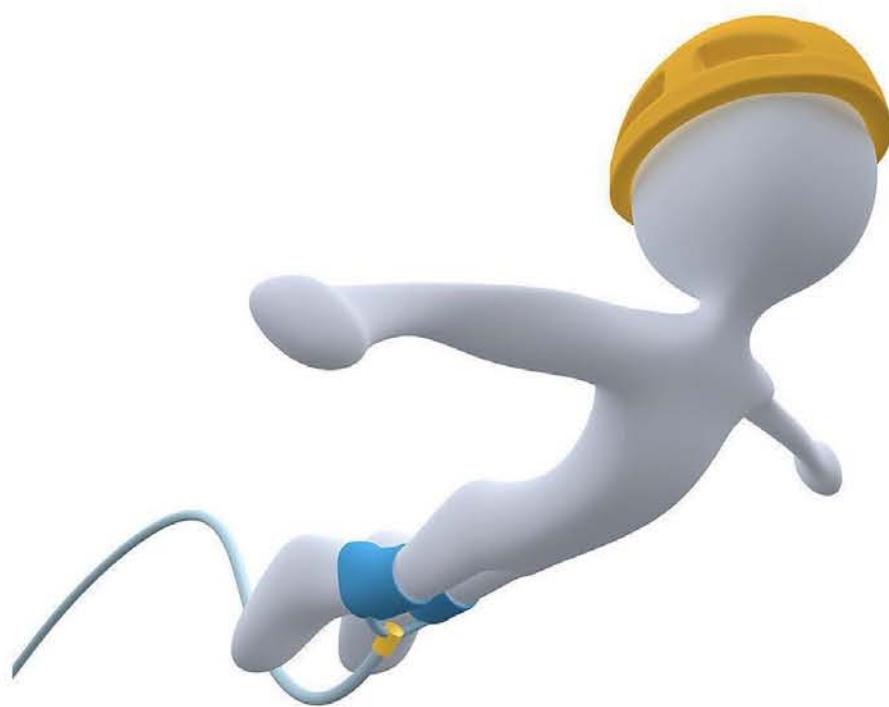


内容全面、实战性强的Maven著作

Maven 实战

基于Maven 3



许晓斌 著

InfoQ企业软件开发丛书

免费在线版本

(非印刷免费在线版)



了解本书更多信息请登录 [本书的官方网站](#)

InfoQ 中文站出品



本书由 InfoQ 中文站免费发放，如果您从其他渠道获取本书，请注册 InfoQ 中文站以支持作者和出版商，并免费下载更多 InfoQ 企业软件开发系列图书。

本迷你书主页为

<http://www.infoq.com/cn/minibooks/maven-in-action>

前言

为什么写这本书？

2007 年的时候，我加入了一个新成立的开发团队，我们一起做一个新的项目。经验较丰富的同事习惯性地开始编写 Ant 脚本，也有人希望能尝试一下 Maven。当时我比较年轻，且富有激情，因此大家决定让我对 Maven 做些研究和实践。于是我慢慢开始学习并推广 Maven，这期间有人支持，也有人抵触，而我则尽力地为大家排除困难，并做一些内部交流，渐渐地，抵触的人越来越少，我的工作也得到了大家的认可。

为什么一开始有人会抵触这一优秀的技术呢？后来我开始反思这一经历，我认为 Maven 陡峭的学习曲线和匮乏的文档是当时最主要的问题。为了能改善这个问题，我开始在博客中撰写各类关于 Maven 的中文博客，翻译了 O'Reilly 出版的《Maven 权威指南》一书，并建立了国内的 Maven 中文社区，不定期地回答各类 Maven 相关问题，这在一定程度上推动了 Maven 这一优秀的技术在国内的传播。

后来我加入了 Maven 之父 Jason Van Zyl 创建的 Sonatype，参与 Nexus 的开发并负责维护 Maven 中央仓库，这些工作使我对开源和 Maven 有了更深的认识，也给了我从头写一本关于 Maven 的书的信心。我希望它能够更贴近国内的技术人员的需求，能够出现在书店的某个角落里，给那些有心发现它的读者带来一丝欣喜。

该书写作后期适逢 Maven 3 的发布，这距离我刚接触 Maven 时已经过去 3 年有余，感叹时光的流逝！Maven 在 2007 年至 2010 年取得了飞速的发展，现在几乎已经成为了所有 Java 开源项目的标配，Struts、Hibernate、Ehcache 等知名的开源项目都使用 Maven 进行管理。据了解，国内也有越来越多的知名的软件公司开始使用 Maven 管理他们的项目，例如阿里巴巴和淘宝。



许晓斌 (Juven Xu)，国内社区公认的 Maven 技术专家、Maven 中文用户组创始人、Maven 技术的先驱和积极推动者。对 Maven 有深刻的认识，实战经验丰富，不仅撰写了大量关于 Maven 的技术文章，而且还翻译了开源书籍《Maven 权威指南》，对 Maven 技术在国内的普及和发展做出了很大的贡献。就职于 Maven 之父的公司，负责维护 Maven 中央仓库，是 Maven 仓库管理器 Nexus（著名开源软件）的核心开发者之一，曾多次受邀到淘宝等大型企业开展 Maven 方面的培训。此外，他还是开源技术的积极倡导者和推动者，擅长 Java 开发和敏捷开发实践。

本书面向的读者

首先，本书适合所有 Java 程序员阅读。由于自动化构建、依赖管理等问题并不只存在于 Java 世界，因此非 Java 程序员也能够从该书中获益。无论你是从未接触过 Maven、还是已经用了 Maven 很长时间，亦或者想要扩展 Maven，都能从本书获得有价值的参考建议。

其次，本书也适合项目经理阅读，它能帮助你更规范、更高效地管理 Java 项目。

本书的主要内容

第 1 章 对 Maven 做了简要介绍，通过一些程序员熟悉的例子介绍了 Maven 是什么，为什么需要 Maven。建议所有读者都阅读以获得一个大局的印象。

第 2~3 章 对 Maven 的一个入门介绍，这些内容对初学者很有帮助，如果你已经比较熟悉 Maven，可以跳过。

第 4 章 介绍了本书使用的背景案例，后面的很多章节都会基于该案例展开，因此建议读者至少简单浏览一遍。

第 5~8 章 深入阐述了 Maven 的核心概念，包括坐标、依赖、仓库、生命周期、插件、继承和多模块聚合，等等，每个知识点都有实际的案例相佐，建议读者仔细阅读。

第 9 章 介绍使用 Nexus 建立私服，如果你要在实际工作中使用 Maven，这是必不可少的。

第 10~16 章 介绍了一些相对高级且离散的知识点，包括测试、持续集成与 Hudson、Web 项目与自动化部署、自动化版本管理、智能适应环境差异的灵活构建、站点生成，以及 Maven 的 Eclipse 插件 m2eclipse，等等。读者可以根据自己实际需要和兴趣选择性地阅读。

第 17~18 章 介绍了如何编写 Archetype 和 Maven 插件。一般的 Maven 用户在实际工作中往往不需要接触这些知识，如果你需要编写插件扩展 Maven，或者需要编写 Archetype 维护自己的项目骨架以方便团队开发，那么可以仔细阅读这两章的内容。

本迷你书摘取了其中的第 2 章——Maven 安装、第 3 章——Maven 使用入门、第 5 章——坐标和依赖、第 12 章——使用 Maven 构建 Web 应用。

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。

专家推荐

随着近两年 Maven 在国内的普及，越来越多的公司与项目开始接受并使用其作为项目构建与依赖管理工具，Java 开发人员对 Maven 相关的资料的需求也越来越迫切。Juven Xu 作为 Sonatype 的员工和《Maven 权威指南》的译者，对 Maven 有着非常深刻的理解，他为 Maven 中文社区所做的工作也为推动 Maven 的发展做出了非常大的贡献。这本书是 Juven 牺牲了将近一年的业余时间创作而成的，内容全面、实战性强、深度和广度兼备，是中文社区不可多得的优秀参考资料。

——Maven 中文社区

本国语言的 Maven 参考资料永远是受欢迎的，而现在 Juven Xu（许晓斌）——一位活跃在开源社区的知名 Maven 专家正好有条件编写一本关于 Maven 的中文图书。他的新书《Maven 实战》将带领你一步步从认识 Maven 开始走向更高级的现实世界中的真实项目应用。这本书的主要内容不仅包括 Maven 在 Web 领域的应用、使用 Maven 管理版本发布、以及如何编写自己的 Maven 插件，而且还涵盖了许多如何在企业环境中应用 Maven 的技术细节，例如 Eclipse 集成、Nexus 仓库管理器以及用 Hudson 进行持续集成等。如果你是一个正在使用 Maven 的中国程序员，该书是必备的！

——John Smart Wakaleo Consulting 首席咨询顾问，

《Java Power Tools》（O'Reilly 出版）作者

内容简介

本书由国内社区公认的 Maven 专家 Juven Xu 亲自执笔，内容的权威性毋庸置疑。

本书是国内第一本公开出版的 Maven 专著。它内容新颖，基于最新发布的 Maven 3.0，不仅详尽讲解了 Maven 3.0 的所有新功能和特性，而且还将这些新功能和特性与 Maven 2.x 版本进行了对比，以便于正在使用 Maven 2.x 版本的用户能更好地理解。本书它内容全面，以从专家的角度阐释 Maven 的价值开篇，全面介绍了 Maven 的安装、配置和基本使用方法，以便于初学者参考；详细讲解了坐标和依赖、Maven 仓库、生命周期和插件、聚合与继承等 Maven 的核心概念，建议所有读者仔细阅读；系统性地阐述了使用 Nexus 建立私服、使用 Maven 进行测试、使用 Hudson 进行持续集成、使用 Maven 构建 Web 应用、Maven 的版本管理、Maven 的灵活构建、生成项目站点和 Maven 的 m2eclipse 插件等实用性较强的高级知识，读者可有选择性的阅读；扩展性地讲解了如何 Maven 和 Archetype 插件，这部分内容对需要编写插件扩展 Maven 或需要编写 Archetype 维护自己的项目骨架以更便于团队开发的读者来说尤为有帮助。它实战性强，不仅绝大部分知识点都有相应的案例，而且本书还在第 4 章设计了一个背景案例，后面的很多章节都是围绕这个案例展开的，可操作性极强。

本书适合所有 Java 程序员阅读，无论你是从未使用过 Maven，亦或是已经使用 Maven 很长一段时间了，相信你都能从本书中获得有价值的参考。本书也适合所有项目经理阅读，它能帮助你更规范、更高效地管理 Java 项目。

你是否早已厌倦了日复一日的手工构建工作？你是否对各个项目风格迥异的构建系统感到恐惧？Maven——这一 Java 社区事实标准的项目管理工具，能帮你从琐碎的手工劳动中解脱出来，帮你规范整个组织的构建系统。不仅如此，它还有依赖管理、自动生成项目站点等超酷的特性，已经有无数的开源项目使用它来构建项目并促进团队交流，每天都有数以万计的开发者在访问中央仓库以获取他们需要的依赖。

本书内容全面而系统，Maven 的原理、使用方法和高级应用尽含其中；注重实战是本书的另一个特点，不仅在各个知识点都辅有大量的小案例，而且还有一个综合性的案例贯穿全书。如果你想使用 Maven，或正在使用 Maven，本书将给你绝佳的指导。



我们的**使命**：成为关注软件开发领域变化和创新的专业网站

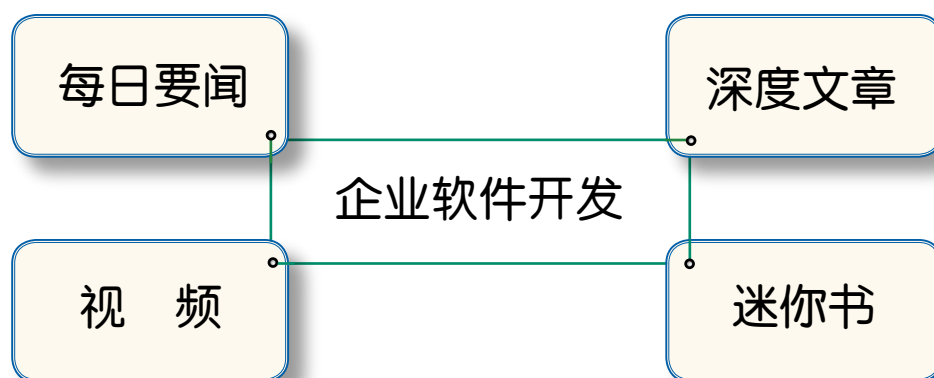
我们的**定位**：关注高级决策人员和大中型企业

我们的**社区**：Java、.NET、Ruby、SOA、Agile、Architecture

我们的**特质**：个性化RSS定制、国际化内容同步更新

我们的**团队**：超过30位领域专家担当社区编辑

.....



目录

前言	I
本书面向的读者	II
致谢	III
专家推荐	IV
内容简介	V
第 1 章 MAVEN 的安装和配置	- 1 -
1.1 在WINDOWS上安装MAVEN	- 1 -
1.1.1 检查JDK安装.....	- 1 -
1.1.2 下载Maven	- 1 -
1.1.3 本地安装.....	- 2 -
1.1.4 升级Maven	- 3 -
1.2 在基于UNIX的系统上安装MAVEN	- 3 -
1.2.1 下载和安装.....	- 4 -
1.2.2 升级Maven	- 5 -
1.3 安装目录分析.....	- 5 -
1.3.1 M2_HOME.....	- 5 -
1.3.2 ~/.m2.....	- 7 -
1.4 设置HTTP代理	- 7 -
1.5 安装M2ECLIPSE	- 8 -
1.6 安装NETBEANS MAVEN插件.....	- 13 -
1.7 MAVEN安装最佳实践	- 15 -
1.7.1 设置MAVEN_OPTS环境变量.....	- 15 -
1.7.2 配置用户范围settings.xml	- 16 -
1.7.3 不要使用IDE内嵌的Maven.....	- 16 -
1.8 小结.....	- 18 -
第 2 章 MAVEN使用入门	- 19 -
2.1 编写POM.....	- 19 -
2.2 编写主代码.....	- 20 -

2.3 编写测试代码.....	- 22 -
2.4 打包和运行.....	- 26 -
2.5 使用ARCHETYPE生成项目骨架	- 28 -
2.6 M2ECLIPSE简单使用	- 30 -
2.6.1 导入Maven项目.....	- 30 -
2.6.2 创建Maven项目.....	- 31 -
2.6.3 运行mvn命令.....	- 32 -
2.7 NETBEANS MAVEN插件简单使用.....	- 33 -
2.7.1 打开Maven项目.....	- 33 -
2.7.2 创建Maven项目.....	- 35 -
2.7.3 运行mvn命令.....	- 35 -
2.8 小结.....	- 36 -
第 3 章 坐标和依赖	- 37 -
3.1 何为MAVEN坐标	- 37 -
3.2 坐标详解.....	- 38 -
3.3 ACCOUNT-MAIL.....	- 40 -
3.3.1 account-email 的POM	- 40 -
3.3.2 account-email的主代码	- 42 -
3.3.3 account-email的测试代码	- 46 -
3.3.4 构建account-email	- 48 -
3.4 依赖的配置.....	- 48 -
3.5 依赖范围.....	- 49 -
3.6 传递性依赖.....	- 51 -
3.6.1 何为传递性依赖.....	- 51 -
3.6.2 传递性依赖和依赖范围.....	- 52 -
3.7 依赖调解 (DEPENDENCY MEDIATION)	- 53 -
3.8 可选依赖.....	- 53 -
3.9 最佳实践.....	- 55 -
3.9.1 排除依赖.....	- 56 -
3.9.2 归类依赖.....	- 57 -
3.9.3 优化依赖.....	- 59 -
3.10 小结.....	- 61 -
第 4 章 使用MAVEN构建WEB应用	- 62 -

4.1 WEB项目的目录结构	- 62 -
4.2 ACCOUNT-SERVICE	- 64 -
4.2.1 account-service的POM	- 65 -
4.2.2 account-service的主代码	- 66 -
4.3 ACCOUNT-WEB	- 71 -
4.3.1 account-web的POM	- 71 -
4.3.2 account-web的主代码	- 73 -
4.4 使用JETTY-MAVEN-PLUGIN进行测试	- 80 -
4.5 使用CARGO实现自动化部署	- 82 -
4.5.1 部署至本地Web容器	- 83 -
4.5.2 部署至远程Web容器	- 85 -
4.6 小结	- 86 -

第 1 章 Maven 的安装和配置

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。

1.1 在 Windows 上安装 Maven

1.1.1 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 1-1 所示：

```
C:\Users\Juven Xu>echo %JAVA_HOME%  
D:\java\jdk1.6.0_07  
  
C:\Users\Juven Xu>java -version  
java version "1.6.0_07"  
Java(TM) SE Runtime Environment (build 1.6.0_07-b06)  
Java HotSpot(TM) Client VM (build 10.0-b23, mixed mode, sharing)
```

图 1-1 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 1.1.3 节。

1.1.2 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！

1.1.3 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 1-2 所示：



图 1-2 Windows 中系统环境变量配置

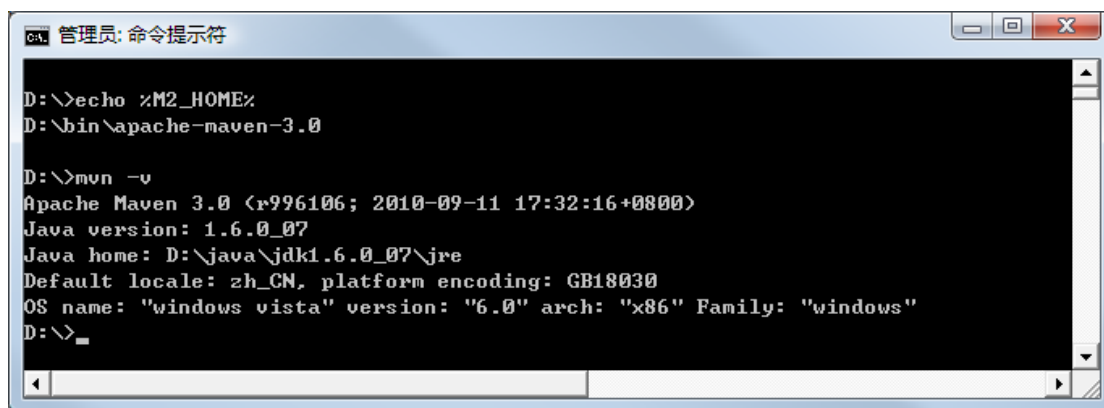
这里需要提一下的是 `Path` 环境变量，当我们在 `cmd` 中输入命令时，Windows 首先会在当前目录中寻找可执行文件或脚本，如果没有找到，Windows 会接着遍历环境变量 `Path` 中定义的路径。由于我们将 `%M2_HOME%\bin` 添加到了 `Path` 中，而这里 `%M2_HOME%` 实际上

