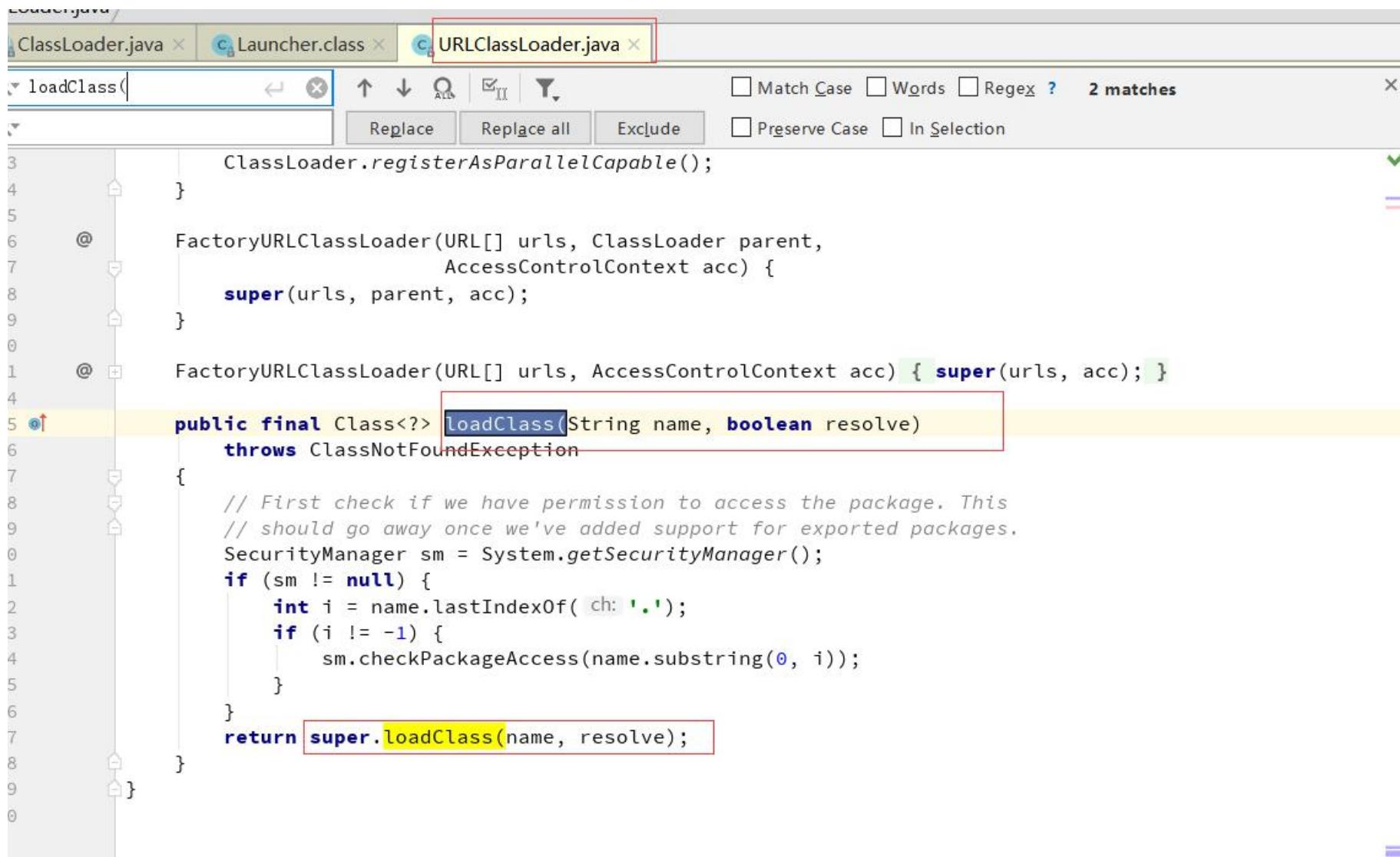
ClassLoader.java x Launcher.class x

Decompiled .class file, bytecode version: 52.0 (Java 8)

```
9
0     static class AppClassLoader extends URLClassLoader {
1         final URLClassPath ucp = SharedSecrets.getJavaNetAccess().getURLClassPath(urlClassLoader: this);
2
3         public static ClassLoader getAppClassLoader(final ClassLoader var0) throws IOException {...}
4
5         @AppClassLoader(URL[] var1, ClassLoader var2) {...}
6
7         public Class<?> loadClass(String var1, boolean var2) throws ClassNotFoundException {
8             int var3 = var1.lastIndexOf(ch: 46);
9             if (var3 != -1) {
0                 SecurityManager var4 = System.getSecurityManager();
1                 if (var4 != null) {
2                     var4.checkPackageAccess(var1.substring(0, var3));
3                 }
4             }
5
6             if (this.ucp.knownToNotExist(var1)) {
7                 Class var5 = this.findLoadedClass(var1);
8                 if (var5 != null) {
9                     if (var2) {
0                         this.resolveClass(var5);
1                     }
2
3                     return var5;
4                 } else {
5                     throw new ClassNotFoundException(var1);
6                 }
7             } else {
8                 return super.loadClass(var1, var2);
9             }
1         }
2
3     }
4
5 }
6
7 }
```

URLClassLoader





The screenshot shows a Java code editor interface with the following details:

- File Tabs:** ClassLoader.java, Launcher.class, URLClassLoader.java (highlighted with a red border).
- Search Bar:** Contains the search term "loadClass()", search mode "All", and search options: Match Case, Words, Regex, Preserve Case, In Selection. It also displays "2 matches".
- Code Preview:** Shows the first few lines of the URLClassLoader.java source code.
- Search Results:** Two matches are highlighted:
 - Line 5: `public final Class<?> loadClass(String name, boolean resolve)`
 - Line 7: `return super.loadClass(name, resolve);`

```
ClassLoader.registerAsParallelCapable();  
}  
}  
@ FactoryURLClassLoader(URL[] urls, ClassLoader parent,  
                         AccessControlContext acc) {  
    super(urls, parent, acc);  
}  
@ FactoryURLClassLoader(URL[] urls, AccessControlContext acc) { super(urls, acc); }  
public final Class<?> loadClass(String name, boolean resolve)  
throws ClassNotFoundException  
{  
    // First check if we have permission to access the package. This  
    // should go away once we've added support for exported packages.  
    SecurityManager sm = System.getSecurityManager();  
    if (sm != null) {  
        int i = name.lastIndexOf('.');  
        if (i != -1) {  
            sm.checkPackageAccess(name.substring(0, i));  
        }  
    }  
    return super.loadClass(name, resolve);  
}
```

SecureClassLoader



```
* classes with an associated code source and permissions which are
* retrieved by the system policy by default.
*
* @author Li Gong
* @author Roland Schemers
*/
public class SecureClassLoader extends ClassLoader {
    /*
     * If initialization succeed this is set to true and security checks will
     * succeed. Otherwise the object is not initialized and the object is
     * useless.
     */
    private final boolean initialized;

    // HashMap that maps CodeSource to ProtectionDomain
    // @GuardedBy("pdcache")
    private final HashMap<CodeSource, ProtectionDomain> pdcache =
        new HashMap<>( initialCapacity: 11);

    private static final Debug debug = Debug.getInstance("scl");

    static {
        ClassLoader.registerAsParallelCapable();
    }

    /**
     * Creates a new SecureClassLoader using the specified parent
     * class loader for delegation.
    
```

Tomcat 中的类加载解决方案

Tomcat 类加载的考虑

隔离性

Web 应用类库相互隔离，避免依赖库或者应用包相互影响，比如有两个 Web 应用，一个采用了 Spring 4,一个采用了 Spring 5,而如果如果采用同一个类加载器，那么 Web 应用将会由于 jar 包覆盖而无法启动成功。



The screenshot shows a Java project named "ref-comet" in an IDE. The project structure on the left includes .idea, .settings, src (with main and java subfolders), and com.xiangxue (with config, emitter, SseController, and normal subfolders). The right side displays the contents of the pom.xml file, specifically the dependency section:

```
<version>4.3.11.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.3.11.RELEASE</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

The screenshot shows the IntelliJ IDEA interface with a Maven project named "ref-comet-5". The left pane displays the project structure, including ".idea", ".settings", "src" (with "main" and "webapp" subfolders), and "target" (with "classes", "maven-status", and "ref-comet-5" subfolders). The right pane shows the content of the "pom.xml" file:

```
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.1.5.RELEASE</version>
</dependency>

<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-test</artifactId>
<version>5.1.5.RELEASE</version>
<scope>test</scope>
</dependency>

<dependency>
<groupId>javax.servlet</groupId>
```

灵活性

因为隔离性，所以 Web 应用之间的类加载器相互独立，如果一个 Web 应用重新部署时，该应用的类加载器重新加载，同时不会影响其他 web 应用。

比如：不需要重启 Tomcat 的创建 xml 文件的类加载，

还有 context 元素中的 reloadable 字段：如果设置为 true 的话，Tomcat 将检测该项目是否变更，当检测到变更时，自动重新加载 Web 应用。

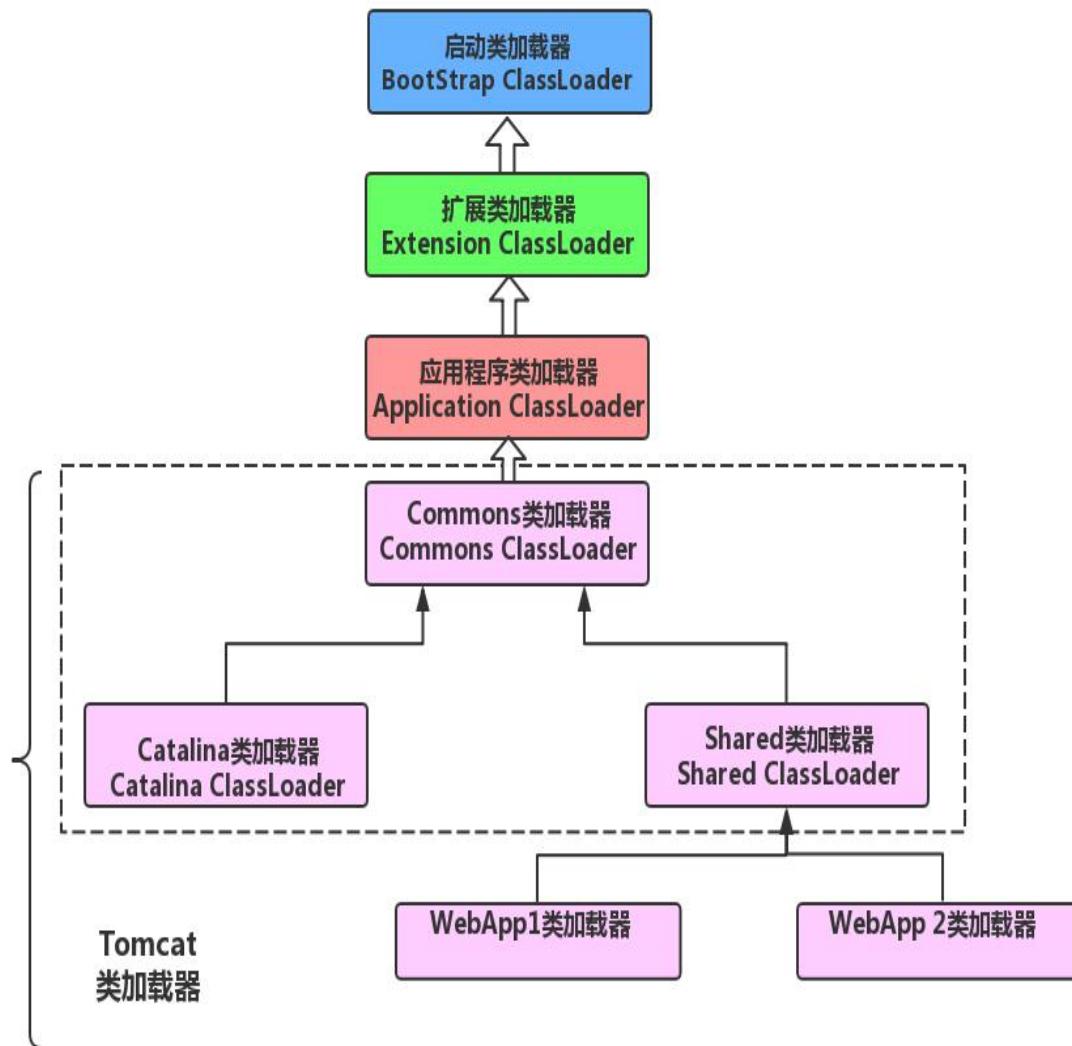
性能

由于每一个 Web 应用都有一个类加载器，所以在 Web 应用在加载类时，不会搜索其他 Web 应用包含的 Jar 包，性能自然高于只有一个类加载的情况。

Tomcat 中的类加载器

Tomcat 提供 3 个基础类加载器（common、catalina、shared）和 Web 应用类加载器。





Tomcat 中的类加载源码分析

三个基础类加载器

介绍

3 个基础类加载器的加载路径在 `catalina.properties` 配置，默认情况下，3 个基础类加载器的实例都是一个。

`createClassLoader` 调用 `ClassLoaderFactory` 属于一种工厂模式，并且都是使用 `URLClassLoader`

Bootstrap.java

```
143  
144     // 初始化三个类加载器以及确定父子关系  
145     private void initClassLoaders() {  
146         try {  
147             // commonLoader 的加载路径为 common.loader  
148             commonLoader = createClassLoader( name: "common", parent: null );  
149             if( commonLoader == null ) {  
150                 // no config file, default to this loader - we might be in a 'single' env.  
151                 commonLoader = this.getClass().getClassLoader();  
152             }  
153             // 加载路径为 server.loader, 默认为空, 父类加载器为 commonLoader  
154             catalinaLoader = createClassLoader( name: "server", commonLoader );  
155             // 加载路径为 shared.loader, 默认为空, 父类加载器为 commonLoader  
156             sharedLoader = createClassLoader( name: "shared", commonLoader );  
157         } catch (Throwable t) {  
158             handleThrowable(t);  
159             log.error( message: "Class loader creation threw exception", t );  
160             System.exit( status: 1 );  
161         }  
162     }  
163
```

```
private ClassLoader createClassLoader(String name, ClassLoader parent)
throws Exception {

