

Maven

笔记位置: http://heavy_code_industry.gitee.io/code_heavy_industry/pro002-maven/

Maven

第一章 Maven概述

第一节 为什么要学习Maven

- 1、Maven作为依赖管理工具
- 2、Maven作为构建管理工具
- 3、结论

第二节 什么是Maven

- 1、构建
- 2、依赖
- 3、Maven的工作机制

第二章 Maven核心程序解压和配置

第一节 Maven核心程序解压和配置

- 1、Maven官网地址
- 2、解压Maven核心程序
- 3、指定本地仓库
- 4、配置阿里云提供的镜像仓库
- 5、配置Maven工程的基础JDK版本

第二节 配置环境变量

- 1、检查 JAVA_HOME配置是否正确
- 2、配置MAVEN_HOME
- 3、配置PATH
- 4、验证

第三章 使用Maven——命令行环境

第一节 实验一：根据坐标创建Maven工程

- 1、Maven核心概念：坐标
- 2、实验操作
 - ①创建目录作为后面操作的工作空间
 - ②在工作空间目录下打开命令行窗口
 - ③使用命令生成Maven工程
 - ④调整
 - ⑤自动生成的pom.xml解读
- 3、Maven核心概念：POM
 - ①含义
 - ②模型化思想
 - ③对应的配置文件
- 4、Maven核心概念：约定的目录结构
 - ①各个目录的作用
 - ②约定目录结构的意义
 - ③约定大于配置

第二节 实验二：在Maven工程中编写代码

- 1、主体程序

2、测试程序

第三节 实验三：执行Maven的构建命令

- 1、要求
- 2、清理操作
- 3、编译操作
- 4、测试操作
- 5、打包操作
- 6、安装操作

第四节 实验四：创建Maven版的Web工程

- 1、说明
- 2、操作
- 3、生成的pom.xml文件
- 4、生成的Web工程的目录结构
- 5、创建Servlet
 - ①在main目录下创建Java目录
 - ②在java目录下创建Servlet类所在的包的目录
 - ③在包下创建Servlet类
 - ④在Web.xml中注册Servlet
- 6、在index.jsp页面编写超链接
- 7、编译
- 8、配置对servlet-api.jar包的依赖
- 9、将Web工程打包为war包
- 10、将war包部署到Tomcat上运行

第五节 实验五：让Web工程依赖于Java工程

- 1、注意
- 2、操作
- 3、在Web工程中，编写测试代码
 - ①补充创建目录
 - ②确认web工程依赖了junit
 - ③创建测试类
- 4、执行Maven命令
 - ①测试命令
 - ②打包命令
 - ③查看当前Web工程所依赖的jar包的列表
 - ④以树形结构查看当前Web工程的依赖信息

第六节 实验六：测试依赖范围

- 1、依赖范围
 - ①compile和test的对比
 - ②compile和provided的对比
 - ③结论
- 2、测试
 - ①验证compile范围对main目录有效
 - ②验证test范围对main目录无效
 - ③验证test和provided范围不参与服务器部署
 - ④验证provided范围对测试程序有效

第七节 实验七：测试依赖的传递性

- 1、依赖的传递性
 - ①概念
 - ②传递的原则

2、使用compile范围依赖spring-core

3、验证test和provided范围不能传递

第八节 实验八：测试依赖的排除

1、概念

2、配置方式

3、测试

第九节 实验九：继承

1、概念

2、作用

3、举例

4、操作

①创建父工程

②创建模块工程

③查看被添加新内容的父工程pom.xml

④解读子工程的pom.xml

⑤在父工程中配置依赖的统一管理

⑥在子工程中引用哪些被父工程管理的依赖

⑦在父工程中升级依赖信息的版本

⑧在父工程中声明自定义属性

5、实际意义

第十节 实验十：聚合

1、聚合本身的含义

2、Maven中的聚合

3、好处

4、聚合的配置

5、依赖循环问题

第四章 使用Maven——IDEA环境

第一节 创建父工程

1、创建Project

2、开启自动导入

第二节 配置Maven信息

第三节 创建Java模块工程

①创建新的**maven模块**，选择parent为父工程

②执行项目的方法

第四节 创建Web模块工程

1、创建模块

2、修改打包方式

3、Web设定

4、借助IDEA生成web.xml

5、设置Web资源根目录

第五节 其他操作

1、在IDEA中执行Maven命令

①直接输入

②手动输入

2、在IDEA中查看某个模块的依赖信息

3、工程导入

①来自版本控制系统

②来自工程目录

4、模块导入	
①情景再现	
②导入Java类型模块	
③导入Web类型模块	
第五章 其他核心概念	
1、生命周期	
①作用	
②三个生命周期	
③特点	
2、插件和目标	
①插件	
②目标	
3、仓库	
第六章 单一架构案例	
第一节 创建工程，引入依赖	
第二节 搭建环境：持久化层	
第三节 搭建环境：事务控制	
第四节 搭建环境：表述层	
第五节 搭建环境：辅助功能	
第六节 业务功能：登录	
第七节 业务功能：显示奏折列表	
第八节 业务功能：显示奏折详情	
第九节 业务功能：批复奏折	
第十节 业务功能：登录检查	
第十一节 打包部署	
附录	

第一章 Maven概述

第一节 为什么要学习Maven

1、Maven作为依赖管理工具

1. jar包的规模

1. 随着我们使用越来越多的框架，或者框架封装程度越来越高，项目中使用jar包也越来越多，项目中，一个模块里面用到上百个jar包是非常正常的。
2. 比如只用到SpringBoot、SpringCloud框架中的三个功能，Nacos服务注册发现、Web框架环境、图模板技术Thymeleaf需要106个jar包。而使用Maven**只需要三个配置依赖**

```
<!--Nacos服务注册发现启动器-->
<dependency>
    <group>com.alibaba.cloud</group>
    <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
```

```
</dependency>
<!--web启动器依赖-->
<dependency>
    <group>org.springframework.boot</group>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!--视图模板技术thymeleaf-->
<dependency>
    <group>org.springframework.boot</group>
    <artifactId>spring-boot-starter-
thymeleaf</artifactId>
</dependency>
```

2. jar包的来源

1. 这个jar包所属技术的官网。官网通常是英文界面，网站的结构又不尽相同，甚至找到下载链接还发现需要通过特殊的工具下载。
2. 第三方网站提供下载，问题是不规范，在使用中出现各种问题，jar包的名称、jar包的版本、jar包内的具体细节
3. 而使用Maven后，依赖对应的jar包能够**自动下载，方便，快捷又规范**

3. jar包之间的依赖关系

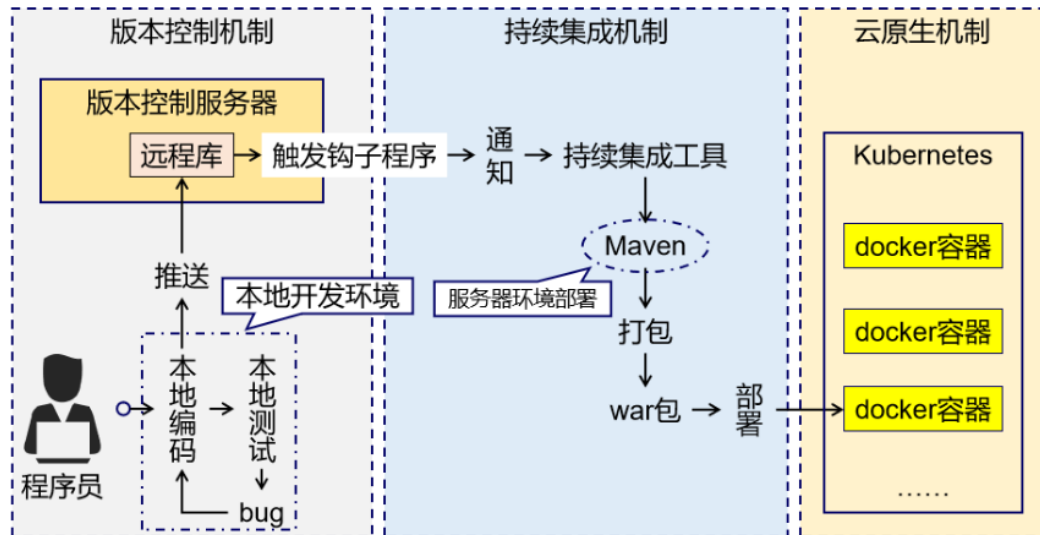
1. 框架中使用的 jar 包，不仅数量庞大，而且彼此之间存在错综复杂的依赖关系。依赖关系的复杂程度，已经上升到了完全不能靠人力手动解决的程度。另外，jar 包之间有可能产生冲突。进一步增加了我们在 jar 包使用过程中的难度。
2. 而实际上 jar 包之间的依赖关系是普遍存在的，如果要由程序员手动梳理无疑会增加极高的学习成本，而这些工作又对实现业务功能毫无帮助。
3. 而使用 Maven 则几乎不需要管理这些关系，极个别的地方调整一下即可，极大的减轻了我们的工作量。

2、Maven作为构建管理工具

1. 没有注意过的构建

1. 可以不使用 Maven，但是**构建必须要做**。当我们使用 IDEA 进行开发时，构建是 IDEA 替我们做的。

2. 脱离IDE环境仍需要构建



3、结论

1. 管理规模庞大的jar包，需要专门的工具
2. 脱离IDE环境执行构建操作，需要专门工具

第二节 什么是Maven

Maven是Apache软件基金会组织维护的一款专门为Java项目提供**构建和依赖管理**支持的工具。

1、构建

Java项目开发过程中，构建指的是使用【原材料生产产品】的过程

1. 原材料

1. Java源代码
2. 基于HTML和Thymeleaf文件
3. 图片
4. 配置文件
5. 等等……

2. 产品

1. 一个可以在服务器上运行的项目

构建过程包含的主要环节

- **清理**：删除上一次构建的结果，为下一次构建做好准备
- **编译**：Java源程序编译成*.class字节码文件
- **测试**：运行提前准备好的测试程序
- **报告**：针对刚才测试的结果生成一个全面的信息
- **打包**

- Java工程：jar包

- Web工程：war包
- 安装：把一个Maven工程经过打包操作生成jar包或war包存入Maven仓库
- 部署
 - 部署jar包：把一个jar包部署到Nexus私服服务器上
 - 部署war包：借助相关Maven插件（例如cargo），将war包部署到Tomcat服务器上

2、依赖

如果A工程里面用到了B工程的类、接口、配置文件等这样的资源，那么我们就可以说A依赖B

配置管理中要解决的具体问题：

- jar包的下载：使用Maven之后，jar包会规范的远程仓库下载到本地
- jar包之间的依赖：通过依赖的传递性自动完成
- jar阿伯之间的冲突：通过对依赖的配置进行调整，让某些jar包不会被导入

3、Maven的工作机制

第二章 Maven核心程序解压和配置

第一节 Maven核心程序解压和配置

1、Maven官网地址

首页 Maven-Welcome to Apache Maven <https://maven.apache.org/>

下载页面 Maven-Download Apache Maven <https://maven.apache.org/download.cgi>

下载页面

Files

Maven is distributed in several formats for your convenience. Simply pick a ready-made binary distribution archive and follow the [installation instructions](#). Use a source archive if you intend to build Maven yourself.

In order to guard against corrupted downloads/installations, it is highly recommended to [verify the signature](#) of the release bundles against the public [KEYS](#) used by the Apache Maven developers.

	Link	Checksums	Signature
Binary tar.gz archive	apache-maven-3.8.4-bin.tar.gz	apache-maven-3.8.4-bin.tar.gz.sha512	apache-maven-3.8.4-bin.tar.gz.asc
Binary zip archive	apache-maven-3.8.4-bin.zip	apache-maven-3.8.4-bin.zip.sha512	apache-maven-3.8.4-bin.zip.asc
Source tar.gz archive	apache-maven-3.8.4-src.tar.gz	apache-maven-3.8.4-src.tar.gz.sha512	apache-maven-3.8.4-src.tar.gz.asc
Source zip archive	apache-maven-3.8.4-src.zip	apache-maven-3.8.4-src.zip.sha512	apache-maven-3.8.4-src.zip.asc

具体下载地址 <https://dlcdn.apache.org/maven/maven-3/3.8.4/binaries/apache-maven-3.8.4-bin.zip>

2、解压Maven核心程序

核心程序压缩包：apache-maven-3.8.4-bin.zip，**解压到非中文，没有空格的目录**

在解压目录中，我们需要着重关注 Maven 的核心配置文件：**conf/settings.xml**

3、指定本地仓库

本地仓库默认值：用户家目录/.m2/repository。由于本地仓库的默认位置是在用户的家目录下，而家目录往往是在 C 盘，也就是系统盘。将来 Maven 仓库中 jar 包越来越多，仓库体积越来越大，可能会拖慢 C 盘运行速度，影响系统性能。所以建议将 Maven 的本地仓库放在其他盘符下。配置方式如下

```
<!-- localRepository
| The path to the local repository maven will use to store artifacts.
|
| Default: ${user.home}/.m2/repository
<localRepository>/path/to/local/repo</localRepository>
-->
<localRepository>D:\maven-repository</localRepository>
```

本地仓库这个目录，我们手动创建一个空的目录即可

记住：一定要把localRepository标签从注释中拿出来

注意：本地仓库本身也需要一个使用**非中文、没有空格的目录**

4、配置阿里云提供的镜像仓库

Maven 下载 jar 包默认访问境外的中央仓库，而国外网站速度很慢。改成阿里云提供的镜像仓库，**访问国内网站**，可以让 Maven 下载 jar 包的时候速度更快。配置的方式是：