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 1-3 所示：



```
管理员: 命令提示符  
D:\>echo %M2_HOME%  
D:\bin\apache-maven-3.0  
  
D:\>mvn -v  
Apache Maven 3.0 (r996106; 2010-09-11 17:32:16+0800)  
Java version: 1.6.0_07  
Java home: D:\java\jdk1.6.0_07\jre  
Default locale: zh_CN, platform encoding: GB18030  
OS name: "windows vista" version: "6.0" arch: "x86" Family: "windows"  
D:\>
```

图 1-3 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。

1.1.4 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。

1.2 在基于Unix的系统上安装Maven

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。

1.2.1 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 1-4 所示：

```
juven@juven-ubuntu:~$ echo $JAVA_HOME
/usr/local/jdk1.6.0_11
juven@juven-ubuntu:~$ java -version
java version "1.6.0_11"
Java(TM) SE Runtime Environment (build 1.6.0_11-b03)
Java HotSpot(TM) Server VM (build 11.0-b16, mixed mode)
```

图 1-4 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 1-5 所示：


```
[juven@sonatype02 bin]$ echo $M2_HOME
/home/juven/bin/apache-maven
[juven@sonatype02 bin]$ mvn -v
Apache Maven 3.0 (r996106; 2010-09-11 04:32:16-0500)
Java version: 1.6.0_14
Java home: /opt/java/sdk/Sun/jdk1.6.0_14/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux" version: "2.6.18-128.1.1.el5" arch: "i386" Family: "unix"
[juven@sonatype02 bin]$
```

图 1-5 Linux 中检查 Maven 安装

1.2.2 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。

1.3 安装目录分析

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。

1.3.1 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。

1.3.2 ~/.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（注：这里指原书的第 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。

1.4 设置HTTP代理

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。

1.5 安装m2eclipse

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 1-6 所示的界面：

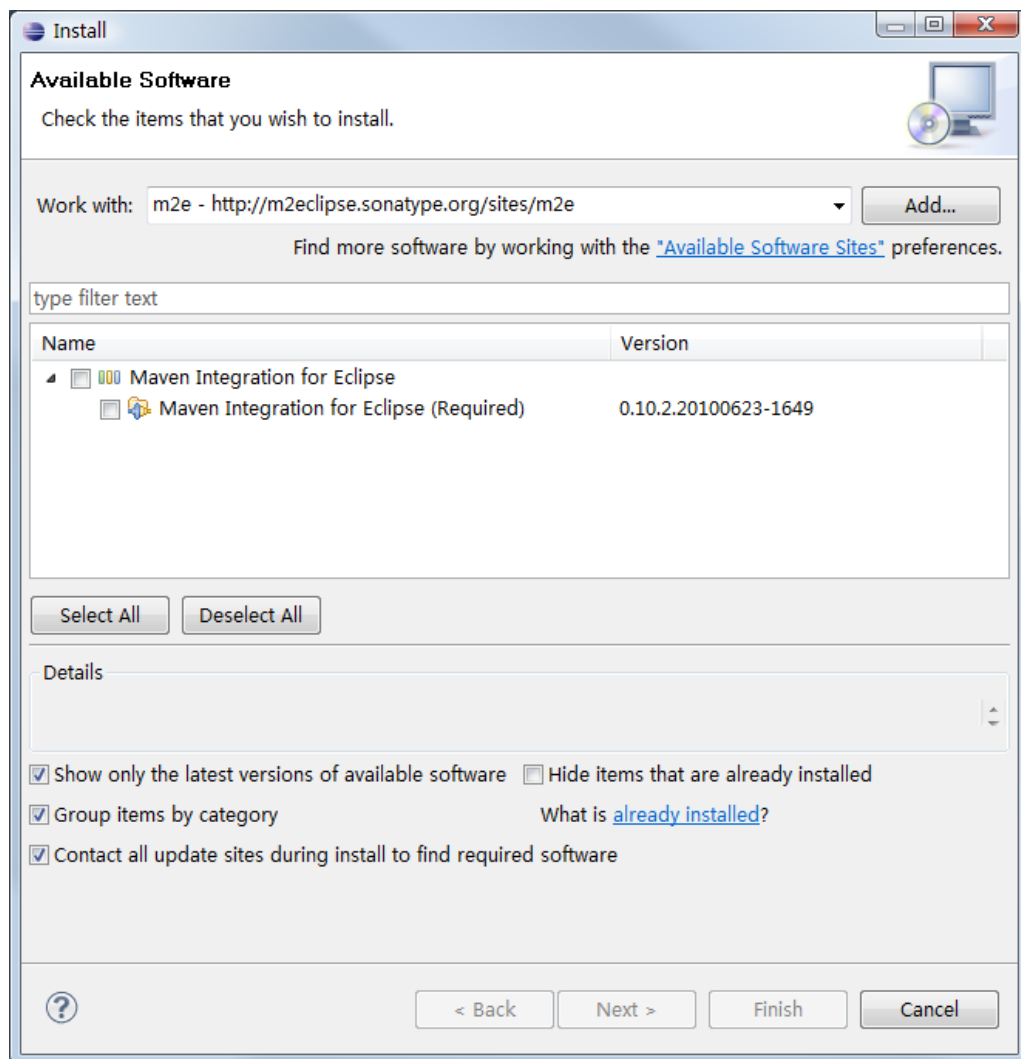


图 1-6 m2eclipse 的核心安装资源列表

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 1-7 所示：

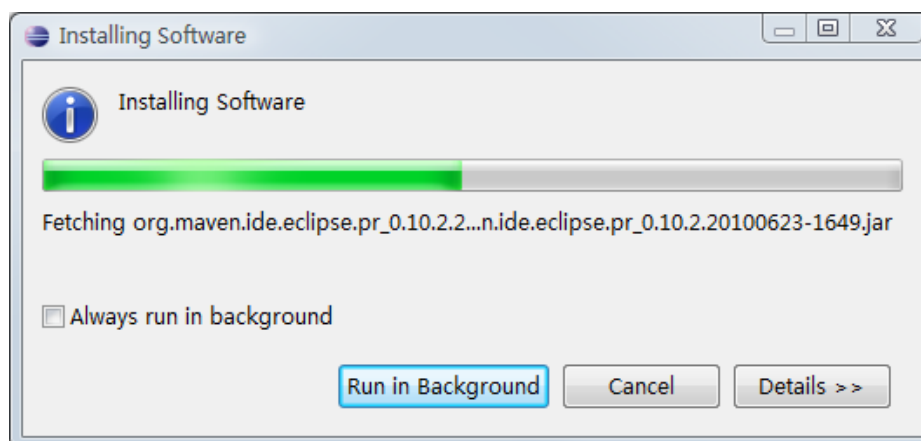


图 1-7：m2eclipse 安装进度

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 1-8 的组件列表：

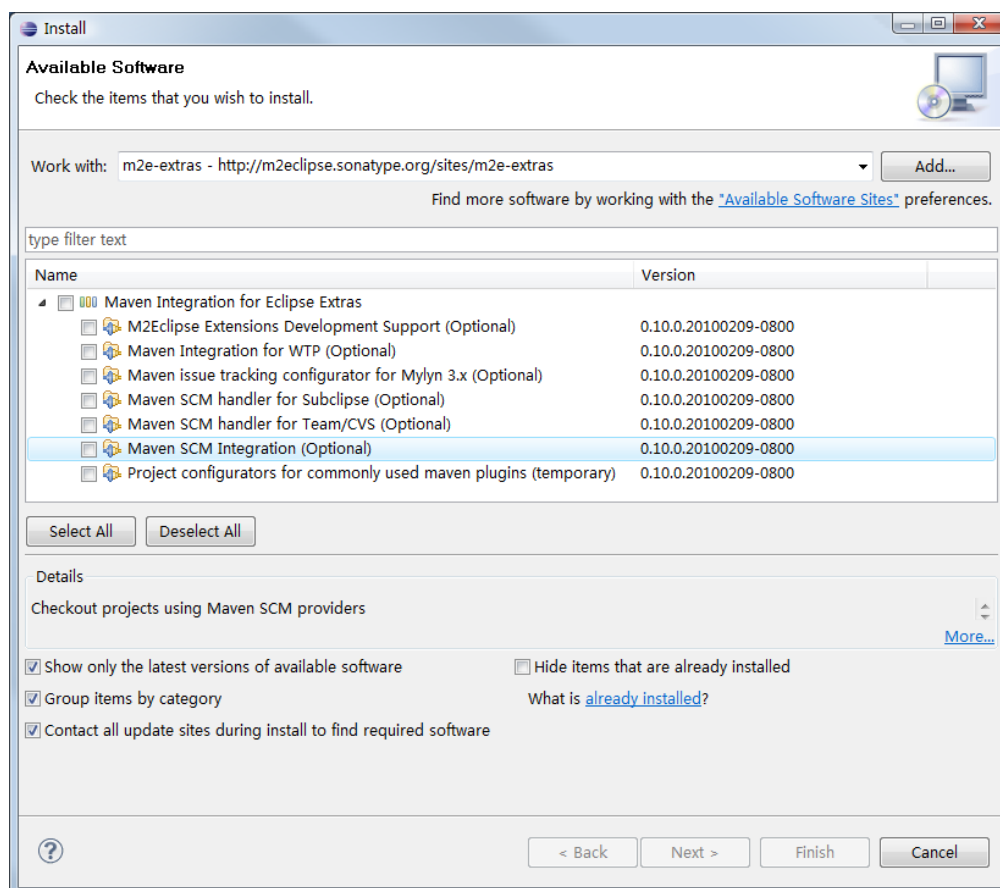


图 1-8：m2eclipse 的额外组件安装资源列表

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，不过前提是首先需要安装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 1-9 所示：

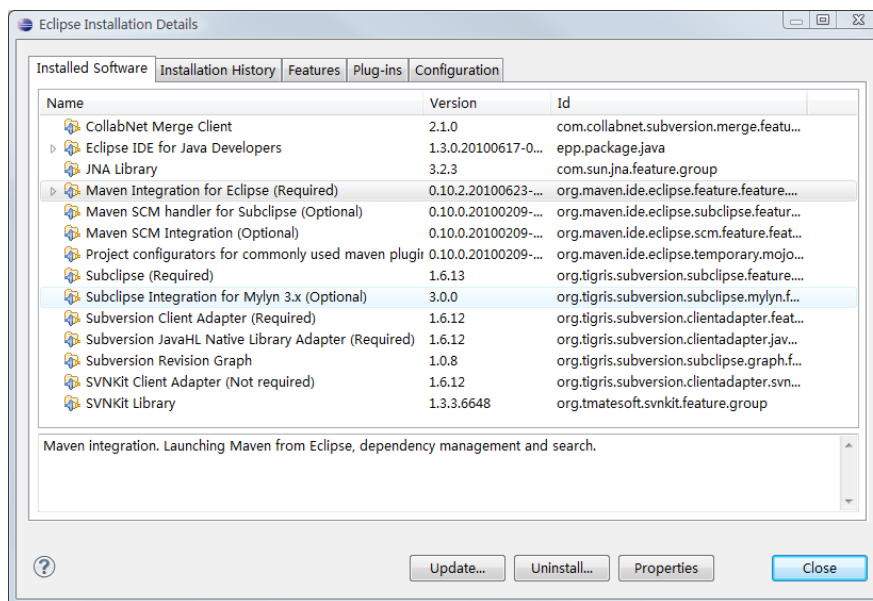


图 1-9m2eclipse 安装结果

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 1-10 所示的对话框：

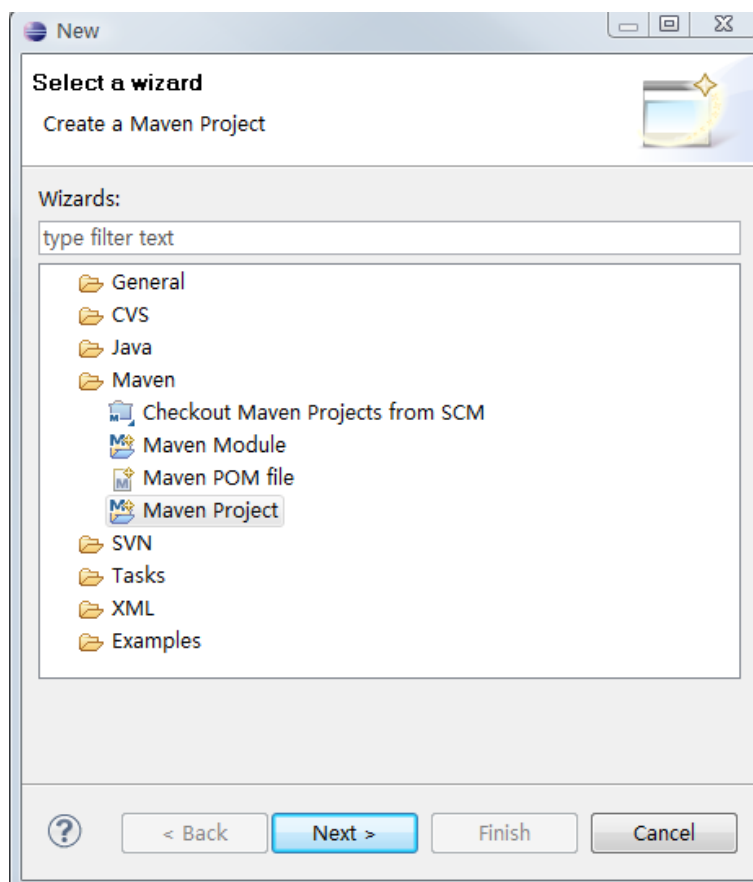


图 1-10 Eclipse 中创建 Maven 项目向导

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
```

1.6 安装NetBeans Maven插件

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 1-11 所示：

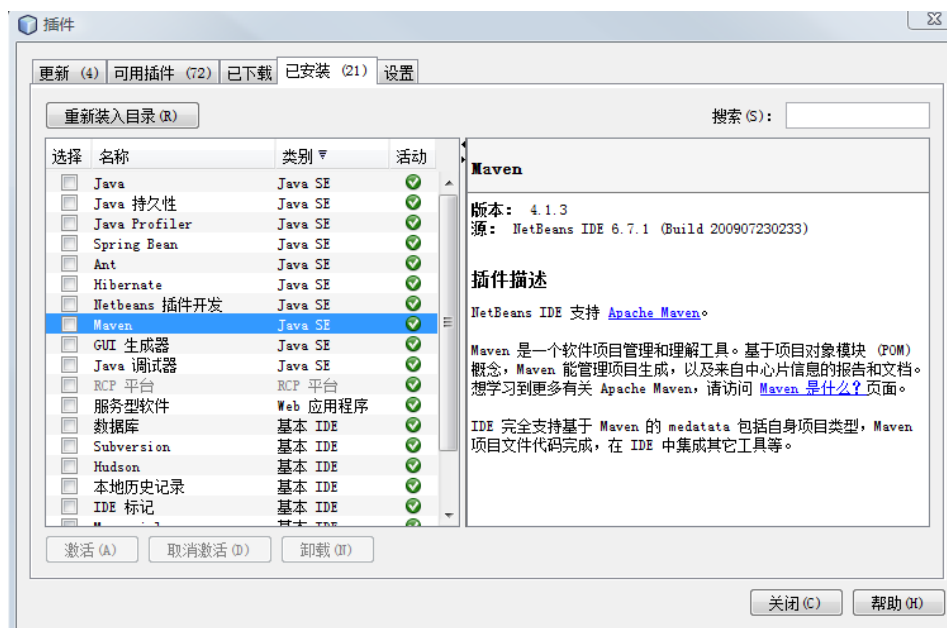


图 1-11 已安装的 NetBeans Maven 插件

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 1-12 所示：



图 1-12 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 1-13 所示：



图 1-13 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。

1.7 Maven安装最佳实践

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。

1.7.1 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.OutOfMemoryError`。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。

1.7.2 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。

1.7.3 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 1-14 所示：

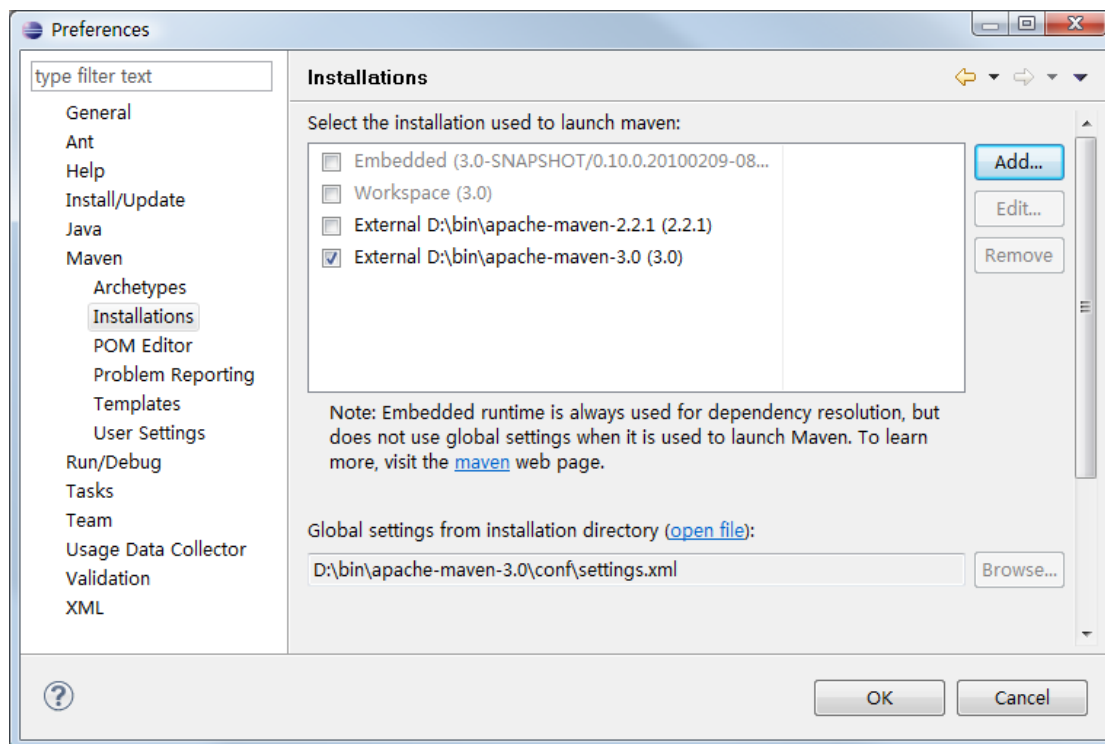


图 1-14 在 Eclipse 中使用外部 Maven

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 1-15 所示的配置：

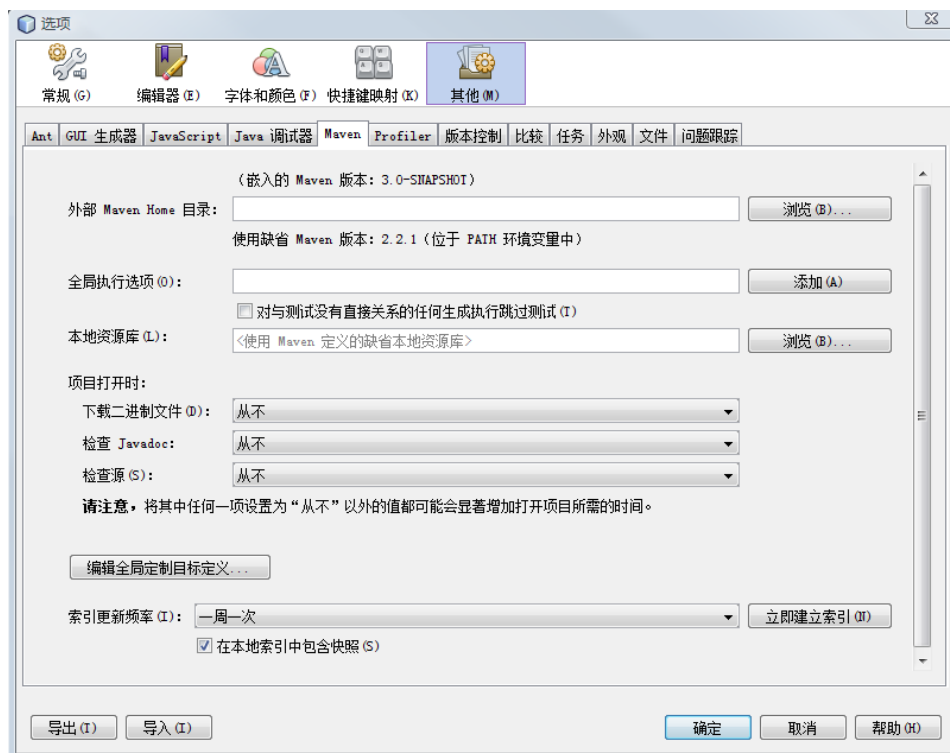


图 1-15 在 NetBeans 中使用外部 Maven

1.8 小结

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 2 章 Maven 使用入门

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。

2.1 编写 POM

就像 Make 的 Makefile，Ant 的 build.xml 一样，Maven 项目的核心是 pom.xml。POM（Project Object Model，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 2-1：

代码清单 2-1：Hello World 的 POM

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>hello-world</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Hello World Project</name>
</project>
```

