# pom.xml文件结构详解

<https://github.com/dunwu/java-tutorial/blob/master/docs/javatool/build/maven/maven-pom.md>

## 一、pom简介：

### 1、什么是pom：

POM是Project Object Model的缩写，即项目对象模型。

pom.xml就是Maven的配置文件，用以描述项目的各种信息。

当Maven运行的时候，它向项目对象模型（POM）查看关于这个项目的信息。POM回答类似这样的问题：这个项目是什么类型？这个项目的名称是什么？这个项目的构建有自定义么？

### 2、pom配置一览：

<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">

<modelVersion>4.0.0</modelVersion>

<!-- The Basics -->

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

<packaging>...</packaging>

<dependencies>...</dependencies>

<parent>...</parent>

<dependencyManagement>...</dependencyManagement>

<modules>...</modules>

<properties>...</properties>

<!-- Build Settings -->

<build>...</build>

<reporting>...</reporting>

<!-- More Project Information -->

<name>...</name>

<description>...</description>

<url>...</url>

<inceptionYear>...</inceptionYear>

<licenses>...</licenses>

<organization>...</organization>

<developers>...</developers>

<contributors>...</contributors>

<!-- Environment Settings -->

<issueManagement>...</issueManagement>

<ciManagement>...</ciManagement>

<mailingLists>...</mailingLists>

<scm>...</scm>

<prerequisites>...</prerequisites>

<repositories>...</repositories>

<pluginRepositories>...</pluginRepositories>

<distributionManagement>...</distributionManagement>

<profiles>...</profiles>

</project>

### 3、effective-pom：

当Maven运行的时候，它是根据项目的pom.xml里设置的组合来运行的，一个最上级的POM定义了Maven的安装目录，在这个目录中全局的默认值被定义了，（可能）还有一些用户定义的设置。想要看这个“有效的（effective）”POM，或者说Maven真正运行根据的POM，在项目中运行运行下面的命令：

mvn help:effctive-pom

## 二、project节点：

project是pom.xml中描述符的根。

## 三、modelVersion：

modelVersion指定pom.xml符合哪个版本的描述。Maven2和Maven3只能为4.0.0

## 四、Maven坐标：

在Maven中，根据groupId、artifactId、version组合成groupId:arrifactId:version来唯一标识一个Jar包。

### 1、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者公司。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会很难定义。

最后，groupId的表示方式与Java包名的表示方式类似，通常与域名反向一一对应（因为groupId要求唯一，而正好域名就是唯一的）。上例中，groupId为org.sonatype.nexus，org.sonatype表示Sonatype公司建立的一个非盈利组织，nexus表示Nexus这一实际项目，该groupId与域名nexus.sonatype.org对应。

### 2、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。在默认情况下，Maven生成的构件，其文件名会以artifactId作为开头，如nexus-indexer-2.0.0.jar，使用实际项目名称作为前缀之后，就能方便从一个lib文件夹中找到某个项目的一组构件。

考虑有5个项目（实际项目），每个项目都有一个core模块，如果没有前缀，我们会看到很多core-1.2.jar这样的文件，加上实际项目名前缀之后，便能很容易区分foo-core-1.2.jar、bar-core.1.2.jar……

【注】 不要在artifactId中包含点(.)

### 3、version：

1）maven有自己的版本规范，一般是如下定义 major version、minor verion、incremental version-qualifier，比如1.2.3-beta-01。要说明的是，maven自己判断版本的算法是major、minor、incremental部分用数字比较，qualifier部分用字符串比较，所以要小心alpha-2和alpha-15的比较关系，最好用alpha-02的格式。

2） maven在版本管理时候可以用几个特殊的字符串SNAPSHOT、RELEASE、LATEST。比如1.0-SNAPSHOT。各个部分的含义和处理逻辑如下：

1. SNAPSHOT：这个版本一般用于开发过程中，表示不稳定的版本。
2. LATEST：指某个构建的最新发布，这个发布可能是一个发布版，也可能是一个SNAPSHOT版，具体看哪个时间最后。
3. RELEASE：指最后一个发布版。

### 4、packaging：

该元素定义Maven项目的打包方式。

首先，打包方式通常与所生成构件的文件扩展名对应，如上例中packaging为jar，最终的文件名为nexus-indexer-2.0.0.jar，而使用war打包方式的Maven项目，最终生成的构件会有一个.war文件，不过这不是绝对的。

其次，打包方式会影响到构建的生命周期，比如jar打包和war打包会使用不同的命令。

最后，当不定义packaging的时候，Maven会使用默认值jar。

## 五、依赖配置：

### 1、dependencies：

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

...

<dependencies>

<dependency>

<groupId>org.apache.maven</groupId>

<artifactId>maven-embedder</artifactId>

<version>2.0</version>

<type>jar</type>

<scope>test</scope>

<optional>true</optional>

<exclusions>

<exclusion>

<groupId>org.apache.maven</groupId>

<artifactId>maven-core</artifactId>

</exclusion>

</exclusions>

</dependency>

...

</dependencies>

...

