Maven

一、 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。

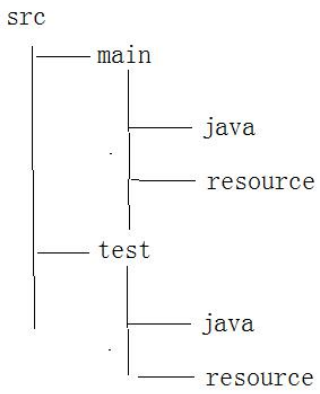
2. pom.xml

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

3. 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



二、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。

1. 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

2. 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。

```
<project>
  …
  <dependencies>
    <dependency>
      <groupId>…</groupId>
      <artifactId>…</artifactId>
      <version>…</version>
      <type>…</type>
      <scope>…</scope>
      <optional>…</optional>
      <exclusions>
        <exclusion>
        …
        </exclusion>
      …
      </exclusions>
    </dependency>
    …
  </dependencies>
  …
</project>
```

3. 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。

依赖范围（Scope）	对于编译 classpath 有效	对于测试 classpath 有效	对于运行时 classpath 有效	例　子
compile	Y	Y	Y	spring-core
test	—	Y	—	JUnit
provided	Y	Y	—	servlet-api
runtime	—	Y	Y	JDBC 驱动实现
system	Y	Y	—	本地的，Maven 仓库之外的类库文件

4. 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。

5. 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。

6. 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。

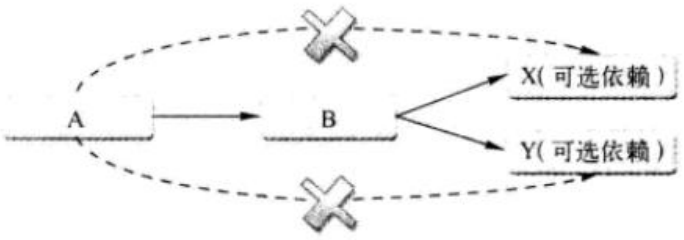


图 5-3　可选依赖

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

7. 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)

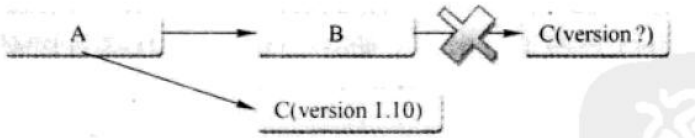


图 5-4 排除依赖

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

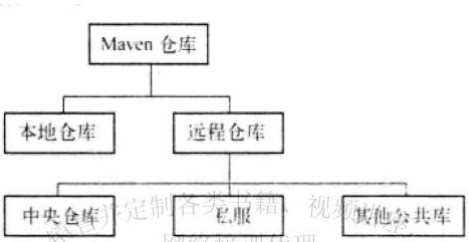
三、仓库

1. 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。



2. 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。

3. 远程仓库

3.1 中央仓库

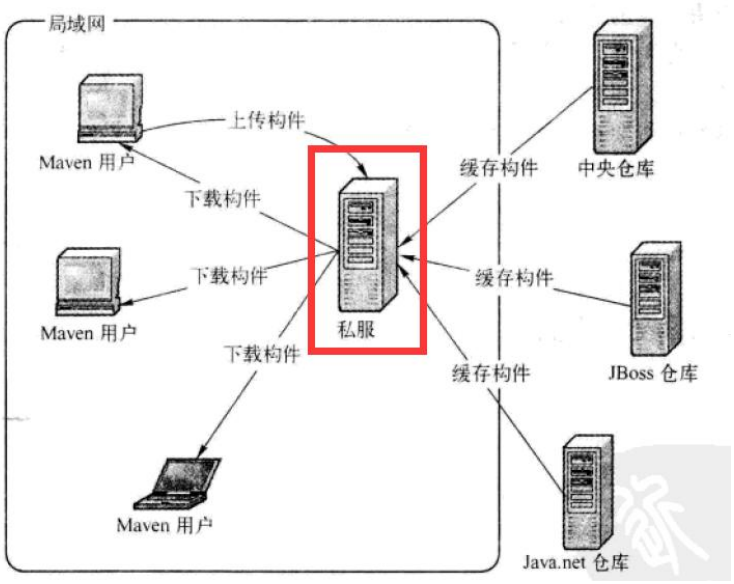
中央仓库是默认的远程仓库，Maven 的安装文件中的超级 POM 自带了中央仓库的配置。

