一、Maven 依赖

1、依赖元素

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${springframework.version}</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

groupId	必选，实际隶属项目，组织标识（包名）
artifactId	必选，其中的模块，项目名称
version	必选，版本号，项目的当前版本
type	可选，依赖类型，默认 jar
scope	可选，依赖范围，默认 compile
optional	可选，标记依赖是否可选，默认 false
exclusion	可选，排除传递依赖性，默认空
packaging	项目的打包方式，最为常见的 jar 和 war 两种

【说明】关于 packaging 的例子

```
<groupId>com.mycompany.app</groupId>
<artifactId>myapp</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

```
<groupId>com.mycompany.app</groupId>
<artifactId>myWebApp</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
```

2、依赖配置

```
    <!--添加依赖配置-->
    <dependencies>
        <!--项目要使用到 junit 的 jar 包，所以在这里添加 junit 的 jar 包的依赖-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.9</version>
            <scope>test</scope>
        </dependency>
        <!--项目要使用到 Hello 的 jar 包，所以在这里添加 Hello 的 jar 包的依赖-->
        <dependency>
            <groupId>me.gacl.maven</groupId>
            <artifactId>Hello</artifactId>
            <version>0.0.1-SNAPSHOT</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
```

3、依赖范围

Maven 在**编译项目主代码**的时候需要使用一套 classpath

Maven 在**编译和执行测试**的时候会使用另外一套 classpath

Maven 在**实际运行项目**的时候又会使用一套 classpath

依赖范围就是用来控制依赖与这三种 classpath(**编译 classpath、测试 classpath、运行 classpath**)的关系

【Maven 的 6 种依赖范围】

a) **test** 范围指的是**测试范围有效，**只对测试 classpath 有效， 在编译和打包时都不会使用这个依赖，典型例子如 junit

b) **compile** 范围指的是**编译范围有效**，其下的 maven 依赖，对于编译，测试，运行 classpath 都有效. 在编译和打包时都会将依赖存储进去

c) **provided** 依赖：在**编译**和**测试的过程有效**，最后生成 war 包时不会加入，诸如：servlet-api，因为 servlet-api，tomcat 等 web 服务器已经存在了，如果再打包会冲突

d) **runtime** 在**运行的时候**依赖，在编译的时候不依赖，与 provided 相对，典型例子如 jdbc

e) **import**：(maven2.0.9 及以上)：导入依赖范围，它不会对三种实际 Classpath 产生影响

f) **system**：系统依赖范围。其和三种 classpath 的关系，与 provided 一样。但是使用此依赖范围必须通过 SystemPath 元素显示地指定依赖文件的路径。由于与本机系统绑定，不依赖 Maven 仓库解析，所以可能会造成建构的不可移植，谨慎使用。如：

```
1.  <dependencies>
2.          <dependency>
3.              <groupId>javax.sql</groupId>
4.              <artifactId>jdbc-stdext</artifactId>
5.              <version>2.0</version>
6.              <scope>system</scope>
7.              <systemPath>${java.home}/lib/rt.jar</systemPath>
8.          </dependency>
9.      <dependencies>
```

【说明】默认的依赖范围是 compile

【注意】A --> L1.0　B --> L2.0　C --> A 、B

==> A B 谁先在 C 的 pom 中声明，C 就引入先声明的 L 的版本

【依赖范围表】

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、依赖的传递性

【理解依赖的传递性】

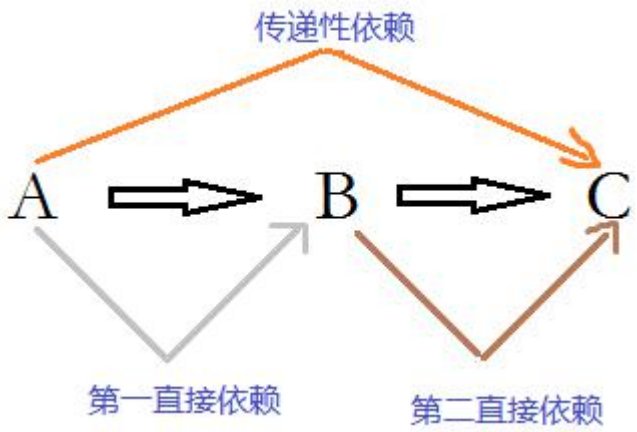
创建了一个Maven Project-----learnDependency，然后我们引入了spring-core这个依赖，然后我们打开spring-core的pom.xml发现，spring-core也有自己的依赖：commons-logging，而且该依赖没有声明依赖范围，那么默认的就是compile，所以这时我们就可以说：commons-logging也是learnDependency的一个依赖，这时我们就将这种依赖称之为传递性依赖，commons-logging是learnDependency的一个传递性依赖。有了传递性依赖，我们就可以在使用的时候不去考虑我们引入的依赖到底是否需要其它依赖,和是否引入多余的依赖,Maven会解析各个直接依赖的pom，将必要的间接依赖引入到项目中。

【关于出错的原因】

简单的说,一般项目中出现问题多数是因为**重复的引用**或者**引用了较低版本的依赖**，或者**是他们的依赖范围发生了变化**。

【细说依赖的传递性】

假设：A 依赖于 B，B 依赖于 C，那么我们就说 A 对于 B 是**第一直接依赖**，B 对于 C 是**第二直接依赖**，A 对于 C 是**传递性依赖**。



因为依赖是有**依赖范围的**，那么对于这种**传递性依赖**，Maven 又是如何**界定其依赖范围**的呢？

a) 当**第二直接依赖**的范围是 compile 的时候，**传递性依赖**的范围与**第一直接依赖**的范围一致；

b) 当**第二直接依赖**的范围是 test 的时候，依赖**不会得以传递**

c) 当**第二直接依赖**的范围是 provided 的时候，**只**传递**第一依赖范围**也为 provided 的依赖，且**传递性依赖**的范围同样是 provided；

d) 当**第二直接依赖**的范围是 runtime 的时候，**传递性依赖**的范围与**第一直接依赖**的范围**一致**，但 compile **除外**，此时**传递性依赖范围**为 runtime

	compile	**test**	**provided**	**runtime**
compile	compile			runtime
test	test			test
provided	provided		provided	provided

runtime	runtime			runtime

【说明】左侧第一列表示第一直接依赖范围，最上面一行表示第二直接依赖

【关于依赖传递性的常见问题】

常见问题一：依赖的重复引入

之前说过 Maven 可以有效的解决依赖的重复引入问题，但是为什么我们在项目还会出现这类问题呢？先让我们来看一下 Maven 是如何处理重复引入问题的：

情景一：我们在项目中分别引入了 2 个依赖 A 和 B，A 又依赖的 C，C 又依赖了 D，B 也依赖了 D，但是这个时候 C 依赖的 D 和 B 依赖的 D 的版本是不同的：

项目 X→A→C→D

项目 X→B→D

也就是说，当前项目引入了 2 次 D 依赖，那么这时，Maven 将采用第一原则：

路径最近原则

情景二：我们在项目中分别引入了 2 个依赖 A 和 B，而 A 和 B 又都引入了 C，但是，此时 A 依赖的 C 和 B 依赖的 C 版本是不一致的，那么这个时候 Maven 如何处理呢？

项目 X→A→C(V1.0)

项目 X→B→C(V2.0)

这时，第一原则已经不起作用了，在 Maven2.0.8 及之前的版本中和 Maven2.0.9 之后的版本 Maven 对于这种情况的处理方式是不一致的，确切的说在 Maven2.0.8 及之前的版本中 Maven 究竟会解析哪个版本的依赖，这是不确定的,在 Maven2.0.9 之后的版本中，制定了第二原则：第一声明者优先

就是说，它取决于在 POM 中依赖声明的顺序。

这个问题就说明了，为什么我们常常遇到的可以正常运行的项目，然后我们增加了一个看似无关的依赖，然后项目就出现了错误，就是这个传递性依赖搞的鬼。

【补充】可选依赖

为什么会有可选依赖呢？是因为某一个项目实现了多个特性，但是我们在面向对象的设计中，有一个原则叫：单一职责性原则，就是强调在一个类只有一项职责，而不是糅合了太多的功能，所以一般这种可选依赖很少会出现。

常见问题二：默认引入的依赖（第二直接依赖的版本过低或者依赖了不稳定的快照）

这个问题我们在开发中也经常遇到，在某个第二直接依赖中引入了 1.0 版本，但是我们现在想使用 2.0 版本，这时我们要如何解决？引入一个名词：**排除依赖**，也可以叫替换依赖。想实现依赖排除，然后替换成自己想要的依赖，这时我们要用到的一个配置是**<exclusions>**和**<exclusion>**，我们可以使用这一元素**声明排除依赖**，然后**显示的声明我们想要的依赖**，在<exclusions>中可以声明一个或多个<exclusion>来排除一个或多个传递性依赖。

【注】声明<exclusion>的时候只需要声明 groupId 和 artifactId 就能唯一定位依赖图中的某个依赖。

A -------> B ------×----C(version1.0)

|

|

C(version2.0)

【关于排除依赖的补充】

由于 maven2.x 会传递解析依赖，所以很有可能一些你不需要的依赖也会包含在工程类路径中。例如，一些你依赖的工程可能没有正确的声明它们的依赖集。为了解决这种特殊情况，maven2.x 已经引入了显式排除依赖的概念。排除是设置在 pom 中指定的依赖上，并且由指定的 groupId 和 artifactId 来标注。当你构建工程时，该物件不会像解析加载依赖一样被加载到你工程的类路径中。

1. 如何使用排除依赖

我们在 pom 中的<dependency>段下面加上<exclusions>标签。

