

Maven高级

今日目标

- 理解并实现分模块开发
- 能够使用聚合工程快速构建项目
- 能够使用继承简化项目配置
- 能够根据需求配置生成、开发、测试环境，并在各个环境间切换运行
- 了解Maven的私服

1. 分模块开发

1.1 分模块开发设计

(1) 按照功能拆分

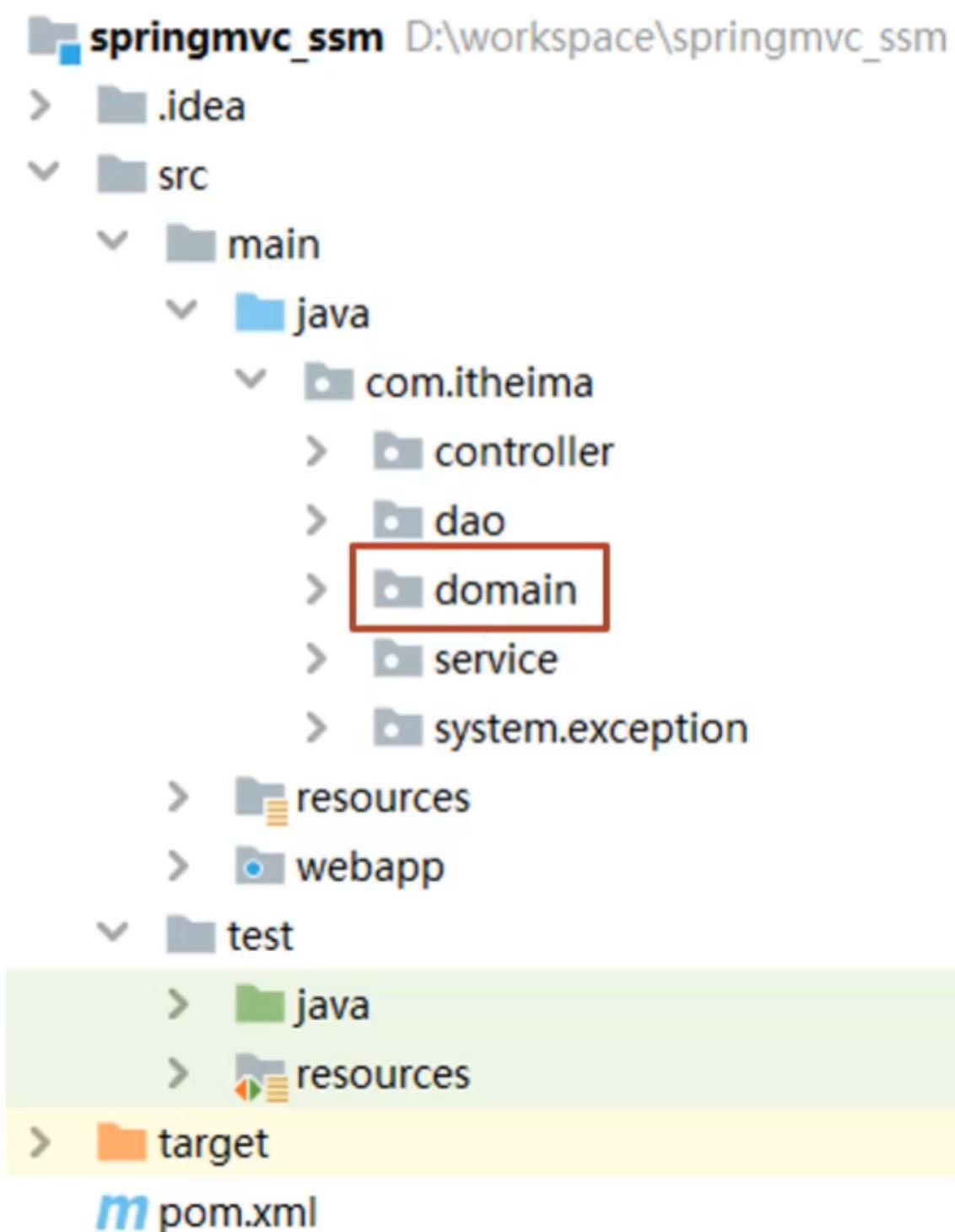
我们现在的项目都是在一个模块中，比如前面的SSM整合开发。虽然这样做功能也都实现了，但是也存在了一些问题，我们拿银行的项目为例来聊聊这个事。

- 网络没有那么发达的时候，我们需要到银行柜台或者取款机进行业务操作
- 随着互联网的发展，我们有了电脑以后，就可以在网页上登录银行网站使用U盾进行业务操作
- 再来就是随着智能手机的普及，我们只需要用手机登录APP就可以进行业务操作

上面三个场景出现的时间是不相同的，如果非要把三个场景的模块代码放入到一个项目，那么当其中某一个模块代码出现问题，就会导致整个项目无法正常启动，从而导致银行的多个业务都无法正常办理。所以我们会**按照功能**将项目进行拆分。

(2) 按照模块拆分

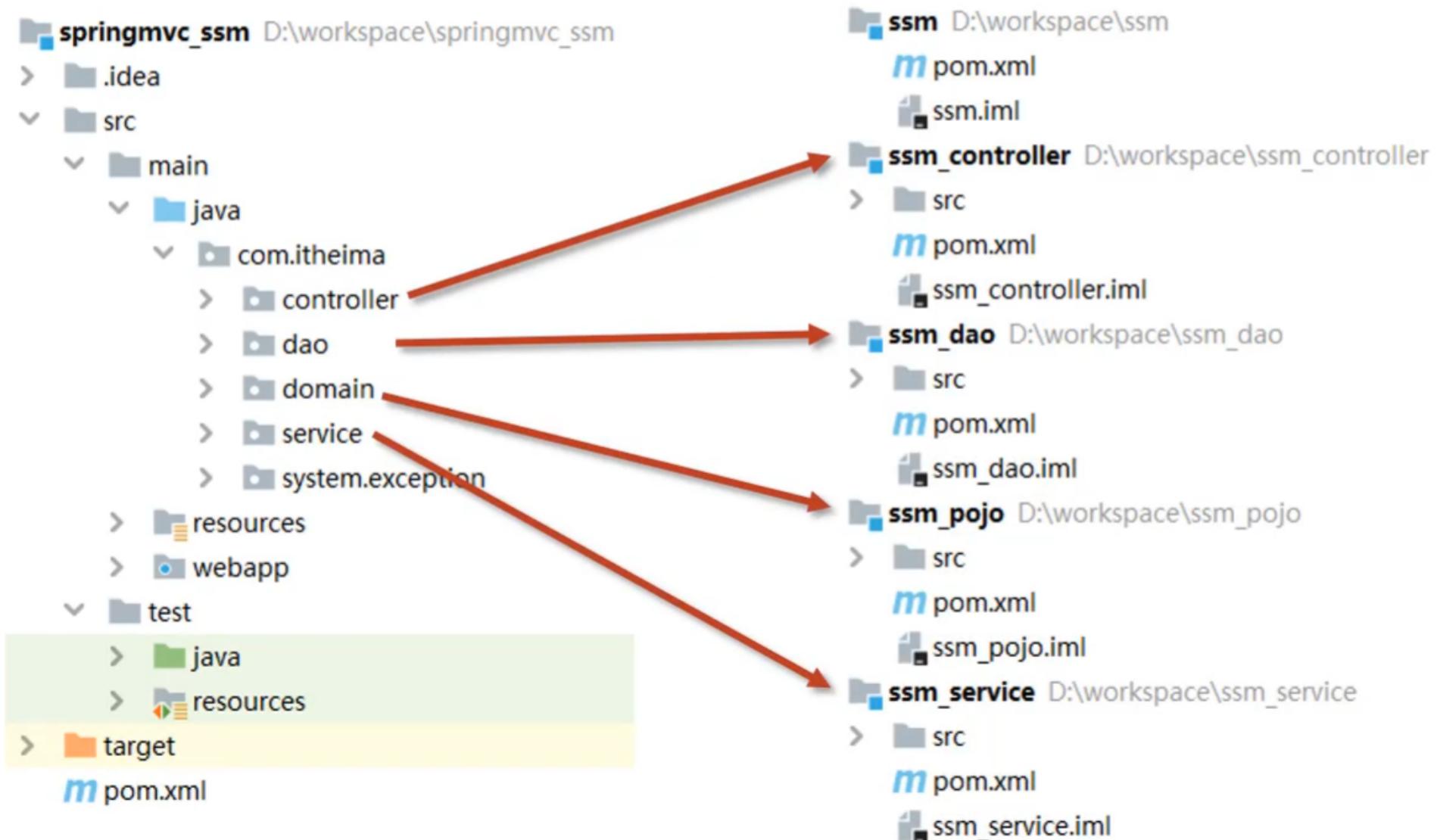
比如电商的项目中，有订单和商品两个模块，订单中需要包含商品的详细信息，所以需要商品的模型类，商品模块也会用到商品的模型类，这个时候如果两个模块中都写模型类，就会出现重复代码，后期的维护成本就比较高。我们就想能不能将它们公共的部分抽取成一个独立的模块，其他模块要想使用可以像添加第三方jar包依赖一样来使用我们自己抽取的模块，这样就解决了代码重复的问题，这种拆分方式就说我们所说的**按照模块**拆分。