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

```
<project>
    <repositories>
        <repository>
            <id>jboss</id>
            <name>JBoss Repository</name>
            <url>http://repository.jboss.com/maven2</url>
            <releases>
                <enabled>true</enabled>
                <updatePolicy>daily</updatePolicy>
                <checksumPolicy>ignore</checksumPolicy>
            </releases>
            <snapshots>
                <enabled>false</enabled>
                <updatePolicy>daily</updatePolicy>
                <checksumPolicy>ignore</checksumPolicy>
            </snapshots>
            <layout>default</layout>
        </repository>
    </repositories>
</project>
```

(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>
    <servers>
        <server>
            <id></id>
            <username></username>
            <password></password>
        </server>
    </servers>
</settings>
```

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：

```
<project>
    <distributionManagement>
        <repository>
            <id></id>
            <name></name>
            <url></url>
        </repository>
        <snapshotRepository>
            <id></id>
            <name></name>
            <url></url>
        </snapshotRepository>
    </distributionManagement>
</project>
```

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。

四、镜像

1. 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 在中国的镜像。

2. 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。

```
<settings>
    <mirrors>
        <mirror>
            <id>maven.net.cn</id>
            <name>one of the central mirrors in China</name>
            <url>http://maven.net.cn/content/groups/public</url>
            <mirrorOf>central</mirrorOf>
        </mirror>
    </mirrors>
</settings>
```

五、生命周期和插件

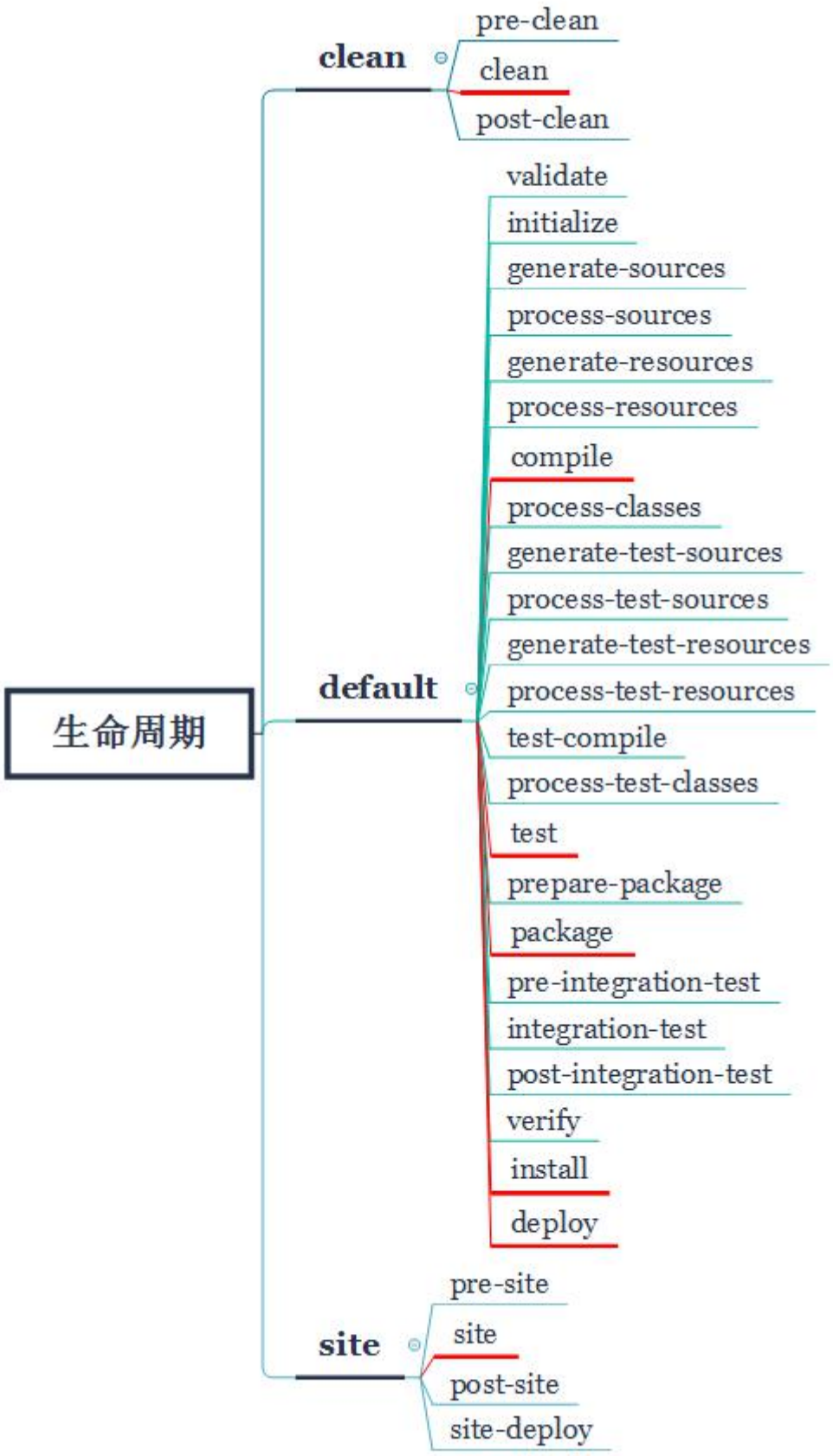
1. 生命周期

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。

1.1 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】

2. 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。

