[1. 概述 4](#_Toc343847991)

[1.1 Maven是什么 4](#_Toc343847992)

[1.2 Maven不是什么 5](#_Toc343847993)

[1.2.1 Maven不是ant，不是make 5](#_Toc343847994)

[1.2.2 Maven不是ivy 5](#_Toc343847995)

[1.3 Maven可以帮我们做什么 5](#_Toc343847996)

[1.3.1 Build 5](#_Toc343847997)

[1.3.2 依赖关系 6](#_Toc343847998)

[1.3.3 版本控制 6](#_Toc343847999)

[1.3.4 输出管理 6](#_Toc343848000)

[1.3.5 项目关系管理 7](#_Toc343848001)

[1.3.6 文档和构建结果 7](#_Toc343848002)

[1.4 Maven与敏捷开发 7](#_Toc343848003)

[1.4.1 简单 7](#_Toc343848004)

[1.4.2 沟通 7](#_Toc343848005)

[1.4.3 反馈 8](#_Toc343848006)

[1.4.4 TDD 8](#_Toc343848007)

[1.4.5 快速构建 8](#_Toc343848008)

[1.4.6 持续集成 8](#_Toc343848009)

[1.5 Maven使用结构图 8](#_Toc343848010)

[1.5.1 对于开发人员 9](#_Toc343848011)

[1.5.2 对于PM管理层 10](#_Toc343848012)

[2. Maven的安装配置 10](#_Toc343848013)

[2.1 Windows安装 10](#_Toc343848014)

[2.1.1 JDK安装 11](#_Toc343848015)

[2.1.2 Maven安装 11](#_Toc343848016)

[2.2 Linux安装 13](#_Toc343848017)

[2.2.1 JDK安装 13](#_Toc343848018)

[2.2.2 Maven安装 14](#_Toc343848019)

[2.3 Maven目录结构与相关配置 15](#_Toc343848020)

[2.3.1 M2\_HOME 15](#_Toc343848021)

[2.3.2 ~/.M2 16](#_Toc343848022)

[2.3.3 Eclipse插件安装 16](#_Toc343848023)

[2.3.4 MAVEN\_OPTS 19](#_Toc343848024)

[2.3.5 setting.xml 20](#_Toc343848025)

[3. Maven使用 23](#_Toc343848026)

[3.1 使用Maven建立项目 23](#_Toc343848027)

[3.1.1 准备与确认 24](#_Toc343848028)

[3.1.2 开始啦 25](#_Toc343848029)

[3.2 坐标 42](#_Toc343848030)

[3.3 依赖 44](#_Toc343848031)

[3.3.1 依赖配置 44](#_Toc343848032)

[3.3.2 依赖范围 45](#_Toc343848033)

[3.3.3 传递依赖 45](#_Toc343848034)

[3.3.4 依赖调解 48](#_Toc343848035)

[3.3.5 可选依赖 48](#_Toc343848036)

[3.3.6 排除依赖 48](#_Toc343848037)

[3.3.7 归类依赖 49](#_Toc343848038)

[3.3.8 优化依赖 49](#_Toc343848039)

[3.4 仓库 52](#_Toc343848040)

[3.4.1 本地仓库 52](#_Toc343848041)

[3.4.2 远程仓库 53](#_Toc343848042)

[3.4.3 Maven中央仓库 53](#_Toc343848043)

[3.4.4 私服 54](#_Toc343848044)

[3.5 生命周期与插件 55](#_Toc343848045)

[3.5.1 三套生命周期 56](#_Toc343848046)

[3.5.2 命令行与生命周期 57](#_Toc343848047)

[3.5.3 插件目标 57](#_Toc343848048)

[4. 聚合与继承 58](#_Toc343848049)

[4.1 聚合 58](#_Toc343848050)

[4.2 继承 58](#_Toc343848051)

[4.3 聚合与继承 58](#_Toc343848052)

[4.4 约定优于配置 58](#_Toc343848053)

[5. Nexus私服 58](#_Toc343848054)

[5.1 Nexus安装 58](#_Toc343848055)

[5.1.1 Windows安装 59](#_Toc343848056)

[5.1.2 Linux安装 61](#_Toc343848057)

[5.2 Nexus的仓库 63](#_Toc343848058)

[5.3 建立宿主仓库 64](#_Toc343848059)

[5.4 建立代理仓库 65](#_Toc343848060)

[5.5 建立仓库组 66](#_Toc343848061)

[5.6 Nexus的索引与搜索 67](#_Toc343848062)

[5.7 配置Maven从Nexus下载构件 69](#_Toc343848063)

[5.8 部署构件到Nexus 69](#_Toc343848064)

[5.9 手动部署构件到Nexus 69](#_Toc343848065)

[5.10 为项目分配独立仓库 69](#_Toc343848066)

[5.11 权限管理 69](#_Toc343848067)

[6. Maven+SVN+Nexus的Web项目实例 69](#_Toc343848068)

[6.1 Maven项目的创建与配置 70](#_Toc343848069)

[6.1.1 创建Maven项目 70](#_Toc343848070)

[6.1.2 配置pom.xml 73](#_Toc343848071)

[6.1.3 运行Maven项目 74](#_Toc343848072)

[6.2 SVN管理与维护 76](#_Toc343848073)

[6.2.1 SVN check in 76](#_Toc343848074)

[6.2.2 SVN check out 78](#_Toc343848075)

[7. Hudson持续集成 81](#_Toc343848076)

[7.1 持续集成与敏捷开发 81](#_Toc343848077)

[7.2 Hudson安装 81](#_Toc343848078)

[7.3 Hudson用户与权限 81](#_Toc343848079)

[7.4 Hudson配置 81](#_Toc343848080)

[7.5 创建一个任务 81](#_Toc343848081)

[7.6 Hudson的工作目录 81](#_Toc343848082)

[7.7 邮件反馈 81](#_Toc343848083)

# 概述

本文描述了Maven、Nexus、SVN的集成使用，但对于Maven的聚合和继承不描述，Hudson的使用方式以后补充。

给大家明确一点，Maven、Nexus、SVN只是一种工具，我们不能按照学习某种编程语言一样，花费大量的时间和精力，或者去读一本厚厚的参考书籍，我们只需要知道工具如何帮我们解决问题就可以了。

我的原则是用到什么学什么，而不是为了学什么而用什么。所以在本文档中也体现了这一点，有许多Maven的东西我没有提到，因为我们没有用到，希望其他同事以后能够补充。

## Maven是什么

Maven中文意思为知识的积累，好吧，也可以翻译成专家。Maven是一个跨平台的项目管理工具。

提到项目管理工具我们想到什么？微软的Project，国内知名的开源产品zentao？No，Maven跟他们不一样，至少我还没感觉到他们有相似的地方。

Maven主要用于基于Java平台的项目构建、依赖管理及项目信息管理。也就是说不管是瀑布式开发，还是敏捷开发，或是DDD开发，我们都可以使用Maven。

## Maven不是什么

### Maven不是ant，不是make

ant和make需要编写脚本，Maven不需要，Maven使用约定优于配置的方法，一些开发常用的操作和步骤已经固化在Maven中。

比如，我们可以通过mvn clean compile命令来清理项目并编译，通过tomcat:run命令将项目发布到tomcat上运行。

也就是说从清理、编译、测试到打包部署，不需要写任何的脚本，而是通过插件来完成。

### Maven不是ivy

ivy依赖于ant，仅提供依赖管理功能，由上可知Maven不只是管理依赖，如果把ant和ivy放在一起，倒是可以跟Maven比较一下。

## Maven可以帮我们做什么

### Build

构建，Maven使用插件来支持多种应用程序的构建，比如在上面的mvn clean命令，如果我们看控制台输出就可以看到：[clean:clean]等字样，这里前面那个clean是插件，后面的clean是前面clean插件的目标。

### 依赖关系

之前我们使用springMVC，我们总是从spring的官方网站上下载下来一堆Jar包放到我们的项目里，我相信好多人都不知道哪些包是必须的，哪些包有什么作用，有时候一个包有好多个版本，开发人员自己都不知道哪个包有用，哪个包没用，都不敢删除，这令PM十分头疼。

OK，好事来了，Maven项目中必须存在一个pom.xml，我们要把我们用的到Jar包放到这个文件中，而且不用我们自己去下载这些Jar包啦，Maven可以帮我们下载，从哪里下载呢，这就是后面提到的Maven的中央仓库。

是不是特别像ivy？

### 版本控制

千万不要理解错了，这里的版本是指打包时候的版本，不是SVN的代码版本控制。Maven有自己的版本生成策略，这个后面我们可以讲到。

### 输出管理

项目构建的结果就是输出，Maven可以将一个项目输出为Jar包，也可以将一个项目打成War包。

如果我们的项目打成包，又能提供REST接口，会怎么样？

### 项目关系管理

假如有一个项目，其下有几个子项目，我们可以很方便的在pom.xml文件通过parent元素来指定父子关系。这个目前我们没有使用过。

### 文档和构建结果

Maven中的site插件支持javadoc、产品文档等各种信息的发布。

## Maven与敏捷开发

因为我一直想把敏捷开发带到项目管理中来，所以这次使用Maven也是为了这个目的，来看看Maven能够帮我们实现哪些极限编程中的核心价值。

### 简单

Maven暴露了一组一致、简洁的操作接口，在Eclipse下还有Maven插件，帮助我们使用成熟稳定的插件来简化构建系统的复杂度。

### 沟通

Maven与SVN集成使用，让团队中所有的开发人员都能知道最新一次构建的结果，以便于开发人员了解项目的状态，促进团队的交流与沟通。

### 反馈

Maven与Hudson集成，可以很快的完成项目的打包部署，让用户看到项目的最新情况，以此来得到用户的反馈。

### TDD

测试是Maven生命周期的一部分，并且Maven有成熟的插件支持流行的测试框架，如JUnit、TestNG。

### 快速构建

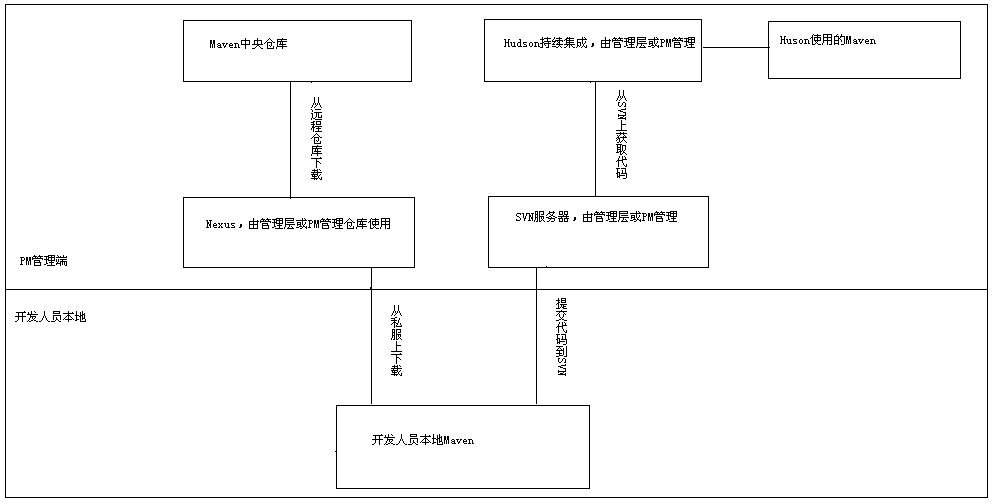
这是Maven比较擅长做的事，它可以快速地一次性的清理、编译、测试、打包、部署。

### 持续集成

Maven和Hudson集成，可以保证项目在很短的周期内集成最新的代码。持续集成我们在后面Hudson一章详细讲到。

## Maven使用结构图

下图是我们这边一直使用Maven的整体结构图。



### 对于开发人员

#### pom.xml文件

项目中多出了pom.xml文件，其中注明了依赖包的坐标，也就是说以后不再自己下载依赖的Jar包，而是要依赖于Maven本地仓库中的包。

如果Nexus私服上有这些包，则直接从Nexus上下载，如果Nexus上没有，则从Maven的中央仓库上来下载，下载完后会缓存到私服上。

#### SVN

代码编写完成之后，和之前一样，提交SVN，这点没有变化。

如果使用Hudson持续集成工具，或者使用服务器上的Maven进行编译，开发人员则只需提交代码和pom.xml文件，不再提交项目依赖的Jar包。

### 对于PM管理层

#### Nexus管理

为了避免大家都从Maven的中央仓库上去下载依赖包，才引进Nexus，需要PM管理员来管理，可以分不同的权限，分不同的项目来建立不同的组和镜像，这个我们在后面详细描述，另外还需要管理的是公司内部上传到Nexus的工具包。

#### SVN管理

SVN这点没有任何变化。

#### Maven或Hudson编译打包部署

目前杨跃组是使用ant打包，如果换成Maven，是一样，如果使用Hudson，工作量会减少很多。

# Maven的安装配置

Maven的安装配置非常简单，大家随我来。

## Windows安装

我这里使用的是Windows XP SP3，其它Windows环境相关不多。

### JDK安装

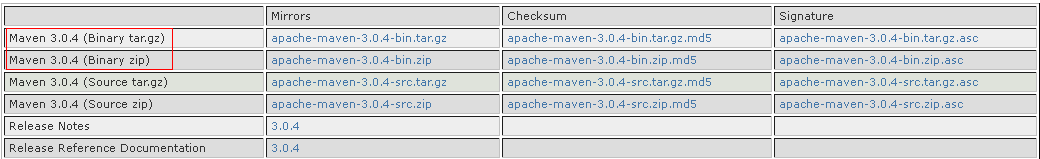
还用多说吗？好吧，强调一点，一定要设置Java的环境变量，强烈建议Windows下把JAVA\_HOME设置在系统环境变量里，然后在path中加入%JAVA\_HOME%\bin。记着安装设置好了一定要检查一下，如下图：



可以看到，我使用的是1.6.0\_30版本。

### Maven安装

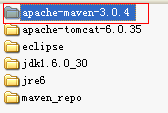
从<http://maven.apache.org/download.html>去下载最新的Maven，从这个页面中我们可以看到不同平台的版本：



