# Maven简介

相关教程地址：http://blog.csdn.net/u012152619/article/details/51473410

## 定义

Maven是一个项目管理工具，它包含了一个项目对象模型 (POM：Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。

## Maven的作用

1. 构建工程，合理叙述项目间的依赖关系，管理各种引用的jar包，防止发生冲突。
2. 编译代码，还能帮你自动运行单元测试，打包代码，生成报表。
3. 帮你部署项目，生成Web站点。
4. 连接**开发测试部署**，将各个彼此独立的过程打通的一个自动化工具。

# Maven的生命周期

## 定义

Maven的生命周期就是对所有构建过程抽象与统一。Maven有**三套相互独立**的生命周期：

* CleanLifecycle 在进行真正的构建之前进行一些清理工作。
* DefaultLifecycle 构建的核心部分，编译，测试，打包，部署等等。
* SiteLifecycle 生成项目报告，站点，发布站点。

## Maven的调度机制

每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。maven中所有的执行动作(goal)都需要指明自己在这个过程中的执行位置，然后maven执行的时候，就依照过程的发展依次调用这些goal进行各种处理。

## Clean生命周期：

1. pre-clean ：执行一些需要在clean之前完成的工作。
2. clean ：移除所有上一次构建生成的文件。
3. post-clean ：执行一些需要在clean之后立刻完成的工作。

* 在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行。也就是说，“mvn clean” 等同于 “mvn pre-clean clean”

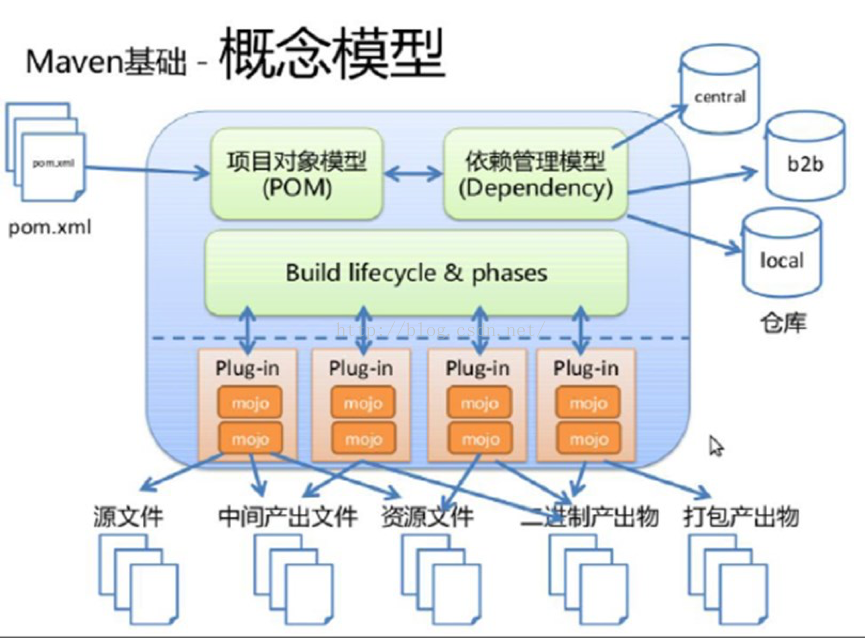
## Default生命周期 – 构建生命周期：

1. validat ：验证项目是否正确，以及所有为了完整构建必要的信息是否可用
2. generate-sources ：生成所有需要包含在编译过程中的源代码
3. process-sources ：处理源代码，比如过滤一些值
4. generate-resources ：生成所有需要包含在打包过程中的资源文件
5. **\*process-resources**  ：复制并处理资源文件至目标目录，准备打包
6. compile ：编译项目的源代码
7. process-classes ：后处理编译生成的文件，例如对Java类进行字节码增强（bytecode enhancement）
8. generate-test-sources ：生成所有包含在测试编译过程中的测试源码
9. **\*process-test-sources ：**处理测试源码，比如过滤一些值
10. generate-test-resources ：生成测试需要的资源文件
11. process-test-resources ：复制并处理测试资源文件至测试目标目录
12. test-compile ：编译测试源码至测试目标目录
13. **\*test ：**使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布
14. prepare-package ：在真正的打包之前，执行一些准备打包必要的操作
15. **\*package ：**将编译好的代码打包成可分发的格式，如JAR，WAR，或者EAR
16. pre-integration-test：执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境
17. integration-test ：如果有必要的话，处理包并发布至集成测试可以运行的环境
18. post-integration-test ：执行一些在集成测试运行之后需要的动作。如清理集成测试环境。
19. verify ：执行所有检查，验证包是有效的，符合质量规范
20. **\*install ：**安装包至本地仓库，以备本地的其它项目作为依赖使用
21. **\*deploy ：**复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关）

