

# IT楠老师maven教学资料

**B站：IT楠老师 公众号：IT楠说java QQ群：1083478826 新知大数据**

制作不易、如果觉的好不妨打个赏：



## 一、项目构建

**给你一套源代码，你怎么能跑起来？**

不能把！就是几个文件夹，几个文件。是不能运行的！

是不是需要和idea打交道，告诉idea怎么样才能运行起来，比如知道main方法在哪里？配置文件在哪里，编译好的文件输出到哪里，是不是？当然eclipse也一样。

**平时我们是怎么构建项目的，项目怎么运行起来呢？**

idea帮我们编译

我们依靠点击构建项目

一切设置好以后，使用工具（idea）帮我们打包

**项目构建中几个关键点？**

- 1、jdk啥版本
- 2、哪些文件夹是干啥的！！源文件？配置文件？测试文件？在哪里？
- 3、如果是web工程，web.xml放哪里？
- 4、编译文件，编译后的文件放在哪里。

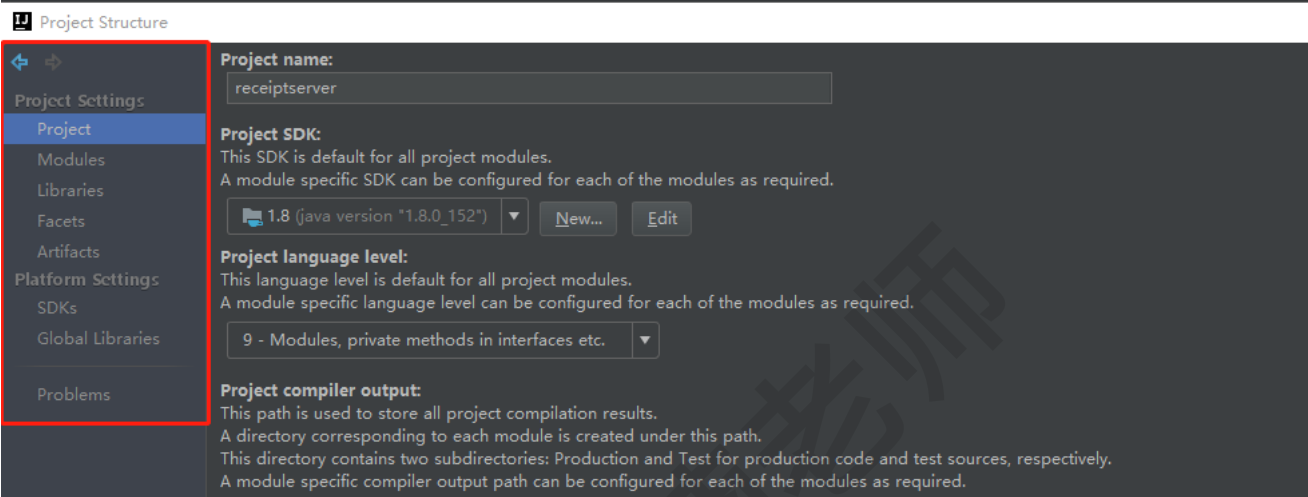
5、打包，打包成什么文件

1、先聊一聊idea的项目结构

好好说说Project Structure

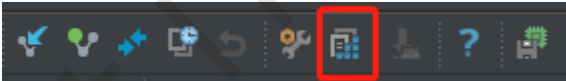
- 英 /'strʌktʃə(r)/

“项目结构”对话框允许您管理项目和IDE级别的元素，例如Modules，Facets，Libraries，Artifacts和SDK。



打开方式有两种

1、通过工具栏

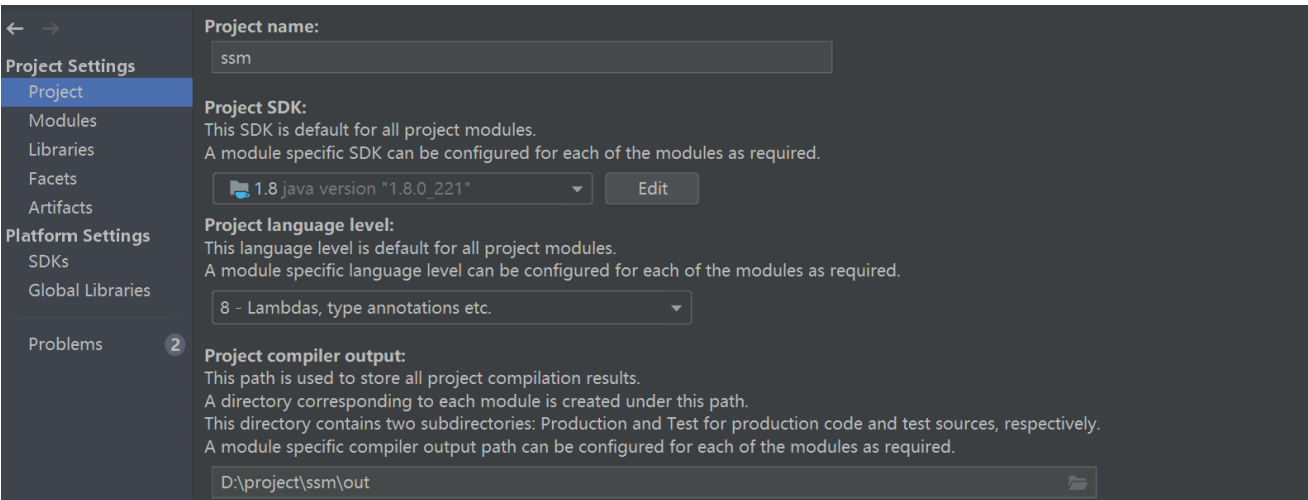


2、通过快捷键

Ctrl+Shift+Alt+S

(1) Project选项

指定项目名称，SDK，语言级别和编译器输出路径。

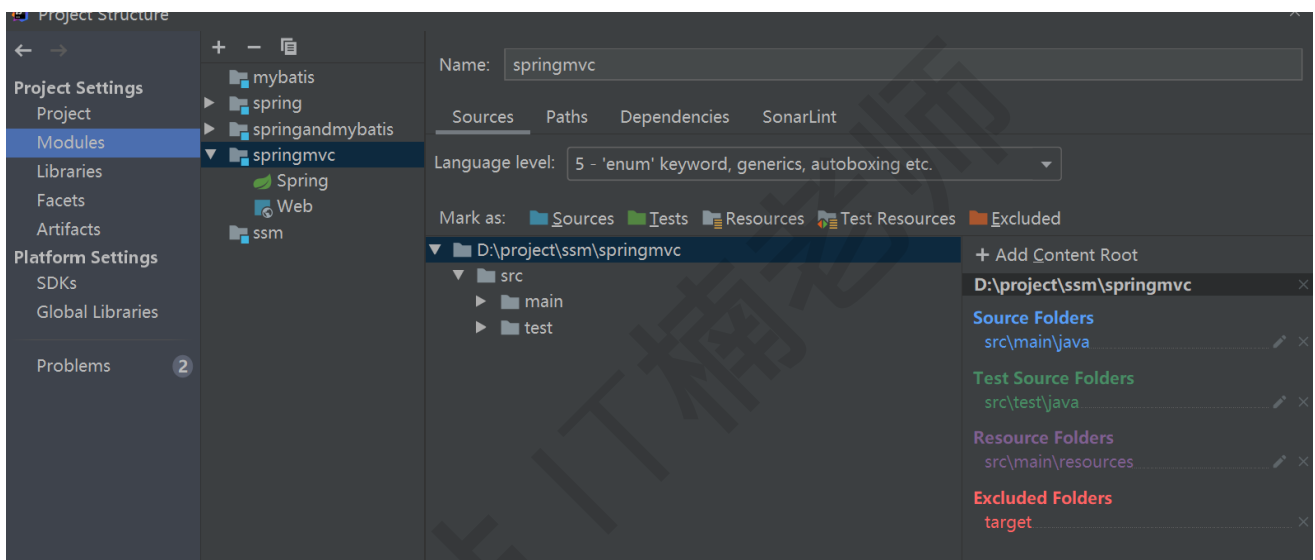


- **Project name**：项目名，使用此字段编辑项目名称。
- **Project SDK**：项目SDK，选择项目SDK。如果所需SDK不在列表中，请单击“New”，然后选择所需的SDK类型。然后，在打开的对话框中，选择SDK主目录，然后单击确定。要查看或编辑所选SDK的名称和内容，请单击“Edit”。（SDK页面将打开。）
- **Project language level**：选择要支持的Java语言级别。选定的级别将被用作项目默认值。
- **Project compiler output**：项目编译器输出，指定IntelliJ IDEA将存储编译结果的路径。单击选择路径对话框中browseButton 的目录。

指定目录中的两个子目录将被创建：**production** 为生产代码。**test** 为测试来源。在这些子目录中，将为每个模块创建单独的输出目录。输出路径可以在模块级重新定义。

## (2) Modules 选项

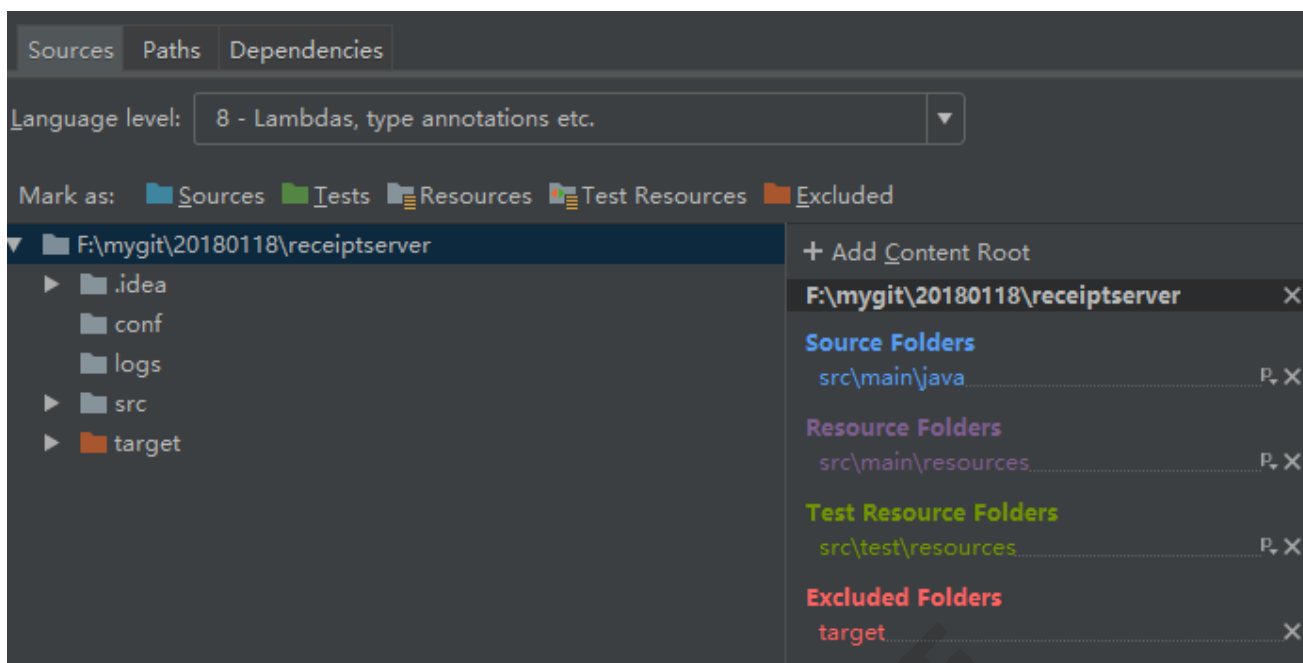
指定项目名称，SDK，语言级别和编译器输出路径。








- **Name**：项目名称
- **Sources**：这里对Module的开发目录进行文件夹分类，就是说这个module里有什么内容，说明了不同性质的内容放在哪里。注意，这些不同内容的标记代表了一个标准Java工程的各项内容，IntelliJ就是根据这些标记来识别一个Java工程的各项内容的，比如，它会用javac去编译标记为Sources的源码，打包的时候会把标记为Resources的资源拷贝到jar包中，并且忽略标记为Excluded的内容。左边显示的是在选中内容的预览。
- **Paths**：为模块配置编译器输出路径，还可以指定与模块关联的外部JavaDocs和外部注释的位置。
- **Dependencies**：在此选项卡上，您可以定义模块SDK并形成模块依赖关系列表。

