第 7 章

生命周期和插件

本章内容

- ❑ 何为生命周期
- ❑ 生命周期详解
- ❑ 插件目标
- ❑ 插件绑定
- ❑ 插件配置
- ❑ 获取插件信息
- ❑ 从命令行调用插件
- ❑ 插件解析机制
- ❑ 小结

除了坐标、依赖以及仓库之外，Maven 另外两个核心概念是生命周期和插件。在有关 Maven 的日常使用中，命令行的输入往往就对应了生命周期，如 mvn package 就表示执行默认生命周期阶段 package。Maven 的生命周期是抽象的，其实际行为都由插件来完成，如 package 阶段的任务可能就会由 maven-jar-plugin 完成。生命周期和插件两者协同工作，密不可分，本章对它们进行深入介绍。

7.1　何为生命周期

在 Maven 出现之前，项目构建的生命周期就已经存在，软件开发人员每天都在对项目进行清理、编译、测试及部署。虽然大家都在不停地做构建工作，但公司和公司间、项目和项目间，往往使用不同的方式做类似的工作。有的项目以手工的方式在执行编译测试，有的项目写了自动化脚本执行编译测试。可以想象的是，虽然各种手工方式十分类似，但不可能完全一样；同样地，对于自动化脚本，大家也是各写各的，能满足自身需求即可，换个项目就需要重头再来。

Maven 的生命周期就是为了对所有的构建过程进行抽象和统一。Maven 从大量项目和构建工具中学习和反思，然后总结了一套高度完善的、易扩展的生命周期。这个生命周期包含了项目的清理、初始化、编译、测试、打包、集成测试、验证、部署和站点生成等几乎所有构建步骤。也就是说，几乎所有项目的构建，都能映射到这样一个生命周期上。

Maven 的生命周期是抽象的，这意味着生命周期本身不做任何实际的工作，在 Maven 的设计中，实际的任务（如编译源代码）都交由插件来完成。这种思想与设计模式中的模板方法（Template Method）非常相似。模板方法模式在父类中定义算法的整体结构，子类可以通过实现或者重写父类的方法来控制实际的行为，这样既保证了算法有足够的可扩展性，又能够严格控制算法的整体结构。如下的模板方法抽象类能够很好地体现 Maven 生命周期的概念，见代码清单 7-1。

代码清单 7-1　模拟生命周期的模板方法抽象类

```
package com.juvenxu.mvnbook.template.method;

public abstract class AbstractBuild
{
    public void build()
    {
        initialize();
        compile();
        test();
        packagee();
        integrationTest();
        deploy();
    }

    protected abstract void initialize();
```

```
    protected abstract void compile();

    protected abstract void test();

    protected abstract void packagee();

    protected abstract void integrationTest();

    protected abstract void deploy();
}
```

这段代码非常简单，build()方法定义了整个构建的过程，依次初始化、编译、测试、打包（由于 package 与 Java 关键字冲突，这里使用了单词 packagee）、集成测试和部署，但是这个类中没有具体实现初始化、编译、测试等行为，它们都交由子类去实现。

虽然上述代码和 Maven 实际代码相去甚远，Maven 的生命周期包含更多的步骤和更复杂的逻辑，但它们的基本理念是相同的。生命周期抽象了构建的各个步骤，定义了它们的次序，但没有提供具体实现。那么谁来实现这些步骤呢？不能让用户为了编译而写一堆代码，为了测试又写一堆代码，那不就成了大家在重复发明轮子吗？Maven 当然必须考虑这一点，因此它设计了插件机制。每个构建步骤都可以绑定一个或者多个插件行为，而且 Maven 为大多数构建步骤编写并绑定了默认插件。例如，针对编译的插件有 maven-compiler-plugin，针对测试的插件有 maven-surefire-plugin 等。虽然在大多数时间里，用户几乎都不会觉察到插件的存在，但实际上编译是由 maven-compiler-plugin 完成的，而测试是由 maven-surefire-plugin 完成的。当用户有特殊需要的时候，也可以配置插件定制构建行为，甚至自己编写插件。生命周期和插件的关系如图 7-1 所示。

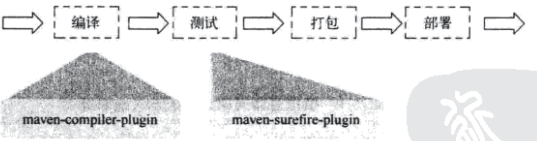

图 7-1 生命周期和插件的关系

Maven 定义的生命周期和插件机制一方面保证了所有 Maven 项目有一致的构建标准，另一方面又通过默认插件简化和稳定了实际项目的构建。此外，该机制还提供了足够的扩展空间，用户可以通过配置现有插件或者自行编写插件来自定义构建行为。

7.2 生命周期详解

到目前为止，本书只是介绍了 Maven 生命周期背后的指导思想，要想熟练地使用 Maven，还必须详细了解其生命周期的具体定义和使用方式。

7.2.1　三套生命周期

初学者往往会以为 Maven 的生命周期是一个整体，其实不然，Maven 拥有三套相互独立的生命周期，它们分别为 clean、default 和 site。clean 生命周期的目的是清理项目，default 生命周期的目的是构建项目，而 site 生命周期的目的是建立项目站点。

每个生命周期包含一些阶段（phase），这些阶段是有顺序的，并且后面的阶段依赖于前面的阶段，用户和 Maven 最直接的交互方式就是调用这些生命周期阶段。以 clean 生命周期为例，它包含的阶段有 pre-clean、clean 和 post-clean。当用户调用 pre-clean 的时候，只有 pre-clean 阶段得以执行；当用户调用 clean 的时候，pre-clean 和 clean 阶段会得以顺序执行；当用户调用 post-clean 的时候，pre-clean、clean 和 post-clean 会得以顺序执行。

