# Maven

## Maven简介

### 1. 构建构建&依赖管理工具&项目信息管理工具

作为Apache组织中的一个颇为成功的开源项目，Maven主要服务于基于Java平台的项目构建、依赖管理和信息管理。

Maven是一个异常强大的构建工具，能够帮助门额自动化构建过程，从清理、编译、测试到生成报告，再到打包和部署。

Maven是跨平台的，在Windows、Linux、Mac上都可以使用同样的命令。

Maven最大化地消除了构建的重复，抽象了构建声明周期。Maven不仅是构建工具，还是一个依赖管理工具和项目信息管理工具。

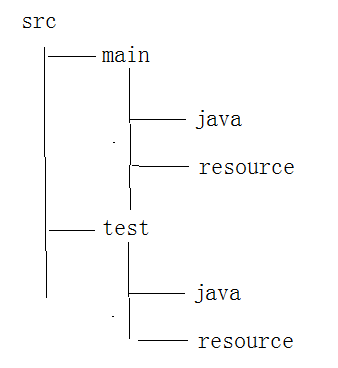
### pom.xml

POM(Project Object Model 项目对象模型)是maven的核心，定义了项目的基本信息，用于描述项目如何构建，声明项目依赖等。

* **modelVersion**：指定了当前POM模型的版本，对于Maven2和Maven3来说，它只能是4.0.0
* **groupId**:项目所属的组
* **artifactId**:当前Maven项目在组中唯一的ID
* **version**：项目当前的版本
* **scope**: 依赖范围，默认值compile，对主代码和测试代码都有效
* **name**：声明一个对用户更加友好的项目名称
* **packaging**: 默认打包类型为jar

### Maven约定--约定优于配置

* 在项目的根目录放置pom.xml
* 在src/main/java放置项目主代码
* 在src/test/java中放置项目测试代码



## 坐标和依赖

世界上任何一个构件都可以使用Maven坐标唯一标识，Maven坐标的元素包括groupId、artifactId、version、packaging、classifier。

### 坐标详解

* groupId：当前Maven项目隶属的实际项目，通常域名反向。
* artifactId：推荐使用实际项目名称作为artifactId
* version：Maven项目当前所处的版本
* packaging：Maven项目的打包方式，默认值jar.
* classifier：用来帮助定义构建输出的一些附属构件，与主构件对应。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，因此不能直接定义项目的classifier。

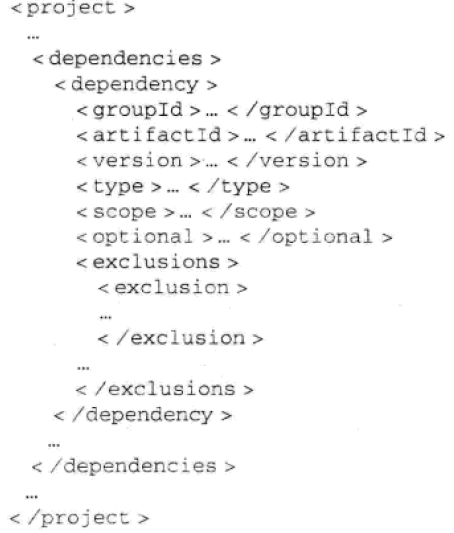
groupId、artifactId、version是必须定义 的，packaging是可选的(默认值jar)，而classifier是不能直接定义的.

项目构件的文件名一般的规则为：artifactId-version [-classifier].packaging

### 依赖配置

完整的依赖声明包含以下元素：

* groupId、artifactId、version：依赖的基本坐标
* type: 依赖的类型，对应于项目坐标定义的packaging,不必声明，默认值为jar
* scope：依赖的范围
* optional: 标记依赖是否可选
* exclusions: 用来排除传递性依赖。



