Maven中文手册

Maven最早始于Jarkarta Turbine项目，目的是为了简化构建过程。这个项目下面有 几个子项目，每个子项目都有自己的Ant文件，但是区别很小，而且生成的JAR 文件都要check in到CVS中。我们希望有一种标准的方式来构建这些项目，一种可以 清晰描述项目的方式，一种易于发布项目信息的方式，一种在多个项目之间共享JARs 的方式。

这个结果就是产生了一个可以用于构建、管理任何基于java的项目。我们希望我们创造 的这个工具可以让Java开发者的日常工作更加轻松，并有助于理解基于java的项目.

项目对象模型

Maven是基于项目对象模型(POM)的概念而创建的。在这个模型中，所有由Maven产生的 artifact都是清晰定义的项目模型的结果。构建，文档，源码度量，源码交叉引用和其他 任何由Maven plug-in提供的东西都是由POM来控制的。

POM 处理机制

这篇文档简单的描述了Maven执行过程中是如何处理POM的。这里有一些简单的POM例子 来展示继承机制和插值机制。

POM 插值机制

POM(通常以project.xml的名字出现)现在已经被当作Jelly脚本来处理了。大部分时 候，用户无须关心project.xml文件是不是真正的Jelly脚本，但是，如果需要的话， 也可以使用内置值。我也不愿看到逻辑控制语句出现在project.xml中，但是由于 project.xml实际上已经是一个隐含的jelly的脚本，所以它会有足够的灵活性:-)。 下面是一个简单的例子。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project>

  <pomVersion>3</pomVersion>
  <groupId>maven</groupId>
  <artifactId>maven</artifactId>
  <name>Maven</name>
```

```
  <currentVersion>1.0-b5-dev</currentVersion>
  <organization>
    <name>Apache Software Foundation</name>
    <url> http://jakarta.apache.org/</url>
    <logo>/images/jakarta-logo-blue.gif</logo>
  </organization>
  <inceptionYear>2001</inceptionYear>
  <package>org.apache.${pom.artifactId}</package>
  <logo>/images/${pom.artifactId}.jpg</logo>

  <description>Maven is a project that was created in ${pom.inceptionYear}.</description>
  <shortDescription>${pom.name} is a Java Project Management Tool</shortDescription>

</project>
```

POM 继承机制

现在有一种简单方式可以用于在POM中进行继承，下面是一个简单的例子:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <project>
    <extend>project.xml</extend>

    <groupId>super-extendo</groupId>
    <artifactId>super-extendo<artifactId>
    <name>Super Extendo</name>

    <build>
      <unitTest>
        <includes>
          <include>**/*Test*.java</include>
        </includes>
        <excludes>
          <exclude>**/TestAll.java</exclude>
          <exclude>**/*Abstract*.java</exclude>
        </excludes>
      </unitTest>
    </build>
  </project>
```

目前对POM父对象的解析还相对较弱，现在对一层以上的继承还没有做过任何测试。尽管如此， 单层继承加上插值机制已经能够给带来很多好处。这些机制的意图在于简化构建的共用问题。

你可以这样定义主模板:

```
<project>
  <pomVersion>3</pomVersion>
  <groupId>commons</groupId>
  <artifactId>commons-master</artifactId>
  <name>Commons Master Maven POM</name>

  <organization>
    <name>Apache Software Foundation</name>
    <url> http://www.apache.org</url>
  </organization>

  <gumpRepositoryId>jakarta</gumpRepositoryId>

  <url> http://jakarta.apache.org/commons/${pom.artifactId}.html</url>
  <issueTrackingUrl> http://nagoya.apache.org/</issueTrackingUrl>
  <siteAddress>jakarta.apache.org</siteAddress>
  <siteDirectory>/www/jakarta.apache.org/commons/${pom.artifactId}/</siteDirectory>
  <distributionDirectory>
    /www/jakarta.apache.org/builds/jakarta-commons/${pom.artifactId}/
  </distributionDirectory>

  <repository>
    <connection>
      scm:cvs server:anoncvs@cvs.apache.org:/home/cvspublic:jakarta-commons/${pom.artifactId}
    </connection>
    <url> http://cvs.apache.org/viewcvs/jakarta-commons/${pom.artifactId}/</url>
  </repository>
  ...
</project>
```

子POM对象可以这样定义:

```
<project>
```

```
  <groupId>commons-betwixt</groupId>
  <artifactId>commons-betwixt</artifactId>
  <name>Betwixt</name>
  ...
</project>
```

