Maven

# Maven基本概念

在不用Maven的时候，比如说以前我们用Ant构建项目，在项目目录下，往往会看到一个名为/lib的子目录，那里存放着各类第三方依赖jar文件，如log4j.jar，junit.jar等等。每建立一个项目，你都需要建立这样的一个/lib目录，然后复制一对jar文件，这是很明显的重复。重复永远是噩梦的起点，多个项目不共用相同的jar文件，不仅会造成磁盘资源的浪费，也使得版本的一致性管理变得困难。此外，如果你使用版本管理工具，如github（你没有使用版本管理工具？马上试试SVN吧，它能帮你解决很多头疼的问题），你需要将大量的jar文件提交到代码库里，可是版本管理工具在处理二进制文件方面并不出色。

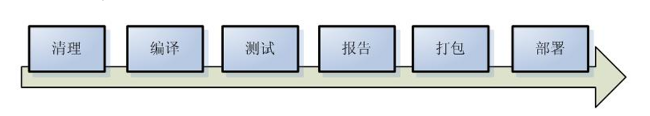
Maven仓库就是放置所有JAR文件（WAR，ZIP，POM等等）的地方，所有Maven项目可以从同一个Maven仓库中获取自己所需要的依赖JAR，这节省了磁盘资源。此外，由于Maven仓库中所有的JAR都有其自己的坐标，该坐标告诉Maven它的组ID，构件ID，版本，打包方式等等，因此Maven项目可以方便的进行依赖版本管理。你也不在需要提交JAR文件到SCM仓库中，你可以建立一个组织层次的Maven仓库，供所有成员使用。

简言之，Maven仓库能帮助我们管理构件（主要是JAR）。

Maven是一个跨平台的项目管理工具，它是基于项目对象模型(POM)，主要用于基于java平台的项目构建，依赖管理，以通过一小段描述信息来管理项目的构建，报告和文档的软件项目管理工具。

## 项目构建

项目构建过程包括【清理项目】→【编译项目】→【测试项目】→【生成测试报告】→【打包项目】→【部署项目】这几个步骤，这六个步骤就是一个项目的完整构建过程。



理想的项目构建是高度自动化，跨平台，可重用的组件，标准化的，使用maven就可以帮我们完成上述所说的项目构建过程。

## 依赖管理

依赖指的是jar包之间的相互依赖，比如我们搭建一个Struts2的开发框架时，光光有struts2-core-2.3.16.3.jar这个jar包是不行的，struts2-core-2.3.16.3.jar还依赖其它的jar包，依赖管理指的就是使用Maven来管理项目中使用到的jar包，Maven管理的方式就是“自动下载项目所需要的jar包，统一管理jar包之间的依赖关系”。

## 使用Maven的好处

Maven中使用约定，约定java源代码代码必须放在哪个目录下，编译好的java代码又必须放到哪个目录下，这些目录都有明确的约定。

Maven的每一个动作都拥有一个生命周期，例如执行 mvn install 就可以自动执行编译，测试，打包等构建过程

只需要定义一个pom.xml,然后把源码放到默认的目录，Maven帮我们处理其他事情

使用Maven可以进行项目高度自动化构建，依赖管理(这是使用Maven最大的好处)，仓库管理。

## 为什么要使用maven；

一个基本web项目是从视图层（H5，CSS，Js等前端技术）到控制层（SpringMVC，Struts2）到事务处理层（Spring IOC，AOP）再到持久层（SpringData，Hibernate，Mybatis）最后到数据库（Mysql，Oracle，Mongodb等）。咦！！！好像没有Maven什么事？

但我们试想：

①如果给一个项目添加 jar 包，我们是不是手动COPY到WEB-INF/lib目录下的？

②如果 jar 包之间发生依赖问题和版本冲突？

③如果项目多了，相同 jar 包占用的存储空间会越来越大，我们是不是要犯强迫症了？

怎么解决？

借助Maven，使其以一种规范的方式下载设置的jar包

Maven在下载jar包的同时，还会自动将被依赖的jar导入

可以设置一个Maven pom父项目来管理jar包，让其他项目继承它

当项目越来越复杂，规模越来越大的时候，Maven的作用就会越来越明显。

Maven是一款服务于Java平台用的自动化构建工具，同时它也是用Java编写的。

# Maven下载

　　下载地址：https://maven.apache.org

得到一个压缩包，解压，可以看到maven的组成目录，maven路径最好不要包括中文

Maven目录分析

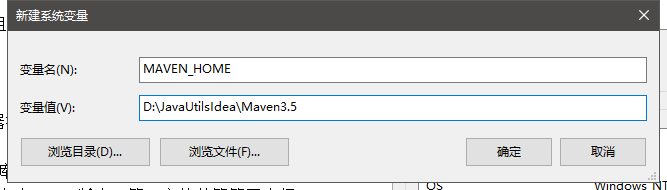
* bin：含有mvn运行的脚本
* boot：含有plexus-classworlds类加载器框架
* conf：含有settings.xml配置文件
* lib：含有Maven运行时所需要的java类库
* LICENSE.txt, NOTICE.txt, README.txt针对Maven版本，第三方软件等简要介绍

# Maven安装

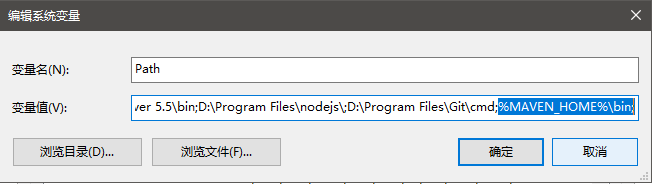
1. 首先要确保电脑上已经安装了JDK(要jdk 1.6+的版本)，配置好JDK的环境变量
2. 对下载的压缩文件进行解压缩

解压到如下目录(**解压目录最好不要有中文**)