经过两个案例的分析，我们就知道：

- 将原始模块按照功能拆分成若干个子模块，方便模块间的相互调用，接口共享。

刚刚我们说了可以将domain层进行拆分，除了domain层，我们也可以将其他的层也拆成一个个对立的模块，如：



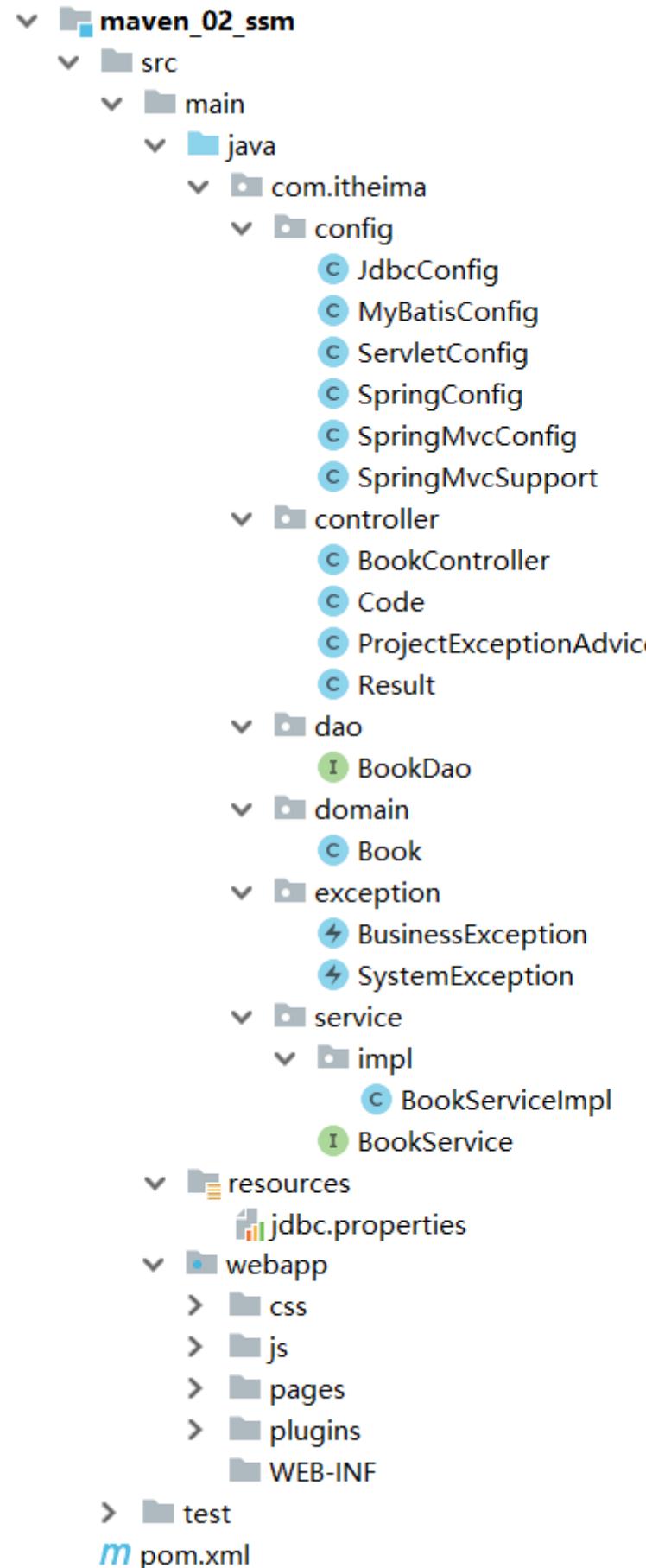
这样的话，项目中的每一层都可以单独维护，也可以很方便的被别人使用。关于分模块开发的意义，我们就说完了，说了这么多好处，那么该如何实现呢？

1.2 分模块开发实现

前面我们已经完成了SSM整合，接下来，咱们就基于SSM整合的项目来实现对项目的拆分。

1.2.1 环境准备

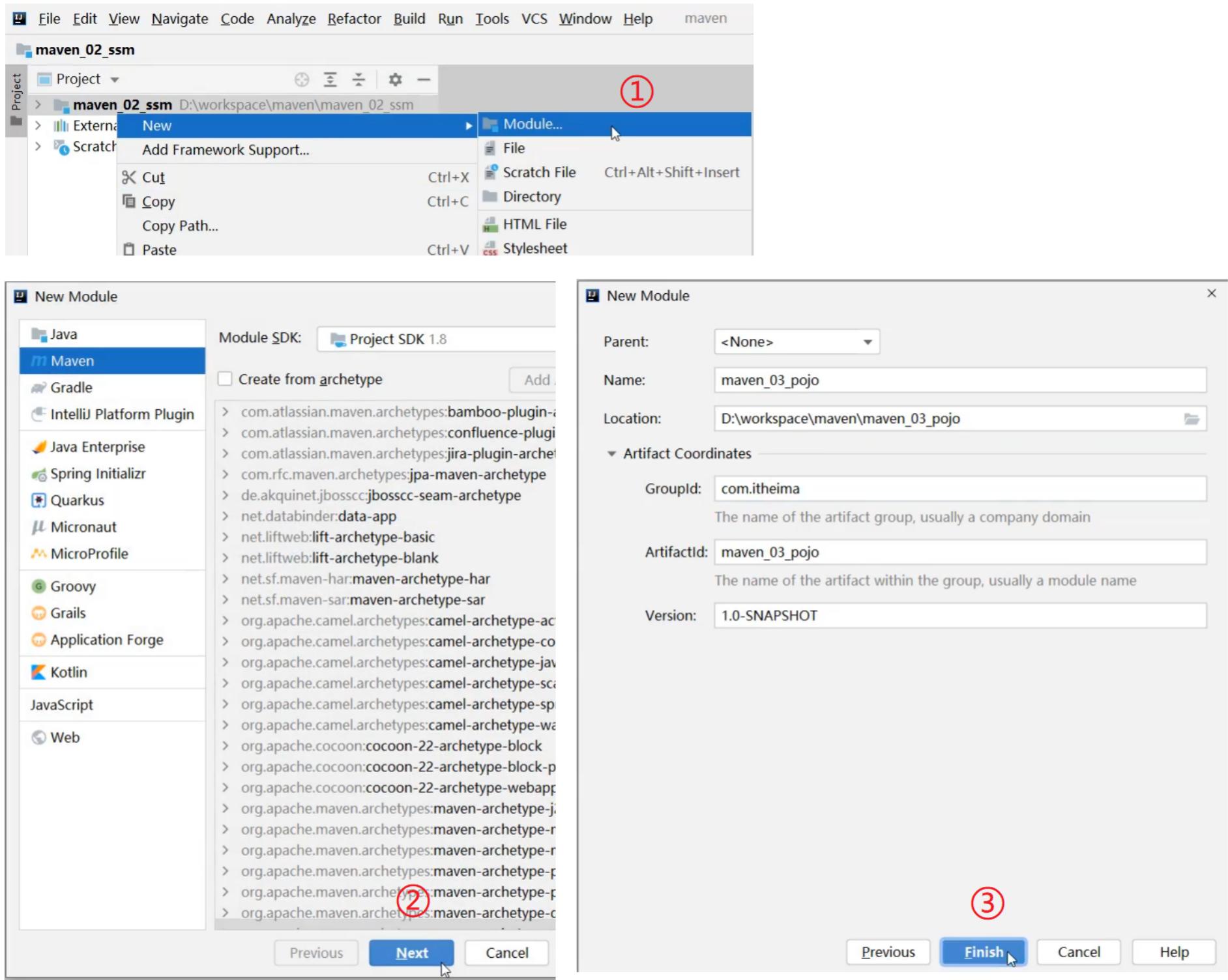
将资料\maven_02_ssm部署到IDEA中，将环境快速准备好，部署成功后，项目的格式如下：



1.2.2 抽取domain层

步骤1：创建新模块

创建一个名称为maven_03_pojo的jar项目，为什么项目名是从02到03这样创建，原因后面我们会提到，这块的名称可以任意。



步骤2：项目中创建domain包

在 maven_03_pojo 项目中创建 com.itheima.domain 包，并将 maven_02_ssm 中 Book 类拷贝到该包中



步骤3：删除原项目中的domain包

删除后， maven_02_ssm 项目中用到 Book 的类中都会有红色提示，如下：

```
1 package com.itheima.service;
2
3 import ...
7
8 @Transactional
9 public interface BookService {
10
11     /**
12      * 保存
13      * @param book
14      * @return
15     */
16     public boolean save(Book book);
17
18     /**
19      * 修改
20      * @param book
21      * @return
22     */
23     public boolean update(Book book);
24 }
```

说明:出错的原因是maven_02_ss中已经将Book类删除，所以该项目找不到Book类，所以报错
要想解决上述问题，我们需要在maven_02_ss中添加maven_03_pojo的依赖。

步骤4:建立依赖关系

在maven_02_ss项目的pom.xml添加maven_03_pojo的依赖

```
1 <dependency>
2     <groupId>com.itheima</groupId>
3     <artifactId>maven_03_pojo</artifactId>
4     <version>1.0-SNAPSHOT</version>
5 </dependency>
```

因为添加了依赖，所以在maven_02_ss中就已经能找到Book类，所以刚才的报红提示就会消失。

步骤5:编译maven_02_ss项目

编译maven_02_ss你会在控制台看到如下错误

```
maven_02_ssm / pom.xml
Project pom.xml (maven_02_ssm) pom.xml (maven_03_pojo)
src/main/java/com.itheima/config/controller/dao/exception/service/iml
<artifactId>maven_03_pojo</artifactId>
<version>1.0-SNAPSHOT</version>
</project>

Run: maven_02_ssm [compile]
maven_02_ssm [compil] 1 sec, 636 ms
com.itheima:maven_02_ssm 283 ms
dependencies 1 warnir 175 ms
  Could not find artifact com.itheima:maven_03_pojo:jar:1.0-SNAPSHOT
    The POM for com.itheima:maven_03_pojo:jar:1.0-SNAPSHOT could not be resolved
      Could not find artifact com.itheima:maven_03_pojo:jar:1.0-SNAPSHOT
      Could not find artifact com.itheima:maven_03_pojo:jar:1.0-SNAPSHOT

com.itheima:maven_02_ssm:war:1.0-SNAPSHOT Could not find artifact com.itheima:maven_03_pojo:jar:1.0-SNAPSHOT -> [Help 1]
```

