Maven 可能涉及许多东西，但简而言之，Maven 试图将模式应用于项目的构建基础结构，以便通过提供使用最佳实践的明确路径来提高理解和生产力。Maven 本质上是一个项目管理和理解工具，因此提供了一种帮助管理的方法：

* Builds
* Documentation
* Reporting
* Dependencies
* SCMs
* Releases
* Distribution

Maven 可以采用标准约定和做法来加快开发周期，同时帮助您实现更高的成功率，从而为您的构建过程带来好处。

为了创建我们的第一个Maven项目，我们将使用Maven的archetype机制。archetype被定义为原始模式或模型，所有其他同类事物都从中制造。在 Maven 中，archetype是项目的模板，它与一些用户输入相结合，生成根据用户要求定制的工作 Maven 项目。我们将向您展示archetype机制现在是如何工作的，但如果您想了解有关原型的更多信息，请参阅我们的原型简介。

开始创建您的第一个项目！为了创建最简单的 Maven 项目，请从命令行执行以下内容：

1. mvn -B archetype:generate \
2. -DarchetypeGroupId=org.apache.maven.archetypes \
3. -DgroupId=com.mycompany.app \
4. -DartifactId=my-app

执行此命令后，您会注意到发生了一些事情。首先，您会注意到为新项目创建了名为 my-app 的目录，并且此目录包含名为 pom.xml 的文件，该文件应如下所示：

pom.xml 包含此项目的Project Object Model （POM）。POM 是 Maven 的基本工作单位。请务必记住，因为 Maven 本质上是以项目为中心的，因为一切都围绕着项目的概念

简而言之，POM 包含有关项目的每一条重要信息，本质上是一站式购物，用于查找与您的项目相关的任何内容。了解 POM 很重要，我们鼓励新用户参阅 POM 简介。

1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
6. <groupId>com.mycompany.app</groupId>
7. <artifactId>my-app</artifactId>
8. <packaging>jar</packaging>
9. <version>1.0-SNAPSHOT</version>
10. <name>Maven Quick Start Archetype</name>
11. <url>http://maven.apache.org</url>
12. <dependencies>
13. <dependency>
14. <groupId>junit</groupId>
15. <artifactId>junit</artifactId>
16. <version>4.11</version>
17. <scope>test</scope>
18. </dependency>
19. </dependencies>
20. </project>

Project 这是所有 Maven pom.xml 文件中的顶级元素。

modelVersion 此元素指示此 POM 正在使用的对象模型的版本。模型本身的变化非常少，但如果 Maven 开发人员认为有必要更改模型，则它是强制性的，以确保使用的稳定性。

groupId此元素指示创建项目的组织或组的唯一标识符。GroupId 是项目的关键标识符之一，通常基于组织的完全限定域名。例如，org.apache.maven.插件是所有 Maven 插件的指定组 Id。

artifactId 此元素指示此项目正在生成的主工件的唯一基名称。项目的主要工件通常是 JAR 文件。源捆绑包等辅助工件也使用工件Id作为最终名称的一部分。Maven 生产的典型工件的形式<artifactId>-<version>.<extension>（例如，myapp-1.0.jar）。

packaging 此元素指示此工件使用的包装类型（例如 JAR、WAR、EAR 等）。这不仅意味着产生的工件是 JAR、WAR 或 EAR，还可以指示要用作生成过程一部分的特定生命周期。（生命周期是我们将在指南中进一步讨论的主题。现在，请记住，项目指示的打包可以在自定义生成生命周期中发挥作用。打包元素的默认值为 JAR，因此不必为大多数项目指定此值。

Version 此元素指示项目生成的项目的版本。Maven 在帮助您进行版本管理方面大有作为，您经常会看到一个版本中的 SNAPSHOT 指示符，这表明项目处于开发状态。我们将在本指南中讨论snapshots（快照）的使用以及它们如何进一步工作。

name此元素指示用于项目的显示名称。这通常用于 Maven 生成的文档。

url 此元素指示项目站点的找不到位置。这通常用于 Maven 生成的文档

Change to the directory where pom.xml is created by archetype:generate and execute the following command to compile your application sources:

1. mvn compile

第一次执行此（或任何其他）命令时，Maven 将需要下载实现该命令所需的所有插件和相关依赖项。

从输出中可以看出，编译的类被放置在${basedir}/target/classes中，这是 Maven 采用的另一个标准约定

现在，您成功编译了应用程序的源，现在您已经有了一些要编译和执行的单元测试

Execute the following command:

1. mvn test

Maven 这次下载了更多的依赖项。这些是执行测试所需的依赖项和插件（它已经具有编译所需的依赖项，并且不会再下载它们）。

在编译和执行测试之前，Maven 会编译主代码（所有这些类都是最新的，因为我们自上次编译以来没有更改任何内容）。

如果只想编译测试源（但不执行测试），可以执行以下操作：

1. mvn test-compile

制作 JAR 文件可以通过执行以下命令来完成：

1. mvn package

现在，您需要在本地存储库中安装已生成的项目（JAR 文件）（${user.home}/.m2/repository存储库是默认位置）。为此，请执行以下命令：

1. mvn install

请注意，surefire 插件（用于执行测试）查找具有特定命名约定的文件中包含的测试。默认情况下，包含的测试包括：

* \*\*/\*Test.java
* \*\*/Test\*.java
* \*\*/\*TestCase.java

默认排除为：

* \*\*/Abstract\*Test.java
* \*\*/Abstract\*TestCase.java

您已经走过了设置、构建、测试、打包和安装典型 Maven 项目的过程

如果你已经注意到，你能够做的一切，到这一点已经由18行文件，即项目的模型或POM驱动。如果您查看一个典型的 Ant 生成文件，该文件提供了我们迄今实现的相同功能，您会发现它的大小已经是 POM 的两倍，我们才刚刚开始！Maven 为您提供的功能要大得多，

Maven 还能提供更多的功能，而且在目前POM 的基础上无需添加其他东西

那么，你还能免费得到什么？有很多Maven插件，开箱即用。我们将在这里特别提到一个，因为它是Maven非常珍贵的功能之一：不需要其他任何工作，你目前的POM有足够的信息来为你的项目生成一个网站。

您很可能想要自定义 Maven 网站，但如果您被时间所压，则只需执行以下命令，就能提供项目的基本信息：

1. mvn site

请注意，下面所示的 pom.xml 文件中的版本标记的值有后缀： -SNAPSHOT。

<version>1.0-SNAPSHOT</version>

SNAPSHOT 值是指开发分支上的"最新"代码，并且不保证代码是稳定的或不变的

在发布过程中，版本号x.y-SNAPSHOT更改为 x.y。发布过程还将开发版本增量为 x.（y=1）-SNAPSHOT。例如，版本 1.0-SNAPSHOT 作为版本 1.0 发布，新的开发版本是版本 1.1-SNAPSHOT。

The release process also increments the development version to x.(y+1)-SNAPSHOT. For example, version 1.0-SNAPSHOT is released as version 1.0, and the new development version is version 1.1-SNAPSHOT.

每当您想要自定义 Maven 项目的build时，通过添加或重新配置插件来完成。

在此示例中，我们将配置 Java 编译器以允许 JDK 5.0 源。这非常简单，只需将此添加到 POM 中即可：

 <build>

 <plugins>

 <plugin>

 <groupId>org.apache.maven.plugins</groupId>

 <artifactId>maven-compiler-plugin</artifactId>

 <version>3.3</version>

 <configuration>

 <source>1.5</source>

 <target>1.5</target>

 </configuration>

 </plugin>

 </plugins>

 </build>

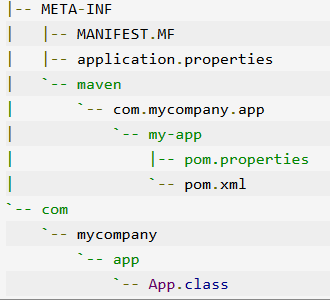
您会注意到，Maven 中的所有插件看起来都像依赖项 - 在某些方面它们就是。此插件将自动下载和使用 - 包括特定版本，如果您指定它（默认是使用最新的可用）。

configuration元素将给定的参数应用于编译器插件中的每个目标。在上述情况下，编译器插件已用作生成过程的一部分，这仅会更改配置。还可以向流程添加新目标，并配置特定目标。有关此信息，请参阅生成生命周期简介。

