

1. Maven 概述

1.1. Maven 简介

Maven 翻译为“专家”、“内行”；是一个采用纯 Java 编写的开源项目管理工具，Maven 采用了一种被称之为 **Project Object Model (POM)** 概念来管理项目，所有的项目配置信息都被定义在一个叫做 **POM.xml** 的文件中，通过该文件 Maven 可以管理项目的整个声明周期，包括清除、编译、测试、报告、打包、部署等。目前 Apache 下绝大多数项目都已经采用 Maven 进行管理。而 Maven 本身还支持多种插件，可以方便更灵活的控制项目，开发人员的主要任务应该是关注业务逻辑并去实现它，而不是把时间浪费在学习如何在不同的环境中去依赖 jar 包、项目部署等。Maven 正是为了将开发人员从这些任务中解脱出来而诞生的一个项目管理工具。

其官网地址为：<http://maven.apache.org>

1.1.1. Maven 是什么

Maven 是基于项目对象模型(POM Project Object Model)，可以通过一小段描述信息（配置文件）来管理项目的构建、报告和文档的软件**项目管理工具**。

1.1.2. Maven 功能

Maven 是跨平台的项目管理工具。主要服务于基于 Java 平台的项目构建，依赖管理和项目信息管理。

- 什么是项目构建？



- 什么是理想的项目构建？

高度自动化，跨平台，可重用的组件，标准化的

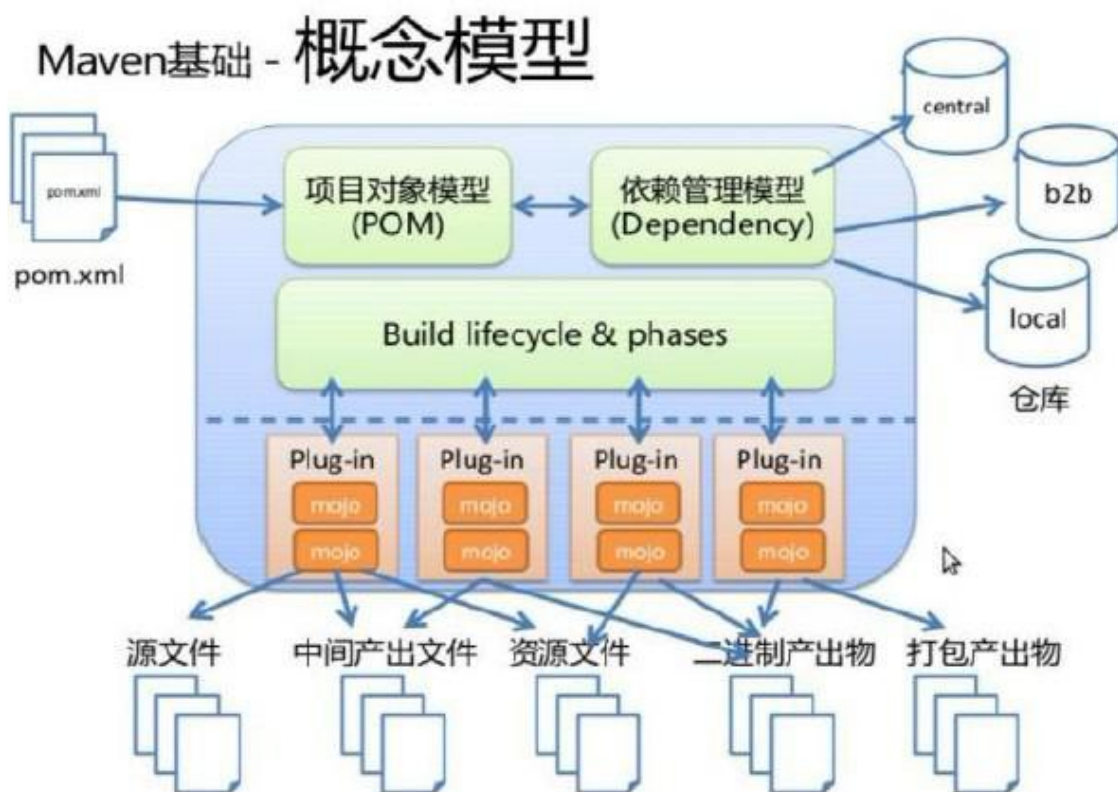
- 什么是依赖？为什么要进行依赖管理？

自动下载，统一依赖管理

- 有哪些项目信息？

项目名称描述，开发人员信息等

1.2. Maven 模型



1.3. 为什么使用 Maven

- 使用开发工具？

手工操作较多，编译、测试、部署等工作都是独立的，很难一步完成
每个人的 IDE 配置都不同，很容易出现本地代码换个地方编译就出错

- 使用 Ant？

没有一个约定的目录结构

必须明确让 ant 做什么，什么时候做，然后编译，打包

没有生命周期，必须定义目标及其实现的任务序列

没有集成依赖管理

● 使用 Maven?

拥有约定，知道你的代码在哪里，放到哪里去

拥有一个生命周期，例如执行 `mvn install` 就可以自动执行编译，测试，打包等构建过程

只需要定义一个 `pom.xml`，然后把源码放到默认的目录，Maven 帮你处理其他事情

拥有依赖管理，仓库管理

● 传统方式管理 jar 依赖的问题：

1) jar 冲突

2) jar 依赖

3) jar 体积过大

4) jar 在不同阶段无法个性化配置

● 使用 maven 方式管理 jar 依赖的好处：

1) 解决 jar 冲突

2) 解决 jar 依赖问题

3) jar 文件不用在每个项目保存，只需要放在仓库即可

4) maven 可以指定 jar 的依赖范围

2. Maven 安装与配置

环境要求：Maven 3.3+ 需要使用 jdk 1.7+

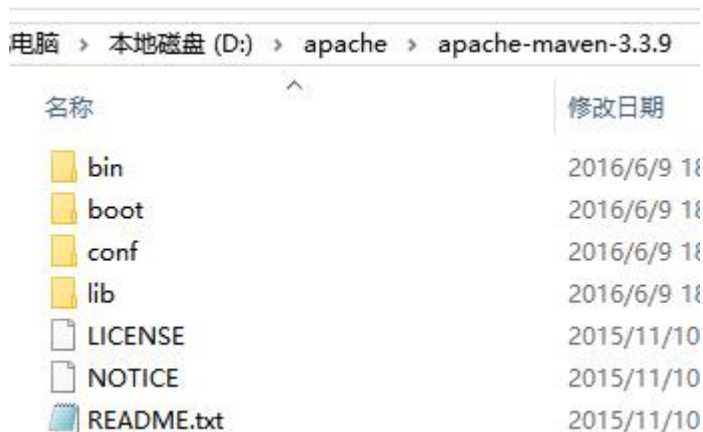
2.1. 下载

下载地址：<http://maven.apache.org/download.cgi>

2.2. 安装与配置

2.2.1. 解压文件

解压 maven 压缩包“apache-maven-3.3.9-bin.zip”到一个路径（尽量编码路径中不要包含中文）



名称	修改日期
bin	2016/6/9 18
boot	2016/6/9 18
conf	2016/6/9 18
lib	2016/6/9 18
LICENSE	2015/11/10
NOTICE	2015/11/10
README.txt	2015/11/10

2.2.2. 了解 Maven 目录

bin: 含有 mvn 运行的脚本

boot: 含有 plexus-classworlds 类加载器框架

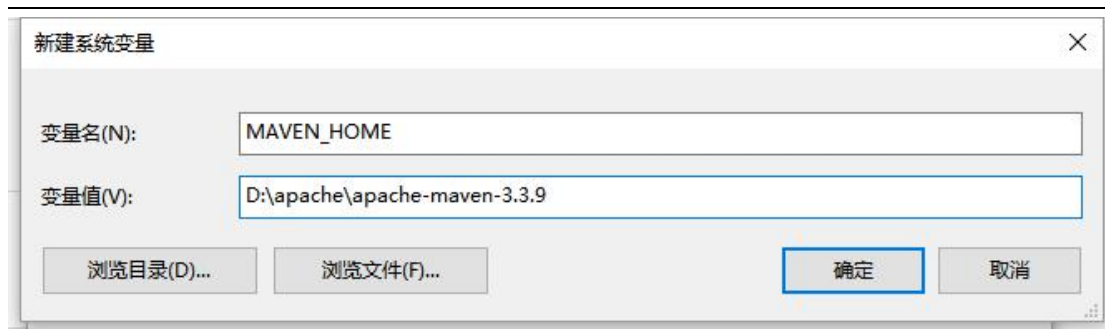
lib: 含有 Maven 运行时所需要的 java 类库

conf: 含有 settings.xml 配置文件

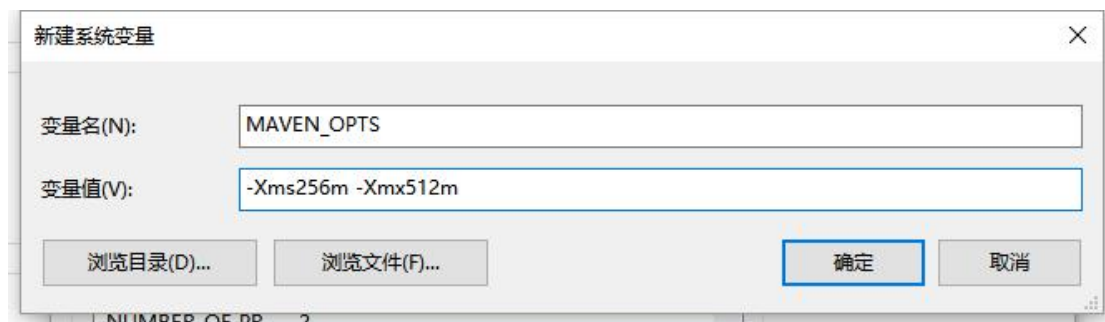
settings.xml 中默认的用户库: \${user.home}/.m2/repository[通过 maven 下载的 jar 包都会存储到此仓库中]

