

HTTP&Tomcat&Servlet

今日目标：

- 了解JavaWeb开发的技术栈
- 理解HTTP协议和HTTP请求与响应数据的格式
- 掌握Tomcat的使用
- 掌握在IDEA中使用Tomcat插件
- 理解Servlet的执行流程和生命周期
- 掌握Servlet的使用和相关配置

1. Web概述

1.1 Web和JavaWeb的概念

Web是全球广域网，也称为万维网（www），能够通过浏览器访问的网站。在我们日常的生活中，经常会使用浏览器去访问百度、京东、传智官网等这些网站，这些网站统称为Web网站。如下就是通过浏览器访问传智官网的界面：



我们知道了什么是Web，那么JavaWeb又是什么呢？顾名思义JavaWeb就是用Java技术来解决相关web互联网领域的技术栈。等学习完JavaWeb之后，同学们就可以使用Java语言开发我们上述所说的网站。而国内很多大型网站公司也是首选Java语言来解决web互联网相关的问题。那都有哪些公司的系统是使用Java语言的呢？



语言开发互联网系统是有很多技术栈需要大家了解，具体都有哪些呢？

1.2 JavaWeb技术栈

了解JavaWeb技术栈之前，有一个很重要的概念要介绍。

1.2.1 B/S架构

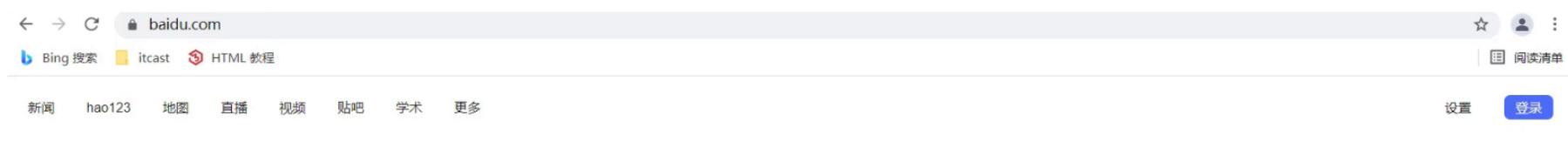
什么是B/S架构？B/S 架构：Browser/Server，浏览器/服务器 架构模式，它的特点是，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web资源，服务器把Web资源发送给浏览器即可。大家可以通过下面这张图来回想下我们平常的上网过程：



- 打开浏览器访问百度首页，输入要搜索的内容，点击回车或百度一下，就可以获取和搜索相关的内容
- 思考下搜索的内容并不在我们自己的点上，那么这些内容从何而来？答案很明显是从百度服务器返回给我们的
- 日常百度的小细节，逢年过节百度的logo会更换不同的图片，服务端发生变化，客户端不需做任务事情就能获取最新内容
- 所以说B/S架构的好处：易于维护升级：服务器端升级后，客户端无需任何部署就可以使用到新的版本。了解了什么是B/S架构后，作为后台开发工程师的我们将来主要关注的是服务端的开发和维护工作。在服务端将来会放很多资源，都有哪些资源呢？

1.2.2 静态资源

- 静态资源主要包含HTML、CSS、JavaScript、图片等，主要负责页面的展示。
- 我们之前已经学过前端网页制作三剑客（HTML+CSS+JavaScript），使用这些技术我们就可以制作出效果比较丰富的网页，将来展现给用户。但是由于做出来的这些内容都是静态的，这就会导致所有的人看到的内容将是一模一样。
- 在日常上网的过程中，我们除了看到这些好看的页面以外，还会碰到很多动态内容，比如我们常见的百度登录效果：



百度一下

张三登录以后在网页的右上角看到的是 张三，而李四登录以后看到的则是李四。所以不同的用户访问相同的资源看到的内容大多数是不一样的，要想实现这样的效果，光靠静态资源是无法实现的。

1.2.3 动态资源

- 动态资源主要包含Servlet、JSP等，主要用来负责逻辑处理。
- 动态资源处理完逻辑后会把得到的结果交给静态资源来进行展示，动态资源和静态资源要结合一起使用。
- 动态资源虽然可以处理逻辑，但是当用户来登录百度的时候，就需要输入用户名和密码，这个时候我们就又需要解决的一个问题是，用户在注册的时候填入的用户名和密码、以及我们经常会访问到一些数据列表的内容展示（如下图所示），这些数据都存储在哪里？我们需要的时候又是从哪里来取呢？

1.2.4 数据库

- 数据库主要负责存储数据。

- 整个Web的访问过程就如下图所示：



(1) 浏览器发送一个请求到服务端，去请求所需要的相关资源；(2) 资源分为动态资源和静态资源，动态资源可以是使用Java代码按照Servlet和JSP的规范编写的内容；(3) 在Java代码可以进行业务处理也可以从数据库中读取数据；(4) 拿到数据后，把数据交给HTML页面进行展示，再结合CSS和JavaScript使展示效果更好；(5) 服务端将静态资源响应给浏览器；(6) 浏览器将这些资源进行解析；(7) 解析后将效果展示在浏览器，用户就可以看到最终的结果。在整个Web的访问过程中，会设计到很多技术，这些技术有已经学习过的，也有还未涉及到的内容，都有哪些还没有涉及到呢？

1.2.5 HTTP协议

- HTTP协议：主要定义通信规则
- 浏览器发送请求给服务器，服务器响应数据给浏览器，这整个过程都需要遵守一定的规则，之前大家学习过TCP、UDP，这些都属于规则，这里我们需要使用的是HTTP协议，这也是一种规则。

1.2.6 Web服务器

- Web服务器：负责解析HTTP协议，解析请求数据，并发送响应数据
- 浏览器按照HTTP协议发送请求和数据，后台就需要一个Web服务器软件来根据HTTP协议解析请求和数据，然后把处理结果再按照HTTP协议发送给浏览器
- Web服务器软件有很多，我们课程中将学习的是目前最为常用的Tomcat服务器

到目前为止，关于JavaWeb中用到的技术栈我们就介绍完了，这里面就只有HTTP协议、Servlet、JSP以及Tomcat这些知识是没有学习过的，所以整个Web核心主要就是来学习这些技术。

1.3 Web核心课程安排



整个Web核心，我们总共有六天的学习内容，分别是：

- 第一天：HTTP、Tomcat、Servlet
- 第二天：Request（请求）、Response（响应）
- 第三天：JSP、会话技术（Cookie、Session）
- 第四天：Filter（过滤器）、Listener（监听器）
- 第五天：Ajax、Vue、ElementUI
- 第六天：综合案例

(1) Request是从客户端向服务端发出的请求对象，

(2) Response是从服务端响应给客户端的结果对象，

(3) JSP是动态网页技术，

(4) 会话技术是用来存储客户端和服务端交互所产生的数据，

(5) 过滤器是用来拦截客户端的请求，

(6) 监听器是用来监听特定事件，

(7) Ajax、Vue、ElementUI都是属于前端技术

这些技术都该如何来使用，我们后面会一个个进行详细的讲解。接下来我们来学习下HTTP、Tomcat和Servlet。

2, HTTP

2.1 简介

HTTP概念

HyperText Transfer Protocol, 超文本传输协议, 规定了浏览器和服务器之间**数据传输的规则**。

- 数据传输的规则指的是请求数据和响应数据需要按照指定的格式进行传输。
- 如果想知道具体的格式, 可以打开浏览器, 点击 F12 打开开发者工具, 点击 Network 来查看某一次请求的请求数据和响应数据具体的格式内容, 如下图所示:

注意:在浏览器中如果看不到上述内容, 需要清除浏览器的浏览数据。chrome浏览器可以使用`ctrl+shift+Del`进行清除。

所以学习HTTP主要就是学习请求和响应数据的具体格式内容。

HTTP协议特点

HTTP协议有它自己的一些特点, 分别是:

- 基于TCP协议: 面向连接, 安全
TCP是一种面向连接的(建立连接之前是需要经过三次握手)、可靠的、基于字节流的传输层通信协议, 在数据传输方面更安全。
- 基于请求-响应模型的: 一次请求对应一次响应
请求和响应是一一对应关系
- HTTP协议是无状态协议: 对于事物处理没有记忆能力。每次请求-响应都是独立的
无状态指的是客户端发送HTTP请求给服务端之后, 服务端根据请求响应数据, 响应完后, 不会记录任何信息。这种特性有优点也有缺点,
 - 缺点: 多次请求间不能共享数据
 - 优点: 速度快

请求之间无法共享数据会引发的问题, 如:

- 京东购物, 加入购物车 和去购物车结算 是两次请求,
- HTTP协议的无状态特性, 加入购物车请求响应结束后, 并未记录加入购物车是何商品
- 发起去购物车结算的请求后, 因为无法获取哪些商品加入了购物车, 会导致此次请求无法正确展示数据

具体使用的时候, 我们发现京东是可以正常展示数据的, 原因是Java早已考虑到这个问题, 并提出了使用会话技术(cookie, session)来解决这个问题。具体如何来做, 我们后面会详细讲到。刚才提到HTTP协议是规定了请求和响应数据的格式, 那具体的格式是什么呢?

2.2 请求数据格式

2.2.1 格式介绍

请求数据总共分为三部分内容, 分别是**请求行**、**请求头**、**请求体**

```
GET / HTTP/1.1
Host: www.itcast.cn
Connection: keep-alive
Cache-Control: max-age=0 Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 Chrome/91.0.4472.106
...
```

- 请求行: HTTP请求中的第一行数据, 请求行包含三块内容, 分别是 GET[请求方式] / [请求URL路径] HTTP/1.1[HTTP协议及版本]
请求方式有七种, 最常用的是GET和POST
- 请求头: 第二行开始, 格式为key: value形式
请求头中会包含若干个属性, 常见的HTTP请求头有:

Host: 表示请求的主机名
User-Agent: 浏览器版本, 例如Chrome浏览器的标识类似Mozilla/5.0 ... Chrome/79, IE浏览器的标识类似Mozilla/5.0 (Windows NT ...) like Gecko;
Accept: 表示浏览器能接收的资源类型, 如text/*, image/*或者/*表示所有;
Accept-Language: 表示浏览器偏好的语言, 服务器可以据此返回不同语言的网页;
Accept-Encoding: 表示浏览器可以支持的压缩类型, 例如gzip, deflate等。

这些数据有什么用处?

举例说明: 服务端可以根据请求头中的内容来获取客户端的相关信息, 有了这些信息服务端就可以处理不同的业务需求, 比如:

- 不同浏览器解析HTML和CSS标签的结果会有不一致, 所以就会导致相同的代码在不同的浏览器会出现不同的效果
- 服务端根据客户端请求头中的数据获取到客户端的浏览器类型, 就可以根据不同的浏览器设置不同的代码来达到一致的效果
- 这就是我们常说的浏览器兼容问题

- 请求体: POST请求的最后一部分, 存储请求参数

```
POST / HTTP/1.1
Host: www.itcast.cn
Connection: keep-alive
Cache-Control: max-age=0 Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 Chrome/91.0.4472.106

username=superbaby&password=123456
```

如上图红线框的内容就是请求体的内容, 请求体和请求头之间是有一个空行隔开。此时浏览器发送的是POST请求, 为什么不能使用GET呢? 这时就需要回顾GET和POST两个请求之间的区别了:

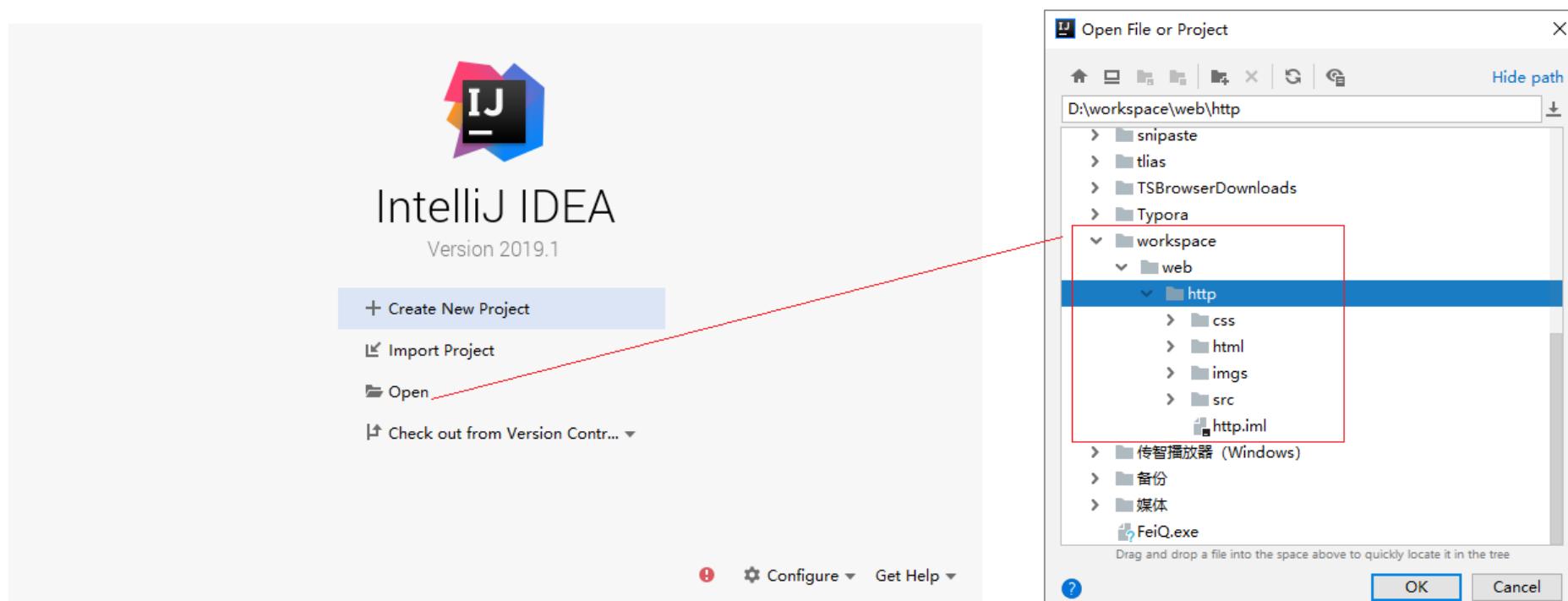
- GET请求请求参数在请求行中, 没有请求体, POST请求请求参数在请求体中
- GET请求请求参数大小有限制, POST没有

2.2.2 实例演示

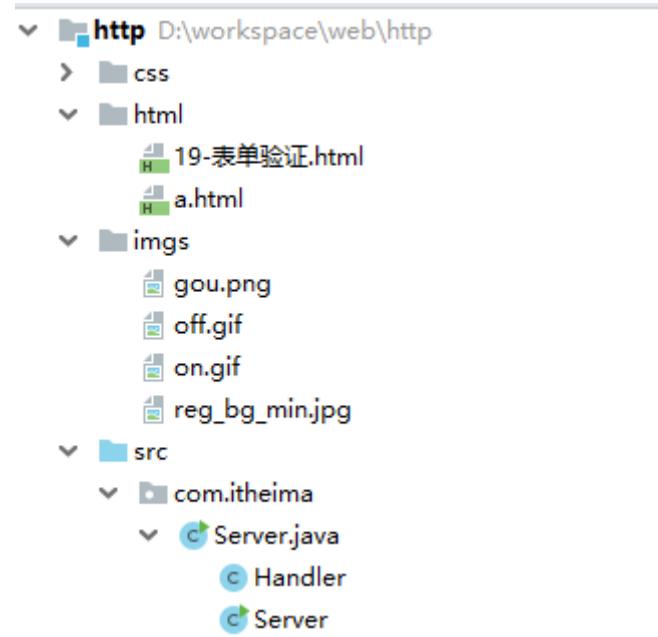
把 代码\http 拷贝到IDEA的工作目录中, 比如 D:\workspace\web 目录,



使用IDEA打开



打开后, 可以点击项目中的 html\19-表单验证.html, 使用浏览器打开, 通过修改页面中form表单的method属性来测试GET请求和POST请求的参数携带方式。



小结：

1. 请求数据中包含三部分内容，分别是请求行、请求头和请求体
2. POST请求数据在请求体中，GET请求数据在请求行上

2.3 响应数据格式

2.3.1 格式介绍

响应数据总共分为三部分内容，分别是**响应行**、**响应头**、**响应体**

```

HTTP/1.1 200 OK
Server: Tengine
Content-Type: text/html
Transfer-Encoding: chunked...

<html>
<head>
    <title></title>
</head>
<body></body>
</html>

```

- 响应行：响应数据的第一行，响应行包含三块内容，分别是 HTTP/1.1 [HTTP协议及版本] 200 [响应状态码] ok [状态码的描述]
- 响应头：第二行开始，格式为key: value形式

响应头中会包含若干个属性，常见的HTTP响应头有：

```

Content-Type: 表示该响应内容的类型，例如text/html, image/jpeg;
Content-Length: 表示该响应内容的长度（字节数）;
Content-Encoding: 表示该响应压缩算法，例如gzip;
Cache-Control: 指示客户端如何缓存，例如max-age=300表示可以最多缓存300秒

```

- 响应体：最后一部分。存放响应数据

上图中...这部分内容就是响应体，它和响应头之间有一个空行隔开。

2.3.2 响应状态码

参考：资料/1.HTTP/《响应状态码.md》

关于响应状态码，我们先主要认识三个状态码，其余的等后期用到了再去掌握：

- 200 ok 客户端请求成功
- 404 Not Found 请求资源不存在
- 500 Internal Server Error 服务端发生不可预期的错误

2.3.3 自定义服务器

在前面我们导入到IDEA中的http项目中，有一个Server.java类，这里面就是自定义的一个服务器代码，主要使用到的是ServerSocket 和 Socket

```

package com.itheima;

import sun.misc.IOUtils;

```

```

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
/*
    自定义服务器
*/
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(8080); // 监听指定端口
        System.out.println("server is running...");
        while (true){
            Socket sock = ss.accept();
            System.out.println("connected from " + sock.getRemoteSocketAddress());
            Thread t = new Handler(sock);
            t.start();
        }
    }
}

class Handler extends Thread {
    Socket sock;

    public Handler(Socket sock) {
        this.sock = sock;
    }

    public void run() {
        try (InputStream input = this.sock.getInputStream()) {
            try (OutputStream output = this.sock.getOutputStream()) {
                handle(input, output);
            }
        } catch (Exception e) {
            try {
                this.sock.close();
            } catch (IOException ioe) {
            }
            System.out.println("client disconnected.");
        }
    }
}

private void handle(InputStream input, OutputStream output) throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(input, StandardCharsets.UTF_8));
    BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
    // 读取HTTP请求:
    boolean requestOk = false;
    String first = reader.readLine();
    if (first.startsWith("GET / HTTP/1.")) {
        requestOk = true;
    }
    for (;;) {
        String header = reader.readLine();
        if (header.isEmpty()) { // 读取到空行时, HTTP Header读取完毕
            break;
        }
        System.out.println(header);
    }
    System.out.println(requestOk ? "Response OK" : "Response Error");
    if (!requestOk) {
        // 发送错误响应:
        writer.write("HTTP/1.0 404 Not Found\r\n");
        writer.write("Content-Length: 0\r\n");
        writer.write("\r\n");
        writer.flush();
    } else {
        // 发送成功响应:

        // 读取html文件, 转换为字符串
        BufferedReader br = new BufferedReader(new FileReader("http/html/a.html"));
        StringBuilder data = new StringBuilder();
        String line = null;
        while ((line = br.readLine()) != null){
            data.append(line);
        }
        br.close();
        int length = data.toString().getBytes(StandardCharsets.UTF_8).length;

        writer.write("HTTP/1.1 200 OK\r\n");
    }
}

