

Maven 权威指南

来源: http://books.sonatype.com/maven-book/reference_zh/public-book.html

Authors

Tim O'Brien (Sonatype, Inc.) , John Casey (Sonatype, Inc.) , Brian Fox (Sonatype, Inc.) , Bruce Snyder () , Jason Van Zyl (Sonatype, Inc.) , Juven Xu ()

Chapter 1. 介绍 Apache Maven

[1.1. Maven... 它是什么?](#)

[1.2. 约定优于配置 \(Convention Over Configuration\)](#)

[1.3. 一个一般的接口](#)

[1.4. 基于 Maven 插件的全局性重用](#)

[1.5. 一个“项目”的概念模型](#)

[1.6. Maven 是 Ant 的另一种选择么?](#)

[1.7. 比较 Maven 和 Ant](#)

[1.8. 总结](#)

虽然网络上有许多 Maven 的参考文章,但是没有一篇单独的,编写规范的介绍 Maven 的文字,它需要是一本细心编排的入门指南和参考手册。我们做的,正是试图提供这样的,包含许多使用参考的文字。

1.1. Maven... 它是什么?

如何回答这个问题要看你怎么看这个问题。绝大部分 Maven 用户都称 Maven 是一个“构建工具”:一个用来把源代码构建成可发布的构件的工具。构建工程师和项目经理会说 Maven 是一个更复杂的东西:一个项目管理工具。那么区别是什么?像 Ant 这样的构建工具仅仅是关注预处理,编译,打包,测试和分发。像 Maven 这样的—个项目管理工具提供了构建工具所提供功能的超集。除了提供构建的功能, Maven 还可以生成报告,生成 Web 站点,并且帮助推动工作团 队成员间的交流。

一个更正式的 [Apache Maven](#) 的定义: Maven 是一个项目管理工具,它包含了一个项目对象模型 (Project Object Model),一组标准集合,一个项目生命周期(Project Lifecycle),一个依赖管理系统(Dependency Management System),和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。当你使用 Maven 的时候,你用一个明确定义的项目对象模型来描述你的项目,然后 Maven 可以应用横切的逻辑,这些逻辑来自一组共享的(或者自定义的)插件。

别让 Maven 是一个“项目管理”工具的事实吓跑你。如果你只是在找一个构建工具, Maven 能做这个工作。事实上,本书的一些章节将会涉及使用 Maven 来构建和分发你的项目。

1.2. 约定优于配置 (Convention Over Configuration)

约定优于配置是一个简单的概念。系统，类库，框架应该假定合理的默认值，而非要求提供不必要的配置。流行的框架如 [Ruby on Rails](#) 和 [EJB3](#) 已经开始坚持这些原则，以对像原始的 [EJB 2.1](#) 规范那样的框架的配置复杂度做出反应。一个约定优于配置的例子就像 [EJB3](#) 持久化，将一个特殊的 [Bean](#) 持久化，你所需要做的只是将这个类标注为 `@Entity`。框架将会假定表名和列名是基于类名和属性名。系统也提供了一些钩子，当有需要的时候你可以重写这些名字，但是，在大部分情况下，你会发现使用框架提供的默认值会让你的项目运行的更快。

Maven 通过给项目提供明智的默认行为来融合这个概念。在没有自定义的情况下，源代码假定是在 `${basedir}/src/main/java`，资源文件假定是在 `${basedir}/src/main/resources`。测试代码假定是在 `${basedir}/src/test`。项目假定会产生一个 **JAR** 文件。**Maven** 假定你想要把编译好的字节码放到 `${basedir}/target/classes` 并且在 `${basedir}/target` 创建一个可分发的 **JAR** 文件。虽然这看起来无关紧要，但是想想大部分基于 **Ant** 的构建必须为每个子项目定义这些目录。**Maven** 对约定优于配置的应用不仅仅是简单的目录位置，**Maven** 的核心插件使用了一组通用的约定，以用来编译源代码，打包可分发的构件，生成 **web** 站点，还有许多其他的过程。**Maven** 的力量来自它的“武断”，它有一个定义好的生命周期和一组知道如何构建和装配软件的通用插件。如果你遵循这些约定，**Maven** 只需要几乎为零的工作——仅仅是将你的源代码放到正确的目录，**Maven** 将会帮你处理剩下的事情。

使用“遵循约定优于配置”系统的一个副作用是用户可能会觉得他们被强迫使用一种特殊的方法。当然 **Maven** 有一些核心观点不应该被怀疑，但是其实很多默认行为还是可配置的。例如项目源码的资源文件的位置可以被自定义，**JAR** 文件的名字可以被自定义，在开发自定义插件的时候，几乎任何行为可以被裁剪以满足你特定的环境需求。如果你不想遵循约定，**Maven** 也会允许你自定义默认值来适应你的需求。

1.3. 一个一般的接口

在 **Maven** 为构建软件提供一个一般的接口之前，每个单独的项目都专门有人来管理一个完全自定义的构建系统。开发人员必须在开发软件之外去学习每个他们要参与的新项目的构建系统的特点。在 2001 年，构建一个项目如 [Turbine](#) 和构建另外一个项目如 [Tomcat](#)，两者方法是完全不同的。如果一个新的进行静态源码分析的源码分析工具面世了，或者如果有人开发了一个新的单元测试框架，每个人都必须放下手头的工作去想办法使这个新东西适应每个项目的自定义构建环境。如何运行单元测试？世界上有一千种不同的答案。构建环境由无数无休止的关于工具和构建程序的争论所描述刻画。**Maven** 之前的时代是低效率的时代，是“构建工程师”的时代。

现在，大部分开源开发者已经或者正在使用 **Maven** 来管理他们新的软件项目。这种转变不仅仅是开发人员从一种构建工具转移到另外一种构建工具，更是开发人员开始为他们的项目采用一种一般的接口。随着软件系统变得越来越模块化，构建系统变得更复杂，而项目的数量更是如火箭般飞速上升。在 **Maven** 之前，当你想要从 **Subversion** 签出一个项目如 [Apache ActiveMQ](#) 或 [Apache ServiceMix](#)，然后从源码进行构建，你需要为每个项目留出一个小时来理解给它的构建系统。这个项目需要构建什么？需要现在什么类库？把类库放哪里？构建中我该运行什么目标？最好的情况下，理解一

个新项目的构建需要几分钟，最坏的情况下（例如 Jakarta 项目的旧的 Servlet API 实现），一个项目的构建特别的困难，以至于花了几个小时以后，新的贡献者也只能编辑源码和编译项目。现在，你只要签出源码，然后运行：**mvn install**。虽然 Maven 有很多优点，包括依赖管理和通过插件重用一般的构建逻辑，但它成功的最核心原因是它定义了构建软件的一般的接口。每当你看到一个使用 Maven 的项目如 [Apache Wicket](#)，你就可以假设你能签出它的源码然后使用 **mvn install** 构建它，没什么好争论的。你知道点火开关在哪里，你知道油门在右边，刹车在左边。

1.4. 基于 Maven 插件的全局性重用

Maven 的核心其实不做什么实际的事情，除了解析一些 XML 文档，管理生命周期与插件之外，它什么也不懂。Maven 被设计成将主要的职责委派给一组 Maven 插件，这些插件可以影响 Maven 生命周期，提供对目标的访问。绝大多数 Maven 的动作发生于 Maven 插件的目标，如编译源码，打包二进制代码，发布站点和其它构建任务。你从 Apache 下载的 Maven 不知道如何打包 WAR 文件，也不知道如何运行单元测试，Maven 大部分的智能是由插件实现的，而插件从 Maven 仓库获得。事实上，第一次你用全新的 Maven 安装运行诸如 **mvn install** 命令的时候，它会从中央 Maven 仓库下载大部分核心 Maven 插件。这不仅仅是一个最小化 Maven 分发包大小的技巧，这种方式更能让你升级插件以给你项目的构建提高能力。Maven 从远程仓库获取依赖和插件的这一事实允许了构建逻辑的全局性重用。

Maven Surefire 插件是负责运行单元测试的插件。从版本 1.0 发展到目前广泛使用的在 JUnit 基础上增加了 TestNG 测试框架支持的版本。这种发展并没有破坏向后兼容性，如果你使用之前 Surefire 插件编译运行你的 JUnit 3 单元测试，然后你升级到了最新版本的 Surefire 插件，你的测试仍然能成功运行。但是，我们又获得了新的功能，如果你想使用 TestNG 运行单元测试，那么感谢 Surefire 插件的维护者，你已经可以使用 TestNG 了。你也能运行支持注解的 JUnit 4 单元测试。不用升级 Maven 安装或者新装任何软件，你就能获得这些功能。更重要的是，除了 POM 中一个插件的版本号，你不需要更改你项目的任何东西。

这种机制不仅仅适用于 Surefire 插件，项目使用 Compiler 插件进行编译，通过 Jar 插件变成 JAR 文件，还有一些插件生成报告，运行 JRuby 和 Groovy 的代码，以及一些用来向远程服务器发布站点的插件。Maven 将一般的构建任务抽象成插件，同时这些插件得到了很好的维护以及全局的共享，你不需要从头开始自定义你项目的构建系统然后提供支持。你完全可以从 Maven 插件获益，这些插件有人维护，可以从远程仓库下载到。这就是基于 Maven 插件的全局性重用。

1.5. 一个“项目”的概念模型



Maven 维护了一个项目的模型，你不仅仅需要把源码编译成字节码，你还需要开发软件项目的描述信息，为项目指定一组唯一的坐标。你要描述项目的属性。项目的许可证是什么？谁开发这个项目，为这个项目做贡献？这个项目依赖于其它什么项目没有？Maven 不仅仅是一个“构建工具”，它不仅仅是在类似于 make 和 Ant 的工具的基础上的改进，它是包含了一组关于软件项目和软件开发的语义规则的平台。这个基于每一个项目定义模型实现了如下特征：

依赖管理

由于项目是根据一个包含组标识符，构件标识符和版本的唯一的坐标定义的。项目间可以使用这些坐标来声明依赖。

远程仓库

和项目依赖相关的，我们可以使用定义在项目对象模型（POM）中的坐标来创建 Maven 构件的仓库。

全局性构建逻辑重用

插件被编写成和项目模型对象（POM）一起工作，它们没有被设计成操作某一个已知位置的特定文件。一切都被抽象到模型中，插件配置和自定义行为都在模型中进行。

工具可移植性 / 集成

像 Eclipse, NetBeans, 和 IntelliJ 这样的工具现在有共同的地方来找到项目的信息。在 Maven 出现之前，每个 IDE 都有不同的方法来存储实际上是自定义项目对象模型(POM)的信息。Maven 标准化了这种描述，而虽然每个 IDE 仍然继续维护它的自定义项目文件，但这些文件现在可以很容易的由模型生成。

便于搜索和过滤构件

像 Nexus 这样的工具允许你使用存储在 POM 中的信息对仓库中的内容进行索引和搜索。

Maven 为软件项目的语义一致性描述的开端提供了一个基础。

1.6. Maven 是 Ant 的另一种选择么？



当然，Maven 是 Ant 的另一种选择，但是 [Apache Ant](#) 继续是一个伟大的，被广泛使用的工具。它已经是多年以来 Java 构建的统治者，而你很容易的在你项目的 Maven 构建中集成 Ant 构建脚本。这是 Maven 项目一种很常见的使用模式。而另一方面，随着越来越多的开源项目转移到 Maven 用它作为项目管理平台，开发人员开始意识到 Maven 不仅仅简化了构建管理任务，它也帮助鼓励开发人员的软件项目使用通用的接口。Maven 不仅仅是一个工具，它更是一个平台，当你只是将 Maven 考虑成 Ant 的另一种选择的时候，你是在比较苹果和橘子。“Maven”包含了很多构建工具以外的东西。

有一个核心观点使得所有的关于 Maven 和 Ant, Maven 和 Buildr, Maven 和 Grandle 的争论变得无关紧要。Maven 并不是完全根据你构建系统的机制来定义的，它不是为你构建的不同任务编写脚本，它提倡一组标注，一个一般的接口，一个生命周期，一个标准的仓库格式，一个标准的目录布局，等等。它当然也不太在意 POM 的格式正好是 XML 还是 YAML 还是 Ruby。它比这些大得多，Maven 涉及的比构建工具本身多得多。当本书讨论 Maven 的时候，它也设计到支持 Maven 的软件，系统和标准。Buildr, Ivy, Gradle, 所有这些工具都和 Maven 帮助创建的仓库格式交互，而你可以很容易的使用如 Nexus 这样的工具来支持一个完全由 Buildr 编写的构建。Nexus 将在本书后面介绍。

虽然 **Maven** 是很多类似工具的另一个选择？但社区需要向前发展，就要看清楚技术是资本经济中不友好的竞争者之间持续的、零和的游戏。这可能是大企业之前相互关联的方式，但是和开源社区的工作方式没太大关系。“谁是胜利者？**Ant** 还是 **Maven**”这个大标题没什么建设性意义。如果你非要我们来回答这个问题，我们会很明确的说作为构建的基本技术，**Maven** 是 **Ant** 的更好选择；同时，**Maven** 的边界在持续的移动，**Maven** 的社区也在持续的是试图找到新的方法，使其更通用，互操作性更好，更易协同工作。**Maven** 的核心财产是声明性构建，依赖管理，仓库管理，基于插件的高度和重用，但是当前，和开源社区相互协作以降低“企业级构建”的低效率这个目标来比，这些想法的特定实现没那么重要。

1.7. 比较 **Maven** 和 **Ant**



虽然上一节应该已经让你确信本书的作者没有兴趣挑起 **Apache Ant** 和 **Apache Maven** 之间的争执，但是我们认识到许多组织必须在 **Apache Ant** 和 **Apache Maven** 之间做一个选择。本节我们对比一下这两个工具。

Ant 在构建过程方面十分优秀，它是一个基于任务和依赖的构建系统。每个任务包含一组由 XML 编码的指令。有 `copy` 任务和 `javac` 任务，以及 `jar` 任务。在你使用 **Ant** 的时候，你为 **Ant** 提供特定的指令以编译和打包你的输出。看下面的例子，一个简单的 `build.xml` 文件：

Example 1.1. 一个简单的 **Ant build.xml** 文件

```
<project name="my-project" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist" location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
```

```

<target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>

<target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
</target>
</project>

```

在这个简单的 Ant 例子中，你能看到，你需要明确的告诉 Ant 你想让它做什么。有一个包含 javac 任务的编译目标用来将 src/main/java 的源码编译至 target/classes 目录。你必须明确告诉 Ant 你的源码在哪里，结果字节码你想存储在哪里，如何将这些字节码打包成 JAR 文件。虽然最近有些进展以帮助 Ant 减少程序，但一个开发者对 Ant 的感受是用 XML 编写程序语言。

用 Maven 样例与之前的 Ant 样例做个比较。在 Maven 中，要从 Java 源码创建一个 JAR 文件，你只需要创建一个简单的 pom.xml，将你的源码放在 \${basedir}/src/main/java，然后从命令行运行 **mvn install**。下面的样例 Maven pom.xml 文件能完成和之前 Ant 样例所做的同样的事情。

Example 1.2. 一个简单的 Maven pom.xml

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>

```

这就是你 pom.xml 的全部。从命令行运行 **mvn install** 会处理资源文件，编译源代码，运行单元测试，创建一个 JAR，然后把这个 JAR 安装到本地仓库以为其它项目提供重用性。不用做任何修改，你可以运行 **mvn site**，然后在 target/site 目录找到一个 index.html 文件，这个文件链接了 JavaDoc 和一些关于源代码的报告。

诚然，这是一个最简单的样例项目。一个只包含源代码并且生成一个 JAR 的项目。一个遵循 Maven 的约定，不需要任何依赖和定制的项目。如果我们想要定制行为，我们的 pom.xml 的大小将会增加，在最大的项目中，你能看到一个非常复杂的 Maven POM 的集合，它们包含了大量的插件定制和依赖声明。但是，虽然你项目的 POM 文

件变得增大，它们包含的信息与一个差不多大小的基于 **Ant** 项目的构建文件的信息是完全不同的。**Maven POM** 包含声明：“这是一个 **JAR** 项目”，“源代码在 `src/main/java` 目录”。**Ant** 构建文件包含显式的指令：“这是一个项目”，“源代码在 `src/main/java`”，“针对这个目录运行 `javac`”，“把结果放到 `target/classes`”，“从.....创建一个 **JAR**”，等等。**Ant** 必须的过程必须是显式的，而 **Maven** 有一些“内置”的东西使它知道源代码在哪里，如何处理它们。

该例中 **Ant** 和 **Maven** 的区别是：

Apache Ant

- **Ant** 没有正式的约定如一个一般项目的目录结构，你必须明确的告诉 **Ant** 哪里去找源代码，哪里放置输出。随着时间的推移，非正式的约定出现了，但是它们还没有在产品中模式化。
- **Ant** 是程序化的，你必须明确的告诉 **Ant** 做什么，什么时候做。你必须告诉它去编译，然后复制，然后压缩。
- **Ant** 没有生命周期，你必须定义目标和目标之间的依赖。你必须手工为每个目标附上一个任务序列。

Apache Maven

- **Maven** 拥有约定，因为你遵循了约定，它已经知道你的源代码在哪里。它把字节码放到 `target/classes`，然后在 `target` 生成一个 **JAR** 文件。
- **Maven** 是声明式的。你需要做的只是创建一个 `pom.xml` 文件然后将源代码放到默认的目录。**Maven** 会帮你处理其它的事情。
- **Maven** 有一个生命周期，当你运行 **mvn install** 的时候被调用。这条命令告诉 **Maven** 执行一系列的有序的步骤，直到到达你指定的生命周期。遍历生命周期旅途中的一个影响就是，**Maven** 运行了许多默认的插件目标，这些目标完成了像编译和创建一个 **JAR** 文件这样的工作。

Maven 以插件的形式为一些一般的项目任务提供了内置的智能。如果你想要编写运行单元测试，你需要做的只是编写测试然后放到 `${basedir}/src/test/java`，添加一个对于 **TestNG** 或者 **JUnit** 的测试范围依赖，然后运行 **mvn test**。如果你想要部署一个 **web** 应用而非 **JAR**，你需要做的是改变你的项目类型为 **war**，然后把你文档根目录置为 `${basedir}/src/main/webapp`。当然，你可以用 **Ant** 做这些事情，但是你将需要从零开始写这些指令。使用 **Ant**，你首先需要确定 **JUnit JAR** 文件应该放在哪里，然后你需要创建一个包含这个 **JUnit JAR** 文件的 `classpath`，然后告诉 **Ant** 它应该从哪里去找测试源代码，编写一个目标来编译测试源代码为字节码，使用 **JUnit** 来执行单元测试。

没有诸如 **antlibs** 和 **Ivy** 等技术的支持（即使有了这些支持技术），**Ant** 给人感觉是自定义的程序化构建。项目中一组高效的坚持约定的 **Maven POM**，相对于 **Ant** 的配置文件，只有很少的 **XML**。**Maven** 的另一个优点是它依靠广泛公用的 **Maven** 插件。所有人使用 **Maven Surefire** 插件来运行单元测试，如果有人添加了一些针对新的测试框架的支持，你可以仅仅通过在你项目的 **POM** 中升级某个特定插件的版本来获得新的功能。

使用 **Maven** 还是 **Ant** 的决定不是非此即彼的，**Ant** 在复杂的构建中还有它的位置。如果你目前的构建包含一些高度自定义的过程，或者你已经写了一些 **Ant** 脚本通过一种明确的方法完成一个明确的过程，而这种过程不适合 **Maven** 标准，你仍然可以在 **Maven** 中用这些脚本。作为一个 **Maven** 的核心插件，**Ant** 还是可用的。自定义的插件可以用 **Ant** 来实现，**Maven** 项目可以配置成在生命周期中运行 **Ant** 的脚本。

1.8. 总结



我们刻意的使这篇介绍保持得简短。我们略述了一些 **Maven** 定义的要点，它们合起来是什么，它是基于什么改进的，同时介绍了其它构建工具。下一章将深入一个简单的项目，看 **Maven** 如何能够通过最小数量的配置来执行典型的任务。

Chapter 2. 安装和运行 Maven

[2.1. 验证你的 Java 安装](#)

[2.2. 下载 Maven](#)

[2.3. 安装 Maven](#)

[2.3.1. 在 Mac OSX 上安装 Maven](#)

[2.3.2. 在 Microsoft Windows 上安装 Maven](#)

[2.3.3. 在 Linux 上安装 Maven](#)

[2.3.4. 在 FreeBSD 或 OpenBSD 上安装 Maven](#)

[2.4. 验证 Maven 安装](#)

[2.5. Maven 安装细节](#)

[2.5.1. 用户相关配置和仓库](#)

[2.5.2. 升级 Maven](#)

[2.6. 获得 Maven 帮助](#)

[2.7. 使用 Maven Help 插件](#)

[2.7.1. 描述一个 Maven 插件](#)

[2.8. 关于 Apache 软件许可证](#)

本章包含了在许多不同平台上安装 **Maven** 的详细指令。我们会介绍得尽可能详细以减少安装过程中可能出现的问题，而不去假设你已经熟悉安装软件和设置环境变量。本章的唯一前提是你已经安装了恰当的 **JDK**。如果你仅仅对安装感兴趣，读完[下载 Maven](#)和[安装 Maven](#)后，你就可以直接去看本书其它的部分。如果你对 **Maven** 安装的细节感兴趣，本章将会给你提供一个简要的介绍，以及 **Apache** 软件许可证，版本 2.0 的介绍。

2.1. 验证你的 Java 安装



尽管 **Maven** 可以运行在 **Java 1.4** 上，但本书假设你在至少 **Java 5** 上运行。尽管使用你操作系统上最新的稳定版本的 **JDK**。本书的例子在 **Java 5** 或者 **Java 6** 上都能运行。

```
% java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode, sharing)
```

Maven 能在所有的验证过的 Java™ 兼容的 JDK 上工作，也包括一些未被验证 JDK 实现。本书的所有样例是基于 Sun 官方的 JDK 编写和测试的。如果你正在使用 Linux，你可能需要自己下载 Sun 的 JDK，并且确定 JDK 的版本（运行 **java -version**）。目前 Sun 已经将 Java 开源，因此这种情况将来有望得到改进，将来很有可能 Sun JRE 和 JDK 会被默认安装在 Linux 上。但直到现在为止，你还是需要自己去下载。

2.2. 下载 Maven



你可以从 Apache Maven 项目的 web 站点下载 Maven：

<http://maven.apache.org/download.html>.

当你下载 Maven 的时候，确认你选择了最新版本的 Apache Maven。本书书写的时候最新版本是 Maven 2.0.9。如果你不熟悉 Apache 软件许可证，你需要在使用产品之前去熟悉该许可证的条款。更多的关于 Apache 软件许可证的信息可以 [Section 2.8, “关于 Apache 软件许可证”](#) 找到。

2.3. 安装 Maven



操作系统之间有很大的区别，像 Mac OSX 和微软的 Windows，而且不同版本 Windows 之间也有微妙的差别。幸运的是，在所有操作系统上安装 Maven 的过程，相对来说还是比较直接的。下面的小节概括了在许多操作系统上安装 Maven 的最佳实践。

2.3.1. 在 Mac OSX 上安装 Maven



你可以从 <http://maven.apache.org/download.html> 下载 Maven 的二进制版本。下载最新的，下载格式最方便你使用的版本。找个地方存放它，并把存档文件解开。如果你把存档文件解压到 `/usr/local/maven-2.0.9`；你可能会需要创建一个符号链接，那样就能更容易使用，当你升级 Maven 的时候也不再需要改变环境变量。

```
/usr/local % ln -s maven-2.0.9 maven
/usr/local % export M2_HOME=/usr/local/maven
/usr/local % export PATH=${M2_HOME}/bin:${PATH}
```

将 Maven 安装好后，你还需要做一些事情以确保它正确工作。你需要将它的 bin 目录（该例中为 `/usr/local/maven/bin`）添加到你的命令行路径下。你还需要设置 `M2_HOME` 环境变量，其对应值为 Maven 的根目录（该例中为 `/usr/local/maven`）。

Note

在 OS X Tiger 和 OS X Leopard 上安装指令是相同的。有报告称 Maven 2.0.6 正和 XCode 的预览版本一起发布。如果你安装了 XCode，从命令行运行 `mvn` 检查一下它是否可用。XCode 把 Maven 安装在了 `/usr/share/maven`。我们强烈建议安装最新版本的 Maven 2.0.9，因为随着 Maven 2.0.9 的发布很多 bug 被修正了，还有很多改进。

你还需要把 `M2_HOME` 和 `PATH` 写到一个脚本里，每次登陆的时候运行这个脚本。把下面的几行加入到 `.bash_login`。

```
export M2_HOME=/usr/local/maven
export PATH=${M2_HOME}/bin:${PATH}
```

一旦你把这几行加入到你的环境中，你就可以在命令行运行 Maven 了。

Note

这些安装指令假设你正运行 `bash`。

2.3.2. 在 Microsoft Windows 上安装 Maven



在 Windows 上安装 Maven 和在 Mac OS X 上安装 Maven 十分类似，最主要的区别在于安装位置和设置环境变量。在这里假设 Maven 安装目录是 `c:\Program Files\maven-2.0.9`，但是，只要你设置的正确的环境变量，把 Maven 安装到其它目录也一样。当你把 Maven 解压到安装目录后，你需要设置两个环境变量——`PATH` 和 `M2_HOME`。设置这两个环境变量，键入下面的命令：

```
C:\Users\tobrien > set M2_HOME=c:\Program Files\maven-2.0.9
C:\Users\tobrien > set PATH=%PATH%;%M2_HOME%\bin
```

在命令行设置环境变量后，你可以在当前会话使用 Maven，但是，除非你通过控制面板把它们加入系统变量，你将需要每次登陆系统的时候运行这两行命令。你应该在 Microsoft Windows 中通过控制面板修改这两个变量。

2.3.3. 在 Linux 上安装 Maven



遵循 [Section 2.3.1, “在 Mac OS X 上安装 Maven”](#) 的步骤，在 Linux 机器上安装 Maven。

2.3.4. 在 FreeBSD 或 OpenBSD 上安装 Maven



遵循 [Section 2.3.1, “在 Mac OS X 上安装 Maven”](#) 的步骤，在 FreeBSD 或者 OpenBSD 机器上安装 Maven。T

2.4. 验证 Maven 安装



当 Maven 安装完成后，你可以检查一下它是否真得装好了，通过在命令行运行 **mvn -v**。如果 Maven 装好了，你应该能看到类似于下面的输出。

```
$ mvn -v
Maven 2.0.9
```

如果你看到了这样的输出，那么 Maven 已经成功安装了。如果你看不到，而且你的操作系统找不到 **mvn** 命令，那么确认一下 PATH 和 M2_HOME 环境变量是否已经正确设置了。

2.5. Maven 安装细节



Maven 的下载文件只有大概 1.5 MiB，它能达到如此苗条的大小是因为 Maven 的内核被设计成根据需从远程仓库获取插件和依赖。当你开始使用 Maven，它会开始下载插件到本地仓库中，就像 [Section 2.5.1, “用户相关配置和仓库”](#) 所描述的那样。对此你可能比较好奇，让我们先很快的看一下 Maven 的安装目录是什么样子。

```
/usr/local/maven $ ls -pl
LICENSE.txt
NOTICE.txt
README.txt
bin/
boot/
conf/
lib/
```

LICENSE.txt 包含了 Apache Maven 的软件许可证。[Section 2.8, “关于 Apache 软件许可证”](#) 会详细描述该许可证。NOTICE.txt 包含了一些 Maven 依赖的类库所需要的通告及权限。README.txt 包含了一些安装指令。bin/ 目录包含了运行 Maven 的 mvn 脚本。boot/ 目录包含了一个负责创建 Maven 运行所需要的类装载器的 JAR 文件(classwords-1.1.jar)。conf/ 目录包含了一个全局的 settings.xml 文件，该文件用来自定义你机器上 Maven 的一些行为。如果你需要自定义 Maven，更通常的做法是覆写 ~/.m2 目录下的 settings.xml 文件，每个用户都有对应的这个目录。lib/ 目录有了一个包含 Maven 核心的 JAR 文件(maven-2.0.9-uber.jar)。

2.5.1. 用户相关配置和仓库



当你不再仅仅满足于使用 Maven，还想扩展它的时候，你会注意到 Maven 创建了一些本地的用户相关的文件，还有在你 home 目录的本地仓库。在 ~/.m2 目录下有：

```
~/.m2/settings.xml
```

该文件包含了用户相关的认证，仓库和其它信息的配置，用来自定义 Maven 的行为。

~/.m2/repository/

该目录是你本地的仓库。当你从远程 **Maven** 仓库下载依赖的时候，**Maven** 在你本地仓库存储了这个依赖的一个副本。

Note

在 Unix(和 OSX)上，可以用 ~ 符号来表示你的 home 目录，（如~/bin 表示 /home/tobrien/bin）。在 Windows 上，我们仍然使用 ~ 来表示你的 home 目录。在 Windows XP 上，你的 home 目录是 C:\Documents and Settings\tobrien，在 Windows Vista 上，你的 home 目录是 C:\Users\tobrien。从现在开始，你应该能够理解这种路径表示，并翻译成你操作系统上的对应路径。

2.5.2. 升级 Maven



如果你遵循 [Section 2.3.1, “在 Mac OSX 上安装 Maven”](#)和 [Section 2.3.3, “在 Linux 上安装 Maven”](#)，在 Mac OSX 或者 Unix 机器上安装了 **Maven**。那么把 **Maven** 升级成较新的版本是很容易的事情。只要在当前版本 **Maven**

（/usr/local/maven-2.0.9）旁边安装新版本的 **Maven**

（/usr/local/maven-2.future），然后将符号链接 /usr/local/maven 从 /usr/local/maven-2.0.9 改成 /usr/local/maven-2.future 即可。你已经将 M2_HOME 环境变量指向了 /usr/local/maven，因此你不需要更改任何环境变量。

如果你在 Windows 上安装了 **Maven**，将 **Maven** 解压到 c:\Program Files\maven-2.future，然后更新你的 M2_HOME 环境变量。

2.6. 获得 Maven 帮助



虽然本书的目的是作为一本全面的参考手册，但是仍然会有一些主题我们会不小心遗漏，一些特殊情况 and 技巧我们也覆盖不到。**Maven** 的核心十分简单，它所做的工作其实都交给插件了。插件太多了，以至于不可能在一本书上全部覆盖。你将会碰到一些本书没有涉及的问题，碰到这种情况，我们建议你在下面的地址去寻找答案。

<http://maven.apache.org>

你首先应该看看这里，**Maven** 的 web 站点包含了丰富的信息及文档。每个插件都有几页的文档，这里还有一系列“快速开始”的文档，它们是本书内容的十分有帮助的补充。虽然 **Maven** 站点包含了大量信息，它同时也可能让你迷惑沮丧。那里提供了一个自定义的 **Google** 搜索框，以用来搜索已有的 **Maven** 站点信息，它能比通用的 **Google** 搜索提供更好的结果。

Maven User Mailing List

Maven 用户邮件列表是用户问问题的地方。在你问问题之前，你可以先搜索一下之前的讨论，有可能找到相关问题。问一个已经问过的问题，而不先查一下该问题是否存在了，这种形式不太好。有很多有用的邮件列表归档浏览器，我们发现 **Nabble** 最有用。你可以在这里浏览邮件列表：

<http://www.nabble.com/Maven---Users-f178.html>。你也可以按照这里的指令来加入用户邮件列表：<http://maven.apache.org/mail-lists.html>。

<http://www.sonatype.com>

Sonatype 维护了一个本书的在线副本，以及其它 Maven 相关的指南。

Note

除去一些专门的 Maven 贡献者所做的十分优秀的工作，Maven web 站点组织的比较糟糕，有很多不完整的，甚至有时候有些误导人的文档片段。在整个 Maven 社区里，插件文档的一般标准十分缺乏，一些插件的文档十分的丰富，但是另外一些连基本的使用命令都没有。通常你最好是在用户邮件列表里面去搜索下解决方案。

2.7. 使用 Maven Help 插件



本书中，我们将会介绍 Maven 插件，讲述 Maven 项目对象模型(POM)文件，settings 文件，还有 profile。有些时候，你需要一个工具来帮助你理解一些 Maven 使用的模型，以及某个插件有什么可用的目标。Maven Help 插件能让你列出活动的 Maven Profile，显示一个实际 POM (effective POM)，打印实际 settings (effective settings)，或者列出 Maven 插件的属性。

Note

如果想看一下 POM 和插件的概念性介绍，参照第三章：一个简单的 Maven 项目。

Maven Help 插件有四个目标。前三个目标是—— active-profiles, effective-pom 和 effective-settings —— 描述一个特定的项目，它们必须在项目的目录下运行。最后一个目标—— describe ——相对比较复杂，展示某个插件或者插件目标的相关信息。

help:active-profiles

列出当前构建中活动的 Profile (项目的，用户的，全局的)。

help:effective-pom

显示当前构建的实际 POM，包含活动的 Profile。

help:effective-settings

打印出项目的实际 settings，包括从全局的 settings 和用户级别 settings 继承的配置。

help:describe

描述插件的属性。它不需要在项目目录下运行。但是你必须提供你想要描述插件的 groupId 和 artifactId。

2.7.1. 描述一个 Maven 插件



一旦你开始使用 **Maven**，你会花很多时间去试图获得 **Maven** 插件的信息：插件如何工作？配置参数是什么？目标是什么？你会经常使用 `help:describe` 目标来获取这些信息。通过 `plugin` 参数你可以指定你想要研究哪个插件，你可以传入插件的前缀（如 `help` 插件就是 `maven-help-plugin`），或者可以是 `groupId:artifact[:version]`，这里 **version** 是可选的。比如，下面的命令使用 `help` 插件的 `describe` 目标来输出 **Maven Help** 插件的信息。

```
$ mvn help:describe -Dplugin=help
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:    2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
the build environment. It includes the ability to view the effective
POM and settings files, after inheritance and active profiles
have been applied, as well as a describe a particular plugin goal to give
usage information.

...
```

通过设置 `plugin` 参数来运行 `describe` 目标，输出为该插件的 **Maven** 坐标，目标前缀，和该插件的一个简要介绍。尽管这些信息非常有帮助，你通常还是需要了解更多的详情。如果你想要 **Help** 插件输出完整的带有参数的目标列表，只要运行带有参数 `full` 的 `help:describe` 目标就可以了，像这样：

```
$ mvn help:describe -Dplugin=help -Dfull
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:    2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
the build environment. It includes the ability to view the effective
POM and settings files, after inheritance and active profiles
have been applied, as well as a describe a particular plugin goal to give
usage
information.

Mojos:
```

```
=====
Goal: 'active-profiles'
=====
```

```
Description:
```

```
Lists the profiles which are currently active for this build.
```

```
Implementation: org.apache.maven.plugins.help.ActiveProfilesMojo
```

```
Language: java
```

```
Parameters:
```

```
-----
[0] Name: output
```

```
Type: java.io.File
```

```
Required: false
```

```
Directly editable: true
```

```
Description:
```

```
This is an optional parameter for a file destination for the output of this
mojo...the
```

```
listing of active profiles per project.
```

```
-----
[1] Name: projects
```

```
Type: java.util.List
```

```
Required: true
```

```
Directly editable: false
```

```
Description:
```

```
This is the list of projects currently slated to be built by Maven.
```

```
-----
This mojo doesn't have any component requirements.
```

```
=====
... removed the other goals ...
```

该选项能让你查看插件所有的目标及相关参数。但是有时候这些信息显得太多了。这时候你可以获取单个目标的信息，设置 `mojo` 参数和 `plugin` 参数。下面的命令列出了 **Compiler** 插件的 `compile` 目标的所有信息

```
$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull
```

Note

什么？ Mojo ？ 在 Maven 里面， 一个插件目标也被认为是一个 “Mojo” 。

2.8. 关于 Apache 软件许可证



Apache Maven 是基于 Apache 许可证 2.0 版 发布的。如果你想阅读该许可证，你可以查阅 `${M2_HOME}/LICENSE.txt` 或者从开源发起组织的网站上查阅

<http://www.opensource.org/licenses/apache2.0.php>。

很有可能你正在阅读本书但你又不是律师。如果你想知道 Apache 许可证 2.0 版意味着什么，Apache 软件基金会收集了一个很有帮助的，关于许可证的常见问题解答

（FAQ）：<http://www.apache.org/foundation/licence-FAQ.html>。这里是对问题“我不是律师，所有这些是什么意思？”的回答。

它允许你：

- 自由的下载和使用 Apache 软件，无论是软件的整体还是部分， 也无论是出于个人目的，公司内部目的，还是商业目的。
- 在你创建的类库或分发版本里使用 Apache 软件。

它禁止你：

- 在没有正当的权限下重新分发任何源于 Apache 的软件或软件片段。
- 以任何可能声明或暗示基金会认可你的分发版本的形式下使用 Apache 软件基金会拥有的标志。
- 以任何可能声明或暗示你创建了 Apache 软件的形式下使用 Apache 软件基金会拥有的标志。

它要求你：

- 在你重新分发的包含 Apache 软件的软件里，包含一份该许可证的副本。
- 对于任何包含 Apache 软件的分发版本，提供给 Apache 软件基金会清楚的权限。

它不要求你：

- 在任何你再次发布的包含 Apache 软件的版本里，包含 Apache 软件本身源代码，或者你做的改动的源码。
- 提交你对软件的改动至 Apache 软件基金会（虽然我们鼓励这种反馈）。

• Part I. Maven 实战

- 第一本关于 Maven 的书是来自 [O'Reilly](#) 的 [Maven 开发者笔记](#)，这本书通过一系列步骤介绍 Maven。开发者笔记系列书籍背后的想法是，当开发人员和另一个开发人员坐在一起，经历他曾经用来学习和编码的整个思考过程，这样会学得最好。虽然开发者笔记系列成功了，但笔记格式有一个缺点：笔记被设计成“集中关注于目标”，它让你通过一系列步骤来完成某个特定的目标。而有大的参考书，或者“动物”书，提供全面的材料，覆盖了整个的课题。两种书都有优缺点，因此只出版一种书是有问题的。
- 为了阐明这个问题，考虑如下情形，数万的读者读完开发者笔记后，他们都知道如何创建一个简单的项目，比方说一个 Maven 项目从一组源文件创建一个 WAR。但是，当他们想知道更多的细节或者类似于 Assembly 插件的详细描述的时候，他们会陷入僵局。因为没有关于 Maven 的写得很好的参考手册，他们需要在 Maven 站点上搜寻插件文档，或者在一系列邮件列表中不停挑选。当人们真正开始钻研 Maven 的时候，他们开始在 Maven 站点上阅读无数的由很多不同的开发人员编写的粗糙的 HTML 文档，这些开发人员对为插件编写文档有着完全不同的想法：数百的开发人员有着不同的书写风格，语气以其方言。除去很多好心的志愿者所做的努力，在 Maven 站点上阅读插件文档，最好的情况下，令人有受挫感，最坏的情况下，成为了抛弃 Maven 的理由。很多情况下 Maven 用户感觉“被骗了”因为他们不能从站点文档上找到答案。
- 虽然第一本 Maven 开发者笔记吸引了很多 Maven 的新用户，为很多人培训了最基本的 Maven 用例，但同样多的读者，当他们不能找到准确的写得很好的参考材料的时候，感觉到了挫败感。很多年来，缺少可靠的（或者权威的）参考资料阻碍了 Maven 的发展；这也成了一种 Maven 用户社区成长的阻碍力量。本书想要改变这种情况，通过提供，第一：在 [Part I, “Maven 实战”](#) 中更新原本的 Maven 开发者笔记，第二：在 [Part II, “Maven Reference”](#) 中第一次尝试提供全面的参考。在你手里的（或者屏幕上的）实际上是二书合一。
- 本书的这个部分中，我们不会抛弃开发者笔记中的描述性推进方式，这是帮助人们“以实战方式”学习 Maven 的很有价值的材料。在本书的第一部分中我们“边做边介绍”，然后在 [Part II, “Maven Reference”](#) 中我们填充空白，钻研细节，介绍那些 Maven 新手不感兴趣的高级话题。[Part II, “Maven Reference”](#) 可能使用与样例项目无关的一个参考列表和一程序列表，而 [Part I, “Maven 实战”](#) 将由实际样例和故事驱动。读完 [Part I, “Maven 实战”](#) 后，你将会拥有几个月内开始使用 Maven 所需要的一切。只有当你需要通过编写定制插件来开始自定义你的项目，或者想了解特定插件细节的时候，才可能需要回到 [Part II, “Maven Reference”](#)。

Chapter 3. 一个简单的 Maven 项目

[3.1. 简介](#)

[3.1.1. 下载本章的例子](#)

[3.2. 创建一个简单的项目](#)

[3.3. 构建一个简单的项目](#)

[3.4. 简单的项目对象模型 \(Project Object Model\)](#)

[3.5. 核心概念](#)

[3.5.1. Maven 插件和目标 \(Plugins and Goals\)](#)

[3.5.2. Maven 生命周期 \(Lifecycle\)](#)

[3.5.3. Maven 坐标 \(Coordinates\)](#)

[3.5.4. Maven 仓库 \(Repositories\)](#)

[3.5.5. Maven 依赖管理 \(Dependency Management\)](#)

[3.5.6. 站点生成和报告 \(Site Generation and Reporting\)](#)

[3.6. 小结](#)

3.1. 简介



本章我们介绍一个用 Maven Archetype 插件从空白开始创建的简单项目。当你跟着这个简单项目的开发过程，你会看到这个简单的应用给我们提供了介绍 Maven 核心概念的机会。

在你开始使用 Maven 做复杂的，多模块的构建之前，我们需要从基础开始。如果你之前用过 Maven，你将会注意到这里很好的照顾到了细节。你的构建倾向于“只要能工作”，只有当你需要编写插件来自定义默认行为的时候，才需要深入 Maven 的细节。另一方面，当你需要深入 Maven 细节的时候，对 Maven 核心概念的彻底理解是至关重要的。本章致力于为你介绍一个尽可能简单的 Maven 项目，然后介绍一些使 Maven 成为一个可靠的构建平台的核心概念。读完本章后，你将会对构建生命周期 (build lifecycle)，Maven 仓库 (repositories)，依赖管理 (dependency management) 和项目对象模型 (Project Object Model) 有一个基本的理解。

3.1.1. 下载本章的例子



本章开发了一个十分简单的例子，它将被用来探究 Maven 的核心概念。如果你跟着本章表述的步骤，你应该不需要下载这些例子来重新创建那些 Maven 已经生成好的代码。我们将会使用 Maven Archetype 插件来创建这个简单的项目，本章不会以任何方式修改这个项目。如果你更喜欢通过最终的例子源代码来阅读本章，本章的例子项目和这本书的例子代码可以从这里下载到：

<http://www.sonatype.com/book/mvn-examples-1.0.zip> 或者

<http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>。解压存档文件到任何目录下，然后到 **ch03/** 目录。在 **ch03/** 目录你将看到一个名字为 **simple/** 的目录，它包含了本章的源代码。如果你希望在 Web 浏览器里看这些例子代码，访问 <http://www.sonatype.com/book/examples-1.0> 并且点击 **ch03/** 目录。

3.2. 创建一个简单的项目



开始一个新的 Maven 项目，在命令行使用 Maven Archetype 插件。

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                        -DartifactId=simple \

-DpackageName=org.sonatype.mavenbook
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] artifact org.apache.maven.plugins:maven-archetype-plugin: checking for
\
updates from central
[INFO]
-----
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:create] (aggregator-style)
[INFO] -----
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
checking for updates from central
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch03
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/svnw/sonatype/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: simple
[INFO] * End of debug info from resources from generated POM *
[INFO] Archetype created in dir: /Users/tobrien/svnw/sonatype/examples/simple
```

mvn 是 Maven2 的命令。archetype:create 称为一个 Maven 目标 (goal)。如果你熟悉 Apache Ant，一个 Maven 目标类似于一个 Ant 目标 (target)；它们都描述了将会在构建中完成的工作单元 (unit of work)。而像 `-Dname=value` 这样的对是将会被传到目标中的参数，它们使用 `-D` 属性这样的形式^[1]，类似于你通过命令行向 Java 虚拟机传递系统属性。archetype:create 这个目标的目的通过 archetype 快速创建一个项目。在这里，一个 archetype 被定义为“一个原始的模型或者类型，在它之后其它类似的东西与之匹配：一个原型(prototype)”。Maven 有许多可用的 archetype，从生成一个简单的 Swing 应用，到一个复杂的 Web 应用。本章我们用最基本的 archetype 来创建一个入门项目的骨架。这个插件的前缀是“archetype”，目标为“create”。^[1]

我们已经生成了一个项目，看一下 Maven 在 simple 目录下创建的目录结构：

```
simple/❶
simple/pom.xml❷
  /src/
  /src/main/❸
    /main/java
  /src/test/❹
    /test/java
```

这个生成的目录遵循 Maven 标准目录布局，我们之后会去看更多的细节，但是，现在让我们只是尝试了解这些基本的目录。

- ❶ Maven Archtype 插件创建了一个与 artifactId 匹配的目录——simple。这是项目的基础目录。
- ❷ 每个项目在文件 pom.xml 里有它的项目对象模型 (POM)。这个文件描述了这个项目，配置了插件，声明了依赖。
- ❸ 我们项目的源码或资源文件被放在了 src/main 目录下面。在我们简单 Java 项目这样的情况下，这个目录包含了一下 java 类和一些配置文件。在其它的项目中，它可能是 web 应用的文档根目录，或者还放一些应用服务器的配置文件。在一个 Java 项目中，Java 类放在 src/main/java 下面，而 classpath 资源文件放在 src/main/resources 下面。
- ❹ 我们项目的测试用例放在 src/test 下。在这个目录下面，src/test/java 存放像使用 JUnit 或者 TestNG 这样的 Java 测试类。目录 src/test/resources 下存放测试 classpath 资源文件。

Maven Archtype 插件生成了一个简单的类 org.sonatype.mavenbook.App，它是一个仅有 13 行代码的 Java，所做的只是在 main 方法中输出一行消息：

```
package org.sonatype.mavenbook;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

最简单的 Maven archetype 生成最简单的 Maven 项目：一个往标准输出打印“Hello World”的程序。

3.3. 构建一个简单的项目



一旦你遵循 [Section 3.2, “创建一个简单的项目”](#) 使用 **Maven Archetype** 插件创建了一个项目，你会希望构建并打包这个应用。想要构建打包这个应用，在包含 `pom.xml` 的目录下运行 **mvn install**。

```
$ mvn install
[INFO] Scanning for projects...
[INFO]
-----
-
[INFO] Building simple
[INFO]    task-segment: [install]
[INFO]
-----
-
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /simple/target/classes
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Compiling 1 source file to /simple/target/test-classes
[INFO] [surefire:test]
[INFO] Surefire report directory: /simple/target/surefire-reports

-----
T E S T S
-----

Running org.sonatype.mavenbook.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.105 sec

Results :

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

[INFO] [jar:jar]
[INFO] Building jar: /simple/target/simple-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing /simple/target/simple-1.0-SNAPSHOT.jar to \
~/.m2/repository/org/sonatype/mavenbook/ch03/simple/1.0-SNAPSHOT/ \
simple-1.0-SNAPSHOT.jar
```

你已经创建了，编译了，测试了，打包了，并且安装了(installed)最简单的 Maven 项目。在命令行运行它以向你自己验证这个程序能工作。

```
$ java -cp target/simple-1.0-SNAPSHOT.jar org.sonatype.mavenbook.App
Hello World!
```

3.4. 简单的项目对象模型 (Project Object Model)



当 Maven 运行的时候它向项目对象模型(POM)查看关于这个项目的信息。POM 回答类似这样的问题：这个项目是什么类型的？这个项目的名称是什么？这个项目的构建有自定义么？这里是一个由 Maven Archetype 插件的 create 目标创建的默认的 pom.xml 文件。

Example 3.1. Simple 项目的 pom.xml 文件

```
<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>org.sonatype.mavenbook.ch03</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

这个 pom.xml 文件是你将会面对的 Maven 项目中最基础的 POM，一般来说一个 POM 文件会复杂得多：定义多个依赖，自定义插件行为。最开始的几个元素——groupId, artifactId, packaging, version——是 Maven 的坐标(coordinates)，它们唯一标识了一个项目。name 和 url 是 POM 提供的描述性元素，它们给人提供了可阅读的名字，将一个项目关联到了项目 web 站点。最后，dependencies 元素定义了一个单独的，测试范围(test-scoped)依赖，依赖于称为 JUnit 的单元测试框架。这些话题将会 [Section 3.5, “核心概念”](#)被深入介绍，当前，你所需知道的是，pom.xml 是一个让 Maven 跑起来的文件。

当 Maven 运行的时候，它是根据项目的 pom.xml 里设置的组合来运行的，一个

最上级的 POM 定义了 Maven 的安装目录,在这个目录中全局的默认值被定义了, (可能) 还有一些用户定义的设置。想要看这个“有效的 (effective)”POM, 或者说 Maven 真正运行根据的 POM, 在 simple 项目的基础目录下跑下面的命令。

```
$ mvn help:effective-pom
```

一旦你运行了此命令, 你应该能看到一个大得多的 POM, 它暴露了 Maven 的默认设置

3.5. 核心概念



我们已经第一次运行了 Maven, 是时候介绍一些 Maven 的核心概念了。在之前的例子中, 我们生了一个项目, 它包含了一个 POM 和一些源代码, 它们一起组成了 Maven 的标准目录布局。之后你用生命周期阶段(phase)作为参数来运行 Maven, 这个阶段会提示 Maven 运行一系列 Maven 插件的目标。最后, 你把 Maven 构件(artifact)安装(install)到了你本地仓库(repository)。等等? 什么是生命周期? 什么是“本地仓库”? 下面的小结阐述了一些 Maven 的核心概念。

3.5.1. Maven 插件和目标 (Plugins and Goals)



在前面一节中, 我们用两种类型的命令行参数运行了 Maven。第一条命令是一条单个的插件目标, Archetype 插件的 create 目标。Maven 第二次运行是一个生命周期阶段 -install。为了运行单个的 Maven 插件目标, 我们使用 **mvn archetype:create** 这样的语法, 这里 archetype 是一个插件标识而 create 是目标标识。当 Maven 运行一个插件目标, 它向标准输出打印出插件标识和目标标识:

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 \
                        -DartifactId=simple \

-DpackageName=org.sonatype.mavenbook
...
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
        checking for updates from central
...
```

一个 Maven 插件是一个单个或者多个目标的集合。Maven 插件的例子有一些简单但核心的插件, 像 Jar 插件, 它包含了一组创建 JAR 文件的目标, Compiler 插件, 它包含了一组编译源代码和测试代码的目标, 或者 Surefire 插件, 它包含一组运行单元测试和生成测试报告的目标。而其它的, 更有专门的插件包括: Hibernate3 插件, 用来集成流行的持久化框架 Hibernate, JRuby 插件, 它让你能够让运行 ruby 称为 Maven 构建的一部分或者用 Ruby 来编写 Maven 插件。Maven 也提供了自定义插件的能力。一个定制的插件可以用 Java 编写, 或者用一些其它的语言如 Ant, Groovy, beanshell 和之前提到的 Ruby。

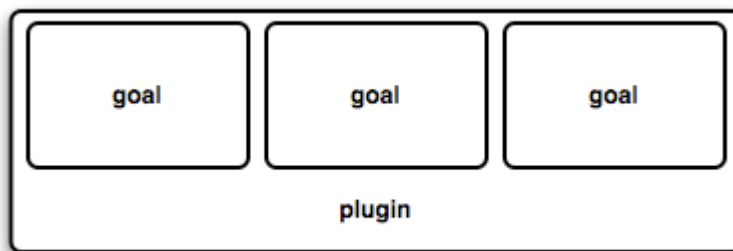


Figure 3.1. 一个插件包含一些目标

一个目标是一个明确的任务，它可以作为单独的目标运行，或者作为一个大的构建的一部分和其它目标一起运行。一个目标是 Maven 中的一个“工作单元(unit of work)”。目标的例子包括 **Compiler** 插件中的 `compile` 目标，它用来编译项目中的所有源文件，或者 **Surefire** 插件中的 `test` 目标，用来运行单元测试。目标通过配置属性进行配置，以用来定制行为。例如，**Compiler** 插件的 `compile` 目标定义了一组配置参数，它们允许你设置目标 JDK 版本或者选择是否用编译优化。在之前的例子中，我们通过命令行参数 **-DgroupId=org.sonatype.mavenbook.ch03** 和 **-DartifactId=simple** 向 **Archetype** 插件的 `create` 目标传入了 `groupId` 和 `artifactId` 配置参数。我们也向 `create` 目标传入了 `packageName` 参数，它的值为 `org.sonatype.mavenbook`。如果我们忽略了 `packageName` 参数，那么包名的默认值为 `org.sonatype.mavenbook.ch03`。

Note

当提到一个插件目标的时候，我们常常用速记符号：`pluginId:goalId`。例如，当提到 **Archetype** 插件的 `create` 目标的时候，我们写成 `archetype:create`。目标定义了一些参数，这些参数可以定义一些明智的默认值。在 `archetype:create` 这个例子中，我们并没有在命令行中指定这个目标创建什么类型的 **archetype**，我们简单的传入一个 `groupId` 和一个 `artifactId`。这是我们对于约定优于配置(*convention over configuration*)的第一笔。这里 `create` 目标的约定，或者默认值，是创建一个简单的项目，叫做 **Quickstart**。`create` 目标定义了一个配置属性 `archetypeArtifactId`，它有一个默认值为 `maven-archetype-quickstart`。**Quickstart** **archetype** 生成了一个最小项目的躯壳，包括一个 POM 和一个类。**Archetype** 插件比第一个例子中的样子强大得多，但是这是一个快速开始新项目的不错的方法。在本书的后面，我们将会让你看到 **Archetype** 插件可以用来生成复杂如 web 应用的项目，以及你如何能够使用 **Archetype** 插件来定义你自己项目的集合。

Maven 的核心对你项目构建中特定的任务几乎毫无所知。就它本身来说，Maven 不知道如何编译你的代码，它甚至不知道如何制作一个 JAR 文件，它把所有这些任务代理给了 Maven 插件，像 **Compiler** 插件和 **Jar** 插件，它们在需要的时候被下载下来并且定时的从 Maven 中央仓库更新。当你下载 Maven 的时候，你得到的是一个包含了基本躯壳的 Maven 核心，它知道如何解析命令行，管理 `classpath`，解析 POM 文件，在需要的时候下载 Maven 插件。通过保持 **Compiler**

插件和 Maven 核心分离，并且提供更新机制，用户很容易能使用编译器最新的版本。通过这种方式，Maven 插件提供了通用构建逻辑的全局重用性，有不会在构建周期中定义编译任务，有使用了所有 Maven 用户共享的 Compiler 插件。如果有对 Compiler 插件的改进，每个使用 Maven 的项目可以立刻从这种变化中得到好处。（并且，如果你不喜欢这个 Compiler 插件，你可以用你的实现来覆盖它）。

3.5.2. Maven 生命周期 (Lifecycle)



上一节中，我们运行的第二个命令是 **mvn package**。命令行并没有指定一个插件目标，而是指定了一个 Maven 生命周期阶段。一个阶段是在被 Maven 称为“构建生命周期”中的一个步骤。生命周期是包含在一个项目构建中的一系列有序的阶段。Maven 可以支持许多不同的生命周期，但是最常用的生命周期是默认的 Maven 生命周期，这个生命周期中一开始的一个阶段是验证项目的基本完整性，最后的一个阶段是把一个项目发布成产品。生命周期的阶段被特地留得含糊，单独的定义为验证(validation)，测试(testing)，或者发布(deployment)，而他们对不同项目来说意味着不同的事情。例如，打包(package)这个阶段在一个项目里生成一个 JAR，它也就意味着“将一个项目打成一个 jar”，而在另外一个项目里，打包这个阶段可能生成一个 WAR 文件。[Figure 3.2, “一个生命周期是一些阶段的序列”](#)展示了默认 Maven 生命周期的简单样子。

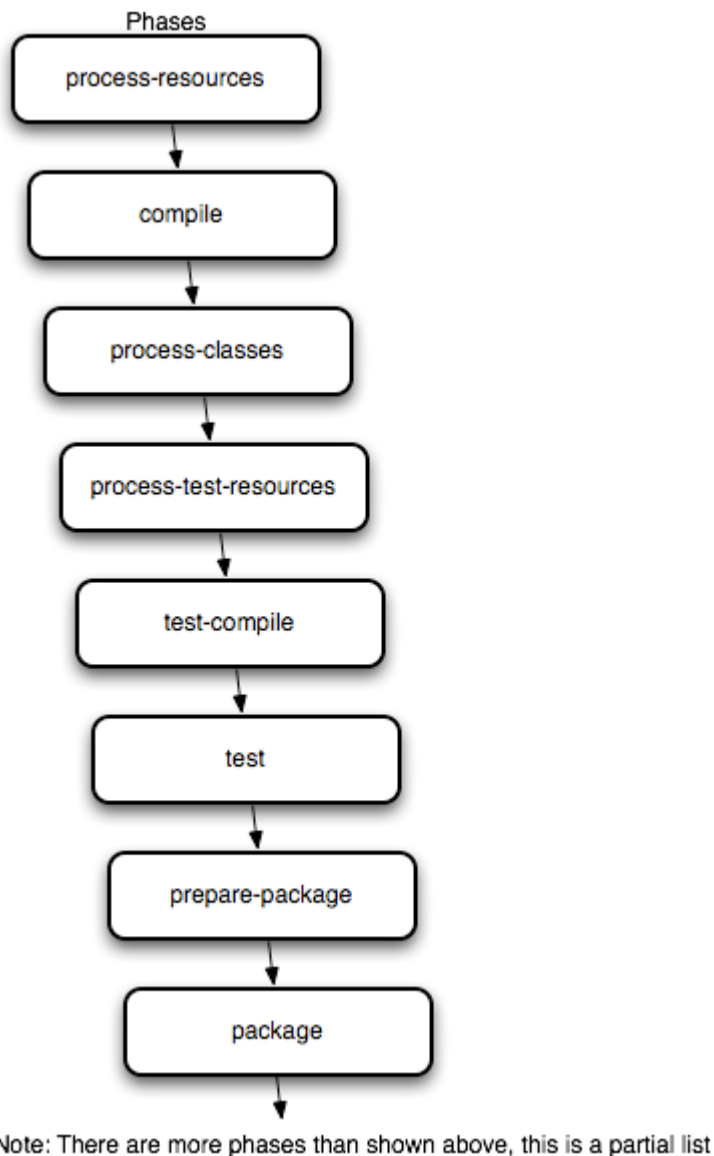


Figure 3.2. 一个生命周期是一些阶段的序列

插件目标可以附着在生命周期阶段上。随着 Maven 沿着生命周期的阶段移动，它会执行附着在特定阶段上的目标。每个阶段可能绑定了零个或者多个目标。在之前的小节里，当你运行 **mvn package**，你可能已经注意到了不止一个目标被执行了。检查运行 **mvn package** 之后的输出，会注意到那些被运行的各种目标。当这个简单例子到达 package 阶段的时候，它运行了 Jar 插件的 jar 目标。既然我们的简单的 quickstart 项目（默认）是 jar 包类型，jar:jar 目标被就绑定到了打包阶段。

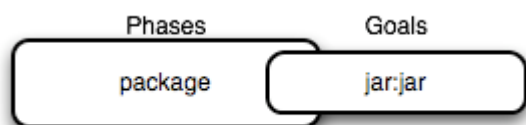


Figure 3.3. 一个目标绑定到一个阶段

我们知道，在包类型为 jar 的项目中，打包阶段将会创建一个 JAR 文件。但是，在它之前的目标做什么呢，像 `compiler:compile` 和 `surefire:test`？在 Maven 经过它生命周期中 `package` 之前的阶段的时候，这些目标被运行了；Maven 执行一个阶段的时候，它首先会有序的执行前面的所有阶段，到命令行指定的那个阶段为止。每个阶段对应了零个或者多个目标。我们没有进行任何插件配置或者定制，所以这个例子绑定了一组标准插件的目标到默认的生命周期。当 Maven 经过以 `package` 为结尾的默认生命周期的时候，下面的目标按顺序被执行：

resources:resources

Resources 插件的 `resources` 目标绑定到了 `resources` 阶段。这个目标复制 `src/main/resources` 下的所有资源和其它任何配置的资源目录，到输出目录。

compiler:compile

Compiler 插件的 `compile` 目标绑定到了 `compile` 阶段。这个目标编译 `src/main/java` 下的所有源代码和其他任何配置的资源目录，到输出目录。

resources:testResources

Resources 插件的 `testResources` 目标绑定到了 `test-resources` 阶段。这个目标复制 `src/test/resources` 下的所有资源和其它任何的配置的测试资源目录，到测试输出目录。

compiler:testCompile

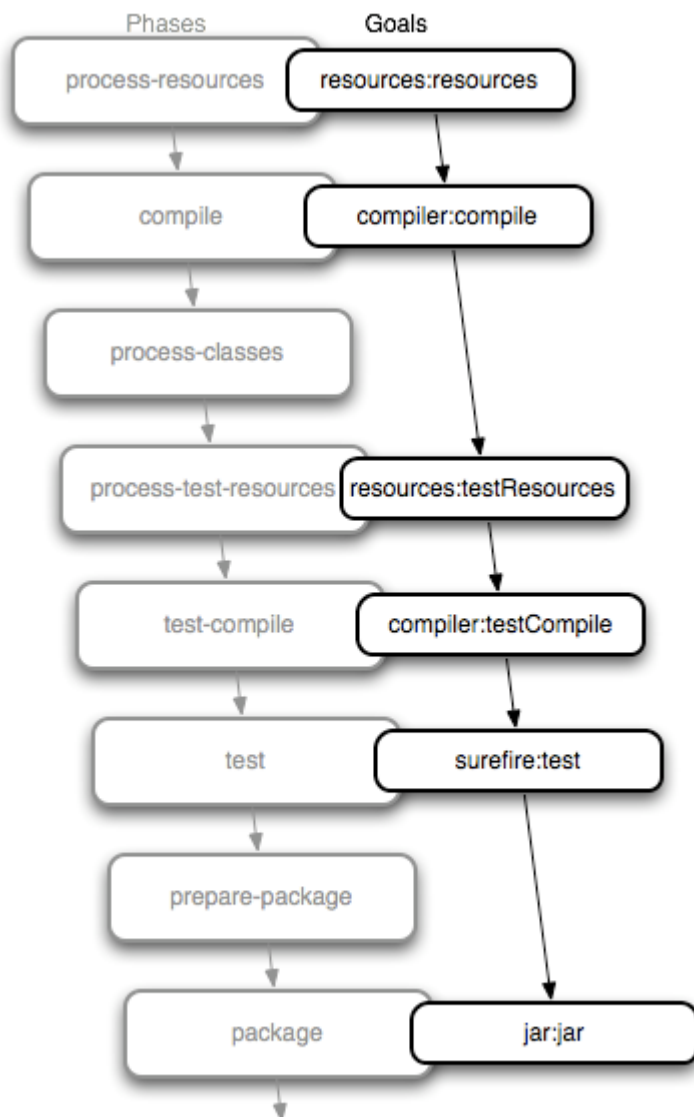
Compiler 插件的 `testCompile` 目标绑定到了 `test-compile` 阶段。这个目标编译 `src/test/java` 下的测试用例和其它任何的配置的测试资源目录，到测试输出目录。

surefire:test

Surefire 插件的 `test` 目标绑定到了 `test` 阶段。这个目标运行所有的测试并且创建那些捕捉详细测试结果的输出文件。默认情况下，如果有测试失败，这个目标会终止。

jar:jar

Jar 插件的 `jar` 目标绑定到了 `package` 阶段。这个目标把输出目录打包成 JAR 文件。



Note: There are more phases than shown above, this is a partial list

Figure 3.4. 被绑定的目标随着它们阶段的运行而运行

总结得来说，当我们运行 **mvn package**，Maven 运行到打包为止的所有阶段，在 Maven 沿着生命周期一步步向前的过程中，它运行绑定在每个阶段上的所有目标。你也可以像下面这样显式的指定一系列插件目标，以得到同样的结果：

```
mvn resources:resources \
    compiler:compile \
    resources:testResources \
    compiler:testCompile \
    surefire:test \
    jar:jar
```

运行 package 阶段能很好的跟踪一个特定的构建中包含的所有目标，它也允许每个项目使用 Maven 来遵循一组定义明确的标准。而这个生命周期能让开发人员从一个 Maven 项目跳到另外一个 Maven 项目，而不用知道太多每个项目构建的

细节。如果你能够构建一个 Maven 项目，那么你就能构建所有的 Maven 项目。

3.5.3. Maven 坐标 (Coordinates)



Archetype 插件通过名字为 pom.xml 的文件创建了一个项目。这就是项目对象模型 (POM)，一个项目的声明性描述。当 Maven 运行一个目标的时候，每个目标都会访问定义在项目 POM 里的信息。当 jar:jar 目标需要创建一个 JAR 文件的时候，它通过观察 POM 来找出这个 Jar 文件的名称。当 compiler:compile 任务编译 Java 源代码为字节码的时候，它通过观察 POM 来看是否有编译目标的参数。目标在 POM 的上下文中运行。目标是我们希望针对项目运行的动作，而项目是通过 POM 定义的。POM 为项目命名，提供了项目的一组唯一标识符（坐标），并且通过依赖 (dependencies)，父 (parents) 和先决条件 (prerequisite) 来定义和其它项目的关系。POM 也可以自定义插件行为，提供项目相关的社区和开发人员的信息。

Maven 坐标定义了一组标识，它们可以用来唯一标识一个项目，一个依赖，或者 Maven POM 里的一个插件。看一下下面的 POM。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mavenbook</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

coordinates

Figure 3.5. 一个 Maven 项目的坐标

我们加亮了这个项目的坐标：groupId, artifactId, version 和 packaging。这些组合的标识符拼成了一个项目的坐标^[2]。^[2]就像任何其它的坐标系统，一个 Maven 坐标是一个地址，即“空间”里的某个点：从一般到特殊。当一个项目通过依赖，插件或者父项目引用和另外一个项目关联的时候，Maven 通过坐标来精确定位一个项目。Maven 坐标通常用冒号来作为分隔符来书写，像这样的格式：groupId:artifactId:packaging:version。在上面的 pom.xml 中，它的坐标可以表示为 mavenbook:my-app:jar:1.0-SNAPSHOT。这个符号也适用于项目依赖，我们的项目依赖 JUnit 的 3.8.1 版本，它包含了一个对 junit:junit:jar:3.8.1 的依赖。

groupId

d 团体，公司，小组，组织，项目，或者其它团体。团体标识的约定是，它以创建这个项目的组织名称的逆向域名(reverse domain name)开头。来自 Sonatype 的项目有一个以 `com.sonatype` 开头的 `groupId`，而 Apache Software 的项目有以 `org.apache` 开头的 `groupId`。

artifactId

在 `groupId` 下的表示一个单独项目的唯一标识符。

version

一个项目的特定版本。发布的项目有一个固定的版本标识来指向该项目的某一个特定的版本。而正在开发中的项目可以用一个特殊的标识，这种标识给版本加上一个“SNAPSHOT”的标记。

项目的打包格式也是 **Maven** 坐标的重要组成部分，但是它不是项目唯一标识符的一个部分。一个项目的 `groupId:artifactId:version` 使之成为一个独一无二的项目；你不能同时有一个拥有同样的 `groupId`, `artifactId` 和 `version` 标识的项目。

packaging

项目的类型，默认是 `jar`，描述了项目打包后的输出。类型为 `jar` 的项目产生一个 **JAR** 文件，类型为 `war` 的项目产生一个 **web** 应用。

在其它“**Maven** 化”项目构成的巨大空间中，的这四个元素是定位和使用某个特定项目的关键因素。**Maven** 仓库(repositories)（公共的，私有的，和本地的）是通过这些标识符来组织的。当一个项目被安装到本地的 **Maven** 仓库，它立刻能被任何其它的项目所使用。而我们所需要做的只是，在其它项目用使用 **Maven** 的唯一坐标来加入对这个特定构件的依赖。

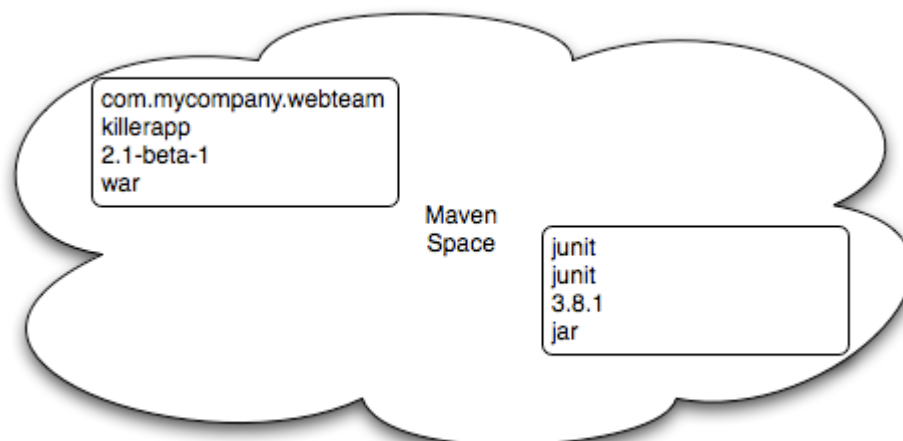


Figure 3.6. Maven 空间是项目的一个坐标系统

3.5.4. Maven 仓库(Repositories)



当你第一次运行 Maven 的时候，你会注意到 Maven 从一个远程的 Maven 仓库下载了许多文件。如果这个简单的项目是你第一次运行 Maven，那么当触发 `resources:resource` 目标的时候，它首先会做的事情是去下载最新版本的 Resources 插件。在 Maven 中，构件和插件是在它们被需要的时候从远程的仓库取来的。初始的 Maven 下载包的大小相当的小（1.8 兆），其中一个原因是事实上这个初始 Maven 不包括很多插件。它只包含了几近赤裸的最少值，而在需要的时候再从远程仓库去取。Maven 自带了一个用来下载 Maven 核心插件和依赖的远程仓库地址（<http://repo1.maven.org/maven2>）。

你常常会写这样一个项目，这个项目依赖于一些既不免费也不公开的包。在这种情况下，你需要要么在你组织的网络里安装一个定制的仓库，要么手动的安装这些依赖。默认的远程仓库可以被替换，或者增加一个你组织维护的自定义 Maven 仓库的引用。有许多现成的项目允许组织管理和维护公共 Maven 仓库的镜像。是什么让 Maven 仓库成为一个 Maven 仓库的呢？Maven 仓库是通过结构来定义的，一个 Maven 仓库是项目构件的一个集合，这些构件存储在一个目录结构下面，它们的格式能很容易的被 Maven 所理解。在一个 Maven 仓库中，所有的东西存储在一个与 Maven 项目坐标十分匹配的目录结构中。你可以打开浏览器，然后浏览中央 Maven 仓库 <http://repo1.maven.org/maven2/> 来看这样的结构。你会看到坐标为 `org.apache.commons:commons-email:1.1` 的构件能在目录 `/org/apache/commons/commons-email/1.1/` 下找到，文件名为 `commons-email-1.1.jar`。Maven 仓库的标准是按照下面的目录格式来存储构件，相对于仓库的根目录：

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

Maven 从远程仓库下载构件和插件到你本机上，存储在你的本地 Maven 仓库里。一旦 Maven 已经从远程仓库下载了一个构件，它将永远不需要再下载一次，因为 maven 会首先在本地仓库查找插件，然后才是其它地方。在 Windows XP 上，你的本地仓库很可能在 `C:\Documents and Settings\USERNAME\.m2\repository`，在 Windows Vista 上，会是 `C:\Users\USERNAME\.m2\repository`。在 Unix 系统上，你的本地仓库在 `~/.m2/repository`。当你创建像前一节创建的简单项目时，`install` 阶段执行一个目标，把你项目的构件安装到你的本地仓库。

在你的本地仓库，你应该可以看到我们的简单项目创建出来的构件。如果你运行 **mvn install** 命令，Maven 会把我们项目的构件安装到本地仓库。试一下。

