Maven 面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Maven 面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

> ：实际上，面试中基本不会问 Maven 。因为，也没啥好问的。所以，本文胖友可以作为啥呢？我也不造，就当简单过过一些知识点吧。

Maven 是什么?

Maven 主要服务于基于 Java 平台的项目构建、依赖管理和项目信息管理。

Maven 的主要功能主要分为 5 点：

- 依赖管理系统
- 多模块构建
- 一致的项目结构
- 一致的构建模型和插件机制

📣 你们项目为什么选用 Maven 进行构建?

- 首先，Maven 是一个优秀的项目构建工具。使用maven，可以很方便的对项目进行分模块构建，这样在开发和测试打包部署时，效率会提高很多。
- 其次，Maven 可以进行依赖的管理。使用 Maven ，可以将不同系统的依赖进行统一管理，并且可以进行依赖之间的传递和继承。

📣 Maven 规约是什么?

- `/src/main/java/` ： Java 源码。
- `/src/main/resource` ： Java 配置文件，资源文件。
- `/src/test/java/` ： Java 测试代码。
- `/src/test/resource` ： Java 测试配置文件，资源文件。
- `/target` ： 文件编译过程中生成的 `.class` 文件、jar、war 等等。
- `pom.xml` ： 配置文件

Maven 要负责项目的自动化构建，以编译为例，Maven 要想自动进行编译，那么它必须知道 Java 的源文件保存在哪里，这样约定之后，不用我们手动指定位置，Maven 能知道位置，从而帮我们完成自动编译。

遵循**“约定>>>配置>>>编码”**。即能进行配置的不要去编码指定，能事先约定规则的不要去进行配置。这样既减轻了劳动力，也能防止出错。

Maven 常用命令

- `mvn archetype: create` ： 创建 Maven 项目。
- `mvn compile` ： 编译源代码。

- `mvn deploy` ： 发布项目。
- `mvn test-compile` ： 编译测试源代码。
- `mvn test` ： 运行应用程序中的单元测试。
- `mvn site` ： 生成项目相关信息的网站。
- `mvn clean` ： 清除项目目录中的生成结果。
- `mvn package` ： 根据项目生成的 jar/war 等。
- `mvn install` ： 在本地 Repository 中安装 jar 。
- `mvn eclipse:eclipse` ： 生成 Eclipse 项目文件。
- `mvn jetty:run` 启动 Jetty 服务。
- `mvn tomcat:run` ： 启动 Tomcat 服务。
- `mvn clean package -Dmaven.test.skip=true` ： 清除以前的包后重新打包，跳过测试类。

用到最多的命令

- `mvn eclipse:clean` ： 清除 Project 中以前的编译的东西，重新再来。
- `mvn eclipse:eclipse` ： 开始编译 Maven 的 Project 。
- `mvn clean package` ： 清除以前的包后重新打包。

Maven 有哪些优点和缺点

📣 1）优点

- 简化了项目依赖管理。

  > 当年，多少人被 SSH 整合搞死搞活，很多时候，是因为依赖不完整，或者版本不正确。自从 Maven 出来后，终于可以无痛了~当然，也有一部分功劳是 Spring Boot ，这是后话。

- 易于上手，对于新手可能一个 `mvn clean package` 命令就可能满足我们的工作。
- 便于与持续集成工具(Jenkins)整合。
- 便于项目升级，无论是项目本身升级还是项目使用的依赖升级。
- 有助于多模块项目的开发，一个模块开发好后，发布到仓库，依赖该模块时可以直接从仓库更新，而不用自己去编译。
- Maven 有很多插件，便于功能扩展，比如生产站点，自动发布版本等。