较之于生命周期阶段的前后依赖关系，三套生命周期本身是相互独立的，用户可以仅仅调用 clean 生命周期的某个阶段，或者仅仅调用 default 生命周期的某个阶段，而不会对其他生命周期产生任何影响。例如，当用户调用 clean 生命周期的 clean 阶段的时候，不会触发 default 生命周期的任何阶段，反之亦然，当用户调用 default 生命周期的 compile 阶段的时候，也不会触发 clean 生命周期的任何阶段。

7.2.2　clean 生命周期

clean 生命周期的目的是清理项目，它包含三个阶段：

1）**pre-clean** 执行一些清理前需要完成的工作。

2）**clean** 清理上一次构建生成的文件。

3）**post-clean** 执行一些清理后需要完成的工作。

7.2.3　default 生命周期

default 生命周期定义了真正构建时所需要执行的所有步骤，它是所有生命周期中最核心的部分，其包含的阶段如下，这里笔者只对重要的阶段进行解释：

- **validate**
- **initialize**
- **generate-sources**
- **process-sources** 处理项目主资源文件。一般来说，是对 src/main/resources 目录的内容进行变量替换等工作后，复制到项目输出的主 classpath 目录中。
- **generate-resources**
- **process-resources**
- **compile** 编译项目的主源码。一般来说，是编译 src/main/java 目录下的 Java 文件至项目输出的主 classpath 目录中。
- **process-classes**
- **generate-test-sources**

- **process-test-sources** 处理项目测试资源文件。一般来说，是对 src/test/resources 目录的内容进行变量替换等工作后，复制到项目输出的测试 classpath 目录中。
- **generate-test-resources**
- **process-test-resources**
- **test-compile** 编译项目的测试代码。一般来说，是编译 src/test/java 目录下的 Java 文件至项目输出的测试 classpath 目录中。
- **process-test-classes**
- **test** 使用单元测试框架运行测试，测试代码不会被打包或部署。
- **prepare-package**
- **package** 接受编译好的代码，打包成可发布的格式，如 JAR。
- **pre-integration-test**
- **integration-test**
- **post-integration-test**
- **verify**
- **install** 将包安装到 Maven 本地仓库，供本地其他 Maven 项目使用。
- **deploy** 将最终的包复制到远程仓库，供其他开发人员和 Maven 项目使用。

对于上述未加解释的阶段，读者也应该能够根据名字大概猜到其用途，若想了解进一步的这些阶段的详细信息，可以参阅官方的解释：http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html。

7.2.4 site 生命周期

site 生命周期的目的是建立和发布项目站点，Maven 能够基于 POM 所包含的信息，自动生成一个友好的站点，方便团队交流和发布项目信息。该生命周期包含如下阶段：

- **pre-site** 执行一些在生成项目站点之前需要完成的工作。
- **site** 生成项目站点文档。
- **post-site** 执行一些在生成项目站点之后需要完成的工作。
- **site-deploy** 将生成的项目站点发布到服务器上。

7.2.5 命令行与生命周期

从命令行执行 Maven 任务的最主要方式就是调用 Maven 的生命周期阶段。需要注意的是，各个生命周期是相互独立的，而一个生命周期的阶段是有前后依赖关系的。下面以一些常见的 Maven 命令为例，解释其执行的生命周期阶段：

- **$mvn clean**：该命令调用 clean 生命周期的 clean 阶段。实际执行的阶段为 clean 生命周期的 pre-clean 和 clean 阶段。
- **$mvn test**：该命令调用 default 生命周期的 test 阶段。实际执行的阶段为 default 生命周期的 validate、initialize 等，直到 test 的所有阶段。这也解释了为什么在执行测试的

时候，项目的代码能够自动得以编译。

- **$mvn clean install**：该命令调用 clean 生命周期的 clean 阶段和 default 生命周期的 install 阶段。实际执行的阶段为 clean 生命周期的 pre-clean、clean 阶段，以及 default 生命周期的从 validate 至 install 的所有阶段。该命令结合了两个生命周期，在执行真正的项目构建之前清理项目是一个很好的实践。
- **$mvn clean deploy site-deploy**：该命令调用 clean 生命周期的 clean 阶段、default 生命周期的 deploy 阶段，以及 site 生命周期的 site-deploy 阶段。实际执行的阶段为 clean 生命周期的 pre-clean、clean 阶段，default 生命周期的所有阶段，以及 site 生命周期的所有阶段。该命令结合了 Maven 所有三个生命周期，且 deploy 为 default 生命周期的最后一个阶段，site-deploy 为 site 生命周期的最后一个阶段。

由于 Maven 中主要的生命周期阶段并不多，而常用的 Maven 命令实际都是基于这些阶段简单组合而成的，因此只要对 Maven 生命周期有一个基本的理解，读者就可以正确而熟练地使用 Maven 命令。

7.3 插件目标

在进一步详述插件和生命周期的绑定关系之前，必须先了解插件目标（Plugin Goal）的概念。我们知道，Maven 的核心仅仅定义了抽象的生命周期，具体的任务是交由插件完成的，插件以独立的构件形式存在，因此，Maven 核心的分发包只有不到3MB 的大小，Maven 会在需要的时候下载并使用插件。

