

# Maven 实战问题和最佳实践

静默虚空 Java后端 2019-11-07

点击上方 Java后端, 选择 设为星标

优质文章, 及时送达

作者 | dunwu

来源 | [github.com/dunwu/java-tutorial](https://github.com/dunwu/java-tutorial)

## 一、常见问题

### 1、dependencies 和 dependencyManagement, plugins 和 pluginManagement 有什么区别?

dependencyManagement 是表示依赖 jar 包的声明, 即你在项目中的 dependencyManagement 下声明了依赖, maven 不会加载该依赖, dependencyManagement 声明可以被继承。

dependencyManagement 的一个使用案例是当有父子项目的时候, 父项目中可以利用 dependencyManagement 声明子项目中需要用到的依赖 jar 包, 之后, 当某个或者某几个子项目需要加载该插件的时候, 就可以在子项目中 dependencies 节点只配置 groupId 和 artifactId 就可以完成插件的引用。

dependencyManagement 主要是为了统一管理插件, 确保所有子项目使用的插件版本保持一致, 类似的还有 plugins 和 pluginManagement。

### 2、IDEA 修改 JDK 版本后编译报错

#### 错误现象

修改 JDK 版本, 指定 maven-compiler-plugin 的 source 和 target 为 1.8。

然后, 在 IntelliJ IDEA 中执行 maven 指令, 报错:

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.0:compile (default-compile) on project ap
```

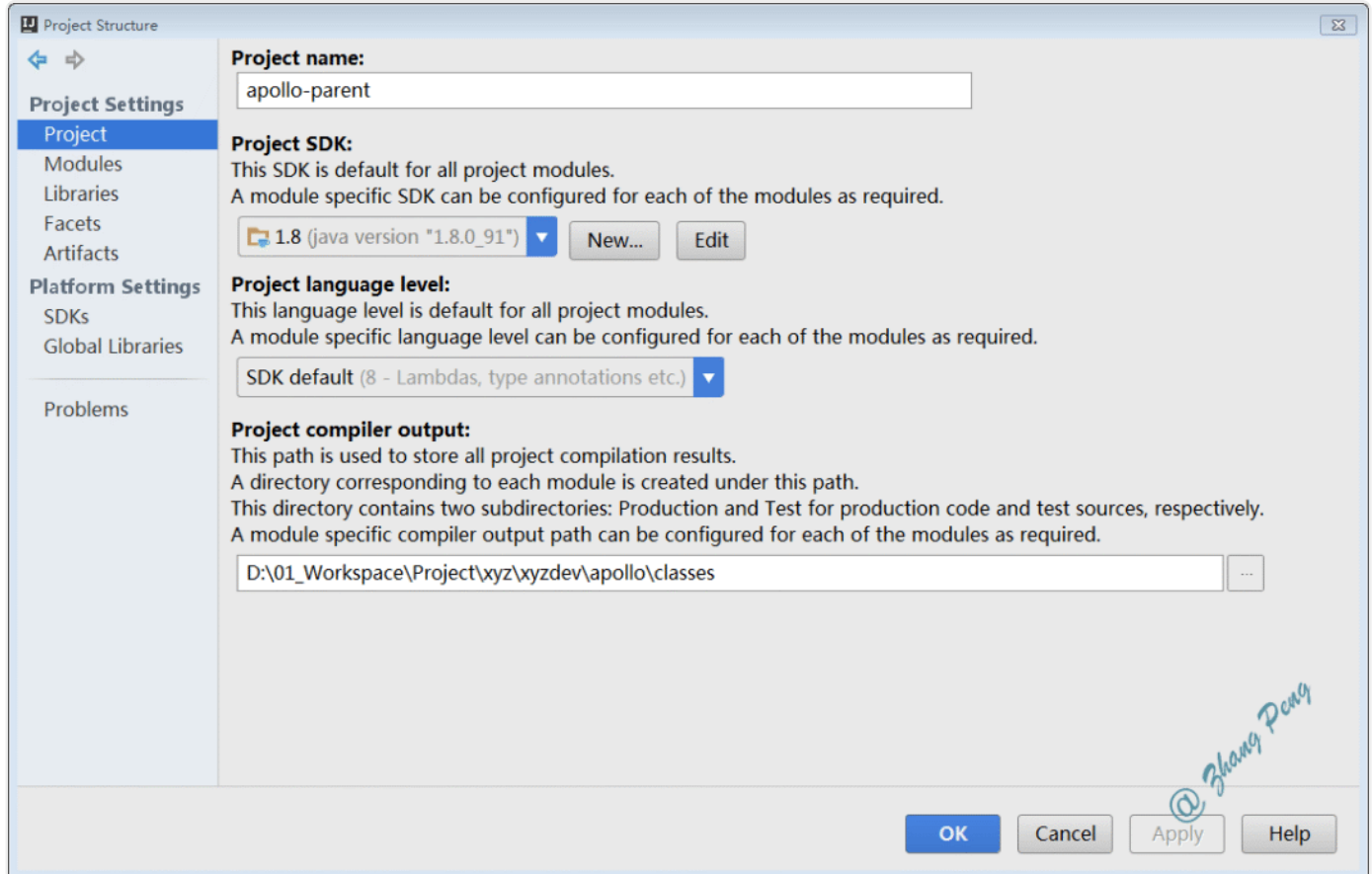
#### 错误原因

maven 的 JDK 源与指定的 JDK 编译版本不符。

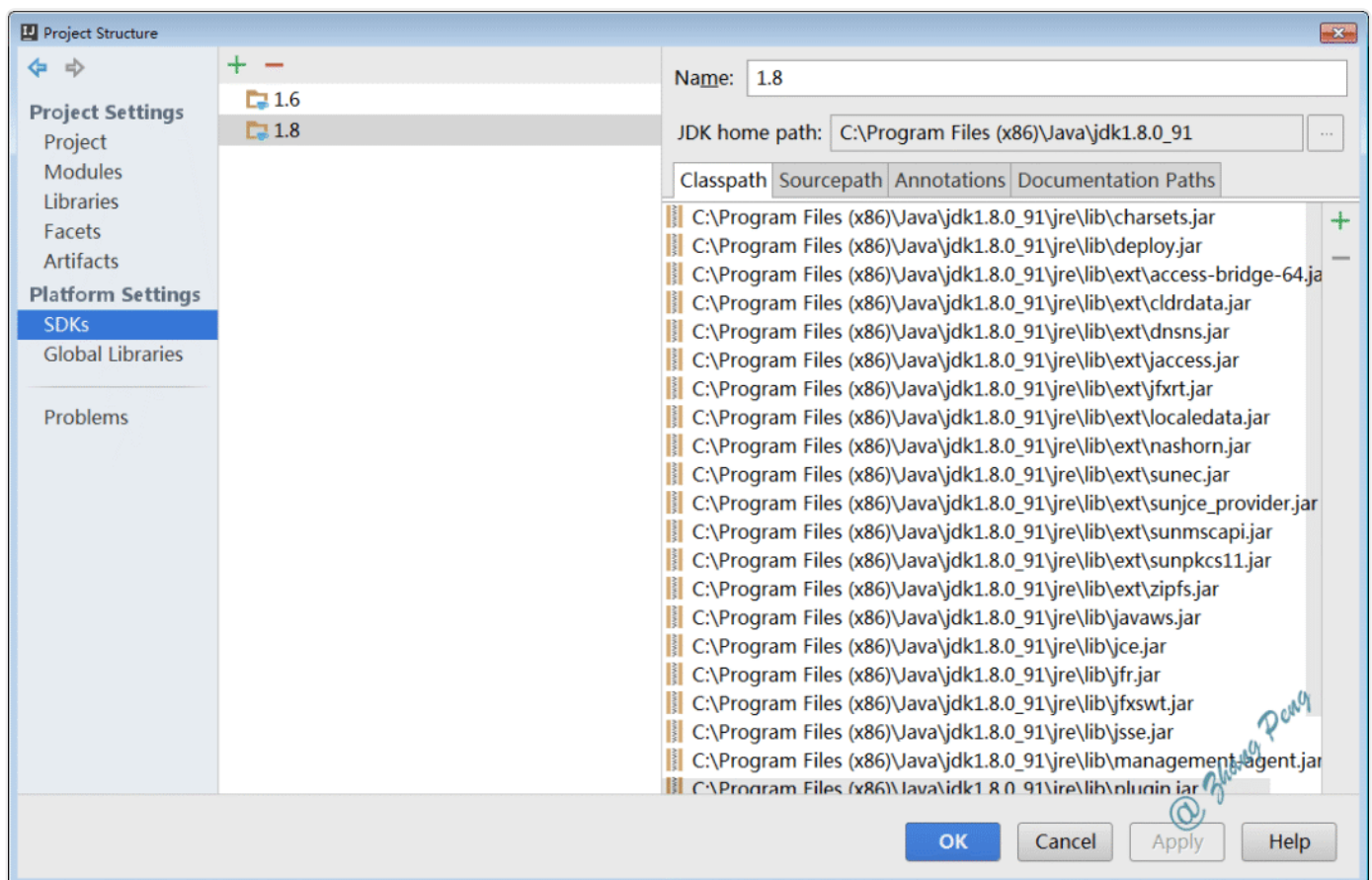
#### 排错手段

##### ■ 查看 Project Settings

Project SDK 是否正确

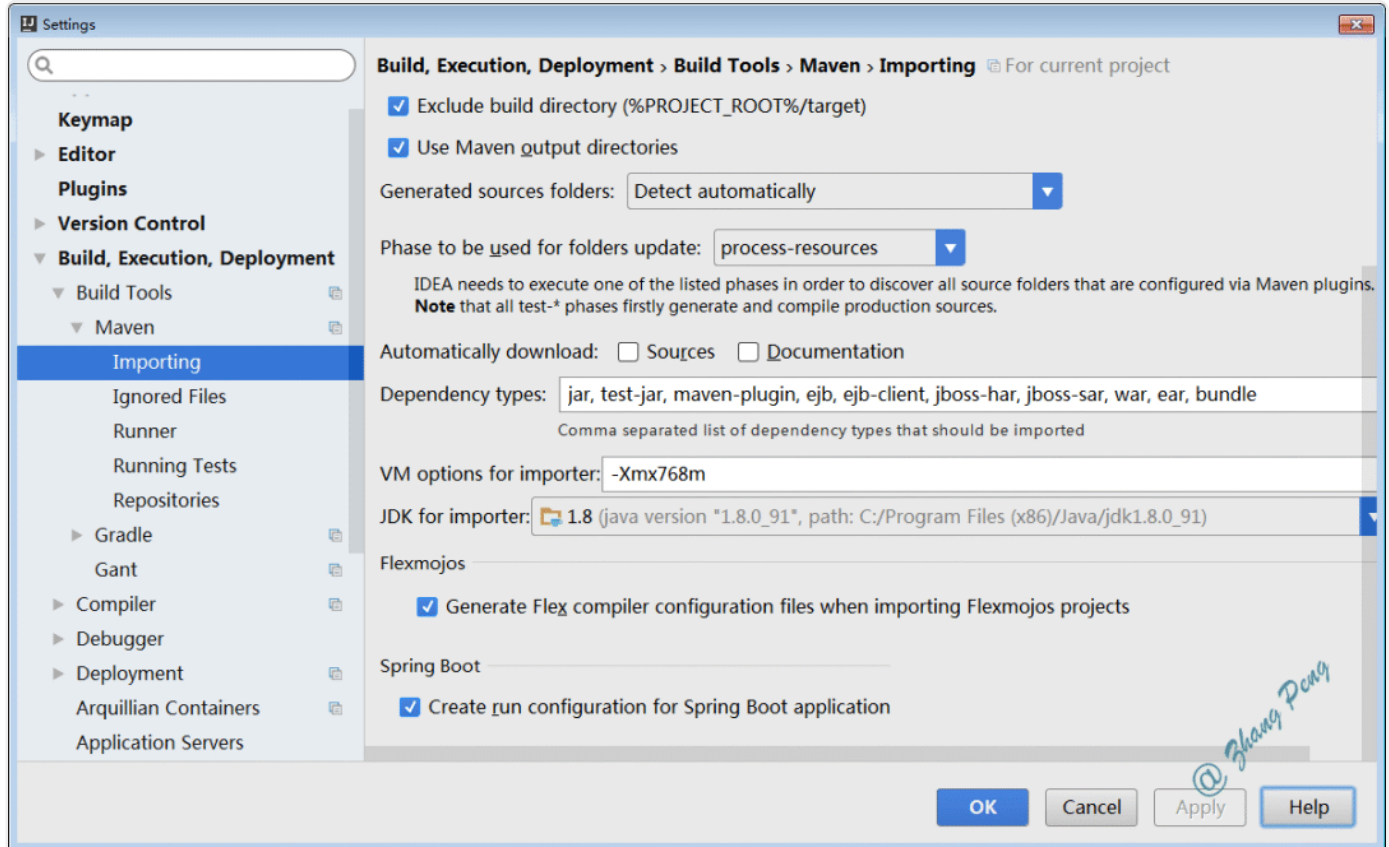


SDK 路径是否正确

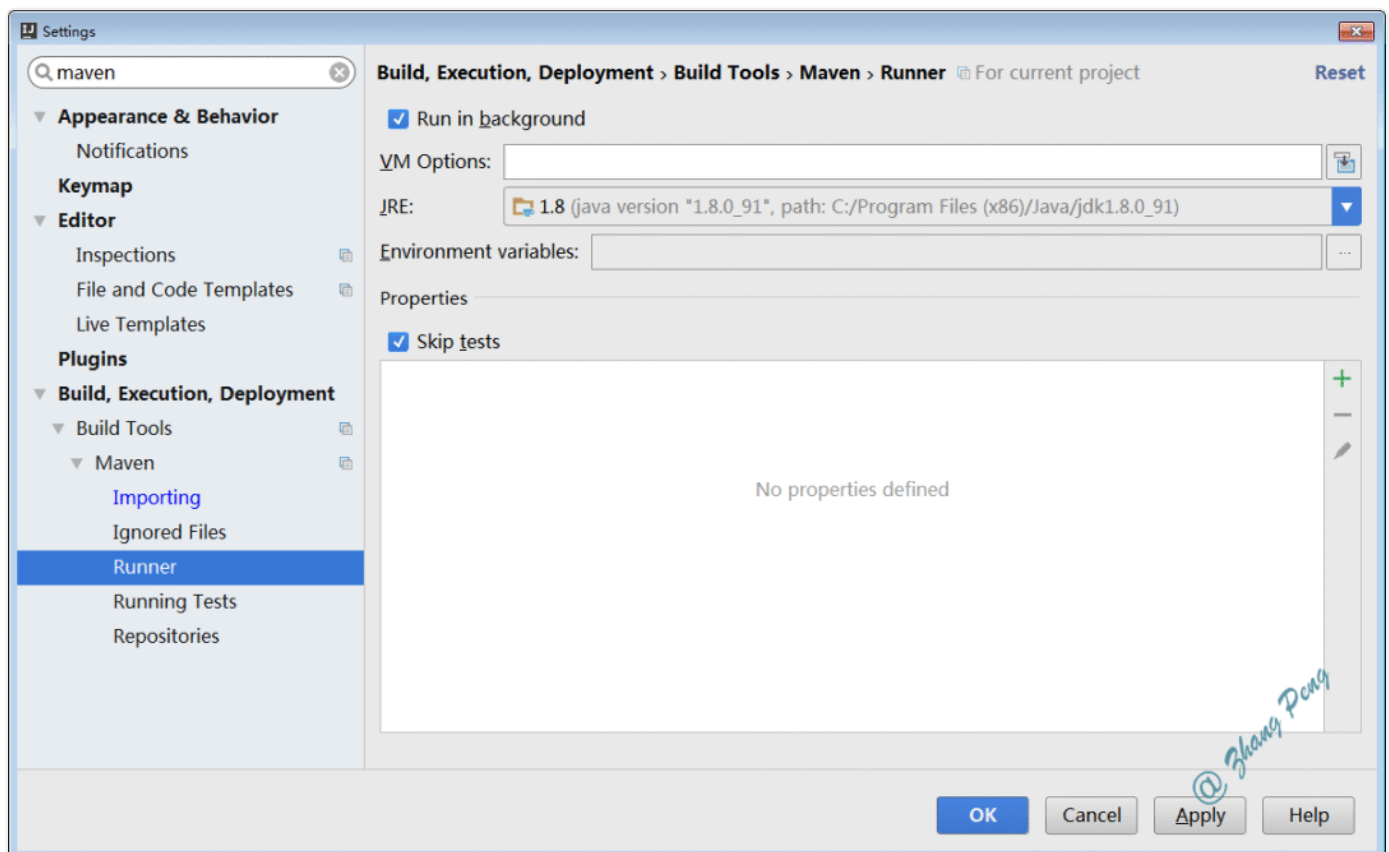


■ 查看 Settings > Maven 的配置

JDK for importer 是否正确



Runner 是否正确



### 3、重复引入依赖

在 Idea 中，选中 Module，使用 Ctrl+Alt+Shift+U，打开依赖图，检索是否存在重复引用的情况。如果存在重复引用，可以将多余的引用删除。

Tips：关注微信公众号：Java后端，每日获取技术博文推送。

### 4、如何打包一个可以直接运行的 Spring Boot jar 包

可以使用 spring-boot-maven-plugin 插件

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
```