```
<project>
  ...
  <dependencies>
   <dependency>
    <groupId>sample.ProjectA</groupId>
    <artifactId>Project-A</artifactId>
    <version>1.0</version>
    <scope>compile</scope>
    <exclusions>
     <exclusion> <!-- declare the exclusion here -->
      <groupId>sample.ProjectB</groupId>
      <artifactId>Project-B</artifactId>
     </exclusion>
    </exclusions>
   </dependency>
  </dependencies>
</project>
```

2.排除依赖是如何工作的并且在何时使用它（作为最后一种解决方法）

Project-A

-> Project-B

-> Project-D <! -- This dependency should be excluded -->

-> Project-E

-> Project-F

-> Project C

如上图所示，Project-A 依赖 Project-B 和 C，Project-B 依赖 Project-D，Project-D 依赖 Project-E 和 F，默认的 Project-A 的类路径会包含：

B, C, D, E, F

如果由于我们知道 Project-D 的某些依赖在仓库中丢失,那么我们不想 Project-D 和它所有的依赖加载到 Project-A 的类路径中，而且也不想/不需要依赖 Project-D 的 Project-B 的功能。在这种情况下，Project-B 的开发者会提供一个 Project-D 的<optional>true</optional>依赖，如下：

```
<dependency>
  <groupId>sample.ProjectD</groupId>
  <artifactId>ProjectD</artifactId>
  <version>1.0-SNAPSHOT</version>
  <optional>true</optional>
</dependency>
```

然而，并不能达到你想要的效果。作为最后一种解决方法你仍然可以选择将它在 Project-A 中排除掉。如下：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.ProjectA</groupId>
  <artifactId>Project-A</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
  <dependencies>
   <dependency>
    <groupId>sample.ProjectB</groupId>
```

```
        <artifactId>Project-B</artifactId>
        <version>1.0-SNAPSHOT</version>
        <exclusions>
         <exclusion>
          <!-- Exclude Project-D from Project-B -->
          <groupId>sample.ProjectD</groupId>
          <artifactId>Project-D</artifactId>
         </exclusion>
        </exclusions>
       </dependency>
      </dependencies>
    </project>
```

如果我们将 Project-A 部署到一个仓库中，而且 Project-X 声明了一个普通依赖到 Project-A。那么 Project-D 是不是依旧从类路径中排除了？

Project-X -> Project-A

答案是 yes。Project-A 已经声明了它不需要 Project-D，所以它不会作为 Project-A 的传递依赖而引入。那么考虑下图的 Project-X 依赖 Project-Y，

```
Project-X -> Project-Y
                -> Project-B
                    -> Project-D
             ...
```

Project-Y 依赖于 Project-B，并且它有需要 Project-D 所支持的功能，所以不能在 Project-D 的依赖列表中声明排除。也可再提供一个额外的我们可以解析 Project-E 的仓库。在这种情况下，**不能将 Project-D 全局排除**，因为它是 Project-Y 的合法依赖。

