Maven教程

极客学院出版

前言

Apache Maven 是一套软件工程管理和整合工具。基于工程对象模型（POM）的概念，通过一个中央信息管理模块，Maven 能够管理项目的构建、报告和文档。

本教程将教你如何在使用 Java 开发的工程中，或者任何其他编程语言中使用 Maven。

适合人群

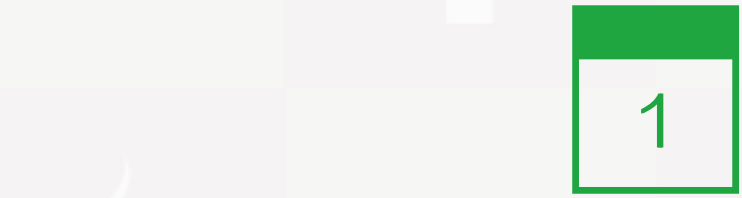
本教程主要针对初学者，帮助他们学习 Maven 工具的基本功能。完成本教程的学习后你的 Apache Maven 的专业知识将达到中等水平，随后你可以学习更高级的知识了。

学习前提

我们假定你将使用 Maven 来管理企业级的 Java 项目开发。所以，如果你掌握软件开发、Java SE、Java EE 开发框架和开发流程，对学习本教程会更有帮助。

更新日期	更新内容
2015-05-17	第一版发布

目录



Maven – 概述

Maven 是什么?

Maven 是一个项目管理和整合工具。Maven 为开发者提供了一套完整的构建生命周期框架。开发团队几乎不用花多少时间就能够自动完成工程的基础构建配置，因为 Maven 使用了一个标准的目录结构和一个默认的构建生命周期。

在有多个开发团队环境的情况下，Maven 能够在很短的时间内使得每项工作都按照标准进行。因为大部分的工程配置操作都非常简单并且可复用，在创建报告、检查、构建和测试自动配置时，Maven 可以让开发者的工作变得更简单。

Maven 能够帮助开发者完成以下工作:

- 构建
- 文档生成
- 报告
- 依赖
- SCMs
- 发布
- 分发
- 邮件列表

总的来说，Maven 简化了工程的构建过程，并对其标准化。它无缝衔接了编译、发布、文档生成、团队合作和其他任务。Maven 提高了重用性，负责了大部分构建相关的任务。

Maven 的历史

Maven 最初是在 Jakarta Turbine 项目中为了简化构建过程而设计的。项目中有几个子工程，每个工程包含稍有不同的 ANT 文件。JAR 文件使用 CVS 管理。

Apache 小组随后开发了 Maven，能够同时构建多个工程、发布工程信息、部署工程、在几个工程中共享 JAR 文件，并且协助团队合作。

Maven 的目标

Maven 的主要目的是为开发者提供

- 一个可复用、可维护、更易理解的工程综合模型
- 与这个模型交互的插件或者工具

Maven 工程结构和内容被定义在一个 xml 文件中 – pom.xml，是 Project Object Model (POM) 的简称，此文件是整个 Maven 系统的基础组件。详细内容请参考 Maven POM () 部分。

约定优于配置

Maven 使用约定而不是配置，意味着开发者不需要再自己创建构建过程。

开发者不需要再关心每一个配置细节。Maven 为工程提供了合理的默认行为。当创建 Maven 工程时，Maven 会创建默认的工程结构。开发者只需要合理的放置文件，而在 pom.xml 中不再需要定义任何配置。

举例说明，下面的表格展示了工程源码文件、资源文件的默认配置，和其他一些配置。假定 `${basedir}` 表示工程目录：

配置项	默认值
source code	${basedir}/src/main/java
resources	${basedir}/src/main/resources
Tests	${basedir}/src/test
Complied byte code	${basedir}/target
distributable JAR	${basedir}/target/classes

为了构建工程，Maven 为开发者提供了选项来配置生命周期目标和工程依赖（依赖于 Maven 的插件扩展功能和默认的约定）。大部分的工程管理和构建相关的任务是由 Maven 插件完成的。

开发人员不需要了解每个插件是如何工作的，就能够构建任何给定的 Maven 工程。详细内容请参考 Maven 插件部分。

2

Maven – 环境配置

Maven 是一个基于 Java 的工具，所以要做的第一件事情就是安装 JDK。

系统要求

项目	要求
JDK	Maven 3.3 要求 JDK 1.7 或以上 Maven 3.2 要求 JDK 1.6 或以上 Maven 3.0/3.1 要求 JDK 1.5 或以上
内存	没有最低要求
磁盘	Maven 自身安装需要大约 10 MB 空间。除此之外，额外的磁盘空间将用于你的本地 Maven 仓库。你本地仓库的大小取决于使用情况，但预期至少 500 MB
操作系统	没有最低要求

步骤 1：检查 Java 安装

现在打开控制台，执行下面的 `java` 命令。

操作系统	任务	命令
Windows	打开命令控制台	`c:\> java -version`
Linux	打开命令终端	`$ java -version`
Mac	打开终端	`machine:~ joseph$ java -version`

我们来验证一下所有平台上的输出：

操作系统	输出
Windows	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Linux	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
Mac	java version "1.6.0_21" Java(TM) SE Runtime Environment (build 1.6.0_21-b07) Java HotSpot(TM)64-Bit Server VM (build 17.0-b17, mixed mode, sharing)

如果你没有安装 Java，从以下网址安装 Java 软件开发套件（SDK）：http://www.oracle.com/technetwork/java/javase/downloads/index.html。我们假定你安装的 Java 版本为1.6.0_21。

步骤 2：设置 Java 环境

设置 JAVA_HOME 环境变量，并指向你机器上的 Java 安装目录。例如：

操作系统	输出
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	`export JAVA_HOME=/usr/local/java-current`
Mac	`export JAVA_HOME=/Library/Java/Home`

将 Java 编译器地址添加到系统路径中。

操作系统	输出
Windows	将字符串“;C:\Program Files\Java\jdk1.6.0_21\bin”添加到系统变量“Path”的末尾
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

使用上面提到的 java -version 命令验证 Java 安装。

步骤 3：下载 Maven 文件

从以下网址下载 Maven 3.2.5：http://maven.apache.org/download.html

步骤 4：解压 Maven 文件

解压文件到你想要的位置来安装 Maven 3.2.5，你会得到 apache-maven-3.2.5 子目录。

操作系统	位置 (根据你的安装位置而定)
Windows	`C:\Program Files\Apache Software Foundation\apache-maven-3.2.5`
Linux	`/usr/local/apache-maven`
Mac	`/usr/local/apache-maven`

步骤 5：设置 Maven 环境变量

添加 M2_HOME、M2、MAVEN_OPTS 到环境变量中。

操作系统	输出
Windows	使用系统属性设置环境变量。 M2_HOME=C:\Program Files\Apache Software Foundation\apache-maven-3.2.5 M2=%M2_HOME%\bin MAVEN_OPTS=-Xms256m -Xmx512m
Linux	打开命令终端设置环境变量。 export M2_HOME=/usr/local/apache-maven/apache-maven-3.2.5 export M2=$M2_HOME/bin export MAVEN_OPTS=-Xms256m -Xmx512m
Mac	打开命令终端设置环境变量。 export M2_HOME=/usr/local/apache-maven/apache-maven-3.2.5 export M2=$M2_HOME/bin export MAVEN_OPTS=-Xms256m -Xmx512m

步骤 6：添加 Maven bin 目录到系统路径中

现在添加 M2 变量到系统“Path”变量中

操作系统	输出
Windows	添加字符串 “;%M2%” 到系统“Path”变量末尾
Linux	export PATH=$M2:$PATH
Mac	export PATH=$M2:$PATH

步骤 7：验证 Maven 安装

现在打开控制台，执行以下 mvn 命令。

操作系统	输出	命令
Windows	打开命令控制台	`c:\> mvn --version`
Linux	打开命令终端	`$ mvn --version`
Mac	打开终端	`machine:~ joseph$ mvn --version`

最后，验证以上命令的输出，应该是像下面这样：

操作系统	输出
Windows	Apache Maven 3.2.5 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre
Linux	Apache Maven 3.2.5 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre

操作系统	输出
Mac	Apache Maven 3.2.5 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.6.0_21 Java home: C:\Program Files\Java\jdk1.6.0_21\jre

恭喜！你完成了所有的设置，开始使用 Apache Maven 吧。

3

Maven – POM

POM 代表工程对象模型。它是使用 Maven 工作时的基本组建，是一个 xml 文件。它被放在工程根目录下，文件命名为 pom.xml。

POM 包含了关于工程和各种配置细节的信息，Maven 使用这些信息构建工程。

POM 也包含了目标和插件。当执行一个任务或者目标时，Maven 会查找当前目录下的 POM，从其中读取所需要的配置信息，然后执行目标。能够在 POM 中设置的一些配置如下：

- project dependencies
- plugins
- goals
- build profiles
- project version
- developers
- mailing list

在创建 POM 之前，我们首先确定工程组（groupId），及其名称（artifactId）和版本，在仓库中这些属性是工程的唯一标识。

POM 举例

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.companyname.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>

</project>
```

需要说明的是每个工程应该只有一个 POM 文件。

- 所有的 POM 文件需要 project 元素和三个必须的字段：groupId, artifactId,version。
- 在仓库中的工程标识为 groupId:artifactId:version
- POM.xml 的根元素是 project，它有三个主要的子节点：

节点	描述
groupId	这是工程组的标识。它在一个组织或者项目中通常是唯一的。例如，一个银行组织 com.company.bank 拥有所有的和银行相关的项目。
artifactId	这是工程的标识。它通常是工程的名称。例如，消费者银行。groupId 和 artifactId 一起定义了 artifact 在仓库中的位置。
version	这是工程的版本号。在 artifact 的仓库中，它用来区分不同的版本。例如: com.company.bank:consumer-banking:1.0 com.company.bank:consumer-banking:1.1.

Super POM

所有的 POM 都继承自一个父 POM（无论是否显式定义了这个父 POM）。父 POM 也被称作 ** Super POM**，它包含了一些可以被继承的默认设置。

Maven 使用 effective pom（Super pom 加上工程自己的配置）来执行相关的目标，它帮助开发者在 pom.xml 中做尽可能少的配置，当然这些配置可以被方便的重写。

查看 Super POM 默认配置的一个简单方法是执行以下命令：**mvn help:effective-pom**

在你的电脑上的任意目录下创建一个 pom.xml 文件，使用上面提到的示例 pom 中的内容。

在下面的例子中，我们在 `C:\MVN\project` 目录中创建了一个 pom.xml 文件。

现在打开命令控制台，到 pom.xml 所在的目录下执行以下 **mvn** 命令。

```
C:\MVN\project>mvn help:effective-pom
```

Maven 将会开始处理并显示 effective-pom。

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.project-group:project-name:jar:1.0
[INFO]    task-segment: [help:effective-pom] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:effective-pom {execution: default-cli}]
[INFO]

.....

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
```

```
[INFO] Finished at: Thu Jul 05 11:41:51 IST 2012
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
```

Effective POM 的结果就像在控制台中显示的一样，经过继承、插值之后，使配置生效。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ============================================================== -->
<!--                                                                -->
<!-- Generated by Maven Help Plugin on 2012-07-05T11:41:51          -->
<!-- See: http://maven.apache.org/plugins/maven-help-plugin/        -->
<!--                                                                -->
<!-- ============================================================== -->

<!-- ============================================================== -->
<!--                                                                -->
<!-- Effective POM for project                                      -->
<!-- 'com.companyname.project-group:project-name:jar:1.0'           -->
<!--                                                                -->
<!-- ============================================================== -->

<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 h
ttp://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.project-group</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <build>
    <sourceDirectory>C:\MVN\project\src\main\java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>C:\MVN\project\src\test\java</testSourceDirectory>
    <outputDirectory>C:\MVN\project\target\classes</outputDirectory>
    <testOutputDirectory>C:\MVN\project\target\test-classes</testOutputDirectory>
    <resources>
      <resource>
        <mergeId>resource-0</mergeId>
        <directory>C:\MVN\project\src\main\resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <mergeId>resource-1</mergeId>
        <directory>C:\MVN\project\src\test\resources</directory>
      </testResource>
    </testResources>
```

```
<directory>C:\MVN\project\target</directory>
<finalName>project-1.0</finalName>
<pluginManagement>
 <plugins>
  <plugin>
   <artifactId>maven-antrun-plugin</artifactId>
   <version>1.3</version>
  </plugin>
  <plugin>
   <artifactId>maven-assembly-plugin</artifactId>
   <version>2.2-beta-2</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>2.4</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>2.5</version>
      </plugin>
      <plugin>
        <artifactId>maven-plugin-plugin</artifactId>
        <version>2.4.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-rar-plugin</artifactId>
        <version>2.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.0-beta-8</version>
      </plugin>
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>2.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <version>2.0-beta-7</version>
      </plugin>
      <plugin>
        <artifactId>maven-source-plugin</artifactId>
        <version>2.0.4</version>
      </plugin>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.4.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1-alpha-2</version>
      </plugin>
    </plugins>
  </pluginManagement>
  <plugins>
    <plugin>
      <artifactId>maven-help-plugin</artifactId>
      <version>2.1.1</version>
    </plugin>
  </plugins>
</build>
<repositories>
```

```
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Maven Repository Switchboard</name>
    <url>http://repo1.maven.org/maven2</url>
  </repository>
 </repositories>
 <pluginRepositories>
  <pluginRepository>
    <releases>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Maven Plugin Repository</name>
    <url>http://repo1.maven.org/maven2</url>
  </pluginRepository>
 </pluginRepositories>
 <reporting>
  <outputDirectory>C:\MVN\project\target/site</outputDirectory>
 </reporting>
</project>
```

在上面的 pom.xml 中，你可以看到 Maven 在执行目标时需要用到的默认工程源码目录结构、输出目录、需要的插件、仓库和报表目录。

Maven 的 pom.xml 文件也不需要手工编写。

Maven 提供了大量的原型插件来创建工程，包括工程结构和 pom.xml。

详细内容请参考 Maven – 插件 () 和 Maven – 创建工程 () 部分的内容。

4

Maven – 构建生命周期

什么是构建生命周期

构建生命周期是一组阶段的序列（sequence of phases），每个阶段定义了目标被执行的顺序。这里的阶段是生命周期的一部分。

举例说明，一个典型的 Maven 构建生命周期是由以下几个阶段的序列组成的：

阶段	处理	描述
prepare-resources	资源拷贝	本阶段可以自定义需要拷贝的资源
compile	编译	本阶段完成源代码编译
package	打包	本阶段根据 pom.xml 中描述的打包配置创建 JAR / WAR 包
install	安装	本阶段在本地 / 远程仓库中安装工程包

当需要在某个特定阶段之前或之后执行目标时，可以使用 pre 和 post 来定义这个**目标**。

当 Maven 开始构建工程，会按照所定义的阶段序列的顺序执行每个阶段注册的目标。Maven 有以下三个标准的生命周期：

- clean
- default(or build)
- site

目标表示一个特定的、对构建和管理工程有帮助的任务。它可能绑定了 0 个或多个构建阶段。没有绑定任何构建阶段的目标可以在构建生命周期之外被直接调用执行。

执行的顺序依赖于目标和构建阶段被调用的顺序。例如，考虑下面的命令。clean 和 package 参数是构建阶段，而 dependency:copy-dependencies 是一个目标。

```
mvn clean dependency:copy-dependencies package
```

这里的 clean 阶段将会被首先执行，然后 dependency:copy-dependencies 目标会被执行，最终 package 阶段被执行。

Clean 生命周期

当我们执行 *mvn post-clean* 命令时，Maven 调用 clean 生命周期，它包含以下阶段。

- pre-clean
- clean

- post-clean

Maven 的 clean 目标（clean:clean）绑定到了 clean 生命周期的 clean 阶段。它的 clean:clean 目标通过删除构建目录删除了构建输出。所以当 mvn clean 命令执行时，Maven 删除了构建目录。

我们可以通过在上面的 clean 生命周期的任何阶段定义目标来修改这部分的操作行为。

在下面的例子中，我们将 maven-antrun-plugin:run 目标添加到 pre-clean、clean 和 post-clean 阶段中。这样我们可以在 clean 生命周期的各个阶段显示文本信息。

我们已经在 `C:\MVN\project` 目录下创建了一个 `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/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
  <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <id>id.pre-clean</id>
      <phase>pre-clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>pre-clean phase</echo>
        </tasks>
      </configuration>
    </execution>
    <execution>
      <id>id.clean</id>
      <phase>clean</phase>
      <goals>
       <goal>run</goal>
      </goals>
```

```
      <configuration>
        <tasks>
          <echo>clean phase</echo>
        </tasks>
      </configuration>
    </execution>
    <execution>
      <id>id.post-clean</id>
      <phase>post-clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>post-clean phase</echo>
        </tasks>
      </configuration>
    </execution>
  </executions>
  </plugin>
</plugins>
</build>
</project>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。

```
C:\MVN\project>mvn post-clean
```

Maven 将会开始处理并显示 clean 生命周期的所有阶段。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-clean}]
[INFO] Executing tasks
    [echo] pre-clean phase
[INFO] Executed tasks
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
    [echo] clean phase
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.post-clean}]
[INFO] Executing tasks
```

```
    [echo] post-clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012
[INFO] Final Memory: 4M/44M
[INFO] ------------------------------------------------------------------------
```