📣 2）缺点

- Maven 是一个庞大的构建系统，学习难度大。

  > 这里的学习，更多指的完整学习。如果基本使用，并不会存在该问题。

- Maven 采用约定优于配置的策略(convention over configuration)，虽然上手容易，但是一旦出了问题，难于调试。

  > 这个确实，略微痛苦。

- 当依赖很多时，m2eclipse 老是搞得 Eclipse 很卡。

  > 使用 IDEA ， 而不是 Eclipse ， 完美解决。

- 中国的网络环境差，很多 repository 无法访问，比如 Google Code、 JBoss 仓库无法访问等。

  > 这个也好解决，在 `<mirrors>` 中增加阿里巴巴的 Maven 私服，具体可以参见 《提高 Maven 速度——Maven 仓库修改成国内阿里巴巴地址》 文章。

对比其它构建工具

📣 什么是构建?

> ：这个回答，也适用于「你们项目为什么选用 Maven 进行构建？」。

我们都知道，写完代码之后需要进行编译和运行，以笔者自身为例，使用 IDE 写完代码，需要进行编译，再生成 war 包，以便部署到 Tomcat。

在编写 Java 代码的时候，我们除了需要调用 JDK 的 API ，还需要调用许多第三方的 API ，加入没有构建工具，你需要把这些 jar 包下载到本地，然后添加进入工程，在 IDE 中进行添加设置。这种方式非常繁琐，并且在遇到版本升级，Git 同步等时候，程序会变得非常脆弱，极易产生未知错误。所以便有了构建工具的产生，它可以让我们专注于写代码，而不需要考虑如何导入 jar 包，如何升级 jar 包版本，以及 git 多人协作等等问题。这是在编译过程中的优势，在运行和发布的过程中，构建工具依然可以帮助我们将工程生成指定格式的文件。

Java 世界中主要有三大构建工具：Ant、Maven和Gradle。经过几年的发展，Ant 几乎销声匿迹，Maven 也日薄西山，而 Gradle 的发展则如日中天。

📣 Maven 和 Gradle 对比?

详细的，参见《Maven 和 Gradle 对比》文章。

📣 Maven 和 Ant 有什么区别?

详细的，参见文章：

> 应该蛮多人，学习的时候，已经不考虑 Ant 这个东西了。

- 《Java 面试题：Maven 和 ANT 有什么区别？》
- 《Java 基础面试题 Ant 和 Maven》

Maven 坐标的含义?

```
<!-- FROM https://github.com/junit-team/junit4/blob/master/pom.xml -->
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.13-BETA</version>
```

Maven 给我们制定了一套规则 —— 使用坐标进行**唯一标识**。Maven 的坐标元素包括 groupId、artifactId、version、packaging、classfier 。

只要我们提供正确的坐标元素，Maven 就能找到对应的构件，首先去你的本地仓库查找，没有的话再去远程仓库下载。如果没有配置远程仓库，会默认从中央仓库地址(http://repo1.maven.org/maven2)下载构件，该中央仓库包含了世界上大部分流行的开源项目构件，但不一定所有构件都有。

在我们自己开发项目的时候，也是要给我们的项目定义坐标的，这是强制性要求，只有这样，其他项目才能引用该项目的构件。

- groupId：定义当前 Maven 项目隶属的实际项目。
  - 首先，Maven 项目和实际项目不一定是一对一的关系。比如 Spring FrameWork 这一实际项目，其对应的 Maven 项目会有很多，如 `spring-core`、`spring-context` 等。这是由于 Maven 中模块的概念，因此，一个实际项目往往会被划分成很多模块。

