**[Maven的安装、配置及使用入门](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033805.html)**

**咖啡与工具**   
本书相当一部分的内容是在苏州十全街边的Solo咖啡馆完成的，老板Yin亲手烘焙咖啡豆、并能做出据说是苏州最好的咖啡，这小桥流水畔的温馨小 屋能够帮我消除紧张和焦虑，和Yin有一句没一句的聊天也是相当的轻松。Yin还教会了我如何自己研磨咖啡豆、手冲滴率咖啡，让我能够每天在家里也能享受 香气四溢的新鲜咖啡。   
本书的书稿是使用Git和Unfuddle（http://unfuddle.com/）进行管理的，书中的大量截图是通过Jing（http://www.techsmith.com/jing/）制作的。   
JuvenXu   
2010年10月于苏州Solo咖啡   
  
  
**致谢**   
感谢费晓峰，是你最早让我学习使用Maven，并在我开始学习的过程中给予了不少帮助。   
感谢Maven开源社区特别是Maven的创立者Jason Van Zyl，是你们一起创造了如此优秀的开源工具，造福了全世界这么多的开发人员。   
感谢我的家人，一年来，我的大部分原来属于你们的业余时间都给了这本书，感谢你们的理解和支持。   
感谢二少、Garin、Sutra、JTux、红人、linux\_china、Chris、Jdonee、zc0922、还有很多Maven中文社区的朋友，你们给了本书不少建议，并在我写作过程中不断鼓励我和支持我，你们是我写作最大的动力之一。   
最后感谢本书的策划编辑杨福川和曾珊，我从你们身上学到了很多，你们是最专业的、最棒的。

**第1章 Maven简介 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

1.1  何为Maven/2

1.2  为什么需要Maven/4

1.3  Maven与极限编程/7  
1.4  被误解的Maven/8  
1.5  小结/9

**1.1 何为Maven [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven这个词可以翻译为“知识的积累”，也可以翻译为“专 家”或“内行”。本书将介绍Maven这一跨平台的项目管理工具。作为Apache组织中的一个颇为成功的开源项目，Maven主要服务于基于Java平 台的项目构建、依赖管理和项目信息管理。无论是小型的开源类库项目，还是大型的企业级应用；无论是传统的瀑布式开发，还是流行的敏捷模式，Maven都能 大显身手。

**1.1.1  何为构建**

不管你是否意识到，构建（build）是每一位程序员每天都在做的工作。早上来 到公司，我们做的第一件事情就是从源码库签出最新的源码，然后进行单元测试，如果发现失败的测试，会找相关的同事一起调试，修复错误代码。接着回到自己的 工作上来，编写自己的单元测试及产品代码，我们会感激IDE随时报出的编译错误提示。  
忙到午饭时间，代码编写得差不多了，测试也通过了，开心地享 用午餐，然后休息。下午先在昏昏沉沉中开了个例会，会议结束后喝杯咖啡继续工作。刚才在会上经理要求看测试报告，于是找了相关工具集成进IDE，生成了像 模像样的测试覆盖率报告，接着发了一封电子邮件给经理，松了口气。谁料QA小组又发过来了几个bug，没办法，先本地重现再说，于是熟练地用IDE生成了 一个WAR包，部署到Web容器下，启动容器。看到熟悉的界面了，遵循bug报告，一步步重现了bug……快下班的时候，bug修好了，提交代码，通知 QA小组，在愉快中结束了一天的工作。

仔细总结一下，我们会发现，除了编写源代码，我们每天有相当一部分时间花在了编 译、运行单元测试、生成文档、打包和部署等烦琐且不起眼的工作上，这就是构建。如果我们现在还手工这样做，那成本也太高了，于是有人用软件的方法让这一系 列工作完全自动化，使得软件的构建可以像全自动流水线一样，只需要一条简单的命令，所有烦琐的步骤都能够自动完成，很快就能得到最终结果。

**1.1.2  Maven是优秀的构建工具**

前面介绍了Maven的用途之一是服务于构建，它是一个异常强大的构建工具，能 够帮我们自动化构建过程，从清理、编译、测试到生成报告，再到打包和部署。我们不需要也不应该一遍又一遍地输入命令，一次又一次地点击鼠标，我们要做的是 使用Maven配置好项目，然后输入简单的命令(如mvn clean install)，Maven会帮我们处理那些烦琐的任务。

Maven是跨平台的，这意味着无论是在Windows上，还是在Linux或者Mac上，都可以使用同样的命令。

我们一直在不停地寻找避免重复的方法。设计的重复、编码的重复、文档的重复，当 然还有构建的重复。Maven最大化地消除了构建的重复，抽象了构建生命周期，并且为绝大部分的构建任务提供了已实现的插件，我们不再需要定义过程，甚至 不需要再去实现这些过程中的一些任务。最简单的例子是测试，我们没必要告诉Maven去测试，更不需要告诉Maven如何运行测试，只需要遵循Maven 的约定编写好测试用例，当我们运行构建的时候，这些测试便会自动运行。

想象一下，Maven抽象了一个完整的构建生命周期模型，这个模型吸取了大量其 他的构建脚本和构建工具的优点，总结了大量项目的实际需求。如果遵循这个模型，可以避免很多不必要的错误，可以直接使用大量成熟的Maven插件来完成我 们的任务（很多时候我们可能都不知道自己在使用Maven插件）。此外，如果有非常特殊的需求，我们也可以轻松实现自己的插件。

Maven还有一个优点，它能帮助我们标准化构建过程。在Maven之前，十个项目可能有十种构建方式；有了Maven之后，所有项目的构建命令都是简单一致的，这极大地避免了不必要的学习成本，而且有利于促进项目团队的标准化。

综上所述，Maven作为一个构建工具，不仅能帮我们自动化构建，还能够抽象构建过程，提供构建任务实现；它跨平台，对外提供了一致的操作接口，这一切足以使它成为优秀的、流行的构建工具。

**1.1.3  Maven不仅仅是构建工具**

Java不仅是一门编程语言，还是一个平台，通过JRuby和Jython，我 们可以在Java平台上编写和运行Ruby和Python程序。我们也应该认识到，Maven不仅是构建工具，还是一个依赖管理工具和项目信息管理工具。 它提供了中央仓库，能帮我们自动下载构件。

在这个开源的年代里，几乎任何Java应用都会借用一些第三方的开源类库，这些 类库都可通过依赖的方式引入到项目中来。随着依赖的增多，版本不一致、版本冲突、依赖臃肿等问题都会接踵而来。手工解决这些问题是十分枯燥的，幸运的是 Maven提供了一个优秀的解决方案，它通过一个坐标系统准确地定位每一个构件（artifact），也就是通过一组坐标Maven能够找到任何一个 Java类库（如jar文件）。Maven给这个类库世界引入了经纬，让它们变得有秩序，于是我们可以借助它来有序地管理依赖，轻松地解决那些繁杂的依赖 问题。

Maven还能帮助我们管理原本分散在项目中各个角落的项目信息，包括项目描 述、开发者列表、版本控制系统地址、许可证、缺陷管理系统地址等。这些微小的变化看起来很琐碎，并不起眼，但却在不知不觉中为我们节省了大量寻找信息的时 间。除了直接的项目信息，通过Maven自动生成的站点，以及一些已有的插件，我们还能够轻松获得项目文档、测试报告、静态分析报告、源码版本日志报告等 非常具有价值的项目信息。

Maven还为全世界的Java开发者提供了一个免费的中央仓库，在其中几乎可以找到任何的流行开源类库。通过一些Maven的衍生工具（如Nexus），我们还能对其进行快速地搜索。只要定位了坐标，Maven就能够帮我们自动下载，省去了手工劳动。

使用Maven还能享受一个额外的好处，即Maven对于项目目录结构、测试用 例命名方式等内容都有既定的规则，只要遵循了这些成熟的规则，用户在项目间切换的时候就免去了额外的学习成本，可以说是约定优于配置 （Convention Over Configuration）。

**1.2 为什么需要Maven [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven不是Java领域唯一的构建管理的解决方案。本节将通过一些简单的例子解释Maven的必要性，并介绍其他构建解决方案，如IDE、Make和Ant，并将它们与Maven进行比较。

**1.2.1  组装PC和品牌PC**

笔者初中时开始接触计算机，到了高中时更是梦寐以求希望拥有一台自己的计算机。我的第一台计算机是赛扬733的，选购是一个漫长的过程，我先阅读了大量的杂志以了解各类配件的优劣，CPU、内存、主板、显卡，甚至声卡，我都仔细地挑选，后来还跑了很多商家，调货、讨价还价，组装好后自己装操作系统和驱动程序……虽然这花费了我大量时间，但我很享受这个过程。可是事实证明，装出来的机器稳定性不怎么好。

一年前我需要配一台工作站，这时候我已经没有太多时间去研究电脑配件了。我选择了某知名PC供应商的在线商店，大概浏览了一下主流的机型，选择了我需要的配置，然后下单、付款。接着PC供应商帮我组装电脑、安装操作系统和驱动程序。一周后，物流公司将电脑送到我的家里，我接上显示器、电源、鼠标和键盘就能直接使用了。这为我节省了大量时间，而且这台电脑十分稳定，商家在把电脑发送给我之前已经进行了很好的测试。对了，我还能享受两年的售后服务。

使用脚本建立高度自定义的构建系统就像买组装PC，耗时费力，结果也不一定很好。当然，你可以享受从无到有的乐趣，但恐怕实际项目中无法给你那么多时间。使用Maven就像购买品牌PC，省时省力，并能得到成熟的构建系统，还能得到来自于Maven社区的大量支持。唯一与购买品牌PC不同的是，Maven是开源的，你无须为此付费。如果有兴趣，你还能去了解Maven是如何工作的，而我们无法知道那些PC巨头的商业秘密。

**1.2.2  IDE不是万能的**

当然，我们无法否认优秀的IDE能大大提高开发效率。当前主流的IDE如Eclipse和NetBeans等都提供了强大的文本编辑、调试甚至重构功能。虽然使用简单的文本编辑器和命令行也能完成绝大部分开发工作，但很少有人愿意那样做。然而，IDE是有其天生缺陷的：

* IDE依赖大量的手工操作。编译、测试、代码生成等工作都是相互独立的，很难一键完成所有工作。手工劳动往往意味着低效，意味着容易出错。
* 很难在项目中统一所有的IDE配置，每个人都有自己的喜好。也正是由于这个原因，一个在机器A上可以成功运行的任务，到了机器B的IDE中可能就会失败。

我们应该合理利用IDE，而不是过多地依赖它。对于构建这样的任务，在IDE中一次次地点击鼠标是愚蠢的行为。Maven是这方面的专家，而且主流IDE都集成了Maven，我们可以在IDE中方便地运行Maven执行构建。

**1.2.3  Make**

Make也许是最早的构建工具，它由Stuart Feldman于1977年在Bell实验室创建。Stuart Feldman也因此于2003年获得了ACM国际计算机组织颁发的软件系统奖。目前Make有很多衍生实现，包括最流行的GNU Make和BSD Make，还有Windows平台的Microsoft nmake等。

Make由一个名为Makefile的脚本文件驱动，该文件使用Make自己定义的语法格式。其基本组成部分为一系列规则（Rules），而每一条规则又包括目标（Target）、依赖（Prerequisite）和命令（Command）。Makefile的基本结构如下：

1. <span style="font-size: small;">TARGET… : PREREQUISITE…
2. COMMAND
3. …
4. …
5. </span>

     Make通过一系列目标和依赖将整个构建过程串联起来，同时利用本地命令完成每个目标的实际行为。Make的强大之处在于它可以利用所有系统的本地命令，尤其是UNIX/Linux系统，丰富的功能、强大的命令能够帮助Make快速高效地完成任务。

但是，Make将自己和操作系统绑定在一起了。也就是说，使用Make，就不能实现（至少很难）跨平台的构建，这对于Java来说是非常不友好的。此外，Makefile的语法也成问题，很多人抱怨Make构建失败的原因往往是一个难以发现的空格或Tab使用错误。

**1.2.4  Ant**

Ant不是指蚂蚁，而是意指“另一个整洁的工具”（Another Neat Tool），它最早用来构建著名的Tomcat，其作者James Duncan Davidson创作它的动机就是因为受不了Makefile的语法格式。我们可以将Ant看成是一个Java版本的Make，也正因为使用了Java，Ant是跨平台的。此外，Ant使用XML定义构建脚本，相对于Makefile来说，这也更加友好。

与Make类似，Ant有一个构建脚本build.xml，如下所示：

<?xml version="1.0"?>

<project name="Hello" default="compile">

<target name="compile" description="compile the Java source code to class files">

<mkdir dir="classes"/>

<javac srcdir="." destdir="classes"/>

</target>

<target name="jar" depends="compile" description="create a Jar file ">

<jar destfile="hello.jar">

<fileset dir="classes" includes="\*\*/\*.class"/>

<manifest>

<attribute name="Main.Class" value="HelloProgram"/>

</manifest>

</jar>

</target>

</project>

build.xml的基本结构也是目标（target）、依赖（depends），以及实现目标的任务。比如在上面的脚本中，jar目标用来创建应用程序jar文件，该目标依赖于compile目标，后者执行的任务是创建一个名为classes的文件夹，编译当前目录的java文件至classes目录。compile目标完成后，jar目标再执行自己的任务。Ant有大量内置的用Java实现的任务，这保证了其跨平台的特质，同时，Ant也有特殊的任务exec来执行本地命令。

和Make一样，Ant也都是过程式的，开发者显式地指定每一个目标，以及完成该目标所需要执行的任务。针对每一个项目，开发者都需要重新编写这一过程，这里其实隐含着很大的重复。Maven是声明式的，项目构建过程和过程各个阶段所需的工作都由插件实现，并且大部分插件都是现成的，开发者只需要声明项目的基本元素，Maven就执行内置的、完整的构建过程。这在很大程度上消除了重复。

Ant是没有依赖管理的，所以很长一段时间Ant用户都不得不手工管理依赖，这是一个令人头疼的问题。幸运的是，Ant用户现在可以借助Ivy管理依赖。而对于Maven用户来说，依赖管理是理所当然的，Maven不仅内置了依赖管理，更有一个可能拥有全世界最多Java开源软件包的中央仓库，Maven用户无须进行任何配置就可以直接享用。

**1.2.5  不重复发明轮子**

**【该小节内容整理自网友Arthas最早在Maven中文MSN的群内的讨论，在此表示感谢】**

小张是一家小型民营软件公司的程序员，他所在的公司要开发一个新的Web项目。经过协商，决定使用Spring、iBatis和Tapstry。jar包去哪里找呢？公司里估计没有人能把Spring、iBatis和Tapstry所使用的jar包一个不少地找出来。大家的做法是，先到Spring的站点上去找一个spring.with.dependencies，然后去iBatis的网站上把所有列出来的jar包下载下来，对Tapstry、Apache commons等执行同样的操作。项目还没有开始，WEB.INF/lib下已经有近百个jar包了，带版本号的、不带版本号的、有用的、没用的、相冲突的，怎一个“乱”字了得！

在项目开发过程中，小张不时地发现版本错误和版本冲突问题，他只能硬着头皮逐一解决。项目开发到一半，经理发现最终部署的应用的体积实在太大了，要求小张去掉一些没用的jar包，于是小张只能加班加点地一个个删……

小张隐隐地觉得这些依赖需要一个框架或者系统来进行管理。

小张喜欢学习流行的技术，前几年Ant十分流行，他学了，并成为了公司这方面的专家。小张知道，Ant打包，无非就是创建目录，复制文件，编译源代码，使用一堆任务，如copydir、fileset、classpath、ref、target，然后再jar、zip、war，打包就成功了。

项目经理发话了：“兄弟们，新项目来了，小张，你来写Ant脚本！”

“是，保证完成任务！”接着，小张继续创建一个新的XML文件。target clean; target compile; target jar; …… 不知道他是否想过，在他写的这么多的Ant脚本中，有多少是重复劳动，有多少代码会在一个又一个项目中重现。既然都差不多，有些甚至完全相同，为什么每次都要重新编写？

终于有一天，小张意识到了这个问题，想复用Ant脚本，于是在开会时他说：“以后就都用我这个规范的Ant脚本吧，新的项目只要遵循我定义的目录结构就可以了。”经理听后觉得很有道理：“嗯，确实是个进步。”

这时新来的研究生发言了：“经理，用Maven吧，这个在开源社区很流行，比Ant更方便。”小张一听很惊讶，Maven真比自己的“规范化Ant”强大？其实他不知道自己只是在重新发明轮子，Maven已经有一大把现成的插件，全世界都在用，你自己不用写任何代码！

为什么没有人说“我自己写的代码最灵活，所以我不用Spring，我自己实现IoC；我不用Hibernate，我自己封装JDBC”？

**1.3 Maven与极限编程 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

极限编程（XP）是近些年在软件行业红得发紫的敏捷开发方法，它强调拥抱变化。该软件开发方法的创始人Kent Beck提出了XP所追求的价值、实施原则和推荐实践。下面看一下*Maven*是如何适应XP的。  
首先看一下Maven如何帮助XP团队实现一些核心价值：

* 简单。Maven暴露了一组一致、简洁的操作接口，能帮助团队成员从原来的高度自定义的、复杂的构建系统中解脱出来，使用Maven现有的成熟的、稳定的组件也能简化构建系统的复杂度。
* 交流与反馈。与版本控制系统结合后，所有人都能执行最新的构建并快速得到反馈。此外，自动生成的项目报告也能帮助成员了解项目的状态，促进团队的交流。

此外，Maven更能无缝地支持或者融入到一些主要的XP实践中：

* 测试驱动开发（TDD）。TDD强调测试先行，所有产品都应该由测试用例覆盖。而测试是Maven生命周期的最重要的组成部分之一，并且Maven有现成的成熟插件支持业界流行的测试框架，如JUnit和TestNG。
* 十分钟构建。十分钟构建强调我们能够随时快速地从源码构建出最终的产品。这正是Maven所擅长的，只需要一些配置，之后用一条简单的命令就能让Maven帮你清理、编译、测试、打包、部署，然后得到最终的产品。
* 持续集成（CI）。CI强调项目以很短的周期（如15分钟）集成最新的代码。 实际上,CI的前提是源码管理系统和构建系统。目前业界流行的CI服务器如Hudson和CruiseControl都能很好地和Maven进行集成。也 就是说，使用Maven后，持续集成会变得更加方便。
* 富有信息的工作区。 这条实践强调开发者能够快速方便地了解到项目的最新状态。当然，Maven并不会帮你把测试覆盖率报告贴到墙上，也不会在你的工作台上放个鸭子告诉你构建 失败了。不过使用Maven发布的项目报告站点，并配置你需要的项目报告，如测试覆盖率报告，都能帮你把信息推送到开发者眼前。

上述这些实践并非只在XP中适用。事实上，除了其他敏捷开发方法如SCRUM之外，几乎任何软件开发方法都能借鉴这些实践。也就是说，Maven几乎能够很好地支持任何软件开发方法。  
例 如，在传统的瀑布模型开发中，项目依次要经历需求开发、分析、设计、编码、测试和集成发布阶段。从设计和编码阶段开始，就可以使用Maven来建立项目的 构建系统。在设计阶段，也完全可以针对设计开发测试用例，然后再编写代码来满足这些测试用例。然而，有了自动化构建系统，我们可以节省很多手动的测试时 间。此外，尽早地使用构建系统集成团队的代码，对项目也是百利而无一害。最后，Maven还能帮助我们快速地发布项目。

**第2章 Maven的安装和配置 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

第1章介绍了Maven是什么，以及为什么要使用Maven， 我们将从本章实际开始实际接触Maven。本章首先将介绍如何在主流的操作系统下安装Maven，并详细解释Maven的安装文件；其次还会介绍如何在主 流的IDE中集成Maven，以及Maven安装的最佳实践。  
2.1 在Windows上安装Maven  
2.2 在基于Unix的系统上安装Maven  
2.3 安装目录分析  
2.4 设置HTTP代理  
2.5 安装m2eclipse  
2.6 安装NetBeans Maven插件  
2.7 Maven安装最佳实践  
2.8 小结

**2.1 在Windows上安装Maven [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

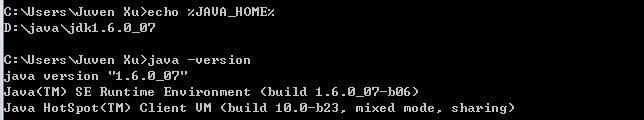
**2.1.1 检查JDK安装**

在安装Maven之前，首先要确认你已经正确安装了JDK。Maven可以运行在JDK 1.4及以上的版本上。本书的所有样例都基于JDK 5及以上版本。打开Windows的命令行，运行如下的命令来检查你的Java安装：

C:\Users\Juven Xu>echo %JAVA\_HOME%

C:\Users\Juven Xu>java -version

结果如图2-1所示：

  
 图2-1 Windows中检查Java安装

上述命令首先检查环境变量JAVA\_HOME是否指向了正确的JDK目录，接着尝试运行java命令。如果Windows无法执行java命令，或者无法找到JAVA\_HOME环境变量。你就需要检查Java是否安装了，或者环境变量是否设置正确。关于环境变量的设置，请参考2.1.3节。

**2.1.2 下载Maven**

请访问Maven的下载页面：<http://maven.apache.org/download.html>，其中包含针对不同平台的各种版本的Maven下载文件。对于首次接触Maven的读者来说，推荐使用Maven 3.0，，因此下载apache-maven-3.0-bin.zip。当然，如果你对Maven的源代码感兴趣并想自己构建Maven，还可以下载apache-maven-3.0 -src.zip。该下载页面还提供了md5校验和（checksum）文件和asc数字签名文件，可以用来检验Maven分发包的正确性和安全性。

在本书编写的时候，Maven 2的最新版本是2.2.1，Maven 3基本完全兼容Maven 2，而且较之于Maven 2它性能更好，还有不少功能的改进，如果你之前一直使用Maven 2，现在正犹豫是否要升级，那就大可不必担心了，快点尝试下Maven 3吧！

**2.1.3 本地安装**

将安装文件解压到你指定的目录中，如：

D:\bin>jar xvf "C:\Users\Juven Xu\Downloads\apache-maven-3.0--bin.zip"

这里的Maven安装目录是D:\bin\apache-maven-3.0，接着需要设置环境变量，将Maven安装配置到操作系统环境中。

打开系统属性面板（桌面上右键单击“我的电脑”→“属性”），点击**高级系统设置**，再点击**环境变量**，在**系统变量**中新建一个变量，变量名为*M2\_HOME*，变量值为Maven的安装目录*D:\bin\apache-maven-3.0*。点击**确定**，接着在系统变量中找到一个名为Path的变量，在变量值的末尾加上*%M2\_HOME%\bin;*，注意多个值之间需要有分号隔开，然后点击**确定**。至此，环境变量设置完成，详细情况如图2-2所示：



