**Maven3.0.4学习教程**

[1、 前言 2](#_Toc345188316)

[2、 maven初步 3](#_Toc345188317)

[2.1. 安装 3](#_Toc345188318)

[2.2. 配置本地仓库 3](#_Toc345188319)

[2.3. 新建maven项目 4](#_Toc345188320)

[2.4. Maven目录结构说明 5](#_Toc345188321)

[2.5. Maven命令说明 5](#_Toc345188322)

[2.6. 如何在eclipse装maven插件 6](#_Toc345188323)

[3、 依赖 11](#_Toc345188324)

[3.1. Pom.xml文件结构说明 11](#_Toc345188325)

[3.2. 依赖 13](#_Toc345188326)

[3.3. 依赖范围说明 14](#_Toc345188327)

[4、 仓库 15](#_Toc345188328)

[4.1. 本地仓库/中央仓库/远程仓库 15](#_Toc345188329)

[5、 聚合与继承 16](#_Toc345188330)

[5.1. 聚合 16](#_Toc345188331)

[5.2. 继承 17](#_Toc345188332)

[5.3. 聚合与继承的关系 22](#_Toc345188333)

[5.4. Maven可继承的POM 元素 22](#_Toc345188334)

[5.5. 有关版本说明 23](#_Toc345188335)

[6、 私服 23](#_Toc345188336)

[6.1. 私服安装 23](#_Toc345188337)

[6.2. 私服介绍 24](#_Toc345188338)

[6.3. 开启索引下载 26](#_Toc345188339)

[6.4. 将nexus私服配置到maven中 26](#_Toc345188340)

[6.5. 项目发布到私服中 27](#_Toc345188341)

[6.6. 如何将中央仓库中找不到的jar上传到私服中 28](#_Toc345188342)

[7、 生命周期和插件 30](#_Toc345188343)

[7.1. 生命周期 30](#_Toc345188344)

[7.2. 插件 32](#_Toc345188345)

[8、 测试 33](#_Toc345188346)

[8.1. 如何进行项目测试 33](#_Toc345188347)

[8.2. 如何忽略测试 34](#_Toc345188348)

[8.3. 测试报告生成 34](#_Toc345188349)

[9、 项目发布 35](#_Toc345188350)

[9.1. Web项目创建 35](#_Toc345188351)

[9.2. 在pom.xml中添加依赖包 35](#_Toc345188352)

[9.3. 项目部署 36](#_Toc345188353)

[10、 站点发布 43](#_Toc345188354)

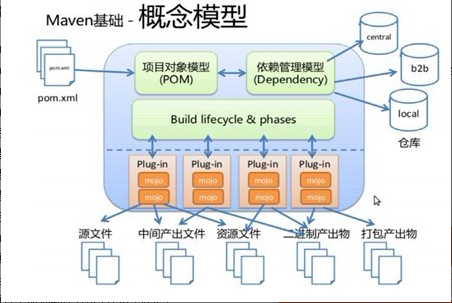
# 前言

maven是一个项目构建和管理的工具，提供了帮助管理构建、文档、报告、依赖、scms、发布、分发的方法。可以方便的编译代码、进行依赖管理、项目发布等。

maven的好处在于可以将项目过程规范化、自动化、高效化以及强大的可扩展性，利用maven自身及其插件还可以获得代码检查报告、单元测试覆盖率、实现持续集成等等。

Maven的基本原理很简单，采用远程仓库和本地仓库以及一个核心的配置文件pom.xml，pom.xml中定义的jar文件从远程仓库下载到本地仓库，各个应用使用同一个本地仓库的jar，同一个版本的jar只需下载一次，而且避免每个应用都去拷贝jar。同时它采用了现在流行的插件体系架构，所以maven的核心非常的小，只有几兆大小的文件，在执行maven任务时，才会自动下载需要的插件。

具体概念模型如下图：



Maven官方把maven定义为一个项目管理工具，下面我们来看看maven给我们的项目管理做了哪些工作？

* **项目标准化**

**Maven项目具有统一的项目结构，**这个项目结构是参考业界的最佳实践而成，为后面使用统一的maven命令打下了基础，如测试mvn test、打包mvn package等，无需写一行脚本，就可以方便的实现众多功能。

* **文档和报告**

使用mvn site可以快速生成项目站点，apache很多开源项目站点都采用maven生成，会出现built by maven字样的图标。

* **类库管理**

**类库管理是maven一个比较核心的功能，我们就需要将项目所依赖的类库加入到pom.xml中，那么maven会自动将依赖的类库下载到本地，并且下载的类库如果还依赖其他的类库，它也会自动下载过来，这样我们就不需要一个一个类库去下载了。**

* **发布管理**

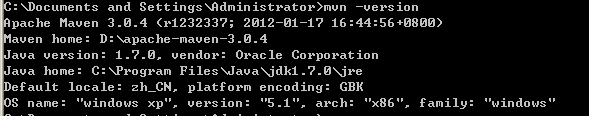
使用maven可以方便的进行项目发表管理。在项目开发到一定阶段，可以使用mvn package打包，它会自动先运行mvn test，跑所有的testcase，只有全部通过才能正确打包。生成的war包（如果项目的packaging为war）在target目录下。这个 war包与使用ant脚本生成一样，但无需一行脚本，这也是maven对比ant的优势之一。使用mvn install将编译和打包好的文件发布到distributionManager指定的远程repository。使用mvn deploy可以自动管理项目版本。

本学习教程的目的在于突出maven的重要知识点，提高对maven的学习效率。如果大家掌握了这些知识点，对于使用maven是没问题的。里面的实例多是我一个一个运行和测试过，希望对想要了解的maven的朋友们有所帮助。如果大家想全面掌握maven，推荐大家去看许晓斌写得《maven实战》，这本书写得相当不错，我也是学习了这本书后，提炼出maven的知识重点，来跟大家分享我的学习成果。

# maven初步

## 安装

* 从<http://maven.apache.org/download.html> 下载maven的安装包，目前最新的版本是3.0.4。
* 将下载的安装包解压到特定的目录下，设置环境变量中的path和M2\_HOME，以便在命令行中可以直接使用。如果在命令行中输入：mvn –version,出现下面这个界面，说明我们的maven已经安装成功。



## 配置本地仓库

对于仓库是什么概念，我们这里稍微说明一下，maven中的仓库就是放置各个jar和插件的地方，如果配置的是本地仓库，那么我们所引用的jar都是从本地来的。下面我们来看一个文件，文件放在maven的安装目录下面的conf子目录下面，大家打开这个文件夹可以看到有个叫settings.xml的文件夹，这个里面包含仓库地址、镜像、插件、代理等配置，也是maven中一个核心配置文件，在后面我们会经常跟她打交道。下面我来看看这个文件以及如何配置本地仓库。

* 打开这个文件，我们会看到Default: ~/.m2/repository这样一句话，这句话就告诉我们maven的默认仓库位置，默认是在c盘目录下面，我电脑的位置是在C:\Documents and Settings\Administrator\.m2这个目录下面，而c盘往往是系统盘，放在此目录下面的文件不太安全，所以我们需要更改一下默认的仓库位置，我电脑是放在了D:/java/maven/repository
* 大家可以在自己的电脑上建立一个仓库，休息setting中的仓库目录位置， <localRepository>D:/java/maven/repository</localRepository>。为了保持核心的配置文件不变，我们将settings.xml文件拷贝到这个目录下面。以后只要对这个文件做修改就可以了。

## 新建maven项目

我们以简单的helloworld来作为入门的实例，有些人说掌握了helloworld你就掌握了这门技术的一半了，对于maven来说，你掌握helloworld，你可能还稀里糊涂的，不要着急，后面我会慢慢讲解各个知识点。

* 新建pom.xml,其内容如下：



* 新建项目，由于maven有固定的目录格式，我们必须遵照maven的规则来建项目。先把具体的规则说一下，我们再来创建这个项目。

1. 源代码应该放置到src/main/java中
2. 源代码的资源文件应该放置在src/main/resources文件夹中
3. 测试代码应该放置到src/test/java中
4. 测试代码的资源文件应该放置在src/test/resources文件夹中

* 新建目录

1. 依次新建目录src/main/java/com/pango/maven/helloworld和src/test/java/com/pango/maven/helloworld
2. 分别在两个helloworld目录下新建HelloWorld.java和HelloWorldTest.java,其内容如下：

HelloWorld.java

Package com.pango.maven.helloworld

Public class HelloWorld{

Public String sayHello(){

Return “hello world”;

}

Public static void main(String args[]){

System.out.println(new HelloWorld().sayHello());

}

}

HelloWorldTest.java

Package com.pango.maven.helloworld

Public class HelloWorldTest{

@Test

Public void testSayHello(){

HelloWorld helloWorld = new HelloWorld();

String result = helloWorld.sayHello();

assertEquals(“hello world”,result);

}

}

## Maven目录结构说明

Maven总体目录结构如下图：



* Bin：改目录包含了mvn运行的脚本，这些脚本用来配置java命令，准备好classpath和相关的Java系统属性，然后执行Java命令。
* Boot： 该目录只包含一个文件，以maven 3.0为例，该文件为plexus-classworlds-2.2.3.jar。plexus-classworlds是一个类加载器框架，相对于默认的java类加载器，它提供了更丰富的语法以方便配置，Maven使用该框架加载自己的类库。更多关于classworlds的信息请参考<http://classworlds.codehaus.org/>。对于一般的Maven用户来说，不必关心该文件。
* Conf： 该目录包含了一个非常重要的文件settings.xml。
* Lib： 该目录包含了所有Maven运行时需要的Java类库，Maven本身是分模块开发的，因此用户能看到诸如mavn-core-3.0.jar、maven-model-3.0.jar之类的文件，此外这里还包含一些Maven用到的第三方依赖如common-cli-1.2.jar、google-collection-1.0.jar等等。