①将原有的例子配置注释掉

```
<!-- <mirror>
<id>maven-default-http-blocker</id>
<mirrorOf>external:http:*</mirrorOf>
<name>Pseudo repository to mirror external repositories initially
using HTTP.</name>
<url>http://0.0.0.0/</url>
<blocked>true</blocked>
</mirror> -->
```

②加入我们的配置


```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorOf>central</mirrorOf>
  <name>Nexus aliyun</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

5、配置Maven工程的基础JDK版本

如果按照默认配置运行，Java 工程使用的默认 JDK 版本是 1.5，而我们熟悉和常用的是 JDK 1.8 版本。修改配置的方式是：将 profile 标签整个复制到 settings.xml 文件的 profiles 标签内

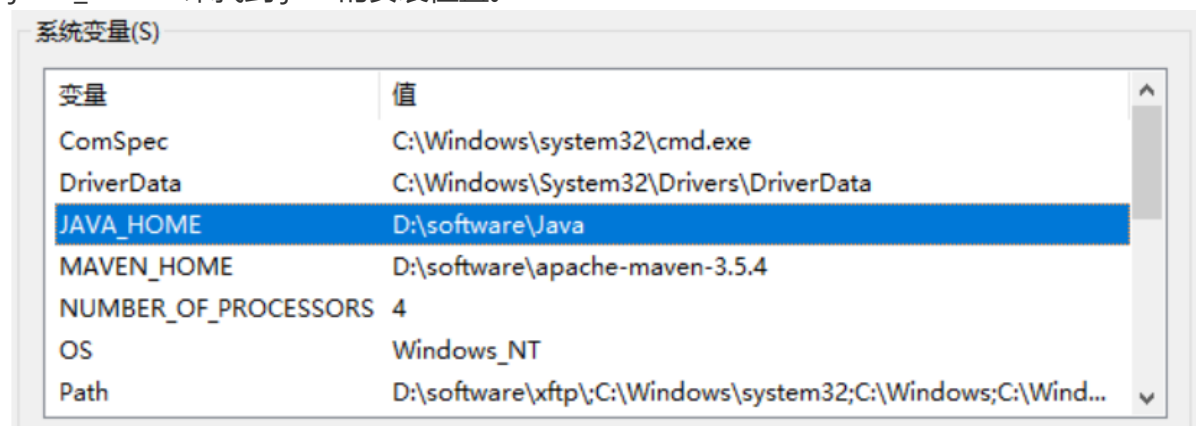
```
<profile>
  <id>jdk-1.8</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>1.8</jdk>
  </activation>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>

    <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
  </properties>
</profile>
```

第二节 配置环境变量

1、检查 JAVA_HOME配置是否正确

Maven 是一个用 Java 语言开发的程序，它必须基于 JDK 来运行，需要通过 JAVA_HOME 来找到 JDK 的安装位置。

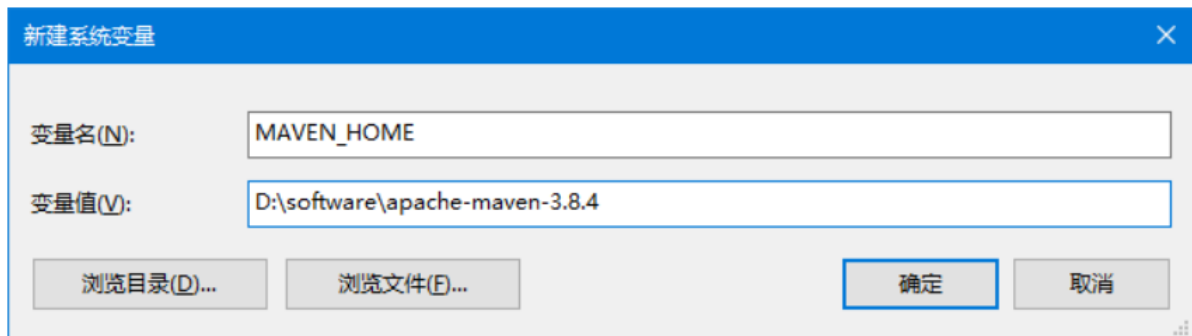


可以使用下面的命令验证

```
C:\Users\Administrator>echo %JAVA_HOME%
D:\software\Java

C:\Users\Administrator>java -version
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)
```

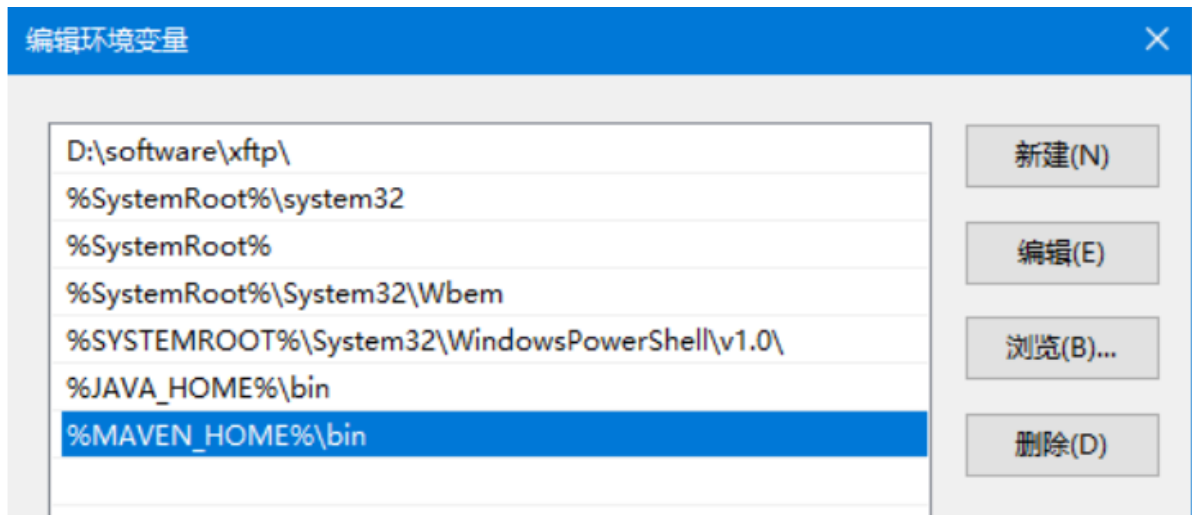
2、配置MAVEN_HOME



TIP

- 配置环境变量的规律
- XXX_HOME通常指的是bin目录的上一级
- PATH指向的是bin目录

3、配置PATH



4、验证

```
C:\Users\Administrator>mvn -v
```

第三章 使用Maven——命令行环境

第一节 实验一：根据坐标城建Maven工程

1、Maven核心概念：坐标

①数学中的坐标：使用x、y、z三个向量作为空间的坐标系，可以在空间中**唯一**定位到一个点

②Maven中的坐标

1. 向量说明

1. **使用三个向量在Maven的仓库中唯一**定位到一个jar包

2. **groupId**——公司或组织的id

3. **artifactId**——一个项目或者项目中的一个模块的id

4. **version**——版本号

2. 三个向量的取值方式

1. **groupId**——公司或组织的域名的倒序，通常会加上项目名称

1. com.atguigu.maven

2. **artifactId**——模块的名称，将来作为Maven工程的工程名

3. **version**——模块的版本号，根据自己需要设定

1. SNAPSHOT表示快照版本，正在迭代过程中，不稳定的版本

2. RELEASE表示正式版本

3. 举例

1. groupId: com.atguigu.maven

2. artifactId: pro01-atguigu-maven

3. version: 1.0-SNAPSHOT

③坐标和仓库中jar包的存储路径之间的对应关系

坐标：

```
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>2.5</version>
```

上面坐标对应的jar包在Maven本地仓库的位置：

Maven本地仓库目录\javax\servlet\servlet-api\2.5\servlet-api-2.5.jar

一定要会根据坐标到本地仓库中找到对应的jar包

2、实验操作

①创建目录作为后面操作的工作空间

例如：D:\maven-workspace\space201026

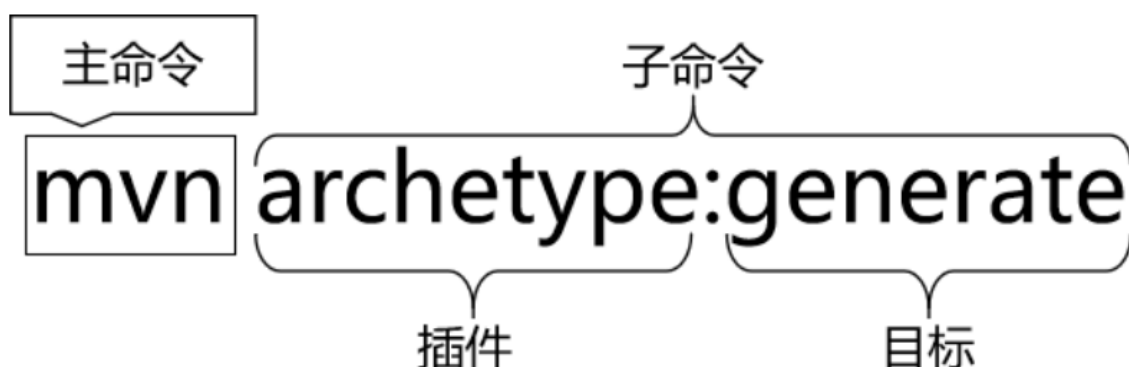
WARNING

1. 此时已经有三个目录

1. Maven核心程序
2. Maven本地仓库
3. 本地工作空间

②在工作空间目录下打开命令行窗口

③使用命令生成Maven工程



运行 `mvn archetype:generate` 命令

1. Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 7: **【直接回车，使用默认值】**
2. Define value for property 'groupId': com.atguigu.maven
3. Define value for property 'artifactId': pro01-maven-java
4. Define value for property 'version' 1.0-SNAPSHOT: : **【直接回车，使用默认值】**
5. Define value for property 'package' com.atguigu.maven: : **【直接回车，使用默认值】**
6. Confirm properties configuration: groupId: com.atguigu.maven artifactId: pro01-maven-java version: 1.0-SNAPSHOT package: com.atguigu.maven Y: : **【直接回车，表示确认。如果前面有输入错误，想要重新输入，则输入 N 再回车。】**

④调整

Maven默认生成的工程，对junit依赖的较低的3.8.1版本，我们可以改成比较合适的4.12版本。自动生成的App.java和AppTest.java可以删除

```
<!-- 依赖信息配置 -->
```

```
<!-- dependencies复数标签：里面包含dependency单数标签 -->
```

```

<dependencies>
  <!--dependency单数标签：配置一个具体的依赖-->
  <dependency>
    <!--通过坐标来依赖其他jar包-->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>

    <!--依赖范围-->
    <scope>test</scope>
  </dependency>
</dependencies>

```

⑤自动生成的pom.xml解读

```

<!--当前Maven工程的坐标-->
<groupId>com.atguigu.maven</groupId>
<artifactId>pro01-maven-java</artifactId>
<version>1.0-SNAPSHOT</version>

<!--当前Maven工程的打包方式，可以选择以下三种值-->
<!--jar:表示这个工程是一个java工程-->
<!--war:表示这个工程是一个web工程-->
<!--pom:表示这个工程是管理其他工程的工程-->
<packaging>jar</packaging>

<name>pro01-maven-java</name>
<url>http://maven.apache.org</url>

<properties>
  <!--工程构建过程中读取源码时使用的字符集-->
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<!--当前工程所依赖的jar包-->
<dependencies>
  <!--使用dependency配置一个具体的依赖-->
  <dependency>