另一个可以满足的常见用例是将资源打包到 JAR 文件中，无需更改上面的 POM。对于此常见任务，Maven 再次依赖于标准目录布局，这意味着通过使用标准 Maven 约定，只需将这些资源放在标准目录结构中即可在 JAR 中打包资源。

我们添加了目录 ${basedir}/src/main/resources，我们将要打包到 JAR 中的任何资源放入其中。Maven 采用的简单规则是：放置在 ${basedir}/src/main/resources目录中的任何目录或文件都打包在 JAR 中，从 JAR 的根开始，结构完全相同。





您还会注意到一些其他文件，如 META-INF/MANIFEST.MF 以及 pom.xml 和 pom.properties 文件。这些是 Maven 中 JAR 的生成的标准配置。如果愿意，您可以创建自己的清单，但默认情况下，Maven 将生成一个清单。

pom.xml 和 pom.properties 文件打包在 JAR 中，以便 Maven 生成的每个工件都是自描述的，并且还允许您在需要时在您自己的应用程序中使用元数据。一个简单的用途可能是检索应用程序的版本。对 POM 文件进行操作需要您使用一些 Maven 实用程序，但可以使用标准 Java API 使用这些属性，

有时，资源文件需要包含只能在生成时才提供的值。为了在 Maven 中实现此目的，请使用语法 ${<property name>}.将包含该值的属性引用到资源文件中。该属性可以是 pom.xml 中定义的值之一、在用户设置中定义的值settings.xml、在外部属性文件中定义的属性或 system 属性。

要在复制时让 Maven 筛选资源，只需将 pom.xml 中的资源目录筛选设置为 true：

 <build>

 <resources>

 <resource>

 <directory>src/main/resources</directory>

 <filtering>true</filtering>

 </resource>

 </resources>

 </build>

为了引用在 pom.xml 中定义的属性，属性名称使用定义值的 XML 元素的名称。允许"pom"作为项目（root） 元素的别名。因此${project.name} 引用项目的名称，${project.version} 引用项目的版本，${project.build.finalName} 是指打包生成项目时创建的文件的最终名称。POM 的某些元素具有默认值，因此无需在 pom.xml 中显式定义这些值，以便此处提供这些值。同样，可以使用以"settings "开头的属性名称引用用户settings.xml中的值（例如，${settings.localRepository}引用用户本地存储库的路径）。

让我们向应用程序添加几个属性文件（我们将其放在 src/main/resources目录中），其值将在筛选资源时提供：

# application.properties

application.name=${project.name}

application.version=${project.version}

完成该操作后，可以执行以下命令（process-resources是复制和筛选资源的生成生命周期阶段）：

1. mvn process-resources

要引用在外部文件中定义的属性，您只需在 pom.xml 中添加对此外部文件的引用。首先，让我们创建外部属性文件， src/main/filters/filter.properties:

1. # filter.properties
2. my.filter.value=hello!

接下来，我们将在 pom.xml 中添加对此新文件的引用：

 <build>

 <filters>

 <filter>src/main/filters/filter.properties</filter>

 </filters>

 <resources>

 <resource>

 <directory>src/main/resources</directory>

 <filtering>true</filtering>

 </resource>

 </resources>

 </build>

作为在外部文件中定义 my.filter.value 属性的替代方法，您还可以在 pom.xml 的属性部分中定义该属性

 <properties>

 <my.filter.value>hello</my.filter.value>

 </properties>

筛选资源还可以从系统属性获取值;构建到 Java 中的系统属性（如 java.version 或 user.home）或使用标准 Java -D 参数在命令行上定义的属性。为了继续这个示例，让我们将application.properties文件更改为如下所示：

 java.version=${java.version}

 command.line.prop=${command.line.prop}

现在，当您执行以下命令（请注意命令行上command.line.prop属性的定义）时，application.properties 文件将包含来自 system 属性的值。

1. mvn process-resources "-Dcommand.line.prop=hello again"

