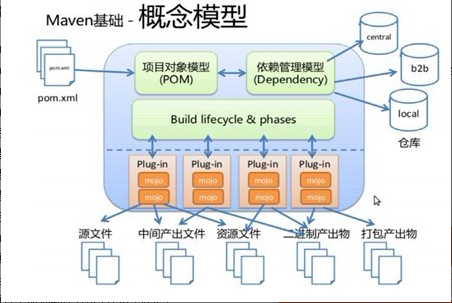
# 什么是Maven

maven是一个项目构建和管理的工具，提供了帮助管理构建、文档、报告、依赖、scm、发布、分发的方法。可以方便的编译代码、进行依赖管理、项目发布等。

maven的好处在于可以将项目过程规范化、自动化、高效化以及强大的可扩展性，利用maven自身及其插件还可以获得代码检查报告、单元测试覆盖率、实现持续集成等等。

Maven的基本原理很简单，采用远程仓库和本地仓库以及一个核心的配置文件pom.xml，pom.xml中定义的jar文件从远程仓库下载到本地仓库，各个应用使用同一个本地仓库的jar，同一个版本的jar只需下载一次，而且避免每个应用都去拷贝jar。同时它采用了现在流行的插件体系架构，所以maven的核心非常的小，只有几兆大小的文件，在执行maven任务时，才会自动下载需要的插件。具体概念模型如下图：



Maven官方把maven定义为一个项目管理工具，下面我们来看看maven给我们的项目管理做了哪些工作？

* **项目标准化**

**Maven项目具有统一的项目结构，**这个项目结构是参考业界的最佳实践而成，为后面使用统一的maven命令打下了基础，如测试mvn test、打包mvn package等，无需写一行脚本，就可以方便的实现众多功能。

* **文档和报告**

使用mvn site可以快速生成项目站点，apache很多开源项目站点都采用maven生成，会出现built by maven字样的图标。

* **类库管理**

**类库管理是maven一个比较核心的功能，我们就需要将项目所依赖的类库加入到pom.xml中，那么maven会自动将依赖的类库下载到本地，并且下载的类库如果还依赖其他的类库，它也会自动下载过来，这样我们就不需要一个一个类库去下载了。**

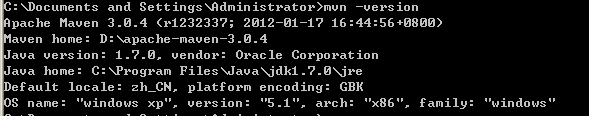
* **发布管理**

使用maven可以方便的进行项目发表管理。在项目开发到一定阶段，可以使用mvn package打包，它会自动先运行mvn test，跑所有的testcase，只有全部通过才能正确打包。生成的war包（如果项目的packaging为war）在target目录下。

# Maven安装及介绍

## 安装

* 从<http://maven.apache.org/download.html> 下载maven的安装包。
* 将下载的安装包解压到特定的目录下，设置环境变量中的path和M2\_HOME，以便在命令行中可以直接使用。如果在命令行中输入：mvn –version,出现下面这个界面，说明我们的maven已经安装成功。



## 配置本地仓库

对于仓库是什么概念，我们这里稍微说明一下，maven中的仓库就是放置各个jar和插件的地方，如果配置的是本地仓库，那么我们所引用的jar都是从本地来的。下面我们来看一个文件，文件放在maven的安装目录下面的conf子目录下面，大家打开这个文件夹可以看到有个叫settings.xml的文件夹，这个里面包含仓库地址、镜像、插件、代理等配置，也是maven中一个核心配置文件，在后面我们会经常跟她打交道。下面我来看看这个文件以及如何配置本地仓库。

* 打开这个文件，我们会看到Default: ~/.m2/repository这样一句话，这句话就告诉我们maven的默认仓库位置，默认是在c盘目录下面，我电脑的位置是在C:\Documents and Settings\Administrator\.m2这个目录下面，而c盘往往是系统盘，放在此目录下面的文件不太安全，所以我们需要更改一下默认的仓库位置，我电脑是放在了D:/java/maven/repository。
* 大家可以在自己的电脑上建立一个仓库，修改setting中的仓库目录位置， <localRepository>D:/java/maven/repository</localRepository>。为了保持核心的配置文件不变，我们将settings.xml文件拷贝到这个目录下面。以后只要对这个文件做修改就可以了。

