

第一讲 Maven

一、Maven 介绍

1. 什么是 Maven

一个对 Maven 比较正式的定义是这么说的：Maven 是一个项目管理工具，它包含了一个项目对象模型 (POM: Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标 (goal)的逻辑。

2. Maven 能解决什么问题

项目开发不仅仅是写写代码，期间会伴随着各种必不可少的事情要做，比如：

- 我们需要引用各种 jar 包，尤其是比较大的工程，引用的 jar 包往往有几十个乃至上百个，每用到一种 jar 包，都需要手动引入工程目录，而且经常遇到各种让人抓狂的 jar 包冲突，版本冲突。
- 世界上没有不存在 bug 的代码，为了减少 bug，因此写完代码，还要写一些单元测试，然后一个个的运行来检验代码质量。
- 写完代码后，需要把代码与各种配置文件、资源整合到一起，定型打包，如果是 web 项目，还需要将之发布到服务器。

Maven 就可以解决上面所提到的这些问题，能帮你构建工程，管理 jar 包，编译代码，还能帮你自动运行单元测试，打包，生成报表，甚至能帮你部署项目，生成 Web 站点。

3. Maven 的两个经典作用

① 依赖管理

Maven 的一个核心特性就是依赖管理。当我们涉及到多模块的项目（包含成百个模块或者子项目），管理依赖就变成一项困难的任务。Maven 展示出了它对处理这种情形的高度控制。传统的 WEB 项目中，我们必须将工程所依赖的 jar 包复制到工程中，导致了工程的变得很大。那么 maven 工程是如何使得工程变得很少呢？

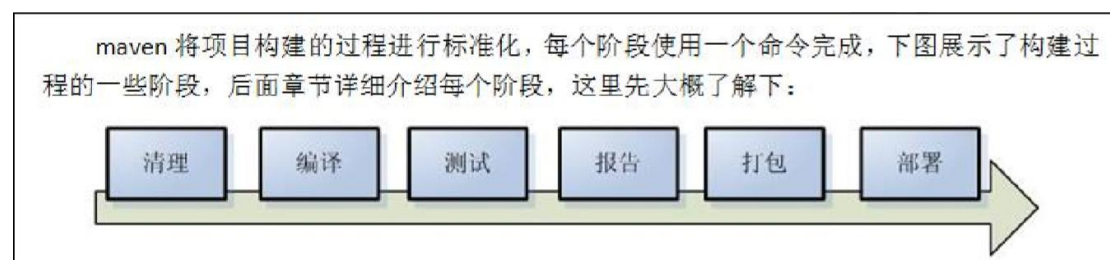
maven 工程中不直接将 jar 包导入到工程中，而是通过在 pom.xml 文件中添加所需 jar 包的坐标，这样就很好的避免了 jar 直接引入进来，在需要用到 jar 包的时候，只要查找 pom.xml 文件，再通过 pom.xml 文件中的坐标，到一个专门用于“存放 jar 包的仓库”（maven 仓库）中根据坐标从而找到这些 jar 包，再把这些 jar 包拿去运行。

② 项目的一键构建

项目，往往都要经历编译、测试、运行、打包、安装、部署等一系列过程。

什么是构建？指的是项目从编译、测试、运行、打包、安装，部署整个过程都交给 maven 进行管理，这个过程称为构建。一键构建指的是整个构建过程，使用 maven 一个命令可以轻松完成整个工作。

