**什么是pom?**  
    pom作为项目对象模型。通过xml表示maven项目，使用pom.xml来实现。主要描述了项目：包括配置文件；开发者需要遵循的规则，缺陷管理系统，组织和licenses，项目的url，项目的依赖性，以及其他所有的项目相关因素。

**快速察看：**  
<project>  
  <modelVersion>4.0.0</modelVersion>  
<!--maven2.0必须是这样写，现在是maven2唯一支持的版本-->  
  <!-- 基础设置 -->  
  <groupId>...</groupId>  
  <artifactId>...</artifactId>  
  <version>...</version>  
  <packaging>...</packaging>

  <name>...</name>

  <url>...</url>  
  <dependencies>...</dependencies>  
  <parent>...</parent>  
  <dependencyManagement>...</dependencyManagement>  
  <modules>...</modules>  
  <properties>...</properties>  
  
  <!--构建设置 -->  
  <build>...</build>  
  <reporting>...</reporting>  
  
  <!-- 更多项目信息 -->  
  <name>...</name>  
  <description>...</description>  
  <url>...</url>  
  <inceptionYear>...</inceptionYear>  
  <licenses>...</licenses>  
  <organization>...</organization>  
  <developers>...</developers>  
  <contributors>...</contributors>  
  
  <!-- 环境设置-->  
  <issueManagement>...</issueManagement>  
  <ciManagement>...</ciManagement>  
  <mailingLists>...</mailingLists>   
  <scm>...</scm>  
  <prerequisites>...</prerequisites>  
  <repositories>...</repositories>  
  <pluginRepositories>...</pluginRepositories>  
  <distributionManagement>...</distributionManagement>  
  <profiles>...</profiles>  
</project>

基本内容：

POM包括了所有的项目信息

groupId:项目或者组织的唯一标志，并且配置时生成路径也是由此生成，如org.myproject.mojo生成的相对路径为：/org/myproject/mojo

artifactId:项目的通用名称

version:项目的版本

packaging:打包机制，如pom,jar,maven-plugin,ejb,war,ear,rar,par

name:用户描述项目的名称，无关紧要的东西，可选

url:应该是只是写明开发团队的网站，无关紧要，可选

classifer:分类

其中groupId,artifactId,version,packaging这四项组成了项目的唯一坐标。一般情况下，前面三项就可以组成项目的唯一坐标了。

POM关系：主要为依赖，继承，合成

依赖关系：

<dependencies>  
    <dependency>  
      <groupId>junit</groupId>  
      <artifactId>junit</artifactId>  
      <version>4.0</version>  
      <type>jar</type>  
      <scope>test</scope>  
      <optional>true</optional>  
    </dependency>

    <dependency>

        <groupId>com.alibaba.china.shared</groupId>

        <artifactId>alibaba.apollo.webx</artifactId>

        <version>2.5.0</version>

        <exclusions>

          <exclusion>

            <artifactId>org.slf4j.slf4j-api</artifactId>

            <groupId>com.alibaba.external</groupId>

          </exclusion>

          ....

        </exclusions>

......

</dependencies>

其中groupId, artifactId, version这三个组合标示依赖的具体工程，而且 这个依赖工程必需是maven中心包管理范围内的，如果碰上非开源包，maven支持不了这个包，那么则有有三种 方法处理：

1.本地安装这个插件install plugin

例如：mvn install:intall-file -Dfile=non-maven-proj.jar -DgroupId=som.group -DartifactId=non-maven-proj -Dversion=1

2.创建自己的repositories并且部署这个包，使用类似上面的deploy:deploy-file命令，

3.设置scope为system,并且指定系统路径。

dependency里属性介绍：

type：默认为jar类型，常用的类型有：jar,ejb-client,test-jar...,可设置plugins中的extensions值为true后在增加 新的类型，

scope：是用来指定当前包的依赖范围，[maven的依赖范围](http://blog.csdn.net/zhuxinhua/archive/2010/08/04/5789278.aspx)

optional:设置指依赖是否可选，默认为false,即子项目默认都继承，为true,则子项目必需显示的引入，与dependencyManagement里定义的依赖类似 。

exclusions：如果X需要A,A包含B依赖，那么X可以声明不要B依赖，只要在exclusions中声明exclusion.

exclusion:是将B从依赖树中删除，如上配置，alibaba.apollo.webx不想使用com.alibaba.external  ,但是alibaba.apollo.webx是集成了com.alibaba.external,r所以就需要排除掉.