错误信息为：不能解决maven_02_ssm项目的依赖问题，找不到maven_03_pojo这个jar包。

为什么找不到呢？

原因是Maven会从本地仓库找对应的jar包，但是本地仓库又不存在该jar包所以会报错。

在IDEA中是有maven_03_pojo这个项目，所以我们只需要将maven_03_pojo项目安装到本地仓库即可。

步骤6：将项目安装本地仓库

将需要被依赖的项目maven_03_pojo，使用maven的install命令，将其安装到Maven的本地仓库中。

```
<artifactId>maven_03_pojo</artifactId>
<version>1.0-SNAPSHOT</version>
</project>
```

```
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ maven_03_pojo ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ maven_03_pojo ---
[INFO] Building jar: D:\workspace\maven\maven_03_pojo\target\maven_03_pojo-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ maven_03_pojo ---
[INFO] Installing D:\workspace\maven\maven_03_pojo\target\maven_03_pojo-1.0-SNAPSHOT.jar to D:\maven\repository\com\itheima\maven_03_pojo\1.0-SNAPSHOT\maven_03_pojo-1.0-SNAPSHOT.jar
[INFO] Installing D:\workspace\maven\maven_03_pojo\pom.xml to D:\maven\repository\com\itheima\maven_03_pojo\1.0-SNAPSHOT\maven_03_pojo-1.0-SNAPSHOT.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.892 s
[INFO] Finished at: 2021-05-28T11:04:53+08:00
[INFO] -----
```

安装成功后，在对应的路径下就看到安装好的jar包

1.0-SNAPSHOT			
共享 查看		此电脑 > 本地磁盘 (D:) > maven > repository > com > itheima > maven_03_pojo > 1.0-SNAPSHOT	
名称	修改日期	类型	大小
_remote.repositories	2021/5/28 11:04	REPOSITORIES ...	1 KB
maven_03_pojo-1.0-SNAPSHOT.jar	2021/5/28 11:04	Executable Jar File	3 KB
maven_03_pojo-1.0-SNAPSHOT.pom	2021/5/28 10:58	POM 文件	1 KB
maven-metadata-local.xml	2021/5/28 11:04	XML 文档	1 KB

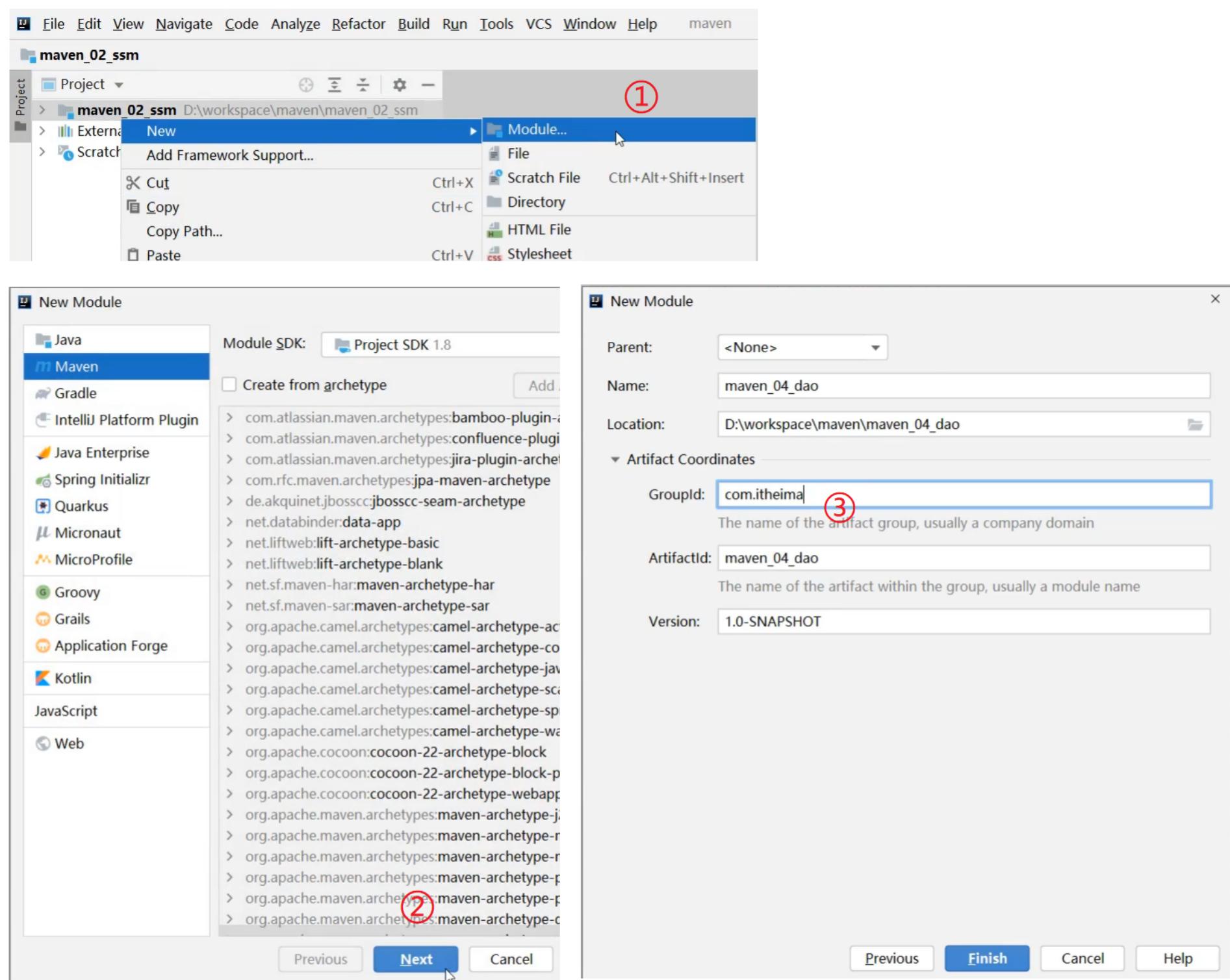
说明:具体安装在哪里，和你们自己电脑上Maven的本地仓库配置的位置有关。

当再次执行maven_02_ssm的compile的命令后，就已经能够成功编译。

1.2.3 抽取Dao层

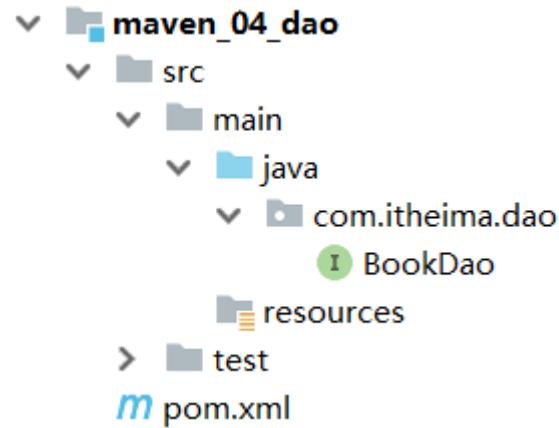
步骤1: 创建新模块

创建一个名称为maven_04_dao的jar项目



步骤2: 项目中创建dao包

在maven_04_dao项目中创建com.itheima.dao包，并将maven_02_ssm中BookDao类拷贝到该包中



在maven_04_dao中会有如下几个问题需要解决下：

```

public interface BookDao {

    //    @Insert("insert into tbl_book vi
    @Insert("insert into tbl_book (ty
    public int save(Book book);

    @Update("update tbl_book set type
    public int update(Book book);

    @Delete("delete from tbl_book whe
    public int delete(Integer id);

    @Select("select * from tbl_book wl
    public Book getById(Integer id);

    @Select("select * from tbl_book")
    public List<Book> getAll();
}

```

- 项目maven_04_dao的BookDao接口中Book类找不到报错
 - 解决方案在maven_04_dao项目的pom.xml中添加maven_03_pojo项目

```

1 <dependencies>
2   <dependency>
3     <groupId>com.itheima</groupId>
4     <artifactId>maven_03_pojo</artifactId>
5     <version>1.0-SNAPSHOT</version>
6   </dependency>
7 </dependencies>

```

- 项目maven_04_dao的BookDao接口中，Mybatis的增删改查注解报错
 - 解决方案在maven_04_dao项目的pom.xml中添加mybatis的相关依赖