你可以尝试修改 mvn clean 命令，来显示 pre-clean 和 clean，而在 post-clean 阶段不执行任何操作。

Default (or Build) 生命周期

这是 Maven 的主要生命周期，被用于构建应用。包括下面的 23 个阶段。

生命周期阶段	描述
validate	检查工程配置是否正确，完成构建过程的所有必要信息是否能够获取到。
initialize	初始化构建状态，例如设置属性。
generate-sources	生成编译阶段需要包含的任何源码文件。
process-sources	处理源代码，例如，过滤任何值（filter any value）。
generate-resources	生成工程包中需要包含的资源文件。
process-resources	拷贝和处理资源文件到目的目录中，为打包阶段做准备。
compile	编译工程源码。
process-classes	处理编译生成的文件，例如 Java Class 字节码的加强和优化。
generate-test-sources	生成编译阶段需要包含的任何测试源代码。
process-test-sources	处理测试源代码，例如，过滤任何值（filter any values)。
test-compile	编译测试源代码到测试目的目录。
process-test-classes	处理测试代码文件编译后生成的文件。
test	使用适当的单元测试框架（例如JUnit）运行测试。
prepare-package	在真正打包之前，为准备打包执行任何必要的操作。
package	获取编译后的代码，并按照可发布的格式进行打包，例如 JAR、WAR 或者 EAR 文件。
pre-integration-test	在集成测试执行之前，执行所需的操作。例如，设置所需的环境变量。
integration-test	处理和部署必须的工程包到集成测试能够运行的环境中。
post-integration-test	在集成测试被执行后执行必要的操作。例如，清理环境。
verify	运行检查操作来验证工程包是有效的，并满足质量要求。
install	安装工程包到本地仓库中，该仓库可以作为本地其他工程的依赖。

生命周期阶段	描述
deploy	拷贝最终的工程包到远程仓库中，以共享给其他开发人员和工程。

有一些与 Maven 生命周期相关的重要概念需要说明：

当一个阶段通过 Maven 命令调用时，例如 mvn compile，只有该阶段之前以及包括该阶段在内的所有阶段会被执行。

不同的 maven 目标将根据打包的类型（JAR / WAR / EAR），被绑定到不同的 Maven 生命周期阶段。

在下面的例子中，我们将 maven-antrun-plugin:run 目标添加到 Build 生命周期的一部分阶段中。这样我们可以显示生命周期的文本信息。

我们已经更新了 C:\MVN\project 目录下的 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/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
<executions>
   <execution>
      <id>id.validate</id>
      <phase>validate</phase>
      <goals>
         <goal>run</goal>
      </goals>
      <configuration>
         <tasks>
            <echo>validate phase</echo>
         </tasks>
      </configuration>
   </execution>
   <execution>
      <id>id.compile</id>
      <phase>compile</phase>
```

```
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>compile phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
    <id>id.test</id>
    <phase>test</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>test phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
      <id>id.package</id>
      <phase>package</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
      <tasks>
        <echo>package phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
    <id>id.deploy</id>
    <phase>deploy</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
    <tasks>
      <echo>deploy phase</echo>
    </tasks>
    </configuration>
  </execution>
```

```
</executions>
</plugin>
</plugins>
</build>
</project>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行以下 mvn 命令。

```
C:\MVN\project>mvn compile
```

Maven 将会开始处理并显示直到编译阶段的构建生命周期的各个阶段。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [compile]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.validate}]
[INFO] Executing tasks
    [echo] validate phase
[INFO] Executed tasks
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\project\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [antrun:run {execution: id.compile}]
[INFO] Executing tasks
    [echo] compile phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sat Jul 07 20:18:25 IST 2012
[INFO] Final Memory: 7M/64M
[INFO] ------------------------------------------------------------------------
```

Site 生命周期

Maven Site 插件一般用来创建新的报告文档、部署站点等。

阶段：

- pre-site
- site
- post-site
- site-deploy

在下面的例子中，我们将 maven-antrun-plugin:run 目标添加到 Site 生命周期的所有阶段中。这样我们可以显示生命周期的所有文本信息。

我们已经更新了 C:\MVN\project 目录下的 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/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-antrun-plugin</artifactId>
<version>1.1</version>
  <executions>
    <execution>
      <id>id.pre-site</id>
      <phase>pre-site</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>pre-site phase</echo>
        </tasks>
      </configuration>
    </execution>
    <execution>
      <id>id.site</id>
      <phase>site</phase>
      <goals>
      <goal>run</goal>
      </goals>
```

```
      <configuration>
        <tasks>
          <echo>site phase</echo>
        </tasks>
      </configuration>
    </execution>
    <execution>
      <id>id.post-site</id>
      <phase>post-site</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>post-site phase</echo>
        </tasks>
      </configuration>
    </execution>
    <execution>
      <id>id.site-deploy</id>
      <phase>site-deploy</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>site-deploy phase</echo>
        </tasks>
      </configuration>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行以下 mvn 命令。

```
C:\MVN\project>mvn site
```

Maven 将会开始处理并显示直到 site 阶段的 site 生命周期的各个阶段。

```
[INFO] Scanning for projects... [INFO] ------------------------------------------------
--------------- [INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0 [INFO] tas
k-segment: [site] [INFO] ------------------------------------------------------------
```

----- [INFO] [antrun:run {execution: id.pre-site}] [INFO] Executing tasks [echo] pre-site phase [INFO] Executed tasks [INFO] [site:site {execution: default-site}] [INFO] Generating "About" report. [INFO] Generating "Issue Tracking" report. [INFO] Generating "Project Team" report. [INFO] Generating "Dependencies" report. [INFO] Generating "Project Plugins" report. [INFO] Generating "Continuous Integration" report. [INFO] Generating "Source Repository" report. [INFO] Generating "Project License" report. [INFO] Generating "Mailing Lists" report. [INFO] Generating "Plugin Management" report. [INFO] Generating "Project Summary" report. [INFO] [antrun:run {execution: id.site}] [INFO] Executing tasks [echo] site phase [INFO] Executed tasks [INFO] -- [INFO] BUILD SUCCESSFUL [INFO] -- [INFO] Total time: 3 seconds [INFO] Finished at: Sat Jul 07 15:25:10 IST 2012 [INFO] Final Memory: 24M/149M [INFO] --

5

Maven – 构建配置文件

什么是构建配置文件?

构建配置文件是一组配置的集合，用来设置或者覆盖 Maven 构建的默认配置。使用构建配置文件，可以为不同的环境定制构建过程，例如 Producation 和 Development 环境。

Profile 在 pom.xml 中使用 activeProfiles / profiles 元素指定，并且可以用很多方式触发。Profile 在构建时修改 POM，并且为变量设置不同的目标环境（例如，在开发、测试和产品环境中的数据库服务器路径）。

Profile 类型

Profile 主要有三种类型。

类型	在哪里定义
Per Project	定义在工程 POM 文件 pom.xml 中
Per User	定义在 Maven 设置 xml 文件中（%USER_HOME%/.m2/settings.xml）
Global	定义在 Maven 全局配置 xml 文件中（%M2_HOME%/conf/settings.xml）

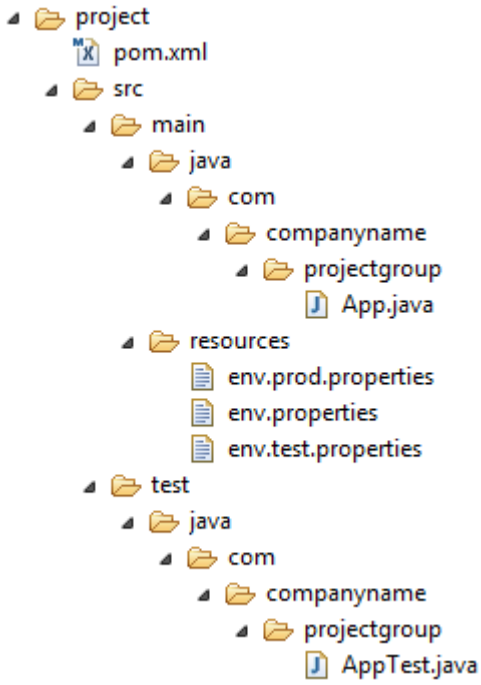
Profile 激活

Maven 的 Profile 能够通过几种不同的方式激活。

- 显式使用命令控制台输入
- 通过 maven 设置
- 基于环境变量（用户 / 系统变量）
- 操作系统配置（例如，Windows family）
- 现存 / 缺失 文件

Profile 激活示例

我们假定你的工程目录像下面这样：



图片 5.1 Maven Build Profile

现在，在 src/main/resources 目录下有三个环境配置文件：

文件名称	描述
env.properties	没有配置文件时的默认配置
env.test.properties	使用测试配置文件时的测试配置
env.prod.properties	使用产品配置文件时的产品配置

显式 Profile 激活

在接下来的例子中，我们将 attach maven-antrun-plugin:run 目标添加到测试阶段中。这样可以我们在不同的 Profile 中输出文本信息。我们将使用 pom.xml 来定义不同的 Profile，并在命令控制台中使用 maven 命令激活 Profile。

假定，我们在 C:\MVN\project 目录下创建了以下的 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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.projectgroup</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
```

```
  <profiles>
    <profile>
    <id>test</id>
    <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <phase>test</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
            <tasks>
              <echo>Using env.test.properties</echo>
        <copy file="src/main/resources/env.test.propertiestofile
        ="${project.build.outputDirectory}/env.properties"/>
            </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
    </build>
    </profile>
  </profiles>
</project>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行以下 mvn 命令。使用 -P 选项指定 Profile 的名称。

```
C:\MVN\project>mvn test -Ptest
```

Maven 将开始处理并显示 test Profile 的结果。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] Copying 3 resources
```

```
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\project\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MVN\project\target\surefire-reports
##  ##  T E S T SThere are no tests to run.

Results :

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

[INFO] [antrun:run {execution: default}]
[INFO] Executing tasks
    [echo] Using env.test.properties
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sun Jul 08 14:55:41 IST 2012
[INFO] Final Memory: 8M/64M
[INFO] ------------------------------------------------------------------------
```

现在我们练习一下，你可以按照下面的步骤做：

- 在 pom.xml 的 profiles 元素中添加另一个 profile 元素（拷贝已有的 profile 元素并粘贴到 profiles 元素结尾）。
- 将此 profile 元素的 id 从 test 修改为 normal。
- 将任务部分修改为 echo env.properties，以及 copy env.properties 到目标目录
- 再次重复以上三个步骤，修改 id 为 prod，修改 task 部分为 env.prod.properties
- 全部就这些了。现在你有了三个构建配置文件（normal / test / prod）。

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。使用 -P 选项指定 Profile 的名称。

```
C:\MVN\project>mvn test -Pnormal
C:\MVN\project>mvn test -Pprod
```

检查构建的输出看看有什么不同。

通过 Maven 设置激活 Profile

打开 Maven 的 settings.xml 文件，该文件可以在 %USER_HOME%/.m2 目录下找到，%USER_HOME% 表示用户主目录。如果 settings.xml 文件不存在则需要创建一个。

像在下面例子中展示的一样，使用 activeProfiles 节点添加 test 配置作为激活的 Profile。

```
<settings 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/xsd/settings-1.0.0.xsd">
  <mirrors>
    <mirror>
      <id>maven.dev.snaponglobal.com</id>
      <name>Internal Artifactory Maven repository</name>
      <url>http://repo1.maven.org/maven2/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  <activeProfiles>
    <activeProfile>test</activeProfile>
  </activeProfiles>
</settings>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。不要使用 -P 选项指定 Profile 的名称。Maven 将显示被激活的 test Profile 的结果。

```
C:\MVN\project>mvn test
```

通过环境变量激活 Profile

现在从 maven 的 settings.xml 中删除激活的 Profile，并更新 pom.xml 中的 test Profile。将下面的内容添加到 profile 元素的 activation 元素中。

当系统属性 “env” 被设置为 “test” 时，test 配置将会被触发。创建一个环境变量 “env” 并设置它的值为 “test”。

```
<profile>
  <id>test</id>
```

```
  <activation>
    <property>
      <name>env</name>
      <value>test</value>
    </property>
  </activation>
</profile>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。

```
C:\MVN\project>mvn test
```

通过操作系统激活 Profile

activation 元素包含下面的操作系统信息。当系统为 windows XP 时，test Profile 将会被触发。

```
<profile>
  <id>test</id>
  <activation>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
  </activation>
</profile>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。不要使用 –P 选项指定 Profile 的名称。Maven 将显示被激活的 test Profile 的结果。

```
C:\MVN\project>mvn test
```

通过现存 / 缺失的文件激活 Profile

现在使用 activation 元素包含下面的操作系统信息。当 *target/generated-sources/axistools/wsdl2java/com/companyname/group* 缺失时，test Profile 将会被触发。

```
<profile>
  <id>test</id>
  <activation>
    <file>
      <missing>target/generated-sources/axistools/wsdl2java/
```

```
      com/companyname/group</missing>
    </file>
  </activation>
</profile>
```

现在打开命令控制台，跳转到 pom.xml 所在目录，并执行下面的 mvn 命令。不要使用 –P 选项指定 Profile 的名称。Maven 将显示被激活的 test Profile 的结果。

```
C:\MVN\project>mvn test
```

6

Maven – 仓库

什么是 Maven 仓库?

在 Maven 的术语中，仓库是一个位置（place），例如目录，可以存储所有的工程 jar 文件、library jar 文件、插件或任何其他的工程指定的文件。

Maven 仓库有三种类型：

- 本地（local）
- 中央（central）
- 远程（remote）

本地仓库

Maven 本地仓库是机器上的一个文件夹。它在你第一次运行任何 maven 命令的时候创建。

Maven 本地仓库保存你的工程的所有依赖（library jar、plugin jar 等）。当你运行一次 Maven 构建，Maven 会自动下载所有依赖的 jar 文件到本地仓库中。它避免了每次构建时都引用存放在远程机器上的依赖文件。

Maven 本地仓库默认被创建在 %USER_HOME% 目录下。要修改默认位置，在 %M2_HOME%\conf 目录中的 Maven 的 settings.xml 文件中定义另一个路径。

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
  http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

当你运行 Maven 命令，Maven 将下载依赖的文件到你指定的路径中。

中央仓库

Maven 中央仓库是由 Maven 社区提供的仓库，其中包含了大量常用的库。

中央仓库的关键概念：

- 这个仓库由 Maven 社区管理。
- 不需要配置。

- 需要通过网络才能访问。

要浏览中央仓库的内容，maven 社区提供了一个 URL：http://search.maven.org/#browse。使用这个仓库，开发人员可以搜索所有可以获取的代码库。

远程仓库

如果 Maven 在中央仓库中也找不到依赖的库文件，它会停止构建过程并输出错误信息到控制台。为避免这种情况，Maven 提供了远程仓库的概念，它是开发人员自己定制仓库，包含了所需要的代码库或者其他工程中用到的 jar 文件。

举例说明，使用下面的 POM.xml，Maven 将从远程仓库中下载该 pom.xml 中声明的所依赖的（在中央仓库中获取不到的）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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.projectgroup</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>com.companyname.common-lib</groupId>
      <artifactId>common-lib</artifactId>
      <version>1.0.0</version>
    </dependency>
  <dependencies>
  <repositories>
    <repository>
      <id>companyname.lib1</id>
      <url>http://download.companyname.org/maven2/lib1</url>
    </repository>
    <repository>
      <id>companyname.lib2</id>
      <url>http://download.companyname.org/maven2/lib2</url>
    </repository>
  </repositories>
</project>
```

Maven 依赖搜索顺序

当我们执行 Maven 构建命令时，Maven 开始按照以下顺序查找依赖的库：

- **步骤 1** – 在本地仓库中搜索，如果找不到，执行步骤 2，如果找到了则执行其他操作。
- **步骤 2** – 在中央仓库中搜索，如果找不到，并且有一个或多个远程仓库已经设置，则执行步骤 4，如果找到了则下载到本地仓库中已被将来引用。
- **步骤 3** – 如果远程仓库没有被设置，Maven 将简单的停滞处理并抛出错误（无法找到依赖的文件）。
- **步骤 4** – 在一个或多个远程仓库中搜索依赖的文件，如果找到则下载到本地仓库已被将来引用，否则 Maven 将停止处理并抛出错误（无法找到依赖的文件）。

7

Maven – 插件

什么是 Maven 插件?