Maven 规范化构建流程如下：



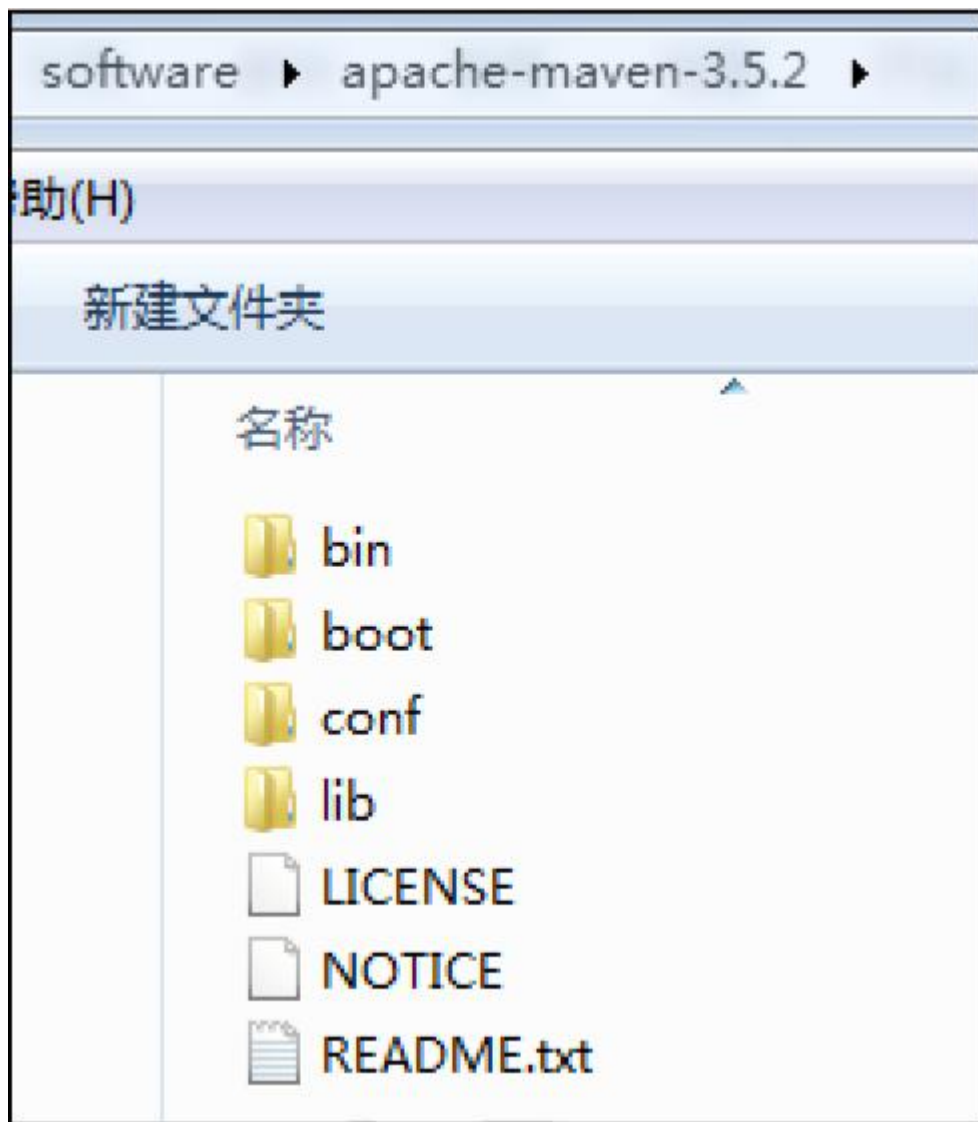
二、Maven 的使用

1. Maven 的安装

① Maven 软件的下载

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

Maven 下载后，将 Maven 解压到一个没有中文没有空格的路径下，比如 D:\software\maven 下面。解压后目录结构如下：



bin:存放了 maven 的命令

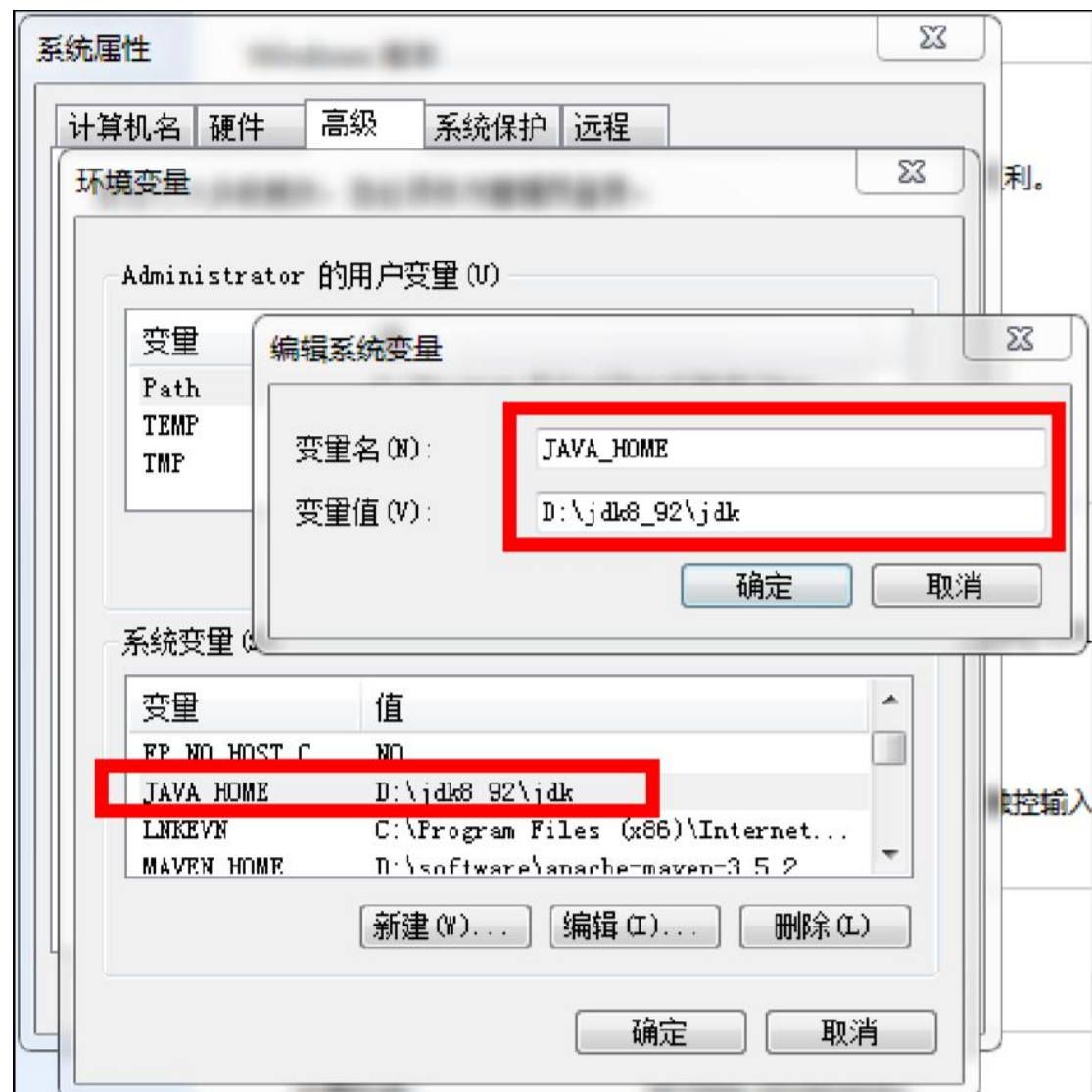
boot:存放了一些 maven 本身的引导程序，如类加载器等

conf:存放了 maven 的一些配置文件，如 setting.xml 文件

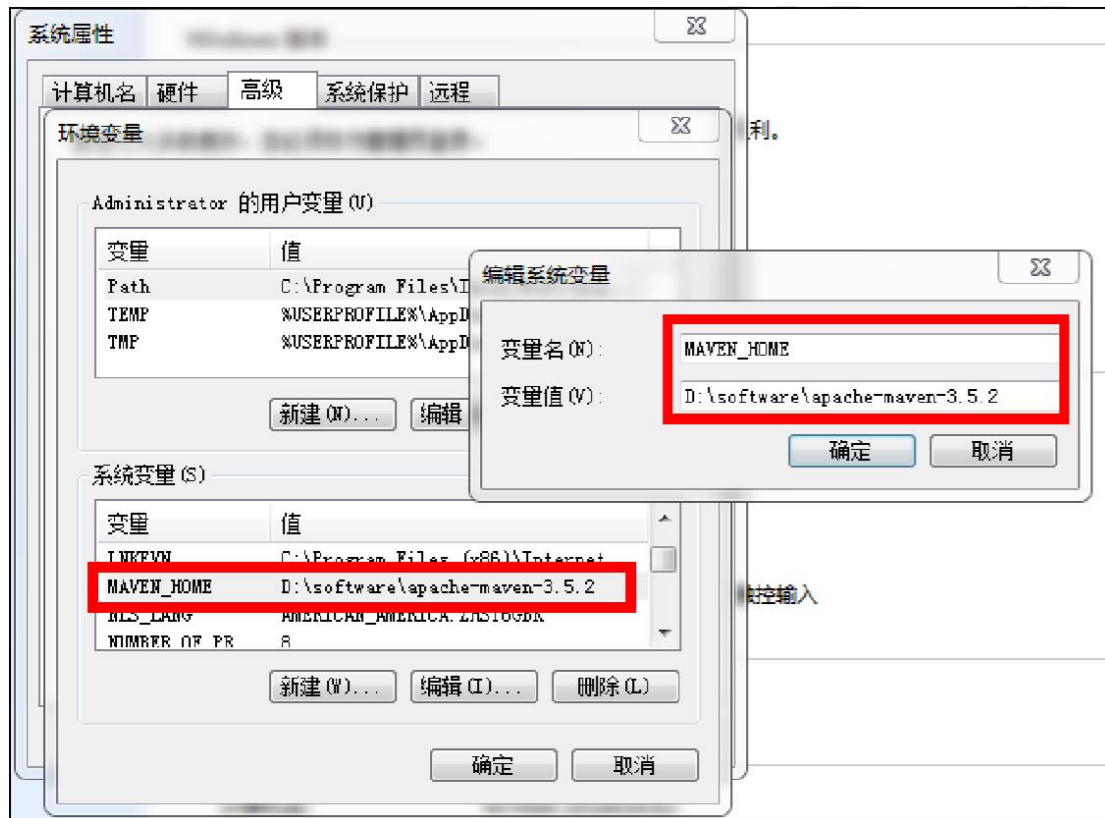
lib:存放了 maven 本身运行所需的一些 jar 包

② Maven 配置

检查 JDK 的安装目录



配置 MAVEN_HOME ，变量值就是你的 maven 安装 的路径（bin 目录之前一级目录）



③ Maven 软件版本测试

找开 cmd 命令，输入 mvn -v 命令，如下图：

```
C:\Users\Administrator>mvn -v
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d; 2017-10-18T15:58:13+08:00)
Maven home: D:\software\apache-maven-3.5.2\bin\..
Java version: 1.8.0_92 vendor: Oracle Corporation
Java home: D:\jdk8_92\jdk\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

2. Maven 仓库

① Maven 仓库的分类

maven 的工作需要从仓库下载一些 jar 包，如下图所示，本地的项目 A、项目 B 等都会通过 maven 软件从远程仓库（可以理解为互联网上的仓库）下载 jar 包并存在本地仓库，本地仓库就是本地文件夹，当第二次需要此 jar 包时则不再从远程仓库下载，因为本地仓库已经存在了，可以将本地仓库理解为缓存，有了本地仓库就不用每次从远程仓库下载了。



- 本地仓库：用来存储从远程仓库或中央仓库下载的插件和 jar 包，项目使用一些插件或 jar 包，优先从本地仓库查找。默认本地仓库位置在 `${user.dir}/.m2/repository`，`${user.dir}` 表示 windows 用户目录。



- 远程仓库：如果本地需要插件或者 jar 包，本地仓库没有，默认去远程仓库下载。远程仓库可以在互联网内也可以在局域网内。