1. 设置系统环境变量：MAVEN\_HOME



1. 设置环境变量Path，将%MAVEN\_HOME%\bin加入Path中，一定要注意要用分号；与其他值隔开，如下图所示：

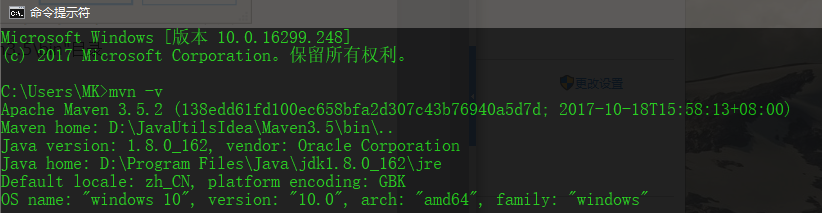


**%MAVEN\_HOME%\bin**代表的就是"D:\JavaUtilsIdea\Maven3.5\bin"目录

1. **验证Maven安装是否成功**

　　打开cmd窗口

　　输入"mvn –v"命令 查看Maven的相关信息，如下图所示：



　　能够出现这样的信息就说明Maven的安装已经成功了。

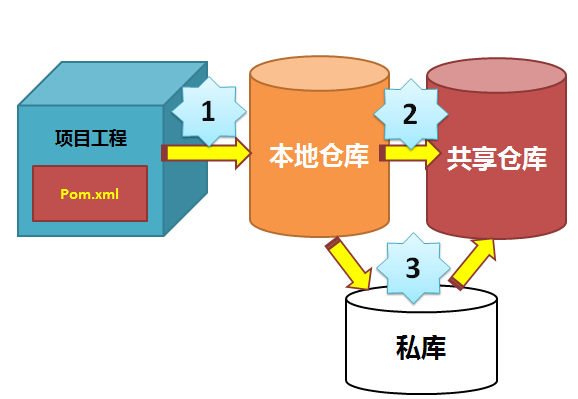
1. 设置MAVEN\_OPTS环境变量(可选配置)

　　由于Maven命令实际上是执行了Java命令，所以可以通过JAVA命令参数的方式来设置MAVEN运行参数。MAVEN\_OPTS环境变量正是用于此用途

　　MAVEN\_OPTS

　　-Xms128m -Xmx512m，分别设置JVM的最小和最大内存

# Mven的简单使用



**4.1 修改settings.xml文件，指定本机任意目录为本地仓库**

<localRepository>H:/maven\_repository</localRepository>

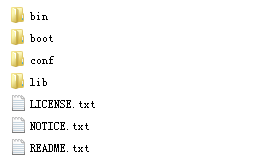
指定仓库的路径

在这里mavenRepository就是仓库的路径

配置镜像仓库：加速构建

|  |
| --- |
| <mirror>  <id>nexus-aliyun</id>  <mirrorOf>\*</mirrorOf>  <name>Nexus aliyun</name>  <url>http://maven.aliyun.com/nexus/content/groups/public</url>  </mirror> |

## Maven的配置



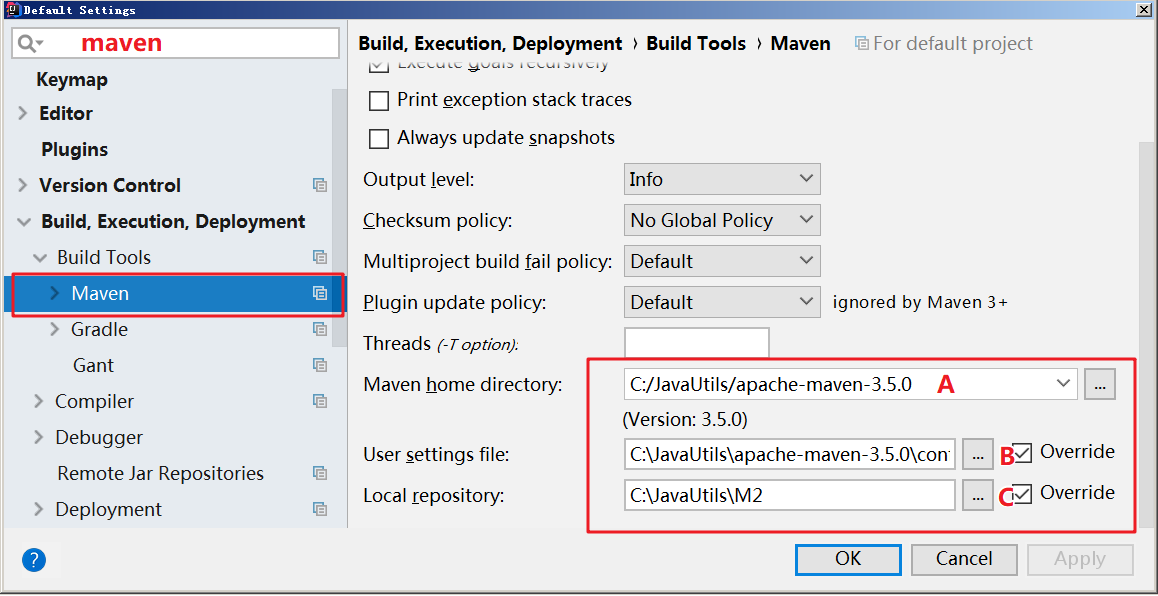
说明：

bin中存放可执行的二进制文件

conf存放settings.xml文件

lib 运行maven所依赖的jar包

将配置好的maven插件集成到idea里面

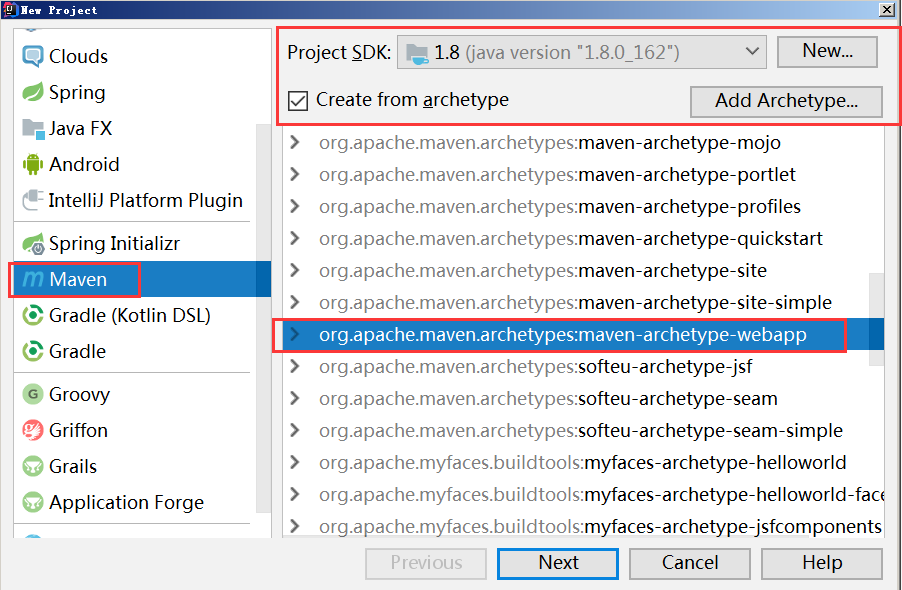


A：选择maven插件的home目录

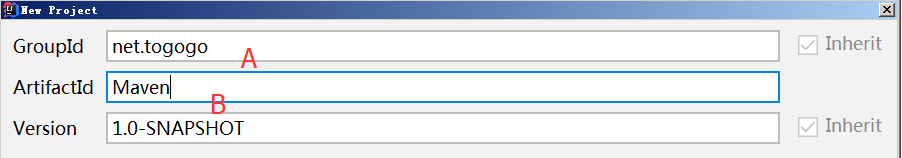
B：选中maven插件home目录下面的conf目录下面的setting.xml文件