    <!-- 在dependency标签内使用具体的坐标依赖我们需要的一个jar包 -->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <!-- scope标签配置依赖的范围 -->
    <scope>test</scope>
  </dependency>
</dependencies>

```

3、Maven核心概念：POM

①含义

POM: Project Object Model, 项目对象模型。和POM类似的是: DOM (Document Object Model), 文档对象模型。它们都是模型化思想的具体体现。

②模型化思想

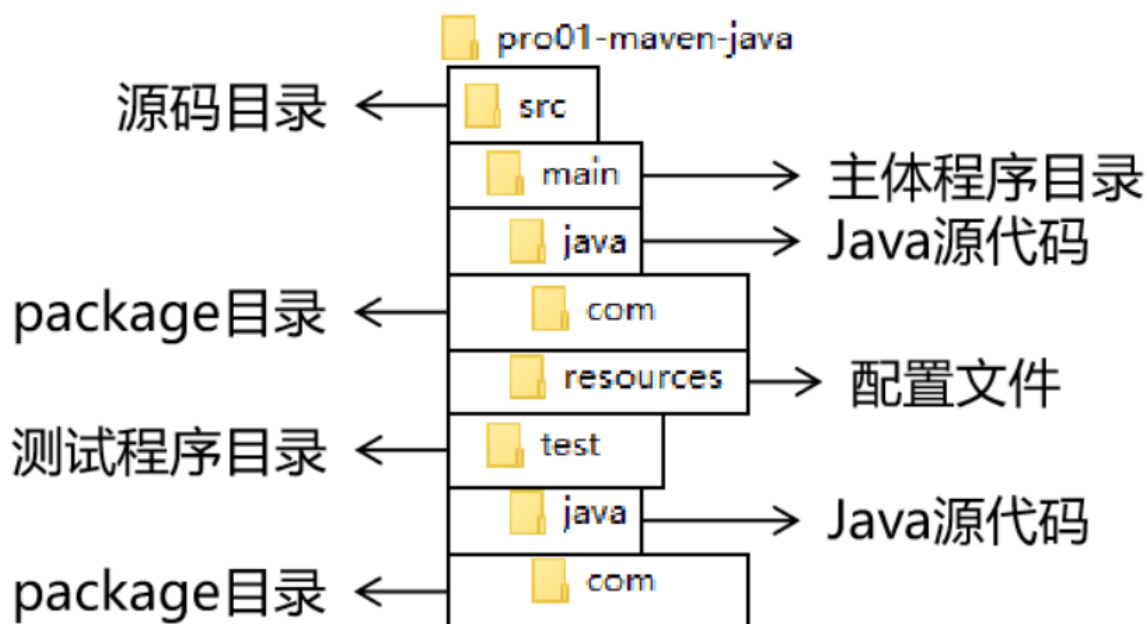
POM表示将工程抽象一个模型, 再用程序中的对象来描述这个模型, 这样我们就可以用程序来管理项目了。我们在开发过程中, 最具体的做法就是将现实生活中的事务抽象为模型, 然后封装模型相关的数据作为一个对象, 这样就可以在程序中计算与现实事物相关的数据

③对应的配置文件

POM理念集中体现在Maven工程根目录下**pom.xml**这个配置文件中, 所以这个pom.xml配置文件就是Maven工程的核心配置文件。其学习Maven就是学这个文件怎么配置, 各个配置有什么用。

4、Maven核心概念：约定的目录结构

①各个目录的作用



另外还有一个target目录专门存放构建操作输出的结果。

②约定目录结构的意义

Maven为了让构建过程能够尽可能自动化完成, 所以必须约定目录结构的作用。例如: Maven执行编译操作, 必须先去找Java源程序目录读取Java源代码, 然后执行编译, 最后把编译结果存放在target目录。

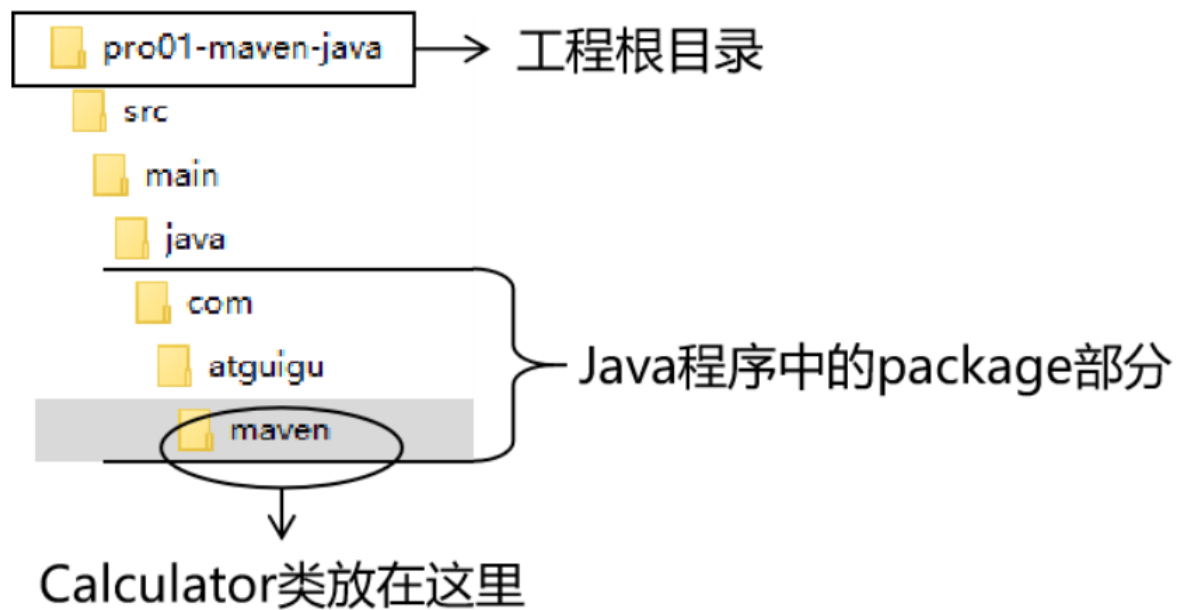
③约定大于配置

Maven对于目录结构这个问题，没有采用配置的方式，而是基于约定，这样让我们在开发郭长城中非常方便，如果每次创建Maven工程后，还需要针对这个目录进行详细的配置，那肯定非常麻烦

目前开发领域的技术发展趋势为：约定大于配置，配置大于代码

第二节 实验二：在Maven工程中编写代码

1、主体程序

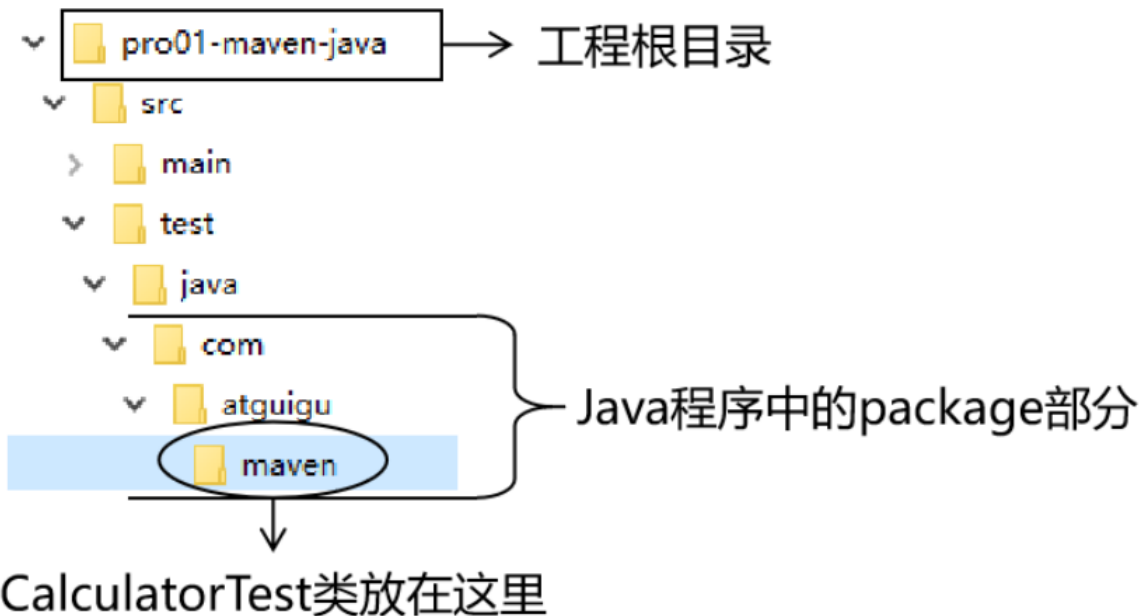


主体程序指的是被测试的程序，同时也是将来在项目中真正要使用的程序

放置在根目录\src\main\java\com\atguigu\maven

```
//例如
package com.atguigu.maven;
public class Calculator{
    public int sum(int i,int j){
        return i+j;
    }
}
```

2、测试程序



```
package com.atquigu.maven;

import org.junit.Test;
import com.atquigu.maven.Calculator;

// 静态导入的效果是将Assert类中的静态资源导入当前类
// 这样一来，在当前类中就可以直接使用Assert类中的静态资源，不需要写类名
import static org.junit.Assert.*;

public class CalculatorTest{

    @Test
    public void testSum(){

        // 1.创建Calculator对象
        Calculator calculator = new Calculator();

        // 2.调用Calculator对象的方法，获取到程序运行实际的结果
        int actualResult = calculator.sum(5, 3);

        // 3.声明一个变量，表示程序运行期待的结果
        int expectedResult = 8;

        // 4.使用断言来判断实际结果和期待结果是否一致
        // 如果一致：测试通过，不会抛出异常
        // 如果不一致：抛出异常，测试失败
        assertEquals(expectedResult, actualResult);

    }

}
```


第三节 实验三：执行Maven的构建命令

1、要求

运行Maven中和构建操作相关命令是，必须进入到pom.xml所在的目录。

如果没有进入该目录，那么会看到一下的错误信息

```
The goal you specified requires a project to execute but there is no
POM in this directory
```

TIP

mvn -v命令和构建操作无关，只要正确配置了PATH，在任何目录下执行都可以。而构建相关的命令要在pom.xml所在的目录下运行——操作哪个工程，就进入这个工程的pom.xml目录。

2、清理操作

mvn clean

效果：删除target目录

3、编译操作

主程序编译：**mvn compile**

测试程序编译：**mvn test-compile**

主体程序编译结果存放目录：target/classes

测试程序编译结果存放目录：target/test-classes

4、测试操作

mvn test

测试的报告存放的目录：target/surefire-reports

5、打包操作

mvn package

打包的结果——jar包，存放的目录：target

6、安装操作

mvn install

```
[INFO] Installing D:\maven-workspace\space201026\pro01-maven-java\target\pro01-maven-java-1.0-SNAPSHOT.jar to D:\maven-rep1026\com\atguigu\maven\pro01-maven-java\1.0-SNAPSHOT\pro01-maven-java-1.0-SNAPSHOT.jar
[INFO] Installing D:\maven-workspace\space201026\pro01-maven-java\pom.xml to D:\maven-rep1026\com\atguigu\maven\pro01-maven-java\1.0-SNAPSHOT\pro01-maven-java-1.0-SNAPSHOT.pom
```

安装的效果是将本地构建过程中生成的jar包存入Maven本地仓库，这个jar包在Maven仓库中的路径是根据**它的坐标生成的**

坐标信息如下

```
<groupId>com.atguigu.maven</groupId>
<artifactId>pro01-maven-java</artifactId>
<version>1.0-SNAPSHOT</version>
```

在Maven仓库中生成的路径为

本地仓库\com\atguigu\maven\pro01-maven-java\1.0-SNAPSHOT\pro01-java-1.0-SNAPSHOT.jar

另外，安装操作还会将pom.xml文件转换为XXX.pom文件一起存入本地仓库。所以我们在Maven的本地仓库中向看一个jar包原始的pom.xml文件时，查看对应XXX.pom文件即可，他们的名字发生了变化，本质上是同一个文件

可以先清理后打包，确保是最新的版本

第四节 实验四：创建Maven版的Web工程

1、说明

使用mvn archetype:generate命令生成Web工程时，需要用专门的archetype。这个专门生成Web工程骨架的archetype可以参照官网看到它的用法

Maven Webapp Archetype

maven-archetype-webapp is an archetype which generates a sample Maven webapp project.

```
1. project
2. |-- pom.xml
3. |-- src
4. |   |-- main
5. |   |   |-- webapp
6. |   |   |   |-- WEB-INF
7. |   |   |   |   |-- web.xml
8. |   |   |   |   |-- index.jsp
```

Usage

To generate a new project from this archetype, type:

```
1. mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
```

例如

```
mvn archetype:generate -
DarchetypeGroupId=org.apache.maven.archetypes -
DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
```

参数 archetypeGroupId、archetypeArtifactId、archetypeVersion 用来指定现在使用的 maven-archetype-webapp 的坐标。

2、操作

注意：如果在一个工程的目录下执行mvn archetype:generate 命令，那么Maven会报错：**不能在一个非pom的工程下再创建其他工程**。所以不要再已经创建好的工程里面再创建新的工程，回到**工作空间根目录**再操作

然后运行生成工程的命令

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
```

下面是操作提示

Define value for property 'groupId': com.atguigu.maven Define value for property 'artifactId': pro02-maven-web Define value for property 'version' 1.0-SNAPSHOT: : 【直接回车，使用默认值】

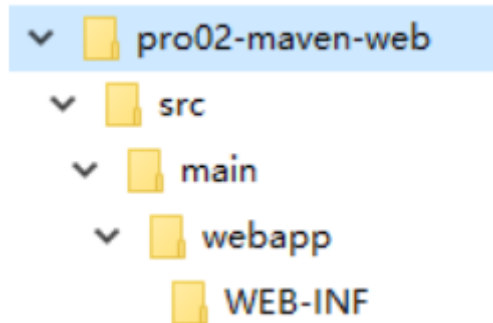
Define value for property 'package' com.atguigu.maven: : 【直接回车，使用默认值】 Confirm properties configuration: groupId: com.atguigu.maven artifactId: pro02-maven-web version: 1.0-SNAPSHOT package: com.atguigu.maven Y: : 【直接回车，表示确认】

3、生成的pom.xml文件

确认打包的方式是war包形式

```
<packaging>war</packaging>
```

4、生成的Web工程的目录结构



webapp目录下有index.jsp

WEB-INF目录下有web.xml

5、创建Servlet

①在main目录下创建Java目录

②在java目录下创建Servlet类所在的包的目录

③在包下创建Servlet类

```
package com.atguigu.maven;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

public class HelloServlet extends HttpServlet{

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {

        response.getWriter().write("hello maven web");

    }

}
```

④在Web.xml中注册Servlet

```
<servlet>
    <servlet-name>helloServlet</servlet-name>
    <servlet-class>com.atguigu.maven.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>helloServlet</servlet-name>
    <url-pattern>/helloServlet</url-pattern>
</servlet-mapping>
```

6、在index.jsp页面编写超链接

```
<html>
<body>
<h2>Hello world!</h2>
<a href="helloServlet">Access Servlet</a>
</body>
</html>
```

7、编译

此时直接执行编译mvn compile命令报错【包不存在等】

报错信息说明：我们的 Web 工程用到了 HttpServlet 这个类，而 HttpServlet 这个类属于 servlet-api.jar 这个 jar 包。此时我们说，Web 工程需要依赖 servlet-api.jar 包。


8、配置对servlet-api.jar包的依赖

对于不知道详细信息的依赖可以到<https://mvnrepository.com/>网站查询。使用关键词搜索，然后在搜索结果列表中选择适合的使用。

找到对应道德jar之后点击版本号直接复制坐标信息。


Found 35709 results

Sort: **relevance** | popular | newest




1. Java Servlet API
[javax.servlet](#) » [javax.servlet-api](#)
Java Servlet API
Last Release on Apr 20, 2018

13,968 usages
GPL CDDL




2. JavaServlet(TM) Specification
[javax.servlet](#) » [servlet-api](#)
JavaServlet(TM) Specification
Last Release on Apr 17, 2008

