

## 1 章 Maven 简介



官网: <http://maven.apache.org/>

### 1.1 软件是一个工程

我们在日常生活常能听到工程这个词，像桥梁工程、道路工程、南水北调工程等等。

工程说简单点就是各个行业的从业人员通过总结规律或者方法，以最短的时间和人力、物力来做出高效可靠的东西。我们也就能理解桥梁工程，其实就是人们通过经验的总结和各種研究得出来的、用来修建桥梁时所采用的高效的方法，当然这种方法是可复用的。我们将这种作工程的思想应用到软件上，于是就产生了一软件工程。

软件工程：为了能够实现软件的流水线式生产，在设计和构建软件时能够有一种规范和工程化的方法，人们便提出了软件工程概念。

上面的内容做个日常生活的类比，做道菜就是一个工程。今天心情好，想吃红烧肉，自动动手做：

- 1.想买什么猪的那个位置的肉，黑猪肉，土猪肉等，使用冰糖还是绵白糖，用什么牌子的酱油等
- 2.到菜市场购买各种原料。
- 3.准备材料，洗肉，切块，花椒，大料等
- 4.开始做了，肉下锅炖煮，不同时间加入花椒，大料，冰糖，酱油
- 5.炖煮一段时间后，看肉是否熟透，尝试口味，是不是咸了等等
- 6.做好了，开始吃了
- 7.需要刷碗，倒垃圾

这些工作从头做到尾步骤非常繁琐，每个步骤都是费时费力的。所以才出现净菜，半成品菜，拿回家直接做可以了，只做 6,7 步骤就可以。

软件开发需要编译代码—>开发人员自己测试代码—>把代码打包—>部署项目到测试服务器—>测试人员测试功能—>测试测试出 bug

开发人员需要修改 bug—>开发人员自己测试代码—>把代码打包—>部署项目到测试服务器—>测试人员测试功能—>直到符合功能要求。

上述过程需要重复多次，大型项目中构建项目比较复杂，有很多的配置文件，jar 文件，多个子项目等等。都用人力完成费时费力，效率比较低。maven 可以让我们从上面的工作中解脱出来。

**maven 是自动化构建工具。**

## 1.2 传统项目开发存在的问题

一个项目做成一个工程，造成工程比较庞大，需要使用多模块来划分项目；

项目中需要的数量众多的 jar 包，需要手动下载并引入，并且多个项目需要的 jar 包存在重复的问题；

项目中需要的 jar 包有版本兼容的问题，需要手动解决；

项目中需要的 jar 包又依赖其它的 jar 包，需要手动解决。

## 1.3 Maven 概述

Maven 是 Apache 软件基金会组织维护的一款自动化构建工具，专注服务于 Java 平台的项目构建和依赖管理。Maven 这个单词的本意是：专家，内行。读音是['meɪv(ə)n]或['mevn]。

Maven 是目前最流行的自动化构建工具，对于生产环境下多框架、多模块整合开发有重要作用，Maven 是一款在大型项目开发过程中不可或缺的重要工具。

Maven 可以整合多个项目之间的引用关系，我们可以根据业务和分层需要任意拆分一个项目；

Maven 提供规范的管理各个常用 jar 包及其各个版本，并且可以自动下载和引入项目中；

Maven 可以根据指定版本自动解决 jar 包版本兼容问题；

Maven 可以把 jar 包所依赖的其它 jar 包自动下载并引入项目。

类似自动化构建工具还有：Ant, Maven, Gradle。

构建过程中的各个环节：清理、编译、测试、报告、打包、安装、部署。

构建（build），是面向过程的(从开始到结尾的多个步骤)，涉及到多个环节的协同工作。

构建过程的几个主要环节

①清理：删除以前的编译结果，为重新编译做好准备。

②编译：将Java源程序编译为字节码文件。

③测试：针对项目中的关键点进行测试，确保项目在迭代开发过程中关键点的正确性。

④报告：在每一次测试后以标准的格式记录和展示测试结果。

⑤打包：将一个包含诸多文件的工程封装为一个压缩文件用于安装或部署。Java 工程对应 jar 包，Web 工程对应war包。

⑥安装：在Maven环境下特指将打包的结果——jar包或war包安装到本地仓库中。

⑦部署：将打包的结果部署到远程仓库或将war包部署到服务器上运行

## 1.4 Maven 核心概念

Maven能够实现自动化构建是和它的内部原理分不开的，这里我们从 Maven的九个核心概念入手，看看Maven是如何实现自动化构建的

①POM

②约定的目录结构

③坐标

④依赖管理

⑤仓库管理

⑥生命周期

⑦插件和目标

⑧继承

⑨聚合

## 1.5 安装 Maven 环境

- 1、确保安装了 java 环境:maven 本身就是 java 写的，所以要求必须安装 JDK。

查看 java 环境变量：echo %JAVA\_HOME%

- 2、下载并解压 maven 安装程序：

<http://maven.apache.org/download.cgi>

- 3、配置 Maven 的环境变量：

MAVEN\_HOME=d:/apache-maven-3.3.9 或者 M2\_HOME=d:/apache-maven-3.3.9

path=%MAVEN\_HOME%/bin; 或者%M2\_HOME%/bin;

- 4、验证是否安装成功:

`mvn -v`

## 2 章 Maven 的核心概念:

### 2.1 Maven 工程约定目录结构

maven 中约定的目录结构:

```
Hello
|---src
|---|---main
|---|---|---java
|---|---|---resources
|---|---test
|---|---|---java
|---|---|---resources
|---pom.xml
```

说明：Hello:根目录，也就是工程名

src：源代码

main：主程序

java：主程序的 java 源码

resources：主程序的配置文件

test：测试程序

java：测试程序的 java 源码

resources：测试程序的配置文件

pom.xml：Maven 工程的核心配置文件。

一般情况下，我们习惯上采取的措施是：约定>配置>编码

maven 的 pom.xml 记录的关于构建项目的各个方面的设置，maven 从 pom.xml 文件开始，按照助约定的工程目录编译，测试，打包，部署，发布项目。

### 2.1.1 第一个 maven 工程

按照如下步骤，实现第一个 maven 项目，以 maven 推荐的约定方式创建目录，类文件。

- 1.某个目录中创建文件夹 Hello
- 2.在 Hello 中创建子目录 src
- 3.拷贝 pom.xml 到 Hello 目录和 src 是同级放置的。
- 4.进入 src 目录，创建 main， test 目录
- 5.进入 main 目录，创建 java，resources 目录。
- 6.进入 java 目录，创建目录 com/bjpowernode/
- 6.在 com/bjpowernode/目录下创建 HelloMaven.java 文件，定义 `int addNumber(int n1,n2){ return n1+n2};`  
定义 `public static void main(String args[]) { System.out.println("Hello Manven"); //也可以调用 addNumber()方法 }`
- 7.进入到 Hello 目录在，执行 `mvn compile`
- 8.进入到 target/classes 目录执行 `java com.bjpowernode.HelloMaven`

## 2.2 POM 文件

即 Project Object Model 项目对象模型。Maven 把一个项目的结构和内容抽象成一个模型，在 xml 文件中  
进行声明，以方便进行构建和描述，pom.xml 是 Maven 的灵魂。所以，maven 环境搭建好之后，所有的学习和  
操作都是关于 pom.xml 的。

pom.xml 初识：

基本信息		
modelVersion	Maven 模型的版本，对于 Maven2 和 Maven3 来说，它只能是 4.0.0	
groupId	组织 id，一般是公司域名的倒写。 格式可以为： 1. 域名倒写。 例如 com.baidu 2. 域名倒写+项目名。例如 com.baidu.appolo	groupId 、 artifactId 、 version 三个元素

artifactId	项目名称，也是模块名称，对应 groupId 中 项目中的子项目。	生 成 了 一 个 Maven 项目的基 本坐标，在众多的 maven 项目中可 以唯一定位到某一 个项目。坐标也决 定着将来项目在仓 库中的路径及名 称。
version	项目的版本号。如果项目还在开发中，是不稳定版本，通常在版本后带-SNAPSHOT version 使用三位数字标识，例如 1.1.0	
packaging	项目打包的类型，可以使 jar、war、rar、ear、pom，默认是 jar	
依赖		
dependencies 和 dependency	Maven 的一个重要作用就是管理 jar 包，为了一个项目可以构建或运行，项目中不可避免的，会依赖很多其他的 jar 包，在 Maven 中，这些 jar 就被称为依赖，使用标签 dependency 来配置。而这种依赖的配置正是通过坐标来定位的，由此我们也不难看出，maven 把所有的 jar 包也都视为项目存在了。	
配置属性		
properties	properties 是 用 来 定 义 一 些 配 置 属 性 的 ， 例 如 project.build.sourceEncoding（项目构建源码编码方式），可以设置为 UTF-8，防止中文乱码，也可定义相关构建版本号，便于日后统一升级。	
构建		
build	build 表示与构建相关的配置，例如设置编译插件的 jdk 版本	
继承		
parent	在 Maven 中，如果多个模块都需要声明相同的配置，例如：groupId、version、有相同的依赖、或者相同的组件配置等，也有类似 Java 的继承机制，用 parent 声明要继承的父工程的 pom 配置。	
聚合		
modules	在 Maven 的多模块开发中，为了统一构建整个项目的所有模块，可以提供一个额外的模块，该模块打包方式为 pom，并且在其中使用 modules 聚合的其它模块，这样通过本模块就可以一键自动识别模块间的依赖关系来构建所有模块，叫 Maven 的聚合。	

## 2.3 仓库

### 2.3.1 仓库的概念

现在我们对maven工程有一个大概的认识了，那现在思考一个问题，maven怎么就这么神奇，我们写完的工程交给他之后，他就能自动帮我们管理，我们依赖的jar包它从哪儿获取呢？有同学说已经安装了，在它的安装包啊，大家可以看一下maven下载下来才8M，我们需要的jar包有时候都几百兆甚至几个G，它从哪儿弄去呢？其实，maven有仓库的概念。在Maven中，任何一个依赖、插件或者项目构建的输出，都可以称之为构件。Maven核心程序仅仅定义了自动化构建项目的生命周期，但具体的构建工作是由特定的构件完成的。而且为了提高构建的效率和构件复用，maven把所有的构件统一存储在某一个位置，这个位置就叫做仓库。

