Maven

坐标和依赖

- 坐标
  - groupId、artifactId、version是必须定义的，packaging是可选的（默认为jar），而classifier是不能直接定义的。
  - 项目构件的文件名是与坐标相对应的，一般的规则为artifactId-version [-classifier].packaging，[-classifier]表示可选。
  - groupId　定义当前Maven项目隶属的实际项目
  - 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
  - version　该元素定义Maven项目当前所处的版本
  - packaging　该元素定义Maven项目的打包方式
    - jar包
    - war包
    - pom包
    - maven-plugin
    - ear
  - classifier　该元素用来定义构建输出的一些附属构建，例如主构件是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
- 依赖
  - groupId、artifactId、version 坐标核心的三要素　如果没有version的话测默认取仓库中最新的Release版本，如果父模块中有定义则用父模块中的版本
  - type　依赖的类型，对应于项目坐标定义的packaging。大部分情况下，该元素不必声明，其默认值为jar。
  - scope　依赖的范围　用来控制依赖与这三种classpath（编译classpath、测试classpath、运行classpath）的关系
    - compile　编译依赖范围。如果没有指定，就会默认使用该依赖范围。使用此依赖范围的Maven依赖，对于编译、测试、运行三种classpath都有效。
    - test　测试依赖范围。使用此依赖范围的Maven依赖，只对于测试classpath有效，在编译主代码或者运行项目的使用时将无法使用此类依赖。
    - provided　已提供依赖范围。使用此依赖范围的Maven依赖，对于编译和测试classpath有效，但在运行时无效。
    - runtime　运行时依赖范围。使用此依赖范围的Maven依赖，对于测试和运行classpath有效，但在编译主代码时无效。
    - system　系统依赖范围。该依赖与三种classpath的关系，和provided依赖范围完全一致，但是，使用system范围的依赖时必须通过systemPath元素显式地指定依赖文件的路径。
    - 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。

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

  - optional　标记依赖是否可选　可选依赖不会被依赖传递
  - exclusions　用来排除传递性依赖　依赖中使用exclusions元素声明排除依赖，exclusions可以包含一个或者多个exclusion子元素，因此可以排除一个或者多个传递性依赖，需要注意的是，声明exclusion的时候只需要groupId和artifactId，而不需要version元素，这是因为只需要groupId和artifactId就能唯一定位依赖图中的某个依赖。
  - 依赖传递规则
    - 假设 A依赖B，B依赖C，那么我们说A和B是的第一直接依赖关系，B和C 是第二直接依赖关系。
    - 左侧第一列为第一依赖，第一行为第二依赖
  - 依赖调节规则
    - 最短路径优先原则　如果有两个不同版本的依赖时取依赖路径最短版本
    - 如果两个版本路径一样长，则取决与POM中依赖声明的顺序
  - 分析依赖的命令
    - mvn dependency:list　解析成依赖列表
    - mvn dependency:tree　解析成依赖树
    - mvn dependency:analyze　解析会将如下依赖打印出来
      - Used undeclared dependencies　使用到的未显示声明的依赖
      - Unused declared dependencies　未使用到的已显示声音的依赖
  - 依赖的聚合

```
<modules>
    <module>子模块1</module>
    <module>子模块2</module>
</modules>
```