```

```

        writer.write("Connection: keep-alive\r\n");
        writer.write("Content-Type: text/html\r\n");
        writer.write("Content-Length: " + length + "\r\n");
        writer.write("\r\n"); // 空行标识Header和Body的分隔
        writer.write(data.toString());
        writer.flush();
    }
}
}

```

上面代码，大家不需要自己写，主要通过上述代码，只需要大家了解到服务器可以使用java完成编写，是可以接受页面发送的请求和响应数据给前端浏览器的，真正用到的Web服务器，我们不会自己写，都是使用目前比较流行的web服务器，比如Tomcat

小结

1. 响应数据中包含三部分内容，分别是响应行、响应头和响应体
2. 掌握200, 404, 500这三个响应状态码所代表含义，分布是成功、所访问资源不存在和服务的错误

3, Tomcat

3.1 简介

3.1.1 什么是Web服务器

Web服务器是一个应用程序（软件），对HTTP协议的操作进行封装，使得程序员不必直接对协议进行操作，让Web开发更加便捷。主要功能是“提供网上信息浏览服务”。



Web服务器是安装在服务器端的一款软件，将来我们把自己写的Web项目部署到Web Tomcat服务器软件中，当Web服务器软件启动后，部署在Web服务器软件中的页面就可以直接通过浏览器来访问了。

Web服务器软件使用步骤

- 准备静态资源
- 下载安装Web服务器软件
- 将静态资源部署到Web服务器上
- 启动Web服务器使用浏览器访问对应的资源

上述内容在演示的时候，使用的是Apache下的Tomcat软件，至于Tomcat软件如何使用，后面会详细的讲到。而对于Web服务器来说，实现的方案有很多，Tomcat只是其中的一种，而除了Tomcat以外，还有很多优秀的Web服务器，比如：



Tomcat就是一款软件，我们主要是以学习如何去使用为主。具体我们会从以下这些方向去学习：

1. 简介：初步认识下Tomcat
2. 基本使用：安装、卸载、启动、关闭、配置和项目部署，这些都是对Tomcat的基本操作
3. IDEA中如何创建Maven Web项目
4. IDEA中如何使用Tomcat，后面这两个都是我们以后开发经常会用到的方式

首先我们来认识下Tomcat。

Tomcat

Tomcat的相关概念：

- Tomcat是Apache软件基金会一个核心项目，是一个开源免费的轻量级Web服务器，支持Servlet/JSP少量JavaEE规范。

- 概念中提到了JavaEE规范，那什么又是JavaEE规范呢？

JavaEE: Java Enterprise Edition, Java企业版。指Java企业级开发的技术规范总和。包含13项技术规范: JDBC、JNDI、EJB、RMI、JSP、Servlet、XML、JMS、Java IDL、JTS、JTA、JavaMail、JAF。

- 因为Tomcat支持Servlet/JSP规范，所以Tomcat也被称为Web容器、Servlet容器。Servlet需要依赖Tomcat才能运行。

- Tomcat的官网：<https://tomcat.apache.org/> 从官网上可以下载对应的版本进行使用。

Tomcat的LOGO



小结

通过这一节的学习，我们需要掌握以下内容：

- Web服务器的作用

封装HTTP协议操作，简化开发

可以将Web项目部署到服务器中，对外提供网上浏览服务

- Tomcat是一个轻量级的Web服务器，支持Servlet/JSP少量JavaEE规范，也称为Web容器，Servlet容器。

3.2 基本使用

Tomcat总共分两部分学习，先来学习Tomcat的基本使用，包括Tomcat的[下载、安装、卸载、启动和关闭](#)。

3.2.1 下载

直接从官网下载

Binary Distributions 软件
<ul style="list-style-type: none"> Core: <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> tar.gz (pgp, sha512) Deployer: <ul style="list-style-type: none"> zip (pgp, sha512) tar.gz (pgp, sha512) Embedded: <ul style="list-style-type: none"> tar.gz (pgp, sha512) zip (pgp, sha512)
Source Code Distributions 源码
<ul style="list-style-type: none"> tar.gz (pgp, sha512) zip (pgp, sha512)

大家可以自行下载，也可以直接使用资料中已经下载好的资源，

Tomcat的软件程序 资料/2. Tomcat/apache-tomcat-8.5.68-windows-x64.zip

Tomcat的源码 资料/2. Tomcat/tomcat源码/apache-tomcat-8.5.68-src.zip

3.2.2 安装

Tomcat是绿色版，直接解压即可

- 在D盘的software目录下，将apache-tomcat-8.5.68-windows-x64.zip进行解压缩，会得到一个apache-tomcat-8.5.68的目录，Tomcat就已经安装成功。

注意，Tomcat在解压缩的时候，解压所在的目录可以任意，但最好解压到一个不包含中文和空格的目录，因为后期在部署项目的时候，如果路径有中文或者空格可能会导致程序部署失败。

- 打开 apache-tomcat-8.5.68 目录就能看到如下目录结构，每个目录中包含的内容需要认识下，

bin	可执行文件存放目录
conf	配置文件存放目录
lib	tomcat依赖的jar包
logs	日志文件
temp	临时文件
webapps	应用发布目录
work	工作目录

bin: 目录下有两类文件，一种是以 .bat 结尾的，是 Windows 系统的可执行文件，一种是以 .sh 结尾的，是 Linux 系统的可执行文件。

webapps: 就是以后项目部署的目录

至此，Tomcat 的安装就已经完成。

3.2.3 卸载

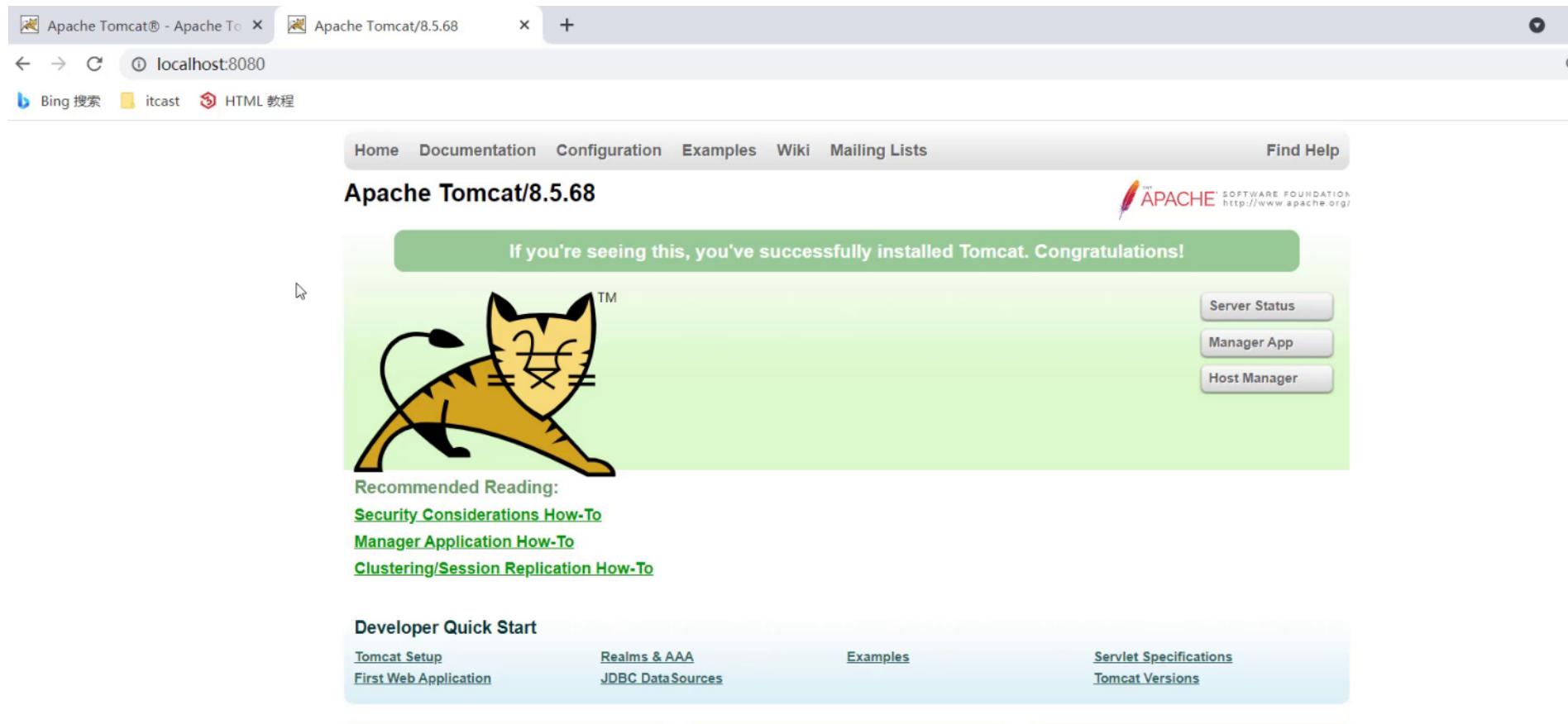
卸载比较简单，可以直接删除目录即可

3.2.4 启动

双击：bin\startup.bat



启动后，通过浏览器访问 <http://localhost:8080> 能看到 Apache Tomcat 的内容就说明 Tomcat 已经启动成功。



注意：启动的过程中，控制台有中文乱码，需要修改 conf/logging.properties

```
java.util.logging.ConsoleHandler.encoding = UTF-8 GBK
```

3.2.5 关闭

关闭有三种方式

- 直接关掉运行窗口：强制关闭 [不建议]
- bin\shutdown.bat：正常关闭
- ctrl+c：正常关闭

3.2.6 配置

修改端口

- Tomcat 默认的端口是 8080，要想修改 Tomcat 启动的端口号，需要修改 conf/server.xml

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

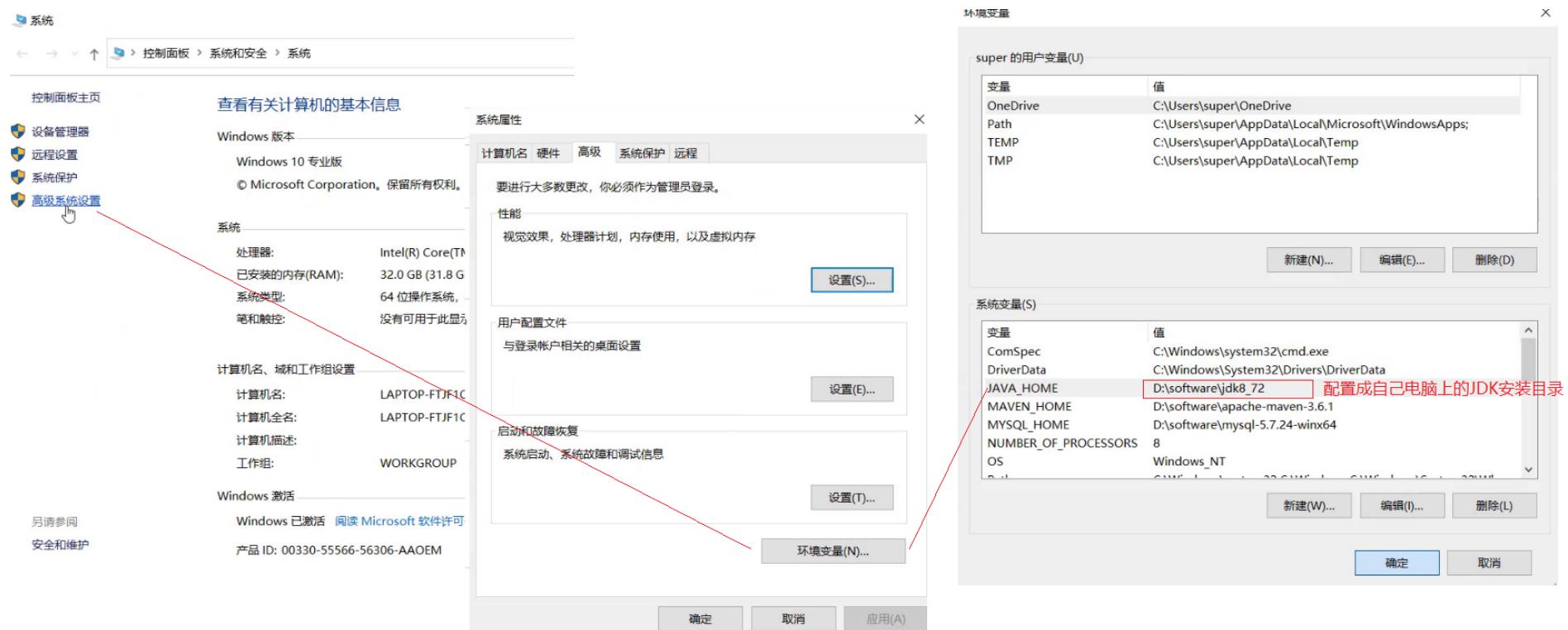
注：HTTP协议默认端口号为80，如果将Tomcat端口号改为80，则将来访问Tomcat时，将不用输入端口号。

启动时可能出现的错误

- Tomcat的端口号取值范围是0-65535之间任意未被占用的端口，如果设置的端口号被占用，启动的时候就会包如下的错误

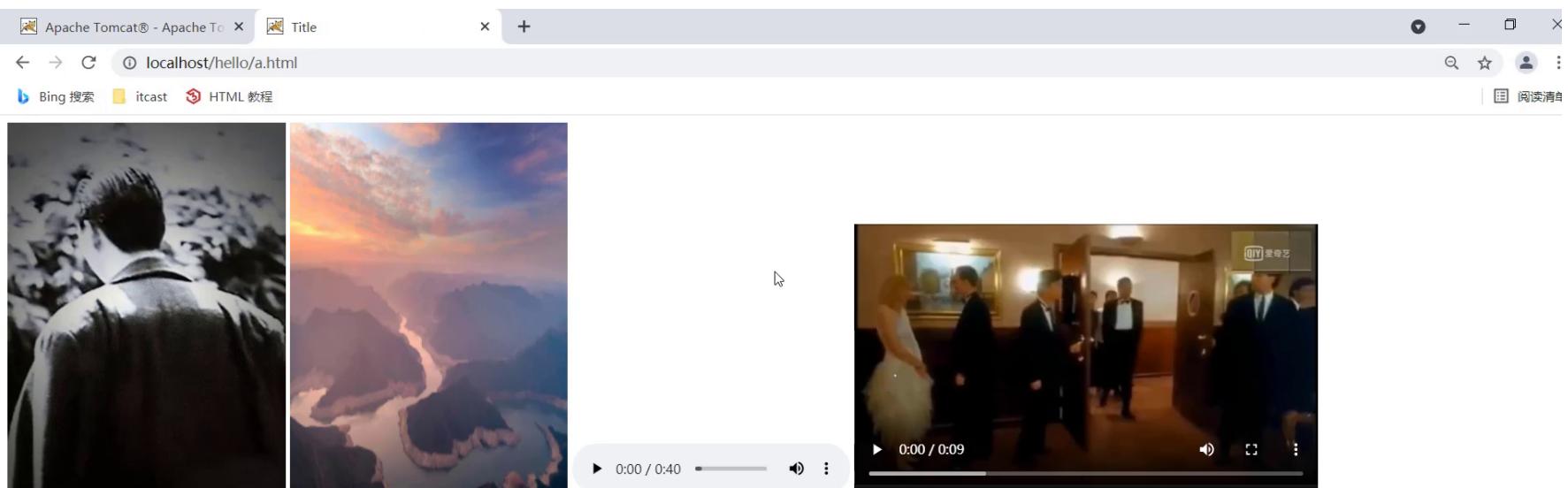
```
at org.apache.catalina.startup.Bootstrap.load(Bootstrap.java:146)
at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:45)
Caused by: java.net.BindException: Address already in use: bind
at sun.nio.ch.Net.bind0(Native Method)
at sun.nio.ch.Net.bind(Net.java:433)
at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:65)
at org.apache.catalina.startup.Bootstrap.bind(Bootstrap.java:175)
```

- Tomcat启动的时候，启动窗口一闪而过：需要检查JAVA_HOME环境变量是否正确配置



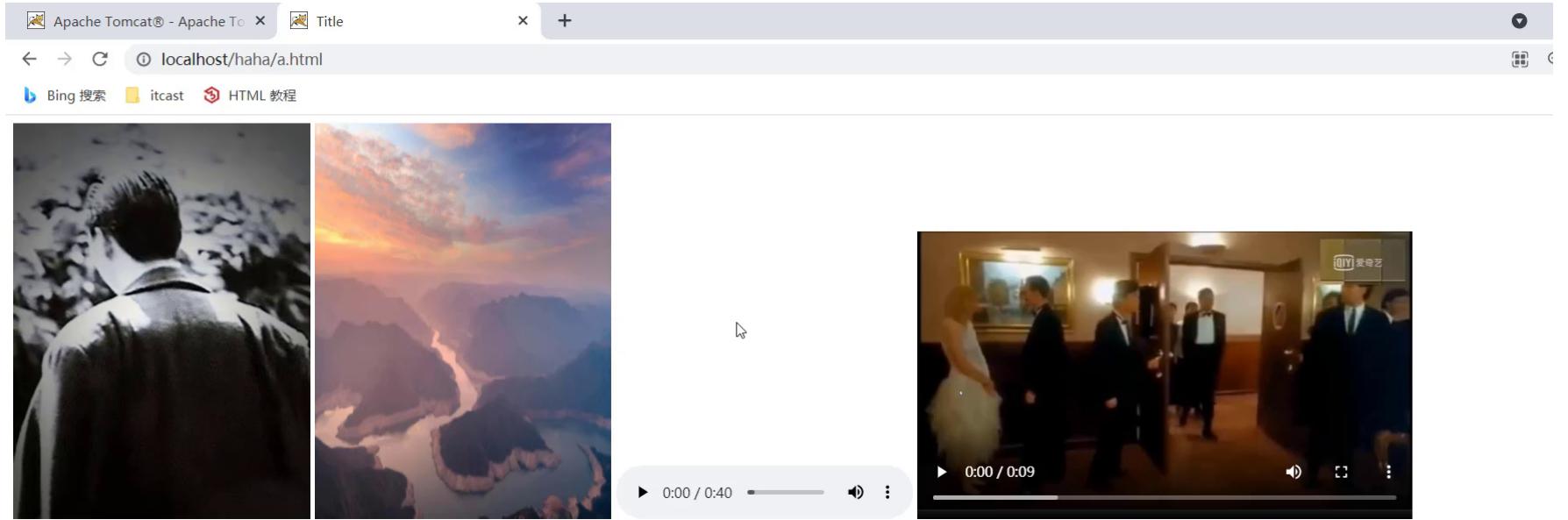
3.2.7 部署

- Tomcat部署项目：将项目放置到webapps目录下，即部署完成。
 - 将 `资料/2. Tomcat/hello` 目录拷贝到Tomcat的webapps目录下
 - 通过浏览器访问 `http://localhost/hello/a.html`，能看到下面的内容就说明项目已经部署成功。