这样你就可以在父POM对象中，将子POM对象的${pom.artifactId}替换进去。有许多项目的 构建都以相似的方式进行构建，这样一来，对于项目的公共部分，你就可以使用一个主模板， 然后在子POM对象project.xml中定义有区别的部分，而这部分内容通常很少。

这种机制确实还可以简化那些需要产生多个JAR包的项目。由于project.xml和标准Ant构建 不会相互干扰，我计划在公共部分测试继承机制。

如果你对使用这种机制，DVSL报告会变成什么样感到疑惑，我要说，你很上路。我已经修改 了DVSL报告来适应POM本身，这就是说DVSL转换是基于java对象的。在使用继承和插值机制的 时候，为了正确的产生DVSL报告，这是很有必要的。象上面列出的子模板是无法工作的，我们 需要全面的解析POM。我能说的是，它可以工作了！我所使用的处理方式可能不是最有效率的 方式，但仍有提升的空间。因为POM只会被处理一次(不管怎么说，这就它的原理，我可能漏了 某些东西)，然后到处使用，至少这就是我以前试图做的事情，所以我们很有可能会取得平衡。

如果你不使用继承和插值，那么一切照常工作。maven站点本身一切ok，有几个刚部署的站点 已经使用了我昨晚提交的东西了。

使用插件
Maven是一个很紧凑的内核，围绕着它的是许许多多的插件。Maven所有的功能都是由插件来提供 的。

maven.xml文件
项目中的maven.xml文件是Maven在执行过程中要使用的“定制”文件。

在这个文件中，你可以加入Maven构建中需要的额外处理。或者在Maven的“目标”前后附加自己 的代码，如jar 或 test。

Maven使用Jelly 脚本语言, 任何合法的jelly标签都可以在maven.xml中使用。

Maven所采用的goal功能是由werkz标签库提供。更多的信息请看 wiki页面.

简单的maven.xml例子
注意由于Maven并未缺省的定义'compile'目标，下面这个例子没法运行。

这是一个简单的maven.xml例子

```
<project
  default="nightly-build"
  xmlns:j="jelly:core"
  xmlns:u="jelly:util">

  <goal name="nightly-build">
    <!-- Any ant task, or jelly tags can go here thanks to jeez -->
    <j:set var="goals" value="compile,test" />
    <mkdir dir="${maven.build.dir}" />
    <u:tokenize var="goals" delim=",">${goals}</u:tokenize>
    <j:forEach items="${goals}" var="goal" indexVar="goalNumber">
      Now attaining goal number ${goalNumber}, which is ${goal}
      <attainGoal name="${goal}" />
    </j:forEach>
  </goal>

</project>
```

你可能会注意到这里一些有意思的东西，下面我们将逐一解释。

project节点
project节点, <project>, 是任何 maven.xml 文件的根节点。

项目节点有一个缺省的属性：default="nightly-build"，如果用户只是简单键入 没有参数的maven命令，Maven就会用nightly-build 目标作为缺省目标。

接下来是几个名字空间的声明，如：

xmlns:j="jelly:core"
所有以j:作为前缀的节点，Jelly都会把它视为在core标识下 预定义的标签库。

xmlns:u="jelly:util"
所有以u:作为前缀的节点，Jelly都会把它视为在标识下 预定义的标签库。

所有在maven.xml文件使用的Jelly标签库，都必须在project节点中定义，并且 分配一个名称空间前缀。

Maven已经预先包含了jeez标签库作为空前缀。这个标签库在一个名称空间内包含了 ant 和 werkz 标签库。这样，任何werkz或ant标签都无须名称空间 即可使用，同时也简化了ant的迁移过程。

目标
goal是一个 werkz 标签，类似于Ant的target；它是包含了一系列可执行标签的容器。

由于jeez 标签库已经由maven预先注册了，一个目标(goal)可以包含任何合法的 Ant 标签。

为了执行在maven.xml中定义的目标，你只需要在命令行中为Maven指定目标名即可。要执行例子中定 义的nightly-build你只需执行命令:

maven nightly-build
Maven插件定义的目标需要在目标前加上插件名，这样一来，就不会和你自己的goal冲突，如 jar:jar就是 jar 插件定义的目标，用于创建项目的jar包。