在另外一种场景中，如果我们不需要的依赖是 Project-E 而不是 Project-D，我们如何排除它呢？如下图：

```
Project-A
  -> Project-B
    -> Project-D
      -> Project-E <!-- Exclude this dependency -->
```

-> Project-F

-> Project C

排除会影响到在依赖图上所有它的声明点以后的部分。如果你想排除 Project-E 而不是 Project-D，可以简单的将排除指向 Project-E。但是你无法将排除作用到 Project-D，因为你无法改变 Project-D 的 pom，如果你想这样，你应该用选择依赖而不是排除，或者将 Project-D 分成多个子工程，每个只有普通依赖。

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>sample.ProjectA</groupId>
  <artifactId>Project-A</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
  <dependencies>
   <dependency>
    <groupId>sample.ProjectB</groupId>
    <artifactId>Project-B</artifactId>
    <version>1.0-SNAPSHOT</version>
    <exclusions>
     <exclusion>
      <!-- Exclude Project-E from Project-B -->
      <groupId>sample.ProjectE</groupId>
      <artifactId>Project-E</artifactId>
     </exclusion>
    </exclusions>
   </dependency>
  </dependencie>
```

常见问题三：解决重复的配置

我们在开发中也经常遇到这样的情况，比如在使用 spring framework 的时候，他们都是来自于同一个项目的不同模块，因此这些依赖的版本都是相同的，而且在将来升级的时候，这些版本也会一起被升级，这时 Maven 又提供了一种解决方案------使用 properties 元素定义 Maven 属性，然后引用。

示例：

[html] view plain copy

```
1.  <properties>
2.      <springframework.version>2.5.6</springframework.version>
3.  </properties>
```

这个时候我们就可以在声明依赖的时候使用**${springframework.version}**来替换具体的版本号

[html] view plain copy

```
1.  <dependency>
2.      <groupId>org.springframework</groupId>
3.      <artifactId>spring-context-support</artifactId>
4.      <version>${springframework.version}</version>
5.  </dependency>
```

5、聚合和继承

1. 聚合

如果我们想**一次构建多个项目模块**，那我们就需要对多个项目模块进行聚合

1）聚合配置代码

```
<modules>
        <module>模块一</module>
        <module>模块二</module>
        <module>模块三</module>
```

```
5 </modules>
```

例如：对项目的 Hello、HelloFriend、MakeFriends 这三个模块进行聚合

```
1 <modules>
2       <module>../Hello</module>
3       <module>../HelloFriend</module>
4       <module>../MakeFriends</module>
5 </modules>
```

其中 module 的路径**为相对路径**。

再例如：

为了能够使用一条命令就能构建 account-email 和 account-persist 两个模块，我们需要建立一个额外的名为 account-aggregator 的模块，然后通过该模块构建整个项目的所有模块。 account-aggregator 本身也是个 Maven 项目，它的 POM 如下

Xml 代码

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-aggregator</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging> pom </packaging>
    <name>Account Aggregator</name>
     <modules>
        <module>account-email</module>
        <module>account-persist</module>
     </modules>
</project>
```

【注意】**packaging 的类型为 pom ， module 的值是一个以当前 POM 为主目录的相对路径。**

2.继承

继承为了**消除重复**，我们把很多相同的配置提取出来，例如：grouptId，version等

1）继承配置代码

```
<parent>
        <groupId>me.gacl.maven</groupId>
        <artifactId>ParentProject</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../ParentProject/pom.xml</relativePath>
</parent>
```

2）继承代码中定义属性

继承代码过程中，可以定义属性，例如：

```
<properties>
    <project.build.sourceEncoding>
        UTF-8
     </project.build.sourceEncoding>
    <junit.version>4.9</junit.version>
    <maven.version>0.0.1-SNAPSHOT</maven.version>
</properties>
```

访问属性的方式为${junit.version}，例如：

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

3）父模块用 dependencyManagement 进行管理

```
<dependencyManagement>
    <dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
```

```
        <scope>test</scope>
    </dependency>
    <dependency>
            <groupId>cn.itcast.maven</groupId>
            <artifactId>HelloFriend</artifactId>
            <version>${maven.version}</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

这样的好处是子模块可以有选择行的继承，而不需要全部继承。

【例子】

父模块 POM 如下：

Xml 代码

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId> account-parent </artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    <name>Account Parent</name>
</project>
```

子模块声明继承如下：

Xml 代码

```xml
<project>
    <modelVersion>4.0.0</modelVersion>

    < parent >
        <groupId>com.juvenxu.mvnbook.account</groupId>
        <artifactId> account-parent </artifactId>
        <version>1.0.0-SNAPSHOT</version>
        < relativePath >../account-parent/pom.xml</ relativePath>
    </ parent >

    <artifactId> account-email </artifactId>
    <name>Account Email</name>
  ...
</project>
```

最后,同样还需要把 account-parent 加入到聚合模块 account-aggregator 中。聚合的 POM 如下:

Xml 代码

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.juvenxu.mvnbook.account</groupId>
    <artifactId>account-aggregator</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging> pom </packaging>
    <name>Account Aggregator</name>
    <modules>
        <module>account-email</module>
        <module>account-persist</module>
        <module> account-parent</module>
    </modules>
</project>
```

【注意】

a) 子模块没有声明 groupId 和 version,这两个属性继承至父模块。但如果子模块有不同与父模块的 groupId、version，也可指定;

b) 不应该继承 artifactId,如果 groupId，version，artifactId 完全继承的话会造成坐标冲突;另外即使使用不同的 groupId 或 version,同样的 artifactId 也容易产生混淆。

c) 使用继承后 parent 也必须像自模块一样加入到聚合模块中。也就是在在聚合模块的 pom 中加入<module>account-parent</module>

6、聚合与继承的关系

区别 ：

1. 对于聚合模块来说，它知道有哪些被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。

2.对于继承关系的父 POM 来说，它不知道有哪些子模块继承与它，但那些子模块都必须知道自己的父 POM 是什么。

共同点 ：

1.聚合 POM 与继承关系中的父 POM 的 packaging 都是 pom

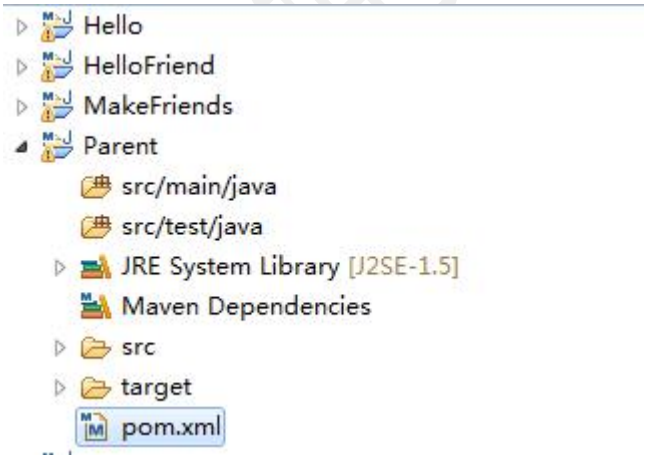
2.聚合模块与继承关系中的父模块除了 POM 之外都没有实际的内容。

聚合关系与继承关系的比较

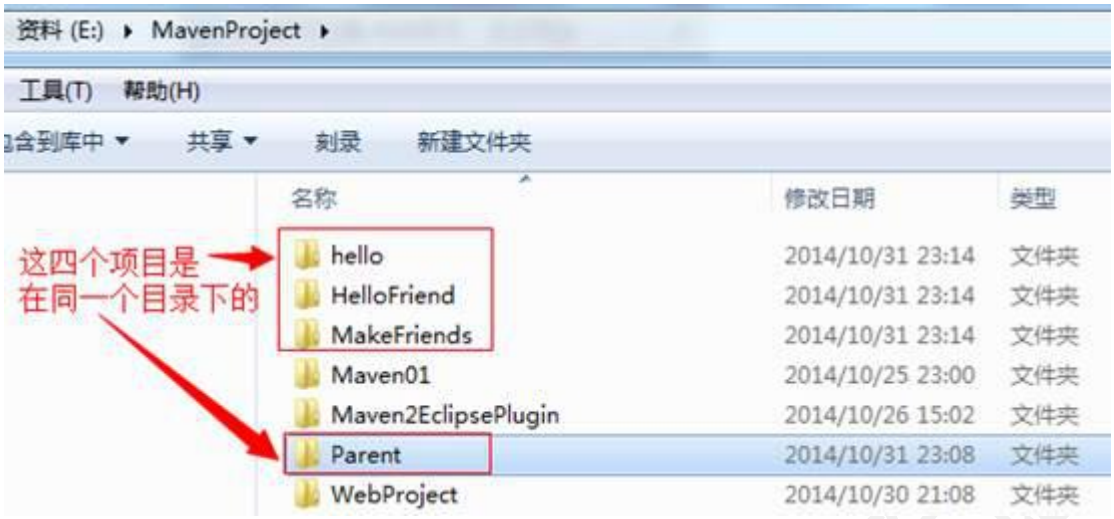
注：在现有的实际项目中一个 POM 既是聚合 POM，又是父 POM，这么做主要是为了方便

7、关于聚合与继承的练习

创建四个 Maven 项目，如下图所示：



这四个项目放在同一个目录下，方便后面进行聚合和继承



Parent 项目是其它三个项目的父项目，主要是用来配置一些公共的配置，其它三个项目再通过继承的方式拥有 Parent 项目中的配置，首先配置 Parent 项目的 pom.xml，添加对项目的 Hello、HelloFriend、MakeFriends 这三个模块进行聚合以及 jar 包依赖，pom.xml 的配置信息如下：

Parent 项目的 pom.xml 配置

```
 1 <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 2     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 3     <modelVersion>4.0.0</modelVersion>
 4
 5     <groupId>me.gacl.maven</groupId>
 6     <artifactId>Parent</artifactId>
 7     <version>0.0.1-SNAPSHOT</version>
 8     <packaging>pom</packaging>
 9
10     <name>Parent</name>
11     <url>http://maven.apache.org</url>
12
13     <!-- 对项目的 Hello、HelloFriend、MakeFriends 这三个模块进行聚合 -->
14     <modules>
15         <module>../Hello</module>
16         <module>../HelloFriend</module>
17         <module>../MakeFriends</module>
18     </modules>
19
20     <!-- 定义属性 -->
```

```
21      <properties>
22          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
23          <junit.version>4.9</junit.version>
24          <maven.version>0.0.1-SNAPSHOT</maven.version>
25      </properties>
26
27      <!-- 用 dependencyManagement 进行 jar 包依赖管理 -->
28      <dependencyManagement>
29          <!-- 配置 jar 包依赖 -->
30          <dependencies>
31              <dependency>
32                  <groupId>junit</groupId>
33                  <artifactId>junit</artifactId>
34                  <!-- 访问 junit.version 属性 -->
35                  <version>${junit.version}</version>
36                  <scope>test</scope>
37              </dependency>
38              <dependency>
39                  <groupId>me.gacl.maven</groupId>
40                  <artifactId>Hello</artifactId>
41                  <!-- 访问 maven.version 属性 -->
42                  <version>${maven.version}</version>
43                  <scope>compile</scope>
44              </dependency>
45              <dependency>
46                  <groupId>me.gacl.maven</groupId>
47                  <artifactId>HelloFriend</artifactId>
48                  <!-- 访问 maven.version 属性 -->
49                  <version>${maven.version}</version>
50              </dependency>
51          </dependencies>
52      </dependencyManagement>
53  </project>
```

在 Hello 项目的 pom.xml 中继承 Parent 项目的 pom.xml 配置

```
 1 <project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 2 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 3
```

```
    <modelVersion>4.0.0</modelVersion>
    <artifactId>Hello</artifactId>

        <!-- 继承 Parent 项目中的 pom.xml 配置 -->
        <parent>
            <groupId>me.gacl.maven</groupId>
            <artifactId>Parent</artifactId>
            <version>0.0.1-SNAPSHOT</version>
            <!-- 使用相对路径 -->
            <relativePath>../Parent/pom.xml</relativePath>
        </parent>

        <dependencies>
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
            </dependency>
        </dependencies>
</project>
```

在 HelloFriend 项目的 pom.xml 中继承 Parent 项目的 pom.xml 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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <artifactId>HelloFriend</artifactId>
    <name>HelloFriend</name>

    <!-- 继承 Parent 项目中的 pom.xml 配置 -->
    <parent>
        <groupId>me.gacl.maven</groupId>
        <artifactId>Parent</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../Parent/pom.xml</relativePath>
    </parent>
    <dependencies>
        <dependency>
            <!—
                Parent 项目的 pom.xml 文件配置中已经指明了要使用的 Junit 的版本号，因此在
```

