

# Maven 实战

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。

[《Maven 实战》](#)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。



## [《Maven 实战》](#)

### 作者简介：

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。

### 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 Consuting 首席咨询顾问，《Java Power Tools》（O'Reilly 出版）作者

## 目 录

1. 《Maven 实战》前言
2. 第 1 章 Maven 简介
3. 1.1 何为 Maven
4. 1.2 为什么需要 Maven
5. 1.3 Maven 与极限编程
6. 第 2 章 Maven 的安装和配置
7. 2.1 在 Windows 上安装 Maven
8. 2.2 在基于 Unix 的系统上安装 Maven
9. 2.3 安装目录分析
- 10.2.4 设置 HTTP 代理
- 11.2.5 安装 m2eclipse
- 12.2.6 安装 NetBeans Maven 插件
- 13.2.7 Maven 安装最佳实践
- 14.2.8 小结
- 15.第 3 章 Maven 使用入门
- 16.3.1 编写 POM
- 17.3.2 编写主代码
- 18.3.3 编写测试代码
- 19.3.4 打包和运行
- 20.3.5 使用 Archetype 生成项目骨架
- 21.3.6 m2eclipse 简单使用
- 22.3.7 NetBeans Maven 插件简单使用
- 23.3.8 小结

## 1.1 《Maven 实战》前言 TOP

### 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。

## 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 章是对 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。

## 本书代码下载

大家可以从我的网站下载本书的代码：<http://www.juvenxu.com/mvn-in-action/>，也可以通过我的网站与我取得联系，欢迎大家与我交流任何关于本书的问题和关于 Maven 的问题。

## 咖啡与工具

本书相当一部分的内容是在苏州十全街边的 Solo 咖啡馆完成的，老板 Yin 亲手烘焙咖啡豆、并能做出据说是苏州最好的咖啡，这小桥流水畔的温馨小屋能够帮我消除紧张和焦虑，和 Yin 有一句没一句的聊天也是相当的轻松。Yin 还教会了我如何自己研磨咖啡豆、手冲滴率咖啡，让我能够每天在家里也能享受香气四溢的新鲜咖啡。

本书的书稿是使用 Git 和 Unfuddle（<http://unfuddle.com/>）进行管理的，书中的大量截图是通过 Jing（<http://www.techsmith.com/jing/>）制作的。

JuvenXu

2010 年 10 月于苏州 Solo 咖啡

## 致谢

感谢费晓峰，是你最早让我学习使用 Maven，并在我开始学习的过程中给予了不少帮助。

感谢 Maven 开源社区特别是 Maven 的创立者 Jason Van Zyl，是你们一起创造了如此优秀的开源工具，造福了全世界这么多的开发人员。

感谢我的家人，一年来，我的大部分原来属于你们的业余时间都给了这本书，感谢你们的理解和支持。

感谢二少、Garin、Sutra、JTux、红人、linux\_china、Chris、Jdonee、zc0922、还有很多 Maven 中文社区的朋友，你们给了本书不少建议，并在我写作过程中不断鼓励我和支持我，你们是我写作最大的动力之一。

最后感谢本书的策划编辑杨福川和曾珊，我从你们身上学到了很多，你们是最专业的、最棒的。

## 1.2 第 1 章 Maven 简介 TOP

1.1 何为 Maven/2

1.2 为什么需要 Maven/4

1.3 Maven 与极限编程/7

1.4 被误解的 Maven/8

1.5 小结/9

### 1.3 1.1 何为 Maven TOP

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

#### 1.4 1.1.1 何为构建

不管你是否意识到，构建（build）是每一位程序员每天都在做的工作。早上来到公司，我们做的第一件事情就是从源码库签出最新的源码，然后进行单元测试，如果发现失败的测试，会找相关的同事一起调试，修复错误代码。接着回到自己的工作上来，编写自己的单元测试及产品代码，我们会感激 IDE 随时报出的编译错误提示。

忙到午饭时间，代码编写得差不多了，测试也通过了，开心地享用午餐，然后休息。下午先在昏昏沉沉中开了个例会，会议结束后喝杯咖啡继续工作。刚才在会上经理要求看测试报告，于是找了相关工具集成进 IDE，生成了像模像样的测试覆盖率报告，接着发了一封电子邮件给经理，松了口气。谁料 QA 小组又发过来了几个 bug，没办法，先本地重现再说，于是熟练地用 IDE 生成了一个 WAR 包，部署到 Web 容器下，启动容器。看到熟悉的界面了，遵循 bug 报告，一步步重现了 bug……快下班的时候，bug 修好了，提交代码，通知 QA 小组，在愉快中结束了一天的工作。

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

## 1.5 1.1.2 Maven 是优秀的构建工具

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

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

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

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

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

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

## 1.6 1.1.3 Maven 不仅仅是构建工具

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

在这个开源的年代里，几乎任何 Java 应用都会借用一些第三方的开源类库，这些类库都可通过依赖的方式引入到项目中来。随着依赖的增多，版本不一致、版本冲突、依赖臃肿等问题都会接踵而来。手工解决这些问题是十分枯燥的，幸运的是 Maven 提供了一个优秀的解决方案，它通过一个坐标系统准确地定位每一个构件（artifact），也就是通过一组坐标 Maven 能够找到任何一个 Java



类库（如 jar 文件）。Maven 给这个类库世界引入了经纬，让它们变得有秩序，于是我们可以借助它来有序地管理依赖，轻松地解决那些繁杂的依赖问题。

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

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

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

## 1.7 1.2 为什么需要 Maven TOP

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

### 1.8 1.2.1 组装 PC 和品牌 PC

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

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

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

## 1.9 1.2.2 IDE 不是万能的

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

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

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

## 1.10 1.2.3 Make

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

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

Java 代码 

1. `<SPAN style="FONT-SIZE: small">TARGET... : PREREQUISITE...`
2. `COMMAND`
3. `...`
4. `...`
5. `</SPAN>`

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

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

## 1.11 1.2.4 Ant

Ant 不是指蚂蚁，而是意指“另一个整洁的工具”（Another Neat Tool），它最早用来构建著名的 Tomcat，其作

者 James Duncan Davidson 创作它的动机就是因为受不了 Makefile 的语法格式。我们可以将 Ant 看成是一个 Java 版本的 Make，也正因为使用了 Java，Ant 是跨平台的。此外，Ant 使用 XML 定义构建脚本，相对于 Makefile 来说，这也更加友好。

与 Make 类似，Ant 有一个构建脚本 build.xml，如下所示：

```
<?xml version="1.0"?>

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

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

        <mkdir dir="classes"/>

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

    </target>

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

        <jar destfile="hello.jar">

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

            <manifest>

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

            </manifest>

        </jar>

    </target>

</project>
```

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

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



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

## 1.12 1.2.5 不重复发明轮子

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

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

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

小张隐隐地觉得这些依赖需要一个框架或者系统来进行管理。

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

项目经理发话了：“兄弟们，新项目来了，小张，你来写 Ant 脚本！”

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

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

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

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

## 1.13 1.3 Maven 与极限编程 TOP

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

首先看一下 Maven 如何帮助 XP 团队实现一些核心价值：

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

此外，Maven 更能无缝地支持或者融入到一些主要的 XP 实践中：

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

上述这些实践并非只在 XP 中适用。事实上，除了其他敏捷开发方法如 SCRUM 之外，几乎任何软件开发方法都能借鉴这些实践。也就是说，Maven 几乎能够很好地支持任何软件开发方法。

例如，在传统的瀑布模型开发中，项目依次要经历需求开发、分析、设计、编码、测试和集成发布阶段。从设计和编码阶段开始，就可以使用 Maven 来建立项目的构建系统。在设计阶段，也完全可以针对设计开发测试用例，然后再编写代码来满足这些测试用例。然而，有了自动化构建系统，我们可以节省很多手动的测试时间。此外，尽早地使用构建系统集成团队的代码，对项目也是百利而无一害。最后，Maven 还能帮助我们快速地发布项目。

## 1.14 第 2 章 Maven 的安装和配置 TOP

第 1 章介绍了 Maven 是什么，以及为什么要使用 Maven，我们将从本章实际开始实际接触 Maven。本章首先将介绍如何在主流的操作系统下安装 Maven，并详细解释 Maven 的安装文件；其次还会介绍如何在主流的 IDE 中集成 Maven，以及 Maven 安装的最佳实践。

2.1 在 Windows 上安装 Maven

2.2 在基于 Unix 的系统上安装 Maven

2.3 安装目录分析

- 2.4 设置 HTTP 代理
- 2.5 安装 m2eclipse
- 2.6 安装 NetBeans Maven 插件
- 2.7 Maven 安装最佳实践
- 2.8 小结

## 1.15 2.1 在 Windows 上安装 Maven TOP

### 1.16 2.1.1 检查 JDK 安装

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

```
C:\Users\Juven Xu>echo %JAVA_HOME%
```

```
C:\Users\Juven Xu>java -version
```

结果如图 2-1 所示：



```
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 2-1 Windows 中检查 Java 安装

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

### 1.17 2.1.2 下载 Maven

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

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

快点尝试下 Maven 3 吧！

### 1.18 2.1.3 本地安装

将安装文件解压到你指定的目录中，如：

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

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

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



图 2-2 Windows 中系统环境变量配置

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

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

