**概要：**

1. maven 基本概念
2. maven 核心配置

一、maven 安装与核心概念

概要：

1. maven 安装
2. maven 编译(compile)
3. 执行测试用例(test)
4. maven 打包
5. maven 依懒管理

1、安装：

1. 官网下载 Maven （<http://maven.apache.org/download.cgi>）
2. 解压指定目录
3. 配置环境变量
4. 检查安装是否成功 （mvn -version）

maven 是什么？它的基本功能是什么？ **编译**、**打包**、**测试、依赖管理**直观感受一下maven编译打包的过程。

2、maven 编译

**maven 编译过程演示**

* 创建maven项目。
* 创建src 文件
* 编写 pom 文件
* 执行编译命令

编写pom 文件基础配置

<project 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/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

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

<artifactId>my-project</artifactId>

<version>1.0.SNAPSHOT</version>

</project>

#mvn 编译命令

mvn compile

---------------------------

[INFO] No sources to compile

[INFO] ---------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ---------------------------------------------------------------

[INFO] Total time: 0.473 s

[INFO] Finished at: 2018-08-05T15:55:44+08:00

[INFO] Final Memory: 6M/153M

[INFO] ---------------------------------------------------------------

请注意，在上述配置和命令当中，我们并没有指定源码文件在哪里？最后编译到哪里去？在这里

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;

3、Maven打包

**maven 打包演示**

#mvn 打包命令

mvn package

4、maven 单元测试演示

* 编写测试类
* 执行测试命令

编译测试类

# 创建测试目录

mkdir -p /src/test/java

# 编写 测试类

vim TestHello.java

#测试类代码------------------------

package com.test.tuling;

public class TestHello{

        public void sayHelloTest(){

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

        }

}

执行测试指令:

#执行测试

mvn test

执行完指令发现没有执行我们的测试方法，这是为何？原因在于maven 当中的测试类又做了约定，约定必须是Test开头的类名与test 开头的方法才会执行。

重新修改方法名后 在执行mvn test 即可正常执行。

package com.test.tuling;

public class TestHello{

public void testsayHelloTest(){

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

}

}

通常测试我们是通过junit 来编译测试用例，这时就就需添加junit 的依赖。

5、maven 依赖管理

* 在pom 文件中添加junit 依赖
* 修改测试类，加入junit 代码
* 执行测试命令

加入依懒配置

<dependencies>

<dependency>

  <groupId>junit</groupId>

  <artifactId>junit</artifactId>

  <version>4.0</version>

  <scope>test</scope>

</dependency>

</dependencies>

修改测试类引入junit 类.

//引入junit 类

import org.junit.Assert;

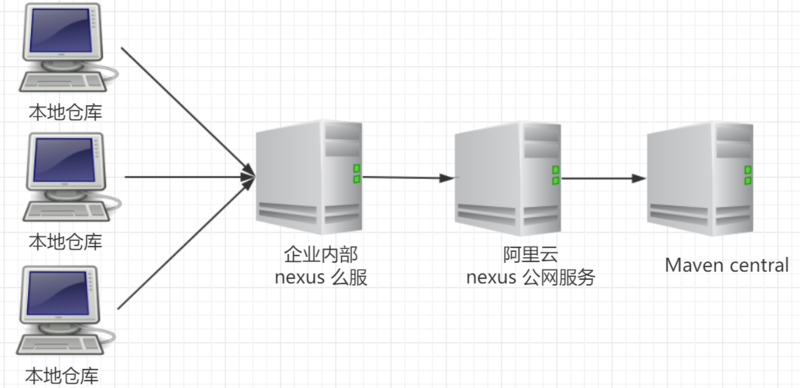
import org.junit.Test;

Assert.assertEquals("","hi");

注意：当我们在classPath 当中加入 junit ，原来以test 开头的方法不会被执行，必须加入 @Test注解才能被执行。

**提问：**

在刚才的演示过程当中 ，junit jar包在哪里？是怎么加入到classPath 当中去的？maven 是在执行test 命令的时间 动态从本地仓库中去引入junit jar 包，如果找不到就会去远程仓库下载，然后在引入。



**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。

二、maven核心配置

概要：

1. 项目依懒(内部、外部)
2. 项目聚合与继承
3. 项目构建配置

项目依懒

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。

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。

* + 举例演示Spring MVC的依赖网络

<!-- 添加spring web mvc演示 -->

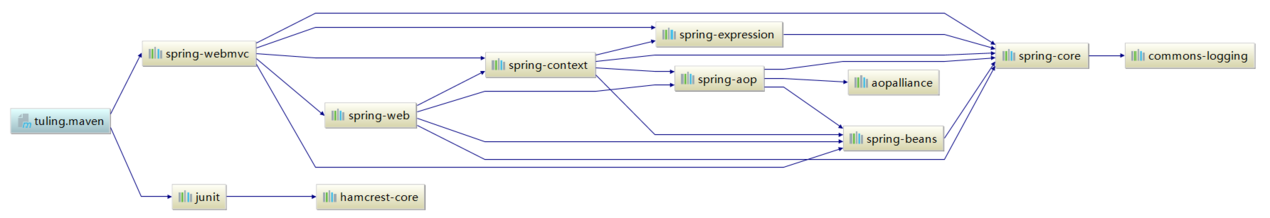
<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.0.4.RELEASE</version>

</dependency>



在刚刚的演示当中，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。

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>

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。

* 演示可选依赖的效果。

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。

<!-- 排除指定项目 -->

<exclusions>

    <exclusion>

       <groupId>org.springframework</groupId>

       <artifactId>spring-web</artifactId>

    </exclusion>

</exclusions>

* 演示排除依赖

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

**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

项目聚合与继承

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>

* + 演示聚合的配置

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。

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> 声明该功程需要的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>

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

${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

项目构建配置

1. 构建资源配置
2. 编译插件
3. 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。替换源可 紧 -Dkey 和pom 中的<properties> 值 或  <filters> 中指定的properties 文件。