C：如过B配置正确 C会自动填写本地仓库路径！

在idea里面创建maven工程：

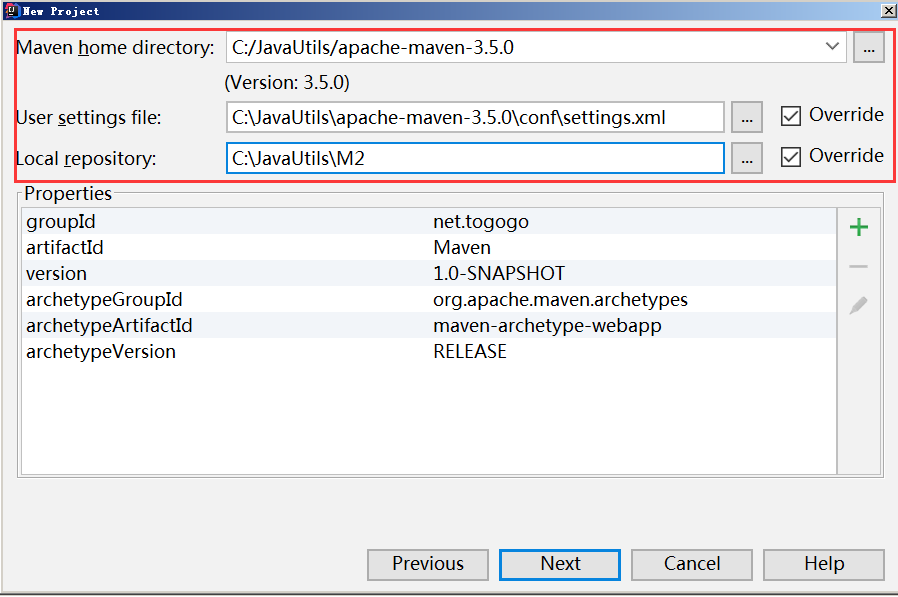


注意：一定要连接网络（保持网络畅通）

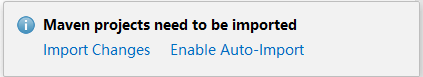


A：工程目录

B：工程名称



校验maven插件使用是否正确，不正确手动修改（针对当前项目有效）



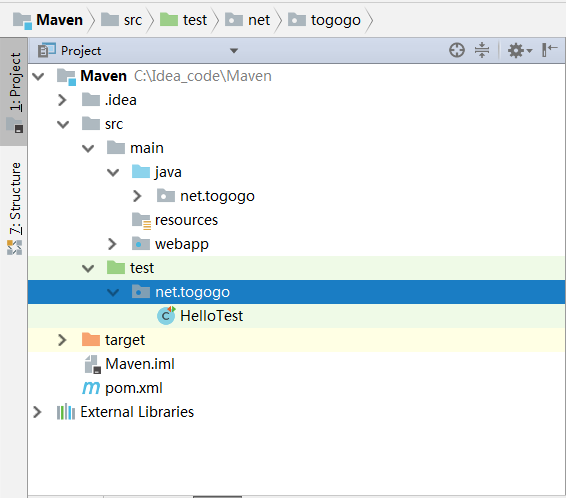
自动导入包

## Maven项目的目录约定

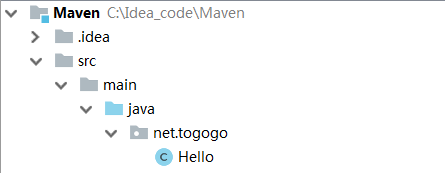
MavenProjectRoot(项目根目录)  
   |----src  
   |     |----main  
   |     |         |----java ——存放项目的.java文件  
   |     |         |----resources ——存放项目资源文件，如spring, hibernate配置文件  
   |     |----test  
   |     |         |----java ——存放所有测试.java文件，如JUnit测试类  
   |     |         |----resources ——存放项目资源文件，如spring, hibernate配置文件  
   |----target ——项目输出位置  
   |----pom.xml ----用于标识该项目是一个Maven项目

## 手动创建Maven项目，使用Maven

### 在idea建立一个项目Hello

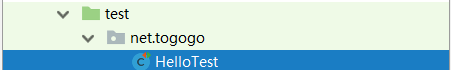


### 编写Hello类



|  |
| --- |
| **public class** Hello {  **public** String sayHello(String name){  **return "你好啊，"**+ name;  } } |

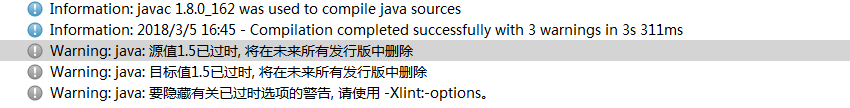
### 在src/test/java中创建一个包net.togogo创建一个测试类HelloTest



### 编写测试类

|  |
| --- |
| **public class** HelloTest {  @Test  **public void** helloTest(){  Hello hello = **new** Hello();  System.***out***.println(hello.sayHello(**"togogo"**));  } } |

### 提示异常：



解决办法：默认使用jdk1.5改成1.8即可

|  |
| --- |
| <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  <**encoding**>UTF-8</**encoding**>  </**configuration**> </**plugin**> |