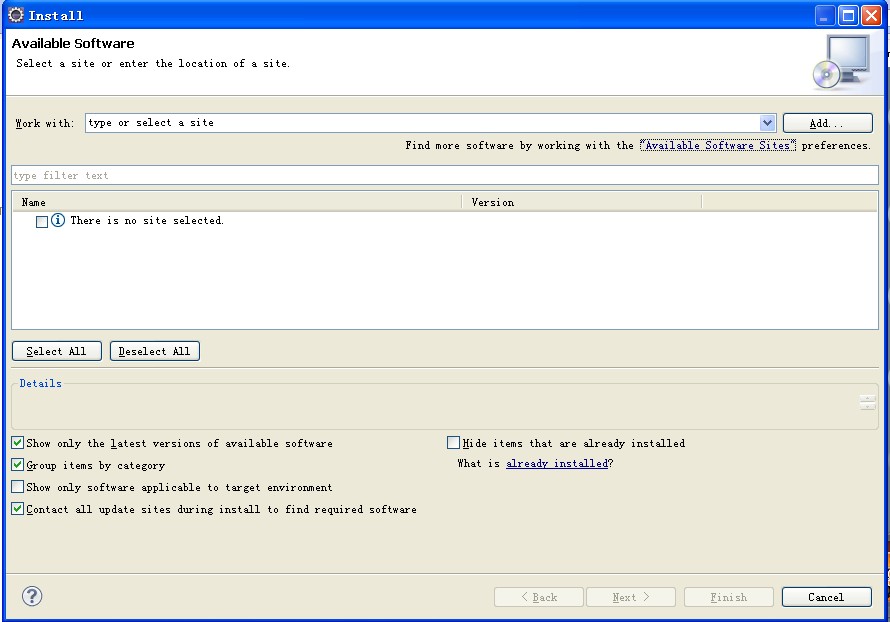
## Maven命令说明

1. mvn clean -->表示运行清理操作（会默认把target文件夹中的数据清理）
2. mvn clean compile-->表示先运行清理之后运行编译，会见代码编译到target文件夹中
3. mvn clean test-->运行清理和测试
4. mvn clean package-->运行清理和打包
5. mvn clean install-->运行清理和安装，会将打好的包安装到本地仓库中，以便其他的项目可以调用
6. mvn clean deploy-->运行清理和发布（发布到私服上面）

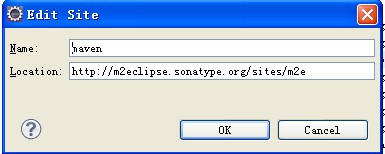
## 如何在eclipse装maven插件

从上面可以看到，我们手动创建一个maven项目是及其麻烦的，为此我们要借助工具来帮我们完成这些工作，这里就讲大家用得最多的eclipse为例，来讲讲在eclipse中如何装maven插件和如何创建maven项目。

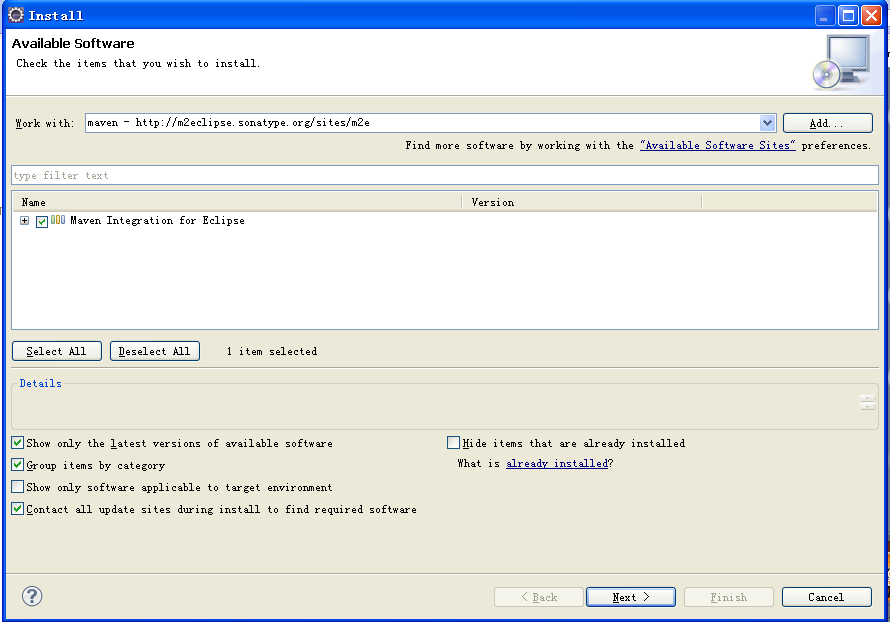
1. 打开eclipse，help->install如下图：



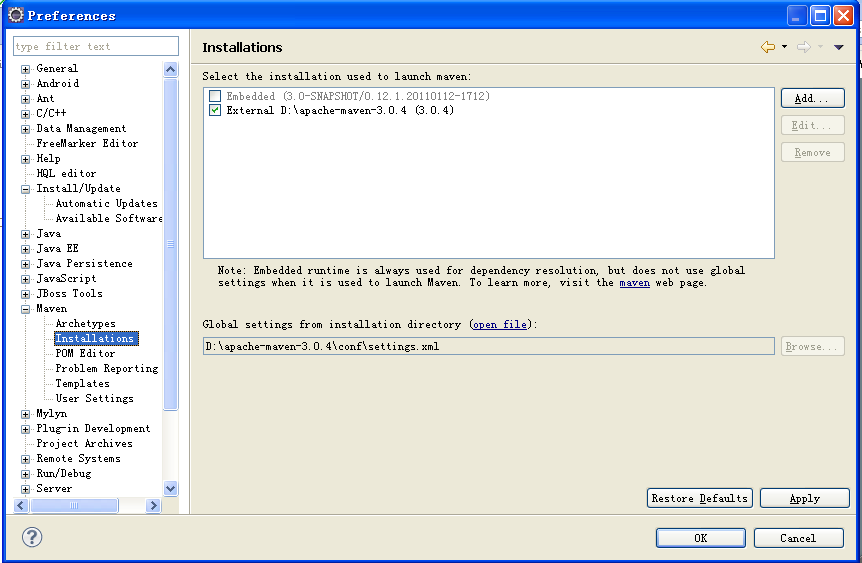
1. 点击add，在弹出的窗口中输入下面的信息



1. 点击ok后，通过网络请求将获取插件的情况，如下图：

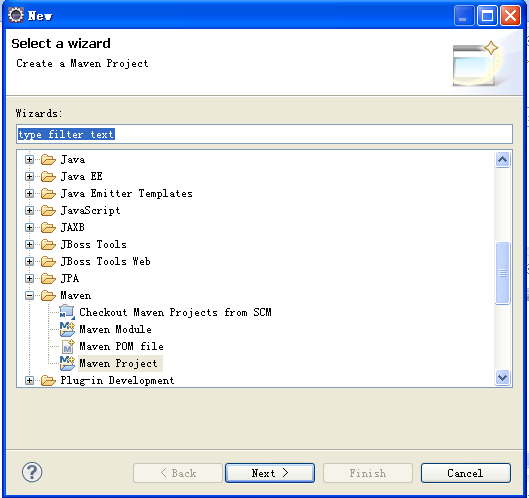


1. 一直点击下一步，如果碰到要求你同意协议的地方你同意协议即可，装完后eclipse会有一次重启的过程，重启后如果在windows->preference中的目录里面看到maven，说明我们已经安装成功，插件默认可能会内置一个maven，为了稳定起见，我们需要把它定位到我们自己安装的maven目录里面，查看下图：

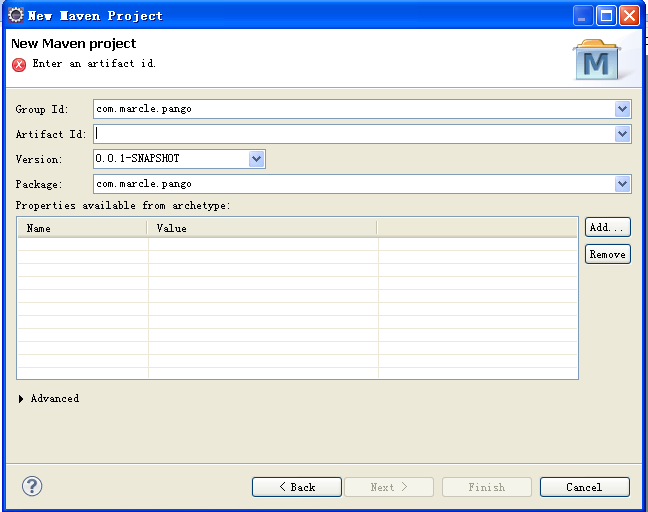


在add里面选择自己maven安装的目录，然后把最后一项user settings 定位自己的settings.xml目录中，这样maven插件算是装完了。

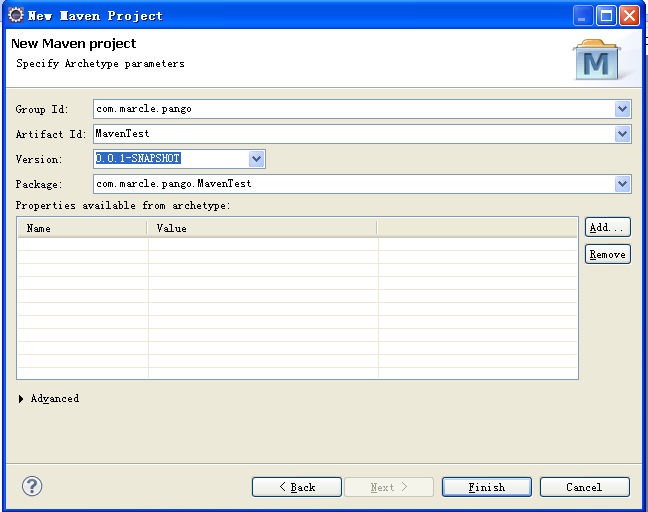
1. 下面我们看看如何在eclipse中创建maven项目，点击工具栏上面的file->new->other 找到maven，如下图



