# 1、maven的基本命令

默认远程仓库：

默认远程仓库 maven central 其配置在

maven-model-builder-3.2.1.jar\org\apache\maven\model\pom-4.0.0.xml 位置

本地仓库位置：

本地仓库位置默认在 ~/.m2/respository 下

要修改 ${M2\_HOME}/conf/settings.xml  来指定仓库目录

<!-- 指定本地仓库目录-->

 <localRepository>G:\.m2\repository</localRepository>

maven 核心功能总结：

1. maven 核心作用是编译、测试、打包。
2. 根目录下的pom.xml 文件设置分组ID与artifactId。
3. maven 基于约定的方式从项目中获取源码与资源文件进行编译打包。
4. 对于项目所依懒的组件与会本地仓库引用，如果本地仓库不存在则会从中央仓库下载。

## 1.1 基本命令

mvn compile 编译

mvn clean: 清空，每次更新项目的内容，Maven是不会自动编译更新的，所以target目录下的内容是缓存的，每次部署前都需要把最新数据更新到target目录下。

当运行mvn clean后，target下的内容都会被删除，然后生成最新的数据。

mvn test：运行测试，该命令执行后会去运行所有的单元测试，如果想要单独运行某个测试类里的单元测试，可用如下命令：mvn -Dtest=TestApp1 test

mvn package：此命令包含一系列过程：validate-->compile-->test-->package

Maven根据pom文件里packaging的配置，决定是生成jar文件还是war文件，并放到target目录下，mvn –Dtest package 只打包不测试。

mvn clean install：该命令会把你的项目打包并放到本地仓库，别人就可以引用了。

此命令包含以下过程：validate-->compile-->test-->package-->integration-test-->verify-->install。

mvn deploy：将jar上传到私服

## 1.2 使用

### 1、compile

maven 采用了约定的方式从指项目结构中获取源码与资源文件进行编译打包。

* 1. 主源码文件：${project}/src/main/java
  2. 主资源文件：${project}/src/main/resources
  3. 测试源码文件：${project}/src/test/java
  4. 测试资源文件：${project}/src/test/resources

将java 文件移至 src/main/java 目录，重新执行编译.

mv src/hello.java /src/main/java/hello.java

mvn compile;

### 2、mvn test

在没引入junit的情况下，在test文件下编写测试用例，所有的测试类与测试方法使用test开头，则maven会自动识别并执行

package com.test.tuling;

public class TestHello{

        public void sayHelloTest(){

                System.out.println("run test .....");

        }

}

执行命令：mvn test

如果引入了junit，则需要对测试方法加上@Test注解

### 3、 mvn deploy 上传私服

**1、通过命令上传**

mvn deploy:deploy-file -DgroupId=xxx.xxx.com -DartifactId=xx -Dversion=1.3.3 -Dpackaging=jar  -Dfile=E:\emay\eucp-sms-sdk-1.3.3.jar -Durl=http://ip:port/repository/maven-releases -DrepositoryId=maven-releases

-DgroupId=jar包的组名（项目groupId）

-DartifactId=jar包名称

-Dversion=jar包版本

-Dfile=jar包绝对路径

-Durl=http://ip:port/repository/maven-releases   nexus服务器上仓库的地址

-DrepositoryId=yang nexus服务器上仓库的名称

**2、通过代码开发工具上传**

在自己的maven的setting.xml配置server节点

<server>

<id>releases</id>

<username>admin</username>

<password>admin123</password>

</server>

<server>

<id>snapshots</id>

<username>admin</username>

<password>admin123</password>

</server>

项目下pom.xml中添加如下内容

<!-- 项目发布管理 -->

<distributionManagement>

<repository>

<id>releases</id>

<name>User Project Release</name>

<url>http://ip:port/repository/releases/</url>

</repository>

<snapshotRepository>

<id>snapshots</id>

<name>User Project SNAPSHOTS</name>

<url>http://ip:port/repository/snapshots/</url>

</snapshotRepository>

< /distributionManagement>

**snapshot快照库和release发布库的区别和作用**

在使用maven过程中，我们在开发阶段经常性的会有很多公共库处于不稳定状态，随时需要修改并发布，可能一天就要发布一次，遇到bug时，甚至一天要发布N次。我们知道，maven的依赖管理是基于版本管理的，对于发布状态的artifact，如果版本号相同，即使我们内部的镜像服务器上的组件比本地新，maven也不会主动下载的。如果我们在开发阶段都是基于正式发布版本来做依赖管理，那么遇到这个问题，就需要升级组件的版本号，可这样就明显不符合要求和实际情况了。但是，如果是基于快照版本，那么问题就自热而然的解决了，而maven已经为我们准备好了这一切。