- 中央仓库：在 maven 软件中内置一个远程仓库地址 `http://repo1.maven.org/maven2`，它是中央仓库，服务于整个互联网，它是由 Maven 团队自己维护，里面存储了非常全的 jar 包，它包含了世界上大部分流行的开源项目构件。

② Maven 本地仓库的配置

在 `MAVE_HOME/conf/settings.xml` 文件中配置本地仓库位置 (maven 的安装目录下)：
打开 `settings.xml` 文件，配置如下：

```

46 <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
47     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
48     xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
49   <!-- localRepository
50        | The path to the local repository maven will use to store artifacts.
51        |
52        | Default: ${user.home}/.m2/repository
53   <localRepository>path/to/local/repo</localRepository>
54   -->
55   <localRepository>D:/repository</localRepository>

```

D:\repository目录是maven本地仓库所在的目录

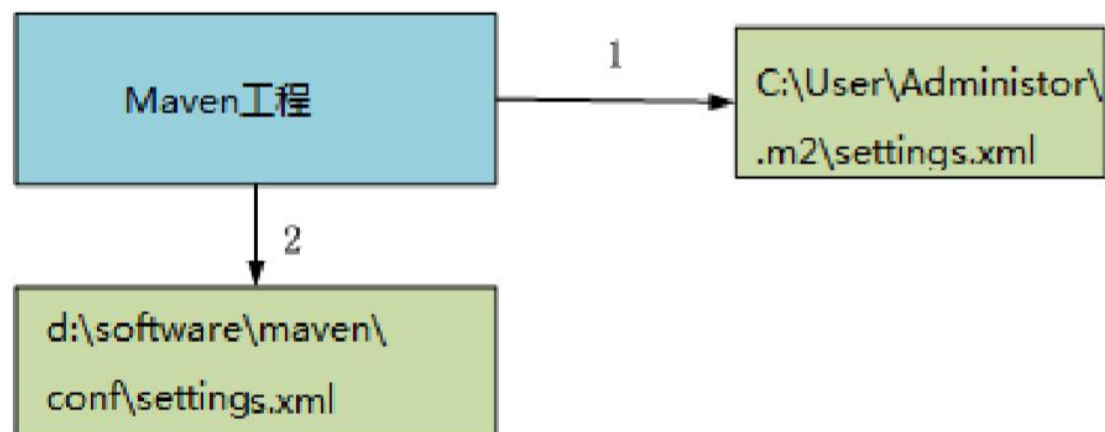
③ 全局 setting 与用户 setting

maven 仓库地址、私服等配置信息需要在 setting.xml 文件中配置，分为全局配置和用户配置。

在 maven 安装目录下的有 conf/setting.xml 文件，此 setting.xml 文件用于 maven 的所有 project 项目，它作为 maven 的全局配置。

如需要个性配置则需要在用户配置中设置，用户配置的 setting.xml 文件默认的位置在：
\${user.dir} /.m2/settings.xml 目录中，\${user.dir} 指 windows 中的用户目录。

maven 会先找用户配置，如果找到则以用户配置文件为准，否则使用全局配置文件。

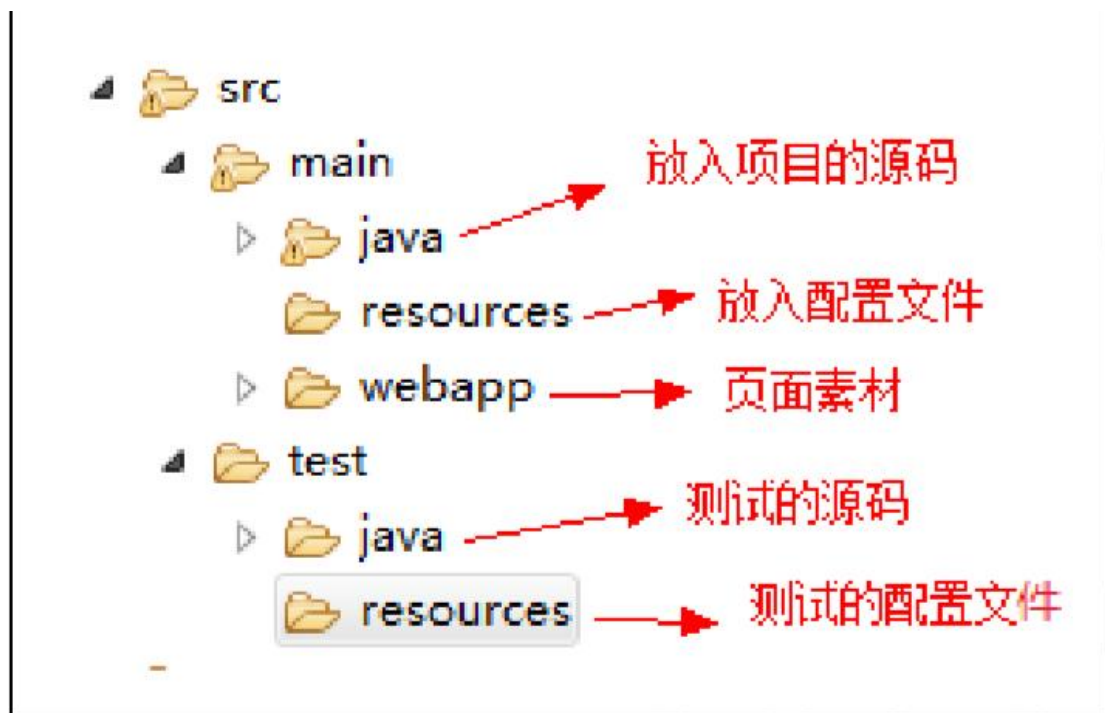


3. Maven 工程的认识

① Maven 工程的目录结构



作为一个 maven 工程，它的 `src` 目录和 `pom.xml` 是必备的。进入 `src` 目录后，我们发现它里面的目录结构如下：



`src/main/java` —— 存放项目的 `.java` 文件

`src/main/resources` —— 存放项目资源文件，如 `spring`, `hibernate` 配置文件