2.2.3. 添加系统环境变量 MAVEN_HOME

MAVEN_HOME : E:\apache-maven-3.3.9-bin（注意：配置为你自己的 maven 路径）

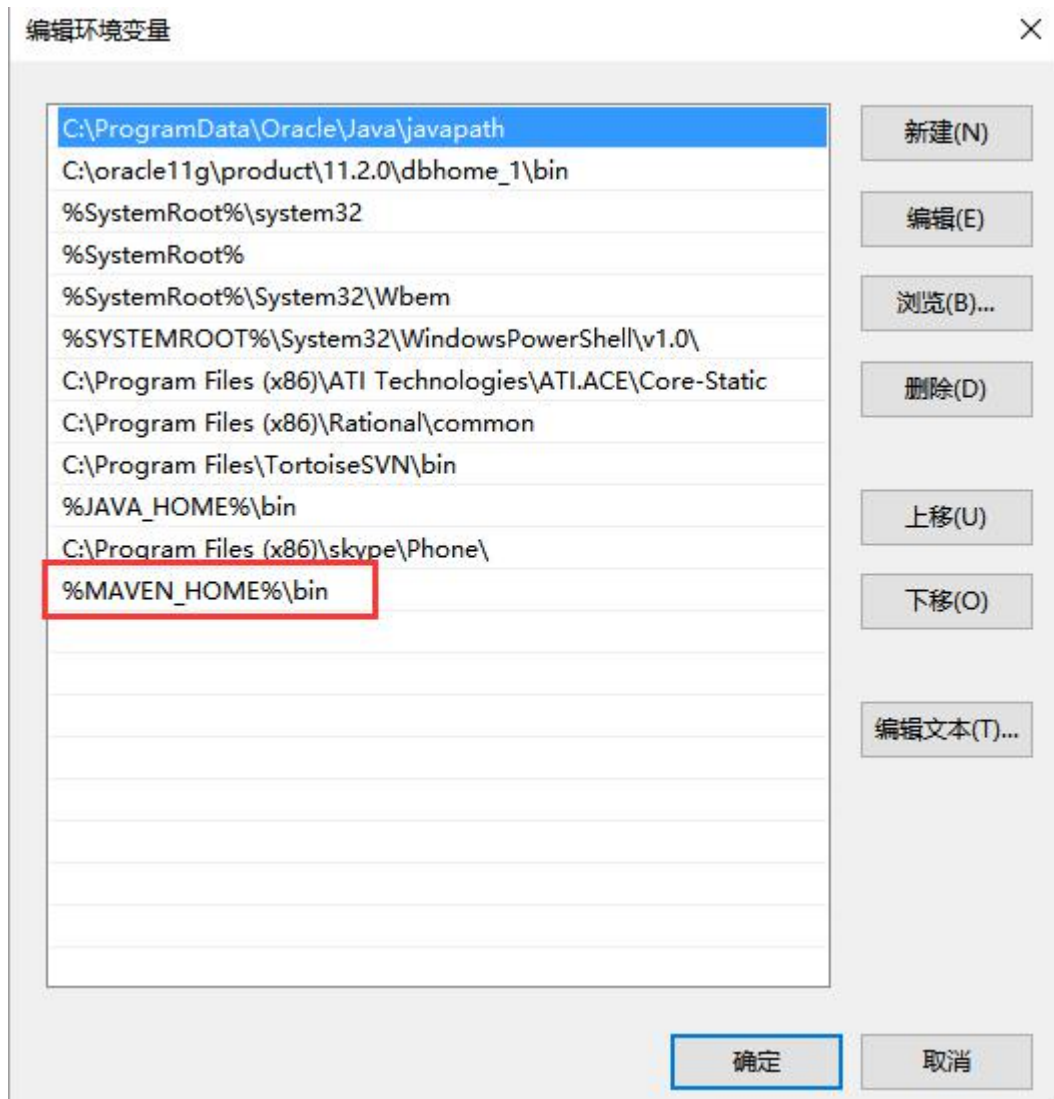


MAVEN_OPTS : -Xms256m -Xmx512m (注意: 可以不配置)



2.2.4. 设置系统环境变量 Path

在 Path 中追加: %MAVEN_HOME%\bin



2.2.5. 验证

打开 cmd 输入: mvn -version

```
C:\Users\admin>mvn -version
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-11T00:41:47+08:00)
Maven home: D:\apache\apache-maven-3.3.9\bin\..
Java version: 1.7.0_11, vendor: Oracle Corporation
Java home: D:\sun\jdk1.7.0_11\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 8", version: "6.2", arch: "amd64", family: "windows"
```

2.3. settings.xml 配置文件

节点名称	说明
localRepository	这个值是构建系统的本地仓库的路径。默认的值是 <code>\${user.home}/.m2/repository</code> 。如果一个系统想让所有登陆的用户都用同一个本地仓库的话，这个值是极其有用的。
interactiveMode	如果 Maven 要试图与用户交互来得到输入就设置为 <code>true</code> ，否则就设置为 <code>false</code> ，默认为 <code>true</code> 。
offline	如果构建系统要在离线模式下工作，设置为 <code>true</code> ，默认为 <code>false</code> 。如果构建服务器因为网络故障或者安全问题不能与远程仓库相连，那么这个设置是非常有用的。
pluginGroups	当插件的组织 Id (<code>groupId</code>) 没有显式提供时，供搜寻插件组织 Id (<code>groupId</code>) 的列表。该元素包含一个 <code>pluginGroup</code> 元素列表，每个子元素包含了一个组织 Id (<code>groupId</code>)。当我们使用某个插件，并且没有在命令行为其提供组织 Id (<code>groupId</code>) 的时候，Maven 就会使用该列表。默认情况下该列表包含了 <code>org.apache.maven.plugins</code> 。
proxies	用来配置不同的代理，多代理 <code>profiles</code> 可以应对笔记本或移动设备的工作环境：通过简单的设置 <code>profile id</code> 就可以很容易的更换整个代理配置。
servers	配置服务端的设置；一般用于设置安全认证等信息。一些设置如安全证书不应该和 <code>pom.xml</code> 一起分发。这种类型的信息应该存在于构建服务器上的 <code>settings.xml</code> 文件中。
mirrors	镜像库。确定使用的仓库；为仓库列表配置的下载镜像列表。
profiles	根据环境参数来调整构建配置；需要激活才能生效。
activeProfiles	手动激活 <code>profiles</code> 的列表【注意：必须与 <code>prifile</code> 的 <code>id</code> 一致】

2.4. 配置本地仓库

Maven 的默认本地仓库在：\${user.home}/.m2/repository；这地址可以在 settings.xml 中修改指定自定义的仓库路径。

【自定义仓库路径】

找到\${maven_home}/conf/settings.xml 文件，修改如下：

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.
  <!-- 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:\apache\apache-maven-3.3.9\repository</localRepository>
```

需要注意的是上图中的自定义路径必须存在。repository 是本地仓库，也即本地下载的 jar 存放路径，在资料中解压“repository.zip”当对应目录，然后将这个解压后路径配置到上述地址即可。（不使用解压的也是可以的，那么以后需要依赖包时需要联网自动下载。）

3. Maven 项目规约

src/main/java —— 存放项目的.java 文件（开发源代码）

src/main/resources —— 存放项目配置文件，如果没有配置文件该目录可无，如 spring, hibernate 配置文件

src/main/webapp —— 存放 web 项目资源文件（web 项目才有）

src/test/java —— 存放所有测试.java 文件（测试源代码）

src/test/resources —— 测试配置文件，如果没有配置文件该目录可无

target —— 项目输出位置（可无）

pom.xml —— maven 项目核心配置文件

也就是如果是一个 Maven 项目那么它的根目录下必定存在 src 文件夹和 pom.xml。

pom.xml 文件:

project: 任何要 build 的事物，Maven 都认为它们是工程。这些工程被定义为工程对象模型（POM，Project Object Model）。一个工程可以依赖其它工程，一个工程也可以由多个子工程构成。

POM: pom(pom.xml)是 Maven 的核心文件，它是指示 Maven 如何工作的元数据文件，类似于 Ant 的 build.xml 文件。pom.xml 文件位于每个工程的根目录下。

Plug-in: Maven 是由插件组织的，它的每一个功能都由插件提供。插件提供 goal，并根据在 pom 中找到元数据去完成工作。

4. 命令行方式构建 Maven 项目

4.1. 创建 Maven 项目

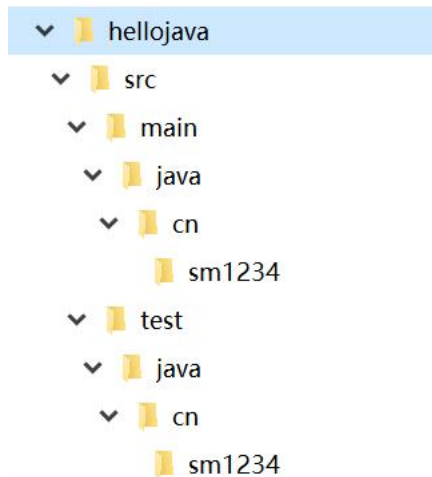
4.1.1.1. 使用命令行创建 java 项目

在命令行中可以通过 Maven 中的命令（插件）可以自动创建文件结构和自动生成 pom.xml 文件。

1) 执行 cmd 命令

