Maven

项目管理工具

# 课程计划

1. maven的介绍
2. maven的安装配置
3. 创建maven工程
4. M2Eclipse（eclipse插件，右键创建maven工程）
5. Maven的核心概念
   1. 坐标
   2. 依赖管理
   3. 生命周期
   4. 插件
   5. 继承
   6. 聚合
6. maven的仓库管理

# maven的介绍

## 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？-缺少依赖

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

1. 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。

## 什么是maven

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

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（depend）、项目构建（build）。

依赖指的就是jar包。

## 什么是构建

程序员1

程序员2

程序员3

Svn服务器

仿真环境

下载源码

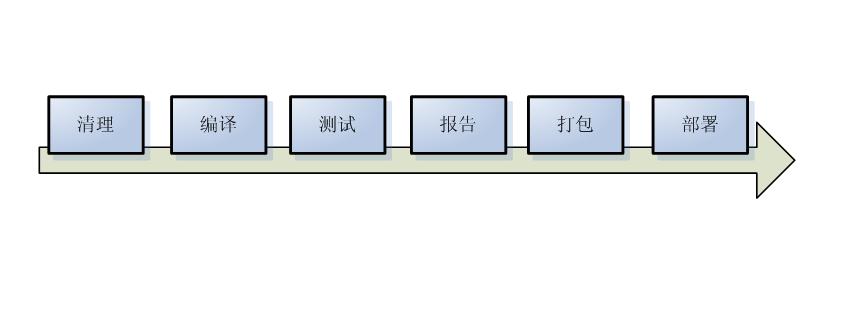
项目构建

编译、打包、部署

Tomcat

测试

**构建过程：**



## 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

1. 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

1. **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，但是需要按照maven的规范去进行代码的开发。也就是说maven是有约束的。

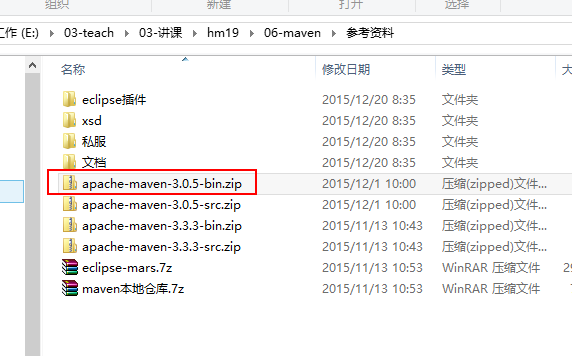
# Maven的安装配置

## 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

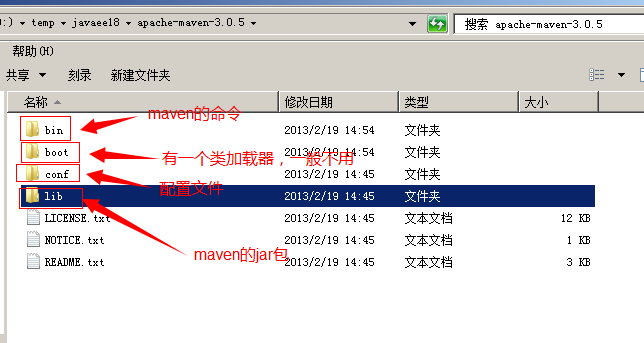
Maven是使用java开发，需要安装jdk1.6以上，推荐使用1.7



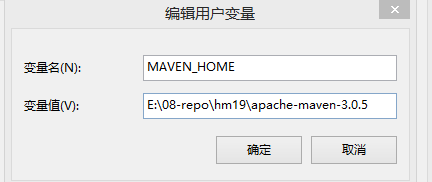
## 安装maven

第一步：安装jdk1.6及以上

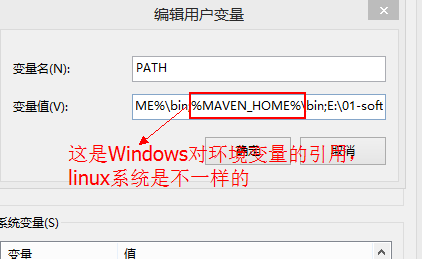
第二步：将maven下载的压缩包进行解压缩



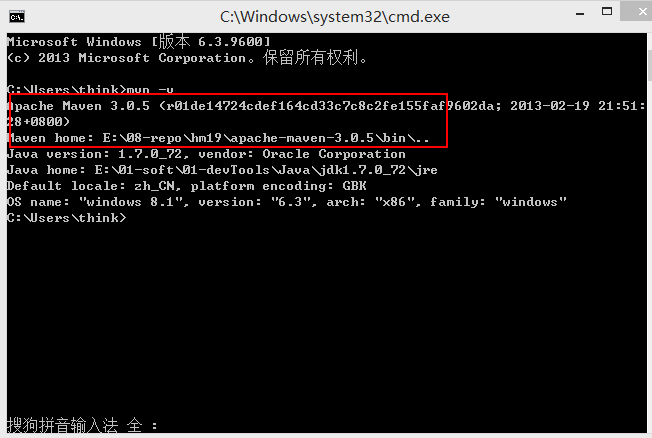
第三步：配置maven的环境变量MAVEN\_HOME



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



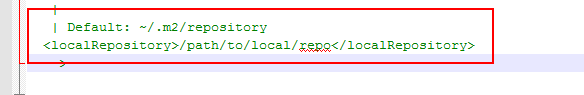
## 配置maven

在maven中有两个配置文件：用户配置、全局配置（默认）(类似sublime)

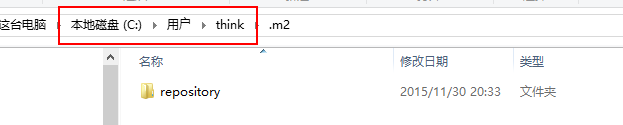
### 全局配置

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

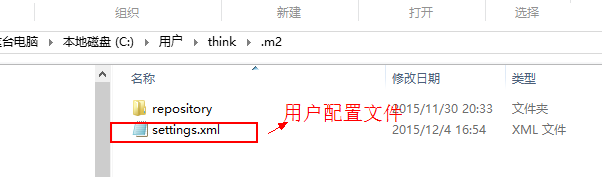


默认在系统的用户目录下的m2/repository中，该目录是本地仓库的目录（下载的依赖包会保存在这个目录中，可以通过localRepository来自定义存储路径）。

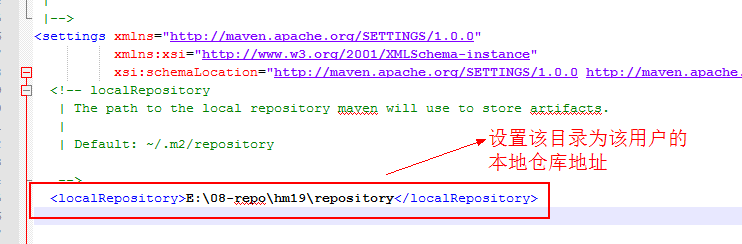


### 用户配置

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（me:类似sublime的默认配置和用户配置,用户配置能够覆盖同名的默认配置）。