Jelly编程
在每个目标里，由Jelly标签提供功能，让我们来看看例子里的这些代码。

set
<j:set var="goals" value="compile,test" />这行就是一个jelly的core标签set，它使用了project节点中定义的前缀 j:

set标签设置了一个由var属性定义的Jelly变量，值由 value 指定。和Ant的proerties不一样，Jelly变量在被赋值后仍可以改变。

mkdir
<mkdir dir="${maven.build.dir}" />等同于Ant任务 mkdir, 用于创建目录，目录名由变量 ${maven.build.dir}指定。

tokenize
<u:tokenize var="goals" delim=",">${goals}</u:tokenize>这行执行的是Jelly tokenize 标签。这是Jelly util 标签库中标签， 这个标签库已经在项目节点中预先定义：u:

tokenize标签在将节点中的内容分离成一个list，用于后续的处理。

var 属性就是将被于新list的变量。
delim 属性是用于分割字符串中的分隔符。

在这个例子中，tokenize 标签中节点值是一个变量：goals, 在前几行中，这是一个由逗号分隔、compile 与 test 的字符串。

forEach
<j:forEach items="${goals}" var="goal" indexVar="goalNumber">
  Now attaining goal number ${goalNumber}, which is ${goal}
  <attainGoal name="${goal}" />
</j:forEach>forEach标签提供简单循环功能，节点值就是循环体。

items 属性是一个表达式，是在循环过程中需要遍历的值集合。
集合中的值被逐个取出，存放在var 属性指定的变量中。你可以在 forEach 循环体访问这个变量。
indexVar 属性指定了一个计数器(起始基数为0)变量，用于在处理 过程中计数。

forEach 标签的节点值输出了一些在处理过程中的关于目标的文本，并使用 attainGoal werkz 标签来获得(执行?)这些目标。

Maven 配置
属性的处理机制
Maven按下面的顺序依次读入properties文件：

${project.home}/project.properties
${project.home}/build.properties
${user.home}/build.properties

读入的属性遵循“最后的定义有效”原则。也就是说，Maven依次读入properties文件，以新的定义 覆盖旧的定义。${user.home}/build.properties是Maven最后处理的文件。我们把 这些文件称为Maven处理的标准属性文件集。

另外，上述properties文件处理完毕后，Maven才会开始处理系统属性。所以，如果在命令行中使用 象-Dproperty=value这样的形式来指定的属性，它会覆盖先前定义的同名属性。

插件的属性
上述属性文件处理完后，Maven才会开始调用插件，但是 PluginManager 会禁止插件 覆盖已经设置的属性值。这和Maven一般的属性处理方式是相反的，这是因为插件只能在Maven内部的 其他部分初始化完毕后才能处理，但是我们又希望以这样顺序去处理属性：

处理Plug-in 缺省属性
处理${project.home}/project.properties
处理${project.home}/build.properties
处理${user.home}/build.properties

这样一来，即使插件处于属性处理链的后端，也能覆盖插件的任何缺省属性。例如，Checkstyle插件 就定义了如下缺省属性：

maven.checkstyle.format = sun
你可能已经猜出来了，Checksytle插件把Sun的编码规范作为缺省的格式。但是我们仍然可以在标准属性 文件集的任何一个文件中覆盖这个属性。所以如果在${project.home}/project.properties 文件定义了如下属性值：

maven.checkstyle.format = turbine
Checkstyle就会使用Turbine的编码规范。

行为属性
下面的这些属性可以改变Maven的“行为方式”。

属性 描述 缺省值
maven.build.dest 目录名，存放编译生成的类 ${maven.build.dir}/classes
maven.build.dir 存放构建结果的目录，如类文件，文档，单元测试报告等等。
注意： 在${user.home}/build.properties文件中改变maven.build.dir 的缺省值或许会得到一个 较为个性化的目录布局。但是，这会干扰Maven由从源码开始的编译工作，因为它假设jar包 会被创建到${basedir}/target/目录中。
${basedir}/target
maven.build.src 源码目录 ${maven.build.dir}/src
maven.conf.dir 配置文件目录 ${basedir}/conf
maven.docs.dest html格式报告的输出目录 ${maven.build.dir}/docs
maven.docs.omitXmlDeclaration 产生的文档所应包含的xml声明头，如：
<?xml version="1.0"?> false
maven.docs.outputencoding 生成文档的缺省编码 ISO-8859-1
maven.docs.src 用户提供的文档目录 ${basedir}/xdocs
maven.gen.docs xdocs文档转换成html后的输出目录 ${maven.build.dir}/generated-xdocs
maven.home.local maven用于写入用户信息的本机目录，如解开插件包、缓冲数据。 ${user.home}/.maven
maven.mode.online 是否连接internet true
maven.plugin.dir 插件的存放目录 ${maven.home}/plugins
maven.plugin.unpacked.dir 用于展开安装插件的目录 ${maven.home.local}/plugins
maven.repo.central 在进行dist:deploy处理的时候，发行包所要部署的目标机器 login.ibiblio.org

