

# Maven Study

## 1. Maven 功能

### Maven 功能

Maven 能够帮助开发者完成以下工作：

- 构建
- 文档生成
- 报告
- 依赖
- SCMs
- 发布
- 分发
- 邮件列表

## 2. 约定配置： 很重要！！用这个文件生成文件夹结构！！！！（文件夹的命名需要按照这个来）！！！！

### 约定配置

Maven 提倡使用一个共同的标准目录结构，**Maven 使用约定优于配置的原则**，大家尽可能的遵守这样的目录结构。如下所示：

目录	目的
<u><code>\${basedir}</code></u>	<u>存放pom.xml和所有的子目录</u>
<code>\${basedir}/src/main/java</code>	项目的java源代码
<u><code>\${basedir}/src/main/resources</code></u>	<u>项目的资源，比如说property文件，springmvc.xml</u>
<code>\${basedir}/src/test/java</code>	项目的测试类，比如说JUnit代码
<code>\${basedir}/src/test/resources</code>	测试用的资源
<code>\${basedir}/src/main/webapp/WEB-INF</code>	web应用文件目录，web项目的信息，比如存放web.xml、本地图片、jsp视图页面
<code>\${basedir}/target</code>	打包输出目录
<code>\${basedir}/target/classes</code>	编译输出目录
<code>\${basedir}/target/test-classes</code>	测试编译输出目录
<code>Test.java</code>	<u>Maven只会自动运行符合该命名规则的测试类</u>
<code>~/m2/repository</code>	Maven默认的本地仓库目录位置

## Maven 的 Snapshot 版本与 Release 版本:



832

### Maven 的 Snapshot 版本与 Release 版本

1、Snapshot 版本代表不稳定、尚处于开发中的版本。

2、Release 版本则代表稳定的版本。

3、什么情况下该用 SNAPSHOT?

协同开发时, 如果 A 依赖构件 B, 由于 B 会更新, B 应该使用 SNAPSHOT 来标识自己。这种做法的必要性可以反证如下:

- a. 如果 B 不用 SNAPSHOT, 而是每次更新后都使用一个稳定的版本, 那版本号就会升得太快, 每天一升甚至每个小时一升, 这就是对版本号的滥用。

- b. 如果 B 不用 SNAPSHOT, 但一直使用一个单一的 Release 版本号, 那当 B 更新后, A 可能并不会接受到更新。因为 A 所使用的 repository 一般不会频繁更新 release 版本的缓存 (即本地 repository), 所以 B 以不换版本号的方式更新后, A 在拿 B 时发现本地已有这个版本, 就不会去远程 Repository 下载最新的 B

4、不用 Release 版本, 在所有地方都用 SNAPSHOT 版本行不行?

不行。正式环境中不得使用 snapshot 版本的库。比如说, 今天你依赖某个 snapshot 版本的第三方库成功构建了自己的应用, 明天再构建时可能会失败, 因为今晚第三方可能已经更新了它的 snapshot 库。你再次构建时, Maven 会去远程 repository 下载 snapshot 的最新版本, 你构建时用的库就是新的 jar 文件了, 这时正确性就很难保证了。

任人欺凌小师妹 5年前 (2018-09-30)

### 3. Maven POM(Project Object Model, 项目对象模型)

POM( Project Object Model, 项目对象模型 ) 是 Maven 工程的基本工作单元，是一个 XML 文件，包含了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。

执行任务或目标时，Maven 会在当前目录中查找 POM。它读取 POM，获取所需的配置信息，然后执行目标。

POM 中可以指定以下配置：

- 项目依赖
- 插件
- 执行目标
- 项目构建 profile
- 项目版本
- 项目开发者列表
- 相关邮件列表信息

在创建 POM 之前，我们首先需要描述项目组 (groupId)，项目的唯一 ID。

在创建 POM 之前，我们首先需要描述项目组 (groupId)，项目的唯一ID。

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- 模型版本 -->
  <modelVersion>4.0.0</modelVersion>
  <!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成，如com.companyname.project-group, maven会将该项目打成的jar包放本地路径: /com/companyname/project-group -->
  <groupId>com.companyname.project-group</groupId>

  <!-- 项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->
  <artifactId>project</artifactId>

