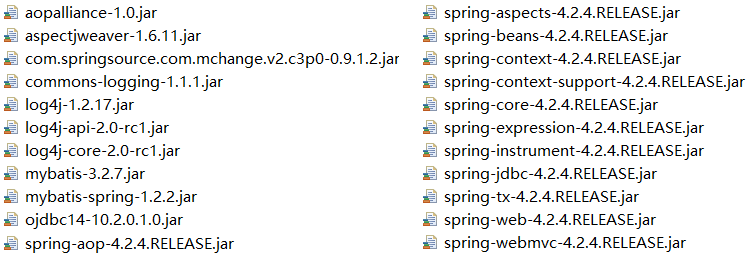
# 前言

这是一个常见的SSM整合项目中所需要的jar包



一些很直观的问题

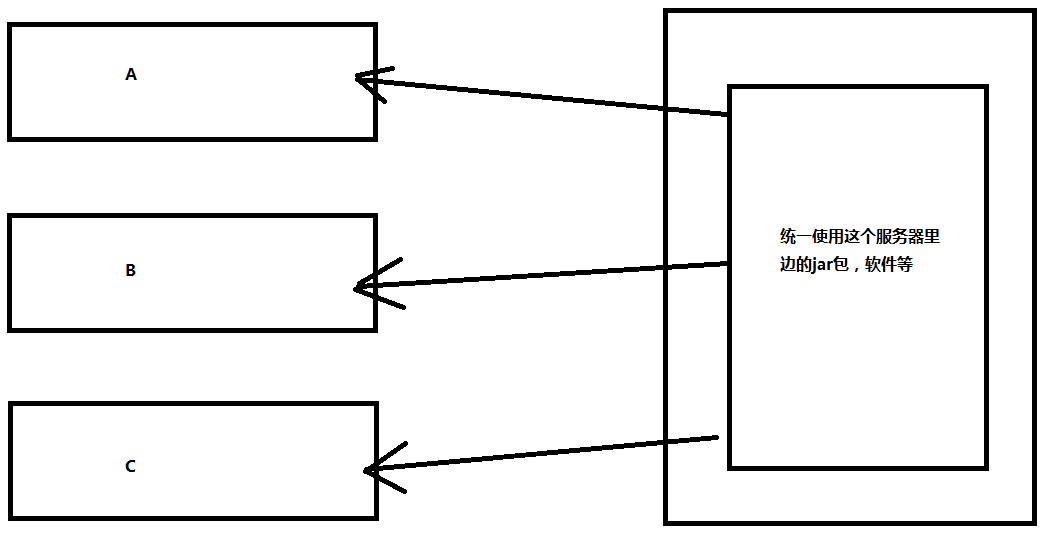
A的代码B，C下载下来以后却不能直接编译运行【ABC三个人的开发环境不一样】

公司有了三个新项目，那就把上边的jar包复制三份吧。

有四个项目要升级jar包，这四个项目的每个项目由四个人共同开发，那就再复制十六次吧。

新入职的员工我要怎么告诉他每个jar包使用哪个版本呢

为了解决这些问题，我需要写一个程序，帮我统一开发环境，统一jar包，以及开发中所有遇到的问题。它就是Maven



# Maven简介

Maven是一个[项目管理工具](https://baike.baidu.com/item/%E9%A1%B9%E7%9B%AE%E7%AE%A1%E7%90%86%E5%B7%A5%E5%85%B7)，它包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个[项目生命周期](https://baike.baidu.com/item/%E9%A1%B9%E7%9B%AE%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F)(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中[插件](https://baike.baidu.com/item/%E6%8F%92%E4%BB%B6)(plugin)目标(goal)的逻辑。

Maven主要服务于基于Java平台的**项目构建**，**依赖管理**和**项目信息管理**。

『项目管理』的角度来看，Maven 提供了如下这些功能

1. 项目对象模型（POM）：将整个项目本身抽象、封装为应用程序中的一个对象，以便于管理和操作。
2. 全局性构建逻辑重用：Maven 对整个构建过程进行封装之后，程序员只需要指定配置信息即可完成构建。让构建过程从 Ant 的『编程式』升级到了 Maven 的『声明式』。
3. 构件的标准集合：在 Maven 提供的标准框架体系内，所有的构件都可以按照统一的规范生成和使用。
4. 构件关系定义：Maven 定义了构件之间的三种基本关系，让大型应用系统可以使用 Maven 来进行管理
   1. 继承关系：通过从上到下的继承关系，将各个子构件中的重复信息提取到父构件中统一管理
   2. 聚合关系：将多个构件聚合为一个整体，便于统一操作
      1. 依赖关系：Maven 定义了依赖的范围、依赖的传递、依赖的排除、版本仲裁机制等一系列规范和标准，让大型项目可以有序容纳数百甚至更多依赖
   3. 插件目标系统：Maven 核心程序定义抽象的生命周期，然后将插件的目标绑定到生命周期中的特定阶段，实现了标准和具体实现解耦合，让 Maven 程序极具扩展性
5. 项目描述信息的维护：我们不仅可以在 POM 中声明项目描述信息，更可以将整个项目相关信息收集起来生成 HTML 页面组成的一个可以直接访问的站点。这些项目描述信息包括：
   1. 公司或组织信息
   2. 项目许可证
   3. 开发成员信息
   4. issue 管理信息
   5. SCM 信息

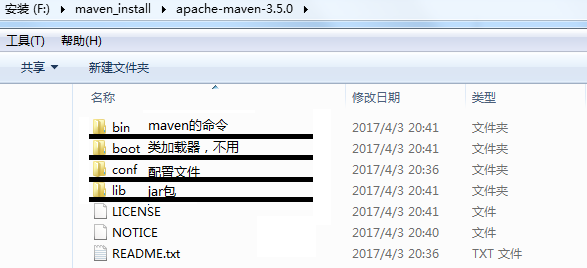
# Maven的安装

Maven是使用java开发，需要安装jdk1.6以上，推荐使用1.8，所以安装配置Maven之前，请务必保证jdk正确安装，javahome正确配置！

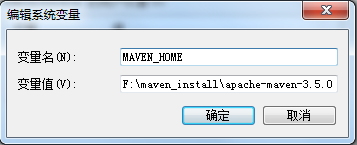
一：下载maven



二：解压缩【注意：路径中最好不要有中文】



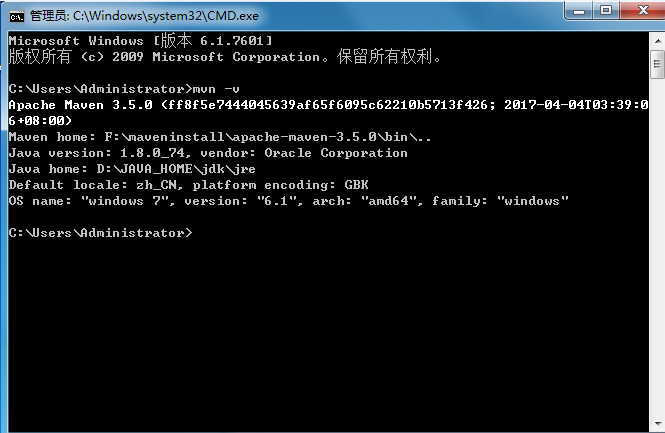
第三步：配置环境变量MAVEN\_HOME



第四步：将MAVEN\_HOME追加到PATH环境变量



第五步：验证是否安装成功，打开控制台，输入mvn –v



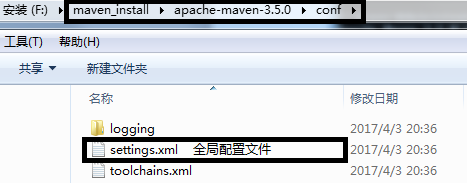
至此，Maven安装成功！

# Maven的配置

Maven的配置分为**全局配置文件**和**用户配置文件**。

## 全局配置文件

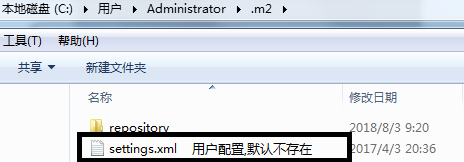
全局配置文件存放地址在maven安装目录/conf/目录中，不用理会



## 用户配置文件

用户配置文件默认不存在，需要复制一份全局配置文件settings.xml文件作为用户配置文件。一般将该文件放在~/.m2/目录中。

~代表用户目录。



修改用户配置文件

一：创建一个空的目录作为Maven的仓库，以后所有的jar包都将下载到这个目录中



二：打开用户配置文件进行修改



**注意：**

**一旦设置用户配置，全局配置将不再生效，否则使用全局配置文件中的默认路径仓库。**

**如果在用户目录下没有.m2目录无所谓。只需要记住自己的用户配置文件存放的位置即可。**

# Maven工程目录结构简介

Project

|-src ——源码目录

| |-main ——主体程序目录

| | |-java —— 存放项目的.java文件，例如com/xiaoka/maven/HelloWorld.java

| | |-resources —— 存放项目资源文件，如spring, hibernate配置文件

| |-test ——测试程序目录

| |-java ——存放所有测试.java文件，如JUnit测试类

| |-resources —— 测试资源文件

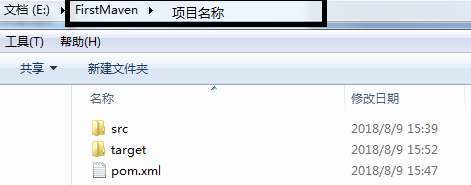
|-target —— 目标文件输出位置例如.class、.jar、.war文件

|-pom.xml ——maven项目核心配置文件

# 第一个Maven工程

## 手动创建

### 一：根据创建Maven工程所需要的目录和文件



注意：目前pom.xml是一个空文件即可

### 二：创建HelloMaven.java

在E:\FirstMaven\src\main\java\com\xiaoka\maven目录中创建HelloMaven.java文件

package com.xiaoka.maven;

public class HelloMaven{

public String hello(String name){

return "HelloMaven"+name;

}

}

### 三：创建TestHelloMaven.java

在E:\FirstMaven\src\test\java\com\xiaoka\maven目录中创建TestHelloMaven.java

package com.xiaoka.maven;

import org.junit.Test;

public class TestHelloMaven{

@Test

public void test(){

HelloMaven h = new HelloMaven();

String result = h.hello("小钻风");

System.out.println(result);

}

}

### 四：配置pom.xml文件

<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">

<!-- 版本：4.0.0 -->

<modelVersion>4.0.0</modelVersion>

<!-- 组织名称：组织名称+项目名称 -->

<groupId>com.xiaoka.maven</groupId>

<!—工程/模块名称 -->

<artifactId>FirstMaven</artifactId>

<!-- 当前项目版本号：同一个项目开发过程中可以发布多个版本，此处标示0.0.1版 -->

<version>0.0.1</version>

<!-- 名称：可省略 -->

<name>FirstMaven</name>

<!-- 依赖关系 -->

<dependencies>

<!-- 依赖设置 -->

<dependency>

<!-- 依赖组织名称 -->

<groupId>junit</groupId>

<!-- 依赖项目名称 -->

<artifactId>junit</artifactId>

<!-- 依赖版本名称 -->

<version>4.9</version>

<!-- 依赖范围：test包下依赖该设置 -->

<scope>test</scope>

</dependency>

</dependencies>

</project>

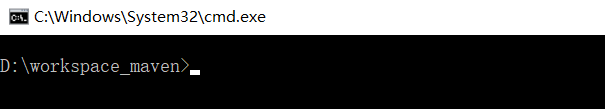
**到此，maven工程就创建完毕**

## 使用命令创建

### 第一步

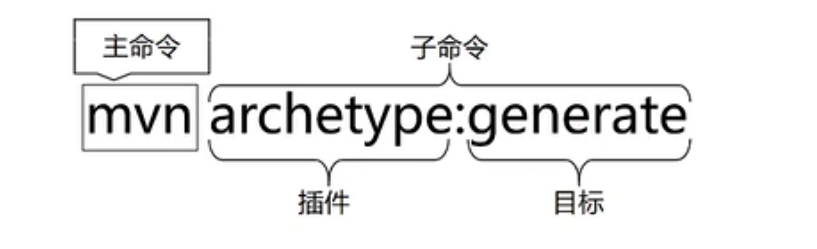
在cmd控制台，进入要创建工程的工作空间目录，为了方便学习，我们使用一个空目录

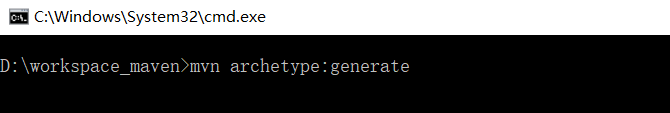




### 第二步

直接输入下列命令，执行





### 第三步

整个过程，需要自己输入GAV坐标和包名称即可完成

Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 7:【直接回车，使用默认值，如果这个值没出来，那么输入它再回车】

Define value for property 'groupId': com.xiaoka.maven

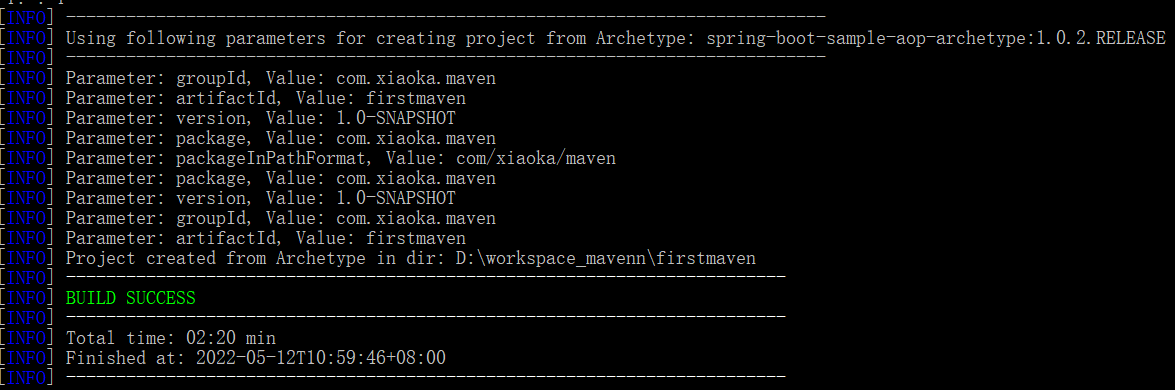
Define value for property 'artifactId': firstmaven

Define value for property 'version' 1.0-SNAPSHOT: :【直接回车，使用默认值】

Define value for property 'package' com. xiaoka.maven: :【直接回车，使用默认值】

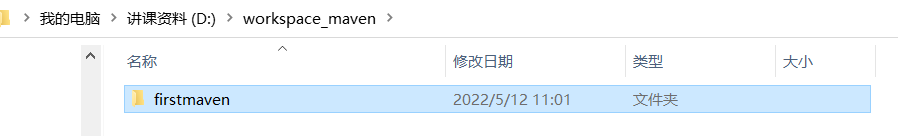
Confirm properties configuration: groupId: com. xiaoka.maven artifactId: firstmaven version: 1.0-SNAPSHOT package: com. xiaoka.maven Y: :【直接回车，表示确认。如果前面有输入错误，想要重新输入，则输入 N 再回车。】

到此，maven项目创建完成



查看工作空间,工程已经生成。

图一



图二



## pom.xml解读

<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 : 从Maven2开始就是固定的4.0.0

代表了当前pom.xml文件所采用的标签结构

-->

<modelVersion>4.0.0</modelVersion>

<!-- 当前Maven工程的坐标 -->

<groupId>com.xiaoka.maven</groupId>

<artifactId>firstmaven</artifactId>

<version>1.0-SNAPSHOT</version>

<!-- 当前Maven工程的打包方式，可选值有下面三种： -->

<!-- jar：表示这个工程是一个Java工程 -->

<!-- war：表示这个工程是一个Web工程 -->

<!-- pom：表示这个工程是“管理其他工程”的工程 -->

<packaging>jar</packaging>

<!--当前工程的名称-->

<name>firstmaven</name>

<!-- 没实际意义 -->

<url>http://www.example.com</url>

<properties>

<!-- 工程构建过程中读取源码时使用的字符集 -->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.7</maven.compiler.source>

<maven.compiler.target>1.7</maven.compiler.target>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<!-- scope标签配置依赖的范围 -->

<scope>test</scope>

</dependency>

</dependencies>

<build> 暂时不用理会 </build>

</project>

## 创建Maven版的web工程

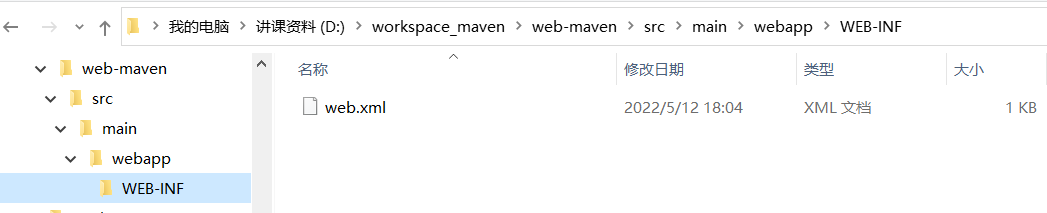
使用 mvn archetype:generate 命令生成 Web 工程时，需要使用一个专门的 archetype。这个专门生成 Web 工程骨架的 archetype：这个大家不用记忆，直接复制来用即可。

mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4

可以看到，web工程打包方式是war包。

<packaging>war</packaging>

Web工程目录结构

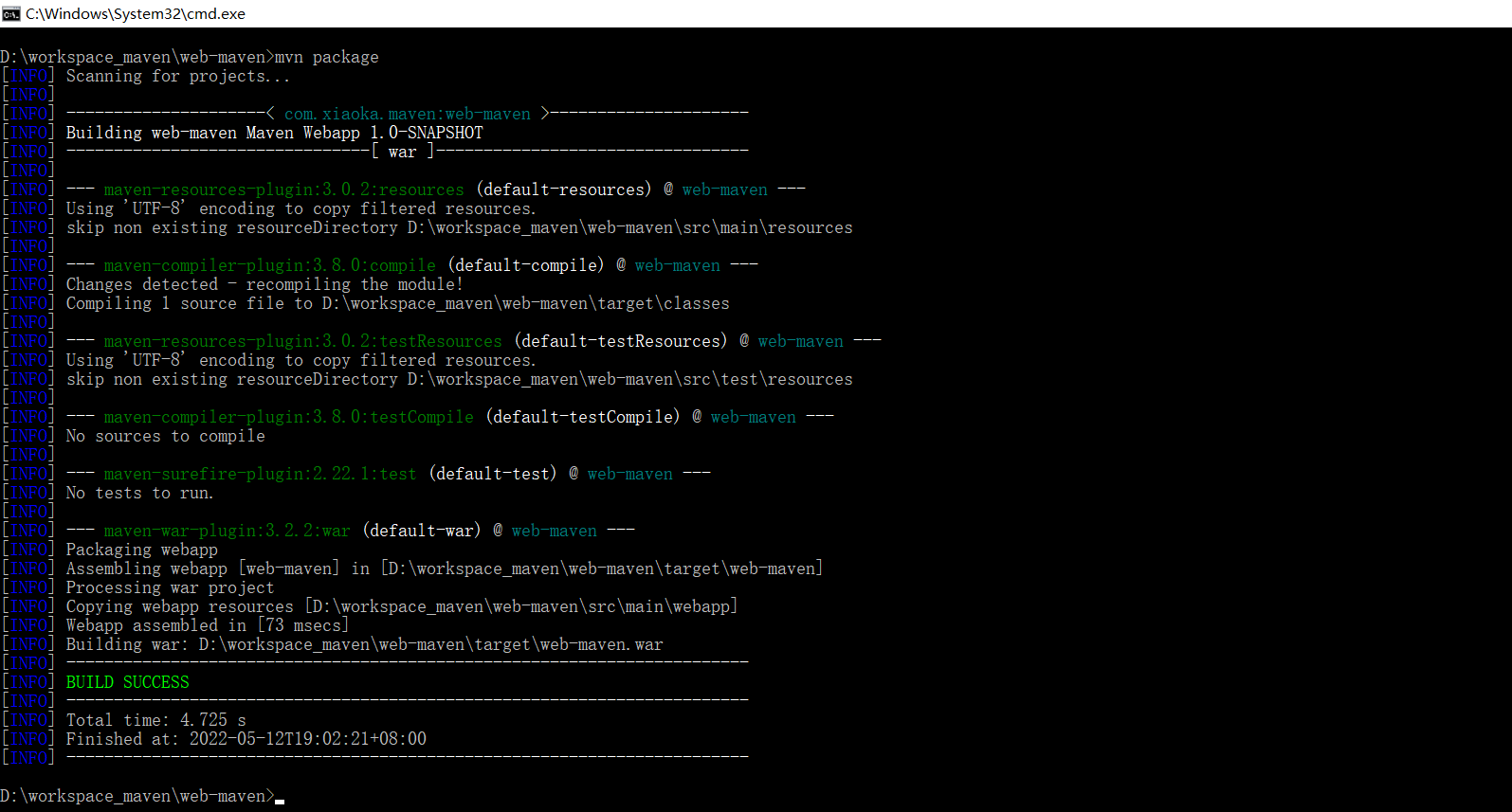


webapp 目录下有 index.jsp

WEB-INF 目录下有 web.xml

其它没有的目录，可以手动补齐(选择性操作)。

执行打包命令mvn package，可以看到编译-打包出来的是war包在target目录中。



感兴趣的同学可以将war包放入到tomcat服务器中去运行，访问。唯一的缺陷是现在的所有的动作都是手动完成的。

需要注意的是

1.第一次进行编译会下载一些maven的插件，以后就不会再下载了。

2.由于我们的web工程中没有任何Java代码，所以编译过程会有日志提示没有资源需要编译，感兴趣的同学可以把需要的Java目录补齐，导入必要的servlet依赖以后，创建一个servlet【记得创建必要的包和导入必要的Java类】，再进行编译，就会出现较多的东西了。放入到tomcat中是完全可以正常运行的。

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

# Maven的命令

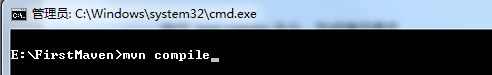
Maven作为一个项目管理工具，可以通过命令来编译，运行，测试，发布等等项目操作。

其命令需要在pom.xml所在目录中执行。操作哪个工程，就进入这个工程的 pom.xml文件目录，否则就会报错。

## Mvn compile

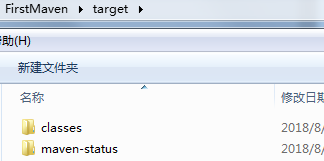
Mvn compile命令用来完成主体程序编译操作。执行完毕后，会生成target目录。

