Maven 简介

1.1 何为 Maven

Maven 这个词可以翻译为“知识的积累”，或者翻译成“专家”或“内行”。本书我们介绍 Maven 这一跨平台的项目管理工具。作为 Apache 组织中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，项目信息管理。无论是小型的开源类库项目，或者是大型的企业级应用，无论是传统的瀑布式开发，或者是流行的敏捷模式，Maven 都十分适用。

1.1.1 何为构建（Build）

不管你有没有意识到，构建是每个程序员每天都在做的工作。早晨跑到公司，吃完手里的早饭吃后，我们从源码库签出最新的源码，然后跑一下单元测试，发现有一些失败的测试，于是找相关的同事一起调试一下，修复了错误代码。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，心满意足的吃饭休息。下午先昏昏沉沉得开了个例会，完了喝杯咖啡继续工作。刚才会上经理要求看测试报告，OK，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，发了封电子邮件给经理，松了口气。同时看到 QA 那边又发过来了几个 bug，没办法，先本地重现再说，于是熟练的点击 IDE 一些按钮生成了一个 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，通知下 QA，愉快的结束了一天的工作。
如果仔细总结下，我们会发现除了编写源代码，每天工作的相当一部分时间花在了编译，运行单元测试，生成文档，打包，部署等等繁琐且不起眼的工作上，这就是构建。我们往往觉察不到这部分时间的流逝，其实，只要稍微花点心思，这一系列的工作完全是可以自动化的。软件的构建可以像全自动流水线一样，只要一条简单的命令，所有繁琐的步骤都能够自动完成，我们很快就能得到最终的结果。

1.1.2 Maven 是优秀的构建工具

前面介绍了 Maven 的用途之一是服务于构建，它是一个异常强大的构建工具，它帮我们自动化构建过程，从清理，编译，测试，到生成报告，到打包，部署。我们不需要，也不应该一遍又一遍的输入命令，一次又一次的点击鼠标。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 会帮我们处理那些繁琐的任务。
Maven 是跨平台的，这意味着无论你是在 windows 上，还是在 Linux 或者 Mac 上，你都可以使用同样的命令，构建同样的项目，得到同样的结果。
我们总是不停在寻找方法避免重复，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 最大化的消除了构建的重复，它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，我们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，我们更不需要告诉 Maven 如何运行测试，我们只需要遵循 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它构建脚本，构建工具的优点，总结了大量项目的实际需求。遵循这个模型，我们可以避免很多前人的错误，我们可以直接使用大量成熟的 Maven 插件来完成我们的任务（很多时候你可能都不知道你在使用 Maven 插件）。此外，如果你真得有非常特殊的需求，你也可以轻松实现自己的插件。
Maven 还有一个优点是它能帮助我们标准化构建过程，在 Maven 之前，可能十个项目有十种构建的方式，现在，只要是使用了 Maven，所有项目的构建命令都是简单一致的，这极大的减免了不必要的学习成本，并能帮助促进项目团队的标准化。
综上我们可以看到，Maven 作为一个构建工具，它不仅仅能帮我们自动化构建，它还能够抽象构建过程，提供构建任务实现，它跨平台，并且对外提供了一致的操作接口，这一切足以使它成为优秀的，流行的构建工具。

1.1.3 Maven 不仅仅是构建工具

很多人仅仅认识到 Java 是一门编程语言，但 Java 不只如此，它还是一个平台，通过 JRuby 和 Jython，程序员们可以在 Java 平台上编写和运行 Ruby 和 Python 程序。
这里我也想提醒读者，Maven 不仅仅是构建工具，它还是一个依赖管理工具，还是一个项目信息管理工具。它提供了中央仓库，并能帮你自动下载构件。
在这个开源的年代里，几乎任何一个 Java 应用都会借用一大堆第三方的开源类库，这些类库都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，依赖臃肿等等问题都会显露出来。手工解决这些问题是一件枯燥乏味，确有不得不面对的事情。Maven 提供了一个优秀的解决方案，它通过一个坐标系统准确定位每一个构件（Artifact），也就是用，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经纬，带来的秩序，于是我们就能借助它帮助我们有序的管理依赖，轻松的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，等等。这些微小的变化看起来很琐碎，并不起眼，但往往在不知不觉中节省了我们大量的寻找信息的时间。这些还只是直接的项目信息，通过 Maven 自动生成的站点和一些已有的插件，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，等等非常具有价值的项目信息。
Maven 还给全世界的 Java 开发者提供了一个免费的中央仓库，在这里，我们可以找到几乎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得搜索。只要定位了坐标，Maven 还能够帮我们自动下载。不再需要一个个的点击开源项目首页，不再需要寻找下载链接，不再需要手工的去下载。

1.2 为什么需要 Maven

Maven 不是 Java 领域唯一的构建管理的解决方案。我们将通过一些简单的例子解释 Maven 的必要性，并介绍其它构建解决方案，如 IDE, Make 和 Ant，并将它们与 Maven 作一些比较。

1.2.1 组装 PC 和品牌 PC

我初中时候开始接触计算机，到了高中更是疯狂的想要一台自己的计算机，终于我得到了我的第一台计算机，它是赛扬 733 的。这是一个漫长的过程，我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，甚至是声卡，我都仔细挑选，后来还跑了很多商家，调货，讨价还价，组装好，接着自己装操作系统，驱动程序……我很享受这个过程，虽然这花费了我大量时间，而且事实证明最后装出来的机器稳定性也不怎样。

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，我也变懒了。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选择，点击选择了我需要的配置，然后下单，付款。接着 PC 供应商会帮我组装电脑，安装操作系统和驱动程序，一周后，物流公司将我的电脑送到了家里，我简单地连接了显示器电源，就能直接使用了。这节省了我大量的时间，而且这台电脑十分稳定，我知道商家在把电脑发送给我之前已经进行了很好的测试。哦，对了，我还能享受两年的售后服务，虽然这是收费的。

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，但恐怕实际项目中没法给你那么多时间。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，你还能得到来自于 Maven 社区的大量支持。唯一于购买品牌 PC 不同的是，Maven 是开源的，你无须为此付费，如果需要，你还是能去了解 Maven 是如何工作的。而我可不知道那些 PC 巨头的商业秘密。

1.2.2 IDE 不是万能的

当然，我们无法否认优秀的 IDE 帮助我们大大提高了开发效率。当前主流的 IDE 如 Eclipse 和 NetBeans 都提供了强大的文本编辑，调式，甚至重构功能。虽然使用简单的文本编辑器加上命令行也能完成几乎任何开发工作，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，或者代码生成，等等这些工作都是相互独立的，很难一键完成所有工作。而手工劳动往往就意味着低效，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，而不是过多的依赖它。对于构建这样的任务，在 IDE 中一次次点击鼠标是愚蠢的行为。Maven 是这方面的专家，而且主流 IDE 如 Eclipse 和 NetBeans 都集成了 Maven，这样，我们就可以在 IDE 中方便地运行 Maven 执行构建。

1.2.3 Make

Make 也许是最早的构建工具，它由 Stuart Feldman 于 1977 年在 Bell 实验室创建。他也因此于 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，以及 BSD Make，还有 Windows 平台的 Microsoft nmake 等等。

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），而每一条规则的构成部分又包括：目标（Target），依赖（Prerequisite），和命令（Command）。Makefile 基本的结构如下：

```
TARGET… : PREREQUISITE…
COMMAND
….
…
```

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一切系统的本地命令，尤其是对于 Unix/Linux 系统来说，丰富的功能强大的命令能够帮助 Make 快速高效的完成任务。

但是，Make 将自己和操作系统绑定在一起了，也就是说，使用 Make，你就不能实现（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，很多人抱怨他们 Make 构建失败的原因往往是一个难以发现的空格或 Tab 使用错误。

1.2.4 Ant

Ant 不是指蚂蚁(英文中 Ant 是蚂蚁的意思),它意指“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 创作该工具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，也正是使用了 Java，Ant 是跨平台的；此外，Ant 使用 XML 定义构建脚本，这相对于 Makefile 来说也更加的友好。

