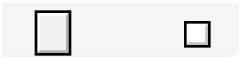







# Maven入门指南

本指南旨在作为第一次使用Maven的人员的参考，但也可作为具有独立参考和常见用例解决方案的烹饪手册。对于初次使用的用户，建议您按顺序逐步浏览材料。对于更熟悉Maven的用户，本指南致力于为手头的需求提供快速解决方案。此时假设您已下载Maven并在本地计算机上安装了Maven。如果您还没有这样做，请参阅下载和安装说明。

好的，所以你现在安装了Maven，我们准备好了。在我们进入示例之前，我们将简要介绍一下Maven是什么以及它如何帮助您完成日常工作以及与团队成员的协作。当然，Maven将为小型项目工作，但Maven通过允许团队成员专注于项目的利益相关者所需要的内容来帮助团队更有效地运作。您可以将构建基础结构留给Maven！

## 第



## 什么是Maven？

乍一看，Maven看起来很多东西，但简而言之，Maven试图将模式应用于项目的构建基础架构，以

便通过提供使用最佳实践的明确路径来提高理解力和生产力。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：

- 构建
- 文档
- 报告
- 依赖
- 供应链管理系统
- 发布
- 分配

如果您想对Maven的更多背景信息，你可以检查出的Maven的哲学和Maven的的历史。现在让我们继续讨论用户如何使用Maven从中受益。

## Maven如何使我的开发过程受益？

Maven可以通过采用标准惯例和实践来加速您的开发周期，同时帮助您获得更高的成功率，从而为您的构建过程带来好处。

现在我们已经介绍了Maven的一些历史和目的，让我们进入一些真实的例子，让你开始和Maven一起运行！

## 我如何设置 Maven？

对Maven的默认值通常是足够了，但如果你需要更改缓存位置或背后一个HTTP代理服务器，您需要创建配置。有关更多信息，请参阅配置Maven指南。

## 我如何制作我的第一个 Maven项目？

我们将一头扎进创建您的第一个Maven项目！为了创建我们的第一个Maven项目，我们将使用Maven的原型机制。原型被定义为原始模式或模型，从中创建所有其他相同类型的东西。在Maven中，原型是项目的模板，它与一些用户输入相结合，以生成一个根据用户要求定制的工作Maven项目。我们将向您展示原型机制现在如何工作，但如果您想了解更多关于原型的信息，请参阅我们的原型简介。

在创建你的第一个项目！要创建最简单的Maven项目，请从命令行执行以下命令：

```
mvn -B原型：生成\
  -DarchetypeGroupId=org。阿帕奇。maven。原型\
  -DgroupId=com。我的公司。app
  -DartifactId=我的-应用程序
```

执行完此命令后，您会注意到发生了一些事情。首先，您将注意到已为新项目创建了名为`my-app`的目录，该目录包含一个名为`pom.xml`的文件，该文件应如下所示：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

如何贡献
获得帮助
问题管理
获取Maven来源
Maven团队

项目文件
项目信息

MAVEN项目
原型
神器解决
Doxia
JR
Maven的父POM
插件
插件测试
插件工具
资源包
SCM
共享组件
皮肤
万全
车皮

ASF
Apache的工作原理
基础
赞助Apache
谢谢

Custom Search

Follow

Built by: maven

```
6.   <groupId> com.mycompany.app </ groupId>
7.   <artifactId> my-app </ artifactId>
8.   <包装> 罐子</ packaging>
9.   <version> 1.0-SNAPSHOT </ version>
10.  <name> Maven Quick Start Archetype </ name>
11.  <url> http://maven.apache.org </ url>
12.  <依赖性>
13.    <依赖性>
14.      <groupId> junit </ groupId>
15.      <artifactId> junit </ artifactId>
16.      <version> 4.11 </ version>
17.      <scope> test </ scope>
18.    </依赖性>
19.  </依赖>
20. </项目>
```

`pom.xml`包含此项目的项目对象模型（POM）。POM是Maven的基本工作单元。这一点很重要，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包含有关您项目的所有重要信息，并且基本上是一站式购物，用于查找与您的项目相关的任何内容。了解POM很重要，鼓励新用户参考POM简介。

这是一个非常简单的POM，但仍然显示每个POM包含的关键元素，所以让我们逐一介绍它们以熟悉POM要点：

- **project**这是所有Maven pom.xml文件中的顶级元素。
- **modelVersion**此元素指示此POM使用的对象模型的版本。模型本身的版本很少发生变化，但如果Maven开发人员认为有必要更改模型，则必须确保使用的稳定性。
- **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.plugins`是所有Maven插件的指定groupId。
- **artifactId**此元素指示此项目生成的主工件的唯一基本名称。项目的主要工件通常是JAR文件。像源包这样的辅助工件也使用artifactId作为其最终名称的一部分。Maven生成的典型工件将具有<artifactId> - <version>。<extension>形式（例如，`myapp-1.0.jar`）。
- **packaging**此元素指示此工件要使用的包类型（例如JAR，WAR，EAR等）。这不仅意味着生成的工件是JAR，WAR还是EAR，还可以指示要在构建过程中使用的特定生命周期。（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，项目的指示包装可以在定制构建生命周期中起作用。）`包装`元素的默认值是JAR所以你不必为大多数项目指定这个。
- **version**此元素指示项目生成的工件的版本。Maven在很大程度上帮助您进行版本管理，您经常会在一个版本中看到`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生成的文档中。
- **url**此元素指示可以找到项目站点的位置。这通常用于Maven生成的文档中。
- **description**此元素提供项目的基本描述。这通常用于Maven生成的文档中。