    - 其次，groupId 不应该对应项目隶属的组织或公司。原因很简单，一个组织下会有很多实际项目，如果 groupId 只定义到组织级别，而后面我们会看到，artifactId 只能定义 Maven 项目(模块)，那么实际项目这个层次将难以定义。
    - 最后，groupId 的表示方式与 Java 包名的表达方式类似，通常与域名反向一一对应。
    - 上例中，groupId 为 `junit` ，是不是感觉很特殊，这样也是可以的，因为全世界就这么个 junit ， 它也没有很多分支。
- artifactId：该元素定义当前实际项目中的一个 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 junit ， junit 就是实际的项目名称，方便而且直观。
    - 在默认情况下，Maven 生成的构件，会以 artifactId 作为文件头。例如 `junit-3.8.1.jar` ，使用实际项目名称作为前缀，就能方便的从本地仓库找到某个项目的构件。
- version：该元素定义了使用构件的版本。
    - 如上例中 junit 的版本是 `4.13-BETA` ，你也可以改为 `4.1.2` 表示使用 `4.1.2` 版本的 junit 。
- packaging：定义 Maven 项目打包的方式，使用构件的什么包。打包方式通常与所生成构件的文件扩展名对应。
    - 如上例中没有 `packaging` ，则默认为 jar 包，最终的文件名为 `junit-4.13-BETA.jar` 。
    - 当然，也可以打包成 war 等。
- classifier：该元素用来帮助定义构建输出的一些附件。附属构件与主构件对应。
    - 如上例中的主构件为 `junit-4.13-BETA.jar` ，该项目可能还会通过一些插件生成如 `junit-4.13-BETA-javadoc.jar` 、 `junit-4.13-BETA-sources.jar` ，这样附属构件也就拥有了自己唯一的坐标。

上述 5 个元素中：

- `groupId` 、 `artifactId` 、 `version` 是必须定义的。
- `packaging` 是可选的(默认为 jar)。
- 而 `classfier` 是不能直接定义的，需要结合插件使用。

Maven 版本规则?

Maven 主要是这样定义版本规则的： `<主版本>.<次版本>.<增量版本>` 。比如说 `1.2.3` ，主版本是 1 ，次版本是 2 ，增量版本是 3 。

- 主版本，一般来说代表了项目的重大的架构变更，比如说 Maven 1 和 Maven 2 ，在架构上已经两样了，将来的 Maven 3 和 Maven 2 也会有很大的变化。
- 次版本，一般代表了一些功能的增加或变化，但没有架构的变化，比如说Nexus 1.3 较之于 Nexus 1.2 来说，增加了一系列新的或者改进的功能（仓库镜像支持，改进的仓库管理界面等等），但从大的架构上来说，1.3 和 1.2 没什么区别。
- 增量版本，一般是一些小的 bug fix ， 不会有重大的功能变化。

一般来说，在我们发布一次重要的版本之后，随之会开发新的版本。比如说， `myapp-1.1` 发布之后，就着手开发 `myapp-1.2` 了。由于 `myapp-1.2` 有新的主要功能的添加和变化，在发布测试前，它会变得不稳定，而 `myapp-1.1` 是一个比较稳定的版本，现在的问题是，我们在 `myapp-1.1中` 发现了一些 BUG（当然在 1.2 中也存在），为了能够在一段时间内修复 BUG 并仍然发布稳定的版本，我们就会用到分支(branch)，我们基于 1.1 开启一个分支 1.1.1 ，在这个分支中修复 BUG ，并快速发布。这既保证了版本的稳定，也能够使bug得到快速修复，也不同停止 1.2 的开发。只是，每次修复分支 1.1.1 中的 BUG 后，需要 merge 代码到 1.2 中。

多模块如何聚合?

配置一个打包类型为 `pom` 的聚合模块，然后在该 `pom` 中使用元素声明要聚合的模块。具体的，参见 《Maven系列五：多模块项目中的聚合和继承》 文章。

Maven `<dependencie />` 是什么?