maven.repo.central.directory 在进行dist:deploy处理的时候，发行包所要部署的目标目录。 /public/html/maven
maven.repo.local 本机repository，用于存储下载的jar包。 ${maven.home.local}/repository
maven.repo.remote 远程repository，如果本机repository没有的jar包，maven会从这里下载。 http://www.ibiblio.org/maven
maven.repo.remote.enabled 是否使用远程repository。 true
maven.scp.executable 用于安全拷贝的可执行程序 scp
maven.src.dir 基础源代码目录 ${basedir}/src
maven.ssh.executable 远程命令可执行程序 scp

使用代理
如果你只能通过代理访问，不要紧，Maven为你提供了下列属性：

代理属性 描述
maven.proxy.host 代理的IP地址
maven.proxy.port 代理端口
maven.proxy.username 如果代理需要认证的话，在这里填写用户名。 User name if your proxy requires authentication.

maven.proxy.password 如果代理需要认证的话，在这里填写密码。

如果你实在需要代理，最好在${user.home}/build.properties文件中指定。

```
## -------------------------------------------------------
## ${user.home}/build.properties
## -------------------------------------------------------
.
.
maven.proxy.host = my.proxyserver.com
maven.proxy.port = 8080
maven.proxy.username = username
maven.proxy.password = password
.
.
```

使用多个远程Repositories
你可以在任何Maven可以处理的属性文件中指定使用多个远程Repositories，如：

maven.repo.remote = http://www.ibiblio.org/maven /, http://www.mycompany.com/maven/

项目设置
开始一个新项目
如果你是第一次使用Maven，或者开始一个新的项目，你可以使用GenApp来自动创建Maven项目树。

maven -Dpackage=com.mycompany.app genapp
执行该命令后，屏幕显示如下：

```
 __  __
|  \/  |__ Jakarta _ ___
| |\/| / _ \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|   v. 1.0-beta-9


    [mkdir] Created dir: <target-directory>/genapp/src/java/com/mycompany/app
    [copy] Copying 1 file to <target-directory>/genapp/src/java/com/mycompany/app
    [mkdir] Created dir: <target-directory>/genapp/src/test/com/mycompany/app
    [copy] Copying 3 files to <target-directory>/genapp/src/test/com/mycompany/app
    [copy] Copying 2 files to <target-directory>/genapp
    [copy] Copying 1 file to <target-directory>/genapp/src/conf
    BUILD SUCCESSFUL
    Total time:  3 seconds
```

执行完毕，即生成下面的完整项目目录树：

```
.
|-- project.properties
|-- project.xml
`-- src
    |-- conf
    |   `-- app.properties
    |-- java
    |   `-- com
    |       `-- mycompany
    |           `-- app
    |               `-- App.java
    `-- test
        `-- com
            `-- mycompany
```

```
    `-- app
        |-- AbstractTestCase.java
        |-- AppTest.java
        `-- NaughtyTest.java
```

构建
Maven采用了集中管理库的理念。所有用于构建的jar包都是从集中管理的中心库上取出。目前， 我们的中心库放在这儿 Ibiblio. 在一个典型的Maven项目中，构建所需的 jar包会从中心库下载。Maven只会取回依赖链条上缺失的包。如果你使用Maven来构建几个项目， 这些项目可能会共享一些依赖包：Maven可以在任意个项目中共享同一个包，而无须在同一系统 中保持一个jar包的多个拷贝。

构建生命周期

在CVS中存放jar文件
我们不推荐你在CVS中存放jar包。Maven提倡在本机repository中存放用于构建共享的jar包或 和其他项目包。许多项目都依赖于诸如XML解析器、标准例程，这在Ant构建中，经常会被复制 多份。在Maven中，这些标准例程就存放在本地repository中，供任何项目构建使用。

重载已声明的依赖包
你可能会发现，在某些情况下允许重载POM中已经声明的依赖包会很方便，甚至很有必要。 你可能会希望使用文件系统中的某个JAR包，或者你希望简单重新指定某个包的版本。对于 这样的情况，Maven提供了一种简单的方式，允许你选择构建所需的jar包。我们严重建议 你设置本机repository，但是在简单至上，或在由ant构建迁移的情况下，JAR包重载特性 会显得非常方便。

