# 1、关键名词

    **Project**：任何您想build的事物，Maven都可以认为它们是工程。这些工程被定义为工程对象模型(POM，Poject Object Model)。一个工程可以依赖其它的工程；一个工程也可以由多个子工程构成。  
    **POM**：POM(pom.xml)是Maven的核心文件，它是指示Maven如何工作的元数据文件，类似于Ant中的build.xml文件。POM文件位于每个工程的根目录中。  
    **GroupId**：groupId是一个工程的在全局中唯一的标识符，一般地，它就是工程名。groupId有利于使用一个完全的包名，将一个工程从其它有类似名称的工程里区别出来。  
    **Artifact**：artifact是工程将要产生或需要使用的文件，它可以是jar文件，源文件，二进制文件，war文件，甚至是pom文件。每个artifact都由groupId和artifactId组合的标识符唯一识别。需要被使用(依赖)的artifact都要放在仓库(见Repository)中，否则Maven无法找到(识别)它们。  
    **Dependency**：为了能够build或运行，一个典型的Java工程会依赖其它的包。在Maven中，这些被依赖的包就被称为dependency。dependency一般是其它工程的artifact。

**Plug-in**：Maven是由插件组织的，它的每一个功能都是由插件提供的。插件提供goal(类似于Ant中的target)，并根据在POM中找到的元数据去完成工作。主要的Maven插件要是由Java写成的，但它也支持用Beanshell或Ant脚本写成的插件。  
    **Repository**：仓库用于存放artifact，它可以是本地仓库，也可以是远程仓库。Maven有一个默认的远程仓库--central。在Windows平台上，本地仓库的默认地址是*User\_Home*\.m2\repository。  
    **Snapshot**：工程中可以(也应该)有一个特殊版本，它的版本号包括*SNAPSHOT*字样。该版本可以告诉Maven，该工程正处于开发阶段，会经常更新(但还未发布)。当其它工程使用此类型版本的artifact时，Maven会在仓库中寻找该artifact的最新版本，并自动下载、使用该最新版。

**Release:** 已发布版本，它的版本号包括RELEASE字样。该版本可以告诉Maven，该工程已发布文档版本，可以被其他工程使用。

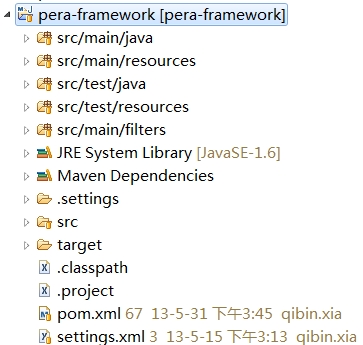
# 2、Maven Build Life Cycle

    软件项目一般都有相似的开发过程：准备，编译，测试，打包和部署，Maven将上述过程称为Build Life Cycle。在Maven中，这些生命周期由一系列的短语组成，每个短语对应着一个(或多个)操作；或对应着一个(或多个)goal(类似于Ant中的target)。  
    如编译源文件的命令mvn compile中的compile是一个生命周期短语。同时该命令也可以等价于mvn compiler:compile，其中的compiler是一个插件，它提供了compile(**此compile与mvn compile中的compile意义不同**)goal；compiler还可提供另一个goal--testCompile，该goal用于编译junit测试类。  
    在执行某一个生命周期时，Maven会首先执行该生命周期之前的其它周期。如要执行compile，那么将首先执行validate，generate-source，process-source和generate-resources，最后再执行compile本身。*Maven中默认的生命周如下图*：



# 3、标准目录布局

    Maven为工程中的源文件，资源文件，配置文件，生成的输出和文档都制定了一个标准的目录结构。Maven鼓励使用标准目录布局，这样就不需要进行额外的配置，而且有助于各个不同工程之间的联接。当然，Maven也允许定制个性的目录布局，这就需要进行更多的配置。*于Maven的标准目录布局如下图：*



# 4、Maven的优点

    [1]build逻辑可以被重用。在Ant中可能需要多次重复地写相同的语句，但由于POM的继承性，可以复用其它的POM文件中的语句。这样既可以写出清晰的build语句，又可以构造出层次关系良好的build工程。  
    [2]不必关注build工作的实现细节。我们只需要使用一些build生命周期短语就可以达到我们的目标，而不必管Maven是如何做到这些的。如，只需要告诉Maven要安装(install)，那么它自然就会验证，编译，打包，及安装。  
    [3]Maven会自动加载工程依赖的artifact所依赖的其它artifact(Transitive Dependency)，而不用显示的将这些artifact全部写到dependency中。  
    [4]如果完全使用Maven的标准目录布局，那么可以极大地减少配置细节。