11,124 usages
GPL GPL CDDL



3. Servlet API
[org.mortbay.jetty](#) » [servlet-api](#)
Servlet API
Last Release on Feb 24, 2010

283 usages
EPL Apache



4. Tomcat Servlet API
[org.apache.tomcat](#) » [tomcat-servlet-api](#)
jakarta.servlet package

263 usages

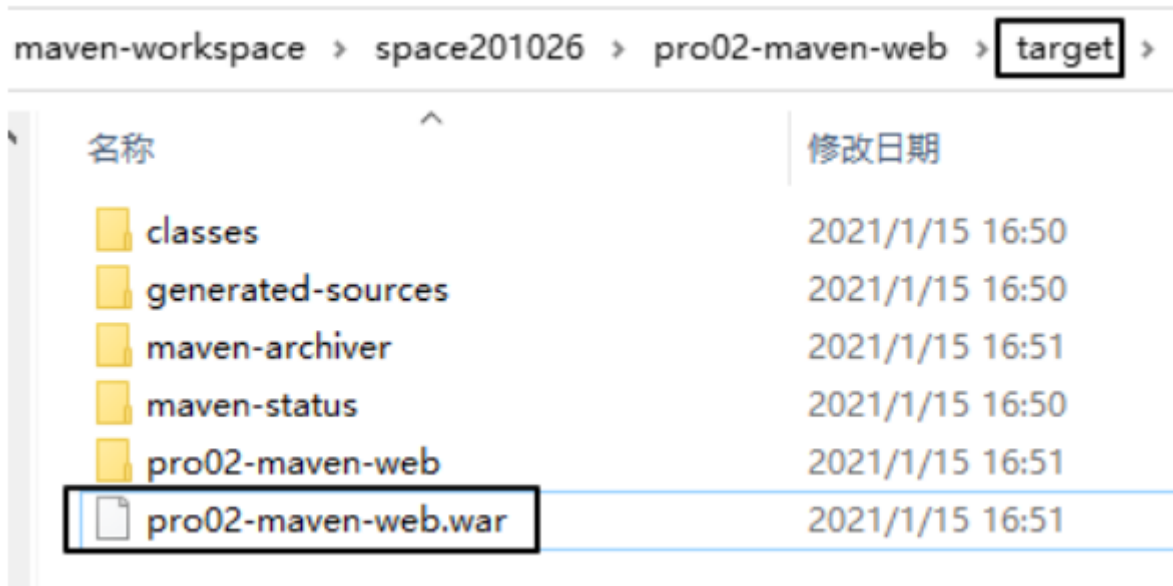
比如，我们找到的 `servlet-api` 的依赖信息：

```
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

这样就可以把上面的信息加入 `pom.xml`。重新执行 `mvn compile` 命令。

9、将Web工程打包为war包

运行 mvn package 命令，生成 war 包的位置如下图所示：



10、将war包部署到Tomcat上运行

将 war 包复制到 Tomcat/webapps 目录下

启动Tomcat

通过浏览器尝试访问：<http://localhost:8080/pro02-maven-web/index.jsp>

第五节 实验五：让Web工程依赖于Java工程

1、注意

明确一个意识：从来只有 Web 工程依赖 Java 工程，没有反过来 Java 工程依赖 Web 工程。本质上来说，Web 工程依赖的 Java 工程其实就是 Web 工程里导入的 jar 包。最终 Java 工程会变成 jar 包，放在 Web 工程的 WEB-INF/lib 目录下。

2、操作

在pro02-maven-web工程的pom.xml中，找到dependencies标签，在dependencies标签配置如下

```
<!-- 配置对Java工程pro01-maven-java的依赖 -->
<!-- 具体的配置方式：在dependency标签内使用坐标实现依赖 -->
<dependency>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>pro01-maven-java</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

3、在Web工程中，编写测试代码

①补充创建目录

pro02-maven-web\src\test\java\com\atguigu\maven

②确认web工程依赖了junit

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

③创建测试类

把Java工程的CalculatorTest.java类复制到pro02-maven-web\src\test\java\com\atguigu\maven目录下

4、执行Maven命令

①测试命令

mvn test

说明：测试操作中会提前自动执行编译操作，测试成功就说明编译也是成功的。

②打包命令

mvn package

通过查看war包内的结构，我们看到被Web工程依赖的Java工程确实是会变成Web工程的WEB-INF/lib目录下的jar包。

③查看当前Web工程所依赖的jar包的列表

mvn dependency:list 【所显示出来的依赖也就是当前工程可以使用的】

TIP

1. [INFO] The following files have been resolved:
2. [INFO] org.hamcrest:hamcrest-core:jar:1.3:test
3. [INFO] javax.servlet:javax.servlet-api:jar:3.1.0:provided
4. [INFO] com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
5. [INFO] junit:junit:jar:4.12:test

说明：javax.servlet:javax.servlet-api:jar:3.1.0:provided 格式显示的是一个jar包的坐标信息。格式是：

TIP

1. groupId:artifactId:打包方式:version:依赖的范围

这样的格式虽然和我们 XML 配置文件中坐标的格式不同，但是本质上还是坐标信息，大家需要能够认识这样的格式，将来从 Maven 命令的日志或错误信息中看到这样格式的信息，就能够识别出来这是坐标。进而根据坐标到Maven 仓库找到对应的jar包，用这样的方式解决我们遇到的报错的情况。

④以树形结构查看当前Web工程的依赖信息

mvn dependency:tree 【可以查看依赖之间的依赖关系】

TIP

1. [INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT
2. [INFO] +- junit:junit:jar:4.12:test
3. [INFO] | - org.hamcrest:hamcrest-core:jar:1.3:test
4. [INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
5. [INFO] - com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile

我们在 pom.xml 中并没有依赖 hamcrest-core，但是它却被加入了我们依赖的列表。原因是：junit 依赖了hamcrest-core，然后基于依赖的传递性，hamcrest-core 被传递到我们的工程了。

第六节 实验六：测试依赖范围

1、依赖范围

标签的位置：dependencies/dependency/scope

标签的可选值：compile、test、provided、system、runtime、import

①compile和test的对比

	main目录 (空间)	test目录 (空间)	开发过程 (时间)	部署到服务器 (时间)
compile	有效	有效	有效	有效
test	无效	有效	有效	无效

②compile和provided的对比

	main目录 (空间)	test目录 (空间)	开发过程 (时间)	部署到服务器 (时间)
compile	有效	有效	有效	有效
provided	有效	有效	有效	无效

③结论

compile：通常使用的**第三方框架的jar包**这样在项目中运行时真正要用到的jar包都是以compile范围进行依赖的。比如SSM框架所需要的jar包

test：测试过程中使用的jar包，以test范围依赖进来。比如junit

provided：在开发过程中需要用到的“服务器上的jar包”通常以provided范围依赖进来。比如servlet-api、jsp-api。而这个范围的jar包之所以不参与部署，不放进war包，就是避免和服务上已有的同类jar包产生冲突，同时减轻服务器的负担。“服务器上已经有了，你就别带啦！”

2、测试

①验证compile范围对main目录有效

TIP

main目录下的类：HelloServlet 使用compile范围导入的依赖：pro01-atguigu-maven

验证：使用compile范围导入的依赖对main目录下的类来说是有效的

有效：HelloServlet 能够使用 pro01-atguigu-maven 工程中的 Calculator 类

验证方式：在 HelloServlet 类中导入 Calculator 类，然后编译就说明有效。

②验证test范围对main目录无效

测试方法：在主程序中导入org.junit.Test这个注解，然后执行编译

具体操作：在pro01-maven-java\src\main\java\com\atguigu\maven目录下修改Calculator.java

```
package com.atguigu.maven;

import org.junit.Test;

public class Calculator {

    public int sum(int i, int j){
        return i + j;
    }

}
```

执行Maven编译命令

```
[ERROR] /D:/maven-workspace/space201026/pro01-maven-
java/src/main/java/com/atguigu/maven/Calculator.java:[3,17] 程序包
org.junit不存在
```

③验证test和provided范围不参与服务器部署

其实就是验证：通过compile范围依赖的jar包会放入war包，通过test范围依赖的jar包不会放入war包。

④验证provided范围对测试程序有效

测试方式是在pro02-maven-web的测试程序中加入servlet-api.jar包中的类。

修改：pro02-maven-

web\src\test*\java\com\atguigu\maven*CalculatorTest.java**

```
package com.atguigu.maven;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

import org.junit.Test;
import com.atguigu.maven.Calculator;

// 静态导入的效果是将Assert类中的静态资源导入当前类
// 这样一来，在当前类中就可以直接使用Assert类中的静态资源，不需要写类名
import static org.junit.Assert.*;

public class CalculatorTest{

    @Test
    public void testSum(){

        // 1.创建Calculator对象
        Calculator calculator = new Calculator();

        // 2.调用Calculator对象的方法，获取到程序运行实际的结果
        int actualResult = calculator.sum(5, 3);

        // 3.声明一个变量，表示程序运行期待的结果
        int expectedResult = 8;

        // 4.使用断言来判断实际结果和期待结果是否一致
        // 如果一致：测试通过，不会抛出异常
        // 如果不一致：抛出异常，测试失败
        assertEquals(expectedResult, actualResult);

    }

}
```

然后运行Maven的编译命令：mvn compile

然后看到编译成功。

第七节 实验七：测试依赖的传递性

1、依赖的传递性

①概念

A依赖B，B依赖C，那么在A没有配置对C的依赖的情况下，A里面能不能直接使用C？

②传递的原则

在A依赖B，B依赖C的前提下，C是否能够传递到A，取决于B依赖C使用的依赖范围

- **B依赖C时使用compile范围：可以传递**
- **B依赖C时使用test或provided范围：不可以传递**，所以需要这样的jar包，就必须在需要的地方明确配置依赖才可以

2、使用compile范围依赖spring-core

测试方式：让pro01-maven-java工程依赖spring-core

具体操作：编译pro01-maven-java工程目录下的pom.xml

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
```

使用mvn dependency:tree命令查看效果：

TIP

[INFO] com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT

[INFO] +- junit:junit:jar:4.12:test

[INFO] | - org.hamcrest:hamcrest-core:jar:1.3:test

[INFO] - org.springframework:spring-core:jar:4.0.0.RELEASE:compile

[INFO] - commons-logging:commons-logging:jar:1.1.1:compile

还可以在Web工程中，使用mvn dependency:tree命令查看效果（需要重新讲pro01-maven-java安装到仓库）

TIP

[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT

[INFO] +- junit:junit:jar:4.12:test

[INFO] | - org.hamcrest:hamcrest-core:jar:1.3:test

[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided

[INFO] - com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile

[INFO] - org.springframework:spring-core:jar:4.0.0.RELEASE:compile

[INFO] - commons-logging:commons-logging:jar:1.1.1:compile

3、验证test和provided范围不能传递

从上面的例子已经能够看到，pro01-maven-java 依赖了 junit，但是在 pro02-maven-web 工程中查看依赖树的时候并没有看到 junit。

要验证 provided 范围不能传递，可以在 pro01-maven-java 工程中加入 servlet-api 的依赖。

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

效果和之前一样：

[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT

[INFO] +- junit:junit:jar:4.12:test

[INFO] | - org.hamcrest:hamcrest-core:jar:1.3:test

[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided

[INFO] - com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile

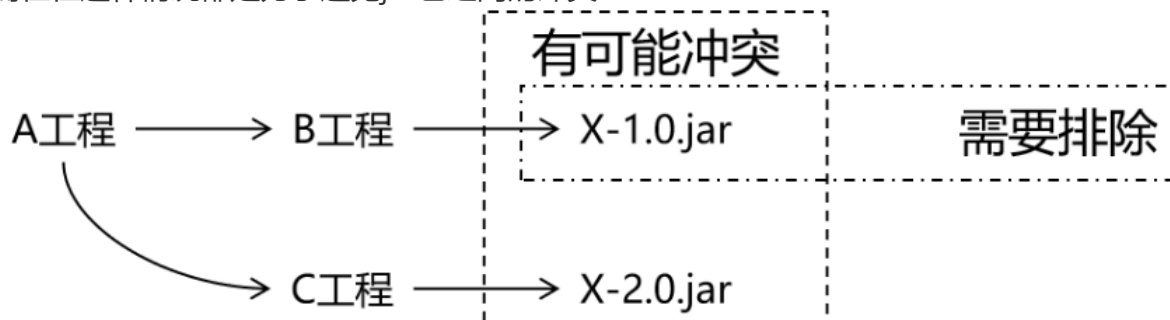
[INFO] - org.springframework:spring-core:jar:4.0.0.RELEASE:compile

[INFO] - commons-logging:commons-logging:jar:1.1.1:compile

第八节 实验八：测试依赖的排除

1、概念

当A依赖B，B依赖C而且C可以传递到A的时候，A不想要C，需要在A里面把C排除掉。而往往这种情况都是为了避免jar包之间的冲突



所以配置依赖的排除其实就是阻止某些jar包的传递。因为这样的jar包传递过来会和其他jar包冲突。

2、配置方式

```
<dependency>
  <groupId>com.atguigu.maven</groupId>
  <artifactId>pro01-maven-java</artifactId>
  <version>1.0-SNAPSHOT</version>
  <scope>compile</scope>

