以下引用官方的生命周期解释https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html：

一、构建生命周期基础：

Maven基于构建生命周期的中心概念。这意味着构建和分发特定工件（项目）的过程被明确定义。

对于构建项目的人员，这意味着只需要学习一小堆命令即可构建任何Maven项目，**POM**将确保他们获得所需的结果。

有三个内置的生命周期：**默认（default）**，**清洁（clean）**和**站点（site）**。在**默认（default）**的生命周期处理你的项目部署，将**清洁（clean）**的生命周期处理项目的清理，而**网站（site）**的生命周期处理你的项目站点文档的创建。

二、构建生命周期是由阶段组成的：

这些构建生命周期中的每一个由构建阶段的不同列表定义，其中构建阶段表示生命周期中的阶段。

例如，**默认（default）**的生命周期包括以下阶段（注意：这里是简化的阶段，用于生命周期阶段的完整列表，请参阅下方**生命周期参考**）：

- **验证（validate）** - 验证项目是否正确，所有必要的信息可用
- **编译（compile）** - 编译项目的源代码
- **测试（test）** - 使用合适的单元测试框架测试编译的源代码。这些测试不应该要求代码被打包或部署
- **打包（package）** - 采用编译的代码，并以其可分配格式（如JAR）进行打包。
- **验证（verify）** - 对集成测试的结果执行任何检查，以确保满足质量标准
- **安装（install）** - 将软件包安装到本地存储库中，用作本地其他项目的依赖项
- **部署（deploy）** - 在构建环境中完成，将最终的包复制到远程存储库以与其他开发人员和项目共享。

这些生命周期阶段（以及此处未显示的其他生命周期阶段）依次执行，以完成默认生命周期。给定上述生命周期阶段，这意味着当使用默认生命周期时，Maven将首先验证项目，然后尝试编译源代码，运行这些源代码，打包二进制文件（例如jar），运行集成测试软件包，验证集成测试，将验证的软件包安装到本地存储库，然后将安装的软件包部署到远程存储库。

换句话说，在生命周期里面阶段是连续的，在不出错的前提下，比如执行**打包（package）**时就一定是执行了**测试（test）**之后再执行。

三、生命周期参考：

以下列出了**默认（default）**，**清洁（clean）**和**站点（site）**生命周期的所有构建阶段，它们按照指定的顺序执行的顺序执行。

清洁（clean）生命周期

预清洁（pre-clean）	执行实际项目清理之前所需的流程
清洁（clean）	删除以前构建生成的所有文件
后清洁（post-clean）	执行完成项目清理所需的流程

默认（default）生命周期

验证（validate）	验证项目是正确的，所有必要的信息可用。
初始化（initialize）	初始化构建状态，例如设置属性或创建目录。
产生来源（generate-sources）	生成包含在编译中的任何源代码。
流程源（process-sources）	处理源代码，例如过滤任何值。
生成资源（generate-resources）	生成包含在包中的资源。
流程资源（process-resources）	将资源复制并处理到目标目录中，准备打包。
编译（compile）	编译项目的源代码。
工艺类（process-classes）	从编译后处理生成的文件，例如对Java类进行字节码增强。
生成测试来源（generate-test-sources）	生成包含在编译中的任何测试源代码。
流程测试来源（process-test-sources）	处理测试源代码，例如过滤任何值。
生成测试资源（generate-test-resources）	创建测试资源。
流程测试资源（process-test-resources）	将资源复制并处理到测试目标目录中。
测试编译（test-compile）	将测试源代码编译到测试目标目录中
流程检验类（process-test-classes）	从测试编译中处理生成的文件，例如对Java类进行字节码增强。对于Maven 2.0.5及以上版本。
测试（test）	使用合适的单元测试框架运行测试。这些测试不应该要求代码被打包或部署。
制备包（prepare-package）	在实际包装之前，执行必要的准备包装的操作。这通常会导致打包的处理版本的包。（Maven 2.1及以上）
打包（package）	采取编译的代码，并以其可分发的格式（如JAR）进行打包。
预集成测试（pre-integration-test）	在执行集成测试之前执行所需的操作。这可能涉及诸如设置所需环境等。
集成测试（integration-test）	如果需要，可以将该包过程并部署到可以运行集成测试的环境中。
整合后的测试（post-integration-test）	执行集成测试后执行所需的操作。这可能包括清理环境。
校验（verify）	运行任何检查以验证包装是否有效并符合质量标准。
安装（install）	将软件包安装到本地存储库中，以作为本地其他项目的依赖关系。
部署（deploy）	在集成或发布环境中完成，将最终软件包复制到远程存储库，以与其他开发人员和项目共享。

站点（site）生命周期

预网站（pre-site）	在实际的项目现场生成之前执行所需的进程
网站（site）	生成项目的站点文档
后网站（post-site）	执行完成站点生成所需的进程，并准备站点部署
网站部署（site-deploy）	将生成的站点文档部署到指定的Web服务器

