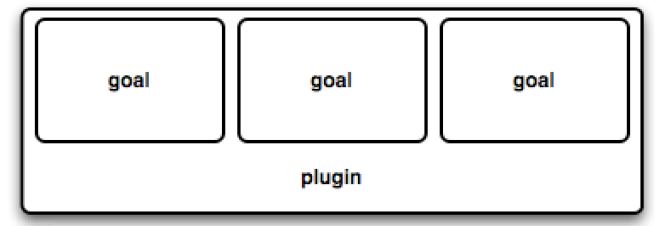
# 插件和目标：

## 1.1 插件

一个Maven插件是一个单个或者多个目标的集合。

Maven插件的例子有一些简单但核心的插件，像Jar插件，它包含了一组创建Jar文件的目标。Compiler插件，它包含了一组编译源代码和测试代码的目标，或者surefire插件，它包含一组运行单元测试和生成测试报告的目标。



## 1.2 目标：

一个目标是一个明确的任务，它可以作为单独的目标运行，或者作为一个大的构建的一部分和其它目标一起执行。一个目标是Maven中的一个工作单元（unit of work）。

# 生命周期：

**生命周期是包含在一个项目构建中的一系列有序的阶段**。

Maven可以支持许多不同的生命周期，但是最常用的生命周期就是默认的生命周期。

插件目标可以附着在生命周期阶段上。随着Maven沿着生命周期的阶段移动，它会执行附着在特定阶段上的目标。

每个阶段可能绑定了零个或多个目标。

# Maven坐标：

## 3.1 坐标：

|  |
| --- |
| <groupId>org.snoatype.nexus</groupId>  <artifactId>nexus-indexer</artifactId>  <version>2.0.0</version>  <packaging>jar</packaging> |

### groupId：

定义当前Maven项目隶属的实际项目。

首先，Maven项目和实际项目不一定是一对一的关系。比如Spring Framework这一实际项目，其对应的Maven项目会有很多，如spring-core、spring-context等。这是由于Maven中模块的概念，因此，一个实际项目往往会被划分成很多模块。

其次，groupId不应该对应项目隶属的组织或者公司。原因很简单，一个组织下会有很多实际项目，如果groupId只定义到组织级别，而后面我们会看到，artifactId只能定义Maven项目（模块），那么实际项目这个层次将会很难定义。

最后，groupId的表示方式与Java包名的表示方式类似，通常与域名反向一一对应。上例中，groupId为org.sonatype.nexus，org.sonatype表示Sonatype公司简历的一个非盈利组织，nexus表示Nexus这一实际项目，该groupId与域名nexus.sonatype.org对应。

### artifactId：

该元素定义实际项目中的一个Maven项目（模块），推荐的做法是使用实际项目名称作为artifactId的前缀。

比如上例中的artifactId是nexus-indexer，使用了实际项目名nexus作为前缀，这样做的好处是方便寻找实际构件。在默认情况下，Maven生成的构件，其文件名会以artifactId作为开头，如nexus-indexer-2.0.0.jar，使用实际项目名称作为前缀之后，就能方便从一个lib文件夹中找到某个项目的一组构件。

考虑有5个项目（实际项目），每个项目都有一个core模块，如果没有前缀，我们会看到很多core-1.2.jar这样的文件，加上实际项目名前缀之后，便能很容易区分foo-core-1.2.jar、bar-core.1.2.jar……

### version：

该元素定义Maven项目当前所处的版本，如上例中nexus-indexer的版本是2.0.0。需要注意的是，Maven定义了一套完整的版本规范，以及快照（SNAPSHOT）的概念。

### packaging：

该元素定义Maven项目的打包方式。

首先，打包方式通常与所生成构件的文件扩展名对应，如上例中packaging为jar，最终的文件名为nexus-indexer-2.0.0.jar，而使用war打包方式的Maven项目，最终生成的构件会有一个.war文件，不过这不是绝对的。