```
C:\Users\Juven Xu>echo %M2_HOME%
```

```
C:\Users\Juven Xu>mvn -v
```

运行结果如图 2-3 所示：

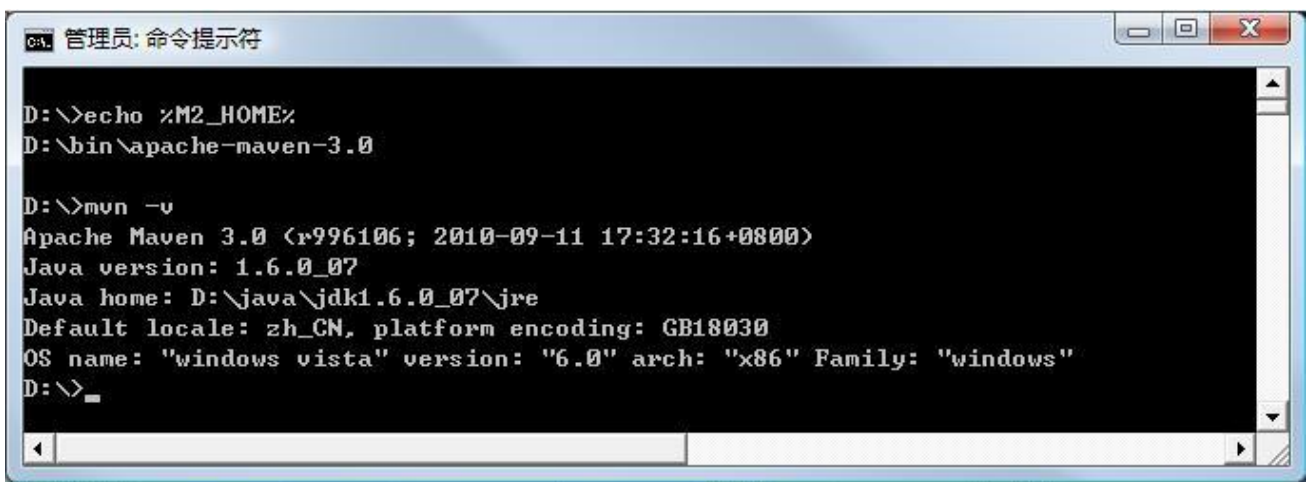


图 2-3 Windows 中检查 Maven 安装

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

## 1.19 2.1.4 升级 Maven

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

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



## 1.20 2.2 在基于 Unix 的系统上安装 Maven TOP

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

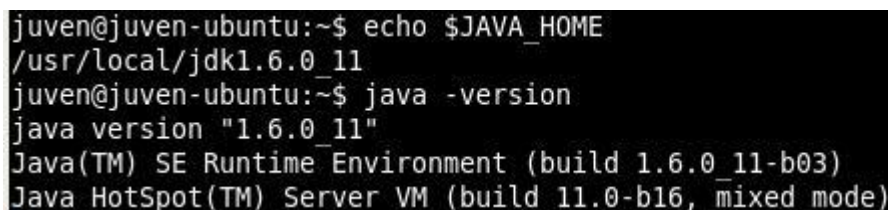
### 1.21 2.2.1 下载和安装

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

```
juven@juven-ubuntu:~$ echo $JAVA_HOME
```

```
juven@juven-ubuntu:~$ java -version
```

运行结果如图 2-4 所示：



```
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 2-4 Linux 中检查 Java 安装

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

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

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

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

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



```
juven@juven-ubuntu:bin$ export M2_HOME=/home/juven/bin/apache-maven
juven@juven-ubuntu:bin$ export PATH=$PATH:$M2_HOME/bin
```

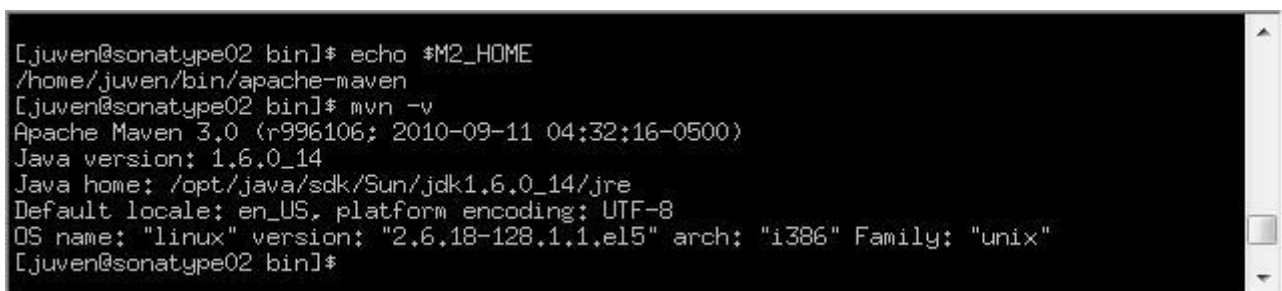
一般来说，需要将这两行命令加入到系统的登录 shell 脚本中去，以我现在的 Ubuntu 8.10 为例，编辑 ~/.bashrc 文件，添加这两行命令。这样，每次启动一个终端，这些配置就能自动执行。

至此，安装完成，我们可以运行以下命令检查 Maven 安装：

```
juven@juven-ubuntu:bin$ echo $M2_HOME
```

```
juven@juven-ubuntu:bin$ mvn -version
```