四、生命周期阶段在命令行中的调用

在开发环境中，使用以下调用构建并将工件安装到本地存储库中。

```
mvn install
```

此命令在执行安装之前按顺序（**验证（validate）**，**编译（compile）**，**打包（package）**等）执行每个默认生命周期阶段。在这种情况下，您只需要调用最后一个构建阶段来执行，**安装（install）**。

在构建环境中，使用以下调用将工件清理地构建并部署到共享存储库中。

```
mvn clean deploy
```

相同的命令可以在多模块场景（即具有一个或多个子项目的项目）中使用。Maven遍历每个子项目并执行**清洁（clean）**，然后执行**部署（deploy）**（包括所有之前的构建阶段步骤）。

注意：在我们开发阶段，有一些生命周期的阶段，比如**验证（validate）**这些，基本很少用到。只要使用关键的几个基本能满足需求。

五、通常情况在命令行只调用某些特定的阶段

以连字符（**pre-***，**post-***或**process-***）命名的阶段通常不会从命令行直接调用。这些阶段对构建进行排序，生成在构建之外无用的中间结果。在调用集成测试的情况下，环境可能处于挂起状态。

诸如Jacoco和执行容器插件（如Tomcat，Cargo和Docker）的代码覆盖工具将目标绑定到**预集成测试（pre-integration-test）**阶段以准备集成测试容器环境。这些插件还将目标绑定到**整合后的测试（post-integration-test）**阶段，以收集覆盖统计信息或停止集成测试容器。

故障安全和代码覆盖插件将目标绑定到**集成测试（integration-test）**和**验证（verify）**阶段。最终结果是在**验证（verify）**阶段后可以使用测试和覆盖率报告。如果从命令行调用**集成测试（integration-test）**阶段，则不会生成任何报告。整合测试容器环境处于更糟糕的悬挂状态; Tomcat网络服务器或Docker实例将保持运行，并且Maven本身甚至可能不会终止。

提示：再次明确，在生命周期的阶段上，只有特定的几个阶段对于构建有意义。一些无用的阶段只起到了中间阶段的作用，换句话说只是一个过客。

六、调用由插件目标（Plugin Goals）组成的构建阶段

然而，即使构建阶段负责构建生命周期中的特定步骤，其执行这些职责的方式可能会有所不同。这是通过声明绑定到这些构建阶段的插件目标来完成的。

插件目标代表一个特定的任务（比构建阶段更精细），有助于项目的构建和管理。它可能被限制在零个或多个构建阶段。不限于任何构建阶段的目标可以通过直接调用在构建生命周期之外执行。执行顺序取决于调用目标和构建阶段的顺序。例如，考虑下面的命令。该**清洁（clean）**和**打包（package）**是建立阶段，而**dependency:copy-dependencies**是（一个插件）的目标。

```
mvn clean dependency:copy-dependencies package
```

如果要执行此操作，则将首先执行**清洁（clean）**阶段（意味着它将运行清洁生命周期的所有先前阶段以及**清洁（clean）**阶段本身），然后执行**dependency:copy-dependencies**目标，然后才能最终执行**打包（package）**阶段（以及默认生命周期的所有之前的构建阶段）。

而且，如果一个目标被绑定到一个或者多个构建阶段，那么在所有这些阶段都会调用这个目标。

此外，构建阶段也可以有零个或多个目标。如果构建阶段没有绑定目标，则构建阶段将不会执行。但是，如果它有一个或多个目标，它将执行所有这些目标（注意：在Maven 2.0.5及更高版本中，绑定到阶段的多个目标的执行顺序与POM中声明的顺序相同，但不支持同一插件的多个实例，同一个插件的多个实例被分组一起执行要在Maven 2.0.11及以上）。

提示：其实简单点理解就是说dependency是一个插件，在我们执行生命周期阶段时，可以调用这个插件做特定的事，其中copy-dependencies就是特定的事，那么上面的命令可以这么理解，在执行clean后就会执行dependecy这个插件，最后再执行package；如果dependecy这个插件执行过程异常，package就不会执行到。还有就是一个命令可以有多个插件，也可以一个插件都没有。

提示2：dependency:copy-dependencies这样的形式最标准的官方说法：左边dependency为插件，右边copy-dependencies为插件的目标，通常还有一种说法就是命令行参数。

七、使用构建生命周期来设置项目

构建生命周期足够简单，但是当您为项目配置Maven构建时，您如何将任务分配到每个构建阶段？

打包

第一个也是最常见的方法是通过同样命名的POM元素为您的项目设置**打包**。一些有效的打**包**值是**jar**，**war**，**ear**和**pom**。如果没有指定包装值，它将默认为**jar**。