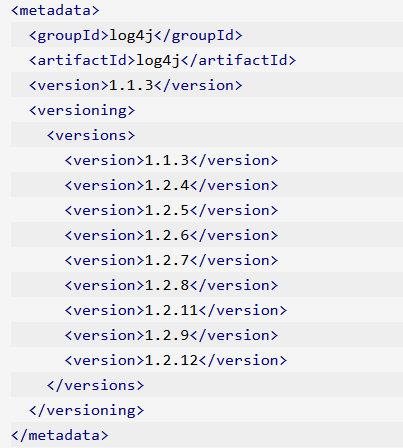
您可能已经注意到 POM 中作为示例使用的dependencies元素。事实上，您一直在使用外部依赖项，但在这里我们将更详细地讨论其工作原理。

对于每个外部依赖项，您需要定义至少 4 个内容：groupId、artifactId 、version和scope (作用域)。scope元素指示项目如何使用该依赖项，并且可以是编译、测试和运行时等值。

通过此信息有关依赖项，Maven 将能够在生成项目时引用依赖项。Maven 从何处引用依赖项？Maven 查找本地存储库（${user.home}/.m2/repository是默认位置），以查找所有依赖项。

每当项目引用本地存储库中不可用的依赖项时，Maven 将从远程存储库下载依赖项到本地存储库。默认情况下，可以在http://repo.maven.apache.org/maven2/找到（和浏览）远程存储库 Maven。您还可以设置自己的远程存储库（可能是公司的中央存储库），以代替默认远程存储库或添加到默认远程存储库之外。

让我们为我们的项目添加另一个依赖项。假设我们已经向代码添加了一些日志记录，并且需要将 log4j 添加为依赖项。首先，我们需要知道 log4j 的 groupId、工件 ID 和版本是什么。Maven 中心上的相应目录称为 /maven2/log4j/log4j。该目录中有一个称为 maven-metadata.xml 的文件。以下是 log4j 的 maven 元数据.xml 的外观：



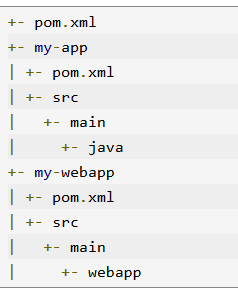
要将 jar 部署到外部存储库，必须在 pom.xml 中配置存储库 URL，并在settings.xml.中配置与存储库连接的身份验证信息。

下面是使用 scp 和用户名/密码身份验证的示例：

**How do I build more than one project at once?**

处理多个模块的概念内置于 Maven 中。在本节中，我们将演示如何在上面构建 WAR，并在一个步骤中包括以前的 JAR。

首先，我们需要在其他两个目录之上添加一个父 pom.xml 文件，因此它应该如下所示：



1. <project xmlns="http://maven.apache.org/POM/4.0.0"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4. http://maven.apache.org/xsd/maven-4.0.0.xsd">
5. <modelVersion>4.0.0</modelVersion>
7. <groupId>com.mycompany.app</groupId>
8. <artifactId>app</artifactId>
9. <version>1.0-SNAPSHOT</version>
10. <packaging>pom</packaging>
12. <modules>
13. <module>my-app</module>
14. <module>my-webapp</module>
15. </modules>
16. </project>

添加 Webapp 对 JAR 的依赖，因此将其添加到我的 webapp/pom.xml 中

 <dependency>

 <groupId>com.mycompany.app</groupId>

 <artifactId>my-app</artifactId>

 <version>1.0-SNAPSHOT</version>

 </dependency>

Finally, add the following <parent> element to both of the other pom.xml files in the subdirectories:

 <parent>

 <groupId>com.mycompany.app</groupId>

 <artifactId>app</artifactId>

 <version>1.0-SNAPSHOT</version>

 </parent>

在上层目录运行

1. mvn verify

WAR 现已在my-webapp/target/my-webapp.war 中创建，并且 JAR 包括在其中

这是如何工作的？首先，父 POM的packaging元素值是pom并且定义了模块列表。这告诉 Maven 在项目集上运行所有操作，而不仅仅是当前操作（要重写此行为，可以使用 --非递归命令行选项）。

接下来，我们告诉 WAR，它需要my-app JAR。这做了几件事：它使它在WAR中的任何代码的类路径上可用（在这种情况下没有），它确保JAR始终在WAR之前构建，并且它指示WAR插件将JAR纳入其库目录。

你可能已经注意到，Junit-4.11.jar是一种依赖，但最终并没有在WAR中结束。原因是<scope> test </scope>元素 它仅需要测试

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's site documentation.

