**1.前言**

Maven，发音是[`meivin]，"专家"的 意思。它是一个很好的项目管理工具，很早就进入了我的必备工具行列，但是这次为了把project1项目完全迁移并应用maven，所以对maven进行 了一些深入的学习。写这个学习笔记的目的，一个是为了自己备忘，二则希望能够为其他人学习使用maven 缩短一些时间。

**2. maven概要**

首先我把maven的概念快速的梳理一下，让我们快速地建立起一个比较精确的maven应用场景。

**2.1 maven不是什么**

读书时候要先限定范围，避免一些有害的遐想。要说maven不是什么，我们可以从如下几个要点来展开

1. maven不是ant，也不是make。   
   我们以前接触的构建工具，需要写一些详细的步骤，比如： compile project1/src/\*.java 等类似的语句。这些语句正是我们使用ant和make所要编写的东西。maven采用了"约定优于配置"的方法，一些开发常用的操作和步骤已经固化在 maven中，所以使用者不再需要去编写那些烦人的语句了。同时，maven内置了开发流程的支持，它不仅能够编译，同样能够打包、发布，也能够一气呵成 做完这些所有的步骤。
2. maven不是ivy   
   依赖管理是maven的功能之一，虽然很多人包括我以前都是只用它的依赖管理功能，但是要深入运用的话，我们就可以看到更多的内容。更重要的是，maven在依赖关系中加入了scope的概念，进一步细化了依赖关系的划分。

**2.2 maven是什么**

maven将自己定位为一个项目管理工具。它负责管理项目开发过程中的几乎所有的东西：

1. 版本   
   maven有自己的版本定义和规则
2. 构建   
   maven支持许多种的应用程序类型，对于每一种支持的应用程序类型都定义好了一组构建规则和工具集。
3. 输出物管理   
   maven可以管理项目构建的产物，并将其加入到用户库中。这个功能可以用于项目组和其他部门之间的交付行为。
4. 依赖关系   
   maven对依赖关系的特性进行细致的分析和划分，避免开发过程中的依赖混乱和相互污染行为
5. 文档和构建结果   
   maven的site命令支持各种文档信息的发布，包括构建过程的各种输出，javadoc，产品文档等。
6. 项目关系   
   一个大型的项目通常有几个小项目或者模块组成，用maven可以很方便地管理
7. 移植性管理   
   maven可以针对不同的开发场景，输出不同种类的输出结果。

**2.3 maven的生命周期**

maven把项目的构建划分为不同的生命周期(lifecycle)，在我看来，划分的已经是非常仔细了，大家可以参考[这里](http://www.sonatype.com/books/maven-book/reference_zh/lifecycle.html#tbl-default-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。

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。

**2.4 maven的"约定优于配置"**

所谓的"约定优于配置"，在maven中并不是完全不可以修改的，他们只是一些配置的默认值而已。但是使用者除非必要，并不需要去修改那些约定内容。maven默认的文件存放结构如下：

* /项目目录
  + pom.xml 用于maven的配置文件
  + /src 源代码目录
    - /src/main 工程源代码目录
      * /src/main/java 工程java源代码目录
    - /src/main/resource 工程的资源目录
    - /src/test 单元测试目录
      * /src/test/java
  + /target 输出目录，所有的输出物都存放在这个目录下
    - /target/classes 编译之后的class文件

每一个阶段的任务都知道怎么正确完成自己的工作，比如compile任务就知道从src/main/java下编译所有的java文件，并把它的输出class文件存放到target/classes中。

对maven来说，采用"约定优于配置"的策略可以减少修改配置的工作量，也可以降低学习成本，更重要的是，给项目引入了统一的规范。

**2.5 maven的版本规范**

maven使用如下几个要素来唯一定位某一个输出物： groupId:artifactId:packaging:version 。比如 org.springframework:spring:2.5 。每个部分的解释如下：

* groupId   
  团 体，公司，小组，组织，项目，或者其它团体。团体标识的约定是，它以创建这个项目的组织名称的逆向域名(reverse domain name)开头。来自Sonatype的项目有一个以com.sonatype开头的groupId，而Apache Software的项目有以org.apache开头的groupId。
* artifactId   
  在groupId下的表示一个单独项目的唯一标识符。比如我们的tomcat, commons等。不要在artifactId中包含点号(.)。
* version   
  一个项目的特定版本。发布的项目有一个固定的版本标识来指向该项目的某一个特定的版本。而正在开发中的项目可以用一个特殊的标识，这种标识给版本加上一个"SNAPSHOT"的标记。   
  虽 然项目的打包格式也是Maven坐标的重要组成部分，但是它不是项目唯一标识符的一个部分。一个项目的 groupId:artifactId:version使之成为一个独一无二的项目；你不能同时有一个拥有同样的groupId, artifactId和version标识的项目。
* packaging   
  项目的类型，默认是jar，描述了项目打包后的输出。类型为jar的项目产生一个JAR文件，类型为war的项目产生一个web应用。
* classifier   
  很 少使用的坐标，一般都可以忽略classifiers。如果你要发布同样的代码，但是由于技术原因需要生成两个单独的构件，你就要使用一个分类器 （classifier）。例如，如果你想要构建两个单独的构件成JAR，一个使用Java 1.4编译器，另一个使用Java 6编译器，你就可以使用分类器来生成两个单独的JAR构件，它们有同样的groupId:artifactId:version组合。如果你的项目使用本 地扩展类库，你可以使用分类器为每一个目标平台生成一个构件。分类器常用于打包构件的源码，JavaDoc或者二进制集合。

maven有自己的版本规范，一般是如下定义 <major version>.<minor version>.<incremental version>-<qualifier> ，比如1.2.3-beta-01。要说明的是，maven自己判断版本的算法是major,minor,incremental部分用数字比 较，qualifier部分用字符串比较，所以要小心 alpha-2和alpha-15的比较关系，最好用 alpha-02的格式。

maven在版本管理时候可以使用几个特殊的字符串 SNAPSHOT ,LATEST ,RELEASE 。比如"1.0-SNAPSHOT"。各个部分的含义和处理逻辑如下说明：