对于插件本身，为了能够复用代码，它往往能够完成多个任务。例如 maven-dependency-plugin，它能够基于项目依赖做很多事情。它能够分析项目依赖，帮助找出潜在的无用依赖；它能够列出项目的依赖树，帮助分析依赖来源；它能够列出项目所有已解析的依赖，等等。为每个这样的功能编写一个独立的插件显然是不可取的，因为这些任务背后有很多可以复用的代码，因此，这些功能聚集在一个插件里，每个功能就是一个插件目标。

maven-dependency-plugin 有十多个目标，每个目标对应了一个功能，上述提到的几个功能分别对应的插件目标为 dependency:analyze、dependency:tree 和 dependency:list。这是一种通用的写法，冒号前面是插件前缀，冒号后面是该插件的目标。类似地，还可以写出 compiler:compile（这是 maven-compiler-plugin 的 compile 目标）和 surefire:test（这是 maven-surefire-plugin 的 test 目标）。

7.4 插件绑定

Maven 的生命周期与插件相互绑定，用以完成实际的构建任务。具体而言，是生命周期的阶段与插件的目标相互绑定，以完成某个具体的构建任务。例如项目编译这一任务，它对应了 default 生命周期的 compile 这一阶段，而 maven-compiler-plugin 这一插件的 compile

目标能够完成该任务。因此，将它们绑定，就能实现项目编译的目的，如图 7-2 所示。

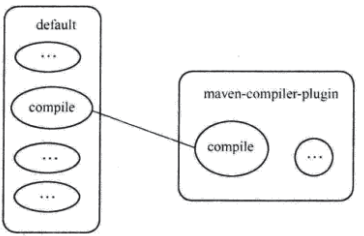

图 7-2 生命周期阶段与插件目标绑定

7.4.1 内置绑定

为了能让用户几乎不用任何配置就能构建 Maven 项目，Maven 在核心为一些主要的生命周期阶段绑定了很多插件的目标，当用户通过命令行调用生命周期阶段的时候，对应的插件目标就会执行相应的任务。

clean 生命周期仅有 pre-clean、clean 和 post-clean 三个阶段，其中的 clean 与 maven-clean-plugin:clean 绑定。maven-clean-plugin 仅有 clean 这一个目标，其作用就是删除项目的输出目录。clean 生命周期阶段与插件目标的绑定关系如表 7-1 所示。

site 生命周期有 pre-site、site、post-site 和 site-deploy 四个阶段，其中，site 和 maven-site-plugin:site 相互绑定，site-deploy 和 maven-site-plugin:depoy 相互绑定。maven-site-plugin 有很多目标，其中，site 目标用来生成项目站点，deploy 目标用来将项目站点部署到远程服务器上。site 生命周期阶段与插件目标的绑定关系如表 7-2 所示。

表 7-1 clean 生命周期阶段与插件目标的绑定关系

生命周期阶段	插件目标
pre-clean	
clean	maven-clean-plugin:clean
post-clean	

表 7-2 site 生命周期阶段与插件目标的绑定关系

生命周期阶段	插件目标
pre-site	
site	maven-site-plugin:site
post-site	
site-deploy	maven-site-plugin:deploy

相对于 clean 和 site 生命周期来说，default 生命周期与插件目标的绑定关系就显得复杂一些。这是因为对于任何项目来说，例如 jar 项目和 war 项目，它们的项目清理和站点生成任务是一样的，不过构建过程会有区别。例如 jar 项目需要打成 JAR 包，而 war 项目需要打成 WAR 包。

由于项目的打包类型会影响构建的具体过程，因此，default 生命周期的阶段与插件目标的绑定关系由项目打包类型所决定，打包类型是通过 POM 中的 packaging 元素定义的，具体可回顾第 5.2 节 。最常见、最重要的打包类型是 jar，它也是默认的打包类型。基于该打包类型的项目，其 default 生命周期的内置插件绑定关系及具体任务如表 7-3 所示。

表 7-3　default 生命周期的内置插件绑定关系及具体任务（打包类型：jar）

生命周期阶段	插件目标	执行任务
process-resources	maven-resources-plugin:resources	复制主资源文件至主输出目录
compile	maven-compiler-plugin:compile	编译主代码至主输出目录
process-test-resources	maven-resources-plugin:testResources	复制测试资源文件至测试输出目录
test-compile	maven-compiler-plugin:testCompile	编译测试代码至测试输出目录
test	maven-surefire-plugin:test	执行测试用例
package	maven-jar-plugin:jar	创建项目 jar 包
install	maven-install-plugin:install	将项目输出构件安装到本地仓库
deploy	maven-deploy-plugin:deploy	将项目输出构件部署到远程仓库

注意，表 7-3 只列出了拥有插件绑定关系的阶段，default 生命周期还有很多其他阶段，默认它们没有绑定任何插件，因此也没有任何实际行为。

除了默认的打包类型 jar 之外，常见的打包类型还有 war、pom、maven-plugin、ear 等。它们的 default 生命周期与插件目标的绑定关系可参阅 Maven 官方文档：http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Built-in_Lifecycle_Bindings，这里不再赘述。

读者可以从 Maven 的命令行输出中看到在项目构建过程执行了哪些插件目标，例如基于 account-email 执行 **mvn clean install** 命令，可以看到如下输出，见代码清单 7-2。