```
This plugin has 23 goals:

dependency:analyze
  Description: Analyzes the dependencies of this project and determines which
    are: used and declared; used and undeclared; unused and declared. This goal
    is intended to be used standalone, thus it always executes the test-compile
    phase - use the dependency:analyze-only goal instead when participating in
    the build lifecycle.
    By default, maven-dependency-analyzer is used to perform the analysis, with
    limitations due to the fact that it works at bytecode level, but any
    analyzer can be plugged in through analyzer parameter.

dependency:analyze-dep-mgt
  Description: This mojo looks at the dependencies after final resolution and
    looks for mismatches in your dependencyManagement section. In versions of
    maven prior to 2.0.6, it was possible to inherit versions that didn't match
    your dependencyManagement. See MNG-1577 for more info. This mojo is also
    useful for just detecting projects that override the dependencyManagement
    directly. Set ignoreDirect to false to detect these otherwise normal
    conditions.

dependency:analyze-duplicate
  Description: Analyzes the <dependencies/> and <dependencyManagement/> tags
    in the pom.xml and determines the duplicate declared dependencies.
```

3. 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。

表 7-1 clean 生命周期阶段与插件目标的绑定关系

生命周期阶段	插件目标
pre-clean	
clean	maven-clean-plugin:clean
post-clean	

表 7-2　site 生命周期阶段与插件目标的绑定关系

生命周期阶段	插件目标
pre-site	
site	maven-site-plugin:site
post-site	
site-deploy	maven-site-plugin:deploy

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。

表 7-3　default 生命周期的内置插件绑定关系及具体任务（打包类型：jar）

生命周期阶段	插件目标	执行任务
process-resources	maven-resources-plugin:resources	复制主资源文件至主输出目录
compile	maven-compiler-plugin:compile	编译主代码至主输出目录
process-test-resources	maven-resources-plugin:testResources	复制测试资源文件至测试输出目录
test-compile	maven-compiler-plugin:testCompile	编译测试代码至测试输出目录
test	maven-surefire-plugin:test	执行测试用例
package	maven-jar-plugin:jar	创建项目 jar 包
install	maven-install-plugin:install	将项目输出构件安装到本地仓库
deploy	maven-deploy-plugin:deploy	将项目输出构件部署到远程仓库

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。