该命令会将主体程序代码编译的子节码文件存放到目录：target/classes中。



第一次执行命令会看到下载很多东西，这是maven在下载它所需要的jar包，插件等。

执行命令以后查看生成的target目录



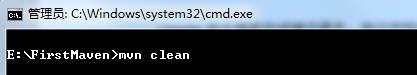
## Mvn test-compile

Mvn test-compile命令用来完成测试程序编译操作。执行完毕后，会生成target目录。

该命令会将测试程序代码编译的子节码文件存放到目录：target/test-classes中。

## Mvn clean

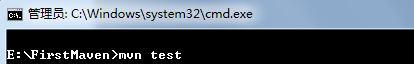
Mvn clean命令用于删除target目录

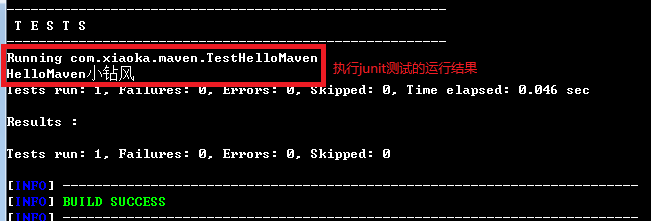


## Mvn test

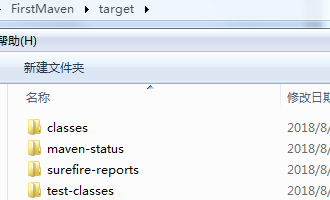
Mvn test命令用于执行junit测试,

测试的报告存放的目录：target/surefire-reports



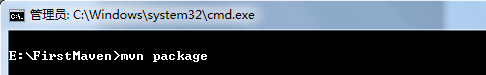


执行完毕后，会在target目录中生成三个文件夹：surefire、surefire-reports（测试报告）、test-classes（测试的字节码文件）

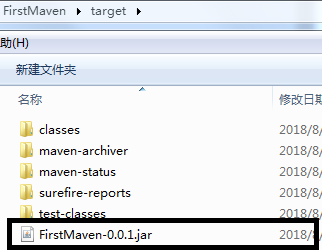


## Mvn package

Mvn package命令用于完成打包操作执行完毕后，会在target目录中生成一个文件，该文件可能是jar、war。



执行结果

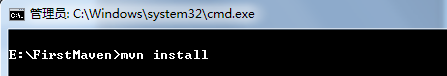


用解压缩软件打开jar包，可以看到，只有主体程序的字节码文件在jar包中，测试程序并没有任何字节码文件，这是因为测试程序在正式使用的时候并不需要。

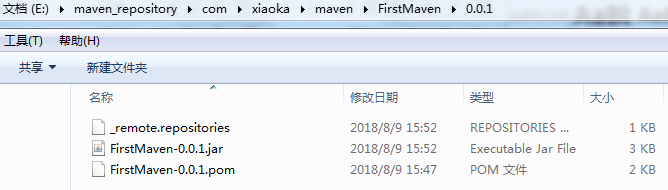
## Mvn install

Mvn install的效果是将本地构建过程中生成的 jar 包存入 Maven 本地仓库。

这个 jar 包在 Maven 仓库中的路径是根据它的坐标生成的。



执行结果



另外，安装操作还会将 pom.xml 文件转换为 XXX.pom 文件一起存入本地仓库。所以我们在 Maven 的本地仓库中想看一个 jar 包原始的 pom.xml 文件时，查看对应 XXX.pom 文件即可，它们是名字发生了改变，本质上是同一个文件。

后期无论是要部署到私服，还是本地工程之间相互依赖，都需要先install到本地。

# Maven的组合命令

**clean compile**

组合指令，先执行clean，再执行compile，通常应用于上线前执行，清除测试类

**clean test**

组合指令，先执行clean，再执行test，通常应用于测试环节

**clean package**

先执行clean，再执行package，将项目打包，通常应用于发布前

执行过程：

清理————清空环境

编译————编译源码

测试————测试源码

打包————将编译的非测试类打包

**clean install**

组合指令，先执行clean，再执行install，将项目打包，通常应用于发布前

执行过程：

清理————清空环境

编译————编译源码

测试————测试源码

打包————将编译的非测试类打包

部署————将打好的包发布到资源仓库中

# Maven的核心概念

## POM

### 含义

POM：**P**roject **O**bject **M**odel，项目对象模型。和 POM 类似的是：DOM（Document Object Model），文档对象模型。它们都是模型化思想的具体体现。

### 模型化思想

POM 表示将工程抽象为一个模型，再用程序中的对象来描述这个模型。这样我们就可以用程序来管理项目了。我们在开发过程中，最基本的做法就是将现实生活中的事物抽象为模型，然后封装模型相关的数据作为一个对象，这样就可以在程序中计算与现实事物相关的数据。

### 对应的配置文件

POM 理念集中体现在 Maven 工程根目录下 **pom.xml** 这个配置文件中。所以这个 pom.xml 配置文件就是 Maven 工程的核心配置文件。其实学习 Maven 就是学这个文件怎么配置，各个配置有什么用。

## 坐标

有A,B两个公司

A公司有A1和A2两个项目

B公司有B1和B2两个项目

假设A1项目是给长城贴瓷砖，A2项目是给喜马拉雅装电梯。

那么A1项目的完成需要多个工程协作才可以，例如前期的运输工程我们用A11表示和后期的施工工程，我们用A12表示。那么每一个工程都是项目中的一个模块 ： 即一个项目中有多个工程(模块)。

例如我们会说要开发一个电商的项目，该项目里边包含了用户模块（工程），订单模块（工程），物流模块（工程），而Maven构建出来的jar包，则是一个个的具体工程，被我们应用在了具体的项目中。

这样就可以得出来一个结论 ： A,B,C….无数个公司有无数个项目，每个项目中有若干个工程，每个工厂都被Maven构建成了一个jar包。即：Maven世界中拥有大量的jar包（构建）。

那么，如何精确的寻找到这些构建(jar包)呢？这是一件非常重要的事情。

在Maven的世界中，使用Maven的坐标来标志一个唯一的Maven构建（jar包）。

## 坐标的构成

1. **groupId**

定义当前Maven组织名称，通常由公司名称+项目名称组成，这样就可以在成千上万的构建中确定唯一的开发者（属于哪个公司开发的哪个项目，即寻找到A公司的A1项目）

1. **artifactId**

定义工程（模块）名称，即寻找到A1项目中的A11工程,这样就可以寻找到一个唯一的构建（jar包）

1. **Version**

定义当前工程(模块)的版本号（第一期工程，第二期工程…）

既然使用坐标可以精确的描述一个唯一的maven构建，那么就可以通过坐标精确的寻找到该jar包的物理路径

例如 ：

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

对应的物理路径则是

Maven本地仓库根目录\javax\servlet\servlet-api\2.5\servlet-api-2.5.jar

## 依赖管理

### 依赖范围

依赖范围用于指定当前依赖在哪个阶段有效。使用标签scope来指定依赖范围。

Scope标签的取值有六个 ： **compile**/**test**/**provided**/system/runtime/import，对于初学者来讲，首先学会前三个即可。

我们从空间和时间两个角度来比较分析，不同的取值，对依赖产生的不同结果。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 命令取值 | Main目录[空间] | Test目录[空间] | 开发过程[时间] | 部署到服务器[运行时间] |
| Compile | Y | Y | Y | Y |
| Test | N | Y | Y | N |
| provided | Y | Y | Y | N |

表格解读 ：

凡取值Y表示能用，取值N表示不能用。

例如，junit测试，测试过程中使用的 jar 包，所以用 test 范围依赖进来。

那么在main目录中的Java类不能使用junit测试，在测试目录中的Java类才能使用junit测试。在开发过程中可以使用，在项目真正发布运行以后不能使用。

provided：在开发过程中需要用到的“服务器上的 jar 包”通常以 provided 范围依赖进来。比如 servlet-api、jsp-api。而这个范围的 jar 包之所以不参与部署、不放进 war 包，就是避免和服务器上已有的同类 jar 包产生冲突，同时减轻服务器的负担。说白了就是：“**服务器上已经有了，你就别带啦！**”

对于此表格的验证，没有必要在程序中运行，只需要通过导包，编译，即可的出来结论。取值Y的地方应该编译通过，取值N的地方编译结果是失败。

至于以后在工作中，如何对每个依赖的scope取值，大家不用担心，在maven仓库中，每个依赖的坐标中就包含了依赖范围的取值，没写就是使用默认值compile，大家直接复制过来用即可，根本不需要重新修改scope的取值。

Import

直译过来就是导入的意思，而它也正是导入依赖的意思。

管理依赖最基本的办法是继承父工程，但是和 Java 类一样，Maven 也是单继承的。如果不同体系的依赖信息封装在不同 POM 中了，没办法继承多个父工程怎么办？这时就可以使用 import 依赖范围。

典型案例当然是在项目中引入 SpringBoot、SpringCloud 依赖：



import 依赖范围使用要求：

* 打包类型必须是 pom
* 必须放在 dependencyManagement 中

System

以Windows系统环境下开发为例，假设D:\maven \hello-maven-1.0-SNAPSHOT.jar

想要引入到我们的项目中，此时我们就可以将依赖配置为 system 范围：

<dependency>

<groupId>com.xiaoka.maven</groupId>

<artifactId>hello-maven</artifactId>

<version>1.0-SNAPSHOT</version>

<systemPath>D:\maven \hello-maven-1.0-SNAPSHOT.jar</systemPath>

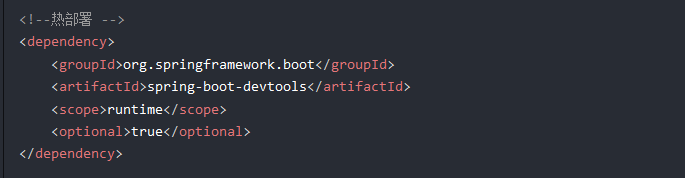
<scope>system</scope>

</dependency>

所以，system范围是引入本地硬盘，绝对路径中的依赖，但是很明显：这样引入依赖完全不具有可移植性，所以不要使用。如果需要引入体系外 jar 包我们后面会讲专门的办法。

Runtime

专门用于编译时不需要，但是运行时需要的 jar 包。比如：编译时我们根据接口调用方法，但是实际运行时需要的是接口的实现类。典型案例是：

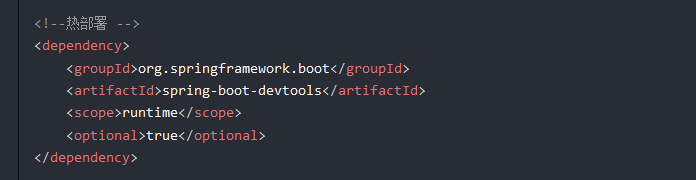


可选依赖

其核心含义是：Project X 依赖 Project A，A 中一部分 X 用不到的代码依赖了 B，那么对 X 来说 B 就是『可有可无』的。

可以理解为，如果为true，则表示该依赖不会传递下去，如果为false，则会传递下去

通过<optional>true</ optional >来配置可选依赖。热部署的依赖就是可选依赖。



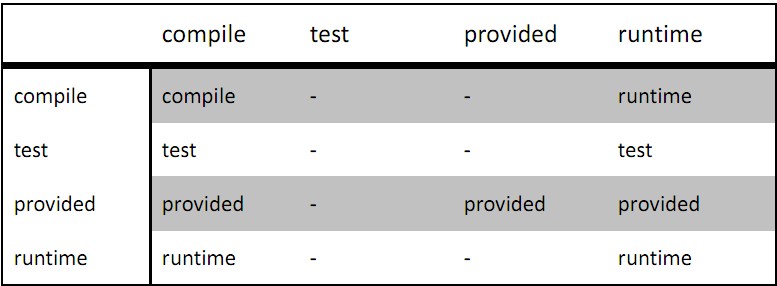
。

### 依赖传递

有A,B,C三个工程

C工程依赖B工程，B工程依赖A工程

B工程就是C工程的第一直接依赖，A工程则是C工程的第二直接依赖。



左边第一列表示第一直接依赖范围

上面第一行表示第二直接依赖范围

中间的交叉单元格表示传递性依赖范围。

总结：

当第二依赖的范围是compile的时候，传递性依赖的范围与第一直接依赖的范围一致。

当第二直接依赖的范围是test的时候，依赖不会得以传递。

当第二依赖的范围是provided的时候，只传递第一直接依赖范围也为provided的依赖，且传递性依赖的范围同样为 provided；

当第二直接依赖的范围是runtime的时候，传递性依赖的范围与第一直接依赖的范围一致，但compile例外，此时传递的依赖范围为runtime；

这个表格看起来稍微有些复杂，但是我们可以简单的记住一个结论：

A→B→C

此时，B是第一直接依赖，C是第二依赖

compile

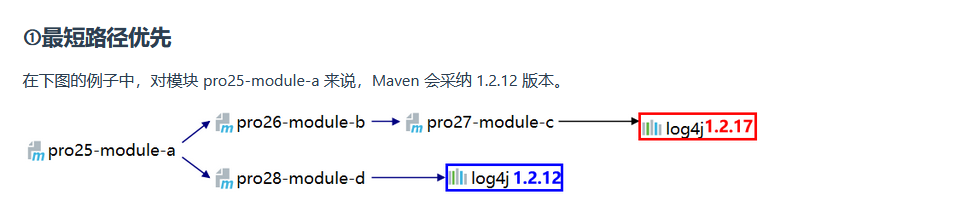
A → B → C

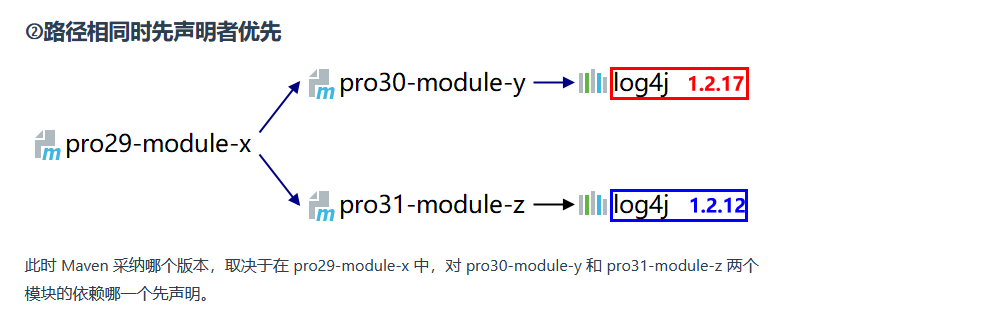
仅当B对C的依赖是compile的时候，A才可以依赖到C。如果B对C的依赖是test或者provided，那么C无法传递到A。此时如果A要用C，则必须单独添加对C的依赖。

### 依赖冲突

#### 版本仲裁

Maven 的版本仲裁机制只是在没有人为干预的情况下，自主决定 jar 包版本的一个办法。而实际上我们要使用具体的哪一个版本，还要取决于项目中的实际情况。所以在项目正常运行的情况下，jar 包版本可以由 Maven 仲裁，不必我们操心；而发生冲突时 Maven 仲裁决定的版本无法满足要求，此时就应该由程序员明确指定 jar 包版本。





距离pom.xml顶端越近，就是越先声明。

#### 依赖冲突

##### 谁需要面对 jar 包冲突

先给结论：编订依赖列表的程序员。初次设定一组依赖，因为尚未经过验证，所以确实有可能存在各种问题，需要做有针对性的调整。那么谁来做这件事呢？我们最不希望看到的就是：团队中每个程序员都需要自己去找依赖，即使是做同一个项目，每个模块也各加各的依赖，没有统一管理。那前人踩过的坑，后人还要再踩一遍。而且大家用的依赖有很多细节都不一样，版本更是五花八门，这就让事情变得更加复杂。

所以虽然初期需要根据项目开发和实际运行情况对依赖配置不断调整，最终确定一个各方面都 OK 的版本。但是一旦确定下来，放在父工程中做依赖管理，各个子模块各取所需，这样基本上就能很好的避免问题的扩散。

即使开发中遇到了新问题，也可以回到源头检查、调整 dependencyManagement 配置的列表——而不是每个模块都要改。所以学完这一节你应该就会对前面讲过的『继承』有了更深的理解。

##### 表现形式

由于实际开发时我们往往都会整合使用很多大型框架，所以一个项目中哪怕只是一个模块也会涉及到大量 jar 包。数以百计的 jar 包要彼此协调、精密配合才能保证程序正常运行。而规模如此庞大的 jar 包组合在一起难免会有磕磕碰碰。最关键的是由于 jar 包冲突所导致的问题非常诡异，这里我们只能罗列较为典型的问题，而没法保证穷举。

但是我们仍然能够指出一点：一般来说，由于我们自己编写代码、配置文件写错所导致的问题通常能够在异常信息中看到我们自己类的全类名或配置文件的所在路径。如果整个错误信息中完全没有我们负责的部分，全部是框架、第三方工具包里面的类报错，这往往就是 jar 包的问题所引起的。

而具体的表现形式中，主要体现为找不到类或找不到方法。

##### 冲突的具体表现

**抛异常：找不到类**

此时抛出的常见的异常类型：

* java.lang.**ClassNotFoundException**：编译过程中找不到类
* java.lang.**NoClassDefFoundError**：运行过程中找不到类
* java.lang.**LinkageError**：不同类加载器分别加载的多个类有相同的全限定名

我们来举个例子：

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.x.x</version>

</dependency>

httpclient 这个 jar 包中有一个类：org.apache.http.conn.ssl.NoopHostnameVerifier。这个类在较低版本中没有，但在较高版本存在。比如：

| **jar 包版本** | **是否存在** |
| --- | --- |
| 4.3.6 | 否 |
| 4.4 | 是 |

那当我们确实需要用到 NoopHostnameVerifier 这个类，我们看到 Maven 通过依赖传递机制引入了这个 jar 包，所以没有明确地显式声明对这个 jar 包的依赖。可是 Maven 传递过来的 jar 包是 4.3.6 版本，里面没有包含我们需要的类，就会抛出异常。

而『冲突』体现在：4.3.6 和 4.4 这两个版本的 jar 包都被框架所依赖的 jar 包给传递进来了，但是假设 Maven 根据**『版本仲裁』**规则实际采纳的是 4.3.6。

**抛异常：找不到方法**

程序找不到符合预期的方法。这种情况多见于通过反射调用方法，所以经常会导致：java.lang.NoSuchMethodError。比如 antlr:antlr:x.x.x 这个包中有一个接口：antlr.collections.AST

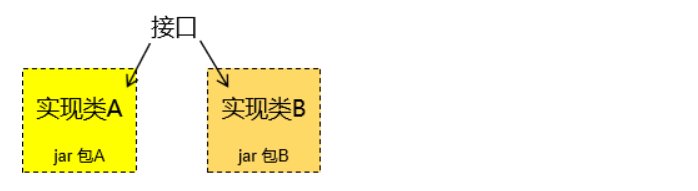
| **版本** | **getLine()方法** |
| --- | --- |
| 2.7.2 | 无 |
| 2.7.6 | 有 |

**没报错但结果不对**

发生这种情况比较典型的原因是：

两个 jar 包中的类分别实现了同一个接口，这本来是很正常的。

但是问题在于：由于没有注意命名规范，两个不同实现类恰巧是同一个名字。

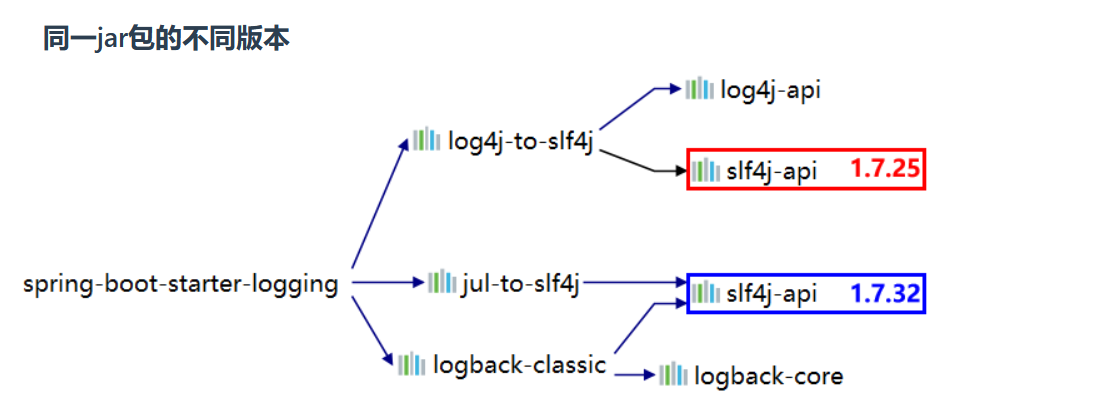


具体例子是有的同学在实际工作中遇到过：项目中部分模块使用 log4j 打印日志；其它模块使用 logback，编译运行都不会冲突，但是会引起日志服务降级，让你的 log 配置文件失效。比如：你指定了 error 级别输出，但是冲突就会导致 info、debug 都在输出。

**本质**

以上表现形式归根到底是**两种基本情况**导致的：

* 同一jar包的不同版本
* 不同jar包中包含同名类





当然，如果全限定名相同，类中的代码也完全相同，那么用着也行。问题是如果**『全限定名相同』**，但是**『代码不同』**，那可太坑了。我们随便找一个来看看：





##### 冲突的解决方案

概述

很多情况下常用框架之间的整合容易出现的冲突问题都有人总结过了，拿抛出的异常搜索一下基本上就可以直接找到对应的 jar 包。我们接下来要说的是通用方法。

不管具体使用的是什么工具，基本思路无非是这么两步：

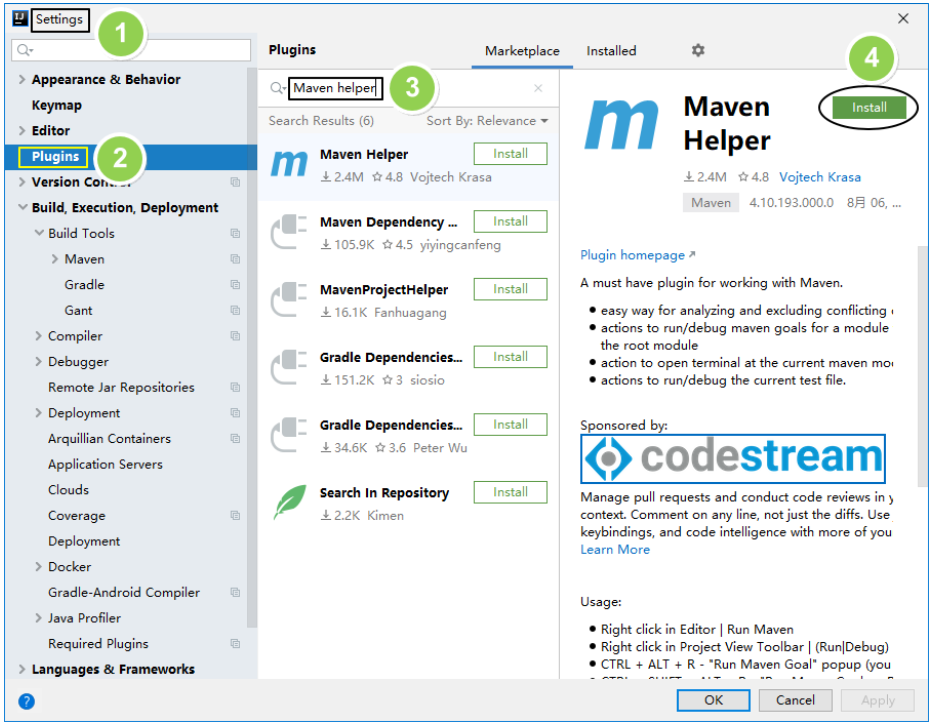
* 第一步：把彼此冲突的 jar 包找到
* 第二步：在冲突的 jar 包中选定一个。具体做法无非是通过 exclusions 排除依赖，或是明确声明依赖。

