Maven使用手册

# Maven简介

## 什么是Maven

Maven是一个跨平台的项目管理工具

主要负责Java项目的构建、依赖管理和项目信息管理

## 为什么需要Maven

Maven能帮助我们很容易的进行项目构建，不依赖于IDE等

Maven有很好的依赖管理

利用Maven进行持续集成

# 下载和安装

|  |
| --- |
| #第一步：下载，http://maven.apache.org/download.html  #第二步：安装，解压下载的zip包  #第三步：环境变量的设置  M2\_HOME: Maven的安装目录 （D:\maven）  M2: %M2\_HOME%\bin  将M2添加到环境变量path中: path:%M2%  第四步：进行测试mvn –v，会出现如下信息： |

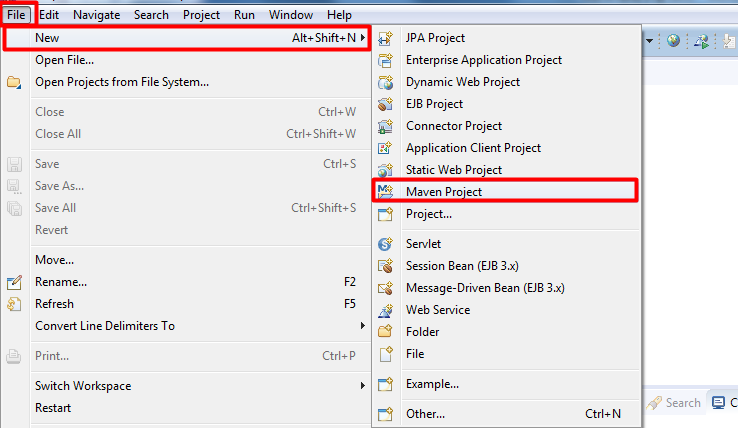
# 构建Maven项目

## Maven 项目结构

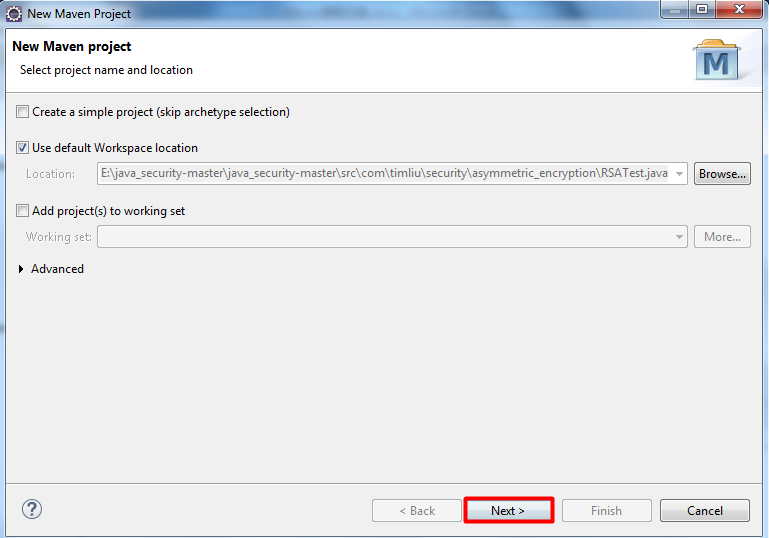
|  |
| --- |
| app  |-- pom.xml  |-- src  |-- main  | |-- java  | |-- resources  | |-- webapp  |-- test  |-- java  |-- resources |