其次，打包方式会影响到构建的生命周期，比如jar打包和war打包会使用不同的命令。

最后，当不定义packaging的时候，Maven会使用默认值jar。

### classfier：

该元素用来帮助定义构建输出的一些附属信息。

附属构件与主构件对应，如上例中的主构件是nexus-indexer-2.0.0.jar，该项目可能还会通过使用一些插件生成如nexus-indexer-2.0.0-javadoc.jar、nexus-indexer-2.0.0-sources.jar这样一些附属构件，其中包含了Java文档和源代码。这时候，javadoc和sources就是两个附属构件的classfier。这样，附属构件也就拥有了自己唯一的坐标。

还有一个关于classfier的典型例子是TestNG，TestNG的主构件是基于java1.4平台的，而它又提供了一个classfier为jdk5的附属构件。

注意，不能直接定义项目的classfier，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成的。

上述5个元素中，groupId、artifactId、version是必须定义的，packaging是可选的（默认为jar），而classfier是不能直接定义的。

同时，项目构件的文件名是与坐标相对应的，一般的规则为artifactId-version-[classfier].packaging。

这里还要强调一点，packaging并非一定与构件拓展名对应，比如packaging为maven-plugin的构件拓展名为jar。

## 3.2 版本：

### 3.2.1 项目版本：

一个Maven项目发布版本号用version编码，用来分组和排序。Maven中的版本包含了以下部分：主版本，此版本，增量版本，限定版本。

一个版本中，这些部分对应如下的格式：

|  |
| --- |
| <major version>.<minor version>.<incremental version>-<qulifier> |

例如：版本“1.3.5”由一个主版本1，一个次版本3，和一个增量版本5组成。而一个版本“5”只有主版本5，没有次版本和增量版本。

限定版本用来标识里程碑构建：alpha和beta发布，限定版本号通过**连字符**与版本号，次版本，或增量版本隔离。例如，版本“1.3-beta-01”有一个主版本1，次版本3，和一个beta限定版本“beta-01”。

如果你的版本号与格式<主版本>.<次版本>.<增量版本>-<限定版本>相匹配，它就能被正确的比较：“1.2.3”将被评价成是一个比“1.0.2”更新的构件，这种比较基于主版本、次版本，和增量版本的数值。如果你的版本发布号没有符合本节介绍的标准，那么你的版本号只会根据字符串比较：“1.0.1b”和“1.2.1b”会使用字符串比较。

### 3.2.2 版本构建号：

我们还需要对版本号的限定版本进行排序。以版本号“1.2.3-alpha-2”和“1.2.3-alpha-10”为例，这里“alpha-2”对应了第二次alpha构建，而“alpha-10”对应了第十次alpha构建。虽然我们认为“alpha-10”应该是比“alpha-2”更新的构建，但Maven排序的结果是“alpha-10”比“alpha-2”更旧，问题的原因就是我们刚才讨论的Maven处理版本号的方式。

Maven会将限定版本后面的数字认为一个构建版本。换句话说，这里限定版本是“alpha”，而构建版本是2，虽然Maven被设计成构建版本和限定版本分离，但目前这种解析还是失效的。因此，**“alpha-2”和“alpha-10”是使用字符串进行比较的**，而根据ASCII码排序规则“alpha-2”在“alpha-10”前面。要避开这种限制，你需要你的限定版本号使用一些技巧。如果你使用“alpha-02”和“alpha-10”，这个问题就消失了，一旦Maven能正确的解析版本构建号之后，这种工作方式也还能用。

### 3.2.3 SNAPSHOT版本：

Maven版本可以包含一个字符串字面量来表示项目正处于活动的开发状态。**如果一个版本号包含字符串“SNAPSHOT”，Maven就会在安装或发布这个组件的时候将该符号开展为一个日期和时间值，转换为UTC(协调世界时)**。

例如，如果你的项目有个版本为“1.0-SNAPSHOT”并且将你这个项目的构建部署到了一个Maven仓库，如果你在UTC时间2008年2月7号下午11:08部署了这个版本，Maven就会将这个版本展开为“1.0-20080207-230803-1”。换句话说，当你发布一个SNAPSHOT，你没有发布一个软件模块，你只是发布了一个特定时间的快照。