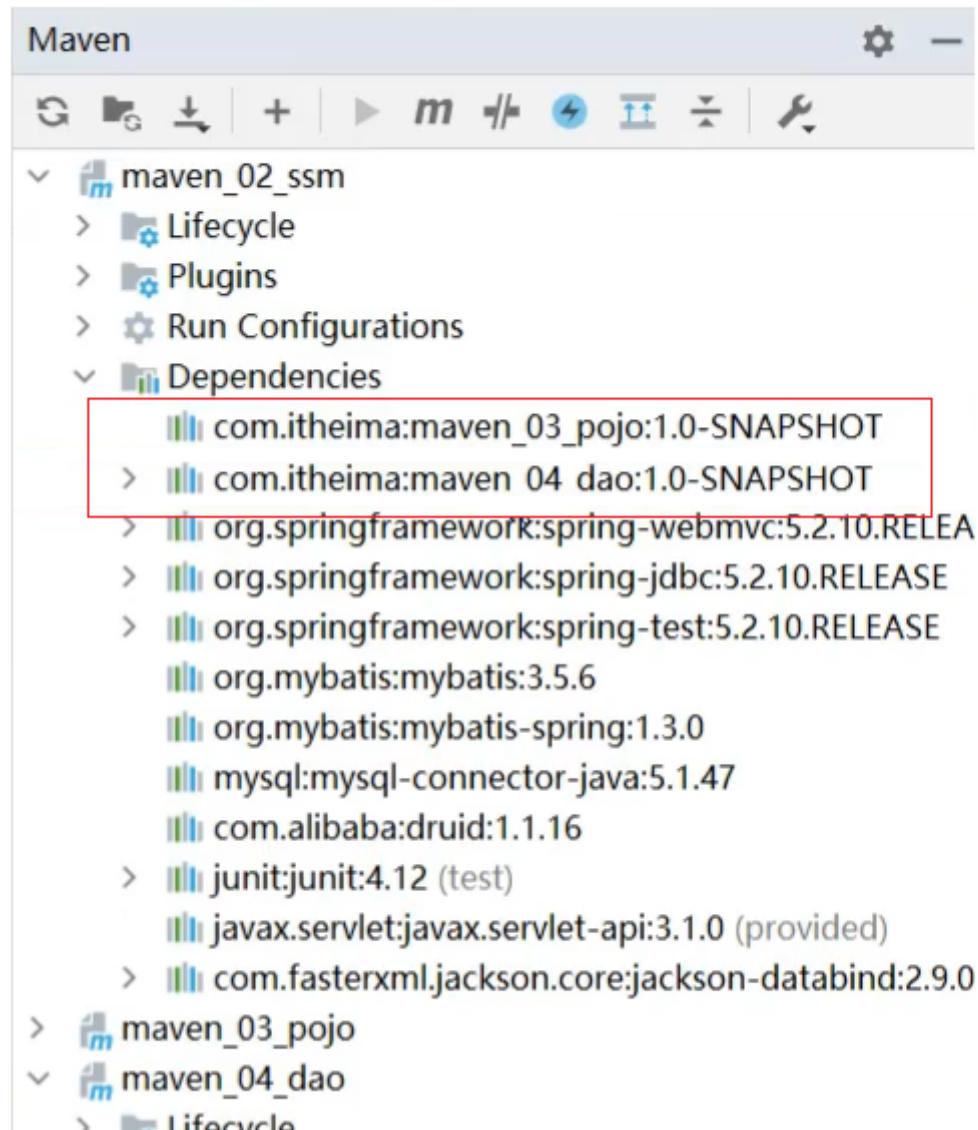
```
1 <dependencies>
2   <dependency>
3     <groupId>org.mybatis</groupId>
4     <artifactId>mybatis</artifactId>
5     <version>3.5.6</version>
6   </dependency>
7
8   <dependency>
9     <groupId>mysql</groupId>
10    <artifactId>mysql-connector-java</artifactId>
11    <version>5.1.47</version>
12  </dependency>
13 </dependencies>
```

步骤3：删除原项目中的dao包

删除Dao包以后，因为maven_02_ssm中的BookServiceImpl类中有使用到Dao的内容，所以需要在maven_02_ssm的pom.xml添加maven_04_dao的依赖

```
1 <dependency>
2   <groupId>com.itheima</groupId>
3   <artifactId>maven_04_dao</artifactId>
4   <version>1.0-SNAPSHOT</version>
5 </dependency>
```

此时在maven_02_ssm项目中就已经添加了maven_03_pojo和maven_04_dao包



再次对maven_02_ssm项目进行编译，又会报错，如下：

The screenshot shows a Maven project structure in the IDE. The pom.xml file is open, showing dependencies for 'maven_03_pojo' and 'maven_04_dao'. A red arrow points from the 'compile' option in the Maven lifecycle menu to the terminal window below, which displays an error message: 'project com.itheima:maven_02_ssm:war:1.0-SNAPSHOT: Could not find artifact com.itheima:maven_04_dao:jar:1.0-SNAPSHOT -> [Help 1]'. The terminal also shows build times for each step.

```

8 <artifactId>maven_02_ssm</artifactId>
9 <version>1.0-SNAPSHOT</version>
10 <packaging>war</packaging>
11
12 <dependencies>
13
14 <!-- 依赖domain运行-->
15 <dependency>
16 <groupId>com.itheima</groupId>
17 <artifactId>maven_03_pojo</artifactId>
18 <version>1.0-SNAPSHOT</version>
19 </dependency>
20 <!-- 依赖dao运行-->
21 <dependency>
22 <groupId>com.itheima</groupId>
23 <artifactId>maven_04_dao</artifactId>
24 <version>1.0-SNAPSHOT</version>
25 </dependency>

```

project > dependencies > dependency > version

```

658 ms
273 ms
157 ms
project com.itheima:maven_02_ssm:war:1.0-SNAPSHOT: Could not find artifact com.itheima:maven_04_dao:jar:1.0-SNAPSHOT -> [Help 1]

```

和刚才的错误原因是一样的，maven在仓库中没有找到maven_04_dao，所以此时我们只需要将maven_04_dao安装到Maven的本地仓库即可。

步骤4：将项目安装到本地仓库

将需要被依赖的项目maven_04_dao，使用maven的install命令，将其安装到Maven的本地仓库中。

The screenshot shows a Maven project structure in the IDE. The pom.xml file is open, showing dependencies for 'maven_03_pojo' and 'maven_04_dao'. A red arrow points from the 'install' option in the Maven lifecycle menu to the terminal window below, which displays a success message: '[INFO] BUILD SUCCESS'. The terminal also shows build times for each step.

```

10 <packaging>war</packaging>
11
12 <dependencies>
13
14 <!-- 依赖domain运行-->
15 <dependency>
16 <groupId>com.itheima</groupId>
17 <artifactId>maven_03_pojo</artifactId>
18 <version>1.0-SNAPSHOT</version>
19 </dependency>
20 <!-- 依赖dao运行-->
21 <dependency>
22 <groupId>com.itheima</groupId>
23 <artifactId>maven_04_dao</artifactId>
24 <version>1.0-SNAPSHOT</version>
25 </dependency>

```

project > dependencies > dependency > version

```

ms [INFO] -----
ms [INFO] BUILD SUCCESS
ms [INFO] -----
j (L [INFO] Total time: 2.160 s
ms [INFO] Finished at: 2021-05-28T11:10:17+08:00
T INFO1 -----

```

安装成功后，在对应的路径下就看到了安装好对应的jar包

The screenshot shows a file explorer displaying the local Maven repository at 'D:\maven\repository'. It lists two directories: 'maven_03_pojo' and 'maven_04_dao'. The 'maven_04_dao' directory is highlighted with a red border.

名称	修改日期	类型	大小
maven_03_pojo	2021/5/28 11:04	文件夹	
maven_04_dao	2021/5/28 11:10	文件夹	

当再次执行maven_02_ssm的compile的指令后，就已经能够成功编译。

1.2.4 运行测试并总结

将抽取后的项目进行运行，测试之前的增删改查功能依然能够使用。

所以对于项目的拆分，大致会有如下几个步骤：

(1) 创建Maven模块

(2) 书写模块代码

分模块开发需要先针对模块功能进行设计，再进行编码。不会先将工程开发完毕，然后进行拆分。拆分方式可以按照功能拆也可以按照模块拆。

(3) 通过maven指令安装模块到本地仓库(install 指令)