maven中的仓库分为两种，snapshot快照仓库和release发布仓库。snapshot快照仓库用于保存开发过程中的不稳定版本，release正式仓库则是用来保存稳定的发行版本。定义一个组件/模块为快照版本，只需要在pom文件中在该模块的版本号后加上-SNAPSHOT即可(注意这里必须是大写)，如下：



maven2会根据模块的版本号(pom文件中的version)中是否带有-SNAPSHOT来判断是快照版本还是正式版本。如果是快照版本，那么在mvn deploy时会自动发布到快照版本库中，而使用快照版本的模块，在不更改版本号的情况下，直接编译打包时，maven会自动从镜像服务器上下载最新的快照版本。如果是正式发布版本（**父子项目的版本号中都不能有SNAPSHOT**），那么在mvn deploy时会自动发布到正式版本库中，而使用正式版本的模块，在不更改版本号的情况下，编译打包时如果本地已经存在该版本的模块则不会主动去镜像服务器上下载。

所以，我们在开发阶段，可以将公用库的版本设置为快照版本，而被依赖组件则引用快照版本进行开发，在公用库的快照版本更新后，我们也不需要修改pom文件提示版本号来下载新的版本，直接mvn执行相关编译、打包命令即可重新下载最新的快照库了，从而也方便了我们进行开发。

# 2 、maven的核心配置

项目依懒(内部、外部)

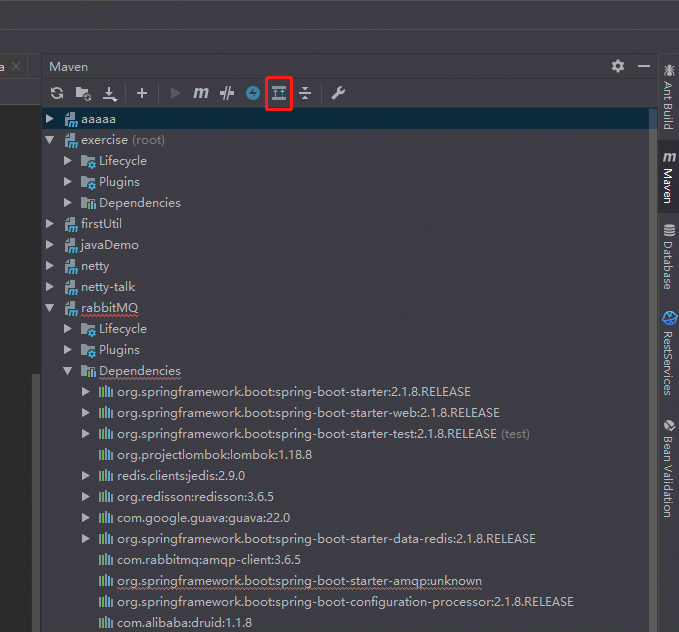
项目聚合与继承

项目构建配置

## 2.1 项目依赖

项目依赖是指maven 通过依赖传播、依赖优先原则、可选依赖、排除依赖、依赖范围等特性来管理项目ClassPath。

通过此操作查看maven依赖树



### 1.1 依赖传播特性

我们的项目通常需要依赖第三方组件，而第三方组件又会依赖其它组件遇到这种情况Maven会将依赖网络中的所有节点都会加入ClassPath当中，这就是Maven的依赖传播特性。