## Eclipse中新建Maven webapp

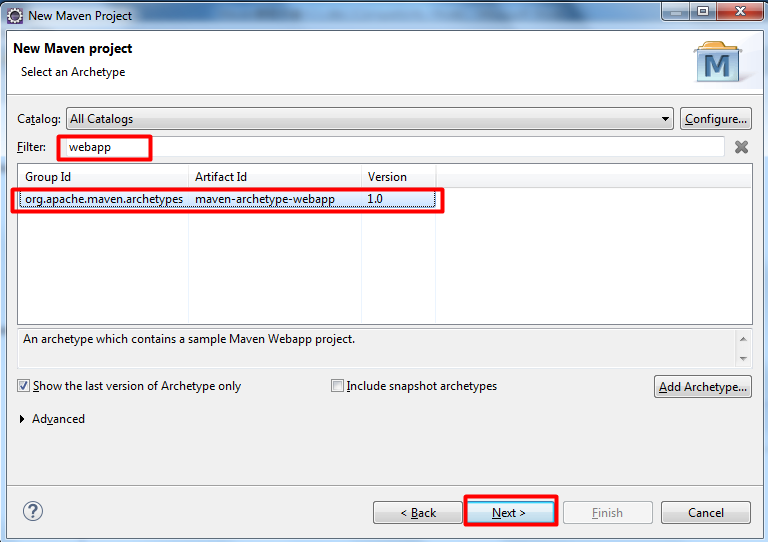
第一步：File->new project，点击下一步



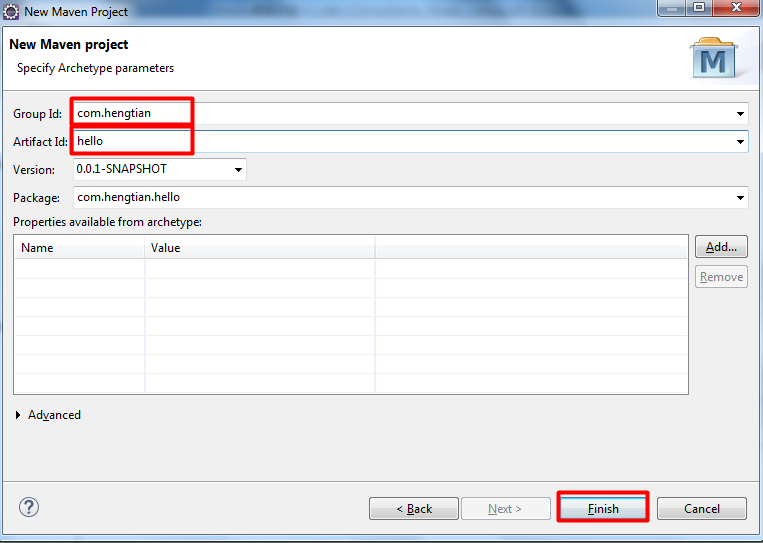
第二步：输入next



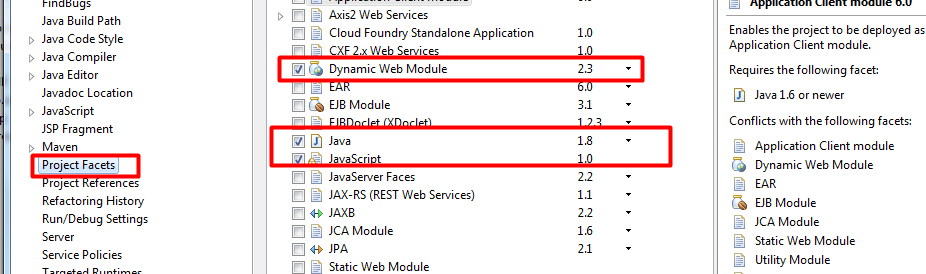
第三步：在filter中输入webapp搜索，并选中红色框中标注，点击下一步



第四步：填写项目坐标，并点击完成



第五步：设置jre和facets



# 生命周期

Maven拥有三套相互独立的生命周期，它们分别为clean，default和site。 每个生命周期包含一些阶段，这些阶段是有顺序的，并且后面的阶段依赖于前面的阶段，用户和Maven最直接的交互方式就是调用这些生命周期阶段。 以clean生命周期为例，包含的阶段有pre-clean, clean 和 post clean。当用户调用pre-clean的时候，只有pre-clean得以执行，当用户调用clean的时候，pre-clean和clean阶段会得以顺序执行；当用户调用post-clean的时候，pre-clean,clean,post-clean会得以顺序执行。

较之于生命周期阶段的前后依赖关系，三套生命周期本身是相互独立的，用户可以仅仅调用clean生命周期的某个阶段，或者仅仅调用default生命周期的某个阶段，而不会对其他生命周期产生任何影响。

## clean 生命周期

clean生命周期的目的是清理项目，它包含三个阶段：

* pre-clean 执行一些清理前需要完成的工作。
* clean 清理上一次构建生成的文件。
* post-clean 执行一些清理后需要完成的工作。

## default 生命周期

default生命周期定义了真正构件时所需要执行的所有步骤，它是生命周期中最核心的部分，它包含的阶段如下：

* validate 验证项目是否正确和所有需要的相关资源是否可用
* initialize 初始化构建
* generate-sources
* process-sources 处理源代码
* generate-resources
* process-resources 处理项目主资源文件。对src/main/resources目录的内容进行变量替换等工作后，复制到项目输出的主classpath目录中。
* compile 编译项目的主源代码
* process-classes
* generate-test-sources
* process-test-sources 处理项目测试资源文件
* generate-test-resources
* process-test-resources 处理测试的资源文件
* test-compile 编译项目的测试代码
* process-test-classes
* test 使用单元测试框架运行测试，测试代码不会被打包或部署
* prepare-package 做好打包的准备
* package 接受编译好的代码，打包成可发布的格式
* pre-integration-test
* integration-test
* post integration-test
* verify
* install 将包安装到Maven本地仓库，供本地其他Maven项目使用
* deploy 将最终的包复制到远程仓库，供其他Maven项目使用