团队内部开发需要发布模块功能到团队内部可共享的仓库中(私服)，私服我们后面会讲解。

2. 依赖管理

我们现在已经能把项目拆分成一个个独立的模块，当在其他项目中想要使用独立出来的这些模块，只需要在其pom.xml使用标签来进行jar包的引入即可。

其实就是依赖，关于依赖管理里面都涉及哪些内容，我们就一个个来学习下：

- 依赖传递
- 可选依赖
- 排除依赖

我们先来说说什么是依赖：

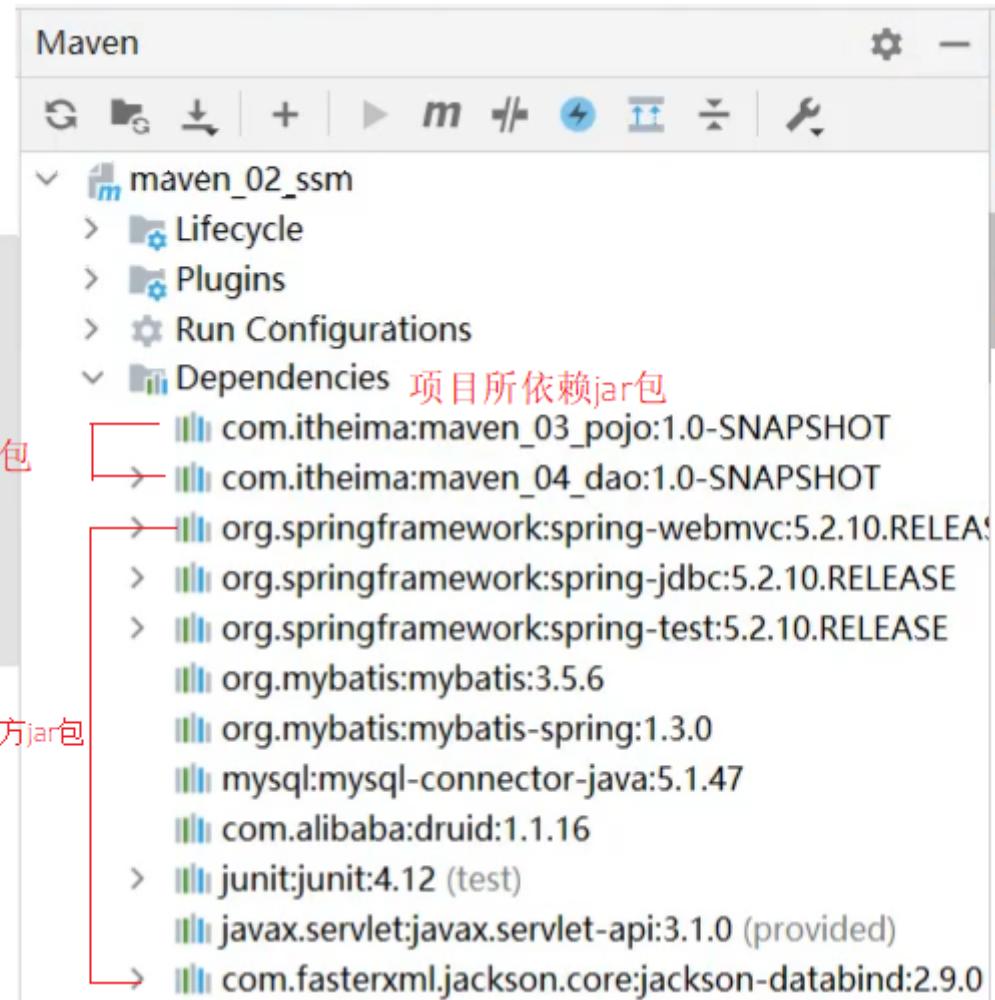
依赖指当前项目运行所需的jar，一个项目可以设置多个依赖。

格式为：

```
1 <!--设置当前项目所依赖的所有jar-->
2 <dependencies>
3   <!--设置具体的依赖-->
4   <dependency>
5     <!--依赖所属群组id-->
6     <groupId>org.springframework</groupId>
7     <!--依赖所属项目id-->
8     <artifactId>spring-webmvc</artifactId>
9     <!--依赖版本号-->
10    <version>5.2.10.RELEASE</version>
11  </dependency>
12 </dependencies>
```

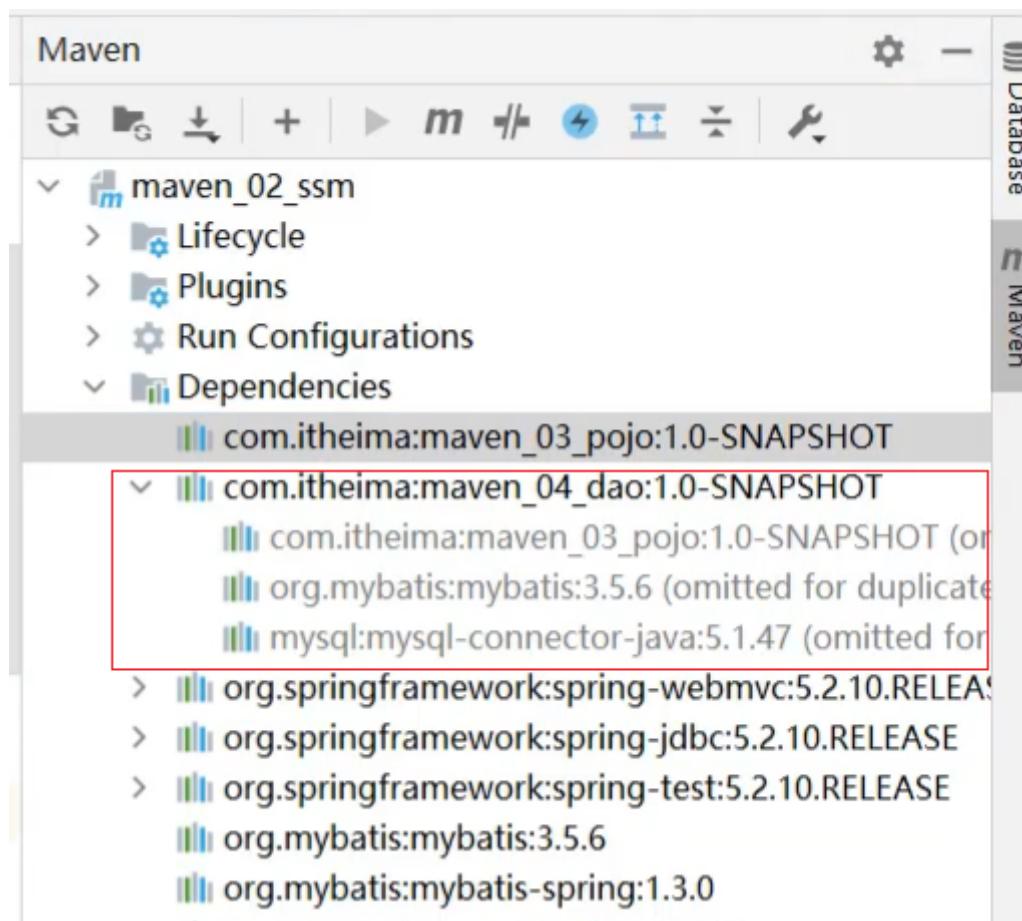
2.1 依赖传递与冲突问题

回到我们刚才的项目案例中，打开Maven的面板，你会发现：



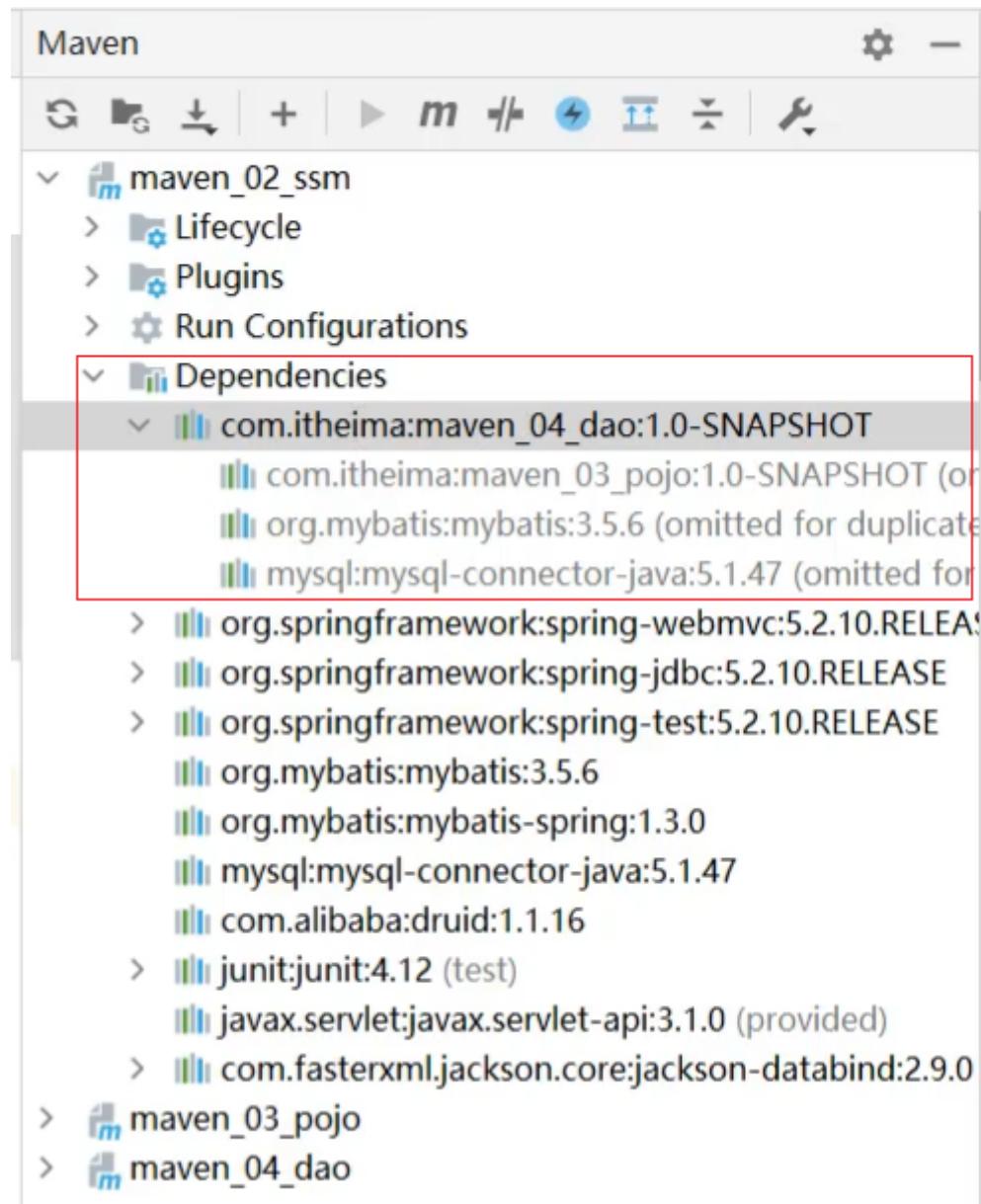
在项目所依赖的这些jar包中，有一个比较大的区别就是有的依赖前面有箭头>，有的依赖前面没有。
那么这个箭头所代表的含义是什么？