## (3) Sources选项

对module的开发目录进行文件夹分类，以让idea明白怎么去对待他们，明确哪些是存放源代码的文件夹，哪些是存放静态文件的文件夹，哪些是存放测试代码的文件夹，哪些是被排除编译的文件夹。



Language level: 语言级别列表，使用此列表为模块选择Java语言级别。可用选项对应于JDK版本。

图标	命令	描述
	来源	使用此图标或命令将所选文件夹或文件夹分配给源文件夹类别。
	测试	使用此图标或命令将选定的一个或多个文件夹分配给测试源。
	资源	对于Java模块：使用此图标或命令将所选文件夹或文件夹分配给资源。
	测试资源	对于Java模块：使用此图标或命令将选定的一个或多个文件夹分配给测试资源。
	排除	使用此图标或命令可以排除选定的文件夹。

Sources: 源代码存放的文件，蓝色。

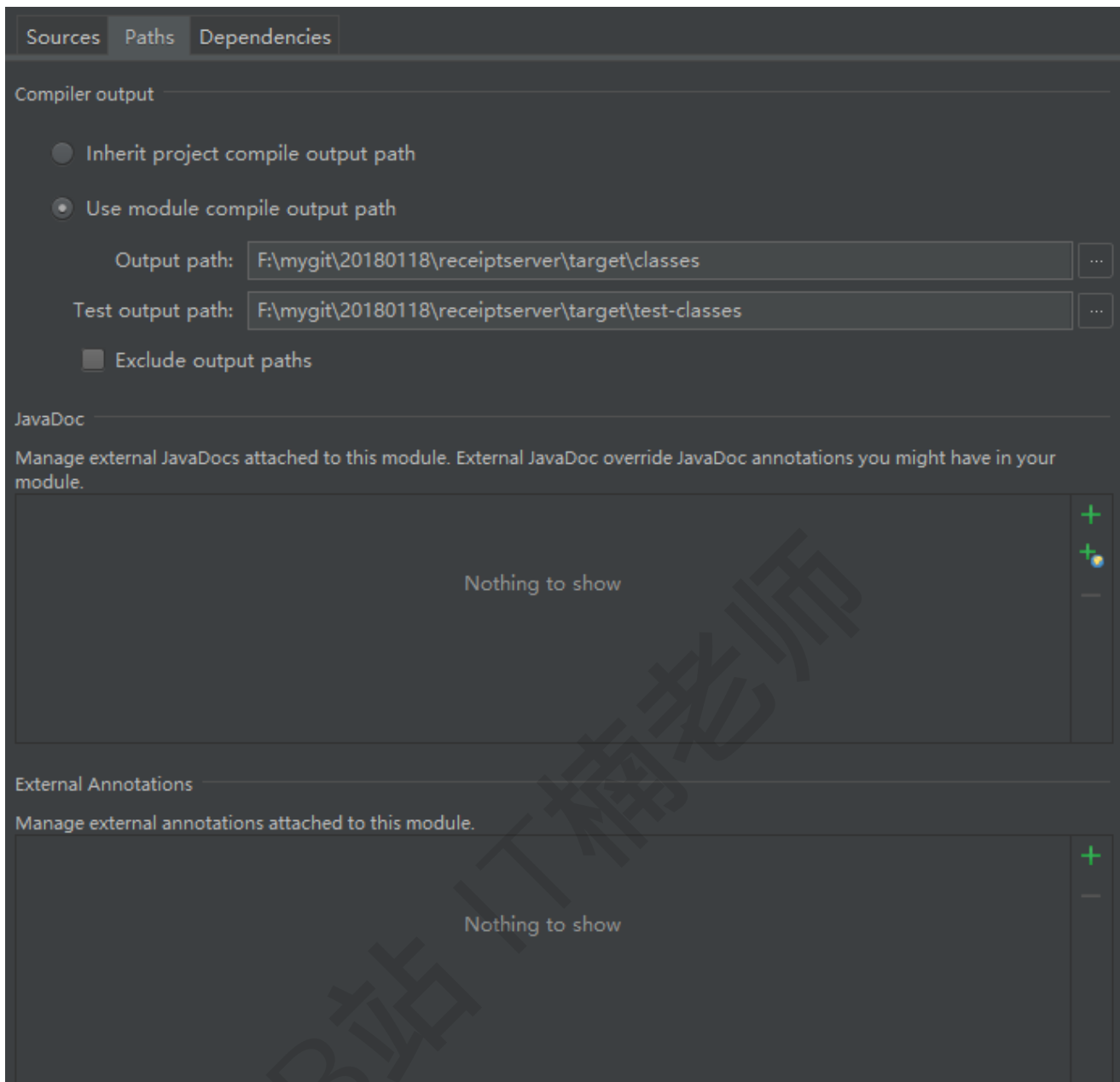
Tests: 设置测试代码存放的文件件，绿色。

Resources: 一般对应着Sources文件，一般放配置文件，如：db.properties。

Test Resources: 这个对应着Tests文件夹，存放着Tests代码的配置文件。

Excluded: 设置配出编译检查的文件，例如我们在project模块设置的out文件夹。

## (4) Paths



- Compiler output: 编译输出路径。
- Inherit project compile output path: 继承项目编译输出路径 选择此选项以使用为项目指定的路径。即上面在Project选项中设置的out文件路径。
- Use module compile output path: 使用模块编译输出路径。

Output path: 编译输出路径。

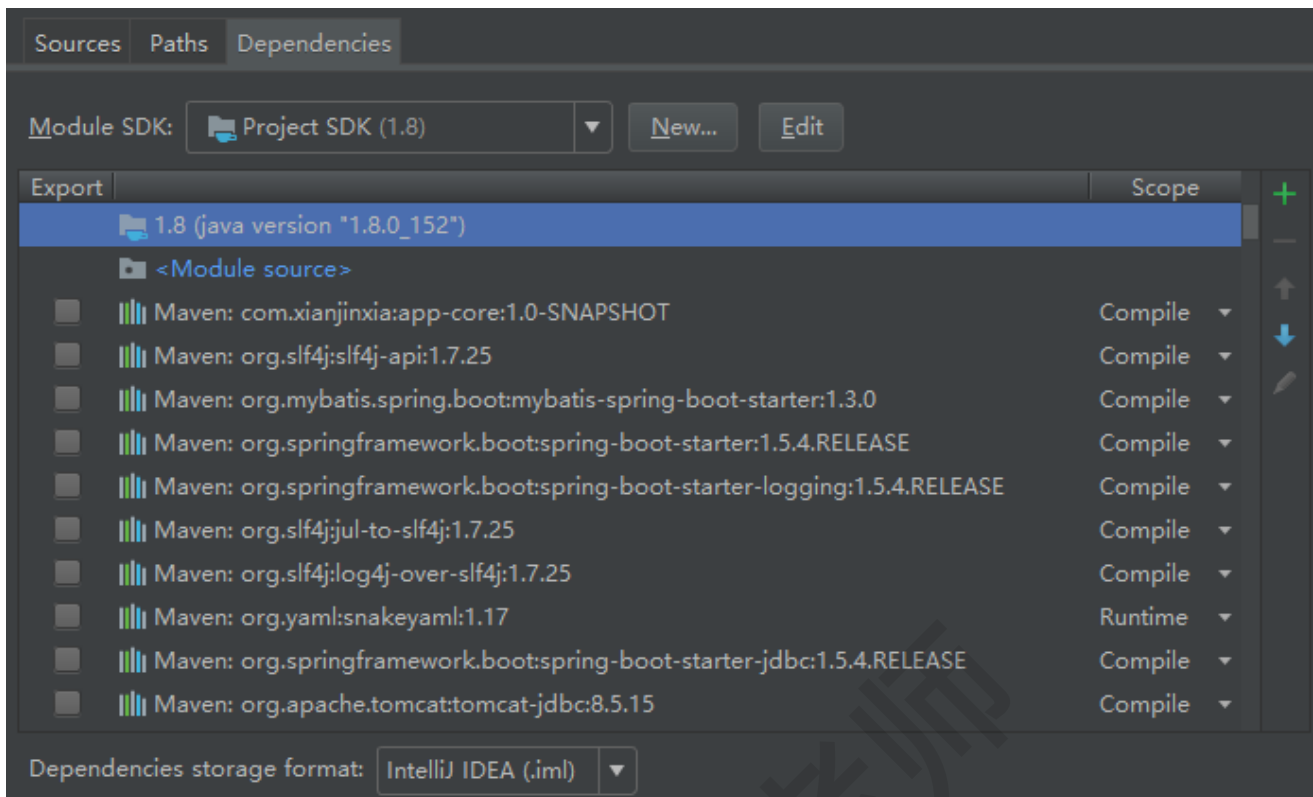
Test output path: 测试代码编译输出路径。

Exclude output paths: 排除输出路径，选中此复选框可以排除输出目录。

- JavaDoc: 使用可用控件组合与模块关联的外部JavaDocs存储位置的列表。
- External Annotations: 外部注释。使用+和-管理与模块关联的外部注释的位置（目录）列表。

## (5) Dependencies

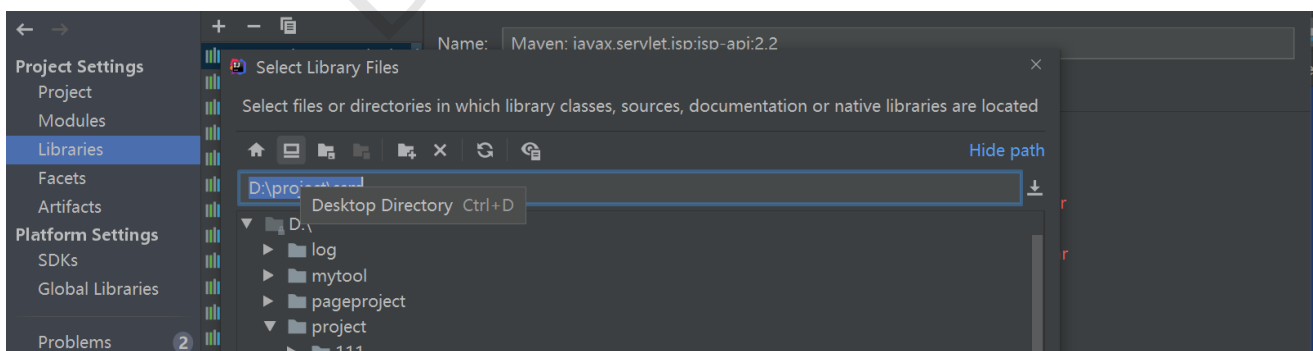
在此选项卡上，您可以定义模块SDK并形成模块依赖关系列表。



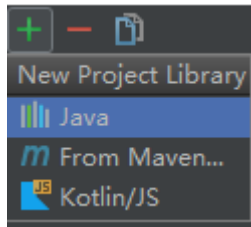
- Module SDK：模块SDK，选择模块SDK。  
(要将项目SDK与模块相关联，请选择Project SDK。请注意，如果稍后更改了项目SDK，模块SDK将相应更改。如果所需SDK不在列表中，请单击“新建”，然后选择所需的SDK类型。然后，在打开的对话框中，选择SDK主目录，然后单击确定。要查看或编辑所选SDK的名称和内容，请单击编辑。(SDK页面将打开。))
- 依赖列表
- 相关性存储格式，选择用于存储依赖关系的格式（作为IntelliJ IDEA模块或Eclipse项目）。该选项对使用不同开发工具的团队有帮助。