## Site生命周期

1. pre-site ：执行一些需要在生成站点文档之前完成的工作。
2. **\*site ：**生成项目的站点文档。
3. post-site ：执行一些需要在生成站点文档之后完成的工作，并且为部署做准备。
4. **\*site-deploy ：**将生成的站点文档部署到特定的服务器上。

# Maven插件



## 定义

Maven本质上是一个插件框架，它的**核心并不执行任何具体的构建任务**，所有这些任务都交给插件来完成，像编译是通过maven-compile-plugin实现的、测试是通过maven-surefire-plugin实现的，maven也内置了很多插件，所以我们在项目进行编译、测试、打包的过程是没有感觉到。

## 插件作用

进一步说，每个任务对应了一个插件目标（goal），每个插件会有一个或者多个目标，例如maven-compiler-plugin的compile目标用来编译位于src/main/java/目录下的主源码，testCompile目标用来编译位于src/test/java/目录下的测试源码。

## 插件列表

第一个列表的GroupId为org.apache.maven.plugins，这里的插件最为成熟，具体地址为：<http://maven.apache.org/plugins/index.html>。

第二个列表的GroupId为org.codehaus.mojo，这里的插件没有那么核心，但也有不少十分有用，其地址为：http://mojo.codehaus.org/plugins.html。

## Maven提供以下两种类型的插件

构建插件 ：在生成过程中执行，并应在pom.xml中的<build/>元素进行配置

报告插件 ：在网站生成期间执行的，应该在pom.xml中的<reporting/>元素进行配置。

一个插件通常提供了一组目标，可使用以下语法来执行：

mvn [plugin-name]:[goal-name]

例如，一个Java项目使用了编译器插件，通过运行以下命令编译

mvn compiler:compile

# Maven常用命令

## Maven的命令格式

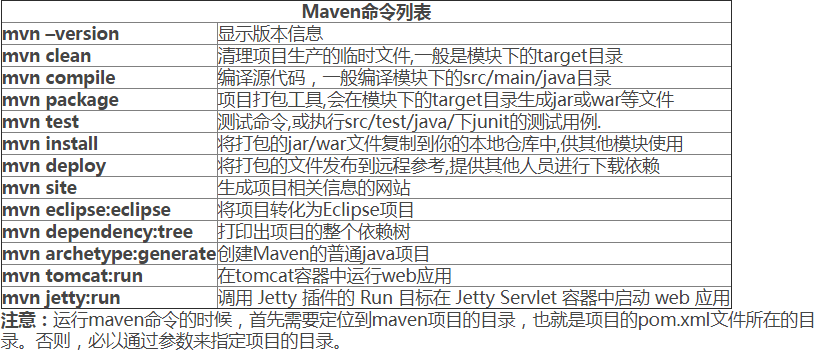
mvn [plugin-name]:[goal-name]

表示 ：执行“plugin-name”插件的“goal-name”目标（或者称为动作）。

## 调用插件的两种方式：

* 第一种方式是将插件目标与生命周期阶段（lifecycle phase）绑定，这样用户在命令行只是输入生命周期阶段而已，例如Maven默认将maven-compiler-plugin的compile目标与compile生命周期阶段绑定，因此命令mvn compile实际上是先定位到compile这一生命周期阶段，然后再根据绑定关系调用maven-compiler-plugin的compile目标。
* 第二种方式是直接在命令行指定要执行的插件目标，例如mvnarchetype:generate 就表示调用maven-archetype-plugin的generate目标，这种带冒号的调用方式与生命周期无关。

## Maven命令列表



## 命令参数

1. - D 传入属性参数

比如：

mvn package -D maven.test.skip = true

以“-D”开头，将“maven.test.skip”的值设为“true”,就是告诉maven打包的时候跳过单元测试。同理，“mvn deploy-Dmaven.test.skip=true”代表部署项目并跳过单元测试。

1. - P 使用指定的Profile配置

比如项目开发需要有多个环境，一般为开发，测试，预发，正式4个环境，

profiles定义了各个环境的变量id，filters中定义了变量配置文件的地址，其中地址中的环境变量就是上面profile中定义的值，resources中是定义哪些目录下的文件会被配置文件中定义的变量替换。

通过maven可以实现按不同环境进行打包部署，命令为: mvn package -P dev

其中“dev“为环境的变量id,代表使用Id为“dev”的profile。

1. -e 显示maven运行出错的信息。
2. -o 离线执行命令,即不去远程仓库更新包。
3. -X 显示maven允许的debug信息。
4. -U 强制去远程更新snapshot的插件或依赖，默认每天只更新一次。