    String value = CatalinaProperties.getProperty(name + ".loader");
    // catalinaLoader与sharedLoader的加载路径均为空，所以直接返回commonLoader对象，默认3者为同一个对象
    if ((value == null) || (value.equals("")))
        return parent;

    value = replace(value);
```

```
catalina.properties
52 #      may not appear in a path.
53 common.loader="${catalina.base}/lib","${catalina.base}/lib/*.jar","${catalina.home}/lib","${catalina.home}/lib/*.jar"
54
55
56 server.loader=
57
58
59 shared.loader=
```

默认情况三个是一个实例，但是可以通过修改配置创建 3 个不同的类加载机制，使它们各司其职。

举个例子：如果我们不想实现自己的会话存储方案，并且这个方案依赖了一些第三方包，我们不希望这些包对 Web 应用可见，因此我们可以配置 server.loader 创建独立的 Catalina 类加载器。

共享性：

Tomcat 通过 Common 类加载器实现了 Jar 包在应用服务器与 Web 应用之间的共享，

通过 Shared 类加载器实现了 Jar 包在 Web 应用之间的共享

通过 Catalina 类加载器加载服务器依赖的类。

```
org > apache > catalina > startup > Bootstrap
```

```
perLoader.java x JspCompilationContext.java x JspServletWrapper.java x Bootstrap.java x
```

```
// 加载启动类Catalina并调用其process()方法
if (log.isDebugEnabled())
    log.debug("Loading startup class");
Class<?> startupClass = catalinaLoader.loadClass( name: "org.apache.catalina.startup.Catalina" );
Object startupInstance = startupClass.getConstructor().newInstance();

// 设置共享扩展类加载器sharedLoader
if (log.isDebugEnabled())
    log.debug("Setting startup class properties");
String methodName = "setParentClassLoader";
Class<?> paramTypes[] = new Class[1];
paramTypes[0] = Class.forName("java.lang.ClassLoader");
Object paramValues[] = new Object[1];
//这里把shared加载器传递给catalina
paramValues[0] = sharedLoader;
Method method =
    startupInstance.getClass().getMethod(methodName, paramTypes);
method.invoke(startupInstance, paramValues);

catalinaDaemon = startupInstance;
}
```

类加载工厂

因为类加载需要做很多事情，比如读取字节数组、验证、解析、初始化等。而 Java 提供的 URLClassLoader 类能够方便的将 Jar、Class 或者网络资源加载到内存中。而 Tomcat 中则用一个工厂类， ClassLoaderFactory 把创建类加载器的细节进行封装，可以通过它方便的创建自定义类加载器。



The screenshot shows a Java code editor interface with several tabs at the top: StandardServer.java, Catalina.java, Bootstrap.java (which is the active tab), ClassLoaderFactory.java, and StandardService.java.

The search bar at the top indicates a search for "initClassLoaders" has been performed, resulting in 2 matches.

The code editor displays the following Java code:

```
3     }
4
5     /**
6      * Initialize daemon.
7      * @throws Exception Fatal initialization error
8     */
9     public void init() throws Exception {
10        System.out.println("Bootstrap--init()");
11        //类加载器初始化
12        initClassLoaders();
13    }
```

A red rectangular box highlights the entire body of the `init` method, from the opening brace to the closing brace. A yellow horizontal bar highlights the line containing the call to `initClassLoaders()`.

```
//初始化三个类加载器以及确定父子关系
private void initClassLoaders() {
    try {
        // commonLoader的加载路径为common.loader
        commonLoader = createClassLoader( name: "common", parent: null );
        if( commonLoader == null ) {
            // no config file, default to this loader - we might be in a 'single' env.
            commonLoader=this.getClass().getClassLoader();
        }
        // 加载路径为server.loader, 默认为空, 父类加载器为commonLoader
        catalinaLoader = createClassLoader( name: "server", commonLoader );
        // 加载路径为shared.loader, 默认为空, 父类加载器为commonLoader
        sharedLoader = createClassLoader( name: "shared", commonLoader );
    } catch (Throwable t) {
        handleThrowable(t);
        log.error( message: "Class loader creation threw exception", t );
        System.exit( status: 1 );
    }
}
```



```
private ClassLoader createClassLoader(String name, ClassLoader parent)
throws Exception {

String value = CatalinaProperties.getProperty(name + ".loader");
// catalinaLoader与sharedLoader的加载路径均为空, 所以直接返回commonLoader对象, 默认3
if ((value == null) || (value.equals("")))

```



andardServer.java x c Catalina.java x c Bootstrap.java x c ClassLoaderFactory.java x c StandardService.java x

nitClassLoaders ↻ ✎ ⌂ 🔍 ↻ 🔍 Match Case Words Regex ? 2 matches

Replace Replace all Exclude Preserve Case In Selection

```
        new Repository(repository, RepositoryType.URL));
    continue;
} catch (MalformedURLException e) {
    // Ignore
}

// 本地仓库
if (repository.endsWith("*.jar")) {
    repository = repository.substring(
        0, repository.length() - "*.jar".length());
    repositories.add(
        new Repository(repository, RepositoryType.GLOB));
} else if (repository.endsWith(".jar")) {
    repositories.add(
        new Repository(repository, RepositoryType.JAR));
} else {
    repositories.add(
        new Repository(repository, RepositoryType.DIR));
}
}

return ClassLoaderFactory.createClassLoader(repositories, parent);
}
```



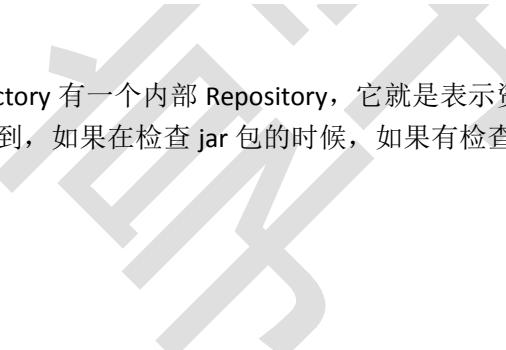
```
Server.java ✘  C Catalina.java ✘  C Bootstrap.java ✘  C ClassLoaderFactory.java ✘  C StandardService.java ✘
    * @exception Exception if an error occurs constructing the class loader
    */
public static ClassLoader createClassLoader(List<Repository> repositories,
                                             final ClassLoader parent)
throws Exception {

    if (log.isDebugEnabled())
        log.debug("Creating new class loader");

    // Construct the "class path" for this class loader
    Set<URL> set = new LinkedHashSet<>();

    if (repositories != null) {
        for (Repository repository : repositories) {
            if (repository.getType() != RepositoryType.URL) {
                URL url = buildClassLoaderUrl(repository.getLocation());
                if (log.isDebugEnabled())
                    log.debug(" Including URL " + url);
                set.add(url);
            }
        }
    }
}
```





```
StandardServer.java ×  c Catalina.java ×  c Bootstrap.java ×  c ClassLoaderFactory.java ×  c StandardService.java
```

```
        }
```

```
    }
```

```
    private static URL buildClassLoaderUrl(File file) throws MalformedURLException {
        // Could be a directory or a file
        String urlString = file.toURI().toString();
        urlString = urlString.replaceAll(regex: "!/", replacement: "%21/");
        return new URL(urlString);
    }
```

```
    public enum RepositoryType {
        DIR, //表示整个目录下的资源，包括所有Class、jar包以及其他类型资源
        GLOB, //表示整个目录下所有的Jar包资源，仅仅是.jar后缀的资源
        JAR, //表示单个Jar包资源
        URL //表示从URL上获取的Jar包资源
    }
```

使用加载器工厂的好处

1. ClassLoadFactory 有一个内部 Repository，它就是表示资源的类，资源的类型用一个 RepositoryType 的枚举表示。
2. 同时我们看到，如果在检查 jar 包的时候，如果有检查的 URL 地址的如果检查有异常就忽略掉，可以确保部分类加载正确。

尽早设置线程上下文类加载器

```
apache > catalina > startup > Bootstrap
Bootstrap.java X catalina.properties X Thread.java X ClassLoaderFactory.java X UP
* Initialize daemon.
* @throws Exception Fatal initialization error
*/
public void init() throws Exception {
    System.out.println("Bootstrap--init()");
    //类加载器初始化
    initClassLoaders();

    //设置线程上下文类加载器来解决有可能的ClassNotFoundException问题
    Thread.currentThread().setContextClassLoader(catalinaLoader);

    SecurityClassLoader.securityClassLoader(catalinaLoader);
}
```

每一个运行线程中有一个成员 ContextClassLoader, 用于在运行时动态载入其他类, 当程序中没有显示声明由哪个类加载器去加载哪个类, 将默认由当前线程类加载器加载, 所以一般系统默认的 ContextClassLoader 是系统类加载器。

一般在实际的系统上, 使用线程上下文类加载器, 可以设置不同的加载方式, 这个也是 Java 灵活的类加载方式的体现, 也可以很轻松的打破双亲委派模式, 同时也会避免类加载的异常。

Webapp 类加载器

每个 web 应用会对一个 Context 节点, 在 JVM 中就会对应一个 org.apache.catalina.core.StandardContext 对象, 而每一个 StandardContext 对象内部都有一个加载器实例 loader 实例变量。这个 loader 实际上是 WebappLoader 对象。而每一个 WebappLoader 对象内部关联了一个 classLoader 变量(就这个类的定义中, 可以看到该变量的类型是 org.apache.catalina.loader.WebappClassLoader)。

所以, 这里一个 web 应用会对应一个 StandardContext 一个 WebappLoader 一个 WebappClassLoader。

一个 web 应用对应着一个 StandardContext 实例，每个 web 应用都拥有独立 web 应用类加载器(WebappClassLoader)，这个类加载器在 StandardContext.startInternal() 中被构造了出来。



The screenshot shows a Java code editor with the file `StandardContext.java` open. The code is part of the Apache Tomcat source code. The `startInternal()` method is highlighted, and the line `if(log.isDebugEnabled())` is also highlighted with a yellow background. The code snippet is as follows:

```
30     * that prevents this component from being used
31     */
32     @Override
33     protected synchronized void startInternal() throws LifecycleException {
34
35         if(log.isDebugEnabled())
36             log.debug("Starting " + getBaseName());
37
38         // Send j2ee.state.starting notification
39         if (this.getObjectName() != null) {
40             Notification notification = new Notification( type: "j2ee.state.starting",
41                 this.getObjectName(), sequenceNumber.getAndIncrement());
42             broadcaster.sendNotification(notification);
43         }
44     }
```



```
java>org>apache>catalina>core>StandardContext
```

Bootstrap.java X ContainerBase.java X Container.java X StandardContext.java X WebappLoader.java X Context.java

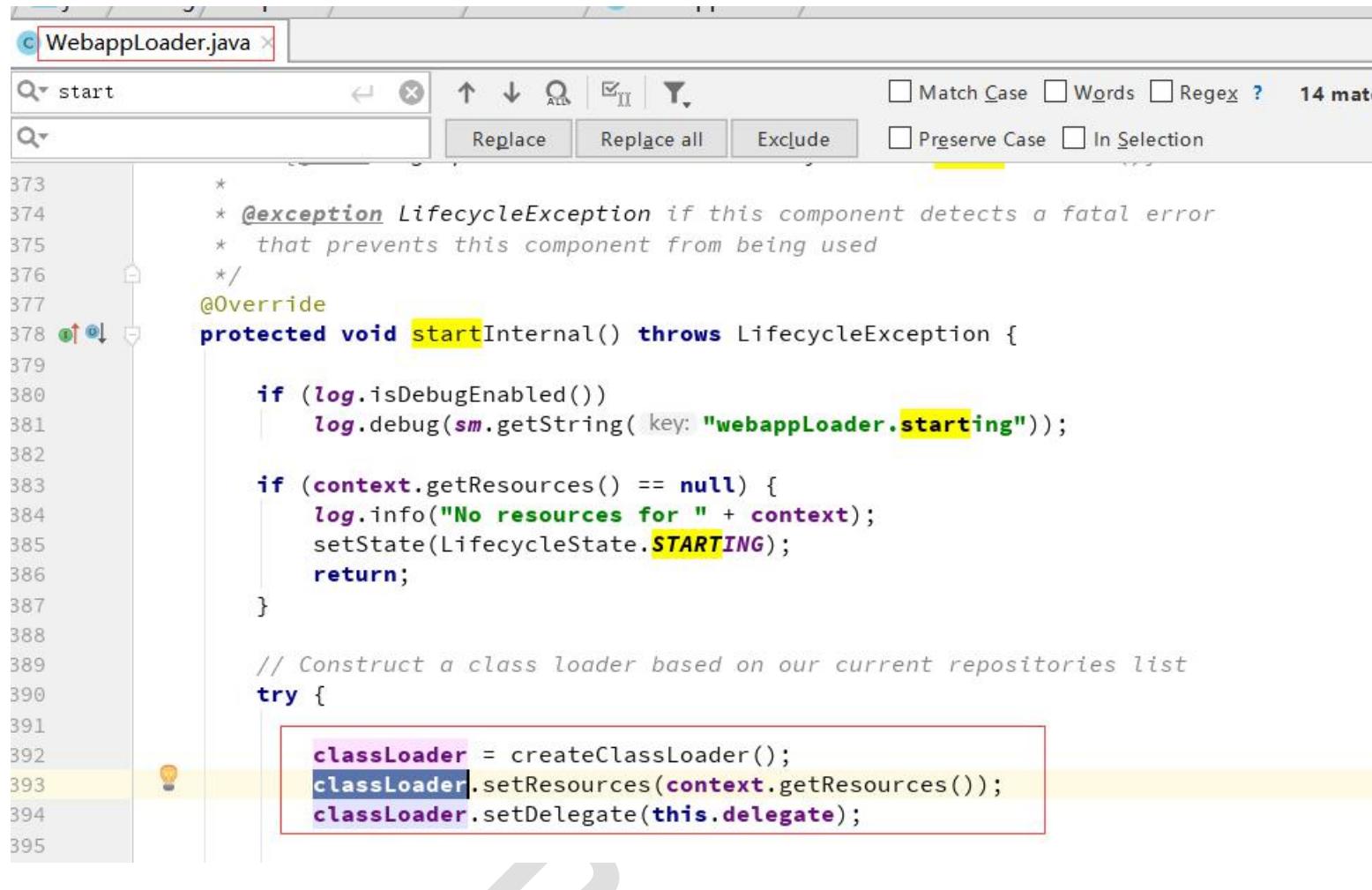
Q startint ↺ 🔍 ⌂ ⌄ Match Case Words Regex ? 2 matches

Q Replace Replace all Exclude Preserve Case In Selection