与 Make 类似，Ant 有一个构建脚本 build.xml，如下：

```
<?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 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，开发者显式的指定每一个目标，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。而 Maven 是声明式的，项目构建过程以及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，那是一个令人头疼的问题，好在现在 Ant 的用户可以借助 Ivy 管理依赖。而对于 Maven 用户来说，依赖管理是理所当然的，Maven 不仅仅内置了依赖管理，更拥有一个可能拥有全世界最多 Java 开源软件包的中央仓库，Maven 用户无须进行任何配置就可以直接享用。

1.2.5 现状，困惑

（注：该小节内容整理自网友 Arthas 最早在 Maven 中文 MSN 群中的讨论，在此致谢）

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。大家一致协商确定使用 Spring，iBatis 和 Tapestry。Jar 包哪里找呢？公司里估计没有人能把 Spring，iBatis 和 Tapstry 所使用的 jar 包一个不拉的找出来。大家的做法是，先到 Spring 站点去找个 spring-with-dependencies，然后去 iBatis 把所有列出来的 jar 也拉下来，同理还有 Tapstry，Apache commons…。项目还没有开发呢，WEB-INF/lib 下已经有近百个 jar 了，带版本号的，不带版本号的，有用的，没用的，冲突的，怎一个乱字了得！
随着项目的开发，小张不时地发现版本错误的问题，也遇到过版本冲突，他只能硬着头皮一个个解决。项目开发到一半，经理发现最终部署的应用体积实在太大了，要求小张去掉一些没用的 jar 包，于是小张只能加班加点的一个个删选……
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，于是他也学了，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，最后再 jar，zip，war，然后打包就成功了。
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; … 他没有发现，他写的这么多的 Ant 脚本中，有多少是重复劳动，有多少内容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，小张意识到了问题，并想出来复用 Ant 脚本，于是开会时他说：“以后就都用我这个规范的 Ant 脚本吧，只要新的项目遵循我定义的目录结构就可以了”经理一听觉得很有道理，“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，全世界人都在用，你自己都不用写什么东西。
为什么没人说：“我自己写的代码最灵活，所以我不用 Spring，我自己实现 IoC，我不用 Hibernate，我自己封装 JDBC”？

1.3 Maven 与极限编程

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，以及推荐实践。让我们看一下 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，复杂的构建系统中摆脱出来。使用 Maven 现有的成熟，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 更能无缝的支持或者融入到一些主要的 XP 实践中去：

- *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 有现成的成熟的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的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，之后一条简单的命令就能让 Maven 帮你清理，编译，测试，打包，部署，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 都能很好的和 Maven 进行集成。这也就是说，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的了解到项目的最新状态。当然，Maven 并不会帮你把测试覆盖率报告贴到墙上，也不会在你的工作台上放个鸭子告诉你构建失败了。不过使用 Maven 发布的项目报告站点，并配置你需要的项目报告如测试覆盖率报告，这都能帮你发信息推送到开发者眼前。

上述的这些实践并非只在 XP 中适用，事实上，除了其他敏捷开发方法如 SCRUM 之外，几乎其它任何软件开发方法都能借鉴这些实践，这也就是说，Maven 能够很好的支持几乎任何软件开发方法。

比如说在传统的瀑布模型开发中，项目需要顺序经历需求开发，分析，设计，编码，测试，及集成发布阶段。从设计和编码阶段，我们就可以使用 Maven 来建立项目的构建系统，在设计阶段，我们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。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，尽早的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 也能帮助我们快速的发布项目。

1.4 被误解的 Maven

C++之父 Bjarne Stroustrup 说说一句话“有两种计算机语言：一些语言天天被人骂，还有一些没人用”。当然这话也不全对，大红大紫的 Ruby 不仅有人用，而且骂得人也少。不过用户最多的 Java 得到的骂声就不绝于耳了。Maven 的用户可不少，它的邮件列表目前在 Apache 项目中排在第四位：http://www.nabble.com/Apache-f90.html。

让我们看看 Maven 受到了哪些质疑，而这里我也对这些质疑逐一解释：

“Maven 对于 IDE 的支持如 Eclipse 和 IDEA 较差，bug 多且不稳定。”

Maven 相对于 JUnit，Ant 来说，显得比较年轻，因此 IDE 集成等衍生产品还不够全面和成熟。但是，我们一定要知道，使用 Maven 最高效的方式永远是命令行，IDE 对于自动化构建有着天生的缺陷。此外，我们也看到 eclipse 的 maven 插件——m2eclipse 是一个比较优秀和成熟的工具，此外，Netbeans 也在积极的为集成 Maven 而努力。

“Maven 采用了一个糟糕的插件系统来执行构建，新的，破损的插件会让你的构建莫名其妙的失败”

自 Maven 2.0.9 开始，所有核心的插件都被设定了稳定版本，这意味着日常使用 Maven 几乎

不会受到不稳定插件的影响。此外，Maven 社区也提倡为任何你使用的插件设定稳定的版本，如果用户有着好的实践不去采纳，遇到了问题就抱怨，这未免不够公允。

“缺乏文档是理解和使用 Maven 的一个主要障碍”

嗯，这是事实。Maven 官方站点的文档十分凌乱，各种插件的文档更是需要费力寻找。Sonatype 编写的《Maven 权威指南》很好的改善了这一状况。本书编写的目的也是帮助大家理解和使用 Maven。

“Maven 过于复杂，它就是构建系统的 EJB2”

不要指望 Maven 十分简单，这几乎是不可能的，Maven 是用来管理项目的，想想看，清理，编译，测试，打包，发布，以及一些自定义的过程。这本身就是件复杂的事情。目前 Java 社区还有比 Maven 更强大，更简单的构建工具么？答案是否定的。我们可以尝试去帮助 Maven 变得更简单，而绝不是抛弃它，然后自己实现一套更加复杂的构建系统。

“Maven 的仓库十分混乱，当无法从仓库中得到需要的类库时，我需要手工下载复制到本地仓库中”

Maven 的中央仓库确实不完美，你也许会发现某个 jar 包出现在两个不同的路径下，这不是 Maven 的错，这是开源项目本身改变自身的坐标。再者，如果没有中央仓库，你将不得不去开源项目首页寻找下载链接，这不是更费事么？要知道现在有很多的 Maven 仓库搜索服务。无法从中央仓库找到你需要的类库？由于许可证等因素，这是完全有可能的，这时你需要做的是建立一个组织内部的仓库服务器，你会发现这会给你带来许多意想不到的好处。

1.4 小结

本章只是从概念上简单的介绍了一下 Maven，通过本章，你应该大致了解 Maven 是什么，它有什么用途。这里我还将 Maven 与其它流行的构建工具如 Make 和 Ant 做了一些比较和分析，如果你没用过 Maven，但有 Make 或者 Ant 的使用经验，相信这能帮助你更好的了解和对比所有这些工具。

将 Maven 和极限编程结合起来分析是为了让大家从另一个角度来了解 Maven，毕竟软件的开发离不开对于软件过程的理解。

本章最后还收集一些用户对 Maven 的误解，并逐条分析解释，希望能够让大家消除误解，积极的接受 Maven。

第二章 Maven 安装

前面一章介绍了 Maven 是什么，以及为什么要使用 Maven。从这一章开始，我们正式实际接触 Maven。本章我们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，我们还会介绍如何在主流的 IDE 中集成 Maven。

2.1 在 Windows 上安装 Maven

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。不过本书的所有样例都基于 JDK 5 及以上版本。打开 windows 的命令行（Windows 键+R，然后输入 cmd，回车），运行如下的命令来检查你的 Java 安装：

```
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)
```

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

2.1.2 下载 Maven

打开浏览器，访问 Maven 的下载页面：http://maven.apache.org/download.html。你会看到各种版本 Maven 的各种格式的下载链接。在编写本书的时候，Maven 最新的版本是 2.2.1，因此下载 apache-maven-2.2.1-bin.zip。当然，如果你对 Maven 的源代码感兴趣并想自己构建 Maven，你还可以下载 apache-maven-2.2.1-src.zip。该下载页面还提供了 md5 校验和文件及 asc 签名文件，可以用来验证 Maven 分发包的正确性和安全性。

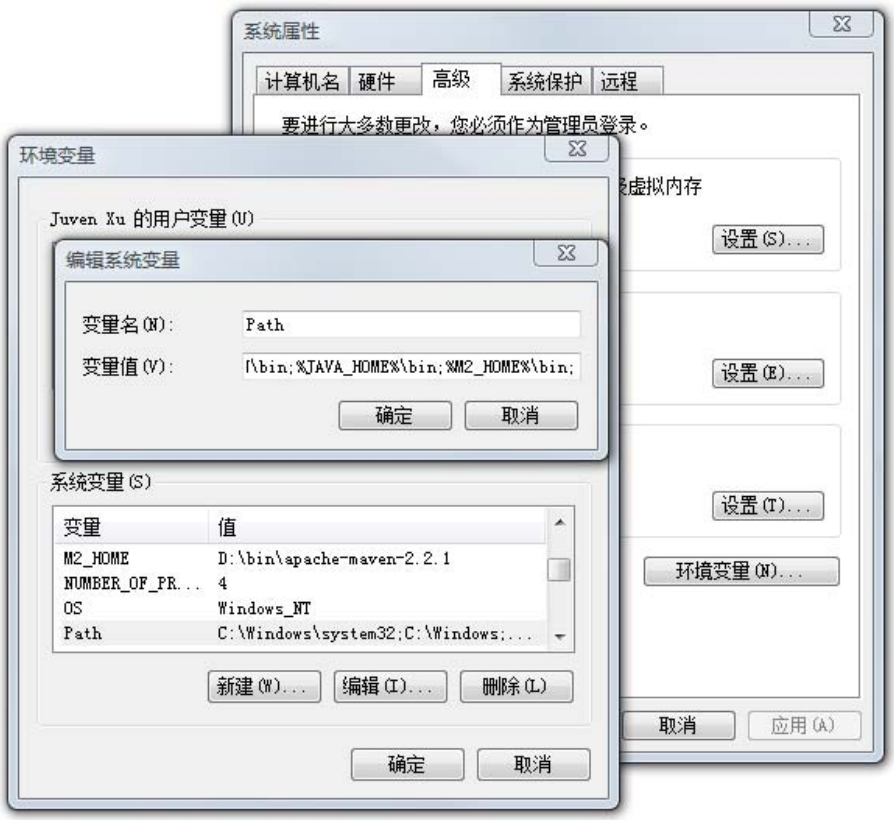
2.1.3 本地安装

使用你喜爱的解压工具将安装文件释放到你指定的目录中，如：

```
D:\bin>jar xvf "C:\Users\Juven Xu\Downloads\apache-maven-2.2.1-bin.zip"
```

这里我们的 Maven 安装目录是 D:\bin\apache-maven-2.2.1，接着我们需要设置环境变量，将 Maven 安装配置到操作系统环境中。

使用 *Windows 键+Pause 键*调出系统信息面板，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*，变量值为我们的 Maven 安装目录，这里是 *D:\bin\apache-maven-2.2.1*。点击确定，接着我们在系统变量中找到一个名为 Path 的变量，在其变量值的末尾加上*%M2_HOME%\bin;*，注意多个值之间需要有分号隔开。点击确定，至此，环境变量设置完成，详细情况如下图所示：



这里需要提一下的是 Path 环境变量，当我们在 cmd 中输入命令的时候，Windows 首先会在当前目录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 安装目录，因此，Windows 会在执行命令的时候搜索目录 *D:\bin\apache-maven-2.2.1\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 安装：

```
C:\Users\Juven Xu>echo %M2_HOME%
D:\bin\apache-maven-2.2.1

C:\Users\Juven Xu>mvn -version
Apache Maven 2.2.1 (r801777; 2009-08-07 03:16:01+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"
```

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** 执行了我们第一条 Maven 命令，以检查 Windows 是否能够找到正确的 mvn 执行脚本。

2.1.4 升级 Maven

Maven 还是比较年轻的，更新也比较频繁，因此我们也会有需要更新 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。比如，假设 Maven 推出了新版本 2.3.0，我们将其下载然后解压至目录 *D:\bin\apache-maven-2.3.0*，接着，遵照前一节描述的步骤编辑环境变量 M2_HOME，更新其值为 D:\bin\apache-maven-2.3.0。至此更新就完成了。当然，同理如果你发现你需要使用某一个旧版本的 Maven，也只需要编辑 M2_HOME 环境变量指向旧版本的安装目录。

2.2 在基于 Unix 的系统上安装 Maven

由于 Maven 是跨平台的，它可以在任何一种主流的操作系统上运行，本节介绍如何在基于 Unix 的系统上安装 Maven。这些系统包括 Linux、Mac OS 以及 FreeBSD 等等。

2.2.1 下载和安装

首先，和在 Windows 上安装 Maven 一样，你需要检查 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 $JAVA_HOME
/usr/local/jdk1.6.0_11
juven@juven-ubuntu:~$ 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)
```