  <!--使用excludes标签配置依赖的排除-->
  <excludes>
    <!--在exclude标签配置一个具体的排除-->
    <exclude>
      <!--指定要排除的依赖的坐标（不需要写version）-->
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclude>
  </excludes>
</dependency>
```

3、测试

测试的方式：在pro02-maven-web工程中配置对commons-logging的排除

运行mvn dependency:tree

发现在spring-core下面没有commons-logging

第九节 实验九：继承

1、概念

Maven工程之间，A工程继承B工程

- B工程——父工程
- A工程——子工程

本质上是A工程的pom.xml中的配置继承了B工程中pom.xml的配置

2、作用

在父工程中统一管理项目中的依赖信息，具体来说管理依赖信息的版本

它的背景是：

- 对一个比较大型的项目进行了模块拆分
- 一个project下面，创建了很多个model
- 每一个model都需要配置自己的依赖信息

它背后的需求是

- 在每一个 module 中各自维护各自的依赖信息很容易发生出入，不易统一管理。
- 使用同一个框架内的不同 jar 包，它们应该是同一个版本，所以整个项目中使用的框架版本需要统一。
- 使用框架时所需要的 jar 包组合（或者说依赖信息组合）需要经过长期摸索和反复调试，最终确定一个可用组合。这个耗费很大精力总结出来的方案不应该在新的项目中重新摸索。

通过在父工程中为整个项目维护依赖信息的组合既保证了整个项目使用规范，准备的jar包；又能够将以往的经验沉淀下来，节约时间和精力

3、举例

一个工程中依赖多个spring的jar包

```
[INFO] +- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO] | \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.springframework:spring-beans:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-context:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-
expression:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-aop:jar:4.0.0.RELEASE:compile
[INFO] | \- aopalliance:aopalliance:jar:1.0:compile
```

使用 Spring 时要求所有 Spring 自己的 jar 包版本必须一致。为了能够对这些 jar 包的版本进行统一管理，我们使用继承这个机制，将所有版本信息统一在父工程中进行管理。

4、操作

①创建父工程

创建的过程和前面创建pro01-maven-java一样

工程名称: pro03-maven-parent

工程创建好之后，**需要修改打包方式【pom】**

```
<groupId>com.atguigu.maven</groupId>
<artifactId>pro03-maven-parent</artifactId>
<version>1.0-SNAPSHOT</version>

<!--当前工程作为父工程，它要去管理子工程，所以打包方式为pom-->
```

只有打包方式为 pom 的 Maven 工程能够管理其他 Maven 工程。打包方式为 pom 的 Maven 工程中不写业务代码，它是专门管理其他 Maven 工程的工程。

②创建模块工程

模块工程类似于IDEA的module，所以需要进入**pro03-maven-parent工程的根目录**，然后运行**mvn archetype:generate**命令来创建模块工程

③查看被添加新内容的父工程pom.xml

下面modules和module标签是聚合功能的配置

```
<modules>
  <module>pro04-maven-module</module>
  <module>pro05-maven-module</module>
  <module>pro06-maven-module</module>
</modules>
```

④解读子工程的pom.xml


```

<!--使用parent标签指定当前工程的父工程-->
<parent>
    <!--子工程的坐标-->
    <groupId>com.atguigu.maven</groupId>
    <artifactId>pro03-maven-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>

<!--子工程的坐标>
<!-- 如果子工程坐标中的groupId和version与父工程一致，那么可以省略 -->
<!-- <groupId>com.atguigu.maven</groupId> -->
<artifactId>pro04-maven-module</artifactId>
<!-- <version>1.0-SNAPSHOT</version> -->

```

⑤在父工程中配置依赖的统一管理

注意：即使在父工程配置了对依赖的管理，子工程需要使用具体哪一个依赖还是需要明确配置，只不过可以不用版本号，由父工程统一管理

- 对于已经在父工程进行了管理的依赖，子工程引用时可以不写version
- 子工程省略version标签，子工程采纳的就是父工程管理的版本
- 子工程写了version标签
 - 子工程的version和父工程的一样，最终还是采纳这个版本
 - 子工程的version和父工程的不一样，那么子工程管理的版本覆盖了父工程管理的版本，并最终采纳子工程的版本

```

<!-- 使用dependencyManagement标签配置对依赖的管理 -->
<!-- 被管理的依赖并没有真正被引入到工程 -->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
    </dependencies>
</dependencyManagement>

```

```

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>4.0.0.RELEASE</version>
        </dependency>
    </dependencies>
</dependencyManagement>

```

⑥在子工程中引用哪些被父工程管理的依赖

关键点：省略版本号

```

<!-- 子工程引用父工程中的依赖信息时，可以把版本号去掉。 -->
<!-- 把版本号去掉就表示子工程中这个依赖的版本由父工程决定。 -->
<!-- 具体来说是由父工程的dependencyManagement来决定。 -->
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-expression</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
    </dependency>
</dependencies>

```

⑦在父工程中升级依赖信息的版本

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>4.1.4.RELEASE</version>
</dependency>
```

然后载子工程中运行mvn dependency:list, 效果如下

TIP

```
[INFO] org.springframework:spring-aop:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-core:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-context:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-beans:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-expression:jar:4.1.4.RELEASE:compile
```

⑧在父工程中声明自定义属性

```
<!--通过自定义属性，统一指定Spring的版本-->
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

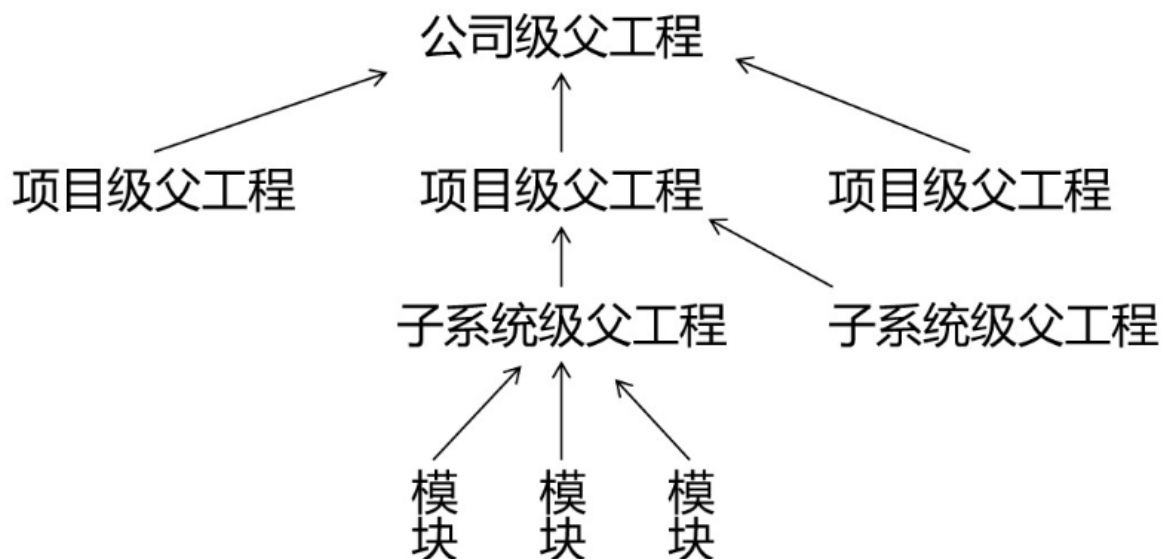
  <!--自定义标签，维护Spring版本数据-->
  <atguigu.spring.version>4.3.6.RELEASE</atguigu.spring.version>
</properties>
```

在需要的地方使用 `${}` 的形式类引用自定义的属性名

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-org</artifactId>
  <version>${atguigu.spring.version}</version>
</dependency>
```

真正实现“一处修改，处处生效”

5、实际意义



编写一套符合要求，开发各种功能都能正常工作的依赖组合并不容易，如果公司里面已经有人总结了成熟的组合方案，那么再开发新项目时，如果不使用原有的积累，而是重新摸索，会浪费大量的时间，为了提高效率，我们可以使用工程继承的机制，让成熟的依赖组合能够保留下来。

第十节 实验十：聚合

1、聚合本身的含义

部分组成整体

2、Maven中的聚合

使用一个总的“总工程”将各个模块工程汇集起来，作为一个整体对应完整的项目

- 项目：整体
- 模块：部分

概念的对应关系

- 从继承关系角度来看
 - 父工程
 - 子工程
- 从聚合关系角度来看
 - 总工程
 - 模块工程

3、好处

- 一键执行Maven命令：很多构建命令都可以在总工程中一键执行

以mvn install命令为例：Maven要求有父工程时先安装父工程；有依赖的工程时，先安装被依赖的工程。我们自己考虑这些规则会很麻烦。**但是工程聚合之后，在总工程执行mvn install可以一键完成安装，而且会自动按照正确的顺序执行。**

- 配置聚合之后，各个模块工程会在总工程中展示一个列表，让项目中的各个模块一目了然

4、聚合的配置

在总工程中配置modules即可

```
<modules>
  <module>pro04-maven-module</module>
  <module>pro05-maven-module</module>
  <module>pro06-maven-module</module>
</modules>
```

5、依赖循环问题

如果A工程依赖B工程，B工程依赖C工程，C工程又反过来依赖A工程，那么在执行构建操作时会报下面的错误

DANGER