有关可在POM中使用的元素的完整参考，请参阅我们的POM参考。现在让我们回到手头的项目。

在第一个项目的原型生成之后，您还会注意到已创建以下目录结构：

```
1. 我- 应用程序
2. |  - pom 。XML
3. `  -  src
```

```
|  - 主要
| ` - java
| ` -用
| ` - myCompany的
| ` -  app
|                   `-- App.java
` - 测试
    ` - java
      ` - 用
        ` - myCompany的
          ` -  app
            `-- AppTest.java
```

如您所见，从原型创建的项目具有POM，应用程序源的源树和测试源的源树。这是Maven项目的标准布局（应用程序源位于`$ {basedir} / src / main / java`中，测试源位于`$ {basedir} / src / test / java`中，其中$ {basedir}表示包含`pom`的目录`.xml`）。

如果您手动创建Maven项目，这是我们建议使用的目录结构。这是一个Maven约定，要了解更多信息，您可以阅读我们的标准目录布局简介。

现在我们有一个POM，一些应用程序源，以及一些你可能会问的测试源.....

## 如何编译我的应用程序源？

切换到archetype创建pom.xml的目录：生成并执行以下命令来编译应用程序源：

```
mvn编译
```

执行此命令后，您应该看到如下输出：

```
[ INFO ] ------------------------------------------------ ------------------------
------
[ INFO ] 构建Maven 快速入门原型
[ INFO ]     任务- 细分：[ 编译]
[ INFO ] ------------------------------------------------ ------------------------
------
[ INFO ] 神器组织。阿帕奇。maven 。插件：maven - 资源- 插件：\
  检查用于更新由中央
...
[ INFO ] 神器组织。阿帕奇。maven 。插件：maven - 编译器- 插件：\
  检查用于更新由中央
...
[ INFO ] [ 资源：资源]
...
[ INFO ] [ 编译器：编译]
将1个源文件编译为<dir> / my - app / target / classes
[ INFO ] ------------------------------------------------ ------------------------
------
[ 信息]建立成功
[ INFO ] ------------------------------------------------ ------------------------
```

```
    ------
18. [ INFO ] 总时间: 3 分54 秒
19. [ INFO ] 成品在: 周五9月的23 15 : 48 : 34 GMT - 05 : 00 2005
20. [ INFO ] 最终记忆: 2M / 6M
21. [ INFO ] ------------------------------------------ ----------------------
    ------
```

第一次执行此（或任何其他）命令时，Maven将需要下载完成命令所需的所有插件和相关依赖项。从一个干净的Maven安装，这可能需要一段时间（在上面的输出中，它花了将近4分钟）。如果再次执行该命令，Maven现在将拥有它所需的内容，因此它不需要下载任何新内容，并且能够更快地执行命令。

从输出中可以看出，编译后的类被放在`$ {basedir} / target / classes`中，这是Maven采用的另一个标准约定。所以，如果你是一个敏锐的观察者，你会注意到通过使用标准约定，上面的POM非常小，你不必明确告诉Maven你的任何来源在哪里或输出应该去哪里。遵循标准的Maven约定，您可以轻松完成大量工作！就像偶然的比较一样，让我们来看看你在Ant中可能要做的事情来完成同样的事情。

现在，这只是编译一个应用程序源代码树，所显示的Ant脚本与上面显示的POM几乎相同。但是我们会看到只用那个简单的POM我们能做多少！

## 如何编译测试源并运行单元测试？

现在你已经成功编译了应用程序的源代码，现在你已经有了一些你想要编译和执行的单元测试（因为每个程序员总是编写并执行他们的单元测试* nudge nudge wink wink *）。