```
019     if (ok) {  
020         resourcesStart();  
021     }  
022     //Webapp类加载的设置  
023     //这里的获取和设置类加载  
024     if (getLoader() == null) {  
025         WebappLoader webappLoader = new WebappLoader(getParentClassLoader());  
026         webappLoader.setDelegate(getDelegate());  
027         setLoader(webappLoader);  
028     }  
029  
030     // An explicit cookie processor hasn't been specified: use the default
```

注意这里：设置加载器和获取加载器都使用了读写锁机制，确保多线程情况下对共享资源的访问不会出现问题。

同时因为 Tomcat 的生命周期管理，必定会调用 WebappLoader.java 的 startInternal()方法，该方法中 new 出了



The screenshot shows an IDE interface with the file `WebappLoader.java` open. A search dialog is visible at the top, with the search term `start` entered. The search results count is `14 matches`. The code editor displays the following Java code:

```
373     *
374     * @exception LifecycleException if this component detects a fatal error
375     * that prevents this component from being used
376     */
377     @Override
378     protected void startInternal() throws LifecycleException {
379
380         if (log.isDebugEnabled())
381             log.debug(sm.getString(key: "webappLoader.starting"));
382
383         if (context.getResources() == null) {
384             log.info("No resources for " + context);
385             setState(LifecycleState.STARTING);
386             return;
387         }
388
389         // Construct a class loader based on our current repositories list
390         try {
391
392             classLoader = createClassLoader();
393             classLoader.setResources(context.getResources());
394             classLoader.setDelegate(this.delegate);
395         }
```

A red rectangular box highlights the following code block:

```
classLoader = createClassLoader();
classLoader.setResources(context.getResources());
classLoader.setDelegate(this.delegate);
```

```
 501     /**
 502      * Create associated classLoader.
 503      */
 504     private WebappClassLoaderBase createClassLoader()
 505         throws Exception {
 506
 507         Class<?> clazz = Class.forName(loaderClass);
 508         WebappClassLoaderBase classLoader = null;
 509
 510         if (parentClassLoader == null) {
 511             parentClassLoader = context.getParentClassLoader();
 512         }
 513         Class<?>[] argTypes = { ClassLoader.class };
 514         Object[] args = { parentClassLoader };
 515         Constructor<?> constr = clazz.getConstructor(argTypes);
 516         classLoader = (WebappClassLoaderBase) constr.newInstance(args);
 517
 518         return classLoader;
 519     }
 520 }
```

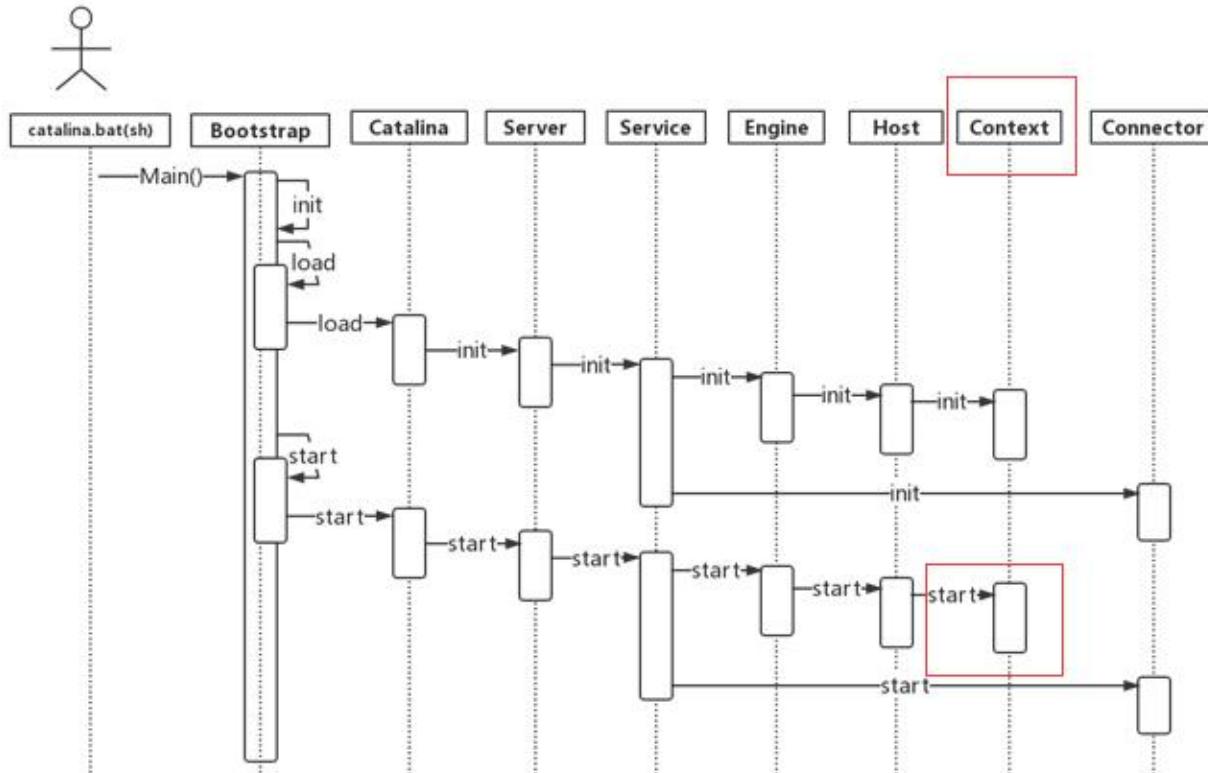
所以总结一句话，如果没有弄懂 Tomcat 的启动流程，以及弄懂 Tomcat 的生命周期的管理，很多地方的源码是没有办法没有看懂，所以看源码也有一个先后顺序。

热加载源码分析

当配置信息中有 reloadable 的属性，并且值为 true 时，Tomcat 怎么去完成这个过程呢？

```
<Context path="/" docBase="D:\work_tomcat\ref-comet" reloadable="true"/>
```

还是看源码，据 Tomcat 的启动流程，我们分析下 Context 的初始化 start 方法，根据之前的课程我们可知，Context 只是一个接口，具体实现类是 StandardContext，我们分析下 startInternal 方法（此方法由之前的抽屉骨架类中的 start 方法触发）



我发现有一个线程启动的方法 `threadStart()`,

apache-tomcat-8.5.42-src > java > org > apache > catalina > core > StandardContext

Bootstrap.java ContainerBase.java StandardContext.java Engine.java

```
        }
        // Load and initialize all "load on startup" servlets
        if (ok) {
            if (!loadOnStartup(findChildren())){
                log.error(sm.getString( key: "standardContext.servletFail"));
                ok = false;
            }
        }
        // 启动ContainerBackgroundProcessor
        super.threadStart();
    } finally {
        // Unbinding thread
    }
}
```

Find Usages

Cut Ctrl+X
Copy Ctrl+C
Copy as Plain Text
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Paste from History... Ctrl+Shift+V
Paste Simple Ctrl+Alt+Shift+V
Column Selection Mode Alt+Shift+Insert
Find Usages Ctrl+G
Refactor >
Folding >
Analyze >
Search with Google
Go To >
Generate... Alt+Insert
Recompile 'StandardContext.java' Ctrl+Shift+F9

Usages of threadStart() in All Places

Method

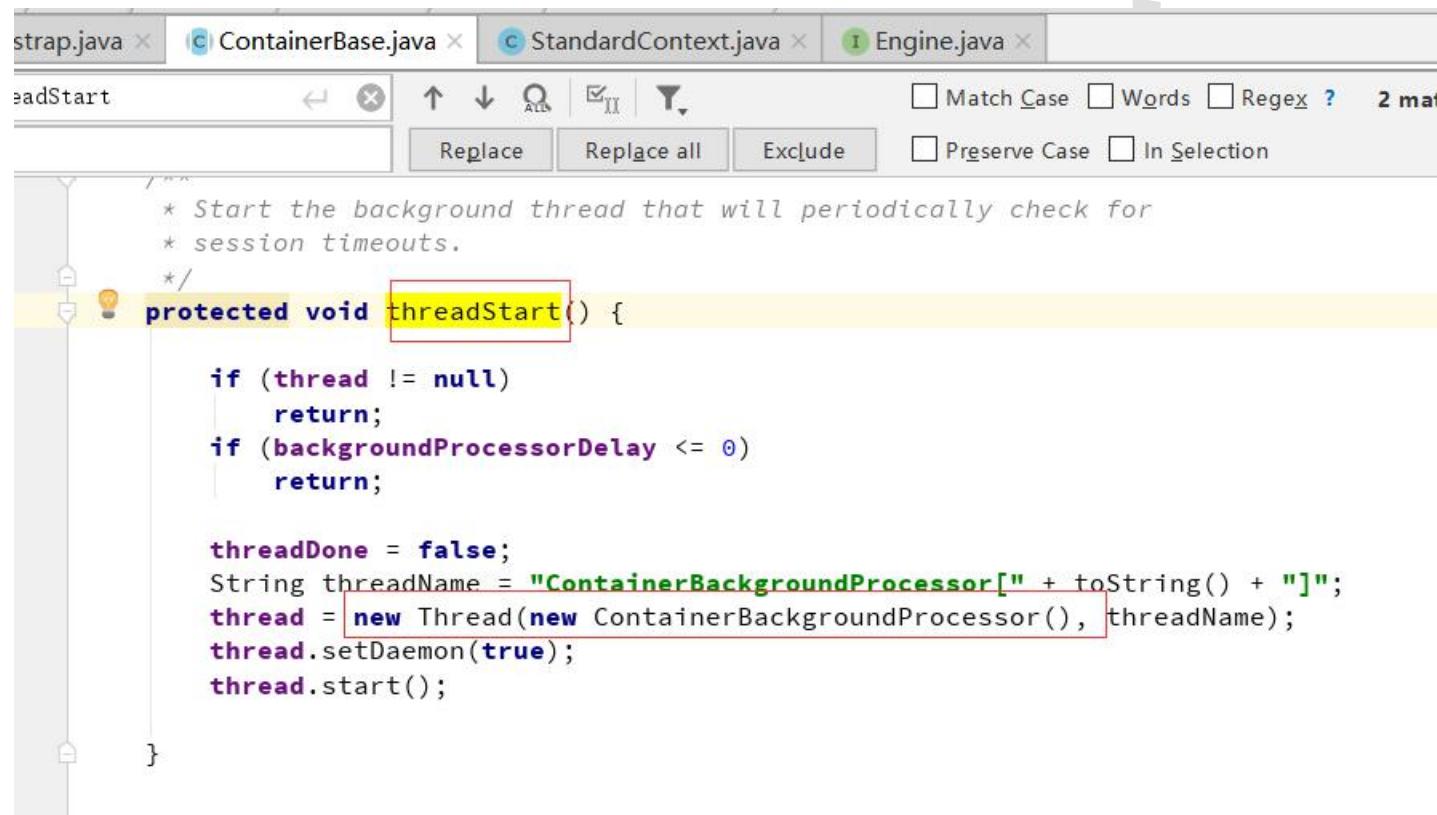
- threadStart()

Found usages 2 usages

- Unclassified usage 2 usages
 - org.apache.catalina.core 2 usages
 - ContainerBase 1 usage
 - startInternal() 1 usage
 - StandardContext 1 usage
 - startInternal() 1 usage

966 threadStart();
5278 super.threadStart();

由上图我们知道，这个是父类中调用，



```
strap.java ✘ ContainerBase.java ✘ StandardContext.java ✘ Engine.java ✘
readStart ⌂ X ↑ ↓ ALD II T Match Case Words Regex ? 2 mat
Replace Replace all Exclude Preserve Case In Selection

/*
 * Start the background thread that will periodically check for
 * session timeouts.
 */
protected void threadStart() {
    if (thread != null)
        return;
    if (backgroundProcessorDelay <= 0)
        return;

    threadDone = false;
    String threadName = "ContainerBackgroundProcessor[" + toString() + "]";
    thread = new Thread(new ContainerBackgroundProcessor(), threadName);
    thread.setDaemon(true);
    thread.start();
}
```

```
* Private thread class to invoke the backgroundProcess method
* of this container and its children after a fixed delay.
*/
protected class ContainerBackgroundProcessor implements Runnable {

    @Override
    public void run() {
        Throwable t = null;
        String unexpectedDeathMessage = sm.getString(
            key: "containerBase.backgroundProcess.unexpectedThreadDeath",
            Thread.currentThread().getName());
        try {
            while (!threadDone) {
                try {
                    Thread.sleep( millis: backgroundProcessorDelay * 1000L);
                } catch (InterruptedException e) {
                    // Ignore
                }
                if (!threadDone) {
                    processChildren(container: ContainerBase.this);
                }
            }
        } catch (RuntimeException|Error e) {
            t = e;
            throw e;
        } finally {
```



```
        }

    protected void processChildren(Container container) {
    ClassLoader originalClassLoader = null;

    try {
        if (container instanceof Context) {
            Loader loader = ((Context) container).getLoader();
            // Loader will be null for FailedContext instances
            if (loader == null) {
                return;
            }

            // Ensure background processing for Contexts and Wrappers
            // is performed under the web app's class loader
            originalClassLoader = ((Context) container).bind(usePrivilegedAction: fa
        }
        container.backgroundProcess();
    Container[] children = container.findChildren();
    for (int i = 0; i < children.length; i++) {
        if (children[i].getBackgroundProcessorDelay() <= 0) {
            processChildren(children[i]);
        }
    }
}
```



The screenshot shows the Apache Tomcat source code in the `WebappLoader.java` file. The code is annotated with several red boxes highlighting specific sections:

- A red box surrounds the `@Override` annotation and the `backgroundProcess()` method definition.
- A red box highlights the line `//设置线程类加载器的类加载器 为WebappClassLoader`.
- A red box highlights the line `Thread.currentThread().setContextClassLoader`.
- A red box highlights the line `(WebappLoader.class.getClassLoader())`.
- A red box highlights the line `context.reload();`.