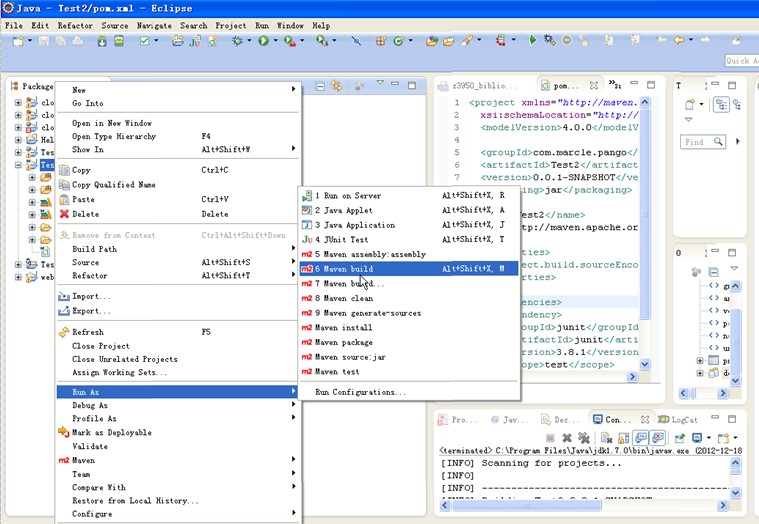
选择maven project 一直点击next，最后面会出现下面这样一个框：



在此界面中输入坐标信息，点击finish即可完成maven项目创建，如下图：

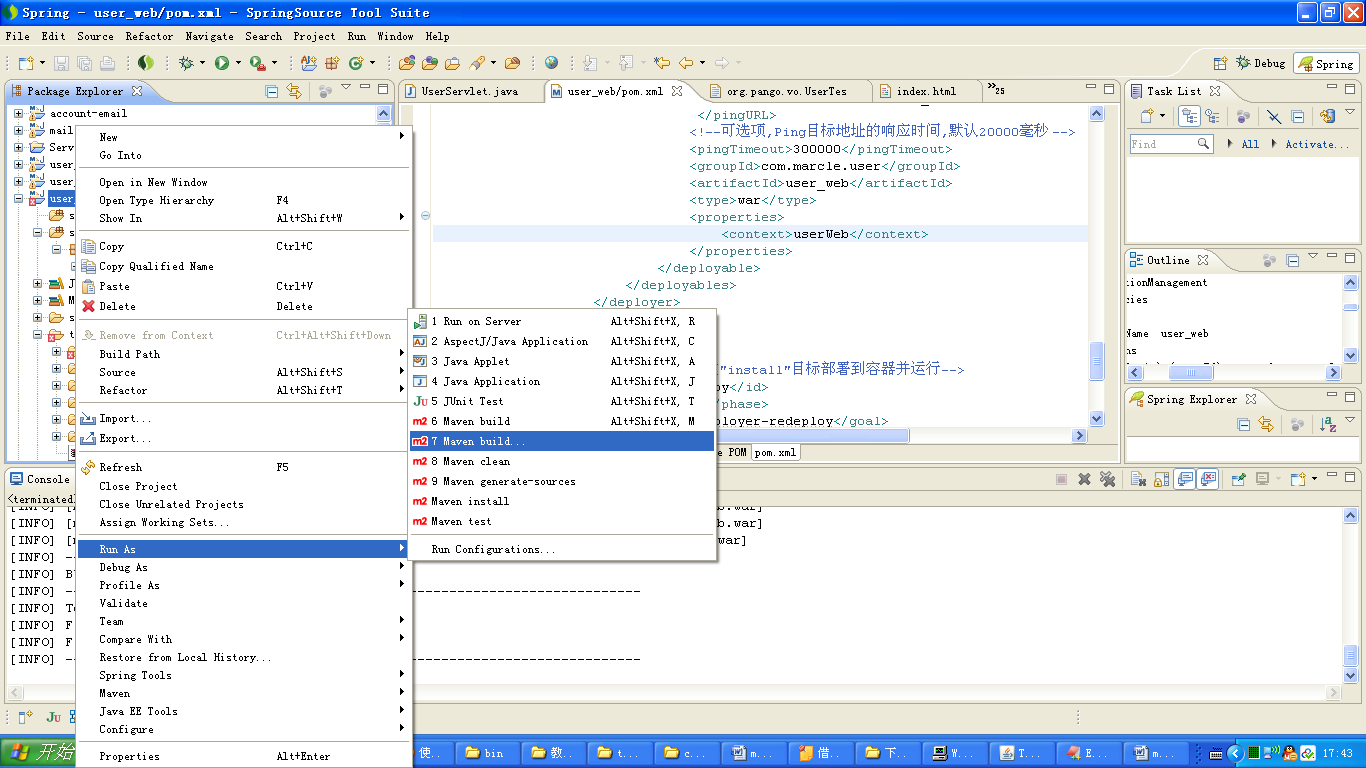


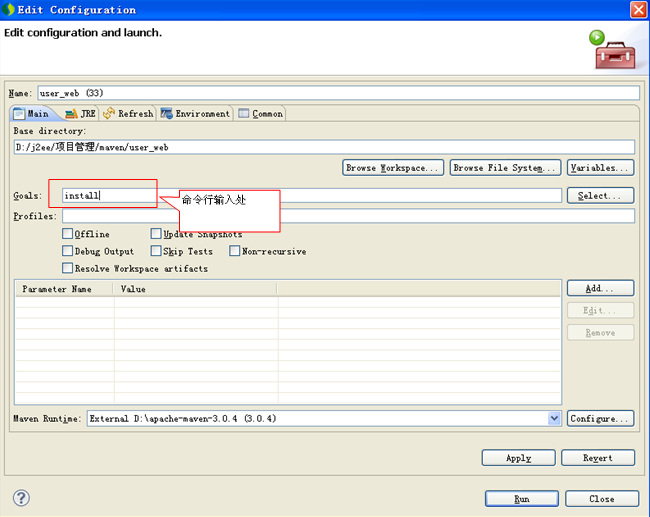
1. 下面我们来看看如何在eclipse中运行maven，



我可以从上图可以看到，当我们安装maven后，系统自动多了maven运行的命令参数，当然我们选择maven build我们可以输入自己的命令来运行maven的相关操作。运行后会在控制台中打印相关信息。

1. 如何输入其他命令





然后点击run即可运行。

# 依赖

## Pom.xml文件结构说明

通过上面第一章，我们对pom.xml已经有了初步的印象，我们通过一个发送邮件maven项目来说明maven中pom.xml主要标签的意思及用法，我来看下pom.xml文件的结构：

<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.pango.maven</groupId>

<artifactId>mail</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>mail</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<org.springframework-version>3.1.2.RELEASE</org.springframework-version>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${org.springframework-version}</version>

</dependency>

<dependency>

<groupId>javax.mail</groupId>

<artifactId>mail</artifactId>

<version>1.4.1</version>

</dependency>

<dependency>

<groupId>com.icegreen</groupId>

<artifactId>greenmail</artifactId>

<version>1.3.1b</version>

<scope>test</scope>

<exclusions>

<exclusion>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

</project>

我们可以看到maven的pom.xml文件结构非常清晰，把项目创建好后，我们基本上是在dependencies元素下添加一些子元素及一些插件，下面我们来介绍一下各个元素的含义。

1. Project是所有pom.xml的根元素，并且在里面定义了命名空间和xsd元素
2. ModelVersion 当前pom模型的版本
3. GroupId定义当前maven项目隶属的实际项目
4. ArtifactId定义项目中的某个模块名称
5. Version 定义maven项目当前所处的版本号
6. Packaging定义maven项目的打包方式。
7. Dependencies 元素底下就是加入依赖包的地方，那么我们从哪里查询依赖包呢，可以查询的地方比较多，我给出一个大家用得比较多的仓库，<http://mvnrepository.com>。
8. 每个dependency都是一个依赖包，依赖包也就是在dependency里面定义各个依赖包的坐标，这样maven就会从网上下载依赖包到你本地仓库中，有所不同的是dependency元素包含了一个子元素，这个就是对maven生命周期的一个说明，当然除了上面四个子元素外，还包含几个其他的元素。
9. type说明依赖的类型
10. optional标记依赖是否可选
11. exclusions 用来排斥传递依赖

我们具体来看看这个结构：

<dependency>

<groupId>实际项目</groupId>

<artifactId>模块</artifactId>

<version>版本</version>

<type>依赖类型</type>

<scope>依赖范围</scope>

<optional>依赖是否可选</optional>

<!—主要用于排除传递性依赖-->

<exclusions>

<exclusion>

<groupId>…</groupId>

<artifactId>…</artifactId>

</exclusion>

</exclusions>

</dependency>

Maven是通过groupId、artifactId、version这三类似于坐标的元素来确定唯一性的，因此这三个元素对于每个依赖多是必须的。

## 依赖

何为依赖？比如上面的列子中的mail，它需要跟activation-1.1.jar一起才能实现邮件的发送，我们称mail依赖activation-1.1.jar，在maven中，如果mail包中依赖activation-1.1.jar包。如果在以前我们需要手动的去找activation-1.1.jar这个包，找不到我们还不知哪里报错，在这里我们就看到maven的好处了，它就是一个仓库，它里面有各种各样的包。

依赖具有传递性，比如A->B,B->C,那么A间接的依赖于C，这就是依赖的传递性，其中B为A的第一直接依赖，C为A的第二直接依赖。我们通过下面这个表格来说明第一直接依赖和第二直接依赖之间，其中最左边一栏是第一直接依赖，最上面那一栏为第二直接依赖。中间交叉的是传递性依赖范围。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Compile | Test | Provided | Runtime |
| Compile | Compile |  |  | Runtime |
| Test | Test |  |  | Test |
| Provided | Provided |  | Provided | Provided |
| Runtime | Runtime |  |  | Runtime |

下面我们来思考这样一个问题，如果A->B->C->X(1.0),A->D-X(2.0),即A间接依赖X，我们可以看到有两条路径都依赖X，那么maven将会选择哪个版本的X？maven当中有一套自己的规则，我们来说明一下，maven传递性依赖的一些规则以及如何排除依赖冲突。

Maven里面对于传递性依赖有以下几个规则：

1. 最短路径原则：如果A对于依赖路径中有两个相同的jar包，那么选择路径短的那个包
2. 第一声明优先原则：如果A对于依赖路径中有两个相同的jar包，路径也相同，那么依赖写在前面的优先。
3. 可选依赖不会传递，如A->B,B->C,B->D,A对B直接依赖，B对C和D是可选依赖，那么在A中不会引入C和D

最后我们来说说如何排除依赖，我们在前面已经写过，在这个案例中，greenmail依赖于slf4j，而这个包在我们案例中是没有任何意义的，为此我们需要在greenmail中排除对slf4j的依赖，通过下面这个方式，就可以排除依赖了：

<exclusions>

<exclusion>

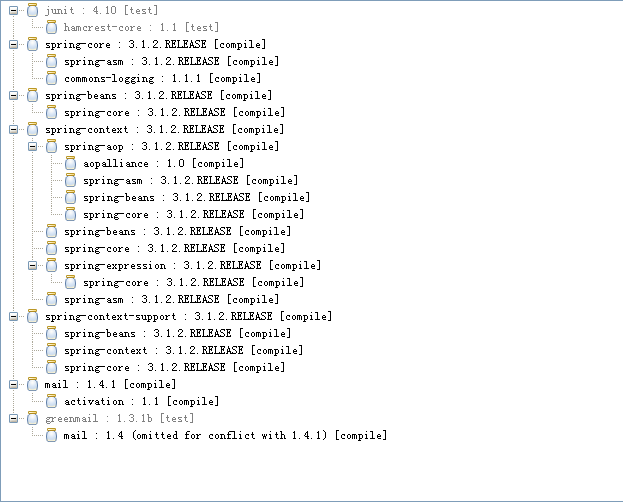
<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</exclusion>

