# Maven的安装

       安装Maven之前要确保已经安装好了jdk，并且配置好了环境变量JAVA\_HOME。具体安装步骤如下：

1.     从apache网上下载maven项目的压缩包。下载地址为：http://maven.apache.org/download.html。比如现在最新的Maven版本是3.0.4，那么我下载好的安装文件就是apache-maven-3.0.4.zip。

2.     将下载后的压缩包解压到Maven的安装目录，比如说是D:\\develop，那么解压后就是D:\\develop\\apache-maven-3.0.4。

3.     添加环境变量M2\_HOME，其值为Maven的家目录，如D:\\develop\\apache-maven-3.0.4。

4.     添加环境变量M2，其值将为maven安装目录的bin目录，即D:\\develop\\apache-maven-3.0.4\\bin，对于windows系统也可以使用%M2\_HOME%\\bin，对于linux系统也可以使用$M2\_HOME/bin。

5.     将环境变量M2加入到PATH变量中，对于windows系统，可以在PATH变量的值后面加上“;%M2%”，对于linux系统，可以使用“export path=$path:$M2”

6.     还有一个可选的环境变量MAVEN\_OPTS，该环境变量主要是配置Maven在使用jdk的时候指定JVM属性的。如指定其值为“-Xms256m -Xmx512m”。

经过以上几步之后Maven就安装成功了。接下来我们可以在命令窗口使用mvn --version来验证一下Maven是否安装成功。如能正确输出Maven的安装版本，则表示它安装成功了。

# 配置Maven的环境信息

在Maven中提供了一个settings.xml文件来定义Maven的全局环境信息。这个文件会存在于Maven的安装目录的conf子目录下面，或者是用户家目录的.m2子目录下面。我们可以通过这个文件来定义本地仓库、远程仓库和联网使用的代理信息等。

其实相对于多用户的PC机而言，在Maven安装目录的conf子目录下面的settings.xml才是真正的全局的配置。而用户家目录的.m2子目录下面的settings.xml的配置只是针对当前用户的。当这两个文件同时存在的时候，那么对于相同的配置信息用户家目录下面的settings.xml中定义的会覆盖Maven安装目录下面的settings.xml中的定义。用户家目录下的settings.xml文件一般是不存在的，但是Maven允许我们在这里定义我们自己的settings.xml，如果需要在这里定义我们自己的settings.xml的时候就可以把Maven安装目录下面的settings.xml文件拷贝到用户家目录的.m2目录下，然后改成自己想要的样子。

先来看一个基本的settings.xml的样子：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<settings** xmlns="http://maven.apache.org/SETTINGS/1.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd"**>**
6. **<localRepository>**D:\\develop\\mavenRepository**</localRepository>**
7. **<interactiveMode>**true**</interactiveMode>**
8. **<offline>**false**</offline>**
9. **<pluginGroups>**
11. **</pluginGroups>**
13. **<proxies>**
14. **<proxy>**
15. **<id>**optional**</id>**
16. **<active>**true**</active>**
17. **<protocol>**http**</protocol>**
18. **<username>**proxyuser**</username>**
19. **<password>**proxypass**</password>**
20. **<host>**proxy.host.net**</host>**
21. **<port>**80**</port>**
22. **<nonProxyHosts>**local.net|some.host.com**</nonProxyHosts>**
23. **</proxy>**
24. **</proxies>**
26. **<servers>**
27. **<server>**
28. **<id>**deploymentRepo**</id>**
29. **<username>**repouser**</username>**
30. **<password>**repopwd**</password>**
31. **</server>**
32. **</servers>**
34. **<mirrors>**
35. **<mirror>**
36. **<id>**mirrorId**</id>**
37. **<mirrorOf>**repositoryId**</mirrorOf>**
38. **<name>**Human Readable Name for this Mirror.**</name>**
39. **<url>**http://my.repository.com/repo/path**</url>**
40. **</mirror>**
41. **</mirrors>**
43. **<profiles>**
44. **<profile>**
45. **<id>**jdk-1.5**</id>**
46. **<activation>**
47. **<jdk>**1.5**</jdk>**
48. **</activation>**
49. **<repositories>**
50. **<repository>**
51. **<id>**jdk15**</id>**
52. **<name>**jdk1.5**</name>**
53. **<url>**http://www.myhost.com/maven/jdk15**</url>**
54. **<layout>**default**</layout>**
55. **<snapshotPolicy>**always**</snapshotPolicy>**
56. **</repository>**
57. **</repositories>**
58. **</profile>**
59. **</profiles>**
60. **<activeProfiles>**
61. **<activeProfile>**jdk-1.5**</activeProfile>**
62. **</activeProfiles>**
63. **</settings>**

settings.xml中主要包括以下元素：

**localRepository**：表示Maven用来在本地储存信息的本地仓库的目录。默认是用户家目录下面的.m2/repository目录。

**interactiveMode**：表示是否使用交互模式，默认是true；如果设为false，那么当Maven需要用户进行输入的时候，它会使用一个默认值。

**offline**：表示是否离线，默认是false。这个属性表示在Maven进行项目编译和部署等操作时是否允许Maven进行联网来下载所需要的信息。

**pluginGroups**：在pluginGroups元素下面可以定义一系列的pluginGroup元素。表示当通过plugin的前缀来解析plugin的时候到哪里寻找。pluginGroup元素指定的是plugin的groupId。默认情况下，Maven会自动把org.apache.maven.plugins和org.codehaus.mojo添加到pluginGroups下。

**proxies**：其下面可以定义一系列的proxy子元素，表示Maven在进行联网时需要使用到的代理。当设置了多个代理的时候第一个标记active为true的代理将会被使用。下面是一个使用代理的例子：

**Xml代码  [收藏代码](javascript:void())**

1. **<proxies>**
2. **<proxy>**
3. **<id>**xxx**</id>**
4. **<active>**true**</active>**
5. **<protocol>**http**</protocol>**
6. **<username>**用户名**</username>**
7. **<password>**密码**</password>**
8. **<host>**代理服务器地址**</host>**
9. **<port>**代理服务器的端口**</port>**
10. **<nonProxyHosts>**不使用代理的主机**</nonProxyHosts>**
11. **</proxy>**
12. **</proxies>**

**servers**：其下面可以定义一系列的server子元素，表示当需要连接到一个远程服务器的时候需要使用到的验证方式。这主要有username/password和privateKey/passphrase这两种方式。以下是一个使用servers的示例：

**Xml代码  [收藏代码](javascript:void())**

1. **<servers>**
2. **<server>**
3. **<id>**id**</id>**
4. **<username>**用户名**</username>**
5. **<password>**密码**</password>**
6. **</server>**
7. **</servers>**

**mirrors**：用于定义一系列的远程仓库的镜像。我们可以在pom中定义一个下载工件的时候所使用的远程仓库。但是有时候这个远程仓库会比较忙，所以这个时候人们就想着给它创建镜像以缓解远程仓库的压力，也就是说会把对远程仓库的请求转换到对其镜像地址的请求。每个远程仓库都会有一个id，这样我们就可以创建自己的mirror来关联到该仓库，那么以后需要从远程仓库下载工件的时候Maven就可以从我们定义好的mirror站点来下载，这可以很好的缓解我们远程仓库的压力。在我们定义的mirror中每个远程仓库都只能有一个mirror与它关联，也就是说你不能同时配置多个mirror的mirrorOf指向同一个repositoryId。

看以下是一个使用mirrors的例子：

**Xml代码  [收藏代码](javascript:void())**

1. **<mirrors>**
2. **<mirror>**
3. **<id>**mirrorId**</id>**
4. **<mirrorOf>**repositoryId**</mirrorOf>**
5. **<name>**定义一个容易看懂的名称 **</name>**
6. **<url>**http://my.repository.com/repo/path**</url>**
7. **</mirror>**
8. **</mirrors>**

  id：是用来区别mirror的，所有的mirror不能有相同的id

  mirrorOf：用来表示该mirror是关联的哪一个仓库，其值为其关联仓库的id。当要同时关联多个仓库时，这多个仓库之间可以用逗号隔开；当要关联所有的仓库时，可以使用“\*”表示；当要关联除某一个仓库以外的其他所有仓库时，可以表示为“\*,!repositoryId”；当要关联不是localhost或用file请求的仓库时，可以表示为“external:\*”。

  url：表示该镜像的url。当Maven在建立系统的时候就会使用这个url来连接到我们的远程仓库。

**profiles**：用于指定一系列的profile。profile元素由activation、repositories、pluginRepositories和properties四个元素组成。当一个profile在settings.xml中是处于活动状态并且在pom.xml中定义了一个相同id的profile时，settings.xml中的profile会覆盖pom.xml中的profile。

（1）activation：这是profile中最重要的元素。跟pom.xml中的profile一样，settings.xml中的profile也可以在特定环境下改变一些值，而这些环境是通过activation元素来指定的。

       看下面一个例子：

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**test**</id>**
4. **<activation>**
5. **<activeByDefault>**false**</activeByDefault>**
6. **<jdk>**1.6**</jdk>**
7. **<os>**
8. **<name>**Windows 7**</name>**
9. **<family>**Windows**</family>**
10. **<arch>**x86**</arch>**
11. **<version>**5.1.2600**</version>**
12. **</os>**
13. **<property>**
14. **<name>**mavenVersion**</name>**
15. **<value>**2.0.3**</value>**
16. **</property>**
17. **<file>**
18. **<exists>**${basedir}/file2.properties**</exists>**
19. **<missing>**${basedir}/file1.properties**</missing>**
20. **</file>**
21. **</activation>**
22. ...
23. **</profile>**
24. **</profiles>**

       在上面这段代码中，当所有的约束条件都满足的时候就会激活这个profile。

  jdk：表示当jdk的版本满足条件的时候激活，在这里是1.6。这里的版本还可以用一个范围来表示，如

<jdk>[1.4,1.7)</jdk>表示1.4、1.5和1.6满足；

<jdk>[1.4,1.7]</jdk>表示1.4、1.5、1.6和1.7满足；

  os：表示当操作系统满足条件的时候激活。

  property：property是键值对的形式，表示当Maven检测到了这样一个键值对的时候就激活该profile。

(1)下面的示例表示当存在属性hello的时候激活该profile。

**Xml代码  [收藏代码](javascript:void())**

1. **<property>**
2. **<name>**hello**</name>**
3. **</property>**

(2)下面的示例表示当属性hello的值为world的时候激活该profile。

**Xml代码  [收藏代码](javascript:void())**

1. **<property>**
2. **<name>**hello**</name>**
3. **<value>**world**</value>**
4. **</property>**

这个时候如果要激活该profile的话，可以在调用Maven指令的时候加上参数hello并指定其值为world，如：

mvn compile –Dhello=world

  file：表示当文件存在或不存在的时候激活，exists表示存在，missing表示不存在。如下面的例子表示当文件hello/world不存在的时候激活该profile。

**Xml代码  [收藏代码](javascript:void())**

1. **<profile>**
2. **<activation>**
3. **<file>**
4. **<missing>**hello/world**</missing>**
5. **</file>**
6. **</activation>**
7. **</profile>**

  activeByDefault：当其值为true的时候表示如果没有其他的profile处于激活状态的时候，该profile将自动被激活。

（2）properties：用于定义属性键值对的。当该profile是激活状态的时候，properties下面指定的属性都可以在pom.xml中使用。

（3）repositories：用于定义远程仓库的，当该profile是激活状态的时候，这里面定义的远程仓库将作为当前pom的远程仓库。

**Xml代码  [收藏代码](javascript:void())**

1. **<repositories>**
2. **<repository>**
3. **<id>**codehausSnapshots**</id>**
4. **<name>**Codehaus Snapshots**</name>**
5. **<releases>**
6. **<enabled>**false**</enabled>**
7. **<updatePolicy>**always**</updatePolicy>**
8. **<checksumPolicy>**warn**</checksumPolicy>**
9. **</releases>**
10. **<snapshots>**
11. **<enabled>**true**</enabled>**
12. **<updatePolicy>**never**</updatePolicy>**
13. **<checksumPolicy>**fail**</checksumPolicy>**
14. **</snapshots>**
15. **<url>**http://snapshots.maven.codehaus.org/maven2**</url>**
16. **<layout>**default**</layout>**
17. **</repository>**
18. **</repositories>**

  releases、snapshots：这是对于工件的类型的限制。

  enabled：表示这个仓库是否允许这种类型的工件

  updatePolicy：表示多久尝试更新一次。可选值有always、daily、interval:minutes（表示每多久更新一次）和never。

  checksumPolicy：当Maven在部署项目到仓库的时候会连同校验文件一起提交，checksumPolicy表示当这个校验文件缺失或不正确的时候该如何处理，可选项有ignore、fail和warn。

（4）pluginRepositories：在Maven中有两种类型的仓库，一种是存储工件的仓库，另一种就是存储plugin插件的仓库。pluginRepositories的定义和repositories的定义类似，它表示Maven在哪些地方可以找到所需要的插件。

**activeProfiles：**底包含一系列的activeProfile元素，表示对于所有的pom都处于活跃状态的profile。如：

**Xml代码  [收藏代码](javascript:void())**

1. **<activeProfiles>**
2. **<activeProfile>**alwaysActiveProfile**</activeProfile>**
3. **<activeProfile>**anotherAlwaysActiveProfile**</activeProfile>**
4. **</activeProfiles>**