```
java org.apache.catalina.loader WebappLoader
bootstrap.java ✘ ContainerBase.java ✘ Container.java ✘ StandardContext.java ✘ WebappLoader.java ✘ Context.java ✘

    /**
     * 执行定期任务，如重新加载等。此方法将在此容器的类加载上下文中调用。
     */
    @Override
    public void backgroundProcess() {
        if (reloadable && modified()) {
            try {
                //设置线程类加载器的类加载器 为WebappClassLoader
                Thread.currentThread().setContextClassLoader
                    (WebappLoader.class.getClassLoader());
                if (context != null) {
                    //终要执行的重新加载的方法: StandardContext类的reload():
                    context.reload();
                }
            } finally {
                if (context != null && context.getLoader() != null) {
                    Thread.currentThread().setContextClassLoader
                        (context.getLoader().getClassLoader());
                }
            }
        }
    }
}
```

JasperLoader



第四节 Tomcat 性能优化

嵌入式 Tomcat

为什么要嵌入式

为什么需要嵌入式启动，我们由之前一般的 Tomcat 启动可知，Tomcat 组件非常多，启动流程步骤比较多，但是往往我们一般就只需要简单快速的部署一个 Web 项目，所以简单实用的嵌入式 Tomcat 就诞生而来。

部署复杂度

如果按照传统部署，我们需要下载 Tomcat，同时需要配置服务器，同时还需要修改端口，同时也要避免应用系统的 jar 包与服务器中存在的 lib 包的冲突，所有的这些都会增加部署的复杂度，并且这种配置大部分还是一次性的，不可重用。如果你遇到大规模的服务器集群环境（部署 N 多个应用）时，会增加我们的运维成本，如果按照嵌入式启动，这种方式几乎是一键式的，可以把以上问题轻松的解决。

架构约束

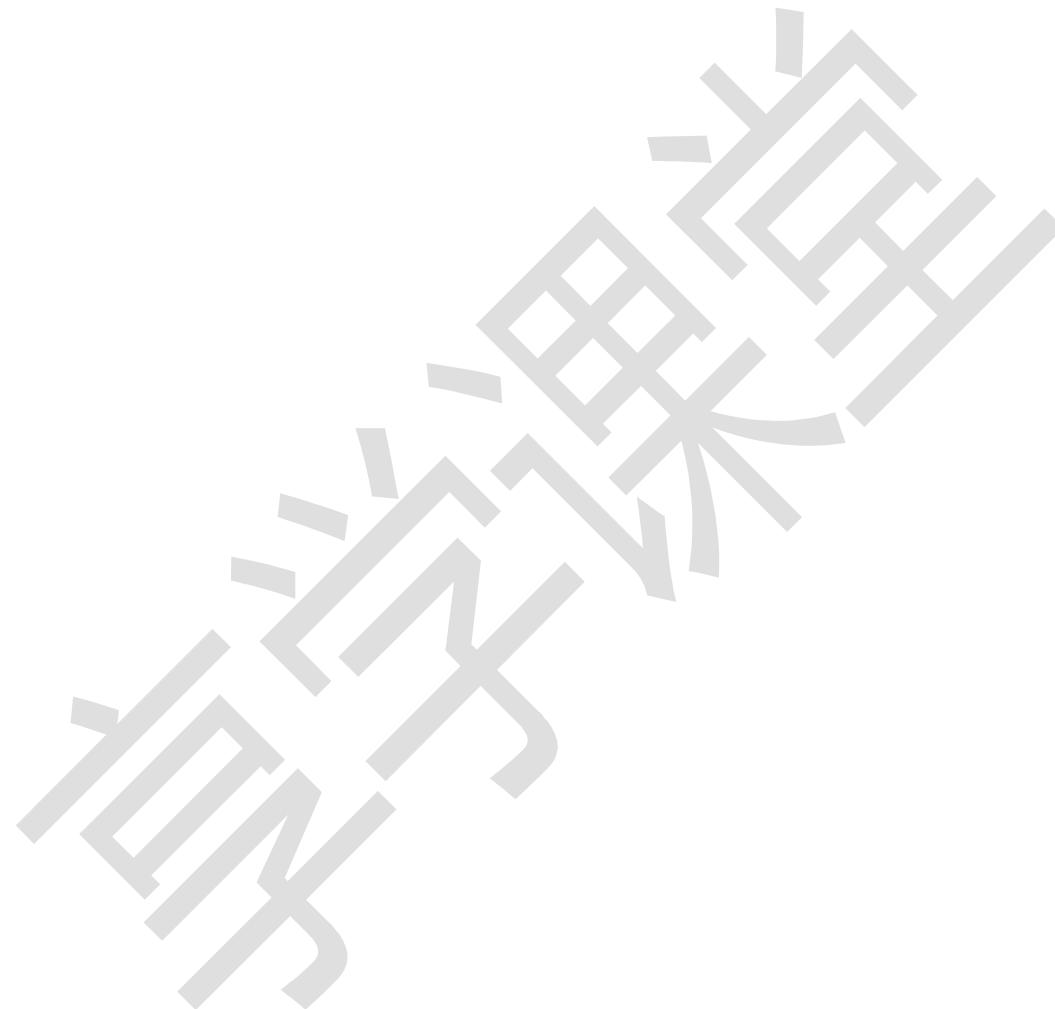
Tomcat 启动的时候默认不单单只启动了 HTTP 协议，还有 AJP 协议等等，如果我们就只想简简单单的启动一个 HTTP 服务，同时不想启动那些多组件，可以使用嵌入式 Tomcat，避免在部署启动时的架构约束。

微服务架构

现在微服务已经是主流的架构，其中微服务中每项服务都拥有自己的进程并利用轻量化机制实现通讯。这些服务都是围绕业务功能建立，可以自动化部署或独立部署。将微服务架构与 Tomcat 技术相结合，可以轻松将系统部署到云服务器上。当前 SpringBoot 支持的嵌入式服务器组件就是 Tomcat.

嵌入式启动实战

启动 *Servlet*



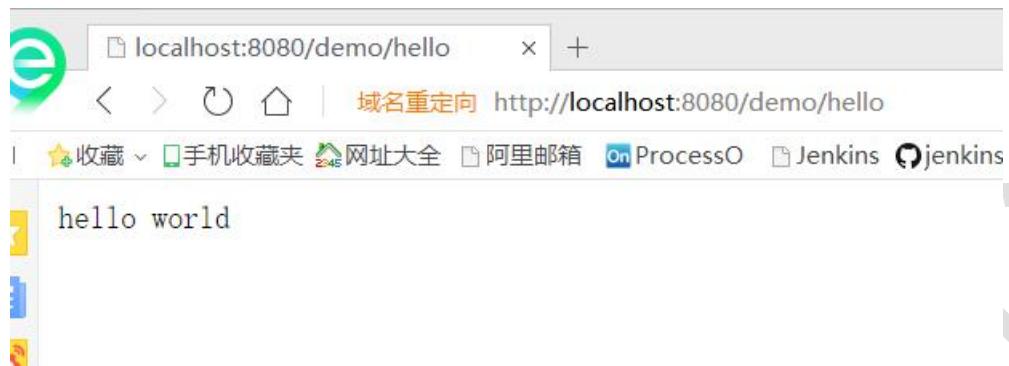
m enjoy-tomcat x

```
15     <properties>
16         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17         <maven.compiler.source>1.8</maven.compiler.source>
18         <maven.compiler.target>1.8</maven.compiler.target>
19     </properties>
20
21     <dependencies>
22         <dependency>
23             <groupId>junit</groupId>
24             <artifactId>junit</artifactId>
25             <version>4.11</version>
26             <scope>test</scope>
27         </dependency>
28         <!--嵌入式Tomcat -->
29         <dependency>
30             <groupId>org.apache.tomcat.embed</groupId>
31             <artifactId>tomcat-embed-core</artifactId>
32             <version>8.5.34</version>
33         </dependency>
34         <!--嵌入式Tomcat的JSP 支持-->
35         <dependency>
36             <groupId>org.apache.tomcat.embed</groupId>
37             <artifactId>tomcat-embed-jasper</artifactId>
38             <version>8.5.34</version>
39         </dependency>
40
41     </dependencies>
42 </project>
43
```

ServletDemo.java

```
8  import javax.servlet.ServletResponse;
9  import javax.servlet.http.HttpServlet;
10 import java.io.IOException;
11
12 /**
13 * @author 【享学课堂】 King老师
14 */
15 public class ServletDemo {
16     public static void main(String[] args) throws Exception {
17         Tomcat tomcat = new Tomcat();
18         HttpServlet servlet = service(req, res) → { res.getWriter().write(s: "hello world"); };
19         Context context = tomcat.addContext(contextPath: "/demo", docBase: null);
20         Tomcat.addServlet(context, servletName: "/hello", servlet);
21         context.addServletMappingDecoded(s: "/hello", s1: "/hello");
22         tomcat.init();
23         tomcat.start();
24         tomcat.getServer().await(); //用于阻塞主程序，等待请求过来
25     }
26 }
```

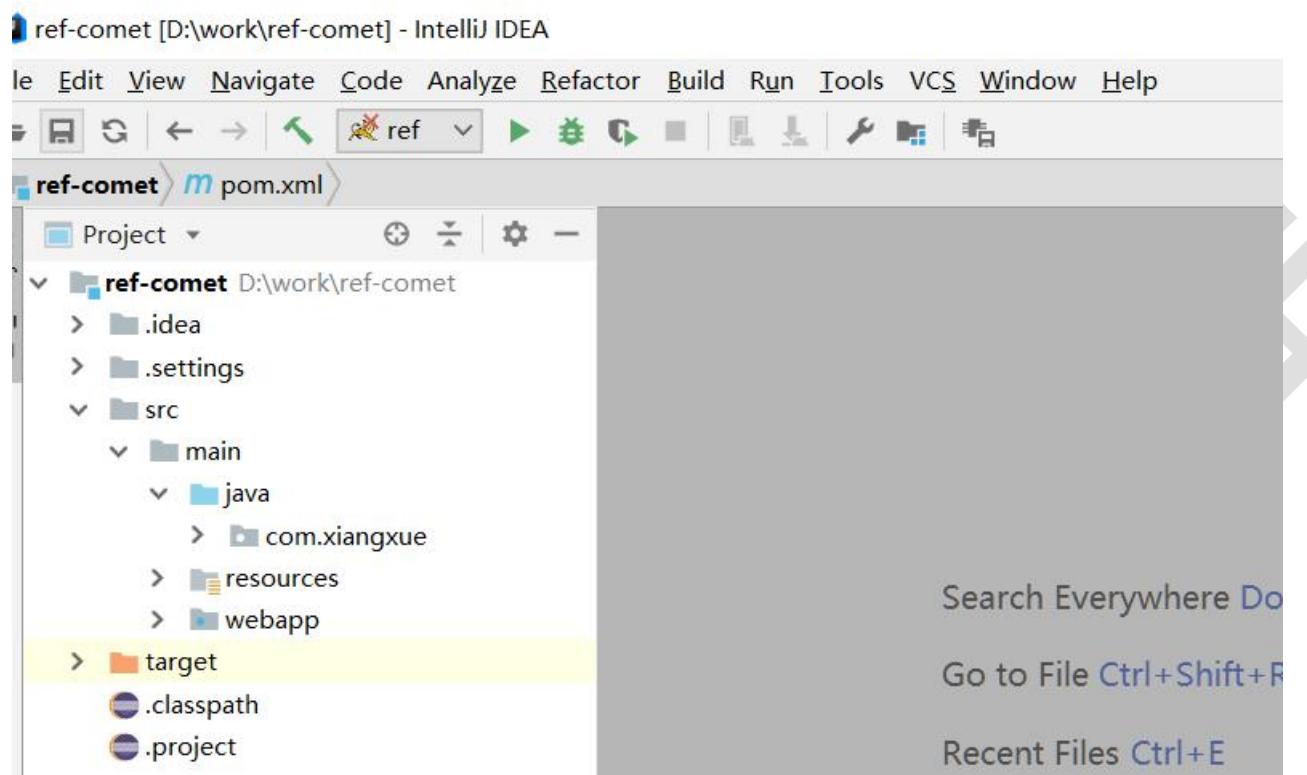




通过实战可知，我们添加了一个 `HttpServlet` 提供对外服务，这样我们的 Tomcat 就可以简单的提供一个 `Servlet` 服务，而不提供页面访问（JSP、HTML 等），可以非常灵活的控制 `Servlet` 的加载以及请求链接的分配，而不受限制于 `Context` 与应用的一一对应关系。

嵌入式启动应用

这个项目工程是一个服务器推送的工程，包含了 JSP



```
m enjoy-tomcat x
15   <properties>
16     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
17     <maven.compiler.source>1.8</maven.compiler.source>
18     <maven.compiler.target>1.8</maven.compiler.target>
19   </properties>
20
21   <dependencies>
22     <dependency>
23       <groupId>junit</groupId>
24       <artifactId>junit</artifactId>
25       <version>4.11</version>
26       <scope>test</scope>
27     </dependency>
28     <!--嵌入式Tomcat -->
29     <dependency>
30       <groupId>org.apache.tomcat.embed</groupId>
31       <artifactId>tomcat-embed-core</artifactId>
32       <version>8.5.34</version>
33     </dependency>
34     <!--嵌入式Tomcat的JSP 支持-->
35     <dependency>
36       <groupId>org.apache.tomcat.embed</groupId>
37       <artifactId>tomcat-embed-jasper</artifactId>
38       <version>8.5.34</version>
39     </dependency>
40
41   </dependencies>
42 </project>
43
```

WebAppDemo.java

```
1 package cn.enjoy.tomcat.embedded;
2
3 import org.apache.catalina.startup.Tomcat;
4
5 /**
6  * author 【享学课堂】 King老师
7 */
8 public class WebAppDemo {
9     public static void main(String[] args) throws Exception {
10         Tomcat tomcat = new Tomcat();
11         tomcat.addWebapp( contextPath: "/ref", docBase: "D:\\work_tomcat\\ref-comet" );
12         tomcat.init();
13         tomcat.start();
14         tomcat.getServer().await();
15     }
16 }
```



服务器推送技术演示

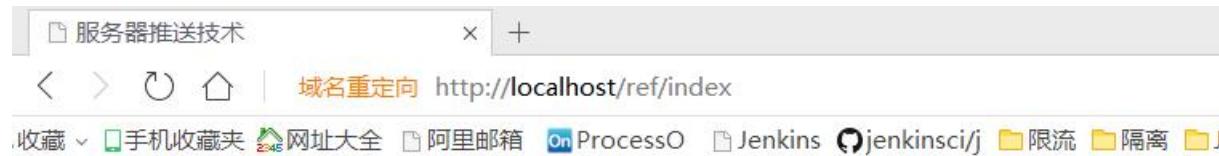
- [查看服务器时间](#)
- [Servlet异步-推送实时新闻](#)
- [SSE-贵金属期货价格实时查询](#)
- [在线支付](#)

修改端口为 80



WebAppDemo.java

