

# 1 课程计划

- 1、maven 的介绍
- 2、maven 的安裝配置
- 3、创建 maven 工程
- 4、M2Eclipse
- 5、Maven 的核心概念
  - a) 坐标
  - b) 依赖管理
  - c) 生命周期
  - d) 插件
  - e) 继承
  - f) 聚合
- 6、maven 的仓库管理

## 2 maven 的介绍

### 2.1 开发中遇到的问题

- 1、都是同样的代码，为什么在我的机器上可以编译执行，而在他的机器上就不行？
- 2、为什么在我的机器上可以正常打包，而配置管理员却打不出来？
- 3、项目组加入了新的人员，我要给他说明编译环境如何设置，但是让我挠头的是，有些细节我也记不清楚了。
- 4、我的项目依赖一些 jar 包，我应该把他们放哪里？放源码库里？
- 5、这是我开发的第二个项目，还是需要上面的那些 jar 包，再把它们复制到我当前项目的 svn 库里吧
- 6、现在是第三次，再复制一次吧 ----- 这样真的好吗？
- 7、我写了一个数据库相关的通用类，并且推荐给了其他项目组，现在已经有五个项目组在使用它了，今天我发现了一个 bug，并修正了它，我会把 jar 包通过邮件发给其他项目组  
-----这不是一个好的分发机制，太多的环节可能导致出现 bug
- 7、项目进入测试阶段，每天都要向测试服务器部署一版。每次都手动部署，太麻烦了。

## 2.2 什么是 maven

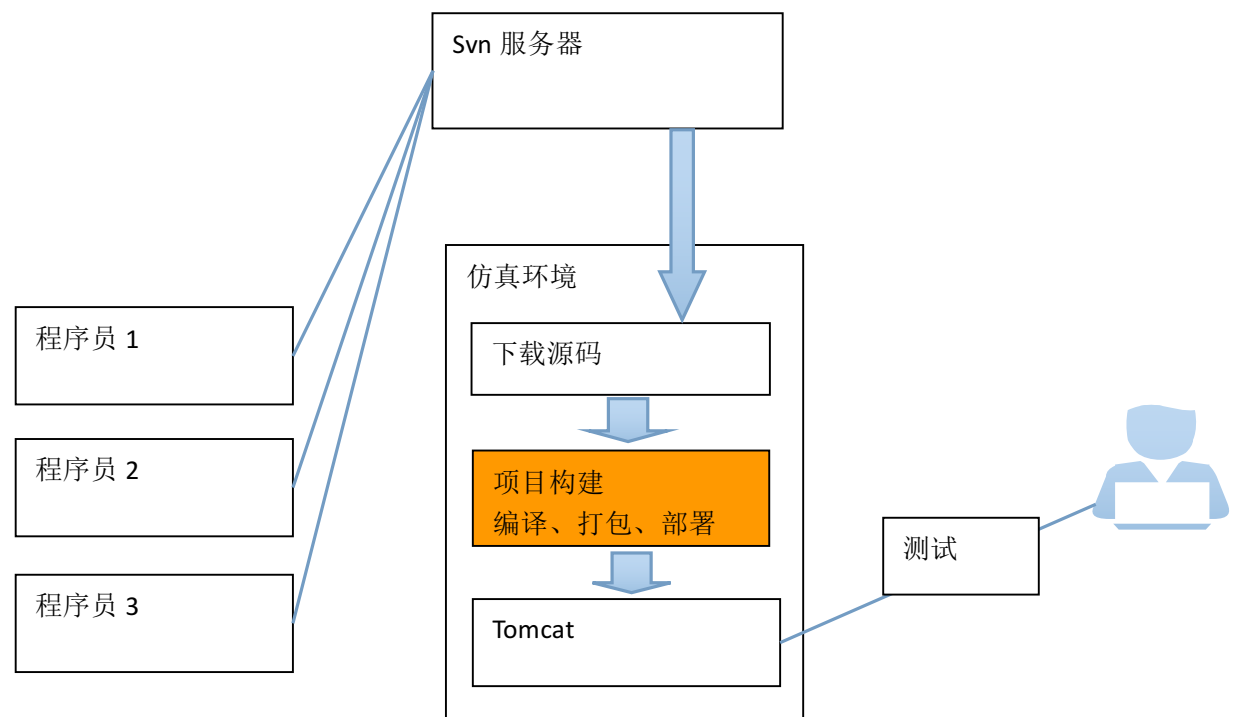
Maven 是基于 POM（工程对象模型），通过一小段描述来对项目的代码、报告、文件进管理的工具。

Maven 是一个跨平台的项目管理工具，它是使用 java 开发的，它要依赖于 jdk1.6 及以上

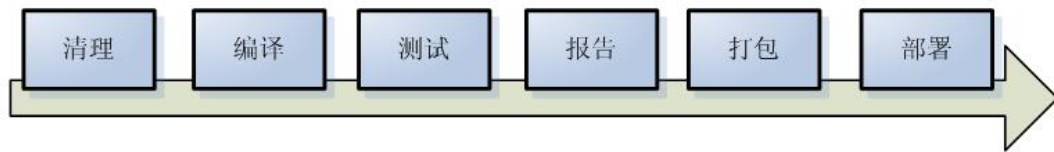
Maven 主要有两大功能：管理依赖、项目构建。

依赖指的就是 jar 包。

## 2.3 什么是构建



构建过程：



## 2.4 项目构建的方式

### 1、Eclipse

使用 eclipse 进行项目构建，相对来说，步骤比较零散，不好操作

### 2、Ant

它是一个专门的项目构建工具，它可以通过一些配置来完成项目构建，这些配置要明确的告诉 ant，源码包在哪？目标 class 文件应该存放在哪？资源文件应该在哪

### 3、Maven

它是一个项目管理工具，他也是一个项目构建工具，通过使用 maven，可以对项目进行快速简单的构建，它不需要告诉 maven 很多信息，但是需要安装 maven 去的规范去进行代码的开发。也就是说 maven 是有约束的。

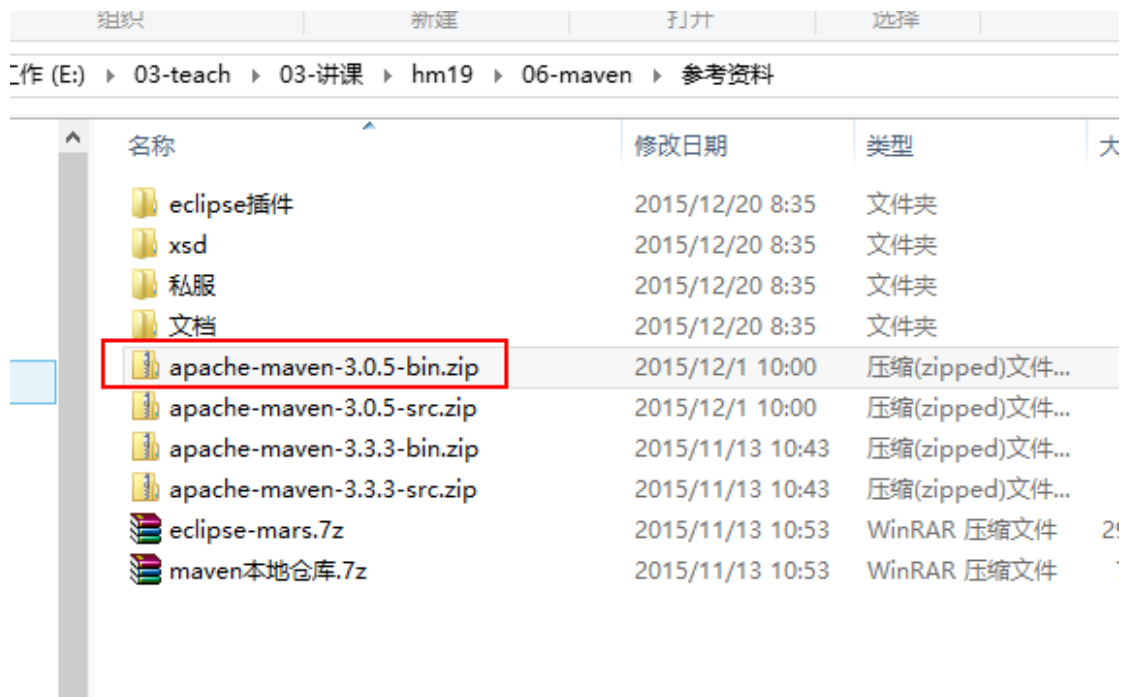
## 3 Maven 的安装配置

### 3.1 下载 maven

官方网站: <http://maven.apache.org>

本课程使用的 maven 的版本为 3.0.5

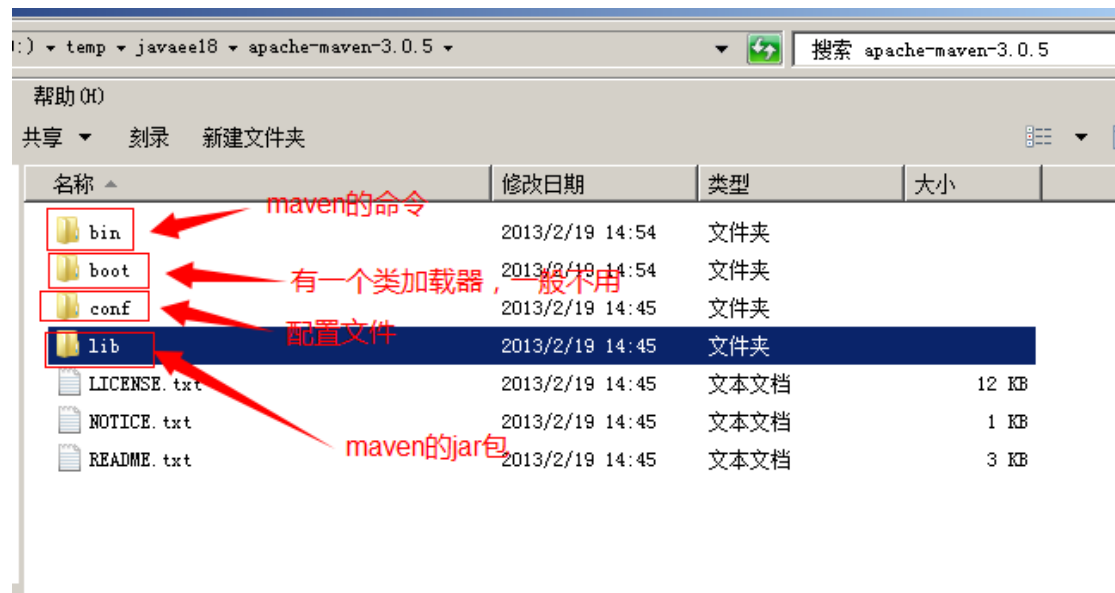
Maven 是使用 java 开发，需要安装 jdk1.6 以上，推荐使用 1.7



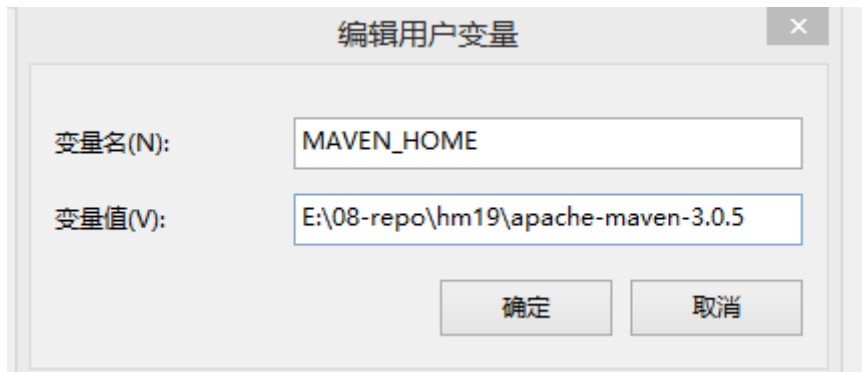
## 3.2 安装 maven

第一步：安装 jdk1.6 及以上

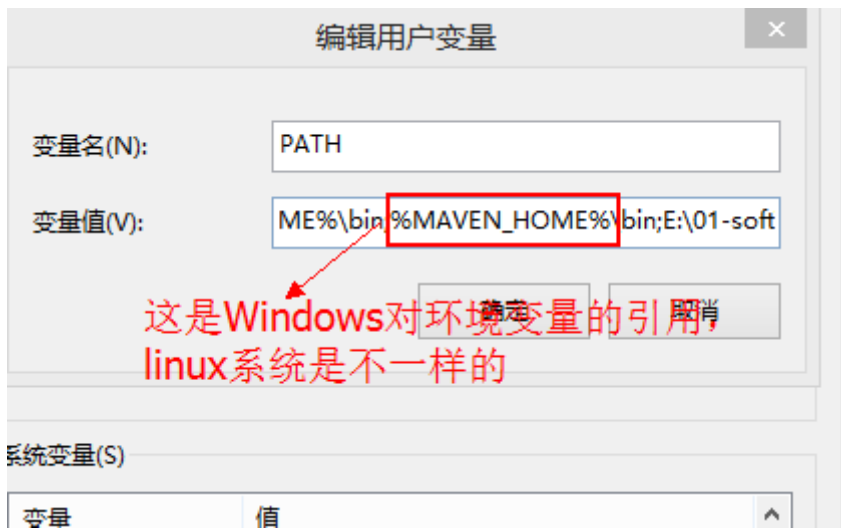
第二步：将 maven 下载的压缩包进行解压缩



第三步：配置 maven 的环境变量 MAVEN\_HOME



第四步：配置 maven 的环境变量 PATH



第五步：测试 maven 是否安装成功，在系统命令行中执行命令：mvn -v

```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

C:\Users\think>mvn -u
Apache Maven 3.0.5 (r01de14724cdef164cd33c7c8c2fe155faf9602da; 2013-02-19 21:51:
28+0800)
Maven home: E:\08-repo\hm19\apache-maven-3.0.5\bin\..
Java version: 1.7.0_72, vendor: Oracle Corporation
Java home: E:\01-soft\01-devTools\Java\jdk1.7.0_72\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 8.1", version: "6.3", arch: "x86", family: "windows"
C:\Users\think>
```