# 使用Maven

## 使用maven建立自己的项目

   (1)切换到项目所在的目录，如：d:\\develop\\apps

   (2)执行如下指令：

      mvn archetype:generate 这样就可以根据提示来建立一个maven项目

      后面可以接很多参数，常用的有：

        -DgroupId=com.company.app 组id

        -DartifactId=app 项目名称，maven会根据这个名称在当前目录下新建一个名为该名称的目录用于建立项目

        -DinteractiveMode=false  是否已交互模式进行，如果是false的话就会采用默认设置建立项目

      以下是一个示例：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn archetype:generate -DgroupId=com.tiantian -DartifactId=jxc -DinteractiveMode=false

        运行上面的代码就会在d:\\develop\\apps下面新建一个名为jxc的maven项目

   上面建立的只是一个简单的java应用，那如果我要建立一个web应用该如何呢？以下是一个示例：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DgroupId=com.myhost.myapp -DartifactId=myapp

   其实使用Maven建立简单项目时是省略了参数archetypeArtifactId，当没有指定该参数时默认将使用maven-archetype-quickstart。所以要把上面建立一个普通java应用的Maven项目的指令写完整的话就应该是:

**Cmd代码  [收藏代码](javascript:void())**

1. Mvn archetype:generate –DarchetypeGroupId=org.apache.maven.archetypes –DarchetypeArtifactId=maven-archetype-quickstart –DgroupId=com.tiantian –DartifactId=jxc –DinteractiveMode=false

   从上面我们可以看出要建立什么样的项目就需要使用什么样的archetype，archetype就是Maven定义好的项目类型的模板，一个对应项目的目录架构。archetype是通过参数archetypeArtifactId来指定的。除了上面介绍的两种archetype之外，Maven还为我们提供了以下几种archetype。

|  |  |
| --- | --- |
| **archetypeArtifactId** | **描述** |
| **maven-archetype-archetype** | 包含一个archetype的例子，主要用于当我们要建立自己的archetype的时候 |
| **maven-archetype-j2ee-simple** | 包含一个简单的j2ee应用的例子 |
| **maven-archetype-plugin** | 包含一个Maven plugin的例子 |
| **maven-archetype-plugin-site** | 包含一个Maven plugin site的例子 |
| **Maven-archetype-portlet** | 包含一个portlet的例子 |
| **Maven-archetype-simple** | 包含一个简单maven项目 |
| **Maven-archetype-site** | 包含一个maven site的例子，它能够展示一些支持的文档类型，包括APT、XDoc和FML |
| **Maven-archetype-site-simple** | 包含一个maven site的例子 |

       在后续的内容中将介绍关于如何构建自己的archetype的内容。

## 使用mvc compile进行源码编译

   在目录d:\\develop\\apps下面建立了项目jxc之后，我们就可以进入目录d:\\develop\\apps\\jxc执行指令mvn compile进行编译了。编译后的文件将放在项目根目录下的target目录中。

## 使用mvc test-compile编译测试源码

   在目录d:\\develop\\apps下建立了项目jxc之后，我们就可以在cmd中切换目录到d:\\develop\\apps\\jxc，然后执行指令mvc test-compile进行测试源码编译。

## 使用mvc test编译源码和测试源码进行测试

   在目录d:\\develop\\apps\\jxc下面执行指令mvc test将先编译源码，再编译测试源码，然后执行测试代码。

## 使用mvn package进行项目打包

   在项目所在的目录执行指令mvn package可以进行项目打包，打包方式是在项目根目录下的pom.xml文件中的packaging元素定义的，如果定义为jar则打为jar包，如果是war则打为war包，如果是ear则为ear包。如果pom.xml中没有定义packaging元素，则会使用默认值jar。打包后的文件将会放在项目根目录下的target目录下。打包后的文件名将是在pom.xml文件中定义的artifactId-version的形式，比如在pom.xml中定义的artifactId是hello，定义的version是1.0，则打包后生成的文件的名称将是hello-1.0。

## 使用mvn install安装jar包到maven的本地仓库

   使用mvn install可以把mvn package打包好的jar安装到maven的本地仓库。本地仓库默认是在~/.m2/repository，可以在maven安装目录下的conf/settings文件中的localRepository标签中定义本地仓库的路径。

   当我们需要安装一个本地jar包到本地资源库的时候我们可以使用“mvn install:install-file”指令，使用该指令时有几个参数需要指定。file：表示本地jar包所在的路径；groupId：表示jar包被安装到本地仓库之后的groupId；artifactId：表示jar包被安装到本地仓库之后的artifactId；version：表示安装到本地仓库之后对应的版本；packging：表示当前组件被引用的类型，既然我们是安装jar包则这里对应的自然是jar了。示例，假如现在我们需要安装一个jar包“c:\jars\abc.jar”到我们的Maven本地仓库，那么我们可以在命令窗口执行以下命令：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn install:install-file -Dfile=c:\jars\abc.jar -DgroupId=com.tiantian -DartifactId=abc -Dversion=1.0 -Dpackaging=jar

    这样就会把我们的“c:\jars\abc.jar”以groupId为“com.tiantian”，artifactId为“abc”，version为“1.0”，packaging类型为jar安装到我们的本地仓库，之后我们就可以直接在我们本地的Maven项目中引用该依赖类型了。如：

**Xml代码  [收藏代码](javascript:void())**

1. **<dependency>**
2. **<groupId>**com.tiantian**</groupId>**
3. **<artifactId>**abc**</artifactId>**
4. **<version>**1.0**</version>**
5. **</dependency>**

## 使用mvn deploy可以安装当前项目到远程仓库

## 使用mvn clean可以清除存放临时文件的target目录

## 把资源文件一并打到jar包中

如果需要把一些资源文件也一并打包到jar包里面的话，需要在${basedir}/src/main下新建一个resources目录，然后把所有的资源文件都放到这个目录中，这样这些文件都会放到类路径下面。如果需要在测试代码中访问到对应的资源，那么相对的就需要在${basedir}/src/test下新建一个resources目录，然后把相应的资源文件放在这个目录下。

## 过滤资源文件

   有的时候有些资源文件中的值我们需要在编译的时候动态的指定，maven允许我们在建立文件的时候以${property\_name}的方式指定，这样当我们进行编译的时候就会自动的把property\_name对应的值替换${property\_name}。这个property可以是pom.xml中定义的值，也可以是settings.xml中定义的值，也可以是定义在外部属性文件中的值，还可以是系统属性。maven这种动态替换属性值的功能默认是关闭的，如果要打开的话需要在项目的pom.xml文件中指定filtering的值为true，默认是false。如下示例：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
5. **<groupId>**com.tiantian.mavenApp**</groupId>**
6. **<artifactId>**mavenApp**</artifactId>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **<packaging>**jar**</packaging>**
10. **<name>**mavenAppTest**</name>**
11. **<url>**http://maven.apache.org**</url>**
13. **<properties>**
14. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
15. **</properties>**
17. **<dependencies>**
18. **<dependency>**
19. **<groupId>**junit**</groupId>**
20. **<artifactId>**junit**</artifactId>**
21. **<version>**3.8.1**</version>**
22. **<scope>**test**</scope>**
23. **</dependency>**
24. **</dependencies>**
26. **<build>**
27. **<resources>**
28. **<resource>**
29. **<directory>**src/main/resources**</directory>**
30. **<filtering>**true**</filtering>**
31. **</resource>**
32. **</resources>**
33. **</build>**
34. **</project>**

   从上面的文件我们可以看出，filtering这个参数是针对于特定的资源文件目录的，而且我们还可以使用directory来指定资源文件的存放位置，默认是src/main/resources。这个build元素是有默认值的，因为我们需要改变filtering的值，所以需要重新定义build，然后覆盖里面的默认值。

### 使用pom.xml和settings.xml中的元素element作为属性

   我们可以使用pom.xml文件中的element对应的值来作为属性值。在maven中，使用pom来表示pom.xml中的根元素project，所以我们可以使用${pom.name}来表示project元素下面的name元素，使用${pom.version}来表示project元素下面的version元素。有些元素在pom.xml文件中是没有明确定义的，但是那不代表它们不存在，这是因为它们有一个默认值，像这种元素我们也可以直接拿来使用。同样的我们需要使用settings.xml中的元素的时候可以使用settings前缀来关联，如${settings.localRepository}就是在settings.xml中定义的localRepository元素的值。下面是一组示例，文件都是定义在resources目录下的：

   文件application.properties

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=${pom.name}
2. project\_version=${pom.version}
3. project\_modelVersion=${pom.modelVersion}
4. project\_artifactId=${pom.artifactId}
5. settings\_localRepository=${settings.localRepository}

   文件test.txt

**Txt代码  [收藏代码](javascript:void())**

1. project\_version=${pom.version}
2. project\_modelVersion=${pom.modelVersion}
3. project\_artifactId=${pom.artifactId}
4. settings\_localRepository=${settings.localRepository}

   在resources目录下定义了上面两个文件之后，在命令行模式下进入pom.xml文件所在的那一级目录，也就是项目的根目录，之后使用mvn compile进行编译，编译后可以到target/classes目录下找到对应的资源文件，打开可以看到如下内容：

   application.properties文件的内容：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=mavenAppTest
2. project\_version=1.0-SNAPSHOT
3. project\_modelVersion=4.0.0
4. project\_artifactId=mavenApp
5. settings\_localRepository=D:\\develop\\mavenRepository

   test.txt文件的内容：

**Txt代码  [收藏代码](javascript:void())**

1. project\_version=1.0-SNAPSHOT
2. project\_modelVersion=4.0.0
3. project\_artifactId=mavenApp
4. settings\_localRepository=D:\\develop\\mavenRepository

   如果我们只需要对资源文件进行处理的话也可以使用mvn process-resources指令，该指令是专门用于处理资源文件的，而mvn compile是在编译的时候处理了资源文件。

### 使用外部文件的属性关联

       要使用外部文件的属性来关联资源文件里面定义的属性值，那么我们就需要告诉maven应该去哪里找这个属性文件，这是通过在pom.xml中指定filter的值来指定的。

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
5. **<groupId>**com.tiantian.mavenApp**</groupId>**
6. **<artifactId>**mavenApp**</artifactId>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **<packaging>**jar**</packaging>**
10. **<name>**mavenAppTest**</name>**
11. **<url>**http://maven.apache.org**</url>**
13. **<properties>**
14. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
15. **</properties>**
17. **<dependencies>**
18. **<dependency>**
19. **<groupId>**junit**</groupId>**
20. **<artifactId>**junit**</artifactId>**
21. **<version>**3.8.1**</version>**
22. **<scope>**test**</scope>**
23. **</dependency>**
24. **</dependencies>**
26. **<build>**
27. **<filters>**
28. **<filter>**src/main/filters/testFilter.properties**</filter>**
29. **</filters>**
30. **<resources>**
31. **<resource>**
32. **<directory>**src/main/resources**</directory>**
33. **<filtering>**true**</filtering>**
34. **</resource>**
35. **</resources>**
36. **</build>**
37. **</project>**

   在上面代码中我们通过build下面的filters下面的filter元素指定了一个外部用于资源文件过滤的属性文件src/main/filters/testFilter.properties，接着我们在src/main/filters下面建立testFilter.properties文件，其内容如下：

**Properties代码  [收藏代码](javascript:void())**

1. myFilter.maven.test=test
2. myFilter.maven.username=andy

然后我们在resources目录下面定义一个application.properties文件，内容如下：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=${pom.name}
2. project\_version=${pom.version}
3. project\_modelVersion=${pom.modelVersion}
4. project\_artifactId=${pom.artifactId}
5. settings\_localRepository=${settings.localRepository}
6. filter\_username=${myFilter.maven.username}

   接下来我们执行mvn process-resources指令来处理资源文件，将在target/classes下面对应的application.properties文件中看到如下内容：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=mavenAppTest
2. project\_version=1.0-SNAPSHOT
3. project\_modelVersion=4.0.0
4. project\_artifactId=mavenApp
5. settings\_localRepository=D:\\develop\\mavenRepository
6. filter\_username=andy

   我们可以看到我们在application.properties文件中定义的${myFilter.maven.username}已经被指定的外部属性文件testFilter.properties的属性myFilter.maven.username对应的属性值andy所取代。像这种指定外部属性文件来过滤资源文件的方式也可以在pom.xml中的properties元素下指定对应的子元素作为属性名称的形式来达到相同的效果。在上面示例的基础上，我们先在pom.xml文件的properties元素下面加一个myFilter.maven.username元素，然后指定其值为super，这时候pom.xml文件会是这个样子：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
5. **<groupId>**com.tiantian.mavenApp**</groupId>**
6. **<artifactId>**mavenApp**</artifactId>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **<packaging>**jar**</packaging>**
10. **<name>**mavenAppTest**</name>**
11. **<url>**http://maven.apache.org**</url>**
13. **<properties>**
14. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
15. **<myFilter.maven.username>**super**</myFilter.maven.username>**
16. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**junit**</groupId>**
21. **<artifactId>**junit**</artifactId>**
22. **<version>**3.8.1**</version>**
23. **<scope>**test**</scope>**
24. **</dependency>**
25. **</dependencies>**
27. **<build>**
28. **<filters>**
29. **<filter>**src/main/filters/testFilter.properties**</filter>**
30. **</filters>**
31. **<resources>**
32. **<resource>**
33. **<directory>**src/main/resources**</directory>**
34. **<filtering>**true**</filtering>**
35. **</resource>**
36. **</resources>**
37. **</build>**
38. **</project>**

   接着，我们再次运行mvn process-resources指令，然后查看target/classes目录下的application.properties文件的内容会是这个样子：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=mavenAppTest
2. project\_version=1.0-SNAPSHOT
3. project\_modelVersion=4.0.0
4. project\_artifactId=mavenApp
5. settings\_localRepository=D:\\develop\\mavenRepository
6. filter\_username=super