```
1 package cn.enjoy.tomcat.embedded;
2
3 import org.apache.catalina.startup.Tomcat;
4
5 /**
6  * Author 【享学课堂】 King老师
7 */
8 public class WebAppDemo {
9     public static void main(String[] args) throws Exception {
10         Tomcat tomcat = new Tomcat();
11         tomcat.addWebapp( contextPath: "/ref", docBase: "D:\\work_tomcat\\ref-comet" );
12         tomcat.getConnector().setPort(80); //指向此处的箭头
13         tomcat.init();
14
15         tomcat.start();
16         tomcat.getServer().await();
17     }
18 }
19
```



服务器推送技术演示

- [查看服务器时间](#)
- [Servlet异步-推送实时新闻](#)
- [SSE-贵金属期货价格实时查询](#)
- [在线支付](#)

看源码可知，Tomcat 通过一个内部类 `DefaultWebXmlListener` 来加载 Web 应用的配置。

```
public Context addWebapp(Host host, String contextPath, String docBase, LifecycleListener config) {
    this.silence(host, contextPath);
    Context ctx = this.createContext(host, contextPath);
    ctx.setPath(contextPath);
    ctx.setDocBase(docBase);
    ctx.addLifecycleListener(this.getDefaultWebXmlListener());
    ctx.setConfigFile(this.getWebappConfigFile(docBase, contextPath));
    ctx.addLifecycleListener(config);
    if (config instanceof ContextConfig) {
        ((ContextConfig)config).setDefaultWebXml(this.noDefaultWebXmlPath());
    }
}
```

嵌入式常用的配置

setConnector:用于设置链接器，包括协议、I/O、端口、压缩、加密等等。

setHost:用于设置 Host

setBaseDir::用于设置临时文件的目录，这些目录用于存放 JSP 生成的源代码及 Class 文件

SpringBoot 中使用嵌入式 Tomcat

没有用过 SpringBoot 的同学可以不看。

```
MySpringBootApplication.java ×
```

```
1 package com.enjoy;
2 import org.springframework.boot.SpringApplication;
3 import org.springframework.boot.autoconfigure.SpringBootApplication;
4
5 @SpringBootApplication
6 public class MySpringBootApplication {
7     //相当于启动了Tomcat,端口默认为8080
8     public static void main(String[] args) {
9         SpringApplication.run(MySpringBootApplication.class, args);
10    }
11 }
12 }
```

MySpringBootApplication.java x SpringApplication.class x

Decompiled .class file, bytecode version: 50.0 (Java 6) Download Sources Choose Sources.

```
735
736     public void setListeners(Collection<? extends ApplicationListener<?>> listeners) {
737         this.listeners = new ArrayList();
738         this.listeners.addAll(listeners);
739     }
740
741     public void addListeners(ApplicationListener... listeners) { this.listeners.addAll(Arrays.a
742
743     public Set<ApplicationListener<?>> getListeners() { return asUnmodifiableOrderedSet(this.li
744
745     public static ConfigurableApplicationContext run(Object source, String... args) {
746         return run(new Object[]{source}, args);
747     }
748
749     @
750     public static ConfigurableApplicationContext run(Object[] sources, String[] args) {
751         return (new SpringApplication(sources)).run(args);
752     }
753
754     public static void main(String[] args) throws Exception {
755
756
757 }
```



lySpringBootApplication.java x SpringApplication.class x

Compiled .class file, bytecode version: 50.0 (Java 6) Download Sources Choose Source

```
    return null;
}

@ public ConfigurableApplicationContext run(String... args) {
    StopWatch stopWatch = new StopWatch();
    stopWatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    this.configureHeadlessProperty();
    SpringApplicationRunListeners listeners = this.getRunListeners(args);
    listeners.started();

    try {
        ApplicationArguments applicationArguments = new DefaultApplicationArguments(args)
        ConfigurableEnvironment environment = this.prepareEnvironment(listeners, applicat
        Banner printedBanner = this.printBanner(environment);
        context = this.createApplicationContext();
        new FailureAnalyzers(context);
        this.prepareContext(context, environment, listeners, applicationArguments, printe
        this.refreshContext(context);      ← tomcat本身就是容器
        this.afterRefresh(context, applicationArguments);
        listeners.finished(context, (Throwable)null);
    } finally {
        stopWatch.stop();
        if (stopWatch.isRunning())
            stopWatch.stop();
    }
}
```



SpringFramework / BOOT / SpringApplication /

MySpringBootApplication.java x SpringApplication.class x

Decompiled .class file, bytecode version: 50.0 (Java 6) Download Sources C

```
103     context.getBeanFactory().registerSingleton(s: "springApplicationArguments",  
104     if (printedBanner != null) {  
105         context.getBeanFactory().registerSingleton(s: "springBootBanner", printedBanner);  
106     }  
107  
108     Set<Object> sources = this.getSources();  
109     Assert.notEmpty(sources, message: "Sources must not be empty");  
110     this.load(context, sources.toArray(new Object[sources.size()]));  
111     listeners.contextLoaded(context);  
112 }  
113  
114     private void refreshContext(ConfigurableApplicationContext context) {  
115         this.refresh(context);  
116         if (this.registerShutdownHook) {  
117             try {  
118                 context.registerShutdownHook();  
119             } catch (AccessControlException var3) {  
120                 ;  
121             }  
122         }  
123     }
```

ngframework > boot > SpringApplication

MySpringBootApplication.java < SpringApplication.class <

Decompiled .class file, bytecode version: 50.0 (Java 6) [Download Sources](#) [Choose Source](#)

```
    if (context instanceof AbstractApplicationContext) {
        return (BeanDefinitionRegistry)((AbstractApplicationContext)context).getBeanFact
    } else {
        throw new IllegalStateException("Could not locate BeanDefinitionRegistry");
    }
}

protected BeanDefinitionLoader createBeanDefinitionLoader(BeanDefinitionRegistry registry) {
    return new BeanDefinitionLoader(registry, sources);
}

@ protected void refresh(ApplicationContext applicationContext) {
    Assert.isInstanceOf(AbstractApplicationContext.class, applicationContext);
    ((AbstractApplicationContext)applicationContext).refresh();
}

protected void afterRefresh(ConfigurableApplicationContext context, ApplicationArguments
```

Decompiled .class file, bytecode version: 50.0 (Java 6)

```
247
248     }
249
250     public Collection<ApplicationListener<?>> getApplicationListeners() { return this.applicationLister...
251
252     public void refresh() throws BeansException, IllegalStateException {
253         Object var1 = this.startupShutdownMonitor;
254         synchronized(this.startupShutdownMonitor) {
255             this.prepareRefresh();
256             ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();
257             this.prepareBeanFactory(beanFactory);
258
259             try {
260                 this.postProcessBeanFactory(beanFactory);
261                 this.invokeBeanFactoryPostProcessors(beanFactory);
262                 this.registerBeanPostProcessors(beanFactory);
263                 this.initMessageSource();
264                 this.initApplicationEventMulticaster();
265                 this.onRefresh();
266             }
267         }
268     }
```

找实现类的方法

Choose Implementation of **AbstractApplicationContext.onRefresh()** (5 found)

- AbstractApplicationContext (org.springframework.context.support)
- AbstractRefreshableWebApplicationContext (org.springframework.web.context.support) Maven: org.springframework:spring-web:4.3.9.RELEASE
- EmbeddedWebApplicationContext (org.springframework.boot.context.embedded) Maven: org.springframework.boot:spring-boot:1.4.3.RELEASE
- GenericWebApplicationContext (org.springframework.web.context.support) Maven: org.springframework:spring-web:4.3.9.RELEASE
- StaticWebApplicationContext (org.springframework.web.context.support) Maven: org.springframework:spring-web:4.3.9.RELEASE



springframework boot context embedded EmbeddedWebApplicationContext

- Application.java X SpringApplication.class X AbstractApplicationContext.class X EmbeddedWebApplicationContext.class X

Decompiled .class file, bytecode version: 50.0 (Java 6) Download Sources Choose Sources...

```
3.3
8.0
51
52     public final void refresh() throws BeansException, IllegalStateException {
53         try {
54             super.refresh();
55         } catch (RuntimeException var2) {
56             this.stopAndReleaseEmbeddedServletContainer();
57             throw var2;
58         }
59     }
60
61     protected void onRefresh() {
62         super.onRefresh();
63
64         try {
65             this.createEmbeddedServletContainer();
66         } catch (Throwable var2) {
67             throw new ApplicationContextException("Unable to start embedded container", var2);
68         }
69     }
70
71     protected void finishRefresh() {
72         super.finishRefresh();
73         EmbeddedServletContainer localContainer = this.startEmbeddedServletContainer();
```

spring-boot-1.4.7.RELEASE.jar org.springframework.boot.context.embedded.EmbeddedWebApplicationContext

Application.java X SpringApplication.class X AbstractApplicationContext.class X EmbeddedWebApplicationContext.class X

Decompiled .class file, bytecode version: 50.0 (Java 6)

```
14     if (localContainer != null) {
15         this.publishEvent(new EmbeddedServletContainerInitializedEvent(applicationContext: this, localContainer))
16     }
17 }
18
19 protected void onClose() {
20     super.onClose();
21     this.stopAndReleaseEmbeddedServletContainer();
22 }
23
24 private void createEmbeddedServletContainer() {
25     EmbeddedServletContainer localContainer = this.embeddedServletContainer;
26     ServletContext localServletContext = this.getServletContext();
27     if (localContainer == null && localServletContext == null) {
28         EmbeddedServletContainerFactory containerFactory = this.getEmbeddedServletContainerFactory();
29         this.embeddedServletContainer = containerFactory.getEmbeddedServletContainer(new ServletContextInitializer...)
```

Choose Implementation of `EmbeddedServletContainerFactory.getEmbeddedServletContainer(ServletContextInitializer...)` (3 found)

- JettyEmbeddedServletContainerFactory (org.springframework.boot.context.embedded.jetty) Maven: org.springframework.boot:spring-boot:1.4.7.RELEASE
- TomcatEmbeddedServletContainerFactory (org.springframework.boot.context.embedded.tomcat) Maven: org.springframework.boot:spring-boot:1.4.7.RELEASE
- UndertowEmbeddedServletContainerFactory (org.springframework.boot.context.embedded.undertow) Maven: org.springframework.boot:spring-boot:1.4.7.RELEASE

Art Build
Bean Validation
Database



```
Recompiled .class file, bytecode version: 50.0 (Java 6) Download Sources Choose Sources...
4j:jul-to-slf4j:1.7.25
public TomcatEmbeddedServletContainerFactory(String contextPath, int port) {
    super(contextPath, port);
    this.uriEncoding = DEFAULT_CHARSET;
}

public EmbeddedServletContainer getEmbeddedServletContainer(ServletContextInitializer... initializers) {
    Tomcat tomcat = new Tomcat();
    File baseDir = this.baseDirectory != null ? this.baseDirectory : this.createTempDir( prefix: "tomcat" );
    tomcat.setBaseDir(baseDir.getAbsolutePath());
    Connector connector = new Connector(this.protocol);
    tomcat.getService().addConnector(connector);
    this.customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    this.configureEngine(tomcat.getEngine());
    Iterator var5 = this.additionalTomcatConnectors.iterator();

    while(var5.hasNext()) {
        Connector additionalConnector = (Connector)var5.next();
        tomcat.getService().addConnector(additionalConnector);
    }