图2-2 Windows中系统环境变量配置

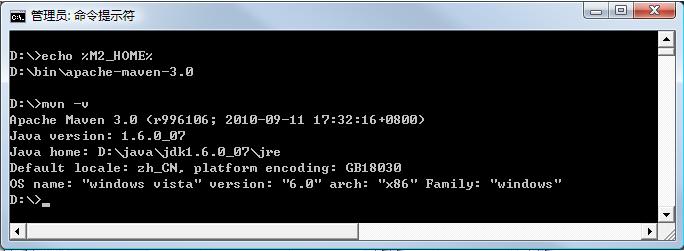
这里需要提一下的是Path环境变量，当我们在cmd中输入命令时，Windows首先会在当前目录中寻找可执行文件或脚本，如果没有找到，Windows会接着遍历环境变量Path中定义的路径。由于我们将*%M2\_HOME%\bin*添加到了Path中，而这里%M2\_HOME%实际上是引用了我们前面定义的另一个变量，其值是Maven的安装目录。因此，Windows会在执行命令时搜索目录*D:\bin\apache-maven-3.0\bin*，而mvn执行脚本的位置就是这里。

明白了环境变量的作用，现在打开一个新的cmd窗口（这里强调新的窗口是因为新的环境变量配置需要新的cmd窗口才能生效），运行如下命令检查Maven的安装情况：

C:\Users\Juven Xu>**echo %M2\_HOME%**

C:\Users\Juven Xu>mvn -v

运行结果如图2-3所示：

  
 图2-3 Windows中检查Maven安装

第一条命令**echo %M2\_HOME%**用来检查环境变量M2\_HOME是否指向了正确的Maven安装目录；而**mvn –version**执行了第一条Maven命令，以检查Windows是否能够找到正确的mvn执行脚本。

**2.1.4 升级Maven**

Maven还比较年轻，更新比较频繁，因此用户往往会需要更新Maven安装以获得更多更酷的新特性，以及避免一些旧的bug。

在Windows上更新Maven非常简便，只需要下载新的Maven安装文件，解压至本地目录，然后更新M2\_HOME环境变量便可。例如，假设Maven推出了新版本3.1，我们将其下载然后解压至目录*D:\bin\apache-maven-3.1*，接着遵照前一节描述的步骤编辑环境变量M2\_HOME，更改其值为D:\bin\apache-maven-3.1。至此，更新就完成了。同理，如果你需要使用某一个旧版本的Maven，也只需要编辑M2\_HOME环境变量指向旧版本的安装目录。

**2.2 在基于Unix的系统上安装Maven [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven是跨平台的，它可以在任何一种主流的操作系统上运行，本节将介绍如何在基于Unix的系统（包括Linux、Mac OS以及FreeBSD等）上安装Maven。

**2.2.1 下载和安装**

首先，与在Windows上安装Maven一样，需要检查JAVA\_HOME环境变量以及Java命令，细节不再赘述，命令如下：

juven@juven-ubuntu:~$ echo $JAVA\_HOME

juven@juven-ubuntu:~$ java –version

运行结果如图2-4所示：

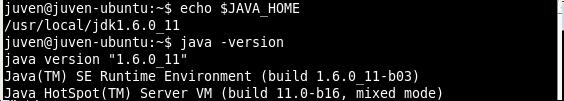


图2-4 Linux中检查Java安装

接着到http://maven.apache.org/download.html下载Maven安装文件，如apache-maven-3.0-bin.tar.gz，然后解压到本地目录：

juven@juven-ubuntu:bin$ tar -xvzf apache-maven-3.0-bin.tar.gz

现在已经创建好了一个Maven安装目录apache-maven-3.0，虽然直接使用该目录配置环境变量之后就能使用Maven了，但这里我更推荐做法是，在安装目录旁平行地创建一个符号链接，以方便日后的升级：

juven@juven-ubuntu:bin$ ln -s apache-maven-3.0 apache-maven  
juven@juven-ubuntu:bin$ ls -l  
total 4  
lrwxrwxrwx 1 juven juven   18 2009-09-20 15:43 apache-maven -> apache-maven-3.0  
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-3.0

接下来，我们需要设置M2\_HOME环境变量指向符号链接apache-maven-，并且把Maven安装目录下的bin/文件夹添加到系统环境变量PATH中去：

juven@juven-ubuntu:bin$ export M2\_HOME=/home/juven/bin/apache-maven  
juven@juven-ubuntu:bin$ export PATH=$PATH:$M2\_HOME/bin

一般来说，需要将这两行命令加入到系统的登录shell脚本中去，以我现在的Ubuntu 8.10为例，编辑~/.bashrc文件，添加这两行命令。这样，每次启动一个终端，这些配置就能自动执行。

至此，安装完成，我们可以运行以下命令检查Maven安装：

juven@juven-ubuntu:bin$ echo $M2\_HOME

juven@juven-ubuntu:bin$ mvn –version

运行结果如图2-5所示：

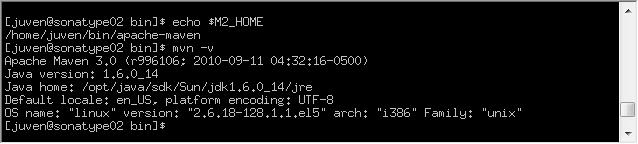


 图2-5 Linux中检查Maven安装

**2.2.2 升级Maven**

在基于Unix的系统上，可以利用符号链接这一工具来简化Maven的升级，不必像在Windows上那样，每次升级都必须更新环境变量。

前一小节中我们提到，解压Maven安装包到本地之后，平行地创建一个符号链接，然后在配置环境变量时引用该符号链接，这样做是为了方便升级。现在，假设我们需要升级到新的Maven 3.1版本，同理，将安装包解压到与前一版本平行的目录下，然后更新符号链接指向3.1版的目录便可：

juven@juven-ubuntu:bin$ rm apache-maven  
juven@juven-ubuntu:bin$ ln -s apache-maven-3.1/ apache-maven  
juven@juven-ubuntu:bin$ ls -l  
total 8  
lrwxrwxrwx 1 juven juven   17 2009-09-20 16:13 apache-maven -> apache-maven-3.1 /  
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-3.0drwxr-xr-x 2 juven juven 4096 2009-09-20 16:09 apache-maven-3.1

同理，可以很方便地切换到Maven的任意一个版本。现在升级完成了，可以运行**mvn -v**进行检查。

**2.3 安装目录分析 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章前面的内容讲述了如何在各种操作系统中安装和升级Maven。现在我们来仔细分析一下Maven的安装文件。

**2.3.1 M2\_HOME**

前面我们讲到设置M2\_HOME环境变量指向Maven的安装目录，本书之后所有使用M2\_HOME的地方都指代了该安装目录，让我们看一下该目录的结构和内容：

bin  
boot  
conf  
lib  
LICENSE.txt  
NOTICE.txt  
README.txt

* Bin： 该目录包含了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。其中*mvn*是基于UNIX平台的shell脚本，*mvn.bat*是基于Windows平台的bat脚本。在命令行输入任何一条mvn命令时，实际上就是在调用这些脚本。该目录还包含了*mvnDebug*和*mvnDebug.bat*两个文件，同样，前者是UNIX平台的shell脚本，后者是windows的bat脚本。那么mvn和mvnDebug有什么区别和关系呢？打开文件我们就可以看到，两者基本是一样的，只是mvnDebug多了一条MAVEN\_DEBUG\_OPTS配置，作用就是在运行Maven时开启debug，以便调试Maven本身。此外，该目录还包含*m2.conf*文件，这是classworlds的配置文件，稍微会介绍classworlds。
* 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*。直接修改该文件，就能在机器上全局地定制Maven的行为。一般情况下，我们更偏向于复制该文件至*~/.m2/*目录下（这里~表示用户目录），然后修改该文件，在用户范围定制Maven的行为。本书的后面将会多次提到该settings.xml，并逐步分析其中的各个元素。
* 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。（对于Maven 2来说，该目录只包含一个如*maven-2.2.1-uber.jar*的文件原本各为独立JAR文件的Maven模块和第三方类库都被拆解后重新合并到了这个JAR文件中）。可以说，这个lib目录就是真正的Maven。关于该文件，还有一点值得一提的是，用户可以在这个目录中找到Maven内置的超级POM，这一点在8.5小节详细解释。其他： *LICENSE.txt*记录了Maven使用的软件许可证Apache License Version 2.0； *NOTICE.txt*记录了Maven包含的第三方软件；而*README.txt*则包含了Maven的简要介绍，包括安装需求及如何安装的简要指令等等。

**2.3.2 ~/.m2**

在讲述该小节之前，我们先运行一条简单的命令：**mvn help:system**。该命令会打印出所有的Java系统属性和环境变量，这些信息对我们日常的编程工作很有帮助。这里暂不解释help:system涉及的语法，运行这条命令的目的是为了让Maven执行一个真正的任务。我们可以从命令行输出看到Maven会下载maven-help-plugin，包括pom文件和jar文件。这些文件都被下载到了Maven本地仓库中。

现在打开用户目录，比如当前的用户目录是*C:\Users\Juven Xu\*，你可以在Vista和Windows7中找到类似的用户目录。如果是更早版本的Windows，该目录应该类似于C:\Document and Settings\Juven Xu\。在基于Unix的系统上，直接输入**cd** 回车，就可以转到用户目录。为了方便，本书统一使用符号 **~** 指代用户目录。

在用户目录下，我们可以发现.m2文件夹。默认情况下，该文件夹下放置了Maven本地仓库.m2/repository。所有的Maven构件（artifact）都被存储到该仓库中，以方便重用。我们可以到*~/.m2/repository/org/apache/maven/plugins/maven-help-plugins/*目录下找到刚才下载的maven-help-plugin的pom文件和jar文件。Maven根据一套规则来确定任何一个构件在仓库中的位置，这一点本书第6章将会详细阐述。由于Maven仓库是通过简单文件系统透明地展示给Maven用户的，有些时候可以绕过Maven直接查看或修改仓库文件，在遇到疑难问题时，这往往十分有用。

默认情况下，*~/.m2*目录下除了repository仓库之外就没有其他目录和文件了，不过大多数Maven用户需要复制*M2\_HOME/conf/settings.xml*文件到*~/.m2/settings.xml*。这是一条最佳实践，我们将在本章最后一小节详细解释。

**2.4 设置HTTP代理 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

有时候你所在的公司由于安全因素考虑，要求你使用通过安全认证的代理访问因特网。这种情况下，就需要为Maven配置HTTP代理，才能让它正常访问外部仓库，以下载所需要的资源。

首先确认自己无法直接访问公共的Maven中央仓库，直接运行命令**ping repo1.maven.org**可以检查网络。如果真的需要代理，先检查一下代理服务器是否畅通，比如现在有一个IP地址为218.14.227.197，端口为3128的代理服务，我们可以运行**telnet 218.14.227.197 3128**来检测该地址的该端口是否畅通。如果得到出错信息，需要先获取正确的代理服务信息；如果telnet连接正确，则输入ctrl+]，然后q，回车，退出即可。

检查完毕之后，编辑~/.m2/settings.xml文件（如果没有该文件，则复制$M2\_HOME/conf/settings.xml）。添加代理配置如下：

<settings>  
…    
<proxies>

    <proxy>

      <id>my-proxy</id>

      <active>true</active>

      <protocol>http</protocol>

      <host>218.14.227.197</host>

      <port>3128</port>

      <!--

      <username>\*\*\*</username>

      <password>\*\*\*</password>

      <nonProxyHosts>repository.mycom.com|\*.google.com</nonProxyHosts>

      -->

    </proxy>

  </proxies>  
  …  
</settings>

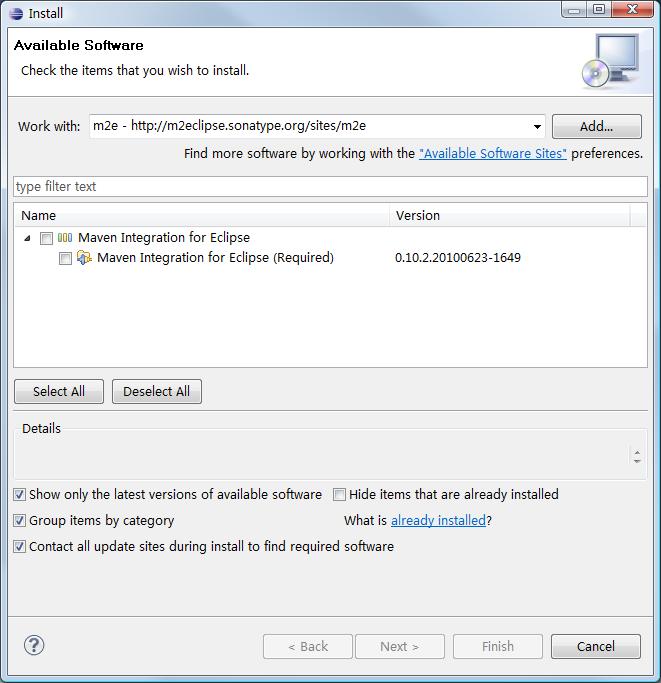
这段配置十分简单，proxies下可以有多个proxy元素，如果你声明了多个proxy元素，则默认情况下第一个被激活的proxy会生效。这里声明了一个id为my-proxy的代理，active的值为true表示激活该代理，protocol表示使用的代理协议，这里是http。当然，最重要的是指定正确的主机名（host元素）和端口（port元素）。上述XML配置中我注释掉了username、password、nonProxyHost几个元素，当你的代理服务需要认证时，就需要配置username和password。nonProxyHost元素用来指定哪些主机名不需要代理，可以使用 | 符号来分隔多个主机名。此外，该配置也支持通配符，如\*.google.com表示所有以google.com结尾的域名访问都不要通过代理。

**2.5 安装m2eclipse [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

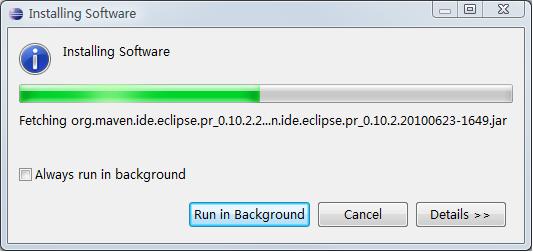
Eclipse是一款非常优秀的IDE。除了基本的语法标亮、代码补齐、XML编辑等基本功能外，最新版的Eclipse还能很好地支持重构，并且集成了JUnit、CVS、Mylyn等各种流行工具。可惜Eclipse默认没有集成对Maven的支持。幸运的是，由Maven之父Jason Van Zyl创立的Sonatype公司建立了m2eclipse项目，这是Eclipse下的一款十分强大的Maven插件，可以访问<http://m2eclipse.sonatype.org/> 了解更多该项目的信息。

本小节将先介绍如何安装m2eclipse插件，本书后续的章节会逐步介绍m2eclipse插件的使用。

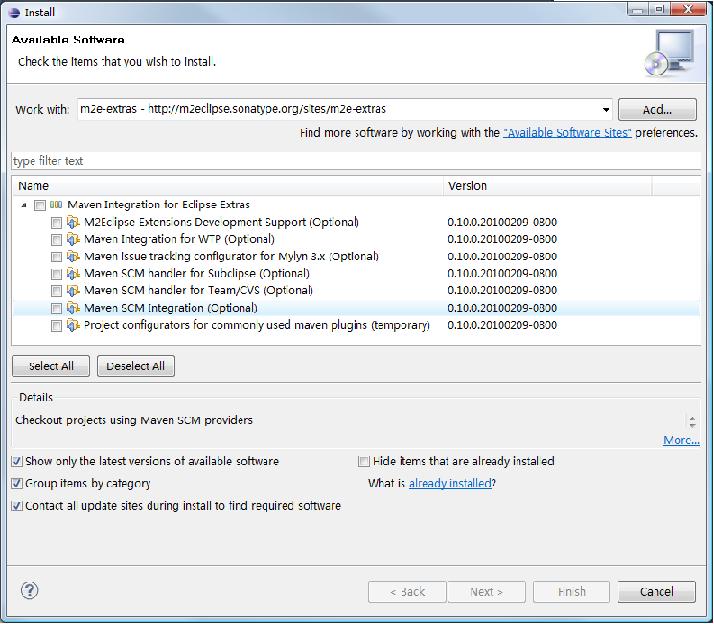
现在我以Eclipse 3.6为例逐步讲解m2eclipse的安装。启动Eclipse之后，在菜单栏中选择**Help**，然后选择**Install New Software…**，接着你会看到一个Install对话框，点击**Work with:**字段边上的**Add按钮**，你会得到一个新的Add Repository对话框，在**Name**字段中输入*m2e*，Location字段中输入<http://m2eclipse.sonatype.org/sites/m2e>，然后点击**OK**。Eclipse会下载m2eclipse安装站点上的资源信息。等待资源载入完成之后，我们再将其全部展开，就能看到图2-6所示的界面：

  
 图2-6  m2eclipse的核心安装资源列表

如图显示了m2eclipse的核心模块Maven Integration for Eclipse (Required)，选择后点击**Next >**，Eclipse会自动计算模块间依赖，然后给出一个将被安装的模块列表，确认无误后，继续点击**Next >**，这时我们会看到许可证信息，m2eclipse使用的开源许可证是Eclipse Public License v1.0，选择**I accept the terms of the license agreements**，然后点击**Finish**，接着就耐心等待Eclipse下载安装这些模块，如图2-7所示：

  
 图2-7：m2eclipse安装进度

除了核心组件之外，m2eclipse还提供了一组额外组件，主要是为了方便与其它工具如Subversion进行集成，这些组件的安装地址为<http://m2eclipse.sonatype.org/sites/m2e-extras>。使用前面类似的安装方法，我们可以看到如图2-8的组件列表：

  
 图2-8：m2eclipse的额外组件安装资源列表

下面简单解释一下这些组件的用途：

**1. 重要的**

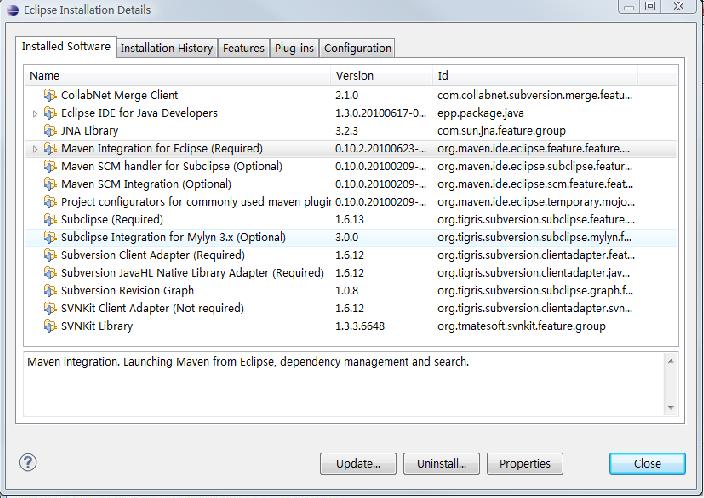
* Maven SCM handler for Subclipse  (Optional）：Subversion是非常流行的版本管理工具，该模块能够帮助我们直接从Subversion服务器签出Maven项目，不过前提是需要首先安装Subclipse（<http://subclipse.tigris.org/>）。
* Maven SCM Integration (Optional）：Eclipse环境中Maven与SCM集成核心的模块，它利用各种SCM工具如SVN实现Maven项目的签出和具体化等操作。

**2. 不重要的**

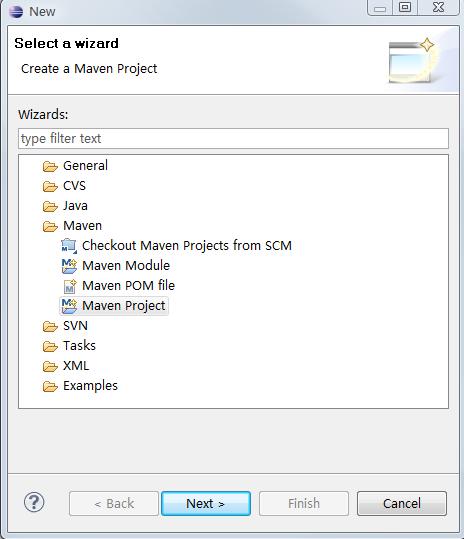
* Maven issue tracking configurator for Mylyn 3.x (Optional）：该模块能够帮助我们使用POM中的缺陷跟踪系统信息连接Mylyn至服务器。
* Maven SCM handler for Team/CVS (Optional）：该模块帮助我们从CVS服务器签出Maven项目，如果你还在使用CVS，就需要安装它。
* Maven Integration for WTP (Optional）：使用该模块可以让Eclipse自动读取POM信息并配置WTP项目。、
* M2eclipse Extensions Development Support (Optional)：用来支持扩展m2eclipse，一般用户不会用到。
* Project configurators for commonly used maven plugins (temporary)：一个临时的组件，用来支持一些Maven插件与Eclipse的集成，建议安装。

读者可以根据自己的需要安装相应组件，具体步骤不再赘述。

待安装完毕后，重启Eclipse，现在让我们验证一下m2eclipse是否正确安装了。首先，点击菜单栏中的**Help**，然后选择**About Eclipse**，在弹出的对话框中，点击**Installation Details**按钮，会得到一个对话框，在**Installed Software**标签栏中，检查刚才我们选择的模块是否在这个列表中，如图2-9所示：

  
 图2-9m2eclipse安装结果

如果一切没问题，我们再检查一下Eclipse现在是否已经支持创建Maven项目，依次点击菜单栏中的**File**→**New**→**Other**，在弹出的对话框中，找到Maven一项，再将其展开，你应该能够看到如图2-10所示的对话框：

  
 图2-10 Eclipse中创建Maven项目向导

如果一切正常，说明m2eclipse已经正确安装了。

最后，关于m2eclipse的安装，需要提醒的一点是，你可能会在使用m2eclipse时遇到类似这样的错误：

09-10-6 上午01时14分49秒: Eclipse is running in a JRE, but a JDK is required  
Some Maven plugins may not work when importing projects or updating source folders.