   我们可以看到filter\_username已经由src/main/filters/testFilter.properties中指定的属性myFilter.maven.username的值user变成了pom.xml中properties元素的子元素myFilter.maven.username的值super。由此我们也可以看出在pom.xml文件内部定义的属性值将具有更高的优先级，它会覆盖外部属性文件中相同的属性定义。

   跟需要替换属性值的资源文件可以是不同的文件类型一样，用于关联属性的外部文件也可以是不同的文件类型。为了证明这个问题，我们先在pom.xml中新增一个filter元素，其值指向src/main/filters/testFilter.txt，然后在src/main/filters下新建文件内容为myFilter.maven.fileType=text的testFilter.txt文件，接着在资源文件application.properties中新增一行内容为filter\_fileType=${myFilter.maven.fileType}，之后运行mvn process-resources指令，之后我们可以看到target/classes下的资源文件application.properties的文件内容如下：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=mavenAppTest
2. project\_version=1.0-SNAPSHOT
3. project\_modelVersion=4.0.0
4. project\_artifactId=mavenApp
5. settings\_localRepository=D:\\develop\\mavenRepository
6. filter\_username=super
7. filter\_fileType=text

### 使用系统属性

       过滤资源还可以使用来自系统属性的值，比如java.version，或者在命令行使用-D参数指定的参数。我们来做一下实验，先把我们的application.properties文件修改为如下内容：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=${pom.name}
2. java\_version=${java.version}
3. user\_home=${user.home}
4. command.line.prop=${command.line.prop}

接下来，我们执行指令mvn process-resources –Dcommand.line.prop=hello，之后我们可以看到target/classes下文件application.properties的内容为：

**Properties代码  [收藏代码](javascript:void())**

1. project\_name=mavenAppTest
2. java\_version=1.7.0\_07
3. user\_home=C:\\Users\\andy
4. command.line.prop=hello

## 在pom.xml中定义project的外部依赖包

对于每一个外部依赖都有4个元素是必须定义的，它们是groupId、artifactId、version和scope。其中groupId、artifactId和version必须和依赖包中使用maven打包时定义的pom.xml中定义的相对应的元素的值相同。比如说我们有一个maven项目叫projectA，它的pom.xml中定义的groupId、artifactId和version如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<groupId>**com.tiantian.projectA**</groupId>**
2. **<artifactId>**projectA**</artifactId>**
3. **<version>**1.0**</version>**

       之后我们用mvn package把projectA打包为一个jar包，然后使用mvn install把打好的jar包安装到本地仓库。这个时候有另一个项目projectB，它需要在它的pom.xml中定义对projectA的依赖，这个时候它的pom.xml中定义对projectA依赖的groupId、artifactId和version都应该和projectA的pom.xml中定义的保持一致，具体代码如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<dependency>**
2. **<groupId>**com.tiantian.projectA**</groupId>**
3. **<artifactId>**projectA**</artifactId>**
4. **<version>**1.0**</version>**
5. **<scope>**running**</scope>**
6. **</dependency>**

       元素scope的值表示引用的作用范围，主要取值有compile、test、provided、runtime和system。关于scope的更多介绍将在以后的文中作更多的介绍。对于pom.xml中使用dependency定义的引用信息，maven在需要使用的时候会先从本地仓库取，如果在本地仓库取不到的时候就会尝试从远程仓库下载。

       当我们需要引用一个依赖包，而不知道其对应的groupId等信息的时候，我们可以在网站<http://mvnrepository.com/>中进行搜索寻找对应的依赖包。

## 使用mvn dependency:tree可以查看项目的依赖关系

 进入到项目的pom.xml文件所在的目录后，使用mvn dependency:tree指令可以查看当前项目的依赖关系。

## 修改Maven同时下载artifact的最大数

默认情况下，Maven同时下载artifact的数量是5，这个可以通过参数maven.artifact.threads。比如如果我们希望一次只下载3个artifact的时候，我们就可以这样用：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn –Dmaven.artifact.threads=3 install

       如果需要这个参数永久生效的话我们就可以把这个参数定义到环境变量MAVEN\_OPTS中。

## 在断开网络的情况下执行Maven指令

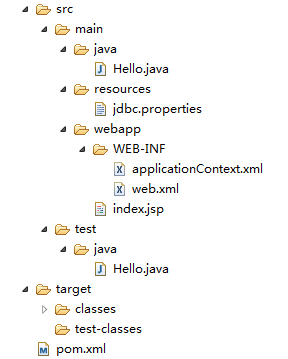
有时候我们连不上网或者我们需要在不访问网络的情况下进行某些操作，这个时候我们就可以在执行Maven指令的时候使用-o参数，比如：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn –o package

## 构建自己的archetype

正如前面内容所介绍的那样，archetype就是Maven给我们提供的一种建立Maven项目的模板，它可以为我们建立我们自己想要的项目的初始目录结构和内容。但是Maven本身为我们提供的那些archetype在实际应用中是远远不够的，这就需要我们来扩展构建自己的archetype。archetype实际上也是一个Maven工件，所以当我们需要构建自己的archetype的时候也是通过对Maven项目的定义来构建的。下面将以一个示例来谈一下我们该如何构建属于自己的archetype，这个例子只是用于介绍如何构建自己的archetype的，其中文档结构和内容等不合理的地方我们在这里就不做深究了。假设我们开发的都是web项目，然后经常使用spring进行开发，然后我们需要创建一个目录结构如下的archetype。



第一步：创建一个简单的Maven项目，定义好它的groupId、artifactId和version，这三个属性就对应了我们需要建立的archetype的三个主要属性，在我们以后使用该archetype的时候会用到。这里我们使用“mvn archetype:generate –DgroupId=com.tiantian –DartifactId=myArchetype –Dversion=1.0 –DinteractiveMode=false”指令建立一个简单的Maven项目。这个时候myArchetype的pom.xml应该是如下这样:

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.tiantian**</groupId>**
5. **<artifactId>**myArchetype**</artifactId>**
6. **<packaging>**jar**</packaging>**
7. **<version>**1.0**</version>**
8. **<name>**myArchetype**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<dependencies>**
11. **<dependency>**
12. **<groupId>**junit**</groupId>**
13. **<artifactId>**junit**</artifactId>**
14. **<version>**3.8.1**</version>**
15. **<scope>**test**</scope>**
16. **</dependency>**
17. **</dependencies>**
18. **</project>**
19. 关于这段代码我们可以把里面无关紧要的内容删掉，那么最简单是形式应该是如下所示：
20. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
21. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
22. **<modelVersion>**4.0.0**</modelVersion>**
23. **<groupId>**com.tiantian**</groupId>**
24. **<artifactId>**myArchetype**</artifactId>**
25. **<version>**1.0**</version>**
26. **</project>**

第二步：建立一个archetype.xml文件，用于描述需要构建的archetype，主要是描述该archetype建立的项目将包含哪些文件。这个文件必须放在“src/main/resources/META-INF/maven”下面。在这里我们这个archetype.xml的内容会如下所示：

**Xml代码  [收藏代码](javascript:void())**

1. **<archetype** xmlns="http://maven.apache.org/plugins/maven-archetype-plugin/archetype/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-plugin/archetype/1.0.0 http://maven.apache.org/xsd/archetype-1.0.0.xsd"**>**
3. **<id>**myArchetype**</id>**
4. **<sources>**
5. **<source>**src/main/java/Hello.java**</source>**
6. **</sources>**
7. **<resources>**
8. **<resource>**src/main/resources/jdbc.properties**</resource>**
9. **<resource>**src/main/webapp/WEB-INF/web.xml**</resource>**
10. **<resource>**src/main/webapp/WEB-INF/application.xml**</resource>**
11. **<resource>**src/main/webapp/index.jsp**</resource>**
12. **</resources>**
13. **<testSources>**
14. **<source>**src/test/java/Hello.java**</source>**
15. **</testSources>**
16. **</archetype>**

id是必须配置的，而且必须和我们定义的archetype的artifactId一致。sources用于定义java源码对应的目录，其下可以定义一系列的source元素。testSources用于定义测试源码对应的目录，其下也是可以定义一系列的source元素。resources、testResources和siteResources下面可以定义一系列的resource元素，表示资源文件所处的位置。

第三步：在archetype.xml中定义了该archetype将包含哪些文件之后我们就需要定义这些文件。那么这些文件是定义在哪里的呢？这些文件的位置都是当前项目的resources/archetype-resources目录下面对应的位置。即“src/main/resources/archetype-resources /src/main/java/Hello.java”、

“src/main/resources/archetype-resources /src/main/resources/jdbc.properties”、

“src/main/resources/archetype-resources /src/main/webapp/WEB-INF/web.xml”、

“src/main/resources/archetype-resources /src/main/webapp/WEB-INF/application.xml”、

“src/main/resources/archetype-resources /src/main/webapp/index.jsp”、

“src/main/resources/archetype-resources /src/test/java/Hello.java”。

这其中当然也包括pom.xml文件的定义。其目录结构大概如下图所示：



这个时候我们也可以指定上述文件的初始内容，比如web.xml的内容为：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"**>**
7. **<welcome-file-list>**
8. **<welcome-file>**index.do**</welcome-file>**
9. **</welcome-file-list>**
11. **<listener>**
12. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
13. **</listener>**
15. **<filter>**
16. **<filter-name>**encodingFilter**</filter-name>**
17. **<filter-class>**org.springframework.web.filter.CharacterEncodingFilter**</filter-class>**
18. **<init-param>**
19. **<param-name>**encoding**</param-name>**
20. **<param-value>**GBK**</param-value>**
21. **</init-param>**
22. **</filter>**
23. **<filter-mapping>**
24. **<filter-name>**encodingFilter**</filter-name>**
25. **<url-pattern>**\*.do**</url-pattern>**
26. **</filter-mapping>**
28. **<servlet>**
29. **<servlet-name>**${artifactId}**</servlet-name>**
30. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
31. **<init-param>**
32. **<param-name>**contextConfigLocation**</param-name>**
33. **<param-value>**classpath:${artifactId}-servlet.xml**</param-value>**
34. **</init-param>**
35. **<load-on-startup>**1**</load-on-startup>**
36. **</servlet>**
37. **<servlet-mapping>**
38. **<servlet-name>**${artifactId}**</servlet-name>**
39. **<url-pattern>**\*.do**</url-pattern>**
40. **</servlet-mapping>**
41. **</web-app>**

       从上面这段代码我们可以看出，我们已经把一些针对特定项目要特定配置的项给参数化了。这样当我们建立项目的时候给定的参数会自动覆盖这些变量。

接下来就是定义我们生成的pom.xml文件的内容了。这里我们如下定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**${groupId}**</groupId>**
5. **<artifactId>**${artifactId}**</artifactId>**
6. **<packaging>**war**</packaging>**
7. **<version>**${version}**</version>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**org.springframework**</groupId>**
11. **<artifactId>**spring-context**</artifactId>**
12. **<version>**3.0.0.RELEASE**</version>**
13. **<scope>**runtime**</scope>**
14. **</dependency>**
15. **</dependencies>**
16. **</project>**

从上面的定义中我们可以看出这里的groupId、artifactId和version都是使用的参数形式，而packaging和dependency就是我们直接写好的。这是因为我们需要使用这个archetype来建立同一类型的不同的artifact。

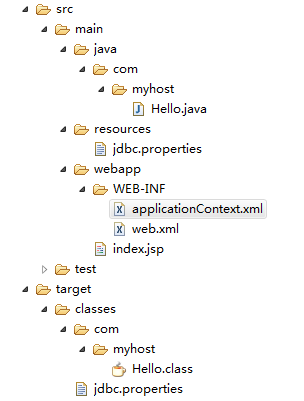
第四步：使用“mvn install”安装该新建的archetype工件到本地仓库。

经过这几步之后我们就可以使用自己定义的myArchetype进行自己项目的建立了。如：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn archetype:generate –DarchetypeGroupId=com.tiantian –DarchetypeArtifactId=myArchetype –DarchetypeVersion=1.0 –DgroupId=com.myhost –DartifactId=myapp –Dversion=1.0

使用该指令建立的项目的目录结构如下所示：



其中web.xml的内容变为：

**Xml代码  [收藏代码](javascript:void())**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"**>**
7. **<welcome-file-list>**
8. **<welcome-file>**index.do**</welcome-file>**
9. **</welcome-file-list>**
11. **<listener>**
12. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
13. **</listener>**
15. **<filter>**
16. **<filter-name>**encodingFilter**</filter-name>**
17. **<filter-class>**org.springframework.web.filter.CharacterEncodingFilter**</filter-class>**
18. **<init-param>**
19. **<param-name>**encoding**</param-name>**
20. **<param-value>**GBK**</param-value>**
21. **</init-param>**
22. **</filter>**
23. **<filter-mapping>**
24. **<filter-name>**encodingFilter**</filter-name>**
25. **<url-pattern>**\*.do**</url-pattern>**
26. **</filter-mapping>**
28. **<servlet>**
29. **<servlet-name>**myapp**</servlet-name>**
30. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
31. **<init-param>**
32. **<param-name>**contextConfigLocation**</param-name>**
33. **<param-value>**classpath:myapp-servlet.xml**</param-value>**
34. **</init-param>**
35. **<load-on-startup>**1**</load-on-startup>**
36. **</servlet>**
37. **<servlet-mapping>**
38. **<servlet-name>**myapp**</servlet-name>**
39. **<url-pattern>**\*.do**</url-pattern>**
40. **</servlet-mapping>**
42. **</web-app>**