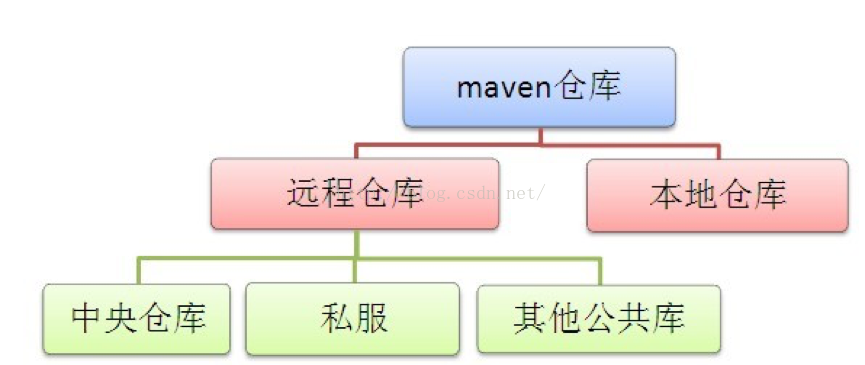
# Maven仓库 – 用于jar包管理

## 定义：

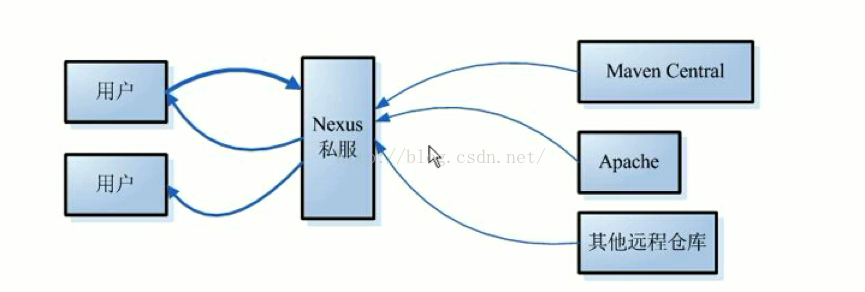
**本地仓库**：在本地电脑集中存储工程所需jar包（还有插件等）的地方被称之为仓库（Repository）。

**远程仓库**：中央仓库是默认的远程仓库，maven在安装的时候，自带的默认中央仓库地址为http://repo1.maven.org/maven2/，此仓库由Maven社区管理，包含了绝大多数流行的开源Java构件，以及源码、作者信息、SCM、信息、许可证信息等。一般来说，简单的Java项目依赖的构件都可以在这里下载到。Maven社区提供了一个中央仓库的搜索地。还有中央仓库的镜像仓库，就是中央仓库的副本。

实际开发中，一般不会使用maven默认的中央仓库，现在业界使用最广泛的仓库地址为： http://mvnrepository.com/，比默认的中央仓库更快、更全、更稳定，谁用谁知道。



1. **私服仓库：**一般来讲，公司都会通过自己的私有服务器在局域网内架设一个仓库代理。私服可以看作一种特殊的远程仓库，代理广域网上的远程仓库，供局域网内的Maven用户使用。当Maven需要下载构件的时候，先从私服请求，如果私服上不存在该构件，则从外部的远程仓库下载，缓存在私服上之后，再为Maven的下载请求提供服务。



## Maven私服的优点

* 1. 可以把公司的私有jar包，以及无法从外部仓库下载到的构件上传到私服上，供公司内部使用；
  2. 节省自己的外网带宽：减少重复请求造成的外网带宽消耗；
  3. 加速Maven构建：如果项目配置了很多外部远程仓库的时候，构建速度就会大大降低；
  4. 提高稳定性，增强控制：Internet不稳定的时候，maven构建也会变的不稳定，一些私服软件还提供了其他的功能