```
mvn archetype:generate -DarchetypeCatalog=internal -DgroupId=cn.sm1234
-DartifactId=hellojava -DarchetypeArtifactId=maven-archetype-quickstart
-Dversion=0.0.1-snapshot
```

2) 创建结果



参数说明：

#核心命令 mvn 框架:生成 即生成 Maven 项目最基本的目录结构

mvn archetype:generate

#读取 archetype-catalog.xml 文件的位置：内置的

-DarchetypeCatalog=internal

#公司域名倒写

-DgroupId=cn.sm1234

#项目名称

-DartifactId=hellojava

#Maven 项目的模板：最简单的 Maven 项目模板

-DarchetypeArtifactId=maven-archetype-quickstart

#项目版本号，snapshot 内测版，release 正式发行版

-Dversion=0.0.1-snapshot

4.1.1.2. 使用命令行创建 web 项目

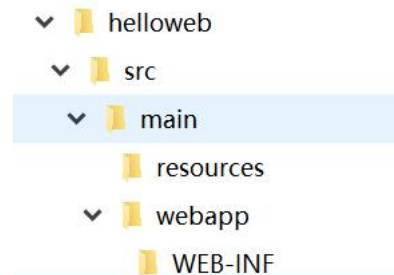
创建 web 项目时，不但创建目录结构和 pom.xml，并创建 webapp 目录放置 web 资源文件。

1) 执行 cmd 命令

```
mvn archetype:generate -DarchetypeCatalog=internal -DgroupId=cn.sm1234
```

```
-DartifactId=helloweb -DarchetypeArtifactId=maven-archetype-webapp  
-Dversion=0.0.1-snapshot
```

2) 创建结果



参数说明：

```
#核心命令 mvn 框架:生成 即生成 Maven 项目最基本的目录结构  
mvn archetype:generate  
#读取 archetype-catalog.xml 文件的位置：内置的  
-DarchetypeCatalog=internal  
#公司域名倒写  
-DgroupId=cn.sm1234  
#项目名称  
-DartifactId=helloweb  
#Maven 项目的模板：Maven web 项目模板  
-DarchetypeArtifactId=maven-archetype-webapp  
#项目版本号  
-Dversion=0.0.1-snapshot
```

4.2. 构建 Maven 项目命令的使用

4.2.1. Maven 常用命令

注意：先进入项目目录后再操作！