接着到 http://maven.apache.org/download.html 下载 Maven 安装文件，如 apache-maven-2.2.1-bin.tar.gz，解压到本地目录：

```
juven@juven-ubuntu:bin$ gunzip apache-maven-2.2.1-bin.tar.gz
juven@juven-ubuntu:bin$ tar -xf apache-maven-2.2.1-bin.tar
```

现在，我们已经创建好了一个 Maven 安装目录 apache-maven-2.2.1，虽然直接使用该目录配置环境变量之后我们就能使用 Maven 了，但这里我更推荐与这个安装目录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-2.2.1 apache-maven
juven@juven-ubuntu:bin$ ls -l
total 4
lrwxrwxrwx 1 juven juven   18 2009-09-20 15:43 apache-maven -> apache-maven-2.2.1
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-2.2.1
```

接下来，我们需要设置 M2_HOME 环境变量指向符号链接 apache-maven-2.2.1，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，我们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 $M2_HOME
/home/juven/bin/apache-maven
juven@juven-ubuntu:bin$ mvn -version
Apache Maven 2.2.1 (r801777; 2009-08-07 03:16:01+0800)
Java version: 1.6.0_11
Java home: /usr/local/jdk1.6.0_11/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux" version: "2.6.27-7-generic" arch: "i386" Family: "unix"
```

2.2.2 升级 Maven

在基于 Unix 的系统上，我们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

在配置环境变量的时候引用该符号链接，这样做是为了方便升级。现在，假 设我们需要升级到新的 Maven 2.3 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 2.3 目录便可：

```
juven@juven-ubuntu:bin$ rm apache-maven
juven@juven-ubuntu:bin$ ln -s apache-maven-2.3/ apache-maven
juven@juven-ubuntu:bin$ ls -l
total 8
lrwxrwxrwx 1 juven juven   17 2009-09-20 16:13 apache-maven -> apache-maven-2.3/
drwxr-xr-x 6 juven juven 4096 2009-09-20 15:39 apache-maven-2.2.1
drwxr-xr-x 2 juven juven 4096 2009-09-20 16:09 apache-maven-2.3
```

同样的道理，我们可以很方便地切换 Maven 到任意的一个版本。现在，升级完成了，我们可以运行 **mvn -version** 进行检查。

2.3 安装目录分析

本章前面的部分讲述了如何在各种操作系统中安装及升级 Maven。现在我们来仔细看一下 Maven 的安装文件。

2.3.1 M2_HOME

前面我们讲到设置 M2_HOME 环境变量指向 Maven 的安装目录，本书之后所有使用 M2_HOME 的地方都指代了该安装目录，让我们看一下该目录的结构和内容：

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

- Bin：该目录包含了 mvn 运行的脚本，这些脚本用来配置 java 命令，准备好 classpath，还有相关的一些 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。我们在命令行输入任何一条 mvn 命令的时候，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 的时候开启 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，我们稍微会介绍 classworlds。

- Boot：该目录只包含一个文件，以 maven 2.2.1 为例，该文件为 *classworlds-1.1.jar*。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 Maven 用户来说，我们不必关心该文件。
- 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，我们就能在机器上全局的定制 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，逐步分析里面的各个元素。
- Lib：该目录包含一个如 *maven-2.2.1-uber.jar* 的文件。该文件包含了所有运行 Maven 需要的类文件，原本各为一个 jar 的第三方依赖如 commons-cli 都被拆解后重新合并到了这个 jar 文件中。也可以说，这个 jar 文件就是真正的 Maven。关于该文件还值得一提的是它包含了 Maven 超级 POM，其路径为 */org/apache/maven/project/pom-4.0.0.xml*，所有的 Maven 项目都继承自这个文件，本书后面将会对其详细解释。
- 其它： *LICENSE.txt* 记录了 Maven 使用的软件许可证 Apache License Version 2.0；*NOTICE.txt* 记录了 Maven 包含的第三方软件；而 *README.txt* 则包含了 Maven 的简要介绍，包括安装需求及如何安装的简要指令等等。

2.3.2 ~/.m2

在讲述该小节之前，我们先运行一条简单的命令：**mvn help:system**。该命令会打印出所有的 Java 系统属性（System Properties）和环境变量（Environment Variables）。这些信息对于我们日常的编程工作很有帮助。这里我暂时不解释 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 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，比如我目前的用户目录是 *C:\Users\Juven Xu*，你可以在 Vista 和 Windows7 中找到类似的用户目录，而之前版本的 Windows，该目录应该是类似于 C:\Document and Settings\Juven Xu\。在基于 Unix 的系统上，直接输入 **cd ~**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，这一点本书后面将会详细阐述。由于 Maven 仓库是通过简单文件系统透明地展示给 Maven 用户的，有些时候我们可以绕过 Maven 直接查看或修改仓库文件，在遇到疑难问题的时候，这往往十分有用。

默认情况下，*~/.m2* 目录下除了 repository 仓库之外就没有其它目录和文件了，不过大多数 Maven 用户会需要复制 *M2_HOME/conf/settings.xml* 文件到*~/.m2/settings.xml*。这是一条最佳实践，我们将在本章最后一小节详细解释。

2.4 设置 HTTP 代理

有时候你所在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。这个情况下，我们就需要为 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，就先需要获取正确的代理服务信息，如果 telnet 连接正确，没有出错信息，则输入 ctrl+]，然后 q，回车，退出即可。