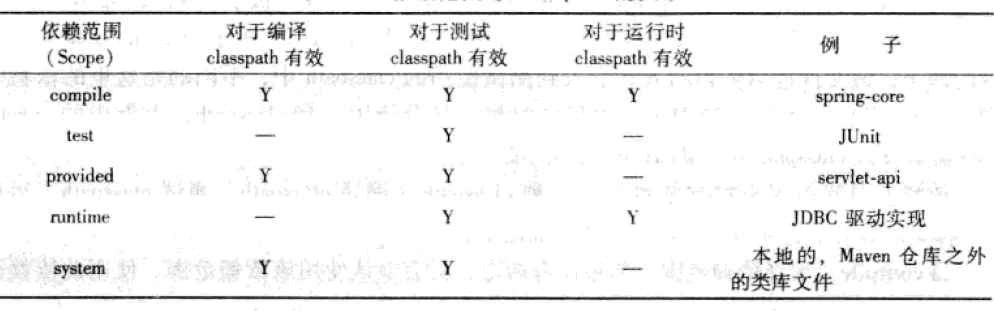
### 依赖范围

测试范围用元素scope表示。依赖范围是用来控制依赖于编译classpath、测试classpath、运行classpath的关系。

* compile: 编译依赖范围，默认的依赖范围，对编译、测试、运行三种classpath都有效。
* test: 测试依赖范围。只对测试classpath有效，在编译主代码或者运行项目的时候将无法使用此类依赖。
* provided: 已提供依赖范围，对于编译和测试有效，不会打包进发布包中。
* runtime: 对测试、运行classpath有效，但编译主代码时无效。
* system: 和provided一致。不从maven仓库获取该jar,而是通过systemPath指定该jar的路径

|  |
| --- |
| <dependency>  <groupId>javax.sql</groupId>  <artifactId>jdbc.stdext</artifactId>  <version>2.0</version>  <scope>system</scope>  <systemPath>${java.home}/lib/rt.jar</systemPath>  </dependency> |

* Import：导入依赖范围。



### 传递性依赖

Maven会解析各个直接依赖的POM，将那些必要的间接依赖，以传递性依赖的形式引入到当前的项目中。

①当第二直接依赖的范围是compile和runtime的时候，传递性依赖的范围与第一直接依赖的范围一致；

②当第二直接依赖的范围是test的时候，依赖不会得以传递；

③当第二直接依赖的范围是provided，且第一直接依赖范围也是provided的时候，传递性依赖的范围同样为provided。

### 依赖冲突&依赖调解

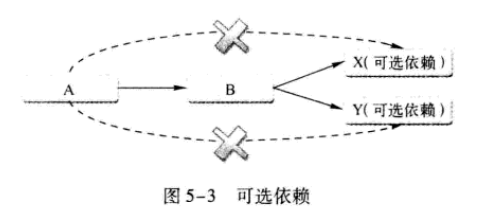
两条依赖路径上有某个依赖的两个版本，造成依赖冲突。

调解原则：

* 第一原则：路径最近者优先。
* 第二原则：第一声明者优先。顺序最靠前的那个依赖优胜。

### 可选依赖(optional)

可选依赖在传递性依赖中不被传递。Optional表示某个依赖为可选依赖。

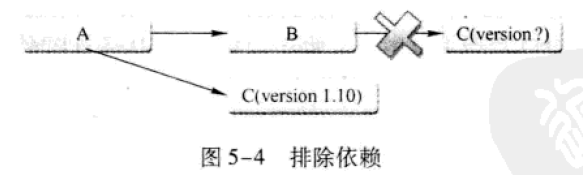


|  |
| --- |
| A->B、B->X(可选)、B->Y(可选)，则X、Y将不会对A有任何影响 |

### 最佳实践

##### 排除依赖

使用exclusions元素声明排除依赖，声明exclusion的时候只需要groupId和artifactId，而不需要version元素(Maven解析后的依赖中，不可能出现groupId和artifactId相同两个依赖)



##### 归类依赖

|  |
| --- |
| <properties>  <xxx.version>2.5.6</xxx.version>  </properties> |

首先使用properties定义Maven属性，Maven运行的时候会将所有${xx.version}替换为properties定义的属性值。

##### 优化依赖

查看已解析依赖：

1. mvn dependency:tree 树形形式
2. mvn dependency:list 列表形式

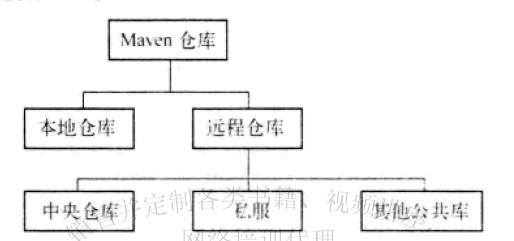
## 仓库

### 定义

仓库分为两类：本地仓库和远程仓库。

中央仓库是Maven核心自带的远程仓库，在默认配置下，当本地仓库没有Maven需要的构件的时候，就会尝试从中央仓库下载。

私服是另一种特殊的远程仓库，为了节省带宽和时间，应该在局域网内架设一个私有的仓库服务器，用其代理所有外部的远程仓库。



### 本地仓库

修改本地仓库地址：

|  |
| --- |
| <settings>  <localRepository>本地仓库地址</localRepository>  </settings> |