## site 生命周期

site生命周期的目的是建立和发布项目站点，Maven能够基于POM所包含的信息，自动生成一个友好的站点，方便团队交流和发布项目信息。该生命周期包含如下阶段：

* pre-site 执行一些在生成项目站点之前需要完成的工作
* site 生成项目站点文档
* post-site 执行一些在生成项目站点之后需要完成的工作
* site-deploy 将生成的项目站点发布到服务器上

# 坐标

在我们开发Maven项目的时候，需要为其定义适当的坐标，这是Maven强制要求的。在这个基础上，其他Maven项目才能应用该项目生成的构件。

Maven坐标为各种构件引入了秩序，任何一个构件都必须明确定义自己的坐标，而一组Maven坐标是通过一些元素定义的，它们是groupId,artifactId,version,packaging,class-sifer。下面是一组坐标定义：

|  |
| --- |
| <groupId>com.mycompany.app</groupId>  <artifactId>my-app</artifactId>  <packaging>jar</packaging>  <version>0.0.1-SNAPSHOT</version> |

下面讲解一下各个坐标元素：

**groupId** ：定义当前Maven项目隶属的实际项目。首先，Maven项目和实际项目不一定是一对一的关系。比如SpringFrameWork这一实际项目，其对应的Maven项目会有很多，如spring-core,spring-context等。这是由于Maven中模块的概念，因此，一个实际项目往往会被划分成很多模块。其次，groupId不应该对应项目隶属的组织或公司。原因很简单，一个组织下会有很多实际项目，如果groupId只定义到组织级别，而后面我们会看到，artifact  
Id只能定义Maven项目（模块），那么实际项目这个层次将难以定义。最后，groupId的表示方式与Java包名的表达方式类似，通常与域名反向一一对应。

**artifactId** : 该元素定义当前实际项目中的一个Maven项目（模块），推荐的做法是使用实际项目名称作为artifactId的前缀。比如上例中的my-app。

**version** : 该元素定义Maven项目当前的版本 （0.0.1SNAPSHOT）

**packaging** ：定义Maven项目打包的方式，首先，打包方式通常与所生成构件的文件扩展名对应，如上例中的packaging为jar,最终的文件名为my-app-0.0.1-SNAPSHOT.jar。也可以打包成war, ear等。当不定义packaging的时候，Maven 会使用默认值jar

**classifier**: 该元素用来帮助定义构建输出的一些附件。附属构件与主构件对应，如上例中的主构件为my-app-0.0.1-SNAPSHOT.jar,该项目可能还会通过一些插件生成如my-app-0.0.1-SNAPSHOT-javadoc.jar,my-app-0.0.1-SNAPSHOT-sources.jar, 这样附属构件也就拥有了自己唯一的坐标

# 依赖

## 依赖配置

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.6</version>  <scope>compile</scope>  </dependency>  </dependencies> |

groupId, artifactId, version: 依赖的基本坐标

type:依赖的类型，对应于项目坐标定义的packaging。大部分情况下，该元素不必声明，其默认值为jar。

scope: 依赖范围(**test--测试程序有效，不打包 ；compile--测试和主程序都有效，打包；provided --测试和主程序都有效，不打包 如：javax.servlet-api。**)

Optional: 标记依赖是否可选

Exclusions:排除传递性依赖

## 依赖范围(scope)

Compile: 编译，测试，运行都有效

Test：只对测试有效

Provided: 已提供依赖范围，容器提供

Runtime: 运行时有效

System: 系统依赖范围

## 分类器

GAV是Maven坐标最基本最重要的组成部分，但GAV不是全部。还有一个元素叫做分类器（classifier），90%的情况你不会用到它，但有些时候，分类器非常不可或缺。举个简单的例子，当我们需要依赖TestNG的时候，简单的声明GAV会出错，因为TestNG强制需要你提供分类器，以区别jdk14和jdk15，我们需要这样声明对TestNG的依赖：

|  |
| --- |
| <dependency>  <groupId>org.testng</groupId>  <artifactId>testng</artifactId>  <version>5.7</version>  <classifier>jdk15</classifier>  </dependency> |