执行以下命令：

```
1. mvn测试
```

执行此命令后，您应该看到如下输出：

```
1. [ INFO ] ------------------------------------------ ----------------------
   ------
2. [ INFO ] 构建Maven 快速入门原型
3. [ INFO ]     任务- 细分: [ 测试]
4. [ INFO ] ------------------------------------------ ----------------------
   ------
5. [ INFO ] 神器组织。阿帕奇。maven 。插件: maven - surefire - 插件: \
6.   检查用于更新由中央
7. ...
8. [ INFO ] [ 资源: 资源]
9. [ INFO ] [ 编译器: 编译]
10. [ INFO ] 无需编译- 所有课程都是最新的
11. [ INFO ] [ 资源: testResources ]
12. [ INFO ] [ 编译器: testCompile ]
13. 将1个源文件编译为C : \ Test \ Maven2 \ test \ my - app \ target \ test - classes
14. ...
15. [ INFO ] [ surefire : test ]
16. [ INFO ] 设置报告目录: C : \ Test \ Maven2 \ test \ my - app \ target / surefire
```

```
    - 报告
17.
18. -------------------------------------------------------
19.  试验
20. -------------------------------------------------------
21. [ surefire ]正在运行com 。我的公司。app 。AppTest
22. [ 万无一失] 试验运行: 1 , 故障: 0 , 错误: 0 , 时间经过的: 0 秒
23.
24. 结果:
25. [ surefire ] 测试运行: 1 , 失败: 0 , 错误: 0
26.
27. [ INFO ] ------------------------------------------- ----------------------
    ------
28. [ 信息]建立成功
29. [ INFO ] ------------------------------------------- ----------------------
    ------
30. [ INFO ] 总时间: 15 秒
31. [ INFO ] 成品在: 星期四10月的06 08 : 12 : 17 MDT 2005
32. [ INFO ] 最终记忆: 2M / 8M
33. [ INFO ] ------------------------------------------- ----------------------
    ------
```

有关输出的注意事项:

- Maven这次下载了更多依赖项。这些是执行测试所必需的依赖项和插件（它已经具有编译所需的依赖项，不会再次下载它们）。
- 在编译和执行测试之前，Maven编译主代码（所有这些类都是最新的，因为自从我们最后编译以来我们没有改变任何东西）。

如果您只想编译测试源（但不执行测试），则可以执行以下操作:

```
1. mvn test - 编译
```

既然您可以编译应用程序源代码，编译测试并执行测试，那么您将需要继续执行下一个逻辑步骤，这样您就会问...

## 如何创建JAR并将其安装在本地存储库中？

制作JAR文件非常简单，可以通过执行以下命令来完成:

```
1. mvn 包
```

如果您查看项目的POM，您会注意到`包装`元素设置为`jar`。这就是Maven知道如何从上面的命令生成一个JAR文件（稍后我们将详细讨论）。您现在可以查看`$ {basedir} / target`目录，您将看到生成的JAR文件。

现在，您需要在本地存储库中安装您生成的工件（JAR文件）（`$ {user.home} /。m2 / repository`是默认位置）。有关存储库的更多信息，请参阅我们的存储库简介，但让我们继续安装我们的工件！为此，请执行以下命令:

```
        安装
```

```
mvn
```

执行此命令后，您应该看到以下输出：

```
[ INFO ] ------------------------------------------- ----------------------
------
[ INFO ] 构建Maven 快速入门原型
[ INFO ]     任务- 细分: [ 安装]
[ INFO ] ------------------------------------------- ----------------------
------
[ INFO ] [ 资源: 资源]
[ INFO ] [ 编译器: 编译]
将1个源文件编译为<dir> / my - app / target / classes
[ INFO ] [ 资源: testResources ]
[ INFO ] [ 编译器: testCompile ]
将1个源文件编译为<dir> / my - app / target / test - classes
[ INFO ] [ surefire : test ]
[ INFO ] 设置报告目录: <dir> / my - app / target / surefire - 报告

-------------------------------------------------------
 试验
-------------------------------------------------------
[ surefire ]正在运行com 。我的公司。app 。AppTest
[ 万无一失] 试验运行: 1 , 故障: 0 , 错误: 0 , 时间经过: 0.001 秒

结果:
[ surefire ] 测试运行: 1 , 失败: 0 , 错误: 0

[ INFO ] [ jar : jar ]
[ INFO ] 构建jar : <dir> / my - app / target / my - app - 1.0 - SNAPSHOT 。罐
[ INFO ] [ 安装: 安装]
[ INFO ] 安装<dir> / my - app / target / my - app - 1.0 - SNAPSHOT 。罐子里
   < local - repository > / com / mycompany / app / my - app / 1.0 - SNAPSHOT /
my - app - 1.0 - SNAPSHOT 。罐
[ INFO ] ------------------------------------------- ----------------------
------
[ 信息]建立成功
[ INFO ] ------------------------------------------- ----------------------
------
[ INFO ] 总时间: 5 秒
[ INFO ] 成品在: 星期二10月的04 13 : 20 : 32 GMT - 05 : 00 2005
[ INFO ] 最终记忆: 3M / 8M
[ INFO ] ------------------------------------------- ----------------------
------
```