```
[ERROR] [ERROR] The projects in the reactor contain a cyclic
reference:
```

这个错误的含义是：循环引用。

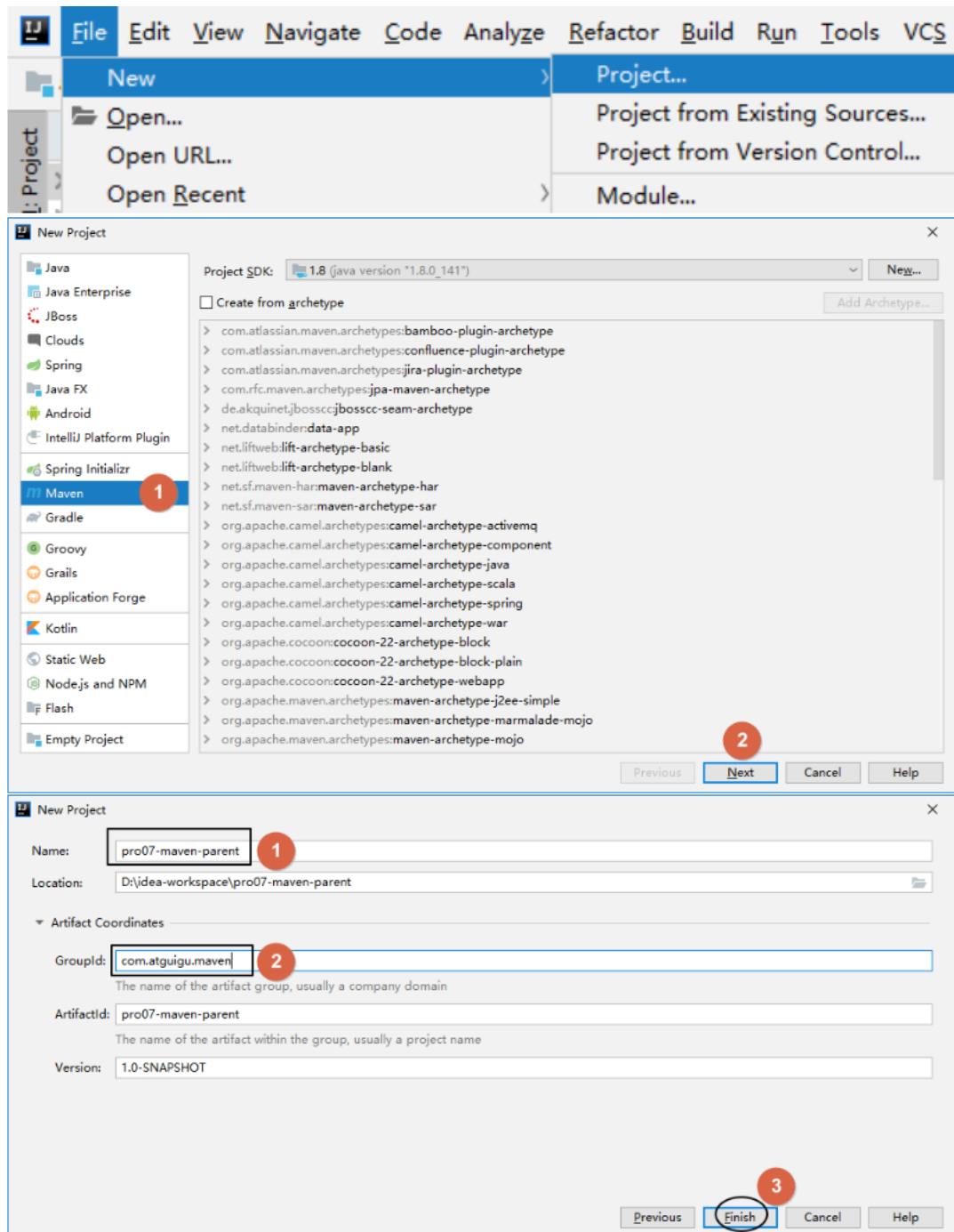
第四章 使用Maven——IDEA环境

第一节 创建父工程

1、创建Project

1. 新建Maven项目

2. 输入工程坐标信息

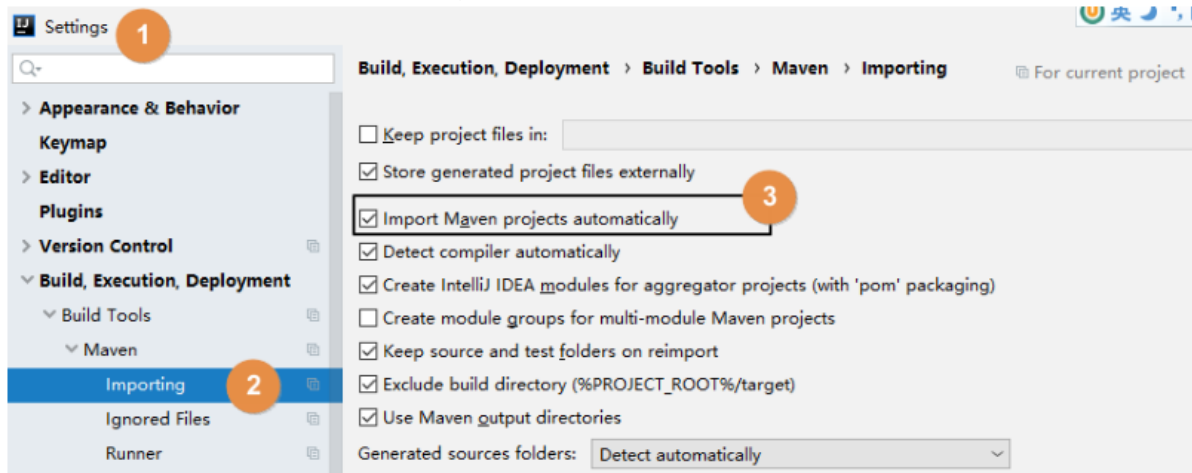


2、开启自动导入

创建Project后，IDEA会自动弹出【Maven projects need to imported】的提示，选择【Enable Auto-Import】意思是启动自动导入。

这个自动导入**一定要开启**，因为Project，Module新创建或pom.xml每次修改时都应该让IDEA重新加载Maven信息。这对Maven目录结构认定，Java源程序编译，依赖jar包的导入都有非常关键的影响。

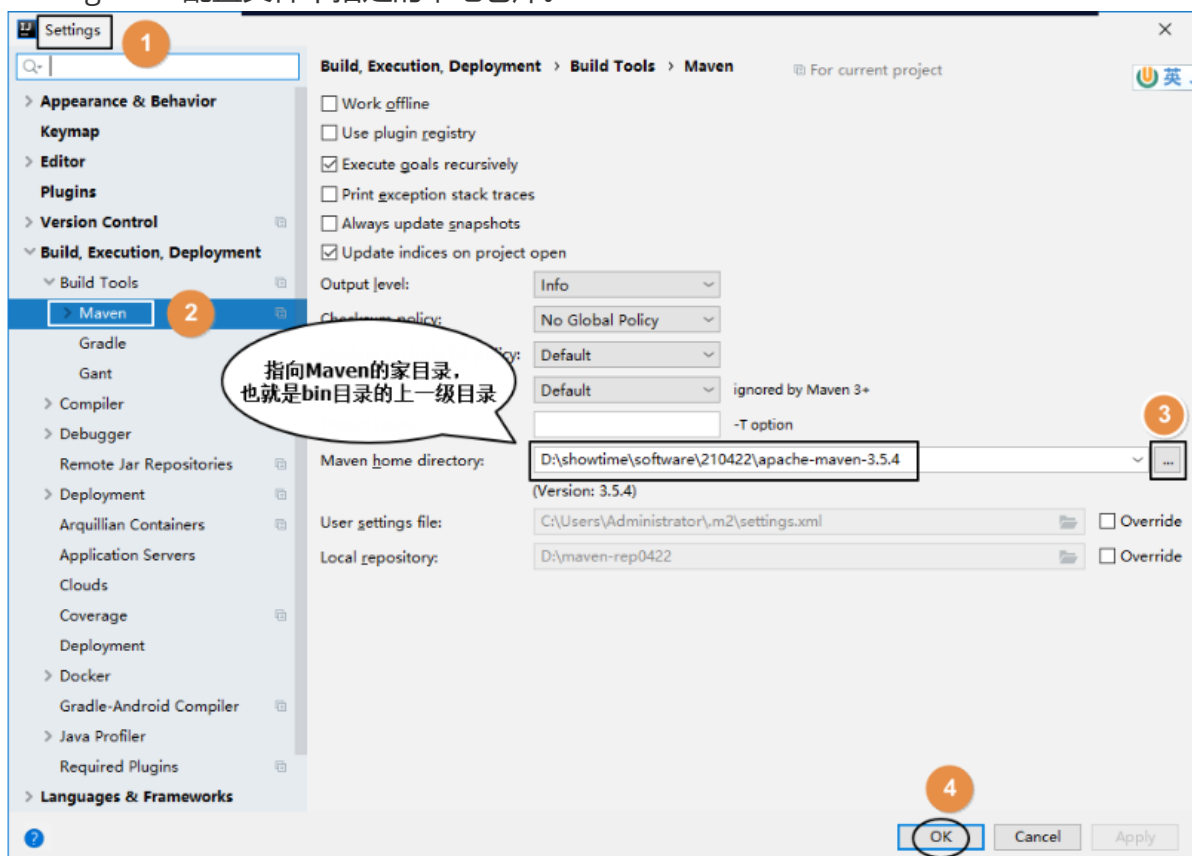
另外也可以通过IDEA的Settings设置来启动；



第二节 配置Maven信息

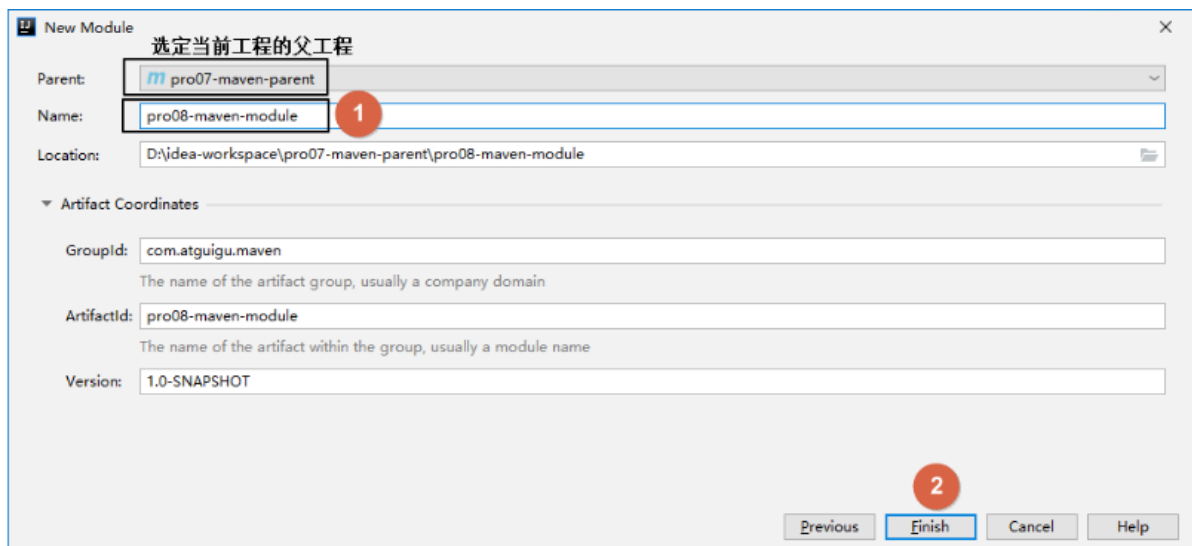
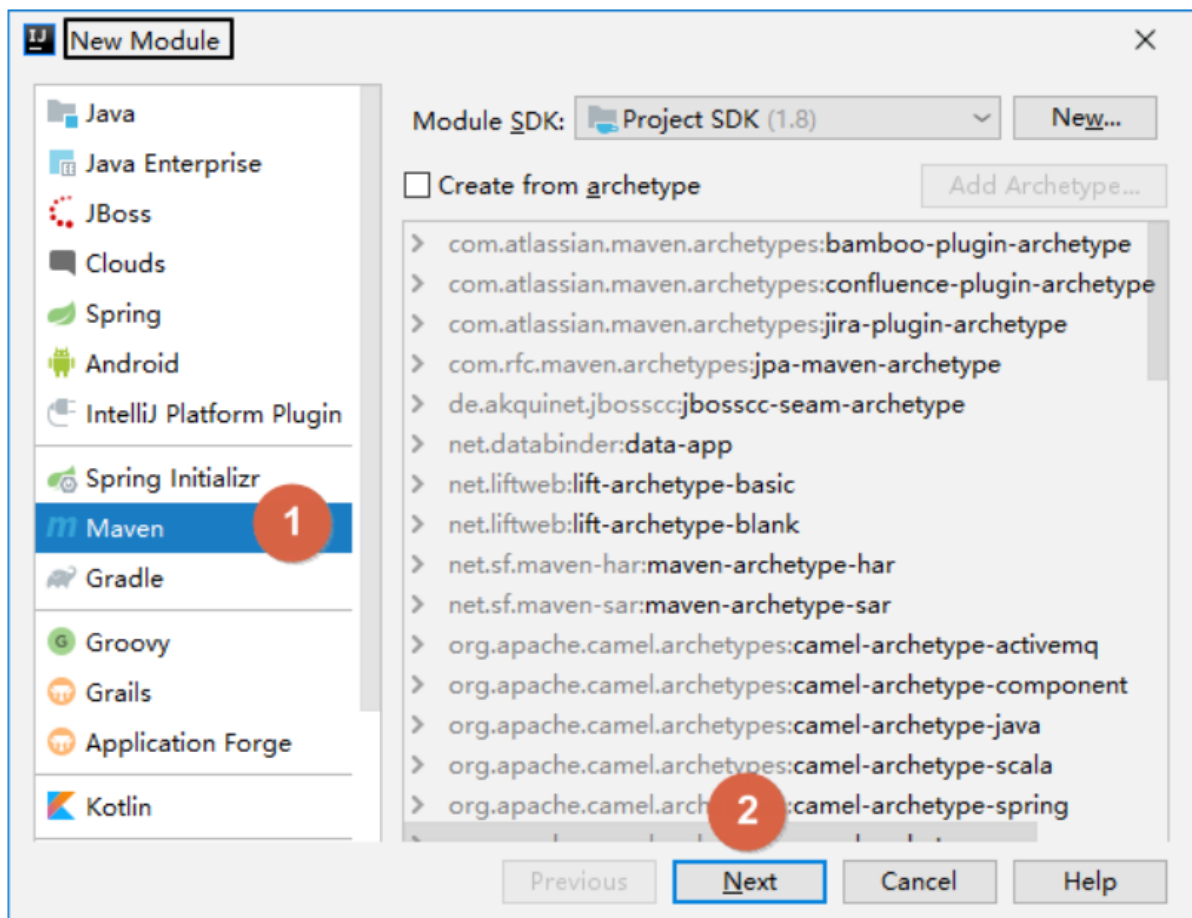
每次创建 Project 后都需要设置 Maven 家目录位置，否则 IDEA 将使用内置的 Maven 核心程序（不稳定）并使用默认的本地仓库位置。这样一来，我们在命令行操作过程中已下载好的 jar 包就白下载了，默认的本地仓库通常在 C 盘，还影响系统运行。

配置之后，IDEA 会根据我们在这里指定的 Maven 家目录自动识别到我们在 settings.xml 配置文件中指定的本地仓库。



第三节 创建Java模块工程

①创建新的maven模块，选择parent为父工程



②执行项目的方法

1. IDEA插件Maven选项卡，在对应的工程展开，选择生命周期（Lifecycle）——直接双击命令
2. IDEA插件Maven选项卡，在对应的工程展开，选择插件（Plugs）——双击对应的命令

3. IDEA插件Maven选项卡，点击最上面的**M标志**【执行Maven目标】——输入命令并选择正确的工程然后回车执行

1. 跳过测试的命令——**mvn clean install -Dmaven.test.skip=true**

第四节 创建Web模块工程

1、创建模块

按照前面的同样操作创建模块，**此时**这个模块其实还是一个**Java模块**。

2、修改打包方式

Web 模块将来打包当然应该是 **war** 包