```
$ mvn install
...
[INFO] [install:install]
[INFO] Installing .../simple-1.0-SNAPSHOT.jar to \
      ~/.m2/repository/org/sonatype/mavenbook/simple/1.0-SNAPSHOT/ \
      simple-1.0-SNAPSHOT.jar
...
```

就像你能从这个命令的输出看到的，Maven 把我们项目的 JAR 文件安装到了我们的本地 Maven 仓库。Maven 在本地项目中通过本地仓库来共享依赖。如果你开发了两个项目——项目 A 和项目 B——项目 B 依赖于项目 A 产生的构件。当构建项目 B 的时候，Maven 会从本地仓库取得项目 A 的构件。Maven 仓库既是一个从远程仓库下载的构件的缓存，也允许你的项目相互依赖。

3.5.5. Maven 依赖管理 (Dependency Management)



在本章的 simple 样例中，Maven 处理了 JUnit 依赖的坐标——`junit:junit:3.8.1`，指向本地 Maven 仓库中的 `/junit/junit/3.8.1/junit-3.8.1.jar`。这种基于 Maven 坐标的定位构件的能力能让我们在项目的 POM 中定义依赖。如果你检查 simple 项目的 `pom.xml` 文件，你会看到有一个文件中有一个段专门处理 dependencies，那里面包含了一个单独的依赖——JUnit。

一个复杂的项目将会包含很多依赖，也有可能包含依赖于其它构件的依赖。这是 Maven 最强大的特征之一，它支持了传递性依赖（transitive dependencies）。假如你的项目依赖于一个库，而这个库又依赖于五个或者十个其它的库（就像 Spring 或者 Hibernate 那样）。你不必找出所有这些依赖然后把它们写在你的 `pom.xml` 里，你只需要加上你直接依赖的那些库，Maven 会隐式的把这些库间接依赖的库也加入到你的项目中。Maven 也会处理这些依赖中的冲突，同时能让你自定义默认行为，或者排除一些特定的传递性依赖。

让我们看一下你运行前面的样例的时候那些下载到你本地仓库的依赖。看一下这个目录：`~/.m2/repository/junit/junit/3.8.1/`。如果你一直跟着本章的样例，那么这里会有文件 `junit-3.8.1.jar` 和 `junit-3.8.1.pom`，还有 Maven 用来验证已下载构件准确性的校验和文件。需要注意的是 Maven 不只是下载 JUnit 的 JAR 文件，它同时为这个 JUnit 依赖下载了一个 POM 文件。Maven 同时下载构件和 POM 文件的这种行为，对 Maven 支持传递性依赖来说非常重要。

当你把项目的构件安装到本地仓库时，你会发现在和 JAR 文件同一目录下，Maven 发布了一个稍微修改过的 `pom.xml` 的版本。存储 POM 文件在仓库里提供给其它项目了该项目的信息，其中最重要的就是它有哪些依赖。如果项目 B 依赖于项目 A，那么它也依赖于项目 A 的依赖。当 Maven 通过一组 Maven 坐标来处理依赖构件的时候，它也会获取 POM，通过依赖的 POM 来寻找传递性依赖。那些传递性依赖就会被添加到当前项目的依赖列表中。

在 Maven 中一个依赖不仅仅是一个 JAR。它是一个 POM 文件，这个 POM 可能也声明了对其它构件的依赖。这些依赖的依赖叫做传递性依赖，Maven 仓库不仅仅存储二进制文件，也存储了这些构建的元数据（metadata），才使传递性依赖成为可能。下图展现了一个传递性依赖的可能场景。

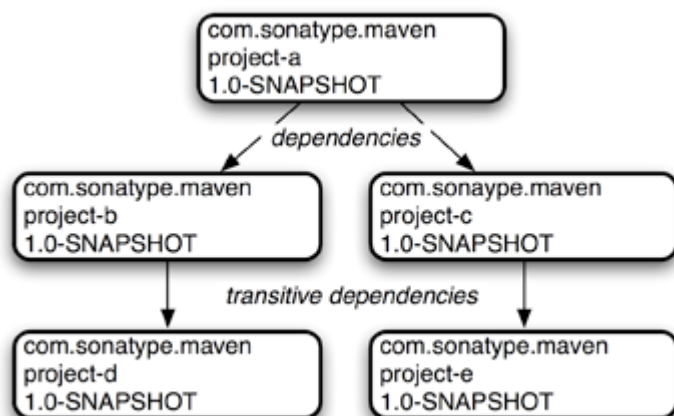


Figure 3.7. Maven 处理传递性依赖

在上图中，项目 A 依赖于项目 B 和 C，项目 B 依赖于项目 D，项目 C 依赖于项目 E，但是项目 A 所需要做的只是定义对 B 和 C 的依赖。当你的项目依赖于其它的项目，而这些项目又有一些小的依赖时(向 Hibernate, Apache Struts 或者 Spring Framework)，传递性依赖使之变得相当的方便。Maven 同时也提供了一种机制，能让你排除一些你不想要的传递性依赖。

Maven 也提供了不同的依赖范围(dependency scope)。Simple 项目的 pom.xml 包含了一个依赖——junit:junit:jar:3.8.1——范围是 test。当一个依赖的范围是 test 的时候，说明它在 Compiler 插件运行 compile 目标的时候是不可用的。它只有在运行 compiler:testCompile 和 surefire:test 目标的时候才会被加入到 classpath 中。

当为项目创建 JAR 文件的时候，它的依赖不会被捆绑在生成的构件中，他们只是用来编译。当用 Maven 来创建 WAR 或者 EAR，你可以配置 Maven 让它在生成的构件中捆绑依赖，你也可以配置 Maven，使用 provided 范围，让它排除 WAR 文件中特定的依赖。provided 范围告诉 Maven 一个依赖在编译的时候需要，但是它不应该被捆绑在构建的输出中。当你开发 web 应用的时候 provided 范围变得十分有用，你需要通过 Servlet API 来编译你的代码，但是你不希望 Servlet API 的 JAR 文件包含在你 web 应用的 WEB-INF/lib 目录中。

3.5.6. 站点生成和报告 (Site Generation and Reporting)



另外一个 Maven 的重要特征是，它能生成文档和报告。在 simple 项目的目录下，运行以下命令：

```
$ mvn site
```

这将会运行 site 生命周期阶段。它不像默认生命周期那样，管理代码生成，操作资源，编译，打包等等。Site 生命周期只关心处理在 src/site 目录下的 site 内容，还有生成报告。在这个命令运行过之后，你将会在 target/site 目录下看到一个项目 web 站点。载入 target/site/index.html 你会看到项目站点的基本外貌。它包含了一些报告，它们在左手边的导航目录的“项目报告”下面。它也包

含了项目相关的信息，依赖和相关开发人员信息，在“项目信息”下面。Simple 项目的 web 站点大部分是空的，因为 POM 只包含了比较少的信息，只有项目坐标，名称，URL 和一个 test 依赖。

在这个站点上，你会注意到一些默认的报告已经可以访问了，有一个报告详细描述了测试的结果。这个单元测试报告描述了项目中所有单元测试的成功和失败信息。另外一个报告生成了项目 API 的 JavaDoc。Maven 提供了很完整的可配置的报告，像 Clover 报告检查单元测试覆盖率，JXR 报告生成 HTML 源代码相互间引用，这在代码审查的时候非常有用，PMD 报告针对各种编码问题来分析源代码，JDepend 报告分析源代码中各个包之间的依赖。通过在 pom.xml 中配置那些报告被包含在构建中，站点报告就可以被定制了。

3.6. 小结



我们创建了一个 simple 项目，将其打包为一个 Jar，安装到了 Maven 仓库使之能被其它项目使用，最后生成了带有文档的站点。不写一行代码，不碰一个配置文件，我们就做到了这些。我们花了一些时间来研究 Maven 核心概念的定义。在下一章，我们将会自定义并修改我们项目的 pom.xml 文件，加入一些依赖，配置一些单元测试。

[1] "-D<name>=<value>"这种格式不是 Maven 定义的，它其实是 Java 用来设置系统属性的方式，可以通过“java -help”查看 Java 的解释。Maven 的 bin 目录下的脚本文件仅仅是把属性传入 Java 而已。

[2] 还有第五个，名为 classifier 的很少使用的坐标，将在本书后面介绍。现在你尽管可以忽略 classifiers。

Chapter 4. 定制一个 Maven 项目

[4.1. 介绍](#)

[4.1.1. 下载本章样例](#)

[4.2. 定义 Simple Weather 项目](#)

[4.2.1. Yahoo! Weather RSS](#)

[4.3. 创建 Simple Weather 项目](#)

[4.4. 定制项目信息](#)

[4.5. 添加新的依赖](#)

[4.6. Simple Weather 源码](#)

[4.7. 添加资源](#)

[4.8. 运行 Simple Weather 项目](#)

[4.8.1. Maven Exec 插件](#)

[4.8.2. 浏览你的项目依赖](#)

[4.9. 编写单元测试](#)

[4.10. 添加测试范围依赖](#)

[4.11. 添加单元测试资源](#)

[4.12. 执行单元测试](#)

[4.12.1. 忽略测试失败](#)

[4.12.2. 跳过单元测试](#)

[4.13. 构建一个打包好的命令行应用程序](#)

4.1. 介绍



本章在上一章所介绍信息的基础上进行开发。你将创建一个由 **Maven Archetype** 插件生成的项目，添加一些依赖和一些源代码，并且根据你的需要定制项目。本章最后，你将知道如何使用 **Maven** 开始创建真正的项目。

4.1.1. 下载本章样例



本章我们将开发一个和 **Yahoo! Weather web** 服务交互的实用程序。虽然没有样例源码你也应该能够理解这个开发过程，但还是推荐你下载本章样例源码以作为参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，

<http://www.sonatype.com/book/mvn-examples-1.0.zip> 或者

<http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>。解压存档文件至任意目录，然后到 `ch04/` 目录。在 `ch04/` 目录你会看到一个名为 `simple-weather` 的目录，它包含了本章开发出来的 **Maven** 项目。如果你想要在浏览器里看样例代码，访问 <http://www.sonatype.com/book/examples-1.0>，然后点击 `ch04/` 目录。

4.2. 定义 Simple Weather 项目



在定制本项目之前，让我们退后一步，讨论下这个 `simple weather` 项目。这个 `simple weather` 项目是什么？它是一个被设计成用来示范一些 **Maven** 特征的样例。它能代表一类你可能需要构建的应用程序。这个 `simple weather` 是一个基本的命令行驱动的应用程序，它接受邮政编码输入，然后从 **Yahoo! Weather RSS** 源获取数据，然后解析数据并把结果打印到标准输出。我们选择该项目是有许多因素的。首先，它很直观；用户通过命令行提供输入，程序读取邮政编码，对 **Yahoo! Weather** 提交请求，之后解析结果，格式化之后输入到屏幕。这个样例是个简单的 `main()` 函数加上一些相关支持的类；没有企业级框架需要介绍或解释，只有 **XML** 解析和一些日志语句。其次，它提供很好的机会来介绍一些有趣的类库，如 **Velocity**, **Dom4j** 和 **Log4j**。虽然本书集中于 **Maven**，但我们不会回避那些介绍有趣工具的机会。最后，这是一个能在一章内介绍，开发及部署的样例。

4.2.1. Yahoo! Weather RSS



在开始构建这个应用之前，你需要了解一下 **Yahoo! Weather RSS** 源。该服务是基于以下条款提供的：

“该数据源免费提供给个人和非营利性组织，作为个人或其它非商业用途。 我们要求你提供给 Yahoo! Weather 连接你数据源应用的权限。”

换句话说，如果你考虑集成该数据源到你的商业 web 站点上，请再仔细考虑考虑，该数据源可作为个人或其它非商业性用途。 本章我们提倡的使用是个人教育用途。 要了解更多的 Yahoo! Weather 服务条款，请参考 Yahoo! Weather API 文档：

<http://developer.yahoo.com/weather/>。

4.3. 创建 Simple Weather 项目



首先，让我们用 Maven Archetype 插件创建这个 simple weather 项目的基本轮廓。 运行下面的命令，创建新项目：

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch04 \
-DartifactId=simple-weather \
-DpackageName=org.sonatype.mavenbook \
-Dversion=1.0
[INFO] [archetype:create]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
checking for updates from central
[INFO] -----
[INFO] Using following parameters for creating Archetype: \
maven-archetype-quickstart:RELEASE
[INFO] -----
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch04
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: ~/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0
[INFO] Parameter: artifactId, Value: simple-weather
[INFO] *** End of debug info from resources from generated POM ***
[INFO] Archetype created indir: ~/examples/simple-weather
```

在 Maven Archetype 插件创建好了这个项目之后，进入到 simple-weather 目录，看一下 pom.xml。你会看到如下的 XML 文档：

Example 4.1. simple-weather 项目的初始 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch04</groupId>
  <artifactId>simple-weather</artifactId>
```

```

<packaging>jar</packaging>
<version>1.0</version>
<name>simple-weather2</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

请注意我们给 `archetype:create` 目标传入了 `version` 参数。它覆写了默认值 `1.0-SNAPSHOT`。本项目中，正如你从 `pom.xml` 的 `version` 元素看到的，我们正在开发 `simple-weather` 项目的 `1.0` 版本。

4.4. 定制项目信息



在开始编写代码之前，让我们先定制一些项目的信息。我们想要做的是添加一些关于项目许可证，组织以及项目相关开发人员的一些信息。这些都是你期望能在大部分项目中看到的标准信息。下面的文档展示了提供组织信息，许可证信息和开发人员信息的 XML。

Example 4.2. 为 `pom.xml` 添加组织，法律和开发人员信息

```

<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">
  ...

  <name>simple-weather</name>
  <url>http://www.sonatype.com</url>

  <licenses>
    <license>
      <name>Apache 2</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
      <comments>A business-friendly OSS license</comments>
    </license>
  </licenses>

```

```

<organization>
  <name>Sonatype</name>
  <url>http://www.sonatype.com</url>
</organization>

<developers>
  <developer>
    <id>jason</id>
    <name>Jason Van Zyl</name>
    <email>jason@maven.org</email>
    <url>http://www.sonatype.com</url>
    <organization>Sonatype</organization>
    <organizationUrl>http://www.sonatype.com</organizationUrl>
    <roles>
      <role>developer</role>
    </roles>
    <timezone>-6</timezone>
  </developer>
</developers>
...
</project>

```

[Example 4.2, “为 pom.xml 添加组织，法律和开发人员信息”](#) 中的省略号是为了使代码清单变得简短。当你在 pom.xml 中看到 project 元素的开始标签后面跟着“...”或者在 project 元素的结束标签前有“...”，这说明我们没有展示整个 pom.xml 文件。在上述情况中，licenses, organization 和 developers 元素是加在 dependencies 元素之前的。

4.5. 添加新的依赖



Simple weather 应用程序必须要完成以下三个任务：从 Yahoo! Weather 获取 XML 数据，解析 XML 数据，打印格式化的输出至标准输出。为了完成这三个任务，我们需要为项目的 pom.xml 引入一些新的依赖。为了解析来自 Yahoo! 的 XML 响应，我们将会使用 Dom4J 和 Jaxen，为了格式化这个命令程序的输出，我们将会使用 Velocity，我们还需要加入对 Log4j 的依赖，用来做日志。加入这些依赖之后，我们的 dependencies 元素就成了以下模样：

Example 4.3. 添加 Dom4J, Jaxen, Velocity 和 Log4J 作为依赖

```

<project>
  [...]
  <dependencies>
    <dependency>

```

```

    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.5</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
[...]
</project>

```

正如你从上面所看到的，我们在范围为 *test* 的 JUnit 依赖基础上又加了四个依赖元素。如果你把这些依赖添加到项目的 *pom.xml* 文件然后运行 ***mvn install***，你会看到 Maven 下载这些依赖及其它传递性依赖到你的本地 Maven 仓库。我们如何找这些依赖呢？我们都“知道”适当的 *groupId* 和 *artifactId* 的值吗？有些依赖（像 Log4J）被广泛使用，以至于每次你需要使用它们的时候你都会记得它们的 *groupId* 和 *artifactId*。而 Velocity, Dom4J 和 Jaxen 是通过一个十分有用的站点 <http://www.mvnrepository.com> 来定位的。该站点提供了针对 Maven 仓库的搜索接口，你可以用它来搜索依赖。你可以自己测试一下，载入 <http://www.mvnrepository.com> 然后搜索一些常用的类库，如 Hibernate 或者 Spring Framework。当你在这上面搜索构件时，它会显示一个 *artifactId* 和所有 Maven 中央仓库所知道的版本。点击某个特定的版本后，它会载入一个页面，这个页面就包括了你需要复制到你自己的项目 *pom.xml* 中的依赖元素。你经常会发现某个特定的类库拥有多个的

groupId, 这个时候你需要通过 mvnrepository.com 来帮助确定你到底需要怎样配置你的依赖。

4.6. Simple Weather 源码



Simple Weather 命令行应用程序包含五个 Java 类。

org.sonatype.mavenbook.weather.Main

这个类包含了一个静态的 `main()` 函数，即系统的入口。

org.sonatype.mavenbook.weather.Weather

Weather 类是个很简单的 Java Bean，它保存了天气报告的地点和其它一些关键元素，如气温和湿度。

org.sonatype.mavenbook.weather.YahooRetriever

YahooRetriever 连接到 Yahoo! Weather 并且返回来自数据源数据的 `InputStream`。

org.sonatype.mavenbook.weather.YahooParser

YahooParser 解析来自 Yahoo! Weather 的 XML，返回 Weather 对象。

org.sonatype.mavenbook.weather.WeatherFormatter

WeatherFormatter 接受 Weather 对象，创建 `VelocityContext`，根据 Velocity 模板生成结果。

这里我们不是想要详细阐述样例中的代码，但解释一下程序中使之运行的核心代码还是必要的。我们假设大部分读者已经下载的本书的源码，但也不会忘记那些按着书一步一步往下看的读者。本小节列出了 `simple-weather` 项目的类，这些类都放在同一个包下面，`org.sonatype.mavenbook.weather`。

让我们删掉由 `archetype:create` 生成 `App` 类和 `AppTest` 类，然后加入我们新的包。在 Maven 项目中，所有项目的源代码都存储在 `src/main/java` 目录。在新项目的基础目录下，运行下面的命令：

```
$ cd src/test/java/org/sonatype/mavenbook
$ rm AppTest.java
$ cd ../../../../..
$ cd src/main/java/org/sonatype/mavenbook
$ rm App.java
$ mkdir weather
$ cd weather
```

你已经创建了一个新的包 `org.sonatype.mavenbook.weather`。现在，我们需要把那些类放到这个目录下面。用你最喜欢的编辑器，创建一个新文件，名字为 `Weather.java`，内容如下：

Example 4.4. Simple Weather 的 Weather 模型对象

```
package org.sonatype.mavenbook.weather;
```

```

public class Weather {
    private String city;
    private String region;
    private String country;
    private String condition;
    private String temp;
    private String chill;
    private String humidity;

    public Weather() {}

    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }

    public String getRegion() { return region; }
    public void setRegion(String region) { this.region = region; }

    public String getCountry() { return country; }
    public void setCountry(String country) { this.country = country; }

    public String getCondition() { return condition; }
    public void setCondition(String condition) { this.condition = condition; }

    public String getTemp() { return temp; }
    public void setTemp(String temp) { this.temp = temp; }

    public String getChill() { return chill; }
    public void setChill(String chill) { this.chill = chill; }

    public String getHumidity() { return humidity; }
    public void setHumidity(String humidity) { this.humidity = humidity; }
}

```

Weather 类定义了一个简单的 **bean**，用来存储由 **Yahoo! Weather** 数据源解析出来的天气信息。天气数据源提供了丰富的信息，从日出日落时间，到风速和风向。为了让这个例子保持简单，Weather 模型对象只保存温度，湿度和当前天气情况的文字描述等信息。

在同一目录下，创建 Main.java 文件。Main 这个类有一个静态的 main() 函数——样例程序的入口。

Example 4.5. Simple Weather 的 Main 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.apache.log4j.PropertyConfigurator;

public class Main {

    public static void main(String[] args) throws Exception {
        // Configure Log4J
        PropertyConfigurator.configure(Main.class.getClassLoader()
            .getResource("log4j.properties"));

        // Read the Zip Code from the Command-line (if none supplied, use 60202)
        int zipcode = 60202;
        try {
            zipcode = Integer.parseInt(args[0]);
        } catch( Exception e ) {}

        // Start the program
        new Main(zipcode).start();
    }

    private int zip;

    public Main(int zip) {
        this.zip = zip;
    }

    public void start() throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        System.out.print( new WeatherFormatter().format( weather ) );
    }
}
```

上例中的 `main()` 函数通过获取 `classpath` 中的资源文件来配置 `Log4J`，之后它试图从命令行读取邮政编码。如果在读取邮政编码的时候抛出了异常，程序会设置默认邮政编码为 `60202`。一旦有了邮政编码，它初始化一个 `Main` 对象，调用该对象的 `start()` 方法。而 `start()` 方法会调用 `YahooRetriever` 来获取天气的 XML 数据。`YahooRetriever` 返回一个 `InputStream`，传给 `YahooParser`。`YahooParser` 解析 XML 数据并返回 `Weather` 对象。最后，`WeatherFormatter` 接受一个 `Weather` 对象并返回一个格式化的 `String`，打印到标准输出。

在相同目录下创建文件 `YahooRetriever.java`，内容如下：

Example 4.6. Simple Weather 的 YahooRetriever 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import org.apache.log4j.Logger;

public class YahooRetriever {

    private static Logger log = Logger.getLogger(YahooRetriever.class);

    public InputStream retrieve(int zipcode) throws Exception {
        log.info( "Retrieving Weather Data" );
        String url = "http://weather.yahooapis.com/forecastrss?p=" + zipcode;
        URLConnection conn = new URL(url).openConnection();
        return conn.getInputStream();
    }
}
```

这个简单的类打开一个连接到 `Yahoo! Weather API` 的 `URLConnection` 并返回一个 `InputStream`。我们还需要在该目录下创建文件 `YahooParser.java` 用以解析这个数据源。

Example 4.7. Simple Weather 的 YahooParser 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import org.apache.log4j.Logger;
import org.dom4j.Document;
import org.dom4j.DocumentFactory;
```

```

import org.dom4j.io.SAXReader;

public class YahooParser {

    private static Logger log = Logger.getLogger(YahooParser.class);

    public Weather parse(InputStream inputStream) throws Exception {
        Weather weather = new Weather();

        log.info( "Creating XML Reader" );
        SAXReader xmlReader = createXmlReader();
        Document doc = xmlReader.read( inputStream );

        log.info( "Parsing XML Response" );
        weather.setCity( doc.valueOf( "/rss/channel/y:location/@city" ) );
        weather.setRegion( doc.valueOf( "/rss/channel/y:location/@region" ) );
        weather.setCountry( doc.valueOf( "/rss/channel/y:location/@country" ) );

        weather.setCondition( doc.valueOf( "/rss/channel/item/y:condition/@text" ) );
        weather.setTemp( doc.valueOf( "/rss/channel/item/y:condition/@temp" ) );
        weather.setChill( doc.valueOf( "/rss/channel/y:wind/@chill" ) );

        weather.setHumidity( doc.valueOf( "/rss/channel/y:atmosphere/@humidity" ) );

        return weather;
    }

    private SAXReader createXmlReader() {
        Map<String,String> uris = new HashMap<String,String>();
        uris.put( "y", "http://xml.weather.yahoo.com/ns/rss/1.0" );

        DocumentFactory factory = new DocumentFactory();
        factory.setXPathNamespaceURIs( uris );

        SAXReader xmlReader = new SAXReader();
        xmlReader.setDocumentFactory( factory );
        return xmlReader;
    }
}

```

YahooParser 是本例中最复杂的类，我们不会深入 Dom4J 或者 Jaxen 的细节，但是这个类还是需要一些解释。YahooParser 的 parse() 方法接受一个 InputStrem 然后返回一个 Weather 对象。为了完成这一目标，它需要用 Dom4J 来解析 XML 文档。因为我们对 Yahoo! Weather XML 命名空间的元素感兴趣，我们需要用

`createXmlReader()` 方法创建一个包含命名空间信息的 `SAXReader` 。一旦我们创建了这个 `reader` 并且解析了文档，得到了返回的 `org.dom4j.Document` ，只需要简单的使用 `XPath` 表达式来获取需要的信息，而不是遍历所有的子元素。本例中 `Dom4J` 提供了 XML 解析功能，而 `Jaxen` 提供了 `XPath` 功能。

我们已经创建了 `Weather` 对象，我们需要格式化输出以供人阅读。在同一目录中创建一个名为 `WeatherFormatter.java` 的文件。

Example 4.8. Simple Weather 的 WeatherFormatter 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String format( Weather weather ) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader()
                                   .getResourceAsStream( "output.vm" ) );
        VelocityContext context = new VelocityContext();
        context.put( "weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }
}
```

`WeatherFormatter` 使用 `Velocity` 来呈现一个模板。`format()` 方法接受一个 `Weather bean` 然后返回格式化好的 `String` 。`format()` 方法做的第一件事是从 `classpath` 载入名字为 `output.vm` 的 `Velocity` 模板。然后我们创建一个 `VelocityContext` ，它需要一个 `Weather` 对象来填充。一个 `StringWriter` 被创建用来存放模板生成的结果数据。通过调用 `Velocity.evaluate()` ，给模板赋值，结果作为 `String` 返回。

在我们能够运行该样例程序之前，我们需要往 `classpath` 添加一些资源。

4.7. 添加资源



本项目依赖于两个 `classpath` 资源：Main 类通过 `classpath` 资源 `log4j.properties` 来配置 **Log4J**，`WeatherFormatter` 引用了一个在 `classpath` 中的名为 `output.vm` 的 **Velocity** 模板。这两个资源都需要在默认包中（或者 `classpath` 的根目录）。

为了添加这些资源，我们需要在项目的基础目录下创建一个新的目录——`src/main/resources`。由于任务 `archetype:create` 没有创建这个目录，我们需要通过在项目的基础目录下运行下面的命令来创建它：

```
$ cd src/main
$ mkdir resources
$ cd resources
```

在这个资源目录创建好之后，我们可以加入这两个资源。首先，往目录 `resources` 加入文件 `log4j.properties`。

Example 4.9. Simple Weather 的 Log4J 配置文件

```
# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=INFO, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r %-5p %c{1} %x - %m%n
```

这个 `log4j.properties` 文件简单配置了 **Log4J**，使其使用 `PatternLayout` 往标准输出打印所有日志信息。最后，我们需要创建 `output.vm`，它是这个命令行程序用来呈现输出的 **Velocity** 模板。在 `resources` 目录创建 `output.vm`。

Example 4.10. Simple Weather 的 Output Velocity 模板

```
*****
Current Weather Conditions for:
  ${weather.city}, ${weather.region}, ${weather.country}

Temperature: ${weather.temp}
Condition: ${weather.condition}
Humidity: ${weather.humidity}
Wind Chill: ${weather.chill}
*****
```

这个模板包含了许多对名为 `weather` 的变量的引用。这个 `weather` 变量是传给 `WeatherFormatter` 的那个 `Weather bean`，`${weather.temp}` 语法简化的表示获

取并显示 temp 这个 bean 属性的值。现在我们已经在了正确的地方有了我们项目的所有代码，我们可以使用 Maven 来运行这个样例。

4.8. 运行 Simple Weather 项目



使用来自 [Codehaus Mojo 项目](#) 的 Exec 插件，我们可以运行这个程序。在项目的基础目录下运行以下命令，以运行该程序的 Main 类。

```
$ mvn install
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main
...
[INFO] [exec:java]
0   INFO  YahooRetriever - Retrieving Weather Data
134 INFO  YahooParser - Creating XML Reader
333 INFO  YahooParser - Parsing XML Response
420 INFO  WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
    Evanston, IL, US

Temperature: 45
    Condition: Cloudy
    Humidity: 76
    Wind Chill: 38
*****
...
```

我们没有为 Main 类提供命令行参数，因此程序按照默认的邮编执行——60202。正如你能看到的，我们已经成功的运行了 Simple Weather 命令行工具，从 Yahoo! Weather 获取了一些数据，解析了结果，并且通过 Velocity 格式化了结果数据。我们仅仅写了项目的源代码，往 pom.xml 添加了一些最少的配置。注意我们这里没有引入“构建过程”。我们不需要定义如何或哪里让 Java 编译器编译我们的源代码，我们不需要指导构建系统在运行样例程序的时候如何定位二进制文件，我们所需要做的是包含一些依赖，用来定位合适的 Maven 坐标。

4.8.1. Maven Exec 插件



Exec 插件允许你运行 Java 类和其它脚本。它不是 Maven 核心插件，但它可以从 [Codehaus](#) 的 [Mojo](#) 项目得到。想要查看 Exec 插件的完整描述，运行：

```
$ mvn help:describe -Dplugin=exec -Dfull
```

这会列出所有 Maven Exec 插件可用的目标。Help 插件同时也会列出 Exec 插件的有效参数，如果你想要定制 Exec 插件的行为，传入命令行参数，你应该使用 help:describe 提供的文档作为指南。虽然 Exec 插件很有用，在开发过程中用来

运行测试之外，你不应该依赖它来运行你的应用程序。想要更健壮的解决方案，使用 **Maven Assembly** 插件，它在 [Section 4.13, “构建一个打包好的命令行应用程序”](#) 中被描述。

4.8.2. 浏览你的项目依赖



Exec 插件让我们能够在不往 **classpath** 载入适当的依赖的情况下，运行这个程序。在任何其它的构建系统能够中，我们必须复制所有程序依赖到类似于 **lib/** 的目录，这个目录包含一个 **JAR** 文件的集合。那样，我们就必须写一个简单的脚本，在 **classpath** 中包含我们程序的二进制代码和我们的依赖。只有那样我们才能运行 **java org.sonatype.mavenbook.weather.Main**。**Exec** 能做这样的工作是因为 **Maven** 已经知道如何创建和管理你的 **classpath** 和你的依赖。

了解你项目的 **classpath** 包含了哪些依赖是很方便也很有用的。这个项目不仅包含了一些类库如 **Dom4J**，**Log4J**，**Jaxen**，和 **Velocity**，它同时也引入了一些传递性依赖。如果你需要找出 **classpath** 中有什么，你可以使用 **Maven Dependency** 插件来打印出已解决依赖的列表。要打印出 **Simple Weather** 项目的这个列表，运行 **dependency:resolve** 目标。

```
$ mvn dependency:resolve
...
[INFO] [dependency:resolve]
[INFO]
[INFO] The following files have been resolved:
[INFO]     com.ibm.icu:icu4j:jar:2.6.1 (scope = compile)
[INFO]     commons-collections:commons-collections:jar:3.1 (scope = compile)
[INFO]     commons-lang:commons-lang:jar:2.1 (scope = compile)
[INFO]     dom4j:dom4j:jar:1.6.1 (scope = compile)
[INFO]     jaxen:jaxen:jar:1.1.1 (scope = compile)
[INFO]     jdom:jdom:jar:1.0 (scope = compile)
[INFO]     junit:junit:jar:3.8.1 (scope = test)
[INFO]     log4j:log4j:jar:1.2.14 (scope = compile)
[INFO]     oro:oro:jar:2.0.8 (scope = compile)
[INFO]     velocity:velocity:jar:1.5 (scope = compile)
[INFO]     xalan:xalan:jar:2.6.0 (scope = compile)
[INFO]     xerces:xercesImpl:jar:2.6.2 (scope = compile)
[INFO]     xerces:xmlParserAPIs:jar:2.6.2 (scope = compile)
[INFO]     xml-apis:xml-apis:jar:1.0.b2 (scope = compile)
[INFO]     xom:xom:jar:1.0 (scope = compile)
```

正如你能看到的，我们项目拥有一个很大的依赖集合。虽然我们只是为四个类库引入了直接的依赖，看来我们实际共引入了 **15** 个依赖。**Dom4J** 依赖于 **Xerces** 和 **XML** 解析器 **API**，**Jaxen** 依赖于 **Xalan**，后者也就在 **classpath** 中可用了。

Dependency 插件将会打印出最终的你项目编译所基于的所有依赖的组合。如果你想知道你项目的整个依赖树，你可以运行 **dependency:tree** 目标。

```
$ mvn dependency:tree
```

```
...
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.ch04:simple-weather:jar:1.0
[INFO] +- log4j:log4j:jar:1.2.14:compile
[INFO] +- dom4j:dom4j:jar:1.6.1:compile
[INFO] | \- xml-apis:xml-apis:jar:1.0.b2:compile
[INFO] +- jaxen:jaxen:jar:1.1.1:compile
[INFO] | +- jdom:jdom:jar:1.0:compile
[INFO] | +- xerces:xercesImpl:jar:2.6.2:compile
[INFO] | \- xom:xom:jar:1.0:compile
[INFO] |     +- xerces:xmlParserAPIs:jar:2.6.2:compile
[INFO] |     +- xalan:xalan:jar:2.6.0:compile
[INFO] |     \- com.ibm.icu:icu4j:jar:2.6.1:compile
[INFO] +- velocity:velocity:jar:1.5:compile
[INFO] | +- commons-collections:commons-collections:jar:3.1:compile
[INFO] | +- commons-lang:commons-lang:jar:2.1:compile
[INFO] | \- oro:oro:jar:2.0.8:compile
[INFO] +- org.apache.commons:commons-io:jar:1.3.2:test
[INFO] \- junit:junit:jar:3.8.1:test
...
```

如果你还不满足，或者想要查看完整的依赖踪迹，包含那些因为冲突或者其它原因而被拒绝引入的构件，打开 **Maven** 的调试标记运行：

```
$ mvn install -X
```

```
...
[DEBUG] org.sonatype.mavenbook.ch04:simple-weather:jar:1.0 (selected for null)
[DEBUG]   log4j:log4j:jar:1.2.14:compile (selected for compile)
[DEBUG]   dom4j:dom4j:jar:1.6.1:compile (selected for compile)
[DEBUG]     xml-apis:xml-apis:jar:1.0.b2:compile (selected for compile)
[DEBUG]   jaxen:jaxen:jar:1.1.1:compile (selected for compile)
[DEBUG]     jaxen:jaxen:jar:1.1-beta-6:compile (removed - causes a cycle in the graph)
[DEBUG]     jaxen:jaxen:jar:1.0-FCS:compile (removed - causes a cycle in the graph)
[DEBUG]     jdom:jdom:jar:1.0:compile (selected for compile)
[DEBUG]     xml-apis:xml-apis:jar:1.3.02:compile (removed - nearer found: 1.0.b2)
[DEBUG]     xerces:xercesImpl:jar:2.6.2:compile (selected for compile)
[DEBUG]     xom:xom:jar:1.0:compile (selected for compile)
[DEBUG]       xerces:xmlParserAPIs:jar:2.6.2:compile (selected for compile)
[DEBUG]       xalan:xalan:jar:2.6.0:compile (selected for compile)
```

```
[DEBUG]      xml-apis:xml-apis:1.0.b2.
[DEBUG]      com.ibm.icu:icu4j:jar:2.6.1:compile (selected for compile)
[DEBUG]      velocity:velocity:jar:1.5:compile (selected for compile)
[DEBUG]      commons-collections:commons-collections:jar:3.1:compile
(selected for compile)
[DEBUG]      commons-lang:commons-lang:jar:2.1:compile (selected for compile)
[DEBUG]      oro:oro:jar:2.0.8:compile (selected for compile)
[DEBUG]      junit:junit:jar:3.8.1:test (selected for test)
```

从调试输出我们看到一些依赖管理系统工作的内部信息。你在这里看到的是项目的依赖树。Maven 正打印出你项目的所有的依赖，以及这些依赖的依赖（还有依赖的依赖的依赖）的完整的 Maven 坐标。你能看到 simple-weather 依赖于 jaxen，jaxen 依赖于 xom，xom 接着依赖于 icu4j。从该输出你能看到 Maven 正在创建一个依赖图，排除重复，解决不同版本之间的冲突。如果你的依赖有问题，通常在 dependency:tree 所生成的列表基础上更深入一点会有帮助；开启调试输出允许你看到 Maven 工作时的依赖机制。

4.9. 编写单元测试



Maven 内建了对单元测试的支持，测试是 Maven 默认生命周期的一部分。让我们给 Simple Weather 项目添加一些单元测试。首先，在 src/test/java 下面创建包 org.sonatype.mavenbook.weather。

```
$ cd src/test/java
$ cd org/sonatype/mavenbook
$ mkdir -p weather/yahoo
$ cd weather/yahoo
```

目前，我们将会创建两个单元测试。第一个单元测试会测试 YahooParser，第二个会测试 WeatherFormatter。在 weather 包中，创建一个带有一以下内容的文件，名称为 YahooParserTest.java。

Example 4.11. Simple Weather 的 YahooParserTest 单元测试

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import junit.framework.TestCase;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.YahooParser;

public class YahooParserTest extends TestCase {

    public YahooParserTest(String name) {
```

```

    super(name);
}

public void testParser() throws Exception {
    InputStream nyData =
        getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
    Weather weather = new YahooParser().parse( nyData );
    assertEquals( "New York", weather.getCity() );
    assertEquals( "NY", weather.getRegion() );
    assertEquals( "US", weather.getCountry() );
    assertEquals( "39", weather.getTemp() );
    assertEquals( "Fair", weather.getCondition() );
    assertEquals( "39", weather.getChill() );
    assertEquals( "67", weather.getHumidity() );
}
}

```

YahooParserTest 继承了 JUnit 定义的 TestCase 类。它遵循了 JUnit 测试的惯例模式：一个构造函数接受一个单独的 String 参数并调用父类的构造函数，还有一系列以“test”开头的公有方法，做为单元测试被调用。我们定义了一个单独的测试方法，testParser，通过解析一个值已知的 XML 文档来测试 YahooParser。测试 XML 文档命名为 ny-weather.xml，从 classpath 载入。我们将在 [Section 4.11, “添加单元测试资源”](#) 添加测试资源。在我们这个 Maven 项目的目录布局中，文件 ny-weather.xml 可以从包含测试资源的目录——

`${basedir}/src/test/resources` 中找到，路径为 `org/sonatype/mavenbook/weather/yahoo/ny-weather.xml`。该文件作为一个 InputStream 被读入，传给 YahooParser 的 parse() 方法。parse() 方法返回一个 Weather 对象，该对象通过一系列由 TestCase 定义的 assertEquals() 调用而被测试。

在同一目录下创建一个名为 WeatherFormatterTest.java 的文件。

Example 4.12. Simple Weather 的 WeatherFormatterTest 单元测试

```

package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import org.apache.commons.io.IOUtils;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.WeatherFormatter;
import org.sonatype.mavenbook.weather.YahooParser;

import junit.framework.TestCase;

```

```

public class WeatherFormatterTest extends TestCase {

    public WeatherFormatterTest(String name) {
        super(name);
    }

    public void testFormat() throws Exception {
        InputStream nyData =
            getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
        Weather weather = new YahooParser().parse( nyData );
        String formattedResult = new WeatherFormatter().format( weather );
        InputStream expected =

        getClass().getClassLoader().getResourceAsStream("format-expected.dat");
        assertEquals( IOUtils.toString( expected ).trim(),
            formattedResult.trim() );
    }
}

```

该项目中的第二个单元测试测试 WeatherFormatter。和 YahooParserTest 一样，WeatherFormatter 同样也继承 JUnit 的 TestCase 类。这个单独的测试通过单元测试的 classpath 从 \${basedir}/src/test/resources 的 org/sonatype/mavenbook/weather/yahoo 目录读取同样的测试资源文件。我们将会在 [Section 4.11, “添加单元测试资源”](#) 添加测试资源。WeatherFormatterTest 首先调用 YahooParser 解析出 Weather 对象，然后用 WeatherFormatter 格式化这个对象。我们的期望输出被存储在一个名为 format-expected.dat 的文件中，该文件存放在和 ny-weather.xml 同样的目录中。要比较测试输出和期望输出，我们将期望输出作为 InputStream 读入，然后使用 Commons IO 的 IOUtils 类来把文件转化为 String。然后使用 assertEquals() 比较这个 String 和测试输出。

4.10. 添加测试范围依赖



在类 WeatherFormatterTest 中我们用了一个来自于 Apache Commons IO 的工具——IOUtils 类。IOUtils 提供了许多很有帮助的静态方法，能帮助让很多工作摆脱繁琐的 I/O 操作。在这个单元测试中我们使用了 IOUtils.toString() 来复制 classpath 中资源 format.expected.dat 中的数据至 String。不用 Commons IO 我们也能完成这件事情，但是那需要额外的六七行代码来处理像 InputStreamReader 和 StringWriter 这样的对象。我们使用 Commons IO 的主要原因是，能有理由添加对 Commons IO 的测试范围依赖。

测试范围依赖是一个只在测试编译和测试运行时在 classpath 中有效的依赖。如果你的项目是以 war 或者 ear 形式打包的，测试范围依赖就不会被包含在项目的打包输

出中。要添加一个测试范围依赖，在你项目的 `dependencies` 小节中添加如下 `dependency` 元素。

Example 4.13. 添加一个测试范围依赖

```
<project>
...
<dependencies>
...
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-io</artifactId>
  <version>1.3.2</version>
  <scope>test</scope>
</dependency>
...
</dependencies>
</project>
```

当你往 `pom.xml` 中添加了 this 依赖以后，运行 `mvn dependency:resolve` 你会看到 `commons-io` 出现在在依赖列表中，范围是 `test`。在我们可以运行该项目的单元测试之前，我们还需要做一件事情。那就是创建单元测试依赖的 `classpath` 资源。测试范围依赖将在 9.4.1 节“依赖范围”中详细解释。

4.11. 添加单元测试资源



一个单元测试需要访问针对测试的一组资源。通常你需要在测试 `classpath` 中存储一些包含期望结果的文件，以及包含模拟输入的文件。在本项目中，我们为 `YahooParserTest` 准备了一个名为 `ny-weather.xml` 的测试 XML 文档，还有一个名为 `format-expected.dat` 的文件，包含了 `WeatherFormatter` 的期望输出。

要添加测试资源，你需要创建目录 `src/test/resources`。这是 `Maven` 寻找测试资源的默认目录。在你的项目基础目录下运行下面的命令以创建该目录。

```
$ cd src/test
$ mkdir resources
$ cd resources
```

当你创建好这个资源目录之后，在资源目录下创建一个名为 `format-expected.dat` 的文件。

Example 4.14. Simple Weather 的 WeatherFormatterTest 期望输出

```
*****
Current Weather Conditions for:
  New York, NY, US
```

Temperature: 39
Condition: Fair
Humidity: 67
Wind Chill: 39

这个文件应该看起来很熟悉了，它和你用 Maven Exec 插件运行 Simple Weather 项目得到的输出是一样的。你需要在资源目录添加的第二个文件是 ny-weather.xml 。

Example 4.15. Simple Weather 的 YahooParserTest XML 输入

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
    xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo! Weather - New York, NY</title>
    <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY</link>
    <description>Yahoo! Weather for New York, NY</description>
    <language>en-us</language>
    <lastBuildDate>Sat, 10 Nov 2007 8:51 pm EDT</lastBuildDate>

    <ttl>60</ttl>
    <yweather:location city="New York" region="NY" country="US" />
    <yweather:units temperature="F" distance="mi" pressure="in" speed="mph" />
    <yweather:wind chill="39" direction="0" speed="0" />
    <yweather:atmosphere humidity="67" visibility="1609" pressure="30.18"
        rising="1" />
    <yweather:astronomy sunrise="6:36 am" sunset="4:43 pm" />
    <image>
    <title>Yahoo! Weather</title>

    <width>142</width>
    <height>18</height>
    <link>http://weather.yahoo.com/</link>
    <url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
    </image>
    <item>
    <title>Conditions for New York, NY at 8:51 pm EDT</title>

    <geo:lat>40.67</geo:lat>
    <geo:long>-73.94</geo:long>
    <link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/</link>
    <pubDate>Sat, 10 Nov 2007 8:51 pm EDT</pubDate>
    <yweather:condition text="Fair" code="33" temp="39"
        date="Sat, 10 Nov 2007 8:51 pm EDT" />
```

```

<description><![CDATA[
<br />
<b>Current Conditions:</b><br />
Fair, 39 F<br /><br />
<b>Forecast:</b><br />
  Sat - Partly Cloudy. High: 45 Low: 32<br />
  Sun - Sunny. High: 50 Low: 38<br />
<br />
]]></description>
<yweather:forecast day="Sat" date="10 Nov 2007" low="32" high="45"
  text="Partly Cloudy" code="29" />

<yweather:forecast day="Sun" date="11 Nov 2007" low="38" high="50"
  text="Sunny" code="32" />
  <guid isPermaLink="false">10002_2007_11_10_20_51_EDT</guid>
</item>
</channel>
</rss>

```

该文件包含了一个给 YahooParserTest 用的 XML 文档。有了这个文件，我们不用从 Yahoo! Weather 获取 XML 响应就能测试 YahooParser 了。

4.12. 执行单元测试



既然你的项目已经有单元测试了，那么让它们运行起来吧。你不必为了运行单元测试做什么特殊的事情，test 阶段是 Maven 生命周期中常规的一部分。当你运行 **mvn package** 或者 **mvn install** 的时候你也运行了测试。如果你想要运行到 test 阶段为止的所有生命周期阶段，运行 **mvn test**。

```

$ mvn test
...
[INFO] [surefire:test]
[INFO] Surefire report directory:
~/examples/simple-weather/target/surefire-reports

-----

T E S T S

-----

Running org.sonatype.mavenbook.weather.yahoo.WeatherFormatterTest
0   INFO  YahooParser - Creating XML Reader
177 INFO  YahooParser - Parsing XML Response
239 INFO  WeatherFormatter - Formatting Weather Data
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.547 sec

```

```
Running org.sonatype.mavenbook.weather.yahoo.YahooParserTest
475 INFO YahooParser - Creating XML Reader
483 INFO YahooParser - Parsing XML Response
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

从命令行运行 **mvn test** 使 **Maven** 执行到 test 阶段为止的所有生命周期阶段。**Maven Surefire** 插件有一个 test 目标，该目标被绑定在了 test 阶段。test 目标执行项目中所有能在 src/test/java 找到的并且文件名与 ****/*Test*.java**，****/*Test.java** 和 ****/*TestCase.java** 匹配的所有单元测试。在本例中，你能看到 **Surefire** 插件的 test 目标执行了 **WeatherFormatterTest** 和 **YahooParserTest**。在 **Maven Surefire** 插件执行 **JUnit** 测试的时候，它同时也在 `${basedir}/target/surefire-reports` 目录下生成 XML 和常规文本报告。如果你的测试失败了，你可以去查看这个目录，里面有单元测试生成的异常堆栈信息和错误信息。

4.12.1. 忽略测试失败



通常，你会开发一个带有很多失败单元测试的系统。如果你正在实践测试驱动开发 (TDD)，你可能会使用测试失败来衡量你离项目完成有多远。如果你有失败的单元测试，但你仍然希望产生构建输出，你就必须告诉 **Maven** 让它忽略测试失败。当 **Maven** 遇到一个测试失败，它默认的行为是停止当前的构建。如果你希望继续构建项目，即使 **Surefire** 插件遇到了失败的单元测试，你就需要设置 **Surefire** 的 `testFailureIgnore` 这个配置属性为 `true`。

Example 4.16. 忽略单元测试失败

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

该插件文档

(<http://maven.apache.org/plugins/maven-surefire-plugin/test-mojo.html>)

说明，这个参数声明为一个表达式：

Example 4.17. 插件参数表达式

`testFailureIgnore` Set this to true to ignore a failure during testing. Its use is NOT RECOMMENDED, but quite convenient on occasion.

- * Type: boolean
- * Required: No
- * Expression: `${maven.test.failure.ignore}`

这个表达式可以从命令行通过 `-D` 参数设置。

```
$ mvn test -Dmaven.test.failure.ignore=true
```

4.12.2. 跳过单元测试



你可能想要配置 **Maven** 使其完全跳过单元测试。可能你有一个很大的系统，单元测试需要花好多分钟来完成，而你不想在生成最终输出前等单元测试完成。你可能正工作在一个遗留系统上面，这个系统有一系列的失败的单元测试，你可能仅仅想要生成一个 **JAR** 而不是去修复所有的单元测试。**Maven** 提供了跳过单元测试的能力，只需要使用 **Surefire** 插件的 `skip` 参数。在命令行，只要简单的给任何目标添加 `maven.test.skip` 属性就能跳过测试：

```
$ mvn install -Dmaven.test.skip=true
...
[INFO] [compiler:testCompile]
[INFO] Not compiling test sources
[INFO] [surefire:test]
[INFO] Tests are skipped.
...
```

当 **Surefire** 插件到达 `test` 目标的时候，如果 `maven.test.skip` 设置为 `true`，它就会跳过单元测试。另一种配置 **Maven** 跳过单元测试的方法是给你项目的 `pom.xml` 添加这个配置。你需要为你的 `build` 添加 `plugin` 元素。

Example 4.18. 跳过单元测试

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
```

```
    <configuration>
      <skip>true</skip>
    </configuration>
  </plugin>
</plugins>
</build>
[...]
```

4.13. 构建一个打包好的命令行应用程序



在 [Section 4.12, “执行单元测试”](#)，我们使用 **Maven Exec** 插件运行了 **Simple Weather** 应用程序。虽然 **Maven Exec** 能运行程序并且产生输出，你不能就把 **Maven** 当成是你程序运行的容器。如果你把这个命令程序分发给其他人，你大概就需要分发一个 **JAR** 或者一个 **ZIP** 存档文件或者 **TAR** 压缩过的 **GZIP** 文件。下面的小节介绍了使用 **Maven Assembly** 插件的预定义装配描述符生成一个可分发的 **JAR** 文件的过程，该文件包含了项目的二进制文件和所有的依赖。

Maven Assembly 插件是一个用来创建你应用程序特有分发包的插件。你可以使用 **Maven Assembly** 插件以你希望的任何形式来装配输出，只需定义一个自定义的装配描述符。后面的章节我们会说明如何创建一个自定义装配描述符，为 **Simple Weather** 应用程序生成一个更复杂的存档文件。本章我们将会使用预定义的 **jar-with-dependencies** 格式。要配置 **Maven Assembly** 插件，我们需要在 **pom.xml** 中的 **build** 配置中添加如下的 **plugin** 配置。

Example 4.19. 配置 **Maven** 装配描述符

```
<project>
[...]
```

```
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
[...]
```

添加好这些配置以后，你可以通过运行 **mvn assembly:assembly** 来构建这个装配。

```
$ mvn install assembly:assembly
...
[INFO] [jar:jar]
[INFO] Building jar: ~/examples/simple-weather/target/simple-weather-1.0.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: \
      .m2/repository/dom4j/dom4j/1.6.1/dom4j-1.6.1.jar into \
      /tmp/archived-file-set.1437961776.tmp
[INFO]
Expanding: .m2/repository/commons-lang/commons-lang/2.1/commons-lang-2.1.jar
\
      into /tmp/archived-file-set.305257225.tmp
... (Maven Expands all dependencies into a temporary directory) ...
[INFO] Building jar: \
~/examples/simple-weather/target/simple-weather-1.0-jar-with-dependencies.jar
```

在 `target/simple-weather-1.0-jar-with-dependencies.jar` 装配好之后，我们可以在命令行重新运行 `Main` 类。在你项目的基础目录下运行以下命令：

```
$ cd target
$ java -cp simple-weather-1.0-jar-with-dependencies.jar
org.sonatype.mavenbook.weather.Main 10002
0   INFO  YahooRetriever - Retrieving Weather Data
221 INFO  YahooParser - Creating XML Reader
399 INFO  YahooParser - Parsing XML Response
474 INFO  WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
New York, NY, US

Temperature: 44
Condition: Fair
Humidity: 40
Wind Chill: 40
*****
```

`jar-with-dependencies` 格式创建一个包含所有 `simple-weather` 项目的二进制代码以及所有依赖解压出来的二进制代码的 `JAR` 文件。这个略微非常规的格式产生了一个 **9 MiB** 大小的 `JAR` 文件，包含了大概 **5290** 个类。但是它确实给那些使用

Maven 开发的应用程序提供了一个易于分发的格式。本书的后面，我们会说明如何创建一个自定义的装配描述符来生成一个更标准的分发包。

Chapter 5. 一个简单的 Web 应用

[5.1. 介绍](#)

[5.1.1. 下载本章样例](#)

[5.2. 定义这个简单的 Web 应用](#)

[5.3. 创建这个简单的 Web 应用](#)

[5.4. 配置 Jetty 插件](#)

[5.5. 添加一个简单的 Servlet](#)

[5.6. 添加 J2EE 依赖](#)

[5.7. 小结](#)

5.1. 介绍



本章我们使用 Maven Archetype 插件创建一个简单的 web 应用程序。我们将会在一个名为 Jetty 的 Servlet 容器中运行这个 web 应用程序，同时添加一些依赖，编写一个简单的 Servlet，并且生成一个 WAR 文件。本章最后，你将能够开始使用 Maven 来提高你开发 web 应用程序的速度。

5.1.1. 下载本章样例



本章的样例是通过 Maven Archetype 插件生成的。虽然没有样例源码你也应该能够理解这个开发过程，但还是推荐你下载样例源码作为参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，

<http://www.sonatype.com/book/mvn-examples-1.0.zip> 或者

<http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>。解开存档文件至任意目录，然后到 ch05/ 目录。在 ch05/ 目录你会看到一个名为 simple-webapp/ 的目录，它包含了本章开发出来的 Maven 项目。如果你想要在浏览器里看样例代码，访问 <http://www.sonatype.com/book/examples-1.0>，然后点击 ch05/ 目录。

5.2. 定义这个简单的 Web 应用



我们已经有意地使本章关注于一个简单 Web 应用 (POWA) —— 一个 servlet 和一个 JSP 页面。在接下来的二十多页中，我们不会告诉你如何开发你的 Struts 2, Tapestry, Wicket, JSF, 或者 Waffle 应用，我们也不会涉及到集成诸如 Plexus, Guice 或者 Spring Framework 之类的 IoC 容器。本章的目标是展示给你看开发 web 应用的时候 Maven 提供的基本设备，不多，也不少。本书的后面，我们将会看一下开发两个 web 应用，一个使用了 Hibernate, Velocity 和 Spring Framework，另外一个使用了 Plexus。

5.3. 创建这个简单的 Web 应用



创建你的 web 应用程序项目，运行 **mvn archetype:create**，加上参数 `artifactId` 和 `groupId`。指定 `archetypeArtifactId` 为 `maven-archetype-webapp`。如此便创建了恰到好处的目录结构和 Maven POM。

```
~/examples$ mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch05 \
-DartifactId=simple-webapp \
-DpackageName=org.sonatype.mavenbook \
-DarchetypeArtifactId=maven-archetype-webapp
[INFO] [archetype:create]
[INFO]
-----
-
[INFO] Using following parameters for creating Archetype:
maven-archetype-webapp:RELEASE
[INFO]
-----
-
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.ch05
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: ~/examples
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: simple-webapp
[INFO] ***** End of debug info from resources from generated
POM *****
[INFO] Archetype created in dir: ~/examples/simple-webapp
```

在 **Maven Archetype** 插件创建好了项目之后，切换目录至 `simple-web` 后看一下 `pom.xml`。你会看到如下的 XML 文档：

Example 5.1. simple-web 项目的初始 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch05</groupId>
  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
```

```

<version>1.0-SNAPSHOT</version>
<name>simple-webapp Maven Webapp</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <finalName>simple-webapp</finalName>
</build>
</project>

```

注意 `packaging` 元素包含的值是 `war`。这种打包类型配置让 **Maven** 以 **WAR** 文件的形式生成一个 **web** 应用。一个打包类型为 `war` 的项目，将会在 `target/` 目录创建一个 **WAR** 文件，这个文件的默认名称是 `${artifactId}-${version}.war`。对于这个项目，默认的 **WAR** 文件是 `target/simple-webapp-1.0-SNAPSHOT.war`。在这个 `simple-webapp` 项目中，我们已经通过在项目的构建配置中加入 `finalName` 元素来自定义这个生成的 **WAR** 文件的名称。根据 `simple-webapp` 的 `finalName`，`package` 阶段生成的 **WAR** 文件为 `target/simple-webapp.war`。

5.4. 配置 Jetty 插件



在你已经编译，测试并且打包了你的 **web** 应用之后，你会想要将它部署到一个 **servlet** 容器中，然后测试一下由 **Maven Archetype** 插件创建的 `index.jsp`。通常情况下，你需要下载 **Jetty** 或者 **Apache Tomcat**，解压分发包，复制你的应用程序 **WAR** 文件至 `webapps/` 目录，然后启动你的容器。现在，实现同样的目的，你不再需要做这些事情。取而代之的是，你可以使用 **Maven Jetty** 插件在 **Maven** 中运行你的 **web** 应用。为此，你需要在项目的 `pom.xml` 中配置 **Maven Jetty** 插件。在你项目的构建配置中添加如下插件元素：

Example 5.2. 配置 Jetty 插件

```

<project>
  [...]
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>

```

```
</plugin>
</plugins>
</build>
[...]
</project>
```

在项目的 pom.xml 中 配置好 **Maven Jetty** 插件之后，你就可以调用 **Jetty** 插件的 **Run** 目标在 **Jetty Servlet** 容器中启动你的 **web** 应用。如下运行 **mvn jetty:run**：

```
~/examples$ mvn jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[INFO] Webapp source directory = \
      /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp
[INFO] web.xml file = \
      /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp/WEB-INF
      /web.xml
[INFO] Classes =
      /Users/tobrien/svnw/sonatype/examples/simple-webapp/target/classes
2007-11-17 22:11:50.532::INFO: Logging to STDERR via
org.mortbay.log.StdErrLog
[INFO] Context path = /simple-webapp
[INFO] Tmp directory = determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] Webapp directory = \
      /Users/tobrien/svnw/sonatype/examples/simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-17 22:11:50.673::INFO: jetty-6.1.6rc1
2007-11-17 22:11:50.846::INFO: No Transaction manager found - if your webapp
requires one, \
      please configure one.
2007-11-17 22:11:51.057::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

当 **Maven** 启动了 **Jetty Servlet** 容器之后，在浏览器中载入 URL <http://localhost:8080/simple-webapp/>。Archetype 生成的简单页面 index.jsp 没什么价值；它包含了一个文本为“Hello World!”的二级标题。**Maven** 认为 **web** 应用程序的文档根目录为 src/main/webapp。这个目录就是存放 index.jsp 的目录。index.jsp 的内容为 [Example 5.3. “src/main/webapp/index.jsp 的内容”](#)：

Example 5.3. src/main/webapp/index.jsp 的内容

```
<html>
  <body>
    <h2>Hello World!</h2>
  </body>
</html>
```

在 src/main/webapp/WEB-INF 目录中我们会找到可能是最小的 web 应用程序描述符 web.xml。

Example 5.4. src/main/webapp/WEB-INF/web.xml 的内容

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

<web-app>
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

5.5. 添加一个简单的 Servlet



一个只有一个单独的 JSP 页面而没有任何配置好的 servlet 的 web 应用程序基本是没用的。让我们为这个应用添加一个简单的 servlet，同时为 pom.xml 和 web.xml 做些改动以支持这个变化。首先，我们需要在目录 src/main/java 下创建一个名为 org.sonatype.mavenbook.web 的新的包。

```
$ mkdir -p src/main/java/org/sonatype/mavenbook/web
$ cd src/main/java/org/sonatype/mavenbook/web
```

包创建好之后，切换目录至 src/main/java/org/sonatype/mavenbook/web，创建一个名为 SimpleServlet.java 的 servlet 类，代码如下：

Example 5.5. SimpleServlet 类

```
package org.sonatype.mavenbook.web;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
```

```

        PrintWriter out = response.getWriter();
        out.println( "SimpleServlet Executed" );
        out.flush();
        out.close();
    }
}

```

我们的 `SimpleServlet` 仅此而已，一个往响应 `Writer` 打印一条简单信息的 `servlet`。为了把这个 `servlet` 添加到你的 `web` 应用，并且使其与请求路径匹配，需要添加如下的 `servlet` 和 `servlet-mapping` 元素至你项目的 `web.xml` 文件。文件 `web.xml` 可以在目录 `src/main/webapp/WEB-INF` 中找到。

Example 5.6. 匹配 Simple Servlet

```

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

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.SimpleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
</web-app>

```

一切文件就绪，准备测试这个 `servlet`。src/main/java 下的类，以及 `web.xml` 已经被更新了。在启动 `Jetty` 插件之前，运行 `mvn compile` 以编译你的项目：

```

~/examples$ mvn compile
...
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to ~/examples/ch05/simple-webapp/target/classes
[INFO]
-----
[ERROR] BUILD FAILURE
[INFO]
-----
[INFO] Compilation failure

```

```
~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[4,0] \
    package javax.servlet does not exist

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[5,0] \
    package javax.servlet.http does not exist

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[7,35] \
    cannot find symbol
    symbol:   class HttpServlet
    public class SimpleServlet extends HttpServlet {

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[8,22] \
    cannot find symbol
    symbol   : class HttpServletRequest
    location: class org.sonatype.mavenbook.web.SimpleServlet

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[9,22] \
    cannot find symbol
    symbol   : class HttpServletResponse
    location: class org.sonatype.mavenbook.web.SimpleServlet

~/ch05/simple-webapp/src/main/java/org/sonatype/mavenbook/web/SimpleServlet
.java:[10,15] \
    cannot find symbol
    symbol   : class ServletException
    location: class org.sonatype.mavenbook.web.SimpleServlet
```

编译失败了，因为你的 **Maven** 项目没有对 **Servlet API** 的依赖。在下一节，我们会为你项目的 **POM** 添加 **Servlet API**。

5.6. 添加 J2EE 依赖



为了编写一个 `servlet`，我们需要添加 `Servlet API` 作为项目依赖。`Servlet` 规格说明是一个 `JAR` 文件，它能从 `Sun Microsystems` 的站点下载到 <http://java.sun.com/products/servlet/download.html>。JAR 文件下载好之后你需要把它安装到位于 `~/.m2/repository` 的 `Maven` 本地仓库。你必须为所有 `Sun Microsystems` 维护的 `J2EE API` 重复同样的过程，包括 `JNDI`，`JDBC`，`Servlet`，`JSP`，`JTA`，以及其它。如果你不想这么做因为觉得这样太无聊了，其实不只有你这么认为。幸运的是，有一种更简单的方法来下载所有这些类库并安装到本地仓库——`Apache Geronimo` 的独立的开源实现。

很多年以来，获取 `Servlet` 规格说明 `JAR` 文件的唯一方式是从 `Sun Microsystems` 下载。你必须到 `Sun` 的 `web` 站点，同意并且点击它的许可证协议，这样才能访问 `Servlet JAR`。这是必须的，因为 `Sun` 的规格说明 `JAR` 文件并没有使用一个允许再次分发的许可证。很多年来编写一个 `Servlet` 或者使用 `JDBC` 之前你必须手工下载 `Sun` 的构件。这很乏味并且令人恼火，直到 `Apache Geronimo` 项目创建了很多通过 `Sun` 认证的企业级规格说明实现。这些规格说明 `JAR` 是按照 `Apache` 软件许可证版本 2.0 发布的，该许可证允许对源代码和二进制文件进行免费的再次分发。现在，对你的程序来说，从 `Sun Microsystems` 下载的 `Servlet API JAR` 和从 `Apache Geronimo` 项目下载的 `JAR` 没什么大的差别。它们同样都通过了 `Sun Microsystems` 的严格的一个测试兼容性工具箱(TCK)。

添加像 `JSP API` 或者 `Servlet API` 这样的依赖现在很简单明了了，不再需要你从 `web` 站点手工下载 `JAR` 文件然后再安装到本地仓库。关键是你必须知道去哪里找，使用什么 `groupId`，`artifactId`，和 `version` 来引用恰当的 `Apache Geronimo` 实现。给 `pom.xml` 添加如下的依赖元素以添加对 `Servlet` 规格说明 `API` 的依赖。

Example 5.7. 添加 `Servlet 2.4` 规格说明作为依赖

```
<project>
[... ]
<dependencies>
[... ]
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-servlet_2.4_spec</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
[... ]
</project>
```

所有 `Apache Geronimo` 规格说明的实现的 `groupId` 都是 `org.apache.geronimo.specs`。这个 `artifactId` 包含了你熟悉的规格说明的版本