如果引入了第三方 jar 包，如何打包？

首先，要添加依赖

```
<dependency>
<groupId>io.github.dunwu</groupId>
<artifactId>dunwu-common</artifactId>
<version>1.0.0</version>
<scope>system</scope>
<systemPath>${project.basedir}/src/main/resources/lib/dunwu-common-1.0.0.jar</systemPath>
</dependency>
```

接着，需要配置 spring-boot-maven-plugin 插件：

```
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>repackage</goal>
</goals>
</execution>
</executions>
<configuration>
<includeSystemScope>true</includeSystemScope>
</configuration>
</plugin>
</plugins>
</build>
```

## 5、去哪儿找 maven dependency ？

问：刚接触 maven 的新手，往往会有这样的疑问，我该去哪儿找 jar？

答：官方推荐的搜索 maven dependency 网址：

- <https://search.maven.org>
- <https://repository.apache.org>
- <https://mvnrepository.com>

## 6、如何指定编码？

问：众所周知，不同编码格式常常会产生意想不到的诡异问题，那么 maven 构建时如何指定 maven 构建时的编码？

答：在 properties 中指定 `project.build.sourceEncoding`

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

## 7、如何指定 JDK 版本？

问：如何指定 maven 构建时的 JDK 版本？

答：有两种方法：

(1) properties 方式

```
<project>
...
<properties>
<maven.compiler.source>1.7</maven.compiler.source>
<maven.compiler.target>1.7</maven.compiler.target>
</properties>
...
</project>
```

(2) 使用 maven-compiler-plugin 插件，并指定 source 和 target 版本

```
<build>
...
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.3</version>
<configuration>
<source>1.7</source>
<target>1.7</target>
</configuration>
</plugin>
</plugins>
...
</build>
```

## 8、如何避免将 dependency 打包到构件中？

答：指定 maven dependency 的 scope 为 `provided`，这意味着：依赖关系将在运行时由其容器或 JDK 提供。具有此范围的依赖关系不会传递，也不会捆绑在诸如 WAR 之类的包中，也不会包含在运行时类路径中。

## 9、如何跳过单元测试

问：执行 mvn package 或 mvn install 时，会自动编译所有单元测试(src/test/java 目录下的代码)，如何跳过这一步？

答：在执行命令的后面，添加命令行参数 `-Dmaven.test.skip=true` 或者 `-DskipTests=true`

## 10、IDEA 修改 JDK 版本后编译报错

错误现象

修改 JDK 版本，指定 maven-compiler-plugin 的 source 和 target 为 1.8。

然后，在 IntelliJ IDEA 中执行 maven 指令，报错：

```
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.0:compile (default-compile) on project apollo-parent: Compilation failure: Compilation failure: [1] D:\01_Workspace\Project\xyz\xyzdev\apollo\classes: 无法生成类文件，因为目标平台与源版本不兼容。请指定源和目标版本。 -> [Help 1]
```