如果一个工程是parent或者aggregation（即mutil-module的）的，那么必须在packing赋值为pom,child工程从parent继承的包括：dependencies,developers,contributors,plugin lists,reports lists,plugin execution with matching ids,plugin configuration

parent的使用方法如下：

<parent>   
    <groupId>org.codehaus.mojo</groupId>   
    <artifactId>my-parent</artifactId>   
    <version>2.0</version>   
    <relativePath>../my-parent</relativePath>   
  </parent>

relativePath是可选的,maven会首先搜索这个地址,在搜索本地远程repositories之前.

dependencyManagement：是用于帮助管理chidren的dependencies的。例如如果parent使用dependencyManagement定义了一个dependencyon junit:junit4.0,那么 它的children就可以只引用 groupId和artifactId,而version就可以通过parent来设置，这样的好处就是可以集中管理 依赖的详情

modules:对于多模块的project,outer-module没有必需考虑inner-module的dependencies,当列出modules的时候，modules的顺序是不重要的，因为maven会自动根据依赖关系来拓扑排序，

modules例子如下 ：

<module>my-project</module>

<module>other-project</module>

properties:是为pom定义一些常量，在pom中的其它地方可以直接引用。

定义方式如下：

<properties>

      <file.encoding>UTF-8</file\_encoding>

      <[**Java**](http://lib.csdn.net/base/java).source.version>1.5</java\_source\_version>

      <java.target.version>1.5</java\_target\_version>

</properties>

使用方式 如下 ：

${file.encoding}

还可以使用project.xx引用pom里定义的其它属性：如$(project.version}

build设置:

defaultGoal:默认的目标，必须跟命令行上的参数相同，如：jar:jar,或者与时期parse相同,例如install

directory:指定build target目标的目录，默认为$(basedir}/target,即项目根目录下的target

finalName:指定去掉后缀的工程名字，例如：默认为${artifactId}-${version}

filters:用于定义指定filter属性的位置，例如filter元素赋值filters/filter1.properties,那么这个文件里面就可以定义name=value对，这个name=value对的值就可以在工程pom中通过${name}引用，默认的filter目录是${basedir}/src/main/fiters/

resources:描述工程中资源的位置

<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>

targetPath:指定build资源到哪个目录，默认是base directory

filtering:指定是否将filter文件(即上面说的filters里定义的\*.property文件)的变量值在这个resource文件有效,例如上面就指定那些变量值在configuration文件无效。

directory:指定属性文件的目录，build的过程需要找到它，并且将其放到targetPath下，默认的directory是${basedir}/src/main/resources

includes:指定包含文件的patterns,符合样式并且在directory目录下的文件将会包含进project的资源文件。

excludes:指定不包含在内的patterns,如果inclues与excludes有冲突，那么excludes胜利，那些符合冲突的样式的文件是不会包含进来的。

testResources:这个模块包含[**测试**](http://lib.csdn.net/base/softwaretest)资源元素，其内容定义与resources类似，不同的一点是默认的测试资源路径是${basedir}/src/test/resources,测试资源是不部署的。

plugins配置：

<plugin>   
        <groupId>org.apache.maven.plugins</groupId>   
        <artifactId>maven-jar-plugin</artifactId>   
        <version>2.0</version>   
        <extensions>false</extensions>   
        <inherited>true</inherited>   
        <configuration>   
          <classifier>test</classifier>   
        </configuration>   
        <dependencies>...</dependencies>   
        <executions>...</executions>   
      </plugin>

extensions:true or false, 决定是否要load这个plugin的extensions，默认为true.