代码清单 7-2　Maven 输出中包含了生命周期阶段与插件的绑定关系

```
    [INFO] ------------------------------------------------------------
    [INFO] Building Account Email 1.0.0-SNAPSHOT
    [INFO] ------------------------------------------------------------
    [INFO]
    [INFO] --- maven-clean-plugin:2.3:clean (default-clean) @ account-email ---
    [INFO] Deleting file set: D:\git-juven\maven-book\code\ch-5\account-email\target…
    …
    [INFO] --- maven-resources-plugin:2.4.1:resources (default-resources) @ account-email ---
    [INFO] Using 'UTF-8' encoding to copy filtered resources.
    …
    [INFO] --- maven-compiler-plugin:2.0.2:compile (default-compile) @ account-email ---
    [INFO] Compiling 3 source files to D:\git-juven\maven-book\code\…
    …
    [INFO] --- maven-resources-plugin:2.4.1:testResources (default-testResources) @ account-email ---
```

```
[INFO] Using 'UTF-8' encoding to copy filtered resources.
…
[INFO] --- maven-compiler-plugin:2.0.2:testCompile (default-testCompile) @ ac-
count-email ---
[INFO] Compiling 1 source file to …
…
[INFO] --- maven-surefire-plugin:2.4.3:test (default-test) @ account-email --
[INFO] Surefire report directory: D:\git-juven\maven-book\code\…
…
[INFO] --- maven-jar-plugin:2.2:jar (default-jar) @ account-email ---
[INFO] Building jar: D:\git-juven\maven-book\code\….
…
[INFO] --- maven-install-plugin:2.3:install (default-install) @ account-email
[INFO] Installing D:\git-juven\maven-book\code\…
…
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
```

从输出中可以看到，执行的插件目标依次为 maven-clean-plugin:clean、maven-resources-plugin:resources、maven-compiler-plugin:compile、maven-resources-plugin:testResources、maven-compiler-plugin:testCompile、maven-surefire-plugin:test、maven-jar-plugin:jar 和 maven-install-plugin:install。我们知道，**mvn clean install** 命令实际调用了 clean 生命周期的 pre-clean、clean 阶段，以及 default 生命周期的从 validate 至 install 所有阶段。在此基础上，通过对照表 7-1 和表 7-3，就能从理论上得到将会执行的插件目标任务，而实际的输出完全验证了这一点。

7.4.2 自定义绑定

除了内置绑定以外，用户还能够自己选择将某个插件目标绑定到生命周期的某个阶段上，这种自定义绑定方式能让 Maven 项目在构建过程中执行更多更富特色的任务。

一个常见的例子是创建项目的源码 jar 包，内置的插件绑定关系中并没有涉及这一任务，因此需要用户自行配置。maven-source-plugin 可以帮助我们完成该任务，它的 jar-no-fork 目标能够将项目的主代码打包成 jar 文件，可以将其绑定到 default 生命周期的 verify 阶段上，在执行完集成测试后和安装构件之前创建源码 jar 包。具体配置见代码清单 7-3。

代码清单 7-3 自定义绑定插件目标

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-source-plugin</artifactId>
            <version>2.1.1</version>
            <executions>
                <execution>
                    <id>attach-sources</id>
                    <phase>verify</phase>
                    <goals>
                        <goal>jar-no-fork</goal>
```

```
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

在 POM 的 build 元素下的 plugins 子元素中声明插件的使用，该例中用到的是 maven-source-plugin，其 groupId 为 org. apache. maven. plugins，这也是 Maven 官方插件的 groupId，紧接着 artifactId 为 maven-source-plugin，version 为 2. 1. 1。对于自定义绑定的插件，用户总是应该声明一个非快照版本，这样可以避免由于插件版本变化造成的构建不稳定性。

上述配置中，除了基本的插件坐标声明外，还有插件执行配置，executions 下每个 execution 子元素可以用来配置执行一个任务。该例中配置了一个 id 为 attach-sources 的任务，通过 phrase 配置，将其绑定到 verify 生命周期阶段上，再通过 goals 配置指定要执行的插件目标。至此，自定义插件绑定完成。运行 **mvn verify** 就能看到如下输出：

```
[INFO] --- maven-source-plugin:2.1.1:jar-no-fork (attach-sources) @ my-proj --
[INFO] Building jar: D:\code\ch-7\target\my-proj-0.0.1-SNAPSHOT-sources.jar
```

我们可以看到，当执行 verify 生命周期阶段的时候，maven-source-plugin：jar-no-fork 会得以执行，它会创建一个以-sources. jar 结尾的源码文件包。

有时候，即使不通过 phase 元素配置生命周期阶段，插件目标也能够绑定到生命周期中去。例如，可以尝试删除上述配置中的 phase 一行，再次执行 **mvn verify**，仍然可以看到 maven-source-plugin:jar-no-fork 得以执行。出现这种现象的原因是：有很多插件的目标在编写时已经定义了默认绑定阶段。可以使用 maven-help-plugin 查看插件详细信息，了解插件目标的默认绑定阶段。运行命令如下：

```
$ mvn help:describe-Dplugin = org.apache.maven.plugins:maven-source-plugin:
2.1.1-Ddetail
```

该命令输出对应插件的详细信息。在输出信息中，能够看到关于目标 jar-no-fork 的如下信息：

```
…
source:jar-no-fork
  Description: This goal bundles all the sources into a jar archive. This
    goal functions the same as the jar goal but does not fork the build and is
    suitable for attaching to the build lifecycle.
  Deprecated. No reason given
  Implementation: org.apache.maven.plugin.source.SourceJarNoForkMojo
  Language: java
  Bound to phase: package

  Available parameters:
…
```

该输出包含了一段关于 jar-no-fork 目标的描述，这里关心的是 Bound to phase 这一项，它表示该目标默认绑定的生命周期阶段（这里是 package）。也就是说，当用户配置使用

maven-source-plugin 的 jar-no-fork 目标的时候，如果不指定 phase 参数，该目标就会被绑定到 package 阶段。

我们知道，当插件目标被绑定到不同的生命周期阶段的时候，其执行顺序会由生命周期阶段的先后顺序决定。如果多个目标被绑定到同一个阶段，它们的执行顺序会是怎样？答案很简单，当多个插件目标绑定到同一个阶段的时候，这些插件声明的先后顺序决定了目标的执行顺序。