打开前面的箭头，你会发现这个jar包下面还包含有其他的jar包



你会发现有两个maven_03_pojo的依赖被加载到Dependencies中，那么maven_04_dao中的maven_03_pojo能不能使用呢？

要想验证非常简单，只需要把maven_02_ssm项目中pom.xml关于maven_03_pojo的依赖注释或删除掉



在Dependencies中移除自己所添加maven_03_pojo依赖后，打开BookServiceImpl的类，你会发现Book类依然存在，可以被正常使用

```
@Transactional
public interface BookService {

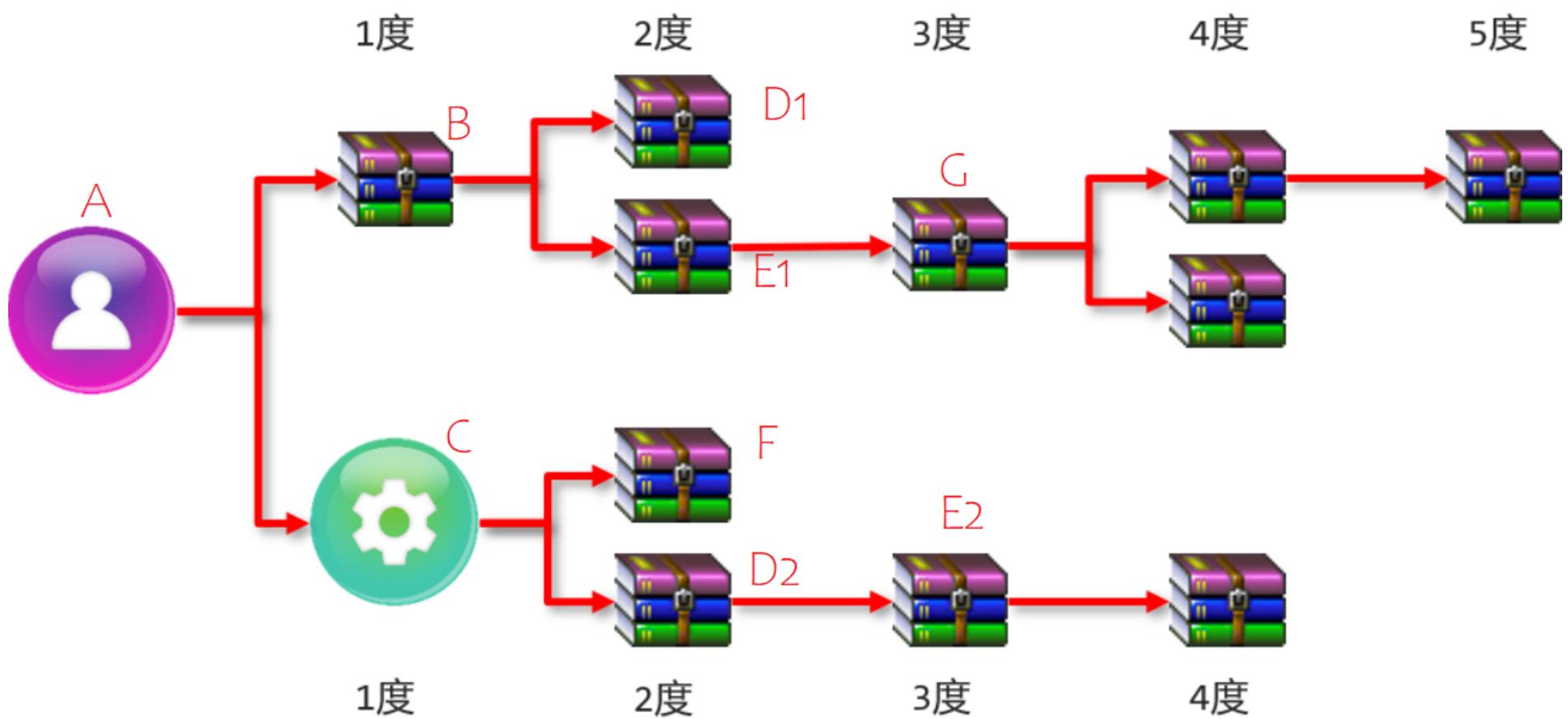
    /**
     * 保存
     * @param book
     * @return
     */
    public boolean save(Book book);

    /**
     * 修改
     * @param book
     * @return
     */
    public boolean update(Book book);

    /**
     * 按id删除
     * @param id
     */
}
```

这个特性其实就是要讲解的**依赖传递**。

依赖是具有传递性的：



说明: A代表自己的项目； B, C, D, E, F, G代表的是项目所依赖的jar包； D1和D2 E1和E2代表是相同jar包的不同版本

(1) A依赖了B和C, B和C有分别依赖了其他jar包，所以在A项目中就可以使用上面所有jar包，这就是所说的依赖传递

(2) 依赖传递有直接依赖和间接依赖

- 相对于A来说，A直接依赖B和C，间接依赖了D1, E1, G, F, D2和E2
- 相对于B来说，B直接依赖了D1和E1，间接依赖了G
- 直接依赖和间接依赖是一个相对的概念

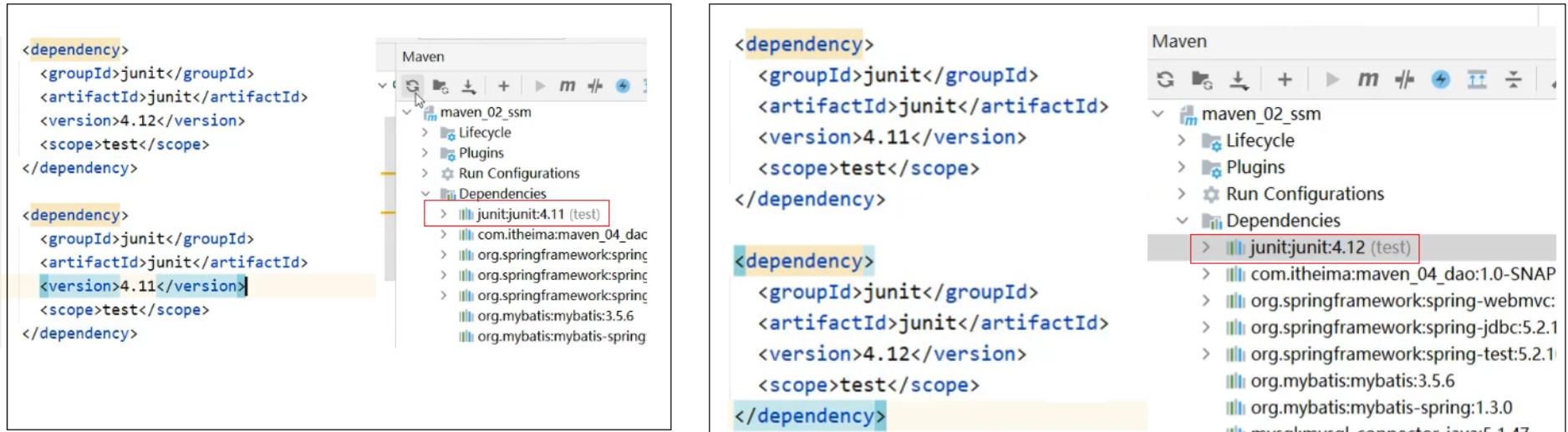
(3) 因为有依赖传递的存在，就会导致jar包在依赖的过程中出现冲突问题，具体什么是冲突?Maven是如何解决冲突的?

这里所说的**依赖冲突**是指项目依赖的某一个jar包，有多个不同的版本，因而造成类包版本冲突。

情况一：在maven_02_ssm的pom.xml中添加两个不同版本的Junit依赖：

```

1 <dependencies>
2   <dependency>
3     <groupId>junit</groupId>
4     <artifactId>junit</artifactId>
5     <version>4.12</version>
6     <scope>test</scope>
7   </dependency>
8
9   <dependency>
10    <groupId>junit</groupId>
11    <artifactId>junit</artifactId>
12    <version>4.11</version>
13    <scope>test</scope>
14  </dependency>
15 </dependencies>
```



通过对比，会发现一个结论

- 特殊优先：当同级配置了相同资源的不同版本，后配置的覆盖先配置的。

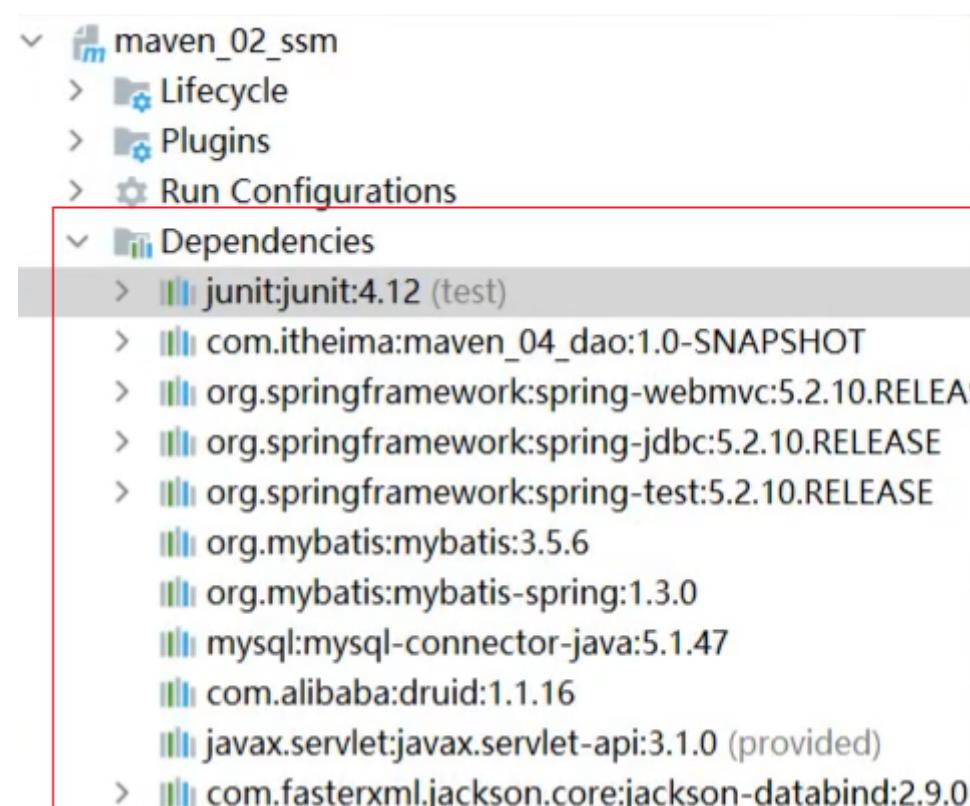
情况二：路径优先：当依赖中出现相同的资源时，层级越深，优先级越低，层级越浅，优先级越高