安装完Maven后，如果不执行Maven命令，本地仓库目录是不存在的，但用户输入第一条Maven命令后，Maven才会创建本地仓库。

### 远程仓库

#### 3.1 中央仓库

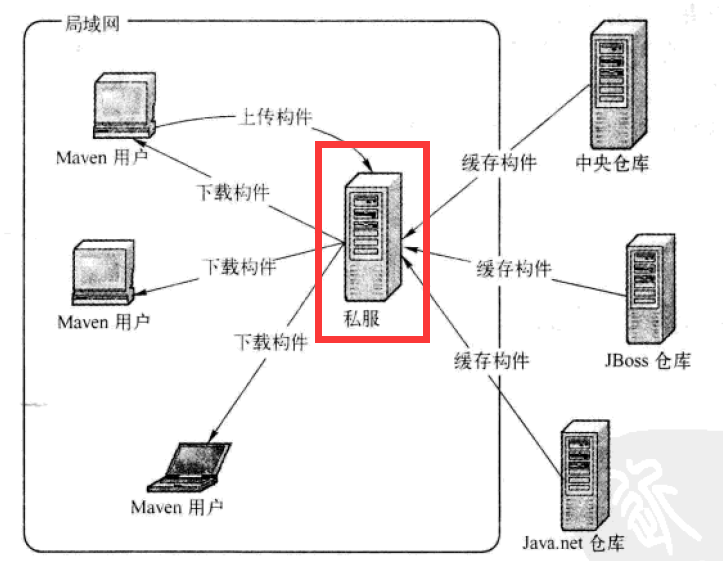
中央仓库是默认的远程仓库，Maven的安装文件中的超级POM自带了中央仓库的配置。

#### 3.2 私服

私服是一种特殊的远程仓库，它是架设在局域网内的仓库服务，私服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，它从私服请求，如果私服上不存在构件，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。

一些无法从远程仓库中下载到的构件也能从本地上传到私服上供Maven用户使用。



#### 3.3 远程仓库配置



(Maven只会从JBoss仓库下载发布版的构件，而不会下载快照版的构件)

任何一个仓库声明的id必须是唯一的。

**Layout仓库布局**

* default表示仓库的布局是Maven2及Maven3的默认布局
* Legacy表示仓库的布局是Maven1的布局

**updatePolicy配置从远程仓库检查更新的频率**

* daily：默认值，表示Maven每天检查一次。
* never: 从不检查更新。
* always: 每次构建都检查更新
* interval: X 每隔X分钟检查一次更新

通过maven-metadata-local.xml的时间戳与远程仓库进行比较，判断是否需要更新。

**checksumPolicy配置Maven检查校验和文件的策略**

Maven检查校验和文件失败时：

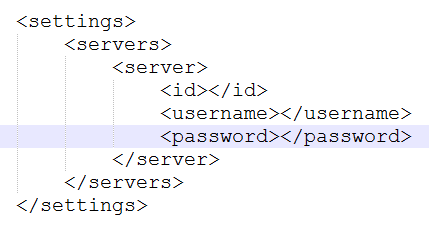
* warn：在构建时输出警告信息
* fail: 构建失败
* ignore: 使Maven完全忽略校验和错误

|  |
| --- |
| 安装nexus私服，修改sha1值，设置checksumPolicy，验证checksumPolicy工作原理。 |

#### 3.4 远程仓库认证

配置远程仓库信息是在POM文件中，而配置远程仓库认证信息必须配置在settings.xml文件中。

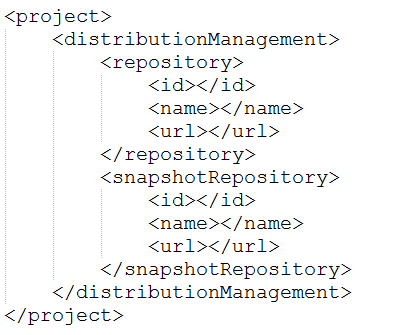
在settings.xml中配置仓库认证信息：



settings.xml中server元素的id必须与POM中需要认证的repository元素的id完全一致。

#### 3.5 部署远程仓库

在pom.xml文件中配置distributionManagement元素：



distributionManagement包含repository和snapshotRepository子元素，分别表示发布版本构建和快照版本构件的部署仓库。

#### 3.6 强制更新

从命令行加入参数-U，强制检查更新，Maven就会忽略<updatePolicy>的配置。

## 镜像

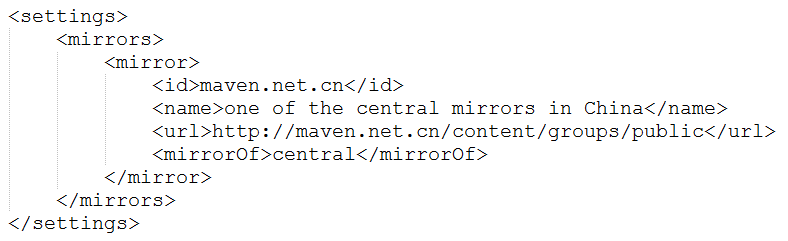
### 定义

如果仓库X可以提供仓库Y存储的所有内容，那么就认为X是Y的一个镜像。例如：

<http://maven.net.cn/content/groups/public>是中央仓库[http://repo1.maven.org/maven2](http://maven.net.cn/content/groups/public)在中国的镜像。