运行结果如图 2-5 所示：



```
[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 2-5 Linux 中检查 Maven 安装

## 1.22 2.2.2 升级 Maven

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

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

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

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

## 1.23 2.3 安装目录分析 TOP

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

### 1.24 2.3.1 M2\_HOME

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

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

- **Bin:** 该目录包含了 mvn 运行的脚本，这些脚本用来配置 Java 命令，准备好 classpath 和相关的 Java 系统属性，然后执行 Java 命令。其中 *mvn* 是基于 UNIX 平台的 shell 脚本，*mvn.bat* 是基于 Windows 平台的 bat 脚本。在命令行输入任何一条 mvn 命令时，实际上就是在调用这些脚本。该目录还包含了 *mvnDebug* 和 *mvnDebug.bat* 两个文件，同样，前者是 UNIX 平台的 shell 脚本，后者是 windows 的 bat 脚本。那么 mvn 和 mvnDebug 有什么区别和关系呢？打开文件我们就可以看到，两者基本是一样的，只是 mvnDebug 多了一条 MAVEN\_DEBUG\_OPTS 配置，作用就是在运行 Maven 时开启 debug，以便调试 Maven 本身。此外，该目录还包含 *m2.conf* 文件，这是 classworlds 的配置文件，稍微会介绍 classworlds。
- **Boot:** 该目录只包含一个文件，以 maven 3.0 为例，该文件为 *plexus-classworlds-2.2.3.jar*。plexus-classworlds 是一个类加载器框架，相对于默认的 java 类加载器，它提供了更丰富的语法以方便配置，Maven 使用该框架加载自己的类库。更多关于 classworlds 的信息请参考 <http://classworlds.codehaus.org/>。对于一般的 Maven 用户来说，不必关心该文件。
- **Conf:** 该目录包含了一个非常重要的文件 *settings.xml*。直接修改该文件，就能在机器上全局地定制 Maven 的行为。一般情况下，我们更偏向于复制该文件至 *~/.m2/* 目录下（这里 *~* 表示用户目录），然后修改该文件，在用户范围定制 Maven 的行为。本书的后面将会多次提到该 *settings.xml*，并逐步分析其中的各个元素。
- **Lib:** 该目录包含了所有 Maven 运行时需要的 Java 类库，Maven 本身是分模块开发的，因此用户能看到诸如 *mavn-core-3.0.jar*、*maven-model-3.0.jar* 之类的文件，此外这里还包含一些 Maven 用到的第三方依赖如 *common-cli-1.2.jar*、*google-collection-1.0.jar* 等等。（对于 Maven 2 来说，该目录只包含一个如 *maven-2.2.1-uber.jar* 的文件原本各为独立 JAR 文件的 Maven 模块和第三方类库都被拆解后重新合并到了这个 JAR 文件中）。可以说，这个 lib 目录就是真正的 Maven。关于该文件，还有一点值得一提的是，用户可以在这个目录中找到 Maven 内置的超级 POM，这一点在 8.5 小节详细解释。其他：*LICENSE.txt* 记录了 Maven 使用的软件许可证 Apache License Version 2.0；*NOTICE.txt* 记录了 Maven 包含的第三方软件；而 *README.txt* 则包含了 Maven 的简要介绍，包括安装需求及如何安装的简要指令等等。

## 1.25 2.3.2 ~/.m2

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

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

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

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

## 1.26 2.4 设置 HTTP 代理 TOP

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