**使用IDEA 的 Maven Helper 插件来找到冲突的jar包**

**使用Maven 的 enforcer 插件来找到冲突的jar包**

**IDEA 的 Maven Helper 插件安装和使用**

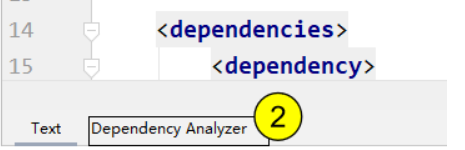
这个插件是 IDEA 中安装的插件，不是 Maven 插件。它能够给我们罗列出来同一个 jar 包的不同版本，以及它们的来源。但是对不同 jar 包中同名的类没有办法。



**基于pom.xml的依赖冲突分析**

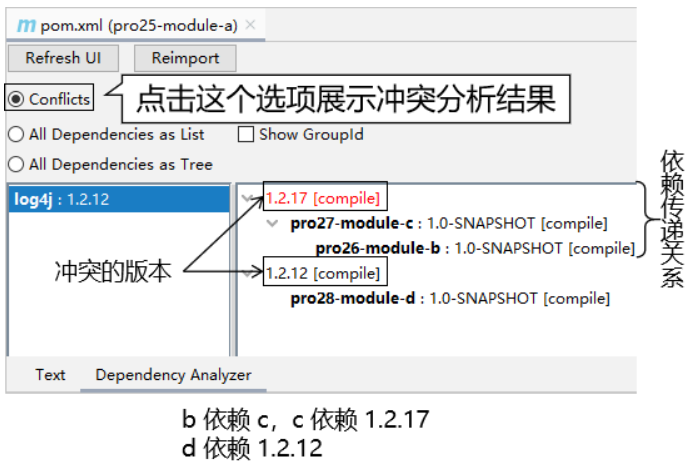
双击pom.xml文件





注意：这个功能需要安装了 IDEA 的 Maven helper 插件才有

查看冲突分析结果



**Maven 的 enforcer 插件**

使用 Maven 的 enforcer 插件既可以检测同一个 jar 包的不同版本，又可以检测不同 jar 包中同名的类。

案例演示

引入 netty 依赖

这里我们引入两个对 netty 的依赖，展示不同 jar 包中有同名类的情况。

<dependencies>

<dependency>

<groupId>org.jboss.netty</groupId>

<artifactId>netty</artifactId>

<version>3.2.10.Final</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty</artifactId>

<version>3.9.2.Final</version>

</dependency>

</dependencies>

**配置 enforcer 插件**

<build>

<pluginManagement>

<plugins>

<plugin>

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

<artifactId>maven-enforcer-plugin</artifactId>

<version>1.4.1</version>

<executions>

<execution>

<id>enforce-dependencies</id>

<phase>validate</phase>

<goals>

<goal>display-info</goal>

<goal>enforce</goal>

</goals>

</execution>

</executions>

<dependencies>

<dependency>

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

<artifactId>extra-enforcer-rules</artifactId>

<version>1.0-beta-4</version>

</dependency>

</dependencies>

<configuration>

<rules>

<banDuplicateClasses>

<findAllDuplicates>true</findAllDuplicates>

</banDuplicateClasses>

</rules>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

执行如下 Maven 命令：

mvn clean package enforcer:enforce

部门运行结果

[INFO] --- maven-enforcer-plugin:1.4.1:enforce (default-cli) @ pro32-duplicate-class ---  
[WARNING] Rule 0: org.apache.maven.plugins.enforcer.BanDuplicateClasses failed with message:  
Duplicate classes found:

Found in:  
io.netty:netty:jar:3.9.2.Final:compile  
org.jboss.netty:netty:jar:3.2.10.Final:compile  
Duplicate classes:  
org/jboss/netty/channel/socket/ServerSocketChannelConfig.class  
org/jboss/netty/channel/socket/nio/NioSocketChannelConfig.class  
org/jboss/netty/util/internal/jzlib/Deflate.class  
org/jboss/netty/handler/codec/serialization/ObjectDecoder.class  
org/jboss/netty/util/internal/ConcurrentHashMap$HashIterator.class

总结 ：使用maven管理jar包都这么麻烦，如果没有maven，纯靠人力来管理这纷繁复杂的依赖关系，那几乎不是人力可以解决的。。。

================================注释掉=====================================

在maven中存在两种冲突方式：一种是跨pom文件的冲突，一个是同一个pom文件中的冲突。

跨pom文件的冲突

MavenFirst的pom文件中依赖来junit的4.11版本，那边MavenSecond和MavenThird中都是使用了4.11版本。

<!-- FirstMaven的pom.文件 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>compile</scope>

</dependency>

如果MavenSecond中重新依赖junit的4.12版本，那么MavenSecond和MavenThird中都是使用了4.12本，这体现来依赖的就近使用原则。

<!-- MavenSecond的pom.xml文件 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>compile</scope>

</dependency>

同一个pom文件的冲突

<!-- MavenSecond的pom.文件 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>compile</scope>

</dependency>

<!-- 同一个pom.xml引入同一个构建，按照就近原则，越往下越近 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>compile</scope>

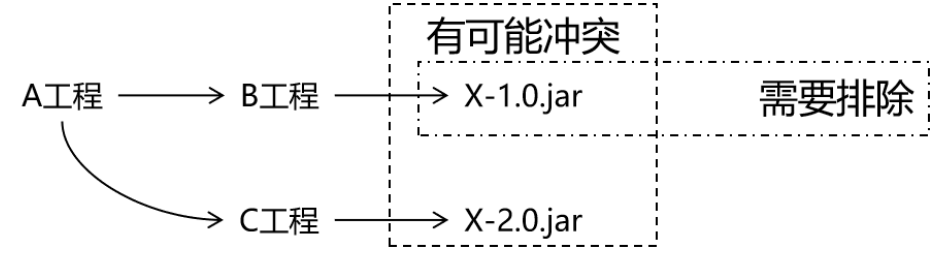
</dependency>

================================注释掉=====================================

### 排除依赖

如下图依赖关系，A依赖B和C。B依赖X-1.0.jar,C依赖X-2.0.jar，并且x-1.0.jar和x-2.0.jar都可以依赖传递给A。那么此时A里边就有两个版本不同的x.jar包。这种情况，为了避免两个x.jar包在A工程中冲突，我们必须阻断一个x.jar包的传递，使得A工程中只有一个x.jar包。

这就是排除依赖，Exclusions标签可以排除依赖。



用法非常简单， 只需要注意的是，我们要排除x.jar传递到A，那么这个Exclusions标签是要在A工程中的pom.xml文件中。

A.pom.xml文件

<dependency>

<groupId>com.xiaoka.maven</groupId>

<artifactId>web-maven<artifactId>

<version>1.0-SNAPSHOT</version>

<scope>compile</scope>

<!-- 使用excludes标签配置依赖的排除 -->

<exclusions>

<!-- 在exclude标签中配置一个具体的排除 -->

<exclusion>

<!-- 指定要排除的依赖的坐标（不需要写version） -->

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

这样commons-logging就无法通过web-maven传递给A工程了。

### 查看依赖

首先要明确一点：从来只有 Web 工程依赖 Java 工程，没有反过来 Java 工程依赖 Web 工程。本质上来说，Web 工程依赖的 Java 工程其实就是 Web 工程里导入的 jar 包。最终 Java 工程会变成 jar 包，放在 Web 工程的 WEB-INF/lib 目录下。

查看当前 Web 工程所依赖的 jar 包的列表

mvn dependency:list

以树形结构查看当前 Web 工程的依赖信息

mvn dependency:tree

## 生命周期

为了让构建过程自动化完成，Maven 设定了三个生命周期，生命周期中的每一个环节对应构建过程中的一个操作。

生命周期是由多个阶段（Phase）组成。每个阶段都是一个完整的功能，比如mvn clean中的clean就是一个阶段。

Maven有三个生命周期：clean生命周期、default生命周期、site生命周期

生命周期可以理解为项目构建的步骤集合。

### Clean生命周期

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

mvn clean命令，等同于 mvn pre-clean clean。只要执行后面的命令，那么前面的命令都会执行，不需要再重新去输入命令。

有Clean生命周期，在生命周期又有clean阶段。

### Default生命周期（重点）

validate

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。或者部署到服务器上运行（需借助插件，例如：cargo）。

在maven中，只要在同一个生命周期，你执行后面的阶段，那么前面的阶段也会被执行，而且不需要额外去输入前面的阶段，这样大大减轻了程序员的工作。

### Site生命周期

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

## 插件

插件（plugin），每个插件都能实现一个阶段的功能。Maven的核心是生命周期，但是生命周期相当于主要指定了maven命令执行的流程顺序，而没有真正实现流程的功能，功能是由插件来实现的。

比如：compile就是一个插件实现的功能。

### 编译插件

<plugin>

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

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

### Tomcat插件

如果使用maven的tomcat插件的话，那么本地则不需要安装tomcat。

<plugin>

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

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<port>80</port>

<path>/</path>

</configuration>

</plugin>

注意：创建Web工程选择打包方式需要使用war包

## 继承

在maven中的继承，指的是pom文件的继承。其主要的作用如下 ：

在父工程中统一管理项目中的依赖信息，具体来说是管理依赖信息的版本。

它的背景是：

对一个比较大型的项目进行了模块拆分。

一个 project 下面，创建了很多个 module。

每一个 module 都需要配置自己的依赖信息。

它背后的需求是：

在每一个 module 中各自维护各自的依赖信息很容易混乱，不方便统一管理。

使用同一个框架内的不同 jar 包，它们应该是同一个版本，所以整个项目中使用的框架版本需要统一。

使用框架时所需要的 jar 包组合（或者说依赖信息组合）需要经过长期摸索和反复调试，最终确定一个可用组合。这个耗费很大精力总结出来的方案不应该在新的项目中重新摸索。

通过在父工程中为整个项目维护依赖信息的组合既**保证了整个项目使用规范、准确的 jar 包**；又能够将**以往的经验沉淀**下来，节约时间和精力。

一：创建父工程Mavenfu

创建的过程和前面使用命令创建maven工程一样。

工程名称： maven-fu

工程创建好之后，要修改它的打包方式,打开maven-parent.pom.xml进行修改：

<groupId>com.xiaoka.maven</groupId>

<artifactId>maven-fu</artifactId>

<version>1.0-SNAPSHOT</version>

<!-- 当前工程作为父工程，它要去管理子工程，所以打包方式必须是 pom -->

<packaging>pom</packaging>

注意：父工程打包方式必须是pom形式。

只有打包方式为 pom 的 Maven 工程能够管理其他 Maven 工程。打包方式为 pom 的 Maven 工程中不写业务代码，它是专门管理其他 Maven 工程的工程。

还有，父工程作为统一依赖管理，它本身不依赖别人，所以，通常把父工程中的依赖信息全部删除。

二：创建子工程

子工程也叫模块工程，类似于 IDEA 中的 module，所以需要**进入 maven-fu工程的根目录**，然后运行 mvn archetype:generate 命令来创建子/模块工程。

假设我们要创建三个子/模块工程，那么就在**maven-fu工程的根目录执行三次**mvn archetype:generate命令，输入三个子工程的GAV，通常，子工程的GroupID和父工程一致。当然不一样也行，没有强制要求。



然后再查看父工程 pom.xml文件，可以看到三个模块信息被添加到了父工程的pom.xml文件中。

Maven-fu.pom.xml文件

<modules>

<module>zi01</module>

<module>zi02</module>

<module>zi03</module>

</modules>

查看任意一个子工程的pom.xml文件信息，可以看到有父工程的配置信息。

子工程的zi01.pom.xml文件

<!-- 使用parent标签指定当前工程的父工程 -->

<parent>

<!-- 父工程的坐标 -->

<groupId>com.xiaoka.maven</groupId>

<artifactId>maven-fu </artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<!-- 子工程的坐标 -->

<!-- 如果子工程坐标中的groupId和version与父工程一致，那么可以省略 -->

<!-- <groupId>com.xiaoka.maven</groupId> -->

<artifactId>zi01</artifactId>

<!-- <version>1.0-SNAPSHOT</version> -->

### 父工程统一管理依赖jar包

<!—这是父工程的pom.xml文件 -->

<!-- 使用dependencyManagement标签配置对依赖的管理 -->

<!-- 被管理的依赖并没有真正被引入到工程中 -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

</dependencies>

</dependencyManagement>

<!--这是子工程的pom.xml文件 -->

<!--父工程只做管理，不做导入，所以子工程还需要再具体的导入需要的依赖。 -->

<!--子工程引用父工程中的依赖信息时，可以把版本号去掉。 -->

<!--把版本号去掉就表示子工程中这个依赖的版本号由父工程决定。 -->

<!--具体来说是由父工程的dependencyManagement来决定。 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<!—如果不想用父工程中定义的版本号，则子工程需要显示的把version号写出来-->

<!-- <version>4.13</version> -->

</dependency>

感兴趣的同学可以现在就体验一下由父工程统一管理版本的好处，只需要修改父工程中的依赖版本号，所有子工程中相对应的依赖的版本号会统一得到修改，和父工程保持一致。

### 在父工程中声明自定义属性-父工程抽取版本号

<!--properties中的子标签可以任意指定 -->

<!-- 通过自定义属性，统一指定依赖的版本号 -->

<!—在mavenfu.pom.xml文件中自定义属性-->

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- 自定义标签，维护junit版本号 -->

<xiaoka.junit.version>4.3.6.RELEASE</xiaoka. junit.version>

</properties>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

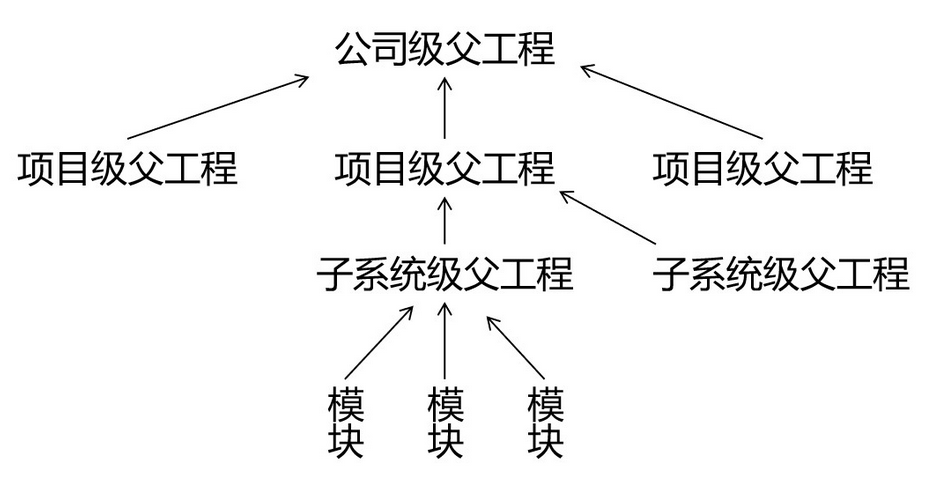
<!—在mavenfu.pom.xml文件中引用自定义属性值-->

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

</dependency>

如果不这样做，假设父工程中管理10个依赖，更新版本号的时候，虽然只需要更新父工程中的版本号即可，但是我们需要在父工程的pom文件中进行十次修改操作，而使用了自定义属性以后，则只需要修改自定义属性的值，那么父工程中的10个依赖由于都是引用的属性值，所以就这10个依赖的版本号就都得到了修改，真正实现了一处修改，处处改变的效果，使管理更加方便。

### 生产环境图



编写一套符合要求、开发各种功能都能正常工作的依赖组合并不容易。如果公司里已经有人总结了成熟的组合方案，那么再开发新项目时，如果不使用原有的积累，而是重新摸索，会浪费大量的时间。为了提高效率，我们可以使用工程继承的机制，让成熟的依赖组合方案能够保留下来。

如上图所示，公司级的父工程中管理的就是成熟的依赖组合方案，各个新项目、子系统各取所需即可。

## 聚合

### 聚合的概念

这个小节，我要先告诉大家结论再往后讲解 ：我们已经完成了聚合工程的创建和配置。

首先看概念，聚合，顾名思义，就是将各个部分聚集起来，组成一个整体，此即为聚合。

所以，我们需要使用一个“总工程”将各个“模块工程”汇集起来，作为一个整体对应完整的项目。

所以，我们需要在总工程中配置各个 modules：

Mavenfu.pom.xml文件

<modules>

<module>zi01</module>

<module>zi02</module>

<module>zi03</module>

</modules>

到这里我们发现，聚合的配置和继承的形式是一样的，所以先告诉了大家结论，就是为了强调，继承就是聚合，聚合就是继承，因此，我们创建完继承工程的时候，实际上就是创建了聚合工程，只不过，它们是从两个不同的角度来看待的：

从继承关系角度来看：

* 父工程
* 子工程

从聚合关系角度来看：

* 总工程
* 模块工程

### 聚合概念的好处

有了聚合的概念以后，我们就可以将父工程和子工程统一看成一个整体，这样我们就可以通过操作父工程（总工程）来实现对所有子工程的操作，因为它们是一个整体！

以 mvn install 命令为例，我们只需要在父工程中执行一次mvn install命令，那么所有的子工程都会被安装到本地仓库中，而不用一个个的去安装。更重要的是：Maven 要求有父工程时先安装父工程；有依赖的工程时，先安装被依赖的工程。我们自己考虑这些规则会很麻烦。但是工程聚合之后，在总工程执行 mvn install 可以一键完成安装，而且会自动按照正确的顺序执行。

注意，很多Maven 命令都可以一键执行：很多构建命令都可以在“总工程”中一键执行。

### 依赖死循环

如果 A 工程依赖 B 工程，B 工程依赖 C 工程，C 工程又反过来依赖 A 工程，那么在执行构建操作时会报下面的错误：

DANGER

[ERROR] [ERROR] The projects in the reactor contain a cyclic reference:

这个错误的含义是：循环引用。

## Maven的仓库

### 本地仓库

在当前电脑上，为电脑上所有 Maven 工程服务。

默认在~/m2目录中，通常后期自己设置一个本地其它路径。

### 远程仓库

需要联网才可以访问。具体分为

1. 局域网

我们自己搭建的 Maven 私服，例如使用 Nexus 技术。

1. Internet
   1. 中央仓库

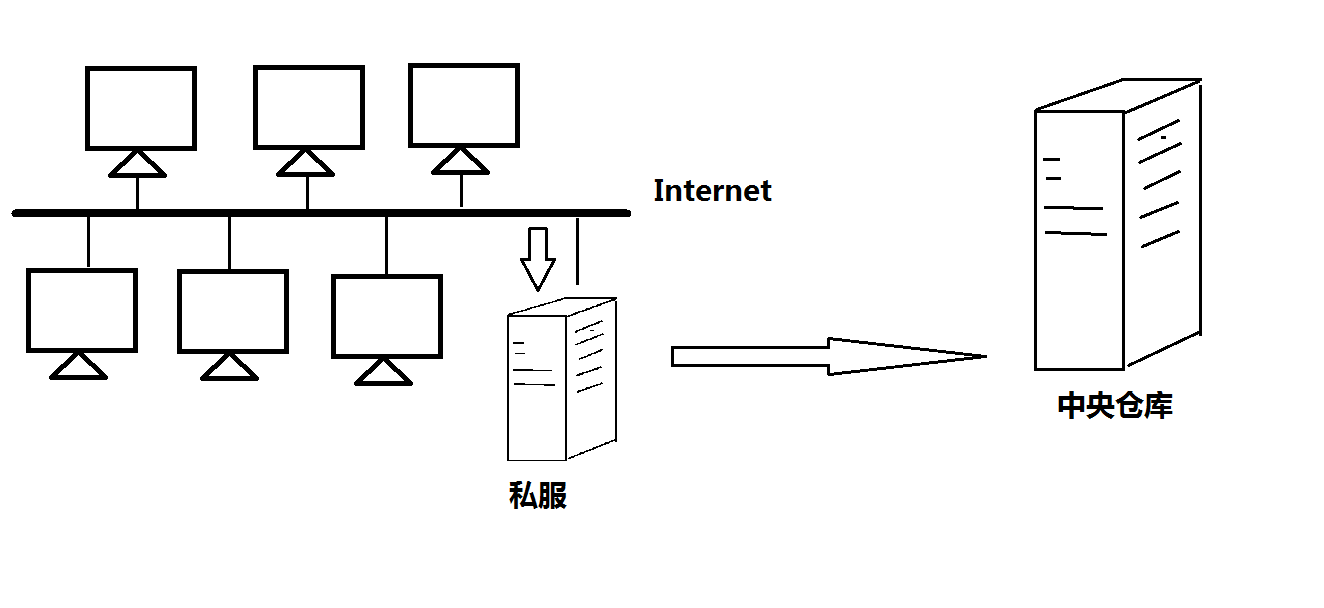
在国外，访问速度经常受限制。

* 1. 镜像仓库

内容和中央仓库保持一致，但是能够分担中央仓库的负载，同时让用户能够就近访问提高下载速度，例如：Nexus，aliyun，华为云

建议：不要中央仓库和阿里云镜像混用，否则 jar 包来源不纯，彼此冲突。

专门搜索 Maven 依赖信息的网站：https://mvnrepository.com/



### Maven私服-Nexus

为所有来自中央仓库的构建安装提供本地缓存。

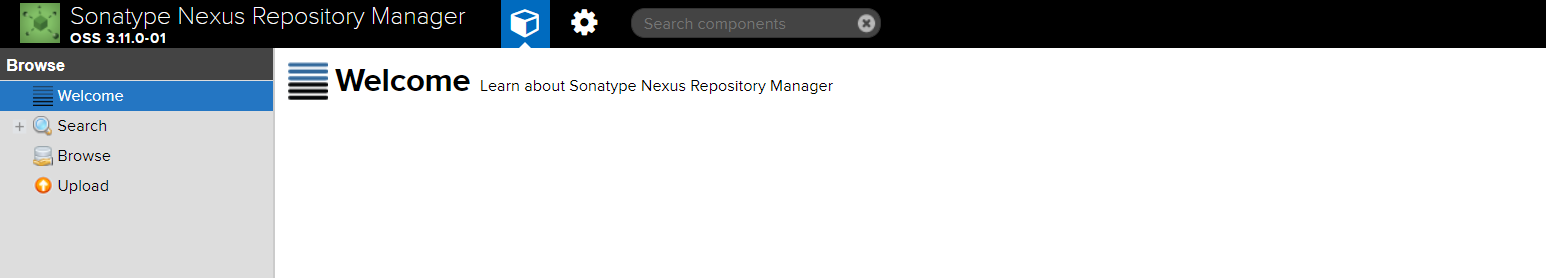
下载网站：<http://nexus.sonatype.org/>

Nexus安装

访问URL: <http://localhost:8081/nexus>

### 登陆

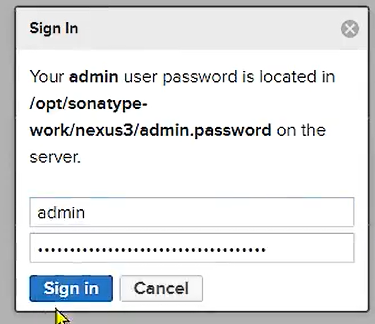
http://localhost:8081/nuxus



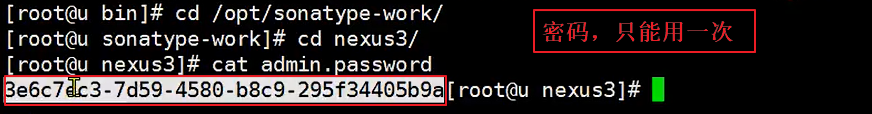


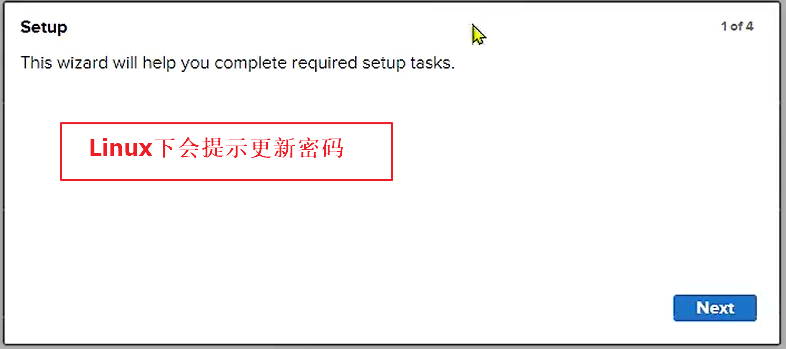
Windows下的用户名称和密码是admin,admin123,

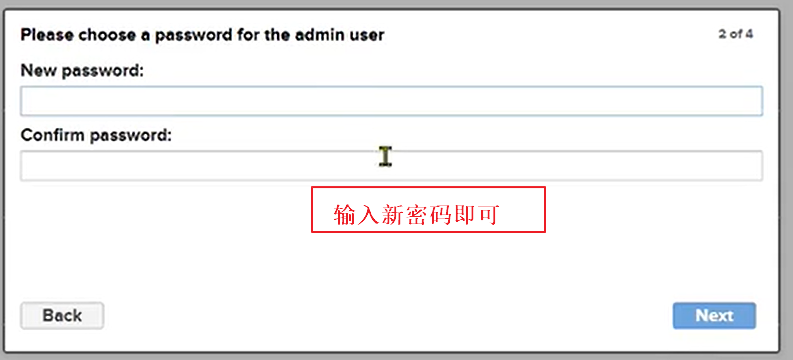
Linux下的密码在这个文件中

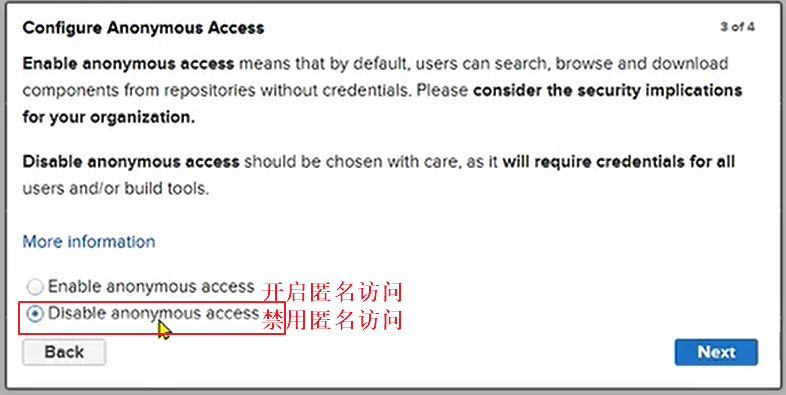


查看密码

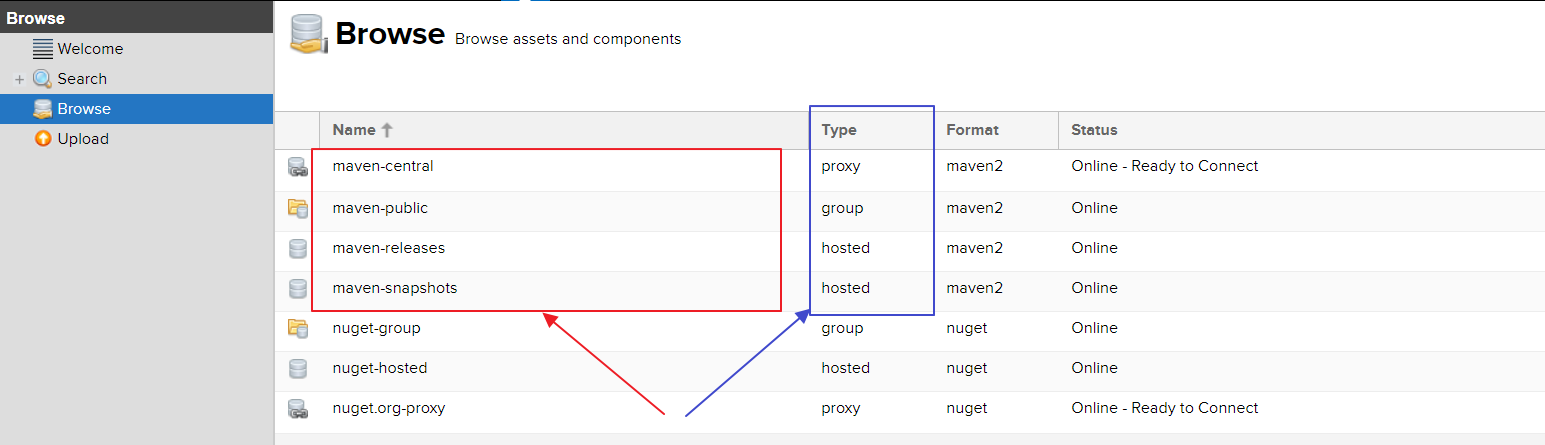








### 查看仓库



默认情况下，这些仓库都是空的。

### 仓库说明

| **仓库类型** | **说明** |
| --- | --- |
| proxy | 某个远程仓库的代理 |
| group | 存放：通过 Nexus 获取的第三方 jar 包 |
| hosted | 存放：本团队其他开发人员部署到 Nexus 的 jar 包 |

| **仓库名称** | **说明** |
| --- | --- |
| maven-central | Nexus 对 Maven 中央仓库的代理（通过它下载中央仓库的jar包） |
| maven-public | Nexus 默认创建，供开发人员下载使用的组仓库（我们使用的仓库） |
| maven-releasse | Nexus 默认创建，供开发人员部署自己 jar 包的宿主仓库 要求 releasse 版本（正式jar包） |
| maven-snapshots | Nexus 默认创建，供开发人员部署自己 jar 包的宿主仓库 要求 snapshots 版本（jar包快照版本） |

=============================注释掉=========================================

**仓库有4种类型 :**

* group(仓库组)：一组仓库的集合
* hosted(宿主)：配置第三方仓库 （包括公司内部私服 ）
* proxy(代理)：私服会对中央仓库进行代理，用户连接私服，私服自动去中央仓库下载jar包或者插件
* virtual(虚拟)：兼容Maven1 版本的jar或者插件

**Nexus的仓库和仓库组介绍:**

* 3rd party: 一个策略为Release的宿主类型仓库，用来部署无法从公共仓库获得的第三方发布版本构建
* Apache Snapshots: 一个策略为Snapshot的代理仓库，用来代理Apache Maven仓库的快照版本构建
* Central: 代理Maven中央仓库
* Central M1 shadow: 代理Maven1 版本 中央仓库
* Codehaus Snapshots: 一个策略为Snapshot的代理仓库，用来代理Codehaus Maven仓库的快照版本构件
* Releases: 一个策略为Release的宿主类型仓库，用来部署组织内部的发布版本构件
* Snapshots: 一个策略为Snapshot的宿主类型仓库，用来部署组织内部的快照版本构件
* **Public Repositories:该仓库组将上述所有策略为Release的仓库聚合并通过一致的地址提供服务**

===============================================================================

### 配置所有构建均从私服下载

在本地仓库的setting.xml中配置如下：

<mirror>

<id>nexus-mine</id>

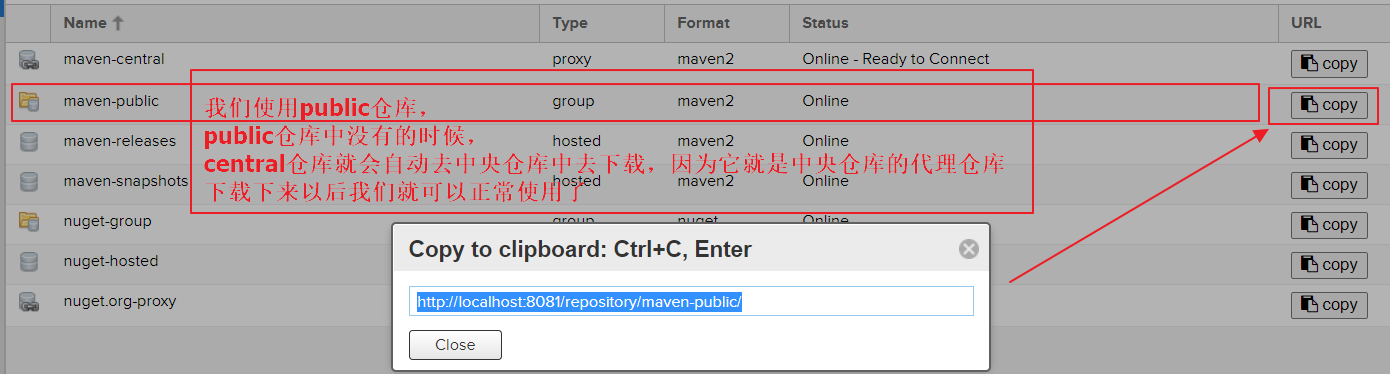
<mirrorOf>central</mirrorOf>

<name>Nexus mine</name>

<url>http://localhost:8081/repository/maven-public/</url>

</mirror>

这里的 url 标签是这么来的：



把上图中看到的地址复制出来即可。如果我们在前面允许了匿名访问，到这里就够了。但如果我们禁用了匿名访问，那么接下来我们还要继续配置 settings.xml：

在servers标签中配置

<server>

<id>nexus-mine</id>

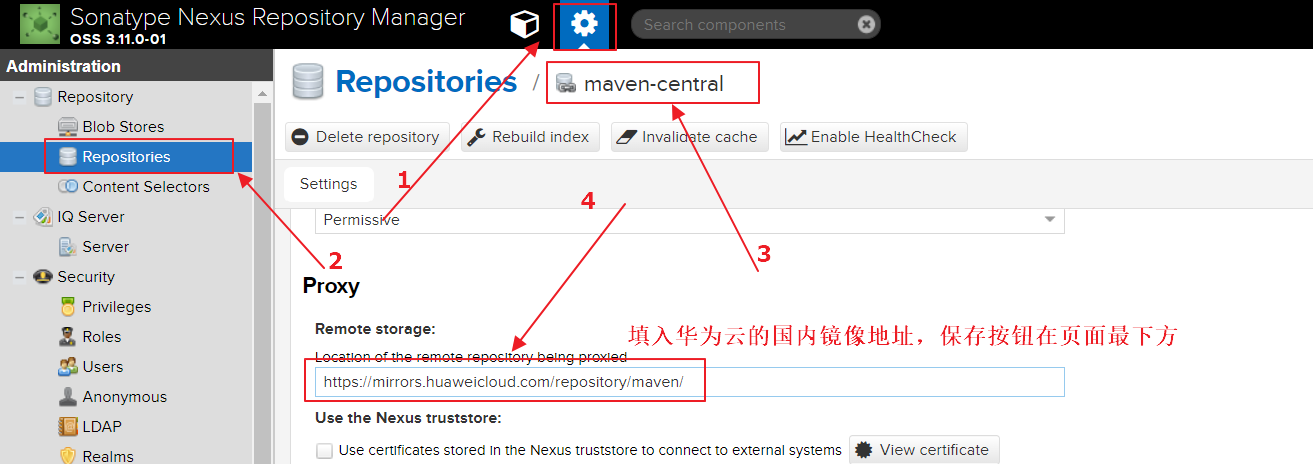
<username>admin</username>

<password>admin123</password>

</server>

这里需要**格外注意**：server 标签内的 id 标签值必须和 mirror 标签中的 id 值一样。

此时，nuxus是从中央仓库来下载jar包，速度非常慢，所以我们要设置私服从国内镜像下来，例如阿里云，华为云等。



#### 发布jar包到私服

配置 Maven 工程 ： 在父工程的dependencies标签外面给自己的maven工程添加上如下配置

<distributionManagement>

<snapshotRepository>

<id>nexus-mine</id>

<name>Nexus Snapshot</name>

<url> http://localhost:8081/repository/maven-snapshots/</url>

</snapshotRepository>

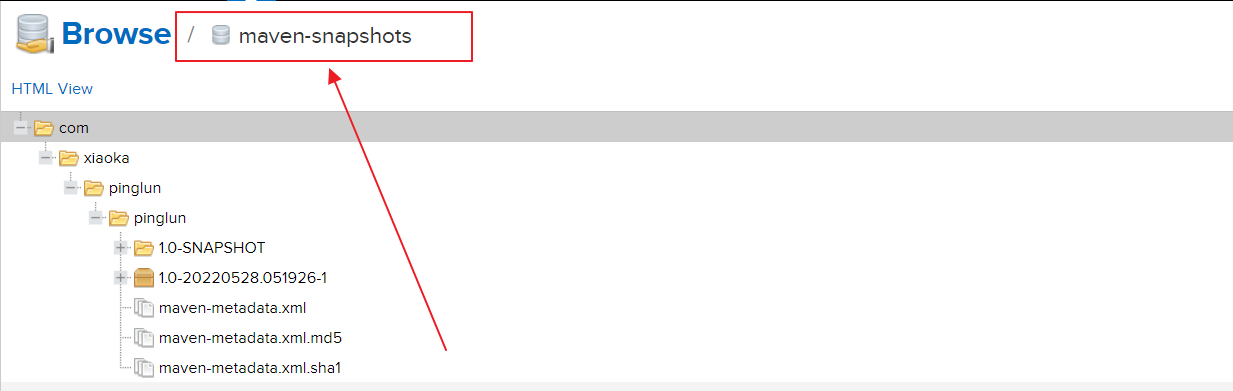
</distributionManagement>

这里 snapshotRepository 的 id 标签也必须和 settings.xml 中指定的 mirror 标签的 id 属性一致。

Url地址应该是私服的snapshots仓库的url，去私服中复制即可，和复制public仓库地址一个操作。

然后执行命令 ： mvn clean deploy

最后，在私服的快照仓库中可以看到自己的工程已经发布到私服。



#### 使用别人部署的jar包

我们知道 ：

* 默认访问的 Nexus 仓库：maven-public
* 存放别人部署 jar 包的仓库：maven-snapshots

这是两个不同的仓库，所以，要使用别人部署在私服的jar包，则需要在父工程中增加仓库的配置。

而这个配置实则非常简单

<repositories>

<repository>

<id>nexus-mine</id>

<name>nexus-mine</name>

<url>http://localhost:8081/repository/maven-snapshots/</url>

<snapshots>

<enabled>true</enabled>

</snapshots>

<releases>

<enabled>true</enabled>

</releases>

</repository>

</repositories>

#### 第三方jar包

在实际开发中确实有可能用到一些 jar 包并非是用 Maven 的方式发布，这种第三方的jar包是独立与maven体系之外的，那自然也没法通过 Maven 导入。例如包括一些人脸识别用的 jar 包、海康视频监控 jar 包等等。这个时候，就需要我们手动将该jar包构建成maven的方式来使用。

随便准备一个maven构建中不存在的jar包，比如使用eclipse生成一个。

将该 jar 包安装到 Maven 仓库

这里我们使用 install 插件的 install-file 目标：

mvn install:install-file -Dfile=[体系外 jar 包路径] \

-DgroupId=[给体系外 jar 包强行设定坐标] \

-DartifactId=[给体系外 jar 包强行设定坐标] \

-Dversion=1 \

-Dpackage=jar

注意（Windows 系统下使用 ^ 符号换行；Linux 系统用 \换行）：

随意在一个mavne项目中执行instll命令即可 ：

mvn install:install-file -Dfile=d:/kkk.jar ^ -DgroupId=com.xiaoka.kkk ^ -DartifactId=kkk ^ -Dversion=2.0 ^ -Dpackaging=jar

可以看到第三方jar包被安装到了本地仓库

# 重识POM

## POM的四个层次

* 超级 POM

Maven 在构建过程中有很多默认的设定。例如：源文件存放的目录、测试源文件存放的目录、构建输出的目录……等等。但是其实这些要素也都是被 Maven 定义过的。定义的位置就是：**超级 POM**。

所以我们自己的 POM 即使没有明确指定一个父工程（父 POM），其实也默认继承了超级 POM。就好比一个 Java 类默认继承了 Object 类。

超级 POM 中定义了哪些东西呢？一起来简单看一下。

<project>

<modelVersion>4.0.0</modelVersion>

<!--超级POM中定义了中央仓库，所以默认访问中央仓库-->

<repositories>

<repository>

<id>central</id>

<name>Central Repository</name>

<url>https://repo.maven.apache.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>central</id>

<name>Central Repository</name>

<url>https://repo.maven.apache.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<updatePolicy>never</updatePolicy>

</releases>

</pluginRepository>

</pluginRepositories>

<build>

<directory>${project.basedir}/target</directory>

<outputDirectory>${project.build.directory}/classes</outputDirectory>

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

<testOutputDirectory>${project.build.directory}/test-classes</testOutputDirectory>

<!--超级POM中定义了maven工程的各个文件的存放位置，所以maven工程的目录结构被"固定"了下来-->

<sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>

<scriptSourceDirectory>${project.basedir}/src/main/scripts</scriptSourceDirectory>

<testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>

<resources>

<resource>

<directory>${project.basedir}/src/main/resources</directory>

</resource>

</resources>

<testResources>

<testResource>

<directory>${project.basedir}/src/test/resources</directory>

</testResource>

</testResources>

<pluginManagement>

<!-- NOTE: These plugins will be removed from future versions of the super POM -->

<!-- They are kept for the moment as they are very unlikely to conflict with lifecycle mappings (MNG-4453) -->

<plugins>

<plugin>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.3</version>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<version>2.2-beta-5</version>

</plugin>

<plugin>

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

<version>2.8</version>

</plugin>

<plugin>

<artifactId>maven-release-plugin</artifactId>

<version>2.5.3</version>

</plugin>

</plugins>

</pluginManagement>

</build>

<reporting>

<outputDirectory>${project.build.directory}/site</outputDirectory>

</reporting>

<profiles>

<!-- NOTE: The release profile will be removed from future versions of the super POM -->

<profile>

<id>release-profile</id>

<activation>

<property>

<name>performRelease</name>

<value>true</value>

</property>

</activation>

<build>

<plugins>

<plugin>

<inherited>true</inherited>

<artifactId>maven-source-plugin</artifactId>

<executions>

<execution>

<id>attach-sources</id>

<goals>

<goal>jar-no-fork</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<inherited>true</inherited>

<artifactId>maven-javadoc-plugin</artifactId>

<executions>

<execution>

<id>attach-javadocs</id>

<goals>

<goal>jar</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<inherited>true</inherited>

<artifactId>maven-deploy-plugin</artifactId>

<configuration>

<updateReleaseInfo>true</updateReleaseInfo>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

</project>

* 父 POM

和 Java 类一样，POM 之间其实也是**单继承**的。如果我们给一个 POM 指定了父 POM，那么继承关系如下



* 当前POM

当前工程/模块中的POM文件

* 有效 POM

有效 POM 英文翻译为 effective POM，它的概念是这样的——在 POM 的继承关系中，子 POM 可以覆盖父 POM 中的配置；如果子 POM 没有覆盖，那么父 POM 中的配置将会被继承。按照这个规则，继承关系中的所有 POM 叠加到一起，就得到了一个最终生效的 POM。显然 Maven 实际运行过程中，执行构建操作就是按照这个最终生效的 POM 来运行的。这个最终生效的 POM 就是**有效 POM**，英文叫**effective POM**。

查看有效 POM

mvn help:effective-pom

## 小结

综上所述，平时我们使用和配置的 POM 其实大致是由四个层次组成的：

* 超级 POM：所有 POM 默认继承，只是有直接和间接之分。
* 父 POM：这一层可能没有，可能有一层，也可能有很多层。
* 当前 pom.xml 配置的 POM：我们最多关注和最多使用的一层。
* 有效 POM：隐含的一层，但是实际上真正生效的一层。