# 创建maven工程

## Maven工程结构

gradle管理的项目也是这个结构

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

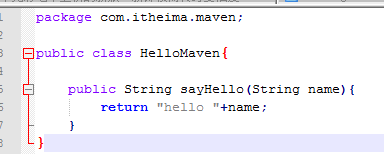
|--target（class文件、报告等信息存储的地方）

|--pom.xml（maven工程的描述文件）

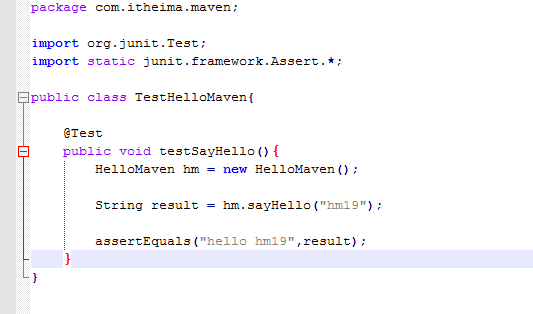
## 创建HelloMaven工程

### 第一步：安装maven的工程结构创建helloMaven工程

### 第二步：创建HelloMaven.java



### 第三步：创建TestHelloMaven.java



### 第四步：编辑pom.xml文件



## Maven命令的使用

Maven的命令要在pom.xml所在目录中去执行（gradle也是如此）

### Mvn compile

编译的命令

### Mvn clean

第一次运行可能会下载一些插件

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

### Mvn test

测试命令，该命令会将test目录中的源码进行编译

### Mvn package

打包命令

### Mvn install

安装命令，会将打好的包，安装到本地仓库（即将包发布到本地配置的maven仓库）

### 组合命令

#### Mvn clean compile

先清空再编译

#### 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

#### 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

#### 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

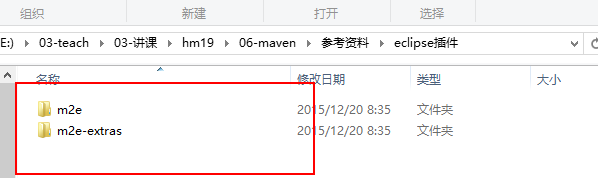
打包————将编译的非测试类打包

部署————将打好的包发布到资源仓库中

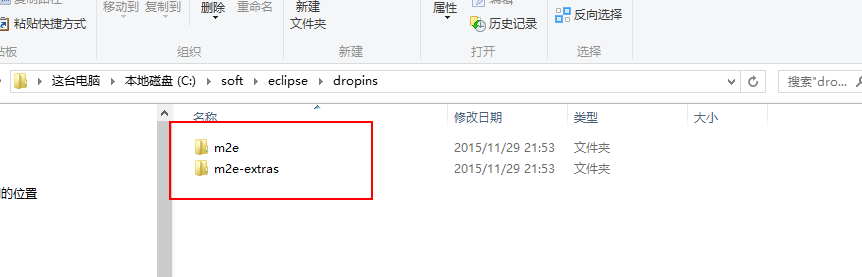
# M2Eclipse

## 安装M2Eclipse

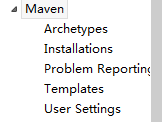