Maven 实际上是一个依赖插件执行的框架，每个任务实际上是由插件完成。Maven 插件通常被用来:

- 创建 jar 文件
- 创建 war 文件
- 编译代码文件
- 代码单元测试
- 创建工程文档
- 创建工程报告

插件通常提供了一个目标的集合，并且可以使用下面的语法执行:

```
mvn [plugin-name]:[goal-name]
```

例如，一个 Java 工程可以使用 maven-compiler-plugin 的 compile-goal 编译，使用以下命令:

```
mvn compiler:compile
```

插件类型

Maven 提供了下面两种类型的插件:

类型	描述
Build plugins	在构建时执行，并在 pom.xml 的 元素中配置。
Reporting plugins	在网站生成过程中执行，并在 pom.xml 的 元素中配置。

下面是一些常用插件的列表:

插件	描述
clean	构建之后清理目标文件。删除目标目录。
compiler	编译 Java 源文件。
surefile	运行 JUnit 单元测试。创建测试报告。
jar	从当前工程中构建 JAR 文件。
war	从当前工程中构建 WAR 文件。
javadoc	为工程生成 Javadoc。
antrun	从构建过程的任意一个阶段中运行一个 ant 任务的集合。

例子

我们已经在我们的例子中大量使用了 maven-antrun-plugin 来输出数据到控制台上。请查看 Maven – 构建配置文件 () 章节。让我们用一种更好的方式理解这部分内容，在 C:\MVN\project 目录下创建一个 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/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.companyname.projectgroup</groupId>
<artifactId>project</artifactId>
<version>1.0</version>
<build>
<plugins>
  <plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-antrun-plugin</artifactId>
  <version>1.1</version>
  <executions>
    <execution>
      <id>id.clean</id>
      <phase>clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>clean phase</echo>
        </tasks>
      </configuration>
    </execution>
  </executions>
  </plugin>
</plugins>
</build>
</project>
```

接下来，打开命令终端跳转到 pom.xml 所在的目录，并执行下面的 mvn 命令。

```
C:\MVN\project>mvn clean
```

Maven 将开始处理并显示 clean 生命周期的 clean 阶段。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed – com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
    [echo] clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012
[INFO] Final Memory: 4M/44M
[INFO] ------------------------------------------------------------------------
```

上面的例子展示了以下关键概念：

- 插件是在 pom.xml 中使用 plugins 元素定义的。
- 每个插件可以有多个目标。
- 你可以定义阶段，插件会使用它的 phase 元素开始处理。我们已经使用了 clean 阶段。
- 你可以通过绑定到插件的目标的方式来配置要执行的任务。我们已经绑定了 echo 任务到 maven-antrun-plugin 的 run 目标。
- 就是这样，Maven 将处理剩下的事情。它将下载本地仓库中获取不到的插件，并开始处理。

8

Maven – 创建工程

Maven 使用**原型（archetype）**插件创建工程。要创建一个简单的 Java 应用，我们将使用 maven-archetype-quickstart 插件。在下面的例子中，我们将在 C:\MVN 文件夹下创建一个基于 maven 的 java 应用工程。

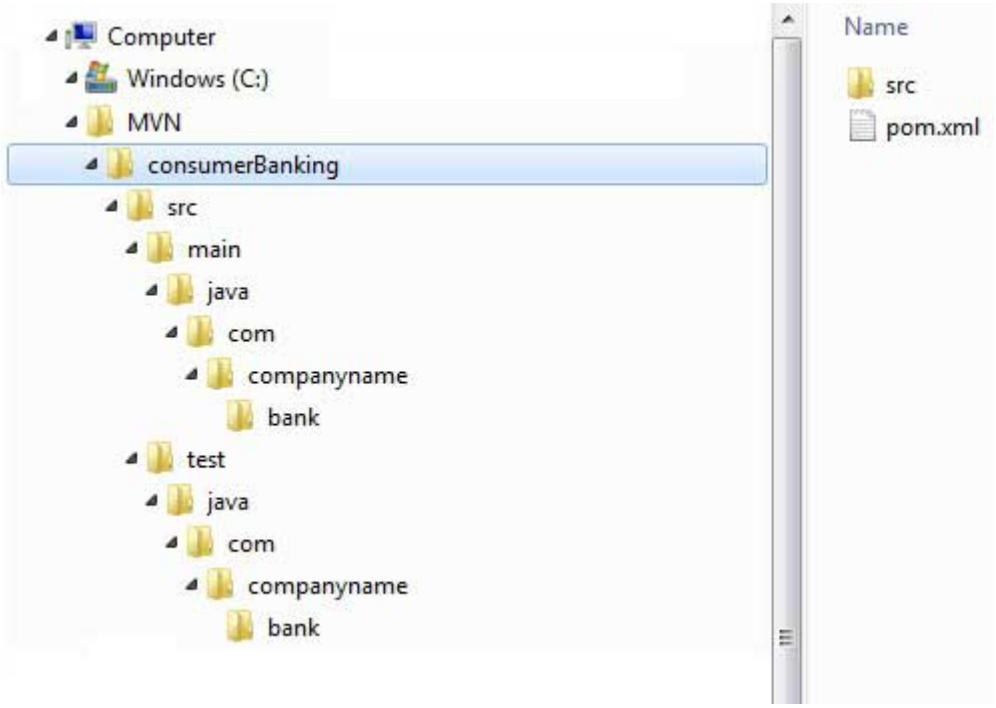
我们打开命令控制台，跳转到 C:\MVN 目录，并执行下面的 **mvn** 命令。

```
C:\MVN>mvn archetype:generate
-DgroupId=com.companyname.bank
-DartifactId=consumerBanking
-DarchetypeArtifactId=maven-archetype-quickstart
-DinteractiveMode=false
```

Maven 将开始处理，并将创建完成的 java 应用工程结构。

```
INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Batch mode
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project
 from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.bank
[INFO] Parameter: packageName, Value: com.companyname.bank
[INFO] Parameter: package, Value: com.companyname.bank
[INFO] Parameter: artifactId, Value: consumerBanking
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\consumerBanking
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 14 seconds
[INFO] Finished at: Tue Jul 10 15:38:58 IST 2012
[INFO] Final Memory: 21M/124M
[INFO] ------------------------------------------------------------------------
```

现在跳转到 C:/MVN 目录。有将看到一个名为 consumerBanking 的 java 应用工程（就像在 artifactId 中设定的一样）。Maven 使用一套标准的目录结构，就像这样：



图片 8.1 Java application project structure

使用上面的例子，我们可以知道下面几个关键概念：

文件夹结构	描述
consumerBanking	包含 src 文件夹和 pom.xml
src/main/java contains	java 代码文件在包结构下（com/companyName/bank）。
src/main/test contains	测试代码文件在包结构下（com/companyName/bank）。
src/main/resources	包含了 图片 / 属性 文件（在上面的例子中，我们需要手动创建这个结构）。

Maven 也创建了一个简单的 Java 源文件和 Java 测试文件。打开 C:\MVN\consumerBanking\src\main\java\com\companyname\bank 文件夹，可以看到 App.java 文件。

```
package com.companyname.bank;

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

打开 C:\MVN\consumerBanking\src\test\java\com\companyname\bank 文件夹，可以看到 AppTest.java。

```
package com.companyname.bank;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class AppTest extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
    public AppTest( String testName )
    {
        super( testName );
    }

    /**
     * @return the suite of tests being tested
     */
    public static Test suite()
    {
        return new TestSuite( AppTest.class );
    }

    /**
     * Rigourous Test :-)
     */
    public void testApp()
    {
        assertTrue( true );
    }
}
```

开发人员需要将他们的文件按照上面表格中提到的结构放置好，接下来 Maven 将会搞定所有构建相关的复杂任务。

在下个章节中，我们将讨论如何使用 maven 构建和测试工程：Maven – 构建 & 测试工程 () 。

9

Maven – 构建 & 测试工程

我们在创建工程章节中学到的是如何使用 Maven 创建 Java 应用。现在我们将看到如何构建和测试这个应用。

跳转到 C:/MVN 目录下，既你的 java 应用目录下。打开 consumerBanking 文件夹。你将看到 POM.xml 文件中有下面的内容。

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.projectgroup</groupId>
    <artifactId>project</artifactId>
    <version>1.0</version>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
      </dependency>
    </dependencies>
</project>
```

可以看到，Maven 已经添加了 JUnit 作为测试框架。默认 Maven 添加了一个源码文件 App.java 和一个测试文件 AppTest.java 到上个章节中我们提到的默认目录结构中。

打开命令控制台，跳转到 C:\MVN\consumerBanking 目录下，并执行以下 mvn 命令。

```
C:\MVN\consumerBanking>mvn clean package
```

Maven 将开始构建工程。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\MVN\consumerBanking\target
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\
resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MVN\consumerBanking\target\classes
```

```
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\consumerBanking\src\test\
resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\MVN\consumerBanking\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MVN\consumerBanking\target\
## surefire-reports##  T E S T SRunning com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec

Results :

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

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\consumerBanking\target\
consumerBanking-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012
[INFO] Final Memory: 16M/89M
[INFO] ------------------------------------------------------------------------
```

你已经构建了你的工程并创建了最终的 jar 文件，下面是要学习的关键概念：

- 我们给了 maven 两个目标，首先清理目标目录（clean），然后打包工程构建的输出为 jar（package）文件。
- 打包好的 jar 文件可以在 consumerBanking\target 中获得，名称为 consumerBanking-1.0-SNAPSHOT.jar。
- 测试报告存放在 consumerBanking\target\surefire-reports 文件夹中。
- Maven 编译源码文件，以及测试源码文件。
- 接着 Maven 运行测试用例。
- 最后 Maven 创建工程包。

现在打开命令控制台，跳转到 C:\MVN\consumerBanking\target\classes 目录，并执行下面的 java 命令。

```
C:\MVN\consumerBanking\target\classes>java com.companyname.bank.App
```

你可以看到结果：

```
Hello World!
```

添加 Java 源文件

我们看看如何添加其他的 Java 文件到工程中。打开 C:\MVN\consumerBanking\src\main\java\com\companyname\bank 文件夹，在其中创建 Util 类 Util.java。

```
package com.companyname.bank;

public class Util
{
  public static void printMessage(String message){
     System.out.println(message);
  }
}
```

更新 App 类来使用 Util 类。

```
package com.companyname.bank;

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

现在打开命令控制台，跳转到 C:\MVN\consumerBanking 目录下，并执行下面的 mvn 命令。

```
C:\MVN\consumerBanking>mvn clean compile
```

在 Maven 构建成功之后，跳转到 C:\MVN\consumerBanking\target\classes 目录下，并执行下面的 java 命令。

```
C:\MVN\consumerBanking\target\classes>java –cp com.companyname.bank.App
```

你可以看到结果：

```
Hello World!
```

10

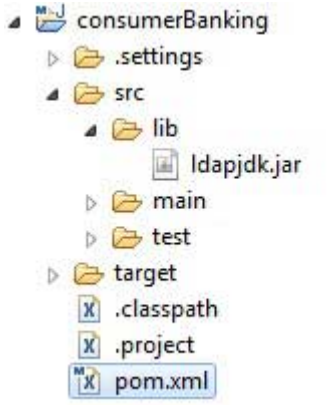
Maven – 外部依赖

现在，如你所知道的，Maven的依赖管理使用的是 Maven – 仓库 () 的概念。但是如果在远程仓库和中央仓库中，依赖不能被满足，如何解决呢? Maven 使用外部依赖的概念来解决这个问题。

例如，让我们对在 Maven – 创建工程 () 部分创建的项目做以下修改:

- 在 src 文件夹下添加 lib 文件夹
- 复制任何 jar 文件到 lib 文件夹下。我们使用的是 ldapjdk.jar ，它是为 LDAP 操作的一个帮助库

现在，我们的工程结构应该像下图一样:



图片 10.1 external-project-structure

现在你有了自己的工程库（library），通常情况下它会包含一些任何仓库无法使用，并且 maven 也无法下载的 jar 文件。如果你的代码正在使用这个库，那么 Maven 的构建过程将会失败，因为在编译阶段它不能下载或者引用这个库。

为了处理这种情况，让我们用以下方式，将这个外部依赖添加到 maven pom.xml 中。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.companyname.bank</groupId>
  <artifactId>consumerBanking</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>consumerBanking</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

```
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>ldapjdk</groupId>
      <artifactId>ldapjdk</artifactId>
      <scope>system</scope>
      <version>1.0</version>
      <systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath>
    </dependency>
  </dependencies>

</project>
```

上例中， `<dependencies>` 的第二个 `<dependency>` 元素，阐明了**外部依赖**的关键概念。

- 外部依赖（library jar location）能够像其他依赖一样在 pom.xml 中配置。
- 指定 groupId 为 library 的名称。
- 指定 artifactId 为 library 的名称。
- 指定作用域（scope）为系统。
- 指定相对于工程位置的系统路径。

希望现在你懂得了有关外部依赖的知识，你将能够在你的 Maven 工程中指定外部依赖。

11

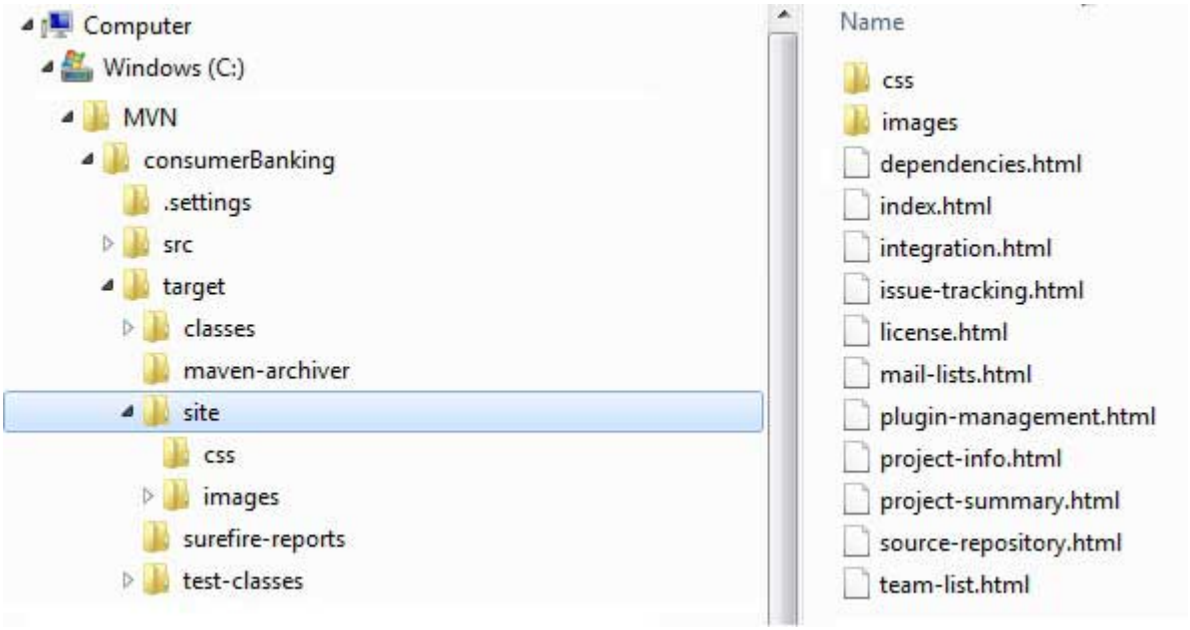
Maven – 工程文档

本教程将教你如何创建应用程序的文档。那么让我们开始吧，在 C:/ MVN 目录下，创建你的 java consumerBanking 应用程序。打开 consumerBanking 文件夹并执行以下 mvn 命令。

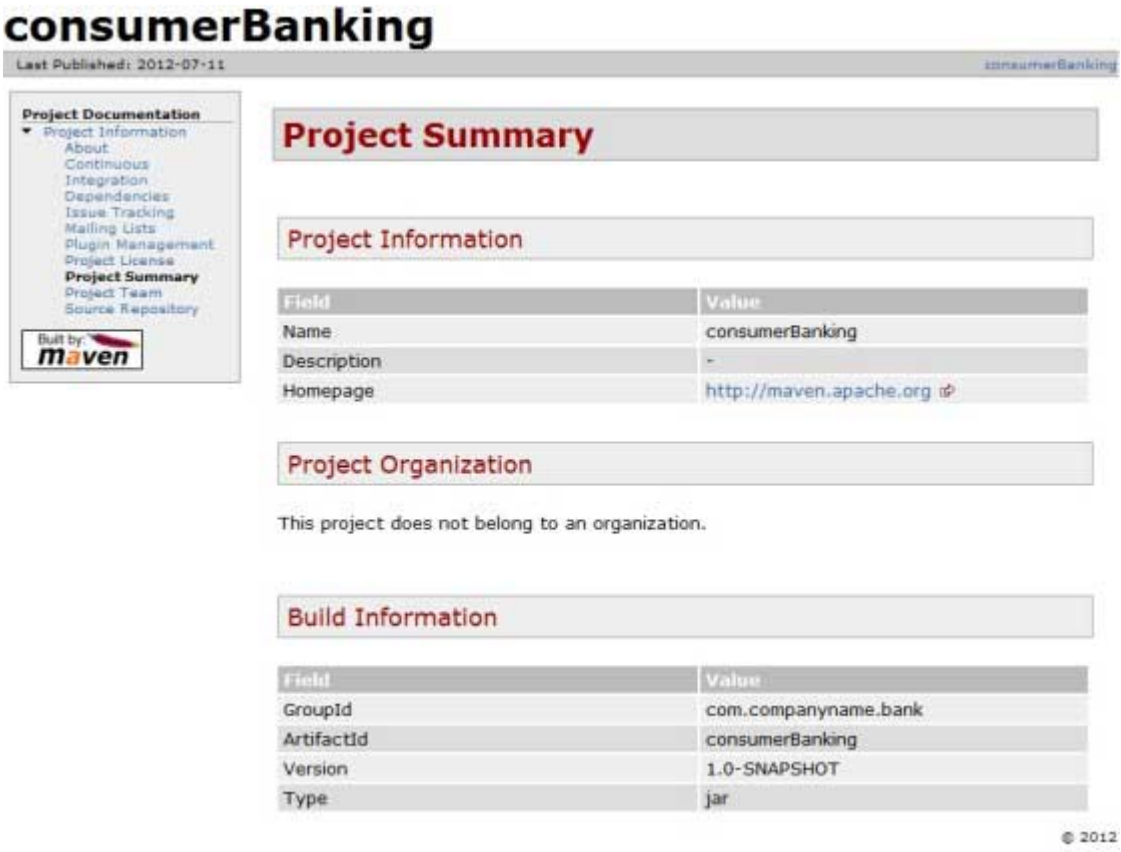
```
C:\MVN>mvn site
```

Maven 将开始构建工程。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]task-segment: [site]
[INFO] ------------------------------------------------------------------------
[INFO] [site:site {execution: default-site}]
[INFO] artifact org.apache.maven.skins:maven-default-skin:
checking for updates from central
[INFO] Generating "About" report.
[INFO] Generating "Issue Tracking" report.
[INFO] Generating "Project Team" report.
[INFO] Generating "Dependencies" report.
[INFO] Generating "Continuous Integration" report.
[INFO] Generating "Source Repository" report.
[INFO] Generating "Project License" report.
[INFO] Generating "Mailing Lists" report.
[INFO] Generating "Plugin Management" report.
[INFO] Generating "Project Summary" report.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 11 18:11:18 IST 2012
[INFO] Final Memory: 23M/148M
[INFO] ------------------------------------------------------------------------
```



打开 C:\MVN\consumerBanking\target\site 文件夹。点击 index.html 就可以看到文档了。



Maven 使用称作 Doxia 的文件处理引擎创建文档，它将多个源格式的文件转换为一个共通的文档模型。要编写工程文档，你可以使用以下能够被 Doxia 解析的几种常用格式来编写。

格式名称	描述	引用
XDoc	Maven 1.x 版本的文档格式	http://jakarta.apache.org/site/jakarta-site2.html
FML	FAQ 文档格式	http://maven.apache.org/doxia/references/fml-format.html
XHTML	可扩展 HTML 格式	http://en.wikipedia.org/wiki/XHTML

12

Maven – 工程模板

Maven 使用原型（Archetype）概念为用户提供了大量不同类型的工程模版（614 个）。Maven 使用下面的命令帮助用户快速创建 java 项目。