7.5 插件配置

完成了插件和生命周期的绑定之后，用户还可以配置插件目标的参数，进一步调整插件目标所执行的任务，以满足项目的需求。几乎所有 Maven 插件的目标都有一些可配置的参数，用户可以通过命令行和 POM 配置等方式来配置这些参数。

7.5.1 命令行插件配置

在日常的 Maven 使用中，我们会经常从命令行输入并执行 Maven 命令。在这种情况下，如果能够方便地更改某些插件的行为，无疑会十分方便。很多插件目标的参数都支持从命令行配置，用户可以在 Maven 命令中使用-D 参数，并伴随一个参数键 = 参数值的形式，来配置插件目标的参数。

例如，maven-surefire-plugin 提供了一个 maven. test. skip 参数，当其值为 true 的时候，就会跳过执行测试。于是，在运行命令的时候，加上如下-D 参数就能跳过测试：

```
$ mvn install-Dmaven.test.skip=true
```

参数-D 是 Java 自带的，其功能是通过命令行设置一个 Java 系统属性，Maven 简单地重用了该参数，在准备插件的时候检查系统属性，便实现了插件参数的配置。

7.5.2 POM 中插件全局配置

并不是所有的插件参数都适合从命令行配置，有些参数的值从项目创建到项目发布都不会改变，或者说很少改变，对于这种情况，在 POM 文件中一次性配置就显然比重复在命令行输入要方便。

用户可以在声明插件的时候，对此插件进行一个全局的配置。也就是说，所有该基于该插件目标的任务，都会使用这些配置。例如，我们通常会需要配置 maven-compiler-plugin 告诉它编译 Java 1. 5 版本的源文件，生成与 JVM 1. 5 兼容的字节码文件，见代码清单 7-4。

代码清单 7-4 在 POM 中对插件进行全局配置

```
<build>
  <plugins>
    <plugin>
```

```
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

这样，不管绑定到 compile 阶段的 maven-compiler-plugin:compile 任务，还是绑定到 test-compiler 阶段的 maven-compiler-plugin:testCompiler 任务，就都能够使用该配置，基于 Java 1.5 版本进行编译。

7.5.3　POM 中插件任务配置

除了为插件配置全局的参数，用户还可以为某个插件任务配置特定的参数。以 maven-antrun-plugin 为例，它有一个目标 run，可以用来在 Maven 中调用 Ant 任务。用户将 maven-antrun-plugin:run 绑定到多个生命周期阶段上，再加以不同的配置，就可以让 Maven 在不同的生命阶段执行不同的任务，见代码清单 7-5。

代码清单 7-5　在 POM 中对插件进行任务配置

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.3</version>
      <executions>
        <execution>
          <id>ant-validate</id>
          <phase>validate</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration>
            <tasks>
              <echo>I'm bound to validate phase.</echo>
            </tasks>
          </configuration>
        </execution>
        <execution>
          <id>ant-verify</id>
          <phase>verify</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration>
            <tasks>
```

```
                    <echo>I'm bound to verify phase.</echo>
                </tasks>
            </configuration>
        </execution>
    </executions>
</plugin>
</plugins>
</build>
```

在上述代码片段中，首先，maven-antrun-plugin:run 与 validate 阶段绑定，从而构成一个 id 为 ant-validate 的任务。插件全局配置中的 configuration 元素位于 plugin 元素下面，而这里的 configuration 元素则位于 execution 元素下，表示这是特定任务的配置，而非插件整体的配置。这个 ant-validate 任务配置了一个 echo Ant 任务，向命令行输出一段文字，表示该任务是绑定到 validate 阶段的。第二个任务的 id 为 ant-verify，它绑定到了 verify 阶段，同样它也输出一段文字到命令行，告诉该任务绑定到了 verify 阶段。

7.6 获取插件信息

仅仅理解如何配置使用插件是不够的。当遇到一个构建任务的时候，用户还需要知道去哪里寻找合适的插件，以帮助完成任务。找到正确的插件之后，还要详细了解该插件的配置点。由于 Maven 的插件非常多，而且这其中的大部分没有完善的文档，因此，使用正确的插件并进行正确的配置，其实并不是一件容易的事。

7.6.1 在线插件信息

基本上所有主要的 Maven 插件都来自 Apache 和 Codehaus。由于 Maven 本身是属于 Apache 软件基金会的，因此它有很多官方的插件，每天都有成千上万的 Maven 用户在使用这些插件，它们具有非常好的稳定性。详细的列表可以在这个地址得到：http://maven.apache.org/plugins/index.html，单击某个插件的链接便可以得到进一步的信息。所有官方插件能在这里下载：http://repo1.maven.org/maven2/org/apache/maven/plugins/。

除了 Apache 上的官方插件之外，托管于 Codehaus 上的 Mojo 项目也提供了大量了 Maven 插件，详细的列表可以访问：http://mojo.codehaus.org/plugins.html。需要注意的是，这些插件的文档和可靠性相对较差，在使用时，如果遇到问题，往往只能自己去看源代码。所有 Codehaus 的 Maven 插件能在这里下载：http://repository.codehaus.org/org/codehaus/mojo/。

由于上述两个站点提供的插件非常多，而实际使用中常用的插件远不会是这个数量，因此附录 C 归纳了一些比较常用的插件。

虽然并非所有插件都提供了完善的文档，但一些核心插件的文档还是非常丰富的。以 maven-surefire-plugin 为例，访问 http://maven.apache.org/plugins/maven-surefire-plugin/可以看到该插件的简要介绍、包含的目标、使用介绍、FAQ 以及很多实例，如图 7-3 所示。