## 仓库的配置 – 作用是指明本地仓库和远程仓库的位置

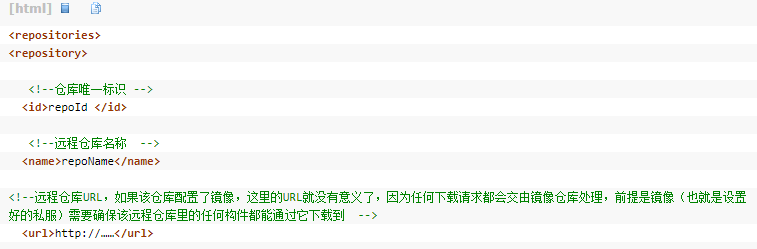
1. 设置镜像：

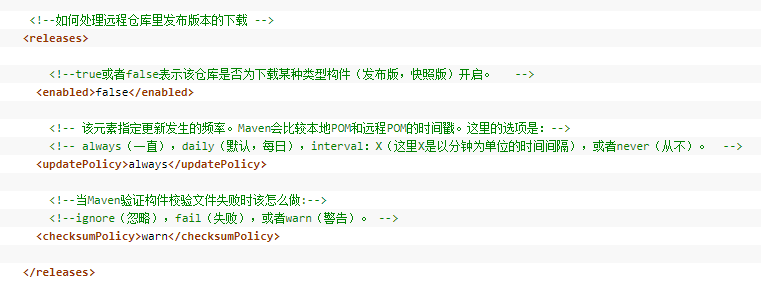


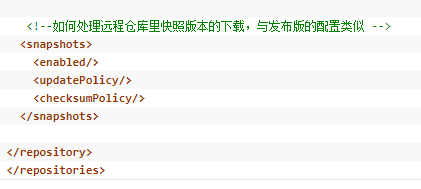
节点<mirrors>下面可以配置多个镜像，<mirrorOf>用于指明是哪个仓库的镜像，上例中使用通配符“\*”表明该私服是所有仓库的镜像，不管本地使用了多少种远程仓库，需要下载构件时都会从私服请求。如果只想将私服设置成某一个远程仓库的镜像，使用<mirrorOf>指定该远程仓库的ID即可。

1. 设置远程仓库 – 在profile节点下

* 管理jar包的远程仓库配置







* 管理插件的远程仓库配置

可以配置多个远程仓库，用<id>加以区分。除此之外，还有一个与<repositories>并列存在<pluginRepositories>节点，用来配置插件的远程仓库。

仓库主要存储两种构件。

第一种构件被用作其它构件的依赖，最常见的就是各类jar包。这是中央仓库中存储的大部分构件类型。

另外一种构件类型是插件，Maven插件是一种特殊类型的构件。由于这个原因，插件仓库独立于其它仓库。<pluginRepositories>节点与<repositories>节点除了根节点的名字不一样，子元素的结构与配置方法完全一样：



* 远程仓库有releases和snapshots两组配置，POM就可以在每个单独的仓库中，为每种类型的构件采取不同的策略。例如，有时候会只为开发目的开启对快照版本下载的支持，就需要把<releases>中的<enabled >设为“false”，而<snapshots>中的<enabled >设为“true”。
* 由于远程仓库的配置是挂在<profile>节点下面，如果配置有多个<profile>节点，那么就可能有多种远程仓库的设置方案，该方案是否生效是由它的父节点<profile>是否被激活决定的。

1. 设置发布权限 – 用于开发者之间的资源共享，互相调用接口jar包

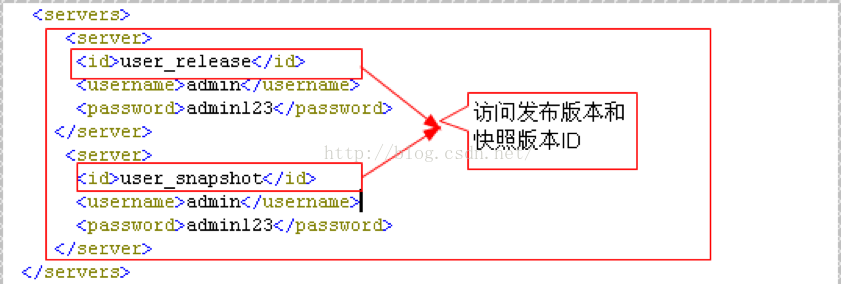
* 解决的方式是这样，每个模块完成了某个阶段性的功能，都会将提供对外服务的接口打成jar包，传到公司的私服当中，谁要使用该模块的功能，只需要在pom.xml文件中声明一下，maven就会像下载其他jar包那样把它引入你的工程
* 在POM中声明的版本一般是快照版：



各个模块会不断的上传新的jar包，如果本地项目依赖的是快照版，那么maven一旦发现该jar包有新的发布，就会将它下载下来替代以前的旧版本。比如，支付模块在测试的时候发现有个bug，修复了一下，然后将快照版发布到私服。而你只需要专注于下单模块的开发，所依赖的支付模块的更新由maven处理，不需要关心。一旦你开发的模块修复了一个bug，或者添加了一个新功能等修改，只需要将发布一次快照版本到私服即可，谁需要依赖你的接口谁自然会去私服下载，你也不用关心。