### 2.3.2 仓库存什么

仓库是存放东西的，Maven 仓库的是：

1. Maven 的插件，插件也是一些 jar，这些 jar 可以完成一定的功能。
2. 我们自己开发项目的模块
3. 第三方框架或工具的 jar 包

### 2.3.3 仓库的分类

根据仓库存储的位置，把仓库分为本地仓库和远程仓库。

**本地仓库**，存在于当前电脑上，默认存放在 `~\.m2\repository` 中，为本机上所有的 Maven 工程服务。你也可以通过 Maven 的配置文件 `Maven_home/conf/settings.xml` 中修改本地仓库所在的目录。

~ 是用户的主目录，windows 系统中是 `c: /user/` 登录系统的用户名

**远程仓库**，分为为全世界范围内的开发人员提供服务的中央仓库、为全世界范围内某些特定的用户提供服务的中央仓库镜像、为本公司提供服务自己架设的私服。中央仓库是 maven 默认的远程仓库，其地址是：`http://repo.maven.apache.org/maven2/`

中央仓库，包含了绝大多数流行的开源 Java 构件，以及源码、作者信息、许可证信息等。一般来说，简单的 Java 项目依赖的构件都可以在这里下载得到。

私服是一种特殊的远程仓库，它是架设在局域网内的仓库服务，私服代理广域网上的远程仓库，供局域网内的 Maven 用户使用。当 Maven 需要下载构件的时候，它从私服请求，如果私服上不存在该构件，则从外部的远程仓库下载，缓存在私服上之后，再为 Maven 的下载请求提供服务。我们还可以把一些无法从外部仓库下载到的构件上传到私服上。

分类说明：

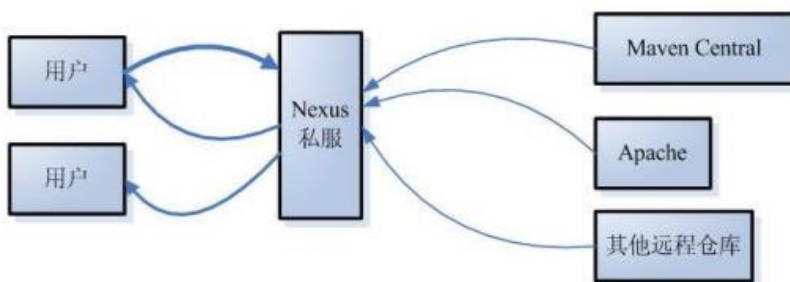
1) 本地仓库：本机当前电脑上的资源存储位置，为本机上所有 Maven 工程提供服务

2) 远程仓库：不在本机上，通过网络才能使用。多电脑共享使用的。

①：**中央仓库**：通过 Internet 访问，为全世界所有 Maven 工程服务。最权威的。

②：**中央仓库的镜像**：架设在不同位置，欧洲，美洲，亚洲等每个洲都有若干的服务器，为中央仓库分担流量。减轻中央仓库的访问，下载的压力。所在洲的用户首先访问的是本洲的镜像服务器。

③：**私服**：在局域网环境中部署的服务器，为当前局域网范围内的所有 Maven 工程服务。公司中常常使用这种方式。



### 2.3.4 Maven 对仓库的使用

在 Maven 构建项目的过程中如果需要某些插件，首先会到 Maven 的本地仓库中查找，如果找到则可以直接使用；如果找不到，它会自动连接外网，到远程中央仓库中查找；如果远程仓库中能找到，则先把所需要的插件下载到本地仓库，然后再使用，并且下次再用到相同的插件也可以直接使用本地仓库的；如果没有外网或者远程仓库中也找不到，则构建失败。

资源搜索地址：`https://mvnrepository.com/`



## 2.4 Maven 的生命周期

对项目的构建是建立在生命周期模型上的，它明确定义项目生命周期各个阶段，并且对于每一个阶段提供相对应的命令，对开发者而言仅仅需要掌握一小堆的命令就可以完成项目各个阶段的构建工作。

构建项目时按照生命周期顺序构建，每一个阶段都有特定的插件来完成。不论现在要执行生命周期中的哪个阶段，都是从这个生命周期的最初阶段开始的。

对于我们程序员而言，无论我们要进行哪个阶段的构建，直接执行相应的命令即可，无需担心它前边阶段是否构建，Maven 都会自动构建。这也就是 Maven 这种自动化构建工具给我们带来的好处。

## 2.5 Maven 的常用命令

Maven 对所有的功能都提供相对应的命令，要想知道 maven 都有哪些命令，那要看 maven 有哪些功能。一开始就跟大家说了，maven 三大功能：管理依赖、构建项目、管理项目信息。管理依赖，只需要声明就可以自动到仓库下载；管理项目信息其实就是生成一个站点文档，一个命令就可以解决，最后再说；那 maven 功能的主体其实就是项目构建。

Maven 提供一个项目构建的模型，把编译、测试、打包、部署等都对应成一个个的生命周期阶段，并对每一个阶段提供相应的命令，程序员只需要掌握一小堆命令，就可以完成项目的构建过程。

`mvn clean` 清理(会删除原来编译和测试的目录，即 `target` 目录，但是已经 `install` 到仓库里的包不会删除)

`mvn compile` 编译主程序(会在当前目录下生成一个 `target`，里边存放编译主程序之后生成的字节码文件)

`mvn test-compile`

编译测试程序(会在当前目录下生成一个 `target`，里边存放编译测试程序之后生成的字节码文件)

`mvn test` 测试(会生成一个目录 `surefire-reports`，保存测试结果)

`mvn package`

打包主程序(会编译、编译测试、测试、并且按照 `pom.xml` 配置把主程序打包生成 `jar` 包或者 `war` 包)

`mvn install` 安装主程序(会把本工程打包，并且按照本工程的坐标保存到本地仓库中)

`mvn deploy` 部署主程序(会把本工程打包，按照本工程的坐标保存到本地库中，并且还会保存到私服仓库中。还会自动把项目部署到 `web` 容器中)。

**注意：执行以上命令必须在命令行进入 `pom.xml` 所在目录！**

### 2.5.1 练习 maven 命令

以第一个 maven 项目为例，进入到 Hello 目录中执行 maven 各种命令

1.准备工作，`pom.xml` 添加依赖

```
<dependencies>
  <!-- 单元测试 -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
  </dependency>
</dependencies>
```

2.在 `Hello/src/test/` 目录下创建 `java`，`resources` 目录

3.在 `Hello/src/test/java` 目录下，创建 `com/bjpowernode/` 目录

4. 在 `com/bjpowernode/` 目录中创建 `MyTest.java` 文件

导入类:

```
import org.junit.Assert;
import org.junit.Test;
```

定义方法

```
public class MyTest {
    @Test
    public void testAddNumber(){
        System.out.println("执行 HelloMaven 类 addNumber()方法");
        int n1 = 10;
        int n2 = 20;
        int res = 0;
        HelloMaven hello =new HelloMaven();
        res = hello.addNumber(n1,n2);
        // 期望值，实际值
        Assert.assertEquals(30,res);
    }
}
```

5.执行先执行 **mvn compile** ,观察目录结构的变化, **生成 target 目录**

6.在执行 **mvn clean**, 观察 **target 目录被清除**

7.执行 **mvn compile**

8.进入 **target/classes** 目录执行 **java com.bjpowernode.HelloMaven**

9.进入 **Hello** 目录, 执行 **mvn test-compile** 生成 **test-target** 测试编译后的目录

10. 进入 **Hello** 目录, 执行 **mvn test** 执行 **MyTest** 类中方法, 生成测试报告

11.进入目录 **surefire-reports**, 查看测试报告

12.修改 **MyTest.java**, 增加测试方法

```
@Test
public void testAddNumber2(){
    System.out.println("执行 HelloMaven 类 addNumber()方法");
    int n1 = 20;
    int n2 = 30;
    int res = 0;
    HelloMaven hello =new HelloMaven();
    res = hello.addNumber(n1,n2);
    // 期望值，实际值
    Assert.assertEquals(60,res);
}
```

13. 进入 **Hello** 目录, 执行 **mvn test-compile**

14. 进入 **Hello** 目录, 执行 **mvn test**

15. 修改 **testAddNumber2()**方法中 **60** 为 **50**

16. 进入 **Hello** 目录, 执行 **mvn package** , 生成 **xxx.jar** 文件, 这就是所说的打包

17. 进入 **Hello** 目录, 执行 **mvn install** , 把 **xxx.jar** 文件安装到本地 **maven** 仓库, 安装成功后查看仓库中的 **jar** 文件