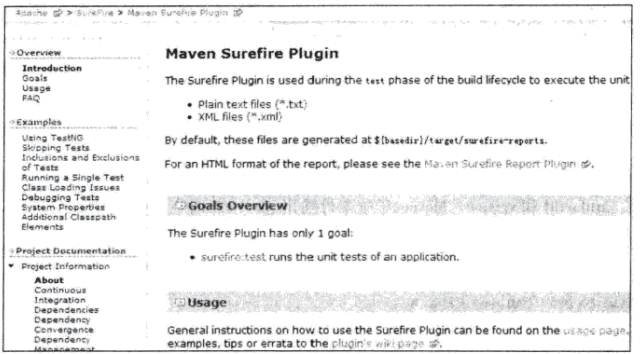

图 7-3　maven-surefire-plugin 的文档页面

一般来说，通过阅读插件文档中的使用介绍和实例，就应该能够在自己的项目中很好地使用该插件。但当我们想了解非常细节的目标参数时，就需要进一步访问该插件每个目标的文档。以 maven-surefire-plugin 为例（见第 7.5.1 节），可以通过在命令行传入 maven. test. skip 参数来跳过测试执行，而执行测试的插件目标是 surefire:test，访问其文档：http://maven. apache. org/plugins/maven-surefire-plugin/test-mojo. html，可以找到目标参数 skip，如图 7-4 所示。

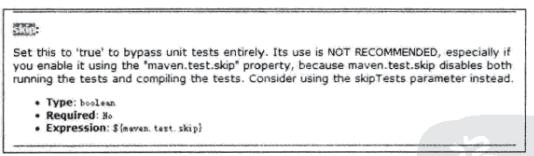

图 7-4　maven-surefire-plugin：test 的 skip 参数

文档详细解释了该参数的作用、类型等信息。基于该信息，用户可以在 POM 中配置 maven-surefire-plugin 的 skip 参数为 true 来跳过测试。这个时候读者可能会不理解了，之前在命令行传入的参数不是 maven. test. skip 吗？的确如此，虽然对于该插件目标的作用是一样的，但从命令行传入的参数确实不同于该插件目标的参数名称。命令行参数是由该插件参数的表达式（Expression）决定的。从图 7-4 中能够看到，surefire:test skip 参数的表达式为 ${maven. test. skip}，它表示可以在命令行以-Dmaven. test. skip = true 的方式配置该目标。并不是所有插件目标参数都有表达式，也就是说，一些插件目标参数只能在 POM 中配置。

7.6.2 使用 maven-help-plugin 描述插件

除了访问在线的插件文档之外，还可以借助 maven-help-plugin 来获取插件的详细信息。可以运行如下命令来获取 maven-compiler-plugin 2.1 版本的信息：

```
$ mvn help:describe -Dplugin = org.apache.maven.plugins:maven-compiler-
plugin:2.1
```

这里执行的是 maven-help-plugin 的 describe 目标，在参数 plugin 中输入需要描述插件的 groupId、artifactId 和 version。Maven 在命令行输出 maven-compiler-plugin 的简要信息，包括该插件的坐标、目标前缀和目标等，见代码清单 7-6。

代码清单 7-6 使用 maven-help-plugin 获取插件信息

```
Name: Maven Compiler Plugin
Description: The Compiler Plugin is used to compile the sources of your
  project.
Group Id: org.apache.maven.plugins
Artifact Id: maven-compiler-plugin
Version: 2.1
Goal Prefix: compiler

This plugin has 3 goals:

compiler:compile
  Description: Compiles application sources

compiler:help
  Description: Display help information on maven-compiler-plugin.
    Call
      mvn compiler:help-Ddetail = true-Dgoal = <goal-name>
    to display parameter details.

compiler:testCompile
  Description: Compiles application test sources.

For more information, run 'mvn help:describe [...]-Ddetail'
```

对于坐标和插件目标，不再多做解释，这里值得一提的是目标前缀（Goal Prefix），其作用是方便在命令行直接运行插件。在第 7.8 节会做进一步解释。maven-compiler-plugin 的目标前缀是 compiler。

在描述插件的时候，还可以省去版本信息，让 Maven 自动获取最新版本来进行表述。例如：

```
$ mvn help:describe-Dplugin = org.apache.maven.plugins:maven-compiler-plugin
```

进一步简化，可以使用插件目标前缀替换坐标。例如：

```
$ mvn help:describe-Dplugin = compiler
```

如果想仅仅描述某个插件目标的信息，可以加上 goal 参数：

```
$ mvn help:describe-Dplugin=compiler-Dgoal=compile
```

如果想让 maven-help-plugin 输出更详细的信息，可以加上 detail 参数：

```
$ mvn help:describe-Dplugin=compiler-Ddetail
```

读者可以在实际环境中使用 help:describe 描述一些常用插件的信息，以得到更加直观的感受。

7.7　从命令行调用插件

如果在命令行运行 **mvn-h** 来显示 mvn 命令帮助，就可以看到如下的信息：

```
usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
...
```

该信息告诉了我们 mvn 命令的基本用法，options 表示可用的选项，mvn 命令有 20 多个选项，这里暂不详述，读者可以根据说明来了解每个选项的作用。除了选项之外，mvn 命令后面可以添加一个或者多个 goal 和 phase，它们分别是指插件目标和生命周期阶段。第 7.2.5 节已经详细介绍了如何通过该参数控制 Maven 的生命周期。现在我们关心的是另外一个参数：goal。