### 第一步：将以下目录中的文件拷贝



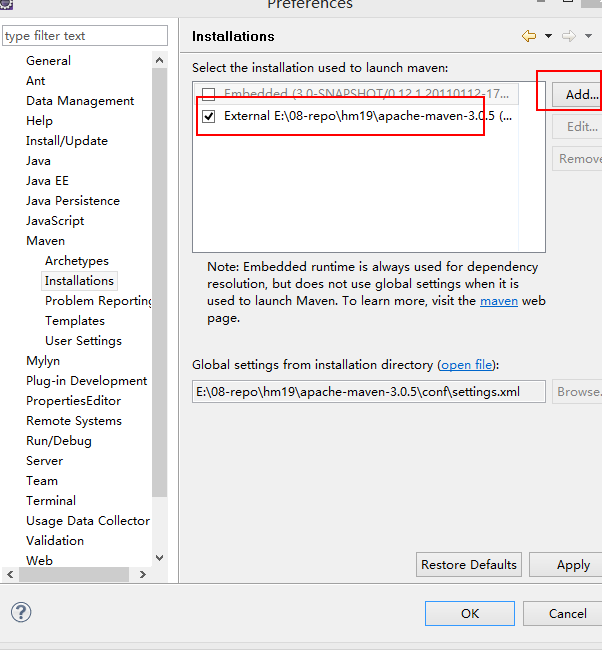
### 第二步：拷贝到eclipse中的dropins目录



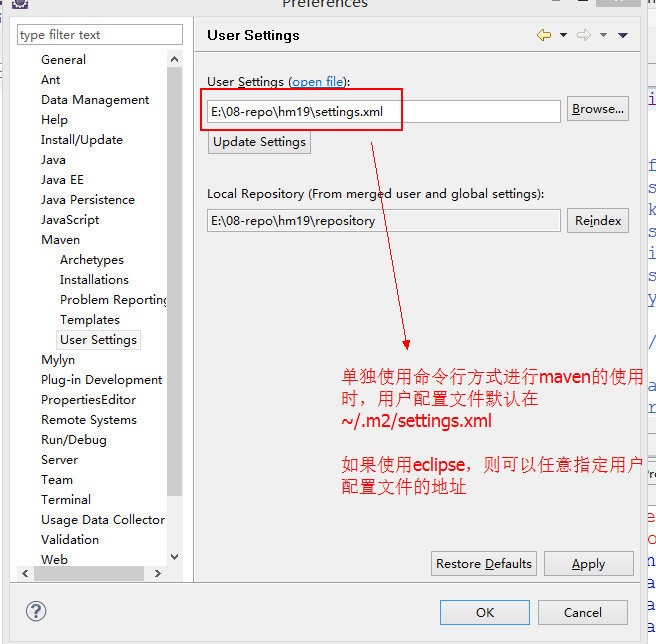
### 第三步：查看eclipse中是否安装成功



### 第四步：设置maven的安装路径



### 第五步：设置maven的用户配置

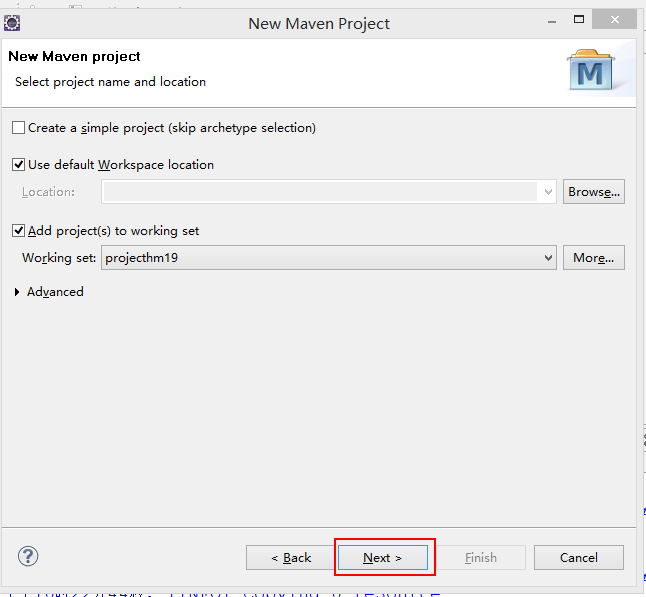


## 创建MavenFirst工程

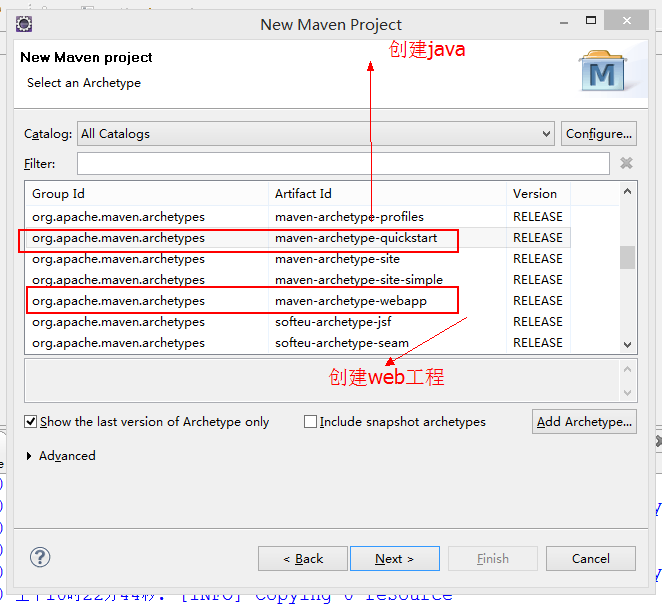
第一步：创建maven工程



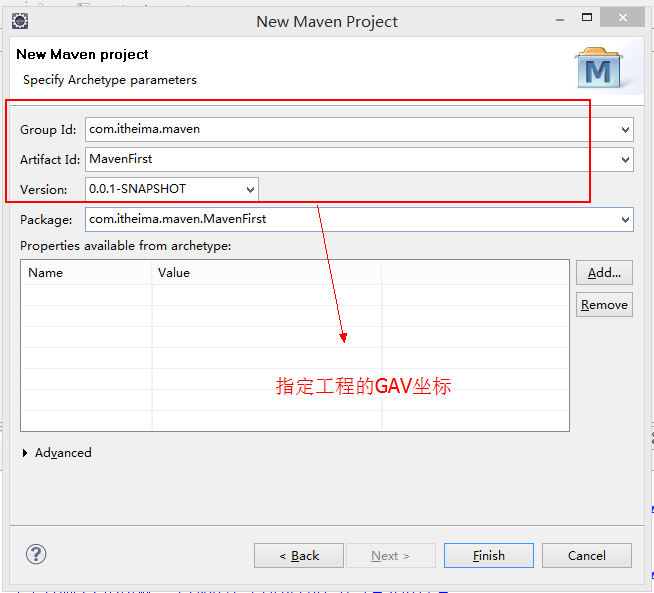
第二步：next



第三步：next

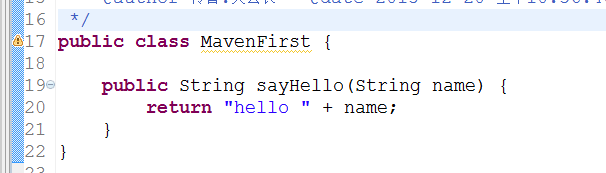


第四步：

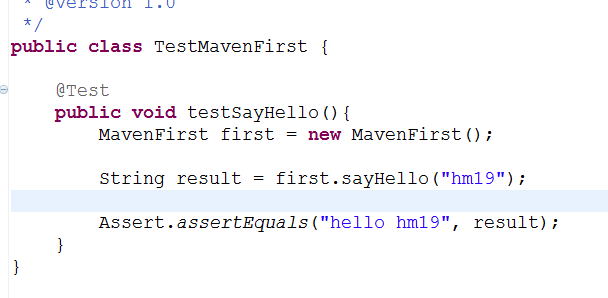


第五步：点击finish，创建maven工程

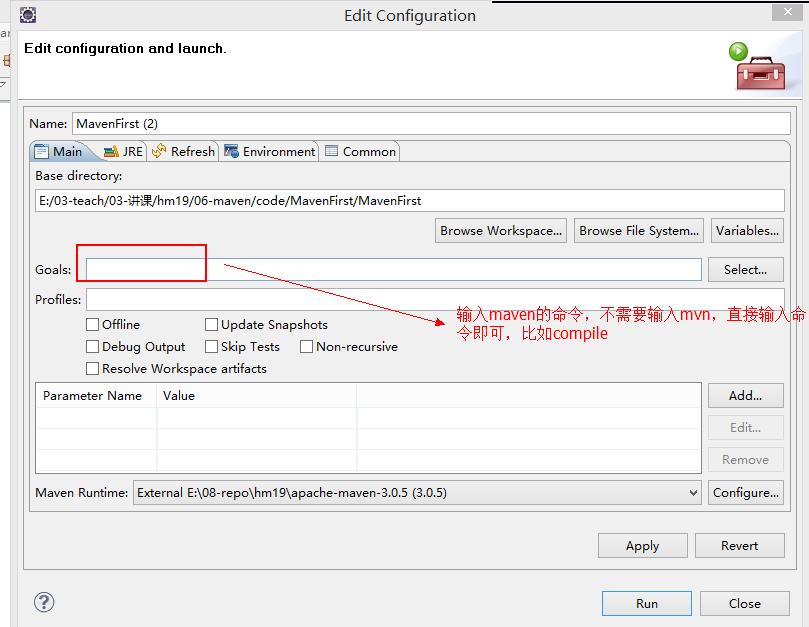
第六步：创建MavenFirst.java



第七步：创建TestMavenFirst.java



第七步：使用eclipse的选项执行maven命令

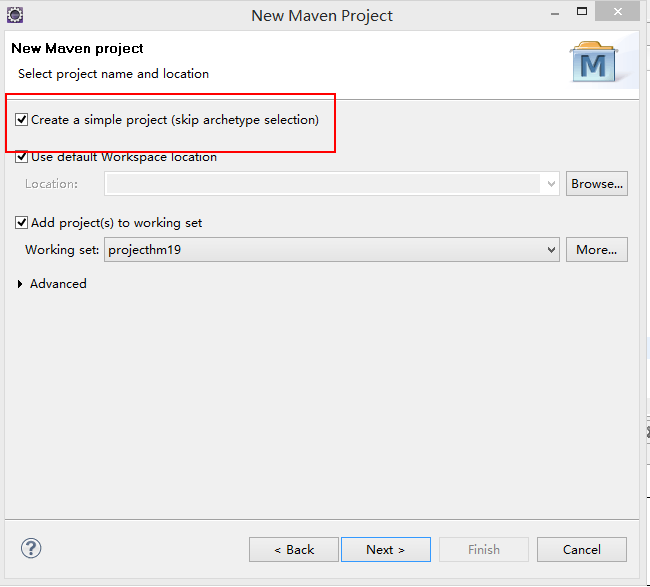


## 创建MavenSecond工程

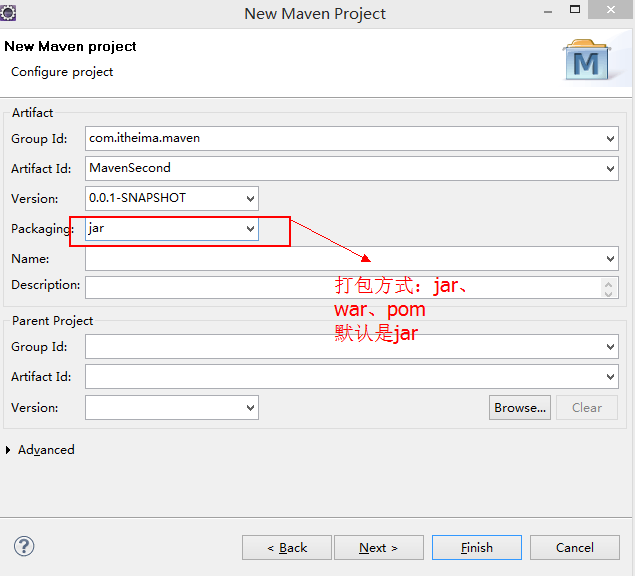
第一步：创建maven工程



第二步：next

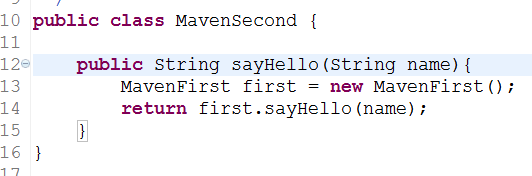


第三步：next

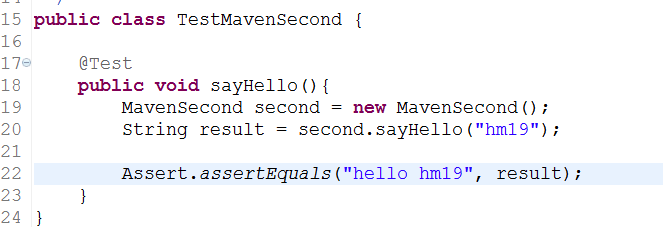


第四步：finish，创建maven工程

第五步：创建MavenSecond.java



第六步：创建TestMavenSecond.java



# Maven的核心概念

## 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。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（即jar包），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

