# Maven的生命周期

1. <https://www.runoob.com/maven/maven-build-life-cycle.html>

## 前言：

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的拓展性，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的体现Maven生命周期的概念，见代码清单7-1：

|  |
| --- |
| 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();  } |

Maven有三套相互独立的生命周期，分别是：

1）Clean(清理)生命周期；

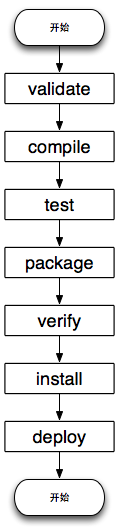
2）Default(默认)生命周期；

3）Site(站点)生命周期；

**每套生命周期都由一组有序阶段（Phase）组成**，平时我们在命令行输入的命令总会对应于一个特定的阶段，如mvn clean就是执行clean阶段。

## 零、典型的构建周期：

一个典型的Maven构建周期由以下几个阶段的序列组成：



|  |  |  |
| --- | --- | --- |
| 阶段 | 处理 | 描述 |
| validate | 验证项目 | 验证项目是否正确且所有必须信息是可用的 |
| compile | 执行编译 | 源代码编译在此阶段完成 |
| test | 测试 | 使用适当的单元测试框架进行测试 |
| package | 打包 | 创建JAR/WAR包 |
| verify | 检查 | 对集成测试的结果进行检查,以保证质量达标 |
| install | 安装 | 安装打包的项目到本地仓库,以供其他项目使用 |
| deploy | 部署 | 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 |

## 一、Clean生命周期：

在进行真正的构建之前执行一些清理工作。

Clean生命周期一共包含三个阶段：

1）pre-clean：执行一些需要在clean之前完成的工作；

2）clean：移除所有上一次构建生成的文件；

3）post-clean：执行一些需要在clean之后立刻完成的工作；

比如，**mvn clean** 就是执行Clean生命周期的clean阶段。在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，**mvn clean** 等同于 **mvn pre-clean clean**

## 二、Site生命周期：

1）pre-site：执行一些需要在生成站点文档之前的工作；

2）site：生成项目的站点文档；

3）post-site：执行一些需要在生成站点文档之后完成的工作，并且为部署做准备；

4）site-deploy：将生成的站点文档部署到特定的服务器上；

## 三、Default生命周期：

这里解释一些比较重要和常用的阶段。

|  |  |
| --- | --- |
| 生命周期阶段 | 描述 |
| validate(校验) | 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 |
| initialize(初始化) | 初始化构建状态,比如设置属性值 |
| generate-source  (生成源代码) | 生成包含在编译阶段中的任何源代码 |
| process-source  (处理源代码) | 处理源代码,比如:过滤任意值 |
| generate-resource  (生成资源文件) | 生成将会包含在项目中的资源文件 |
| process-resource  (处理资源文件) | 复制和处理资源到目标目录,为打包阶段做准备 |
| compile(编译) | 编译项目的源代码 |
| process-class  (处理类文件) | 处理编译生成的文件,比如说对Java Class文件做字节码改善优化 |
| generate-test-source  (生成测试源代码) | 生成包含在编译阶段中的任何测试源代码 |
| process-test-source  (处理测试源代码) | 处理测试源代码,比如说,过滤任意值 |
| generate-test-resource  (生成测试资源文件) | 为测试创建资源文件 |
| process-test-resource  (处理测试资源文件) | 复制和处理测试资源到目标目录 |
| test-compile  (编译测试源码) | 编译测试源码到测试目标目录 |
| process-test-classes  (处理测试资源文件) | 处理测试源代码生成的文件 |
| test(测试) | 使用合适的测试单元框架进行测试 |
| prepare-package  (准备打包) | 在实际打包之前,执行任何的必要的操作为打包做准备 |
| package(打包) | 将编译后的代码打包成可分发格式的文件,比如JAR、WAR或者EAR文件 |
| pre-integration-test  (集成测试前) | 在执行继承测试前进行必要的动作。 |
| integration-test  (集成测试) | 处理和部署项目到可以运行集成测试环境中 |
| post-integration-test  (集成测试后) | 在执行集成测试后进行必要的动作 |
| verify(验证) | 运行任意的检查来验证项目包有效且达到质量标准 |
| install(安装) | 安装项目到本地仓库,这样项目包可以用作其他本地项目的依赖 |
| deploy(部署) | 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 |