号；例如，如果你要引入 **Servlet 2.3** 规格说明，你将使用的 `artifactId` 是 `geronimo-servlet_2.3_spec`，如果你想要引入 **Servlet 2.4** 规格说明，那么你的 `artifactId` 将会是 `geronimo-servlet_2.4_spec`。你必须看一下 **Maven** 的公共仓库来决定使用什么版本。最好使用某个规格说明实现的最新版本。如果你在寻找某个特定的 **Sun** 规格说明对应的 **Apache Geronimo** 项目，我们已经在附录归纳了一个可用规格说明的列表。

这里还有必要指出的是我们的这个依赖使用了 `provided` 范围。这个范围告诉 **Maven** `jar` 文件已经由容器“提供”了，因此不再需要包含在 `war` 中。

如果你对在这个简单 **web** 应用编写自定义 **JSP** 标签感兴趣，你将需要添加对 **JSP 2.0** 规格说明的依赖。使用以下配置来加入这个依赖。

Example 5.8. 添加 JSP 2.0 规格说明作为依赖

```
<project>
[... ]
<dependencies>
[... ]
<dependency>
  <groupId>org.apache.geronimo.specs</groupId>
  <artifactId>geronimo-jsp_2.0_spec</artifactId>
  <version>1.1</version>
  <scope>provided</scope>
</dependency>
</dependencies>
[... ]
</project>
```

在添加好这个 **Servlet** 规格说明依赖之后，运行 **mvn clean install**，然后运行 **mvn jetty:run**。

```
[tobrien@t1 simple-webapp]$ mvn clean install
...
[tobrien@t1 simple-webapp]$ mvn jetty:run
[INFO] [jetty:run]
...
2007-12-14 16:18:31.305::INFO: jetty-6.1.6rc1
2007-12-14 16:18:31.453::INFO: No Transaction manager found - if your webapp
requires one, \
    please configure one.
2007-12-14 16:18:32.745::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

到此为止，你应该能够获取 **SimpleServlet** 的输出。在命令行，你可以使用 **curl** 在标准输出打印 **servlet** 的输出。

```
~/examples$ curl http://localhost:8080/simple-webapp/simple
```

SimpleServlet Executed

5.7. 小结



阅读完本章之后，你应该能够启动一个简单的 **web** 应用程序。本章并没有详细描述用很多不同的方式来创建一个完整的 **web** 应用，其它章节对那些包含很多流行 **web** 框架和技术的项目提供了更全面的介绍。

Chapter 6. 一个多模块项目

[6.1. 简介](#)

[6.1.1. 下载本章样例](#)

[6.2. simple-parent 项目](#)

[6.3. simple-weather 模块](#)

[6.4. simple-webapp 模块](#)

[6.5. 构建这个多模块项目](#)

[6.6. 运行 Web 应用](#)

6.1. 简介



本章，我们创建一个结合了前两章样例的多模块项目。[???](#)中开发的 **simple-weather** 代码将会与 [Chapter 5, 一个简单的Web 应用](#)中定义的 **simple-webapp** 结合以创建一个新的 **web** 应用，它获取天气预报信息然后显示在 **web** 页面上。本章最后，你能将使用 **Maven** 开发复杂的，多模块项目。

6.1.1. 下载本章样例



该样例中开发的多模块项目包含了[???](#)和 [Chapter 5, 一个简单的Web 应用](#)中项目的修改的版本，我们不会再使用 **Maven Archetype** 插件来生成这个多模块项目。我们强烈建议当你在阅读本章内容的时候，下载样例代码作为一个补充参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，

<http://www.sonatype.com/book/mvn-examples-1.0.zip> 或者

<http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>。解开存档文件至

任意目录，然后到 **ch06/** 目录。在 **ch06/** 目录你会看到一个名为 **simple-parent/** 的目录，它包含了本章开发出来的多模块 **Maven** 项目。在这个 **simple-parent/** 项目目录中，你会看到一个 **pom.xml**，以及两个子模块目录 **simple-weather/** 和 **simple-webapp/**。如果你想要在浏览器里看样例代码，访问

<http://www.sonatype.com/book/examples-1.0>，然后点击 **ch06/** 目录。

6.2. simple-parent 项目



一个多模块项目通过一个父 POM 引用一个或多个子模块来定义。在 simple-parent/ 目录中你能找到一个父 POM（也称为顶层 POM）为 simple-parent/pom.xml。

Example 6.1. simple-parent 项目的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.ch06</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 6 Simple Parent Project</name>

  <modules>
    <module>simple-weather</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <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>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>
</project>
```

```
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

注意 **simple-parent** 定义了一组 **Maven** 坐标: **groupId** 是 **org.sonatype.mavenbook**, **artifactId** 是 **simple-parent**, **version** 是 **1.0**。这个父项目不像之前的项目那样创建一个 **JAR** 或者一个 **WAR**, 它仅仅是一个引用其它 **Maven** 项目的 **POM**。像 **simple-parent** 这样仅提供项目对象模型的项目, 正确的打包类型是 **pom**。pom.xml 中下一部分列出了项目的子模块。这些模块在 **modules** 元素中定义, 每个 **modules** 元素对应了一个 **simple-parent/**目录下的子目录。**Maven** 知道去这些子目录寻找 pom.xml 文件, 并且, 在构建的 **simp-parent** 的时候, 它会将这些子模块包含到要构建的项目中。

最后, 我们定义了一些将会被所有子模块继承的设置。**simple-parent** 的 **build** 部分配置了所有 **Java** 编译的目标是 **Java 5 JVM**。因为 **compiler** 插件默认绑定到了生命周期, 我们就可以使用 **pluginManagement** 部分来配置。我们将会在后面的章节详细讨论 **pluginManagement**, 区分为默认的插件提供配置和真正的绑定插件是很容易的。**dependencies** 元素将 **JUnit 3.8.1** 添加为一个全局的依赖。**build** 配置和 **dependencies** 都会被所有的子模块继承。使用 **POM** 继承允许你添加一些全局的依赖如 **JUnit** 和 **Log4J**。

6.3. simple-weather 模块



我们要看的第一个子模块是 **simple-weather** 子模块。这个子模块包含了所有用来与 **Yahoo! weather** 信息源交互的类。

Example 6.2. simple-weather 模块的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch06</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 6 Simple Weather API</name>
```

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.5</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

在 simple-weather 的 pom.xml 文件中我们看到该模块使用一组 **Maven** 坐标引用了一个父 POM。simple-weather 的父 POM 通过一个值为 org.sonatype.mavenbook 的 groupId, 一个值为 simple-parent 的 artifactId, 以及一个值为 1.0 的 version 来定义。注意子模块中我们不再需要重新定义 groupId 和 version, 它们都从父项目继承了。

Example 6.3. WeatherService 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

public class WeatherService {

    public WeatherService() {}

    public String retrieveForecast( String zip ) throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        return new WeatherFormatter().format( weather );
    }
}
```

WeatherService 类在 src/main/java/org/sonatype/mavenbook/weather 中定义, 它简单的调用[???](#)中定义的三个对象。在本章的样例中, 我们正创建一个单独的项目, 它包含了将会在 web 应用项目被引用的 service 对象。这是一个在企业级 Java 开发中常见的模型, 通常一个复杂的应用包含了不止一个的简单 web 应用。你可能拥有一个企业应用, 它包含了多个 web 应用, 以及一些命令行应用。通常你会想要重构那些通用的逻辑至一个 service 类, 以被很多项目重用。这就是我们创建 WeatherService 类的理由, 在此之后, 你就能看到 simple-webapp 项目是如何引用在 simple-weather 中定义的 service 对象。

retrieveForecast() 方法接受一个包含邮政编码的 String。之后这个邮政编码参数被传给 YahooRetriever 的 retrieve() 方法, 后者从 Yahoo! Weather 获取 XML。从 YahooRetriever 返回的 XML 被传给 YahooParser 的 parse() 方法, 后者继而又返回一个 Weather 对象。之后这个 Weather 对象被 WeatherFormatter 格式化成一个像样的 String。

6.4. simple-webapp 模块



simple-webapp 模块是在 simple-parent 项目中引用的第二个子模块。这个 web 项目依赖于 simple-weather 模块，并且包含了一些用来展示从 Yahoo! Weather 服务查询到的结果的 servlet。

Example 6.4. simple-webapp 模块的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch06</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>simple-webapp Maven Webapp</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch06</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

simple-weather 模块定义了一个十分简单的 `servlet`，它从 HTTP 请求读取一个邮政编码，调用 [Example 6.3, “WeatherService 类”](#)中展示的 `WeatherService`，然后将结果打印至 HTTP 响应的 `Writer`。

Example 6.5. simple-webapp 的 WeatherServlet

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.WeatherService;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class WeatherServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String zip = request.getParameter( "zip" );
        WeatherService weatherService = new WeatherService();
        PrintWriter out = response.getWriter();
        try {
            out.println( weatherService.retrieveForecast( zip ) );
        } catch( Exception e ) {
            out.println( "Error Retrieving Forecast: " + e.getMessage() );
        }
        out.flush();
        out.close();
    }
}
```

在 `WeatherServlet` 中，我们初始化了一个在 `simple-weather` 中定义的 `WeatherService` 类的实例。在请求参数中提供的邮政编码被传给 `retrieveForecast()` 方法，并且返回结果被打印至 HTTP 响应的 `Writer`。

最后，`src/main/webapp/WEB-INF` 目录下的 `web.xml` 将所有这一切绑在一起。`web.xml` 中的 `servlet` 和 `servlet-mapping` 元素将路径 `/weather` 匹配至 `WeatherServlet`。

Example 6.6. simple-webapp 的 web.xml

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

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
```

```

    <servlet-name>simple</servlet-name>
    <servlet-class>org. sonatype. mavenbook. web. SimpleServlet</servlet-class>
</servlet>
<servlet>
    <servlet-name>weather</servlet-name>

<servlet-class>org. sonatype. mavenbook. web. WeatherServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/weather</url-pattern>
</servlet-mapping>
</web-app>

```

6.5. 构建这个多模块项目



既然 simple-weather 项目包含了所有与 Yahoo! Weather 服务交互的代码，以及 simple-webapp 项目包含了一个简单的 **servlet**，是时候将这个应用编译并打包成一个 **WAR** 文件了。为此，你会想要以合适的顺序编译并安装这两个项目；以为 simple-webapp 依赖于 simple-weather，simple-weather 的 **JAR** 需要在 simple-webapp 项目被编译之前就被创建好。为此，你需要从 simple-parent 项目运行 **mvn clean install** 命令。

```

~/examples/ch06/simple-parent$ mvn clean install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Simple Parent Project
[INFO]   simple-weather
[INFO]   simple-webapp Maven Webapp
[INFO]
-----
[INFO] Building simple-weather
[INFO]   task-segment: [clean, install]
[INFO]
-----
[...]
[INFO] [install:install]
[INFO] Installing simple-weather-1.0.jar to simple-weather-1.0.jar

```

```
[INFO]
-----
[INFO] Building simple-webapp Maven Webapp
[INFO]    task-segment: [clean, install]
[INFO]
-----
[...]
[INFO] [install:install]
[INFO] Installing simple-webapp.war to simple-webapp-1.0.war
[INFO]
[INFO]
-----

[INFO] Reactor Summary:
[INFO]
-----

[INFO] Simple Parent Project ..... SUCCESS [3.041s]
[INFO] simple-weather ..... SUCCESS [4.802s]
[INFO] simple-webapp Maven Webapp ..... SUCCESS [3.065s]
[INFO]
-----
```

当 **Maven** 执行一个带有子模块的项目的时候，**Maven** 首先载入父 **POM**，然后定位所有的子模块 **POM**。**Maven** 然后将所有这些项目的 **POM** 放入到一个称为 **Maven** 反应堆（**Reactor**）的东西中，由它负责分析模块之间的依赖关系。这个反应堆处理组件的排序，以确保相互独立的模块能以适当的顺序被编译和安装。

Note

除非需要做更改，反应堆一直维持定义在 **POM** 中的模块的顺序。为此一个有帮助的思维模型是，那些依赖于兄弟项目的项目在列表中被“向下按”，直到依赖顺序被满足。在很少的情况下，重新安排你构建的模块顺序可能很方便——例如你想要一个频繁的不稳定的模块接近构建的开端。

一旦反应堆解决了项目构建的顺序，**Maven** 就会在多模块构建中为每个模块执行特定的目标。本例中，你能看到 **Maven** 在 `simple-webapp` 之前构建了 `simple-weather`，为每个子模块执行了 **mvn clean install**。

Note

当你在命令行运行 **Maven** 的时候你经常会在任何其它生命周期阶段前指定 `clean` 生命周期阶段。当你指定 `clean`，你就确认了在编译和打包一个应用之前，**Maven** 会移除旧的输出。运行 `clean` 不是必要的，但这是一个确保你正执行“干净构建”的十分有用的预防措施。

6.6. 运行 Web 应用



一旦这个多模块项目已经通过从父项目 `simple-project` 执行 **`mvn clean install`** 行安装好了，你可以切换目录至 `simple-webapp` 项目，然后运行 Jetty 插件的 `Run` 目标。

```
~/examples/ch06/simple-parent/simple-webapp $ mvn jetty:run
[INFO]
-----
[INFO] Building simple-webapp Maven Webapp
[INFO]    task-segment: [jetty:run]
[INFO]
-----
[...]
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[...]
[INFO] Webapp directory = ~/examples/ch06/simple-parent/\
                        simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-18 1:58:26.980::INFO:   jetty-6.1.6rc1
2007-11-18 1:58:26.125::INFO: No Transaction manager found
2007-11-18 1:58:27.633::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Jetty 启动之后，在浏览器载入

<http://localhost:8080/simple-webapp/weather?zip=01201>，你应该看到格式化的天气输出。

Chapter 7. 多模块企业级项目

[7.1. 简介](#)

[7.1.1. 下载本章样例](#)

[7.1.2. 多模块企业级项目](#)

[7.1.3. 本例中所用的技术](#)

[7.2. simple-parent 项目](#)

[7.3. simple-model 模块](#)

[7.4. simple-weather 模块](#)

[7.5. simple-persist 模块](#)

[7.6. simple-webapp 模块](#)

[7.7. 运行这个 Web 应用](#)

[7.8. simple-command 模块](#)

[7.9. 运行这个命令行程序](#)

[7.10. 小结](#)

[7.10.1. 编写接口项目程序](#)

7.1. 简介



本章，我们创建一个多模块项目，它从 [Chapter 6, 一个多模块项目](#)和 [Chapter 5, 一个简单的Web 应用](#)的样例演化成一个使用了 Spring Framework 和 Hibernate 创建的，从 Yahoo! Weather 信息源读取数据，包含一个简单 web 应用和一个命令行工具的项目。[???中](#)开发的 simple-weather 代码将会和 [Chapter 5, 一个简单的Web 应用](#)中开发的 simple-weather 项目结合。在创建这个多模块项目的过程中，我们将会探索 Maven 并且讨论用不同方式来创建模块化项目以鼓励重用。

7.1.1. 下载本章样例



该样例中开发的多模块项目包含了[???和 Chapter 5, 一个简单的Web 应用](#)中项目的修改的版本，我们不会再使用 Maven Archetype 插件来生成这个多模块项目。我们强烈建议当你在阅读本章内容的时候，下载样例代码作为一个补充参考。本章的样例项目包含在本书的样例代码中，你可以从两个地方下载，

<http://www.sonatype.com/book/mvn-examples-1.0.zip> 或者

<http://www.sonatype.com/book/mvn-examples-1.0.tar.gz>。解开存档文件至任意目录，然后到 ch07/目录。在 ch07/目录你会看到一个名为 simple-parent/的目录，它包含了本章开发出来的多模块 Maven 项目。在这个 simple-parent/项目目录中，你会看到一个 pom.xml，以及五个子模块目录 simple-model/, simple-persist/, simple-command/, simple-weather/和 simple-webapp/。如果你想要在浏览器里看样例代码，访问 <http://www.sonatype.com/book/examples-1.0>，然后点击 ch07/目录。

7.1.2. 多模块企业级项目



展示一个巨大企业级项目的复杂度远远超出了本书的范围。这样的项目的特征有，多数数据库，与外部系统的集成，子项目通过部门来划分。这些项目通常跨越了数千行代码，牵涉了数十或数百软件开发者的努力。虽然这样的完整样例超出了本书的范围，我们仍然可以为你提供一个能让你想起大型企业应用的样例项目。在小结中我们提议了一些在本章描述之外的模块化可能性。

本章，我们将会看一个多模块 Maven 项目，它将产生两个应用程序：一个对于 Yahoo! Weather 信息源的命令行查询工具，以及查询同样信息源的一个 web 应用。两个应用都会将查询结果存储到一个内嵌数据库中。都允许用户从内嵌数据库中获取历史天气数据。都会重用应用程序逻辑，并且共享一个持久化类库。本章样例基于在[???中](#)介绍的 Yahoo! Weather 解析代码构建。该项目被划分成如 [Figure 7.1, “多模块企业级应用的模块关系”](#)所示的五个子项目。

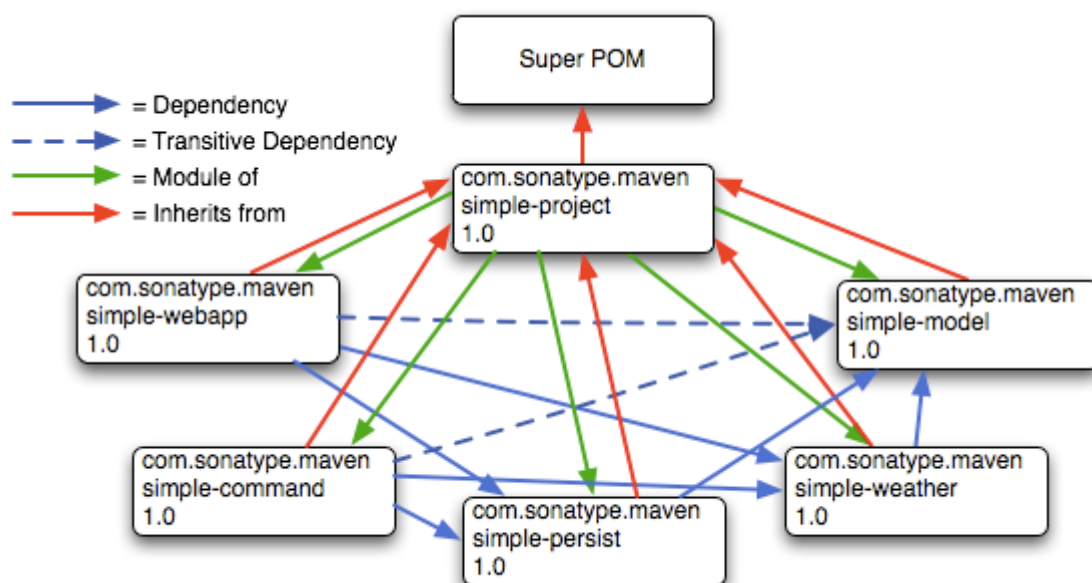


Figure 7.1. 多模块企业级应用的模块关系

在 [Figure 7.1, “多模块企业级应用的模块关系”](#)中，你能看到 `simple-parent` 有五个子模块，它们分别是：

simple-model

该模块定义了一个简单的对象模型，对从 **Yashoo! Weather** 信息源返回的数据建模。该对象模型包含了 `Weather`, `Condition`, `Atmosphere`, `Location`, 和 `Wind` 对象。当我们的应用程序解析 **Yahoo! Weather** 信息源的时候，`simple-weather` 中定义的解析器会解析 `XML` 并创建供应用程序使用的 `Weather` 对象。该项目还包含了使用 **Hibernate 3** 标注符标注的模型对象，它们在 `simple-persist` 的逻辑中被用来映射每个模型对象至关系数据库中对应的表。

simple-weather

该模块包含了所有用来从 **Yahoo! Weather** 数据源获取数据并解析结果 `XML` 的逻辑。从数据源返回的 `XML` 被转换成 `simple-model` 中定义的模型对象。`simple-weather` 有一个对 `simple-model` 的依赖。`simple-weather` 定义了一个 `WeatherService` 对象，该对象会被 `simple-command` 和 `simple-webapp` 项目引用。

simple-persist

该模块包含了一些数据访问对象(DAO)，这些对象将 `Weather` 对象存储在一个内嵌数据库中。这个多模块项目中的两个应用都会使用 `simple-persist` 中定义的 `DAO` 来将数据存储至内嵌数据库中。本项目中定义的 `DAO` 能理解并返回 `simple-model` 定义的模型对象。`simple-persist` 有一个对 `simple-model` 的依赖，它也依赖于模型对象上的 **Hibernate** 标注。

simple-webapp

这个 web 应用项目包含了两个 Spring MVC 控制器实现，控制器使用了 simple-weather 中定义的 WeatherService，以及 simple-persist 中定义的 DAO。simple-webapp 有对于 simple-weather 和 simple-persist 的直接依赖；还有一个对于 simple-model 的传递性依赖。

simple-command

该模块包含了一个用来查询 Yahoo! Weather 信息源的简单命令行工具。它包含了一个带有静态 main() 方法的类，与 simple-weather 中定义的 WeatherService 和 simple-persist 中定义的 DAO 交互。simple-command 有对于 simple-weather 和 simple-persist 的直接依赖；还有一个对于 simple-model 的传递性依赖。

本章设计的项目一方面够简单，以能在一本书中介绍，又够复杂，能提供一组五个子模块。该样例有一个带有五个类的模型项目，带有两个服务类的持久化类库，带有五六类的天气解析类库，但是一个现实系统可能有一个带有数百对象的模型项目，很多持久化类库，以及跨越多个部门的服务类库。虽然我们试图确保本例中的代码尽可能的直接以能在短时间内理解，但我们也不怕麻烦的以模块化的方式构建了这个项目。你可能会要看一下本章的样例，然后会认为 Maven 为我们这个只有五个类的模型项目带来了太多的复杂度。虽然使用 Maven 确实建议一定程度的模块化，但这里我们不怕麻烦的将样例项目弄得复杂，目的是展示 Maven 的多模块特性。

7.1.3. 本例中所用的技术



本章样例中涉及了一些十分流行，但与 Maven 没有直接关系的而技术。这些技术是 Spring Framework 和 Hibernate。Spring Framework 是一个反转控制(IoC)容器，以及一组目的在于简化与各种 J2EE 类库交互的框架。使用 Spring Framework 作为应用程序开发的基础框架能让你访问很多有用的抽象接口，它们能简化与持久化框架如 Hibernate 或者 iBatis 的交互，以及企业 API 如 JDBC，JNDI，和 JMS。Spring Framework 在过去一些年变得十分流行，作为对来自 Sun 微系统的重量级企业标准的替代。Hibernate 是一个被广泛使用的对象-关系映射框架，能让你与关系数据库的交互就像它们是 Java 对象的集合一样。本例关注构建一个简单的 web 应用和一个命令行应用，它们使用 Spring Framework 为应用暴露了一组可重用的组件，使用 Hibernate 将天气数据持久化至内嵌数据库。

我们决定包含对这些框架的参考以展示在使用 Maven 的时候如何使用这些技术构建项目。虽然本章中我们会大概介绍这些技术，但不是完整的解释这些技术。要了解更多关于 Spring Framework 的信息，请查看该项目的 web 站点：

<http://www.springframework.org/>。要了解更多关于 Hibernate 和 Hibernate 标注的信息，请查看该项目的 web 站点：<http://www.hibernate.org>。本章使用了 HSQLDB 作为一个内嵌数据库；要了解更多的关于该数据库的信息，访问该项目的 web 站点：<http://hsqldb.org/>。

7.2. simple-parent 项目



该 simple-parent 项目有一个 pom.xml，它引用了五个子模块：simple-command, simple-model, simple-weather, simple-persist, 和 simple-webapp。顶层的 pom.xml 在 [Example 7.1. “simple-parent 项目的 POM”](#) 中显示。

Example 7.1. simple-parent 项目的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <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>
    </pluginManagement>
  </build>

  <dependencies>
```

```

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>3.8.1</version>
  <scope>test</scope>
</dependency>
</dependencies>
</project>

```

注意这个父 POM 和 [Example 6.1, “simple-parent 项目的 POM”](#) 中定义的父 POM 的相似度。这两个 POM 唯一真正不同的地方是子模块列表。前面的样例中只列出了两个子模块，而这里的父 POM 列出了五个子模块。下面的小节会详细讨论所有这五个子模块。因为我们的样例使用了 Java 标注，我们将编译器的目标配置成 Java 5 JVM。

7.3. simple-model 模块



大多数企业项目需要做的第一件事情是建立对象模型。一个对象模型抓取了系统中一组核心的领域对象。一个银行系统可能会有包括 Account, Customer, Transaction 的对象模型。或者有一个对体育比赛比分建模的系统，有一个 Team 和一个 Game 对象。不管它是什么，你都需要将你系统中的概念建模成对象模型。在 Maven 项目中，将这个对象模型分割成单独的项目以被广泛引用，是一种常用的实践。在我们这个系统中，我们将每个对 Yahoo! Weather 数据源的查询建模成为 Weather 对象，它本身又引用了四个其它的对象。风向，风速等存储 Wind 在对象中。地址信息包括邮编，城市等信息存储在 Location 类中。大气信息如湿度，可见度，气压等存储在 Atmosphere 类中。而对环境，气温，以及观察日期的文本描述存储在 Condition 类中。

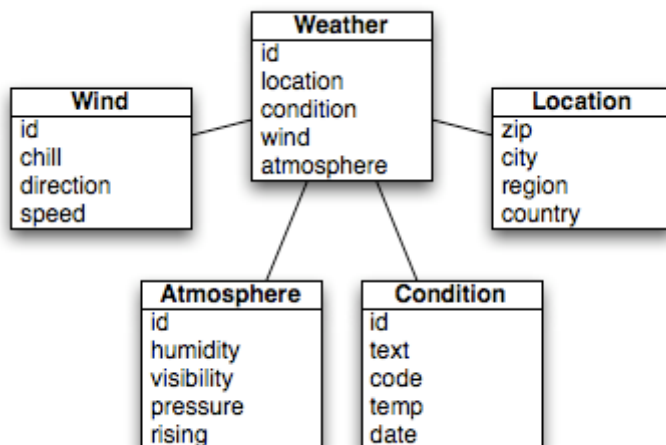


Figure 7.2. 天气数据的简单对象模型

这个简单对象模型的 pom.xml 文件含有一个依赖需要一些解释。我们的对象模型用 **Hibernate** 标注符标注了。我们用这些标注来映射模型对象至关系数据库中的表。这个依赖是 org.hibernate:hibernate-annotations:3.3.0.ga。看一下 [Example 7.2, “simple-model 的 pom.xml”](#) 中显示的 pom.xml, 然后看接下来几个展示这些标注的例子。

Example 7.2. simple-model 的 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
  </dependencies>
</project>
```

在 src/main/java/org/sonatype/mavenbook/weather/model 中, 有 Weather.java, 它是一个标注过的 Weather 型对象。这个 Weather 对象是一个简单的 Java bean。这意味着它的私有成员变量如 id, location, condition, wind, atmosphere, 和 date, 通过公共的 **getter** 和 **setter** 方法暴露, 并且遵循这样的模式: 如果一个属性名为 name, 那么会有一个公有的无参数方法 getName(), 还有一个带有一个参数的 setter 方法 setName(String name)。我们只是展示了 id 属性的 **getter** 和 **setter** 方法, 其它属性的 **getter** 和 **setter** 方法类似, 所以这里跳过了, 以节省篇幅。请看 [Example 7.3, “标注的 Weather 模型对象”](#)。

Example 7.3. 标注的 Weather 模型对象

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

import java.util.Date;

@Entity
@NamedQueries({
    @NamedQuery(name= "Weather.byLocation",
        query= "from Weather w where w.location = :location")
})
public class Weather {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne(cascade=CascadeType.ALL)
    private Location location;

    @OneToOne(mappedBy= "weather", cascade=CascadeType.ALL)
    private Condition condition;

    @OneToOne(mappedBy= "weather", cascade=CascadeType.ALL)
    private Wind wind;

    @OneToOne(mappedBy= "weather", cascade=CascadeType.ALL)
    private Atmosphere atmosphere;

    private Date date;

    public Weather() {}

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }

    // All getter and setter methods omitted...
}
```

在 Weather 类中，我们使用 **Hibernate** 标注以为 simple-persist 项目提供指导。这些标注由 **Hibernate** 用来将对象与关系数据库映射。尽管对 **Hibernate** 标注的完整解释超出了本书的范围，这里是一个为好奇者的简单解释。**@Entity** 标注标记一个类为持

久化对象。我们省略了这个类的@Table 标注, 因此 **Hibernate** 将会使用这个类的名字来映射表名。@NamedQueries 注解定义了一个 simple-persist 中 WeatherDAO 使用的查询。@NamedQuery 注解中的查询语句是用一个叫做 **Hibernate** 查询语言(HQL)编写的。每个成员变量的注解定义了这一列的类型, 以及该列暗示的表关联关系。

Id

id 属性用@Id 进行标注。这标记 id 属性为一个包含数据库表主键的属性。@GeneratedValue 控制新的主键值如何产生。该例中, 我们使用 IDENTITY GenerationType, 它使用了下层数据库的主键生成设施。

Location

每个 Weather 对象实例对应了一个 Location 对象。一个 Location 对象含有一个邮政编码, 而@ManyToOne 确认所有指向同一个 Location 对象的 Weather 对象引用了同样一个实例。@ManyToOne 的 cascade 属性确保每次我们持久化一个 Weather 对象的时候也会持久化一个 Location 对象。

Condition, Wind, Atmosphere

这些对象的每一个都作为@OneToOne 而且 CascadeType 为 ALL 进行映射。这意味着每次我们保存一个 Weather 对象, 我们将会往 Weather 表, Condition 表, Wind 表, 和 Atmosphere 表, 插入一行,

Date

Date 没有被标注, 这以为着 **Hibernate** 将会使用所有列的默认值来定义该映射。列名将会是 date, 列的类型会是匹配 Date 对象的适当时间。

Note

如果你有一个希望从表映射中忽略的属性, 你可以使用@Transient 标注这个属性。

接着, 看一下一个二级的模型对象, Condition, 如 [Example 7.4, "simple-model 的 Condition 模型对象"](#)所示。这个类同样也存在于 src/main/java/org/sonatype/mavenbook/weather/model。

Example 7.4. simple-model 的 Condition 模型对象

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

@Entity
public class Condition {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String text;
    private String code;
    private String temp;
```

```

private String date;

@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="weather_id", nullable=false)
private Weather weather;

public Condition() {}

public Integer getId() { return id; }
public void setId(Integer id) { this.id = id; }

// All getter and setter methods omitted...
}

```

这个 Condition 类类似于 Weather 类。它被标注为一个@Entity，在 id 属性上也有相似的标注。text, code, temp, 和 date 属性也都使用默认的列设置，weather 属性使用了@OneToOne 进行标注，而另一个标注通过一个名为 weather_id 的外键引用关联的 Weather 对象。

7.4. simple-weather 模块



我们将要检查的下一个模块可以被认为是一个“服务”。这个 simple-weather 模块包含了所有从 Yahoo! Weather RSS 数据源获取数据并解析的必要逻辑。虽然 simple-weather 只包含了三个 Java 类和一个 JUnit 测试，它还将展现为一个单独的组件，WeatherService，同时为简单 web 应用和简单命令行工具服务。通常来说一个企业级项目会包含一些 API 模块，这写模块包含了重要的业务逻辑，或者与外部系统交互的逻辑。一个银行系统可能有一个模块，从第三方数据提供者获取并解析数据，而一个显示赛事比分的系统可能会与一个提供实时篮球或足球比分的 XML 数据源进行交互。在 [Example 7.5, “simple-weather 模块的 POM”](#)中，该模块封装了所有的网络活动，以及与 Yahoo! Weather 交互涉及的 XML 解析活动。其它依赖于该模块的项目只要简单的调用 WeatherService 的 retrieveForecast() 方法，该方法接受一个邮政编码作为参数，返回一个 Weather 对象。

Example 7.5. simple-weather 模块的 POM

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>

```

```

</parent>
<artifactId>simple-weather</artifactId>
<packaging>jar</packaging>

<name>Simple Weather API</name>

<dependencies>
  <dependency>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-model</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

这个 simple-weather POM 扩展了 simple-parent POM，设置打包方式为 jar，然后添加了下列的依赖：

org.sonatype.mavenbook.ch07:simple-model:1.0

simple-weather 将 Yahoo! Weather RSS 数据源解析成一个 Weather 对象。它有一个对 simple-model 的直接依赖。。

log4j:log4j:1.2.14

simple-weather 使用 Log4J 类库来打印日志信息。

dom4j:dom4j:1.6.1 and jaxen:jaxen:1.1.1

这两个依赖都用来解析从 Yahoo! Weather 返回的 XML。

org.apache.commons:commons-io:1.3.2 (scope=test)

这个 test 范围的依赖是由 YahooParserTest 使用的。

接下来是 WeatherService 类，在 [Example 7.6, “WeatherService 类”](#) 中显示。这个类和 [Example 6.3, “WeatherService 类”](#) 中的 WeatherService 类看起来很像。虽然 WeatherService 名字一样，但与本章中的样例还是有细微的差别。这个版本的 retrieveForecast() 方法返回一个 Weather 对象，而格式就留给调用 WeatherService 的程序去处理。其它的主要变化是，YahooRetriever 和 YahooParser 都是 WeatherService bean 的 bean 属性。

Example 7.6. WeatherService 类

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherService {

    private YahooRetriever yahooRetriever;
    private YahooParser yahooParser;

    public WeatherService() {}

    public Weather retrieveForecast(String zip) throws Exception {
        // Retrieve Data
        InputStream dataIn = yahooRetriever.retrieve(zip);

        // Parse DataS
        Weather weather = yahooParser.parse(zip, dataIn);

        return weather;
    }

    public YahooRetriever getYahooRetriever() {
        return yahooRetriever;
    }

    public void setYahooRetriever(YahooRetriever yahooRetriever) {
        this.yahooRetriever = yahooRetriever;
    }
}
```

```

public YahooParser getYahooParser() {
    return yahooParser;
}

public void setYahooParser(YahooParser yahooParser) {
    this.yahooParser = yahooParser;
}
}

```

最后，在这个项目中我们有一个由 **Spring Framework** 用来创建 **ApplicationContext** 的 **XML** 文件。首先，一些解释：两个应用程序，**web** 应用和命令行工具，都需要和 **WeatherService** 类交互，而且它们都使用名字 **weatherService** 从 **Spring ApplicationContext** 获取此类的一个实例。我们的 **web** 应用使用一个与 **WeatherService** 实例关联的 **Spring MVC** 控制器，我们的命令行应用在静态 **main()** 方法中从 **ApplicationContext** 载入这个 **WeatherService**。为了鼓励重用，我们已经在 **src/main/resources** 中包含了一个 **applicationContext-weather.xml** 文件，这样便在 **classpath** 中可用。依赖于 **simple-weather** 模块的模块可以使用 **Spring Framework** 中的 **ClasspathXmlApplicationContext** 载入这个 **Application Context**。之后它们就能引用名为 **weatherService** 的 **WeatherService** 实例。

Example 7.7. simple-weather 模块的 Spring Application Context

```

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    default-lazy-init="true">

    <bean id="weatherService"
        class="org.sonatype.mavenbook.weather.WeatherService">
        <property name="yahooRetriever" ref="yahooRetriever"/>
        <property name="yahooParser" ref="yahooParser"/>
    </bean>

    <bean id="yahooRetriever"
        class="org.sonatype.mavenbook.weather.YahooRetriever"/>

    <bean id="yahooParser"
        class="org.sonatype.mavenbook.weather.YahooParser"/>
</beans>

```

该文档定义了三个 **bean**: yahooParser, yahooRetriever, 和 weatherService。weatherService **bean** 是 WeatherService 的一个实例, 这个 XML 文档通过引用对应类的命名实例来填充 yahooParser 和 yahooRetriever 属性。可以将这个 applicationContext-weather.xml 文件看作是定义了这个多模块项目中一个子系统的架构。其它项目如 simple-webapp 和 simple-command 可以引用这个上下文, 获取一个已经建立好与 YahooRetriever 和 YahooParser 实例关系的 WeatherService 实例。

7.5. simple-persist 模块



该模块定义了两个简单的数据访问对象 (DAO)。一个 DAO 是一个提供持久化操作接口的对象。在这个应用中我们使用了对象关系映射 (ORM) 框架 Hibernate, DAO 通常在对象旁边定义。在本项目中, 我们定义两个 DAO 对象: WeatherDAO 和 LocationDAO。WeatherDAO 类允许我们保存一个 Weather 对象至数据库, 根据 id 获取一个 Weather 对象, 获取匹配特定 Location 的 Weather 对象。LocationDAO 有一个方法允许我们根据邮政编码获取 Location 对象。首先, 让我们看一下 [Example 7.8. “simple-persist 的 POM”](#) 中的 simple-persist POM。

Example 7.8. simple-persist 的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-model</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
```

```

<artifactId>hibernate</artifactId>
<version>3.2.5.ga</version>
<exclusions>
  <exclusion>
    <groupId>javax.transaction</groupId>
    <artifactId>jta</artifactId>
  </exclusion>
</exclusions>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>3.3.0.ga</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-commons-annotations</artifactId>
  <version>3.3.0.ga</version>
</dependency>
<dependency>
  <groupId>org.apache.geronimo.specs</groupId>
  <artifactId>geronimo-jta_1.1_spec</artifactId>
  <version>1.1</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.0.7</version>
</dependency>
</dependencies>
</project>

```

这个 POM 文件引用 simple-parent 作为一个父 POM，它定义了一些依赖。simple-persist 的 POM 中列出的依赖有：

org.sonatype.mavenbook.ch07:simple-model:1.0

就像 simple-weather 模块一样，这个持久化模块引用了 simple-model 中定义的核心模型对象。

org.hibernate:hibernate:3.2.5.ga

我们定义了一个对 Hibernate 版本 3.2.5ga 的依赖，但注意我们排除了 Hibernate 的一个依赖。这么做是因为 javax.transaction:javax 依赖在公共 Maven 仓库中不可用。此依赖正好是 Sun 依赖中的一个，不能免费在中央 Maven 仓库中提供。为了避免烦人的信息告诉我们去下载非免费的依赖，我们

简单的从 **Hibernate** 排除这个依赖然后添加一个 **geronimo-jta_1.1_spec** 依赖。

org.apache.geronimo.specs:geronimo-jta_1.1_spec:1.1

就像 **Servlet** 和 **JSP API**，**Apache Geronimo** 项目也根据 **Apache** 许可证友好的发布了一些认证过的企业级 **API**。这意味着不管什么时候某个组件告诉你它依赖于 **JDBC**，**JNDI**，和 **JTA API**，你都可以查一下 **groupId** 为 **org.apache.geronimo.specs** 下的对应类库。

org.springframework:spring:2.0.7

这里包含了整个 **Spring Framework** 作为一个依赖。

Note

只依赖于你正使用的 **Spring** 组件是一个很好的实践。**Spring Framework** 项目很友好的创建了一些有针对性的构件如 **spring-hibernate3**。

为什么依赖于 **Spring** 呢？当和 **Hibernate** 集成的时候，**Spring** 允许我们使用一些帮助类如 **HibernateDaoSupport**。作为一个 **HibernateDaoSupport** 的样例，看一下 [Example 7.9, “simple-persist”的 WeatherDAO 类](#) 中的 **WeatherDAO** 代码。

Example 7.9. simple-persist'的 WeatherDAO 类

```
package org.sonatype.mavenbook.weather.persist;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherDAO extends HibernateDaoSupport {

    public WeatherDAO() {}

    public void save(Weather weather) {
        getHibernateTemplate().save( weather );
    }

    public Weather load(Integer id) {
        return (Weather) getHibernateTemplate().load( Weather.class, id);
    }
}
```

```

@SuppressWarnings( "unchecked" )
public List<Weather> recentForLocation( final Location location ) {
    return (List<Weather>) getHibernateTemplate().execute(
        new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Query query = getSession().getNamedQuery( "Weather.byLocation" );
                query.setParameter( "location", location );
                return new ArrayList<Weather>( query.list() );
            }
        } );
}
}

```

就是这样。你已经编写了一个类，它能插入新的行，根据主键选取，以及能查找所有 **Weather** 表根据 **id** 连接 **Location** 表的结果。很显然，我们不能将书停在这里，然后花 500 页给你解释 **Hibernate** 的运作详情，但我们能做一些快速简单的解释：

- ❶ 继承 **HibernateDaoSupport** 的类。这个类会和 **Hibernate SessionFactory** 关联，后者将被用来创建 **Hibernate Session** 对象。在 **Hibernate** 中，每个操作都涉及 **Session** 对象，一个 **Session** 是访问下层数据库的中介，它也负责管理对 **DataSource** 的 **JDBC** 连接。继承 **HibernateDaoSupport** 也意味着我们能够使用 **getHibernateTemplate()** 访问 **HibernateTemplate**。能使用 **HibernateTemplate** 完成的操作例子有.....
- ❷ **save()** 方法接受一个 **Weather** 实例然后调用 **HibernateTemplate** 上的 **save()** 方法。
HibernateTemplate 简化了常见的 **Hibernate** 操作的调用，并别将所有数据库特有的异常转换成了运行时异常。这里我们调用 **save()**，它往 **Weather** 表中插入一条新的记录。可选的操作有 **update()**，它更新已存在的一行，或者 **saveOrUpdate()**，它会根据 **Weather** 中的 **non-null id** 属性是否存在，执行保存或者更新。
- ❸ **load()** 方法，同样，也只是调用 **HibernateTemplate** 实例的方法。**HibernateTemplate** 上的 **load()** 接受一个 **Class** 对象和一个 **Serializable** 对象。本例中，**Serializable** 对应于要载入的 **Weather** 对象的 **id** 的值。
- ❹ 最后一个方法 **recentForLocation()** 调用定义在 **Weather** 模型对象中的 **NamedQuery**。如果你的记忆力足够好，你就知道 **Weather** 模型对象定义了一个命名查询 "**Weather.byLocation**"，查询为 "**from Weather w where w.location = :location**"。我们通过 **HibernateCallback** 中的 **Hibernate Session** 对象来载入 **NamedQuery**，**HibernateCallback** 由 **HibernateTemplate** 的 **execute()** 方法执行。在这个方法中你能看到我们填充了一个命名参数 **location**，它来自于 **recentForLocation()** 方法的参数。

现在是时候阐明一些情况了。**HibernateDaoSupport** 和 **HibernateTemplate** 是来自于 **Spring Framework** 的类。它们由 **Spring Framework** 创建，目的是减少编写 **Hibernate DAO** 对象的痛苦。为了支持这个 **DAO**，我们需要在 **simple-persist** 的 **Spring ApplicationContext** 定义中做一些配置。 [Example 7.10. "simple-persist](#)

的 [Spring Application Context](#)"中显示的 XML 文档存储在 src/main/resources, 名为 applicationContext-persist.xml。

Example 7.10. simple-persist 的 Spring Application Context

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"
       default-lazy-init="true">

    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <property name="annotatedClasses">
            <list>

<value>org.sonatype.mavenbook.weather.model.Atmosphere</value>

<value>org.sonatype.mavenbook.weather.model.Condition</value>

<value>org.sonatype.mavenbook.weather.model.Location</value>
                <value>org.sonatype.mavenbook.weather.model.Weather</value>
                <value>org.sonatype.mavenbook.weather.model.Wind</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.show_sql">false</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.transaction.factory_class">
                    org.hibernate.transaction.JDBCTransactionFactory
                </prop>
                <prop key="hibernate.dialect">
                    org.hibernate.dialect.HSQLDialect
                </prop>
                <prop key="hibernate.connection.pool_size">0</prop>
                <prop key="hibernate.connection.driver_class">
                    org.hsqldb.jdbcDriver
                </prop>
                <prop key="hibernate.connection.url">
                    jdbc:hsqldb:data/weather;shutdown=true
                </prop>
                <prop key="hibernate.connection.username">sa</prop>
                <prop key="hibernate.connection.password"></prop>
            </props>
        </property>
    </bean>
</beans>
```

```

        <prop key="hibernate.connection.autocommit">true</prop>
        <prop key="hibernate.jdbc.batch_size">0</prop>
    </props>
</property>
</bean>

<bean id="locationDAO"
      class="org.sonatype.mavenbook.weather.persist.LocationDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="weatherDAO"
      class="org.sonatype.mavenbook.weather.persist.WeatherDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
</beans>

```

在这个 **application context** 中,我们完成了一些事情。**DAO** 从 **sessionFactory bean** 获取 **Hibernate Session** 对象。这个 **bean** 是一个 **AnnotationSessionFactoryBean** 的实例,并由一系列 **annotatedClasses** 填充。注意这列标注类就是定义在我们 **simple-model** 模块中的那些类。接下来, **sessionFactory** 通过一组 **Hibernate** 配置属性(**hibernateProperties**)配置。该例中,我们的 **Hibernate** 属性定义了许多设置:

hibernate.dialect

该设置控制如何生成数据库的 **SQL**。由于我们正在使用 **HSQLDB** 数据库,我们的数据库方言设置成 **org.hibernate.dialect.HSQLDialect**。**Hibernate** 有所有主流数据库的方言,如 **Oracle**, **MySQL**, **Postgres** 和 **SQL Server**。

hibernate.connection.*

该例中,我们从 **Spring** 配置中配置 **JDBC** 连接属性。我们的应用被配置成运行在 **./data/weather** 目录下的 **HSQLDB** 上。在实际的企业应用中,你更可能会使用如 **JNDI** 的东西以从你的应用程序代码中抽出数据库配置。

最后,在这个 **bean** 定义文件中,两个 **simple-persist DAO** 对象被创建并给予了对于刚定义的 **sessionFactory bean** 的引用。就像 **simple-weather** 中的 **Spring application context**,这个 **applicationContext-persist.xml** 文件定义了一个大型企业应用设计中一个子模块的架构。如果你曾经从事过大量持久化类的集合相关的工作,你可能会发现,这些与你应用程序独立的 **application context** 文件,能帮助你快速的理解所有类之间的关系。

simple-persist 中还有最后一块不清楚的地方。本章后面,我们将看到如何使用 **Maven Hibernate3** 插件,根据标注的模型对象来生成数据库 **schema**。为了使它正确工作, **Maven Hibernate3** 插件需要读取 **JDBC** 连接配置参数,那一列标注的类,以及 **src/main/resources** 中名为 **hibernate.cfg.xml** 文件的 **Hibernate** 配置。该文件(它重复了一些 **applicationContext-persist.xml** 中的配置)的目的是能让

Maven Hibernate3 插件能仅仅依靠标注就能生成数据定义语言（DDL）。如 [Example 7.11. “simple-persist 的 hibernate.cfg.xml”](#)。

Example 7.11. simple-persist 的 hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Database connection settings -->
    <property
name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:data/weather</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">>true</property>

    <!-- JDBC connection pool (use the built-in one) -->
    <property name="connection.pool_size">1</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">>true</property>

    <!-- disable batching so HSQLDB will propagate errors correctly. -->
    <property name="jdbc.batch_size">0</property>

    <!-- List all the mapping documents we're using -->
    <mapping class="org.sonatype.mavenbook.weather.model.Atmosphere"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Condition"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Location"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Weather"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Wind"/>
```

```
</session-factory>
</hibernate-configuration>
```

[Example 7.10, “simple-persist 的 Spring Application Context”](#)和[Example 7.1, “simple-parent 项目的 POM”](#)的内容是冗余的。Spring Application Context XML 是被 web 应用和命令行应用使用的，而 hibernate.cfg.xml 的存在只是为了支持 Maven Hibernate3 插件。本章的后面，我们将会看到如何使用 hibernate.cfg.xml 和 Maven Hibernate3 插件，根据 simple-model 中定义的标注对象模型，来生成一个数据库 schema。hibernate.cfg.xml 会配置 JDBC 连接属性并且为 Maven Hibernate3 插件枚举标注模型类的列表。

7.6. simple-webapp 模块



该 web 应用中项目 simple-webapp 中定义。这个简单 web 应用项目将会定义两个 Spring MVC 控制器：WeatherController 和 HistoryController。两者都会引用 simple-weather 和 simple-persist 中定义的组件。Spring 容器在应用程序的 web.xml 中配置，该文件引用了 simple-weather 中的 applicationContext-weather.xml 文件和 simple-persist 中的 applicationContext-persist.xml 文件。这个简单 web 应用的组件架构如[Figure 7.3, “Spring MVC 控制器引用 simple-weather 和 simple-persist 中的组件”](#)显示。

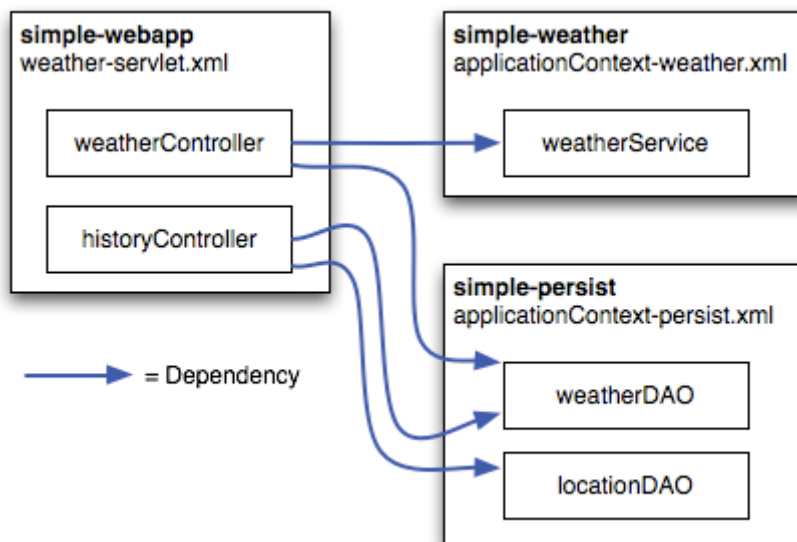


Figure 7.3. Spring MVC 控制器引用 simple-weather 和 simple-persist 中的组件

simple-webapp 的 POM 如[Example 7.12, “simple-webapp 的 POM”](#)显示。

Example 7.12. simple-webapp 的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Simple Web Application</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.ch07</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
```

```

<plugins>
  <plugin>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>maven-jetty-plugin</artifactId>
    <dependencies>
      <dependency>
        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
      </dependency>
    </dependencies>
  </plugin>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>hibernate3-maven-plugin</artifactId>
    <version>2.0</version>
    <configuration>
      <components>
        <component>
          <name>hbm2ddl</name>
          <implementation>annotationconfiguration</implementation>
        </component>
      </components>
    </configuration>
    <dependencies>
      <dependency>
        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
      </dependency>
    </dependencies>
  </plugin>
</plugins>
</build>
</project>

```

随着书本的推进以及样例变得越来越大，你会注意到 pom.xml 开始呈现得有一些笨重，这里我们配置了四个依赖和两个插件。让我们详细查看一下这个 POM 然后详述其中一些重要的配置点：

- ❶ simple-webapp 项目定义了四个依赖：来自于 Apache Geronimo 的 Servlet 2.4 规格说明实现，simple-weather 服务类库，simple-persist 持久化类库，以及整个 Spring Framework 2.0.7。

- ② **Maven Jetty** 插件以最简单的方式加入到该项目，我们只是添加一个引用了对应 `groupId` 和 `artifactId` 的 `plugin` 元素。配置这个插件如此平常意味着这个插件的开发者做了很好的工作提供了足够的默认值，在大部分情况下不需要被重写。如果你需要重写一些 **Jetty** 插件的配置，那么就需要提供 `configuration` 元素。
- ③ 在我们的 **build** 配置中，我们还配置了 **Maven Hibernate3** 插件来访问内嵌的 **HSQLDB** 实例。要让 **Maven Hibernate3** 插件能成功的使用 **JDBC** 连接该数据库，该插件需要引用 `classpath` 中的 **HSQLDB JDBC** 驱动。为了让这个插件能使用该依赖，我们在 `plugin` 声明下面添加了一个 `dependency` 声明。在该例中，我们引用了 `hsqldb:hsqldb:1.8.0.7`。这个 **Hibernate** 插件也需要 **JDBC** 驱动来创建数据库，所以我们也它在配置中添加了 this 依赖。
- ④ 这个 **Maven Hibernate** 插件正是该 **POM** 变得有趣的地方。在下一节，我们将会运行 `hbm2ddl` 目标来生成 **HSQLDB** 数据库。在这个 `pom.xml` 中，我们包含了对 `hibernate3-maven-plugin` 版本 2.0 的引用，该插件由 **Codehaus Mojo** 维护。
- ⑤ **Maven Hibernate3** 插件有不同的方法获取 **Hibernate** 映射信息，这些信息适用于 **Hibernate3** 插件的不同用例。如果你正在使用 **Hibernate** 映射 **XML** 文件 (`.hbm.xml`)，你会要使用 `hbm2java` 目标生成模型类，你会将 `implementation` 设置成 `configuration`。如果你使用 **Hibernate3** 插件逆向工程从一个数据库产生 `.hbm.xml` 文件和模型类，你会需要一个 `jdbccconfiguration` 的 `implementation`。在本例中，我们使用现存的标注对象模型来生成一个数据库。换句话说，我们有我们的 **Hibernate** 映射，但我们还没有数据库。在这种用例中，正确的 `implementation` 值应该是 `annotationconfiguration`。**Maven Hibernate3** 插件在后面一节 [Section 7.7, “运行这个 Web 应用”](#) 中详细讨论。

Note

一个常见的错误是使用 `extensions` 配置添加一个插件需要的依赖。这是强烈不推荐的因为 `extensions` 会在你的项目中造成 `classpath` 污染，以及其它令人讨厌的副作用。此外，`extensions` 行为正在 2.1 中被重做，最后你都会要改变它。唯一的对 `extensions` 的正常使用是定义新的 `wagon` 实现。

接下来，我们将我们的注意力转移到两个处理所有请求的 **Spring MVC** 控制器。两个控制器都引用了在 `simple-weather` 和 `simple-persist` 中定义的 `bean`。

Example 7.13. simple-webapp WeatherController

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;
import org.sonatype.mavenbook.weather.WeatherService;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class WeatherController implements Controller {

    private WeatherService weatherService;
```

```

private WeatherDAO weatherDAO;

public ModelAndView handleRequest(HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    String zip = request.getParameter("zip");
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save(weather);
    return new ModelAndView("weather", "weather", weather);
}

public WeatherService getWeatherService() {
    return weatherService;
}

public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
}

public WeatherDAO getWeatherDAO() {
    return weatherDAO;
}

public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
}
}

```

WeatherController 实现了 **MVC Controller** 接口，该接口强制要求实现如上例中的 `handleRequest()` 方法。如果你看一下该方法的主要内容，你会看到它调用了 `weatherService` 实例变量的 `retrieveForecast()` 方法。不像前面的章节中，有一个 **Servlet** 来初始化 `WeatherService` 类，`WeatherController` 是一个带有 `weatherService` 属性的 **bean**。**Spring Ioc** 容器会负责将 `weatherService` 组件注入到控制器。同时也注意我们并没有在这个控制器实现中使用 `WeatherFormatter`；而是将 `retrieveForecast()` 返回的 `Weather` 对象传递给了 `ModelAndView` 的构造函数。`ModelAndView` 类将被用来呈现 **Velocity** 模板，这个模板有对 `{weather}` 变量的引用。`weather.vm` 模板存储在 `src/main/webapp/WEB-INF/vm`，如[???](#)所示。

在这个 `WeatherController` 中，在我们呈现天气预报输出之前，我们将 `WeatherService` 返回的 `Weather` 对象传递给 `WeatherDAO` 的 `save()` 方法。这里我们使用 **Hibernate** 将 `Weather` 对象保存到 **HSQldb** 数据库。之后，在 `HistoryController` 中，我们将看如何能够获取由 `WeatherController` 保存的天气预报历史信息。

Example 7.14. 由 WeatherController 呈现的 weather.vm 模板

```
<b>Current Weather Conditions for:
    ${weather.location.city}, ${weather.location.region},
    ${weather.location.country}</b><br/>

<ul>
    <li>Temperature: ${weather.condition.temp}</li>
    <li>Condition: ${weather.condition.text}</li>
    <li>Humidity: ${weather.atmosphere.humidity}</li>
    <li>Wind Chill: ${weather.wind.chill}</li>
    <li>Date: ${weather.date}</li>
</ul>
```

Velocity 模板的语法简单易懂，变量通过`${}`标记引用。大括弧里面的表达式引用 weather 变量的一个属性，或者该变量属性的属性。weather 变量是由 WeatherController 传递给该模板的。

HistoryController 用来获取那些已经由 WeatherController 请求过的最近历史天气预报信息。任何时候当我们从 WeatherController 获取预报的时候，该控制器通过 WeatherDAO 将 Weather 对象保存至数据库。WeatherDAO 然后使用 Hibernate 将 Weather 对象剖析成一组相关数据库表的记录行。HistoryController 如 [Example 7.15, “simple-web 的 HistoryController”](#)所示。

Example 7.15. simple-web 的 HistoryController

```
package org.sonatype.mavenbook.web;

import java.util.*;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.sonatype.mavenbook.weather.model.*;
import org.sonatype.mavenbook.weather.persist.*;

public class HistoryController implements Controller {

    private LocationDAO locationDAO;
    private WeatherDAO weatherDAO;

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        String zip = request.getParameter("zip");
        Location location = locationDAO.findByZip(zip);
        List<Weather> weathers = weatherDAO.recentForLocation( location );
    }
}
```

```

    Map<String, Object> model = new HashMap<String, Object>();
    model.put( "location", location );
    model.put( "weathers", weathers );

    return new ModelAndView( "history", model );
}

public WeatherDAO getWeatherDAO() {
    return weatherDAO;
}

public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
}

public LocationDAO getLocationDAO() {
    return locationDAO;
}

public void setLocationDAO(LocationDAO locationDAO) {
    this.locationDAO = locationDAO;
}
}

```

HistoryController 被注入了两个定义在 simple-persist 中的 DAO 对象。这两个 DAO 是 HistoryController 的 bean 属性：WeatherDAO 和 LocationDAO。HistoryController 的目标是获取一个与 zip 参数对应的 Weather 对象列表。当 WeatherDAO 将 Weather 对象保存至数据库，它不只是保存邮编，它保存了一个与 simple-model 中 Weather 对象相关的 Location 对象。为了获取一个 Weather 对象的 List，首先通过调用 LocationDAO 的 findByZip() 方法，获取与 zip 参数对应的 Location 对象。

一旦获得了 Location 对象，HistoryController 之后就会尝试获取与给定的 Location 相匹配的 Weather 对象。在获取了 List<Weather>之后，一个 HashMap 被创建以存储两个变量，供如[???中](#)显示的 history.vm Velocity 模板使用。

Example 7.16. 由 HistoryController 呈现的 history.vm

```

<b>
Weather History for: ${location.city}, ${location.region},
${location.country}
</b>
<br/>

#foreach( $weather in $weathers )
    <ul>

```

```

    <li>Temperature: $weather.condition.temp</li>
    <li>Condition: $weather.condition.text</li>
    <li>Humidity: $weather.atmosphere.humidity</li>
    <li>Wind Chill: $weather.wind.chill</li>
    <li>Date: $weather.date</li>
  </ul>
#end

```

src/main/webapp/WEB-INF/vm 中的 history.vm 模板引用了 location 变量以输出天气预报位置的相关信息。该模板使用了一个 **Velocity** 的控制结构,为了循环 weathers 变量中的每个元素。weathers 中的每个元素被赋给了一个名为 weather 的变量, #foreach 和 #end 中间的模板用来呈现每个预报输出。

你已经看到了这些 Controller 实现,以及它们如何引用定义在 simple-weather 和 simple-persist 中的其它 **bean**,它们相应 HTTP 请求,让那些知道如何呈现 **Velocity** 模板的神奇的模板系统来控制输出。所有的魔法都在位于 src/main/webapp/WEB-INF/weather-servlet.xml 的 **Spring application context** 中配置。这个 XML 文件配置了控制器并引用了其它 **Spring** 管理的 **bean**,它由 ServletContextListener 载入,后者同时也被配置从 classpath 中载入了 applicationContext-weather.xml 和 applicationContext-persist.xml。让我们仔细看一下[???](#)中展示的 weather-servlet.xml。

Example 7.17. weather-servlet.xml 中的 **Spring** 控制器配置

```

<beans>
  <bean id="weatherController"
        class="org.sonatype.mavenbook.web.WeatherController">
    <property name="weatherService" ref="weatherService"/>
    <property name="weatherDAO" ref="weatherDAO"/>
  </bean>

  <bean id="historyController"
        class="org.sonatype.mavenbook.web.HistoryController">
    <property name="weatherDAO" ref="weatherDAO"/>
    <property name="locationDAO" ref="locationDAO"/>
  </bean>

  <!-- you can have more than one handler defined -->
  <bean id="urlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
      <map>
        <entry key="/weather.x">
          <ref bean="weatherController" />
        </entry>
      </map>
    </property>
  </bean>

```

```

        <entry key="/history.x">
            <ref bean="historyController" />
        </entry>
    </map>
</property>
</bean>

<bean id="velocityConfig"
class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
    <property name="resourceLoaderPath" value="/WEB-INF/vm/" />
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
    <property name="cache" value="true" />
    <property name="prefix" value="" />
    <property name="suffix" value=".vm" />
    <property name="exposeSpringMacroHelpers" value="true" />
</bean>
</beans>

```

- ❶ weather-servlet.xml 定义了两个控制器作为 **Spring** 管理的 **bean**。weatherController 有两个属性，引用 weatherService 和 weatherDAO。historyController 引用了 weatherDAO 和 locationDAO **bean**。当 **ApplicationContext** 被创建的时候，它所处的环境能够访问 simple-persist 和 simple-weather 中定义的 **ApplicationContext**。在[???](#)中你将看到如何配置 **Spring** 以归并多个 **Spring** 配置文件的组件。
- ❷ urlMapping **bean** 定义了调用 WeatherController 和 HistoryController 的 **URL** 模式。该例中，我们使用 SimpleUrlHandlerMapping，将 /weather.x 映射到 WeatherController，将 /history.x 映射到 HistoryController。
- ❸ 由于我们正使用 **Velocity** 模板引擎，我们需要传入一些配置选项。在 velocityConfig **bean** 中，我们告诉 **Velocity** 从 /WEB-INF/vm 目录中寻找所有的模板。
- ❹ 最后，viewResolver 使用 VelocityViewResolver 类配置。**Spring** 中有很多 viewResolver 实现，从用来呈现 **JSP** 或者 **JSTL** 页面的标准 viewResolver，到用来呈现 **Freemarker** 模板的 viewResolver。本例中，我们配置 **Velocity** 模板引擎，设置默认的前缀和后缀，它们将被自动附加到那些传给 ModelAndView 的模板名前后。

最后，simple-webapp 项目中有一个 web.xml，提供了这个 **web** 应用的基本配置。web.xml 文件如[???](#)所示：

Example 7.18. simple-webapp 的 web.xml

```
<web-app id="simple-webapp" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>Simple Web Application</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:applicationContext-weather.xml
      classpath:applicationContext-persist.xml
    </param-value>
  </context-param>

  <context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>/WEB-INF/log4j.properties</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.util.Log4jConfigListener
    </listener-class>
  </listener>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>*.x</url-pattern>
```

```
</servlet-mapping>
</web-app>
```

- ❶ 这里有一些魔法能让我们在项目中重用 applicationContext-weather.xml 和 applicationContext-persist.xml。contextConfigLocation 由 ContextLoaderListener 用来创建一个 ApplicationContext。当一个 **weather servlet** 被创建的时候, 222 中的 weather-servlet.xml 将由此 contextConfigLocation 中创建的 ApplicationContext 赋值。用这种方式, 你可以在另外的项目中定义一组 **bean**, 然后可以通过 **classpath** 引用这些 **bean**。由于 simple-persist 和 simple-weather 的 **JAR** 将会位于 WEB-INF/lib, 我们所要做的只是使用 classpath: 前缀来引用这些文件。(另一种选择是将所有这些文件拷贝到 /WEB-INF, 然后用过如 /WEB-INF/applicationContext-persist.xml 的方式引用它们)。
- ❷ log4jConfigLocation 用来告诉 Log4JConfigListener 哪里去寻找 **Log4J** 日志配置。该例中, 我们告诉 **Log4J** 在 /WEB-INF/log4j.properties 中寻找。
- ❸ 这里确保当 **web** 应用启动的时候 **Log4J** 系统被配置。将 Log4JConfigListener 放在 ContextLoaderListener 前面十分重要; 否则你可能丢失那些指向阻止应用启动问题的重要日志信息。如果你有一个特别大的 **Spring** 管理的 **bean** 集合, 而其中一个碰巧在应用启动的时候出问题了, 应用很可能不能启动。如果你在 **Spring** 启动之前有了日志, 你就有机会看到警告或错误信息。如果你在 **Spring** 启动之前没有配置日志, 你就不知道为什么你的应用不能启动了。
- ❹ ContextLoaderListener 本质上是一个 **Spring** 容器。当应用启动的时候, 这个监听器会根据 contextConfigLocation 参数构造一个 ApplicationContext。
- ❺ 我们定义一个名为 weather 的 **Spring MVC DispatcherServlet**。这会让 **Spring** 从 /WEB-INF/weather-servlet.xml 寻找 **Spring** 配置文件。你可以拥有任意多的 DispatcherServlet, 一个 DispatcherServlet 可以包含一个或多个 **Spring MVC Controller** 实现。
- ❻ 所有以 .x 结尾的请求都会被路由至 weather servlet。注意 .x 扩展名没有任何特殊的意义, 这是一个随意的选择, 你可以使用任意你喜欢的 **URL** 模式。

7.7. 运行这个 Web 应用



为了运行这个 **web** 应用, 你首先需要使用 **Hibernate3** 插件构造数据库。为此, 在项目 simple-webapp 目录下运行如下命令:

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO]
```

```

[INFO] Building Chapter 7 Simple Web Application
[INFO]     task-segment: [hibernate3:hbm2ddl]
[INFO]
-----

[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151 INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO]
-----

[INFO] BUILD SUCCESSFUL
[INFO]
-----

```

在此之后，应该有一个`${basedir}/data` 目录包含了 **HSQldb** 数据库。你可以使用 **jetty** 启动 **web** 应用：

```

$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO]
-----

[INFO] Building Chapter 7 Simple Web Application
[INFO]     task-segment: [jetty:run]
[INFO]
-----

[INFO] Preparing jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: Chapter 7 Simple Web Application
...
[INFO] Context path = /simple-webapp
[INFO] Tmp directory = determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] Starting jetty 6.1.7 ...
2008-03-25 10:28:03.639::INFO: jetty-6.1.7
...
2147 INFO DispatcherServlet - FrameworkServlet 'weather': \
        initialization completed in 1654 ms
2008-03-25 10:28:06.341::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server

```

Jetty 启动之后，你可以载入

<http://localhost:8080/simple-webapp/weather.x?zip=60202>，然后就能在你的浏览器中看到 **Evanston, IL** 的天气。更改邮编后你就能看到你自己的天气报告了。

Current Weather Conditions for: Evanston, IL, US

- * Temperature: 42
- * Condition: Partly Cloudy
- * Humidity: 55
- * Wind Chill: 34
- * Date: Tue Mar 25 10:29:45 CDT 2008

7.8. simple-command 模块



simple-command 项目是 simple-webapp 的一个命令行版本。这个命令行工具有这同样的模块依赖: simple-persist 和 simple-weather。现在你需要从命令行运行这个 simple-command 工具, 而非通过 web 浏览器与该应用交互。

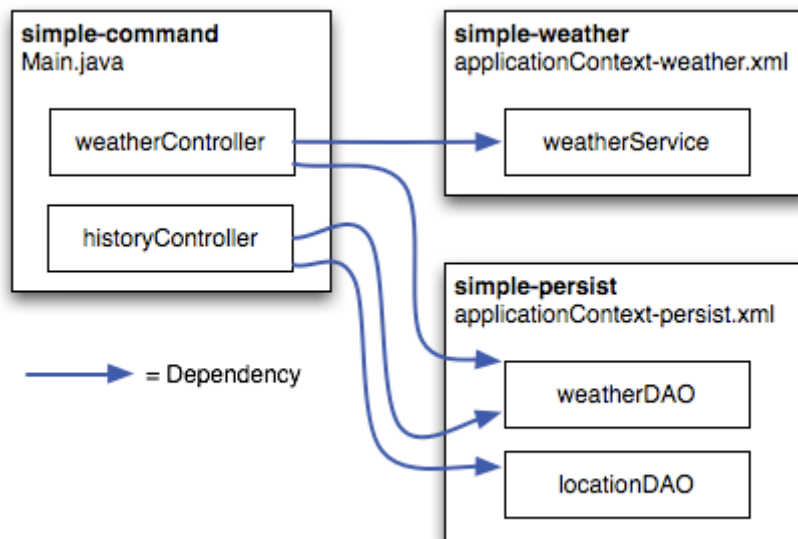


Figure 7.4. 引用 simple-weather 和 simple-persist 的命令行应用

Example 7.19. simple-command 的 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch07</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
```

```
<artifactId>simple-command</artifactId>
<packaging>jar</packaging>
<name>Simple Command Line Tool</name>

<build>
  <finalName>${project.artifactId}</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>hibernate3-maven-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <components>
          <component>
            <name>hbm2ddl</name>
            <implementation>annotationconfiguration</implementation>
          </component>
        </components>
      </configuration>
      <dependencies>
        <dependency>
```

```

        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
    </dependency>
</dependencies>
</plugin>
</plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.sonatype.mavenbook.ch07</groupId>
        <artifactId>simple-weather</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>org.sonatype.mavenbook.ch07</groupId>
        <artifactId>simple-persist</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.0.7</version>
    </dependency>
    <dependency>
        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
    </dependency>
</dependencies>
</project>