这是因为Eclipse默认是运行在JRE上的，而m2eclipse的一些功能要求使用JDK，解决方法是配置Eclipse安装目录的eclipse.ini文件，添加vm配置指向JDK，如：

--launcher.XXMaxPermSize

256m

-vm

D:\java\jdk1.6.0\_07\bin\javaw.exe

-vmargs

-Dosgi.requiredJavaVersion=1.5

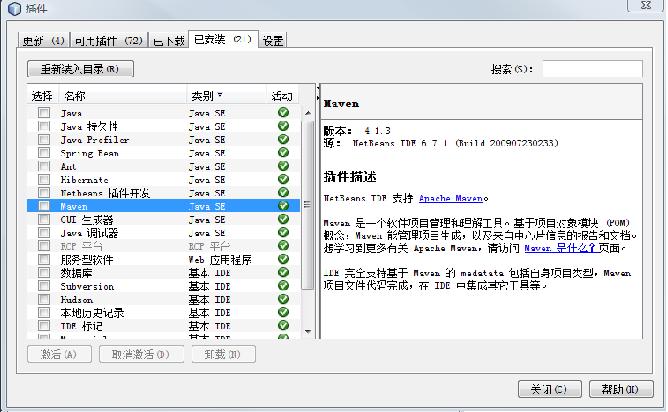
-Xms128m

-Xmx256m

**2.6 安装NetBeans Maven插件 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本小节会先介绍如何在NetBeans上安装Maven插件，后面的章节中还会介绍NetBeans中具体的Maven操作。

首先，如果你正在使用NetBeans 6.7及以上版本，那么Maven插件已经预装了。你可以检查Maven插件安装，点击菜单栏中的**工具**，接着选择**插件**，在弹出的插件对话框中选择**已安装**标签，你应该能够看到Maven插件，如图2-11所示：

  
 图2-11 已安装的NetBeans Maven插件

如果你在使用NetBeans 6.7之前的版本，或者由于某些原因NetBeans Maven插件被卸载了，那么你就需要安装NetBeans Maven插件，下面我们以NetBeans 6.1为例，介绍Maven插件的安装。

同样，点击菜单栏中的**工具**，选择**插件**，在弹出的插件对话框中选择**可用插件**标签，接着在右边的搜索框内输入Maven，这时你会在左边的列表中看到一个名为Maven的插件，选择该插件，然后点击下面的安装按钮，如图2-12所示：



图2-12 安装NetBeans Maven插件

接着在随后的对话框中根据提示操作，阅读相关许可证并接受，NetBeans会自动帮我们下载并安装Maven插件，结束之后会提示安装完成，之后再点击插件对话框的**已安装**标签，就能看到已经激活的Maven插件。

最后，为了确认Maven插件确实已经正确安装了，可以看一下NetBeans是否已经拥有创建Maven项目的相关菜单。在菜单栏中选择**文件**，然后选择**新建项目**，这时应该能够看到项目类别中有**Maven**一项，选择该类别，右边会相应地显示**Maven项目**和**基于现有POM的Maven项目**，如图2-13所示：

  
 图2-13 NetBeans中创建Maven项目向导

如果你能看到类似的对话框，说明NetBeans Maven已经正确安装了。

**2.7 Maven安装最佳实践 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本节介绍一些在安装Maven过程中不是必须的，但十分有用的实践。

**2.7.1 设置MAVEN\_OPTS环境变量**

本章前面介绍Maven安装目录时我们了解到，运行mvn命令实际上是执行了Java命令，既然是运行Java，那么运行Java命令可用的参数当然也应该在运行mvn命令时可用。这个时候，MAVEN\_OPTS环境变量就能派上用场。

我们通常需要设置MAVEN\_OPTS的值为：*-Xms128m -Xmx512m*，因为Java默认的最大可用内存往往不能够满足Maven运行的需要，比如在项目较大时，使用Maven生成项目站点需要占用大量的内存，如果没有该配置，我们很容易得到*java.lang.OutOfMemeoryError*。因此，一开始就配置该变量是推荐的做法。

关于如何设置环境变量，请参考前面设置M2\_HOME环境变量的做法，尽量不要直接修改mvn.bat或者mvn这两个Maven执行脚本文件。因为如果修改了脚本文件，升级Maven时你就不得不再次修改，一来麻烦，二来容易忘记。同理，我们应该尽可能地不去修改任何Maven安装目录下的文件。

**2.7.2 配置用户范围settings.xml**

Maven用户可以选择配置*$M2\_HOME/conf/settings.xml*或者*~/.m2/settings.xml*。前者是全局范围的，整台机器上的所有用户都会直接受到该配置的影响，而后者是用户范围的，只有当前用户才会受到该配置的影响。

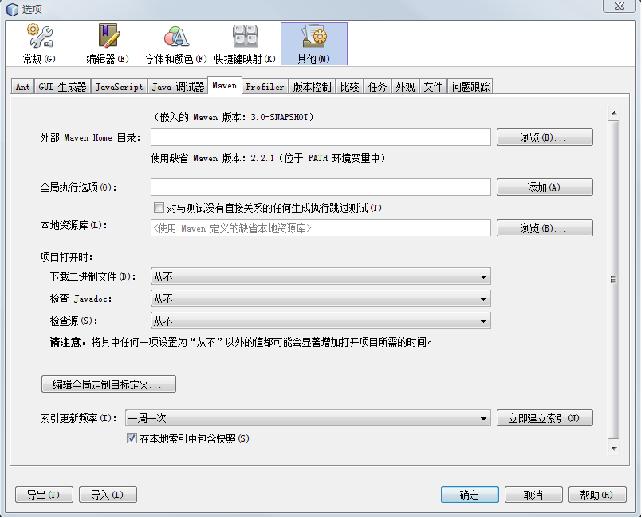
我们推荐使用用户范围的settings.xml，主要原因是为了避免无意识地影响到系统中的其他用户。当然，如果你有切实的需求，需要统一系统中所有用户的settings.xml配置，当然应该使用全局范围的settings.xml。

除了影响范围这一因素，配置用户范围settings.xml文件还便于Maven升级。直接修改conf目录下的settings.xml会导致Maven升级不便，每次升级到新版本的Maven，都需要复制settings.xml文件，如果使用*~/.m2*目录下的settings.xml，就不会影响到Maven安装文件，升级时就不需要触动settings.xml文件。

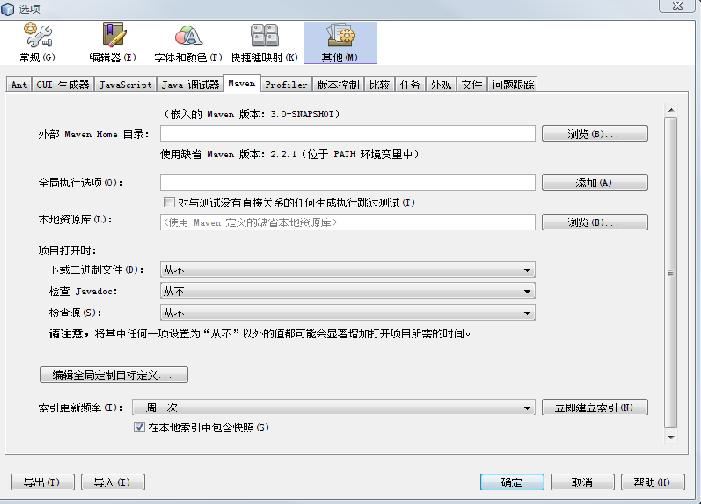
**2.7.3 不要使用IDE内嵌的Maven**

无论是Eclipse还是NetBeans，当我们集成Maven时，都会安装上一个内嵌的Maven，这个内嵌的Maven通常会比较新，但不一定很稳定，而且往往也会和我们在命令行使用的Maven不是同一个版本。这里有会出现两个潜在的问题：首先，较新版本的Maven存在很多不稳定因素，容易造成一些难以理解的问题；其次，除了IDE，我们也经常还会使用命令行的Maven，如果版本不一致，容易造成构建行为的不一致，这是我们所不希望看到的。因此，我们应该在IDE中配置Maven插件时使用与命令行一致的Maven。

在m2eclipse环境中，点击菜单栏中的**Windows**，然后选择**Preferences**，在弹出的对话框中，展开左边的**Maven**项，选择**Installation**子项，在右边的面板中，我们能够看到有一个默认的**Embedded** Maven安装被选中了，点击**Add…**然后选择我们的Maven安装目录M2\_HOME，添加完毕之后选择这一个外部的Maven，如图2-14所示：

  
 图2-14 在Eclipse中使用外部Maven

NetBeans Maven插件默认会侦测PATH环境变量，因此会直接使用与命令行一致的Maven环境。依次点击菜单栏中的**工具**→**选项**→**其他**→**Maven**标签栏，你就能看到如图2-15所示的配置：

  
 图2-15 在NetBeans中使用外部Maven

**2.8 小结 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章详细介绍了在各种操作系统平台上安装Maven，并对Maven安装目录进行了深入的分析，在命令行的基础上，本章又进一步介绍了Maven与主流IDE Eclipse及NetBeans的集成，本章最后还介绍了一些与Maven安装相关的最佳实践。本书下一章会创建一个Hello World项目，带领读者配置和构建Maven项目。

**第3章 Maven使用入门 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

到目前为止，我们已经大概了解并安装好了Maven，现在，我们开始 创建一个最简单的Hello World项目。如果你是初次接触Maven，我建议你按照本章的内容一步步地编写代码并执行，可能你会碰到一些概念暂时难以理解，不用着急，记下这些疑 难点，相信本书的后续章节会帮你逐一解答。   
3.1 编写POM   
3.2 编写主代码   
3.3 编写测试代码   
3.4 打包和运行   
3.5 使用Archetype生成项目骨架   
3.6 m2eclipse简单使用   
3.7 NetBeans Maven插件简单使用   
3.8 小结

**3.1 编写POM [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

就像Make的Makefile，Ant的build.xml一样，Maven项目的核心是pom.xml。POM（**P**roject **O**bject **M**odel，项目对象模型）定义了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。现在我们先为Hello World项目编写一个最简单的pom.xml。

首先创建一个名为hello-world的文件夹（本书中各章的代码都会对应一个以ch开头的项目），打开该文件夹，新建一个名为pom.xml的文件，输入其内容如代码清单3-1：

代码清单3-1：Hello World的POM

1. <span style="font-size: small;"><?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5. http://maven.apache.org/maven-v4\_0\_0.xsd">
6. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.juvenxu.mvnbook</groupId>
8. <artifactId>hello-world</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Hello World Project</name>
11. </project>
12. </span>

     代码的第一行是XML头，指定了该xml文档的版本和编码方式。紧接着是project元素，project是所有pom.xml的根元素，它还声明了一些POM相关的命名空间及xsd元素，虽然这些属性不是必须的，但使用这些属性能够让第三方工具（如IDE中的XML编辑器）帮助我们快速编辑POM。

根元素下的第一个子元素modelVersion指定了当前POM模型的版本，对于Maven2及Maven 3来说，它只能是4.0.0。

这段代码中最重要的是groupId，artifactId和version三行。这三个元素定义了一个项目基本的坐标，在Maven的世界，任何的jar、pom或者war都是以基于这些基本的坐标进行区分的。

groupId定义了项目属于哪个组，这个组往往和项目所在的组织或公司存在关联，譬如你在googlecode上建立了一个名为myapp的项目，那么groupId就应该是com.googlecode.myapp，如果你的公司是mycom，有一个项目为myapp，那么groupId就应该是com.mycom.myapp。本书中所有的代码都基于groupId com.juvenxu.mvnbook。

artifactId定义了当前Maven项目在组中唯一的ID，我们为这个Hello World项目定义artifactId为hello-world，本书其他章节代码会被分配其他的artifactId。而在前面的groupId为com.googlecode.myapp的例子中，你可能会为不同的子项目（模块）分配artifactId，如：myapp-util、myapp-domain、myapp-web等等。

顾名思义，version指定了Hello World项目当前的版本——1.0-SNAPSHOT。SNAPSHOT意为快照，说明该项目还处于开发中，是不稳定的版本。随着项目的发展，version会不断更新，如升级为1.0、1.1-SNAPSHOT、1.1、2.0等等。本书的6.5小节会详细介绍SNAPSHOT，第13章介绍如何使用Maven管理项目版本的升级发布。

最后一个name元素声明了一个对于用户更为友好的项目名称，虽然这不是必须的，但我还是推荐为每个POM声明name，以方便信息交流。

没有任何实际的Java代码，我们就能够定义一个Maven项目的POM，这体现了Maven的一大优点，它能让项目对象模型最大程度地与实际代码相独立，我们可以称之为解耦，或者正交性，这在很大程度上避免了Java代码和POM代码的相互影响。比如当项目需要升级版本时，只需要修改POM，而不需要更改Java代码；而在POM稳定之后，日常的Java代码开发工作基本不涉及POM的修改。

**3.2 编写主代码 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

项目主代码和测试代码不同，项目的主代码会被打包到最终的构件中（比如jar），而测试代码只在运行测试时用到，不会被打包。默认情况下，Maven假设项目主代码位于*src/main/java*目录，我们遵循Maven的约定，创建该目录，然后在该目录下创建文件com/juvenxu/mvnbook/helloworld/*HelloWorld.java*，其内容如代码清单3-2：

代码清单3-2：Hello World的主代码

1. <span style="font-size: small;">package com.juvenxu.mvnbook.helloworld;
3. public class HelloWorld
4. {
5. public String sayHello()
6. {
7. return "Hello Maven";
8. }
10. public static void main(String[] args)
11. {
12. System.out.print( new HelloWorld().sayHello() );
13. }
14. }
15. </span>

       这是一个简单的Java类，它有一个sayHello()方法，返回一个String。同时这个类还带有一个main方法，创建一个HelloWorld实例，调用sayHello()方法，并将结果输出到控制台。

关于该Java代码有两点需要注意。首先，在95%以上的情况下，我们应该把项目主代码放到*src/main/java/*目录下（遵循Maven的约定），而无须额外的配置，Maven会自动搜寻该目录找到项目主代码。其次，该Java类的包名是com.juvenxu.mvnbook.helloworld，这与我们之前在POM中定义的groupId和artifactId相吻合。一般来说，项目中Java类的包都应该基于项目的groupId和artifactId，这样更加清晰，更加符合逻辑，也方便搜索构件或者Java类。

代码编写完毕后，我们使用Maven进行编译，在项目根目录下运行命令 **mvn clean compile** ，我们会得到如下输出：

1. <span style="font-size: small;">[INFO] Scanning for projects...
2. [INFO] ------------------------------------------------------------------------
3. [INFO] Building Maven Hello World Project
4. [INFO]    task-segment: [clean, compile]
5. [INFO] ------------------------------------------------------------------------
6. [INFO] [clean:clean {execution: default-clean}]
7. [INFO] Deleting directory D:\code\hello-world\target
8. [INFO] [resources:resources {execution: default-resources}]
9. [INFO] skip non existing resourceDirectory D: \code\hello-world\src\main\resources
10. [INFO] [compiler:compile {execution: default-compile}]
11. [INFO] Compiling 1 source file to D: \code\hello-world\target\classes
12. [INFO] ------------------------------------------------------------------------
13. [INFO] BUILD SUCCESSFUL
14. [INFO] ------------------------------------------------------------------------
15. [INFO] Total time: 1 second
16. [INFO] Finished at: Fri Oct 09 02:08:09 CST 2009
17. [INFO] Final Memory: 9M/16M
18. [INFO] ------------------------------------------------------------------------
19. </span>

clean告诉Maven清理输出目录*target/*，compile告诉Maven编译项目主代码，从输出中我们看到Maven首先执行了clean:clean任务，删除*target/*目录，默认情况下Maven构建的所有输出都在*target/*目录中；接着执行resources:resources任务（未定义项目资源，暂且略过）；最后执行compiler:compile任务，将项目主代码编译至*target/classes*目录(编译好的类为*com/juvenxu/mvnbook/helloworld/HelloWorld.Class*）。

上文提到的clean:clean、resources:resources，以及compiler:compile对应了一些Maven插件及插件目标，比如clean:clean是clean插件的clean目标，compiler:compile是compiler插件的compile目标，后文会详细讲述Maven插件及其编写方法。

至此，Maven在没有任何额外的配置的情况下就执行了项目的清理和编译任务，接下来，我们编写一些单元测试代码并让Maven执行自动化测试。

**3.3 编写测试代码 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

为了使项目结构保持清晰，主代码与测试代码应该分别位于独立的目录中。3.2节讲过Maven项目中默认的主代码目录是*src/main/java*，对应地，Maven项目中默认的测试代码目录是*src/test/java*。因此，在编写测试用例之前，我们先创建该目录。

在Java世界中，由Kent Beck和Erich Gamma建立的JUnit是事实上的单元测试标准。要使用JUnit，我们首先需要为Hello World项目添加一个JUnit依赖，修改项目的POM如代码清单3-3：

代码清单3-3：为Hello World的POM添加依赖

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5. http://maven.apache.org/maven-v4\_0\_0.xsd">
6. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.juvenxu.mvnbook</groupId>
8. <artifactId>hello-world</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Hello World Project</name>
11. <dependencies>
12. <dependency>
13. <groupId>junit</groupId>
14. <artifactId>junit</artifactId>
15. <version>4.7</version>
16. <scope>test</scope>
17. </dependency>
18. </dependencies>
19. </project>

     代码中添加了dependencies元素，该元素下可以包含多个dependency元素以声明项目的依赖，这里我们添加了一个依赖——groupId是junit，artifactId是junit，version是4.7。前面我们提到groupId、artifactId和version是任何一个Maven项目最基本的坐标，JUnit也不例外，有了这段声明，Maven就能够自动下载junit-4.7.jar。也许你会问，Maven从哪里下载这个jar呢？在Maven之前，我们可以去JUnit的官网下载分发包。而现在有了Maven，它会自动访问中央仓库（<http://repo1.maven.org/maven2/>），下载需要的文件。读者也可以自己访问该仓库，打开路径junit/junit/4.7/，就能看到junit-4.7.pom和junit-4.7.jar。本书第6章会详细介绍Maven仓库及中央仓库。

上述POM代码中还有一个值为test的元素scope，scope为依赖范围，若依赖范围为test则表示该依赖只对测试有效，换句话说，测试代码中的import JUnit代码是没有问题的，但是如果我们在主代码中用import JUnit代码，就会造成编译错误。如果不声明依赖范围，那么默认值就是compile，表示该依赖对主代码和测试代码都有效。

配置了测试依赖，接着就可以编写测试类，回顾一下前面的HelloWorld类，现在我们要测试该类的sayHello()方法，检查其返回值是否为“Hello Maven”。在src/test/java目录下创建文件，其内容如代码清单3-4：

代码清单3-4：Hello World的测试代码

1. package com.juvenxu.mvnbook.helloworld;
2. import static org.junit.Assert.assertEquals;
3. import org.junit.Test;
5. public class HelloWorldTest
6. {
7. @Test
8. public void testSayHello()
9. {
10. HelloWorld helloWorld = new HelloWorld();
12. String result = helloWorld.sayHello();
14. assertEquals( "Hello Maven", result );
15. }
16. }

一个典型的单元测试包含三个步骤：一，准备测试类及数据；二，执行要测试的行为；三，检查结果。上述样例中，我们首先初始化了一个要测试的HelloWorld实例，接着执行该实例的sayHello()方法并保存结果到result变量中，最后使用JUnit框架的Assert类检查结果是否为我们期望的”Hello Maven”。在JUnit 3中，约定所有需要执行测试的方法都以test开头，这里我们使用了JUnit 4，但我们仍然遵循这一约定，在JUnit 4中，需要执行的测试方法都应该以@Test进行标注。

测试用例编写完毕之后就可以调用Maven执行测试，运行 **mvn clean test** ：

1. [INFO] Scanning for projects...
2. [INFO] ------------------------------------------------------------------------
3. [INFO] Building Maven Hello World Project
4. [INFO]    task-segment: [clean, test]
5. [INFO] ------------------------------------------------------------------------
6. [INFO] [clean:clean {execution: default-clean}]
7. [INFO] Deleting directory D:\git-juven\mvnbook\code\hello-world\target
8. [INFO] [resources:resources {execution: default-resources}]
9. …
10. Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.pom
11. 1K downloaded  (junit-4.7.pom)
12. [INFO] [compiler:compile {execution: default-compile}]
13. [INFO] Compiling 1 source file to D: \code\hello-world\target\classes
14. [INFO] [resources:testResources {execution: default-testResources}]
15. …
16. Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.jar
17. 226K downloaded  (junit-4.7.jar)
18. [INFO] [compiler:testCompile {execution: default-testCompile}]
19. [INFO] Compiling 1 source file to D:\ code\hello-world\target\test-classes
20. [INFO] ------------------------------------------------------------------------
21. [ERROR] BUILD FAILURE
22. [INFO] ------------------------------------------------------------------------
23. [INFO] Compilation failure
24. D:\code\hello-world\src\test\java\com\juvenxu\mvnbook\helloworld\HelloWorldTest.java:[8,5] -source 1.3 中不支持注释
25. （请使用 -source 5 或更高版本以启用注释）
26. @Test
27. [INFO] ------------------------------------------------------------------------
28. [INFO] For more information, run Maven with the -e switch
29. …

不幸的是构建失败了，不过我们先耐心分析一下这段输出（为了本书的简洁，一些不重要的信息我用省略号略去了）。命令行输入的是mvn clean test，而Maven实际执行的可不止这两个任务，还有clean:clean、resources:resources、compiler:compile、resources:testResources以及compiler:testCompile。暂时我们需要了解的是，在Maven执行测试（test）之前，它会先自动执行项目主资源处理，主代码编译，测试资源处理，测试代码编译等工作，这是Maven生命周期的一个特性，本书后续章节会详细解释Maven的生命周期。

从输出中我们还看到：Maven从中央仓库下载了junit-4.7.pom和junit-4.7.jar这两个文件到本地仓库（*~/.m2/repository*）中，供所有Maven项目使用。

构建在执行compiler:testCompile任务的时候失败了，Maven输出提示我们需要使用-source 5或更高版本以启动注释，也就是前面提到的JUnit 4的@Test注解。这是Maven初学者常常会遇到的一个问题。由于历史原因，Maven的核心插件之一compiler插件默认只支持编译Java 1.3，因此我们需要配置该插件使其支持Java 5，见代码清单3-5：

代码清单3-5：配置maven-compiler-plugin支持Java 5

1. <project>
2. …
3. <build>
4. <plugins>
5. <plugin>
6. <groupId>org.apache.maven.plugins</groupId>
7. <artifactId>maven-compiler-plugin</artifactId>
8. <configuration>
9. <source>1.5</source>
10. <target>1.5</target>
11. </configuration>
12. </plugin>
13. </plugins>
14. </build>
15. …
16. </project>

该POM省略了除插件配置以外的其他部分，我们暂且不去关心插件配置的细节，只需要知道compiler插件支持Java 5的编译。现在再执行**mvn clean test，**输出如下：

1. …
2. [INFO] [compiler:testCompile {execution: default-testCompile}]
3. [INFO] Compiling 1 source file to D: \code\hello-world\target\test-classes
4. [INFO] [surefire:test {execution: default-test}]
5. [INFO] Surefire report directory: D:\code\hello-world\target\surefire-reports
6. -------------------------------------------------------
7. T E S T S
8. -------------------------------------------------------
9. Running com.juvenxu.mvnbook.helloworld.HelloWorldTest
10. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.055 sec
11. Results :
12. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
13. [INFO] ------------------------------------------------------------------------
14. [INFO] BUILD SUCCESSFUL
15. [INFO] ------------------------------------------------------------------------
16. …

我们看到compiler:testCompile任务执行成功了，测试代码通过编译之后在*target/test-classes*下生成了二进制文件，紧接着surefire:test任务运行测试，surefire是Maven世界中负责执行测试的插件，这里它运行测试用例HelloWorldTest，并且输出测试报告，显示一共运行了多少测试，失败了多少，出错了多少，跳过了多少。显然，我们的测试通过了——BUILD SUCCESSFUL。