## Maven目录结构说明

Maven总体目录结构如下图：



## Maven命令说明

* mvn clean -->表示运行清理操作（会默认把target文件夹中的数据清理）
* mvn clean compile-->表示先运行清理之后运行编译，会见代码编译到target文件夹中
* mvn clean test-->运行清理和测试
* mvn clean package-->运行清理和打包
* mvn clean install-->运行清理和安装，会将打好的包安装到本地仓库中，以便其他的项目可以调用
* mvn clean deploy-->运行清理和发布

## 生命周期和插件

#### Maven的生命周期

maven把项目的构建划分为不同的生命周期(lifecycle)。粗略一点的话，它这个过程(phase)包括：编译、测试、打包、集成测试、验证、部署。maven中所有的执行动作(goal)都需要指明自己在这个过程中的执行位置，然后maven执行的时候，就依照过程的发展依次调用这些goal进行各种处理。

maven的生命周期是及其灵活，她生命周期的每个阶段是通过插件来实现的，maven也内置了很多插件，所以我们在项目进行编译、测试、打包的过程是没有感觉到。像编译是通过maven-compile-plugin实现的、测试是通过maven-surefire-plugin实现的。

这个也是maven的一个基本调度机制。一般来说，位置稍后的过程都会依赖于之前的过程。当然，maven同样提供了配置文件，可以依照用户要求，跳过某些阶段。

**三种生命周期**

下面列出了default、clean和site生命周期的所有构建阶段，这些阶段按照指定的顺序执行。

* 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。

**clean生命周期**

| 执行阶段 | 描述说明 |
| --- | --- |
| pre-clean | 在实际的项目清理之前执行所需的过程 |
| clean | 删除前一个构建生成的所有文件 |
| post-clean | 执行完成项目清理所需的过程 |

**Default生命周期**

| 执行阶段 | 描述说明 |
| --- | --- |
| validate | 验证项目是正确的，所有必要的信息都是可用的。 |
| initialize | 初始化构建状态，例如设置属性或创建目录。 |
| generate-sources | 生成包含在编译中的任何源代码。 |
| process-sources | 处理源代码，例如过滤任何值。 |
| generate-resources | 生成包含在包中的资源。 |
| process-resources | 将资源复制并处理到目标目录中，准备打包。 |
| compile | 编译项目的源代码。 |
| process-classes | 从编译后生成生成的文件，例如在Java类上执行字节码增强。 |
| generate-test-sources | 生成包含在编译中的任何测试源代码。 |
| process-test-sources | 处理测试源代码，例如过滤任何值。 |
| generate-test-resources | 为测试创建资源。 |
| process-test-resources | 将资源复制并处理到测试目标目录中。 |
| test-compile | 将测试源代码编译到测试目标目录 |
| process-test-classes | 从测试编译后post-process生成文件，例如在Java类上执行字节码增强。对于Maven 2.0.5和以上。 |
| test | 使用合适的单元测试框架运行测试。这些测试不应该要求打包或部署代码。 |
| prepare-package | 在实际包装前执行必要的准备工作。这通常会导致包的一个未打包的、经过处理的版本。(Maven 2.1及以上) |
| package | 使用已编译的代码，并将其打包成可部署格式，例如JAR。 |
| pre-integration-test | 执行集成测试之前需要执行的操作。这可能涉及到设置所需的环境等问题。 |
| integration-test | 在需要集成测试的环境中，处理并部署包。 |
| post-integration-test | 执行集成测试后所需要的操作。这可能包括清理环境。 |
| verify | 运行任何检查以验证包是否有效，并满足质量标准。 |
| install | 将该包安装到本地存储库中，作为本地其他项目的依赖项。 |
| deploy | 在集成或发布环境中完成，将最终包复制到远程存储库中，以便与其他开发人员和项目共享。 |