```

这个 POM 创建一个包含了如 [Example 7.20, “simple-command 的 Main 类”](#) 所示的 `org.sonatype.mavenbook.weather.Main` 类的 JAR 文件。在这个 POM 中我们配置了 **Maven Assembly** 插件使用内置的名为 `jar-with-dependencies` 的装配描述符来创建一个 JAR 文件，该文件包含了运行应用所需要的所有二进制代码，包括项目本身的字节码以及所有依赖文件的字节码。

Example 7.20. simple-command 的 Main 类

```
package org.sonatype.mavenbook.weather;

import java.util.List;

import org.apache.log4j.PropertyConfigurator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.LocationDAO;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;

public class Main {

    private WeatherService weatherService;
    private WeatherDAO weatherDAO;
    private LocationDAO locationDAO;

    public static void main(String[] args) throws Exception {
        // Configure Log4J
        PropertyConfigurator.configure(Main.class.getClassLoader().getResource(
            "log4j.properties"));

        // Read the Zip Code from the Command-line (if none supplied, use 60202)
        String zipcode = "60202";
        try {
            zipcode = args[0];
        } catch (Exception e) {
        }

        // Read the Operation from the Command-line (if none supplied use weather)
        String operation = "weather";
        try {
            operation = args[1];
        } catch (Exception e) {
        }

        // Start the program
        Main main = new Main(zipcode);

        ApplicationContext context =
            new ClassPathXmlApplicationContext(
```

```

        new String[] { "classpath:applicationContext-weather.xml",
                        "classpath:applicationContext-persist.xml" });
    main.weatherService = (WeatherService)
context.getBean("weatherService");
    main.locationDAO = (LocationDAO) context.getBean("locationDAO");
    main.weatherDAO = (WeatherDAO) context.getBean("weatherDAO");
    if( operation.equals("weather")) {
        main.getWeather();
    } else {
        main.getHistory();
    }
}

private String zip;

public Main(String zip) {
    this.zip = zip;
}

public void getWeather() throws Exception {
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save( weather );
    System.out.print(new WeatherFormatter().formatWeather(weather));
}

public void getHistory() throws Exception {
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation(location);
    System.out.print(new WeatherFormatter().formatHistory(location,
weathers));
}
}

```

这个 Main 类有对于 WeatherDAO, LocationDAO, 以及 WeatherService 的引用。该类的静态 main() 方法:

- 从第一个命令行参数读取邮编。
- 从第二个命令行参数读取操作。如果操作是“weather”，最新的天气将会从 web 服务获得。如果操作是“history”，该程序会从本地数据库获取历史天气记录。
- 根据来自于 simple-persist 和 simple-weather 的两个 XML 文件载入 Spring ApplicationContext。
- 创建一个 Main 的实例。

- 使用来自于 **Spring** ApplicationContext 的 **bean** 填充 weatherService, weatherDAO, 和 locationDAO。
- 根据特定的操作运行相应的 getWeather() 或者 getHistory() 方法。

在 web 应用中我们使用 **Spring** VelocityViewResolver 来呈现一个 **Velocity** 模板。在这个单机实现中给我们需要编写一个简单的类来通过 **Velocity** 模板呈现天气数据。

[Example 7.21, “WeatherFormatter 使用 Velocity 模板呈现天气数据”](#)是 WeatherFormatter 的代码清单, 这个类有两个方法来呈现天气报告和天气历史信息。

Example 7.21. WeatherFormatter 使用 Velocity 模板呈现天气数据

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;
import java.util.List;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String formatWeather( Weather weather ) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                                   getResourceAsStream( "weather.vm" ) );
        VelocityContext context = new VelocityContext();
        context.put( "weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }

    public String formatHistory( Location location, List<Weather> weathers )
        throws Exception {
        log.info( "Formatting History Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
```

```

        getResourceAsStream( "history.vm" );
    VelocityContext context = new VelocityContext();
    context.put( "location", location );
    context.put( "weathers", weathers );
    StringWriter writer = new StringWriter();
    Velocity.evaluate(context, writer, "", reader);
    return writer.toString();
}
}

```

weather.vm 模板简单的打印邮编对应的城市, 国家, 区域以及当前的气温。history.vm 模板打印位置信息并遍历存储在本地数据库中的天气预报记录。两者都位于 `${basedir}/src/main/resources`。

Example 7.22. weather.vm Velocity 模板

```

*****
Current Weather Conditions for:
    ${weather.location.city},
    ${weather.location.region},
    ${weather.location.country}
*****

* Temperature: ${weather.condition.temp}
* Condition: ${weather.condition.text}
* Humidity: ${weather.atmosphere.humidity}
* Wind Chill: ${weather.wind.chill}
* Date: ${weather.date}

```

Example 7.23. history.vm Velocity 模板

```

Weather History for:
${location.city},
${location.region},
${location.country}

#foreach( $weather in $weathers )
*****
    * Temperature: $weather.condition.temp
    * Condition: $weather.condition.text
    * Humidity: $weather.atmosphere.humidity
    * Wind Chill: $weather.wind.chill
    * Date: $weather.date
#end

```

7.9. 运行这个命令程序



simple-command 项目被配置成创建一个单独的包含项目本身字节码以及所有依赖字节码的 JAR 文件。要创建这个装配制品，从 simple-command 项目目录运行 **Maven Assembly** 插件的 assembly 目标：

```
$ mvn assembly:assembly
[INFO]
-----
[INFO] Building Chapter 7 Simple Command Line Tool
[INFO]    task-segment: [assembly:assembly] (aggregator-style)
[INFO]
-----
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
...
[INFO] [jar:jar]
[INFO] Building
jar: .../simple-parent/simple-command/target/simple-command.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: .../.m2/repository/.../simple-weather-1-SNAPSHOT.jar into \
/tmp/archived-file-set.93251505.tmp
[INFO] Expanding: .../.m2/repository/.../simple-model-1-SNAPSHOT.jar into \
/tmp/archived-file-set.2012480870.tmp
[INFO] Expanding: .../.m2/repository/.../hibernate-3.2.5.ga.jar into \
/tmp/archived-file-set.1296516202.tmp
... skipping 25 lines of dependency unpacking ...
[INFO] Expanding: .../.m2/repository/.../velocity-1.5.jar into
/tmp/archived-file-set.379482226.tmp
[INFO] Expanding: .../.m2/repository/.../commons-lang-2.1.jar into \
```

```
/tmp/archived-file-set.1329200163.tmp
[INFO] Expanding: .../.m2/repository/.../oro-2.0.8.jar into
/tmp/archived-file-set.1993155327.tmp
[INFO] Building
jar: .../simple-parent/simple-command/target/simple-command-jar-with-dependencies.jar
```

构建过程经过了生命周期中编译字节码，运行测试，然后最终为该项目构建一个 **JAR**。然后 `assembly:assembly` 目标创建一个带有依赖的 **JAR**，它将所有依赖解压到一个临时目录，然后将所有字节码收集到 `target/` 目录下一个名为 `simple-command-jar-with-dependencies.jar` 的 **JAR** 中。这个“超级的”**JAR** 的体重为 **15MB**。

在你运行这个命令行工具之前，你需要调用 **Hibernate3** 插件的 `hbm2ddl` 目标来创建 **HSQldb** 数据库。在 `simple-command` 目录运行如下的命令：

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO]

-----

[INFO] Building Chapter 7 Simple Command Line Tool
[INFO]    task-segment: [hibernate3:hbm2ddl]
[INFO]

-----

[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151 INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO]

-----

[INFO] BUILD SUCCESSFUL
[INFO]

-----
```

在此之后，你应该能在 `simple-command` 下看到 `data/` 目录。该 `data/` 目录包含了 **HSQldb** 数据库。要运行这个命令行天气预报器，在 `simple-command/` 项目目录下运行如下命令：

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
      org.sonatype.mavenbook.weather.Main 60202
2321 INFO YahooRetriever - Retrieving Weather Data
2489 INFO YahooParser - Creating XML Reader
2581 INFO YahooParser - Parsing XML Response
2875 INFO WeatherFormatter - Formatting Weather Data
*****
```

Current Weather Conditions for:

Evanston,
IL,
US

- * Temperature: 75
- * Condition: Partly Cloudy
- * Humidity: 64
- * Wind Chill: 75
- * Date: Wed Aug 06 09:35:30 CDT 2008

要运行历史查询，运行如下命令：

```
$ java -cp target/simple-command-jar-with-dependencies.jar \
    org.sonatype.mavenbook.weather.Main 60202 history
```

2470 INFO WeatherFormatter - Formatting History Data

Weather History for:

Evanston, IL, US

- * Temperature: 39
- * Condition: Heavy Rain
- * Humidity: 93
- * Wind Chill: 36
- * Date: 2007-12-02 13:45:27.187

- * Temperature: 75
- * Condition: Partly Cloudy
- * Humidity: 64
- * Wind Chill: 75
- * Date: 2008-08-06 09:24:11.725

- * Temperature: 75
- * Condition: Partly Cloudy
- * Humidity: 64
- * Wind Chill: 75
- * Date: 2008-08-06 09:27:28.475

7.10. 小结



到目前为止我们花了大量时间在一些不是和 **Maven** 直接相关的话题上。这么做是为了演示一些完整的并且有意义的样例项目，以能让你用来帮助实现你的现实系统。我们并没有走任何捷径来快速生成完好的结果，也没有使用 **Ruby on Rails** 式样的东西让你感到惊讶目眩，说你可以在“简单的 10 分钟内！”创建并完成一个 **Java** 企业级应用。市场上这样的东西太多了，有太多的人试图卖给你最简单的框架，又只需要你投入零的时间关注。本章我们想要做的是展现给你整个的画面，整个多模块构建的生态系统。这里我们展现的 **Maven** 处于一个你能经常看到的自然应用的上下文中——不是快餐式的，10 分钟的简单介绍，往 **Apache Ant** 扔烂泥，说服你采用 **Apache Maven**。

如果你离开本章，想知道它到底和 **Maven** 有什么关系，那么我们就成功了。我们演示了一个复杂的项目集合，使用了流行的框架，并且使用声明式构建将它们绑在了一起。事实上本章 60% 以上的内容是在解释 **Spring** 和 **Hibernate**，而大部分时间，**Maven** 暂时离开了。这样是可行的。它让你集中注意力于应用本身，而非构建过程。我们专门的来讨论那些在这个人造项目中用到的技术，而不是花时间讨论 **Maven**，以及那些为了“构建一个集成了 **Spring** 和 **Hibernate** 的构建”你必须做的工作。如果你开始使用 **Maven**，并且花时间学习它，你肯定会开始获益，因为你不需要花时间去编写一些程式化的构建脚本。你不用再花时间去考虑构建中那些平常的方面。

你可以使用本章介绍的骨架项目作为你自己项目的基础，这么做的机会是，你会发现，你会根据需要创建越来越多的模块。例如，基于本章样例的项目可能有两个单独的模型模块，两个持久化模块以将数据持久化到不同数据库，一些 **web** 应用，以及一个 **Java** 手机应用。总的来说，这个现实系统包含了 15 个相关的模块。重点是，你已经看到了本书中最复杂的多模块样例，但你应该知道该样例也仅仅触及了所有 **Maven** 可能性的表面。

7.10.1. 编写接口项目程序



本章展现了一个多模块项目，该项目比 [Chapter 6, 一个多模块项目](#) 中展示的简单样例复杂得多，然而它也只是现实项目的一个简化。在大型项目中，你可能发现你正构建一个如 [Figure 7.5, “编写接口项目程序”](#) 的系统。

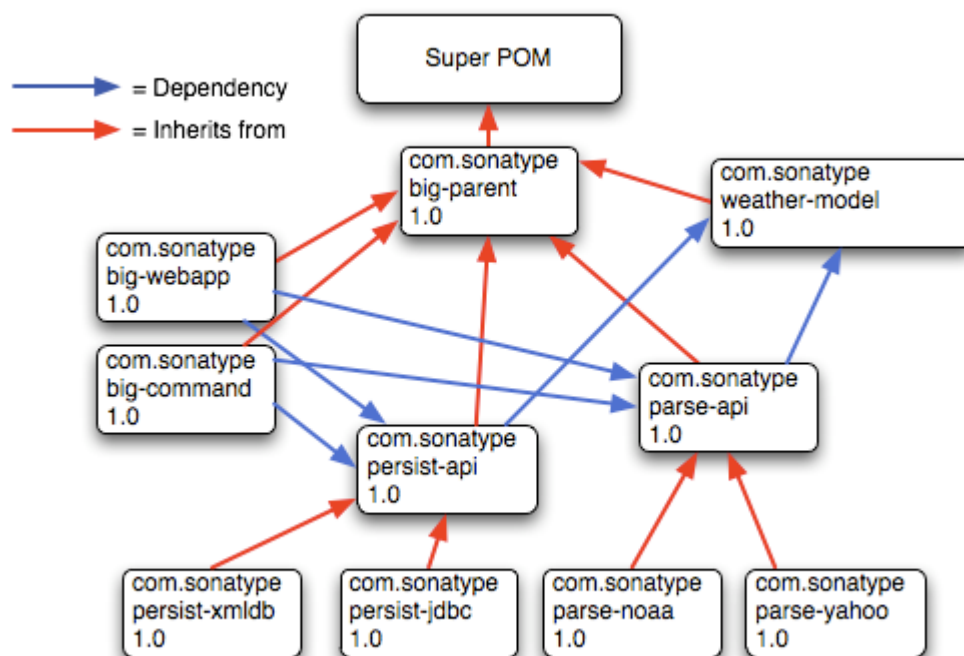


Figure 7.5. 编写接口项目程序

当我们使用术语接口项目的时候，我们是指一个只包含了接口和常量的 **Maven** 项目。在 [Figure 7.5, “编写接口项目程序”](#) 中，接口项目是 `persist-api` 和 `parse-api`。如果 `big-command` 和 `big-webapp` 针对于 `persist-api` 中定义的接口编写，那么它就能很容易的切换到另一种持久化类库的实现。该图显示了两个 `persist-api` 项目的实现，一个将数据存储于 **XML** 数据库，另一个将数据存储于关系数据库。如果你使用本章中的一些概念，你就能看到如何仅仅通过传入一个标记，让程序换入一个不同的 **Spring application context XML** 文件，换出持久化实现的数据源。就像应用程序本身的 **OO** 设计一样，通常将接口 **Maven** 项目从实现 **Maven** 项目中分离是一种很明智的做法。

Chapter 8. 优化和重构 POM

[8.1. 简介](#)

[8.2. POM 清理](#)

[8.3. 优化依赖](#)

[8.4. 优化插件](#)

[8.5. 使用 Maven Dependency 插件进行优化](#)

[8.6. 最终的 POM](#)

[8.7. 小结](#)

8.1. 简介



在 [Chapter 7, 多模块企业级项目](#) 中，我们展示了很多 Maven POM 文件结合起来生成一个功能齐全的多模块构建。虽然那一章的样例想要表现一个真实的应用——与数据库交互，与 web 服务交互，本身又暴露两个接口：一个 web 应用，还有一个命令行；但这毕竟是我们杜撰出来的样例。要表现一个真实项目的复杂度可远远超出了你正阅读的本书的能力范围。真实生活中的项目往往经历了很多年的发展，由很多关注点不同的不同组的开发者维护。现实项目中，你通常会评价由其它人所做的决定或者设计。本章，我们回头看一下 [Part I, “Maven 实战”](#) 中的样例，问问我们自己，基于我们对 Maven 的了解，能否对该样例做更合理的优化。Maven 很强大，能根据你的需要变得很简单或者很复杂。因此，通常完成同样一个任务有很多种方法，而且通常没有一个“最正确”的方式来配置你的 Maven 项目。

别误解最后一句话，认为它批准你可以要求 Maven 做一些本非它设计目的的事情。虽然 Maven 允许多样的方式，但当然也有“一种 Maven 的方式”，如此使用 Maven 你将能够有更高的生产效率，因为 Maven 本身就是被设计成这么用的。本章要做的就是传达一些能在已有项目上进行的优化手段。为什么我们不在一开始就介绍一个优化的 POM 呢？为教育而设计的 POM 与为了效率而设计的 POM 有着不同的需求。虽然在你的 ~/.m2/settings.xml 中定义一些设置比在 pom.xml 中声明一个 profile 简单得多，但写书及读书主要的还是一步一步来，确保没有在你没准备的好情况下就介绍新的概念。在 [Part I, “Maven 实战”](#) 中，我们也花力气避免让太多的信息淹没读者，这样，我们就跳过了一些很重要的概念，如将在本章介绍的 dependencyManagement。

在 [Part I, “Maven 实战”](#) 中有一些例子，本书的作者通过捷径掩盖了一些重要的细节，以带你关注那些特定章节的重点。你学习了如何创建一个 Maven 项目，不用去辛苦的阅读几百页的所有原理介绍，就能编译并安装它。这么做是因为我们相信对于新的 Maven 用户来说，快速的呈现结果比在一个持续过长的故事中迂回更重要。在你开始使用 Maven 之后，你应该知道如何分析项目和的 POM。本章，我们回头看看 [Chapter 7, 多模块企业级项目](#) 中所留下的样例。

8.2. POM 清理



优化一个多模块项目的 **POM** 最好通过几步来做，因为我们需要关注很多区域。总的来说，我们是寻找一个 **POM** 中的重复，或者多个兄弟 **POM** 中的重复。当你开始一个项目，或者项目进化得很快，有一些依赖和插件的重复是可以接受的，但随着项目的成熟以及模块的增多，你会需要花一些时间来重构共同的依赖和配置点。随着项目的变大，使你的 **POM** 更高效能很大程度的帮助你管理复杂度。不管什么时候遇到一些重复的信息片段，通常都有更好的配置方式。

8.3. 优化依赖



如果你仔细看一下 [Chapter 7, 多模块企业级项目](#) 中创建的不同 **POM**，就会注意到几种重复模式。我们能看到的第一种模式是：一些依赖如 **spring** 和 **hibernate-annotations** 在多个模块中被声明。每个 **hibernate** 依赖都重复排除了 **javax.transaction**。第二种重复模式是：有一些依赖是关联的，共享同样的版本。这种情况通常是因为某个项目的发布版本中包含了多个紧密关联的组件。例如，看一下依赖 **hibernate-annotations** 和 **hibernate-commons-annotations**，两者的版本都是 **3.3.0.ga**，而且我们可以预料这两个依赖的版本只会一起向前改变。

hibernate-annotations 和 **hibernate-commons-annotations** 都是 **JBoss** 发布的同一个项目的组件，当有新的版本发布的时候，两个依赖都会改变。最后的重复模式是：兄弟模块依赖和兄弟模块版本的重复。**Maven** 提供的简单机制能让你将所有这些依赖重构到一个父 **POM**。

就像你项目的源码一样，任何时候你在 **POM** 中有重复，你就开启了通往麻烦之路的大门。重复依赖声明使得很难保证一个大项目中的版本一致性。当你只有两个或者三个模块的时候，可能这不是一个大问题，但当你的组织正使用一个大型的，多模块 **Maven** 构建来管理分布于很多部门的数百个组件，一个依赖间的版本不匹配能够造成混乱。项目中一个对于名为 **ASM** 的字节码操作包的依赖版本不一致，即使处于项目层次的三层以下，如果该模块被依赖，还是可以影响到由另一个完全不同的开发组维护的 **web** 应用。单元测试会通过因为它们是基于一个版本的依赖运行的，但产品可能会灾难性的失败，原因是包（比如这里是 **war**）里存在有不同版本的类库。如果你拥有数十个项目使用比如 **Hibernate** 这样的东西，每个项目重复那些依赖和排除配置，那么有人搞坏构建的平均发生时间就会很短。由于你的 **Maven** 项目变得很复杂，依赖列表也会增大，你需要在父 **POM** 中巩固版本和依赖声明。

兄弟模块版本的重复可以造成一个特殊的令人讨厌的问题，该问题不是直接由 **Maven** 造成的，只有在你多次遇到这个问题之后你才会有认识。如果你使用 **Maven Release** 插件来运行你的发布，所有这些兄弟依赖版本都会被自动更新，因此维护它们就不是什么问题。如果 **simple-web** 版本 **1.3-SNAPSHOT** 依赖于 **simple-persist** 版本 **1.3-SNAPSHOT**，并且你正执行一次对于两个项目的版本 **1.3** 发布，**Maven Release** 插件很聪明，能够自动更改整个多模块项目中的所有 **POM**。使用 **Realse** 插件来运行发布能够自动将你构建的所有版本增加到 **1.4-SNAPSHOT**，并且 **release** 插件会将代码变更提交至代码库。发布一个大型的多模块项目会变得更简单，直到.....

当开发人员将更改合并到 POM 中并影响了一个正进行的版本发布的时候，问题就产生了。通常一个开发人员合并更改并偶然的错误处理了对于兄弟依赖的冲突，不注意的回退到了前一个发布的版本。由于同一个依赖的连续版本通常是相互兼容的，当开发人员构建的时候，问题不会出现，甚至暂时在持续构建系统也不会发现。想像一下一个十分复杂的构建，主干上都是 1.4-SNAPSHOT 的组件，现在有一个开发人员 A，将项目层次深处的组件 A 更新至依赖于组件 B 的 1.3-SNAPSHOT 版本。虽然大部分开发者都使用 1.4-SNAPSHOT 了，如果组件 B 的 1.3-SNAPSHOT 和 1.4-SNAPSHOT 相互兼容的话，该构建还是会成功。Maven 会继续使用从开发者本地仓库获取的组件 B 的 1.3-SNAPSHOT 版本构建该项目。所有事情开起来都很流畅，项目构建，持续集成构建都没问题，有人可能有一个关于组件 B 的诡异的 bug，我们也暂时将其认为是一个小问题记下来，然后继续下面的事情。

有个人，让我们称其为马虎先生，在组件 A 中有一个合并冲突，然后错误的将组件 A 对于组件 B 的依赖设为了 1.3-SNAPSHOT，而项目的其它部分继续向前推进。一堆开发人员试图修复组件 B 的 bug，诡异的是他们在产品环境中看不到 bug 被修复了。偶然间有人看了下组件 A 然后意识到这个依赖指向了一个错误的版本。幸运的是，这个 bug 还没有大到要消耗很多钱或时间，但是马虎先生感到自己十分愚蠢，由于这次的兄弟依赖关系混乱问题，人们也没以前那么信任他了。（还好，马虎先生意识到这是一个用户行为错误而非 Maven 的错，但可能它会写一个糟糕的博客去无休止的抱怨 Maven 来使自己好受一点。）

幸运的是，只要你做一些微小的更改，依赖重复和兄弟依赖不匹配问题就能简单的预防。我们要做的第一件事是找出所有被用于一个以上模块的依赖，然后将其向上移到父 POM 的 dependencyManagement 片段。我们先不管兄弟依赖。simple-parent 的 POM 现在包含内容如下：

```
<project>
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
```

```

    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>3.3.0.ga</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
</dependencyManagement>
...
</project>

```

在这些依赖配置被上移之后，我们需要为每个 POM 移除这些依赖的版本，否则它们会覆盖定义在父项目中的 dependencyManagement。这里我只是简单展示一下 simple-model:

```

<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
  </dependencies>
  ...
</project>

```

下一步我们应该做的是修复 hibernate-annotations 和 hibernate-commons-annotations 的版本重复问题，因为这两个版本应该是一致的，我们通过创建一个称为 hibernate-annotations-version 的属性。结果 simple-parent 的片段看起来这样：

```

<project>
...
<properties>
  <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
</properties>

<dependencyManagement>
...
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>${hibernate.annotations.version}</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>${hibernate.annotations.version}</version>
  </dependency>
...
</dependencyManagement>
...
</project>

```

我们需要处理的最后一个问题是兄弟依赖。一种方案是像其它依赖一样将它们移到父项目的 `dependencyManagement` 中，在最顶层的父项目中定义所有兄弟项目的版本。这样做当然是可以的，但我们也可以使用内建属性 `${project.groupId}` 和 `${project.version}` 来解决这个版本问题。由于它们是兄弟依赖，在父项目中枚举它们也不能获得更多的价值，因此我们依赖于内置的 `${project.version}` 属性。由于我们都共享一个共同的组，因此，通过使用内置的 `${project.groupId}` 属性引用当前 POM 的组，我们能够提前保证这些声明是正确的。simple-command 依赖片段现在变成了这样：

```

<project>
...
<dependencies>
...
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-weather</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-persist</artifactId>

```

```
<version>${project.version}</version>
</dependency>
...
</dependencies>
...
</project>
```

总结一下我们为了降低依赖重复而完成的两项优化：

上移共同的依赖至 `dependencyManagement`

如果多于一个项目依赖于一个特定的依赖，你可以在 `dependencyManagement` 中列出这个依赖。父 **POM** 包含一个版本和一组排除配置，所有的子 **POM** 需要使用 `groupId` 和 `artifactId` 引用这个依赖。如果依赖已经在 `dependencyManagement` 中列出，子项目可以忽略版本和排除配置。

为兄弟项目使用内置的项目 `version` 和 `groupId`

使用 `{project.version}` 和 `${project.groupId}` 来引用兄弟项目。兄弟项目基本上一直共享同样的 `groupId`，也基本上一直共享同样的发布版本。使用 `${project.version}` 可以帮你避免前面提到的兄弟版本不一致问题。

8.4. 优化插件



如果我们看一下不同的插件配置，就能看到 **HSQLDB** 依赖在很多地方有重复。不幸的是，`dependencyManagement` 不适用于插件依赖，但我们仍然可以使用属性来巩固版本。大部分复杂的 **Maven** 多模块项目倾向于在顶层 **POM** 中定义所有的版本。这个顶层 **POM** 就成了影响整个项目的更改焦点。将版本号看成是 **Java** 类中的字符串字面量，如果你经常重复一个字面量，你可能会将它定义为一个变量，当它需要变化的时候，你就只需要在一个地方更改它。将 **HSQLDB** 的版本集中到顶层的 **POM** 中，就产生了如下的属性元素。

```
<project>
...
<properties>
  <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  <hsqldb.version>1.8.0.7</hsqldb.version>
</properties>
...
</project>
```

下一件我们要注意的事情是 `hibernate3-maven-plugin` 配置在 `simple-webapp` 和 `simple-command` 模块中重复了。我们可以在顶层 **POM** 中管理这个插件配置，就像我们在顶层 **POM** 中使用 `dependencyManagement` 片段管理依赖一样。为此，我们要使用元素顶层 **POM** `build` 元素下的 `pluginManagement` 元素。

```
<project>
...
```

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
...
</project>
```

8.5. 使用 Maven Dependency 插件进行优化



在大型的项目中，随着依赖数目的增加，一些额外的依赖会悄悄进入项目的 POM 中。而当依赖改变的时候，你又常常会留下一些不再被使用的依赖，而又有时候，你会忘记显示声明你需要的类库依赖。由于 Maven 2.x 会在编译范围引入传递性依赖，你的项目可能编译没问题，但在产品阶段不能运行。考虑这种情况，当一个项目使用一些被广泛使用的类库如 Jakarta Commons Beanutils。你没有显式的声明对 Beanutils 的依赖，你的项目依赖于一个项目如 Hibernate，而后者有对 Beanutils 的传递性依赖。你的项目可能编译成功并很好的运行，但当你将 Hibernate 升级到一个新版本，而它不再依赖于 Beanutils，你就会遇到编译和运行错误了，这种情况直到项目不能编译才能显现。同时，由于你没有显式的列出依赖的版本，Maven 不能帮你解析可能出现的版本冲突问题。

一个好的经验方法是，总是为你代码引用的类显式声明依赖。如果你要引入 Commons Beanutils 类，你应该声明一个对于 Commons Beanutils 的直接依赖。幸运的是，通过字节码分析，Maven Dependency 插件能够帮助你发现对于依赖的直接引用。使用我们之前优化过的新的 POM，让我们看一下是否会有错误突然出现。

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Chapter 8 Simple Parent Project
[INFO]   Chapter 8 Simple Object Model
[INFO]   Chapter 8 Simple Weather API
[INFO]   Chapter 8 Simple Persistence API
[INFO]   Chapter 8 Simple Command Line Tool
[INFO]   Chapter 8 Simple Web Application
[INFO]   Chapter 8 Parent Project
[INFO] Searching repository for plugin with prefix: 'dependency'.

...

[INFO]
-----
[INFO] Building Chapter 8 Simple Object Model
[INFO]   task-segment: [dependency:analyze]
[INFO]
-----

[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
```

```
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]     javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]     org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[WARNING]     org.hibernate:hibernate:jar:3.2.5.ga:compile
[WARNING]     junit:junit:jar:3.8.1:test

...

[INFO]

-----

[INFO] Building Chapter 8 Simple Web Application
[INFO]     task-segment: [dependency:analyze]
[INFO]

-----

[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]     org.sonatype.mavenbook.ch08:simple-model:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]     org.apache.velocity:velocity:jar:1.5:compile
[WARNING]     javax.servlet:jstl:jar:1.1.2:compile
[WARNING]     taglibs:standard:jar:1.1.2:compile
[WARNING]     junit:junit:jar:3.8.1:test
```

在上面截取的输出中，我们能看到运行 `dependency:analyze` 目标的结果。该目标分析这个项目，查看是否有直接依赖，或者一些引用了但不是直接声明的依赖。在 `simple-model` 项目中，**dependency** 插件指出有一个对于 `javax.persistence:persistence-api` 的“使用了但未声明的依赖”。为了进一步调查，到 `simple-model` 目录下运行 `dependency:tree` 目标，该目标会列出项目中所有的直接和传递性依赖。

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO]

-----

[INFO] Building Chapter 8 Simple Object Model
[INFO]    task-segment: [dependency:tree]
[INFO]

-----

[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.ch08:simple-model:jar:1.0
[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[INFO] |   \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
[INFO] |   +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] |   +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |   +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |   +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |   +- antlr:antlr:jar:2.7.6:compile
[INFO] |   +- cglib:cglib:jar:2.1_3:compile
[INFO] |   +- asm:asm:jar:1.5.3:compile
[INFO] |   \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO]

-----

[INFO] BUILD SUCCESSFUL
[INFO]

-----

```

从上面的输出我们可以看到 persistence-api 依赖来自于 hibernate。对该模块源码的粗略扫描会展现很多的 javax.persistence 的 **import** 语句，这让我们确信直接引用了这个依赖。简单的修复手段是添加对这个依赖的直接引用。该例中，我们将依赖版本放到 simple-parent 的 dependencyManagement 片段中，因为这个依赖链接到 **Hibernate**，而 **Hibernate** 的版本是在这里声明的。最终你会想要升级的项目的 **Hibernate** 版本，在 **Hibernate** 依赖旁边列出 persistence-api 依赖的版本能让你在将来升级父 POM 中的 **Hibernate** 版本的时候，更明显的看到两者关联。

如果你查看 simple-web 模块的 dependency:analyze 输出，你会看到这里我们也需要添加对 simple-model 的直接依赖。simple-webapp 中的代码直接引用 simple-model 中的模型对象，而现在 simple-model 是通过 simple-persist 的传递性依赖暴露给 simple-webapp 的。既然兄弟依赖共享同样的 version 和 groupId，因此这个依赖可以在 simple-webapp 的 pom.xml 中用 \${project.groupId} 和 \${project.version} 定义。

Maven Dependency 插件是如何发现这些问题的呢？`dependency:analyze` 如何知道什么类和依赖是你项目的字节码直接引用的？**Dependency** 插件使用 [ObjectWeb ASM](#) 工具包来分析字节码。**Dependency** 插件使用 **ASM** 来遍历当前项目中的所有类，构建一个所有其它被引用的类的列表。之后它遍历所有的依赖，直接依赖和传递性依赖，然后标记所有在直接依赖中发现的类。任何没有在直接依赖中找到的类会在传递性依赖中被发现，然后，“使用的，但未声明的依赖”列表就产生了。

相反的，未使用的，但声明的依赖列表就相对比较容易验证了，而且该列表没有“使用的，但未声明的依赖”有用。一种情况，一些依赖只在运行时或测试时使用，它们不会在字节码中被发现。你能在输出中很明显的看到它们的存在，例如，**JUnit** 就在这个列表中，但是它是需要的，因为它被用来做单元测试。你也会在 **simple-web** 模块中注意到 **Velocity** 和 **Servlet API** 依赖出现在这个列表中，它们也是需要的，因为，虽然项目的类中没有任何对这些依赖的直接引用，但在运行的时候它们是必要的。

小心移除那些未使用，但声明的依赖，除非你拥有很好的测试覆盖率，否则你很可能引入了一个运行时错误。字节码优化还会突然出现更险恶的问题，例如，编译器可以合法的替换常量的值，优化并移除引用。此时移除依赖会造成编译错误，但是工具却分析显示该依赖未被使用。将来的 **Maven Dependency** 插件版本会提供个更好的技术来检测或忽略这些类型的问题。

你应该定期的使用 `dependency:analyze` 工具来检测你项目中的这些普遍错误。你可以配置它，当一些条件被发现的时候，让构建失败，它也能够用来生成报告。

8.6. 最终的 POM



作为一个最后的总结，最终的 **POM** 文件被列出以作为本章的参考。这是 **simple-parent** 的顶层 **POM**。

Example 8.1. **simple-parent** 的最终 **POM**

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 8 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
  </modules>
</project>
```

```
<module>simple-webapp</module>
</modules>

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<properties>
  <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  <hsqldb.version>1.8.0.7</hsqldb.version>
</properties>
<dependencyManagement>
  <dependencies>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.0.7</version>
</dependency>
<dependency>
  <groupId>org.apache.velocity</groupId>
  <artifactId>velocity</artifactId>
  <version>1.5</version>
</dependency>
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>persistence-api</artifactId>
  <version>1.0</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-annotations</artifactId>
  <version>${hibernate.annotations.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-commons-annotations</artifactId>
  <version>${hibernate.annotations.version}</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate</artifactId>
  <version>3.2.5.ga</version>
  <exclusions>
    <exclusion>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>
    </exclusion>
  </exclusions>
</dependency>
</dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
```

```
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

下面是该工具的命令行版本，simple-command 的最终 POM。

Example 8.2. simple-command 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Chapter 8 Simple Command Line Tool</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifest>
                <mainClass>org.sonatype.mavenbook.weather.Main</mainClass>
                <addClasspath>true</addClasspath>
              </manifest>
            </archive>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>true</testFailureIgnore>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

```

    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
</build>

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-weather</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-persist</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
  </dependency>
</dependencies>
</project>

```

接下来是 simple-model 项目的 POM。simple-model 包含了所有应用的模型对象。

Example 8.3. simple-model 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
    </dependency>
  </dependencies>
</project>
```

下一个 POM 是 simple-persist 项目的 POM。simple-persist 项目包含了使用 Hibernate 实现的所有持久化逻辑。

Example 8.4. simple-persist 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-jta_1.1_spec</artifactId>
      <version>1.1</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
  </dependencies>
```

```
</project>
```

simple-weather 项目包含了解析 Yahoo! Weather RSS 信息源的所有逻辑。该项目依赖于 simple-model 项目。

Example 8.5. simple-weather 的最终 POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Weather API</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
```

```

    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

最后，simple-webapp 项目与 simple-weather 项目生成的类库交互，同时也将获得的天气预报数据存储至 HSQLDB 数据库。

Example 8.6. simple-webapp 的最终 POM

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.ch08</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Chapter 8 Simple Web Application</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.geronimo.specs</groupId>
      <artifactId>geronimo-servlet_2.4_spec</artifactId>
      <version>1.1.1</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-weather</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-persist</artifactId>

```

```
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
</dependency>
<dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
</dependency>
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
</dependency>
</dependencies>
<build>
    <finalName>simple-webapp</finalName>
    <plugins>
        <plugin>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>maven-jetty-plugin</artifactId>
            <version>6.1.9</version>
            <dependencies>
                <dependency>
                    <groupId>hsqldb</groupId>
                    <artifactId>hsqldb</artifactId>
                    <version>${hsqldb.version}</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

8.7. 小结



本章为你展现了一些改进控制你项目依赖和插件的技术，以使你未来的构建维护变得轻松。我们推荐定期的用这种方式复查你的构建以确保重复以及相应的潜在问题能最小化。随着项目的成熟，新的依赖难免被引入，你可能发现之前在一个地方使用的一个依赖现在在 10 个地方使用了，需要将它上移。使用和未使用的依赖列表不停的在变化，它们也能够使用 Maven Dependency 插件轻松的清理。

Part II. Maven Reference

Table of Contents

[9. 项目对象模型](#)

[9.1. 简介](#)

[9.2. POM](#)

[9.2.1. 超级 POM](#)

[9.2.2. 最简单的 POM](#)

[9.2.3. 有效 POM](#)

[9.2.4. 真正的 POM](#)

[9.3. POM 语法](#)

[9.3.1. 项目版本](#)

[9.3.1.1. 版本构建号](#)

[9.3.1.2. SNAPSHOT 版本](#)

[9.3.1.3. LATEST 和 RELEASE 版本](#)

[9.3.2. 属性引用](#)

[9.4. 项目依赖](#)

[9.4.1. 依赖范围](#)

[9.4.2. 可选依赖](#)

[9.4.3. 依赖版本界限](#)

[9.4.4. 传递性依赖](#)

[9.4.4.1. 传递性依赖和范围](#)

[9.4.5. 冲突解决](#)

[9.4.6. 依赖管理](#)

[9.5. 项目关系](#)

[9.5.1. 坐标详解](#)

[9.5.2. 多模块项目](#)

[9.5.3. 项目继承](#)

[9.6. POM 最佳实践](#)

[9.6.1. 依赖归类](#)

[9.6.2. 多模块 vs. 继承](#)

[9.6.2.1. 简单项目](#)

[9.6.2.2. 多模块企业级项目](#)

[9.6.2.3. 原型父项目](#)

[10. 构建生命周期](#)

[10.1. 简介](#)

[10.1.1. 清理生命周期 \(clean\)](#)

[10.1.2. 默认生命周期 \(default\)](#)

[10.1.3. 站点生命周期 \(site\)](#)

[10.2. 打包相关生命周期](#)

[10.2.1. JAR](#)

[10.2.2. POM](#)

[10.2.3. Maven Plugin](#)

[10.2.4. EJB](#)

[10.2.5. WAR](#)

[10.2.6. EAR](#)

[10.2.7. 其它打包类型](#)

[10.3. 通用生命周期目标](#)

[10.3.1. Process Resources](#)

[10.3.2. Compile](#)

[10.3.3. Process Test Resources](#)

[10.3.4. Test Compile](#)

[10.3.5. Test](#)

[10.3.6. Install](#)

[10.3.7. Deploy](#)

[11. 构建 Profile](#)

[11.1. Profile 是用来做什么的?](#)

[11.1.1. 什么是构建可移植性](#)

[11.1.1.1. 不可移植构建](#)

[11.1.1.2. 环境可移植性](#)

[11.1.1.3. 组织（内部）可移植性](#)

[11.1.1.4. 广泛（全局）可移植性](#)

[11.1.2. 选择一个适当级别的可移植性](#)

[11.2. 通过 Maven Profiles 实现可移植性](#)

[11.2.1. 覆盖一个项目对象模型](#)

[11.3. 激活 Profile](#)

[11.3.1. 激活配置](#)

[11.3.2. 通过属性缺失激活](#)

[11.4. 外部 Profile](#)

[11.5. Settings Profile](#)

[11.5.1. 全局 Settings Profile](#)

[11.6. 列出活动的 Profile](#)

[11.7. 提示和技巧](#)

[11.7.1. 常见的环境](#)

[11.7.2. 安全保护](#)

[11.7.3. 平台分类器](#)

[11.8. 小结](#)

[12. Maven Assemblies](#)

[12.1. Introduction](#)

[12.2. Assembly Basics](#)

[12.2.1. Predefined Assembly Descriptors](#)

[12.2.2. Building an Assembly](#)

[12.2.3. Assemblies as Dependencies](#)

[12.2.4. Assembling Assemblies via Assembly Dependencies](#)

[12.3. Overview of the Assembly Descriptor](#)

[12.4. The Assembly Descriptor](#)

[12.4.1. Property References in Assembly Descriptors](#)

[12.4.2. Required Assembly Information](#)

[12.5. Controlling the Contents of an Assembly](#)

[12.5.1. Files Section](#)

[12.5.2. FileSets Section](#)

[12.5.3. Default Exclusion Patterns for fileSets](#)

[12.5.4. dependencySets Section](#)

[12.5.4.1. Customizing Dependency Output Location](#)

[12.5.4.2. Interpolation of Properties in Dependency Output Location](#)

[12.5.4.3. Including and Excluding Dependencies by Scope](#)

[12.5.4.4. Fine Tuning: Dependency Includes and Excludes](#)

[12.5.4.5. Transitive Dependencies, Project Attachments, and Project Artifacts](#)

[12.5.4.6. Advanced Unpacking Options](#)

[12.5.4.7. Summarizing Dependency Sets](#)

[12.5.5. moduleSets Sections](#)

[12.5.5.1. Module Selection](#)

[12.5.5.2. Sources Section](#)

[12.5.5.3. Interpolation of outputDirectoryMapping in moduleSets](#)

[12.5.5.4. Binaries section](#)

[12.5.5.5. moduleSets, Parent POMs and the binaries Section](#)

[12.5.6. Repositories Section](#)

[12.5.7. Managing the Assembly's Root Directory](#)

[12.5.8. componentDescriptors and containerDescriptorHandlers](#)

[12.6. Best Practices](#)

[12.6.1. Standard, Reusable Assembly Descriptors](#)

[12.6.2. Distribution \(Aggregating\) Assemblies](#)

[12.7. Summary](#)

[13. Properties and Resource Filtering](#)

[13.1. Introduction](#)

[13.2. Maven Properties](#)

[13.2.1. Maven Project Properties](#)

[13.2.2. Maven Settings Properties](#)

[13.2.3. Environment Variable Properties](#)

[13.2.4. Java System Properties](#)

[13.2.5. User-defined Properties](#)

[13.3. Resource Filtering](#)

[14. Maven 和 Eclipse: m2eclipse](#)

[14.1. 简介](#)

[14.2. m2eclipse](#)

[14.3. 安装 m2eclipse 插件](#)

[14.3.1. 安装前提条件](#)

[14.3.1.1. 安装 Subclipse](#)

[14.3.1.2. 安装 Mylyn](#)

[14.3.1.3. 安装 AspectJ Tools Platform \(AJDT\)](#)

[14.3.1.4. 安装 Web Tools Platform \(WTP\)](#)

[14.3.2. 安装 m2eclipse](#)

[14.4. 开启 Maven 控制台](#)

[14.5. 创建一个 Maven 项目](#)

[14.5.1. 从 SCM 签出一个 Maven 项目](#)

[14.5.2. 用 Maven Archetype 创建一个 Maven 项目](#)

[14.5.3. 创建一个 Maven 模块](#)

[14.6. 创建一个 Maven POM 文件](#)

[14.7. 导入 Maven 项目](#)

[14.7.1. 导入一个 Maven 项目](#)

[14.7.2. 具体化一个 Maven 项目](#)

[14.8. 运行 Maven 构建](#)

[14.9. 使用 Maven 进行工作](#)

[14.9.1. 添加及更新依赖或插件](#)

[14.9.2. 创建一个 Maven 模块](#)

[14.9.3. 下载源码](#)

[14.9.4. 打开项目页面](#)

[14.9.5. 解析依赖](#)

[14.10. 使用 Maven 仓库进行工作](#)

[14.10.1. 搜索 Maven 构件和 Java 类](#)

[14.10.2. 为 Maven 仓库编制索引](#)

[14.11. 使用基于表单的 POM 编辑器](#)

[14.12. 在 m2eclipse 中分析项目依赖](#)

[14.13. Maven 选项](#)

[14.14. 小结](#)

[15. Site Generation](#)

[15.1. Introduction](#)

[15.2. Building a Project Site with Maven](#)

[15.3. Customizing the Site Descriptor](#)

[15.3.1. Customizing the Header Graphics](#)

[15.3.2. Customizing the Navigation Menu](#)

[15.4. Site Directory Structure](#)

[15.5. Writing Project Documentation](#)

[15.5.1. APT Example](#)

[15.5.2. FML Example](#)

[15.6. Deploying Your Project Website](#)

[15.6.1. Configuring Server Authentication](#)

[15.6.2. Configuring File and Directory Modes](#)

[15.7. Customizing Site Appearance](#)

[15.7.1. Customizing the Site CSS](#)

[15.7.2. Create a Custom Site Template](#)

[15.7.3. Reusable Website Skins](#)

[15.7.4. Creating a Custom Theme CSS](#)

[15.7.5. Customizing Site Templates in a Skin](#)

[15.8. Tips and Tricks](#)

[15.8.1. Inject XHTML into HEAD](#)

[15.8.2. Add Links under Your Site Logo](#)

[15.8.3. Add Breadcrumbs to Your Site](#)

[15.8.4. Add the Project Version](#)

[15.8.5. Modify the Publication Date Format and Location](#)

[15.8.6. Using Doxia Macros](#)

[16. 仓库管理器](#)

[16.1. 简介](#)

[16.1.1. Nexus 历史](#)

[16.2. 安装 Nexus](#)

[16.2.1. 从 Sonatype 下载 Nexus](#)

[16.2.2. 安装 Nexus](#)

[16.2.3. 运行 Nexus](#)

[16.2.4. 安装后检查单](#)

[16.2.5. 为 Redhat/Fedora/CentOS 设置启动脚本](#)

[16.2.6. 升级 Nexus 版本](#)

[16.3. 使用 Nexus](#)

[16.3.1. 浏览仓库](#)

[16.3.2. 浏览组](#)

[16.3.3. 搜索构件](#)

[16.3.4. 浏览系统 RSS 源](#)

[16.3.5. 浏览日志文件和配置](#)

[16.3.6. 更改你的密码](#)

[16.4. 配置 Maven 使用 Nexus](#)

[16.4.1. 使用 Nexus 中央代理仓库](#)

[16.4.2. 使用 Nexus 作为快照仓库](#)

[16.4.3. 为缺少的依赖添加仓库](#)

[16.4.4. 添加一个新的仓库](#)

[16.4.5. 添加一个仓库至一个组](#)

[16.5. 配置 Nexus](#)

[16.5.1. 定制服务器配置](#)

[16.5.2. 管理仓库](#)

[16.5.3. 管理组](#)

[16.5.4. 管理路由](#)

[16.5.5. 网络配置](#)

[16.6. 维护仓库](#)

[16.7. 部署构件至 Nexus](#)

[16.7.1. 部署发布版](#)

[16.7.2. 部署快照版](#)

[16.7.3. 部署第三方构件](#)

[17. Writing Plugins](#)

[17.1. Introduction](#)

[17.2. Programming Maven](#)

[17.2.1. What is Inversion of Control?](#)

[17.2.2. Introduction to Plexus](#)

[17.2.3. Why Plexus?](#)

[17.2.4. What is a Plugin?](#)

[17.3. Plugin Descriptor](#)

[17.3.1. Top-level Plugin Descriptor Elements](#)

[17.3.2. Mojo Configuration](#)

[17.3.3. Plugin Dependencies](#)

[17.4. Writing a Custom Plugin](#)

[17.4.1. Creating a Plugin Project](#)

[17.4.2. A Simple Java Mojo](#)

[17.4.3. Configuring a Plugin Prefix](#)

[17.4.4. Logging from a Plugin](#)

[17.4.5. Mojo Class Annotations](#)

[17.4.6. When a Mojo Fails](#)

[17.5. Mojo Parameters](#)

[17.5.1. Supplying Values for Mojo Parameters](#)

[17.5.2. Multi-valued Mojo Parameters](#)

[17.5.3. Depending on Plexus Components](#)

[17.5.4. Mojo Parameter Annotations](#)

[17.6. Plugins and the Maven Lifecycle](#)

[17.6.1. Executing a Parallel Lifecycle](#)

[17.6.2. Creating a Custom Lifecycle](#)

[17.6.3. Overriding the Default Lifecycle](#)

[18. Writing Plugins in Alternative Languages](#)

[18.1. Writing Plugins in Ant](#)

[18.2. Creating an Ant Plugin](#)

[18.3. Writing Plugins in JRuby](#)

[18.3.1. Creating a JRuby Plugin](#)

[18.3.2. Ruby Mojo Implementations](#)

[18.3.3. Logging from a Ruby Mojo](#)

[18.3.4. Raising a MojoError](#)

[18.3.5. Referencing Plexus Components from JRuby](#)

[18.4. Writing Plugins in Groovy](#)

[18.4.1. Creating a Groovy Plugin](#)

Chapter 9. 项目对象模型

[9.1. 简介](#)

[9.2. POM](#)

[9.2.1. 超级 POM](#)

[9.2.2. 最简单的 POM](#)

[9.2.3. 有效 POM](#)

[9.2.4. 真正的 POM](#)

[9.3. POM 语法](#)

[9.3.1. 项目版本](#)

[9.3.1.1. 版本构建号](#)

[9.3.1.2. SNAPSHOT 版本](#)

[9.3.1.3. LATEST 和 RELEASE 版本](#)

[9.3.2. 属性引用](#)

[9.4. 项目依赖](#)

[9.4.1. 依赖范围](#)

[9.4.2. 可选依赖](#)

[9.4.3. 依赖版本界限](#)

[9.4.4. 传递性依赖](#)

[9.4.4.1. 传递性依赖和范围](#)

[9.4.5. 冲突解决](#)

[9.4.6. 依赖管理](#)

[9.5. 项目关系](#)

[9.5.1. 坐标详解](#)

[9.5.2. 多模块项目](#)

[9.5.3. 项目继承](#)

[9.6. POM 最佳实践](#)

[9.6.1. 依赖归类](#)

[9.6.2. 多模块 vs. 继承](#)

[9.6.2.1. 简单项目](#)

[9.6.2.2. 多模块企业级项目](#)

[9.6.2.3. 原型父项目](#)

9.1. 简介



本章讨论 Maven 的核心概念——项目对象模型。在 POM 中，项目的坐标和结构被声明，构建被配置，与其它项目的关联也被定义。pom.xml 文件定义了一个 Maven 项目。

9.2. POM



Maven 项目，依赖，构建配置，以及构件：所有这些都是要建模和表述的对象。这些对象通过一个名为项目对象模型(Project Object Model, POM)的 XML 文件描述。这个 POM 告诉 Maven 它正处理什么类型的项目，如何修改默认的行为来从源码生成输出。同样的方式，一个 Java Web 应用有一个 web.xml 文件来描述，配置，及自定义该应用，一个 Maven 项目则通过一个 pom.xml 文件定义。该文件是 Maven 中一个项目的描述性陈述；也是当 Maven 构建项目的时候需要理解的一份“地图”。

你可以将 pom.xml 看成是类似于 Makefile 或者 Ant 中的 build.xml。当你使用 GNU make 来构建诸如 MySQL 软件的时候，你通常会有一个名为 Makefile 的文件，它包含了显式的指令来清理，编译，打包以及部署一个应用。在这一点上，Make，Ant，和 Maven 是相似的，它们都依赖于一个统一命名的文件如 Makefile，build.xml，或者 pom.xml，但相似的地方也仅此而已。如果你看一下 Maven 的 pom.xml，POM 的主要内容是处理描述信息：哪里存放源代码？哪里存放资源文件？打包方式是什么？如果你看一下 Ant 的 build.xml 文件，你会看到完全不同的东西。那里有显式的指令来执行一些任务，如编译一组 Java 类。Maven 的 POM 是声明性的，虽然你可以通过 Maven Ant 插件来引入一些过程式的自定义指令，但大部分时间里，你不需要去了解项目构建的过程细节。

POM 也不只是仅仅针对于构建 Java 项目。虽然本书的大部分样例都是 Java 应用，但是在 Maven 的项目对象模型定义中没有任何 Java 特定的东西。虽然 Maven 的默认插件是从一组源码，测试，和资源来构建一个 JAR 文件。但你同样可以为一个包含 C# 源码，使用微软工具处理一些微软私有的二进制文件的项目来定义一个 POM。类似的，你也可以为一本技术书籍定义一个 POM。事实上，本书的源码和本书的样例正是用一个 Maven 多模块项目组织的，我们使用一个 Maven Docbook 插件，将标准的 Docbook XSL 应用到一系列章节的 XML 文件上。还有人编写了 Maven 插件来将 Adobe Flex 代码构建成 SWC 和 SWF，也还有人使用了 Maven 来构建 C 编写的项目。

我们已经确定了 POM 是描述性和声明性的，它不像 Ant 或者 Make 那样提供显式的指令，我们也注意到 POM 的概念不是特定于 Java 的。让我们深入更多的细节，看一下 [Figure 9.1, “项目对象模型”](#)，纵览一下 POM 的内容。

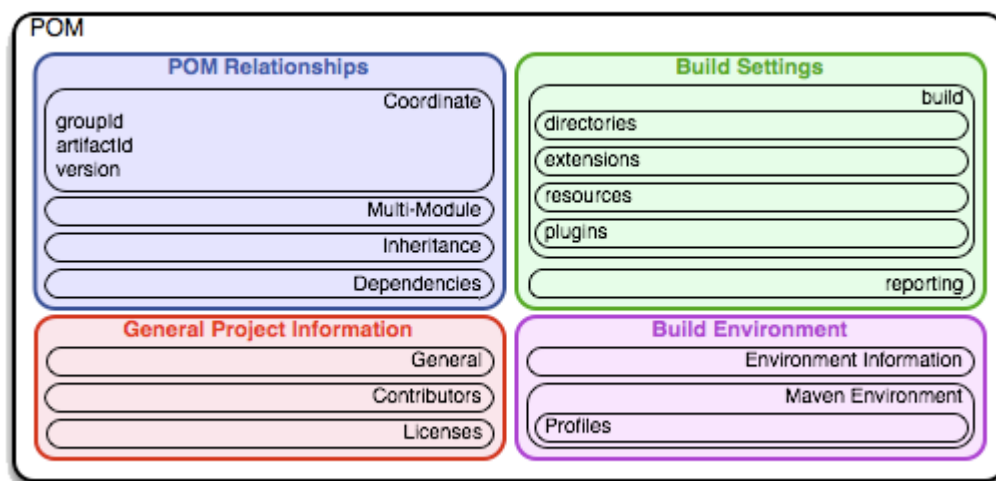


Figure 9.1. 项目对象模型

POM 包含了四类描述和配置：

项目总体信息

它包含了一个项目的名称，项目的 URL，发起组织，以及项目的开发者贡献者列表和许可证。

构建设置

在这一部分，我们自定义 Maven 构建的默认行为。我们可以更改源码和测试代码的位置，可以添加新的插件，可以将插件目标绑定到生命周期，我们还可以自定义站点生成参数。

构建环境

构建环境包含了一些能在不同使用环境中 激活的 profile。例如，在开发过程中你可能会想要将应用部署到一个开发服务器上，而在产品环境中你会需要将应用部署到产品服务器上。构建环境为特定的环境定制了构建设置，通常它还会由 ~/.m2 中的自定义 settings.xml 补充。这个 settings 文件将会在 [Chapter 11, 构建 Profile](#) 中，以及 [Section A.1, “Quick Overview”](#) 中的 [Appendix A, Appendix: Settings Details](#) 小节中讨论。

POM 关系

一个项目很少孤立存在；它会依赖于其它项目，可能从父项目继承 POM 设置，它要定义自身的坐标，可能还会包含子模块。

9.2.1. 超级 POM



在深入钻研一些样例 POM 之前，让我们先快速看一下超级 POM。所有的 Maven 项目的 POM 都扩展自超级 POM。超级 POM 定义了一组被所有项目共享的默认设置。它是 Maven 安装的一部分，可以在 `{M2_HOME}/lib` 中的 `maven-2.0.9-uber.jar` 文件中找到。如果你看一下这个 JAR 文件，你会看到在包 `org.apache.maven.project` 下看到一个名为 `pom-4.0.0.xml` 的文件。这个 Maven 的超级 POM 如 [Example 9.1, “超级 POM”](#) 所示。

Example 9.1. 超级 POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>

  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <updatePolicy>never</updatePolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>

  <build>
    <directory>target</directory>
    <outputDirectory>target/classes</outputDirectory>
```

```
<finalName>${pom.artifactId}-${pom.version}</finalName>
<testOutputDirectory>target/test-classes</testOutputDirectory>
<sourceDirectory>src/main/java</sourceDirectory>
<scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
<testSourceDirectory>src/test/java</testSourceDirectory>
<resources>
  <resource>
    <directory>src/main/resources</directory>
  </resource>
</resources>
<testResources>
  <testResource>
    <directory>src/test/resources</directory>
  </testResource>
</testResources>
</build>
```

```
<pluginManagement>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.1</version>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2-beta-1</version>
    </plugin>
    <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
    </plugin>
    <plugin>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.0</version>
    </plugin>
    <plugin>
      <artifactId>maven-deploy-plugin</artifactId>
      <version>2.3</version>
    </plugin>
    <plugin>
```

```
<artifactId>maven-ear-plugin</artifactId>
<version>2.3.1</version>
</plugin>
<plugin>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.1</version>
</plugin>
<plugin>
  <artifactId>maven-install-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.4</version>
</plugin>
<plugin>
  <artifactId>maven-plugin-plugin</artifactId>
  <version>2.3</version>
</plugin>
<plugin>
  <artifactId>maven-rar-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.0-beta-7</version>
</plugin>
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>2.0-beta-6</version>
</plugin>
<plugin>
  <artifactId>maven-source-plugin</artifactId>
  <version>2.0.4</version>
</plugin>
<plugin>
```

```
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.4.2</version>
    </plugin>
    <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1-alpha-1</version>
    </plugin>
</plugins>
</pluginManagement>

<reporting>
    <outputDirectory>target/site</outputDirectory>
</reporting>
</project>
```

这个超级 POM 定义了一些由所有项目继承的标准配置变量。对这些变量的简单解释如下：

- ❶ 默认的超级 POM 定义了一个单独的远程 Maven 仓库，ID 为 central。这是所有 Maven 客户端默认配置访问的中央 Maven 仓库。该配置可以通过一个自定义的 settings.xml 文件来覆盖。注意这个默认的超级 POM 关闭了从中央 Maven 仓库下载 snapshot 构件的功能。如果你需要使用一个 snapshot 仓库，你就要在你的 pom.xml 或者 settings.xml 中自定义仓库设置。Settings 和 profile 将会在 [Chapter 11. 构建 Profile](#) 中和 [Section A.1, “Quick Overview”](#) 中的 [Appendix A, Appendix: Settings Details](#) 小节中具体介绍。
- ❷ 中央 Maven 仓库同时也包含 Maven 插件。默认的插件仓库就是这个中央仓库。Snapshot 被关闭了，而且更新策略被设置成了“从不”，这意味着 Maven 将永远不会自动更新一个插件，即使新版本的插件发布了。
- ❸ build 元素设置 Maven 标准目录布局中那些目录的默认值。
- ❹ 从 Maven 2.0.9 开始，超级 POM 为核心插件提供了默认版本。这么做是为那些没有在它们 POM 中指定插件版本的用户提供一些稳定性。

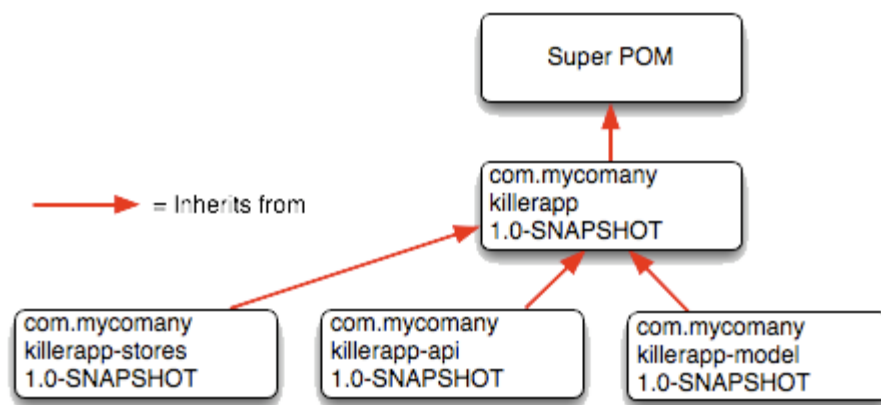


Figure 9.2. 超级 POM 永远是最基础的父 POM

9.2.2. 最简单的 POM



所有的 Maven POM 都继承自超级 POM（在前面的小节 [Section 9.2.1, “超级 POM”](#) 中介绍）。如果你只是编写一个简单的项目，从 `src/main/java` 目录的源码生成一个 JAR，想要运行 `src/test/java` 中的 JUnit 测试，想要使用 `mvn site` 构建一个项目站点，你不需要自定义任何东西。在这种情况下，你所需要的，是如 [Example 9.2, “最简单的 POM”](#) 所示的一个最简单的 POM。这个 POM 定义了 `groupId`, `artifactId` 和 `version`：这三项是所有项目都需要的坐标。

Example 9.2. 最简单的 POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

对一个简单的项目来说，这样一个简单的 POM 已经足够了——例如，一个生成单个 JAR 文件的 Java 类库。它不需要和任何其它项目关联，没有任何依赖，也缺少基本的信息如名字和 URL。如果创建了这个文件，然后创建了一个子目录 `src/main/java`，并且放入了一些源代码，运行 `mvn package` 将会生成一个名为 `target/simple-project-1.jar` 的 JAR 文件。

9.2.3. 有效 POM



最简单的 POM 能带给我们“有效 POM”的概念。由于 POM 可以从其它 POM 继承配置，你就需要一直考虑超级 POM，再加上任何父 POM，以及最后当前项目的 POM 这些所有配置的结合。Maven 始于超级 POM，然后使用一个或多个父 POM 覆盖默认配置，最后使用当前项目的 POM 来覆盖之前生成的配置结果。最后你得到了一个混合了各个 POM 配置的有效 POM。如果你想要查看项目的有效 POM，你需要运行 `Maven Help`

插件的 effective-pom 目标, 该插件已经之前在小节 [Section 2.7, “使用 Maven Help 插件”](#) 中介绍。在 pom.xml 文件所在的目录执行以下的命令以运行 effective-pom 目标:

```
$ mvn help:effective-pom
```

执行 effective-pom 目标应该会打印出一个 XML 文档, 该文档的内容是超级 POM 和 [Example 9.2, “最简单的 POM”](#) 中 POM 内容的合并。

9.2.4. 真正的 POM



这里我们就不再设计一组 POM, 一步步的来看了, 你可以自己查看一下 [Part I, “Maven 实战”](#) 中的样例。Maven 就像是变色龙, 你可以挑选你想要使用的特性。对于一些开源项目来说, 列出开发者和贡献者, 生成整洁的项目文档, 使用 Maven Release 插件来自动管理版本发布可能很有价值。但另一方面, 一些在大公司环境下的小型团队中工作的人可能对 Maven 的分发管理功能或者开发成员列表功能不感兴趣。本章的剩余部分将会单独的讨论 POM 的特性。我们不会用数十页的一组相关 POM 来炮轰你, 而会集中于为 POM 中的特定的小节创建优良的参考内容。本章, 我们也会讨论 POM 之间的关系, 但不会在这里展示这样一个项目。如果你想要看这样一个示例, 请参考 [Chapter 7, 多模块企业级项目](#)。

9.3. POM 语法



Maven 项目中的 POM 永远都是基础目录下的一个名为 pom.xml 的文件。这个 XML 文档可以以 XML 声明开头, 或者你也可以选择忽略它。所有的 POM 的值都通过 XML 元素的形式体现。

9.3.1. 项目版本



一个 Maven 项目发布版本号用 version 编码, 用来分组和排序发布。Maven 中的版本包含了以下部分: 主版本, 次版本, 增量版本, 和限定版本号。一个版本中, 这些部分对应如下的格式:

```
<major version>.<minor version>.<incremental version>-<qualifier>
```

例如: 版本“1.3.5”由一个主版本 1, 一个次版本 3, 和一个增量版本 5。而一个版本“5”只有主版本 5, 没有次版本和增量版本。限定版本用来标识里程碑构建: alpha 和 beta 发布, 限定版本通过连字符与主版本, 次版本或增量版本隔离。例如, 版本“1.3-beta-01”有一个主版本 1, 次版本 3, 和一个限定版本“beta-01”。

当你想要在你的 POM 中使用版本界限的时候, 保持你的版本号与标准一致十分重要。在 [Section 9.4.3, “依赖版本界限”](#) 中介绍的版本界限, 允许你声明一个带有版本界限的依赖, 只有你遵循标准的时候该功能才被支持。因为 Maven 根据本节中介绍的版本号格式来对版本进行排序。

如果你的版本号与格式 <主版本>.<次版本>.<增量版本>-<限定版本>相匹配, 它就能被正确的比较; “1.2.3”将被评价成是一个比“1.0.2”更新的构件, 这种比较基于主版

本，次版本，和增量版本的数值。如果你的版本发布号没有符合本节介绍的标准，那么你的版本号只会根据字符串被比较；“1.0.1b”和“1.2.0b”会使用字符串比较。

9.3.1.1. 版本构建号



我们还需要对版本号的限定版本进行排序。以版本号“1.2.3-alpha-2”和“1.2.3-alpha-10”为例，这里“alpha-2”对应了第二次 alpha 构建，而“alpha-10”对应了第十次 alpha 构建。虽然“alpha-10”应该被认为是比“alpha-2”更新的构建，但 Maven 排序的结果是“alpha-10”比“alpha-2”更旧，问题的原因就是我们刚才讨论的 Maven 处理版本号的方式。

Maven 会将限定版本后面的数字认作一个构建版本。换句话说，这里限定版本是“alpha”，而构建版本是 2。虽然 Maven 被设计成将构建版本和限定版本分离，但目前这种解析还是失效的。因此，“alpha-2”和“alpha-10”是使用字符串进行比较的，而根据字母和数字“alpha-10”在“alpha-2”前面。要避开这种限制，你需要对你的限定版本使用一些技巧。如果你使用“alpha-02”和“alpha-10”，这个问题就消除了，一旦 Maven 能正确的解析版本构建号之后，这种工作方式也还是能用。

9.3.1.2. SNAPSHOT 版本



Maven 版本可以包含一个字符串字面量来表示项目正处于活动的开发状态。如果一个版本包含字符串“SNAPSHOT”，Maven 就会在安装或发布这个组件的时候将该符号展开为一个日期和时间值，转换为 UTC（协调世界时）。例如，如果你的项目有个版本为“1.0-SNAPSHOT”并且你将这个项目的构件部署到了一个 Maven 仓库，如果你在 UTC2008 年 2 月 7 号下午 11:08 部署了这个版本，Maven 就会将这个版本展开成“1.0-20080207-230803-1”。换句话说，当你发布一个 snapshot，你没有发布一个软件模块，你只是发布了一个特定时间的快照版本。

那么为什么要使用这种方式呢？SNAPSHOT 版本在项目活动的开发过程中使用。如果你的项目依赖的一个组件正处于开发过程中，你可以依赖于一个 SNAPSHOT 版本，在你运行构建的时候 Maven 会定期的从仓库下载最新的 snapshot。类似的，如果你系统的下一个发布版本是“1.4”你的项目需要拥有一个“1.4-SNAPSHOT”的版本，之后它被正式发布。

作为一个默认设置，Maven 不会从远程仓库检查 SNAPSHOT 版本，要依赖于 SNAPSHOT 版本，用户必须在 POM 中使用 repository 和 pluginRepository 元素显式的开启下载 snapshot 的功能。

当发布一个项目的时候，你需要解析所有对 SNAPSHOT 版本的依赖至正式发布的版本。如果一个项目依赖于 SNAPSHOT，那么这个依赖很不稳定，它随时可能变化。发布到非 snapshot 的 Maven 仓库（如 <http://repo1.maven.org/maven2>）的构件不能依赖于任何 SNAPSHOT 版本，因为 Maven 的超级 POM 对于中央仓库关闭了 snapshot。SNAPSHOT 版本只用于开发过程。

9.3.1.3. LATEST 和 RELEASE 版本



当你依赖于一个插件或一个依赖，你可以使用特殊的版本值 **LATEST** 或者 **RELEASE**。**LATEST** 是指某个特定构件最新的发布版或者快照版(snapshot)，最近被部署到某个特定仓库的构件。**RELEASE** 是指仓库中最后的一个非快照版本。总得来说，设计软件去依赖于一个构件的不明确的版本，并不是一个好的实践。如果你处于软件开发过程中，你可能想要使用 **RELEASE** 或者 **LATEST**，这么做十分方便，你也不用为每次一个第三方类库新版本的发布而去更新你配置的版本号。但当你发布软件的时候，你总是应该确定你的项目依赖于某个特定的版本，以减少构建的不确定性，免得被其它不受你控制的软件版本影响。如果无论如何你都要使用 **LATEST** 和 **RELEASE**，那么要小心使用。

Maven 2.0.9 之后，Maven 在超级 POM 中锁住了一些通用及核心 Maven 插件的版本号，以将某个特定版本 Maven 的核心 Maven 插件组标准化。这个变化在 Maven 2.0.9 中被引入，为 Maven 构建带来了稳定性和重现性。在 Maven 2.0.9 之前，Maven 会自动将核心插件更新至 **LATEST** 版本。这种行为导致了很多奇怪现象，因为新版本的插件可能会有一些 bug，甚至是行为变更，这往往使得原来的构建失败。当 Maven 自动更新核心插件的时候，我们就不能保证构建的重现性，因为插件随时都可能从中央仓库更新至一个新的版本。从 Maven 2.0.9 开始，Maven 从根本上锁住了一组核心插件的版本。非核心插件，或者说没有在超级 POM 中指定版本的插件仍然会使用 **LATEST** 版本去从仓库获取构件。由于这个原因，你在构件中使用任何一个自定义非核心插件的时候，都应该显式的指定版本号。

9.3.2. 属性引用



一个 POM 可以通过一对大括弧和前面一个美元符号来包含 对属性的引用。例如，考虑如下的 POM：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <build>
    <finalName>${project.groupId}-${project.artifactId}</finalName>
  </build>
</project>
```