1. 设置私服权限

* 配置私服的信息是在pom文件中，但是认证信息则是在setting.xml中，这是因为pom文件往往是被提交到代码仓库中供所有成员访问的，而setting.xml是存放在本地的，这样是安全的。



# setting.xml配置文件 -- 详解见settings文件

## 存放位置

* + 安装的地方：${M2\_HOME}/conf/settings.xml
  + 用户的目录：${user.home}/.m2/settings.xml

前者又被叫做全局配置，对操作系统的所有使用者生效；后者被称为用户配置，只对当前操作系统的使用者生效。如果两者都存在，它们的内容将被合并，并且用户范围的settings.xml会覆盖全局的settings.xml。

## 注意事项

1. Maven安装后，用户目录下不会自动生成settings.xml，只有全局配置文件。如果需要创建用户范围的settings.xml，可以将安装路径下的settings复制到目录${user.home}/.m2/。Maven默认的settings.xml是一个包含了注释和例子的模板，可以快速的修改它来达到你的要求。
2. 全局配置一旦更改，所有的用户都会受到影响，而且如果maven进行升级，所有的配置都会被清除，所以要提前复制和备份${M2\_HOME}/conf/settings.xml文件，一般情况下不推荐配置全局的settings.xml。

# pom.xml文件详解

## 定义 -- 项目级别的配置文件

setting.xml主要用于配置maven的运行环境等一系列通用的属性，是全局级别的配置文件；而pom.xml主要描述了项目的maven坐标，依赖关系，开发者需要遵循的规则，缺陷管理系统，组织和licenses，以及其他所有的项目相关因素，是项目级别的配置文件。

（POM基本配置，见pom详解文件）

## classifier属性：

有时候引用某个jar包，classifier不写的话会报错。

classifier元素用来帮助定义构件输出的一些附属构件。附属构件与主构件对应，比如主

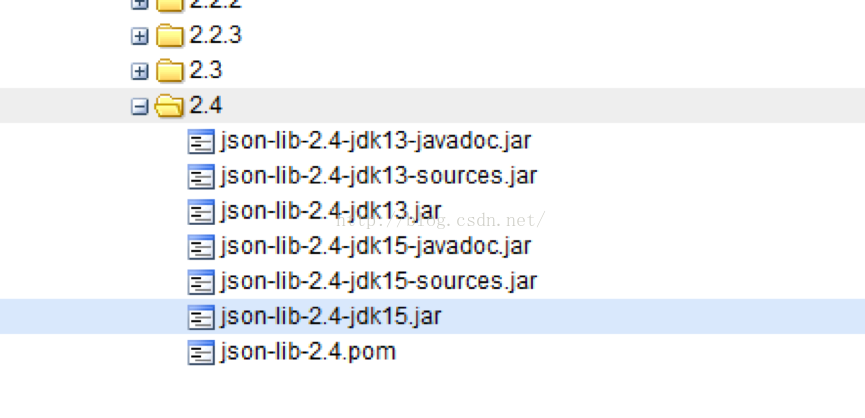
构件是 kimi-app-2.0.0.jar，该项目可能还会通过使用一些插件生成如：

kimi-app-2.0.0-javadoc.jar （Java文档）、 kimi-app-2.0.0-sources.jar（Java源代码）

这样两个附属构件。这时候，javadoc、sources就是这两个附属构件的classifier，这样附属构件也就拥有了自己唯一的坐标。

* classifier的用途在于:

1. maven download javadoc / sources jar包的时候，需要借助classifier指明要下载那个附属构件
2. 引入依赖的时候，有时候仅凭groupId、artifactId、version无法唯一的确定某个构件，需要借助classifier来进一步明确目标。比如JSON-lib，有时候会同一个版本会提供多个jar包，在JDK1.5环境下是一套，在JDK1.3环境下是一套：



引用它的时候就要注明JDK版本，否则maven不知道你到底需要哪一套jar包：



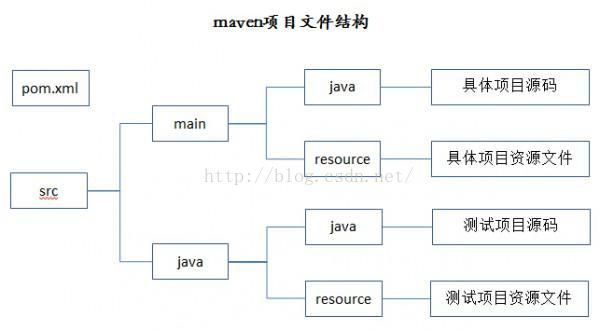
## 构建配置<build></build>

--- 具体见相关文件

# 约定优于配置

## 定义

maven有自己的一套默认配置，使用者除非必要，并不需要去修改那些约定内容。这就是所谓的“约定优于配置”。



* 注意事项：

每一个阶段的任务都知道怎么正确完成自己的工作，比如compile任务就知道从src/main/java下编译所有的java文件，并把它的输出class文件存放到target/classes中。