在检查完毕之后，我们编辑~/.m2/settings.xml 文件（如果你没有该文件，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>
```

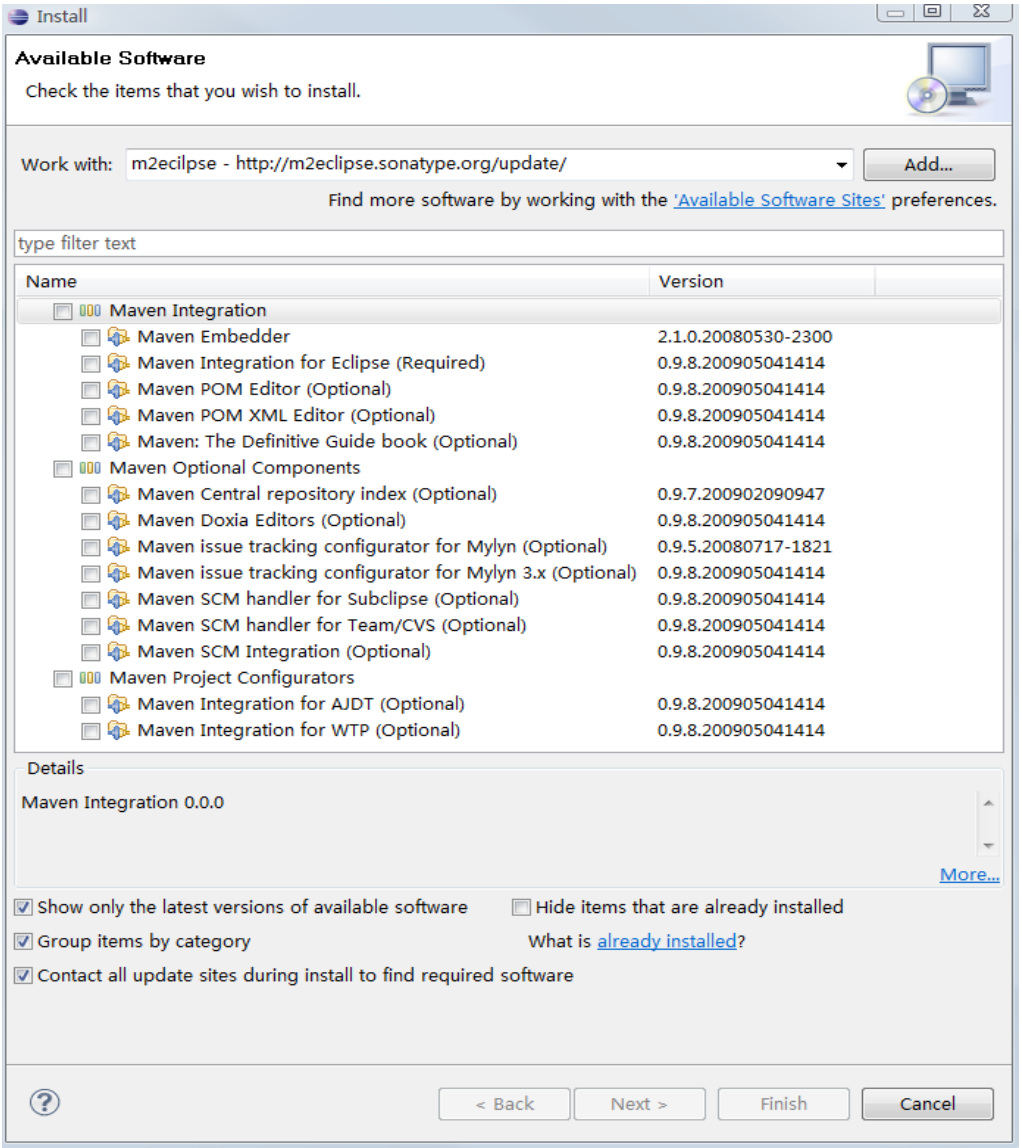
这段配置十分简单，proxies 下可以有多个 proxy 元素，如果你声明了多个 proxy 元素，则默认情况下第一个被激活的 proxy 会生效。这里我们声明了一个 id 为 my-proxy，被激活的，代理协议为 http 的代理，当然，最重要的是指定正确的主机名（host 元素）和端口（port 元素）。上述配置中我使用了 XML 注释注释掉了 username、password、nonProxyHost 几个元素，当你的代理服务需要认证的时候，你就需要配置 username 和 password；而 nonProxyHost 元素用来指定哪些主机名不需要代理，我们可以使用 | 符号来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

2.5 安装 m2eclipse

Eclipse 是一款非常优秀的，在 Java 社区广受赞誉的 IDE。除了基本的语法标亮，代码补齐，XML 编辑等基本功能，最新的 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，然后幸运的是，由 Maven 之父 Jason Van Zyl 创立的 Sonatype 公司建立了 m2eclipse 项目，这是一款 Eclipse 下十分强大的 Maven 插件，我们可以访问 http://m2eclipse.sonatype.org/ 了解更多该项目的信息。

本小节我们先介绍如何安装 m2eclipse 插件，在本书后续的章节中，我们会逐步介绍 m2eclipse 插件的使用。

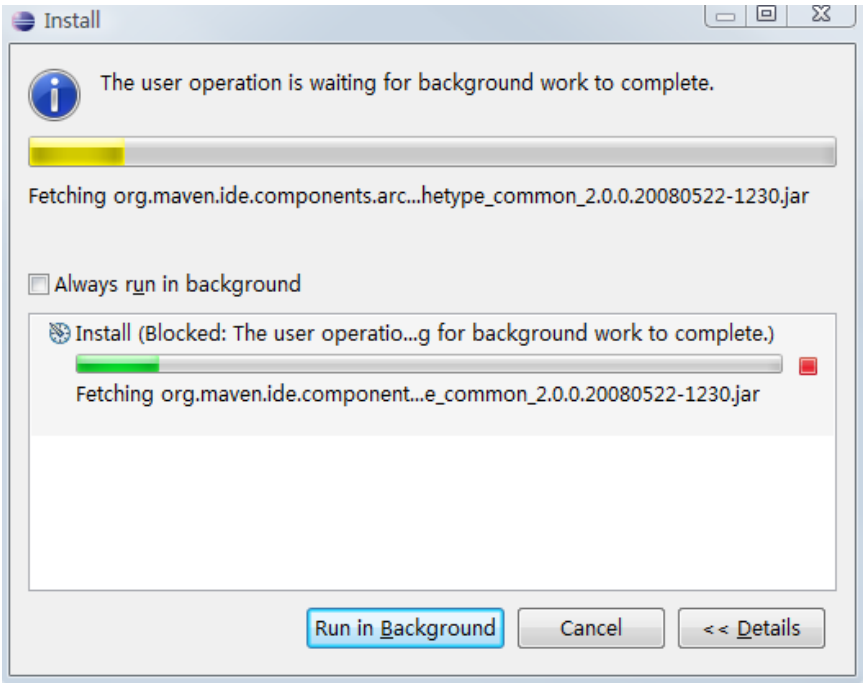
现在我以 Eclipse 3.5 为例逐步讲解 m2eclipse 的安装。启动 Eclipse 之后，在菜单栏中选择 **Help**，然后选择 **Install New Software...**，接着你会看到一个 Install 对话框，在 **Work with:** 字段中输入 *http://m2eclipse.sonatype.org/update/*，然后点击 **Add**，然后你会得到一个新的 Add Site 对话框，在 **Name** 字段中输入 *m2eclipse*，点击 **OK**。Eclipse 会下载 m2eclipse 安装站点上的资源信息。等待资源载入完成之后，我们再将其全部展开，就能看到如下的界面：



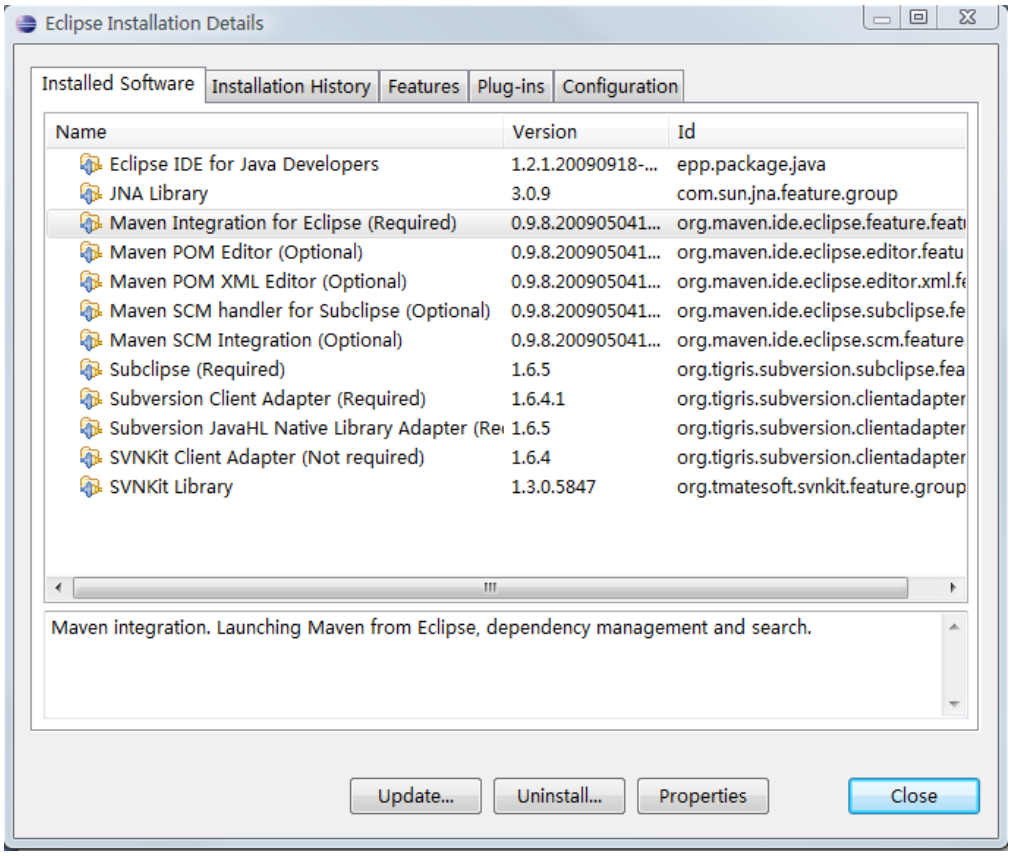
看到这么多的安装项，你可能会被吓到，其实这里面大多数模块都是可选的(Optional)，很多我们都不需要关心。下面我根据重要程度简要介绍一下这些模块，一般来说，你需要安装那些重要的模块，然后再根据自己特定的需要看是否需要安装哪些不重要的模块。

- **重要的**
  - Maven Integration for Eclipse (Required)：正如名字所示，这是必须的组件，它是 m2eclipse 的核心，有了它 m2eclipse 才能工作。
  - Maven POM Editor (Optional)：该模块能够帮助我们更加直观的编辑 Maven 的 POM 文件，甚至能以图形化的方式帮助分析依赖，因此非常重要。
  - Maven POM XML Editor (Optional)：该模块能够帮助我们在 XML 模式下更加高效的编辑 POM 文件。
  - Maven Central repository index (Optional)：有了它，我们就能够在 Eclipse 中快速的浏览和搜索 Maven 中央仓库的构件。
  - 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，不过前提是你首先需要安装 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。
- **不重要的**
  - Maven Embedder：这是 m2eclipse 内嵌的 Maven 环境，但直接使用与命令行一致的 Maven 环境是更推荐的做法。
  - Maven: The Definitive Guide book (Optional)：我们完全可以通过 web 阅读本书，见 http://www.sonatype.com/books/maven-book/reference/public-book.html，本书还在不断更新中，其大部分已经被翻译成中文，不过更新速率较低，中文版见 http://www.sonatype.com/books/maven-book/reference_zh/public-book.html。
  - Maven Doxia Editors (Optional)：该模块能够帮助我们编辑 APT，FML，以及 XDoc 等文档格式，不过对于大部分 Maven 用户来说，不需要该功能。
  - Maven issue tracking configurator for Mylyn (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 issue tracking configurator for Mylyn 3.x (Optional)：同上。
  - 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 AJDT (Optional)：使用该模块我们可以让 Eclipse 自动读取 POM 中的 AspectJ 信息并配置 AspectJ 项目。
  - Maven Integration for WTP (Optional)：使用该模块我们可以让 Eclipse 自动读取 POM 信息并配置 WTP 项目。

现在，根据你自己的需要选择想要安装的模块，然后点击 **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 >**，这个时候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，如下图：

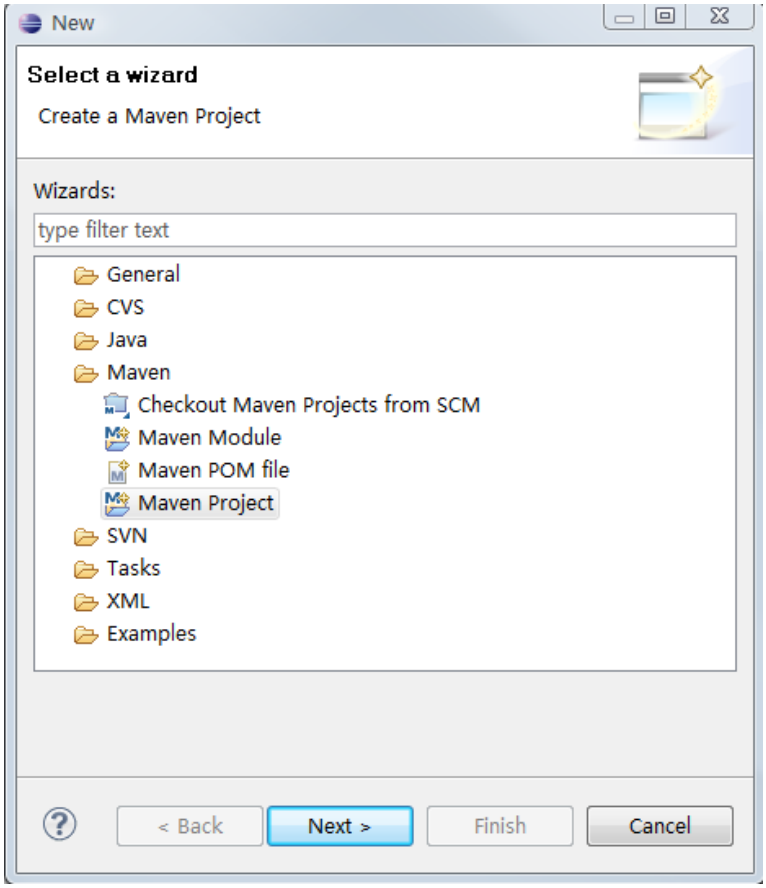


待安装完毕后，重启 Eclipse，现在让我们验证一下 m2eclipse 是否被正确安装了。首先，点击菜单栏中的 **Help**，然后选择 **About Eclipse**，在弹出的对话框中，点击 **Installation Details** 按钮，你会得到一个对话框，在 **Installed Software** 标签栏中，检查刚才我们选择的模块是否在这个列表中，如下图：



如果一切没问题，我们再检查一下现在 Eclipse 是否已经支持创建 Maven 项目，点击菜

单栏中的 **File**，然后选择 **New**，再选择 **Other**，在弹出的对话框中，找到 Maven 一项，再将其展开，你应该能够看到如下的对话框：



如果一切正常，说明 m2eclipse 已经正确安装了。

最后关于 m2eclipse 安装需要提醒的一点是，你可能会在使用 m2eclipse 的时候遇到类似这样的错误：

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

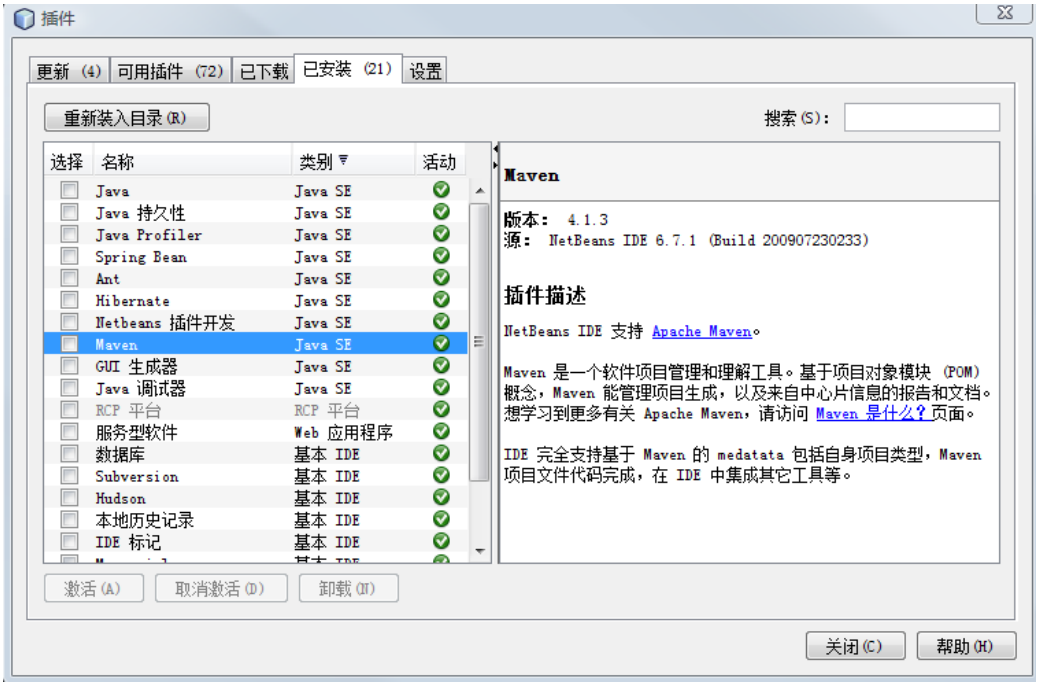
这是因为 eclipse 默认是运行在 JRE 上的，而 m2eclipse 的一些功能要求使用 JDK，解决方法是配置 Eclipse 安装目录的 eclipse.ini 文件，添加 vm 配置指向 JDK，如：