* SNAPSHOT   
  如 果一个版本包含字符串"SNAPSHOT"，Maven就会在安装或发布这个组件的时候将该符号展开为一个日期和时间值，转换为UTC时间。例 如，"1.0-SNAPSHOT"会在2010年5月5日下午2点10分发布时候变成1.0-20100505-141000-1。   
  这个词只能用于开发过程中，因为一般来说，项目组都会频繁发布一些版本，最后实际发布的时候，会在这些snapshot版本中寻找一个稳定的， 用于正式发布，比如1.4版本发布之前，就会有一系列的1.4-SNAPSHOT，而实际发布的1.4，也是从中拿出来的一个稳定版。
* LATEST   
  指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个snapshot版，具体看哪个时间最后。
* RELEASE   
  指最后一个发布版。

**2.6 maven的组成部分**

maven把整个maven管理的项目分为几个部分，一个部分是源代码，包括源代码本身、相关的各种资源，一个部分则是单元测试用例，另外一部分则是各种maven的插件。对于这几个部分，maven可以独立管理他们，包括各种外部依赖关系。

**2.7 maven的依赖管理**

依赖管理一般是最吸引人使用maven的功能特性了，这个特性让开发者只需要关注代码的直接依赖，比如我们用了spring，就加入spring依赖说明就可以了，至于spring自己还依赖哪些外部的东西，maven帮我们搞定。

任意一个外部依赖说明包含如下几个要素：groupId, artifactId, version, scope, type, optional。其中前3个是必须的，各自含义如下：

* groupId 必须
* artifactId 必须
* version 必须。   
  这里的version可以用区间表达式来表示，比如(2.0,)表示>2.0，[2.0,3.0)表示2.0<=ver<3.0；多个条件之间用逗号分隔，比如[1,3),[5,7]。
* scope 作用域限制
* type 一般在pom引用依赖时候出现，其他时候不用
* optional 是否可选依赖

maven认为，程序对外部的依赖会随着程序的所处阶段和应用场景而变化，所以maven中的依赖关系有作用域(scope)的限制。在maven中，scope包含如下的取值：

* 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。测试范围依赖在之前的???中介绍过。
* 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）。

另外，代码有代码自己的依赖，各个maven使用的插件也可以有自己的依赖关系。依赖也可以是可选的，比如我们代码中没有任何cache依赖，但是hibernate可能要配置cache，所以该cache的依赖就是可选的。

**2.8 多项目管理**

maven的多项目管理也是非常强大的。一般来说，maven要求同一个工程的所有子项目都放置到同一个目录下，每一个子目录代表一个项目，比如

* 总项目/
  + pom.xml 总项目的pom配置文件
  + 子项目1/
    - pom.xml 子项目1的pom文件
  + 子项目2/
    - pom.xml 子项目2的pom文件

按照这种格式存放，就是继承方式，所有具体子项目的pom.xml都会继承总项目pom的内容，取值为子项目pom内容优先。

要设置继承方式，首先要在总项目的pom中加入如下配置

< modules >   
    < module > simple-weather </ module >   
    < module > simple-webapp </ module >   
</ modules >

其次在每个子项目中加入

< parent >   
    < groupId > org.sonatype.mavenbook.ch06 </ groupId >   
    < artifactId > simple-parent </ artifactId >   
    < version > 1.0 </ version >   
</ parent >

即可。

当然，继承不是唯一的配置文件共用方式，maven还支持引用方式。引用pom的方式更简单，在依赖中加入一个type为pom的依赖即可。

< project >   
    < description > This is a project requiring JDBC </ description >   
    ...   
    < dependencies >   
        ...   
        < dependency >   
            < groupId > org.sonatype.mavenbook </ groupId >   
            < artifactId > persistence-deps </ artifactId >   
            < version > 1.0 </ version >   
            < type > pom </ type >   
        </ dependency >   
    </ dependencies >   
</ project >

**2.9 属性**

用户可以在maven中定义一些属性，然后在其他地方用${xxx}进行引用。比如：

< project >   
    < modelVersion > 4.0.0 </ modelVersion >   
    ...   
    < properties >   
        < var1 > value1 </ var1 >   
    </ properties >   
</ project >

maven提供了三个隐式的变量，用来访问系统环境变量、POM信息和maven的settings：

* env   
  暴露操作系统的环境变量，比如env.PATH
* project   
  暴露POM中的内容，用点号(.)的路径来引用POM元素的值，比如${project.artifactId}。另外，java的系统属性比如user.dir等，也暴露在这里。
* settings   
  暴 露maven的settings的信息，也可以用点号(.)来引用。maven把系统配置文件存放在maven的安装目录中，把用户相关的配置文件存放 在~/.m2/settings.xml(unix)或者%USERPROFILE%/.m2/settings.xml(windows)中。

**2.10 maven的profile**

profile 是maven的一个重要特性，它可以让maven能够自动适应外部的环境变化，比如同一个项目，在linux下编译linux的版本，在win下编译 win的版本等。一个项目可以设置多个profile，也可以在同一时间设置多个profile被激活（active）的。自动激活的 profile的条件可以是各种各样的设定条件，组合放置在activation节点中，也可以通过命令行直接指定。profile包含的其他配置内容可 以覆盖掉pom定义的相应值。如果认为profile设置比较复杂，可以将所有的profiles内容移动到专门的 profiles.xml 文件中，不过记得和pom.xml放在一起。

activation节点中的激活条件中常见的有如下几个：

1. os   
   判断操作系统相关的参数，它包含如下可以自由组合的子节点元素
   * message - 规则失败之后显示的消息
   * arch - 匹配cpu结构，常见为x86
   * family - 匹配操作系统家族，常见的取值为：dos，mac，netware，os/2，unix，windows，win9x，os/400等
   * name - 匹配操作系统的名字
   * version - 匹配的操作系统版本号
   * display - 检测到操作系统之后显示的信息
2. jdk   
   检查jdk版本，可以用区间表示。
3. property   
   检查属性值，本节点可以包含name和value两个子节点。
4. file   
   检查文件相关内容，包含两个子节点：exists和missing，用于分别检查文件存在和不存在两种情况。

**3. maven的操作和使用**

maven的操作有两种方式，一种是通过mvn命令行命令，一种是使用maven的eclipse插件。因为使用eclipse的maven插件操作起来比较容易，这里就只介绍使用mvn命令行的操作。