首先确认自己无法直接访问公共的 Maven 中央仓库，直接运行命令 **ping repol.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+j**，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。

## 1.27 2.5 安装 m2eclipse TOP

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

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

现在我以 Eclipse 3.6 为例逐步讲解 m2eclipse 的安装。启动 Eclipse 之后，在菜单栏中选择 **Help**，然后选择

**Install New Software...**，接着你会看到一个 Install 对话框，点击 **Work with:** 字段边上的 **Add 按钮**，你会得到一个新的 Add Repository 对话框，在 **Name** 字段中输入 *m2e*，**Location** 字段中输入 <http://m2eclipse.sonatype.org/sites/m2e>，然后点击 **OK**。Eclipse 会下载 m2eclipse 安装站点上的资源信息。等待资源载入完成之后，我们再将全部展开，就能看到图 2-6 所示的界面：

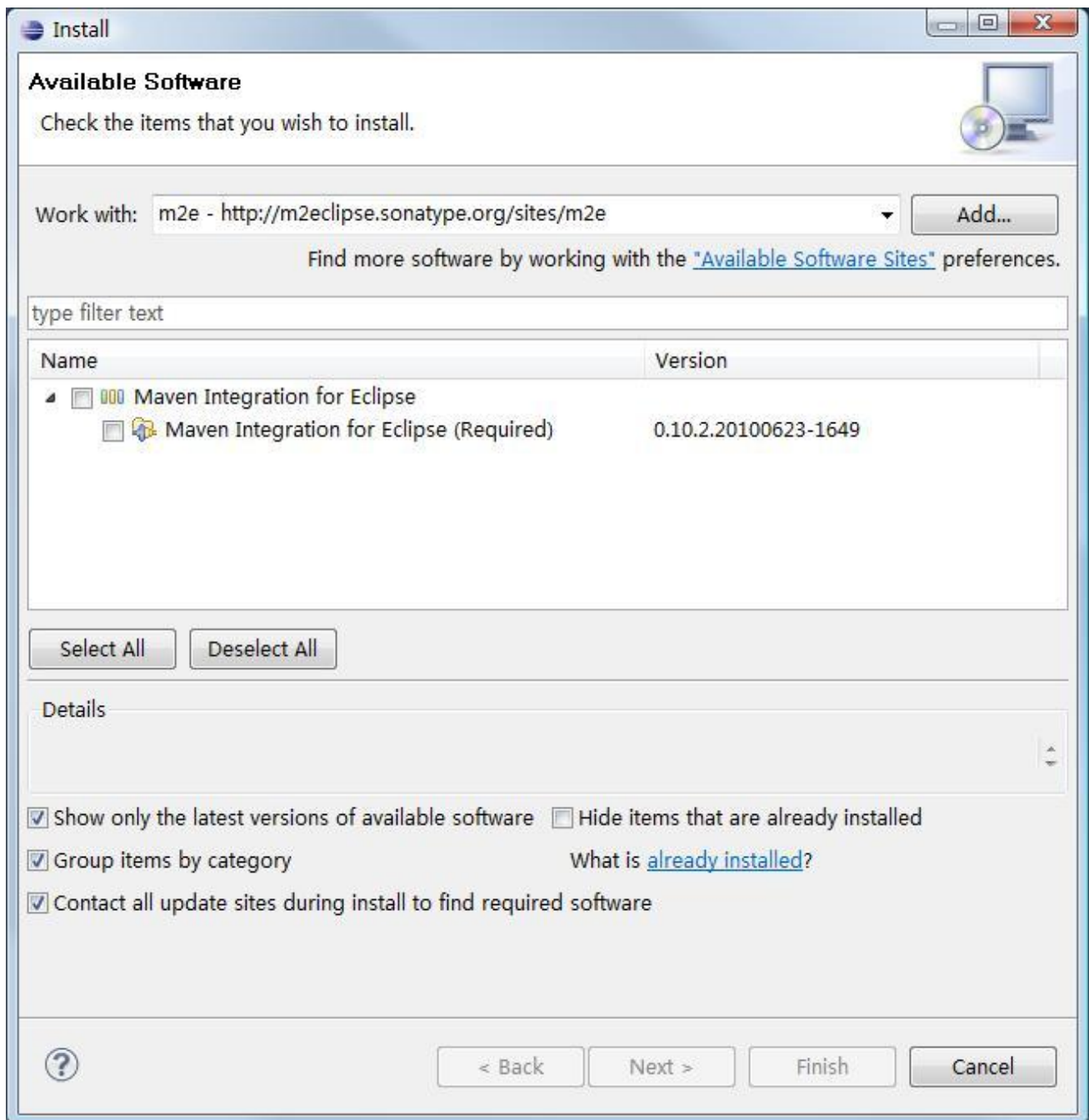


图 2-6 m2eclipse 的核心安装资源列表

如图显示了 m2eclipse 的核心模块 Maven Integration for Eclipse (Required)，选择后点击 **Next >**，Eclipse 会自动计算模块间依赖，然后给出一个将被安装的模块列表，确认无误后，继续点击 **Next >**，这时我们会看到许可证

信息, m2eclipse 使用的开源许可证是 Eclipse Public License v1.0, 选择 **I accept the terms of the license agreements**, 然后点击 **Finish**, 接着就耐心等待 Eclipse 下载安装这些模块, 如图 2-7 所示:

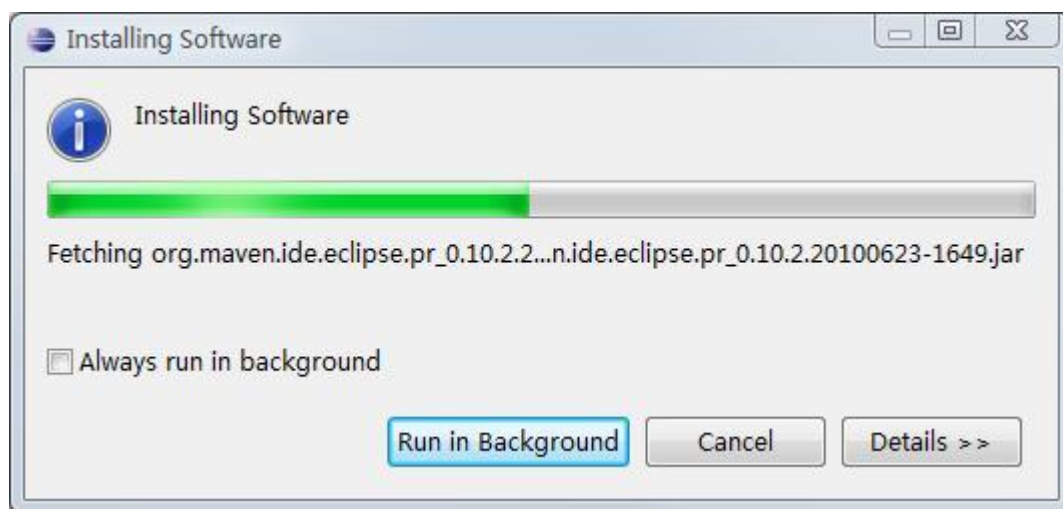


图 2-7: m2eclipse 安装进度

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



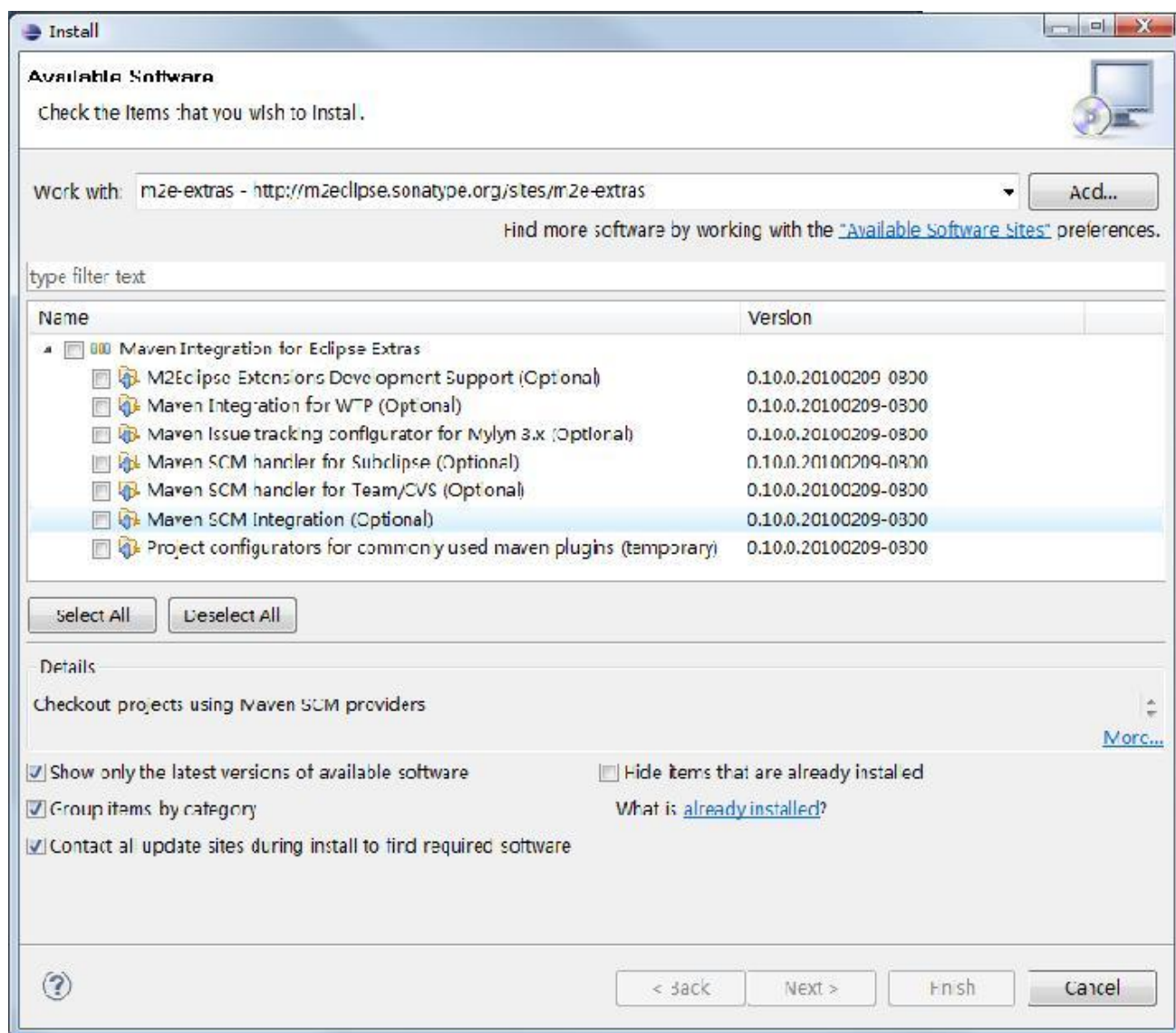


图 2-8: m2eclipse 的额外组件安装资源列表

下面简单解释一下这些组件的用途:

## 1. 重要的

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

## 2. 不重要的

- **Maven issue tracking configurator for Mylyn 3.x (Optional):** 该模块能够帮助我们使用 POM 中的缺陷跟踪系统信息连接 Mylyn 至服务器。

- **Maven SCM handler for Team/CVS (Optional):** 该模块帮助我们 从 CVS 服务器签出 Maven 项目，如果你还在使用 CVS，就需要安装它。
- **Maven Integration for WTP (Optional):** 使用该模块可以让 Eclipse 自动读取 POM 信息并配置 WTP 项目。
- **M2eclipse Extensions Development Support (Optional):** 用来支持扩展 m2eclipse，一般用户不会用到。
- **Project configurators for commonly used maven plugins (temporary):** 一个临时的组件，用来支持一些 Maven 插件与 Eclipse 的集成，建议安装。

读者可以根据自己的需要安装相应组件，具体步骤不再赘述。

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

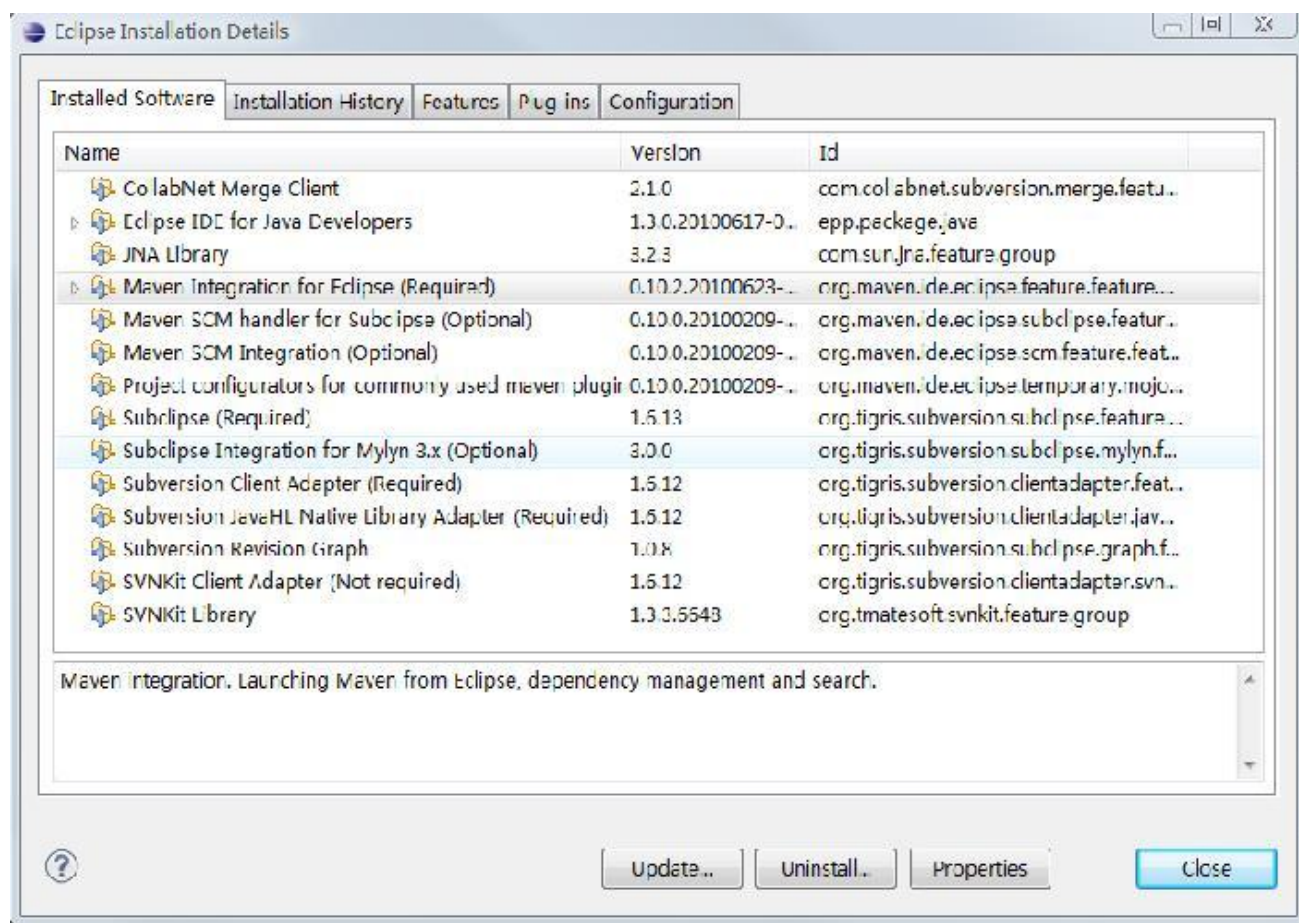


图 2-9m2eclipse 安装结果

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

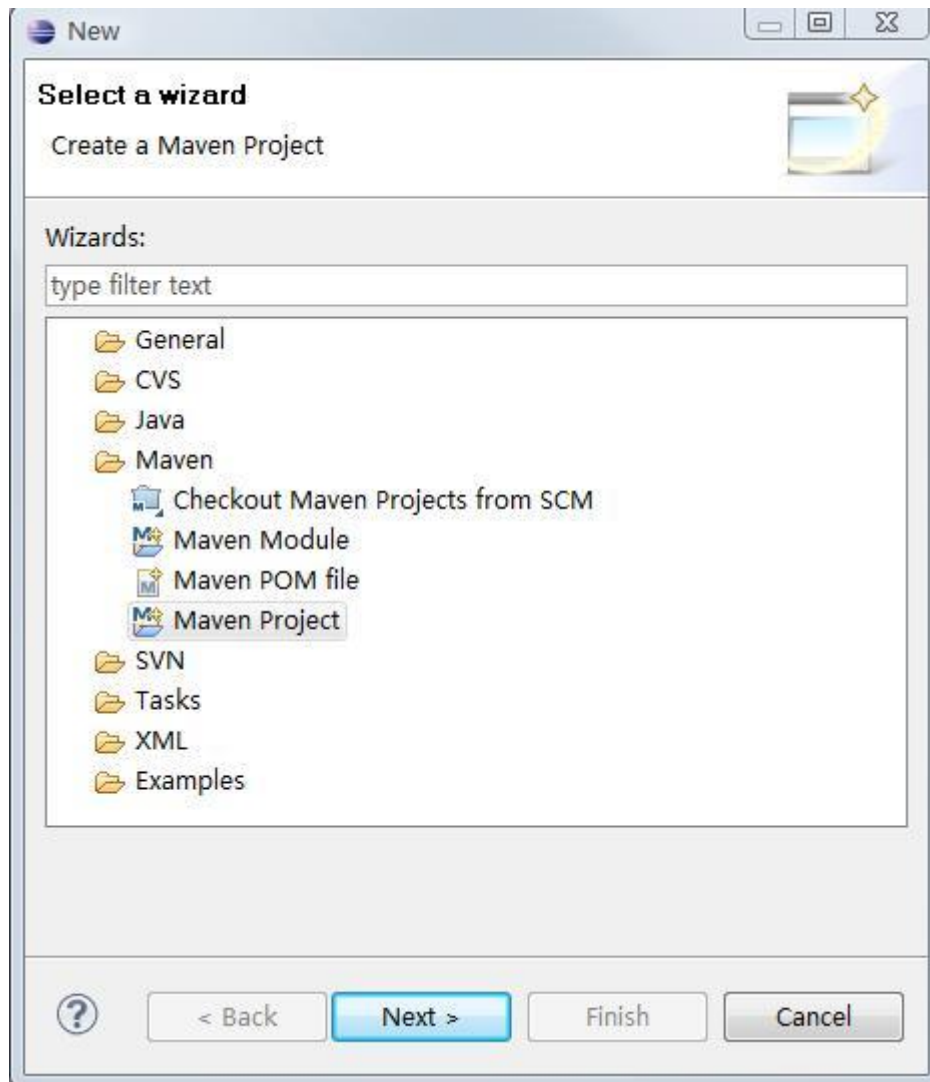


图 2-10 Eclipse 中创建 Maven 项目向导

如果一切正常，说明 m2eclipse 已经正确安装了。

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

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

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