## 属性的声明与查看

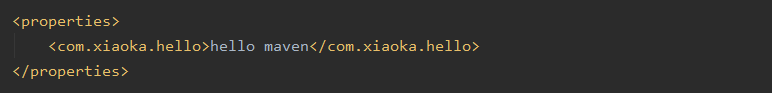
Pom.xml文件中声明了非常多的属性，我们可以通过help插件来查看这些属性和它的值。

Maven的help插件具体有如下几个

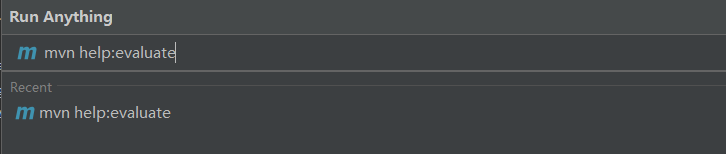
| **目标** | **说明** |
| --- | --- |
| help:active-profiles | 列出当前已激活的 profile |
| help:all-profiles | 列出当前工程所有可用 profile |
| help:describe | 描述一个插件和/或 Mojo 的属性 |
| help:effective-pom | 以 XML 格式展示有效 POM |
| help:effective-settings | 为当前工程以 XML 格式展示计算得到的 settings 配置 |
| **help:evaluate** | 计算用户在交互模式下给出的 Maven 表达式 |
| help:system | 显示平台详细信息列表，如系统属性和环境变量 |

## 使用 help:evaluate 查看属性值

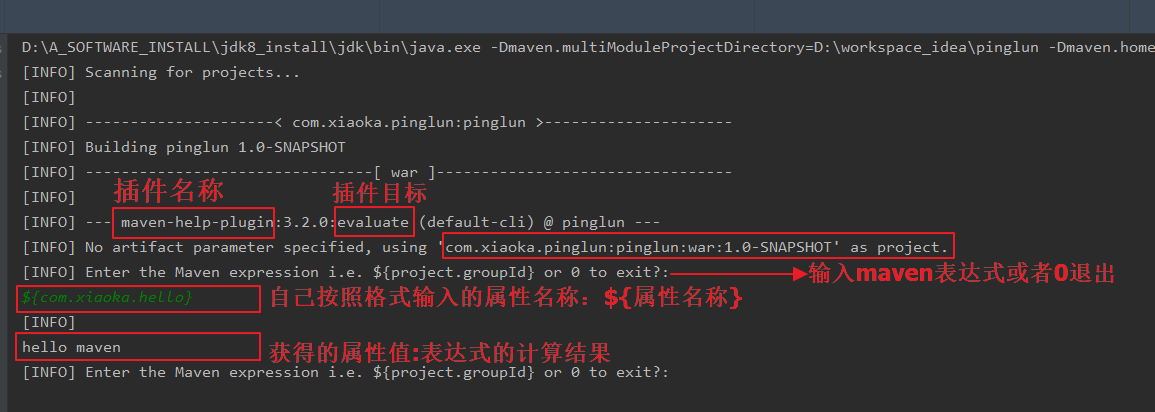
### 定义属性



### 运行命令



### 查看运行结果



### 使用 Maven 访问系统属性

首先，我们可以使用Java代码来查看系统的各个属性

Properties properties = System.getProperties();

Set<Object> propNameSet = properties.keySet();

for (Object propName : propNameSet) {

String propValue = properties.getProperty((String) propName);

System.out.println(propName + " = " + propValue);

}

运行结果

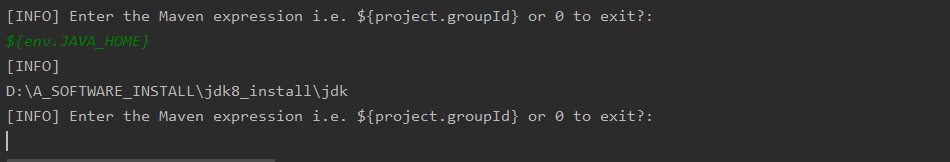
java.runtime.name = Java(TM) SE Runtime Environment   
sun.boot.library.path = D:\software\Java\jre\bin   
java.vm.version = 25.141-b15   
java.vm.vendor = Oracle Corporation   
java.vendor.url = http://java.oracle.com/   
path.separator = ;   
java.vm.name = Java HotSpot(TM) 64-Bit Server VM   
file.encoding.pkg = sun.io   
user.country = CN   
user.script =   
sun.java.launcher = SUN\_STANDARD   
sun.os.patch.level =   
java.vm.specification.name = Java Virtual Machine Specification   
user.dir = D:\idea2019workspace\atguigu-maven-test-prepare   
java.runtime.version = 1.8.0\_141-b15   
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment   
java.endorsed.dirs = D:\software\Java\jre\lib\endorsed   
os.arch = amd64   
java.io.tmpdir = C:\Users\ADMINI~1\AppData\Local\Temp\   
line.separator =   
java.vm.specification.vendor = Oracle Corporation   
user.variant =   
os.name = Windows 10   
sun.jnu.encoding = GBK   
java.library.path = D:\software\Java\bin;C:\WINDOWS\Sun\Java\bin;C:\WIN……   
java.specification.name = Java Platform API Specification   
java.class.version = 52.0   
sun.management.compiler = HotSpot 64-Bit Tiered Compilers   
os.version = 10.0   
user.home = C:\Users\Administrator   
user.timezone =   
java.awt.printerjob = sun.awt.windows.WPrinterJob   
file.encoding = UTF-8   
java.specification.version = 1.8   
java.class.path = D:\software\Java\jre\lib\charsets.jar;D:\softw……   
user.name = Administrator   
java.vm.specification.version = 1.8   
sun.java.command = com.atguigu.maven.MyTest   
java.home = D:\software\Java\jre   
sun.arch.data.model = 64   
user.language = zh   
java.specification.vendor = Oracle Corporation   
awt.toolkit = sun.awt.windows.WToolkit   
java.vm.info = mixed mode   
java.version = 1.8.0\_141   
java.ext.dirs = D:\software\Java\jre\lib\ext;C:\WINDOWS\Sun\Java\lib\ext   
sun.boot.class.path = D:\software\Java\jre\lib\resources.jar;D:\sof……   
java.vendor = Oracle Corporation   
file.separator = \   
java.vendor.url.bug = http://bugreport.sun.com/bugreport/   
sun.io.unicode.encoding = UnicodeLittle   
sun.cpu.endian = little   
sun.desktop = windows   
sun.cpu.isalist = amd64

当然，我们也可以通过maven来查看系统的属性和值



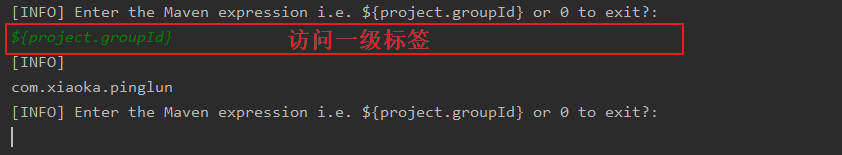
可以看到，和使用Java代码获取到的结果一样。

### 访问系统环境变量



### 访问 project 属性

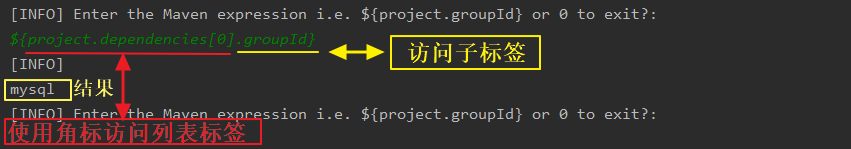
使用以project开始的表达式 ${project.xxx} 可以访问当前 POM 中的元素值。



使用下列表达式格式访问列表标签和子标签

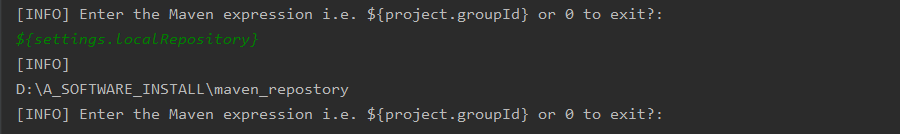
${project.标签名.子标签名}

${project.标签名[下标]}



### 访问 settings 全局配置

使用以settings开始的表达式，${settings.标签名} 可以访问 settings.xml 中配置的元素值。



### 有啥用

定义属性，查看属性，这些都有啥用？主要作用就体现在两个方面：

* 在当前 pom.xml 文件中引用属性

例如自定义属性，然后被引用属性值，实现一处修改，处处生效的效果（父工程抽取版本号）

* 资源过滤功能

在非 Maven 配置文件中引用属性，由 Maven 在处理资源时将引用属性的表达式替换为属性值，由于各个框架都有该功能，所以这里作了解即可。

## Build标签详解

在实际使用 Maven 的过程中，我们会发现 build 标签有时候有，有时候没，这是怎么回事呢？其实通过有效 POM 我们能够看到，build 标签的相关配置其实一直都在，只是在我们需要定制构建过程的时候才会通过配置 build 标签覆盖默认值或补充配置。这一点我们可以通过打印有效 POM 来看到。

所以**本质**上来说：我们配置的 build 标签都是对**超级 POM 配置**的**叠加**。那我们又为什么要在默认配置的基础上叠加呢？很简单，在默认配置无法满足需求的时候**定制构建过程**。

Build标签完整实例

<build>

<sourceDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\src\main\java</sourceDirectory>

<scriptSourceDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\src\main\scripts</scriptSourceDirectory>

<testSourceDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\src\test\java</testSourceDirectory>

<outputDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\target\classes</outputDirectory>

<testOutputDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\target\test-classes</testOutputDirectory>

<resources>

<resource>

<directory>D:\idea2019workspace\atguigu-maven-test-prepare\src\main\resources</directory>

</resource>

</resources>

<testResources>

<testResource>

<directory>D:\idea2019workspace\atguigu-maven-test-prepare\src\test\resources</directory>

</testResource>

</testResources>

<directory>D:\idea2019workspace\atguigu-maven-test-prepare\target</directory>

<finalName>atguigu-maven-test-prepare-1.0-SNAPSHOT</finalName>

<pluginManagement>

<plugins>

<plugin>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.3</version>

</plugin>

<plugin>

<artifactId>maven-assembly-plugin</artifactId>

<version>2.2-beta-5</version>

</plugin>

<plugin>

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

<version>2.8</version>

</plugin>

<plugin>

<artifactId>maven-release-plugin</artifactId>

<version>2.5.3</version>

</plugin>

</plugins>

</pluginManagement>

<plugins>

<plugin>

<artifactId>maven-clean-plugin</artifactId>

<version>2.5</version>

<executions>

<execution>

<id>default-clean</id>

<phase>clean</phase>

<goals>

<goal>clean</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-resources-plugin</artifactId>

<version>2.6</version>

<executions>

<execution>

<id>default-testResources</id>

<phase>process-test-resources</phase>

<goals>

<goal>testResources</goal>

</goals>

</execution>

<execution>

<id>default-resources</id>

<phase>process-resources</phase>

<goals>

<goal>resources</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-jar-plugin</artifactId>

<version>2.4</version>

<executions>

<execution>

<id>default-jar</id>

<phase>package</phase>

<goals>

<goal>jar</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<executions>

<execution>

<id>default-compile</id>

<phase>compile</phase>

<goals>

<goal>compile</goal>

</goals>

</execution>

<execution>

<id>default-testCompile</id>

<phase>test-compile</phase>

<goals>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.12.4</version>

<executions>

<execution>

<id>default-test</id>

<phase>test</phase>

<goals>

<goal>test</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-install-plugin</artifactId>

<version>2.4</version>

<executions>

<execution>

<id>default-install</id>

<phase>install</phase>

<goals>

<goal>install</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-deploy-plugin</artifactId>

<version>2.7</version>

<executions>

<execution>

<id>default-deploy</id>

<phase>deploy</phase>

<goals>

<goal>deploy</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<artifactId>maven-site-plugin</artifactId>

<version>3.3</version>

<executions>

<execution>

<id>default-site</id>

<phase>site</phase>

<goals>

<goal>site</goal>

</goals>

<configuration>

<outputDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\target\site</outputDirectory>

<reportPlugins>

<reportPlugin>

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

<artifactId>maven-project-info-reports-plugin</artifactId>

</reportPlugin>

</reportPlugins>

</configuration>

</execution>

<execution>

<id>default-deploy</id>

<phase>site-deploy</phase>

<goals>

<goal>deploy</goal>

</goals>

<configuration>

<outputDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\target\site</outputDirectory>

<reportPlugins>

<reportPlugin>

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

<artifactId>maven-project-info-reports-plugin</artifactId>

</reportPlugin>

</reportPlugins>

</configuration>

</execution>

</executions>

<configuration>

<outputDirectory>D:\idea2019workspace\atguigu-maven-test-prepare\target\site</outputDirectory>

<reportPlugins>

<reportPlugin>

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

<artifactId>maven-project-info-reports-plugin</artifactId>

</reportPlugin>

</reportPlugins>

</configuration>

</plugin>

</plugins>

</build>

### Build标签的组成

从完整示例中我们能够看到，build 标签的子标签大致包含三个主体部分：

1. 定义约定的目录结构



我们能看到各个目录的作用如下：

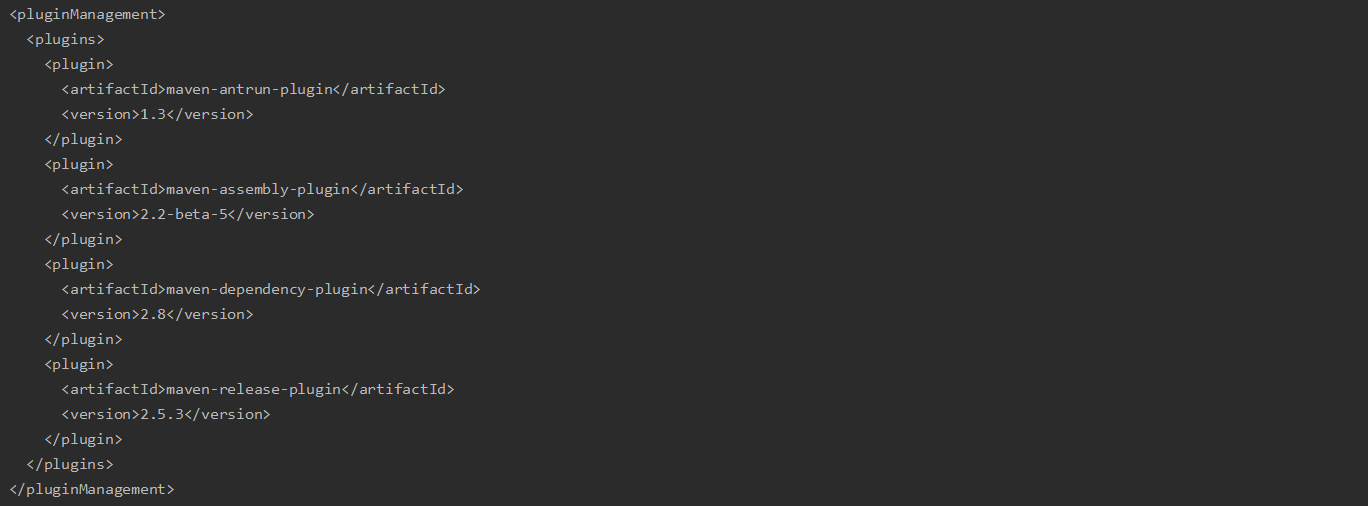
| **目录名** | **作用** |
| --- | --- |
| sourceDirectory | 主体源程序存放目录 |
| scriptSourceDirectory | 脚本源程序存放目录 |
| testSourceDirectory | 测试源程序存放目录 |
| outputDirectory | 主体源程序编译结果输出目录 |
| testOutputDirectory | 测试源程序编译结果输出目录 |
| resources | 主体资源文件存放目录 |
| testResources | 测试资源文件存放目录 |
| directory | 构建结果输出目录 |

1. 备用插件管理

pluginManagement 标签存放着几个极少用到的插件：

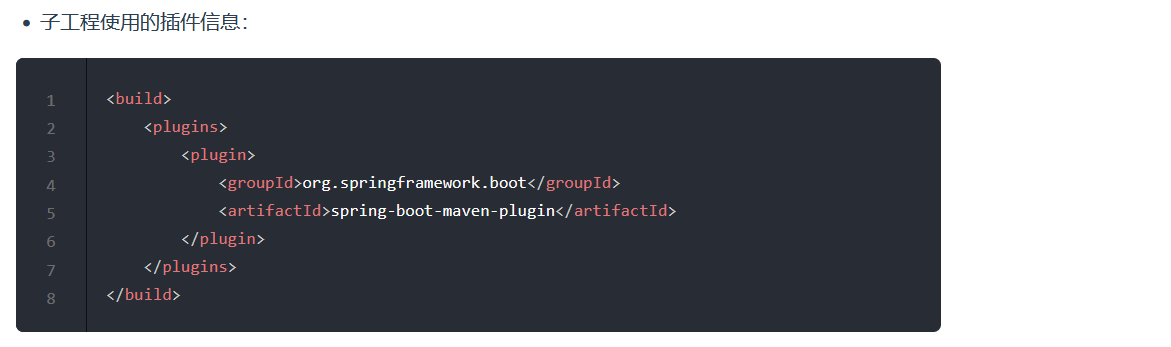
* maven-antrun-plugin
* maven-assembly-plugin
* maven-dependency-plugin
* maven-release-plugin

通过 pluginManagement 标签管理起来的插件就像 dependencyManagement 一样，子工程使用时可以省略版本号，起到在父工程中统一管理版本的效果



请看下面例子





1. 生命周期插件

plugins 标签存放的是默认生命周期中实际会用到的插件，这些插件想必大家都不陌生，所以抛开插件本身不谈，我们来看看一个 plugin 标签的结构：



#### 坐标部分

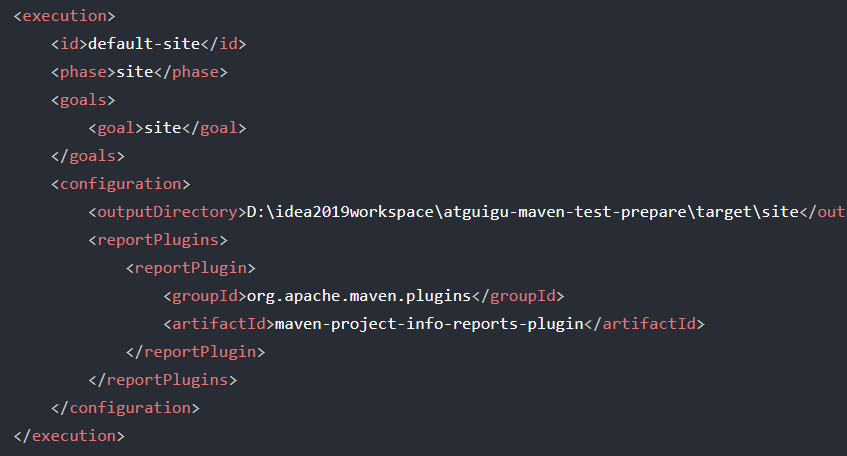
artifactId 和 version 标签定义了插件的坐标，作为 Maven 的自带插件这里省略了 groupId。

#### 执行部分

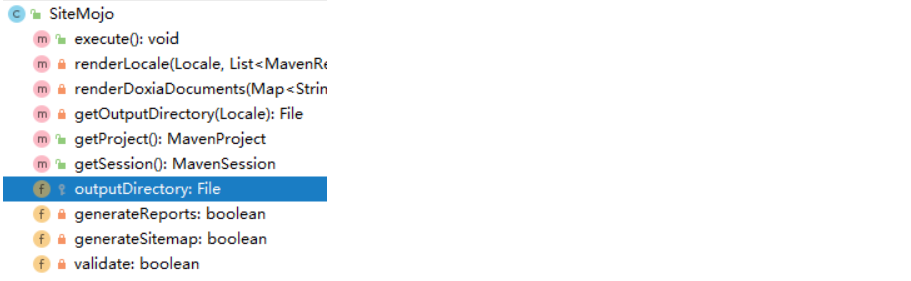
executions 标签内可以配置多个 execution 标签，execution 标签内：

* id：指定唯一标识
* phase：关联的生命周期阶段
* goals/goal：关联指定生命周期的目标
  + goals 标签中可以配置多个 goal 标签，表示一个生命周期环节可以对应当前插件的多个目标。

另外，插件目标的执行过程可以进行配置，例如 maven-site-plugin 插件的 site 目标：



configuration 标签内进行配置时使用的标签是插件本身定义的。就以 maven-site-plugin 插件为例，它的核心类是 org.apache.maven.plugins.site.render.SiteMojo，在这个类中我们看到了 outputDirectory 属性：



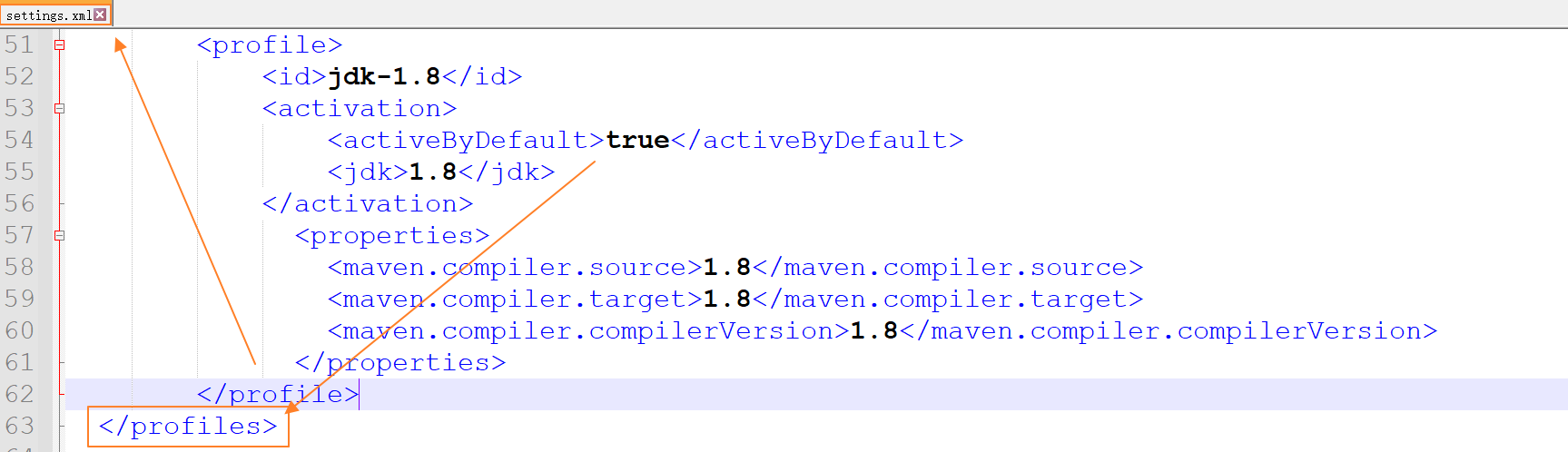
SiteMojo 的父类是：AbstractSiteRenderingMojo，在父类中我们看到 reportPlugins 属性：