**Site生命周期**

| 执行阶段 | 描述说明 |
| --- | --- |
| pre-site | 在实际的项目站点生成之前执行过程 |
| site | 生成项目的站点文档 |
| post-site | 执行确定站点生成的过程，并为站点部署做好准备 |
| site-deploy | 将生成的站点文档部署到指定的web服务器 |

注意：执行某个生命周期的某个阶段不会影响其它的生命周期！

如果要同时执行多个生命周期的阶段可在命令行输入多个命令，中间以空格隔开，例如： clean package 该命令执行clean生命周期的clean阶段和default生命周期的package阶段。

#### 插件

通过上面的生命周期我们可以了解到，不同的生命周期绑定不同的插件；同时我们知道，下载下来的maven核心的东西不过3-4M，它主要就是通过插件来完成这些工作的，一旦碰到没有的插件，它会跑到相应的地方下载，然后来完成整个过程。那么在我们的项目中如何使用插件呢？

打开<http://maven.apache.org/plugins/index.html>网址，我们可以看到apache下面的很多插件，apache下面的插件是比较正规的，它里面的信息非常详细。下面我们来看看里面有个source的插件的用法。

Source插件是对源码进行打包的一个插件，默认情况下，它会将生成的源码放在工程目录的target下面。

Source插件具有五个目标：

* [source:aggregate](http://maven.apache.org/plugins/maven-source-plugin/aggregate-mojo.html) 将所有模块的代码聚合到一个模块中
* [source:jar](http://maven.apache.org/plugins/maven-source-plugin/jar-mojo.html) is used to bundle the main sources of the project into a jar archive.
* [source:test-jar](http://maven.apache.org/plugins/maven-source-plugin/test-jar-mojo.html) on the other hand, is used to bundle the test sources of the project into a jar archive.
* [source:jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/jar-no-fork-mojo.html) is similar to **jar** but does not fork the build lifecycle.
* [source:test-jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/test-jar-no-fork-mojo.html) is similar to **test-jar** but does not fork the build lifecycle.

在我们的工程中，在后面引入下面这段配置：

<build>

<plugins>

<plugin>

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

<artifactId>maven-source-plugin</artifactId>

<version>2.1.2</version>

</plugin>

</plugins>

</build>

上面这段配置就是对源码进行打包的插件，我们运行[source:jar-no-fork](http://maven.apache.org/plugins/maven-source-plugin/jar-no-fork-mojo.html)，那么在项目的目录底下的target会生成一个类似于user-core-0.0.1-SNAPSHOT-sources.jar这样的文件，即项目的源文件。那么如何将这个插件与特定的生命周期绑定呢？我们来看下面这段配置：



通过这段配置，大家可以用 mvn package 当将项目打包的同时会将源代码进行打包。

Apache里面还有很多有用的插件，大家可以自己去试一下，里面说明很详细，大家只要按着官方文档进行配置，一般情况下是没问题的。

# pom

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

## Pom.xml文件结构

1. Project是所有pom.xml的根元素，并且在里面定义了命名空间和xsd元素
2. ModelVersion 当前pom模型的版本
3. GroupId定义当前maven项目隶属的实际项目
4. ArtifactId定义项目中的某个模块名称
5. Version 定义maven项目当前所处的版本号
6. Packaging定义maven项目的打包方式。
7. Dependencies 元素底下就是加入依赖包的地方，那么我们从哪里查询依赖包呢，可以查询的地方比较多，我给出一个大家用得比较多的仓库，<http://mvnrepository.com>。
8. 每个dependency都是一个依赖包，依赖包也就是在dependency里面定义各个依赖包的坐标，这样maven就会从网上下载依赖包到你本地仓库中，有所不同的是dependency元素包含了一个子元素，这个就是对maven生命周期的一个说明，当然除了上面四个子元素外，还包含几个其他的元素。
9. type说明依赖的类型
10. optional标记依赖是否可选
11. exclusions 用来排斥传递依赖