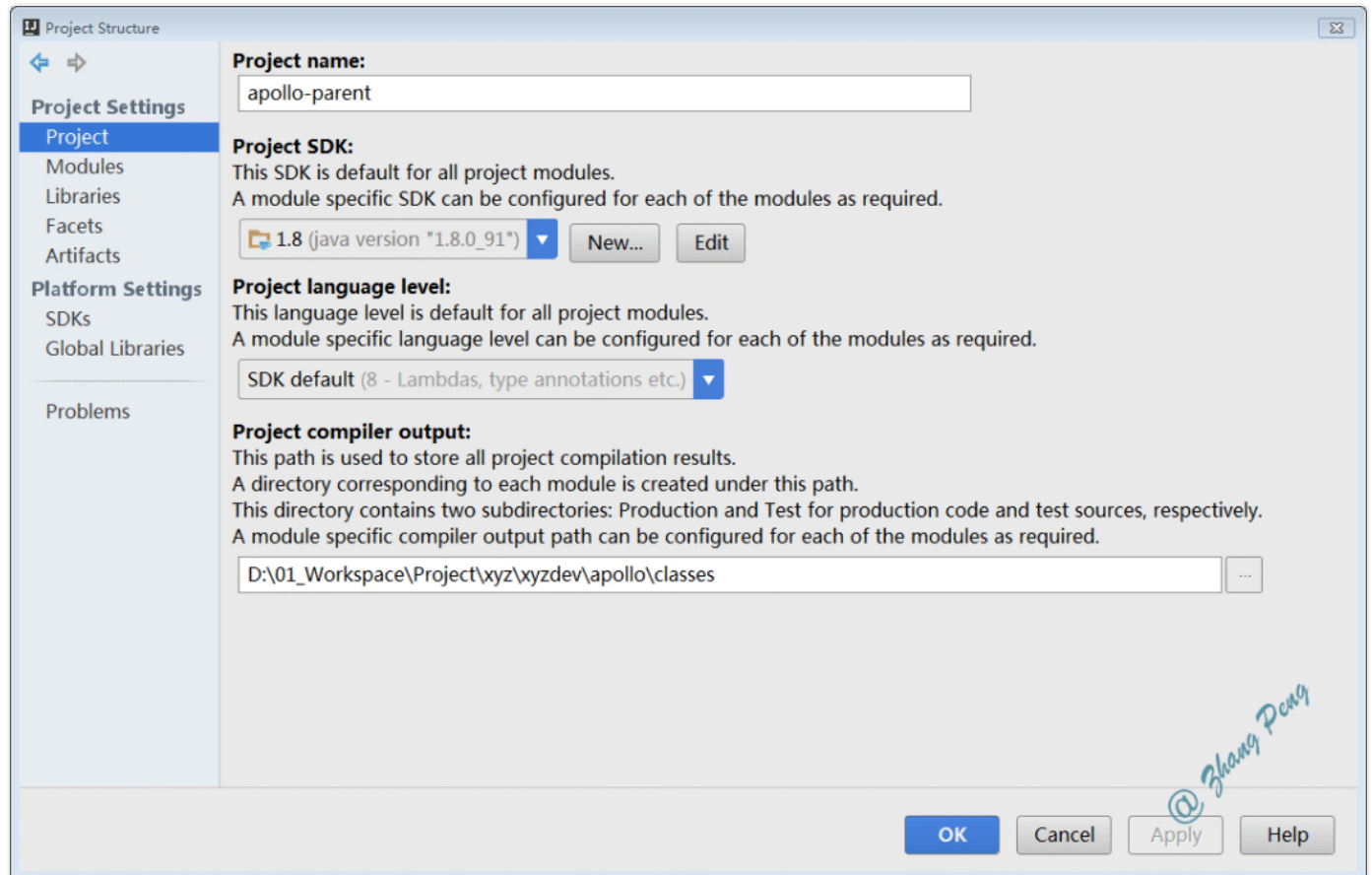
## 错误原因

maven 的 JDK 源与指定的 JDK 编译版本不符。

## 排错手段

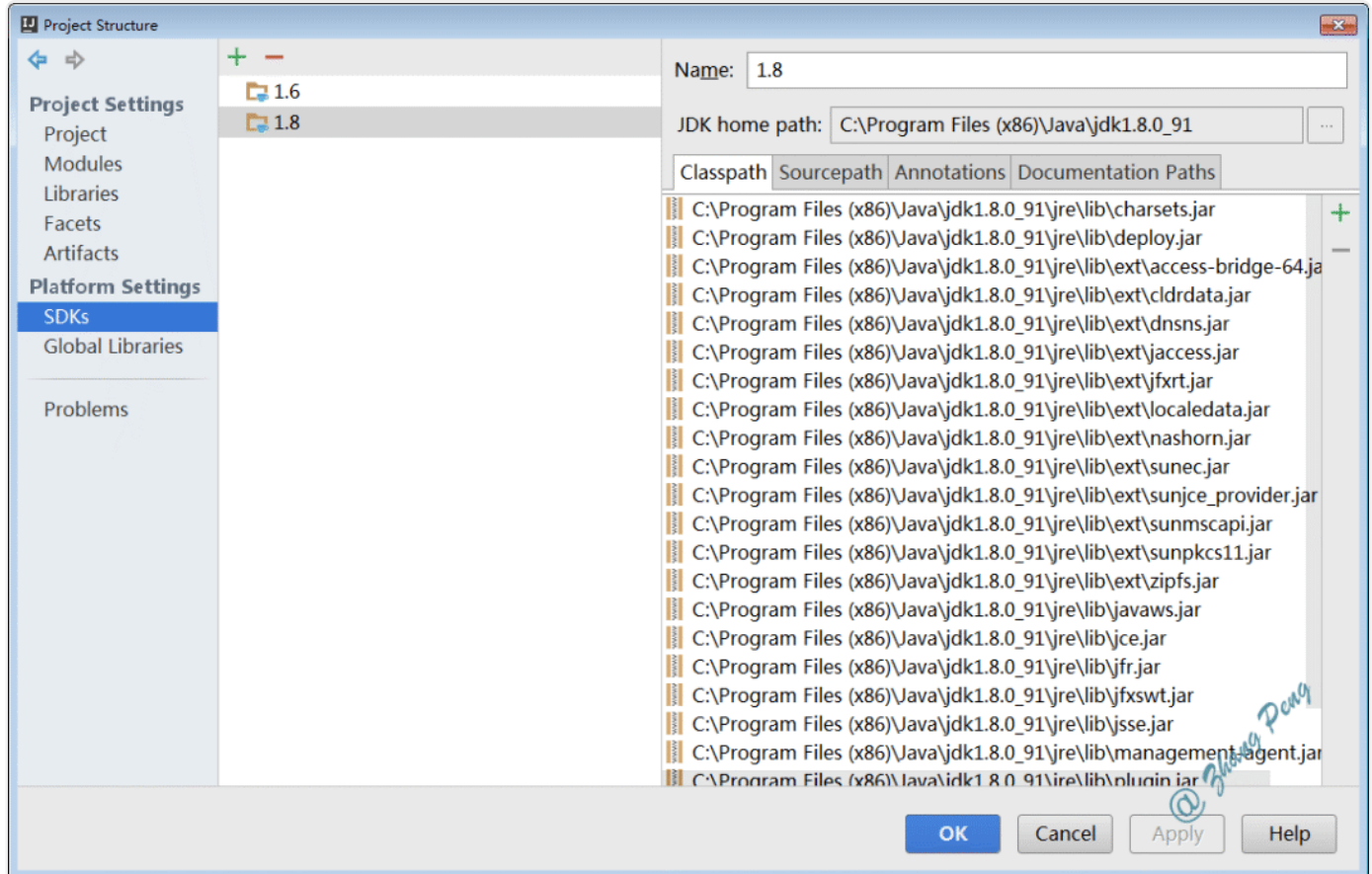
- 查看 Project Settings

Project SDK 是否正确



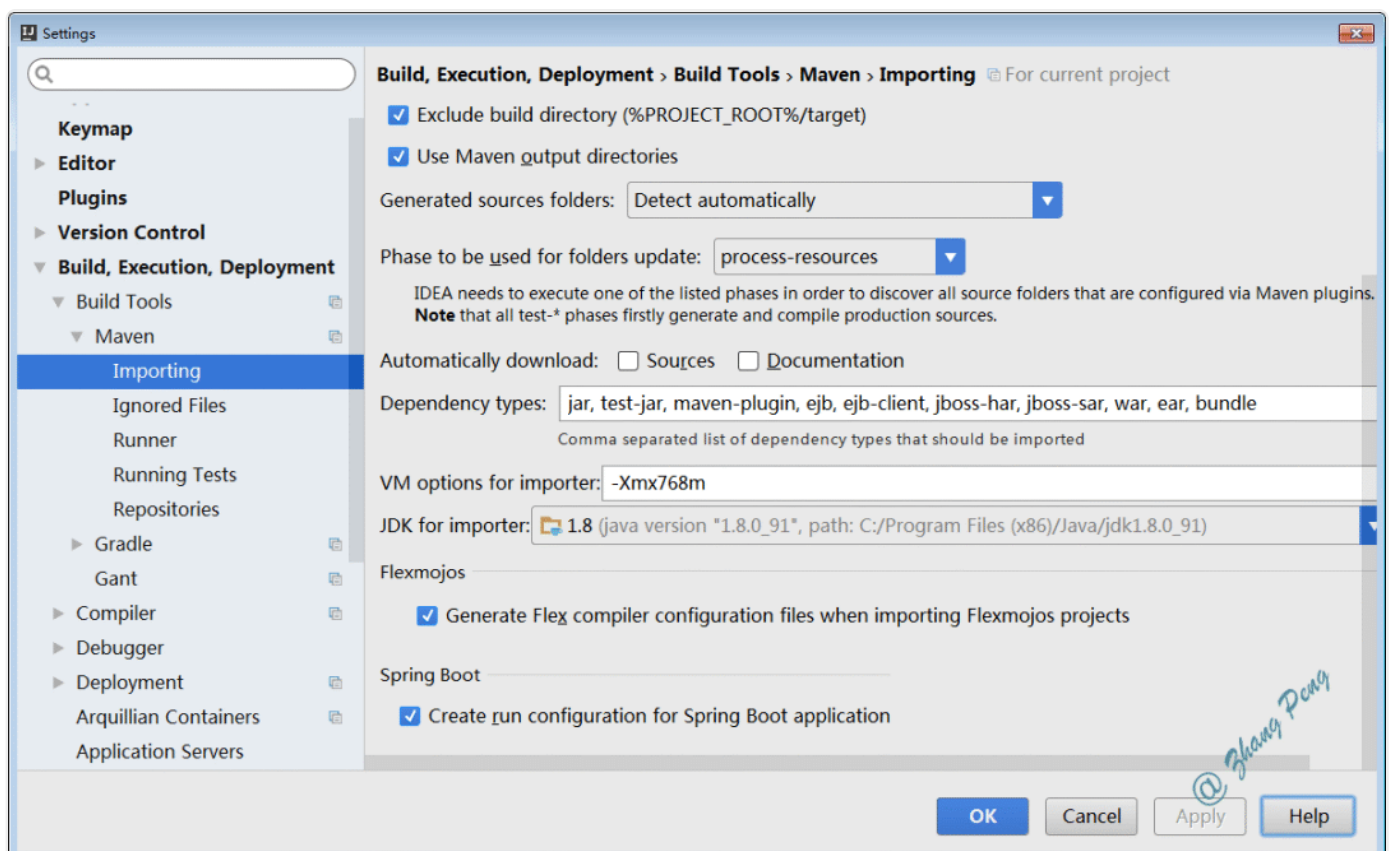
SDK 路径是否正确



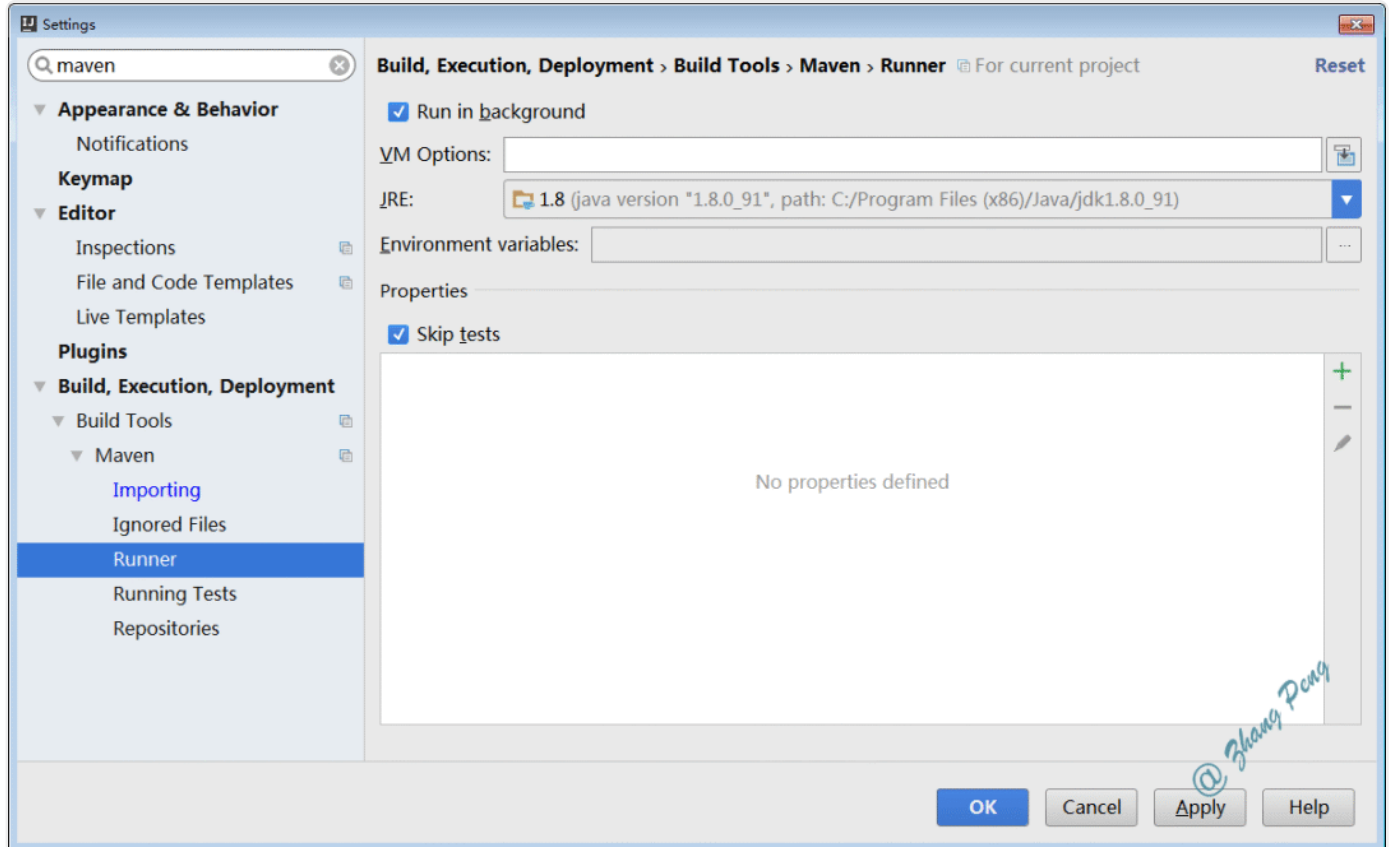


#### ■ 查看 Settings > Maven 的配置

JDK for importer 是否正确



Runner 是否正确



## 11、重复引入依赖

在 Idea 中，选中 Module，使用 Ctrl+Alt+Shift+U，打开依赖图，检索是否存在重复引用的情况。

## 12、如何引入本地 jar

问：有时候，需要引入在中央仓库找不到的 jar，但又想通过 maven 进行管理，那么应该如何做到呢？

答：可以通过设置 dependency 的 scope 为 system 来引入本地 jar。

例：

- 将私有 jar 放置在 resources/lib 下，然后以如下方式添加依赖：
- groupId 和 artifactId 可以按照 jar 包中的 package 设置，只要和其他 jar 不冲突即可。

```
<dependency>
  <groupId>xxx</groupId>
  <artifactId>xxx</artifactId>
  <version>1.0.0</version>
  <scope>system</scope>
  <systemPath>${project.basedir}/src/main/resources/lib/xxx-6.0.0.jar</systemPath>
</dependency>
```

## 13、如何排除依赖

问：如何排除依赖一个依赖关系？比方项目中使用的 libA 依赖某个库的 1.0 版。libB 以来某个库的 2.0 版，如今想统一使用 2.0 版，怎样去掉 1.0 版的依赖？

答：通过 exclusion 排除指定依赖即可。

例：



```
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.4.12</version>
  <optional>true</optional>
  <exclusions>
    <exclusion>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

## 二、最佳实践

### 1、通过 bom 统一管理版本

采用类似 `spring-boot-dependencies` 的方式统一管理依赖版本。

spring-boot-dependencies 的 pom.xml 形式：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.1.9.RELEASE</version>
  <packaging>pom</packaging>

  <!-- 省略 -->

  <!-- 依赖包版本管理 -->
  <dependencyManagement>
    <dependencies>
      <!-- 省略 -->
    </dependencies>
  </dependencyManagement>

  <build>
    <!-- 插件版本管理 -->
    <pluginManagement>
      <plugins>
        <!-- 省略 -->
      </plugins>
    </pluginManagement>
  </build>
</project>
```

其他项目引入 spring-boot-dependencies 来管理依赖版本的方式：

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>${spring-boot.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

- END -

如果看到这里, 说明你喜欢这篇文章, 请[转发、点赞](#)。微信搜索「web\_resource」, 关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



#### 推荐阅读

1. Spring Boot 全局异常处理整理
2. 细说 Java 主流日志工具库
3. 9 个爱不释手的 JSON 工具
4. 12306 的架构到底有多牛逼?
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号: Java后端

喜欢文章, 点个在看 

[阅读原文](#)

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 学 Maven，看这一篇就够了

Java后端 2019-12-19

以下文章来源于江南一点雨，作者江南一点雨



江南一点雨

一站式Java全栈技术学习平台！



作者 | 松哥

公众号 | 江南一点雨

## 1 Maven 介绍

### 1.1 为什么使用 Maven

由于 Java 的生态非常丰富，无论你想实现什么功能，都能找到对应的工具类，这些工具类都是以 jar 包的形式出现的，例如 Spring、SpringMVC、MyBatis、数据库驱动，等等，都是以 jar 包的形式出现的，jar 包之间会有关联，在使用一个依赖之前，还需要确定这个依赖所依赖的其他依赖，所以，当项目比较大的时候，依赖管理会变得非常麻烦臃肿，这是 Maven 解决的第一个问题。

Maven 还可以处理多模块项目。简单的项目，单模块分包处理即可，如果项目比较复杂，要做成多模块项目，例如一个电商项目有订单模块、会员模块、商品模块、支付模块...，一般来说，多模块项目，每一个模块无法独立运行，要多个模块合在一起，项目才可以运行，这个时候，借助 Maven 工具，可以实现项目的一键打包。



Maven 之前，我们更多的是使用 Ant 的项目构建工具，Ant 有一个特点，每次都得写，每次都写的差不多，配置也臃肿。所以，后来搞出来 Maven。Maven 就是最先进的版本构建工具吗？不是的，只不过，目前在 Java 领域 Maven 使用比较多。除了 Maven，还有 Gradle。

”

### 1.2 Maven 是什么

Maven 是一个项目管理工具，它包含了一个项目对象模型 (Project Object Model)，反映在配置中，就是一个 pom.xml 文件。是一组标准集合，一个项目的生命周期、一个依赖管理系统，另外还包括定义在项目生命周期阶段的插件(plugin)以及目标(goal)。

当我们使用 Maven 的使用，通过一个自定义的项目对象模型，pom.xml 来详细描述我们自己的项目。

Maven 中的有两大核心：

- 依赖管理：对 jar 的统一管理(Maven 提供了一个 Maven 的中央仓库，<https://mvnrepository.com/>，当我们在项目中添加完依赖之后，Maven 会自动去中央仓库下载相关的依赖，并且解决依赖的依赖问题)
- 项目构建：对项目进行编译、测试、打包、部署、上传到私服等

## 2. Maven 安装

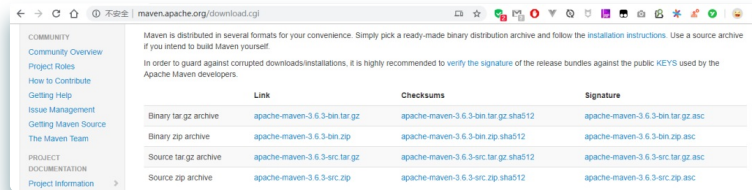
- Maven 是 Java 项目，因此必须先安装 JDK。

```
C:\Users\javaboy>java -version
java version "13.0.1" 2019-10-15
Java(TM) SE Runtime Environment (build 13.0.1+9)
Java HotSpot(TM) 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
C:\Users\javaboy>
```

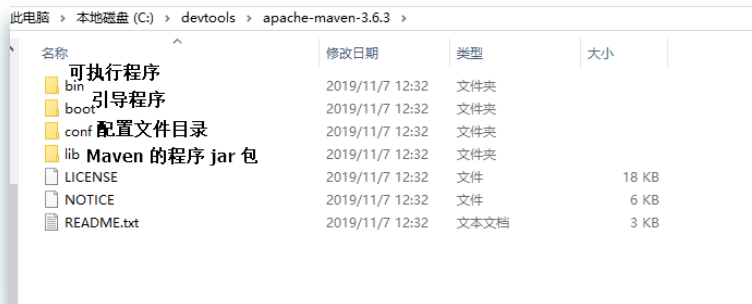
下载 Maven：

- 下载 Maven

下载地址：<http://maven.apache.org/download.cgi>

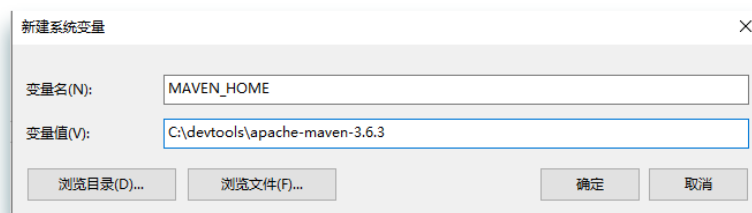


- 解压并配置

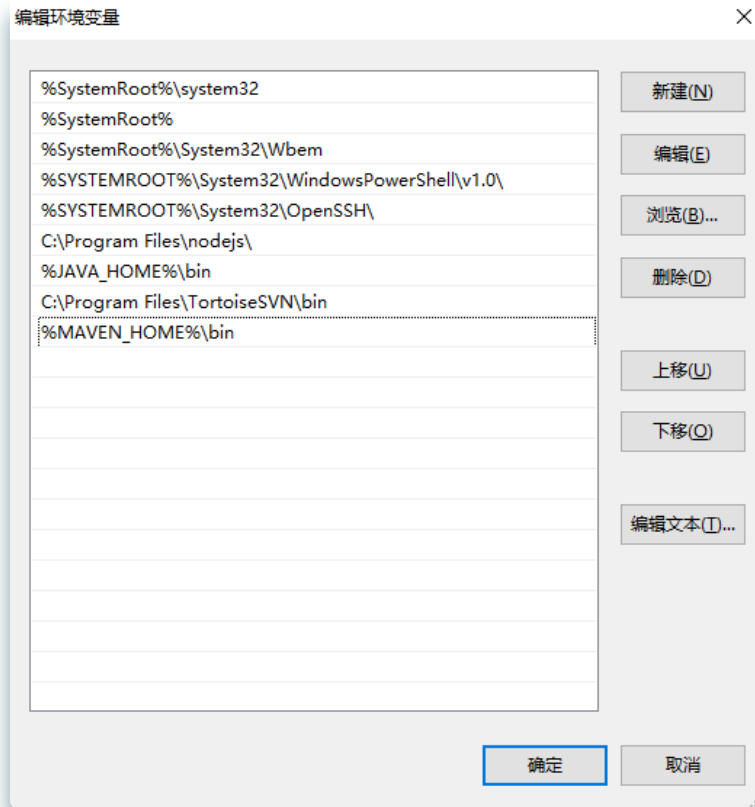


配置，只需要配置环境变量即可：

首先配置 MAVEN\_HOME：



然后配置环境变量：



- 检验安装

```
C:\Users\javaboy>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: C:\devtools\apache-maven-3.6.3\bin\..
Java version: 13.0.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-13.0.1
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
C:\Users\javaboy>
```



如果使用了 IntelliJ IDEA，可以不用去额外下载 Maven，直接使用 IDEA 中自带的 Maven 插件即可。IntelliJ IDEA 中自带的 Maven 插件在 `|ideaIU-2019.2.4.win\plugins\maven\lib\maven3`

Maven 的安装整体上来说比较简单，只需要下载安装包，然后解压并配置环境变量即可。不过，我一般其实建议大家使用 IDEA 自带的 Maven 插件，主要是用着方便。

## 3. Maven 配置

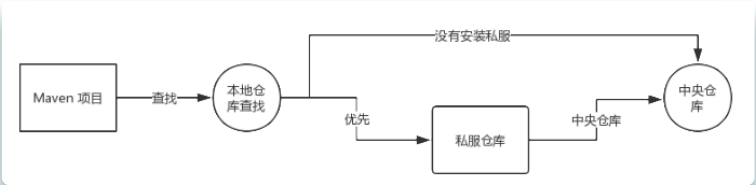
实际上，没有特殊需求的话，安装好之后直接就可以用了。一般来说，还是需要稍微配置一下，比如中央仓库的问题。默认使用 Maven 自己的中央仓库，使用起来网速比较慢，这个时候，可以通过修改配置文件，将仓库改成国内的镜像仓库，国内仓库使用较多的是阿里巴巴的仓库。

### 3.1 仓库类型

仓库类型	说明
本地仓库	就是你自己电脑上的仓库，每个人电脑上都有一个仓库，默认位置在当前用户名\m2\repository
私服仓库	一般来说是公司内部搭建的 Maven 私服，处于局域网中，访问速度较快，这个仓库中存放的 jar 一般就是公司内部自己开发的 jar
中央仓库	有 Apache 团队来维护，包含了大部分的 jar，早期不包含 Oracle 数据库驱动，从 2019 年 8 月开始，包含了 Oracle 驱动



现在存在 3 个仓库,那么 jar 包如何查找呢?



### 3.2 本地仓库配置

本地仓库默认位置在 `当前用户名\.m2\repository` ,这个位置可以自定义,但是不建议大家自定义这个地址,有几个原因:

- 1. 虽然所有的本地的 jar 都放在这个仓库中,但是并不会占用很大的空间。
- 2. 默认的位置比较隐蔽,不容易碰到

技术上来说,当然是可以自定义本地仓库位置的,在 `conf/settings.xml` 中自定义本地仓库位置:

```
<!--
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
  | The path to the local repository maven will use to store artifacts.
  | Default: $(user.home)/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->
  <localRepository>/path/to/local/repo</localRepository>
  <!-- interactiveMode
  | This will determine whether maven prompts you when it needs input. If set to false,
  | maven will use a sensible default value, perhaps based on some other setting, for
  | the parameter in question.
  -->
```

在这个位置配置本地仓库地址

### 3.3 远程镜像配置

由于默认的中央仓库下载较慢,因此,也可以将远程仓库地址改为阿里巴巴的仓库地址:

```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorOf>central</mirrorOf>
  <name>Nexus aliyun</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

这段配置,加在 `settings.xml` 中的 `mirrors` 节点中:

```
<!--
<mirrors>
  <mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>central</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
  </mirror>
  <!-- mirror
  | Specifies a repository mirror site to use instead of a given repository. The repository that
  | this mirror serves has an ID that matches the mirrorOf element of this mirror. IDs are used
  | for inheritance and direct lookup purposes, and must be unique across the set of mirrors.
  |
  <mirror>
    <id>mirrorId</id>
    <mirrorOf>repositoryId</mirrorOf>
    <name>Human Readable Name for this Mirror.</name>
    <url>http://mv.repository.com/repo/path</url>
  </mirror>
  -->
</mirrors>
```

加了这一段配置之后,以后就不会去 Maven 中央仓库上去下载依赖了,而是去阿里巴巴的仓库下载,这样可以有效提高下载速度。

## 4. Maven 常用命令

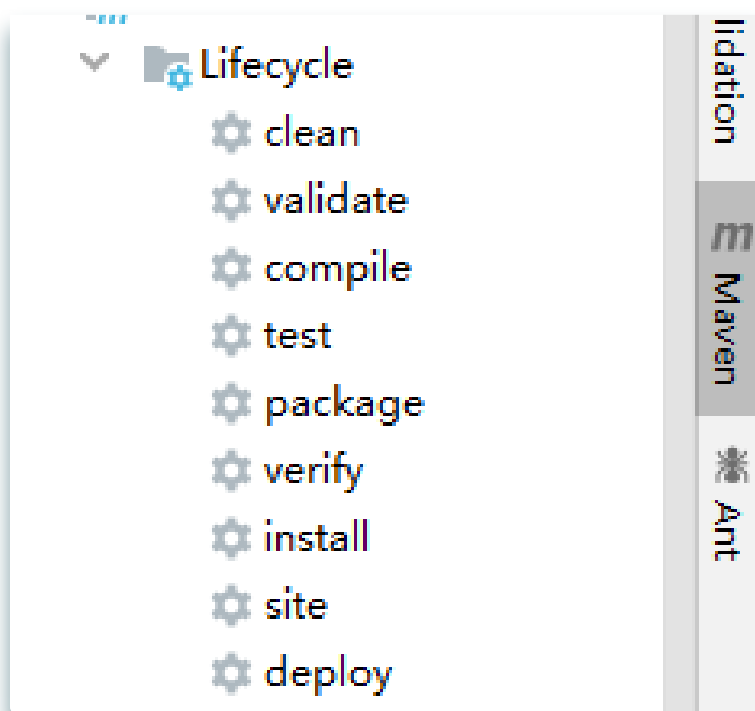
Maven 中有一些常见的命令,如果使用 Eclipse 需要手动敲命令,如果使用 IDEA 的话,可以不用命令,直接点点点就可以了。

常用命令	中文含义说明	
mvn clean	清理	这个命令可以用来清理已经编译好的文件
mvn compile	编译	将 Java 代码编译成 Class 文件
mvn test	测试	项目测试
mvn package	打包	根据用户的配置, 将项目打成 jar 包或者 war 包
mvn install	安装	手动向本地仓库安装一个 jar
mvn deploy	上传	将 jar 上传到私服

这里需要注意的是, 这些命令都不是独立运行的, 它有一个顺序。举个简单例子:

我想将 jar 上传到私服, 那么就要构建 jar, 就需要执行 package 命令, 要打包, 当然也需要测试, 那就要走 mvn test 命令, 要测试就要先编译....., 因此, 最终所有的命令都会执行一遍。不过, 开发者也可以手动配置不执行某一个命令, 这就是跳过。一般是, 除了测试, 其他步骤都不建议跳过。

当然, 如果开发者使用了 IDEA , 这些命令不用手动敲, 点一下就行:



## 4.1 通过命令来构建项目

可以直接通过命令来构建一个 Maven 项目, 不过在实际开发中, 一般使用 Eclipse 或者 IDEA 就可以直接创建 Maven 项目了。

创建命令:

```
mvn archetype:generate -DgroupId=org.javaboy -DartifactId=firstapp -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

```
F:\workspace\workspace\maven\mn archetype:generate -DgroupId=org.javaboy -DartifactId=firstapp -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
[INFO] Scanning for projects...
```

看到如下提示, 表示项目创建成功:

```

[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO]
[INFO] Parameter: basedir, Value: F:\workspace5\workspace\maven
[INFO] Parameter: package, Value: org.javaboy
[INFO] Parameter: groupId, Value: org.javaboy
[INFO] Parameter: artifactId, Value: firstapp
[INFO] Parameter: packageName, Value: org.javaboy
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: F:\workspace5\workspace\maven\firstapp
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 01:28 min
[INFO] Finished at: 2019-12-02T11:08:54+08:00
[INFO]

```

项目创建成功后，就两个文件：

src	2019/12/2 11:08	文件夹	
pom.xml	2019/12/2 11:08	XML 文档	1 KB

说明对一个任何一个项目而言，最最核心的就是这两个。



pom.xml 中，则定义了所有的项目配置。

```

<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.javaboy</groupId>
  <artifactId>firstapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>firstapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

## 4.2 对项目进行打包

接下来，我们通过 mvn package 命令可以将刚刚创建的项目打成一个 jar 包。

在打包之前，需要配置 JDK 的版本至少为 7 以上，因此，我们还需要手动修改一下 pom.xml 文件，即添加如下配置：

```

<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.javaboy</groupId>
  <artifactId>firstapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>firstapp</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.encoding>UTF-8</maven.compiler.encoding>
    <java.version>1.8</java.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

添加完成后，执行打包命令，注意执行所有命令时，命令行要定位到 pom.xml 文件所在的目录，看到如下提示，表示项目打包成功。

```
[INFO] Building jar: F:\workspace5\workspace\maven\firstapp\target\firstapp-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 24.575 s
[INFO] Finished at: 2019-12-02T11:39:36+08:00
[INFO] -----
```

## 4.3 将项目安装到本地仓库

如果要将项目安装到本地仓库,可以直接执行 `mvn install` 命令,注意,`mvn install` 命令会包含上面的 `mvn package` 过程。

```
[INFO] Installing F:\workspace5\workspace\maven\firstapp\pom.xml to C:\Users\javaboy\.m2\repository\org\javaboy\firstapp\1.0-SNAPSHOT\firstapp-1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 23.545 s
[INFO] Finished at: 2019-12-02T14:10:37+08:00
[INFO] -----
```

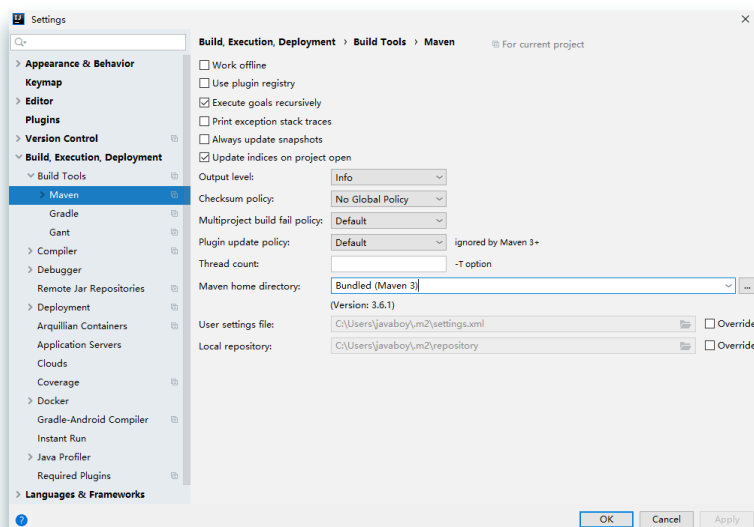
安装到本地仓库之后,这个时候,点开自己的本地仓库,就可以看到相关的 jar 了。

## 5. IDEA 中使用 Maven

不同于 Eclipse,IDEA 安装完成后,就可以直接使用 Maven 了。

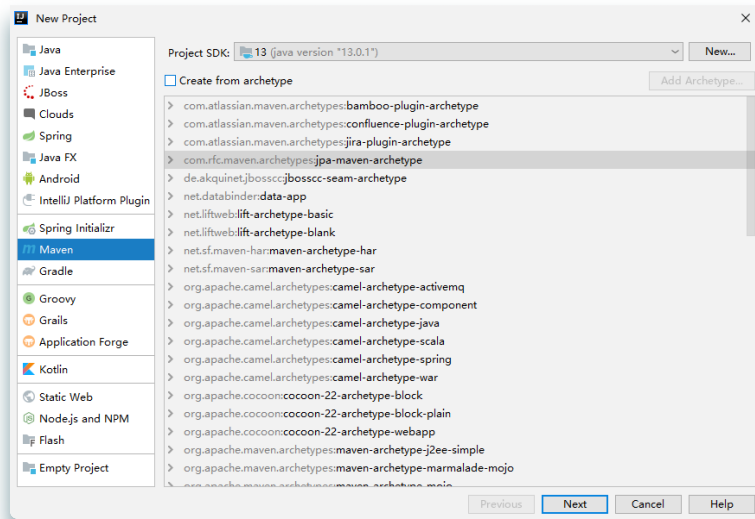
### 5.1 Maven 相关配置

IDEA 中,Maven 的配置在 `File->Settings->Build,Execution,Deployment->Build Tools->Maven` :

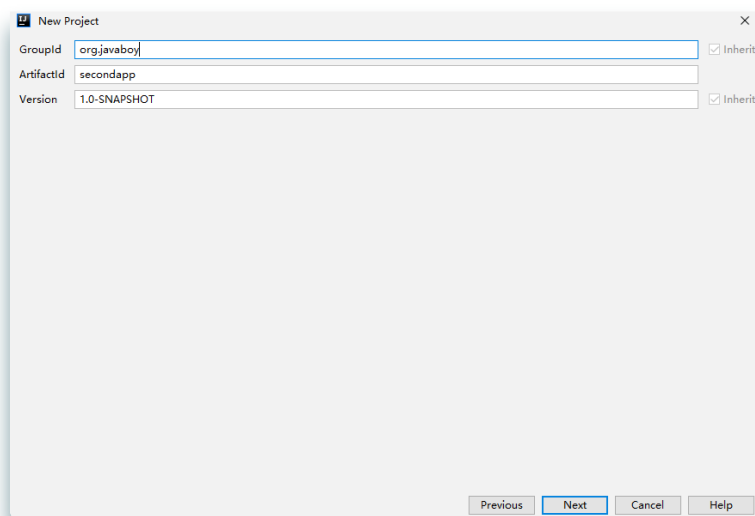


### 5.2 JavaSE 工程创建

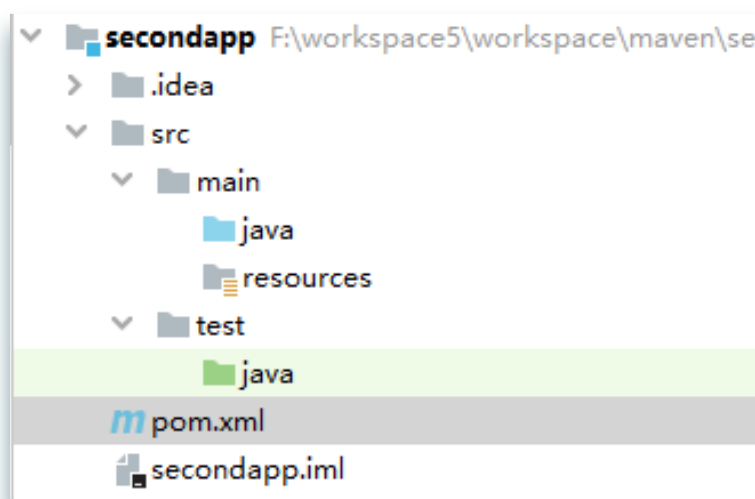
首先在创建一个工程时,选择 `Maven` 工程:



如果勾选上 `Create from archetype`，则表示可以根据一个项目骨架（项目模板）来创建一个新的工程，不过，如果只是创建 `JavaSE` 项目，则不用选择项目骨架。直接 `Next` 即可。然后填入项目的坐标，即 `groupId` 和 `artifactId`。



填完之后，直接 `Next` 即可。这样，我们会获取一个 `JavaSE` 工程，项目结构和你用命令创建出来的项目一模一样。



### 5.3 JavaWeb 工程创建

在 `IDEA` 中，创建 `Maven Web` 项目，有两种思路：

- 首先创建一个 `JavaSE` 项目，然后手动将 `JavaSE` 项目改造成一个 `JavaWeb` 项目
- 创建项目时选择项目骨架，骨架就选择 `webapp`

两种方式中，推荐使用第一种方式。

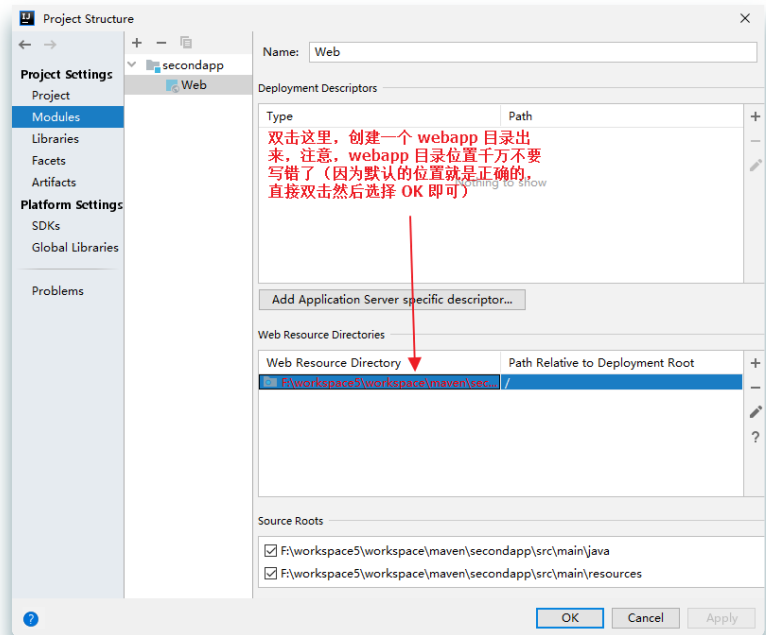
### 5.3.1 改造 JavaSE 项目

这种方式，首先创建一个 JavaSE 项目，创建步骤和上面的一致。

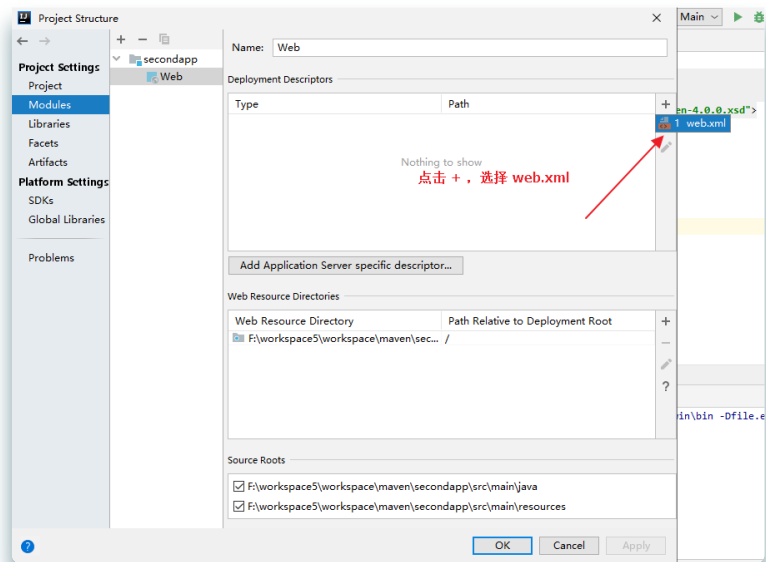
项目创建完成后，**首先修改 pom.xml ，配置项目的打包格式为 war 包。**这样，IDEA 就知道当前项目是一个 Web 项目：