</project>

1. groupId、artifactId、version：和Maven坐标中的意义相同。
2. type：对应packaging的类型，如果不使用type标签，Maven默认为jar。
3. scope：此元素指的是任务的路径（编译和运行时，测试等）以及如何限制依赖关系的传递性。有5种可用的限定范围。
4. compile：如果没有指定scope标签，maven默认这个scope。被依赖的Maven构件在compile、runtime和test的时候都可以在classpath中找到。
5. provided：被依赖的Maven构件在compile和test的时候都可以在classpath中找到，在runtime的时候由JDK或容器提供。
6. runtime：被依赖的Maven构件在runtime和test的时候都可以在classpath中找到，在compile时不是必须的。
7. test：被依赖的Maven构件在test的时候可以在classapath中找到，在compile和runtime时不是必须的。
8. system：被依赖的Maven构件在runtime和test的时候都可以在classpath中找到，在runtime的时候必须显式将jar加入到classpath中。
9. systempath：仅当依赖范围是system时才使用。否则，如果设置此元素，构建将失败。该路径必须是绝对路径，因此建议使用properties来指定特定的路径。
10. optional：当前Maven项目的构建被其它项目依赖，此处被依赖的Maven构建相对于其它项目来说是不必须的。
11. exclusions：包含一个或多个排除元素，每个排除元素都包含一个表示要排除的依赖关系的groupId和artifactId。

### 2、parent：

maven支持继承功能，子POM可以使用parent指定父POM，然后继承其配置。

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.codehaus.mojo</groupId>

<artifactId>my-parent</artifactId>

<version>2.0</version>

<relativePath>../my-parent</relativePath>

</parent>

<artifactId>my-project</artifactId>

</project>

1. relativePath：指定父项目的相对路径。

### 3、dependencyManagement：

dependencyManagement是表示依赖jar包的声明。即你在项目中的dependencyManagement下声明了依赖，maven不会加载该依赖，dependencyManagement声明可以被子POM继承。

dependencyManagement的一个使用案例是当有父子项目的时候，父项目中可以利用dependencyManagement声明子项目中需要用到的依赖jar包，之后，当某个或者某几个子项目需要加载该依赖的时候，就可以在子项目中dependencies节点只配置groupId和artifactId就可以完成依赖的引用。

dependencyManagement主要是为了统一管理依赖包的版本，确保所有子项目使用的版本一致，类似的还有plugins和pluginManagement。

### 4、modules：

子模块列表

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.codehaus.mojo</groupId>

<artifactId>my-parent</artifactId>

<version>2.0</version>

<packaging>pom</packaging>

<modules>

<module>my-project</module>

<module>another-project</module>

<module>third-project/pom-example.xml</module>

</modules>

</project>

**【注】<module>标签里的内容，是子模块的目录名，Maven知道去这些子目录寻找pom.xml文件。**

### 5、properties：

属性列表，定义的属性可以在pom.xml文件中任意位置使用。使用方式为${properties}

<project>

...

<properties>

<maven.compiler.source>1.7<maven.compiler.source>

<maven.compiler.target>1.7<maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

</properties>

...

</project>

## 六、构建配置：

### 1、build：

build可以分为“project build”和“profile build”。

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

...

<!-- "Project Build" contains more elements than just the BaseBuild set -->

<build>...</build>

<profiles>

<profile>

<!-- "Profile Build" contains a subset of "Project Build"s elements -->

<build>...</build>

</profile>

</profiles>

</project>

### 2、基本构建配置：

<build>

<defaultGoal>install</defaultGoal>

<directory>${basedir}/target</directory>

<finalName>${artifactId}-${version}</finalName>

<filters>

<filter>filters/filter1.properties</filter>

</filters>

...

</build>

1. defaultGoal：默认执行目标或阶段。如果给出一个目标，它应该被定义为它在命令行中（如jar：jar）。如定义了一个阶段（如安装），也是如此。
2. directory：构建时的输出路径。默认为：${basedir}/target
3. finalName：这是项目的最终构件名称（不包括文件拓展名，例如：my-project-1.0.jar）
4. filter：定义 \*.properties 文件，其中包含用于接受其设置的资料的属性列表（如下所述）。换句话说，过滤器文件中定义的“name=value”对在代码中替换${name}字符串。

### 3、resource：

资源的配置，资源文件通常不是代码，不需要编译，而是在项目需要捆绑使用的内容

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

<build>

...

<resources>

<resource>

<targetPath>META-INF/plexus</targetPath>

<filtering>false</filtering>

<directory>${basedir}/src/main/plexus</directory>

<includes>

<include>configuration.xml</include>

</includes>

<excludes>

<exclude>\*\*/\*.properties</exclude>

</excludes>

</resource>

</resources>

<testResources>

...

</testResources>

...

</build>

</project>

### 4、extension：

拓展是在此构建中使用的artifacts的列表。它们将被包含在运行构建的classpath中。他们可以启用对构建过程的拓展（例如为wagon传输机制添加一个ftp提供程序），并使活动的插件能够对构建生命周期进行更改。简而言之，拓展是在构建期间激活的artifacts。拓展不需要实际执行任何操作，也不包含Mojo。因此，拓展对于指定普通插件接口的多个实现中的一个是非常好的。

<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

https://maven.apache.org/xsd/maven-4.0.0.xsd">

...

<build>

...

<extensions>

<extension>

<groupId>org.apache.maven.wagon</groupId>

<artifactId>wagon-ftp</artifactId>

<version>1.0-alpha-3</version>

</extension>

</extensions>

...

</build>

</project>

### 5、reporting：

报告包含特定针对site生成阶段的元素。某些Maven插件可以生成reporting元素下配置的报告，例如：生成javadoc报告。reporting与build元素配置插件的功能相似。明显的区别在于：在执行块中插件目标的控制不是细粒度的，报表通过配置reportSet元素来精细控制。而微妙的区别在于reporting元素下 configuration元素可以用作build下的configuration，尽管相反的情况并非如此。

另一个区别就是plugin下的outputDirectory元素。在报告的情况下，默认输出目录为${basedir}/target/site。