全局配置: ${M2\_HOME}/conf/settings.xml

用户配置: ${user.home}/.m2/settings.xml

note：用户配置优先于全局配置。${user.home} 和和所有其他系统属性只能在3.0+版本上使用。请注意windows和Linux使用变量的区别。

1. settings.xml详解

声明规范

<?xml version="1.0" encoding="UTF-8"?>

<settings 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/settings-1.0.0.xsd">

localRepository

<!-- 本地仓库的路径。默认值为${user.home}/.m2/repository。 -->

<localRepository>usr/local/maven</localRepository>

interactiveMode

<!--Maven是否需要和用户交互以获得输入。如果Maven需要和用户交互以获得输入，则设置成true，反之则应为false。默认为true。-->

<interactiveMode>true</interactiveMode>

usePluginRegistry

<!--Maven是否需要使用plugin-registry.xml文件来管理插件版本。如果需要让Maven使用文件${user.home}/.m2/plugin-registry.xml来管理插件版本，则设为true。默认为false。-->

<usePluginRegistry>false</usePluginRegistry>

offline

<!--表示Maven是否需要在离线模式下运行。如果构建系统需要在离线模式下运行，则为true，默认为false。当由于网络设置原因或者安全因素，构建服务器不能连接远程仓库的时候，该配置就十分有用。 -->

<offline>false</offline>

pluginGroups

[复制代码](javascript:void(0);)

<!--当插件的组织Id（groupId）没有显式提供时，供搜寻插件组织Id（groupId）的列表。该元素包含一个pluginGroup元素列表，每个子元素包含了一个组织Id（groupId）。当我们使用某个插件，并且没有在命令行为其提供组织Id（groupId）的时候，Maven就会使用该列表。默认情况下该列表包含了org.apache.maven.plugins和org.codehaus.mojo -->

<pluginGroups>

<!--plugin的组织Id（groupId） -->

<pluginGroup>org.codehaus.mojo</pluginGroup>

</pluginGroups>

[复制代码](javascript:void(0);)

proxies

[复制代码](javascript:void(0);)

<!--用来配置不同的代理，多代理profiles 可以应对笔记本或移动设备的工作环境：通过简单的设置profile id就可以很容易的更换整个代理配置。 -->

<proxies>

<!--代理元素包含配置代理时需要的信息-->

<proxy>

<!--代理的唯一定义符，用来区分不同的代理元素。-->

<id>myproxy</id>

<!--该代理是否是激活的那个。true则激活代理。当我们声明了一组代理，而某个时候只需要激活一个代理的时候，该元素就可以派上用处。 -->

<active>true</active>

<!--代理的协议。 协议://主机名:端口，分隔成离散的元素以方便配置。-->

<protocol>http</protocol>

<!--代理的主机名。协议://主机名:端口，分隔成离散的元素以方便配置。 -->

<host>proxy.somewhere.com</host>

<!--代理的端口。协议://主机名:端口，分隔成离散的元素以方便配置。 -->

<port>8080</port>

<!--代理的用户名，用户名和密码表示代理服务器认证的登录名和密码。 -->

<username>proxyuser</username>

<!--代理的密码，用户名和密码表示代理服务器认证的登录名和密码。 -->

<password>somepassword</password>

<!--不该被代理的主机名列表。该列表的分隔符由代理服务器指定；例子中使用了竖线分隔符，使用逗号分隔也很常见。-->

<nonProxyHosts>\*.google.com|ibiblio.org</nonProxyHosts>

</proxy>

</proxies>

[复制代码](javascript:void(0);)

servers

[复制代码](javascript:void(0);)

<!--配置服务端的一些设置。一些设置如安全证书不应该和pom.xml一起分发。这种类型的信息应该存在于构建服务器上的settings.xml文件中。-->

<servers>

<!--服务器元素包含配置服务器时需要的信息 -->

<server>

<!--这是server的id（注意不是用户登陆的id），该id与distributionManagement中repository元素的id相匹配。-->

<id>server001</id>

<!--鉴权用户名。鉴权用户名和鉴权密码表示服务器认证所需要的登录名和密码。 -->

<username>my\_login</username>

<!--鉴权密码 。鉴权用户名和鉴权密码表示服务器认证所需要的登录名和密码。密码加密功能已被添加到2.1.0 +。详情请访问密码加密页面-->

<password>my\_password</password>

<!--鉴权时使用的私钥位置。和前两个元素类似，私钥位置和私钥密码指定了一个私钥的路径（默认是${user.home}/.ssh/id\_dsa）以及如果需要的话，一个密语。将来passphrase和password元素可能会被提取到外部，但目前它们必须在settings.xml文件以纯文本的形式声明。 -->