</exclusions>

对于我们不需要的jar，我们通过这个方式来排除依赖。我们来看看我们这个案例依赖情况图：



基本符合我们上面所讲的规律。

## 依赖范围说明

由于不同的包在不同的地方用到，像junit我们只有在做测试的时候会用到这个包，在我们项目发布的时候，用不到这个包；还有servlet-api，在项目编译的时候将会用到这个包，而项目发布的时候就不会用到这个包，因为一般容器已经自带这个包，如果我们导入，有可能会出现冲突，所以maven引入了依赖范围这个概念，即我们上面提到的scope来解决这个问题。Maven中有主要有以下这几种依赖范围：

1. test范围指的是测试范围有效，在编译和打包时都不会使用这个依赖
2. compile范围指的是编译范围有效，在编译和打包时都会将依赖存储进去
3. provided依赖：在编译和测试的过程有效，最后生成war包时不会加入，诸如：servlet-api，因为servlet-api，tomcat等web服务器已经存在了，如果再打包会冲突
4. runtime在运行的时候依赖，在编译的时候不依赖

# 仓库

## 本地仓库/中央仓库/远程仓库



本地仓库是指存在于我们本机的仓库，在我们加入依赖时候，首先会跑到我们的本地库去找，如果找不到则会跑到远程仓库中去找。对于依赖的包大家可以从这个地址进行搜索：<http://mvnrepository.com/>。

中央仓库这里我解释为供所有本地仓库下载依赖包的一个仓库，它是部署在局域网内的一个仓库。

远程仓库是指在中央仓库找不到的依赖包则会跑到远程仓库去下载的仓库，它需要互联网来实现依赖包的下载。

下面我们来看一个问题：

项目组有甲和乙，甲做了A模块，已做了B模块，而B模块需要用到A模块的内容，那么我们怎样来解决这个问题，我们通常的方法是将A模块提交到svn后，然后乙再将项目同步过来导入到他自己的仓库中。我们看到过程非常复杂，并且甲对A模块做了修改之后，乙需要不断同步A的代码，也是一个非常繁琐的过程。这样我们就需要一个中央仓库来解决这个问题，也就是上图画得中央仓库。

通过上图我们可以看到，在一个公司或单位中，部署一个中央仓库，本地用户先到本地仓库查找相关的依赖包，如果找不到则跑到中央仓库中去找，如果中央仓库找不到，那么将会跑到外网去找。对于我们项目提交，我们会统一提交的中央仓库，这样无论甲还乙我们都可以引用中央仓库中的jar包，这样就可以得到一个比较完美解决方法。 建立中央仓库我们还有另一个好处，可以节省我们的带宽，只要有一个用户加过一个依赖包，那么第二用户就可以直接从中央仓库中下载，而不是从远程仓库下载，这个是在局域网内操作的，速度特别快，特别是对于一些网络不稳定的地方，这样做有极大的好处。对于如何建立中央仓库，我们将在下一章做讲解。

# 聚合与继承

## 聚合

假设我们这个项目有core模块、dao模块、service模块、web模块，那么如何将我们的项目各个模块进行一次编译、打包、发布即可，而不用我们一个一个模块进行编译、打包、发布呢？这就需要maven中聚合来解决这个问题。这样我们需要构建另外一个模块user\_parent, ，然后通过该模块构建整个项目的所有模块。看下面的pom.xml配置：

<modelVersion>4.0.0</modelVersion>

<groupId>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>

../user-core

</module>

<module>

../user-log

</module>

<module>

../user-dao

</module>

<module>

../user-service

</module>

</modules>

注意：**packaging的类型为pom** **，module的值是一个以当前POM为主目录的相对路径。**

这样配置后，我们只需要一次性运行user\_parent这个模块，即可将所有模块进行编译、打包、发布。

## 继承

如果多个模块出现相同的依赖包，这样在pom.xml文件的内容出现了冗余、重复的内容，解决这个问题其实使用Maven的继承机制即可，就像Java的继承一样，父类就像一个模板，子类继承自父类，那么有些通用的方法、变量都不必在子类中再重复声明了。Maven的继承机制类似，在一个父级别的Maven的pom文件中定义了相关的常量、依赖、插件等等配置后，实际项目模块可以继承此父项目 的pom文件，重复的项不必显示的再声明一遍了，相当于父Maven项目就是个模板，等着其他子模块去继承。不过父Maven项目要高度抽象，高度提取公 共的部分（交集）。下面我们来看看maven中继承怎么使用。

在user\_parent模块中做如下配置：

<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>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>

../user-core

</module>

<module>

../user-log

</module>

<module>

../user-dao

</module>

<module>

../user-service

</module>

</modules>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<junit.version>4.10</junit.version>

</properties>

<dependencies>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-dao</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-log</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-core</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>3.6.10.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.19</version>

</dependency>

<dependency>

<groupId>javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.12.1.GA</version>

</dependency>

<dependency>

<groupId>org.ow2.orchestra.eclipse.birt</groupId>

<artifactId>org.ow2.orchestra.eclipse.birt.chart.engine</artifactId>

<version>3.7.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

写法很简单，就跟普通的依赖是一样的写法，我们把需要的继承的包直接写dependencies中即可。

下面我们来看看如何在其他项目中如何继承这个项目中的jar包，看下面这段配置：

<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>

<parent>

<groupId>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<relativePath>../user-parent/pom.xml</relativePath>

</parent>

<artifactId>user-core</artifactId>

<packaging>jar</packaging>

<name>user-core</name>

<url>http://maven.apache.org</url>

</project>

我们就是通过parent这个元素来继承父项目中的包，在parent中定义parent项目的坐标以及父项目中pom的相对位置。

那么有些人就会问，如果每个项目中都继承这个父项目，有些项目中不是出现很多没用的包，确实会存在这个问题，不过不要着急，maven已经为我们提供这样的解决方法。

对于需要选择继承的我们可以把依赖放入<dependencyManagement>，我们看下面的配置：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-dao</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-log</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-core</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>3.6.10.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.19</version>

</dependency>

<dependency>

<groupId>javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.12.1.GA</version>

</dependency>

<dependency>

<groupId>org.ow2.orchestra.eclipse.birt</groupId>

<artifactId>org.ow2.orchestra.eclipse.birt.chart.engine</artifactId>

<version>3.7.0</version>

</dependency>

</dependencies>

</dependencyManagement>

对于选择继承，把继承的配置加入到dependencyManagement中，对于继承的项目，我需要引入依赖的声明，如下：

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

</dependency>

我们不需要进行版本的声明，这样就可以根据自己的需要引入需要的包，而不会继承全部的包。

对于每个项目都需要继承的，我们按下面这种方式加入依赖管理：

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

这样继承的每个项目，就都会有这个jar包了 。

## ****聚合与继承的关系****

**区别** ：

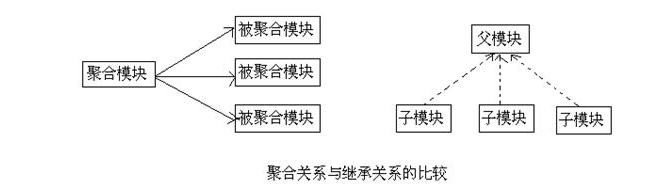
**1**.对于聚合模块来说，它知道有哪些被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。

2.对于继承关系的父 POM来说，它不知道有哪些子模块继承与它，但那些子模块都必须知道自己的父 POM是什么。

**共同点** ：

1.聚合 POM与继承关系中的父POM的 packaging都是pom

2.聚合模块与继承关系中的父模块除了 POM之外都没有实际的内容。



**注：在现有的实际项目中一个** **POM既是聚合POM，又是父** **POM，这么做主要是为了方便**

## ****Maven可继承的POM**** ****元素****

1. groupId ：项目组 ID ，项目坐标的核心元素；
2. version ：项目版本，项目坐标的核心元素；
3. description ：项目的描述信息；
4. organization ：项目的组织信息；
5. inceptionYear ：项目的创始年份；
6. url ：项目的 url 地址
7. develoers ：项目的开发者信息；
8. contributors ：项目的贡献者信息；
9. distributionManagerment ：项目的部署信息；
10. issueManagement ：缺陷跟踪系统信息；
11. ciManagement ：项目的持续继承信息；
12. scm ：项目的版本控制信息；
13. mailingListserv ：项目的邮件列表信息；
14. properties ：自定义的 Maven 属性；
15. dependencies ：项目的依赖配置；
16. dependencyManagement ：醒目的依赖管理配置；
17. repositories ：项目的仓库配置；
18. build ：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等；
19. reporting ：包括项目的报告输出目录配置、报告插件配置等。

## 有关版本说明

对于用过svn或者cvs的朋友们，都会知道，每次修改都会提交一个版本到服务器上，对于我们平常所搭建的项目，大家可能没有注意到版本这个概念。其实版本对于叠加式开发的项目是个很重要的概念，通过上面的依赖，我们就可以清楚地看到一个version，这个就是引入依赖包的版本。

那么版本一共可以分为几个层次，一般来说，版本可以分为：总版本号.分支版本号.小版本号-里程碑版本。

总版本号的变动一般表示框架的变动

分支版本号：一般表示增加了一些功能

小版本号：在分支版本上面进行bug的修复

里程碑：SNAPSHOT-->alpha-->beta-->release-->GA

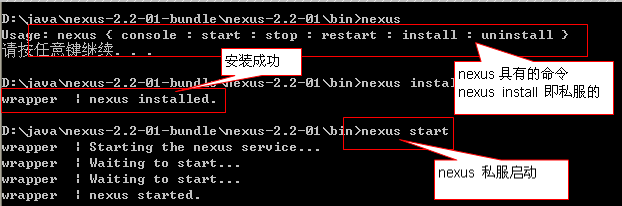
user0.0.1-SNAPSHOT-->user0.0.1-Release--->user1.0.0SHAPSHOT -->user1.0.0-Rlease

                          -->user0.1.0-SNAPSHOT-->user0.1.0-Rlease

# 私服

## 私服安装

* 从<http://www.sonatype.org/downloads/nexus-2.2-01-bundle.zip下载nexus>包，我下载的是最新版本2.2的。
* 将其解压到某个目录下面，在path环境变量中，加入bin目录，以便在命令行下直接操作，具体的安装过程是很简单的，看下图：