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，我们为此 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（注：这里指原书的 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。

2.2 编写主代码

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 2-2：

代码清单 2-2：Hello World 的主代码

```
package com.juvenxu.mvnbook.helloworld;
```

```
public class HelloWorld
{
    public String sayHello()
    {
        return "Hello Maven";
    }

    public static void main(String[] args)
    {
        System.out.print( new HelloWorld().sayHello() );
    }
}
```

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：

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Maven Hello World Project
[INFO]   task-segment: [clean, compile]
[INFO] -----
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory D:\code\hello-world\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] skip non existing resourceDirectory D:\code\hello-world\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\classes
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Fri Oct 09 02:08:09 CST 2009
[INFO] Final Memory: 9M/16M
[INFO] -----
```

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。

2.3 编写测试代码

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 2-3：

代码清单 2-3：为 Hello World 的 POM 添加依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>hello-world</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Hello World Project</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.7</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</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（注：这里指原书的第 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 2-4：

代码清单 2-4：Hello World 的测试代码

```
package com.juvenxu.mvnbook.helloworld;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class HelloWorldTest
{
    @Test
    public void testSayHello()
    {
        HelloWorld helloWorld = new HelloWorld();

        String result = helloWorld.sayHello();

        assertEquals( "Hello Maven", result );
    }
}
```

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**：

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Maven Hello World Project
[INFO]   task-segment: [clean, test]
[INFO] -----
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory D:\git-juven\mvnbook\code\hello-world\target
[INFO] [resources:resources {execution: default-resources}]
...
Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.pom
1K downloaded (junit-4.7.pom)
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
...
Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.jar
226K downloaded (junit-4.7.jar)
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\test-classes
[INFO] -----
[ERROR] BUILD FAILURE
[INFO] -----
[INFO] Compilation failure
D:\code\hello-
world\src\test\java\com\juvenxu\mvnbook\helloworld\HelloWorldTest.java:[8,5] -source 1.3
中不支持注释
( 请使用 -source 5 或更高版本以启用注释)
@Test
[INFO] -----
[INFO] For more information, run Maven with the -e switch
...
```

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 2-5：

代码清单 2-5：配置 `maven-compiler-plugin` 支持 Java 5

```
<project>
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
...
</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：

```
...
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: D:\code\hello-world\target\surefire-reports
-----
T E S T S
-----
Running com.juvenxu.mvnbook.helloworld.HelloWorldTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.055 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
```



```
...
```

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。

2.4 打包和运行

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：

```
...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
...
```

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步骤，执行 `mvn clean install`：

```
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
[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
[INFO] -----
[INFO] BUILD SUCCESSFUL
...
```

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：

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>1.2.1</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
<configuration>
<transformers>
<transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
<mainClass>com.juvenxu.mvnbook.helloworld.HelloWorld</mainClass>
</transformer>
</transformers>
</configuration>
</execution>
</executions>
</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。

2.5 使用 Archetype 生成项目骨架

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：

```
Define value for groupId: : com.juvenxu.mvnbook
Define value for artifactId: : hello-world
Define value for version: 1.0-SNAPSHOT: :
Define value for package: com.juvenxu.mvnbook: : com.juvenxu.mvnbook.helloworld
Confirm properties configuration:
groupId: com.juvenxu.mvnbook
artifactId: hello-world
version: 1.0-SNAPSHOT
package: com.juvenxu.mvnbook.helloworld
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。

2.6 m2eclipse简单使用

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。

2.6.1 导入Maven项目

第 2 章介绍了如何安装 m2eclipse，现在，我们使用 m2eclipse 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 2-1 所示：

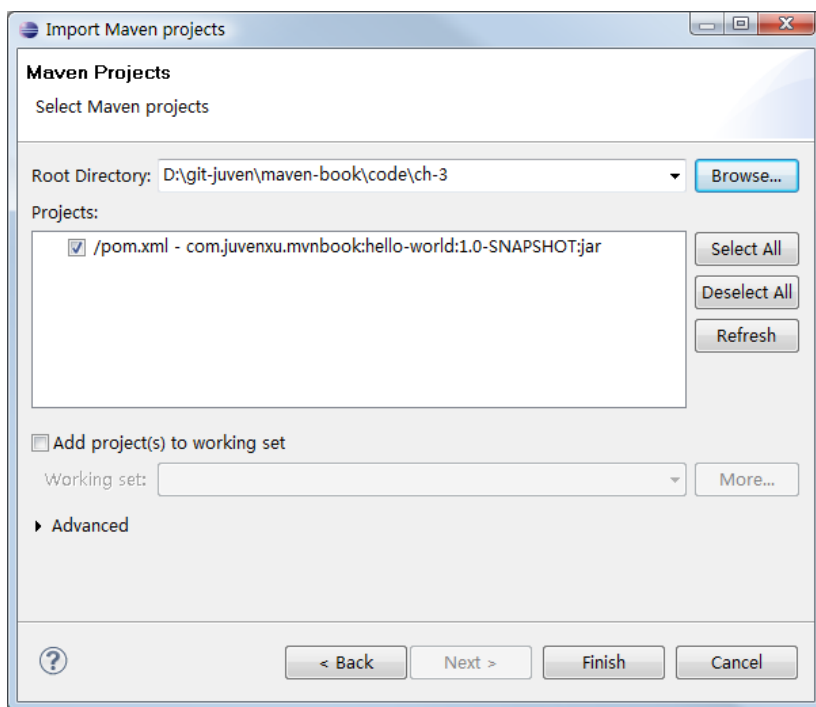


图 2-1 在 Eclipse 中导入 Maven 项目

点击 **Finish** 之后，m2eclipse 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 2-2 所示的项目结构：

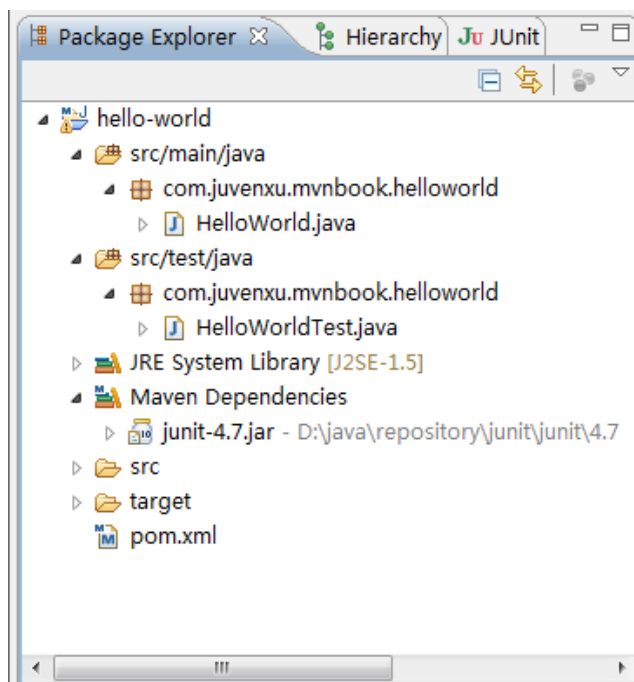


图 2-2 Eclipse 中导入的 Maven 项目结构

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）。

2.6.2 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 2-3 所示：

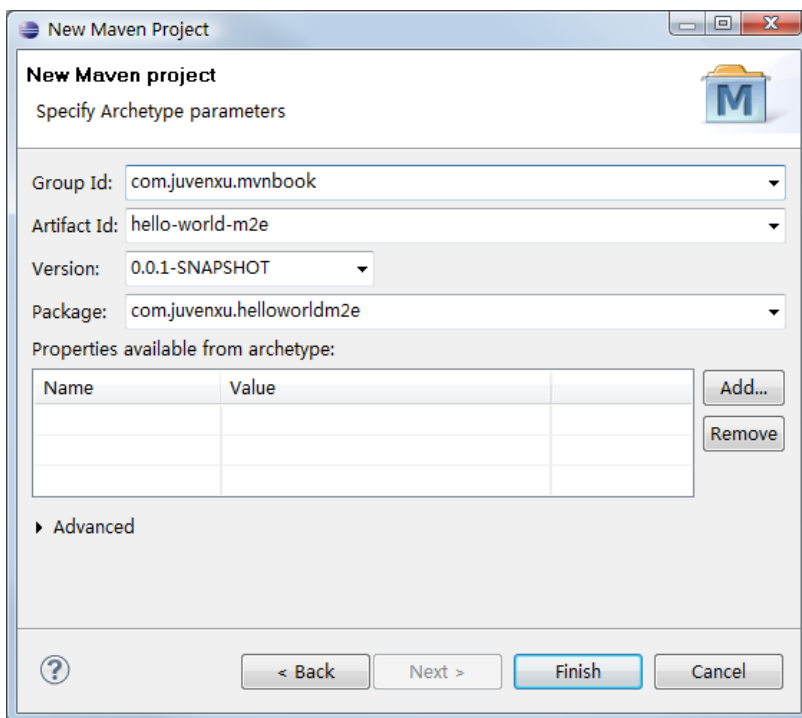


图 2-3 在 Eclipse 中使用 Archetype 创建项目

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。

2.6.3 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 2-4 所示：

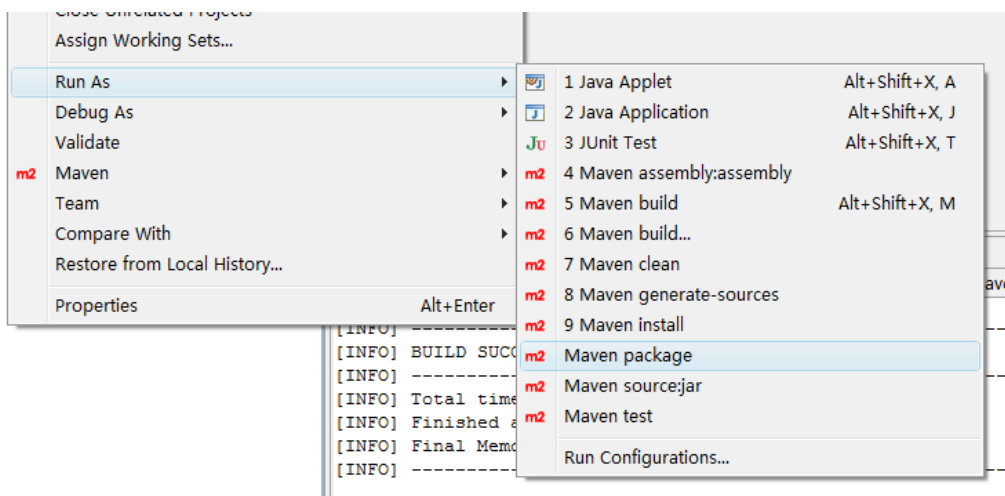


图 2-4 在 Eclipse 中运行默认 mvn 命令

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 build...** 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 2-5 就是自定义 Maven 运行命令的界面：