<privateKey>${usr.home}/.ssh/id\_dsa</privateKey>

<!--鉴权时使用的私钥密码。-->

<passphrase>some\_passphrase</passphrase>

<!--文件被创建时的权限。如果在部署的时候会创建一个仓库文件或者目录，这时候就可以使用权限（permission）。这两个元素合法的值是一个三位数字，其对应了unix文件系统的权限，如664，或者775。 -->

<filePermissions>664</filePermissions>

<!--目录被创建时的权限。 -->

<directoryPermissions>775</directoryPermissions>

</server>

</servers>

[复制代码](javascript:void(0);)

mirrors

[复制代码](javascript:void(0);)

<!--为仓库列表配置的下载镜像列表。高级设置请参阅镜像设置页面 -->

<mirrors>

<!--给定仓库的下载镜像。 -->

<mirror>

<!--该镜像的唯一标识符。id用来区分不同的mirror元素。 -->

<id>planetmirror.com</id>

<!--镜像名称 -->

<name>PlanetMirror Australia</name>

<!--该镜像的URL。构建系统会优先考虑使用该URL，而非使用默认的服务器URL。 -->

<url>http://downloads.planetmirror.com/pub/maven2</url>

<!--被镜像的服务器的id。例如，如果我们要设置了一个Maven中央仓库（http://repo.maven.apache.org/maven2/）的镜像，就需要将该元素设置成central。这必须和中央仓库的id central完全一致。-->

<mirrorOf>central</mirrorOf>

</mirror>

</mirrors>

[复制代码](javascript:void(0);)

profiles

[复制代码](javascript:void(0);)

<!--根据环境参数来调整构建配置的列表。settings.xml中的profile元素是pom.xml中profile元素的裁剪版本。它包含了id，activation, repositories, pluginRepositories和 properties元素。这里的profile元素只包含这五个子元素是因为这里只关心构建系统这个整体（这正是settings.xml文件的角色定位），而非单独的项目对象模型设置。如果一个settings中的profile被激活，它的值会覆盖任何其它定义在POM中或者profile.xml中的带有相同id的profile。 -->

<profiles>

<!--根据环境参数来调整的构件的配置-->

<profile>

<!--该配置的唯一标识符。 -->

<id>test</id>

[复制代码](javascript:void(0);)

Activation

[复制代码](javascript:void(0);)

<!--自动触发profile的条件逻辑。Activation是profile的开启钥匙。如POM中的profile一样，profile的力量来自于它能够在某些特定的环境中自动使用某些特定的值；这些环境通过activation元素指定。activation元素并不是激活profile的唯一方式。settings.xml文件中的activeProfile元素可以包含profile的id。profile也可以通过在命令行，使用-P标记和逗号分隔的列表来显式的激活（如，-P test）。-->

<activation>

<!--profile默认是否激活的标识-->

<activeByDefault>false</activeByDefault>

<!--当匹配的jdk被检测到，profile被激活。例如，1.4激活JDK1.4，1.4.0\_2，而!1.4激活所有版本不是以1.4开头的JDK。-->

<jdk>1.5</jdk>

<!--当匹配的操作系统属性被检测到，profile被激活。os元素可以定义一些操作系统相关的属性。-->

<os>

<!--激活profile的操作系统的名字 -->

<name>Windows XP</name>

<!--激活profile的操作系统所属家族(如 'windows') -->

<family>Windows</family>

<!--激活profile的操作系统体系结构 -->

<arch>x86</arch>

<!--激活profile的操作系统版本-->

<version>5.1.2600</version>

</os>

<!--如果Maven检测到某一个属性（其值可以在POM中通过${name}引用），其拥有对应的name = 值，Profile就会被激活。如果值字段是空的，那么存在属性名称字段就会激活profile，否则按区分大小写方式匹配属性值字段-->