## (6) Libraries

在此选项卡上，您可以定义模块SDK并形成模块依赖关系列表。



首先，可以创建一个新的项目库，可以设置分类。



可以添加本地jar包，网络来源的jar包，排除jar包，删除jar包。



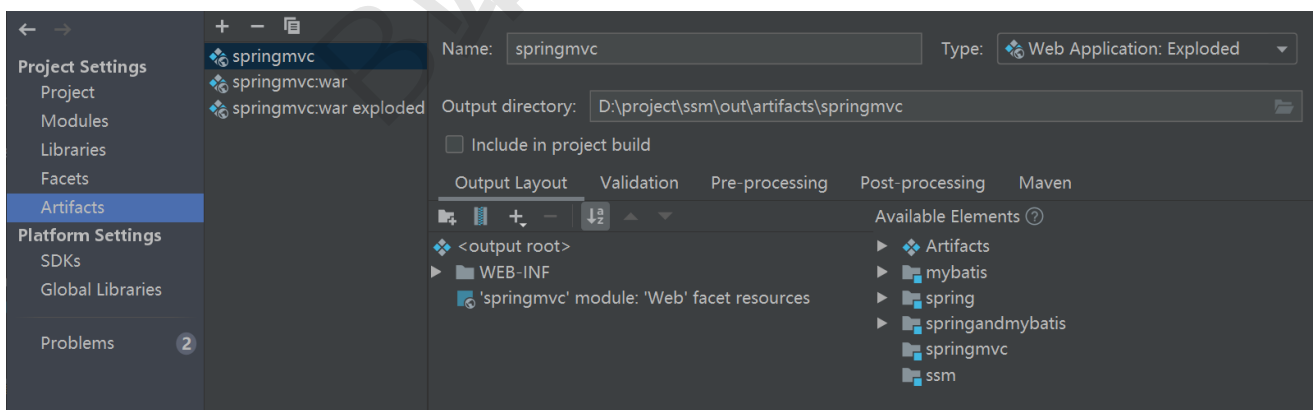
## (7) Facets

表示这个 module 有什么特征，比如 Web，Spring 和 Hibernate 等；



## (8) Artifacts

- **Artifact** 是 maven 中的一个概念，表示某个 module 要如何打包，例如 war exploded、war、jar 等等这种打包形式；一个 module 有了 Artifacts 就可以部署到应用服务器中了！
- 在给项目配置 Artifacts 的时候有好多个 type 的选项，exploded 是什么意思？explode 在这里你可以理解为展开，不压缩的意思。也就是 war、jar 等产出物没压缩前的目录结构。
- 建议在开发的时候使用这种模式，便于修改了文件的效果立刻显现出来。
- 默认情况下，IDEA 的 Modules 和 Artifacts 的 output 目录已经设置好了，不需要更改。
- 打成 war 包的时候会自动在 WEB-INF 目录下生产 classes 目录，然后把编译后的文件放进去。



## (9) SDKS

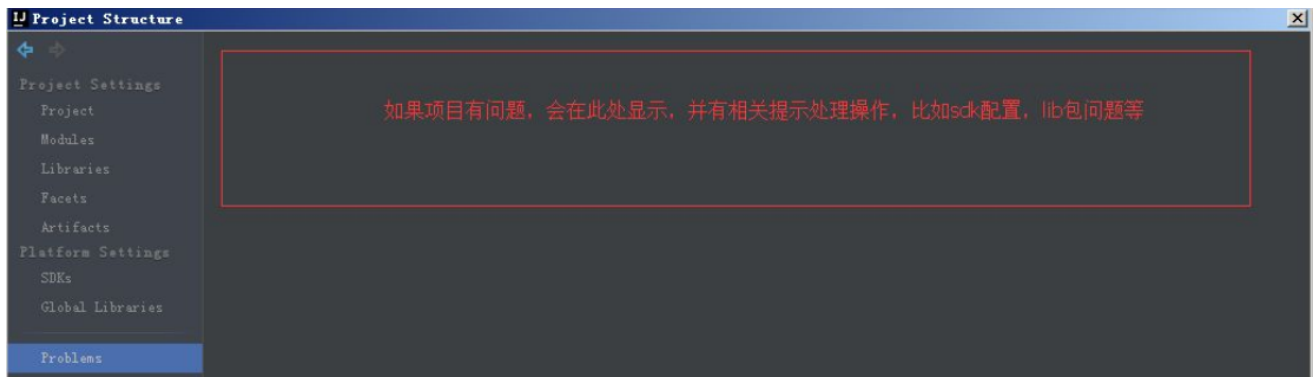
系统开发工具，全局 SDK 配置。

## (10) Global libraries

全局类库，可以配置一些常用的类库。

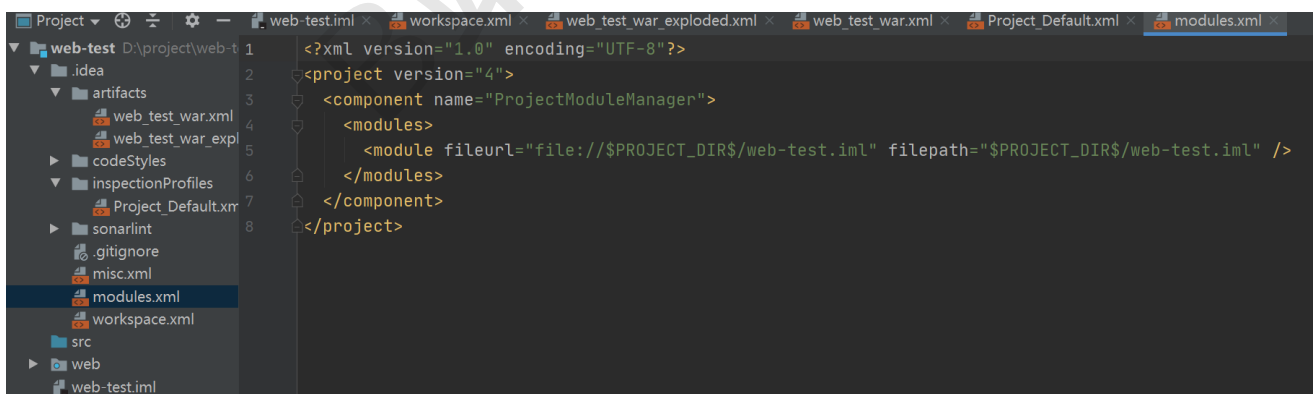
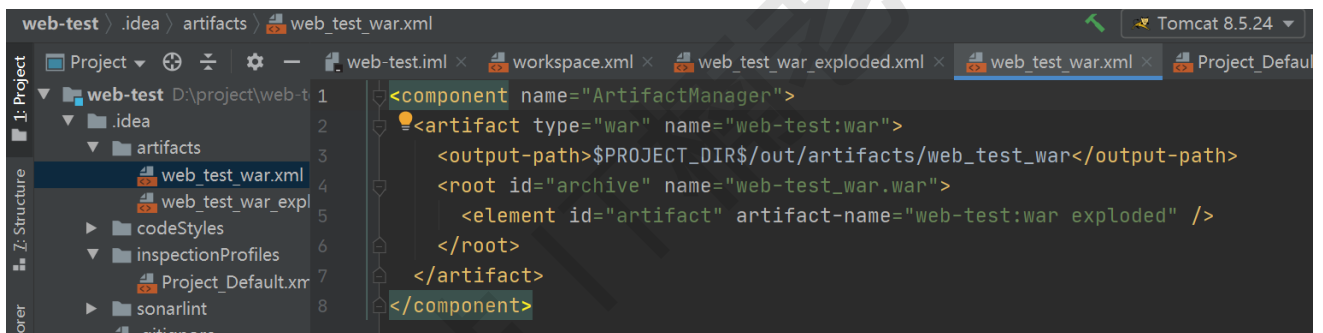
## (11) Problems

问题，在项目异常的时候很有用，可以根据提示进行项目修复（FIXED）。

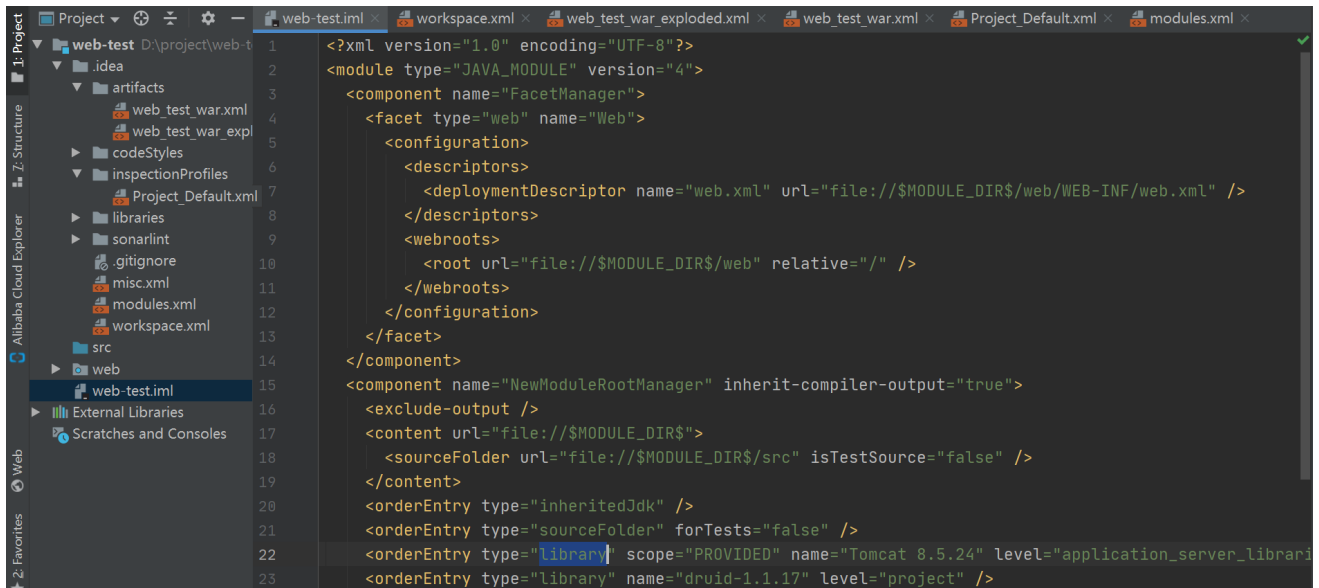


## (12) .idea和.iml

我们设置了半天看看我们设置的信息被保存在哪里？







这样别的idea打开项目时就能读取出来，并能明确项目的构建信息

问，eclipse直接打开能用吗？

## 二、Maven概述

以上的工作，需要我们自行构建，而且idea和eclipse，甚至一些其他的开发工具构建的时候是不一样的。

那么有没有一种统一的方式，甚至无需手动点击，通过配置就可以了。

当然我们看到了我们的idea其实也是通过之文件来记录构建信息的，那么既然构建如此重要，形成一套规范化的，统一的便捷的构建工具就势在必行，于是出现了 **maven**，当然还有 **gradle**，他们的功能异常强大。

这样有什么好处

- 统一管理jar包，自动导入jar及其依赖，这样是很初学者唯一能感受出来的好处，确实牛逼啊。
- 项目移植之后甚至不需要安装开发工具，只需要maven加命令就能跑起来，降低学习成本。
- 使我们的项目流水线成为可能，因为使用简单的命令我们就能完成项目的编译，打包，发布等工作，就让程序操作程序成为了可能，大名鼎鼎的jenkins技能做到这一点。

### 1、Maven下载安装

下载地址：<http://maven.apache.org/>

小知识点：作为一个java程序员 apache 网站的规律得知道都是 项目名.apache.org

- <http://maven.apache.org/>
- <http://tomcat.apache.org/>
- <http://dubbo.apache.org/>
- <http://hadoop.apache.org/>

**Apache软件基金会**（也就是Apache Software Foundation，简称为ASF），是专门为支持开源软件项目而办的一个非盈利性组织。在它所支持的Apache项目与子项目中，所发行的软件产品都遵循Apache许可证（Apache License）。