但是呢随着项目的增大，项目中的资源也会越来越多，项目在拷贝的过程中也会越来越费时间，该如何解决呢？

- 一般JavaWeb项目会被打包称war包，然后将war包放到Webapps目录下，Tomcat会自动解压缩war文件
 - 将 `资料/2. Tomcat/haha.war` 目录拷贝到Tomcat的webapps目录下
 - Tomcat检测到war包后会自动完成解压缩，在webapps目录下就会多一个haha目录
 - 通过浏览器访问 `http://localhost/haha/a.html`，能看到下面的内容就说明项目已经部署成功。



至此，Tomcat的部署就已经完成了，至于如何获得项目对应的war包，后期我们会借助于IDEA工具来生成。

3.3 Maven创建Web项目

介绍完Tomcat的基本使用后，我们来学习在IDEA中如何创建Maven Web项目，学习这种方式的原因是以后Tomcat中运行的绝大多数都是Web项目，而使用Maven工具能更加简单快捷的把Web项目给创建出来，所以Maven的Web项目具体如何来构建呢？

在真正创建Maven Web项目之前，我们先要知道Web项目长什么样子，具体的结构是什么？

3.3.1 Web项目结构

Web项目的结构分为：开发中的项目和开发完可以部署的Web项目，这两种项目的结构是不一样的，我们一个个来介绍下：

- Maven Web项目结构：开发中的项目



- 开发完成部署的Web项目



- 开发项目通过执行Maven打包命令`package`，可以获取到部署的Web项目目录
- 编译后的Java字节码文件和resources的资源文件，会被放到WEB-INF下的classes目录下
- pom.xml中依赖坐标对应的jar包，会被放入WEB-INF下的lib目录下

3.3.2 创建Maven Web项目

介绍完Maven Web的项目结构后，接下来使用Maven来创建Web项目，创建方式有两种：使用骨架和不使用骨架

使用骨架

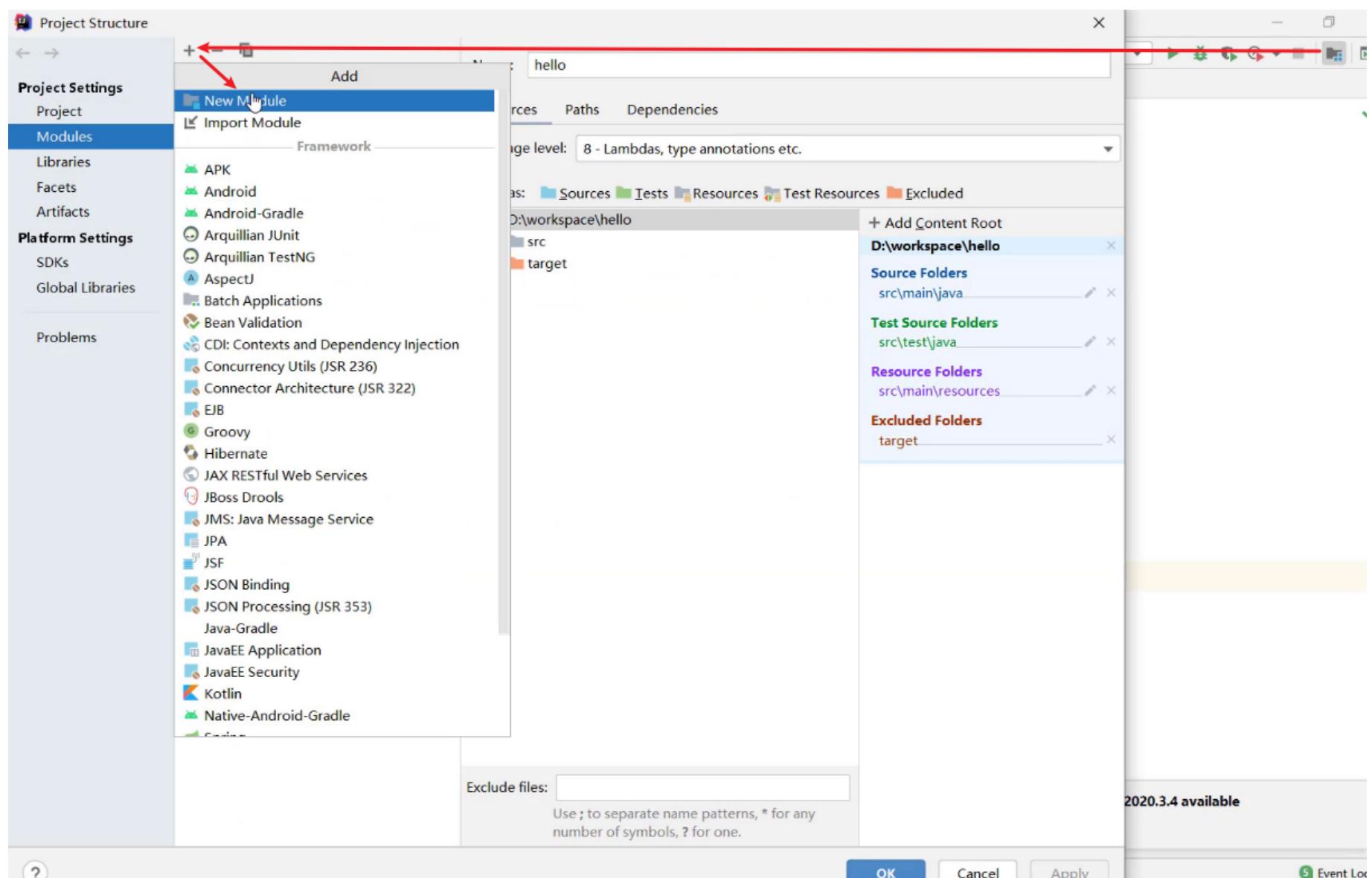
具体的步骤包含：

1. 创建Maven项目
2. 选择使用Web项目骨架
3. 输入Maven项目坐标创建项目
4. 确认Maven相关的配置信息后，完成项目创建

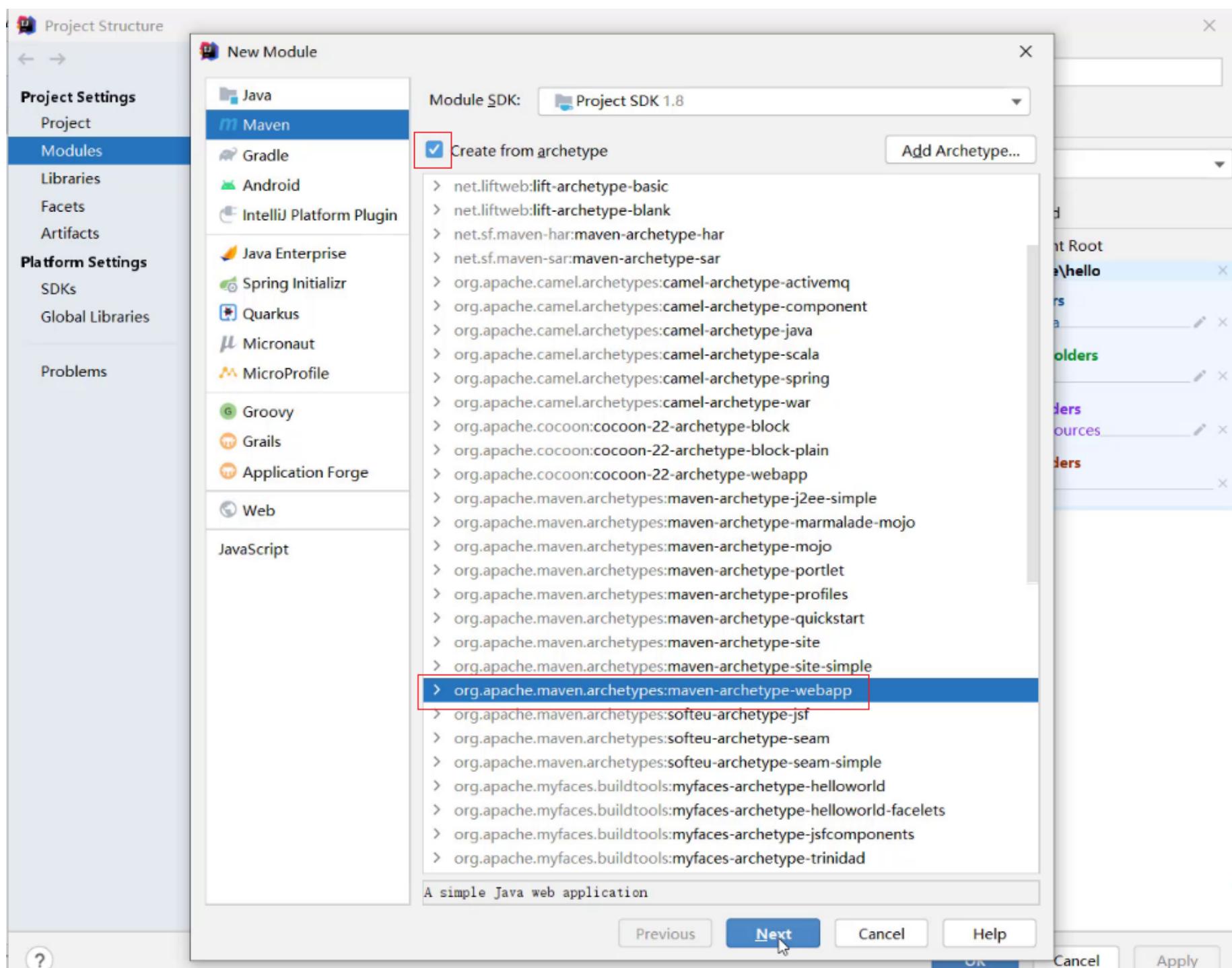
5.删除pom.xml中多余内容

6.补齐Maven Web项目缺失的目录结构

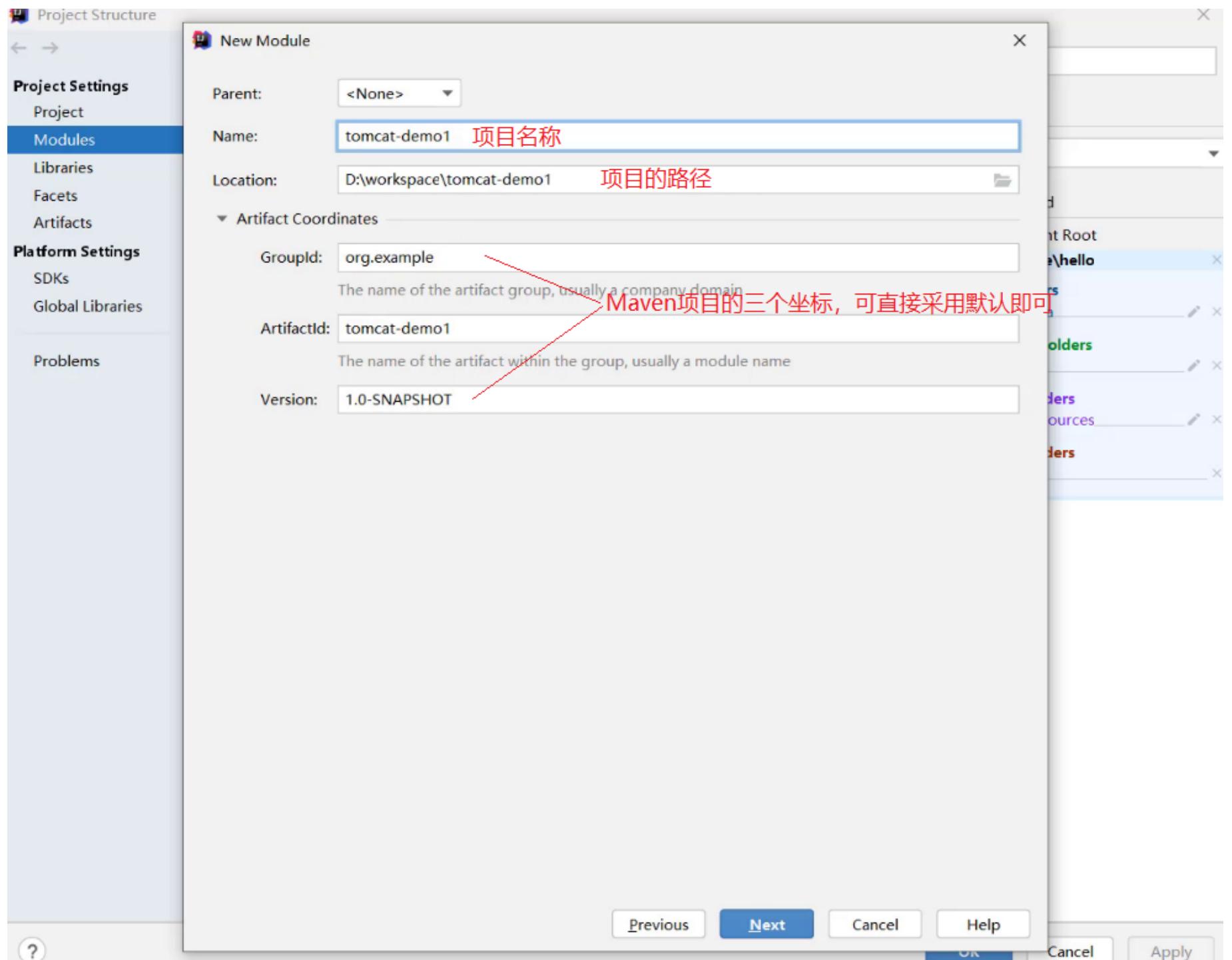
1. 创建Maven项目



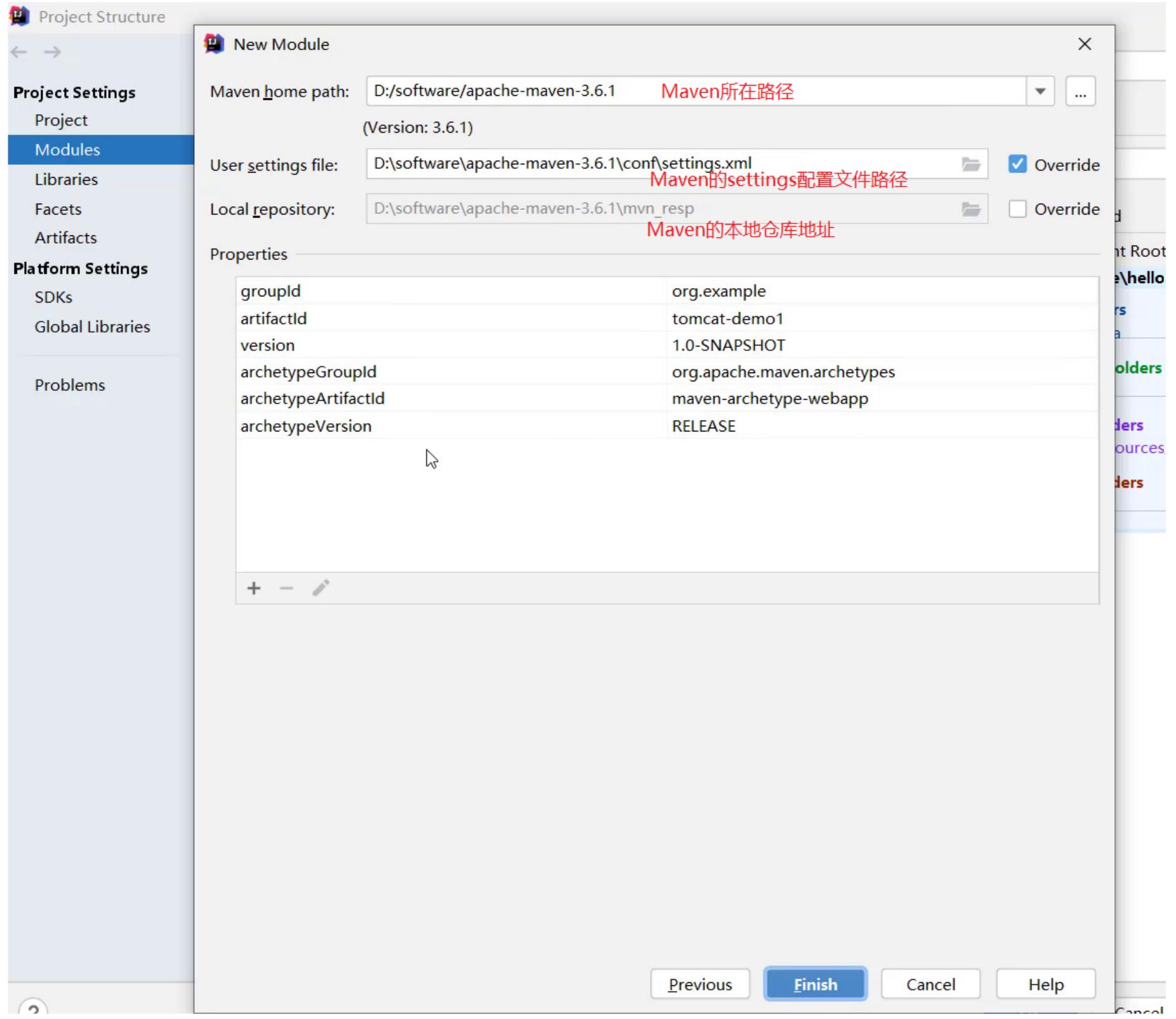
2. 选择使用Web项目骨架



3. 输入Maven项目坐标创建项目



4. 确认Maven相关的配置信息后，完成项目创建



5. 删除pom.xml中多余内容，只留下面的这些内容，注意打包方式 jar和war的区别

```

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

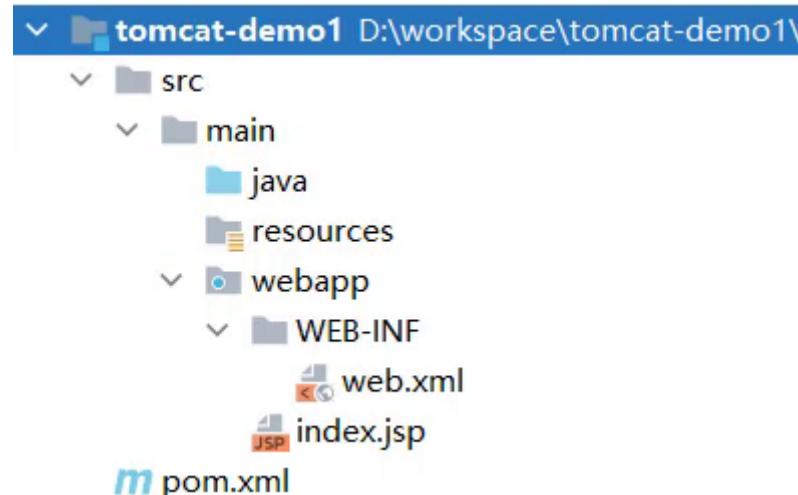
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>tomcat-demo1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <!--
    <packaging>: 打包方式
      * jar: 默认值,
      * war: web项目
  -->
  <packaging>war</packaging>

</project>