## 打包相关生命周期：

打包指的是<packaging>指定的内容，**绑定到每个阶段的特定目标默认根据项目的打包类型设置**。一个打包类型为jar的项目和一个打包类型为war的项目拥有不同的两组默认目标。<packaging>元素影响构建一个项目需要的步骤。

举个打包如何影响构建的例子，考虑有两个项目：一个打包类型是pom，一个打包类型是jar。在package阶段，打包类型为pom的项目会运行site:attach-descriptor目标，而打包类型为jar的项目会运行jar:jar目标。

下面的小节描述了Maven中内建打包类型的生命周期。

### Jar：

JAR是默认的打包类型，是最常用的，因此也就是生命周期配置中最经常遇到的打包类型。JAR生命周期默认如下表所示：



### POM：

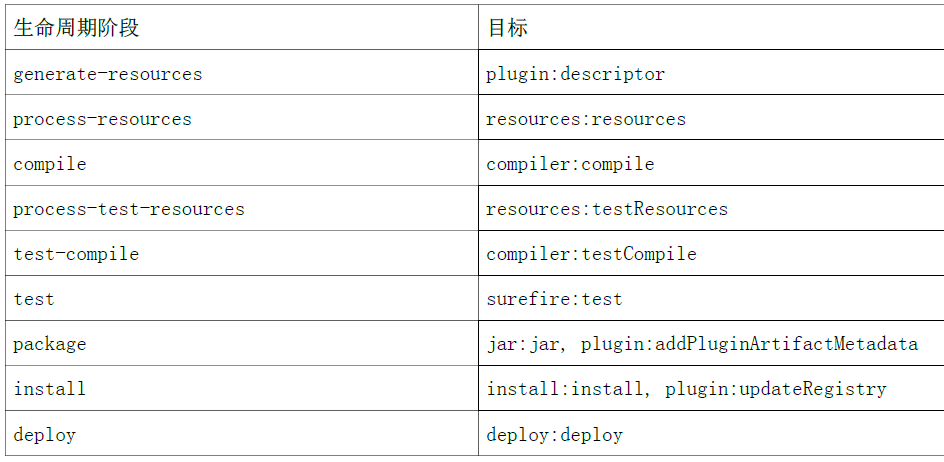
POM是最简单的打包类型。不像一个Jar，Sar或Ear，它生成的构件只是它本身。没有代码需要编译或者测试，也没有资源需要处理。打包类型为POM的项目的默认目标为下表所示：



### Maven Plugin：

该打包类型和Jar打包类型类似，除了三个目标：plugin:descriptor，plugin:addPluginArtifactMatedata，plugin:updateRegistry。这些目标生成一个描述文件，对仓库数据执行一些修改。

打包类型为maven-plugin的项目默认目标如下表：



### EJB：

EJB，或者说企业Java Bean，是企业级Java中模型驱动开发的常见数据访问机制。Maven提供了对EJB2和EJB3的支持。你必须配置EJB插件来为EJB3指定打包类型，否则该插件默认为EJB2.1，并寻找某些EJB配置文件是否存在。打包类型为EJB的项目默认目标如下表：



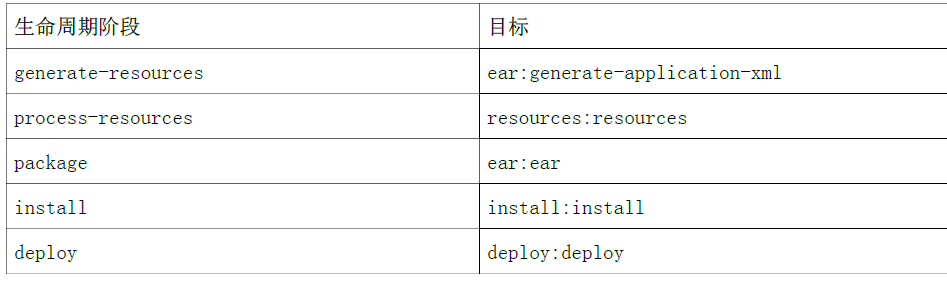
### WAR：

WAR打包类型和JAR以及EJB类似。例外是这里的package目标是war:war。注意war:war插件需要一个web.xml配置文件在项目的src/main/webapp/WEB-INF目录中。打包类型为war的项目默认目标如下表：