<!-- 添加spring web mvc演示 -->

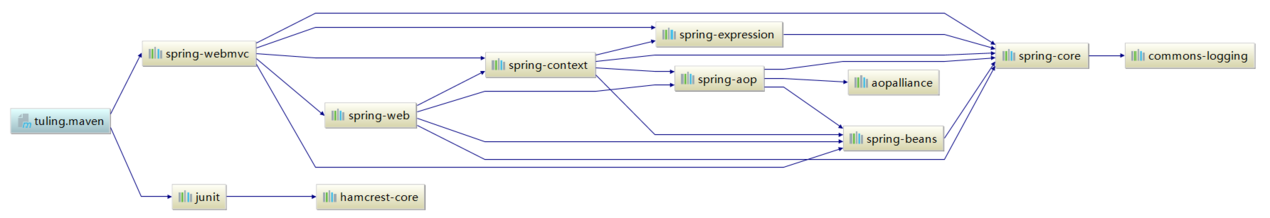
<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.4.RELEASE</version>

</dependency>



项目直接依赖了spring-webmvc 叫直接依赖，而对commons-logging 依赖是通过webmvc传递的所以叫间接依赖。

### 1.2 依赖优先原则

基于依赖传播特性，导致整个依赖网络会很复杂，难免会出现相同组件不同版本的情况。Maven此时会基于依赖优先原则选择其中一个版本。

第一原则：最短路径优先。

第二原则：相同路径下配置在前的优先。

* + 第一原则演示

<!-- 直接添加commons-logging -->

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

上述例子中commons-logging 通过spring-webmvc 依赖了1.1.3，而项目中直接依赖了1.2，基于最短路径原则项目最终引入的是1.2 版本。

* + 第二原则演示：

步骤：

1. 添加一个新工程Project B
2. 配置Project B 依赖 spring-web.3.2.9.RELEASE
3. 当前工程直接依赖 Project B

配置完之后，当前工程 project A 有两条路径可以依赖 spring-web,选择哪一条 就取决于 对 webmvc 和 Project B的配置先后顺序。

Project A==> spring-webmvc.4.0.0.RELEASE ==> spring-web.4.0.0.RELEASE

Project A==> Project B 1.0.SNAPSHOT ==>spring-web.3.2.9.RELEASE

注意：在同一pom文件，第二原则不在适应。如下配置，最终引用的是1.2 版本，而不是配置在前面的1.1.1版本.在同一pom文件中相同的依赖会选择配置在后面的

<!-- 在1.2 之前添加 commons-logging -->

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.2</version>

</dependency>

### 1.3 可选依赖

可选依赖表示这个依赖不是必须的。通过在 <dependency> 添  <optional>true</optional> 表示，默认是不可选的。可选依赖不会被传递。当别的模块依赖此模块时此依赖不会被引入

### 1.4 排除依赖

即排除指定的间接依赖。通过配置 <exclusions> 配置排除指定组件，在build标签中配置。

<!-- 排除指定项目 -->

<exclusions>

    <exclusion>

       <groupId>org.springframework</groupId>

       <artifactId>spring-web</artifactId>

    </exclusion>

</exclusions>

### 1.5 依赖范围

像junit 这个组件 我们只有在运行测试用例的时候去要用到，这就没有必要在打包的时候把junit.jar 包过构建进去，可以通过Mave 的依赖范围配置<scope>来达到这种目的。maven 总共支持以下四种依赖范围：

**compile(默认)**: 编译范围，编译和打包都会依赖。

**provided**：提供范围，编译时依赖，但不会打包进去。如：servlet-api.jar，此jar包运行时由容器提供（tomcat）

**runtime**：运行时范围，打包时依赖，编译不会。如：mysql-connector-java.jar

**test**：测试范围，编译运行测试用例依赖，不会打包进去。如：junit.jar

**system**：表示由系统中CLASSPATH指定。编译时依赖，不会打包进去。配合<systemPath> 一起使用。示例：java.home下的tool.jar

system 除了可以用于引入系统classpath 中包，也可以用于引入系统非maven 收录的第三方Jar，做法是将第三方Jar放置在项目的lib目录下，然后配置相对路径，但因system 不会打包进去所以需要配合 maven-dependency-plugin 插件配合使用。当然推荐大家还是通过 将第三方Jar手动install 到仓库。