```

6. 补齐Maven Web项目缺失的目录结构，默认没有java和resources目录，需要手动完成创建补齐，最终的目录结果如下

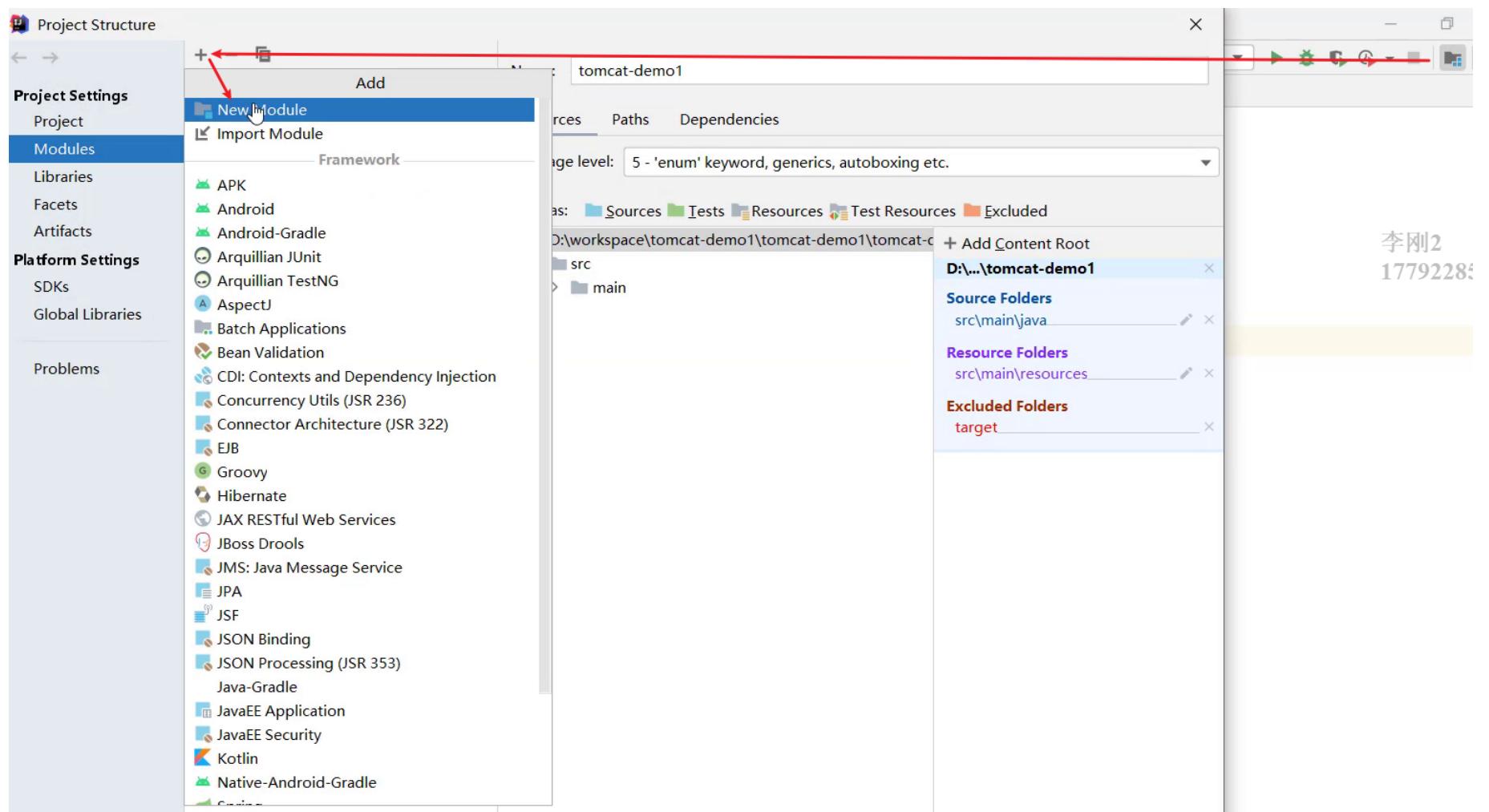


不使用骨架

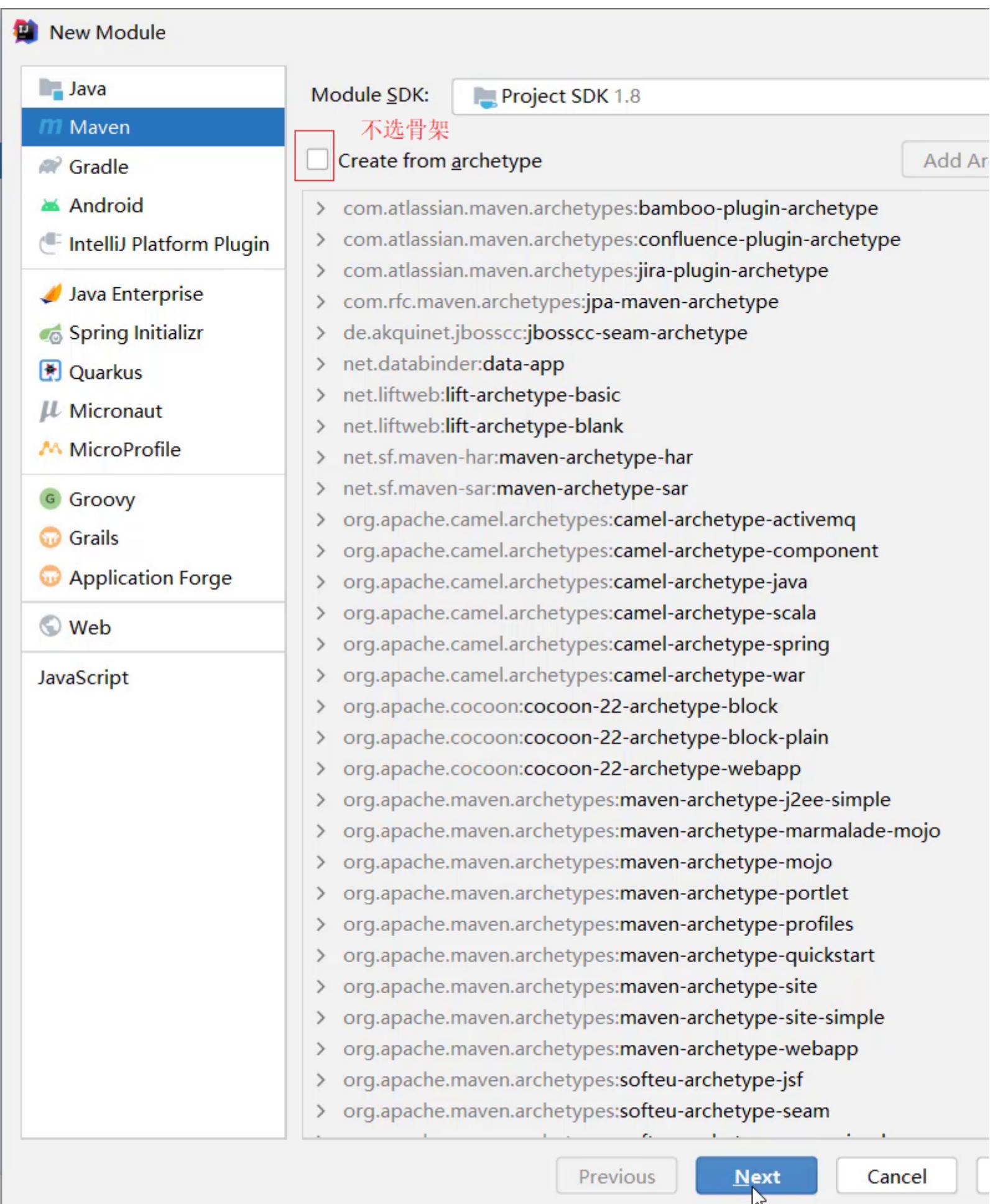
具体的步骤包含：

1. 创建Maven项目
2. 选择不使用Web项目骨架
3. 输入Maven项目坐标创建项目
4. 在pom.xml设置打包方式为war
5. 补齐Maven Web项目缺失webapp的目录结构
6. 补齐Maven Web项目缺失WEB-INF/web.xml的目录结构

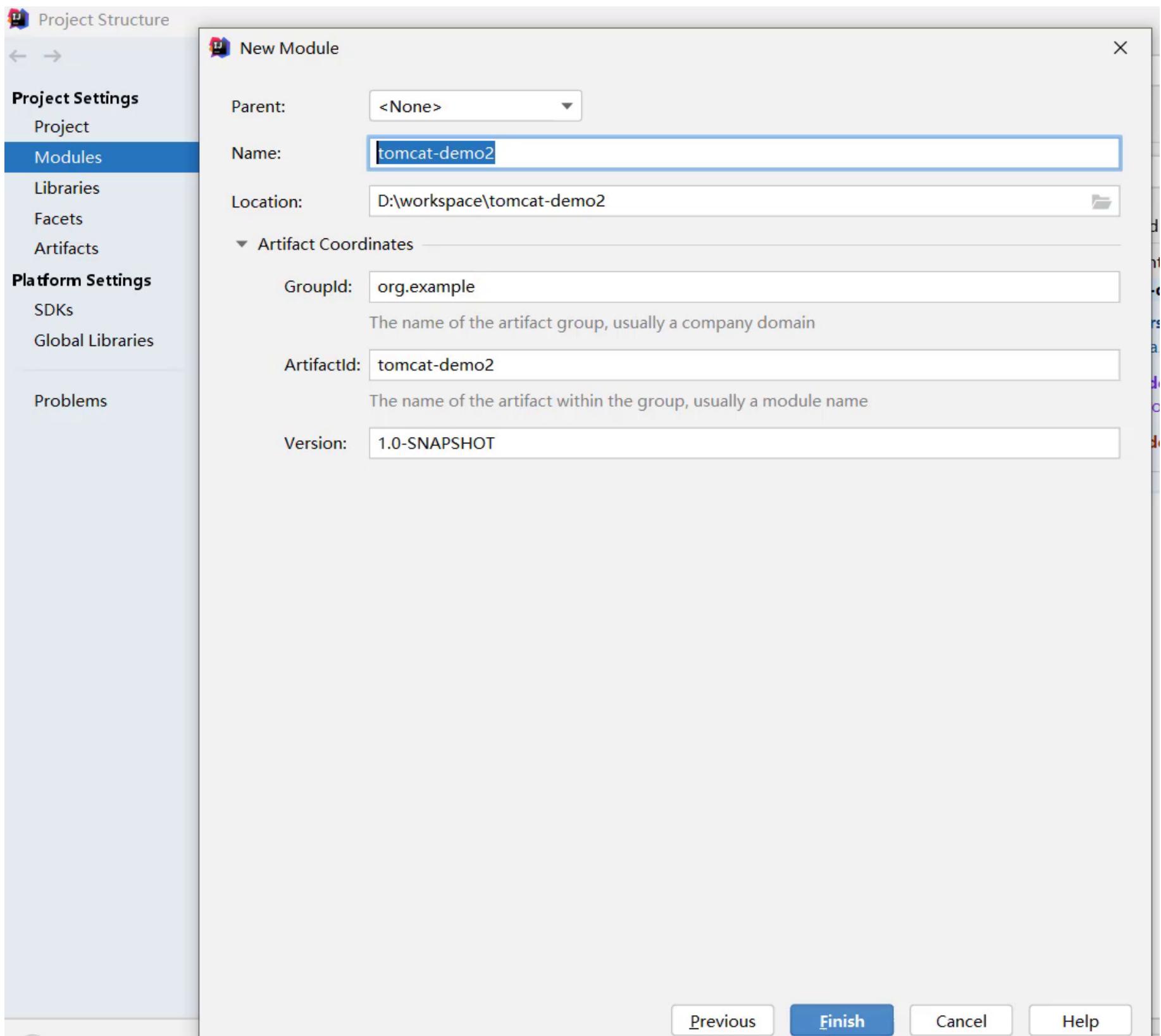
1. 创建Maven项目



2. 选择不使用Web项目骨架



3. 输入Maven项目坐标创建项目



4. 在pom.xml设置打包方式为war，默认是不写代表打包方式为jar

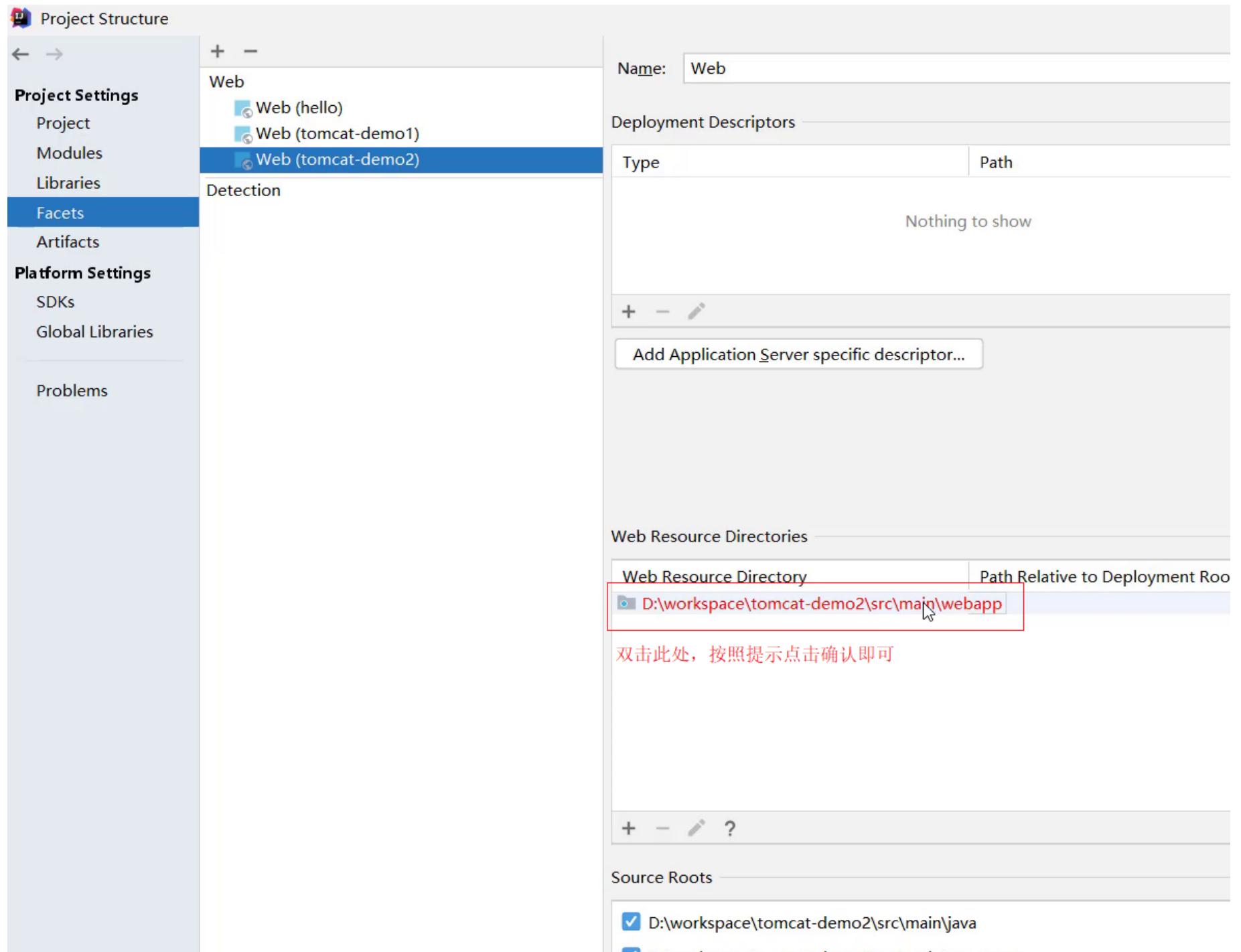
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.a
<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>
<artifactId>tomcat-demo2</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

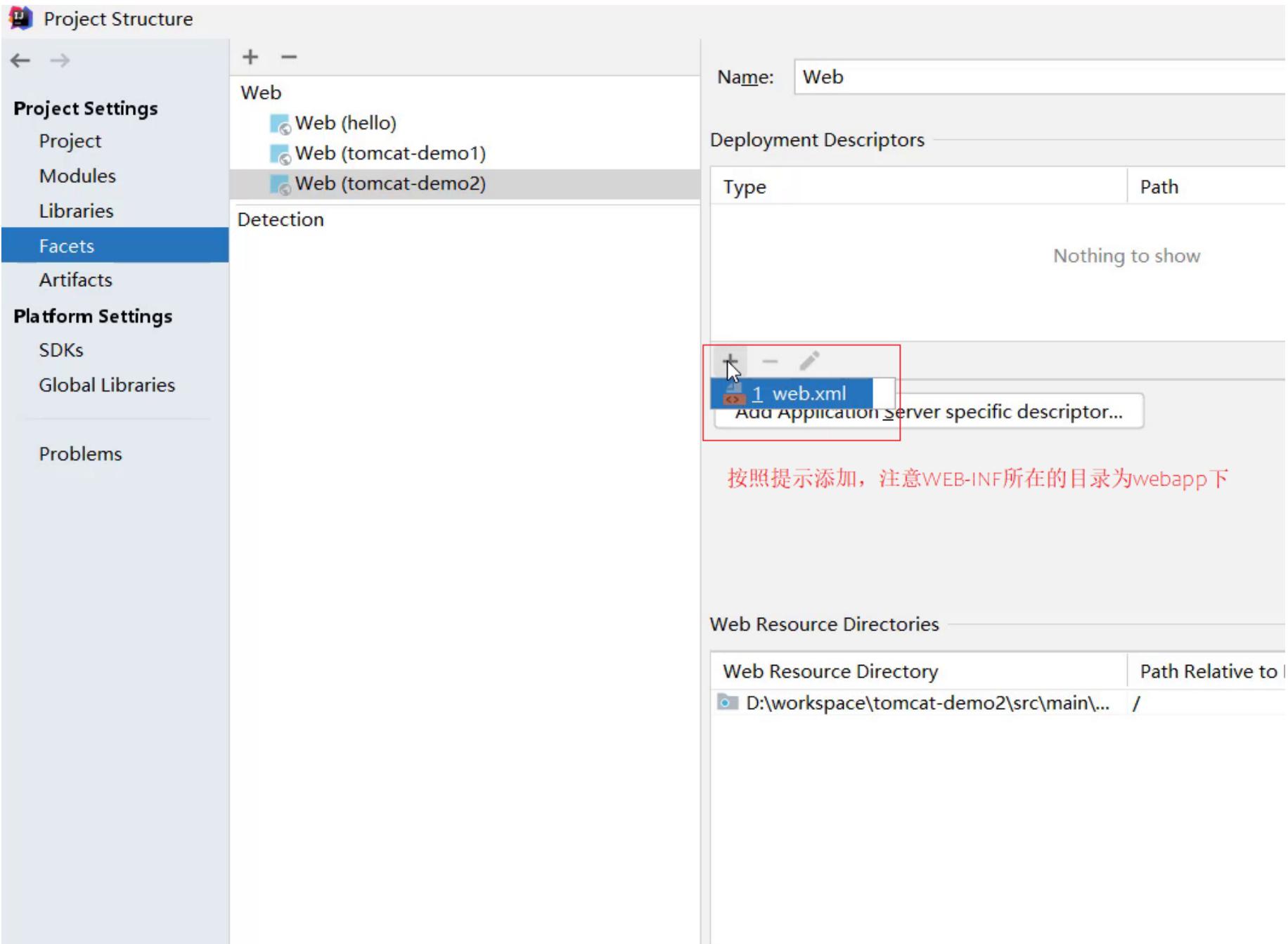
<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
</properties>