**3.4 打包和运行 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

将项目进行编译、测试之后，下一个重要步骤就是打包（package）。Hello World的POM中没有指定打包类型，使用默认打包类型jar，我们可以简单地执行命令 **mvn clean package** 进行打包，可以看到如下输出：

1. …
2. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
4. [INFO] [jar:jar {execution: default-jar}]
5. [INFO] Building jar: D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
6. [INFO]
7. --------------------------------------------------------------------
8. [INFO] BUILD SUCCESSFUL
9. …

类似地，Maven会在打包之前执行编译、测试等操作。这里我们看到jar:jar任务负责打包，实际上就是jar插件的jar目标将项目主代码打包成一个名为*hello-world-1.0-SNAPSHOT.jar*的文件，该文件也位于*target/*输出目录中，它是根据artifact-version.jar规则进行命名的，如有需要，我们还可以使用*finalName*来自定义该文件的名称，这里暂且不展开，本书后面会详细解释。

至此，我们得到了项目的输出，如果有需要的话，就可以复制这个jar文件到其他项目的Classpath中从而使用HelloWorld类。但是，如何才能让其他的Maven项目直接引用这个jar呢？我们还需要一个安装的步骤，执行 **mvn clean install**：

1. …
2. [INFO] [jar:jar {execution: default-jar}]
3. [INFO] Building jar: D: \code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
4. [INFO] [install:install {execution: default-install}]
5. [INFO] Installing D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar to C:\Users\juven\.m2\repository\com\juvenxu\mvnbook\hello-world\1.0-SNAPSHOT\hello-world-1.0-SNAPSHOT.jar
6. [INFO]
7. ------------------------------------------------------------------------
8. [INFO] BUILD SUCCESSFUL
9. …

在打包之后，我们又执行了安装任务install:install，从输出我们看到该任务将项目输出的jar安装到了Maven本地仓库中，我们可以打开相应的文件夹看到Hello World项目的pom和jar。之前讲述JUnit的POM及jar的下载的时候，我们说只有构件被下载到本地仓库后，才能由所有Maven项目使用，这里是同样的道理，只有将Hello World的构件安装到本地仓库之后，其他Maven项目才能使用它。

我们已经将体验了Maven最主要的命令：**mvn clean compile、mvn clean test、mvn clean package、mvn clean install**。执行test之前是会先执行compile的，执行package之前是会先执行test的，而类似地，install之前会执行package。我们可以在任何一个Maven项目中执行这些命令，而且我们已经清楚它们是用来做什么的。

到目前为止，我们还没有运行Hello World项目，不要忘了HelloWorld类可是有一个main方法的。默认打包生成的jar是不能够直接运行的，因为带有main方法的类信息不会添加到manifest中(我们可以打开jar文件中的*META-INF/MANIFEST.MF*文件，将无法看到Main-Class一行)。为了生成可执行的jar文件，我们需要借助maven-shade-plugin，配置该插件如下：

1. <plugin>
2. <groupId>org.apache.maven.plugins</groupId>
3. <artifactId>maven-shade-plugin</artifactId>
4. <version>1.2.1</version>
5. <executions>
6. <execution>
7. <phase>package</phase>
8. <goals>
9. <goal>shade</goal>
10. </goals>
11. <configuration>
12. <transformers>
13. <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">            <mainClass>com.juvenxu.mvnbook.helloworld.HelloWorld</mainClass>
14. </transformer>
15. </transformers>
16. </configuration>
17. </execution>
18. </executions>
19. </plugin>

plugin元素在POM中的相对位置应该在<project><build><plugins>下面。我们配置了mainClass为com.juvenxu.mvnbook.helloworld.HelloWorld，项目在打包时会将该信息放到MANIFEST中。现在执行 **mvn clean install** ，待构建完成之后打开target/目录，我们可以看到*hello-world-1.0-SNAPSHOT.jar*和*original-hello-world-1.0-SNAPSHOT.jar*，前者是带有Main-Class信息的可运行jar，后者是原始的jar，打开*hello-world-1.0-SNAPSHOT.jar*的*META-INF/MANIFEST.MF*，可以看到它包含这样一行信息：

Main-Class: com.juvenxu.mvnbook.helloworld.HelloWorld

现在，我们在项目根目录中执行该jar文件：

D: \code\hello-world>java -jar target\hello-world-1.0-SNAPSHOT.jar

Hello Maven

控制台输出为Hello Maven，这正是我们所期望的。

本小节介绍了Hello World项目，侧重点是Maven而非Java代码本身，介绍了POM、Maven项目结构、以及如何编译、测试、打包，等等。

**3.5 使用Archetype生成项目骨架 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Hello World项目中有一些Maven的约定：在项目的根目录中放置pom.xml，在src/main/java目录中放置项目的主代码，在src/test/java中放置项目的测试代码。我之所以一步一步地展示这些步骤，是为了能让可能是Maven初学者的你得到最实际的感受。我们称这些基本的目录结构和pom.xml文件内容称为项目的骨架，当你第一次创建项目骨架的时候，你还会饶有兴趣地去体会这些默认约定背后的思想，第二次，第三次，你也许还会满意自己的熟练程度，但第四、第五次做同样的事情，就会让程序员恼火了，为此Maven提供了Archetype以帮助我们快速勾勒出项目骨架。

还是以Hello World为例，我们使用maven archetype来创建该项目的骨架，离开当前的Maven项目目录。

如果是Maven 3，简单的运行：

mvn archetype:generate

如果是Maven 2，最好运行如下命令：

mvn org.apache.maven.plugins:maven-archetype-plugin:2.0-alpha-5:generate

很多资料会让你直接使用更为简单的 mvn archetype:generate 命令，但在Maven2中这是不安全的，因为该命令没有指定archetype插件的版本，于是Maven会自动去下载最新的版本，进而可能得到不稳定的SNAPSHOT版本，导致运行失败。然而在Maven 3中，即使用户没有指定版本，Maven也只会解析最新的稳定版本，因此这是安全的，具体内容见7.7小节。

我们实际上是在运行插件maven-archetype-plugin，注意冒号的分隔，其格式为 *groupId:artifactId:version:goal* ，org.apache.maven.plugins 是maven官方插件的groupId，maven-archetype-plugin 是archetype插件的artifactId，2.0-alpha-5 是目前该插件最新的稳定版，generate是我们要使用的插件目标。

紧接着我们会看到一段长长的输出，有很多可用的archetype供我们选择，包括著名的Appfuse项目的archetype，JPA项目的archetype等等。每一个archetype前面都会对应有一个编号，同时命令行会提示一个默认的编号，其对应的archetype为maven-archetype-quickstart，我们直接回车以选择该archetype，紧接着Maven会提示我们输入要创建项目的groupId、artifactId、 version、以及包名package，如下输入并确认：

1. Define value for groupId: : com.juvenxu.mvnbook
2. Define value for artifactId: : hello-world
3. Define value for version:  1.0-SNAPSHOT: :
4. Define value for package:  com.juvenxu.mvnbook: : com.juvenxu.mvnbook.helloworld
5. Confirm properties configuration:
6. groupId: com.juvenxu.mvnbook
7. artifactId: hello-world
8. version: 1.0-SNAPSHOT
9. package: com.juvenxu.mvnbook.helloworld
10. Y: : Y

Archetype插件将根据我们提供的信息创建项目骨架。在当前目录下，Archetype插件会创建一个名为hello-world（我们定义的artifactId）的子目录，从中可以看到项目的基本结构：基本的*pom.xml*已经被创建，里面包含了必要的信息以及一个junit依赖；主代码目录*src/main/java*已经被创建，在该目录下还有一个Java类*com.juvenxu.mvnbook.helloworld.App*，注意这里使用到了我们刚才定义的包名，而这个类也仅仅只有一个简单的输出Hello World!的main方法；测试代码目录*src/test/java*也被创建好了，并且包含了一个测试用例*com.juvenxu.mvnbook.helloworld.AppTest*。

Archetype可以帮助我们迅速地构建起项目的骨架，在前面的例子中，我们完全可以在Archetype生成的骨架的基础上开发Hello World项目以节省我们大量时间。

此外，我们这里仅仅是看到了一个最简单的archetype，如果你有很多项目拥有类似的自定义项目结构以及配置文件，你完全可以一劳永逸地开发自己的archetype，然后在这些项目中使用自定义的archetype来快速生成项目骨架，本书后面的章节会详细阐述如何开发Maven Archetype。

**3.6 m2eclipse简单使用 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

介绍前面Hello World项目的时候，我们并没有涉及IDE，如此简单的一个项目，使用最简单的编辑器也能很快完成，但对于稍微大一些的项目来说，没有IDE就是不可想象的，本节我们先介绍m2eclipse的基本使用。

**3.6.1 导入Maven项目**

第2章介绍了如何安装m2eclipse，现在，我们使用m2ecilpse导入Hello World项目。选择菜单项**File**，然后选择**Import**，我们会看到一个Import对话框，在该对话框中选择General目录下的**Maven Projects**，然后点击**Next**，就会出现**Import Projects**对话框，在该对话框中点击**Browse…**选择Hello World的根目录（即包含pom.xml文件的那个目录），这时对话框中的**Projects:**部分就会显示该目录包含的Maven项目，如图3-1所示：

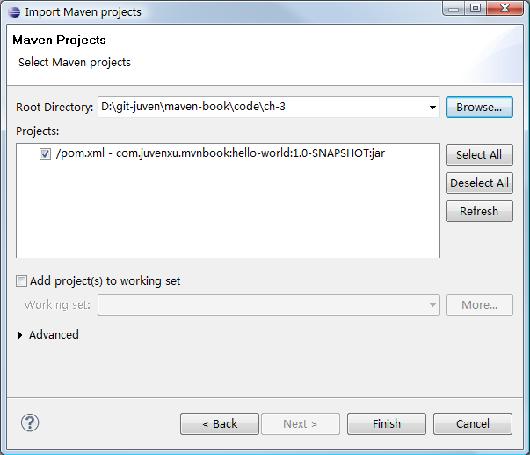


 图3-1 在Eclipse中导入Maven项目

点击Finish之后，m2ecilpse就会将该项目导入到当前的workspace中，导入完成之后，我们就可以在Package Explorer视图中看到如图3-2所示的项目结构：

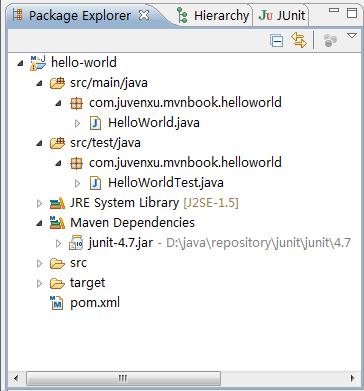


 图3-2 Eclipse中导入的Maven项目结构

我们看到主代码目录*src/main/java*和测试代码目录*src/test/java*成了Eclipse中的资源目录，包和类的结构也十分清晰，当然pom.xml永远在项目的根目录下，而从这个视图中我们甚至还能看到项目的依赖junit-4.7.jar，其实际的位置指向了Maven本地仓库（这里我自定义了Maven本地仓库地址为*D:\java\repository*，后续章节会介绍如何自定义本地仓库位置）。

**3.6.2 创建Maven项目**

创建一个Maven项目也十分简单，选择菜单项**File -> New -> Other**，在弹出的对话框中选择Maven下的**Maven Project**，然后点击**Next >**，在弹出的**New Maven Project**对话框中，我们使用默认的选项（不要选择Create a simple project选项，那样我们就能使用Maven Archetype），点击**Next >**，此时m2eclipse会提示我们选择一个Archetype，我们选择**maven-archetype-quickstart**，再点击**Next >**。由于m2eclipse实际上是在使用maven-archetype-plugin插件创建项目，因此这个步骤与上一节我们使用archetype创建项目骨架类似，输入groupId,、artifactId、version、package（暂时我们不考虑Properties），如图3-3所示：

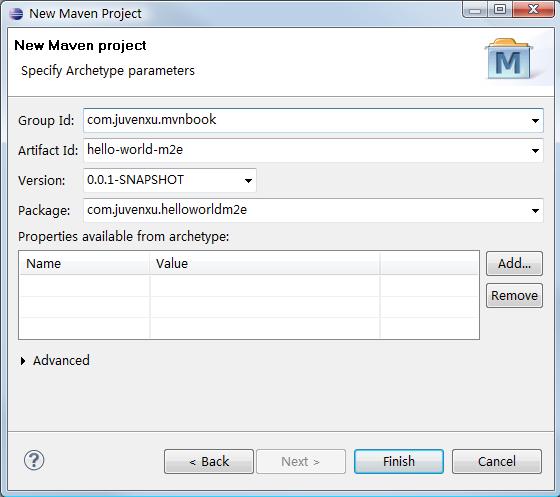


 图3-3 在Eclipse中使用Archetype创建项目

注意，为了不和前面已导入的Hello World项目产生冲突和混淆，我们使用不同的artifactId和package。OK，点击Finish，Maven项目就创建完成了，其结构与前一个已导入的Hello World项目基本一致。

**3.6.3 运行mvn命令**

我们需要在命令行输入如mvn clean install之类的命令来执行maven构建，m2eclipse中也有对应的功能，在Maven项目或者pom.xml上右击，再选择Run As，就能看到如下的常见Maven命令，如图3-4所示：

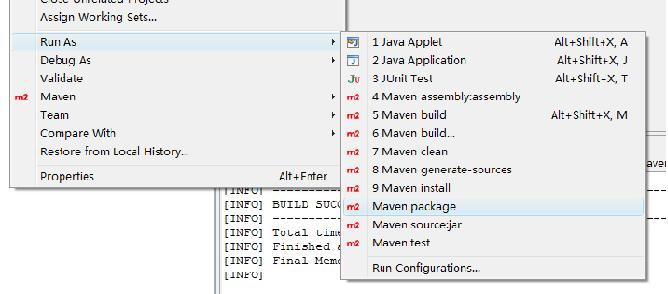


 图3-4 在Eclipse中运行默认mvn命令

选择想要执行的Maven命令就能执行相应的构建，同时我们也能在Eclipse的console中看到构建输出。这里常见的一个问题是，默认选项中没有我们想要执行的Maven命令怎么办？比如，默认带有mvn test，但我们想执行mvn clean test，很简单，选择**Maven buid…** 以自定义Maven运行命令，在弹出对话框中的**Goals**一项中输入我们想要执行的命令，如clean test，设置一下Name，点击**Run**即可。并且，下一次我们选择**Maven build**，或者使用快捷键Alt + Shift + X, M快速执行Maven构建的时候，上次的配置直接就能在历史记录中找到。图3-5就是自定义Maven运行命令的界面：

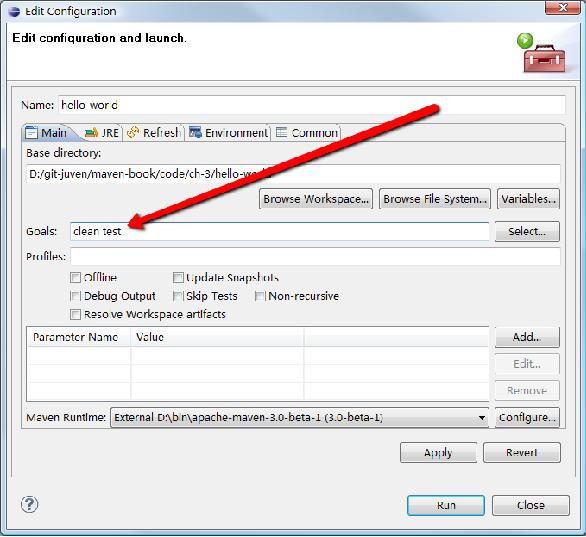


 图3-5 在Eclipse中自定义mvn命令

**3.7 NetBeans Maven插件简单使用 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

NetBeans的Maven插件也十分简单易用，我们可以轻松地在NetBeans中导入现有的Maven项目，或者使用Archetype创建Maven项目，我们也能够在NetBeans中直接运行mvn命令。

**3.7.1 打开Maven项目**

与其说***打开***Maven项目，不如称之为***导入***更为合适，因为这个项目不需要是NetBeans创建的Maven项目，不过这里我们还是遵照NetBeans菜单中使用的名称。

选择菜单栏中的**文件**，然后选择**打开项目**，直接定位到Hello World项目的根目录，NetBeans会十分智能地识别出Maven项目，如图3-6所示：

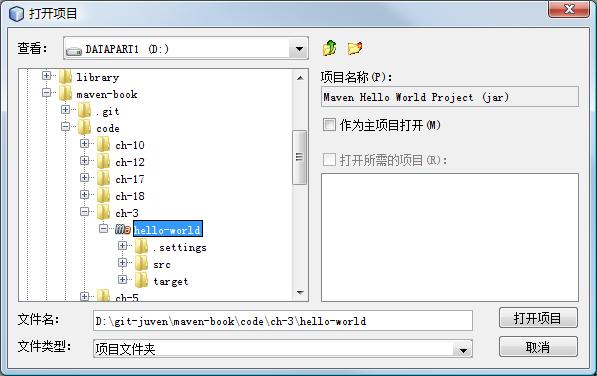


图3-6 在NetBeans中导入Maven项目

Maven项目的图标有别于一般的文件夹，点击**打开项目**后，Hello World项目就会被导入到NetBeans中，在**项目**视图中可以看到如图3-7所示的项目结构：

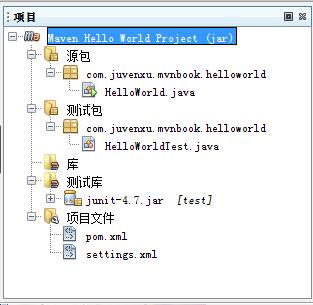


 图3-7 NetBeans中导入的Maven项目结构

NetBeans中项目主代码目录的名称为**源包**，测试代码目录成了**测试包**，编译范围依赖为**库**，测试范围依赖为**测试库**，这里我们也能看到pom.xml，NetBeans甚至还帮我们引用了settings.xml。

**3.7.2 创建Maven项目**

在NetBeans中创建Maven项目同样十分轻松，在菜单栏中选择**文件**，然后**新建项目**，在弹出的对话框中，选择项目类别为**Maven**，项目为**Maven项目**，点击“下一步”之后，对话框会提示我们选择Maven原型（即Maven Archtype），我们选择**Maven快速启动原型（1.0）**，（即前文提到的maven-archetype-quickstart），点击“下一步”之后，输入项目的基本信息，这些信息在之前讨论archetype及在m2eclipse中创建Maven项目的时候都仔细解释过，不再详述，如图3-8所示：

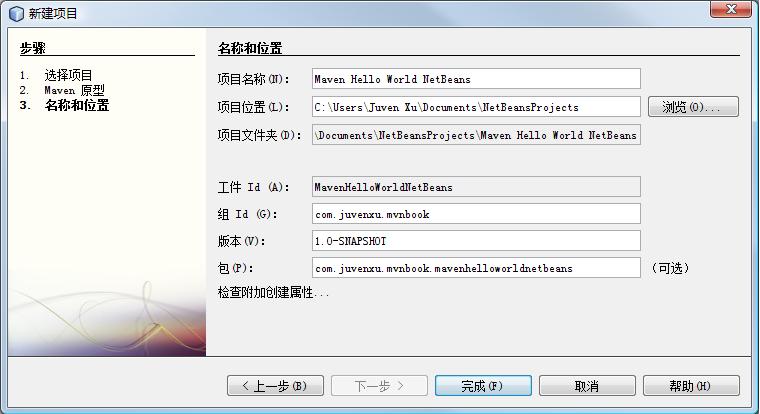


 图3-8 在NetBeans中使用Archetype创建Maven项目

点击完成之后，一个新的Maven项目就创建好了。

**3.7.3 运行mvn命令**

NetBeans在默认情况下提供两种Maven运行方式，点击菜单栏中的**运行**，我们可以看到**生成项目**和**清理并生成项目**两个选项，我们可以尝试“点击运行Maven构建”，根据NetBeans控制台的输出，我们就能发现它们实际上对应了**mvn install**和**mvn clean install**两个命令。

在实际开发过程中，我们往往不会满足于这两种简单的方式，比如，有时候我们只想执行项目的测试，而不需要打包，这时我们就希望能够执行**mvn clean test**命令，所幸的是NetBeans Maven插件完全支持自定义的mvn命令配置。

在菜单栏中选择**工具**，接着选择**选项**，在对话框中，最上面一栏选择**其他**，下面选择**Maven**标签栏，在这里我们可以对NetBeans Maven插件进行全局的配置（还记得第2章中我们如何配置NetBeans使用外部Maven么？）。现在，选择倒数第三行的**编辑全局定制目标定义…**，我们添加一个名为**Maven Test**的操作，执行目标为clean test，暂时不考虑其他配置选项，如图3-9所示：

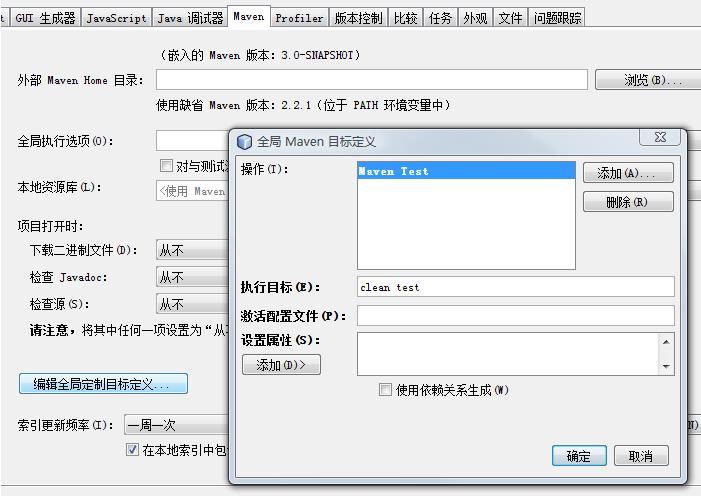


图3-9 在NetBeans中自定义mvn命令

点击“缺省保存该配置”，在Maven项目上右击，选择**定制**，就能看到刚才配置好的Maven运行操作，选择**Maven Test**之后，终端将执行mvn clean test。值得一提的是，我们也可以在项目上右击，选择**定制**，再选择**目标…**再输入想要执行的Maven目标（如clean package)，点击确定之后NetBeans就会执行相应的Maven命令。这种方式十分便捷，但这是临时的，该配置不会被保存，也不会有历史记录。

**3.8 小结 [Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章以尽可能简单且详细的方式叙述了一个Hello World项目，重点解释了POM的基本内容、Maven项目的基本结构、以及构建项目基本的Maven命令。在此基础上，还介绍了如何使用 Archetype快速创建项目骨架。最后讲述的是如何在Eclipse和NetBeans中导入、创建及构建Maven项目。