使用myArchetype建立的Maven项目myapp中的pom.xml的内容如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.myhost**</groupId>**
5. **<artifactId>**myapp**</artifactId>**
6. **<packaging>**war**</packaging>**
7. **<version>**1.0**</version>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**org.springframework**</groupId>**
11. **<artifactId>**spring-context**</artifactId>**
12. **<version>**3.0.0.RELEASE**</version>**
13. **<scope>**runtime**</scope>**
14. **</dependency>**
15. **</dependencies>**
16. **</project>**

       以上就是自己构建Maven的archetype的基本步骤。其实这些东西如果全部需要自己去建好的话还是比较麻烦的。针对于构建自己的archetype，Maven也给我们提供了一个建立自己的archetype的archetype，那就是3.1节里面提过的maven-archetype-archetype。下面就简单介绍一下利用Maven提供的maven-archetype-archetype来建立我们自己的archetype。

**Cmd代码  [收藏代码](javascript:void())**

1. mvn archetype:generate
2. –DarchetypeGroupId=org.apache.maven.archetypes
3. –DarchetypeArtifactId=maven-archetype-archetype
4. –DgroupId=com.tiantian  –DartifactId=myArchetype
5. –Dversion=1.0 –DinteractiveMode=false

       通过上面的指令可以建立简单的archetype的示例，接着我们就可以根据自己的需要修改需要建立的archetype的描述文件archetype.xml，然后在对应的archetype-resources目录下建立对应的文件即可。接下来的操作一样是通过“mvn install”指令安装该archetype工件到本地仓库，这样我们以后就可以通过该archetype来建立我们预定好的Maven项目了。

## 使用mvn javadoc:jar生成对应的javadoc

在Maven项目的根目录即pom.xml文件所在的目录下使用mvn javadoc:jar可以生成当前项目对应的java doc。

## 使用mvn dependency:sources下载对应的源码

在Maven项目的根目录即pom.xml文件所在的目录下使用mvn dependency:sources可以下载对应依赖包的源码。

# profile介绍

## rofile简介

profile可以让我们定义一系列的配置信息，然后指定其激活条件。这样我们就可以定义多个profile，然后每个profile对应不同的激活条件和配置信息，从而达到不同环境使用不同配置信息的效果。比如说，我们可以通过profile定义在jdk1.5以上使用一套配置信息，在jdk1.5以下使用另外一套配置信息；或者有时候我们可以通过操作系统的不同来使用不同的配置信息，比如windows下是一套信息，linux下又是另外一套信息，等等。具体的激活条件有哪些我在后文会讲到。

## profile的定义位置

对于使用Maven3，我们可以有多个地方定义profile。定义的地方不同，它的作用范围也不同。

（1）    针对于特定项目的profile配置我们可以定义在该项目的pom.xml中。

（2）    针对于特定用户的profile配置，我们可以在用户的settings.xml文件中定义profile。该文件在用户家目录下的“.m2”目录下。

（3）    全局的profile配置。全局的profile是定义在Maven安装目录下的“conf/settings.xml”文件中的。

## profile中能定义的信息

profile中能够定义的配置信息跟profile所处的位置是相关的。以下就分两种情况来讨论，一种是定义在settings.xml中，另一种是定义在pom.xml中。

## profile定义在settings.xml中

当profile定义在settings.xml中时意味着该profile是全局的，它会对所有项目或者某一用户的所有项目都产生作用。因为它是全局的，所以在settings.xml中只能定义一些相对而言范围宽泛一点的配置信息，比如远程仓库等。而一些比较细致一点的需要根据项目的不同来定义的就需要定义在项目的pom.xml中。具体而言，能够定义在settings.xml中的信息有<repositories>、<pluginRepositories>和<properties>。定义在<properties>里面的键值对可以在pom.xml中使用。

## profile定义在pom.xml中

定义在pom.xml中的profile可以定义更多的信息。主要有以下这些：

l  <repositories>

l  <pluginRepositories>

l  <dependencies>

l  <plugins>

l  <properties>

l  <dependencyManagement>

l  <distributionManagement>

l  还有build元素下面的子元素，主要包括：

<defaultGoal>

<resources>

<testResources>

<finalName>

## profile的激活方式

Maven给我们提供了多种不同的profile激活方式。比如我们可以使用-P参数显示的激活一个profile，也可以根据环境条件的设置让它自动激活等。下面将对它们一一进行介绍：

## 使用activeByDefault设置激活

先看下面一个配置

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<properties>**
5. **<hello>**world**</hello>**
6. **</properties>**
7. **<activation>**
8. **<activeByDefault>**true**</activeByDefault>**
9. **</activation>**
10. **</profile>**
12. **<profile>**
13. **<id>**profileTest2**</id>**
14. **<properties>**
15. **<hello>**andy**</hello>**
16. **</properties>**
17. **</profile>**
18. **</profiles>**

        我们可以在profile中的activation元素中指定激活条件，当没有指定条件，然后指定activeByDefault为true的时候就表示当没有指定其他profile为激活状态时，该profile就默认会被激活。所以当我们调用mvn package的时候上面的profileTest1将会被激活，但是当我们使用mvn package –P profileTest2的时候将激活profileTest2，而这个时候profileTest1将不会被激活。

**4.4.2  在settings.xml中使用activeProfiles指定处于激活状态的profile**

我们可以在settings.xml中使用activeProfiles来指定需要激活的profile，这种方式激活的profile将所有情况下都处于激活状态。比如现在我们定义了如下两个profile

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<properties>**
5. **<hello>**world**</hello>**
6. **</properties>**
7. **</profile>**
9. **<profile>**
10. **<id>**profileTest2**</id>**
11. **<properties>**
12. **<hello>**andy**</hello>**
13. **</properties>**
14. **</profile>**
15. **</profiles>**

       这里的profile可以是定义在settings.xml中的，也可以是定义在pom.xml中的。这个时候如果我们需要指定profileTest1为激活状态，那么我们就可以在settings.xml中定义activeProfiles，具体定义如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<activeProfiles>**
2. **<activeProfile>**profileTest1**</activeProfile>**
3. **</activeProfiles>**

       考虑这样一种情况，我们在activeProfiles下同时定义了多个需要激活的profile。这里还拿上面的profile定义来举例，我们定义了同时激活profileTest1和profileTest2。

**Xml代码  [收藏代码](javascript:void())**

1. **<activeProfiles>**
2. **<activeProfile>**profileTest1**</activeProfile>**
3. **<activeProfile>**profileTest2**</activeProfile>**
4. **</activeProfiles>**

       从profileTest1和profileTest2我们可以看出它们共同定义了属性hello。那么这个时候我在pom.xml中使用属性hello的时候，它到底取的哪个值呢？是根据activeProfile定义的顺序，后面的覆盖前面的吗？根据我的测试，答案是非也，它是根据profile定义的先后顺序来进行覆盖取值的，然后后面定义的会覆盖前面定义的。

**4.4.3  使用-P参数显示的激活一个profile**

假设我们现在有如下定义的profiles

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<properties>**
5. **<hello>**world**</hello>**
6. **</properties>**
7. **</profile>**
8. **<profile>**
9. **<id>**profileTest2**</id>**
10. **<properties>**
11. **<hello>**andy**</hello>**
12. **</properties>**
13. **</profile>**
14. **<profiles>**

       那么当我们在进行Maven操作时就可以使用-P参数显示的指定当前激活的是哪一个profile了。比如我们需要在对项目进行打包的时候使用id为profileTest1的profile，我们就可以这样做：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn package –P profileTest1

       当我们使用activeByDefault或settings.xml中定义了处于激活的profile，但是当我们在进行某些操作的时候又不想它处于激活状态，这个时候我们可以这样做：

**Cmd代码  [收藏代码](javascript:void())**

1. Mvn package –P !profileTest1

       这里假设profileTest1是在settings.xml中使用activeProfile标记的处于激活状态的profile，那么当我们使用“-P !profile”的时候就表示在当前操作中该profile将不处于激活状态。

**4.4.4****根据环境来激活profile**

profile一个非常重要的特性就是它可以根据不同的环境来激活，比如说根据操作系统的不同激活不同的profile，也可以根据jdk版本的不同激活不同的profile，等等。

**4.4.4.1根据jdk来激活profile**

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<jdk>**1.5**</jdk>**
5. **</profile>**
6. **<profiles>**

       上面情况表示在jdk为1.5版本系列的时候激活profileTest1。

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<jdk>**[1.4,1.7)**</jdk>**
5. **</profile>**
6. **<profiles>**

       上面的情况表示在jdk为1.4、1.5和1.6的时候激活profileTest1。

**4.4.4.2根据操作系统来激活profile**

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<os>**
6. **<name>**Windows XP**</name>**
7. **<family>**Windows**</family>**
8. **<arch>**x86**</arch>**
9. **<version>**5.1.2600**</version>**
10. **</os>**
11. **</activation>**
12. **</profile>**
13. **</profiles>**

       上面的情况就是根据操作系统的类型来激活profileTest1。

**4.4.4.3根据系统属性来激活profile**

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<property>**
6. **<name>**hello**</name>**
7. **<value>**world**</value>**
8. **</property>**
9. **</activation>**
10. **</profile>**
11. **</profiles>**

上面的profileTest1将在提供了系统属性hello，并且其值为world的时候激活。下面的做法可以激活profileTest1。

**Cmd代码  [收藏代码](javascript:void())**

1. mvn package –Dhello=world

       当是下面的这种定义形式时，profileTest1将在指定了系统属性hello，且其值为任意值的时候被激活。

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<property>**
6. **<name>**hello**</name>**
7. **</property>**
8. **</activation>**
9. **</profile>**
10. **</profiles>**

**4.4.4.4根据文件是否存在激活profile**

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<file>**
6. **<exists>**target**</exists>**
7. **</file>**
8. **</activation>**
9. **</profile>**
10. **</profiles>**

上面的定义表示当存在target文件时激活profileTest1。

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<file>**
6. **<missing>**target**</missing>**
7. **</file>**
8. **</activation>**
9. **</profile>**
10. **</profiles>**

       上面的定义表示当不存在target文件时激活profileTest1。

**4.5     查看当前处于激活状态的profile**

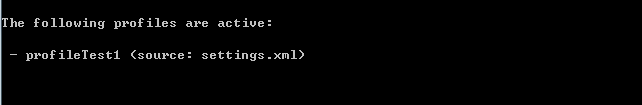
我们可以同时定义多个profile，那么在建立项目的过程中，到底激活的是哪一个profile呢？Maven为我们提供了一个指令可以查看当前处于激活状态的profile都有哪些，这个指定就是mvn help:active-profiles。

现在假设我们的settings.xml文件中有如下profile的定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<profiles>**
2. **<profile>**
3. **<id>**profileTest1**</id>**
4. **<activation>**
5. **<file>**
6. **<missing>**target**</missing>**
7. **</file>**
8. **</activation>**
9. **</profile>**
10. **</profiles>**
12. **<activeProfiles>**
13. **<activeProfile>**profileTest1**</activeProfile>**
14. **</activeProfiles>**

       这个时候我们可以看到，我们已经定义了profileTest1始终为激活状态，这个时候我们使用mvn help:active-profiles查看处于激活状态的profile时，就会打印出如下内容：



# Maven仓库介绍

## 简介

Maven仓库是用来存储工件的。Maven仓库主要包括本地仓库和远程仓库。当Maven需要一个工件的时候，它会先从本地仓库取，只有在本地仓库不存在对应的工件的时候，才会从远程仓库把对应的工件下载下来保存在本地仓库，并从本地仓库获取所需的工件。默认的本地仓库位置是用户家目录下面的“.m2/repository”，我们可以在Maven安装目录下面的settings.xml文件中通过localRepository定义本地仓库的位置。默认的远程仓库是“<http://repo.maven.apache.org/maven2/>”。

在这里我想陈述一个问题，Maven官方是不主张我们在使用Maven的时候还把项目依赖的一些jar包提交到svn等版本控制系统中进行版本控制。这主要有以下原因：

1)      我们已经使用了Maven来管理我们的依赖jar包，那么这些jar包就都会保存在本地仓库中，我们没有必要每个项目都保存一个依赖jar包的拷贝，这会浪费很多的磁盘空间，也违背了Maven设计的初衷。

2)      当我们没有把依赖的jar包提交到版本控制系统的时候，这也就意味着我们这个项目的容量会相对而言小一些，这给我们带来的优点就是在我们检出该项目的时候速度会相对而言更快一些。

3)      Jar包一般都不需要进行版本控制，因为它的变化一般不大，我们很少会去更改一个jar包。

有的时候因为安全或者速度的原因，我们不允许直接从互联网上下载依赖的jar包，这个时候内部的仓库就出来了。我们可以从这个内部仓库下载工件，也可以把工件发布到该仓库。这个内部仓库的概念就相当于是公司内部自己管理了一套工件库，而且可以自由的往这个工件库里面发布公司自己的工件供大家共享。

## 安装第三方jar包到本地仓库

有的时候可能我们需要把本地的第三方jar包安装到我们的Maven本地仓库来构建我们的Maven项目。Maven为我们提供了一个指令可以很轻松的帮我们实现这个功能。

**Cmd代码  [收藏代码](javascript:void())**

1. mvn install:install-file -Dfile=<filePath> -DgroupId=<groupId> -DartifactId=<artifactId> -Dversion=<version> -Dpackaging=<packagingType>