你会注意到maven下载了一个名为testng-5.7-jdk15.jar的文件。其命名模式实际上是<artifactId>-<version>-<classifier>.<packaging>。理解了这个模式以后，你就会发现很多文件其实都是默认构件的分类器扩展，如 myapp-1.0-test.jar, myapp-1.0-sources.jar。

分类器还有一个非常有用的用途是：我们可以用它来声明对test构件的依赖，比如，我们在一个核心模块的src/test/java中声明了一些基础类，然后我们发现这些测试基础类对于很多其它模块的测试类都有用。没有分类器，我们是没有办法去依赖src/test/java中的内容的，因为这些内容不会被打包到主构件中，它们单独的被打包成一个模式为<artifactId>-<version>-test.jar的文件。我们可以使用分类器来依赖这样的test构件：

|  |
| --- |
| <dependency>  <groupId>org.myorg.myapp</groupId>  <artifactId>core</artifactId>  <version>${project.version}</version>  <classifier>test</classifier>  </dependency>  Ps:理解了分类器，那么可供依赖的资源就变得更加丰富。 |

## 依赖归类

如果我们项目中用到很多关于Spring Framework的依赖，它们分别是org.springframework:spring-core:2.5.6, org.springframework:spring-beans:2.5.6,org.springframework:spring-context:2.5.6,它们都是来自同一项目的不同模块。因此，所有这些依赖的版本都是相同的，而且可以预见，如果将来需要升级Spring Framework，这些依赖的版本会一起升级。因此，我们应该在一个唯一的地方定义版本，并且在dependency声明引用这一版本，这一在Spring Framework升级的时候只需要修改一处即可。

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>    <groupId>com.mycompany.app</groupId>  <artifactId>my-app-simple</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>my-app-simple</name>  <properties>  **<springframework.version>4.0.0</springframework.version>**  </properties>    <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  **<version>${springframework.version}</version>**  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  **<version>${springframework.version}</version>**  </dependency>  </dependencies>  </project> |

## 依赖传递

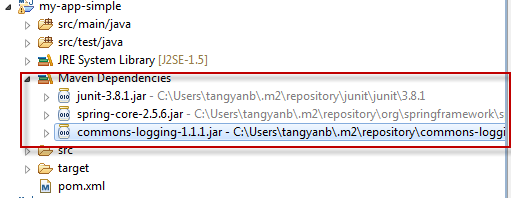
下面我们看一个简单的项目，读者可从附件中下载源码，POM.xml配置如下：

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>    <groupId>com.mycompany.app</groupId>  <artifactId>my-app-simple</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>    <name>my-app-simple</name>  <url>http://maven.apache.org</url>    <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>    <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency>    <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>2.5.6</version>  </dependency>  </dependencies>  </project> |

我们可以看到此项目引入依赖junit和spring-core，我们可以在Maven仓库中查找spring-core构件，POM中我们会看到该文件包含了一个commons-logging依赖：

|  |
| --- |
| <dependency>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  <version>1.1.1</version>  </dependency> |

那么该依赖会传递到当前项目中，这就是依赖的传递性，打开项目查看Maven dependencies:



## 排除间接依赖

当我们引入第三方jar包的时候，难免会引入传递性依赖，有些时候这是好事，然而有些时候我们不需要其中的一些传递性依赖

比如上例中的项目，我们不想引入传递性依赖commons-logging，我们可以使用exclusions元素声明排除依赖，exclusions可以包含一个或者多个exclusion子元素，因此可以排除一个或者多个传递性依赖。需要注意的是，声明exclusions的时候只需要groupId和artifactId，而不需要version元素，这是因为只需要groupId和artifactId就能唯一定位依赖图中的某个依赖。换句话说，Maven解析后的依赖中，不可能出现groupId和artifactId相同，但是version不同的两个依赖。

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>2.5.6</version>  <exclusions>  <exclusion>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

## 处理依赖冲突

使用Maven时是否遇到过诸如"NoSuchMethodError"或者"ClassNot  
FoundException"之类的问题，甚至发生这些问题的Java类你没都没有听说过。要搞清楚这里面的缘由，我们得学习Maven对依赖冲突的处理机制。Maven采用"最近获胜策略（nearest wins strategy）"的方式处理依赖冲突，即如果一个项目最终依赖于相同artifact的多个版本，在依赖树中离项目最近的那个版本将被使用。**如果依赖路径长度相同，先声明dependency的依赖的版本会被使用。**