inherited:是否让子pom继承，ture or false 默认为true.

configuration:通常用于私有不开源的plugin,不能够详细了解plugin的内部工作原理，但使plugin满足的properties

dependencies:与pom基础的dependencies的结构和功能都相同，只是plugin的dependencies用于plugin,而pom的denpendencies用于项目本身。在plugin的dependencies主要用于改变plugin原来的dependencies，例如排除一些用不到的dependency或者修改dependency的版本等，详细请看pom的denpendencies.

executions:plugin也有很多个目标，每个目标具有不同的配置，executions就是设定plugin的目标，

<execution>   
            <id>echodir</id>   
            <goals>   
              <goal>run</goal>   
            </goals>   
            <phase>verify</phase>   
            <inherited>false</inherited>   
            <configuration>   
              <tasks>   
                <echo>Build Dir: ${project.build.directory}</echo>   
              </tasks>   
            </configuration>   
          </execution>

id:标识符

goals:里面列出一系列的goals元素，例如上面的run goal

phase:声明goals执行的时期，例如：verify

inherited:是否传递execution到子pom里。

configuration:设置execution下列表的goals的设置，而不是plugin所有的goals的设置

pluginManagement配置：

pluginManagement的作用类似于denpendencyManagement,只是denpendencyManagement是用于管理项目jar包依赖，pluginManagement是用于管理plugin。与pom build里的plugins区别是，这里的plugin是列出来，然后让子pom来决定是否引用。

例如：

<pluginManagement>

      <plugins>   
        <plugin>   
          <groupId>org.apache.maven.plugins</groupId>   
          <artifactId>maven-jar-plugin</artifactId>   
          <version>2.2</version>   
          <executions>   
            <execution>   
              <id>pre-process-classes</id>   
              <phase>compile</phase>   
              <goals>   
                <goal>jar</goal>   
              </goals>   
              <configuration>   
                <classifier>pre-process</classifier>   
              </configuration>   
            </execution>   
          </executions>   
        </plugin>   
      </plugins>   
    </pluginManagement>   
子pom引用方法：   
在pom的build里的plugins引用：   
    <plugins>   
      <plugin>   
        <groupId>org.apache.maven.plugins</groupId>   
        <artifactId>maven-jar-plugin</artifactId>   
      </plugin>   
    </plugins>

build里的directories:

<sourceDirectory>${basedir}/src/main/java</sourceDirectory>   
    <scriptSourceDirectory>${basedir}/src/main/scripts</scriptSourceDirectory>   
    <testSourceDirectory>${basedir}/src/test/java</testSourceDirectory>   
    <outputDirectory>${basedir}/target/classes</outputDirectory>   
    <testOutputDirectory>${basedir}/target/test-classes</testOutputDirectory>

这几个元素只在parent build element里面定义，他们设置多种路径结构，他们并不在profile里，所以不能通过profile来修改

build 里面的Extensions：   
它们是一系列build过程中要使用的产品，他们会包含在running bulid‘s classpath里面。他们可以开启extensions，也可以通过提供条件来激活plugins。简单来讲，extensions是在build过程被激活的产品   
    <extensions>   
      <extension>   
        <groupId>org.apache.maven.wagon</groupId>   
        <artifactId>wagon-ftp</artifactId>   
        <version>1.0-alpha-3</version>   
      </extension>   
    </extensions>

reporting设置:

reporting包含site生成阶段的一些元素，某些maven plugin可以生成reports并且在reporting下配置。例如javadoc,maven site等，在reporting下配置的report plugin的方法与build几乎一样，最不同的是build的plugin goals在executions下设置，而reporting的configures goals在reporttest。

excludeDefaults:是否排除site generator默认产生的reports

outputDirectory，默认的dir变成:${basedir}/target/site

report sets:设置execution goals,相当于build里面的executions,不同的是不能够bind a report to another phase,只能够是site