### 配置镜像

在settings.xml文件中配置远程仓库的镜像，mirrorOf的值表示该配置为中央仓库的镜像。任何对于中央仓库的请求都会转至该镜像。



## 生命周期和插件

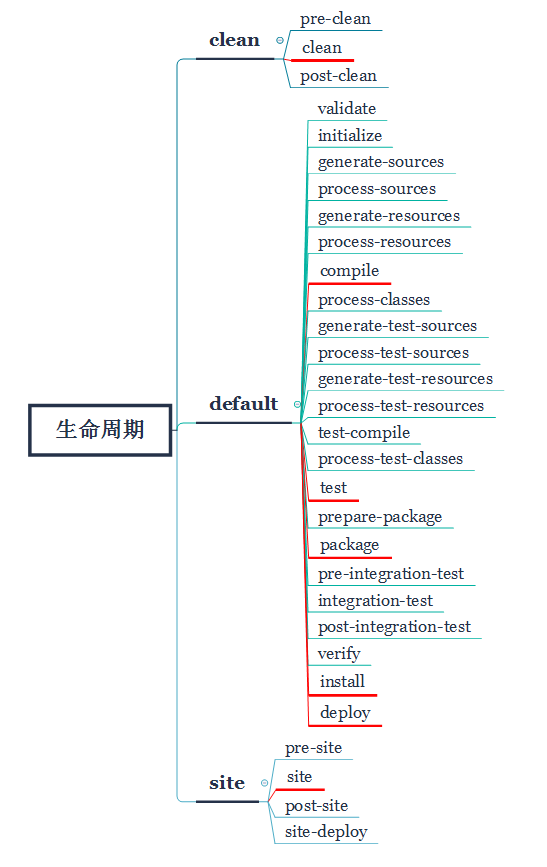
### 生命周期

Maven的生命中期就是为了对所有的构建过程进行抽象和统一。Maven的生命周期是抽象的，本身不做任何实际的工作，实际的任务都交由插件来完成。

#### 三套生命周期&阶段

Maven拥有三套相互独立的生命周期：clean、default和site.