我们具体来看看这个结构：

<dependency>

<groupId>实际项目</groupId>

<artifactId>模块</artifactId>

<version>版本</version>

<type>依赖类型</type>

<scope>依赖范围</scope>

<optional>依赖是否可选</optional>

<!—主要用于排除传递性依赖-->

<exclusions>

<exclusion>

<groupId>…</groupId>

<artifactId>…</artifactId>

</exclusion>

</exclusions>

</dependency>

Maven是通过groupId、artifactId、version这三类似于坐标的元素来确定唯一性的，因此这三个元素对于每个依赖多是必须的。

## 依赖

何为依赖？比如上面的列子中的mail，它需要跟activation-1.1.jar一起才能实现邮件的发送，我们称mail依赖activation-1.1.jar，在maven中，如果mail包中依赖activation-1.1.jar包。如果在以前我们需要手动的去找activation-1.1.jar这个包，找不到我们还不知哪里报错，在这里我们就看到maven的好处了，它就是一个仓库，它里面有各种各样的包。

依赖具有传递性，比如A->B,B->C,那么A间接的依赖于C，这就是依赖的传递性，其中B为A的第一直接依赖，C为A的第二直接依赖。我们通过下面这个表格来说明第一直接依赖和第二直接依赖之间，其中最左边一栏是第一直接依赖，最上面那一栏为第二直接依赖。中间交叉的是传递性依赖范围。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Compile | Test | Provided | Runtime |
| Compile | Compile |  |  | Runtime |
| Test | Test |  |  | Test |
| Provided | Provided |  | Provided | Provided |
| Runtime | Runtime |  |  | Runtime |

下面我们来思考这样一个问题，如果A->B->C->X(1.0),A->D-X(2.0),即A间接依赖X，我们可以看到有两条路径都依赖X，那么maven将会选择哪个版本的X？maven当中有一套自己的规则，我们来说明一下，maven传递性依赖的一些规则以及如何排除依赖冲突。

Maven里面对于传递性依赖有以下几个规则：

1. 最短路径原则：如果A对于依赖路径中有两个相同的jar包，那么选择路径短的那个包
2. 第一声明优先原则：如果A对于依赖路径中有两个相同的jar包，路径也相同，那么依赖写在前面的优先。
3. 可选依赖不会传递，如A->B,B->C,B->D,A对B直接依赖，B对C和D是可选依赖，那么在A中不会引入C和D

最后我们来说说如何排除依赖，我们在前面已经写过，在这个案例中，greenmail依赖于slf4j，而这个包在我们案例中是没有任何意义的，为此我们需要在greenmail中排除对slf4j的依赖，通过下面这个方式，就可以排除依赖了：

<exclusions>

<exclusion>

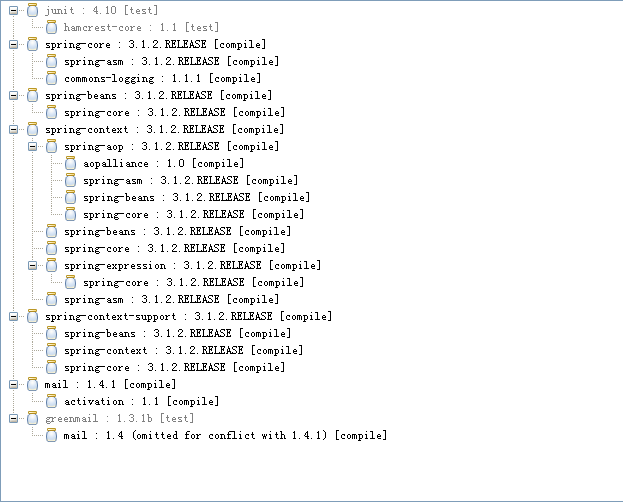
<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

</exclusion>

</exclusions>

对于我们不需要的jar，我们通过这个方式来排除依赖。我们来看看我们这个案例依赖情况图：



基本符合我们上面所讲的规律。

## 依赖范围说明

由于不同的包在不同的地方用到，像junit我们只有在做测试的时候会用到这个包，在我们项目发布的时候，用不到这个包；还有servlet-api，在项目编译的时候将会用到这个包，而项目发布的时候就不会用到这个包，因为一般容器已经自带这个包，如果我们导入，有可能会出现冲突，所以maven引入了依赖范围这个概念，即我们上面提到的scope来解决这个问题。Maven中有主要有以下这几种依赖范围：