**结论**：每个插件能够做哪些设置都是各个插件自己规定的，无法一概而论。

### Build标签应用-配置Maven工程JDK的版本

第一种方式 ：在配置文件settings.xml文件中指定



图中的配置内容 ：

<profile>

<!—ID标签，唯一标识符-->

<id>jdk-1.8</id>

<!—activation标签：激活方式-->

<activation>

<!-- 配置是否默认激活 -->

<activeByDefault>true</activeByDefault>

<!—jdk标签：标识当前profile标签可以根据JDK版本来激活-->

<jdk>1.8</jdk>

</activation>

<!—其它标签 ：当前profile被激活后采用的配置-->

<properties>

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

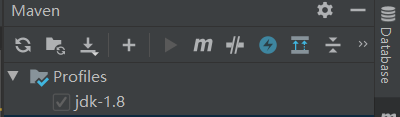
<maven.compiler.target>1.8</maven.compiler.target>

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

</properties>

</profile>

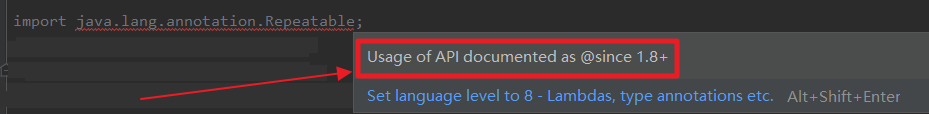
在IDEA中可以看到，我们使用的是JDK1.8版本

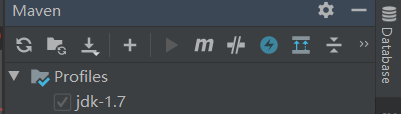


验证我们确实使用的JDK1.8版本

随便找一个Java类，导入import java.lang.annotation.Repeatable;

因为该注解是1.8新增的，所以不此时不会报错。然后修改settings.xml中的配置，将配置jdk版本修改为1.7及以下，或者干脆注释掉（maven默认使用jdk5），可以看到此时编译报错了。





第二种方式 ： 使用build标签配置构建过程

<!-- build 标签：意思是告诉 Maven，你的构建行为，我要开始定制了！ -->

<build>

<!-- plugins 标签：Maven 你给我听好了，你给我构建的时候要用到这些插件！ -->

<plugins>

<!-- plugin 标签：这是我要指定的一个具体的插件 -->

<plugin>

<!-- 插件的坐标。此处引用的 maven-compiler-plugin 插件不是第三方的，是一个 Maven 自带的插件。 -->

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<!-- configuration 标签：配置 maven-compiler-plugin 插件 -->

<configuration>

<!-- 具体配置信息会因为插件不同、需求不同而有所差异 -->

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

为了验证build标签配置生效，我们故意让settings.xml中JDK版本配置为1.7，同时添加build标签，指定使用JDK版本为1.8，我们可以看到，虽然IDEA中显示使用1.7，但是导入的1.8特有的注解不再编译报错了。



但是编译不报错了，证明在build中配置的1.8生效了



两种配置方式比较

* settings.xml 中配置：仅在本地生效，如果脱离当前 settings.xml 能够覆盖的范围，则无法生效。
* 在当前 Maven 工程 pom.xml 中配置：无论在哪个环境执行编译等构建操作都有效

### Build标签应用-配置Mybatis逆向工程

使用 Mybatis 的逆向工程需要使用如下配置，MBG 插件的特点是需要提供插件所需的依赖：

<!-- 控制 Maven 在构建过程中相关配置 -->

<build>

<!-- 构建过程中用到的插件 -->

<plugins>

<!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->

<plugin>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-maven-plugin</artifactId>

<version>1.3.0</version>

<!-- 插件的依赖 -->

<dependencies>

<!-- 逆向工程的核心依赖 -->

<dependency>

<groupId>org.mybatis.generator</groupId>

<artifactId>mybatis-generator-core</artifactId>

<version>1.3.2</version>

</dependency>

<!-- 数据库连接池 -->

<dependency>

<groupId>com.mchange</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.2</version>

</dependency>

<!-- MySQL驱动 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.8</version>

</dependency>

</dependencies>

</plugin>

</plugins>

</build>

### Build标签应用-SpringBoot 定制化打包

很显然 spring-boot-maven-plugin 并不是 Maven 自带的插件，而是 SpringBoot 提供的，用来改变 Maven 默认的构建行为。具体来说是改变打包的行为。默认情况下 Maven 调用 maven-jar-plugin 插件的 jar 目标，生成普通的 jar 包。

普通 jar 包没法使用 java -jar xxx.jar 这样的命令来启动、运行，但是 SpringBoot 的设计理念就是每一个『微服务』导出为一个 jar 包，这个 jar 包可以使用 java -jar xxx.jar 这样的命令直接启动运行。

这样一来，打包的方式肯定要进行调整。所以 SpringBoot 提供了 spring-boot-maven-plugin 这个插件来定制打包行为。

所有的一切已经都被 SpringBoot 封装好了，所以配置非常简单，提供插件坐标即可。

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

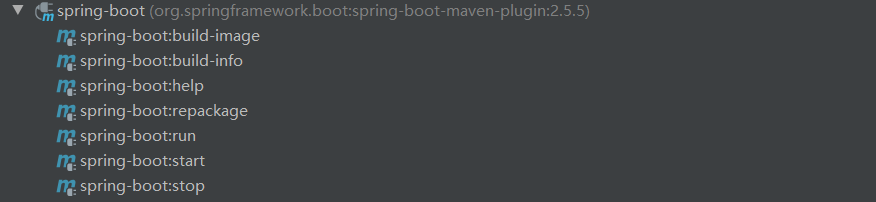
<version>2.5.5</version>

</plugin>

</plugins>

</build>

插件的七个目标



### 小结

通常需要用到 build 标签的时候底层都会帮我们封装好，需要我们配置的地方不多。即使有些地方需要我们配置，也不会真的我们自己去写，把现成的案例复制过来就行。

所以对 build 标签来说，我们的掌握要求就是：**能大致看懂**就行。

## Profile标签详解

通常情况下，我们至少有三种运行环境：

* 开发环境：供不同开发工程师开发的各个模块之间互相调用、访问；内部使用
* 测试环境：供测试工程师对项目的各个模块进行功能测试；内部使用
* 生产环境：供最终用户访问——所以这是正式的运行环境，对外提供服务

而我们这里的『环境』仍然只是一个笼统的说法，实际工作中一整套运行环境会包含很多种不同服务器：

* MySQL
* Redis
* ElasticSearch
* RabbitMQ
* FastDFS
* Nginx
* Tomcat
* ……

就拿其中的 MySQL 来说，不同环境下的访问参数肯定完全不同：

| **开发环境** | **测试环境** | **生产环境** |
| --- | --- | --- |
| dev.driver=com.mysql.jdbc.Driver dev.url=jdbc:mysql://124.71.36.17:3306/db-sys dev.username=root dev.password=atguigu | test.driver=com.mysql.jdbc.Driver test.url=jdbc:mysql://124.71.36.89:3306/db-sys test.username=dev-team test.password=atguigu | product.driver=com.mysql.jdbc.Driver product.url=jdbc:mysql://39.107.88.164:3306/prod-db-sys product.username=root product.password=atguigu |

可是代码只有一套。如果在 jdbc.properties 里面来回改，那就太麻烦了，而且很容易遗漏或写错，增加调试的难度和工作量。所以最好的办法就是把适用于各种不同环境的配置信息分别准备好，部署哪个环境就激活哪个配置。

在 Maven 中，使用 profile 机制来管理不同环境下的配置信息。但是解决同类问题的类似机制在其他框架中也有，而且从模块划分的角度来说，持久化层的信息放在构建工具中配置也违反了『高内聚，低耦合』的原则。

所以 Maven 的 profile 我们了解一下即可，不必深究。

### profile 声明和使用的基本逻辑

* 首先为每一个环境声明一个 profile
  + 环境 A：profile A
  + 环境 B：profile B
  + 环境 C：profile C
  + ……
* 然后激活某一个 profile

### 默认 profile

其实即使我们在 pom.xml 中不配置 profile 标签，也已经用到 profile了。为什么呢？因为根标签 project 下所有标签相当于都是在设定默认的 profile。这样一来我们也就很容易理解下面这句话：project 标签下除了 modelVersion 和坐标标签之外，其它标签都可以配置到 profile 中。

### profile 配置

**外部视角：配置文件**

外部视角来看，profile 可以在下面两种配置文件中配置：

* settings.xml：全局生效。其中我们最熟悉的就是配置 JDK 1.8。
* pom.xml：当前 POM 生效

**内部实现：具体标签**

从内部视角来看，配置 profile 有如下语法要求

**profiles/profile 标签**

* 由于 profile 天然代表众多可选配置中的一个所以由复数形式的 profiles 标签统一管理。
* 由于 profile 标签覆盖了 pom.xml 中的默认配置，所以 profiles 标签通常是 pom.xml 中的最后一个标签。

**id 标签**

每个 profile 都必须有一个 id 标签，指定该 profile 的唯一标识。这个 id 标签的值会在命令行调用 profile 时被用到。这个命令格式是：-P<profile id>。

**其它允许出现的标签**

一个 profile 可以覆盖项目的最终名称、项目依赖、插件配置等各个方面以影响构建行为。

* build
  + defaultGoal
  + finalName
  + resources
  + testResources
  + plugins
* reporting
* modules
* dependencies
* dependencyManagement
* repositories
* pluginRepositories
* properties

### 激活 profile

**默认配置默认被激活**

前面提到了，POM 中没有在 profile 标签里的就是默认的 profile，当然默认被激活。

**基于环境信息激活**

环境信息包含：JDK 版本、操作系统参数、文件、属性等各个方面。一个 profile 一旦被激活，那么它定义的所有配置都会覆盖原来 POM 中对应层次的元素。大家可以参考下面的标签结构：

<profile>

<!—ID标签，唯一标识符-->

<id>dev</id>

<!—activation标签：激活方式-->

<activation>

<!-- 配置是否默认激活 -->

<activeByDefault>false</activeByDefault>

<!—下面是多个激活条件

可以根据jdk版本激活

可以根据os系统标签版本激活

可以根据属性激活

也可以根据某个文件激活

-->

<jdk>1.5</jdk>

<os>

<name>Windows XP</name>

<family>Windows</family>

<arch>x86</arch>

<version>5.1.2600</version>

</os>

<property>

<name>mavenVersion</name>

<value>2.0.5</value>

</property>

<file>

<exists>file2.properties</exists>

<missing>file1.properties</missing>

</file>

</activation>

<!—其它标签 ：当前profile被激活后采用的配置-->

</profile>

这里有个问题是：多个激活条件之间是什么关系呢？

* Maven **3.2.2 之前**：遇到第一个满足的条件即可激活——**或**的关系。

即上列多个条件满足一个就激活当前profile

* Maven **3.2.2 开始**：各条件均需满足——**且**的关系。

即上列多个条件全部满足才可以激活当前profile.

现在假设有如下 profile ，配置在 JDK 版本为 1.6 时被激活，那么不需要将版本号指定到末尾，只需要指定到大版本号1.6即可。

Maven 会自动检测当前环境安装的 JDK 版本，只要 JDK 版本是以 1.6 开头都算符合激活条件。下面几个JDK版本都符合激活条件：

* 1.6.0\_03
* 1.6.0\_02
* ……

<profiles>

<profile>

<id>JDK1.6</id>

<activation>

<!-- 指定激活条件为：JDK 1.6 -->

<jdk>1.6</jdk>

</activation>

……

</profile>

</profiles>

**命令行激活**

1. **列出活动的 profile**

# 列出所有激活的 profile，以及它们在哪里定义

mvn help:active-profiles

1. **指定某个具体 profile**

mvn 命令 -P<profile id>

1. **操作演示命令激活某个profile**

随便找一个Java类创建一个集合

ArrayList<Object> objects = new ArrayList<>();

记住，前边有泛型，后边没有泛型，这种运算符号叫做diamond运算符号。这种写法是jdk1.7以前不允许的，现在我们配置一个profile使用jdk1.6,那么编译就一定会报错。

Diamond运算符简介 ：

diamond（钻石）运算符是java 1.7 以上版本的新特性之一，其目的是为了增强对通用实例创建的类型判断，简化泛型类的实例化：

如java 1.7以下实例化必须将泛型类填写完整：List<Integer> p = new ArrayList<Integer>();

而在java1.7中则可以省略泛型类：List<Integer> p = new ArrayList<>();

在1.7版本中通过diamond运算符<>从引用的声明中推断类型，从而简化代码。

在当前工程中新增一个profile配置

在pom.xml的最后添加如下配置，使用jdk1.6

<profiles>

<profile>

<id>myJDKProfile</id>

<!-- build 标签：意思是告诉 Maven，你的构建行为，我要开始定制了！ -->

<build>

<!-- plugins 标签：Maven 你给我听好了，你给我构建的时候要用到这些插件！ -->

<plugins>

<!-- plugin 标签：这是我要指定的一个具体的插件 -->

<plugin>

<!-- 插件的坐标。此处引用的 maven-compiler-plugin 插件不是第三方的，是一个 Maven 自带的插件。 -->

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

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<!-- configuration 标签：配置 maven-compiler-plugin 插件 -->

<configuration>

<!-- 具体配置信息会因为插件不同、需求不同而有所差异 -->

<source>1.6</source>

<target>1.6</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</profile>

</profiles>

**执行激活命令**



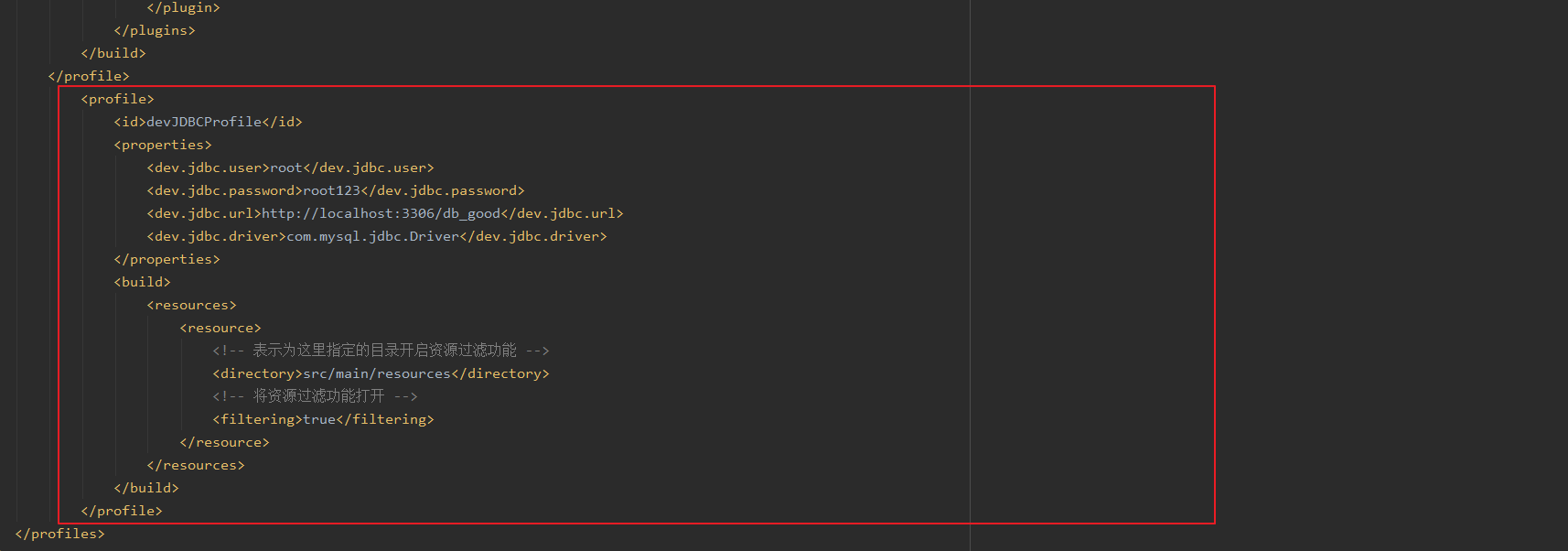
执行测试命令，可以看到，编译失败，而将刚刚的profile的配置修改为1.7及以上，再次执行命令，就可以看到编译通过了。

### 资源属性过滤

前面介绍了Profile是多个不同的运行环境，为了能够在多个环境种进行切换，Maven提供了一种『资源属性过滤』的机制，实现各不同运行环境切换。具体则是通过属性替换实现不同环境下使用不同的参数。

**操作演示**

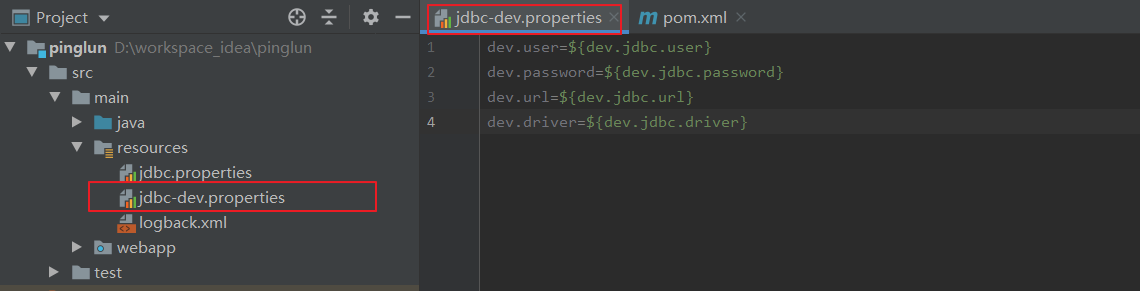
1.配置profile



配置内容如下

<profile>  
 <id>devJDBCProfile</id>  
 <properties>  
 <dev.jdbc.user>root</dev.jdbc.user>  
 <dev.jdbc.password>root123</dev.jdbc.password>  
 <dev.jdbc.url>http://localhost:3306/db\_good</dev.jdbc.url>  
 <dev.jdbc.driver>com.mysql.jdbc.Driver</dev.jdbc.driver>  
 </properties>  
 <build>  
 <resources>  
 <resource>  
 <!-- 表示为这里指定的目录开启资源过滤功能 -->  
 <directory>src/main/resources</directory>  
 <!-- 将资源过滤功能打开 -->  
 <filtering>true</filtering>  
 </resource>  
 </resources>  
 </build>  
</profile>