对于这种有依赖冲突所导致的问题，我们有两种解决方法：

* 显式加入对冲突jar包依赖
* 使用  <exclusion>将比较老的jar包依赖排除掉

# 聚合和继承

## 聚合

聚合项目的目的是一条Maven命令可以构建多个模块，packaging为pom

Modules中定义模块，module中为子模块的相对路径

|  |
| --- |
| **#在父工程配置聚合工程**  <project>  <modules>  <module>../child1 </module>  <module>../child2</module>  </modules>  </project> |

## 继承

|  |
| --- |
| #父工程  <groupId>com.ixo</groupId>  <artifactId>manage</artifactId>  <version>0.0.1-SNAPSHOT</version>  **<packaging>pom</packaging>**  #commom子工程声明父工程  <parent>  <groupId> com.ixo </groupId>  <artifactId> manage </artifactId>  <version>0.0.1-SNAPSHOT</version>  **<!-- 相对路径-->**  **<relativePath>../parent-project/pom.xml</relativePath>**  </parent> |

# Setting文件

settings.xml对于maven来说相当于全局性的配置，用于所有的项目，当Maven运行过程中的各种配置，例如pom.xml，不想绑定到一个固定的project或者要分配给用户时，我们使用settings.xml中的settings元素来确定这些配置。这包含了本地仓库位置，远程仓库服务器以及认证信息等。

settings.xml存在于两个地方：

1.安装的地方：$M2\_HOME/conf/settings.xml（全局配置）

2.用户的目录：${user.home}/.m2/settings.xml（用户配置，优先级更高）

下面是settings下的顶层元素的一个概览：

|  |
| --- |
| <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0  http://maven.apache.org/xsd/settings-1.0.0.xsd">  <localRepository/>  <interactiveMode/>  <usePluginRegistry/>  <offline/>  <pluginGroups/>  <servers/>  <mirrors/>  <proxies/>  <profiles/>  <activeProfiles/>  </settings> |

## 简单值

**localRepository**：这个值是构建系统的本地仓库的路径。默认的值是${user.home}/.m2/repository.如果一个系统想让所有登陆的用户都用同一个本地仓库的话，这个值是极其有用的。

**interactiveMode**：如果Maven要试图与用户交互来得到输入就设置为true，否则就设置为false，默认为true。

usePluginRegistry：如果Maven使用${user.home}/.m2/plugin-registry.xml来管理plugin的版本，就设置为true，默认为false。

**offline**：如果构建系统要在离线模式下工作，设置为true，默认为false。如果构建服务器因为网络故障或者安全问题不能与远程仓库相连，那么这个设置是非常有用的。

## PluginGroups（插件组）

这个元素包含了一系列pluginGroup元素，每个又包含了一个groupId。当一个plugin被使用，而它的groupId没有被提供的时候，这个列表将被搜索。这个列表自动的包含了org.apache.maven.plugins和org.codehaus.mojo。

|  |
| --- |
| <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0  http://maven.apache.org/xsd/settings-1.0.0.xsd">  ...  <pluginGroups>  <pluginGroup>org.mortbay.jetty</pluginGroup>  </pluginGroups>  ...  </settings> |

## Servers（服务器）

第一：定义jar包下载的Maven仓库

第二： 定义部署服务器

|  |
| --- |
| <servers>  <server>  <id>tomcat</id>  <username>bruce</username>  <password>password</password>  </server>  <server>  <id>shiyue</id>  <username>admin</username>  <password>password</password>  </server>  </servers>  tomcat: 部署服务器  shiyue: Mave私服 |

## Mirrors（镜像）

指定仓库的地址，则默认从指定的镜像下载jar包及插件

|  |
| --- |
| <mirrors>  <mirror>  <id>mirrorId</id>  <mirrorOf>\*</mirrorOf>  <name>Human Readable Name for this Mirror.</name>  <url>http://host:port/nexus-2.1.2/content/groups/public</url>  </mirror>  <!-- aliyun镜像-->  <mirror>  <id>alimaven</id>  <name>aliyun maven</name>  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>  <mirrorOf>central</mirrorOf>  </mirror>  </mirrors> |

## Proxies（代理）

有时候你所在的公司基于安全因素考虑，要求你使用通过安全认证的代理访问因特网。这时就需要为Maven配置HTTP代理。

|  |
| --- |
| <proxies>  <proxy>  <id>optional</id>  <active>true</active>  <protocol>http</protocol>  <username>proxyuser</username>  <password>proxypass</password>  <host>proxy.host.net</host>  <port>80</port>  <nonProxyHosts>local.net|some.host.com</nonProxyHosts>  </proxy>  </proxies> |