然后，选中 JavaSE 工程，右键单击，选择 Open Module Settings，或者直接按 F4，然后选择 Web，如下图：

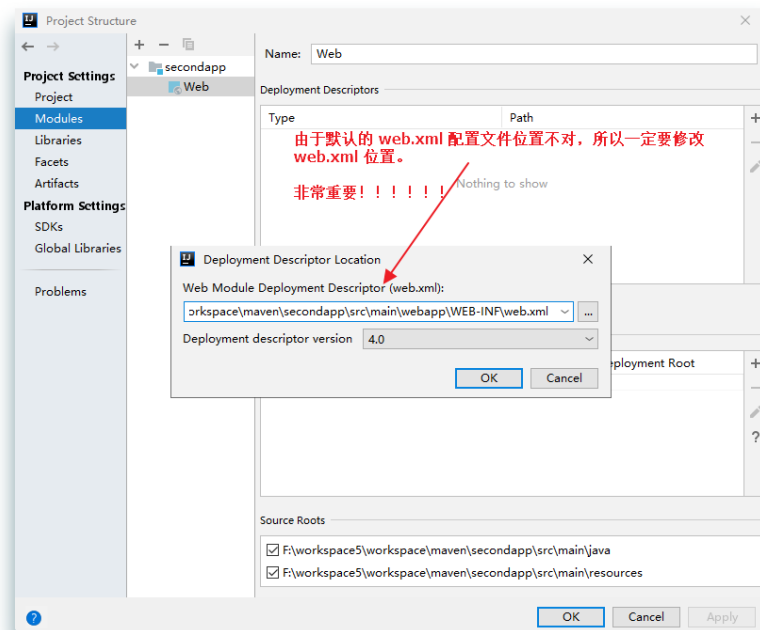


接下来，在 webapp 目录中，添加 web.xml 文件。





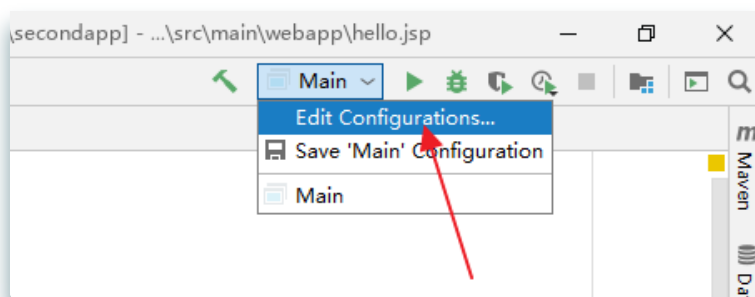
注意，一定要修改 web.xml 文件位置：



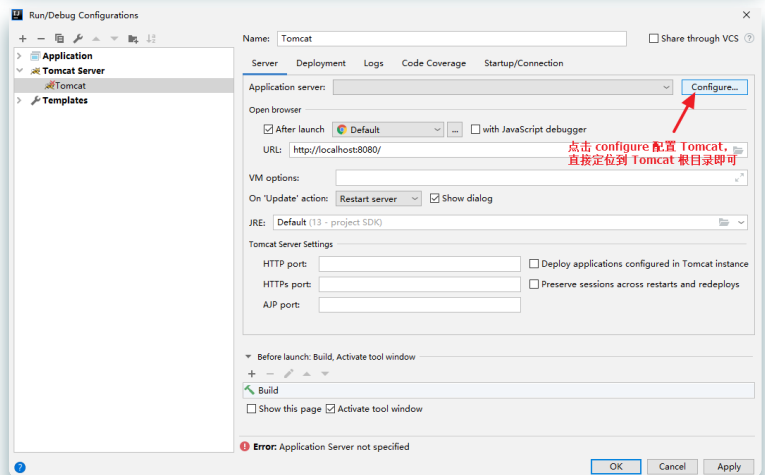
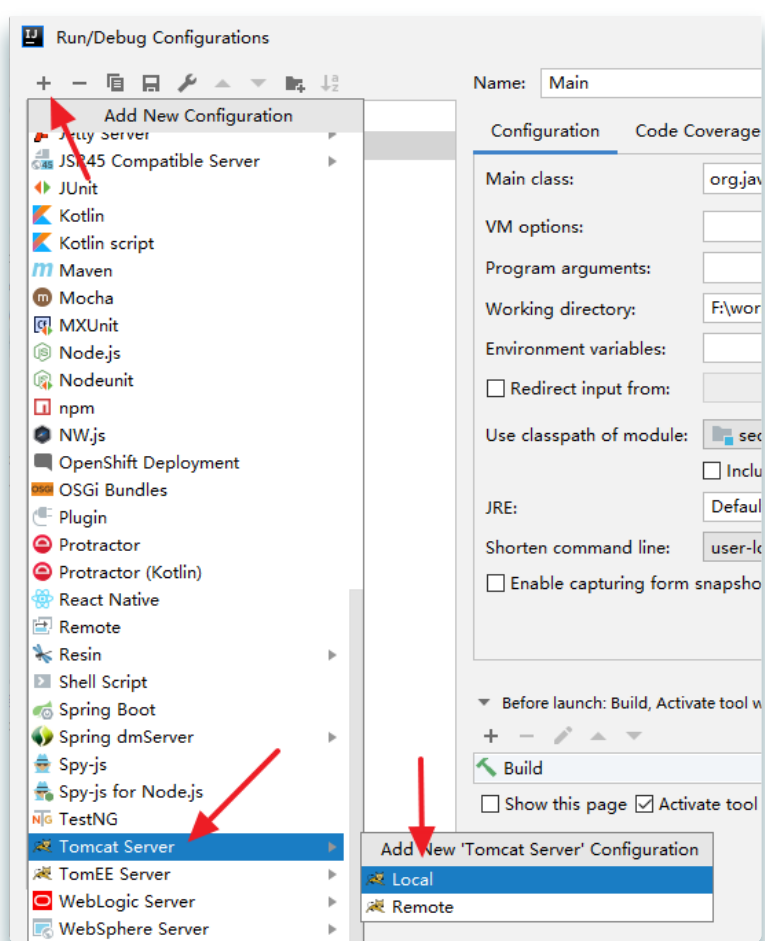
配置完成后，点击 OK 退出。

项目创建完成后，接下来就是部署了。

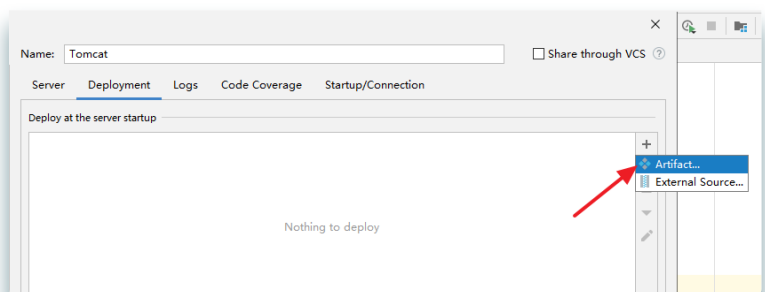
部署，首先点击 IDEA 右上角的 Edit Configurations：

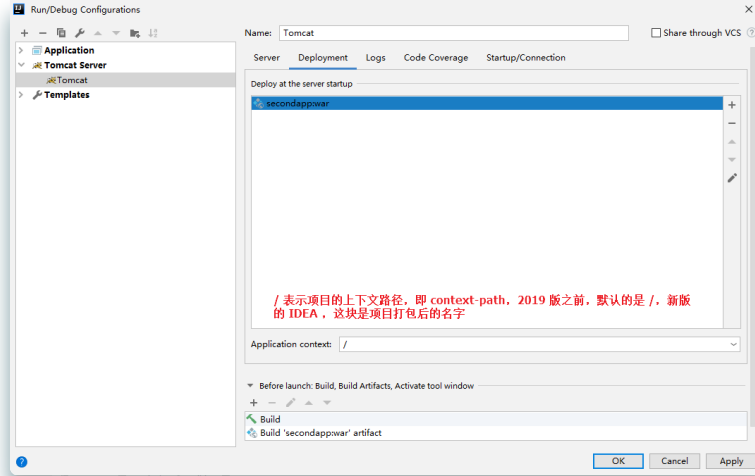


然后，配置 Tomcat：

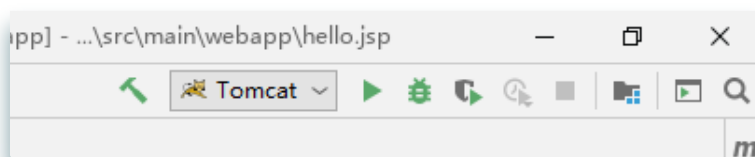


接下来选择 Deployment 选项卡，配置要发布的项目：



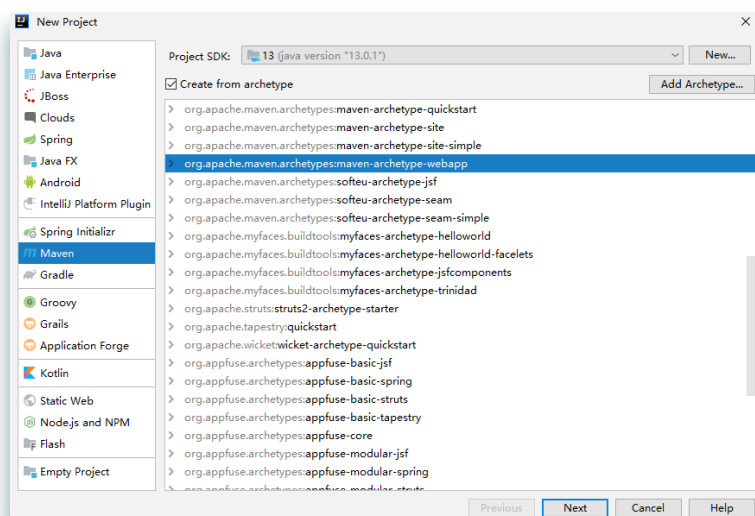


最后，点击 IDEA 右上角的三角符号，启动项目。



### 5.3.2 通过 webapp 骨架直接创建

这种方式比较简单，基本上不需要额外的配置，项目创建完成后，就是一个 web 项目。只需要我们在创建项目时，选择 webapp 骨架即可。



选择骨架之后，后面的步骤和前文一致。

项目创建成功后，只有 webapp 目录，这个时候，自己手动创建 java 和 resources 目录，创建完成后，右键单击，选择 Mark Directory As，将 java 目录标记为 sources root，将 resources 目录标记为 resources root 即可。

凡是在 IDEA 右下角看到了 Enable Auto Import 按钮，一定点一下

## 6. Maven 依赖管理

Maven 项目，如果需要使用第三方的控件，都是通过依赖管理来完成的。这里用到的一个东西就是 pom.xml 文件，概念叫做项目对象模型 (POM, Project Object Model)，我们在 pom.xml 中定义了 Maven 项目的形式，所以，pom.xml 相当于是 Maven 项目的一个地图。就类似于 web.xml 文件用来描述三大 web 组件一样。

这个地图中都涉及到哪些东西呢？

## 6.1 Maven 坐标

```
<dependencies>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
</dependencies>
```

- dependencies

在 dependencies 标签中，添加项目需要的 jar 所对应的 maven 坐标。

- dependency

一个 dependency 标签表示一个坐标

- groupId

团体、公司、组织机构等等的唯一标识。团体标识的约定是它以创建这个项目的组织名称的逆向域名（例如 org.javaboy）开头。一个 Maven 坐标必须要包含 groupId。一些典型的 groupId 如 apache 的 groupId 是 org.apache。

- artifactId

artifactId 相当于在一个组织中项目的唯一标识符。

- version

一个项目的版本。一个项目的话，可能会有多个版本。如果是正在开发的项目，我们可以给版本号加上一个 SNAPSHOT，表示这是一个快照版（新建项目的默认版本号就是快照版）

- scope

表示依赖范围。

依赖范围	编译有效	测试有效	运行时有效	打包有效	例子
Compile	√	√	√	√	spring-core
test	×	√	×	×	JUnit
provided	√	√	×	×	servlet-api
runtime	×	√	√	√	JDBC驱动
system	√	√	×	×	本地maven仓库之外的类库

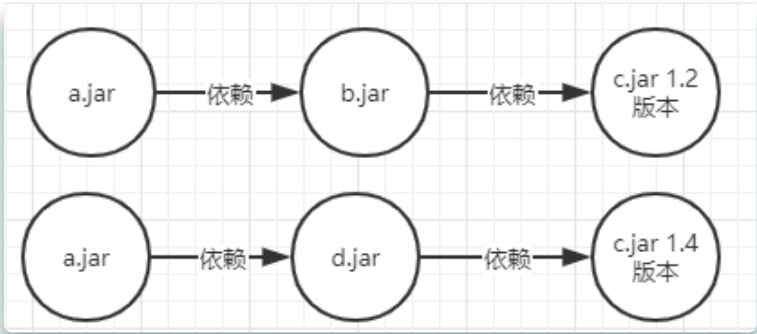
我们添加了很多依赖，但是不同依赖的使用范围是不一样的。最典型的有两个，一个是数据库驱动，另一个是单元测试。

数据库驱动，在使用的过程中，我们自己写代码，写的是 JDBC 代码，只有在项目运行时，才需要执行 MySQL 驱动中的代码。所以，MySQL 驱动这个依赖在添加到项目中之后，可以设置它的 scope 为 runtime，编译的时候不生效。

单元测试，只在测试的时候生效，所以可以设置它的 scope 为 test，这样，当项目打包发布时，单元测试的依赖就不会跟着发布。

## 6.2 依赖冲突

- 依赖冲突产生的原因

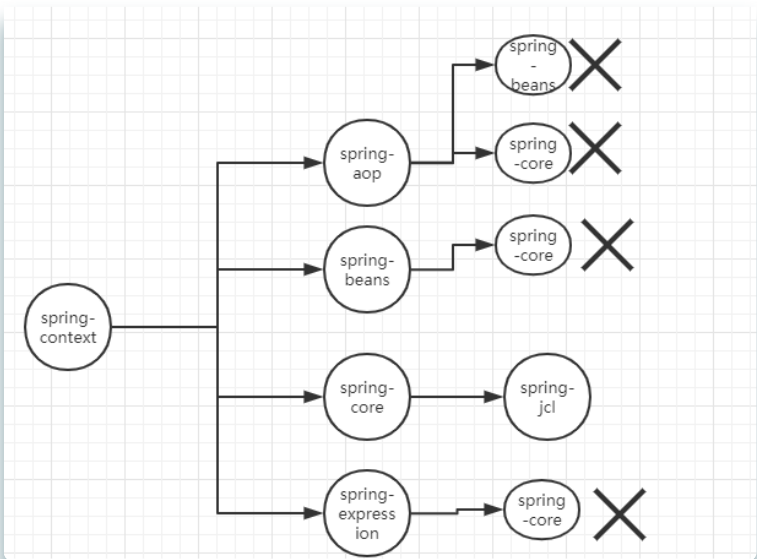


在图中，a.jar 依赖 b.jar，同时 a.jar 依赖 d.jar，这个时候，a 和 b、d 的关系是直接依赖的关系，a 和 c 的关系是间接依赖的关系。

### 6.2.1 冲突解决

1. 先定义先使用
2. 路径最近原则（直接声明使用）

以 spring-context 为例，下图中 x 表示失效的依赖（优先级低的依赖，即路径近的依赖优先使用）：



上面这两条是默认行为。

我们也可以手动控制。手动控制主要是通过排除依赖来实现，如下：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.1.9.RELEASE</version>
  <exclusions>
    <exclusion>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

这个表示从 `spring-context` 中排除 `spring-core` 依赖。

## 7. Maven 私服

Maven 仓库管理也叫 Maven 私服或者代理仓库。使用 Maven 私服有两个目的：

1. 私服是一个介于开发者和远程仓库之间的代理
2. 私服可以用来部署公司自己的 jar

### 7.1 Nexus 介绍

Nexus 是一个强大的 Maven 仓库管理工具，使用 Nexus 可以方便的管理内部仓库同时简化外部仓库的访问。官网是：<https://www.sonatype.com/>

### 7.2 安装

- 下载

下载地址：<https://www.sonatype.com/download-oss-sonatype>

- 解压

将下载下来的压缩包，拷贝到一个没有中文的路径下，然后解压。

- 启动

解压之后，打开 cmd 窗口（以管理员身份打开 cmd 窗口），然后定位了 nexus 解压目录，执行 `nexus.exe/run` 命令启动服务。

```
F:\nexus-3.14.0-04-win64\nexus-3.14.0-04\bin\nexus.exe/run
Preparing JRE ...
2019-12-03 09:57:16,929+0800 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.pax.logging.NexusLogActivator - start
2019-12-03 09:57:20,508+0800 WARN [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4J[Initialis
er - Your logging framework class org.ops4j.pax.logging.slf4j.Slf4jLogger is not known - if it needs access to the stand
ard println methods on the console you will need to register it by calling registerLoggingSystemPackage
2019-12-03 09:57:20,511+0800 INFO [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4J - Packag
e org.ops4j.pax.logging.slf4j registered, all classes within it or subpackages of it will be allowed to print to System
.out and System.err
```



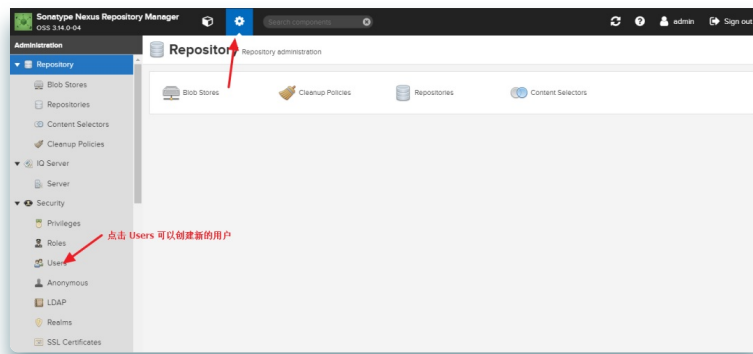
这个启动稍微有点慢，大概有 1 两分钟的样子

”

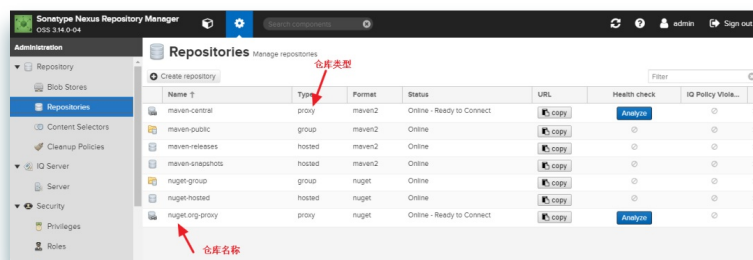
启动成功后，浏览器输入 `http://localhost:8081` 打开管理页面。

打开管理页面后，点击右上角上的登录按钮进行登录，默认的用户名/密码是 `admin/admin123`。当然，用户也可以点击设置按钮，手动配置其他用户。





点击 Repositories 可以查看仓库详细信息：



## 7.2.1 仓库类型

名称	说明
proxy	表示这个仓库是一个远程仓库的代理，最典型的就是代理 Maven 中央仓库
hosted	宿主仓库，公司自己开发的一些 jar 存放在宿主仓库中，以及一些在 Maven 中央仓库上没有的 jar
group	仓库组，包含代理仓库和宿主仓库
virtual	虚拟仓库

## 7.2.2 上传 jar

上传 jar，配置两个地方：

- Maven 的 conf/settings.xml 文件配置：

```
<server>
<id>releases</id>
<username>admin</username>
<password>admin123</password>
</server>
<server>
<id>snapshots</id>
<username>admin</username>
<password>admin123</password>
</server>
```