**3.1 maven的配置文件**

maven 的主执行程序为mvn.bat，linux下为mvn.sh，这两个程序都很简单，它们的共同用途就是收集一些参数，然后用 java.exe来运行maven的Main函数。maven同样需要有配置文件，名字叫做settings.xml，它放在两个地方，一个是maven 安装目录的conf目录下，对所有使用该maven的用户都起作用，我们称为主配置文件，另外一个放在 %USERPROFILE%/.m2/settings.xml下，我们成为用户配置文件，只对当前用户有效，且可以覆盖主配置文件的参数内容。还有就是 项目级别的配置信息了，它存放在每一个maven管理的项目目录下，叫pom.xml，主要用于配置项目相关的一些内容，当然，如果有必要，用户也可以在 pom中写一些配置，覆盖住配置文件和用户配置文件的设置参数内容。

一般来说，settings文件配置的是比如repository库路径之类的全局信息，具体可以参考[官方网站的文章](http://maven.apache.org/settings.html) 。

**3.2 创建新工程**

要创建一个新的maven工程，我们需要给我们的工程指定几个必要的要素，就是maven产品坐标的几个要素：groupId, artifactId，如果愿意，你也可以指定version和package名称。我们先看一个简单的创建命令：

d:\work\temp>mvn archetype:create -DgroupId=com.abc -DartifactId=product1 -DarchetypeArtifactId=maven-archetype-webapp

首先看这里的命令行参数的传递结构，怪异的 -D参数=值 的方式是 java.exe 要求的方式。这个命令创建一个web工程，目录结构是一个标准的maven结构，如下：

D: .   
└─mywebapp   
    │  pom . xml   
    │   
    └─src   
        └─main   
            ├─resources   
            └─webapp   
                │  index . jsp   
                │   
                └─WEB-INF   
                    web . xml

大家要注意，这里目录结构的布局实际上是由参数 archetypeArtifactId 来决定的，因为这里传入的是 maven-archetype-webapp 如果我们传入其他的就会创建不同的结构，默认值为 maven-archetype-quickstart ，有兴趣的读者可以参考[更详细的列表](http://docs.codehaus.org/display/MAVENUSER/Archetypes+List) ，我把部分常用的列表在这里：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Artifact** | **Group** | **Version** | **Repository** | **Description** |
| maven-archetype-j2ee-simple | org.apache.maven.archetypes |  |  | A simple J2EE Java application |
| maven-archetype-marmalade-mojo | org.apache.maven.archetypes |  |  | A Maven plugin development project using marmalade |
| maven-archetype-plugin | org.apache.maven.archetypes |  |  | A Maven Java plugin development project |
| maven-archetype-portlet | org.apache.maven.archetypes |  |  | A simple portlet application |
| maven-archetype-profiles | org.apache.maven.archetypes |  |  |  |
| maven-archetype-quickstart | org.apache.maven.archetypes |  |  |  |
| maven-archetype-simple | org.apache.maven.archetypes |  |  |  |
| maven-archetype-site-simple | org.apache.maven.archetypes |  |  | A simple site generation project |
| maven-archetype-site | org.apache.maven.archetypes |  |  | A more complex site project |
| maven-archetype-webapp | org.apache.maven.archetypes |  |  | A simple Java web application |
| maven-archetype-har | net.sf.maven-har | 0.9 |  | Hibernate Archive |
| maven-archetype-sar | net.sf.maven-sar | 0.9 |  | JBoss Service Archive |

大家可以参考[更详细的 archetype:create 帮助](http://maven.apache.org/archetype/maven-archetype-plugin/create-mojo.html) 和 [archtype参考信息](http://maven.apache.org/archetype/maven-archetype-plugin/) 。

**3.3 maven的多项目管理**

多项目管理是maven的主要特色之一，对于一个大型工程，用maven来管理他们之间复杂的依赖关系，是再好不过了。maven的项目配置之间的关系有两种：继承关系和引用关系。   
maven默认根据目录结构来设定pom的继承关系，即下级目录的pom默认继承上级目录的pom。要设定两者之间的关系很简单，上级pom如下设置：

< modules >   
    < module > ABCCommon </ module >   
    < module > ABCCore </ module >   
    < module > ABCTools </ module >   
</ modules >

要记住的是，这里的module是目录名，不是子工程的artifactId。子工程如下设置：

< parent >   
    < groupId > com.abc.product1 </ groupId >   
    < artifactId > abc-product1 </ artifactId >   
    < version > 1.0.0-SNAPSHOT </ version >   
</ parent >   
< artifactId > abc-my-module2 </ artifactId >   
< packaging > jar </ packaging >

这样两者就相互关联起来了，继承关系就设定完毕，所有父工程的配置内容都会自动在子工程中生效，除非子工程有相同的配置覆盖。如果你不喜欢层层递进的目录结构来实现继承，也可以在parent中加入 <relativePath>../a-parent/pom.xml</relativePath> 来制定parent项目的相对目录。继承关系通常用在项目共同特性的抽取上，通过抽取公共特性，可以大幅度减少子项目的配置工作量。

引用关系是另外一种复用的方式，maven中配置引用关系也很简单，加入一个 type 为 pom 的依赖即可。

< dependency >   
    < groupId > org.sonatype.mavenbook </ groupId >   
    < artifactId > persistence-deps </ artifactId >   
    < version > 1.0 </ version >   
    < type > pom </ type >   
</ dependency >

但是无论是父项目还是引用项目，这些工程都必须用 mvn install 或者 mvn deploy 安装到本地库才行，否则会报告依赖没有找到，eclipse编译时候也会出错。

需要特别提出的是复用过程中，父项目的pom中可以定义 dependencyManagement 节点，其中存放依赖关系，但是这个依赖关系只是定义，不会真的产生效果，如果子项目想要使用这个依赖关系，可以在本身的 dependency 中添加一个简化的引用

< dependency >   
    < groupId > org.springframework </ groupId >   
    < artifactId > spring </ artifactId >   
</ dependency >

这种方法可以避免版本号满天飞的情况。

**3.4 安装库文件到maven库中**

在maven中一般都会用到安装库文件的功能，一则是我们常用的hibernate要使用jmx库，但是因为sun的license限制，所以无法将其直接包含在repository中。所以我们使用mvn命令把jar安装到我们本地的repository中

mvn install:install-file -DgroupId=com.sun.jdmk -DartifactId=jmxtools -Dversion=1.2.1 -Dpackaging=jar -Dfile=/path/to/file

如果我们想把它安装到公司的repository中，需要使用命令

mvn deploy:deploy-file -DgroupId=com.sun.jdmk -DartifactId=jmxtools -Dversion=1.2.1 -Dpackaging=jar -Dfile=/path/to/file -Durl=http://xxx.ss.com/sss.xxx -DrepositoryId=release-repo