每个生命周期包含一些阶段(phase)。生命周期的阶段是有顺序的，并且后面的阶段依赖于前面的阶段。而三套生命周期是相互独立的。

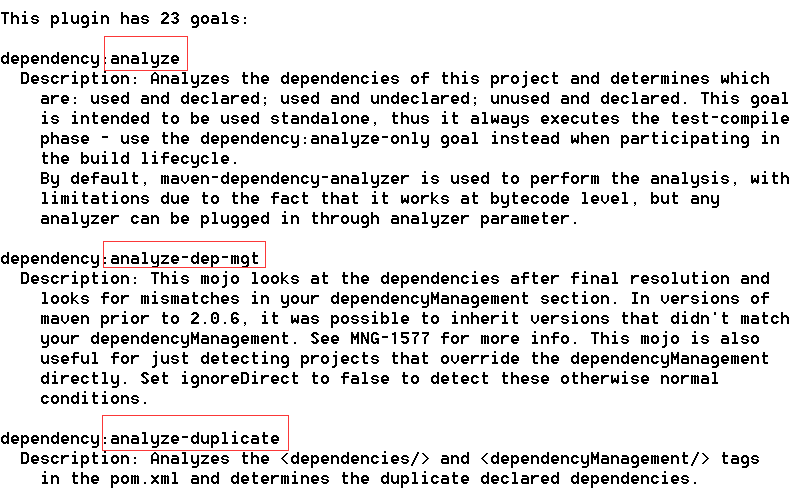


各个生命周期是相互独立的，而一个生命周期的阶段是有前后依赖关系的。

命令行常用的调用Maven的方式是【mvn 生命周期阶段】

### 插件目标

很多功能聚集在一个插件里，每个功能就是一个插件目标。

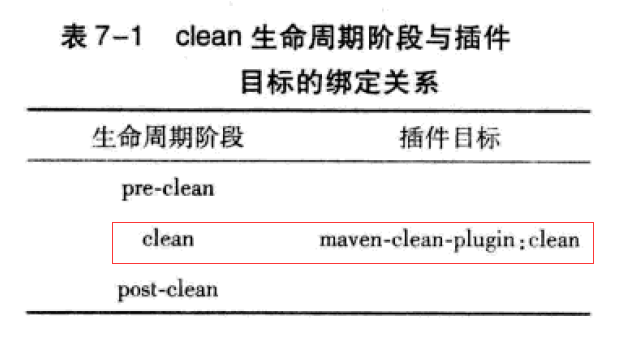


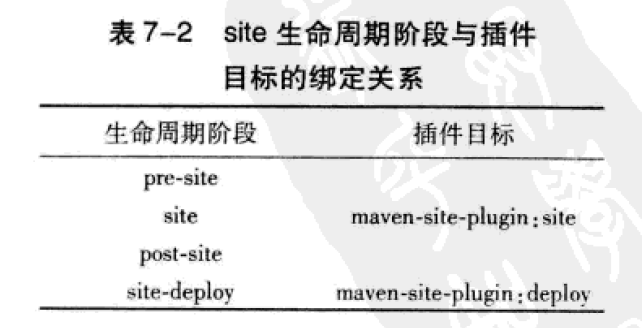
### 插件绑定

生命周期的阶段与插件的目标相互绑定，以完成某个具体的构建任务。

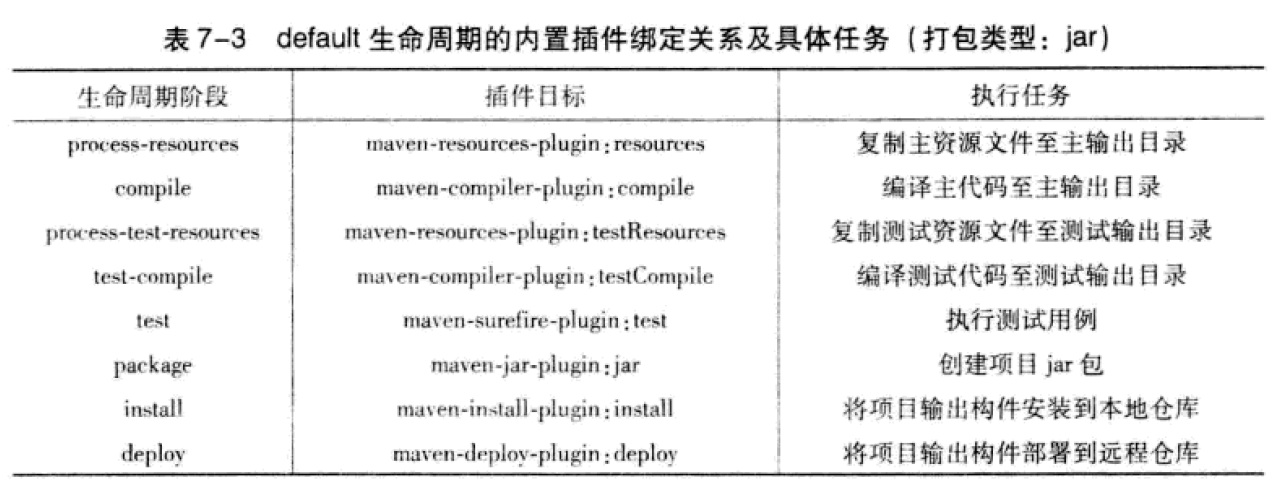
#### 3.1 内置绑定

为了能让用户几乎不用任何配置就能构建Maven项目，Maven为一些主要的生命周期阶段绑定了很多插件的目标。



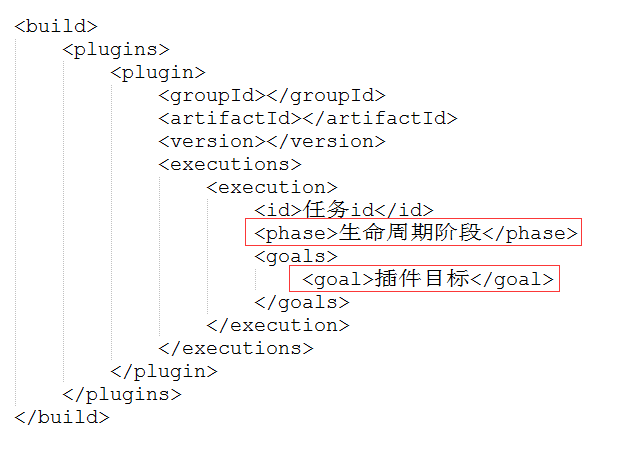


由于项目的打包类型会影响构建的具体过程，因此default生命周期的阶段与插件目标的绑定关系由项目打包类型所决定。



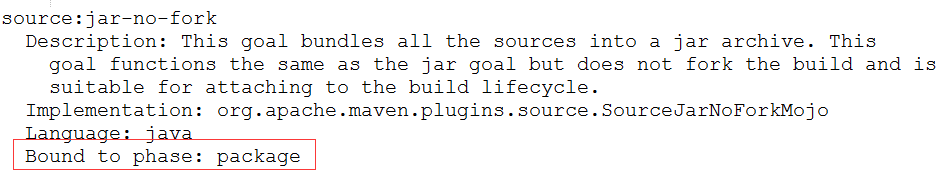
#### 3.2 自定义绑定

除了内置绑定外，用户还能够自己选择将某个插件目标绑定到生命周期的某个阶段以执行更多更富特色的任务。



很多插件的目标在编写时已经定义了默认绑定阶段。可以通过下面的命令进行查看：

mvn help:describe -Dplugin=source -Ddetail



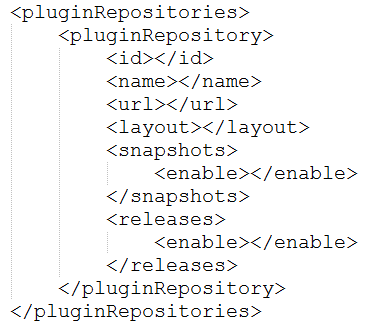
当插件目标被绑定到不同的生命周期阶段的时候，其执行顺序会由生命周期阶段的先后顺序决定；当多个插件目标绑定到同一个阶段的时候，这些插件声明的先后顺序决定了目标的执行顺序。

### 插件配置

* 命令行插件配置：在Maven命令中使用-D参数，伴随参数键=参数值