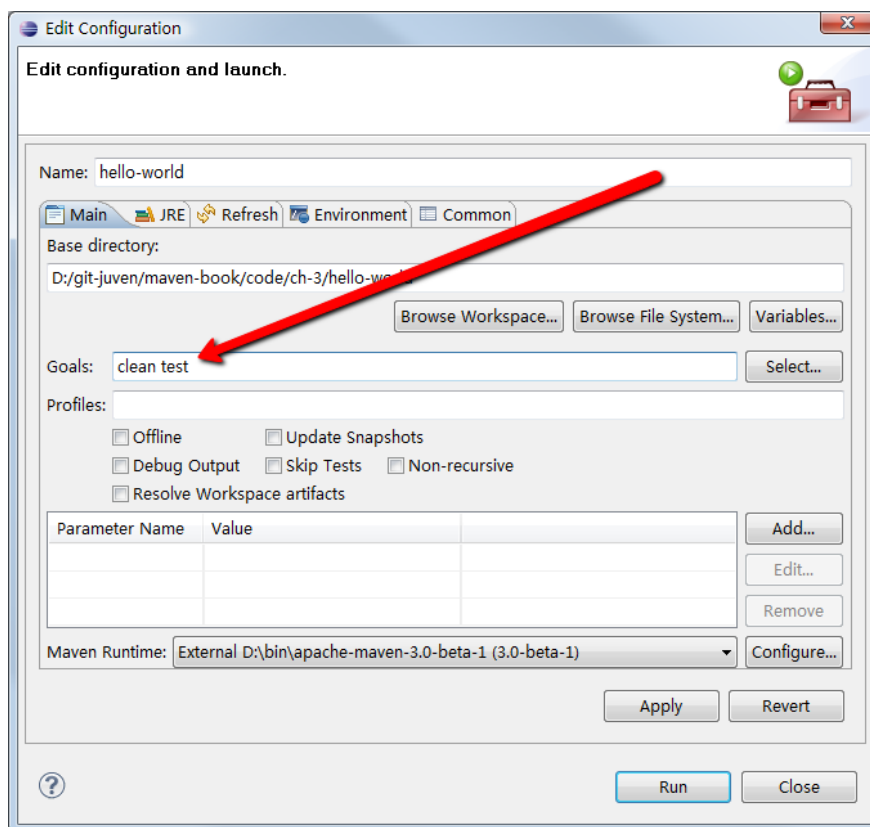


图 2-5 在 Eclipse 中自定义 mvn 命令

2.7 NetBeans Maven 插件简单使用

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。

2.7.1 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 2-6 所示：

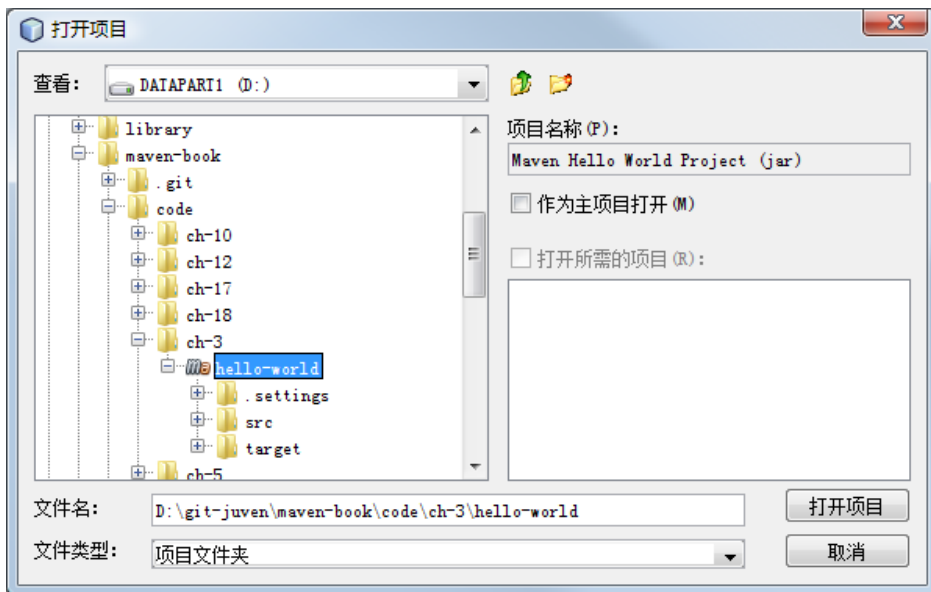


图 2-6 在 NetBeans 中导入 Maven 项目

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 2-7 所示的项目结构：

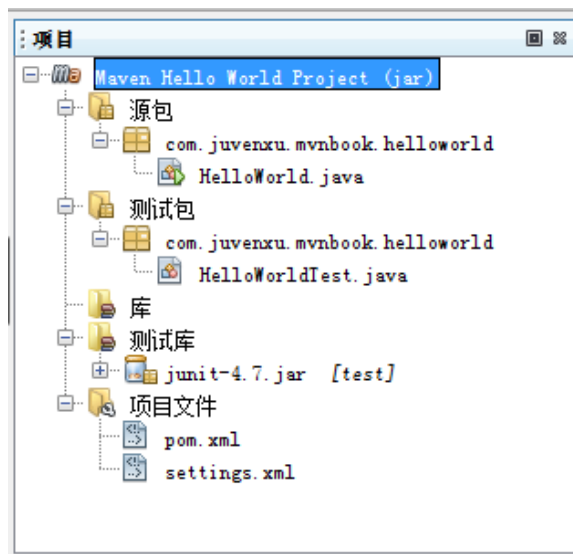


图 2-7 NetBeans 中导入的 Maven 项目结构

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。

2.7.2 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 2-8 所示：

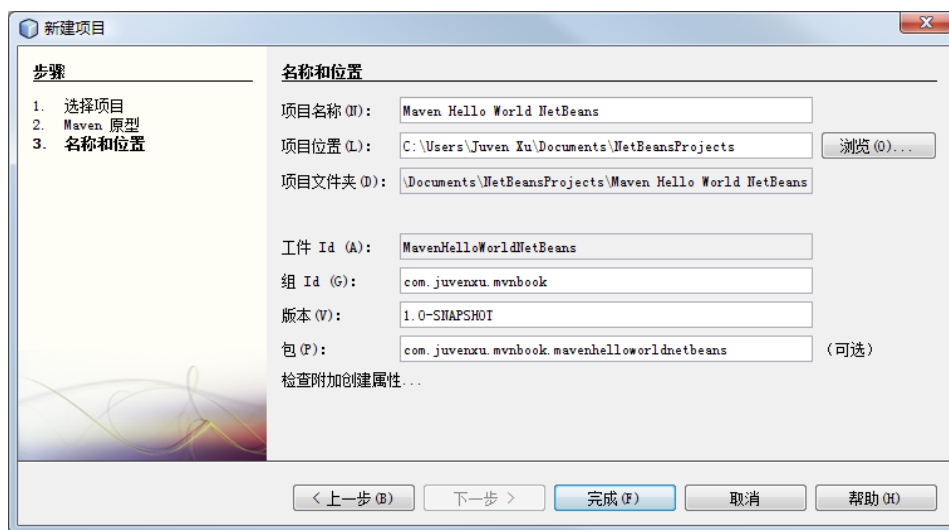


图 2-8 在 NetBeans 中使用 Archetype 创建 Maven 项目

点击完成之后，一个新的 Maven 项目就创建好了。

2.7.3 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 2-9 所示：

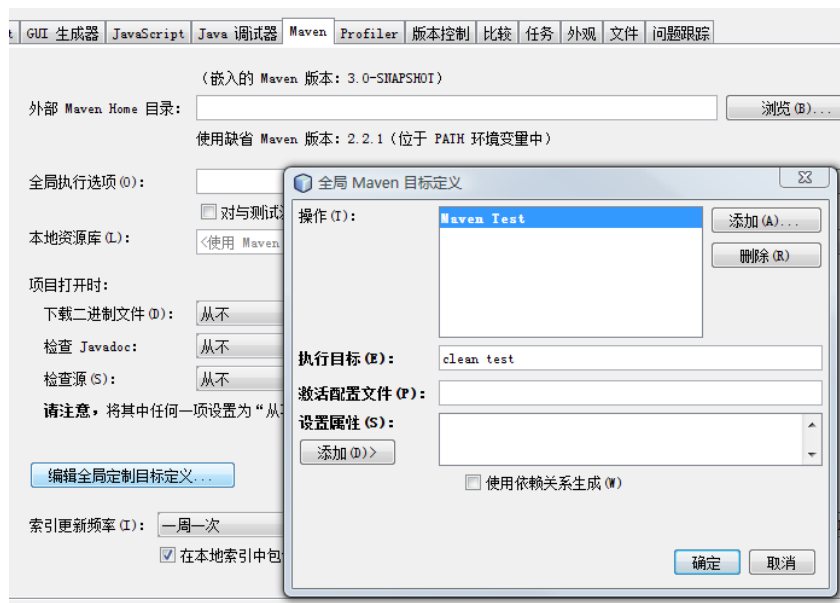


图 2-9 在 NetBeans 中自定义 mvn 命令

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。

2.8 小结

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。



Java — .NET — Ruby — SOA — Agile — Architecture

Java社区：企业Java社区的**变化与创新**

.NET社区：.NET和微软的其它**企业软件开发**解决方案

Ruby社区：面向Web和企业开发的Ruby，主要关注**Ruby on Rails**

SOA社区：关于大中型企业内**面向服务架构**的一切

Agile社区：敏捷软件开发和**项目经理**

Architecture社区：设计、技术趋势及**架构师**所感兴趣的话题

第 3 章 坐标和依赖

正如本书第 1 章所述，Maven 的一大功能是管理项目依赖。为了能自动化地解析任何一个 Java 构件，Maven 就必须将它们唯一标识，这就依赖管理的底层基础——坐标。本章将详细分析 Maven 坐标的作用，解释其每一个元素；在此基础上，再介绍如何配置 Maven，以帮助我们管理项目依赖以及相关的经验和技巧。

3.1 何为Maven坐标

关于坐标（Coordinate），大家最熟悉的定义应该来自于平面几何，在一个平面坐标系中，坐标(x,y)表示该平面上与 x 轴距离为 y，与 y 轴距离为 x 的一点，任何一个坐标都能够唯一标识该平面中的一点。

在实际生活中，我们也可以将地址看成是一种坐标。省、市、区、街道……等一系列信息同样可以唯一标识城市中的任一居住地址和工作地址。邮局和快递公司正是基于这样一种坐标进行日常工作的。

对应于平面中的点和城市中的地址，Maven的世界中拥有数量非常巨大的构件，也就是大家平时用的一些jar、war等文件。在Maven为这些构件引入坐标概念之前，我们无法使用任何一种方式来唯一标识所有这些构件，因此，当需要用到Spring Framework依赖的时候，大家会去Spring Framework网站寻找，当需要用到log4j依赖的时候，大家又会去Apache网站寻找。又因为各个项目的网站风格迥异，大量的时间花费在了搜索、浏览网页等工作上面。没有统一的规范、统一的法则，该工作就无法自动化。重复地搜索、浏览网页和下载类似的jar文件，这本就应该交给机器来做。而机器工作必须基于预定义的规则，Maven定义了这样一组规则：世界上任何一个构件都可以使用Maven坐标唯一标识，Maven坐标的元素包括groupId、artifactId、version、packaging、classifier。现在，只要我们提供正确的坐标元素，Maven就能找到对应的构件，比如说，当需要使用Java5 平台上TestNG的 5.8 版本时，就告诉Maven："groupId=org.testng; artifactId=testng; version=5.8; classifier=jdk15"，Maven就会从仓库中寻找相应的构件供我们使用。也许你会奇怪，"Maven是从哪里下载构件的呢？"答案其实很简单，Maven内置了一个中央仓库的地址（<http://repo1.maven.org/maven2>），该中央仓库包含了世界上大部分流行的开源项目构件，Maven会在需要的时候去那里下载。

在我们开发自己项目的时候，也需要为其定义适当的坐标，这是 Maven 强制要求的，在这个基础上，其他 Maven 项目才能引用该项目生成的构件。见图 3-1。

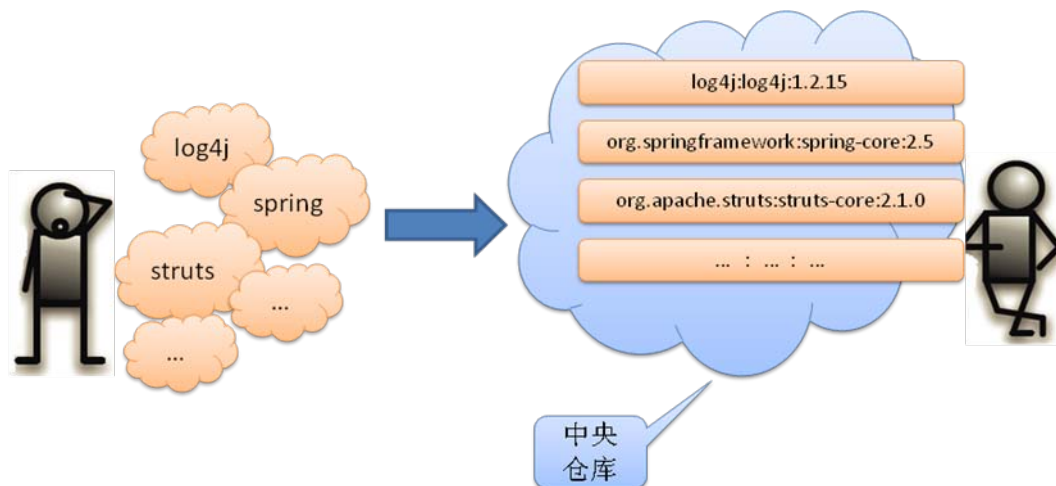


图 3-1 坐标为构件引入秩序

3.2 坐标详解

Maven 坐标为各种构件引入了秩序，任何一个构件都必须明确定义自己的坐标，而一组 Maven 坐标是通过一些元素定义的，它们是：groupId、artifactId、version、packaging、classifier。让我们先看一组坐标定义，如下：

```
<groupId>org.sonatype.nexus</groupId>
<artifactId>nexus-indexer</artifactId>
<version>2.0.0</version>
<packaging>jar</packaging>
```

这是 nexus-indexer 的坐标定义，nexus-indexer 是一个对 Maven 仓库编纂索引并提供搜索功能的类库，它是 Nexus 项目的一个子模块。本书后面会详细介绍 Nexus。上述代码片段中，其坐标分别为：groupId:org.sonatype.nexus、artifactId:nexus-indexer、version:2.0.0、packaging:jar，没有 classifier。下面详细解释一下各个坐标元素：

❑ **groupId**：定义当前 Maven 项目隶属的实际项目。首先，Maven 项目和实际项目不一定是一对一的关系，比如 SpringFramework 这一实际项目，其对应的 Maven 项目会有很多，如 spring-core，spring-context 等等。这是由于 Maven 中模块的概念，因此，一个实际项目往往会被划分成很多模块。其次，groupId 不应该对应项目隶属的组织或公司，原因很简单，一个组织下会有很多实际项目，如果 groupId 只定义到组织级别，而后面我们会看到，artifactId 只能定义 Maven 项目（模块），那么实际项目这个层将难以定义。最后，groupId 的表示方式与 Java 包名的表示方式类似，通常与域名反向——对应。上例中，groupId 为 org.sonatype.nexus，org.sonatype 表示 Sonatype 公司建立的一个非盈利性组织，nexus 表示 Nexus 这一实际项目，该 groupId 与域名 nexus.sonatype.org 对应。

- ❑ **artifactId**：该元素定义实际项目中的一个 Maven 项目（模块），推荐的做法是使用实际项目名称作为 artifactId 的前缀，比如上例中的 artifactId 是 nexus-indexer，使用了实际项目名 nexus 作为前缀，这样做的好处是方便寻找实际构件。在默认情况下，Maven 生成的构件，其文件名会以 artifactId 作为开头，如 nexus-indexer-2.0.0.jar，使用实际项目名称作为前缀之后，就能方便从一个 lib 文件夹中找到某个项目的一组构件。考虑有 5 个项目，每个项目都有一个 core 模块，如果没有前缀，我们会看到很多 core-1.2.jar 这样的文件，加上实际项目名前缀之后，便能很容易区分 foo-core-1.2.jar、bar-core-1.2.jar.....
- ❑ **version**：该元素定义 Maven 项目当前所处的版本，如上例中 nexus-indexer 的版本是 2.0.0。需要注意的是，Maven 定义了一套完成的版本规范，以及快照（SNAPSHOT）的概念。本书 13 章详细讨论版本管理内容。
- ❑ **packaging**：该元素定义 Maven 项目的打包方式。首先，打包方式通常与所生成构件的文件扩展名对应，如上例中 packaging 为 jar，最终的文件名为 nexus-indexer-2.0.0.jar，而使用 war 打包方式的 Maven 项目，最终生成的构件会有一个 .war 文件，不过这不是绝对的。其次，打包方式会影响到构建的生命周期，比如 jar 打包和 war 打包会使用不同的命令。最后，当我们不定义 packaging 的时候，Maven 会使用默认值 jar。
- ❑ **classifier**：该元素用来帮助定义构建输出的一些附属构件。附属构件与主构件对应，如上例中的主构件是 nexus-indexer-2.0.0.jar，该项目可能还会通过使用一些插件生成如 nexus-indexer-2.0.0-javadoc.jar、nexus-indexer-2.0.0-sources.jar 这样一些附属构件，其包含了 Java 文档和源代码。这时候，javadoc 和 sources 就是这两个附属构件的 classifier。这样，附属构件也就拥有了自己唯一的坐标。还有一个关于 classifier 的典型例子是 TestNG，TestNG 的主构件是基于 Java 1.4 平台的，而它又提供了一个 classifier 为 jdk5 的附属构件。注意，我们不能直接定义项目的 classifier，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成。

上述 5 个元素中，groupId、artifactId、version 是必须定义的，packaging 是可选的（默认为 jar），而 classifier 是不能直接定义的。

同时，项目构件的文件名是与坐标相对应的，一般的规则为 artifactId-version[-classifier].packaging，[-classifier]表示可选，比如上例 nexus-indexer 的主构件为 nexus-indexer-2.0.0.jar，附属构件有 nexus-indexer-2.0.0-javadoc.jar。这里还要强调的一点是，packaging 并非一定与构件扩展名对应，比如 packaging 为 maven-plugin 的构件扩展名为 jar。

此外，Maven 仓库的布局也是基于 Maven 坐标，这一点会在介绍 Maven 仓库的时候详细解释。

理解清楚城市中地址的定义方式后，邮递员就能够开始工作了；同样地，理解清楚 Maven 坐标之后，我们就能开始讨论 Maven 的依赖管理了。

3.3 account-mail

在详细讨论 Maven 依赖之前，我们先稍微回顾一下上一章提到的背景案例。案例中有一个 email 模块负责发送账户激活的电子邮件，本节就详细阐述该模块的实现，包括 POM 配置、主代码和测试代码。由于该背景案例的实现是基于 Spring Framework，因此还会涉及到相关的 Spring 配置。

3.3.1 account-email 的 POM

首先看一下该模块的 POM，见代码清单 3-1。

代码清单 3-1：account-email 的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juven.mvnbook.account</groupId>
  <artifactId>account-email</artifactId>
  <name>Account Email</name>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>2.5.6</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</artifactId>
      <version>2.5.6</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>2.5.6</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
```

```
        <artifactId>spring-context-support</artifactId>
        <version>2.5.6</version>
    </dependency>
    <dependency>
        <groupId>javax.mail</groupId>
        <artifactId>mail</artifactId>
        <version>1.4.1</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.7</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.icegreen</groupId>
        <artifactId>greenmail</artifactId>
        <version>1.3.1b</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.5</source>
                <target>1.5</target>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

先观察该项目模块的坐标，groupId：com.juvenxu.mvnbook.account；artifactId：account-email；version：1.0.0-SNAPSHOT。由于该模块属于账户注册服务项目的一部分，因此，其groupId对应了account项目。紧接着，该模块的artifactId仍然以account作为前缀，以方便区分其他项目的构建。最后，1.0.0-SNAPSHOT表示该版本处于开发中，还不稳定。

再看dependencies元素，其包含了多个dependency子元素，这是POM中定义项目依赖的位置。以第一个依赖为例，其groupId:artifactId:version为org.springframework:spring-core:2.5.6，这便是依赖的坐标，任何一个Maven项目都需要定义自己的坐标，当这个

Maven 项目成为其他 Maven 项目的依赖的时候，这组坐标就体现了其价值。本例中的 spring-core，以及后面的 spring-beans、spring-context、spring-context-support 是 Spring Framework 实现依赖注入等功能必要的构件，由于本书的关注点在于 Maven，只会涉及简单的 Spring Framework 的使用，不会详细解释 Spring Framework 的用法，如果读者有不清楚的地方，请参阅 Spring Framework 相关的文档。

在 spring-context-support 之后，有一个依赖为 javax.mail:mail:1.4.1，这是实现发送必须的类型库。

紧接着的依赖为 junit:junit:4.7，JUnit 是 Java 社区事实上的单元测试标准，详细信息请参阅 <http://www.junit.org/>，这个依赖特殊的地方在于一个值为 test 的 scope 子元素，scope 用来定义依赖范围，这里读者暂时只需要了解当依赖范围是 test 的时候，该依赖只会被加入到测试代码的 classpath 中。也就是说，对于项目主代码，该依赖是没有任何作用的。JUnit 是单元测试框架，只有在测试的时候才需要，因此使用该依赖范围。

随后的依赖是 com.icegreen:greenmail:1.31.b，其依赖范围同样为 test。这时也许你已经猜到，该依赖同样只服务于测试目的，GreenMail 是开源的邮件服务测试套件，account-email 模块使用该套件来测试邮件的发送，关于 GreenMail 的详细信息可访问 <http://www.icegreen.com/greenmail/>。

最后，POM 中有一段关于 maven-compiler-plugin 的配置，其目的是开启 Java 5 的支持，第 3 章已经对该配置做过解释，这里不再赘述。

3.3.2 account-email 的主代码

Account-email 项目 Java 主代码位于 src/main/java，资源文件（非 Java）位于 src/main/resources 目录下。

Account-email 只有一个很简单的接口，见代码清单 3-2。

代码清单 3-2：AccountEmailService.java

```
package com.juvenxu.mvnbook.account.email;

public interface AccountEmailService
{
    void sendMail( String to, String subject, String htmlText )
        throws AccountEmailException;
}
```

sendMail() 方法用来发送 html 格式的邮件，to 为接收地址，subject 为邮件主题，htmlText 为邮件内容，如果发送邮件出错，则抛出 AccountEmailException 异常。

对应于该接口的实现见代码清单 3-3：

代码清单 3-3：AccountEmailServiceImpl.java

```
package com.juvenxu.mvnbook.account.email;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

public class AccountEmailServiceImpl
    implements AccountEmailService
{
    private JavaMailSender javaMailSender;

    private String systemEmail;

    public void sendMail( String to, String subject, String htmlText )
        throws AccountEmailException
    {
        try
        {
            MimeMessage msg = javaMailSender.createMimeMessage();
            MimeMessageHelper msgHelper = new MimeMessageHelper( msg );

            msgHelper.setFrom( systemEmail );
            msgHelper.setTo( to );
            msgHelper.setSubject( subject );
            msgHelper.setText( htmlText, true );

            javaMailSender.send( msg );
        }
        catch ( MessagingException e )
        {
            throw new AccountEmailException( "Failed to send mail.", e );
        }
    }

    public JavaMailSender getJavaMailSender()
    {
        return javaMailSender;
    }

    public void setJavaMailSender( JavaMailSender javaMailSender )
    {

```

```
        this.javaMailSender = javaMailSender;
    }

    public String getSystemEmail()
    {
        return systemEmail;
    }

    public void setSystemEmail( String systemEmail )
    {
        this.systemEmail = systemEmail;
    }
}
```