【注】发布到REALEASE仓库的构件不能依赖于任何SNAPSHOT版本，因为Maven的超级POM对于中央仓库关闭了SNAPSHOT。

### 3.2.4 LATEST和REALEASE版本：

当你依赖一个插件或一个依赖，你可以使用特殊的版本值LATEST或者RELEASE。

LATEST是指某个特定构件最新的发布版或者快照版，最近被部署到某个特定仓库的构建。

RELEASE是指仓库中最后一个非快照版本。

总的来说，设计软件去依赖一个构建的不明确版本，并不是一个好的实践。

【参考】https://www.cnblogs.com/huang0925/p/5169624.html

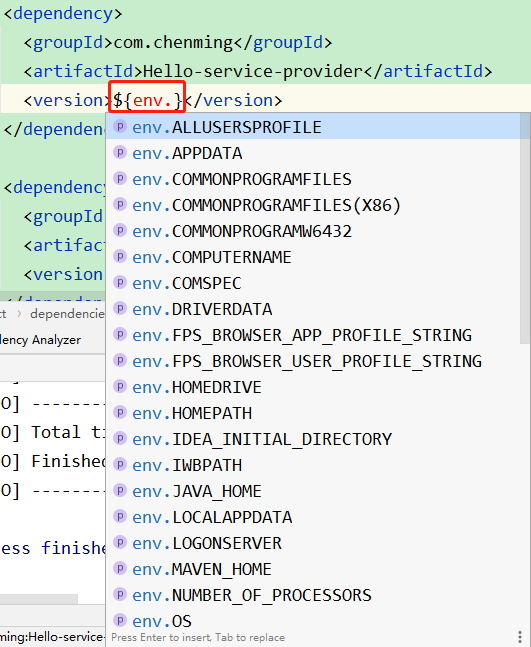
# 属性引用：

一个POM可以通过一对大括号和前面一个美元符号来包含对属性的引用。

Maven提供了三个隐式变量，可以用来访问：环境变量、POM信息、Maven Settings

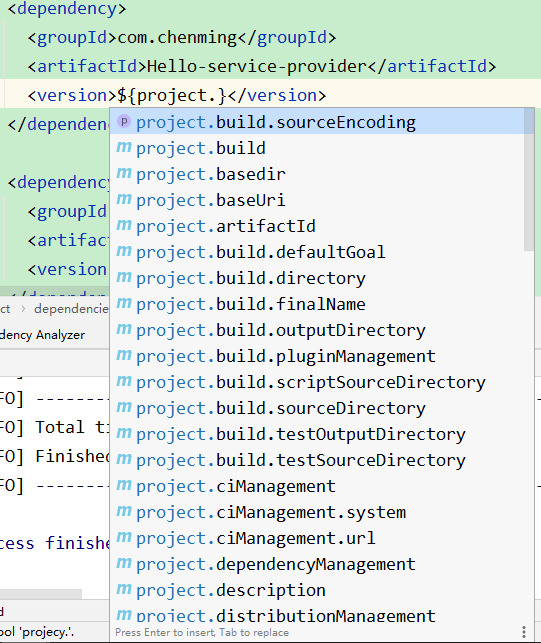
## env：

env变量暴露了你操作系统或者shell的环境变量。



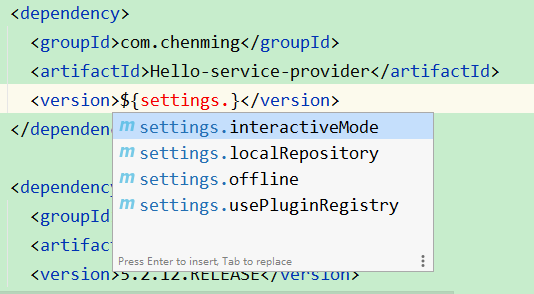
## 4.2 project：

project变量暴露了POM设置。



## 4.3 settings：

settings变量暴露了Maven settings信息。



## 4.4 其他属性：

除了上面三个隐式的变量，我们还可以引用系统属性，以及任何在Maven POM中和构建profile中自定义的属性组。

# 项目依赖：

当为项目打包(package)创建Jar文件的时候，它的依赖不会被捆绑在生成的构件中，它们只是用来编译。

当为项目打包(package)创建War或者Ear是，它的依赖会被捆绑在生成的构件中，当然你也可以配置provided范围，让它排除War文件中特定的依赖。

## 依赖范围：

首先需要知道，Maven在编译项目主代码的时候需要使用一套classpath。在上例中，编译项目主代码的时候需要用到spring-core，该文件以以来的方式被引入到classpath中。 其次，Maven在编译和执行测试的时候会使用另外一套classpath。上例中的JUnit就是一个很好的例子，该文件也以依赖的方式引入到测试使用的classpath中，不同的是这里的依赖范围是test。

