第5章

深入 Maven 构建工具

5.1 Maven 生命周期

Maven 有 3 种生命周期：Clean 生命周期、Default 生命周期和 Site 生命周期，Default 生命周期是 Manen 的主要生命周期。

5.1.1 项目构建过程与 Maven 生命周期

Maven 用于项目构建和管理。Maven 把项目构建和管理过程定义为生命周期。

一个典型的 Maven 生命周期是由 7 个阶段组成的，如图 5.1、表 5.1 所示。

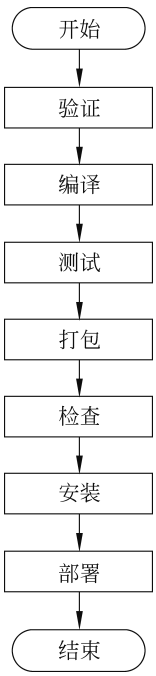

图 5.1 Maven 生命周期

表 5.1 Maven 生命周期

阶　段	描　述
验证(validate)	验证项目是否正确且所有必需信息是可用的
编译(compile)	源代码编译在此阶段完成
测试(test)	使用适当的单元测试框架(例如 JUnit)进行测试
打包(package)	创建 jar/war 包,如在 pom.xml 中定义的包
检查(verify)	对集成测试的结果进行检查,以保证项目质量达标
安装(install)	安装打包的项目到本地仓库,以供其他项目使用
部署(deploy)	复制最终的项目包到远程仓库中,以共享给其他开发人员和项目

为了完成 Default 生命周期,这些阶段(包括其他未列出的生命周期阶段)将按顺序执行。

Maven 的 3 种生命周期相互独立的,这 3 种生命周期不能看成一个整体。每个生命周期的各个环节都是由各种插件完成的。

有些项目的管理不在 Maven 的生命周期中,例如项目骨架的创建不在 Maven 的生命周期中。

5.1.2 Clean 生命周期

Clean 生命周期是在进行真正的构建之前需要进行的清理工作。Clean 生命周期包含 3 个阶段:

(1) 清理前(pre-clean):执行一些需要在清理之前完成的工作。

(2) 清理(clean):移除所有上一次构建生成的文件。

(3) 清理后(post-clean):执行一些需要在清理之后立刻完成的工作。

5.1.3 Default 生命周期

Default 生命周期是 Maven 的主要生命周期,用于构建应用,包括 23 个阶段,如表 5.2 所示。

表 5.2 Maven 的 Default 生命周期

阶　段	描　述
校验(validate)	校验项目是否正确并且所有必要的信息可以完成项目的构建过程
初始化(initialize)	初始化构建状态,例如设置属性值
生成源代码(generate-sources)	生成包含在编译阶段中的任何源代码
处理源代码(process-sources)	处理源代码,例如过滤任意值

续表

阶　段	描　述
生成资源文件(generate-resources)	生成将包含在项目包中的资源文件
处理资源文件(process-resources)	处理资源并复制到目标目录,为打包阶段做好准备
编译(compile)	编译项目的源代码
处理类文件(process-classes)	处理编译生成的文件,例如对 Java class 文件进行字节码改善优化
生成测试源代码(generate-test-sources)	生成包含在编译阶段中的任何测试源代码
处理测试源代码(process-test-sources)	处理测试源代码,例如过滤任意值
生成测试资源文件(generate-test-resources)	为测试创建资源文件
处理测试资源文件(process-test-resources)	处理测试资源并复制到目标目录
编译测试源代码(test-compile)	编译测试源代码到测试目标目录
处理测试类文件(process-test-classes)	处理测试源代码编译生成的文件
测试(test)	使用合适的单元测试框架运行测试(JUint 是其中之一)
准备打包(prepare-package)	在实际打包之前,执行必要的操作,为打包做准备
打包(package)	将编译后的代码打包成可分发格式的文件,例如 jar、war 或者 ear 文件
集成测试前(pre-integration-test)	在执行集成测试前进行必要的操作。例如搭建需要的环境
集成测试(integration-test)	处理和部署项目到可以执行集成测试的环境中
集成测试后(post-integration-test)	在执行集成测试完成后进行必要的操作,例如清理集成测试环境
验证(verify)	进行必要的检查,验证项目包有效且达到质量标准
安装(install)	安装项目包到本地仓库,这样项目包可以用作其他本地项目的依赖包
部署(deploy)	将最终的项目包复制到远程仓库中,与其他开发者和项目共享

5.1.4　Site 生命周期

Site 生命周期生成和部署项目的站点。Site 生命周期包括 4 个阶段:

(1) 生成站点前(pre-site):执行一些需要在生成站点之前完成的工作。

(2) 生成站点(site):生成项目的站点。

(3) 生成站点后(post-site):执行一些需要在生成站点之后完成的工作,并且为站点部署做准备。

(4) 站点部署(site-deploy):将生成的站点部署到特定的服务器上。

5.1.5 生命周期内各阶段和生命周期之间的关系

在一种生命周期内，前后阶段是相互依赖的，如果运行一个阶段的命令，则在该阶段之前的阶段都会被执行。例如，在 Clean 生命周期内，有 3 个阶段：清理前、清理和清理后，如果执行 mvn post-clean 命令，则在此之前的阶段都会被执行。这样大大简化了 Maven 命令的输入。

这 3 种生命周期相互独立。当用户调用 clean 命令时，清理前和清理阶段会顺序执行，但不会影响 Default 生命周期的任何阶段。

5.1.6 Maven 常用命令

Maven 常用命令简要介绍如下。

mvn archetype:generate：创建 Maven 项目。

mvn clean：清除项目目录中的生成结果。

mvn compile：编译项目源代码。

mvn package：将项目打包。

mvn install：将项目的 jar 包安装在本地仓库中。

mvn deploy：发布项目。

mvn test-compile：编译测试源代码。

mvn test：运行应用程序中的单元测试。

mvn site：生成项目相关信息的站点。

mvn eclipse：生成 Eclipse 项目文件。

mvn jetty：启动 Jetty 服务。

mvn tomcat：启动 Tomcat 服务。

5.2 Maven 插 件

5.2.1 Maven 插件框架

Maven 本质上是一个插件框架，它的核心并不执行任何具体的构建任务，所有任务都交给插件完成。Maven 本地插件库如图 5.2 所示。

例如，编译源代码是由 maven-compiler-plugin 插件完成的。

进一步说，每个任务对应一个插件目标(goal)，每个插件会有一个或者多个目标。

例如，maven-compiler-plugin 插件的 compile 目标用来编译位于 src\main\java 目录下的主源代码，testCompile 目标用来编译位于 src\test\java 目录下的测试源代码。