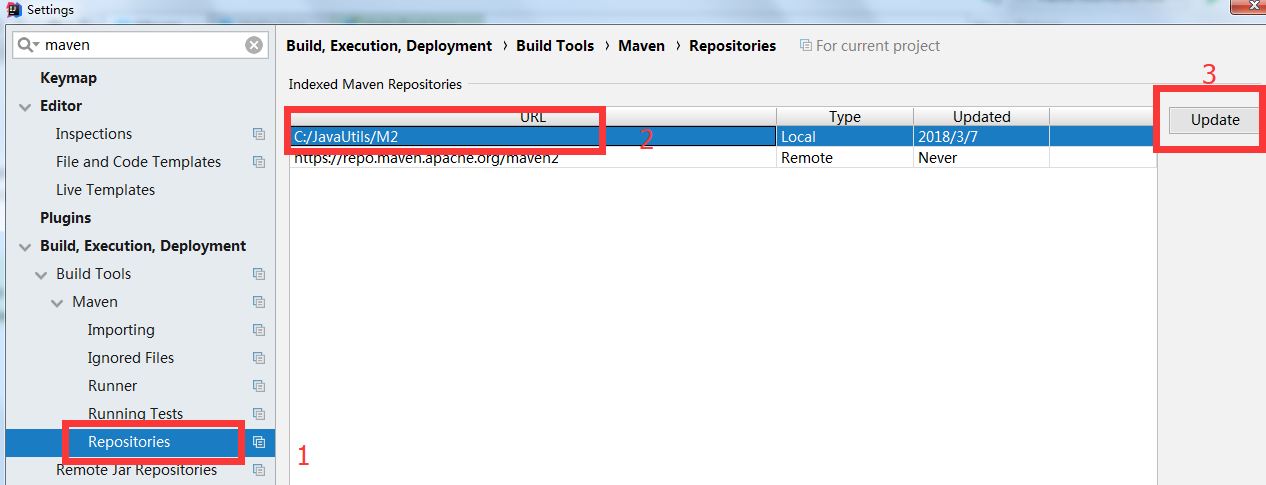
### 在项目根目录下编辑pom.xml文件

|  |
| --- |
| <**project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>net.togogo</**groupId**>  <**artifactId**>Maven</**artifactId**>  <**packaging**>war</**packaging**>  <**version**>1.0-SNAPSHOT</**version**>  <**name**>Maven Maven Webapp</**name**>  <**url**>http://maven.apache.org</**url**>   <**properties**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  <**scope**>test</**scope**>  </**dependency**>  </**dependencies**>   <**build**>  <**finalName**>Maven</**finalName**>  </**build**> </**project**> |

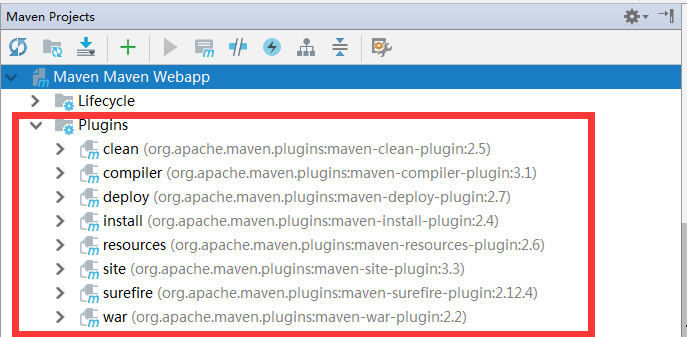
说明：

* project：表示一个工程
* modelVersion：为版本号

编写pom.xml文件获得自动提示（前提是仓库中已经存在此包）



### 使用maven命令



Clean:构建完成后清理

Compiler:编译Java源代码。

Deploy:将构建的工件部署到远程存储库。

Install:将构建的工件安装到本地存储库中。

Resources:将资源复制到输出目录以包含在JAR中。

Site:为当前项目生成项目报告，站点，一个网站。

Surefire:在隔离的类加载器中运行JUnit单元测试。

War:从当前项目构建WAR

Validate：验证 确保当前配置与pom的内容是有效的。

Verify：检验 检验可部署归档的有效性和完整性。

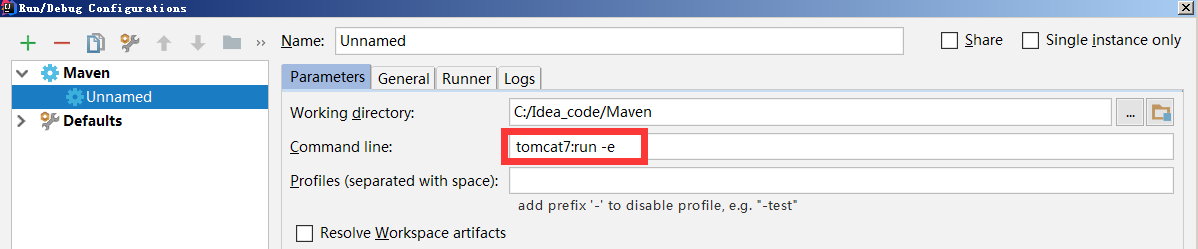
Maven工具栏使用：



配置maventomcat插件

|  |
| --- |
| <**plugins**>  *<!-- 配置Tomcat插件 -->* <**plugin**>  <**groupId**>org.apache.tomcat.maven</**groupId**>  <**artifactId**>tomcat7-maven-plugin</**artifactId**>  <version>2.2</version>  <**configuration**>  *<!-- 这里配置端口号和访问路径 -->* <**port**>8080</**port**>  <**path**>/</**path**>  </**configuration**>  </**plugin**> </**plugins**> |

使用tomcat7插件：



Maven生命周期：

Maven它有一个十分完善的生命周期模型(lifecycle)，这个生命周期可以从两方面来理解，第一，顾名思义，运行Maven的每个步骤都由它来定义的，这种预定义的默认行为使得我们使用Maven变得简单，相比而言，Ant的每个步骤都要你手工去定义。第二，这个模型是一种标准，在不同的项目中，使用Maven的接口是一样的，这样就不用去仔细理解每个项目的构建了，一般情况下，**mvn clean install** 这样的命令是通用的。

Maven有三套相互独立的生命周期：

* 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。

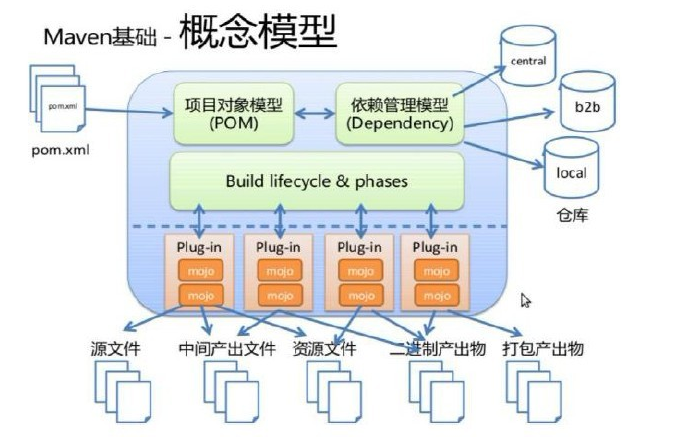
它们是相互独立的，你可以仅仅调用clean来清理工作目录，仅仅调用site来生成站点。