对maven来说，采用"约定优于配置"的策略可以减少修改配置的工作量，也可以降低学习成本，更重要的是，给项目引入了统一的规范。

## 版本规范

maven有自己的版本规范，一般是如下定义：

<majorversion>.<minor version>.<incremental version>-<qualifier>，比如1.2.3-beta-01。

要说明的是，maven自己判断版本的算法是major,minor,incremental部分用数字比较，qualifier部分用字符串比较，所以要小心 alpha-2和alpha-15的比较关系，最好用 alpha-02的格式。

* 注意事项：

maven在版本管理时候可以使用几个特殊的字符串 SNAPSHOT ,LATEST ,RELEASE 。比如"1.0-SNAPSHOT"。各个部分的含义和处理逻辑如下说明：

1. SNAPSHOT

如果一个版本包含字符串"SNAPSHOT"，Maven就会在安装或发布这个组件的时候将该符号展开为一个日期和时间值，转换为UTC时间。例如，"1.0-SNAPSHOT"会在2010年5月5日下午2点10分发布时候变成1.0-20100505-141000-1。这个词只能用于开发过程中，因为一般来说，项目组都会频繁发布一些版本，最后实际发布的时候，会在这些snapshot版本中寻找一个稳定的，用于正式发 布，比如1.4版本发布之前，就会有一系列的1.4-SNAPSHOT，而实际发布的1.4，也是从中拿出来的一个稳定版。