<!-- system 的通常使用方式-->

<dependency>

    <groupId>com.sun</groupId>

    <artifactId>tools</artifactId>

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

    <scope>system</scope>

    <optional>true</optional>

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

</dependency>

<!-- system 另外使用方式 ,将工程内的jar直接引入 -->

<dependency>

    <groupId>jsr</groupId>

    <artifactId>jsr</artifactId>

    <version>3.5</version>

    <scope>system</scope>

    <optional>true</optional>

    <systemPath>${basedir}/lib/jsr305.jar</systemPath>

</dependency>

<!-- 通过插件 将system 的jar 打包进去。 -->

<plugin>

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

<artifactId>maven-dependency-plugin</artifactId>

<version>2.10</version>

<executions>

<execution>

<id>copy-dependencies</id>

<phase>compile</phase>

<goals>

<goal>copy-dependencies</goal>

</goals>

<configuration>

<outputDirectory>${project.build.directory}/${project.build.finalName}/WEB-INF/lib</outputDirectory>

<includeScope>system</includeScope>

<excludeGroupIds>com.sun</excludeGroupIds>

</configuration>

</execution>

</executions>

</plugin>

#手动加入本地仓库

mvn install:install-file -Dfile=abc\_client\_v1.20.jar -DgroupId=tuling -DartifactId=tuling-client -Dversion=1.20 -Dpackaging=jar

## 2.2 项目的聚合与继承

### 2.1 聚合

是指将多个模块整合在一起，统一构建，避免一个一个的构建。聚合需要个父工程，然后使用 <modules> 进行配置其中对应的是子工程的相对路径

<modules>

    <module>tuling-client</module>

    <module>tuling-server</module>

</modules>

### 2.2 继承

继承是指子工程直接继承父工程 当中的属性、依赖、插件等配置，避免重复配置。

1. 属性继承：
2. 依赖继承：
3. 插件继承：

上面的三个配置子工程都可以进行重写，重写之后以子工程的为准。

### 2.3 依赖管理

通过继承的特性，子工程是可以间接依赖父工程的依赖，但多个子工程依赖有时并不一至，这时就可以在父工程中加入 <dependencyManagement> 声明该工程需要的JAR包，然后在子工程中引入。

<！-- 父工程中声明 junit 4.12 -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

</dependencies>

</dependencyManagement>

<!-- 子工程中引入 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

### 2.4 项目属性

通过 <properties> 配置属性参数，可以简化配置。

<!-- 配置proName属性 -->

<properties>

<proName>ddd</proName>

</properties>

<!-- 引用方式 -->

${proName}

maven 默认的属性

${basedir} 项目根目录，根目录下有src与target两个子目录

${version}表示项目版本;

${project.basedir}同${basedir};

${project.version}表示项目版本,与${version}相同;

${project.build.directory} 构建目录，缺省为target

${project.build.sourceEncoding}表示主源码的编码格式;

${project.build.sourceDirectory}表示主源码路径;

${project.build.finalName}表示输出文件名称;

${project.build.outputDirectory} 构建过程输出目录，缺省为target/classes

## 2.3项目构建

构建资源配置

编译插件

profile 指定编译环境

构建资源配置

基本配置示例：

<defaultGoal>package</defaultGoal>

<directory>${basedir}/target2</directory>

<finalName>${artifactId}-${version}</finalName>

说明：

defaultGoal，执行构建时默认的goal或phase，如jar:jar或者package等

directory，构建的结果所在的路径，默认为${basedir}/target目录

finalName，构建的最终结果的名字，该名字可能在其他plugin中被改变

<resources> 配置示例

<resources>

<resource>

<directory>src/main/java</directory>

<includes>

<include>\*\*/\*.MF</include>

<include>\*\*/\*.XML</include>

</includes>

<filtering>true</filtering>

</resource>

<resource>

<directory>src/main/resources</directory>

<includes>

<include>\*\*/\*</include>

<include>\*</include>

</includes>

<filtering>true</filtering>

</resource>

</resources>

说明：

* resources，build过程中涉及的资源文件
  + targetPath，资源文件的目标路径
  + directory，资源文件的路径，默认位于${basedir}/src/main/resources/目录下
  + includes，一组文件名的匹配模式，被匹配的资源文件将被构建过程处理
  + excludes，一组文件名的匹配模式，被匹配的资源文件将被构建过程忽略。同时被includes和excludes匹配的资源文件，将被忽略。
  + filtering： 默认false ，true 表示 通过参数 对 资源文件中 的${key} 在编译时进行动态变更。替换源文件 -Dkey 和pom 中的<properties> 值或  <filters> 中指定的properties 文件。