</project>
```

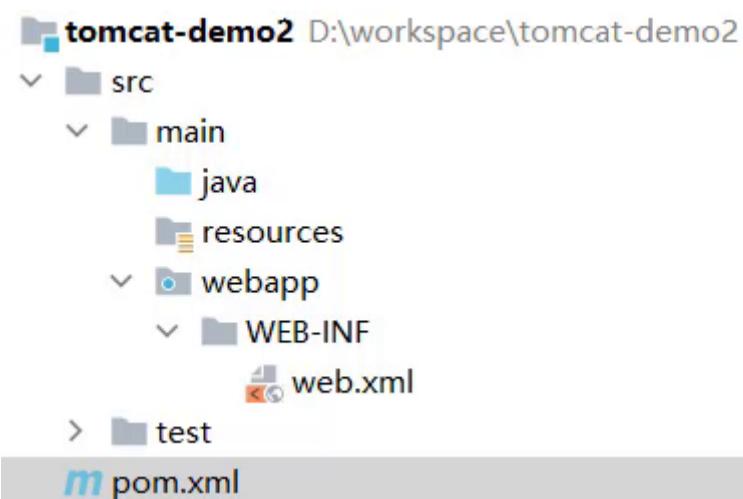
5. 补齐Maven Web项目缺失webapp的目录结构



6. 补齐Maven Web项目缺失WEB-INF/web.xml的目录结构



7. 补充完后，最终的项目结构如下：

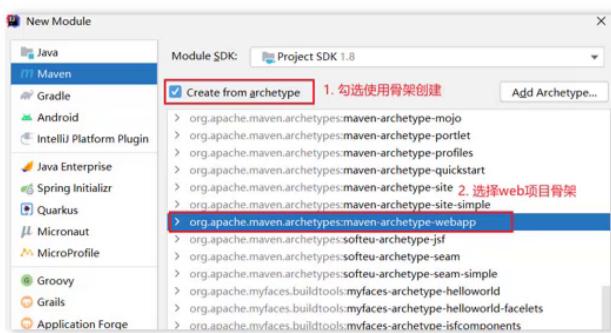


上述两种方式，创建的web项目，都不是很全，需要手动补充内容，至于最终采用哪种方式来创建Maven Web项目，都是可以的，根据各自的喜好来选择使用即可。

小结

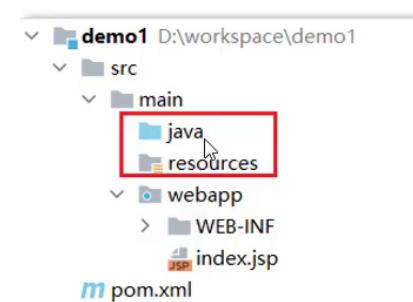
1. 掌握Maven Web项目的目录结构
2. 掌握使用骨架的方式创建Maven Web项目

1. 选择web项目骨架，创建项目

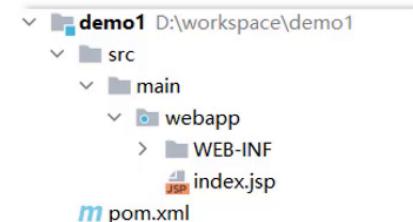


2. 删除pom.xml中多余的坐标

```
<build>
  <finalName>demo1</finalName>
  <pluginManagement><!-- lock down pl
    <plugins>
      <plugin>
```

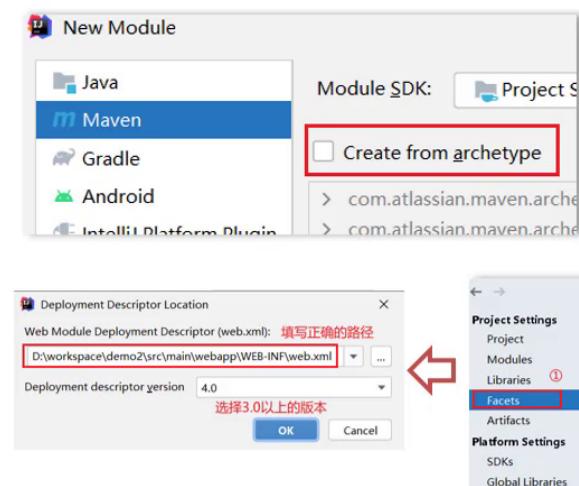


3. 补齐缺失的目录结构



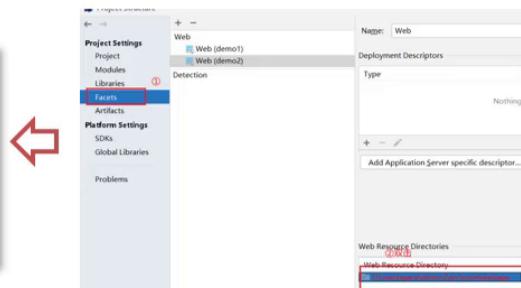
3. 掌握不使用骨架的方式创建Maven Web项目

1. 选择web项目骨架，创建项目

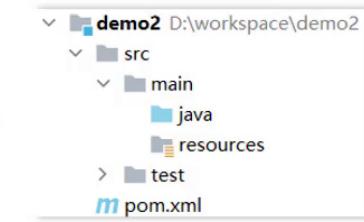


2. pom.xml中添加打包方式为war