首先，该 AccountEmailServiceImpl 类有一个私有字段 javaMailSender，该字段的类型 org.springframework.mail.javamail.JavaMailSender 是来自于 Spring Framework 的帮助简化邮件发送的工具类库，对应于该字段有一组 getter()和 setter()方法，它们用来帮助实现依赖注入，本节随后会讲述 Spring Framework 依赖注入相关的配置。

在 sendMail()的方法实现中，首先使用 javaMailSender 创建一个 MimeMessage，该 msg 对应了将要发送的邮件。接着使用 MimeMessageHelper 帮助设置该邮件的发送地址、收件地址、主题以及内容，msgHelper.setText(htmlText, true)中的 true 表示邮件的内容为 html 格式。最后，使用 javaMailSender 发送该邮件，如果发送出错，则捕捉 MessageException 异常，包装后再抛出该模块自己定义的 AccountEmailException 异常。

这段 Java 代码中没有邮件服务器配置信息，这得益于 Spring Framework 的依赖注入，这些配置都通过外部的配置注入到了 javaMailSender 中，相关配置信息都在 src/main/resources/account-email.xml 这个配置文件中，见代码清单 3-4。

代码清单 3-4：account-email.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:service.properties" />
    </bean>

    <bean
        class="org.springframework.mail.javamail.JavaMailSenderImpl"
        id="javaMailSender">
        <property name="protocol" value="{email.protocol}" />
```

```
<property name="host" value="${email.host}" />
<property name="port" value="${email.port}" />
<property name="username" value="${email.username}" />
<property name="password" value="${email.password}" />
<property name="javaMailProperties">
    <props>
        <prop key="mail.${email.protocol}.auth">${email.auth}</prop>
    </props>
</property>
</bean>

<bean id="accountEmailService"
    class="com.juvenxu.mvnbook.account.email.AccountEmailServiceImpl">
    <property name="javaMailSender" ref="javaMailSender" />
    <property name="systemEmail" value="${email.systemEmail}" />
</bean>
</beans>
```

Spring Framework 会使用该 XML 配置创建 ApplicationContext，以实现依赖注入。该配置文件定义了一些 bean，基本对应了 Java 程序中的对象。首先解释下 id 为 propertyConfigurer 的 bean，其实现为 org.springframework.beans.factory.config.PropertyPlaceholderConfigurer，这是 Spring Framework 中用来帮助载入 properties 文件的组件，这里定义 location 的值为 classpath:account-email.properties，表示从 classpath 的根路径下载入名为 account-email.properties 文件中的属性。

接着定义 id 为 javaMailSender 的 bean，其实现为 org.springframework.mail.javamail.JavaMailSenderImpl，这里需要定义邮件服务器的一些配置，包括协议、端口、主机、用户名、密码，是否需要认证等属性。这段配置还使用了 Spring Framework 的属性引用，比如 host 的值为 \${email.host}，之前定义 propertyConfigurer 的作用就在于此，这么做可以将邮件服务器相关的配置分离到外部的 properties 文件中，比如可以定义这样一个 properties 文件，配置 javaMailSender 使用 gmail：

```
email.protocol=smtps
email.host=smtp.gmail.com
email.port=465
email.username=your-id@gmail.com
email.password=your-password
email.auth=true
email.systemEmail=your-id@juvenxu.com
```

这样，javaMailSender 实际使用的 protocol 就会成为 smtps，host 会成为 smtp.gmail.com，同理还有 port、username 等其他属性。

最后一个 bean 是 `accountEmailService`，对应了之前描述的 `com.juvenxu.mvnbook.account.email.AccountEmailServiceImpl`，配置中将另外一个 bean `javaMailSender` 注入，使其成为该类 `javaMailSender` 字段的值。

上述就是 Spring Framework 相关的配置，这里不再进一步深入，读者如果有不是很理解的地方，请查询 Spring Framework 相关文档。

3.3.3 account-email 的测试代码

测试相关的 Java 代码位于 `src/test/java` 目录，相关的资源文件则位于 `src/test/resources` 目录。

该模块需要测试的只有一个 `AccountEmailService.sendMail()` 接口。为此，需要配置并启动一个测试使用的邮件服务器，然后提供对应的 `properties` 配置文件供 Spring Framework 载入以配置程序。准备就绪之后，调用该接口发送邮件，然后检查邮件是否发送正确。最后，关闭测试邮件服务器，见代码清单 3-5。

代码清单 3-5：AccountEmailServiceTest.java

```
package com.juvenxu.mvnbook.account.email;

import static junit.framework.Assert.assertEquals;

import javax.mail.Message;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.icegreen.greenmail.util.GreenMail;
import com.icegreen.greenmail.util.GreenMailUtil;
import com.icegreen.greenmail.util.ServerSetup;

public class AccountEmailServiceTest
{
    private GreenMail greenMail;

    @Before
    public void startMailServer()
        throws Exception
    {
        greenMail = new GreenMail( ServerSetup.SMTP );
        greenMail.setUser( "test@juvenxu.com", "123456" );
    }
}
```

```
        greenMail.start();
    }

    @Test
    public void testSendMail()
        throws Exception
    {
        ApplicationContext ctx = new ClassPathXmlApplicationContext( "account-email.xml" );
        AccountEmailService accountEmailService = (AccountEmailService)
ctx.getBean( "accountEmailService" );

        String subject = "Test Subject";
        String htmlText = "<h3>Test</h3>";
        accountEmailService.sendMail( "test@juvenxu.com", "test2@juvenxu.com", subject,
htmlText );

        greenMail.waitForIncomingEmail( 2000, 1 );

        Message[] msgs = greenMail.getReceivedMessages();
        assertEquals( 1, msgs.length );
        assertEquals( subject, msgs[0].getSubject() );
        assertEquals( htmlText, GreenMailUtil.getBody( msgs[0] ).trim() );
    }

    @After
    public void stopMailServer()
        throws Exception
    {
        greenMail.stop();
    }
}
```

这里我们使用 GreenMail 作为测试邮件服务器，在 startMailServer() 中，基于 SMTP 协议初始化 GreenMail，然后创建一个邮件账户并启动邮件服务，该服务默认会监听 25 端口。如果你的机器已经有程序使用该端口，请配置自定义的 ServerSetup 实例使用其他端口。startMailServer() 方法使用了 @before 标注，表示该方法会先于测试方法（@test）之前执行。

对应于 startMailServer()，该测试还有一个 stopMailServer() 方法，标注 @After 表示执行测试方法之后会调用该方法，停止 GreenMail 的邮件服务。

代码的重点在于使用了 @Test 标注的 testSendMail() 方法，该方法首先会根据 classpath 路径中的 account-email.xml 配置创建一个 Spring Framework 的 ApplicationContext，然后从这个 ctx 中获取需要测试的 id 为 accountEmailService 的 bean，并转换成 AccountEmailService

接口，针对接口测试是一个单元测试的最佳实践。得到了 AccountEmailService 之后，就能调用其 sendMail()方法发送电子邮件，当然，这个时候不能忘了邮件服务器的配置，其位于 src/test/resources/account-email.properties：

```
email.protocol=smtp
email.host=localhost
email.port=25
email.username=test@juvenxu.com
email.password=123456
email.auth=true
email.systemEmail=your-id@juvenxu.com
```

这段配置与之前 GreenMail 的配置对应，使用了 smtp 协议，使用本机的 25 端口，并有用用户名、密码等认证配置。

回到测试方法中，邮件发送完毕后，再使用 GreenMail 进行检查。greenMail.waitForIncomingEmail(2000, 1)表示接收一封邮件，最多等待 2 秒。由于 GreenMail 服务完全基于内存，实际上基本不会超过 2 秒。随后的几行代码读取收到的邮件，检查邮件的数目以及第一封邮件的主题和内容。

这时，可以运行 `mvn clean test` 执行测试，Maven 会编译主代码和测试代码，并执行测试，报告一个测试得以正确执行，构建成功。

3.3.4 构建account-email

使用 `mvn clean install` 构建 account-email，Maven 会根据 POM 配置自动下载所需要的依赖构件，执行编译、测试、打包等工作，最后将项目生成的构件 account-email-1.0.0-SNAPSHOT.jar 安装到本地仓库中，这时，该模块就能供其他 Maven 项目使用了。

3.4 依赖的配置

本章的 5.3.1 小节已经罗列一些简单的依赖配置，读者可以看到依赖会有基本的 groupId、artifactId 和 version 等元素组成。其实一个依赖声明可以包含如下的一些元素：

```
<project>
...
<dependencies>
  <dependency>
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
    <type>...</type>
    <scope>...</scope>
    <optional>...</optional>
    <exclusions>
```

```
<exclusion>
...
</exclusion>
...
</exclusions>
</dependency>
...
</dependencies>
...
</project>
```

根元素 `project` 下的 `dependencies` 可以包含一个或者多个 `dependency` 元素，以声明一个或者多个项目依赖。每个依赖可以包含的元素有：

- ❑ **groupId**、**artifactId** 和 **version**：依赖的基本坐标，对于任何一个依赖来说，基本坐标是最重要的，Maven 根据坐标才能找到需要的依赖。
- ❑ **type**：依赖的类型，对应于项目坐标定义的 `packaging`。大部分情况下，该元素不必声明，其默认值为 `jar`。
- ❑ **scope**：依赖的范围，见本章 5.5 小节。
- ❑ **optional**：标记依赖是否可选，见本章 5.8 小节。
- ❑ **exclusions**：用来排除传递性依赖，见本章 5.9.1 小节。

大部分依赖声明只包含基本坐标，然而在一些特殊情况下，其它元素至关重要，本章下面的小节会对它们的原理和使用方式详细介绍。

3.5 依赖范围

上一节提到，JUnit 依赖的测试范围是 `test`，测试范围用元素 `scope` 表示。本节将详细解释什么是测试范围，以及各种测试范围的效果和用途。

首先需要知道，Maven 在编译项目主代码的时候需要使用一套 `classpath`。在上例中，编译项目主代码的时候需要用到 `spring-core`，该文件以依赖的方式被引入到 `classpath` 中。其次，Maven 在编译和执行测试的时候会使用另外一套 `classpath`，上例中的 JUnit 就是一个很好的例子，该文件也以依赖的方式引入到测试使用的 `classpath` 中，不同的是这里的依赖范围是 `test`。最后，实际运行 Maven 项目的时候，又会使用一套 `classpath`，上例中的 `spring-core` 需要在此 `classpath` 中，而 JUnit 则不需要。

依赖范围就是用来控制依赖与这 3 种 `classpath`（编译 `classpath`、测试 `classpath`、运行 `classpath`）的关系，Maven 有以下几种依赖范围：

- ❑ **Compile**：编译依赖范围。如果没有指定，就会默认使用该依赖范围。使用此依赖范围的 Maven 依赖，对于编译、测试、运行三种 classpath 都有效。典型的例子是 spring-core，在编译、测试和运行的时候都需要使用该依赖。
- ❑ **Test**：测试依赖范围。使用此依赖范围的 Maven 依赖，只对于测试 classpath 有效，在编译主代码或者运行项目的使用时将无法使用此类依赖。典型的例子是 JUnit，它只有在编译测试代码及运行测试的时候才需要。
- ❑ **Provided**：已提供依赖范围。使用此依赖范围的 Maven 依赖，对于编译和测试 classpath 有效，但在运行时无效。典型的例子是 servlet-api，编译和测试项目的时候需要该依赖，但在运行项目的时候，由于容器已经提供，就不需要 Maven 重复的引入一遍。
- ❑ **Runtime**：运行时依赖范围。使用此依赖范围的 Maven 依赖，对于测试和运行 classpath 有效，但在编译主代码时无效。典型的例子是 JDBC 驱动实现，项目主代码的编译只需要 JDK 提供的 JDBC 接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体 JDBC 驱动。
- ❑ **System**：系统依赖范围。该依赖与 3 种 classpath 的关系，和 Provided 依赖范围完全一致。但是，使用 System 范围的依赖时必须通过 systemPath 元素显式地指定依赖文件的路径。由于此类依赖不是通过 Maven 仓库解析的，而且往往与本机系统绑定，可能造成构建的不可移植，因此应该谨慎使用。systemPath 元素可以引用环境变量，如：

```
<dependency>
  <groupId>javax.sql</groupId>
  <artifactId>jdbc-stdext</artifactId>
  <version>2.0</version>
  <scope>system</scope>
  <systemPath>${java.home}/lib/rt.jar</systemPath>
</dependency>
```

- ❑ **Import** (Maven 2.0.9 及以上)：导入依赖范围。该依赖范围不会对 3 种 classpath 产生实际的影响，本书将在 8.3.3 节介绍 Maven 依赖和 dependencyManagement 的时候详细介绍此依赖范围。

上述除 import 以外的各种依赖范围与 3 种 classpath 的关系如表 3-1 所示：

表 3-1：依赖范围与 classpath 的关系

依赖范围 (Scope)	对于编译 classpath 有效	对于测试 classpath 有效	对于运行时 classpath 有效	例子

compile	Y	Y	Y	spring-core
test	-	Y	-	JUnit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC 驱动实现
system	Y	Y	-	本地的，Maven 仓库之外的类库文件

3.6 传递性依赖

3.6.1 何为传递性依赖

考虑一个基于 Spring Framework 的项目，如果不使用 Maven，那么在项目中就需要手动下载相关依赖，由于 Spring Framework 又会依赖于其他开源类库，因此实际中往往会下载一个很大的如 spring-framework-2.5.6-with-dependencies.zip 的包，这里包含了所有 Spring Framework 的 jar 包，以及所有它依赖的其他 jar 包。这么做往往就引入了很多不必要的依赖。另一种做法是只下载 spring-framework-2.5.6.zip 这样一个包，这里不包含其他相关依赖，到实际使用的时候，再根据出错信息，或者查询相关文档，加入需要的其他依赖。很显然，这也是一件非常麻烦的事情。

Maven 的传递性依赖机制可以很好地解决这一问题，以 account-email 项目为例，该项目有一个 org.springframework:spring-core:2.5.6 的依赖，而实际上 spring-core 也有它自己的依赖，我们可以直接访问位于中央仓库的该构件的 POM：<http://repo1.maven.org/maven2/org/springframework/spring-core/2.5.6/spring-core-2.5.6.pom>，该文件包含了一个 commons-logging 的依赖，见代码清单 3-6。

代码清单 3-6：spring-core 的 commons-logging 依赖

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

该依赖没有声明依赖范围，那么其依赖范围就是默认的 compile。同时回顾一下 account-email，spring-core 的依赖范围也是 compile。

Account-mail 有一个 compile 范围的 spring-core 依赖，spring-core 有一个 compile 范围的 commons-logging 依赖，那么 commons-logging 就会成为 account-email 的 compile 范围依赖，commons-logging 是 account-email 的一个传递性依赖，如图 3-2 所示：

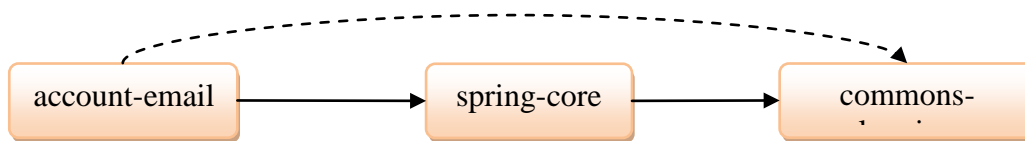


图 3-2 传递性依赖

有了传递性依赖机制，我们在使用 Spring Framework 的时候就不用去考虑它依赖了什么，也不用担心引入多余的依赖。Maven 会解析各个直接依赖的 POM，将那些必要的间接依赖，以传递性依赖的形式引入到当前的项目中。

3.6.2 传递性依赖和依赖范围

依赖范围不仅可以控制依赖与 3 种 classpath 的关系，还对传递性依赖产生影响。上面的例子中，account-email 对于 spring-core 的依赖范围是 compile，spring-core 对于 commons-logging 的依赖范围是 compile，那么 account-email 对于 commons-logging 这一传递性依赖的范围也就是 compile。假设 A 依赖于 B，B 依赖于 C，我们说 A 对于 B 是第一直接依赖，B 对于 C 是第二直接依赖，A 对于 C 是传递性依赖。第一直接依赖的范围和第二直接依赖的范围决定了传递性依赖的范围，如表 3-2 所示，最左边一行表示第一直接依赖范围，最上面一行表示第二直接依赖范围，中间的交叉单元格则表示传递性依赖范围：

表 3-2：依赖范围影响传递性依赖

	compile	test	provided	runtime
compile	compile	-	-	runtime
test	test	-	-	test
provided	provided	-	provided	provided
runtime	runtime	-	-	runtime

为了能够帮助读者更好地理解这个表格，这里再举个例子，account-email 项目有一个 com.icegreen:greenmail:1.3.1b 的直接依赖，我们说这是第一直接依赖，其依赖范围是 test；而 greenmail 又有一个 javax.mail:mail:1.4 的直接依赖，我们说这是第二直接依赖，其依赖范围是 compile。显然 javax.mail:mail:1.4 是 account-email 的传递性依赖，对照表 3-2 可以知道，当第一直接依赖范围为 test，第二直接依赖范围是 compile 的时候，传递性依赖的范围是 test，因此 javax.mail:mail:1.4 是 account-email 的一个范围是 test 的传递性依赖。

仔细观察一下表格，可以发现这样的规律：当第二直接依赖的范围是 `compile` 的时候，传递性依赖的范围与第一直接依赖的范围一致；当第二直接依赖的范围是 `test` 的时候，依赖不会得以传递；当第二直接依赖的范围是 `provided` 的时候，只传递第一直接依赖范围也为 `provided` 的依赖，且传递性依赖的范围同样为 `provided`；当第二直接依赖的范围是 `runtime` 的时候，传递性依赖的范围与第一直接依赖的范围一致，但 `compile` 例外，此时传递性依赖的范围为 `runtime`。

3.7 依赖调解 (Dependency Mediation)

Maven 引入的传递性依赖机制，一方面大大简化和方便了依赖声明，大部分情况下我们只需要关心项目的直接依赖是什么，而不用考虑这些直接依赖会引入什么传递性依赖。但有时候，当传递性依赖造成问题的时候，我们就需要清楚地知道该传递性依赖是从哪条依赖路径引入的。

例如，项目 A 有这样的依赖关系： $A \rightarrow B \rightarrow C \rightarrow X(1.0)$ 、 $A \rightarrow D \rightarrow X(2.0)$ ，X 是 A 的传递性依赖，但是两条依赖路径上有两个版本的 X，那么哪个 X 会被 Maven 解析使用呢？两个版本都被解析显然是不对的，因为那会造成依赖重复，因此必须选择一个。Maven 依赖调解的第一原则是：路径最近者优先。该例中 $X(1.0)$ 的路径长度为 3，而 $X(2.0)$ 的路径长度为 2，因此 $X(2.0)$ 会被解析使用。

依赖调解第一原则不能解决所有问题，比如这样的依赖关系： $A \rightarrow B \rightarrow Y(1.0)$ 、 $A \rightarrow C \rightarrow Y(2.0)$ ， $Y(1.0)$ 和 $Y(2.0)$ 的依赖路径长度是一样的，都为 2。那么到底谁会被解析使用呢？在 Maven 2.0.8 及之前的版本中，这是不确定的，但是从 Maven 2.0.9 开始，为了尽可能避免构建的不确定性，Maven 定义了依赖调解的第二原则：第一声明者优先。在依赖路径长度相等的前提下，在 POM 中依赖声明的顺序决定了谁会被解析使用，顺序最靠前的那个依赖优胜。该例中，如果 B 的依赖声明在 C 之前，那么 $Y(1.0)$ 就会被解析使用。

3.8 可选依赖

假设有这样一个依赖关系，项目 A 依赖于项目 B，B 依赖于项目 X 和 Y，B 对于 X 和 Y 的依赖都是可选依赖： $A \rightarrow B$ 、 $B \rightarrow X(\text{可选})$ 、 $B \rightarrow Y(\text{可选})$ 。根据传递性依赖的定义，如果所有这三个依赖的范围都是 `compile`，那么 X、Y 就是 A 的 `compile` 范围传递性依赖。然而，由于这里 X、Y 是可选依赖，依赖将不会得以传递，换句话说，X、Y 将不会对 A 有任何影响，如图 3-3：

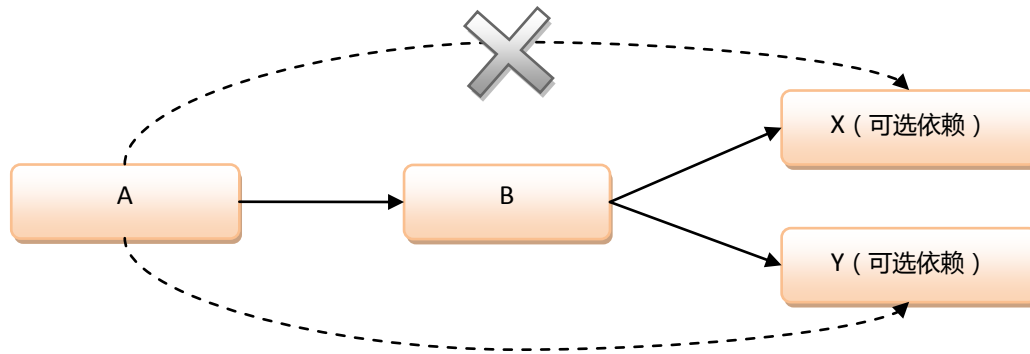


图 3-3 可选依赖

为什么要使用可选依赖这一特性呢？可能项目 B 实现了两个特性，其中的特性一依赖于 X，特性二依赖于 Y，而且这两个特性是互斥的，用户不可能同时使用两个特性。比如 B 是一个持久层隔离工具包，它支持多种数据库，包括 MySQL，PostgreSQL 等等，在构建这个工具包的时候，需要这两种数据库的驱动程序，但在使用这个工具包的时候，只会依赖一种数据库。

项目 B 的依赖声明见代码清单 3-7。

代码清单 3-7：可选依赖的配置

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>project-b</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.10</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>8.4-701.jdbc3</version>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>
```