- A通过B间接依赖到E1
- A通过C间接依赖到E2
- A就会间接依赖到E1和E2，Maven会按照层级来选择，E1是2度，E2是3度，所以最终会选择E1

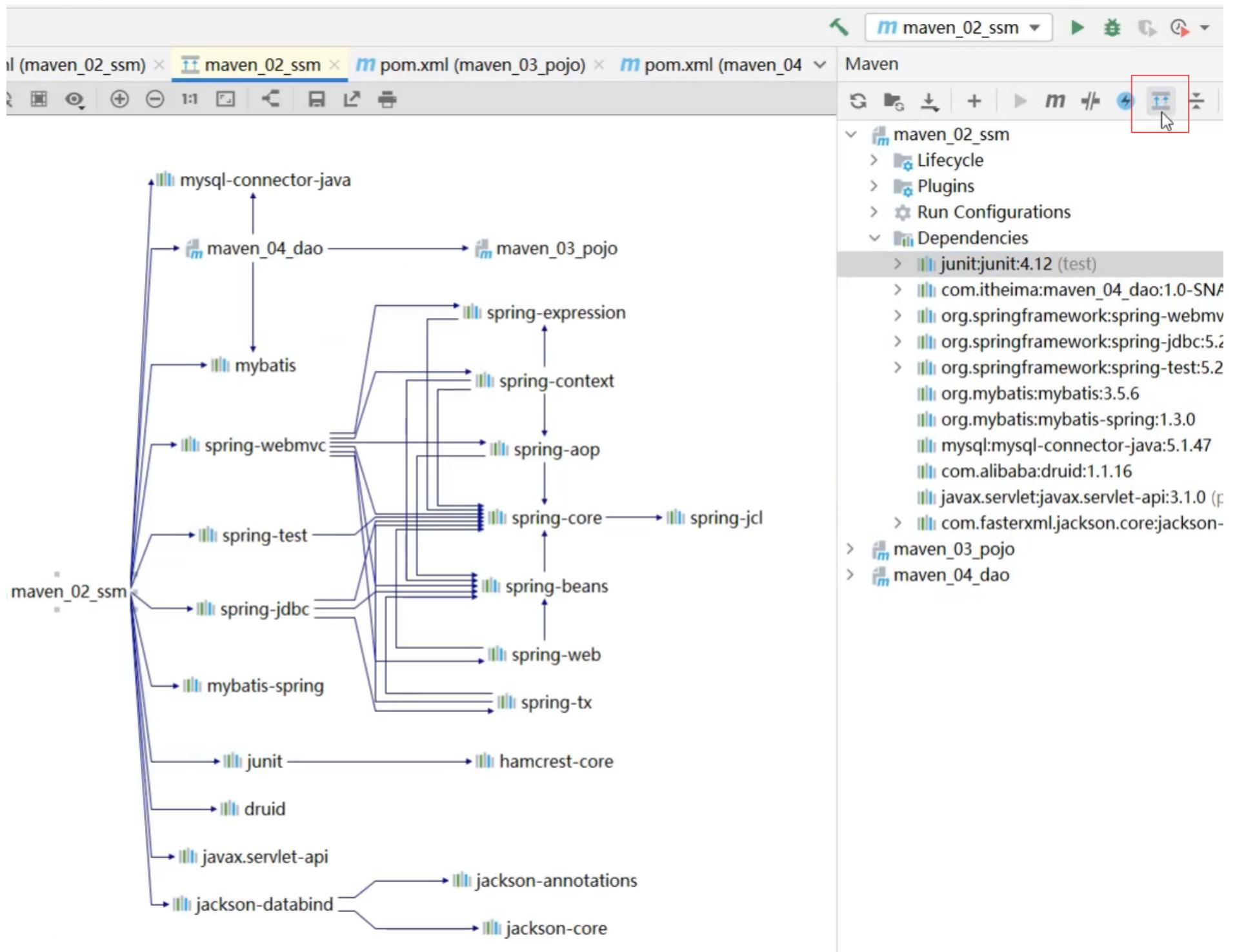
情况三：声明优先：当资源在相同层级被依赖时，配置顺序靠前的覆盖配置顺序靠后的

- A通过B间接依赖到D1
- A通过C间接依赖到D2
- D1和D2都是两度，这个时候就不能按照层级来选择，需要按照声明来，谁先声明用谁，也就是说B在C之前声明，这个时候使用的是D1，反之则为D2

但是对应上面这些结果，大家不需要刻意去记它。因为不管Maven怎么选，最终的结果都会在Maven的Dependencies面板中展示出来，展示的是哪个版本，也就是说它选择的就是哪个版本，如：



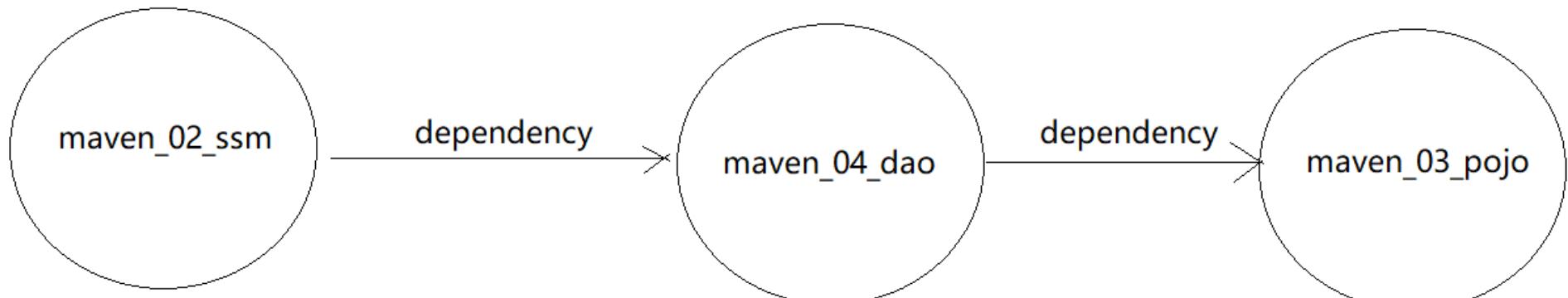
如果想更全面的查看Maven中各个坐标的关系，可以点击Maven面板中的 show Dependencies



在这个视图中就能很明显的展示出jar包之间的相互依赖关系。

2.2 可选依赖和排除依赖

依赖传递介绍完以后，我们来思考一个问题，



- maven_02_ssm 依赖了 maven_04_dao
- maven_04_dao 依赖了 maven_03_pojo
- 因为现在有依赖传递，所以 maven_02_ssm 能够使用到 maven_03_pojo 的内容
- 如果说现在不想让 maven_02_ssm 依赖到 maven_03_pojo，有哪些解决方案？

说明：在真实使用的过程中，maven_02_ssm 中是需要用到 maven_03_pojo 的，我们这里只是用这个例子描述我们的需求。因为有时候，maven_04_dao出于某些因素的考虑，就是不想让别人使用自己所依赖的 maven_03_pojo。

方案一：可选依赖

- 可选依赖指对外隐藏当前所依赖的资源---不透明

在maven_04_dao的pom.xml，在引入maven_03_pojo的时候，添加optional

```

1 <dependency>
2   <groupId>com.itheima</groupId>
3   <artifactId>maven_03_pojo</artifactId>
4   <version>1.0-SNAPSHOT</version>
5   <!--可选依赖是隐藏当前工程所依赖的资源，隐藏后对应资源将不具有依赖传递-->
6   <optional>true</optional>
7 </dependency>

```

此时BookServiceImpl就已经报错了，说明由于maven_04_dao将maven_03_pojo设置成可选依赖，导致maven_02_ssm无法引用到maven_03_pojo中的内容，导致Book类找不到。

```

xml (maven_02_ssm) × BookService.java × pom.xml (m
package com.itheima.service;

import ...

@Transactional
public interface BookService {

    /**
     * 保存
     * @param book
     * @return
     */
    public boolean save(Book book);

    /**
     * 修改
     * @param book
     * @return
     */
    public boolean update(Book book);

    /**
     * 按id删除
     * @param id
     * @return
     */
}

```

方案二：排除依赖

- 排除依赖指主动断开依赖的资源，被排除的资源无需指定版本---不需要

前面我们已经通过可选依赖实现了阻断maven_03_pojo的依赖传递，对于排除依赖，则指的是已经有依赖的事实，也就是说maven_02_ssm项目中已经通过依赖传递用到了maven_03_pojo，此时我们需要做的是将其进行排除，所以接下来需要修改maven_02_ssm的pom.xml

```

1 <dependency>
2   <groupId>com.itheima</groupId>
3   <artifactId>maven_04_dao</artifactId>
4   <version>1.0-SNAPSHOT</version>
5   <!--排除依赖是隐藏当前资源对应的依赖关系-->
6   <exclusions>
7     <exclusion>
8       <groupId>com.itheima</groupId>
9       <artifactId>maven_03_pojo</artifactId>
10      </exclusion>
11    </exclusions>
12  </dependency>