可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

|  |
| --- |
| 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 |
| * 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 |

# maven的核心概念

## 项目对象模型

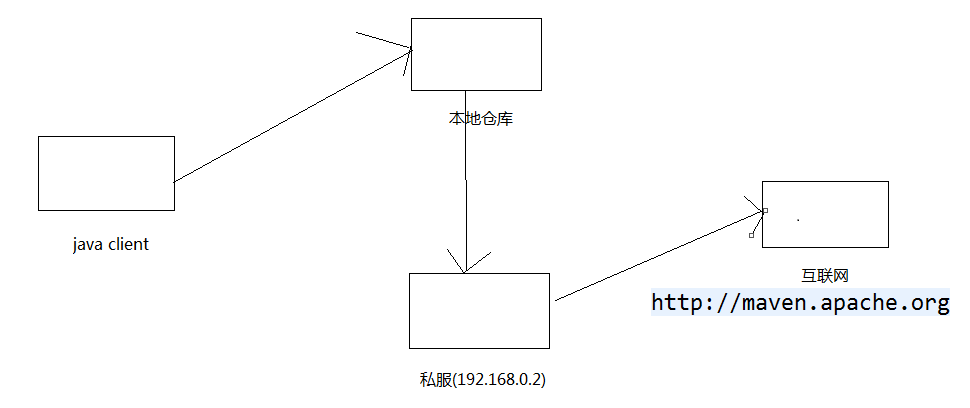


说明：

maven根据pom.xml文件，把它转化成项目对象模型(POM)，这个时候要解析依赖关系，然后去相对应的maven库中查找到依赖的jar包。

在clean，compile，test，package等阶段都有相应的Plug-in来做这些事情。

而这些plug-in会产生一些中间产物。



### maven坐标的主要组成

groupId：定义当前maven项目属于哪个项目

artifactId：定义实际项目中的某一个模块

version：定义当前项目的当前版本

packaging：定义当前项目的打包方式

根据这些坐标，在maven库中可以找到唯一的jar包

## 依赖管理

### scope的选值

在POM 4中，<dependency>中还引入了<scope>，它主要管理依赖的部署。目前<scope>可以使用5个值：   
  
\* compile，缺省值，适用于所有阶段，会随着项目一起发布。   
\* provided，类似compile，期望JDK、容器或使用者会提供这个依赖。如servlet.jar。   
\* runtime，只在运行时使用，如JDBC驱动，适用运行和测试阶段。   
\* test，只在测试时使用，用于编译和运行测试代码。不会随项目发布。   
\* system，类似provided，需要显式提供包含依赖的jar，Maven不会在Repository中查找它。

依赖范围控制哪些依赖在哪些classpath 中可用，哪些依赖包含在一个应用中。让我们详细看一下每一种范围：

**compile （编译范围）**

compile是默认的范围；如果没有提供一个范围，那该依赖的范围就是编译范围。编译范围依赖在所有的classpath 中可用，同时它们也会被打包。

**provided （已提供范围）**

provided 依赖只有在当JDK 或者一个容器已提供该依赖之后才使用。例如， 如果你开发了一个web 应用，你可能在编译 classpath 中需要可用的Servlet API 来编译一个servlet，但是你不会想要在打包好的WAR 中包含这个Servlet API；这个Servlet API JAR 由你的应用服务器或者servlet 容器提供。已提供范围的依赖在编译classpath （不是运行时）可用。它们不是传递性的，也不会被打包。

**runtime （运行时范围）**

runtime 依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如，你可能在编译的时候只需要JDBC API JAR，而只有在运行的时候才需要JDBC  
驱动实现。

**test （测试范围）**

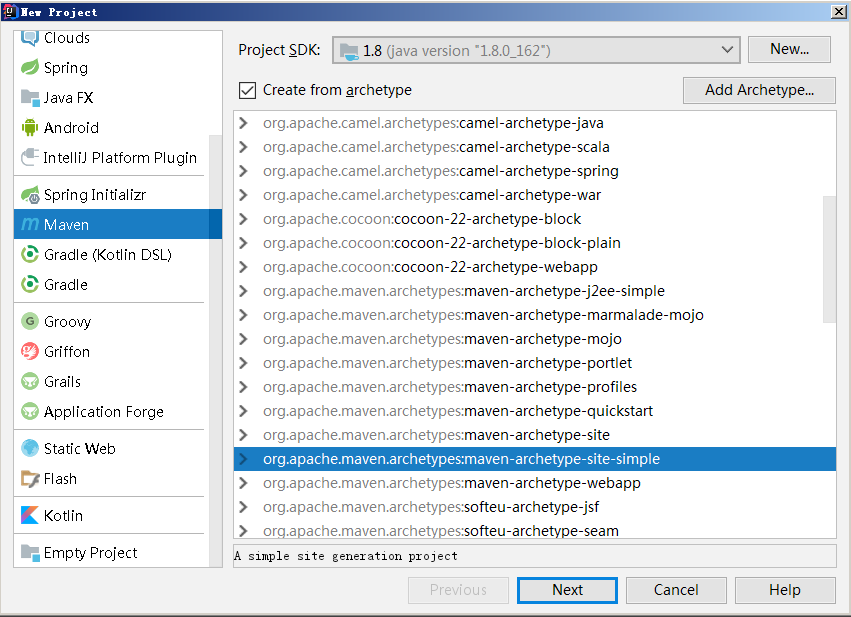
test范围依赖 在一般的编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用。

**system （系统范围）**

system范围依赖与provided 类似，但是你必须显式的提供一个对于本地系统中JAR 文件的路径。这么做是为了允许基于本地对象编译，而这些对象是系统类库的一部分。这样的构件应该是一直可用的，Maven 也不会在仓库中去寻找它。如果你将一个依赖范围设置成系统范围，你必须同时提供一个 systemPath 元素。注意该范围是不推荐使用的（你应该一直尽量去从公共或定制的 Maven 仓库中引用依赖）。