为了使用JAR包重载特性，你必须将maven.jar.override 属性设置为 on。 只要设置了这个属性，就可以在任何maven处理的属性文件中重载JAR包。

关于JAR重载的指令有两种。第一种允许你指定某个特定的包的路径，作为JAR包；的二种 允许你指定某个特定的JAR包（这个jar包必须已经存在于本机repository）。下面是这两 种方式：

```
maven.jar.artifactId = [path]

maven.jar.artifactId = [version]
```

下面是一个使用JAR重载指令的属性文件例子：

```
# ----------------------------------------------------------------------
# M A V E N  J A R  O V E R R I D E
# ----------------------------------------------------------------------
maven.jar.override = on

# ----------------------------------------------------------------------
# Jars set explicity by path.
# ----------------------------------------------------------------------
maven.jar.a = ${basedir}/lib/a.jar
maven.jar.b = ${basedir}/lib/b.jar

# ----------------------------------------------------------------------
# Jars set explicity by version.
# ----------------------------------------------------------------------
maven.jar.classworlds = 1.0-beta-1
```

使用SNAPSHOT依赖

在Maven中，SNAPSHOP是某个特定项目的最新构建的jar包。如果你的项目依赖于另一个频繁变更 的项目，你可以在你的POM中声明SNAPSHOP依赖，以保持与那个项目的同步。例如，在你的POM中 这样写，你可以与Jelly保持同步更新。

```
<project>
  ...
  <dependencies>

   <dependency>
     <groupId>commons-jelly</groupId>
     <artifactId>commons-jelly</groupId>
     <version>SNAPSHOT</version>
   </dependency>

  </dependencies>
 ...
</project>
```

上述语句的含义是：总是使用Maven repository中最新的Jelly构建包。SNAPSHOT依赖 总会导致依赖失败。在这种情况下，Maven就会去Maven repository取得一个SNAPSHOT 依赖包的拷贝。如果你正在离线工作，Maven就会发出警告：“SNAPSHO依赖包可能已经 过期”。

多项目构建和Reactor

在Maven中，Reactor是一个用于控制多项目构建的工具。Reactor用Werkz包,并根据每个项目 各自定义的依赖关系来安排构建顺序。Reactor的意图在于：鼓励创建/重构具有更小、更松散、 更有一致性单元的项目。r源于面向组件编程，在向组件编程中，整个系统由许多松散的个体 构成，这些个体需要聚合起来作为一个整体进行部署。

Reactor可以为某个项目集执行一个任意编排的目标列表。所以，如果使用Reactor的话，它 可以用来做项目站点生成，任意jar包生成这样的事情。

目前 db.apache.org 站点就是收集了其子项目的 丰富信息、并与一系列的Velocity 模板合成的。这一过程是由 Jelly Velocity 标签库来完成的。 这里是其中的机制，有兴趣可以看看。

Plexus(一个基于Avalon的容器)组件的manifest是遍历所有Plexus组件、并聚合而成的。这里 使用的还是Velocity模板和Jelly Volocity标签库。 这里是其中的机制，有兴趣可以看看。

这里是一个使用Reactor标签，遍历一系列组件，最后产生项目站点的例子:

```
 <goal
  name="project-site"
  description="Generate the site for the entire project">

  <!-- Generate the site for each of the components. -->

  <maven:reactor
    basedir="${basedir}/components"
    postProcessing="true"
    includes="*/project.xml"
    excludes="bad-component/*"
    goals="site"
    banner="Generating Site"
    ignoreFailures="true"/>

  <!--
   |
   | At this point because 'postProcessing' was turned on we have all
   | the processed POMs available to us in the ${reactorProjects} variable.
   |
   -->
```

```
    <j:forEach var="reactorProject" items="${reactorProjects}">

      ... Do whatever you want with the harvested information ...

    </j:forEach>
  </goal>
```

如果你需要聚合N个项目的文档，形成一个统一的视图，你可以参考 db.apache.org站点，它从所有的子项目中收集信息，最终形成一个有导航条，邮件列表，开发者列表和源码repository统一的站点。这解决 了在收集子项目信息过程中的冗余、易于产生错误的问题。db.apache.org的子项目只需关注 自身的信息，却可以自动的集成到高层的项目站点中。