**第1章 Maven简介** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

1.1  何为Maven/2

1.2  为什么需要Maven/4

1.3  Maven与极限编程/7  
1.4  被误解的Maven/8  
1.5  小结/9

**1.1 何为Maven** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven这个词可以翻译为“知识的积累”，也可以翻译为“专 家”或“内行”。本书将介绍Maven这一跨平台的项目管理工具。作为Apache组织中的一个颇为成功的开源项目，Maven主要服务于基于Java平 台的项目构建、依赖管理和项目信息管理。无论是小型的开源类库项目，还是大型的企业级应用；无论是传统的瀑布式开发，还是流行的敏捷模式，Maven都能 大显身手。

**1.1.1  何为构建**

不管你是否意识到，构建（build）是每一位程序员每天都在做的工作。早上来 到公司，我们做的第一件事情就是从源码库签出最新的源码，然后进行单元测试，如果发现失败的测试，会找相关的同事一起调试，修复错误代码。接着回到自己的 工作上来，编写自己的单元测试及产品代码，我们会感激IDE随时报出的编译错误提示。  
忙到午饭时间，代码编写得差不多了，测试也通过了，开心地享 用午餐，然后休息。下午先在昏昏沉沉中开了个例会，会议结束后喝杯咖啡继续工作。刚才在会上经理要求看测试报告，于是找了相关工具集成进IDE，生成了像 模像样的测试覆盖率报告，接着发了一封电子邮件给经理，松了口气。谁料QA小组又发过来了几个bug，没办法，先本地重现再说，于是熟练地用IDE生成了 一个WAR包，部署到Web容器下，启动容器。看到熟悉的界面了，遵循bug报告，一步步重现了bug……快下班的时候，bug修好了，提交代码，通知 QA小组，在愉快中结束了一天的工作。

仔细总结一下，我们会发现，除了编写源代码，我们每天有相当一部分时间花在了编 译、运行单元测试、生成文档、打包和部署等烦琐且不起眼的工作上，这就是构建。如果我们现在还手工这样做，那成本也太高了，于是有人用软件的方法让这一系 列工作完全自动化，使得软件的构建可以像全自动流水线一样，只需要一条简单的命令，所有烦琐的步骤都能够自动完成，很快就能得到最终结果。

**1.1.2  Maven是优秀的构建工具**

前面介绍了Maven的用途之一是服务于构建，它是一个异常强大的构建工具，能 够帮我们自动化构建过程，从清理、编译、测试到生成报告，再到打包和部署。我们不需要也不应该一遍又一遍地输入命令，一次又一次地点击鼠标，我们要做的是 使用Maven配置好项目，然后输入简单的命令(如mvn clean install)，Maven会帮我们处理那些烦琐的任务。

Maven是跨平台的，这意味着无论是在Windows上，还是在Linux或者Mac上，都可以使用同样的命令。

我们一直在不停地寻找避免重复的方法。设计的重复、编码的重复、文档的重复，当 然还有构建的重复。Maven最大化地消除了构建的重复，抽象了构建生命周期，并且为绝大部分的构建任务提供了已实现的插件，我们不再需要定义过程，甚至 不需要再去实现这些过程中的一些任务。最简单的例子是测试，我们没必要告诉Maven去测试，更不需要告诉Maven如何运行测试，只需要遵循Maven 的约定编写好测试用例，当我们运行构建的时候，这些测试便会自动运行。

想象一下，Maven抽象了一个完整的构建生命周期模型，这个模型吸取了大量其 他的构建脚本和构建工具的优点，总结了大量项目的实际需求。如果遵循这个模型，可以避免很多不必要的错误，可以直接使用大量成熟的Maven插件来完成我 们的任务（很多时候我们可能都不知道自己在使用Maven插件）。此外，如果有非常特殊的需求，我们也可以轻松实现自己的插件。

Maven还有一个优点，它能帮助我们标准化构建过程。在Maven之前，十个项目可能有十种构建方式；有了Maven之后，所有项目的构建命令都是简单一致的，这极大地避免了不必要的学习成本，而且有利于促进项目团队的标准化。

综上所述，Maven作为一个构建工具，不仅能帮我们自动化构建，还能够抽象构建过程，提供构建任务实现；它跨平台，对外提供了一致的操作接口，这一切足以使它成为优秀的、流行的构建工具。

**1.1.3  Maven不仅仅是构建工具**

Java不仅是一门编程语言，还是一个平台，通过JRuby和Jython，我 们可以在Java平台上编写和运行Ruby和Python程序。我们也应该认识到，Maven不仅是构建工具，还是一个依赖管理工具和项目信息管理工具。 它提供了中央仓库，能帮我们自动下载构件。

在这个开源的年代里，几乎任何Java应用都会借用一些第三方的开源类库，这些 类库都可通过依赖的方式引入到项目中来。随着依赖的增多，版本不一致、版本冲突、依赖臃肿等问题都会接踵而来。手工解决这些问题是十分枯燥的，幸运的是 Maven提供了一个优秀的解决方案，它通过一个坐标系统准确地定位每一个构件（artifact），也就是通过一组坐标Maven能够找到任何一个 Java类库（如jar文件）。Maven给这个类库世界引入了经纬，让它们变得有秩序，于是我们可以借助它来有序地管理依赖，轻松地解决那些繁杂的依赖 问题。

Maven还能帮助我们管理原本分散在项目中各个角落的项目信息，包括项目描 述、开发者列表、版本控制系统地址、许可证、缺陷管理系统地址等。这些微小的变化看起来很琐碎，并不起眼，但却在不知不觉中为我们节省了大量寻找信息的时 间。除了直接的项目信息，通过Maven自动生成的站点，以及一些已有的插件，我们还能够轻松获得项目文档、测试报告、静态分析报告、源码版本日志报告等 非常具有价值的项目信息。

Maven还为全世界的Java开发者提供了一个免费的中央仓库，在其中几乎可以找到任何的流行开源类库。通过一些Maven的衍生工具（如Nexus），我们还能对其进行快速地搜索。只要定位了坐标，Maven就能够帮我们自动下载，省去了手工劳动。

使用Maven还能享受一个额外的好处，即Maven对于项目目录结构、测试用 例命名方式等内容都有既定的规则，只要遵循了这些成熟的规则，用户在项目间切换的时候就免去了额外的学习成本，可以说是约定优于配置 （Convention Over Configuration）。

**1.2 为什么需要Maven** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven不是Java领域唯一的构建管理的解决方案。本节将通过一些简单的例子解释Maven的必要性，并介绍其他构建解决方案，如IDE、Make和Ant，并将它们与Maven进行比较。

**1.2.1  组装PC和品牌PC**

笔者初中时开始接触计算机，到了高中时更是梦寐以求希望拥有一台自己的计算机。我的第一台计算机是赛扬733的，选购是一个漫长的过程，我先阅读了大量的杂志以了解各类配件的优劣，CPU、内存、主板、显卡，甚至声卡，我都仔细地挑选，后来还跑了很多商家，调货、讨价还价，组装好后自己装操作系统和驱动程序……虽然这花费了我大量时间，但我很享受这个过程。可是事实证明，装出来的机器稳定性不怎么好。

一年前我需要配一台工作站，这时候我已经没有太多时间去研究电脑配件了。我选择了某知名PC供应商的在线商店，大概浏览了一下主流的机型，选择了我需要的配置，然后下单、付款。接着PC供应商帮我组装电脑、安装操作系统和驱动程序。一周后，物流公司将电脑送到我的家里，我接上显示器、电源、鼠标和键盘就能直接使用了。这为我节省了大量时间，而且这台电脑十分稳定，商家在把电脑发送给我之前已经进行了很好的测试。对了，我还能享受两年的售后服务。

使用脚本建立高度自定义的构建系统就像买组装PC，耗时费力，结果也不一定很好。当然，你可以享受从无到有的乐趣，但恐怕实际项目中无法给你那么多时间。使用Maven就像购买品牌PC，省时省力，并能得到成熟的构建系统，还能得到来自于Maven社区的大量支持。唯一与购买品牌PC不同的是，Maven是开源的，你无须为此付费。如果有兴趣，你还能去了解Maven是如何工作的，而我们无法知道那些PC巨头的商业秘密。

**1.2.2  IDE不是万能的**

当然，我们无法否认优秀的IDE能大大提高开发效率。当前主流的IDE如Eclipse和NetBeans等都提供了强大的文本编辑、调试甚至重构功能。虽然使用简单的文本编辑器和命令行也能完成绝大部分开发工作，但很少有人愿意那样做。然而，IDE是有其天生缺陷的：

* IDE依赖大量的手工操作。编译、测试、代码生成等工作都是相互独立的，很难一键完成所有工作。手工劳动往往意味着低效，意味着容易出错。
* 很难在项目中统一所有的IDE配置，每个人都有自己的喜好。也正是由于这个原因，一个在机器A上可以成功运行的任务，到了机器B的IDE中可能就会失败。

我们应该合理利用IDE，而不是过多地依赖它。对于构建这样的任务，在IDE中一次次地点击鼠标是愚蠢的行为。Maven是这方面的专家，而且主流IDE都集成了Maven，我们可以在IDE中方便地运行Maven执行构建。

**1.2.3  Make**

Make也许是最早的构建工具，它由Stuart Feldman于1977年在Bell实验室创建。Stuart Feldman也因此于2003年获得了ACM国际计算机组织颁发的软件系统奖。目前Make有很多衍生实现，包括最流行的GNU Make和BSD Make，还有Windows平台的Microsoft nmake等。

Make由一个名为Makefile的脚本文件驱动，该文件使用Make自己定义的语法格式。其基本组成部分为一系列规则（Rules），而每一条规则又包括目标（Target）、依赖（Prerequisite）和命令（Command）。Makefile的基本结构如下：

1. <span style="font-size: small;">TARGET… : PREREQUISITE…
2. COMMAND
3. …
4. …
5. </span>

     Make通过一系列目标和依赖将整个构建过程串联起来，同时利用本地命令完成每个目标的实际行为。Make的强大之处在于它可以利用所有系统的本地命令，尤其是UNIX/Linux系统，丰富的功能、强大的命令能够帮助Make快速高效地完成任务。

但是，Make将自己和操作系统绑定在一起了。也就是说，使用Make，就不能实现（至少很难）跨平台的构建，这对于Java来说是非常不友好的。此外，Makefile的语法也成问题，很多人抱怨Make构建失败的原因往往是一个难以发现的空格或Tab使用错误。

**1.2.4  Ant**

Ant不是指蚂蚁，而是意指“另一个整洁的工具”（Another Neat Tool），它最早用来构建著名的Tomcat，其作者James Duncan Davidson创作它的动机就是因为受不了Makefile的语法格式。我们可以将Ant看成是一个Java版本的Make，也正因为使用了Java，Ant是跨平台的。此外，Ant使用XML定义构建脚本，相对于Makefile来说，这也更加友好。

与Make类似，Ant有一个构建脚本build.xml，如下所示：

<?xml version="1.0"?>

<project name="Hello" default="compile">

<target name="compile" description="compile the Java source code to class files">

<mkdir dir="classes"/>

<javac srcdir="." destdir="classes"/>

</target>

<target name="jar" depends="compile" description="create a Jar file ">

<jar destfile="hello.jar">

<fileset dir="classes" includes="\*\*/\*.class"/>

<manifest>

<attribute name="Main.Class" value="HelloProgram"/>

</manifest>

</jar>

</target>

</project>

build.xml的基本结构也是目标（target）、依赖（depends），以及实现目标的任务。比如在上面的脚本中，jar目标用来创建应用程序jar文件，该目标依赖于compile目标，后者执行的任务是创建一个名为classes的文件夹，编译当前目录的java文件至classes目录。compile目标完成后，jar目标再执行自己的任务。Ant有大量内置的用Java实现的任务，这保证了其跨平台的特质，同时，Ant也有特殊的任务exec来执行本地命令。

和Make一样，Ant也都是过程式的，开发者显式地指定每一个目标，以及完成该目标所需要执行的任务。针对每一个项目，开发者都需要重新编写这一过程，这里其实隐含着很大的重复。Maven是声明式的，项目构建过程和过程各个阶段所需的工作都由插件实现，并且大部分插件都是现成的，开发者只需要声明项目的基本元素，Maven就执行内置的、完整的构建过程。这在很大程度上消除了重复。

Ant是没有依赖管理的，所以很长一段时间Ant用户都不得不手工管理依赖，这是一个令人头疼的问题。幸运的是，Ant用户现在可以借助Ivy管理依赖。而对于Maven用户来说，依赖管理是理所当然的，Maven不仅内置了依赖管理，更有一个可能拥有全世界最多Java开源软件包的中央仓库，Maven用户无须进行任何配置就可以直接享用。

**1.2.5  不重复发明轮子**

**【该小节内容整理自网友Arthas最早在Maven中文MSN的群内的讨论，在此表示感谢】**

小张是一家小型民营软件公司的程序员，他所在的公司要开发一个新的Web项目。经过协商，决定使用Spring、iBatis和Tapstry。jar包去哪里找呢？公司里估计没有人能把Spring、iBatis和Tapstry所使用的jar包一个不少地找出来。大家的做法是，先到Spring的站点上去找一个spring.with.dependencies，然后去iBatis的网站上把所有列出来的jar包下载下来，对Tapstry、Apache commons等执行同样的操作。项目还没有开始，WEB.INF/lib下已经有近百个jar包了，带版本号的、不带版本号的、有用的、没用的、相冲突的，怎一个“乱”字了得！

在项目开发过程中，小张不时地发现版本错误和版本冲突问题，他只能硬着头皮逐一解决。项目开发到一半，经理发现最终部署的应用的体积实在太大了，要求小张去掉一些没用的jar包，于是小张只能加班加点地一个个删……

小张隐隐地觉得这些依赖需要一个框架或者系统来进行管理。

小张喜欢学习流行的技术，前几年Ant十分流行，他学了，并成为了公司这方面的专家。小张知道，Ant打包，无非就是创建目录，复制文件，编译源代码，使用一堆任务，如copydir、fileset、classpath、ref、target，然后再jar、zip、war，打包就成功了。

项目经理发话了：“兄弟们，新项目来了，小张，你来写Ant脚本！”

“是，保证完成任务！”接着，小张继续创建一个新的XML文件。target clean; target compile; target jar; …… 不知道他是否想过，在他写的这么多的Ant脚本中，有多少是重复劳动，有多少代码会在一个又一个项目中重现。既然都差不多，有些甚至完全相同，为什么每次都要重新编写？

终于有一天，小张意识到了这个问题，想复用Ant脚本，于是在开会时他说：“以后就都用我这个规范的Ant脚本吧，新的项目只要遵循我定义的目录结构就可以了。”经理听后觉得很有道理：“嗯，确实是个进步。”

这时新来的研究生发言了：“经理，用Maven吧，这个在开源社区很流行，比Ant更方便。”小张一听很惊讶，Maven真比自己的“规范化Ant”强大？其实他不知道自己只是在重新发明轮子，Maven已经有一大把现成的插件，全世界都在用，你自己不用写任何代码！

为什么没有人说“我自己写的代码最灵活，所以我不用Spring，我自己实现IoC；我不用Hibernate，我自己封装JDBC”？

**1.3 Maven与极限编程** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

极限编程（XP）是近些年在软件行业红得发紫的敏捷开发方法，它强调拥抱变化。该软件开发方法的创始人Kent Beck提出了XP所追求的价值、实施原则和推荐实践。下面看一下Maven是如何适应XP的。  
首先看一下Maven如何帮助XP团队实现一些核心价值：

* 简单。Maven暴露了一组一致、简洁的操作接口，能帮助团队成员从原来的高度自定义的、复杂的构建系统中解脱出来，使用Maven现有的成熟的、稳定的组件也能简化构建系统的复杂度。
* 交流与反馈。与版本控制系统结合后，所有人都能执行最新的构建并快速得到反馈。此外，自动生成的项目报告也能帮助成员了解项目的状态，促进团队的交流。

此外，Maven更能无缝地支持或者融入到一些主要的XP实践中：

* 测试驱动开发（TDD）。TDD强调测试先行，所有产品都应该由测试用例覆盖。而测试是Maven生命周期的最重要的组成部分之一，并且Maven有现成的成熟插件支持业界流行的测试框架，如JUnit和TestNG。
* 十分钟构建。十分钟构建强调我们能够随时快速地从源码构建出最终的产品。这正是Maven所擅长的，只需要一些配置，之后用一条简单的命令就能让Maven帮你清理、编译、测试、打包、部署，然后得到最终的产品。
* 持续集成（CI）。CI强调项目以很短的周期（如15分钟）集成最新的代码。 实际上,CI的前提是源码管理系统和构建系统。目前业界流行的CI服务器如Hudson和CruiseControl都能很好地和Maven进行集成。也 就是说，使用Maven后，持续集成会变得更加方便。
* 富有信息的工作区。 这条实践强调开发者能够快速方便地了解到项目的最新状态。当然，Maven并不会帮你把测试覆盖率报告贴到墙上，也不会在你的工作台上放个鸭子告诉你构建 失败了。不过使用Maven发布的项目报告站点，并配置你需要的项目报告，如测试覆盖率报告，都能帮你把信息推送到开发者眼前。

上述这些实践并非只在XP中适用。事实上，除了其他敏捷开发方法如SCRUM之外，几乎任何软件开发方法都能借鉴这些实践。也就是说，Maven几乎能够很好地支持任何软件开发方法。  
例 如，在传统的瀑布模型开发中，项目依次要经历需求开发、分析、设计、编码、测试和集成发布阶段。从设计和编码阶段开始，就可以使用Maven来建立项目的 构建系统。在设计阶段，也完全可以针对设计开发测试用例，然后再编写代码来满足这些测试用例。然而，有了自动化构建系统，我们可以节省很多手动的测试时 间。此外，尽早地使用构建系统集成团队的代码，对项目也是百利而无一害。最后，Maven还能帮助我们快速地发布项目。

**第2章 Maven的安装和配置** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

第1章介绍了Maven是什么，以及为什么要使用Maven， 我们将从本章实际开始实际接触Maven。本章首先将介绍如何在主流的操作系统下安装Maven，并详细解释Maven的安装文件；其次还会介绍如何在主 流的IDE中集成Maven，以及Maven安装的最佳实践。  
2.1 在Windows上安装Maven  
2.2 在基于Unix的系统上安装Maven  
2.3 安装目录分析  
2.4 设置HTTP代理  
2.5 安装m2eclipse  
2.6 安装NetBeans Maven插件  
2.7 Maven安装最佳实践  
2.8 小结

**2.1 在Windows上安装Maven** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

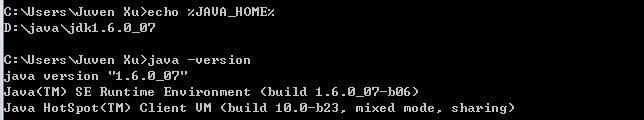
**2.1.1 检查JDK安装**

在安装Maven之前，首先要确认你已经正确安装了JDK。Maven可以运行在JDK 1.4及以上的版本上。本书的所有样例都基于JDK 5及以上版本。打开Windows的命令行，运行如下的命令来检查你的Java安装：

C:\Users\Juven Xu>echo %JAVA\_HOME%

C:\Users\Juven Xu>java -version

结果如图2-1所示：

  
 图2-1 Windows中检查Java安装

上述命令首先检查环境变量JAVA\_HOME是否指向了正确的JDK目录，接着尝试运行java命令。如果Windows无法执行java命令，或者无法找到JAVA\_HOME环境变量。你就需要检查Java是否安装了，或者环境变量是否设置正确。关于环境变量的设置，请参考2.1.3节。

**2.1.2 下载Maven**

请访问Maven的下载页面：<http://maven.apache.org/download.html>，其中包含针对不同平台的各种版本的Maven下载文件。对于首次接触Maven的读者来说，推荐使用Maven 3.0，，因此下载apache-maven-3.0-bin.zip。当然，如果你对Maven的源代码感兴趣并想自己构建Maven，还可以下载apache-maven-3.0 -src.zip。该下载页面还提供了md5校验和（checksum）文件和asc数字签名文件，可以用来检验Maven分发包的正确性和安全性。

在本书编写的时候，Maven 2的最新版本是2.2.1，Maven 3基本完全兼容Maven 2，而且较之于Maven 2它性能更好，还有不少功能的改进，如果你之前一直使用Maven 2，现在正犹豫是否要升级，那就大可不必担心了，快点尝试下Maven 3吧！

**2.1.3 本地安装**

将安装文件解压到你指定的目录中，如：

D:\bin>jar xvf "C:\Users\Juven Xu\Downloads\apache-maven-3.0--bin.zip"

这里的Maven安装目录是D:\bin\apache-maven-3.0，接着需要设置环境变量，将Maven安装配置到操作系统环境中。

打开系统属性面板（桌面上右键单击“我的电脑”→“属性”），点击**高级系统设置**，再点击**环境变量**，在**系统变量**中新建一个变量，变量名为*M2\_HOME*，变量值为Maven的安装目录*D:\bin\apache-maven-3.0*。点击**确定**，接着在系统变量中找到一个名为Path的变量，在变量值的末尾加上*%M2\_HOME%\bin;*，注意多个值之间需要有分号隔开，然后点击**确定**。至此，环境变量设置完成，详细情况如图2-2所示：



图2-2 Windows中系统环境变量配置

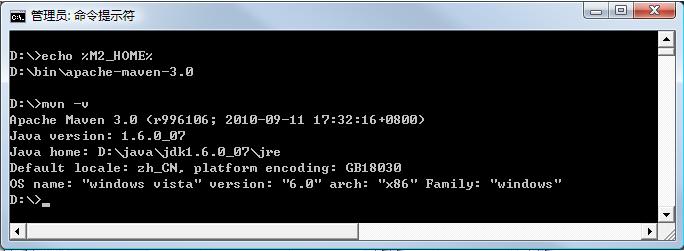
这里需要提一下的是Path环境变量，当我们在cmd中输入命令时，Windows首先会在当前目录中寻找可执行文件或脚本，如果没有找到，Windows会接着遍历环境变量Path中定义的路径。由于我们将*%M2\_HOME%\bin*添加到了Path中，而这里%M2\_HOME%实际上是引用了我们前面定义的另一个变量，其值是Maven的安装目录。因此，Windows会在执行命令时搜索目录*D:\bin\apache-maven-3.0\bin*，而mvn执行脚本的位置就是这里。

明白了环境变量的作用，现在打开一个新的cmd窗口（这里强调新的窗口是因为新的环境变量配置需要新的cmd窗口才能生效），运行如下命令检查Maven的安装情况：

C:\Users\Juven Xu>**echo %M2\_HOME%**

C:\Users\Juven Xu>mvn -v

运行结果如图2-3所示：

  
 图2-3 Windows中检查Maven安装

第一条命令**echo %M2\_HOME%**用来检查环境变量M2\_HOME是否指向了正确的Maven安装目录；而**mvn –version**执行了第一条Maven命令，以检查Windows是否能够找到正确的mvn执行脚本。