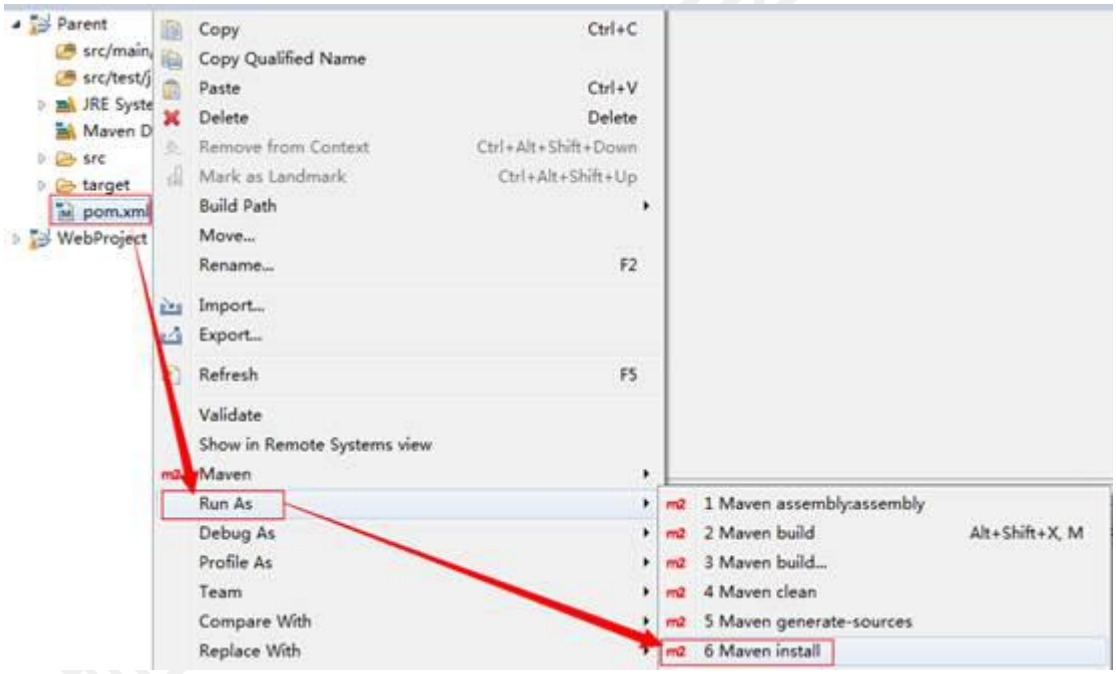
```
                这里添加 junit 的依赖时， 可以不指明<version></version>和
                <scope>test</scope>，会直接从 Parent 项目的 pom.xml 继承
            -->
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
        <!-- HelloFriend 项目中使用到了 Hello 项目中的类，因此需要添加对 Hello.jar 的依赖
        Hello.jar 的<version>和<scope>也已经在 Parent 项目的 pom.xml 文件配置中已经指明了
        因此这里也可以省略不写了
        -->
        <dependency>
            <groupId>me.gacl.maven</groupId>
            <artifactId>Hello</artifactId>
        </dependency>
    </dependencies>
</project>
```

在 MakeFriends 项目的 pom.xml 中继承 Parent 项目的 pom.xml 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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <artifactId>MakeFriends</artifactId>
    <!-- 继承 Parent 项目中的 pom.xml 配置 -->
    <parent>
        <groupId>me.gacl.maven</groupId>
        <artifactId>Parent</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../Parent/pom.xml</relativePath>
    </parent>
    <dependencies>
        <dependency>
        <!—
        Parent 项目的 pom.xml 文件配置中已经指明了要使用的 Junit 的版本
        号，因此在这里添加 junit 的依赖时，可以不指明<version></version>和
        <scope>test</scope>，会直接从 Parent 项目的 pom.xml 继承
        -->
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
        </dependency>
        <dependency>
```

```
20          <!—
            MakeFriends 项目中使用到了 HelloFriend 项目中的类，因此需要添加
            对 HelloFriend.jar 的依赖 HelloFriend.jar 的<version>和<scope>也已经
            在 Parent 项目的 pom.xml 文件配置中已经指明了
22          因此这里也可以省略不写了
23          -->
24              <groupId>me.gacl.maven</groupId>
25              <artifactId>HelloFriend</artifactId>
26          </dependency>
27      </dependencies>
28  </project>
```

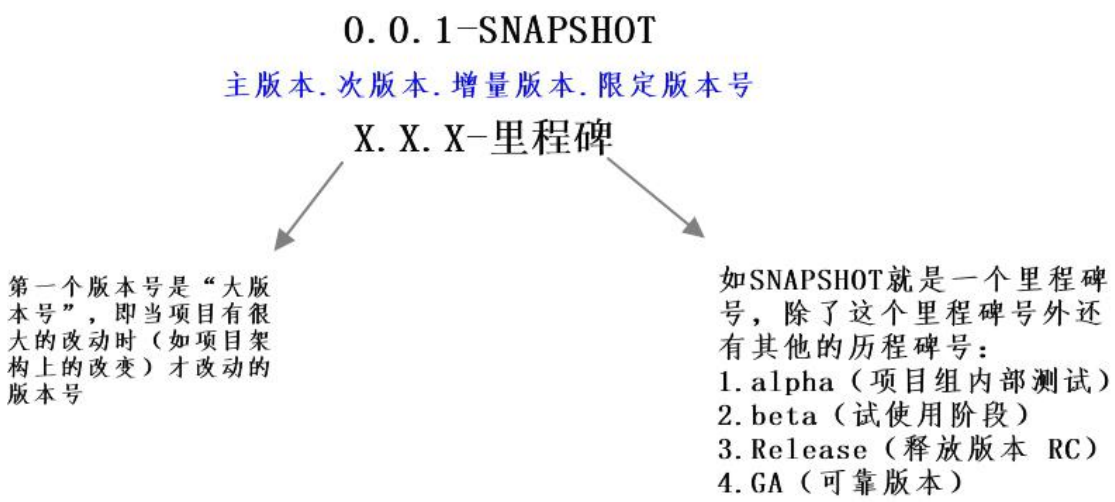
以上的四个项目的 pom.xml 经过这样的配置之后，就完成了在 Parent 项目中聚合 Hello、HelloFriend、MakeFriends 这三个子项目(子模块)，而 Hello、HelloFriend、MakeFriends 这三个子项目(子模块)也继承了 Parent 项目中的公共配置，这样就可以使用 Maven 一次性构建所有的项目了，如下图所示：



选中 Parent 项目的 pom.xml 文件→【Run As】→【Maven install】，这样 Maven 就会一次性同时构建 Parent、Hello、HelloFriend、MakeFriends 这四个项目，如下图所示：