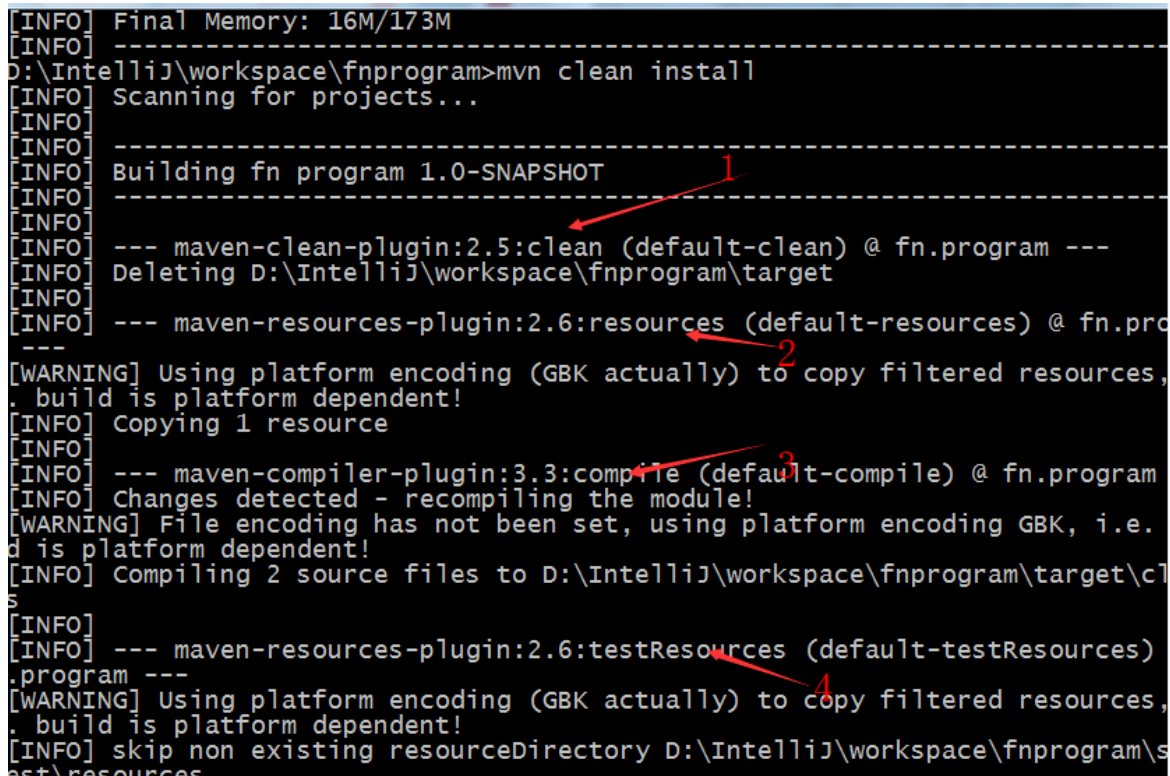
## 2.6 插件

maven 过程构建周期, 由 maven 的插件 **plugin** 来执行完成。



官网插件说明: <http://maven.apache.org/plugins/>

在项目根目录下执行: mvn clean install



```
[INFO] Final Memory: 16M/173M
[INFO] -----
D:\IntelliJ\workspace\fnprogram>mvn clean install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building fn program 1.0-SNAPSHOT
[INFO] -----
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ fn.program ---
[INFO] Deleting D:\IntelliJ\workspace\fnprogram\target
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ fn.pro
---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources,
. build is platform dependent!
[INFO] Copying 1 resource
[INFO] --- maven-compiler-plugin:3.3:compile (default-compile) @ fn.program
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding GBK, i.e.
d is platform dependent!
[INFO] Compiling 2 source files to D:\IntelliJ\workspace\fnprogram\target\cl
s
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources)
.program ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources,
. build is platform dependent!
[INFO] skip non existing resourceDirectory D:\IntelliJ\workspace\fnprogram\s
st\resources
```

The screenshot shows the output of the `mvn clean install` command. Four red arrows point to specific plugin names in the log: 1 points to `maven-clean-plugin:2.5:clean`, 2 points to `maven-resources-plugin:2.6:resources`, 3 points to `maven-compiler-plugin:3.3:compile`, and 4 points to `maven-resources-plugin:2.6:testResources`.

解释说明

### 2.6.1 clean 插件 maven-clean-plugin:2.5

clean 阶段是独立的一个阶段, 功能就是清除工程目前下的 target 目录

### 2.6.2 resources 插件 maven-resources-plugin:2.6

resource 插件的功能就是把项目需要的配置文件拷贝到指定的目当, 默认是拷贝 `src\main\resources` 目录下的件到 `classes` 目录下

### 2.6.3 compile 插件 maven-compiler-plugin

compile 插件执行时先调用 resources 插件, 功能就是把 `src\mainjava` 源码编译成字节码生成 class 文件, 并把编译好的 class 文件输出到 `target\classes` 目录下

### 2.6.4 test 测试插件

单元测试所用的 compile 和 resources 插件和主代码是相同的, 但执行的目标不行, 目标 `testCompile` 和 `testResources` 是把 `src\test\java` 下的代码编译成字节码输出到 `target\test-classes`, 同时把 `src\test\resources` 下的配置文件拷贝到 `target\test-classes`

## 2.6.5 package 打包插件 maven-jar-plugin

这个插件是把 class 文件、配置文件打成一个 jar(war 或其它格式)包

## 2.6.6 deploy 发布插件 maven-install-plugin

发布插件的功能就是把构建好的 artifact 部署到本地仓库，还有一个 deploy 插件是将构建好的 artifact 部署到远程仓库

## 2.6.7 常用插件

插件可以在自己的项目中设置，最常使用的是 maven 编译插件。设置项目使用的 jdk 版本时通过编译插件指定。pom.xml 文件<build>中设置。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

## 2.7 坐标(gav)

Maven 把任何一个插件都作为仓库中的一个项目进行管理，用一组(三个)向量组成的坐标来表示。坐标在仓库中可以唯一定位一个 Maven 项目。

groupId: 组织名，通常是公司或组织域名倒序+项目名

artifactId: 模块名，通常是工程名

version: 版本号

需要特别指出的是，项目在仓库中的位置是由坐标来决定的：groupId、artifactId 和 version 决定项目在仓库中的路径，artifactId 和 version 决定 jar 包的名称。

## 2.8 依赖(dependency)

一个 Maven 项目正常运行需要其它项目的支持，Maven 会根据坐标自动到本地仓库中进行查找。对于程序员自己的 Maven 项目需要进行安装，才能保存到仓库中。

不用 maven 的时候所有的 jar 都不是你的，需要去各个地方下载拷贝，用了 maven 所有的 jar 包都是你的，想要谁，叫谁的名字就行。maven 帮你下载。

