## Maven

简介：

构建：编译，运行单元测试，生成文档，打包，部署

优点：作为构建工具，可以帮助我们自动化构建，抽象构建过程，提供构建任务实现；跨平台，对外提供一致的操作接口。

不仅仅是构建工具，还是一个依赖管理工具和项目信息管理工具。提供了中央仓库，帮助我们自动下载构件。

### maven与极限编程(XP)

### maven安装与配置

#### windows环境

1.需要先安装JDK环境

2.下载解压maven压缩包

3.配置maven环境变量

cmd命令执行顺序：windows首先在当前目录寻找可执行脚本，没有就到Path中定义的路径找

4.检查环境变量配置 echo %MAVEN\_HOME%

#### linux环境

alt+shift+x,m

指令：

Mvn clean 清空target目录

Mvn compile 编译 生成class

Mvn package 打包 生成jar文件

Mvn install 安装 将生成的jar文件安装到本地仓库,这样其他maven项目就可以依赖使用

Mvn deploy 发布 将jar包发布到私服仓库

install -DskipTests 跳过测试

Mvn –Dmaven.test.skip=true -X 打包时跳过测试用例，-X输出打包信息

Maven 可配置多套环境（develop,test,stage）开发环境，测试环境，生产环境

Mvn –Ptest 以test的配置打包

Mvn versions:set –DnewVersion=0.5.0 -DgenerateBackupPoms=false 统一修改pom文件中版本信息

Echo %JAVA\_HOME% 输出Java环境变量配置

Java –version

Mvn –v 检查安装的maven环境

#### Maven目录

* bin:包含mvn运行脚本
* boot:只包含一个plexus-classworlds-\*\*\*.jar，是一个类加载器框架，比java默认的类加载器提供了更丰富的语法以方便配置。Maven使用该框架加载自己的类库
* conf: settings.xml.全局定制Maven行为，**一般将该文件复制到~/.m2/目录下（建义最佳实践）**，在用户范围定制maven行为
* lib:包含maven运行时需要的java类库

配置maven代理：

Maven中央库：<http://repo1.maven.org/>

如果访问不了



Active：表示激活代理

如果代理需要认证就配置用户名密码,<nonProxyHosts>配置不需要代理的主机名,|分割多个值

Eclipse重要组件：

插件地址<http://m2eclipse.sonatype.org/sites/m2e-extras>

* Maven SCM handler for Subclipse(Optional)：帮助从SVN服务器签出Maven项目，需要先集成SVN(<http://subclipse.tigris.org/>)
* Maven SCM Integration(Optional):Maven与SCM集成，利用各种SCM工具(如SVN)实现Maven项目的签出和具体化等操作
* Project configurators for commonly used maven plugins(temporary)：临时组件，支持一些Maven插件与Eclipse的集成，建议安装。

### Maven入门

#### Pom

groupId：定义项目属于哪个组，如com.mycompany.myapp

artifactId：定义当前Maven项目在组中唯一的ID

version：表示版本，1.0-SNAPSHOT，表示快照版本（不稳定版本）。1.1等等

一般项目中**Java类的包都应该基于项目的groupId和artifactId**，这样结构更加清晰，符合逻辑

mvn clean compile ：清空target/目录，编译项目代码

运行测试：

mvn clean test

包含了 clean:clean，resources:resources，compiler:compile，resources:testResources，compiler:testCompile

及主资源处理，主代码编译，测试资源处理，测试代码编译等工作(maven生命周期的特性)

#### 可运行jar包

需要maven-shade-plugin插件:

….