请注意，surefire插件（执行测试）会查找包含在具有特定命名约定的文件中的测试。默认情况下，包括的测试是：

- `** / * Test.java`

- `** /测试*的.java`
- `** / * TestCase.java`

默认排除是：

- `** /摘要* Test.java`
- `** /摘要* TestCase.java`

您已经完成了设置，构建，测试，打包和安装典型Maven项目的过程。这可能是Maven所做的绝大部分项目，如果您已经注意到，到目前为止您所做的一切都是由18行文件驱动的，即项目模型或POM。如果你看一个典型的Ant 构建文件，它提供了我们迄今为止所实现的相同功能，你会发现它已经是POM的两倍，我们刚刚开始！您可以从Maven获得更多功能，而无需像我们目前的POM那样添加任何POM。要从我们的示例Ant构建文件中获取更多功能，您必须继续进行容易出错的添加。

那么还有什么可以免费获得的？有很多Maven插件可以开箱即用，甚至可以像我们上面那样使用简单的POM。我们在这里特别提到一个，因为它是Maven非常珍贵的功能之一：没有任何工作，这个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您的Maven站点，但是如果您需要时间，只需执行以下命令即可提供有关项目的基本信息：

```
1. mvn网站
```

还有许多其他独立目标也可以执行，例如：

```
1. mvn干净
```

这将在启动之前删除包含所有构建数据的`目标`目录，以使其新鲜。

# 什么是SNAPSHOT版本？

请注意，下面显示的`pom.xml`文件中的**version**标记的值具有后缀：`-SNAPSHOT`。

```
1. <project xmlns =“http://maven.apache.org/POM/4.0.0”
2.   ...
3.   <groupId> ... </ groupId>
4.   <artifactId> my-app </ artifactId>
5.   ...
6.   <version> 1.0-SNAPSHOT </ version>
7.   <name> Maven Quick Start Archetype </ name>
8.   ...
```

该`快照`值指的是沿着一个发展分支的“最新”的代码，并且不保证该代码是稳定的或不变。相反，“发布”版本中的代码（没有后缀`SNAPSHOT的`任何版本值）是不变的。

换句话说，SNAPSHOT版本是最终'发布'版本之前的'开发'版本。SNAPSHOT比它的版本“更老”。

在发布过程中，**xy-SNAPSHOT**的版本更改为**xy**。发布过程还将开发版本增加到**x**。（**y + 1**）**-SNAPSHOT**。例如，版本**1.0-SNAPSHOT**作为版本**1.0**发布，新版本版本版本为**1.1-SNAPSHOT**。

# 我如何使用插件？

每当您想要为Maven项目自定义构建时，都可以通过添加或重新配置插件来完成。

**Maven 1.0**用户注意事项：在Maven 1.0中，您可以在`maven.xml`中添加一些`preGoal`，在`project.properties`中添加一些条目。在这里，它有点不同。

对于此示例，我们将配置Java编译器以允许JDK 5.0源。这就像将其添加到您的POM一样简单：

```
...
<建立>
  <插件>
    <插件>
      <groupId> org 。阿帕奇。maven 。插件< / groupId>
      <artifactId> maven-compiler-plugin </ artifactId >
      <version> 3.3 < / version>
      <结构>
        <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 - 在某些方面它们都是。此插件将自动下载和使用 - 如果您请求，则包括特定版本（默认为使用最新版本）。

在`配置`要素适用于给定参数，从编译器插件每一个目标。在上面的例子中，编译器插件已经被用作构建过程的一部分，这只是改变了配置。还可以向流程添加新目标，并配置特定目标。有关此信息，请参阅构建生命周期简介。

要了解插件可用的配置，您可以查看插件列表并导航到您正在使用的插件和目标。有关如何配置插件的可用参数的一般信息，请参阅配置插件指南。

## 如何向JAR添加资源？

另一个可以满足的常见用例是不需要更改我们上面的POM就是在JAR文件中打包资源。对于这个常见任务，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，您可以简单地将这些资源放在标准目录结构中来封装JAR中的资源。

您在下面的示例中看到我们添加了目录`$ {basedir} / src / main / resources`，我们将要包装的任何资源放入JAR中。Maven采用的简单规则是：`$ {basedir} / src / main / resources`目录中的任何目录或文件都打包在JAR中，其结构与JAR基础相同。

```
我- 应用程序
|  - pom 。XML
` -  src
   |  - 主要
   | |  -  java
   | | ` - 用
   | | ` - 我的公司
   | | ` - app
```

```
|   |               `-- App.java
| ` - 资源
| ` -  META-INF
|           `-- application.properties
` - 测试
    ` - java
        ` - 用
            ` - myCompany的
                ` -  app
                    `-- AppTest.java
```

因此，您可以在我们的示例中看到，我们在该目录中有一个带有`application.properties`文件的`META-INF`目录。如果您解压缩Maven为您创建的JAR并查看它，您会看到以下内容：