### 安装以及配置环境变量，学过点java的都会

1. 解压
2. 配置MAVEN\_HOME
3. 配置path, %MAVEN\_HOME%\bin
4. cmd执行 mvn -v，出现以下界面，成功

```
Microsoft Windows [版本 10.0.18363.836]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\zn>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: D:\mytool\apache-maven-3.6.3\bin\..
Java version: 1.8.0_221, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_221
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

## 2、maven核心全局配置文件

此电脑 > 本地磁盘 (D:) > mytool > apache-maven-3.6.3 > conf					搜索"conf"
名称	修改日期	类型	大小		
logging	2020/5/12 9:32	文件夹			
settings.xml	2020/5/12 9:38	XML 文档	11 KB		
toolchains.xml	2019/11/7 12:32	XML 文档	4 KB		

### 先照着配置

#### (1) 配置路径

```
<localRepository>D:/repository</localRepository>
```

#### (2) 配置阿里云镜像

要不啥也下不动

```

<mirrors>
  <mirror>
    <id>alimaven</id>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    <mirrorOf>central</mirrorOf>
  </mirror>
</mirrors>

```

### (3) 配置全局编译jdk版本

```

<profile>
  <id>jdk-1.8</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>1.8</jdk>
  </activation>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
  </properties>
</profile>

```

## 3、常常鲜，体验

### maven标准目录

```

src
|--main
|  |--java      源代码目录
|  |--resources  资源目录
|--test
|  |--java      测试代码目录
|  |--resources  测试资源目录
|--target
|  |--classes    编译后的class文件目录
|  |--test-classes 编译后的测试class文件目录
pom.xml          Maven工程配置文件

```

这是大部分Maven工程的目录结构，在这个基础上可以合理地增删目录。

pom.xml的基本要求：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <artifactId>test</artifactId>
  <groupId>org.xinzhi</groupId>
  <version>1.0-SNAPSHOT</version>

</project>
```

写个Hello.java

```
public class Hello{
    public static void main(String args[]){
        System.out.println("Hello maven!");
    }
}
```

可以再 `resources` 文件夹下新建 `db.properties` 配置文件;

执行

```
mvn compile
```

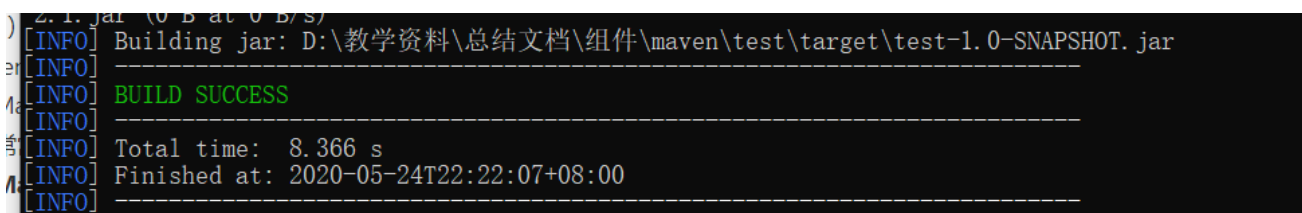
结果, 并生成target目录

```
D:\教学资料\总结文档\组件\maven\test>mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.xinzhi:test >-----
[INFO] Building test 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ test ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ test ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding GBK, i.e. build is platform dependent!
[INFO] Compiling 1 source file to D:\教学资料\总结文档\组件\maven\test\target\classes
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 1.159 s
[INFO] Finished at: 2020-05-24T22:14:40+08:00
[INFO]
[INFO] -----
```

结果发现我们编译的class文件和resources中的配置文件都放在了一起



执行 mvn package



观察target中已经有了我们打包好的jar包



观察名字是不是我们项目的名字加版本号，当然此jar包无法运行，应为maven也不知道你的main方法在哪里，咱们后边讲。

## 4、Maven生命周期

**maven生命周期其实就是描述了一个项目从源代码到部署的整个周期**

Maven有三个内置的生命周期：**默认 (default)**，**清洁 (clean)** 和**站点 (site)**。

- 清洁 (clean) 为执行以下工作做必要的清理。就是我们经常做的，删除out文件夹。
- 默认 (default) 真正进行项目编译打包等工作的阶段
- 站点 (site) 生成项目报告，站点，发布站点

**默认 (default) 的生命周期包括以下阶段 (该阶段经过简化，实际上更加复杂)：**

1. 验证 (validate) - 验证项目是否正确，所有必要的信息可用。
2. 编译 (compile) - 编译项目的源代码。
3. 测试 (test) - 使用合适的单元测试框架测试编译的源代码。这些测试不应该要求代码被打包或部署。
4. 打包 (package) - 采用编译的代码，并以其可分配格式 (如JAR) 进行打包。
5. 验证 (verify) - 对集成测试的结果执行任何检查，以确保满足质量标准。
6. 安装 (install) - 将软件包安装到本地存储库中，用作本地其他项目的依赖项。
7. 部署 (deploy) - 在构建环境中完成，将最终的包复制到远程存储库以与其他开发人员和项目共享。

```
mvn install
```

此命令在执行安装之前按顺序（验证（validate），编译（compile），打包（package）等）执行每个默认生命周期阶段。在这种情况下，您只需要调用最后一个构建阶段来执行，安装（install）。

在构建环境中，使用以下调用将工件清理地构建并部署到共享存储库中。

```
mvn clean deploy
```

相同的命令可以在多模块场景（即具有一个或多个子项目的项目）中使用。Maven遍历每个子项目并执行清洁（clean），然后执行部署（deploy）（包括所有之前的构建阶段步骤）。

注意：在我们开发阶段，有一些生命周期的阶段，比如验证（validate）这些，基本很少用到。只要使用关键的几个基本能满足需求。

## 5、Maven 常用命令

下面maven比较常见的一些命令。

命令	说明
mvn -version	显示版本信息
mvn clean	清理项目生产的临时文件,一般是模块下的target目录
mvn compile	编译源代码，一般编译模块下的src/main/java目录
mvn package	项目打包工具,会在模块下的target目录生成jar或war等文件
mvn test	测试命令,或执行src/test/java/下junit的测试用例
mvn install	将打包的jar/war文件复制到你的本地仓库中,供其他模块使用
mvn deploy	将打包的文件发布到远程参考,提供其他人员进行下载依赖
mvn site	生成项目相关信息的网站
mvn dependency:tree	打印出项目的整个依赖树
mvn archetype:generate	创建Maven的普通java项目
mvn tomcat:run	在tomcat容器中运行web应用

## 6、Maven的版本规范（我们的项目）

所有的软件都用版本

Maven使用如下几个要素来定位一个项目，因此它们又称为项目的坐标。

- **groupId** 团体、组织的标识符。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开头。一般对应着JAVA的包的结构，例如org.apache。
- **artifactId** 单独项目的唯一标识符。比如我们的tomcat, commons等。不要在artifactId中包含点号(.)。
- **version** 项目的版本。
- **packaging** 项目的类型，默认是jar，描述了项目打包后的输出。类型为jar的项目产生一个JAR文件，类型为war的项目产生一个web应用。

Maven在版本管理时候可以使用几个特殊的字符串 SNAPSHOT, LATEST, RELEASE。比如"1.0-SNAPSHOT"。各个部分的含义和处理逻辑如下说明：

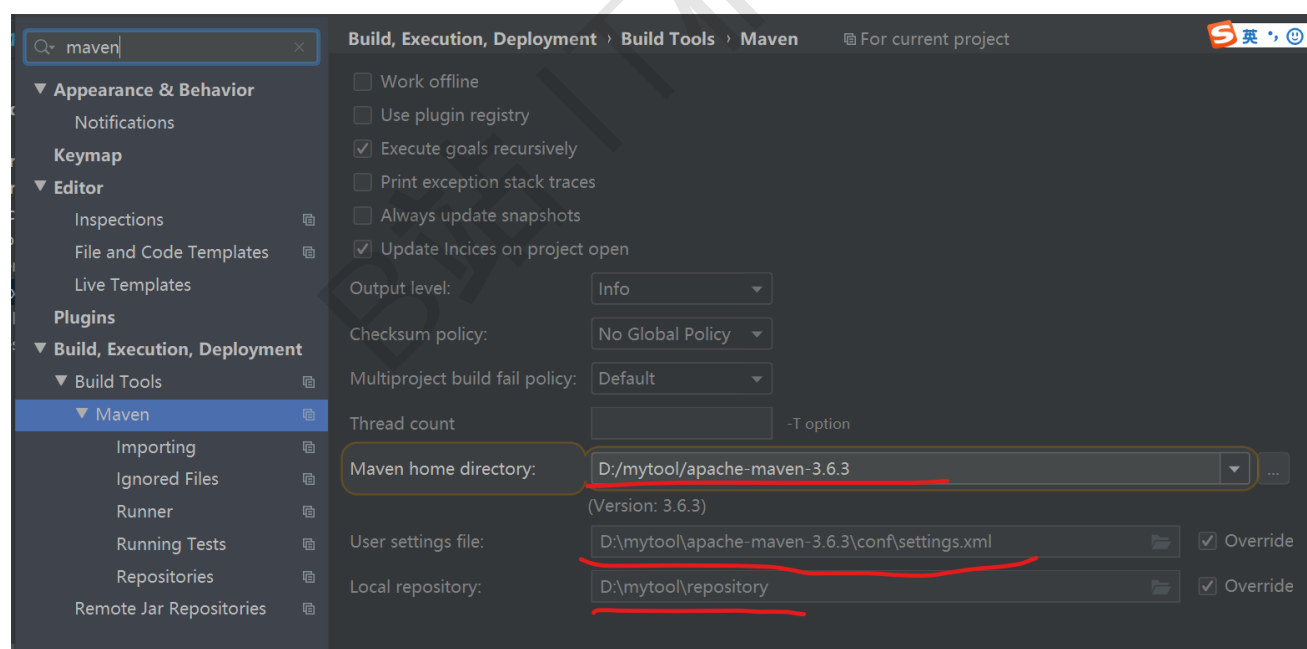
- **SNAPSHOT** 这个版本一般用于开发过程中，表示不稳定的版本。
- **LATEST** 指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个snapshot版，具体看哪个时间最后。
- **RELEASE** 指最后一个发布版。

## 7、在idea中配置maven

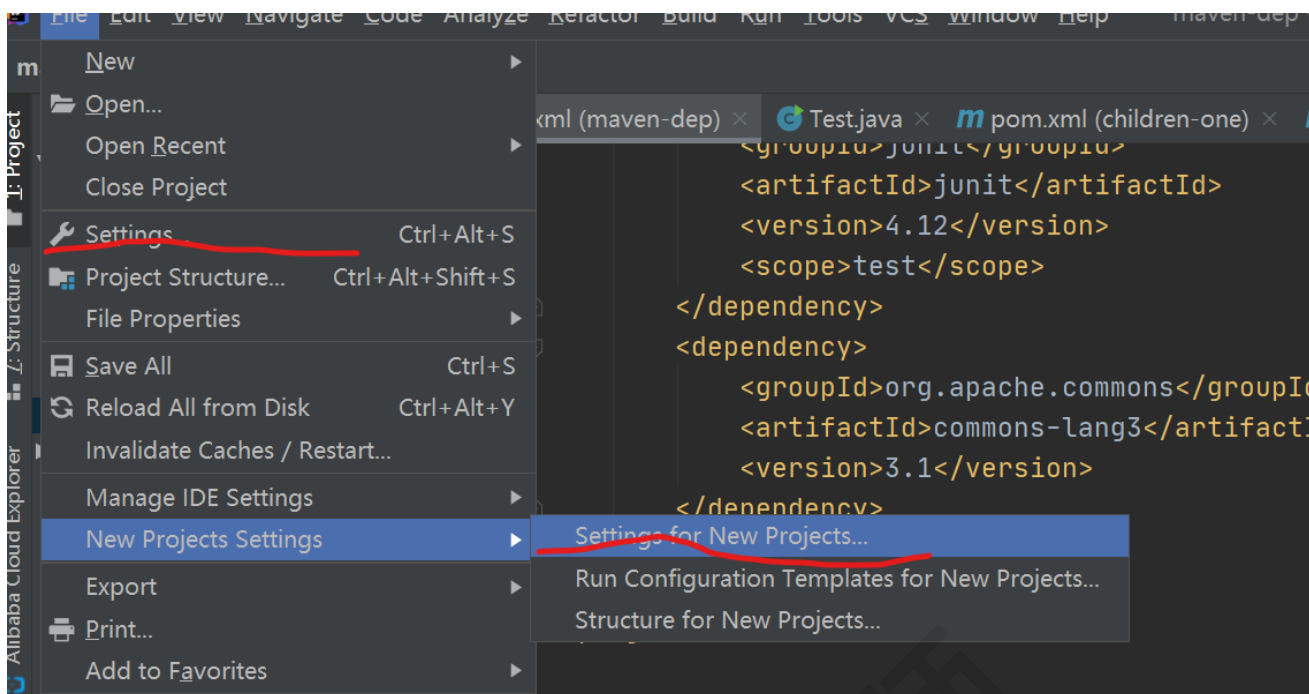
把画红线的东西全部配置成自己的。最后一个仓库，在你的其他盘找一个地方新建repository文件夹，自己要知道，选中，如果勾选不了就选择都选override。