```
mvn archetype:generate
```

什么是原型?

原型是一个 Maven 插件，它的任务是根据模板创建一个项目结构。我们将使用 quickstart 原型插件创建一个简单的 java 应用程序。

使用工程模板

让我们打开命令控制台，跳转到 C:\ > MVN 目录并执行以下 mvn 命令

```
C:\MVN>mvn archetype:generate
```

Maven 将开始处理，并要求选择所需的原型

```
INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
...
600: remote -> org.trailsframework:trails-archetype (-)
601: remote -> org.trailsframework:trails-secure-archetype (-)
602: remote -> org.tynamo:tynamo-archetype (-)
603: remote -> org.wicketstuff.scala:wicket-scala-archetype (-)
604: remote -> org.wicketstuff.scala:wicketstuff-scala-archetype
Basic setup for a project that combines Scala and Wicket,
depending on the Wicket-Scala project.
Includes an example Specs test.)
605: remote -> org.wikbook:wikbook.archetype (-)
606: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-glassfish (-)
607: remote -> org.xaloon.archetype:xaloon-archetype-wicket-jpa-spring (-)
608: remote -> org.xwiki.commons:xwiki-commons-component-archetype
(Make it easy to create a maven project for creating XWiki Components.)
609: remote -> org.xwiki.rendering:xwiki-rendering-archetype-macro
(Make it easy to create a maven project for creating XWiki Rendering Macros.)
610: remote -> org.zkoss:zk-archetype-component (The ZK Component archetype)
```

```
611: remote -> org.zkoss:zk-archetype-webapp (The ZK wepapp archetype)
612: remote -> ru.circumflex:circumflex-archetype (-)
613: remote -> se.vgregion.javg.maven.archetypes:javg-minimal-archetype (-)
614: remote -> sk.seges.sesam:sesam-annotation-archetype (-)
Choose a number or apply filter
(format: [groupId:]artifactId, case sensitive contains): 203:
```

按下 Enter 选择默认选项 (203:maven-archetype-quickstart)

Maven 将询问原型的版本

```
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6:
```

按下 Enter 选择默认选项 (6:maven-archetype-quickstart:1.1)

Maven 将询问工程细节。按要求输入工程细节。如果要使用默认值则直接按 Enter 键。你也可以输入自己的值。

```
Define value for property 'groupId': : com.companyname.insurance
Define value for property 'artifactId': : health
Define value for property 'version': 1.0-SNAPSHOT:
Define value for property 'package': com.companyname.insurance:
```

Maven 将要求确认工程细节。按 enter 或按 Y

```
Confirm properties configuration:
groupId: com.companyname.insurance
artifactId: health
version: 1.0-SNAPSHOT
package: com.companyname.insurance
Y:
```

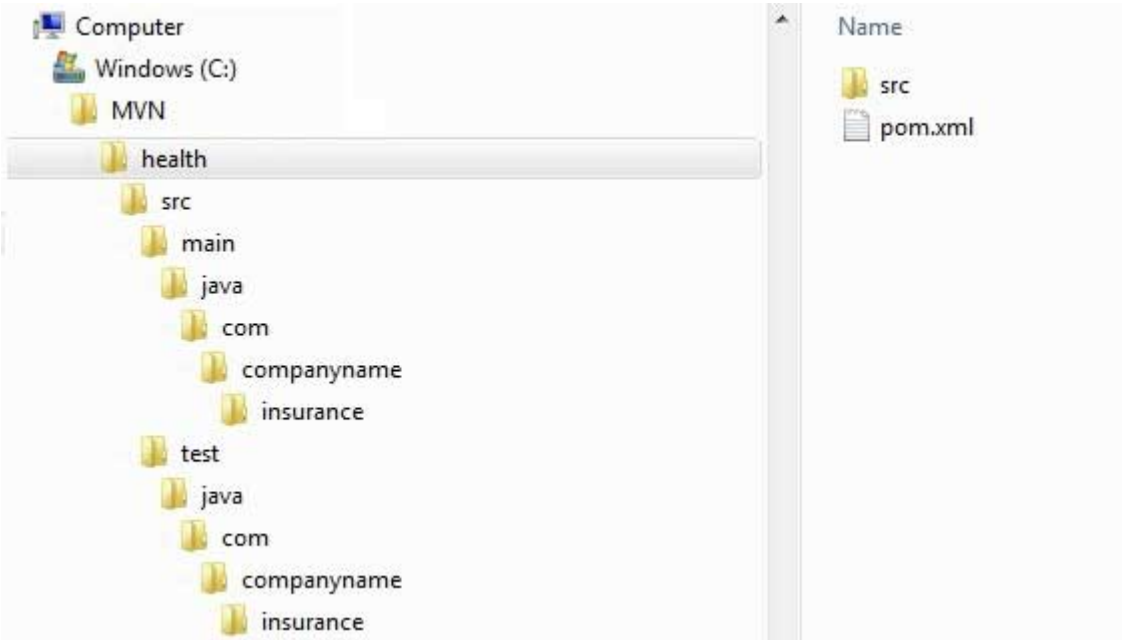
现在 Maven 将开始创建工程结构，显示如下:

```
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-quickstart:1.1
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.insurance
```

```
[INFO] Parameter: packageName, Value: com.companyname.insurance
[INFO] Parameter: package, Value: com.companyname.insurance
[INFO] Parameter: artifactId, Value: health
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\health
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 minutes 12 seconds
[INFO] Finished at: Fri Jul 13 11:10:12 IST 2012
[INFO] Final Memory: 20M/90M
[INFO] ------------------------------------------------------------------------
```

创建的项目

现在转到 C:\ > MVN 目录。你会看到一个名为 health 的 java 应用程序项目，就像在项目创建的时候建立的 artifactId 名称一样。 Maven 将创建一个有标准目录布局的工程，如下所示:



创建 POM.xml

Maven 为工程生成一个 POM.xml文件，如下所列:

```
<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/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.companyname.insurance</groupId>
 <artifactId>health</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>
 <name>health</name>
 <url>http://maven.apache.org</url>
 <properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>
 <dependencies>
   <dependency>
   <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>3.8.1</version>
     <scope>test</scope>
   </dependency>
 </dependencies>
</project>
```

创建 App.java

Maven 生成一个 java 源文件示例，工程的 App.java 如下所示:

路径：C:\ > MVN > health > src > main > java > com > companyname > insurance > App.java

```
package com.companyname.insurance;

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

创建 AppTest.java

Maven 生成一个 java 源文件示例，工程的 AppTest.java 如下所示：

路径为： C:\ > MVN > health > src > test > java > com > companyname > insurance > AppTest.java

```
package com.companyname.insurance;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
* Unit test for simple App.
*/
public class AppTest
   extends TestCase
{
   /**
   * Create the test case
   *
   * @param testName name of the test case
   */
  public AppTest( String testName )
  {
     super( testName );
  }

  /**
  * @return the suite of tests being tested
  */
  public static Test suite()
  {
     return new TestSuite( AppTest.class );
  }

  /**
  * Rigourous Test :-)
  */
  public void testApp()
  {
     assertTrue( true );
```

```
 }
}
```

就这样。现在你可以看到 Maven 的强大之处。你可以用 maven 简单的命令创建任何类型的工程，并且可以启动您的开发。

13

Maven – 快照

大型软件应用程序通常由多个模块组成，这是多个团队工作于同一应用程序的不同模块的常见场景。例如一个团队工作负责应用程序的前端应用用户接口工程（app-ui.jar:1.0)），同时他们使用数据服务工程（data-service.jar:1.0）。

现在负责数据服务的团队可能正在进行修正 bug 或者增强功能，并快速迭代，然后他们几乎每天都会 release 工程库文件到远程仓库中。

现在如果数据服务团队每天上传新的版本，那么就会有下面的问题：

- 每次数据服务团队发布了一版更新的代码时，都要告诉应用接口团队。
- 应用接口团队需要定期更新他们的 pom.xml 来得到更新的版本

为了解决这样的情况，**快照**概念发挥了作用.

什么是快照?

快照是一个特殊的版本，它表示当前开发的一个副本。与常规版本不同，Maven 为每一次构建从远程仓库中检出一份新的快照版本。

现在数据服务团队会将每次更新的代码的快照（例如 data-service:1.0-SNAPSHOT）发布到仓库中，来替换旧的快照 jar 文件。

快照 vs 版本

对于版本，Maven 一旦下载了指定的版本（例如 data-service:1.0），它将不会尝试从仓库里再次下载一个新的 1.0 版本。想要下载新的代码，数据服务版本需要被升级到 1.1。

对于快照，每次用户接口团队构建他们的项目时，Maven 将自动获取最新的快照（data-service:1.0-SNAPSHOT）。

应用用户接口 pom.xml

应用用户接口工程正在使用 1.0 版本的数据服务的快照

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

```
  <groupId>app-ui</groupId>
  <artifactId>app-ui</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <name>health</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
    <groupId>data-service</groupId>
      <artifactId>data-service</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

数据服务 pom.xml

数据服务工程为每个微小的变化 release 1.0 快照

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>data-service</groupId>
  <artifactId>data-service</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>health</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  </project>
```

虽然，对于快照，Maven 每次自动获取最新的快照，但你可以在任何 maven 命令中使用 -U 参数强制 maven 下载最新的快照。

```
mvn clean package -U
```

让我们打开命令控制台，进入 C:\ > MVN > app-ui 目录并执行以下 mvn 命令。

```
C:\MVN\app-ui>mvn clean package -U
```

Maven将在下载数据服务的最新快照后，开始构建工程。

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] Downloading data-service:1.0-SNAPSHOT
[INFO] 290K downloaded.
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\MVN\app-ui\target
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\app-ui\src\main\
resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MVN\app-ui\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\MVN\app-ui\src\test\
resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\MVN\app-ui\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MVN\app-ui\target\
## surefire-reports## T E S T SRunning com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec

Results :

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

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-ui\target\
app-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
```

```
[INFO] Finished at: Tue Jul 10 16:52:18 IST 2012
[INFO] Final Memory: 16M/89M
[INFO] ------------------------------------------------------------------------
```

14

Maven – 构建自动化

构建自动化定义为一种场景：一旦该工程成功构建完成，其相关的依赖工程即开始构建，目的是为了保证其依赖项目的稳定。

实例

考虑一个团队正在开发一个关于总线核心 Api（称其为 bus-core-api）的工程，依赖它的工程有 2 个，分别为网页 UI（称其为 app-web-ui）和应用程序桌面 UI（称其为 app-desktop-ui）。

app-web-ui 工程使用 1.0-SNAPSHOT 总线核心 Api 工程，其 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>app-web-ui</groupId>
  <artifactId>app-web-ui</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
    <groupId>bus-core-api</groupId>
      <artifactId>bus-core-api</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

app-desktop-ui 工程也正在使用 1.0-SNAPSHOT 总线核心 Api 工程，其 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>app-desktop-ui</groupId>
  <artifactId>app-desktop-ui</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
    <groupId>bus-core-api</groupId>
      <artifactId>bus-core-api</artifactId>
      <version>1.0-SNAPSHOT</version>
```

```
    </dependency>
  </dependencies>
</project>
```

bus-core-api 工程的 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>bus-core-api</groupId>
  <artifactId>bus-core-api</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
</project>
```

现在，app-web-ui 和 app-desktop-ui 工程的团队需要保证当 bus-core-api 工程有变化时他们自己相应的工程可以随时被构建。

使用快照可以保证最新的 bus-core-api 工程可以被使用，但是为了达到上述的需求，我们仍需做一些额外的工作。

我们有 2 种方式：

- 在 bus-core-api 的 pom 文件里添加一个编译目标来提醒 app-web-ui 工程和 app-desktop-ui 工程启动创建。
- 使用一个持续集成（CI）的服务器，比如 Hudson，来实现自动化创建。

使用 Maven

更新 bus-core-api 工程的 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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>bus-core-api</groupId>
  <artifactId>bus-core-api</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <build>
```

```
  <plugins>
  <plugin>
  <artifactId>maven-invoker-plugin</artifactId>
  <version>1.6</version>
    <configuration>
      <debug>true</debug>
      <pomIncludes>
        <pomInclude>app-web-ui/pom.xml</pomInclude>
        <pomInclude>app-desktop-ui/pom.xml</pomInclude>
      </pomIncludes>
    </configuration>
    <executions>
      <execution>
        <id>build</id>
        <goals>
          <goal>run</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  </plugins>
  <build>
</project>
```

打开命令终端，进入到 C:\ > MVN > bus-core-api 的目录下，然后执行如下的 mvn 的命令。

```
C:\MVN\bus-core-api>mvn clean package -U
```

Maven 将会开始构建 bus-core-api 工程，输出日志如下：

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building bus-core-api
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\bus-core-ui\target\
bus-core-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

一旦 bus-core-api 构建成功，Maven 将会自动开始构建 app-web-ui 项目， 日志如下：

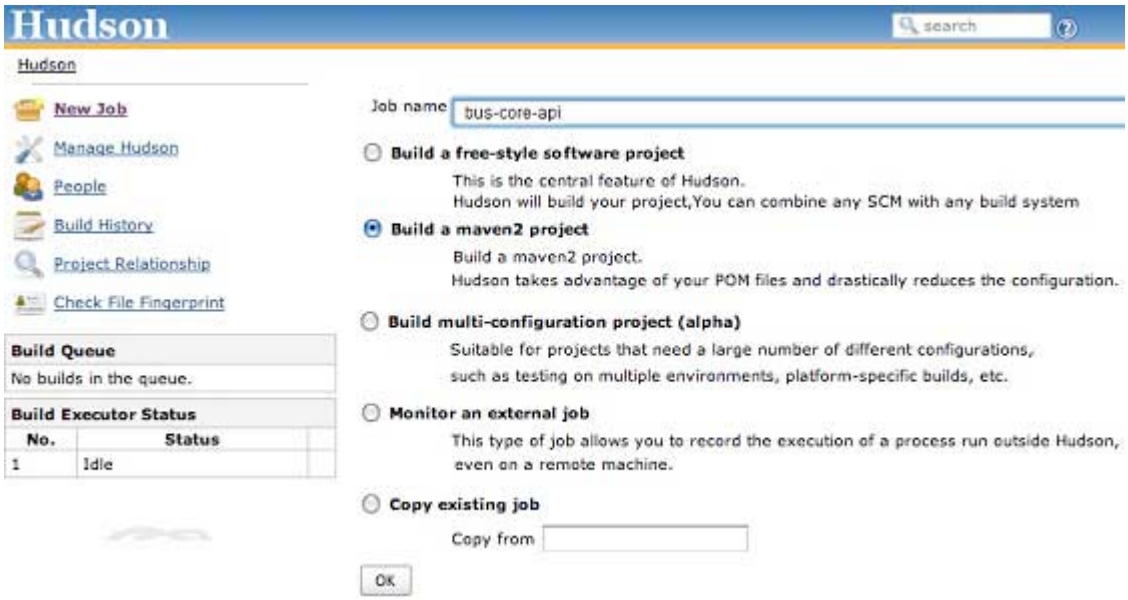
```
[INFO] ------------------------------------------------------------------
[INFO] Building app-web-ui
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-web-ui\target\
app-web-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
```

等到 app-web-ui 创建成功，Maven 接着开始构建 app-desktop-ui 工程，日志输出如下：

```
[INFO] ------------------------------------------------------------------
[INFO] Building app-desktop-ui
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------------
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\MVN\app-desktop-ui\target\
app-desktop-ui-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------
```