       参数file表示需要安装的第三方jar包在本地的路径；

       参数groupId用于定义该jar包安装后的groupId；

       参数artifactId用于定义该jar包安装后的artifactId；

       参数version用于定义该jar包安装后的版本；

       参数packaging用于定义该jar包安装后的打包类型。

       比如现在我想把我电脑上的“D:\develop\lib\mysql-connector-java-5.1.12-bin.jar”安装到我的Maven本地仓库，那么我就可以在命令窗口运行以下指令来达到这个目的：

       mvn install:install-file -Dfile=D:\develop\lib\mysql-connector-5.1.12-bin.jar -DgroupId=mysql -DartifactId=mysql -Dversion=5.1.12 -Dpackaging=jar

       之后在其他Maven项目中我们就可以根据定义好的groupId、artifactId、version和packaging类型来添加这里定义好的mysql-connector-5.1.12-bin.jar的引用了。

## 部署第三方jar包到远程仓库

前面安装到本地仓库的第三方jar包只能是在本地使用，这样其他人是无法访问到的。如果需要其他人也能访问到的话，我们就需要把它部署到我们的远程仓库上去。我们可以使用以下Maven指令来部署一个第三方jar包到指定的远程仓库。

**Cmd代码  [收藏代码](javascript:void())**

1. mvn deploy:deploy-file
2. -Dfile=<filePath>
3. -DgroupId=<groupId>
4. -DartifactId=<artifactId>
5. -Dversion=<version>
6. -Dpackaging=<packagingType>
7. -DrepositoryId=<repositoryId>
8. -Durl=<urlOfTheRepositoryToDeploy>

把第三方jar包部署到远程仓库的参数和安装第三方jar包到本地仓库类似，但它多了两个参数，一个是repositoryId和url。repositoryId表示需要部署到的远程仓库的id，这个远程仓库是定义在settings.xml中的；url表示需要部署到的远程仓库的url。

默认情况下，使用deploy:deploy-file部署的第三方jar包将会生成一个通用的pom。如果在部署的过程中不需要生成这个pom，我们可以在使用该指令的时候加上参数“-DgeneratePom=false”。

如果我们在使用deploy:deploy-file部署第三方jar到远程仓库需要使用一个已有的pom的时候，我们需要在使用该指令的时候加上参数“-DpomFile=<pomFilePath>”。如：

**Cmd代码  [收藏代码](javascript:void())**

1. mvn deploy:deploy-file
2. -Dfile=<filePath>
3. -DpomFile=<pomFilePath>
4. -DrepositoryId=<repositoryId>
5. -Durl=<urlOfTheRepositoryToDeploy>

细心的读者可能已经发现了，我们在使用了参数pomFile的时候没有指定groupId、artifactId、version和packaging参数。这是因为当我们指定了pomFile的时候这些参数都可以从指定的pom文件中获得。

当我们需要部署的是一个源码jar包的时候，packaging应该指定为java-source，而且generatePom应该指定为false。

# Maven的pom.xml介绍

## 简介

       pom.xml文件是Maven进行工作的主要配置文件。在这个文件中我们可以配置Maven项目的groupId、artifactId和version等Maven项目必须的元素；可以配置Maven项目需要使用的远程仓库；可以定义Maven项目打包的形式；可以定义Maven项目的资源依赖关系等等。对于一个最简单的pom.xml的定义必须包含modelVersion、groupId、artifactId和version这四个元素，当然这其中的元素也是可以从它的父项目中继承的。在Maven中，使用groupId、artifactId和version组成groupdId:artifactId:version的形式来唯一确定一个项目。

## pom.xml的继承、聚合与依赖

       我们知道Maven在建立项目的时候是基于Maven项目下的pom.xml进行的，我们项目依赖的信息和一些基本信息都是在这个文件里面定义的。那如果当我们有多个项目要进行，这多个项目有些配置内容是相同的，有些是要彼此关联的，那如果按照传统的做法的话我们就需要在多个项目中都定义这些重复的内容。这无疑是非常耗费时间和不易维护的。好在Maven给我们提供了一个pom的继承和聚合的功能。

       对于使用java的人而言，继承这个词大家应该都不陌生。要继承pom就需要有一个父pom，在Maven中定义了超级pom.xml，任何没有申明自己父pom.xml的pom.xml都将默认继承自这个超级pom.xml。

       先来看一下这个超级pom.xml的定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<modelVersion>**4.0.0**</modelVersion>**
3. **<name>**Maven Default Project**</name>**
5. **<repositories>**
6. **<repository>**
7. **<id>**central**</id>**
8. **<name>**Maven Repository Switchboard**</name>**
9. **<layout>**default**</layout>**
10. **<url>**http://repo1.maven.org/maven2**</url>**
11. **<snapshots>**
12. **<enabled>**false**</enabled>**
13. **</snapshots>**
14. **</repository>**
15. **</repositories>**
17. **<pluginRepositories>**
18. **<pluginRepository>**
19. **<id>**central**</id>**
20. **<name>**Maven Plugin Repository**</name>**
21. **<url>**http://repo1.maven.org/maven2**</url>**
22. **<layout>**default**</layout>**
23. **<snapshots>**
24. **<enabled>**false**</enabled>**
25. **</snapshots>**
26. **<releases>**
27. **<updatePolicy>**never**</updatePolicy>**
28. **</releases>**
29. **</pluginRepository>**
30. **</pluginRepositories>**
32. **<build>**
33. **<directory>**${project.basedir}/target**</directory>**
34. **<outputDirectory>**${project.build.directory}/classes**</outputDirectory>**
35. **<finalName>**${project.artifactId}-${project.version}**</finalName>**
36. **<testOutputDirectory>**${project.build.directory}/test-classes**</testOutputDirectory>**
37. **<sourceDirectory>**${project.basedir}/src/main/java**</sourceDirectory>**
38. <!-- TODO: MNG-3731 maven-plugin-tools-api < 2.4.4 expect this to be relative... -->
39. **<scriptSourceDirectory>**src/main/scripts**</scriptSourceDirectory>**
40. **<testSourceDirectory>**${project.basedir}/src/test/java**</testSourceDirectory>**
41. **<resources>**
42. **<resource>**
43. **<directory>**${project.basedir}/src/main/resources**</directory>**
44. **</resource>**
45. **</resources>**
46. **<testResources>**
47. **<testResource>**
48. **<directory>**${project.basedir}/src/test/resources**</directory>**
49. **</testResource>**
50. **</testResources>**
51. **<pluginManagement>**
52. **<plugins>**
53. **<plugin>**
54. **<artifactId>**maven-antrun-plugin**</artifactId>**
55. **<version>**1.3**</version>**
56. **</plugin>**
57. **<plugin>**
58. **<artifactId>**maven-assembly-plugin**</artifactId>**
59. **<version>**2.2-beta-2**</version>**
60. **</plugin>**
61. **<plugin>**
62. **<artifactId>**maven-clean-plugin**</artifactId>**
63. **<version>**2.2**</version>**
64. **</plugin>**
65. **<plugin>**
66. **<artifactId>**maven-compiler-plugin**</artifactId>**
67. **<version>**2.0.2**</version>**
68. **</plugin>**
69. **<plugin>**
70. **<artifactId>**maven-dependency-plugin**</artifactId>**
71. **<version>**2.0**</version>**
72. **</plugin>**
73. **<plugin>**
74. **<artifactId>**maven-deploy-plugin**</artifactId>**
75. **<version>**2.4**</version>**
76. **</plugin>**
77. **<plugin>**
78. **<artifactId>**maven-ear-plugin**</artifactId>**
79. **<version>**2.3.1**</version>**
80. **</plugin>**
81. **<plugin>**
82. **<artifactId>**maven-ejb-plugin**</artifactId>**
83. **<version>**2.1**</version>**
84. **</plugin>**
85. **<plugin>**
86. **<artifactId>**maven-install-plugin**</artifactId>**
87. **<version>**2.2**</version>**
88. **</plugin>**
89. **<plugin>**
90. **<artifactId>**maven-jar-plugin**</artifactId>**
91. **<version>**2.2**</version>**
92. **</plugin>**
93. **<plugin>**
94. **<artifactId>**maven-javadoc-plugin**</artifactId>**
95. **<version>**2.5**</version>**
96. **</plugin>**
97. **<plugin>**
98. **<artifactId>**maven-plugin-plugin**</artifactId>**
99. **<version>**2.4.3**</version>**
100. **</plugin>**
101. **<plugin>**
102. **<artifactId>**maven-rar-plugin**</artifactId>**
103. **<version>**2.2**</version>**
104. **</plugin>**
105. **<plugin>**
106. **<artifactId>**maven-release-plugin**</artifactId>**
107. **<version>**2.0-beta-8**</version>**
108. **</plugin>**
109. **<plugin>**
110. **<artifactId>**maven-resources-plugin**</artifactId>**
111. **<version>**2.3**</version>**
112. **</plugin>**
113. **<plugin>**
114. **<artifactId>**maven-site-plugin**</artifactId>**
115. **<version>**2.0-beta-7**</version>**
116. **</plugin>**
117. **<plugin>**
118. **<artifactId>**maven-source-plugin**</artifactId>**
119. **<version>**2.0.4**</version>**
120. **</plugin>**
121. **<plugin>**
122. **<artifactId>**maven-surefire-plugin**</artifactId>**
123. **<version>**2.4.3**</version>**
124. **</plugin>**
125. **<plugin>**
126. **<artifactId>**maven-war-plugin**</artifactId>**
127. **<version>**2.1-alpha-2**</version>**
128. **</plugin>**
129. **</plugins>**
130. **</pluginManagement>**
131. **</build>**
133. **<reporting>**
134. **<outputDirectory>**${project.build.directory}/site**</outputDirectory>**
135. **</reporting>**
136. **<profiles>**
137. **<profile>**
138. **<id>**release-profile**</id>**
140. **<activation>**
141. **<property>**
142. **<name>**performRelease**</name>**
143. **<value>**true**</value>**
144. **</property>**
145. **</activation>**
147. **<build>**
148. **<plugins>**
149. **<plugin>**
150. **<inherited>**true**</inherited>**
151. **<groupId>**org.apache.maven.plugins**</groupId>**
152. **<artifactId>**maven-source-plugin**</artifactId>**
153. **<executions>**
154. **<execution>**
155. **<id>**attach-sources**</id>**
156. **<goals>**
157. **<goal>**jar**</goal>**
158. **</goals>**
159. **</execution>**
160. **</executions>**
161. **</plugin>**
162. **<plugin>**
163. **<inherited>**true**</inherited>**
164. **<groupId>**org.apache.maven.plugins**</groupId>**
165. **<artifactId>**maven-javadoc-plugin**</artifactId>**
166. **<executions>**
167. **<execution>**
168. **<id>**attach-javadocs**</id>**
169. **<goals>**
170. **<goal>**jar**</goal>**
171. **</goals>**
172. **</execution>**
173. **</executions>**
174. **</plugin>**
175. **<plugin>**
176. **<inherited>**true**</inherited>**
177. **<groupId>**org.apache.maven.plugins**</groupId>**
178. **<artifactId>**maven-deploy-plugin**</artifactId>**
179. **<configuration>**
180. **<updateReleaseInfo>**true**</updateReleaseInfo>**
181. **</configuration>**
182. **</plugin>**
183. **</plugins>**
184. **</build>**
185. **</profile>**
186. **</profiles>**
188. **</project>**

       对于一个pom.xml来说有几个元素是必须定义的，一个是project根元素，然后就是它里面的modelVersion、groupId、artifactId和version。由上面的超级pom.xml的内容我们可以看到pom.xml中没有groupId、artifactId和version的定义，所以我们在建立自己的pom.xml的时候就需要定义这三个元素。和java里面的继承类似，子pom.xml会完全继承父pom.xml中所有的元素，而且对于相同的元素，一般子pom.xml中的会覆盖父pom.xml中的元素，但是有几个特殊的元素它们会进行合并而不是覆盖。这些特殊的元素是：

  dependencies

  developers

  contributors

  plugin列表，包括plugin下面的reports

  resources

### 继承

#### 被继承项目与继承项目是父子目录关系

       现在假设我们有一个项目projectA，它的pom.xml定义如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectA**</artifactId>**
6. **<packaging>**jar**</packaging>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **</project>**

       然后我们有另一个项目projectB，而且projectB是跟projectA的pom.xml文件处于同一个目录下，这时候如果projectB需要继承自projectA的话我们可以这样定义projectB的pom.xml文件。

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<parent>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectA**</artifactId>**
6. **<version>**1.0-SNAPSHOT**</version>**
7. **</parent>**
8. **<modelVersion>**4.0.0**</modelVersion>**
9. **<groupId>**com.tiantian.mavenTest**</groupId>**
10. **<artifactId>**projectB**</artifactId>**
11. **<packaging>**jar**</packaging>**
12. **<version>**1.0-SNAPSHOT**</version>**
13. **</project>**

       由projectB的pom.xml文件的定义我们可以知道，当需要继承指定的一个Maven项目时，我们需要在自己的pom.xml中定义一个parent元素，在这个元素中指明需要继承项目的groupId、artifactId和version。

#### 被继承项目与继承项目的目录结构不是父子关系

       当被继承项目与继承项目的目录结构不是父子关系的时候，我们再利用上面的配置是不能实现Maven项目的继承关系的，这个时候我们就需要在子项目的pom.xml文件定义中的parent元素下再加上一个relativePath元素的定义，用以描述父项目的pom.xml文件相对于子项目的pom.xml文件的位置。

       假设我们现在还是有上面两个项目，projectA和projectB，projectB还是继承自projectA，但是现在projectB不在projectA的子目录中，而是与projectA处于同一目录中。这个时候projectA和projectB的目录结构如下：