## 传递依赖

聚合工程：



新建好的父类工程

## 仓库管理

可以根据maven坐标定义每一个jar包在仓库中的存储位置。

大致为：groupId/artifactId/version/artifactId-version.packaging.jar

### 仓库的分类

本地仓库

~/.m2/repository/

每一个用户也可以拥有一个本地仓库

远程仓库

中央仓库：Maven默认的远程仓库

http://repo1.maven.org/maven2

私服：是一种特殊的远程仓库，它是架设在局域网内的仓库

镜像：用来替代中央仓库，速度一般比中央仓库快

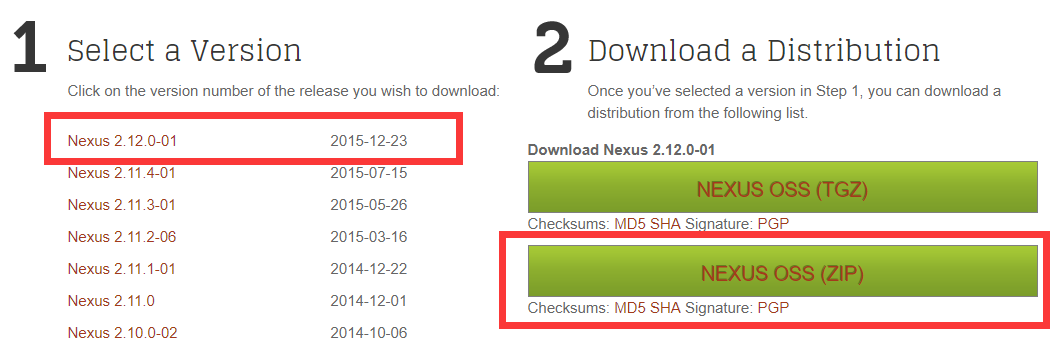
### 配置私服

下载nexus：

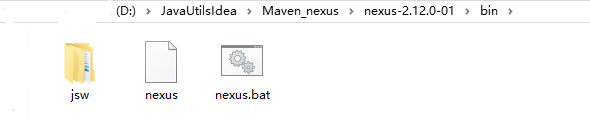
Nexus是maven仓库管理器，通过nexus可以搭建maven仓库，nexus还提供了强大的管理功能，构建搜索等功能；

下载地址：<http://www.sonatype.org/nexus/archived/#step2top>

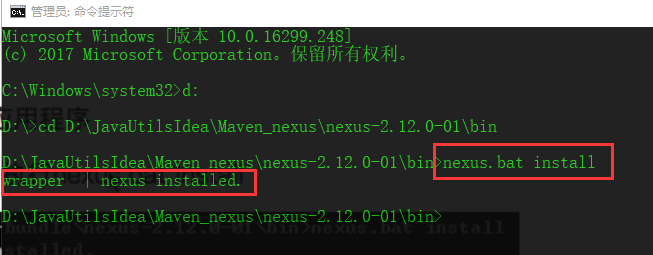
https://sonatype-download.global.ssl.fastly.net/nexus/oss/nexus-2.12.0-01-bundle.zip



下载后解压到任意目录 进入bin目录



使用管理员身份执行nexus.bat install



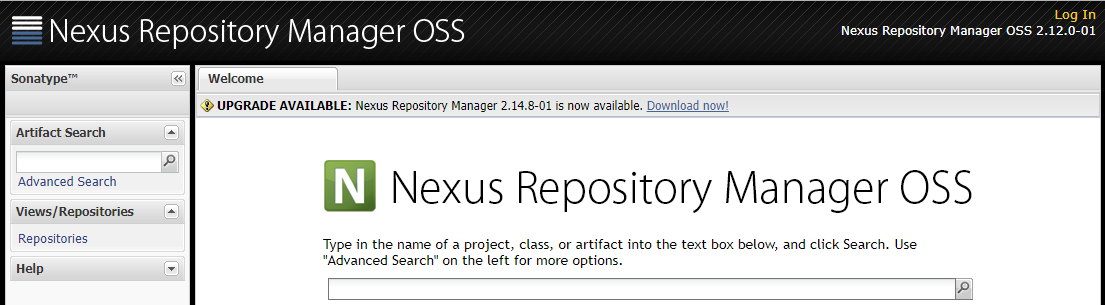
打开服务



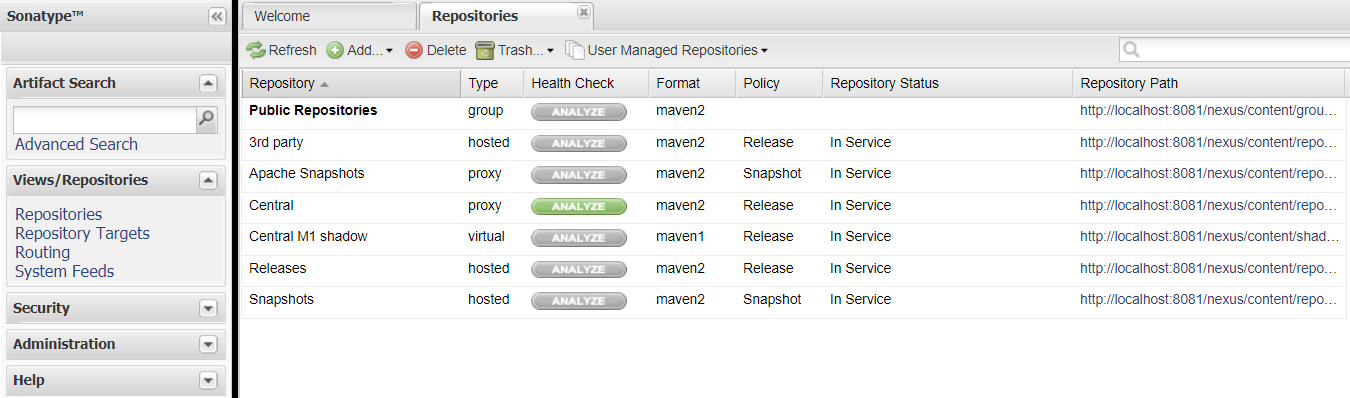
启动服务

|  |
| --- |
| 卸载nexus：  执行nexus.bat uninstall |

浏览器访问http://localhost:8081/nexus/#welcome

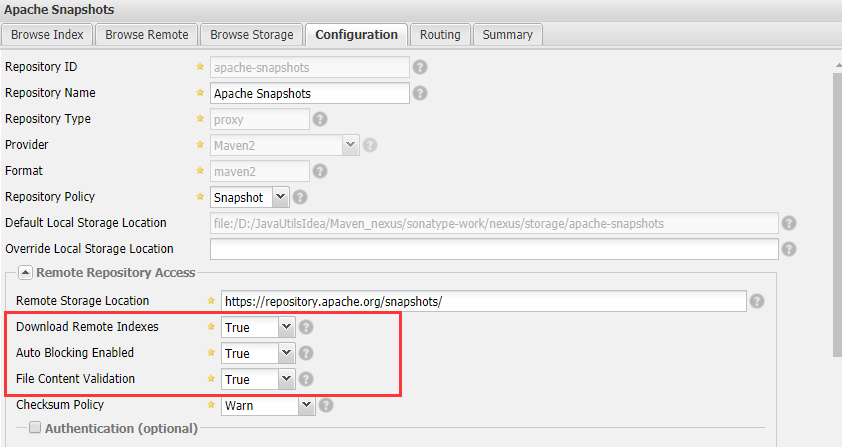


点击nexus主要右上角的 Log In按钮，管理员初始密码为 admin/admin123  登陆进去之点击 左侧的 【Repositories】菜单，会看的如下几个仓库