对于我们的工程输出，如果需要放置到公司的repository中的话，可以通过配置pom来实现

< distributionManagement >   
    < repository >   
        < id > mycompany-repository </ id >   
        < name > MyCompany Repository </ name >   
        < url > scp://repository.mycompany.com/repository/maven2 </ url >   
    </ repository >   
</ distributionManagement >

这里使用的scp方式提交库文件，还有其他方式可以使用，请参考faq部分。然后记得在你的settings.xml中加入这一内容

< settings xmlns ="http://maven.apache.org/SETTINGS/1.0.0"   
          xmlns:xsi ="http://www.w3.org/2001/XMLSchema-instance"   
          xsi:schemaLocation ="http://maven.apache.org/SETTINGS/1.0.0   
          http://maven.apache.org/xsd/settings-1.0.0.xsd" >   
    ...   
    < servers >   
        < server >   
            < id > mycompany-repository </ id >   
            < username > jvanzyl </ username >   
            <!-- Default value is ~/.ssh/id\_dsa -->   
            < privateKey > /path/to/identity </ privateKey >   
            < passphrase > my\_key\_passphrase </ passphrase >   
        </ server >   
    </ servers >   
    ...   
</ settings >

**3.5 maven的变量**

maven定义了很多变量属性,参考这里<http://docs.codehaus.org/display/MAVENUSER/MavenPropertiesGuide>

1. 内置属性
   * ${basedir } represents the directory containing pom.xml
   * ${version } equivalent to ${project.version } or ${pom.version }
2. Pom/Project properties   
   所有pom中的元素都可以用 project. 前缀进行引用,以下是部分常用的
   * ${project.build.directory } results in the path to your "target" dir, this is the same as ${pom.project.build.directory }
   * ${project.build. outputD irectory } results in the path to your "target/classes" dir
   * ${project.name } refers to the name of the project.
   * ${project.version } refers to the version of the project.
   * ${project.build.finalName } refers to the final name of the file created when the built project is packaged
3. 本地用户设定   
   所有用的的 settings.xml 中的设定都可以通过 settings. 前缀进行引用
   * ${settings.localRepository } refers to the path of the user's local repository.
   * ${maven.repo.local } also works for backward compatibility with maven1 ??
4. 环境变量   
   系统的环境变量通过 env. 前缀引用
   * ${env.M2\_HOME } returns the Maven2 installation path.
   * ${java.home } specifies the path to the current JRE\_HOME environment use with relative paths to get for example:   
     <jvm>${java.home}../bin/java.exe</jvm>
5. java系统属性   
   所有JVM中定义的java系统属性.
6. 用户在pom中定义的自定义属性

< project >   
    ...   
    < properties >   
        < my.filter.value > hello </ my.filter.value >   
    </ properties >   
    ...   
</ project >

则引用 ${my.filter.value } 就会得到值 hello

1. 上级工程的变量   
   上级工程的pom中的变量用前缀 ${project.parent } 引用. 上级工程的版本也可以这样引用: ${parent.version }.

**3.6 maven的使用**

我们已经知道maven预定义了许多的阶段（phase），每个插件都依附于这些阶段，并且在进入某个阶段的时候，调用运行这些相关插件的功能。我们先来看完整的maven生命周期：

|  |  |
| --- | --- |
| **生命周期** | **阶段描述** |
| validate | 验证项目是否正确，以及所有为了完整构建必要的信息是否可用 |
| generate-sources | 生成所有需要包含在编译过程中的源代码 |
| process-sources | 处理源代码，比如过滤一些值 |
| generate-resources | 生成所有需要包含在打包过程中的资源文件 |
| process-resources | 复制并处理资源文件至目标目录，准备打包 |
| compile | 编译项目的源代码 |
| process-classes | 后处理编译生成的文件，例如对Java类进行字节码增强（bytecode enhancement） |
| generate-test-sources | 生成所有包含在测试编译过程中的测试源码 |
| process-test-sources | 处理测试源码，比如过滤一些值 |
| generate-test-resources | 生成测试需要的资源文件 |
| process-test-resources | 复制并处理测试资源文件至测试目标目录 |
| test-compile | 编译测试源码至测试目标目录 |
| test | 使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布 |
| prepare-package | 在真正的打包之前，执行一些准备打包必要的操作。这通常会产生一个包的展开的处理过的版本（将会在Maven 2.1+中实现） |
| package | 将编译好的代码打包成可分发的格式，如JAR，WAR，或者EAR |
| pre-integration-test | 执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境 |
| integration-test | 如果有必要的话，处理包并发布至集成测试可以运行的环境 |
| post-integration-test | 执行一些在集成测试运行之后需要的动作。如清理集成测试环境。 |
| verify | 执行所有检查，验证包是有效的，符合质量规范 |
| install | 安装包至本地仓库，以备本地的其它项目作为依赖使用 |
| deploy | 复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关） |

maven核心的插件列表可以参考 <http://maven.apache.org/plugins/index.html> 。这里仅列举几个常用的插件及其配置参数：