### Maven坐标主要组成

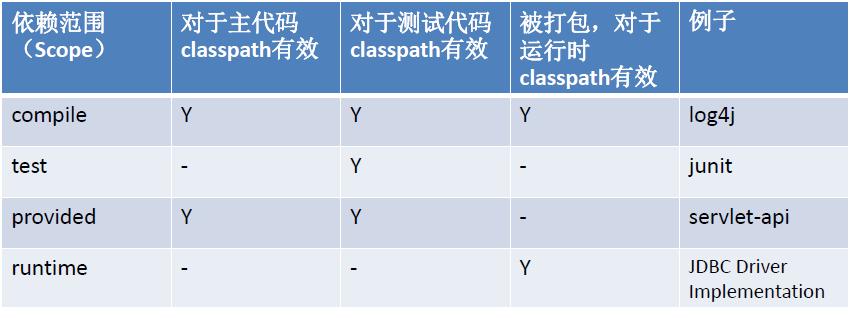
**groupId**：定义当前Maven组织名称

**artifactId**：定义实际项目名称

**version**：定义当前项目的当前版本

## 依赖管理

### 依赖范围



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

### 依赖传递

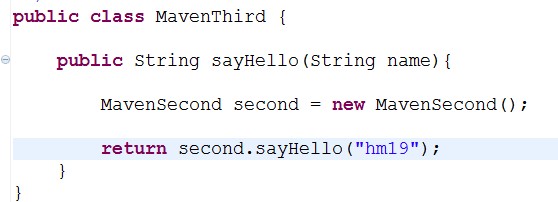
A、B、C

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

#### 创建MavenThird工程

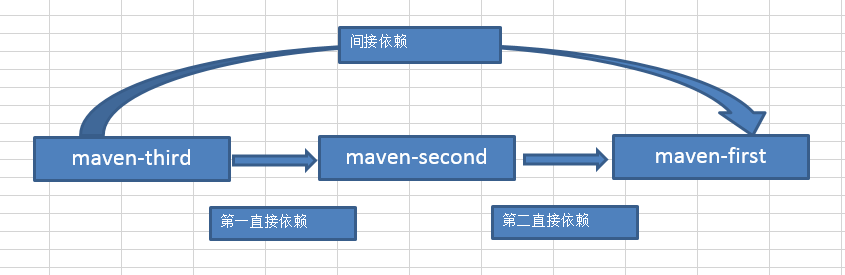
第一步：创建mavenThird工程

第二步：创建MavenThird.java

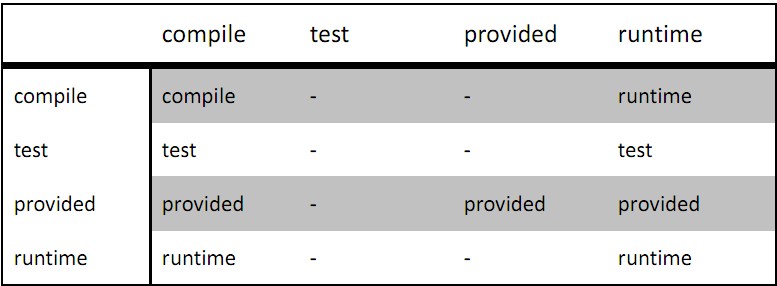


第三步：创建TestMavenThird.java

#### 分析第一解决依赖和第二直接依赖



#### 依赖范围传递

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；

### 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；

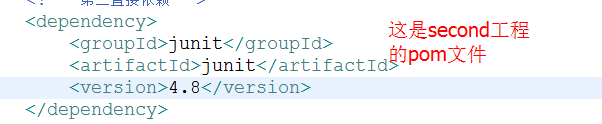
一种是同一个pom文件中的冲突。

#### 跨pom文件的冲突

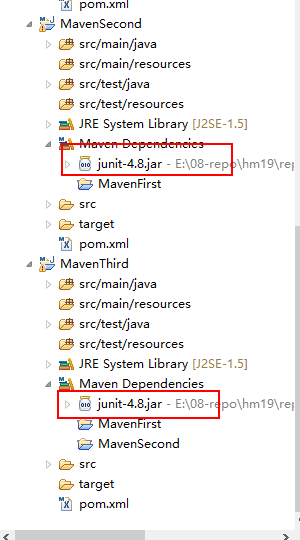
MavenFirst的pom文件中依赖来junit的4.9版本，那边MavenSecond和MavenThird中都是使用了4.9版本。



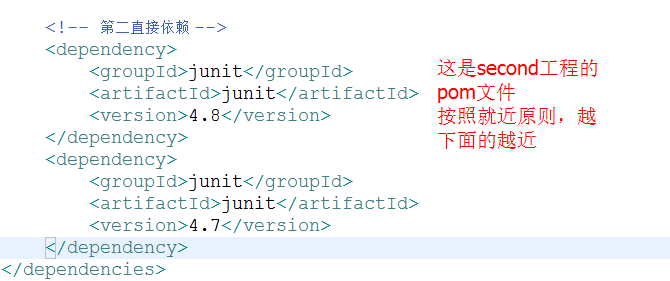
如果MavenSecond中重新依赖junit的4.8版本，那么MavenSecond和MavenThird中都是使用了4.8本，这体现了依赖的就近使用原则。



依赖的jar包如下：

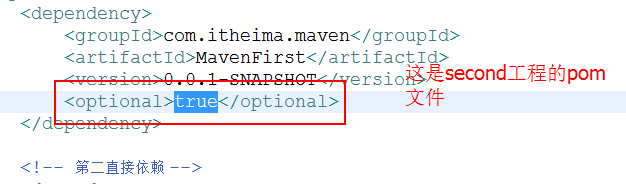


#### 同一个pom文件的冲突



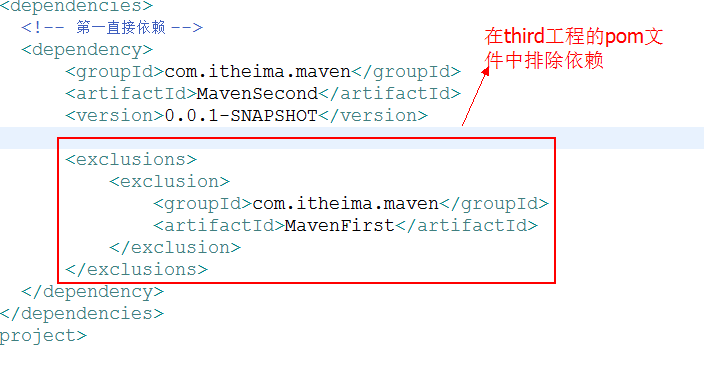
### 可选依赖

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。



### 排除依赖

Exclusions标签可以排除依赖



## 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。

### 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。

### 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，只要在同一个生命周期，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。