```
--launcher.XXMaxPermSize
256m
-vm
D:\java\jdk1.6.0_07\bin\javaw.exe
-vmargs
-Dosgi.requiredJavaVersion=1.5
-Xms128m
-Xmx256m
```

2.6 安装 NetBeans Maven 插件

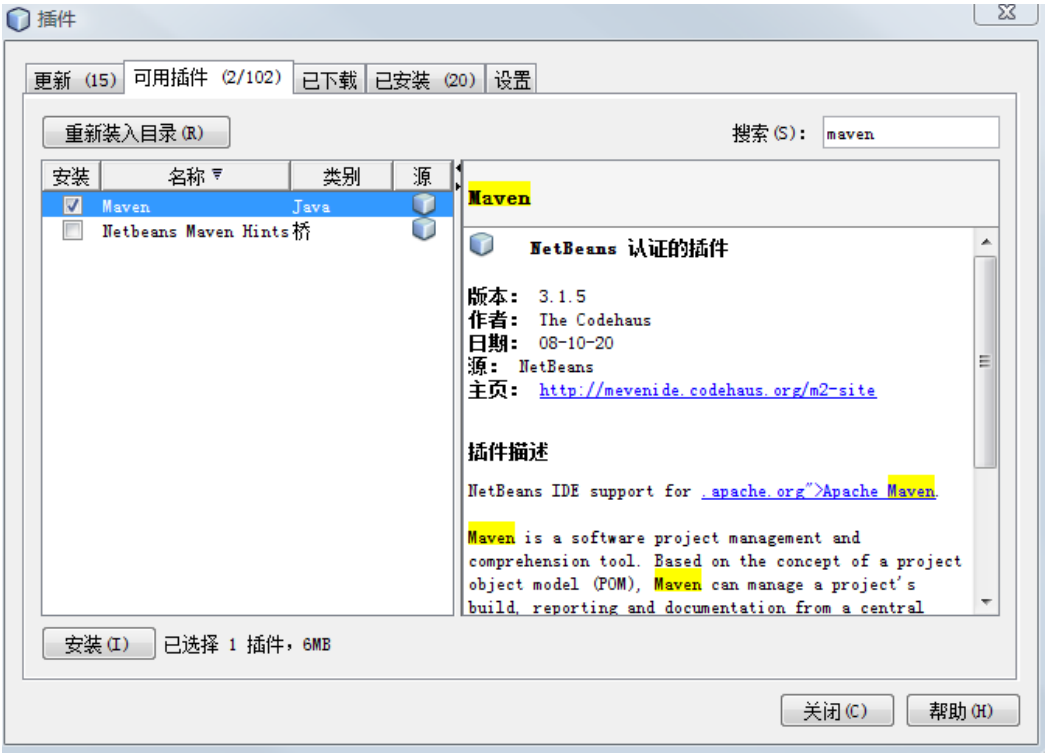
NetBeans IDE 无疑也是最流行的 JAVA IDE 之一，关于 Eclipse 和 NetBeans 孰优孰劣的争论可以 Google 出一大堆来，如果你更偏爱 NetBeans，那么肯定会对 NetBeans 与 Maven 的集成感兴趣。本章的这个小节先介绍如何在 NetBeans 上安装 Maven 插件，在后面的章节，我们还会介绍 NetBeans 中具体的 Maven 操作。

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，如下图：



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，如下图：



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，结束之后会提示我们安装完成。之后我们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，我们可以看一下 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**，这时我们应该能够看到项目类别中有 **Maven** 一项，选择该类别，右边会相应的显示 **Maven 项目**和**基于现有 POM 的 Maven 项目**，如下图所示：

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

2.7 Maven 安装最佳实践

本节介绍一些在安装 Maven 过程中不是必须的，但十分有用的实践。

2.7.1 设置 MAVEN_OPTS 环境变量

本章前面介绍 Maven 安装目录的时候我们了解到，运行 mvn 命令实际上是执行了 java 命令，既然是运行 java，那么一些运行 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处。

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

关于如何设置环境变量，请参考前面设置 M2_HOME 环境变量的做法，尽量不要直接修改 mvn.bat 或者 mvn 这两个 Maven 执行脚本文件。因为如果修改了脚本文件，升级 Maven 的时候你就不得不再次修改，一来麻烦，而来容易忘记。同理，我们应该尽可能地不去修改任何 Maven 安装目录下的文件。

2.7.2 配置用户范围 settings.xml

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

我们推荐使用用户范围的 settings.xml，主要原因是为了避免无意识地影响到系统中其它用户，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，则当然应该使用全局范围的 settings.xml。

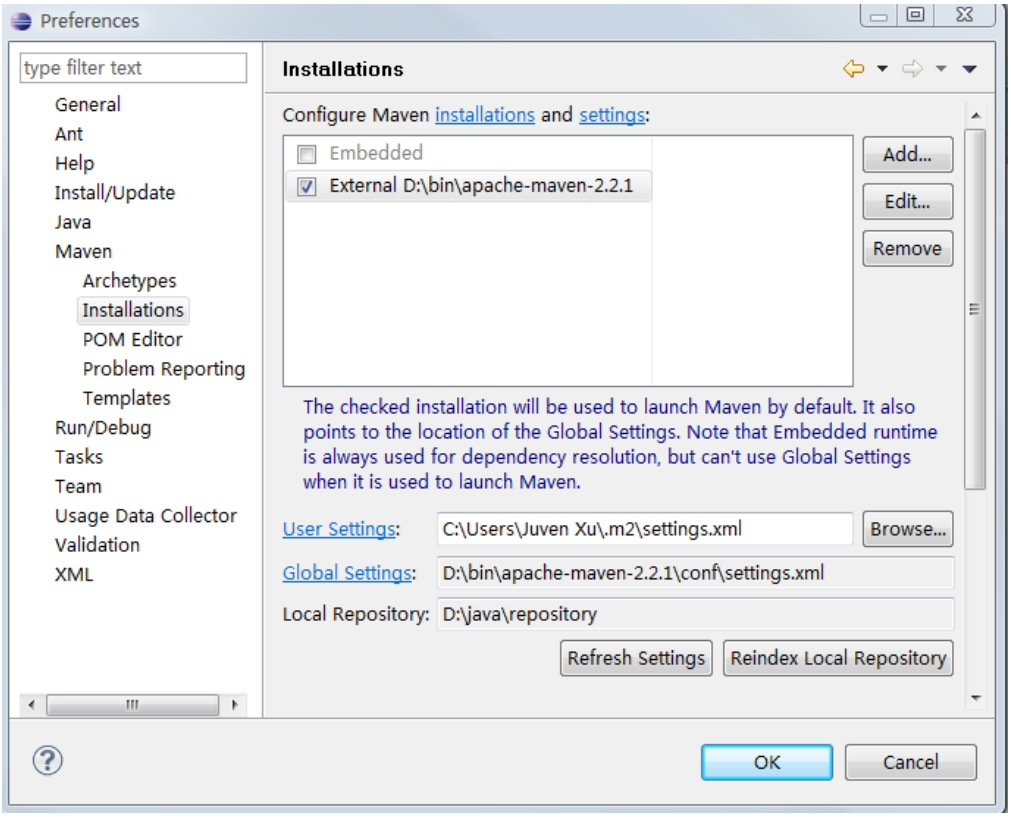
除了影响范围这一因素，配置用户范围 settings.xml 文件还能够方便 Maven 的升级。直接修改 conf 目录下的 settings.xml 会造成 Maven 升级的不便，每次升级到新版本的 Maven，我们就需要复制 settings.xml 文件，而如果使用*~/.m2* 目录下的 settings.xml，就不会影响到 Maven 安装文件，升级时就不需要触动 settings.xml 文件。

2.7.3 不要使用 IDE 内嵌的 Maven

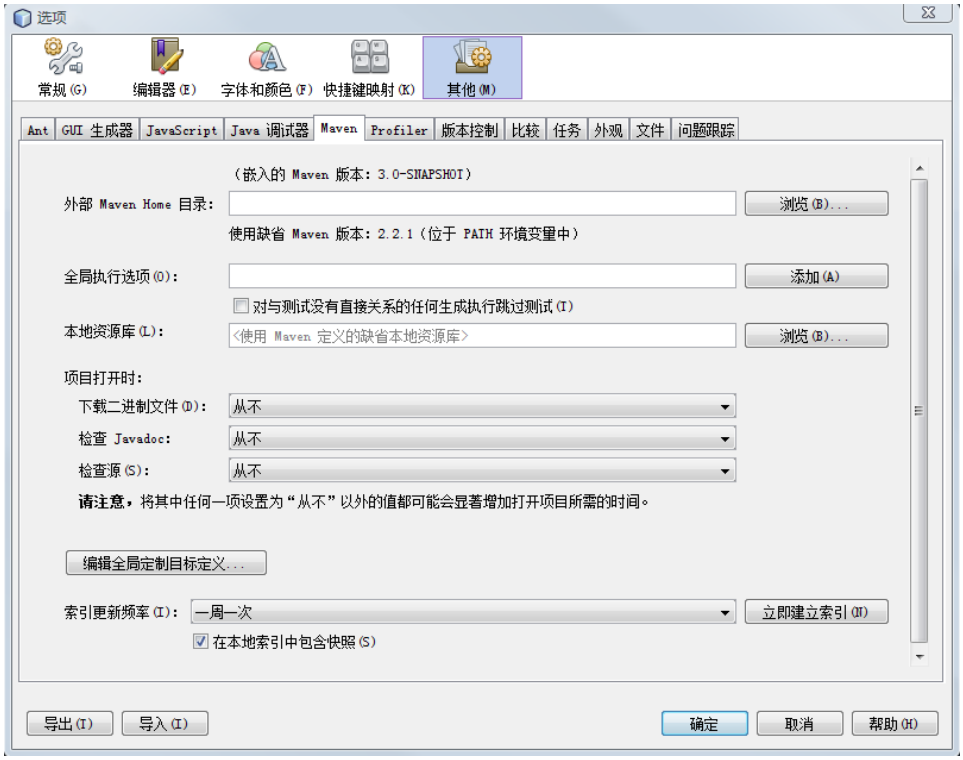
不论是 Eclipse 还是 NetBeans，当我们集成 Maven 的时候，都会安装上一个内嵌的 Maven，这个内嵌的 Maven 通常会比较新，但不一定很稳定，当然往往也会和我们在命令行使用的 Maven 不是同一个版本。这里有会出现两个潜在的问题，首先，较新版本的 Maven 存在很多不稳定因素，容易造成一些难以理解的问题。其次，由于除了 IDE，我们也经常还会使用命令行的 Maven，如果版本不一致，容易造成构建行为的不一致，这是我们所不希望看到的。因此，我们应该在 IDE 中配置 Maven 插件使用与命令行一致的 Maven。

在 m2eclipse 环境中，点击菜单栏中的 **Windows**，然后选择 **Prefrences**，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。如下图：



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。点击菜单栏中的**工具**，选择**选项**，选择**其他**标签栏，再选择 **Maven** 子标签栏，你就能看到如下的配置：


选项
常规(G)
编辑器(E)
字体和颜色(F)
快捷键映射(K)
其他(M)
Ant
GUI 生成器
JavaScript
Java 调试器
Maven
Profiler
版本控制
比较
任务
外观
文件
问题跟踪
（嵌入的 Maven 版本：3.0-SNAPSHOT）
外部 Maven Home 目录：
浏览(B)...
使用缺省 Maven 版本：2.2.1（位于 PATH 环境变量中）
全局执行选项(O)：
添加(A)
对与测试没有直接关系的任何生成执行跳过测试(T)
本地资源库(L)：
<使用 Maven 定义的缺省本地资源库>
浏览(B)...
项目打开时：
下载二进制文件(D)：
从不
检查 Javadoc：
从不
检查源(S)：
从不
请注意，将其中任何一项设置为“从不”以外的值都可能会显著增加打开项目所需的时间。
编辑全局定制目标定义...
索引更新频率(I)：
一周一次
立即建立索引(N)
在本地索引中包含快照(S)
导出(I)
导入(I)
确定
取消
帮助(H)

第三章 Hello World

到目前为止，我们已经大概了解了 Maven 是什么，并且已经在自己的操作环境中安装好了 Maven。现在，如同学习大多数软件技术一样，我们开始创建一个最简单的 Hello World 项目。如果你初次接触 Maven，我建议你按照本章的内容一步步地编写代码并执行，以得到最直观的感受，阅读永远无法替代实践的效果，可能在这个过程中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个解答。

3.1 编写 POM

就像 Make 的 Makefile，Ant 的 build.xml，Maven 项目的核心是 pom.xml。POM 的意思是项目对象模型（**P**roject **O**bject **M**odel），在这个模型中我们定义项目的基本信息，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 ch-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，输入其内容如下：