```
<packaging>war</packaging>
```

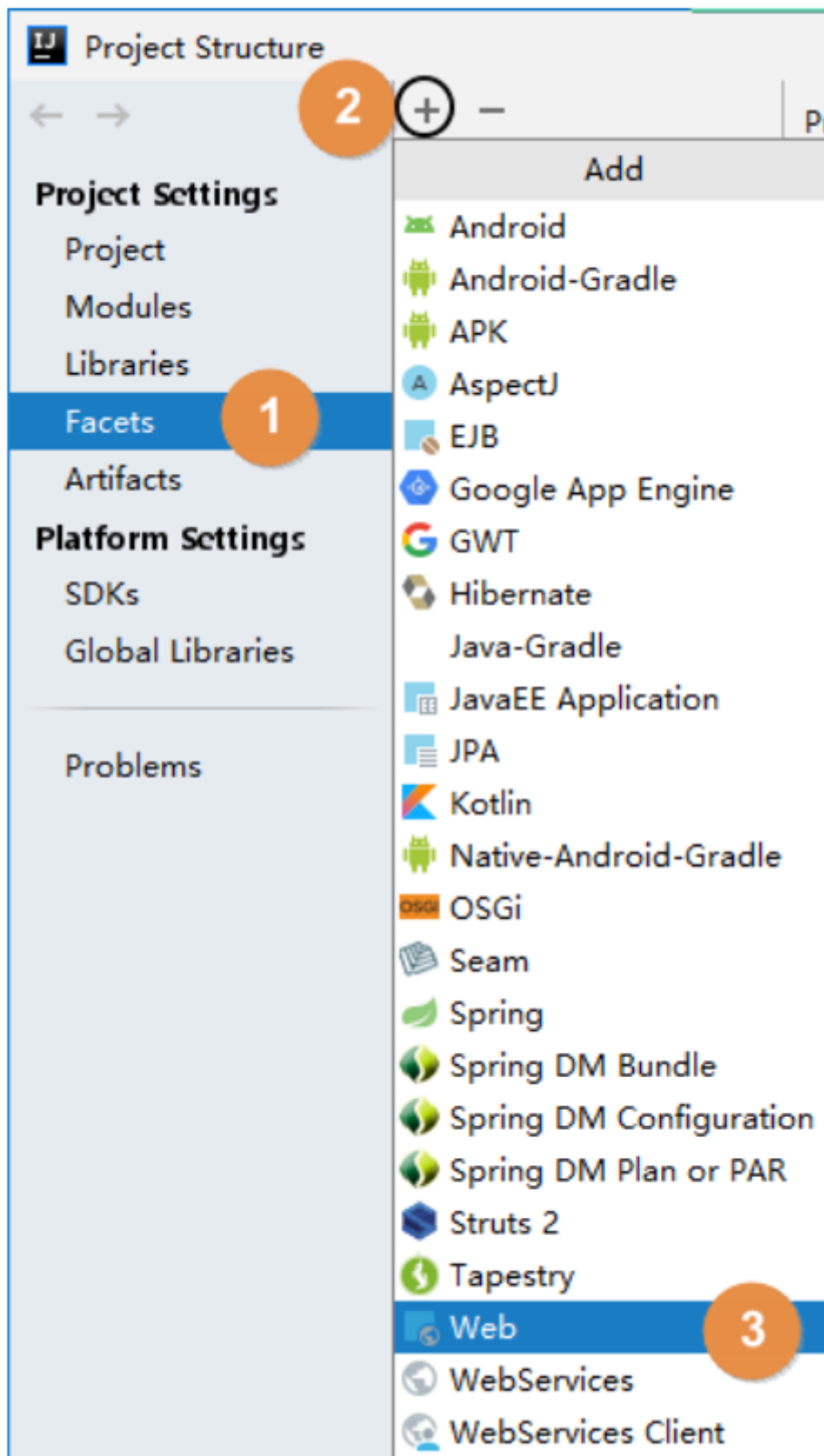
3、Web设定

首先打开**项目结构**菜单

然后到工件【**factets**】下查看IDEA是否已经帮我们自动生成了Web设定，正常来说只要我们确实设置了打包方式为war，那么会自动生成Web设定



另外，对于 IDEA 2018 诸版本没有自动生成 Web 设定，那么请参照下面两图，我们自己创建：



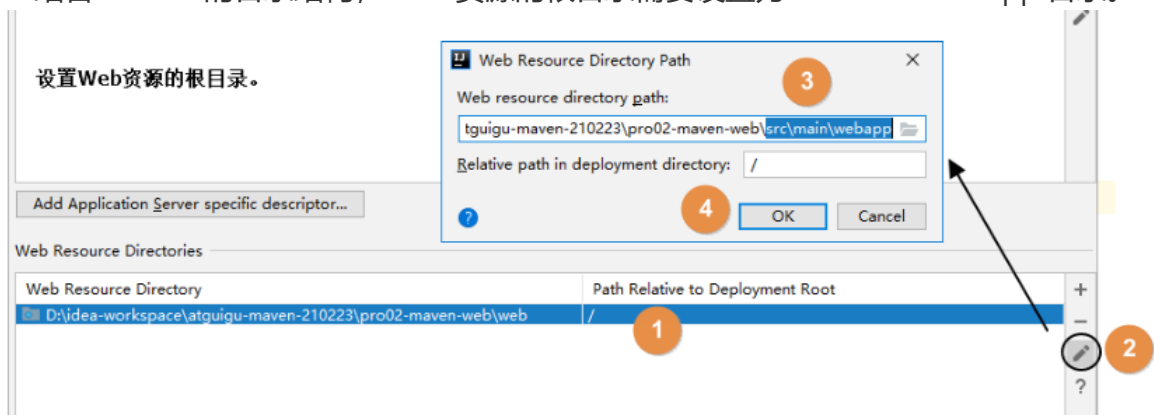
4、借助IDEA生成web.xml

1. 先修改web.xml的生成位置
2. 选择熟悉的版本，常用的版本



5、设置Web资源根目录

结合 Maven 的目录结构，Web 资源的根目录需要设置为 src/main/webapp 目录。

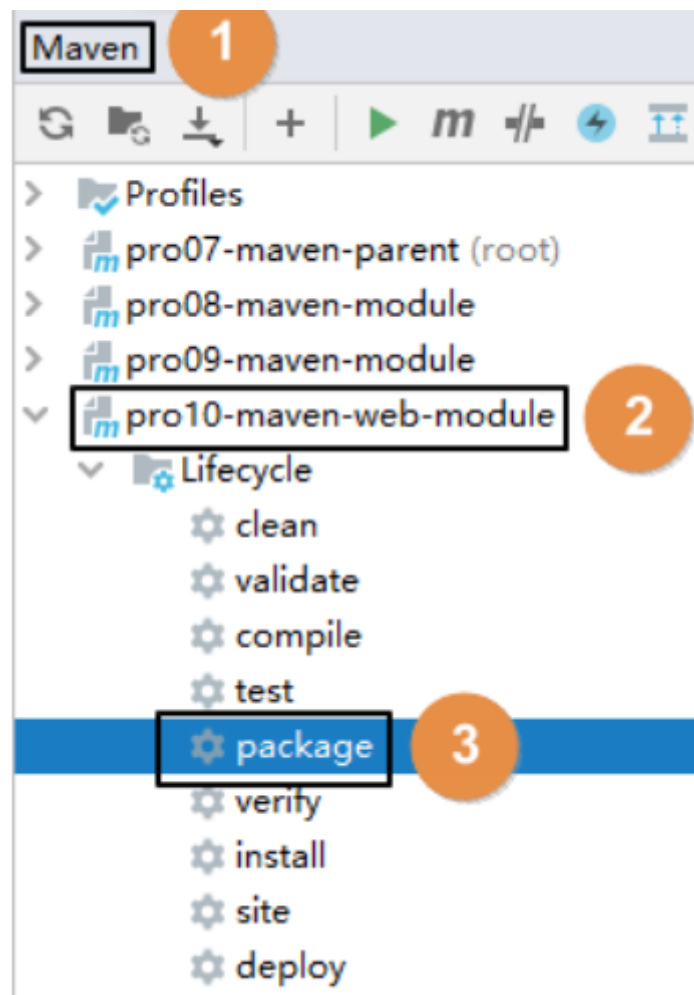


第五节 其他操作

1、在IDEA中执行Maven命令

①直接输入

IDEA插件Maven选项卡，在对应的工程展开，选择生命周期（Lifecycle）——直接双击命令



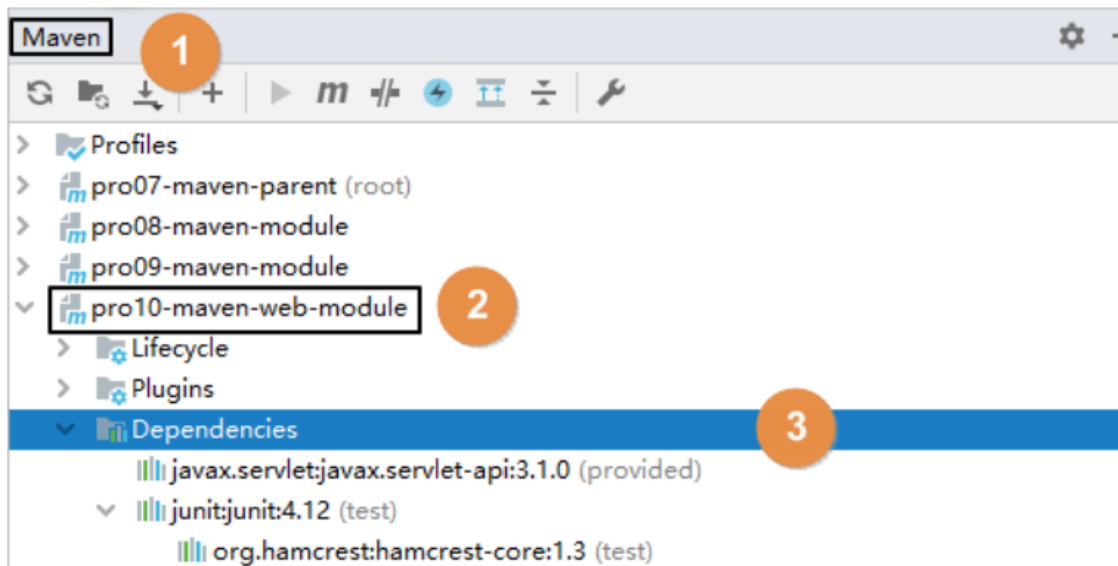
②手动输入

IDEA插件Maven选项卡，点击最上面的**M标志【执行Maven目标】**——输入命令并选择正确的工程然后回车执行