OK，我们下载这两个，因为一会我们还要在Linux下安装配置。注意，我们使用的是Maven3.0.4版本，并不是Maven2。

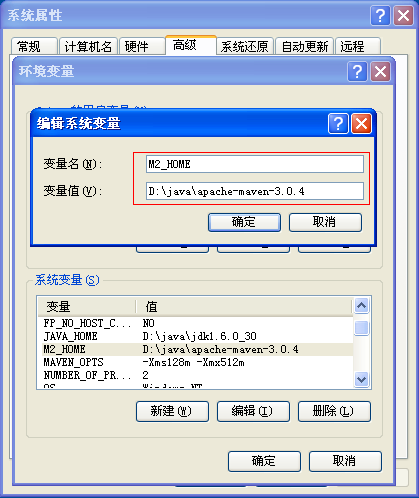
#### 解压文件

我们下载的是apache-maven-3.0.4-bin.zip包，用什么工具解压一下都可以。解压后如下图：



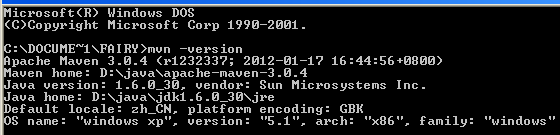
#### 配置环境变量

在我的电脑->右键->属性->高级->环境变量->系统变量中添加一个M2\_HOME，这个值指向刚刚解压的Maven目录，如下图：



OK，我们在path中加入%M2\_HOME%\bin，记着不要忘了分号。

打开一个命令行，试一下：



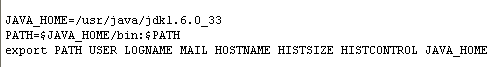
至此，Windows环境下Maven就安装完成了。

## Linux安装

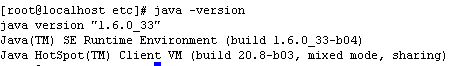
我这里的演示环境是CentOS6.0，其它Linux环境相差不多。

### JDK安装

这里仍然强调一点，一定要设置java环境变量，建议将JAVA\_HOME放到/etc/profile中去，如下图：



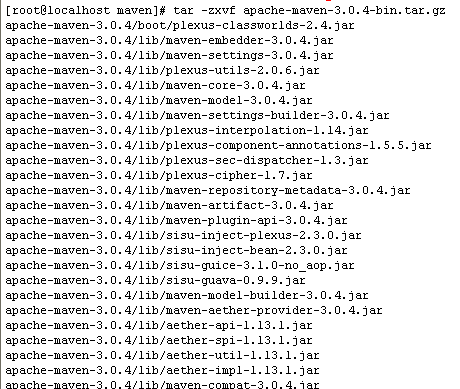
测试一下：



可以看到，我使用的版本是 1.6.0\_33。

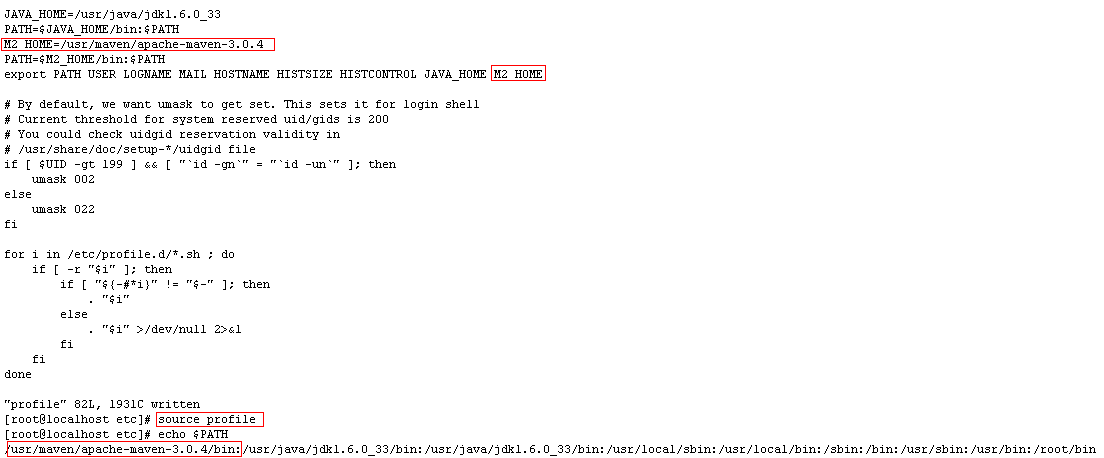
### Maven安装

#### 解压

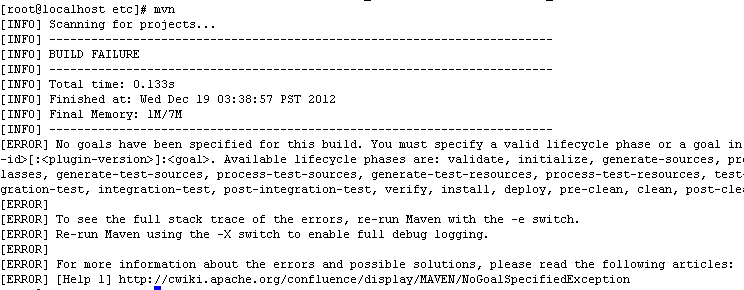


#### 设置环境变量

同JDK一样，把M2\_HOME及$M2\_HOME/bin放在/etc/profile文件中去：



OK，试一下：



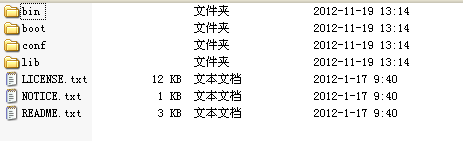
至此，Linux环境下Maven安装完成。

## Maven目录结构与相关配置

这里为了方便演示，以后所有的示例都是Windows平台。

### M2\_HOME

来看一下M2\_HOME目录，即Maven安装上当下的文件：



#### bin目录

这个目录包含了mvn运行的脚本，这些脚本用来配置Java命令，准备好classpath及相关的Java系统属性，然后执行Java命令。

mvn.bat在Windows下是bat批处理文件，在Linux下则是mvn，是shell脚本。

有兴趣的可以打开这些文件看一下。

#### boot目录

这个目录下只有一个文件：plexus-classworlds-2.4.jar，这个文件是个类加载器框架。

#### conf目录

这里有非常重要的settings.xml，修改这个文件就能全局定义Maven的行为。在这里我建议直接修改这个目录下的文件，而不是修改~/.m2/下的settings.xml文件，目的是为了在一台机器上统一。

#### lib目录

这个目录包含了Maven运行时需要的所有的Jar包。

### ~/.M2

~代表当前用户目录。这个目录下也有一个settings.xml，一般我们不使用这个配置文件，所以建议删除~/.M2这个目录。

### Eclipse插件安装

在Eclipse中安装Maven插件是为了开发人员能够在IDE中方便的使用Maven。

#### 安装

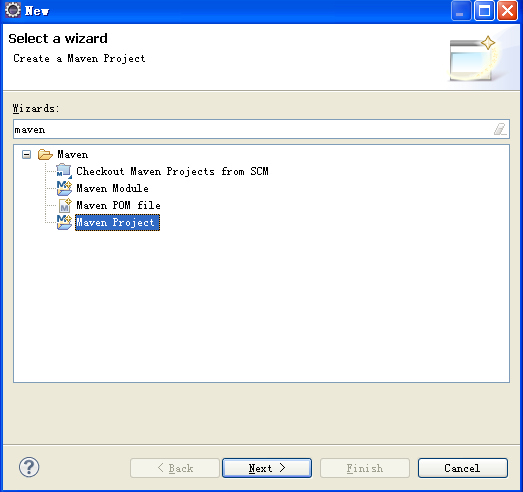
在Eclipse中选择Help菜单，选择Eclipse Marketplace…，可以先在Popular中查看是不是有Maven插件，运气好的话，就可以看到：



直接点击后面的Install按钮即可安装。

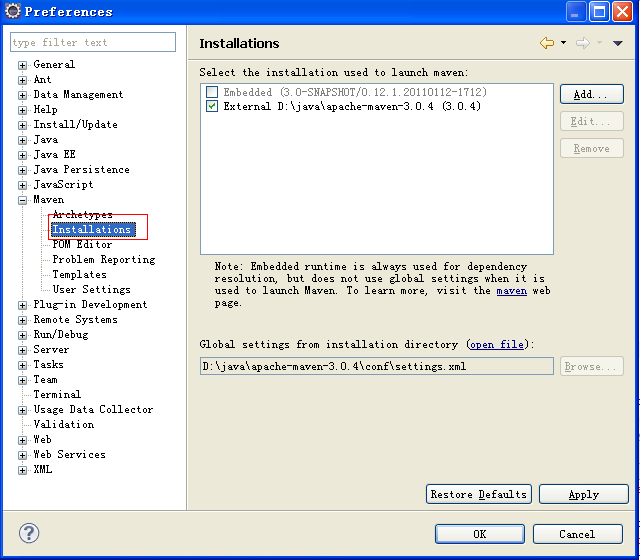
如果Popular，即流行中没有，则也可以看看Recent，即最近大家使用的，如果还没有，那么就只能使用search了，不过这个应该比较慢。

安装完成之后，我们可以通过Eclipse中File->new->Other…，通过在弹出的对话框中填写Maven，即可找到Maven相关项目信息：



#### 配置

在Eclipse中选择Window菜单，单击Preferences，看到那个Maven了吗？这里我们关心的是Installations：



我们不使用Eclipse内嵌的Maven，记住，一定不能使用。所以，我们点击Add…，选择我们Maven的安装路径，添加一个，并把上面内嵌的Maven前面的复选框中的勾去掉，点击OK关闭这个页面即完成配置。

### MAVEN\_OPTS

在2.3.1.1我们了解到，运行mvn命令其实是执行了java命令，而我们使用Maven生成站点需要大量的内存，如果我们不设置MAVEN\_OPTS，则很容易出现java.lang.OutOfMemoryError错误，尤其是在Hudson中，一定要设置这个环境变量。

在这里，我们有两种方式来设置这个变量，一种是在mvn.bat中设置，另外一种就是像设置M2\_HOME或JAVA\_HOME一样，设置成系统环境变量，我们强烈建议使用系统环境变量的方式来设置，而不使用修改mvn.bat或Linux下修改mvn的方式。

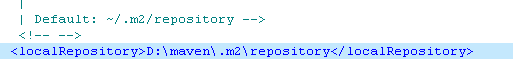
在Hudson中，Hudson有自己设置的方式，不需要设置成环境变量，这点一定注意，后面在Hudson一节讲到。

### settings.xml

终于到了这个重要而又简单的家伙了，重要是一定的，因为这个文件决定了Maven的行为，简单，是因为所有的配置都有说明，还有例子。我们找几个重要的说一下。

#### localRepository

看上面的注释就可以看出来，这个是用来配置本地仓库的，也就是从Maven的中央仓库或是从Nexus上下载的Jar包，放到本地什么位置。



看上面default的值，所以这里我们强烈建议把这个值设置，以免我们自己都不知道下载的Jar在什么位置。

#### proxies

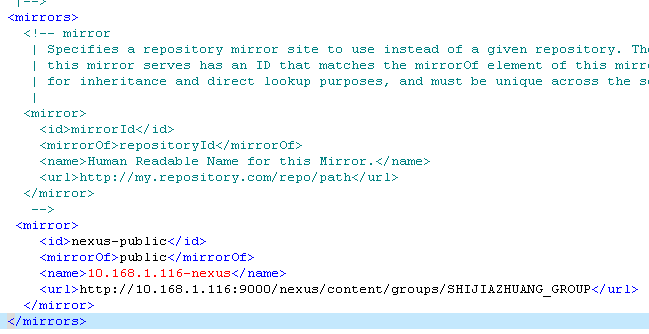
当我们公司使用代理上网时，才会设置一个上网的代理，目前来看，我们不需要设置。

#### servers

我们前面知道Maven可以从自己的中央仓库里下载Jar，这个中央仓库确实提供我们的项目用到的所有的Jar包，但在公网上不只一个这样的仓库，如果一些仓库需要认证，那么就需要配置server，后面在仓库一节，我们会详细说明。

#### mirror

如果仓库X可以提供Y仓库存储的所有内容，那么我们说X是Y的镜像，仓库的概念我们在后面详细说明。换句话说，任何一个从仓库Y可以获得的artifact，都能够从镜像X中获得。如果理解这一点，当我们说私服Nexus是所有仓库的镜像就不难理解了。如我的例子：



<http://10.168.1.116:9000/nexus/content/groups/SHIJIAZHUANG_GROUP>是我们Nexus私服的一个仓库组，这个概念我们后面详细说明，这个就是所有仓库，包含Maven中央仓库的镜像。

在上面还要注意一个mirrorof，这个我们在Nexus私服一节会详细说明。

为什么要使用镜像呢，很好理解，在地理位置上，X可能比Y离得我们更近，那么我们下载依赖包就快，当然我们自己架设的私服更快。

#### profile

profile本身是轮廓的意思，也就是说，我们可以在这里定义一些默认的东西，比如JDK的版本，当然这里定义的东西我们可以在pom.mxl里来覆盖。



这里我定义的是默认使用我们私服的镜像。

#### activeProfiles

这是激活默认的一些profile列表。

# Maven使用

## 使用Maven建立项目

这里我们在命令行下使用Maven的插件来建立项目，并不使用Eclipse这样的IDE来建立，目的是为了让大家先了解Maven的简单用法及相关概念。后面我们会使用Eclipse结合Maven、SVN、Nexus来建立一个相对真实的Web项目。

相信没人喜欢自己手动创建pom.xml，也没有人喜欢自己手动建立java的包结构，所以，我们使用archetype插件来生成一个项目骨架，然后在这个基础上我们完成项目功能。

Archetype，中文意思为骨架，在这里，archetype是Maven的一个插件。

### 准备与确认

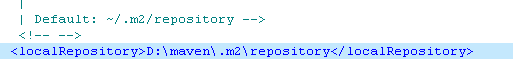
在开始之前我们先确认两个问题，一个是Maven版本，一个就是settings.xml里的仓库配置及本地仓库的位置。

#### Maven版本

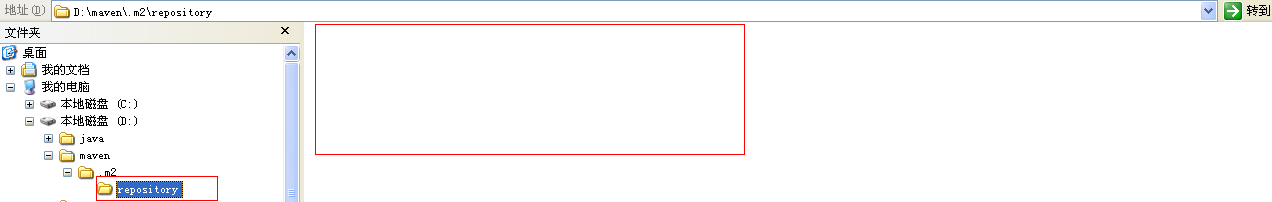
强调一下我们使用的是Maven3，而不是Maven2，两者在使用插件的时候有不同的地方，Maven2相对Maven3复杂一点，有兴趣的可以去试一下。