**2.1.4 升级Maven**

Maven还比较年轻，更新比较频繁，因此用户往往会需要更新Maven安装以获得更多更酷的新特性，以及避免一些旧的bug。

在Windows上更新Maven非常简便，只需要下载新的Maven安装文件，解压至本地目录，然后更新M2\_HOME环境变量便可。例如，假设Maven推出了新版本3.1，我们将其下载然后解压至目录*D:\bin\apache-maven-3.1*，接着遵照前一节描述的步骤编辑环境变量M2\_HOME，更改其值为D:\bin\apache-maven-3.1。至此，更新就完成了。同理，如果你需要使用某一个旧版本的Maven，也只需要编辑M2\_HOME环境变量指向旧版本的安装目录。

**2.2 在基于Unix的系统上安装Maven** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Maven是跨平台的，它可以在任何一种主流的操作系统上运行，本节将介绍如何在基于Unix的系统（包括Linux、Mac OS以及FreeBSD等）上安装Maven。

**2.2.1 下载和安装**

首先，与在Windows上安装Maven一样，需要检查JAVA\_HOME环境变量以及Java命令，细节不再赘述，命令如下：

juven@juven-ubuntu:~$ echo $JAVA\_HOME

juven@juven-ubuntu:~$ java –version

运行结果如图2-4所示：

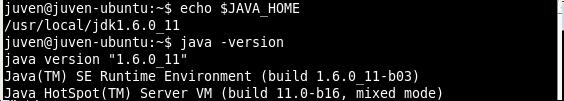


图2-4 Linux中检查Java安装

接着到http://maven.apache.org/download.html下载Maven安装文件，如apache-maven-3.0-bin.tar.gz，然后解压到本地目录：

juven@juven-ubuntu:bin$ tar -xvzf apache-maven-3.0-bin.tar.gz

现在已经创建好了一个Maven安装目录apache-maven-3.0，虽然直接使用该目录配置环境变量之后就能使用Maven了，但这里我更推荐做法是，在安装目录旁平行地创建一个符号链接，以方便日后的升级：

juven@juven-ubuntu:bin$ ln -s apache-maven-3.0 apache-maven  
juven@juven-ubuntu:bin$ ls -l  
total 4  
lrwxrwxrwx 1 juven juven   18 2009-09-20 15:43 apache-maven -> apache-maven-3.0  
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-3.0

接下来，我们需要设置M2\_HOME环境变量指向符号链接apache-maven-，并且把Maven安装目录下的bin/文件夹添加到系统环境变量PATH中去：

juven@juven-ubuntu:bin$ export M2\_HOME=/home/juven/bin/apache-maven  
juven@juven-ubuntu:bin$ export PATH=$PATH:$M2\_HOME/bin

一般来说，需要将这两行命令加入到系统的登录shell脚本中去，以我现在的Ubuntu 8.10为例，编辑~/.bashrc文件，添加这两行命令。这样，每次启动一个终端，这些配置就能自动执行。

至此，安装完成，我们可以运行以下命令检查Maven安装：

juven@juven-ubuntu:bin$ echo $M2\_HOME

juven@juven-ubuntu:bin$ mvn –version

运行结果如图2-5所示：

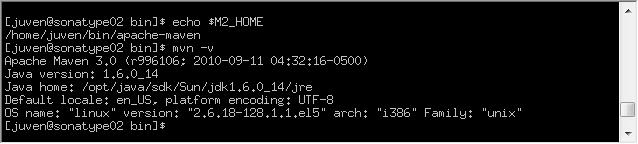


 图2-5 Linux中检查Maven安装

**2.2.2 升级Maven**

在基于Unix的系统上，可以利用符号链接这一工具来简化Maven的升级，不必像在Windows上那样，每次升级都必须更新环境变量。

前一小节中我们提到，解压Maven安装包到本地之后，平行地创建一个符号链接，然后在配置环境变量时引用该符号链接，这样做是为了方便升级。现在，假设我们需要升级到新的Maven 3.1版本，同理，将安装包解压到与前一版本平行的目录下，然后更新符号链接指向3.1版的目录便可：

juven@juven-ubuntu:bin$ rm apache-maven  
juven@juven-ubuntu:bin$ ln -s apache-maven-3.1/ apache-maven  
juven@juven-ubuntu:bin$ ls -l  
total 8  
lrwxrwxrwx 1 juven juven   17 2009-09-20 16:13 apache-maven -> apache-maven-3.1 /  
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-3.0drwxr-xr-x 2 juven juven 4096 2009-09-20 16:09 apache-maven-3.1

同理，可以很方便地切换到Maven的任意一个版本。现在升级完成了，可以运行**mvn -v**进行检查。

**2.3 安装目录分析** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章前面的内容讲述了如何在各种操作系统中安装和升级Maven。现在我们来仔细分析一下Maven的安装文件。

**2.3.1 M2\_HOME**

前面我们讲到设置M2\_HOME环境变量指向Maven的安装目录，本书之后所有使用M2\_HOME的地方都指代了该安装目录，让我们看一下该目录的结构和内容：

bin  
boot  
conf  
lib  
LICENSE.txt  
NOTICE.txt  
README.txt

* Bin： 该目录包含了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。其中*mvn*是基于UNIX平台的shell脚本，*mvn.bat*是基于Windows平台的bat脚本。在命令行输入任何一条mvn命令时，实际上就是在调用这些脚本。该目录还包含了*mvnDebug*和*mvnDebug.bat*两个文件，同样，前者是UNIX平台的shell脚本，后者是windows的bat脚本。那么mvn和mvnDebug有什么区别和关系呢？打开文件我们就可以看到，两者基本是一样的，只是mvnDebug多了一条MAVEN\_DEBUG\_OPTS配置，作用就是在运行Maven时开启debug，以便调试Maven本身。此外，该目录还包含*m2.conf*文件，这是classworlds的配置文件，稍微会介绍classworlds。
* 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*。直接修改该文件，就能在机器上全局地定制Maven的行为。一般情况下，我们更偏向于复制该文件至*~/.m2/*目录下（这里~表示用户目录），然后修改该文件，在用户范围定制Maven的行为。本书的后面将会多次提到该settings.xml，并逐步分析其中的各个元素。
* 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。（对于Maven 2来说，该目录只包含一个如*maven-2.2.1-uber.jar*的文件原本各为独立JAR文件的Maven模块和第三方类库都被拆解后重新合并到了这个JAR文件中）。可以说，这个lib目录就是真正的Maven。关于该文件，还有一点值得一提的是，用户可以在这个目录中找到Maven内置的超级POM，这一点在8.5小节详细解释。其他： *LICENSE.txt*记录了Maven使用的软件许可证Apache License Version 2.0； *NOTICE.txt*记录了Maven包含的第三方软件；而*README.txt*则包含了Maven的简要介绍，包括安装需求及如何安装的简要指令等等。

**2.3.2 ~/.m2**

在讲述该小节之前，我们先运行一条简单的命令：**mvn help:system**。该命令会打印出所有的Java系统属性和环境变量，这些信息对我们日常的编程工作很有帮助。这里暂不解释help:system涉及的语法，运行这条命令的目的是为了让Maven执行一个真正的任务。我们可以从命令行输出看到Maven会下载maven-help-plugin，包括pom文件和jar文件。这些文件都被下载到了Maven本地仓库中。

现在打开用户目录，比如当前的用户目录是*C:\Users\Juven Xu\*，你可以在Vista和Windows7中找到类似的用户目录。如果是更早版本的Windows，该目录应该类似于C:\Document and Settings\Juven Xu\。在基于Unix的系统上，直接输入**cd** 回车，就可以转到用户目录。为了方便，本书统一使用符号 **~** 指代用户目录。

在用户目录下，我们可以发现.m2文件夹。默认情况下，该文件夹下放置了Maven本地仓库.m2/repository。所有的Maven构件（artifact）都被存储到该仓库中，以方便重用。我们可以到*~/.m2/repository/org/apache/maven/plugins/maven-help-plugins/*目录下找到刚才下载的maven-help-plugin的pom文件和jar文件。Maven根据一套规则来确定任何一个构件在仓库中的位置，这一点本书第6章将会详细阐述。由于Maven仓库是通过简单文件系统透明地展示给Maven用户的，有些时候可以绕过Maven直接查看或修改仓库文件，在遇到疑难问题时，这往往十分有用。

默认情况下，*~/.m2*目录下除了repository仓库之外就没有其他目录和文件了，不过大多数Maven用户需要复制*M2\_HOME/conf/settings.xml*文件到*~/.m2/settings.xml*。这是一条最佳实践，我们将在本章最后一小节详细解释。

**2.4 设置HTTP代理** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

有时候你所在的公司由于安全因素考虑，要求你使用通过安全认证的代理访问因特网。这种情况下，就需要为Maven配置HTTP代理，才能让它正常访问外部仓库，以下载所需要的资源。

首先确认自己无法直接访问公共的Maven中央仓库，直接运行命令**ping repo1.maven.org**可以检查网络。如果真的需要代理，先检查一下代理服务器是否畅通，比如现在有一个IP地址为218.14.227.197，端口为3128的代理服务，我们可以运行**telnet 218.14.227.197 3128**来检测该地址的该端口是否畅通。如果得到出错信息，需要先获取正确的代理服务信息；如果telnet连接正确，则输入ctrl+]，然后q，回车，退出即可。

检查完毕之后，编辑~/.m2/settings.xml文件（如果没有该文件，则复制$M2\_HOME/conf/settings.xml）。添加代理配置如下：

<settings>  
…    
<proxies>

    <proxy>

      <id>my-proxy</id>

      <active>true</active>

      <protocol>http</protocol>

      <host>218.14.227.197</host>

      <port>3128</port>

      <!--

      <username>\*\*\*</username>

      <password>\*\*\*</password>

      <nonProxyHosts>repository.mycom.com|\*.google.com</nonProxyHosts>

      -->

    </proxy>

  </proxies>  
  …  
</settings>

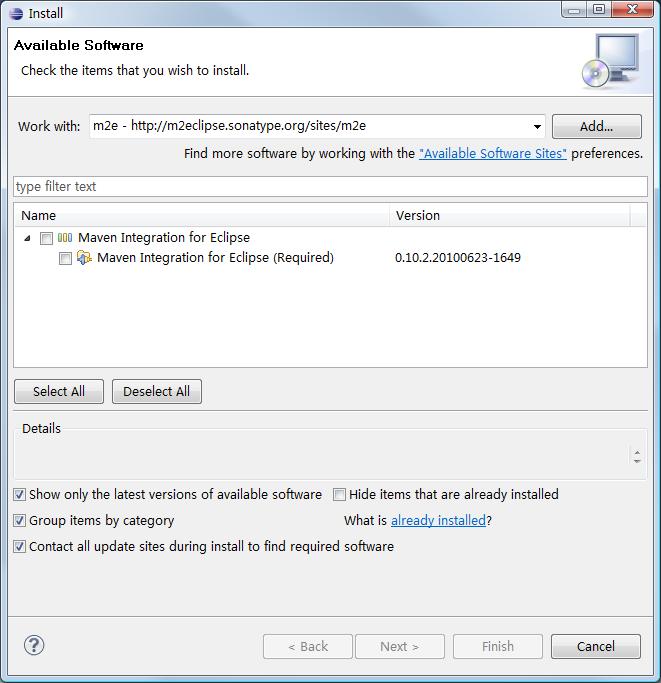
这段配置十分简单，proxies下可以有多个proxy元素，如果你声明了多个proxy元素，则默认情况下第一个被激活的proxy会生效。这里声明了一个id为my-proxy的代理，active的值为true表示激活该代理，protocol表示使用的代理协议，这里是http。当然，最重要的是指定正确的主机名（host元素）和端口（port元素）。上述XML配置中我注释掉了username、password、nonProxyHost几个元素，当你的代理服务需要认证时，就需要配置username和password。nonProxyHost元素用来指定哪些主机名不需要代理，可以使用 | 符号来分隔多个主机名。此外，该配置也支持通配符，如\*.google.com表示所有以google.com结尾的域名访问都不要通过代理。

**2.5 安装m2eclipse** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

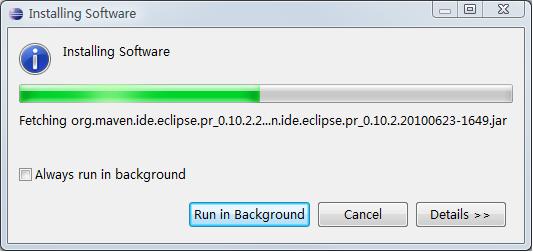
Eclipse是一款非常优秀的IDE。除了基本的语法标亮、代码补齐、XML编辑等基本功能外，最新版的Eclipse还能很好地支持重构，并且集成了JUnit、CVS、Mylyn等各种流行工具。可惜Eclipse默认没有集成对Maven的支持。幸运的是，由Maven之父Jason Van Zyl创立的Sonatype公司建立了m2eclipse项目，这是Eclipse下的一款十分强大的Maven插件，可以访问<http://m2eclipse.sonatype.org/> 了解更多该项目的信息。

本小节将先介绍如何安装m2eclipse插件，本书后续的章节会逐步介绍m2eclipse插件的使用。

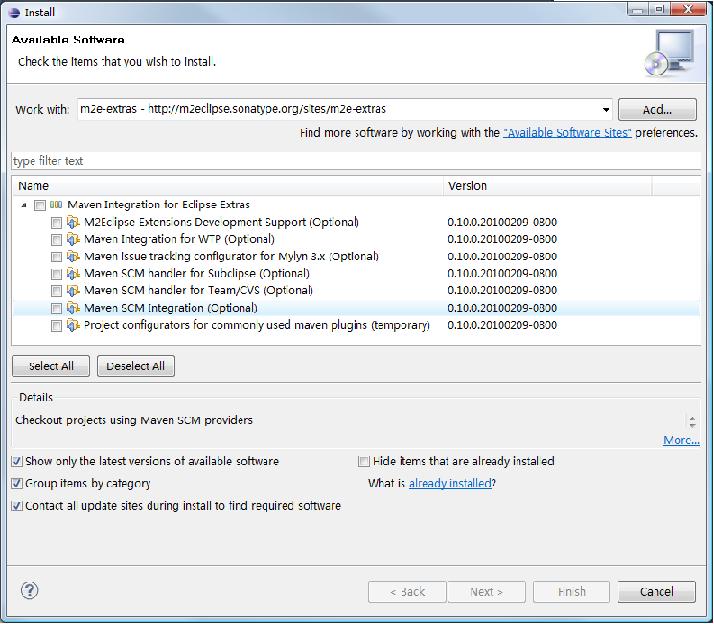
现在我以Eclipse 3.6为例逐步讲解m2eclipse的安装。启动Eclipse之后，在菜单栏中选择**Help**，然后选择**Install New Software…**，接着你会看到一个Install对话框，点击**Work with:**字段边上的**Add按钮**，你会得到一个新的Add Repository对话框，在**Name**字段中输入*m2e*，Location字段中输入<http://m2eclipse.sonatype.org/sites/m2e>，然后点击**OK**。Eclipse会下载m2eclipse安装站点上的资源信息。等待资源载入完成之后，我们再将其全部展开，就能看到图2-6所示的界面：

  
 图2-6  m2eclipse的核心安装资源列表

如图显示了m2eclipse的核心模块Maven Integration for Eclipse (Required)，选择后点击**Next >**，Eclipse会自动计算模块间依赖，然后给出一个将被安装的模块列表，确认无误后，继续点击**Next >**，这时我们会看到许可证信息，m2eclipse使用的开源许可证是Eclipse Public License v1.0，选择**I accept the terms of the license agreements**，然后点击**Finish**，接着就耐心等待Eclipse下载安装这些模块，如图2-7所示：

  
 图2-7：m2eclipse安装进度

除了核心组件之外，m2eclipse还提供了一组额外组件，主要是为了方便与其它工具如Subversion进行集成，这些组件的安装地址为<http://m2eclipse.sonatype.org/sites/m2e-extras>。使用前面类似的安装方法，我们可以看到如图2-8的组件列表：

  
 图2-8：m2eclipse的额外组件安装资源列表

下面简单解释一下这些组件的用途：

**1. 重要的**

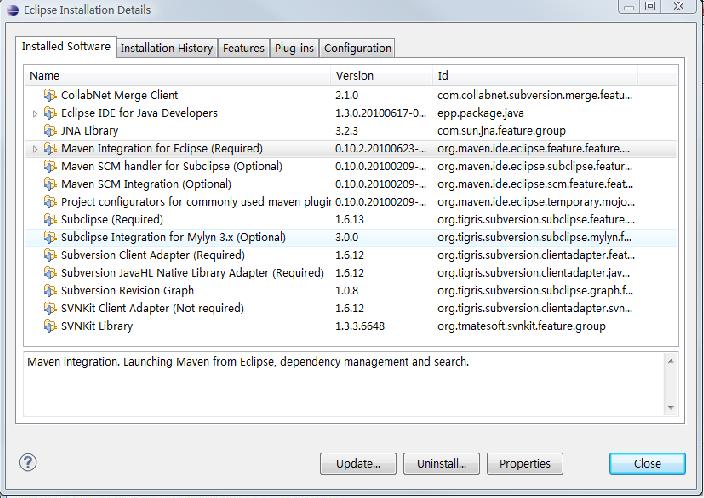
* Maven SCM handler for Subclipse  (Optional）：Subversion是非常流行的版本管理工具，该模块能够帮助我们直接从Subversion服务器签出Maven项目，不过前提是需要首先安装Subclipse（<http://subclipse.tigris.org/>）。
* Maven SCM Integration (Optional）：Eclipse环境中Maven与SCM集成核心的模块，它利用各种SCM工具如SVN实现Maven项目的签出和具体化等操作。

**2. 不重要的**

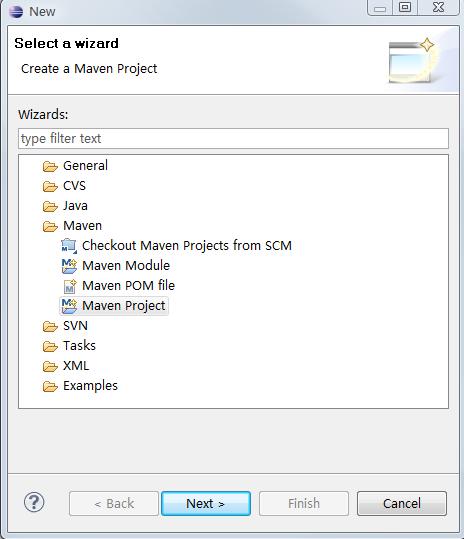
* Maven issue tracking configurator for Mylyn 3.x (Optional）：该模块能够帮助我们使用POM中的缺陷跟踪系统信息连接Mylyn至服务器。
* Maven SCM handler for Team/CVS (Optional）：该模块帮助我们从CVS服务器签出Maven项目，如果你还在使用CVS，就需要安装它。
* Maven Integration for WTP (Optional）：使用该模块可以让Eclipse自动读取POM信息并配置WTP项目。、
* M2eclipse Extensions Development Support (Optional)：用来支持扩展m2eclipse，一般用户不会用到。
* Project configurators for commonly used maven plugins (temporary)：一个临时的组件，用来支持一些Maven插件与Eclipse的集成，建议安装。

读者可以根据自己的需要安装相应组件，具体步骤不再赘述。

待安装完毕后，重启Eclipse，现在让我们验证一下m2eclipse是否正确安装了。首先，点击菜单栏中的**Help**，然后选择**About Eclipse**，在弹出的对话框中，点击**Installation Details**按钮，会得到一个对话框，在**Installed Software**标签栏中，检查刚才我们选择的模块是否在这个列表中，如图2-9所示：

  
 图2-9m2eclipse安装结果

如果一切没问题，我们再检查一下Eclipse现在是否已经支持创建Maven项目，依次点击菜单栏中的**File**→**New**→**Other**，在弹出的对话框中，找到Maven一项，再将其展开，你应该能够看到如图2-10所示的对话框：

  
 图2-10 Eclipse中创建Maven项目向导

如果一切正常，说明m2eclipse已经正确安装了。

最后，关于m2eclipse的安装，需要提醒的一点是，你可能会在使用m2eclipse时遇到类似这样的错误：

09-10-6 上午01时14分49秒: Eclipse is running in a JRE, but a JDK is required  
Some Maven plugins may not work when importing projects or updating source folders.