       ------projectA

              ------pom.xml

       ------projectB

              ------pom.xml

       这个时候我们可以看出projectA的pom.xml相对于projectB的pom.xml的位置是“../projectA/pom.xml”，所以这个时候projectB的pom.xml的定义应该如下所示：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>**
3. **<parent>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectA**</artifactId>**
6. **<version>**1.0-SNAPSHOT**</version>**
7. **<relativePath>**../projectA/pom.xml**</relativePath>**
8. **</parent>**
9. **<modelVersion>**4.0.0**</modelVersion>**
10. **<groupId>**com.tiantian.mavenTest**</groupId>**
11. **<artifactId>**projectB**</artifactId>**
12. **<packaging>**jar**</packaging>**
13. **<version>**1.0-SNAPSHOT**</version>**
14. **</project>**

### 聚合

       对于聚合这个概念搞java的人应该都不会陌生。先来说说我对聚合和被聚合的理解，比如说如果projectA聚合到projectB，那么我们就可以说projectA是projectB的子模块， projectB是被聚合项目，也可以类似于继承那样称为父项目。对于聚合而言，这个主体应该是被聚合的项目。所以，我们需要在被聚合的项目中定义它的子模块，而不是像继承那样在子项目中定义父项目。具体做法是：

  修改被聚合项目的pom.xml中的packaging元素的值为pom

  在被聚合项目的pom.xml中的modules元素下指定它的子模块项目

对于聚合而言，当我们在被聚合的项目上使用Maven命令时，实际上这些命令都会在它的子模块项目上使用。这就是Maven中聚合的一个非常重要的作用。假设这样一种情况，你同时需要打包或者编译projectA、projectB、projectC和projectD，按照正常的逻辑我们一个一个项目去使用mvn compile或mvn package进行编译和打包，对于使用Maven而言，你还是这样使用的话是非常麻烦的。因为Maven给我们提供了聚合的功能。我们只需要再定义一个超级项目，然后在超级项目的pom.xml中定义这个几个项目都是聚合到这个超级项目的。之后我们只需要对这个超级项目进行mvn compile，它就会把那些子模块项目都进行编译。

#### 被聚合项目和子模块项目在目录结构上是父子关系

还拿上面定义的projectA和projectB来举例子，现在假设我们需要把projectB聚合到projectA中。projectA和projectB的目录结构如下所示：

------projectA

       ------projectB

              -----pom.xml

       ------pom.xml

这个时候projectA的pom.xml应该这样定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectA**</artifactId>**
6. **<version>**1.0-SNAPSHOT**</version>**
7. **<packaging>**pom**</packaging>**
8. **<modules>**
9. **<module>**projectB**</module>**
10. **</modules>**
11. **</project>**

由上面的定义我们可以看到被聚合的项目的packaging类型应该为pom，而且一个项目可以有多个子模块项目。对于聚合这种情况，我们使用子模块项目的artifactId来作为module的值，表示子模块项目相对于被聚合项目的地址，在上面的示例中就表示子模块projectB是处在被聚合项目的子目录下，即与被聚合项目的pom.xml处于同一目录。这里使用的module值是子模块projectB对应的目录名projectB，而不是子模块对应的artifactId。这个时候当我们对projectA进行mvn package命令时，实际上Maven也会对projectB进行打包。

#### 被聚合项目与子模块项目在目录结构上不是父子关系

那么当被聚合项目与子模块项目在目录结构上不是父子关系的时候，我们应该怎么来进行聚合呢？还是像继承那样使用relativePath元素吗？答案是非也，具体做法是在module元素中指定以相对路径的方式指定子模块。我们来看下面一个例子。

继续使用上面的projectA和projectB，还是需要把projectB聚合到projectA，但是projectA和projectB的目录结构不再是父子关系，而是如下所示的这种关系：

------projectA

       ------pom.xml

------projectB

       ------pom.xml

这个时候projectA的pom.xml文件就应该这样定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
5. **<groupId>**com.tiantian.mavenTest**</groupId>**
6. **<artifactId>**projectA**</artifactId>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **<packaging>**pom**</packaging>**
9. **<modules>**
10. **<module>**../projectB**</module>**
11. **</modules>**
12. **</project>**

注意看module的值是“../projectB”，我们知道“..”是代表当前目录的上层目录，所以它表示子模块projectB是被聚合项目projectA的pom.xml文件所在目录（即projectA）的上层目录下面的子目录，即与projectA处于同一目录层次。**注意，这里的projectB对应的是projectB这个项目的目录名称，而不是它的artifactId。**

#### 聚合与继承同时进行

       假设有这样一种情况，有两个项目，projectA和projectB，现在我们需要projectB继承projectA，同时需要把projectB聚合到projectA。然后projectA和projectB的目录结构如下：

       ------projectA

              ------pom.xml

       ------projectB

              ------pom.xml

       那么这个时候按照上面说的那样，projectA的pom.xml中需要定义它的packaging为pom，需要定义它的modules，所以projectA的pom.xml应该这样定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectA**</artifactId>**
6. **<version>**1.0-SNAPSHOT**</version>**
7. **<packaging>**pom**</packaging>**
8. **<modules>**
9. **<module>**../projectB**</module>**
10. **</modules>**
11. **</project>**

       而projectB是继承自projectA的，所以我们需要在projectB的pom.xml文件中新增一个parent元素，用以定义它继承的项目信息。所以projectB的pom.xml文件的内容应该这样定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**com.tiantian.mavenTest**</groupId>**
6. **<artifactId>**projectA**</artifactId>**
7. **<version>**1.0-SNAPSHOT**</version>**
8. **<relativePath>**../projectA/pom.xml**</relativePath>**
9. **</parent>**
10. **<groupId>**com.tiantian.mavenTest**</groupId>**
11. **<artifactId>**projectB**</artifactId>**
12. **<version>**1.0-SNAPSHOT**</version>**
13. **<packaging>**jar**</packaging>**
14. **</project>**

### 依赖Dependency

       项目之间的依赖是通过pom.xml文件里面的dependencies元素下面的dependency元素进行的。一个dependency元素定义一个依赖关系。在dependency元素中我们主要通过依赖项目的groupId、artifactId和version来定义所依赖的项目。

       先来看一个简单的项目依赖的示例吧，假设我现在有一个项目projectA，然后它里面有对junit的依赖，那么它的pom.xml就类似以下这个样子：

**Xml代码  [收藏代码](javascript:void())**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.tiantian.mavenTest**</groupId>**
5. **<artifactId>**projectB**</artifactId>**
6. **<version>**1.0-SNAPSHOT**</version>**
7. **<packaging>**jar**</packaging>**
9. **<dependencies>**
10. **<dependency>**
11. **<groupId>**junit**</groupId>**
12. **<artifactId>**junit**</artifactId>**
13. **<version>**3.8.1**</version>**
14. **<scope>**test**</scope>**
15. **<optional>**true**</optional>**
16. **</dependency>**
17. **</dependencies>**
18. **</project>**

       在dependency元素中除了可以指定依赖项目的groupId、artifactId和version之外，还可以指定以下元素：

  type：对应于依赖项目的packaging类型，默认是jar

  scope：表示依赖项目的一个作用范围。scope的主要取值范围如下（还有一个是在Maven2.0.9以后版本才支持的import，关于import作用域将在后文《Dependency介绍》中做介绍）：

  compile：这是它的默认值，这种类型很容易让人产生误解，以为只有在编译的时候才是需要的，其实这种类型表示所有的情况都是有用的，包括编译和运行时。而且这种类型的依赖性是可以传递的。

  provided：这个跟compile很类似，但是它表示你期望这个依赖项目在运行时由JDK或者容器来提供。这种类型表示该依赖只有在测试和编译的情况下才有效，在运行时将由JDK或者容器提供。这种类型的依赖性是不可传递的。

  runtime：这种类型表示该依赖在编译的时候不是必须的，只有在运行的时候才是必须的。

  test：这表示这种依赖只有测试的时候才需要，正常情况下是不需要的。

  system：这种类型跟provided类似，唯一不同的就是这种类型的依赖我们要自己提供jar包，这需要与另一个元素systemPath来结合使用。systemPath将指向我们系统上的jar包的路径，而且必须是给定的绝对路径。

  systemPath：上面已经说过了这个元素是在scope的值为system的时候用于指定依赖的jar包在系统上的位置的，而且是绝对路径。该元素必须在依赖的 jar包的scope为system时才能使用，否则Maven将报错。

  optional：当该项目本身作为其他项目的一个依赖时标记该依赖为可选项。假设现在projectA有一个依赖性projectB，我们把projectB这个依赖项设为optional，这表示projectB在projectA的运行时不一定会用到。这个时候如果我们有另一个项目projectC，它依赖于projectA，那么这个时候因为projectB对于projectA是可选的，所以Maven在建立projectC的时候就不会安装projectB，这个时候如果projectC确实需要使用到projectB，那么它就可以定义自己对projectB的依赖。当一个依赖是可选的时候，我们把optional元素的值设为true，否则就不设置optional元素。

  exclusions：考虑这样一种情况，我们的projectA依赖于projectB，然后projectB又依赖于projectC，但是在projectA里面我们不需要projectB依赖的projectC，那么这个时候我们就可以在依赖projectB的时候使用exclusions元素下面的exclusion排除projectC。这个时候我们可以这样定义projectA对projectB的依赖：

**Xml代码  [收藏代码](javascript:void())**

1. **<dependencies>**
2. **<dependency>**
3. **<groupId>**com.tiantian.mavenTest**</groupId>**
4. **<artifactId>**projectB**</artifactId>**
5. **<version>**1.0-SNAPSHOT**</version>**
6. **<exclusions>**
7. **<exclusion>**
8. **<groupId>**com.tiantian.mavenTest**</groupId>**
9. **<artifactId>**projectC**</artifactId>**
10. **</exclusion>**
11. **</exclusions>**
12. **</dependency>**
13. **</dependencies>**

## 属性

在pom.xml文件中我们可以使用${propertyName}的形式引用属性。这个propertyName有以下几种形式：

  env.propertyName：这种形式表示引用的是环境变量，比如我们需要引用当前系统的环境变量PATH的时候，就可以使用${env.PATH}。

  project.propertyName：这种形式表示引用的是当前这个pom.xml中project根元素下面的子元素的值。比如我们需要引用当前project下面的version的时候，就可以使用${project.version}。

  settings.propertyName：这种形式引用的是Maven本地配置文件settings.xml或本地Maven安装目录下的settings.xml文件根元素settings下的元素。比如我们需要引用settings下的本地仓库localRepository元素的值时，我们可以用${settings.localRepository}

  java的系统属性，所有在java中使用java.lang.System.getProperties()能够获取到的属性都可以在pom.xml中引用，比如${java.home}。

  pom.xml中properties元素下面的子元素作为属性。假如在pom.xml中有如下一段代码<properties><hello.world>helloWorld</hello.world></properties>，那么我们就可以使用${hello.world}引用到对应的helloWorld。

# Dependency介绍

## 依赖的传递性

当项目A依赖于B，而B又依赖于C的时候，自然的A会依赖于C，这样Maven在建立项目A的时候，会自动加载对C的依赖。

**依赖传递对版本的选择**

假设A依赖于B和C，然后B依赖于D，D又依赖于E1.0，C直接依赖于E2.0，那么这个时候A依赖的是E1.0还是E2.0，还是这两个都依赖呢？两个都依赖是肯定不行的，因为它们可能会有冲突的地方。这个时候就涉及到Maven中依赖传递对版本的选择问题。依赖传递在选择版本的时候首先是根据深度选择的。当一个项目同时经过不同的路径依赖于同一个组件时，会选择其深度最短的对应组件进行依赖。举例来说，假设A->B->C->D1.0，A->E->D2.0，那么这个时候A就会选择对D相对路径短的组件来进行依赖，也就是D2.0。那么当深度一样的时候Maven会如何选择呢？即A->B->D1.0和A->C->D2.0，这个时候Maven会如何选择A所依赖的D的版本呢？这种情况Maven会根据申明的依赖顺序来进行选择，先申明的会被作为依赖包。向前面这种情况，如果先申明对B的依赖，则A依赖的就是D1.0，如果先申明对C的依赖，则A依赖的就是D2.0。

**使用exclusion排除依赖**

假设有这样一种依赖关系，A->B->C，这个时候由于某些原因，我们不需要对C的依赖，但是我们又必须要对B的依赖，这个时候该怎么办呢？针对这种情况，Maven给我们提供了一个exclusion功能，我们可以在添加A对B的依赖时申明不需要引进B对C的依赖。具体做法如下：

**Xml代码  [收藏代码](javascript:void())**

1. **<dependencies>**
2. **<dependency>**
3. **<groupId>**groupB**</groupId>**
4. **<artifactId>**artifactB**</artifactId>**
5. **<version>**1.0**</version>**
6. **<exclusions>**
7. **<exclusion>**
8. **<groupId>**groupC**</groupId>**
9. **<artifactId>**artifactC**</artifactId>**
10. **</exclusion>**
11. **</exclusions>**
12. **</dependency>**
13. ...
14. **</dependencies>**

**可选的依赖项**

可选的依赖项表示可有可无，不一定需要的，它只是做一个标记。为了便于大家理解，我们先看这样一种情况，假设项目B依赖于项目C，这个时候我们把B对C的依赖利用optional标记为可选的，它意味着B中只有部分地方用到了C，并不是必须要的，当你依赖于B，但是又不需要使用到B的C功能时，可以不依赖于C。这样当A->B->C时，在建立项目A的时候将不会加入对C的依赖，因为C对B是可选的，我们不一定会用到C。但是在建立项目B的时候，Maven就会加入对C的依赖。也就是说这种标记为optional的依赖项对项目本身而言是没有什么影响的，它影响的是以该项目作为依赖项的其他项目，如这里的项目A。这种可选的依赖项有一个好处就是它会默认的作为exclusion项排除。