```
[INFO] Parent ..................................... SUCCESS [  0.372 s]
[INFO] Hello ...................................... SUCCESS [  1.935 s]
[INFO] HelloFriend ................................ SUCCESS [  0.473 s]
[INFO] MakeFriends ................................ SUCCESS [  0.459 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

8、Maven 版本控制的一般规则



二、Maven 仓库

1、仓库的概念

在 Maven 中，任何一个依赖、插件或者项目构建的输出，都可以称之为构件。Maven 在某个统一的位置存储所有项目的共享的构件，这个统一的位置，我们就称之为仓库。（仓库就是存放依赖和插件的地方）任何的构件都有唯一的坐标，Maven 根据这个坐标定义了构件在仓库中的唯一存储路径。

2、仓库的分类

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：① 中央仓库 ② 私服 ③其它公共库

3、仓库详解

1.本地仓库

是 Maven 在本地存储构件的地方。Maven 本地仓库的默认位置：无论是 Windows 还是 Linux，在用户的目录下都有一个.m2/repository/的仓库目录，这就是 Maven 仓库的默认位置。

【注】maven 的本地仓库，在安装 maven 后并不会创建，它是在第一次执行 maven 命令的时候才被创建

【更改默认的本地仓库位置】

更改本地仓库有两种方式：① 基于用户范围　② 基于全局范围

1）更改配置用户范围的本地仓库：先在/.m2/目录下创建 settings.xml 文件，然后在~/.m2/settings.xml，设置 localRepository 元素的值为想要的仓库地址

```
<settings>
  <localRepository>D:\maven_new_repository</localRepository>
</settings>
```

这时候，maven 的本地仓库地址就变成了　D:\maven_new_repository ，注：此时配置的 maven 的本地仓库是属于用户范围的。

2）更改配置全局范围的本地仓库：在 M2_HOME/conf/settings.xml 中更改配置，更改配置的方法同上

【注】此时更改后，所有的用户都会受到影响，而且如果 maven 进行升级，那么所有的配置都会被清除，所以要提前复制和备份 M2_HOME/conf/settings.xml 文件，一般情况下不推荐配置全局的 settings.xml

2.远程仓库

1）中央仓库

中央仓库是默认的远程仓库，maven 在安装的时候，自带的就是中央仓库的配置在 maven 的聚合与继承中我们说过，**所有的 maven 项目都会继承超级 pom**，具体的说，包含了下面配置的 pom 我们就称之为超级 pom

```
<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>
```

中央仓库包含了绝大多数流行的开源 Java 构件，以及源码、作者信息、SCM、信息、许可证信息等。一般来说，简单的 Java 项目依赖的构件都可以在这里下载到。

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。

【私服的特性】

a) 节省自己的外网带宽：减少重复请求造成的外网带宽消耗

b) 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

c) 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

d) 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

e) 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

【当前主流的私服】

1. Apache 的 Archiva

2. JFrog 的 Artifactory

3. **Sonatype 的 Nexus**

3）远程仓库的配置

配置远程仓库将引入新的配置元素：<repositories>、<repository>

在<repositories>元素下，可以使用<repository>子元素声明**一个或者多个**远程仓库。

例子：

```
<repositories>
    <repository>
      <id>jboss</id>
      <name>JBoss Repository</name>
      <url>http://repository.jboss.com/maven2/</url>
      <releases>
        <!-- never,always,interval n -->
        <updatePolicy>daily</updatePolicy>
        <enabled>true</enabled>
        <!-- fail,ignore -->
        <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <layout>default</layout>
    </repository>
  </repositories>
```

【新元素】

<updatePolicy>元素：

表示更新的频率，值有：never, always,interval,daily, daily 为默认值

<checksumPolicy>元素：

表示 maven 检查和检验文件的策略，warn 为默认值

出于安全方面的考虑，有时我们要对远程仓库的访问进行认证，一般将认证信息配置在 settings.xml 中：

```
<span style="white-space:pre"></span>
<servers>
   <server>
      <id>same with repository id in pom</id>
      <username>username</username>
      <password>pwd</password>
   </server>
</servers>
```

【注】这里的 id 必须与 POM 中需要认证的 repository 元素的 Id 一致。

4、如何将生成的项目部署到远程仓库

完成这项工作，也需要在 POM 中进行配置，这里有新引入了一个元素：

<distributionManagement>

distributionManagement 包含了 2 个子元素：

repository	发布版本构件的仓库

snapshotRepository	快照版本的仓库

这两个元素都需要配置 id(该远程仓库的唯一标识)，name，url(表示该仓库的地址)，向远程仓库中部署构件，需要进行认证，配置同上。配置正确后运行：

```
mvn clean deploy
```

【说明】正确的看待快照

之前我们在配置 pom 的时候，对于快照的配置都很谨慎，或者说很少用快照的版本，原因是它还很不稳定，极容易给我们的系统带来未知的错误，让我们很难查找。其实快照版本也并不是一无是处，快照最大的用途是用在开发的过程中，尤其是有模块依赖的时候，比如说 AB 两个模块同时开发，A 依赖于 B，开发过程中 AB 都是持续集成的开发，不断的修改 POM 文件和构建工程，这时候版本同步就成了一个很大的问题。使用快照就可以达到这一目的。