离线构建

如果你需要离线构建，你得在命令行中使用离线选项:

maven -o jar:jar

这等同于:

maven -Dmaven.mode.online=false jar:jar

或者你可以在属性文件中设置 maven.mode.online 属性为false。

在联机模式下，maven总是下载 SNAPSHOT 依赖包, 请参考 使用SNAPSHOT依赖。如果本机repository 上的当前版本比服务器上的还要新，你也许不想maven覆盖你的当前版本。如果不想下载，你可以 设置属性:

maven.repo.remote.enabled=false

在maven beta 9版本中，这个属性并没有起作用(新版本已经解决了这个问题)。可以这样来清空 maven.repo.remote 属性:

maven.repo.remote.enabled=

在命令行或属性文件中设置都一样。

在下面的情形中，你需要同时设置maven.repo.remote.enabled=false maven.mode.online=true：希望javadoc插件的"-link"选项起作用，而又不希望 下载任何(包括snapshot)包。

命令行选项介绍

如果你用命令行来构建，这里提供了一个关于命令行选项的快速参考。

Options:
-D,--define arg 定义一个系统属性
-E,--emacs 输出简洁的日志信息
-X,--debug 输出debug信息
-b,--nobanner 禁止 logo banner
-d,--dir arg 设置指定的工作目录
-e,--exception 输出异常堆栈信息
-f,--find arg 通过搜寻项目文件来确定执行的项目文件及其工作目录
-g,--goals 显示可用的目标
-h,--help 显示帮助信息
-i,--info 显示系统信息
-o,--offline 离线构建
-p,--pom arg 指定项目文件
-v,--version 显示版本号
测试
测试资源
通常情况下，你需要在测试类路径上为测试指定某些资源。你可以在POM中使用

下面的例子说明了如何递归的包含在${basedir}/src/test目录中所有以.xml和.properties为扩展名 的文件：

```
<project>
  ...
  <build>
    ...
    <unitTest>
      ...
      <resources>
        <resource>
          <directory>${basedir}/src/test</directory>
          <includes>
            <include>**/*.xml</include>
            <include>**/*.properties</include>
          </includes>
        </resource>
      </resources>
    </unitTest>
  </build>
```

</project>
下面的例子说明了如何递归的包含在${basedir}/src/test目录中所有以.xml和.properties为扩展名 的文件，但naughty.properties文件除外。请注意增加的project/build/unitTest/resources/excludes 节点：

```
<project>
  ...
  <build>
    ...
    <unitTest>
      ...
      <resources>
        <resource>
          <directory>${basedir}/src/test</directory>
          <includes>
            <include>**/*.xml</include>
            <include>**/*.properties</include>
          </includes>
          <excludes>
            <exclude>naughty.properties</exclude>
          </excludes>
        </resource>
      </resources>
    </unitTest>
  </build>
</project>
```

运行一个单独的测试
下面的命令运行了一个单独的测试：

```
maven -Dtestcase=org.foo.bar.MyTest test:single-test
```

打包
JAR 资源

```
<project>
  ...
  <build>
    <resources>
      <!-- A -->
```

```
      <resource>
        <directory>${basedir}/src/conf</directory>
        <includes>
          <include>*.xml</include>
          <include>*.properties</include>
        </includes>
      </resource>
      <!-- B -->
      <resource>
        <directory>${basedir}/src/messages</directory>
        <targetPath>org/apache/maven/messages</targetPath>
        <includes>
          <include>messages*.properties</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

部署

固化SNAPSHOT依赖

在开发时使用SNAPSHOT依赖包非常方便，但是在部署的时候，你需要把所有的SNAPSHOT依赖包 固定下来，以便发布一个包含固定jar的版本。如果你发布了一个含SNAPSHOT依赖包的项目，而 这些SNAPSHOT在项目发布后发生了变更，你可能很快就会发现这个版本没法儿工作。

当SNAPSHOT以时间戳为版本号部署到Maven上时，Maven可以识别foo-SNAPSHOT.jar实际上相当 于foo-20030101.010101.jar。当部署的时刻来临，你可以用下面的命令，让Maven把SNAPSHOT 版本号变为时间戳版本号。

maven convert-snapshots

Maven会提供交互的方式让你选择哪个SNAPSHOPT依赖包应该固化下来。这一过程结束后，Maven 就会把刚才你的选择写会你的POM。

如果你希望Maven简单的、尽可能最好的固化SNAPSHOT依赖包，你可以用下面的命令:

maven convert-snapshots-auto

拷贝依赖JAR包

在maven.xml这样写下面的语句是最简单的拷贝依赖包的方法。