### 3.3 配置 maven

在 maven 中有两个配置文件：用户配置、全局配置（默认）

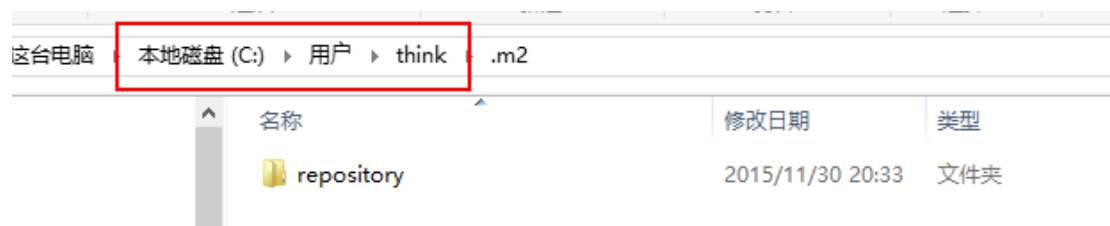
#### 3.3.1 全局配置

在 maven 安装目录的 conf 里面有一个 settings.xml 文件，这个文件就是 maven 的全局配置文件。

该文件中配置来 maven 本地仓库的地址

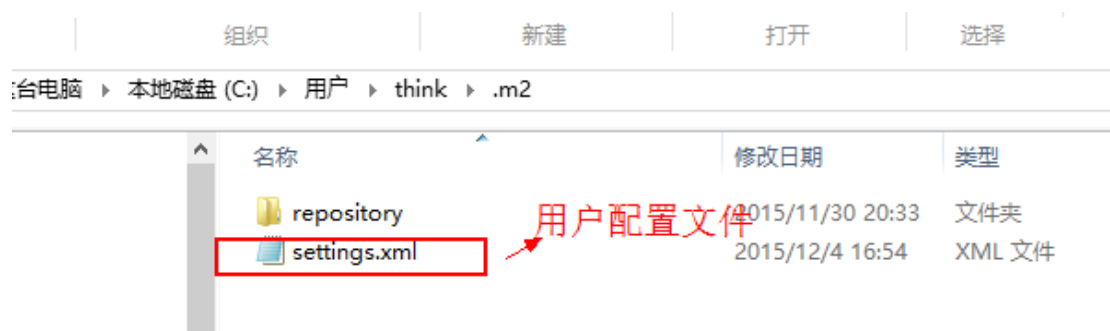
```
<!-- Default: ~/.m2/repository -->
<localRepository>path/to/local/repo</localRepository>
```

默认在系统的用户目录下的 `m2/repository` 中，该目录是本地仓库的目录。



### 3.3.2 用户配置

用户配置文件的地址：~/.m2/settings.xml，该文件默认是没有，需要将全局配置文件拷贝一份到该目录下。



重新指定本地仓库地址，如果不指定，则默认是~/.m2/repository 目录，如果用户配置文件不存在，则使用全局配置文件的配置。



## 4 创建 maven 工程

### 4.1 Maven 工程结构

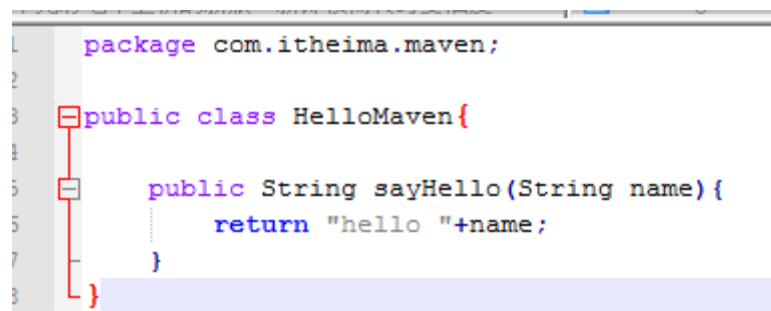
## Project

- |--src (源码包)
  - |--main (正常的源码包)
    - |--java (.java 文件的目录)
    - |--resources (资源文件的目录)
  - |--test (测试的源码包)
    - |--java
    - |--resources
- |--target (class 文件、报告等信息存储的地方)
- |--pom.xml (maven 工程的描述文件)

## 4.2 创建 HelloMaven 工程

### 4.2.1 第一步：安装 maven 的工程结构创建 helloMaven 工程

### 4.2.2 第二步：创建 HelloMaven.java



```
1 package com.itheima.maven;
2
3 public class HelloMaven{
4
5     public String sayHello(String name){
6         return "hello "+name;
7     }
8 }
```



### 4.2.3 第三步：创建 TestHelloMaven.java

```
package com.itheima.maven;

import org.junit.Test;
import static junit.framework.Assert.*;

public class TestHelloMaven{

    @Test
    public void testSayHello(){
        HelloMaven hm = new HelloMaven();

        String result = hm.sayHello("hm19");

        assertEquals("hello hm19",result);
    }
}
```

### 4.2.4 第四步：编辑 pom.xml 文件



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <!-- 版本: 4.0.0 -->
    <modelVersion>4.0.0</modelVersion>
    <!-- 组织名称: 暂时使用 组织名称+项目名称 作为组织名称 -->
    <!-- 组织名称, 实际名称 按照访问路径规范设置, 通常以功能作为名称, eg: junit spring -->
    <groupId>com.itheima.maven</groupId>
    <!-- 项目名称 -->
    <artifactId>HelloMaven</artifactId>
    <!-- 当前项目版本号: 同一个项目开发过程中可以发布多个版本, 此处标示0.0.1版 -->
    <!-- 当前项目版本号: 每个工程发布后可以发布多个版本, 依赖时调取不同的版本, 使用不同的版本号 -->
    <version>0.0.1</version>
    <!-- 名称, 可省略 -->
    <name>Hello</name>

    <!-- 依赖关系 -->
    <dependencies>
        <!-- 依赖设置 -->
        <dependency>
            <!-- 依赖组织名称 -->
            <groupId>junit</groupId>
            <!-- 依赖项目名称 -->
            <artifactId>junit</artifactId>
            <!-- 依赖版本名称 -->
            <version>4.9</version>
            <!-- 依赖范围: test包下依赖该设置 -->
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

pom文件中这三个配置是必不可少的，它们指定来该项目的GAV坐标

## 4.3 Maven 命令的使用

Maven 的命令要在 pom.xml 所在目录中去执行

### 4.3.1 Mvn compile

编译的命令

### 4.3.2 Mvn clean

清除命令，清除已经编译好的 class 文件，具体说清除的是 target 目录中的文件

### 4.3.3 Mvn test

测试命令，该命令会将 test 目录中的源码进行编译

### 4.3.4 Mvn package

打包命令

### 4.3.5 Mvn install

安装命令，会将打好的包，安装到本地仓库

### 4.3.6 组合命令

#### 4.3.6.1 Mvn clean compile

先清空再编译

#### 4.3.6.2 mvn clean test 命令

cmd 中录入 mvn clean test 命令

组合指令，先执行 clean，再执行 test，通常应用于测试环节

### 4.3.6.3 mvn clean package 命令

cmd 中录入 mvn clean package 命令

组合指令，先执行 clean，再执行 package，将项目打包，通常应用于发布前执行过程：

清理———清空环境  
编译———编译源码  
测试———测试源码  
打包———将编译的非测试类打包

### 4.3.6.4 mvn clean install 命令

cmd 中录入 mvn clean install 查看仓库，当前项目被发布到仓库中

组合指令，先执行 clean，再执行 install，将项目打包，通常应用于发布前执行过程：

清理———清空环境  
编译———编译源码  
测试———测试源码  
打包———将编译的非测试类打包  
部署———将打好的包发布到资源仓库中

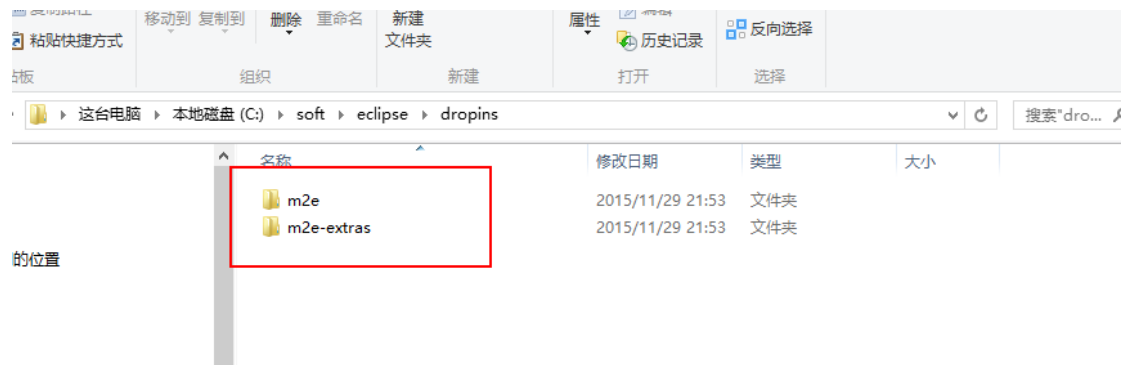
## 5 M2Eclipse

### 5.1 安装 M2Eclipse

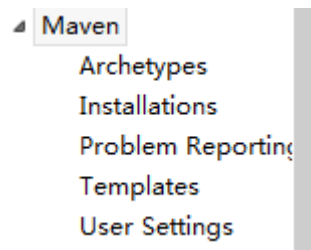
#### 5.1.1 第一步：将以下目录中的文件拷贝

组织	新建	打开	选择	
E:) > 03-teach > 03-讲课 > hm19 > 06-maven > 参考资料 > eclipse插件				
名称	修改日期	类型	大小	
m2e	2015/12/20 8:35	文件夹		
m2e-extras	2015/12/20 8:35	文件夹		