```
--launcher.XXMaxPermSize
```

```
256m
```

```
-vm
```

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

-vmargs

-Dosgi.requiredJavaVersion=1.5

-Xms128m

-Xmx256m

## 1.28 2.6 安装 NetBeans Maven 插件 [↑TOP](#)

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

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

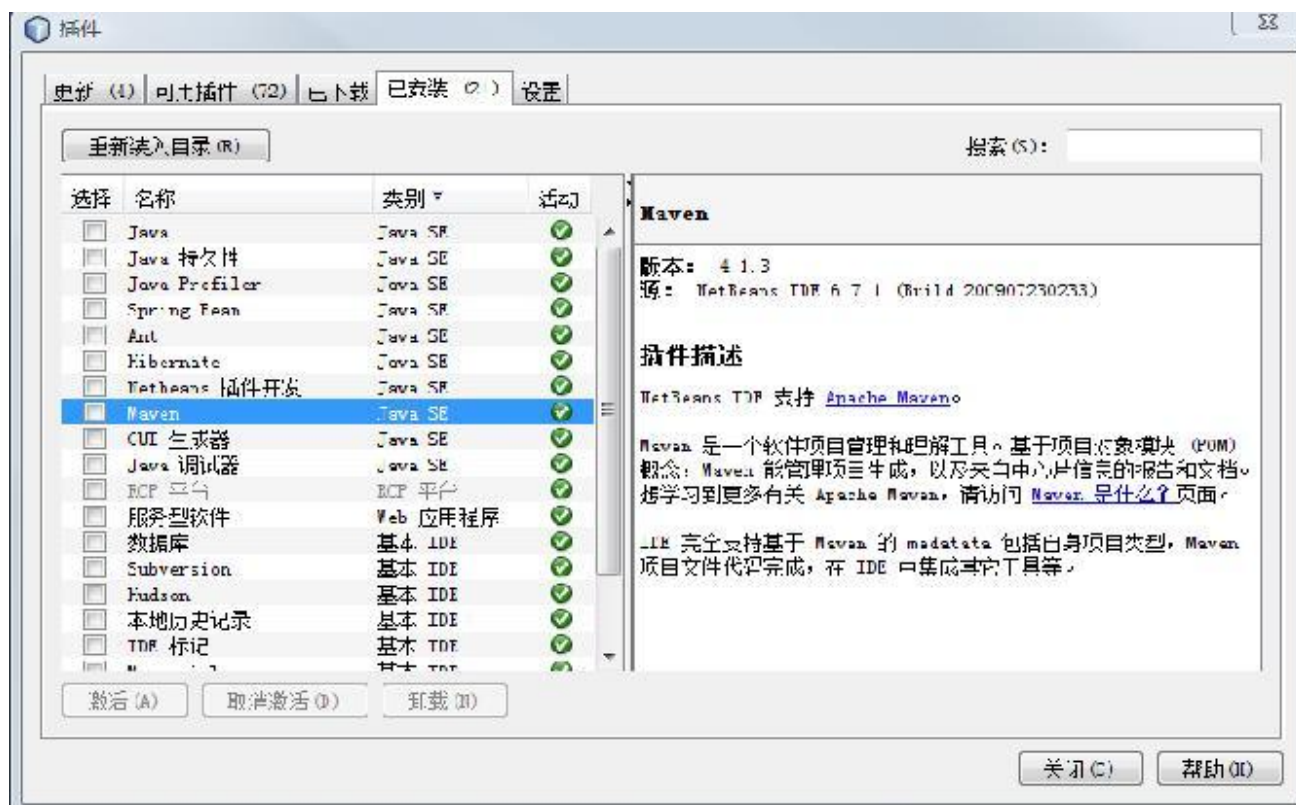


图 2-11 已安装的 NetBeans Maven 插件

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



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



图 2-12 安装 NetBeans Maven 插件

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

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



图 2-13 NetBeans 中创建 Maven 项目向导

如果你能看到类似的对话框，说明 NetBeans Maven 已经正确安装了。

## 1.29 2.7 Maven 安装最佳实践 [TOP](#)

本节介绍一些在安装 Maven 过程中不是必须的，但十分有用的实践。

### 1.30 2.7.1 设置 MAVEN\_OPTS 环境变量

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

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

### 1.31 2.7.2 配置用户范围 settings.xml

Maven 用户可以选择配置 `$M2_HOME/conf/settings.xml` 或者 `~/.m2/settings.xml`。前者是全局范围的，整台机器上的所有用户都会直接受到该配置的影响，而后者是用户范围的，只有当前用户才会受到该配置的影响。

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

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

### 1.32 2.7.3 不要使用 IDE 内嵌的 Maven

无论是 Eclipse 还是 NetBeans，当我们集成 Maven 时，都会安装上一个内嵌的 Maven，这个内嵌的 Maven 通常会比较新，但不一定很稳定，而且往往也会和我们在命令行使用的 Maven 不是同一个版本。这里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，这是我们不希望看到的。因此，我们应该在 IDE 中配置 Maven 插件时使用与命令行一致的 Maven。

在 m2eclipse 环境中，点击菜单栏中的 **Windows**，然后选择 **Preferences**，在弹出的对话框中，展开左边的 **Maven** 项，选择 **Installation** 子项，在右边的面板中，我们能够看到有一个默认的 **Embedded Maven** 安装被选中了，点击 **Add...** 然后选择我们的 Maven 安装目录 `M2_HOME`，添加完毕之后选择这一个外部的 Maven，如图 2-14 所示：

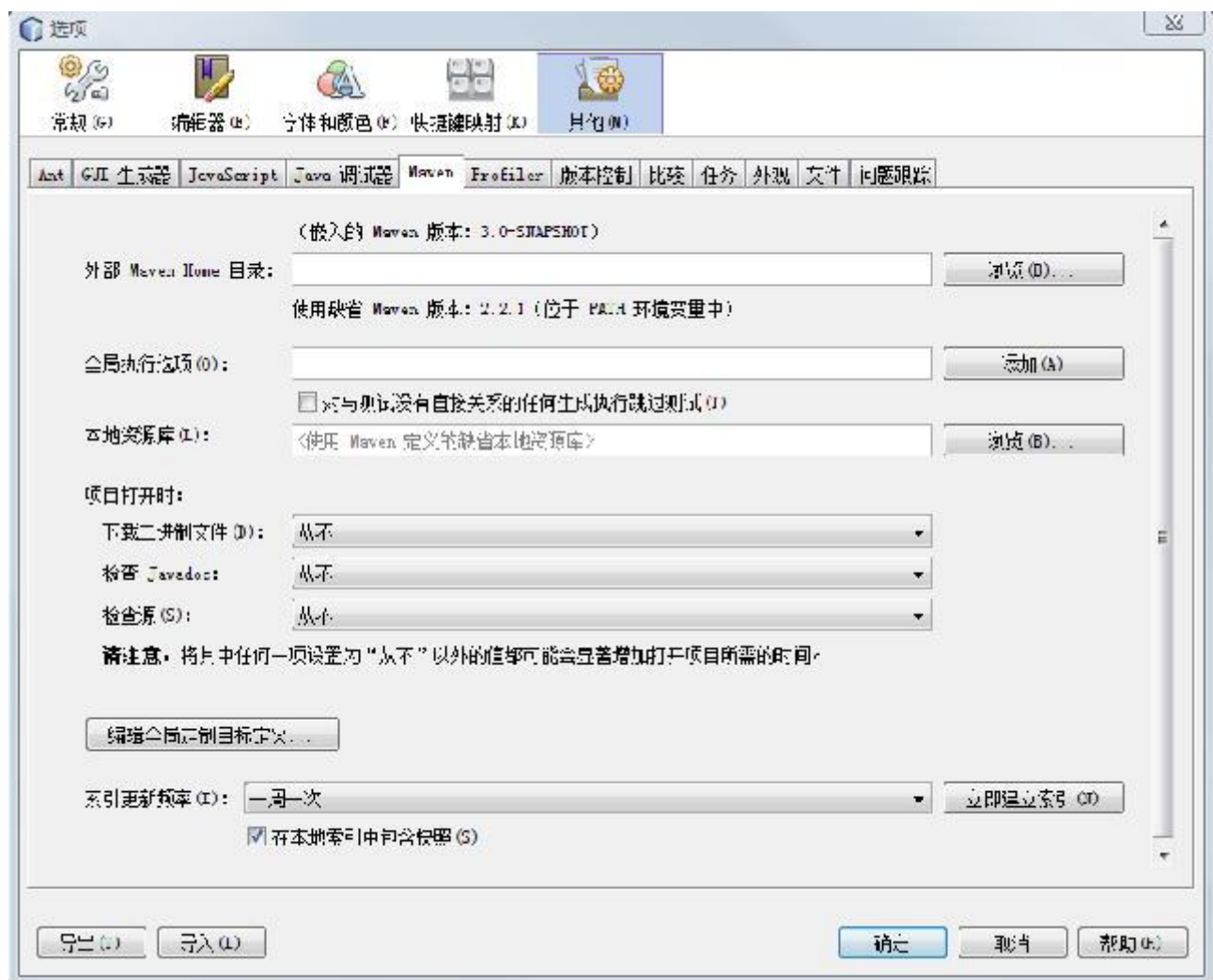


图 2-14 在 Eclipse 中使用外部 Maven

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

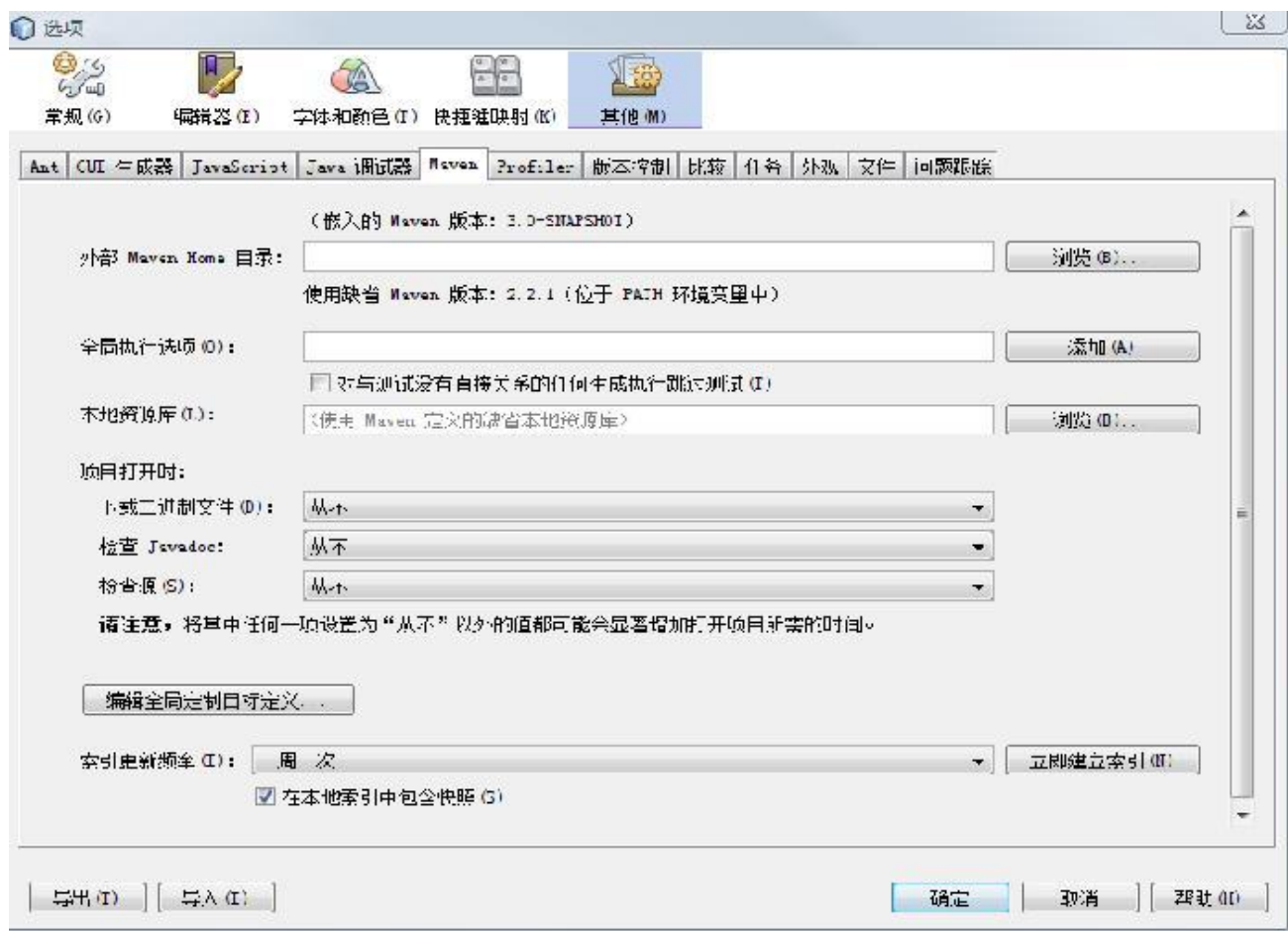


图 2-15 在 NetBeans 中使用外部 Maven

## 1.33 2.8 小结 [TOP](#)

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

## 1.34 第 3 章 Maven 使用入门 [TOP](#)

到目前为止，我们已经大概了解并安装好了 Maven，现在，我们开始创建一个最简单的 Hello World 项目。如果你是初次接触 Maven，我建议你按照本章的内容一步步地编写代码并执行，可能你会碰到一些概念暂时难以理解，不用着急，记下这些疑难点，相信本书的后续章节会帮你逐一解答。

### 3.1 编写 POM

### 3.2 编写主代码

### 3.3 编写测试代码

### 3.4 打包和运行

### 3.5 使用 Archetype 生成项目骨架

### 3.6 m2eclipse 简单使用

### 3.7 NetBeans Maven 插件简单使用


### 3.8 小结

## 1.35 3.1 编写 POM TOP

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

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

代码清单 3-1: Hello World 的 POM

Java 代码 

```
1. <SPAN style="FONT-SIZE: small"><?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0"
3.     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5. http://maven.apache.org/maven-v4_0_0.xsd">
6.   <modelVersion>4.0.0</modelVersion>
7.   <groupId>com.juvenxu.mvnbook</groupId>
8.   <artifactId>hello-world</artifactId>
9.   <version>1.0-SNAPSHOT</version>
10.  <name>Maven Hello World Project</name>
11. </project>
12. </SPAN>
```

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

根元素下的第一个子元素modelVersion指定了当前POM模型的版本,对于Maven2及Maven3来说,它只能是4.0.0。

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

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

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

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

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

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

## 1.36 3.2 编写主代码 TOP

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

代码清单 3-2: Hello World 的主代码

Java 代码 

1. `<SPAN style="FONT-SIZE: small">package com.juvenxu.mvnbook.helloworld;`
- 2.
3. `public class HelloWorld`