```
<project
  xmlns:deploy="deploy">

  <goal name="deploy-my-project">
    ...
    <deploy:copy-deps todir="/path"/>
    ...
  </goal>

</project>
```

用deploy:copy-deps标签你就可以简单的把项目所有的依赖包拷贝到任何 目录，如果希望排除某个依赖包，需要给出一个依赖包的id列表，列表中依赖包就不会被拷贝。

```
<project
  xmlns:deploy="deploy">

  <goal name="deploy-my-project">
    ...
    <deploy:copy-deps todir="/path" excludes="servletapi,commons-util"/>
    ...
  </goal>
</project>
```

命名约定

这部分将简要的描述项目对象模型(POM)中命名约定。本篇文档希望可以统一各式各样jar包命名 方法，这些jar包一般为java开发者社区广泛使用。

规则和指南

项目

一个项目必须由一个唯一的的标识，标识由a-z小写字母和连线号构成，其首字母必须以小写字母 开头。

```
<project>
  <groupId>bar</groupId>
  <artifactId>foo</artifactId>
  ...
</project>
```

所有项目间的引用都由组id和包id构成。到现在为止，在POM与此相关的是下面将要（上面提到的？）谈到的项目

的 依赖声明。

项目的名称应该易于阅读，可用于文档。

```
<project>
  <groupId>bar</groupId>
  <artifactId>foo</artifactId>
  <name>The Grand Master Foo</name>
  ...
</project>
```

项目应该有一个组标识，组标识是名称的基础部分。

```
<project>
  <groupId>bar</groupId>
  <artifactId>foo</artifactId>
  <name>The Grand Master Foo</name>
  ...
</project>
```

所有项目发布的包都应基于项目唯一的标识，并且放在基于项目的组id的目录中。对于上面的 项目来说，假定发布的包是jar类型的包，我们就会有如下目录结构:

```
repository
 |
 +-- bar
     |-- distribution
     `-- jar
         |-- foo-1.0.jar
         `-- foo-2.0.jar
```

依赖

一个理想的典型依赖声明的例子可能会象这样:

```
<project>
  <groupId>yourProject</groupId>
  <artifactId>yourArtifact</artifactId>
  <name>Your Groovey Machine</name>
  ...
```

```
  <dependencies>

   <dependency>
     <groupId>bar</groupId>
     <artifactId>foo</artifactId>
     <version>1.0</version>
   </dependency>

 </dependencies>

</project>
```

这样一来，依赖于id为foo1.0 版本的bar项目 属于org.foo.bar组。这个依赖会被解析为本地repository中jar文件。上面的情形假定发布的包是基于包id命名的。因此对于上述依赖，Maven就会采用 foo-1.0.jar作为jar名。

就像用户反映的那样，这种理想的方式并不能适应所有的项目。有几种情形我们必须考虑调整 理想的依赖机制：

发布的jar包的并未使用项目作为基础名，如xercesImpl就是所有发布的jar包的基础名 它与gump id和maven中的id都不一样。
发布的jar包没有使用任何版本声明，如许多commons组件并没有把版本号作为包名的一部分。
有些项目即没有使用项目id作为基础名也没有版本声明。例如，最坏的情形是Java Activation Framework 的jar包，它没有遵循其他Sun的命名规则，也没有在jar包中声明 版本号，甚至在manifest中都没有任何版本信息。

在多数情形下，任何例外的情况都可以用 <jar>(可选节点)、或者是重命名来解决。 现在，许多Jarkata的产品在repository已经被重命名，因为在不久的未来，绝大多数的Jarkarta的 项目将用maven进行构建。但这不是公司政策，我们不需要解决所有情况的问题。

发布多个包的项目
Maven的依赖机制完全支持任何形式的多项目包。 Maven's dependency mechanism completely supports multiple project artifacts of any given type.

下面的代码包含了ant的主jar包和optional包加上hypothetical包。

```
<dependencies>

  <!-- A -->
```

```
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant</artifactId>
    <version>1.4.1</version>
  </dependency>

  <!-- B -->
  <dependency>
    <groupId>ant</groupId>
    <artifactId>ant-optional</artifactId>
    <version>1.4.1</version>
  </dependency>

  <!-- C -->
  <dependency>
    <groupId>ant</groupId>
    <artifactId>poorly-named</artifactId>
    <version>1.4.1</version>
  </dependency>