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<groupId>**groupB**</groupId>**
3. **<artifactId>**artifactB**</artifactId>**
4. **<version>**1.0**</version>**
5. **<dependencies>**
6. **<dependency>**
7. **<groupId>**groupC**</groupId>**
8. **<artifactId>**artifactC**</artifactId>**
9. **<version>**1.0**</version>**
10. **<optional>**true**</optional>**
11. **</dependency>**
12. ...
13. **</dependencies>**
14. **</project>**

**Xml代码  [收藏代码](javascript:void())**

2. **<project>**
3. **<groupId>**groupA**</groupId>**
4. **<artifactId>**artifactA**</artifactId>**
5. **<version>**1.0**</version>**
6. **<dependencies>**
7. **<dependency>**
8. **<groupId>**groupB**</groupId>**
9. **<artifactId>**artifactB**</artifactId>**
10. **<version>**1.0**</version>**
11. **</dependency>**
12. ...
13. **</dependencies>**
14. **</project>**

## 依赖项的作用域

在定义项目的依赖项的时候，我们可以通过scope来指定该依赖项的作用范围。scope的取值有compile、runtime、test、provided、system和import。

**compile**：这是依赖项的默认作用范围，即当没有指定依赖项的scope时默认使用compile。compile范围内的依赖项在所有情况下都是有效的，包括运行、测试和编译时。

**runtime**：表示该依赖项只有在运行时才是需要的，在编译的时候不需要。这种类型的依赖项将在运行和test的类路径下可以访问。

**test**：表示该依赖项只对测试时有用，包括测试代码的编译和运行，对于正常的项目运行是没有影响的。

**provided**：表示该依赖项将由JDK或者运行容器在运行时提供，也就是说由Maven提供的该依赖项我们只有在编译和测试时才会用到，而在运行时将由JDK或者运行容器提供。

**system**：当scope为system时，表示该依赖项是我们自己提供的，不需要Maven到仓库里面去找。指定scope为system需要与另一个属性元素systemPath一起使用，它表示该依赖项在当前系统的位置，使用的是绝对路径。比如官网给出了一个关于使用JDK的tools.jar的例子：

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. ...
3. **<dependencies>**
4. **<dependency>**
5. **<groupId>**sun.jdk**</groupId>**
6. **<artifactId>**tools**</artifactId>**
7. **<version>**1.5.0**</version>**
8. **<scope>**system**</scope>**
9. **<systemPath>**${java.home}/../lib/tools.jar**</systemPath>**
10. **</dependency>**
11. **</dependencies>**
12. ...
13. **</project>**

**import**：关于import的介绍，可以看后文中对引入依赖项的介绍。

## dependencyManagement介绍

dependencyManagement主要有两个作用，一个是集中管理项目的依赖项，另一个就是控制使用的依赖项的版本。

### 集中管理依赖项

看这样一种情况，我们有以下两个项目：

projectA

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<groupId>**groupA**</groupId>**
3. **<artifactId>**artifactA**</artifactId>**
4. **<version>**1.0**</version>**
5. **<dependencies>**
6. **<dependency>**
7. **<groupId>**groupC**</groupId>**
8. **<artifactId>**artifactC**</artifactId>**
9. **<version>**1.0**</version>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**groupD**</groupId>**
13. **<artifactId>**artifactD**</artifactId>**
14. **<version>**1.0**</version>**
15. **</dependency>**
16. **</dependencies>**
17. ...
18. **</project>**

       ProjectB

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<groupId>**groupB**</groupId>**
3. **<artifactId>**artifactB**</artifactId>**
4. **<version>**1.0**</version>**
5. **<dependencies>**
6. **<dependency>**
7. **<groupId>**groupC**</groupId>**
8. **<artifactId>**artifactC**</artifactId>**
9. **<version>**1.0**</version>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**groupE**</groupId>**
13. **<artifactId>**artifactE**</artifactId>**
14. **<version>**1.0**</version>**
15. **<type>**bar**</type>**
16. **</dependency>**
17. **</dependencies>**
18. ...
19. **</project>**

       从这两个项目中我们可以看出来，它们都依赖了artifactC，这样我们就可以创建一个新的项目，使用其dependencyManagement来统一管理这些依赖项。我们创建项目P来管理这些依赖项：

       projectP

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. ...
3. **<dependencyManagement>**
4. **<dependencies>**
5. **<dependency>**
6. **<groupId>**groupC**</groupId>**
7. **<artifactId>**artifactC**</artifactId>**
8. **<version>**1.0**</version>**
9. **</dependency>**
10. **<dependency>**
11. **<groupId>**groupD**</groupId>**
12. **<artifactId>**artifactD**</artifactId>**
13. **<version>**1.0**</version>**
14. **</dependency>**
15. **<dependency>**
16. **<groupId>**groupE**</groupId>**
17. **<artifactId>**artifactE**</artifactId>**
18. **<version>**1.0**</version>**
19. **<type>**bar**</type>**
20. **</dependency>**
21. **</dependencies>**
22. **</dependencyManagement>**
23. ...
24. **</project>**

       之后我们就可以这样来定义我们的projectA和projectB。

projectA

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<modelVersion>**4.0**</modelVersion>**
3. **<groupId>**groupA**</groupId>**
4. **<artifactId>**artifactA**</artifactId>**
5. **<version>**1.0**</version>**
6. **<dependencies>**
7. **<dependency>**
8. **<groupId>**groupC**</groupId>**
9. **<artifactId>**artifactC**</artifactId>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**groupD**</groupId>**
13. **<artifactId>**artifactD**</artifactId>**
14. **</dependency>**
15. **</dependencies>**
16. ...
17. **</project>**

       ProjectB

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<modelVersion>**4.0**</modelVersion>**
3. **<groupId>**groupB**</groupId>**
4. **<artifactId>**artifactB**</artifactId>**
5. **<version>**1.0**</version>**
6. **<dependencies>**
7. **<dependency>**
8. **<groupId>**groupC**</groupId>**
9. **<artifactId>**artifactC**</artifactId>**
10. **</dependency>**
11. **<dependency>**
12. **<groupId>**groupE**</groupId>**
13. **<artifactId>**artifactE**</artifactId>**
14. <!--因为artifactE的类型不是默认的jar，所以这里需要指定依赖项的类型-->
15. **<type>**bar**</type>**
16. **</dependency>**
17. **</dependencies>**
18. ...
19. **</project>**

从上面的定义我们可以看出，我们已经简化了projectA和projectB依赖项的定义，我们可以只在projectA和projectB中申明需要使用的依赖项，而不必指定其对应的版本和作用范围等信息，当指定了这些信息时子项目中的定义会覆盖父项目的dependencyManagement中的定义，否则将使用dependencyManagement中的定义。这样也可以很方便我们对依赖项的版本进行统一管理。

### 控制依赖项使用的版本

看下面这样一种情况

projectA

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<modelVersion>**4.0**</modelVersion>**
3. **<groupId>**groupA**</groupId>**
4. **<artifactId>**artifactA**</artifactId>**
5. **<version>**1.0**</version>**
6. **<packaging>**pom**</packaging>**
7. **<dependencyManagement>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**groupA**</groupId>**
11. **<artifactId>**A**</artifactId>**
12. **<version>**1.5**</version>**
13. **</dependency>**
14. **<dependency>**
15. **<groupId>**groupA**</groupId>**
16. **<artifactId>**B**</artifactId>**
17. **<version>**1.0**</version>**
18. **</dependency>**
19. **<dependency>**
20. **<groupId>**groupA**</groupId>**
21. **<artifactId>**C**</artifactId>**
22. **<version>**1.0**</version>**
23. **</dependency>**
24. **<dependency>**
25. **<groupId>**groupA**</groupId>**
26. **<artifactId>**D**</artifactId>**
27. **<version>**1.6**</version>**
28. **</dependency>**
29. **</dependencies>**
30. **</dependencyManagement>**
31. ...
32. **</project>**

       ProjectB

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. **<modelVersion>**4.0**</modelVersion>**
3. **<groupId>**groupB**</groupId>**
4. **<artifactId>**artifactB**</artifactId>**
5. **<version>**1.0**</version>**
6. **<packaging>**pom**</packaging>**
7. **<dependencyManagement>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**groupA**</groupId>**
11. **<artifactId>**A**</artifactId>**
12. **<version>**1.0**</version>**
13. **</dependency>**
14. **</dependencies>**
15. **</dependencyManagement>**
16. **<dependencies>**
17. **<dependency>**
18. **<groupId>**groupA**</groupId>**
19. **<artifactId>**B**</artifactId>**
20. **<version>**1.8**</version>**
21. **</dependency>**
22. **<dependency>**
23. **<groupId>**groupA**</groupId>**
24. **<artifactId>**C**</artifactId>**
25. **<scope>**runtime**</scope>**
26. **</dependency>**
27. **</dependencies>**
28. ...
29. **</project>**

       这样，当我们对projectB进行Maven操作的时候，依赖项A、B、C和D将会如下选择：

       当A是B或者C的一个依赖项的时候，projectB将使用的是A的1.0版本，因为A是申明在projectB的dependencyManagement中的，此外申明在dependencyManagement中的依赖项将比项目的依赖项的依赖项有更高的优先级，还有就是当前项目的申明将比其父项目的申明具有更高的优先级。

       依赖项B将使用版本1.8，因为依赖项B是直接申明在projectB中的，而且指定了版本为1.8，所以当前项目申明的版本会覆盖父项目的dependencyManagement中指定的版本。

       依赖项C将使用版本1.0，但是其作用范围将会是runtime。因为依赖项C也是直接申明在projectB中的，而且其没有指定自己所使用的版本，所以将使用其父项目的dependencyManagement中指定的版本。此外，其指定了自己所作用的范围是runtime，所以它会覆盖其父项目的dependencyManagement中所指定依赖项的默认的compile作用范围。

       依赖项D将使用版本1.6。当D是B或者C的一个依赖项的时候，projectB将使用D的1.6版本，因为项目D是申明在dependencyManagement中的，而且在dependencyManagement中申明的依赖项将比间接的依赖项具有更高的优先级。

## 引入依赖项

我们知道通过继承一个项目我们可以在子项目中很好的申明需要使用的父项目的dependencyManagement定义的依赖项。但是因为每个项目都只能申明唯一的一个父项目，这在某些时候就会限制我们的项目建立。为此Maven为我们提供了一种方法，那就是通过设定依赖项的scope为import。注意这种方式只有在Maven2.0.9以上的版本才能使用。

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. ...
3. **<groupId>**groupA**</groupId>**
4. **<artifactId>**artifactA**</artifactId>**
5. **<version>**1.0**</version>**
6. **<packaging>**pom**</packaging>**
7. **<dependencyManagement>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**test**</groupId>**
11. **<artifactId>**A**</artifactId>**
12. **<version>**1.0**</version>**
13. **</dependency>**
14. **<dependency>**
15. **<groupId>**test**</groupId>**
16. **<artifactId>**B**</artifactId>**
17. **<version>**1.1**</version>**
18. **</dependency>**
19. **<dependency>**
20. **<groupId>**test**</groupId>**
21. **<artifactId>**C**</artifactId>**
22. **<version>**1.2**</version>**
23. **</dependency>**
24. **<dependency>**
25. **<groupId>**test**</groupId>**
26. **<artifactId>**D**</artifactId>**
27. **<version>**1.3**</version>**
28. **</dependency>**
29. **</dependencies>**
30. **</dependencyManagement>**
31. ...
32. **</project>**

上面这个项目是用来统一管理项目的依赖项的。那么按照之前的那种继承机制，这个时候我们的项目artifactB是如下这样定义的：

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. ...
3. **<parent>**
4. **<groupId>**groupA**</groupId>**
5. **<artifactId>**artifactA**</artifactId>**
6. **<version>**1.0**</version>**
7. **</parent>**
8. **<groupId>**groupA**</groupId>**
9. **<artifactId>**artifactB**</artifactId>**
10. **<version>**1.0**</version>**
11. **<packaging>**pom**</packaging>**
12. **<dependencies>**
13. **<dependency>**
14. **<groupId>**test**</groupId>**
15. **<artifactId>**A**</artifactId>**
16. **</dependency>**
17. **<dependency>**
18. **<groupId>**test**</groupId>**
19. **<artifactId>**D**</artifactId>**
20. **<scope>**runtime**</scope>**
21. **</dependency>**
22. **</dependencies>**
23. ...
24. **</project>**

       那么如果按照import这种形式的话，artifactB可以如下定义：

**Xml代码  [收藏代码](javascript:void())**

1. **<project>**
2. ...
3. **<groupId>**groupA**</groupId>**
4. **<artifactId>**artifactB**</artifactId>**
5. **<version>**1.0**</version>**
6. **<packaging>**pom**</packaging>**
7. **<dependencyManagement>**
8. **<dependencies>**
9. **<dependency>**
10. **<groupId>**groupA**</groupId>**
11. **<artifactId>**artifactA**</artifactId>**
12. **<version>**1.0**</version>**
13. **<type>**pom**</type>**
14. **<scope>**import**</scope>**
15. **</dependency>**
16. **</dependencies>**
17. **</dependencyManagement>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**test**</groupId>**
21. **<artifactId>**A**</artifactId>**
22. **</dependency>**
23. **<dependency>**
24. **<groupId>**test**</groupId>**
25. **<artifactId>**D**</artifactId>**
26. **<scope>**runtime**</scope>**
27. **</dependency>**
28. **</dependencies>**
29. ...
31. **</project>**