```


4.  {
5.      public String sayHello()
6.      {
7.          return "Hello Maven";
8.      }
9.
10. public static void main(String[] args)
11. {
12.     System.out.print( new HelloWorld().sayHello() );
13. }
14. }
15. </SPAN>

```

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

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

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

Java 代码 

```

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

```



17. [INFO] Final Memory: 9M/16M

18. [INFO] -----

19. </SPAN>

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

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

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

## 1.37 3.3 编写测试代码 TOP

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

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

代码清单 3-3：为 Hello World 的 POM 添加依赖

Java 代码 

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

```
12. <dependency>
13.   <groupId>junit</groupId>
14.   <artifactId>junit</artifactId>
15.   <version>4.7</version>
16.   <scope>test</scope>
17. </dependency>
18. </dependencies>
19. </project>
```

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

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

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

代码清单 3-4：Hello World 的测试代码

Java 代码 

```
1. package com.juvenxu.mvnbook.helloworld;
2. import static org.junit.Assert.assertEquals;
3. import org.junit.Test;
4.
5. public class HelloWorldTest
6. {
7.     @Test
8.     public void testSayHello()
9.     {
10.         HelloWorld helloWorld = new HelloWorld();
11.
12.         String result = helloWorld.sayHello();
13.
```

```

14.     assertEquals( "Hello Maven", result );
15. }
16. }

```

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

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

Java 代码 

```

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

```

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

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

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

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

Java 代码 

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

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

## Java 代码

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

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

## 1.38 3.4 打包和运行 TOP

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

## Java 代码

```
1. ...
2. Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
3.
4. [INFO] [jar:jar {execution: default-jar}]
5. [INFO] Building jar: D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
6. [INFO]
7. -----
8. [INFO] BUILD SUCCESSFUL
9. ...
```

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

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

Java 代码 

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

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

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

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

Java 代码 

```
1. <plugin>
```



```
2. <groupId>org.apache.maven.plugins</groupId>
3. <artifactId>maven-shade-plugin</artifactId>
4. <version>1.2.1</version>
5. <executions>
6.   <execution>
7.     <phase>package</phase>
8.     <goals>
9.       <goal>shade</goal>
10.    </goals>
11.  </configuration>
12.  <transformers>
13.    <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
14.      <mainClass>com.juvenxu.mvnbook.helloworld.HelloWorld</mainClass>
15.    </transformer>
16.  </transformers>
17. </configuration>
18. </execution>
19. </executions>
20. </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。

## 1.39 3.5 使用 Archetype 生成项目骨架 TOP

Hello World 项目中有一些 Maven 的约定：在项目的根目录中放置 pom.xml，在 src/main/java 目录中放置项目

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

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

如果是 Maven 3，简单的运行：

```
mvn archetype:generate
```

如果是 Maven 2，最好运行如下命令：

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

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

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

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：

Java 代码 

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

Archetype 插件将根据我们提供的信息创建项目骨架。在当前目录下，Archetype 插件会创建一个名为

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

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

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

## 1.40 3.6 m2eclipse 简单使用 [TOP](#)

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

### 1.41 3.6.1 导入 Maven 项目

第 2 章介绍了如何安装 `m2eclipse`，现在，我们使用 `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 3-1 所示：