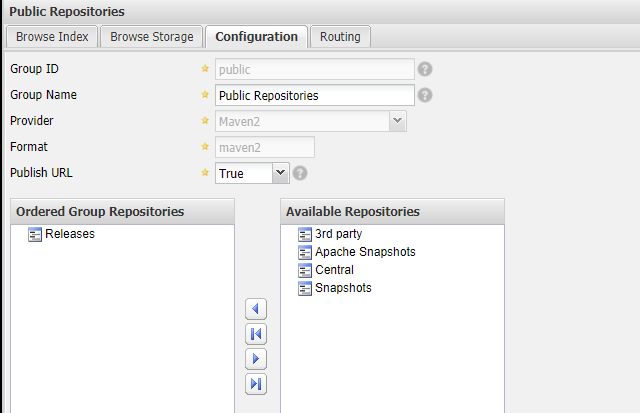


简单配置：

点击Apache Snapshots仓库，进行如下配置，开启远程索引下载，点击save按钮：



将现有的仓库，Apache snapshots ,3dpart,central 等仓库 加入 public 组中，直接在在界面中 将右侧的仓库移左边 保存



在maven中私用nexus：

修改setting文件：将public组地址直接配置到镜像里面即可！

|  |
| --- |
| <mirror>  <id>nexus</id>  <mirrorOf>central</mirrorOf>  <name>internal nexus repository</name>  <url>http://localhost:8081/nexus/content/groups/public/</url>  </mirror> |

配置好后就会通过私服下载。停用私服直接把私服路径去掉即可

# Maven仓库

## 本地仓库 vs. 远程仓库

运行Maven的时候，Maven所需要的任何构件都是直接从本地仓库获取的。如果本地仓库没有，它会首先尝试从远程仓库下载构件至本地仓库，然后再使用本地仓库的构件。

比如说，你的项目配置了junit-3.8的依赖，在你运行**mvn test** 的时候，Maven需要使用junit-3.8的jar文件，它首先根据坐标查找本地仓库，如果找到，就直接使用。如果没有，Maven会检查可用的远程仓库配置，然后逐个尝试这些远程仓库去下载junit-3.8的jar文件，如果远程仓库存在该文件，Maven会将其下载到本地仓库中，继而使用。如果尝试过所有远程仓库之后，Maven还是没能够下载到该文件，它就会报错。

Maven缺省的本地仓库地址为${user.home}/.m2/repository 。也就是说，一个用户会对应的拥有一个本地仓库。

你也可以自定义本地仓库的位置，修改${user.home}/.m2/settings.xml ：

**Xml代码**



1. **<settings>**
2. ...
3. **<localRepository>**D:\java\repository**</localRepository>**
4. ...
5. **</settings>**

你还可以在运行时指定本地仓库位置：

**mvn clean install -Dmaven.repo.local=/home/juven/myrepo/**

还有一点需要理解的是，当我们运行install的时候，Maven实际上是将项目生成的构件安装到了本地仓库，也就是说，只有install了之后，其它项目才能使用此项目生成的构件。

了解了本地仓库，接着了解一下Maven缺省的远程仓库，即Maven中央仓库。

安装好Maven之后，我们可以建立一个简单的项目，配置一些简单的依赖，然后运行mvn clean install，项目就构建好了。我们没有手工的去下载任何jar文件，这一切都是因为Maven中央仓库的存在，当Maven在本地仓库找不到需要的jar文件时，它会查找远程仓库，而一个原始的Maven安装就自带了一个远程仓库——Maven中央仓库。

这个Maven中央仓库是在哪里定义的呢？在我的机器上，我安装了maven-2.0.10，我可以找到这个文件：${M2\_HOME}/lib/maven-2.0.10-uber.jar ，打开该文件，能找到超级POM：\org\apache\maven\project\pom-4.0.0.xml ，它是所有Maven POM的父POM，所有Maven项目继承该配置，你可以在这个POM中发现如下配置：

**Xml代码**



1. **<repositories>**
2. **<repository>**
3. **<id>**central**</id>**
4. **<name>**Maven Repository Switchboard**</name>**
5. **<layout>**default**</layout>**
6. **<url>**http://repo1.maven.org/maven2**</url>**
7. **<snapshots>**
8. **<enabled>**false**</enabled>**
9. **</snapshots>**
10. **</repository>**
11. **</repositories>**

关于远程仓库的配置，下面的小节我会详细解释，这里我们只要知道，中央仓库的id为central，远程url地址为http://repo1.maven.org/maven2，它关闭了snapshot版本构件下载的支持。

## 在POM中配置远程仓库

前面我们看到超级POM配置了ID为central的远程仓库，我们可以在POM中配置其它的远程仓库。这样做的原因有很多，比如你有一个局域网的远程仓库，使用该仓库能大大提高下载速度，继而提高构建速度，也有可能你依赖的一个jar在central中找不到，它只存在于某个特定的公共仓库，这样你也不得不添加那个远程仓库的配置。

这里我配置一个远程仓库指向中央仓库的中国镜像：

**Xml代码**



1. **<project>**
2. ...
3. **<repositories>**
4. **<repository>**
5. **<id>**maven-net-cn**</id>**
6. **<name>**Maven China Mirror**</name>**
7. **<url>**http://maven.net.cn/content/groups/public/**</url>**
8. **<releases>**
9. **<enabled>**true**</enabled>**
10. **</releases>**
11. **<snapshots>**
12. **<enabled>**false**</enabled>**
13. **</snapshots>**
14. **</repository>**
15. **</repositories>**
16. **<pluginRepositories>**
17. **<pluginRepository>**
18. **<id>**maven-net-cn**</id>**
19. **<name>**Maven China Mirror**</name>**
20. **<url>**http://maven.net.cn/content/groups/public/**</url>**
21. **<releases>**
22. **<enabled>**true**</enabled>**
23. **</releases>**
24. **<snapshots>**
25. **<enabled>**false**</enabled>**
26. **</snapshots>**
27. **</pluginRepository>**
28. **</pluginRepositories>**
29. ...
30. **</project>**