如果不选择仓库会把jar包下载至C盘的下边目录，不好维护，还占用c盘空间。

C:\Users\zn\.m2\repository



两处都要配置，一个是当前项目的maven配置，一个是新建项目的maven配置。



### 三、Maven依赖（重点）

**maven管理依赖也就是jar包牛逼之处是不用我们自己下载，会从一些地方自动下载**

- maven远程仓库: <https://mvnrepository.com/>
- maven远程仓库: <https://maven.aliyun.com/mvn/search>

**maven工程中我们依靠在pom.xml文件进行配置完成jar包管理工作（依赖）**

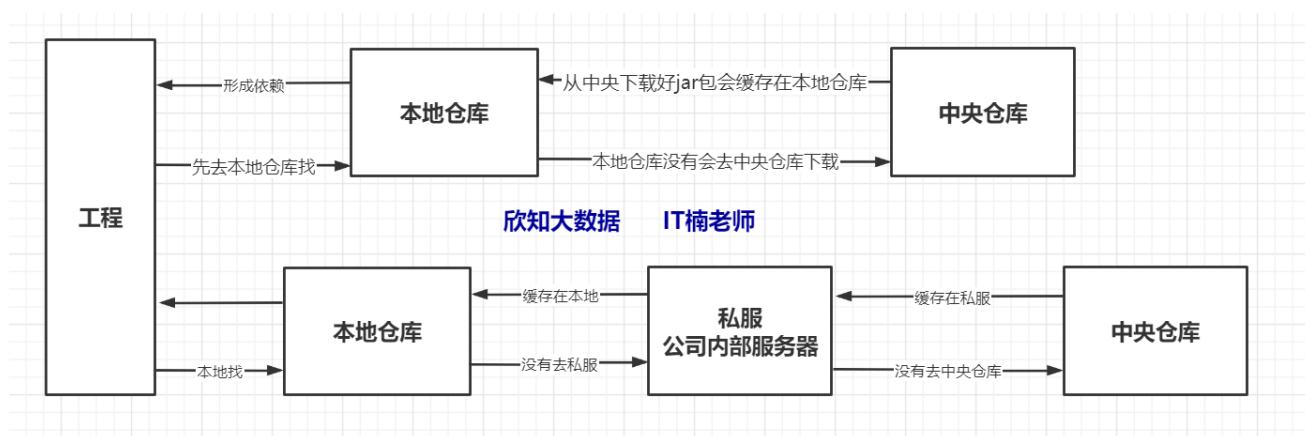
在工程中引入某个jar包，只需要在 pom.xml 中引入jar包的坐标，比如引入log4j的依赖：

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.7</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Maven 通过 groupId、artifactId 与 version 三个向量来定位Maven仓库其jar包所在的位置，并把对应的jar包引入到工程中来。

**jar包会自动下载，流程如下**





## 1、依赖范围

### 了解、classpath是个什么东西

顾名思义，就是编译好的 `class` 文件所在的路径。

事实上，我们的类加载器（`classloader`）就是去对应的 `classpath` 中加载 `class` 二进制文件。

### 普通java项目

名称	压缩后大小	原始大小	类型
test.jar			
com			
xinshi			
entity			
META-INF			
hello.xml	8	6	XML 文档

META-INF中有个文件，有以下内容，告诉jvm执行的时候去哪个类里找main方法。

普通的java工程类路径就是最外层的目录。

```

Manifest-Version: 1.0
Main-Class: com.xinshi.HelloUser

```

### web项目

咱们的src打包后会放在

名称	压缩后大小	原始大小	类型
web-test_war.war			
WEB-INF			
classes			
com			
xinshi			
web.xml	167	304	XML 文档

web-test_war.war	名称	压缩后大小	原始大小	类型
WEB-INF	WEB-INF			
index.jsp	index.jsp	219	299	JSP 文件

src目录下的配置文件会和class文件一样，自动copy到应用的 WEB-INF/classes目录下，所以普通jar包的类路径就是根路径，没有资源，如果有配置文件也放在src目录下，他会同步打包在类路径下。

所以web项目的classpath是 WEB-INF/classes

## maven项目

maven工程会将 src/main/java 和 src/main/resources 文件夹下的文件全部打包在classpath中。运行时他们两个的文件夹下的文件会被放在一个文件夹下。

maven 项目不同的阶段引入到classpath中的依赖是不同的，例如，

- 编译时，maven 会将与编译相关的依赖引入classpath中
- 测试时，maven会将测试相关的的依赖引入到classpath中
- 运行时，maven会将与运行相关的依赖引入classpath中

而依赖范围就是用来控制依赖于这三种classpath的关系。

## scope标签就是依赖范围的配置

该项默认配置compile,可选配置还有test、provided、runtime、system、import。

其中compile、test和provided使用较多，下面依次介绍。

**有些jar包（如servlet-api）运行时其实是不需要的，因为tomcat里有，但编译时是需要的，因为编译的时候没有tomcat环境**

**有些jar只在测试的时候才能用到。比如junit，真是运行不需要的**

**有些jar运行，测试时必须要有，编译时不需要，如jdbc驱动，编译时用的都是jdk中的接口，运行时我们才使用反射注册了驱动。**

向以上的这些jar包不是说使用默认的compile一定不行，但是设置成合适的范围更好，当然有事会有问题，比如你引入的servlet-api和tomcat自带的的不同，就会出问题。

## 1.1 编译依赖范围 (compile)

**该范围就是默认依赖范围**，此依赖范围对于编译、测试、运行三种 classpath 都有效，举个简单的例子，假如项目中有 fastjson 的依赖，那么 fastjson 不管是在编译，测试，还是运行都会被用到，因此 fastjson 必须是编译范围（构件默认的是编译范围，所以依赖范围是编译范围的无须显示指定）

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.68</version>
</dependency>
```

## 1.2 测试依赖范围(test)

使用此依赖范围的依赖，只对测试classpath有效，在编译主代码和项目运行时，都将无法使用该依赖，最典型的例子就是 Junit，构件在测试时才需要，所以它的依赖范围是测试，因此它的依赖范围需要显示指定为、  
<scope>test</scope>，当然不显示指定依赖范围也不会报错，但是该依赖会被加入到编译和运行的classpath中，造成不必要的浪费。

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.7</version>
  <scope>test</scope>
</dependency>
```

## 1.3 已提供依赖范围(provided)

使用该依赖范围的maven依赖，只对编译和测试的classpath有效，对运行的classpath无效，典型的例子就是 servlet-api，编译和测试该项目的时候需要该依赖，但是在运行时，web容器已经提供的该依赖，所以运行时就不再需要此依赖，如果不显示指定该依赖范围，并且容器依赖的版本和maven依赖的版本不一致的话，可能会引起版本冲突，造成不良影响。

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
  <scope>provided</scope>
</dependency>
```

## 1.4 运行时依赖范围(runtime)

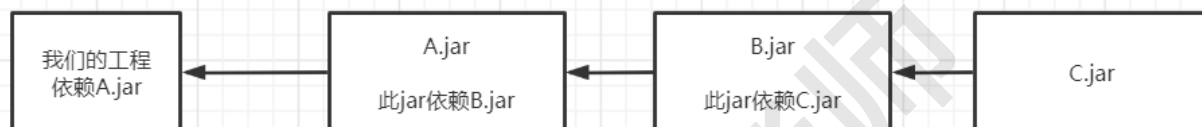
使用该依赖范围的maven依赖，只对测试和运行的classpath有效，对编译的classpath无效，典型例子就是JDBC的驱动实现，项目主代码编译的时候只需要JDK提供的JDBC接口，只有在测试和运行的时候才需要实现上述接口的具体JDBC驱动。

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.25</version>
  <scope>runtime</scope>
</dependency>
```

其他的范围不常用，有兴趣自己研究

## 2、依赖的传递

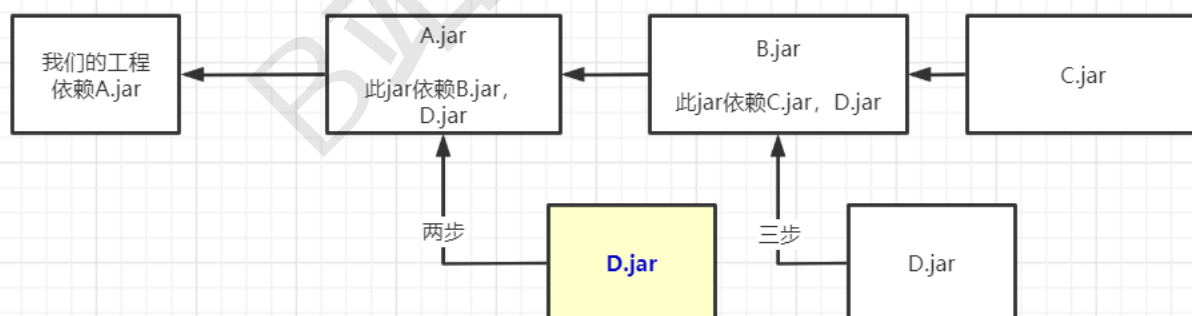
jar其实也是别人写的工程，他也会依赖其他的jar包，传递性让我们可以不用关系我们所依赖的jar他依赖了哪些jar，只要我们添加了依赖，他会自动将他所依赖的jar统统依赖进来。



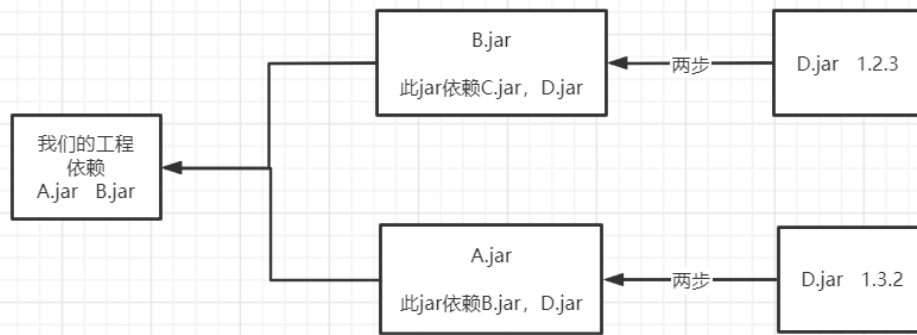
我们只需依赖A.jar，其他的会自动传递进来。

### 依赖传递的原则：

- **最短路径优先原则：**如果A依赖于B，B依赖于C，在B和C 中同时有log4j的依赖，并且这两个版本不一致，那么A会根据最短路径原则，在A中会传递过来B的log4j版本。



- **路径相同先声明原则：**如果我们的工程同时依赖于B和A，B和C没有依赖关系，并且都有D的依赖，且版本不一致，那么会引入在pom.xml中先声明依赖的log4j版本。



```
<dependency>
  <groupId>com.xinzi</groupId>
  <artifactId>B</artifactId>
  <version>1.5.3</version>
</dependency>
<dependency>
  <groupId>com.xinzhi</groupId>
  <artifactId>A</artifactId>
  <version>1.12.2</version>
</dependency>
```