如果你将这段 XML 放入 pom.xml，然后运行 **mvn help:effective-pom**，你会看到输出包含这一行：

```
...
<finalName>org.sonatype.mavenbook-project-a</finalName>
...
```

在 Maven 读取一个 POM 的时候，它会在载入 POM XML 的时候替换这些属性的引用。在 Maven 的高级使用中 Maven 属性经常出现，这些属性和其它系统中的属性如 Ant

或者 **Velocity** 类似。它们是一些由 `${...}` 划界的变量。**Maven** 提供了三个隐式的变量，可以用来访问环境变量，POM 信息，和 **Maven Settings**：

env

`env` 变量 暴露了你操作系统或者 **shell** 的环境变量。例如，在 **Maven POM** 中一个对 `${env.PATH}` 的引用将会被 `${PATH}` 环境变量替换（或者 **Windows** 中的 `%PATH%`）。

project

`project` 变量暴露了 **POM**。你可以使用点标记（.）的路径来引用 **POM** 元素的值。例如，在本节中我们使用过 `groupId` 和 `artifactId` 来设置构建配置中的 `finalName` 元素。这个属性引用的语法是：

`${project.groupId}-${project.artifactId}`。

settings

`settings` 变量暴露了 **Maven settings** 信息。可以使用点标记（.）的路径来引用 `settings.xml` 文件中元素的值。例如，`${settings.offline}` 会引用 `~/.m2/settings.xml` 文件中 `offline` 元素的值。

Note

你可能在老的构建中看到使用 `${pom.xxx}` 或者仅仅 `${xxx}` 来引用 **POM** 属性。这些方法已被弃用，我们只应该使用 `${project.xxx}`。

除了这三个隐式的变量，你还可以引用系统属性，以及任何在 **Maven POM** 中和构建 **profile** 中自定义的属性组。

Java 系统属性

所有可以通过 `java.lang.System` 中 `getProperties()` 方法访问的属性都被暴露成 **POM** 属性。一些系统属性的例子是：`${user.name}`，`${user.home}`，`${java.home}`，和 `${os.name}`。一个完整的系统属性列表可以在 `java.lang.System` 类的 **Javadoc** 中找到。

x

我们还可以通过 `pom.xml` 或者 `settings.xml` 中的 `properties` 元素设置自己的属性，或者还可以使用外部载入的文件中属性。如果你在 `pom.xml` 中设置了一个名为 `fooBar` 的属性，该属性就可以通过 `${fooBar}` 引用。当你构建一个系统，它针对不同的部署环境过滤资源，那么自定义属性就变得十分有用。这里是在 **POM** 中设置 `${foo}=bar` 的语法：

```
<properties>
  <foo>bar</foo>
</properties>
```

要了解更复杂的可用属性列表，查看 [Chapter 13, Properties and Resource Filtering](#)。

9.4. 项目依赖



Maven 可以管理内部和外部依赖。一个 Java 项目的外部依赖可能是如 Plexus, Spring Framework, 或者 Log4J 的类库。一个内部的依赖就像在“一个简单的 web 应用”中描述的那样, web 项目依赖于另外一个包含服务类, 模型类, 或者持久化逻辑的项目。[Example 9.3, “项目依赖”](#)展示了一些项目依赖的例子。

Example 9.3. 项目依赖

```
<project>
...
<dependencies>
  <dependency>
    <groupId>org.codehaus.xfire</groupId>
    <artifactId>xfire-java5</artifactId>
    <version>1.2.5</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-servlet_2.4_spec</artifactId>
    <version>1.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
...
</project>
```

这里的第一个依赖是对于来自 Codehaus 的 XFire SOAP 库的编译范围（compile）依赖。如果你的项目在编译, 测试, 和运行中都依赖于一个类库, 你就要使用这种依赖。第二种依赖是一个对于 JUnit 测试范围（test）的依赖。当你只有在测试的时候才引用类库的时候, 你就要使用测试范围依赖。[Example 9.3, “项目依赖”](#)中最后一个依赖是对于由 Apache Geronimo 项目实现的 Servlet 2.4 API 的依赖。最后一项依赖的范围是已提供的（provided）依赖。当你的开发过程只有在编译和测试时需要一个类库, 而该类库在运行的时候由容器提供, 那么你就需要使用已提供范围的依赖。

9.4.1. 依赖范围



Example 9.3, “项目依赖”简要介绍了三种依赖范围: `compile`, `test`, 和 `provided`。范围控制哪些依赖在哪些 `classpath` 中可用, 哪些依赖包含在一个应用中。让我们详细看一下每一种范围:

compile (编译范围)

`compile` 是默认的范围; 如果没有提供一个范围, 那该依赖的范围就是编译范围。编译范围依赖在所有的 `classpath` 中可用, 同时它们也会被打包。

provided (已提供范围)

`provided` 依赖只有在当 `JDK` 或者一个容器已提供该依赖之后才使用。例如, 如果你开发了一个 `web` 应用, 你可能在编译 `classpath` 中需要可用的 `Servlet API` 来编译一个 `servlet`, 但是你不会想要在打包好的 `WAR` 中包含这个 `Servlet API`; 这个 `Servlet API JAR` 由你的应用服务器或者 `servlet` 容器提供。已提供范围的依赖在编译 `classpath` (不是运行时) 可用。它们不是传递性的, 也不会被打包。

runtime (运行时范围)

`runtime` 依赖在运行和测试系统的时候需要, 但在编译的时候不需要。比如, 你可能在编译的时候只需要 `JDBC API JAR`, 而只有在运行的时候才需要 `JDBC` 驱动实现。

test (测试范围)

`test` 范围依赖 在一般的 编译和运行时都不需要, 它们只有在测试编译和测试运行阶段可用。测试范围依赖在之前的[???](#)中介绍过。

system (系统范围)

`system` 范围依赖与 `provided` 类似, 但是你必须显式的提供一个对于本地系统中 `JAR` 文件的路径。这么做是为了允许基于本地对象编译, 而这些对象是系统类库的一部分。这样的构件应该是一直可用的, **Maven** 也不会去仓库中寻找它。如果你将一个依赖范围设置成系统范围, 你必须同时提供一个 `systemPath` 元素。注意该范围是不推荐使用的 (你应该一直尽量去从公共或定制的 **Maven** 仓库中引用依赖)。

9.4.2. 可选依赖



假定你正在开发一个类库, 该类库提供高速缓存行为。你想要使用一些已存在的能够提供文件系统快速缓存和分布式快速缓存的类库, 而非从空白开始写自己的快速缓存系统。再假定你想要能让最终用户选择使用文件系统高速缓存或者内存分布式高速缓存。为了缓存文件系统, 你会要使用免费的类库如 **EHCache**

(<http://ehcache.sourceforge.net/>), 为了分布式内存缓存, 你想要使用免费的类库如 **SwarmCache** (<http://swarmcache.sourceforge.net/>)。你将编写一个接口, 并且创建一个可以被配置成使用 **EHCache** 或者 **SwarmCache** 的类库, 但是你想要避免为所有依赖于你类库的项目添加全部这两个缓存类库的的依赖。

换句话说，编译这个项目的时候你需要两个依赖类库，但是你不希望在使用你类库的项目中，这两个依赖类库同时作为传递性运行时依赖出现。你可以使用如 [Example 9.4. “声明可选依赖”](#)中的可选依赖来完成这个任务。

Example 9.4. 声明可选依赖

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>net.sf.ehcache</groupId>
      <artifactId>ehcache</artifactId>
      <version>1.4.1</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>swarmcache</groupId>
      <artifactId>swarmcache</artifactId>
      <version>1.0RC2</version>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.13</version>
    </dependency>
  </dependencies>
</project>
```

在你将这些依赖声明为可选之后，你就需要在依赖于 my-project 的项目中显式的引用对应的依赖。例如，如果你正编写一个应用，它依赖于 my-project，并且想要使用 EHCache 实现，你就需要在你项目添加如下的 dependency 元素。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-application</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0.0</version>
```

```

</dependency>
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>swarmcache</artifactId>
  <version>1.4.1</version>
</dependency>
</dependencies>
</project>

```

在理想的世界中，你不需要使用可选依赖。你可以将 **EHCache** 相关的代码放到 `my-project-ehcache` 子模块中，将 **SwarmCache** 相关的代码放到 `my-project-swarmcache` 子模块中，而非创建一个带有一系列可选依赖的大项目。这样，其它项目就可以只引用特定实现的项目，发挥传递性依赖的功效，而不用去引用 `my-project` 项目，再自己声明特定的依赖。

9.4.3. 依赖版本界限



你并不是必须为依赖声明某个特定的版本，你可以指定一个满足给定依赖的版本界限。例如，你可以指定你的项目依赖于 **JUnit** 的 **3.8** 或以上版本，或者说依赖于 **JUnit 1.2.10** 和 **1.2.14** 之间的某个版本。你可以使用如下的字符来围绕一个或多个版本号，来实现版本界限。

(,)

不包含量词

[,]

包含量词

例如，如果你想要访问 **JUnit** 任意的大于等于 **3.8** 但小于 **4.0** 的版本，你的依赖可以如 [Example 9.5, “指定一个依赖界限: JUnit 3.8 - JUnit 4.0”](#)编写：

Example 9.5. 指定一个依赖界限: JUnit 3.8 - JUnit 4.0

```

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8, 4.0)</version>
  <scope>test</scope>
</dependency>

```

如果想要依赖 **JUnit** 任意的不大于 **3.8.1** 的版本，你可以只指定一个上包含边界，如 [Example 9.6, “指定一个依赖界限: JUnit ≤ 3.8.1”](#)所示：

Example 9.6. 指定一个依赖界限: JUnit <= 3.8.1

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[, 3.8.1]</version>ex-de
  <scope>test</scope>
</dependency>
```

在逗号前面或者后面的版本不是必须的, 这种空缺意味着正无穷或者负无穷。例如, “[4.0,)”意思是任何大于等于 4.0 的版本, “(,2.0)”意思是任意小于 2.0 的版本。 “[1.2]”意思是只有版本 1.2, 没有其它。

Note

当声明一个“正常的”版本如 JUnit 3.8.2, 内部它其实被表述成“允许任何版本, 但最好是 3.8.2”。意思是当侦测到版本冲突的时候, Maven 会使用冲突算法来选择最好的版本。如果你指定 [3.8.2], 它意味只有 3.8.2 会被使用, 没有其它。如果其它什么地方有一个版本指定了 [3.8.1], 你会得到一个构建失败报告, 告诉你有版本冲突。我指出这一点是要让你知道有这一选项, 但要保守的使用它, 只有在确实需要的时候才使用。更好的做法是通过 dependencyManagement 来解决冲突。

9.4.4. 传递性依赖



一个传递性依赖就是对于一个依赖的依赖。如果 project-a 依赖于 project-b, 而后者接着依赖于 project-c, 那么 project-c 就被认为是 project-a 的传递性依赖。如果 project-c 依赖于 project-d, 那么 project-d 就也被认为是 project-a 的传递性依赖。Maven 的部分吸引力是由于它能够管理传递性依赖, 并且能够帮助开发者屏蔽掉跟踪所有编译期和运行期依赖的细节。你可以只依赖于一些包如 Spring Framework, 而不用担心 Spring Framework 的所有依赖, Maven 帮你自动管理了, 你不用自己去详细了解配置。

Maven 是怎样完成这件事情的呢? 它建立一个依赖图, 并且处理一些可能发生的冲突和重叠。例如, 如果 Maven 看到有两个项目依赖于同样的 groupId 和 artifactId, 它会自动整理出使用哪个依赖, 选择那个最新版本的依赖。虽然这听起来很方便, 但在一些边界情况中, 传递性依赖会造成一些配置问题。在这种情况下, 你可以使用依赖排除。

9.4.4.1. 传递性依赖和范围



[Section 9.4.1, “依赖范围”](#)中提到的每种依赖范围不仅仅影响声明项目中的依赖范围, 它也对所传递性依赖起作用。表达该信息最简单的方式是通过一张表来表述, 如 [Table 9.1, “范围如何影响传递性依赖”](#)。最顶层一行代表了传递性依赖的范围。最左边的一列代表了直接依赖的范围。行与列的交叉就是为某个传递性依赖指定的范围。表中的空格意思是该传递性依赖被忽略。

Table 9.1. 范围如何影响传递性依赖

直接范围	传递性范围			
	<i>compile</i>	<i>provided</i>	<i>runtime</i>	<i>test</i>
<i>compile</i>	compile	-	runtime	-
<i>provided</i>	provided	provided	provided	-
<i>runtime</i>	runtime	-	runtime	-
<i>test</i>	test	-	test	-

要阐明传递性依赖范围到直接依赖范围的关系，考虑如下例子。如果 project-a 包含一个对于 project-b 的测试范围依赖，后者包含一个对于 project-c 的编译范围依赖。project-c 将会是 project-a 的测试范围传递性依赖。

你可以将这看成是一个作用于依赖范围上的传递性边界。那些已提供范围和测试范围的传递性依赖往往不对项目产生影响。该规则的例外是已提供范围传递性依赖到已提供范围直接依赖还是项目的一个已提供范围依赖。编译范围和运行时范围的传递性依赖通常会影响到一个项目，无论它的直接依赖范围是什么。编译范围的传递性依赖将会和直接依赖产生与后者相同范围的结果。运行时范围的传递性依赖也会和直接依赖产生与后者相同范围的结果，除非当直接依赖是编译范围的时候，结果是运行时范围。

9.4.5. 冲突解决



有很多时候你需要排除一个传递性依赖，比如当你依赖于一个项目，后者又继而依赖于另外一个项目，但你的希望是，要么整个的排除这个传递性依赖，要么用另外一个提供同样功能的依赖来替代这个传递性依赖。[Example 9.7, “排除一个传递性依赖”](#)展示的例子中添加了一个对于 project-a 的依赖，但排除了传递性依赖 project-b。

Example 9.7. 排除一个传递性依赖

```
<dependency>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>project-b</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

通常，你会想要使用另外一个实现来替代一个传递性依赖。比如，如果你正依赖于一个类库，该类库又依赖于 Sun JTA API，你会想要替换这个传递性依赖。Hibernate 是

一个例子。Hibernate 依赖于 Sun JTA API，而后者在中央 Maven 仓库中不可用，因为它不能免费分发的。幸运的是，Apache Geronimo 项目创建了一些可以免费分发的独立实现类库。为了用另外的依赖来替换这个传递性依赖，你需要排除这个传递性依赖，然后在你的项目中再声明一个依赖。[Example 9.8, “排除并替换一个传递性依赖”](#)展示了这样一个替换的样例。

Example 9.8. 排除并替换一个传递性依赖

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

在 [Example 9.8, “排除并替换一个传递性依赖”](#) 中，没有什么标记说依赖 `geronimo-jta_1.1_spec` 是一个替换，它只是正好提供了和原来的 JTA 依赖一样的 API。以下列举了一些你可能想要排除或者替换传递性依赖的情况：

- 构建的 `groupId` 和 `artifactId` 已经更改了，而当前的项目需要一个与传递性依赖不同名称的版本——结果是 `classpath` 中出现了同样项目的两份内容。一般来说 Maven 会捕捉到这种冲突并且使用该项目的一个单独的版本，但是当 `artifactId` 和 `artifactId` 不一样的时候，Maven 就会认为它们是两种不同的类库。
- 某个构件没有在你的项目中被使用，而且该传递性依赖没有被标志为可选依赖。在这种情况下，你可能想要排除这种依赖，因为它不是你的系统需要的东西，你要尽量减少应用程序分发时的类库数目。
- 一个构件已经在运行时的容器中提供了，因此不应该被包含在你的构件中。该情况的一个例子是，如果一个依赖依赖于如 Servlet API 的东西，并且你又要确保这样的依赖没有包含在 web 应用的 WEB-INF/lib 目录中。
- 为了排除一个可能是多个实现的 API 的依赖。这种情况在 [Example 9.8, “排除并替换一个传递性依赖”](#) 中阐述；有一个 Sun API，需要点击许可证，并且需要

耗时的手工安装到自定义仓库，对于同样的 API 有可免费分发版本，在中央 Maven 仓库中可用（Geronimo's JTA 实现）。

9.4.6. 依赖管理



当你在你的超级复杂的企业中采用 Maven 之后，你有了两百多个相互关联的 Maven 项目，你开始想知道是否有一个更好的方法来处理依赖版本。如果每一个使用如 MySQL 数据库驱动依赖的项目都需要独立的列出该依赖的版本，在你需要升级到一个新版本的时候你就会遇到问题。由于这些版本号分散在你的项目树中，你需要手工的编写每一个引用该依赖的 pom.xml，确保每个地方的版本号都更改了。即使使用了 **find**，**xargs**，和 **awk**，你仍然有漏掉一个 POM 的风险。

幸运的是，Maven 在 dependencyManagement 元素中为你提供了一种方式来统一依赖版本号。你经常会在一个组织或者项目的最顶层的父 POM 中看到 dependencyManagement 元素。使用 pom.xml 中的 dependencyManagement 元素能让你在子项目中引用一个依赖而不用显式的列出版本号。Maven 会沿着父子层次向上走，直到找到一个拥有 dependencyManagement 元素的项目，然后它就会使用在这个 dependencyManagement 元素中指定的版本号。

例如，如果你有一大组项目使用 MySQL Java connector 版本 5.1.2，你可以在你的多模块项目的顶层 POM 中定义如下的 dependencyManagement 元素。

Example 9.9. 在一个顶层 POM 中定义依赖版本

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
```

然后，在子项目中，你可以使用如下的依赖 XML 添加一个对 MySQL Java Connector 的依赖：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
```

```
<artifactId>a-parent</artifactId>
<version>1.0.0</version>
</parent>
<artifactId>project-a</artifactId>
...
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
</project>
```

你应该注意到该子项目没有显式的列出 `mysql-connector-java` 依赖的版本。由于这个依赖在顶层 **POM** 的 `dependencyManagement` 元素中定义了，该版本号就会传播到所有子项目的 `mysql-connector-java` 依赖中。注意如果子项目定义了一个版本，它将覆盖顶层 **POM** 的 `dependencyManagement` 元素中的版本。那就是：只有在子项目没有直接声明一个版本的时候，`dependencyManagement` 定义的版本才会被使用。

顶层 **POM** 中的依赖管理与在一个广泛共享的父 **POM** 中定义一个依赖是不同的。对初学者来说，所有依赖都会被继承。如果 `mysql-connector-java` 在顶层父项目中被作为一个依赖列出，这个层次中的所有项目都将引用该依赖。为了不添加一些不必要的依赖，使用 `dependencyManagement` 能让你统一并集中化依赖版本的管理，而不用添加那些会被所有子项目继承的依赖。换句话说，`dependencyManagement` 元素和一个环境变量一样，能让你在一个项目下面的任何地方声明一个依赖而不用指定一个版本号。

9.5. 项目关系



使用 **Maven** 的引人注目的原因之一是它使得追踪依赖（以及依赖的依赖）的过程非常容易。当一个项目依赖于另一个项目生成的构件，我们就说这个构件是一个依赖。在 **Java** 项目的情况下，这可以简单到比如一个项目依赖与外部的如 **Log4J** 或 **JUnit** 依赖。依赖可以为外部依赖建模，也可以管理一组相关项目的依赖，如果 `project-a` 依赖于 `project-b`，**Maven** 就能够很聪明的知道 `project-b` 必须在 `project-a` 之前构建。

项目关系不仅仅是依赖以及解决一个项目需要能构建出一个构件。**Maven** 可以建模的关系还包括，某个项目是父项目，某个项目是子模块。本节给你项目中各种关系的概览，并且告诉你如何配置这些关系。

9.5.1. 坐标详解



坐标为一个项目定义一个唯一的位置，它们首先在 [Chapter 3, 一个简单的 Maven 项目](#) 中介绍过。项目使用 Maven 坐标与其它项目关联。一个项目不是简单的依赖于另一个项目，而是一个带有 groupId, artifactId, 和 version 的项目依赖于另一个带有 groupId, artifactId, 和 version 的项目。回顾一下，Maven 坐标有三部分组成：

groupId

一个 groupId 归类了一组相关的构件。组定义符基本上类似于一个 Java 包名。例如：groupId org.apache.maven 是所有由 Apache Maven 项目生成的构件的基本 groupId。组定义符在 Maven 仓库中被翻译成路径，例如，groupId org.apache.maven 可以在 [repo1.maven.org](http://repo1.maven.org/maven2/org/apache/maven) 的 /maven2/org/apache/maven 目录下找到。

artifactId

artifactId 是项目的主要定义符。当你生成一个构件，这个构件将由 artifactId 命名。当你引用一个项目，你就需要使用 artifactId 来引用它。artifactId 和 groupId 的组合必须是唯一的。换句话说，你不能有两个不同的项目拥有同样的 artifactId 和 groupId；在某个特定的 groupId 下，artifactId 也必须是唯一的。

Note

虽然 '.' 在 groupId 中很常用，而你应该避免在 artifactId 中使用它。因为在解析一个完整限定名字至子模块的时候，这会引发问题。

version

当一个构件发布的时候，它是使用一个版本号发布的。该版本号是一个数字定义符如“1.0”，“1.1.1”，或“1.1.2-alpha-01”。你也可以使用所谓的快照（snapshot）版本。一个快照版是一个处于开发过程中的组件的版本，快照版本号通常以 SNAPSHOT 结尾；如，“1.0-SNAPSHOT”，“1.1.1-SNAPSHOT”，和“1-SNAPSHOT”。[Section 9.3.1, “项目版本”](#)介绍了版本和版本界限。

还有第四个，也是最少用到的限定符：

classifier

如果你要发布同样的代码，但是由于技术原因需要生成两个单独的构件，你就要使用一个分类器（classifier）。例如，如果你想要构建两个单独的构件成 JAR，一个使用 Java 1.4 编译器，另一个使用 Java 6 编译器，你就可以使用分类器来生成两个单独的 JAR 构件，它们有同样的 groupId:artifactId:version 组合。如果你的项目使用本地扩展类库，你可以使用分类器为每一个目标平台生成一个构件。分类器常用于打包构件的源码，JavaDoc 或者二进制集合。

当我们在本书说到依赖的时候，我们通常使用如下的简短标志来描述一个依赖：groupId:artifactId:version。要引用 Spring Framework 的 2.5 版本，我们可以使用 org.springframework:spring:2.5。当你要求 Maven 使用 Maven Dependency 插件打印出依赖列表的时候，你也会看到 Maven 倾向于使用这种简短的依赖标志来打印日志信息。

9.5.2. 多模块项目



多模块项目是那些包含一系列待构建模块的项目。一个多模块项目的打包类型总是 pom，很少生成一个构件。一个模块项目的存在只是为了将很多项目归类在一起，成为一个构建。[Figure 9.3, “多模块项目关系”](#)展示了一个项目层次，它包含了两个打包类型为 pom 的父项目，另外三个项目的打包类型是 jar：

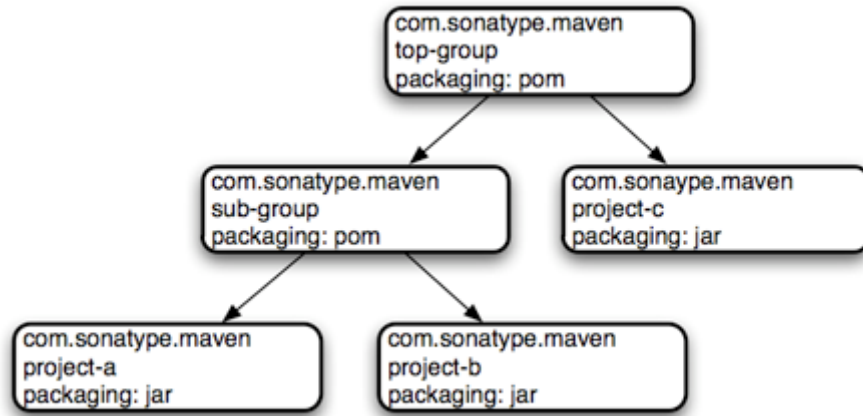


Figure 9.3. 多模块项目关系

文件系统上的目录结构也反映了该模块关系。[Figure 9.3, “多模块项目关系”](#)中的一组项目拥有如下的目录结构：

```
top-group/pom.xml
top-group/sub-group/pom.xml
top-group/sub-group/project-a/pom.xml
top-group/sub-group/project-b/pom.xml
top-group/project-c/pom.xml
```

这些项目相互关联，因为在 POM 中 top-group 和 sub-group 引用了子模块。例如，项目 org.sonatype.mavenbook:top-group 是一个打包类型为 pom 的多模块项目。该项目的 pom.xml 包含如下的 modules 元素：

Example 9.10. top-group 的 modules 元素

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>top-group</artifactId>
  ...
  <modules>
    <module>sub-group</module>
    <module>project-c</module>
  </modules>
  ...
</project>
```

当 Maven 读取 top-group 的 POM 的时候，它会检查它的 `modules` 元素，看到 top-group 引用了项目 sub-group 和 project-c。之后 Maven 会在它们的每个子目录中寻找 pom.xml。Maven 为每一个子模块重复这个过程：它会读取 sub-group/pom.xml 然后看到 sub-group 项目通过如下的 `modules` 元素引用了两个项目。

Example 9.11. sub-group 的 modules 元素

```
<project>
...
<modules>
  <module>project-a</module>
  <module>project-b</module>
</modules>
...
</project>
```

注意我们称多模块项目下的项目为“模块”而不是“子项目”。这是有目的的，是为了而不将由多模块项目归类的项目与那些从其它项目继承 POM 信息的项目混淆。

9.5.3. 项目继承



有些情况你会想要一个项目从父 POM 中继承一些值。你可能正构建一个大型的系统，你不想一遍又一遍的重复同样的依赖元素。如果你的项目通过 `parent` 元素使用继承，你就可以避免这种重复。当一个项目声明一个 `parent` 的时候，它从父项目的 POM 中继承信息。它也可以覆盖父 POM 中的值，或者添加一些新的值。

所有的 Maven POM 从父 POM 中继承值。如果一个 POM 没有通过 `parent` 元素指定一个直接的父项目，那这个 POM 就会从超级 POM 继承值。[Example 9.12, “项目继承”](#)展示了 a-parent 的 `parent` 元素，它继承了 a-parent 项目定义的 POM。

Example 9.12. 项目继承

```
<project>
  <parent>
    <groupId>com.training.killerapp</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
</project>
```

在 project-a 中运行 `mvn help:effective-pom` 显示一个 POM，该 POM 合并了超级 POM，a-parent 中定义的 POM，以及 project-a 中定义的 POM。project-a 显式的和隐式的继承关系如 [Figure 9.4, “a-parent 和 project 的项目继承关系”](#)所示：

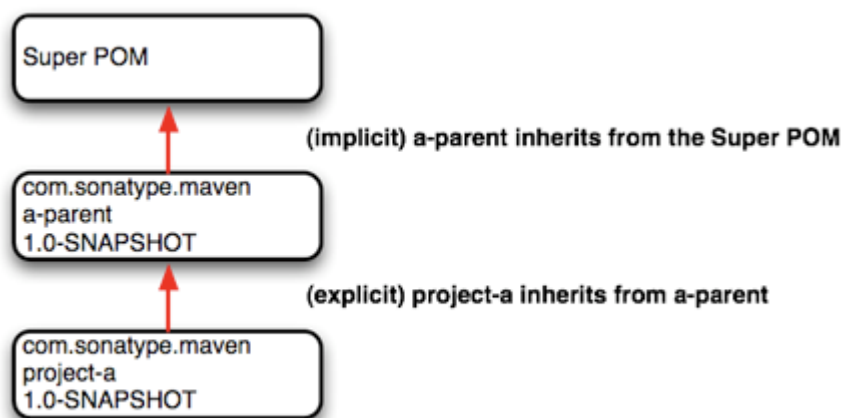


Figure 9.4. a-parent 和 project 的项目继承关系

当一个项目指定一个父项目的时候，Maven 在读取当前项目的 POM 之前，会使用这个父 POM 作为起始点。它继承所有东西，包括 groupId 和 version。你会注意到 project-a 没有指定 groupId 和 version，它们从 a-parent 继承而来。有了 parent 元素，一个 POM 就只需要定义一个 artifactId。但这不是强制的，project-a 可以有一个不同的 groupId 和 version，但如果不提供值，Maven 就会使用在父 POM 中指定的值。如果你开始使用 Maven 来管理和构建大型的多模块项目，你就会常常创建许多共享一组通用的 groupId 和 version 的项目。

当你继承一个 POM，你可以选择直接使用继承的 POM 信息，或者选择覆盖它。以下是一个 Maven POM 从它父 POM 中继承的项目列表：

- 定义符（groupId 和 artifactId 中至少有一个必须被覆盖）
- 依赖
- 开发者和贡献者
- 插件列表
- 报告列表
- 插件执行（id 匹配的执行会被合并）
- 插件配置

当 Maven 继承依赖的时候，它会将父项目中定义的依赖添加到子项目中。你可以使用 Maven 的这一特征来指定一些在所有项目被广泛使用的依赖，让它们从顶层 POM 中继承。例如，如果你的系统全局使用 Log4J 日志框架，你可以在你的顶层 POM 中列出该依赖。任何从该项目继承 POM 信息的项目会自动拥有 Log4J 依赖。类似的，如果你能确定每个项目都在使用同样版本的一个 Maven 插件，你可以在顶层父 POM 的 pluginManagement 元素中显式的列出该 Maven 插件的版本。

Maven 假设父 POM 在本地仓库中可用，或者在当前项目的父目录(../pom.xml) 中可用。如果两个位置都不可用，默认行为还可以通过 relativePath 元素被覆盖。例如，一些组织更喜欢一个平坦的项目结构，父项目的 pom.xml 并不在子项目的父目录中。它可能在项目的兄弟目录中。如果你的子项目在目录 ./project-a 中，父项目在名

为./a-parent 的目录中，你可以使用如下的配置来指定 parent-a 的 POM 的相对位置。

```
<project>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../a-parent/pom.xml</relativePath>
  </parent>
  <artifactId>project-a</artifactId>
</project>
```

9.6. POM 最佳实践



Maven 可以用来管理那些简单的单模块系统，或者复杂到拥有数百个相互关联的子模块的项目。学习 Maven 过程的有一部分不仅仅是弄清楚 Maven 配置的语法，而是学习“Maven 方式”——一组使用 Maven 组织和构建项目的最佳实践。本节试图展现一些这样的知识来帮助你采用最佳实践，你就不用从头开始去 Maven 的邮件列表的数年的内容中寻找这些技巧。

9.6.1. 依赖归类



如果你有一组逻辑上归类在一起的依赖。你可以创建一个打包方式为 pom 项目来将这些依赖归在一起。例如，让我们假设你的应用程序使用 Hibernate，一种流行的对象关系映射框架。所有使用 Hibernate 的项目可能同时依赖于 Spring Framework 和 MySQL JDBC 驱动。你可以创建一个特殊的 POM，它除了声明一组通用依赖之外什么也不做。这样你就不需要在每个使用 Hibernate，Spring 和 MySQL 的项目中包含所有这些依赖。你可以创建一个项目叫做 persistence-deps（持久化依赖的简称），然后让每个需要持久化的项目依赖于这个提供便利的项目。

Example 9.13. 在一个单独的 POM 项目中巩固依赖

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>${hibernateVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
```

```

    <artifactId>hibernate-annotations</artifactId>
    <version>${hibernateAnnotationsVersion}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-hibernate3</artifactId>
    <version>${springVersion}</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysqlVersion}</version>
</dependency>
</dependencies>
<properties>
    <mysqlVersion>(5.1,)</mysqlVersion>
    <springVersion>(2.0.6,)</springVersion>
    <hibernateVersion>3.2.5.ga</hibernateVersion>
    <hibernateAnnotationsVersion>3.3.0.ga</hibernateAnnotationsVersion>
</properties>
</project>

```

如果你在一个名为 persistence-deps 的目录下创建了这个项目，你需要做的只是创建该 pom.xml 并且运行 **mvn install**。由于打包类型是 pom，这个 POM 被安装到你的本地仓库。你现在就可以添加这个项目作为一个依赖，所有该项目的依赖就会被添加到你的项目中。当我们声明一个对于 persistence-deps 项目的依赖的时候，不要忘了指定依赖类型为 pom。

Example 9.14. 声明一个对于 POM 的依赖

```

<project>
  <description>This is a project requiring JDBC</description>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>persistence-deps</artifactId>
      <version>1.0</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>

```

如果之后你决定切换一个不同的 **JDBC** 驱动（比如，**JTDS**），只要替换 `persistence-deps` 项目中的依赖，使用 `sourceforge.jtds:jtds` 而不再是 `mysql:mysql-java-connector`，然后更新版本号。所有依赖于 `persistence-deps` 的项目，如果它们决定更新一个新的依赖版本，就会使用 **JTDS**。巩固相互关联的依赖是一种减少 `pom.xml` 文件长度的很好的方法。如果你需要在项目间共享一组很多的依赖，你也可以建立在项目间建立父子关系，然后将所有共同的依赖重构到父项目中，但是这种父子方式的缺点是一个项目只能有一个父项目。有时候将类似的依赖归类在一起并且使用 `pom` 依赖是更明智的做法。因为这样你的项目就可以根据需要引用很多巩固依赖 **POM**。

Note

当 **Maven** 使用一种“最近者胜出”方式解决依赖的时候，它会用到依赖的深度。当使用上述提到的依赖归类技术的时候，会把依赖推入整个树的更深一层。当在选择用 `pom` 归类依赖或者用父 **POM** 的 `dependenctManagement` 的时候，需要留意这一点。

9.6.2. 多模块 vs. 继承



继承于一个父项目和被一个多模块项目管理是有区别的。一个父项目是指它把所有的值传给它的子项目。一个多模块项目只是说它管理一组子模块，或者说一组子项目。多模块关系从上层往下定义。当建立一个多模块项目的时候，你告诉一个项目它的构建需要包含指定的模块。多模块构建用来将模块聚集到一个单独的构建中。父子关系是从叶节点往上定义的。父子关系更多的是处理一个特定项目的定义。当你给一个子项目关联一个父项目的时候，你告诉 **Maven** 该项目的 **POM** 起源于另一个项目。

为了展示选择继承还是多模块的决策过程，考虑如下的两个例子：用来生成本书的 **Maven** 项目，以及一个包含很多逻辑上同组模块的假想项目。

9.6.2.1. 简单项目



首先，我们看一下 `maven-book` 项目。它的继承和多模块关系如 [Figure 9.5. “maven-book 多模块 vs. 继承”](#)所示。

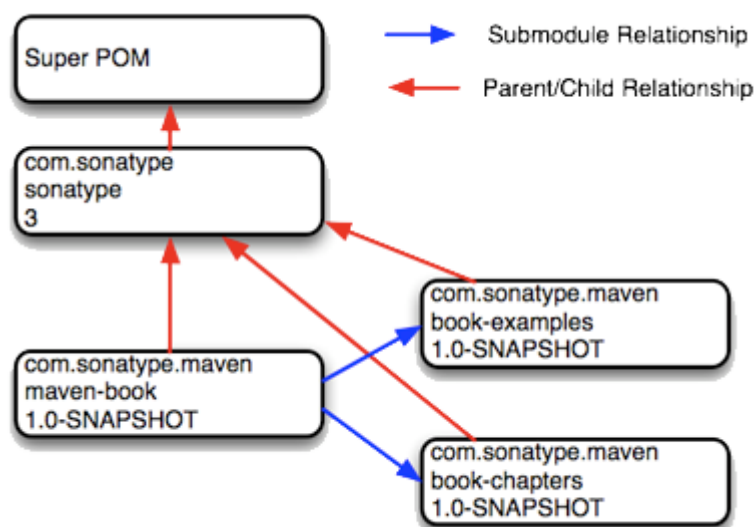


Figure 9.5. maven-book 多模块 vs. 继承

当我们构建你正阅读的 **Maven** 书的时候,我们在名为 `maven-book` 的项目下运行 **mvn package**。该多模块项目包含两个子模块: `book-examples` 和 `book-chapters`。两者都共享同样的父项目,它们只通过同为 `maven-book` 子项目的方式关联。

`book-examples` 构建了可下载的本书样例的 **ZIP** 和 **TGZ** 存档文件。当我们在 `book-examples/` 目录使用 **mvn package** 运行 `book-examples` 的构建的时候,它完全不知道它是 `maven-book` 项目的一部分。`book-examples` 完全不关心 `maven-book`,它知道的是它的父项目是最顶层的 **sonatype POM**,它自身创建样例的归档文件。该例中, `maven-book` 的存在只是为了方便聚集模块构建。

这些书的项目没有定义一个父项目。所有这个三个项目: `maven-book`, `book-examples`, `book-chapters` 和都继承同一个共享的“团体”父项目——**sonatype**。这是一种采用 **Maven** 的组织中常见的实践,一些组织定义一个顶层的团体 **POM**,作为一个默认的父项目为其它项目服务,而不是让每个项目默认去扩展超级 **POM**。在这个书本样例中,并不是一定要让 `book-examples` 和 `book-chapters` 共享同样的父 **POM**,它们是完全不同的两个项目,拥有完全不同的而依赖,不同的构建配置,使用极为不同的插件创建你正阅读的内容。“团体”**POM** 能让组织有机会自定义一些 **Maven** 的默认行为,提供一些组织特定的信息,如配置部署设置和构建 **profile**。

9.6.2.2. 多模块企业级项目



让我们看一下另一个例子,它提供了现实项目中继承和多模块关系存在的更准确的画面。[Figure 9.6, “企业级多模块 vs. 继承”](#)展示了类似于典型企业应用中的一组项目。有一个的公司顶层 **POM**,其 `artifactId` 值为 `sonatype`。有一个名为 `big-system` 的多模块项目,引用了子模块 `server-side` 和 `client-side`。

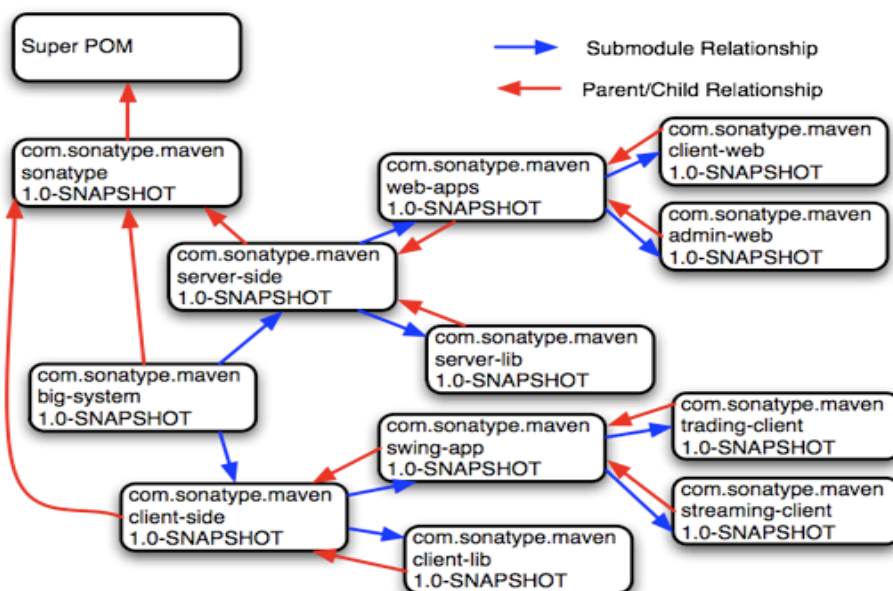


Figure 9.6. 企业级多模块 vs. 继承

这里到底是怎么回事呢？让我们尝试着给这一组混乱的箭头解构。首先，看一下 big-system。这个 big-system 可能就是你要运行 **mvn package** 以构建并测试整个系统的地方。big-system 引用了子模块 client-side 和 server-side。这两个项目都管理了大量运行在服务端或者客户端的代码。让我们看一下 server-side 项目。在 server-side 下面有一个名为 server-lib 的项目和一个名为 web-apps 的多模块项目。在 web-apps 下面有两个 **Java web** 应用：client-web 和 admin-web。

让我们从 client-web 和 admin-web 到 web-apps 开始讨论父子关系。由于这两个 **web** 应用都用同样的 **web** 应用框架实现（假设是 **Wicket**），两个项目都共享同样的一组核心依赖。对 **Servlet API**，**JSP API**，和 **Wicket** 的依赖可以放到 web-apps 项目中。client-web 和 admin-web 都需要依赖 server-lib，它就可以定义为一个 web-apps 和 server-lib 之间的依赖。因为 client-web 和 admin-web 通过继承 web-apps 共享了如此多的配置，它们的 **POM** 很小，只包含定义符，父项目声明和最终构建名称。

本例中，使用父子关系的最主要原因是为了给一组逻辑关联的项目共享依赖和通用配置。所有 big-system 下的项目都通过子模块与其它项目关联，但是并不是所有子模块都被配置成指回该父项目。所有模块都是子模块是为了方便，要构建整个系统，只要到 big-system 项目目录下运行 **mvn package**。再仔细看下图，你会发现在 server-side 和 big-system 之间没有父子关联。这是为什么？**POM** 继承十分有用，但它可能被滥用。当然在需要共享依赖和配置的时候，父子关联需要被使用。但当两个项目截然不同的时候使用父子关联是不明智的。举个例子，server-side 和 client-side 项目。在系统中，让 server-side 和 client-side 都从 big-system 继承通用的 **POM** 是可能的，但一旦这两个子项目的重大分歧出现，你就需要费脑子一方面将构建配置抽离到 big-system 中，另一方面又需要不影响其它的子项目。即使 server-side 和 client-side 同时依赖于 **Log4J**，它们也可能拥有截然不同的插件配置。

有时候基于风格和经验，为了允许项目如 server-side 和 client-side 保持完全独立，少量的重复配置是最小的代价。设计一组继承了五六层 **POM** 的第三方项目永远都不是一个好主意。这样的配置下，你可能不再需要在多个地方重复 **Log4J** 依赖，但你会需要查看五六个 **POM** 来弄清 **Maven** 如何计算出你的有效 **POM**。所有的这些新的复杂度只是为了避免五行依赖声明。在 **Maven** 中，有一种“**Maven** 方式”，但是也有很多其它方式完成同样的事情。这都可归结为一种偏好和风格。大部分情况下，如果你的子模块定义了往回的父项目引用，不会出什么问题，但是你的 **Maven** 使用情况一直在变。

9.6.2.3. 原型父项目



如 [Figure 9.7](#), “为特定的项目使用父项目作为“原型””的例子所示, 这是又一种假想的创造性的方式, 使用继承和多模块构建达到重用依赖的目的。

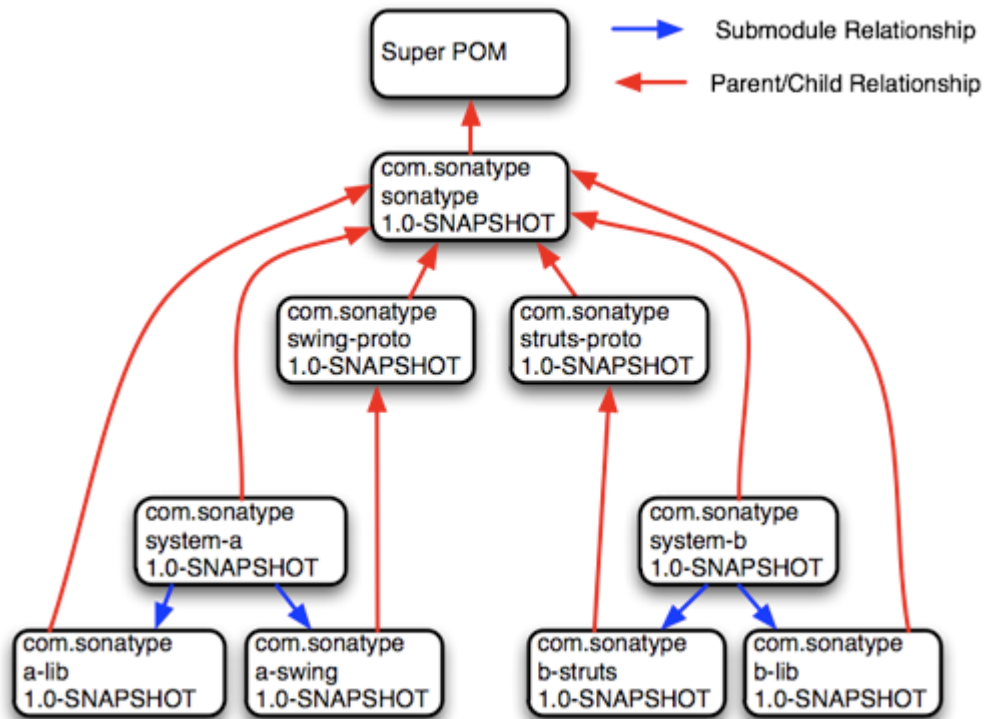


Figure 9.7. 为特定的项目使用父项目作为“原型”

在该例中, 你有两个截然不同的系统: system-a 和 system-b, 各自定义独立的应用。system-a 定义了两个模块 a-lib 和 a-swing。system-a 和 a-lib 两者都定义了顶层的 sonatype POM 作为父项目, 但 a-swing 项目定义了 swing-proto 作为它的父项目。在该系统中, swing-proto 为所有 Swing 应用程序提供了一个基础 POM, 而 struts-proto 为所有 Struts 2 web 应用程序提供了一个基础 POM。sonatype POM 提供了高层的信息如 groupId, 组织信息和构建 profile, struts-proto 定义了所有创建 struts 应用需要的依赖。如果你的开发根据不同的应用有不同的特征, 每类应用又需要遵循同样一组规则, 那么这里介绍的方法很有用。如果你正创建很多 struts 应用, 但是它们很少相互关联, 你可能只需要在 struts-proto 中定义所有通用的东西。这种方式的缺点是有不能在 system-a 和 system-b 项目层次中使用父子关系来共享如开发人员和其它构建配置信息。一个项目只能有一个父项目。

这种方法的另外一个缺点是, 一旦你有一个项目需要“破坏该模型”, 你需要重写父 POM, 或者想办法将自定义信息提取到一个共享的父项目中, 而不让这些自定义信息影响所有子项目。总得来说, 为特定的项目“类型”使用 POM 作为原型不是一种推荐的做法。

Chapter 10. 构建生命周期

[10.1. 简介](#)

[10.1.1. 清理生命周期 \(clean\)](#)

[10.1.2. 默认生命周期 \(default\)](#)

[10.1.3. 站点生命周期 \(site\)](#)

[10.2. 打包相关生命周期](#)

[10.2.1. JAR](#)

[10.2.2. POM](#)

[10.2.3. Maven Plugin](#)

[10.2.4. EJB](#)

[10.2.5. WAR](#)

[10.2.6. EAR](#)

[10.2.7. 其它打包类型](#)

[10.3. 通用生命周期目标](#)

[10.3.1. Process Resources](#)

[10.3.2. Compile](#)

[10.3.3. Process Test Resources](#)

[10.3.4. Test Compile](#)

[10.3.5. Test](#)

[10.3.6. Install](#)

[10.3.7. Deploy](#)

10.1. 简介



Maven 使用 **POM** 描述项目，将其建模成一些名词。**POM** 记录了一个项目的定义：项目包含什么？需要怎样的打包类型？是否包含一个父项目？它的依赖是什么？我们已经在前面的章节展示了如何描述一个项目，但还没有介绍一种允许 **Maven** 针对这些对象进行操作的机制。在 **Maven** 中这些“动词”是由 **Maven** 插件包装的一些目标，它们绑定到一个构建生命周期的阶段中。一个 **Maven** 生命周期包含了一些有序的命名阶段：**prepare-resources**, **compile**, **package**, 和 **install** 以及其它。有个阶段抽象了编译过程，有个阶段抽象了打包过程。而那些 **pre-**和 **post-**阶段可以用来注册一些必须在某些特定阶段之前或之后运行的目标。当你让 **Maven** 构建一个项目的时候，你其实是让它一步步通过那些预定义的有序的阶段，并且运行所有注册到某个特定阶段的目标。一个构建生命周期是一组精心组织的有序的阶段，它的存在能使所有注册的目标变得有序运行。这些目标根据项目的打包类型被选择并绑定。**Maven** 中有三种标准的生命周期：清理（**clean**），默认（**default**）（有时候也称为构建），和站点（**site**）。本章，我们将学习 **Maven** 如何将目标绑定到生命周期，生命周期如何自定义。你同时也会学到默认生命周期阶段的知识。

10.1.1. 清理生命周期 (clean)



第一个你将感兴趣的生命周期是 **Maven** 中最简单的生命周期。运行 **mvn clean** 将调用清理生命周期，它包含了三个生命周期阶段：

- pre-clean
- clean
- post-clean

在清理生命周期中最有意思的阶段是 clean 阶段。**Clean** 插件的 **clean** 目标

(**clean:clean**) 被绑定到清理生命周期中的 clean 阶段。目标 **clean:clean** 通过删除构建目录删除整个构建的输出。如果你没有自定义构建目录位置，那么构建目录就是定义在超级 **POM** 中的 `${basedir}/target`。当你运行 **clean:clean** 目标的时候你并不是直接运行 **mvn clean:clean**，你可以通过执行清理生命周期的 clean 阶段运行该目标。运行 clean 阶段能让 **Maven** 有机会执行其它可能被绑定到 pre-clean 阶段的目标。

例如，假设你想要在 pre-clean 的时候触发一个 **antrun:run** 目标任务来输出一个通知，或者需要在项目构建目录被删除之前将其归档。简单的运行 **clean:clean** 目标不会完整的执行该生命周期，但是指定 clean 阶段就能使用 clean 生命周期，并且逐个的经过生命周期阶段，直到到达 clean 阶段。[Example 10.1, “在 pre-clean 阶段触发一个目标”](#)展示了一个样例，在它的构建配置中，绑定了 **antrun:run** 至 pre-clean 阶段，输出一个警告告诉用户项目构件即将被删除。该例中，**antrun:run** 目标被用来执行一些随意的 **Ant** 命令来检查项目的构件。如果项目的构件将要被删除，它会打印该信息至屏幕。

Example 10.1. 在 pre-clean 阶段触发一个目标

```
<project>
...
<build>
  <plugins>... <plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <id>file-exists</id>
      <phase>pre-clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <!-- adds the ant-contrib tasks (if/then/else used below) -->
          <taskdef resource="net/sf/antcontrib/antcontrib.properties" />
          <available
```

```

file="${project.build.directory}/${project.build.finalName}.${project.packaging}"

        property="file.exists" value="true" />

    <if>
        <not>
            <isset property="file.exists" />
        </not>
        <then>
            <echo>No
                ${project.build.finalName}.${project.packaging} to
                delete</echo>
        </then>
        <else>
            <echo>Deleting
                ${project.build.finalName}.${project.packaging}</echo>
        </else>
    </if>
</tasks>
</configuration>
</execution>
</executions>
<dependencies>
    <dependency>
        <groupId>ant-contrib</groupId>
        <artifactId>ant-contrib</artifactId>
        <version>1.0b2</version>
    </dependency>
</dependencies>
</plugin>
</plugins>
</build>
</project>

```

在带有如上构建配置的项目中运行 **mvn clean** 会生成如下的输出：

```

[INFO] Scanning for projects...
[INFO]
-----
[INFO] Building Your Project
[INFO]    task-segment: [clean]
[INFO]
-----

```

```

[INFO] [antrun:run {execution: file-exists}]
[INFO] Executing tasks
      [echo] Deleting your-project-1.0-SNAPSHOT.jar
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory ~/corp/your-project/target
[INFO] Deleting directory ~/corp/your-project/target/classes
[INFO] Deleting directory ~/corp/your-project/target/test-classes
[INFO]

-----

[INFO] BUILD SUCCESSFUL
[INFO]

-----

[INFO] Total time: 1 second
[INFO] Finished at: Wed Nov 08 11:46:26 CST 2006
[INFO] Final Memory: 2M/5M
[INFO]

-----

```

除了在 pre-clean 阶段配置 **Maven** 去运行一个目标，你也可以自定义 **Clean** 插件去删除构建输出目录以外的文件。你可以配置该插件去删除那些在 `fileSet` 中指定的文件。下面的样例配置了 **Clean** 插件，使用标准的 **Ant** 文件通配符：* 和 **，删除所有 target-other/ 目录中的 .class 文件。

Example 10.2. 自定义 Clean 插件的行为

```

<project>
  <modelVersion>4.0.0</modelVersion>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <configuration>
          <filesets>
            <fileset>
              <directory>target-other</directory>
              <includes>
                <include>*.class</include>
              </includes>
            </fileset>
          </filesets>
        </configuration>
      </plugin>
    </plugins>
  </build>

```

```
</project>
```

10.1.2. 默认生命周期 (default)



大部分 Maven 用户将会对默认生命周期十分熟悉。它是一个软件应用程序构建过程的总体模型。第一个阶段是 validate，最后一个阶段是 deploy。这些默认 Maven 生命周期的阶段如 [Table 10.1, “Maven 默认生命周期阶段”](#)所示：

Table 10.1. Maven 默认生命周期阶段

生命周期阶段	描述
validate	验证项目是否正确，以及所有为了完整构建必要的信息是否可用
generate-sources	生成所有需要包含在编译过程中的源代码
process-sources	处理源代码，比如过滤一些值
generate-resources	生成所有需要包含在打包过程中的资源文件
process-resources	复制并处理资源文件至目标目录，准备打包
compile	编译项目的源代码
process-classes	后处理编译生成的文件，例如对 Java 类进行字节码增强（bytecode enhancement）
generate-test-sources	生成所有包含在测试编译过程中的测试源码
process-test-sources	处理测试源码，比如过滤一些值
generate-test-resources	生成测试需要的资源文件
process-test-resources	复制并处理测试资源文件至测试目标目录
test-compile	编译测试源码至测试目标目录
test	使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布
prepare-package	在真正的打包之前，执行一些准备打包必要的操作。这通常会产生一个包的展开的处理过的版本（将会在 Maven 2.1+ 中实现）
package	将编译好的代码打包成可分发的格式，如 JAR，WAR，或者 EAR
pre-integration-test	执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境
integration-test	如果有必要的话，处理包并发布至集成测试可以运行的环境
post-integration-test	执行一些在集成测试运行之后需要的动作。如清理集成测试环境。
verify	执行所有检查，验证包是有效的，符合质量规范

生命周期阶段	描述
install	安装包至本地仓库，以备本地的其它项目作为依赖使用
deploy	复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关）

10.1.3. 站点生命周期 (site)



Maven 不仅仅能从一个项目构建软件构件，它还能为一个或者一组项目生成项目文档和报告。项目文档和站点生成有一个专有的生命周期，它包含了四个阶段：

- pre-site
- site
- post-site
- site-deploy

默认绑定到站点生命周期的目标是：

- site - site:site
- site-deploy -site:deploy

打包类型通常不更改此生命周期，因为打包类型主要和构件创建有关，和站点生成没有太大的关系。**Site** 插件触发 [Doxia](#) 执行文档生成，以及执行其它报告生成插件。你可以通过运行如下命令从一个 **Maven** 项目生成一个站点：

```
$ mvn site
```

有关更多的 **Maven** 站点生成信息，查看 [Chapter 15. Site Generation](#)。

10.2. 打包相关生命周期



绑定到每个阶段的特定目标默认根据项目的打包类型设置。一个打包类型为 jar 的项目和一个打包类型为 war 的项目拥有不同的两组默认目标。packaging 元素影响构建一个项目需要的步骤。举个打包如何影响构建的例子，考虑有两个项目：一个打包类型是 pom，另外一个为 jar。在 package 阶段，打包类型为 pom 的项目会运行 site:attach-descriptor 目标，而打包类型为 jar 的项目会运行 jar:jar 目标。

下面的小节描述了 **Maven** 中内建打包类型的生命周期。可以使用这些小节来找出哪些默认目标映射到了哪些默认生命周期阶段。

10.2.1. JAR



JAR 是默认的打包类型，是最常用的，因此也就是生命周期配置中最经常遇到的打包类型。JAR 生命周期默认的目标如 [Table 10.2, “JAR 打包默认的目标”](#)所示：

Table 10.2. JAR 打包默认的目标

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

10.2.2. POM



POM 是最简单的打包类型。不像一个 JAR，SAR，或者 EAR，它生成的构件只是它本身。没有代码需要测试或者编译，也没有资源需要处理。打包类型为 POM 的项目的默认目标如 [Table 10.3, “POM 打包默认的目标”](#)所示：

Table 10.3. POM 打包默认的目标

生命周期阶段	目标
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

10.2.3. Maven Plugin



该打包类型和 JAR 打包类型类似，除了三个目标：plugin:descriptor，plugin:addPluginArtifactMetadata，和 plugin:updateRegistry。这些目标生成一个描述文件，对仓库数据执行一些修改。打包类型为 maven-plugin 的项目的默认目标如 [Table 10.4, “maven-plugin 打包默认的目标”](#)所示。

Table 10.4. maven-plugin 打包默认的目标

生命周期阶段	目标
generate-resources	plugin:descriptor

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar, plugin:addPluginArtifactMetadata
install	install:install, plugin:updateRegistry
deploy	deploy:deploy

10.2.4. EJB



EJB，或者说企业 Java Bean，是企业级 Java 中模型驱动开发的常见数据访问机制。Maven 提供了对 EJB 2 和 3 的支持。你必须配置 EJB 插件来为 EJB3 指定打包类型，否则该插件默认认为 EJB 为 2.1，并寻找某些 EJB 配置文件是否存在。打包类型为 EJB 的项目的默认目标如 [Table 10.5, “EJB 打包默认的目标”](#)所示。

Table 10.5. EJB 打包默认的目标

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
install	install:install
deploy	deploy:deploy

10.2.5. WAR



WAR 打包类型和 JAR 以及 EJB 类似。例外是这里的 package 目标是 war:war。注意 war:war 插件需要一个 web.xml 配置文件在项目的 src/main/webapp/WEB-INF 目录中。打包类型为 WAR 的项目的默认目标如 [Table 10.6, “WAR 打包默认的目标”](#)所示。

Table 10.6. WAR 打包默认的目标

生命周期阶段	目标
--------	----

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

10.2.6. EAR



EAR 可能是最简单的 Java EE 结构体，它主要包含一个部署描述符 application.xml 文件，一些资源和一些模块。EAR 插件有个名为 generate-application-xml 的目标，它根据 EAR 项目 POM 的配置生成 application.xml。打包类型为 EAR 的项目的默认目标如 [Table 10.7, “EAR 打包默认的目标”](#)所示。

Table 10.7. EAR 打包默认的目标

生命周期阶段	目标
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

10.2.7. 其它打包类型



以上列表并非是 Maven 中所有可用打包类型。有许多打包格式在外部的项目和插件中可用：NAR（本地归档）打包类型，用来生成 Adobe Flash 和 Flex 内容的项目的 SWF 和 SWC 打包类型，以及很多其它类型。你也可以自定义打包类型，定制默认的生命周期目标来适应你自己项目的打包需求。

为了使用自定义的打包类型，你需要两样东西：一个定义了定制打包类型生命周期的插件，和一个包含该插件的仓库。有些定制打包类型是由中央 Maven 仓库中可用的插件定义的。这里有一个样例项目，它引用了 Israfel Flex 插件，使用自定义打包类型 SWF 根据 Adobe Flex 生成输出。

Example 10.3. 为 Adobe Flex (SWF)定制打包类型

```
<project>
...
<packaging>swf</packaging>
...
<build>
  <plugins>
    <plugin>
      <groupId>net.israfil.mojo</groupId>
      <artifactId>maven-flex2-plugin</artifactId>
      <version>1.4-SNAPSHOT</version>
      <extensions>true</extensions>
      <configuration>
        <debug>true</debug>
        <flexHome>${flex.home}</flexHome>
        <useNetwork>true</useNetwork>
        <main>org/sonatype/mavenbook/Main.mxml</main>
      </configuration>
    </plugin>
  </plugins>
</build>
...
</project>
```

在???中，我们向你展示了如何使用自定义的生命周期创建你自己的打包类型。该样例应该能让你了解到，引用一个定制的打包类型你需要做些什么。你需要做的只是引用那个提供定制打包类型的插件。Israfil Flex 插件是托管在 Google Code 的第三方 Maven 插件，要了解更多的关于此插件的信息，以及如何使用 Maven 编译 Adobe Flex，访问 <http://code.google.com/p/israfil-mojo>。该插件为 SWF 打包类型提供了如下的生命周期。

Table 10.8. SWF 打包的默认生命周期

生命周期阶段	目标
compile	flex2:compile-swc
install	install
deploy	deploy

10.3. 通用生命周期目标



很多打包类型的生命周期有类似的目标。如果你看一下绑定到 **WAR** 和 **JAR** 生命周期的目标，你会发现它们只有在 package 阶段有区别。**WAR** 生命周期的 package 阶段调用了 war:war，而 **JAR** 生命周期的 package 阶段调用了 jar:jar。大部分你将要接触的生命周期共享一些通用生命周期目标，用来管理资源，运行测试，以及编译源代码。本节，我们会详细讨论这些通用的生命周期目标。

10.3.1. Process Resources



大部分生命周期将 resources:resources 目标绑定到 process-resources 阶段。process-resources 阶段处理资源并将资源复制到输出目录。如果你没有自己自定义超级 POM 中的默认目录位置，**Maven** 就会将 `${basedir}/src/main/resources` 中的文件复制到 `${basedir}/target/classes`，或者是由 `${project.build.outputDirectory}` 定义的目录。除了复制资源文件至输出目录，**Maven** 同时也会在资源上应用过滤器，能让你替换资源文件中的一些符号。就像在 POM 中我们通过 `${...}` 标记引用变量一样，你也可以使用同样的语法在你项目的资源文件中引用变量。与 profile 联系起来，这样的特性就能用来生成针对不同部署平台的构件。当我们需要为同一个项目的开发，测试，**staging**，以及产品平台环境生成输出的时候，该特性就十分有用。要了解更多的关于构建 profile 的信息，查看 [Chapter 11, 构建 Profile](#)。

为了阐述资源过滤，假设你有个带有 XML 文件 `src/main/resources/META-INF/service.xml` 的项目。你想要提取出一些配置变量至一个属性文件。换句话说，你可能想要为你的数据库引用 **JDBC URL**，用户名和密码，并且你不想将这些值直接放到 `service.xml` 文件里，而是想要使用一个属性文件来存储你程序中的所有配置点。这么做能让你将所有配置信息固定到单独的一个属性文件中，当你需要面对一个新的部署环境的时候，就很容易更改配置的值。首先，看一下 `src/main/resources/META-INF/service.xml` 的内容。

Example 10.4. 在项目资源中使用属性

```
<service>
  <!-- This URL was set by project version ${project.version} -->
  <url>${jdbc.url}</url>
  <user>${jdbc.username}</user>
  <password>${jdbc.password}</password>
</service>
```

该 XML 文件使用你在 POM 中用到的同样的属性引用语法，第一个引用的变量是 `project`，它同时也是 POM 中的隐式变量。`project` 变量提供了对 POM 信息的访问。接下来的三个变量引用是，`jdbc.url`，`jdbc.username` 和 `jdbc.password`。这些自定义的变量在一个属性文件 `src/main/filters/default.properties` 中定义。

Example 10.5. src/main/filters 中的 default.properties

```
jdbc.url=jdbc:hsqldb:mem:mydb
jdbc.username=sa
jdbc.password=
```

要配置使用该 default.properties 文件的资源过滤，我们需要在这个项目的 POM 中指定两样东西：构建配置的 filters 元素中的属性文件列表，以及一个标记告诉 Maven 资源目录需要过滤。默认的 Maven 行为会跳过过滤，只是将资源复制到输出目录；你需要显式的配置资源过滤，否则 Maven 就会置之不理。这种 Maven 资源过滤的默认行为是为了确保不让 Maven 替换掉一些你不想替换的 \${...} 引用。

Example 10.6. 过滤资源（替换属性）

```
<build>
  <filters>
    <filter>src/main/filters/default.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

正如 Maven 中所有目录一样，资源目录并非一定要在 src/main/resources。这只是定义在超级 POM 中的默认值。你应该也注意到你不需要将你所有的资源合并到一个单独的目录中。你可以将资源分离至 src/main 目录下的独立的目录中。假设你有个项目包含了数百个 XML 文档和数百个图片。你可能希望创建两个目录 src/main/xml 和 src/main/images 来存储这些内容，而不是将它们混合在 src/main/resources 目录中。为了添加资源目录列表，你需要在你的构建配置中加入如下的 resource 元素：

Example 10.7. 配置额外的资源目录

```
<build>
  ...
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>
      <directory>src/main/xml</directory>
    </resource>
    <resource>
      <directory>src/main/images</directory>
    </resource>
  </resources>
</build>
```

```
    </resource>
  </resources>
  ...
</build>
```

当你构建一个项目用来生成控制台程序或者命令行工具的时候，你通常发现自己正编写一个 `shell` 脚本，需要引用构建生成的 `JAR`。当你使用 `assembly` 插件为一个应用程序生成如 `ZIP` 或 `TAR` 的分发包的时候，你可能会将所有的脚本放到如 `src/main/command` 的目录下。在下面的 `POM` 资源配置中，你会看到我们如何使用资源过滤器和一个对项目变量的引用，生成 `JAR` 的最终名称。要了解更多的关于 `Maven Assembly` 插件的信息，参考 [Chapter 12, Maven Assemblies](#)。

Example 10.8. 过滤脚本资源