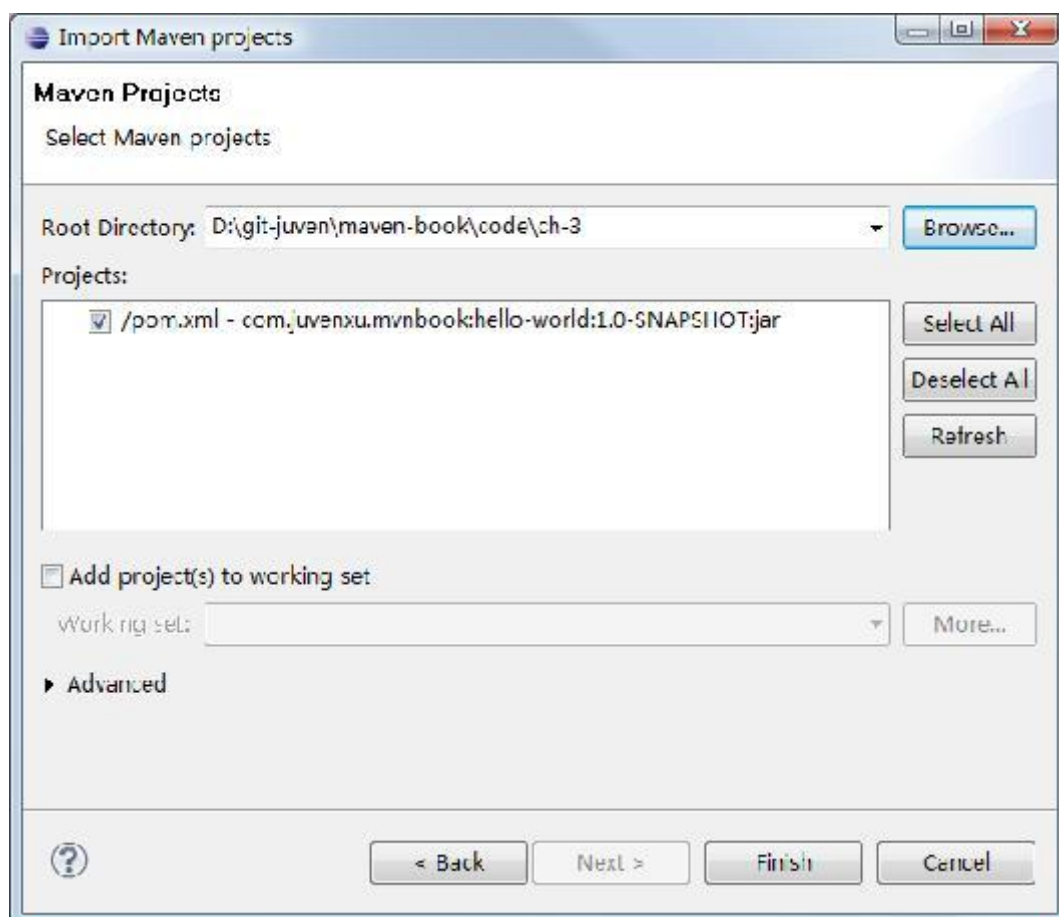


图 3-1 在 Eclipse 中导入 Maven 项目

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

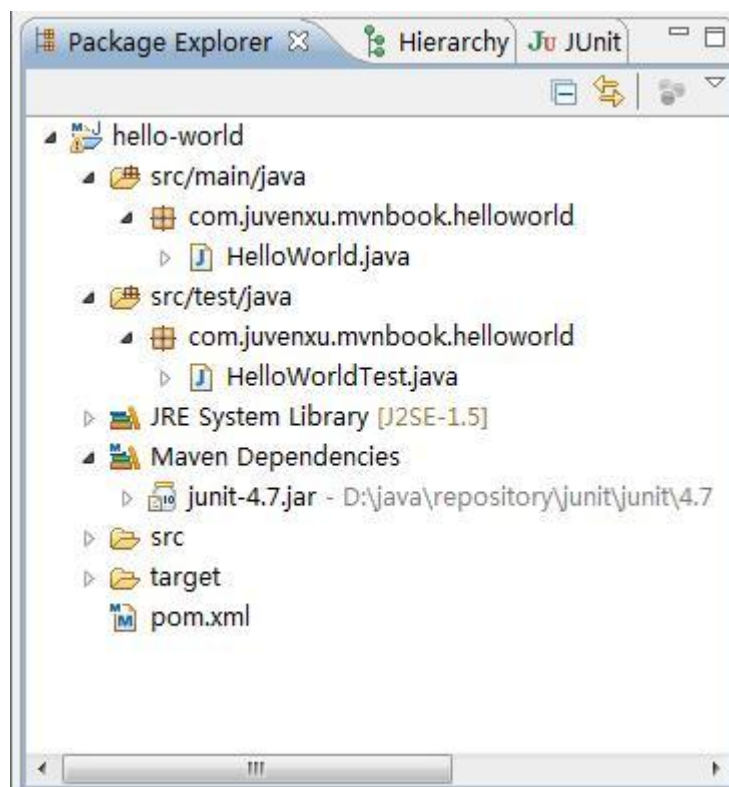


图 3-2 Eclipse 中导入的 Maven 项目结构

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

## 1.42 3.6.2 创建 Maven 项目

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

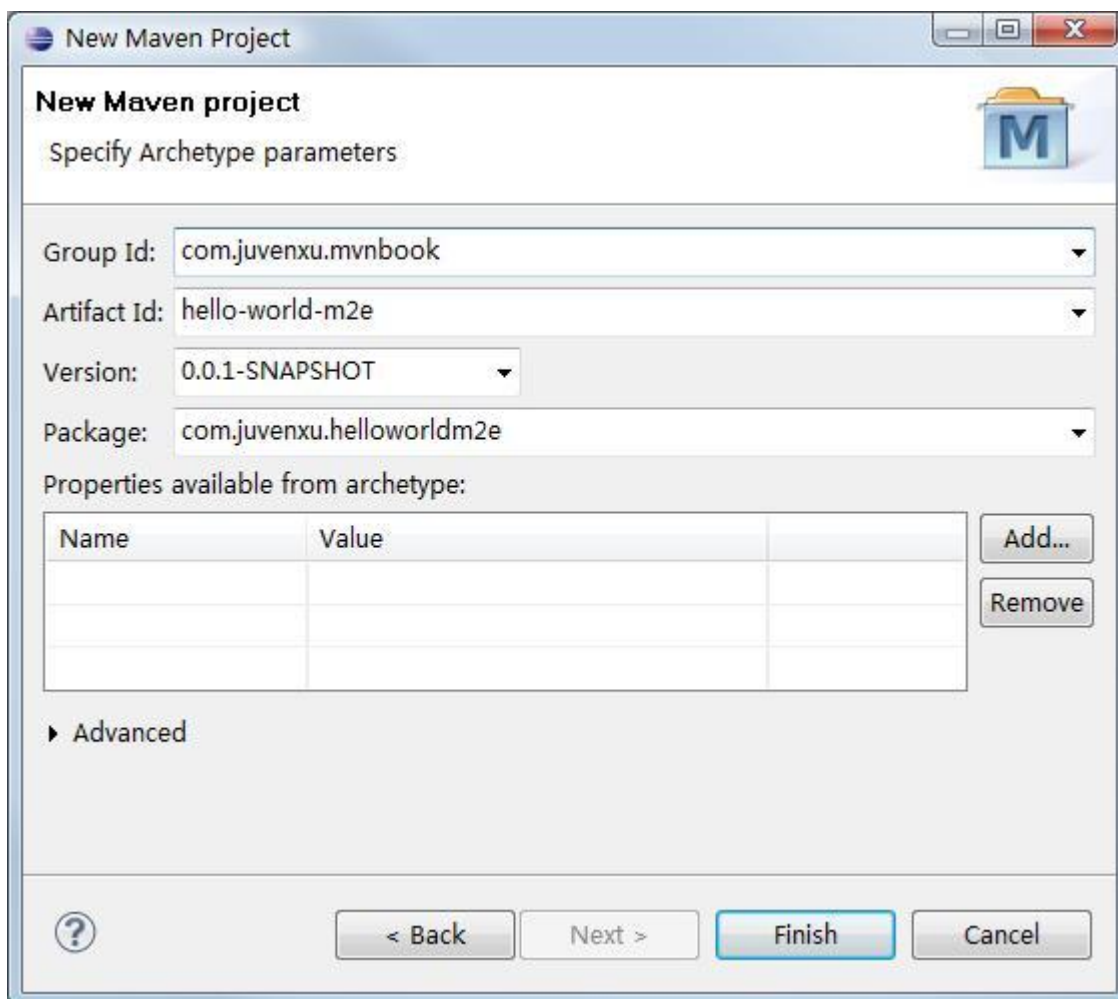


图 3-3 在 Eclipse 中使用 Archetype 创建项目

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

### 1.43 3.6.3 运行 mvn 命令

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



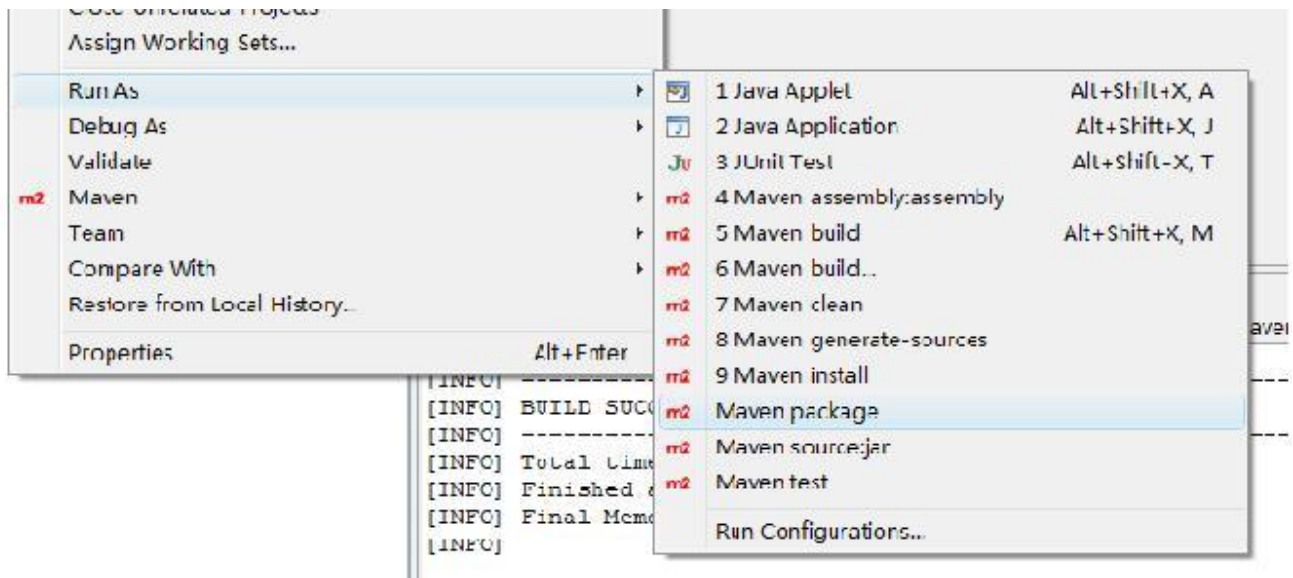


图 3-4 在 Eclipse 中运行默认 mvn 命令

选择想要执行的 Maven 命令就能执行相应的构建，同时我们也能在 Eclipse 的 console 中看到构建输出。这里常见的一个问题是，默认选项中没有我们想要执行的 Maven 命令怎么办？比如，默认带有 `mvn test`，但我们想执行 `mvn clean test`，很简单，选择 **Maven 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 3-5 就是自定义 Maven 运行命令的界面：