这是因为Eclipse默认是运行在JRE上的，而m2eclipse的一些功能要求使用JDK，解决方法是配置Eclipse安装目录的eclipse.ini文件，添加vm配置指向JDK，如：

--launcher.XXMaxPermSize

256m

-vm

D:\java\jdk1.6.0\_07\bin\javaw.exe

-vmargs

-Dosgi.requiredJavaVersion=1.5

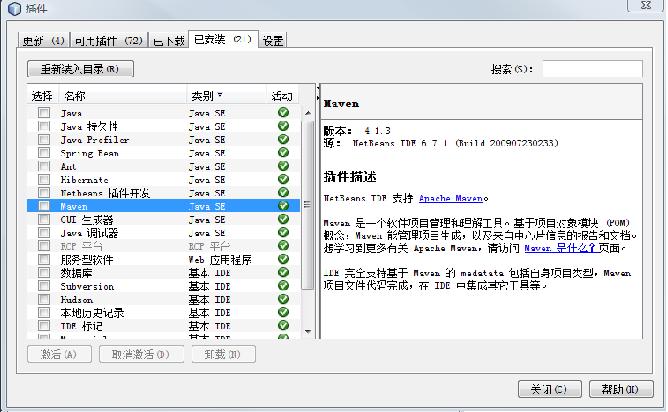
-Xms128m

-Xmx256m

**2.6 安装NetBeans Maven插件** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本小节会先介绍如何在NetBeans上安装Maven插件，后面的章节中还会介绍NetBeans中具体的Maven操作。

首先，如果你正在使用NetBeans 6.7及以上版本，那么Maven插件已经预装了。你可以检查Maven插件安装，点击菜单栏中的**工具**，接着选择**插件**，在弹出的插件对话框中选择**已安装**标签，你应该能够看到Maven插件，如图2-11所示：

  
 图2-11 已安装的NetBeans Maven插件

如果你在使用NetBeans 6.7之前的版本，或者由于某些原因NetBeans Maven插件被卸载了，那么你就需要安装NetBeans Maven插件，下面我们以NetBeans 6.1为例，介绍Maven插件的安装。

同样，点击菜单栏中的**工具**，选择**插件**，在弹出的插件对话框中选择**可用插件**标签，接着在右边的搜索框内输入Maven，这时你会在左边的列表中看到一个名为Maven的插件，选择该插件，然后点击下面的安装按钮，如图2-12所示：



图2-12 安装NetBeans Maven插件

接着在随后的对话框中根据提示操作，阅读相关许可证并接受，NetBeans会自动帮我们下载并安装Maven插件，结束之后会提示安装完成，之后再点击插件对话框的**已安装**标签，就能看到已经激活的Maven插件。

最后，为了确认Maven插件确实已经正确安装了，可以看一下NetBeans是否已经拥有创建Maven项目的相关菜单。在菜单栏中选择**文件**，然后选择**新建项目**，这时应该能够看到项目类别中有**Maven**一项，选择该类别，右边会相应地显示**Maven项目**和**基于现有POM的Maven项目**，如图2-13所示：

  
 图2-13 NetBeans中创建Maven项目向导

如果你能看到类似的对话框，说明NetBeans Maven已经正确安装了。

**2.7 Maven安装最佳实践** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本节介绍一些在安装Maven过程中不是必须的，但十分有用的实践。

**2.7.1 设置MAVEN\_OPTS环境变量**

本章前面介绍Maven安装目录时我们了解到，运行mvn命令实际上是执行了Java命令，既然是运行Java，那么运行Java命令可用的参数当然也应该在运行mvn命令时可用。这个时候，MAVEN\_OPTS环境变量就能派上用场。

我们通常需要设置MAVEN\_OPTS的值为：*-Xms128m -Xmx512m*，因为Java默认的最大可用内存往往不能够满足Maven运行的需要，比如在项目较大时，使用Maven生成项目站点需要占用大量的内存，如果没有该配置，我们很容易得到*java.lang.OutOfMemeoryError*。因此，一开始就配置该变量是推荐的做法。

关于如何设置环境变量，请参考前面设置M2\_HOME环境变量的做法，尽量不要直接修改mvn.bat或者mvn这两个Maven执行脚本文件。因为如果修改了脚本文件，升级Maven时你就不得不再次修改，一来麻烦，二来容易忘记。同理，我们应该尽可能地不去修改任何Maven安装目录下的文件。

**2.7.2 配置用户范围settings.xml**

Maven用户可以选择配置*$M2\_HOME/conf/settings.xml*或者*~/.m2/settings.xml*。前者是全局范围的，整台机器上的所有用户都会直接受到该配置的影响，而后者是用户范围的，只有当前用户才会受到该配置的影响。

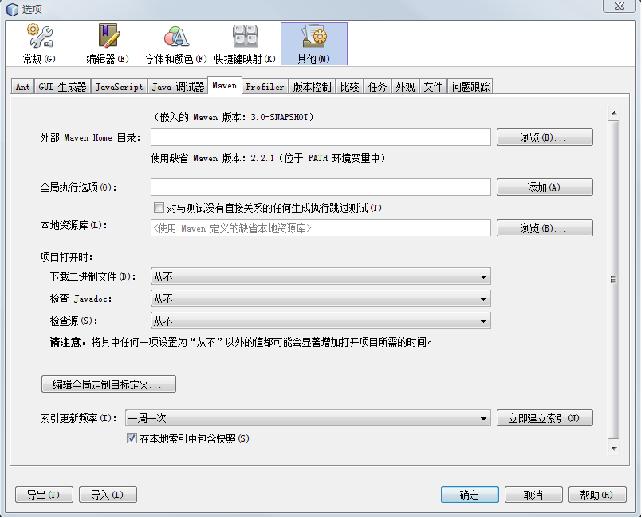
我们推荐使用用户范围的settings.xml，主要原因是为了避免无意识地影响到系统中的其他用户。当然，如果你有切实的需求，需要统一系统中所有用户的settings.xml配置，当然应该使用全局范围的settings.xml。

除了影响范围这一因素，配置用户范围settings.xml文件还便于Maven升级。直接修改conf目录下的settings.xml会导致Maven升级不便，每次升级到新版本的Maven，都需要复制settings.xml文件，如果使用*~/.m2*目录下的settings.xml，就不会影响到Maven安装文件，升级时就不需要触动settings.xml文件。

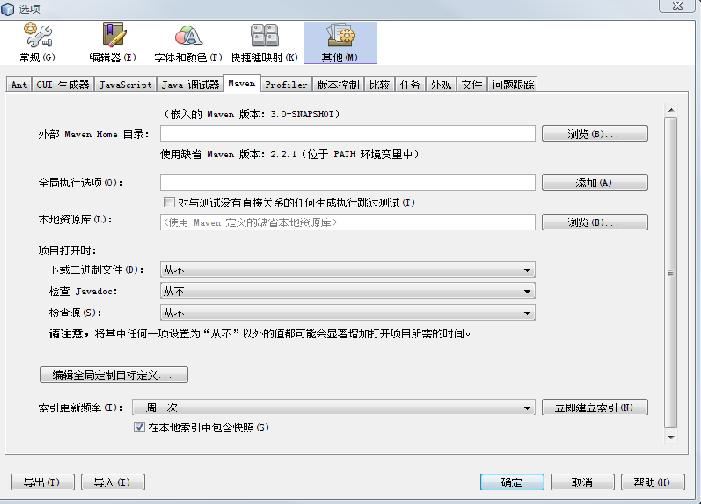
**2.7.3 不要使用IDE内嵌的Maven**

无论是Eclipse还是NetBeans，当我们集成Maven时，都会安装上一个内嵌的Maven，这个内嵌的Maven通常会比较新，但不一定很稳定，而且往往也会和我们在命令行使用的Maven不是同一个版本。这里有会出现两个潜在的问题：首先，较新版本的Maven存在很多不稳定因素，容易造成一些难以理解的问题；其次，除了IDE，我们也经常还会使用命令行的Maven，如果版本不一致，容易造成构建行为的不一致，这是我们所不希望看到的。因此，我们应该在IDE中配置Maven插件时使用与命令行一致的Maven。

在m2eclipse环境中，点击菜单栏中的**Windows**，然后选择**Preferences**，在弹出的对话框中，展开左边的**Maven**项，选择**Installation**子项，在右边的面板中，我们能够看到有一个默认的**Embedded** Maven安装被选中了，点击**Add…**然后选择我们的Maven安装目录M2\_HOME，添加完毕之后选择这一个外部的Maven，如图2-14所示：

  
 图2-14 在Eclipse中使用外部Maven

NetBeans Maven插件默认会侦测PATH环境变量，因此会直接使用与命令行一致的Maven环境。依次点击菜单栏中的**工具**→**选项**→**其他**→**Maven**标签栏，你就能看到如图2-15所示的配置：

  
 图2-15 在NetBeans中使用外部Maven

**2.8 小结** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章详细介绍了在各种操作系统平台上安装Maven，并对Maven安装目录进行了深入的分析，在命令行的基础上，本章又进一步介绍了Maven与主流IDE Eclipse及NetBeans的集成，本章最后还介绍了一些与Maven安装相关的最佳实践。本书下一章会创建一个Hello World项目，带领读者配置和构建Maven项目。

**第3章 Maven使用入门** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

到目前为止，我们已经大概了解并安装好了Maven，现在，我们开始 创建一个最简单的Hello World项目。如果你是初次接触Maven，我建议你按照本章的内容一步步地编写代码并执行，可能你会碰到一些概念暂时难以理解，不用着急，记下这些疑 难点，相信本书的后续章节会帮你逐一解答。   
3.1 编写POM   
3.2 编写主代码   
3.3 编写测试代码   
3.4 打包和运行   
3.5 使用Archetype生成项目骨架   
3.6 m2eclipse简单使用   
3.7 NetBeans Maven插件简单使用   
3.8 小结

**3.1 编写POM** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

就像Make的Makefile，Ant的build.xml一样，Maven项目的核心是pom.xml。POM（**P**roject **O**bject **M**odel，项目对象模型）定义了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。现在我们先为Hello World项目编写一个最简单的pom.xml。

首先创建一个名为hello-world的文件夹（本书中各章的代码都会对应一个以ch开头的项目），打开该文件夹，新建一个名为pom.xml的文件，输入其内容如代码清单3-1：

代码清单3-1：Hello World的POM

1. <span style="font-size: small;"><?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5. http://maven.apache.org/maven-v4\_0\_0.xsd">
6. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.juvenxu.mvnbook</groupId>
8. <artifactId>hello-world</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Hello World Project</name>
11. </project>
12. </span>

     代码的第一行是XML头，指定了该xml文档的版本和编码方式。紧接着是project元素，project是所有pom.xml的根元素，它还声明了一些POM相关的命名空间及xsd元素，虽然这些属性不是必须的，但使用这些属性能够让第三方工具（如IDE中的XML编辑器）帮助我们快速编辑POM。

根元素下的第一个子元素modelVersion指定了当前POM模型的版本，对于Maven2及Maven 3来说，它只能是4.0.0。

这段代码中最重要的是groupId，artifactId和version三行。这三个元素定义了一个项目基本的坐标，在Maven的世界，任何的jar、pom或者war都是以基于这些基本的坐标进行区分的。

groupId定义了项目属于哪个组，这个组往往和项目所在的组织或公司存在关联，譬如你在googlecode上建立了一个名为myapp的项目，那么groupId就应该是com.googlecode.myapp，如果你的公司是mycom，有一个项目为myapp，那么groupId就应该是com.mycom.myapp。本书中所有的代码都基于groupId com.juvenxu.mvnbook。

artifactId定义了当前Maven项目在组中唯一的ID，我们为这个Hello World项目定义artifactId为hello-world，本书其他章节代码会被分配其他的artifactId。而在前面的groupId为com.googlecode.myapp的例子中，你可能会为不同的子项目（模块）分配artifactId，如：myapp-util、myapp-domain、myapp-web等等。

顾名思义，version指定了Hello World项目当前的版本——1.0-SNAPSHOT。SNAPSHOT意为快照，说明该项目还处于开发中，是不稳定的版本。随着项目的发展，version会不断更新，如升级为1.0、1.1-SNAPSHOT、1.1、2.0等等。本书的6.5小节会详细介绍SNAPSHOT，第13章介绍如何使用Maven管理项目版本的升级发布。

最后一个name元素声明了一个对于用户更为友好的项目名称，虽然这不是必须的，但我还是推荐为每个POM声明name，以方便信息交流。

没有任何实际的Java代码，我们就能够定义一个Maven项目的POM，这体现了Maven的一大优点，它能让项目对象模型最大程度地与实际代码相独立，我们可以称之为解耦，或者正交性，这在很大程度上避免了Java代码和POM代码的相互影响。比如当项目需要升级版本时，只需要修改POM，而不需要更改Java代码；而在POM稳定之后，日常的Java代码开发工作基本不涉及POM的修改。

**3.2 编写主代码** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

项目主代码和测试代码不同，项目的主代码会被打包到最终的构件中（比如jar），而测试代码只在运行测试时用到，不会被打包。默认情况下，Maven假设项目主代码位于*src/main/java*目录，我们遵循Maven的约定，创建该目录，然后在该目录下创建文件com/juvenxu/mvnbook/helloworld/*HelloWorld.java*，其内容如代码清单3-2：

代码清单3-2：Hello World的主代码

1. <span style="font-size: small;">package com.juvenxu.mvnbook.helloworld;
3. public class HelloWorld
4. {
5. public String sayHello()
6. {
7. return "Hello Maven";
8. }
10. public static void main(String[] args)
11. {
12. System.out.print( new HelloWorld().sayHello() );
13. }
14. }
15. </span>

       这是一个简单的Java类，它有一个sayHello()方法，返回一个String。同时这个类还带有一个main方法，创建一个HelloWorld实例，调用sayHello()方法，并将结果输出到控制台。

关于该Java代码有两点需要注意。首先，在95%以上的情况下，我们应该把项目主代码放到*src/main/java/*目录下（遵循Maven的约定），而无须额外的配置，Maven会自动搜寻该目录找到项目主代码。其次，该Java类的包名是com.juvenxu.mvnbook.helloworld，这与我们之前在POM中定义的groupId和artifactId相吻合。一般来说，项目中Java类的包都应该基于项目的groupId和artifactId，这样更加清晰，更加符合逻辑，也方便搜索构件或者Java类。

代码编写完毕后，我们使用Maven进行编译，在项目根目录下运行命令 **mvn clean compile** ，我们会得到如下输出：

1. <span style="font-size: small;">[INFO] Scanning for projects...
2. [INFO] ------------------------------------------------------------------------
3. [INFO] Building Maven Hello World Project
4. [INFO]    task-segment: [clean, compile]
5. [INFO] ------------------------------------------------------------------------
6. [INFO] [clean:clean {execution: default-clean}]
7. [INFO] Deleting directory D:\code\hello-world\target
8. [INFO] [resources:resources {execution: default-resources}]
9. [INFO] skip non existing resourceDirectory D: \code\hello-world\src\main\resources
10. [INFO] [compiler:compile {execution: default-compile}]
11. [INFO] Compiling 1 source file to D: \code\hello-world\target\classes
12. [INFO] ------------------------------------------------------------------------
13. [INFO] BUILD SUCCESSFUL
14. [INFO] ------------------------------------------------------------------------
15. [INFO] Total time: 1 second
16. [INFO] Finished at: Fri Oct 09 02:08:09 CST 2009
17. [INFO] Final Memory: 9M/16M
18. [INFO] ------------------------------------------------------------------------
19. </span>

clean告诉Maven清理输出目录*target/*，compile告诉Maven编译项目主代码，从输出中我们看到Maven首先执行了clean:clean任务，删除*target/*目录，默认情况下Maven构建的所有输出都在*target/*目录中；接着执行resources:resources任务（未定义项目资源，暂且略过）；最后执行compiler:compile任务，将项目主代码编译至*target/classes*目录(编译好的类为*com/juvenxu/mvnbook/helloworld/HelloWorld.Class*）。

上文提到的clean:clean、resources:resources，以及compiler:compile对应了一些Maven插件及插件目标，比如clean:clean是clean插件的clean目标，compiler:compile是compiler插件的compile目标，后文会详细讲述Maven插件及其编写方法。

至此，Maven在没有任何额外的配置的情况下就执行了项目的清理和编译任务，接下来，我们编写一些单元测试代码并让Maven执行自动化测试。

**3.3 编写测试代码** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

为了使项目结构保持清晰，主代码与测试代码应该分别位于独立的目录中。3.2节讲过Maven项目中默认的主代码目录是*src/main/java*，对应地，Maven项目中默认的测试代码目录是*src/test/java*。因此，在编写测试用例之前，我们先创建该目录。

在Java世界中，由Kent Beck和Erich Gamma建立的JUnit是事实上的单元测试标准。要使用JUnit，我们首先需要为Hello World项目添加一个JUnit依赖，修改项目的POM如代码清单3-3：

代码清单3-3：为Hello World的POM添加依赖

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5. http://maven.apache.org/maven-v4\_0\_0.xsd">
6. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.juvenxu.mvnbook</groupId>
8. <artifactId>hello-world</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Hello World Project</name>
11. <dependencies>
12. <dependency>
13. <groupId>junit</groupId>
14. <artifactId>junit</artifactId>
15. <version>4.7</version>
16. <scope>test</scope>
17. </dependency>
18. </dependencies>
19. </project>

     代码中添加了dependencies元素，该元素下可以包含多个dependency元素以声明项目的依赖，这里我们添加了一个依赖——groupId是junit，artifactId是junit，version是4.7。前面我们提到groupId、artifactId和version是任何一个Maven项目最基本的坐标，JUnit也不例外，有了这段声明，Maven就能够自动下载junit-4.7.jar。也许你会问，Maven从哪里下载这个jar呢？在Maven之前，我们可以去JUnit的官网下载分发包。而现在有了Maven，它会自动访问中央仓库（<http://repo1.maven.org/maven2/>），下载需要的文件。读者也可以自己访问该仓库，打开路径junit/junit/4.7/，就能看到junit-4.7.pom和junit-4.7.jar。本书第6章会详细介绍Maven仓库及中央仓库。

上述POM代码中还有一个值为test的元素scope，scope为依赖范围，若依赖范围为test则表示该依赖只对测试有效，换句话说，测试代码中的import JUnit代码是没有问题的，但是如果我们在主代码中用import JUnit代码，就会造成编译错误。如果不声明依赖范围，那么默认值就是compile，表示该依赖对主代码和测试代码都有效。

配置了测试依赖，接着就可以编写测试类，回顾一下前面的HelloWorld类，现在我们要测试该类的sayHello()方法，检查其返回值是否为“Hello Maven”。在src/test/java目录下创建文件，其内容如代码清单3-4：

代码清单3-4：Hello World的测试代码

1. package com.juvenxu.mvnbook.helloworld;
2. import static org.junit.Assert.assertEquals;
3. import org.junit.Test;
5. public class HelloWorldTest
6. {
7. @Test
8. public void testSayHello()
9. {
10. HelloWorld helloWorld = new HelloWorld();
12. String result = helloWorld.sayHello();
14. assertEquals( "Hello Maven", result );
15. }
16. }

一个典型的单元测试包含三个步骤：一，准备测试类及数据；二，执行要测试的行为；三，检查结果。上述样例中，我们首先初始化了一个要测试的HelloWorld实例，接着执行该实例的sayHello()方法并保存结果到result变量中，最后使用JUnit框架的Assert类检查结果是否为我们期望的”Hello Maven”。在JUnit 3中，约定所有需要执行测试的方法都以test开头，这里我们使用了JUnit 4，但我们仍然遵循这一约定，在JUnit 4中，需要执行的测试方法都应该以@Test进行标注。

测试用例编写完毕之后就可以调用Maven执行测试，运行 **mvn clean test** ：

1. [INFO] Scanning for projects...
2. [INFO] ------------------------------------------------------------------------
3. [INFO] Building Maven Hello World Project
4. [INFO]    task-segment: [clean, test]
5. [INFO] ------------------------------------------------------------------------
6. [INFO] [clean:clean {execution: default-clean}]
7. [INFO] Deleting directory D:\git-juven\mvnbook\code\hello-world\target
8. [INFO] [resources:resources {execution: default-resources}]
9. …
10. Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.pom
11. 1K downloaded  (junit-4.7.pom)
12. [INFO] [compiler:compile {execution: default-compile}]
13. [INFO] Compiling 1 source file to D: \code\hello-world\target\classes
14. [INFO] [resources:testResources {execution: default-testResources}]
15. …
16. Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.jar
17. 226K downloaded  (junit-4.7.jar)
18. [INFO] [compiler:testCompile {execution: default-testCompile}]
19. [INFO] Compiling 1 source file to D:\ code\hello-world\target\test-classes
20. [INFO] ------------------------------------------------------------------------
21. [ERROR] BUILD FAILURE
22. [INFO] ------------------------------------------------------------------------
23. [INFO] Compilation failure
24. D:\code\hello-world\src\test\java\com\juvenxu\mvnbook\helloworld\HelloWorldTest.java:[8,5] -source 1.3 中不支持注释
25. （请使用 -source 5 或更高版本以启用注释）
26. @Test
27. [INFO] ------------------------------------------------------------------------
28. [INFO] For more information, run Maven with the -e switch
29. …

不幸的是构建失败了，不过我们先耐心分析一下这段输出（为了本书的简洁，一些不重要的信息我用省略号略去了）。命令行输入的是mvn clean test，而Maven实际执行的可不止这两个任务，还有clean:clean、resources:resources、compiler:compile、resources:testResources以及compiler:testCompile。暂时我们需要了解的是，在Maven执行测试（test）之前，它会先自动执行项目主资源处理，主代码编译，测试资源处理，测试代码编译等工作，这是Maven生命周期的一个特性，本书后续章节会详细解释Maven的生命周期。

从输出中我们还看到：Maven从中央仓库下载了junit-4.7.pom和junit-4.7.jar这两个文件到本地仓库（*~/.m2/repository*）中，供所有Maven项目使用。

构建在执行compiler:testCompile任务的时候失败了，Maven输出提示我们需要使用-source 5或更高版本以启动注释，也就是前面提到的JUnit 4的@Test注解。这是Maven初学者常常会遇到的一个问题。由于历史原因，Maven的核心插件之一compiler插件默认只支持编译Java 1.3，因此我们需要配置该插件使其支持Java 5，见代码清单3-5：

代码清单3-5：配置maven-compiler-plugin支持Java 5

1. <project>
2. …
3. <build>
4. <plugins>
5. <plugin>
6. <groupId>org.apache.maven.plugins</groupId>
7. <artifactId>maven-compiler-plugin</artifactId>
8. <configuration>
9. <source>1.5</source>
10. <target>1.5</target>
11. </configuration>
12. </plugin>
13. </plugins>
14. </build>
15. …
16. </project>

该POM省略了除插件配置以外的其他部分，我们暂且不去关心插件配置的细节，只需要知道compiler插件支持Java 5的编译。现在再执行**mvn clean test，**输出如下：

1. …
2. [INFO] [compiler:testCompile {execution: default-testCompile}]
3. [INFO] Compiling 1 source file to D: \code\hello-world\target\test-classes
4. [INFO] [surefire:test {execution: default-test}]
5. [INFO] Surefire report directory: D:\code\hello-world\target\surefire-reports
6. -------------------------------------------------------
7. T E S T S
8. -------------------------------------------------------
9. Running com.juvenxu.mvnbook.helloworld.HelloWorldTest
10. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.055 sec
11. Results :
12. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
13. [INFO] ------------------------------------------------------------------------
14. [INFO] BUILD SUCCESSFUL
15. [INFO] ------------------------------------------------------------------------
16. …

我们看到compiler:testCompile任务执行成功了，测试代码通过编译之后在*target/test-classes*下生成了二进制文件，紧接着surefire:test任务运行测试，surefire是Maven世界中负责执行测试的插件，这里它运行测试用例HelloWorldTest，并且输出测试报告，显示一共运行了多少测试，失败了多少，出错了多少，跳过了多少。显然，我们的测试通过了——BUILD SUCCESSFUL。

**3.4 打包和运行** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

将项目进行编译、测试之后，下一个重要步骤就是打包（package）。Hello World的POM中没有指定打包类型，使用默认打包类型jar，我们可以简单地执行命令 **mvn clean package** 进行打包，可以看到如下输出：

1. …
2. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
4. [INFO] [jar:jar {execution: default-jar}]
5. [INFO] Building jar: D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
6. [INFO]
7. --------------------------------------------------------------------
8. [INFO] BUILD SUCCESSFUL
9. …

类似地，Maven会在打包之前执行编译、测试等操作。这里我们看到jar:jar任务负责打包，实际上就是jar插件的jar目标将项目主代码打包成一个名为*hello-world-1.0-SNAPSHOT.jar*的文件，该文件也位于*target/*输出目录中，它是根据artifact-version.jar规则进行命名的，如有需要，我们还可以使用*finalName*来自定义该文件的名称，这里暂且不展开，本书后面会详细解释。

至此，我们得到了项目的输出，如果有需要的话，就可以复制这个jar文件到其他项目的Classpath中从而使用HelloWorld类。但是，如何才能让其他的Maven项目直接引用这个jar呢？我们还需要一个安装的步骤，执行 **mvn clean install**：

1. …
2. [INFO] [jar:jar {execution: default-jar}]
3. [INFO] Building jar: D: \code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
4. [INFO] [install:install {execution: default-install}]
5. [INFO] Installing D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar to C:\Users\juven\.m2\repository\com\juvenxu\mvnbook\hello-world\1.0-SNAPSHOT\hello-world-1.0-SNAPSHOT.jar
6. [INFO]
7. ------------------------------------------------------------------------
8. [INFO] BUILD SUCCESSFUL
9. …