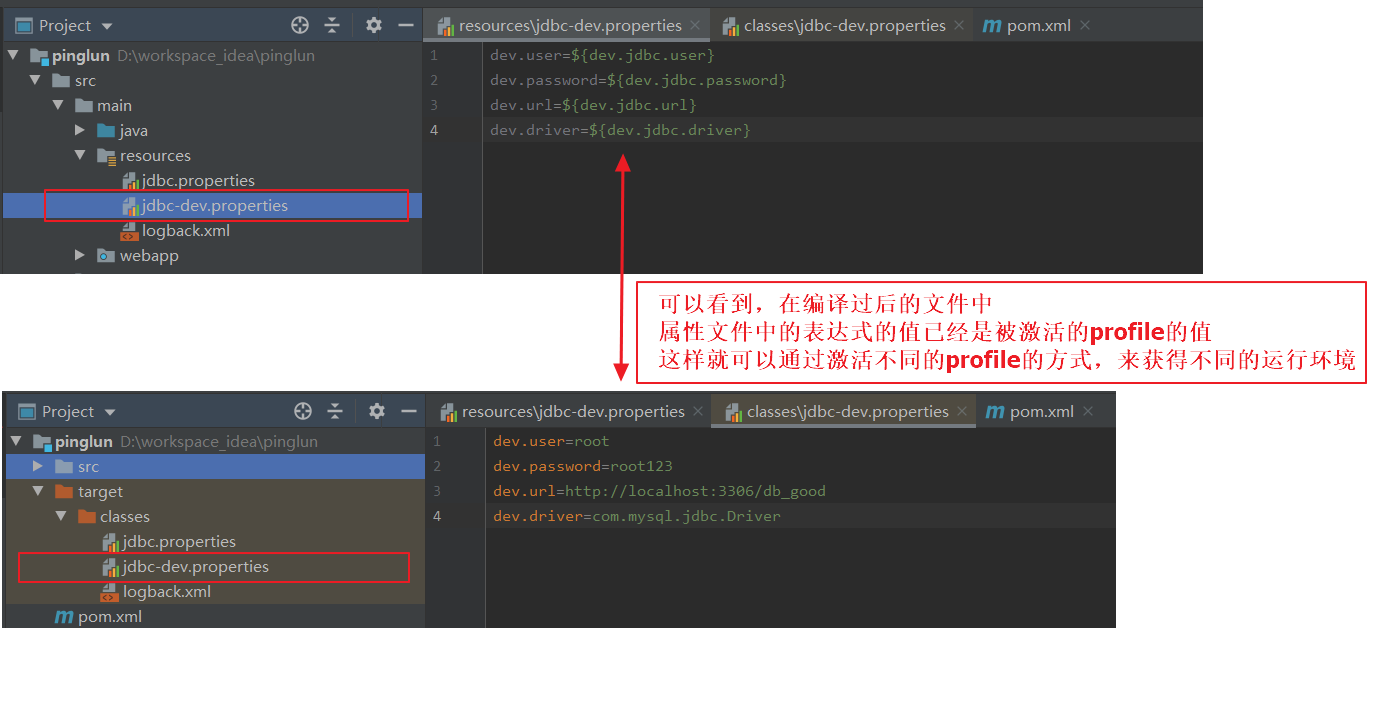
2.创建待处理的资源文件



3.执行处理资源命令

mvn clean resources:resources -PdevJDBCProfile

1. 查看结果



最后

我们时不时会在 resource 标签下看到 includes 和 excludes 标签。它们的作用是：

* includes：指定执行 resource 阶段时要包含到目标位置的资源
* excludes：指定执行 resource 阶段时要排除的资源

来看下面的例子：

<build>

<resources>

<resource>

<!-- 表示为这里指定的目录开启资源过滤功能 -->

<!—当前目录中的资源文件会被过滤-->

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

<!-- 将资源过滤功能打开 -->

<filtering>true</filtering>

<includes>

<!—所有的.properties文件都会放行-->

<include>\*.properties</include>

</includes>

<excludes>

<!—happy.properties将会被排除掉-->

<exclude>happy.properties</exclude>

</excludes>

</resource>

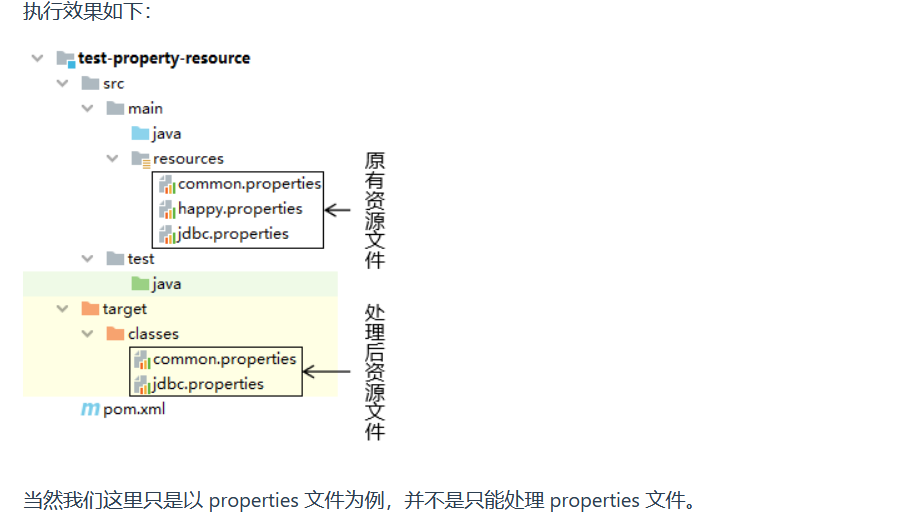
</resources>

</build>

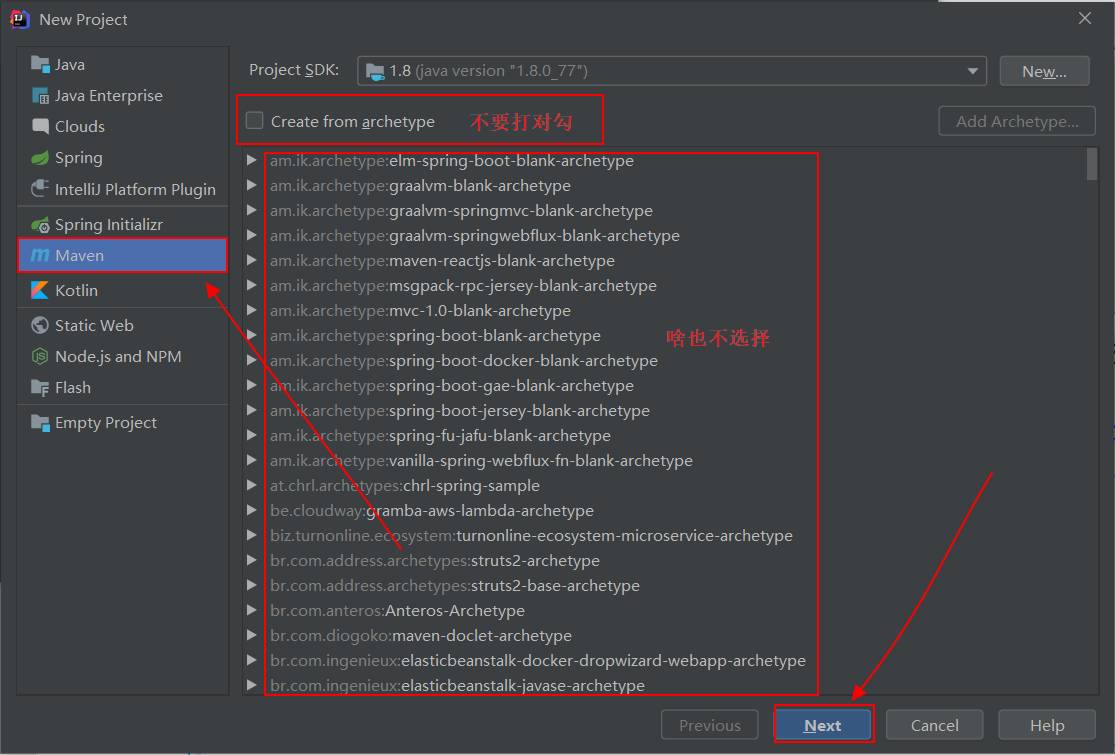
执行处理资源命令：

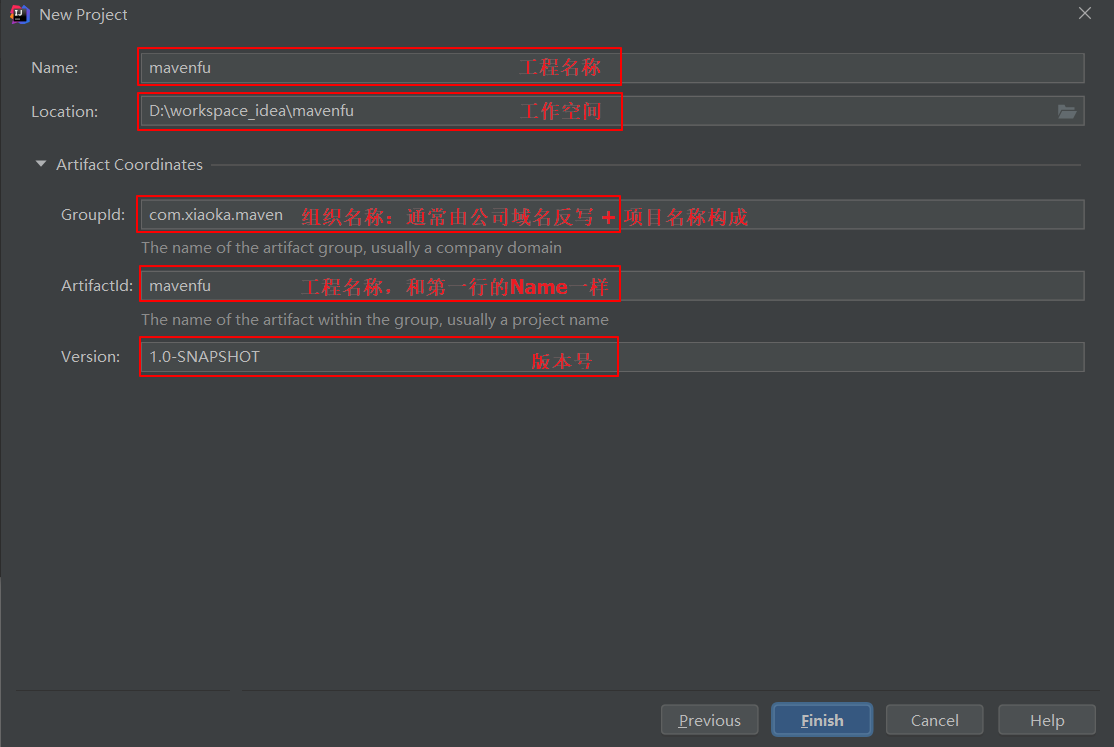
mvn clean resources:resources -PdevJDBCProfile

查看结果



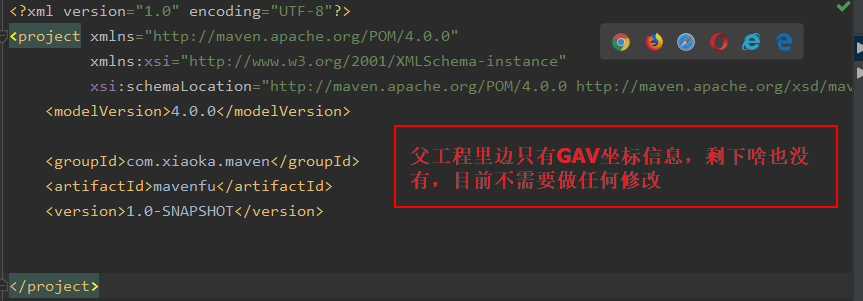
# 使用IDEA创建Maven父工程



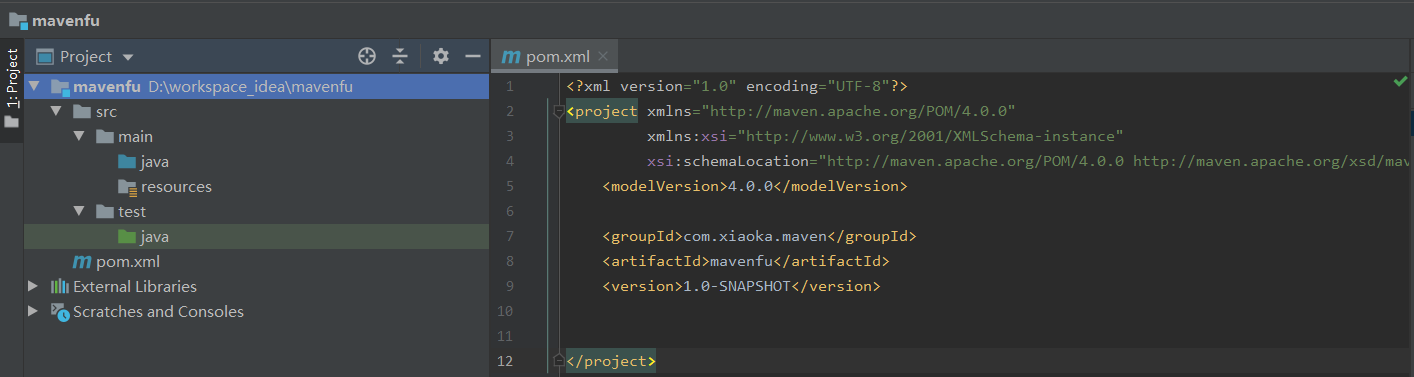


父工程创建完毕

注意，此时我们可以看到，父工程的pom.xml打包方式并没有自动显示出来，此时根本不需要手动添加打包方式，因为创建完子工程以后，父工程的打包方式会自动得到修改。



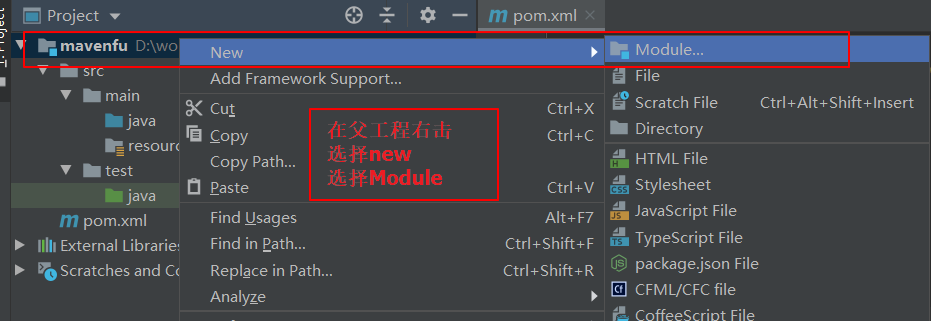
此时父工程的目录结构如图

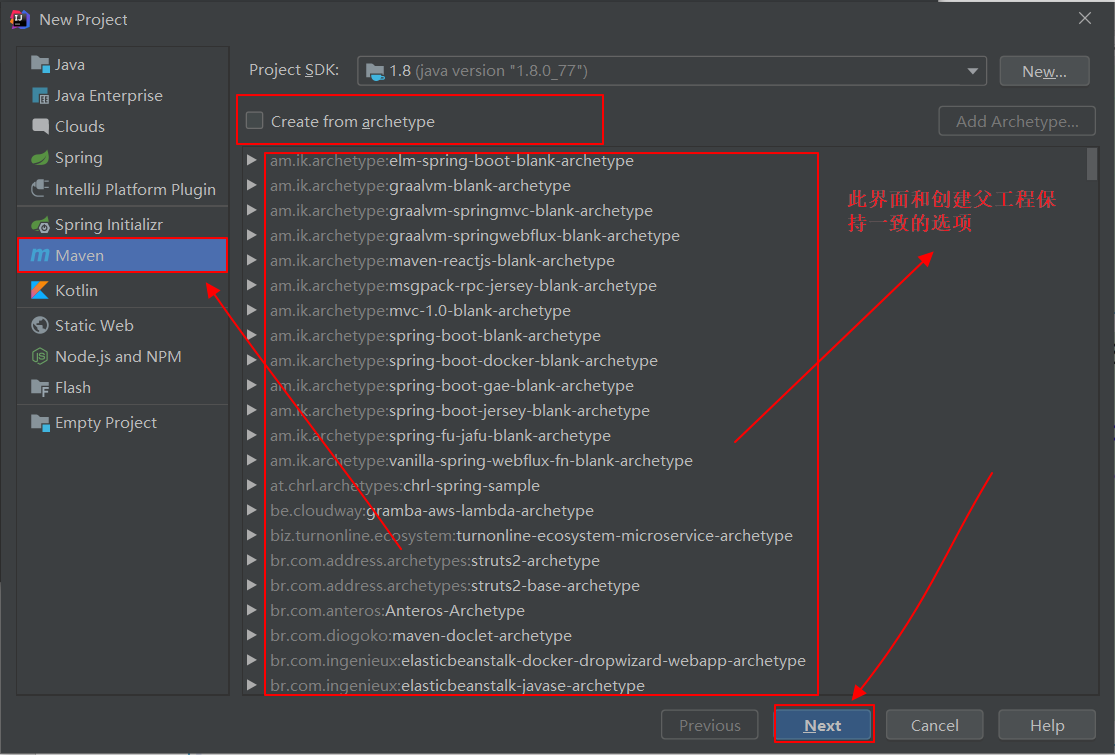


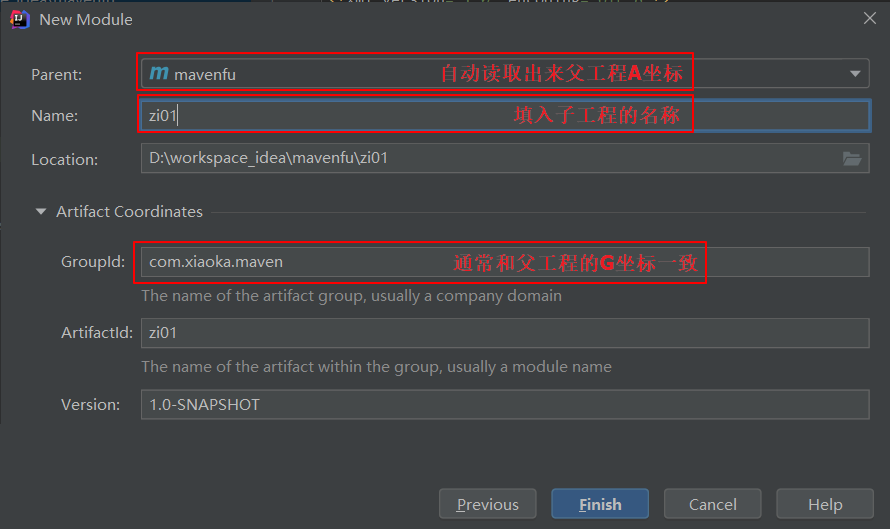
使用IDEA创建Maven子工程

子工程分为Java工程和web工程

## 使用IDEA创建Maven子工程-Java工程

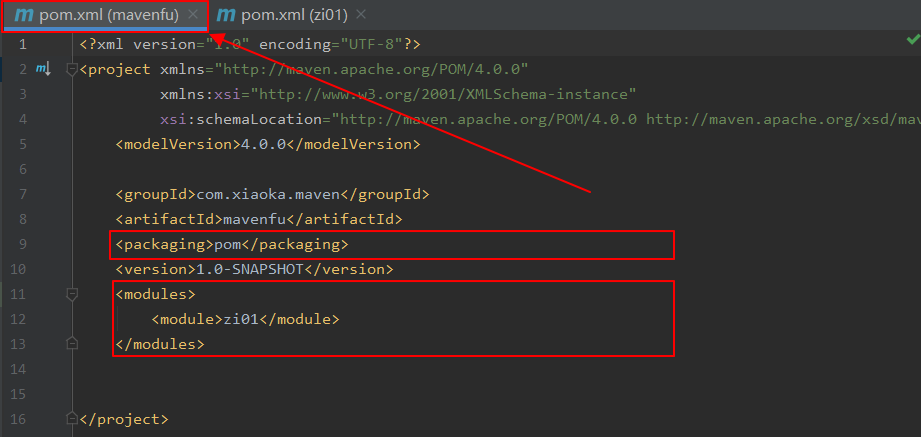


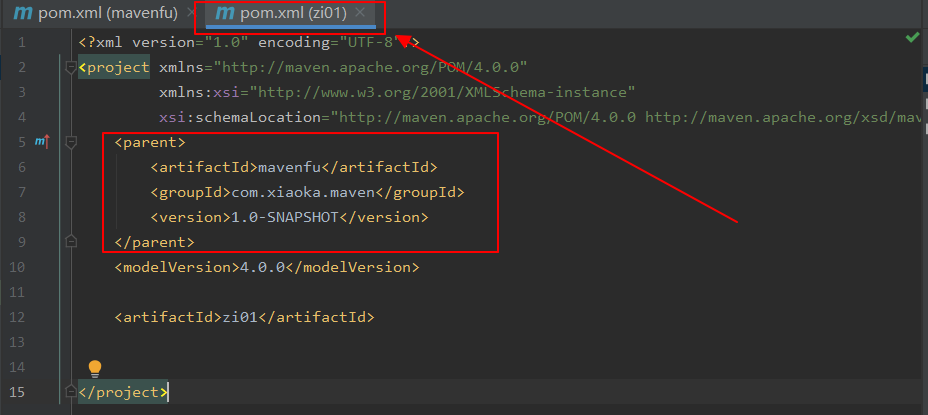




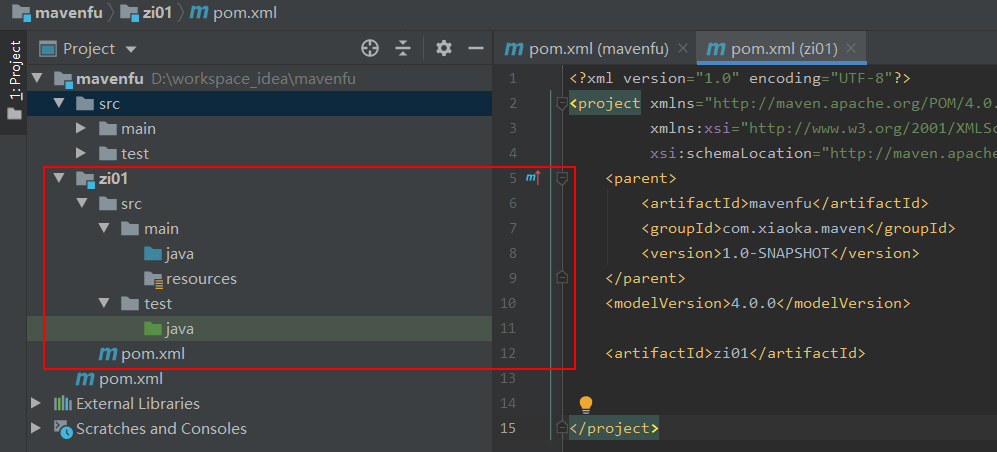
子Java工程创建完毕。

我们来一起看看父工程的pom.xml和子工程的pom.xml文件。





此时，聚合工程的目录结构为



## 使用IDEA创建Maven子工程-web工程

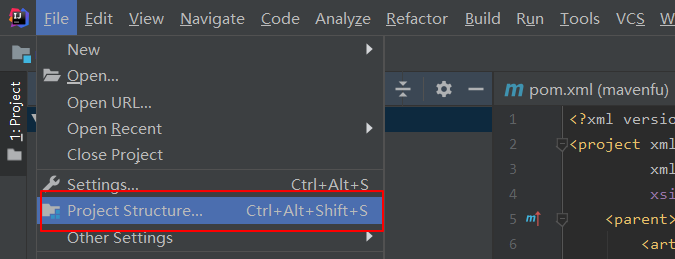
首先按照创建Java子工程的步骤，创建一个子模块工程，取名zi02-web。，**此时**这个模块其实还是一个**Java模块**。