<property>

<!--激活profile的属性的名称-->

<name>mavenVersion</name>

<!--激活profile的属性的值 -->

<value>2.0.3</value>

</property>

<!--提供一个文件名，通过检测该文件的存在或不存在来激活profile。missing检查文件是否存在，如果不存在则激活profile。另一方面，exists则会检查文件是否存在，如果存在则激活profile。-->

<file>

<!--如果指定的文件存在，则激活profile。 -->

<exists>${basedir}/file2.properties</exists>

<!--如果指定的文件不存在，则激活profile。-->

<missing>${basedir}/file1.properties</missing>

</file>

</activation>

[复制代码](javascript:void(0);)

Properties

[复制代码](javascript:void(0);)

<!--对应profile的扩展属性列表。Maven属性和Ant中的属性一样，可以用来存放一些值。这些值可以在POM中的任何地方使用标记${X}来使用，这里X是指属性的名称。属性有五种不同的形式，并且都能在settings.xml文件中访问。

1. env.X: 在一个变量前加上"env."的前缀，会返回一个shell环境变量。例如,"env.PATH"指代了$path环境变量（在Windows上是%PATH%）。

2. project.x：指代了POM中对应的元素值。例如: <project><version>1.0</version></project>通过${project.version}获得version的值。

3. settings.x: 指代了settings.xml中对应元素的值。例如：<settings><offline>false</offline></settings>通过 ${settings.offline}获得offline的值。

4. Java System Properties: 所有可通过java.lang.System.getProperties()访问的属性都能在POM中使用该形式访问，例如 ${java.home}。

5. x: 在<properties/>元素中，或者外部文件中设置，以${someVar}的形式使用。 -->

<properties>

<user.install>${user.home}/our-project</user.install>

</properties>

note：如果该profile被激活，则可以再POM中使用${user.install}。

[复制代码](javascript:void(0);)

Repositories

[复制代码](javascript:void(0);)

<!--远程仓库列表，它是Maven用来填充构建系统本地仓库所使用的一组远程项目。 -->

<repositories>

<!--包含需要连接到远程仓库的信息 -->

<repository>

<!--远程仓库唯一标识-->

<id>codehausSnapshots</id>

<!--远程仓库名称 -->

<name>Codehaus Snapshots</name>

<!--如何处理远程仓库里发布版本的下载-->

<releases>

<!--true或者false表示该仓库是否为下载某种类型构件（发布版，快照版）开启。 -->

<enabled>false</enabled>

<!--该元素指定更新发生的频率。Maven会比较本地POM和远程POM的时间戳。这里的选项是：always（一直），daily（默认，每日），interval：X（这里X是以分钟为单位的时间间隔），或者never（从不）。 -->

<updatePolicy>always</updatePolicy>

<!--当Maven验证构件校验文件失败时该怎么做-ignore（忽略），fail（失败），或者warn（警告）。-->

<checksumPolicy>warn</checksumPolicy>

</releases>

<!--如何处理远程仓库里快照版本的下载。有了releases和snapshots这两组配置，POM就可以在每个单独的仓库中，为每种类型的构件采取不同的策略。例如，可能有人会决定只为开发目的开启对快照版本下载的支持。参见repositories/repository/releases元素-->

<snapshots>

<enabled/><updatePolicy/><checksumPolicy/>

</snapshots>

<!--远程仓库URL，按protocol://hostname/path形式 -->

<url>http://snapshots.maven.codehaus.org/maven2</url>

<!--用于定位和排序构件的仓库布局类型-可以是default（默认）或者legacy（遗留）。Maven 2为其仓库提供了一个默认的布局；然而，Maven 1.x有一种不同的布局。我们可以使用该元素指定布局是default（默认）还是legacy（遗留）。 -->