上述 XML 代码片段中，使用<optional>元素表示 mysql-connector-java 和 postgresql 这两个依赖为可选依赖，它们只会对当前项目 B 产生影响，当其他项目依赖于 B 的时候，这两个

依赖不会被传递。因此，当项目 A 依赖于项目 B 的时候，如果其实际使用基于 MySQL 数据库，那么在项目 A 中就需要显式地声明 `mysql-connector-java` 这一依赖，见代码清单 3-8。

代码清单 3-8：可选依赖不被传递

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>com.juvenxu.mvnbook</groupId>
      <artifactId>project-b</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.10</version>
    </dependency>
  </dependencies>
</project>
```

最后，关于可选依赖需要说明的一点是，在理想的情况下，是不应该使用可选依赖的。前面我们可以看到，使用可选依赖的原因是某一个项目实现了多个特性，在面向对象设计中，有个单一职责性原则，意指一个类应该只有一项职责，而不是糅合太多的功能。这个原则在规划 Maven 项目的时候也同样适用，在上面的例子中，更好的做法是为 MySQL 和 PostgreSQL 分别创建一个 Maven 项目，基于同样的 `groupId` 分配不同的 `artifactId`，如 `com.juvenxu.mvnbook:project-b-mysql` 和 `com.juvenxu.mvnbook:project-b-postgresql`，在各自的 POM 中声明对应的 JDBC 驱动依赖，而且不使用可选依赖，用户则根据需要使用 `project-b-mysql` 或者 `project-b-postgresql`，由于传递性依赖的作用，就不用再声明 JDBC 驱动依赖。

3.9 最佳实践

Maven 依赖涉及的知识点比较多，在理解了主要的功能和原理之后，最需要的当然就是前人的经验总结了，我们称之为最佳实践，本小节归纳了一些使用 Maven 依赖常见的技巧，方便用来避免和处理很多常见的问题。

3.9.1 排除依赖

传递性依赖会给项目隐式的引入很多依赖，这极大地简化了项目依赖的管理，但是有些时候这种特性也会带来问题。例如，当前项目有一个第三方依赖，而这个第三方依赖由于某些原因依赖了另外一个类库的 SNAPSHOT 版本，那么这个 SNAPSHOT 就会成为当前项目的传递性依赖，而 SNAPSHOT 的不稳定性会直接影响到当前的项目，这时就需要排除掉该 SNAPSHOT，并且在当前项目中声明该类库的某个正式发布的版本。还有一些情况，你可能也想要替换某个传递性依赖，比如 Sun JTA API，Hibernate 依赖于这个 JAR，但是由于版权的因素，该类库不在中央仓库中，而 Apache Geronimo 项目有一个对应的实现，这时你就可以排除 Sun JAT API，再声明 Geronimo 的 JTA API 实现，见代码清单 3-9。

代码清单 3-9：排除传递性依赖

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.juvenxu.mvnbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>com.juvenxu.mvnbook</groupId>
      <artifactId>project-b</artifactId>
      <version>1.0.0</version>
      <exclusions>
        <exclusion>
          <groupId>com.juvenxu.mvnbook</groupId>
          <artifactId>project-c</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>com.juvenxu.mvnbook</groupId>
      <artifactId>project-c</artifactId>
      <version>1.1.0</version>
    </dependency>
  </dependencies>
</project>
```

上述代码中，项目 A 依赖于项目 B，但是由于一些原因，不想引入传递性依赖 C，而是自己显式地声明对于项目 C 1.1.0 版本的依赖。代码中使用 exclusions 元素声明排除依赖，exclusions 可以包含一个或者多个 exclusion 子元素，因此可以排除一个或者多个传递性依赖。需要注意的是，声明 exclusion 的时候只需要 groupId 和 artifactId，而不需要 version 元素，这是因为只需要 groupId 和 artifactId 就能唯一定位依赖图中的某个依赖，换句话

说，Maven 解析后的依赖中，不可能出现 groupId 和 artifactId 相同，但是 version 不同的两个依赖，这一点在 5.6 节中已做过解释。该例的依赖解析逻辑如图 3-4 所示：

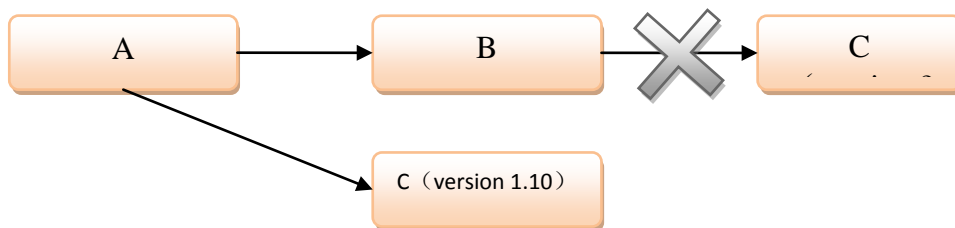


图 3-4 排除依赖

3.9.2 归类依赖

在 3.3.1 节中，有很多关于 Spring Framework 的依赖，它们分别是 org.springframework:spring-core:2.5.6、org.springframework:spring-beans:2.5.6、org.springframework:spring-context:2.5.6 和 org.springframework:spring-context-support:2.5.6，它们是来自同一项目的不同模块，因此，所有这些依赖的版本都是相同的，而且可以预见，如果将来需要升级 Spring Framework，这些依赖的版本会一起升级。这一情况在 Java 中似曾相识，考虑如下简单代码（见代码清单 3-10）：

代码清单 3-10：Java 中重复使用字面量

```
public double c( double r )
{
    return 2 * 3.14 * r;
}

public double s( double r )
{
    return 3.14 * r * r;
}
```

这两个简单的方式计算圆的周长和面积，稍微有经验的程序员一眼就会看出一个问题，使用字面量（3.14）显然不合适，应该使用定义一个常量并在方法中使用，见代码清单 3-11。

代码清单 3-11：Java 中使用常量

```
public final double PI = 3.14;

public double c( double r )
{
    return 2 * PI * r;
}
```

```
public double s( double r )
{
    return PI * r * r;
}
```

使用常量一方面让代码变得更加简洁，更重要的是可以避免重复，在需要更改 PI 的值的时候，只需要修改一处，降低了错误发生的概率。

同理，对于 account-email 中这些 Spring Framework 来说，也应该在一个唯一的地方定义版本，并且在 dependency 声明中引用这一版本，这样，在升级 Spring Framework 的时候就只需要修改一处，实现方式见代码清单 3-12。

代码清单 3-12：使用 Maven 属性归类依赖

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.juven.mvnbook.account</groupId>
<artifactId>account-email</artifactId>
<name>Account Email</name>
<version>1.0.0-SNAPSHOT</version>

<properties>
<springframework.version>2.5.6</springframework.version>
</properties>

<dependencies>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>${springframework.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-beans</artifactId>
<version>${springframework.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
<version>${springframework.version}</version>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-context-support</artifactId>
<version>${springframework.version}</version>
</dependency>
```

```
</dependencies>
```

```
</project>
```

这里简单用到了 Maven 属性（本书 14.1 节详细介绍 Maven 属性），首先使用 properties 元素定义 Maven 属性，该例中定义了一个 springframework.version 子元素，其值为 2.5.6，有了这个属性定义之后，Maven 运行的时候会将 POM 中的所有的 \${springframework.version} 替换成实际值 2.5.6，也就是说，可以使用美元符号和大括弧环绕的方式引用 Maven 属性。然后，将所有 Spring Framework 依赖的版本值用这一属性引用表示，这和在 Java 中用常量 PI 替换 3.14 是同样的道理，不同的只是语法。

3.9.3 优化依赖

在软件开发过程中，程序员会通过重构等方式不断地优化自己的代码，使其变得更简洁、更灵活。同理，程序员也应该能够对 Maven 项目的依赖了然于胸，并对其进行优化，如去除多余的依赖，显式地声明某些必要的依赖。

通过阅读本章前面的内容，读者应该能够了解到：Maven 会自动解析所有项目的直接依赖和传递性依赖，并且根据规则正确判断每个依赖的范围，对于一些依赖冲突，也能进行调节，以确保任何一个构件只有唯一的版本在依赖中存在。在这些工作之后，最后得到的那些依赖被称为已解析依赖（Resolved Dependency）。可以运行如下的命令查看当前项目的已解析依赖：

```
mvn dependency:list
```

在 account-email 项目中执行该命令，结果如图 3-5 所示：

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] -----
[INFO] Building Account Email
[INFO]   task-segment: [dependency:list]
[INFO] -----
[INFO] [dependency:list {execution: default-cli}]
[INFO]
[INFO] The following files have been resolved:
[INFO]   aopalliance:aopalliance:jar:1.0:compile
[INFO]   com.icegreen:greenmail:jar:1.3.1b:test
[INFO]   commons-logging:commons-logging:jar:1.1.1:compile
[INFO]   javax.activation:activation:jar:1.1:compile
[INFO]   javax.mail:mail:jar:1.4.1:compile
[INFO]   junit:junit:jar:4.7:test
[INFO]   org.slf4j:slf4j-api:jar:1.3.1:test
[INFO]   org.springframework:spring-beans:jar:2.5.6:compile
[INFO]   org.springframework:spring-context:jar:2.5.6:compile
[INFO]   org.springframework:spring-context-support:jar:2.5.6:compile
[INFO]   org.springframework:spring-core:jar:2.5.6:compile
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
```


图 3-5 已解析依赖列表

图 3-5 显示了所有 account-email 的已解析依赖，同时，每个依赖的范围也得以明确标示。

在此基础上，还能进一步了解已解析依赖的信息。将直接在当前项目 POM 声明的依赖定义为顶层依赖，而这些顶层依赖的依赖则定义为第二层依赖，以此类推，有第三、第四层依赖。当这些依赖经 Maven 解析后，就会构成一个依赖树，通过这棵依赖树就能很清楚地看到某个依赖是通过哪条传递路径引入的。我们可以运行如下命令查看当前项目的依赖树：

```
mvn dependency:tree
```

在 account-email 中执行该命令，效果如图 3-6 所示：

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] -----
[INFO] Building Account Email
[INFO]   task-segment: [dependency:tree]
[INFO] -----
[INFO] [dependency:tree {execution: default-cli}]
[INFO] com.juven.mvnbook.account:account-email:jar:1.0.0-SNAPSHOT
[INFO] +- org.springframework:spring-core:jar:2.5.6:compile
[INFO] |   \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.springframework:spring-beans:jar:2.5.6:compile
[INFO] +- org.springframework:spring-context:jar:2.5.6:compile
[INFO] |   \- aopalliance:aopalliance:jar:1.0:compile
[INFO] +- org.springframework:spring-context-support:jar:2.5.6:compile
[INFO] +- javax.mail:mail:jar:1.4.1:compile
[INFO] |   \- javax.activation:activation:jar:1.1:compile
[INFO] +- junit:junit:jar:4.7:test
[INFO] \- com.icegreen:greenmail:jar:1.3.1b:test
[INFO]     \- org.slf4j:slf4j-api:jar:1.3.1:test
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
```

图 3-6 已解析依赖树

从图 3-6 中的依赖树我们能够看到，虽然我们没有声明 org.slf4j:slf4j-api:1.3 这一依赖，但它还是经过 com.icegreen:greenmail:1.3 成为了当前项目的传递性依赖，而且其范围是 test。

使用 dependency:list 和 dependency:tree 可以帮助我们详细了解项目中所有依赖的具体信息，在此基础上，还有 dependency:analyze 一个工具可以帮助分析当前项目的依赖。

为了说明该工具的用途，先将 3.3.1 POM 中的 spring-context 这一依赖删除，然后构建项目，你会发现编译、测试和打包都不会有任何问题，通过分析依赖树，可以看到 spring-context 是 spring-context-support 的依赖，因此会得以传递到项目的 classpath 中。现在再运行如下命令：

```
mvn dependency:analyze
```

结果如图 3-7 所示：

```
[INFO] Preparing dependency:analyze
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding <GB18030 actually> to copy filtered
[INFO] Copying 1 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding <GB18030 actually> to copy filtered
[INFO] Copying 1 resource
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze {execution: default-cli}]
[WARNING] Used undeclared dependencies found:
[WARNING]   org.springframework:spring-context:jar:2.5.6:compile
[WARNING] Unused declared dependencies found:
[WARNING]   org.springframework:spring-core:jar:2.5.6:compile
[WARNING]   org.springframework:spring-beans:jar:2.5.6:compile
[INFO] -----
```

图 3-7 使用但未声明的依赖与声明但未使用的依赖

该结果中重要的是两个部分。首先是 Used undeclared dependencies，意指项目中使用到的，但是没有显式声明的依赖，这里是 spring-context。这种依赖意味着潜在的风险，当前项目直接在使用它们，例如有很多相关的 JAVA import 声明，而这种依赖是通过直接依赖传递进来的，当升级直接依赖的时候，相关传递性依赖的版本也可能发生变化，这种变化不易察觉，但是有可能导致当前项目出错，例如由于接口的改变，当前项目中的相关代码无法编译。这种隐藏的、潜在的威胁一旦出现，就往往需要耗费大量的时间来查明真相。因此，显式声明任何项目中直接用到的依赖。

结果中还有一个重要的部分是 Unused declared dependencies，意指项目中未使用的，但显式声明的依赖，这里有 spring-core 和 spring-beans。需要注意的是，对于这样一类依赖，我们不应该简单地直接删除其声明，而是应该仔细分析。由于 dependency:analyze 只会分析编译主代码和测试代码需要用到的依赖，一些执行测试和运行时需要的依赖它就发现不了，很显然，该例中的 spring-core 和 spring-beans 是运行 Spring Framework 项目必要的类库，因此不应该删除依赖声明。当然，有时候确实能通过该信息找到一些没用的依赖，但一定要小心测试。

3.10 小结

本章主要介绍了 Maven 的两个核心概念：坐标和依赖。解释了坐标的来由，并详细阐述了各坐标元素的作用及定义方式。随后我们引入 account-email 这一实际的基于 Spring Framework 的模块，包括了 POM 定义、主代码和测试代码。在这一直观感受的基础上，再花了大篇幅介绍 Maven 依赖，包括依赖范围、传递性依赖、可选依赖等概念。最后，当然少不了关于依赖的一些最佳实践。通过阅读本章，读者应该已经能够透彻地了解 Maven 的依赖管理机制。下一章将会介绍 Maven 另一个核心概念--仓库。

第 4 章 使用Maven构建Web应用

到目前为止，本书讨论的只有打包类型为 JAR 或者 POM 的 Maven 项目。但在现今的互联网时代，我们创建的大部分应用程序都是 Web 应用，在 Java 的世界中，Web 项目的标准打包方式是 WAR (Web Application Archives)，因此本章介绍一个 WAR 模块--account-web，它也来自于本书的账户注册服务背景案例。在介绍该模块之前，本章还会先实现 account-service。此外，读者还能看到如何借助 jetty-maven-plugin 来快速开发和测试 Web 模块，以及使用 Cargo 实现 Web 项目的自动化部署。

4.1 Web项目的目录结构

我们都知道，基于 Java 的 Web 应用，其标准的打包方式是 WAR。WAR 与 JAR 类似，只不过它可以包含更多的内容，如 JSP 文件、Servlet、Java 类、web.xml 配置文件、依赖 JAR 包、静态 web 资源如 HTML、CSS、JavaScript 文件，等等。一个典型的 WAR 文件会有如下目录结构：