```
|  - META - INF
| |  - 清除。MF
|   |-- application.properties
| ` -  maven
|       `-- com.mycompany.app
| ` - 我的应用程序
|               |-- pom.properties
| ` - pom 。XML
` - 用
    ` - myCompany的
        ` -  app
            ` - App 。类
```

如您所见，`$ {basedir} / src / main / resources的内容`可以从JAR的基础开始找到，我们的`application.properties`文件位于`META-INF`目录中。您还会注意到其他一些文件，如`META-INF / MANIFEST.MF`以及`pom.xml`和`pom.properties`文件。这些标准是在Maven中生成JAR的标准。如果您选择，您可以创建自己的清单，但如果不这样做，Maven将默认生成一个清单。（您也可以修改默认清单中的条目。我们稍后会详细介绍。）`pom.xml`和`pom.properties`文件在JAR中打包，以便Maven生成的每个工件都是自描述的，并且如果需要，还允许您在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。在POM文件上操作需要使用一些Maven实用程序，但可以使用标准Java API来使用这些属性，如下所示：

```
#Geven由Maven
#Tue Oct 04 15:43:21 GMT-05: 00 2005
version = 1.0 - SNAPSHOT
groupId = com 。我的公司。应用
artifactId = 我的- 应用程序
```

要为单元测试的类路径添加资源，您将遵循与向JAR添加资源相同的模式，除了您放置资源的目录是$ {basedir} / src / test / resources。此时，您将拥有一个项目目录结构，如下所示：

```
我- 应用程序
|  - pom 。XML
` -  src
```

```
    |  - 主要
    | |  -  java
    | | ` - 用
    | | ` - 我的公司
    | | ` - app
    |   |              `-- App.java
    | ` - 资源
    | ` -  META-INF
    |          |-- application.properties
    ` - 测试
        |  - java
        | ` -用
        | ` - myCompany的
        | ` -  app
        |              `-- AppTest.java
        ` - 资源
            `-- test.properties
```

在单元测试中，您可以使用以下代码的简单代码段来访问测试所需的资源：

```
...

//检索资源
InputStream 是= getClass （）。getResourceAsStream （“/ test.properties” ）;

//对资源做点什么

...
```

## 如何过滤资源文件？

有时，资源文件需要包含只能在构建时提供的值。要在Maven中完成此操作，请使用语法`${<property name>}`将包含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，用户settings.xml中定义的值，外部属性文件中定义的属性或系统属性。

要在复制时让Maven过滤资源，只需为`pom.xml中`的资源目录设置`过滤`为true：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi: schemaLocation = “http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 罐子</ packaging>

  <name> Maven Quick Start Archetype </ name>
```

```
  <url> http://maven.apache.org </ url>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
  </依赖>

  <建立>
    <资源>
      <资源>
        <directory> src / main / resources </ directory>
        <filtering> true </ filtering>
      </资源>
    </资源>
  </建造>
</项目>
```

您会注意到我们必须添加以前不存在的`构建`，`资源`和`资源`元素。此外，我们必须明确声明资源位于src / main / resources目录中。所有这些信息都是以前提供的默认值，但由于`过滤`的默认值为false，我们必须将其添加到我们的pom.xml中，以覆盖该默认值并将`过滤`设置为true。

要引用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（根）元素的别名。所以`$ {project.name}`指的是项目的名称，`$ {project.version}`指的是项目的版本，`$ {project.build.finalName}`指的是建立项目时创建的文件的最终名称包装等等。请注意，POM的某些元素具有默认值，因此不需要在`pom.xml中`明确定义此处可用的值。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}`指的是用户本地存储库的路径。

为了继续我们的示例，让我们将一些属性添加到`application.properties`文件（我们放在`src / main / resources`目录中），其值将在过滤资源时提供：

```
# application.properties
申请。name = $ { project 。名字}
申请。version = $ { project 。版本}
```

有了这个，您就可以执行以下命令（process-resources是复制和过滤资源的构建生命周期阶段）：

```
mvn进程- 资源
```

和`目标/类`下的`application.properties`文件（最终会进入jar）看起来像这样：

```
# application.properties
申请。name = Maven 快速入门原型
申请。version = 1.0 - SNAPSHOT
```

要引用外部文件中定义的属性，您需要做的就是在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并将其命名为`src / main / filters / filter.properties`：

```
# filter.properties
我的。过滤器。value = 你好!
```

接下来，我们将在`pom.xml`中添加对此新文件的引用：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi：schemaLocation = “http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 罐子</ packaging>

  <name> Maven Quick Start Archetype </ name>
  <url> http://maven.apache.org </ url>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
  </依赖>

  <建立>
    <滤波器>
      <filter> src / main / filters / filter.properties </ filter>
    </过滤器>
    <资源>
      <资源>
        <directory> src / main / resources </ directory>
        <filtering> true </ filtering>
      </资源>
    </资源>
  </建造>
</项目>
```

然后，如果我们在`application.properties`文件中添加对此属性的引用：