`src/test/java` —— 存放所有单元测试 `.java` 文件，如 `JUnit` 测试类

src/test/resources —— 测试资源文件

target —— 项目输出位置，编译后的 class 文件会输出到此目录

pom.xml——maven 项目核心配置文件

注意：如果是普通的 java 项目，那么就没有 webapp 目录

三、Maven 常用命令

1. 常用命令

- compile

compile 是 maven 工程的编译命令，作用是将 src/main/java 下的文件编译为 class 文件输出到 target 目录下。

- test

test 是 maven 工程的测试命令 mvn test,会执行 src/test/java 下的单元测试类。cmd 执行 mvn test 执行 src/test/java 下单元测试类，下图为测试结果，运行 1 个测试用例，全部成功。

- clean

clean 是 maven 工程的清理命令，执行 clean 会删除 target 目录及内容。

- package

package 是 maven 工程的打包命令，对于 java 工程执行 package 打成 jar 包，对于 web 工程打成 war 包。

- install

install 是 maven 工程的安装命令，执行 install 将 maven 打成 jar 包或 war 包发布到本地仓库。从运行结果中，可以看出：当后面的命令执行时，前面的操作过程也都会自动执行，

2. Maven 指令的生命周期

maven 对项目构建过程分为三套相互独立的生命周期，请注意这里说的是“三套”，而且“相互独立”，这三套生命周期分别是：

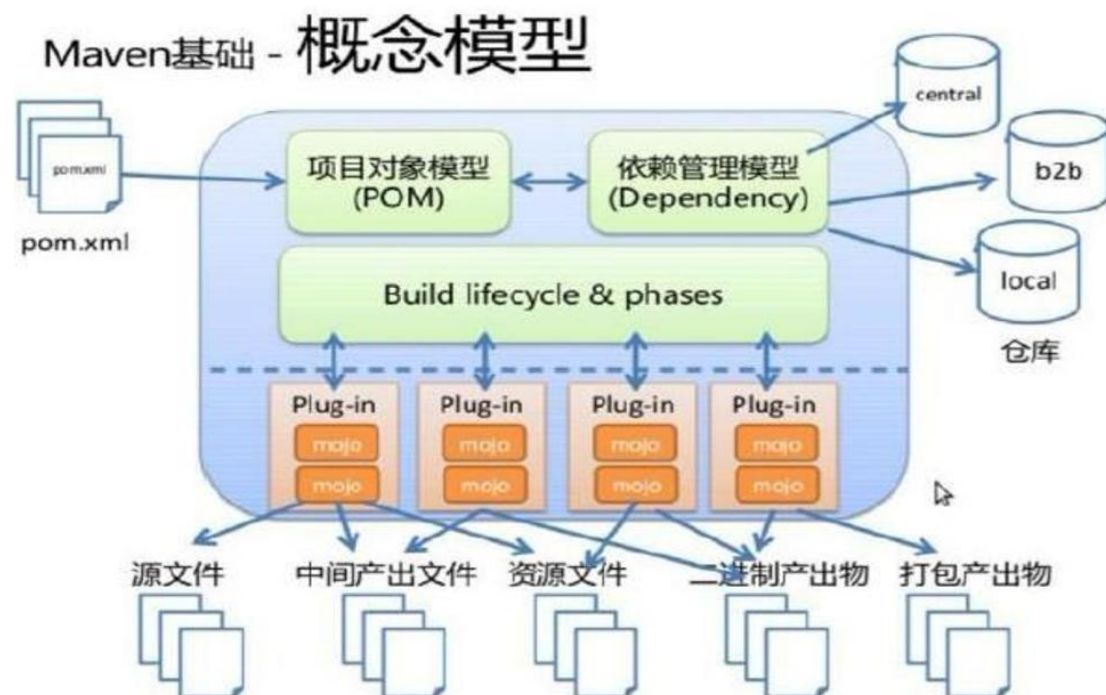
Clean Lifecycle 在进行真正的构建之前进行一些清理工作。

Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。

Site Lifecycle 生成项目报告，站点，发布站点。

3. maven 的概念模型

Maven 包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段 (phase)中插件(plugin)目标(goal)的逻辑。



- 项目对象模型 (Project Object Model)

一个 maven 工程都有一个 pom.xml 文件，通过 pom.xml 文件定义项目的坐标、项目依赖、项目信息、插件目标等。

- 依赖管理系统(Dependency Management System)

通过 maven 的依赖管理对项目所依赖的 jar 包进行统一管理。

比如：项目依赖 junit4.9，通过在 pom.xml 中定义 junit4.9 的依赖即使用 junit4.9，如下所示是 junit4.9 的依赖定义：