### 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

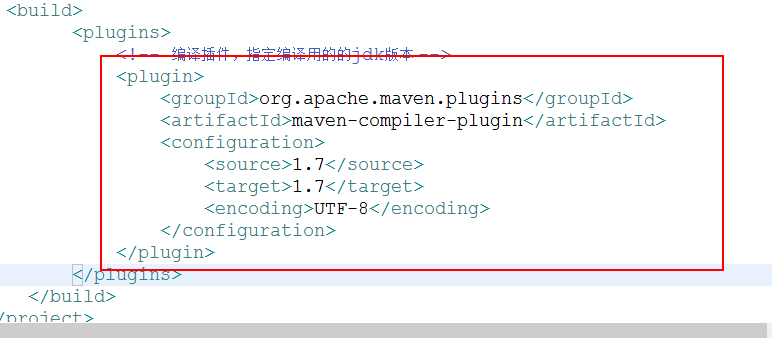
site-deploy 将生成的站点文档部署到特定的服务器上

## 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，功能是由插件来实现的。

比如：compile就是一个插件实现的功能。

### 编译插件



### Tomcat插件

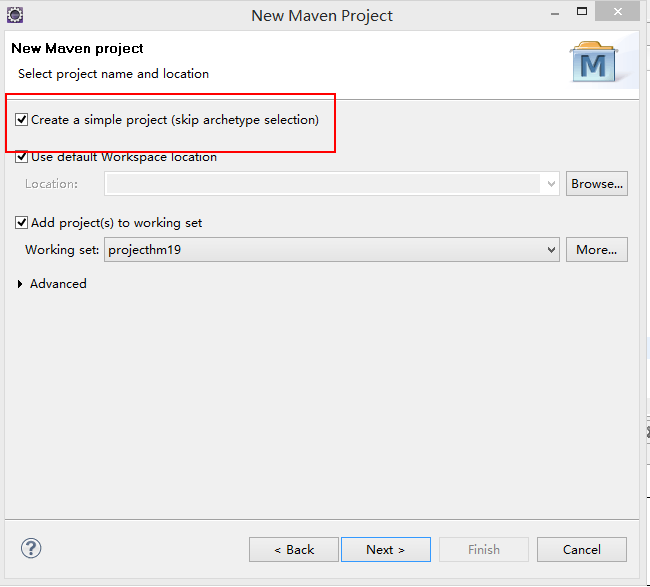
如果使用maven的tomcat插件的话，那么本地则不需要安装tomcat。

#### 创建maven的web工程

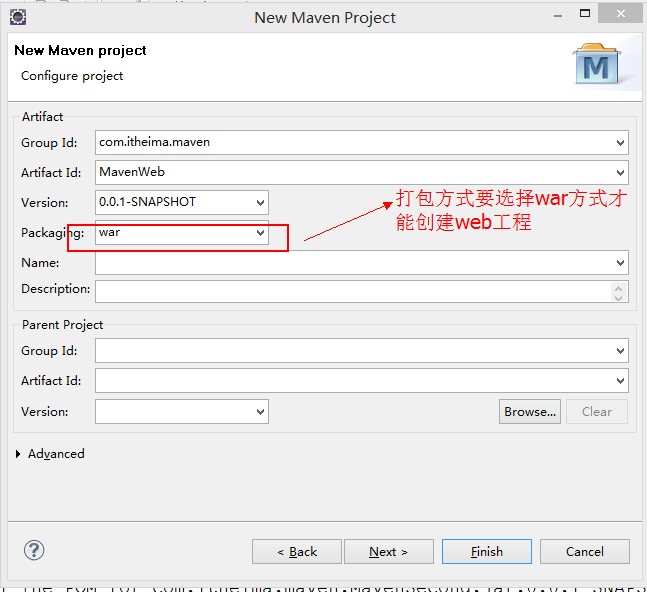
第一步：创建maven工程



第二步：next



第三步：next

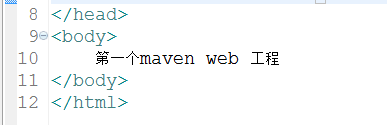


第四步：点击finish创建maven工程

第五步：创建WEB-INF及web.xml文件



第六步：创建index.jsp文件



#### 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

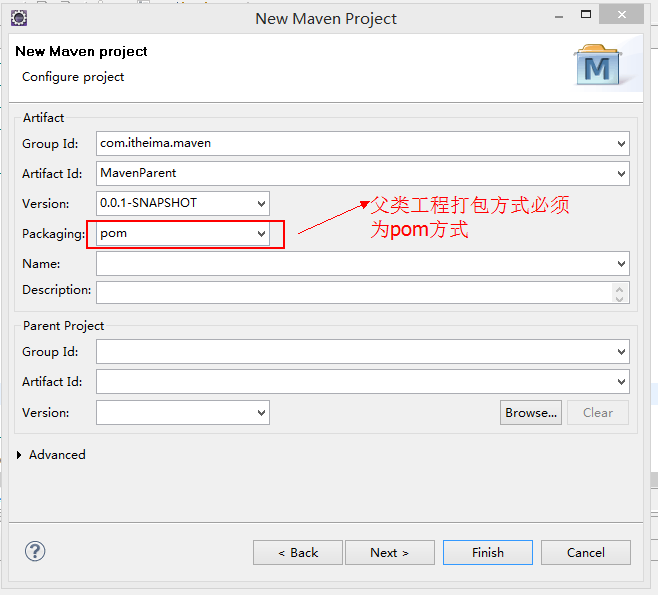


使用tomcat7来运行web工程，它的命令是：**tomcat7:run**

## 继承

在maven中的继承，指的是pom文件的继承