```
<groupId>com.itheima</groupId>
<artifactId>demo2</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
```



3. 补齐缺失的目录结构：webapp



3.4 IDEA使用Tomcat

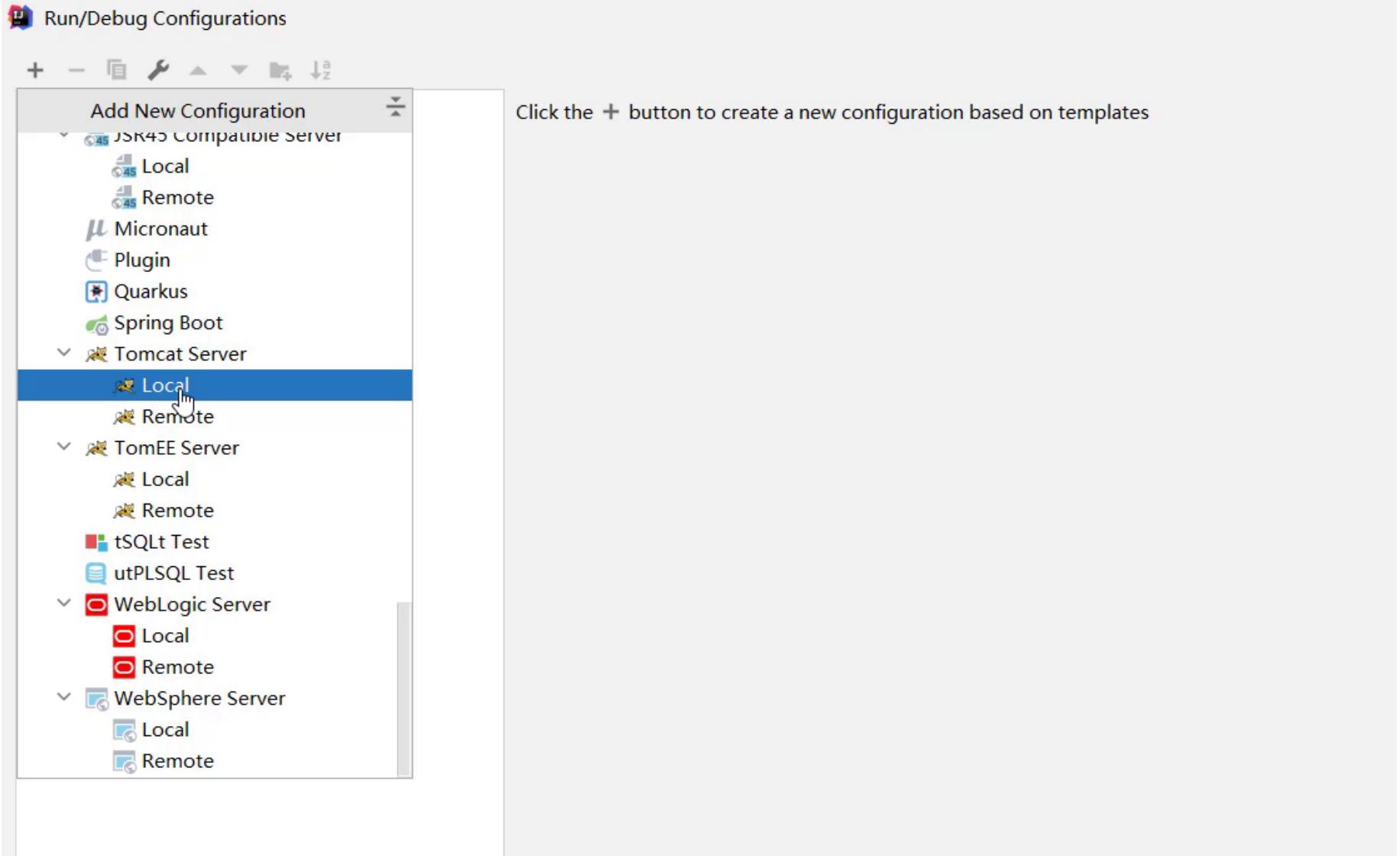
- Maven Web项目创建成功后，通过Maven的package命令可以将项目打成war包，将war文件拷贝到Tomcat的webapps目录下，启动Tomcat就可以将项目部署成功，然后通过浏览器进行访问即可。
- 然而我们在开发的过程中，项目中的内容会经常发生变化，如果按照上面这种方式来部署测试，是非常不方便的
- 如何在IDEA中能快速使用Tomcat呢？

在IDEA中集成使用Tomcat有两种方式，分别是**集成本地Tomcat**和**Tomcat Maven插件**

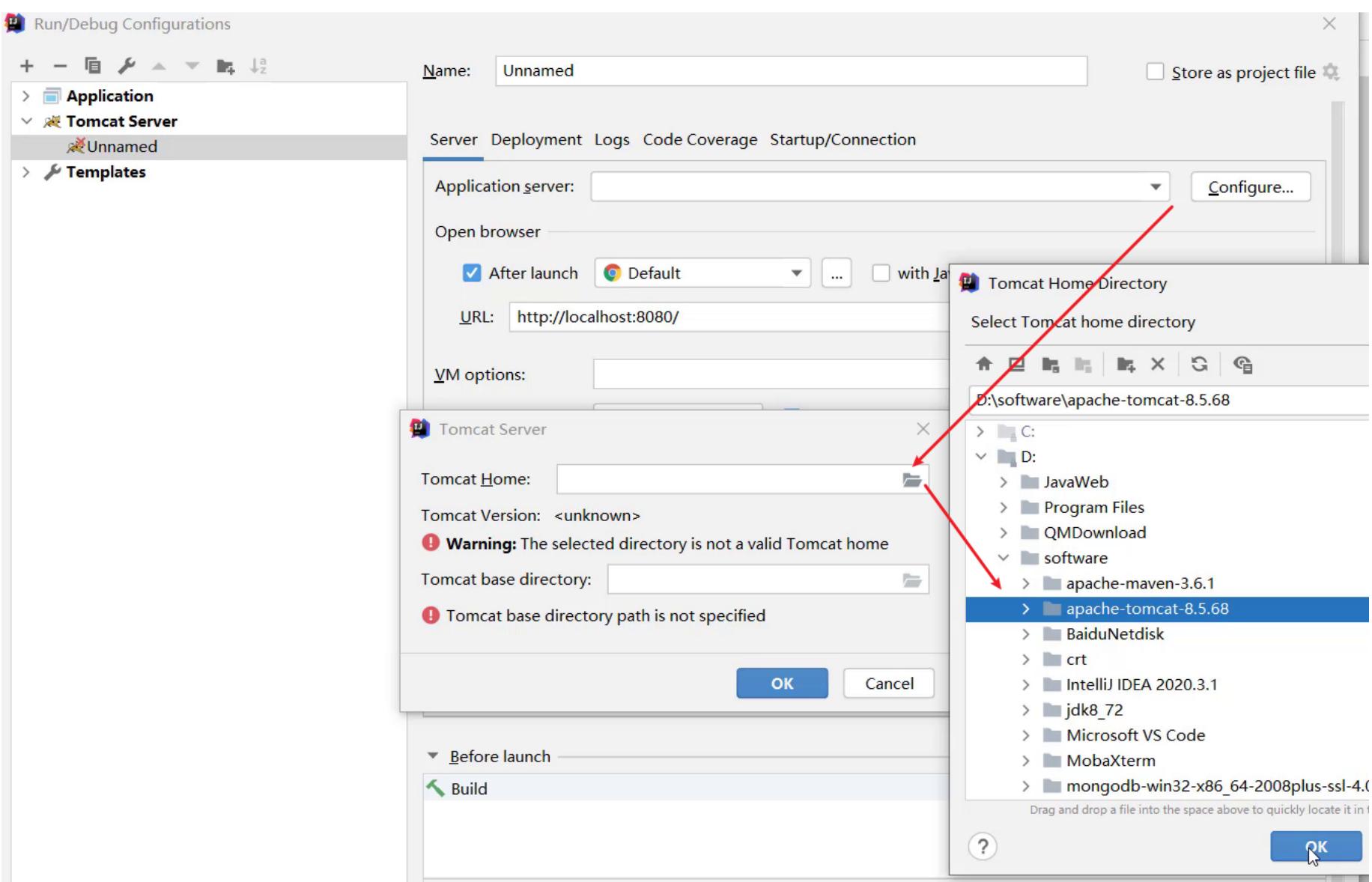
3.4.1 集成本地Tomcat

目标：将刚才本地安装好的Tomcat8集成到IDEA中，完成项目部署，具体的实现步骤

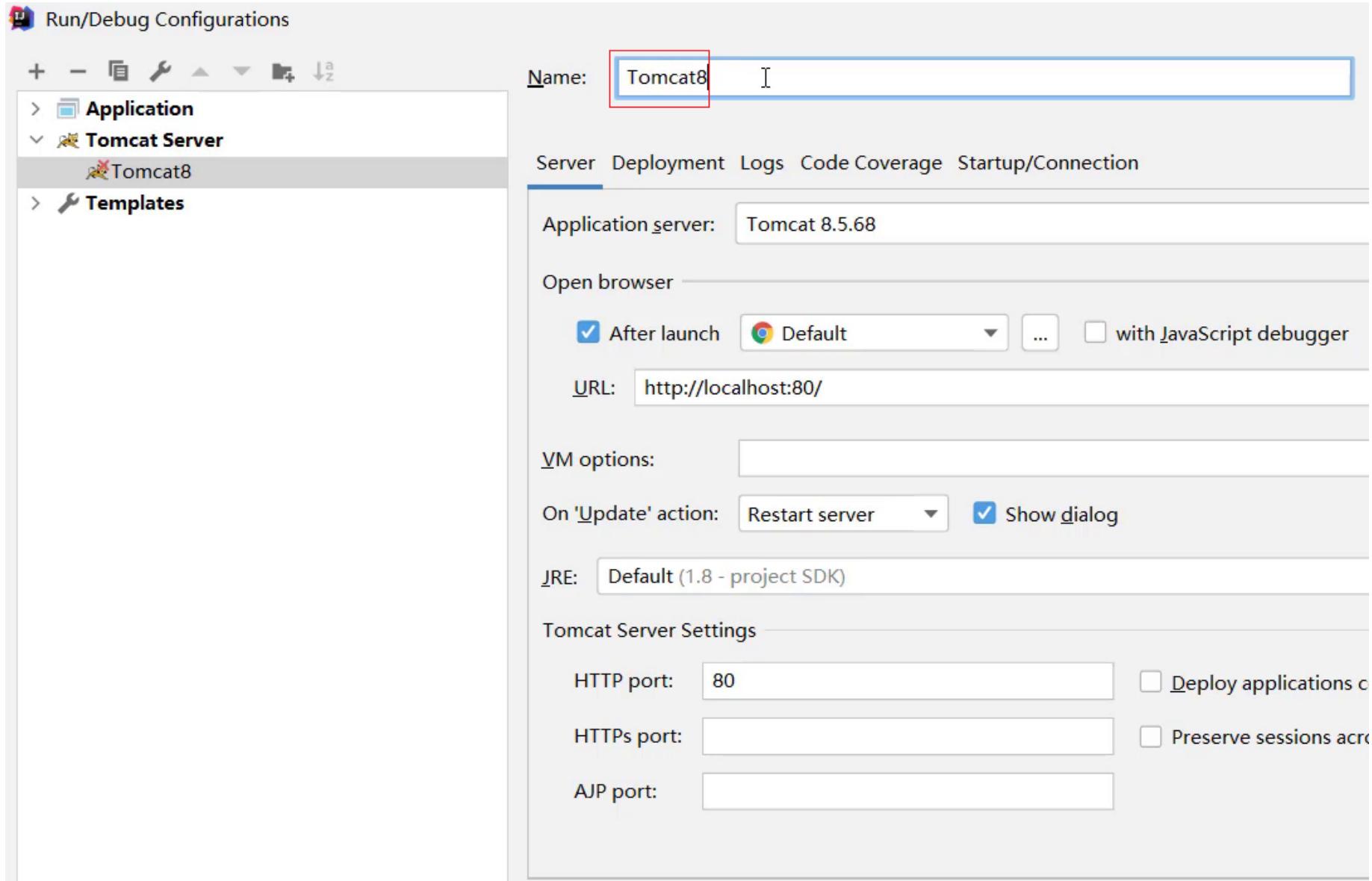
1. 打开添加本地Tomcat的面板



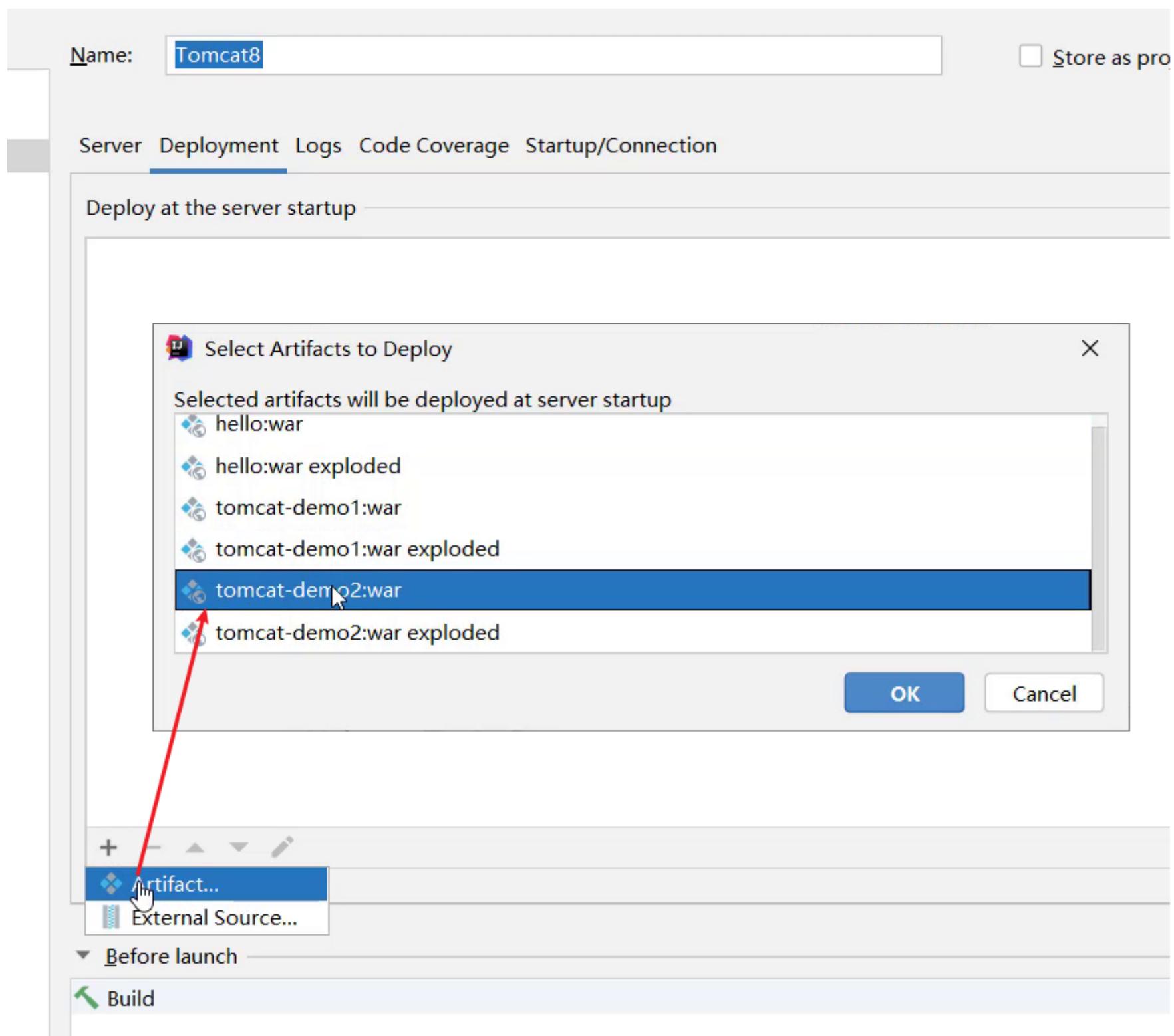
2. 指定本地Tomcat的具体路径



3. 修改Tomcat的名称，此步骤可以不改，只是让名字看起来更有意义，HTTP port中的端口也可以进行修改，比如把8080改成80



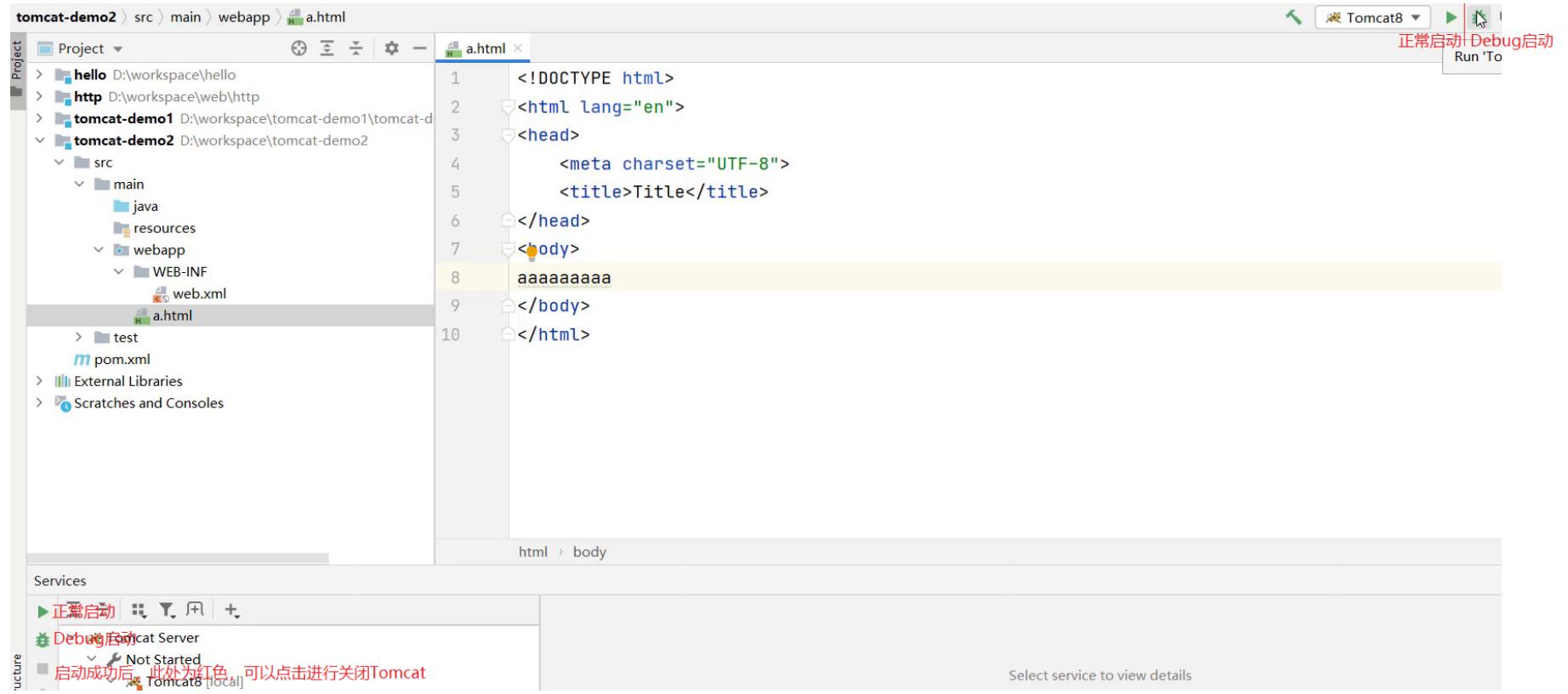
4. 将开发项目部署项目到Tomcat中



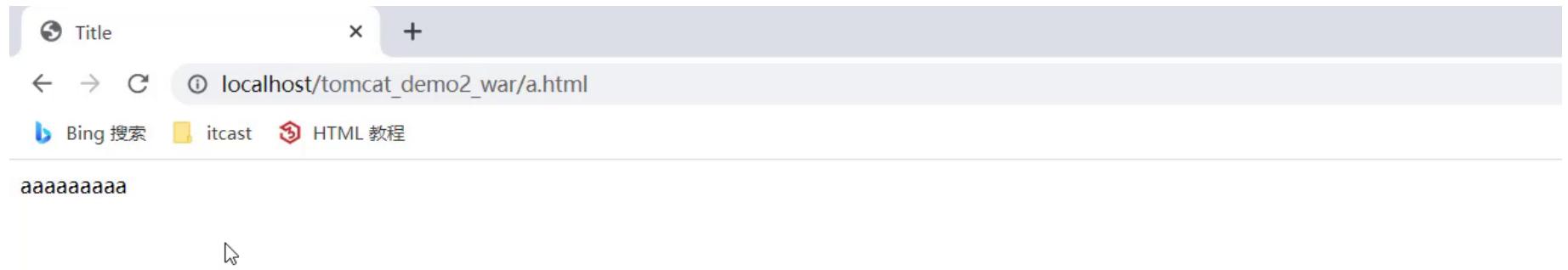
扩展内容： xxx.war 和 xxx.war exploded 这两种部署项目模式的区别？

- war模式是将WEB工程打成war包，把war包发布到Tomcat服务器上
- war exploded模式是将WEB工程以当前文件夹的位置关系发布到Tomcat服务器上
- war模式部署成功后，Tomcat的webapps目录下会有部署的项目内容
- war exploded模式部署成功后，Tomcat的webapps目录下没有，而使用的是项目的target目录下的内容进行部署
- 建议大家都选war模式进行部署，更符合项目部署的实际情况

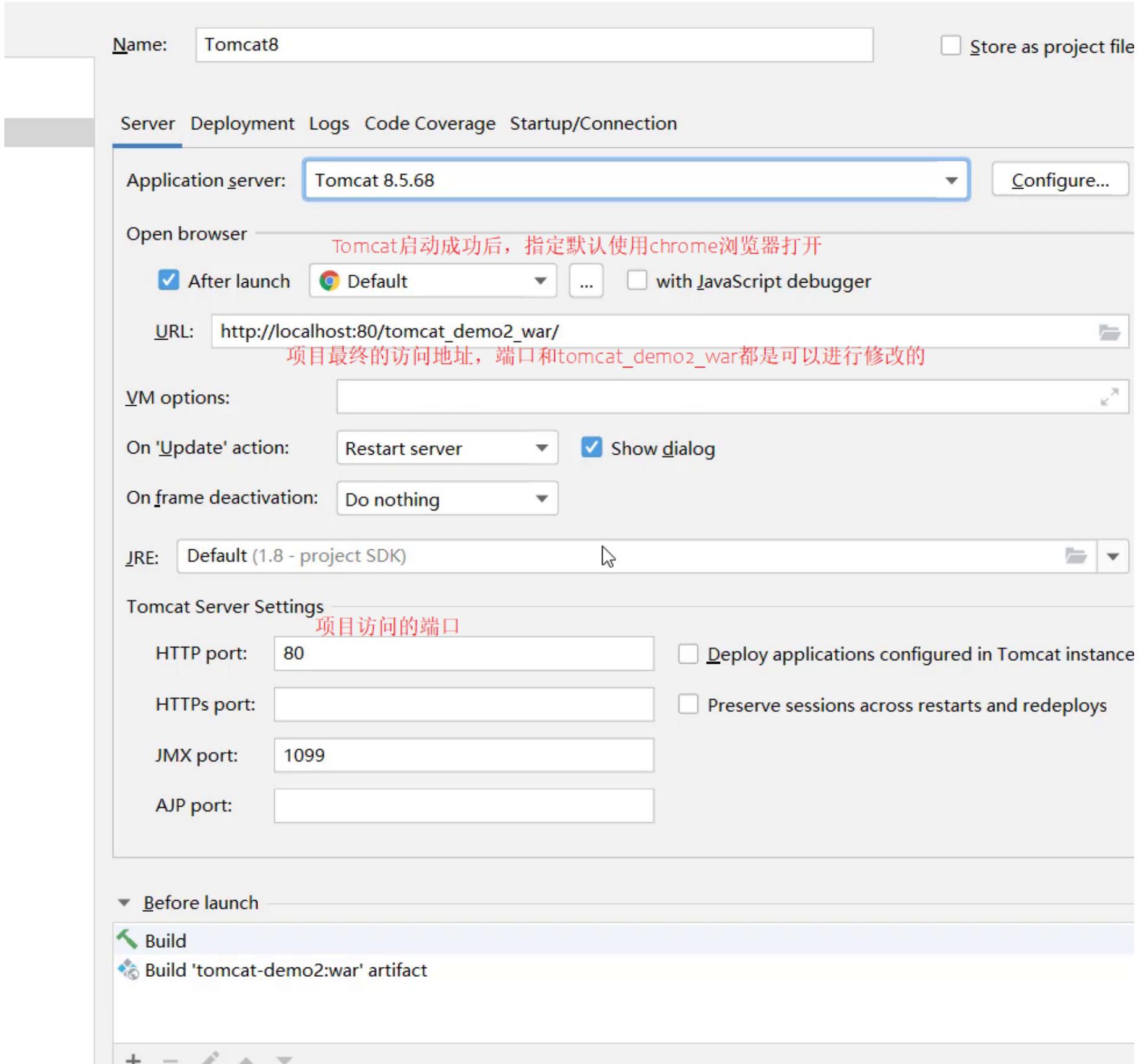
5. 部署成功后，就可以启动项目，为了能更好的看到启动的效果，可以在webapp目录下添加a.html页面



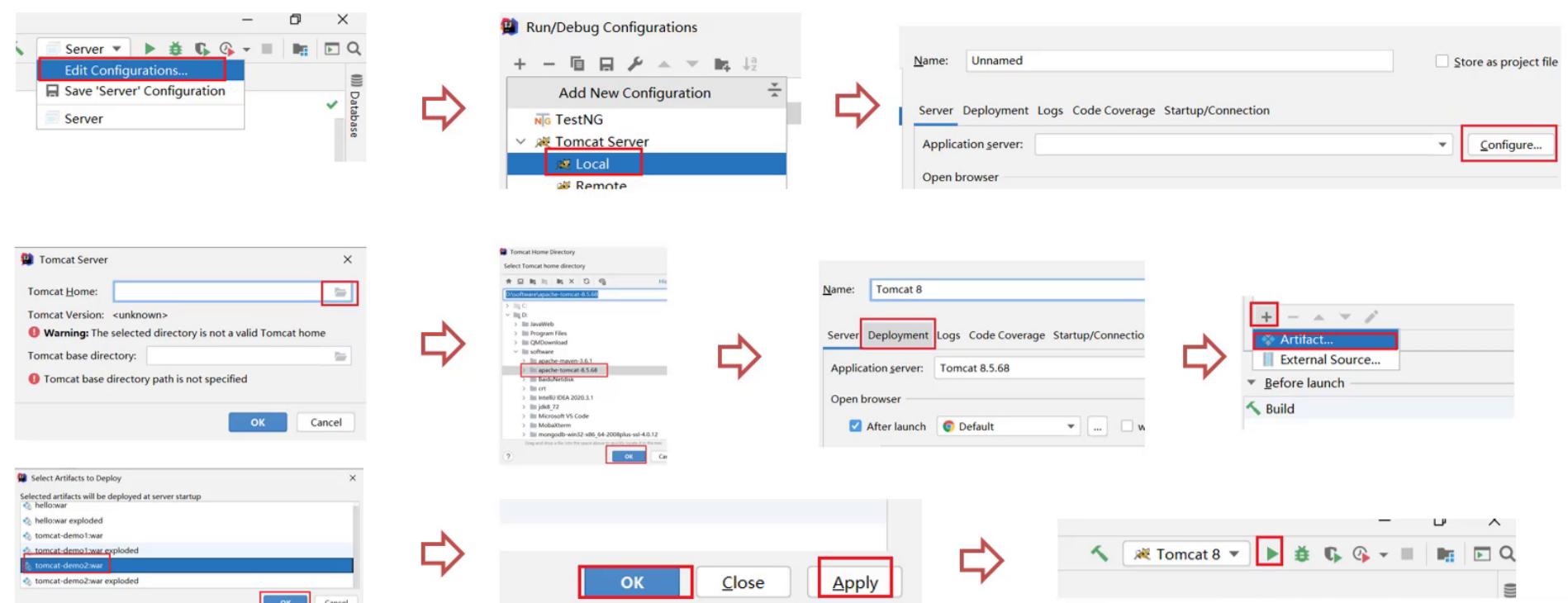
6. 启动成功后，可以通过浏览器进行访问测试



7. 最终的注意事项



至此，IDEA中集成本地Tomcat进行项目部署的内容我们就介绍完了，整体步骤如下，大家需要按照流程进行部署操作练习。



3.4.2 Tomcat Maven插件

在IDEA中使用本地Tomcat进行项目部署，相对来说步骤比较繁琐，所以我们需要一种更简便的方式来替换它，那就是直接使用Maven中的Tomcat插件来部署项目，具体的实现步骤，只需要两步，分别是：

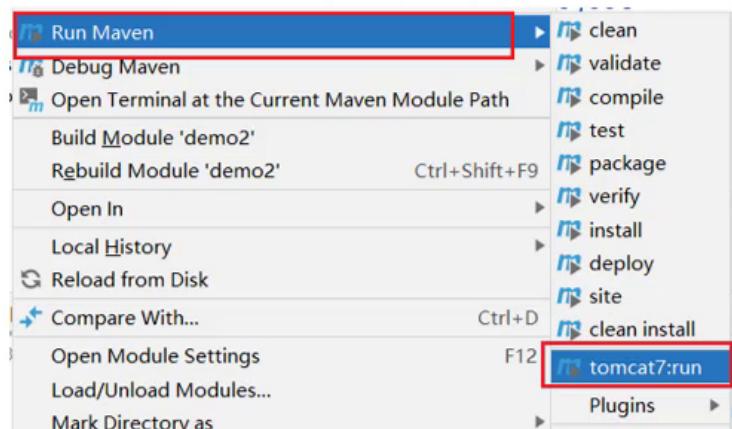
1. 在pom.xml中添加Tomcat插件