```
D:\simple>mvn package
```

命令	说明
mvn clean	清除原来的编译结果
mvn compile	编译
mvn test	运行测试代码；mvn test -Dtest=类名//单独运行测试类
mvn package	打包项目；mvn package -Dmaven.test.skip=true//打包时不执行测试
mvn install	将项目打包并安装到本地仓库
mvn deploy	发布到本地仓库或者服务器

4.2.2. 应用命令构建

构建需要在项目根目录下进行；首次使用 **maven** 进行构建需要连网下载对应的各个命令插件。

运用 mvn clean、mvn compile 。 。 。 mvn install 进行构建。

5. Eclipse 构建 Maven 项目

5.1. 转为 Eclipse 项目

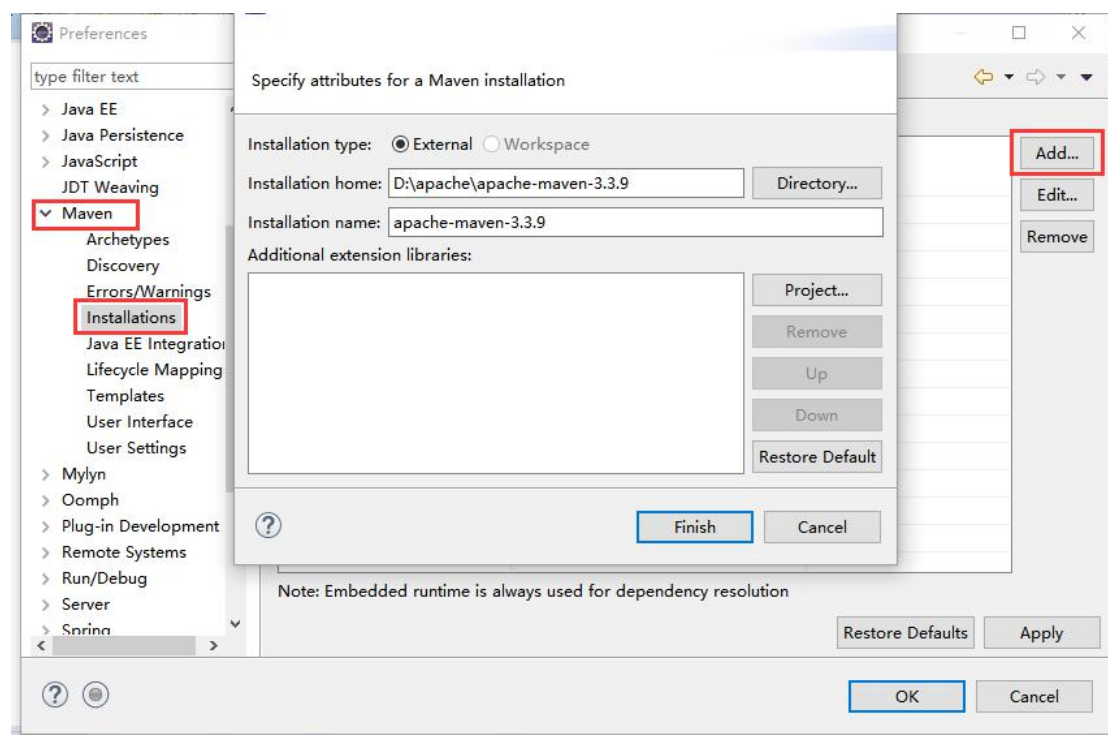
在项目的根目录下打开 cmd；执行命令：mvn eclipse:eclipse；将会出现.classpath 和.project 文件，之后可以将此项目作为一个普通 eclipse 项目导入 eclipse 中。

在项目的根目录下打开 cmd；执行命令：mvn eclipse:clean；则会将.classpath 和.project 文件删除。

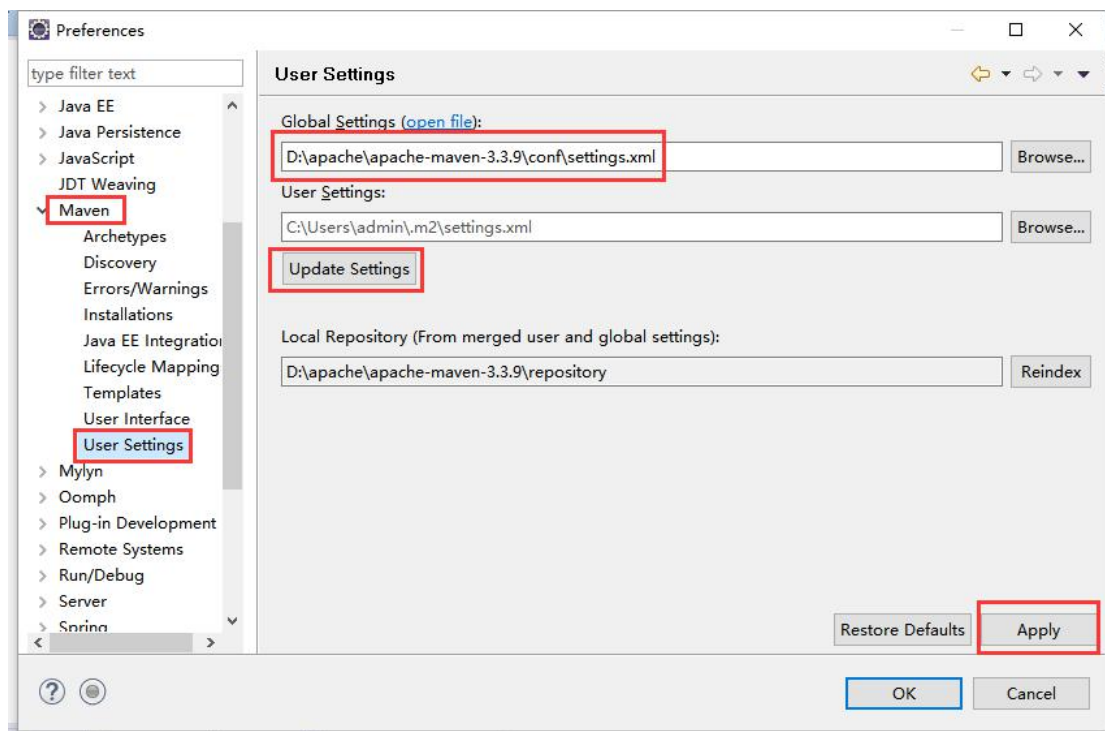
5.2. 配置 Eclipse Maven 环境

在 Eclipse 中使用 Maven 进行构建需要 m2eclipse 插件；m2eclipse 插件（<http://eclipse.org/m2e/>）为 Eclipse 提供了 Maven 的集成。新版的 Eclipse 默认安装了 m2eclipse 插件，如果是旧版的则需要先安装 m2eclipse 插件。

5.2.1. 配置 Maven 地址



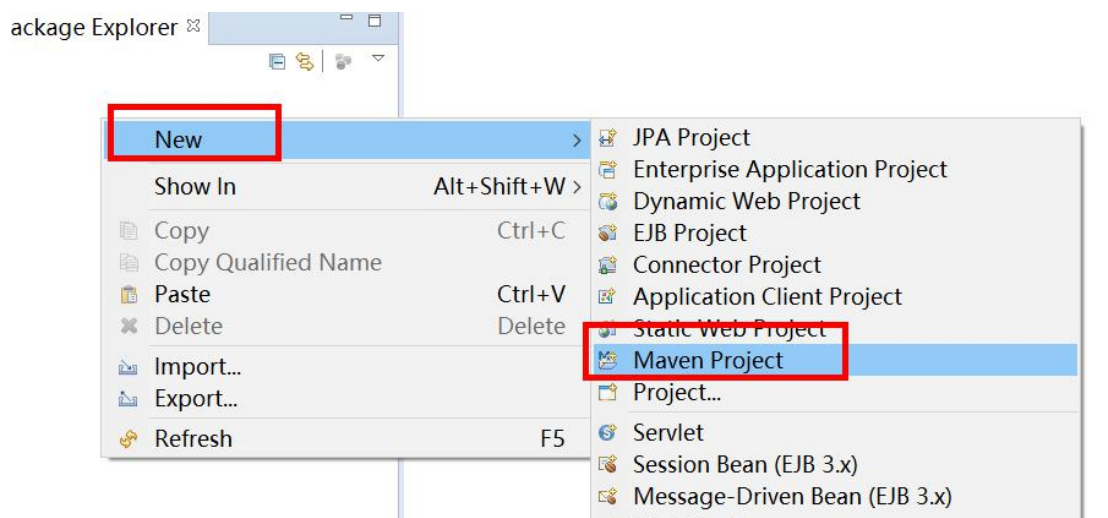
5.2.2. 设置 setting.xml 地址

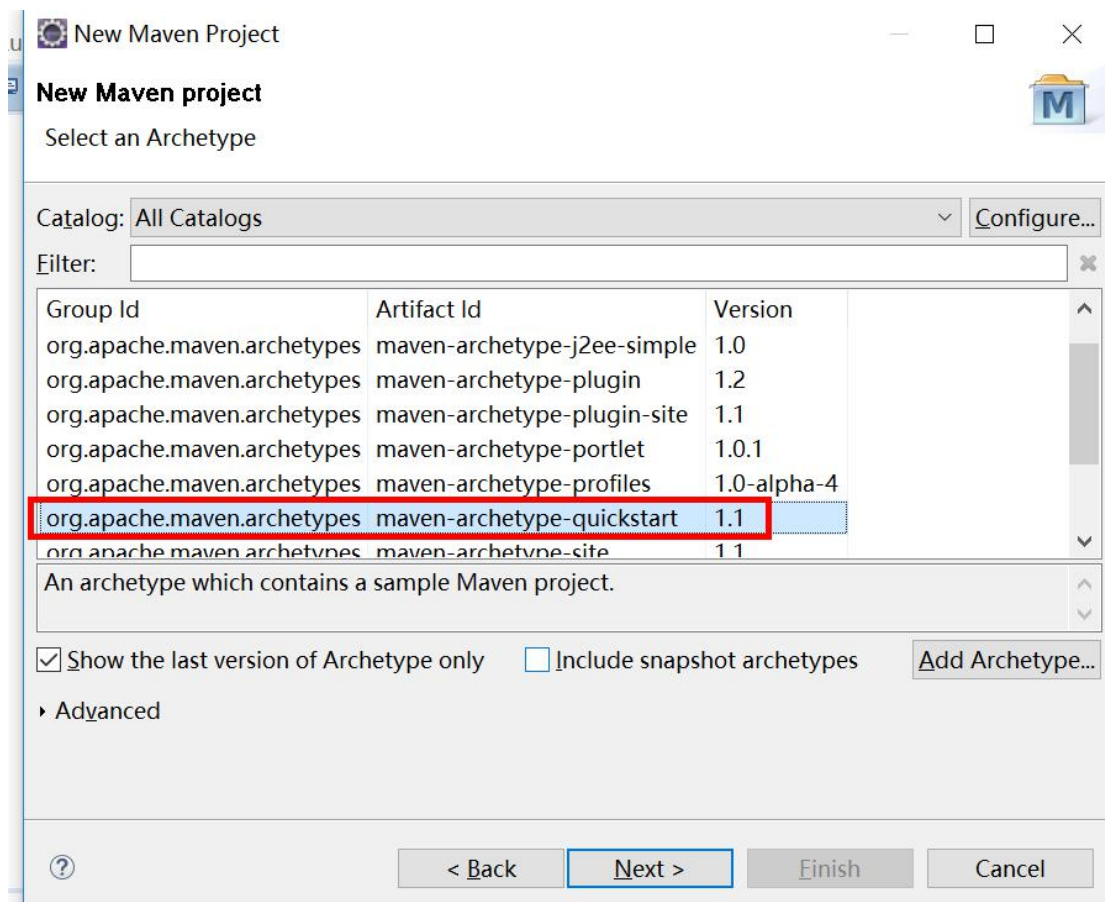
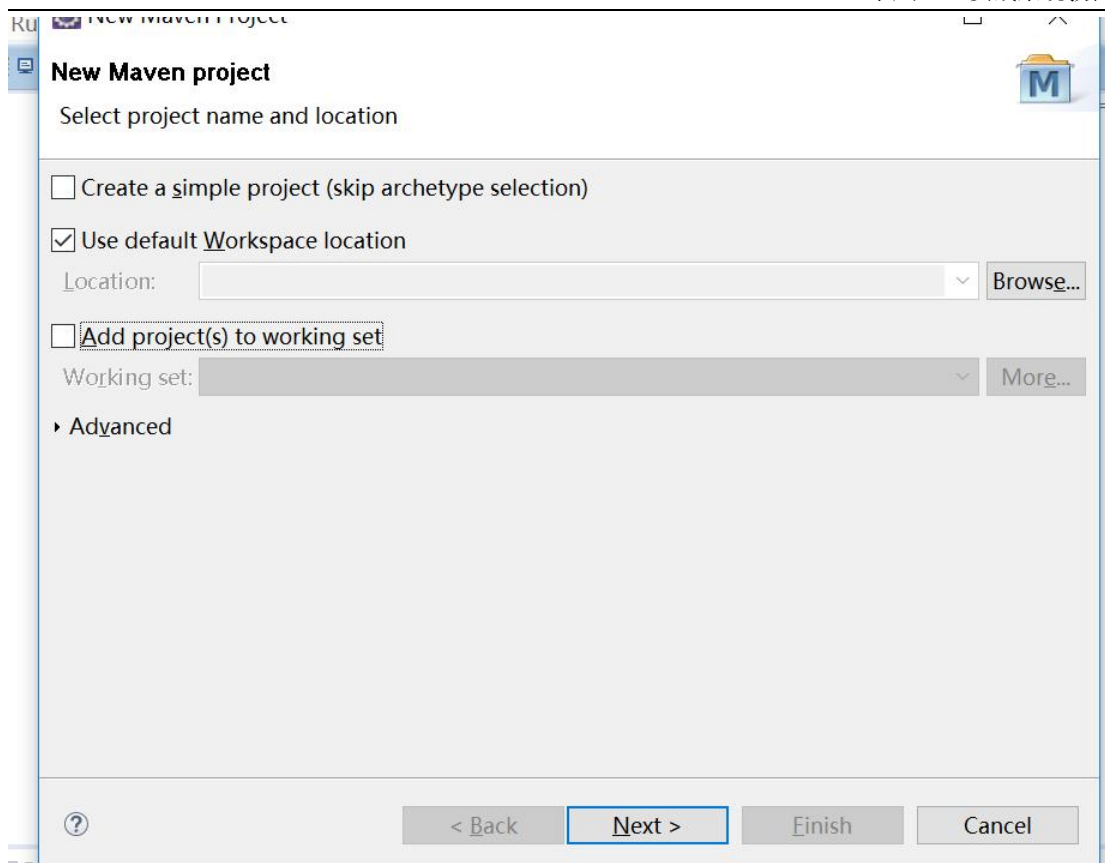


指定全局的 setting.xml 配置文件的路径。

5.3. 创建 Maven 项目

5.3.1. 创建 Maven 自带 Java 项目

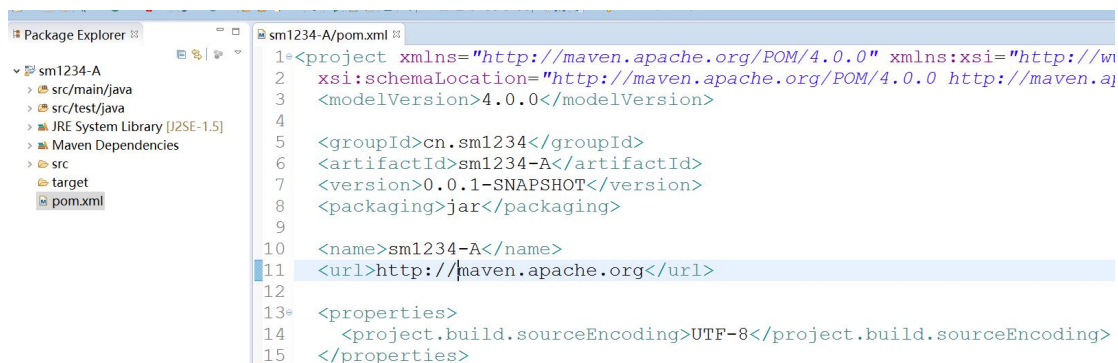
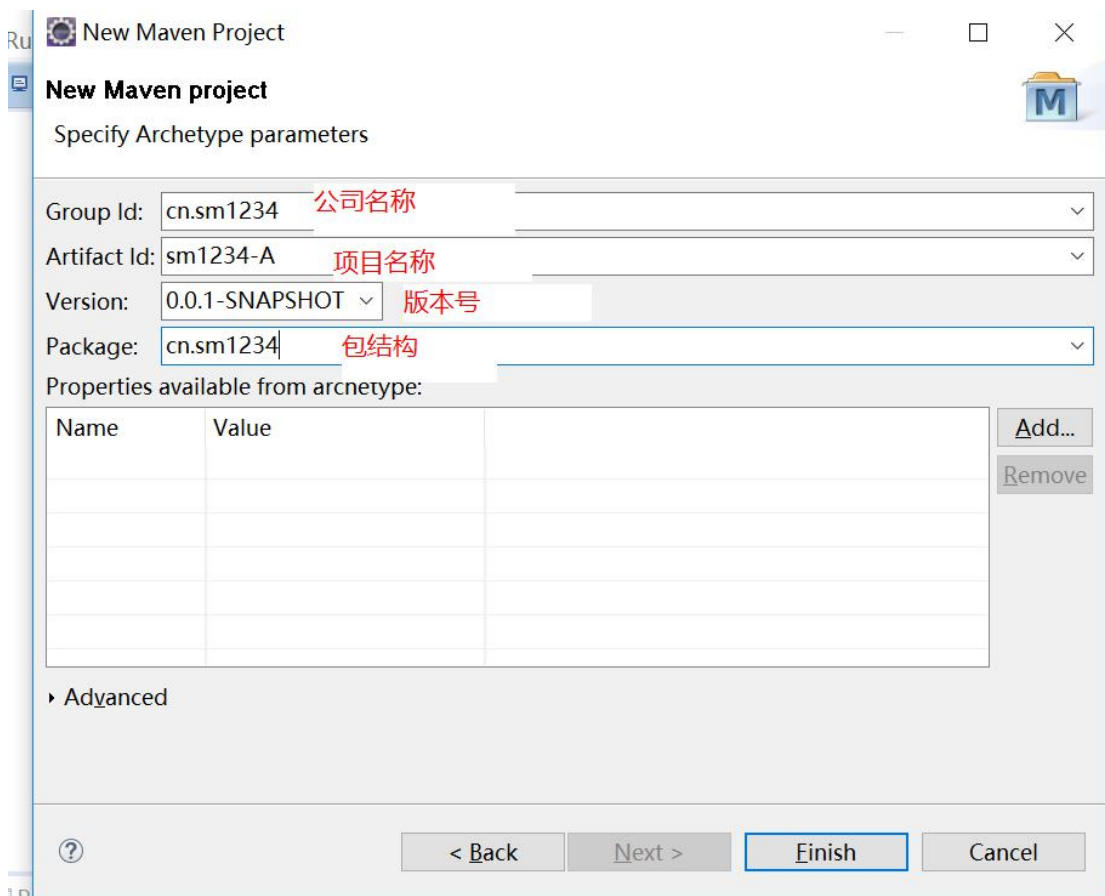




如果上图中的列表没有任何数据，那么返回上一步再执行 next 到此界面，让其

通过网络加载这些项目模型。

点击 Next 之后首次需要到中央仓库下载较多插件会慢一些；请耐心等待。



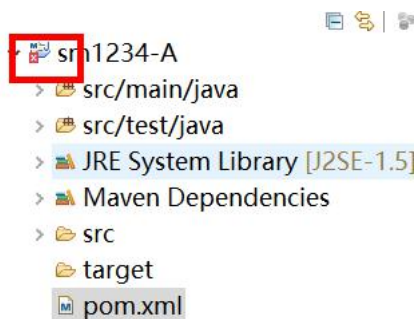
创建 Maven Java 项目完成。创建完项目后默认的编译版本设置为 maven 编译版本 1.7。在 pom.xml 文件中添加如下配置：