<reporting>   
    <plugins>   
      <plugin>   
        ...   
        <reportSets>   
          <reportSet>   
            <id>sunlink</id>   
            <reports>   
              <report>javadoc</report>   
            </reports>   
            <inherited>true</inherited>   
            <configuration>   
              <links>   
                <link>http://java.sun.com/j2se/1.5.0/docs/api/</link>   
              </links>   
            </configuration>   
          </reportSet>   
        </reportSets>   
      </plugin>   
    </plugins>   
  </reporting>   
reporting里面的厄reportSets和build里面的executions的作用都是控制pom的不同粒度去控制build的过程，我们不单要配置plugins，还要配置那些plugins单独的goals。

更多项目信息:

name:项目除了artifactId外，可以定义多个名称  
description: 项目描述  
url: 项目url  
inceptionYear:创始年份

**Licenses**  
<licenses>  
  <license>  
    <name>Apache 2</name>  
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>  
    <distribution>repo</distribution>  
    <comments>A business-friendly OSS license</comments>  
  </license>  
</licenses>

列出本工程直接的licenses，而不要列出dependencies的licenses

配置组织信息:  
  <organization>  
    <name>Codehaus Mojo</name>  
    <url>http://mojo.codehaus.org</url>  
  </organization>

很多工程都受到某些组织运行，这里设置基本信息

配置开发者信息:

例如：一个开发者可以有多个roles，properties是   
<developers>  
    <developer>  
      <id>eric</id>  
      <name>Eric</name>  
      <email>eredmond@codehaus.org</email>  
      <url>http://eric.propellors[**.NET**](http://lib.csdn.net/base/dotnet)</url>  
      <organization>Codehaus</organization>  
      <organizationUrl>http://mojo.codehaus.org</organizationUrl>  
      <roles>  
        <role>architect</role>  
        <role>developer</role>  
      </roles>  
      <timezone>-6</timezone>  
      <properties>  
        <picUrl>http://tinyurl.com/prv4t</picUrl>  
      </properties>  
    </developer>  
  </developers>

环境设置:

issueManagement:bug跟踪管理系统,定义defect tracking system缺陷跟踪系统，比如有（bugzilla,testtrack,clearquest等）.

例如:

  <issueManagement>   
    <system>Bugzilla</system>   
    <url>http://127.0.0.1/bugzilla/</url>   
  </issueManagement>

仓库：

Repositories：pom里面的仓库与setting.xml里的仓库功能是一样的。主要的区别在于，pom里的仓库是个性化的。比如一家大公司里的setting文件是公用 的，所有项目都用一个setting文件，但各个子项目却会引用不同的第三方库，所以就需要在pom里设置自己需要的仓库地址。

repositories：要成为maven2的repository artifact，必须具有pom文件在$BASE\_REPO/groupId/artifactId/version/artifactId-version.pom   
BASE\_REPO可以是本地，也可以是远程的。repository元素就是声明那些去查找的repositories   
默认的central Maven repository在http://repo1.maven.org/maven2/

<repositories>   
    <repository>   
      <releases>   
        <enabled>false</enabled>   
        <updatePolicy>always</updatePolicy>   
        <checksumPolicy>warn</checksumPolicy>   
      </releases>   
      <snapshots>   
        <enabled>true</enabled>   
        <updatePolicy>never</updatePolicy>   
        <checksumPolicy>fail</checksumPolicy>   
      </snapshots>   
      <id>codehausSnapshots</id>   
      <name>Codehaus Snapshots</name>   
      <url>http://snapshots.maven.codehaus.org/maven2</url>   
      <layout>default</layout>   
    </repository>   
  </repositories>

release和snapshots：是artifact的两种policies，pom可以选择那种政策有效。   
enable：本别指定两种类型是否可用，true or false   
updatePolicy:说明更新发生的频率always 或者 never 或者 daily（默认的）或者 interval:X（X是分钟数）

checksumPolicy：当Maven的部署文件到仓库中，它也部署了相应的校验和文件。您可以选择忽略，失败，或缺少或不正确的校验和警告。

layout：maven1.x与maven2有不同的layout，所以可以声明为default或者是legacy（遗留方式maven1.x）。

插件仓库：

pluginRepositories：与Repositories具有类似的结构，只是Repositories是dependencies的home，而这个是plugins 的home。