```
<!-- 依赖关系 -->
<dependencies>
  <!-- 此项目运行使用 junit，所以此项目依赖 junit -->
  <dependency>
    <!-- junit 的项目名称 -->
    <groupId>junit</groupId>
    <!-- junit 的模块名称 -->
    <artifactId>junit</artifactId>
    <!-- junit 版本 -->
    <version>4.9</version>
    <!-- 依赖范围：单元测试时使用 junit -->
    <scope>test</scope>
  </dependency>
```

- 一个项目生命周期(Project Lifecycle)

使用 maven 完成项目的构建，项目构建包括：清理、编译、测试、部署等过程，maven 将这些过程规范为一个生命周期，如下所示是生命周期的各各阶段：



maven 通过执行一些简单命令即可实现上边生命周期的各各过程，比如执行 mvn compile 执行编译、执行 mvn clean 执行清理。

- 一组标准集合

maven 将整个项目管理过程定义一组标准，比如：通过 maven 构建工程有标准的目录结构，有标准的生命周期阶段、依赖管理有标准的坐标定义等。

- 插件(plugin)目标(goal) maven

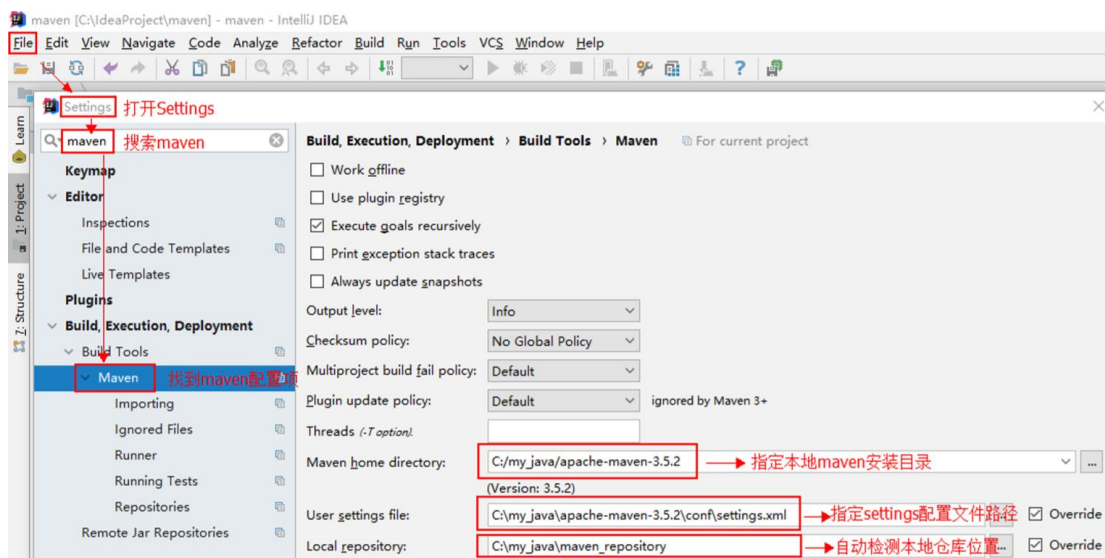
管理项目生命周期过程都是基于插件完成的。

四、idea 开发 maven 项目

1. idea 的 maven 配置

打开 File->Settings 配置 maven

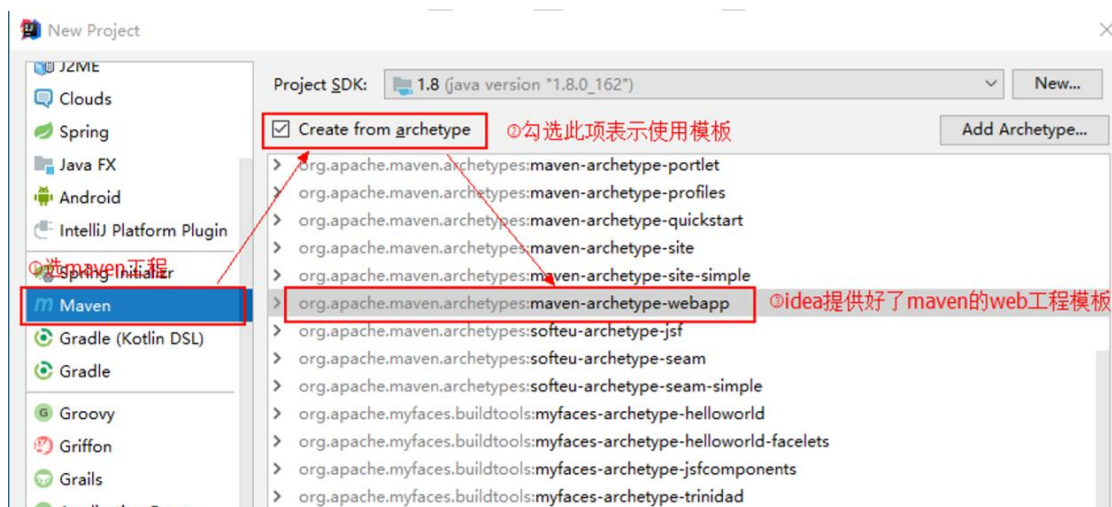
依据图片指示，选择本地 maven 安装目录，指定 maven 安装目录下 conf 文件夹中 settings 配置文件。



2. 创建一个 maven 的 web 工程

打开 idea，选择创建一个新工程

选择 idea 提供好的 maven 的 web 工程模板



点击 Next 填写项目信息

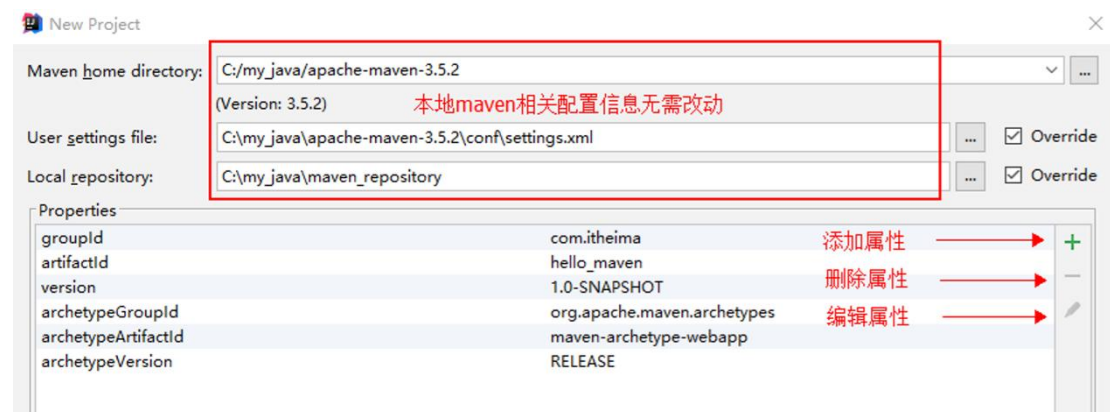


The 'New Project' dialog box shows the following fields:

- GroupId: com.itheima (公司或组织的名称)
- ArtifactId: hello_maven (项目名)
- Version: 1.0-SNAPSHOT (版本号)

There are 'Inherit' checkboxes for GroupId and Version.

点击 Next, 此处不做改动。



The 'New Project' dialog box shows the following fields:

- Maven home directory: C:\my_java\apache-maven-3.5.2 (Version: 3.5.2) (本地maven相关配置信息无需改动)
- User settings file: C:\my_java\apache-maven-3.5.2\conf\settings.xml
- Local repository: C:\my_java\maven_repository

There are 'Override' checkboxes for the User settings file and Local repository.

Properties table:

Property	Value	Action
groupId	com.itheima	添加属性
artifactId	hello_maven	删除属性
version	1.0-SNAPSHOT	编辑属性
archetypeGroupId	org.apache.maven.archetypes	
archetypeArtifactId	maven-archetype-webapp	
archetypeVersion	RELEASE	

点击 Next 选择项目所在目录

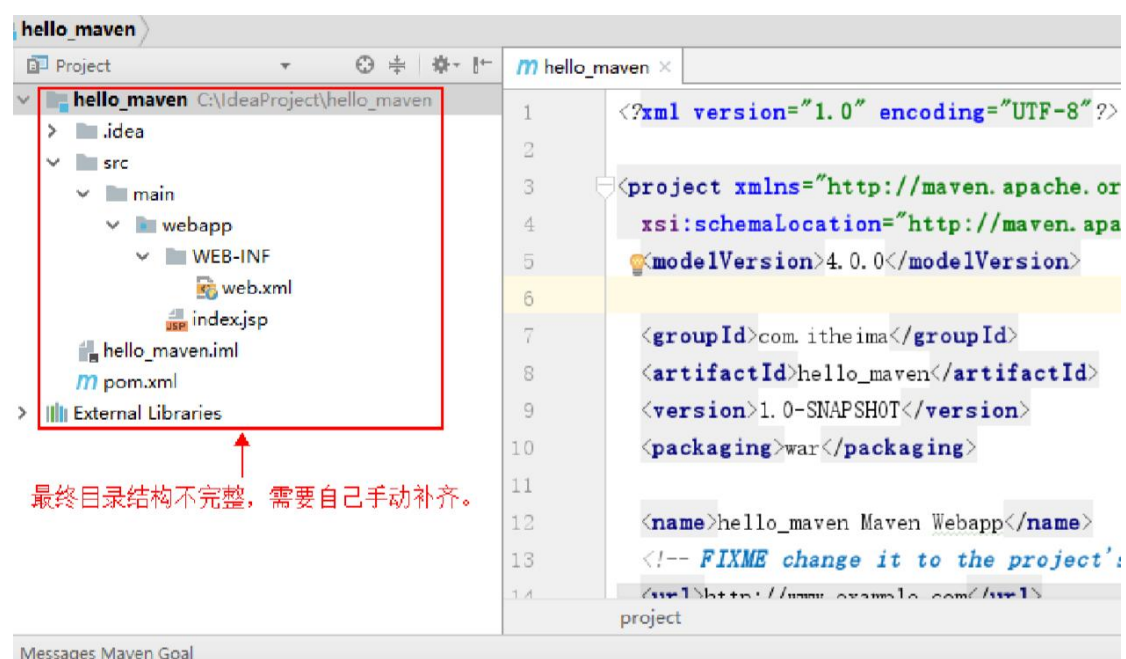


The 'New Project' dialog box shows the following fields:

- Project name: hello_maven
- Project location: C:\IdeaProject\hello_maven

A red arrow points to the '...' button next to the Project location field, with the text: 点击右侧按钮选择项目所在目录

点击 Finish 后开始创建工程, 耐心等待, 直到出现如下界面。



The IDE interface shows the project structure and the pom.xml file.

Project Structure:

- hello_maven (C:\IdeaProject\hello_maven)
- src
 - main
 - webapp
 - WEB-INF
 - web.xml
 - index.jsp
- hello_maven.iml
- pom.xml

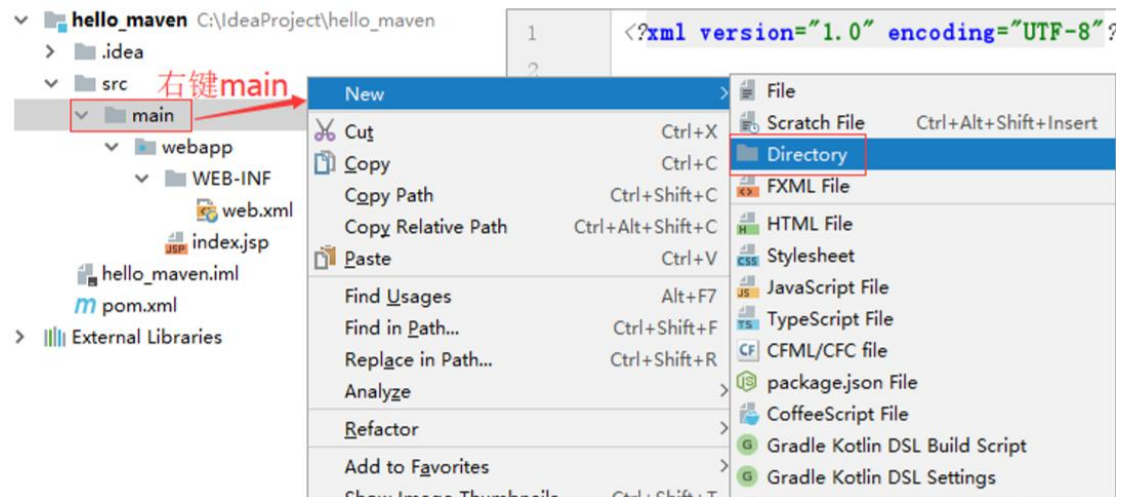
Final directory structure is incomplete, needs to be manually supplemented.

pom.xml content:

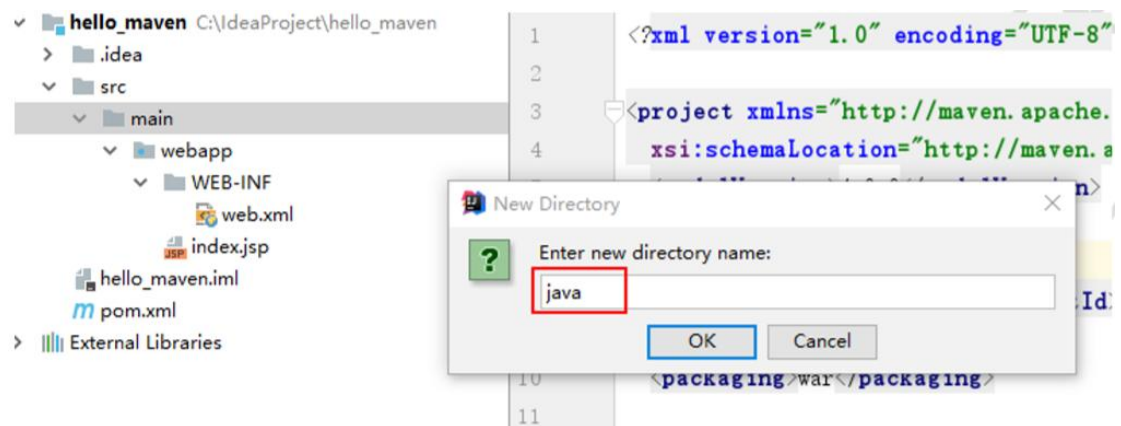
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/xsi:schemaLocation="http://maven.apache.org/xsi:schemaLocation"
<modelVersion>4.0.0</modelVersion>
<groupId>com.itheima</groupId>
<artifactId>hello_maven</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
<name>hello_maven Maven Webapp</name>
<!-- FIXME change it to the project's
<url>http://www.example.com/</url>
project
```

```
Messages Maven Goal
[INFO] Project created from Archetype in dir: C:\Users\hasee\AppData\Local\Temp\archetypetm
[INFO] -----
[INFO] BUILD SUCCESS 控制台项目创建成功提示信息
[INFO] -----
[INFO] Total time: 7.555 s
[INFO] Finished at: 2018-05-10T11:27:50+08:00
```