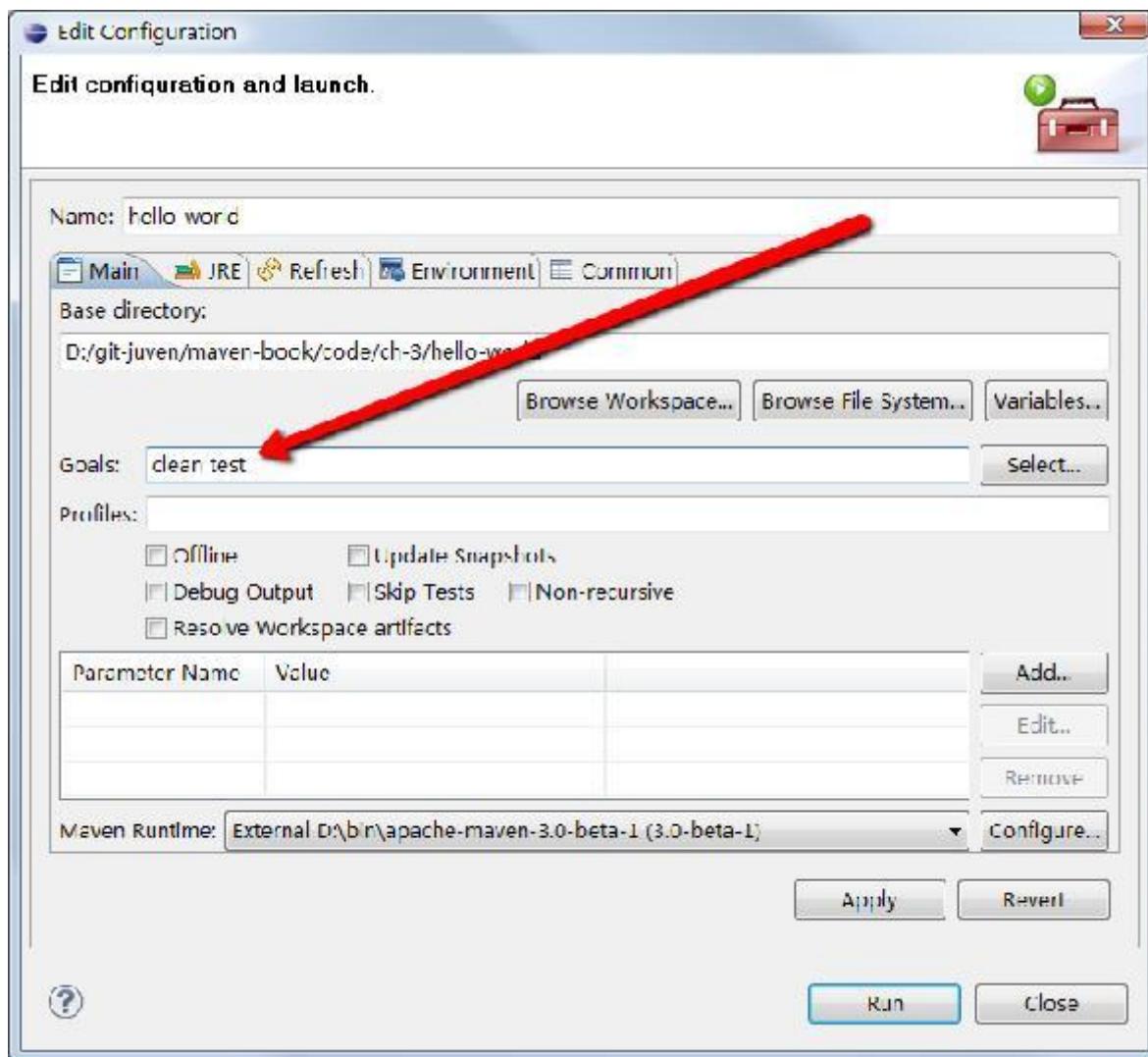


图 3-5 在 Eclipse 中自定义 mvn 命令

## 1.44 3.7 NetBeans Maven 插件简单使用 [TOP](#)

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

### 1.45 3.7.1 打开 Maven 项目

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

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

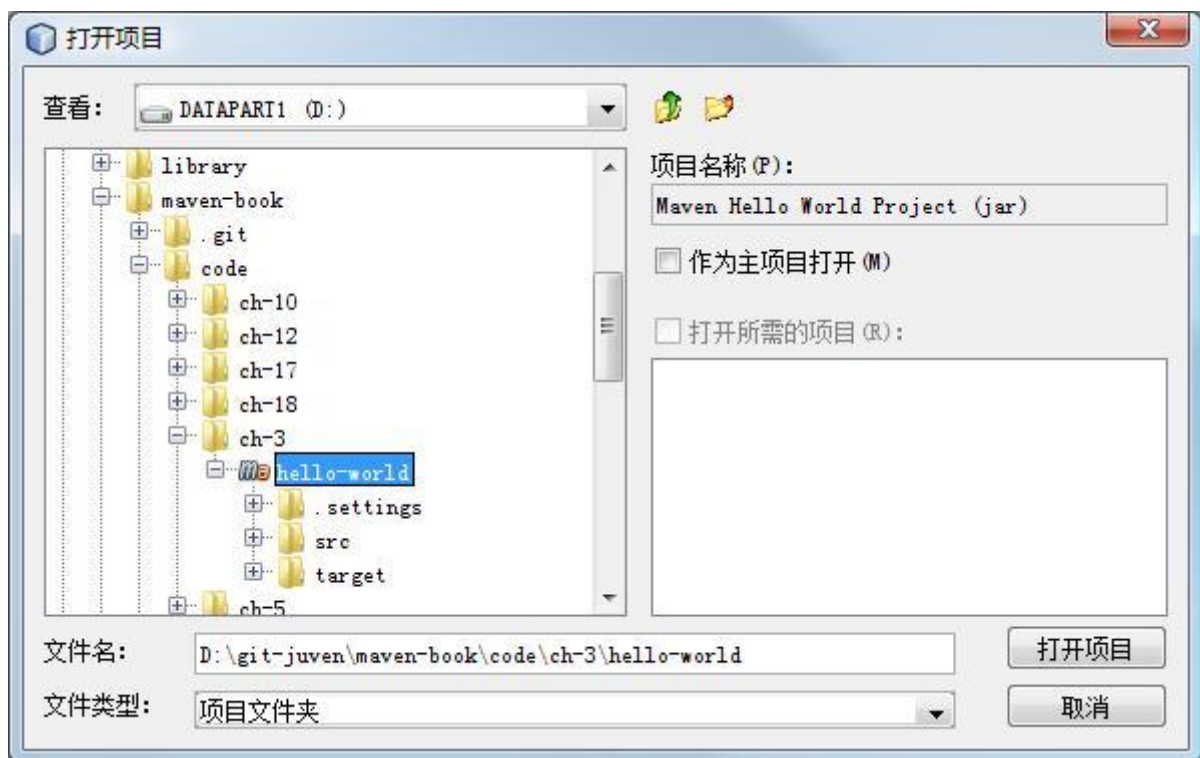


图 3-6 在 NetBeans 中导入 Maven 项目

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

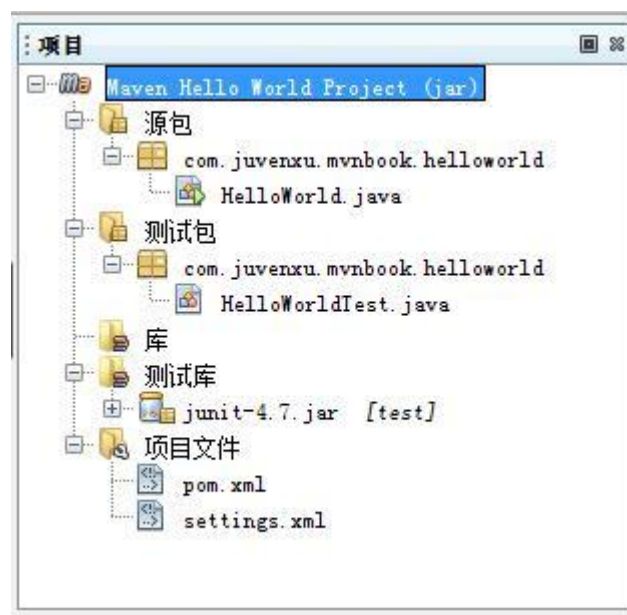


图 3-7 NetBeans 中导入的 Maven 项目结构

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

## 1.46 3.7.2 创建 Maven 项目

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

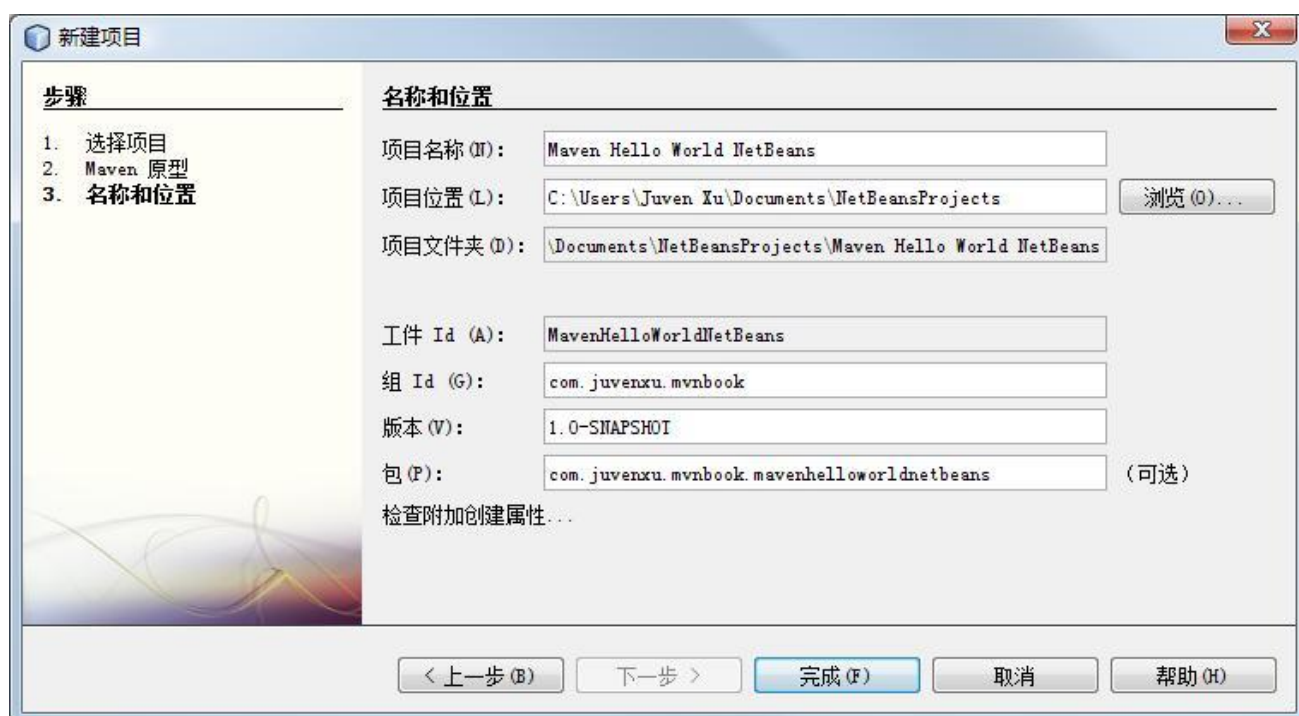


图 3-8 在 NetBeans 中使用 Archetype 创建 Maven 项目

点击完成之后，一个新的 Maven 项目就创建好了。

## 1.47 3.7.3 运行 mvn 命令

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

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

命令配置。

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

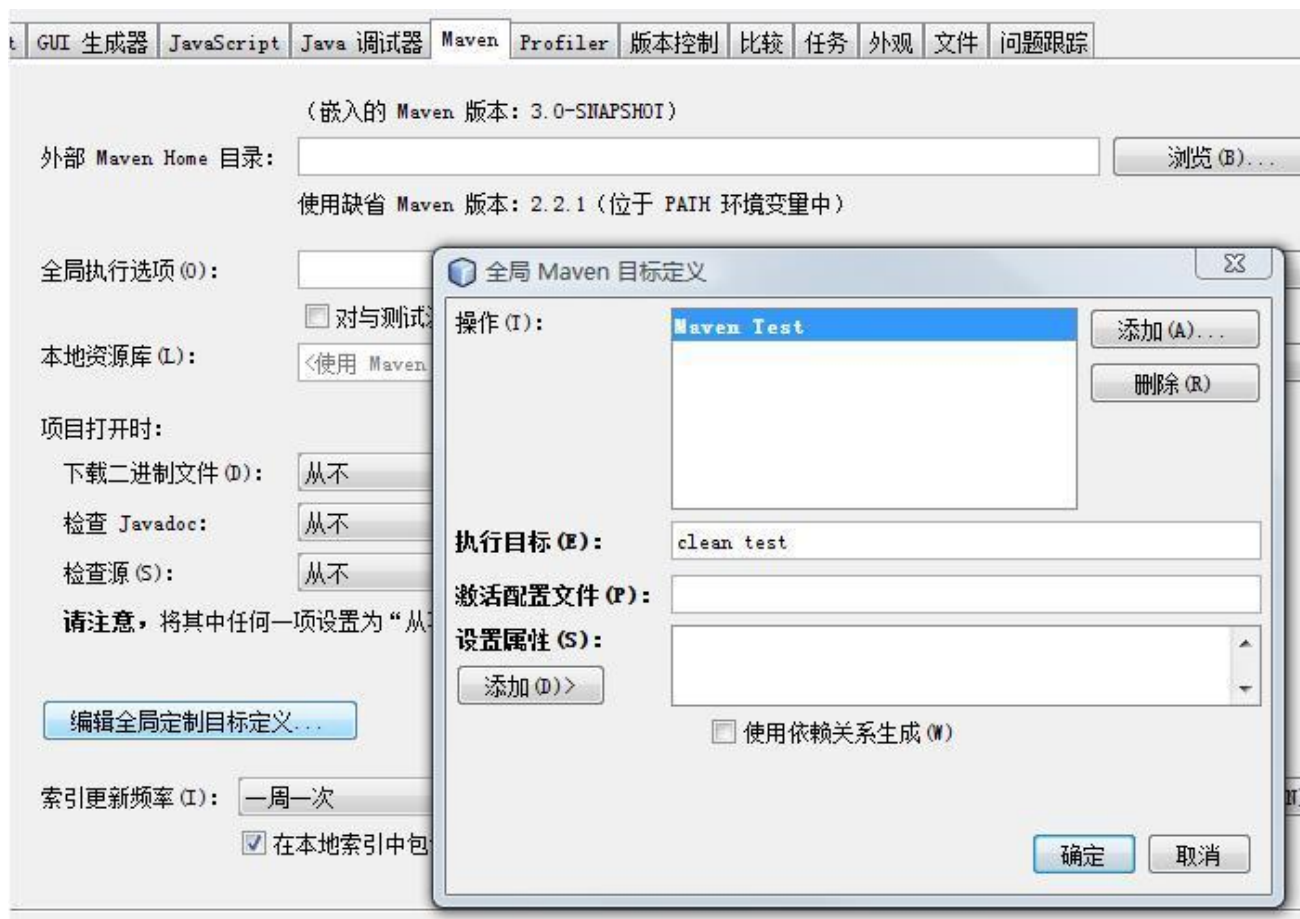


图 3-9 在 NetBeans 中自定义 mvn 命令

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

## 1.48 3.8 小结 TOP

本章以尽可能简单且详细的方式叙述了一个 Hello World 项目，重点解释了 POM 的基本

内容、Maven 项目的基本结构、以及构建项目基本的 Maven 命令。在此基础上，还介绍了如何使用 Archetype 快速创建项目骨架。最后讲述的是如何在 Eclipse 和 NetBeans 中导入、创建及构建 Maven 项目。