# Maven插件

## 编译

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

## 测试

|  |
| --- |
| <!-- 测试插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <configuration>  <!-- true:跳过测试 -->  <skip>true</skip>  <!-- 解决测试乱码问题 -->  <forkMode>once</forkMode>  <argLine>-Dfile.encoding=UTF-8</argLine>  </configuration>  </plugin> |

## 打包

|  |
| --- |
| <!-- war包插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-war-plugin</artifactId>  <version>2.3</version>  </plugin> |

## 打包源码

|  |
| --- |
| <!-- 源码插件 -->  <plugin>  <artifactId>maven-source-plugin</artifactId>  <version>2.1</version>  <configuration>  <attach>true</attach>  </configuration>  <executions>  <execution>  <id>attach-source</id>  <goals>  <goal>jar</goal>  </goals>  </execution>  </executions>  </plugin> |

# Maven 仓库

Maven仓库分为**本地仓库**和**远程仓库**，集中存放项目引用的jar包，无需将jar包放在程序中，结合Maven项目的pom.xml，使得项目管理jar包更容易，有以下几个优点：

* 对于项目来说，无需关心jar包存储问题，只管理好配置即可
* 易于搜索和使用jar包，不需要到网上到处找，仓库提供了几乎所有的jar包资源，并提供专业的搜索引擎，我们很容易就能找到我们所需的jar，并获取它的坐标信息，在Maven项目中增加相应的依赖即可，简单快捷
* 易于管理jar包版本，pom.xml中可以一目了然的看到jar包的版本，且可以通过相关配置来约束项目使用的jar包的版本
* 易于发现并排除jar包的依赖冲突，在传统方式管理jar包的情况下，如果出现多个jar包的依赖出现冲突的时候（如ssh相关的jar包），很难排除冲突和找到互相适应的版本，而使用Maven管理，这些将变的非常简单
* 管理自定义的jar包:有些jar包是我们自己开发的，我们也可以将其放在仓库里，供其他项目使用，Maven仓库使jar包的发布和管理变得简单和有效，且能有效的控制版本变更

## 本地仓库

本地仓库是远程仓库的一个缓冲和子集，当你构建Maven项目的时候，首先会从本地仓库查找资源，如果没有，那么Maven会从远程仓库下载到本地仓库，这样在下次使用的时候就不需要下载了。Maven缺省的本地仓库地址为${user.home}/.m2/repository，也可以在settings.xml文件中修改该地址

## 远程仓库

远程仓库：是指部署在远程的仓库，默认是Maven的中央仓库，也可以是Nexus仓库服务器，远程仓库拥有大量的jar包资源，运用Nexus可以在局域网搭建企业级的Maven仓库

## 仓库的配置

i) Settings.xml中配置远程仓库

|  |
| --- |
| <repositories>  <repository>  <id>jdk14</id>  <name>Repository for JDK 1.4 builds</name>  <url>http://www.myhost.com/maven/jdk14</url>  <layout>default</layout>  <snapshotPolicy>always</snapshotPolicy>  </repository>  </repositories> |

ii) 使用镜像，覆盖中央仓库的地址

有时候我们并不想从中央仓库下载，而是想直接从本地服务器仓库下，配置如下：

|  |
| --- |
| <mirrors>  <mirror>  <id>mirrorId</id>  <mirrorOf>\*</mirrorOf>  <name>Human Readable Name for this Mirror.</name>  <url>http://demo.shiyuesoft.com:8888/nexus-2.1.2/content/groups/public</url>  </mirror>  </mirrors> |

iii) 在Maven项目中pom.xml配置

|  |
| --- |
| <!-- jar包仓库 -->  <repositories>  <repository>  <id>nexus</id>  <url>http://demo.shiyuesoft.com:8888/nexus-2.1.2/content/groups/public</url>  <releases>  <enabled>true</enabled>  </releases>  <snapshots>  <enabled>true</enabled>  </snapshots>  </repository>  </repositories>  <!-- 插件仓库 -->  <pluginRepositories>  <pluginRepository>  <id>nexus</id>  <url>http://demo.shiyuesoft.com:8888/nexus-2.1.2/content/groups/public</url>  <releases>  <enabled>true</enabled>  </releases>  <snapshots>  <enabled>true</enabled>  </snapshots>  </pluginRepository>  </pluginRepositories> |

# 私服Nexus

Nexus是Maven仓库管理器，用来搭建一个本地仓库服务器，这样做的好处是便于管理，节省网络资源，速度快，还有一个非常有用的功能就是可以通过项目的SNAPSHOT版本管理，来进行模块间的高效依赖开发。虽然可以通过中央仓库来获取我们所需要的jar包，但是现实往往是存在很多问题：