```
<build>
    <plugins>
        <plugin>
            <groupId></groupId>
            <artifactId></artifactId>
            <version></version>
            <executions>
                <execution>
                    <id>任务id</id>
                    <phase>生命周期阶段</phase>
                    <goals>
                        <goal>插件目标</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

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

```
source:jar-no-fork
  Description: This goal bundles all the sources into a jar archive. This
    goal functions the same as the jar goal but does not fork the build and is
    suitable for attaching to the build lifecycle.
  Implementation: org.apache.maven.plugins.source.SourceJarNoForkMojo
  Language: java
  Bound to phase: package
```

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。

4. 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

5. 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：

```
<pluginRepositories>
    <pluginRepository>
        <id></id>
        <name></name>
        <url></url>
        <layout></layout>
        <snapshots>
            <enable></enable>
        </snapshots>
        <releases>
            <enable></enable>
        </releases>
    </pluginRepository>
</pluginRepositories>
```

6. 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)

六、聚合和继承

1. 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。

1.1 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。

补：

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)

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。

推荐

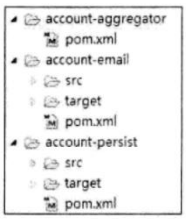


图 8-1 聚合模块的父子目录结构　　图 8-2 聚合模块的平行目录结构

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

```
leeem-demo-core ..................................... SUCCESS [  4.748 s]
leeem-demo-api ...................................... SUCCESS [  0.927 s]
leeem-demo-aggregator ............................... SUCCESS [  0.111 s]
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

1.2 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。

1.2.1 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.

1.2.2 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。

1.3 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

2. 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。

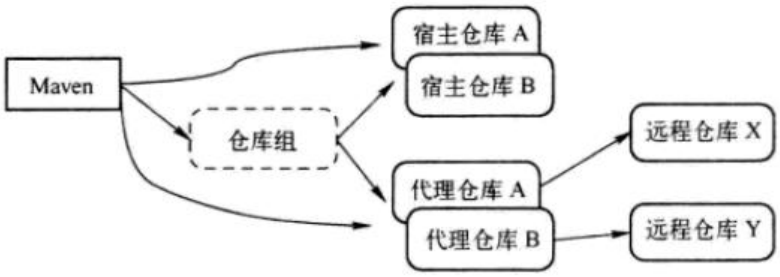
裁剪反应堆

- -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。

七、Nexus 仓库

1. 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。

八、测试

1. 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

2. 动态指定要运行的测试用例

Mvn test -Dtest = Rondom*Test,Rondom1Test

九、Maven 构建 Web 项目

1. 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

十、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>

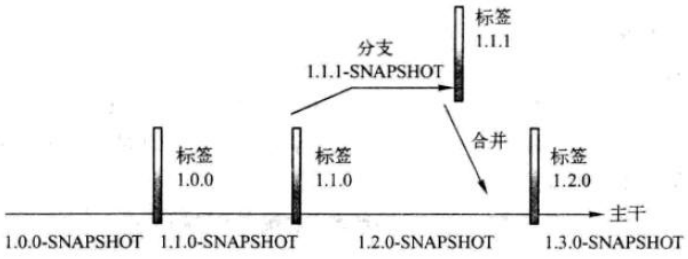


图 13-2　主干、标签和分支与项目版本的关系

十一、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。

1. 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

2. 资源过滤

```
<profiles>
    <profile>
        <id>dev</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
        </properties>
    </profile>
</profiles>
```

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

3. 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

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>test</name>
        <value>x</value>
      </property>
    </activation>
    ...
  </profile>
</profiles>
```

mvn clean install -Dtest=x

用户可以在定义 profile 的时候指定其默认激活。

4. 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。