#### Archetype生成项目骨架

mvn archetype:generate

### Maven坐标体系

* GroupId：定义当前Maven项目隶属的实际项目（一个实际项目可能对应多个maven项目，如：spring项目下，有spring-core,spring-web等），groupId不应该对应项目隶属的组织或公司，因为组织下会有多个项目，artifactId只能定义Maven项目模块，导致实际项目这一层很难定义。

如： groupId: com.sinoair.hlink.acd 定义到公司组织下具体的实际项目

* artifactId：定义实际项目中的一个Maven项目(模块)，通常使用实际项目名作为前缀，如： acd-façade,acd-service等

默认maven打包时会命名为 acd-façade-1.0.0.jar

* version：定义maven项目当前所处的版本
* packaging：定义打包方式，默认使用jar包
* classifier：定义构建输出的附属构件，如Java文档和源代码。不能直接定义

依赖配置中：添加<scope>test</scope>表示只在测试用使用

<dependency>

<groupId>com.sinoair.hlink</groupId>

<artifactId>hlp-ssp-facade</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>test</scope>

</dependency>

测试代码编写  
 @Before 测试代码运行前执行的方法

@Test 测试代码执行的方法

@After 测试代码运行后执行的方法

Mvn clean test 执行测试代码命令，Maven会编译主代码和测试代码，并执行测试

Mvn clean install 自动下载依赖构件，执行编译，测试，打包等工作，生成jar包并安装到本地仓库中，这样，该模块就能供其他Maven项目使用了

### 依赖配置

* groupId,artifactId,version：依赖的基本坐标
* type：依赖的类型，对应坐标定义的packaging。默认为jar
* scope：依赖范围，控制编译classpath,测试classpath,运行classpath关系。
  + compile：编译依赖范围。默认值，对编译，测试，运行都有效
  + test：测试依赖范围。只对测试有效，在编译主代码和运行项目时无法使用，如，junit包
  + provided：已提供范围。对编译和测试有效，运行时无效，如servlet-api
  + runtime：运行时范围。测试和运行有效，编译时无效，如jdbc驱动包
  + system：系统范围。与provided一致，依赖与本地绑定

如：

<dependency>

<groupId>com.sinoair.hlink</groupId>

<artifactId>hlp-ssp-facade</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>system</scope>