```
<build>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple-cmd</artifactId>
  <version>2.3.1</version>
  ...
  <resources>
    <resource>
      <filtering>true</filtering>
      <directory>${basedir}/src/main/command</directory>
      <includes>
        <include>run.bat</include>
        <include>run.sh</include>
      </includes>
      <targetPath>${basedir}</targetPath>
    </resource>
    <resource>
      <directory>${basedir}/src/main/resources</directory>
    </resource>
  </resources>
  ...
</build>
```

如果你在该项目中运行 `mvn process-resources`，最后你会在 `${basedir}` 中得到两个文件，`run.sh` 和 `run.bat`，我们在 `resource` 元素中挑选出这两个文件，配置过滤器，然后设置 `targetPath` 为 `${basedir}`。在第二个 `resource` 元素中，我们配置了默认资源路径，使其在不做过滤的情况下被复制到默认输出目录。[Example 10.8, “过滤脚本资源”](#) 告诉你如何声明两个资源目录，如何给它们提供不同的过滤配置和目标目录配置。[Example 10.8, “过滤脚本资源”](#) 项目的 `src/main/command` 目录下包含了一个含有如下内容的 `run.bat` 文件。

```
@echo off
```

```
java -jar ${project.build.finalName}.jar %*
```

在运行了 **mvn process-resources** 之后，目录\${basedir}下会有一个含有如下内容的名为 run.bat 的文件：

```
@echo off
java -jar simple-cmd-2.3.1.jar %*
```

在带有很多不同种类资源的复杂的项目中，我们会发现将资源分离到多个目录下十分有利，原因之一就是我们可以为特定的资源子集自定义过滤器。针对不同的过滤需求，除了将资源存储到不同的目录下，我们还可以使用更复杂的一组包含/排除模式来匹配那些符合模式的资源文件。

10.3.2. Compile



大部分生命周期将 **Compiler** 插件的 compile 目标绑定到 compile 阶段。该阶段会调用 compile:compile，后者被配置成编译所有的源码并复制到构建输出目录。如果你没有自定义超级 POM 中的值，compile:compile 将会编译 src/main/java 中的所有内容至 target/classes。**Compiler** 插件调用 javac，使用的 **source** 设置为 1.3，默认 **target** 设置为 1.1。换句话说，**Compiler** 插件会假设你所有的 Java 源代码遵循 Java 1.3，目标为 Java 1.1 JVM。如果你想要更改这些设置，你需要在 POM 中为 **Compiler** 插件提供 source 和 target 配置，如 [Example 10.9, “为 Compiler 插件设置 source 和 target 版本”](#)所示。

Example 10.9. 为 Compiler 插件设置 source 和 target 版本

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

要注意我们配置的是 **Compiler** 插件，而不是 compile:compile 目标。如果我们只要为 compile:compile 目标配置 **source** 和 **target**，就要将 configuration 元素放到

compile:compile 目标的 execution 元素下。我们为整个插件配置 source 和 target, 是因为 compile:compile 并不是我们唯一我们感兴趣配置的目标。当 Maven 使用 compile:testCompile 目标编译测试代码的时候, Compiler 插件会被重用, 因此在插件级别配置 source 和 target, 一次就能配置该插件所有的目标。

如果你想要自定义源码的位置, 你也可以更改构建配置。如果你想要存储项目的源码至 src/java 而非 src/main/java, 让构建输出至 classes 而非 target/classes, 你可以覆盖定义在超级 POM 中的 sourceDirectory 的默认值。

Example 10.10. 覆盖默认的源码和输出目录

```
<build>
...
<sourceDirectory>src/java</sourceDirectory>
<outputDirectory>classes</outputDirectory>
...
</build>
```

Warning

虽然让 Maven 屈服于你自己的项目目录结构可能看起来很有必要, 但我们还是要不断强调你应该牺牲自己关于目录结构的想法, 而遵循 Maven 的默认值。这不是说我们要给你洗脑, 要你接受 Maven 的方式, 这是因为如果你的项目遵循大部分的约定, 别人会很容易理解你的项目。因此忘掉你自己的想法, 别那样做。

10.3.3. Process Test Resources



process-test-resources 阶段最难和 process-resources 阶段区别。在 POM 中它们有一些微小的差别, 但大部分是一样的。你可以像过滤一般的资源那样过滤测试资源。测试资源的默认位置定义在超级 POM 中, 为 src/test/resources, 默认的输出目录为 target/test-classes, 由 \${project.build.testOutputDirectory} 定义。

10.3.4. Test Compile



test-compile 阶段基本上和 compile 阶段一致。唯一的不同是会调用 compile:testCompile 编译测试源代码目录至测试构建输出目录。如果你没有在超级 POM 中自定义默认目录, compile:testCompile 将会编译 src/test/java 中的源码至 target/test-classes 目录。

类似源代码目录, 如果你想要自定义测试源码目录和测试编译输出目录的位置, 你可以覆盖 testSourceDirectory 和 testOutputDirectory。如果你想要将测试源代码存储在 src-test/而非 src/test/java, 保存测试字节码至 classes-test/而非 target/test-classes, 你可以使用如下的配置:

Example 10.11. 覆盖测试源码和输出的位置

```
<build>
...
<testSourceDirectory>src-test</testSourceDirectory>
<testOutputDirectory>classes-test</testOutputDirectory>
...
</build>
```

10.3.5. Test



大部分生命周期绑定 Surefire 插件的 test 目标至 test 阶段。Surefire 插件是 Maven 的单元测试插件，Surefire 默认的行为是寻找测试源码目录下所有以 *Test 结尾的类，以 [JUnit](#) 测试的形式运行它们。Surefire 插件也可以配置成运行 [TestNG](#) 单元测试。

运行过 `mvn test` 之后，你应该注意到 Surefire 在 `target/surefire-reports` 目录生成了许多报告。该目录内每个 Surefire 插件运行过的测试都会有相关的两个文件：一个是包含测试运行信息的 XML 文档，另一个是包含单元测试输出的文本文件。如果测试阶段有问题，单元测试失败了，你可以使用 Maven 的输出以及该目录下的内容来追查测试失败的原因。在站点生成的时候，`surefire-reports/` 目录的内容会被用来创建报告，使项目所有单元测试的总体情况清晰明了。

如果你工作的项目有一些失败的单元测试，同时你想让项目生成输出，你需要配置 Surefire 插件在遇到失败的情况下继续一个构建。当遇到单元测试失败的时候，默认行为是停止构建。要覆盖这种行为，你需要设置 Surefire 插件的 `testFailureIgnore` 配置属性为 `true`。

Example 10.12. 配置 Surefire 忽略单元测试失败

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
    ...
  </plugins>
</build>
```

如果你想要整个的跳过测试，你可以运行如下的命令：

```
$ mvn install -Dmaven.test.skip=true
```

maven.test.skip 变量同时控制 **Compiler** 和 **Surefire** 插件，如果你传入 maven.test.skip，就等于告诉 **Maven** 整个的跳过测试。

10.3.6. Install



Install 插件的 **install** 目标基本上都是绑定到 **install** 生命周期阶段。
install:install 目标只不过是项目的主要构件安装到本地仓库。如果你有一个项目，groupId 是 **org.sonatype.mavenbook**，artifactId 是 **simple-test**，version 是 **1.0.2**，那么 **install:install** 目标就会从 **target/simple-test-1.0.2.jar** 复制 **JAR** 文件至
~/.m2/repository/org/sonatype/mavenbook/simple-test/1.0.2/simple-test-1.0.2.jar。如果这个项目的打包类型是 **POM**，那么该目标就仅仅复制 **POM** 到本地仓库。

10.3.7. Deploy



Deploy 插件的 **deploy** 目标通常绑定到 **deploy** 生命周期阶段。该阶段用来将一个构件部署到远程 **Maven** 仓库，当你执行一次发布的时候通常需要更新远程仓库。一次部署过程可以简单到复制一个文件至另外一个目录，或者复杂到使用公钥通过 **SCP** 传送一个文件。部署设置通常包含远程仓库的证书，并且，这样的部署设置通常不会存储在 **pom.xml** 中。部署设置通常可以在用户单独的 ~/.m2/settings.xml 中找到。到现在为止，你要知道的是 **deploy:deploy** 被绑定到 **deploy** 阶段，它会处理传送一个构件至发布仓库，更新一些可能被此次部署影响的仓库信息。

Chapter 11. 构建 Profile

[11.1. Profile 是用来做什么的?](#)

[11.1.1. 什么是构建可移植性](#)

[11.1.1.1. 不可移植构建](#)

[11.1.1.2. 环境可移植性](#)

[11.1.1.3. 组织（内部）可移植性](#)

[11.1.1.4. 广泛（全局）可移植性](#)

[11.1.2. 选择一个适当级别的可移植性](#)

[11.2. 通过 Maven Profiles 实现可移植性](#)

[11.2.1. 覆盖一个项目对象模型](#)

[11.3. 激活 Profile](#)

[11.3.1. 激活配置](#)

[11.3.2. 通过属性缺失激活](#)

[11.4. 外部 Profile](#)

[11.5. Settings Profile](#)

[11.5.1. 全局 Settings Profile](#)

[11.6. 列出活动的 Profile](#)

[11.7. 提示和技巧](#)

[11.7.1. 常见的环境](#)

[11.7.2. 安全保护](#)

[11.7.3. 平台分类器](#)

[11.8. 小结](#)

11.1. Profile 是用来做什么的？



Profile 能让你为一个特殊的环境自定义一个特殊的构建；profile 使得不同环境间构建的可移植性成为可能。

不同的构建环境是什么意思？构建环境的两个例子是产品环境和开发环境。当你在开发环境中工作时，你的系统可能被配置成访问运行在你本机的开发数据库实例，而在产品环境中，你的系统被配置成从产品数据库读取数据。Maven 能让你定义任意数量的构建环境（构建 profile），这些定义可以覆盖 pom.xml 中的任何配置。你可以配置你的应用程序，在“开发”profile 中，访问本地的开发数据库实例，在“产品”profile 中，访问产品数据库。Profile 也可以通过环境和平台被激活，你可以自定义一个构建，它根据不同的操作系统或者不同的 JDK 版本有不同的行为。在我们讨论使用和配置 Maven profile 之前，我们需要定义构建可移植性的概念。

11.1.1. 什么是构建可移植性



一个构建的“可移植性”是指将一个项目在不同的环境中构建的难易度。一个不用做任何自定义配置或者属性文件配置就能工作的构建，比一个需要很多配置才能工作的构建，具有更高的可移植性。构建可移植性最高的项目往往是一些开源项目如 Apache Commons 或者 Apache Velocity，它们自带 Maven 构建配置，不需要或者需要很少的自定义配置。简言之，可移植性最高的项目往往“开箱可用”，而可移植性最低的构建则需要你跳过一个一个难缠的框框，配置平台相关的路径以定位构建工具。在我们展示如何实现构建移植性之前，先浏览一些不同种类的构建可移植性。

11.1.1.1. 不可移植构建



缺少可移植性正是所有构建工具试图防止的——然而，任何工具都能被配置成不可移植（即使 Maven）。一个不可移植的项目只有在在一组特定的环境和标准（比如，你的本地机器）下才能构建。除非你自己一个人工作，不打算将你的应用部署到其它机器上，否则最好完全避免不可移植性。一个不可移植的构建只能在单独的机器上运行，是“一次性的”。Maven 设计提供了使用 profile 自定义构建的能力，阻止不可移植的构建。当一个新的开发人员得到不可移植项目的源码的时候，如果不重写大部分的构建脚本，他就不能构建这个项目。

11.1.1.2. 环境可移植性



如果一个构建有一种机制，能针对不同的环境有特定的行为和配置，那么我们就说该构建具有环境可移植性。例如，一个项目在测试环境中包含一个对于测试数据库的引用，而在产品环境中则引用了产品数据库，那么该项目的构建是环境可移植的。这很有可能是因为该构建针对不同的环境有不同的属性组。当你转移到一个不同的环境中，该环境

未被定义，也没有为其创建 **profile**，那么项目将不能工作。因此，该项目也只是在已定义的环境中可移植。

当一个新的开发人员得到环境可移植项目的源码，他们必须在已定义的环境中运行此构建，否则就需要创建自定义的环境才能构建此项目。

11.1.1.1.3. 组织（内部）可移植性



这一层可移植性的中心是一个项目可能需要访问一些内部资源如源码控制系统或者内部维护的 **Maven** 仓库。大公司的项目可能依赖于一个只对内部开发人员可用的数据库，一个开源项目可能需要一个特定级别的证书来发布 **web** 站点，以及将构建的产品发布到公共仓库。

如果你试图在内网外部（例如，在公司的防火墙外面）从零开始构建一个内部项目，构建会失败。它失败的原因可能是一些必须的自定义插件不可用，或者项目的依赖找不到，因为你没有适当的证书从一个自定义的远程仓库获取依赖。这样的项目只在一个组织内部的环境中拥有可移植性。

11.1.1.1.4. 广泛（全局）可移植性



任何人都可以下载具有广泛可移植性项目的源码，不用为特定的环境自定义构建就能进行编译，安装。这是最高级别的可移植性；构建这样的项目不需要做任何额外的工作。该级别的可移植性对开源项目尤为重要，因为开源项目的潜在贡献者需要能很方便的下载源码进行构建。

任何一个开发者都可以下载具有广泛可移植性项目的源码。

11.1.1.2. 选择一个适当级别的可移植性



很显然，你需要避免创建出最坏的情况：不可移植构建。你可能不幸需要在这样的组织工作或学习：其核心应用的构建是不可移植的。在这样的组织下，没有特定的人员或机器的帮助，你就不可能部署一个应用。如果不和那个维护该不可移植构建的人协调，很难引入新的项目依赖或变化。当个人或小组需要控制项目如何以及何时构建部署的时候，不可移植构建就会由于一些政治环境因素高速增长。“我们该如何构建该系统？哦，我们需要找到 **Jack**，让他帮我们构建，没有其他人能将其部署到产品环境中”这是一种非常危险的情形，该情形比你想象的普遍得多。如果你为这样的组织工作，**Maven** 和 **Maven profile** 能帮你脱离困境。

与之完全相反的可移植性范围是广泛可移植性构建。总的来说广泛可移植性构建是最难达到的构建系统。这样的构建严格要求你依赖的项目和工具是免费分发的，在公共环境中可用。很多商业软件包可能被排除在这种可移植性最高的构建外面，因为只有在接受某个许可证后你才能下载它们。广泛可移植性也限制它的依赖能以 **Maven** 构件的形式分发。例如，如果你依赖于 **Oracle JDBC** 驱动，你的用户就需要手工下载安装它们；这就不是广泛可移植性，因为你必须为那些有兴趣构建你应用的人发布一组关于创建环境的指令。而另一方面，你可以使用一个在公共 **Maven** 仓库中可用的 **JDBC** 驱动如 **MySQL** 或者 **HSQldb**。

如之前叙述的那样，开源项目从拥有最广泛的可移植性构建获益。广泛可移植性构建降低了为开源项目做贡献的低效性。在一个开源项目（如 **Maven**）中，有两个单独的组：

最终用户和开发者。当一个项目的最终用户决定为项目贡献一个补丁，它们就需要从使用构建输出过渡到运行一个构建。它们首先需要成为开发者，而如果很难学会如何构建一个项目，这就成为了最终用户花时间为项目做贡献的妨碍因素。在广泛可移植项目中，最终用户不需要遵循一组神秘的构建指令来开始成为开发者。他们可以签出源码，修改源码，构建，然后提交贡献，这个过程不需要问谁要求帮助建立构建环境。当为开源项目贡献源码的成本更低的时候，你就会看到源码贡献的增长，特别是一些不经意的贡献可能对项目的成功和项目的失败造成很大的影响。在一组广泛的开源项目中使用 Maven 的一个附加作用就是它使得开发者为不同的开源项目贡献源码变得更加容易。

11.2. 通过 Maven Profiles 实现可移植性



Maven 中的 **profile** 是一组可选的配置，可以用来设置或者覆盖配置默认值。有了 **profile**，你就可以为不同的环境定制构建。**profile** 可以在 `pom.xml` 中配置，并给定一个 **id**。然后你就可以在运行 **Maven** 的时候使用的命令行标记告诉 **Maven** 运行特定 **profile** 中的目标。以下 `pom.xml` 使用 **production profile** 覆盖了默认的 **Compiler** 插件设置。

Example 11.1. 使用一个 Maven Profile 覆盖 Compiler 插件设置

```
<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>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>production</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
```

```

        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
            <debug>>false</debug>
            <optimize>>true</optimize>
        </configuration>
    </plugin>
</plugins>
</build>
</profile>
</profiles>
</project>

```

本例中，我们添加了一个名为 **production** 的 **profile**，它覆盖了 **Maven Compiler** 插件的默认配置，现在仔细看一下这个 **profile** 的语法。

- ❶ pom.xml 中的 **profiles** 元素，它包含了一个或者多个 **profile** 元素。由于 **profile** 覆盖了 pom.xml 中的默认设置，**profiles** 通常是 pom.xml 中的最后一个元素。
- ❷ 每个 **profile** 必须要有一个 **id** 元素。这个 **id** 元素包含的名字将在命令行调用 **profile** 时被用到。我们可以通过传给 Maven 一个 **-P<profile_id>** 参数来调用 **profile**。
- ❸ 一个 **profile** 元素可以包含很多其它元素，只要这些元素可以出现在 **POM XML** 文档的 **project** 元素下面。本例中，我们覆盖了 **Compiler** 插件的行为，因此必须覆盖插件配置，该配置通常位于一个 **build** 和一个 **plugins** 元素下面。
- ❹ 我们覆盖了 **Maven Compiler** 插件的配置。确保通过 **production profile** 产生的字节码不会包含调试信息，并且字节码会被编译器优化。

要使用 **production profile** 来运行 **mvn install**，你需要在命令行传入 **-Pproduction** 参数。要验证 **production profile** 覆盖了默认的 **Compiler** 插件配置，可以像这样以开启调试输出(**-X**) 的方式运行 **Maven**。

```

~/examples/profile $ mvn clean install -Pproduction -X
... (omitting debugging output) ...
[DEBUG] Configuring mojo
'o.a.m.plugins:maven-compiler-plugin:2.0.2:testCompile'
[DEBUG] (f) basedir = ~\examples\profile
[DEBUG] (f) buildDirectory = ~\examples\profile\target
...
[DEBUG] (f) compilerId = javac
[DEBUG] (f) debug = false
[DEBUG] (f) failOnError = true
[DEBUG] (f) fork = false
[DEBUG] (f) optimize = true

```

```
[DEBUG] (f) outputDirectory = \
      ~\svn\sonatype\examples\profile\target\test-classes
[DEBUG] (f) outputFileName = simple-1.0-SNAPSHOT
[DEBUG] (f) showDeprecation = false
[DEBUG] (f) showWarnings = false
[DEBUG] (f) staleMillis = 0
[DEBUG] (f) verbose = false
[DEBUG] -- end configuration --
... (omitting debugging output) ...
```

Maven 的调试输出体现了 **production profile** 下 **Compiler** 插件的配置。可以看到，**debug** 被设置成 **false**，**optimize** 设置成 **true**。

11.2.1. 覆盖一个项目对象模型



虽然前面的样例展示了如何覆盖一个 **Maven** 插件的默认配置属性，你仍然没有确切知道 **Maven profile** 能覆盖什么。简单的回答这个问题，**Maven profile** 可以覆盖几乎所有 **pom.xml** 中的配置。**Maven POM** 包含一个名为 **profiles** 的元素，它包含了项目的替代配置，在这个元素下面，每个 **profile** 元素定义了一个单独的 **profile**。每个 **profile** 必须要有一个 **id**，除此之外，它可以包含几乎所有你能在 **project** 下看到的元素。以下的 XML 文档展示了一个 **profile** 允许覆盖的所有的元素。

Example 11.2. Profile 中允许出现的元素

```
<project>
  <profiles>
    <profile>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>...</distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>...</properties>
    </profile>
  </profiles>
</project>
```

一个 Profile 可以覆盖项目构件的最终名称，项目依赖，插件配置以影响构建行为。Profile 还可以覆盖分发配置；例如，如果你通过一个 staging profile 发布一个构件到 staging 服务器上，你就可以创建一个 profile 然后在里面定义 distributionManagement 元素。

11.3. 激活 Profile



在之前的小节中我们介绍了如何为一个特定的目标环境创建一个 profile 以覆盖默认行为。前面的例子中默认的构建是针对开发环境设计的，而 production profile 的存在就是为了为产品环境提供配置。当你需要基于一些变量如操作系统和 JDK 版本来进行配置的时候怎么做呢？Maven 提供了一种针对不同环境参数“激活”一个 profile 的方式，这就叫做 profile 激活。

看如下的例子，假设我们拥有一个 Java 类库，它有一个特定的功能只有在 Java 6 下才可用，它需要使用定义在 [JSR-223](#) 中的脚本引擎。你已经将那部分处理脚本的类库分离到了一个单独的 Maven 模块中，并且希望运行 Java 5 的人们能构建整个项目，而不去构建那部分针对 Java 6 的扩展类库。你可以使用一个 Maven profile，只有构建在 Java 6 JDK 上运行的时候才将脚本扩展模块添加到构建中。首先，让我们看一下这个项目的目录布局，以及我们希望开发者如何构建该系统。

当有人使用 Java 6 JDK 运行 `mvn install` 的时候，你希望构建包含 simple-script 模块，而它们使用 Java 5 的时候，你希望跳过 simple-script 模块的构建。假如在 Java 5 下不能够跳过 simple-script 模块的构建，构建会失败因为 Java 5 的 classpath 中没有 ScriptEngine。让我们看一下该类库项目的 pom.xml：

Example 11.3. 使用 Profile 激活动态包含子模块

```
<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>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

```

<profiles>
  <profile>
    <id>jdk16</id>
    <activation>
      <jdk>1.6</jdk>
    </activation>
    <modules>
      <module>simple-script</module>
    </modules>
  </profile>
</profiles>
</project>

```

如果你在 Java 1.6 下运行 **mvn install**，你会看到 Maven 下行到 simple-script 子目录构建 simple-script 项目。如果你在 Java 1.5 上运行 **mvn install**，Maven 就不会去构建 simple-script 子模块。让我们详细看一下激活配置：

- ❶ activation 元素列出了所有激活 profile 需要的条件。该例中，我们的配置为，当 Java 版本以“1.6”开头的时候 profile 会被激活。这包括“1.6.0_03”，“1.6.0_02”以及所有其它以“1.6”开头的字符串。激活参数不局限于 Java 版本，要了解激活参数的完整列表，请看[激活配置](#)。
- ❷ 在这个 profile 中我们添加了模块 simple-script。这么做会让 Maven 从 simple-script/子目录中寻找到一个 pom.xml 文件。

11.3.1. 激活配置



激活配置元素下可以包含一个或者多个选择器：包含 JDK 版本，操作系统参数，文件，以及属性。当所有标准都被满足的时候一个 profile 才会被激活。例如，一个 profile 可以要求操作系统家族为 Windows，JDK 版本为 1.4，那么该 profile 只有当构建在 Windows 机器上的 Java 1.4 上运行的时候才会被激活。如果该 profile 被激活，那么它定义的所有配置都会覆盖原来 POM 中对应层次的元素，就像使用命令行参数 **-P** 引入该 profile 一样。下面的例子展示了一个 profile，它通过一个十分复杂的配置组合激活，包括操作系统参数，属性，以及 JDK 版本。

Example 11.4. Profile 激活参数：JDK 版本，操作系统参数，以及属性

```

<project>
  ...
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>>false</activeByDefault>
        <jdk>1.5</jdk>
        <os>

```

```

    <name>Windows XP</name>
    <family>Windows</family>
    <arch>x86</arch>
    <version>5.1.2600</version>
  </os>
  <property>
    <name>mavenVersion</name>
    <value>2.0.5</value>
  </property>
  <file>
    <exists>file2.properties</exists>
    <missing>file1.properties</missing>
  </file>
</activation>
...
</profile>
</profiles>
</project>

```

上例定义了一组狭小的激活参数集合。让我们仔细看一下每个激活配置：

- ❶ `activeByDefault` 元素控制一个 `profile` 是否默认被激活。
- ❷ 该 `profile` 只有当 JDK 版本以“1.5”开头的时候才被激活。这包含“1.5.0_01”，“1.5.1”等。
- ❸ 该 `profile` 针对于一个特定的 Windows XP 版本，32 位平台上的 5.1.2600 版本。如果你的项目使用本地插件来构建一个 C 程序，你可能会发现自己正为特定的平台编写项目。
- ❹ `property` 元素告诉 **Maven**，当 `mavenVersion` 属性的值被设置成 2.0.5 的时候才激活 `profile`。
`mavenVersion` 是一个在所有 **Maven** 构建中可用的隐式属性。
- ❺ `file` 元素告诉我们只有当某些文件存在（或者不存在）的时候才激活 `profile`。该例中的 dev `profile` 只有在项目基础目录中存在 `file2.properties` 文件，并且不存在 `file1.properties` 文件的时候才被激活。

11.3.2. 通过属性缺失激活



你可以基于一个属性如 `environment.type` 的值来激活一个 `profile`。当 `environment.type` 等于 `dev` 的时候激活 `development profile`，或者当 `environment.type` 等于 `prod` 的时候激活 `production profile`。你也可以通过一个属性的缺失来激活一个 `profile`。下面的配置中，只有在 **Maven** 运行过程中属性 `environment.type` 不存在 `profile` 才被激活。

Example 11.5. 在属性缺失的情况下激活 Profile

```
<project>
...
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <property>
        <name>!environment.type</name>
      </property>
    </activation>
  </profile>
</profiles>
</project>
```

注意属性名称前面的感叹号。感叹号通常表示“否定”的意思。当没有设置 `${environment.type}` 属性的时候，这个 profile 被激活。

11.4. 外部 Profile



如果你开始大量使用 Maven profile，你会希望将 profile 从 POM 中分离，使用一个单独的文件如 `profiles.xml`。你可以混合使用定义在 `pom.xml` 中和外部 `profiles.xml` 文件中的 profile。只需要将 `profiles` 元素放到 `${basedir}` 目录下的 `profiles.xml` 文件中，然后照常运行 Maven 就可以。`profiles.xml` 文件的大概内容如下：

Example 11.6. 将 profile 放到一个 `profiles.xml` 文件中

```
<profiles>
  <profile>
    <id>development</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <debug>true</debug>
            <optimize>false</optimize>
          </configuration>
        </plugin>
      </plugins>
    </build>
```

```

</profile>
<profile>
  <id>production</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <debug>>false</debug>
          <optimize>>true</optimize>
        </configuration>
      </plugin>
    </plugins>
  </build>
</profile>
</profiles>

```

你可以发现一旦你的 **Profile** 变得很大，再让你管理 pom.xml 会变得很困难，或者说只是因为你觉得将 profiles.xml 文件从 pom.xml 分离出来是一种更干净的方式。不管怎样，调用定义在 pom.xml 中的 **profile** 和调用定义在 profiles.xml 中 **profile** 的方式是一样的。

11.5. Settings Profile



当一个项目需要为特定的环境自定义构建设置的时候，**profile** 很有用，但是为什么一定要为所有 **Maven** 项目覆盖构建设置呢？比如为每个 **Maven** 构建添加一个需要访问的内部仓库。你可以使用一个 **settings profile** 做这件事情。项目 **profile** 关心于覆盖某个项目的配置，而 **settings profile** 可以应用到所有你使用 **Maven** 构建的项目。你可以在两个地方定义 **settings profile**：定义在 ~/.m2/settings.xml 中的用户特定 **settings profile**，或者定义在 \${M2_HOME}/conf/settings.xml 中的全局 **settings profile**。这里是一个定义在 ~/.m2/settings.xml 中的 **settings profile** 的例子，它为所有的构建设置了一些用户特定的配置属性。如下 settings.xml 文件为用户 tobrien 定义：

Example 11.7. 定义用户特定的 Setting Profile (~/.m2/settings.xml)

```

<settings>
  <profiles>
    <profile>
      <id>dev</id>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>

```

```

    <executions>
      <execution>
        <goals>
          <goal>sign</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <keystore>/home/tobrien/java/keystore</keystore>
      <alias>tobrien</alias>
      <storepass>s3cr3tp@ssw0rd</storepass>
      <signedjar>
        ${project.build.directory}/signed/${project.build.finalName}.jar
      </signedjar>
      <verify>true</verify>
    </configuration>
  </plugin>
</profile>
</profiles>
</settings>

```

前面的例子是一个对于用户特定 **settings profile** 的真实用例。该样例中，当发布时为一个 **JAR** 文件签名的时候，设置用户特定的信息如密码和别名。你不会希望将这些配置参数存储到项目共享的 `pom.xml` 或者 `profiles.xml` 文件中，因为它们包含了一些不应该被公开的私人信息。

settings profile 的缺点是它们可能会干扰项目可移植性。如果前面的例子是一个开源项目，如果一个新的开发者没有和其它开发者交流过并手工配置了一个 **settings profile**，他将不能够为一个 **JAR** 签名。这样的情况下，为 **JAR** 签名的安全需求就与全局可移植性构建产生了冲突。在大部分开源项目中，有一些任务需要安全证书：将一个构件发布到远程仓库，发布项目的 **web** 站点，或者为 **JAR** 文件签名。对于这些任务，可以达到的最高级别可移植性只能是组织可移植性。这些高安全性任务通常需要一些手工的 **profile** 安装和配置。

除了显式的使用 **-P** 命令行参数指定 **profile** 的名称。你还可以为所有构建的项目定义一个激活 **profile** 的列表。例如，如果你想要为每个项目激活定义在 `settings.xml` 中的 **dev profile**，你可以在你的 `~/m2/settings.xml` 文件中添加如下的设置：

Example 11.8. 定义激活的 Settings Profile

```

<settings>
  ...
  <activeProfiles>
    <activeProfile>dev</activeProfile>
  </activeProfiles>
</settings>

```

该设置只会激活 **settings profile**，不会激活 **id** 匹配的项目 **profile**。例如，如果你有一个项目，在 `pom.xml` 中定义了一个 **id** 为 `dev` 的 **profile**，那个 **profile** 不会受你 `settings.xml` 中 `activeProfile` 设置的影响。`activeProfile` 设置只对你 `settings.xml` 文件中定义的 **profile** 有效。

11.5.1. 全局 Settings Profile



如同用户特定的 **settings profile** 一样，你也可以在 `${M2_HOME}/conf/settings.xml` 中定义一组全局 **profile**。在这个配置文件中定义的 **profile** 对所有使用该 **Maven** 安装的用户可用。如果你正为一个特定的组织创建一个定制的 **Maven** 分发包，并且你想要确保每个 **Maven** 用户使用一组构建 **profile** 以确保内部可移植性，定义全局 **settings profile** 就十分有用。如果你需要添加自定义插件仓库，或者定义一组只在你组织内部可用的自定义插件，你就可以为你的用户分发一个内置了这些配置的 **Maven**。配置全局 **settings profile** 和配置用户特定的 **settings profile** 的方法完全一样。

11.6. 列出活动的 Profile



Maven profile 可以通过 `pom.xml`，`profiles.xml`，`~/.m2/settings.xml`，或者 `${M2_HOME}/conf/settings.xml` 定义。由于有四个层次，除了记住哪个文件中定义了哪个 **profile**，没什么好的方式可以了解某个特定项目可用的 **profile**。为了更方便的了解某个项目可用的 **profile**，以及它们是在哪里定义的，**Maven Help** 插件定义了一个目标，`active-profiles`，它能列出所有激活的 **profile**，以及它们在哪里定义。你可以如下运行 `active-profiles` 目标：

```
$ mvn help:active-profiles
Active Profiles for Project 'My Project':

The following profiles are active:

- my-settings-profile (source: settings.xml)
- my-external-profile (source: profiles.xml)
- my-internal-profile (source: pom.xml)
```

11.7. 提示和技巧



Profile 可以用来鼓励构建可移植性。如果你的构建需要为不同的平台做一些细微的自定义，或者如果你针对不同的目标平台生成不同的输出，项目 **profile** 就能增加构建可移植性。**Settings profile** 通常会降低构建可移植性，因为它会添加一些必须在开发人员之间沟通的额外项目信息。下面的小节提供了一些如何在你项目中使用 **Maven profile** 的指导意见和想法。

11.7.1. 常见的环境



创建 Maven 项目 **profile** 的最核心动机之一就是为了提供环境特定的配置。在开发环境中，你可能想要生成带有调试信息的字节码，配置你的系统使用开发数据库实例。而在产品环境中，你可能会想要生成一个已签名的 **JAR**，并且配置系统使用产品数据库。本章，我们使用 **dev** 和 **prod** 等标识符定义了很多环境。一种更简单的方式是，定义一些会被环境属性自动激活的 **profile**，在你所有的项目中使用这些通用的环境属性。例如，如果每个项目都有一个 **development profile**，当属性 **environment.type** 的值为 **dev** 时被激活，并且同样的项目还有 **production profile**，当属性 **environment.type** 的值为 **prod** 时被激活。那么，你就可以在你的 **settings.xml** 中创建一个默认的 **profile**，在你的开发机器上，总是将 **environment.type** 设置为 **dev**。这样，所有定义了 **dev profile** 的项目都会由同样的系统变量激活。让我们看看如何完成这件事情，以下的 `~/.m2/settings.xml` 定义了一个 **profile**，将 **environment.type** 属性设置成了 **dev**。

Example 11.9. `~/.m2/settings.xml` 中定义一个设置了 **environment.type** 的默认 **profile**，

```
<settings>
  <profiles>
    <profile>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <environment.type>dev</environment.type>
      </properties>
    </profile>
  </profiles>
</settings>
```

这意味着每次你在你机器上运行 **Maven** 的时候，该 **profile** 总会被激活，属性 **environment.type** 的值被设置为 **dev**。之后你就可以使用这个属性来激活定义在某个如下项目 **pom.xml** 中的 **profile**。让我们看一下如何在项目的 **pom.xml** 定义一个当属性 **environment.type** 的值为 **dev** 时被激活的 **profile**。

Example 11.10. 项目 **profile**，当 **environment.type** 等于 '**dev**' 时被激活

```
<project>
  ...
  <profiles>
    <profile>
      <id>development</id>
      <activation>
        <property>
          <name>environment.type</name>
```

```

        <value>dev</value>
      </property>
    </activation>
  </properties>

  <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
  <database.url>
    jdbc:mysql://localhost:3306/app_dev
  </database.url>
  <database.user>development_user</database.user>
  <database.password>development_password</database.password>
</properties>
</profile>
<profile>
  <id>production</id>
  <activation>
    <property>
      <name>environment.type</name>
      <value>prod</value>
    </property>
  </activation>
</properties>

<database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>

<database.url>jdbc:mysql://master01:3306,slave01:3306/app_prod</database.url>
</database.url>
  <database.user>prod_user</database.user>
</properties>
</profile>
</profiles>
</project>

```

该项目定义了一些属性如 `database.url` 和 `database.user`，它们可能被用来配置一些定义在 `pom.xml` 中的 **Maven** 插件。有很多可用的插件可以用来操作数据库，运行 **SQL**，而且还有如 **Maven Hibernate3** 之类的插件可以帮你在使用持久化框架的时候生成注解模型对象。这其中的一些插件，可以在 `pom.xml` 中被配置成使用这些属性。这些属性还可以被用来过滤资源。本例中，因为我们在 `~/.m2/settings.xml` 中定义了一个 **profile**，设置 `environment.type` 为 `dev`，在这台开发机器上运行 **Maven** 的时候，**development profile** 就一直被激活。做为选择，如果我们想要覆盖缺省值，我们可以在命令行设置这个属性值。如果我们需要激活 **production profile**，我们可以如下运行 **Maven**：

```
~/.examples/profiles $ mvn install -Denvironment.type=prod
```

在命令行设置一个属性可以覆盖定义在`~/m2/settings.xml`中的缺省值。我们可以仅仅定义一个 id 为“dev”的 **profile**，然后直接使用 **-P** 命令行参数调用它，但是使用 `environment.type` 属性允许我们在编写其它项目的 `pom.xml` 时遵循这个标准。在你代码库中的每个项目都可以有一个 **profile**，由定义在每个用户`~/m2/settings.xml`中的项目的 `environment.type` 属性激活。这样，开发人员就能共享开发配置，而不用在不可移植的 `settings.xml` 文件中定义这些配置。

11.7.2. 安全保护



该最佳实践基于前面一小节。在[项目 profile, 当 environment.type 等于'dev'时被激活](#)中，**production profile** 不包含 `database.password` 属性。我特地这么做是为了说明一个概念：将秘密信息放到你的用户特定的 `settings.xml` 文件中。如果你在一个大型组织中开发一个应用，那么很可能开发小组的大部分人不知道产品数据库的密码。如果一个组织的开发小组和运营小组有明确的界限的话，这种安全保护就很常见了。开发人员可以访问开发环境和 **staging** 环境，但是它们往往不被允许访问产品数据库。为什么要这么做有很多原因，尤其是当一个组织处理异常敏感的经济，情报，或者医疗信息的时候。在这样的情况下，产品环境构建只能有开发者领导或者产品运营小组成员执行。当它们使用 `environment.type` 的时候，它们会需要在 `settings.xml` 在中定义如下变量。

Example 11.11. 在用户特定 Settings Profile 中存储秘密信息

```
<settings>
  <profiles>
    <profile>
      <activeByDefault>true</activeByDefault>
      <properties>
        <environment.type>prod</environment.type>
        <database.password>m1ss10nimp0ss1b13</database.password>
      </properties>
    </profile>
  </profiles>
</settings>
```

这个用户定义了一个默认 **profile**，将 `environment.type` 设置成 `prod`，同时也设置了产品数据库密码。当项目构建的时候，**production profile** 由 `environment.type` 属性激活，并且 `database.password` 属性也被填充。这样，你就可以将所有产品项目的配置放到项目的 `pom.xml` 中，而不用在那里配置访问产品数据库必要的秘密信息。

Note

秘密信息通常会和可移植性冲突，但这是合理的。你不会想公开你的秘密信息。

11.7.3. 平台分类器



假设你有一个类库，或者一个项目，它针对不同的平台有不同的输出。即使 **Java** 是平台无关的，但还是有一些时候，你会需要编写一些代码调用平台项目的本地代码。另一

个可能性就是你编写了一些使用 **Maven Native** 插件编译的 C 代码，并且要基于构建平台生成已修饰的构件。你可以通过 **Maven Assembly** 插件或者 **Maven Jar** 插件设置一个分类器。以下的 pom.xml 使用了由操作系统参数激活的 **profile** 来生成已修饰的构件。要了解更多的关于 **Maven Assembly** 插件的信息，请参考 [Chapter 12, Maven Assemblies](#)。

Example 11.12. 使用由平台激活的 Profile 修饰构件

```
<project>
...
<profiles>
  <profile>
    <id>windows</id>
    <activation>
      <os>
        <family>windows</family>
      </os>
    </activation>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <classifier>win</classifier>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
  <profile>
    <id>linux</id>
    <activation>
      <os>
        <family>unix</family>
      </os>
    </activation>
    <build>
      <plugins>
        <plugin>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <classifier>linux</classifier>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

```
    </profile>
  </profiles>
</project>
```

如果操作系统是 Windows 家族，该 pom.xml 就使用"-win"修饰这个 JAR 构建。如果操作系统是 Unix 家族，该构件就由"-linux"修饰。pom.xml 成功的给构件添加修饰符，但是由于在两个 profile 中的 Maven Jar 插件的冗余配置，该文件变得有些累赘了。该样例可以被重写，如下，使用变量替换来减少冗余：

Example 11.13. 使用由平台激活的 Profile 和变量替换修饰构件

```
<project>
...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <classifier>${envClassifier}</classifier>
      </configuration>
    </plugin>
  </plugins>
</build>
...
<profiles>
  <profile>
    <id>windows</id>
    <activation>
      <os>
        <family>windows</family>
      </os>
    </activation>
    <properties>
      <envClassifier>win</envClassifier>
    </properties>
  </profile>
  <profile>
    <id>linux</id>
    <activation>
      <os>
        <family>unix</family>
      </os>
    </activation>
    <properties>
      <envClassifier>linux</envClassifier>
```

```
    </properties>
  </profile>
</profiles>
</project>
```

在这个 pom.xml 中，每个 **profile** 不需要包含一个 build 元素来配置 Jar 插件。每个 **profile** 通过操作系统家族参数激活，并且设置 envClassifier 属性为 win 或者 linux。这个 envClassifier 属性由缺省 pom.xml 的 build 元素引用，以为项目的 JAR 构建添加分类器。JAR 构件将会被命名为 \${finalName}-\${envClassifier}.jar，并且需要按如下的依赖语法引用。

Example 11.14. 依赖于一个已修饰的构件

```
<dependency>
  <groupId>com. mycompany</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <classifier>linux</classifier>
</dependency>
```

11.8. 小结



如果正确的使用 **profile**，它可以为不同的平台很容易的自定义构建。如果你构建中的一些东西需要定义一个平台特定的路径，如应用程序服务器，你可以将这些配置点放到 **profile** 中，然后由操作系统参数激活。如果你有一个项目需要为不同的环境生成不同的构件，你可以为不同的环境和平台自定义 **profile** 特定的插件行为，从而自定义构建行为。使用 **profile**，构建可以变得容易移植，没有必要为一个新的环境重写你的构建逻辑，只需要重写那些需要变化的配置，共享那些可以被共享的配置。

Chapter 14. Maven 和 Eclipse: m2eclipse

[14.1. 简介](#)

[14.2. m2eclipse](#)

[14.3. 安装 m2eclipse 插件](#)

[14.3.1. 安装前提条件](#)

[14.3.1.1. 安装 Subclipse](#)

[14.3.1.2. 安装 Mylyn](#)

[14.3.1.3. 安装 AspectJ Tools Platform \(AJDT\)](#)

[14.3.1.4. 安装 Web Tools Platform \(WTP\)](#)

[14.3.2. 安装 m2eclipse](#)

[14.4. 开启 Maven 控制台](#)

[14.5. 创建一个 Maven 项目](#)

[14.5.1. 从 SCM 签出一个 Maven 项目](#)

[14.5.2. 用 Maven Archetype 创建一个 Maven 项目](#)

[14.5.3. 创建一个 Maven 模块](#)

[14.6. 创建一个 Maven POM 文件](#)

[14.7. 导入 Maven 项目](#)

[14.7.1. 导入一个 Maven 项目](#)

[14.7.2. 具体化一个 Maven 项目](#)

[14.8. 运行 Maven 构建](#)

[14.9. 使用 Maven 进行工作](#)

[14.9.1. 添加及更新依赖或插件](#)

[14.9.2. 创建一个 Maven 模块](#)

[14.9.3. 下载源码](#)

[14.9.4. 打开项目页面](#)

[14.9.5. 解析依赖](#)

[14.10. 使用 Maven 仓库进行工作](#)

[14.10.1. 搜索 Maven 构件和 Java 类](#)

[14.10.2. 为 Maven 仓库编制索引](#)

[14.11. 使用基于表单的 POM 编辑器](#)

[14.12. 在 m2eclipse 中分析项目依赖](#)

[14.13. Maven 选项](#)

[14.14. 小结](#)

14.1. 简介



Eclipse IDE 是目前 Java 开发人群中使用得最广泛的 IDE。Eclipse 有一大堆的插件（请看 <http://www.eclipseplugincentral.com/>），无数的组织在它之上开发他们自己的软件。显然，Eclipse 无处不在。[m2Eclipse](#) 项目在 Eclipse IDE 中提供了对 Maven 的支持，本章，我们将会研究它提供的特性，以帮助你在 Eclipse IDE 中使用 Maven。

14.2. m2eclipse



m2eclipse 插件（<http://m2eclipse.codehaus.org/>）为 Eclipse 提供了 Maven 的集成。m2Eclipse 同时也以挂钩的方式连接了 Subclipse 插件（<http://subclipse.tigris.org/>）和 Mylyn 插件（<http://www.eclipse.org/mylyn/>）的特性。Subclipse 插件为 m2eclipse 提供了与 Subversion 仓库交互的能力，Mylyn 插件为 m2eclipse 提供了与任务集中接口交互的能力，该接口能跟踪开发过程的上下文。m2eclipse 提供的一些特性包括：

- 创建和引入 Maven 项目
- 依赖管理和与 Eclipse classpath 的集成
- 自动下载和更新依赖
- 构件的 Javadoc 及源码解析
- 使用 Maven Archetypes 创建项目
- 浏览，搜索远程 Maven 仓库
- 通过自动更新依赖列表管理 POM
- 从 Maven POM 具体化一个项目
- 从多个 SCM 仓库签出一个 Maven 项目
- 适配嵌套的多模块 Maven 项目至 Eclipse IDE
- 与 Web Tools Project （WTP）集成
- 与 AspectJ Development Tools （AJDT） 集成
- 与 Subclipse 集成
- 与 Mylyn 集成
- 基于表单的 POM 编辑器
- 依赖图的图形化显示
- 依赖树和已解析依赖的 GUI 展现

在上述列表以外 m2eclipse 还有很多其它的特性，本章介绍一些更令人印象深刻的特性。让我们从安装 e2eclipse 插件开始。

14.3. 安装 m2eclipse 插件



要安装 m2Eclipse 插件，你需要符合一些先决条件。你需要运行 Eclipse 3.2 或更高版本，JDK 1.4 或更高版本，你需要确认 Eclipse 是在 JDK 上运行而不是 JRE。在你有了 Eclipse 和兼容的 JDK 之后，你需要安装两个 Eclipse 插件：Subclipse 和 Mylyn。

14.3.1. 安装前提条件



你可以在安装 m2eclipse 的时候安装这些前提条件的软件，只要为每个前提条件软件添加一个远程更新站点至 Eclipse。要安装这些先决条件软件，找到 Help → Software Updates → Find and Install...。选择这个菜单项会载入 Install/Update 对话框。选择“Search for new features to install”选项然后点击 Next。你将会看到一个“Update sites to visit”的列表。点击 New Remote Site...，然后为每一个新的前提条件添加一个新的更新站点。为每个插件添加新的更新站点然后确认新站点被选择了。在你点击 Finish 之后，Eclipse 会让你选择插件组件以安装。选择你想要安装的组件，Eclipse 会下载，安装及配置你的插件。

需要注意的是如果你正在使用 Eclipse 最新的版本 Eclipse 3.4(Ganymede)，安装插件的过程可能会有点不一样。在 Ganymede 中，你需要选择 Help → Software Updates...，它会载入"Software Updates and Add-ons"对话框。在这个对话框中，选择“Available Software”面板然后点击 Add Site...，它会载入"Add Site"对话框。输入更新站点的 URL 然后点击 OK。在"Software Updates and Add-ons"对话框中会出现更新站点上可用的插件。你可以选择你想要安装的模块然后点击 Install... 按钮。Eclipse 会解析所选插件的所有依赖，然后要求你同意插件的许可证。在 Eclipse 安装了新的插件之后，它会征求你的允许以重启。

14.3.1.1. 安装 Subclipse



要安装 Subclipse，使用下面的 Eclipse 插件更新站点。

- Subclipse 1.2: http://subclipse.tigris.org/update_1.2.x

想要了解其它版本的 Subclipse，以及关于 Subclipse 插件更多的信息，请访问 Subclipse 项目的 web 站点: <http://subclipse.tigris.org/>。

14.3.1.2. 安装 Mylyn



要安装集成了 JIRA 支持的 Mylyn，添加 Mylyn Extras 的 Eclipse 更新 URL，如果你的组织使用 [Atlassian's JIRA](#) 来跟踪问题，你会需要这么做。使用下面的更新站点来安装 Mylyn:

- Mylyn (Eclipse 3.3): <http://download.eclipse.org/tools/mylyn/update/e3.3>
- Mylyn (Eclipse 3.4): <http://download.eclipse.org/tools/mylyn/update/e3.4>
- Mylyn Extras (JIRA 支持):
<http://download.eclipse.org/tools/mylyn/update/extras>

想了解关于 Mylyn 项目的更多信息，访问 Mylyn 项目的 web 站点：
<http://www.eclipse.org/mylyn/>。

14.3.1.3. 安装 AspectJ Tools Platform (AJDT)



如果你正在安装 m2eclipse 的 0.9.4 版本，你可能同时也想要安装 Web Tools Platform (WTP) 和 AspectJ Development Tools (AJDT)。使用如下的 eclipse 更新 URL 以安装 AJDT。

- AJDT (Eclipse 3.3): <http://download.eclipse.org/tools/ajdt/33/update>
- AJDT (Eclipse 3.4): <http://download.eclipse.org/tools/ajdt/34/dev/update>

想要了解更多的关于 AJDT 项目的信息，请访问 AJDT 项目的 web 站点
<http://www.eclipse.org/ajdt/>。

14.3.1.4. 安装 Web Tools Platform (WTP)



要安装 Web Tools Platform (WTP)。使用如下的 eclipse 更新 URL，或者直接在 Discovery 站点中寻找 Web Tool Project，该站点应该已经在你的 Eclipse 远程更新站点列表中。

- WTP: <http://download.eclipse.org/webtools/updates/>

关于更多的 Web Tools Platform 的信息，请访问 Web Tools Platform 项目的 web 站点 <http://www.eclipse.org/webtools/>。

14.3.2. 安装 m2eclipse



一旦你已经安装好这些先决条件，你从如下的 Eclipse 更新 URL 安装 m2eclipse 插件：

- m2eclipse 插件: <http://m2eclipse.sonatype.org/update/>

如果你想要安装最新的该插件的快照开发版本，你应该使用如下的开发更新 URL 而非之前的 URL。

- m2eclipse 插件（开发快照）: <http://m2eclipse.sonatype.org/update-dev/>

要安装 m2eclipse，只需要添加一个正确的更新站点。至 Help → Software Updates → Find and Install...，选择这个菜单项后会载入 Install/Update 对话框。选择"Search for new features to install"选项然后点击 Next。你将会看到一个"Update sites to visit"列表。点击 New Remote Site...，然后添加 m2eclipse 的更新站点。确认这个新添加的站点被选中了。在你点击 Finish 之后，Eclipse 会要求你选择要安装的组件。你选好之后 Eclipse 会自动下载，安装，和配置 m2eclipse。

如果你已经成功安装了这个插件，当你打开 Window → Preferences...的时候，你应该能够在选项列表中看到一个 Maven 选项。

14.4. 开启 Maven 控制台



在我们开始查看 m2eclipse 的特征之前，首先让我们开启 Maven 的控制台。通过访问 Window → Show View → Console 来打开控制台视图。然后点击控制台视图右手边的一个小箭头，然后选择 Maven 控制台，如下显示：



Figure 14.1. 在 Eclipse 中开启 Maven 控制台

Maven 控制台显示那些当在命令行运行 Maven 时出现在控制台的 Maven 输出。能看到 Maven 正在干什么，以及根据调试输出来诊断问题，都是很实用的。

14.5. 创建一个 Maven 项目



在 Maven 中，我们使用 archetype 来创建项目。在 Eclipse 中，我们通过新建项目向导来创建项目。Eclipse 中的新建项目向导为创建新项目提供了大量的模板。m2eclipse 插件为这个向导增加如下的功能：

- 从 SCM 仓库签出一个 Maven 项目
- 使用 Maven archetype 创建一个 Maven 项目
- 创建一个 Maven POM 文件

如 [Figure 14.2, “使用 m2eclipse 向导来创建一个新项目”](#) 所示，这三个选项对使用 Maven 的开发人员来说都很重要。让我们逐个看一下。

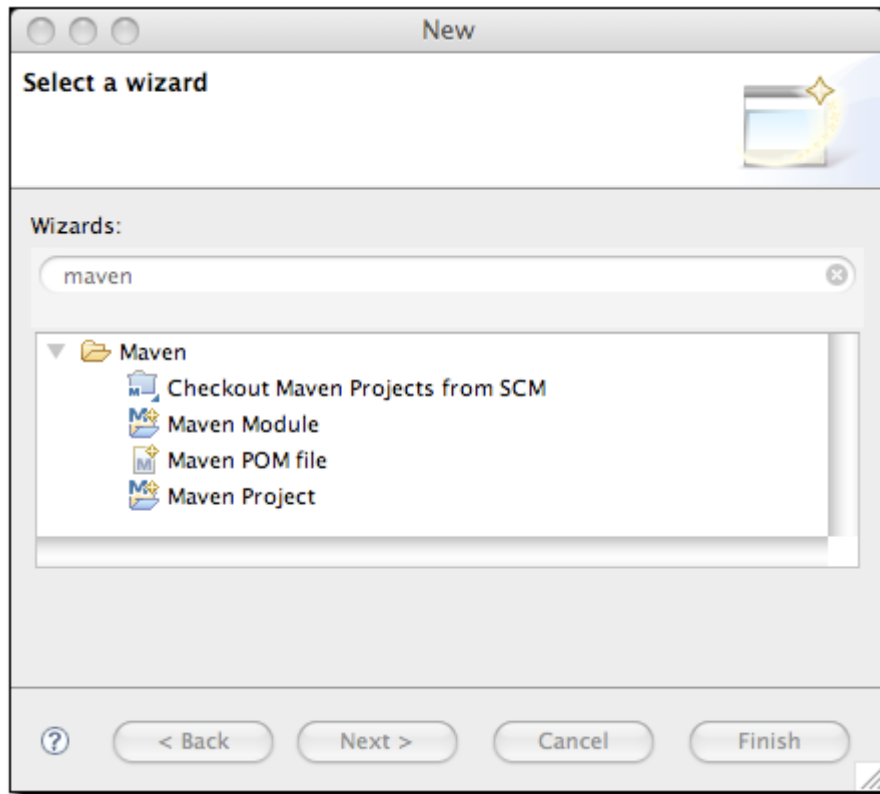


Figure 14.2. 使用 m2eclipse 向导来创建一个新项目

14.5.1. 从 SCM 签出一个 Maven 项目

m2eclipse 提供了直接从 SCM 仓库签出项目的 ability。简单的输入项目的 SCM 信息，它就会为你签出项目至你所选择的位置，如 [Figure 14.3, “从 Subversion 签出一个新的项目”](#)：

Figure 14.3. 从 Subversion 签出一个新的项目

该对话框中还有其它的选项用来浏览 Subversion 仓库的修订版以指定某个特定的修订版，或者直接手工输入修订版本号。这些特性重用了 Subclipse 插件的一些特性以和 Subversion 仓库相互。除了 Subversion，m2eclipse 插件也支持下面的 SCM 提供者：

- Bazaar
- Clearcase
- CVS
- git
- hg

- Perforce
- Starteam
- Subversion
- Synergy
- Visual SourceSafe

14.5.2. 用 Maven Archetype 创建一个 Maven 项目



m2eclipse 提供了使用 Maven Archetype 创建一个 Maven 项目的能力。伴随着 m2eclipse 有许多可用的 Maven Archetype，如 [Figure 14.4, “使用 Maven Archetype 创建一个 Maven 项目”](#)：

Figure 14.4. 使用 Maven Archetype 创建一个 Maven 项目

[Figure 14.4, “使用 Maven Archetype 创建一个 Maven 项目”](#)中的 archetype 列表是由一个叫 Nexus 索引器生成的。Nexus 是一个仓库管理器，将会在 [Chapter 16, 仓库管理器](#)介绍。Nexus 索引器是一个包含了整个 Maven 仓库索引的文件，m2eclipse 使用它来罗列出所有 Maven 仓库中可用的 archetype。到本章更新为止，m2eclipse 大概在这个对话框中有 90 个 archetype。其中比较突出的有：

- 标准的 Maven Archetypes 以创建
 - Maven 插件
 - 简单 Web 应用
 - 简单项目
 - 新的 Maven Archetypes
- [Databinder](#) Archetype (数据驱动的 Wicket 应用程序) 位于 net.databinder
- [Apache Cocoon](#) Archetype 位于 org.apache.cocoon
- [Apache Directory Server](#) Archetype 位于 org.apache.directory.server
- [Apache Geronimo](#) Archetype 位于 org.apache.geronimo.buildsupport
- [Apache MyFaces](#) Archetype 位于 org.apache.myfaces.buildtools
- [Apache Tapestry](#) Archetype 位于 org.apache.tapestry
- [Apache Wicket](#) Archetype 位于 org.apache.wicket
- [AppFuse](#) Archetype 位于 org.appfuse.archetypes
- [Codehaus Cargo](#) Archetype 位于 org.codehaus.cargo
- [Codehaus Castor](#) Archetype 位于 org.codehaus.castor

- [Groovy-based Maven Plugin](#) Archetype (不推荐使用)^[3] 位于 org.codehaus.mojo.groovy
- Jini Archetype
- [Mule](#) Archetype 位于 org.mule.tools
- [Objectweb Fractal](#) Archetype 位于 org.objectweb.fractal
- [Objectweb Petals](#) Archetype 位于 org.objectweb.petals
- ops4j Archetype 位于 org.ops4j
- [Parancoe](#) Archetype 位于 org.parancoe
- slf4j Archetype 位于 org.slf4j
- [Springframework](#) OSGI 和 Web Services Archetype 位于 org.springframework
- [Trails Framework](#) Archetype 位于 org.trailsframework

^[3]这些只是由 Nexus 索引器目录罗列的 archetype，如果你切换目录你会看到其它的 archetype。虽然你看到的结果会有变化，但是以下额外的 archetype 能在 Internal 目录中得到：

- [Atlassian Confluence](#) 插件 Archetype 位于 com.atlassian.maven.archetypes
- [Apache Struts](#) Archetype 位于 org.apache.struts
- Apache Shale Archetype 位于 org.apache.shale

一个目录是对于仓库索引的简单引用。你看可以通过点击在 **catalog** 下拉菜单旁边的 **Configure...**按钮来管理一组 m2eclipse 已经了解的目录。如果你有你自己的 archetype 需要加入到这个列表中，可以点击 **Add Archetype...**。

一旦你选择了一个 archetype，Maven 会从 Maven 仓库取得相应的 artifact 然后使用这个 archetype 创建一个新的 Eclipse 项目。

14.5.3. 创建一个 Maven 模块



m2eclipse 提供了创建一个 Maven 模块的能力。创建一个 Maven 模块和创建一个 Maven 项目几乎一样，它也会用 Maven archetype 创建一个新的 Maven 项目。然而，一个 Maven 模块是另一个 Maven 项目的子项目，后者通常被认为是父项目。

Figure 14.5. 创建一个 Maven 模块

当创建一个新的 Maven 模块的时候你必须选择一个在 Eclipse 中存在的父项目。点击浏览按钮，会看到一个已存在的项目的列表，如 [Figure 14.6, “为一个新的 Maven 模块选择一个父项目”](#)：

Figure 14.6. 为一个新的 Maven 模块选择一个父项目

在该列表中选择一个父项目之后，你回到了创建新 Maven 模块的窗口，父项目字段已被填充，如 [Figure 14.5, “创建一个 Maven 模块”](#)所示。点击 Next 你将会看到来自 [Section 14.5.2, “用 Maven Archetype 创建一个 Maven 项目”](#)的标准 archetype 列表，然后你可以选择用哪个 archetype 来创建 Maven 模块。

14.6. 创建一个 Maven POM 文件



另外一个 m2eclipse 提供的重要特性是它能创建一个新的 Maven POM 文件。m2eclipse 提供了一个向导，可以用来很轻松的为一个已经在 Eclipse 中的项目创建一个新的 POM 文件。这个 POM 创建向导如 [Figure 14.7, “创建一个新的 POM”](#)所示：

Figure 14.7. 创建一个新的 POM

创建一个新的 Maven POM 大致就是选择一个项目，在 m2eclipse 提供的字段中输入 Group Id, Artifact Id, Version, 选择打包类型，以及提供一个名称。点击 Next 按钮开始添加依赖。

Figure 14.8. 为新的 POM 添加依赖

正如你能在 [Figure 14.8, “为新的 POM 添加依赖”](#)看到的，POM 中现在还没有依赖。点击 Add 按钮以向中央 Maven 仓库查询依赖，如 [Figure 14.9, “向中央仓库查询依赖”](#)所示：

Figure 14.9. 向中央仓库查询依赖

查询依赖只是简单的输入你需要的构件的 groupId。 [Figure 14.9, “向中央仓库查询依赖”](#)展示了对 org.apache.commons 的一个查询，其中 commons-vfs 已被展开以查看可用的版本。选中 commons-vfs 的 1.1-SNAPSHOT 版本然后点击 OK，你会回到依赖选择界面，你可以查询更多的构件或者直接点击 finish 按钮以创建 POM。当你搜索依赖的时候，m2eclipse 正使用在 Nexus 仓库管理器中使用的同样的 Nexus 仓库索引。现在你已经看到了 m2eclipse 创建新项目的特性，让我们看下一组类似的将项目引入 Eclipse 的特性。

14.7. 导入 Maven 项目



m2eclipse 为导入 Maven 项目至 Eclipse 提供了三种选择，分别是：

- 导入一个已存在的 Maven 项目
- 从 SCM 签出一个 Maven 项目

- 具体化一个 Maven 项目

[Figure 14.10, “导入一个 Maven 项目”](#) 展示了 m2eclipse 提供的带有 Maven 选项的项目导入向导：

Figure 14.10. 导入一个 Maven 项目

使用 Eclipse 中的命令 **File** → **Import**，然后在过滤字段中输入单词 **maven**，就可以看到 [Figure 14.10, “导入一个 Maven 项目”](#) 的对话框。正如前面提到的，导入一个 Maven 项目至 Eclipse 有三种可用的方法：现存的 Maven 项目，从 SCM 签出一个项目，以及具体化 Maven 项目。

从 Subversion 导入一个 Maven 项目和前一节讨论的从 Subversion 创建一个 Maven 项目是等同的，因此再次讨论就显得冗余了。让我们往前走，看一下导入 Maven 项目至 Eclipse 的另外两个选项。

14.7.1. 导入一个 Maven 项目



m2eclipse 可以通过一个已存在的 pom.xml 导入一个 Maven 项目。通过指向 Maven 项目所在的目录，m2eclipse 能探测到该项目中的所有 POM，然后提供一个这些 POM 的层次列表，如 [Figure 14.11, “导入一个多模块的 Maven 项目”](#)。

Figure 14.11. 导入一个多模块的 Maven 项目

[Figure 14.11, “导入一个多模块的 Maven 项目”](#) 显示了被导入的项目的视图。注意该项目中所有的 POM 是分层的。这让你能够很简单的选择到你想要导入至 Eclipse 的 POM（也就是你想要导入的项目）。当你选择了你想要导入的项目之后，m2eclipse 会使用 Maven 导入并构建这个项目。

14.7.2. 具体化一个 Maven 项目



Maven 还提供了“具体化”一个 Maven 项目的的能力。具体化类似于从 Subversion 签出一个 Maven 项目的过程，但此时 Subversion URL 是从项目的根 POM 文件找到的，而不是手工的输入。如果一个 POM 文件有正确的元素来指定源代码仓库的位置，你就能仅仅通过这个 POM 文件来“具体化”Maven 项目。使用这个特性，你可以浏览中央 Maven 仓库中的项目，然后将其具体化成 Eclipse 项目。如果你的项目依赖于一个第三方的开源库，而且你需要查看这个库的源码，具体化的特性就变得十分方便和实用。现在只需要实用 m2eclipse 魔术般的“具体化”特性将项目导入到 Eclipse 中，而不是去追查项目的 web 站点然后寻找如何将其从 Subversion 签出。

[Figure 14.12, “Materializing a Maven Project”](#) 展示了选择具体化 Maven 项目后的向导：

Figure 14.12. Materializing a Maven Project

注意在这个对话框中 **Maven artifacts** 是空的。这是因为还没有添加项目。为了添加一个项目,你需要点击右边的 **Add** 按钮然后选择一个来自中央 **Maven** 仓库的依赖以添加。[Figure 14.13, “选择一个构件以具体化”](#)展示了如何添加一个项目:

Figure 14.13. 选择一个构件以具体化

当输入查询的时候,候选的依赖将会被在本地 **Maven** 仓库找到。花几秒钟对本地 **Maven** 仓库索引之后,候选依赖列表就会显示。选择一个要添加的依赖然后点击 **OK**,这样它们就会被添加到列表中如 [Figure 14.14, “具体化 Apache Camel”](#)。

Figure 14.14. 具体化 Apache Camel

在添加一个依赖的时候,你有一个选项,让 **m2eclipse** 签出这个构件的所有项目。

14.8. 运行 Maven 构建



m2eclipse 修改了 **Run As...**和 **Debug As...**菜单,以让你能够在 **Eclipse** 中运行 **Maven**。[Figure 14.15, “通过 Run As..运行一个 Eclipse 构建”](#)展示了一个 **m2eclipse** 项目的 **Run As...**菜单。从这个菜单你可以运行一些常用的生命周期过程如 **clean**, **install**, 或者 **package**。你也可以载入运行配置对话框窗口,然后使用参数及更多的选项来配置一个 **Maven** 构建。

Figure 14.15. 通过 Run As..运行一个 Eclipse 构建

如果你需要用更多的选项来配置一个 **Maven** 构建,你可以选择 **Run Configurations...**然后创建一个 **Maven** 构建。[Figure 14.16, “配置一个 Maven 构建作为一个运行配置”](#)展示了配置一个 **Maven** 构建的运行配置对话框。

Figure 14.16. 配置一个 Maven 构建作为一个运行配置

运行配置对话框允许你指定多个目标和 **profile**,它暴露了类似于“**skip tests**”和“**update snapshots**”的选项,并且允许你自定义从项目到 **JRE** 到环境变量的一切。你可以使用这个对话框来支持任何你希望在 **m2eclipse** 中启动的自定义 **Maven** 构建。

14.9. 使用 Maven 进行工作



当项目在 **Eclipse** 中的时候, **m2eclipse** 插件为使用 **Maven** 提供了一组特性。有很多特性使得在 **Eclipse** 中使用 **Maven** 变得十分容易,让我们仔细看一下。在前一节,我

们具体了一个 Maven 项目并且选择了一个来自于 Apache Camel 的名为 camel-core 的子项目。我们将使用这个项目来演示这些特性。

通过在 camel-core 项目上右击，然后选择 Maven 菜单项，你能看到可用的 Maven 特性。[Figure 14.17, “可用的 Maven 特性”](#)展示了这些特性的一个快照。

Figure 14.17. 可用的 Maven 特性

注意在 [Figure 14.17, “可用的 Maven 特性”](#)中 camel-core 项目可用的特性包括：

- 添加依赖和插件
- 更新依赖，快照和源代码文件夹
- 创建一个 Maven 模块
- 下载源代码
- 打开项目的 URL 如项目 Web 页面，问题追踪系统，源码控制，和持续集成工具
- 开启/关闭工作台解析器，嵌套 Maven 模块和依赖管理

这些特性都能帮你节省很多时间，让我们先简单的看一下。

14.9.1. 添加及更新依赖或插件



让我们假设我们想要给 camel-core POM 添加一个依赖或者一个插件。为了示范，我们会添加 commons-lang 作为一个依赖。（请注意添加依赖或者插件的功能完全一样，因此我们就用添加一个依赖作为示范。）

m2eclipse 为给一个项目添加依赖提供了两种选项。第一种选项是通过手动的编辑 POM 文件的 XML 内容来添加一个依赖。这种手动编辑 POM 文件方式的缺点是你必须知道构件的信息，或者，你可以使用下一节讨论的特性来手工的定位仓库索引中的构件信息。好处是在你手工添加依赖并保存 POM 文件之后，项目的 Maven 依赖容器会自动更新以包含这个新的依赖。[Figure 14.18, “手动给项目的 POM 添加一个依赖”](#)展示了如何给 camel-console POM 添加对 commons-lang 的依赖，然后 Maven 依赖容器自动更新并包含了这个依赖。

Figure 14.18. 手动给项目的 POM 添加一个依赖

手动添加依赖效果不错但是它比第二种方式需要更多的工作。在手动给 POM 添加依赖元素的时候，Eclipse 工作台右下角的进程反映了这一动作，如 [Figure 14.19, “更新 Maven 依赖”](#)：

Figure 14.19. 更新 Maven 依赖

第二种添加依赖的方式容易得多，因为你不需要知道构件的除 groupId 以外的信息。

[Figure 14.20, “搜索依赖”](#)展示了这种功能：

Figure 14.20. 搜索依赖

通过简单的在搜索框中输入信息，m2eclipse 会查询仓库索引，显式在本地 Maven 仓库中构件的版本。这种方式更好因为它能节省大量的时间。有了 m2eclipse，你不再需要中央 Maven 仓库中搜寻一个构件版本。