```
<?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.maven-book</groupId>
  <artifactId>ch-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。注意该 project 根元素还声明了一些 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 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，就像我们的身份证号码一样，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使用 groupId com.juvenxu.maven-book。

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 ch-hello-world，本书后面的其他章节代码会被分配其它的 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。

version 顾名思义指定了项目当前的版本，当前这个 Hello World 项目的版本是 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。本书的后面会详细 SNAPSHOT 以及如何使用 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，但我还是推荐为每一 POM 声明 name，以方便信息交流。

没有任何实际的 Java 代码，我们就已经能够定义一个 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的与实际代码相独立，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的时候，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 代码开发工作基本上不会涉及到 POM 的修改。

3.2 编写主代码

现在我们开始编写项目主代码，这里的主代码是和测试代码区分的，默认情况下，项目的主代码会被打包到最终的构件中，比如说 jar，而测试代码只在运行测试时用到，是不会被打包的。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/mavenbook/helloworld/*HelloWorld.java*，其内容如下：

```
package com.juvenxu.mavenbook.helloworld;

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

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

这是一个几乎不能再简单的 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，调用它的 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，这是 Maven 的约定，遵循了该约定，我们就无须额外的配置，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.mavenbook.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，虽然这不是必须的，但显然这样做更加清晰，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** ，我们会得到如下的输出：

```
[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\ch-hello-world\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] skip non existing resourceDirectory D: \code\ch-hello-world\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to D: \code\ch-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] ------------------------------------------------------------------------
```

这里 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这个 *target/*目录；接着执行 resources:resources 任务，由于我们没有定义项目资源，其实该任务什么都不做，我们暂且略过；最后 Maven 执行 compiler:compile 任务，将项目主代码编译至 *target/classes* 目录，打开该目录，我们就能看到编译好的类 *com/juvenxu/mavenbook/helloworld/HelloWorld.class*。

上面提到的 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 其实是 clean 插件的 clean 目标，compiler:compile 其实是 compiler 插件的 compile 目标，本书后面的章节会详细讲述 Maven 插件，以及如何自己编写 Maven 插件。

至此，我们没有进行任何额外的配置，就让 Maven 帮我们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 帮我们执行自动化测试。

3.3 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，前面一节我们了解到，Maven 项目中默认的主代码目录是 *src/main/java*，对应的，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。本书假设你了解 JUnit，即使你暂时不了解也没关系，样例测试代码会十分简单，很容易理解。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 如下：

```
<?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.maven-book</groupId>
  <artifactId>ch-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>
```

上述 XML 代码中我们添加了 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，这个依赖的 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，我们都不得不去 JUnit 的官网下载分发包，而 Maven 的神奇特色之一就是，它有一个中央仓库，你可以访问 http://repo1.maven.org/maven2/，这里有大量的类库，访问子目录 junit/junit/4.7/，我们就能看到 junit-4.7.pom 和 junit-4.7.jar。本书的后面章节会详细介绍 Maven 仓库及中央仓库。

上述 POM 代码中还有一个没有解释的元素是 scope，其值为 test，scope 为依赖范围，而当依赖范围是 test 的时候，就表示该依赖只对测试有效，换句话说，我们在测试代码中 import JUnit 代码是没有问题的，但是如果我们在主代码中 import JUnit 代码，就会造成编译错误。如果不声明依赖范围，那么就是默认值是 compile，表示该依赖对主代码和测试代码都有效。

配置好了测试依赖，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”，我们在 src/test/java 目录下创建文件，其内容如下：

```
package com.juvenxu.mavenbook.helloworld;

import org.junit.Assert;
import org.junit.Test;

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

        String result = helloWorld.sayHello();

        Assert.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，约定所有的需要执行测试方法都以 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。