<systemPath>${java.home}/lib/rt.jar</systemPath>

</dependency>

* optional：依赖是否可选
* exclusions：排除传递性依赖

### 传递性依赖

A依赖B，B依赖C， A就自然依赖C

## 最佳实践

### 排除依赖

如 上ABC的例子，假如C是SNAPSHOT版本，不稳定，这时我们就可以排除B对C的依赖，然后自己再配置一个对C的稳定版本的依赖

如：

<dependency>

<groupId>com.sinoair.hlink</groupId>

<artifactId>B</artifactId>

<version>1.0.0</version>

<!—剔除不稳定版本的依赖-->

<exclusions>

<exclusion>

<groupId> com.sinoair.hlink </groupId>

<artifactId>C</artifactId>

<version>1.0.0-SNAPSHOT </version>

</exclusion>

</exclusions>

</dependency>

<!—自定义对稳定版本的依赖-->

<dependency>

<groupId>com.sinoair.hlink</groupId>

<artifactId>C</artifactId>

<version>1.0.0</version>

</dependency>

### 归类依赖

<!—统一定义版本信息然后引用-->

<properties>

<junit.version>4.12</junit.version>

<properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

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

<scope>test</scope>

</dependency>

<dependencies>

### 优化依赖

Maven会自动解析所有项目的直接依赖和传递性依赖，并且根据规则正确判断每个依赖的范围，对于一些依赖冲突，也能进行调节。最后得到的依赖叫做 已解析依赖。

mvn dependency:list 查看当前项目的已解析依赖。

当前pom中的依赖为顶层依赖

这些顶层依赖的依赖叫做第二层依赖，依次类推第三，第四层依赖

这些依赖就形成一个依赖树

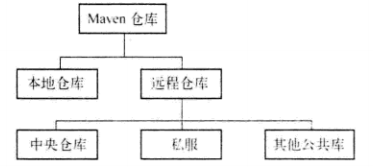
mvn dependency:tree 查看依赖树

mvn dependency:analyze 分析依赖

可以找出一些 用到但没显示声明的依赖和 显示声明但没用到的依赖

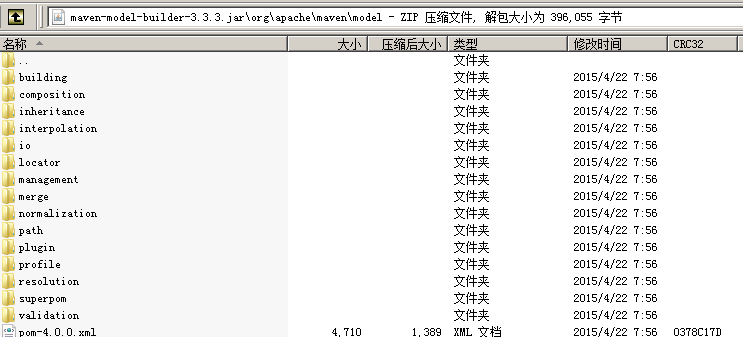
## Maven仓库

仓库分本地仓库和远程仓库。当Maven根据坐标寻找构件时，会首先查看本地仓库，如果本地仓库没有，就会去远程仓库查找，找到后下载到本地仓库再使用。



### 本地仓库

### 中央仓库





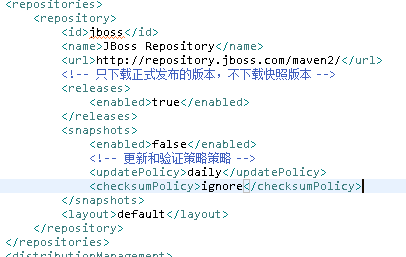
Maven默认配置的远程中央仓库地址。该pom文件是所有maven项目都会继承的超级POM

### 私服

特殊的远程仓库，架设在局域网内的仓库服务

### 远程仓库配置

很多情况下，默认的中央仓库无法满足项目的需求，需要的构件可能存在另一个远程仓库，如JBoss Maven仓库。可在POM中进行配置

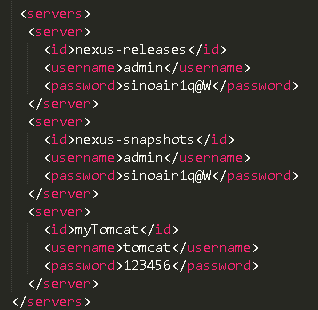


任何一个仓库声明的id必须是唯一的，maven自带的中央仓库使用的id为central，如果其他仓库声明也使用这个id，就会覆盖中央仓库的配置

### 远程仓库的认证

大部分远程仓库无须认证就可以访问，像私服就可以配置一组用户名及密码对访问进行控制。

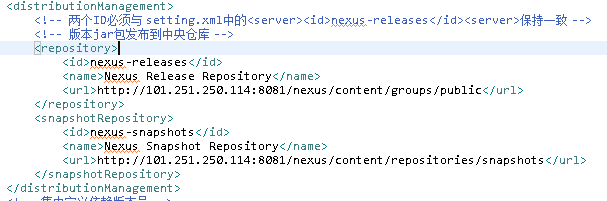
认证信息必须配置在settings.xml文件中



注意id必须与项目repository中id配置一致



本地jar包部署到私服配置



Mvn clean deploy 该命令将项目构件部署到对应的远程仓库

## 从仓库解析依赖的机制

## 镜像

如果仓库X可以提供仓库Y存储的所有内容，那么就可以认为X是Y的一个镜像。换句话说，任何一个可以从仓库Y获得的构件，都能够从它的镜像中获取。

如： 配置中央库的镜像库  


任何对中央仓库的请求都会转至该镜像。可以同样配置其他仓库的镜像。

由于私服可以代理任何外部的公共仓库，因此，对与公司内部maven用户来说，使用一个私服地址就等于使用了所有需要的外部仓库,这样可以将配置集中到私服，从而简化Maven本身的配置

如上配置私服为所有仓库的镜像，\*代表匹配所有远程仓库。