<layout>default</layout>

</repository>

</repositories>

<!--发现插件的远程仓库列表。仓库是两种主要构件的家。第一种构件被用作其它构件的依赖。这是中央仓库中存储的大部分构件类型。另外一种构件类型是插件。Maven插件是一种特殊类型的构件。由于这个原因，插件仓库独立于其它仓库。pluginRepositories元素的结构和repositories元素的结构类似。每个pluginRepository元素指定一个Maven可以用来寻找新插件的远程地址。-->

<pluginRepositories>

<!--包含需要连接到远程插件仓库的信息.参见profiles/profile/repositories/repository元素的说明-->

<pluginRepository>

<releases>

<enabled/><updatePolicy/><checksumPolicy/>

</releases>

<snapshots>

<enabled/><updatePolicy/><checksumPolicy/>

</snapshots>

<id/><name/><url/><layout/>

</pluginRepository>

</pluginRepositories>

</profile>

</profiles>

[复制代码](javascript:void(0);)

activeProfiles

[复制代码](javascript:void(0);)

<!--手动激活profiles的列表，按照profile被应用的顺序定义activeProfile。 该元素包含了一组activeProfile元素，每个activeProfile都含有一个profile id。任何在activeProfile中定义的profile id，不论环境设置如何，其对应的

profile都会被激活。如果没有匹配的profile，则什么都不会发生。例如，env-test是一个activeProfile，则在pom.xml（或者profile.xml）中对应id的profile会被激活。如果运行过程中找不到这样一个profile，Maven则会像往常一样运行。 -->

<activeProfiles>

<!-- -->

<activeProfile>env-test</activeProfile>

</activeProfiles>

</settings>

maven有三套生命周期

1.clean       清理项目

2.default     构建项目

3.site           建立项目站点

每套生命周期都包含了一些阶段，这些阶段是有序的，后面的阶段依赖前面的阶段，

以clean生命周期为例，它包括了pre-clean，clean和post-clean三个阶段，当我们调用pre-clean的时候， 只会执行pre-clean阶段；

当调用clean的时候，pre-clean和clean阶段会以顺序执行；当调用post-clean的时候，pre-clean，clean和post-clean三个阶段会以顺序执行。

这三套生命周期是相互独立的，可以仅仅调用clean声明周期的某个阶段， 或者调用default生命周期的某个阶段，而不会对其他生命周期产生任何影响。

clean生命周期共包含了三个阶段：  
            pre-clean  执行一些需要在clean之前完成的工作  
            clean  移除所有上一次构建生成的文件  
            post-clean  执行一些需要在clean之后立刻完成的工作

default生命周期包含的阶段：  
             validate  
             initialize  
             generate-sources  
             process-sources  
             generate-resources  
             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  
             generate-test-sources    
             process-test-sources   
             generate-test-resources  
             process-test-resources     复制并处理资源文件，至目标测试目录。  
             test-compile     编译测试源代码。  
             process-test-classes   
             test     使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。  
             prepare-package   
             package     接受编译好的代码，打包成可发布的格式，如 JAR 。  
             pre-integration-test  
             integration-test  
             post-integration-test   
             verify   
             install     将包安装至本地仓库，以让其它项目依赖。  
             deploy     将最终的包复制到远程的仓库，以让其它开发人员与项目共享。

site生命周期的各个阶段：  
             pre-site     执行一些需要在生成站点文档之前完成的工作  
             site    生成项目的站点文档  
             post-site     执行一些需要在生成站点文档之后完成的工作，并且为部署做准备  
             site-deploy     将生成的站点文档部署到特定的服务器上

组合阶段执行：

             mvn clean     调用的clean生命周期的clean阶段，实际执行的是pre-clean和clean阶段。

             mvn test         执行的是default生命周期的test阶段， 实际执行的是validate到test阶段。

             mvn clean  package      clean生命周期的pre-clean，clean阶段和 default生命周期的从validate到package阶段。

1. pox.xml详解
2. 指定本地Repository位置

在安装目录的conf/setting.xml中的<localRepository>节点

1. 参数

dependency:copy-dependencies -DoutputDirectory=lib

下载依赖的时候，同时下载src,doc

D:\mvn> mvn -X dependency:sources -DdownloadSources=true -DdownloadJavadocs=true dependency:resolve -Dclassifier=java

doc

mvn archetype:generate