使用持续集成服务器（CI）

由于开发人员不需要每次多一个新的依赖工程时都去更新 bus-core-api 工程的 pom 文件， 因此使用一个持续集成的服务器更加合适，例如，添加一个新的 app-mobile-ui 的工程，它同样依赖于 bus-core-ui 工程。Hudson 将会借助 Maven 的依赖管理功能实现工程的自动化创建。



图片 14.1 automated build

Hunson 把每次创建工程看做一个工作。一旦工程代码合入到 svn 或者其他任何的映射到Hudson上的代码源管理工具上，Hunson 便开始一次的创建工作，等到该工作完成后，它将会自动创建其他相关的依赖工作或者依赖工程。

在上面的例子中，当 bus-core-api 的源代码在 SVN 上有更新时， Hudson 将会启动创建。当创建完成后，Hudson 开始自动寻找其依赖的工程，然后启动 app-web-ui 和 app-desktop-ui 工程。

15

Maven – 依赖管理

Maven 核心特点之一是依赖管理。一旦我们开始处理多模块工程（包含数百个子模块或者子工程）的时候，模块间的依赖关系就变得非常复杂，管理也变得很困难。针对此种情形，Maven 提供了一种高度控制的方法。

传递依赖发现

这种情形经常可见，当一个库 A 依赖于其他库 B. 另一工程 C 想要使用库 A, 那么该工程同样也需要使用到库 B。

Maven 可以避免去搜索所有需要的库资源的这种需求。通过读取工程文件（pom.xml）中的依赖项，Maven 可以找出工程之间的依赖关系。

我们只需要在每个工程的 pom 文件里去定义直接的依赖关系。Maven 则会自动的来接管后续的工作。

通过传递依赖，所有被包含的库的图形可能会快速的增长。当重复的库存在时，可能出现的情形将会持续上升。Maven 提供一些功能来控制可传递的依赖的程度。

功能	功能描述
依赖调节	决定当多个手动创建的版本同时出现时，哪个依赖版本将会被使用。 如果两个依赖版本在依赖树里的深度是一样的时候，第一个被声明的依赖将会被使用。
依赖管理	直接的指定手动创建的某个版本被使用。例如当一个工程 C 在自己的以来管理模块包含工程 B，即 B 依赖于 A， 那么 A 即可指定在 B 被引用时所使用的版本。
依赖范围	包含在构建过程每个阶段的依赖。
依赖排除	任何可传递的依赖都可以通过 "exclusion" 元素被排除在外。举例说明，A 依赖 B， B 依赖 C，因此 A 可以标记 C 为 “被排除的”。
依赖可选	任何可传递的依赖可以被标记为可选的，通过使用 "optional" 元素。例如：A 依赖 B， B 依赖 C。因此，B 可以标记 C 为可选的， 这样 A 就可以不再使用 C。

依赖范围

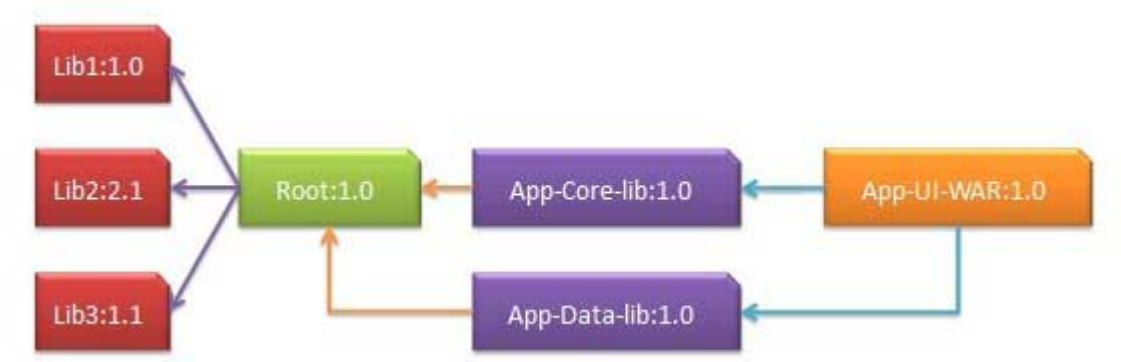
传递依赖发现可以通过使用如下的依赖范围来得到限制：

范围	描述
编译阶段	该范围表明相关依赖是只在工程的类路径下有效。默认取值。
供应阶段	该范围表明相关依赖是由运行时的 JDK 或者 网络服务器提供的。
运行阶段	该范围表明相关依赖在编译阶段不是必须的，但是在执行阶段是必须的。
测试阶段	该范围表明相关依赖只在测试编译阶段和执行阶段。

范围	描述
系统阶段	该范围表明你需要提供一个系统路径。
导入阶段	该范围只在依赖是一个 pom 里定义的依赖时使用。同时，当前工程的POM 文件的 部分定义的依赖关系可以取代某特定的 POM。

依赖管理

通常情况下，在一个共通的工程下，有一系列的工程。在这种情况下，我们可以创建一个公共依赖的 pom 文件，该 pom 包含所有的公共的依赖关系，我们称其为其他子工程 pom 的 pom 父。 接下来的一个例子可以帮助你更好的理解这个概念。



图片 15.1 dependency graph

下面是上述依赖图表的细节：

- App-UI-WAR 依赖于 App-Core-lib 和 App-Data-lib.
- Root 是 App-Core-lib 和 App-Data-lib 的父类。
- Root 在其依赖模块里定义了 Lib1,lib2, Lib3 3个依赖关系。

App-UI-WAR 的 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>App-UI-WAR</artifactId>
    <version>1.0</version>
    <packaging>war</packaging>
    <dependencies>
      <dependency>
```

```
        <groupId>com.companyname.groupname</groupId>
        <artifactId>App-Core-lib</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
    <dependencies>
      <dependency>
        <groupId>com.companyname.groupname</groupId>
        <artifactId>App-Data-lib</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
</project>
```

App-Core-lib 的 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
    <parent>
      <artifactId>Root</artifactId>
      <groupId>com.companyname.groupname</groupId>
      <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>App-Core-lib</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
</project>
```

App-Data-lib 的 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
    <parent>
      <artifactId>Root</artifactId>
      <groupId>com.companyname.groupname</groupId>
      <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>App-Data-lib</artifactId>
```

```
    <version>1.0</version>
    <packaging>jar</packaging>
</project>
```

Root 的 POM 文件如下：

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.companyname.groupname</groupId>
    <artifactId>Root</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
    <dependencies>
      <dependency>
        <groupId>com.companyname.groupname1</groupId>
        <artifactId>Lib1</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
    <dependencies>
      <dependency>
        <groupId>com.companyname.groupname2</groupId>
        <artifactId>Lib2</artifactId>
        <version>2.1</version>
      </dependency>
    </dependencies>
    <dependencies>
      <dependency>
        <groupId>com.companyname.groupname3</groupId>
        <artifactId>Lib3</artifactId>
        <version>1.1</version>
      </dependency>
    </dependencies>
</project>
```

现在，当我们构建 App-UI-WAR 工程时， Maven 将会通过遍历依赖图找到所有的依赖关系，并且构建该应用程序。

通过上面的例子，我们可以学习到以下关键概念：

- 公共的依赖可以使用 pom 父的概念被统一放在一起。App-Data-lib 和 App-Core-lib 工程的依赖在Root 工程里列举了出来（参考 Root的包类型，它是一个 POM）.

- 没有必要在 App-UI-W 里声明 Lib1, lib2, Lib3 是它的依赖。 Maven 通过使用可传递的依赖机制来实现该细节。

16

Maven – 自动化部署

一般情况下，在一个工程开发进程里，一次部署的过程包含需如下步骤：

- 合入每个子工程下的代码到 SVN 或者源代码库，并标记它。
- 从 SVN 下载完整的源代码。
- 构建应用程序。
- 保存构建结果为 WAR 或者 EAR 类型文件并存放到一个共同的指定的网络位置上。
- 从网络上获得该文件并且部署该文件到产品线上。
- 更新文档日期和应用程序的版本号。

问题陈述

通常，将会有很多不同的人参与到上述部署过程中。一个团队可以负责代码的合入工作，另外一个可以负责构建，以此类推。上述的任何一个步骤都可能因为人为的原因没有被执行。例如，较旧的版本没有在网络机器上更新，负责部署的团队再一次部署了旧的版本。

解决方案

通过结合如下的方案来实现自动化部署：

- Maven 构建和发布项目，
- SubVersion, 源代码库用以管理源代码，
- 远程仓库管理工具 (Jfrog/Nexus) 用以管理工程的二进制文件。

更新工程的 POM.xml

我们将会使用 Maven 发布的插件来创建一个自动化发布过程：

例如： bus-core-api 工程的 POM.xml 如下

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>bus-core-api</groupId>
  <artifactId>bus-core-api</artifactId>
```

```
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <scm>
    <url>http://www.svn.com</url>
    <connection>scm:svn:http://localhost:8080/svn/jrepo/trunk/
    Framework</connection>
    <developerConnection>scm:svn:${username}/${password}@localhost:8080:
    common_core_api:1101:code</developerConnection>
  </scm>
  <distributionManagement>
    <repository>
      <id>Core-API-Java-Release</id>
      <name>Release repository</name>
      <url>http://localhost:8081/nexus/content/repositories/
      Core-Api-Release</url>
    </repository>
  </distributionManagement>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.0-beta-9</version>
        <configuration>
          <useReleaseProfile>false</useReleaseProfile>
          <goals>deploy</goals>
          <scmCommentPrefix>[bus-core-api-release-checkin]-<
          /scmCommentPrefix>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

在 pom.xml 里，我们常常会使用到的重要元素如下表：

元素	描述
SCM	配置 SVN 的路径，Maven 将从该路径下将代码取下来。
仓库	成功构建出来的 WAR/EAR/JAR 或者其他的构建结果存放的路径。
插件	maven-release-plugin 用以自动化部署的过程。

Maven Release 插件

Maven 通过 maven-release-plugin 来执行如下很有用的任务：

```
mvn release:clean
```

清理工作空间，保证最新的发布进程成功进行。

```
mvn release:rollback
```

回滚修改的工作空间代码和配置保证发布过程成功进行。

```
mvn release:prepare
```

执行如下多次操作：

- 检查本地是否存在还未提交的修改
- 确保没有快照的依赖
- 改变应用程序的版本信息用以发布
- 更新 POM 文件到 SVN
- 运行测试用例
- 提交修改后的 POM 文件
- 为代码在 SVN 上做标记
- 增加版本号和附加快照以备将来发布
- 提交修改后的 POM 文件到 SVN.

```
mvn release:perform
```

将代码切换到之前做标记的地方，运行 Maven 部署目标来部署 WAR 文件或者构建相应的结构到仓库里。

打开命令终端，进入到 C:\ > MVN >bus-core-api 目录下，然后执行如下的 mvn 命令。

```
C:\MVN\bus-core-api>mvn release:prepare
```

Maven 开始构建整个工程。一旦构建成功即可运行如下 mvn 命令。

```
C:\MVN\bus-core-api>mvn release:perform
```

一旦构建成功，你可以验证在你仓库下上传的 JAR 文件是否生效。

17

Maven – Web 应用

本教程将指导你如何使用 Maven 版本控制系统来管理一个基于 Web 的工程。在此，你将学习到如何创建/构建/部署以及运行 Web 应用程序：

创建 Web 应用

建立一个简单的 Java web 应用，我们可以使用 maven-archetype-webapp 插件。首先我们打开命令控制台，进入 C:\MVN 目录并且执行以下的 mvn 命令。

```
C:\MVN>mvn archetype:generate
-DgroupId=com.companyname.automobile
-DartifactId=trucks
-DarchetypeArtifactId=maven-archetype-webapp
-DinteractiveMode=false
```

Maven 将开始处理并且将创建完整的基于 Web 的 java 应用工程结构。

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Batch mode
[INFO] ------------------------------------------------------------------------
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-webapp:1.0
[INFO] ------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.automobile
[INFO] Parameter: packageName, Value: com.companyname.automobile
[INFO] Parameter: package, Value: com.companyname.automobile
[INFO] Parameter: artifactId, Value: trucks
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\trucks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Tue Jul 17 11:00:00 IST 2012
```

```
[INFO] Final Memory: 20M/89M
[INFO] ------------------------------------------------------------------------
```

现在进入 C:/MVN 目录，你将看到一个名为 trucks（由 artifactId 指定）的 java 应用工程。

图片 17.1 Java web application project structure

Maven 使用一个标准的目录架构，如上示例，我们可以理解以下的关键概念：

文件夹结构	描述
trucks	包含 src 文件夹和 pom.xml
src/main/webapp	包含 index.jsp 和 WEB-INF 文件夹.
src/main/webapp/WEB-INF	包含 web.xml
src/main/resources	包含 images / properties 文件。

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>
  <groupId>com.companyname.automobile</groupId>
  <artifactId>trucks</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>trucks 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>trucks</finalName>
  </build>
</project>
```

如果仔细观察，Maven 还创建了一个示例 JSP 的源文件。

打开 C:\ > MVN > trucks > src > main > webapp > 文件夹, 你将看到 index.jsp.

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

Build Web Application

打开终端，进入 C:\MVN\trucks 目录，然后执行如下 mvn 命令.

```
C:\MVN\trucks>mvn clean package
```

Maven 将会开始构建此工程，日志输出如下:

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building trucks Maven Webapp
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO] [compiler:compile {execution: default-compile}]
[INFO] No sources to compile
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to
copy filtered resources,i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory
C:\MVN\trucks\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
```