我们先看一下<repositories>的配置，你可以在它下面添加多个<repository> ，每个<repository>都有它唯一的ID，一个描述性的name，以及最重要的，远程仓库的url。此外，<releases><enabled>true</enabled></releases>告诉Maven可以从这个仓库下载releases版本的构件，而<snapshots><enabled>false</enabled></snapshots>告诉Maven不要从这个仓库下载snapshot版本的构件。禁止从公共仓库下载snapshot构件是推荐的做法，因为这些构件不稳定，且不受你控制，你应该避免使用。当然，如果你想使用局域网内组织内部的仓库，你可以激活snapshot的支持。

关于<repositories>的更详细的配置及相关解释，请参考：http://www.sonatype.com/books/maven-book/reference\_zh/apas02s08.html。

至于<pluginRepositories>，这是配置Maven从什么地方下载插件构件（Maven的所有实际行为都由其插件完成）。该元素的内部配置和<repository>完全一样，不再解释。

## 在settings.xml中配置远程仓库

我们知道了如何在POM中配置远程仓库，但考虑这样的情况：在一个公司内部，同时进行这3个项目，而且以后随着这几个项目的结束，越来越多的项目会开始；同时，公司内部建立一个Maven仓库。我们统一为所有这些项目配置该仓库，于是不得不为每个项目提供同样的配置。问题出现了，这是重复 ！

其实我们可以做到只配置一次，在哪里配置呢？就是settings.xml。

不过事情没有那么简单，不是简单的将POM中的<repositories>及<pluginRepositories>元素复制到settings.xml中就可以，setting.xml不直接支持 这两个元素。但我们还是有一个并不复杂的解决方案，就是利用profile，如下：

**Xml代码**



1. **<settings>**
2. ...
3. **<profiles>**
4. **<profile>**
5. **<id>**dev**</id>**
6. <!-- repositories and pluginRepositories here-->
7. **</profile>**
8. **</profiles>**
9. **<activeProfiles>**
10. **<activeProfile>**dev**</activeProfile>**
11. **</activeProfiles>**
12. ...
13. **</settings>**

这里我们定义一个id为dev的profile，将所有repositories以及pluginRepositories元素放到这个profile中，然后，使用<activeProfiles>元素自动激活该profile。这样，你就不用再为每个POM重复配置仓库。

使用profile为settings.xml添加仓库提供了一种用户全局范围的仓库配置。

## 镜像

如果你的地理位置附近有一个速度更快的central镜像，或者你想覆盖central仓库配置，或者你想为所有POM使用唯一的一个远程仓库（这个远程仓库代理的所有必要的其它仓库），你可以使用settings.xml中的mirror配置。

以下的mirror配置用maven.net.cn覆盖了Maven自带的central：

**Xml代码**



1. **<settings>**
2. ...
3. **<mirrors>**
4. **<mirror>**
5. **<id>**maven-net-cn**</id>**
6. **<name>**Maven China Mirror**</name>**
7. **<url>**http://maven.net.cn/content/groups/public/**</url>**
8. **<mirrorOf>**central**</mirrorOf>**
9. **</mirror>**
10. **</mirrors>**
11. ...
12. **</settings>**

这里唯一需要解释的是<mirrorOf>，这里我们配置central的镜像，我们也可以配置一个所有仓库的镜像，以保证该镜像是Maven唯一使用的仓库：

**Xml代码**



1. **<settings>**
2. ...
3. **<mirrors>**
4. **<mirror>**
5. **<id>**my-org-repo**</id>**
6. **<name>**Repository in My Orgnization**</name>**
7. **<url>**http://192.168.1.100/maven2**</url>**
8. **<mirrorOf>**\***</mirrorOf>**
9. **</mirror>**
10. **</mirrors>**
11. ...
12. **</settings>**

关于更加高级的镜像配置，可以参考：http://maven.apache.org/guides/mini/guide-mirror-settings.html。

## 分发构件至远程仓库

**mvn install** 会将项目生成的构件安装到本地Maven仓库，**mvn deploy** 用来将项目生成的构件分发到远程Maven仓库。本地Maven仓库的构件只能供当前用户使用，在分发到远程Maven仓库之后，所有能访问该仓库的用户都能使用你的构件。

我们需要配置POM的distributionManagement来指定Maven分发构件的位置，如下：

**Xml代码**



1. **<project>**
2. ...
3. **<distributionManagement>**
4. **<repository>**
5. **<id>**nexus-releases**</id>**
6. **<name>**Nexus Release Repository**</name>**
7. **<url>**http://127.0.0.1:8080/nexus/content/repositories/releases/**</url>**
8. **</repository>**
9. **<snapshotRepository>**
10. **<id>**nexus-snapshots**</id>**
11. **<name>**Nexus Snapshot Repository**</name>**
12. **<url>**http://127.0.0.1:8080/nexus/content/repositories/snapshots/**</url>**
13. **</snapshotRepository>**
14. **</distributionManagement>**
15. ...
16. **</project>**

Maven区别对待release版本的构件和snapshot版本的构件，snapshot为开发过程中的版本，实时，但不稳定，release版本则比较稳定。Maven会根据你项目的版本来判断将构件分发到哪个仓库。

一般来说，分发构件到远程仓库需要认证，如果你没有配置任何认证信息，你往往会得到401错误。这个时候，如下在settings.xml中配置认证信息：

**Xml代码**



1. **<settings>**
2. ...
3. **<servers>**
4. **<server>**
5. **<id>**nexus-releases**</id>**
6. **<username>**admin**</username>**
7. **<password>**admin123**</password>**
8. **</server>**
9. **<server>**
10. **<id>**nexus-snapshots**</id>**
11. **<username>**admin**</username>**
12. **<password>**admin123**</password>**
13. **</server>**
14. **</servers>**
15. ...
16. **</settings>**

需要注意的是，settings.xml中server元素下id的值必须与POM中repository或snapshotRepository下id的值完全一致。将认证信息放到settings下而非POM中，是因为POM往往是它人可见的，而settings.xml是本地的。