**有三个内置的生成生命周期default, clean and site**

**默认生命周期处理项目部署，清理生命周期处理项目清理，而站点生命周期处理项目站点文档的创建。**

**A Build Lifecycle is Made Up of Phases**

**这些生成生命周期中的每一个都由不同的一系列**build phases组成

例如，默认生命周期包括以下phases

validate  **- 验证项目是否正确，并且所有必要的信息都可用**

compile  **- 编译项目的源代码**

test **- 使用合适的单元测试框架测试已编译的源代码。**

package **- 获取已编译的代码，并将其打包为可分发格式，如 JAR。**

verify  **- 对集成测试结果运行任何检查，以确保满足质量标准**

**install - 将包安装到本地存储库中，在本地其他项目中用作依赖项**

**deploy - 在生成环境中完成，将最终包复制到远程存储库，以便与其他开发人员和项目共享。**

**这些生命周期阶段（加上此处未显示的其他生命周期阶段）按顺序执行，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，针对测试运行这些源，打包二进制文件（例如 jar），针对该包运行集成测试，验证集成测试，将已验证的包安装到本地存储库，然后将已安装的包部署到远程存储库。**

**在开发环境中，使用以下调用将工件生成并安装到本地存储库中。**

mvn install

**此命令在执行安装之前按顺序执行默认生命周期每个阶段（验证、编译、包等）。您只需调用要执行的最后一个生成阶段**

**在生成环境中，使用以下调用将项目干净地生成和部署到共享存储库中。**

mvn clean deploy

**A Build Phase is Made Up of Plugin Goals**

**A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases.**

**不绑定到任何生成阶段的目标可以通过直接调用在生成生命周期之外执行。执行顺序取决于调用目标和生成阶段的顺序。**The clean and package arguments are build phases, while the dependency:copy-dependencies is a goal (of a plugin).

mvn clean dependency:copy-dependencies package

**如果要执行此项，将首先执行清理阶段（这意味着它将运行清理生命周期的所有前面阶段，加上清理阶段本身），然后执行目标**dependency:copy-dependencies**，最后执行**package**阶段（及默认生命周期中它之前的所有阶段）。.**

**此外，如果目标绑定到一个或多个构建阶段，则将在所有这些阶段调用该目标。**

**此外，生成阶段也可以具有绑定到它的零个或多个目标。如果生成阶段没有绑定到它的目标，则该生成阶段将不会执行。但是，如果它有一个或多个目标绑定到它，它将执行所有这些目标**

**您为项目构建 Maven 生成时，如何为每个生成阶段分配任务？**

**Packaging**

The first, and most common way, is to set the packaging for your project via the equally named POM element <packaging>. Some of the valid packaging values are jar, war, ear and pom. If no packaging value has been specified, it will default to jar.

Each packaging contains a list of goals to bind to a particular phase. For example, the jar packaging will bind the following goals to build phases of the default lifecycle.



**Plugins**

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven.

**此外，插件可能具有一个或多个目标，其中每个目标表示该插件的功能。**For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

**plugins can contain information that indicates which lifecycle phase to bind a goal to**

**The goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. Note that you can use the <executions> element to gain more control over the order of particular goals.**

**Project Inheritance**

The Super POM is Maven's default POM. All POMs extend the Super POM unless explicitly set, meaning the configuration specified in the Super POM is inherited by the POMs you created for your projects.