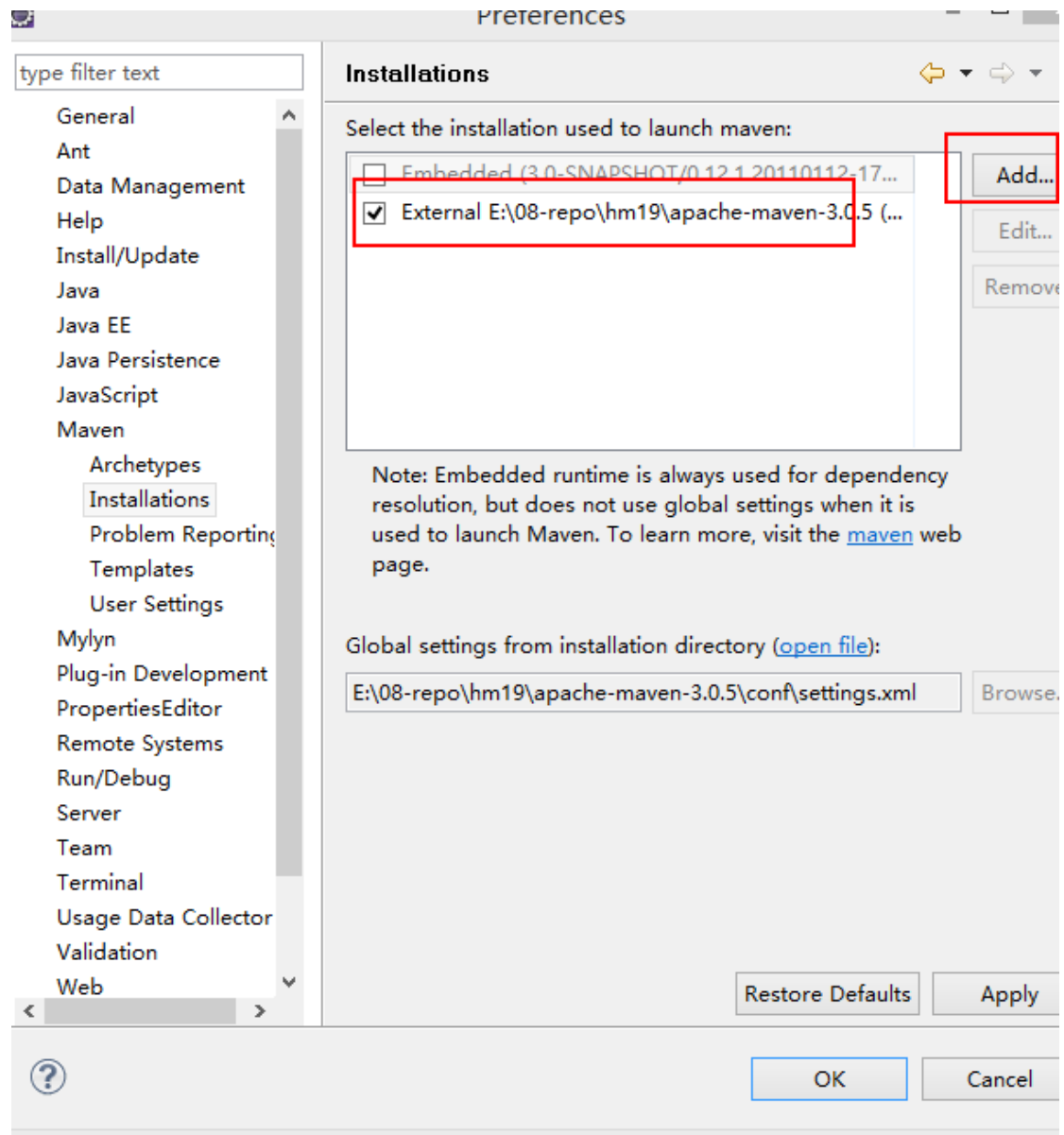
### 5.1.2 第二步：拷贝到 eclipse 中的 dropins 目录



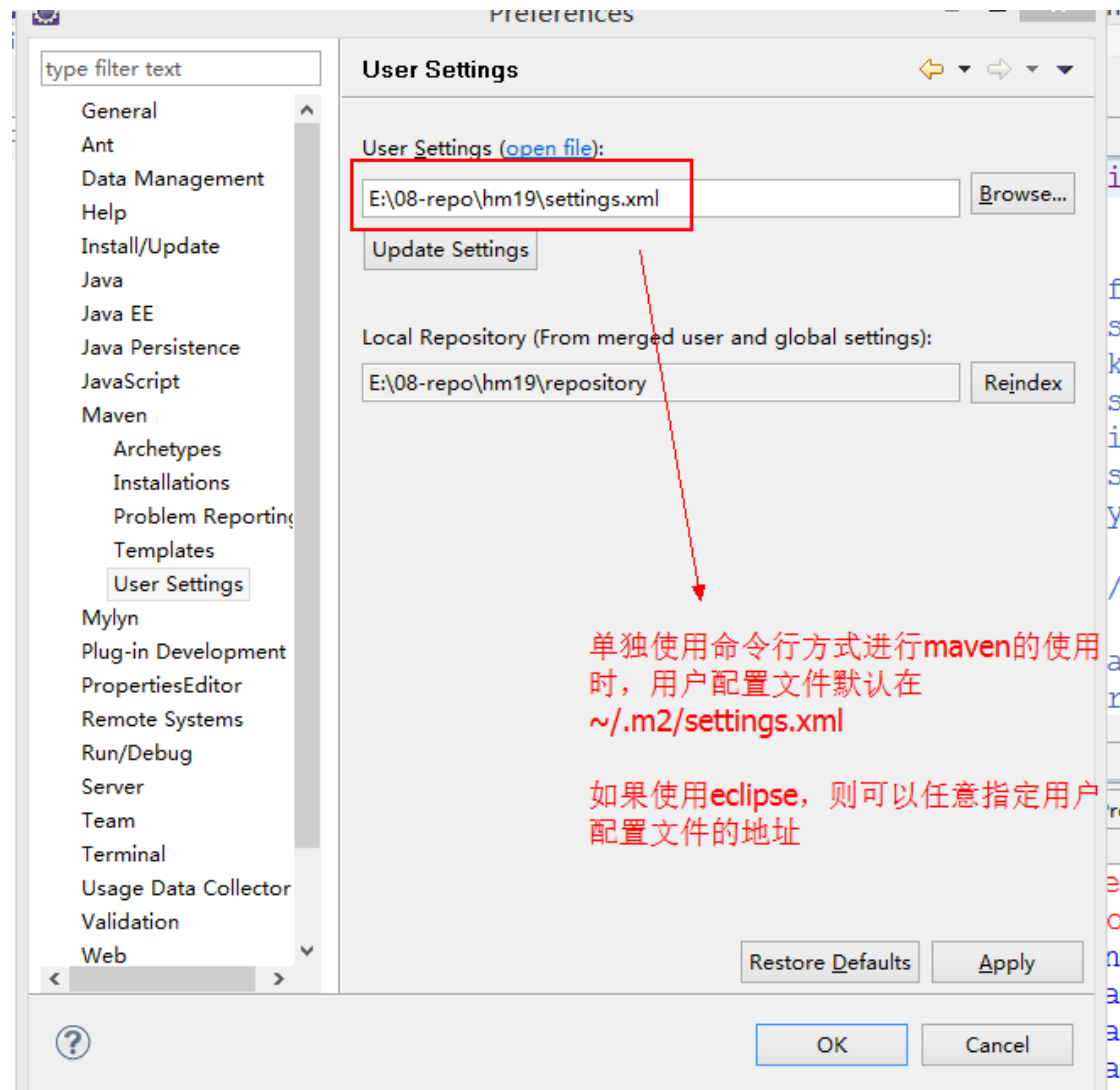
### 5.1.3 第三步：查看 eclipse 中是否安装成功



### 5.1.4 第四步：设置 maven 的安装路径

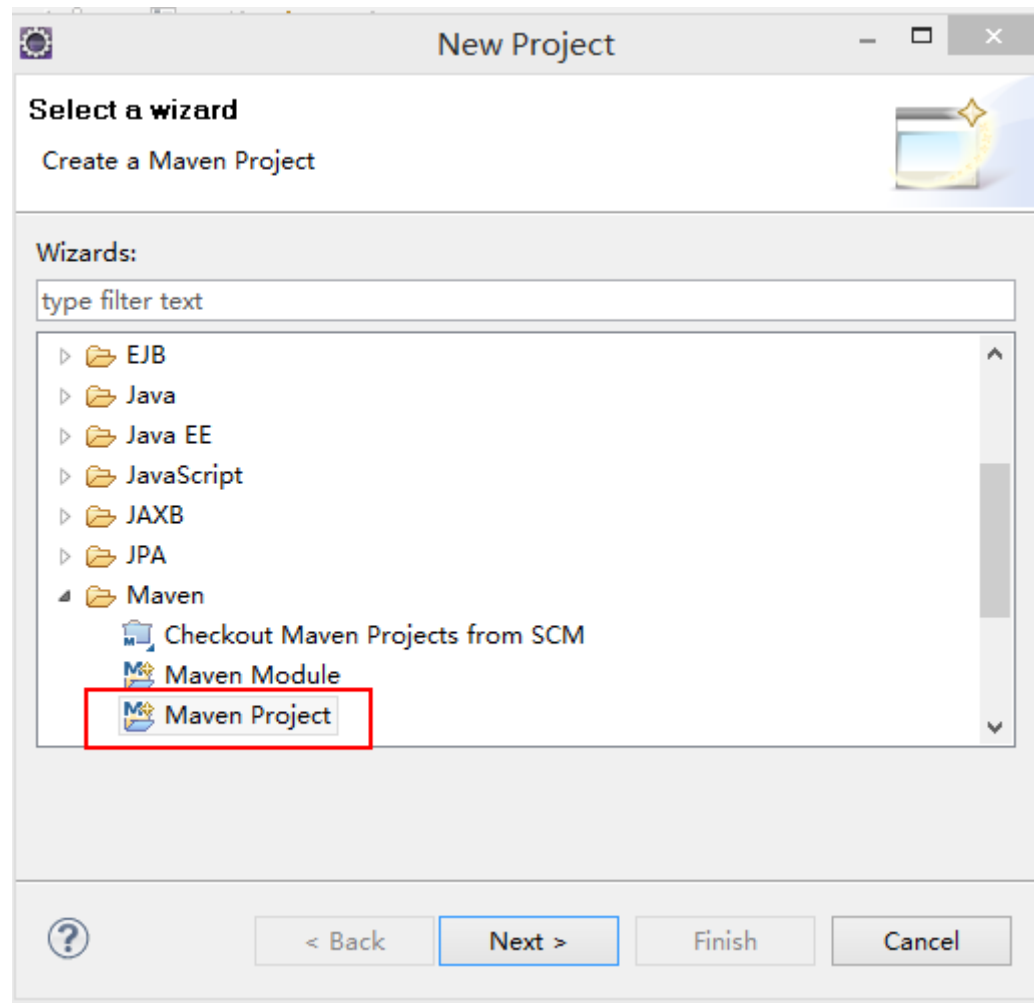


### 5.1.5 第五步：设置 maven 的用户配置

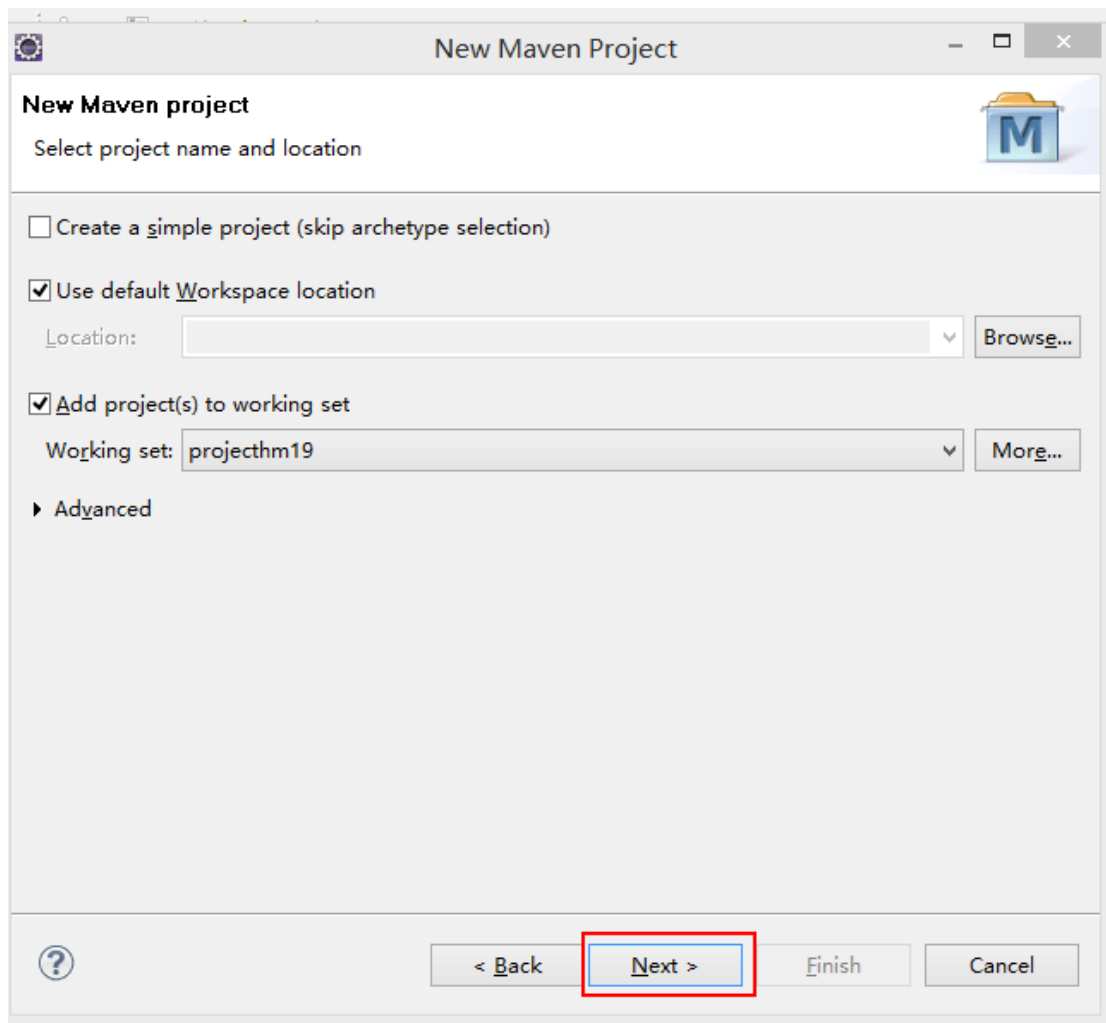


## 5.2 创建 MavenFirst 工程

第一步：创建 maven 工程

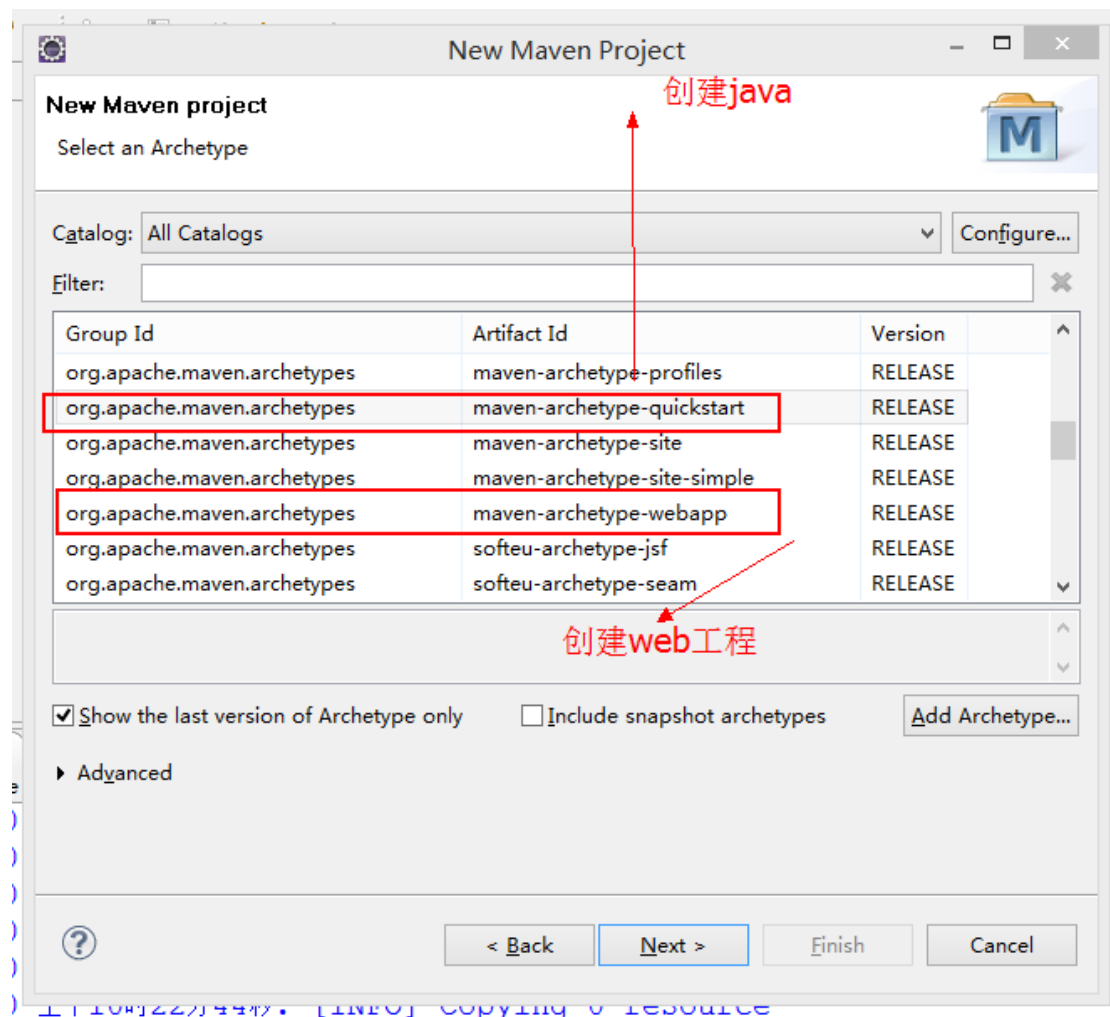


第二步：next



第三步：next





第四步：

New Maven Project

Specify Archetype parameters

Group Id: com.itheima.maven

Artifact Id: MavenFirst

Version: 0.0.1-SNAPSHOT

Package: com.itheima.maven.MavenFirst

Properties available from archetype:

Name	Value

指定工程的GAV坐标

Advanced

< Back Next > Finish Cancel

第五步：点击 finish，创建 maven 工程

第六步：创建 MavenFirst.java

```
16 */
17 public class MavenFirst {
18
19     public String sayHello(String name) {
20         return "hello " + name;
21     }
22 }
23
```