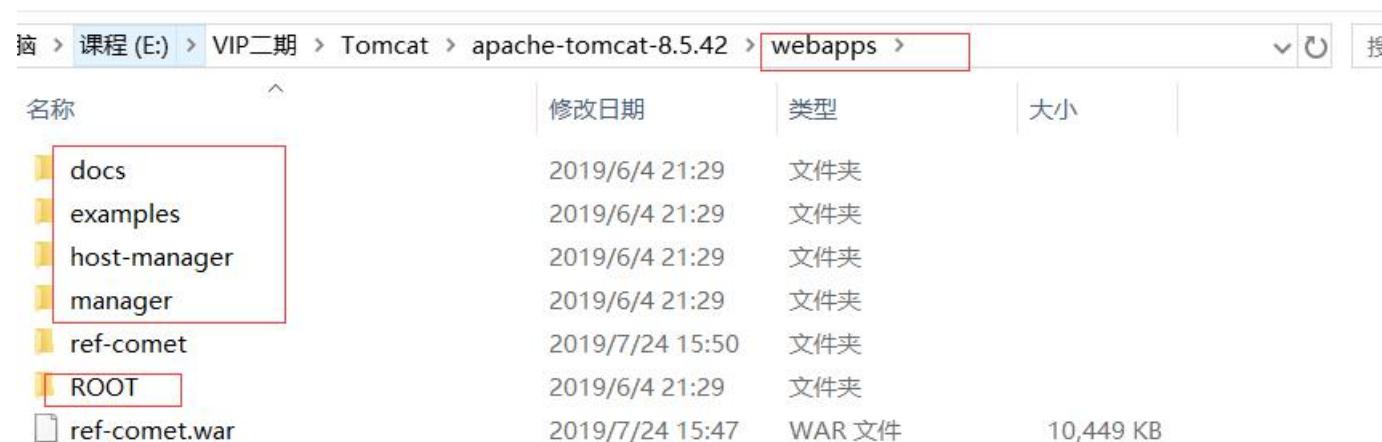
    this.prepareContext(tomcat.getHost(), initializers);
    return this.getTomcatEmbeddedServletContainer(tomcat);
}
```

Tomcat 性能优化

server.xml 优化

其实网络的 Tomcat 配置信息的文章很多，五花八门，King 老师一般推荐使用 Tomcat 自身提供的配置帮助文档，因为你只要下载了 Tomcat，并且启动了它，那么 Tomcat 就会提供最官方，最准确的官方参数说明文档。

下载 Tomcat 后不要删掉默认的程序包。



名称	修改日期	类型	大小
docs	2019/6/4 21:29	文件夹	
examples	2019/6/4 21:29	文件夹	
host-manager	2019/6/4 21:29	文件夹	
manager	2019/6/4 21:29	文件夹	
ref-comet	2019/7/24 15:50	文件夹	
ROOT	2019/6/4 21:29	文件夹	
ref-comet.war	2019/7/24 15:47	WAR 文件	10,449 KB

一般 Tomcat 启动后，不改动端口的话，默认是 8080，我们输入 localhost:8080 访问下。

Apache Tomcat/8.5.42



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](#)

Managing Tomcat

For security, access to the [manager webapp](#) is restricted. Users are defined in:

`$CATALINA_HOME/conf/tomcat-users.xml`

In Tomcat 8.5 access to the manager application is split between different users.
[Read more...](#)

Release Notes

Documentation

[Tomcat 8.5 Documentation](#)

[Tomcat 8.5 Configuration](#)

[Tomcat Wiki](#)

Find additional important configuration information in:

`$CATALINA_HOME/RUNNING.txt`

Getting Help

[FAQ and Mailing Lists](#)

The following mailing lists are available:

[tomcat-announce](#)

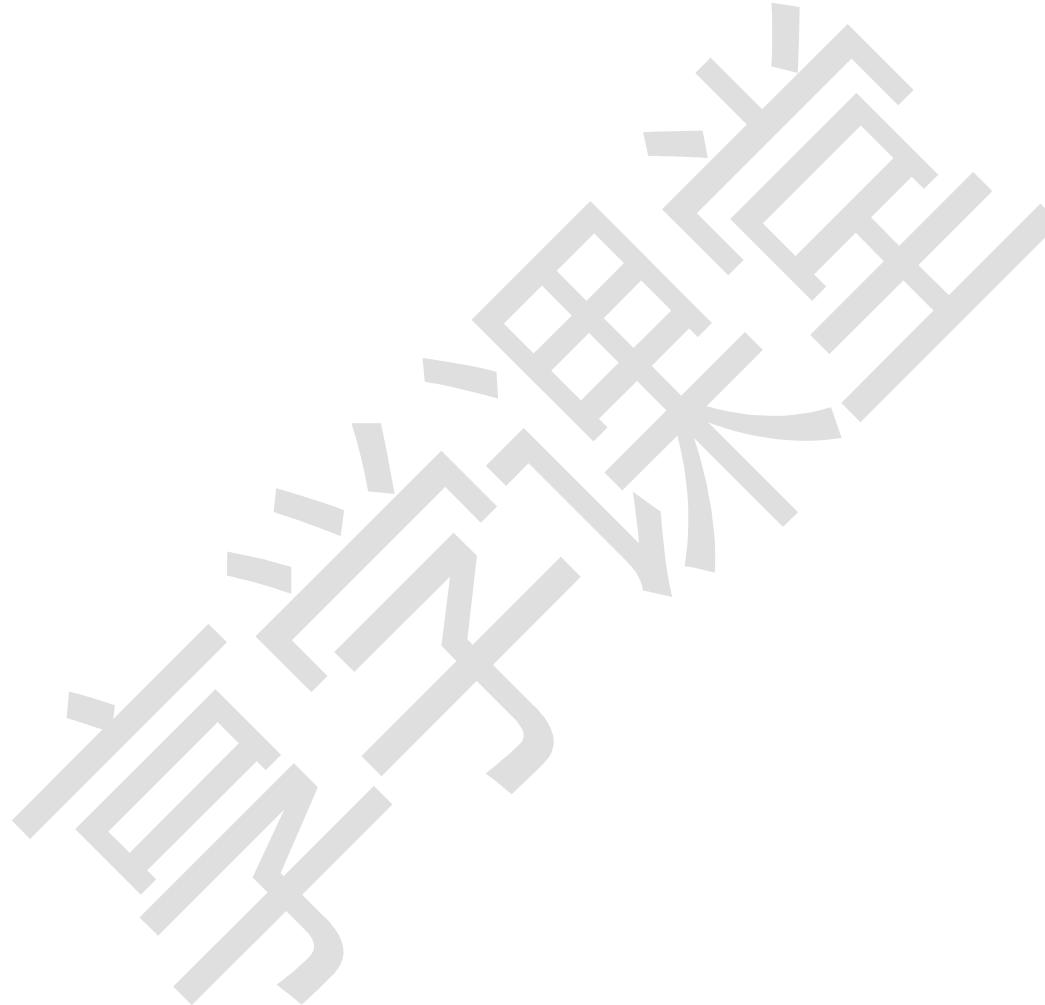
Important announcements, releases, security vulnerability notifications. (Low volume).

[tomcat-users](#)

User support and discussion

[taglibs-user](#)

这个会出现一个英文的 Tomcat 环境界面，包括各种文档说明信息都在此，我推荐使用谷歌浏览器，因为这个浏览器自带翻译功能。



Apache Tomcat / 8.5.42 +

localhost:8080

Java互联网架构师... Data Structure Vis... Google 维基百科, 自由的...

主页 文档 配置 示例 Wiki 邮件列表 查找帮助

Apache Tomcat / 8.5.42

如果你看到这个，你已经成功安装了Tomcat。恭喜！

推荐阅读：

- 安全考虑因素如何
- 经理申请如何
- 集群/会话复制如何

服务器状态

经理App

主持人经理

开发者快速入门

[Tomcat安装程序](#) [领域和AAA](#) [例子](#) [Servlet规范](#)
[第一个Web应用程序](#) [JDBC DataSources](#) [Tomcat版本](#)

最准确的配置信息在此

管理Tomcat

为了安全起见，限制了对**管理器webapp**的访问。用户定义如下：

\$ CATALINA_HOME / conf目录/ tomcat-users.xml中

在Tomcat 8.5中，对管理器应用程序的访问在不同用户之间分配。 [阅读更多...](#)

文档

[Tomcat 8.5文档](#)

Tomcat 8.5配置

[Tomcat Wiki](#)

在以下位置查找其他重要配置信息

获得帮助

常见问题和邮件列表

以下邮件列表可用：

[tomcat-announce](#)
重要公告，发布，安全漏洞通知。（小声）。

[tomcat-users](#)
用户支持和讨论



Connector 连接器





Apache Tomcat 8配置参考

版本8.5.42, 2019年6月4日

链接

Docs Home

配置参考 家

常问问题

用户评论

顶级元素

服务器

服务

执行人

执行者

连接器

HTTP / 1.1

HTTP / 2

AJP

集装箱

下立

HTTP连接器

目录

- [介绍](#)
- [属性](#)
 - 1. [共同属性](#)
 - 2. [标准实施](#)
 - 3. [Java TCP套接字属性](#)
 - 4. [NIO特定配置](#)
 - 5. [NIO2特定配置](#)
 - 6. [APR /本机特定配置](#)
- [嵌套组件](#)
- [特殊功能](#)
 - 1. [HTTP / 1.1和HTTP / 1.0支持](#)
 - 2. [HTTP / 2支持](#)
 - 3. [代理支持](#)
 - 4. [SSL支持](#)
 - 5. [SSL支持 - SSLHostConfig](#)

	port
	<p>此连接器 将在其上创建服务器套接字并等待传入连接的TCP端口号。您的操作系统将只允许一个服务器应用程序侦听特定IP地址上的特定端口号。如果使用特殊值0（零），则Tomcat将随机选择一个空闲端口用于此连接器。这通常仅适用于嵌入式和测试应用程序。</p>

	protocol
	<p>设置协议以处理传入流量。默认值是 <code>HTTP/1.1</code> 使用自动切换机制来选择基于Java NIO的连接器或基于APR / native的连接器。如果 <code>PATH</code> (Windows) 或 <code>LD_LIBRARY_PATH</code> (在大多数unix系统上) 环境变量包含Tomcat本机库，并且 <code>AprLifecycleListener</code> 用于初始化APR的 <code>useAprConnector</code> 属性设置为 <code>true</code>，则将使用APR / 本机连接器。如果找不到本机库或未配置该属性，则将使用基于Java NIO的连接器。请注意，APR / 本机连接器对HTTPS的设置与Java连接器不同。</p> <p>要使用显式协议而不是依赖上述自动切换机制，可以使用以下值：</p> <ul style="list-style-type: none"><code>org.apache.coyote.http11.Http11NioProtocol</code>- 非阻塞Java NIO连接器<code>org.apache.coyote.http11.Http11Nio2Protocol</code>- 非阻塞Java NIO2连接器<code>org.apache.coyote.http11.Http11AprProtocol</code>- APR / 本机连接器。 <p>也可以使用定制实现。</p> <p>看看我们的连接器比较图表。对于http和https，两个Java连接器的配置完全相同。</p> <p>有关APR连接器和APR特定SSL设置的更多信息，请访问APR文档</p>

IO 模型优化策略

连接器模式改为 NIO 模式,NIO 模式最大化压榨了 CPU，把时间片更好利用起来,NIO 适合大量长连接。

maxThreads

此**Connector**要创建的最大请求处理线程数，因此确定可以处理的最大并发请求数。如果未指定，则此属性设置为200.如果执行程序与此连接器关联，则忽略此属性，因为连接器将使用执行程序而不是内部线程池执行任务。请注意，如果配置了执行程序，则会正确记录为此属性设置的任何值，但会报告（例如，通过JMX）-1以明确表示未使用该值。

最大线程优化策略

maxThreads 属性设置为简单 200.这对于单个核心计算机来说很好，但是可以根据生产计算机上的处理器数量[线性](#)增加。在具有四个处理器的计算机上，将此值设置为 800 到 1000 之间的任何值都不会导致问题。如果配置的数量最终远远超过所需的线程数，则当服务器负载较低时，线程池将自然缩减此数字。

压缩 gzip 连接器传输

compressibleMimeType

该值是以逗号分隔的MIME类型列表，可以使用HTTP压缩。默认值为
text/html, text/xml, text/plain, text/css, text/javascript, application/javascript。

compression

所述**连接器**可在试图节省服务器的带宽使用HTTP / 1.1 GZIP压缩。参数的可接受值是“off”（禁用压缩），“on”（允许压缩，导致文本数据被压缩），“force”（在所有情况下强制压缩）或数字整数值（这是等效于“on”，但指定压缩输出之前的最小数据量）。如果内容长度未知且压缩设置为“on”或更具攻击性，则输出也将被压缩。如果未指定，则将此属性设置为“off”。

注意：在使用压缩（节省带宽）和使用sendfile功能（节省CPU周期）之间需要权衡。如果连接器支持sendfile功能，例如NIO连接器，则使用sendfile将优先于压缩。症状将是静态文件大于48 Kb将被解压缩。您可以通过设置useSendfile连接器的属性来关闭sendfile，如下所述，或者更改默认或Web应用程序中[DefaultServlet](#)配置中的sendfile使用率阈值。[conf/web.xml](#)

compressionMinSize

如果**压缩**设置为“on”，则此属性可用于指定压缩输出之前的最小数据量。如果未指定，则此属性默认为“2048”。

客户端和服务器之间的任何主要是文本的通信，无论是HTML，XML还是简单的Unicode，都可以使用简单标准的GZIP算法定期压缩高达90%。这可以对减少网络流量产生巨大影响，允许响应更快地发送回客户端，同时允许更多网络带宽可用于其他网络繁重的应用程序。

```
```xml
<Connector port="80" protocol="HTTP/1.1"
 connectionTimeout="20000"
 redirectPort="8443" executor="tomcatThreadPool" URIEncoding="utf-8"
```

---

```
compression="on"
compressionMinSize="50" noCompressionUserAgents="ozilla, traviata"
compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain" />
```





链接

Docs Home  
配置参考 家常问题  
用户评论

顶级元素

服务器  
服务

执行人

执行者

连接器

HTTP / 1.1  
HTTP / 2  
AJP

集装箱

上下文  
发动机  
主办  
簇

# Apache Tomcat 8配置参考

版本8.5.42, 2019年6月4日



## 执行者 (线程池)

### 目录

- [介绍](#)
- [属性](#)
  1. [共同属性](#)
  2. [标准实施](#)

### 介绍

该**执行程序**表示可组件之间Tomcat中共享的线程池。从历史上看，每个连接器都创建了一个线程池，但这允许您在（主要）连接器之间共享一个线程池，但是当这些连接器配置为支持执行器时，还可以共享其他组件

执行程序必须实现该`org.apache.catalina.Executor`接口。

执行程序是`Service`元素的嵌套元素。并且为了让它被连接器拾取，`Executor`元素必须出现在`server.xml`中的`Connector`元素之前

### 属性

#### 共同属性

**Executor**的所有实现都 支持以下属性：



# Apache Tomcat 8配置参考

版本8.5.42, 2019年6月4日



## 连接

Docs Home  
配置参考家  
常问问题  
用户评论

## 顶级元素

服务器  
服务

## 执行人

执行者

## 连接器

HTTP / 1.1  
HTTP / 2  
AJP

## 集装箱

<Executor

```
name="tomcatThreadPool"<!--线程名称-->
namePrefix="catalina-exec-"
maxThreads="150"<!--最大处理连接数线程-->
minSpareThreads="4" /><!--保留最少线程数--></pre>
```

## 执行者 (线程池)

### 目录

- [介绍](#)
- [属性](#)
  1. [共同属性](#)
  2. [标准实施](#)

### 介绍

该**执行程序**表示可组件之间Tomcat中共享的线程池。从历史上看，每个连接器都创建了一个线程池，但这允许您在（主要）连接器之间共享一个线程池，但是当这些连接器配置为支持执行器时，还可以共享其他组件

执行程序必须实现该org.apache.catalina.Executor接口。

执行程序是Service元素的嵌套元素。并且为了让它被连接器拾取，Executor元素必须出现在server.xml中的Connector元素之前

### 属性

---

```
<!-- 将原有的 Connector 替换为带有线程池的 Connector 如下,其实 servlet.xml 已经有了,只要打开就可以了,将原来的去掉 -->
```

```
<Connector
 executor="tomcatThreadPool"
 port="8080"
 protocol="HTTP/1.1"
 connectionTimeout="20000"
 redirectPort="8443"
 minProcessors="5"<!-- 同时处理请求的最小数 -->
 maxProcessors="75"<!-- 同时处理请求的最大数 -->
 acceptCount="1000" /><!-- 接受最大并发数量 ,超过这个数量就会返回连接被拒绝 -->
```

---

## Executor 的属性



---

默认实现支持以下属性：

属性	描述
threadPriority	(int) 执行程序中线程的线程优先级，默认为 5 (Thread.NORM_PRIORITY常量的值)
daemon	(boolean) 线程是否应该是守护程序线程， 默认为 true
namePrefix	(字符串) 执行程序创建的每个线程的名称前缀。单个线程的线程名称将是namePrefix+threadNumber
maxThreads	(int) 此池中活动线程的最大数量， 默认为 200
minSpareThreads	(int) 最小线程数 (空闲和活动) 始终保持活动状态， 默认为 25
maxIdleTime	(int) 空闲线程关闭之前的毫秒数，除非活动线程数小于或等于minSpareThreads。默认值为60000 (1分钟)
maxQueueSize	(int) 在我们拒绝之前可以排队等待执行的可运行任务的最大数量。默认值是Integer.MAX_VALUE
prestartminSpareThreads	(boolean) 是否应该在启动Executor时启动minSpareThreads， 默认值为 false
threadRenewalDelay	(long) 如果配置了 <a href="#">ThreadLocalLeakPreventionListener</a> ，它将通知此执行程序有关已停止的上下文。上下文停止后，池中的线程将被更新。为避免同时更新所有线程，此选项在任意2个线程的续订之间设置延迟。该值以ms为单位， 默认值为1000ms。如果值为负，则不会续订线程。