```
# application.properties
申请。name = $ { project 。名字}
申请。version = $ { project 。版本}
message = $ { my 。过滤器。价值}
```

下一次执行`mvn process-resources`命令会将我们的新属性值放入`application.properties`中。作为在外部文件中定义my.filter.value属性的替代方法，您也可以在`pom.xml`的`属性`部分中定义它并获得相同的效果（注意我不需要引用`src / main / filters / filter.properties`也是）：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi: schemaLocation = “http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 罐子</ packaging>

  <name> Maven Quick Start Archetype </ name>
  <url> http://maven.apache.org </ url>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
  </依赖>

  <建立>
    <资源>
      <资源>
        <directory> src / main / resources </ directory>
        <filtering> true </ filtering>
      </资源>
    </资源>
  </建造>

  <性能>
    <my.filter.value> hello </my.filter.value>
  </属性>
</项目>
```

过滤资源也可以从系统属性中获取值; Java中内置的系统属性（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。要继续该示例，让我们将`application.properties`文件更改为如下所示：

```
# application.properties
java。version = ${ java。版本}
```

```
3. 命令。线。prop = $ { command 。线。支撑}
```

现在，当您执行以下命令时（请注意命令行上command.line.prop属性的定义），`application.properties`文件将包含系统属性中的值。

```
1. mvn进程- 资源“-Dcommand.line.prop =你好再次”
```

## 我如何使用外部依赖项？

您可能已经注意到我们一直在使用的POM中的`依赖`元素作为示例。事实上，你一直在使用外部依赖，但是在这里我们将更详细地讨论它是如何工作的。有关更全面的介绍，请参阅我们的依赖机制简介。

pom.xml 的`dependencies`部分列出了我们的项目为了构建而需要的所有外部依赖项（在编译时是否需要该依赖项，测试时间，运行时间等等）。现在，我们的项目仅依赖于JUnit（为了清楚起见，我拿出了所有的资源过滤内容）：

```
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.
7.   <groupId> com.mycompany.app </ groupId>
8.   <artifactId> my-app </ artifactId>
9.   <version> 1.0-SNAPSHOT </ version>
10.  <包装> 罐子</ packaging>
11.
12.  <name> Maven Quick Start Archetype </ name>
13.  <url> http://maven.apache.org </ url>
14.
15.  <依赖性>
16.    <依赖性>
17.      <groupId> junit </ groupId>
18.      <artifactId> junit </ artifactId>
19.      <version> 4.11 </ version>
20.      <scope> test </ scope>
21.    </依赖性>
22.  </依赖>
23. </项目>
```

对于每个外部依赖项，您需要至少定义4个内容：groupId，artifactId，version和scope。groupId，artifactId和version与构建该依赖项的项目的`pom.xml中`给出的相同。scope元素指示项目如何使用该依赖项，并且可以是诸如`compile`，`test`和`runtime之类的值`。有关可以为依赖项指定的所有内容的更多信息，请参阅“项目描述符参考”。

有关作为整体的依赖性机制的更多信息，请参阅依赖性机制简介。

有了依赖关系的这些信息，Maven将能够在构建项目时引用依赖关系。Maven在哪里引用依赖关

系？Maven在您的本地存储库中查找（`$ {user.home} /.m2 / repository`是默认位置）以查找所有依赖项。在上一节中，我们将项目（my-app-1.0-SNAPSHOT.jar）中的工件安装到本地存储库中。一旦它安装在那里，另一个项目可以简单地通过将依赖性信息添加到其pom.xml来引用该jar作为依赖：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi: schemaLocation = “http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-other-app </ artifactId>
  ...
  <依赖性>
    ...
    <依赖性>
      <groupId> com.mycompany.app </ groupId>
      <artifactId> my-app </ artifactId>
      <version> 1.0-SNAPSHOT </ version>
      <scope> 编译</ scope>
    </依赖性>
  </依赖>
</项目>
```

在其他地方构建的依赖项怎么样？他们如何进入我的本地存储库？每当项目引用本地存储库中不可用的依赖项时，Maven就会将依赖项从远程存储库下载到本地存储库中。您可能已经注意到Maven在构建您的第一个项目时下载了很多东西（这些下载是用于构建项目的各种插件的依赖项）。默认情况下，可以在http://repo.maven.apache.org/maven2/找到（并浏览）Maven使用的远程存储库。您也可以使用自己的远程存储库（可能是公司的中央存储库）来代替默认远程存储库或使用默认远程存储库。有关存储库的更多信息，请参阅存储库简介.

让我们为我们的项目添加另一个依赖项。假设我们在代码中添加了一些日志记录，需要添加log4j作为依赖项。首先，我们需要知道log4j的groupId，artifactId和version是什么。Maven Central上的相应目录称为/ maven2 / log4j / log4j。在该目录中是一个名为maven-metadata.xml的文件。这是log4j的maven-metadata.xml的样子：

```
<元数据>
  <groupId> log4j </ groupId>
  <artifactId> log4j </ artifactId>
  <version> 1.1.3 </ version>
  <版本>
    <版本>
      <version> 1.1.3 </ version>
      <version> 1.2.4 </ version>
      <version> 1.2.5 </ version>
      <version> 1.2.6 </ version>
      <version> 1.2.7 </ version>
      <version> 1.2.8 </ version>
      <version> 1.2.11 </ version>
      <version> 1.2.9 </ version>
```