其实在快照版本在发布的过程中，Maven 会自动为构件以当前时间戳做标记，有了这个时间戳，我们就可以随时找到最新的快照版本，这样也就解决刚才说的协作开发的问题。

至于 A 如何检查 B 的更新，刚刚在讲配置的时候说过，快照配置中有一个元素可以控制检查更新的频率------updatePolicy

我们也可以使用命令行加参数的形式强制执行让 maven 检查更新：

```
mvn clean install-U
```


5、maven 到底是如何从仓库中解析构件的呢？

【maven 从仓库解析依赖的机制】

a) 当依赖的范围是 system 的时候，Maven 直接从本地文件系统解析构件

b) 根据依赖坐标计算仓库路径后，尝试直接从本地仓库寻找构件，如果发现相应构件，则解析成功

c) 在本地仓库不存在相应的构件情况下，如果依赖的版本是显示的发布版本构件，则遍历所有的远程仓库，发现后下载使用

d) 如果依赖的版本是 RELEASE 或 LATEST，则基于更新策略读取所有远程仓库的元数据，将其于本地仓库的对应元数据合并后，计算出 RELEASE 或者 LATEST 的真实值，然后基于这个真实值检查本地仓库

e) 如果依赖的版本是 SNAPSHOT，则基于更新策略读取所有远程仓库的元数据，将其与本地仓库的对应元数据合并后，得到最新快照版本的值，然后基于该值检查本地仓库或从远程仓库下载

f) 如果最后解析到的构件版本是时间戳格式的快照，则复制其时间戳格式的文件 至 非时间戳格式，并使用该非时间戳格式的构件

【注】一定要记得<release> <enabled>&<snapshot><enabled> ，对于快照也是一样。

在 POM 的依赖声明的时候不推荐使用 LATEST & RELEASE，在 Maven3 中也不再支持在插件配置中使用 LATEST & RELEASE，如果不设置插件版本，那么最终版本和 release 一样，maven 只会解析最新的发布版本构建。

三、Maven 生命周期

一个生命周期由若干个生命周期阶段**组成**，每个生命周期阶段绑定注册若干个**目标**，可人为注册目标到指定处。

1、Maven 三大生命周期

Maven 有三套相互独立的生命周期，请注意这里说的是"三套"，而且"相互独立"，这三套生命周期分别是：

Clean Lifecycle	在进行真正的构建之前进行一些清理工作。
Default Lifecycle	构建的核心部分，编译，测试，打包，部署等等。
Site Lifecycle	生成项目报告，站点，发布站点。

再次强调一下**它们是相互独立的**，你可以仅仅调用 clean 来清理工作目录，仅仅调用 site 来生成站点。当然你也可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

1.clean 生命周期

clean 生命周期每套生命周期都由**一组阶段(Phase)**组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。比如，运行 mvn clean ，这个的 clean 是 Clean 生命周期的一个阶段。有 Clean 生命周期，也有 clean 阶段。Clean 生命周期一共包含了三个阶段：

pre-clean	执行一些需要在 clean 之前完成的工作
clean	移除所有上一次构建生成的文件
post-clean	执行一些需要在 clean 之后立刻完成的工作

"mvn clean" 中的 clean 就是上面的 clean，在一个生命周期中，运行某个阶段的时候，它之前的所有阶段都会被运行，也就是说，"mvn clean"**等同于** mvn pre-clean clean ，如果我们运行 mvn post-clean ，那么 pre-clean，clean 都会被运行。这是 Maven 很重要的一个规则，可以大大简化命令行的输入。

2.site 生命周期

site 生命周期 pre-site 执行一些需要在生成站点文档之前完成的工作

site	生成项目的站点文档
post-site	执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
site-deploy	将生成的站点文档部署到特定的服务器上

这里经常用到的是 site 阶段和 site-deploy 阶段，用以生成和发布 Maven 站点，这可是 Maven 相当强大的功能，Manager 比较喜欢，文档及统计数据自动生成，很好看。

3.default(构建)生命周期

Default 生命周期是 Maven 生命周期中最重要的一个，绝大部分工作都发生在这个生命周期中。这里，只解释一些比较重要和常用的阶段：

1	validate	验证项目一些必要的信息
2	generate-sources	生成编译过程需要的源代码
3	process-sources	处理源代码
4	generate-resources	生成打包需要的资源文件
5	**process-resources**	复制并处理资源文件，至目标目录，准备打包。
6	**compile**	编译项目的源代码。
7	process-classes	后处理字节码文件
8	generate-test-sources	
9	process-test-sources	
10	generate-test-resources	
11	**process-test-resources**	复制并处理资源文件，至目标测试目录。
12	**test-compile**	编译测试源代码。
13	process-test-classes	
14	**test**	使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
15	prepare-package	打包前准备操作
16	**package**	接受编译好的代码，打包成可发布的格式，如 JAR 。
17	pre-integration-test	集成测试运行前操作
18	integration-test	集成测试
19	post-integration-test	集成测试后操作
20	verify	检查
21	**install**	将包安装至本地仓库，以让其它项目依赖。
22	**deploy**	将最终的包复制到远程的仓库，以让其它开发人员与项目共享。

运行任何一个阶段的时候，它前面的所有阶段都会被运行，这也就是为什么我们运行 mvn install 的时候，代码会被编译，测试，打包。此外，Maven 的插件机制是完全依赖 Maven 的生命周期的，因此理解生命周期至关重要。

2、不同打包方式下的生命周期

1.jar packaging

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

2.pom packaging

生成的构件只是它本身，默认打包目标有：

生命周期阶段	目标
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

3.war packaging

生命周期阶段	目标
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

4.其他打包类型

maven plugin, ear, ejb, swf, swc, nar 等，可自定义打包类型。自定义打包类型需要配置定义了定制打包类型生命周期的插件，如为 Adobe Flex (SWF) 定制打包类型：

```
<project>
  ...
   <packaging>swf</packaging>
    ...
     <build>
       <plugins>
           <plugin>
              <groupId>net.israfil.mojo</groupId>
               <artifactId>maven-flex2-plugin</artifactId>
               <version>1.4-SNAPSHOT</version>
               <extensions>true</extensions>
               <configuration>
                 <debug>true</debug>
                 <flexHome>${flex.home}</flexHome>
                 <useNetwork>true</useNetwork>
                 <main>org/sonatype/mavenbook/Main.mxml</main>
```

```
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

3、通用生命周期目标

1.resources:resources

默认绑定到 process-resources 阶段，默认行为是将 src/main/resources 下的资源文件复制到 target/classes 下，且会应用资源过滤器。

资源过滤：即资源变量替换。可以在 xml 资源文件中使用${}引用 properties 配置文件属性值，要想正常使用，还要在 POM 文件增加资源过滤配置（属性替换）。