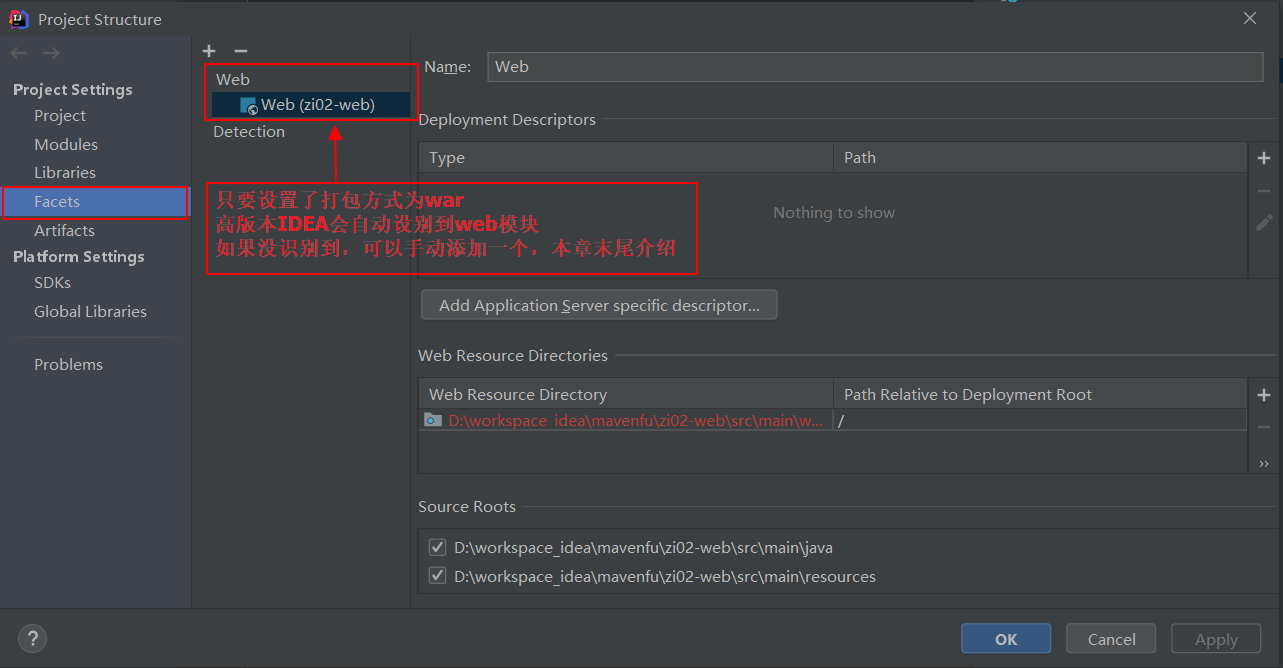
修改zi02-web.pom.xml文件的打包方式为war。

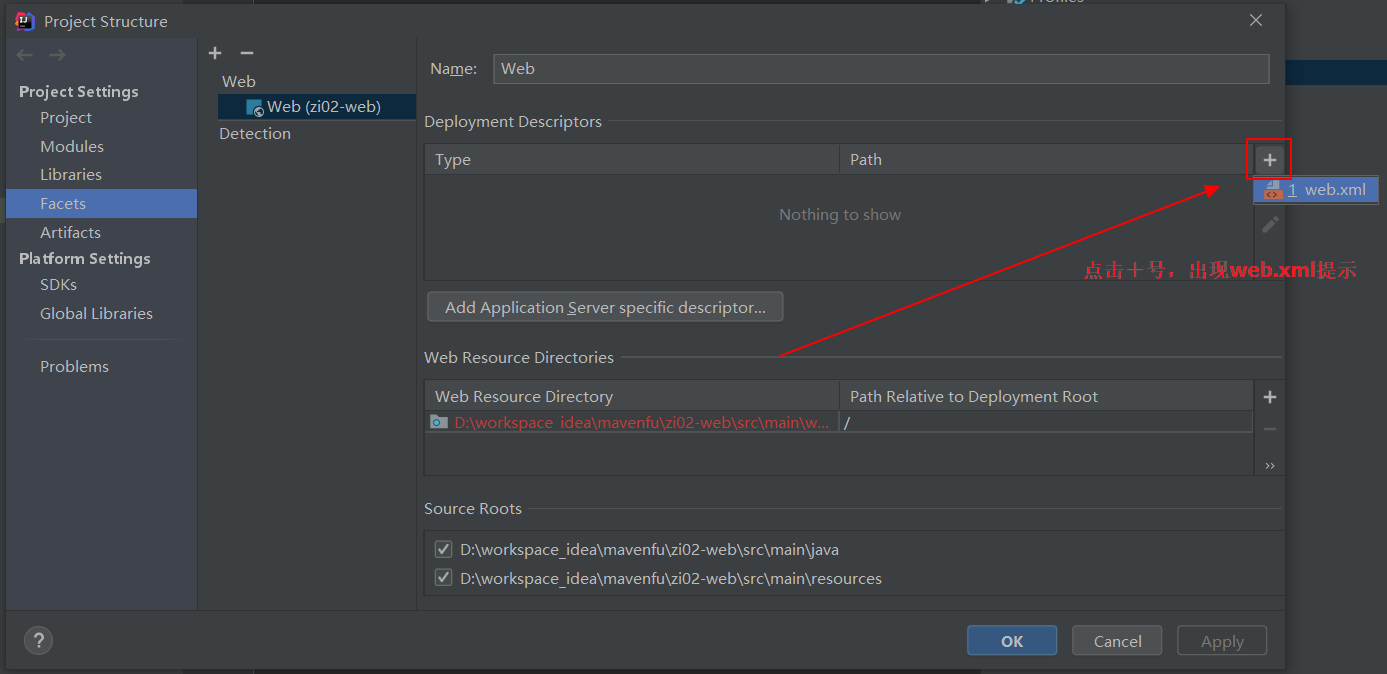
zi02-web.pom.xml文件

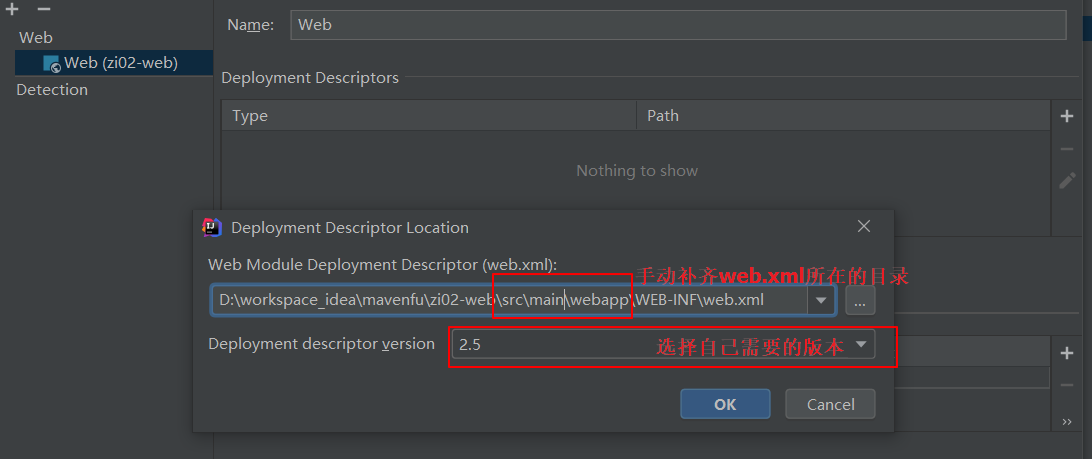
<packaging>war</packaging>

接下来，按图索骥



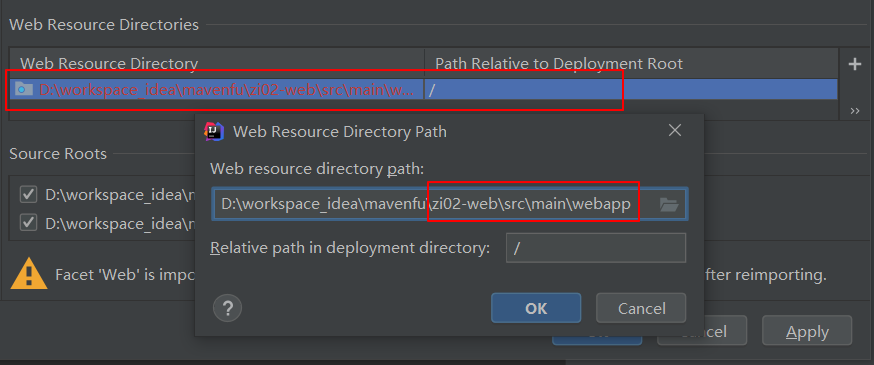


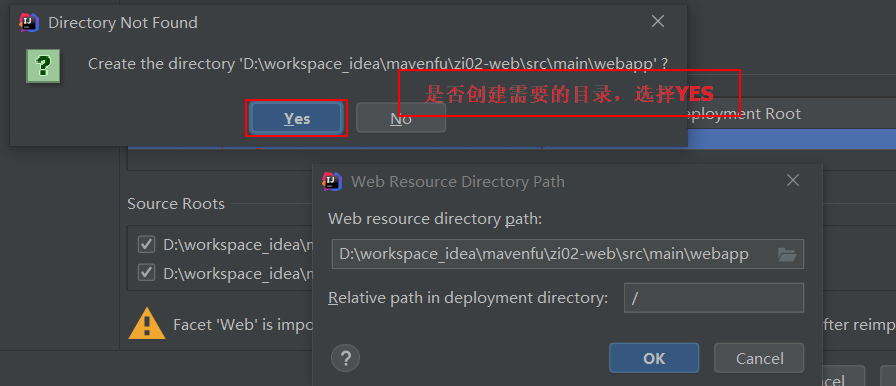




Web.xml的目录为项目/src/main/webapp/WEB-INF/web.xml

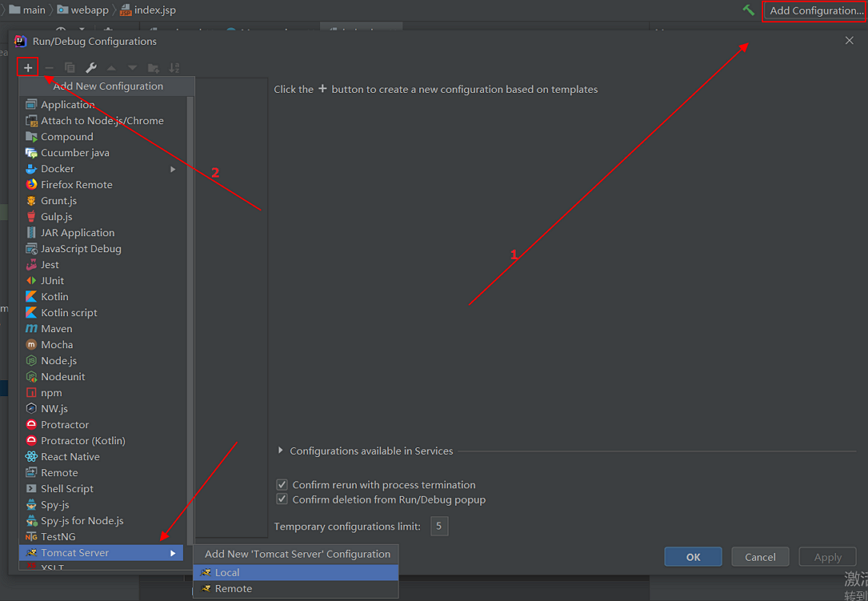
最后，修改web资源目录即可。

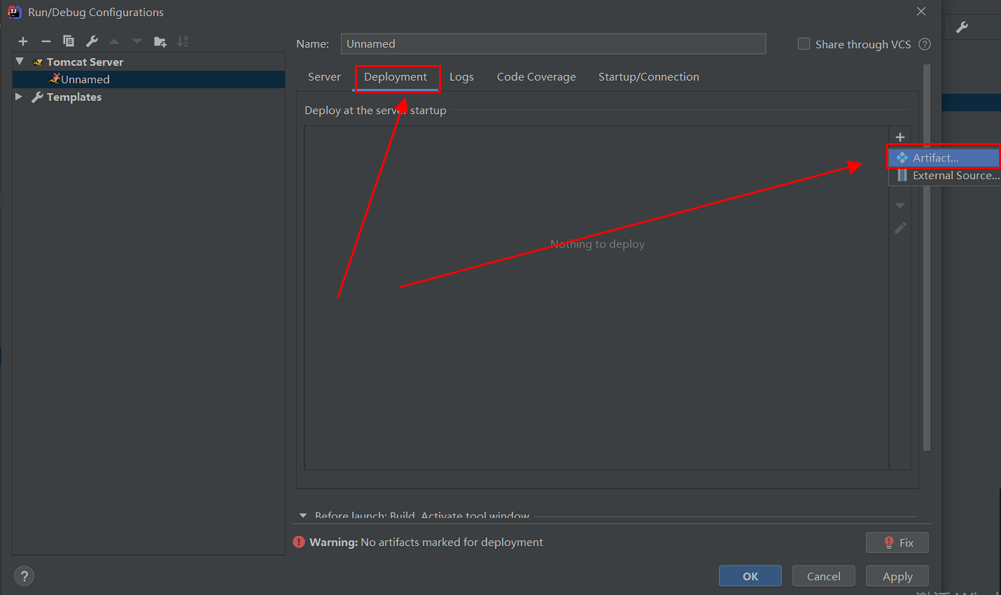


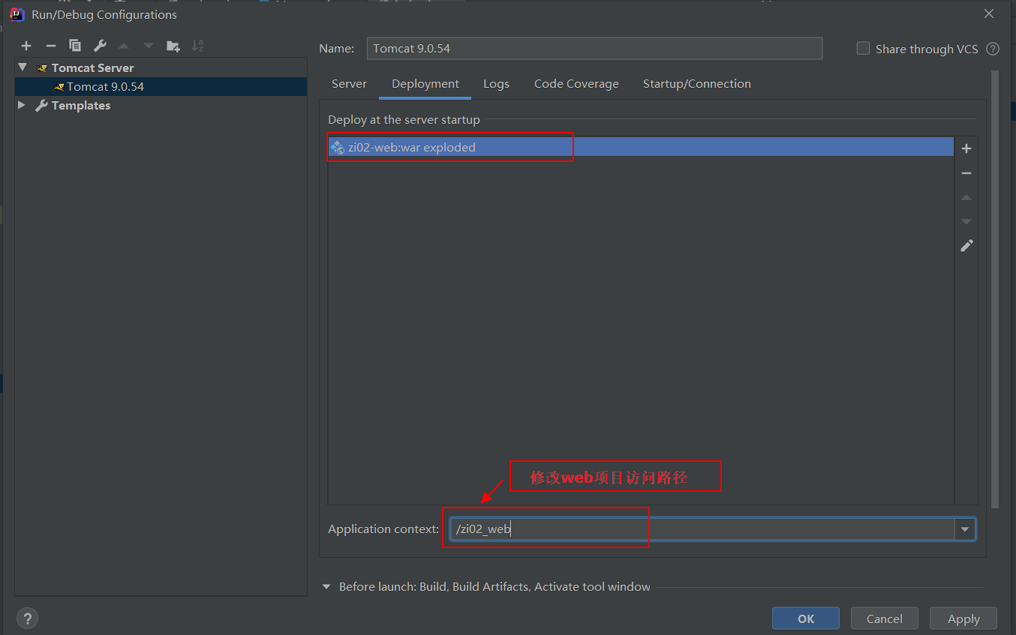


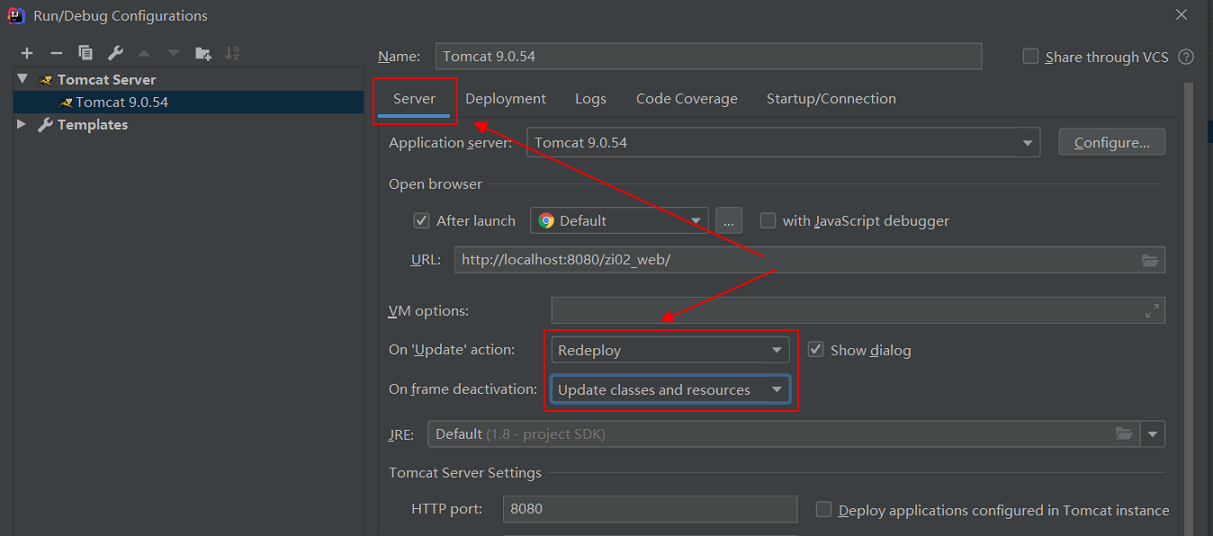
至此，web项目创建完毕。

发布到Tomcat



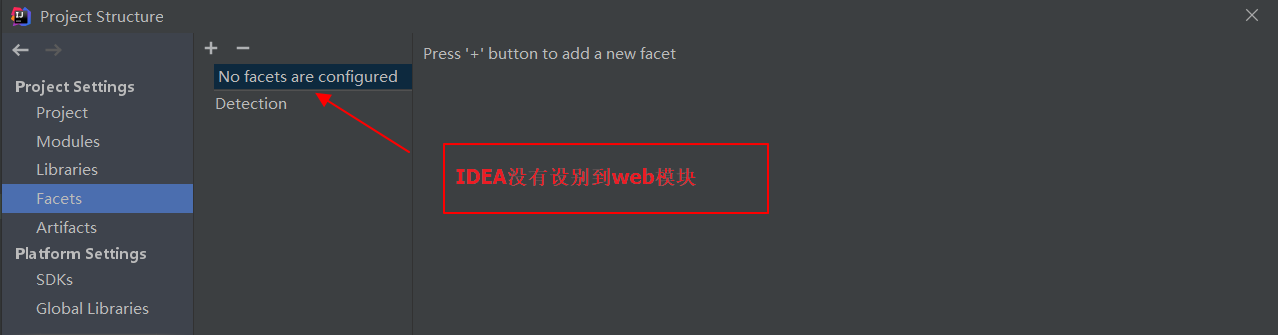


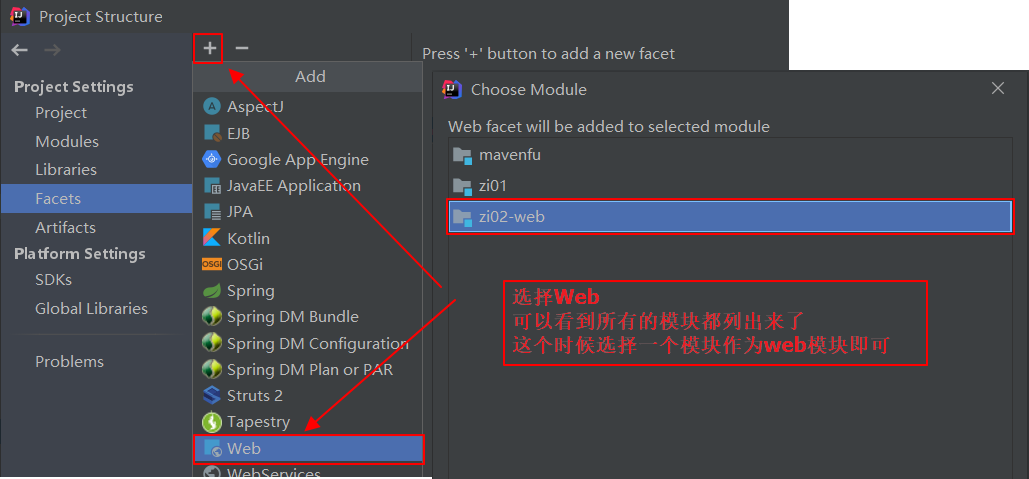




启动tomcat即可。

现在，补充如何手动添加web模块。





## 在IDEA中Maven命令的三种执行方式

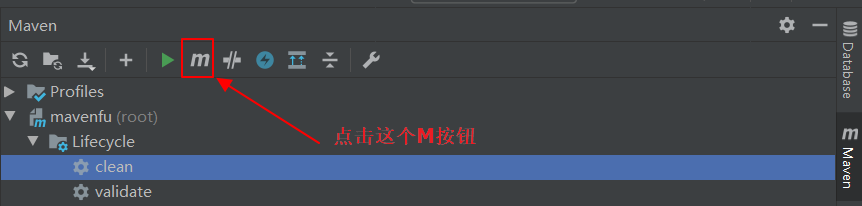
### 图形界面

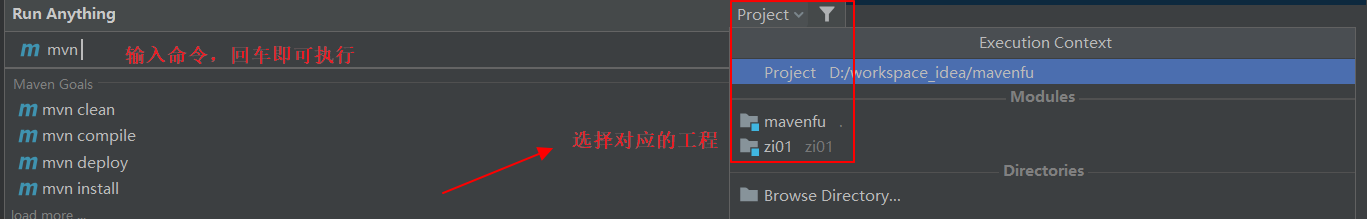
在对应的工程中双击需要执行的命令即可。



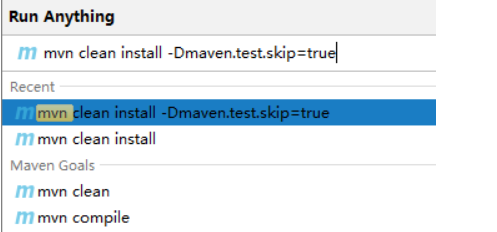
、

### 手动输入





如果有需要，还可以给命令后面附加参数：

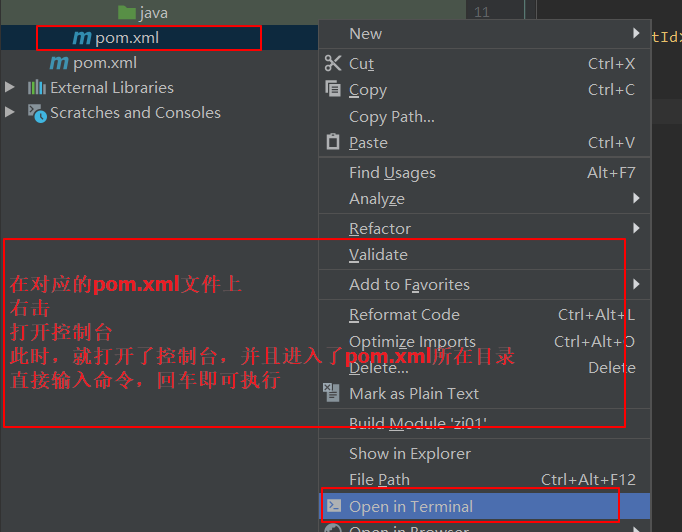


# -D 表示后面要附加命令的参数，字母 D 和后面的参数是紧挨着的，中间没有任何其它字符

# maven.test.skip=true 表示在执行命令的过程中跳过测试

mvn clean install -Dmaven.test.skip=true

### 进入控制台



## 导入一个工程

File ---- open --- 选择工程pom.xml文件所在目录即可。

注意：导入工程以后，记得修改工程的配置信息 ： maven配置，jdk配置，tomcat配置等等。这是导入工程，工程，工程，不是导入子模块！

## 导入一个模块/子工程

首先将整个modual目录复制到工程的根目录，此时IDEA已经可以作为普通目录和普通文件识别到modual的存在。

File --- Project Structer – moduals – 点击➕

Import modual

选择modual目录即可。

如果modual是一个web工程，记得修改web.xml,web资源的目录，并且发布到tomcat。