在要上传 jar 的项目的 pom.xml 文件中，配置上传路径：

```
<distributionManagement>
  <repository>
    <id>releases</id>
    <url>http://localhost:8081/repository/maven-releases/</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <url>http://localhost:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

配置完成后, 点击 `deploy` 按钮, 或者执行 `mvn deploy` 命令就可以将 `jar` 上传到私服上。

### 7.2.3 下载私服上的 jar

直接在项目中添加依赖, 添加完成后, 额外增加私服地址即可:

```
<repositories>
  <repository>
    <id>local-repository</id>
    <url>http://localhost:8081/repository/maven-public/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
  </repository>
</repositories>
```

## 8. 聚合工程

所谓的聚合工程, 实际上也就是多模块项目。在一个比较大的互联网项目中, 项目需要拆分成多个模块进行开发, 比如订单模块、VIP 模块、支付模块、内容管理模块、CMS、CRM 等等。这种拆分方式, 实际上更接近于微服务的思想。在一个模块中, 还可以继续进行拆分, 例如分成 `dao`、`service`、`controller` 等。

有人可能会说, 这个分包不就行了吗?

小项目当然可以分包, 大项目就没法分包了。比如, 在一个大的电商系统中, 有一个子模块叫做用户管理、还有一个子模块叫做订单管理, 这两个子模块都涉及到用户, 像这种情况, 我们就需要将用户类单独提取出来, 做成单独的模块, 供其他模块调用。

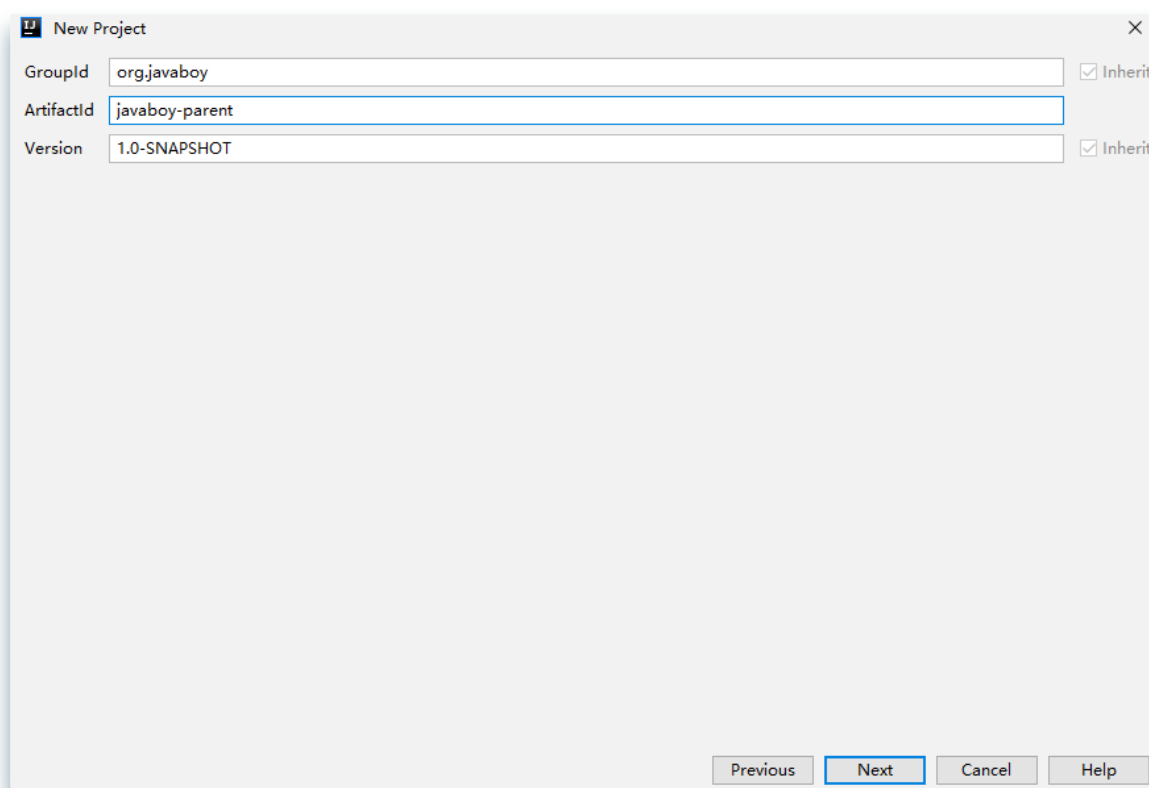
### 8.1 多模块项目展示

```
|--javaboy-parent
  |-- javaboy-cms
  |-- javaboy-crm
  |-- javaboy-manger
  |-- javaboy-manager-model
  |-- javaboy-manager-dao
  |-- javaboy-manager-service
  |-- javaboy-manager-web
```

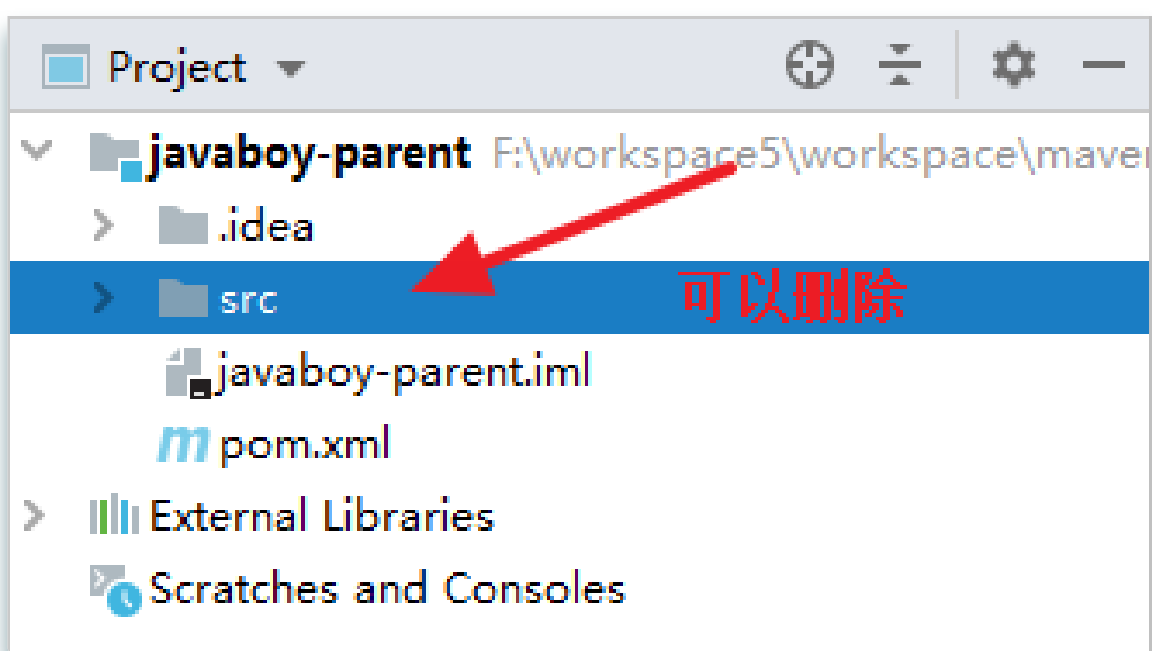
以 `javaboy-manger` 为例, `javaboy-manager` 本身并不提供功能, 它只负责管理他自己的子模块, 而他的子模块每一个都无法独立运行, 需要四个结合在一起, 才可以运行。项目打包时, `model`、`dao`、`service` 都将打包成 `jar`, 然后会自动将打包好的 `jar` 复制到 `web` 中, 再自动将 `web` 打包成 `war` 包。

## 8.2 IDEA 中创建聚合工程

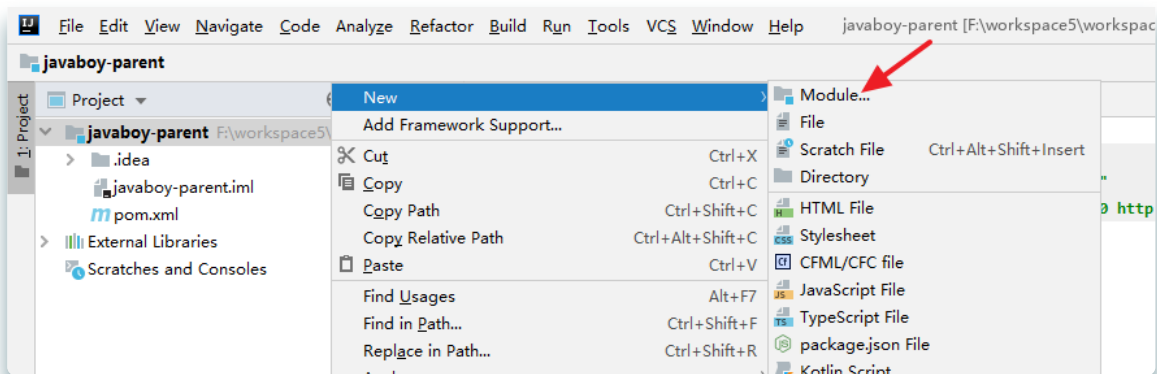
### 1. 创建一个空的 Maven 项目：



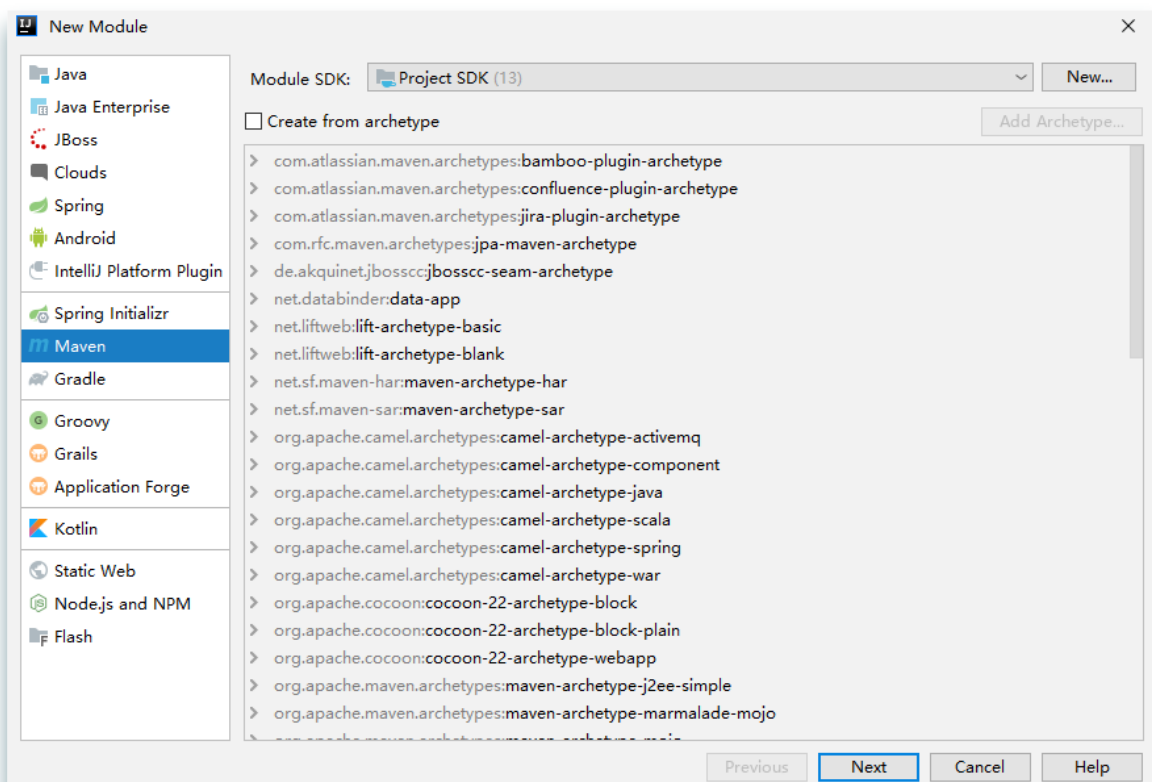
项目创建完成后, 由于 `parent` 并不参与业务的实现, 只是用来管理它的子模块, 因此, `src` 目录可以将其删除。



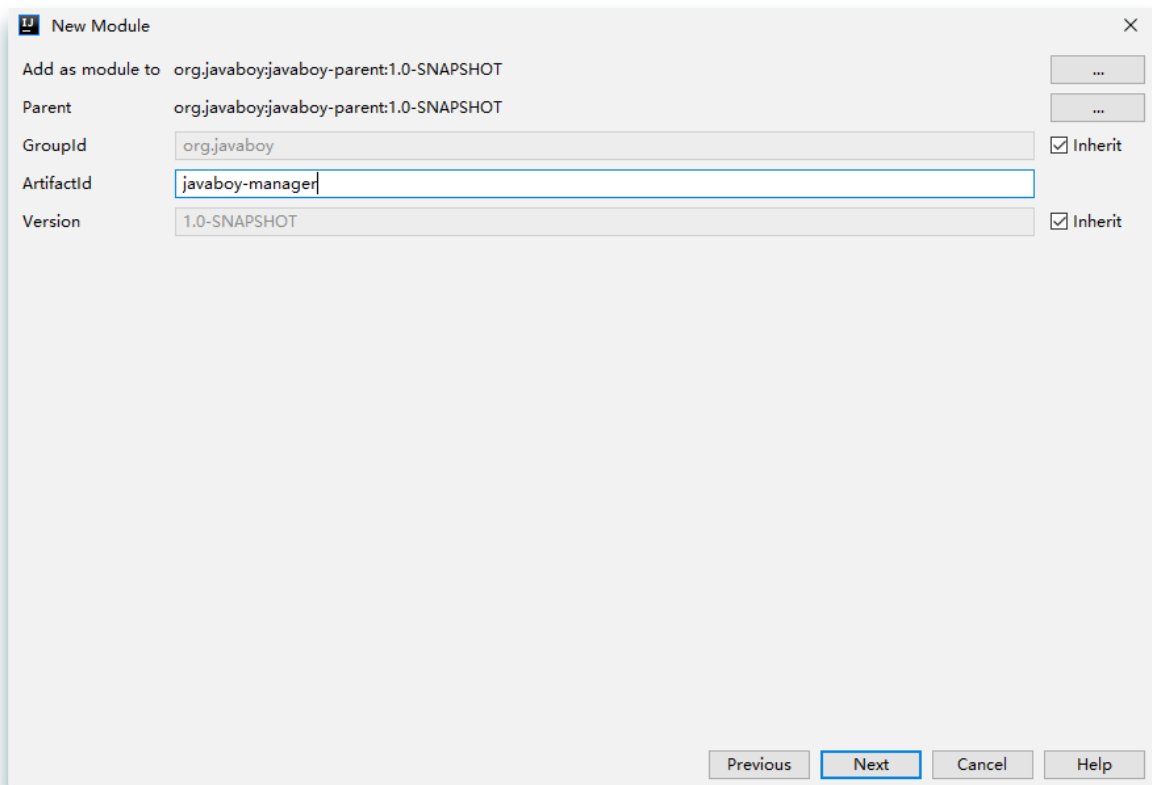
2. 选中当前工程，右键单击，New->Module



然后继续选择创建一个 Maven 项目：



在 IDEA 中，已经默认指明了当前 Module 的 parent，开发者只需要填入当前 Module 的 artifactId 即可：



javaboy-manager 创建完成后,此时,观察 javaboy-parent 的 pom.xml 文件,发现它自动加上了 packaging 属性:

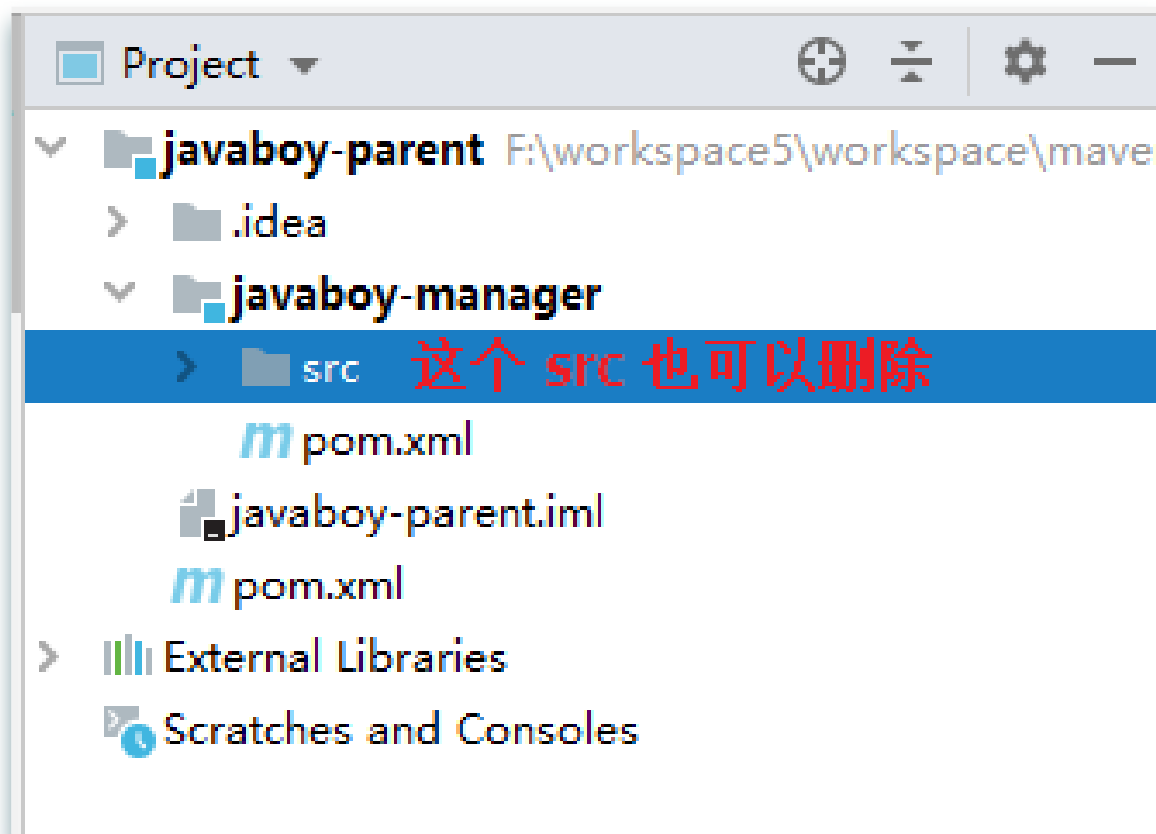


其中,它的 packaging 属性值为 pom,这表示它是一个聚合工程,同时,他还多了 modules 节点,指明了它自己的子模块。同时,注意 javaboy-manager,它自身多了一个 parent 节点,这个 parent 节点描述了它的父模块的属性值:

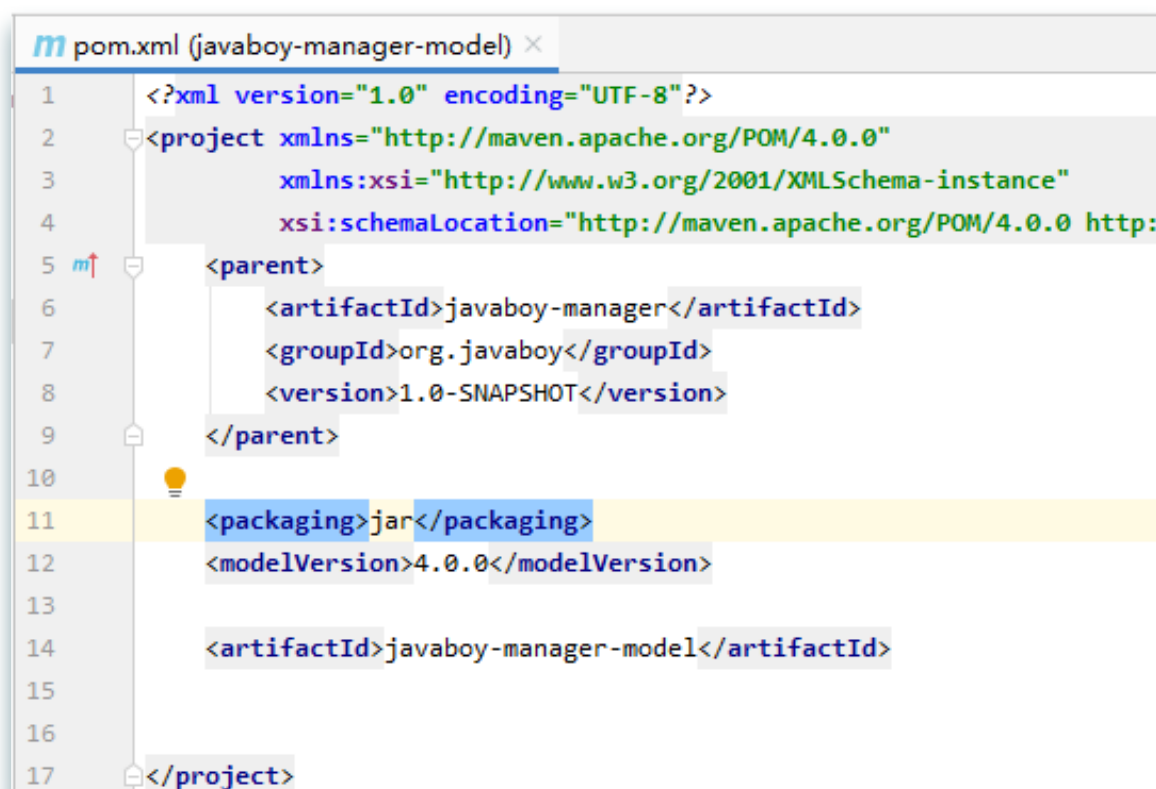
```
<parent>
  <artifactId>javaboy-parent</artifactId>
  <groupId>org.javaboy</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