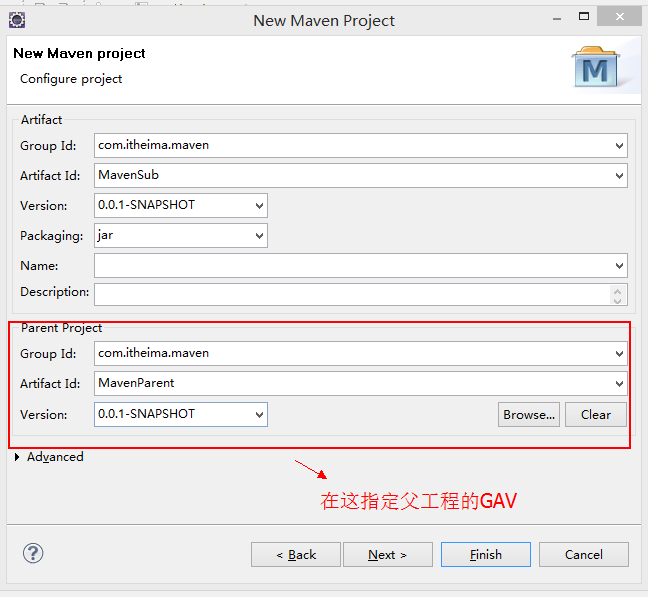
### 创建父工程



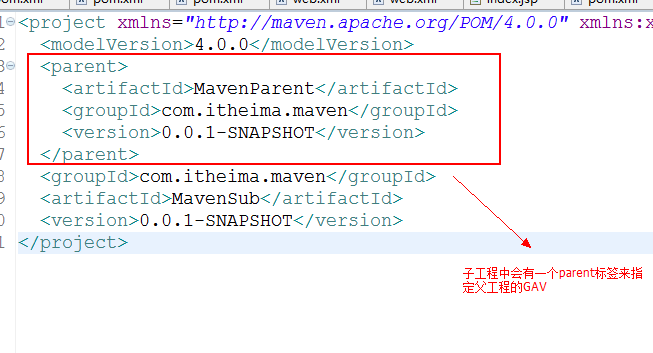
### 创建子工程

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：

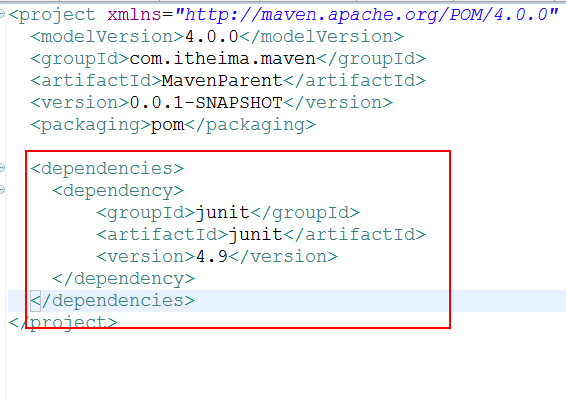


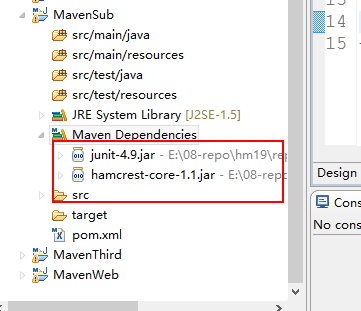
子工程的pom文件



### 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。

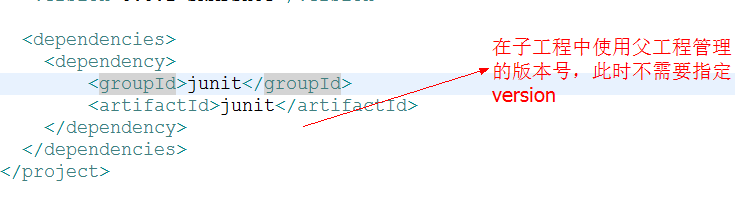




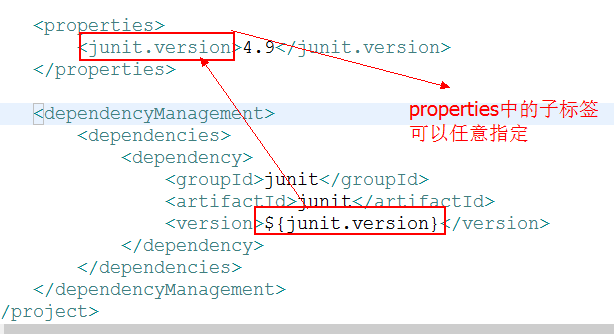
### 父工程统一管理版本号

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：



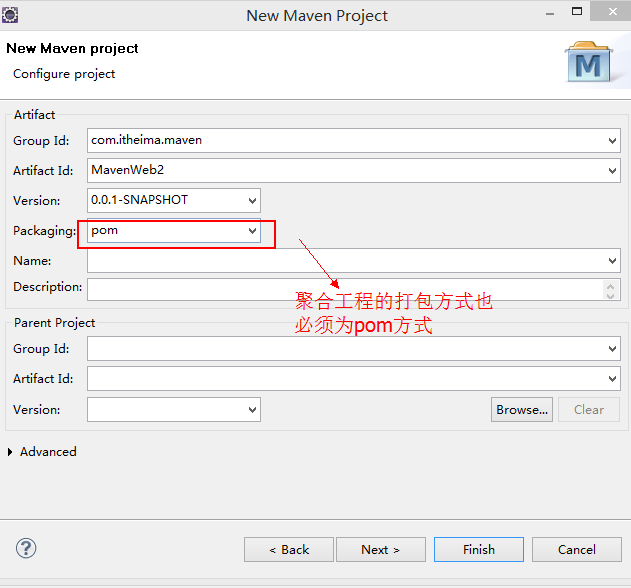
### 父工程抽取版本号



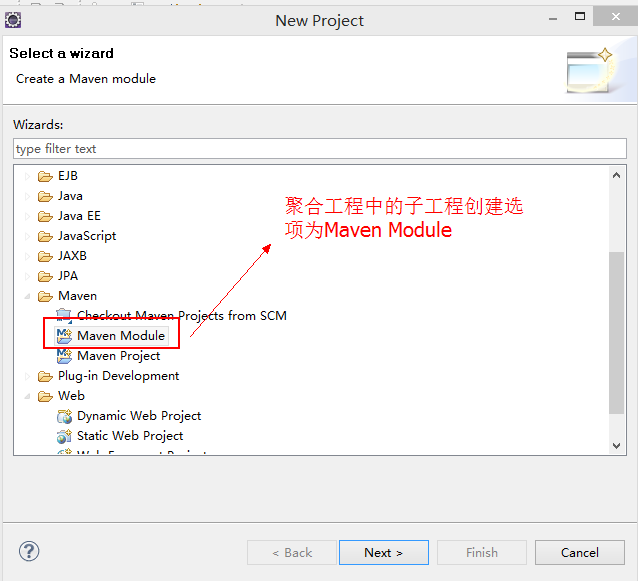
## 聚合

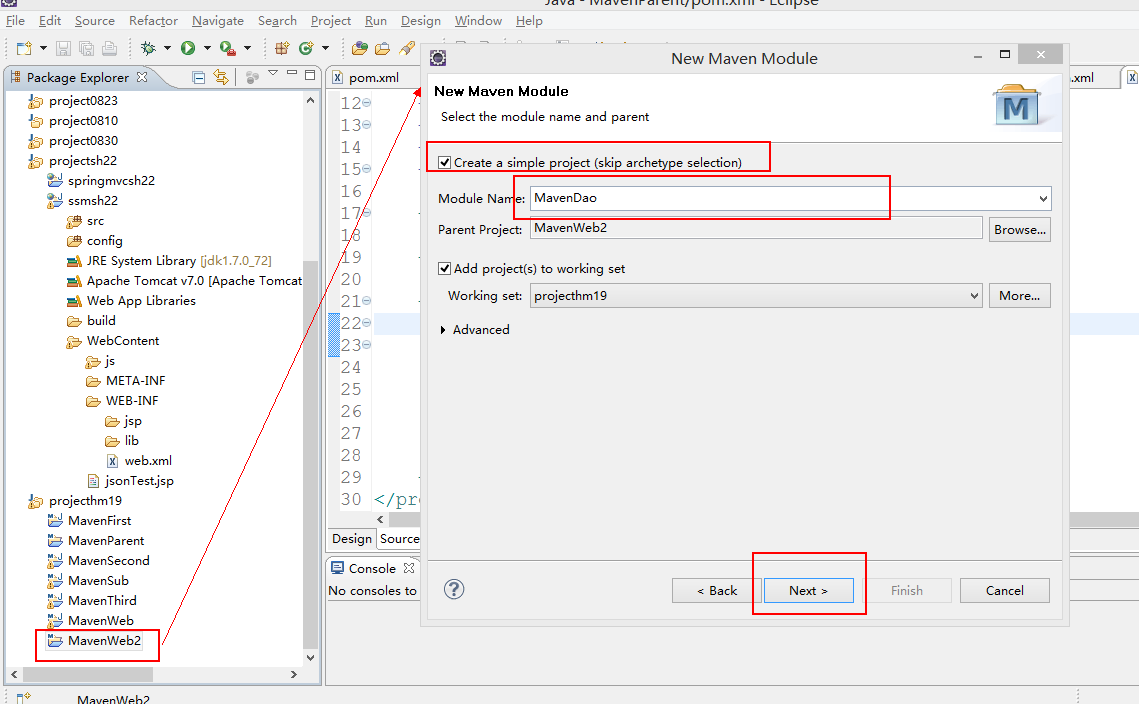
在真实项目中，一个项目有表现层、业务层、持久层，对于业务层和持久层，它们可以在多个工程中被使用，所以一般会将业务层和持久单独创建为java工程，为其他工程依赖。