```

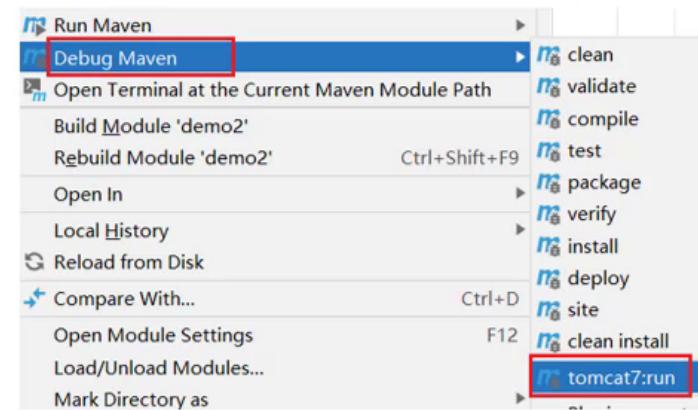
<build>
  <plugins>
    <!--Tomcat插件 -->
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
    </plugin>
  </plugins>
</build>

```

2. 使用Maven Helper插件快速启动项目，选中项目，右键-->Run Maven --> tomcat7:run

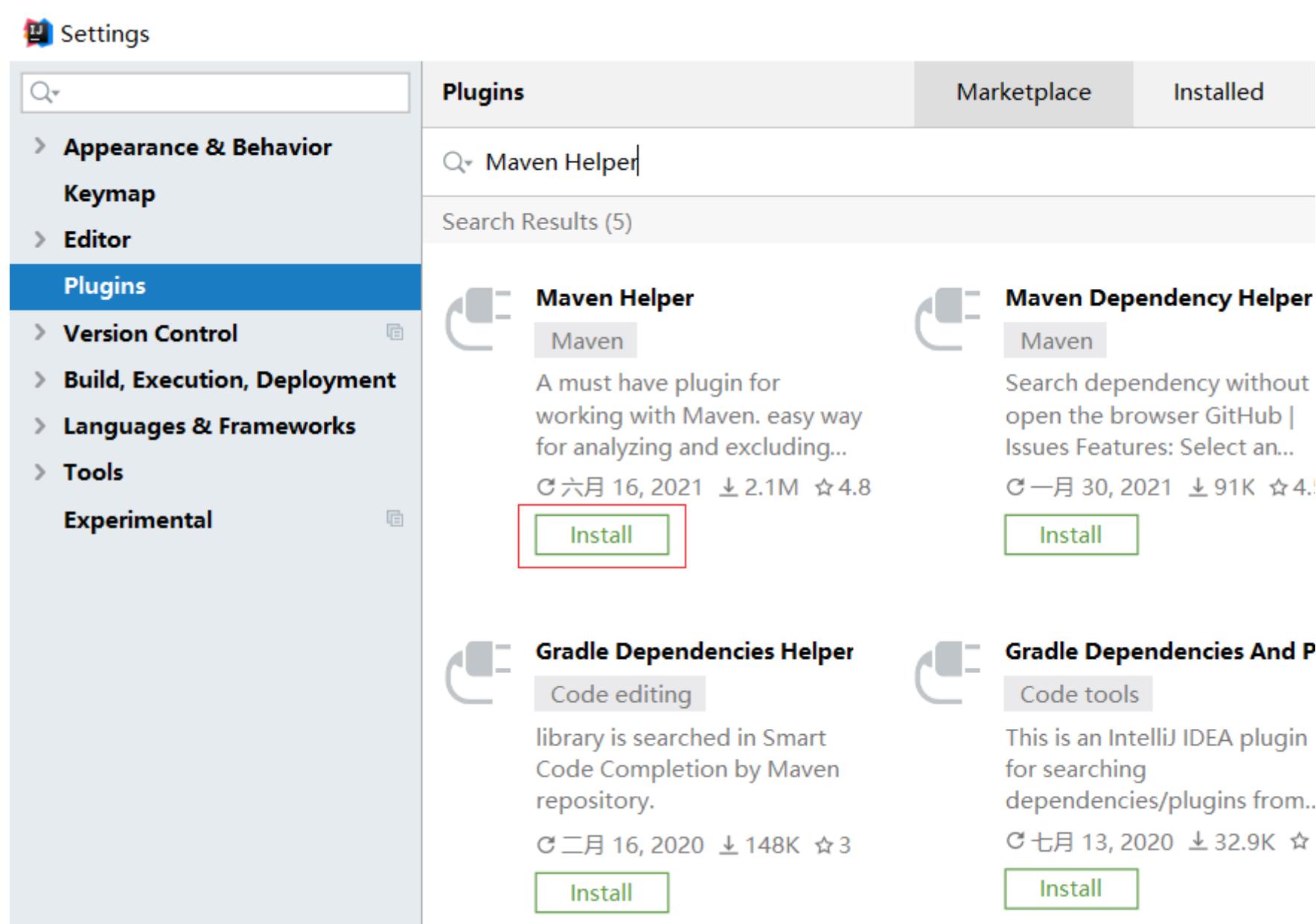


如果需要断点调试，选择 Debug Maven



注意：

- 如果选中项目并右键点击后，看不到Run Maven和Debug Maven，这个时候就需要在IDEA中下载Maven Helper插件，具体的操作方式为：File --> Settings --> Plugins --> Maven Helper ---> Install，安装完后按照提示重启IDEA，就可以看到了。



- Maven Tomcat插件目前只有Tomcat7版本，没有更高的版本可以使用
- 使用Maven Tomcat插件，要想修改Tomcat的端口和访问路径，可以直接修改pom.xml

```

<build>
  <plugins>
    <!--Tomcat插件 -->
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
      <configuration>
        <port>80</port><!--访问端口号 -->
      </configuration>
    </plugin>
  </plugins>
</build>

```

```

<!--项目访问路径
未配置访问路径: http://localhost:80/tomcat-demo2/a.html
配置/后访问路径: http://localhost:80/a.html
如果配置成 /hello,访问路径会变成什么?
答案: http://localhost:80/hello/a.html

-->
<path>/</path>
</configuration>
</plugin>
</plugins>
</build>

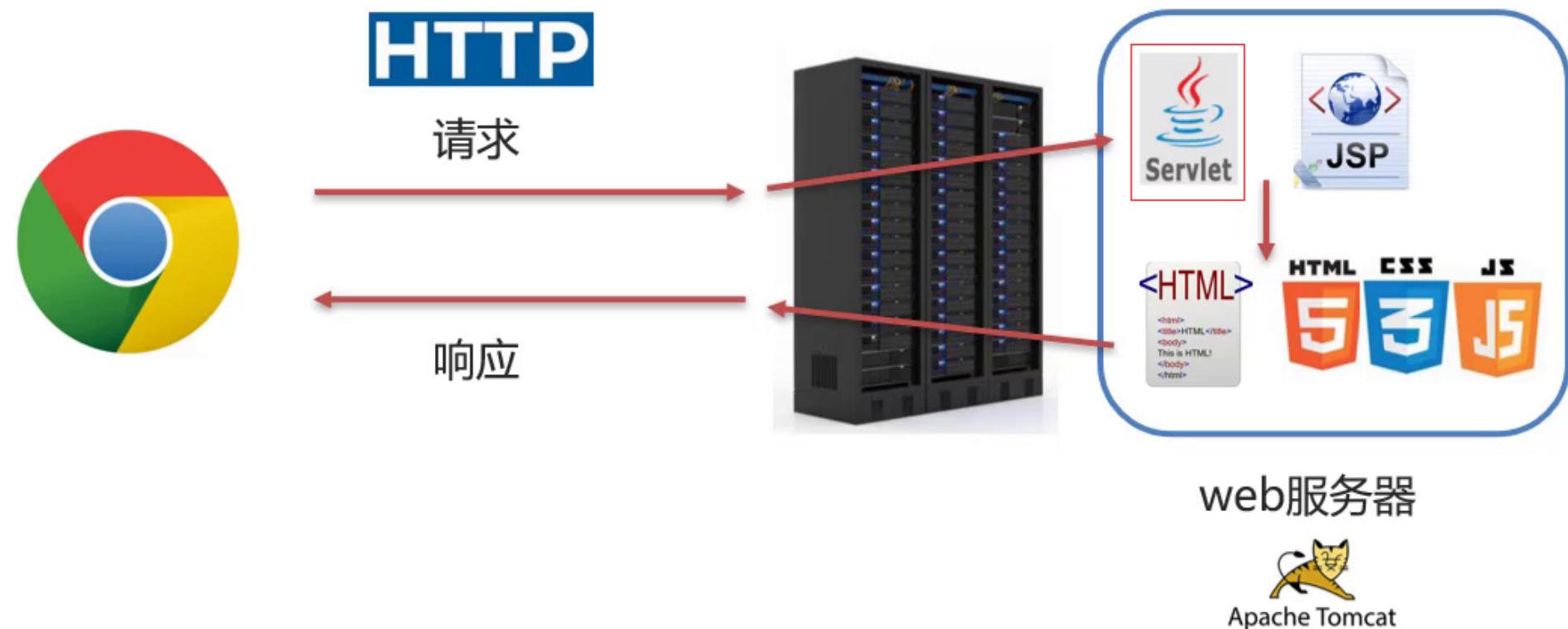
```

小结

通过这一节的学习，大家要掌握在IDEA中使用Tomcat的两种方式，集成本地Tomcat和使用Maven的Tomcat插件。后者更简单，推荐大家使用，但是如果对于Tomcat的版本有比较高的要求，要在Tomcat7以上，这个时候就只能用前者了。

4. Servlet

4.1 简介



- Servlet是JavaWeb最为核心的内容，它是Java提供的一门**动态**web资源开发技术。
- 使用Servlet就可以实现，根据不同的登录用户在页面上动态显示不同内容。
- Servlet是JavaEE规范之一，其实就是一个接口，将来我们需要定义Servlet类实现Servlet接口，并由web服务器运行Servlet

`public interface Servlet`

Defines methods that all servlets must implement.

A servlet is a small Java program that runs within a Web server.

介绍完Servlet是什么以后，接下来我们就按照快速入门->执行流程->生命周期->体系结构->url Pattern配置->XML配置的学习步骤，一步步完成对Servlet的知识学习，首选我们来通过一个入门案例来快速把Servlet用起来。

4.2 快速入门

需求分析：编写一个Servlet类，并使用IDEA中Tomcat插件进行部署，最终通过浏览器访问所编写的servlet程序。

具体的实现步骤为：

1. 创建Web项目 `web-demo`，导入Servlet依赖坐标

```

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <!--
        此处为什么需要添加该标签？
        provided指的是在编译和测试过程中有效，最后生成的war包时不会加入
        因为Tomcat的lib目录中已经有servlet-api这个jar包，如果在生成war包的时候生效就会和Tomcat中的jar包冲突，导致报错
    -->
    <scope>provided</scope>
</dependency>

```

2. 创建：定义一个类，实现Servlet接口，并重写接口中所有方法，并在service方法中输入一句话

```
package com.itheima.web;

import javax.servlet.*;
import java.io.IOException;

public class ServletDemo1 implements Servlet {

    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
        System.out.println("servlet hello world~");
    }

    public void init(ServletConfig servletConfig) throws ServletException {
    }

    public ServletConfig getServletConfig() {
        return null;
    }

    public String getServletInfo() {
        return null;
    }

    public void destroy() {
    }
}
```

3. 配置：在类上使用@WebServlet注解，配置该Servlet的访问路径

```
@WebServlet("/demo1")
```

4. 访问：启动Tomcat，浏览器中输入URL地址访问该Servlet

```
http://localhost:8080/web-demo/demo1
```

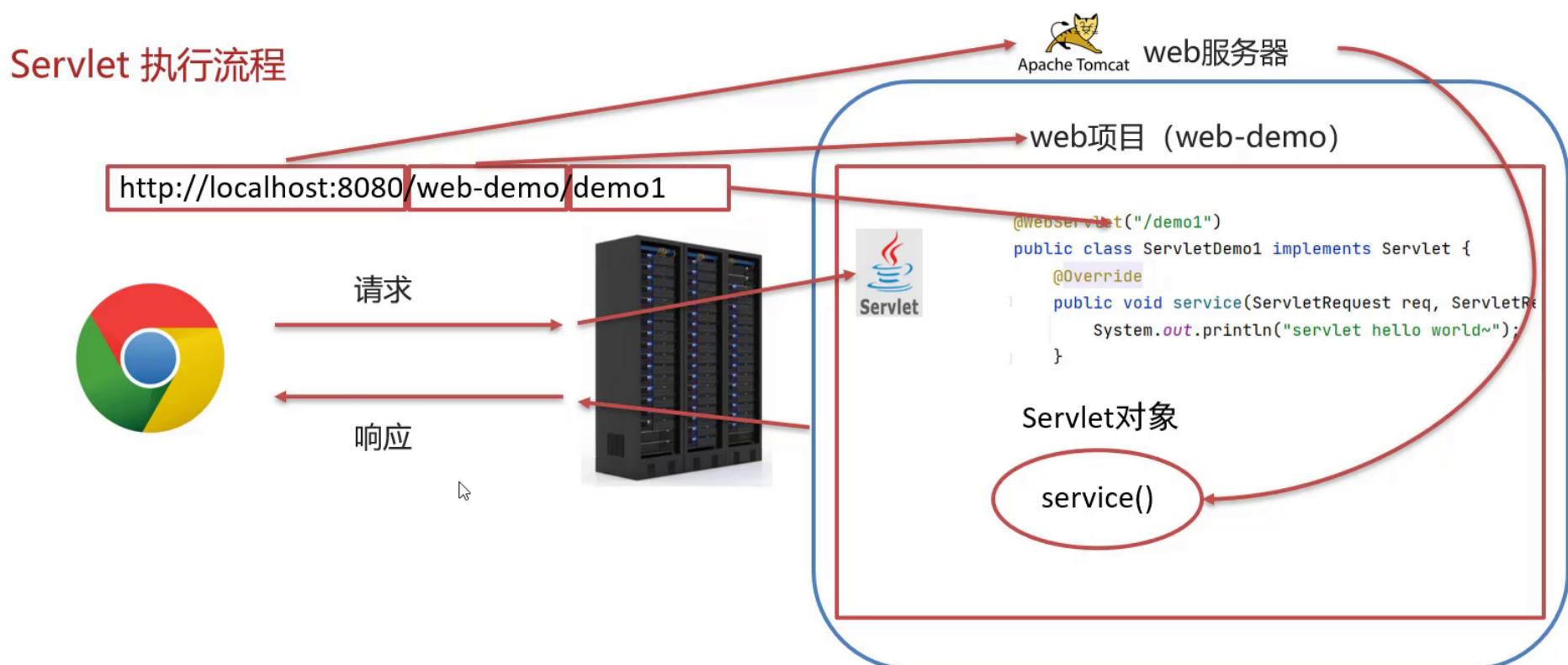
5. 器访问后，在控制台会打印 servlet hello world~ 说明servlet程序已经成功运行。

至此，Servlet的入门案例就已经完成，大家可以按照上面的步骤进行练习了。

4.3 执行流程

Servlet程序已经能正常运行，但是我们需要思考个问题：我们并没有创建ServletDemo1类的对象，也没有调用对象中的service方法，为什么在控制台就打印了servlet hello world~这句话呢？

要想回答上述问题，我们就需要对Servlet的执行流程进行一个学习。



- 浏览器发出 http://localhost:8080/web-demo/demo1 请求，从请求中可以解析出三部分内容，分别是 localhost:8080、web-demo、demo1
 - 根据 localhost:8080 可以找到要访问的 Tomcat Web 服务器
 - 根据 web-demo 可以找到部署在 Tomcat 服务器上的 web-demo 项目
 - 根据 demo1 可以找到要访问的是项目中的哪个 Servlet 类，根据 @WebServlet 后面的值进行匹配
- 找到 ServletDemo1 这个类后，Tomcat Web 服务器就会为 ServletDemo1 这个类创建一个对象，然后调用对象中的 service 方法
 - ServletDemo1 实现了 Servlet 接口，所以类中必然会重写 service 方法供 Tomcat Web 服务器进行调用

- service方法中有ServletRequest和ServletResponse两个参数，ServletRequest封装的是请求数据，ServletResponse封装的是响应数据，后期我们可以通过这两个参数实现前后端的数据交互

小结

介绍完Servlet的执行流程，需要大家掌握两个问题：

1. Servlet由谁创建？Servlet方法由谁调用？

Servlet由web服务器创建，Servlet方法由web服务器调用

2. 服务器怎么知道Servlet中一定有service方法？

因为我们自定义的Servlet，必须实现Servlet接口并复写其方法，而Servlet接口中有service方法

4.4 生命周期

介绍完Servlet的执行流程后，我们知道Servlet是由Tomcat Web服务器帮我们创建的。

接下来咱们再来思考一个问题：**Tomcat什么时候创建的Servlet对象？**

要想回答上述问题，我们就需要对Servlet的生命周期进行一个学习。

- 生命周期：对象的生命周期指一个对象从被创建到被销毁的整个过程。
- Servlet运行在Servlet容器（web服务器）中，其生命周期由容器来管理，分为4个阶段：
 1. **加载和实例化**：默认情况下，当Servlet第一次被访问时，由容器创建Servlet对象

默认情况，Servlet会在第一次访问被容器创建，但是如果创建Servlet比较耗时的话，那么第一个访问的人等待的时间就比较长，用户的体验就比较差，那么我们能不能把Servlet的创建放到服务器启动的时候来创建，具体如何来配置？

```
@WebServlet(urlPatterns = "/demo1", loadOnStartup = 1)
loadOnStartup的取值有两类情况
(1) 负整数：第一次访问时创建Servlet对象
(2) 0或正整数：服务器启动时创建Servlet对象，数字越小优先级越高
```

- 2. **初始化**：在Servlet实例化之后，容器将调用Servlet的**init()**方法初始化这个对象，完成一些如加载配置文件、创建连接等初始化的工作。该方法只调用一次
- 3. **请求处理**：每次请求Servlet时，Servlet容器都会调用Servlet的**service()**方法对请求进行处理
- 4. **服务终止**：当需要释放内存或者容器关闭时，容器就会调用Servlet实例的**destroy()**方法完成资源的释放。在**destroy()**方法调用之后，容器会释放这个Servlet实例，该实例随后会被Java的垃圾收集器所回收

- 通过案例演示下上述的生命周期

```
package com.itheima.web;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import java.io.IOException;
/**
 * Servlet生命周期方法
 */
@WebServlet(urlPatterns = "/demo2", loadOnStartup = 1)
public class ServletDemo2 implements Servlet {

    /**
     * 初始化方法
     * 1. 调用时机：默认情况下，Servlet被第一次访问时，调用
     *      * loadOnStartup：默认为-1，修改为0或者正整数，则会在服务器启动的时候，调用
     * 2. 调用次数：1次
     * @param config
     * @throws ServletException
     */
    public void init(ServletConfig config) throws ServletException {
        System.out.println("init...");
    }

    /**
     * 提供服务
     * 1. 调用时机：每一次Servlet被访问时，调用
     * 2. 调用次数：多次
     * @param req
     * @param res
     * @throws ServletException
     * @throws IOException
     */
    public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        System.out.println("servlet hello world~");
    }
}
```

```

/**
 * 销毁方法
 * 1.调用时机：内存释放或者服务器关闭的时候，Servlet对象会被销毁，调用
 * 2.调用次数：1次
 */
public void destroy() {
    System.out.println("destroy...");
}
public ServletConfig getServletConfig() {
    return null;
}

public String getServletInfo() {
    return null;
}

}

```

注意：如何才能让Servlet中的destroy方法被执行？

The screenshot shows a terminal window within an IDE. The title bar says "Terminal: Local". The command entered is "mvn tomcat7:run". The output shows the Maven build process for a project named "org.example:web-demo". It includes logs for scanning projects, building the web-demo version 1.0-SNAPSHOT, and running the Tomcat 7 plugin. It also shows resource copying and compilation steps.