1. test范围指的是测试范围有效，在编译和打包时都不会使用这个依赖
2. compile范围指的是编译范围有效，在编译和打包时都会将依赖存储进去
3. 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
4. runtime在运行的时候依赖，在编译的时候不依赖

# 仓库

本地仓库/中央仓库/远程仓库



本地仓库是指存在于我们本机的仓库，在我们加入依赖时候，首先会跑到我们的本地库去找，如果找不到则会跑到远程仓库中去找。对于依赖的包大家可以从这个地址进行搜索：<http://mvnrepository.com/>。

中央仓库这里我解释为供所有本地仓库下载依赖包的一个仓库，它是部署在局域网内的一个仓库。

远程仓库是指在中央仓库找不到的依赖包则会跑到远程仓库去下载的仓库，它需要互联网来实现依赖包的下载。

下面我们来看一个问题：

项目组有甲和乙，甲做了A模块，已做了B模块，而B模块需要用到A模块的内容，那么我们怎样来解决这个问题，我们通常的方法是将A模块提交到svn后，然后乙再将项目同步过来导入到他自己的仓库中。我们看到过程非常复杂，并且甲对A模块做了修改之后，乙需要不断同步A的代码，也是一个非常繁琐的过程。这样我们就需要一个中央仓库来解决这个问题，也就是上图画得中央仓库。

通过上图我们可以看到，在一个公司或单位中，部署一个中央仓库，本地用户先到本地仓库查找相关的依赖包，如果找不到则跑到中央仓库中去找，如果中央仓库找不到，那么将会跑到外网去找。对于我们项目提交，我们会统一提交的中央仓库，这样无论甲还乙我们都可以引用中央仓库中的jar包，这样就可以得到一个比较完美解决方法。 建立中央仓库我们还有另一个好处，可以节省我们的带宽，只要有一个用户加过一个依赖包，那么第二用户就可以直接从中央仓库中下载，而不是从远程仓库下载，这个是在局域网内操作的，速度特别快，特别是对于一些网络不稳定的地方，这样做有极大的好处。

# Maven 项目实例

**新建maven项目**

命令行输入：

mvn archetype:generate -DgroupId=com.pt.test -DartifactId=testHelloworld -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

mvn archetype:generate  固定格式

-DgroupId 组织标识（包名）

-DartifactId 项目名称

-DarchetypeArtifactId    指定ArchetypeId

maven-archetype-quickstart，创建一个Java Project；

maven-archetype-webapp，创建一个Web Project

-DinteractiveMode      是否使用交互模式

**Maven pom文件**

<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.pt.test</groupId>

<artifactId>testHelloworld</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>testHelloworld</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>

<build>

<plugins>

<plugin>

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

<artifactId>maven-site-plugin</artifactId>

<version>3.7.1</version>

</plugin>

<plugin>

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

<artifactId>maven-project-info-reports-plugin</artifactId>

<version>3.0.0</version>

</plugin>

</plugins>

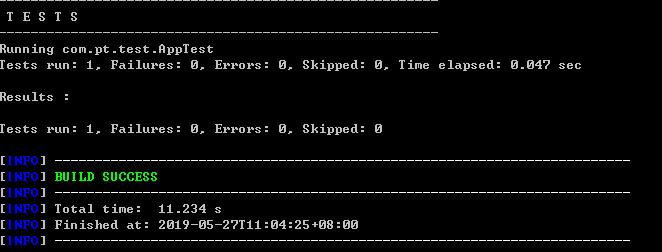
</build>

</project>

# 测试

## 如何进行项目测试

现在的单元测试一般是居于注解的单元测试。我在testHelloworld这个项目中运行mvn test的时候，在控制台输出一下信息：



这个就是测试的一个结果，以上代表运行了1个实例，失败0个，错误0个，忽略0个。

maven怎么知道我们写了哪些测试类呢？这就是约定大于配置的效果，默认情况下，maven会去src/test/java 下面去寻找类，此文件下面的类需要符合以下的规则：