分发管理:

distributionManagement :管理distribution和supporting files。

downloadUrl：是其他项目为了抓取本项目的pom’s artifact而指定的url，就是说告诉pom upload的地址也就是别人可以下载的地址。   
status：这里的状态不要受到我们的设置，maven会自动设置project的状态，有效的值：none：没有声明状态，pom默认的；converted：本project是管理员从原先的maven版本convert到maven2的；partner：以前叫做synched，意思是与partner repository已经进行了同步；deployed：至今为止最经常的状态，意思是制品是从maven2 instance部署的，人工在命令行deploy的就会得到这个；verified：本制品已经经过验证，也就是已经定下来了最终版。   
repository：声明deploy过程中current project会如何变成repository，说明部署到repository的信息。   
    <repository>   
      <uniqueVersion>false</uniqueVersion>   
      <id>corp1</id>   
      <name>Corporate Repository</name>   
      <url>scp://repo1/maven2</url>   
      <layout>default</layout>   
    </repository>   
    <snapshotRepository>   
      <uniqueVersion>true</uniqueVersion>   
      <id>propSnap</id>   
      <name>Propellors Snapshots</name>   
      <url>sftp://propellers[**.net**](http://lib.csdn.net/base/dotnet)/maven</url>   
      <layout>legacy</layout>   
    </snapshotRepository>   
id, name:：唯一性的id，和可读性的name   
uniqueVersion：指定是否产生一个唯一性的version number还是使用address里的其中version部分。true or false   
url：说明location和transport protocol   
layout：default或者legacy

profiles:pom4.0的一个新特性就是具有根据environment来修改设置的能力

它包含可选的activation（profile的触发器）和一系列的changes。例如test过程可能会指向不同的[**数据库**](http://lib.csdn.net/base/mysql)（相对最终的deployment）或者不同的dependencies或者不同的repositories，并且是根据不同的JDK来改变的。那么结构如下：   
  
  <profiles>   
    <profile>   
      <id>test</id>   
      <activation>...</activation>   
      <build>...</build>   
      <modules>...</modules>   
      <repositories>...</repositories>   
      <pluginRepositories>...</pluginRepositories>   
      <dependencies>...</dependencies>   
      <reporting>...</reporting>   
      <dependencyManagement>...</dependencyManagement>   
      <distributionManagement>...</distributionManagement>   
    </profile>   
  </profiles>   
Activation：   
触发这个profile的条件配置如下例：（只需要其中一个成立就可以激活profile，如果第一个条件满足了，那么后面就不会在进行匹配。   
    <profile>   
      <id>test</id>   
      <activation>   
        <activeByDefault>false</activeByDefault>   
        <jdk>1.5</jdk>   
        <os>   
          <name>Windows XP</name>   
          <family>Windows</family>   
          <arch>x86</arch>   
          <version>5.1.2600</version>   
        </os>   
        <property>   
          <name>mavenVersion</name>   
          <value>2.0.3</value>   
        </property>   
        <file>   
          <exists>${basedir}/file2.properties</exists>   
          <missing>${basedir}/file1.properties</missing>   
        </file>   
      </activation>

激活profile的方法有多个：setting文件的activeProfile元素明确指定激活的profile的ID，在命令行上明确激活Profile用-P flag 参数   
查看某个build会激活的profile列表可以用：mvn help:active-profiles