14.9.2. 创建一个 Maven 模块



m2eclipse 使得在一个多模块的 Maven 项目中创建一系列的嵌套项目变得十分容易。如果你有一个父项目，而且你想给这个项目添加一个模块，只需要在项目上右击，打开 Maven 菜单，选择“New Maven Module Project”。m2eclipse 会带你创建一个新项目，之后他会更新父项目的 POM 以包含子模块的引用。在 m2eclipse 出现之前，很难在 Eclipse 中使用 Maven 项目的层次特性。有了 m2eclipse，父子项目关系的底层细节被集成到了开发环境中。

14.9.3. 下载源码



如果中央 Maven 仓库包含了某个特定项目的源码构件，你可以从仓库下载这份源码然后在 Eclipse 环境中使用它。当你正在 Eclipse 中调试一个复杂的问题的时候，没有什么能比在 Eclipse 调试器中的第三方依赖上右击然后研究源码来的更方便的了。选择该选项之后，m2eclipse 会尝试着从 Maven 仓库下载源码构件。如果不能取得源码构件，你应该去问项目的维护者，让他上传适当的 Maven 源码至中央 Maven 仓库。

14.9.4. 打开项目页面



一个 Maven POM 包含一些开发者可能需要查阅的很有价值的 URL。它们包括项目的 web 页面，源代码仓库的 URL，如 Hudson 之类的持续集成系统的 URL，问题追踪系统的 URL。如果这些 URL 在项目的 POM 中存在，m2eclipse 就能在浏览器中打开这些项目页面。

14.9.5. 解析依赖



你可以配置项目让它从 workspace 中解析依赖。这种配置改变了 Maven 定位依赖构件的方式。如果项目被配置成从 workspace 解析依赖构件，这些构件就不需要存在于你的本地仓库。假设项目 a 和项目 b 都在同一个 Eclipse workspace 中，项目 a 依赖于项目 b。如果 workspace 依赖解析被关闭了，项目 a 的 Maven 构建只有在项目 b 的构件存在于本地仓库时才会成功。如果 workspace 依赖解析开启了，m2eclipse 就通过 eclipse workspace 解析这个依赖。换句话说，当 workspace 依赖解析开启的时候，项目之间的相互关联不需要通过本地仓库安装。

你也可以关闭依赖管理。这种配置的效果是告诉 m2eclipse 停止管理你项目的 classpath，也会从你项目中移除 Maven 依赖 classpath 容器。如果你这么做了，管理你项目的 classpath 就全靠你自己了。

14.10. 使用 Maven 仓库进行工作



m2eclipse 也提供了一些工具使得使用 Maven 仓库变得容易一些。这些工具提供的功能包括：

- 搜索构件
- 搜索 Java 类
- 为 Maven 仓库编制索引

14.10.1. 搜索 Maven 构件和 Java 类



m2eclipse 为 Eclipse Navigation 菜单添加几个项目，使搜索 Maven 构件和 Java 类变得容易。点击 Navigate 菜单就能使用这些选项，如 [Figure 14.21, “搜索构件和类”](#)：

Figure 14.21. 搜索构件和类

注意在 [Figure 14.21, “搜索构件和类”](#) 中在 Eclipse Navigate 菜单下面可用的选项名为 Open Maven POM 和 Open Type from Maven。Open Maven POM 选项允许你在 Maven 仓库中搜索 POM，如 [Figure 14.22, “搜索一个 POM”](#)：

Figure 14.22. 搜索一个 POM

选择一个构件然后点击 OK，这个构件的 POM 在 Eclipse 被打开以备浏览或者编辑。当你需要快速看一下某个构件的 POM 的时候，该功能十分方便。

Navigate 菜单中第二个 m2eclipse 选项名为 Open Type from Maven。该特性允许你通过名称在远程仓库中搜索一个 Java 类。打开这个对话框，键入‘factorybean’你就能看到名字带有 FactoryBean 的很多类，如 [Figure 14.23, “在仓库中搜索类”](#)：

Figure 14.23. 在仓库中搜索类

这是一个很能节省时间的特性，有了它，手工在 Maven 仓库中搜索构件中的类成为了过去。如果你需要使用一个特定的类，就打开 Eclipse，至菜单 Navigate 然后搜索类。m2eclipse 会显示一个搜索结果构件的列表。

14.10.2. 为 Maven 仓库编制索引



Maven 索引视图允许你手动的浏览远程仓库的 POM 并在 Eclipse 中打开它们。要查看这个视图，打开 View → Show View → Other，在搜索框中键入单词“maven”，你应该能看到一个名为 Maven 索引的视图，如 [Figure 14.24, “打开](#)

[Maven 索引视图](#)”：

Figure 14.24. 打开 Maven 索引视图

选择这个视图然后点击 OK。你将会看到如 [Figure 14.25, “Maven 索引视图”](#)的 Maven 索引视图。

Figure 14.25. Maven 索引视图

此外, [Figure 14.26, “从索引视图定位一个 POM”](#)展示手动导航至 Maven 索引视图之后, 定位一个 POM。

Figure 14.26. 从索引视图定位一个 POM

在找到 apache-camel 构件之后, 双击它会将在 Eclipse 中打开, 以浏览或编辑。这些特性使得在 Eclipse 中操作远程仓库变得更快更方便。过去一些年你可能已经花了很多时间来手工的进行这些操作——通过浏览器访问仓库, 下载构件然后使用 grep 程序查找类和 POM——你会发现 m2eclipse 是一种受欢迎的更好的变化。

14.11. 使用基于表单的 POM 编辑器



m2eclipse 的最新版本有个基于表单的 POM 编辑器, 能让你通过一个易用的 GUI 接口来编辑项目 pom.xml 的每一个部分。要打开 POM 编辑器, 点击项目的 pom.xml 文件。如果你为 pom.xml 文件定制了编辑器, POM 编辑器不是默认的编辑器, 你可以在这个文件上右击然后选择“Open With... / Maven POM Editor”。POM 编辑器会显示 Overview 标签页如 [Figure 14.27, “idiom-core 的 POM 编辑器的 Overview 标签页”](#)。

一个针对 Maven 的常见的抱怨是, 在十分复杂的多模块项目构件中, 它让开发人员面对十分巨大的 XML 文档。虽然本书的作者相信这只是为类似 Maven 的工具带来的弹性所付出的小小的代价, 但图形化的 POM 编辑器这样的工具能让用户在不知道 Maven POM 背后的 XML 结构的情况下就能使用 Maven。

Figure 14.27. idiom-core 的 POM 编辑器的 Overview 标签页

[Figure 14.27, “idiom-core 的 POM 编辑器的 Overview 标签页”](#)中展示的项目的 artifactId 是 idiom-core。你会注意到项目 idiom-core 的大部分字段是空的。POM 编辑器中没有 groupId 或者 version, 也没有 SCM 信息。这是因为 idiom-core 从一个名为 idiom 的父项目中继承了大部分的信息。如果我们在 POM 编辑器中打开父项目的 pom.xml 文件, 我们会看到如 [Figure 14.28, “idiom 父项目的 POM 编辑器的 Overview 标签页”](#)的 Overview 标签页。

整个 POM 编辑器中各种各样的列表条目上的“打开文件夹”图标说明对应的条目在 Eclipse workspace 中存在，“jar”图标说明构件引用了 Maven 仓库。你可以双击这些条目，在 POM 编辑器中打对应的 POM。这对模块，依赖，插件和其它对应于 Maven 构件的元素都是有效的。POM 编辑器中一些部分中带下划线的字段代表了超链接，可以用来为对应的 Maven 构件打开 POM 编辑器。

Figure 14.28. idiom 父项目的 POM 编辑器的 Overview 标签页

在这个父 POM 中，我们看到了 groupId 和 version 的定义，它还提供了 idiom-core 项目所没有很多信息。POM 编辑器会给你看你能够编辑的 POM 内容，它不会给你看任何继承来的值。如果你想在 POM 编辑器中看 idiom-core 项目的有效 POM，你可以使用 POM 编辑器右上角工具栏的“Show Effective POM”图标，该图标的样子是蓝色 M 字母上面有一个左括弧与等于标记。它会在 POM 编辑器中为项目 idiom-code 载入有效 POM 如 [Figure 14.29, “idiom-core 的有效 POM”](#)。

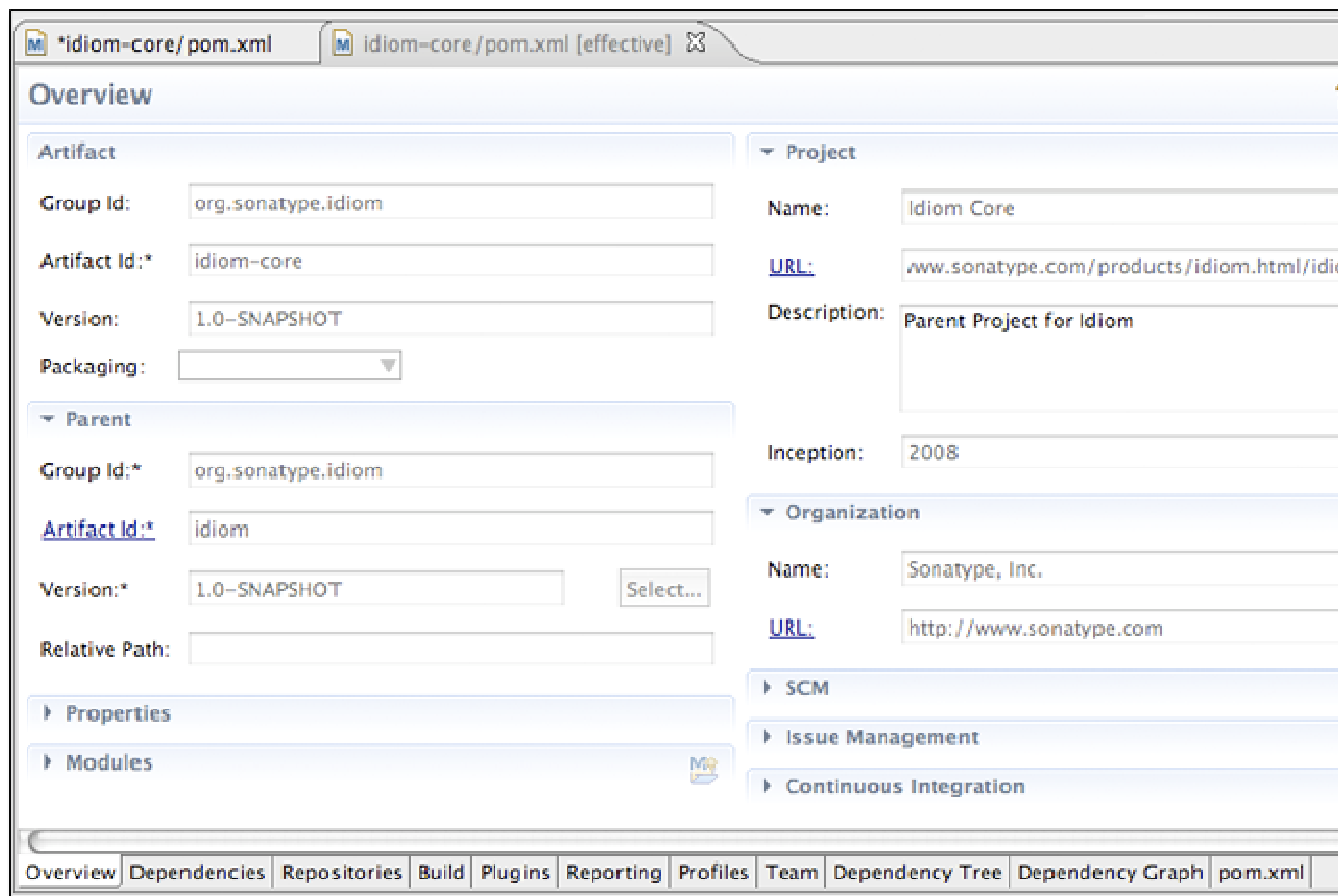


Figure 14.29. idiom-core 的有效 POM

这个有效 POM 归并了 idiom-core 的 POM 和它祖先的 POM(父，祖父，等等。)，它其实就是使用了“mvn help:effective-pom”命令为 idiom-core 编辑器显示有效值。由于 POM 编辑器显示了很多不同 POM 归并而来的组合视图，因此这个有

效 POM 编辑器是只读的，你不能更新这个有效 POM 视图中的任何字段。如果你正在观察 [Figure 14.27, “idiom-core 的 POM 编辑器的 Overview 标签页”](#) 中的 idiom-core 项目的 POM 编辑器，你还能够使用编辑器右上角工具栏上的“Open Parent POM”图标来导航至它的父项目的 POM。POM 编辑器显示了很多来自 POM 的不同信息。最后一个标签页将 pom.xml 显示为一个 XML 文档。在 [Figure 14.30, “POM 编辑器的 Dependencies 标签页”](#) 中有一个依赖标签页，它暴露了一个易用的接口以添加和编辑你项目的依赖，以及 POM 的 dependencyManagement 部分。m2eclipse 插件中的依赖管理屏幕也集成了构件搜索功能。你可以用搜索框来取得“Dependency Details”部分的字段信息。如果你想知道关于某个构件更多的信息，可以使用“Dependency Details”部分的工具栏的“Open Web Page”图标来查看项目的 web 页面。

Figure 14.30. POM 编辑器的 Dependencies 标签页

[Figure 14.31, “POM 编辑器的 Build 标签页”](#) 所示的 build 标签页能让你访问 build 元素的内容。从这个标签你能够自定义源代码目录，添加扩展，改变默认目标名称，以及添加资源目录。

Figure 14.31. POM 编辑器的 Build 标签页

我们只是展示了 POM 编辑器功能的一个很小的子集。如果你对余下的标签页感兴趣，请下载并安装 m2eclipse 插件。

14.12. 在 m2eclipse 中分析项目依赖



最新版本 m2eclipse 的 POM 编辑器提供了一些依赖工具。这些工具承诺要改变人们维护及监视项目传递性依赖的方式。Maven 的主要吸引力之一是它能够管理项目的依赖。如果你正在编写一个依赖于 Spring Framework 的 Hibernate 3 集成的应用程序，你所要做的仅仅是依赖来自中央 Maven 仓库的 spring-hibernate3 构件。Maven 会读取这个构件的 POM 然后添加所有必要的传递性依赖。虽然一开始这是一个吸引人们使用 Maven 的强大特性，但当项目有数十个依赖，每个依赖又有数十个传递性依赖的时候这就变得令人费解。

当你依赖于一个项目，这个项目有一个编写得很差的 POM，它未能将依赖标记为可选，或者当你开始遇到传递性依赖之间的冲突，这个时候问题就出现了。如果你有一个需求要将类似于 commons-logging 或 servlet-api 的依赖排除，又或者你需要弄清楚为什么在某个特定的范围下一个特定的依赖显现了，通常你需要从命令行调用 dependency:tree 和 dependency:resolve 目标来追踪那些令人不愉快的传递性依赖。

这个时候 m2eclipse 的 POM 编辑器就便捷多了。如果你打开一个有很多依赖的项目，你可以打开 Dependency Tree 标签页并查看显示为两列的依赖如 [Figure 14.32, “POM 编辑器的 Dependency Tree 标签页”](#)。面板的左边显示树状的依赖。树的第一层包含了你项目的直接依赖，每个下一层依赖列出了依赖的依赖。

左边的这一块是了解某个特定的依赖如何进入你项目的已解析依赖的很强大方式。面板的右边显示所有已解析的依赖。这是在所有冲突和范围已解决后的有效依赖的列表，也就是你项目将用来编译，测试和打包的有效依赖列表。

Figure 14.32. POM 编辑器的 Dependency Tree 标签页

Dependency Tree 标签页这一特性非常有价值，因为它能被用作一个侦测工具来找出某个特定的依赖是如何进入已解析的依赖列表的。编辑器中的搜索和过滤功能使得搜索和浏览项目的依赖变得十分容易。你可以使用编辑器工具栏的搜索框和“Dependency Hierarchy”及“Resolved Dependencies”部分的“排序”和“过滤”图标来查看依赖。[Figure 14.33, “在依赖树中定位依赖”](#)展示了当你点击“Resolved Dependencies”列表中的 commons-logging 的时候会发生什么。当“Dependencies Hierarchy”部分的过滤器被开启的时候，在已解析依赖上点击的时候，面板左边的依赖树会被过滤，以显示所有对已解析依赖起作用的节点。如果你正在试图去除一个已解析依赖，你可以使用这个工具来找出什么依赖（以及什么传递性依赖）对这个已解析的依赖起作用。换句话说，如果你要从你的依赖集合中去除类似于 commons-logging 这样的依赖，那么 Dependency Tree 标签页就是你想要使用的工具。

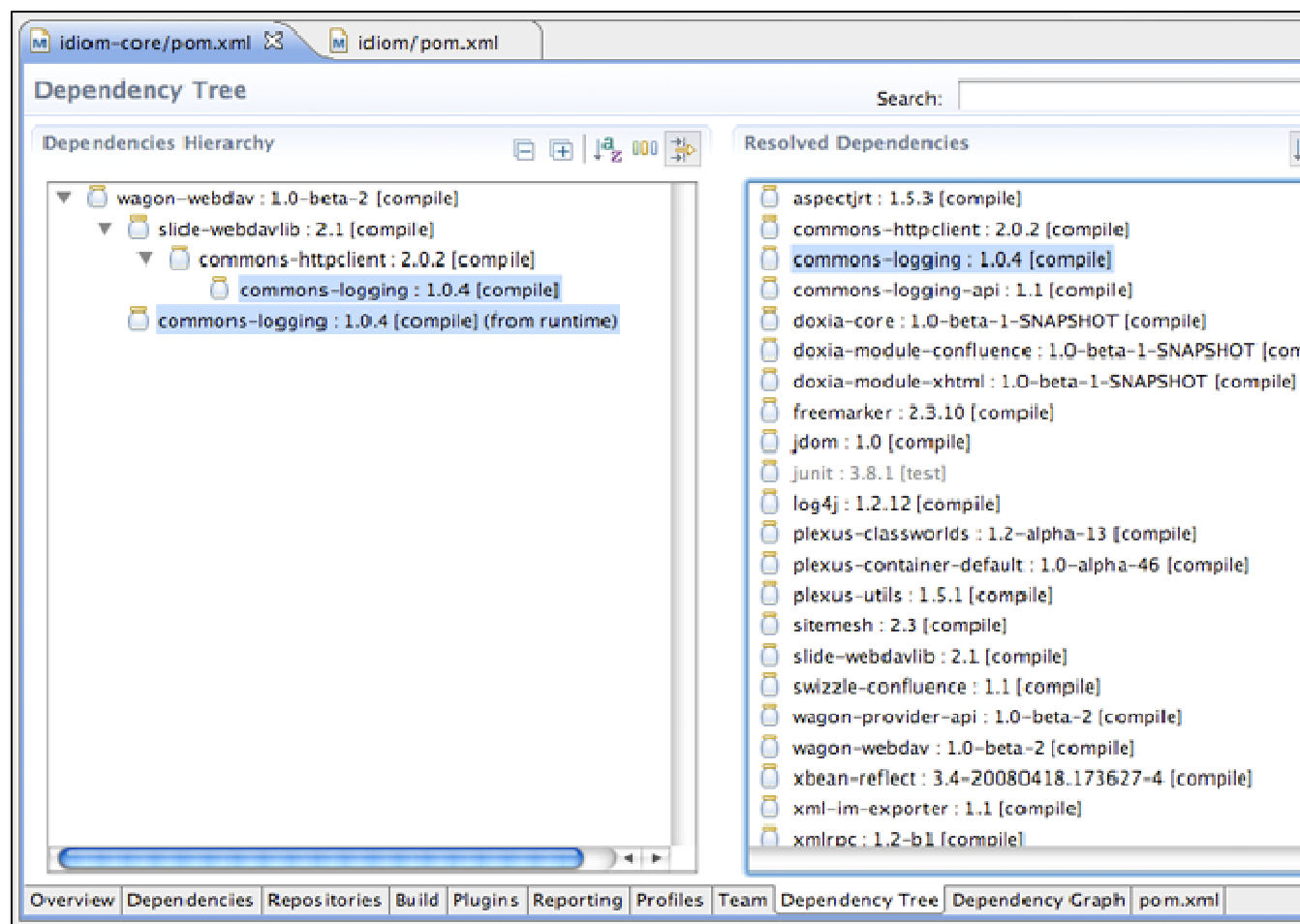


Figure 14.33. 在依赖树中定位依赖

m2eclipse 还能让你以图的形式查看你项目的依赖。[Figure 14.34, “以图的形式查看项目的依赖”](#)展示了 idiom-core 的依赖。最顶上的方框就是 idiom-core 项目，其它的依赖都在它的下面。直接依赖与顶层方框直接相连，传递性依赖则都最终连接这些直接依赖。你可以在图中选择一个节点，跟它相连的依赖会被标亮，或者你可以使用页面顶部的搜索框来寻找匹配的节点。注意每个图节点上的“打开文件夹”图标说明这个构件存在与 Eclipse workspace 中，而“jar”图标说明这个节点的构件指向了 Maven 仓库。

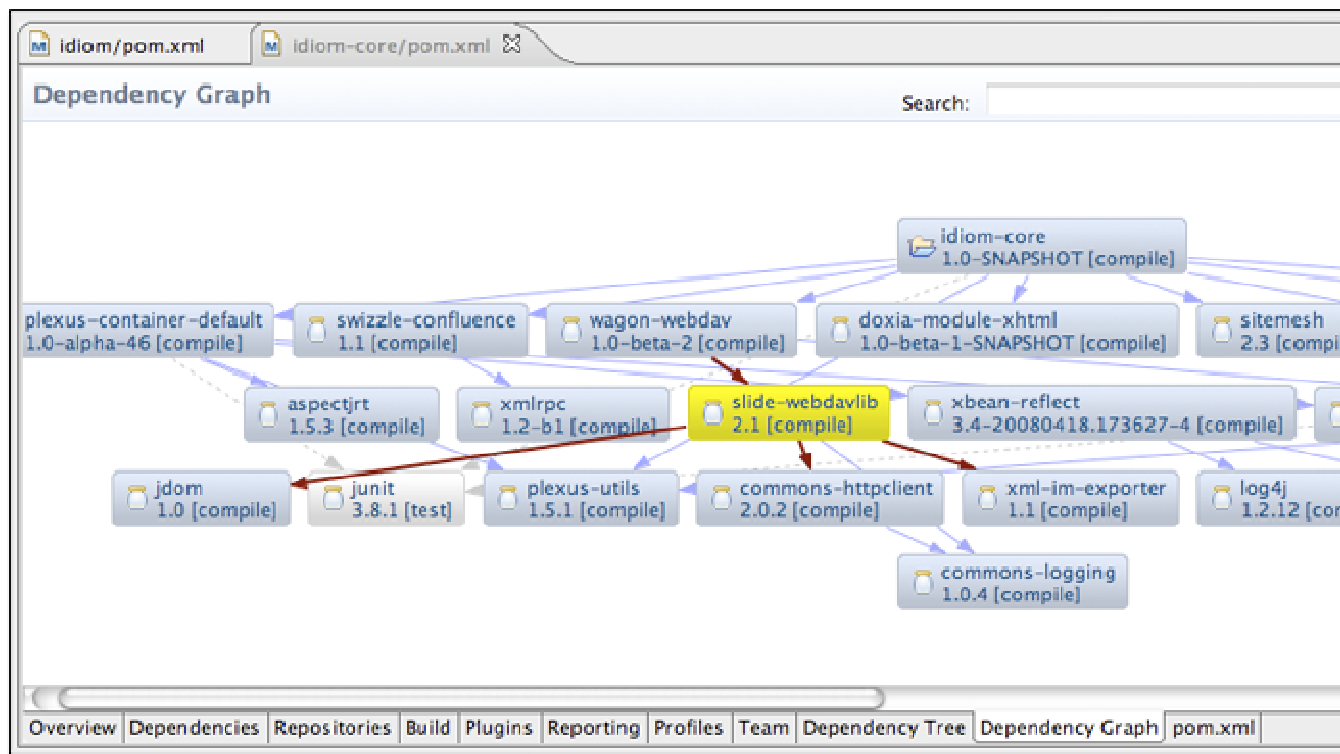


Figure 14.34. 以图的形式查看项目的依赖

通过在编辑器中右击可以改变图的表现形式。你可以选择显示构件的 id, group id, version, scope, 或者卷起节点文本, 显示图标。[Figure 14.35, “依赖图的放射状布局”](#)展示了和 [Figure 14.34, “以图的形式查看项目的依赖”](#)一样的图，但是用了放射状布局。

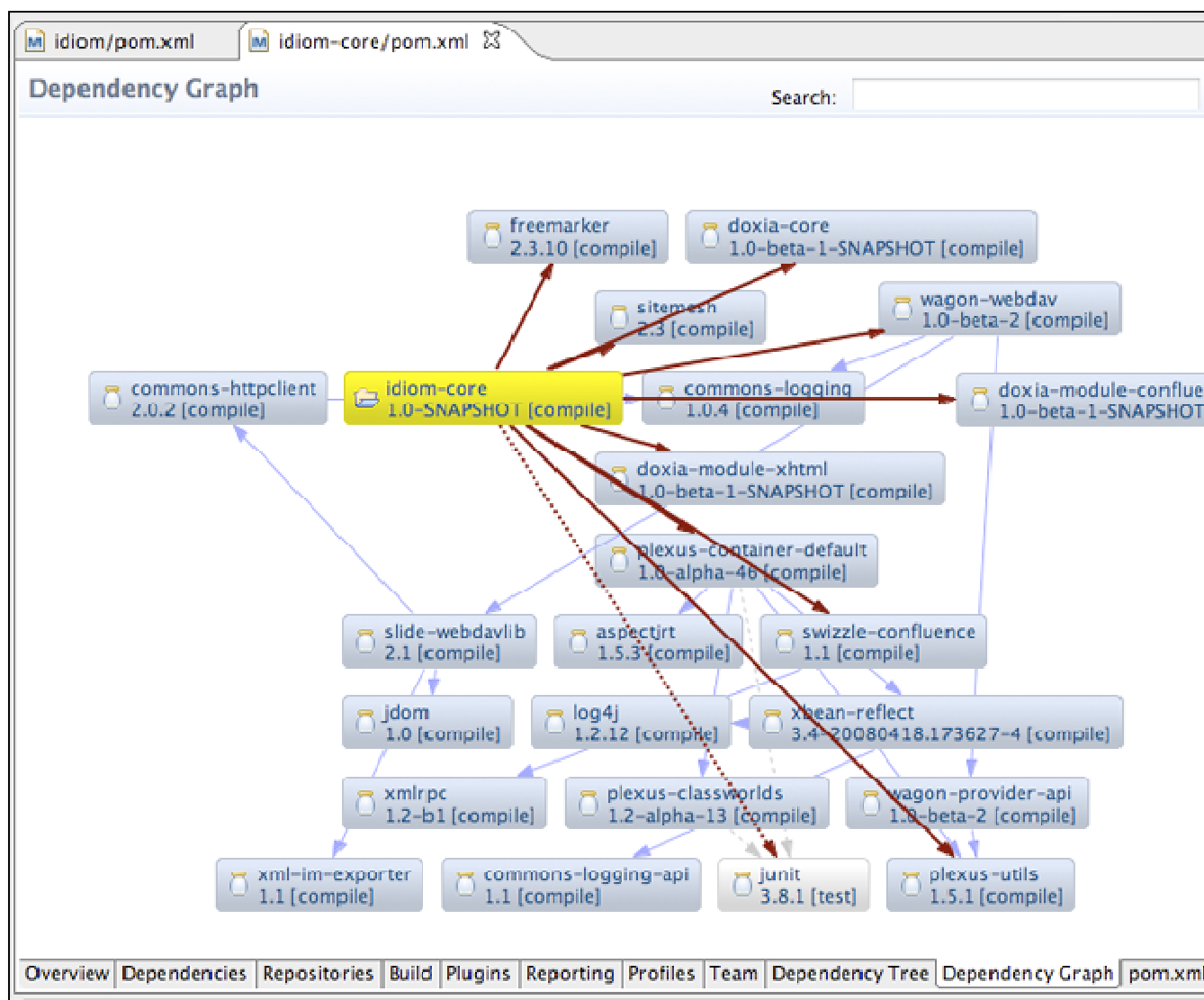


Figure 14.35. 依赖图的放射状布局

14.13. Maven 选项



对于使用 Maven 开发来说,调整 Maven 首选项和一些 Maven 参数是非常重要的,m2eclipse 能让你通过 Eclipse 中的 Maven 首选项页面来调整一些选项。典型情况下当在命令行使用 Maven 的时候,这些首选项是可以在目录`~/.m2`或者命令行选项中设置的。m2eclipse 能让你在 Eclipse IDE 中访问一些最重要的选项。[Figure 14.36, “Eclipse 的 Maven 首选项”](#)展示了在 Eclipse 中的 Maven 首选项页面:

Figure 14.36. Eclipse 的 Maven 首选项

顶部的复选框提供的功能有:

- 在离线的状态下运行 Maven, 关闭从远程仓库进行下载的功能。

- 在 Maven 控制台中开启调试输出
- 从远程 Maven 仓库下载构件的源码 jar 文件
- 从远程 Maven 仓库下载构件的 JavaDoc Jar 文件
- 在启动的时候下载并更新本地的对远程仓库的索引

下一部分提供了一个弹出式菜单，让你选择当项目被引入以及项目的源码文件夹被更新的时候执行什么 Maven 目标。默认的目标名为 process-resources，它会拷贝并处理项目的资源文件至目标目录，以让项目可以随时打包。如果你需要运行一些自定义的目标以处理资源文件或者生成一些支持性配置，那么定制这个目标列表就非常有用。

如果你需要 m2eclipse 帮你选择一个目标，点击按钮 Select...，会看到“Goals”对话框。左边的 [Figure 14.37, “Maven 目标对话框”](#)对话框展示了默认 Maven 生命周期中所有阶段列表的一个目标对话框。

Figure 14.37. Maven 目标对话框

当你第一次看到这个目标对话框的时候，你可能会被它罗列的这么多目标所吓到。事实上，确实有数百个 Maven 插件做各种各样的事情，从生成数据库，到执行集成测试，到运行静态分析，到使用 XFire 生成 web service。目标对话框中，带有可选的目标的插件有超过 200 个，右边的对话框 [Figure 14.37, “Maven 目标对话框”](#)展示了一个 Tomcat Maven 插件的目标被标亮的“Goals”对话框。你可以通过在搜索框内输入文字来缩小可用目标的数量，当你输入文字的时候，m2eclipse 会收缩可用目标的列表，当然，这些目标包含了搜索框内的关键字。另外一个 Maven 选项页面是 Maven 安装配置页面如 [Figure 14.38, “Maven 安装选项页面”](#)：

Figure 14.38. Maven 安装选项页面

这个页面能让你往 Eclipse 环境中添加其它的 Maven 安装。如果你想要让 m2eclipse 插件使用一个不同版本的 Maven，你可以在这个页面配置多个 Maven 安装，这非常类似于在 Eclipse 中配置多个可运行的 Java 虚拟机。一个被称为 Maven 嵌入器的 Maven 的嵌入版本已经被指定了。这就是我们在 Eclipse 中运行 Maven 的版本。如果你有另外一个 Maven 安装，而且你想要用它来运行 Maven 而不是 Maven 嵌入器，你可以实时的通过点击 Add..来添加另外的 Maven。[Figure 14.38, “Maven 安装选项页面”](#)展示了一个列出 Maven 嵌入器，Maven 2.0.9，和 Maven 2.1-SNAPSHOT 的配置页面。

安装配置页面也允许你指定全局 Maven settings 文件的位置。如果你不在这个页面指定该文件的位置，Maven 会使用位于所选 Maven 安装目录的 conf/settings.xml 作为默认全局配置文件。你也可以自定义用户 settings 文件的位置，默认它位于 ~/.m2/settings.xml，你还可以自定义你的本地 Maven 仓库位置，其默认值是 ~/.m2/repository。

Eclipse 选项中还能开启一个装饰器，它的名字是 Maven 版本装饰器。这个选项可以让你在 Eclipse 包浏览器中看到项目的当前版本，如 [Figure 14.39, “开启 Maven 版本装饰器”](#)：

Figure 14.39. 开启 Maven 版本装饰器

要开启这个选项，只要选中 Maven 版本修饰器选项，如在 [Figure 14.39, “开启 Maven 版本装饰器”](#)中标亮的。如果 Maven 版本修饰器没有开启，项目只会在包浏览器中列出它的名称和相对路径，如 [Figure 14.40, “没有 Maven 版本装饰器的包浏览器”](#)：

Figure 14.40. 没有 Maven 版本装饰器的包浏览器

开启了 Maven 版本装饰器之后，项目的名称将会包括当前的项目版本，如 [Figure 14.41, “开启了 Maven 版本装饰器的包浏览器”](#)：

Figure 14.41. 开启了 Maven 版本装饰器的包浏览器

这个特性十分有用，你能很方便的看到项目的版本而不再需要打开 POM 去找 version 元素。

14.14. 小结



m2eclipse 不仅仅是一个为 Eclipse 添加 Maven 支持的简单插件，它集成得非常全面，从创建新项目到定位第三方依赖，难易度的降低都是数量级的。m2eclipse 是对于一个通晓中央 Maven 仓库这一丰富语义资源的 IDE 的第一步尝试。随着越来越多的人开始使用 m2eclipse，更多的项目将会发布 Maven Archetype，更多的项目会看到往 Maven 仓库发布源码构件的价值。如果你曾经在没有一个能理解 Maven 层级结构关系的工具的情况下，尝试一起使用 Eclipse 和 Maven，你就会知道支持嵌套项目的这一特性，对于 Eclipse IDE 和 Maven 的集成来说，是至关重要的。

Chapter 16. 仓库管理器

[16.1. 简介](#)

[16.1.1. Nexus 历史](#)

[16.2. 安装 Nexus](#)

[16.2.1. 从 Sonatype 下载 Nexus](#)

[16.2.2. 安装 Nexus](#)

[16.2.3. 运行 Nexus](#)

[16.2.4. 安装后检查单](#)

[16.2.5. 为 Redhat/Fedora/CentOS 设置启动脚本](#)

[16.2.6. 升级 Nexus 版本](#)

[16.3. 使用 Nexus](#)

[16.3.1. 浏览仓库](#)

[16.3.2. 浏览组](#)

[16.3.3. 搜索构件](#)

[16.3.4. 浏览系统 RSS 源](#)

[16.3.5. 浏览日志文件和配置](#)

[16.3.6. 更改你的密码](#)

[16.4. 配置 Maven 使用 Nexus](#)

[16.4.1. 使用 Nexus 中央代理仓库](#)

[16.4.2. 使用 Nexus 作为快照仓库](#)

[16.4.3. 为缺少的依赖添加仓库](#)

[16.4.4. 添加一个新的仓库](#)

[16.4.5. 添加一个仓库至一个组](#)

[16.5. 配置 Nexus](#)

[16.5.1. 定制服务器配置](#)

[16.5.2. 管理仓库](#)

[16.5.3. 管理组](#)

[16.5.4. 管理路由](#)

[16.5.5. 网络配置](#)

[16.6. 维护仓库](#)

[16.7. 部署构件至 Nexus](#)

[16.7.1. 部署发布版](#)

[16.7.2. 部署快照版](#)

[16.7.3. 部署第三方构件](#)

16.1. 简介



仓库管理器有两个服务目的：首先它的角色是一个高度可配置的介于你的组织与公开 Maven 仓库之间的代理，其次它为你的组织提供了一个可部署你组织内部生成的构件的地方。

代理 Maven 仓库有很多好处。对于一开始使用 Maven 的情况来说，通过为所有的来自中央 Maven 仓库的构件安装一个本地的缓存，你将加速组织内部的所有构建。如果有开发人员想要下载 Spring Framework 的 2.5 版本，并且你在使用 Nexus，那些依赖（以及依赖的依赖）只需要从远程仓库下载一次。如果有一个高速的 Internet 网络连接，这看起来没什么大不了的，但是如果你一直要求你的开发人员去下载几百兆的第三方依赖，那么真正节省的时间将会是 Maven 检查依赖新版本以及下载依赖的时间。通过本地仓库提供 Maven 依赖服务可以节省数百的 HTTP 请求，在大型的多项目构建中，这样可以为一次构件节省几分钟的时间。

除了简单的时间和带宽的节省，仓库管理器为组织提供了一种控制 Maven 下载的机制。你可以详细的设置从公开仓库包含或排除特定的构件。能够控制从核心 Maven 仓库的下载对于很多组织来说是经常是一个必要前提，它们需要维护一个组织中使用依赖的严格控制。一个想要标准化某个如 Hibernate 或者 Spring 依赖版本的组织可以通过在仓库管理器中仅仅提供一个特殊版本的构件来加强这种标准。还有一些组织可能关心确保所有外部的依赖拥有和组织的法律规范相容的许可证。如果一个企业生产了一个分发应用程序，它们可能想要确定没有人不小心添加了一个涉及 GPL 许可证的依赖。仓库管理器为那些需要确信总体架构和政策实施的组织提供了这一层次的控制。

除了控制对远程仓库的访问以外，仓库管理器也为 Maven 的全面使用提供了一些很至关重要的东西。除非你希望你组织的每一个成员下载并构建一个单独的内部项目，否则你会希望为开发人员和部门之间提供一种共享内部项目构件的快照版本和发布版本的机制。Nexus 为你的组织提供了这样的部署目标。在你安装了 Nexus 之后，你可以开始使用 Maven 让它部署快照版和发布版至一个由 Nexus 管理的定制仓库。

16.1.1. Nexus 历史



Tamas Cservenak 在 2005 年 12 月开始为 Proximity 工作，当时他正想办法将它自己的系统和由 Hungarian ISP 提供的慢得难以置信的 ADSL 连接隔离开。Proximity 以一个简单 web 应用的形式启动，用来为有网络连接问题的小型组织代理构件。为 Maven 构件创建一个对中央核心仓库的本地的命令驱动的缓存，能让组织访问来自中央核心仓库的构件，而且它同时也能确保这些构件不会通过很慢的 ADSL 连接来下载，要知道很多开发人员在使用这个连接。在 2007 年，Sonatype 请求 Tamas 帮助创建一个类似的名为 Nexus 的产品。Nexus 目前可以被认为是 Proximity 逻辑上的下一个步伐。

Nexus 目前有一个活动的开发团队包括 Tamas Cservenak, Max Powers, Dmitry

Platonoff 和 Brian Fox。Nexus 的关于索引的部分代码也同时在 m2eclipse 中被使用，这些代码目前由 Eugene Kuleshov 开发。

16.2. 安装 Nexus



16.2.1. 从 Sonatype 下载 Nexus



你可以从 <http://nexus.sonatype.org> 找到关于 Nexus 的信息。要下载 Nexus，访问 <http://nexus.sonatype.org/downloads/>。点击下载链接，下载适用于你平台的存档文件。Nexus 目前有 ZIP 和 Gzip 归档的 TAR 两种可用形式。

16.2.2. 安装 Nexus



安装 Nexus 很简单，打开 Nexus 归档文件至一个目录。如果你正在本地工作站上安装 Nexus，以测试它的运行，你可以将其安装至你的用户目录，或者随便什么地方；Nexus 没有任何硬编码的目录，它能在任意目录运行。如果你下载了一个 ZIP 归档文件，运行：

```
$ unzip nexus-1.0.0-beta-3-bundle.zip
```

如果你下载了 GZip 归档的 TAR 文件，运行：

```
$ tar xvzf nexus-1.0.0-beta-3-bundle.tgz
```

如果你正在一个服务器上安装 Nexus，你可能想要使用的目录不是你的用户目录。在 Unix 机器上，这可能是 /usr/local/nexus-1.0.0-beta-3 和一个指向 Nexus 目录的符号链接 /usr/local/nexus。使用一个通用的符号链接来指向 Nexus 的某个特定版本是一个普遍的实践，它能让你更容易的将 Nexus 更新至最新的版本。

```
$ sudo cp nexus-1.0.0-beta-3-bundle.tgz /usr/local
$ cd /usr/local
$ sudo tar xvzf nexus-1.0.0-beta-3-bundle.tgz
$ ln -s nexus-1.0.0-beta-3 nexus
```

虽然对于 Nexus 的运行来说这不是必要的，你可能想要设置一个环境变量 NEXUS_HOME，指向 Nexus 的安装目录。本章通过 \${NEXUS_HOME} 的形式来引用这个位置。

16.2.3. 运行 Nexus



当你启动 Nexus，就是启动一个 web 服务器，它的默认地址是 localhost:8081。Nexus 在一个名为 Jetty 的 servlet 容器中运行，它使用一个名为 [Tanuki Java Service](#)

[Wrapper](#) 的本地服务包裹器启动。这个服务包裹器可以被配置成以 Windows 服务或 Unix 守护线程的形式运行 Nexus。要启动 Nexus，你需要为你的平台找到合适的启动脚本。要查看可用平台的列表，查看 `${NEXUS_HOME}/bin/jsw` 目录的内容。

下面的例子展示了使用 Mac OSX 的脚本启动 Nexus。首先我们列出 `${NEXUS_HOME}/bin/jsw` 的内容以查看可用的平台，然后用 **chmod** 命令使这个 bin 目录的内容可执行。Mac OSX 包裹器通过调用 **app start** 启动，然后我们 **tail** 在 `${NEXUS_HOME}/container/logs` 中的 `wrapper.log`。Nexus 会初始化自己然后打印出一条信息说明它正在监听 `localhost:8081`。

```
$ cd Nexus
$ ls ./bin/jsw/
aix-ppc-32/          linux-ppc-64/          solaris-sparc-32/
aix-ppc-64/          linux-x86-32/          solaris-sparc-64/
hpux-parisc-32/      linux-x86-64/          solaris-x86-32/
hpux-parisc-64/      macosx-universal-32/  windows-x86-32/
$ chmod -R a+x bin
$ ./container/bin/jsw/macosx-universal-32/app start
Nexus Repository Manager...
$ tail -f container/logs/wrapper.log
INFO ... [ServletContainer:default] - Started
SelectChannelConnector@0.0.0.0:8081
```

到目前为止，Nexus 已经开始运行并监听端口 8081。要使用 Nexus，启动一个 web 浏览器然后输入 URL：<http://localhost:8081/nexus>。点击 web 页面右上角的“Log In”链接，你应该看到如下的登陆对话框。

默认的 NEXUS 用户名和密码是“admin” 和 “admin123”。

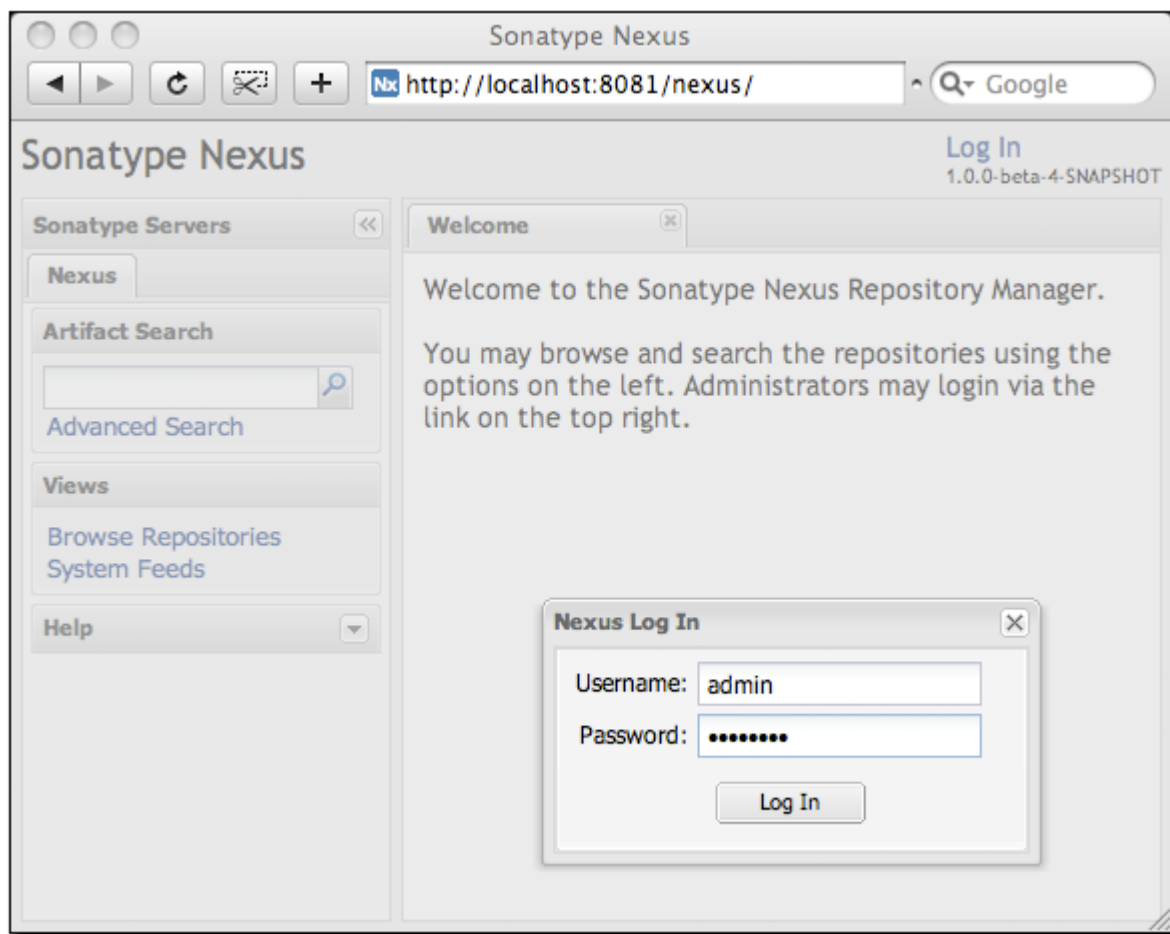


Figure 16.1. Nexus 登录窗口(默认 用户名/密码 是 admin/admin123)

16.2.4. 安装后检查单



Nexus 带有默认的密码和仓库索引设置，它们都需要更改以满足你的安装需要（以及安全）。安装完并运行了 Nexus 后，你需要确认你完成了下列任务：

更改管理员密码和 Email 地址

默认的管理密码是 admin123。对一个全新的 Nexus 安装，你首先要做的是更改这个密码。要更改默认的管理员登陆名"admin"及密码"admin123"，在浏览窗口的左边导航菜单中的 Security 部分点击 Change Password。

配置 SMTP 设置

Nexus 可以发送用来用户名和密码的 email，要开启这个特性，你需要用 SMTP 主机和端口配置 Nexus，以及相关的 Nexus 用来连接邮件服务器的认证参数。为此，载入如 [Section 16.5, “定制服务器配置”](#)中的服务器配置对话框。

开启远程索引下载

Nexus 带有三个重要的代理仓库，有中央 Maven 仓库，Apache 快照仓库，和 Codehaus 快照仓库。它们中的每一个仓库都包含了数千（或数万）的构件，下载每个仓库的所有内容是不切实际的。处于这个原因，大部分仓库维护了一个编录了整个内容的 Lucene 索引，以提供快速和有效的搜索。Nexus 使用这些远程索引搜索构件，但是默认设置我们关闭了索引下载。要下载远程索引，

- 点击 Administration 菜单下面的 Repositories，更改三个代理仓库的 Download Remote Indexes 为 true。你需要为此载入如 [Section 16.6, “维护仓库”](#) 中的对话框。
- 在每个代理仓库上右击然后选择 Re-index。这会触发 Nexus 下载远程的索引文件。

Nexus 下载整个索引可能需要好几分钟，但是一旦你下载好之后，你就能够搜索 Maven 仓库的所有内容了。

Note

Sonatype 想要确信没有创建一个会在默认情况下对中央仓库造成大量拥挤的产品。虽然大部分用户会想要开启远程索引下载，我们还是不想使之成为默认设置，当数百万用户下载一个新版本的 Nexus 继而不断的下载这个 21MB 的中央索引的时候，会制造对我们自己的拒绝服务攻击。如果你想要 Nexus 返回全部的搜索结果，你就必须显式的开启远程索引下载。

16.2.5. 为 Redhat/Fedora/CentOS 设置启动脚本



你可以将 Nexus 配置成自动启动，通过将 app 脚本拷贝到 /etc/init.d 目录。在一个 Redhat 变种的 Linux 系统上（Redhat, Fedora, 或者 CentOS），以 root 用户执行下列操作：

- 复制 `{NEXUS_HOME}/bin/jsw/linux-ppc-64/app`，或 `{NEXUS_HOME}/bin/jsw/linux-x86-32/app`，或 `{NEXUS_HOME}/bin/jsw/linux-x86-64/app` 至 `/etc/init.d/nexus`。
- 使 `/etc/init.d/nexus` 脚本可运行 —— **`chmod 755 /etc/init.d/nexus`**。
- 编辑该脚本，更改下列变量。
 - 更改 APP_NAME 为 "nexus"
 - 更改 APP_LONG_NAME 为 "Sonatype Nexus"
 - 添加一个变量 NEXUS_HOME 指向你的 Nexus 安装目录
 - 添加一个变量 PLATFORM 内容包含 linux-x86-32, linux-x86-64, 或 linux-ppc-64
 - 更改 WRAPPER_CMD 为 `{NEXUS_HOME}/bin/jsw/{PLATFORM}/wrapper`

- 更改 WRAPPER_CONF 为 \${NEXUS_HOME}/conf/wrapper.conf
- 更改 PIDDIR 为 /var/run.
- 添加一个 JAVA_HOME 变量指向你的本地 Java 安装
- 添加 \${JAVA_HOME}/bin 至 PATH
- (可选) 设置 RUN_AS_USER 为 "nexus". 如果你这么做, 你需要:
 - 创建一个 nexus 用户
 - 更改你的 nexus 安装目录的 Owner 和 Group 为 nexus

最后你应该有一个文件/etc/init.d/nexus, 它拥有如下的一些列配置属性 (假设你在/usr/local/nexus 安装 Nexus, 你在/usr/java/latest 安装了 Java):

```

JAVA_HOME=/usr/java/latest
PATH=${PATH}:${JAVA_HOME}/bin
APP_NAME="nexus"
APP_LONG_NAME="Sonatype Nexus"
NEXUS_HOME=/usr/local/nexus
PLATFORM=linux-x86-64
WRAPPER_CMD="${NEXUS_HOME}/bin/jsw/${PLATFORM}/wrapper"
WRAPPER_CONF="${NEXUS_HOME}/conf/wrapper.conf"
PRIORITY=
PIDDIR="/var/run"
#RUN_AS_USER=nexus

```

这个脚本有一个适当的 **chkconfig** 指令, 因此要添加 Nexus 为一个服务, 你要做的是运行如下的命令:

```

$ cd /etc/init.d
$ chkconfig --add nexus
$ chkconfig --levels 345 nexus on
$ service nexus start
Starting Sonatype Nexus...
$ tail -f /usr/local/nexus/logs/wrapper.log

```

第二个命令添加 nexus 为一个服务, 可以由 **service** 命令启动和停止, 可以由 **chkconfig** 命令管理。**chkconfig** 管理/etc/rc[0-6].d 中的符号链接, 当操作系统重启或者在运行级别中转换时, 它们控制服务的启动和停止。第三个命令添加 nexus 至运行级别 3, 4, 和 5。service 命令启动 Nexus, 最后的命令追踪 wrapper.log 以验证 Nexus 成功启动。如果 Nexus 成功启动了你应该看到一个信息告诉你 Nexus 正在端口 8001 监听 HTTP 连接。

16.2.6. 升级 Nexus 版本



升级一个已安装的 Nexus 版本十分容易。每个 Nexus 版本有两个可用的归档文件可下载。完整的归档文件包含 Nexus 应用程序, Nexus 启动脚本, 以及用来保存仓库索引和远程仓库本地缓存的工作目录。如果你大量的使用 Nexus, 这个工

作目录会包含数 G 的构件，你不会希望在每次升级 Nexus 的时候必须重新创建这个仓库。升级下载文件被创建成为用户提供一个方便的形式升级 Nexus，它会保存 Nexus 数据；升级下载文件只包含 Nexus 应用程序代码。第一次你安装 Nexus 的时候，你下载完全的 Nexus 分发版，当你想要升级你的 Nexus 安装，同时保留你的仓库数据的时候，你只要下载升级分发版。

要升级 Nexus，只要下载“upgrade”分发版，而非“bundle”分发版。升级分发版的内容存储在一个包含 nexus 版本号（如 nexus-1.0.0-beta-3）的文件夹中。这个文件夹可以解开至 \$NEXUS_HOME/runtime/apps，不用覆盖当前安装版本的内容。

```
$ cd $NEXUS_HOME/runtime/apps
$ unzip nexus-1.0.0-beta-3-upgrade.zip
```

如果你下载了 GZip 归档的 TAR 文件，运行：

```
$ cd $NEXUS_HOME/runtime/apps
$ tar xvfz nexus-1.0.0-beta-3-upgrade.tgz
```

当升级归档文件解压至 \$NEXUS_HOME/runtime/apps 后，你必须从之前的 Nexus 版本复制配置文件至新安装的版本。从 \$NEXUS_HOME/runtime/apps/nexus/conf/nexus.xml 复制 nexus.xml 至 \$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-3/conf。你应该也复制所有你自定义的日志配置文件 log4j.properties 和 jul-logging.properties。在你从当前的 Nexus 版本复制了配置文件至新版本的 Nexus 后，停止 Nexus 服务器。

现在，你需要重命名 \$NEXUS_HOME/runtime/apps/nexus 目录为一个反映它旧版本号的名称。比如，在这个例子中 \$NEXUS_HOME/runtime/apps/nexus 将成为 \$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-3。然后，将你新版本改为 \$NEXUS_HOME/runtime/apps/nexus。在 Unix 系统上，你需要创建一个符号链接 \$NEXUS_HOME/runtime/apps/nexus 指向 \$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-2。在 Windows 系统上，你可能需要复制 \$NEXUS_HOME/runtime/apps/nexus-1.0.0-beta-2 至 \$NEXUS_HOME/runtime/apps/nexus。在你用新版本的 Nexus 交换了旧版本的 Nexus 后，你应该能使用启动脚本启动 Nexus。Nexus 启动之后，检查 \$NEXUS_HOME/logs/wrapper.log。Nexus 初始化之后，它会打印出 Nexus 版本号。

16.3. 使用 Nexus



Nexus 为那些只需要搜索，浏览构件，以及查阅系统 RSS 源的用户提供了匿名访问。匿名访问级别更改了导航菜单，以及当你在一个仓库上右击时可用的选项。这种只读访问显示了如 [Figure 16.2, “匿名用户的 Nexus 界面”](#) 的用户界面。

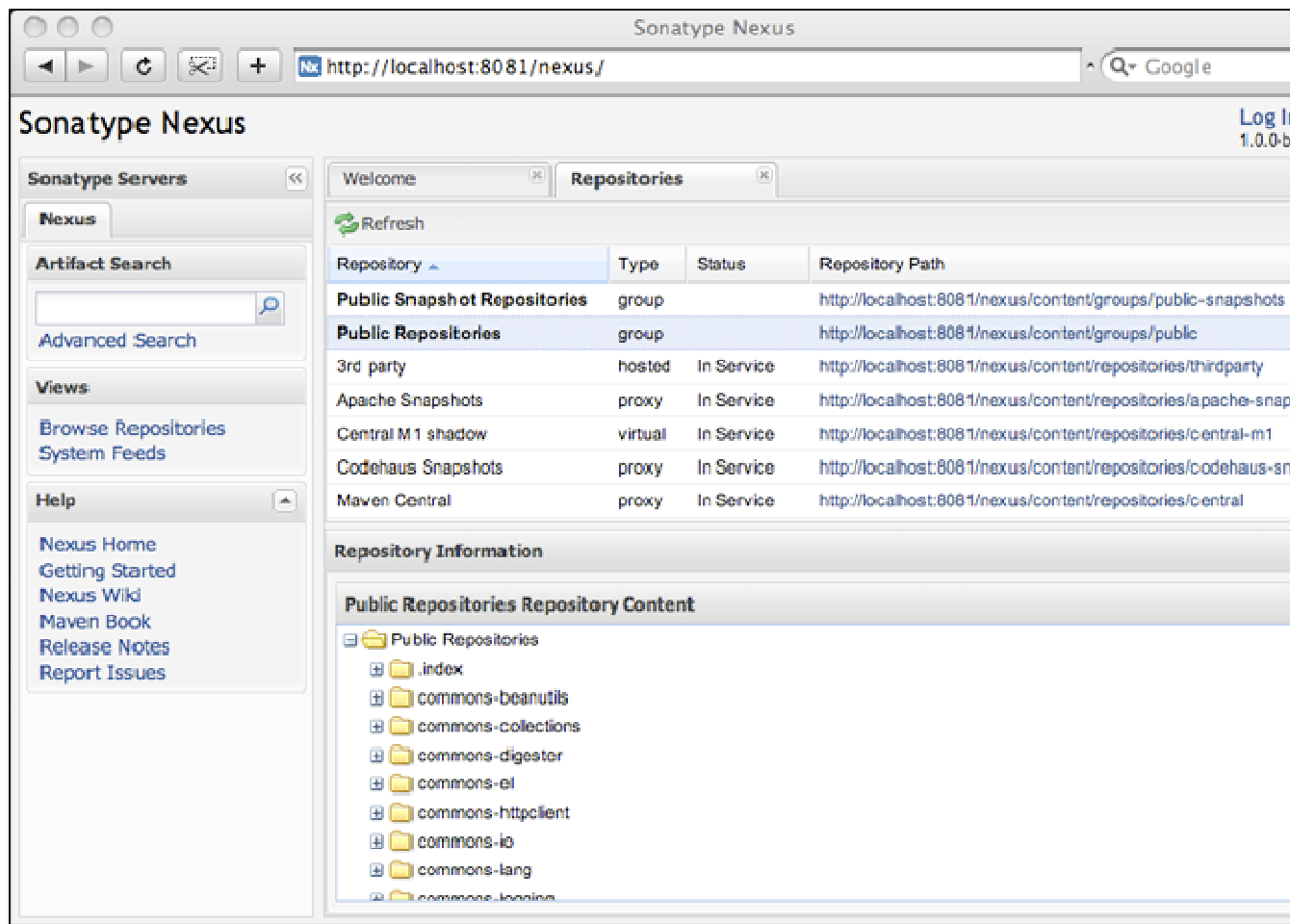


Figure 16.2. 匿名用户的 Nexus 界面

16.3.1. 浏览仓库



Nexus 最直接的用途之一就是浏览 Maven 仓库的结构。如果你点击 Views 菜单下的 Browse Repositories 菜单项。[Figure 16.3, “浏览一个 Nexus 仓库”](#)中的上面一半给你显示了带有仓库类型和仓库状态的组列表和仓库列表。

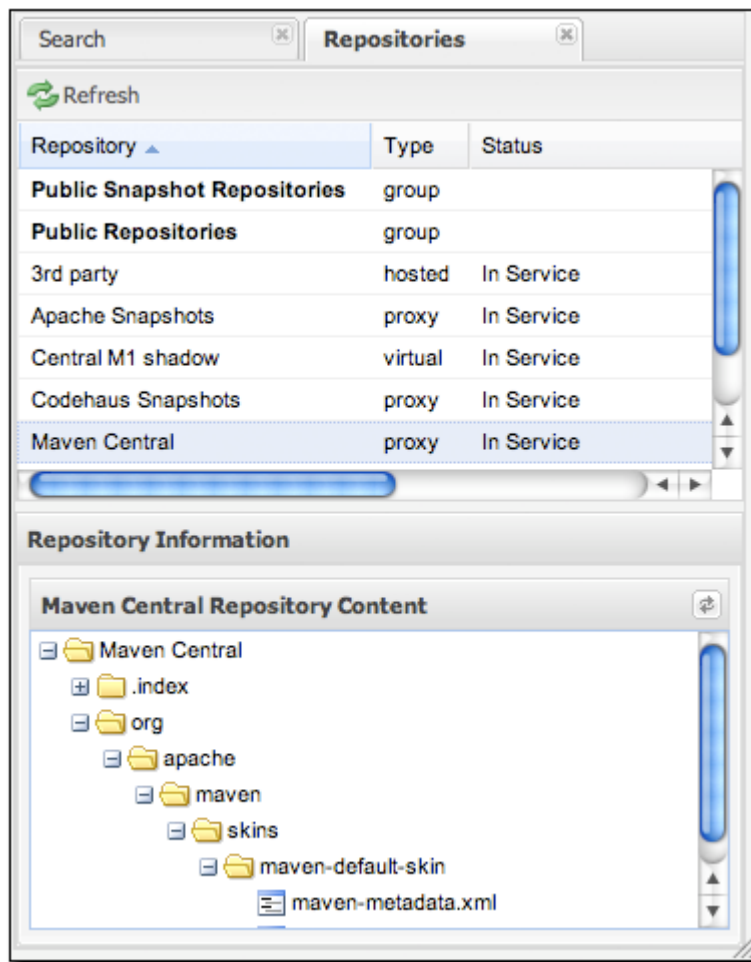


Figure 16.3. 浏览一个 Nexus 仓库

当你浏览一个仓库的时候，你可以在任意一个文件上右击然后直接下载到你本地。这能让你手工获取某个特定的构件，或者在浏览器中检查一个 POM 文件。

16.3.2. 浏览组



Nexus 包含排序好的仓库组，它们能让你通过一个单独的 URL 来暴露一系列的仓库。通常情况下，一个组织会指向两个默认的 Nexus 组：Public Repositories 组和 Public Snapshot Repositories 组。很多最终用户不需要知道哪些构件来自哪个特定的仓库，他们只需要能够浏览公共仓库组就可以了。为了支持这个用例，Maven 允许你浏览一个 Nexus 组的内容，它就像是一归并而来的树状的仓库。[Figure 16.4, “浏览一个 Nexus 组”](#)显示了这个浏览界面，其中一个 Nexus 组被选中以浏览。对用户体验来说，浏览一个 Nexus 组和浏览一个 Nexus 仓库没有任何区别。

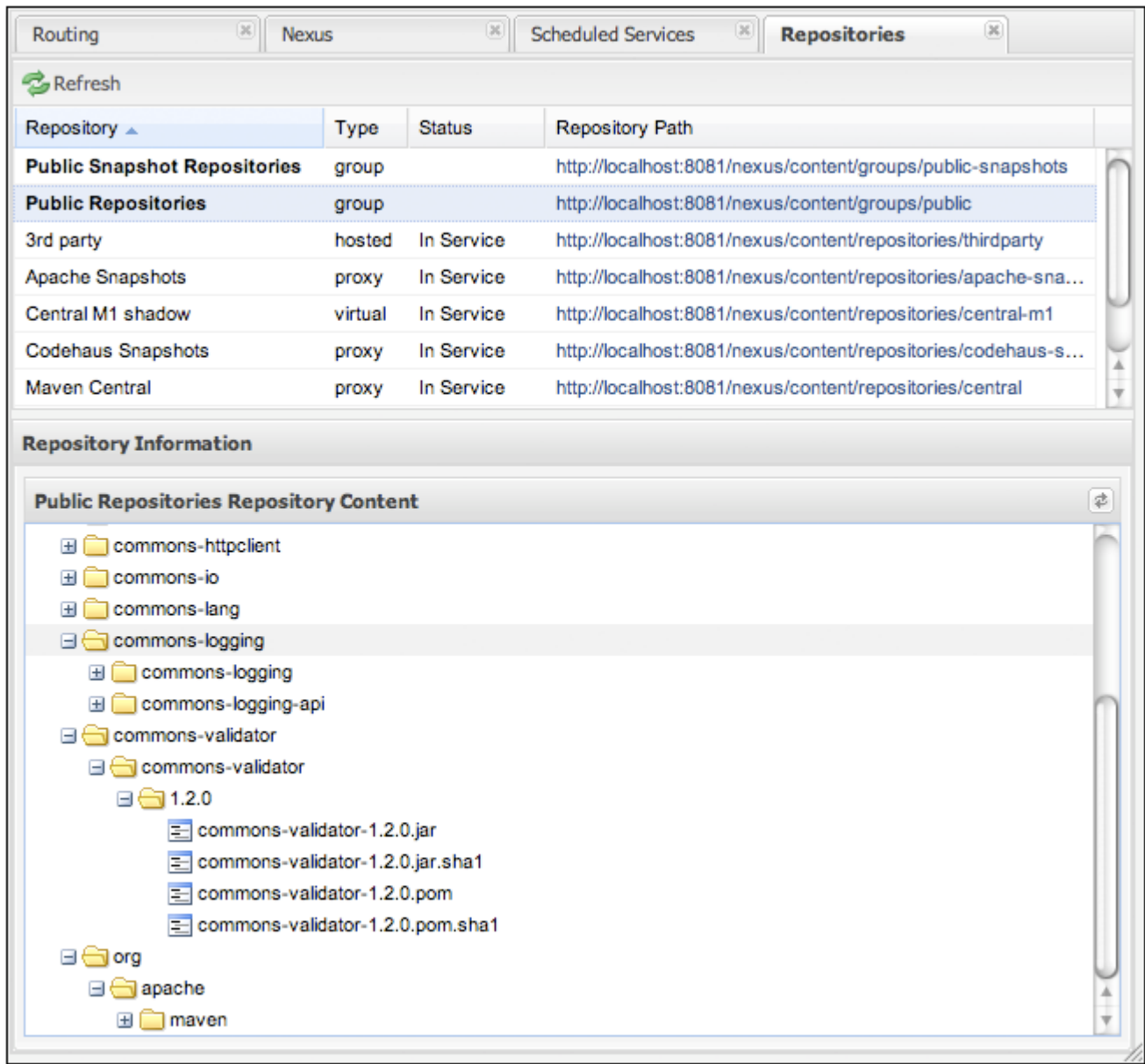


Figure 16.4. 浏览一个 Nexus 组

16.3.3. 搜索构件



在左边的导航区域，紧靠放大镜有一个构件搜索输入框。要通过 groupId 或者 artifactId 搜索一个构件，输入一些文本然后点击放大镜。输入字段 "maven" 然后点击放大镜会产生如 [Figure 16.5, “关键词为 "maven" 的构件搜索结果”](#) 的搜索结果。

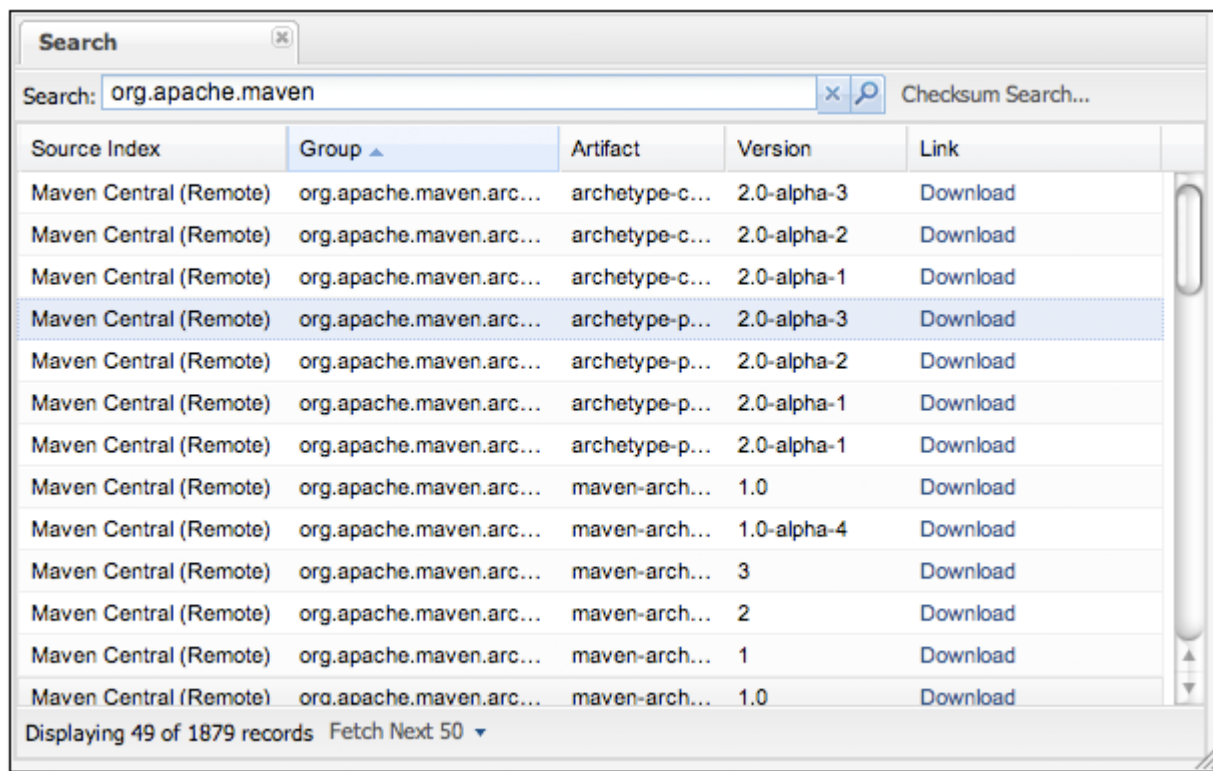


Figure 16.5. 关键词为"maven"的构件搜索结果

在你找出你在要找的构件之后，你可以点击 **Download** 链接来下载这个构件。Nexus 每次为你显示 50 条结果，并且为你浏览其它搜索结果在底部提供了链接。如果你更喜欢看到所有匹配构件的列表，你可以在搜索结果面板底部的下拉菜单中选择 **Fetch All**。