1. [clean插件](http://maven.apache.org/plugins/maven-clean-plugin/)   
   只包含一个goal叫做 clean:clean ，负责清理构建时候创建的文件。 默认清理的位置是如下几个变量指定的路径 project.build.directory, project.build.outputDirectory, project.build.testOutputDirectory, and project.reporting.outputDirectory 。
2. [compiler插件](http://maven.apache.org/plugins/maven-compiler-plugin/)   
   包含2个goal，分别是 compiler:compile 和 compiler:testCompile 。可以到这里查看两者的具体参数设置：[compile](http://maven.apache.org/plugins/maven-compiler-plugin/compile-mojo.html) , [testCompile](http://maven.apache.org/plugins/maven-compiler-plugin/testCompile-mojo.html) 。
3. [surefire插件](http://maven.apache.org/plugins/maven-surefire-plugin/)   
   运行单元测试用例的插件，并且能够生成报表。包含一个goal为 surefire:test 。主要参数testSourceDirectory用来指定测试用例目录，参考[完整用法帮助](http://maven.apache.org/plugins/maven-surefire-plugin/test-mojo.html)
4. [jar](http://maven.apache.org/plugins/maven-jar-plugin/)   
   负责将工程输出打包到jar文件中。包含两个goal，分别是 jar:jar , jar:test-jar 。两个goal负责从classesDirectory或testClassesDirectory中获取所有资源，然后输出jar文件到outputDirectory中。
5. [war](http://maven.apache.org/plugins/maven-war-plugin/)   
   负责打包成war文件。常用goal有 war:war ，负责从warSourceDirectory（默认${basedir}/src/main/webapp）打包所有资源到outputDirectory中。
6. [resources](http://maven.apache.org/plugins/maven-resources-plugin/)   
   负责复制各种资源文件，常用goal有 resources:resources ，负责将资源文件复制到outputDirectory中，默认为${project.build.outputDirectory}。
7. [install](http://maven.apache.org/plugins/maven-install-plugin/)   
   负责将项目输出(install:install)或者某个指定的文件(install:install-file)加入到本机库%USERPROFILE%/.m2/repository中。可以用 install:help 寻求帮助。
8. [deploy](http://maven.apache.org/plugins/maven-deploy-plugin/)   
   负责将项目输出(deploy:deploy)或者某个指定的文件(deploy:deploy-file)加入到公司库中。
9. [site](http://maven.apache.org/plugins/maven-site-plugin/)   
   将 工程所有文档生成网站，生成的网站界面默认和apache的项目站点类似，但是其文档用doxia格式写的，目前不支持docbook，需要用其他插件配 合才能支持。需要指出的是，在maven 2.x系列中和maven3.x的site命令处理是不同的，在旧版本中，用 mvn site 命令可以生成reporting节点中的所有报表，但是在maven3中，reporting过时了，要把这些内容作为 maven-site-plugin的configuration的内容才行。详细内容可以参考<http://www.wakaleo.com/blog/292-site-generation-in-maven-3>

**4. maven的使用问答**

除了以下的几个faq条目之外，还有一些faq可以参考

1. [maven 自己的FAQ](http://maven.apache.org/general.html)

兄弟们如果有其他问题，欢迎跟帖提问！

**4.1 依赖关系**

**1) 问：如何增加删除一个依赖关系？**

答：直接在pom文件中加入一个dependency节点，如果要删除依赖，把对应的dependency节点删除即可。

**2) 问：如何屏蔽一个依赖关系？比如项目中使用的libA依赖某个库的1.0版，libB以来某个库的2.0版，现在想统一使用2.0版，如何去掉1.0版的依赖？**

答：设置exclusion即可。

< dependency >   
    < groupId > org.hibernate </ groupId >   
    < artifactId > hibernate </ artifactId >   
    < version > 3.2.5.ga </ version >   
    < exclusions >   
        < exclusion >   
            < groupId > javax.transaction </ groupId >   
            < artifactId > jta </ artifactId >   
        </ exclusion >   
    </ exclusions >   
</ dependency >

**3) 问：我有一些jar文件要依赖，但是我又不想把这些jar去install到mvn的repository中去，怎么做配置？**

答：加入一个特殊的依赖关系，使用system类型，如下：

< dependency >   
    < groupId > com.abc </ groupId >   
    < artifactId > my-tools </ artifactId >   
    < version > 2.5.0 </ version >   
    < type > jar </ type >   
    < scope > system </ scope >   
    < systemPath > ${basedir}/lib/mylib1.jar </ systemPath >   
</ dependency >

但是要记住，发布的时候不会复制这个jar。需要手工配置，而且其他project依赖这个project的时候，会报告警告。如果没有特殊要求，建议直接注册发布到repository。

**4) 问：在eclipse环境中同时使用maven builder和eclipse builder，并且设置项目依赖关系之后，为什么编译会出现artifact找不到错误，但是直接使用命令行mvn构建则一切正常？**

答：在project属性中去掉java build path中对其他 project 的依赖关系，直接在pom中设置依赖关系即可

<!-- 依赖的其他项目 -->   
< dependency >   
    < groupId > com.abc.project1 </ groupId >   
    < artifactId > abc-project1-common </ artifactId >   
    < version > ${project.version} </ version >   
</ dependency >

另外，保证没有其他错误。

**5) 问：我想让输出的jar包自动包含所有的依赖**

答：使用 assembly 插件即可。

< plugin >   
    < artifactId > maven-assembly-plugin </ artifactId >   
    < configuration >   
        < descriptorRefs >   
            < descriptorRef > jar-with-dependencies </ descriptorRef >   
        </ descriptorRefs >   
    </ configuration >   
</ plugin >

**6) 问：我的测试用例依赖于其他工程的测试用例，如何设置？**

答：maven本身在发布的时候，可以发布单纯的jar，也可以同时发布xxx-tests.jar和xxx-javadoc.jar（大家经常在repository中可以看到类似的东西）。我们自己的项目A要同时输出test.jar可以做如下的设置

<!-- 用于把test代码也做成一个jar -->   
< plugin >   
    < groupId > org.apache.maven.plugins </ groupId >   
    < artifactId > maven-jar-plugin </ artifactId >   
    < executions >   
        < execution >   
            < goals >   
                < goal > test-jar </ goal >   
            </ goals >   
        </ execution >   
    </ executions >   
</ plugin >

然后在其他需要引用的工程B中做如下的dependency设置

< dependency >   
    < groupId > com.abc.XXX </ groupId >   
    < artifactId > 工程A </ artifactId >   
    < version > ${project.version} </ version >   
    < type > test-jar </ type >   
    < scope > test </ scope >   
</ dependency >