因为1.2.3先声明，所以获胜。

#### 特别注意：

不同版本的jar选一个会导致一个问题，1.3.2版本高，A.jar可能用到了高版本的一些新的方法，此时因为某些原因系统选择了低版本，就会导致A.jar报错，无法运行。那么就要想办法把低版本排除掉，一般高版本会兼容低版本。

### 3、依赖的排除

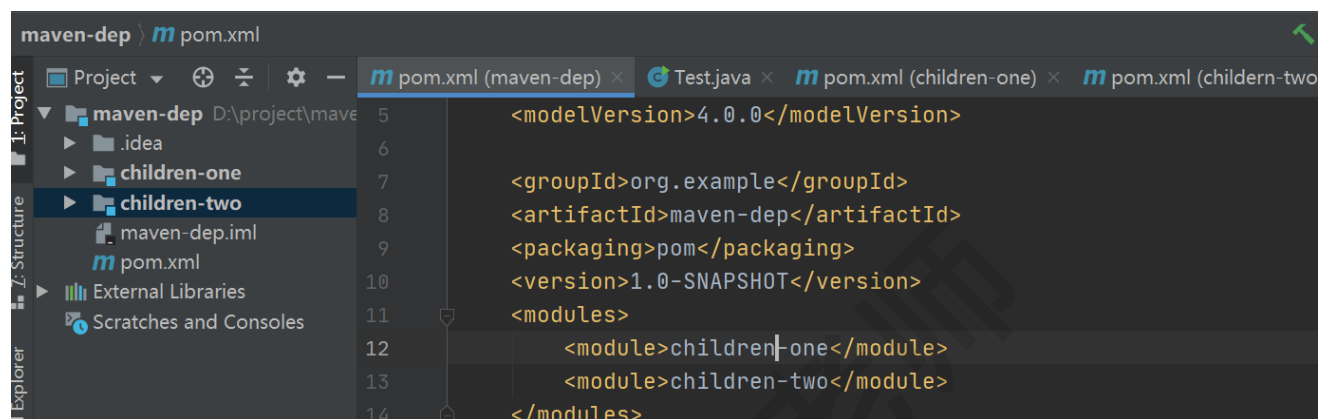
结合上个例子，我们想把低版本的D.jar排除了，就可以这样做，这样系统就只能依赖高版本

```
<dependencies>
  <dependency>
    <groupId>com.xinzi</groupId>
    <artifactId>B</artifactId>
    <version>1.5.3</version>
    <exclusions>
      <exclusion>
        <artifactId>com.xinzhi</artifactId>
        <groupId>D</groupId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>com.xinzhi</groupId>
```

```
<artifactId>A</artifactId>
<version>1.12.2</version>
</dependency>
</dependencies>
```

## 4、聚合和继承

### 分布式开发必须要用



### 聚合模块（父模块）的打包方式必须为pom，否则无法完成构建。

在聚合多个项目时，如果这些被聚合的项目中需要引入相同的Jar，那么可以将这些Jar写入父pom中，各个子项目继承该pom即可。父模块的打包方式必须为pom，否则无法构建项目。

### 通过在各个子模块中配置来表明其继承与哪一个父模块：

```
<parent>
  <artifactId>parent</artifactId>
  <groupId>org.example</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>

<artifactId>children-two</artifactId>
```

可以被继承的POM元素如下：

- **groupId**：项目组ID，项目坐标的核心元素
- **version**：项目版本，项目坐标的核心因素
- **properties**：自定义的Maven属性 一般用于同一制定各个依赖的版本号
- **dependencies**：项目的依赖配置 公共的依赖
- **dependencyManagement**：项目的依赖管理配置
- **repositories**：项目的仓库配置
- **build**：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等

### 一些对项目的描述

- description: 项目的描述信息
- organization: 项目的组织信息
- inceptionYear: 项目的创始年份
- url: 项目的URL地址
- developers: 项目的开发者信息
- contributors: 项目的贡献者信息
- distributionManagement: 项目的部署配置
- issueManagement: 项目的缺陷跟踪系统信息
- ciManagement: 项目的持续集成系统信息
- scm: 项目的版本控制系统
- mailingLists: 项目的邮件列表信息
- reporting: 包括项目的报告输出目录配置、报告插件配置等

## 四、POM文件

### 1、基础配置

一个典型的pom.xml文件配置如下:

```
<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/4.0.0http://maven.apache.org/xsd/maven
-4.0.0.xsd">

    <!-- 模型版本。maven2.0必须是这样写，现在是maven2唯一支持的版本 -->
    <modelVersion>4.0.0</modelVersion>
    <!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如com.xinzhi，maven会将该项目打
成的jar包放本地路径: /com/xinzhi/ -->
    <groupId>com.xinzhi</groupId>
    <!-- 本项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->
    <artifactId>test</artifactId>
    <!-- 本项目目前所处的版本号 -->
    <version>1.0.0-SNAPSHOT</version>

    <!-- 打包的机制，如pom,jar， war，默认为jar -->
    <packaging>jar</packaging>

    <!-- 为pom定义一些常量，在pom中的其它地方可以直接引用 使用方式 如下 : ${file.encoding} -->
    <!-- 常常用来整体控制一些依赖的版本号 -->
    <properties>
        <file.encoding>UTF-8</file.encoding>
        <java.source.version>1.8</java.source.version>
        <java.target.version>1.8</java.target.version>
    </properties>

    <!-- 定义本项目的依赖关系，就是依赖的jar包 -->
```

```

<dependencies>
  <!-- 每个dependency都对应这一个jar包 -->
  <dependency>
    <!--一般情况下，maven是通过groupId、artifactId、version这三个元素值（俗称坐标）来检索该构件，然后引入你的工程。如果别人想引用你现在开发的这个项目（前提是已开发完毕并发布到了远程仓库），-->
    <!--就需要在他的pom文件中新建一个dependency节点，将本项目的groupId、artifactId、version写入，maven就会把你上传的jar包下载到他的本地 -->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>

    <!-- 依赖范围 -->
    <scope>compile</scope>
    <!-- 设置 依赖是否可选，默认为false,即子项目默认都继承。如果为true，则子项目必需显示的引入 -->
    <optional>false</optional>

    <!-- 依赖排除-->
    <exclusions>
      <exclusion>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>

...
</project>

```

一般来说，上面的几个配置项对任何项目都是必不可少的，定义了项目的基本属性。

## 除了dependencies我们还用到了dependencyManagement，区别如下

### dependencies

- 即使在子项目中不写该依赖项，那么子项目仍然会从父项目中继承该依赖项（全部继承）。
- 继承下来就会被编译，如果子项目根本不用这个依赖会增加子工程的负担。

**dependencyManagement**：通常会在父工程中定义，目的是统一各个子模块的依赖版本，有不用实际依赖

- 只是声明依赖，并不实现引入
- 子项目**需要显示的声明需要用的依赖**。如果不在子项目中声明依赖，是不会从父项目中继承下来的；
- 只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且version和scope都读取自父pom;另外如果子项目中指定了版本号，那么会使用子项目中指定的jar版本。

## 2、构建配置

```
<build>
```



```

<!-- 产生的构件的文件名，默认值是${artifactId}-${version}。 -->
<finalName>myPorjectName</finalName>
<!-- 构建产生的所有文件存放的目录，默认为${basedir}/target，即项目根目录下的target -->
<directory>${basedir}/target</directory>
<!--项目相关的所有资源路径列表，例如和项目相关的配置文件、属性文件，这些资源被包含在最终的打包文件里。 -->
    <!--项目源码目录，当构建项目的时候，构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。 -->
    <sourceDirectory>${basedir}\src\main\java</sourceDirectory>

    <!--项目单元测试使用的源码目录，当测试项目的时候，构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。 -->
    <testSourceDirectory>${basedir}\src\test\java</testSourceDirectory>

    <!--被编译过的应用程序class文件存放的目录。 -->
    <outputDirectory>${basedir}\target\classes</outputDirectory>

    <!--被编译过的测试class文件存放的目录。 -->
    <testOutputDirectory>${basedir}\target\test-classes
</testOutputDirectory>
<!-- 以上配置都有默认值，就是约定好了目录就这么建 -->

<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>>false</filtering>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.properties</include>
            <include>**/*.xml</include>
        </includes>
        <filtering>>false</filtering>
    </resource>
</resources>

<!--单元测试相关的所有资源路径，配制方法与resources类似 -->
<testResources>
    <testResource>
        <targetPath />
        <filtering />
        <directory />
        <includes />
        <excludes />
    </testResource>
</testResources>

```

```

<!--使用的插件列表 。 -->
<plugins>
  <plugin>
    ...具体在插件使用中了解
  </plugin>
</plugins>

<!--主要定义插件的共同元素、扩展元素集合，类似于dependencyManagement， -->
<!--所有继承于此项目的子项目都能使用。该插件配置项直到被引用时才会被解析或绑定到生命周期。 -->
<!--给定插件的任何本地配置都会覆盖这里的配置 -->
<pluginManagement>
  <plugins>...</plugins>
</pluginManagement>

</build>

```

我们常用的几个配置

### 关于资源处理的配置

### 有些小伙伴就喜欢在src中填写配置文件

```

<!-- 处理资源被过滤问题 -->
<build>
  <resources>
    <resource>
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>>false</filtering>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>**/*.properties</include>
        <include>**/*.xml</include>
      </includes>
      <filtering>>false</filtering>
    </resource>
  </resources>
</build>

```

### 添加本地jar包

本地jar，如：支付宝jar包放到 src/main/webapp/WEB-INF/lib 文件夹下，如果没有配置，本地没问题，但是线上会找不到sdk类，为什么要引入，因为支付宝jar包再中央仓库没有

```

<!-- geelynote maven的核心插件之-compiler插件默认只支持编译Java 1.4，因此需要加上支持高版本jre的配置，在pom.xml里面加上 增加编译插 -->
<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>
    <compilerArguments>
      <!-- 本地jar，支付宝jar包放到 src/main/webapp/WEB-INF/lib 文件夹下，
      如果没有配置，本地没问题，但是线上会找不到sdk类
      为什么要引入，因为支付宝jar包再中央仓库没有，再比如oracle连接驱动的jar
      -->
      <extdirs>${project.basedir}/src/main/webapp/WEB-INF/lib</extdirs>
    </compilerArguments>
  </configuration>
</plugin>

```

### 3、仓库配置

```

<repositories>
  <repository>
    <id>alimaven</id>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>

```

pom.xml里面的仓库与setting.xml里的仓库功能是一样的。主要的区别在于，pom里的仓库是个性化的。比如一家大公司里的setting文件是公用的，所有项目都用一个setting文件，但各个子项目却会引用不同的第三方库，所以就需要在pom.xml里设置自己需要的仓库地址。

### 4、项目信息配置（知道）

```

<!--项目的名称，Maven产生的文档用 -->
<name>banseon-maven </name>

<!--项目主页的URL，Maven产生的文档用 -->
<url>http://www.c1f.com/ </url>