```
# -D 表示后面要附加命令的参数，字母 D 和后面的参数是紧挨着的，中间没有任何其它字符  
# maven.test.skip=true 表示在执行命令的过程中跳过测试  
mvn clean install -Dmaven.test.skip=true
```

2、在IDEA中查看某个模块的依赖信息

在IDEA插件Maven中点开对应的项目在依赖项【Dependencies】中查看



3、工程导入

Maven工程除了自己创建的，很多情况是别人创建的。而为了参与开发或者是参考学习，我们都需要导入到IDEA中。

①来自版本控制系统

目前我们通常使用的都是 **Git（本地库） + 码云（远程库）** 的版本控制系统，结合 IDEA 的相关操作方式请点[这里 \(opens new window\)](#)查看**克隆远程库**部分。

②来自工程目录

直接使用 IDEA 打开工程目录即可。下面咱们举个例子：

【1】工程压缩包

假设别人发给我们一个 Maven 工程的 zip 压缩包：maven-rest-demo.zip。从码云或 GitHub 上也可以以 ZIP 压缩格式对项目代码打包下载。

【2】解压

如果你的所有 IDEA 工程有一个专门的目录来存放，而不是散落各处，那么首先我们就把 ZIP 包解压到这个指定目录中。

【3】打开

只要我们确认在解压目录下可以直接看到 pom.xml，那就能证明这个解压目录就是我们的工程目录。那么接下来让 IDEA 打开这个目录就可以了。

【4】设置Maven核心程序位置

打开一个新的 Maven 工程，和新创建一个 Maven 工程是一样的，此时 IDEA 的 settings 配置中关于 Maven 仍然是默认值：所以我们还是需要像新建 Maven 工程那样，指定一下 Maven 核心程序位置：

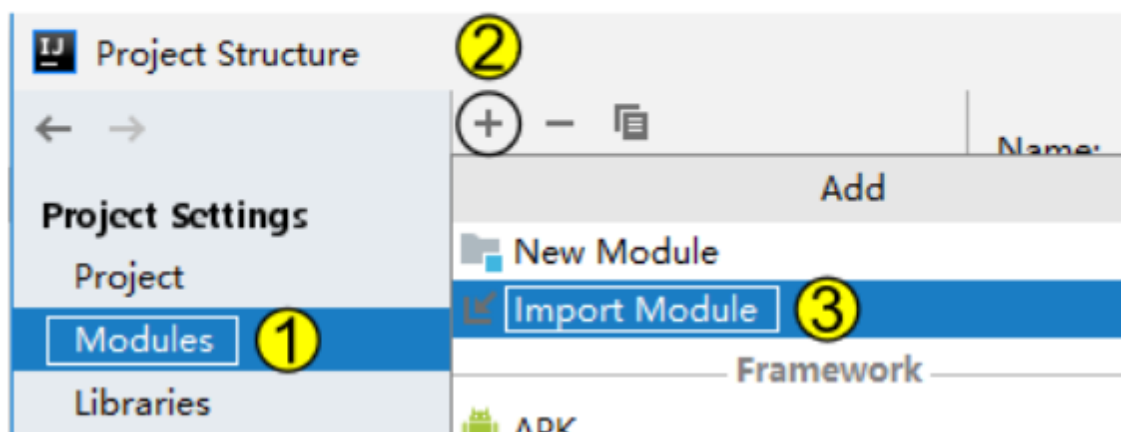
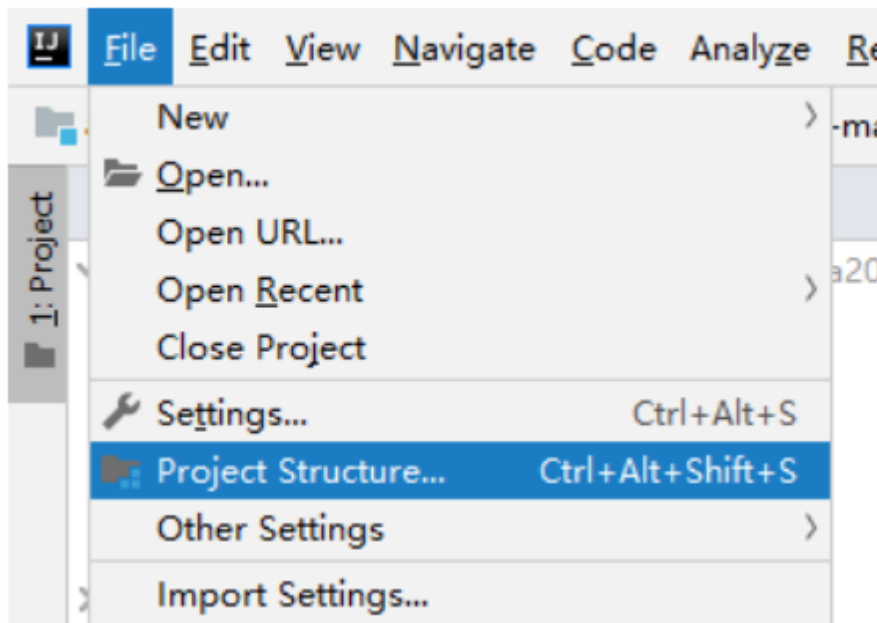
4、模块导入

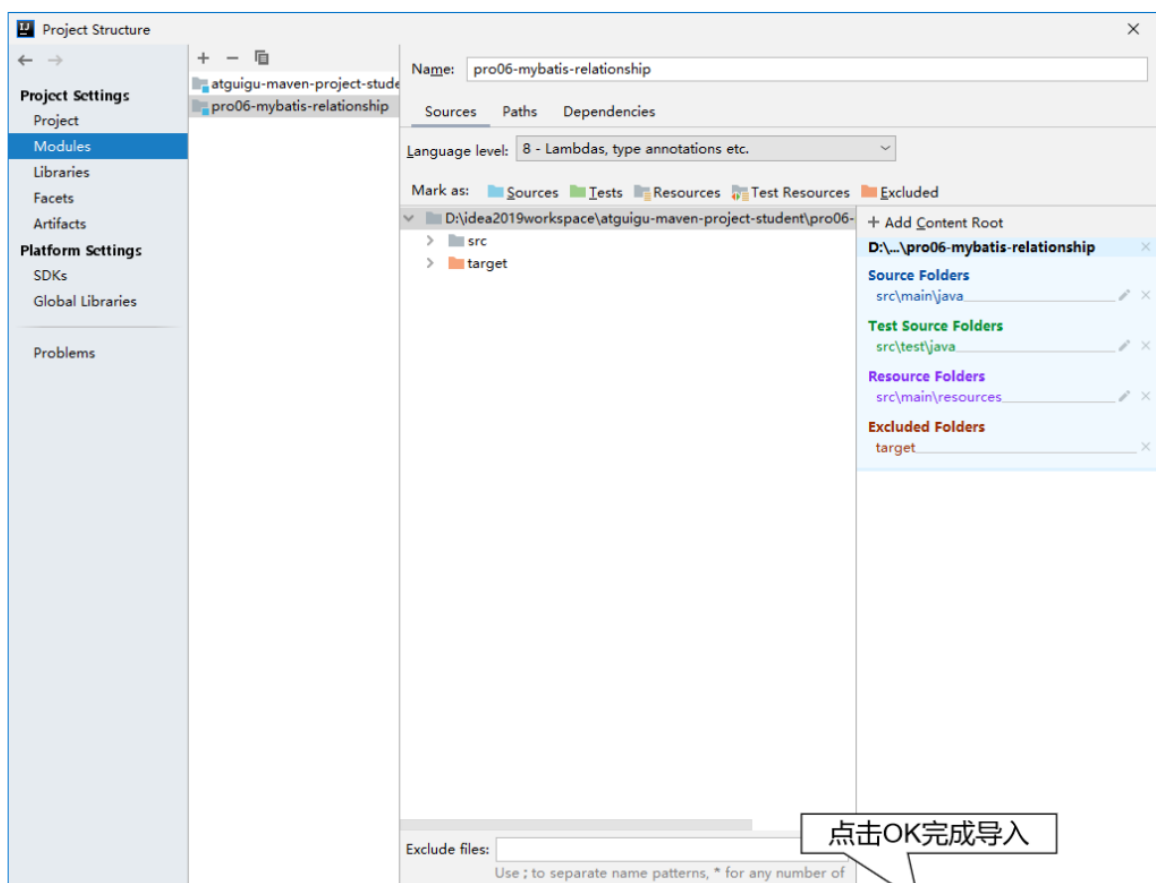
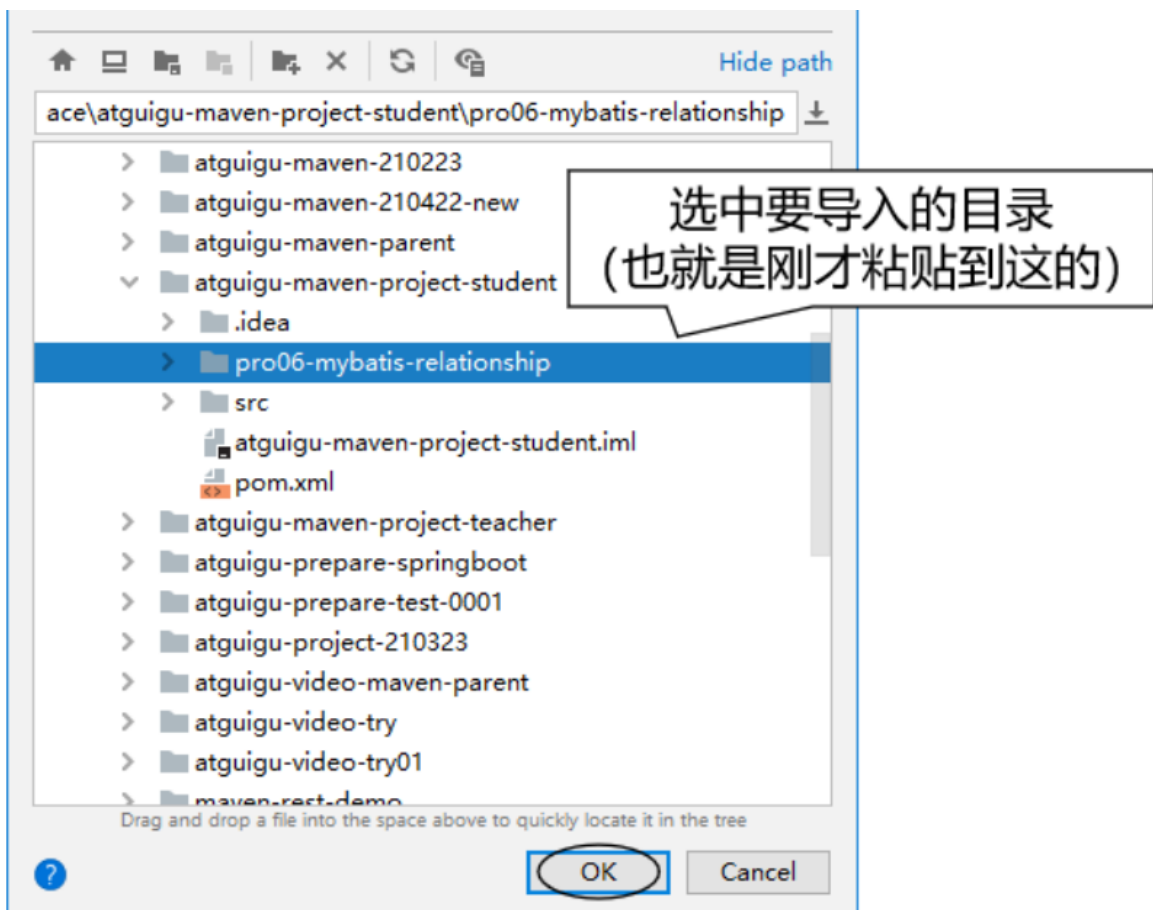
①情景再现

在实际开发中，通常会忽略模块（也就是module）所在的项目（也就是project） 仅仅导入某一个模块本身。这么做很可能是类似这样的情况：比如基于 Maven 学习 SSM 的时候，做练习需要导入老师发给我们的代码参考。

②导入Java类型模块

- 【1】找到老师发的工程目录
- 【2】复制想要导入的模块
- 【3】粘贴到我们自己工程目录下
- 【4】在IDEA中执行导入





【5】修改pom.xml

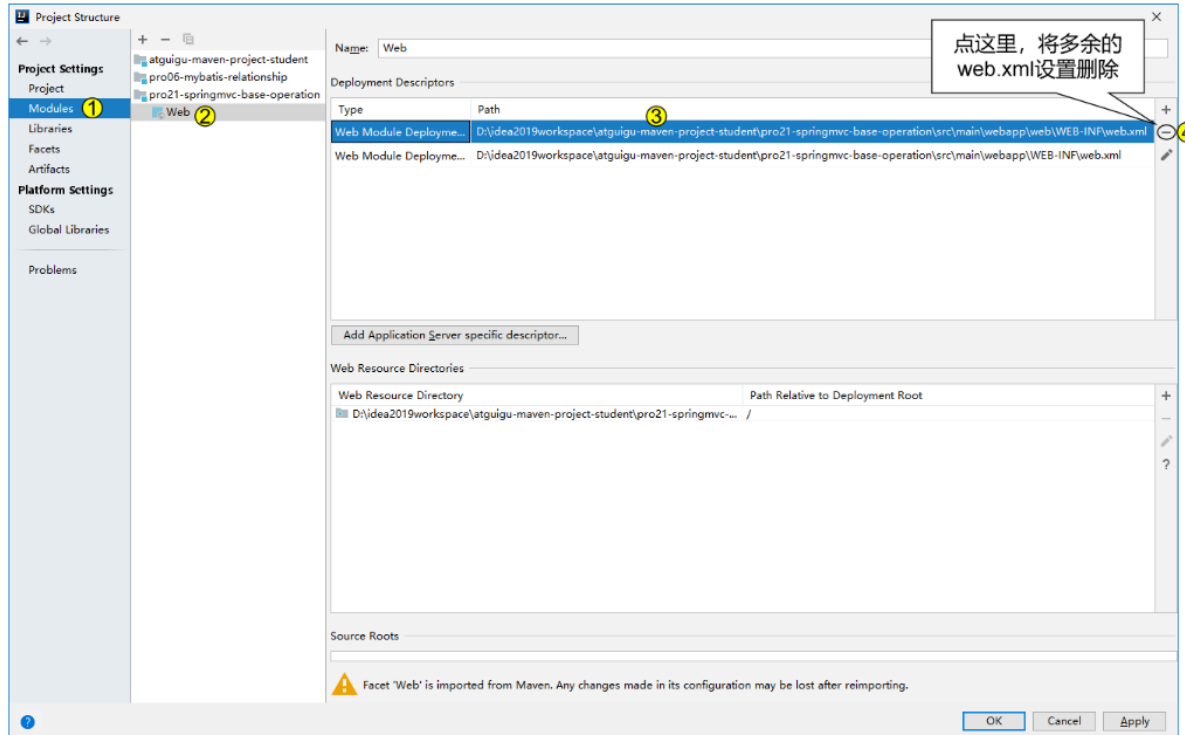
刚刚导入的 module 的父工程坐标还是以前的，需要改成我们自己的 project。

【6】最终效果

留意图标是否正确

③导入Web类型模块

其它操作和上面演示的都一样，只是多一步：删除多余的、不正确的 web.xml 设置。
如下图所示：



第五章 其他核心概念

1、生命周期

①作用

为了让构建过程自动化完成，Maven设定了三个生命周期，生命周期中的每一个环节对应构建中的一个操作。

②三个生命周期

生命周期名称	作用	各个环节
Clean	清理操作相关	pre-clean clean post-clean
Site	生成站点相关	pre-site site post-site deploy-site
Default	主要构建过程	validate generate-sources process-sources generate-resources process-resources 复制并处理资源文件，至目标目录，准备打包。 compile 编译项目 main 目录下的源代码。 process-classes generate-test-sources process-test-sources generate-test-resources process-test-resources 复制并处理资源文件，至目标测试目录。 test-compile 编译测试源代码。 process-test-classes test 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。 prepare-package package 接受编译好的代码，打包成可发布的格式，如JAR。 pre-integration-test integration-test post-integration-test verify install 将包安装至本地仓库，以让其它项目依赖。 deploy 将最终的包复制到远程的仓库，以让其它开发人员共享；或者部署到服务器上运行（需借助插件，例如：cargo）。

③特点

- 前面三个生命周期彼此是独立的。
- 在任何一个生命周期内部，执行任何一个具体环节的操作，都是从本周期最初的位置开始执行，直到指定的地方。（本节记住这句话就行了，其他的都不需要

记)

Maven 之所以这么设计其实就是为了提高构建过程的自动化程度：让使用者只关心最终要干的即可，过程中的各个环节是自动执行的。

2、插件和目标

①插件

Maven 的核心程序仅仅负责宏观调度，不做具体工作。具体工作由 Maven 插件完成。例如：编译是由 `maven-compiler-plugin-3.1.jar` 插件来执行的。

②目标

一个插件可以对应多个目标，而每一个目标都和生命周期中的某一个环节对应。

Default 生命周期中有 `compile` 和 `test-compile` 两个和编译相关的环节，这两个环节对应 `compile` 和 `test-compile` 两个目标，而这两个目标都是由 `maven-compiler-plugin-3.1.jar` 插件来执行的。

3、仓库

- 本地仓库：在当前电脑上，为电脑上所有 Maven 工程服务
- 远程仓库：需要联网
 - 局域网：我们自己搭建的 Maven 私服，例如使用 Nexus 技术。
 - Internet
 - 中央仓库
 - 镜像仓库：内容和中央仓库保持一致，但是能够分担中央仓库的负载，同时让用户能够就近访问提高下载速度，例如：Nexus aliyun

建议：不要中央仓库和阿里云镜像混用，否则 jar 包来源不纯，彼此冲突。

专门搜索 Maven 依赖信息的网站：<https://mvnrepository.com/>

第六章 单一架构案例

第一节 创建工程，引入依赖

第二节 搭建环境：持久化层

第三节 搭建环境：事务控制

第四节 搭建环境：表述层

第五节 搭建环境：辅助功能

第六节 业务功能：登录

第七节 业务功能：显示奏折列表

第八节 业务功能：显示奏折详情

第九节 业务功能：批复奏折

第十节 业务功能：登录检查

第十一节 打包部署

附录

网站: <https://mvnrepository.com>