```
<build>
    <filters>
        <!-- 属性配置文件 -->
        <filter>src/main/filters/default.properties</filter>
    </filters>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <!-- 过滤开关，只有打开才会替换变量 -->
            <filtering>true</filtering>
        </resource>
    </resources>
</build>
```

```
<!--
从这里可以看出，可以配置额外的资源目录，如 ：
<resource>
    <directory>src/main/xml</directory>
    <includes>
        <include>文件名</include>
    </includes>
    <targetPath>目标路径，默认为 target/classes</targetPath>
</resource>
-->
```

2.compile:compile

默认绑定到 compile 阶段，默认行为是编译所有源代码并在构建输出目录存放编译后文件。默认源目录：src/main/java，默认目标目录：target/classes，默认 source 为 1.3，默认 target 为 1.1。

```
<build>
      ...
      <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
      </plugins>
      ...
</build>
```

3.process-test-resources

处理单元测试资源。

4.test-compile

单元测试编译阶段。目标：compile:testCompile，默认输出目录以及自定义方式类同 compile 阶段。

5.Test

大部分生命周期绑定 Surefire 插件的 test 目标至 test 阶段。默认情况下单元测试失败后停止构建，我们可以覆盖这种行为，让构建继续进行（或者直接跳过测试阶段 -Dmaven.test.skip=true）：

```
<build>
    <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <configuration>
            <testFailureIgnore>true</testFailureIgnore>
         </configuration>
       </plugin>
      ...
     </plugins>
</build>
```

6.Install

Install 插件的 install 目标基本上都是绑定到 install 生命周期阶段。install:install 目标将项目的主要构件**安装到本地仓库**。

7.Deploy

Deploy 插件的 deploy 目标通常绑定到 deploy 生命周期阶段。该阶段将一个构件部署到远程 Maven 仓库，更新一些可能被此次部署影响的仓库信息。这里详细暂不讨论。

四、Maven 插件

Maven 是一个实际执行的插件框架，每一个任务实际上是由插件完成的。Maven 的插件通常用于：

① 创建 jar 文件
② 创建 war 文件
③ 编译代码文件
④ 代码进行单元测试
⑤ 创建项目文档
⑥ 创建项目报告

【说明】一个插件通常提供了一组目标，可使用以下语法来执行：

```
mvn [plugin-name]:[goal-name]
```

例如，一个 Java 项目可以与 Maven 的编译器插件的编译目标，通过运行以下命令编译

```
mvn compiler:compile
```

插件类型

Maven 提供以下两种类型的插件：

类型	描述
构建插件	他们在生成过程中执行，并应在 pom.xml 中的<build/>元素进行配置
报告插件	他们的网站生成期间执行的，他们应该在 pom.xml 中的<reporting/>元素进行配置

以下是一些常见的插件列表：

插件	描述

clean	Clean up target after the build. Deletes the target directory.
compiler	Compiles Java source files.
surefile	Run the JUnit unit tests. Creates test reports.
jar	Builds a JAR file from the current project.
war	Builds a WAR file from the current project.
javadoc	Generates Javadoc for the project.
antrun	Runs a set of ant tasks from any phase mentioned of the build.

【例子】我们使用 maven-antrun-plugin 插件在我们的例子打印在控制台上的数据。让我们来了解它在一个更好的方式，让我们创建一个 pom.xml 在 C:MVNproject 文件夹。

```
<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>
  <groupId>com.companyname.projectgroup</groupId>
  <artifactId>project</artifactId>
  <version>1.0</version>
  <build>
    <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.1</version>
            <executions>
                  <execution>
                        <id>id.clean</id>
                        <phase>clean</phase>
                        <goals>
                          <goal>run</goal>
                        </goals>
                        <configuration>
                          <tasks>
                           <echo>clean phase</echo>
                          </tasks>
                        </configuration>
                  </execution>
            </executions>
```

```
        </plugin>
    </plugins>
  </build>
</project>
```

上面的例子说明了以下关键概念：

- 插件使用的插件元素中指定的 pom.xml。
- 每个插件可以有多个目标。
- 你可以从那里插件应该使用它的相位元素开始它的处理定义阶段。我们已经使用 clean 阶段。
- 您可以通过将它们绑定到插件的目标来执行配置任务。我们已经绑定 echo 任务与 maven-antrun-plugin 的运行目标。
- 就这样，Maven 将处理其余部分。它会下载这个插件，如果没有可用的本地存储库并开始处理

五、其他

1、maven 常用的配置变量

1.自定义变量

在创建 Maven 工程后，插件配置中通常会用到一些 Maven 变量，因此需要找个地方对这些变量进行统一定义，下面介绍**如何定义自定义变量**。

在根节点 **project** 下增加 **properties** 节点，所有自定义变量均可以定义在此节点内，如下所示：

```
<!--  全局属性配置  -->
<properties>
  <project.build.name>tools</project.build.name>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

2.maven 内置变量

${basedir}	项目根目录
${project.build.directory}	构建目录，缺省为 target
${project.build.outputDirectory}	构建过程输出目录，缺省为 target/classes
${project.build.finalName}	产出物名称，缺省为 ${project.artifactId}-${project.version}
${project.packaging}	打包类型，缺省为 jar
${project.xxx}	当前 pom 文件的任意节点的内容

2、Maven 的目录结构

好的目录结构可以使开发人员更容易理解项目，为以后的维护工作也打下良好的基础。Maven2 根据业界公认的最佳目录结构，为开发者提供了缺省的标准目录模板。Maven2 的标准目录结构如下：

项目根/	
pom.xml	Maven2 的 pom.xml 文件
src/	
main/	项目主体目录根
java	源代码目录
resources	所需资源目录
filters	资源过滤文件目录
assembly	Assembly descriptors
config	配置文件目录根
test/	项目测试目录根
java	测试代码目录
resources	测试所需资源目录
filters	测试资源过滤文件目录
site	与 site 相关的资源目录
target/	输出目录根
classes	项目主体输出目录
test-classes	项目测试输出目录
site	项目 site 输出目录

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/scripts	Application/Library scripts
src/main/webapp	Web application sources（Web 应用 根目录）
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions required by libraries that the project depends on
README.txt	Project's readme