```

```
<!--项目的详细描述，Maven 产生的文档用。 当这个元素能够用HTML格式描述时 -->
<!-- (例如，CDATA中的文本会被解析器忽略，就可以包含HTML标签)，不鼓励使用纯文本描述。 -->
<!-- 如果你需要修改产生的web站点的索引页面，你应该修改你自己的索引页文件，而不是调整这里的文档。 -->
<description>A maven project to study maven. </description>

<!--项目创建年份，4位数字。当产生版权信息时需要使用这个值。 -->
<inceptionYear />

<!--项目相关邮件列表信息 -->
<mailingLists>
  <!--该元素描述了项目相关的所有邮件列表。自动产生的网站引用这些信息。 -->
  <mailingList>
    <!--邮件的名称 -->
    <name> Demo </name>
    <!--发送邮件的地址或链接，如果是邮件地址，创建文档时，mailto：链接会被自动创建 -->
    <post> clf@126.com</post>
    <!--订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto：链接会被自动创建 -->
    <subscribe> clf@126.com</subscribe>
    <!--取消订阅邮件的地址或链接，如果是邮件地址，创建文档时，mailto：链接会被自动创建 -->
    <unsubscribe> clf@126.com</unsubscribe>
    <!--你可以浏览邮件信息的URL -->
    <archive> http://hi.clf.com/</archive>
  </mailingList>
</mailingLists>

<!--项目开发者列表 -->
<developers>

  <!--某个项目开发者的信息 -->
  <developer>
    <!--SCM里项目开发者的唯一标识符 -->
    <id> HELLO WORLD </id>
    <!--项目开发者的全名 -->
    <name> banseon </name>
    <!--项目开发者的email -->
    <email> banseon@126.com</email>
    <!--项目开发者的主页的URL -->
    <url />
    <!--项目开发者在项目中扮演的角色，角色元素描述了各种角色 -->
    <roles>
      <role> Project Manager</role>
      <role>Architect </role>
    </roles>
    <!--项目开发者所属组织 -->
    <organization> demo</organization>
    <!--项目开发者所属组织的URL -->
    <organizationUrl>http://hi.clf.com/ </organizationUrl>
    <!--项目开发者属性，如即时消息如何处理等 -->
    <properties>
      <dept> No </dept>
    </properties>
    <!--项目开发者所在时区， -11到12范围内的整数。 -->
```

```
        <timezone> -5</timezone>
    </developer>

</developers>

<!--项目的其他贡献者列表 -->
<contributors>

    <!--项目的其他贡献者。参见developers/developer元素 -->
    <contributor>
        <name />
        <email />
        <url />
        <organization />
        <organizationUrl />
        <roles />
        <timezone />
        <properties />
    </contributor>

</contributors>

<!--该元素描述了项目所有License列表。应该只列出该项目的license列表，不要列出依赖项目的license列表。
-->
<!--如果列出多个license，用户可以选择它们中的一个而不是接受所有license。 -->
<licenses>

    <!--描述了项目的license，用于生成项目的web站点的license页面，其他一些报表和validation也会用到
该元素。 -->
    <license>

        <!--license用于法律上的名称 -->
        <name> Apache 2 </name>
        <!--官方的license正文页面的URL -->
        <url>http://www.clf.com/LICENSE-2.0.txt </url>
        <!--项目分发的主要方式： repo，可以从Maven库下载 manual， 用户必须手动下载和安装依赖 -->
        <distribution> repo</distribution>
        <!--关于license的补充信息 -->
        <comments> Abusiness-friendly OSS license </comments>
    </license>

</licenses>

<!--描述项目所属组织的各种属性。Maven产生的文档用 -->
<organization>

    <!--组织的全名 -->
    <name> demo </name>
    <!--组织主页的URL -->
    <url> http://www.clf.com/</url>

</organization>
```

...还有很多

还有一些可以了解的不常用的配置如报表配置、问题管理配置、项目集成配置、profile配置有兴趣的自行学习一下

## 五、Maven仓库

任何的构件都有唯一的坐标，Maven根据这个坐标定义了构件在仓库中的唯一存储路径，

Maven仓库分为2类：

- 本地仓库
- 远程仓库，远程仓库又分成3种：
  - 中央仓库
  - 私服
  - 其它公共库

### 1、本地仓库

本地仓库是Maven在本地存储构建的地方。Maven的本地仓库，在安装maven后并不会创建，它是在第一次执行Maven命令的时候才被创建。

Maven本地仓库的默认位置：无论是Windows还是Linux，在用户的目录下都有一个.m2/repository/的仓库目录，这就是Maven仓库的默认位置。

在Maven的目录下的conf目录下，有一个settings.xml文件，是Maven的配置文件，在里面可以修改本地仓库的位置。

**修改本地仓库位置：**

```
<!-- 本地仓库的路径。默认值为 -->
<localRepository>D:/myworkspace/maven_repository</localRepository>
```

### 2、远程仓库

远程仓库主要用于获取其它人的Maven构件，其中最主要的就是中央仓库。关于如何在Maven中配置远程仓库，可以查看POM文件章节。

#### 2.1 中央仓库

中央仓库是默认的远程仓库，Maven在安装的时候，自带的就是中央仓库的配置。

所有的Maven都会继承超级POM，超级POM中包含如下配置：

```
<repositories>
  <repository>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org</url>
    <layout>default</layout>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

中央仓库包含了绝大多数流行的开源Java构件，以及源码、作者信息、SCM、信息、许可证信息等。

还可以在里面配置优先使用的镜像，比如在国内直接连中央仓库速度较慢，一般使用阿里的镜像仓库。

```
<!--镜像列表-->
<mirrors>
  <!--镜像-->
  <mirror>
    <id>alimaven</id>
    <!--被镜像的服务器ID-->
    <mirrorOf>central</mirrorOf>
    <name>aliyun maven</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
  </mirror>
</mirrors>
```

## 2.2 私服

私服是一种特殊的远程仓库，它是架设在局域网内的仓库服务，私服代理广域网上的远程仓库，供局域网内的Maven用户使用。当Maven需要下载构件的时候，它从私服请求，如果私服上不存在该构件，则从外部的远程仓库下载，缓存在私服上之后，在为Maven的下载请求提供服务。

### Maven私服的优点：

1. 加速构建；
2. 节省带宽；
3. 节省中央maven仓库的带宽；
4. 稳定（应付一旦中央服务器出问题的情况）；
5. 可以建立本地内部仓库；
6. 可以建立公共仓库。

maven 在默认情况下是从中央仓库下载构建，也就是 id 为 central 的仓库。如果没有特殊需求，一般只需要将私服地址配置为镜像，同时配置其代理所有的仓库就可以实现通过私服下载依赖的功能。镜像配置如下：

```
<mirror>
  <id>Nexus Mirror</id>
  <name>Nexus Mirror</name>
  <url>http://localhost:8081/nexus/content/groups/public/</url>
  <mirrorOf>*</mirrorOf>
</mirror>
```

## 实战：使用Docker创建Nexus私服

启动Docker，输入命令：

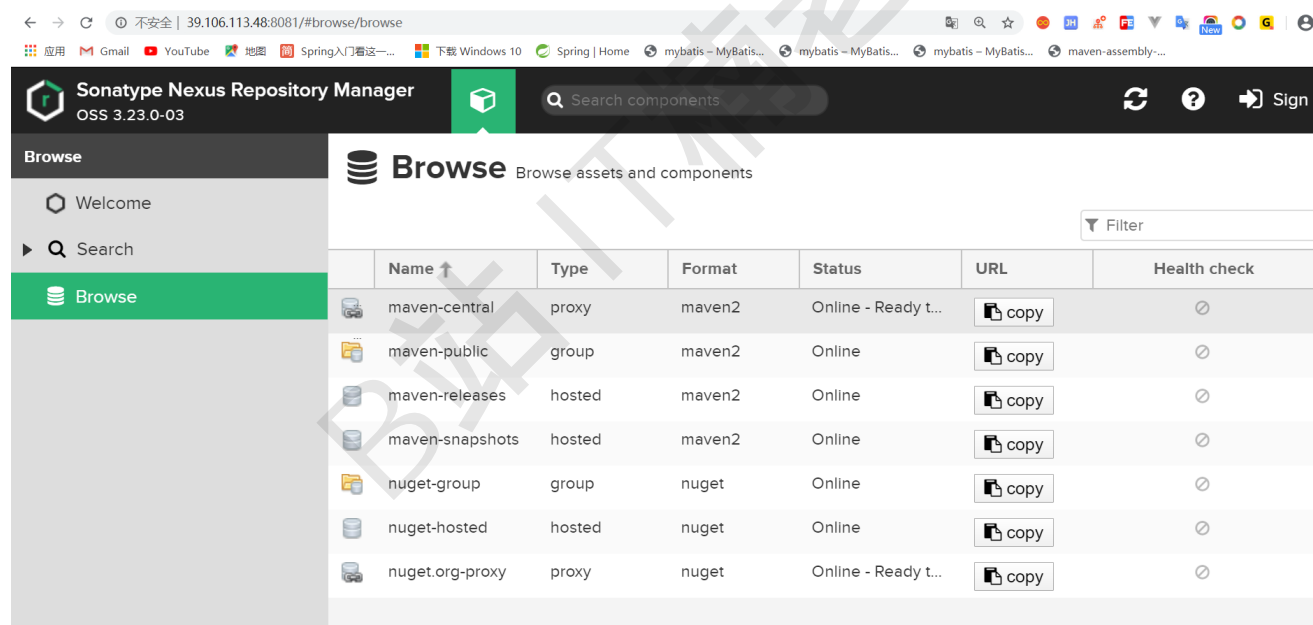
```
docker run -d -p 8081:8081 --name nexus3 sonatype/nexus3
```

运行完毕，在浏览器访问<http://39.106.113.48:8081/> 进入Nexus页面。

如果访问不成功，可以在命令行输入 `docker logs containerId` 查看日志分析原因。

输入默认的用户名和密码admin/admin123即可登录。

进入界面后顶部由2个按钮，分别是仓库和设置。可以看到由4个默认的仓库。



一般来说，Nexus 的仓库分为这么几类：

- hosted 宿主仓库：主要用于部署无法从公共仓库获取的构件（如 oracle 的 JDBC 驱动）以及自己或第三方的项目构件；
- proxy 代理仓库：代理公共的远程仓库；
- virtual 虚拟仓库：用于适配 Maven 1；
- group 仓库组：Nexus 通过仓库组的概念统一管理多个仓库，这样我们在项目中直接请求仓库组即可请求到仓库组管理的多个仓库。



## 以下了解，对私服进行权限认证（了解）

大部分公共的远程仓库无须认证就可以直接访问，但我们在平时的开发中往往会架设自己的Maven远程仓库，出于安全方面的考虑，我们需要提供认证信息才能访问这样的远程仓库。

配置认证信息和配置远程仓库不同，远程仓库可以直接在pom.xml中配置，但是认证信息必须配置在settings.xml文件中。

这是因为pom往往是被提交到代码仓库中供所有成员访问的，而settings.xml一般只存在于本机。因此，在settings.xml中配置认证信息更为安全。

```
<!--配置服务端的一些设置。一些设置如安全证书不应该和pom.xml一起分发。这种类型的信息应该存在于构建服务器上的settings.xml文件中。 -->
<servers>
  <!--服务器元素包含配置服务器时需要的信息 --> <server>
    <!--这是server的id（注意不是用户登陆的id），该id与distributionManagement中repository元素的id相匹配。 -->
    <id>server001</id>
    <!--鉴权用户名。鉴权用户名和鉴权密码表示服务器认证所需要的登录名和密码。 -->
    <username>my_login</username>
    <!--鉴权密码。鉴权用户名和鉴权密码表示服务器认证所需要的登录名和密码。密码加密功能已被添加到2.1.0 +。详情请访问密码加密页面 -->
    <password>my_password</password>
    <!--鉴权时使用的私钥位置。和前两个元素类似，私钥位置和私钥密码指定了一个私钥的路径（默认是${user.home}/.ssh/id_dsa）以及如果需要的话，一个密语。将来passphrase和password元素可能会被提取到外部，但目前它们必须在settings.xml文件以纯文本的形式声明。 -->
    <privateKey>${usr.home}/.ssh/id_dsa</privateKey>
    <!--鉴权时使用的私钥密码。 -->
    <passphrase>some_passphrase</passphrase>
    <!--文件被创建时的权限。如果在部署的时候会创建一个仓库文件或者目录，这时候就可以使用权限(permission)。这两个元素合法的值是一个三位数字，其对应了unix文件系统的权限，如664，或者775。 -->
    <filePermissions>664</filePermissions>
    <!--目录被创建时的权限。 -->
    <directoryPermissions>775</directoryPermissions>
  </server>
</servers>
```

## 六、Maven插件

### 1、Maven 插件介绍

Maven 实际上是一个依赖插件执行的框架，每个任务实际上是由插件完成。Maven 插件通常被用来：

- 打包jar 文件
- 创建 war 文件
- 编译代码文件
- 代码单元测试
- 创建工程文档

- 创建工程报告

插件通常提供了一个目标的集合，并且可以使用下面的语法执行：