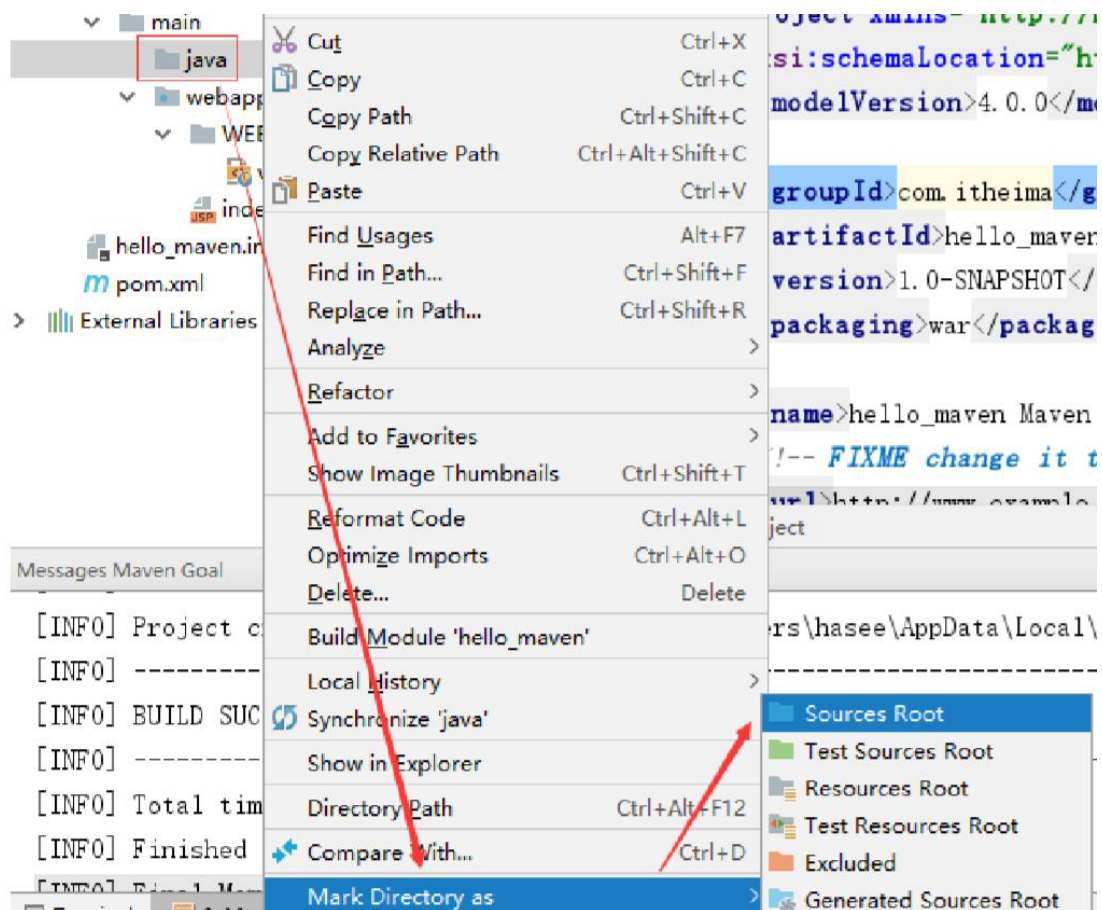
手动添加 src/main/java 目录，如下图右键 main 文件夹 New Directory