这个 parent 不仅仅是一个简单的父子关系描述,它存在继承关系,一般我们可以在 parent 中统一

3. 由于 javaboy-manager 本身也是一个聚合工程，因此，javaboy-manager 的 src 目录也可以删除。



4. 选中 javaboy-manager，右键单击，New->Module 创建一个新的 Maven 模块出来。这个步骤类似于第二步，不在赘述。这里，新的 javaboy-manager-model 创建成功后，我们手动配置它的 packaging 属性值为 jar。



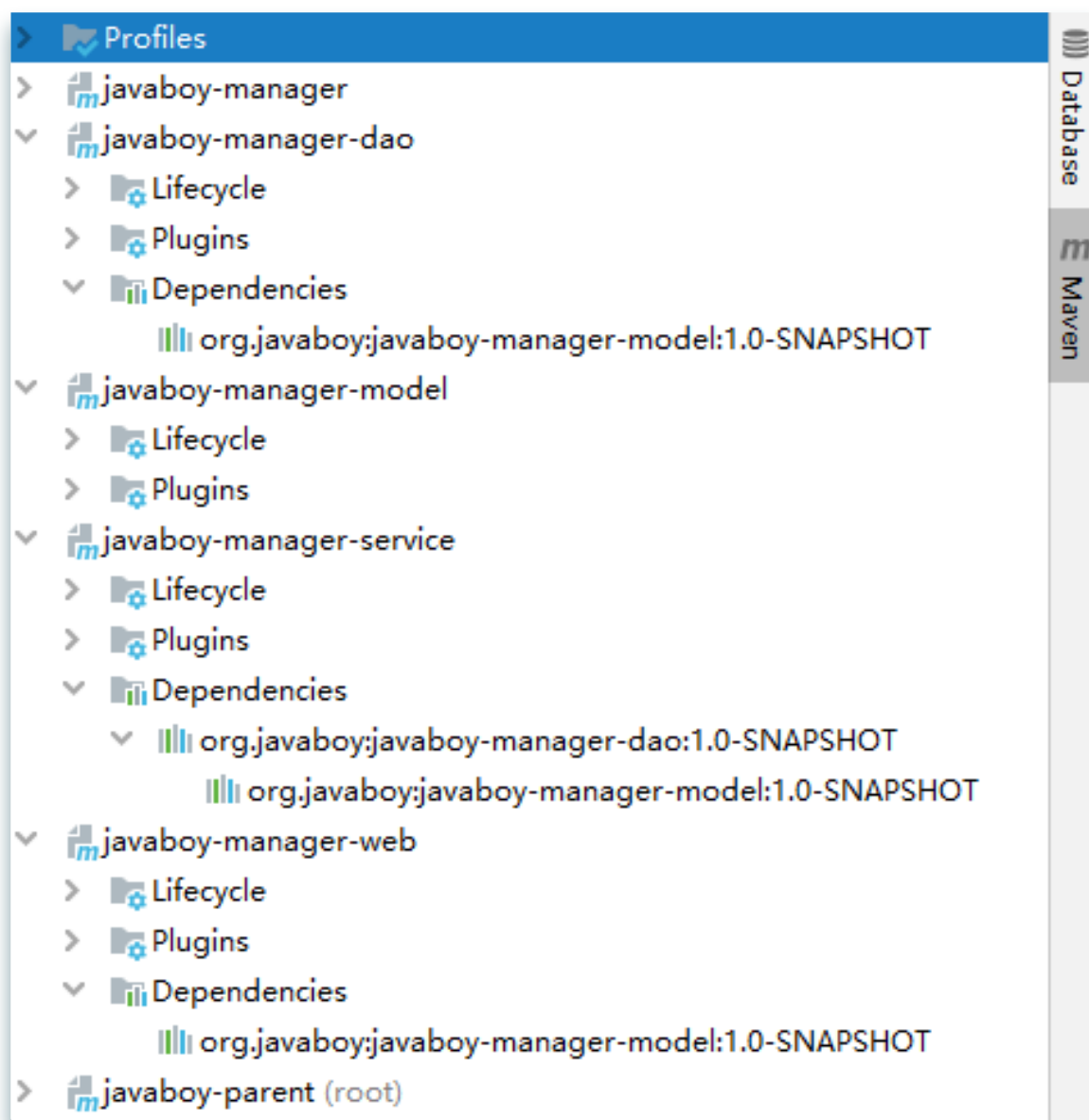
5. 依照第 4 步，再分别创建 javaboy-manager-service 以及 javaboy-manager-dao 6. 继续创建 javaboy-manager-web 模块，不同于其他模块，web 模块需要打包成 war。web 模块创建可以参考【第五篇文章】。7. web 工程创建完成后，完善模块之间的继承关系。



javaboy-manager-web 依赖 javaboy-manager-service  
javaboy-manager-service 依赖 javaboy-manager-dao  
javaboy-manager-dao 依赖 javaboy-manager-model

注意，依赖默认是有传递性的，即在 javaboy-manager-dao 中依赖了 javaboy-manager-model，在 javaboy-manager-service 也能访问到。

配置后的依赖关系如下图：



接下来就可以在不同的模块中写代码，然后进行项目部署了。部署方式参考【第五篇文章】

有一个需要注意的地方，在多模块项目中，web 项目打包需要注意以下问题：

1. 不可以直接单独打包
2. 如果要打包，有两种方式：
  - 第一种就是先手动挨个将 model、dao、service 安装到本地仓库
  - 从聚合工程处打包，即从 web 的 parent 处打包。

- END -

## 推荐阅读

1. 我在华为做外包的真实经历！
2. HashMap 线程不安全的体现
3. 使用 Redis 搭建电商秒杀系统

4. 什么是一致性 Hash 算法?

5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 🍷

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!

# 常用 Maven 插件介绍（收藏大全）

种菜得瓜 Java后端 2019-11-08

点击上方 Java后端, 选择 设为星标

优质文章, 及时送达

作者 | 种菜得瓜

链接 | [cnblogs.com/crazy-fox](https://cnblogs.com/crazy-fox)

我们都知道Maven本质上是一个插件框架，它的核心并不执行任何具体的构建任务，所有这些任务都交给插件来完成，例如编译源代码是由maven-compiler-plugin完成的。进一步说，每个任务对应了一个插件目标(goal)，每个插件会有一个或者多个目标，例如maven-compiler-plugin的compile目标用来编译位于src/main/java/目录下的主源码，testCompile目标用来编译位于src/test/java/目录下的测试源码。

用户可以通过两种方式调用Maven插件目标。第一种方式是将插件目标与生命周期阶段(lifecycle phase)绑定，这样用户在命令行只是输入生命周期阶段而已，例如Maven默认将maven-compiler-plugin的compile目标与 compile生命周期阶段绑定，因此命令mvn compile实际上是先定位到compile这一生命周期阶段，然后再根据绑定关系调用maven-compiler-plugin的compile目标。第二种方式是直接在命令行指定要执行的插件目标，例如mvn archetype:generate 就表示调用maven-archetype-plugin的generate目标，这种带冒号的调用方式与生命周期无关。

认识上述Maven插件的基本概念能帮助你理解Maven的工作机制，不过要想更高效地使用Maven，了解一些常用的插件还是很有必要的，这可以帮助你避免一不小心重新发明轮子。多年来Maven社区积累了大量的经验，并随之形成了一个成熟的插件生态圈。Maven官方有两个插件列表，第一个列表的GroupId为org.apache.maven.plugins，这里的插件最为成熟，具体地址为：<http://maven.apache.org/plugins/index.html>。第二个列表的GroupId为org.codehaus.mojo，这里的插件没有那么核心，但也有不少十分有用，其地址为：<http://mojo.codehaus.org/plugins.html>。

接下来笔者根据自己的经验介绍一些最常用的Maven插件，在不同的环境下它们各自都有其出色的表现，熟练地使用它们能让你的日常构建工作事半功倍。

## maven-antrun-plugin

<http://maven.apache.org/plugins/maven-antrun-plugin/>

maven-antrun-plugin能让用户在Maven项目中运行Ant任务。用户可以直接在该插件的配置以Ant的方式编写Target，然后交给该插件的run目标去执行。在一些由Ant往Maven迁移的项目中，该插件尤其有用。此外当你发现需要编写一些自定义程度很高的任务，同时又觉得Maven不够灵活时，也可以以Ant的方式实现之。maven-antrun-plugin的run目标通常与生命周期绑定运行。

## maven-archetype-plugin

<http://maven.apache.org/archetype/maven-archetype-plugin/>

Archtype指项目的骨架,Maven初学者最开始执行的Maven命令可能就是`mvn archetype:generate`,这实际上就是让`maven-archetype-plugin`生成一个很简单的项目骨架,帮助开发者快速上手。可能也有人看到一些文档写了`mvn archetype:create`,但实际上`create`目标已经被弃用了,取而代之的是`generate`目标,该目标使用交互式的方式提示用户输入必要的信息以创建项目,体验更好。`maven-archetype-plugin`还有一些其他目标帮助用户自己定义项目原型,例如你由一个产品需要交付给很多客户进行二次开发,你就可以为他们提供一个Archtype,帮助他们快速上手。

## maven-assembly-plugin

---

<http://maven.apache.org/plugins/maven-assembly-plugin/>

`maven-assembly-plugin`的用途是制作项目分包,该分包可能包含了项目的可执行文件、源代码、`readme`、平台脚本等等。`maven-assembly-plugin`支持各种主流的格式如`zip`、`tar.gz`、`jar`和`war`等,具体打包哪些文件是高度可控的,例如用户可以按文件级别的粒度、文件集级别的粒度、模块级别的粒度、以及依赖级别的粒度控制打包,此外,包含和排除配置也是支持的。`maven-assembly-plugin`要求用户使用一个名为`assembly.xml`的元数据文件来表述打包,它的`single`目标可以直接在命令行调用,也可以被绑定至生命周期。

## maven-dependency-plugin

---

<http://maven.apache.org/plugins/maven-dependency-plugin/>

`maven-dependency-plugin`最大的用途是帮助分析项目依赖,`dependency:list`能够列出项目最终解析到的依赖列表,`dependency:tree`能进一步的描绘项目依赖树,`dependency:analyze`可以告诉你项目依赖潜在的问题,如果你有直接使用到的却未声明的依赖,该目标就会发出警告。`maven-dependency-plugin`还有很多目标帮助你操作依赖文件,例如`dependency:copy-dependencies`能将项目依赖从本地Maven仓库复制到某个特定的文件夹下面。

## maven-enforcer-plugin

---

<http://maven.apache.org/plugins/maven-enforcer-plugin/>

在一个稍大一点的组织或团队中,你无法保证所有成员都熟悉Maven,那他们做一些比较愚蠢的事情就会变得很正常,例如给项目引入了外部的SNAPSHOT依赖而导致构建不稳定,使用了一个与大家不一致的Maven版本而经常抱怨构建出现诡异问题。`maven-enforcer-plugin`能够帮助你避免之类问题,它允许你创建一系列规则强制大家遵守,包括设定Java版本、设定Maven版本、禁止某些依赖、禁止SNAPSHOT依赖。只要在一个父POM配置规则,然后让大家继承,当规则遭到破坏的时候,Maven就会报错。除了标准的规则之外,你还可以扩展该插件,编写自己的规则。`maven-enforcer-plugin`的`enforce`目标负责检查规则,它默认绑定到生命周期的`validate`阶段。