```
<build>
  <plugins>
    <!-- java编译插件 -->
    <plugin>
```

```
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.2</version>
<configuration>
  <source>1.7</source>
  <target>1.7</target>
  <encoding>UTF-8</encoding>
</configuration>
</plugin>
</plugins>
</build>
```

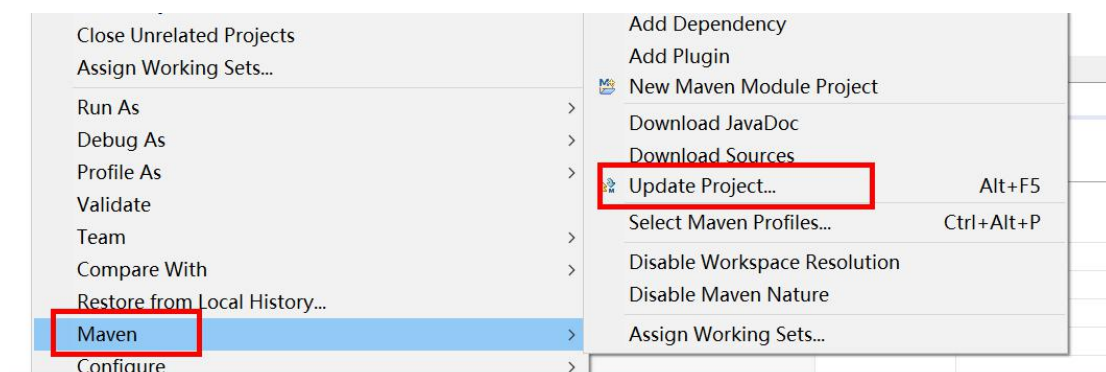
注意：

这时项目会出现红色叉叉：



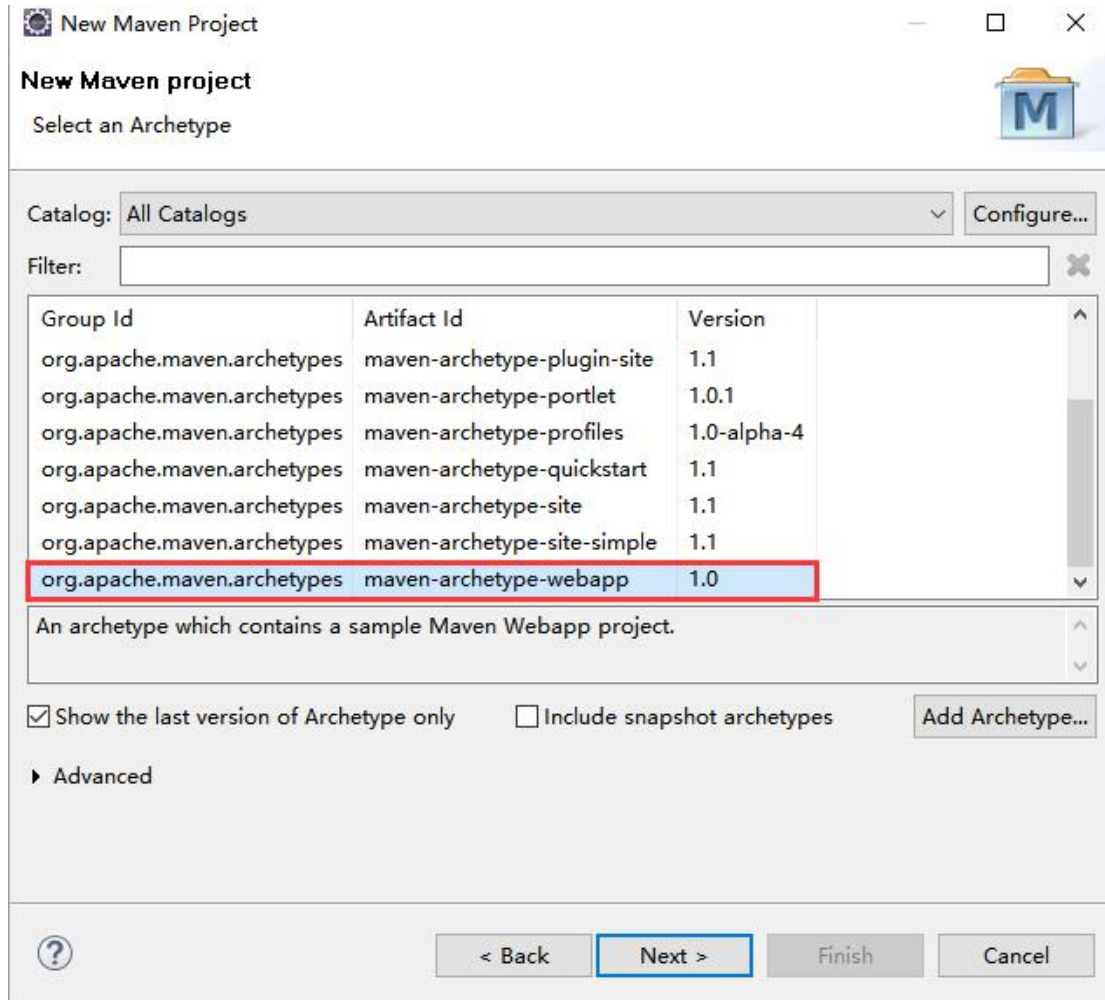
这是以后开发的时候经常出现的问题，是因为我们修改了 `pom.xml` 文件（可能添加了依赖或者插件），但是 `eclipse` 工具没来得及更新。这时需要手动更新项目：

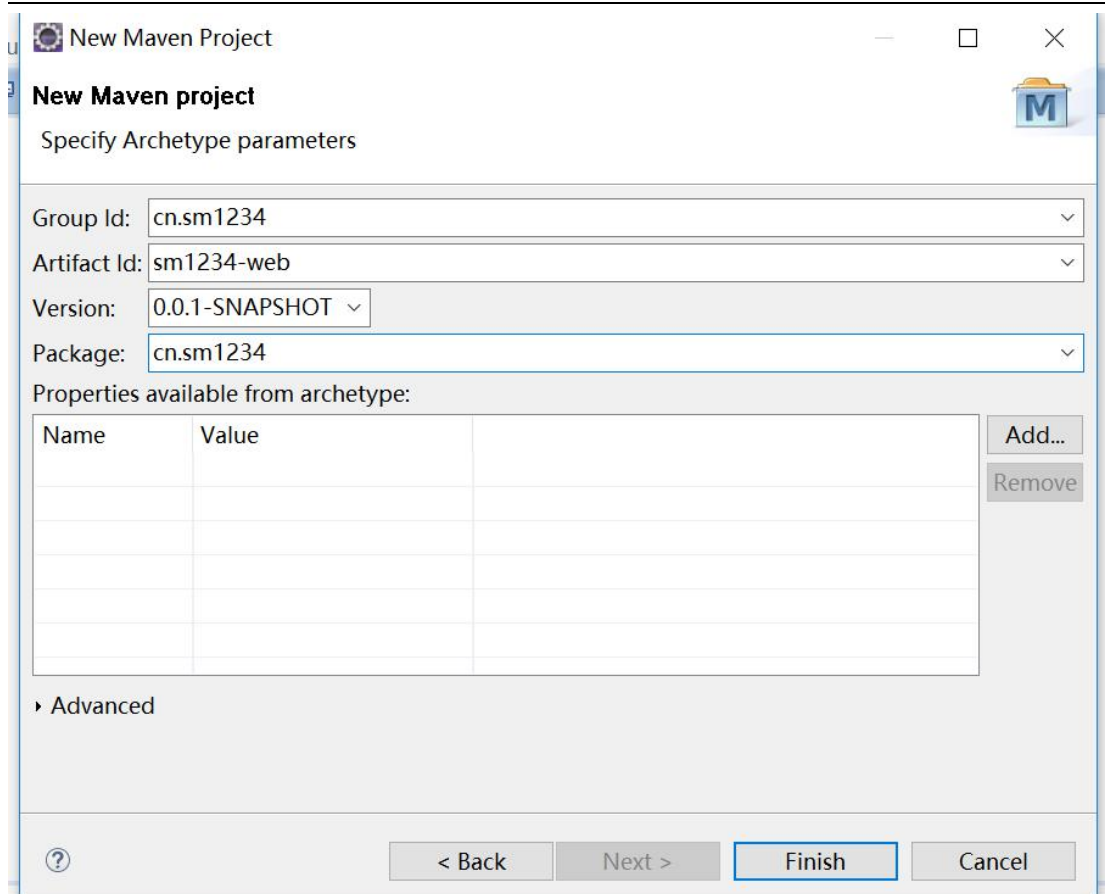
右键项目 -> Maven -> Update Project...



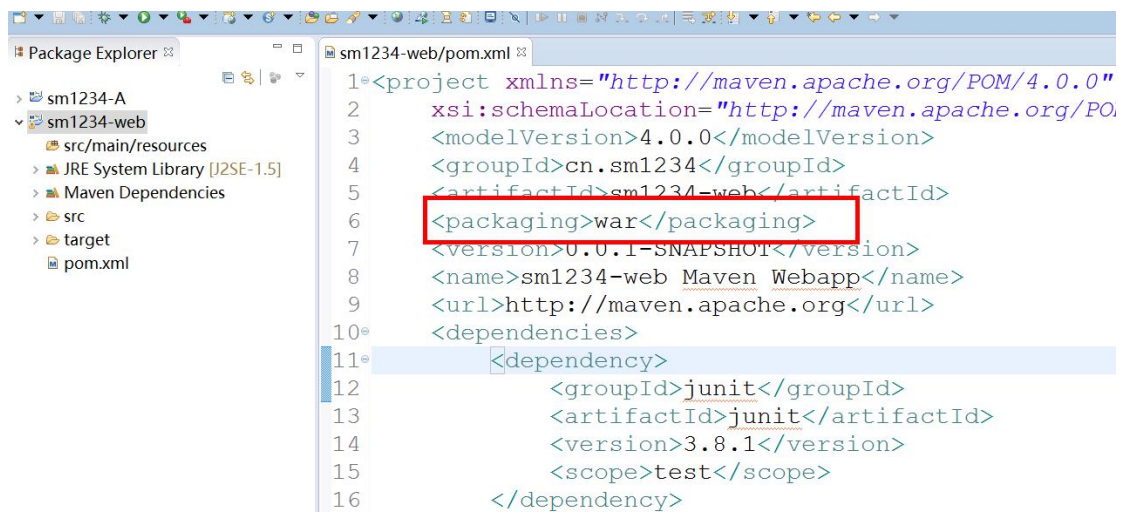
5.3.2. 创建 Maven 自带 Web 项目

创建 Web 项目和 Java 项目步骤类似，只是选择项目模型时有区别；如下图：





最终生成情况:



在创建完后分别设置编译版本（见 Java 项目配置），及添加 servlet/jsp 的依赖；
打开 pom.xml 在 dependencies 标签里面添加依赖如下：