* POM中插件全局配置：build -> plugins ->plugin -> configuration
* POM中插件任务配置：build ->plugins ->plugin ->executions->execution->configuration

### 插件仓库

Maven会区别对待依赖的远程仓库和插件的远程仓库。插件的远程仓库配置：



### 命令行调用方式

* mvn 生命周期阶段 如 mvn clean install
* mvn 插件：插件目标 如 mvn dependency:tree(有些任务不适合绑定在生命周期)

## 聚合和继承

### 聚合&继承

聚合特性能够把项目的各个模块聚合在一起构建；继承特性能帮助抽取各模块相同的依赖和插件等配置，能够简化POM和促进各个模块配置的一致性。

#### 聚合

一个项目的子模块都应该使用相同的groupId,如果它们一起开发和发布，还应该使用相同的version，而且还应该使用一致的前缀。

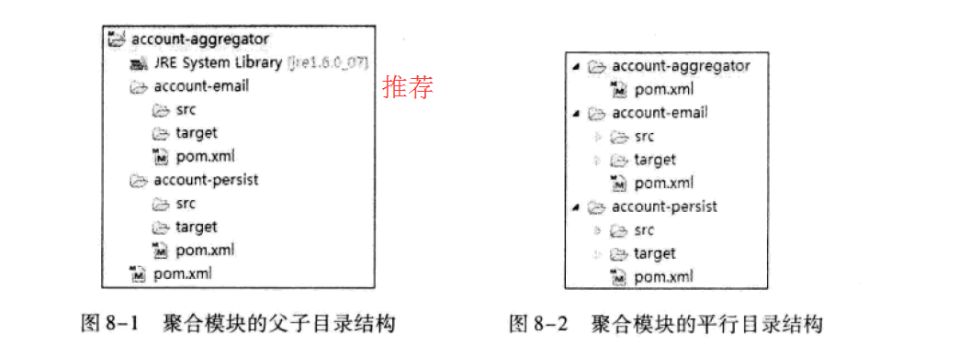
用户可以通过在一个打包方式为pom的Maven项目中声明任意数量的module元素来实现模块的聚合。每个module的值都是一个当前POM的相对目录。

|  |
| --- |
| groupId、artifactId与目录的关系  一、groupId与项目代码目录无关，当部署到本地仓库或远程仓库， groupId对应了仓库的目录结构。  二、模块所处的目录名称应当与其artifactId一致(非Maven要求) |

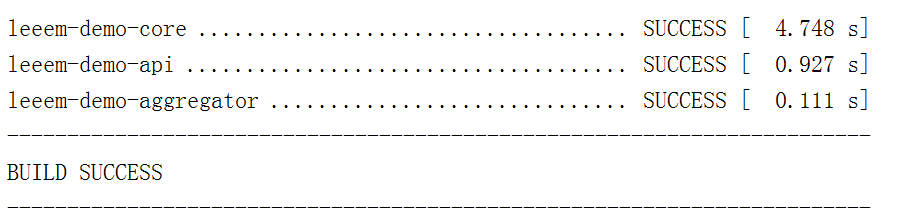
补：

聚合模块打包方式必须为pom，否则无法构建。

聚合模块与子模块的目录层级结构：通常将聚合模块放在项目目录的最顶层，其他模块作为聚合模块的子目录存在。



构建输出中显示的是各模块的名称，是<name>的内容，而不是<artifacted>。



#### 继承

父模块只是为了帮助消除重复配置，因此它本身不包含除POM外的项目文件。

作为父模块的POM，其打包类型也必须为pom.