`<dependencie />`，依赖关系。属性如下：

- `groupId`：依赖项的 `groupId`。
- `artifactId`：依赖项的 `artifactId`。
- `version`：依赖项的 `version`。
- `scope`：依赖项的适用范围。

  > - `compile`：默认值，适用于所有阶段（开发、测试、部署、运行），本 jar 会一直存在所有阶段。
  > - `provided`：只在开发、测试阶段使用，目的是不让 Servlet 容器和你本地仓库的 jar 包冲突。如 `servlet.jar`。
  > - `runtime`：只在运行时使用，如 JDBC 驱动，适用运行和测试阶段。
  > - `test`：只在测试时使用，用于编译和运行测试代码，不会随项目发布。
  > - `system`：类似 `provided`，需要显式提供包含依赖的 jar 包，Maven 不会在 Repository 中查找它。
  > - `import`：用于一个 `<dependencyManagement />` 对另一个 `<dependencyManagement />` 的继承。非常重要，通过它，可以实现类似 《Maven Spring BOM (bill of materials)》 的功能。

- `exclusions`：排除项目中的依赖冲突时使用。

和 区别是什么?

- `<dependencyManagement />`，统一了 Maven 中依赖的版本号，定义在 `dependencie />` 中的依赖，在不指定具体版本号时，就会沿着上层找到 `<dependencyManagement />` 中的依赖，并使用它的版本号。这样的话，当有多个子项目引用同一个依赖时，就不需要重复声明各自的版本号，只需统一使用 `<dependencyManagement />` 中的版本号即可。
- 还有个不同点，`<dependencyManagement />` 中出现的依赖，并不一定会在项目中使用，而 `<dependencie />` 中的依赖，肯定是包含在项目中的。

对于一个多模块项目，如果管理项目依赖的版本?

- 方式一，通过在父模块中声明 `<dependencyManagement />` 和 `<pluginManagement />`，然后让子模块通过元素指定父模块，这样子模块在定义依赖是就可以只定义 `groupId` 和 `artifactId`，自动使用父模块的 `version`，这样统一整个项目的依赖的版本。

  > 继承的方式。

- 方式二，使用 `<dependencie />` 声明 `<scope />` 为 `import` 的依赖，从而引入一个 pom 的 `<dependencyManagement />` 的。具体的，可以看看 《Maven Spring BOM (bill of materials)》 文章。

  > 组合的方式。

Maven 依赖的解析机制是怎么样的?

1. 解析发布(RELEASE)版本：如果本地有，直接使用本地的，没有就向远程仓库请求。
2. 解析快照(SNAPSHOT)版本：合并本地和远程仓库的元数据文件 `groupId/artifactId/version/maven-metadata.xml`，这个文件存的版本都是带时间戳的，将最新的一个改名为不带时间戳的格式供本次编译使用。

3. 解析版本为 LATEST 过于复杂，且解析的结果不稳定，不推荐在项目中使用，感兴趣的同学自己去研究，简而言之就是合并 `groupId/artifactId/maven-metadata.xml` 找到对应的最新版本和包含快照的最新版本。

📣 LASTEST、RELEASE、SNAPSHOT 的区别?

- LASTEST： 是指某个特定构件最新的发布版或者快照版(SNAPSHOT)，最近被部署到某个特定仓库的构件。
- RELEASE： 是指仓库中最后的一个非快照版本。
- SNAPSHOT： 泛指。如果不 SNAPSHOT ，如果名字不变，本地有了不会从远程拉。如果每次更新都改名字，其他用的人也都改名字，太蛋疼了。

> 关于 SNAPSHOT 版本，建议看看 《理解 Maven 中的 SNAPSHOT 版本和正式版本》 。

📣 Maven 依赖原则?

- 1、赖路径最短优先原则。

> 一个项目 Demo 依赖了两个 jar 包，其中 `A-B-C-X(1.0)` ， `A-D-X(2.0)` 。由于 `X(2.0)` 路径最短，所以项目使用的是 `X(2.0)` 。

- 2、pom文 件中申明顺序优先。

> 如果 `A-B-X(1.0)` ， `A-C-X(2.0)` 这样的路径长度一样怎么办呢？这样的情况下，Maven 会根据 pom 文件声明的顺序加载，如果先声明了 B ， 后声明了 C ， 那就最后的依赖就会是 `X(1.0)` 。

- 3、覆写优先

> 子 pom 内声明的优先于父 pom 中的依赖。

📣 如何解决 jar 冲突?