  <!-- 版本号 -->
  <version>1.0</version>
</project>
```

所有 POM 文件都需要 project 元素和三个必需字段：groupId，artifactId，version。

节点	描述
project	工程的根标签。
modelVersion	模型版本需要设置为 4.0。

所有 POM 文件都需要 project 元素和三个必需字段：groupId，artifactId，version。

所有 POM 文件都需要 `project` 元素和三个必需字段: `groupId`, `artifactId`, `version`。

节点	描述
project	工程的根标签。
modelVersion	模型版本需要设置为 4.0。
groupId	这是工程组的标识。它在一个组织或者项目中通常是唯一的。例如，一个银行组织拥有所有的和银行相关的项目。 <code>com.companyname.project-group</code>
artifactId	这是工程的标识。它通常是工程的名称。例如，消费者银行。 <code>groupId</code> 和 <code>artifactId</code> 一起定义了 <code>artifact</code> 在仓库中的位置。
version	这是工程的版本号。在 <code>artifact</code> 的仓库中，它用来区分不同的版本。例如： <code>com.company.bank:consumer-banking:1.0</code> <code>com.company.bank:consumer-banking:1.1</code>

Maven 使用 `effective pom` (`Super pom` 加上工程自己的配置) 来执行相关的目标，它帮助开发者在 `pom.xml` 中做尽可能少的配置，当然这些配置可以被重写。

使用以下命令来查看 `Super POM` 默认配置：

```
mvn help:effective-pom
```

接下来我们创建目录 `MVN/project`，在该目录下创建 `pom.xml`，内容如下：

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <!-- 模型版本 -->
  <modelVersion>4.0.0</modelVersion>
  <!-- 公司或者组织的唯一标识，并且配置时生成的路径也是由此生成，如 com.companyname.project-group，maven 会将该项目打成的 jar 包放本地路径：/com/companyname/project-group -->
  <groupId>com.companyname.project-group</groupId>

  <!-- 项目的唯一 ID，一个 groupId 下面可能多个项目，就是靠 artifactId 来区分的 -->
  <artifactId>project</artifactId>

  <!-- 版本号 -->
  <version>1.0</version>
</project>
```

```
“mvn help:effective-pom”
```

在命令控制台，进入 MVN/project 目录，执行以下命令：

```
C:\MVN\project>mvn help:effective-pom
```

Maven 将会开始处理并显示 effective-pom。

```
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.pom
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:36 min
[INFO] Finished at: 2018-09-05T11:31:28+08:00
[INFO] Final Memory: 15M/149M
[INFO] -----
```

Effective POM 的结果就像在控制台中显示的一样，经过继承、插值之后，使配置生效。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===== -->
<!-- -->
```

在上面的 pom.xml 中，你可以看到 Maven 在执行目标时需要用到的默认工程源码目录结构、输出目录、需要的插件、仓库和报表目录。

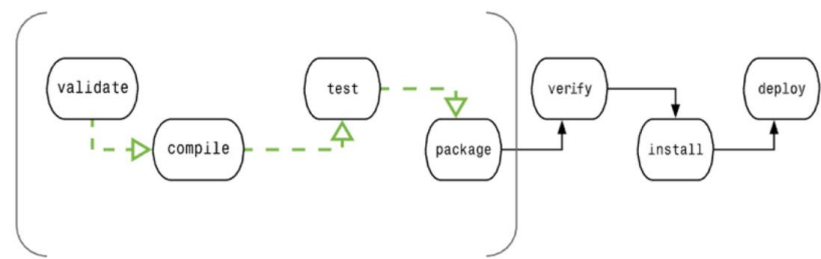
Maven 的 pom.xml 文件也不需要手工编写。

Maven 提供了大量的原型插件来创建工程，包括工程结构和 pom.xml。

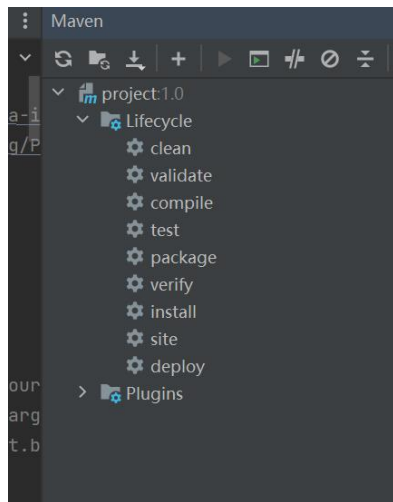
### 4. Maven 构建生命周期

Maven 构建生命周期定义了一个项目构建跟发布的过程。

一个典型的 Maven 构建 (build) 生命周期是由以下几个阶段的序列组成的：



阶段	处理	描述
验证 validate	验证项目	验证项目是否正确且所有必须信息是可用的
编译 compile	执行编译	源代码编译在此阶段完成
测试 Test	测试	使用适当的单元测试框架（例如JUnit）运行测试。
包装 package	打包	将编译后的代码打包成可分发的格式，例如 JAR 或 WAR
检查 verify	检查	对集成测试的结果进行检查，以保证质量达标
安装 install	安装	安装打包的项目到本地仓库，以供其他项目使用
部署 deploy	部署	拷贝最终的工程包到远程仓库中，以共享给其他开发人员和工程



## 4.1. Clean 生命周期

Input:

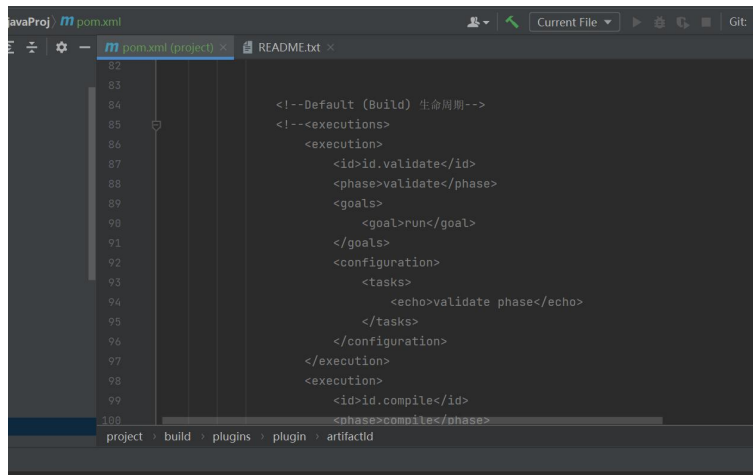
```
<!--Clean 生命周期-->
<!--<executions>
  <execution>
    <id>id.pre-clean</id>
    <phase>pre-clean</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <tasks>
        <echo>pre-clean phase</echo>
      </tasks>
    </configuration>
  </execution>
  <execution>
    <id>id.clean</id>
    <phase>clean</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
```

Output:

```
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

## 4.2. Default (Build) 生命周期

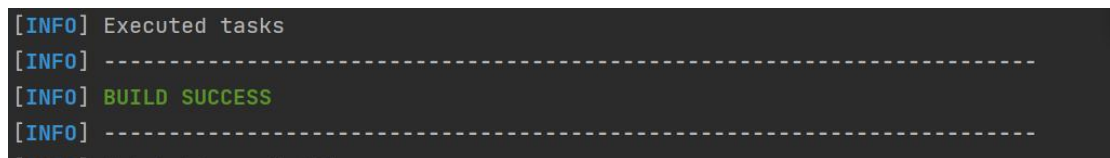
Input:



```
82
83
84 <!--Default (Build) 生命周期-->
85 <!--executions>
86 <execution>
87 <id>id.validate</id>
88 <phase>validate</phase>
89 <goals>
90 <goal>run</goal>
91 </goals>
92 <configuration>
93 <tasks>
94 <echo>validate phase</echo>
95 </tasks>
96 </configuration>
97 </execution>
98 <execution>
99 <id>id.compile</id>
100 <phase>compile</phase>
```

project > build > plugins > plugin > artifactId

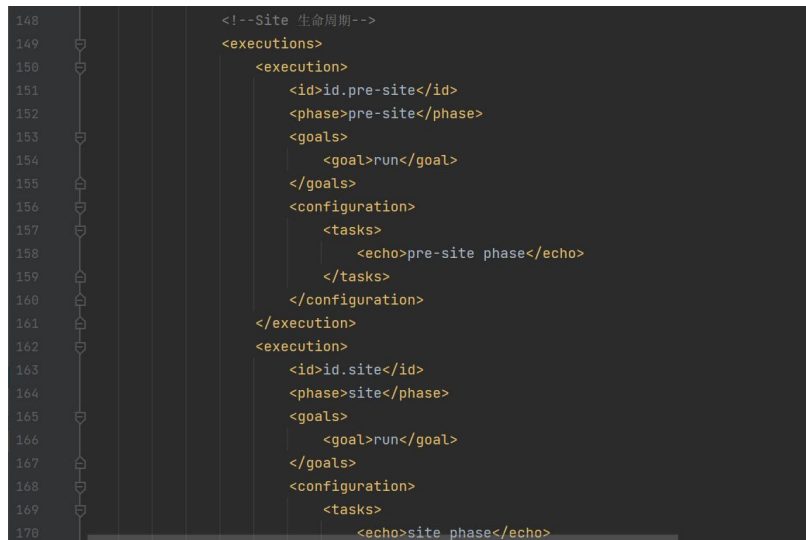
Output:



```
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

## 4.3. Site 生命周期

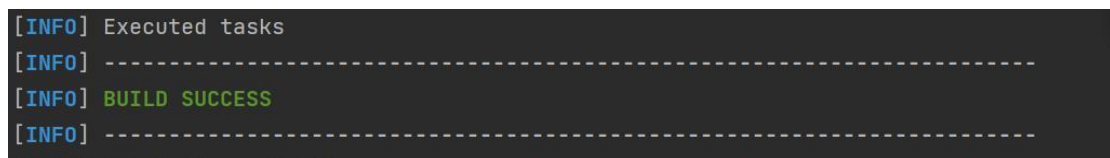
Input:



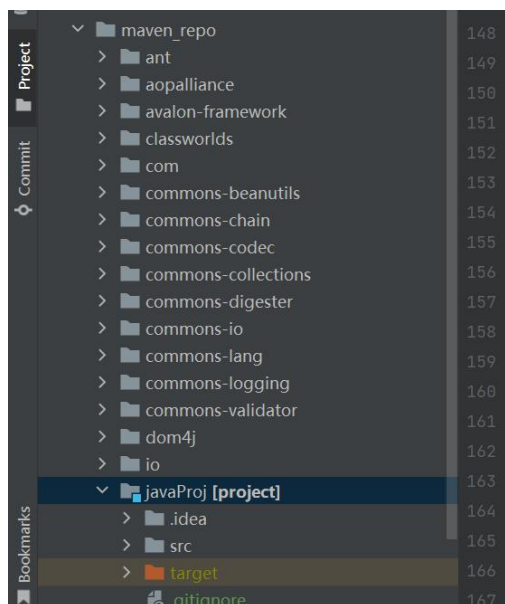
```
148
149 <!--Site 生命周期-->
150 <executions>
151 <execution>
152 <id>id.pre-site</id>
153 <phase>pre-site</phase>
154 <goals>
155 <goal>run</goal>
156 </goals>
157 <configuration>
158 <tasks>
159 <echo>pre-site phase</echo>
160 </tasks>
161 </configuration>
162 </execution>
163 <execution>
164 <id>id.site</id>
165 <phase>site</phase>
166 <goals>
167 <goal>run</goal>
168 </goals>
169 <configuration>
170 <tasks>
171 <echo>site phase</echo>
```

project > build > plugins > plugin > artifactId

Output:



```
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```



## 5. Maven 构建配置文件

### 构建配置文件的类型

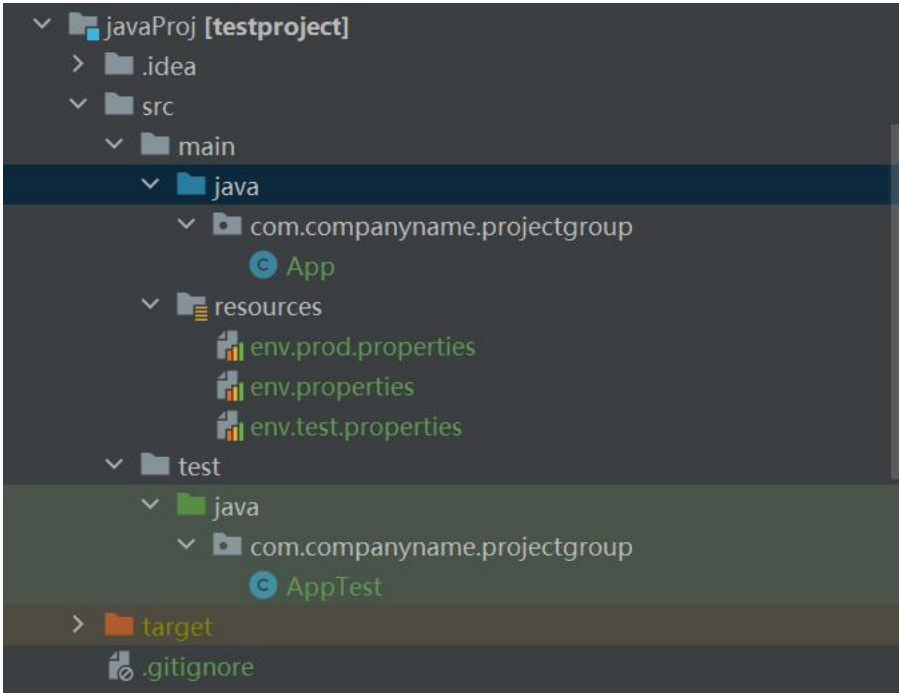
#### 构建配置文件的类型

构建配置文件大体上有三种类型:

类型	在哪定义
项目级 (Per Project)	定义在项目的POM文件pom.xml中
用户级 (Per User)	定义在Maven的设置xml文件中 (%USER_HOME%/m2/settings.xml)
全局 (Global)	定义在 Maven 全局的设置 xml 文件中 (%M2_HOME%/conf/settings.xml)

假定项目结构如下:





其中在src/main/resources文件夹下有三个用于测试文件：

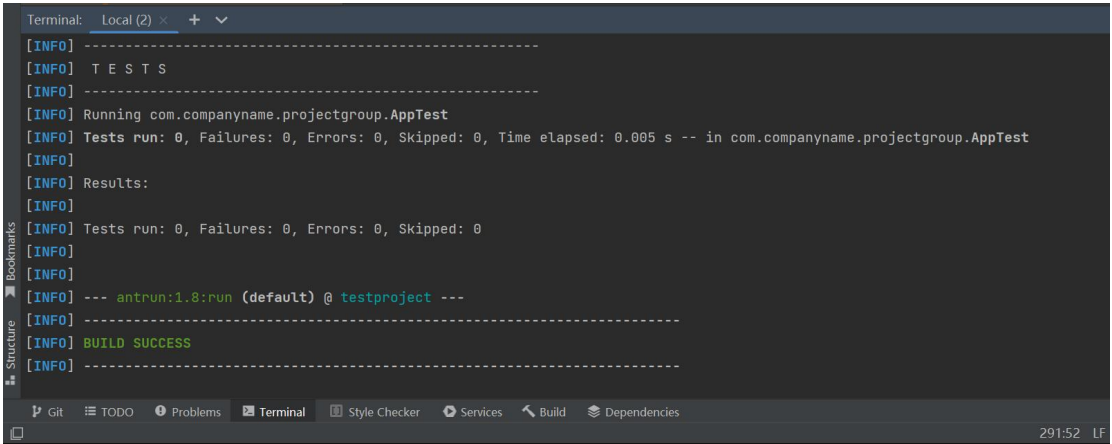
文件名	描述
env.properties	如果未指定配置文件时默认使用的配置。
env.test.properties	当测试配置文件使用时的测试配置。
env.prod.properties	当生产配置文件使用时的生产配置。

**注意：**这三个配置文件并不是代表构建配置文件的功能，而是用于本次测试的目的；比如，我指定了构建配置文件为 **prod** 时，项目就使用 **env.prod.properties**文件。

**注意：**下面的例子仍然是使用 **AntRun** 插件，因为此插件能绑定 **Maven** 生命周期阶段，并通过 **Ant** 的标签不用编写一点代码即可输出信息、复制文件等，经此而已。其余的与本次构建配置文件无关。

配置文件激活：

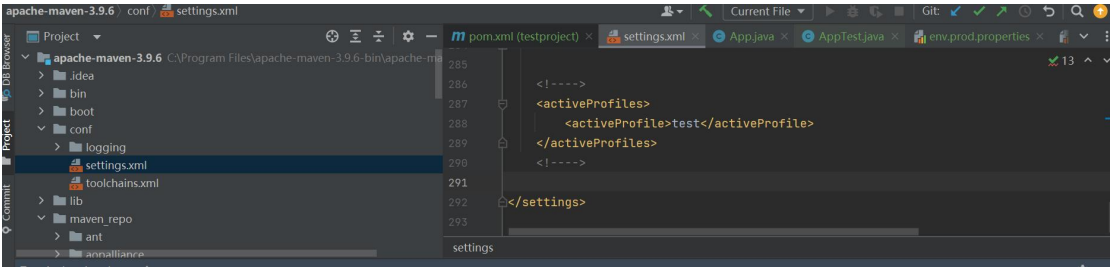
Output: （）



```
[WARNING] Parameter 'tasks' is deprecated: Use target instead
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
  [echo] Using env.test.properties
  [copy] Copying 1 file to C:\Program Files\apache-maven-3.9.6-bin\apache-maven-3.9.6\maven_repo\javaProj\target\classes
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.883 s
[INFO] Finished at: 2024-01-11T10:35:17+08:00
[INFO] -----
PS C:\Program Files\apache-maven-3.9.6-bin\apache-maven-3.9.6\maven_repo\javaProj> $
```