```
      <version> 1.2.12 </ version>
    </版本>
  </版本管理>
</元数据>
```

从这个文件中，我们可以看到我们想要的groupId是“log4j”，artifactId是“log4j”。我们看到许多不同的版本值可供选择; 目前，我们只使用最新版本1.2.12（某些maven-metadata.xml文件也可能指定哪个版本是当前版本）。除了maven-metadata.xml文件，我们还可以看到与log4j库的每个版本对应的目录。在其中的每一个中，我们将找到实际的jar文件（例如log4j-1.2.12.jar）以及一个pom文件（这是依赖项的pom.xml，表明它可能具有的任何进一步的依赖性和其他信息）和另一个maven-metadata.xml文件。还有一个与每个文件对应的md5文件，其中包含这些文件的MD5哈希值。

现在我们知道了我们需要的信息，我们可以将依赖项添加到我们的pom.xml：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi: schemaLocation = “http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 罐子</ packaging>

  <name> Maven Quick Start Archetype </ name>
  <url> http://maven.apache.org </ url>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
    <依赖性>
      <groupId> log4j </ groupId>
      <artifactId> log4j </ artifactId>
      <version> 1.2.12 </ version>
      <scope> 编译</ scope>
    </依赖性>
  </依赖>
</项目>
```

现在，当我们编译项目（`mvn compile`）时，我们将看到Maven为我们下载了log4j依赖项。

## 如何在远程存储库中部署 jar？

要将jar部署到外部存储库，您必须在pom.xml中配置存储库URL，并在settings.xml中配置连接到存

储库的身份验证信息。

以下是使用scp和用户名/密码身份验证的示例：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi：schemaLocation = “http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 罐子</ packaging>

  <name> Maven Quick Start Archetype </ name>
  <url> http://maven.apache.org </ url>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
    <依赖性>
      <groupId> org.apache.codehaus.plexus </ groupId>
      <artifactId> plexus-utils </ artifactId>
      <version> 1.0.4 </ version>
    </依赖性>
  </依赖>

  <建立>
    <滤波器>
      <filter> src / main / filters / filters.properties </ filter>
    </过滤器>
    <资源>
      <资源>
        <directory> src / main / resources </ directory>
        <filtering> true </ filtering>
      </资源>
    </资源>
  </建造>
  <！ -
   |
   |
   |
   - >
  <distributionManagement>
```

```
    <库>
      <id> mycompany-repository </ id>
      <name> MyCompany Repository </ name>
      <url> scp: //repository.mycompany.com/repository/maven2 </ url>
    </存储库>
  </ distributionManagement>
</项目>
```

```
<settings xmlns = “http://maven.apache.org/SETTINGS/1.0.0”
  xmlns: xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi: schemaLocation = “http://maven.apache.org/SETTINGS/1.0.0
                       http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <服务器>
    <服务器>
      <id> mycompany-repository </ id>
      <username> jvanzyl </ username>
      <! - 默认值为~/ .ssh / id_dsa - >
      <privateKey> / path / to / identity </ privateKey> （默认为~/ .ssh / id_dsa）
      <passphrase> my_key_passphrase </ passphrase>
    </服务器>
  </服务器>
  ...
</设置>
```

请注意，如果要连接到在sshd_confing中将参数“PasswordAuthentication”设置为“no”的openssh ssh服务器，则每次进行用户名/密码验证时都必须键入密码（尽管可以使用其他ssh登录）客户端输入用户名和密码）。在这种情况下，您可能希望切换到公钥身份验证。

如果在`settings.xml`中使用密码，则应该小心。有关更多信息，请参阅密码加密。

## 我如何创建文档？

为了让您从Maven的文档系统开始，您可以使用原型机制使用以下命令为现有项目生成站点：

```
mvn原型: 生成\
  - DarchetypeGroupId = org 。阿帕奇。maven 。原型\
  - DarchetypeArtifactId = maven - 原型- 网站\
  - DgroupId = com 。我的公司。app
  - DartifactId = 我的- 应用程序- 网站
```

现在，请转到指南，创建一个站点，以了解如何为项目创建文档。

## 如何构建其他类型的项目？

请注意，生命周期适用于任何项目类型。例如，回到基本目录，我们可以创建一个简单的Web应用程序：

```
mvn原型：生成\
    - DarchetypeGroupId = org 。阿帕奇。maven 。原型\
    - DarchetypeArtifactId = maven - archetype - webapp \
    - DgroupId = com 。我的公司。app
    - DartifactId = 我的- webapp
```

请注意，这些必须全部在一行上。这将创建一个名为`my-webapp`的目录，其中包含以下项目描述符：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi：schemaLocation = “http://maven.apache.org/POM/4.0.0
                       http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> my-webapp </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> 战争</ packaging>

  <依赖性>
    <依赖性>
      <groupId> junit </ groupId>
      <artifactId> junit </ artifactId>
      <version> 4.11 </ version>
      <scope> test </ scope>
    </依赖性>
  </依赖>

  <建立>
    <finalName> my-webapp </ finalName>
  </建造>
</项目>
```