  - 依赖继承

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.6.RELEASE</version>
</parent>
```

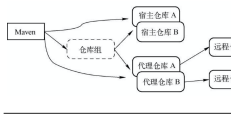
  - 依赖解析规则
    - 依赖包含版本号
      - RELEASE版本
        - 1.根据依赖坐标计算仓库路径，尝试从本地仓库寻找构件，如果有则成功
        - 2.本地仓库找不到时，则遍历所有远程仓库，发现后，下载并解析
      - SNAPSHOT版本
        - 1.根据依赖坐标计算仓库路径，尝试从本地仓库寻找构件，如果有则成功
        - 2.本地仓库找不到时，则遍历所有远程仓库，根据更新策略，读取所有仓库元数据，合并至本地仓库，然后计算SNAPSHOT的最新值，如果最后解析得到的构件版本是时间戳格式的快照，如1.4.1-20091104.121450-121，则复制其时间戳格式的文件至非时间戳格式，如SNAPSHOT，并使用该非时间戳格式的构件。
    - 依赖不包含版本号
      - 1.根据依赖坐标计算仓库路径，尝试从本地仓库寻找构件，如果有则成功
      - 2.本地仓库找不到时，则遍历所有远程仓库，根据更新策略，读取所有仓库元数据，合并至本地仓库，然后计算RELEASE的最新值，然后按照有版本号的方法进行解析(重复上面的RELEASE的1、2)
    - 注意 此处忽略依赖范围为syste的，当 依赖范围为system时则直接从本地解析构件
    - 在解析版本时如果版本号为数值则按数值排序，如果版本号为字符则按字典排序

版本控制

- 理想的发布版本
  - 所有自动化测试应当全部通过。
  - 项目没有配置任何快照版本的依赖。
  - 项目没有配置任何快照版本的插件。
  - 项目所包含的代码已经全部提交到版本控制系统中。
- 版本号格式
  - <主版本>.<次版本>.<增量版本>-<里程碑版本>
    - 主版本：表示了项目的重大架构变更。例如，Maven 2和Maven 1相去甚远；Struts 1和Struts 2采用了不同的架构；JUnit 4和JUnit 3增加了标注支持"
    - 次版本：表示较大范围的功能增加和变化，及Bug修复。例如Nexus 1.5较1.4添加了LDAP的支持，并修复了很多Bug。但从总体架构来说，没有什么变化。
    - 增量版本：一般表示重大Bug的修复。例如项目发布了1.4.0版本之后，发现了一个影响功能的重大Bug，则应该快速发布一个修复了Bug的1.4.1版本。
    - 里程碑版本：顾名思义，这往往指某一个版本的里程碑。例如，Maven 3已经发布了很多里程碑版本，如3.0-alpha-1、3.0-alpha-2、3.0-beta-1等，这样的版本与正式的3.0相比，往往表示不是非常稳定，还需要很多测试。
  - 例如：1.3.4-beta-2

私服

- 安装方式
  - 自带jetty，可直接启动使用　jar包
  - 需要放入服务器中间件中　war包
- 类型
  - group 仓库组
  - hosted 宿主仓库
  - proxy 代理仓库
  - virtual 虚拟仓库
- 区别



插件

- Plugin Goal
  - Maven的核心仅仅定义了抽象的生命周期，具体的任务是交由插件完成的
  - 通用的写法，冒号前面是插件前缀，冒号后面是该插件的目标，例如可以写出compiler：compile
  - 命令格式 mvn [options] [goals] [phase]

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

- 默认的绑定关系
  - 如果在pom 中没有指定插件版本则继承超级pom中插件的版本

生命周期

- clean 生命周期
  - 1）pre-clean执行一些清理前需要完成的工作。
  - 2）clean清理上一次构建生成的文件。
  - 3）post-clean执行一些清理后需要完成的工作
- default 生命周期
  - validate　验证项目是否正确以及是否有所有必要的信息
  - initialize　初始化构建状态，例如设置属性或创建目录。
  - generate-sources　生成任何要包含在编译中的源代码。
  - process-sources　处理项目主资源文件。一般来说，是对src/main/resources目录的内容进行变量替换等工作后，复制到项目输出的主classpath目录中。
  - generate-resources　生成资源以包含在包中
  - process-resources　将资源复制并处理到目标目录中，以备打包。
  - compile　编译项目的主源码。一般来说，是编译src/main/java目录下的java文件至项目输出的主classpath目录中。
  - process-classes　对编译后生成的文件进行后处理，例如对java类进行字节码增强。
  - generate-test-sources　生成任何测试源代码以包含在编译中。
  - process-test-sources　处理项目测试资源文件。一般来说，是对src/test/resources目录的内容进行变量替换等工作后，复制到项目输出的测试classpath目录中。
  - generate-test-resources　创建测试资源。
  - process-test-resources　将资源复制并处理到测试目标目录中。
  - test-compile　编译项目的测试代码。一般来说，是编译src/test/java目录下的java文件至项目输出的测试classpath目录中。
  - process-test-classes　对测试编译生成的文件进行后处理，例如对java类进行字节码增强。
  - test　使用单元测试框架运行测试，测试代码不会被打包或部署。
  - prepare-package　在实际包装之前执行准备包装所需的任何操作。这通常会导致包装的未压缩，已处理版本。
  - package　接受编译好的代码，打包成可发布的格式，如JAR
  - pre-integration-test　在执行集成测试之前执行所需的操作。这可能涉及诸如设置所需环境的事情。
  - integration-test　处理该程序包，并在必要时将其部署到可以运行集成测试的环境中。
  - post-integration-test　在执行集成测试后执行所需的操作。这可能包括清理环境。
  - verify　对集成测试的结果进行任何检查，以确保符合质量标准
  - install　将包安装到Maven本地仓库，供本地其他Maven项目使用。
  - deploy　将最终的包复制到远程仓库，供其他开发人员和Maven项目使用。
- site 生命周期
  - pre-site　执行一些在生成项目站点之前需要完成的工作。
  - site　生成项目站点文档。
  - post-site　执行一些在生成项目站点之后需要完成的工作。
  - site-deploy　将生成的项目站点发布到服务器上。
- 详细请看官网
  - http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

仓库

- 本地仓库
  - 默认情况下在用户的'~/.m2/repository'
  - 如果要自定义仓库则在 settings.xml 中设值 <localRepository>
- 远程仓库
  - 中央仓库
    - M2_HOME/lib/maven-model-builder-3.0.jar /org/apache/maven/model/pom-4.0.0.xml 这个为超级POM
  - 私服

```
<repositories>
  <repository>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
    <layout>default</layout>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

```
<repository>
    <id>nexus</id>
    <name>infisa nexus Repository</name>
    <url>http://192.168.1.62:8081/nexus/content/groups/public/</url>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>ignore</checksumPolicy>
    </snapshots>
</repository>
```

  - 其他公共库
  - updatePolicy　为从远程仓库的更新策略
    checksumPolicy　为校验文件策略
  - 认证信息

```
<settings>
  ...
  <servers>
  <server>
    <id>my-proj</id>
    <username>repo-user</username>
    <password>repo-pwd</password>
  </server>
</servers>
...
</settings>
```

  - 远程仓库部署配置

```
<project>
  ...
  <distributionManagement>
    <repository>
      <id>proj-releases</id>
      <name>Proj Release Repository</name>
      <url>http://192.168.1.100/content/repositories/proj-releases</url>
    </repository>
    <snapshotRepository>
      <id>proj-snapshots</id>
      <name>Proj Snapshot Repository</name>
      <url>http://192.168.1.100/content/repositories/proj-snapshots</url>
    </snapshotRepository>
  </distributionManagement>
  ...
</project>
```

- 仓库布局策略　大致对应关系为groupId/artifactId/version/artifactId-version.packaging
- 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>
```

XMind：ZEN | Trial Mode