我们知道，可以通过 mvn 命令激活生命周期阶段，从而执行那些绑定在生命周期阶段上的插件目标。但 Maven 还支持直接从命令行调用插件目标。Maven 支持这种方式是因为有些任务不适合绑定在生命周期上，例如 maven-help-plugin:describe，我们不需要在构建项目的时候去描述插件信息，又如 maven-dependency-plugin:tree，我们也不需要在构建项目的时候去显示依赖树。因此这些插件目标应该通过如下方式使用：

```
$ mvn help:describe-Dplugin=compiler

$ mvn dependency:tree
```

不过，这里还有一个疑问，describe 是 maven-help-plugin 的目标没错，但冒号前面的 help 是什么呢？它既不是 groupId，也不是 artifactId，Maven 是如何根据该信息找到对应版本插件的呢？同理，为什么不是 maven-dependency-plugin:tree，而是 dependency:tree？

解答该疑问之前，可以先尝试一下如下的命令：

```
$ mvn org.apache.maven.plugins:maven-help-plugin:2.1:describe-Dplugin=compiler

$ mvn org.apache.maven.plugins:maven-dependency-plugin:2.1:tree
```

这两条命令就比较容易理解了，插件的 groupId、artifactId、version 以及 goal 都得以清晰描述。它们的效果与之前的两条命令基本是一样的，但显然前面的命令更简洁，更容易记忆和使用。为了达到该目的，Maven 引入了目标前缀的概念，help 是 maven-help-plugin 的目标前缀，dependency 是 maven-dependency-plugin 的前缀，有了插件前缀，Maven 就能找到对

应的 artifactId。不过，除了 artifactId，Maven 还需要得到 groupId 和 version 才能精确定位到某个插件。下一节将详细解释这个过程。

7.8 插件解析机制

为了方便用户使用和配置插件，Maven 不需要用户提供完整的插件坐标信息，就可以解析得到正确的插件，Maven 的这一特性是一把双刃剑，虽然它简化了插件的使用和配置，可一旦插件的行为出现异常，用户就很难快速定位到出问题的插件构件。例如 **mvn help:system** 这样一条命令，它到底执行了什么插件？该插件的 groupId、artifactId 和 version 分别是什么？这个构件是从哪里来的？本节就详细介绍 Maven 的运行机制，以让读者不仅知其然，更知其所以然。

7.8.1 插件仓库

与依赖构件一样，插件构件同样基于坐标存储在 Maven 仓库中。在需要的时候，Maven 会从本地仓库寻找插件，如果不存在，则从远程仓库查找。找到插件之后，再下载到本地仓库使用。

值得一提的是，Maven 会区别对待依赖的远程仓库与插件的远程仓库，第 6.4 节介绍了如何配置远程仓库，但那种配置只对一般依赖有效果。当 Maven 需要的依赖在本地仓库不存在时，它会去所配置的远程仓库查找，可是当 Maven 需要的插件在本地仓库不存在时，它就不会去这些远程仓库查找。

不同于 repositories 及其 repository 子元素，插件的远程仓库使用 pluginRepositories 和 pluginRepository 配置。例如，Maven 内置了如下的插件远程仓库配置，见代码清单 7-7。

代码清单 7-7 Maven 内置的插件仓库配置

```
<pluginRepositories>
  <pluginRepository>
    <id>central</id>
    <name>Maven Plugin Repository</name>
    <url>http://repo1.maven.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <updatePolicy>never</updatePolicy>
    </releases>
  </pluginRepository>
</pluginRepositories>
```

除了 pluginRepositories 和 pluginRepository 标签不同之外，其余所有子元素表达的含义与第 6.4 节所介绍的依赖远程仓库配置完全一样。我们甚至看到，这个默认插件仓库的地址就是中央仓库，它关闭了对 SNAPSHOT 的支持，以防止引入 SNAPSHOT 版本的插件而导致

不稳定的构建。

一般来说，中央仓库所包含的插件完全能够满足我们的需要，因此也不需要配置其他的插件仓库。只有在很少的情况下，项目使用的插件无法在中央仓库找到，或者自己编写了插件，这个时候可以参考上述的配置，在 POM 或者 settings.xml 中加入其他的插件仓库配置。

7.8.2 插件的默认 groupId

在 POM 中配置插件的时候，如果该插件是 Maven 的官方插件（即如果其 groupId 为 org.apache.maven.plugins），就可以省略 groupId 配置，见代码清单 7-8。

代码清单 7-8 配置官方插件和省略 groupId

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

上述配置中省略了 maven-compiler-plugin 的 groupId，Maven 在解析该插件的时候，会自动用默认 groupId org.apache.maven.plugins 补齐。

笔者不推荐使用 Maven 的这一机制，虽然这么做可以省略一些配置，但这样的配置会让团队中不熟悉 Maven 的成员感到费解，况且能省略的配置也就仅仅一行而已。

7.8.3 解析插件版本

同样是为了简化插件的配置和使用，在用户没有提供插件版本的情况下，Maven 会自动解析插件版本。

首先，Maven 在超级 POM 中为所有核心插件设定了版本，超级 POM 是所有 Maven 项目的父 POM，所有项目都继承这个超级 POM 的配置，因此，即使用户不加任何配置，Maven 使用核心插件的时候，它们的版本就已经确定了。这些插件包括 maven-clean-plugin、maven-compiler-plugin、maven-surefire-plugin 等。