**7)如何让maven将工程依赖的jar复制到WEB-INF/lib目录下？**

**8)我刚刚更新了一下我的nexus库，但是我无法在eclipse中用m2eclipse找到我新增的库文件**

修改pom.xml文件，将旧版jar的依赖内容中的版本直接修改为新版本即可。

**9）我要的jar最新版不在maven的中央库中，我怎么办？**

将依赖的文件安装到本地库，用如下命令可以完成：

mvn install:install-file

-Dfile=<path-to-file>

-DgroupId=<group-id>

-DartifactId=<artifact-id>

-Dversion=<version>

-Dpackaging=<packaging>

-DgeneratePom=true

Where: <path-to-file> the path to the file to load

<group-id> the group that the file should be registered under

<artifact-id> the artifact name for the file

<version> the version of the file

<packaging> the packaging of the file e.g. jar

**10）**

**4.2 变量**

**1) 问：如何使用变量替换？项目中的某个配置文件比如jdbc.properties使用了一些pom中的变量，如何在发布中使用包含真实内容的最终结果文件？**

答：使用资源过滤功能，比如：

< project >   
    ...   
    < properties >   
        < jdbc.driverClassName > com.mysql.jdbc.Driver </ jdbc.driverClassName >   
        < jdbc.url > jdbc:mysql://localhost:3306/development\_db </ jdbc.url >   
        < jdbc.username > dev\_user </ jdbc.username >   
        < jdbc.password > s3cr3tw0rd </ jdbc.password >   
    </ properties >   
    ...   
    < build >   
        < resources >   
            < resource >   
                < directory > src/main/resources </ directory >   
                < filtering > true </ filtering >   
            </ resource >   
        </ resources >   
    </ build >   
    ...   
    < profiles >   
        < profile >   
            < id > production </ id >   
            < properties >   
                < jdbc.driverClassName > oracle.jdbc.driver.OracleDriver </ jdbc.driverClassName >   
                < jdbc.url > jdbc:oracle:thin:@proddb01:1521:PROD </ jdbc.url >   
                < jdbc.username > prod\_user </ jdbc.username >   
                < jdbc.password > s00p3rs3cr3t </ jdbc.password >   
            </ properties >   
        </ profile >   
    </ profiles >   
</ project >

**2) 问： maven-svn-revision-number-plugin 插件说明**

答： maven-svn-revision-number-plugin 可以从 SVN 中获取版本号，并将其变成环境变量，交由其他插件或者profile使用，详细帮助在[这里](http://maven-svn-revision-number-plugin.googlecode.com/svn/site/usage.html) 。一般和resource的filter机制同时使用

< plugins >   
    < plugin >   
        < groupId > com.google.code.maven-svn-revision-number-plugin </ groupId >   
        < artifactId > maven-svn-revision-number-plugin </ artifactId >   
        < version > 1.3 </ version >   
        < executions >   
            < execution >   
                < goals >   
                    < goal > revision </ goal >   
                </ goals >   
            </ execution >   
        </ executions >   
        < configuration >   
            < entries >   
                < entry >   
                    < prefix > prefix </ prefix >   
                </ entry >   
            </ entries >   
        </ configuration >   
    </ plugin >   
</ plugins >

这段代码负责把resource文件中的内容替换成适当内容

repository = ${prefix.repository}   
path = ${prefix.path}   
revision = ${prefix.revision}   
mixedRevisions = ${prefix.mixedRevisions}   
committedRevision = ${prefix.committedRevision}   
status = ${prefix.status}   
specialStatus = ${prefix.specialStatus}

**3）我的程序有些单元测试有错误，如何忽略测试步骤？**

有好几种方法都可以实现跳过单元测试步骤，一种是给mvn增加命令行参数 -Dmaven.test.skip=true 或者 -DskipTests=true ；另外一种是给surefire插件增加参数，如下：

<project> [...]

<build> <plugins> <plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-surefire-plugin</artifactId> <version>2.8</version> <configuration> <skipTests>true</skipTests> </configuration> </plugin> </plugins> </build> [...]</project>

**4) 如果只想运行单个测试用例，能否实现？**

可以，运行时候增加命令行参数 -Dtest=MyTest 即可，其中MyTest是所需要运行的单元测试用例名称，但是不需要包含package部分。

**4.3 编译**

**1) 问：如何给插件指派参数？比如我要设置一些编译参数**

答：以下内容设定编译器编译java1.5的代码

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

要设置其他插件的参数也可以，请参考对应插件的帮助信息

**2) 问：我的目录是非标准的目录结构，如何设置让maven支持？**

答：指定source目录和test-source目录即可。

< build >   
    < directory > target </ directory >   
    < sourceDirectory > src </ sourceDirectory >   
    < scriptSourceDirectory > js/scripts </ scriptSourceDirectory >   
    < testSourceDirectory > test </ testSourceDirectory >   
    < outputDirectory > bin </ outputDirectory >   
    < testOutputDirectory > bin </ testOutputDirectory >   
</ build >

这个例子把源代码设置成了src目录，测试代码在test目录，所以输出到bin目录。这里要注意，directory如果也设置成bin目录的 话，maven打包的时候会引起死循环，因为directory是所有工作存放的地方，默认包含outputDirectory定义的目录在内。

**3) 我源代码是UTF8格式的，我如何在maven中指定？**

设置一个变量即可

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

{color:blue}以上是官方给出的解决方案，但是经过尝试这样只能影响到resource处理时候的编码{color}，真正有用的是如下配置：

{code:xml}

<build>

...

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

...

</build>

{code}

**. 问：我的项目除了main/java目录之外，还加了其他的c++目录，想要一并编译，如何做？**

答：使用native插件，具体配置方法参考[http://mojo.codehaus.org/maven-native/native-maven-plugin/]

{code:xml}

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>native-maven-plugin</artifactId>

<extensions>true</extensions>

<configuration>

</plugin>

{code}

**. 问：我想要把工程的所有依赖的jar都一起打包，怎么办？**

答：首先修改maven的配置文件，给maven-assembly-plugin增加一个jar-with-dependencies的描述。

{code:xml}

<project>

[...]

<build>

<plugins>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<configuration>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

</plugins>

</build>

[...]

</project>

{code}

然后使用命令打包即可：

mvn assembly:assembly

**. 问：我想把main/scripts中的内容一起打包发布，如何做？**

答：在pom中配置额外的资源目录。如果需要的话，还可以指定资源目录的输出位置

{code:xml}

<build>

...

<resources>

<resource>

<filtering>true</filtering>

<directory>src/main/command</directory>

<includes>

<include>run.bat</include>

<include>run.sh</include>

</includes>

<targetPath>/abc</targetPath>

</resource>

<resource>

<directory>src/main/scripts</directory>

</resource>

</resources>

...