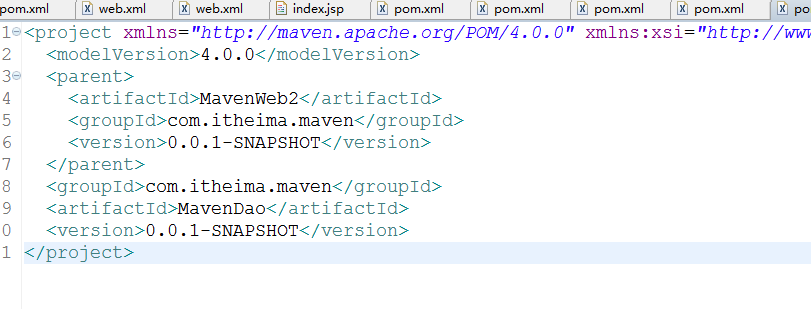
### 创建一个聚合工程



### 创建持久层



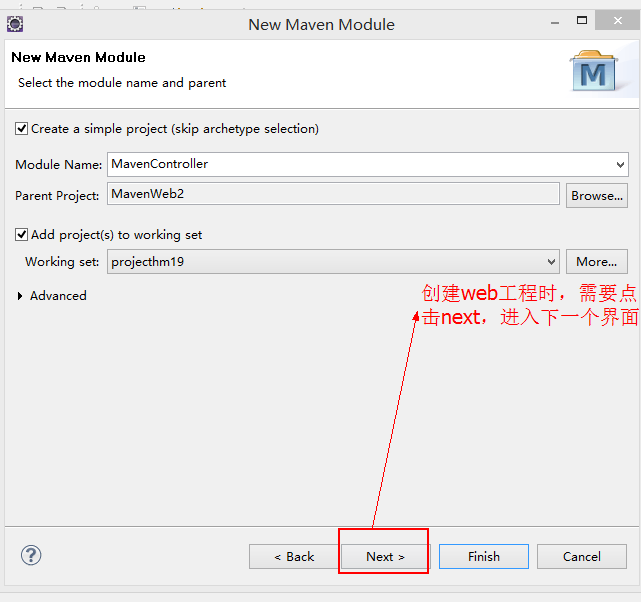


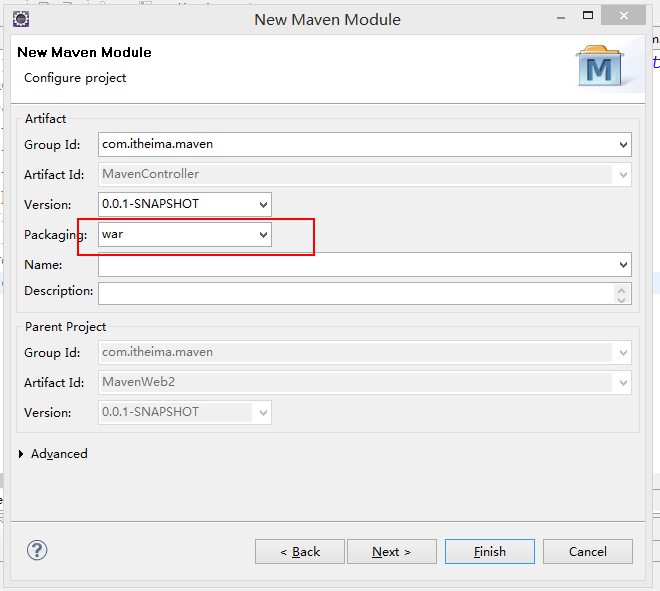


### 创建业务层

和持久层的创建一样

### 表现层





### 聚合为一个工程来运行

聚合工程的pom文件：



# Maven仓库管理

## 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

## 仓库的分类

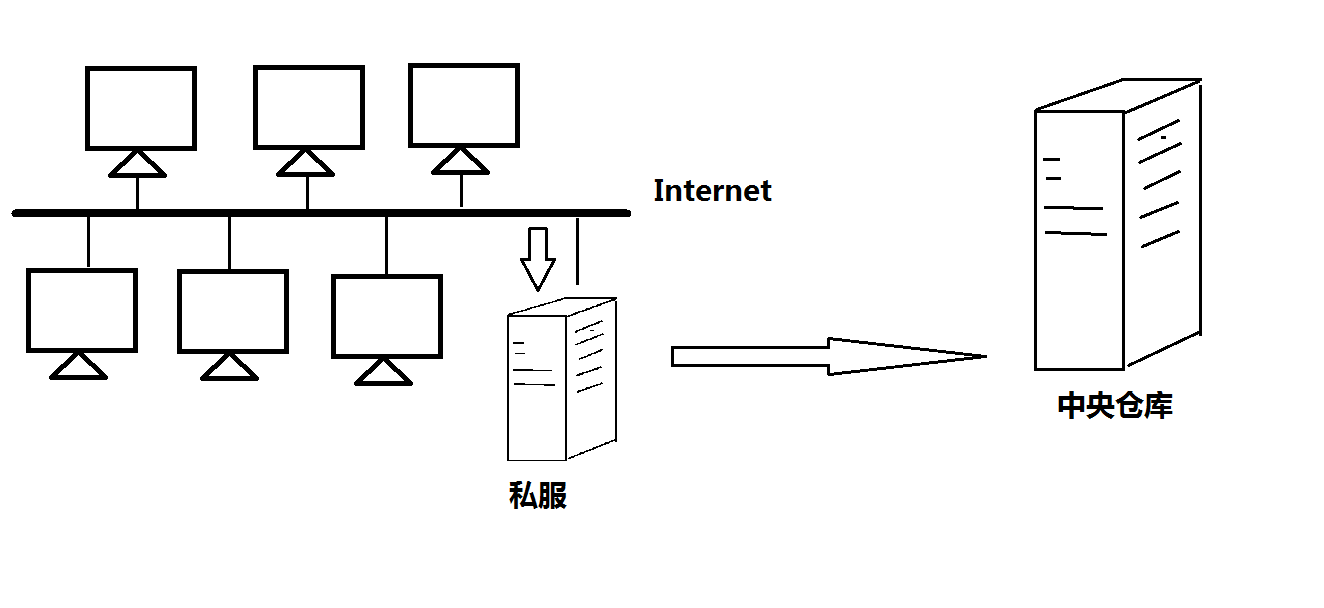
* 本地仓库

默认在user.name/.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



## Maven私服

### 安装Nexus

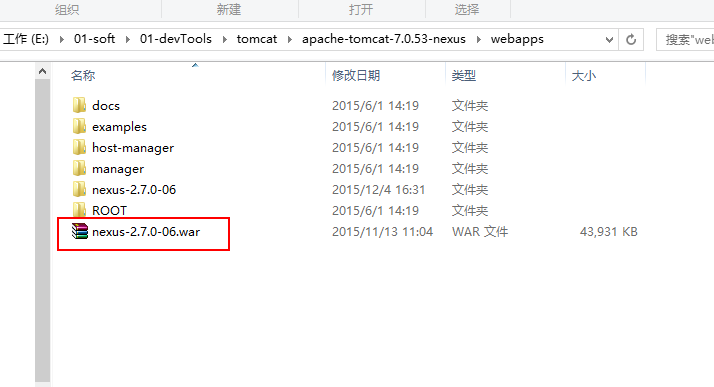
为所有来自中央仓库的构建安装提供本地缓存。

下载网站：<http://nexus.sonatype.org/>

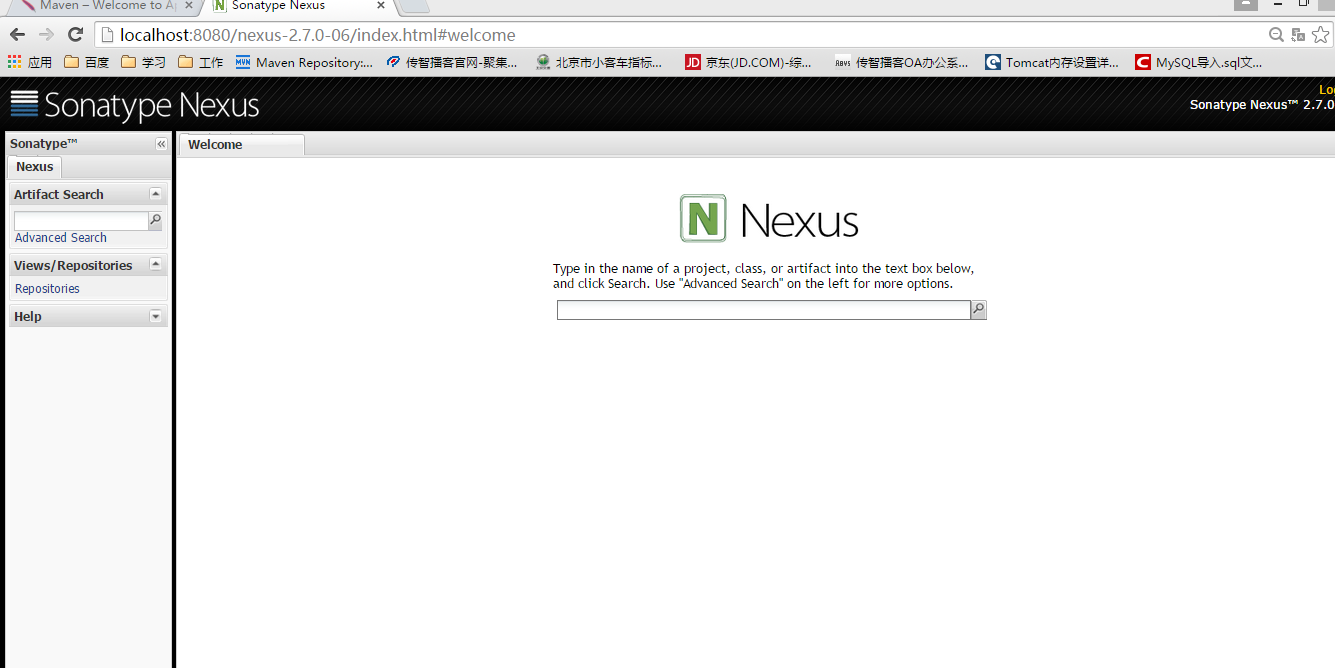
安装版本：nexus-2.7.0-06.war

第一步：安装tomcat