最后，实际运行Maven项目的时候，又会使用一套classpath，上例中的spring-core需要在该classpath中，而JUnit则不需要。

依赖范围就是用来控制依赖与这三种classpath（编译classpath、测试classpath、运行classpath）的关系，Maven有以下集中依赖范围。

### 5.1.1 compile（编译范围）：

compile是默认的范围，如果没有提供一个范围，那么该依赖的范围就是编译范围。编译范围依赖在所有的classpath中可用，同时它们也会被打包。

### 5.1.2 provided（已提供范围）：

provided依赖只有在当JDK或者一个容器已提供该依赖之后才使用。例如，如果你开发一个web项目，你可能在编译classpath中需要可用的Servlet API来编译一个servlet，但是你不会想要在打包好的WAR中包含这个Servlet API；这个Servlet API JAR由你的应用服务器或者servlet容器提供。

已提供范围的依赖在编译classpath（不是运行时）可用，它们不是传递性的，也不会被打包。

### 5.1.3 runtime（运行时范围）：

runtime依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如，你可能在编译的时候只需要JDBC JAR API，而只有在运行的时候才需要JDBC驱动。

### 5.1.4 test（测试范围）：

test范围在一般的编译和运行都不需要，它们只有在测试编译和测试运行阶段可用。test范围的依赖并不会被打包。

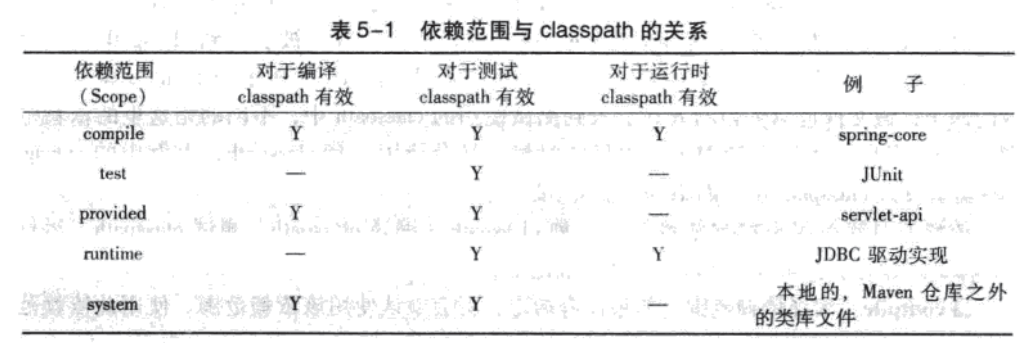
### 5.1.5 system（系统范围）：

system范围依赖于provided类似，但是你必须显式的提供一个对于本地系统中JAR文件的路径。这么做是为了允许基于本地对象编译，而这些对象是系统类库的一部分。这样的构件应该是一直可用的，Maven也不会在仓库中去寻找它。如果你将一个依赖设置成system范围，你必须同时提供一个systemPath元素。注意该范围是不推荐使用的（你应该）

### 5.1.6 import（导入依赖范围）：

该依赖范围不会对三种classpath产生实际的影响。

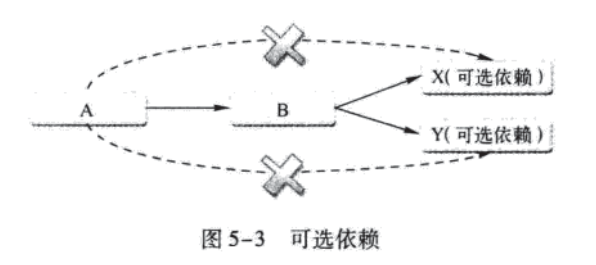
各种依赖范围与三种classpath的关系如表所示：



## 可选依赖：

【参考】[Maven optional关键字透彻图解 - 日拱一兵 - 博客园 (cnblogs.com)](https://www.cnblogs.com/FraserYu/p/11796301.html)

