第 3 章

Maven 使用入门

本章内容

- ❑ 编写 POM
- ❑ 编写主代码
- ❑ 编写测试代码
- ❑ 打包和运行
- ❑ 使用 Archetype 生成项目骨架
- ❑ m2eclipse 简单使用
- ❑ NetBeans Maven 插件简单使用
- ❑ 小结

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

3.1 编写 POM

就像 Make 的 Makefile、Ant 的 build.xml 一样，Maven 项目的核心是 pom.xml。POM（Project Object Model，项目对象模型）定义了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。现在先为 Hello World 项目编写一个最简单的 pom.xml。

首先创建一个名为 hello-world 的文件夹，打开该文件夹，新建一个名为 pom.xml 的文件，输入其内容，如代码清单 3-1 所示。

代码清单 3-1 Hello World 的 POM

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

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

根元素下的第一个子元素 modelVersion 指定了当前 POM 模型的版本，对于 Maven 2 及 Maven 3 来说，它只能是 4.0.0。

这段代码中最重要的是包含 groupId、artifactId 和 version 的三行。这三个元素定义了一个项目基本的坐标，在 Maven 的世界，任何的 jar、pom 或者 war 都是以基于这些基本的坐标进行区分的。

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

artifactId 定义了当前 Maven 项目在组中唯一的 ID，我们为这个 Hello World 项目定义 ar-

tifactId为hello-world，本书其他章节代码会分配其他的artifactId。而在前面的groupId为com.googlecode.myapp的例子中，你可能会为不同的子项目（模块）分配artifacIc，如myapp-util、myapp-domain、myapp-web等。

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

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

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

3.2 编写主代码

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

代码清单3-2 Hello World的主代码

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

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

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

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

关于该Java代码有两点需要注意。首先，在绝大多数情况下，应该把项目主代码放到

src/main/java/目录下（遵循 Maven 的约定），而无须额外的配置，Maven 会自动搜寻该目录找到项目主代码。其次，该 Java 类的包名是 com. juvenxu. mvnbook. helloworld，这与之前在 POM 中定义的 groupId 和 artifactId 相吻合。一般来说，项目中 Java 类的包都应该基于项目的 groupId 和 artifactId，这样更加清晰，更加符合逻辑，也方便搜索构件或者 Java 类。

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

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

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

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

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

3.3　编写测试代码

为了使项目结构保持清晰，主代码与测试代码应该分别位于独立的目录中。3.2 节讲过

Maven项目中默认的主代码目录是src/main/java，对应地，Maven项目中默认的测试代码目录是src/test/java。因此，在编写测试用例之前，应当先创建该目录。

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

代码清单3-3 为Hello World的POM添加依赖

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

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

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

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

代码清单 3-4 Hello World 的测试代码

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

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

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

        String result = helloWorld.sayHello();

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

一个典型的单元测试包含三个步骤：① 准备测试类及数据；② 执行要测试的行为；③ 检查结果。上述样例首先初始化了一个要测试的 HelloWorld 实例，接着执行该实例的 sayHello( ) 方法并保存结果到 result 变量中，最后使用 JUnit 框架的 Assert 类检查结果是否为我们期望的“Hello Maven”。在 JUnit 3 中，约定所有需要执行测试的方法都以 test 开头，这里使用了 JUnit 4，但仍然遵循这一约定。在 JUnit 4 中，需要执行的测试方法都应该以@ Test 进行标注。

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

```
[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Hello World Project
[INFO]    task-segment: [clean, test]
[INFO]
------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory D:\git-juven\mvnbook\code\hello-world\target
[INFO] [resources:resources {execution: default-resources}]
...
Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.pom
1K downloaded (junit-4.7.pom)
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
...
Downloading: http://repo1.maven.org/maven2/junit/junit/4.7/junit-4.7.jar
226K downloaded (junit-4.7.jar)
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to D:\code\hello-world\target\test-classes
[INFO]
------------------------------------------------------------------------
[ERROR] BUILD FAILURE
```

```
[INFO]
------------------------------------------------------------------------
[INFO] Compilation failure
D:\code\hello-world\src\test\java\com\juvenxu\mvnbook\helloworld\Hel-
loWorldTest.java:[8,5] -source 1.3 中不支持注释
(请使用-source 5 或更高版本以启用注释)
    @ Test
[INFO]
------------------------------------------------------------------------
[INFO] For more information, run Maven with the-e switch
...
```