```
- war/  
  + META-INF/  
  + WEB-INF/  
    | + classes/  
    | | + ServletA.class  
    | | + config.properties  
    | | + ...  
    | |  
    | + lib/  
    | | + dom4j-1.4.1.jar  
    | | + mail-1.4.1.jar  
    | | + ...  
    | |  
    | + web.xml  
    |  
  + img/  
  |  
  + css/  
  |  
  + js/  
  |  
  + index.html  
  + sample.jsp
```

一个 WAR 包下至少包含两个子目录：META-INF 和 WEB-INF，前者包含了一些打包元数据信息，我们一般不去关心；后者是 WAR 包的核心，WEB-INF 下必须包含一个 Web 资源表述文件 web.xml，它的子目录 classes 包含所有该 Web 项目的类，而另一个子目录 lib 则包含所有该 Web 项目的依赖 JAR 包，classes 和 lib 目录都会在运行的时候被加入到 Classpath 中。除了 META-INF 和 WEB-INF 外，一般的 WAR 包都会包含很多 Web 资源，例如你往往可以在 WAR 包的根目录下看到很多 html 或者 jsp 文件，此外还能看到一些文件夹如 img、css 和 js，它们会包含对应的文件供页面使用。

同任何其他 Maven 项目一样，Maven 对 Web 项目的布局结构也有一个通用的约定，不过首先要记住的是，用户必须为 Web 项目显式指定打包方式为 war，如代码清单 4-1 所示：

代码清单 4-1：显式指定 Web 项目的打包方式为 war

```
<project>
...
<groupId>com.juvenxu.mvnbook</groupId>
<artifactId>sample-war</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
...
</project>
```

如果不显式地指定 packaging，Maven 会使用默认的 jar 打包方式，从而导致无法正确打包 Web 项目。

Web 项目的类及资源文件同一般 JAR 项目一样，默认位置都是 src/main/java/和 src/main/resources，测试类及测试资源文件的默认位置是 src/test/java/和 src/test/resources/。Web 项目比较特殊的地方在于：它还有一个 Web 资源目录，其默认位置是 src/main/webapp/。一个典型的 Web 项目的 Maven 目录结构如下：

```
+ project
|
+ pom.xml
|
+ src/
  + main/
    | + java/
    | | + ServletA.java
    | | + ...
```

```
| |
| + resources/
| | + config.properties
| | + ...
| |
| + webapp/
|   + WEB-INF/
|     + web.xml
|     |
|     + img/
|     |
|     + css/
|     |
|     + js/
|     +
|     + index.html
|     + sample.jsp
|
+ test/
  + java/
  + resources/
```

在 `src/main/webapp/` 目录下，必须包含一个子目录 `WEB-INF`，该子目录还必须包含 `web.xml` 文件。`src/main/webapp` 目录下的其他文件和目录包括 `html`、`jsp`、`css`、`JavaScript` 等等，它们与 `WAR` 包中的 `Web` 资源完全一致。

在我们使用 Maven 创建 Web 项目之前，必须首先理解这种 Maven 项目结构和 `WAR` 包结构的对应关系，有一点需要注意的是，`WAR` 包中有一个 `lib` 目录包含所有依赖 JAR 包，但 Maven 项目结构中没有这样一个目录，这是因为依赖都配置在 `POM` 中，Maven 在用 `WAR` 方式打包的时候会根据 `POM` 的配置从本地仓库复制相应的 JAR 文件。

4.2 account-service

本章将完成我们的背景案例项目，读者可以回顾第 4 章，除了之前实现的 `account-email`、`account-persist` 和 `account-captcha` 之外，该项目还包括 `account-service` 和 `account-web` 两个模块。其中，`account-service` 用来封装底层三个模块的细节，并对外提供简单的接口，而 `account-web` 仅包含了一些涉及 `Web` 的相关内容，如 `Servlet` 和 `JSP` 等等。

4.2.1 account-service的POM

account-service 用来封装 account-email、account-persist 和 account-captcha 三个模块的细节，因此它肯定需要依赖这三个模块，account-service 的 POM 内容如代码清单 4-2 所示：

代码清单 4-2：account-service 的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>

  <artifactId>account-service</artifactId>
  <name>Account Service</name>

  <properties>
    <greenmail.version>1.3.1b</greenmail.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-email</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-persist</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-captcha</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>
```

```
<dependency>
  <groupId>com.icegreen</groupId>
  <artifactId>greenmail</artifactId>
  <version>${greenmail.version}</version>
  <scope>test</scope>
</dependency>
</dependencies>

<build>
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
      <filtering>true</filtering>
    </testResource>
  </testResources>
</build>
</project>
```

与其他模块一样，account-service 继承自 account-parent，它依赖于 account-email、account-persist 和 account-captcha 三个模块，由于是同一项目中的其他模块，groupId 和 version 都完全一致，因此可以使用 Maven 属性 \${project.groupId} 和 \${project.version} 进行替换，这样可以在升级项目版本的时候减少更改的数量。项目的其他配置如 junit 和 greenmail 依赖，以及测试资源目录过滤配置，都是为了单元测试，前面的章节已经介绍过，不再赘述。

4.2.2 account-service 的主代码

account-service 的目的是封装下层细节，对外暴露尽可能简单的接口，我们先看一下这个接口是怎样的，见代码清单 4-3：

代码清单 4-3：AccountService.java

```
package com.juvenxu.mvnbook.account.service;

public interface AccountService
{
    String generateCaptchaKey()
        throws AccountServiceException;

    byte[] generateCaptchaImage( String captchaKey )
        throws AccountServiceException;

    void signUp( SignUpRequest signUpRequest )
        throws AccountServiceException;
```



```
void activate( String activationNumber )
    throws AccountServiceException;

void login( String id, String password )
    throws AccountServiceException;
}
```

正如 4.3.1 节介绍的那样，该接口提供 5 个方法。generateCaptchaKey()用来生成一个验证码的唯一标识符，generateCaptchaImage()根据这个标识符生成验证码图片，图片以字节流的方式返回。用户需要使用 signUp()方法进行注册，注册信息使用 SignUpRequest 进行封装，这个 SignUpRequest 类是一个简单的 POJO，它包含了注册 ID、email、用户名、密码、验证码标识、验证码值等信息。注册成功之后，用户会得到一个激活链接，该链接包含了一个激活码，这个时候用户需要使用 activate()方法并传入激活码以激活账户。最后，login()方法用来登录。

下面来看一下该接口的实现类 AccountServiceImpl.java，首先它需要使用 3 个底层模块的服务，如代码清单 4-4 所示：

代码清单 4-4：AccountServiceImpl.java 第 1 部分

```
public class AccountServiceImpl
    implements AccountService
{
    private AccountPersistService accountPersistService;

    private AccountEmailService accountEmailService;

    private AccountCaptchaService accountCaptchaService;

    public AccountPersistService getAccountPersistService()
    {
        return accountPersistService;
    }

    public void setAccountPersistService( AccountPersistService
accountPersistService )
    {
        this.accountPersistService = accountPersistService;
    }
    ...
}
```

三个私有变量来自 account-persist、account-email 和 account-captcha 模块，它们都有各自的 get()和 set()方法，并且通过 Spring 注入。

AccountServiceImpl.java 借助 accountCaptchaService 实现验证码的标识符生成及验证码图片生成，如代码清单 4-5 所示：

代码清单 4-5：AccountServiceImpl.java 第 2 部分

```
public byte[] generateCaptchaImage( String captchaKey )
    throws AccountServiceException
{
    try
    {
        return accountCaptchaService.generateCaptchaImage( captchaKey );
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to generate Captcha
Image.", e );
    }
}

public String generateCaptchaKey()
    throws AccountServiceException
{
    try
    {
        return accountCaptchaService.generateCaptchaKey();
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to generate Captcha key.",
e );
    }
}
```

稍微复杂一点的是 signUp()方法的实现，见代码清单 4-6：

代码清单 4-6：AccountServiceImpl.java 第 3 部分

```
private Map<String, String> activationMap = new HashMap<String, String>();

public void signUp( SignUpRequest signUpRequest )
    throws AccountServiceException
{
    try
```

```

    {
        if
( !signUpRequest.getPassword().equals( signUpRequest.getConfirmPassword() ) )
        {
            throw new AccountServiceException( "2 passwords do not match." );
        }

        if ( !accountCaptchaService
            .validateCaptcha(
signUpRequest.getCaptchaValue() ) )
        {
            throw new AccountServiceException( "Incorrect Captcha." );
        }

        Account account = new Account();
        account.setId( signUpRequest.getId() );
        account.setEmail( signUpRequest.getEmail() );
        account.setName( signUpRequest.getName() );
        account.setPassword( signUpRequest.getPassword() );
        account.setActivated( false );

        accountPersistService.createAccount( account );

        String activationId = RandomGenerator.getRandomString();

        activationMap.put( activationId, account.getId() );

        String link = signUpRequest.getActivateServiceUrl().endsWith( "/" ) ?
signUpRequest.getActivateServiceUrl()
            + activationId : signUpRequest.getActivateServiceUrl() + "?key=" +
activationId;

        accountEmailService.sendMail( account.getEmail(), "Please Activate Your
Account", link );
    }
    catch ( AccountCaptchaException e )
    {
        throw new AccountServiceException( "Unable to validate captcha.", e );
    }
    catch ( AccountPersistException e )
    {
        throw new AccountServiceException( "Unable to create account.", e );
    }
    catch ( AccountEmailException e )
    {

```

```
        throw new AccountServiceException( "Unable to send activation mail.", e );
    }

}
```

signUp() 方法首先检查请求中的两个密码是否一致，接着使用 accountCaptchaService 检查验证码，下一步使用请求中的用户信息实例化一个 Account 对象，并使用 accountPersistService 将用户信息保存。下一步是生成一个随机的激活码并保存在临时的 activateMap 中，然后基于该激活码和请求中的服务器 URL 创建一个激活链接，并使用 accountEmailService 将该链接发送给用户。如果其中任何一步发生异常，signUp() 方法会创建一个一致的 AccountServiceException 对象，提供并抛出对应的异常提示信息。

最后再看一下相对简单的 activate() 和 login() 方法，见代码清单 4-7：

代码清单 4-7：AccountServiceImpl.java 第 4 部分

```
public void activate( String activationId )
    throws AccountServiceException
{
    String accountId = activationMap.get( activationId );

    if ( accountId == null )
    {
        throw new AccountServiceException( "Invalid account activation ID." );
    }

    try
    {
        Account account = accountPersistService.readAccount( accountId );
        account.setActivated( true );
        accountPersistService.updateAccount( account );
    }
    catch ( AccountPersistException e )
    {
        throw new AccountServiceException( "Unable to activate account." );
    }
}

public void login( String id, String password )
    throws AccountServiceException
{
    try
    {
```

```
Account account = accountPersistService.readAccount( id );

if ( account == null )
{
    throw new AccountServiceException( "Account does not exist." );
}

if ( !account.isActivated() )
{
    throw new AccountServiceException( "Account is disabled." );
}

if ( !account.getPassword().equals( password ) )
{
    throw new AccountServiceException( "Incorrect password." );
}
}
catch ( AccountPersistException e )
{
    throw new AccountServiceException( "Unable to log in.", e );
}
}
```

activate()方法仅仅是简单根据激活码从临时的 activationMap 中寻找对应的用户 ID，如果找到就更新账户状态为激活。login()方法则是根据 ID 读取用户信息，检查其是否为激活，并比对密码，如果有任何错误则抛出异常。

除了上述代码之外，account-service 还包括一些 Spring 配置文件，以及单元测试代码，这里就不再详细介绍，有兴趣的读者可以自行下载阅读。

4.3 account-web

account-web 是本书背景案例中唯一的 web 模块，本书旨在用该模块来阐述如何使用 Maven 来构建一个 Maven 项目。由于 account-service 已经封装了所有下层细节，account-web 只需要在此基础上提供一些 web 页面，并使用简单 servlet 与后台实现交互控制。读者将会看到一个具体 web 项目的 POM 是怎样的，也将能体会到让 web 模块尽可能简洁带来的好处。

4.3.1 account-web的POM

除了使用打包方式 war 之外，web 项目的 POM 与一般项目并没多大的区别，account-web 的 POM 代码见代码清单 4-8：

代码清单 4-8：account-web 的 POM

```
<?xml version="1.0"?>
<project
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>

  <artifactId>account-web</artifactId>
  <packaging>war</packaging>
  <name>Account Web</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>account-service</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

如上述代码所示，account-web 的 packaging 元素值为 war，表示这是一个 Web 项目，需要以 war 方式进行打包。account-web 依赖于 servlet-api 和 jsp-api 这两个几乎所有 Web 项目都要依赖的包，它们为 servlet 和 jsp 的编写提供支持，需

要注意的是，这两个依赖的范围是 `provided`，表示它们最终不会被打包至 `war` 文件中，这是因为几乎所有 `web` 容器都会提供这两个类库，如果 `war` 包中重复出现，就会导致潜在的依赖冲突问题。`account-web` 还依赖于 `account-service` 和 `spring-web`，其中前者为 `web` 应用提供底层支持，后者为 `web` 应用提供 Spring 的集成支持。

在一些 `web` 项目中，读者可能会看到 `finalName` 元素的配置，该元素用来标识项目生成的主构件的名称，该元素的默认值已在超级 POM 中设定，值为 `${project.artifactId}-${project.version}`，因此代码清单 4-7 对应的主构件名称为 `account-web-1.0.0-SNAPSHOT.war`。不过，这样的名称显然不利于部署，不管是测试环境还是最终产品环境，我们都不想在访问页面的时候输入冗长的地址，因此我们会需要名字更为简洁的 `war` 包，这时可以如下配置 `finalName` 元素：

```
<finalName>account</finalName>
```

经此配置后，项目生成的 `war` 包名称就会成为 `account.war`，更方便部署。

4.3.2 `account-web` 的主代码

`account-web` 的主代码包含了 2 个 JSP 页面和 4 个 Servlet，它们分别为：

- ❑ `signup.jsp`：账户注册页面。
- ❑ `login.jsp`：账户登录页面。
- ❑ `CaptchaImageServlet`：用来生成验证码图片的 Servlet。
- ❑ `LoginServlet`：处理账户注册请求的 Servlet。
- ❑ `ActivateServlet`：处理账户激活的 Servlet。
- ❑ `LoginServlet`：处理账户登录的 Servlet。

Servlet 的配置可以从 `web.xml` 中获得，该文件位于项目的 `src/main/webapp/WEB-INF/` 目录，其内容见代码清单 4-9：

代码清单 4-9：`account-web` 的 `web.xml`

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Sample Maven Project: Account Service</display-name>
```



```

    <listener>
      <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        classpath:/account-persist.xml
        classpath:/account-captcha.xml
        classpath:/account-email.xml
        classpath:/account-service.xml
      </param-value>
    </context-param>
    <servlet>
      <servlet-name>CaptchalImageServlet</servlet-name>
      <servlet-
class>com.juvenxu.mvnbook.account.web.CaptchalImageServlet</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>SignUpServlet</servlet-name>
      <servlet-
class>com.juvenxu.mvnbook.account.web.SignUpServlet</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>ActivateServlet</servlet-name>
      <servlet-
class>com.juvenxu.mvnbook.account.web.ActivateServlet</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>LoginServlet</servlet-name>
      <servlet-class>com.juvenxu.mvnbook.account.web.LoginServlet</servlet-
class>
    </servlet>
    <servlet-mapping>
      <servlet-name>CaptchalImageServlet</servlet-name>
      <url-pattern>/captcha_image</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>SignUpServlet</servlet-name>
      <url-pattern>/signup</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>ActivateServlet</servlet-name>
      <url-pattern>/activate</url-pattern>
    </servlet-mapping>
    <servlet-mapping>

```

```
<servlet-name>LoginServlet</servlet-name>
<url-pattern>/login</url-pattern>
</servlet-mapping>
</web-app>
```

web.xml 首先配置了该 Web 项目的显示名称，接着是一个名为 ContextLoaderListener 的 ServletListener，该 listener 来自 spring-web，它用来为 web 项目启动 Spring 的 IoC 容器，从而实现 Bean 的注入。名为 contextConfigLocation 的 context-param 则用来指定 Spring 配置文件的位置，这里的值是四个模块的 Spring 配置 XML 文件，例如 classpath://account-persist.xml 表示从 classpath 的根路径读取名为 account-persist.xml 的文件，我们知道 account-persist.xml 文件在 account-persist 模块打包后的根路径下，这一 JAR 文件通过依赖的方式被引入到 account-web 的 classpath 下。

web.xml 中的其余部分是 Servlet，包括各个 Servlet 的名称、类名、以及对应的 URL 模式。

下面来看一个位于 src/main/webapp/目录的 signup.jsp 文件，该文件用来呈现账户注册页面，其内容如代码清单 4-10 所示：

代码清单 4-10：signup.jsp

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="com.juvenxu.mvnbook.account.service.*,
    org.springframework.context.ApplicationContext,
    org.springframework.web.context.support.WebApplicationContextUtils"%>
<html>
<head>
<style type="text/css">
...
</style>
</head>
<body>

<%
    ApplicationContext          context          =
WebApplicationContextUtils.getWebApplicationContext( getServletContext() );
    AccountService      accountservice      =      (AccountService)
context.getBean( "accountService" );
    String captchaKey = accountservice.generateCaptchaKey();
%>

<div class="text-field">
```

```

<h2>注册新账户</h2>
<form name="signup" action="signup" method="post">
<label>账户ID : </label><input type="text" name="id"></input><br/>
<label>Email : </label><input type="text" name="email"></input><br/>
<label>显示名称 : </label><input type="text" name="name"></input><br/>
<label>    密    码    :    </label><input    type="password"
name="password"></input><br/>
<label>确认密码 : </label>
<input type="password" name="confirm_password"></input><br/>
<label>    验    证    码    :    </label><input    type="text"
name="captcha_value"></input><br/>
<input type="hidden" name="captcha_key" value="<%=captchaKey%>"/>

</br>
<button>确认并提交</button>
</form>
</div>

</body>
</html>

```

该 JSP 的主题是一个 name 为 signup 的 HTML FORM，其中包含了 ID、Email、名称、密码等字段，这与一般的 HTML 内容并无差别，不同的地方在于，该 JSP 文件引入了 Spring 的 ApplicationContext 类，并且用此类加载后台的 accountService，然后使用 accountService 首先生成一个验证码的 key，然后在 FORM 中使用该 key 调用 captcha_image 对应的 Servlet 生成其标识的验证码图片。需要注意的是，上述代码中略去了 css 片段。

账户注册页面如图 4-



1 所示：

图 4-1：账户注册页面

上述 JSP 中使用到了/captcha_image 这一资源获取验证码图片，根据 web.xml 我们知道该资源对应了 CaptchalImageServlet，下面看一下它的代码，见代码清单 4-11：

代码清单 4-11：CaptchalImageServlet.java