第七步：创建 TestMavenFirst.java

```

^ @version 1.0
*/
public class TestMavenFirst {

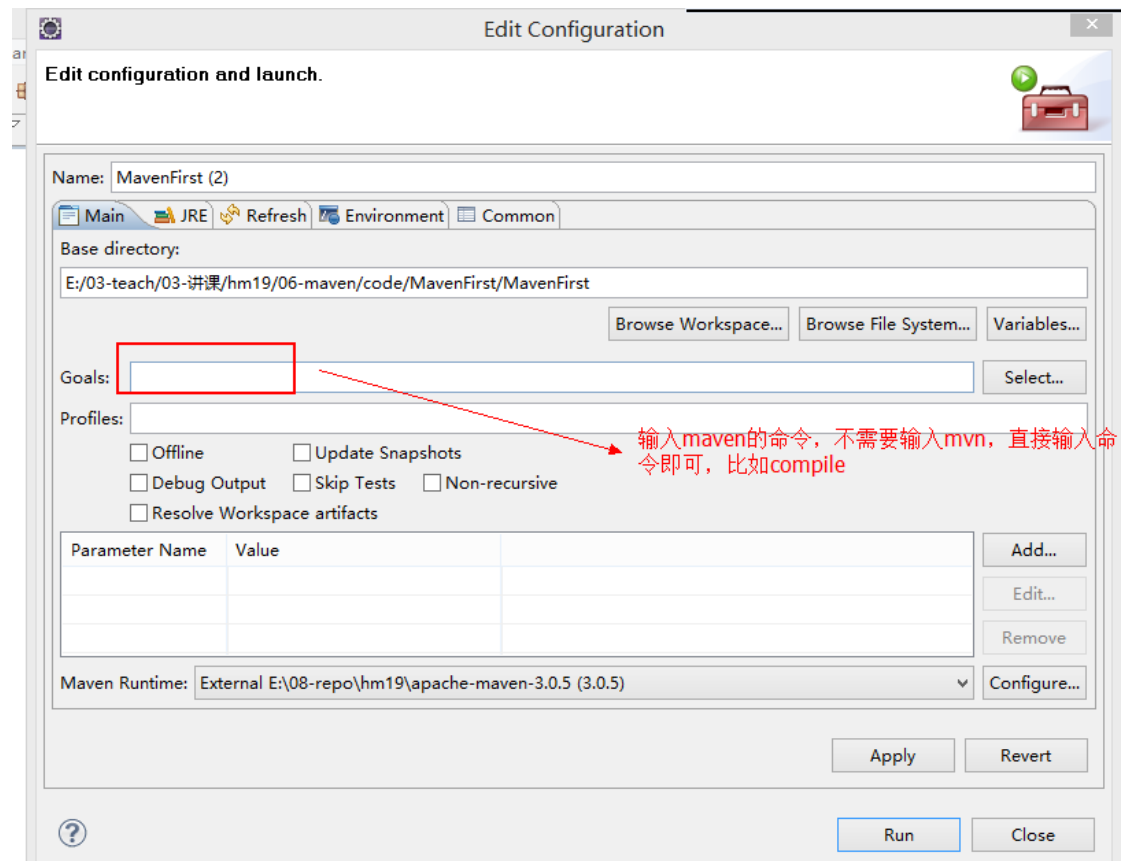
    @Test
    public void testSayHello() {
        MavenFirst first = new MavenFirst();

        String result = first.sayHello("hm19");

        Assert.assertEquals("hello hm19", result);
    }
}

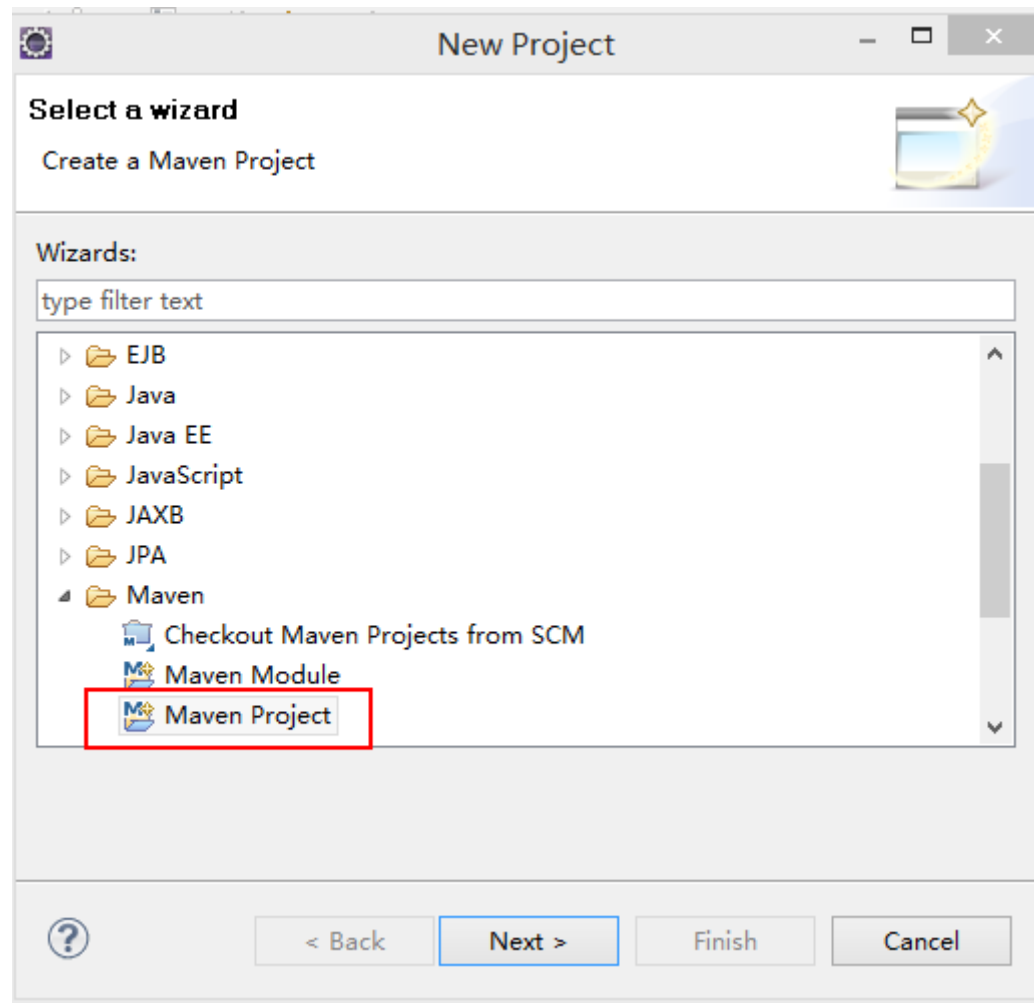
```

第七步：使用 eclipse 的选项执行 maven 命令

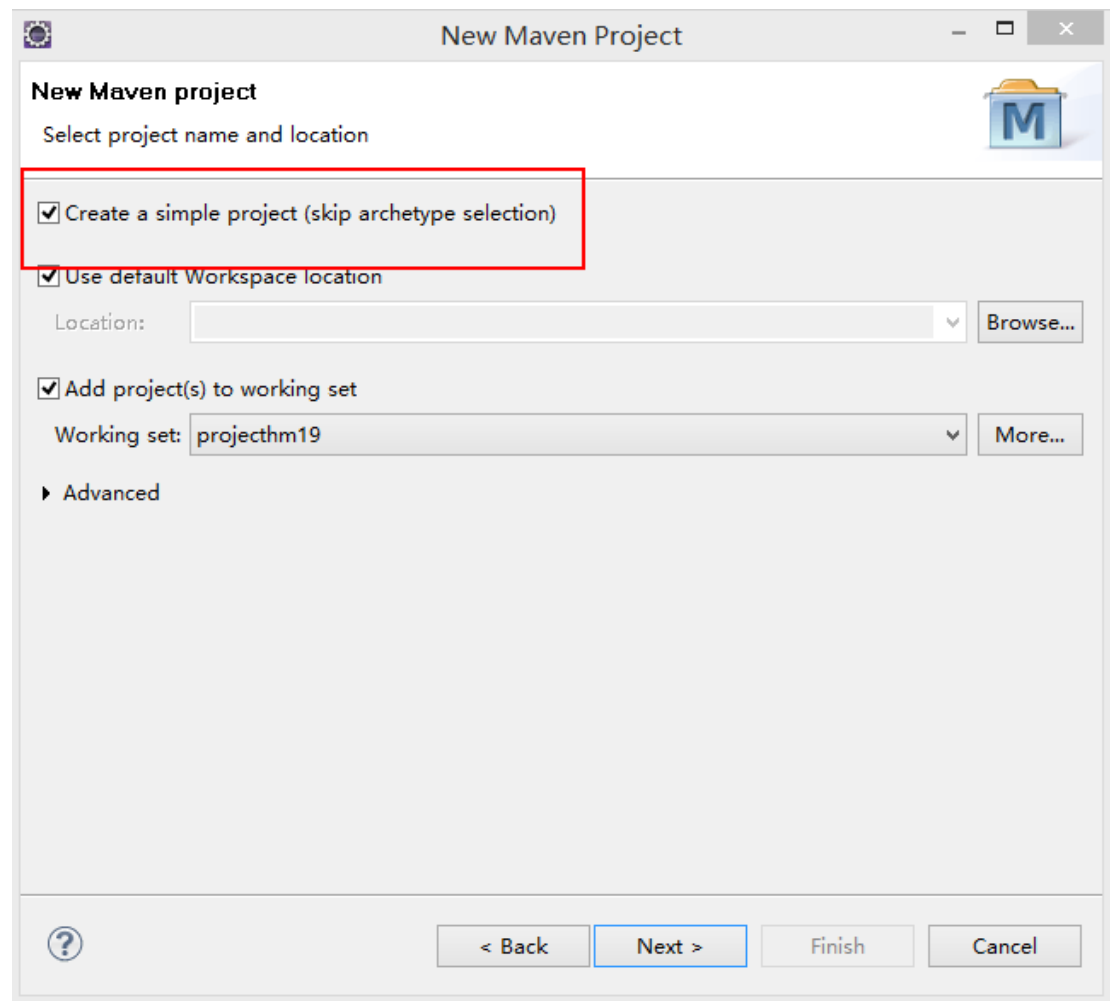


## 5.3 创建 MavenSecond 工程

第一步：创建 maven 工程



第二步：next



第三步：next

New Maven Project

Configure project

Artifact

Group Id: com.itheima.maven

Artifact Id: MavenSecond

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version:

Browse... Clear

Advanced

< Back Next > Finish Cancel

打包方式: jar、war、pom 默认是jar

第四步: finish, 创建 maven 工程

第五步: 创建 MavenSecond.java

```
10 public class MavenSecond {
11
12     public String sayHello(String name){
13         MavenFirst first = new MavenFirst();
14         return first.sayHello(name);
15     }
16 }
17
```

第六步: 创建 TestMavenSecond.java

```
15 public class TestMavenSecond {
16
17     @Test
18     public void sayHello() {
19         MavenSecond second = new MavenSecond();
20         String result = second.sayHello("hm19");
21
22         Assert.assertEquals("hello hm19", result);
23     }
24 }
```

## 6 Maven 的核心概念

### 6.1 坐标

#### 6.1.1 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。在 maven 中坐标就是为了定位一个唯一确定的 jar 包。

Maven 世界拥有大量构建，我们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

#### 6.1.2 Maven 坐标主要组成

**groupId:** 定义当前 Maven 组织名称

**artifactId:** 定义实际项目名称

**version:** 定义当前项目的当前版本

## 6.2 依赖管理

### 6.2.1 依赖范围

依赖范围 (Scope)	对于主代码 classpath有效	对于测试代码 classpath有效	被打包, 对于 运行时 classpath有效	例子
compile	Y	Y	Y	log4j
test	-	Y	-	junit
provided	Y	Y	-	servlet-api
runtime	-	-	Y	JDBC Driver Implementation

其中依赖范围 **scope** 用来控制依赖和编译, 测试, 运行的 classpath 的关系. 主要的是三种依赖关系如下:

- 1.compile: 默认编译依赖范围。对于编译, 测试, 运行三种 classpath 都有效
- 2.test: 测试依赖范围。只对于测试 classpath 有效
- 3.provided: 已提供依赖范围。对于编译, 测试的 classpath 都有效, 但对于运行无效。因为由容器已经提供, 例如 servlet-api
- 4.runtime:运行时提供。例如:jdbc 驱动

### 6.2.2 依赖传递

A、B、C

B 工程依赖 A 工程, C 工程依赖 B 工程, 那么 B 工程是 C 工程的直接依赖, A 工程是 C 工程的间接依赖

#### 6.2.2.1 创建 MavenThird 工程

第一步: 创建 mavenThird 工程

第二步: 创建 MavenThird.java



```

public class MavenThird {

    public String sayHello(String name){

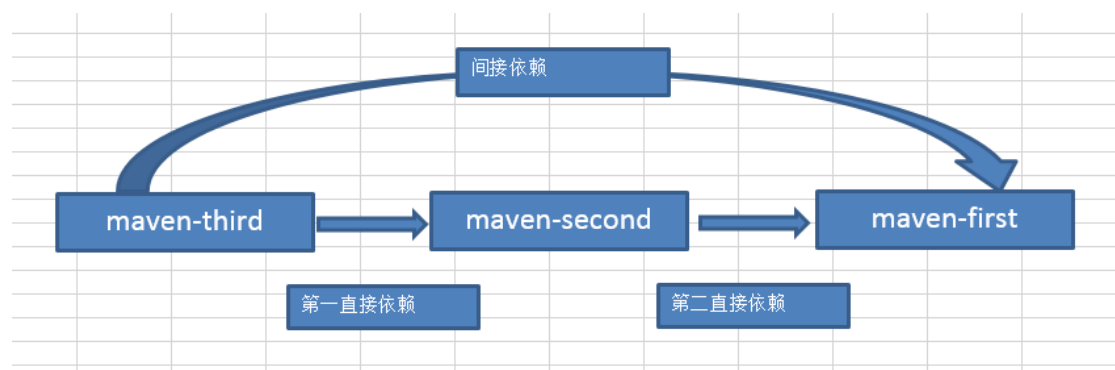
        MavenSecond second = new MavenSecond();

        return second.sayHello("hm19");
    }
}

```

第三步：创建 TestMavenThird.java

### 6.2.2.2 分析第一解决依赖和第二直接依赖



### 6.2.2.3 依赖范围传递

	compile	test	provided	runtime
compile	compile	-	-	runtime
test	test	-	-	test
provided	provided	-	provided	provided
runtime	runtime	-	-	runtime

左边第一列表示第一直接依赖范围  
 上面第一行表示第二直接依赖范围  
 中间的交叉单元格表示传递性依赖范围。

总结：

- 当第二依赖的范围是 **compile** 的时候，传递性依赖的范围与第一直接依赖的范围一致。
- 当第二直接依赖的范围是 **test** 的时候，依赖不会得以传递。
- 当第二依赖的范围是 **provided** 的时候，只传递第一直接依赖范围也为 **provided** 的依赖，且传递性依赖的范围同样为 **provided**；
- 当第二直接依赖的范围是 **runtime** 的时候，传递性依赖的范围与第一直接依赖的范围一致，但 **compile** 例外，此时传递的依赖范围为 **runtime**；

## 6.2.3 依赖冲突

在 maven 中存在两种冲突方式：一种是跨 pom 文件的冲突，一种是同一个 pom 文件中的冲突。

### 6.2.3.1 跨 pom 文件的冲突

MavenFirst 的 pom 文件中依赖来 junit 的 4.9 版本，那边 MavenSecond 和 MavenThird 中都是使用了 4.9 版本。