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

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

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

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

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

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

```
...
[INFO] [compiler:testCompile {execution: default-testCompile}]
```

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

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

3.4 打包和运行

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

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

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

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

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

```
...
[INFO] [jar:jar {execution: default-jar}]
```

```
[INFO] Building jar: D: \code\hello-world\target\hello-world-1.0-SNAPSHOT.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing D:\code\hello-world\target\hello-world-1.0-SNAPSHOT.jar to
C:\Users\juven\.m2\repository\com\juvenxu\mvnbook\hello-world\1.0-SNAPSHOT\
hello-world-1.0-SNAPSHOT.jar
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
...
```

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

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

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

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

plugin 元素在 POM 中的相对位置应该在 <project> <build> <plugins> 下面。我们配置了 mainClass 为 com. juvenxu. mvnbook. helloworld. HelloWorld，项目在打包时会将该信息放到

MANIFEST 中。现在执行 mvn clean install，待构建完成之后打开 target/目录，可以看到 hello-world-1.0-SNAPSHOT.jar 和 original-hello-world-1.0-SNAPSHOT.jar，前者是带有 Main-Class 信息的可运行 jar，后者是原始的 jar，打开 hello-world-1.0-SNAPSHOT.jar 的 META-INF/MANIFEST.MF，可以看到它包含这样一行信息：

```
Main-Class: com.juvenxu.mvnbook.helloworld.HelloWorld
```

现在，在项目根目录中执行该 jar 文件：

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

控制台输出为 Hello Maven，这正是我们所期望的。

本小节介绍了 Hello World 项目，侧重点是 Maven 而非 Java 代码本身，介绍了 POM、Maven 项目结构以及如何编译、测试、打包等。

3.5 使用 Archetype 生成项目骨架

Hello World 项目中有一些 Maven 的约定：在项目的根目录中放置 pom.xml，在 src/main/java 目录中放置项目的主代码，在 src/test/java 中放置项目的测试代码。之所以一步一步地展示这些步骤，是为了能让可能是 Maven 初学者的你得到最实际的感受。我们称这些基本的目录结构和 pom.xml 文件内容称为项目的骨架，当第一次创建项目骨架的时候，你还会饶有兴趣地去体会这些默认约定背后的思想，第二次，第三次，你也许还会满意自己的熟练程度，但第四、第五次做同样的事情，你可能就会恼火了。为此 Maven 提供了 Archetype 以帮助我们快速勾勒出项目骨架。

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

如果是 Maven 3，简单地运行：

```
mvn archetype:generate
```

如果是 Maven 2，最好运行如下命令：

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

很多资料会让你直接使用更为简单的 mvn archetype:generate 命令，但在 Maven 2 中这是不安全的，因为该命令没有指定 Archetype 插件的版本，于是 Maven 会自动去下载最新的版本，进而可能得到不稳定的 SNAPSHOT 版本，导致运行失败。然而在 Maven 3 中，即使用户没有指定版本，Maven 也只会解析最新的稳定版本，因此这是安全的。具体内容见 7.7 节。

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

紧接着会看到一段长长的输出，有很多可用的 Archetype 供选择，包括著名的 Appfuse

项目的Archetype、JPA项目的Archetype等。每一个Archetype前面都会对应有一个编号，同时命令行会提示一个默认的编号，其对应的Archetype为maven-archetype-quickstart，直接回车以选择该Archetype，紧接着Maven会提示输入要创建项目的groupId、artifactId、version以及包名package。如下输入并确认：