```
package com.juvenxu.mvnbook.account.web;

import java.io.IOException;
import ...

public class CaptchalImageServlet
    extends HttpServlet
{
    private ApplicationContext context;

    private static final long serialVersionUID = 5274323889605521606L;

    @Override
    public void init()
        throws ServletException
    {
        super.init();
        context =
WebApplicationContextUtils.getWebApplicationContext( getServletContext() );
    }

    public void doGet( HttpServletRequest request, HttpServletResponse response )
        throws ServletException,
        IOException
    {
        String key = request.getParameter( "key" );

        if ( key == null || key.length() == 0 )
        {
            response.sendError( 400, "No Captcha Key Found" );
        }
        else
        {
            AccountService service = (AccountService)
```

```

context.getBean( "accountService" );

    try
    {
        response.setContentType( "image/jpeg" );
        OutputStream out = response.getOutputStream();
        out.write( service.generateCaptchalImage( key ) );
        out.close();
    }
    catch ( AccountServiceException e )
    {
        response.sendError( 404, e.getMessage() );
    }
}
}
}

```

CaptchalImageServlet 在 init()方法中初始化 Spring 的 ApplicationContext，这一 context 用来获取 Spring Bean。Servlet 的 doGet()方法中首先检查 key 参数，如果为空则返回 HTTP 400 错误，标识客户端的请求不合法，如果不为空，则载入 AccountService 实例，该类的 generateCaptchalImage()方法能够产生一个验证码图片的字节流，我们将其设置成 image/jpeg 格式，并写入到 Servlet 相应的输出流中，客户端就能得到如图 4-1 所示的验证码图片。

代码清单 4-10 中 FROM 的提交目标是 signup，其对应了 SignUpServlet，其内容如代码清单 4-12 所示：

代码清单 4-12：SignUpServlet.java

```

public class SignUpServlet
    extends HttpServlet
{
    private static final long serialVersionUID = 4784742296013868199L;

    private ApplicationContext context;

    @Override
    public void init()
        throws ServletException
    {
        super.init();
        context
WebApplicationContextUtils.getWebApplicationContext( getServletContext() );
    }
}

```

```

@Override
protected void doPost( HttpServletRequest req, HttpServletResponse resp )
    throws ServletException,
        IOException
{
    String id = req.getParameter( "id" );
    String email = req.getParameter( "email" );
    String name = req.getParameter( "name" );
    String password = req.getParameter( "password" );
    String confirmPassword = req.getParameter( "confirm_password" );
    String captchaKey = req.getParameter( "captcha_key" );
    String captchaValue = req.getParameter( "captcha_value" );

    if ( id == null || id.length() == 0 || email == null || email.length() == 0 ||
name == null
        || name.length() == 0 || password == null || password.length() == 0 ||
confirmPassword == null
        || confirmPassword.length() == 0 || captchaKey == null ||
captchaKey.length() == 0 || captchaValue == null
        || captchaValue.length() == 0 )
    {
        resp.sendError( 400, "Parameter Incomplete." );
        return;
    }

    AccountService service = (AccountService)
context.getBean( "accountService" );

    SignUpRequest request = new SignUpRequest();

    request.setId( id );
    request.setEmail( email );
    request.setName( name );
    request.setPassword( password );
    request.setConfirmPassword( confirmPassword );
    request.setCaptchaKey( captchaKey );
    request.setCaptchaValue( captchaValue );

    request.setActivateServiceUrl( getServletContext().getRealPath( "/" ) +
"activate" );

    try
    {
        service.signUp( request );
        resp.getWriter().print( "Account is created, please check your mail box for
activation link." );
    }

```

```
    }  
    catch ( AccountServiceException e )  
    {  
        resp.sendError( 400, e.getMessage() );  
        return;  
    }  
}  
}
```

SignUpServlet 的 doPost()接受客户端的 HTTP POST 请求，首先它读取请求中的 id、name、email 等参数，然后验证这些参数的值是否为空，如果验证正确，则初始化一个 SignUpRequest 实例，其包含了注册账户所需要的各类数据，其中的 activateServiceUrl 表示服务应该基于什么地址发送账户激活链接邮件，这里的值是与 signup 平行的 activate 地址，这正是 ActivationServlet 的地址。SignUpServlet 使用 AccountService 注册账户，所有的细节都已经封装在 AccountService 中，如果注册成功，服务器打印一条简单的提示信息。

上述我们介绍了一个 JSP 和两个 Servlet，它们都非常简单，鉴于篇幅的原因，这里就不再详细解释另外几个 JSP 及 Servlet，感兴趣的读者可以自行下载本书的样例源码。

4.4 使用jetty-maven-plugin进行测试

在进行 Web 开发的时候，我们总是无法避免打开浏览器对应用进行测试，比如为了验证程序功能、验证页面布局，尤其是一些与页面相关的特性，手动部署到 Web 容器进行测试似乎是唯一的方法。近年来出现了很多自动化的 Web 测试技术如 Selenium，它能够录制 Web 操作，生成各种语言脚本，然后自动重复这些操作以进行测试。应该说，这类技术方法是未来的趋势，但无论如何，手动的、亲眼比对验证的测试是无法被完全替代的。测试 Web 页面的做法通常是将项目打包并部署到 Web 容器中，本节介绍如何使用 jetty-maven-plugin，以使这些步骤更为便捷。

在介绍 jetty-maven-plugin 之前，笔者要强调一点，虽然手动的 Web 页面测试是必不可少的，但这种方法绝不应该被滥用。现实中常见的情况是，很多程序员即使修改了一些较底层的代码（如数据库访问、业务逻辑），都会习惯性地打开浏览器测试整个应用，这往往是没有必要的。可以用单元测试覆盖的代码就不应该依赖于 Web 页面测试，且不说页面测试更加耗时耗力，这种方式还无法自动化，更别提重复性了，因此 Web 页面测试应该仅限于页面的层次，例如

JSP、CSS、JavaScript 的修改，其他代码修改（比如数据访问），请编写单元测试。

传统的 Web 测试方法要求我们编译、测试、打包及部署，这往往会消耗数 10 秒至数分钟的时间，jetty-maven-plugin 能够帮助我们节省时间，它能够周期性地检查项目内容，发现变更后自动更新到内置的 Jetty Web 容器中，换句话说，它帮我们省去了打包和部署的步骤。jetty-maven-plugin 默认就很好地支持了 Maven 的项目目录结构，在通常情况下，我们只需要直接在 IDE 中修改源码，IDE 能够执行自动编译，jetty-maven-plugin 发现编译后的文件变化后，自动将其更新到 Jetty 容器，这时我们就可以直接测试 Web 页面了。

使用 jetty-maven-plugin 十分简单，我们指定该插件的坐标，并且稍加配置即可，见代码清单 4-13：

代码清单 4-13：配置 jetty-maven-plugin

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>7.1.6.v20100715</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <webAppConfig>
      <contextPath>/test</contextPath>
    </webAppConfig>
  </configuration>
</plugin>
```

jetty-maven-plugin 并不是官方的 Maven 插件，它的 groupId 是 org.mortbay.jetty，上述代码中我们使用了 Jetty 7 的最新版本。在该插件的配置中，scanIntervalSeconds 顾名思义表示该插件扫描项目变更的时间间隔，这里的配置是每隔 10 秒，需要注意的是，如果不进行配置，该元素的默认值是 0，表示不扫描，用户也就失去了所谓的自动化热部署的功能。上述代码中 webappConfig 元素下的 contextPath 即表示项目部署后的 context path，例如这里的值为 /test，那么用户就可以通过 <http://hostname:port/test/> 访问该应用。

下一步启动 jetty-maven-plugin，不过在这之前需要对 settings.xml 做个微小的修改，本书 7.7.4 小节介绍过，默认情况下，只有 org.apache.maven.plugins 和 org.codehaus.mojo 两个 groupId 下的插件才支持简化的命令行调用，即我们可以运行 `mvn help:system`，但 `mvn jetty:run` 就不行了，因为 maven-help-plugin 的

groupId 是 org.apache.maven.plugins，而 jetty-maven-plugin 的 groupId 是 org.mortbay.jetty，为了能在命令行直接运行 mvn jetty:run，用户需要配置 settings.xml 如下：

```
<settings>
  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
  </pluginGroups>
  ...
</settings>
```

现在可以运行如下命令启动 jetty-maven-plugin：

```
$ mvn jetty:run
```

jetty-maven-plugin 会启动 Jetty，并且默认监听本地的 8080 端口，并将当前项目部署到容器中，同时它还会根据用户配置扫描代码改动。

如果用户希望使用其他端口，可以添加 jetty.port 参数，如

```
$ mvn jetty:run -Djetty.port=9999
```

现在用户就可以打开浏览器通过地址 <http://localhost:9999/test/> 测试应用了，要停止 Jetty，只需要在命令行输入 Ctrl + C 即可。

启动 Jetty 之后，用户可以在 IDE 中修改各类文件，如 JSP、HTML、CSS、JavaScript 甚至是 Java 类，只要不是修改类名、方法名等较大的操作，jetty-maven-plugin 都能够扫描到变更并正确地将变化更新至 Web 容器中，这无疑在很大程度上帮助用户实现快速开发和测试。

上面的内容仅仅展示了 jetty-maven-plugin 最核心的配置点，如果有需要，用户还可以自定义 web.xml 的位置、项目 class 文件的位置、web 资源目录的位置等等信息。用户还能够以 WAR 包的方式部署项目，甚至在 Maven 的生命周期中嵌入 jetty-maven-plugin，例如先启动 Jetty 容器并部署项目，然后执行一些集成测试，最后停止容器。有兴趣进一步研究的读者可以访问该页面：http://wiki.eclipse.org/Jetty/Feature/Jetty_Maven_Plugin。

4.5 使用 Cargo 实现自动化部署

Cargo 是一组帮助用户操作 Web 容器的工具，它能够帮助用户实现自动化部署，而且它几乎支持所有的 Web 容器，如 Tomcat、JBoss、Jetty 和 Glassfish 等等。Cargo 通过 cargo-maven2-plugin 提供了 Maven 集成，Maven 用户可以使用

该插件将 Web 项目部署到 Web 容器中。虽然 cargo-maven2-plugin 和 jetty-maven-plugin 的功能看起来很相似，但它们的目的是不同的，jetty-maven-plugin 主要用来帮助日常快速开发和测试，而 cargo-maven2-plugin 主要服务于自动化部署，例如专门的测试人员只需要一条简单的 Maven 命令，就可以构建项目并部署到 Web 容器中，然后进行功能测试。本节以 Tomcat 6 为例，介绍如何自动化地将 Web 应用部署至本地或远程 Web 容器中。

4.5.1 部署至本地Web容器

Cargo 支持两种本地部署的方式，分别为 standalone 模式和 existing 模式。在 standalone 模式中，Cargo 会从 Web 容器的安装目录复制一份配置到用户指定的目录，然后在此基础上部署应用，每次重新构建的时候，这个目录都会被清空，所有配置被重新生成。而在 existing 模式中，用户需要指定现有的 Web 容器配置目录，然后 Cargo 会直接使用这些配置并将应用部署到其对应的位置。

代码清单 4-14 展示了 standalone 模式的配置样例：

代码清单 4-14：使用 standalone 模式部署应用至本地 Web 容器

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <home>D:\cmd\apache-tomcat-6.0.29</home>
    </container>
    <configuration>
      <type>standalone</type>
      <home>${project.build.directory}/tomcat6x</home>
    </configuration>
  </configuration>
</plugin>
```

cargo-maven2-plugin 的 groupId 是 org.codehaus.cargo，这不属于官方的两个 Maven 插件 groupId，因此用户需要将其添加到 settings.xml 的 pluginGroup 元素中以方便命令行调用。

上述 cargo-maven2-plugin 的具体配置包括了 container 和 configuration 两个元素，configuration 的子元素 type 表示部署的模式（这里是 standalone），与之对应的，configuration 的 home 子元素表示复制容器配置到什么位置，这里的值

为 `${project.build.directory}/tomcat6x`，表示构建输出目录，即 `target/` 下的 `tomcat6x` 子目录。`container` 元素下的 `containerId` 表示容器的类型，`home` 元素表示容器的安装目录。基于该配置，Cargo 会从 `D:\cmd\apache-tomcat-6.0.29` 目录下复制配置到当前项目的 `target/tomcat6x/` 目录下。

现在，要让 Cargo 启动 Tomcat 并部署应用，只需要运行：

```
$ mvn cargo:start
```

以 `account-web` 为例，现在我就可以直接访问如下地址的账户注册页面了。

默认情况 Cargo 会让 Web 容器监听 8080 端口，我们可以通过修改 Cargo 的 `cargo.servlet.port` 属性来改变这一配置，如代码清单 4-15：

代码清单 4-15：更改 Cargo 的 Servlet 监听端口

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <home>D:\cmd\apache-tomcat-6.0.29</home>
    </container>
    <configuration>
      <type>standalone</type>
      <home>${project.build.directory}/tomcat6x</home>
      <properties>
        <cargo.servlet.port>8081</cargo.servlet.port>
      </properties>
    </configuration>
  </configuration>
</plugin>
```

要将应用直接部署到现有的 Web 容器下，需要配置 Cargo 使用 `existing` 模式，如代码清单 4-16：

代码清单 4-16：使用 `existing` 模式部署应用至本地 Web 容器

```
<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
```

```

        <containerId>tomcat6x</containerId>
        <home>D:\cmd\apache-tomcat-6.0.29</home>
    </container>
    <configuration>
        <type>existing</type>
        <home>D:\cmd\apache-tomcat-6.0.29</home>
    </configuration>
</configuration>
</plugin>

```

上述代码中 configuration 元素的 type 子元素的值为 existing，而对应的 home 子元素表示现有的 Web 容器目录，基于该配置运行 mvn cargo:start 之后，便能够在 Tomcat 的 webapps 子目录看到被部署的 Maven 项目。

4.5.2 部署至远程Web容器

除了让 Cargo 直接管理本地 Web 容器然后部署应用之外，我们也可以让 Cargo 部署应用至远程的正在运行的 Web 容器中，当然，前提是拥有该容器的相应管理员权限，相关配置如代码清单 4-17 所示：

代码清单 4-17：部署应用至远程 Web 容器

```

<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <container>
      <containerId>tomcat6x</containerId>
      <type>remote</type>
    </container>
    <configuration>
      <type>runtime</type>
      <properties>
        <cargo.remote.username>admin</cargo.remote.username>
        <cargo.remote.password>admin123</cargo.remote.password>

        <cargo.tomcat.manager.url>http://localhost:8080/manager</cargo.tomcat.manager.url>
      </properties>
    </configuration>
  </configuration>
</plugin>

```

对于远程部署的方式来说，container 元素的 type 子元素的值必须为 remote，如果不显示指定，Cargo 会使用默认值 installed，并寻找对应的容器安装目录或者安装包，对于远程部署方式来说，安装目录或者安装包是不需要的。上述代码中 configuration 的 type 子元素值为 runtime，表示既不使用独立的容器配置，也不使用本地现有的容器配置，而是依赖于一个已运行的容器。properties 元素用来声明一些容器热部署相关的配置，例如这里的 Tomcat 6 就需要提供用户名、密码、以及管理地址。需要注意的是，这部分配置元素对于所有容器来说不是一致的，读者需要查阅对应的 Cargo 文档。

有了上述配置后，我们就可以让 Cargo 部署应用了，运行命令如下：

```
$ mvn cargo:redeploy
```

如果容器中已经部署了当前应用，Cargo 会首先将其卸载，然后再重新部署。

由于自动化部署本身就不是简单的事情，再加上 Cargo 要兼容各种不同类型的 Web 容器，因此 cargo-maven2-plugin 的相关配置会显得相对复杂，这个时候完善的文档就显得尤为重要，如果读者想进一步了解 Cargo，可访问 <http://cargo.codehaus.org/Maven2+plugin>。

4.6 小结

本章介绍的是用 Maven 管理 Web 项目，因此首先讨论了 Web 项目的基本结构，然后分析实现了本书背景案例的最后两个模块：account-service 和 account-web，其中后者是一个典型的 Web 模块。开发 Web 项目的时候，大家往往会使用热部署来实现快速的开发和测试，jetty-maven-plugin 可以帮助实现这一目标。本章最后讨论的是自动化部署，这一技术的主角是 Cargo，有了它，我们可以让 Maven 自动部署应用至本地和远程 Web 容器中。



Maven 实战

----基于 Maven 3

专题策划：张凯峰

责任编辑：霍泰稳

美术编辑：胡伟红

本迷你书主页为

<http://www.infoq.com/cn/minibooks/maven-in-action>

本书属于 InfoQ 企业软件开发丛书。

如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

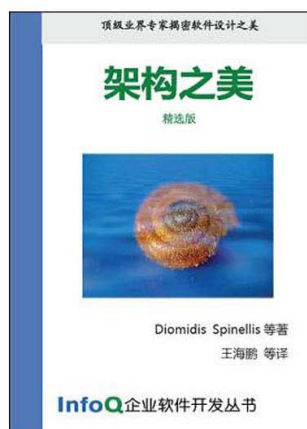
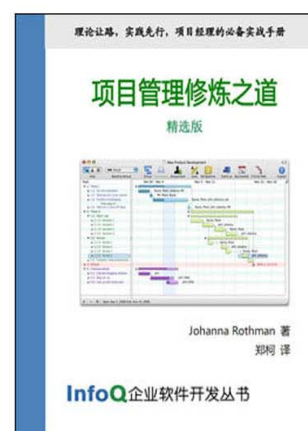
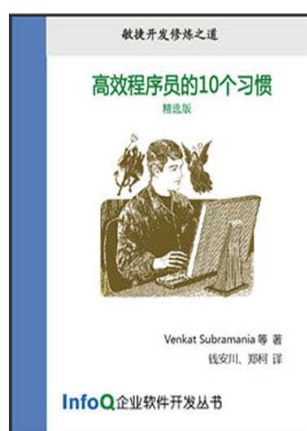
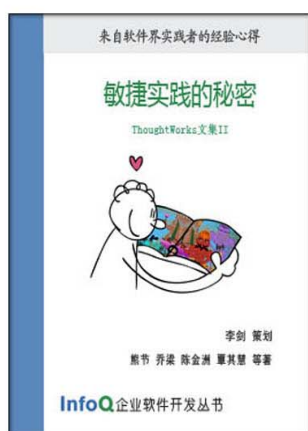
本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译等，请联系 editors@cn.infoq.com。

InfoQ企业软件开发丛书

欢迎免费下载



商务合作: sales@cn.infoq.com

读者反馈/内容提供: editors@cn.infoq.com