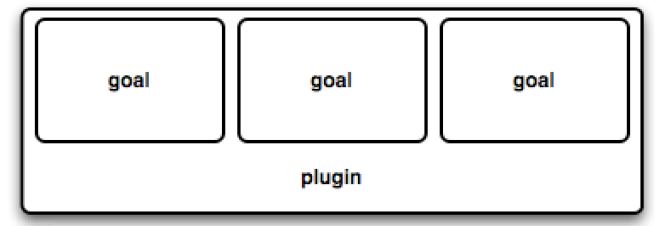
# 插件和目标：

## 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：

1）groupId：公司ID或项目要求唯一，一般是域名的倒写(com.alibaba)，或者域名加上项目名(com.alibaba.driud)，因为域名是唯一的。

2）artifactId：一般是项目名或者子项目名。

3）version：版本号

<groupId>、<artifactId>、<version>三个属性共同组成构件的坐标。

## 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

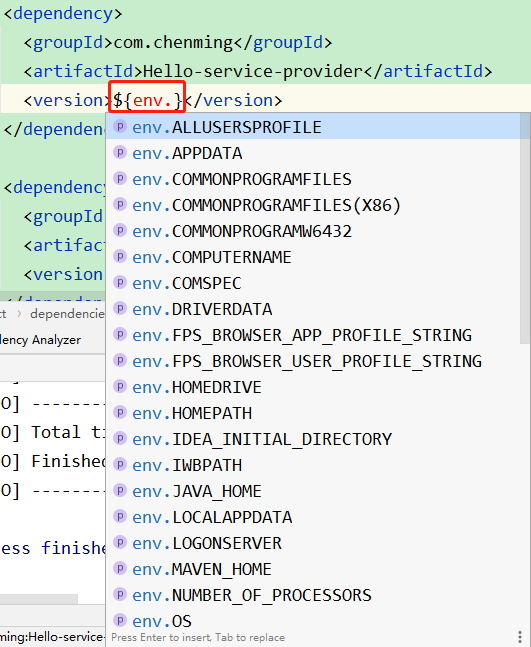
# 属性引用：

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

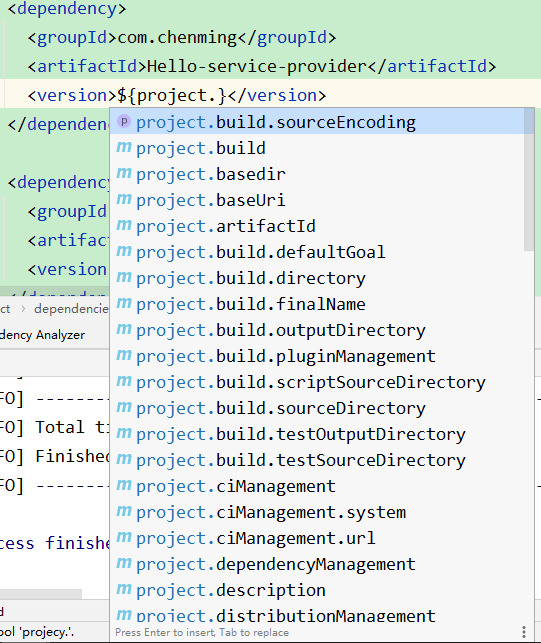
## env：

env变量暴露了你操作系统或者shell的环境变量。



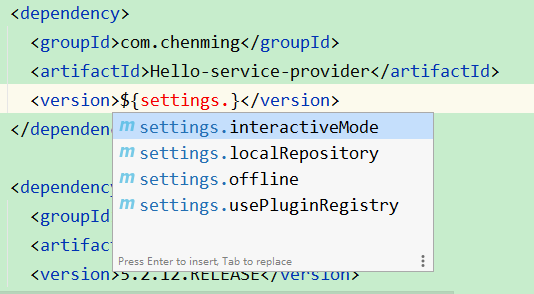
## 4.2 project：

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：

### 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）

## 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)