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

Archetype插件将根据我们提供的信息创建项目骨架。在当前目录下，Archetype插件会创建一个名为hello-world（我们定义的artifactId）的子目录，从中可以看到项目的基本结构：基本的pom.xml已经被创建，里面包含了必要的信息以及一个junit依赖；主代码目录src/main/java已经被创建，在该目录下还有一个Java类com.juvenxu.mvnbook.helloworld.App，注意这里使用到了刚才定义的包名，而这个类也仅仅只有一个简单的输出Hello World!的main方法；测试代码目录src/test/java也被创建好了，并且包含了一个测试用例com.juvenxu.mvnbook.helloworld.AppTest。

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

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

3.6 m2eclipse简单使用

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

3.6.1 导入Maven项目

第2章介绍了如何安装m2eclipse，现在，使用m2ecilpse导入Hello World项目。选择菜单项File，然后选择Import，我们会看到一个Import对话框。在该对话框中选择General目录下的Maven Projects，然后单击Next按钮，就会出现Import Projects对话框。在该对话框中单击Browse按钮选择Hello World的根目录（即包含pom.xml文件的那个目录），这时对话框中的Projects：部分就会显示该目录包含的Maven项目，如图3-1所示。

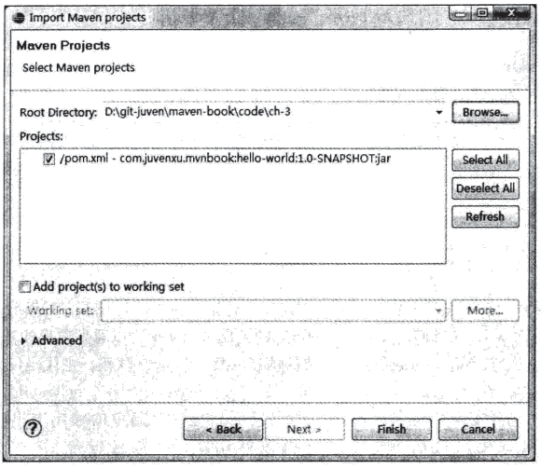

图 3-1 在 Eclipse 中导入 Maven 项目

单击 Finish 按钮之后，m2ecilpse 就会将该项目导入到当前的 workspace 中，导入完成之后，就可以在 Package Explorer 视图中看到图 3-2 所示的项目结构。

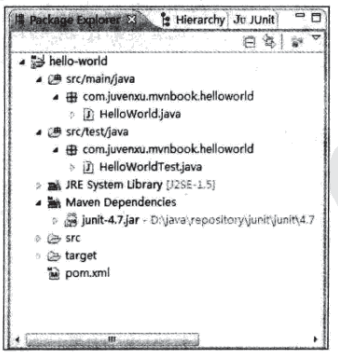

图 3-2 Eclipse 中导入的 Maven 项目结构

我们看到主代码目录 src/main/java 和测试代码目录 src/test/java 成了 Eclipse 中的资源目录，包和类的结构也十分清晰。当然 pom. xml 永远在项目的根目录下，而从这个视图中甚至

还能看到项目的依赖 junit-4.7.jar，其实际的位置指向了 Maven 本地仓库（这里自定义了 Maven 本地仓库地址为 D:\java\repository。后续章节会介绍如何自定义本地仓库位置）。