## 去除 value 访问 tomcat 记录

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

```

## 关闭自动重载，热部署方式

- > 关闭自动重载， 默认是 true(不同版本中有差异)，自动加载增加运行开销并且很容易内存溢出

← → ⌂

① localhost:8080/docs/config/context.html



Java互联网架构师...



Data Structure Vis...



Google



维基百科, 自由的...

用户评论

## 顶级元素

服务器  
服务

## 执行人

执行者

## 连接器

HTTP / 1.1  
HTTP / 2  
AJP

## 集装箱

上下文  
发动机  
主办  
簇

## 嵌套组件

CookieProcessor

- 介绍

1. [并行部署](#)
2. [命名](#)
3. [定义上下文](#)

- 属性

1. [共同属性](#)
2. [标准实施](#)

- 嵌套组件

- 特殊功能

1. [记录](#)
2. [访问日志](#)
3. [自动上下文配置](#)
4. [上下文参数](#)
5. [环境条目](#)
6. [生命周期听众](#)
7. [请求过滤器](#)
8. [资源定义](#)
9. [资源链接](#)
10. [交易](#)

介绍



## 定义上下文

建议不要将`<Context>`元素直接放在`server.xml`文件中。这是因为它使得修改 `Context`配置更具侵入性，因为在`conf/server.xml`不重新启动 Tomcat的情况下无法重新加载主文件。默认的`Context`元素（见下文）也将覆盖直接放在`server.xml`中的任何`<Context>`元素的配置。为防止这种情况，`override`应将`server.xml`中定义的`<Context>`元素的属性设置为`true`。

可以明确定义单个`Context`元素：

`reloadable`

设置为`true`如果您希望Catalina监视更改类 `/WEB-INF/classes/`和`/WEB-INF/lib`更改，并在检测到更改时自动重新加载Web应用程序。此功能在应用程序开发期间非常有用，但它需要大量的运行时开销，不建议在已部署的生产应用程序上使用。这就是为什么此属性的默认设置为`false`。但是，您可以使用[Manager](#) Web应用程序按需触发已部署应用程序的重新加载。

## web.xml 优化

### 去掉不必要的 servlet

如当前应用是 REST 应用(微服务):

- . 去掉不必要的资源：JspServlet
- . session 也可以移除

---

## JspServlet 优化

- . **checkInterval** - 如果“development”属性为 **false** 且“checkInterval”大于 0，则使用后台编译。“checkInterval”是查看 JSP 页面(包括其附属文件)是否需要重新编译的两次检查时间间隔（单位：秒）。缺省值为 0 秒。
- . **classdebuginfo** - 类文件在编译时是否显示调试(debugging)信息？ **true** 或 **false**, 缺省为 **true**。
- . **classpath** - 编译 servlet 时要使用的类路径，当 ServletContext 属性 org.apache.jasper.Constants.SERVLET\_CLASSPATH 未设置的情况下，该参数才有效。在 Tomcat 中使用到 Jasper 时，该属性总被设置。缺省情况下，该路径基于你当前的 web 应用动态生成。
- . **compiler** – Ant 将要使用的 JSP 页面编译器，请查阅 Ant 文档获取更多信息。如果该参数未设置，那么默认的 Eclipse JDT Java 编译器将被用来代替 Ant。没有缺省值。
- . **compilerSourceVM** - 编译源文件时采用哪一个 JDK 版本？(缺省为 JDK 1.4)
- . **compilerTargetVM** - 运行类文件时采用哪一个 JDK 版本？(缺省为 JDK 1.4)
- . **development** - 是否让 Jasper 用于开发模式？如果是，检查 JSPs 修改的频率，将通过设置 modificationTestInterval 参数来完成。**true** 或 **false**, 缺省为 **true**。
- . **displaySourceFragment** - 异常信息中是否包含出错的源代码片段？**true** 或 **false**, 缺省为 **true**。
- . **dumpSmap** - JSR45 调试的 SMAP 信息是否转存到文件？**true** 或 **false**, 缺省为 **false**。当 suppressSmap 为 **true** 时，该参数为 **false**。
- . **enablePooling** - 确定是否共享标签处理器，**true** 或 **false**, 缺省为 **true**。
- . **engineOptionsClass** - 允许指定的类来配置 Jasper。如果没有指定，则使用默认的 Servlet 内置参数(EmbeddedServletOptions)。
- . **errorOnUseBeanInvalidClassAttribute** - 在一个 useBean action 中，当类属性的值不是一个合法的 bean class 时，Jasper 是否抛出异常？**true** 或 **false**, 缺省为 **true**。
- . **fork** - 是否让 Ant 派生出 JSP 页面多个编译，它们将运行在一个独立于 Tomcat 的 JVM 上。**true** 或者 **false**, 缺省为 **true**.
- . **enStringAsCharArray** - 是否把字符串转换为字符数组？在某些情况下会改善性能。缺省为 **false**。
- . **eClassId** - 当使用标签时，发送给 Internet Explorer 的 class-id 的值。缺省为： 8AD9C840-044E-11D1-B3E9-00805F499D93。
- . **javaEncoding** - 生成 java 源文件时采用的 Java 文件编码。缺省为 UTF-8。
- . **keepgenerated** - 是否保存每个页面生成的 java 源代码，而不删除。**true** 或 **false**, 缺省为 **true**。
- . **mappedfile** - 是否对每个输入行都用一条 print 语句来生成静态内容，以方便调试。**true** 或 **false**, 缺省为 **true**。

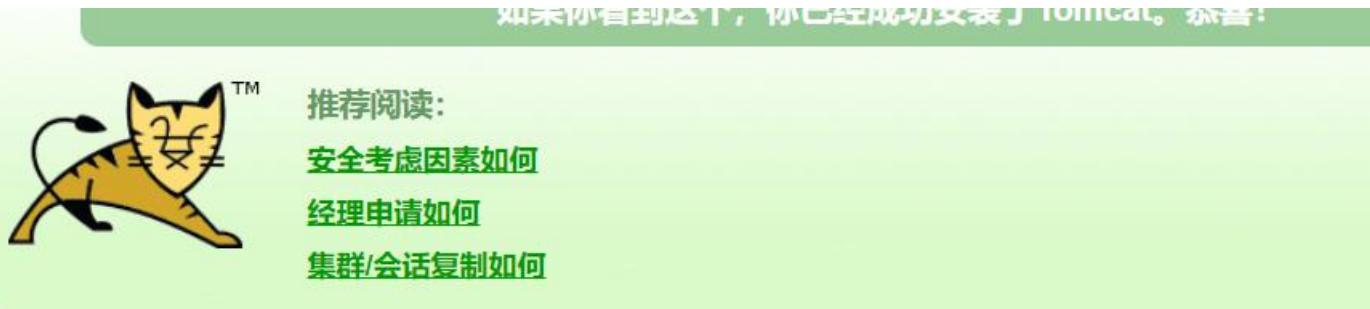
- 
- . **modificationTestInterval** - 检查 JSP 页面修改的间隔时间（单位：秒），在间隔时间内，JSP 及其包含的页面将不会检查。当间隔时间为 0 时，JSP 每一次访问都会被检查。仅仅适用于开发模式（参数 `development` 为 `true`）。缺省为 4 秒。从 JSP 每次开始访问开始计时，N 秒以后检查，变化就编译，每次访问都刷新开始时间，默认 4 秒
  - . **scratchdir** - 当编译 JSP 页面时使用的 `scratch` 目录。缺省为当前 WEB 应用的工作目录。
  - . **suppressSmap** - 是否禁止 JSR45 调试时生成 SMAP 信息？`true` 或 `false`，缺省为 `false`。
  - . **trimSpaces** - 是否去掉模板文本中行为和指令之间的空格。缺省为 `false`。
  - `xpoweredBy` - 确定生成的 `Servlet` 是否加上 `X-Powered-By` 响应头？`true` 或 `false`，缺省为 `false`。

---

**Tomcat 常见问题解决**

**Tomcat 版本升级**





## 开发者快速入门

[Tomcat安装程序](#)

[领域和AAA](#)

[例子](#)

[Serv](#)

[第一个Web应用程序](#)

[JDBC Data Sources](#)

[Tom](#)

### 管理Tomcat

为了安全起见，限制了对[管理器webapp](#)的访问。用户定义如下：

\$ CATALINA\_HOME / conf目录/ tomcat的-users.xml中

在Tomcat 8.5中，对管理器应用程序的访问在不同用户之间分配。[阅读更多...](#)

[发行说明](#)

[更新日志](#)

[迁移指南](#)

[安全通告](#)

### 文档

[Tomcat 8.5文档](#)

[Tomcat 8.5配置](#)

[Tomcat Wiki](#)

在以下位置查找其他重要配置信息

\$ CATALINA\_HOME / RUNNING.txt

开发者可能对以下内容感兴趣：

[Tomcat 8.5 Bug数据库](#)

[Tomcat 8.5 JavaDocs](#)

[Tomcat 8.5 SVN存储库](#)

### 获得帮助

[常见问题和答案](#)

以下邮件列表

[tomcat-announce](#)

重要公告，发布等

[tomcat-users](#)

用户支持和讨论

[taglibs-user](#)

用户支持和讨论

[tomcat-dev](#)

开发邮件列表，讨论等

## 在升级或迁移之前

从一个主要的Apache Tomcat版本更新时，请确保系统上安装的JVM至少支持所需的Java版本。虽然旧版本的Tomcat可能与较新的JVM不兼容，但已知所有当前支持的Apache Tomcat版本（7.0.x, 8.5.x和9.0.x）都可以在Java 8 JVM上正确运行。

从一个主要Tomcat版本迁移到另一个主要版本（例如从Tomcat 7迁移到Tomcat 8，或从Tomcat 8迁移到Tomcat 8.5）时，不应将配置文件从旧版本复制到新版本。建议的方法是从新版本的Apache Tomcat的默认配置开始，并根据需要进行调整。

在同一主要版本中从一个次要版本迁移到另一个次要版本时（例如从Tomcat 7.0.27迁移到Tomcat 7.0.28），您可以保留配置文件，但是您应该检查是否有任何默认值已更改和/或是否已更改添加了任何新元素并相应地调整配置文件。

## 迁移指南

要在9.0.x版本之间进行升级，请参阅“Tomcat 9.0.x迁移指南”的[升级](#)部分。

有关从8.0.x / 8.5.x迁移到9.0.x的信息，请参阅“[Tomcat 9.0.x迁移指南](#)”。

有关在8.5.x版本之间进行升级的信息，请参阅“Tomcat 8.5.x迁移指南”的[升级](#)部分。

有关从8.0.x迁移到8.5.x的信息，请参阅“[Tomcat 8.5.x迁移指南](#)”。

有关8.0.x版本之间的[升级](#)，请参阅“Tomcat 8.0.x迁移指南”的[升级](#)部分。

有关从7.0.x迁移到8.0.x的信息，请参阅“[Tomcat 8.0.x迁移指南](#)”。

有关在7.0.x版本之间进行升级的信息，请参阅“Tomcat 7.0.x迁移指南”的[升级](#)部分。

有关从6.0.x迁移到7.0.x的信息，请参阅“[Tomcat 7.0.x迁移指南](#)”。

有关在6.0.x版本之间进行升级的信息，请参阅“Tomcat 6.0.x迁移指南”的[升级](#)部分。

有关从5.5.x迁移到6.0.x的信息，请参阅“[Tomcat 6.0.x迁移指南](#)”。

---

# Tomcat 常见面试题

## 1.Tomcat 有哪几种部署方式？

### 隐式部署

直接丢文件夹、war、jar 到 webapps 目录，tomcat 会根据文件夹名称自动生成虚拟路径，简单，但是需要重启 Tomcat 服务器，包括要修改端口和访问路径的也需要重启。

### 显式部署

#### 添加 context 元素

server.xml 中的 Host 加入一个 Context（指定路径和文件地址），例如：

```
<Host name="localhost">
 <Context path="/comet" docBase="D:\work_tomcat\ref-comet.war" />
```

即/comet 这个虚拟路径映射到了 D:\work\_tomcat\ref-comet 目录下(war 会解压成文件)，修改完 server.xml 需要重启 tomcat 服务器。

#### 创建 xml 文件

在 conf/Catalina/localhost 中创建 xml 文件，访问路径为文件名，例如：

在 localhost 目录下新建 demo.xml，内容为：

```
<Context docBase="D:\work_tomcat\ref-comet" />
```

不需要写 path，虚拟目录就是文件名 demo，path 默认为/demo，添加 demo.xml 不需要重启 tomcat 服务器。

### 三种方式比较：

隐式部署：可以很快部署，需要人手动移动 Web 应用到 webapps 下，在实际操作中不是很人性化

添加 context 元素：配置速度快，需要配置两个路径，如果 path 为空字符串，则为缺省配置，每次修改 server.xml 文件后都要重新启动 Tomcat 服务器，重新部署。

---

创建 xml 文件:服务器后台会自动部署, 修改一次后台部署一次, 不用重复启动 Tomcat 服务器, 该方式显得更为智能化。

## 2. Tomcat 核心组件是哪些?

Tomcat 的核心组件是连接器(connector)和容器(Container),

连接器(connector)封装了底层的网络请求(Socket 请求及相应处理), 提供了统一的接口.

容器(Container)则专注处理 Servlet, Tomcat 本质上就是 Servlet 容器。

## 3. 什么是 Tomcat 的 Valve

在一个大的组件中直接处理这些繁杂的逻辑处理, 使用管道 (pipeline) 可以把多个对象连接起来, 而 Valve(阀门) 整体看起来就像若干个阀门嵌套在管道中, 而处理逻辑放在阀门上。

管道(Pipeline)就像一个工厂中的生产线, 负责调配工人 (valve) 的位置, valve 则是生产线上负责不同操作的工人。

## 4. Tomcat 有哪几种 Connector 运行模式(优化)?

1. bio(blocking I/O) 同步阻塞 I/O (tomcat8.5 版本已经移除)
2. nio(non-blocking I/O) 同步非阻塞 I/O
3. Nio2/AIO 异步非阻塞 I/O
4. apr(Apache Portable Runtime/Apache 可移植运行库)

对于 I/O 选择, 要根据业务场景来定, 一般高并发场景下, APR 和 NIO2 的性能要优于 NIO 和 BIO, (linux 操作系统支持的 NIO2 由于是一个假的, 并没有真正实现 AIO, 所以一般 linux 上推荐使用 NIO, 如果是 APR 的话, 需要安装 APR 库, 而 Windows 上默认安装了), 所以在 8.5 的版本中默认是 NIO。

---

## 5. tomcat 容器是如何创建 servlet 类实例？用到了什么原理？

当容器启动时，会读取在 webapps 目录下所有的 web 应用中的 web.xml 文件，然后对 xml 文件进行解析，并读取 servlet 注册信息。然后，将每个应用中注册的 servlet 类都进行加载，并通过反射的方式实例化（也有可能是在第一次请求时实例化）。

## 6. tomcat 如何优化？

## 7. tomcat 中 JVM 如何调优

一般我们优化启动时的堆内存设置,Windows 下,在文件{tomcat\_home}/bin/catalina.bat, Unix 下, 在文件\$CATALINA\_HOME/bin/catalina.sh 的前面, 增加如下设置:

```
JAVA_OPTS=" \"$JAVA_OPTS\" -Xms[初始化内存大小] -Xmx[可以使用的最大内存]
```

一般说来，你应该使用物理内存的 80% 作为堆大小。建议设置为 70%；建议设置[[初始化内存大小]]等于[可以使用的最大内存]，这样可以减少平凡分配堆而降低性能。

## 8. 说出 Tomcat 中用到的任意一种设计模式？

## 9. Tomcat 中类加载的顺序

当应用需要到某个类时，则会按照下面的顺序进行类加载：

- 1 使用 bootstrap 引导类加载器加载
- 2 使用 system 系统类加载器加载
- 3 使用应用类加载器在 WEB-INF/classes 中加载

- 
- 4 使用应用类加载器在 WEB-INF/lib 中加载
  - 5 使用 common 类加载器在 CATALINA\_HOME/lib 中加载

## 10. 什么是双亲委派模型？

**定义：**双亲委派模型的工作过程为：如果一个类加载器收到了类请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父加载器去完成，每一层都是如此，因此所有类加载的请求都会传到启动类加载器，只有当父加载器无法完成该请求时，子加载器才去自己加载。

**实现方式：**该模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器不是以继承的关系来实现，而是通过组合关系来复用父加载器的代码。

**意义：**好处双亲委派模型的好处就是 java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如：Object，无论那个类加载器去加载该类，最终都是由启动类加载器进行加载的，因此 Object 类在程序的各种类加载环境中都是一个类。如果不用改模型，那么 java.lang.Object 类存放在 classpath 中，那么系统中就会出现多个 Object 类，程序变得很混乱。

## 11. 既然 Tomcat 不遵循双亲委派机制，那么如果我自己定义一个恶意的 HashMap，会不会有风险呢？

不会有风险，如果有，Tomcat 都运行这么多年了，那群 Tomcat 大神能不改进吗？tomcat 不遵循双亲委派机制，只是自定义的 webAppClassLoader 不遵循，但上层的类加载器还是遵守双亲委派的，

# 第五节 手写 Tomcat

## 什么是 Tomcat

Tomcat 本质上是一款开源轻量级 Web 应用服务器，是一款优秀的 Servlet 容器实现。

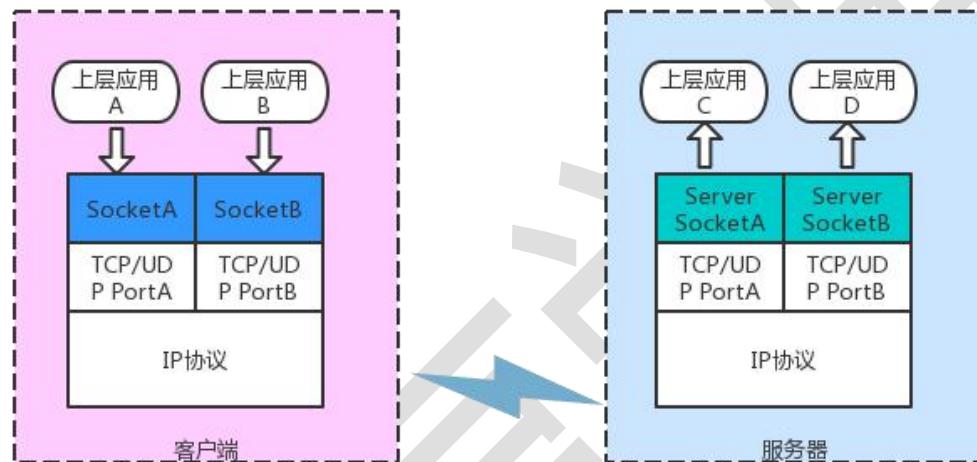
核心 2 点，Web 服务器、Servlet 容器。

阿里中间件团队：<http://jm.taobao.org/about/>

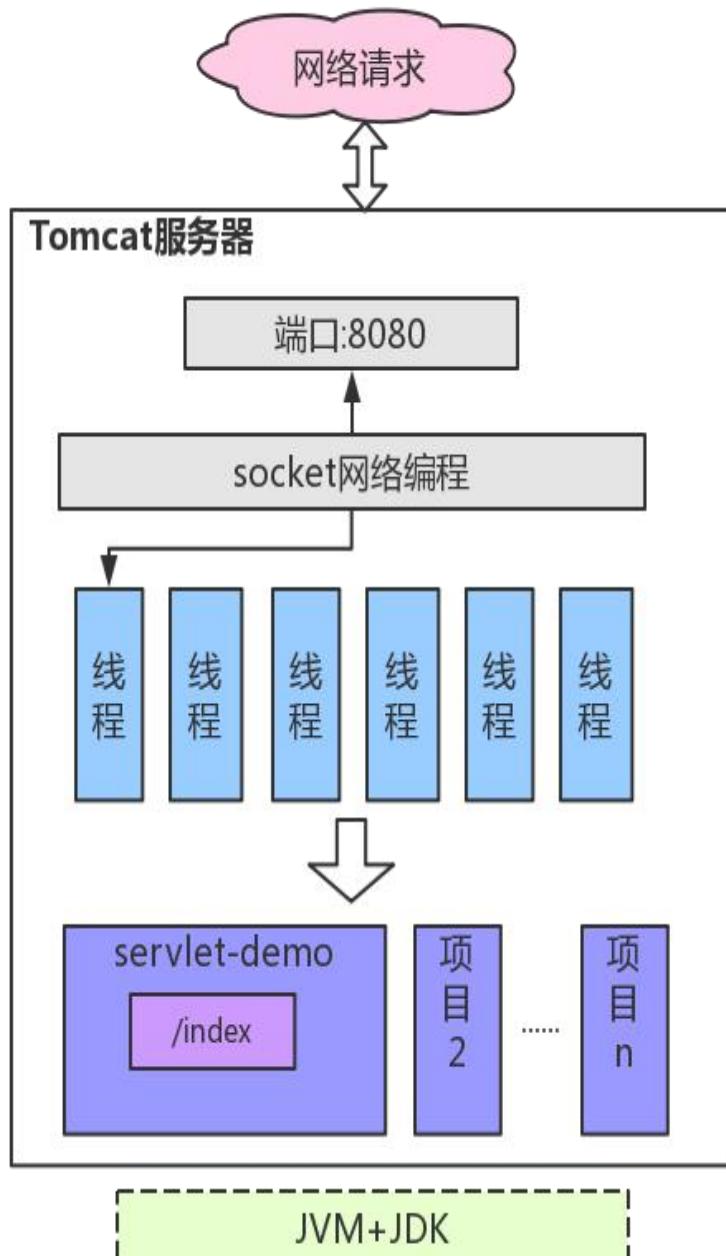
很多公司都手写或者重写 Tomcat. 比如 aliTomcat：[https://help.aliyun.com/document\\_detail/90754.html](https://help.aliyun.com/document_detail/90754.html)

## 网络基础知识

网络通讯，客户端和服务端的关系，java 中建立 Socket 通讯，注意一般 Socket 是基于 TCP 的协议。



还原 Tomcat 本质



---

手写第一步：完成简单的一个 BIO 的通讯



```
*于与Tomcat
*/
public class TomcatServer {
 //JDK提供的连接池
 private static ExecutorService executorService = Executors.newCachedThreadPool();
 //所有的手写 都写在一个方法中
 public static void main(String[] args) throws Exception{
 //1、绑定端口 ---JDK 为我们提供的网络操作的API
 ServerSocket serverSocket = new ServerSocket(port: 8080);
 System.out.println("服务器启动成功!");
 while(!serverSocket.isClosed()){//在等待请求
 //2、有请求过来了 ,使用一个socket: 代表着一个客户端--服务端的连接
 Socket socket = serverSocket.accept();
 //3、使用一个线程来处理请求
 executorService.execute(()->{
 try{
 InputStream inputStream = socket.getInputStream();
 System.out.println("收到请求: ");
 BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
 //字符串
 String msg =null;
 StringBuilder requestInfo = new StringBuilder();
 while ((msg=reader.readLine()) !=null){
 if(msg.length()==0) break;
 requestInfo.append(msg).append("\r\n");
 }
 System.out.println(requestInfo);
 }
 });
 }
 }
}
```

第

## Tomcat 如何和浏览器互动、

浏览器之间通讯必须要符合 HTTP 协议

响应信息加入 HTTP 格式的响应

```
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
```

```
 " + msg + "\r\n");
 requestInfo.append(msg).append("\r\n");
}

System.out.println(requestInfo);

OutputStream outputStream = socket.getOutputStream();
byte[] response ="请求成功!".getBytes();
outputStream.write("HTTP/1.1 200 OK \r\n".getBytes());
outputStream.write("Content-Type:text/html;charset=utf-8 \r\n".getBytes());
outputStream.write(("Content-Length:"+response.length+"\r\n").getBytes());
outputStream.write("\r\n".getBytes());
outputStream.write(response);
outputStream.flush();
System.out.println("----END");
```

# 让代码在 Tomcat 上跑

## 类加载

```
//所有的手写 都写在一个方法中
public static void main(String[] args) throws Exception{
 //0、加载项目（类加载）、容器、HashMap
 Map<String, Object> webAppMap = ProjectLoader.load();

 //1. 绑定端口 --- JDK 为我们提供的网络操作的 API
}

/*
 *
 */
public class ProjectLoader {
 public static Map load() throws Exception{
 //定义一个Map,存储项目信息
 Map<String, Object> webapp = new HashMap<>();

 //servlet
 Map<String, Servlet> servletInstances = new HashMap<>();
 //servlet-mapping
 Map<String, String> servletMapping = new HashMap<>();

 //项目路径
 String webappPath ="D:\\work_public\\servletdemo\\out\\artifacts\\servletdemo_war_exploded\\";
 //JDK提供类加载器 URL
 URL classFile = new URL(spec: "file:"+webappPath+"\\classes\\");
 URLClassLoader urlClassLoader = new URLClassLoader(new URL[]{classFile});
 }
}
```

---

```
//解析xml文件
//1.获取解析器
SAXReader reader = new SAXReader();
//2.获取文档对象(xml)
Document document = reader.read(new File(pathname: webappPath+"\\"+web.xml));
//3.获取根元素
Element root = document.getRootElement();
//4.获取根元素中的子元素
List<Element> childElements = root.elements();
//5.遍历子元素
for (Element element:childElements) {
 //6.判断元素名称为servlet的元素
 if ("servlet".equals(element.getName())) {
 //获取servlet-name元素
 Element servletName = element.element("servlet-name");
 //获取servlet-class元素
 Element servletClass = element.element("servlet-class");
 String strServletName = servletName.getText();
 String strServletClass = servletClass.getText();
 System.out.println("servlet:"+strServletName+"="+strServletClass);
 //1.加载到JVM
 Class<?> classzz= urlClassLoader.loadClass(strServletClass);
 //2 创建对象实例(反射) -servlet;
 Servlet servlet = (Servlet)classzz.newInstance();
 //web.xml的servlet实例化放入hashmap
 servletInstances.put(strServletName, servlet);
 }
 //7.判断元素名称为servlet-mapping的元素
 if ("servlet-mapping".equals(element.getName())) {
 //获取servlet-name元素
 Element servletName = element.element("servlet-name");
 //获取url-pattern元素
 Element urlPattern = element.element("url-pattern");
```

## 解析请求地址，找到对应的 Servlet

```
//要对请求的头进行解析
//获取请求的第一行
String firstline =requestInfo.toString().split(regex: "\r\n") [0];
System.out.println(firstline);
//防止浏览器发空过的请求过来
if(firstline.length()<1){
 return;
}
String servletPath = firstline.split(regex: " ") [1];

//Servlet映射
Map<String, String> servletMapping= (Map<String, String>) webAppMap.get("servlet-mapping");
//Servlet实例
Map<String, Object> servletInstances= (Map<String, Object>) webAppMap.get("servlet");
```

## 构造参数，执行 Servlet 的 service 方法

```
Map<String, Object> servletInstances = (Map<String, Object>) webAppMap.get("servlet");
//是否有正确的url对应的Servlet
if(servletMapping.containsKey(servletpath)){
 String servletname = servletpath.get(servletpath);
 HttpServlet httpServlet = (HttpServlet)servletInstances.get(servletname);
 HttpServletRequest httpServletRequest = createRequest();
 HttpServletResponse httpServletResponse = createResponse();

 httpServlet.service(httpServletRequest, httpServletResponse);

 OutputStream outputStream = socket.getOutputStream();
 byte[] response = "请求成功!".getBytes();
 outputStream.write("HTTP/1.1 200 OK \r\n".getBytes());
 outputStream.write("Content-Type:text/html;charset=utf-8 \r\n".getBytes());
 outputStream.write(("Content-Length:"+response.length+"\r\n").getBytes());
 outputStream.write("\r\n".getBytes());
 outputStream.write(response);
 outputStream.flush();
 System.out.println("----END");
}
```

构建简单的参数

```
}
private static HttpServletRequest createRequest(){
 return new HttpServletRequest() {...};

private static HttpServletResponse createResponse(){...}
```