#### settings.xml中的本地仓库

还记得settings.xml里设置的那个localRepository吗？看下图：



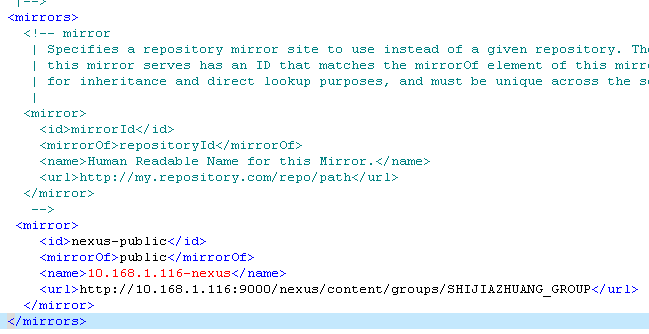
我们检查一下d:\maven\.m2\repository这个目录，看看是不是空的：



这个文件夹是空的，Maven会不会向这个文件夹下放东西我们一会可以看到。

#### settings.xml中的镜像

前面说过什么是镜像，我们再看一下settings.xml里的设置：



因为网络带宽的问题，这里我配置的是我们这边的私服，以方便演示，这里完全可以设置成Maven的中央仓库。

注意，我们至少要配置一个镜像，因为Maven要从仓库中下载依赖包和相关插件。

### 开始啦

因为我们已经把%M2\_HOME$\bin放到了环境变量里，则我们在命令行下键入如下命令：

mvn archetype:generate

先解释一下，archetype是Maven的插件，而generate则是这个插件的一个goal，即目标。

下面是比较详细的过程。

#### 有关下载

可以看到如下页面，并有很多的输出，我们随便截取一段来看：

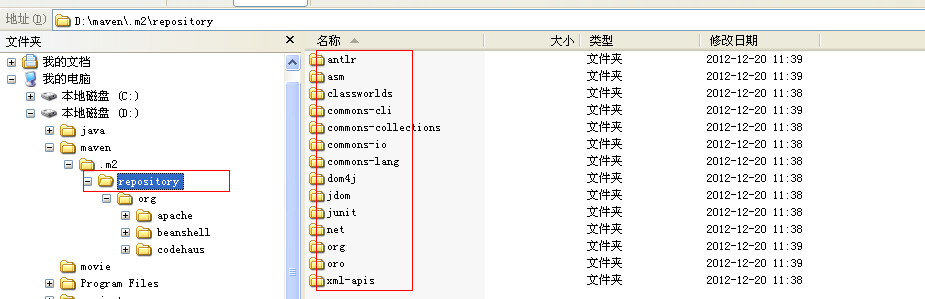


为什么我们使用一个简单的命令会有如此多的输出？Maven在做什么？好吧，它在下载东西，问题又来了，下载的是什么？从哪里下载？下载到什么位置？ 是每次都要下载吗？

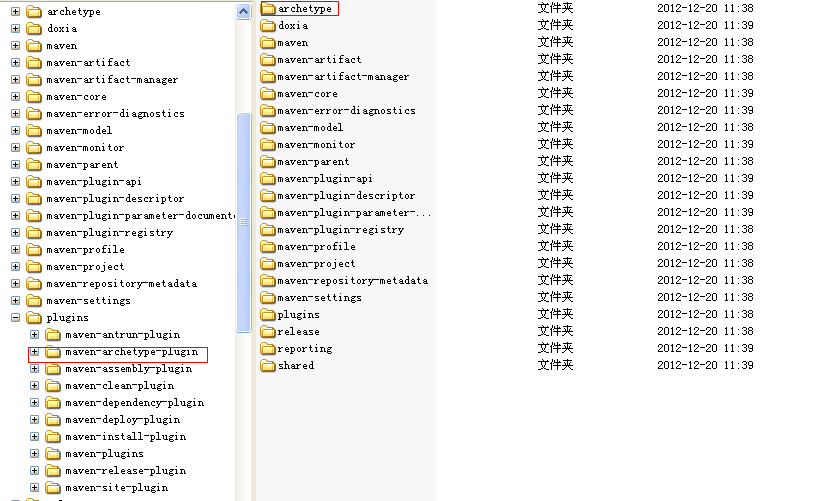
我们可以看到plugins字样，也可以看到相关的一些xml，还有其它的等等，猜一下吧，plugins应该是在下载Maven的相关插件，其它的是什么，一会我们看一下。

我们之前说过，当我们配置了mirror，mirror中可以配置私服地址，也可以配置远程地址，如Maven的中央仓库，好了，当配置了私服的时候，Maven会从私服上下载东西，第一次下载时，私服上也没有怎么办？当然会从远程仓库下载。

我们在上节确认了一下，本地仓库的文件夹里没有东西，现在再看一下：



果真，下载到了本地仓库文件夹中。我们看一下org\maven\plugins这个文件夹：



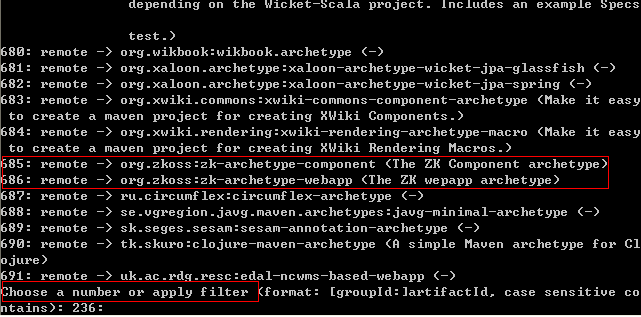
果真有刚刚我们使用的archetype插件。有兴趣我们再浏览一下其它文件夹，看看都有什么。另外，告诉大家我们项目里使用到的jar包也会下载到本地仓库中。

那么，是不是每次都要从私服上或者远程仓库上去下载呢？不会，当本地仓库中有的时候，就会使用本地仓库里的东西，当本地仓库里没有的时候，才会从私服或远程仓库下载。

#### 有关archetype

mvn archetype:generate命令会给我们生成什么样的代码结构？为什么会生成这样的代码结构？

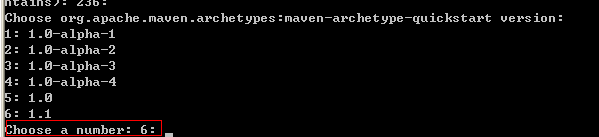
看下图：



我们会看到一有很多的archetype可以选择，这里我标出了一个zk项目的archetype，每个archetype都有一个编号，我们可以键入前面的这个编号以生成这个archetype的代码结构，如果我们不填写，就会提示一个默认的编号，可以看到这个编号为236。这个我们在Eclipse插件生成项目时会比较方便的看到，我们默认回车，继续。

#### 版本

选择archetype之后就会让我们选择一个版本号，如下图：



Maven的版本号定义是这样的：

<主版本>.<次版本>.<增量版本>.-<里程碑版本>

例如：1.2.3-bata-4表示第一重大版本的第二个次要版本的第三增量版本的bata-4里程碑。

* 主版本

表示项目的重大架构变动。如Struts1和Struts2，采用的不同的架构。

* 次版本

表示较大范围的功能增加和变化及Bug修复。如Nexus1.5比1.4版本添加了LDAP支持，并修复了很多的Bug，但总体架构并没什么变化。

* 增量版本

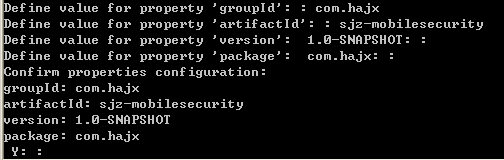
一般表示重大Bug的修复，如我们一个项目1.0.0之后，发现了重大的漏洞Bug，快速修复之后就是1.0.1版本。

* 里程碑版本

指某一个版本的里程碑，这样的版本相对正式的版本，并不稳定，而且需要大量的测试。

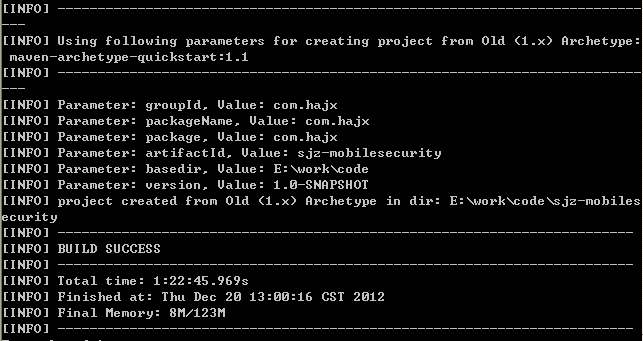
OK，有关Maven版本的其它信息可以参考其它资料，也希望有人能够补充。

#### 生成代码

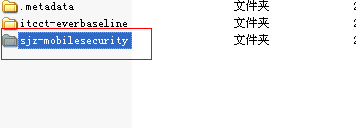


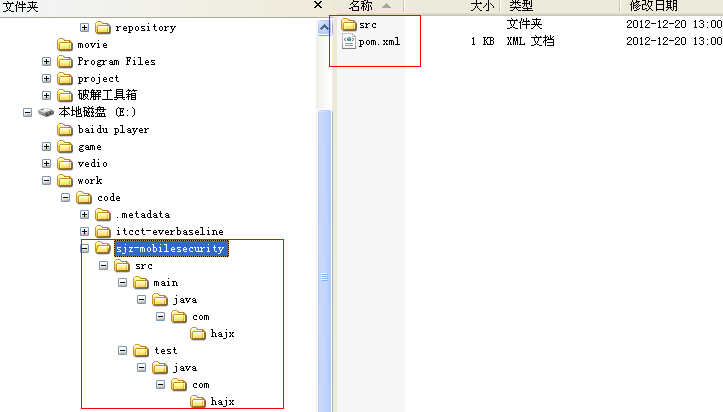
在这个页面，我们填写groupId、artifactId、version、package四项，version我们已经讲过，package是为我们生成的包结构，至于groupId和artifactId，下节会讲到。

键入Y继续。



OK，我们看到成功了，看一下Maven为我们生成了什么：



****

##### 三个文件

我们看一下根目录下的pom.xml文件及Maven为我们生成的两个java文件：

* pom.xml文件：



很简单是不是？groupId、artifactId、version三个构成了这个项目的坐标，坐标是我们下一节讲的概念。

packaging是项目输出的结果，这里默认是jar包。

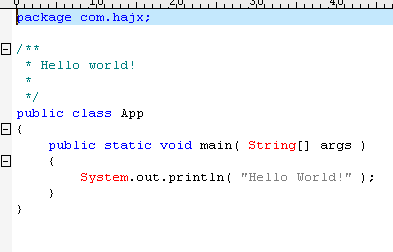
name声明了一个项目名称，这个不是必须的。

properties，在这里我们可以声明一些变量，这些变量可以在下面的引用，比如我们声明spring框架的版本号，以便统一spring相关包的版本。

dependencies，这里就是我们的依赖包声明的地方了，在这里我们可以看到Maven默认的archetype为我们添加了一个junit的依赖，一会我们向这里添加一个其它的依赖包试一下看看有什么变化。

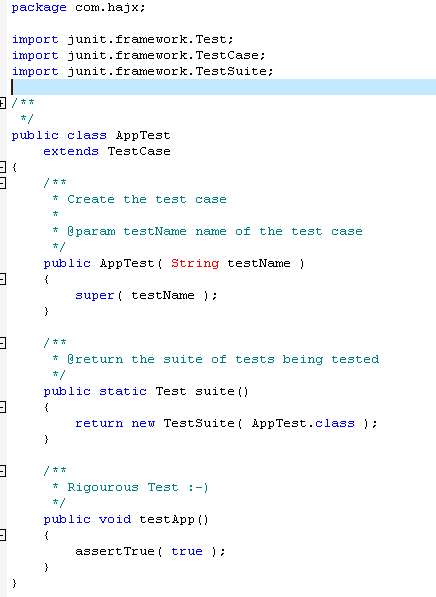
* App.java

在src\main\java\com\hajx文件夹下有一个App.java文件，文件本身代码意义并不大，但这个文件夹请大家注意，默认情况下，Maven假设项目的主代码位于src\main\java目录，这也是我们com.hajx.App.java文件在这个位置的原因，建议大家不要修改这个配置。看看这个java文件里写了什么：



* AppTest.java

同样，请大家注意这个文件的路径，默认情况下，Maven假设项目的测试代码位于src\test\java目录，这也是AppTest.java文件在这个位置的原因，强烈建议大家不要修改这个配置。看看这个文件写了些什么：