```
<!-- JSP相关 -->
<dependency>
    <groupId>jstl</groupId>
    <artifactId>jstl</artifactId>
```

```
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>
```

5.3.3. 快速自定义创建

在 Eclipse 中创建项目时，可以自主根据需要创建 pom/jar/war 三种类似的任何一种而不需要选择具体的项目模型。（开发中用的多！）

New Maven Project

New Maven project

Select project name and location

☒ Create a simple project (skip archetype selection)

☒ Use default Workspace location

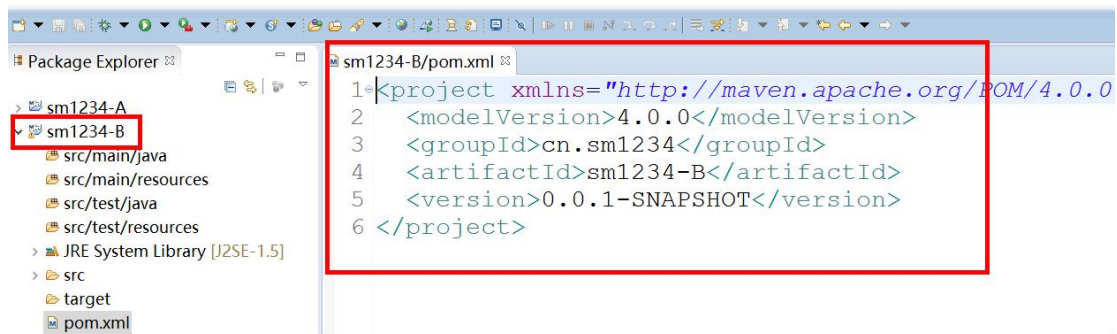
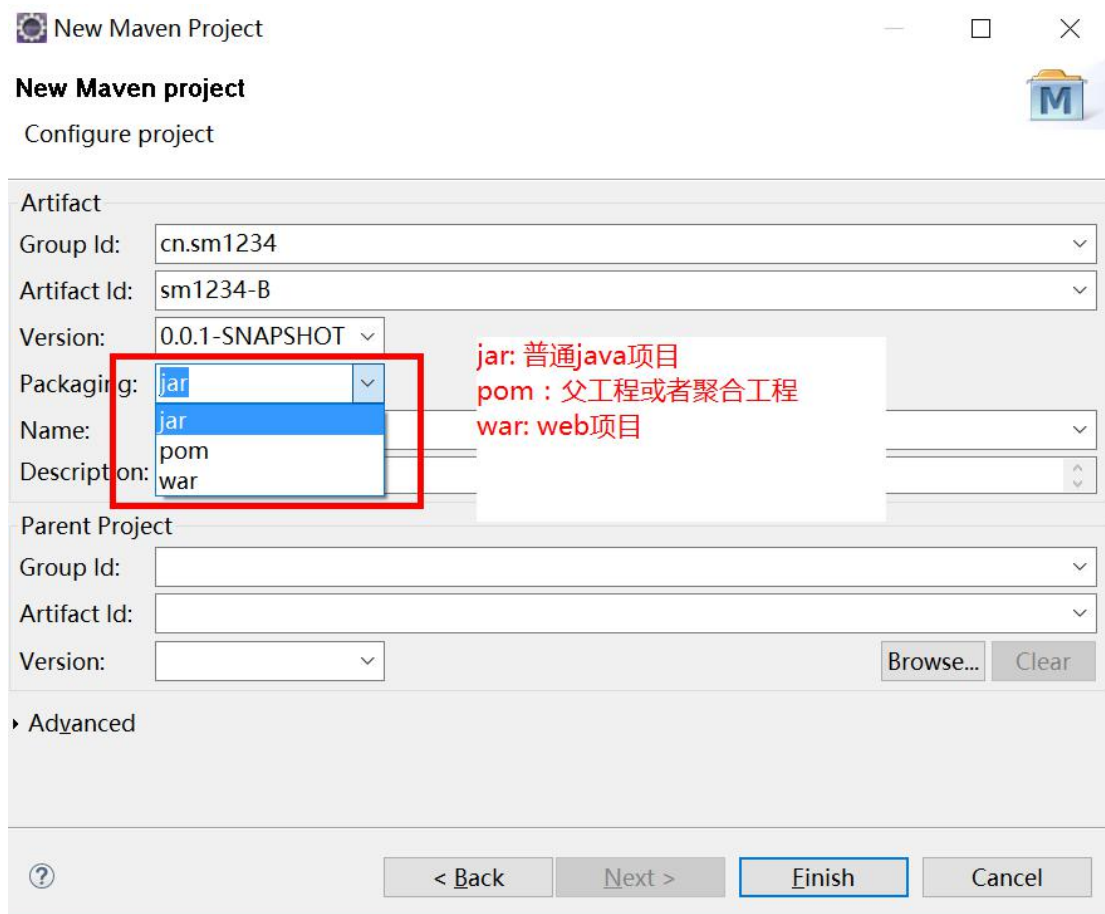
Location: E:\workspaces\sm1234_maven\sm1234-web Browse...

☐ Add project(s) to working set

Working set: More...

▸ Advanced

? < Back Next > Finish Cancel



创建较快速，完成后也需设置 maven 编译版本。

5.4. 使用 Eclipse 构建 Maven 项目

1) 创建 cn.sm1234.HelloMaven