编写完毕测试用例之后，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\maven-book\code\ch-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\ch-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\ch-hello-world\target\test-classes
[INFO] ------------------------------------------------------------------------
[ERROR] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Compilation failure
D:\code\ch-hello-world\src\test\java\com\juvenxu\mavenbook\helloworld\Hello
WorldTest.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，不过我们先耐心的分析一下这段输出（为了本书的简洁，一些不重要的信息我用省略号略去了）。我们在命令行输入的是 mvn clean test，而 Maven 实际执行的可不止 clean 和 test 两个任务，这里我们可以看到 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。

在前面的输出中我们还看到了 Maven 从中央仓库下载了 junit-4.7.pom 和 junit-4.7.jar 两个文件，这两个文件被下载到了本地仓库（*~/.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 的输出告诉我们需要使用 -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：

```
<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>
```

注意该 POM 省略了除插件配置以外的其它部分，我们暂且不去关心插件配置的细节，现在只需要知道这段配置会告诉 Maven 的 compiler 插件支持 Java 5 的编译。现在再执行 **mvn clean test**：

```
…
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to D: \code\ch-hello-world\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: D:\code\ch-hello-world\target\surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.juvenxu.mavenbook.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。很显然，我们的测试通过

了。当我们看到 BUILD SUCCESSFUL 的时候，终于可以稍微舒一口气了。

3.4 打包和运行

将项目进行编译，测试之后，下一个重要步骤是什么？是打包（package）。在 Hello World 的 POM 中我们没有指定打包类型，Maven 就会使用默认打包类型 jar，我们可以简单的执行命令 **mvn clean package** 进行打包，可以看到如下的输出：

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

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

类似的，在打包之前，Maven 会执行编译、测试等工作。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 *ch-hello-world-1.0-SNAPSHOT.jar* 的文件，该文件也位于 *target/*输出目录中。该文件实际上是根据规则 artifact-version.jar 规则进行命名的。

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 文件到其它项目的 classpath 中从而使用 HelloWorld 类。但是，如何才能让其它的 Maven 项目能够直接引用这个 jar 呢？我们还需要一个安装的步骤，执行 **mvn clean install**：

```
…
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: D: \code\ch-hello-world\target\ch-hello-world-1.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing D:\code\ch-hello-world\target\ch-hello-world-1.0-SNAPSHOT.jar to
C:\Users\juven\.m2\repository\com\juvenxu\maven-book\ch-hello-world\1.0-SNAP
SHOT\ch-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，其它 Maven 项目才能使用它。

到目前为止我们已经将 Maven 最主要的命令都体验了一遍，它们分别是 **mvn clean compile**、**mvn clean test**、**mvn clean package**、**mvn clean install**。并且我们知道，执行 test

之前是会先执行 compile 的，执行 package 之前是会先执行 test 的，而类似的，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 项目的，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 方法的类信息不会被添加到 manifest 中，我们可以打开 jar 文件中的 *META-INF/MANIFEST.MF* 文件，是看不到 Main-Class 一行的。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.mavenbook.helloworld.HelloWorld</mainClass>
          </transformer>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

注意 plugin 元素在 POM 中的相对位置，应该在<project><build><plugins>下面。这里我们暂且不去仔细研究这段插件配置，现在只要知道我们配置了 mainClass 为 com.juvenxu.mavenbook.helloworld.HelloWorld，这样项目在打包的时候就会将该信息放到 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 *ch-hello-world-1.0-SNAPSHOT.jar* 和 *original-ch-hello-world-1.0-SNAPSHOT.jar* 两个 jar 文件，第一个是带有 Main-Class 信息的可运行 jar，第二个是原始的 jar，打开 *ch-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.mavenbook.helloworld.HelloWorld
```

现在，我们在项目根目录中执行该 jar 文件：

```
D: \code\ch-hello-world>java -jar target\ch-hello-world-1.0-SNAPSHOT.jar
Hello Maven
```

我们看到控制台输出为 Hello Maven，这正是我们所期望的。

至此，我们完成了 Hello World 项目的介绍，当然，侧重点是在 Maven 上而非 Java 代码本身，我们介绍了 POM、Maven 项目结构、以及如何编译、测试、打包，等等。你应该已经对 Maven 有了一个比较直观的了解，并且可能会有一些疑问，不要着急，带着疑问继续往下阅读，往往能获得更好的效果。

3.5 使用 Archetype 生成项目骨架

在 Hello World 项目中，我们了解到了一些 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，它能帮助我们快速勾勒出项目骨架。

还是以 Hello World 为例，现在我们使用 maven archetype 来创建该项目的骨架，离开当前的 Maven 项目目录，运行如下命令：

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.0-alpha-4:generate
```

（注：很多资料中会让你直接使用更为简单的 mvn archetype:generate 命令，但这是不安全的，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。）