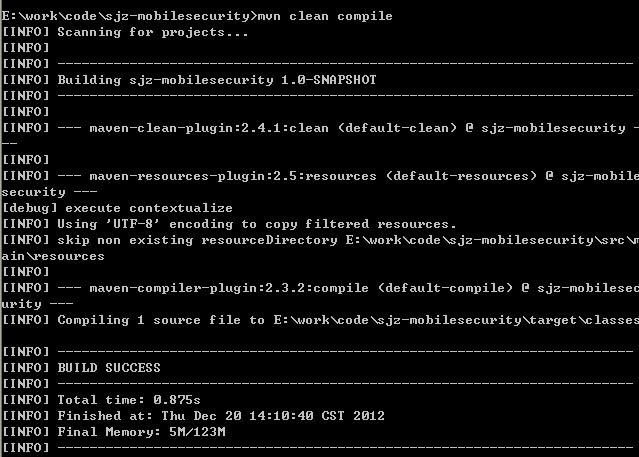


##### 编译、测试、打包、安装

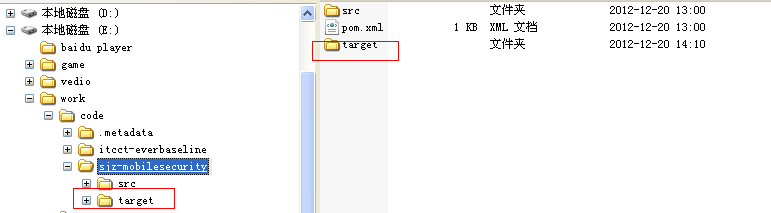
在命令行下，进行这个项目目录，我们看一下如何编译、运行、打包、安装。

* 编译

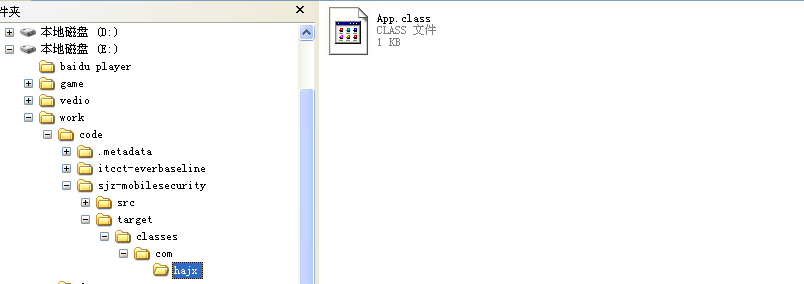
我们键入mvn clean compile命令：



编译成功了，我们看一下文件夹下的变化：



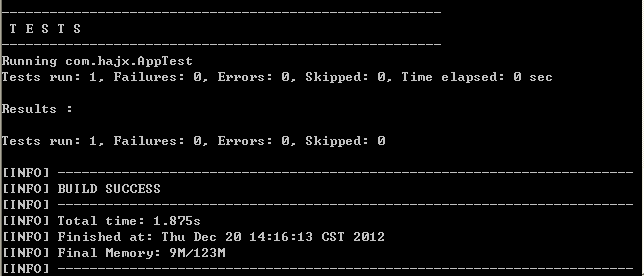
可以发现多出了target文件夹，展开来看一下：

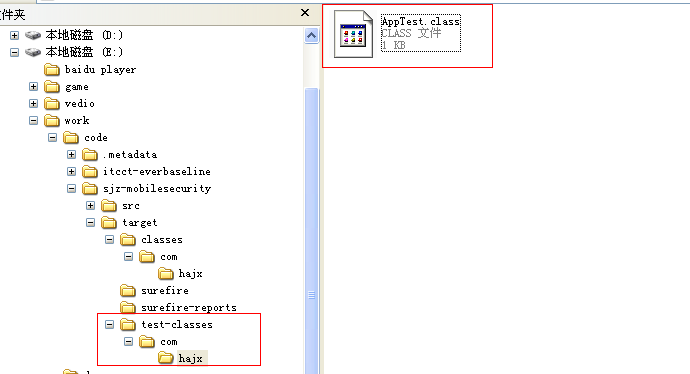


可以看到，主代码已经被编译成.class文件，而测试代码则没有。

* 测试

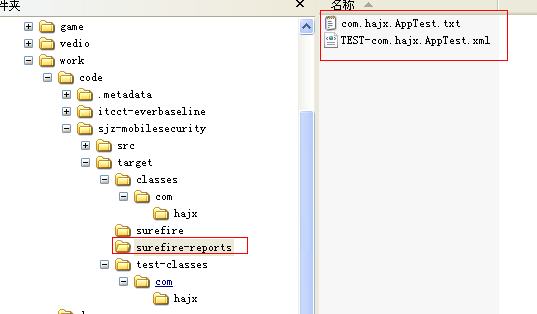
使用mvn clean test看一下：



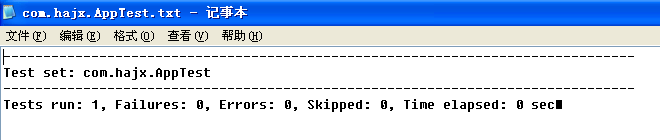


同样，也生成了测试代码的.class文件。不同的是，我们发现在执行测试之前，需要先编译，也就是说我们的测试是依赖于mvn compile的。这跟Maven的生命周期有关，后面会详细描述。

在测试的过程中，我们发现Maven使用了插件surefire，所以在第一次执行测试时会下载此插件及其相关jar包，测试执行后会在target目录下生成surefire及surefire-report文件夹，在surefire-report中是测试报告：



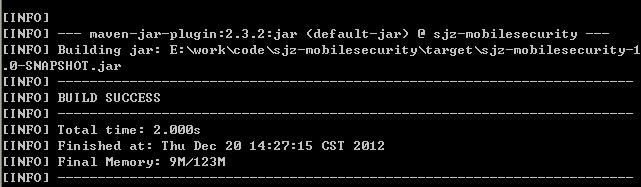
我们打开看一下：



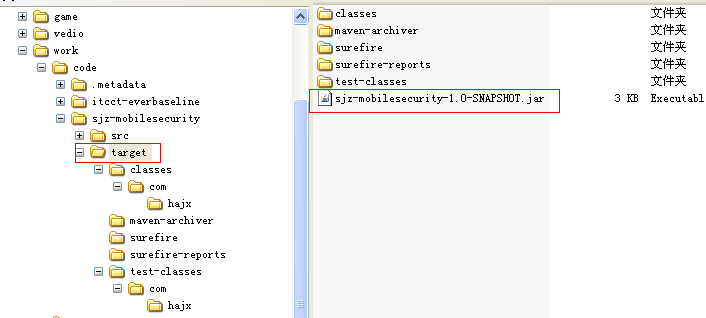
从中可以看出执行了几个测试，失败几个，错误几个，跳过几个，消耗时间。

* 打包

使用mvn clean package看一下：

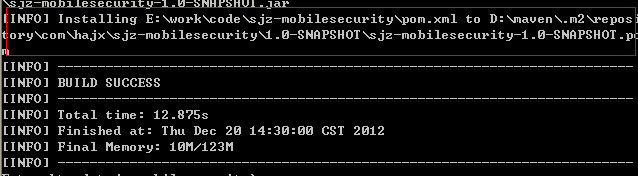


从输出中可以看出，打包依赖于编译、测试过程，另外在target文件夹下我们可以看到生成了jar包：

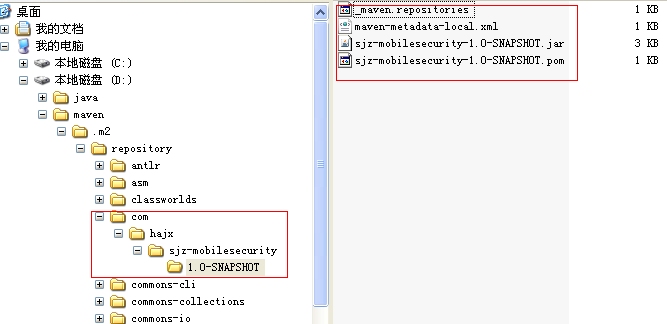


* 安装

使用mvn clean install看一下：



从上面可以看到，Maven把打成的jar包安装到了Maven本地仓库中，我们去看一下：

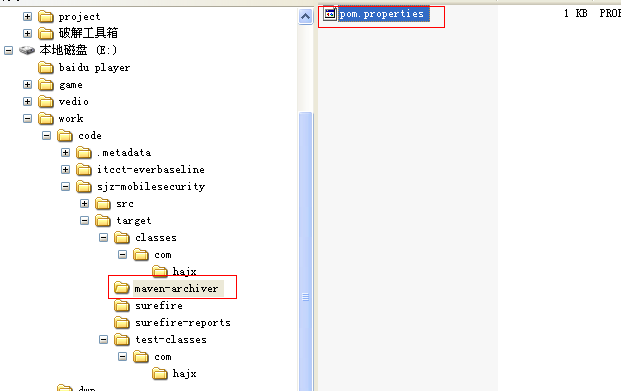


果然有，那么，这个jar包就可以被其它的Maven项目引用了，如何引用呢？看到那个pom文件了吗？打开看一下：

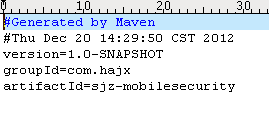


把红色部分拷贝到其它Maven项目中的pom.xml文件里就可以引用本地仓库中是这个jar包了。

同进在项目路径下多出一个文件夹，同时这个文件中也多出了一个文件，我们也看一下：



不知道是什么，打开看一下：

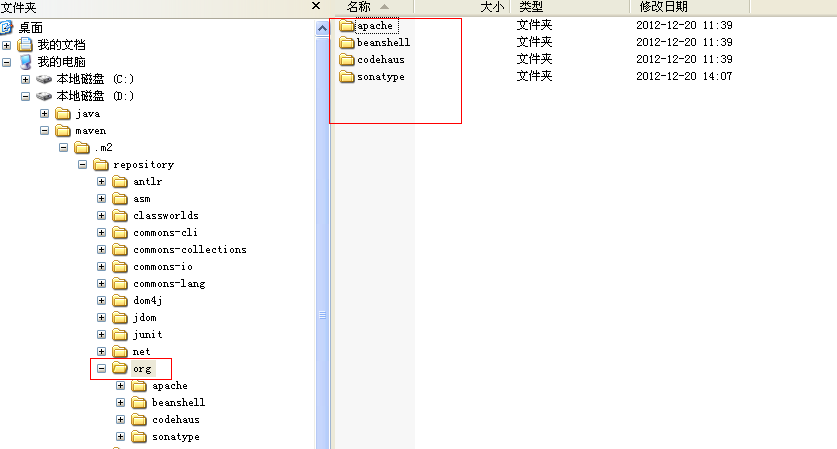


这显然安装的配置。

##### 修改pom.xml

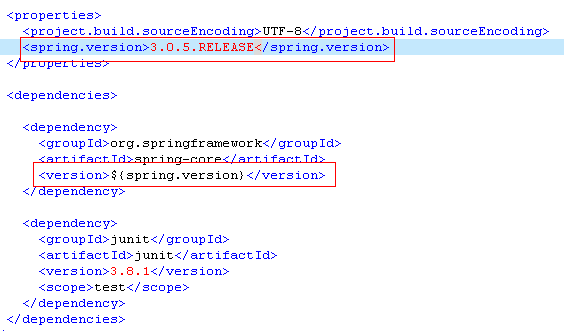
本节我们修改一个pom.xml，在里面添加一个spring核心包的依赖，看看会发生什么。

我们先看一下本地仓库：

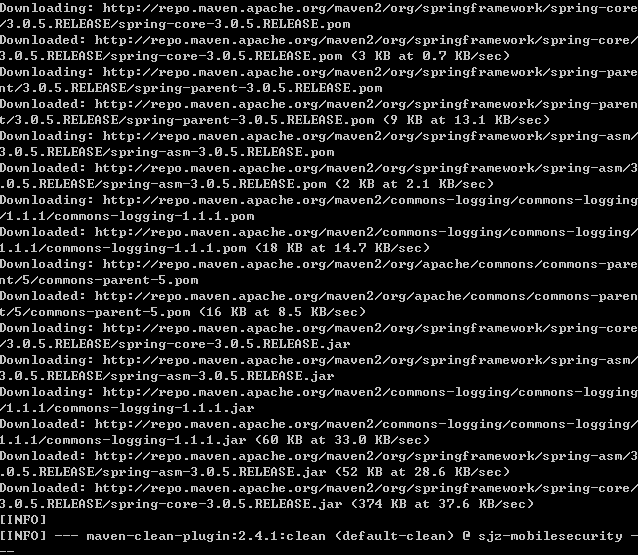


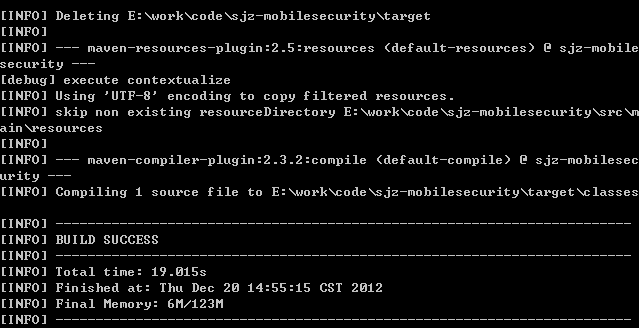
可以看到在本地仓库里并没有spring的相关信息。

我们把pom.xml修改成下面这样：

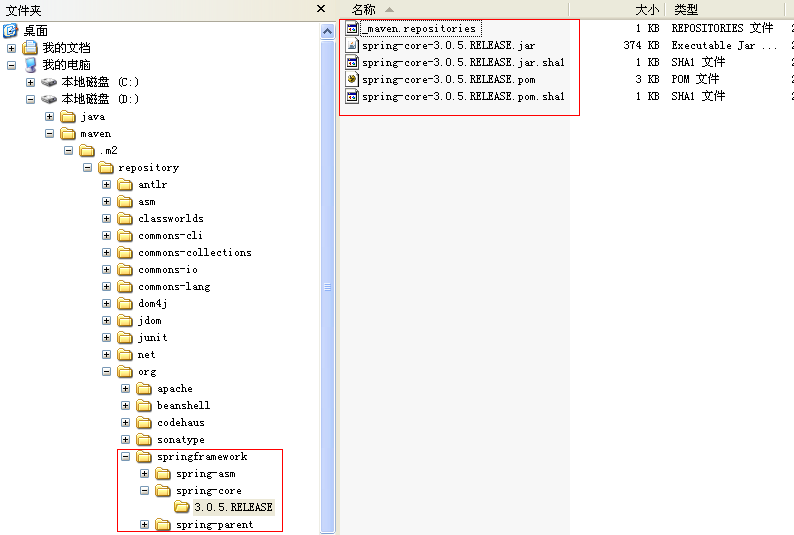


先在properties中定义了一个spring的版本，然后在下面依赖中添加了对spring核心包的依赖，OK，执行一下mvn clean compile：





我们再看一下本地仓库：



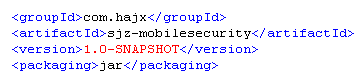
OK，Maven下载了spring的核心包，同进也将它的依赖包也下载下来，这也是我们在控制台看到那么多下载信息的原因。

## 坐标

之前我们用到spring的依赖，会去spring的官方网站上下载，用到log4j依赖的时候，会去Apache官方网站上去找，于是我们就想，如果有人弄一个规范、一个规则，自动化处理jar的依赖多好，不错，Maven就做了这件事，Maven定义了这样的一组规则：世界上任何一个构件都可以使用Maven坐标唯一标识，只要提供正确的坐标，Maven就能找到对应的构件。Maven的中央仓库<http://repo1.maven.org/maven2>就提供了我们所需要的绝大部分包，以后我们不再需要到各个网站上去下载各种依赖包，是不是很棒？

另外，Maven强制要求我们在开发自己项目的时候也定义适当的坐标，这样对团队的合作有相当的好处，没准哪天我们自己写的jar包也出现在Maven的中央仓库上让其他人使用呢。

Maven的坐标是通过一些元素定义的，它们就是我们前面所看到的groupId、artifactId、version、packaging、classifier。如我们上面项目的坐标：



其中没有classifier。下面我们详细解释每个元素：

* groupId

定义当前的Maven项目隶属的实际项目，Maven项目和实际项目不一定是一对一的关系，比如我们这边的EverExploit星空自动化安全评估系统这一实际项目，就包含基线和漏洞扫描两个Maven项目。groupId不应该对应某个组织或公司，上面我使用com.hajx就不合适，因为一个组织或公司下可能有很多实际项目，通常我们使用一个组织或公司的域名加实际项目的名称来命名，如上面的com.hajx应改为com.hajx.security。