3.6.2 创建 Maven 项目

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

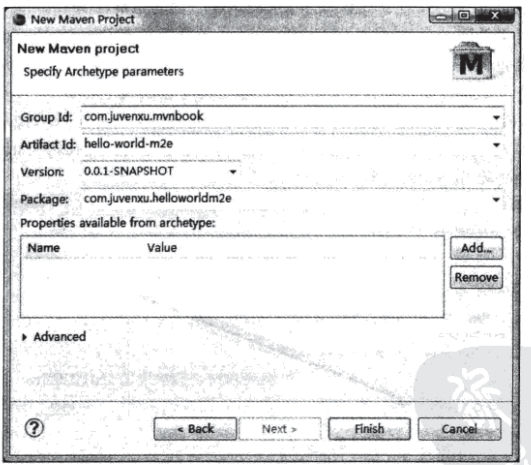

图 3-3 在 Eclipse 中使用 Archetype 创建项目

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

3.6.3 运行 mvn 命令

我们需要在命令行输入如 mvn clean install 之类的命令来执行 maven 构建，m2eclipse 中也有对应的功能。在 Maven 项目或者 pom.xml 上右击，再在弹出的快捷菜单中选择 Run As，就能看到常见的 Maven 命令，如图 3-4 所示。

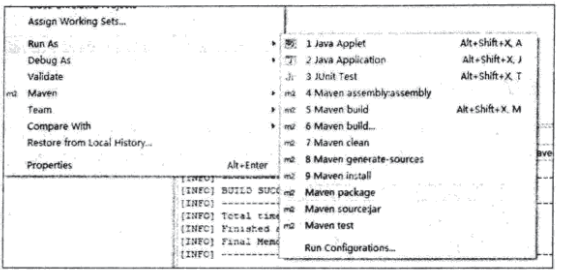

图 3-4　在 Eclipse 中运行默认 mvn 命令

选择想要执行的 Maven 命令就能执行相应的构建，同时也能在 Eclipse 的 console 中看到构建输出。这里常见的一个问题是，默认选项中没有我们想要执行的 Maven 命令怎么办？比如，默认带有 mvn test，但我们想执行 mvn clean test，很简单，选择 Maven build 以自定义 Maven 运行命令，在弹出对话框的 Goals 一项中输入我们想要执行的命令，如 clean test，设置一下 Name，单击 Run 即可。并且，下一次我们选择 Maven build，或者使用快捷键“Alt + Shift + X，M”快速执行 Maven 构建的时候，上次的配置直接就能在历史记录中找到。图 3-5 所示就是自定义 Maven 运行命令的界面。