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.9</version>
  <scope>compile</scope>
</dependency>
</dependencies>
```

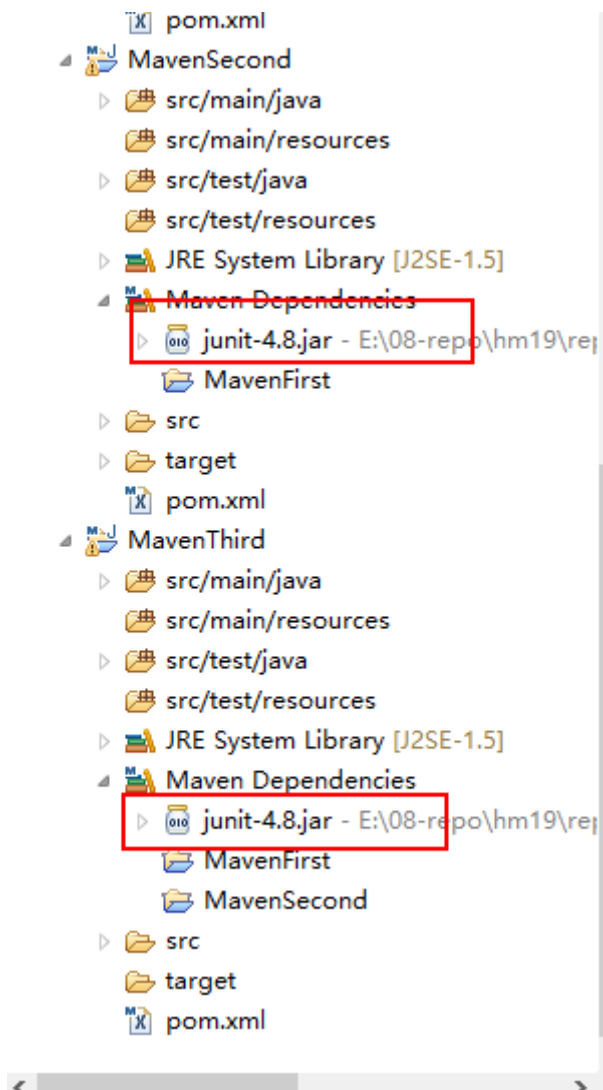
这是first工程的pom文件

如果 MavenSecond 中重新依赖 junit 的 4.8 版本，那么 MavenSecond 和 MavenThird 中都是使用了 4.8 本，这体现来依赖的**就近使用原则**。

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8</version>
</dependency>
```

这是second工程的pom文件

依赖的 jar 包如下：



### 6.2.3.2 同一个 pom 文件的冲突

```

<!-- 第二直接依赖 -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.7</version>
</dependency>
</dependencies>

```

这是second工程的pom文件  
按照就近原则，越下面的越近

## 6.2.4 可选依赖

Optional 标签标示该依赖是否可选，默认是 false。可以理解为，如果为 true，则表示该依赖不会传递下去，如果为 false，则会传递下去。

```
<dependency>
  <groupId>com.itheima.maven</groupId>
  <artifactId>MavenFirst</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <optional>true</optional>
</dependency>

<!-- 第二直接依赖 -->
```

这是second工程的pom文件

## 6.2.5 排除依赖

Exclusions 标签可以排除依赖

```
<dependencies>
  <!-- 第一直接依赖 -->
  <dependency>
    <groupId>com.itheima.maven</groupId>
    <artifactId>MavenSecond</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <exclusions>
      <exclusion>
        <groupId>com.itheima.maven</groupId>
        <artifactId>MavenFirst</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
</project>
```

在third工程的pom文件中排除依赖

## 6.3 生命周期

Maven 有三个生命周期：clean 生命周期、default 生命周期、site 生命周期

生命周期可以理解为项目构建的步骤集合。

生命周期是由多个阶段（Phase）组成。每个阶段都是一个完整的功能，比如 `mvn clean` 中的 `clean` 就是一个阶段。

### 6.3.1 Clean 生命周期

`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` 命令，等同于 `mvn pre-clean clean`。只要执行后面的命令，那么前面的命令都会执行，不需要再重新去输入命令。

有 Clean 生命周期，在生命周期又有 `clean` 阶段。

### 6.3.2 Default 生命周期（重点）

`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`  
**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。

在 **maven** 中，只要在同一生命周期，你执行后面的阶段，那么前面的阶段也会被执行，而且不需要额外去输入前面的阶段，这样大大减轻了程序员的工作。

### 6.3.3 Site 生命周期

```
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
```

## 6.4 插件

插件（**plugin**），每个插件都能实现一个阶段的功能。**Maven** 的核心是生命周期，但是生命周期相当于主要指定了 **maven** 命令执行的流程顺序，而没有真正实现流程的功能，功能是有插件来实现的。

比如：**compile** 就是一个插件实现的功能。

### 6.4.1 编译插件

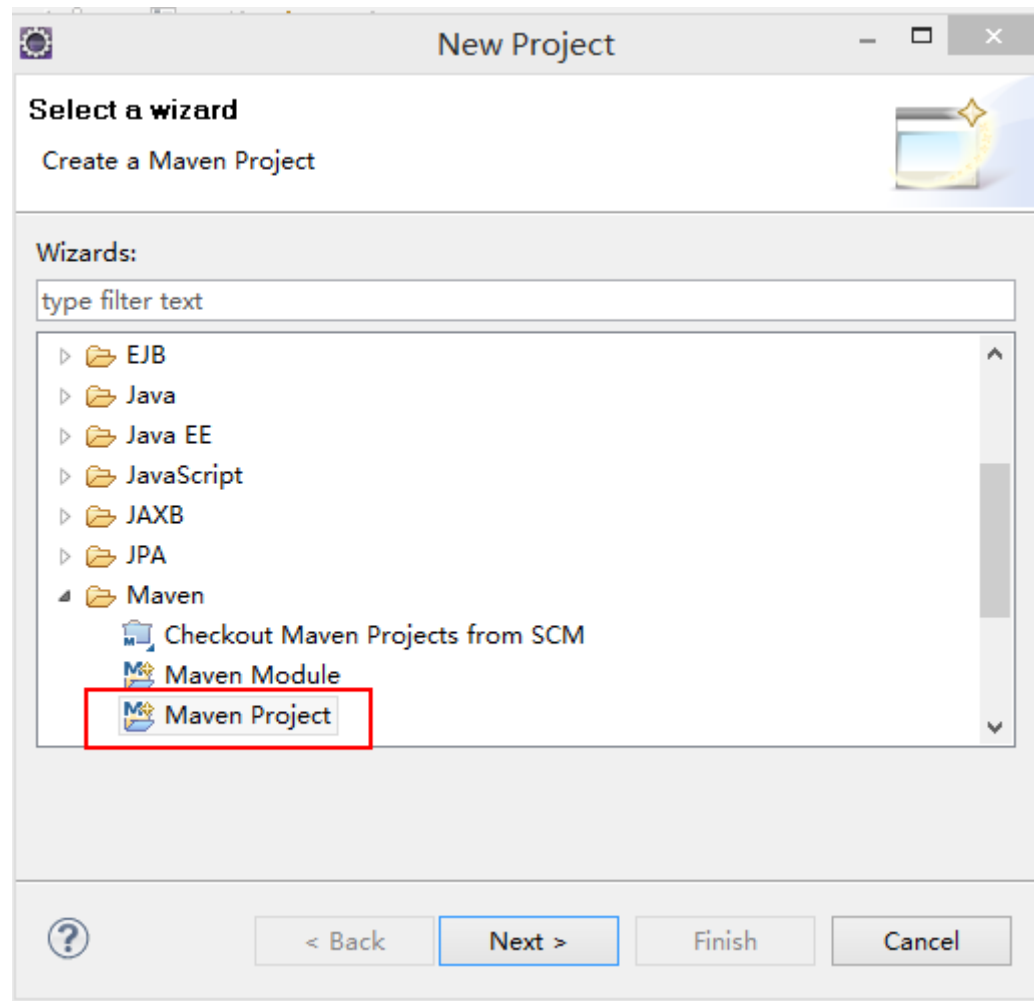
```
<build>
  <plugins>
    <!-- 编译插件，指定编译用的jdk版本 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

### 6.4.2 Tomcat 插件

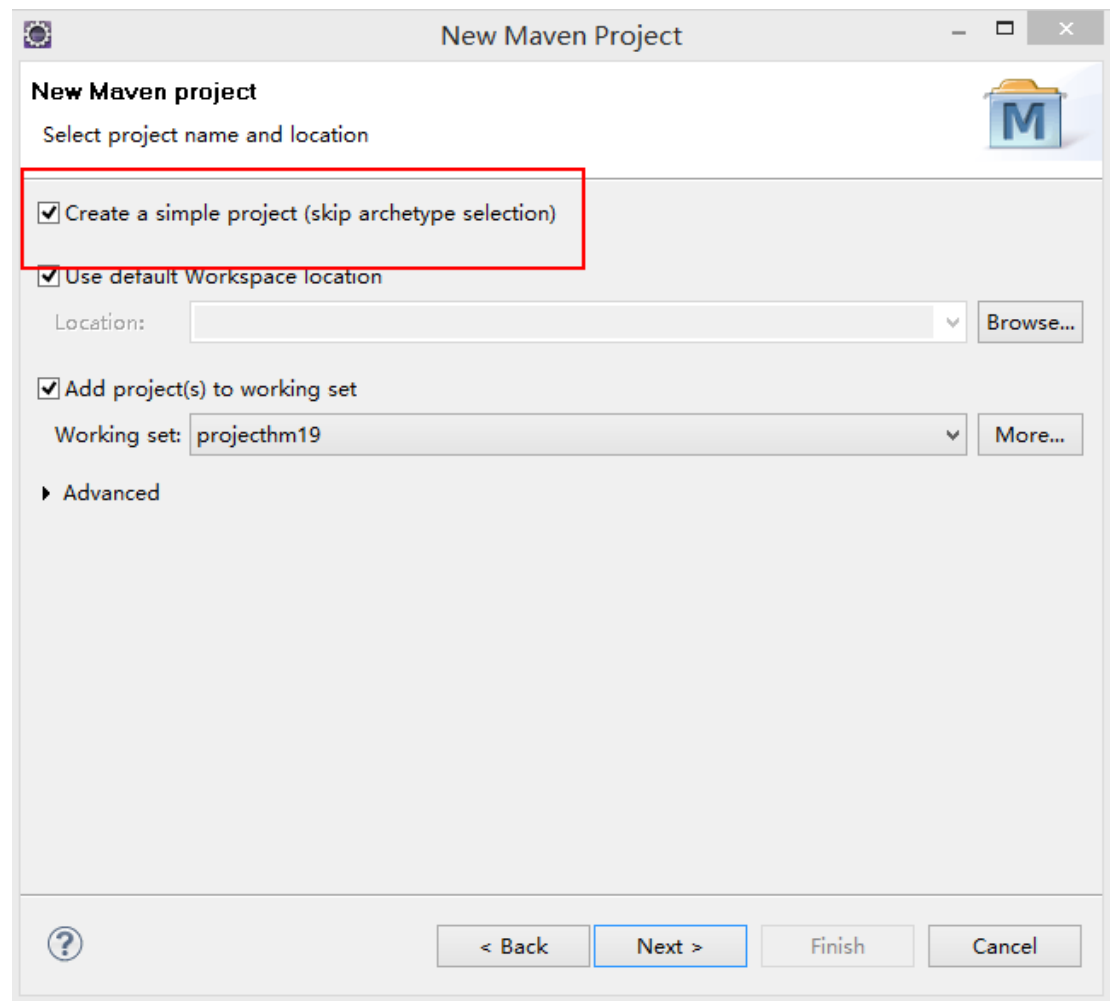
如果使用 **maven** 的 **tomcat** 插件的话，那么本地则不需要安装 **tomcat**。

### 6.4.2.1 创建 maven 的 web 工程

第一步：创建 maven 工程



第二步：next



第三步：next



**New Maven Project**

Configure project

**Artifact**

Group Id:

Artifact Id:

Version:

Packaging:

Name:

Description:

**Parent Project**

Group Id:

Artifact Id:

Version:

▶ Advanced

打包方式要选择war方式才能创建web工程

第四步：点击 finish 创建 maven 工程

第五步：创建 WEB-INF 及 web.xml 文件

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:
xsi:schemaLocation="http://java.sun.com/xml/ns
id="WebApp_ID" version="2.5">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

第六步：创建 index.jsp 文件

```
8 </head>
9 <body>
10     第一个maven web 工程
11 </body>
12 </html>
```

#### 6.4.2.2 使用 tomcat 插件运行 web 工程

默认输入 `tomcat:run` 去使用 `tomcat` 插件来启动 `web` 工程，但是默认的 `tomcat` 插件使用的 `tomcat` 版本是 `tomcat6`

而目前主流的 `tomcat`，是使用的 `tomcat7`，需要手动配置 `tomcat` 插件

```
<build>
  <plugins>
    <plugin>
      <!-- 配置插件 -->
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <configuration>
        <port>80</port>
        <path>/</path>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