* 网速慢，我们可能需要花很长的时间来下载所需要的jar
* 如果我们的公司很大，有几百甚至几千人在用Maven，那么这些人都去通过中央仓库来获取jar，那么这是一个很大的资源浪费
* 如果存在模块之间的依赖开发，我们的伙伴就不能很方便的获取快照版本。
* 在实际开发过程中，有些jar的版本可能在中央仓库里面不存在，或者更新不及时，我们是获取不到这个jar的。

所有以上问题，可以通过Nexus解决。这个仓库是本地的，下载的速度是从远程下载不可比的。它可以为我们公司的所有Maven使用者服务，进行统一管理；它能很好的帮我们进行存在模块依赖的项目的开发；可以添加自己的第三方包

## 下载与安装

|  |
| --- |
| 下载地址：http://www.sonatype.org/nexus/go  执行命令：  cd C:\nexus\nexus-2.0.4\bin\jsw\windows-x86-64\  安装：install-nexus.bat  启动： start-nexus.bat  #解压安装包  tar –zxvf nexus-3.0.0-03-unix.tar.gz  #修改Jdk路径  Nexus3.0.0的版本需要JDK1.8的版本  cd /data/nexus-3.0.0-03/bin  vi nexus  #添加JDK配置  INSTALL4J\_JAVA\_HOME\_OVERRIDE= /usr/java/jdk1.8.0\_65  # 修改使用的用户  vi nexus.rc  #添加配置  run\_as\_user=root  #修改配置文件  vi /home/nexus/nexus-3.0.1-01/etc/org.sonatype.nexus.cfg    #启动nexus服务  ./ nexus start |

## 登录和修改密码

访问：http://localhost:8081

点击右上角登录，用户名/密码：admin/admin123

按如下图所示修改密码

## 搜索jar和仓库介绍

可以在搜索栏搜索想要的jar包和pom，点击repositories可以看到当前的仓库以Maven中心仓库为例，下图是仓库信息的简介

## 自定义仓库

如下图点击add->hosted Repository

输入Repository id和Repository name然后点击save即可

成功后将会在仓库列表中显示

## 上传jar到仓库

上一步新建了仓库名为test，点击test，在Artifact Upload中填写内容如下图，并上传jar包，点击提交即可

## 组合仓库

在仓库列表中有一个名叫Public Repositories的仓库，该仓库是由多个仓库组合而成，客户端将会默认使用Public 仓库里面的jar包，这个Public 仓库是可以进行配置的，如下图，将右边的仓库移到左边，则该仓库将变为可用的。

## 代理第三方仓库

添加Proxy仓库可以代理第三方的仓库，以便扩大私服的jar包索引

# 使用maven profile实现多环境构建

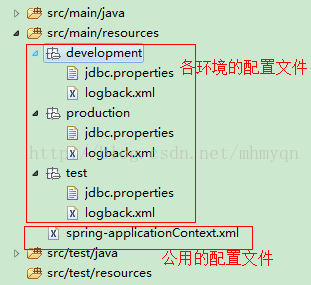
## 配置profile

首先是profile配置，在pom.xml中添加如下profile的配置：

|  |
| --- |
| 这里定义了三个环境，分别是development（开发环境）、test（测试环境）、production（生产环境），其中开发环境是默认激活的（activeByDefault为true），这样如果在不指定profile时默认是开发环境。  同时每个profile还定义了两个属性，其中profiles.active表示被激活的profile的名称，deploy.url表示发布服务器的地址。我们需要在下面使用到这两个属性。  另外host和port分别是发布服务器的主机地址和端口号。  <profiles>  <profile>  <!-- 本地开发环境 -->  <id>development</id>  <properties>  <profiles.active>development</profiles.active>  <deploy.url>http://host:port/manager/text</deploy.url>  </properties>  <activation>  <activeByDefault>true</activeByDefault>  </activation>  </profile>  <profile>  <!-- 测试环境 -->  <id>test</id>  <properties>  <profiles.active>test</profiles.active>  <deploy.url>http://host:port/manager/text</deploy.url>  </properties>  </profile>  <profile>  <!-- 生产环境 -->  <id>production</id>  <properties>  <profiles.active>production</profiles.active>  <deploy.url>http://host:port/manager/text</deploy.url>  </properties>  </profile>  </profiles> |

## 配置文件

针对不同的环境，我们定义不同的配置文件，而这些配置文件都做为资源文件放到maven工程的resources目录下，即src/main/resources目录下，且各个环境的配置分别放到相应的目录下，而所有环境都公用的配置，直接放到src/main/resources目录下即可。如下图所示：