* artifactId

artifactId定义的是实际项目中的一个Maven项目模块，推荐的做法是使用实际项目的前缀加模块名称，如上面的artifactId可以为mobilesecurity-msg，表示手机安防项目的短信模块。这样的好处在于Maven检索构件。

* version

定义Maven项目当前所处的版本。

* packaging

定义Maven项目的打包方式。

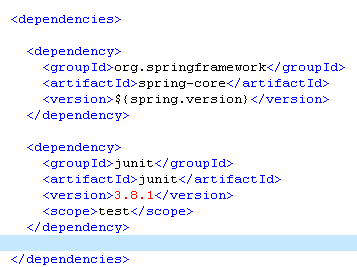
* classifier

定义构件输出的附属构件，我们可以在打包的时候提供javadoc和源文件，这些就是附属构件。

## 依赖

### 依赖配置

我们看pom.xml一个例子：



每个依赖包含的元素有：

* groupId、artifactId、version

依赖坐标

* type

依赖的类型，一般不需要声明，默认为jar。

* scope

依赖范围，稍后详细描述。

* optional

依赖是否可选。

* exclusions

排除传递性依赖，稍后详细描述。

### 依赖范围

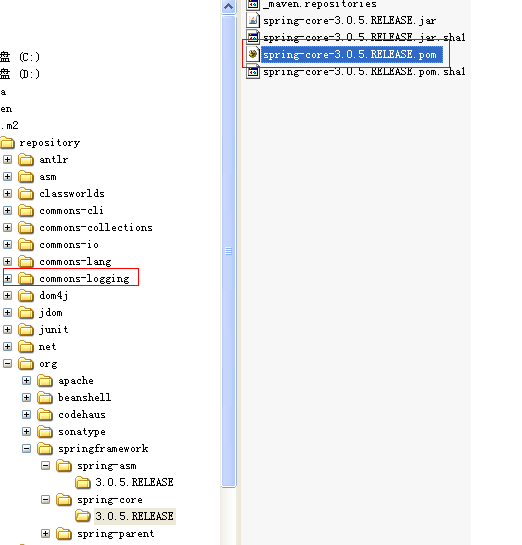
依赖范围用来控制依赖与编译时的classpath、测试时的classpath、运行的classpath的关系。我们有如下的表格表示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **依赖范围** | **对编译CP有效** | **对测试CP有效** | **对运行时CP有效** | **示例** |
| compile | Y | Y | Y | Spring-coue |
| test |  | Y |  | JUnit |
| provided | Y | Y |  | Servlet-api |
| runtime |  | Y | Y | Jdbc驱动 |
| system | Y | Y |  | 本地jar包 |

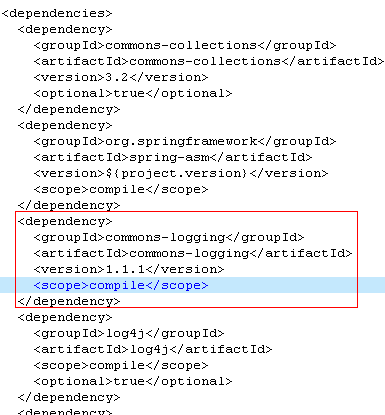
### 传递依赖

目前我们开发一个SpringMVC项目的时候有两个情况发生，一种是我们从spring官方文档上下载全部的依赖包，不管怎么样，全加入项目里；另外一种就是只下载简单的几外，然后项目报错时再一个一个加进这些包依赖的其它包，很麻烦。

有了传递依赖就好办多了，我们看之前我们的项目，我们只加了spring核心包的依赖，却下载了spring核心包所需的其它包，我们看：



我们打开spring-core-3.0.5.RELEASE.pom看一下：



我们的项目依赖于spring核心包，而spring核心包中又有对commons-logging的依赖，依赖范围为compile，则commons-loggin包可称为这个项目的传递依赖。可见，有了传递依赖，我们不再一个一个找依赖包，也不会在项目中出现多余的依赖包，以致于混乱，当然，如果pom.xml比较乱就没办法了。

#### 传递性依赖和依赖范围

假设A依赖于B，B依赖于C，A对B是第一直接依赖，B对C是第二直接依赖，A对C是传递性依赖。下面有如下的表格表示依赖范围对传递性依赖的影响，最左边是第一直接依赖，最上面一行是第二直接依赖，这个表纵向来看比较好看：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **compile** | **test** | **privided** | **runtime** |
| **compile** | compile |  |  | runtime |
| **test** | test |  |  | test |
| **privided** | privided |  | provided | privided |
| **runtime** | runtime |  |  | runtime |

### 依赖调解

* 路径优先原则

A->B->C->D(1.0)和A->X->D(2.0)，此时会解析哪个D呢？显然，根据路径优先原则，使用D(2.0)

* 第一声明优先

A->B->C(1.0)和A->X->C(2.0)，同样，根据声明顺序，在pom.xml文件里，哪个先声明，先使用哪个。

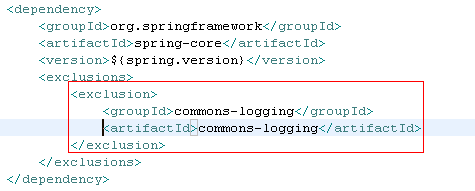
### 可选依赖

假设在我们的项目中存在一个项目B，B项目依赖于mysql-connector-java，这个依赖为可选依赖，当A项目依赖于B时，必须明确声明对mysql-connector-java的依赖，即使A对B、B对mysql-connector-java之间的依赖范围都是compile。

可选依赖肯定是被依赖者存在多样性，所以这样的依赖违反了单一出现原则，应该尽量避免。

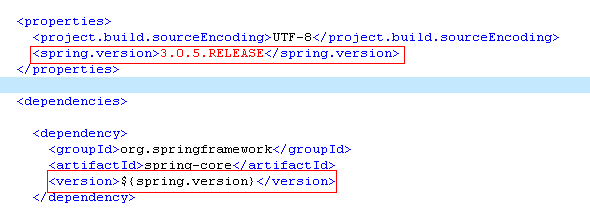
### 排除依赖

当出现A->B->C(1.0)和A->X->C(2.0)情况或类似A->B->C->D(1.0)和A->X->D(2.0)时，我们要注意判断，可以使用exclusion来将某个依赖项排除。如下面：



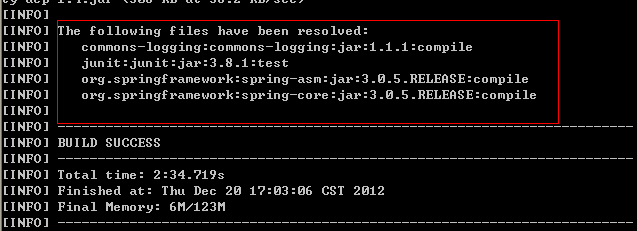
### 归类依赖

当我们使用SpringMVC的一些包时，有可能还会用到SpringSecurity的一些包，那么这些包的版本应该一致，上面我们已经使用到变量了：

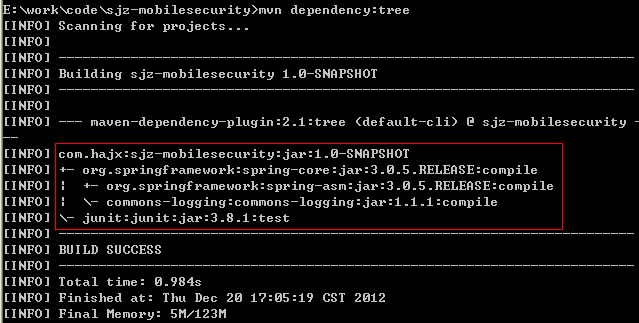


### 优化依赖

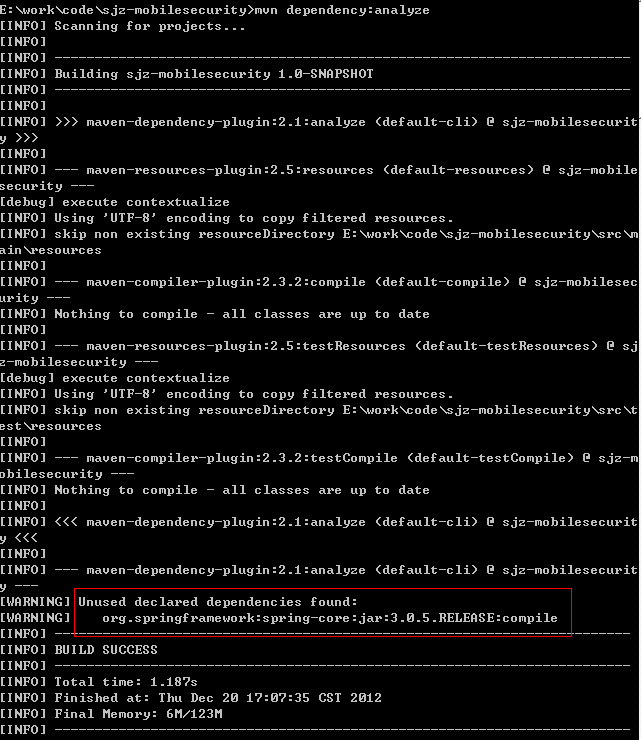
使用mvn dependency:list可以看到当前项目的已解析项，并且已经标出依赖范围：



在此基础上使用mvn dependency:tree可以看到传递依赖：



如果再运行mvn dependency:analyze，再看一下：



对于这个例子，Unused declared dependencies，说明项目中未使用，在这里可以删除，但在实际的项目中还需要具体的分析，因为dependency:analyze只会分析编译主代码和测试代码需要用到的依赖，一些执行测试和运行时的依赖它发现不了。

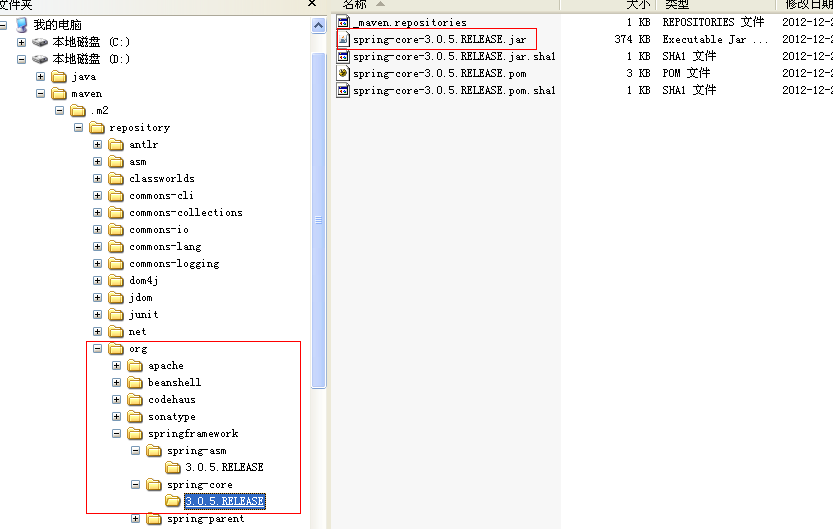
如果发现Used undeclared dependencies时，表示项目中使用到，但没有显式声明的依赖。这种依赖一般是传递依赖造成的，当直接依赖升级的时候，传递依赖的版本也可能会发生变化，这种变化不易发现。

## 仓库

任何一个依赖、一个插件或一个项目的输出（如jar包），都可以叫做一个构件，即一个artifact。

如果把这些构件放在一下位置，以供一个公司或者所有人下载使用，这个位置就叫做仓库。

构件在仓库里保存的路径为：groupID/artifactId/version/artifact-version.packaging，如下图中的spring核心包在本地仓库中的路径：



### 本地仓库

Maven本地仓库就是在localRepository中配置的本地路径。

原来我们每开发一个项目，都要把一些jar包拷贝到项目的目录下，现在只要放在本地仓库中就可以，而且坐标明确，版本清晰，还不需要拷来拷去。

### 远程仓库

如果项目的依赖本地仓库中没有怎么办？当然会到远程仓库里下载到本地仓库中，只要本地仓库中存在，就无需再去远程仓库中去下载。

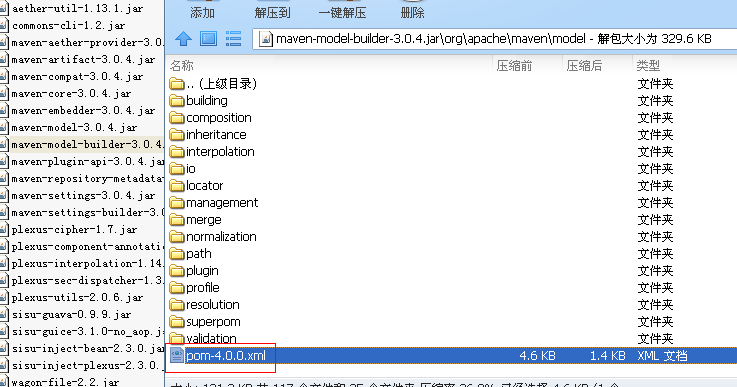
Maven本地仓库只有一个，而远程仓库可以有多个。

远程仓库可以配置在settings.xml里，也可配置在pom.xml里，看大家的使用习惯，当然，如果一个人参与了多个项目，写到pom.xml里是比较合适的。

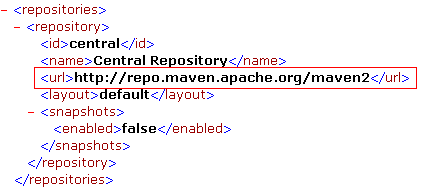
### Maven中央仓库

Maven的中央仓库是一个特殊的远程仓库，在最初的时候Maven的本地仓库是空的，Maven必须知道一个可用的远程仓库，才能下载到需要的构件。

在$M2\_HOME/lib/ maven-model-builder-3.0.4.jar中/org/apache/maven/model/pom.4.0.0.xml：



打开看一下：



### 私服

私服，是架设在局域网中的Maven仓库。当本仓库中没有需要的构件时，可以从私服上去下载，如果私服没有再去远程仓库上去下载，然后缓存到私服上。

我们来看一下私服的好处：

* 节省带宽

在网络情况较差的情况下，如果大家都从中央仓库或远程仓库上下载东西，杨跃是不是要骂人？

* 加速Maven构件

Maven一些内部机制要求Maven在执行构建的时候不断检查远程仓库数据，如果项目配置了多个远程仓库，构建的速度显然会变慢。

* 部署第三方构件

当无法从远程仓库上获得某个构件时怎么办？比如我们使用的加密狗的dll或so文件，当然可以上传到私服上去；当然，如果某个人写了一个比较不错的公用的构件也可以上传上去，让全公司共享。