遇到冲突的时候第一步，要找到 Maven 加载的到时是什么版本的 jar 包，通过们 `mvn dependency:tree` 查看依赖树，或者使用 IDEA Maven Helper 插件。

然后，通过 Maven 的依赖原则来调整坐标在 pom 文件的申明顺序是最好的办法，或者使用将冲突中不想要的 jar 引入的 jar 进行 `<exclusions>` 掉。

Maven 生命周期是怎么样的?

Maven 中有三个独立的生命周期：

- 1、Clean
- 2、Default
- 3、Site

> 每个生命周期都有这个特点：不管用户要求执行的命令对应生命周期中的哪一个阶段，Maven都会自动从当前生命周期的最初位置开始执行，直到完成用户下达的指令

一个完整的项目构建过程通常包括清理、编译、测试、打包、集成测试、验证、部署等步骤，Maven 从中抽取了一套完善的、易扩展的生命周期。Maven 的生命周期是抽象的，其中的具体任务都交由插件来完成。Maven 为大多数构建任务编写并绑定了默认的插件，如针对编译的插件：`maven-compiler-plugin` 。用户也可自行配置或编写插件。

Maven有三套相互独立的生命周期，分别是 Clean、Default 和 Site。每个生命周期包含一些阶段，阶段是有顺序的，后面的阶段依赖于前面的阶段。

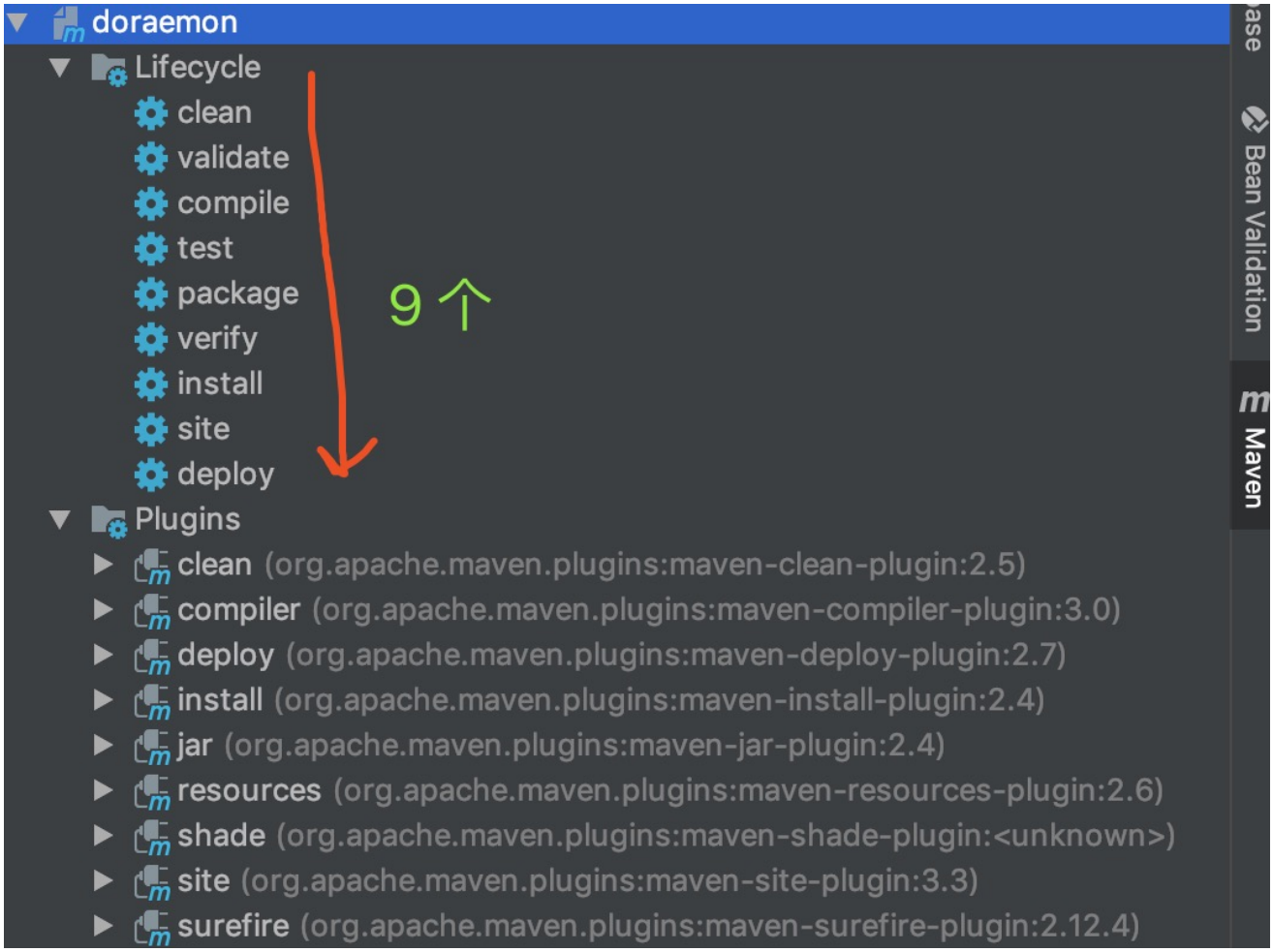
- 1、Clean 生命周期：清理项目，包含三个 phase ：

    - pre-clean：执行清理前需要完成的工作。
    - clean：清理上一次构建生成的文件。
    - post-clean：执行清理后需要完成的工作
- 2、Default 生命周期：构建项目，重要的 phase 如下：
    - validate：验证工程是否正确，所有需要的资源是否可用。
    - compile：编译项目的源代码。
    - test：使用合适的单元测试框架来测试已编译的源代码。这些测试不需要已打包和布署。
    - package：把已编译的代码打包成可发布的格式，比如 jar、war 等。