```
mvn [plugin-name]:[goal-name]
```

例如，一个 Java 工程可以使用 maven-compiler-plugin 的 compile-goal 编译，使用以下命令：

```
mvn compiler:compile
```

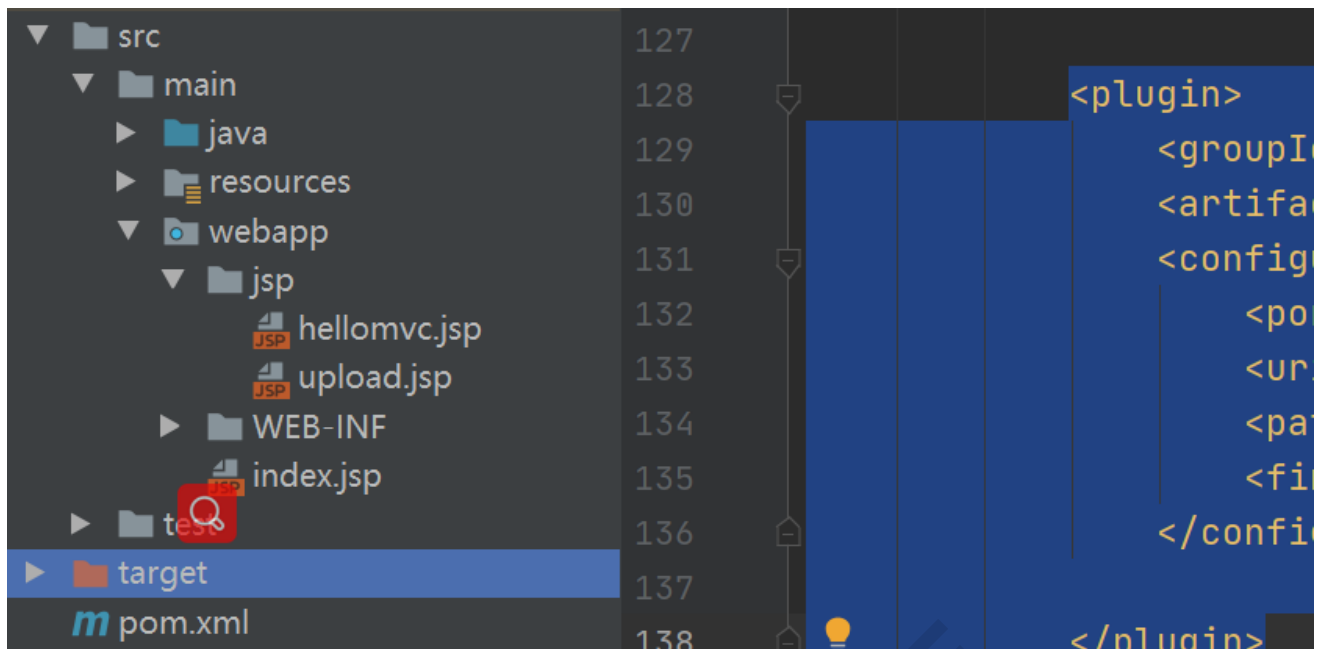
## 2、maven-compiler-plugin

设置maven编译的jdk版本，maven3默认用jdk1.5，maven2默认用jdk1.3

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.8</source> <!-- 源代码使用的JDK版本 -->
    <target>1.8</target> <!-- 需要生成的目标class文件的编译版本 -->
    <encoding>UTF-8</encoding><!-- 字符集编码 -->
    <skipTests>true</skipTests><!-- 跳过测试 -->
  </configuration>
</plugin>
```

## 3、tomcat插件

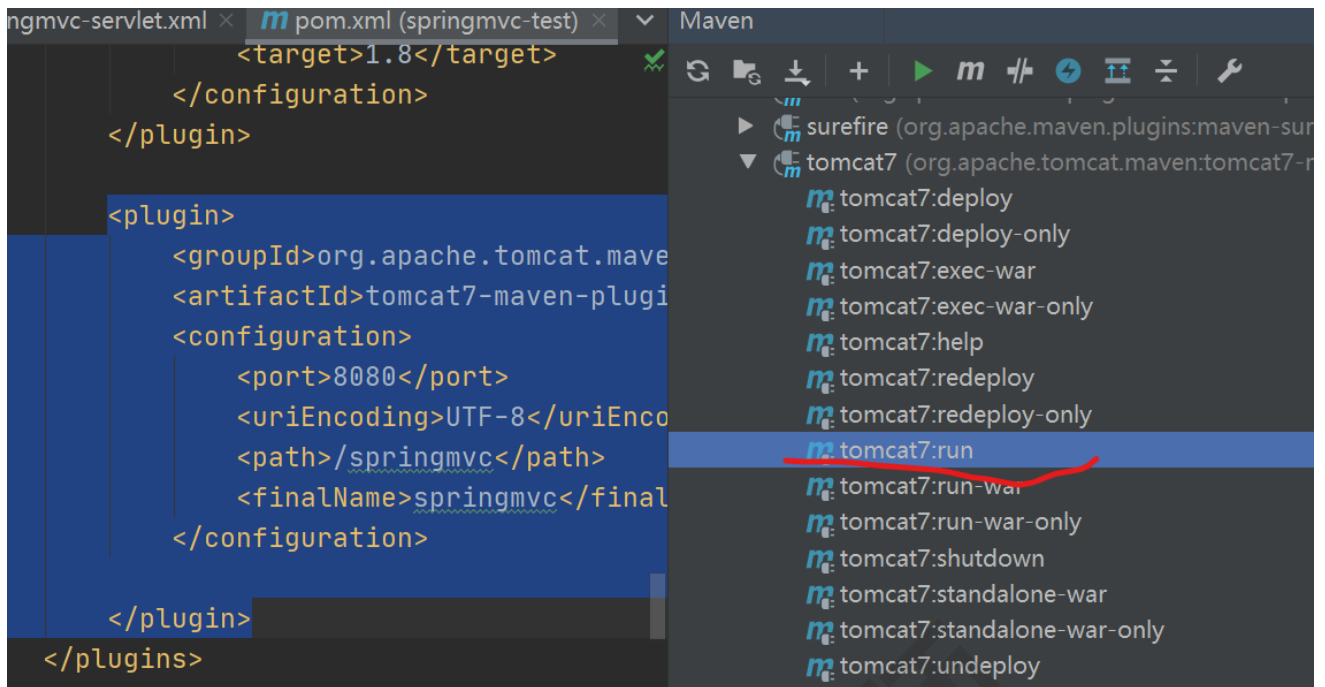
web目录结构



添加插件在build中

```
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <configuration>
    <port>8080</port>
    <uriEncoding>UTF-8</uriEncoding>
    <path>/xinzhi</path>
    <finalName>xinzhi</finalName>
  </configuration>
</plugin>
```

点击idea右侧的maven我们可以方便的看到我们使用了什么插件，并可以点击执行相应的命令



```
D:\project\springmvc-test>mvn tomcat7:run
[INFO] Scanning for projects...
[WARNING]
[WARNING] Some problems were encountered while building the effective model for org.example:springmvc-test:1.0-SNAPSHOT
[WARNING] 'build.plugins.plugin.version' for org.apache.tomcat.maven:tomcat7-maven-plugin is missing
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects
[WARNING]
```

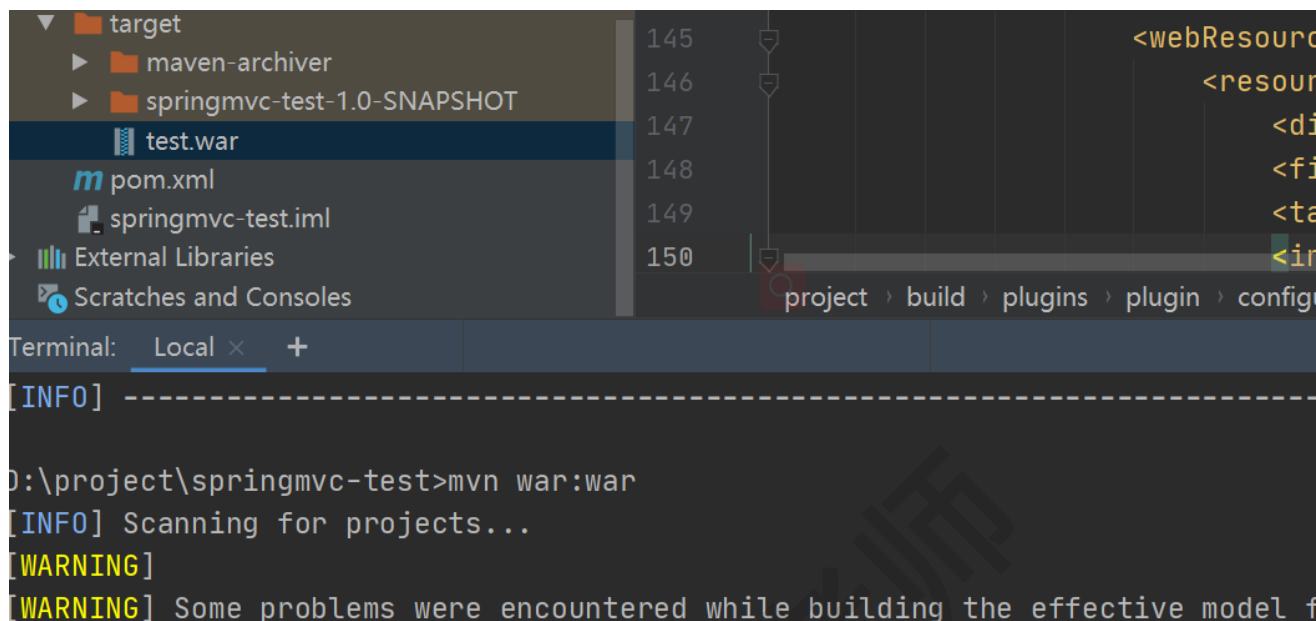
通过插件和命令我们都可以启动项目了，都不用部署到tomcat里了。

## 4、war包打插件

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <warName>test</warName>
    <webResources>
      <resource>
        <directory>src/main/webapp/WEB-INF</directory>
        <filtering>true</filtering>
        <targetPath>WEB-INF</targetPath>
        <includes>
          <include>web.xml</include>
        </includes>
      </resource>
    </webResources>
  </configuration>
</plugin>
```

```
</configuration>
</plugin>
```

### 执行命令 mvn clean package



## 七、Maven项目模板（了解）

Archetype 是一个 Maven 插件，其任务是按照其模板来创建一个项目结构。

执行如下命令即可创建Maven项目模板。

```
mvn archetype:generate
```

常用的archetype有以下2种：

### maven-archetype-quickstart默认的Archetype

- 基本内容包括：
- 一个包含junit依赖声明的pom.xml
- src/main/java主代码目录及一个名为App的类
- src/test/java测试代码目录及一个名为AppTest的测试用例

```
mvn archetype:generate -DgroupId=org.seckill -DartifactId=seckill -
-DarchetypeArtifactId=maven-archetype-webapp
```

### maven-archetype-webapp

一个最简单的Maven war项目模板，当需要快速创建一个Web应用的时候可以使用它。生成的项目内容包括：

- 一个packaging为war且带有junit依赖声明的pom.xml
- src/main/webapp/目录
- src/main/webapp/index.jsp文件
- src/main/webapp/WEB-INF/web.xml文件

« 本地磁盘 (D:) » 教学资料 » 总结文档 » 组件 » maven » seckill » src » main »					▼	🔍
名称	修改日期	类型	大小			
resources	2020/5/25 15:38	文件夹				
webapp	2020/5/25 15:38	文件夹				

其实这个模板并不全。

**B站: IT楠老师 公众号: IT楠说java QQ群: 1083478826 新知大数据**

制作不易、如果觉得的好不妨打个赏:



微信支付

**完结!**