创建一个新的文件夹命名为 java



点击 OK 后，在新的文件夹 java 上右键 -> Make Directory as -> Sources Root



3. 在 pom.xml 文件添加坐标

① 工程坐标定义

每个 maven 工程都需要定义本工程的坐标, 坐标是 maven 对 jar 包的身份定义, 比如:

<!--项目名称, 定义为组织名+项目名, 类似包名-->

```
<groupId>club.semicircle</groupId>
```

<!-- 模块名称 -->

```
<artifactId>hello_maven</artifactId>
```

<!-- 当前项目版本号, snapshot 为快照版本即非正式版本, release 为正式发布版本 -->

```
<version>0.0.1-SNAPSHOT</version>
```

<packaging> : 打包类型

jar: 执行 package 会打成 jar 包

war: 执行 package 会打成 war 包

pom : 用于 maven 工程的继承, 通常父工程设置为 pom

② 坐标的来源方式

添加依赖需要指定依赖 jar 包的坐标, 但是很多情况我们是不知道 jar 包的坐标, 可以通过如下方式查询:

从互联网搜索

<http://search.maven.org/>

<http://mvnrepository.com/>

③ 依赖范围

A 依赖 B, 需要在 A 的 pom.xml 文件中添加 B 的坐标, 添加坐标时需要指定依赖范围, 依赖范围包括:

依赖范围	对于编译 classpath 有效	对于测试 classpath 有效	对于运行时 classpath 有效	例子
compile	Y	Y	Y	spring-core
test	-	Y	-	Junit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC驱动
system	Y	Y	-	本地的, Maven仓库之 外的类库

- compile: 编译范围, 指 A 在编译时依赖 B, 此范围为默认依赖范围。编译范围的依赖会用在 编译、测试、运行, 由于运行时需要所以编译范围的依赖会被打包。
- provided: provided 依赖只有在当 JDK 或者一个容器已提供该依赖之后才使用, provided 依赖在编译和测试时需要, 在运行时不需要, 比如: servlet api 被 tomcat 容器提供。

- runtime: runtime 依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如：jdbc 的驱动包。由于运行时需要所以 runtime 范围的依赖会被打包。
- test: test 范围依赖 在编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用， 比如：junit。由于运行时不需要所以 test 范围依赖不会被打包。
- system: system 范围依赖与 provided 类似，但是你必须显式的提供一个对于本地系统中 JAR 文件的路径，需要指定 systemPath 磁盘路径，system 依赖不推荐使用。
-

在 maven-web 工程中测试各 scop，总结：

默认引入的 jar 包 ----- compile 【默认范围 可以不写】(编译、测试、运行 都有效)
 servlet-api 、jsp-api ----- provided (编译、测试 有效，运行时无效 防止和 tomcat 下 jar 冲突)

jdbc 驱动 jar 包 ---- runtime (测试、运行 有效)

junit ----- test (测试有效)