## 2.通过 Maven 设置



### 2、通过Maven设置激活配置文件

打开 %USER\_HOME%\m2 目录下的 settings.xml 文件，其中 %USER\_HOME% 代表用户主目录。如果 settings.xml 文件不存在就复制 %M2\_HOME%\conf\settings.xml 到 m2 目录，其中 %M2\_HOME% 代表 Maven 的安装目录。

配置 settings.xml 文件，增加 <activeProfiles>属性：

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <activeProfiles>
    <activeProfile>test</activeProfile>
  </activeProfiles>
</settings>
```

执行命令：

```
mvn test
```

提示 1：此时不需要使用 -Ptest 来输入参数了，上面的 settings.xml 文件的 <activeprofile> 已经指定了 test 参数代替了。

提示 2：同样可以使用在 %M2\_HOME%\conf\settings.xml 的文件进行配置，效果一致。

执行结果：

```
D:\开发工程\Github\5_java_example\maventest\test4\testproject
> mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] Building testproject 0.1-SNAPSHOT
[INFO] -----
[INFO]
```

Output:

```
Terminal: Local < + v
[WARNING] Parameter tasks is deprecated, use target instead
[INFO] Executing tasks

main:
  [echo] Using env.test.properties
  [copy] Copying 1 file to C:\Program Files\apache-maven-3.9.6-bin\apache-maven-3.9.6\maven_repo\javaProj\target\classes
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.239 s
[INFO] Finished at: 2024-01-11T10:35:17+08:00
```

### 3. 通过环境变量激活配置环境：

### 3、通过环境变量激活配置文件

先把上一步测试的 `setting.xml` 值全部去掉。

然后在 `pom.xml` 里面的 `<id>` 为 `test` 的 `<profile>` 节点，加入 `<activation>` 节点：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jsoft.test</groupId>
  <artifactId>testproject</artifactId>
  <packaging>jar</packaging>
  <version>0.1-SNAPSHOT</version>
  <name>testproject</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <property>
          <name>env</name>
```

.....  
.....  
.....

## 6. Maven 插件

### Maven 插件

Maven 有以下三个标准的生命周期：

- **clean**：项目清理的处理
- **default(或 build)**：项目部署的处理
- **site**：项目站点文档创建的处理

每个生命周期中都包含着一系列的阶段(phase)。这些 phase 就相当于 Maven 提供的统一的接口，然后这些 phase 的实现由 Maven 的插件来完成。

我们在输入 `mvn` 命令的时候 比如 `mvn clean`，`clean` 对应的就是 Clean 生命周期中的 `clean` 阶段。但是 `clean` 的具体操作是由 `maven-clean-plugin` 来实现的。

所以说 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。

Maven 实际上是一个依赖插件执行的框架，每个任务实际上是由插件完成。Maven 插件通常被用来：

- 创建 `jar` 文件
- 创建 `war` 文件
- 编译代码文件
- 代码单元测试
- 创建工程文档

# 7. Maven 构建 Java 项目

## Maven 构建 Java 项目：

Maven 使用原型 **archetype** 插件创建项目。要创建一个简单的 Java 应用，我们将使用 **maven-archetype-quickstart** 插件。

在下面的例子中，我们将在 C:\MVN 文件夹下创建一个基于 maven 的 java 应用项目。

命令格式如下：

```
mvn archetype:generate "-DgroupId=com.companyname.bank" "-DartifactId=consumerBanking" "-DarchetypeArtifactId=maven-archetype-quickstart" "-DinteractiveMode=false"
```

参数说明：

- **-DgroupId**: 组织名，公司网址的反写 + 项目名称
- **-DartifactId**: 项目名-模块名
- **-DarchetypeArtifactId**: 指定 ArchetypeId, maven-archetype-quickstart, 创建一个简单的 Java 应用
- **-DinteractiveMode**: 是否使用交互模式

Input:

```
PS C:\Program Files\apache-maven-3.9.6-bin\apache-maven-3.9.6\maven_repo> mvn archetype:generate "-DgroupId=com.companyname.bank" "-DartifactId=consumerBanking" "-DarchetypeArtifactId=maven-archetype-quickstart" "-DinteractiveMode=false"
```

Output: (文件结构如下)