安装完成后，nexus会加入到服务中，下次系统重启的时候，nexus服务将会自动重启。

* Nexus私服默认端口为8081，即我们可以通过<http://localhost:8081/nexus/index.html#welcome>来访问我们部署的私服。具体效果如下图：



点击login in通过用户名和密码即可进入系统。默认的用户名和密码是admin/admin123

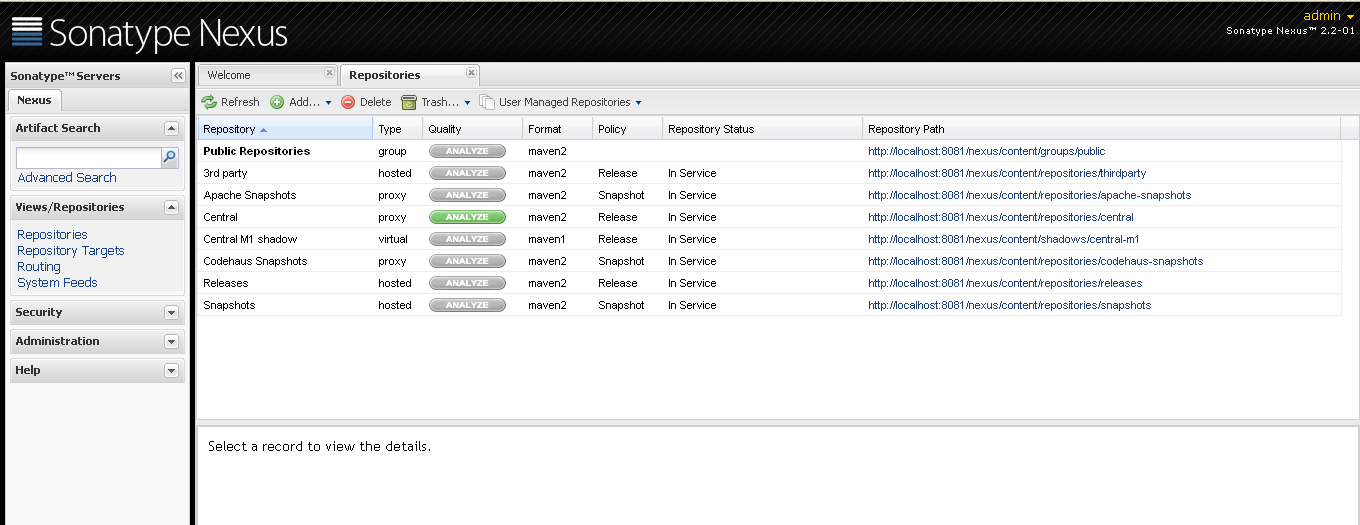


## 私服介绍

通过上面的介绍，我们已经成功部署了nexus私服，nexus是一个对软件管理非常好的工具，如果你开发软件，nexus可以帮助你将你开发的组件与其它开发者或终端用户进行共享。maven的中央仓库对于maven用户来说是及其方便的，但是我们一直推荐维护自己的本地仓库以便在你的开发组下保证更大的稳定性，而nexus可以非常简单地维护本地仓库以及访问外部的仓库，可以很好衔接本地仓库和外部仓库的关系，来达到软件项目的分享。

下面我们对私服做下简要的介绍：

登录后，我们点击repositories，我们将会看到下图：



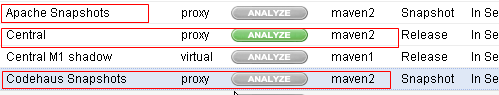
Repositories这块是我们使用nexus的核心东西在此只对这个模块做简要介绍，对于其它的内容，大家可以查看nexus官方文档，官网有本《nxbook》，如果大家想进一步了解nexus，可以下载下来看看。

Repositories就是指仓库，它包含三种类型仓库，分别为host仓库、proxy仓库、group仓库。

1、host的仓库。内部项目的发布仓库



2、proxy的仓库。从远程中央仓库中寻找数据的仓库



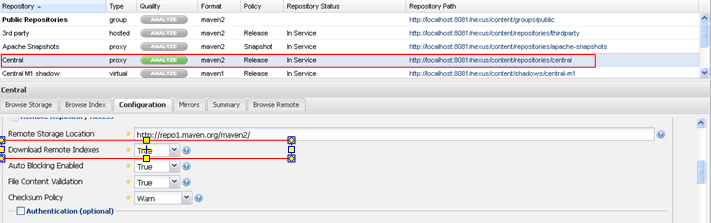
3、group仓库。组仓库用来方便开发人员进行设置的仓库

## 开启索引下载

当我们安装完后，我们直接搜索依赖的包是搜索不到，这就需要我们从nexus官网上下载依赖包的索引，类似于lucene中的索引，方便我们查询用，下面我们来看看如何引入索引。

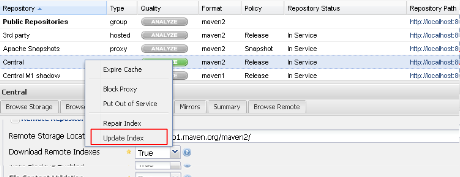
引入索引包的方式有两种，一直是将索引的包下载下来后，放入私服的这个D:\java\nexus-2.2-01-bundle\sonatype-work\nexus\indexer\central-ctx目录下面；另一种方式通过网络之间下载。这里我们介绍第二种方式。

1. 打开nexus的管理后台，进入respositories，点击center栏目，我们看到下面这样的情况。



我们看到上面红色方框的部分，将底下download remote indexs 设置为true，最后点击保存。

2、启动下载索引，右击上面center那一栏，会弹出一个提示框，如下图：



我们只需要点击update index 即可索引的更新，文件有点大，需要一点时间。

索引下载后，我们就可以在私服中搜索我们需要的包。

## 将nexus私服配置到maven中

通过上面的知识点，我们对nexus有了大概了解，那么如何将我们本地的maven配置成从这个私服中下载，这个是我们关心的。

1、设置镜像（设置setting.xml）



2、在settings.xml配置仓库，此时本机中的所有maven项目都会去私有仓库中查询依赖

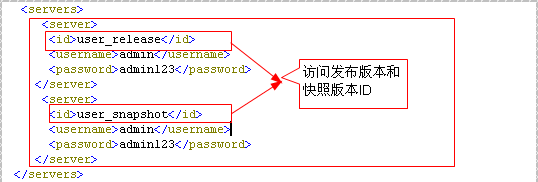


3、settings.xml中激活仓库



## 项目发布到私服中

1. 在settings.xml中，配置具有发布发布版本和快照版本权限的用户。

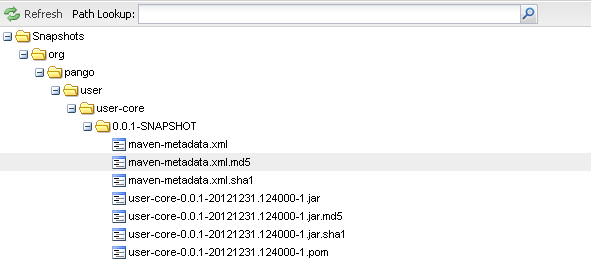


上面的user\_release和user\_snapshot必须与步骤二发布版本和快照版本的id对应起来。

2、maven项目中开启项目发布到私服的方法



经过上面上述配置后，我们既可以将我们的项目发布到私服中，我们也可以利用发布到私服上的项目了，下面这个图就是把我们项目发布到私服中的情况图：



## 如何将中央仓库中找不到的jar上传到私服中

在应用中，很多需要认证的jar包，往往在中央仓库中是找不到的，这怎么办呢？有三种方案：

1. 将jar直接拷贝到web-info下的lib中，然后导入到类路径下面。既然用了maven，不建议使用这种方法
2. 使用maven的命令将jar导入到本地库，命令如下：

mvn install:install-file -DgroupId=org.apache.ws.security -DartifactId=wss4j -Dversion=1.5.8

-Dpackaging=jar -Dfile=c:/wss4j-1.5.8.jar

其中-DgroupId为项目名称 DartifactId 模块名称 –Dversion为版本号 为Dpackaging为打包方式

-Dfile为jar包所在路径

这些参数根据自己的需要进行调整，对于这种方法，没有架设私服的情况下，可以将jar通过这种方式导入到本地库中。在maven项目中，我们通过下面这种方式即可将此jar包引入到项目中。

