**目录结构：**

**src**

-main

-java

-package (依照package创建目录）

-test

-java

-package (依照package创建目录）

Resources

D:\maven1\**src\main\java\com**\imooc\maven01\model\*HelloWorld.java：*

**package** com.imooc.maven01.model;  
  
**public class** HelloWorld{  
 **public** String sayHello(){  
 **return "Hello World!"**;  
 }  
}

D:\maven1\**src\test\java\com**\imooc\maven01\model\*HelloWorldTest.java：*

**package** com.imooc.maven01.model;  
  
**import** org.junit.\*;  
**import** org.junit.Assert.\*;  
  
**public class** HelloWorldTest{  
  
 @Test  
 **public void** testHello(){  
 Assert.assertEquals(**"Hello World!"**,**new** HelloWorld().sayHello());  
 }  
}

**src同级路径的pom.xml文件：**

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

**xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"  
 xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"**>  
 <**modelVersion**>4.0.0</**modelVersion**> // 代表maven的版本，4.0.0是固定的  
 <**groupId**>com.imooc.maven01</**groupId**> //groupId的值就是项目的包名

<**artifactId**>maven01-model</**artifactId**> //artifactId的值是模块名，建议使用项目名

<**version**>0.0.1-SNAPSHOT</**version**> //版本号，快照的版本号（\*-SNAPSHOT）

<**dependencies**> // 项目依赖  
 <**dependency**> //一个个依赖  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>4.10</**version**>  
 </**dependency**>  
</**dependencies**>

</**project**>

src上一级目录，cmd运行： **mvn compile --》** 在src同级生成target(--classes,--maven-status)文件夹。

Target下的classes文件夹放置的是**编译**后的字节码文件：

D:\temp\**target\classes\**com\imooc\maven01\model\HelloWorld.class

Target下的classes文件夹放置的

在src上一级目录，cmd运行： **mvn test --》** 运行test文件。

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

T E S T S

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

Running com.imooc.maven01.model.HelloWorldTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.072 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

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

[INFO] BUILD SUCCESS

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

[INFO] Total time: 4.130 s

[INFO] Finished at: 2018-10-23T17:43:58+08:00

[INFO] Final Memory: 16M/216M

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

src同级生产target（--surefire-reports，--test-classes）文件夹。

子文件夹surefire-reports放置测试报告：

D:\maven1\**target\surefire-reports\**com.imooc.maven01.model.HelloWorldTest.txt,TEST-com.imooc.maven01.model.HelloWorldTest.xml

子文件夹test-classes放置test文件编译后的字节码文件：

D:\maven1\target\test-classes\com\imooc\maven01\model\HelloWorldTest.class

src上一级目录，cmd运行： **mvn package --》** 在target下生成了jar包：

D:\maven1\target\**maven01-model-0.0.1-SNAPSHOT.jar**(不包含test的class文件）

src上一级目录，cmd运行： **mvn clean --》** 删除target文件夹

src上一级目录，cmd运行： **mvn install --》 打包jar包到本地仓库中**

C:\Program Files\**apache-maven-3.5.0\conf\settings.xml**文件中**<localRepository>***D:\mavenLocalRepository***</localRepository>**指定着本地仓库的位置。

**mvn install**后：

D:\*mavenLocalRepository***\com\imooc\maven01\maven01-model**

-**0.0.1-SNAPSHOT**

-\_remote.repositories

-**maven01-model-0.0.1-SNAPSHOT.jar**

**[jar包内的结构**：

-com

-imooc

-maven01

-model

-HelloWorld.class

-META-INF

-maven

-com.imooc.maven01

-maven01-model

-pom.properties

-pom.xml

-MANIFEST.MF

**]**

-maven01-model-0.0.1-SNAPSHOT.pom

[本个模块的pom文件，依赖了哪些模块]

-maven-metadata-local.xml

-maven-metadata-local.xml

**理解**：pom文件的<**groupId**></**groupId**>，<**artifactId**></**artifactId**>，<**version**></**version**>是指着本模块install后，jar包放在maven仓库的文件夹位置（**本地/远程仓库的坐标**）；同时其他模块的pom.xml文件

<**dependencies**>

<**groupId**></**groupId**>，<**artifactId**></**artifactId**>，<**version**></**version**>

</**dependencies**>

写着怎么在maven仓库中找到那个要依赖的jar包。

jar包内的结构就是代码package指定的（例如**package** com.imooc.maven01.model;），同时跟文件夹层级也是一样的。

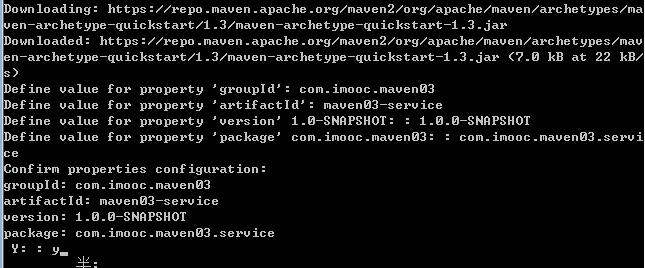
Maven是通过pom代替classpath告诉jar包存在哪里的策略？

自动创建目录骨架

Archetype插件，用于创建符合maven规定的目录骨架（第一次使用会下载东西）：

方式一：在D:\temp\maven3，cmd运行**mvn archetype:generate （按照提示进行选择和一步步输入名称）**





建完后有如下文件夹摘要：D:\temp\maven3\**maven03-service**\src\main\java\**com\imooc\maven03\service**\App.java