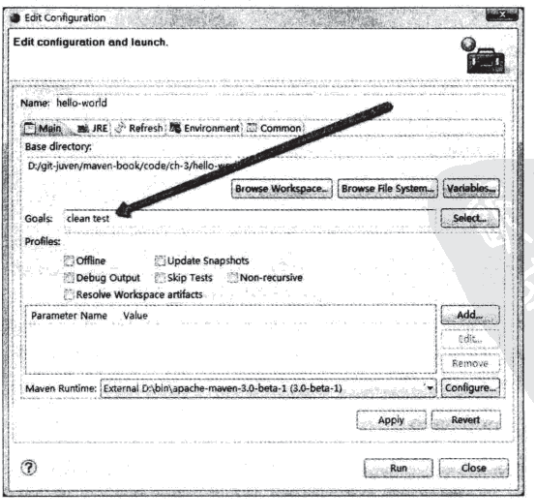

图 3-5　在 Eclipse 中自定义 mvn 命令

3.7　NetBeans Maven 插件简单使用

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

3.7.1　打开 Maven 项目

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

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

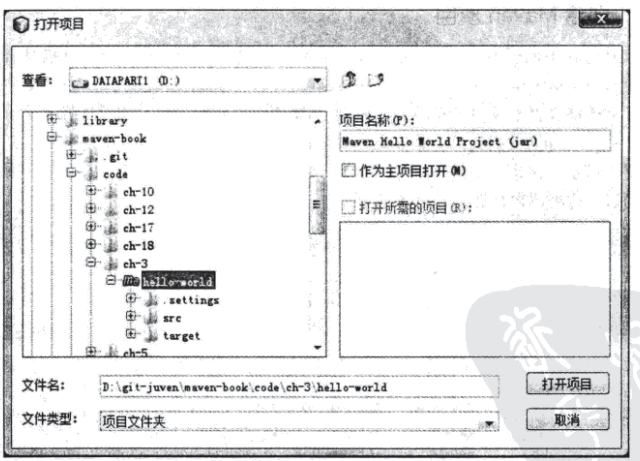

图 3-6　在 NetBeans 中导入 Maven 项目

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

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

图 3-7 NetBeans 中导入的 Maven 项目结构

3.7.2 创建 Maven 项目

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

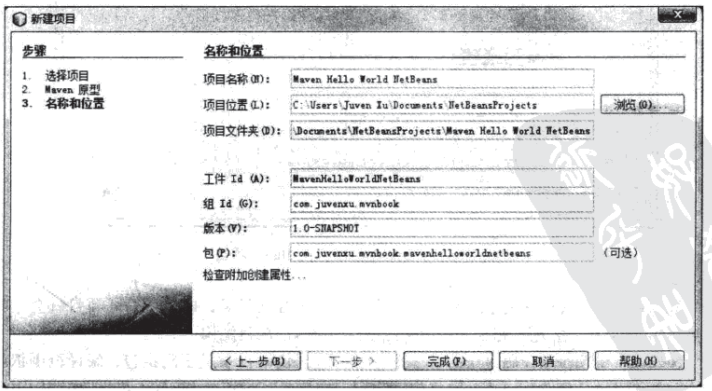

图 3-8 在 NetBeans 中使用 Archetype 创建 Maven 项目

单击“完成”按钮之后，一个新的 Maven 项目就创建好了。

3.7.3　运行 mvn 命令

NetBeans 在默认情况下提供两种 Maven 运行方式，单击菜单栏中的运行，可以看到生成项目和清理并生成项目两个选项。可以尝试“点击运行 Maven 构建”，根据 NetBeans 控制台的输出，就能发现它们实际上对应了 mvn install 和 mvn clean install 两个命令。

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

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

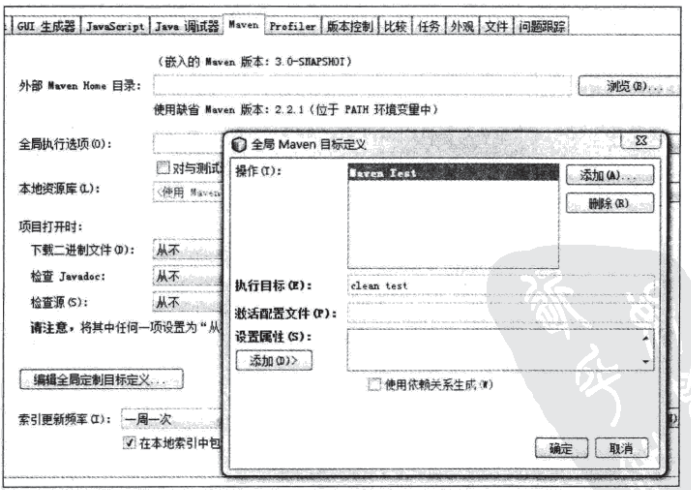

图 3-9　在 NetBeans 中自定义 mvn 命令

单击“缺省保存该配置”，在 Maven 项目上右击，选择定制，就能看到刚才配置好的 Maven 运行操作。选择 Maven Test 之后，终端将执行 mvn clean test。值得一提的是，也可以在项目上右击，选择定制，再选择目标，再输入想要执行的 Maven 目标（如 clean pack-

age)，单击“确定”按钮之后 NetBeans 就会执行相应的 Maven 命令。这种方式十分便捷，但这是临时的，该配置不会被保存，也不会有历史记录。

3.8 小结

本章以尽可能简单且详细的方式叙述了一个 Hello World 项目，重点解释了 POM 的基本内容、Maven 项目的基本结构以及构建项目基本的 Maven 命令。在此基础上，还介绍了如何使用 Archetype 快速创建项目骨架。最后讲述的是如何在 Eclipse 和 NetBeans 中导入、创建及构建 Maven 项目。