每个不同类型的打**包**都包含要绑定到特定阶段的目标列表。例如，**jar**包将绑定以下目标来构建默认生命周期的阶段。
这几乎是一个标准的绑定；然而，一些包装处理它们不同。例如，纯粹的元数据（包装值是**pom**）的项目只将目标绑定到安**装（install）**和**部署（deploy）**阶段（对于某些包装类型的目标到构建阶段绑定的完整列表，请参阅**生命周期参考**）。

请注意，对于某些可用的打包类型，您可能还需要在POM的部分中包含一个特定的插件，并为该插件指定`true`。需要这种插件的一个例子是Plexus插件，它提供**plexus-application**和**plexus-service**打包。

提示：这里简单点可以说不同的包对应不同的生命周期阶段，比如jar包和war包的区别可以参考：https://maven.apache.org/ref/3.5.0/maven-core/default-bindings.html。上方列表可以这么理解，左边是简化的命令，右侧是详细的插件加目标（命令行参数）的形式；切记，**Maven都是以插件的形式存在的**，包括生命周期的阶段同样也是一个个不同的插件组成，比如上面的编译（compile）就是由compiler插件提供，其中compile为这个插件的目标，也可以说是插件的命令行参数。

插件

将目标添加到阶段的第二种方法是在项目中配置插件。插件是为Maven提供目标的工件。此外，插件可以具有一个或多个目标，其中每个目标代表该插件的能力。例如，编译器（**compiler**）插件有两个目标：`compile`和`testCompile`。前者编译主代码的源代码，后者编译测试代码的源代码。

如稍后部分所示，插件可以包含指示将目标绑定到的生命周期阶段的信息。请注意，自己添加插件是不够的，您还必须指定要作为构建的一部分运行的目标。

配置的目标将被添加到已经从选定的**打包**绑定到生命周期的目标。如果将多个目标绑定到特定阶段，则使用的顺序是首先执行来自**打包**装的顺序，然后执行在POM中配置。请注意，您可以使用元素来获得对特定目标的顺序更多的控制。

例如，**Modello**插件默认将目标`modello:java`绑定到`generate-sources`阶段（注意：`modello:java`目标生成Java源代码）。因此，要使用**Modello**插件，并从模型生成源代码并将其合并到构建中，您可以在的部分中将以下内容添加到POM中：

```
... <plugin><groupId>org.codehaus.modellogroupId><artifactId>modello-maven-pluginartifactId><version>1.8.1version>
<executions><execution><configuration><models><model>src/main/mdo/maven.mdomodel>models>
<version>4.0.0version>configuration><goals><goal>javagoal>goals>execution>executions>plugin>...
```

提示：这里所说的更直白的意思就是Modello插件有默认的生命周期阶段，而无需自己手动配置这些阶段。

你可能会想知道为什么这个元素在那里。这样，如果需要，您可以使用不同的配置多次运行相同的目标。还可以使用单独的执行标识，以便在继承或应用配置文件期间，您可以控制目标配置是合并还是转为额外的执行。

当给出与特定阶段匹配的多个执行时，它们按照POM中指定的顺序执行，继承的执行首先运行。

现在，在`modello:java`的情况下，它只在**generate-sources**阶段才有意义。但是一些目标可以在一个以上的阶段中使用，也可能没有合理的默认。对于那些，您可以自己指定阶段。例如，假设您有一个目标**display:time**当前时间到命令行的时间，并希望它在`process-test-resources`阶段运行以指示测试何时开始。这将被配置如下：

```
... <plugin><groupId>com.mycompany.examplegroupId><artifactId>display-maven-pluginartifactId><version>1.0version>
<executions><execution><phase>process-test-resourcesphase><goals><goal>timegoal>goals>execution>executions>plugin>...
```

提示：这里已经是换了一个插件了，与上面的插件不同的是，可以在中指定生命周期阶段。可以说是两个做了一个明显对比。

八、一些参考

完整的Maven生命周期由`maven-core`模块中的`components.xml`文件定义，并附有相关文档供参考。

Maven中2.x中，默认的生命周期的绑定被纳入的`components.xml`，但在Maven的3.x中，它们在一个单独的被定义`default-bindings.xml`描述符。

请参阅生命周期参考和插件绑定，以获取直接从源代码获取的最新文档的默认生命周期参考。

总结：

1、整体来自官方文档，中文来自谷歌翻译，文中难免有一些直译的错误，但是不影响理解。

2、文中的提示为自己的个人理解。

3、在理解生命周期时市面上的说法没有官方来的更直接更明了。

4、切记所有的生命周期的阶段都是以插件的形式存在的，或者这么说吧，Maven的一切就是一个大插件包集合。

5、文中提到的目标，我理解成了命令行参数，比如**dependency:copy-dependencies**整体就是叫做目标，而我喜欢把**copy-dependencies**叫做目标，**dependency**叫做插件。