注意`<packaging>`元素 - 这告诉Maven构建为WAR。转到webapp项目的目录并尝试：

```
mvn 包
```

您将看到`target / my-webapp.war`已构建，并且已执行所有正常步骤。

## 如何一次构建多个项目？

处理多个模块的概念内置于Maven中。在本节中，我们将展示如何构建上面的WAR，并在一个步骤中包含先前的JAR。

首先，我们需要在其他两个上面的目录中添加父`pom.xml`文件，所以它应该如下所示：

```
+  - pom 。XML
+  - 我的- 应用程序
                。
```

```
| +  - pom   XML
| +  - src
| +  - 主要
| +  - java
+  - 我的- webapp
| +  - pom 。XML
| +  - src
| +  - 主要
| +  - webapp
```

您将创建的POM文件应包含以下内容：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi：schemaLocation = “http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion> 4.0.0 </ modelVersion>

  <groupId> com.mycompany.app </ groupId>
  <artifactId> app </ artifactId>
  <version> 1.0-SNAPSHOT </ version>
  <包装> pom </ packaging>

  <模块>
    <module> my-app </ module>
    <module> my-webapp </ module>
  </模块>
</项目>
```

我们需要从webapp依赖JAR，所以将它添加到`my-webapp / pom.xml`：

```
  ...
  <依赖性>
    <依赖性>
      <groupId> com 。我的公司。app < / groupId>
      <artifactId> my-app </ artifactId >
      <version> 1.0 - SNAPSHOT < / version>
    </ dependency >
    ...
  </ dependencies >
```

最后，将以下`<parent>`元素添加到子目录中的其他两个`pom.xml`文件中：

```
<project xmlns = “http://maven.apache.org/POM/4.0.0”
  xmlns：xsi = “http://www.w3.org/2001/XMLSchema-instance”
  xsi：schemaLocation = “http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <母体>
    <groupId> com.mycompany.app </ groupId>
```

```
    <artifactId> app </ artifactId>
    <version> 1.0-SNAPSHOT </ version>
  </父>
  ...
```

现在，尝试它...从顶级目录，运行：

```
mvn验证
```

WAR现在已在`my-webapp / target / my-webapp.war中创建`，并且包含JAR：

```
$ jar tvf my - webapp / target / my - webapp - 1.0 - SNAPSHOT 。战争
   0 周五君24 10 : 59 : 56 美国东部时间2005年META - INF /
 222 至周五6月的24 10 : 59 : 54 EST 2005 META - INF / 清单。MF
   0 周五6月的24 10 : 59 : 56 EST 2005 META - INF / 行家/
   0 周五6月的24 10 : 59 : 56 EST 2005 META - INF / 行家/ 作为。MyCompany的。app /

   0 周五君24 10 : 59 : 56 美国东部时间2005年META - INF / 行家/ COM 。我的公司。app / my - webapp /
3239 周五君24 10 : 59 : 56 美国东部时间2005年META - INF / 行家/ COM 。我的公司。app / my - webapp / pom 。XML
   0 周五君24 10 : 59 : 56 美国东部时间2005年WEB - INF /
 215 周五君24 10 : 59 : 56 美国东部时间2005年WEB - INF / 网页。XML
 123 至周五6月的24 10 : 59 : 56 EST 2005 META - INF / 行家/ COM 。我的公司。app / my - webapp / pom 。性能
  52 星期五君24 10 : 59 : 56 美国东部时间2005年指数。JSP
   0 周五6月的24 10 : 59 : 56 EST 2005 WEB - INF / LIB /
2713 周五君24 10 : 59 : 56 美国东部时间2005年WEB - INF / lib中/ 我- 应用程序- 1.0 - 快照。罐
```

这是如何运作的？首先，父POM创建（称为`app`），包含`pom`和定义的模块列表。这告诉Maven在项目集上运行所有操作而不仅仅是当前项目（要覆盖此行为，可以使用`--non-recursive`命令行选项）。

接下来，我们告诉WAR它需要`my-app` JAR。这样做有以下几点：它使类路径中的任何代码都可用于WAR中的任何代码（在本例中为none），它确保JAR始终在WAR之前构建，并且它指示WAR插件包含JAR in它的库目录。

您可能已经注意到`junit-4.11.jar`是一个依赖项，但没有在WAR中结束。其原因是`<scope> test </ scope>`元素 - 它仅用于测试，因此不包含在Web应用程序中作为编译时依赖性`my-app`。

最后一步是包含父定义。这与您在Maven 1.0中可能熟悉的`extend`元素不同：这确保了即使项目通过在存储库中查找而与其父项分开分配，也可以始终定位POM。