Parent下的relativePath表示父模块POM的相对路径，默认值是../pom.xml,即Maven默认父POM在上一层目录。

当项目构建时，Maven会首先根据relativePath检查父POM，如果找不到，再从本地仓库查找。

##### dependencyManagement

在dependencyManagement元素下的依赖声明不会引入实际的依赖，能够约束dependencies下的依赖使用。

dependencyManagement声明的依赖既不会给父模块引入依赖，也不会给它的子模块引入依赖，不过dependencyManagement的配置是会被继承的。

完整的依赖声明已经包含在父POM中，子模块只需要配置简单的groupId和artifactId就能获得对应的依赖信息，从父POM中的dependencyManagement继承version和scope，从而引入正确的依赖。

如果子模块不声明依赖的使用，即使该依赖已经在父POM的dependencyManagement中声明了，也不会产生任何实际的效果。

使用这种依赖管理机制不能减少太多的POM配置，但是能够统一项目范围中依赖的版本.

##### pluginManagement

在pluginManagement元素中配置的插件依赖不会造成实际的插件调用行为。当在POM中配置了真正的plugin时，并且其groupId和artifacted与pluginManagement中配置的插件匹配时，pluginManagement的配置才会影响实际的插件行为。

#### 二者关系

聚合是为了方便快速构建项目，继承是为了消除重复配置。

* 聚合和继承模块的POM打包类型都为pom.
* 聚合和继承模块除了POM之外都没有实际内容。
* 对于聚合模块，它知道有哪些模块被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。
* 对于继承关系的父模块来说，它不知道有哪些子模块继承于它，但那些子模块都必须知道自己的父POM是什么

### 反应堆

按module从上到下的读取次序还不足以决定反应堆的构建顺序，Maven还需要考虑模块之间的继承和依赖关系。

模块间的依赖关系会将反应堆构成一个有向非循环图，不允许出现循环，否则Maven报错。

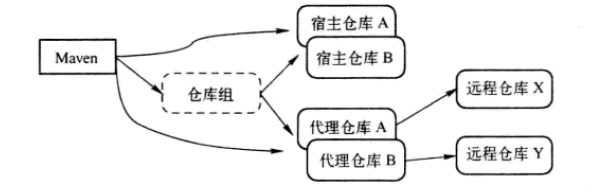
裁剪反应堆

* -am --also-make同时构建所列模块的依赖模块
* -amd --also-make-dependents同时构建依赖于所列模块的模块
* -pl --projects <args> 构建指定的模块
* -rf --resume-from <args> 可以在完整的反应堆构建顺序基础上指定从哪个模块开始构建。

## Nexus仓库

### 仓库

Nexus内置的仓库包括仓库组(group)、宿主仓库(hosted)、代理仓库(proxy)、虚拟仓库(virtual)。仓库的Policy属性表示该仓库为发布版本仓库(Release)还是快照版本仓库(Snapshot).