</build>

{code}

**. 问：我有多个源代码目录，但是maven只支持一个main src和一个test src，怎么办？**

答：使用另外一个插件，并仿照如下配置pom

{code:xml}

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>build-helper-maven-plugin</artifactId>

<version>1.1</version>

<executions>

<execution>

<id>add-source</id>

<phase>generate-sources</phase>

<goals>

<goal>add-source</goal>

</goals>

<configuration>

<sources>

<source>src/config/java</source>

<source>src/main/java</source>

<source>src/member/java</source>

</sources>

</configuration>

</execution>

</executions>

</plugin>

{code}

**. 问：我的源代码目录中有一部分文件我不想让maven编译，怎么做？**

答：使用一个maven插件，然后使用includes和excludes。同理，也可以处理资源的过滤。

{code:xml}

<build>

<sourceDirectory>http://www.cnblogs.com/src/java</sourceDirectory>

<plugins>

<plugin>

<groupId>com.sun.enterprise</groupId>

<artifactId>hk2-maven-plugin</artifactId>

<configuration>

<includes>

<include>com/sun/logging/LogDomains.\*</include>

<include>com/sun/enterprise/util/OS.java</include>

<include>com/sun/enterprise/util/io/FileUtils.java</include>

<include>com/sun/enterprise/util/zip/\*\*</include>

<include>com/sun/enterprise/util/i18n/\*\*</include>

<include>com/sun/enterprise/deployment/backend/IASDeploymentException.java</include>

</includes>

<excludes>

<exclude>com/sun/enterprise/config/ConfigBeansFactory.java</exclude>

<exclude>com/sun/enterprise/config/clientbeans/\*\*</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

<resources>

<resource>

<directory>http://www.cnblogs.com/src/java</directory>

<includes>

<include>\*\*/\*.properties</include>

</includes>

</resource>

</resources>

</build>

{code}

**. 问：我的项目是一个纯的html组成的项目，没有任何的java代码，怎么跳过编译过程？**

答：配置如下

{code:xml}

<build>

<sourceDirectory>src/java</sourceDirectory>

<plugins>

<plugin>

<groupId>com.sun.enterprise</groupId>

<artifactId>hk2-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

{code}

**. 问：我的工程里用hibernate，想在编译时候自动生成ddl，如何做？**

答：添加插件

hibernate3-maven-plugin

，按照如下配置：

{code:xml}

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>hibernate3-maven-plugin</artifactId>

<version>2.1</version>

<configuration>

<components>

<component>

<name>hbm2ddl</name>

<implementation>annotationconfiguration</implementation>

</component>

</components>

</configuration>

<dependencies>

<dependency>

<groupId>hsqldb</groupId>

<artifactId>hsqldb</artifactId>

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

</dependency>

</dependencies>

</plugin>

{code}

**. 问：我能用maven支持eclipse RCP项目吗？**

答：当然可以，你可以使用插件 Tycho，详细内容可以参考这里[<http://mattiasholmqvist.se/2010/02/building-with-tycho-part-1-osgi-bundles/>].

<plugin>

<groupid>org.sonatype.tycho</groupid>

<artifactid>target-platform-configuration</artifactid>

<version>0.7.0</version>

<configuration>

<resolver>p2</resolver>

</configuration>

</plugin>

另外，老牌的pde-maven-plugin就不要用了，已经好几年没见更新了。

**4.4 ant互动**

**1) 如何在maven编译时候运行ant脚本？**

使用专门的antrun插件，并且在target标签内部加入ant的代码

<plugin> <artifactId>maven-antrun-plugin</artifactId> <version>1.6</version> <executions> <execution> <phase> <!-- 生命周期阶段 --> </phase> <configuration> <target> <!-- 加入target内部的代码 --> </target> </configuration> <goals> <goal>run</goal> </goals> </execution> </executions> </plugin>

2）如何在ant脚本中引用maven的classpath？

maven给每一个依赖都生成了一个属性，格式为"groupId:artifactId[:classifier]:type"，比如，如果一下例子就显示依赖的org.apache.common-util的jar文件路径

<echo message="Dependency JAR Path: ${org.apache:common-util:jar}"/>

另外，maven还预定义了四个classpath的引用，他们是

* maven.compile.classpath
* maven.runtime.classpath
* maven.test.classpath
* maven.plugin.classpath

3）如何使用antrun插件运行外部的build文件？

很简单，直接在antrun里边使用ant指令即可，如下：

<plugin> <groupId>org.apache.maven.plugins</groupId> <artifactId>maven-antrun-plugin</artifactId> <version>1.6</version> <executions> <execution> <id>compile</id> <phase>compile</phase> <configuration> <target> <!-- 同时传递内置的classpath给外部ant文件 --> <property name="compile\_classpath" refid="maven.compile.classpath"/> <property name="runtime\_classpath" refid="maven.runtime.classpath"/> <property name="test\_classpath" refid="maven.test.classpath"/> <property name="plugin\_classpath" refid="maven.plugin.classpath"/>

<ant antfile="${basedir}/build.xml"> <target name="test"/> </ant> </target> </configuration> <goals> <goal>run</goal> </goals> </execution> </executions> </plugin>

**. 问：如何在ant中使用maven的功能？**

答：使用ant的[maven task|http://maven.apache.org/ant-tasks/index.html]，不过只有ant 1.6以上和jdk 1.5环境才支持。

**h4. 测试相关**

**. 问：如何忽略某个阶段的结果？比如单元测试不一定要全正确**

答：给插件增加testFailureIgnore参数，并设置为false。如果要屏蔽该阶段，则用

<skip>true</skip>

{code:xml}

<project>

[...]

<build>

<plugins>

<plugin>

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

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

<configuration>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

</plugins>

</build>

[...]

</project>

{code}

**. 问：我如何在maven中加入PMD，CheckStyle，JDepend等检查功能？**

答：加入PMD检查，以下代码如果在

reporting

节点中加入则在

mvn site

中执行，如果在

build

节点中加入，则在build的时候自动运行检查。详细配置参考[pmd插件使用说明|http://maven.apache.org/plugins/maven-pmd-plugin/]

{code:xml}

<plugins>

<plugin>

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

<artifactId>maven-pmd-plugin</artifactId>

<version>2.5</version>

</plugin>

</plugins>

{code}

加入 checkstyle 检查，详细配置参考[checkstyle插件使用说明|http://maven.apache.org/plugins/maven- checkstyle-plugin/]，同样注意放置在reporting和build节点中的区别（所有报表类插件都要同样注意）：

{code:xml}

<plugin>

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

<artifactId>maven-checkstyle-plugin</artifactId>

<version>2.5</version>

</plugin>

{code}

加入 simian 的支持，simian是一个支持代码相似度检查的工具，目前有maven插件，也有checkstyle的插件。它不仅可以检查java，甚至可以支持文 本文件的检查。详细帮助信息参考[这里|http://www.redhillconsulting.com.au/products/simian /]。simian 的 maven插件在[这里|http://mojo.codehaus.org/simian-report-maven-plugin /introduction.html]

{code:xml}

<build>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>simian-maven-plugin</artifactId>

<version>1.6.1</version>

</plugin>

</plugins>

...