```
package cn.sm1234;
```



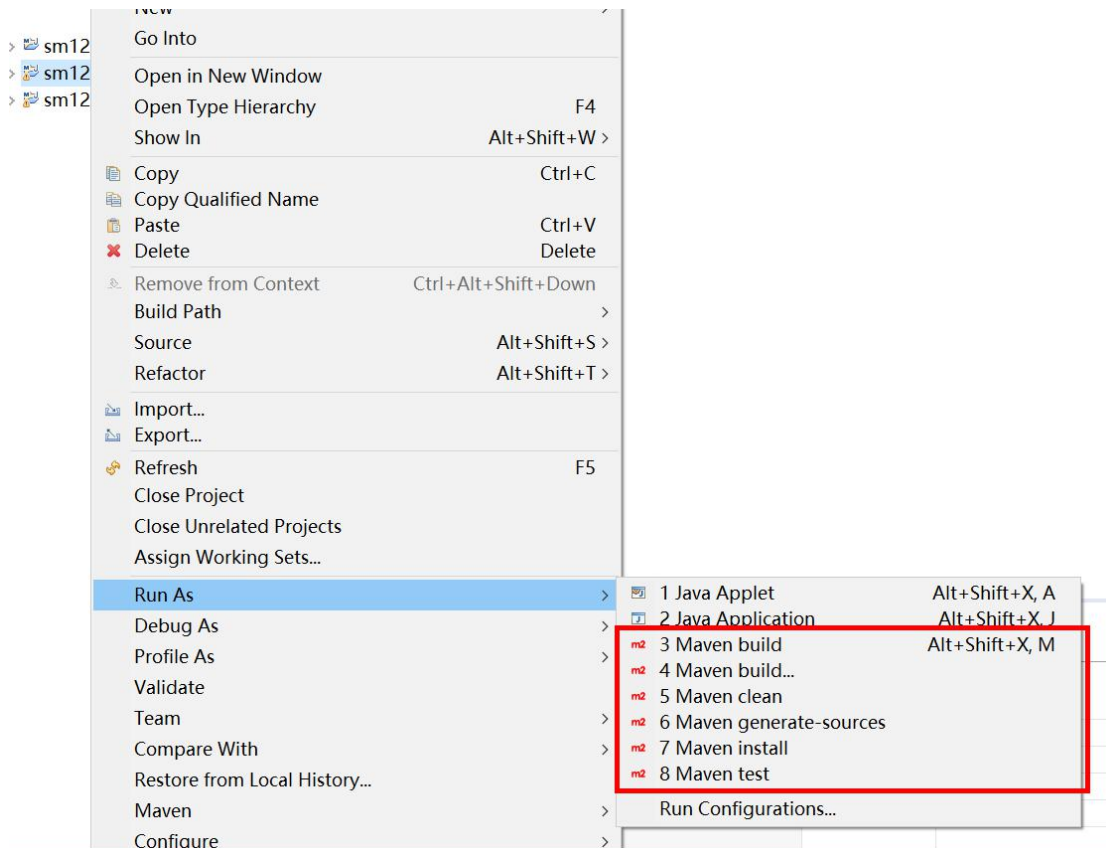
```
public class HelloMaven {  
  
    public String sayHello(String name){  
  
        return "Hello,"+name;  
  
    }  
  
}
```

2) 创建 cn.sm1234.HelloMavenTest

```
package cn.sm1234;  
  
import org.junit.Test;  
  
public class HelloMavenTest {  
  
    @Test  
  
    public void testSayHello(){  
  
        HelloMaven hello = new HelloMaven();  
        System.out.println(hello.sayHello("maven"));  
  
    }  
  
}
```

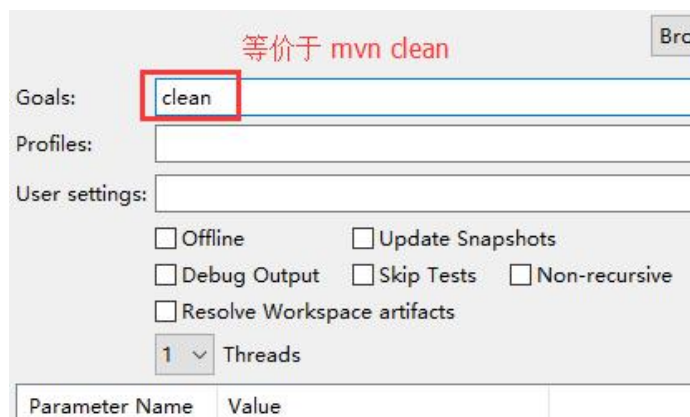
3) 应用 Eclipse 提供命令进行构建

可以选中项目或者 pom.xml，也或者在 pom.xml 中右击，选择 Run As 或者 Debug As，出现如下：



在上述的构建中：

Maven build... 点击后可以设置具体的 Maven 构建命令，这些命令不需要再加 mvn 前缀，如：



Maven generate-sources 表示替在编码过程中引用到的类关联源码。

6. Maven 核心概念

6.1. 仓库

Maven 在某个统一的位置存储所有项目的共享的构件，这个统一的位置，就称之为仓库。（仓库就是存放依赖和插件的地方）Maven 的仓库有两大类：1. 本地仓库 2. 远程仓库，在远程仓库中又分成了 3 种：中央仓库、私服、其它公共库。

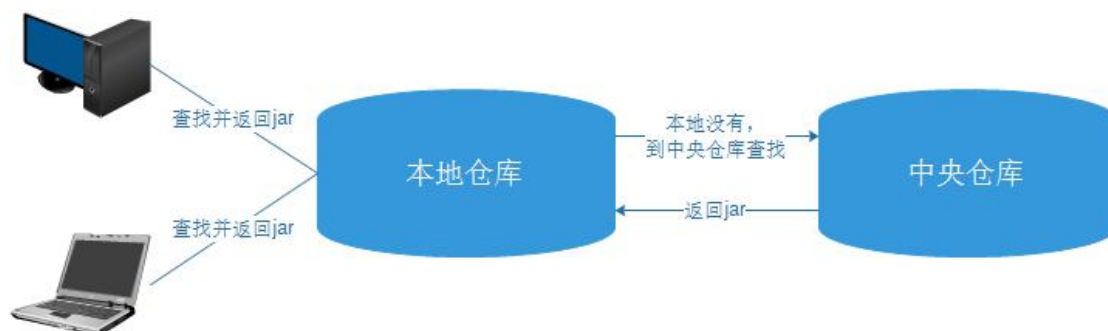
本地仓库：就是 Maven 在本机存储构件的地方。maven 的本地仓库，在安装 maven 后并不会创建，它是在第一次执行 maven 命令的时候才被创建。maven 本地仓库的默认位置：在用户的目录下都只有一个 .m2/repository/ 的仓库目录；可以修改。

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、SCM、信息、许可证信息等。开源的 Java 项目依赖的构件都可以在这里下载到。

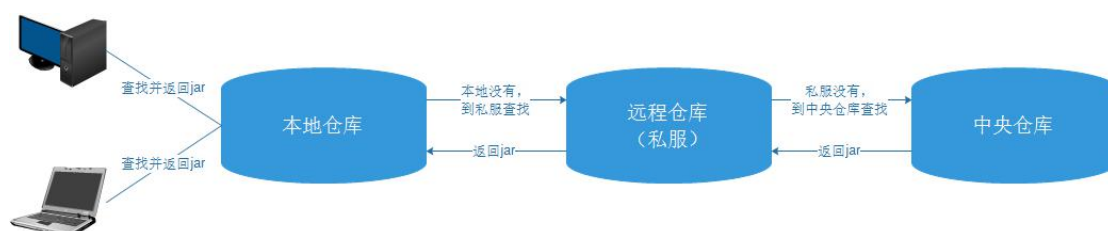
<http://repo1.maven.org/maven2/>

私服：是一种特殊的远程仓库，它是架设在局域网内的仓库

1、没有使用私服的仓库构件下载



2、使用私服的仓库构件下载



6.2. 坐标

- 什么是坐标？

在平面几何中坐标 (x,y) 可以标识平面中唯一的一点

- Maven 坐标主要组成

groupId: 定义当前 Maven 项目隶属项目、组织

artifactId: 定义实际项目中的一个模块

version: 定义当前项目的当前版本

packaging: 定义该项目的打包方式 (pom/jar/war, 默认为 jar)

groupId、artifactId、version 简称为 GAV。

- Maven 为什么使用坐标？

Maven 世界拥有大量构件，需要找一个用来唯一标识一个构建的统一规范

拥有了统一规范，就可以把查找工作交给机器

- 如何获取坐标

<http://mvnrepository.com/> 网站上可以搜索具体的组织或项目关键字，之后复制对应的坐标到 pom.xml 中。如：

Artifact	Download (JAR) (982 KB)
POM File	View
Date	(Sep 04, 2014)
HomePage	https://github.com/spring-projects/spring-
Organization	Spring IO
Issue Tracker	https://jira.springsource.org/browse/SPR

Maven

Gradle

SBT

Ivy

Grape

Leiningen

Build

```
<!-- http://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.1.0.RELEASE</version>
</dependency>
```

☒ Include comment with link to declaration

6.3. 依赖管理（重点）

在 pom.xml 中的节点 dependency 为一个依赖；groupId、artifactId、version 是依赖的基本坐标，缺一不可。

type: 依赖的类型，默认为 jar，说明需要使用 jar 包。

注意：如果 “<type>pom.lastUpdated</type>” 有此内容，要进行删除。

6.3.1. 依赖范围问题（scope 配置）

依赖范围 scope 用来控制依赖和编译，测试，运行的 classpath 的关系。具体的依赖范围有如下 6 种：

- 1) **compile:** 编译域，这个是 Maven 中 scope 的默认值，我们平时不写 scope 配置时默认就是使用这个值。compile 表示被依赖项目需要同当前项目编译时一起进行编译，项目测试期以及本项目运行时期同样生效，打包的时候需要包含进去。
- 2) **test:** 测试域，表示被依赖的项目仅在项目进行测试的时候生效，一般将日志等依赖包（如：Junit）配置为 test，项目运行时不会生效。
- 3) **provided:** provided 意味着打包的时候可以不用打包进去，别的容器会提供，如 Servlet-API，Tomcat 这些容器会提供，所以打包，运行时无需提供。
- 4) **runtime:** 运行域，表示被依赖项目不会参与项目的编译，但项目的测试期和运行时期会参与。与 compile 相比，跳过了编译这个环节。（如：JDBC 驱动）
- 5) **system:** 系统范围,自定义构件，指定 systemPath；跟 provided 相似，但是在系统中要以外部 JAR 包的形式提供，maven 不会在 repository 查找它。
- 6) **import:** 只使用在<dependencyManagement>中，表示从其它的 pom 中导入 dependency 的配置。

依赖范围	说明	编译有效?	运行有效?	测试有效	是否打包	实际应用
------	----	-------	-------	------	------	------

Compile	编译范围	Y	Y	Y	Y	Ssh、poi
Test	测试范围	Y	N	Y	N	Junit
Provide	容器范围	Y	N	Y	N	Servlet.jar
runtime	运行范围	N	Y	Y	Y	驱动

6.3.2. 传递依赖问题

举例：

当前我们的 maven 项目导入 spring-context 的时候，会同时 spring-aop,spring-beans,spring-core 等依赖导入：

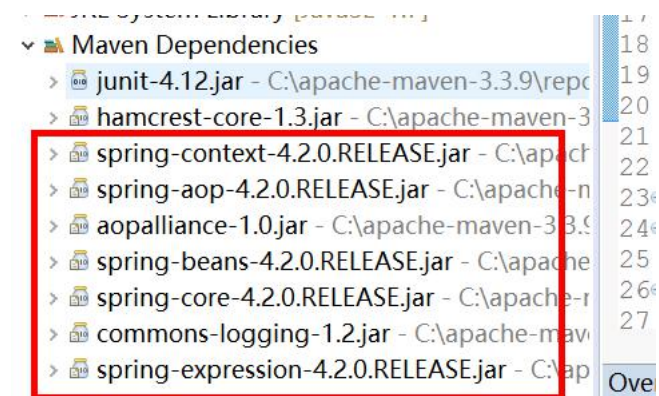
```
<dependency>