1. LATEST

指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个snapshot版，具体看哪个时间最后。

1. RELEASE

指最后一个发布版。

## Maven变量

除了在setting.xml以及pom.xml当中用properties定义的常量，maven还提供了一些隐式的变量，用来访问系统环境变量。

1. 内置属性

${basedir}表示项目根目录,即包含pom.xml文件的目录

${version}表示项目版本

${project.basedir}同${basedir}

${project.baseUri}表示项目文件地址

${maven.build.timestamp}表示项目构件开始时间

1. setting属性

${settings.localRepository }表示本地仓库路径

1. POM属性

${project.build.directory}表示主源码路径

${project.build.sourceEncoding}表示主源码的编码格式

${project.build.sourceDirectory}表示主源码路径

${project.build.finalName}表示输出文件名称

${project.version}表示项目版本,与${version}相同

1. Java系统属性

${user.home}表示用户目录

${java.version}表示Java版本

1. 环境变量属性

${env.JAVA\_HOME}表示JAVA\_HOME环境变量的值

${env.HOME }表示用户目录

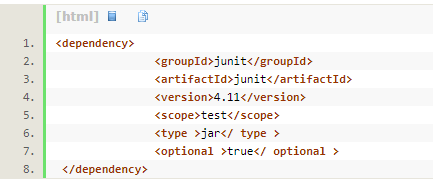
1. 上级工程变量

上级工程的pom中的变量用前缀 ${project.parent } 引用。上级工程的版本也可以这样引用: ${parent.version }

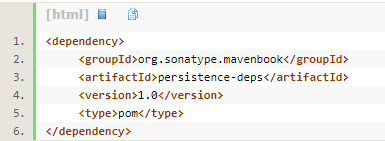
# 项目之间的依赖关系

* 项目的依赖关系主要分为三种：依赖，继承，聚合

## 依赖关系



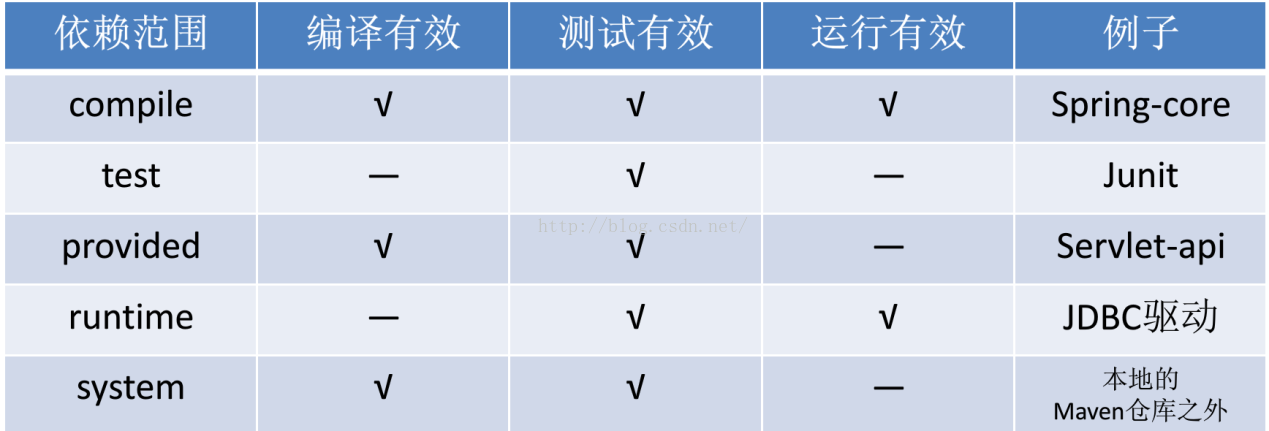
* 1. 任意一个外部依赖说明包含如下几个要素：groupId, artifactId, version, scope, type, optional。其中前3个是必须的。
  2. 这里的version可以用区间表达式来表示，比如(2.0,)表示>2.0，[2.0,3.0)表示2.0<=ver<3.0；多个条件之间用逗号分隔，比如[1,3],[5,7]。
  3. dependency中的type一般不用配置，默认是jar。当type为pom时，代表引用关系：



此时，本项目会将persistence-deps的所有jar包导入依赖库。可以创建一个打包方式为pom项目来将某些通用的依赖归在一起，供其他项目直接引用，不要忘了指定依赖类型为pom(<type>pom</type>)。

## 依赖关系中的 Scope（范围）选项

* 1. compile（编译范围）：compile是默认的范围；如果没有提供一个范围，那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。
  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，如果你开发了一个web应用，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。
  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 API JAR，而只有在运行的时候才需要JDBC驱动实现。
  4. test（测试范围）：test范围依赖在编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用。
  5. system（系统范围）：system范围依赖与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。 如果你将一个依赖范围设置成系统范围，你必须同时提供一个systemPath元素 。注意该范围是不推荐使用的（应该一直尽量去从公共或定制的Maven仓库中引用依赖）。



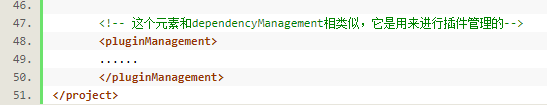
## 继承关系

继承就是**避免重复**，maven的继承也是这样，它还有一个好处就是让项目更加安全。项目之间存在**上下级关系**时就属于继承关系。

* 父目录配置







注意事项：

1. 注意，此时<packaging>必须为pom。
2. 为了项目的正确运行，必须让所有的子项目使用依赖项的统一版本，必须确保应用的各个项目的依赖项和版本一致，才能保证测试的和发布是相同的结果。
3. Maven 使用dependencyManagement 元素来提供了一种管理依赖版本号的方式。通常会在一个组织或者项目的最顶层的父POM 中看到dependencyManagement 元素。使用pom.xml 中的dependencyManagement 元素能让所有在子项目中引用一个依赖而不用显式的列出版本号。Maven 会沿着父子层次向上走，直到找到一个拥有dependencyManagement元素的项目，然后它就会使用在这个dependencyManagement 元素中指定的版本号。
4. 父项目在dependencies声明的依赖，子项目会从全部自动地继承。而父项目在dependencyManagement里只是声明依赖，并不实现引入，因此子项目需要显示的声明需要用的依赖。如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且version和scope都读取自父pom另外如果子项目中指定了版本号，那么会使用子项目中指定的jar版本。

* 子项目的配置





## 聚合关系

maven的多模块管理也是非常强大的。一般来说，maven要求同一个工程的所有模块都放置到同一个目录下，每一个子目录代表一个模块，比如

总项目/

pom.xml 总项目的pom配置文件

子模块1/

pom.xml 子模块1的pom文件

子模块2/

pom.xml子模块2的pom文件

* 总项目配置





* 子模块配置



## 继承和集合的关系

1. 首先，继承与聚合都属于父子关系，并且，聚合 POM与继承关系中的父POM的packaging都是pom。

不同的是，对于聚合模块来说，它知道有哪些被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。对于继承关系的父 POM来说，它不知道有哪些子模块继承与它，但那些子模块都必须知道自己的父 POM是什么。