假设有这样一个依赖关系，项目A依赖于项目B，项目B依赖于项目X和项目Y，B对于X和Y的依赖都是可选的：A -> B、B ->X(可选)、B -> Y(可选)。根据传递性依赖的定义，如果所有这三个依赖的范围都是compile，那么X、Y就是A的compile范围传递性依赖。然而，由于这里X、Y是可选依赖，依赖将不会得以传递。换句话说，X、Y将不会对A由任何影响，如图5-3所示：



为什么要使用可选依赖这一特性呢？可能项目B实现了两个特性，其中的特性一依赖于X、特性二依赖于Y，而且这两个特性是互斥的，用户不可能同时使用这两个 特性。比如B是一个持久层隔离工具包，它支持多种数据库，包括MySQL、PostgreSQL等，在构建这个工具包的时候，需要这两种数据库的驱动程序，但在使用这个工具包的时候，指挥依赖一种数据库。

项目B的依赖声明：



上述XML代码片段中，使用<optional>元素表示mysql-connector-java和postgresql这两个依赖为可选依赖，它们只会对当前项目B产生影响，当其他项目依赖于B的时候，这两个依赖不会被传递。因此，当项目A依赖于项目B的时候，如果其实际使用基于MySQL数据库，那么在项目A中就需要显式的声明mysql-connector-java这一依赖，见代码清单：



最后，关于可选依赖需要说明的一点是，在理想的情况下，是不应该使用可选依赖的。前面我们可以看到，使用可选依赖的原因是某一个项目实现了多个特性，在面向对象设计中，有个单一职责原则，意指一个类应该只有一项职责，而不是糅合太多的功能。这个原则在规划Maven项目的时候也同样适用。

在上面的例子中，更好的做法是为MySQL和PostgreSQL分别创建一个Maven项目，基于同样的groupId分配不同的artifactId，如com.juvenxu.com.mvnbook:project-b-mysql和com.juvenxu.com.mvnbook:project-b-postgresql，在各自的POM中声明对应的JDBC驱动依赖，而且不适用可选依赖，用户则根据需要选择适用project-b-mysql或者project-b-postgresql。由于传递性依赖的作用，就不用再声明JDBC驱动依赖。

## 传递性依赖：

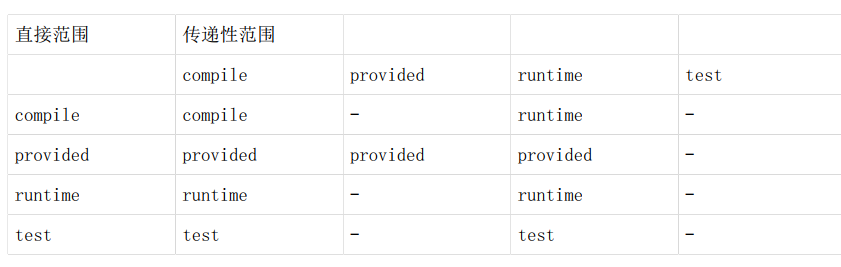
一个传递依赖就是对于一个依赖的依赖。如果project-a依赖于project-b，而后project-b依赖于project-c，那么project-c就被认为是project-a的传递性依赖。如果project-c依赖于project-d，那么project-d就也被认为是project-a的传递性依赖。

Maven的部分吸引力是由于它能够管理传递性依赖，并且能够帮助开发者屏蔽掉跟踪所有编译器和运行期依赖的细节。你可以只依赖一些包如Spring Framework，而不用担心Spring Framework的所有依赖，Maven帮你自动管理了，你不用自己去详细了解。