    <groupId>org.springframework</groupId>

    <artifactId>spring-context</artifactId>

    <version>4.2.0.RELEASE</version>

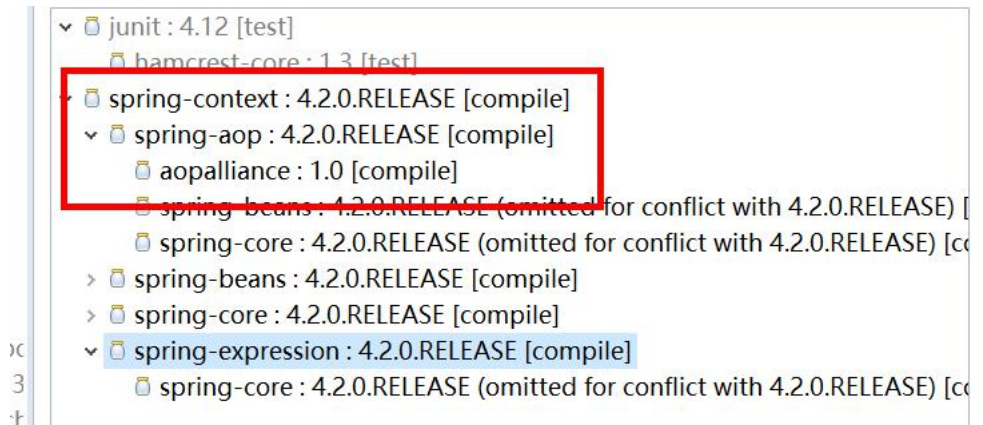
</dependency>
```



打开依赖树：



发现依赖关系是这样的：



说明：**spring-context** 和 **spring-aop** 是第一个直接依赖，而 **spring-aop** 和 **aopalliance** 是第二直接依赖。

为什么我们只导入了 **spring-context** 就会导入其他那么多 jar 呢？

这就是**依赖传递**的作用啦！

而且需要明白，依赖传递会影响依赖访问，大致意思是：

假设 A 依赖于 B，B 依赖于 C，我们说 A 对于 B 是**第一直接依赖**，B 对 C 是**第二直接依赖**，A 对于 C 是**传递性依赖**。第一直接依赖的范围和第二直接依赖的范围决定了传递性依赖的范围。

对于传递性依赖，依赖的范围如下表：

第一 \ 第二	compile	test	provided	runtime
compile	compile	-	-	runtime
test	test	-	-	test
provided	provided	-	provided	provided
runtime	runtime	-	-	runtime

6.3.2.1. 可选依赖（optional 配置）

在依赖节点 `dependency` 中的 `<optional>` 可以控制当前的依赖是否向下传递；默认值为 `false`，表示向下传递。

【示例】A 项目依赖于 `log4j`，然后 B 项目依赖于 A 项目；那么如果在 A 中对 `log4j` 依赖的 `optional` 配置成 `false` 时，B 项目中自动传递依赖于 `log4j`。否则反之。

1) 项目 A 配置 `slf4j` 的依赖并设置 `optional` 为 `true`

```
<dependency>

    <groupId>org.slf4j</groupId>

    <artifactId>slf4j-log4j12</artifactId>

    <version>1.6.4</version>

    <!-- 配置为 true 时不向下传递此依赖，默认为 false -->

    <optional>true</optional>

</dependency>
```

2) 配置项目 B 依赖于项目 A，检查项目 B 的依赖包

```
<!-- 依赖于 itcast-A -->

<dependency>

    <groupId>cn.sm1234</groupId>

    <artifactId>sm1234-A</artifactId>

    <version>0.0.1-SNAPSHOT</version>

</dependency>
```

6.3.2.2. 排除依赖（exclusion 配置）

在 `pom` 中的依赖节点中，如果引入的依赖包含了很多其它的传递依赖，而且项目需要的这些依赖的版本和传递依赖的不相符；那么可以在依赖节点中设置排除依赖节点：`<exclusions>` 然后再添加 `<exclusion>`，其里面的内容包括：

①所包含坐标

②排除依赖包中所包含的依赖关系

【注意】不需要添加版本，直接类别排除

```
<dependency>

    <groupId>org.apache.struts</groupId>

    <artifactId>struts2-spring-plugin</artifactId>

    <version>2.3.24.1</version>

    <exclusions>

        <!-- 排除 spring-core 的传递依赖 -->

        <exclusion>

            <groupId>org.springframework</groupId>

            <artifactId>spring-core</artifactId>

        </exclusion>

    </exclusions>

</dependency>

<dependency>

    <groupId>org.springframework</groupId>

    <artifactId>spring-beans</artifactId>

    <version>4.1.0.RELEASE</version>

</dependency>
```

6.3.3. 依赖冲突问题

6.3.3.1. 不同依赖路径的情况

- 如果依赖的路径不相同的时候，以最短的路径为准。

A->B->C->D->X(1.6)
A->D-X(2.0)

最终 A 依赖的 X 的版本为 2.0

【比如】：项目 A 中，依赖了 slf4j1.6.4 版本的包，通过 slf4j1.6.4 间接依赖 log4j1.2.16 版本；如果项目 A 中直接配置了 log4j 1.2.17 版本，那么最终的版本为 1.2.17。

```
<dependency>

    <groupId>org.slf4j</groupId>

    <artifactId>slf4j-log4j12</artifactId>

    <version>1.6.4</version>

    <!-- 配置为 true 时不向下传递此依赖，默认为 false -->

    <optional>true</optional>

</dependency>


<dependency>

    <groupId>log4j</groupId>

    <artifactId>log4j</artifactId>

    <version>1.2.17</version>

</dependency>
```

6.3.3.2. 相同依赖路径的情况

- 如果直接依赖中包含有同一个坐标不同版本的资源依赖，以配置顺序下方的版本为准

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

<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
```

以这个版本为准

- 如果间接依赖中包含有同一个坐标不同版本的资源依赖，以配置顺序上方的版本为准

如下对应 `spring-core` 的间接依赖的版本号，以 `struts2-spring-plugin` 为准

```
<dependency>

  <groupId>org.apache.struts</groupId>

  <artifactId>struts2-spring-plugin</artifactId>

  <version>2.3.24.1</version>

</dependency>

<dependency>

  <groupId>org.springframework</groupId>

  <artifactId>spring-beans</artifactId>

  <version>4.1.0.RELEASE</version>

</dependency>
```

6.4. 生命周期

Maven 生命周期就是为了对所有的构建过程进行抽象和统一；包括项目清理，初始化，编译，打包，测试，部署等几乎所有构建步骤。

Maven 有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。

再次强调它们是相互独立的，可以仅仅调用 `clean` 来清理工作目录，仅仅调用 `site` 来生成站点。不过也可以直接运行 `mvn clean install site` 运行所有这三套生命周期。

6.4.1. clean 生命周期

`clean` 生命周期每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。比如，运行 `mvn clean`，这个的 `clean` 是 `clean` 生命周期的一个阶段。有 `clean` 生命周期，也有 `clean` 阶段。`clean` 生命周期一共包含了三个阶段：

- `pre-clean` 执行一些需要在 `clean` 之前完成的工作
- `clean` 移除所有上一次构建生成的文件
- `post-clean` 执行一些需要在 `clean` 之后立刻完成的工作

`mvn clean` 中的 `clean` 就是上面的 `clean`，在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，`mvn clean` 等同于 `mvn pre-clean clean`，如果我们运行 `mvn post-clean`，那么 `pre-clean`，`clean` 都会被运行。这是 Maven 很重要的一个规则，可以大大简化命令行的输入。

6.4.2. default 生命周期

default 生命周期 default 生命周期是 Maven 生命周期中最重要的一個，绝大部分工作都发生在这个生命周期中。比较重要和常用的阶段如下：

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 运行任何检查，验证包是否有效且达到质量标准。

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，以让其它开发人员与项目共享。

运行任何一个阶段的时候，它前面的所有阶段都会被运行，这也就是为什么运行 `mvn install` 的时候，代码会被编译，测试，打包。此外，Maven 的插件机制是完全依赖 Maven 的生命周期的。

6.4.3. site 生命周期

site 生命周期包含如下 4 个阶段：

- pre-site 执行一些需要在生成站点文档之前完成的工作
- site 生成项目的站点文档
- post-site 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
- site-deploy 将生成的站点文档部署到特定的服务器上

这里经常用到的是 site 阶段和 site-deploy 阶段，用以生成和发布 Maven 站点，这是 Maven 相当强大的功能，Manager 比较喜欢，文档及统计数据自动生成，很好看。