### EAR：

EAR可能是最简单的JavaEE结构体，它主要包含一个部署描述符application.xml文件，一些资源和一些模块。EAR插件有个名为generate-application-xml的目标，它是根据EAR项目POM的配置生成application.xml。打包类型为EAR的项目的默认目标如下表所示：



### 其它打包类型：

以上列表并非是Maven中所有可用打包类型。有许多打包格式在外部的项目和插件中可用：NAR（本地归档）打包类型，用来生成Adobe Flash和Flex内容的项目和SWF和SWC打包类型，以及很多其它类型。你也可以自定义打包类型，定制默认的生命周期目标来适应你自己项目的需求。

为了使用自定义的打包类型，你需要两样东西：一个定义了定制打包类型生命周期的插件，和一个包含该插件的仓库。有些定制打包类型是由中央仓库中可用的插件定义的。这里有一个样例项目，它引用了Israfil Flex插件，使用自定义打包类型SWF根据Adobe Flex生成输出。

|  |
| --- |
| <project>  ...  <packaging>swf</packaging>  ...  <build>  <plugins>  <plugin>  <groupId>net.israfil.mojo</groupId>  <artifactId>maven-flex2-plugin</artifactId>  <version>1.4-SNAPSHOT</version>  <extensions>true</extensions>  <configuration>  <debug>true</debug>  <flexHome>${flex.home}</flexHome>  <useNetwork>true</useNetwork>  <main>org/sonatype/mavenbook/Main.mxml</main>  </configuration>  </plugin>  </plugins>  </build>  ...  </project> |

## 通用生命周期目标：

很多打包类型的生命周期有类似的目标，如果你看一下绑定到WAR和JAR生命周期的目标，你会发现它们只有package阶段有区别。WAR生命周期的package阶段调用了war:war，而JAR生命周期的package阶段调用了jar:jar。大部分你将接触的生命周期共享一些通用生命周期目标，用来管理资源，运行测试，以及编译源代码。本节，我们会详细讨论这些通用的生命周期目标。

### Process Resource：

大部分生命周期将resources:resources目标绑定到process-resources阶段。process-resources阶段处理资源并将资源复制到输出目录。

如果你没有自己自定义超级POM中的默认目录位置，Maven就会将src/main/resources中的文件复制到target/classes。

除了复制资源文件至输出目录，Maven同时也会在资源上应用过滤器，能让你替换资源文件中的一些符号。就像在POM中我们通过pom.xml标记引用变量一样，你也可以使用同样的语法在你的项目的资源文件中引用变量。与profile联系起来，这样的特性就能用来生成针对不同部署平台的构件。当我们需要为同一个项目的开发、测试、staging，以及产品平遥环境生成输出的时候，该特性就十分有用。

为了阐述资源过滤，假如你有个带有XML文件src/main/resources/META-INF/service.xml的项目。你想要提取出一些配置变量只一个属性文件。换句话说，你可能想要为你的数据库引用JDBC URL，用户名和密码，并且你不想将这些值直接放到service.xml文件里，而是想要使用一个属性文件来存储你程序中的所有配置点。这么做能让你将所有配置信息固定到单独的一个属性文件中，当你需要面对一个新的部署环境的时候，就很容易更改配置的值。首先，看一下src/main/resources/MEAT-INF/service.xml的内容。

|  |
| --- |
| <service>  <!-- This URL was set by project version 0.6-SNAPSHOT -->  <url>${jdbc.url}</url>  <user>${jdbc.username}</user>  <password>${jdbc.password}</password>  </service> |

该XML文件使用你在POM中用到的同样的属性引用语法，第一个引用的变量是project，它同时也是POM的饮食变量。project变量提供了对POM信息的访问。接下来的三个变量引用是：jdbc.url，jdbc.username，jdbc.password。这些自定义的比阿娘在一个属性文件src/main/filters/default.properties中定义。