```
[INFO] No sources to compile
[INFO] [surefire:test {execution: default-test}]
[INFO] No tests to run.
[INFO] [war:war {execution: default-war}]
[INFO] Packaging webapp
[INFO] Assembling webapp[trucks] in [C:\MVN\trucks\target\trucks]
[INFO] Processing war project
[INFO] Copying webapp resources[C:\MVN\trucks\src\main\webapp]
[INFO] Webapp assembled in[77 msecs]
[INFO] Building war: C:\MVN\trucks\target\trucks.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Tue Jul 17 11:22:45 IST 2012
[INFO] Final Memory: 11M/85M
[INFO] ------------------------------------------------------------------------
```

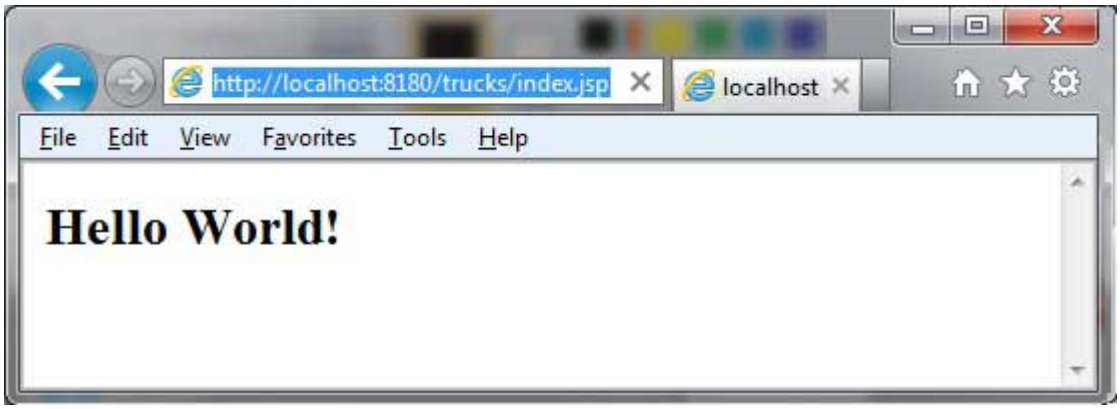
部署 Web 应用

现在拷贝在 C:\ > MVN > trucks > target > 文件夹下的 trucks.war 到你的 web 服务器的 webapp 目录下，并且重启 web 服务。

测试 Web 应用

使用 URL: http://\<server-name>:\<port-number>/trucks/index.jsp 来运行你的 Web 应用。

验证输出结果：



图片 17.2 web page

18

Maven – Elipse IDE

Eclipse 提供一种卓越的插件 m2eclipse (http://www.eclipse.org/m2e/) ，该插件使得 Maven 和 Eclipse 能够无缝集成。

下面列出 m2eclipse 的一些特点：

- 可以在 Eclipse 环境上运行 Maven 的目标文件。
- 可以使用其自带的控制台在 Eclipse 中直接查看 Maven 命令的输出。
- 可以在 IDE 下更新 Maven 的依赖关系。
- 可以使用 Eclipse 开展 Maven 工程的构建。
- Eclipse 基于 Maven 的 pom.xml 来实现自动化管理依赖关系。
- 它解决了 Maven 与 Eclipse 的工作空间之间的依赖，而不需要安装到本地 Maven 的存储库（需要依赖项目在同一个工作区）。
- 它可以自动地从远端的 Maven 库中下载所需要的依赖以及源码。
- 它提供了向导，为建立新 Maven 工程，pom.xml 以及在已有的工程上开启 Maven 支持。
- 它提供了远端的 Maven 存储库的依赖的快速搜索。

安装 m2eclipse 插件

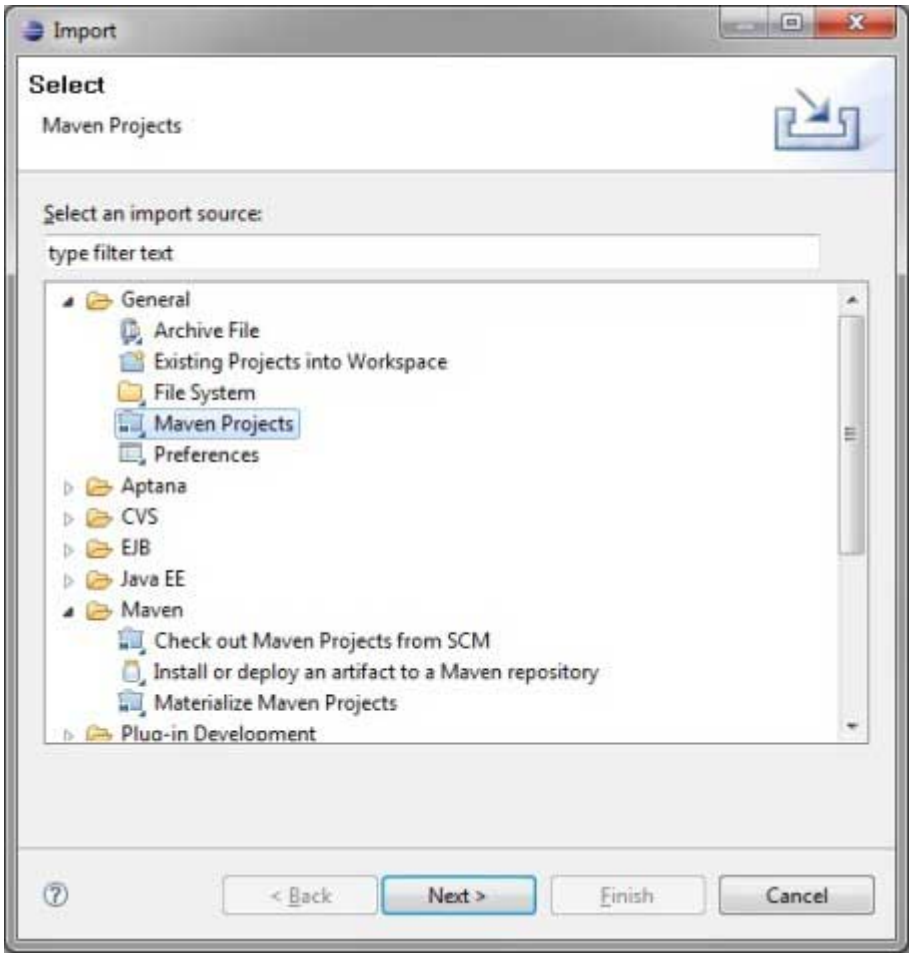
使用以下任意一个链接来安装 m2eclipse:

Eclipse	URL
Eclipse 3.5 (Gallileo)	Installing m2eclipse in Eclipse 3.5 (Gallileo) (http://books.sonatype.com/m2eclipse-book/reference/ch02s03.html)
Eclipse 3.6 (Helios)	Installing m2eclipse in Eclipse 3.6 (Helios) (http://books.sonatype.com/m2eclipse-book/reference/install-sect-marketplace.html)

以下的示例可以帮助你有效地利用集成 Eclipse 和 Maven.

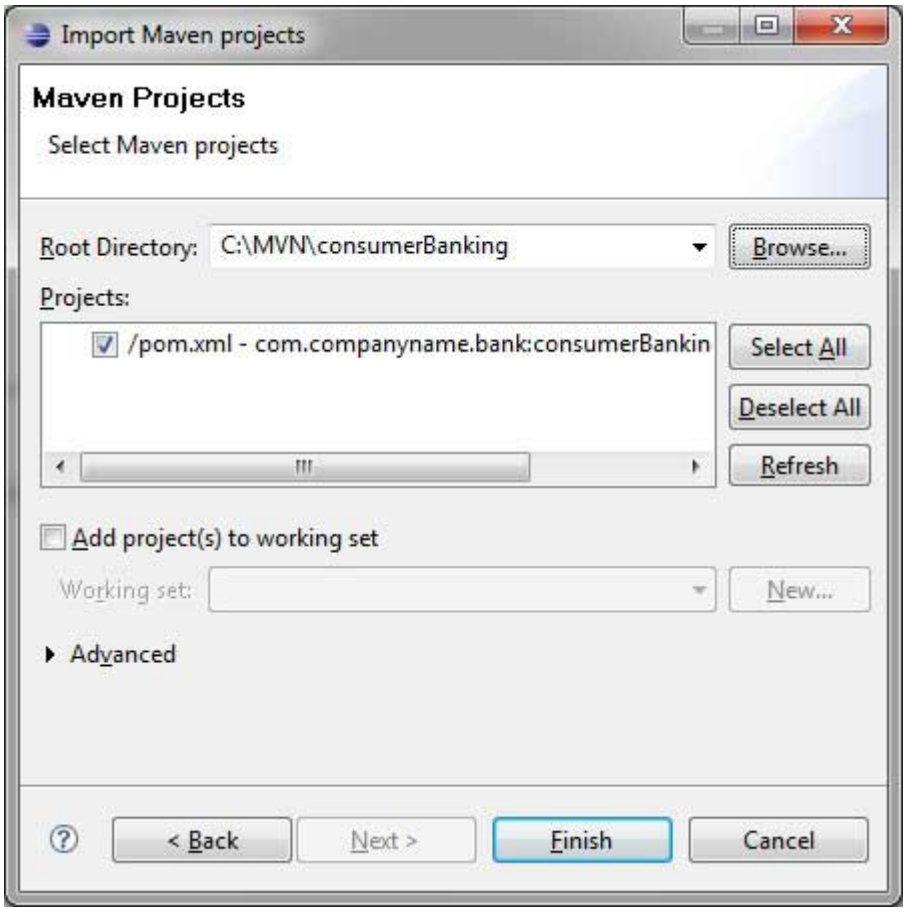
在 Eclipse 中导入一个 Maven 的工程

- 打开 Eclipse.
- 选择 File > Import > option.
- 选择 Maven Projects 选项。点击 Next 按钮。



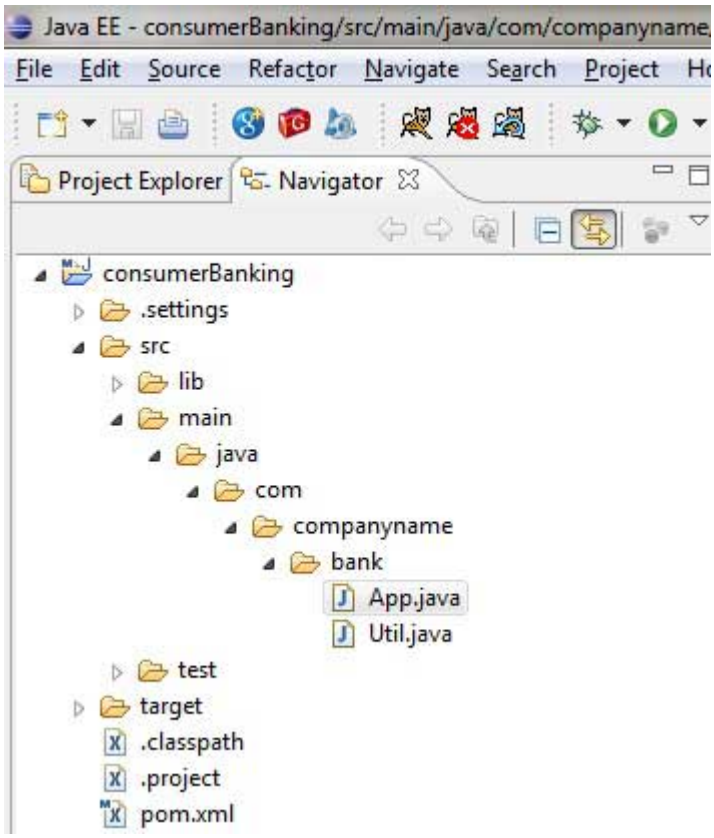
图片 18.1 Import a maven project in Eclipse.

- 选择工程的路径，即使用 Maven 创建一个工程时的存储路径。假设我们创建了一个工程： consumerBanking. 通过 Maven – 创建工程 () 查看如何使用 Maven 创建一个工程。
- 点击 Finish 按钮。



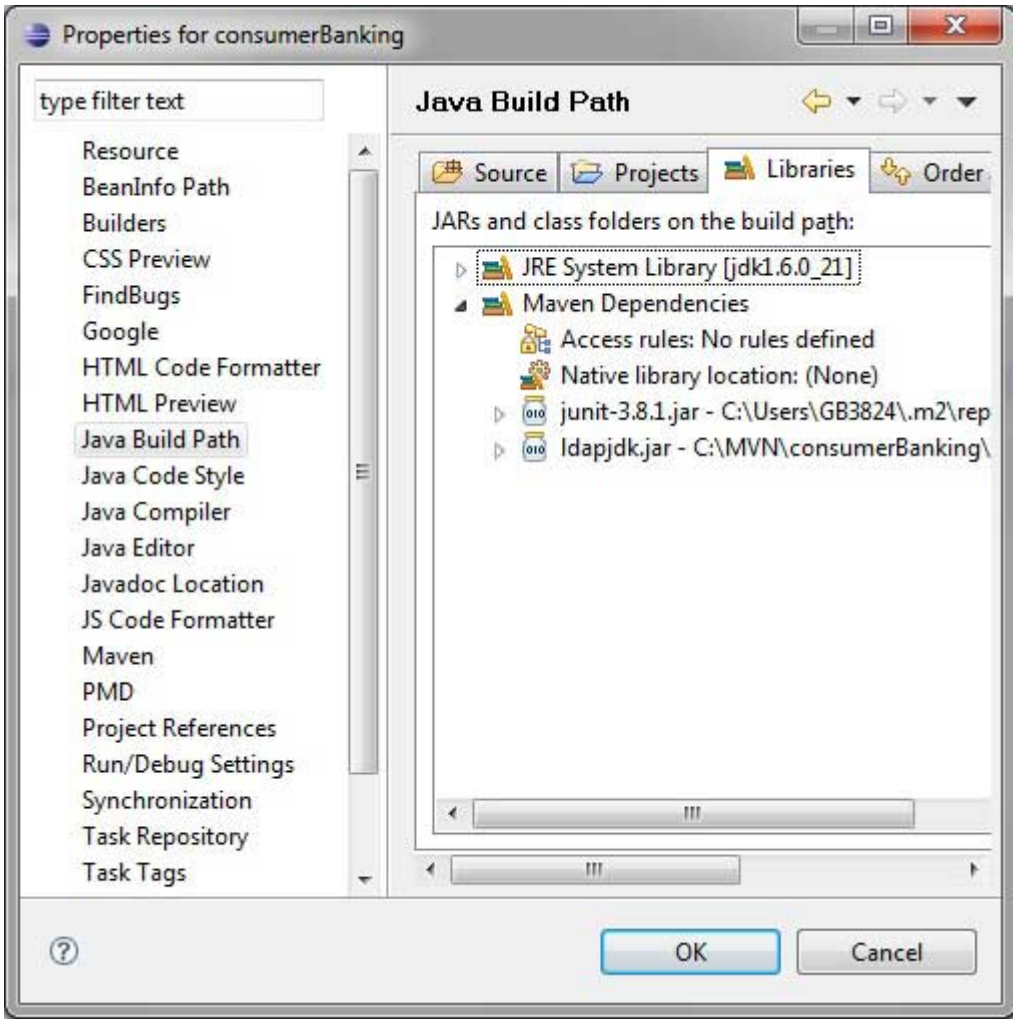
图片 18.2 Import a maven project in Eclipse.

现在，你可以在 Eclipse 中看到 Maven 工程。



图片 18.3 maven project in Eclipse.

看一下 consumerBanking 工程的属性，你可以发现 Eclipse 已经将 Maven 所依赖的都添加到了它的构建路径里了。



图片 18.4 Java Build Path having Maven dependencies.

好了，我们来使用 Eclipse 的编译功能来构建这个 Maven 工程。

- 右键打开 consumerBanking 项目的上下文菜单
- 选择 Run 选项
- 然后选择 maven package 选项

Maven 开始构建工程，你可以在 Eclispe 的控制台看到输出日志。

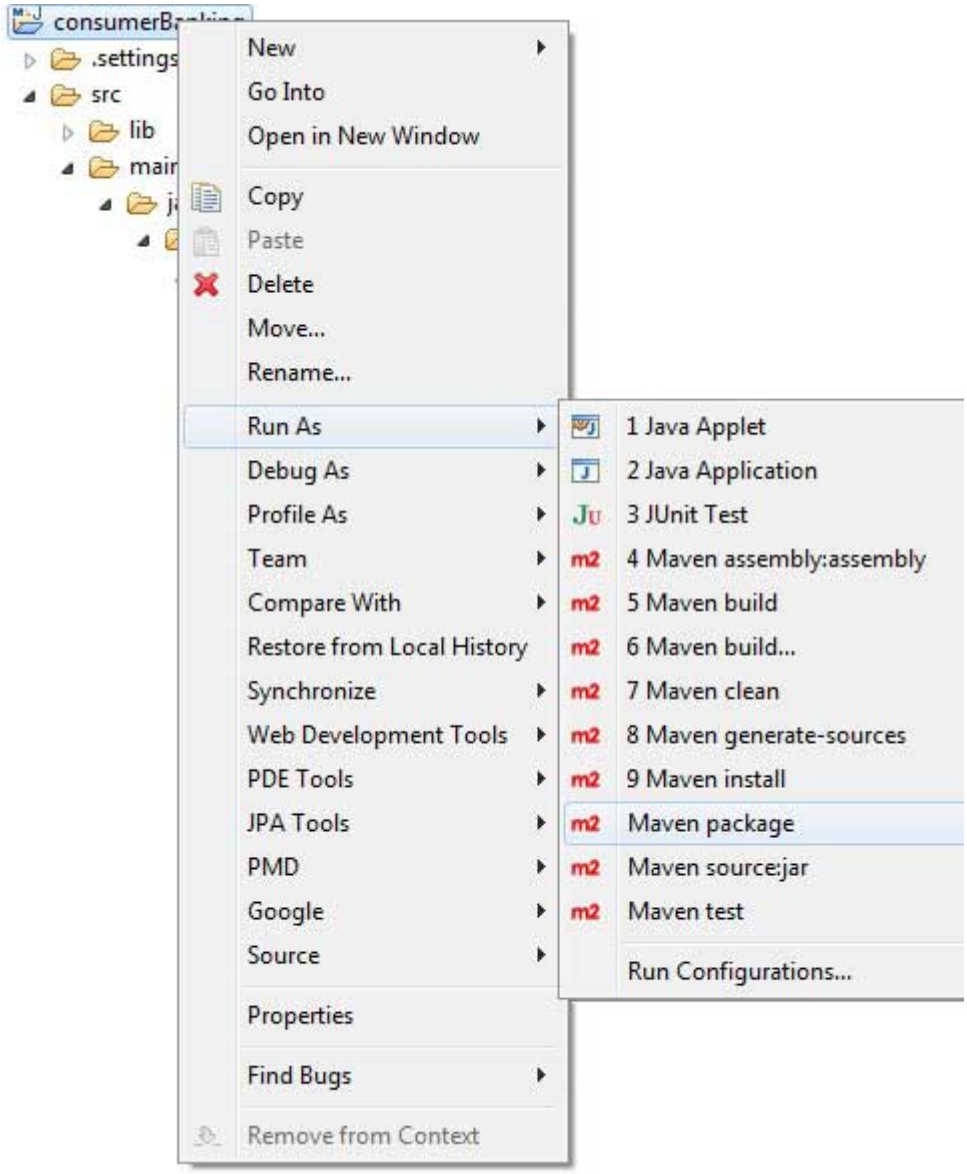
```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO]
[INFO] Id: com.companyname.bank:consumerBanking:jar:1.0-SNAPSHOT
[INFO] task-segment: [package]
[INFO] ------------------------------------------------------------------------
[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] Surefire report directory:
C:\MVN\consumerBanking\target\surefire-reports
##  ##  T E S T SRunning com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec

Results :

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

[INFO] [jar:jar]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Thu Jul 12 18:18:24 IST 2012
[INFO] Final Memory: 2M/15M
[INFO] ------------------------------------------------------------------------
```



图片 18.5 Run maven command using run as optio

现在，右键点击 App.java. 选择 Run As 选项。选择 As Java App.

你将看到如下结果：