### 传递性依赖范围：

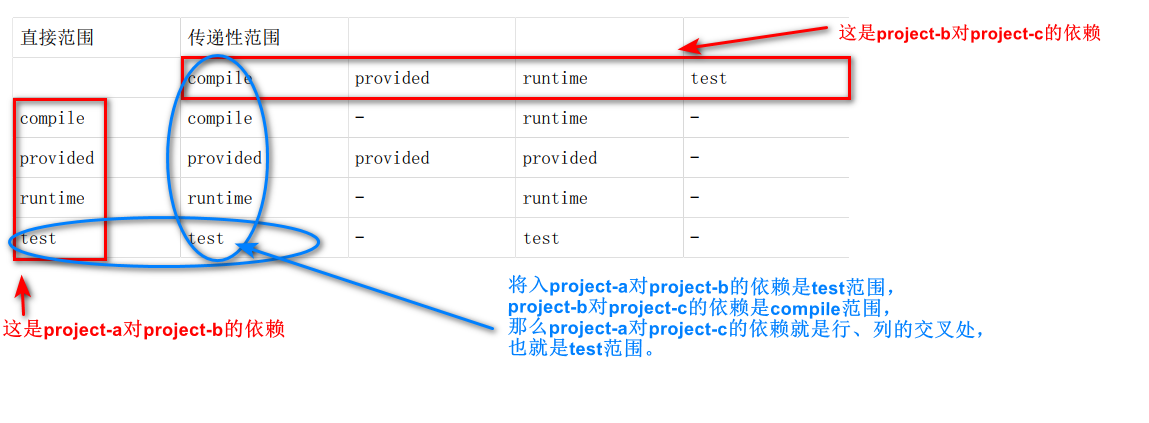
依赖范围中提到的每种依赖范围不仅仅影响声明项目中的依赖范围，它也对所传递依赖起作用。

表达信息最完整的方式是通过一张表来表述：



最顶层一行代表了传递性依赖的范围，最左边的一列代表了直接依赖的范围。行与列的交叉就是某个传递性依赖指定的范围。表中的空格意思是该传传递性依赖被忽略。

要阐明传递性依赖于直接依赖的关系，考虑如下例子。如果project-a包含一个对于project-b的测试范围依赖，project-b包含一个对project-c的编译范围依赖。project-c将是project-a的测试范围传递性依赖。



【规律】《Maven实战 P66》

当第二直接依赖的范围是compile的时候，传递性依赖的范围与第一直接依赖的范围一致；

当第二直接依赖的范围是test的时候，依赖不会得以传递；

当第二直接依赖的范围是provided的时候，只传递第一直接依赖也为provided的依赖，且传递性依赖的范围同样为provided；

当第二直接依赖的范围是runtime的时候，传递性依赖的范围与第一直接依赖的范围一致，但compile例外，此时传递性依赖的范围为runtime。

## 依赖调解：

Maven引入的传递性依赖机制，一方面大大简化和方便了依赖声明，另一方面，大部分情况下我们只需要关心项目的直接依赖是什么，而不用考虑这些直接依赖会引入什么传递性依赖。但有时候，当传递性依赖造成问题的时候，我们就要清除的知道该传递性依赖是从哪条依赖路径引入的。

例如，项目A有这样的依赖关系：A -> B -> C -> X(1.0)、A -> D -> X(2.0)，X是A的传递性依赖，但是两条依赖路径上由两个版本的X，那么哪个X会被Maven解析使用呢？两个版本都被解析显然是不对的，因为那会造成依赖重复，因此必须选择一个。Maven依赖调解（Dependency Mediation）的**第一原则**是：路径最近者优先。该例中X(1.0)的路径长度为3，而X(2.0)的路径长度为2，因此X(2.0)会被解析使用。

依赖调解第一原则不能解决所有的问题，比如这样的依赖关系：A -> B -> Y(1.0)、A -> C -> Y(2.0)，Y(1.0)和Y(2.0)的依赖路径长度是一样的，都为2。那么到底谁会被解析使用呢？在Maven2.0.8及之前的版本中，这是不确定的，但是从Maven2.0.9开始，为了尽可能避免构建的不确定性，Maven定义了依赖的**第二原则**：第一声明者优先。在依赖路径长度相等的前提下，在POM中依赖声明的顺序决定了谁会被解析使用，顺序最靠前的哪个依赖优胜。该例中，如果B的依赖声明在C之前，那么Y(1.0)就会被解析使用。