依赖范围由强到弱的顺序是：compile>provided>runtime>test

④ 项目中添加的坐标

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

设置 jdk 编译版本

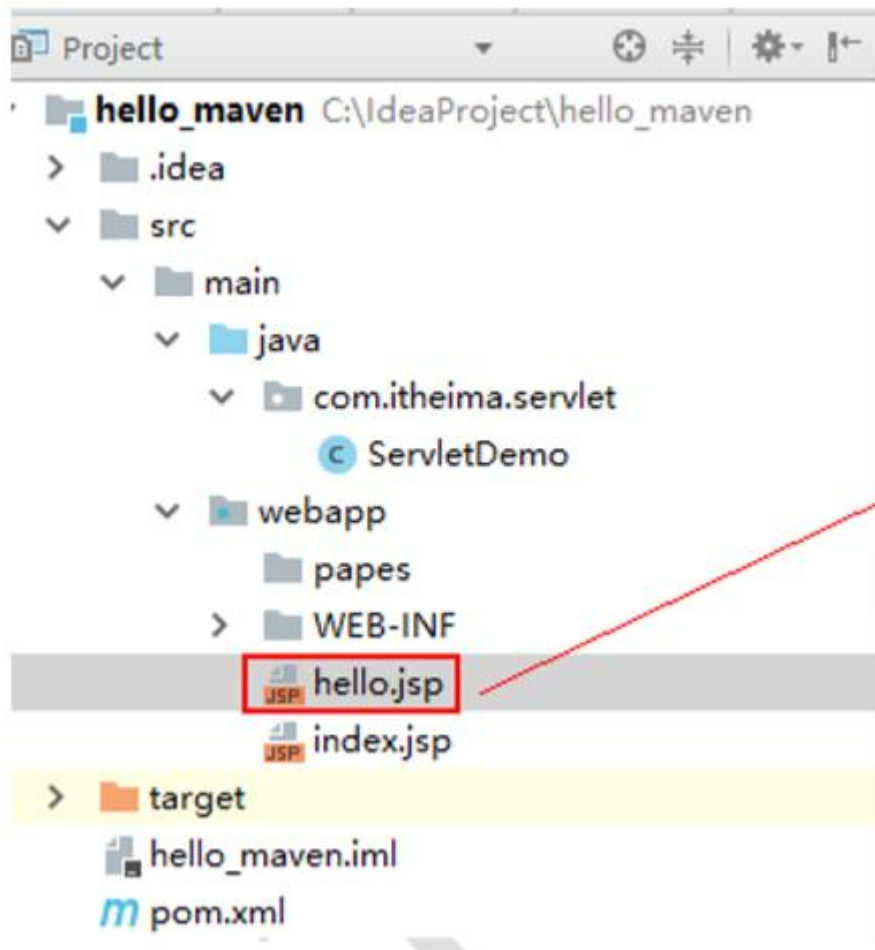
本教程使用 jdk1.8, 需要设置编译版本为 1.8, 这里需要使用 maven 的插件来设置: 在 pom.xml 中加入:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
  </plugins>
</build>
```

4. 编写 servlet

在 src/main/java 中创建 ServletTest

5. 编写 jsp



6. 在 web.xml 中配置 servlet 访问路径

7. 添加 tomcat7 插件

在 pom 文件中添加如下内容

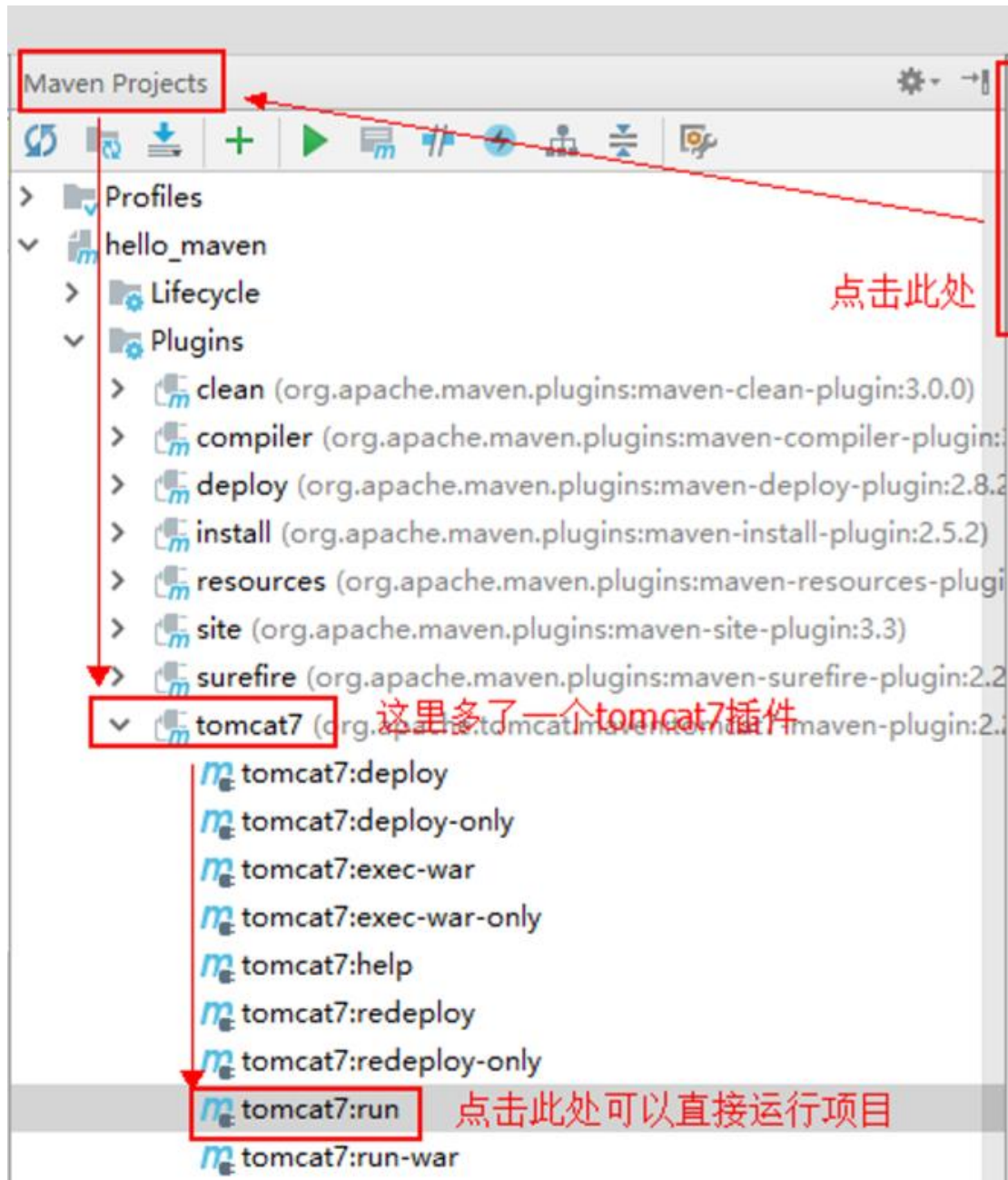
```
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.2</version>
  <configuration>
    <port>8080</port>
    <path>/</path>
  </configuration>
</plugin>
```

端口号

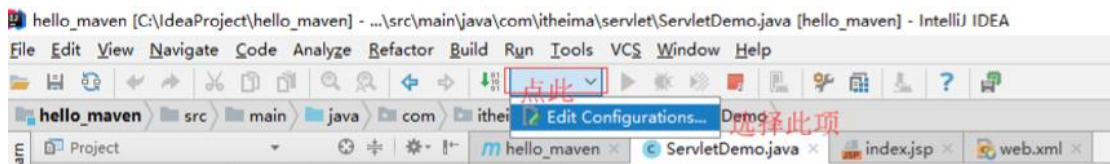
访问路径

8. 运行

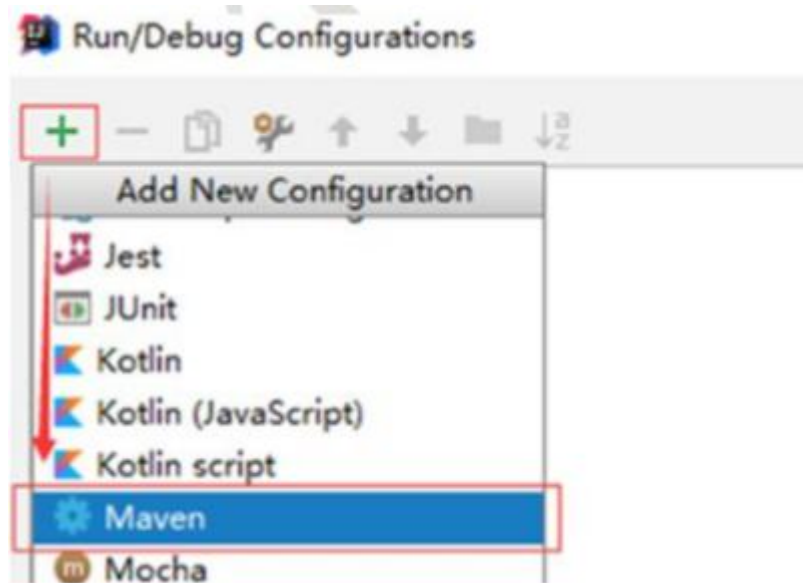
点击 idea 最右侧 Maven Projects， 就可以看到我们新添加的 tomcat7 插件 双击 tomcat7 插件下 tomcat7:run 命令直接运行项目



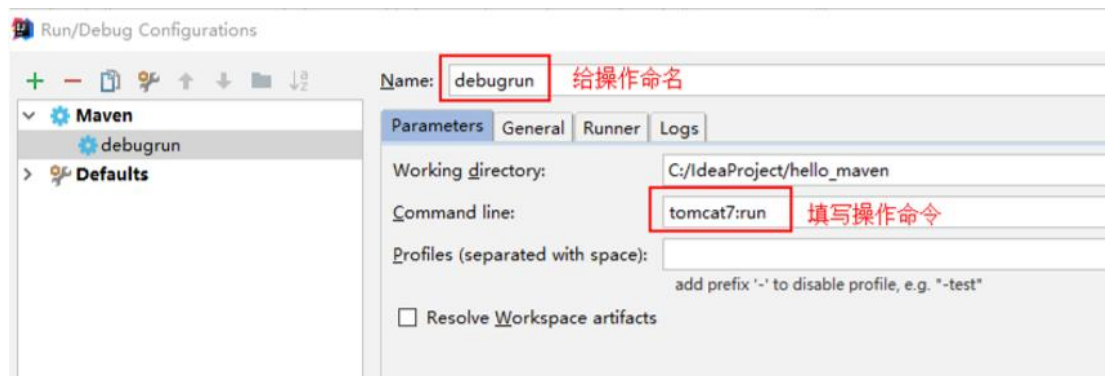
maven 工程运行调试



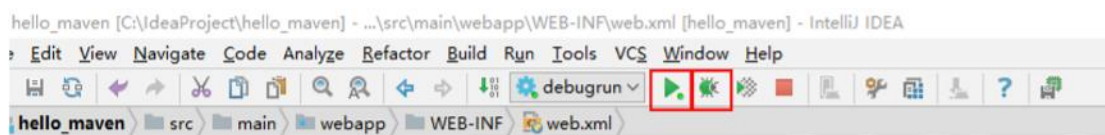
在弹出框中点击如图加号按钮找到 maven 选项



在弹出窗口中填写如下信息



完成后先 Apply 再 OK 结束配置后，可以在主界面找到我们刚才配置的操作名称。



如上图红框选中的两个按钮，左侧是正常启动，右侧是 debug 启动