* Test\*.java:任何目录以Test开头的类
* \*Test.java:任何以Test结尾的类
* \*TestCase.java:任何以TestCase结尾的类

## 如何忽略测试

如果在想节省构建时间，忽略全部的测试，我们可以在pom.xml中加入下面这段配置：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.5</version>

<configuration>

<skipTests>true</skipTests>

</configuration>

</plugin>

这样配置后，将会忽略所有的测试，那么如果要忽略某个测试类该怎么做呢？看下面这个命令：

Mvn test –Dtest=UserTest

这样就会忽略UserTest这个类，如果要忽略多个测试，只要在后面加逗号再加对应的测试类即可。在pom.xml中我们也可以配置不参与测试的类，其配置文件如下：

<plugin>

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

<artifactId>maven-surefire-plugin</artifactId>

<version>2.5</version>

<configuration>

<includes>

<include>

\*\*/\*Test.java

</include>

</includes>

<excludes>

<exclude>

\*\*/UserTest.java

</exclude>

</excludes>

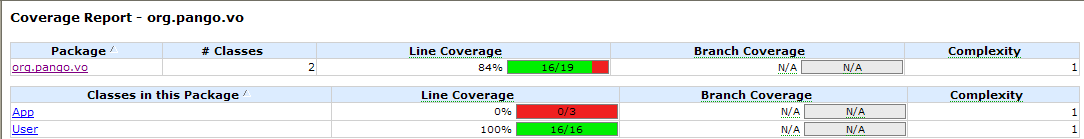
</configuration>

</plugin>

其中includes里面是参与测试的，excludes是不参与测试的。

## 测试报告生成

输入命令cobertura: cobertura,将会在target下生成一个站点测试覆盖率报告，如下图



# 聚合与继承

## 聚合

假设我们这个项目有core模块、dao模块、service模块、web模块，那么如何将我们的项目各个模块进行一次编译、打包、发布即可，而不用我们一个一个模块进行编译、打包、发布呢？这就需要maven中聚合来解决这个问题。这样我们需要构建另外一个模块user\_parent, ，然后通过该模块构建整个项目的所有模块。看下面的pom.xml配置：

<modelVersion>4.0.0</modelVersion>

<groupId>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>

../user-core

</module>

<module>

../user-log

</module>

<module>

../user-dao

</module>

<module>

../user-service

</module>

</modules>

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。**

这样配置后，我们只需要一次性运行user\_parent这个模块，即可将所有模块进行编译、打包、发布。

## 继承

如果多个模块出现相同的依赖包，这样在pom.xml文件的内容出现了冗余、重复的内容，解决这个问题其实使用Maven的继承机制即可，就像Java的继承一样，父类就像一个模板，子类继承自父类，那么有些通用的方法、变量都不必在子类中再重复声明了。Maven的继承机制类似，在一个父级别的Maven的pom文件中定义了相关的常量、依赖、插件等等配置后，实际项目模块可以继承此父项目 的pom文件，重复的项不必显示的再声明一遍了，相当于父Maven项目就是个模板，等着其他子模块去继承。不过父Maven项目要高度抽象，高度提取公 共的部分（交集）。下面我们来看看maven中继承怎么使用。

在user\_parent模块中做如下配置：

<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>

<groupId>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<modules>

<module>

../user-core

</module>

<module>

../user-log

</module>

<module>

../user-dao

</module>

<module>

../user-service

</module>

</modules>

<properties>

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

<junit.version>4.10</junit.version>

</properties>

<dependencies>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-dao</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-log</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-core</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>3.6.10.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.19</version>

</dependency>

<dependency>

<groupId>javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.12.1.GA</version>

</dependency>

<dependency>

<groupId>org.ow2.orchestra.eclipse.birt</groupId>

<artifactId>org.ow2.orchestra.eclipse.birt.chart.engine</artifactId>

<version>3.7.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

写法很简单，就跟普通的依赖是一样的写法，我们把需要的继承的包直接写dependencies中即可。

下面我们来看看如何在其他项目中如何继承这个项目中的jar包，看下面这段配置：

<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>

<parent>

<groupId>org.pango.user</groupId>

<artifactId>user-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<relativePath>../user-parent/pom.xml</relativePath>