# Maven整合Eclipse

   Maven整合Eclipse就是在Eclipse中安装Maven的插件，这样我们就可以直接在Eclipse上使用Maven，而不需要去命令窗口敲Maven命令了。有两个Maven插件我们可以用来整合Eclipse，实际应用中安装其中一个就可以了。

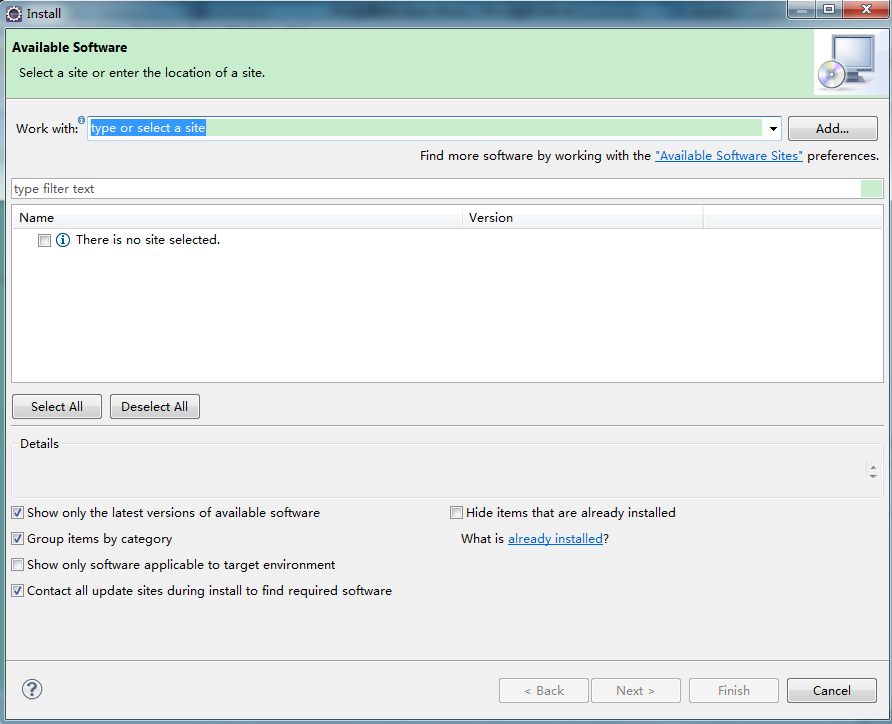
## Eclipse m2e

   给Eclipse安装m2e插件的好处：

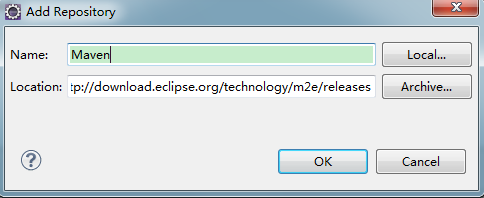
* 可以在Eclipse中使用Maven
* Eclipse中build path依赖管理是基于Maven的pom.xml的。
* 可以从Eclipse的工作空间解决Maven的依赖关系，而不需要把它们安装到Maven的本地仓库中。
* 自动从Maven的远程仓库中下载所需要的依赖包
* 可以使用向导来创建Maven项目
* 可以在Maven的远程仓库中快速搜索依赖包

   可以从地址<http://download.eclipse.org/technology/m2e/releases>在线的给Eclipse安装最新的m2e插件，步骤如下：

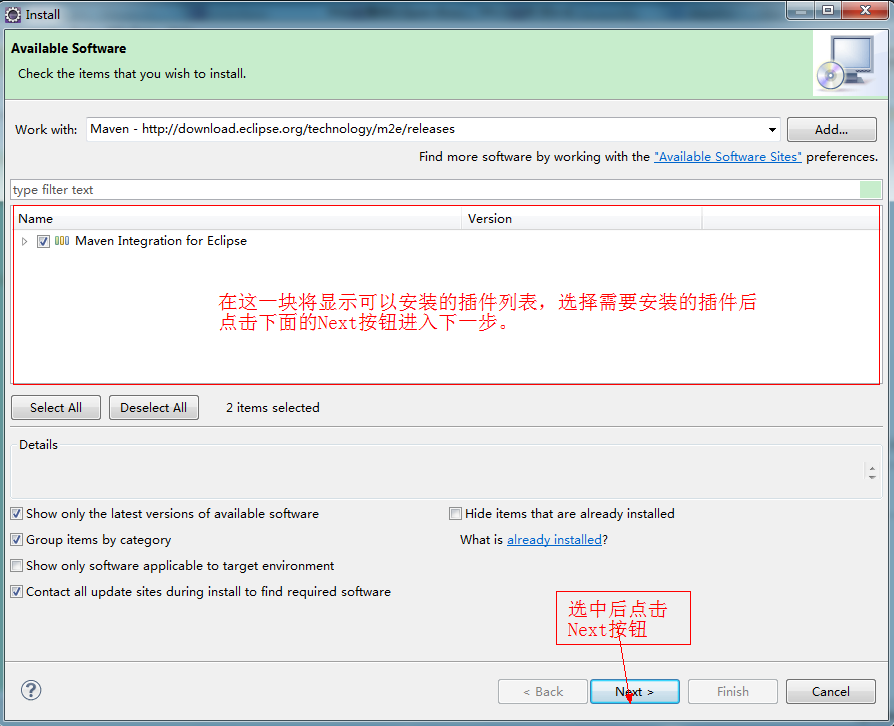
       第一步：选择help菜单下面的install new software菜单项，出现如下界面。



第二步：点击add按钮，

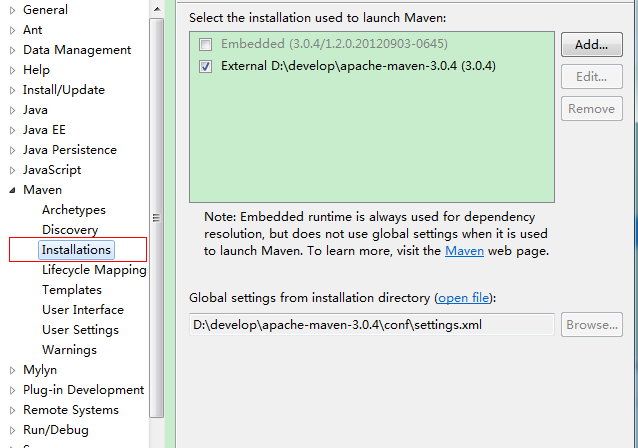


在弹出的对话框中输入插件对应的名称和地址，这里我们可以输入Maven和http://download.eclipse.org/technology/m2e/releases。输入完成后点击OK。这个时候就会把对应地址可以安装的插件都列出来，如下图所示

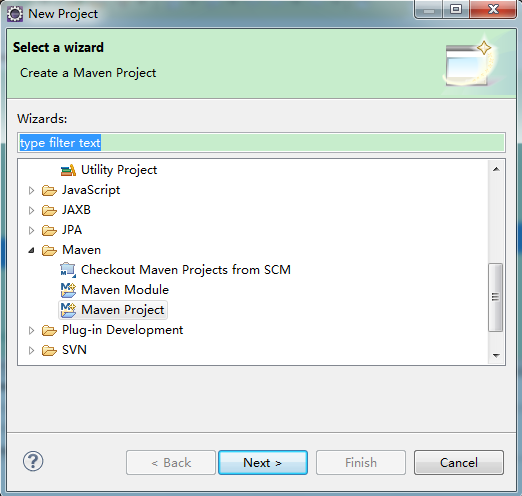


之后可以根据提示来进行安装。安装完成后需要重新启动Eclipse。

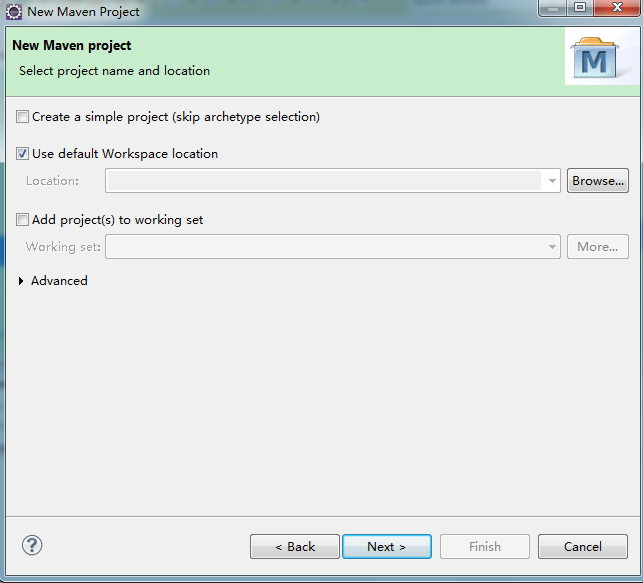
       安装完成重启之后我们可以看到菜单window->Preferences点击进入后会多出一栏Maven的菜单栏。我们可以点开该Maven菜单栏，然后找到对应的installations菜单点击，在出现的右边窗口中我们可以选择add按钮添加我们自己的Maven安装目录。



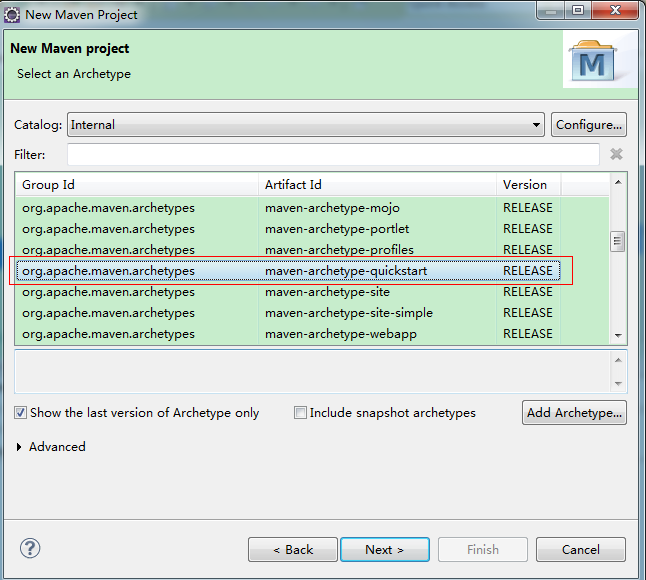
       Maven安装完成后我们就可以在Eclipse中新建自己的Maven项目了。我们可以在Eclipse中选择new一个project，在出现的对话框中选择Maven目录下的Maven Project。



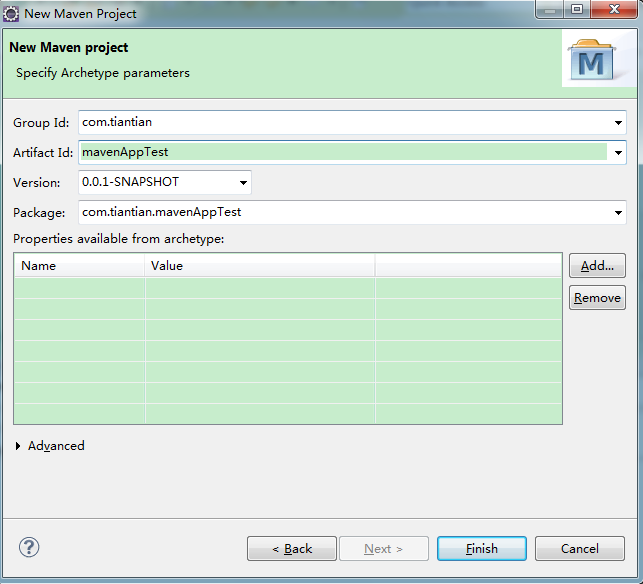
点击下一步：



点击下一步，进入新建Maven项目使用的archetype的选择页面，默认将选择maven-archetype-quickstart建立一个java project。



点击Next进入下一步，在这一步中将输入建立Maven项目核心的groupId、artifactId和version。



## Eclipse IAM

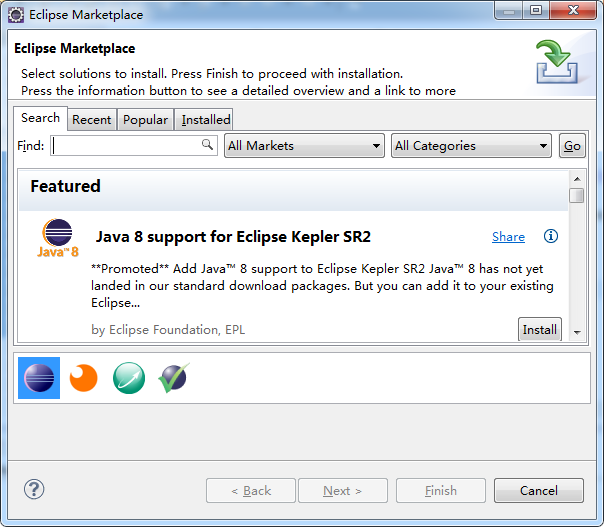
       安装Eclipse IAM插件的URL为http://q4e.googlecode.com/svn/trunk/updatesite-iam/。安装方法跟Eclipse m2e类似。这里就不再过多的赘述。

## 通过Eclipse Marketplace安装Maven插件

       此外，我们也可以通过Eclipse Marketplace来安装Maven插件。具体步骤如下：

       1、选择Eclipse菜单栏的help菜单。

       2、点击Eclipse Marketplace菜单项，点击之后会出现如下界面。



       3、之后在搜索框里面搜索Maven，会出现一系列的与Maven有关的插件，选择对应的插件进行安装即可