<dependency>

<groupId> org.apache.ws.security </groupId>

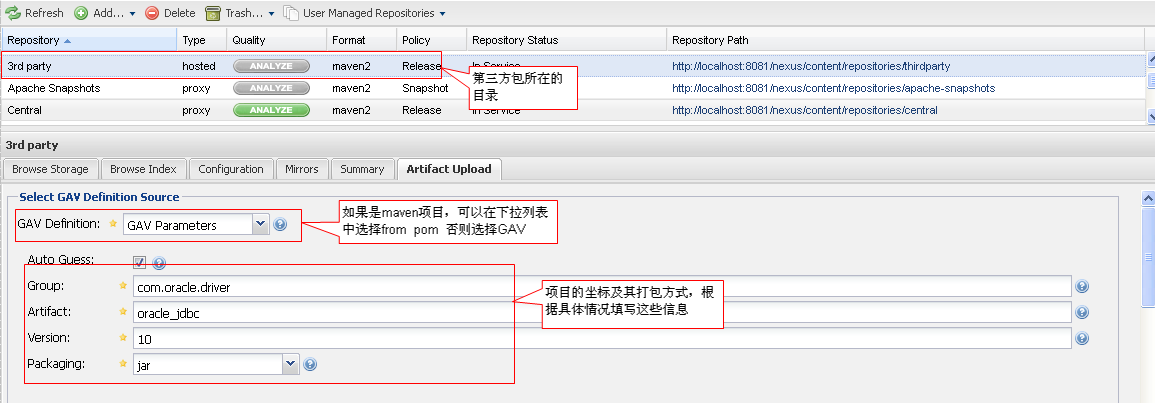
<artifactId> wss4j </artifactId>

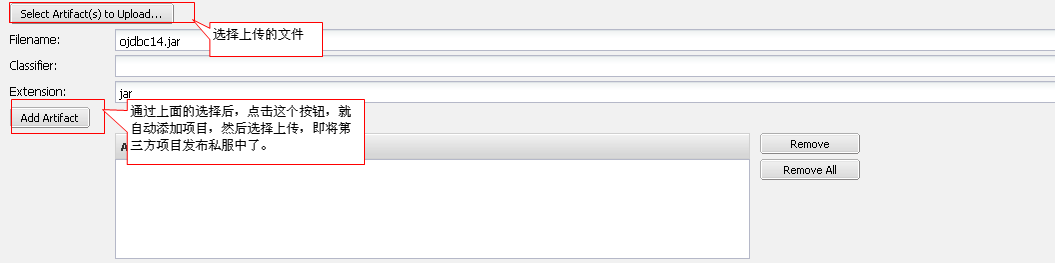
<version>1.5.8 </version>

</dependency>

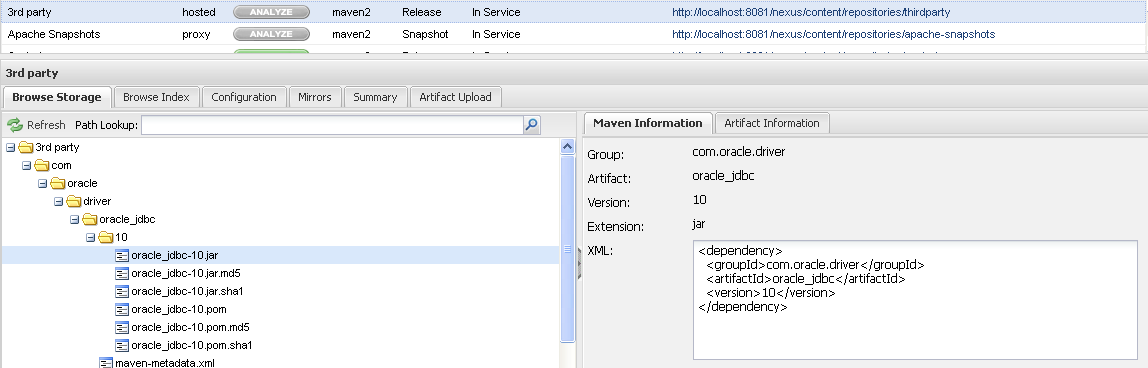
1. 如果架设了私服的，我们可以将jar上传到私服中，以便供一个组织中所有用户使用，下面我以oracle的驱动包为例介绍一下如何将jar包上传到私服中。

* 登录私服，点击[Repositories](http://localhost:8081/nexus/index.html)





* 通过上面的上传后，我们看下上传的结果



这样我们在项目中只要将

<dependency>

<groupId>com.oracle.driver</groupId>

<artifactId>oracle\_jdbc</artifactId>

<version>10</version>

</dependency>

依赖引入到项目中即可。

# 生命周期和插件

## 生命周期

何为maven的生命周期？我们在开发项目的时候，我们不断地在编译、测试、打包、部署等过程，maven的生命周期就是对所有这些过程的一个抽象与统一，她的生命周期包含项目的清理、初始化、编译、测试、打包、集成测试、验证、部署、站点生成等几乎所有的过程，而且maven的生命周期是及其灵活，她生命周期的每个阶段是通过插件来实现的，maven也内置了很多插件，所以我们在项目进行编译、测试、打包的过程是没有感觉到。像编译是通过maven-compile-plugin实现的、测试是通过maven-surefire-plugin实现的。下面我们详细介绍一下maven的生命周期。

Maven有三套相互独立的生命周期，请注意这里说的是“三套”，而且“相互独立”，初学者容易将Maven的生命周期看成一个整体，其实不然。这三套生命周期分别是：

* Clean Lifecycle 在进行真正的构建之前进行一些清理工作。
* Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。
* Site Lifecycle 生成项目报告，站点，发布站点。

我再次强调一下它们是相互独立的，你可以仅仅调用clean来清理工作目录，仅仅调用site来生成站点。当然你也可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

知道了每套生命周期的大概用途和相互关系以后，来逐个详细看一下每套生命周期，Clean和Site相对比较简单，先解释一下。

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

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

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

下面看一下Site生命周期的各个阶段：

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

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

最后，来看一下Maven的最重要的Default生命周期，绝大部分工作都发生在这个生命周期中，这里，我只解释一些比较重要和常用的阶段：

* validate
* generate-sources
* process-sources
* generate-resources
* process-resources     复制并处理资源文件，至目标目录，准备打包。
* compile     编译项目的源代码。
* process-classes
* generate-test-sources
* process-test-sources
* generate-test-resources
* process-test-resources     复制并处理资源文件，至目标测试目录。
* test-compile     编译测试源代码。
* process-test-classes
* test     使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
* prepare-package
* package     接受编译好的代码，打包成可发布的格式，如 JAR 。
* pre-integration-test
* integration-test
* post-integration-test
* verify
* install     将包安装至本地仓库，以让其它项目依赖。
* deploy     将最终的包复制到远程的仓库，以让其它开发人员与项目共享。

基本上，根据名称我们就能猜出每个阶段的用途，关于其它阶段的解释，请参考 <http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

记住，运行任何一个阶段的时候，它前面的所有阶段都会被运行，这也就是为什么我们运行**mvn install** 的时候，代码会被编译，测试，打包。

此外，Maven的插件机制是完全依赖Maven的生命周期的，因此理解生命周期至关重要，接下来我将会进一步解释Maven的插件机制。

## 插件

通过上面的生命周期我们可以了解到，不同的生命周期绑定不同的插件；同时我们知道，下载下来的maven核心的东西不过3-4M，它主要就是通过插件来完成这些工作的，一旦碰到没有的插件，它会跑到相应的地方下载，然后来完成整个过程。那么在我们的项目中如何使用插件呢？

打开<http://maven.apache.org/plugins/index.html>网址，我们可以看到apache下面的很多插件，apache下面的插件是比较正规的，它里面的信息非常详细。下面我们来看看里面有个source的插件的用法。

Source插件是对源码进行打包的一个插件，默认情况下，它会将生成的源码放在工程目录的target下面。

Source插件具有五个目标：

* [source:aggregate](http://maven.apache.org/plugins/maven-source-plugin/aggregate-mojo.html) 将所有模块的代码聚合到一个模块中
* [source:jar](http://maven.apache.org/plugins/maven-source-plugin/jar-mojo.html) is used to bundle the main sources of the project into a jar archive.
* [source:test-jar](http://maven.apache.org/plugins/maven-source-plugin/test-jar-mojo.html) on the other hand, is used to bundle the test sources of the project into a jar archive.
* [source:jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/jar-no-fork-mojo.html) is similar to **jar** but does not fork the build lifecycle.
* [source:test-jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/test-jar-no-fork-mojo.html) is similar to **test-jar** but does not fork the build lifecycle.

在我们的工程中，在后面引入下面这段配置：

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-source-plugin</artifactId>

<version>2.1.2</version>

</plugin>

</plugins>

</build>

上面这段配置就是对源码进行打包的插件，我们运行[source:jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/jar-no-fork-mojo.html)，那么在项目的目录底下的target会生成一个类似于user-core-0.0.1-SNAPSHOT-sources.jar这样的文件，即项目的源文件。那么如何将这个插件与特定的生命周期绑定呢？我们来看下面这段配置：



通过这段配置，大家可以用 mvn package 当将项目打包的同时会将源代码进行打包。

Apache里面还有很多有用的插件，大家可以自己去试一下，里面说明很详细，大家只要按着官方文档进行配置，一般情况下是没问题的。

# 测试

## 如何进行项目测试

做过项目的朋友们都写过单元测试，现在的单元测试一般是居于注解的单元测试。我在user-core这个项目中写了一个单元测试UserTest，当我们运行mvn test的时候，在控制台输出一下信息：

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.453 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

这个就是测试的一个结果，以上代表运行了2个实例，失败0个，错误0个，忽略0个。

有些朋友会觉得奇怪，maven怎么知道我们写了哪些测试类呢？这就是约定大于配置的效果，默认情况下，maven会去src/test/java 下面去寻找类，此文件下面的类需要符合以下的规则：

* Test\*.java:任何目录以Test开头的类
* \*Test.java:任何以Test结尾的类
* \*TestCase.java:任何以TestCase结尾的类

## 如何忽略测试

如果在想节省构建时间，忽略全部的测试，我们可以在pom.xml中加入下面这段配置：

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.5</version>

<configuration>

<skipTests>true</skipTests>

</configuration>

</plugin>

这样配置后，将会忽略所有的测试，那么如果要忽略某个测试类该怎么做呢？看下面这个命令：

Mvn test –Dtest=UserTest

这样就会忽略UserTest这个类，如果要忽略多个测试，只要在后面加逗号再加对应的测试类即可。在pom.xml中我们也可以配置不参与测试的类，其配置文件如下：

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.5</version>

<configuration>

<includes>

<include>

\*\*/\*Test.java

</include>

</includes>

<excludes>

<exclude>

\*\*/UserTest.java

</exclude>

</excludes>

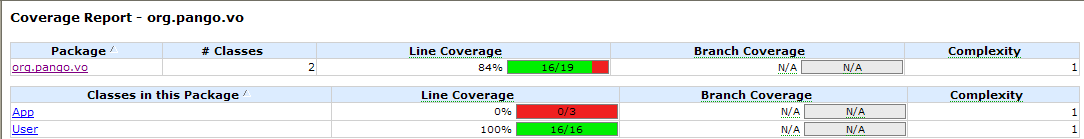
</configuration>

</plugin>

其中includes里面是参与测试的，excludes是不参与测试的。

## 测试报告生成

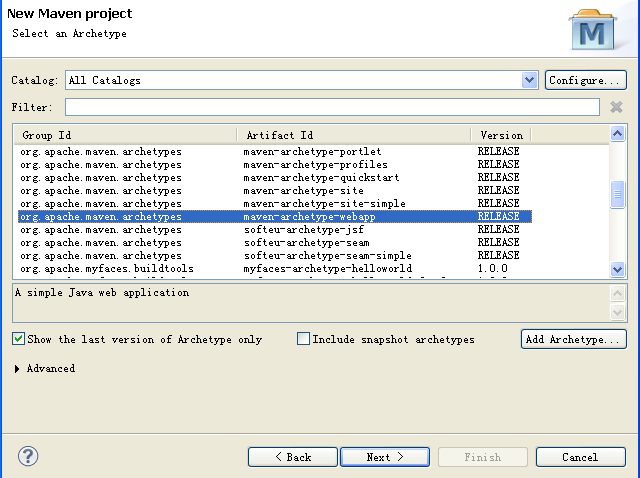
输入命令cobertura: cobertura,将会在target下生成一个站点测试覆盖率报告，如下图