如果用户使用某个插件时没有设定版本，而这个插件又不属于核心插件的范畴，Maven 就会去检查所有仓库中可用的版本，然后做出选择。读者可以回顾一下第 6.6 节中介绍的仓库元数据 groupId/artifactId/maven-metadata.xml。以 maven-compiler-plugin 为例，它在中央仓库的仓库元数据为 http://repo1.maven.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/maven-metadata.xml，其内容见代码清单 7-9。

代码清单 7-9 maven-compiler-plugin 的 groupId/artifactId 仓库元数据

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <versioning>
    <latest>2.1</latest>
    <release>2.1</release>
    <versions>
      <version>2.0-beta-1</version>
      <version>2.0</version>
      <version>2.0.1</version>
      <version>2.0.2</version>
      <version>2.1</version>
    </versions>
    <lastUpdated>20100102092331</lastUpdated>
  </versioning>
</metadata>
```

Maven 遍历本地仓库和所有远程插件仓库，将该路径下的仓库元数据归并后，就能计算出 latest 和 release 的值。latest 表示所有仓库中该构件的最新版本，而 release 表示最新的非快照版本。在 Maven 2 中，插件的版本会被解析至 latest。也就是说，当用户使用某个非核心插件且没有声明版本的时候，Maven 会将版本解析为所有可用仓库中的最新版本，而这个版本也可能是快照版。

当插件的版本为快照版本时，就会出现潜在的问题。Maven 会基于更新策略，检查并使用快照的更新。某个插件可能昨天还用得好好的，今天就出错了，其原因就是这个快照版本的插件发生了变化。为了防止这类问题，Maven 3 调整了解析机制，当插件没有声明版本的时候，不再解析至 latest，而是使用 release。这样就可以避免由于快照频繁更新而导致的插件行为不稳定。

依赖 Maven 解析插件版本其实是不推荐的做法，即使 Maven 3 将版本解析到最新的非快照版，也还是会有潜在的不稳定性。例如，可能某个插件发布了一个新的版本，而这个版本的行为与之前的版本发生了变化，这种变化就可能导致项目构建失败。因此，使用插件的时候，应该一直显式地设定版本，这也解释了 Maven 为什么要在超级 POM 中为核心插件设定版本。

7.8.4 解析插件前缀

前面讲到 mvn 命令行支持使用插件前缀来简化插件的调用，现在解释 Maven 如何根据插件前缀解析得到插件的坐标。

插件前缀与 groupId:artifactId 是一一对应的，这种匹配关系存储在仓库元数据中。与之前提到的 groupId/artifactId/maven-metadata.xml 不同，这里的仓库元数据为 groupId/maven-metadata.xml，那么这里的 groupId 是什么呢？第 7.6.1 节提到主要的插件都位于 http://repo1.maven.org/maven2/org/apache/maven/plugins/和 http://repository.codehaus.org/org/code-

haus/mojo/，相应地，Maven 在解析插件仓库元数据的时候，会默认使用 org. apache. maven. plugins 和 org. codehaus. mojo 两个 groupId。也可以通过配置 settings. xml 让 Maven 检查其他 groupId 上的插件仓库元数据：

```
<settings>
  <pluginGroups>
    <pluginGroup>com.your.plugins</pluginGroup>
  </pluginGroups>
</settings>
```

基于该配置，Maven 就不仅仅会检查 org/apache/maven/plugins/maven-metadata. xml 和 org/codehaus/mojo/maven-metadata. xml，还会检查 com/your/plugins/maven-metadata. xml。

下面看一下插件仓库元数据的内容，见代码清单 7-10。

代码清单 7-10　插件仓库元数据

```
<metadata>
  <plugins>
    <plugin>
      <name>Maven Clean Plugin</name>
      <prefix>clean</prefix>
      <artifactId>maven-clean-plugin</artifactId>
    </plugin>
    <plugin>
      <name>Maven Compiler Plugin</name>
      <prefix>compiler</prefix>
      <artifactId>maven-compiler-plugin</artifactId>
    </plugin>
    <plugin>
      <name>Maven Dependency Plugin</name>
      <prefix>dependency</prefix>
      <artifactId>maven-dependency-plugin</artifactId>
    </plugin>
  </plugins>
</metadata>
```

上述内容是从中央仓库的 org. apache. maven. plugins groupId 下插件仓库元数据中截取的一些片段，从这段数据中就能看到 maven-clean-plugin 的前缀为 clean，maven-compiler-plugin 的前缀为 compiler，maven-dependency-plugin 的前缀为 dependency。

当 Maven 解析到 dependency:tree 这样的命令后，它首先基于默认的 groupId 归并所有插件仓库的元数据 org/apache/maven/plugins/maven-metadata. xml；其次检查归并后的元数据，找到对应的 artifactId 为 maven-dependency-plugin；然后结合当前元数据的 groupId org. apache. maven. plugins；最后使用第 7. 8. 3 节描述的方法解析得到 version，这时就得到了完整的插件坐标。如果 org/apache/maven/plugins/maven-metadata. xml 没有记录该插件前缀，则接着检查其他 groupId 下的元数据，如 org/codehaus/mojo/maven-metadata. xml，以及用户自定义的插件组。如果所有元数据中都不包含该前缀，则报错。

7.9 小结

本章介绍了 Maven 的生命周期和插件这两个重要的概念。不仅解释了生命周期背后的理念，还详细阐述了 clean、default、site 三套生命周期各自的内容。此外，本章还重点介绍了 Maven 插件如何与生命周期绑定，以及如何配置插件行为，如何获取插件信息。读者还能从命令行的视角来理解生命周期和插件。本章最后结合仓库元数据剖析了 Maven 内部的插件解析机制，希望能使得读者对 Maven 有更深刻的理解。