# 3、nexus私服搭建

## 3.1 nexus安装

私服的使用场景

1、公司不能连接公网，可以用私服同一连接

2、公司内部有jar组件共享

Nexus下载

<https://sonatype-download.global.ssl.fastly.net/nexus/oss/nexus-2.14.5-02-bundle.tar.gz>

#解压

shell>tar -zxvf nexus-2.14.5-02-bundle.tar.gz

#在环境变量当中设置启动用户

shell> vim /etc/profile

#添加profile文件。安全起见不建议使用root用户，如果使用其它用户需要加相应权限

export RUN\_AS\_USER=root

配置启动参数：

shell> vi ${nexusBase}/conf/nexus.properties

 #端口号

 application-port=9999

启动与停止nexus

#启动

shell>  ${nexusBase}/bin/nexus start

#停止

shell>  ${nexusBase}/bin/nexus stop

登录nexus 界面

地址：http://{ip}:9999/nexus/

用户名:admin

密码：admin123

## 3.2 仓库种类及配置

3rd party：第三方仓库

Apache Snapshots：apache 快照仓库

Central: maven 中央仓库，当库中没有所需jar包会使用maven默认的国外仓库下载，可将其改为阿里云镜像地址

Releases：私有发布版本仓库

Snapshots：私有 快照版本仓库

在pom中配置远程仓库，可以将其设置于setting文件<profiles>中，下面介绍

<repositories>

<repository>

<id>nexus-public</id>

<name>my nexus repository</name>

<url>http://192.168.0.147:9999/nexus/content/groups/public/</url>

</repository>

</repositories>

或者在settings.xml 文件中配置远程仓库镜像 效果一样，但作用范围广了，配置在pom中的是仓库，maven中的是镜像，通过 <mirrorOf>{仓库ID}</mirrorOf> 指定覆盖仓库的地址 ,如果配置\* 会覆盖所有仓库的地址。

<mirror>

<id>nexus-aliyun</id>

        <mirrorOf>\*</mirrorOf>

        <name>Nexus aliyun</name>

  <url>http://192.168.0.147:9999/nexus/content/groups/public/</url>

</mirror>

如果想要普通的jar包使用阿里云，项目的jar包使用私服可以使用镜像覆盖私服仓库，使用阿里云镜像覆盖central（maven默认仓库的名称）

<mirror>

<id>nexus-aliyun</id>

        <mirrorOf> nexus-public </mirrorOf>

        <name>Nexus aliyun</name>

  <url>http://192.168.0.147:9999/nexus/content/groups/public/</url>

</mirror>

<mirror>

<id>nexus-aliyun</id>

        <mirrorOf> central </mirrorOf>

        <name>Nexus aliyun</name>

  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>

</mirror>

不想在每个项目中都配置仓库，可以将其配置于setting 的profiles中，依然可以被mirror覆盖，此时每个项目都会使用私服了，并且私服地址通过id被mirror覆盖

<profile>

<id> development</id>

<activation>

<jdk>1.8</jdk>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>

</properties>

<repositories>

<repository>

<snapshots>

<enabled>false</enabled>

</snapshots>

<id> nexus-public </id>

<name>my nexus repository</name>

<url>http://192.168.0.147:9999/nexus/content/groups/public/</url>

<repository>

</repositories>

</profile>

发布jar包到私服

配置仓库地址

<distributionManagement>

<repository>

<id>nexus-release</id>

<name>nexus release</name>

<url>http://192.168.0.147:9999/nexus/content/repositories/releases/</url>

</repository>

<snapshotRepository>

<id>nexus-snapshot</id>

<name>nexus snapshot</name>

<url>http://192.168.0.147:9999/nexus/content/repositories/snapshots/</url>

</snapshotRepository>

</distributionManagement>

设置 setting.xml 中设置server，用户需要有deploy权限

<server>

      <id>nexus-snapshot</id>

      <username>deployment</username>

      <password>deployment123</password>

    </server>

<server>

      <id>nexus-release</id>

      <username>deployment</username>

      <password>deployment123</password>

    </server>

执行deploy 命令

mvn deploy