    - integration-test：如有需要，将包处理和发布到一个能够进行集成测试的环境。
    - verify：运行所有检查，验证包是否有效且达到质量标准。
    - install：把包安装到maven本地仓库，可以被其他工程作为依赖来使用。
    - deploy：在集成或者发布环境下执行，将最终版本的包拷贝到远程的repository，使得其他的开发者或者工程可以共享。
- 3、Site 生命周期：建立和发布项目站点，phase 如下：
    - pre-site：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：生成项目站点之后需要完成的工作
    - site-deploy：将项目站点发布到服务器

各个生命周期相互独立，一个生命周期的阶段前后依赖。

- `mvn clean` ：调用 Clean 生命周期的 clean 阶段，实际执行 pre-clean 和 clean 阶段
- `mvn test` ：调用 Default 生命周期的 test 阶段，实际执行 test 以及之前所有阶段
- `mvn clean install` ：调用 Clean 生命周期的 clean 阶段和 Default 生命周期 的 install 阶段，实际执行 pre-clean 和 clean ， install 以及之前所有阶段。

而我们在 IDEA Maven 中，也可以看到 Maven 生命周期的操作：



- clean： 清理自动生成的文件，也就是 target 目录。
- validate： 验证 Maven 描述文件是否有效。
- compile： 编译 java 代码。
- test： 运行测试代码。
- package： 项目打成 jar、war 包等。
- verify： 验证构件包是否有效。
- install： 将构件包安装到本地仓库。
- deploy： 将构件包部署到远程仓库。
- site： 生成项目站点。

我们经常使用 mvn clean package 命令进行项目打包，请问该命令执行了哪些动作来完成该任务?