</dependencies>
```

所以A), B) 和 C)实际上是指向属于同一组的单个包的指针，这就是说，一个单独的依赖就是对某个组中的某个 包的引用。目前artifact如果没有特别说明的话，一般指的是jar包。但是依赖也可能是一个war文件或一个 发行包。我们试图在提供多数情况下(构建大多需要jar包)使用的缺省方式的同时，也顾及灵活性。

远程Repository布局

这部分文档将对Maven的远程repositories做一个概要的说明。目前，主repository位于 Ibiblio，但是你也可以自己创建一 个远程repositories，其结果描述可以在这部分文档中找到。

任何独立的项目在repository都有自己的目录。每个项目有一个唯一的项目id和同名目录， 项目的发行包就放在这个目录中。

项目的目录中可以存放各种各样的artifacts，目前最为广泛使用的两种类型是jar包和发行包 下面是一个远程repository快照:

```
repository
|
```

```
|-- ant
|   |-- distribution
|   `-- jars
|       |-- ant-1.4.1.jar
|       `-- ant-optional-1.4.1.jar
+-- jsse
|-- distribution
`-- jars
    |-- jsse.jar
    |-- jcert.jar
    `-- jnet.jar
```

本地Repository布局

应该有一个接口集合的紧凑实现使本地repository目录结构更加随意一些。现在我只使用 了一个类来实现，显得有些臃肿。我还是认为本地和远程repositories应保持相同的结构 但是为了使用更乐于使用，我开始设计一些接口，用于满足用户自己安排自己本地Repository 布局，相似的远程Repositor也在设计中。尽管目前本地repository仍与远程repository 完全一致，我希望听到更多的用户反馈，以促进这些接口的开发，或者完全抛弃它。

生成项目站点

Maven可以为项目产生一个完成的web站点。这个web站点将包含各种各样、由许多Maven插件 产生的报告，如javadocs，代码度量，单元测试，changlog，及其它... 本节文档接下来的 部分将阐述Maven支持的站点定制选项。

颜色

考虑到一致性，缺省情况下，所有Maven产生的站点都使用相似的外观。这使用户一眼就能 认出由Maven产生的站点，产生熟悉的感觉，更加方便信息的获取。当然，Maven开发组也 考虑到用户可能会希望定制他们的站点。修改Maven产生的站点，最简单的方法是指定你要 使用的颜色搭配。这只需要在project.properties简单的设置适当的属性 即可。下面是所有可用的属性描述在 xdoc plugin。

风格

如果你感觉颜色太过于简单，无法满足你的要求，你可以用自己的stylesheet。为了重载 在${basedir}/xdocs/stylesheets/中的maven.css stylesheet，你可以通过设置maven.javadoc.stylesheet 属性来指定你的 javadoc风格。

报告

最后，如果你希望定制哪些报告需要自动产生，你需要在你的项目描述中包含 reports 标签库。 到目前为止，如果没有reports标签指定，所有插件报告都会产生。这个标签 允许你选择产生哪些报告而排斥另外一些报告，而且允许你安排顺序。

目前的标准 报告集在xdoc中是硬编码实现的，它们是:

maven-changelog-plugin
maven-changes-plugin
maven-checkstyle-plugin
maven-developer-activity-plugin
maven-file-activity-plugin
maven-javadoc-plugin
maven-jdepend-plugin
maven-junit-report-plugin
maven-jxr-plugin
maven-license-plugin
maven-linkcheck-plugin
maven-pmd-plugin
maven-tasklist-plugin

如果你希望产生缺省的报告，但是还想增加或删除一两个，最好的方法是：给 xdoc:register-reports目标写一个前置目标(postGoal)。

```
<postGoal name="xdoc:register-reports">
  <attainGoal name="maven-changelog-plugin:deregister"/>
  <attainGoal name="maven-myown-plugin:register"/>
</postGoal>
```

排除所有Maven产生的内容

某些情况下，用户需要构建只有文档的站点，或是构建聚合许多子项目的站点，但不希望 在站点上有个整体“项目文档”导航条。设置 maven.xdoc.includeProjectDocumentation属性值为no 即可解决这一问题。缺省的，Maven会包含“项目文档”导航条，下面挂的是许多Maven 生成的报告。

发表于 @ 2005年04月15日 01:01:00 | 评论(0) | 举报 | 收藏

新一篇:hibernate中文论坛

用 户 名: huapuyu6 匿名评论

发表评论