* 提高稳定性、增强控制

如果Maven依赖于远程仓库，当网络情况极差时，很可能导致项目无法构建，而私服上缓存了大量的构件，使用私服这个问题得到解决。

* 降低中央仓库的负荷

维护Maven中央仓库这样的东西肯定需要大量的财力和人力，使用私服肯定会给apache省钱。

我们会在第4章详细描述Nexus的使用。

## 生命周期与插件

除了坐标、依赖、仓库之外，Maven另外两个核心的概念就是生命周期和插件。

Maven的生命周期是抽象的，这表示生命周期本身不做任何事，实际的任务，如编译、运行、打包、安装，都是由插件来完成的，Maven只是定义了一组统一的接口而已，可见生命周期和插件两者密不可分。

对于生命周期希望其他同事补充。

### 三套生命周期

Maven有三套相互独立的生命周期，即clean、default、site，每个生命周期又包含一些阶段，即phase，并且这些phase是有顺序的，后面的phase依赖于前面的phase。

* clean

clean目的是清理项目，包含三个阶段：

pre-clean:执行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：执行清理后需要完成的工作

* default

default定义了真正构建时需要执行的所有步骤，是所有生命周期中最核心的部分，它的phase较多，我找出几个常用的：

compile：编译项目主代码，即/src/main/java下的源文件

package：打包项目成可发布的形式，如war包

install：将构件安装到Maven本地仓库

deploy：将最终的包复制到远程仓库，供其他人使用

* site

site的目的是建立和发布项目站点，Maven能基于pom.xml中的信息自动生成一个站点，以便于用户反馈及团队交流。

pre-site：执行生成站点之前需要完成的工作

site：生成站点文档

post-site：执行生成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。

### 命令行与生命周期

Maven命令即是调用Maven项目不同的生命周期阶段，下面是常见的Maven命令，解释一下其执行的生命周期阶段：

* mvn clean

pre-clean；clean

* mvn test

default生命周期的validate、initialize等，直到test的所有阶段

* mvn clean install

clean生命周期的clean和default的install阶段

* mvn clean deploy site-deploy

clean生命周期的clean、default生命周期的deploy阶段以及site生命周期的site-deploy阶段。

### 插件目标

对于mvn dependency:list，list就是dependency插件的目标。Maven核心包不到3M左右，所以Maven在需要的时候会先下载插件。而一个插件可以完成多个任务，如mvn clean compile。

# 聚合与继承

## 聚合

## 继承

## 聚合与继承

## 约定优于配置

# Nexus私服

私服不是Maven的核心概念，但私服能降低中央仓库的负载、节省外网带宽、加速Maven的构建，并且可以部署自己的artifact，这里我们关心节省外网带宽和部署自己的artifact。

本章我们看一下Nexus的安装与使用。

## Nexus安装

从http://www.sonatype.org/nexus/go下载最新版本的Nexus：

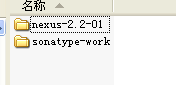


我们都下载下来，因为我们要在Windows和Linux下都测试安装一下。

### Windows安装

#### 解压

将zip包解压后得到两个文件目录：

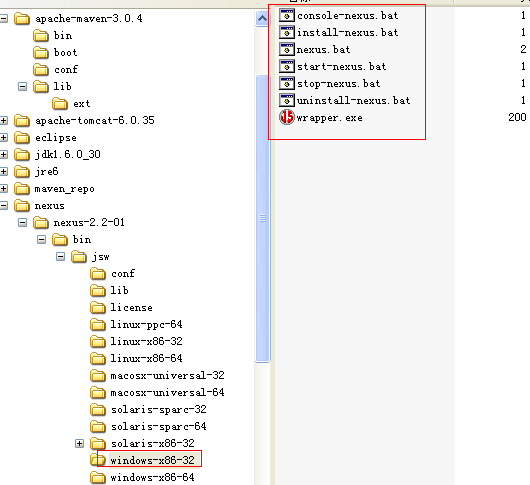


nexus-2.2.-01:这个目录包含了Nexus运行所需要的文件，如启动脚本、依赖jar包等。这个目录是运行Nexus必需的。

sonatype-work：这个目录包含了Nexus生成的配置文件、日志文件、仓库文件等。这个目录不是必需的，Nexus在运行的时候自动创建该目录。

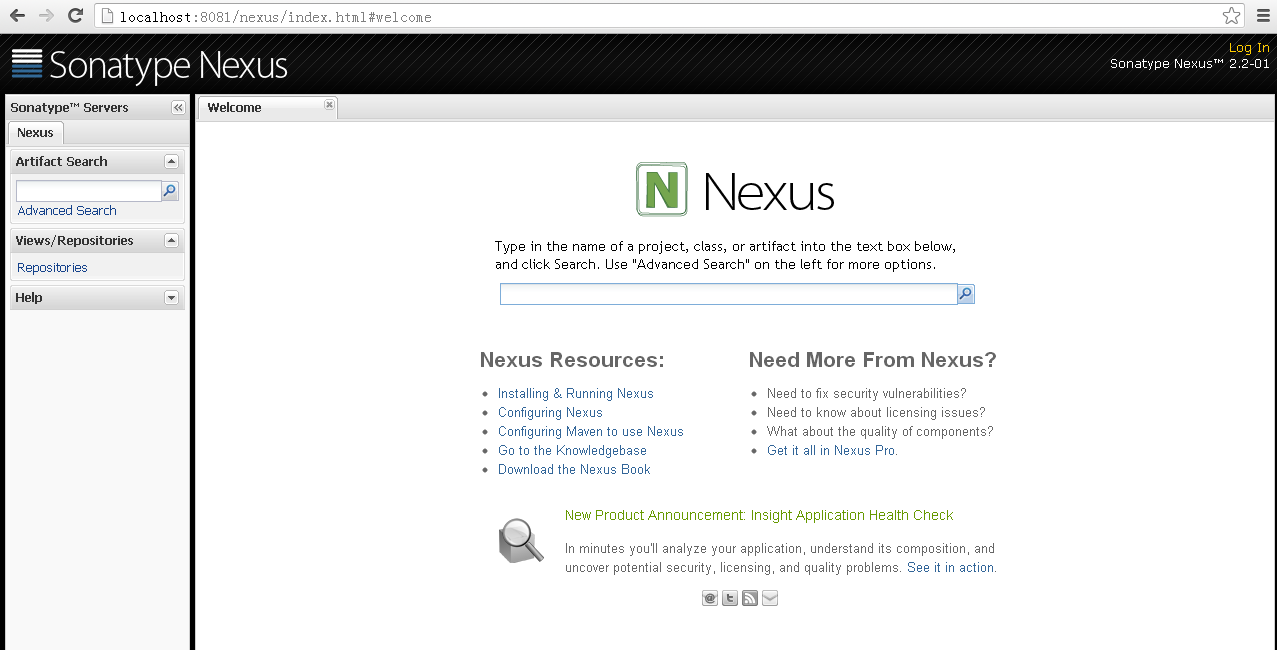
#### 启动

在nexus-2.2-01\bin\jsw目录下，有不同操作系统的文件夹，我的是32位XP，所以进入windows-x86-32文件夹，可以看到有如下的文件：



因为Nexus的Bundle自带了Jetty容器，所以不需要额外的Web容器就可以直接启动，这里我们运行install-nexus.bat，将Nexus安装成Windows服务，让Nexus在Windows启动时自动启动，很方便，然后运行start-nexus.bat，即可启动Nexus。

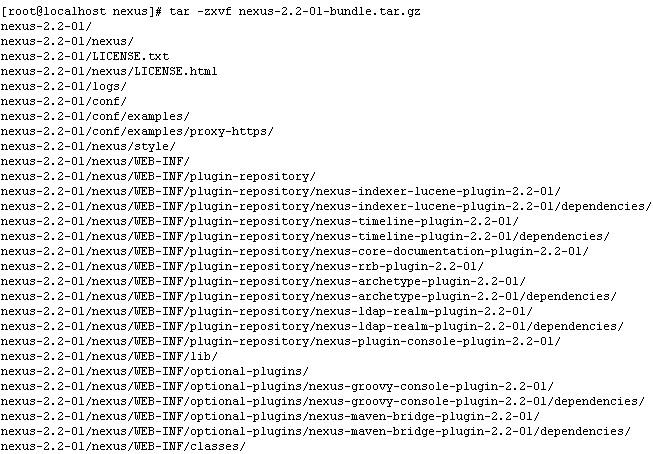
访问一下：<http://localhost:8081/nexus>试一下：



如果想要卸载则运行uninstall-nexus.bat即可完成Nexus的卸载；同样运行stop-nexus.bat即可停止Nexus的Windows服务。

### Linux安装

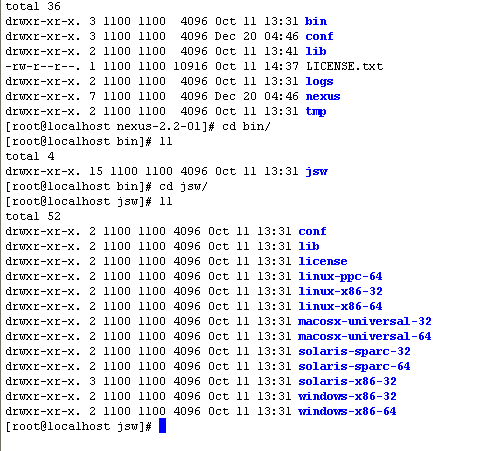
#### 解压



然后我们得到两个文件夹：



进去看一下：



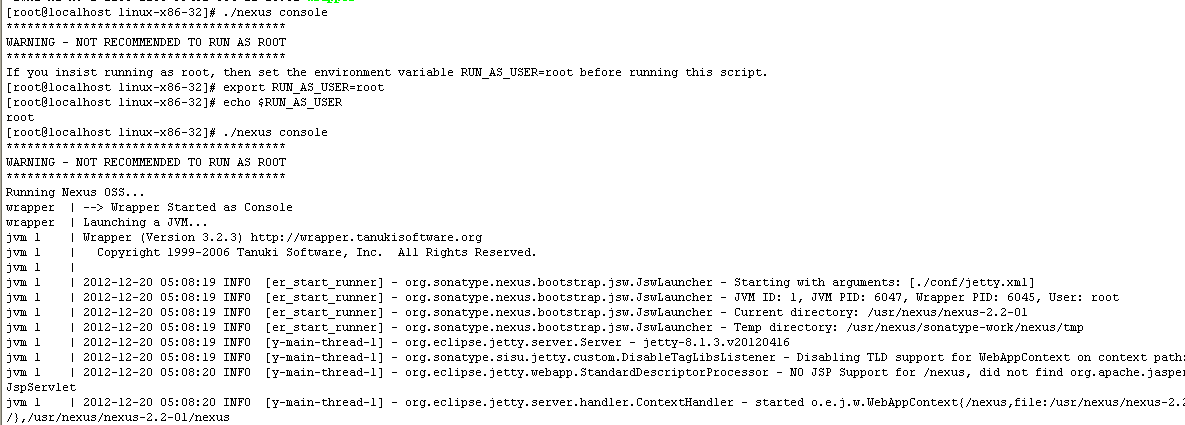
可以看到跟Windows环境一致。

#### 启动

我的Linux环境是CentOS6，所以进入linux-x86-32文件夹：



然后运行./nexus console：

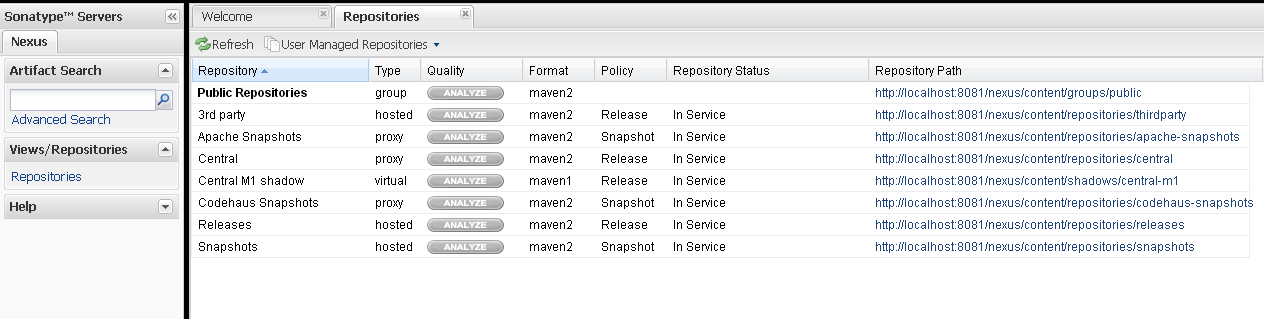


使用Ctrl+C停止Nexus。

不过我经常使用./nexus start和./nexus stop来启动和关闭nexus。

## Nexus的仓库

点击Nexus左边Repositories，可以看到右边的仓库：



从中可以看到仓库有四种：

Group(仓库组)、hosted(宿主仓库)、proxy(代理)、virtual(虚拟)，每个仓库的格式为Maven1或Maven2，这里不涉及Maven1。

Maven可以直接从宿主仓库中下载构件；也可以从代理仓库下载，而代理仓库也会间接地从远程仓库下载并缓存构件；为了方便，我们建立一个仓库组，包含宿主仓库和代理仓库，仓库组没有实际的内容，它会转向其包含的宿主仓库或代理仓库。