5.2.2 与生命周期有关的插件

Maven 内置与生命周期有关的插件。3 种生命周期各阶段与插件目标的绑定关系如表 5.3 至表 5.5 所示。

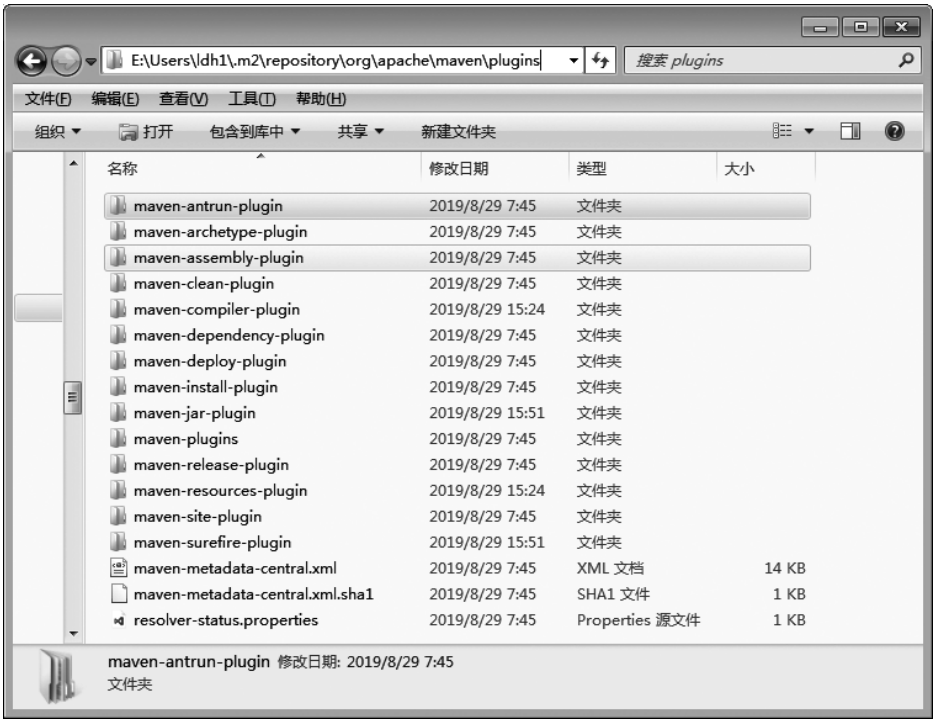

图 5.2　Maven 本地插件库

表 5.3　Clean 生命周期各阶段与插件目标的绑定关系

阶　段	插 件 目 标
清理前	
清理	maven-clean-plugin:clean
清理后	

表 5.4　Default 生命周期各阶段与插件目标的绑定关系及任务

阶　段	插 件 目 标	任　务
处理资源文件	maven-resources-plugin:resources	复制主资源文件至主输出目录
编译	maven-compile-plugin:compile	编译主源代码至主输出目录
处理测试资源文件	maven-resources-plugin:testRresources	复制测试资源文件至测试输出目录
编译测试源代码	maven-compiler-plugin:testCompile	编译测试源代码至测试输出目录
测试	maven-surefire-plugin:test	执行测试用例
打包	maven-jar-plugin:jar	创建项目 jar 包
安装	maven-install-plugin:install	将项目输出构件安装到本地仓库
部署	maven-deploy-plugin:deploy	将项目输出构件部署到远程仓库

表 5.5 Site 生命周期各阶段与插件目标的绑定关系

阶　段	插 件 目 标
生成站点前	
生成站点	maven-site-plugin:site
生成站点后	
站点部署	maven-site-plugin:deploy

5.2.3 插件调用方式

用户可以通过两种方式调用 Maven 插件目标。

(1) 直接在命令行指定要执行的插件及插件目标。

例如,mvn archetype:generate 就表示调用 maven-archetype-plugin 插件的 generate 目标,这种带冒号的调用方式与生命周期无关。在 4.2 节中就是用 mvn archetype:generate 创建 Java 项目骨架:

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=helloapp -
    DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

(2) 将插件目标与生命周期中的阶段绑定。这样,用户在命令行只需要输入生命周期阶段即可。

例如,Maven 默认将 maven-compiler-plugin 插件的 compile 目标与 Default 生命周期的编译阶段绑定。因此,命令 mvn compile 实际上是先定位到 Default 生命周期的编译阶段,然后再根据绑定关系调用 maven-compiler-plugin 插件的 compile 目标。

在 4.6 节中编译 Java 项目就是在命令行只输入生命周期的阶段来调用插件的:

```
mvn compile
```

5.2.4 插件调用方式的差异

生命周期的具体阶段绑定的插件调用(例如 mvn package)与直接执行插件及插件目标(例如 mvn jar:jar package 绑定 maven-jar-plugin:jar)效果是不一样的。

mvn jar 只执行打包这一步。

mvn package 不仅执行 maven-jar-plugin:jar 打包这一步,而且执行 Default 生命周期中打包阶段之前的所有阶段。

1. 直接调用插件打包

用 maven-jar-plugin:2.4:jar 插件(命令为 mvn jar:jar)进行打包,输出结果如图 5.3 所示。

从图 5.3 可以看出,mvn jar:jar 仅仅执行了 maven-jar-plugin:2.4:jar (default-cli) @ helloapp 打包插件进行打包。