无仓库组时，Maven可以直接从宿主仓库下载构件，Maven也可以从代理仓库下载构件，而代理仓库会间接地从远程仓库下载并缓存构件。

有仓库组时，Maven可以从仓库组下载构件，而仓库组没有实际的内容，它会转向其包含的宿主仓库或代理仓库获得实际构件的内容。

## 测试

### 自动测试

在默认情况下，maven-surefire-plugin的test目标会自动执行测试源码路径(默认为src/test/java)下所有符合一组命名模式的测试类：

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

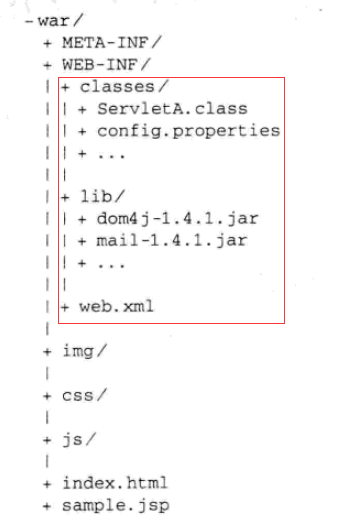
### 动态指定要运行的测试用例

Mvn test -Dtest = Rondom\*Test,Rondom1Test

## Maven构建Web项目

### War

War包结构：

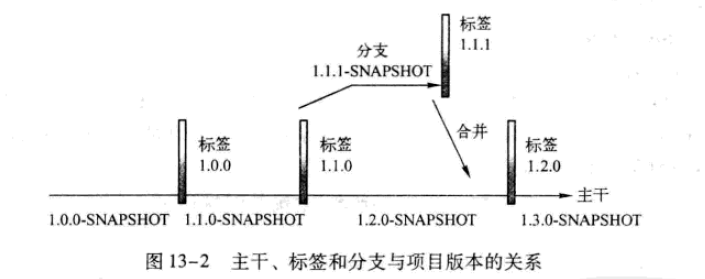


Web项目与一般Jar项目不同的地方在于，它还有一个Web资源目录，其默认位置是src/main/webapp/，该目录下必须包含一个子目录WEB-INF，WEB-INF下面必须包含web.xml

## Maven版本号

Maven的版本号定义约定：

<主版本>.<次版本>.<增量版本>-<里程碑>



## 属性 & Profile & 资源过滤

Maven为了支持构建的灵活性，内置了三大特性：属性、Profile和资源过滤。

### 属性

通过<properties>，用户可以自定义一个或多个属性，然后在POM的其他地方使用${属性名称}的方式引用。

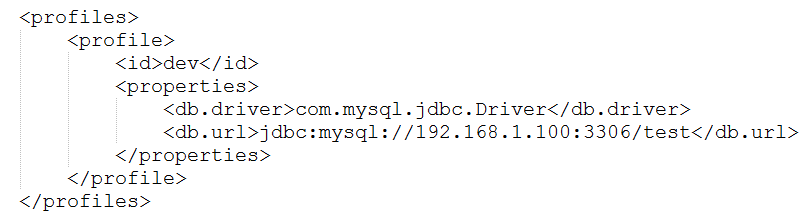
Maven属性包括:

* 内置属性 ${basedir} 、${version}
* POM属性 用户可以使用POM属性引用POM文件中对应元素的值，如${project.artifacted}
* 自定义属性 <properties>
* Settings属性 同POM属性
* Java系统属性 ${user.home}
* 环境变量属性 使用以.env开头的maven属性 ${env.JAVA\_HOME}

Maven的内置属性包括：

* + - ${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;
    - ${maven.build.timestamp.format}表示属性${maven.build.timestamp}的展示格式,默认值为yyyyMMdd-HHmm

### 资源过滤



将数据库配置的变化部分提取成了Maven属性，在POM的profile中定义了这些属性的值，由于Maven属性默认只有在POM中才会解析。还需要为资源目录开启资源过滤。最后，只需要在命令行激活profile，Maven就能够在构建项目的时候使用profile中属性值替换数据库配置文件中的属性引用：

Mvn clean install -Pdev

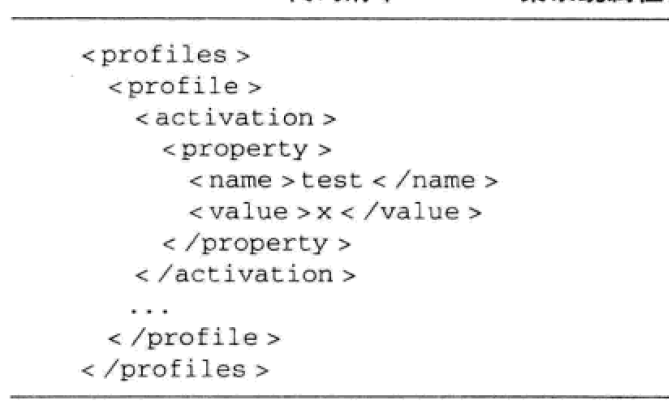
Maven的-P参数表示在命令行激活一个profile

### 激活profile

-P激活(命令行激活)：mvn clean install -Pdev-x,dev-y

Settings文件显示激活：如果用户希望某个profile默认一直处于激活状态，就可以配置settings.xml的activeProfiles元素。表示其配置的profile对于所有项目都处于激活状态。

系统属性激活：用户可以配置当某系统属性存在的时候，自动激活profile



mvn clean install -Dtest=x

用户可以在定义profile的时候指定其默认激活。

### Profile的种类

* Pom.xml
* 用户settings.xml
* 全局settings.xml
* Profiles.xml

与一般的资源文件一样，web资源文件默认不会被过滤。开启一般资源文件的过滤也不会影响到web资源文件。