# Maven的生命周期

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

## 前言：

Maven有三套相互独立的生命周期，分别是：

1）Clean(清理)生命周期；

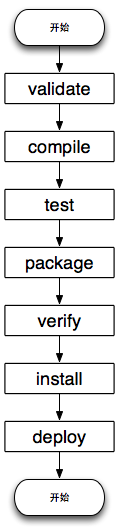
2）Default(默认)生命周期；

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：

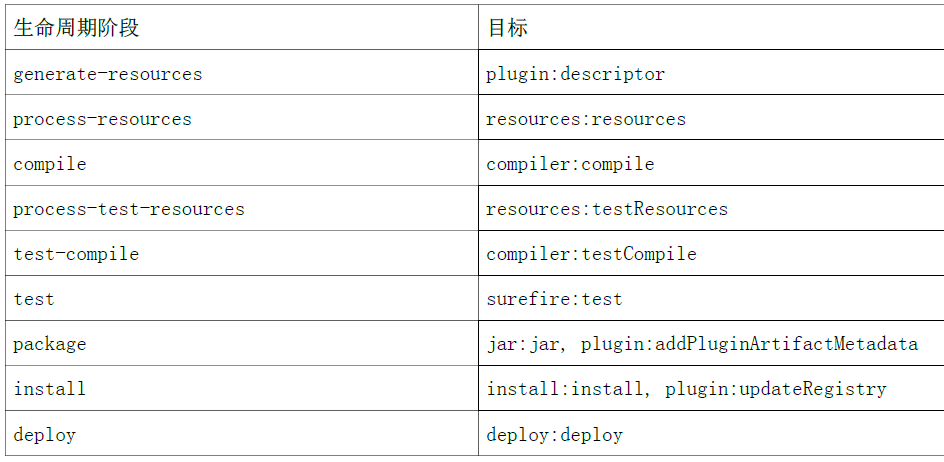
POM是最简单的打包类型。不像一个Jar，Sar或Ear，它生成的构件只是它本身。没有代码需要编译或者测试，也没有资源需要处理。打包类型为POM的项目的默认目标为下表所示：



### 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：



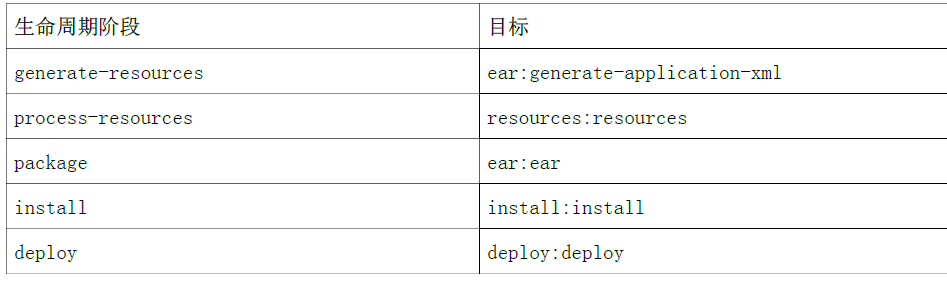
### WAR：

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