</parent>

<artifactId>user-core</artifactId>

<packaging>jar</packaging>

<name>user-core</name>

<url>http://maven.apache.org</url>

</project>

我们就是通过parent这个元素来继承父项目中的包，在parent中定义parent项目的坐标以及父项目中pom的相对位置。

那么有些人就会问，如果每个项目中都继承这个父项目，有些项目中不是出现很多没用的包，确实会存在这个问题，不过不要着急，maven已经为我们提供这样的解决方法。

对于需要选择继承的我们可以把依赖放入<dependencyManagement>，我们看下面的配置：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-dao</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-log</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>user-core</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.4</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>3.6.10.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.19</version>

</dependency>

<dependency>

<groupId>javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.12.1.GA</version>

</dependency>

<dependency>

<groupId>org.ow2.orchestra.eclipse.birt</groupId>

<artifactId>org.ow2.orchestra.eclipse.birt.chart.engine</artifactId>

<version>3.7.0</version>

</dependency>

</dependencies>

</dependencyManagement>

对于选择继承，把继承的配置加入到dependencyManagement中，对于继承的项目，我需要引入依赖的声明，如下：

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

</dependency>

我们不需要进行版本的声明，这样就可以根据自己的需要引入需要的包，而不会继承全部的包。

对于每个项目都需要继承的，我们按下面这种方式加入依赖管理：

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

</dependencies>

这样继承的每个项目，就都会有这个jar包了 。

## ****聚合与继承的关系****

**区别** ：

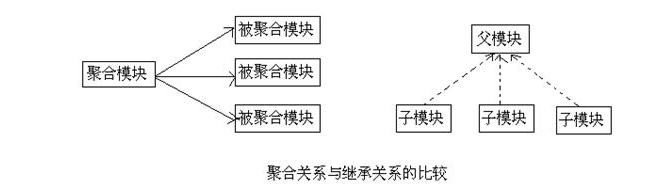
**1**.对于聚合模块来说，它知道有哪些被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。

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。



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

## ****Maven可继承的POM**** ****元素****

groupId ：项目组 ID ，项目坐标的核心元素；

version ：项目版本，项目坐标的核心元素；

description ：项目的描述信息；

organization ：项目的组织信息；

inceptionYear ：项目的创始年份；

url ：项目的 url 地址

develoers ：项目的开发者信息；

contributors ：项目的贡献者信息；

distributionManagerment ：项目的部署信息；

issueManagement ：缺陷跟踪系统信息；

ciManagement ：项目的持续继承信息；

scm ：项目的版本控制信息；

mailingListserv ：项目的邮件列表信息；

properties ：自定义的 Maven 属性；

dependencies ：项目的依赖配置；

dependencyManagement ：醒目的依赖管理配置；

repositories ：项目的仓库配置；

build ：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等；

reporting ：包括项目的报告输出目录配置、报告插件配置等。

## 有关版本说明

对于用过svn或者cvs的朋友们，都会知道，每次修改都会提交一个版本到服务器上，对于我们平常所搭建的项目，大家可能没有注意到版本这个概念。其实版本对于叠加式开发的项目是个很重要的概念，通过上面的依赖，我们就可以清楚地看到一个version，这个就是引入依赖包的版本。

那么版本一共可以分为几个层次，一般来说，版本可以分为：总版本号.分支版本号.小版本号-里程碑版本。

总版本号的变动一般表示框架的变动

分支版本号：一般表示增加了一些功能

小版本号：在分支版本上面进行bug的修复

里程碑：SNAPSHOT-->alpha-->beta-->release-->GA

user0.0.1-SNAPSHOT-->user0.0.1-Release--->user1.0.0SHAPSHOT -->user1.0.0-Rlease

                          -->user0.1.0-SNAPSHOT-->user0.1.0-Rlease