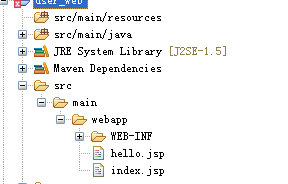
# 项目发布

## Web项目创建

Web项目的创建，其它步骤跟创建其它项目一样，只要注意一下下面这个步骤的时候，大家选择我选中的这个创建方式，创建的即为web项目。



创建完成后的目录格式如下：



## 在pom.xml中添加依赖包

<distributionManagement>

<snapshotRepository>

<id>user\_snapshot</id>

<name>User project snapshot</name>

<url>http://localhost:8088/nexus/content/repositories/MyUserReposSonatype/</url>

</snapshotRepository>

<repository>

<id>user\_release</id>

<name>User project release</name>

<url>http://localhost:8088/nexus/content/repositories/MyUserReposRelease/</url>

</repository>

</distributionManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>servletapi</groupId>

<artifactId>servletapi</artifactId>

<version>2.4</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>jsp-api</artifactId>

<version>2.2</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.19</version>

</dependency>

</dependencies>

注意上面的servletapi和jsp-api是编译的时候必须的包，但是在打包的时候是不需要这两个包的，因为一般容器已经包含这两个包了，所以我将作用范围设置为provided

## 项目部署

传统的部署方法是将项目打包，然后将项目拷贝到tomcat容器下面的webapp中；并且对于每次修改，我们多要进行项目的打包，然后将打包的项目拷贝到webapp目录下，过程非常的复杂。那么maven有没有好的方法来解决此复杂的过程呢？答案是肯定，它解决的方式是通过插件。下面我主要介绍一下将项目发布到jetty和tomcat中的方法。

* Jetty容器发布方法

1. 在pom.xml中加入jetty的配置文件

在pom.xml中加入jetty插件的配置，如下：

<plugin>

<groupId>org.mortbay.jetty</groupId>

<artifactId>jetty-maven-plugin</artifactId>

<configuration>

<!-- 每隔10秒扫描一次 主要用于热部署 -->

<scanIntervalSeconds>10</scanIntervalSeconds>

<webApp>

<!-- 路径 -->

<contextPath>/user</contextPath>

</webApp>

<connectors>

<connector implementation=*"org.eclipse.jetty.server.nio.SelectChannelConnector"*>

<!-- 端口配置 -->

<port>8787</port>

<maxIdleTime>60000</maxIdleTime>

</connector>

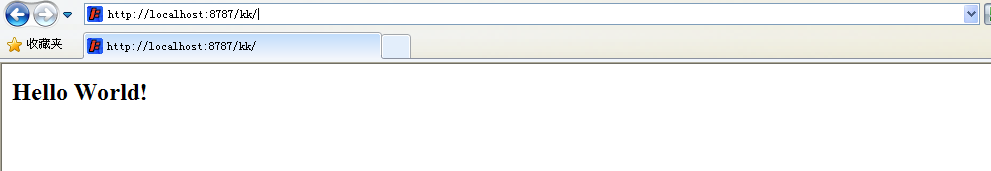
</connectors>

</configuration>

</plugin>

1. 运行jetty:run

运行jetty:run后，我们通过地址<http://localhost:8787/kk/>即可访问我们部署的项目，效果如下：



* Tomcat容器部署

对于tomcat容器部署，在这里我采用的是cargo这个插件，[Cargo](http://cargo.codehaus.org/)是一个基于标准方式发布Web应用程序到Java EE容器的瘦包装器，它能发布到多种Web服务器，如Geronimo、Glassfish、JBoss、Tomcat、Jetty、WebLogic等。

[Cargo](http://cargo.codehaus.org/)包含如下工具和API：

* + 启动/停止/配置Java容器和部署模块到Java容器的API。
  + 解析/创建/合并Java EE模块的API。
  + Ant 任务, Maven 1、 Maven 2 插件。
  + 以插件形式集成在Intellij IDEA 和Netbeans中。

      你可以通过配置您的pom.xml来决定您是否需要本地或者远程部署到您的tomcat服务器。比较遗憾的是cargo的官方文档写得不是很详细。

通过cargo你可以部署到很多web服务器中，如果没有配置服务器，默认情况下它会发布到jetty服务器中，看下面这段非常简单的cargo插件配置：

<build>

<plugins>

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

</plugin>

</plugins>

</build>

这段配置我没有配置web服务器，cargo默认会将项目部署到jetty中，通过cargo：start启动服务；cargo:stop停止服务。默认端口是8080，通过这个地址即可访问项目：<http://localhost:8080/user_web/>。下面我们重点来看看我们用的比较多的tomcat服务器如何部署。

1. 将项目部署至本地web容器

Cargo支持两种本地部署的方式，分别为standalone和existing模式。在standalone模式下，cargo会从web容器的安装目录复制一份到用户指定的目录，然后在此基础上进行部署应用，每次重新构建的时候，这个目录会被清空，所有配置被重新生成。Existing模式则需要用户指定web容器配置目录，然后cargo会将项目部署到此web容器中。

* Standalone模式

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.3.1</version>

<configuration>

<!--以下标记将详细描述你想要部署的容器-->

<container>

<!--

指定"tomcat6x"这个名称非常的重要!因为它是Cargo中一个完整的标识符,如果你把"x"去掉的话,

而只用"tomcat6"的话,它就不能在Tomcat 6.x下正常工作.

-->

<containerId>tomcat6x</containerId>

<home>F:\Tomcat6</home><!--本地Tomcat安装目录 -->

</container>

<!-- 容器的属性 -->

<configuration>

<!-- 如果你在容器属性中指定类型为"installed"的话,你就必须在如下配置type==existing和重配一次安装目录-->

<type>existing</type>

<home>${project.build.directory}/tomcat6x</home>

<properties>

<cargo.servlet.port>8080</cargo.servlet.port>

</properties>

</configuration>

<!-- 准备执行部署 -->

</configuration>

</plugin>

* Existing模式

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.3.1</version>

<configuration>

<!--以下标记将详细描述你想要部署的容器-->

<container>

<!--

指定"tomcat6x"这个名称非常的重要!因为它是Cargo中一个完整的标识符,如果你把"x"去掉的话,

而只用"tomcat6"的话,它就不能在Tomcat 6.x下正常工作.

-->

<containerId>tomcat6x</containerId>

<home>F:\Tomcat6</home><!--本地Tomcat安装目录 -->

</container>

<!-- 容器的属性 -->

<configuration>

<!-- 如果你在容器属性中指定类型为"installed"的话,你就必须在如下配置type==existing和重配一次安装目录-->

<type>existing</type>

<home>F:\Tomcat6</home>

<properties>

<cargo.servlet.port>8080</cargo.servlet.port>

</properties>

</configuration>

<!-- 准备执行部署 -->

</configuration>

</plugin>

通过上述这两种任意一种配置，然后通过命令cargo:start和cargo:stop即可启动和停止服务。

1. 部署至远程web服务器

我们只需要在pom.xml中加入下面这段cargo插件配置即可：

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.3.1</version>

<configuration>

<wait>true</wait>

<container>

<containerId>tomcat6x</containerId>

<type>remote</type><!-- 远程部署类型必须是remote -->

</container>

<!-- 容器的属性 -->

<configuration>

<!-- 如果你在容器属性中指定类型为"remote"的话,那么你必须在如下配置type==runtime -->

<type>runtime</type>

<properties>

<cargo.hostname>localhost</cargo.hostname> <!--主机名,默认localhost -->

<cargo.servlet.port>8888</cargo.servlet.port> <!-- 端口号,默认值 8080 -->

<cargo.tomcat.manager.url>

http://localhost:8888/manager/html/

</cargo.tomcat.manager.url>

<cargo.remote.username>tomcat</cargo.remote.username>

<cargo.remote.password>tomcat</cargo.remote.password>

<cargo.tomcat.ajp.port>8009</cargo.tomcat.ajp.port>

</properties>

</configuration>

<!-- 准备执行部署 -->

</configuration>

</plugin>

远程是命令跟本地有所不同，远程用cargo:deploy部署或者cargo:redeploy重新部署。

1. 在cargo插件中指定要部署的项目

<deployer>

<!-- 类型设置必须跟容器一致 -->

<type>remote</type>

<deployables>

<!--在deployable中指定你要部署的webapp -->

<deployable>

<!--可选项,Ping目标地址判断项目地址发布成功或者失败 -->

<pingURL>

http://localhost:8888/user\_web

</pingURL>

<!--可选项,Ping目标地址的响应时间,默认20000毫秒 -->

<pingTimeout>300000</pingTimeout>

<groupId>com.marcle.user</groupId>

<artifactId>user\_web</artifactId>

<type>war</type>

<properties>

<context>userWeb</context>

</properties>

</deployable>

</deployables>

</deployer>

通过deployer元素即可在插件中部署特定的maven项目

1. 将cargo插件与生命周期绑定

通过前面的实验我们可以知道，我们并没有将cargo插件与特定的生命周期绑定，我们需要通过cargo中的命令来部署我们的项目，我们希望是将cargo插件与特定生命周期绑定，只要我们执行生命周期中的某个阶段，就可以把项目进行发布。我们来看下面这段配置：

<executions>

<execution><!-- 通过"install"目标部署到容器并运行-->

<id>verify-deploy</id>

<phase>install</phase>

<goals><goal>deployer-redeploy</goal>

</goals>

</execution>

<execution><!--通过"pre-clean"目标取消部署并关闭容器-->

<id>clean-undeploy</id>

<phase>pre-clean</phase>

<goals><goal>deployer-undeploy</goal>

</goals>

</execution>

</executions>

这样我们通过install的时候，即可将项目部署到服务器上，下面是pom.xml整个配置：

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>cargo-maven2-plugin</artifactId>