【src/main/filters/default.properties】

|  |
| --- |
| jdbc.url=jdbc:hsqldb:mem:mydb  jdbc.username=sa  jdbc.password= |

要配置使用该default.properties文件的资源过滤，我们需要在这个项目的POM中指定两样东西：构建配置的filters元素中的属性文件列表，以及一个标记告诉Maven资源目录需要过滤。默认的Maven行为会跳过顾虑，只是将资源复制到输出目录：你需要显式的配置资源过滤，否则Maven就会置之不理。这种Maven资源过滤的默认行为是了确保不让Maven替换掉一些你不想替换的pom.xml引用。

例：过滤资源（替换属性）

|  |
| --- |
| <build>  <filters>  <filter>src/main/filters/default.properties</filter>  </filters>  <resources>  <resource>  <directory>src/main/resources</directory>  <filtering>true</filtering>  </resource>  </resources>  </build> |

正如Maven中所有目录一样，资源目录并非一定要在src/main/resources。这种事定义在超级POM中的默认值。你应该也注意到你不需要将所有的资源合并到一个单独的目录中。你可以将资源分离至src/main目录下的独立的目录中。

假设你有个项目包含了数百个XML文档和数百个图片。你可能希望创建两个目录src/main/xml和src/main/images来存储这些内容，而不是将它们混合在src/main/resources目录中。为了添加资源目录列表，你需要在内的构建配置中假如如下的resource元素。

|  |
| --- |
| <build>  ...  <resources>  <resource>  <directory>src/main/resources</directory>  </resource>  <resource>  <directory>src/main/xml</directory>  </resource>  <resource>  <directory>src/main/images</directory>  </resource>  </resources>  ...  </build> |

当你构建一个项目用来生成控制台程序或者命令行工具的时候，你通常发现自己正编写一个shell脚本，需要引用构建生成的JAR。当你使用assembly插件为一个应用程序生成如ZIP或TAR的分发包的时候，你可能会将所有的脚本放到如src/main/command的目录下。在下面的POM资源配置中，你会看到我们如何使用资源过滤器和一个对项目变量的引用，生成JAR的最终名称。

例：过滤脚本资源

|  |
| --- |
| <build>  <groupId>org.sonatype.mavenbook</groupId>  <artifactId>simple-cmd</artifactId>  <version>2.3.1</version>  ...  <resources>  <resource>  <filtering>true</filtering>  <directory>/maven-guide-zh-to-production/...  <includes>  <include>run.bat</include>  <include>run.sh</include>  </includes>  <targetPath>/maven-guide-zh-to-production/...</targetPath>  </resource>  <resource>  <directory>/maven-guide-zh-to-production/...</directory>  </resource>  </resources>  ...  </build> |

如果你在该项目下运行mvn process-resources，最后你会在/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh中得到两个文件，run.sh和run.bat，我们在resources元素中挑选出这两个文件，配置过滤器，然后设置targetPath为/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh。在第二个resource元素中，我们配置默认资源路径，使其在不做过滤的情况下被复制到默认输出目录。

### Compile：

大部分生命周期将Compiler插件的compile目标绑定到compile阶段。该阶段会调用compile:compile，后者被配置成编译所有的源码并复制到构建输出目录。如果你没有自定义超级POM中的值，compile:compile将会编译src/main/java中的所有内容至target/classes。Compiler插件调用javac，使用的source设置为1.3，默认target设置为1.1。换句话说，Compiler插件会假设你所有的Java源代码遵循Java 1.3，目标为Java 1.1 JVM。如果你想要更改这些设置，你需要在POM中为Compiler插件提供source和target配置，如例：

【为Compiler插件设置source和target版本】

|  |
| --- |
| <project>  ...  <build>  ...  <plugins>  <plugin>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.5</source>  <target>1.5</target>  </configuration>  </plugin>  </plugins>  ...  </build>  ...  </project> |

要注意我们配置的是Compiler插件，而不是compile:compile目标。如果我们只要为compiler:compiler目标设置source和target，就要将configuration元素放到compile:compile目标的execution元素下。我们为整个插件设置source和target，是因为compile:compile并不是我们唯一感兴趣配置的目标。当Maven使用compile:testCompile目标编译测试代码的时候，Compiler插件会被重用，因此在插件级别配置source和target，一次就能配置该插件的所有目标。