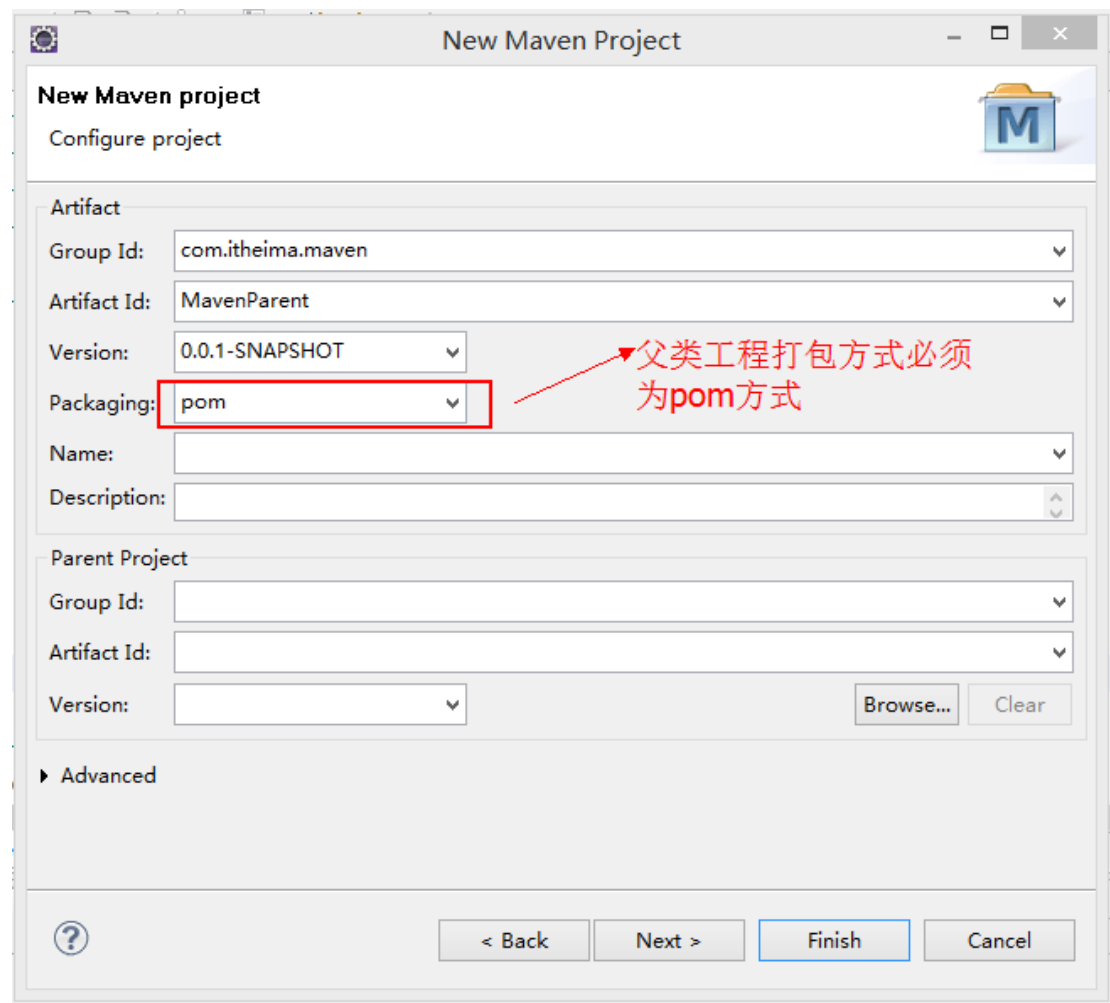
这是tomcat7的插件配置

使用 `tomcat7` 来运行 `web` 工程，它的命令是：**`tomcat7:run`**

### 6.5 继承

在 `maven` 中的继承，指的是 `pom` 文件的继承

## 6.5.1 创建父工程



New Maven Project

Configure project

Artifact

Group Id: com.itheima.maven

Artifact Id: MavenParent

Version: 0.0.1-SNAPSHOT

Packaging: pom

Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version:

Browse... Clear

Advanced

< Back Next > Finish Cancel

父类工程打包方式必须为pom方式

## 6.5.2 创建子工程

创建子工程有两种方式：一种是创建一个新的工程为子工程，另一种是修改老的工程为子工程。

创建新工程为子工程的方式：

New Maven Project

Configure project

Artifact

Group Id: com.itheima.maven

Artifact Id: MavenSub

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name:

Description:

Parent Project

Group Id: com.itheima.maven

Artifact Id: MavenParent

Version: 0.0.1-SNAPSHOT

Browse... Clear

Advanced

在这指定父工程的GAV

< Back Next > Finish Cancel

子工程的 pom 文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>MavenParent</artifactId>
    <groupId>com.itheima.maven</groupId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <groupId>com.itheima.maven</groupId>
  <artifactId>MavenSub</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</project>
```

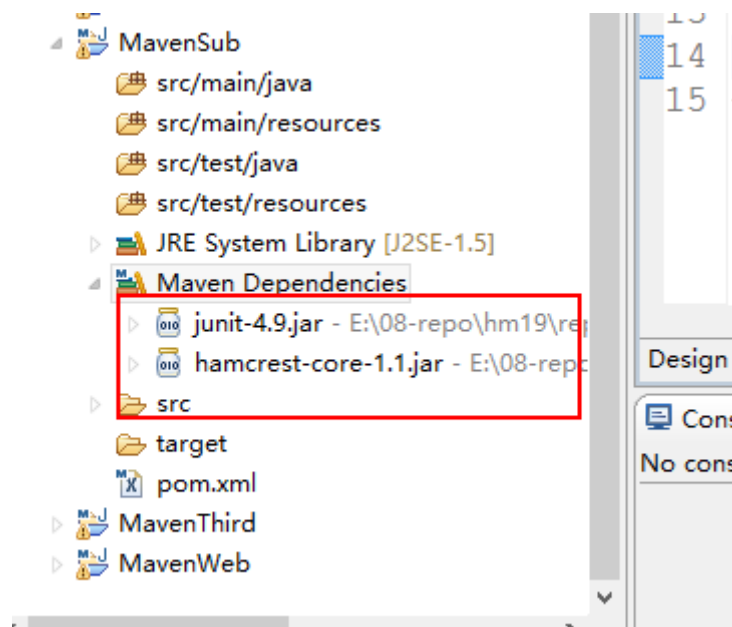
子工程中会有一个parent标签来指定父工程的GAV

### 6.5.3 父工程统一依赖 jar 包

在父工程中对 jar 包进行依赖，在子工程中都会继承此依赖。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.itheima.maven</groupId>
  <artifactId>MavenParent</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.9</version>
    </dependency>
  </dependencies>
</project>
```



## 6.5.4 父工程统一管理版本号

dependencyManagement 标签管理的依赖，其实没有真正依赖，它只是管理依赖的版本。

子工程的 pom 文件：

```
----- , -----  
  
<dependencies>  
  <dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
  </dependency>  
</dependencies>  
</project>
```

在子工程中使用父工程管理的版本号，此时不需要指定 version

## 6.5.5 父工程抽取版本号

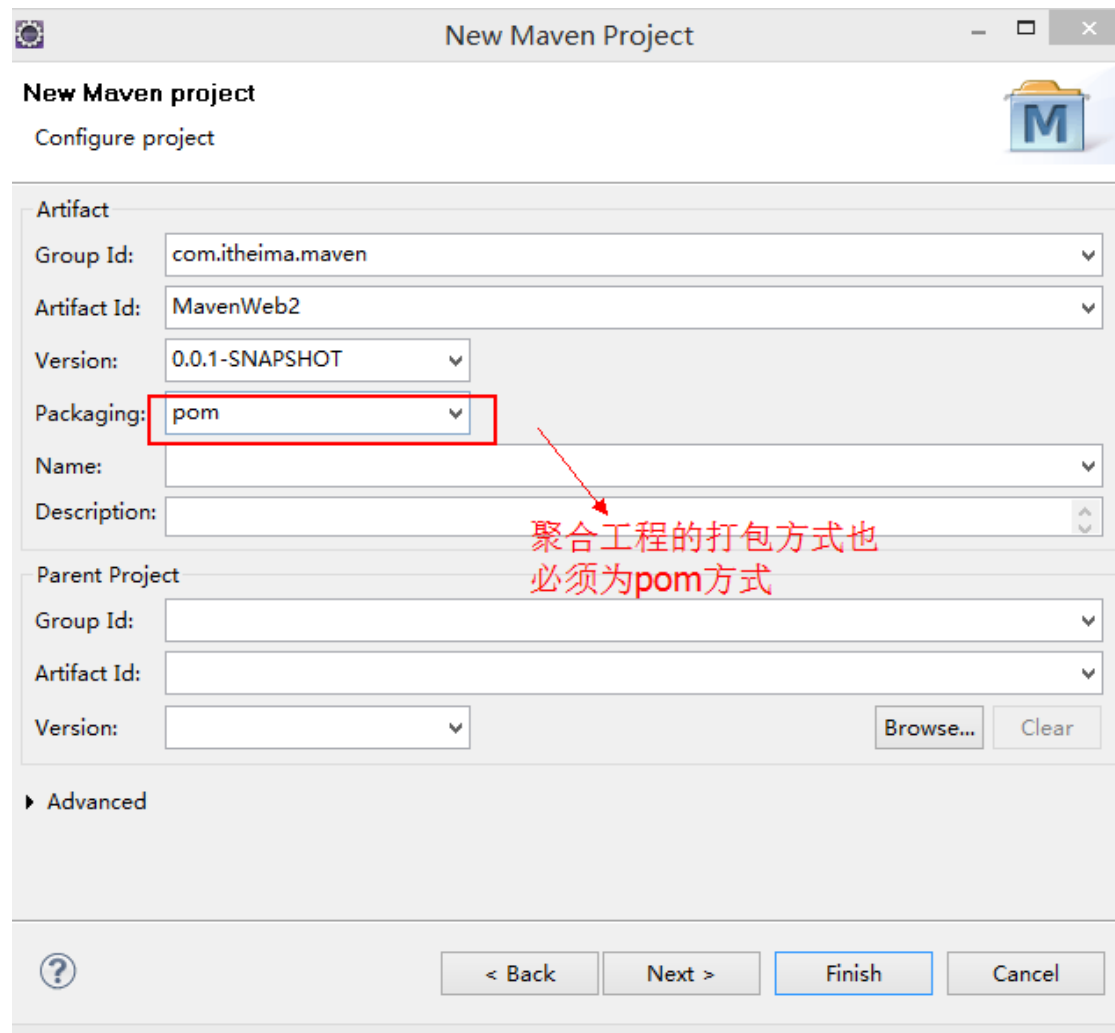
```
<properties>  
  <junit.version>4.9</junit.version>  
</properties>  
  
<dependencyManagement>  
  <dependencies>  
    <dependency>  
      <groupId>junit</groupId>  
      <artifactId>junit</artifactId>  
      <version>${junit.version}</version>  
    </dependency>  
  </dependencies>  
</dependencyManagement>  
</project>
```

properties 中的子标签可以任意指定

## 6.6 聚合

在真实项目中，一个项目有表现层、业务层、持久层，对于业务层和持久层，它们可以在多个工程中被使用，所以一般会将业务层和持久单独创建为 java 工程，为其他工程依赖。

## 6.6.1 创建一个聚合工程



**New Maven Project**

Configure project

**Artifact**

Group Id: com.itheima.maven

Artifact Id: MavenWeb2

Version: 0.0.1-SNAPSHOT

Packaging: pom

Name:

Description:

**Parent Project**

Group Id:

Artifact Id:

Version:

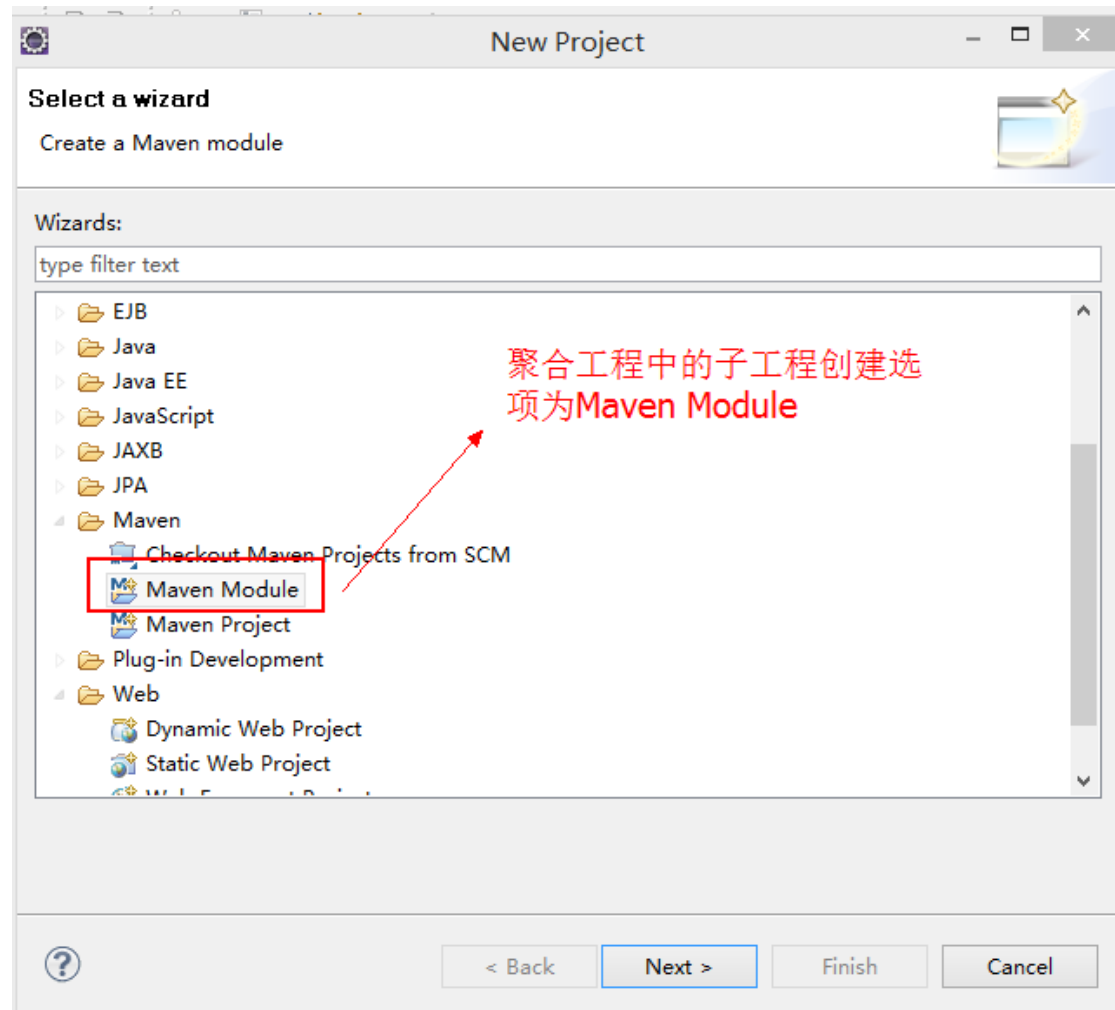
Browse... Clear

► Advanced

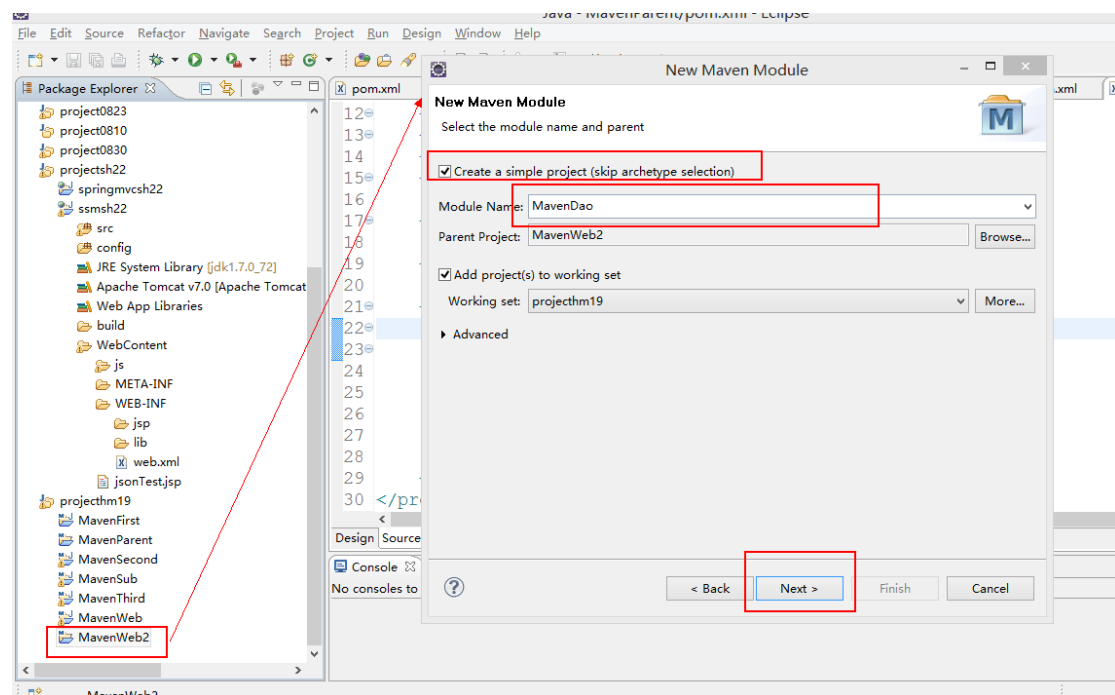
< Back Next > Finish Cancel

聚合工程的打包方式也必须为pom方式

## 6.6.2 创建持久层





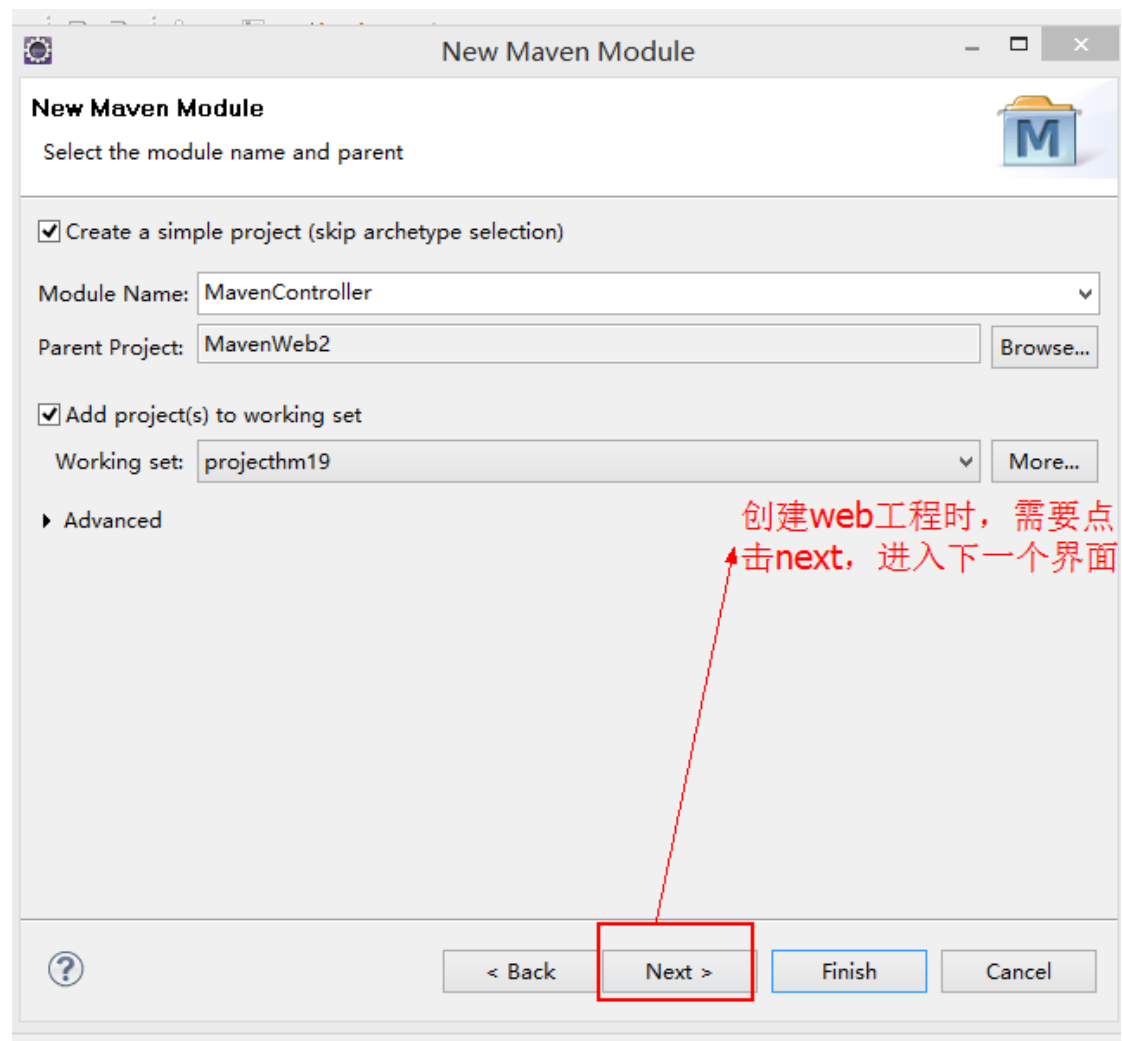


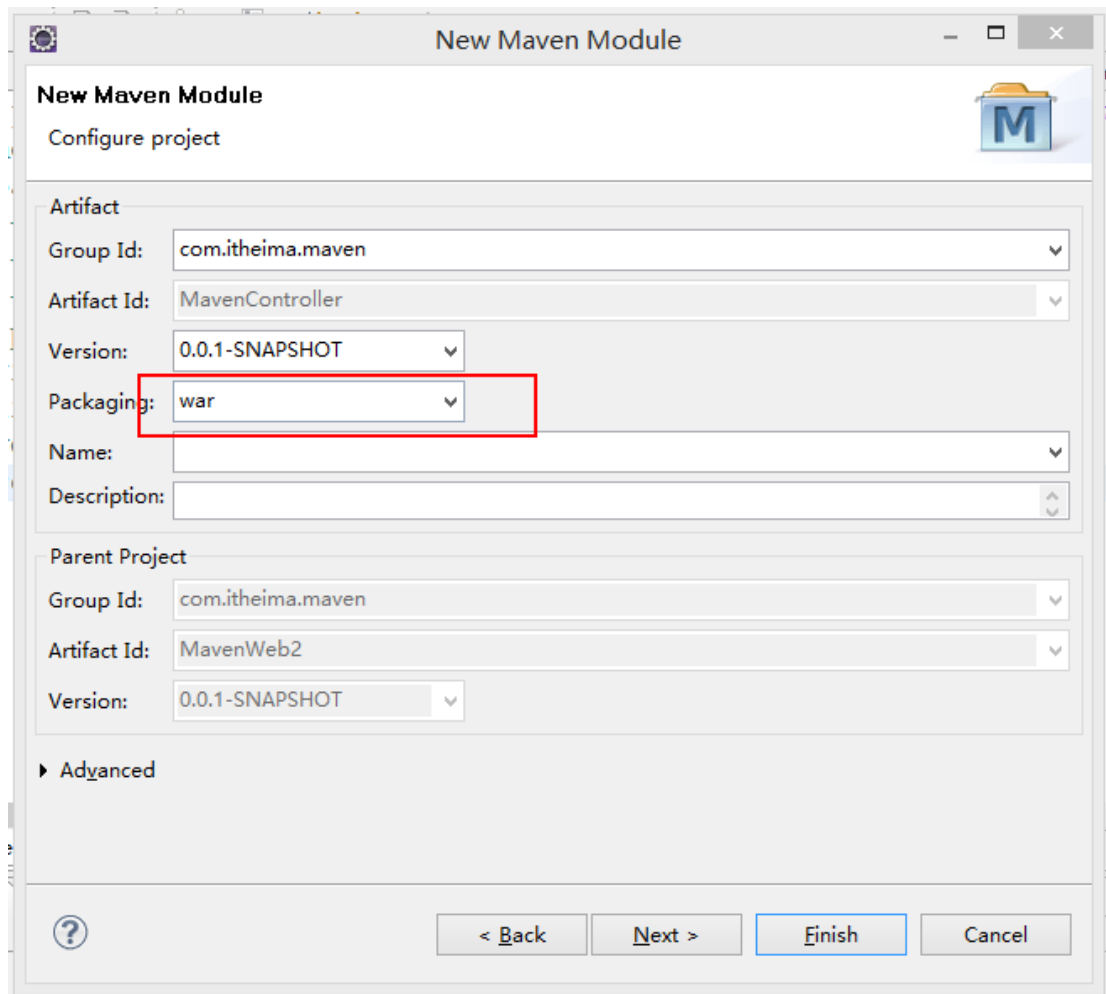
```
pom.xml | web.xml | web.xml | index.jsp | pom.xml | pom.xml | pom.xml | pom.xml | po
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 <modelVersion>4.0.0</modelVersion>
4 <parent>
5 <artifactId>MavenWeb2</artifactId>
6 <groupId>com.itheima.maven</groupId>
7 <version>0.0.1-SNAPSHOT</version>
8 </parent>
9 <groupId>com.itheima.maven</groupId>
10 <artifactId>MavenDao</artifactId>
11 <version>0.0.1-SNAPSHOT</version>
12 </project>
```

## 6.6.3 创建业务层

和持久层的创建一样

## 6.6.4 表现层





## 6.6.5 聚合为一个工程来运行

聚合工程的 pom 文件：



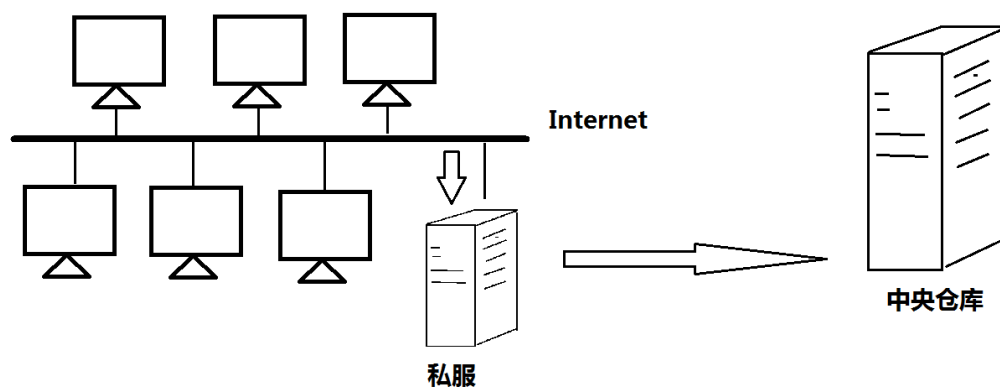
## 7 Maven 仓库管理

### 7.1 什么是 Maven 仓库？

用来统一存储所有 Maven 共享构建的位置就是仓库。根据 Maven 坐标定义每个构建在仓库中唯一存储路径大致为：groupId/artifactId/version/artifactId-version.packaging

### 7.2 仓库的分类

- 本地仓库  
默认在~/.m2/repository，如果在用户配置中有配置，则以用户配置的地址为准
- 远程仓库
  - 中央仓库（不包含有版本的 jar 包）  
<http://repo1.maven.org/maven2>
  - 私服