```
Hello World!
```

19

Maven – NetBeans

NetBeans 6.7 版本或者更新的版本针对 Maven 支持内部构建功能。 针对之前的版本，可以在插件管理器中找到 Maven 插件。在本例中，我们使用 NetBeans 6.9 版本。

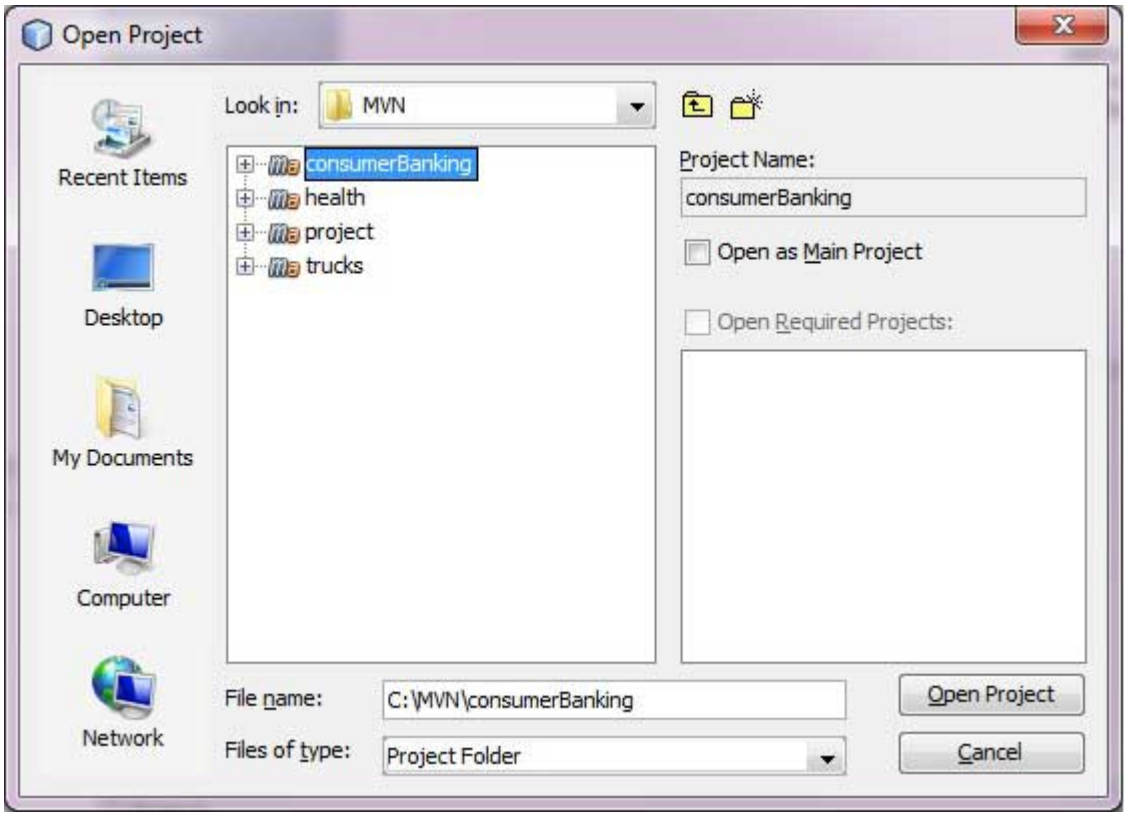
关于 NetBeans 的一些特性如下：

- 可以通过 NetBeans 来运行 Maven 目标。
- 可以在 NetBeans 自己的终端里查看 Maven 命令的输出结果。
- 可以更新 Maven 与 IDE 的依赖。
- 可以在 NetBeans 中启动 Maven 的构建。
- NetBeans 基于 Maven 的 pom.xml 来实现自动化管理依赖关系。
- NetBeans 可以通过自己的工作区解决 Maven 的依赖问题，而无需安装到本地的 Maven 仓库，虽然需要依赖的工程在同一个工作区。
- NetBeans 可以自动从远程 Moven 库上下载需要的依赖和源码。
- NetBeans 提供了创建 Maven 工程，pom.xml 文件的向导。
- NetBeans 提供了 关于Maven 仓库的浏览器，使您可以查看本地存储库和注册在外部的 Maven 仓库。

下面的例子将会帮助你更加充分的认识集成的 NetBeans 和 Maven 的优势。

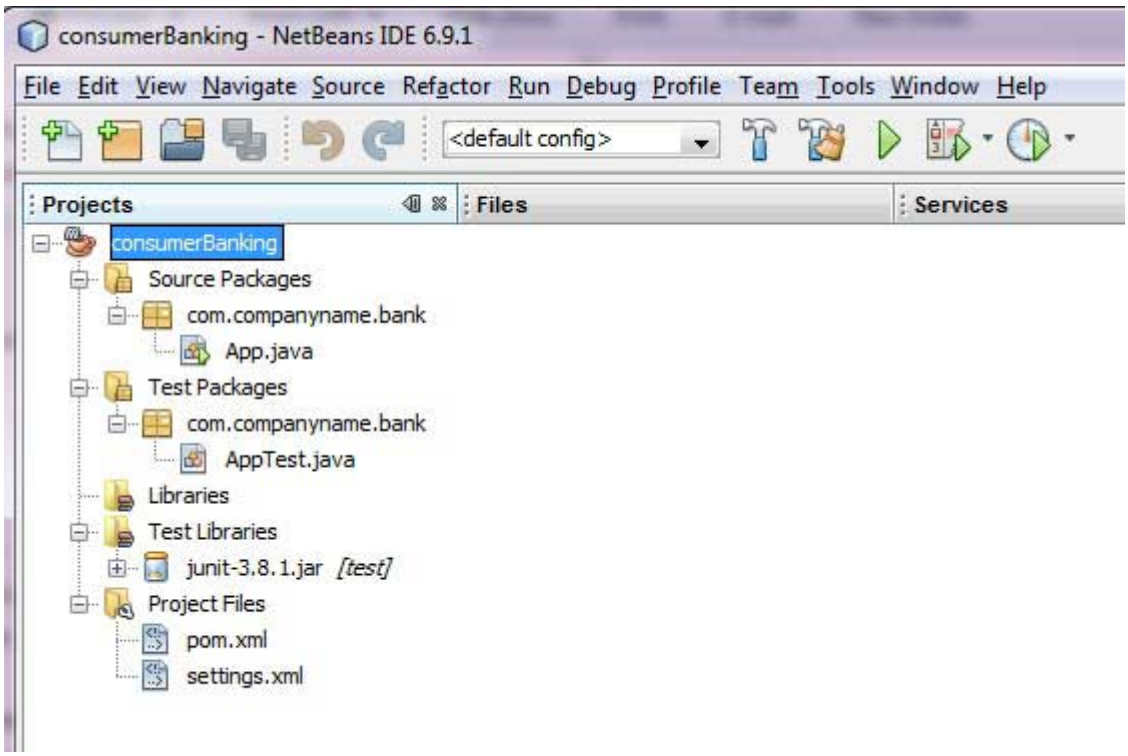
在 NetBeans 里打开一个 Maven 工程

- 打开 NetBeans.
- 选择 File Menu > Open Project 选项.
- 选择工程的路径，即使用 Maven 创建一个工程时的存储路径。假设我们创建了一个工程： consumerBanking. 通过 Maven – 创建工程 () 查看如何使用 Maven 创建一个工程。



图片 19.1 Open a Maven project in NetBeans.

目前为止，你已经可以在 NetBeans 里看到 Maven 工程了。看一下consumerBanking 工程的 Libraries 和 Test Libraries. 你可以发现 NetBeans 已经将 Maven 所依赖的都添加到了它的构建路径里了。

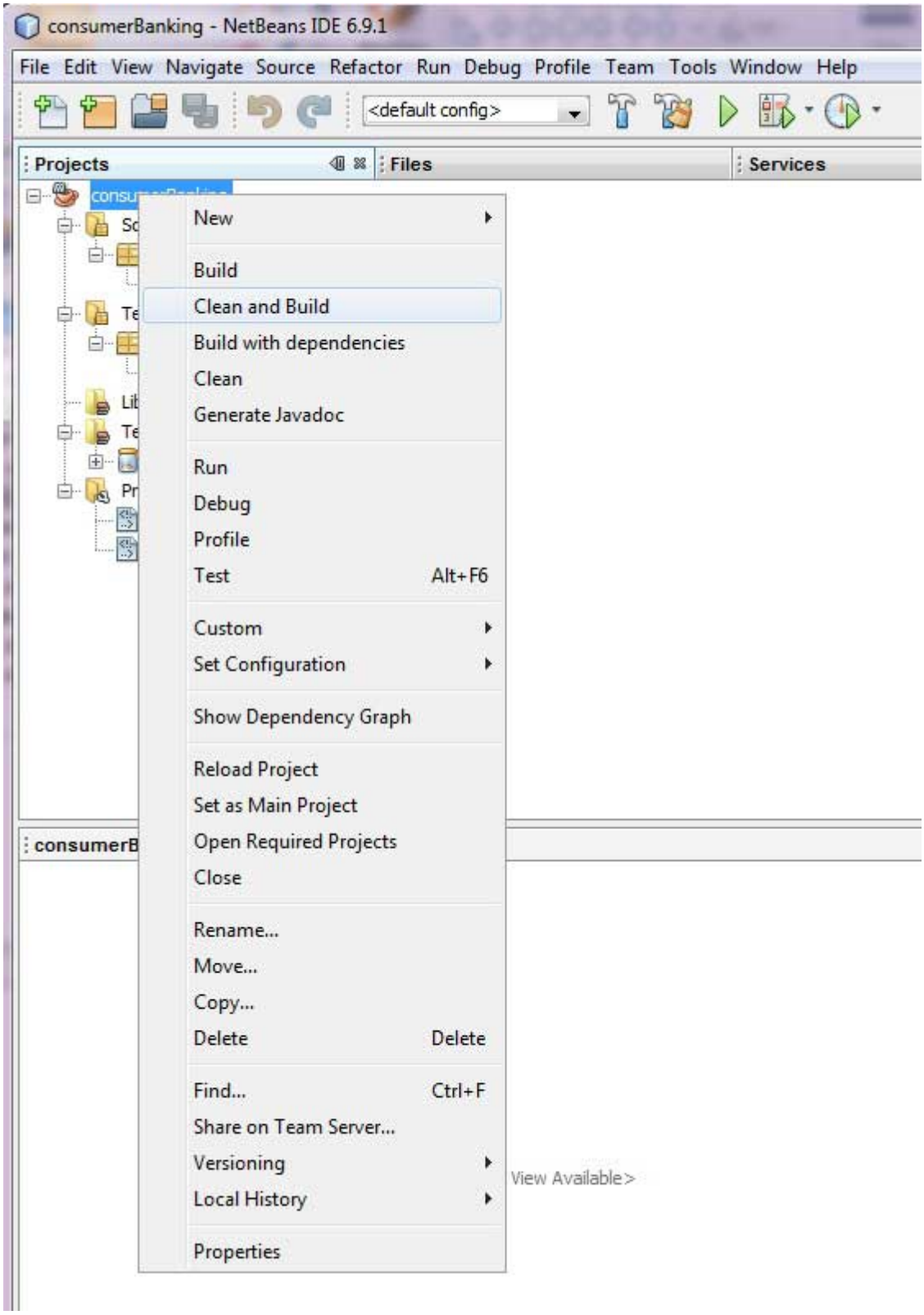


图片 19.2 Maven project in NetBeans.

在 NetBeans 里构建一个 Maven 工程

好了，我们来使用 NetBeans 的编译功能来构建这个 Maven 工程

- 右键点击 consumerBanking 工程打开上下文菜单。
- 选择 "Clean and Build" 选项



图片 19.3 Build a Maven project in NetBeans.

Maven 将会开始构建该工程。你可以在 NetBeans 的终端里查看输出的 log：

```
NetBeans: Executing 'mvn.bat -Dnetbeans.execution=true clean install'
NetBeans:      JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21
## Scanning for projects...Building consumerBanking
##    task-segment: [clean, install][clean:clean]
[resources:resources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources
[compiler:compile]
Compiling 2 source files to C:\MVN\consumerBanking\target\classes
[resources:testResources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\test\resources
[compiler:testCompile]
Compiling 1 source file to C:\MVN\consumerBanking\target\test-classes
[surefire:test]
Surefire report directory: C:\MVN\consumerBanking\target\surefire-reports
##  ##  T E S T SRunning com.companyname.bank.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 sec

Results :

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

[jar:jar]
Building jar: C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
[install:install]
Installing C:\MVN\consumerBanking\target\consumerBanking-1.0-SNAPSHOT.jar
to C:\Users\GB3824\.m2\repository\com\companyname\bank\consumerBanking\
## 1.0-SNAPSHOT\consumerBanking-1.0-SNAPSHOT.jar## BUILD SUCCESSFULTotal time: 9 seconds
Finished at: Thu Jul 19 12:57:28 IST 2012
Final Memory: 16M/85M
------------------------------------------------------------------------
```

在 NetBeans 里运行应用程序

现在，右键点击 App.java 文件。选择 **Run File** 选项。你可以在终端看到如下结果：

```
NetBeans: Executing 'mvn.bat -Dexec.classpathScope=runtime
-Dexec.args=-classpath %classpath com.companyname.bank.App
-Dexec.executable=C:\Program Files\Java\jdk1.6.0_21\bin\java.exe
-Dnetbeans.execution=true process-classes
org.codehaus.mojo:exec-maven-plugin:1.1.1:exec'
```