这里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-4 是本书编写时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.maven-book
Define value for artifactId: : ch-hello-world
Define value for version:  1.0-SNAPSHOT: :
Define value for package: com.juvenxu.maven-book: :
com.juvenxu.mavenbook.helloworld
Confirm properties configuration:
groupId: com.juvenxu.maven-book
artifactId: ch-hello-world
version: 1.0-SNAPSHOT
package: com.juvenxu.mavenbook.helloworld
```

接着 Archetype 插件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 ch-hello-world（我们定义的 artifactId）的子目录，打开子目录，我们能够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.mavenbook.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.mavenbook.helloworld.AppTest*。

使用 Archetype 我们可以迅速的构建起项目的骨架，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，这显然会节省我们大量的时间。

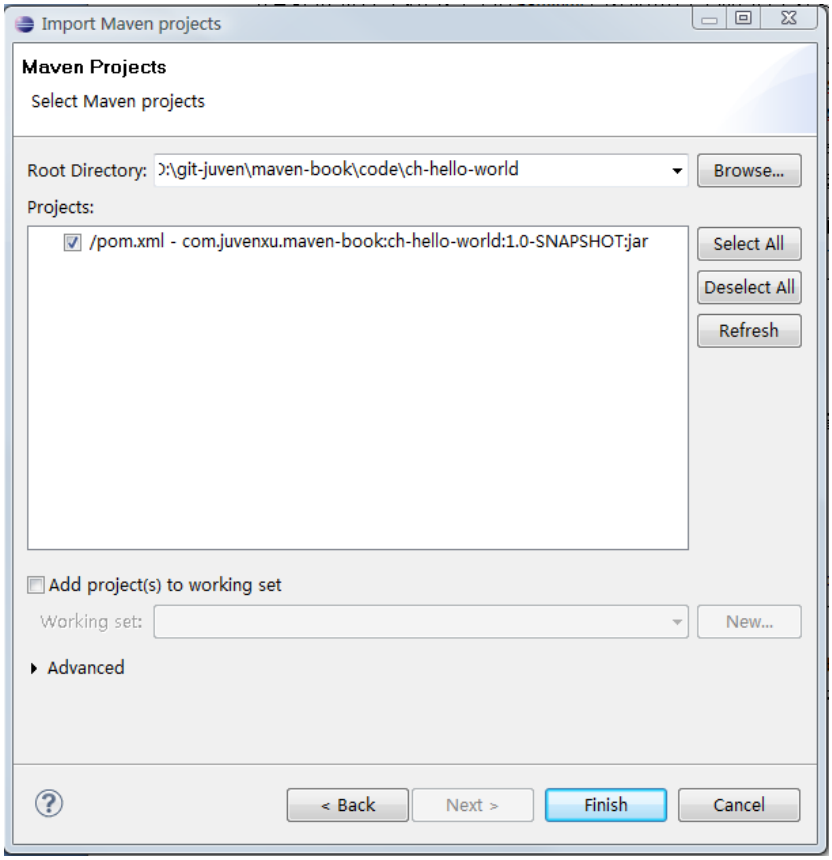
此外，我们这里仅仅是看到了一个最简单的 archetype，如果你有很多项目拥有类似的自定义项目结构以及配置文件，你完全可以一劳永逸地开发自己的 archetype，然后在这些项目中使用自定义的 archetype 来快速生成骨架，本书后面的章节会详细阐述如何开发 Maven Archetype。

3.6 m2eclipse 简单使用

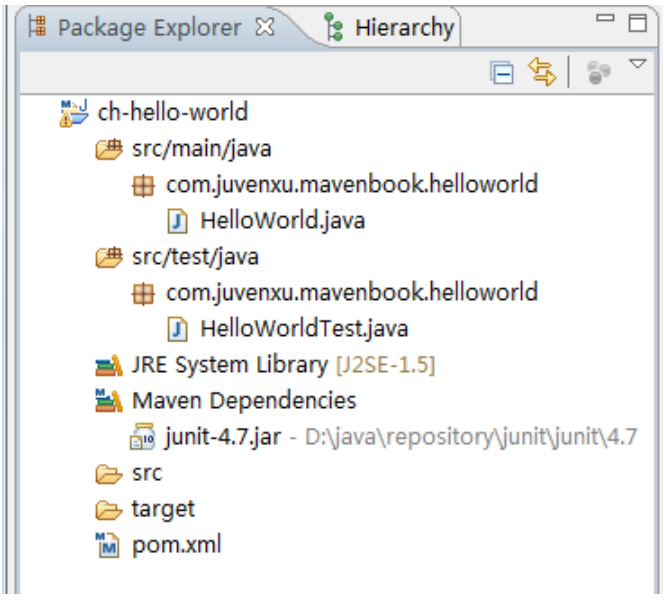
介绍前面 Hello World 项目的时候，我们并没有涉及到 IDE，如此简单的一个项目，使用最简单的编辑器也能很快完成，但对于稍微大一些的项目来说，没有 IDE 就是不可想象的，本节我们先介绍 m2eclipse 的基本使用。

3.6.1 导入 Maven 项目

本书第二章已经介绍了如何安装 m2eclipse，现在，我们使用 m2ecilpse 导入 Hello World 项目。操作十分简单，选择菜单项 **File**，接着选择 **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**，这是就能够看到一个 **Import Projects** 对话框中，在该对话框中点击 **Browse…**选择 Hello World 的根目录，也就是包含 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，如下图：



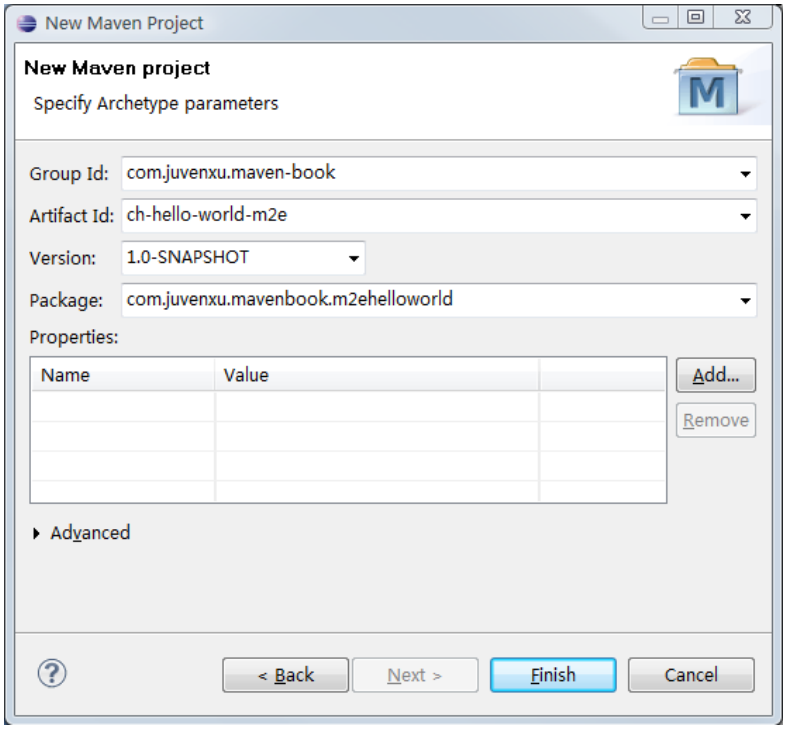
点击 Finish 之后，m2ecilpse 就会将该项目导入到当前的 workspace 中，导入完成之后，我们就可以在 Package Explorer 视图中看到如下的项目结构：



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*，本书的后续内容会介绍如何自定义本地仓库位置）。

3.6.2 创建 Maven 项目

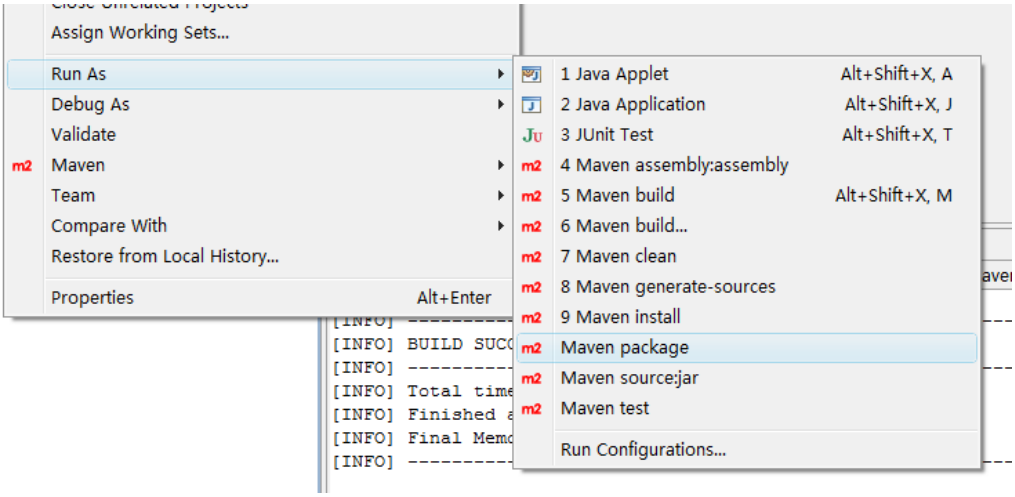
创建一个 Maven 项目也十分简单，选择菜单项 **File -> New -> Other**，在弹出的对话框中选择 Maven 下的 **Maven Project**，然后点击 **Next >**，在弹出的 **New Maven Project** 对话框中，我们使用默认的选项（不要选择 Create a simple project 选项，那样我们就能使用 Maven Archetype），点击 **Next >**，此时 m2eclipse 会提示我们选择一个 Archetype，我们选择 **maven-archetype-quickstart 1.0**，再点击 **Next >**，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），如下图：



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

3.6.3 运行 mvn 命令

在命令行，我们需要输入如 mvn clean install 之类的命令来执行 maven 构建，m2eclipse 中，当然也有对应的功能，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：



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，默认没有我们想要执行的 Maven 命令怎么办？比如，默认带有 mvn test，但我们想执行 mvn clean test，很简单，选择 **Maven buid...** 以自定义 Maven 运行命令，在弹出对话框中的 **Goals** 一项中输入我们想要执行的命令，如 clean test，设置一下 Name，点 **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。

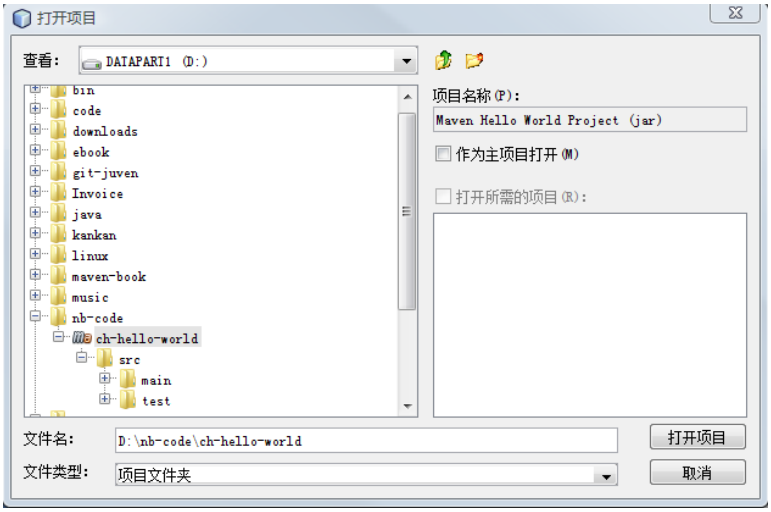
3.7 NetBeans Maven 插件简单使用

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

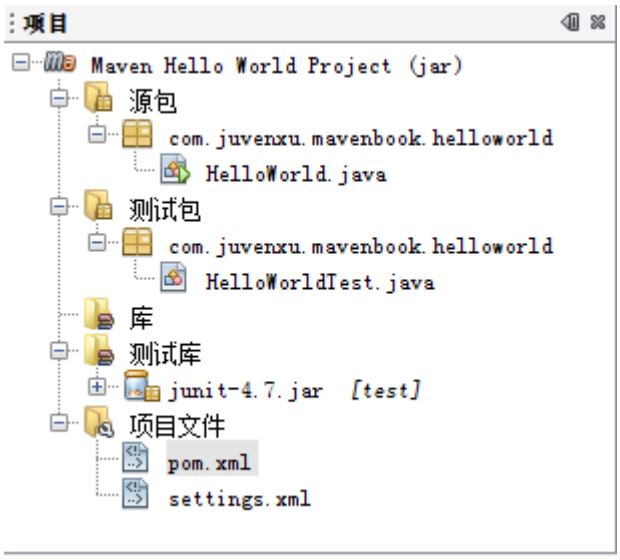
3.7.1 打开 Maven 项目

与其说*打开* Maven 项目，不如称之为*导入*来得更为合适，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 的菜单中使用的名称。

选择菜单栏中的**文件**，然后选择**打开项目**，接着直接定位到我们 Hello World 项目的根目录，NetBeans 会十分智能的识别出 Maven 项目，如下图所示：



我们能注意到 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，现在我们就能够在**项目**视图中看到如下的项目结构：



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

3.7.2 创建 Maven 项目

在 NetBeans 中创建 Maven 项目同样十分轻松，在菜单栏中选择**文件**，然后**新建项目**，在弹出的对话框中，选择项目类别为 **Maven**，项目为 **Maven 项目**，点击下一步之后，对话框接着会提示我们选择 Maven 原型，这里的 Maven 原型其实也就是 Maven Archtype，我们选择 **Maven 快速启动原型（1.0）**，其实也就是前面提到的 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 和讨论在 m2eclipse 中创建 Maven 项目的时候都仔细解释过，不再详述，参见下图：

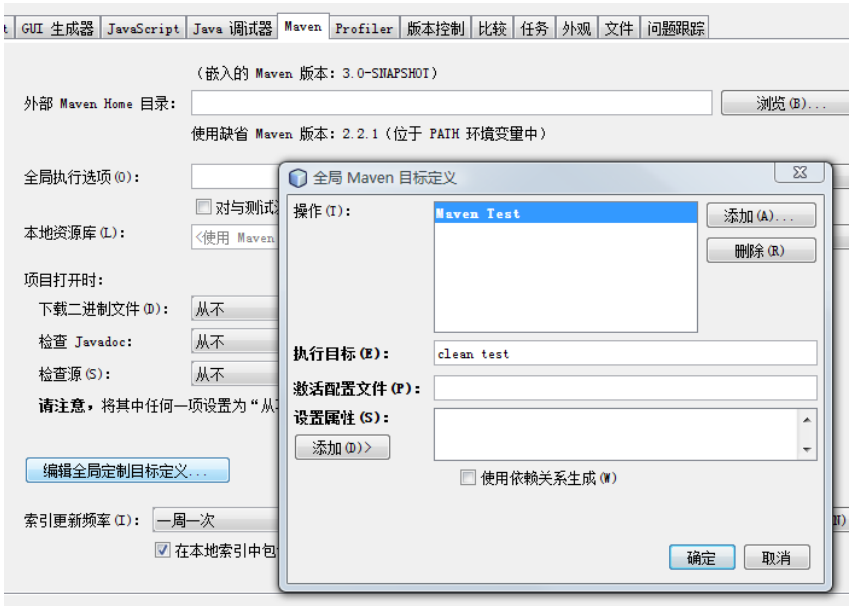
点击完成之后，一个新的 Maven 项目就创建好了。

3.7.3 运行 mvn 命令

默认情况下，NetBeans 给我们提供了两种 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，这个时候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（还记得上一章中我们如何配置 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，暂时不考虑其它配置选项，如下图：



点击缺点保存该配置，现在我们在 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，终端就会反映正在执行 mvn clean test。这里值得一提的是，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。