## 7.3 Maven 私服

### 7.3.1 安装 Nexus

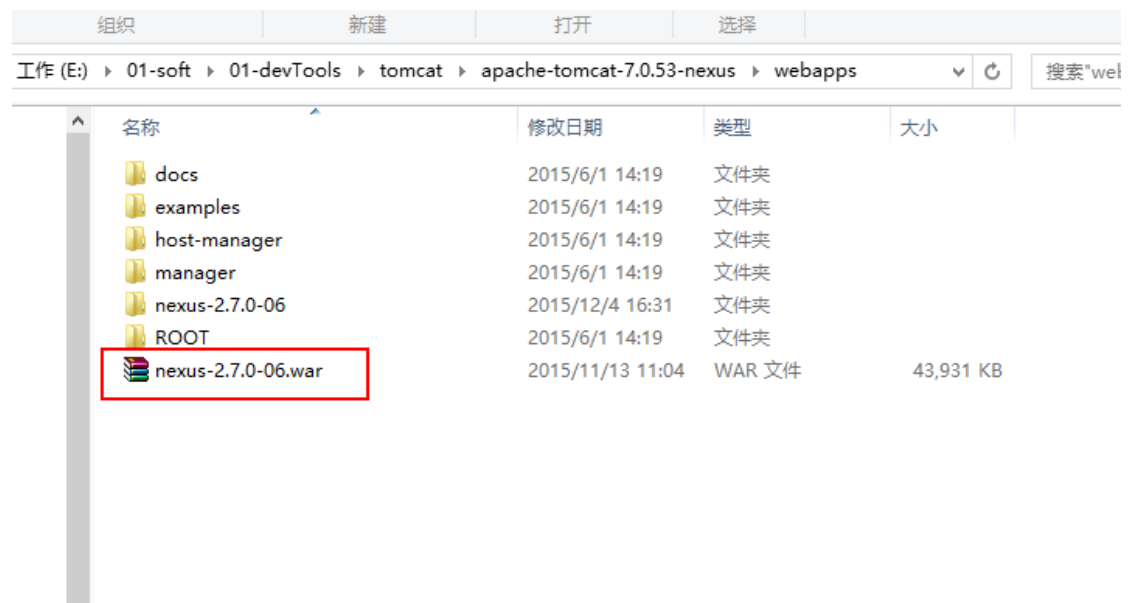
为所有来自中央仓库的构建安装提供本地缓存。

下载网站: <http://nexus.sonatype.org/>

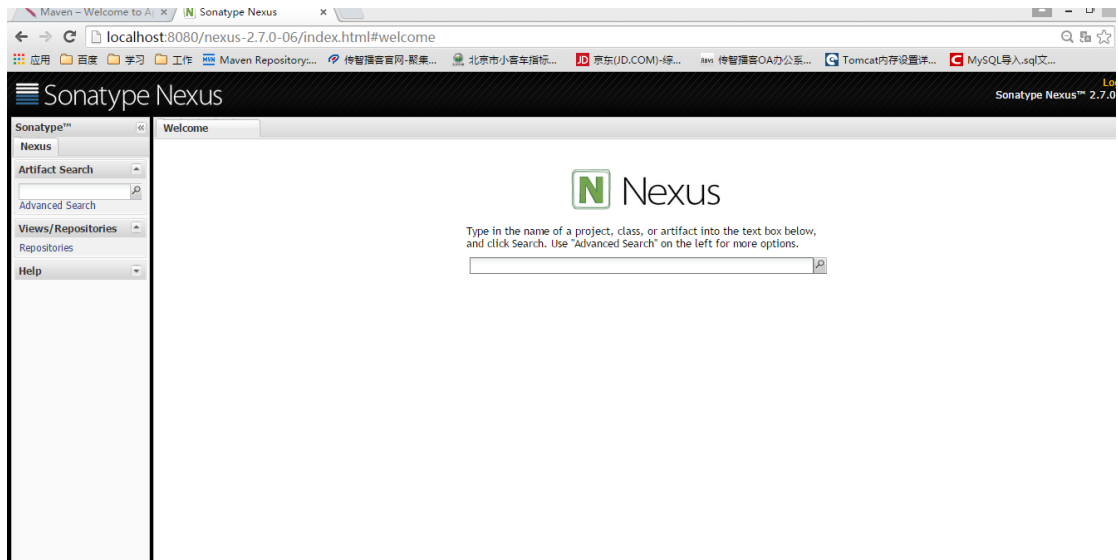
安装版本: nexus-2.7.0-06.war

第一步: 安装 tomcat

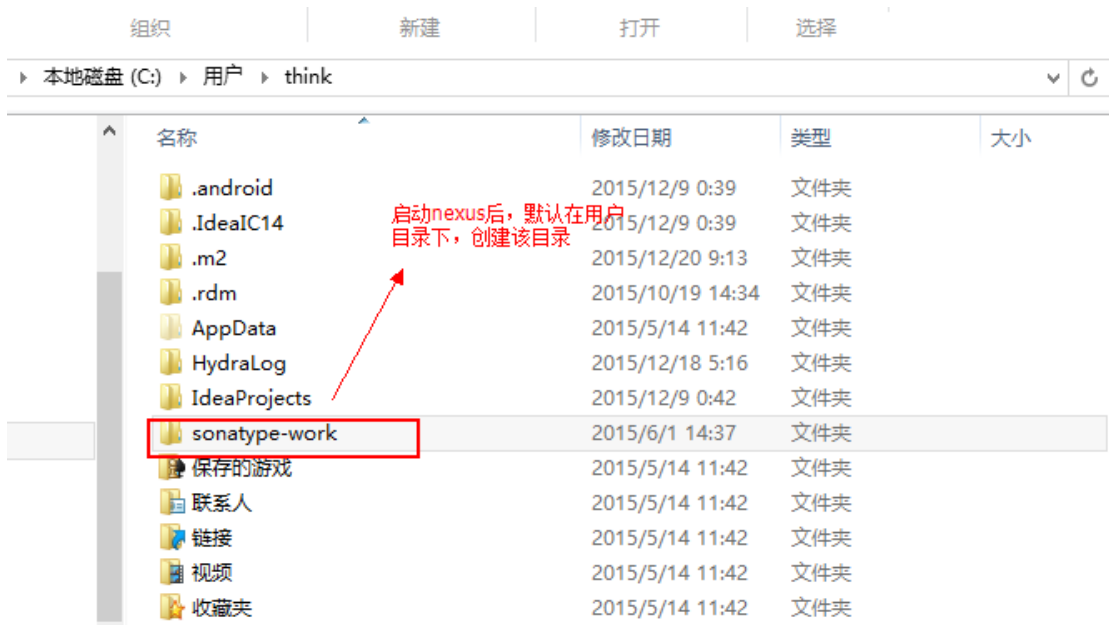
第二步: 将 nexus 的 war 包拷贝到 tomcat 的 webapps 下

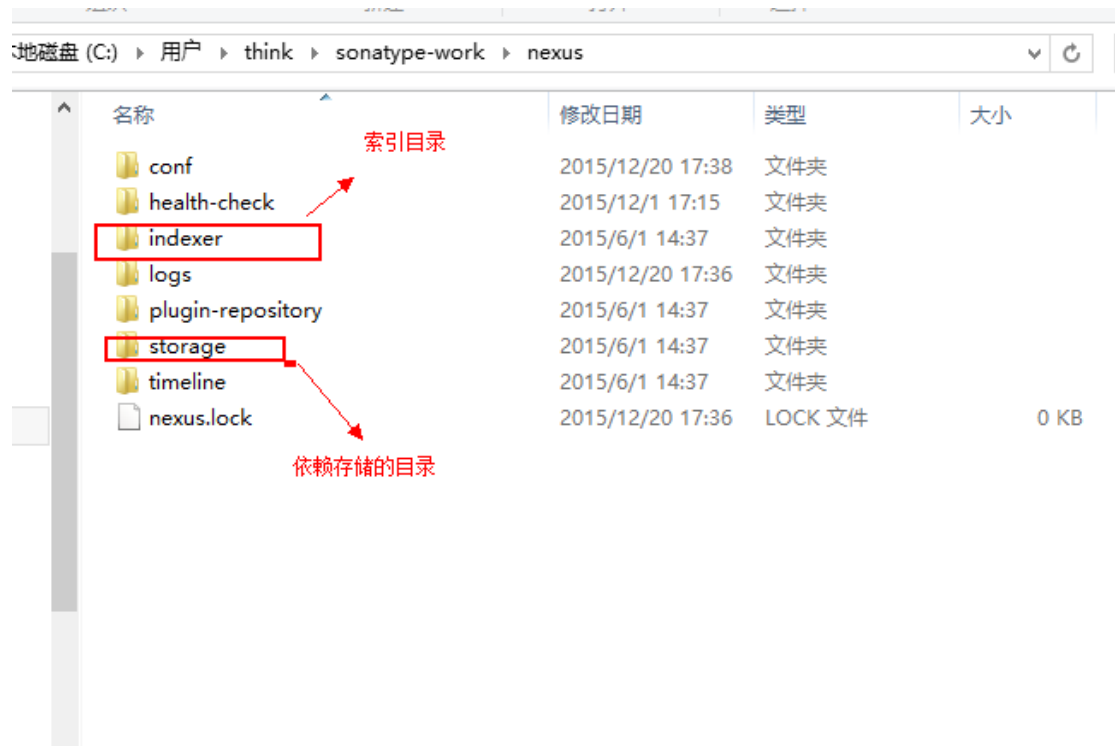


第三步: 启动 tomcat



#### 第四步：nexus 的本地目录





## 7.3.2 访问 Nexus

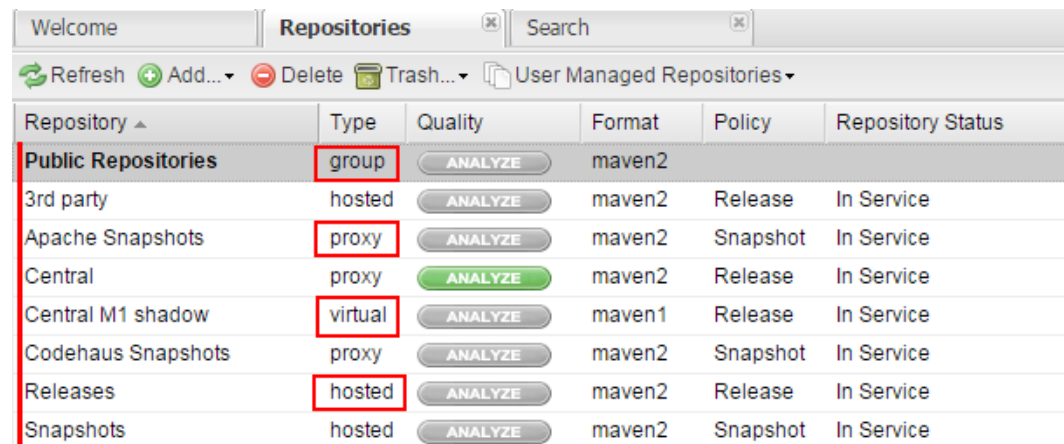
访问 URL: <http://localhost:8080/nexus-2.7.0-06/>

默认账号:

用户名: admin

密码: admin123

### 7.3.3 Nexus 的仓库和仓库组



Repository ▲	Type	Quality	Format	Policy	Repository Status
Public Repositories	group	ANALYZE	maven2		
3rd party	hosted	ANALYZE	maven2	Release	In Service
Apache Snapshots	proxy	ANALYZE	maven2	Snapshot	In Service
Central	proxy	ANALYZE	maven2	Release	In Service
Central M1 shadow	virtual	ANALYZE	maven1	Release	In Service
Codehaus Snapshots	proxy	ANALYZE	maven2	Snapshot	In Service
Releases	hosted	ANALYZE	maven2	Release	In Service
Snapshots	hosted	ANALYZE	maven2	Snapshot	In Service

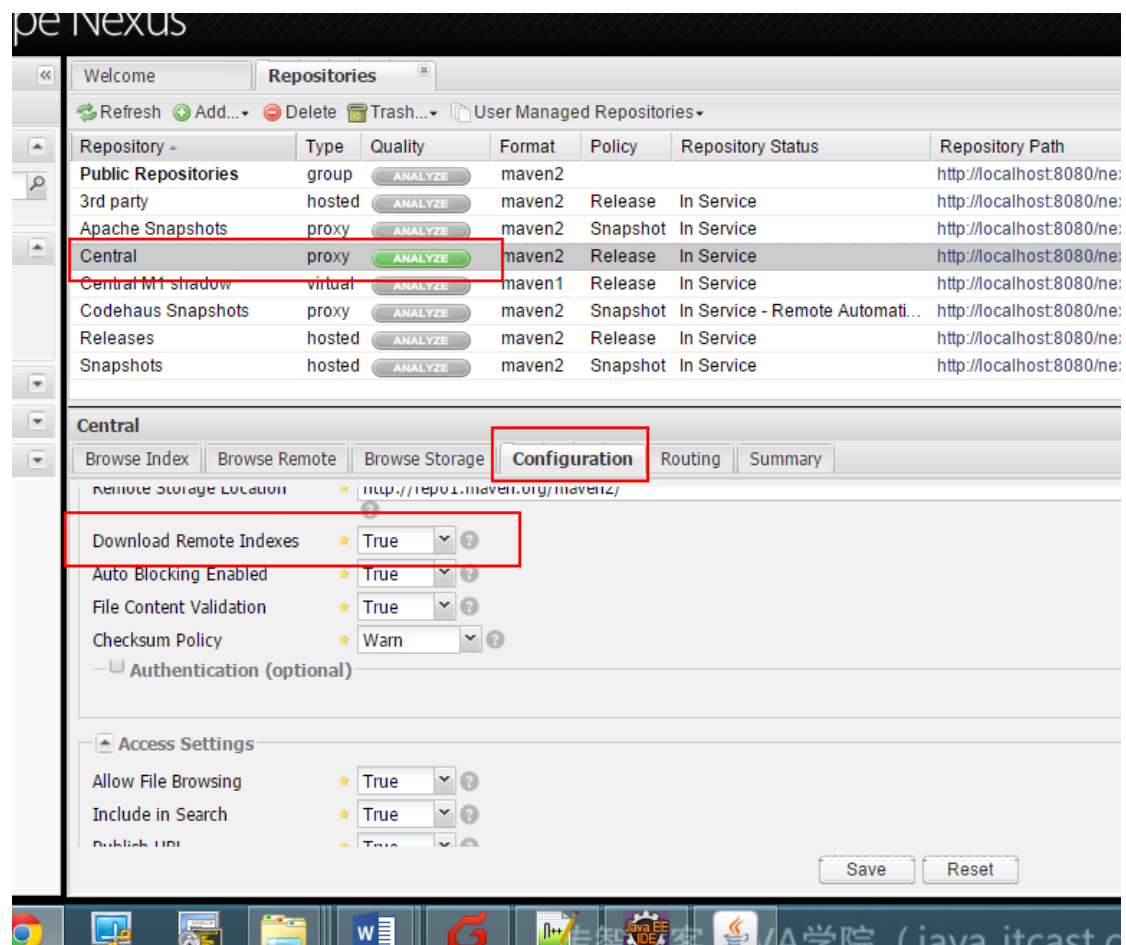
仓库有 4 种类型：

- group(仓库组)：一组仓库的集合
- hosted(宿主)：配置第三方仓库（包括公司内部私服）
- proxy(代理)：私服会对中央仓库进行代理，用户连接私服，私服自动去中央仓库下载 jar 包或者插件
- virtual(虚拟)：兼容 Maven1 版本的 jar 或者插件

Nexus 的仓库和仓库组介绍：

- 3rd party: 一个策略为 Release 的宿主类型仓库，用来部署无法从公共仓库获得的第三方发布版本构建
- Apache Snapshots: 一个策略为 Snapshot 的代理仓库，用来代理 Apache Maven 仓库的快照版本构建
- Central: 代理 Maven 中央仓库
- Central M1 shadow: 代理 Maven1 版本 中央仓库
- Codehaus Snapshots: 一个策略为 Snapshot 的代理仓库，用来代理 Codehaus Maven 仓库的快照版本构件
- Releases: 一个策略为 Release 的宿主类型仓库，用来部署组织内部的发布版本构件
- Snapshots: 一个策略为 Snapshot 的宿主类型仓库，用来部署组织内部的快照版本构件
- **Public Repositories: 该仓库组将上述所有策略为 Release 的仓库聚合并通过一致的地址提供服务**





## 7.3.4 配置所有构建均从私服下载

在本地仓库的 setting.xml 中配置如下：

```
<mirrors>
  <mirror>
    <!--此处配置所有的构建均从私有仓库中下载 *代表所有，也可以写 central -->
    <id>nexus</id>
    <mirrorOf>*</mirrorOf>
    <url>http://localhost:8080/nexus-2.7.0-06/content/groups/public/</url>
  </mirror>
</mirrors>
```

```

<!-- mirror
| Specifies a repository mirror site to use instead of a given repository. Th
| this mirror serves has an ID that matches the mirrorOf element of this mirr
| for inheritance and direct lookup purposes, and must be unique across the s
|-->
<mirror>
  <id>nexus</id>
  <mirrorOf>*</mirrorOf>
  <name>nexus</name>
  <url>http://localhost:8080/nexus-2.7.0-06/content/groups/public/</url>
</mirror>
</mirrors>

```

可以配置\*或者central

## 7.3.5 部署构建到 Nexus

### 7.3.5.1 第一步：Nexus 的访问权限控制

在本地仓库的 setting.xml 中配置如下：

```

<server>
  <id>releases</id>
  <username>admin</username>
  <password>admin123</password>
</server>
<server>
  <id>snapshots</id>
  <username>admin</username>
  <password>admin123</password>
</server>

```

### 7.3.5.2 第二步：配置 pom 文件

在需要构建的项目中修改 pom 文件

```

<distributionManagement>
  <repository>
    <id>releases</id>
    <name>Internal Releases</name>
    <url>http://localhost:8080/nexus-2.7.0-
06/content/repositories/releases/</url>
  </repository>
</distributionManagement>

```

```
</repository>
<snapshotRepository>
  <id>snapshots</id>
  <name>Internal Snapshots</name>
  <url>http://localhost:8080/nexus-2.7.0-
06/content/repositories/snapshots/</url>
</snapshotRepository>
</distributionManagement>
```

### 7.3.5.3 第三步：执行 maven 的 deploy 命令