这些介绍一下几个主要的仓库：

* Central

代理Maven的中央仓库，其policy为release，只会发布和缓存中央仓库的发布版本构件。

* 3rd party

宿主仓库，用于缓存无法从公共仓库获得的第三方构件，如mysql的JDBC驱动和我们自己写的构件。

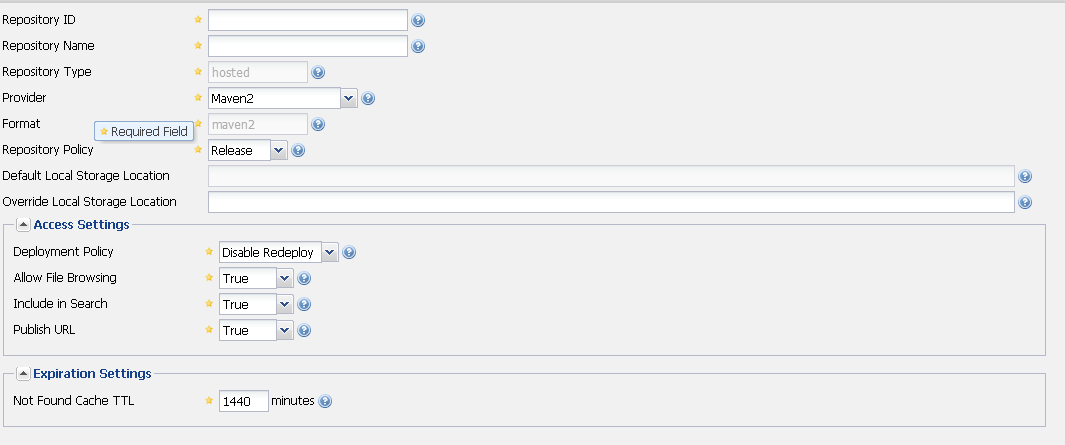
* Codehaus Snapshots

策略为snapshot的代理仓库，代理codehaus maven仓库的快照版本构件。

## 建立宿主仓库

首先登录Nexus，默认的用户名和密码是admin/admin123。

在左边点击Repositories，在右边点击Add，选择Hosted Repository，下面出现：



* Allow File Browsing

表示是否允许浏览仓库内容，一般选择True。

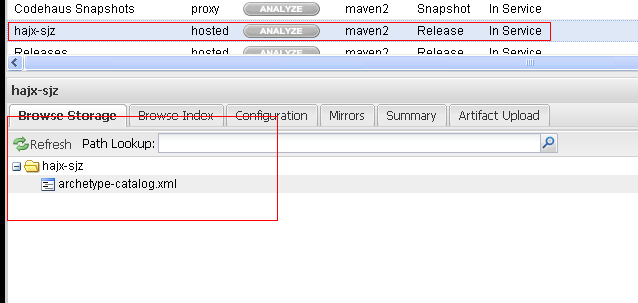
* Include In Search

是否对此仓库进行索引并提供搜索，一般选择True。

* Publish URL

表示是否通过URL提供服务，一般选择True。

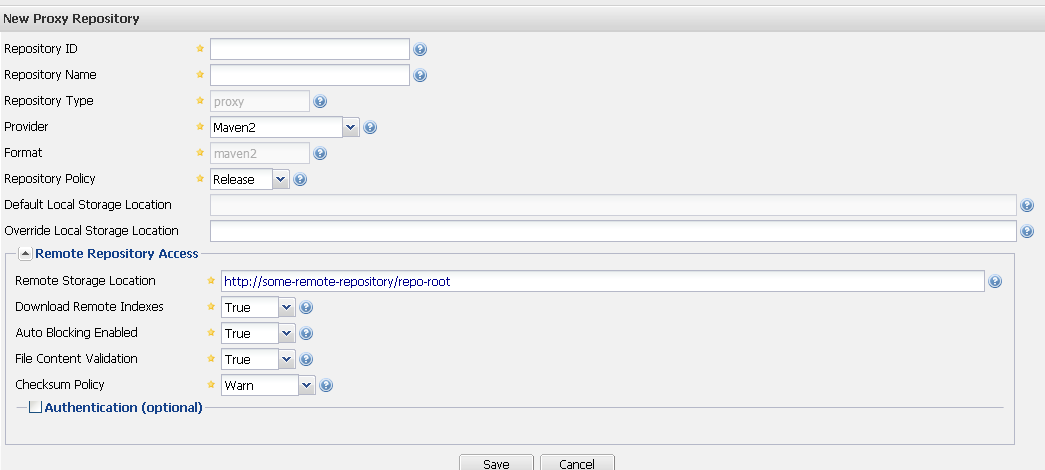
建好之后可以看到：



之所以能看到Browse Storage，是因为上面的Allow File Browsing为True。

## 建立代理仓库

在左边点击Repositories，在右边点击Add，选择Proxy Repository，下面出现：



* Repository Policy Location

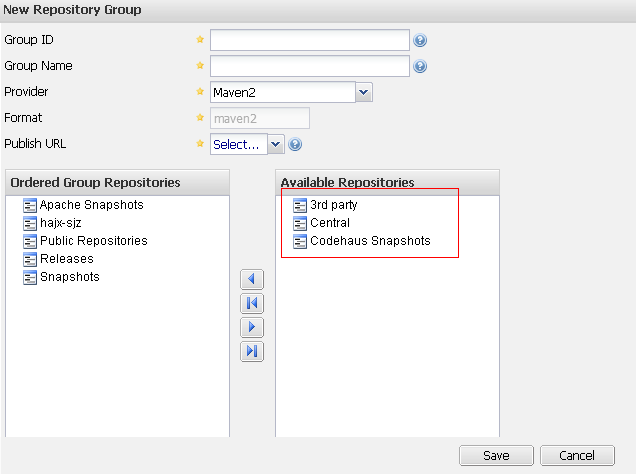
远程仓库的地址。

* Download Remote Indexes

是否下载远程仓库的索引，下载其索引后，即使没有缓存远程仓库的构件，我们还是可以在本地搜索和浏览到远程仓库构件的基本信息。

## 建立仓库组

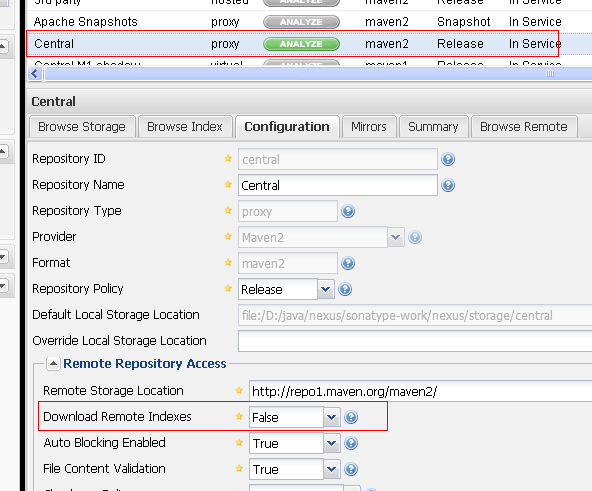
在左边点击Repositories，在右边点击Add，选择Repository Group，下面出现：



一般情况下，我们只需要上图所示的三个仓库即可，需要注意的是仓库组包含的仓库的顺序，这个顺序决定了仓库组遍历其包含的仓库的次序，所以最好将常用的、风速较快的仓库放在上面。

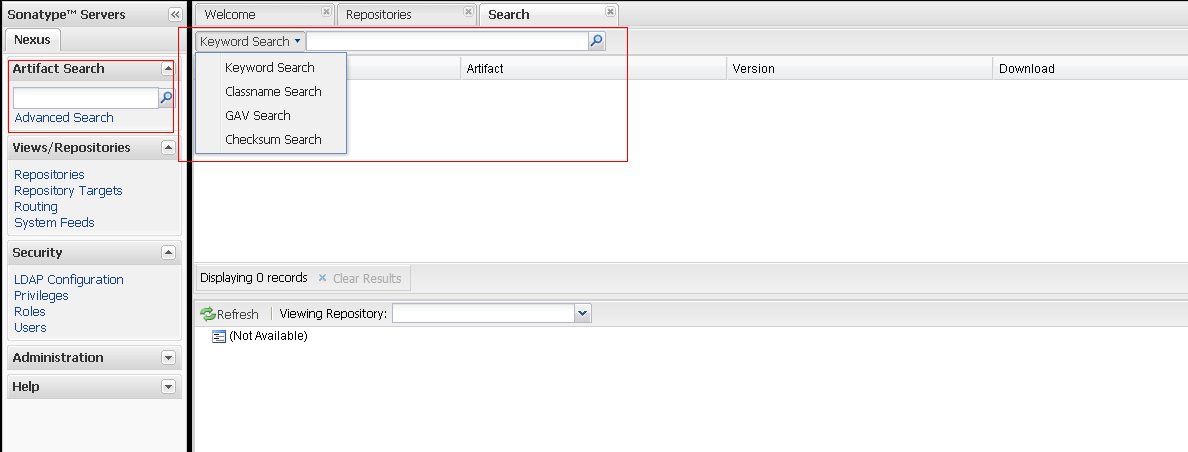
## Nexus的索引与搜索

如果让Nexus提供搜索功能，需要打开Central代理仓库下载远程索引：

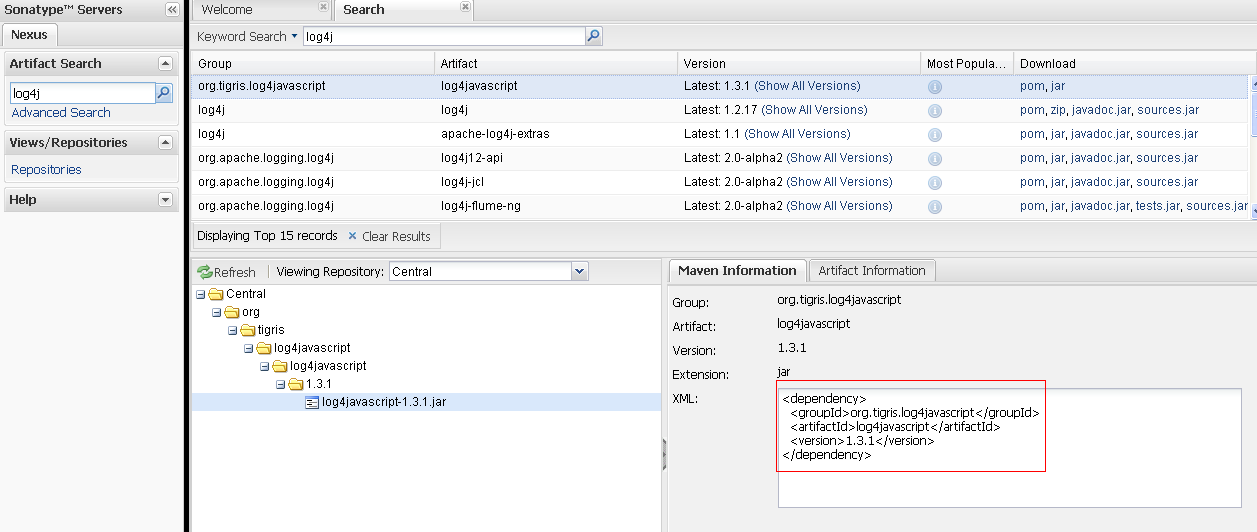


可惜默认这个选项是关闭的，点击保存时，会下载索引。

下载完索引后就可以搜索相关构件了，在左上角：

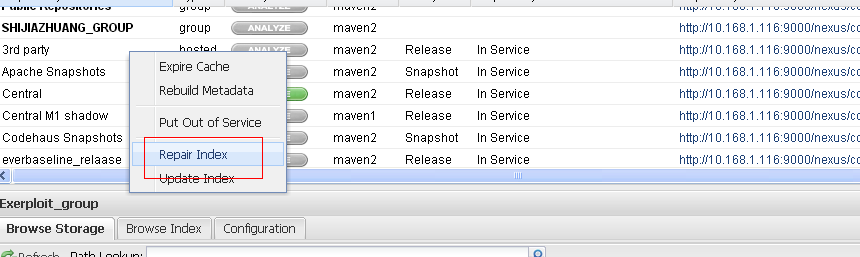


如下图，我们搜索log4j：



看到右边红色标出的东西了吗？我们直接将这段拷贝到我们的pom.xml里就可以了。

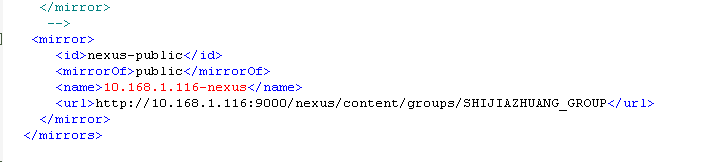
除了下载远程仓库的索引，我们也可以为宿主和代理仓库建立索引，只需要在仓库上右键：



对于宿主仓库来说，Repair Index会扫描其包含的构件并建立索引；对代理仓库来说，Repair Index会扫描所有缓存的构件并建立索引，如果远程仓库也有索引，则下载后与本地索引合并；对仓库组，Repair Index会合并其包含的所有仓库索引。

## 配置Maven从Nexus下载构件

一般我们习惯于修改Maven的settings.xml，不在pom.xml里来添加对仓库的配置。所以见我们的使用的Maven中的settings.xml：



## 部署构件到Nexus

## 手动部署构件到Nexus

## 为项目分配独立仓库

## 权限管理

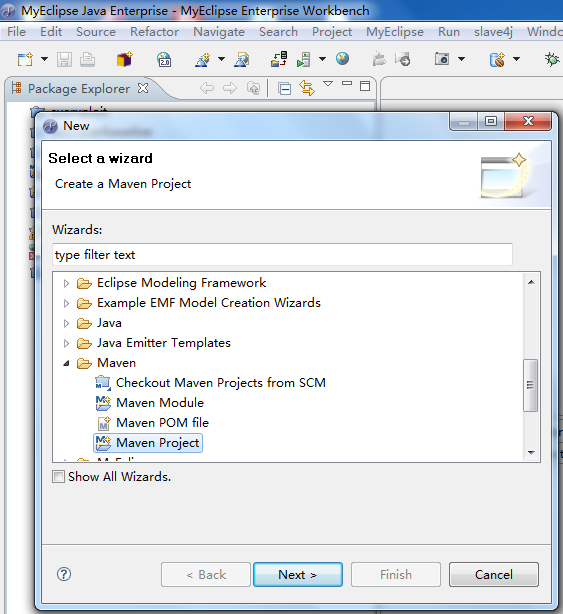
# Maven+SVN+Nexus的Web项目实例

将Maven、Nexus、SVN安装配置好，并在Eclipse里安装完插件之后，我们就可以完成此章的实例。

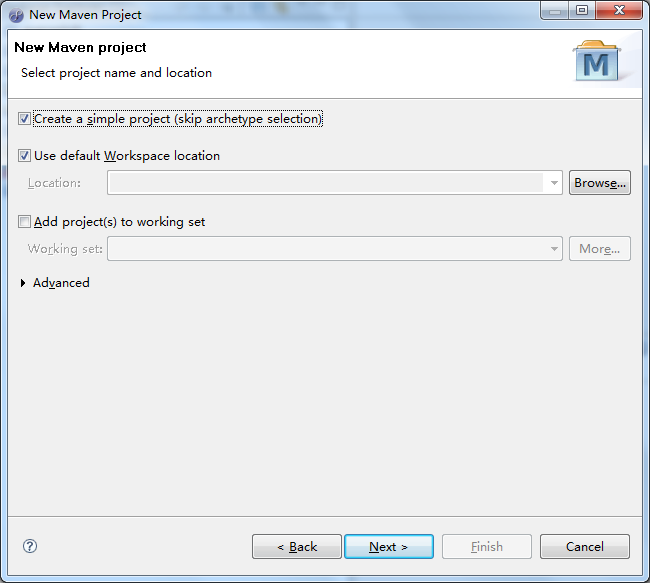
## Maven项目的创建与配置

### 创建Maven项目

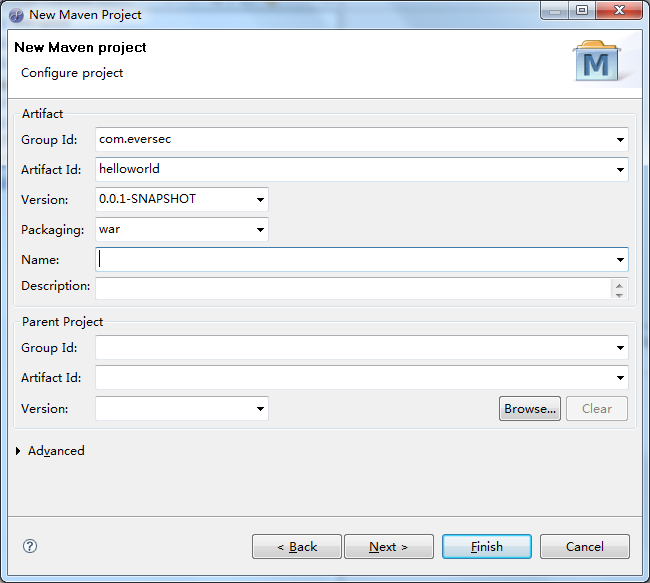
在左边项目区   点击右键  --> New  --> Other --> 找到maven(或者直接输入maven)  -->选择 Maven Project