```

D:\workspace\web-demo>mvn tomcat7:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.example:web-demo >-----
[INFO] Building web-demo 1.0-SNAPSHOT
[INFO] -----[ war ]-----
[INFO]
[INFO] >>> tomcat7-maven-plugin:2.2:run (default-cli) > process-classes @ web-d
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ web-demo
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ web-demo ---

```

在Terminal命令行中，先使用mvn tomcat7:run启动，然后再使用ctrl+c关闭tomcat

小结

这节中需要掌握的内容是：

1. Servlet对象在什么时候被创建的？

默认是第一次访问的时候被创建，可以使用@WebServlet(urlPatterns = "/demo2", loadOnStartup = 1)的loadOnStartup 修改成在服务器启动的时候创建。

2. Servlet生命周期中涉及到的三个方法，这三个方法是什么？什么时候被调用？调用几次？

涉及到三个方法，分别是 init()、service()、destroy()

init方法在Servlet对象被创建的时候执行，只执行1次

service方法在Servlet被访问的时候调用，每访问1次就调用1次

destroy方法在Servlet对象被销毁的时候调用，只执行1次

4.5 方法介绍

Servlet中总共有5个方法，我们已经介绍过其中的三个，剩下的两个方法作用分别是什么？

我们先来回顾下前面讲的三个方法，分别是：

- 初始化方法，在Servlet被创建时执行，只执行一次

```
void init(ServletConfig config)
```

- 提供服务方法，每次Servlet被访问，都会调用该方法

```
void service(ServletRequest req, ServletResponse res)
```

- 销毁方法，当Servlet被销毁时，调用该方法。在内存释放或服务器关闭时销毁Servlet

```
void destroy()
```

剩下的两个方法是：

- 获取Servlet信息

```
String getServletInfo()
//该方法用来返回Servlet的相关信息，没有什么太大的用处，一般我们返回一个空字符串即可
public String getServletInfo() {
    return "";
}
```

- 获取ServletConfig对象

```
ServletConfig getServletConfig()
```

ServletConfig对象，在init方法的参数中有，而Tomcat Web服务器在创建Servlet对象的时候会调用init方法，必定会传入一个ServletConfig对象，我们只需要将服务器传过来的ServletConfig进行返回即可。具体如何操作？

```
package com.itheima.web;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import java.io.IOException;

/**
 * Servlet方法介绍
 */
@WebServlet(urlPatterns = "/demo3", loadOnStartup = 1)
public class ServletDemo3 implements Servlet {

    private ServletConfig servletConfig;
    /**
     * 初始化方法
     * 1. 调用时机：默认情况下，Servlet被第一次访问时，调用
     *      * loadOnStartup：默认为-1，修改为0或者正整数，则会在服务器启动的时候，调用
     * 2. 调用次数：1次
     * @param config
     * @throws ServletException
     */
    public void init(ServletConfig config) throws ServletException {
        this.servletConfig = config;
        System.out.println("init...");
    }
    public ServletConfig getServletConfig() {
        return servletConfig;
    }

    /**
     * 提供服务
     * 1. 调用时机：每一次Servlet被访问时，调用
     * 2. 调用次数：多次
     * @param req
     * @param res
     * @throws ServletException
     * @throws IOException
     */
    public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {
        System.out.println("servlet hello world~");
    }

    /**
     * 销毁方法
     * 1. 调用时机：内存释放或者服务器关闭的时候，Servlet对象会被销毁，调用
     * 2. 调用次数：1次
     */
}
```

```

public void destroy() {
    System.out.println("destroy...");
}

public String getServletInfo() {
    return "";
}

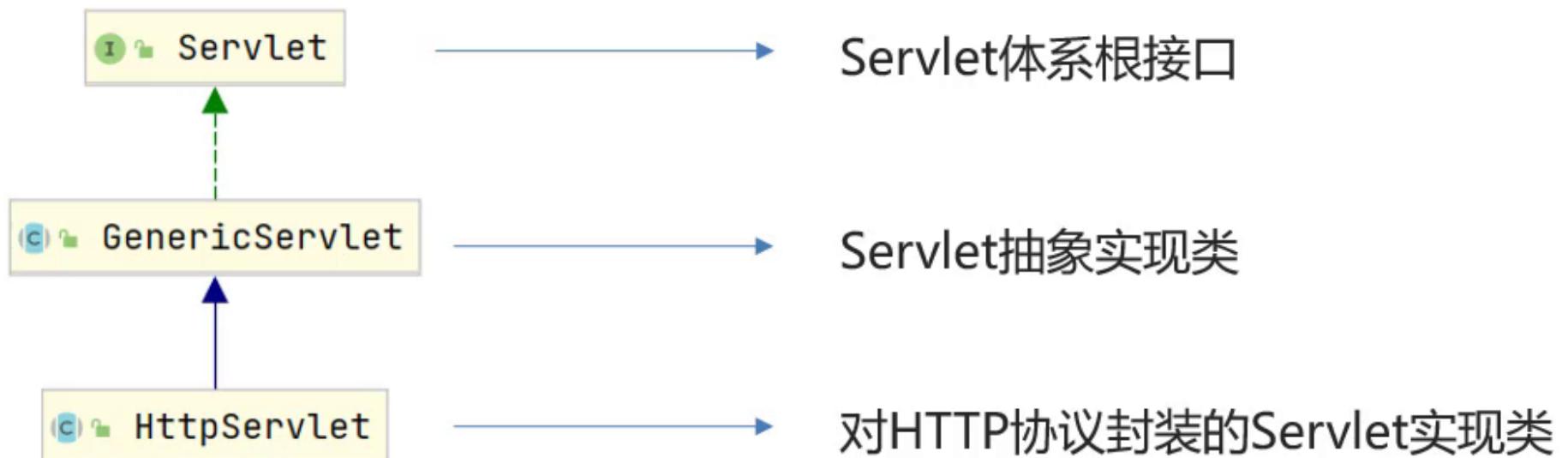
```

getServletInfo()和getServletConfig()这两个方法使用的不是很多，大家了解下。

4.6 体系结构

通过上面的学习，我们知道要想编写一个Servlet就必须要实现Servlet接口，重写接口中的5个方法，虽然已经能完成要求，但是编写起来还是比较麻烦的，因为我们更关注的其实只有service方法，那有没有更简单方式来创建Servlet呢？

要想解决上面的问题，我们需要先对Servlet的体系结构进行下了解：



因为我们将来开发B/S架构的web项目，都是针对HTTP协议，所以我们自定义Servlet，会通过继承**HttpServlet**

具体的编写格式如下：

```

@WebServlet("/demo4")
public class ServletDemo4 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        //TODO GET 请求方式处理逻辑
        System.out.println("get...");
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        //TODO Post 请求方式处理逻辑
        System.out.println("post...");
    }
}

```

- 要想发送一个GET请求，请求该Servlet，只需要通过浏览器发送 `http://localhost:8080/web-demo/demo4`，就能看到doGet方法被执行了
- 要想发送一个POST请求，请求该Servlet，单单通过浏览器是无法实现的，这个时候就需要编写一个form表单来发送请求，在webapp下创建一个 `a.html` 页面，内容如下：

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form action="/web-demo/demo4" method="post">
        <input name="username"/><input type="submit"/>
    </form>
</body>
</html>

```

启动测试，即可看到doPost方法被执行了。

Servlet的简化编写就介绍完了，接着需要思考两个问题：

1. `HttpServlet`中为什么要根据请求方式的不同，调用不同的方法？
2. 如何调用？

针对问题一，我们需要回顾之前的知识点**前端发送GET和POST请求的时候，参数的位置不一致，GET请求参数在请求行中，POST请求参数在请求体中**，为了能处理不同的请求方式，我们得在service方法中进行判断，然后写不同的业务处理，这样能实现，但是每个Servlet类中都将有相似的代码，针对这个问题，有什么可以优化的策略么？

```
package com.itheima.web;

import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/demo5")
public class ServletDemo5 implements Servlet {

    public void init(ServletConfig config) throws ServletException {
    }

    public ServletConfig getServletConfig() {
        return null;
    }

    public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        //如何调用？
        //获取请求方式，根据不同的请求方式进行不同的业务处理
        HttpServletRequest request = (HttpServletRequest)req;
        //1. 获取请求方式
        String method = request.getMethod();
        //2. 判断
        if("GET".equals(method)){
            // get方式的处理逻辑
        }else if("POST".equals(method)){
            // post方式的处理逻辑
        }
    }

    public String getServletInfo() {
        return null;
    }

    public void destroy() {
    }
}
```

要解决上述问题，我们可以对Servlet接口进行继承封装，来简化代码开发。

```
package com.itheima.web;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;

public class MyHttpServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
    }

    public ServletConfig getServletConfig() {
        return null;
    }

    public void service(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
        HttpServletRequest request = (HttpServletRequest)req;
        //1. 获取请求方式
        String method = request.getMethod();
        //2. 判断
        if("GET".equals(method)){
            // get方式的处理逻辑
            doGet(req,res);
        }else if("POST".equals(method)){
            // post方式的处理逻辑
            doPost(req,res);
        }
    }
}
```

```

}

protected void doPost(ServletRequest req, ServletResponse res) {
}

protected void doGet(ServletRequest req, ServletResponse res) {
}

public String getServletInfo() {
    return null;
}

public void destroy() {
}

}

```

有了MyHttpServlet这个类，以后我们再编写Servlet类的时候，只需要继承MyHttpServlet，重写父类中的doGet和doPost方法，就可以用来处理GET和POST请求的业务逻辑。接下来，可以把ServletDemo5代码进行改造

```

@WebServlet("/demo5")
public class ServletDemo5 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {
        System.out.println("get...");
    }

    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
        System.out.println("post...");
    }
}

```

将来页面发送的是GET请求，则会进入到doGet方法中进行执行，如果是POST请求，则进入到doPost方法。这样代码在编写的时候就相对来说更加简单快捷。

类似MyHttpServlet这样的类Servlet中已经为我们提供好了，就是HttpServlet，翻开源码，大家可以搜索service()方法，你会发现HttpServlet做的事更多，不仅可以处理GET和POST还可以处理其他五种请求方式。

```

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < lastModified) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }
    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);
    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);
    } else if (method.equals(METHOD_PUT)) {
        doPut(req, resp);
    } else if (method.equals(METHOD_DELETE)) {
        doDelete(req, resp);
    }
}

```

```

} else if (method.equals(METHOD_OPTIONS)) {
    doOptions(req, resp);

} else if (method.equals(METHOD_TRACE)) {
    doTrace(req, resp);

} else {
    //
    // Note that this means NO servlet supports whatever
    // method was requested, anywhere on this server.
    //

    String errMsg = lStrings.getString("http.method_not_implemented");
    Object[] errArgs = new Object[1];
    errArgs[0] = method;
    errMsg = MessageFormat.format(errMsg, errArgs);

    resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
}
}

```

小结

通过这一节的学习，要掌握：

1. HttpServlet的使用步骤

继承HttpServlet

重写doGet和doPost方法

2. HttpServlet原理

获取请求方式，并根据不同的请求方式，调用不同的doXXX方法

4.7 urlPattern配置

Servlet类编写好后，要想被访问到，就需要配置其访问路径（urlPattern）

- 一个Servlet，可以配置多个urlPattern

`@WebServlet(urlPatterns = {"/demo1", "/demo2"})`

```

package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

/**
 * urlPattern: 一个Servlet可以配置多个访问路径
 */
@WebServlet(urlPatterns = {"/demo7", "/demo8"})
public class ServletDemo7 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo7 get...");
    }

    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}

```

在浏览器上输入 `http://localhost:8080/web-demo/demo7`, `http://localhost:8080/web-demo/demo8` 这两个地址都能访问到ServletDemo7的doGet方法。

- urlPattern配置规则

- 精确匹配

- 配置路径: `@WebServlet("/user/select")`
- 访问路径: `localhost:8080/web-demo/user/select`

```
/*
 * UrlPattern:
 * * 精确匹配
 */
@WebServlet(urlPatterns = "/user/select")
public class ServletDemo8 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo8 get...");
    }
    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

访问路径 `http://localhost:8080/web-demo/user/select`

- 目录匹配

- 配置路径: `@WebServlet("/user/*")`
- 访问路径: `localhost:8080/web-demo/user/aaa`
`localhost:8080/web-demo/user/bbb`

```
package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

/*
 * UrlPattern:
 * * 目录匹配: /user/*
 */
@WebServlet(urlPatterns = "/user/*")
public class ServletDemo9 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo9 get...");
    }
    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

访问路径 `http://localhost:8080/web-demo/user/任意`

思考:

1. 访问路径 `http://localhost:8080/web-demo/user` 是否能访问到 `demo9` 的 `doGet` 方法?
2. 访问路径 `http://localhost:8080/web-demo/user/a/b` 是否能访问到 `demo9` 的 `doGet` 方法?
3. 访问路径 `http://localhost:8080/web-demo/user/select` 是否能访问到 `demo9` 还是 `demo8` 的 `doGet` 方法?

答案是: 能、能、`demo8`, 进而我们可以得到的结论是 `/user/*` 中的 `*` 代表的是零或多个层级访问目录同时精确匹配优先级要高于目录匹配。

- 扩展名匹配

- 配置路径: `@WebServlet("*.do")`

- 访问路径:

localhost:8080/web-demo/aaa.do

localhost:8080/web-demo/bbb.do

```
package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

/**
 * UrlPattern:
 * * 扩展名匹配: *.do
 */
@WebServlet(urlPatterns = "*.do")
public class ServletDemo10 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo10 get...");
    }
    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

访问路径 `http://localhost:8080/web-demo/任意.do`

注意:

1. 如果路径配置的不是扩展名, 那么在路径的前面就必须加 /, 否则会报错

```
Caused by: java.lang.IllegalArgumentException: Invalid <url-pattern> user/* in servlet mapping
    at org.apache.catalina.core.StandardContext.addServletMapping(StandardContext.java:3245)
    at org.apache.catalina.core.StandardContext.addServletMapping(StandardContext.java:3220)
    at org.apache.catalina.deploy.WebXml.configureContext(WebXml.java:1367)
    at org.apache.catalina.startup.ContextConfig.webConfig(ContextConfig.java:1346)
    at org.apache.catalina.startup.ContextConfig.configureStart(ContextConfig.java:878)
    at org.apache.catalina.startup.ContextConfig.lifecycleEvent(ContextConfig.java:376)
    at org.apache.catalina.util.LifecycleSupport.fireLifecycleEvent(LifecycleSupport.java:119)
    at org.apache.catalina.util.LifecycleBase.fireLifecycleEvent(LifecycleBase.java:90)
    at org.apache.catalina.core.StandardContext.startInternal(StandardContext.java:5322)
    at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:150)
    ... 6 more
```

2. 如果路径配置的是 *.do, 那么在 *.do 的前面不能加 /, 否则会报错

```
Caused by: java.lang.IllegalArgumentException: Invalid <url-pattern> /*.do in servlet mapping
    at org.apache.catalina.core.StandardContext.addServletMapping(StandardContext.java:3245)
    at org.apache.catalina.core.StandardContext.addServletMapping(StandardContext.java:3220)
    at org.apache.catalina.deploy.WebXml.configureContext(WebXml.java:1367)
    at org.apache.catalina.startup.ContextConfig.webConfig(ContextConfig.java:1346)
    at org.apache.catalina.startup.ContextConfig.configureStart(ContextConfig.java:878)
    at org.apache.catalina.startup.ContextConfig.lifecycleEvent(ContextConfig.java:376)
    at org.apache.catalina.util.LifecycleSupport.fireLifecycleEvent(LifecycleSupport.java:119)
    at org.apache.catalina.util.LifecycleBase.fireLifecycleEvent(LifecycleBase.java:90)
    at org.apache.catalina.core.StandardContext.startInternal(StandardContext.java:5322)
    at org.apache.catalina.util.LifecycleBase.start(LifecycleBase.java:150)
    ... 6 more
```

- 任意匹配

- 配置路径: `@WebServlet("/")`

- 访问路径:

localhost:8080/web-demo/hehe
localhost:8080/web-demo/haha

```
package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

/**
 * UrlPattern:
 * * * 任意匹配: /
 */
@WebServlet(urlPatterns = "/")
public class ServletDemo11 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo11 get...");
    }
    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

访问路径 `http://localhost:8080/demo-web/任意`

```
package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

/**
 * UrlPattern:
 * * * 任意匹配: /*
 */
@WebServlet(urlPatterns = "/*")
public class ServletDemo12 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {

        System.out.println("demo12 get...");
    }
    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

访问路径 `http://localhost:8080/demo-web/任意`

注意: / 和 /* 的区别?

1. 当我们的项目中的Servlet配置了 "/", 会覆盖掉tomcat中的DefaultServlet, 当其他的url-pattern都匹配不上时都会走这个Servlet
2. 当我们的项目中配置了"/*", 意味着匹配任意访问路径
3. DefaultServlet是用来处理静态资源, 如果配置了"/"会把默认的覆盖掉, 就会引发请求静态资源的时候没有走默认的而是走了自定义的Servlet类, 最终导致静态资源不能被访问

小结

1. urlPattern总共有四种配置方式, 分别是精确匹配、目录匹配、扩展名匹配、任意匹配
2. 五种配置的优先级为 精确匹配 > 目录匹配 > 扩展名匹配 > /* > / , 无需记, 以最终运行结果为准。

4.8 XML配置

前面对应Servlet的配置, 我们都使用的是`@WebServlet`, 这个是Servlet从3.0版本后开始支持注解配置, 3.0版本前只支持XML配置文件的配置方法。

对于XML的配置步骤有两步:

- 编写Servlet类

```
package com.itheima.web;

import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

public class ServletDemo13 extends MyHttpServlet {

    @Override
    protected void doGet(ServletRequest req, ServletResponse res) {
        System.out.println("demo13 get...");
    }

    @Override
    protected void doPost(ServletRequest req, ServletResponse res) {
    }
}
```

- 在web.xml中配置该Servlet

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

    <!--
        Servlet 全类名
    -->
    <servlet>
        <!-- servlet的名称, 名字任意-->
        <servlet-name>demo13</servlet-name>
        <!--servlet的类全名-->
        <servlet-class>com.itheima.web.ServletDemo13</servlet-class>
    </servlet>

    <!--
        Servlet 访问路径
    -->
    <servlet-mapping>
        <!-- servlet的名称, 要和上面的名称一致-->
        <servlet-name>demo13</servlet-name>
        <!-- servlet的访问路径-->
        <url-pattern>/demo13</url-pattern>
    </servlet-mapping>
</web-app>
```

这种配置方式和注解比起来，确认麻烦很多，所以建议大家使用注解来开发。但是大家要认识上面这种配置方式，因为并不是所有的项目都是基于注解开发的。