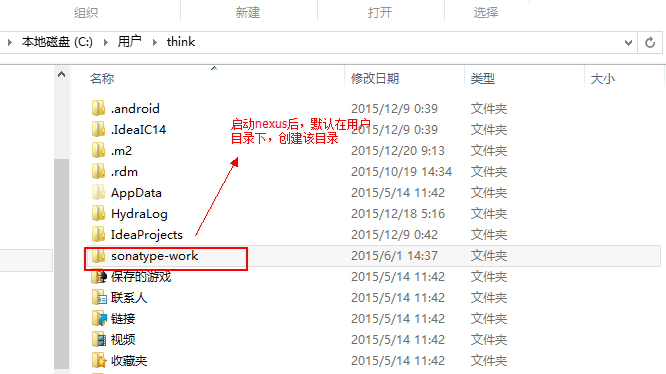
第二步：将nexus的war包拷贝到tomcat的webapps下

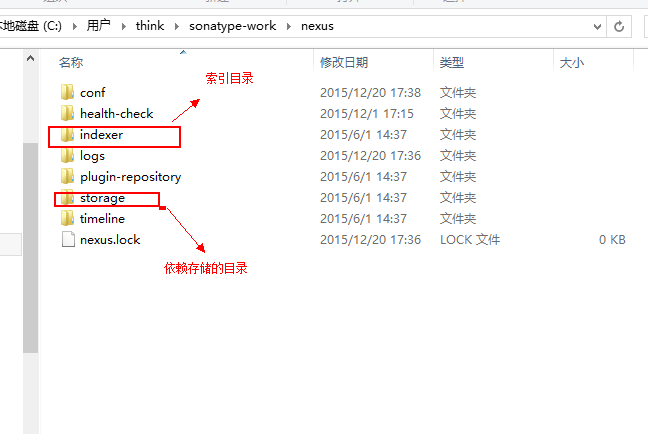


第三步：启动tomcat



第四步：nexus的本地目录





### 访问Nexus

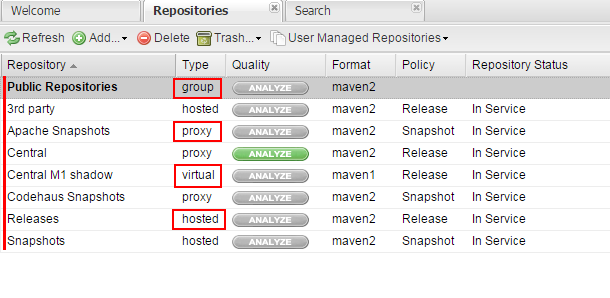
访问URL: <http://localhost:8080/nexus-2.7.0-06/>

默认账号:

用户名： admin

密码： admin123

### Nexus的仓库和仓库组

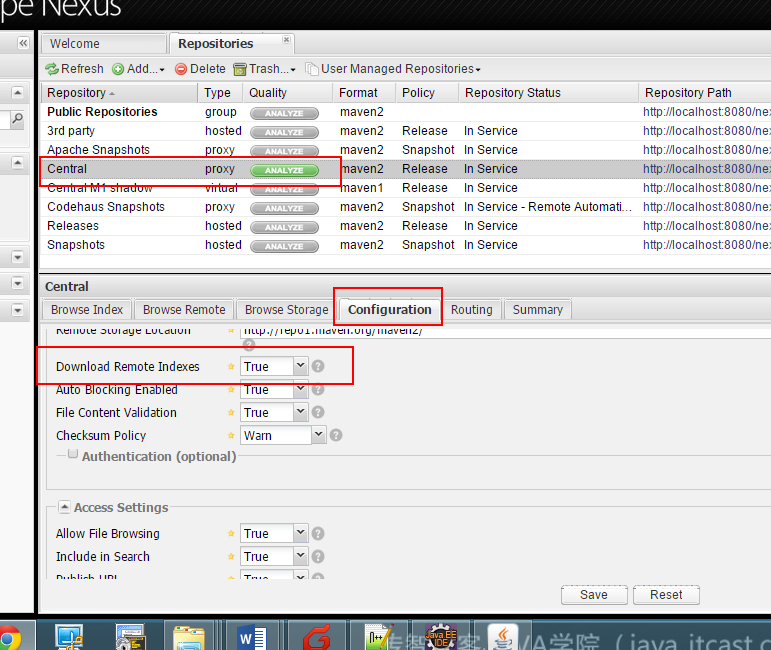


**仓库有4种类型 :**

* 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**



### 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> |



### 部署构建到Nexus

#### 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> |

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

#### 第三步：执行maven的deploy命令