## maven-help-plugin

---

<http://maven.apache.org/plugins/maven-help-plugin/>

maven-help-plugin是一个小巧的辅助工具，最简单的help:system可以打印所有可用的环境变量和Java系统属性。

help:effective-pom和help:effective-settings最为有用，它们分别打印项目的有效POM和有效settings，有效POM是指合并了所有父POM（包括Super POM）后的XML，当你不确定POM的某些信息从何而来时，就可以查看有效POM。有效settings同理，特别是当你发现自己配置的settings.xml没有生效时，就可以用help:effective-settings来验证。此外，maven-help-plugin的describe目标可以帮助你描述任何一个Maven插件的信息，还有all-profiles目标和active-profiles目标帮助查看项目的Profile。

Tips：关注微信公众号：Java后端，获取每日技术博文推送。

## maven-release-plugin

---

<http://maven.apache.org/plugins/maven-release-plugin/>

maven-release-plugin的用途是帮助自动化项目版本发布，它依赖于POM中的SCM信息。release:prepare用来准备版本发布，具体的工作包括检查是否有未提交代码、检查是否有SNAPSHOT依赖、升级项目的SNAPSHOT版本至RELEASE版本、为项目打标签等等。release:perform则是签出标签中的RELEASE源码，构建并发布。版本发布是非常琐碎的工作，它涉及了各种检查，而且由于该工作仅仅是偶尔需要，因此手动操作很容易遗漏一些细节，maven-release-plugin让该工作变得非常快速简便，不易出错。maven-release-plugin的各种目标通常直接在命令行调用，因为版本发布显然不是日常构建生命周期的一部分。

## maven-resources-plugin

---

<http://maven.apache.org/plugins/maven-resources-plugin/>

为了使项目结构更为清晰，Maven区别对待Java代码文件和资源文件，maven-compiler-plugin用来编译Java代码，maven-resources-plugin则用来处理资源文件。默认的主资源文件目录是src/main/resources，很多用户会需要添加额外的资源文件目录，这个时候就可以通过配置maven-resources-plugin来实现。此外，资源文件过滤也是Maven的一大特性，你可以在资源文件中使用\${propertyName}形式的Maven属性，然后配置maven-resources-plugin开启对资源文件的过滤，之后就可以针对不同环境通过命令行或者Profile传入属性的值，以实现更为灵活的构建。

## maven-surefire-plugin

---

<http://maven.apache.org/plugins/maven-surefire-plugin/>

可能是由于历史的原因，Maven 2/3中用于执行测试的插件不是maven-test-plugin，而是maven-surefire-plugin。其实大部分时间内，只要你的测试类遵循通用的命令约定（以Test结尾、以TestCase结尾、或者以Test开头），就几乎不用知晓该插件的存在。然而在当你想要跳过测试、排除某些测试类、或者使用一些TestNG特性的时候，了解maven-surefire-plugin的一些配置选项就很有用了。例如 `mvn test -Dtest=FooTest` 这样一条命令的效果是仅运行FooTest测试类，这是通过控制maven-surefire-plugin的test参数实现的。

## build-helper-maven-plugin

---

<http://mojo.codehaus.org/build-helper-maven-plugin/>

Maven默认只允许指定一个主Java代码目录和一个测试Java代码目录，虽然这其实是个应当尽量遵守的约定，但偶尔你还是会希望能够指定多个 源码目录（例如为了应对遗留项目），build-helper-maven-plugin的add-source目标就是服务于这个目的，通常它被绑定到 默认生命周期的generate-sources阶段以添加额外的源码目录。需要强调的是，这种做法还是不推荐的，因为它破坏了 Maven的约定，而且可能会遇到其他严格遵守约定的插件工具无法正确识别额外的源码目录。

build-helper-maven-plugin的另一个非常有用的目标是attach-artifact，使用该目标你可以以classifier的形式选取部分项目文件生成附属构件，并同时install到本地仓库，也可以deploy到远程仓库。

## exec-maven-plugin

---

<http://mojo.codehaus.org/exec-maven-plugin/>

exec-maven-plugin很好理解，顾名思义，它能让你运行任何本地的系统程序，在某些特定情况下，运行一个Maven外部的程序可能就是最简单的问题解决方案，这就是exec:exec的 用途，当然，该插件还允许你配置相关的程序运行参数。除了exec目标之外，exec-maven-plugin还提供了java目标，该目标要求你 提供一个mainClass参数，然后它能够利用当前项目的依赖作为classpath，在同一个JVM中运行该mainClass。有时候，为了简单的 演示一个命令行Java程序，你可以在POM中配置好exec-maven-plugin的相关运行参数，然后直接在命令运行 mvn exec:java 以查看运行效果。

## jetty-maven-plugin

---

[http://wiki.eclipse.org/Jetty/Feature/Jetty\\_Maven\\_Plugin](http://wiki.eclipse.org/Jetty/Feature/Jetty_Maven_Plugin)

在进行Web开发的时候，打开浏览器对应用进行手动的测试几乎是无法避免的，这种测试方法通常就是将项目打包成war文件，然后部署到Web容器 中，再启动容器进行验证，这显然十分耗时。为了帮助开发者节省时间，jetty-maven-plugin应运而生，它完全兼容 Maven项目的目录结构，能够周期性地检查源文件，一旦发现变更后自动更新到内置的Jetty Web容器中。做一些基本配置后（例如Web应用的contextPath和自动扫描变更的时间间隔），你只要执行 mvn jetty:run ，然后在IDE中修改代码，代码经IDE自动编译后产生变更，再由jetty-maven-plugin侦测到后更新至Jetty容器，这时你就可以直接 测试Web页面了。需要注意的是，jetty-maven-plugin并不是宿主于Apache或Codehaus的官方插件，因此使用的时候需要额外 的配置settings.xml的pluginGroups元素，将org.mortbay.jetty这个pluginGroup加入。

## versions-maven-plugin

---

<http://mojo.codehaus.org/versions-maven-plugin/>

很多Maven用户遇到过这样一个问题，当项目包含大量模块的时候，为他们集体更新版本就变成一件烦人的事情，到底有没有自动化工具能帮助完成这件事情呢？（当然你可以使用sed之类的文本操作工具，不过不在本文讨论范围）答案是肯定的，versions-maven- plugin提供了很多目标帮助你管理Maven项目的各种版本信息。例如最常用的，命令 mvn versions:set -DnewVersion=1.1-SNAPSHOT 就能帮助你把所有模块的版本更新到1.1-SNAPSHOT。该插件还提供了其他一些很有用的目标，display-dependency- updates能告诉你项目依赖有哪些可用的更新；类似的display-plugin-updates能告诉你可用的插件

更新;然后use- latest-versions能自动帮你将所有依赖升级到最新版本。最后,如果你对所做的更改满意,则可以使用 mvn versions:commit 提交,不满意的话也可以使用 mvn versions:revert 进行撤销。

## 小结

---

本文介绍了一些最常用的Maven插件,这里指的“常用”是指经常需要进行配置的插件,事实上我们用Maven的时候很多其它插件也是必须的,例如 默认的编译插件maven-compiler-plugin和默认的打包插件maven-jar-plugin,但因为很少需要对它们进行配置,因此不在 本文讨论范围。了解常用的Maven插件能帮助你事半功半地完成项目构建任务,反之你就可能会因为经常遇到一些难以解决的问题而感到沮丧。本文介绍的插件 基本能覆盖大部分Maven用户的日常使用需要,如果你真有非常特殊的需求,自行编写一个Maven插件也不是难事,更何况还有这么多开放源代码的插件供 你参考。

本文的这个插件列表并不是一个完整列表,读者有兴趣的话也可以去仔细浏览一下Apache和Codehaus Mojo的Maven插件列表,以的到一个更为全面的认识。最后,在线的Maven仓库搜索引擎如<http://search.maven.org/>也能帮助你快速找到自己感兴趣的Maven插件。

---

- END -

如果看到这里,说明你喜欢这篇文章,请**转发、点赞**。微信搜索「web\_resource」,关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓





1. Java 代码是如何一步步输出结果的?
2. IntelliJ IDEA 详细图解最常用的配置
3. Maven 实战问题和最佳实践
4. 12306 的架构到底有多牛逼?
5. 团队开发中 Git 最佳实践



学Java, 请关注公众号: Java后端

喜欢文章, 点个在看 

声明: pdf仅供学习使用, 一切版权归原创公众号所有; 建议持续关注原创公众号获取最新文章, 学习愉快!



# 试试 IDEA 解决 Maven 依赖冲突的高能神器！

桔子 Java后端 2019-12-13

点击上方 Java后端, 选择 设为星标

优质文章, 及时送达

作者 | 桔子

链接 | [segmentfault.com/a/1190000017542396](https://segmentfault.com/a/1190000017542396)

## 1、何为依赖冲突

Maven是个很好用的依赖管理工具，但是再好的东西也不是完美的。Maven的依赖机制会导致Jar包的冲突。举个例子，现在你的项目中，使用了两个Jar包，分别是A和B。现在A需要依赖另一个Jar包C，B也需要依赖C。但是A依赖的C的版本是1.0，B依赖的C的版本是2.0。这时候，Maven会将这1.0的C和2.0的C都下载到你的项目中，这样你的项目中就存在了不同版本的C，这时Maven会依据依赖路径最短优先原则，来决定使用哪个版本的Jar包，而另一个无用的Jar包则未被使用，这就是所谓的依赖冲突。

在大多数时候，依赖冲突可能并不会对系统造成什么异常，因为Maven始终选择了一个Jar包来使用。但是，不排除在某些特定条件下，会出现类似找不到类的异常，所以，只要存在依赖冲突，在我看来，最好还是解决掉，不要给系统留下隐患。

## 2、解决方法

解决依赖冲突的方法，就是使用Maven提供的标签，标签需要放在标签内部，就像下面这样：

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.10.0</version>
  <exclusions>
    <exclusion>
      <artifactId>log4j-api</artifactId>
      <groupId>org.apache.logging.log4j</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

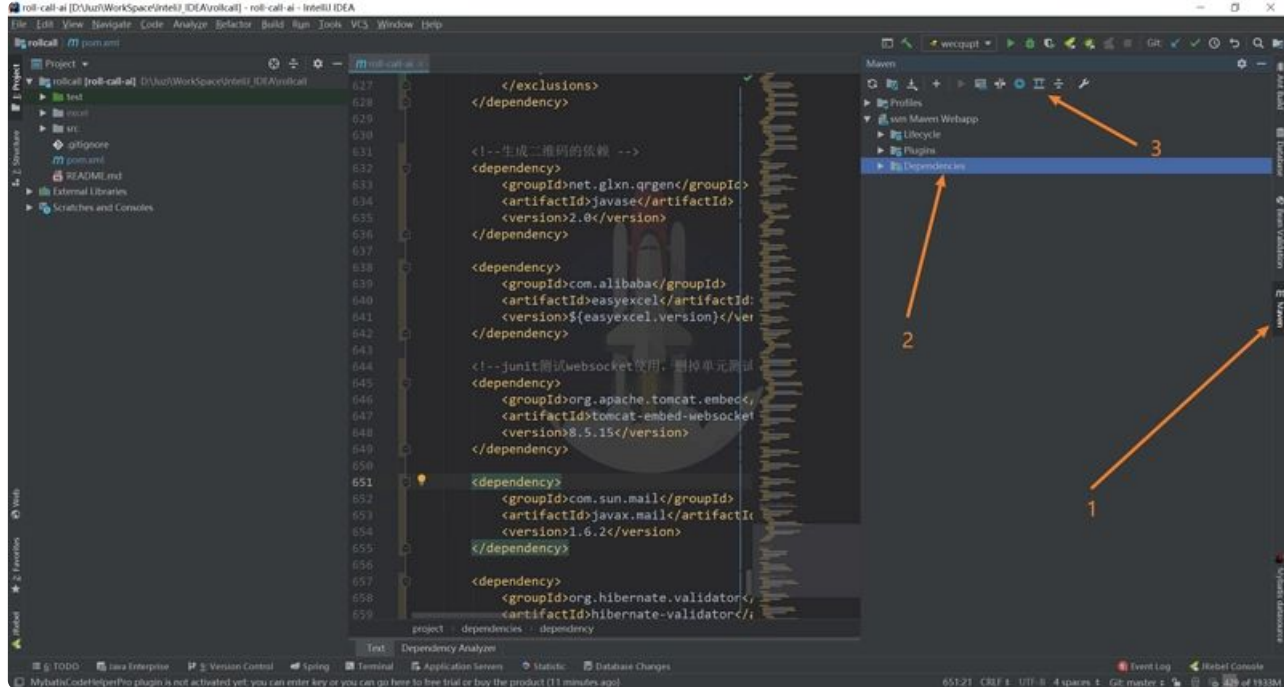
log4j-core 本身是依赖了 log4j-api 的，但是因为一些其他的模块也依赖了 log4j-api，并且两个 log4j-api 版本不同，所以我们使用标签排除掉 log4j-core 所依赖的 log4j-api，这样Maven就不会下载 log4j-core 所依赖的 log4j-api 了，也就保证了我们的项目中只有一个版本的 log4j-api。

## 3、Maven Helper

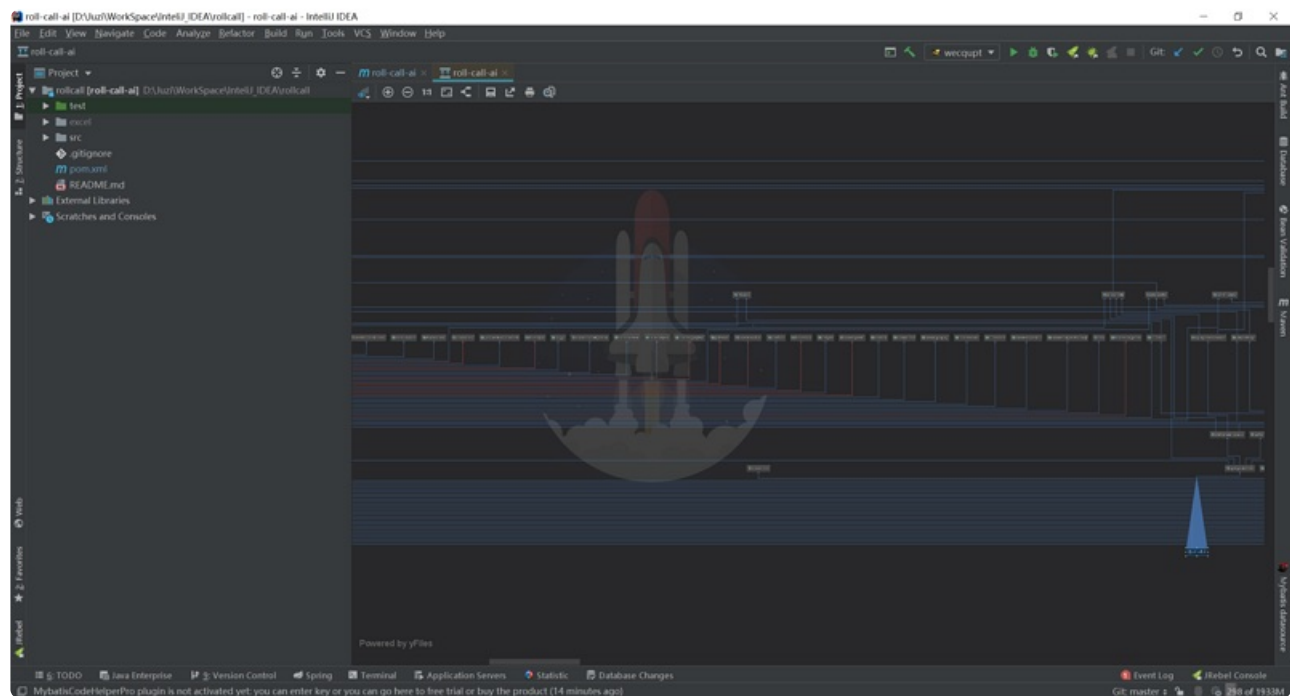
看到这里，你可能会有一个疑问。如何才能知道自己的项目中哪些依赖的Jar包冲突了呢？Maven Helper这个IntelliJ IDEA的插件帮我们解决了这个问题。插件的安装方法我就不讲了，既然你都会Maven了，我相信你也是会安装插件的。

在插件安装好之后，我们打开pom.xml文件，在底部会多出一个**Dependency Analyzer**选项





在图中，我们可以看到有一些红色的实线，这些红色实线就是依赖冲突，蓝色实线则是正常的依赖。




来源: <http://suo.im/6brHfY>

【END】

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



久一 



扫一扫上面的二维码图案，加我微信

### 推荐阅读

1. 我把废旧 Android 手机改造成了 Linux 服务器
2. 动画：一个浏览器是如何工作的？
3. 为什么你学不会递归？
4. 一个女生不主动联系你还有机会吗？
5. 团队开发中 Git 最佳实践



喜欢文章, 点个在看 