```
NetBeans:     JAVA_HOME=C:\Program Files\Java\jdk1.6.0_21
## Scanning for projects...Building consumerBanking
   task-segment: [process-classes,
##    org.codehaus.mojo:exec-maven-plugin:1.1.1:exec][resources:resources]
[WARNING] Using platform encoding (Cp1252 actually)
to copy filtered resources, i.e. build is platform dependent!
skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources
[compiler:compile]
Nothing to compile - all classes are up to date
[exec:exec]
## Hello World!## BUILD SUCCESSFULTotal time: 1 second
Finished at: Thu Jul 19 14:18:13 IST 2012
Final Memory: 7M/64M
------------------------------------------------------------------------
```

20

Maven – IntelliJ IDEA

IntelliJ IDEA 针对 Maven 支持内部构建功能。 在本例中，我们使用 IntelliJ IDEA Community Edition 11.1 的版本。

关于 IntelliJ IDEA 的一些特性如下：

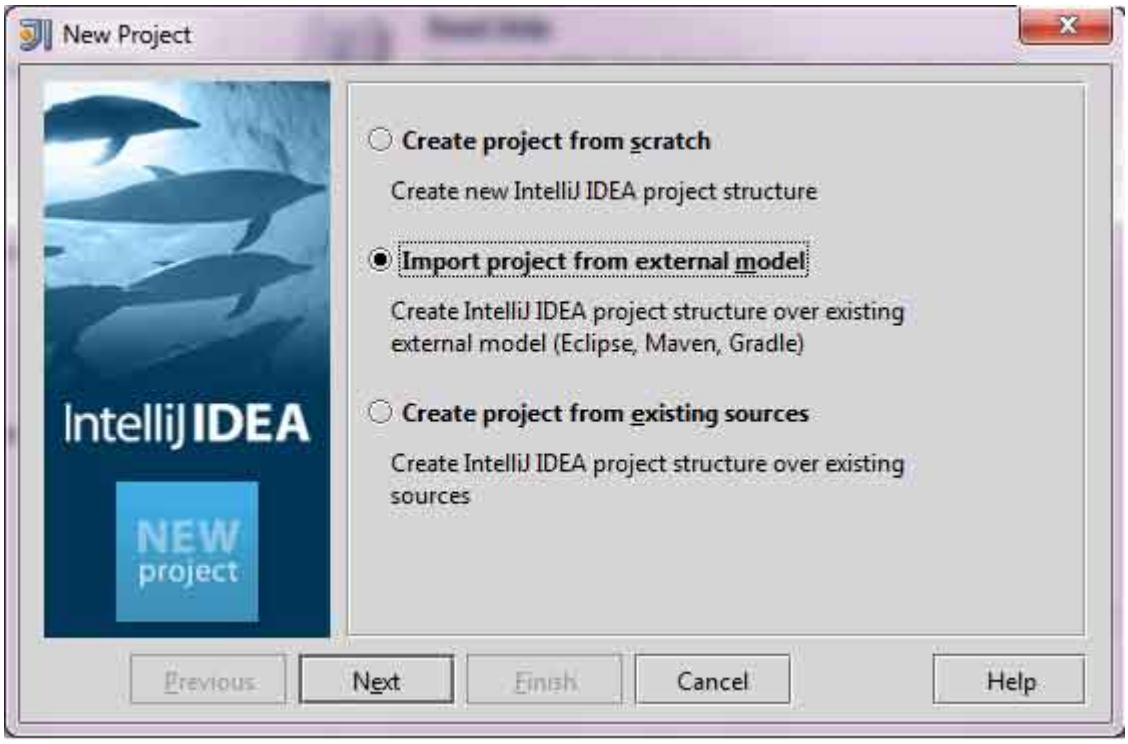
- 可以通过 IntelliJ IDEA 来运行 Maven 目标。
- 可以在 IntelliJ IDEA 自己的终端里查看 Maven 命令的输出结果。
- 可以在 IDE 里更新 Maven 的依赖关系。
- 可以在 IntelliJ IDEA 中启动 Maven 的构建。
- IntelliJ IDEA 基于 Maven 的 pom.xml 来实现自动化管理依赖关系。
- IntelliJ IDEA 可以通过自己的工作区解决 Maven 的依赖问题，而无需安装到本地的 Maven 仓库，虽然需要依赖的工程在同一个工作区。
- IntelliJ IDEA 可以自动从远程 Moven 仓库上下载需要的依赖和源码。
- IntelliJ IDEA 提供了创建 Maven 工程，pom.xml 文件的向导。

下面的例子将会帮助你更加充分的认识集成的 IntelliJ IDEA 和 Maven 的优势。

在 IntelliJ IDEA 里创建一个新的工程

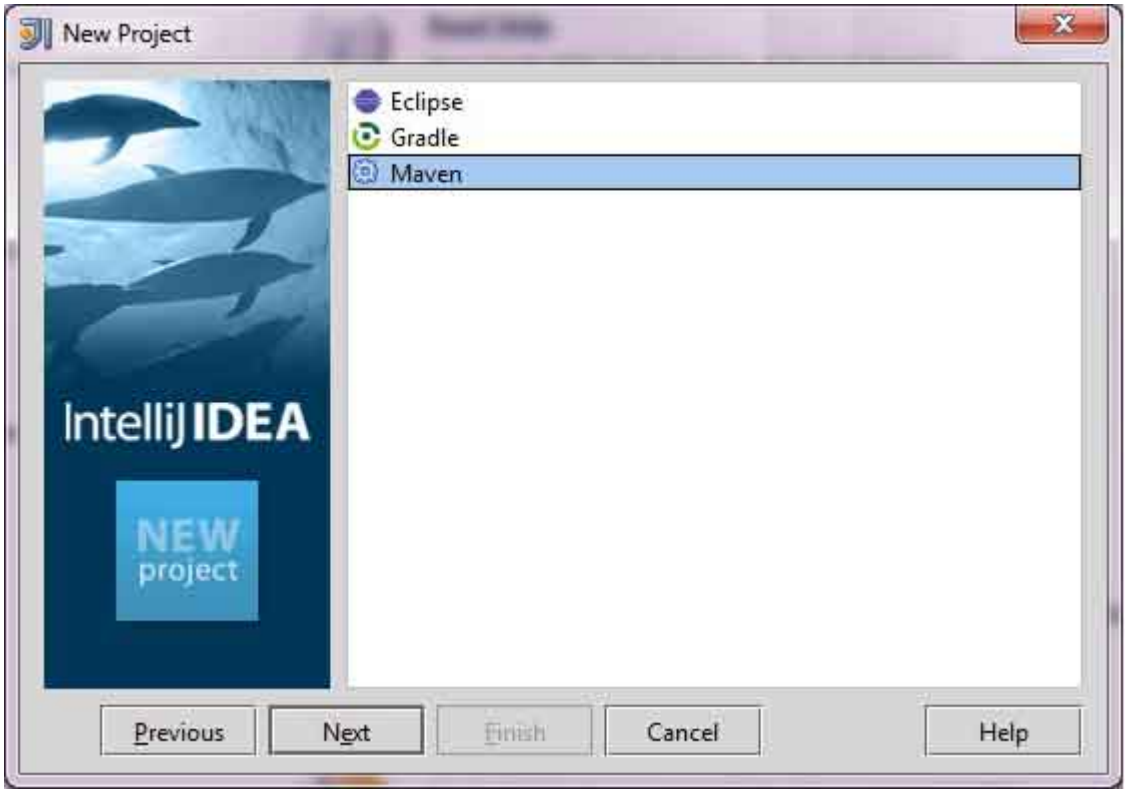
我们将会使用新建工程向导来导入一个 Maven 工程。

- 打开 IntelliJ IDEA.
- 选择 File Menu > New Project 选项。
- 选择 import project from existing model 选项。



图片 20.1 New Project in IntelliJ IDEA, step 1.

- 选择 Maven 选项。

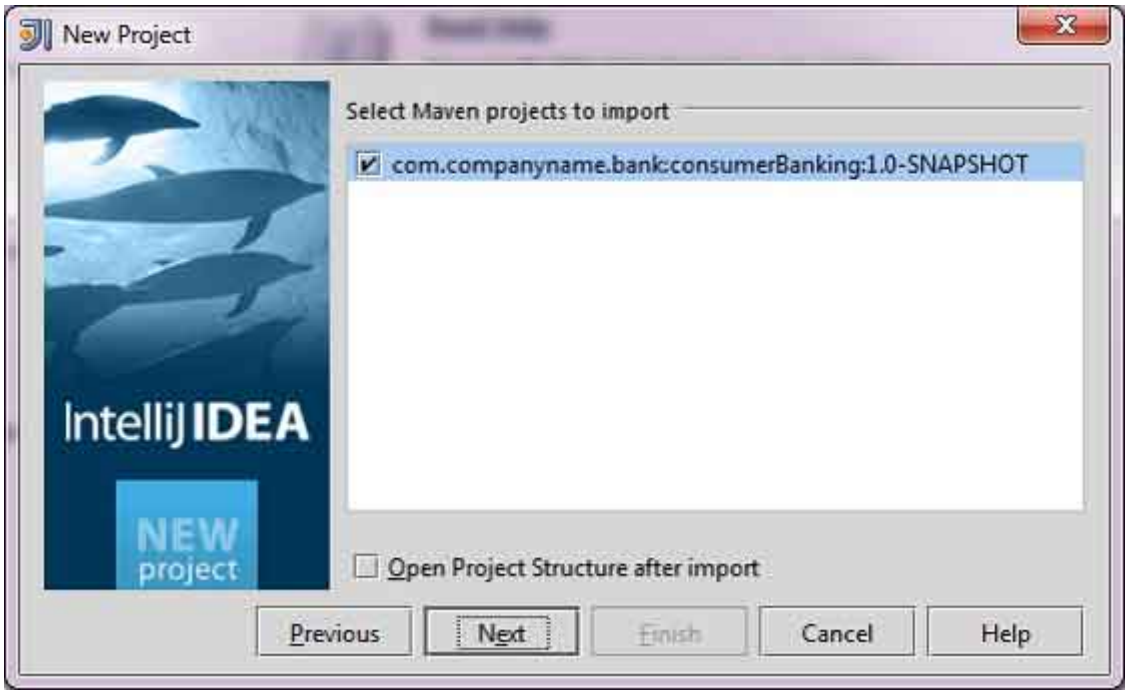


图片 20.2 New Project in IntelliJ IDEA, step 2.

- 选择工程路径，即使用 Maven 创建一个工程时的存储路径。 假设我们创建了一个工程： consumerBanking. 通过 Maven – 创建工程 () 查看如何使用 Maven 创建一个工程。

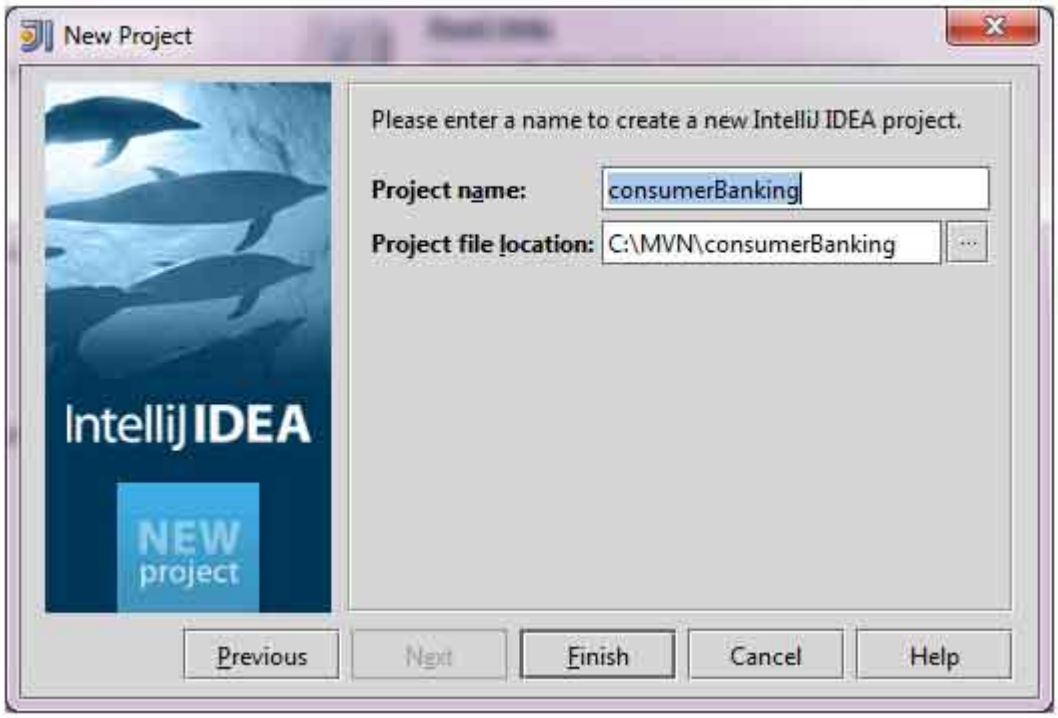
图片 20.3 New Project in IntelliJ IDEA, step 3.

- 选择要导入的 Maven 工程。



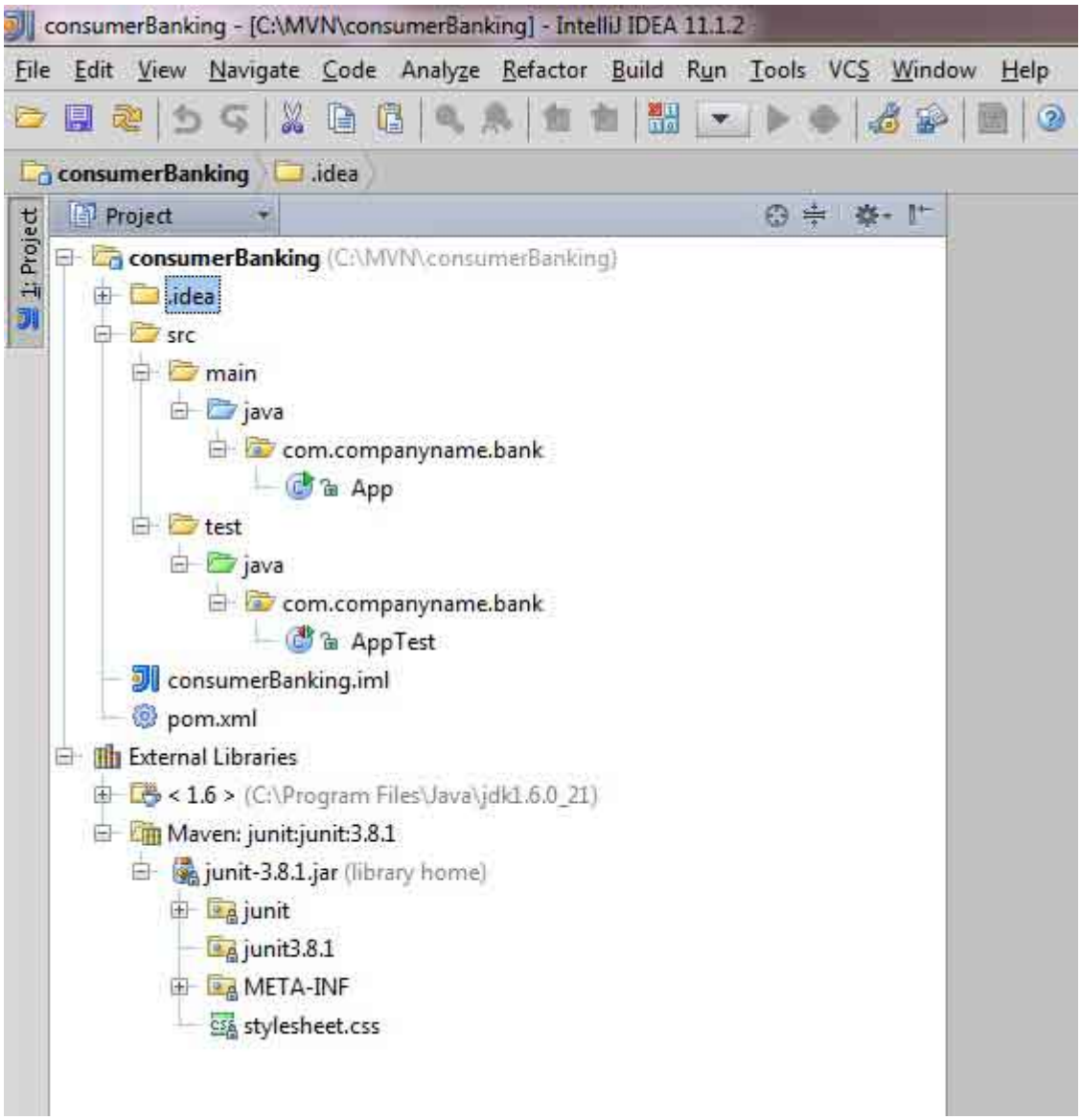
图片 20.4 New Project in IntelliJ IDEA, step 4.

- 输入工程名称，点击 "finish".



图片 20.5 New Project in IntelliJ IDEA, step 5.

目前为止，你已经可以在 IntelliJ IDEA 里看到 Maven 工程了。看一下 consumerBanking 工程的 Libraries 和 Test Libraries. 你可以发现 IntelliJ IDEA 已经将 Maven 所依赖的都添加到了它的构建路径里了。



图片 20.6 Maven project in IntelliJ IDEA.

在 IntelliJ IDEA 里构建一个 Maven 工程

好了，我们来使用 IntelliJ IDEA 的编译功能来构建这个 Maven 工程 。

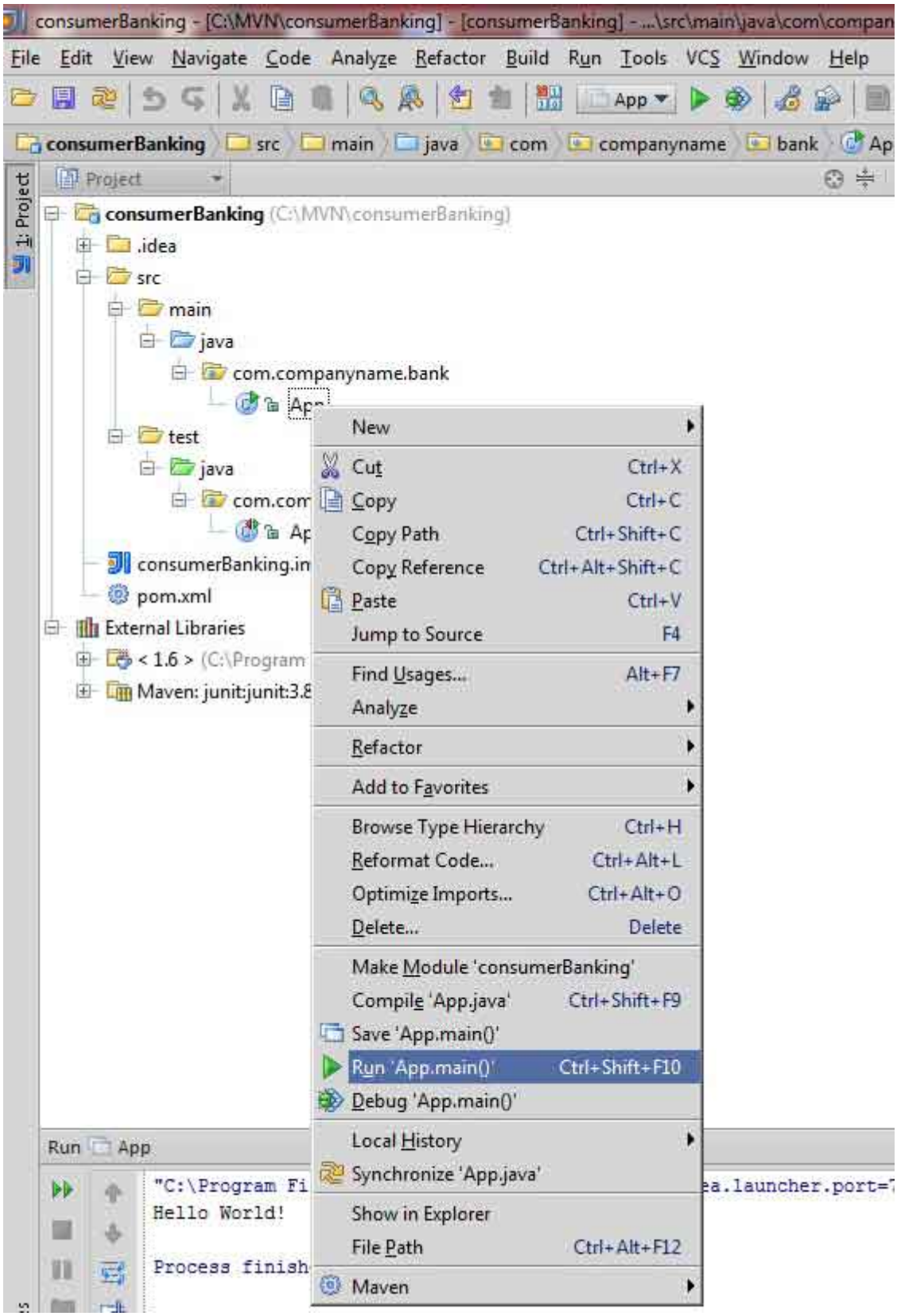
- 选中 consumerBanking 工程。
- 选择 Buid menu > Rebuild Project 选项。

你可以在 IntelliJ IDEA 的终端里看到构建过程输出的log：

```
4:01:56 PM Compilation completed successfully
```

在 IntelliJ IDEA 里运行应用程序

- 选中 consumerBanking 工程。
- 右键点击 App.java 弹出上下文菜单。
- 选择 Run App.main() .



图片 20.7 Maven project in IntelliJ IDEA.

你将会在 IntelliJ IDEA 的终端下看到运行结果的输出。

```
"C:\Program Files\Java\jdk1.6.0_21\bin\java"
-Didea.launcher.port=7533
"-Didea.launcher.bin.path=
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 11.1.2\bin"
-Dfile.encoding=UTF-8
-classpath "C:\Program Files\Java\jdk1.6.0_21\jre\lib\charsets.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\deploy.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\javaws.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jce.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jsse.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\management-agent.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\plugin.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\resources.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\rt.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\dnsns.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\localedata.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunjce_provider.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunmscapi.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunpkcs11.jar
C:\MVN\consumerBanking\target\classes;
C:\Program Files\JetBrains\
IntelliJ IDEA Community Edition 11.1.2\lib\idea_rt.jar"
com.intellij.rt.execution.application.AppMain com.companyname.bank.App
Hello World!
Process finished with exit code 0
```

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/maven/