The Super POM is one example of project inheritance, however you can also introduce your own parent POMs by specifying the parent element in the POM,

 <parent>

 <groupId>com.mycompany.app</groupId>

 <artifactId>my-app</artifactId>

 <version>1</version>

 </parent>

However, that would work if the parent project was already installed in our local repository or was in that specific directory structure (parent pom.xml is one directory higher than that of the module's pom.xml). we would have to add the <relativePath> element to our parent section

 <parent>

 <groupId>com.mycompany.app</groupId>

 <artifactId>my-app</artifactId>

 <version>1</version>

 <relativePath>../parent/pom.xml</relativePath>

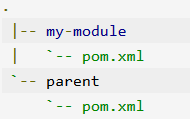
 </parent>

**Project Aggregation**

Project Aggregation is similar to [Project Inheritance](https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Project_Inheritance). But instead of specifying the parent POM from the module, it specifies the modules from the parent POM. By doing so, the parent project now knows its modules, and if a Maven command is invoked against the parent project, that Maven command will then be executed to the parent's modules as well. To do Project Aggregation, you must do the following:

* Change the parent POMs packaging to the value "pom".
* Specify in the parent POM the directories of its modules (children POMs

But what if we change the directory structure to the following:



 <modules>

 <module>../my-module</module>

 </modules>

One of the practices that Maven encourages is *don't repeat yourself*. However, there are circumstances where you will need to use the same value in several different locations. To assist in ensuring the value is only specified once, Maven allows you to use both your own and pre-defined variables in the POM.

Any field of the model that is a single value element can be referenced as a variable. For example, ${project.groupId}, ${project.version}, ${project.build.sourceDirectory} and so on.

These variables are all referenced by the prefix "project.". You may also see references with pom. as the prefix, or the prefix omitted entirely - these forms are now deprecated and should not be used.

You are also able to reference any properties defined in the project as a variable. Consider the following example:

 <properties>

 <mavenVersion>2.1</mavenVersion>

 </properties>

There are the build and the reporting plugins:

 **Build plugins** will be executed during the build and they should be configured in the <build/> element from the POM.

**Reporting plugins** will be executed during the site generation and they should be configured in the <reporting/> element from the POM.

Configuring Build Plugins

A mojo is a Maven plain Old Java Object. Each mojo is an executable goal in Maven, and a plugin is a distribution of one or more related mojos.

Maven plugins (build and reporting) are configured by specifying a <configuration> element where the child elements of the <configuration> element are mapped to fields, or setters, inside your Mojo (remember that a plug-in consists of one or more Mojos where a Mojo maps to a goal)

You can also configure a mojo using the <executions> tag. This is most commonly used for mojos that are intended to participate in some phases of the [build lifecycle](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html).

The first execution with id "execution1" binds this configuration to the test phase. The second execution does not have a <phase> tag, If the goal has a default phase binding then it will execute in that phase. But if the goal is not bound to any lifecycle phase then it simply won't be executed during the build lifecycle.

 <build>

 <plugins>

 <plugin>

 <artifactId>maven-myquery-plugin</artifactId>

 <version>1.0</version>

 <executions>

 <execution>

 <id>execution1</id>

 <phase>test</phase>

 <configuration>

 <url>http://www.foo.com/query</url>

 <timeout>10</timeout>

 <options>

 <option>one</option>

 <option>two</option>

 <option>three</option>

 </options>

 </configuration>

 <goals>

 <goal>query</goal>

 </goals>

 </execution>

 <execution>

 <id>execution2</id>

 <configuration>

 <url>http://www.bar.com/query</url>

 <timeout>15</timeout>

 <options>

 <option>four</option>

 <option>five</option>

 <option>six</option>

 </options>

 </configuration>

 <goals>

 <goal>query</goal>

 </goals>

 </execution>

 </executions>

 </plugin>

 </plugins>

 </build>

1. /\*\*
2. \* @goal query
3. \* @phase package
4. \*/
5. public class MyBindedQueryMojo
6. extends AbstractMojo
7. {
8. @Parameter(property = "query.url", required = true)
9. private String url;
11. @Parameter(property = "timeout", required = false, defaultValue = "50")
12. private int timeout;
14. @Parameter(property = "options")
15. private String[] options;
17. public void execute()
18. throws MojoExecutionException
19. {
20. ...
21. }
22. }

From the above mojo example, MyBindedQueryMojo is by default bound to the package phase (see the @phase notation). But if we want to execute this mojo during the install phase and not with package we can rebind this mojo into a new lifecycle phase using the <phase> tag under <execution>.