<version>1.3.1</version>

<configuration>

<wait>true</wait>

<container>

<containerId>tomcat6x</containerId>

<type>remote</type><!-- 远程部署类型必须是remote -->

</container>

<!-- 容器的属性 -->

<configuration>

<!-- 如果你在容器属性中指定类型为"remote"的话,那么你必须在如下配置type==runtime -->

<type>runtime</type>

<properties>

<cargo.hostname>localhost</cargo.hostname> <!--主机名,默认localhost -->

<cargo.servlet.port>8888</cargo.servlet.port> <!-- 端口号,默认值 8080 -->

<cargo.tomcat.manager.url>

http://localhost:8888/manager/html/

</cargo.tomcat.manager.url>

<cargo.remote.username>tomcat</cargo.remote.username>

<cargo.remote.password>tomcat</cargo.remote.password>

<cargo.tomcat.ajp.port>8009</cargo.tomcat.ajp.port>

</properties>

</configuration>

<!-- 准备执行部署 -->

<deployer>

<!-- 类型设置必须跟容器一致 -->

<type>remote</type>

<deployables>

<!--在deployable中指定你要部署的webapp -->

<deployable>

<!--可选项,Ping目标地址判断项目地址发布成功或者失败 -->

<pingURL>

http://localhost:8888/user\_web

</pingURL>

<!--可选项,Ping目标地址的响应时间,默认20000毫秒 -->

<pingTimeout>300000</pingTimeout>

<groupId>com.marcle.user</groupId>

<artifactId>user\_web</artifactId>

<type>war</type>

<properties>

<context>userWeb</context>

</properties>

</deployable>

</deployables>

</deployer>

</configuration>

<executions>

<execution><!-- 通过"install"目标部署到容器并运行-->

<id>verify-deploy</id>

<phase>install</phase>

<goals><goal>deployer-redeploy</goal>

</goals>

</execution>

<execution><!--通过"pre-clean"目标取消部署并关闭容器-->

<id>clean-undeploy</id>

<phase>pre-clean</phase>

<goals><goal>deployer-undeploy</goal>

</goals>

</execution>

</executions>

</plugin>

# 站点发布

Maven的站点发布功能非常强大，如果我们定义的规范，它会帮我们生成一系列的报告及文档，使用起来非常的方便，当然maven的站点发布也是基于插件形式的，我们需要生成什么样的报告，我们引入什么样的插件即可。像源码、测试覆盖率报告，大家只要在配置文件中引入maven-javadoc-plugin和cobertura-maven-plugin即可。我们先来看看站点发布是怎么用的。

在pom.xml中引入下面这段配置：

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-site-plugin</artifactId>

<version>3.0</version>

<configuration>

<locales>zh\_CN</locales>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

然后在maven命令中输入site，此插件会帮助我们生成一个默认的站点报告。如下图：



如果想要增加站点内容，或者减少项目信息报告，我需要怎么做呢？

* 1. 在项目根目录下（与根pom同级）新建src\site目录，其下新建site.xml文件；
  2. Site里面需要编写什么内容，大家可以参考官网的写法，<http://maven.apache.org/plugins/maven-site-plugin/examples/sitedescriptor.html>，我里面的内容为：

<?xml version=*"1.0"* encoding=*"ISO-8859-1"*?>

<project name=*"Maven"*>

<bannerLeft>

<name>Maven</name>

<src>http://maven.apache.org/images/apache-maven-project.png</src>

<href>http://maven.apache.org/</href>

</bannerLeft>

<bannerRight>

<src>http://maven.apache.org/images/maven-small.gif</src>

</bannerRight>

<body>

<links>

<item name=*"Apache"* href=*"http://www.apache.org/"* />

<item name=*"Maven 1.x"* href=*"http://maven.apache.org/maven-1.x/"*/>

<item name=*"Maven 2"* href=*"http://maven.apache.org/"*/>

</links>

<menu name=*"Maven 2.0"*>

<item name=*"Introduction"* href=*"index.html"*/>

<item name=*"Download"* href=*"download.html"*/>

<item name=*"Release Notes"* href=*"release-notes.html"* />

<item name=*"General Information"* href=*"about.html"*/>

<item name=*"For Maven 1.x Users"* href=*"maven1.html"*/>

<item name=*"Road Map"* href=*"roadmap.html"* />

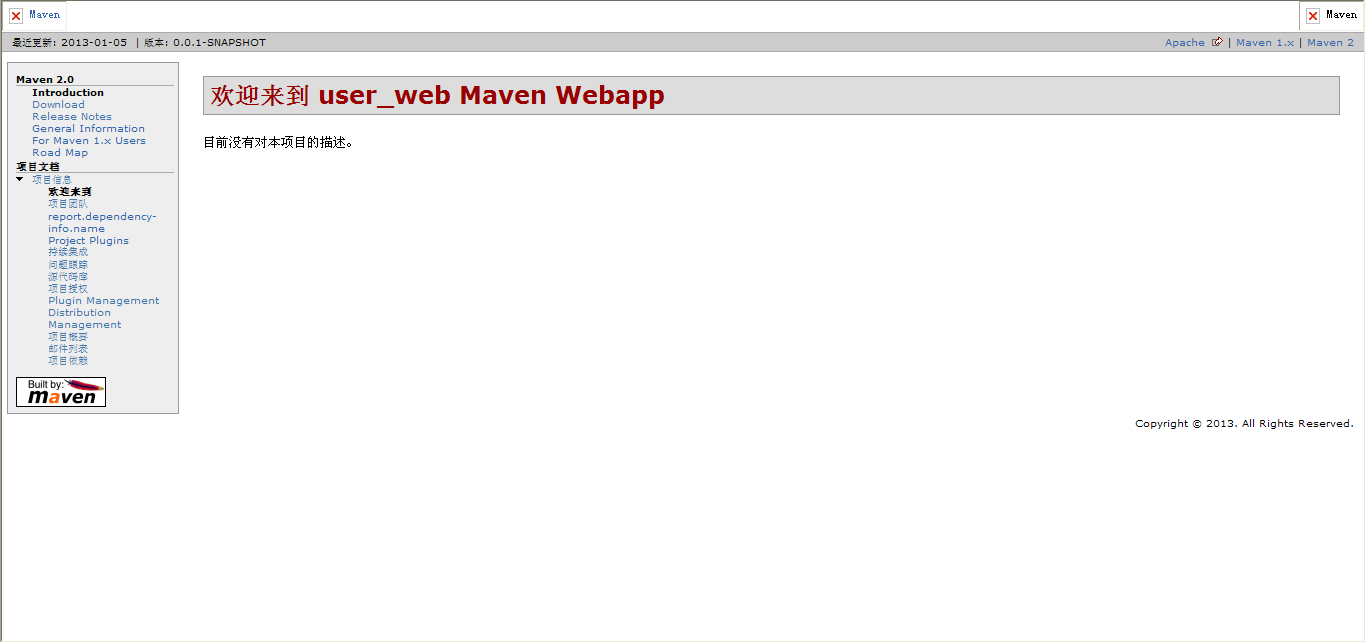
</menu>

<menu ref=*"reports"*/>

</body>

</project>

3、再次生成站点，大家可以看到下面的页面：



从上面的配置中我们可以看到很多报告，很多报告可能我们用不到，那么怎么定义我们需要的报告呢？在pom.xml中加入下面这段配置：

<reporting>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-project-info-reports-plugin</artifactId>

<version>2.1.2</version>

<reportSets>

<reportSet>

<reports>

<report>index</report>

<report>summary</report>

<report>project-team</report>

<report>mailing-list</report>

<report>cim</report>

<report>issue-tracking</report>

<report>license</report>

<report>scm</report>

</reports>

</reportSet>

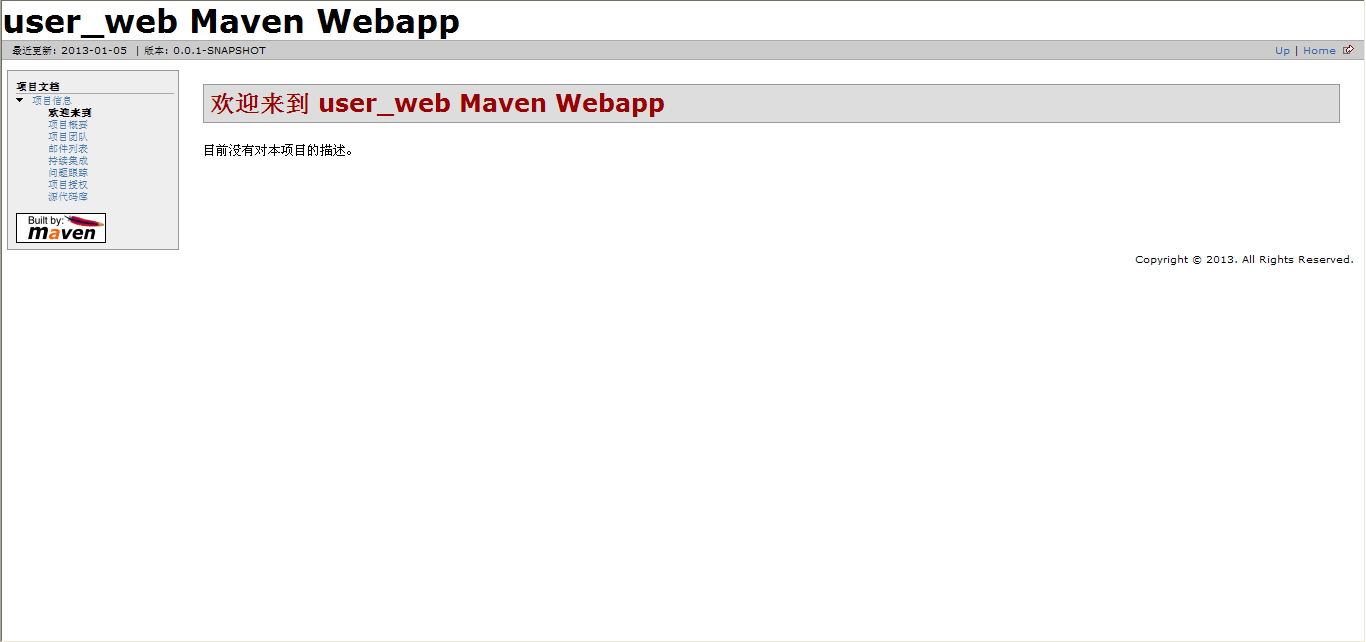
</reportSets>

</plugin>

</plugins>

</reporting>

在reports中定义了我们需要的报告。我们再看看生成站点：



那么我们如何引入其它插件，生成其它报告呢？很简单，我们只需要在在reporting中引入插件即可，下面我们引入一个源码生成的插件看看：

<plugin>

                    <groupId>org.apache.maven.plugins</groupId>

                    <artifactId>maven-javadoc-plugin</artifactId>

                    <version>2.8.1</version>

                 </plugin>

把这段插件配置配置到reporting中，这样生成站点的时候，就可以生成项目的api，用起来非常的简单易用。我在网上找到一篇对大部分插件的一个说明，<http://springsfeng.iteye.com/blog/1457916>大家可以看看。

如果我们运行site:run那么site插件默认会将站点部署到jetty容器下面，大家可以直接通过地址可以对生成的站点进行访问。对于其它的web容器，需要对配置做相应修改，大家可以参考官网。