pom.xml 加入依赖的方式：

log4j 日志依赖

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

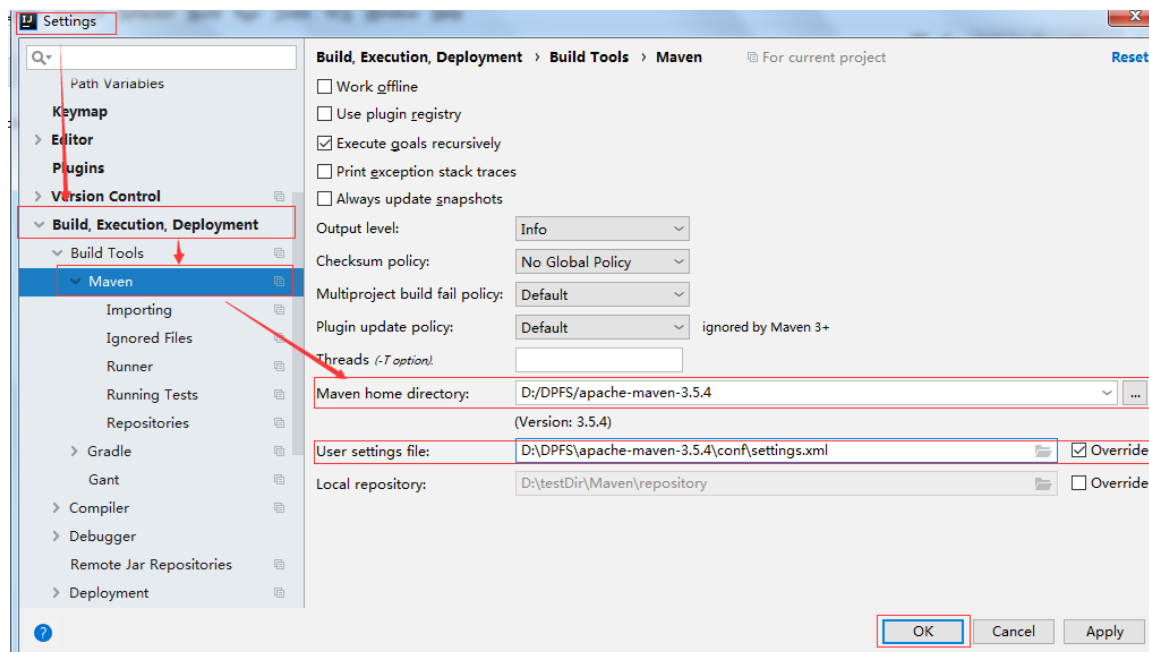
junit 单元测试依赖

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
</dependency>
```

## 3 章 Maven 在 IDEA 中的应用

### 3.1 IDEA 集成 Maven

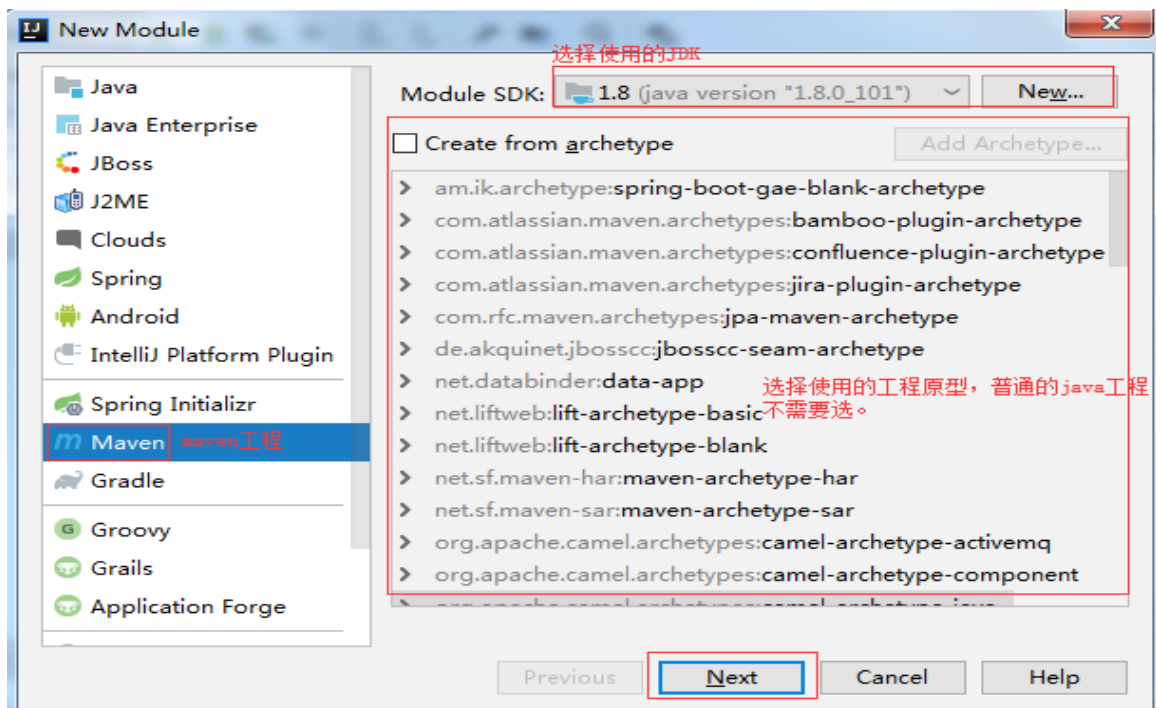
File---->Settings: 设置 maven 安装主目录、maven 的 settings.xml 文件和本地仓库所在位置。



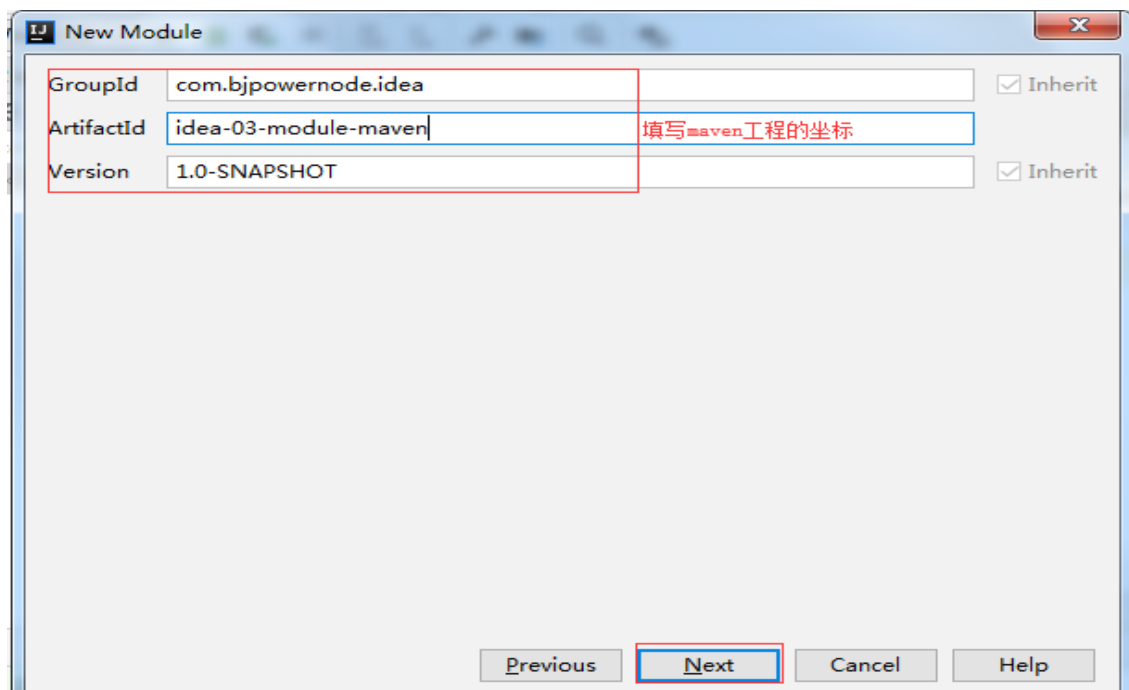
### 3.2 IDEA 创建 Maven 版 java 工程

#### 3.2.1 创建 maven 版 java 工程

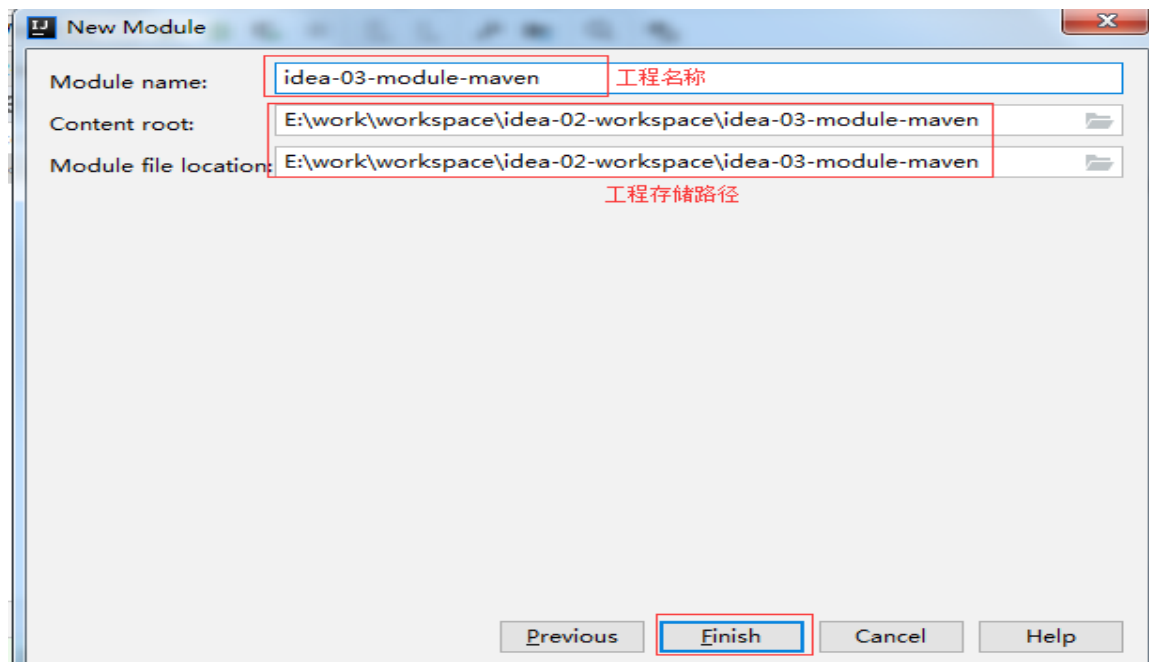
File-->New-->Module...:



### 3.2.2 填写 maven 工程的坐标



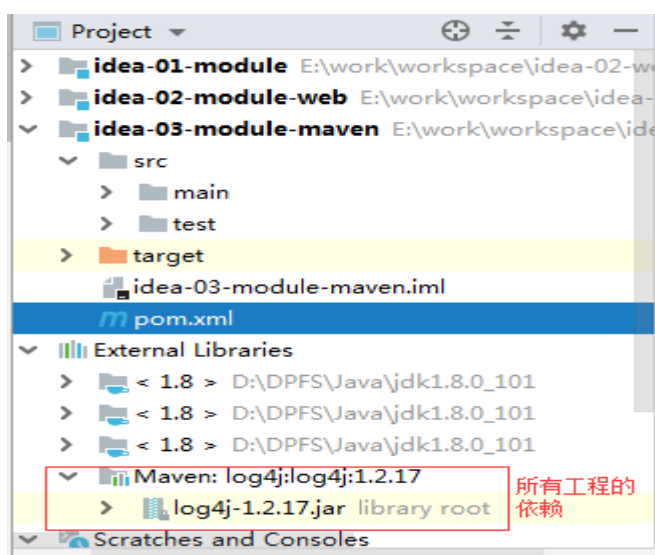
### 3.2.3 填写工程名和存储路径

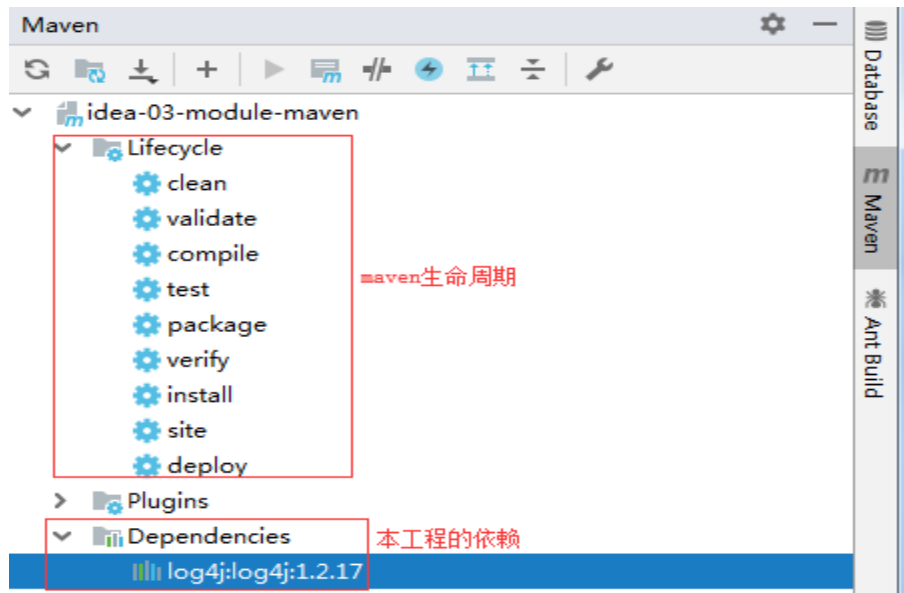


### 3.2.4 pom.xml 加入依赖

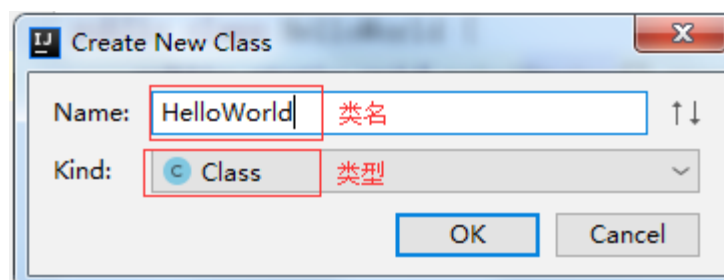
```
<dependency>  
  <groupId>log4j</groupId>  
  <artifactId>log4j</artifactId>  
  <version>1.2.17</version>  
</dependency>
```

### 3.2.5 创建后视图





### 3.2.6 创建测试类

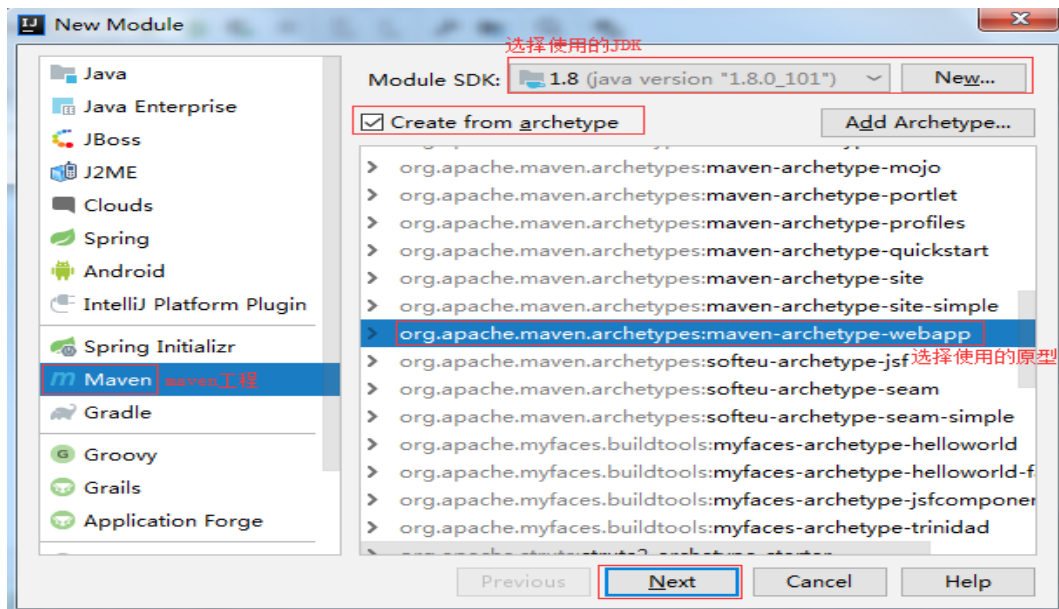


输出一条语句，测试项目构建成功。

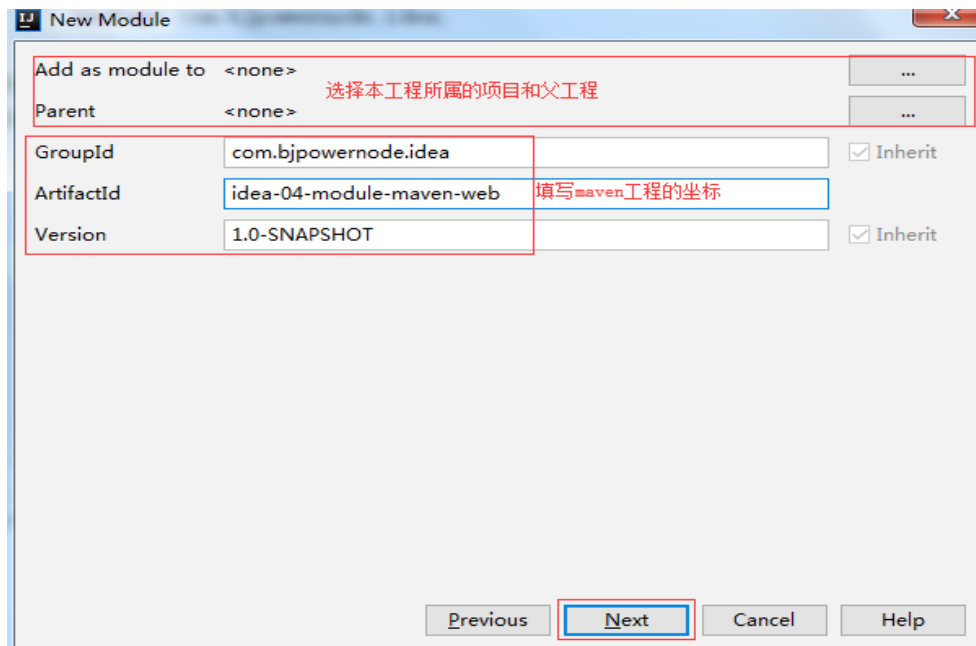
## 3.3 IDEA 创建 Maven 版 web 工程

### 3.3.1 创建 Maven 版 web 工程

File-->New-->Module...:

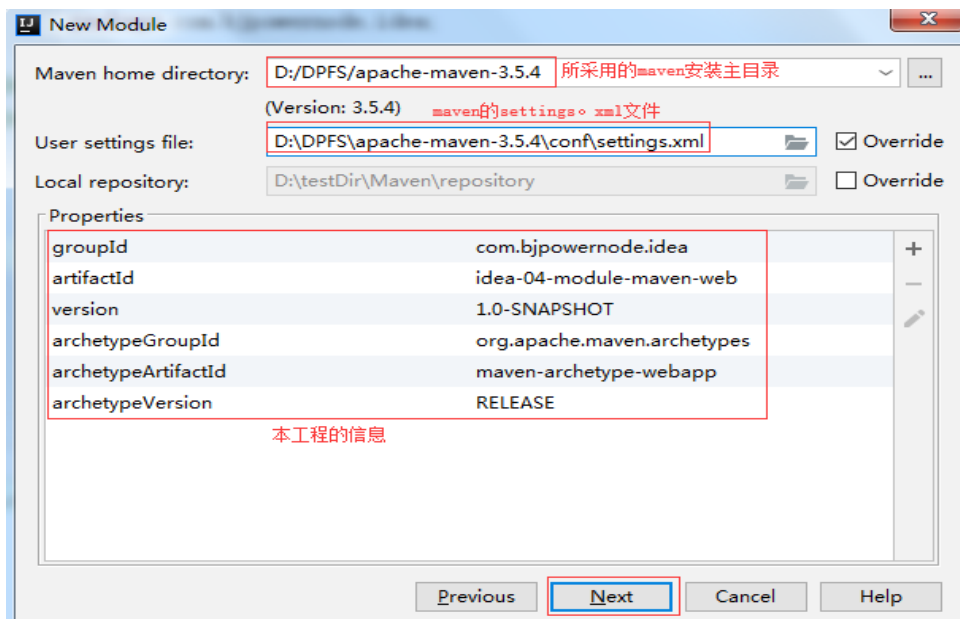


### 3.3.2 设置 module 信息

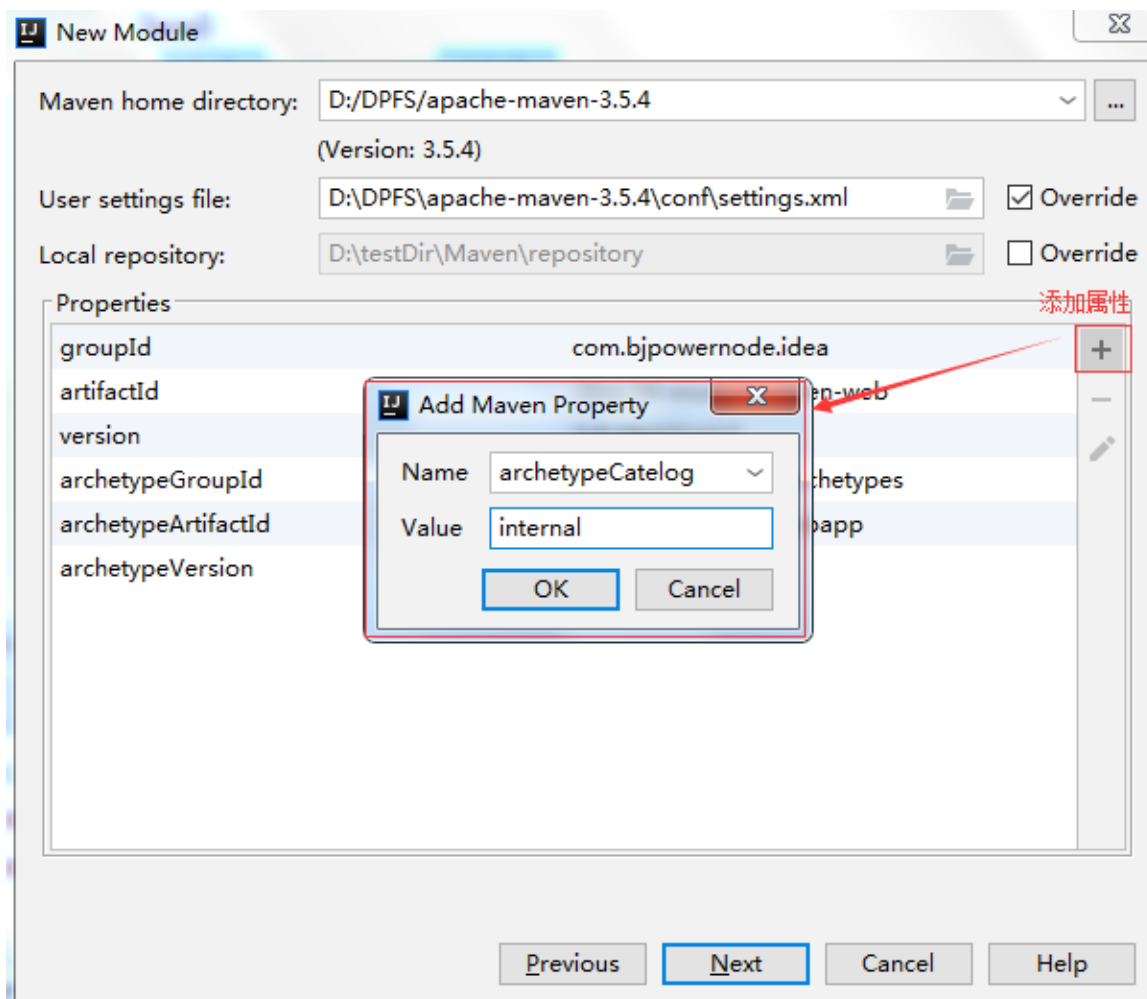




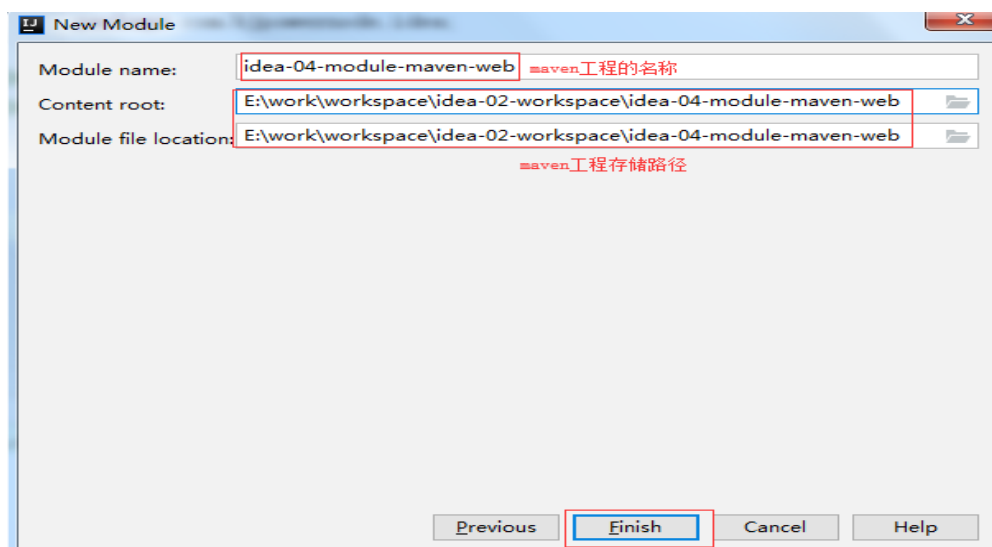
### 3.3.3 设置所使用的 maven:



这个窗口基本上不用修改什么，但是这样会比较慢，有时候如果网速不好，就会卡的比较久，这是因为 maven 这个骨架会从远程仓库加载 archetype 元数据，但是 archetype 又比较多，所以比较卡，这时候可以加个属性 archetypeCatalog = internal，表示仅使用内部元数据：

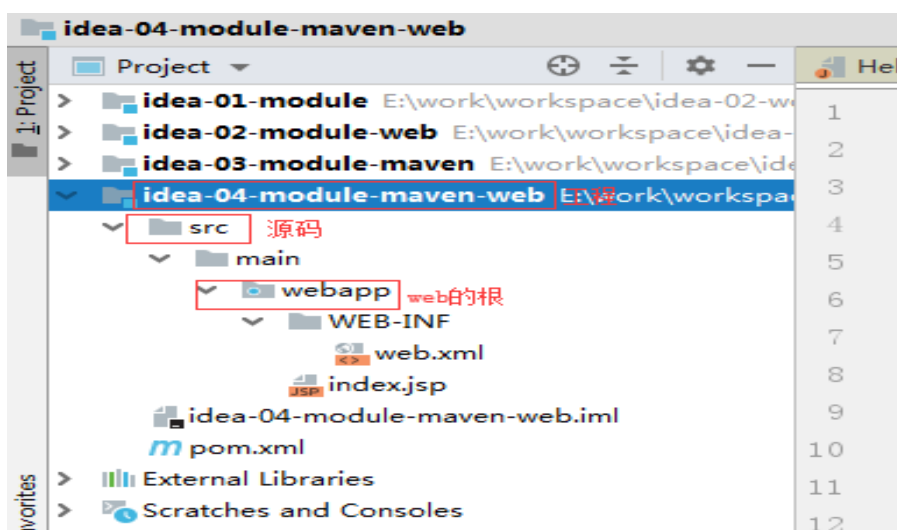


### 3.3.4 填写 maven 工程名称和存储路径



由于要运行 archetype 程序，所以这个过程需要几分钟的时间，当控制台出现“BUILD SUCCESS”时，表示工程创建完成。

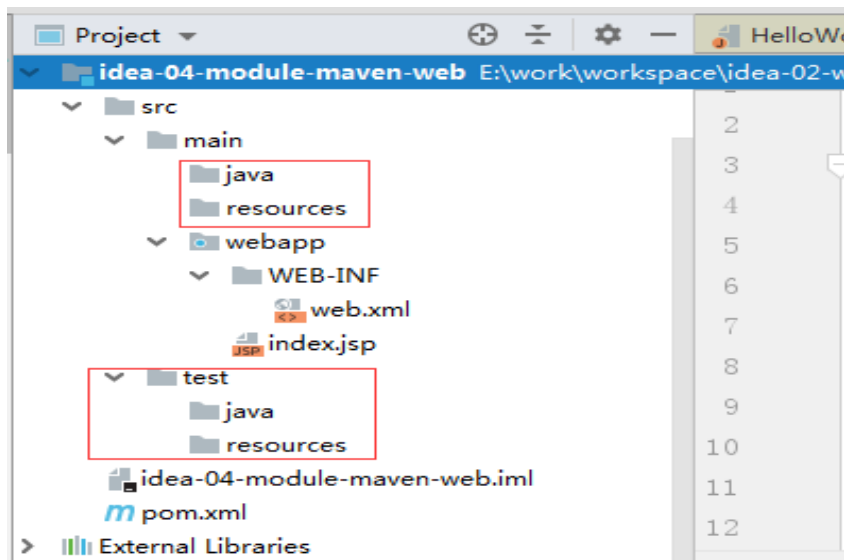
### 3.3.5 创建后视图



显然，按照 maven archetype 原型创建的 maven web 工程缺少 maven 项目的完整结构：src-main-java / resources，src-test-java/resources，所以需要我们手动添加文件目录。

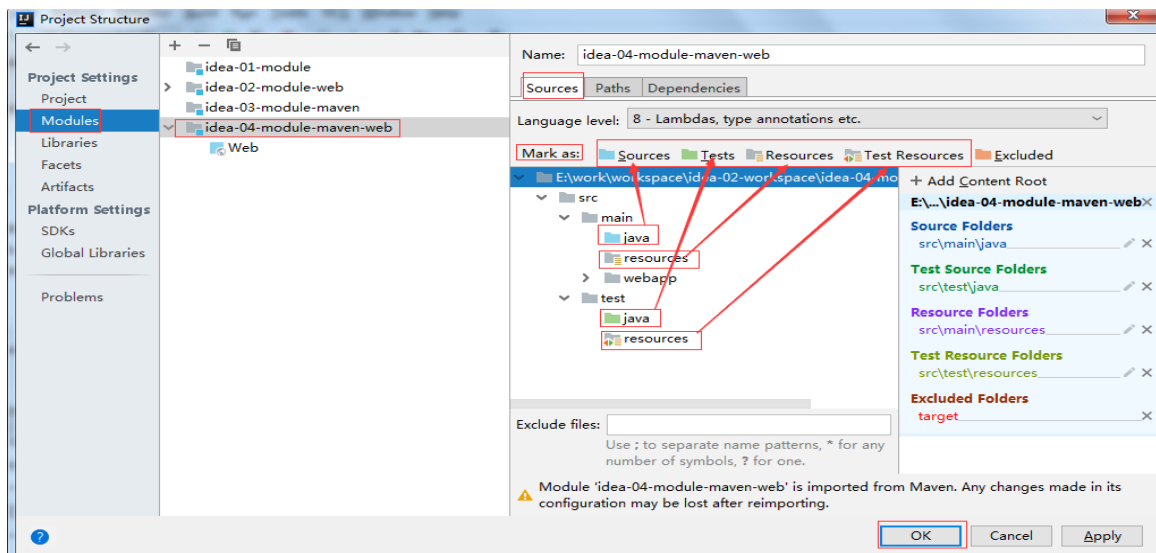
### 3.3.6 创建缺省文件夹

src-main-java / resources，src-test-java/resources



### 3.3.7 把文件夹标识为源码文件夹

File -> Project Structure, 选择 Modules: 右边找到 java 这层机构, 在上面有个“Mask as”, 点下 Sources, 表示这里面是源代码类。



### 3.3.8 pom.xml 添加依赖

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.1</version>
  <scope>provided</scope>
</dependency>
```