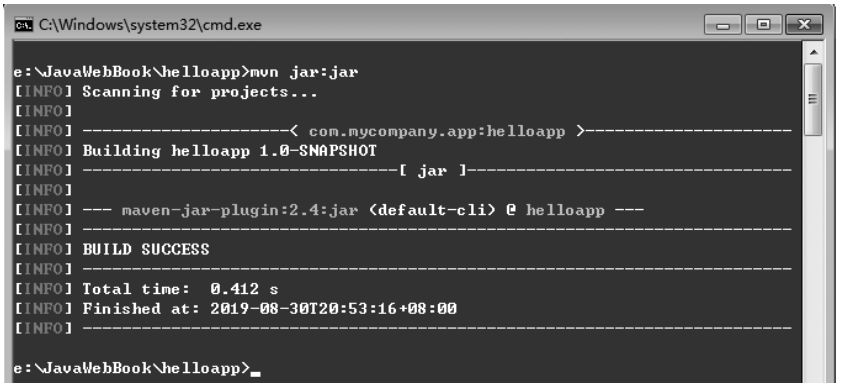

图 5.3　直接调用插件打包的输出结果

2. 用生命周期方式调用插件

用生命周期方式调用插件进行打包使用 mvn package 命令，输出结果如图 5.4 所示。

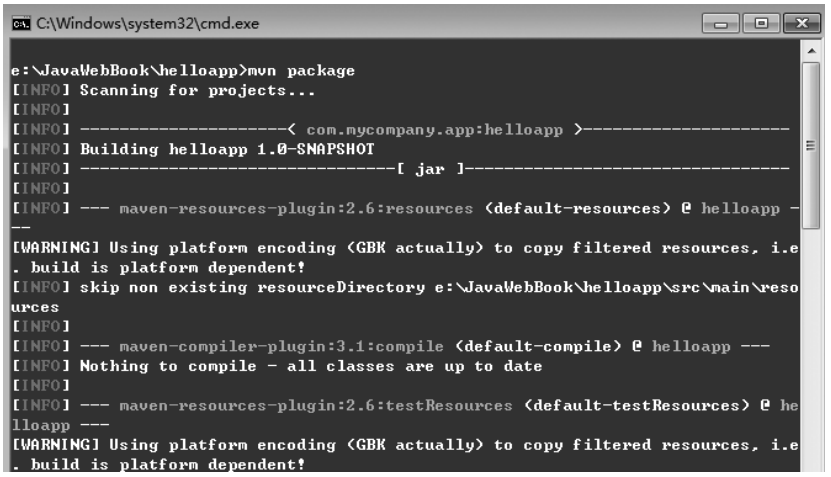

图 5.4　用生命周期方式调用插件打包的输出结果

输出的完整信息如下：

```
[INFO] Scanning for projects...
[INFO] -----------------< com.mycompany.app:helloapp >----------------
[INFO] Building helloapp 1.0-SNAPSHOT
[INFO] --------------------------[ jar ]---------------------------
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ helloapp -
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e.
build is platform dependent!
[INFO] skip non existing resourceDirectory e:\JavaWebBook\helloapp\src\main
\resources
```

```
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ helloapp ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ helloapp ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e.
build is platform dependent!
[INFO] skip non existing resourceDirectory e:\JavaWebBook\helloapp\src\test
\resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ helloapp ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ helloapp ---
[INFO] Surefire report directory: e:\JavaWebBook\helloapp\target\surefire
-reports
-------------------------------------------------------
T E S T S
-------------------------------------------------------
Running com.mycompany.app.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ helloapp ---
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 1.680 s
[INFO] Finished at: 2019-08-30T21:05:03+08:00
[INFO] ------------------------------------------------------------
```

可以看出，mvn package 不仅执行了 maven-jar-plugin：2.4：jar（default-cli）@ helloapp 打包插件进行打包，还执行了 Default 生命周期中打包阶段之前的几个阶段的插件。执行的插件如下：

maven-resources-plugin：2.6：resources

maven-compiler-plugin：3.1：compile

maven-resources-plugin：2.6：testResources

maven-compiler-plugin：3.1：testCompile

maven-surefire-plugin：2.12.4：test

maven-jar-plugin：2.4：jar

5.2.5　插件的配置

插件都有默认配置。当默认配置不满足需求时，需要在 pom.xml 中添加相应插件的配置信息。例如，指定编译 JDK 版本为 1.8，在配置标签＜plugins＞中配置如下：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <!-- since 2.0 -->
    <version>3.7.0</version>
    <configuration>
        <!-- use the Java 8 language features -->
        <source>1.8</source>
        <!-- want the compiled classes to be compatible with JVM 1.8 -->
        <target>1.8</target>
        <!-- The -encoding argument for the Java compiler -->
        <encoding>UTF-8</encoding>
    </configuration>
</plugin>
```

5.2.6　绑定生命周期与插件目标

生命周期大部分阶段都有默认绑定的插件目标，当默认的绑定不满足要求时，可以通过配置＜plugin＞标签实现新的绑定。

在＜plugin＞内由子标签＜executions＞和＜execution＞实现绑定配置，子标签＜phase＞表示生命周期的阶段，子标签＜goal＞表示要绑定的插件目标。

下面的实例表示生命周期的打包阶段与 maven-assembly-plugin 插件的 single 目标绑定：

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <archive>
        <manifest>
            <!--这里要替换成 jar 包 main 方法所在类-->
            <mainClass>com.mycompany.app.App</mainClass>
        </manifest>
      </archive>
      <descriptorRefs>
        <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
    <executions>
      <execution>
```

```
            <id>make-assembly</id><!-- 用于继承合并 -->
            <phase>package</phase><!-- 指定在打包节点执行 jar 包合并操作 -->
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
    </plugin>
```

5.3 Maven 构建配置

5.3.1 <build>标签

构建配置在<build>标签内，插件配置标签<plugin>属于<build>标签的子标签，<build>标签是整个 Maven 构建配置标签，主要用于编译、打包、部署配置。<build>标签、<plugins>标签、<plugin>标签的关系如下：

```
<build>
    <finalName>${project.artifactId}</finalName>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <!-- since 2.0 -->
            <version>3.7.0</version>
              <configuration>
                  <!-- use the Java 8 language features -->
                  <source>1.8</source>
                  <!-- want the compiled classes to be compatible with JVM 1.8 -->
                  <target>1.8</target>
                  <!-- The -encoding argument for the Java compiler -->
                  <encoding>UTF-8</encoding>
              </configuration>
        </plugin>
    </plugins>
</build>
```