# 5、项目依赖

**依赖范围**

compile（编译范围）

compile是默认的范围；如果没有提供一个范围，那该依赖的范围就是编译范围。编译范围依赖在所有的classpath中可用，同时它们也会被打包。

provided（已提供范围）

provided依赖只有在当JDK或者一个容器已提供该依赖之后才使用。例如，如果你开发了一个web应用，你可能在编译classpath中需要可用的Servlet API来编译一个servlet，但是你不会想要在打包好的WAR中包含这个Servlet API；这个Servlet API JAR由你的应用服务器或者servlet容器提供。已提供范围的依赖在编译classpath（不是运行时）可用。它们不是传递性的，也不会被打包。

runtime（运行时范围）

runtime依赖在运行和测试系统的时候需要，但在编译的时候不需要。比如，你可能在编译的时候只需要JDBC API JAR，而只有在运行的时候才需要JDBC驱动实现。

test（测试范围）

test范围依赖 在一般的 编译和运行时都不需要，它们只有在测试编译和测试运行阶段可用。

system（系统范围）

system范围依赖与provided类似，但是你必须显式的提供一个对于本地系统中JAR文件的路径。这么做是为了允许基于本地对象编译，而这些对象是系统类库的一部分。这样的构件应该是一直可用的，Maven也不会在仓库中去寻找它。如果你将一个依赖范围设置成系统范围，你必须同时提供一个systemPath元素。注意该范围是不推荐使用的（你应该一直尽量去从公共或定制的Maven仓库中引用依赖）。

**可选依赖**

编译项目的时候你需要两个依赖类库，但是你不希望在使用你类库的项目中，这两个依赖类库同时作为传递性运行时依赖出现。你可以使用如下方式 “声明可选依赖”中的可选依赖来完成这个任务：

164

例 9.4. 声明可选依赖

<project>

<modelVersion>4.0.0</modelVersion>

<**groupId**>org.sonatype.mavenbook<**/groupId**>

<artifactId>my-project</artifactId>

<version>1.0.0</version>

<dependencies>

<dependency>

<**groupId**>net.sf.ehcache<**/groupId**>

<artifactId>ehcache</artifactId>

<version>1.4.1</version>

<optional>true</optional>

</dependency>

<dependency>

<groupId>swarmcache</groupId>

<artifactId>swarmcache</artifactId>

<version>1.0RC2</version>

<optional>true</optional>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.13</version>

</dependency>

</dependencies>

</project>

**依赖版本界限**

你并不是必须为依赖声明某个特定的版本，你可以指定一个满足给定依赖的版本界限。

例如，你可以指定你的项目依赖于JUnit的3.8或以上版本，或者说依赖于JUnit 1.2.10

和1.2.14之间的某个版本。你可以使用如下的字符来围绕一个或多个版本号，来实现版

本界限。

(, )不包含量词

[, ]包含量词

例如，如果你想要访问JUnit任意的大于等于3.8但小于4.0的版本，你的依赖可以“指定一个依赖界限：JUnit 3.8 - JUnit 4.0”

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>[3.8,4.0)</version>

<scope>test</scope>

</dependency>

**传递性依赖**

一个传递性依赖就是对于一个依赖的依赖。如果project-a依赖于project-b，而后者接着依赖于project-c，那么project-c就被认为是project-a的传递性依赖。如果project-c依赖于project-d，那么project-d就也被认为是project-a的传递性依赖。Maven的部分吸引力是由于它能够管理传递性依赖，并且能够帮助开发者屏蔽掉跟踪所有编译期和运行期依赖的细节。你可以只依赖于一些包如Spring Framework，而不用担心Spring Framework的所有依赖，Maven帮你自动管理了，你不用自己去详细了解配置。Maven是怎样完成这件事情的呢？它建立一个依赖图，并且处理一些可能发生的冲突和重叠。例如，如果Maven看到有两个项目依赖于同样的groupId和artifactId，它会自动整理出使用哪个依赖，选择那个最新版本的依赖。虽然这听起来很方便，但在一些边界情况中，传递性依赖会造成一些配置问题。在这种情况下，你可以使用依赖排除。

**冲突解决**

有很多时候你需要排除一个传递性依赖，比如当你依赖于一个项目，后者又继而依赖于

另外一个项目，但你的希望是，要么整个的排除这个传递性依赖，要么用另外一个提供

同样功能的依赖来替代这个传递性依赖。以下“排除一个传递性依赖”展示的例子

中添加了一个对于project-a的依赖，但排除了传递性依赖project-b。

<**dependency**>

<**groupId**>org.sonatype.mavenbook<**/groupId**>

<**artifactId**>project-a<**/artifactId**>

<**version**>1.0<**/version**>

<**exclusions**>

<**exclusion**>

<**groupId**>org.sonatype.mavenbook<**/groupId**>

<**artifactId**>project-b<**/artifactId**>

<**/exclusion**>

<**/exclusions**>

<**/dependency**>

**依赖归类**

如果你需要在项目间共享一组很多的依赖，你也可以建立在项目间建立父子关系，然后将所有共同的依赖重构到父项目中，但是这种父子方式的缺点是一个项目只能有一个父项目。有时候将类似的依赖归类在一起并且使用pom依赖是更明智的做法。因为这样你的项目就可以根据需要引用很多巩固依赖POM。

参见：https://192.168.50.210/svn/ProductDevDepartment/pera

# 6、实例

参见：https://192.168.50.210/svn/ProductDevDepartment/pera-framework

# 7、小结

  通过以上的叙述与实例，应该可以对Maven有一个粗略的认识了。使用Maven关键是要弄清楚如何写pom.xml文件，就如同使用Ant要会写build.xml文件一样。在POM中可以直接写入Ant的task脚本，也可以调用Ant的build.xml文件(推荐)，所以Maven也可以完成Ant的绝大多数工作(但不必安装Ant)。注意：使用Maven就不要再过多的使用Ant脚本。  
  利用好Maven的继承特性及子工程的关系，可以很好地简化POM文件，并能够构建层次结构良好的工程，有利于工程的维护。