> ：这个问题吧，基本不会问的，了解即可。

在这个命令中我们调用了 Maven 的 clean 周期的 clean 阶段绑定的插件任务，以及 default 周期的 package 阶段绑定的插件任务。

默认执行的任务有（Maven的术语叫 goal ，也有人翻译成目标，我这里用任务啦）：

```
`maven-clean-plugin:clean` ->
`maven-resources-plugin:resources` ->
`maven-compile-plugin:compile` ->
`mavne-resources-plugin:testResources` ->
`maven-compile-plugin:testCompile` ->
`maven-jar-plugin:jar`
```

什么是 Maven 插件?

Maven 生命周期的每一个阶段的具体实现都是由 Maven 插件实现的。插件通常提供了一个目标的集合，并且可以使用下面的语法执行：`mvn [plugin-name]:[goal-name]`

Maven 提供了下面两种类型的插件：

- Build plugins： 在构建时执行，并在 `pom.xml` 的 元素中配置。
- Reporting plugins： 在网站生成过程中执行，并在 `pom.xml` 的元素中配置。

下面是一些常用插件的列表：

- clean： 构建之后清理目标文件。删除目标目录。
- compiler： 编译 Java 源文件。
- surefile： 运行 JUnit 单元测试。创建测试报告。
- jar： 从当前工程中构建 JAR 文件。
- war： 从当前工程中构建 WAR 文件。
- javadoc： 为工程生成 Javadoc 。
- antrun： 从构建过程的任意一个阶段中运行一个 ant 任务的集合。

如何实现自定义插件?

大多数情况下，我们不太需要开发自定义的 Maven 插件，并且面试一般也不会问。当然，感兴趣的胖友，可以看看 《Maven 自定义插件开发》 。

Maven 插件的解析机制?

> ：这个问题，选择性理解即可。

当我们输入 `mvn dependency:tree` 这样的指令，解析的步骤为：

- 1、解析 `groupId`：
  > Maven 使用默认的 groupId 插件为 `org.apache.maven.plugins` 或者 `org.codehaus.mojo`。
- 2、解析 `artifactId` (Maven 的官方叫做插件前缀解析策略)：
  > 合并该 `groupId` 在所有仓库中的元数据库文件（`maven-metadata-repository.xml`），比如 Maven 官方插件的元数据文件所在的目录为 `org\apache\maven\plugins`，该文件下有如下的条目：
  >
  > ```
  > <plugin>
  >   <name>Maven Dependency Plugin</name>
  >   <prefix>dependency</prefix>
  >   <artifactId>maven-dependency-plugin</artifactId>
  > </plugin>
  > ```
  - 通过比较这样的条目，我们就将该命令的 `artifactId` 解析为`maven-dependency-plugin`。
- 3、解析 `version`：
  > 如果你在项目的 `pom` 中声明了该插件的版本，那么直接使用该版本的插件，否则合并所有仓库中 `groupId/artifactId/maven-metadata-repository.xml`，找到最新的发布版本。

对于非官方的插件，有如下两个方法可以选择：

- 使用 `groupId:artifactId:version:goal` 来运行。
- 在 `settings.xml` 中添加 `pluginGroup` 项，这样 Maven 不能在官方的插件库中解析到某个插件，那么就可以去你配置的 `group` 下查找啦。

什么是 Maven 仓库?