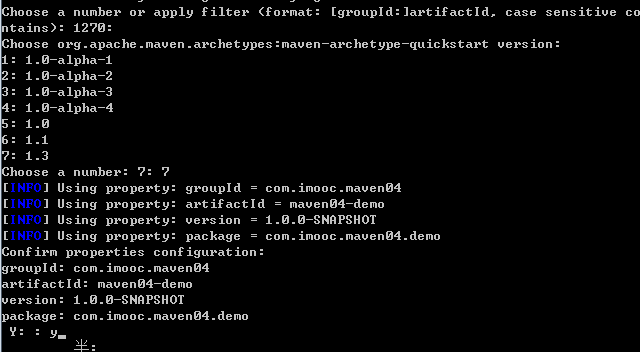
方式二：**mvn archetype:generate -DgroupId=组织名，公司网址的反写+项目名**

**-DartifactId=项目名-模块名**

**-Dversion=版本号**

**-Dpackage=代码所存在的包名**





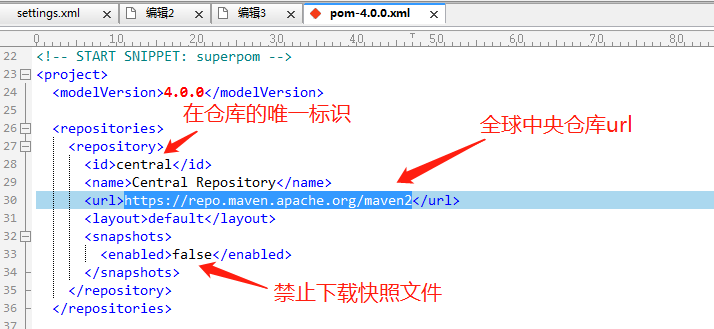
建完后有如下文件夹摘要：D:\temp\maven3\**maven04-demo**\src\main\java\**com\imooc\maven04\demo**\App.java

Maven中的坐标和仓库

C:\Program Files\apache-maven-3.5.0\lib\**maven-model-builder-3.5.0.jar**

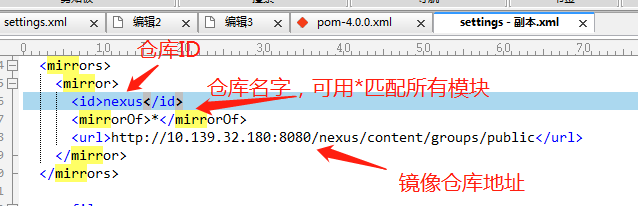
maven-model-builder-3.5.0.jar内org\apache\maven\model\**pom-4.0.0.xml**，这是maven为我们提供的超级pom，所有的pom都继承这里。

**全球中央仓库**



**镜像仓库**

在setting.xml文件中配置：



如果**配置了<mirrors></mirrors> 原有的中央仓库将不会起作用**，只从镜像仓库找模块。

**更改本地仓库位置：**

默认仓库位置：C:\Users\za-zhusifen\.m2\repository

修改C:\Program Files\apache-maven-3.5.0\conf\settings.xml文件中

**<localRepository>**D:/mavenLocalRepository**</localRepository>**的值，修改本地仓库位置。

复制一份setting.xml到D:/mavenLocalRepository（本地仓库位置），之后maven更新版本也不会更改配置？

Maven生命周期：

清理，编译，测试，打包，集成测试，验证，部署

clean,compile,test,package,install

生命周期一:clean 清理项目

pre-clean 执行清理前的工作

clean 清理上一次构建生产的所有文件

post-clean 执行清理后的文件

生命周期二:default 构建项目

compile, test, package, install (package包含执行了compile和test）

生命周期三:site 生产项目站点

pre-site 在生成项目站点前要完成的工作

site 生成项目的站点文档

post-site 在生产项目站点后要完成的工作

site-deploy 发布生产的站点到服务器上

pom.xml解析：



在dependencyManagement下申明的dependencies，Maven并不会去实际下载所依赖的jar包，而是

在dependencyManagement中用一个Map记录了jar的三维坐标。

而被仅仅是被dependencies包裹的元素，Maven是会去仓库实际下载所需要的jar包的，而至于需要下载什么

1：如果dependencies里的dependency自己没有声明version元素，那么maven就

会倒dependencyManagement里面去找有没有对该artifactId和groupId进行过版本声明，如果有，就继承它，如果

没有就会报错，告诉你必须为dependency声明一个version

2：如果dependencies中的dependency声明了version，那么无论dependencyManagement中有无对该jar的version声明，都以dependency里的version为准。