1. 在实际项目中，一个 POM往往既是聚合POM，又是父 POM，它继承了某个项目，本身包含几个子模块，同时肯定会存在普通的依赖关系，就是说，依赖、继承、聚合这三种关系是并存的。
2. Maven可继承的POM 元素列表如下：

groupId ：项目组 ID ，项目坐标的核心元素；

version ：项目版本，项目坐标的核心元素；

description ：项目的描述信息；

organization ：项目的组织信息；

inceptionYear ：项目的创始年份；

url ：项目的 url 地址

develoers ：项目的开发者信息；

contributors ：项目的贡献者信息；

distributionManagerment：项目的部署信息；

issueManagement ：缺陷跟踪系统信息；

ciManagement ：项目的持续继承信息；

scm ：项目的版本控制信息；

mailingListserv ：项目的邮件列表信息；

properties ：自定义的 Maven 属性；

dependencies ：项目的依赖配置；

dependencyManagement：醒目的依赖管理配置；

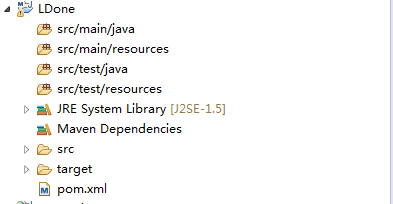
repositories ：项目的仓库配置；

build ：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等；

reporting ：包括项目的报告输出目录配置、报告插件配置等。

# 其他

## Eclipse中Maven项目结构



src/main/java 项目的源代码所在的目录

src/main/resources 项目的资源文件所在的目录

src/main/filters 项目的资源过滤文件所在的目录

src/main/webapp 如果是web项目，则该目录是web应用源代码所在的目录，比如html文件和web.xml等都在该目录下。

src/test/java 测试代码所在的目录

src/test/resources 测试相关的资源文件所在的目录

src/test/filters 测试相关的资源过滤文件所在的目录

/\*

不常用目录结构

src/it 集成测试代码所在的目录，主要是供别的插件使用的。

src/assembly 组件（Assembly）描述符所在的目录

src/site 站点文件

LICENSE.txt 项目的许可文件

NOTICE.txt 该项目依赖的库的注意事项

README.txt 项目的readme文件

\*/

src目录下，包含了项目所有的源代码和资源文件，以及其他项目相关的文件。在该目录的main目录下，包含了构建该项目的artifact（可以理解为时项目的程序部分）所需的代码和资源，而test目录包含了测试相关的代码和资源文件。

在目录main和test下的子目录结构，是非常相似的，都包含了两个子目录：java子目录和resources子目录，这两个目录分别放了源代码和资源文件。不同的是，在main目录下的这两个目录中放置了项目主程序的代码，而在test目录下，放置的是测试相关的代码和资源文件。其中java目录是代码的根目录，在该目录下的子目录，就是Java的包名对应的目录结构了。

target目录，这个目录只要用于存放项目构建的输出文件，比如class文件以及打包后的包文件等。

pom.xml 表示项目对象模型，关于项目的信息。

## pom.xml文件结构



* 关于Maven. 在Maven中，POM是它进行操作的基本单位。Maven的本质是以围绕POM这个概念为中心来管理整个项目的。POM中包含了项目中所有和项目相关的重要的信息，并且提供了类似一站式购物那样的依赖管理方式，可以很方便的找到当前项目关联的所有依赖来项目。

project 这个元素是Maven的pom.xml文件的顶层元素。每个POM文件都是以一个project元素包裹的。

modelVersion 这个元素指定了当前这个pom.xml文件使用的项目对象模型(POM)的版本。

groupId 这个元素表示创建这个项目的组织的唯一标识。这个元素的值是区分一个项目的关键信息之一，它的值通常是由该项目的组织的域名的反写产生的（和Java的包名的机制一样）。

artifactId 这个元素是唯一标识了该项目最终生成的artifact（可以理解为是我们编写代码产生的一个程序，所以可以理解为是手工作品）。一个项目的artifact一般是一个jar文件。一个项目最终生成的artifact的名字的格式是<artifactId>-<version>.<extension>，比如上面的项目生成的artifact为myapp-1.0.jar。

packaging 这个元素指定了该artifact打包的类型（比如：jar、war、ear等）。这不只是意味着该artifact是以什么方式打包，也指定了在构建过程中会用到的特定的生命周期（所谓的生命周期，简单的说就是构建的过程）。packaging元素的默认值是JAR。

version 这个元素指定了项目生成的artifact的版本号。在版本号中会包含当前项目的进度，比如这里的SNAPSHOT。

name 这个元素表示项目的名字，常用于生成的文档中。

url 这个元素表示该项目所在的站点，常用于生成的文档中。

description 这个元素提供了对项目的描述，通常用于生成的文档中