在打包之后，我们又执行了安装任务install:install，从输出我们看到该任务将项目输出的jar安装到了Maven本地仓库中，我们可以打开相应的文件夹看到Hello World项目的pom和jar。之前讲述JUnit的POM及jar的下载的时候，我们说只有构件被下载到本地仓库后，才能由所有Maven项目使用，这里是同样的道理，只有将Hello World的构件安装到本地仓库之后，其他Maven项目才能使用它。

我们已经将体验了Maven最主要的命令：**mvn clean compile、mvn clean test、mvn clean package、mvn clean install**。执行test之前是会先执行compile的，执行package之前是会先执行test的，而类似地，install之前会执行package。我们可以在任何一个Maven项目中执行这些命令，而且我们已经清楚它们是用来做什么的。

到目前为止，我们还没有运行Hello World项目，不要忘了HelloWorld类可是有一个main方法的。默认打包生成的jar是不能够直接运行的，因为带有main方法的类信息不会添加到manifest中(我们可以打开jar文件中的*META-INF/MANIFEST.MF*文件，将无法看到Main-Class一行)。为了生成可执行的jar文件，我们需要借助maven-shade-plugin，配置该插件如下：

1. <plugin>
2. <groupId>org.apache.maven.plugins</groupId>
3. <artifactId>maven-shade-plugin</artifactId>
4. <version>1.2.1</version>
5. <executions>
6. <execution>
7. <phase>package</phase>
8. <goals>
9. <goal>shade</goal>
10. </goals>
11. <configuration>
12. <transformers>
13. <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">            <mainClass>com.juvenxu.mvnbook.helloworld.HelloWorld</mainClass>
14. </transformer>
15. </transformers>
16. </configuration>
17. </execution>
18. </executions>
19. </plugin>

plugin元素在POM中的相对位置应该在<project><build><plugins>下面。我们配置了mainClass为com.juvenxu.mvnbook.helloworld.HelloWorld，项目在打包时会将该信息放到MANIFEST中。现在执行 **mvn clean install** ，待构建完成之后打开target/目录，我们可以看到*hello-world-1.0-SNAPSHOT.jar*和*original-hello-world-1.0-SNAPSHOT.jar*，前者是带有Main-Class信息的可运行jar，后者是原始的jar，打开*hello-world-1.0-SNAPSHOT.jar*的*META-INF/MANIFEST.MF*，可以看到它包含这样一行信息：

Main-Class: com.juvenxu.mvnbook.helloworld.HelloWorld

现在，我们在项目根目录中执行该jar文件：

D: \code\hello-world>java -jar target\hello-world-1.0-SNAPSHOT.jar

Hello Maven

控制台输出为Hello Maven，这正是我们所期望的。

本小节介绍了Hello World项目，侧重点是Maven而非Java代码本身，介绍了POM、Maven项目结构、以及如何编译、测试、打包，等等。

**3.5 使用Archetype生成项目骨架** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

Hello World项目中有一些Maven的约定：在项目的根目录中放置pom.xml，在src/main/java目录中放置项目的主代码，在src/test/java中放置项目的测试代码。我之所以一步一步地展示这些步骤，是为了能让可能是Maven初学者的你得到最实际的感受。我们称这些基本的目录结构和pom.xml文件内容称为项目的骨架，当你第一次创建项目骨架的时候，你还会饶有兴趣地去体会这些默认约定背后的思想，第二次，第三次，你也许还会满意自己的熟练程度，但第四、第五次做同样的事情，就会让程序员恼火了，为此Maven提供了Archetype以帮助我们快速勾勒出项目骨架。

还是以Hello World为例，我们使用maven archetype来创建该项目的骨架，离开当前的Maven项目目录。

如果是Maven 3，简单的运行：

mvn archetype:generate

如果是Maven 2，最好运行如下命令：

mvn org.apache.maven.plugins:maven-archetype-plugin:2.0-alpha-5:generate

很多资料会让你直接使用更为简单的 mvn archetype:generate 命令，但在Maven2中这是不安全的，因为该命令没有指定archetype插件的版本，于是Maven会自动去下载最新的版本，进而可能得到不稳定的SNAPSHOT版本，导致运行失败。然而在Maven 3中，即使用户没有指定版本，Maven也只会解析最新的稳定版本，因此这是安全的，具体内容见7.7小节。

我们实际上是在运行插件maven-archetype-plugin，注意冒号的分隔，其格式为 *groupId:artifactId:version:goal* ，org.apache.maven.plugins 是maven官方插件的groupId，maven-archetype-plugin 是archetype插件的artifactId，2.0-alpha-5 是目前该插件最新的稳定版，generate是我们要使用的插件目标。

紧接着我们会看到一段长长的输出，有很多可用的archetype供我们选择，包括著名的Appfuse项目的archetype，JPA项目的archetype等等。每一个archetype前面都会对应有一个编号，同时命令行会提示一个默认的编号，其对应的archetype为maven-archetype-quickstart，我们直接回车以选择该archetype，紧接着Maven会提示我们输入要创建项目的groupId、artifactId、 version、以及包名package，如下输入并确认：

1. Define value for groupId: : com.juvenxu.mvnbook
2. Define value for artifactId: : hello-world
3. Define value for version:  1.0-SNAPSHOT: :
4. Define value for package:  com.juvenxu.mvnbook: : com.juvenxu.mvnbook.helloworld
5. Confirm properties configuration:
6. groupId: com.juvenxu.mvnbook
7. artifactId: hello-world
8. version: 1.0-SNAPSHOT
9. package: com.juvenxu.mvnbook.helloworld
10. Y: : Y

Archetype插件将根据我们提供的信息创建项目骨架。在当前目录下，Archetype插件会创建一个名为hello-world（我们定义的artifactId）的子目录，从中可以看到项目的基本结构：基本的*pom.xml*已经被创建，里面包含了必要的信息以及一个junit依赖；主代码目录*src/main/java*已经被创建，在该目录下还有一个Java类*com.juvenxu.mvnbook.helloworld.App*，注意这里使用到了我们刚才定义的包名，而这个类也仅仅只有一个简单的输出Hello World!的main方法；测试代码目录*src/test/java*也被创建好了，并且包含了一个测试用例*com.juvenxu.mvnbook.helloworld.AppTest*。

Archetype可以帮助我们迅速地构建起项目的骨架，在前面的例子中，我们完全可以在Archetype生成的骨架的基础上开发Hello World项目以节省我们大量时间。

此外，我们这里仅仅是看到了一个最简单的archetype，如果你有很多项目拥有类似的自定义项目结构以及配置文件，你完全可以一劳永逸地开发自己的archetype，然后在这些项目中使用自定义的archetype来快速生成项目骨架，本书后面的章节会详细阐述如何开发Maven Archetype。

**3.6 m2eclipse简单使用** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

介绍前面Hello World项目的时候，我们并没有涉及IDE，如此简单的一个项目，使用最简单的编辑器也能很快完成，但对于稍微大一些的项目来说，没有IDE就是不可想象的，本节我们先介绍m2eclipse的基本使用。

**3.6.1 导入Maven项目**

第2章介绍了如何安装m2eclipse，现在，我们使用m2ecilpse导入Hello World项目。选择菜单项**File**，然后选择**Import**，我们会看到一个Import对话框，在该对话框中选择General目录下的**Maven Projects**，然后点击**Next**，就会出现**Import Projects**对话框，在该对话框中点击**Browse…**选择Hello World的根目录（即包含pom.xml文件的那个目录），这时对话框中的**Projects:**部分就会显示该目录包含的Maven项目，如图3-1所示：

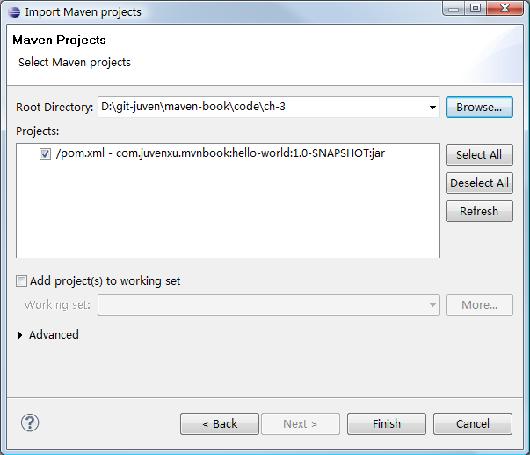


 图3-1 在Eclipse中导入Maven项目

点击Finish之后，m2ecilpse就会将该项目导入到当前的workspace中，导入完成之后，我们就可以在Package Explorer视图中看到如图3-2所示的项目结构：

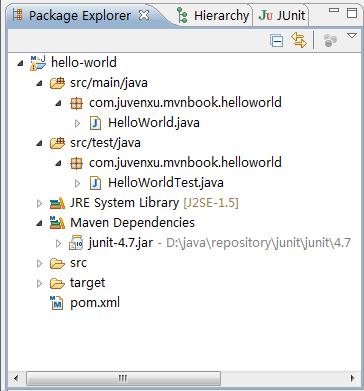


 图3-2 Eclipse中导入的Maven项目结构

我们看到主代码目录*src/main/java*和测试代码目录*src/test/java*成了Eclipse中的资源目录，包和类的结构也十分清晰，当然pom.xml永远在项目的根目录下，而从这个视图中我们甚至还能看到项目的依赖junit-4.7.jar，其实际的位置指向了Maven本地仓库（这里我自定义了Maven本地仓库地址为*D:\java\repository*，后续章节会介绍如何自定义本地仓库位置）。

**3.6.2 创建Maven项目**

创建一个Maven项目也十分简单，选择菜单项**File -> New -> Other**，在弹出的对话框中选择Maven下的**Maven Project**，然后点击**Next >**，在弹出的**New Maven Project**对话框中，我们使用默认的选项（不要选择Create a simple project选项，那样我们就能使用Maven Archetype），点击**Next >**，此时m2eclipse会提示我们选择一个Archetype，我们选择**maven-archetype-quickstart**，再点击**Next >**。由于m2eclipse实际上是在使用maven-archetype-plugin插件创建项目，因此这个步骤与上一节我们使用archetype创建项目骨架类似，输入groupId,、artifactId、version、package（暂时我们不考虑Properties），如图3-3所示：

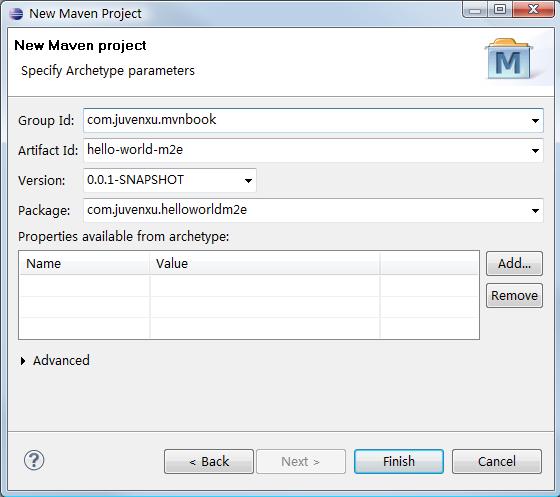


 图3-3 在Eclipse中使用Archetype创建项目

注意，为了不和前面已导入的Hello World项目产生冲突和混淆，我们使用不同的artifactId和package。OK，点击Finish，Maven项目就创建完成了，其结构与前一个已导入的Hello World项目基本一致。

**3.6.3 运行mvn命令**

我们需要在命令行输入如mvn clean install之类的命令来执行maven构建，m2eclipse中也有对应的功能，在Maven项目或者pom.xml上右击，再选择Run As，就能看到如下的常见Maven命令，如图3-4所示：

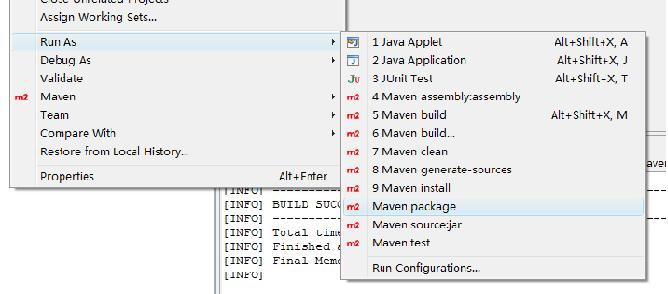


 图3-4 在Eclipse中运行默认mvn命令

选择想要执行的Maven命令就能执行相应的构建，同时我们也能在Eclipse的console中看到构建输出。这里常见的一个问题是，默认选项中没有我们想要执行的Maven命令怎么办？比如，默认带有mvn test，但我们想执行mvn clean test，很简单，选择**Maven buid…** 以自定义Maven运行命令，在弹出对话框中的**Goals**一项中输入我们想要执行的命令，如clean test，设置一下Name，点击**Run**即可。并且，下一次我们选择**Maven build**，或者使用快捷键Alt + Shift + X, M快速执行Maven构建的时候，上次的配置直接就能在历史记录中找到。图3-5就是自定义Maven运行命令的界面：

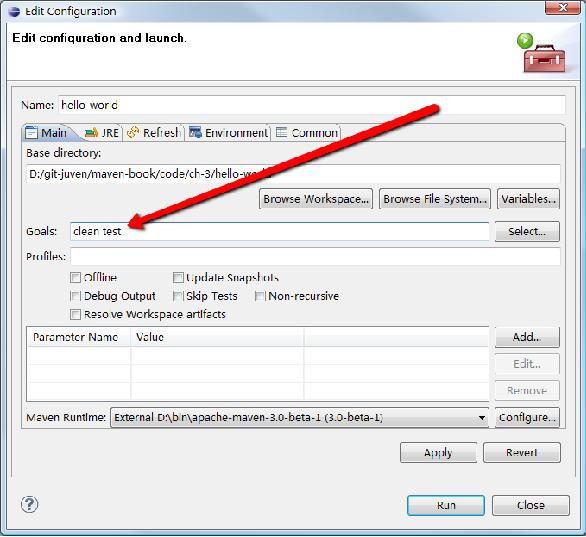


 图3-5 在Eclipse中自定义mvn命令

**3.7 NetBeans Maven插件简单使用** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

NetBeans的Maven插件也十分简单易用，我们可以轻松地在NetBeans中导入现有的Maven项目，或者使用Archetype创建Maven项目，我们也能够在NetBeans中直接运行mvn命令。

**3.7.1 打开Maven项目**

与其说***打开***Maven项目，不如称之为***导入***更为合适，因为这个项目不需要是NetBeans创建的Maven项目，不过这里我们还是遵照NetBeans菜单中使用的名称。

选择菜单栏中的**文件**，然后选择**打开项目**，直接定位到Hello World项目的根目录，NetBeans会十分智能地识别出Maven项目，如图3-6所示：

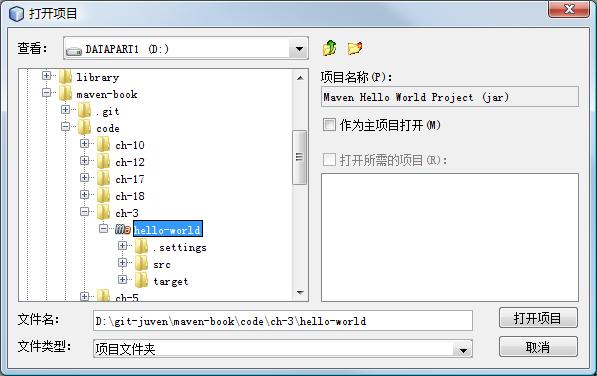


图3-6 在NetBeans中导入Maven项目

Maven项目的图标有别于一般的文件夹，点击**打开项目**后，Hello World项目就会被导入到NetBeans中，在**项目**视图中可以看到如图3-7所示的项目结构：

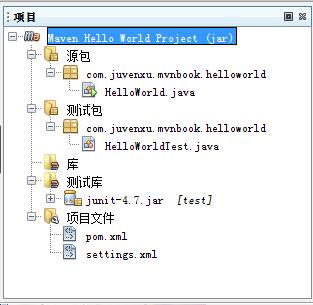


 图3-7 NetBeans中导入的Maven项目结构

NetBeans中项目主代码目录的名称为**源包**，测试代码目录成了**测试包**，编译范围依赖为**库**，测试范围依赖为**测试库**，这里我们也能看到pom.xml，NetBeans甚至还帮我们引用了settings.xml。

**3.7.2 创建Maven项目**

在NetBeans中创建Maven项目同样十分轻松，在菜单栏中选择**文件**，然后**新建项目**，在弹出的对话框中，选择项目类别为**Maven**，项目为**Maven项目**，点击“下一步”之后，对话框会提示我们选择Maven原型（即Maven Archtype），我们选择**Maven快速启动原型（1.0）**，（即前文提到的maven-archetype-quickstart），点击“下一步”之后，输入项目的基本信息，这些信息在之前讨论archetype及在m2eclipse中创建Maven项目的时候都仔细解释过，不再详述，如图3-8所示：

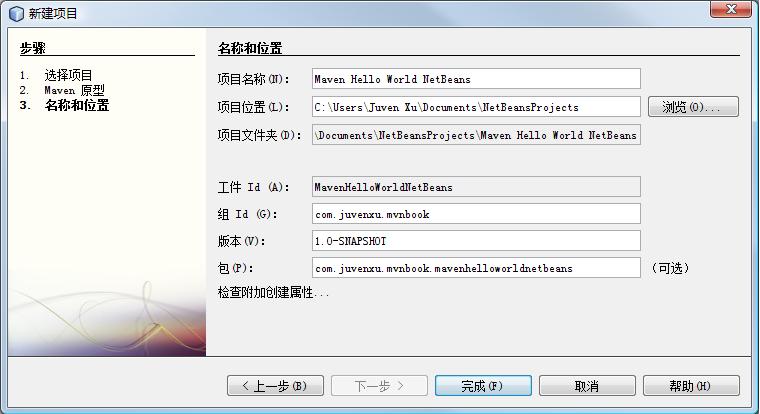


 图3-8 在NetBeans中使用Archetype创建Maven项目

点击完成之后，一个新的Maven项目就创建好了。

**3.7.3 运行mvn命令**

NetBeans在默认情况下提供两种Maven运行方式，点击菜单栏中的**运行**，我们可以看到**生成项目**和**清理并生成项目**两个选项，我们可以尝试“点击运行Maven构建”，根据NetBeans控制台的输出，我们就能发现它们实际上对应了**mvn install**和**mvn clean install**两个命令。

在实际开发过程中，我们往往不会满足于这两种简单的方式，比如，有时候我们只想执行项目的测试，而不需要打包，这时我们就希望能够执行**mvn clean test**命令，所幸的是NetBeans Maven插件完全支持自定义的mvn命令配置。

在菜单栏中选择**工具**，接着选择**选项**，在对话框中，最上面一栏选择**其他**，下面选择**Maven**标签栏，在这里我们可以对NetBeans Maven插件进行全局的配置（还记得第2章中我们如何配置NetBeans使用外部Maven么？）。现在，选择倒数第三行的**编辑全局定制目标定义…**，我们添加一个名为**Maven Test**的操作，执行目标为clean test，暂时不考虑其他配置选项，如图3-9所示：

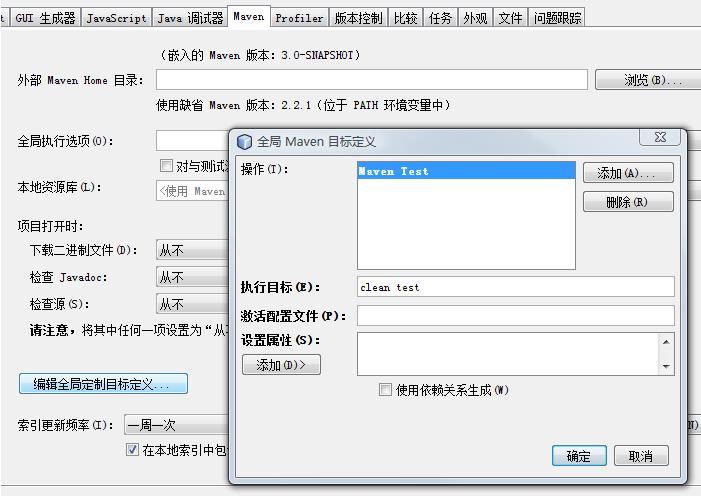


图3-9 在NetBeans中自定义mvn命令

点击“缺省保存该配置”，在Maven项目上右击，选择**定制**，就能看到刚才配置好的Maven运行操作，选择**Maven Test**之后，终端将执行mvn clean test。值得一提的是，我们也可以在项目上右击，选择**定制**，再选择**目标…**再输入想要执行的Maven目标（如clean package)，点击确定之后NetBeans就会执行相应的Maven命令。这种方式十分便捷，但这是临时的，该配置不会被保存，也不会有历史记录。

**3.8 小结** **[Top](http://www.iteye.com/wiki/hzbook/2872-Maven-in-action#top)**

本章以尽可能简单且详细的方式叙述了一个Hello World项目，重点解释了POM的基本内容、Maven项目的基本结构、以及构建项目基本的Maven命令。在此基础上，还介绍了如何使用 Archetype快速创建项目骨架。最后讲述的是如何在Eclipse和NetBeans中导入、创建及构建Maven项目。

分类: [Maven](http://www.cnblogs.com/dcba1112/category/297228.html)

标签: [Maven](http://www.cnblogs.com/dcba1112/tag/Maven/)

27

0

(请您对文章做出评价)

[«](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033798.html) 上一篇：[基于REST架构的Web Service设计](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033798.html)  
[»](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033808.html) 下一篇：[REST Web Service开发实践--Jersey,Google Protocol Buffer, Tomcat结合样例](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033808.html)

posted on 2011-05-01 11:29 [呈顼](http://www.cnblogs.com/dcba1112/) 阅读(175648) 评论(25) [编辑](http://i.cnblogs.com/EditPosts.aspx?postid=2033805) [收藏](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033805.html)

**评论**

[**#1楼**](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033805.html#2145479) **2011-07-07 17:07** [**风树**](http://home.cnblogs.com/u/162757/)

很棒，很多混乱的概念看完一下全清楚了

[支持(0)反对(0)](javascript:void(0);)

[**#2楼**](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033805.html#2312189) **2012-02-20 00:50** [**edhn**](http://home.cnblogs.com/u/377053/)

看完此文我去买了一本《maven实战》

[**#11楼**](http://www.cnblogs.com/dcba1112/archive/2011/05/01/2033805.html#2689395) **2013-05-24 22:57** [**gezi\_mail**](http://home.cnblogs.com/u/532307/)

maven确实 很简单，但是，每次都要编写pom.xml也是很麻烦，比如我要创建一个struts项目，那我就要写很多的依赖（dependency），这些依赖的名 字我还不清楚，很是麻烦，有没有简单快捷的方法，以创建struts项目为例，struts官网有pom.xml文件提供吗？或者告诉我一个最快速的方法