除了通过一个 `groupId` 或者一个 `artifactId` 搜索，Nexus 还有一个功能能让你通过校验和来搜索一个构件。

Warning

让我来猜一下？你安装了 Nexus，使用了搜索框，输入了一个构件的 `group` 的名字，按下搜索，然后什么都没看见。没有结果。Nexus 默认不会去获取远程仓库索引，你需要为那三个 Nexus 自带的仓库激活远程索引的下载。没有这些索引，没有东西可以搜索。你可以在 [Section 16.2.4, “安装后检查单”](#) 中查找激活索引下载的指令。

16.3.4. 浏览系统 RSS 源



Nexus 提供了一些捕捉系统事件的 RSS 源，你可以通过点击 **View** 菜单下的 **System Feeds** 来浏览它们。如 [Figure 16.6, “浏览 Nexus 系统信息源”](#) 中的面板。你可以使用这些简单的界面来浏览最近 Nexus 中发生的关于构件部署，构件缓存，

存储变化的报告。

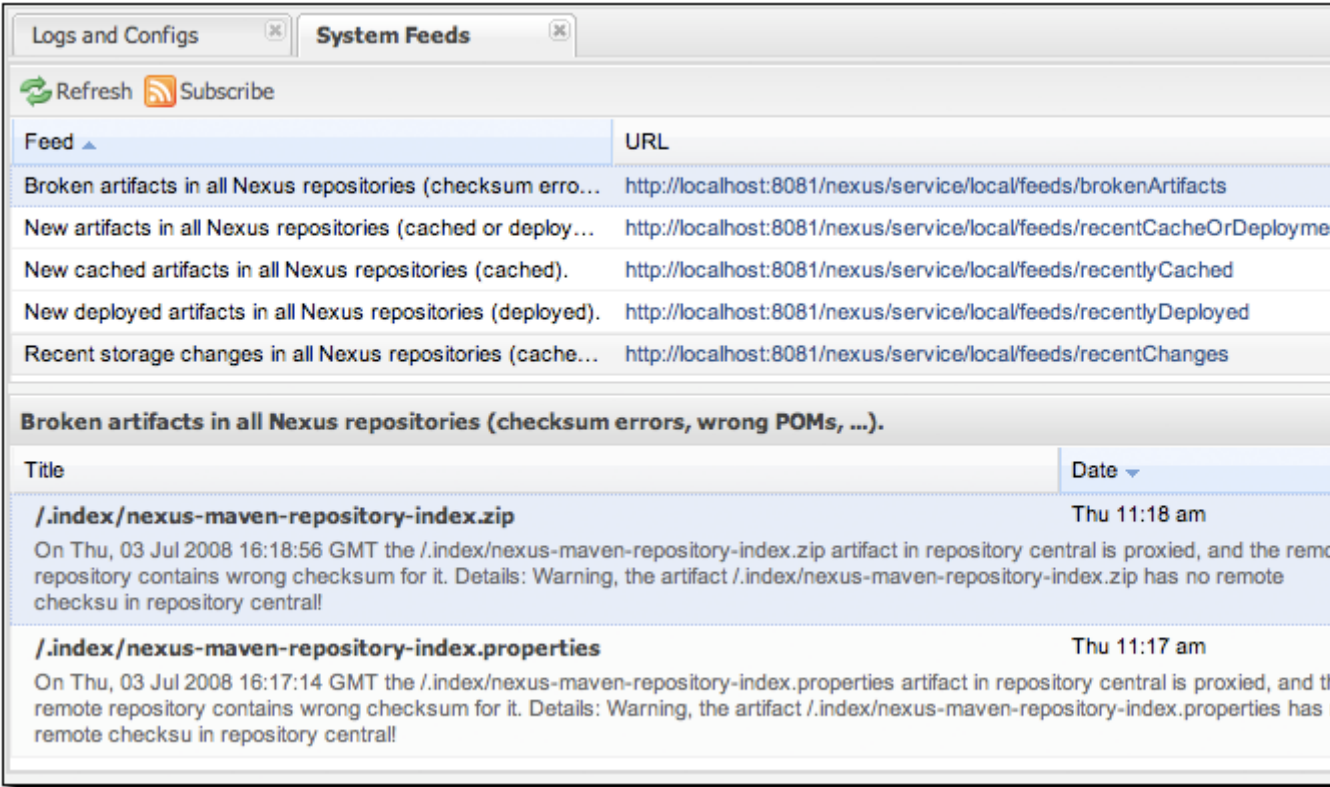


Figure 16.6. 浏览 Nexus 系统信息源

如果你正在一个很大的组织工作，很多开发团队往同样一个 Nexus 实例部署构件，这些信息源就非常有用。有了这样的准备，所有组织开发人员可以为新部署的构件订阅 RSS 信息源，以确保当一个新的发布版提交到 Nexus 后所有的人都知道。将系统事件暴露成 RSS 信息源也将大门向其他人开启，包括一些对该信息更富创意的使用，如将 Nexus 与外部的自动测试系统想连。要访问某个特定信息源的 RSS，在 System Feeds 观察面板中选择一个信息源然后点击 Subscribe 按钮。Nexus 会在你浏览器中载入这个 RSS 信息源，然后你可以在你最喜欢的 RSS 阅读器中订阅这个信息源。

在系统信息源视图中有 6 个可用的信息源，每一个信息源都有一个类似于下面的 URL：

http://localhost:8081/nexus/service/local/feeds/recentChanges

其中 recentChanges 将会被你试图阅读的信息源标识所替换。可能的系统信息源包括：

Table 16.1. 可用的系统信息源

信息源标识符	描述
brokenArtifacts	校验和不匹配，找不到校验和，不可用的 POM
recentCacheOrDeployments	所有仓库中有新的构件（从远程缓存的或者部署上去的）
recentlyCached	所有仓库中有新的缓存构件

信息源标识符	描述
recentlyDeployed	所有仓库中有新的部署的构件
recentChanges	所有缓存，部署，或者删除动作
systemRepositoryStatusChanges	自动或者用户发起的变更（服务失效和阻塞的代理）
systemChanges	启动 Nexus，更改配置，重新编制索引，以及属性重建

16.3.5. 浏览日志文件和配置



日志和配置文件只有在管理员用户的 **Views** 菜单中可见。点击该选项会看到如 [Figure 16.7, “浏览 Nexus 日志和配置文件”](#) 中的对话框。在这个屏幕你可以通过点击 **Download** 按钮旁边的下拉选择菜单来查看一下的日志和配置文件。

nexus.log

把它想成是一个 Nexus 的总体的应用程序日志。除非你是管理员用户，否则你可能不会对这个日志的信息有什么兴趣。如果你正试图调试一个错误，或者你有 Nexus 中未发现的 bug。你会使用这个日志查看器来诊断 Nexus 的问题。

nexus-rest-0.log

核心的 Nexus 服务实际上是一堆 REST 服务，你正使用的 UI 只是和这些 REST 服务交互以配置和查看 Nexus 的组及仓库。这个日志文件反映了由 Nexus UI 和 Nexus REST 服务交互所生成的活动。

nexus.xml

这个 XML 文件包含了大部分你所使用的 Nexus 实例的配置数据。它被存储在 `${NEXUS_HOME}/runtime/apps/nexus/conf/nexus.xml`。

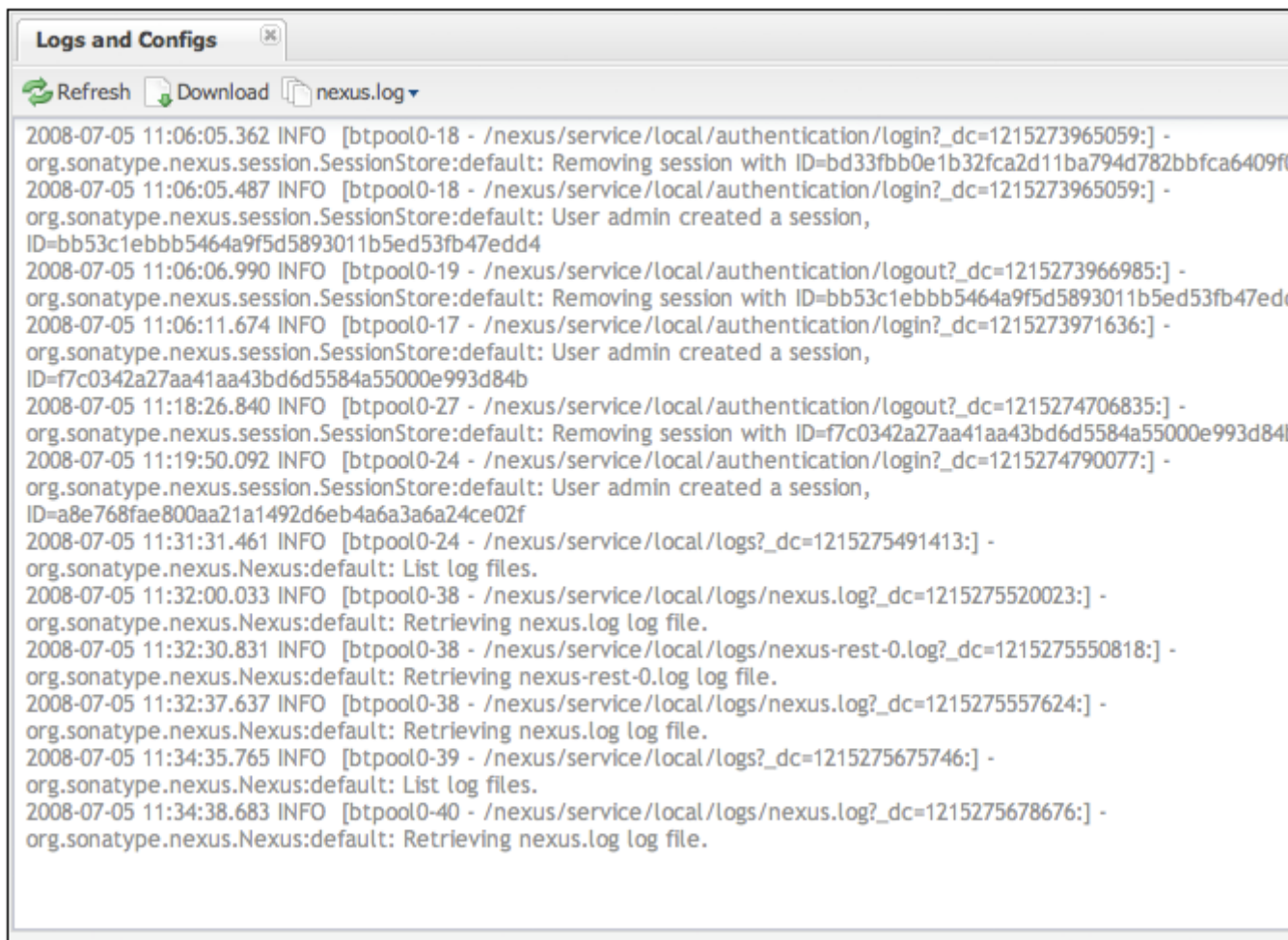


Figure 16.7. 浏览 Nexus 日志和配置文件

16.3.6. 更改你的密码



如果你拥有适当的安全权限，你还会在浏览器的左边看到一个可以更改你密码的选项。要更改你的密码，点击 **change password**，提供你现在的密码，然后输入一个新密码。当你点击 **Change Password** 后，你的 Nexus 密码就被更改了。



Figure 16.8. 更改你的 Nexus 密码

16.4. 配置 Maven 使用 Nexus



要使用 Nexus，你需要配置 Maven 去检查 Nexus 而非公共的仓库。为此，你需要编辑在你的 `~/.m2/settings.xml` 文件中的 `mirror` 配置。首先，我们会演示如何配置 Maven 去检查你的 Nexus 安装而非直接从中央 Maven 仓库获取构件。在我们覆盖了中央仓库并演示了 Nexus 可以工作之后，我们会转回来，提供一个更实际的，包含发布版和快照版的配置集合。

16.4.1. 使用 Nexus 中央代理仓库



要配置 Maven 去查阅 Nexus 而非中央 Maven 仓库，在你的 `~/.m2/settings.xml` 文件中添加如 [Example 16.1, “为 Nexus 配置 Maven Settings \(~/.m2/settings.xml\)”](#) 的 `mirror` 配置。

Example 16.1. 为 Nexus 配置 Maven Settings (`~/.m2/settings.xml`)

```
<?xml version="1.0"?>
<settings>
  ...
  <mirrors>
    <mirror>
      <id>Nexus</id>
      <name>Nexus Public Mirror</name>
      <url>http://localhost:8081/nexus/content/groups/public</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
  ...

```

</settings>

在你将 Nexus 配置成所有仓库的镜像之后，Maven 现在会从本地的 Nexus 安装查阅，而非去外面查阅中央 Maven 仓库。如果对 Nexus 有一个构件请求，本地的 Nexus 安装会提供这个构件。如果 Nexus 没有这个构件，Nexus 会从远程仓库获取这个构件，然后添加至远程仓库的本地镜像。

要测试 Nexus 如何工作的，从你的本地 Maven 仓库中删除一个目录，然后运行 Maven 构建。如果你删除了 ~/.m2/repository/org，你会删除一大堆的依赖（包括 Maven 插件）。下次你运行 Maven 的时候，你应该看到如下的信息：

```
$ mvn clean install
...
Downloading: http://localhost:8081/nexus/content/groups/public/...
3K downloaded
```

这个输出应该能让你相信 Maven 正和你本地的 Nexus 通讯，而非向外面的中央 Maven 仓库获取构件。在你基于本地的 Nexus 运行过一些构建之后，你就可以浏览缓存在你本地 Nexus 中的内容。登陆 Nexus 然后点击导航菜单的左边的构件搜索。在搜索框中输入 "maven"，你应该能看到一些像下面的内容。

16.4.2. 使用 Nexus 作为快照仓库



[Section 16.4.1, “使用 Nexus 中央代理仓库”](#)中的 Maven 配置能让你使用 Nexus 公共组，这个组从 4 个由 Nexus 管理的仓库解析构件，但是它不让你查阅 public-snapshots 组，该组包括了 Apache 和 Codehaus 的快照版。要配置 Maven 让它为发布版和插件都使用 Nexus，你必须配置 Maven，通过往你的 Maven 文件 ~/.m2/settings.xml 中添加如下的镜像配置，使其查阅 Nexus 的组。

Example 16.2. 配置 Maven 使其为发布版和快照版使用 Nexus

```
<settings>
  <mirrors>
    <mirror>
      <!--This is used to direct the public snapshots repo in the
           profile below over to a different nexus group -->
      <id>nexus-public-snapshots</id>
      <mirrorOf>public-snapshots</mirrorOf>

      <url>http://localhost:8081/nexus/content/groups/public-snapshots</url>
    </mirror>
    <mirror>
      <!--This sends everything else to /public -->
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>http://localhost:8081/nexus/content/groups/public</url>
    </mirror>
  </mirrors>
</settings>
```

```
</mirror>
</mirrors>
<profiles>
  <profile>
    <id>development</id>
    <repositories>
      <repository>
        <id>central</id>
        <url>http://central</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>central</id>
        <url>http://central</url>
        <releases><enabled>true</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
  <profile>
    <!--this profile will allow snapshots to be searched when activated-->
    <id>public-snapshots</id>
    <repositories>
      <repository>
        <id>public-snapshots</id>
        <url>http://public-snapshots</url>
        <releases><enabled>false</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>public-snapshots</id>
        <url>http://public-snapshots</url>
        <releases><enabled>false</enabled></releases>
        <snapshots><enabled>true</enabled></snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
<activeProfiles>
```

```
<activeProfile>development</activeProfile>
</activeProfiles>
</settings>
```

在 [Example 16.2, “配置 Maven 使其为发布版和快照版使用 Nexus”](#) 中我们定义了两个 profile: development 和 public-snapshots。development profile 被配置成从 central 仓库下载构件, 通过一个假的 URL `http://central.public-snapshots` 被配置成从 public-snapshot 仓库下载构件, 通过一个假的 URL `http://public-snapshots`。这些假的 URL 被同一 settings.xml 文件中的两个 mirror 配置重写。第一个镜像被配置成覆盖 public-snapshots 仓库, 而使用 public-snapshots Nexus 组。第二个镜像覆盖所有其它的仓库, 而使用 public Nexus 组。有了这些配置, 所有的构建都会包含 public Nexus 组, 如果你想包含 public-snapshots 组, 你必须添加 public-snapshots 这个 Profile, 通过在命令行使用如下的 -P 标志。

```
$ mvn -Ppublic-snapshots clean install
```

16.4.3. 为缺少的依赖添加仓库

如果你已经将你的 Maven settings.xml 配置成使用 Nexus 作为所有公共仓库和所有公共快照仓库的镜像, 你可能会遇到一些项目不能够从你的本地 Nexus 获取需要的构件。这很常见, 因为你经常会构建一些在 pom.xml 中自定义一组 repositories 和 snapshotRepositories 的项目。如果你正在构建开源项目, 或者往你的配置中添加了自定义的第三方 Maven 仓库, 那么这种情况就会发生。作为一个例子, 让我们试试从我们签出的源代码构件 Apache Shindig。什么是 Apache Shindig? 对该例来说这不重要; 我们需要的是一个能很容易签出和构建的样例项目。如果你实在很想知道, Shindig 是在 Apache Incubator 中的一个围绕 Google 的 OpenSocial API 的项目。Shindig 目标是提供一个允许人们运行 OpenSocial 小工具的容器。它给我们提供了一个有趣的样例工程, 因为它有一些没有被加入到中央 Maven 仓库的组件, 于是依赖于一些自定义的 Maven 仓库, 使用 Shindig 我们可以向你展示当 Nexus 没有你要的构件的时候会发生什么, 以及你能够使用怎样的步骤来给 Nexus 添加仓库。

下面的样例假设你已经安装了 Subversion, 并且你正在命令行运行 Subversion。我们使用 Subversion 从 Apache Incubator 签出 Apache Shindig 然后尝试构建它。为此, 执行下面的命令:

```
$ svn co http://svn.apache.org/repos/asf/incubator/shindig/trunk shindig
... Subversion will checkout the trunk of Apache Shindig ...
$ cd shindig
$ mvn install
... Maven will build Shindig ...
Downloading:
http://localhost:8081/nexus/content/groups/public/caja/caja/r820/caja-r820.pom
```

```
Downloading:
http://localhost:8081/nexus/content/groups/public/caja/caja/r820/caja-r820.
jar
[INFO]
-----
[ERROR] BUILD ERROR
[INFO]
-----
[INFO] Failed to resolve artifact.

Missing:
-----
1) caja:caja:jar:r820

    Try downloading the file manually from the project website.

...
-----
1 required artifact is missing.

for artifact:
    org.apache.shindig:gadgets:war:1-SNAPSHOT

from the specified remote repositories:
    oauth (http://oauth.googlecode.com/svn/code/maven),
    central (http://central),
    apache.snapshots (http://people.apache.org/repo/m2-snapshot-repository),
    caja (http://google-caja.googlecode.com/svn/maven)
```

这个构建失败了因为它下载不到一个构件。这个构件有一个 `group` 标识符为 `caja`，`artifactId` 是 `caja`，版本是 `r820`。这是一个存在于自定义仓库 <http://google-caja.googlecode.com/svn/maven> 中的一个构件。Maven 没能够下载到这个构件是因为你的 `settings.xml` 被配置成指引所有的镜像至位于我们 Nexus 安装的 `public` 和 `public-snapshots` 组。即使 Apache Shindig 的 `pom.xml` 定义了一个仓库并且将其指向了 <http://google-caja.googlecode.com/svn/maven>，Nexus 不会从一个它不知道的仓库中去获取构件。事实上，关于这次构建有两个仓库 Nexus 不知道：`caja` 和 `oauth`。[Caja](#) 和 [OAuth](#) 是两个仍然处于开发中的类库。两个项目都被“发布”了，而且 Shindig 所依赖的版本当然不是快照版，但是这些项目没有被发布到中央 Maven 仓库。在我们能构建这个项目之前，我们需要想办法让 Nexus 知道这些仓库。

有两种方法可以解决这个问题。首先，你可以更改你的以 `settings.xml` 覆盖特定的仓库定义符。你可以更改 `settings.xml` 中的 `mirrorOf` 元素为 `"central"`，而非让 Nexus `public` 组 `mirrorOf` 所有的仓库。如果你这么做了，Maven 就会试图直接从 `oauth` 和 `caja` 仓库下载依赖。这行得通，因为 Maven 只会为那些匹配 `settings.xml` 中 `mirrorOf` 元素的仓库去查阅 Nexus。如果 Maven 看到一个仓库

定义符 `caja` 或者 `oauth`，而且没有在你的 `settings.xml` 中看到一个镜像，它会直接去连接这个仓库。

第二种方法，更有趣的选择是添加这些仓库至 Nexus，并且添加这些仓库至 `public` 组。

16.4.4. 添加一个新的仓库



要添加 `caja` 仓库，以管理员登陆 Nexus，在左边导航菜单 `Configuration` 部分中点击 `Repositories` 链接。点击这个链接后会看到一个窗口列出了所有 Nexus 所知道的仓库。之后你想要创建一个新的代理仓库。为此，点击在仓库列表正上方的 `Add` 链接。点击单词 `Add` 右边的朝下的箭头，会看到一个下拉菜单，带有选项：`Hosted`，`Proxy`，和 `Vitual`。既然你要创建一个代理仓库，点击 `Proxy`。之后，你会看到一个如 [Figure 16.9, “添加一个 Nexus 仓库”](#) 的页面。填充那些必填字段，Repository ID 为 `"caja"`，Repository Name 为 `"Google Caja"`。设置 Repository Policy 为 `"Release"`，以及 Remote Storage Location 为 <http://google-caja.googlecode.com/svn/maven>。

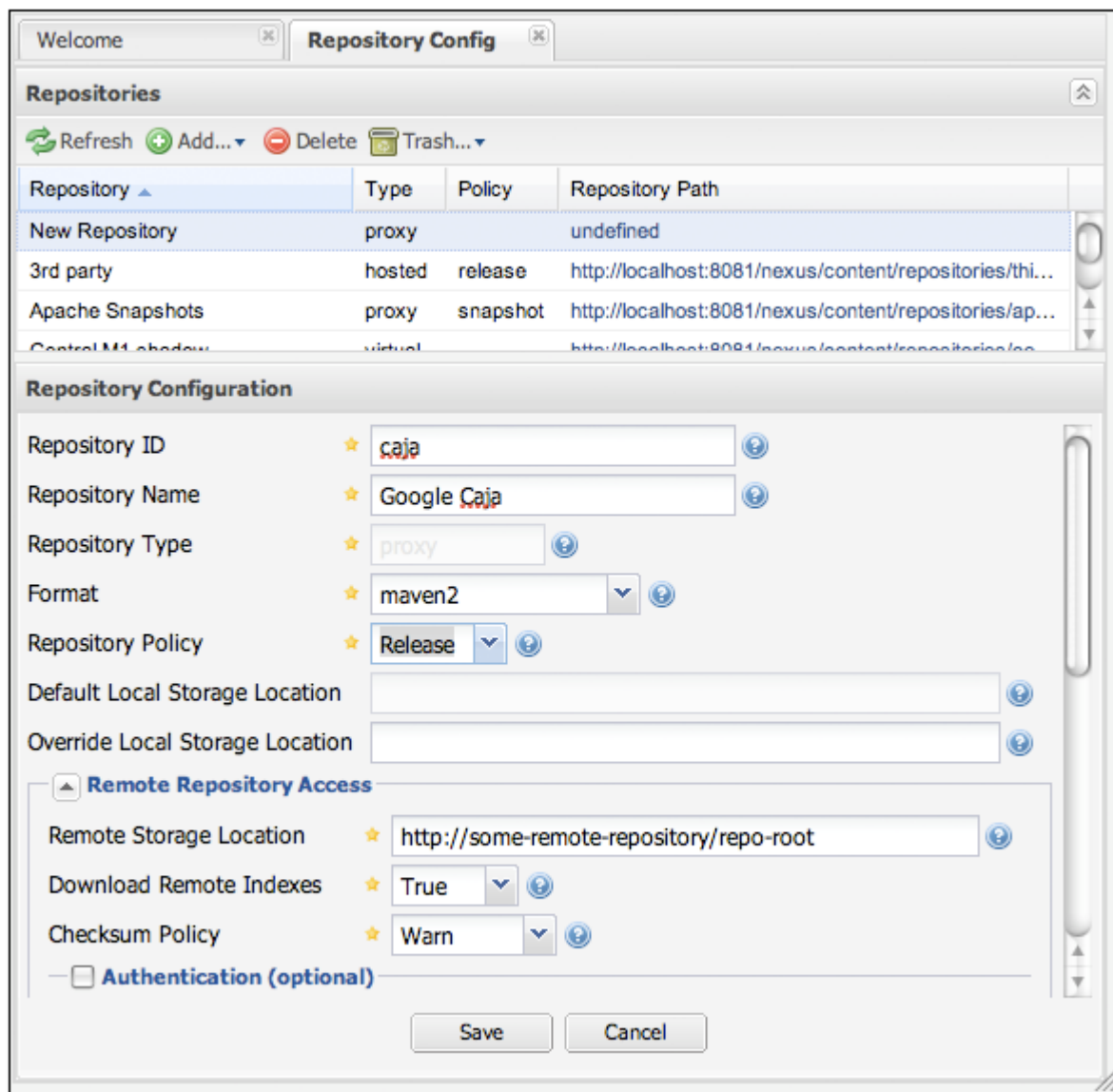


Figure 16.9. 添加一个 Nexus 仓库

在你填写完这个页面之后，点击 Save 按钮。Nexus 就会接受这个 caja 代理仓库的配置。为 oauth 仓库重复同样的工作。创建一个仓库，Repository ID 为 oauth，选择 Release policy，Remote Storage Location 为 <http://oauth.googlecode.com/svn/code/maven>。

16.4.5. 添加一个仓库至一个组



下一步你需要做的是添加这些新的仓库至 public Nexus 组。为此，点击左边导航菜单中 Configuration 部分的 Groups 链接。当你看到组管理页面后，点击 public repositories 组，你应该能看到如 [Figure 16.10, “添加新的仓库至一个 Nexus 组”](#) 的页面。

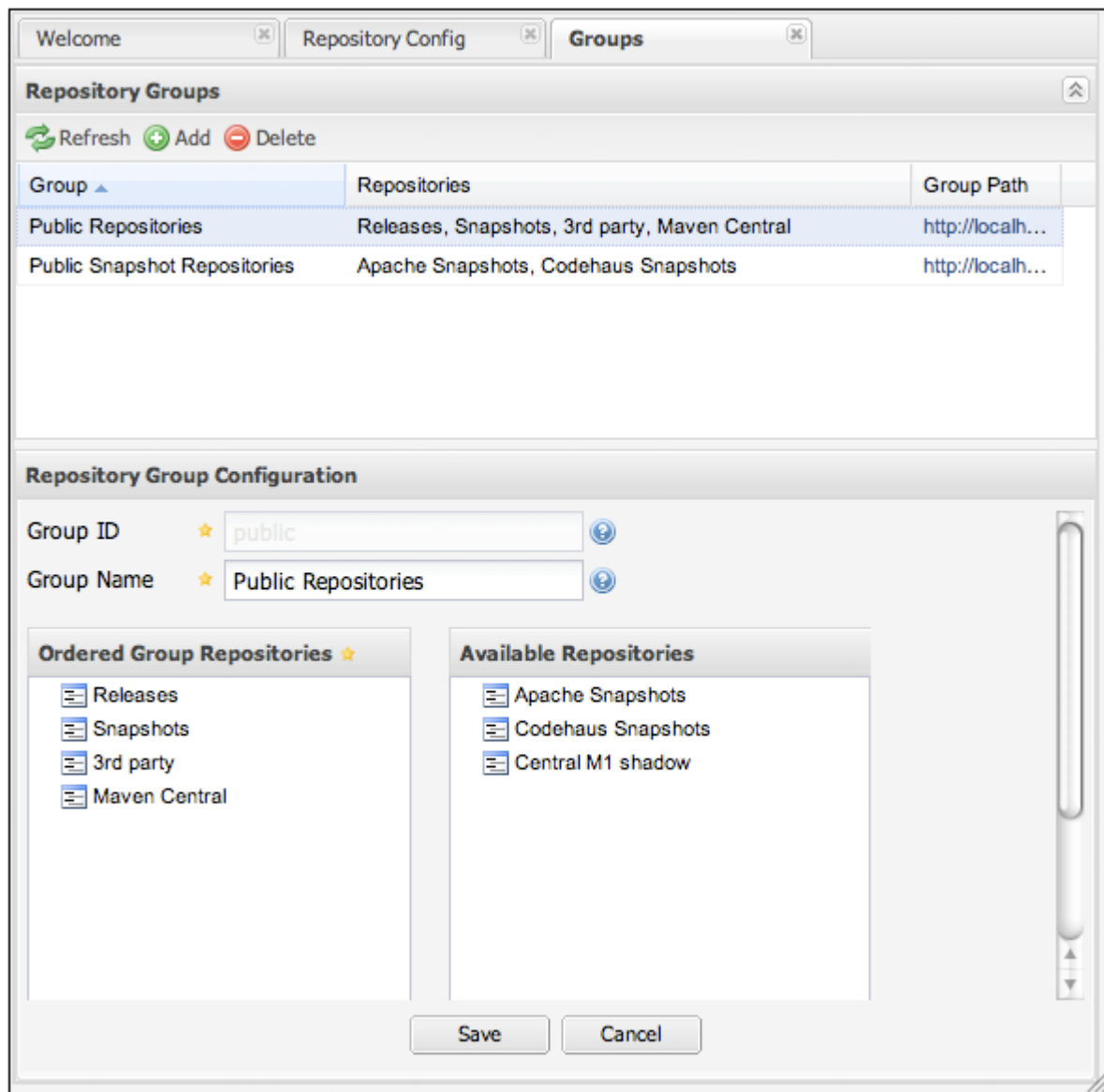


Figure 16.10. 添加新的仓库至一个 Nexus 组

Nexus 使用了一个十分有趣的，名为 [ExtJS](#) 的 Javascript 小工具类库。ExtJS 提供了许多有趣的 UI 小工具，能为用户提供丰富的交互体验。要添加这两个新的仓库至 public Nexus 组，在可用仓库列表中找到仓库，点击你想要添加的仓库然后拖拉进 Ordered Group Repositories。一旦仓库在 Ordered Group Repositories 列表中，你可以点击并拖拉列表中的仓库，以改变为匹配构件进行搜索的仓库的顺序。在 Google Caja 和 Google OAuth 项目仓库被添加到 public Nexus 组之后，你应该能够构建 Apache Shindig 并观察到 Maven 从各自的仓库下载 Caja 和 OAuth。

16.5. 配置 Nexus



本节展示的很多配置页面只对管理员可用。Nexus 允许管理员用户自定义仓库列表，创建仓库组，自定义服务器设置，以及创建 Maven 用来包含或排除某个仓

库构件的路线或者“规则”。

16.5.. 定制服务器配置



在一个实际的 Nexus 安装中，你可能会想要自定义管理员密码，而非使用“admin123”，你可能会想要复写 Nexus 用来存储仓库数据的默认目录。为此，以管理员用户登陆然后点击左边导航菜单 Administration 下面的 Server。服务器配置界面如 [Figure 16.11, “Nexus 服务器配置”](#) 显示。

The screenshot shows the 'Nexus' configuration window with the following settings:

- File Settings**
 - Working Directory: `/Users/tobrien/Applications/nexus/runtime/work/nexus`
 - Log Directory: `/Users/tobrien/Applications/nexus/runtime/apps/nexus/logs`
- SMTP Settings**
 - Hostname: `smtp-host`
 - Port: `25`
 - Username: `smtp-username`
 - Password: `.....`
 - SSL Enabled: ☐
 - TLS Enabled: ☐
 - System Email: `system@nexus.org`
- HTTP Request Settings**
 - User Agent: `Nexus/1.0`
 - Additional URL Parameters: (empty)
 - Request Timeout: `10` seconds
 - Request Retry Attempts: `3`

Buttons: Save, Cancel

Figure 16.11. Nexus 服务器配置

该页面能让你更改：

管理员密码

默认的管理员密码是 admin123。如果你填写这个字段并点击了 Save 按钮，你及更改了这个 Nexus 安装的管理员密码。

工作目录

在 File Settings 组下面，你可以自定义工作目录。如果你的 Nexus 安装将作为很大的仓库的镜像，而且你想要将你的工作目录放到另外一个硬盘分区，你可能会想要自定义工作目录。

日志目录

你可以改变 Nexus 寻找日志的位置。在一个 Unix 机器上，一个通常的实践是将日志文件放到 /var/log。如果你遵循这个实践，你可以使用适当的权限来创建一个 /var/log/nexus 目录。注意这个设置并不会更改 Nexus 记日志的目录，它仅仅是告诉 Nexus 去哪里寻找日志。要更改写日志的位置，你需要修改在你 Nexus 安装的 runtime/apps/nexus/conf 目录下 jul-logging.properties 的和 log4j.properties 文件。

User Agent

这是 Nexus 用来生成 HTTP 请求的标识符。如果 Nexus 需要用一个 HTTP 代理，而且这个代理只有当 User Agent 设置成某个特定值才能工作，你就要更改这个设置。

额外的 URL 参数

这是一列放在对远程仓库的 GET 请求后面的附加参数。你可以用它来添加对请求的定义信息。

请求超时

这是当 Nexus 和外部，远程的仓库交互时等待一个请求成功的时间。

请求重试次数

当遇到一个失败的 HTTP 请求时，Nexus 会重试的次数。

代理主机和代理端口

如果你的组织需要使用一个 HTTP 代理服务器，你可以在这里提供代理主机和代理端口。

代理认证

这一部分配置能让你提供代理认证信息，如用户名和密码，或者用来访问 HTTP 代理的密钥。

16.5.2. 管理仓库



要管理 Nexus 提供的仓库，以管理员用户登陆然后点击左边导航菜单 Administration 下面的 Repositories。Nexus 提供了三种不同的仓库。

代理仓库

一个代理仓库是对远程仓库的一个代理。默认情况下，Nexus 自带了如下配置的代理仓库：

Apache Snapshots

这个仓库包含了来自于 Apache 软件基金会的快照版本。
<http://people.apache.org/repo/m2-snapshot-repository>

Codehaus Snapshots

这个仓库包含了来自于 Codehaus 的快照版本。
<http://snapshots.repository.codehaus.org/>

Central Maven Repository

这是中央 Maven 仓库（发布版本）。 <http://repo1.maven.org/maven2/>

宿主仓库

一个宿主仓库是由 Nexus 托管的仓库。Maven 自带了如下配置的宿主仓库。

3rd Party

这个宿主仓库应该用来存储在公共 Maven 仓库中找不到的第三方依赖。这种依赖的样例有：你组织使用的，商业的，私有的类库如 Oracle JDBC 驱动。

Releases

这个宿主仓库是你组织公布内部发布版本的地方。

Snapshots

这个宿主仓库是你组织发布内部快照版本的地方。

虚拟仓库

一个虚拟仓库作为 Maven 1 的适配器存在。Nexus 自带了一个 central-m1 虚拟仓库。

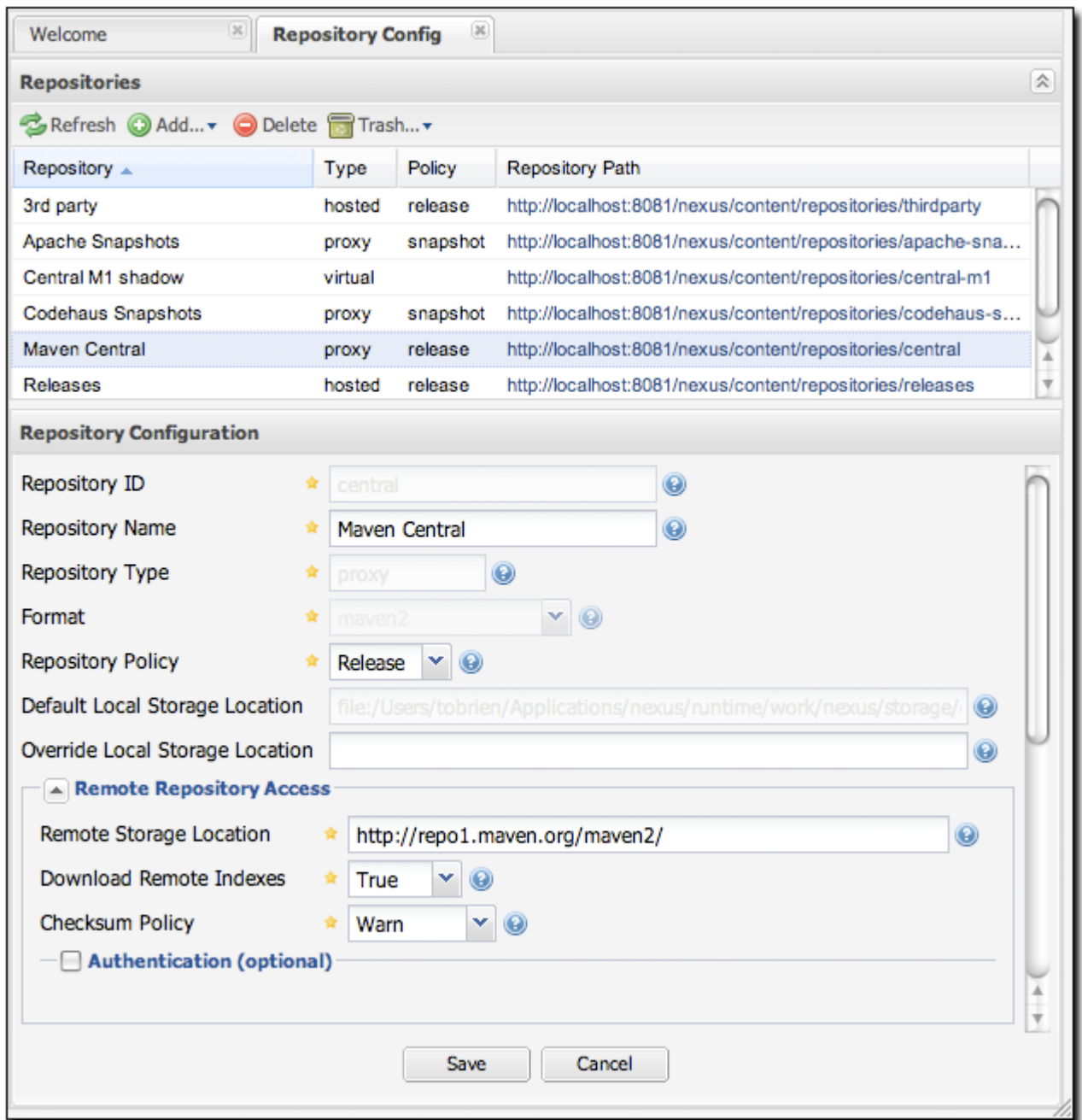


Figure 16.12. 代理仓库的配置页面

[Figure 16.12, “代理仓库的配置页面”](#) 展示了 Nexus 中代理仓库的配置页面。在这个页面中，你可以管理一个外部仓库的设置。本页面中，你可以配置：

仓库 ID

仓库 ID 是将会被用在 Nexus URL 中的标识符。例如，中央代理仓库有一个 ID 为 "central"，这就意味着 Maven 可以直接在 `http://localhost:8081/nexus/content/repositories/central` 访问这个仓库。在一个给定的 Nexus 安装中，仓库 ID 必须是唯一的。ID 是必需的。

仓库名称

仓库的显示名称。名称是必需的。

仓库类型

仓库类型（代理，宿主，或者虚拟）。你不能改变仓库的类型，在你创建一个仓库的时候它就被指定了。

仓库策略

如果一个代理仓库的策略是 **release**，那么它只会访问远程仓库的发布版本构件。如果一个代理仓库的策略是 **snapshot**，它只会下载远程仓库的快照版本构件。

默认存储位置

它不可编辑的，显示出来只是为了参考。这是仓库本地缓存内容的默认存储位置。

覆写存储位置

你可以选择为某个特定的仓库覆写存储位置。如果你关心存储空间，或者想要将某个特定仓库（如中央仓库）的内容放到一个不同的位置，你就可以覆写存储位置。

远程仓库访问

这一部分告诉 Nexus 去哪里寻找远程仓库，以及如何与这个被代理的仓库交互。

远程存储位置

这是远程 Maven 仓库的 URL。

下载远程索引（本图未显示）

这个字段控制下载远程索引。目前只有中央仓库在 <http://repo1.maven.org/maven2/.index> 有一个索引。如果开启它，Nexus 会下载这个索引，并使用它用来搜索，以及为任何要求索引的客户（如 m2eclipse）服务。新的代理仓库的默认值是开启的，但是 Nexus 自带的所有仓库的这个默认值是关闭的。要改变 Nexus 自带的代理仓库设置，更改此选项，保存至仓库，然后给仓库重新编制索引。在这之后，构件搜索会返回中央 Maven 仓库中可用的每一个构件。[Section 16.6, “维护仓库”](#) 详细描述了为仓库重新编制索引的过程。

校验和策略

为一个远程仓库设置校验和策略。这个选项默认设置成 Warn。该设置可能的值包括：

- Ignore - 完全忽略校验和
- Warn - 如果校验和不正确，在日志中打印一个警告
- StrictIfExists - 如果计算出来的校验和与仓库中的校验和不一致，那就拒绝缓存这个构件。只有校验和文件存在的时候才进行检查。
- Strict - 如果计算出来的校验和与仓库中的校验和不一致，或者如果构件没有校验和文件，就拒绝缓存这个构件。

认证

这一部分允许你为一个远程仓库设置用户名，密码，私钥，密钥口令，NT LAN HOST，以及 NT LAN Manager Domain。

访问设置

这一部分为一个仓库配置访问设置。

允许部署

如果允许部署设置成 `true`，Nexus 会允许 Maven 部署构件至这个仓库。允许部署只有对于宿主仓库有意义。

允许文件浏览

如果设置成 `true`，用户可以通过 web 浏览器来浏览这个仓库的内容。

包含在搜索范围中

如果设置成 `true`，当你在 Nexus 中执行构件搜索的时候，该仓库会被搜索。如果设置成 `false`，在搜索时该仓库的内容会被排除。

过期失效设置

Nexus 为构件和元数据维护一份本地的缓存，你可以为代理仓库配置过期失效参数。过期失效设置有：

未找到缓存 TTL

如果 Nexus 找不到一个构件，它会在一个给定的时间内缓存这个结果。换句话说，如果 Nexus 不能在远程仓库中找到一个构件，它不会重复的尝试去解析这个构件，除非超过了这个未找到缓存 TTL 时间。默认值是 1140 分钟（或者 24 小时）。

构件最大年龄

在 Nexus 从远程仓库获取一个新版本的构件前，告诉它构件的最大年龄是多少。带有 `release` 策略的仓库的默认设置是 -1，带有 `snapshot` 策略的仓库的默认值是 1140。

元数据最大年龄

Nexus 从远程仓库获取元数据。只有在超过了元数据最大年龄之后，它才会去获取元数据的更新。该设置的默认值是 1140 分钟（或者 24 小时）。

HTTP 请求设置

这一部分能让你更改对于远程仓库的 HTTP 请求的属性。该部分中你可以配置请求的 User Agent，为请求添加参数，设置超时和重试行为。这一部分涉及的是由 Nexus 对远程被代理 Maven 仓库的 HTTP 请求。

HTTP 代理设置

该部分能让你为从 Nexus 到远程仓库的请求配置 HTTP 代理。你可以配置一个代理主机，端口和认证设置，以告诉 Nexus 为所有对远程仓库的请求使用 HTTP 代理。

16.5.3. 管理组



组是 Nexus 一个强大的特性，它允许你在一个单独的 URL 中组合多个仓库。Nexus 自带了两个组：`public` 和 `public-snapshots`。`public` 组中组合了三个宿主仓库：3rd Party, Releases, 和 Snapshots，还有中央 Maven 仓库。而 `public-snapshots` 组中组合了 Apache Snapshots 和 Codehaus Snapshots 仓库。在 [Section 16.4, “配置](#)

[Maven 使用 Nexus](#)”中我们通过 settings.xml 配置 Maven 从 Nexus 管理的 public 组中寻找构件。[Figure 16.13, “Nexus 中的组配置页面”](#)显示了 Nexus 中的组配置页面，在该图中你可以看到 public 组的内容。

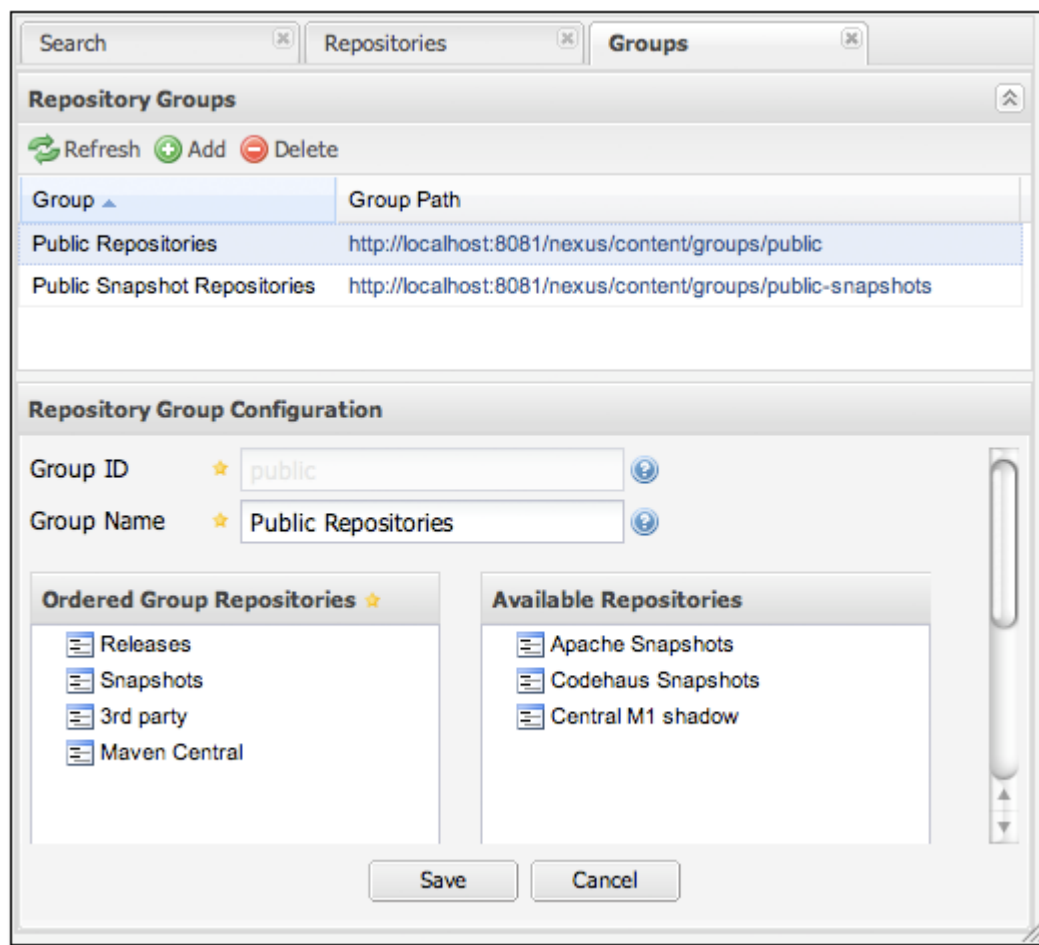


Figure 16.13. Nexus 中的组配置页面

注意在图中有序组仓库中列出的仓库的顺序是很重要的。当 Nexus 在一组仓库中搜索一个构件的时候，它会返回第一个匹配的结果。要更改列表中仓库的顺序，在有序组仓库选择列表中点击并拖拉仓库即可。

16.5.4. 管理路由



Nexus 路由就像是你可以应用到 Nexus 组上的过滤器，当 Nexus 尝试在一个 Nexus 组中寻找构件的时候，路由允许你在一个特定的构件搜索中包含或者排除一些仓库。有很多不同的场景中你可能需要配置路由，最平常的是当你想要确信你正从一个特定组的特定的仓库中获取构件。特别是当你想要确信你正从宿主 Releases 及 Snapshots 仓库中获取你自己组织的构件的时候，路由功能很有用。当你正尝试从一个 Nexus 组中解析一个构件的时候，Nexus 路由十分适用；当 Nexus 从一组仓库中解析一个构件的时候，使用路由允许你更改 Nexus 将查阅的仓库。

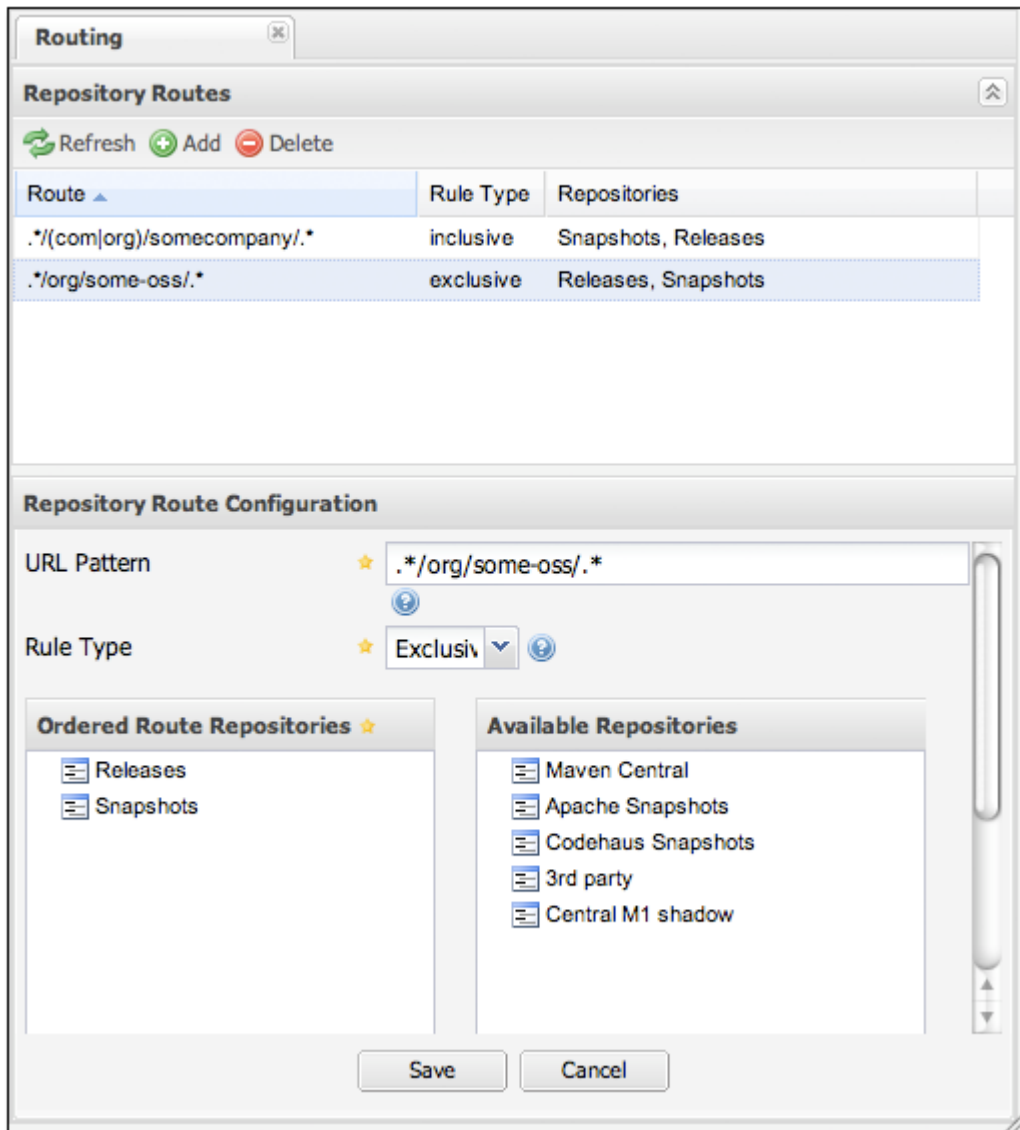


Figure 16.14. Nexus 中的路由配置页面

[Figure 16.14, “Nexus 中的路由配置页面”](#)显示了路由配置页面。在路由上点击会看到一个页面，能让你配置路由的属性。路由的可用配置选项是：

URL 模式

这是 Nexus 用来匹配请求的模式。如果这个模式与请求表达式匹配，Nexus 就会在特定的构件查询中包含或者排除所列出的仓库。在 [Figure 16.14, “Nexus 中的路由配置页面”](#)中的两个模式是：

`.*/(com|org)/somecompany/.*`

这个模式会匹配所有包含`"/com/somecompany/"`或者`"/org/somecompany"`的路径。圆括弧中的表达式匹配 `com` 或者 `org`，`"*"`匹配一个或多个字符。你可以使用一个像这样的路由来匹配你自己组织的构件，并且将这样的请求对应到宿主的 Nexus Releases 及 Snapshots 仓库。

`.* /org/some-oss/.*`

这个模式用作一个排除路由。它匹配所有包含"/org/some-oss/"的路径。这个特殊的排除路由为所有与该路径匹配的构件排除宿主的 Releases 和 Snapshots 仓库。当 Nexus 尝试解析与该路由匹配的构件时，它会排除 Releases 和 Snapshots 仓库。

路由类型

路由类型可以是“包含”或者“排除”。一个包含路由类型定义了一组仓库，当 URL 模式匹配的时候这些仓库将被搜索。同样的情况下，一个排除路由定义的仓库则不会被搜索。

有序路由仓库

这是一个 Nexus 用来搜索以寻找某个特殊构件的一个有序仓库列表。Nexus 从上至下搜索；当它找某个构件的时候，会返回第一个匹配的结果。当它寻找元数据的时候，同一组中所有的仓库会被检查，并且最后返回一个归并的结果。归并的时候，前面的仓库拥有较高的优先权。当一个项目正在寻找 LATEST 或者 RELEASE 版本的时候，这可能会影响结果。在一个 Nexus 组中，你应该在快照版仓库前定义发布版仓库，否则 LATEST 可能会错误的解析成一个快照版本。

在该图中你能看到两个 Nexus 默认带有的假路由。第一个路由是一个包含路由，它是一个自定义路由的例子，一个组织可能用来确信内部生成的构件是从 Releases 和 Snapshots 仓库解析的。如果你组织的 groupId 都由 com.somecompany 开头，并且你们将内部生成的构件部署到了 Releases 和 Snapshots 仓库，该路由让 Nexus 不浪费时间去从公共的 Maven 仓库，如中央 Maven 仓库或者 Apache Snapshots 仓库，去解析构件。

第二个假路由是一个排除路由。当请求路径包含"/org/some-oss"的时候，路由会排除 Releases 和 Snapshots 仓库。如果我们使用"apache"或者"codehaus"替换"some-oss"，这个例子就可能更有意义了。如果这个模式是"/org/apache"，该规则告诉 Nexus，当试图解析这些依赖的时候，排除内部的 Releases 和 Snapshots 仓库。换句话说，不要浪费时间从你组织的内部仓库中去寻找 Apache 依赖。

如果两个路由有冲突在怎么办？Nexus 会在它处理排除路由之前处理包含路由。记住 Nexus 路由只会在当搜索一个组的时候影响 Nexus 解析构件。当 Nexus 开始从一个 Nexus 组中解析一个构件，它开始于组中的一个仓库列表。如果有匹配的包含路由，Nexus 就会使用组中仓库和包含路由中仓库的交集。在 Nexus 组中定义的顺序不会受包含路由的影响。Nexus 之后就会在这个新的组中应用排除路由。最后在这个结果列表中搜索匹配构件。

总结来说，路由还有很多 Nexus 的设计者们未预期到的创新可能性，但是，如果你开始信赖冲突或者重叠路由，我们还是建议你小心。保守使用路由，使用教程中的 URL 模式，随着 Nexus 的发展，将会有更多的特性允许更细类度的规则以让你阻止特定构件和特定构件版本的请求。记住路由只能用在 Nexus 组中，当从某个特定仓库中请求一个构件的时候，路由不会被用到。

16.5.5. 网络配置



默认情况下，Nexus 监听端口 8081。你可以更改这个端口，通过更改

`${NEXUS_HOME}/conf/plexus.properties` 的值，如 [Example 16.3, “\\${NEXUS_HOME}/conf/plexus.properties 的内容”](#)。为此，停止 Nexus，更改文件中 `applicationPort` 的值，然后重启 Nexus。在这之后，你应该能够在 `${NEXUS_HOME}/logs/wrapper.log` 中看到一条日志记录，告诉你 Nexus 在监听更改过的端口。

Example 16.3. `${NEXUS_HOME}/conf/plexus.properties` 的内容

```
applicationPort=8081
runtime=${basedir}/runtime
apps=${runtime}/apps
work=${runtime}/work
webapp=${runtime}/apps/nexus/webapp
nexus.configuration=${runtime}/apps/nexus/conf/nexus.xml
```

16.6. 维护仓库



在你设置了一系列仓库并且将这些仓库组织成组之后，用户就能够通过点击左边菜单 **Views** 部分的 **Repositories** 链接，在 Neuxs UI 上看到一个仓库的列表。Nexus 会显示一个仓库列表。这个列表会显示远程仓库的状态；如果要测试一下，可以编辑你的一个仓库，让它拥有一个垃圾远程存储位置 URL，你会在仓库管理页面上注意到该仓库的状态变化了。点击一个仓库会显示一个树状视图，以让用户能够浏览仓库的内容。

在一个仓库上右击，会看到一系列能用到仓库上的动作。每个仓库上可用的动作有：

查看

载入一个仓库的树状视图。该视图能让你深化到特定的目录以查看仓库的内容。

清空缓存

为仓库清空缓存。它促使 Maven 去远程仓库检查更新或者快照版本。它也会重置未找到缓存。

重新编制索引

促使 Maven 为一个仓库重新编制索引。Nexus 会重新创建它用来搜索构件请求的索引。如果仓库已被配置了下载远程索引，这一选项促使 Nexus 从远程仓库下载远程索引。注意如果你开启了远程索引下载，可能需要花一些时间从远程仓库下载索引。当构件搜索结果开始显示没有缓存或请求过的构件，你会知道远程仓库已经更新了。

阻塞代理 / 允许代理

这可以封锁对远程仓库的请求。如果代理被阻塞了，Nexus 就不会连接到远程仓库去请求更新。要重新开启远程访问，在仓库上右击然后选择允许代理。当你想要控制代理仓库提供的内容的时候，该选项十分有用。如果你想维护从远程仓库下载内容的严格控制，你可以先基于 Nexus 运行你组

织的构建，然后阻塞所有的代理仓库。

服务失效 / 服务生效

该选项允许你让一个仓库失效，使之不可用。Nexus 就会拒绝所有对失效仓库的服务。在你将一个仓库置为不可用之后，你可以通过在一个仓库上右击，选择“服务生效”，来回到可用状态。

16.7. 部署构件至 Nexus



不同的组织有不同的理由将构件部署至内部仓库。在有数百（或数千）开发人员的大型组织内，一个内部 Maven 仓库可以是不同部门之间共享发布版和开发快照版本的有效手段。大部分使用 Maven 的组织最终都会开始将发布版本和构件部署到一个共享的内部仓库。使用 Nexus，可以很容易的部署构件至一个宿主仓库。

要部署构件至 Nexus，在 `distributionManagement` 中提供仓库 URL，然后运行 **mvn deploy**。Maven 会通过一个简单的 HTTP PUT 将项目 POM 和构件推入至你的 Nexus 安装。最初版本的 Nexus 没有为宿主仓库提供任何的安全措施。如果你为宿主仓库开启了部署功能，任何人可以连接并部署构件至这个仓库。如果你的 Nexus 安装能够从公共 Internet 访问，你绝对会想要将这些仓库的部署功能关闭，或者将你的 Nexus 安装放到一个如 Apache HTTPD 的 web 服务器背后。

你项目的 POM 不再需要额外的 wagon 扩展。Nexus 可以和 Maven 内置的 `wagon-http-lightweight` 一起工作。

16.7.1. 部署发布版



要部署一个发布版构件至 Nexus，你需要配置你项目 POM 中 `distributionManagement` 部分的 `repository`。[Example 16.4, “为部署配置发布版本仓库”](#)显示了一个发布版部署仓库的样例，这个发布版本仓库的地址是 <http://localhost:8081/nexus/content/repositories/releases>。

Example 16.4. 为部署配置发布版本仓库

```
<project>
...
<distributionManagement>
...
<repository>
  <id>releases</id>
  <name>Internal Releases</name>
  <url>http://localhost:8081/nexus/content/repositories/releases</url>
</repository>
...
</distributionManagement>
```

```
...
</project>
```

你可以使用你 Nexus 安装的主机和端口来替换 localhost:8081。你的项目有了这个配置之后，你就可以通过执行 **mvn deploy** 命令部署构件。

```
$ mvn deploy
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Sample Project
[INFO]
-----
[INFO] Building Sample Project
[INFO]   task-segment: [deploy]
[INFO]
-----
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing ~/svn/sample/pom.xml to ~/.m2/repository/sample/sample\
                                     /1.0/sample-1.0.pom
[INFO] [deploy:deploy]
altDeploymentRepository = null
[INFO] Retrieving previous build number from snapshots
Uploading: http://localhost:8081/nexus/content/repositories/releases/\
          sample/sample/1.0/sample-1.0.pom
24K uploaded
```

注意 Nexus 可以支持多个宿主仓库；你不需要坚持在默认的 releases 和 snapshots 仓库上。你可以为不同的部门创建不同的宿主仓库，然后将多个仓库组合成一个单独的 Nexus 组。

16.7.2. 部署快照版



要部署快照版本构件至 Nexus，你需要配置你项目 POM 的 distributionManagement 部分的 snapshotRepository。[Example 16.5, “为部署配置快照版本仓库”](#)显示了快照版本部署仓库的样例，该 snapshots 仓库配置的地址是 <http://localhost:8081/nexus/content/repositories/snapshots>。

Example 16.5. 为部署配置快照版本仓库

```
<project>
...
  <distributionManagement>
    ...
    <snapshotRepository>
```

```

    <id>Snapshots</id>
    <name>Internal Snapshots</name>
    <url>http://localhost:8081/nexus/content/repositories/snapshots</url>
  </snapshotRepository>
  ...
</distributionManagement>
...
</project>

```

你可以使用你 Nexus 安装的主机和端口来替换 localhost:8081。你的项目有了这个配置之后，你就可以通过执行 **mvn deploy** 命令部署构件。如果你项目的版本是快照版本(如 1.0-SNAPSHOT)Maven 就会将其部署至 snapshotRepository:

```

$ mvn deploy
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Sample Project
[INFO]
-----
[INFO] Building Sample Project
[INFO]   task-segment: [deploy]
[INFO]
-----
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO] Installing ~/svn/sample/pom.xml to ~/.m2/repository/sample/sample\
/1.0-SNAPSHOT/sample-1.0-20080402.125302.pom
[INFO] [deploy:deploy]
altDeploymentRepository = null
[INFO] Retrieving previous build number from snapshots
Uploading: http://localhost:8081/nexus/content/repositories/releases/\
sample/sample/1.0-SNAPSHOT/sample-1.0-20080402.125302.pom
24K uploaded

```

16.7.3. 部署第三方构件



你的 Maven 项目可以依赖于一个构件，这个构件不能从中央 Maven 仓库或其它公开 Maven 仓库找到。有很多原因可能导致这种情形发生：这个构件可能是私有数据库的 JDBC 驱动如 Oracle，或者你依赖于另一个 JAR，它既不开源也无法免费获得。在这样的情况下，你就需要手动拿来这些构件然后发布到你自己的仓库中。Nexus 提供宿主的"third-party"仓库，就是为了这个目的。

为了阐明发布一个构件至第三方仓库的过程，我们使用一个真实的构件：Oracle

JDBC 驱动。Oracle 发布一个广泛使用的商业数据库产品，该产品带有一个中央 Maven 仓库没有的 JDBC 驱动。虽然中央 Maven 仓库在 <http://repol.maven.org/maven2/com/oracle/ojdbc14/10.2.0.3.0/> 维护了一些 Oracle JDBC 驱动的 POM 信息，但这些只是指向 Oracle 站点的 POM。如果你将下列的依赖添加到你的项目。

Example 16.6. Oracle JDBC JAR 依赖

```
<project>
...
<dependencies>
...
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ojdbc14</artifactId>
  <version>10.2.0.3.0</version>
</dependency>
...
</dependencies>
...
</project>
```

用这个依赖运行一个 Maven 构建，将会产生如下的输出：

```
$ mvn install
...
[INFO]
-----
[ERROR] BUILD ERROR
[INFO]
-----
[INFO] Failed to resolve artifact.

Missing:
-----
1) com.oracle:ojdbc14:jar:10.2.0.3.0

Try downloading the file manually from:

http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html

Then, install it using the command:
    mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
        -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=/path/to/file
```

Alternatively, if you host your own repository you can deploy the file there:

\

```
mvn deploy:deploy-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
    -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=/path/to/file \
    -Durl=[url] -DrepositoryId=[id]
```

Path to dependency:

- 1) sample:sample:jar:1.0-SNAPSHOT
- 2) com.oracle:ojdbc14:jar:10.2.0.3.0

1 required artifact is missing.

Maven 构建失败了因为它不能在 Maven 仓库中找到 Oracle JDBC 驱动。要补救这种情况，你将需要发布 Oracle JDBC 构件至你的 Nexus third-party 仓库。为此，从 http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html 下载 Oracle JDBC 驱动，然后保存至文件 ojdbc.jar。使用以下命令发布该文件至 Nexus:

```
$ mvn deploy:deploy-file -DgroupId=com.oracle -DartifactId=ojdbc14 \
> -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=ojdbc.jar \
> -Durl=http://localhost:8081/nexus/content/repositories/thirdparty \
> -DrepositoryId=thirdparty
...
[INFO] [deploy:deploy-file]
Uploading: http://localhost:8081/nexus/content/repositories/thirdparty/\
           com/oracle/ojdbc14/10.2.0.3.0/ojdbc14-10.2.0.3.0.jar
330K uploaded
[INFO] Retrieving previous metadata from thirdparty
[INFO] Uploading repository metadata for: 'artifact com.oracle:ojdbc14'
[INFO] Retrieving previous metadata from thirdparty
[INFO] Uploading project information for ojdbc14 10.2.0.3.0
```

在你运行 **mvn deploy:deploy-file** 之后，该构件会被发布至 Nexus 的 third-party 仓库。

目前原网站更新的中文章节都已经收录

- 缺失章节:12,13,15,17,18 还有附录,等待补全中,本文档仅供个人参考学习,请勿非法传播,版权归原网站所有:

- Maven: The Definitive Guide by [Sonatype, Inc.](#) is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States License](#).

Based on a work at www.sonatype.com.