🚀 Maven 的仓库只有两大类：

- 1、本地仓库。
- 2、远程仓库。在远程仓库中又分成了 3 种：
  - 中央仓库。
  - 私服。
  - 其它公共库。

🚀 Maven 会先搜索本地仓库（repository），发现本地没有然后从远程仓库（中央仓库）获取。

- 但中央仓库只有一个，最好从其镜象处下载。国内可以用阿里云下的服务器。【*其它公共库*】
- 也有通过 Nexus 搭建的私服进行获取的。【*私服*】

🚀 Maven 中的仓库分为两种，SNAPSHOT 快照仓库和 RELEASE 发布仓库。

- SNAPSHOT 快照仓库用于保存开发过程中的不稳定版本，RELEASE 正式仓库则是用来保存稳定的发行版本。定义一个组件/模块为快照版本，只需要在 `pom` 文件中在该模块的版本号后加上 `-SNAPSHOT` 即可(注意这里必须是大写)，如下：

```
<groupId>cc.mzone</groupId>
<artifactId>m1</artifactId>
<version>0.1-SNAPSHOT</version>
<packaging>jar</packaging>
```

- Maven 会根据模块的版本号(`pom` 文件中的 `version`)中是否带有 `-SNAPSHOT` 来判断是快照版本还是正式版本。
  - 如果是快照版本，那么在 `mvn deploy` 时会自动发布到快照版本库中，会覆盖老的快照版本。而在使用快照版本的模块，在不更改版本号的情况下，直接编译打包时，Maven 会自动从镜像服务器上下载最新的快照版本。
  - 如果是正式发布版本，那么在 `mvn deploy` 时会自动发布到正式版本库中，而使用正式版本的模块，在不更改版本号的情况下，编译打包时如果本地已经存在该版本的模块则不会主动去镜像服务器上下载。

所以，我们在开发阶段，可以将公用库的版本设置为快照版本，而被依赖组件则引用快照版本进行开发，在公用库的快照版本更新后，我们也不需要修改 pom 文件提示版本号来下载新的版本，直接 mvn 执行相关编译、打包命令即可重新下载最新的快照库了，从而也方便了我们进行开发。

📣 什么是私服?

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。我们还可以把一些无法从外部仓库下载到的构件上传到私服上。

🚀 Maven 私服的 5 个特性：

- 1、节省自己的外网带宽：减少重复请求造成的外网带宽消耗。
- 2、加速 Maven 构件：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。
- 3、部署第三方构件：有些构件无法从外部仓库获得的时候，我们可以把这些构件部署到内部仓库(私服)中，供内部 Maven 项目使用。
- 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。
- 5、降低中央仓库的负荷：Maven 中央仓库被请求的数量是巨大的，配置私服也可以大大降低中央仓库的压力。

🚀 当前主流的 Maven 私服：

- Apache 的 Archiva
- JFrog 的 Artifactory
- 【主流】Sonatype 的 Nexus 。

🚀 常见的 Maven 私服的仓库类型：

- （宿主仓库）hosted repository 。
- （代理仓库）proxy repository 。
- （仓库组）group repository 。

对于大多数公司，一般来说使用 Nexus + 阿里云仓库的方式，可参考如下两篇文章：

- 《使用 Nexus 搭建 Maven 私有仓库》
- 《Nexus 配置阿里云仓库》

📣 如何配置远程仓库?

用文本编辑器工具打开 `setting.xml` 文件，增加一个 `<mirror />`：

> ：此处以阿里云为例。

```
<mirrors>
    <mirror>
        <id>nexus-aliyun</id>
        <mirrorOf>*</mirrorOf>
        <name>Nexus aliyun</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public</url>
    </mirror>
</mirrors>
```

📣 如何使用 Maven 将 jar 包发布到中央仓库?

> 本小节为选读，适合开源作者，想要将自己的库发布到中央仓库，进行共享。

详细的，参见《发布 jar 包到 Maven 中央仓库》文章。

彩蛋

犹记得，11 年第一次在项目中引入 Maven 的惊艳，终于无需使用 Ant 带来的种种痛苦，以及每次更新项目，几百 M 一个，还有搭建项目时需要引入各种版本的依赖。

参考与推荐如下文章：

- 《Maven 面试基本知识点》

- 《JAVA 面试常问知识总结（十） —— Maven》
- 《菜鸟面试必知的 Maven 知识》
- 《面试题之 Maven》