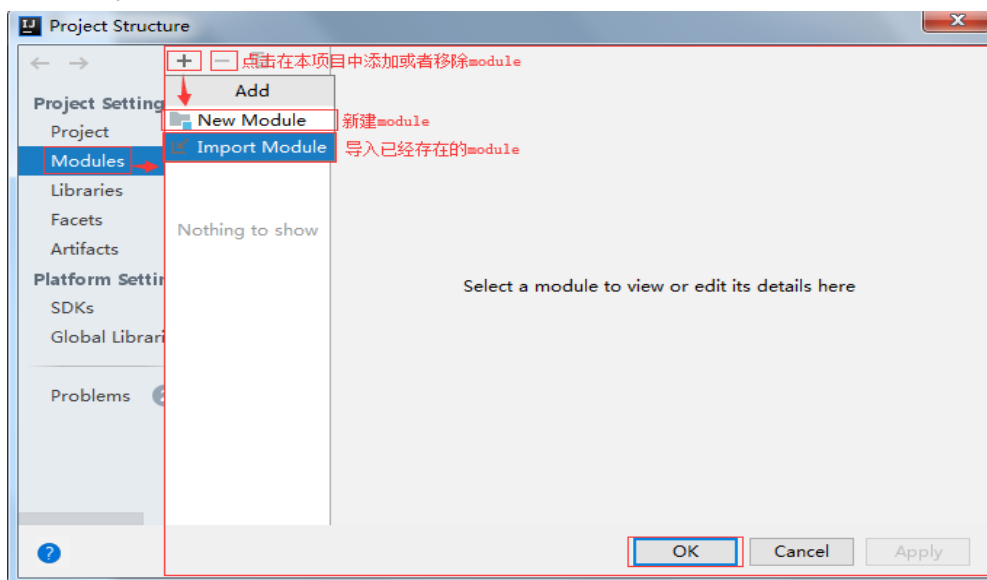
### 3.3.9 编写测试类 HelloWorld 和测试页面 index.jsp