</build>

{code}

加入 jdepend 检查，详细配置参考[jdepend使用说明|http://mojo.codehaus.org/jdepend-maven-plugin/]，

{code:xml}

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>jdepend-maven-plugin</artifactId>

<version>2.0-beta-2</version>

</plugin>

{code}

加入 findbugz 检查，详细配置参考[findbugz使用说明|http://mojo.codehaus.org/findbugs-maven-plugin/usage.html]，

{code:xml}

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>findbugs-maven-plugin</artifactId>

<version>2.0.1</version>

</plugin>

{code}

加入javadoc生成，详细配置参考[javadoc usage|http://maven.apache.org/plugins/maven-javadoc-plugin/usage.html]

{code:xml}

<plugin>

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

<artifactId>maven-javadoc-plugin</artifactId>

<version>2.7</version>

<configuration>

...

</configuration>

</plugin>

{code}

加入 jxr 支持，JXR是一个生成java代码交叉引用和源代码的html格式的工具，详细配置信息参考[jxr usage|http://maven.apache.org/plugins/maven-jxr-plugin/]。注意，jxr没有必要在 build阶段运行。

{code:xml}

<reporting>

<plugins>

<plugin>

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

<artifactId>maven-jxr-plugin</artifactId>

<version>2.1</version>

</plugin>

</plugins>

</reporting>

{code}

加入 Cobertura 支持，它是一个代码覆盖率工具，可以用来评估具有相应测试的源代码的比率。详细帮助在[这里|http://mojo.codehaus.org /cobertura-maven-plugin/index.html]。另外一个功能相似的软件是[EMMA|http: //emma.sourceforge.net/samples.html],详细的帮助在[这里|http://mojo.codehaus.org /emma-maven-plugin/usage.html]。两个产品的比较文章在[这里|http://www.topcoder.com /tc?module=Static&d1=features&d2=030107]，个人倾向于都要用，因为给出的指标不一样，都有参 考作用。

{code:xml|title=Cobertura }

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>cobertura-maven-plugin</artifactId>

<version>2.4</version>

<configuration>

<check>

<branchRate>85</branchRate>

<lineRate>85</lineRate>

<haltOnFailure>true</haltOnFailure>

<totalBranchRate>85</totalBranchRate>

<totalLineRate>85</totalLineRate>

<packageLineRate>85</packageLineRate>

<packageBranchRate>85</packageBranchRate>

<regexes>

<regex>

<pattern>com.example.reallyimportant.\*</pattern>

<branchRate>90</branchRate>

<lineRate>80</lineRate>

</regex>

<regex>

<pattern>com.example.boringcode.\*</pattern>

<branchRate>40</branchRate>

<lineRate>30</lineRate>

</regex>

</regexes>

</check>

</configuration>

<executions>

<execution>

<goals>

<goal>clean</goal>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

{code}

{code:xml|title=EMMA}

<reporting>

...

<plugins>

...

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>emma-maven-plugin</artifactId>

<version>1.0-alpha-3-SNAPSHOT</version>

</plugin>

...

</plugins>

...

</reporting>

{code}

添加 javaNCSS 插件，它是一个java代码的度量工具，详细参考在[这里|http://mojo.codehaus.org/javancss-maven-plugin/]。

{code:xml}

<reporting>

<plugins>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>javancss-maven-plugin</artifactId>

<version>2.0-beta-2</version>

</plugin>

</plugins>

</reporting>

{code}

h4. profile相关

**. 问：profile能够设置为某个变量不存在的条件下激活？**

答：使用！前缀，请看示例：

{code:xml}

<activation>

<property>

<name>!environment.type</name>

</property>

</activation>

{code}

h4. 部署相关

**. 问：其他部署到服务器的方式和配置怎么配？**

答：本文摘自 [http://blog.csdn.net/zyxnetxz/archive/2009/05/18/4199348.aspx]{panel} \*Distribution Management\* 用于配置分发管理，配置相应的产品发布信息,主要用于发布，在执行mvn deploy后表示要发布的位置 \*# 配置到文件系统

{code:xml}

<distributionManagement>

<repository>

<id>proficio-repository<id>

<name>Proficio Repository<name>

<url>file://${basedir}/target/deploy<url>

<repository>

<distributionManagement>

{code}

\*# 使用ssh2配置

{code:xml}

<distributionManagement>

<repository>

<id>proficio-repository<id>

<name>Proficio Repository<name>

<url>scp://sshserver.yourcompany.com/deploy<url>

<repository>

<distributionManagement>

{code}

\*# 使用sftp配置

{code:xml}

<distributionManagement>

<repository>

<id>proficio-repositoryi<d>

<name>Proficio Repository<name>

<url>sftp://ftpserver.yourcompany.com/deploy<url>

<repository>

<distributionManagement>

{code}

\*# 使用外在的ssh配置编译扩展用于指定使用wagon外在ssh提供，用于提供你的文件到相应的远程服务器。

{code:xml}

<distributionManagement>

<repository>

<id>proficio-repository<id>

<name>Proficio Repository<name>

<url>scpexe://sshserver.yourcompany.com/deploy<url>

<repository>

<distributionManagement>

<build>

<extensions>

<extension>

<groupId>org.apache.maven.wagon<groupId>

<artifactId>wagon-ssh-external<artifactId>

<version>1.0-alpha-6<version>

<extension>

<extensions>

<build>

{code}

\*# 使用ftp配置

{code:xml}

<distributionManagement>

<repository>

<id>proficio-repository<id>

<name>Proficio Repository<name>

<url>ftp://ftpserver.yourcompany.com/deploy<url>

<repository>

<distributionManagement>

<build>

<extensions>

<extension>

<groupId>org.apache.maven.wagongroupId>

<artifactId>wagon-ftpartifactId>

<version>1.0-alpha-6version>

<extension>

<extensions>

<build>

{code}

{panel} h4. 插件配置

**. 问：我用maven输出site，如何设置输出为utf8编码？**

答： 配置site插件的编码设置

{code:xml}

...

<plugin>

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

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

<version>2.0-beta-6</version>

<configuration>

<outputEncoding>UTF-8</outputEncoding>

</configuration>

</plugin>

...

{code}

FROM：http://www.cnblogs.com/BigTall/archive/2011/03/23/1993253.html