比如我们在项目中需要引入一个框架A的jar包：a.jar。而A框架的作者为了实现A框架又在其pom里面引入了spring框架，你可以在dependencyManagement中对spring进行版本声明，然后你在dependencies中添加对A框架的引用，那么实际引入的spring版本就是

你在dependencyManagement中声明的版本了



开发时有三种classpath： 1.编译compile 2.测试test 3.运行runtime

<scope></scope>就说来控制依赖与三种classpath的关系。

<scope>compile</scope>：默认的范围，编译测试运行时都有效

<scope>provided</scope>编译测试时有效，运行无效，如运行容器包括的Api，添加会冲突

<scope>runtime</scope> 测试运行时有效

<scope>test</scope> 测试时有效

<scope>system</scope> 与本机系统相关联，可移植性差

<scope>import</scope> 导入的范围，它只使用在dependencyManagement中，表示从其它的pom中导入dependency的配置。

Scope：依赖的范围

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 依赖范围  （Scope） | 对于编译classpath有效 | 对于测试classpath有效 | 对于运行时classpath有效 | 例子 |
| Compile | Y | Y | Y | Spring-core |
| Test | — | Y | — | JUnit |
| Provided | Y | Y | — | Servlet-api |
| Runtime | — | Y | Y | JDBC驱动实现 |
| System | Y | Y | — | 本地的，maven仓库职位的类库文件 |

1、a->b1.0项目之间的依赖关系：

      c->b1.1

      d->a和c

d会**传递依赖**a和c的依赖。

**先声明先优先原则**，路径长度相同，则谁先声明，先解析谁

      这个时候在d在pom中，a和c哪一个依赖先写，d就使用先写的依赖的b版本。

**短路优先**原则：

A->B->C->X(JAR2.0)

A->D->X(JAR1.0)

会优先依赖路径短的X(JAR1.0)

2、a->1.0

      c->b1.1

      d->a和c->b1.0

      f->d,c

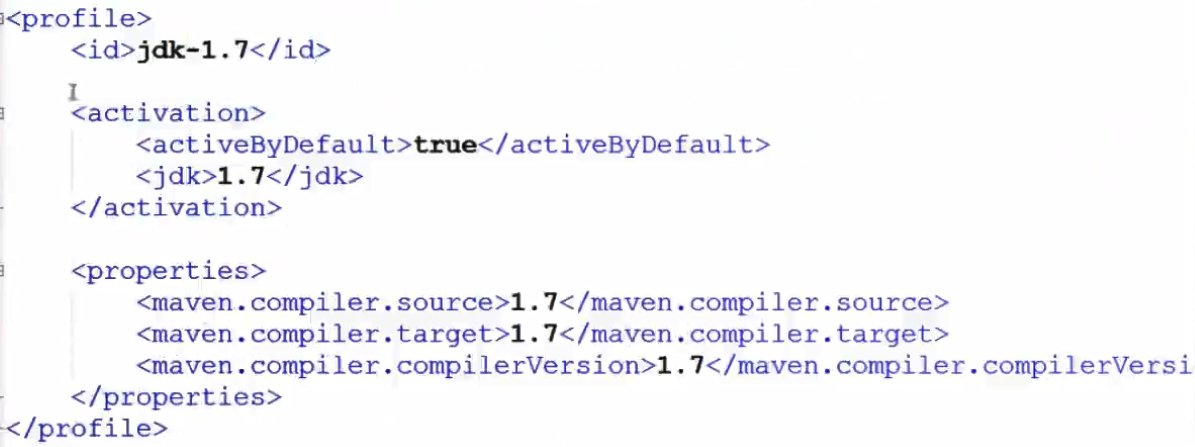
      如果路径的长短不一致就选择最小路径。

**依赖冲突解决**：

如果希望精确的控制依赖包，可以使用依赖的排除功能进行控制（<exclusions>）。

<!--排除依赖传递列表--><!--可以不依赖jar包所依赖的上一层依赖，不继承依赖-->

<exclusions> <exclusion></exclusion> </exclusions>



setting.xml中可以指定默认jdk为1.7。

maven聚合和继承：

**聚合：**多个项目放在一起执行install。setting要有如下配置：

<**!--打包方式默认是jar，还有war、 zip、 pom -->**

<packaging>pom</packaging>

**<!--编译模块，可以一起编译-->**

<modules>

<module>cps-web</module>

<module>cps-infrastructure</module>

<module>cps-param-def</module>

<module>cps-dao</module>

<module>cps-facility</module>

</modules>

**继承：**

举例：A,B,C三个项目聚合，且都依赖了相同的项目junit，

那么可以建个父项目xxx-parent，父项目的settin如下配置：

<packaging>**pom**</packaging>

**<properties>**

**<junit.version>3.8.1</junit.version>**

**</properties>**

**<dependencyManagement>**

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

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

</dependency>

</dependencies>

**</dependencyManagement>**

之后A,B,C三个项目把xxx-parent加入依赖。