## 3.4 IDEA 中导入 Maven 工程(module)

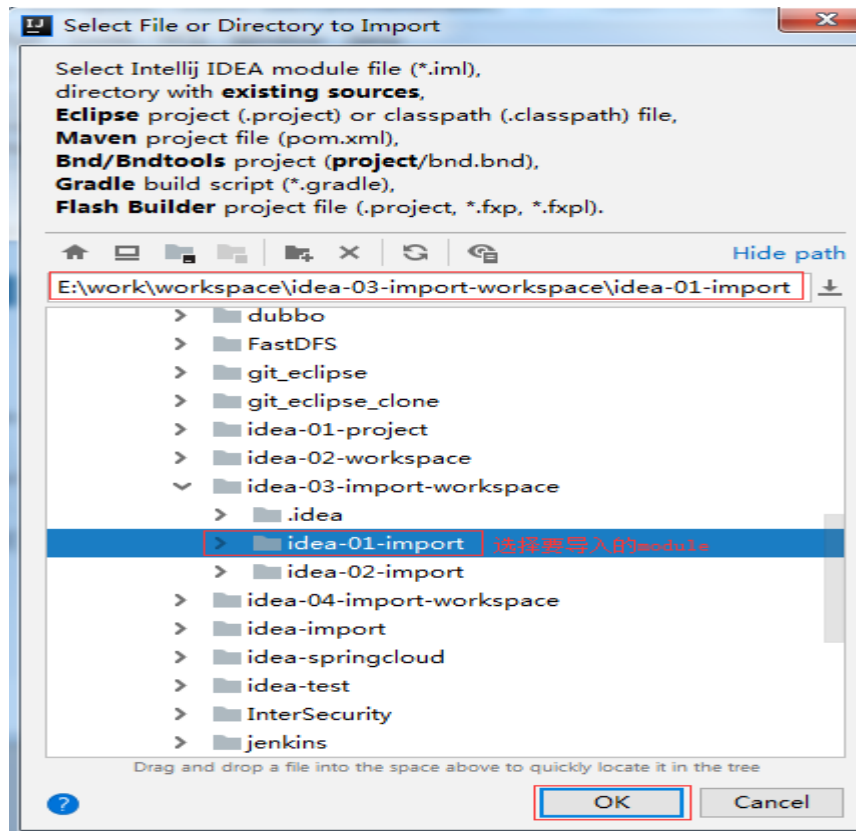
### 3.4.1 新建一个空的 project 作为工作空间

### 3.4.2 在项目结构中导入或移除 module

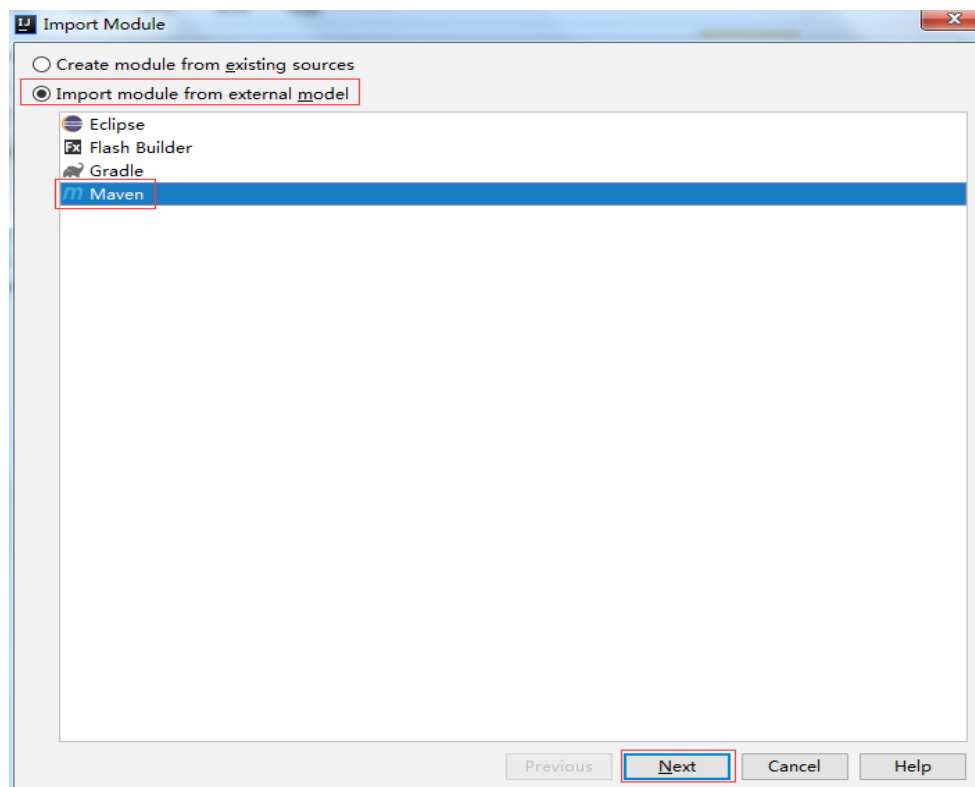
File-->Project Structure...



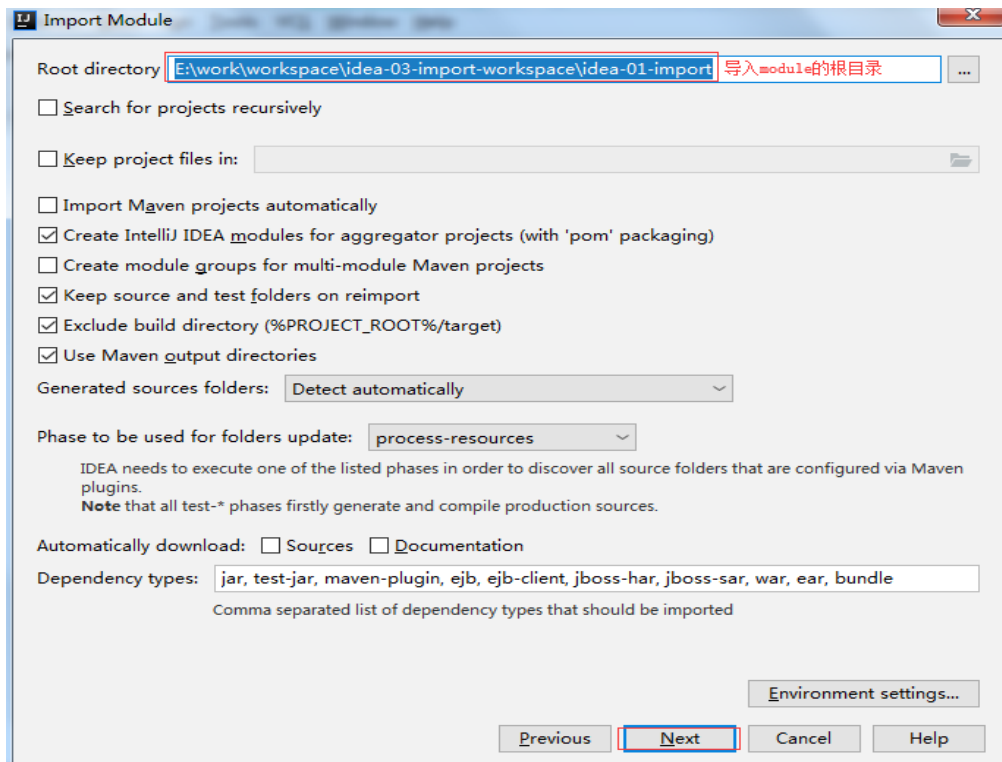
### 3.4.3 选择要导入的 Module:



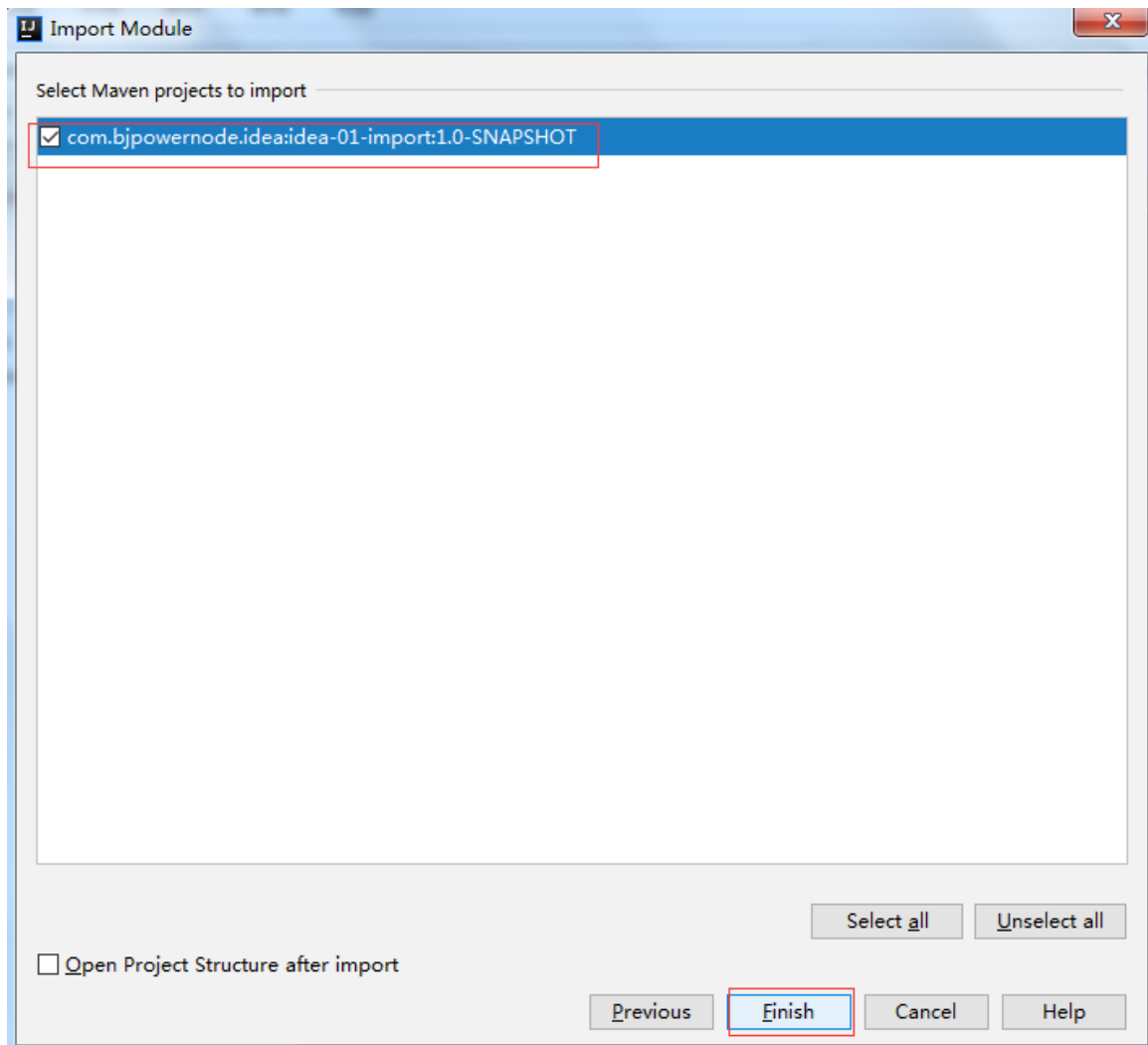
### 3.4.4 选择导入方式



### 3.4.5 选择要导入的项目

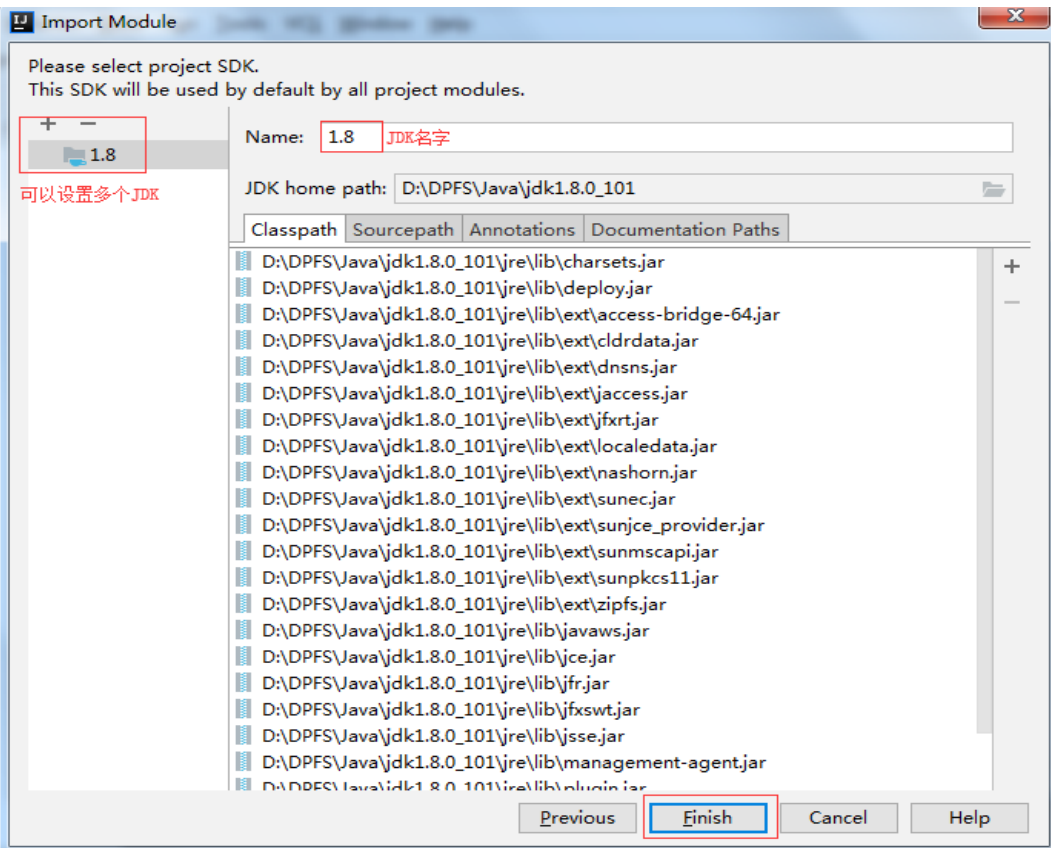


### 3.4.6 选择要导入的 maven 工程

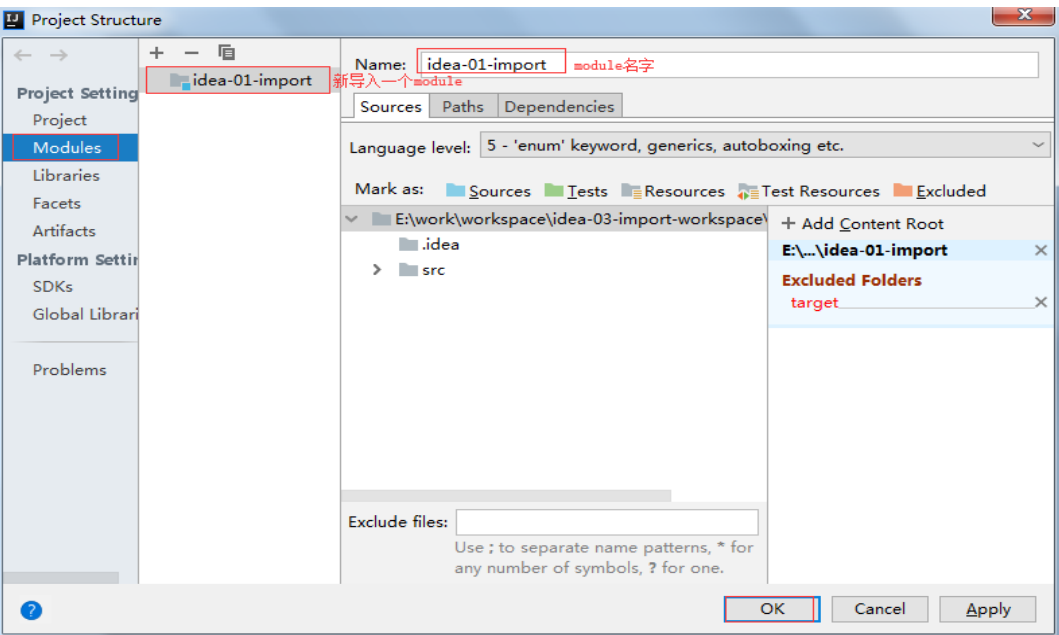




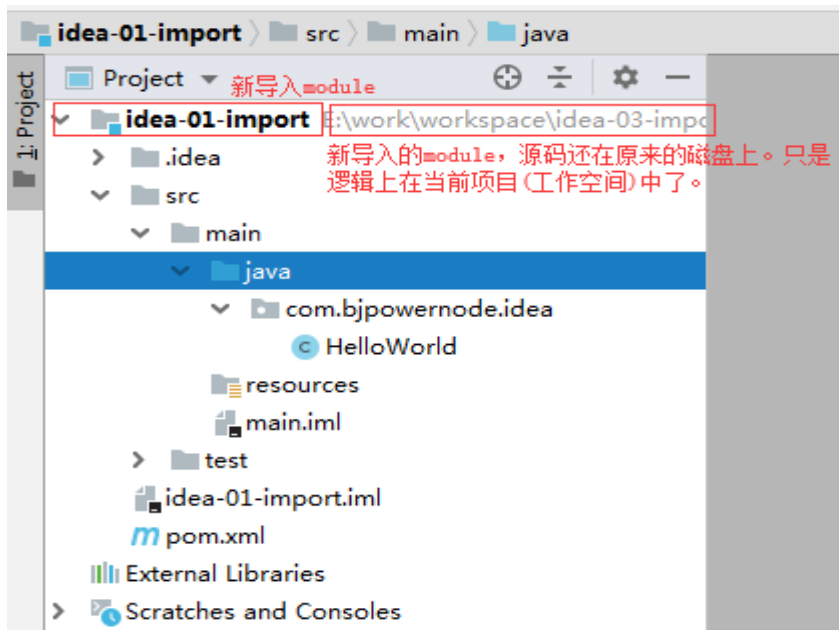
### 3.4.7 设置导入 module 所使用的 JDK



### 3.4.8 导入后项目结构



### 3.4.9 导入后视图



## 4 章 依赖管理

### 4.1 依赖的范围

依赖的范围：compile、test、provided，默认采用 compile

	compile	test	provided
对主程序是否有效	是	否	是
对测试程序是否有效	是	是	是
是否参与打包	是	否	否
是否参与部署	是	否	否

## 5 章 Maven 常用设置

### 5.1 全局变量

在 Maven 的 pom.xml 文件中，<properties>用于定义全局变量，POM 中通过\${property\_name}的形式引用变量的值。  
定义全局变量：

```
<properties>
    <spring.version>4.3.10.RELEASE</spring.version>
</properties>
```

引用全局变量：

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

Maven 系统采用的变量：

```
<properties>
    <maven.compiler.source>1.8</maven.compiler.source> 源码编译 jdk 版本
    <maven.compiler.target>1.8</maven.compiler.target> 运行代码的 jdk 版本
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> 项目构建使用的编码，避免中文乱
码
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding> 生成报告的编码
</properties>
```

## 5.2 指定资源位置

src/main/java 和 src/test/java 这两个目录中的所有 \*.java 文件会分别在 compile 和 test-compile 阶段被编译，编译结果分别放到了 target/classes 和 target/test-classes 目录中，但是这两个目录中的其他文件都会被忽略掉，如果需要把 src 目录下的文件包放到 target/classes 目录，作为输出的 jar 一部分。需要指定资源文件位置。以下内容放到 <build> 标签中。

```
<build>
    <resources>
        <resource>
            <directory>src/main/java</directory><!--所在的目录-->
            <includes><!--包括目录下的.properties,.xml 文件都会扫描到-->
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <!--filtering 选项 false 不启用过滤器， *.property 已经起到过滤的作用了 -->
            <filtering>false</filtering>
        </resource>
    </resources>
</build>
```