如果你想要自定义源码的位置，你也可以更改构建配置，如果你想要存储项目的源码至src/java而非src/main/java，让构建输出至classes而非target/classes，你可以覆盖定义在超级POM中的sourceDirectory的默认值。

|  |
| --- |
| <build>  ...  <sourceDirectory>src/java</sourceDirectory>  <outputDirectory>classes</outputDirectory>  ...  </build> |

### Process Test Resources：

process-test-resources阶段和process-resources阶段很相似，只有一些微小的差别，但大部分是一样的。你可以像过滤一般的资源那样过滤测试资源。测试资源的默认位置定义在超级POM中，为src/test/resources，，默认的输出目录为targert/test-classes，由/usr/local/hudson/hudson-home/jobs/maven-guide-zh-to-production/workspace/content-zh/target/test-classes定义。

### Test Compile：

test-compile阶段基本上和compile阶段一致。唯一的不同是会调用compile:testCompile编译测试源代码目录至测试构建输出目录。如果你没有在超级POM中自定义默认目录，compile:testCompile将会编译src/test/java中的源码至target/test-classes目录。

类似源代码目录，如果你想要自定义测试源码目录和测试编译输出目录的位置，你可以覆盖testSourceDirectory和testOutputDirectory。如果你想要将测试源代码存储在src-test而非src/test/java，保存测试字节码至classes-test/而非target/test-classes，你可以使用如下的配置：

|  |
| --- |
| <build>  ...  <testSourceDirectory>src-test</testSourceDirectory>  <testOutputDirectory>classes-test</testOutputDirectory>  ...  </build> |

### Test：

大部分生命周期绑定Surefire插件的test目标至test阶段。而Surefire插件是Maven的单元测试插件，Surefire默认的行为是寻找测试源码目录下所有以\*Test结尾的类，以JUnit测试的形式运行它们。Surefire插件也可以配置成运行TestNG单元测试。

运行过mvn test之后，你应该注意到Surefire在target/surefire-reports目录生成了许多报告。该目录内每个Surefire插件运行过的测试都会有相关的两个文件：一个是包含测试运行信息的XML文档，另一个是包含单元测试输出的文本文件。如果测试阶段有问题，单元测试失败，你可以使用Maven的输出以及该木哭瞎的内容来追查测试失败的原因。在站点生成的时候，surefire-reports/目录的内容会被用来创建报告，使项目所有单元测试的总体情况清晰明了。

如果你工作的项目有一些失败的单元测试，同时你想让项目生成输出，你需要配置Surefire插件在遇到失败的情况下继续下一个构建。当遇到单元测试失败的时候，默认行为是停止构建。要覆盖这种行为，你需要设置Surefire插件的testFaulure配置属性为true。

【配置Surefire忽略单元测试失败】

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

如果你想要整个的跳过测试，你可以运行如下的命令：

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

maven.test.skip变量同时控制Compile和Surefire插件。

### 6、Install：

Install插件的install目标基本上都是绑定到install生命周期阶段。install:install目标不过是将项目的主要构件安装到本地仓库，如果你有一个项目，groupId是org.sonatype.mavenbook，artifactId是simple-test，version是1.0.2，那么install:install目标就会从target/simple-test-1.0.2.jar复制Jar文件至~/.m2/repository/org/sonatype/mavenbook/simple-test/1.0.2/simple-test-1.0.2.jar。

如果这个项目的打包类型是POM，那么该目标就仅仅复制POM到本地仓库。

### 7、Deploy：

Deploy插件的deploy目标通常绑定到deploy生命周期阶段。该阶段用来讲一个构件部署到远程Maven仓库，当你执行一次发布的时候通常需要更新远程仓库。一次部署过程可以简单到复制一个文件至另一个目录，或者复杂到使用公钥通过SCP传送一个文件。部署设置通常包含远程仓库的证书，并且，这样的部署设置通常不会存储在pom.xml中。部署设置通常可以在用户单独的~/.m2/settings.xml中找到。到现在为止，你要知道的是是deploy:deploy被绑定到deploy阶段，它会传送一个构件至发布仓库，更新一些可能被此次部署影响的仓库信息。