点击Next，

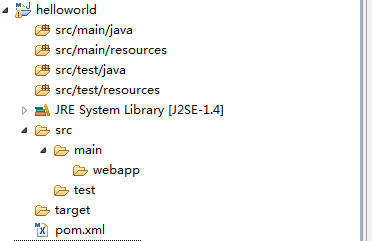


勾选第一个选择框，创建一个简单通用的Maven项目，勾选第二个选择框，选择存放到默认的工作空间，然后点击Next，



如图所示，依次填写Group Id（可随便填写，最好是填写公司或组织名称）、Artifact Id（项目名），选择Packaging（打包方式），然后点击Finish。

创建成功后，我们会看到如下图所示的项目结构：



源文件全部都放在src/main/java目录下

资源文件全部都放在src/main/resources/目录下

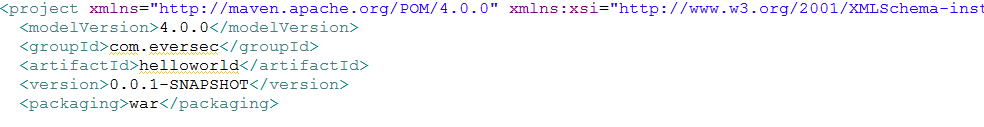
同理，测试的源文件跟资源文件分别放在

src/test/java和src/test/resources目录下

页面、样式文件以及图片等资源全都放在webapp目录下

### 配置pom.xml

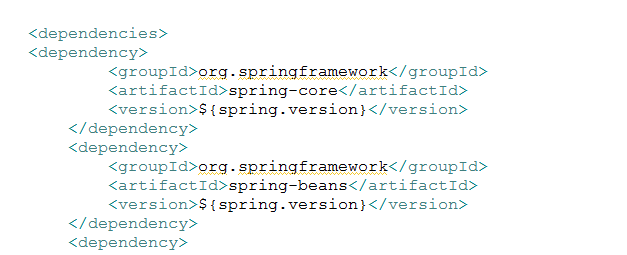
如图，是新建的Maven项目时生成的pom.xml自带的，关于项目本身的一些描述以及项目打包方式。



如图，一些常用的变量可以配置在properties属性中



如图，项目中依赖的jar包，配置在dependencies属性中，常用的jar包依赖可以从Nexus私服中，通过查询获取，一些不常用的，中央仓库都查询不到的，需要先上传，然后把生成jar包依赖拷贝粘贴到pom.xml文件中。

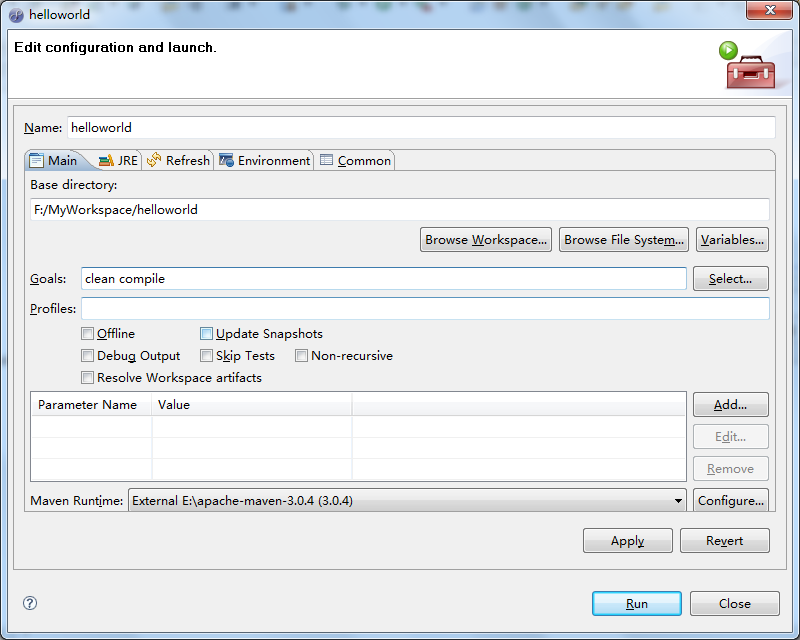


如图，plugins属性中配置了一些打包、发布需要用到的插件，如：tomcat服务器



### 运行Maven项目

在Maven项目中点右键🡪选择run as🡪选择Maven build



在Goals:后面填写运行目标，然后点击Run

Goails命令：

clean 清理项目

compile 编译项目

install打包项目

tomcat:run使用Maven集成的tomcat服务器运行项目

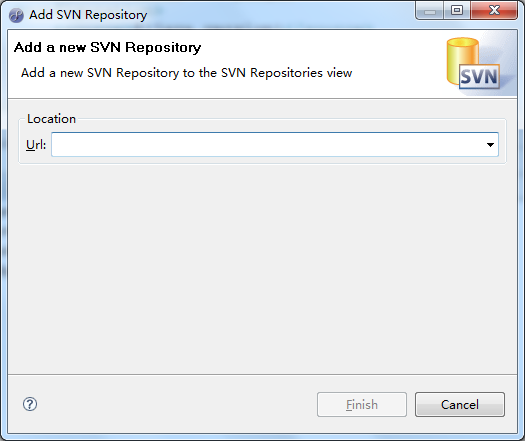
tomcat:redeploy使用pom.xml中配置的tomcat服务器发布项目

注意：在发布项目之前，要确保目标tomcat服务器已经正常启动。

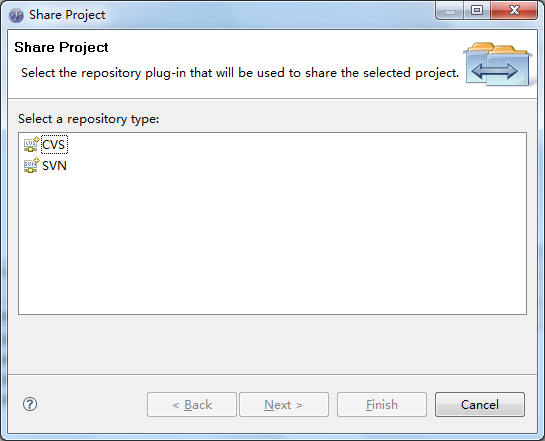
## SVN管理与维护

### SVN check in

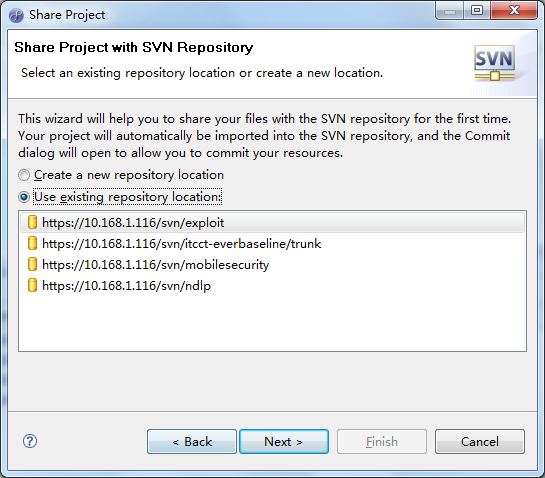
新建SVN资源库



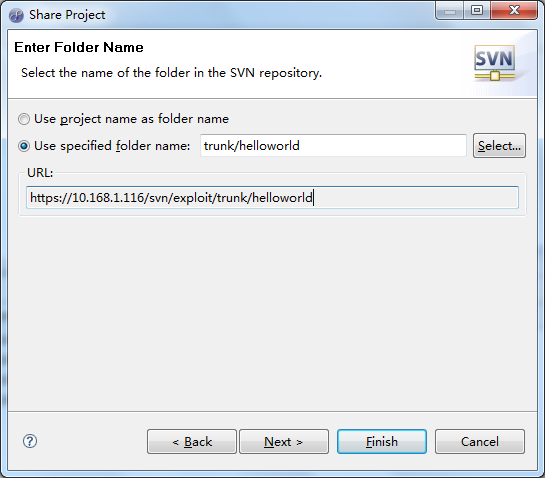
在项目上点右键🡪Team🡪Share Project🡪选择SVN



点击Next



选择新建的SVN资源库，点击Next

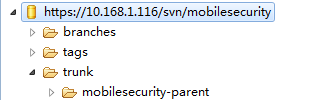


选择你要check in的SVN目录，点击Next，点击Finish。

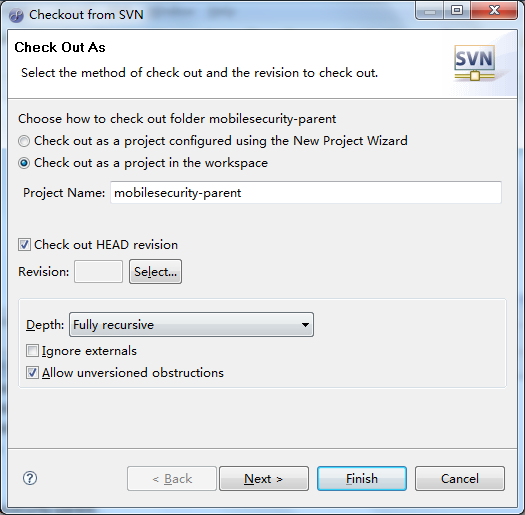
注意：提交项目，只是把src目录下面的内容以及pom.xml文件提交到SVN资源库，不像原来似的，需要把项目中引用的所有jar包提交到SVN。

### SVN check out

从svn资源库中，选择需要检出的项目，右键选择check out

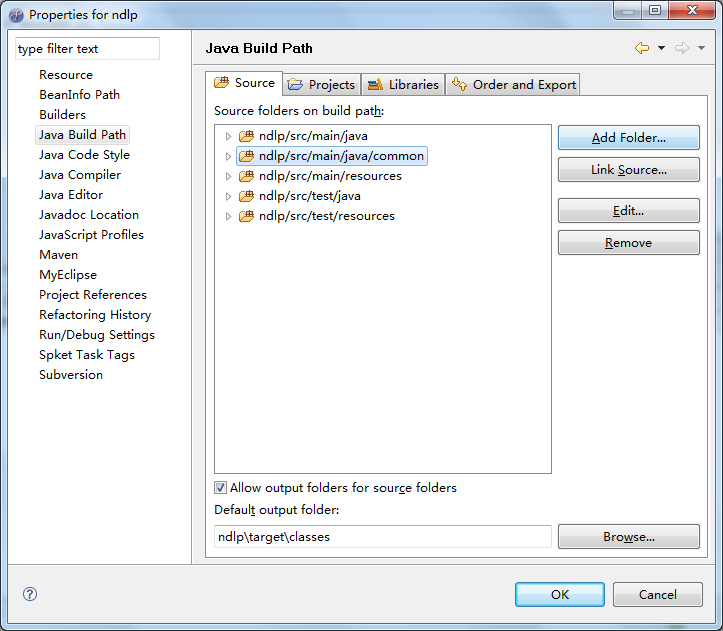


选择第二项，然后点击Next，点击Finish

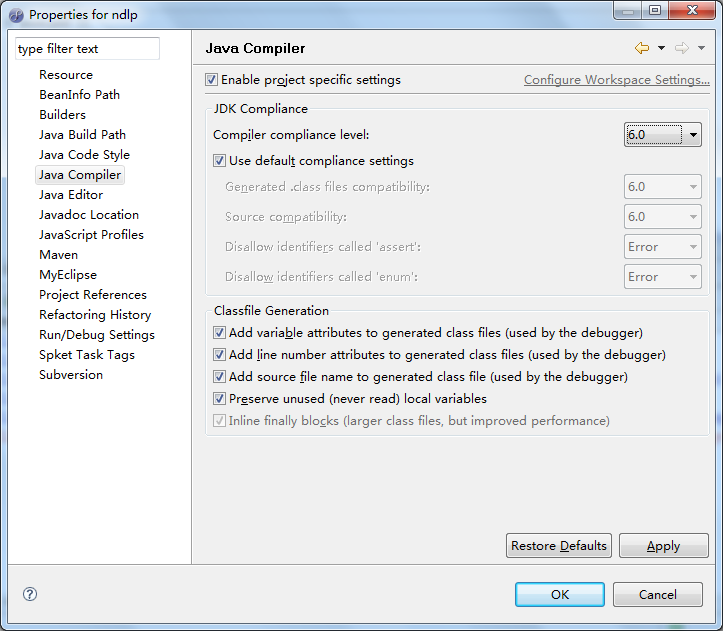


项目检出后，并没有完事，需要做一些简单的配置，才可正常使用。

选择检出的项目，右键🡪Maven🡪选择Enable Dependency Management，添加Maven依赖。



如图所示，修改项目的构建路径，有多个源文件夹的，要通过Add Folder方式将源文件夹展示出来，注意，Default output folder一定得是target\classes目录才正确。



编译器版本，选择最高版本6.0的，然后点击OK。

# Hudson持续集成

## 持续集成与敏捷开发

## Hudson安装

## Hudson用户与权限

## Hudson配置

## 创建一个任务

## Hudson的工作目录

## 邮件反馈