如图所示，开发环境、测试环境、生产环境的配置文件分别放到src/main/resources目录下的development、test、production三个子目录中，而所有环境都公用的配置文件spring-applicationContext.xml直接放到src/main/resources目录下。其中jdbc.properties配置数据源、logback.xml配置日志。

## maven资源插件配置

在pom中的build节点下，配置资源文件的位置，如下所示：

|  |
| --- |
| 首先第一个资源文件位置src/main/resources需要排队提各个环境的配置文件，各个环境的配置我们在第二个<resource>节点中通过前面在profile中配置的profiles.active属性来指定。即src/main/resources/${profiles.active}。这样在激活指定的profile时，会加载指定目录下的配置文件，如当前激活的是production profile，那么这个资源目录就是src/main/resources/production。这样就达到了不同环境加载不同配置的目的。  <build>  <resources>  <!-- 定义资源文件-->  <resource>  <directory>src/main/resources</directory>  <!-- 资源根目录排除各环境的配置，使用单独的资源目录来指定 -->  <excludes>  <exclude>test/\*</exclude>  <exclude>production/\*</exclude>  <exclude>development/\*</exclude>  </excludes>  </resource>  <resource>  <directory>src/main/resources/${profiles.active}</directory>  </resource>  </resources>  </build> |

# maven测试和自动部署

## Maven 测试

|  |
| --- |
| #配置文件  <plugins>  <plugin>  <!-- test plugin -->  <artifactId>maven-surefire-plugin</artifactId>  <version>2.5</version>  <configuration>  <includes> <include>com/hp/it/mda/services/\*Test.java</include>  </includes>  <!-- the test files will not be execute 11  <excludes>  <exclude></exclude>  </excludes>  -->  </configuration>  </plugin>  </plugins>  Include 需要进行测试的Java文件  Exclude 不需要执行测试的文件 |
| #测试报告  默认情况下，maven-surefire-plugin会在项目的target/surefire-reports目录下生成两种格式的错误报告，简单文本格式和与Junit兼容的xml格式。 |

## 自动部署

|  |
| --- |
| <plugin>  <groupId>org.codehaus.cargo</groupId>  <artifactId>cargo-maven2-plugin</artifactId>  <version>1.0.6</version>  <configuration>  <container>  <containerId>tomcat6x</containerId>  <type>remote</type>  </container>  <configuration>  <type>runtime</type>  <properties>  <cargo.remote.username>tomcat</cargo.remote.username>  <cargo.remote.password>tomcat</cargo.remote.password>  <cargo.tomcat.manager.url>http://localhost:8080/manager</cargo.tomcat.manager.url>  </properties>  </configuration>  </configuration>  </plugin> |

# maven生成JavaDoc

在pom.xml中配置如下：

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-javadoc-plugin</artifactId>  <version>2.7</version>  <executions>  <execution>  <id>attach-javadocs</id>  <goals>  <goal>jar</goal>  </goals>  </execution>  </executions>  </plugin>  然后运行命令：  mvn javadoc:jar |

# 常用命令

|  |
| --- |
| mvn compile ：编译源代码  mvn test：编译源代码，测试代码，执行测试  mvn test-compile：编译源代码，测试代码  mvn package：把项目打成jar或者war包  mvn install：将jar包放入本地仓库中  mvn clean：删除target目录  mvn deploy：部署jar包至远程仓库  创建maven项目(目录骨架)：mvn archetype:generate -DgroupId=组织名  -DartifactId=模块名 -Dversion=版本号 -Dpackage=代码所存在的包名  mvn archetype:generate -DgroupId=com.hengtian -DartifactId=maven03 -Dversion=1.0.0SNAPSHOT -Dpackage=com.hengtian  **#mvn常用参数**  mvn -e 显示详细错误  mvn -U 强制更新snapshot类型的插件或依赖库（否则maven一天只会更新一次snapshot依赖）  mvn -o 运行offline模式，不联网更新依赖  mvn -N仅在当前项目模块执行命令，关闭reactor  -am --also-make 同时构建所列模块的依赖模块； -amd -also-make-dependents 同时构建依赖于所列模块的模块； -pl --projects <arg> 构建制定的模块，模块间用逗号分隔； -rf -resume-from <arg> 从指定的模块恢复反应堆  mvn -Dxxx=yyy指定java全局属性  mvn -Pxxx引用profile xxx  例如：clean install -pl 3t-admin -am -Pqa -DskipTests=true |