```

这样操作后，BookServiceImpl中的Book类一样也会报错。

当然exclusions标签带s说明我们是可以依次排除多个依赖到的jar包，比如maven_04_dao中有依赖junit和mybatis，我们也可以一并将其排除。

```

1 <dependency>
2   <groupId>com.itheima</groupId>
3   <artifactId>maven_04_dao</artifactId>
4   <version>1.0-SNAPSHOT</version>
5   <!--排除依赖是隐藏当前资源对应的依赖关系-->
6   <exclusions>
7     <exclusion>
8       <groupId>com.itheima</groupId>
9       <artifactId>maven_03_pojo</artifactId>
10      </exclusion>
11      <exclusion>
12        <groupId>log4j</groupId>
13        <artifactId>log4j</artifactId>
14      </exclusion>
15      <exclusion>
16        <groupId>org.mybatis</groupId>
17        <artifactId>mybatis</artifactId>
18      </exclusion>
19    </exclusions>
20  </dependency>

```

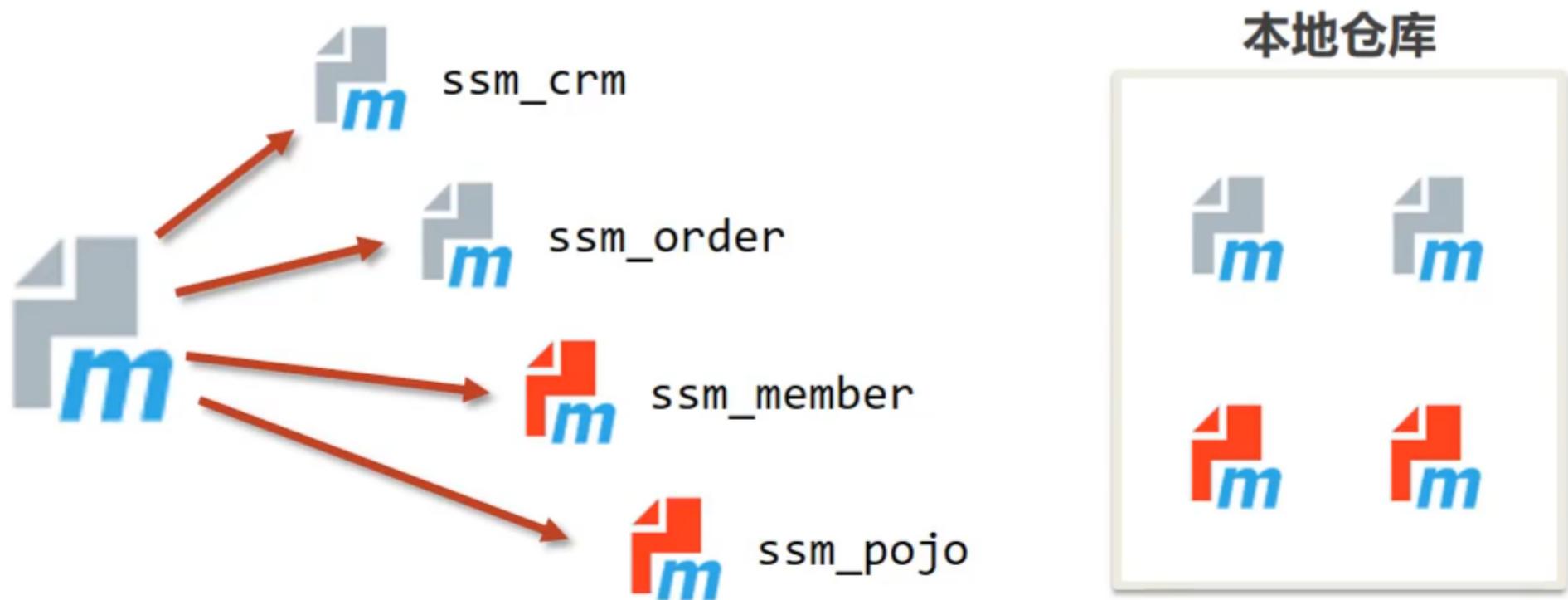
介绍我这两种方式后，简单来梳理下，就是

- A依赖B，B依赖C，C通过依赖传递会被A使用到，现在要想办法让A不去依赖C
- 可选依赖是在B上设置<optional>，A不知道有C的存在，
- 排除依赖是在A上设置<exclusions>，A知道有C的存在，主动将其排除掉。

3，聚合和继承

我们的项目已经从以前的单模块，变成了现在的多模块开发。项目一旦变成了多模块开发以后，就会引发一些问题，在这一节中我们主要会学习两个内容聚合和继承，用这两个知识来解决下分模块后的一些问题。

3.1 聚合



- 分模块开发后，需要将这四个项目都安装到本地仓库，目前我们只能通过项目Maven面板的 `install` 来安装，并且需要安装四个，如果我们的项目足够多，那么一个个安装起来还是比较麻烦的
- 如果四个项目都已经安装成功，当 `ssm_pojo` 发生变化后，我们就得将 `ssm_pojo` 重新安装到 maven 仓库，但是为了确保我们对 `ssm_pojo` 的修改不会影响到其他项目模块，我们需要对所有的模块进行重新编译，那又需要将所有的模块再来一遍

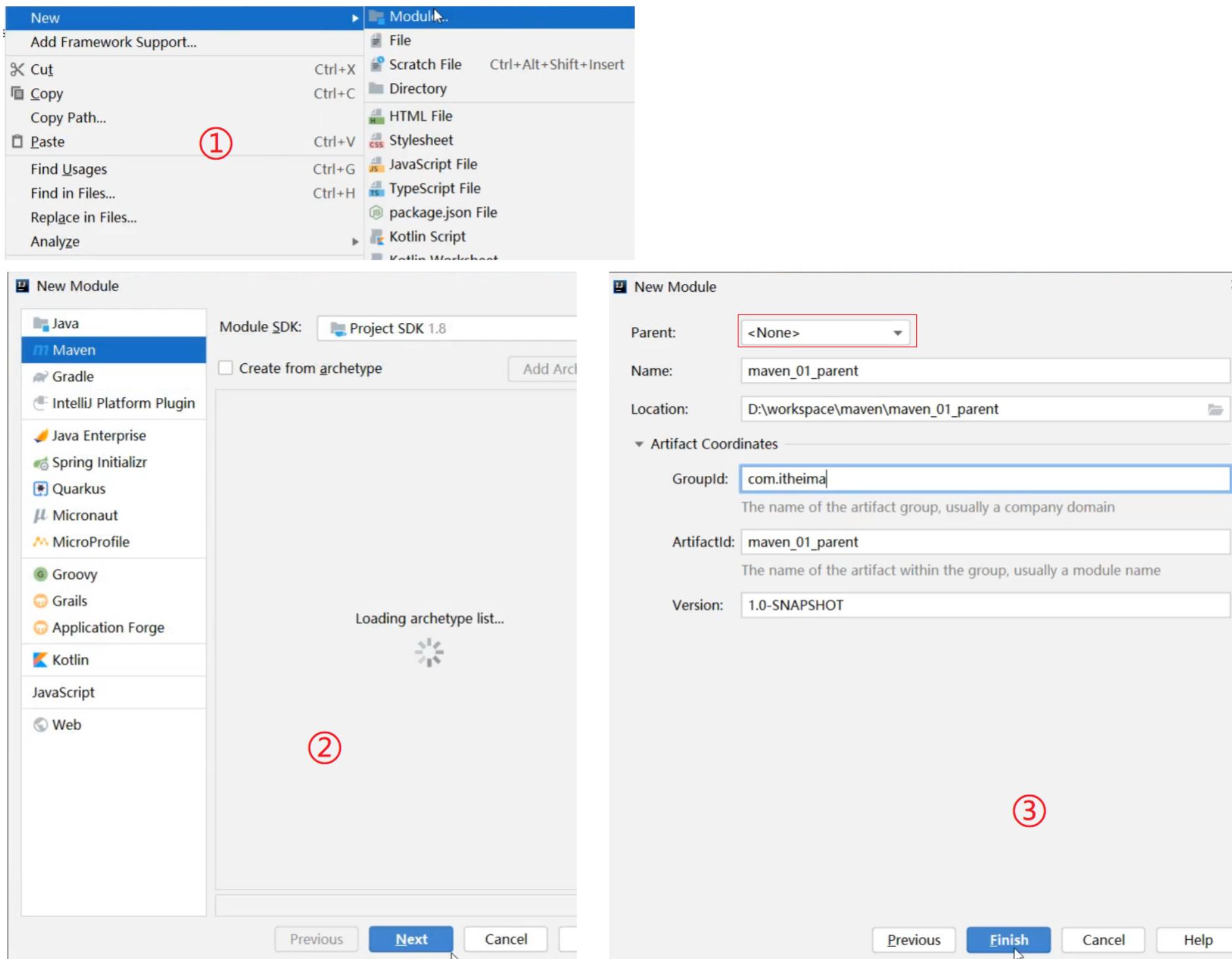
项目少的话还好，但是如果项目多的话，一个个操作项目就容易出现漏掉或重复操作的问题，所以我们就能不能抽取一个项目，把所有的项目管理起来，以后我们要想操作这些项目，只需要操作这一个项目，其他所有的项目都走一样的流程，这个不就很省事省力。

这就用到了我们接下来要讲解的**聚合**，

- 所谓聚合：将多个模块组织成一个整体，同时进行项目构建的过程称为聚合
- 聚合工程：通常是一个不具有业务功能的“空”工程（有且仅有一个pom文件）
- 作用：使用聚合工程可以将多个工程编组，通过对聚合工程进行构建，实现对所包含的模块进行同步构建
 - 当工程中某个模块发生更新（变更）时，必须保障工程中与已更新模块关联的模块同步更新，此时可以使用聚合工程来解决批量模块同步构建的问题。

关于聚合具体的实现步骤为：

步骤1：创建一个空的maven项目



步骤2：将项目的打包方式改为pom

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7
8     <groupId>com.itheima</groupId>
9     <artifactId>maven_01_parent</artifactId>
10    <version>1.0-RELEASE</version>
11    <packaging>pom</packaging>
12 </project>

```

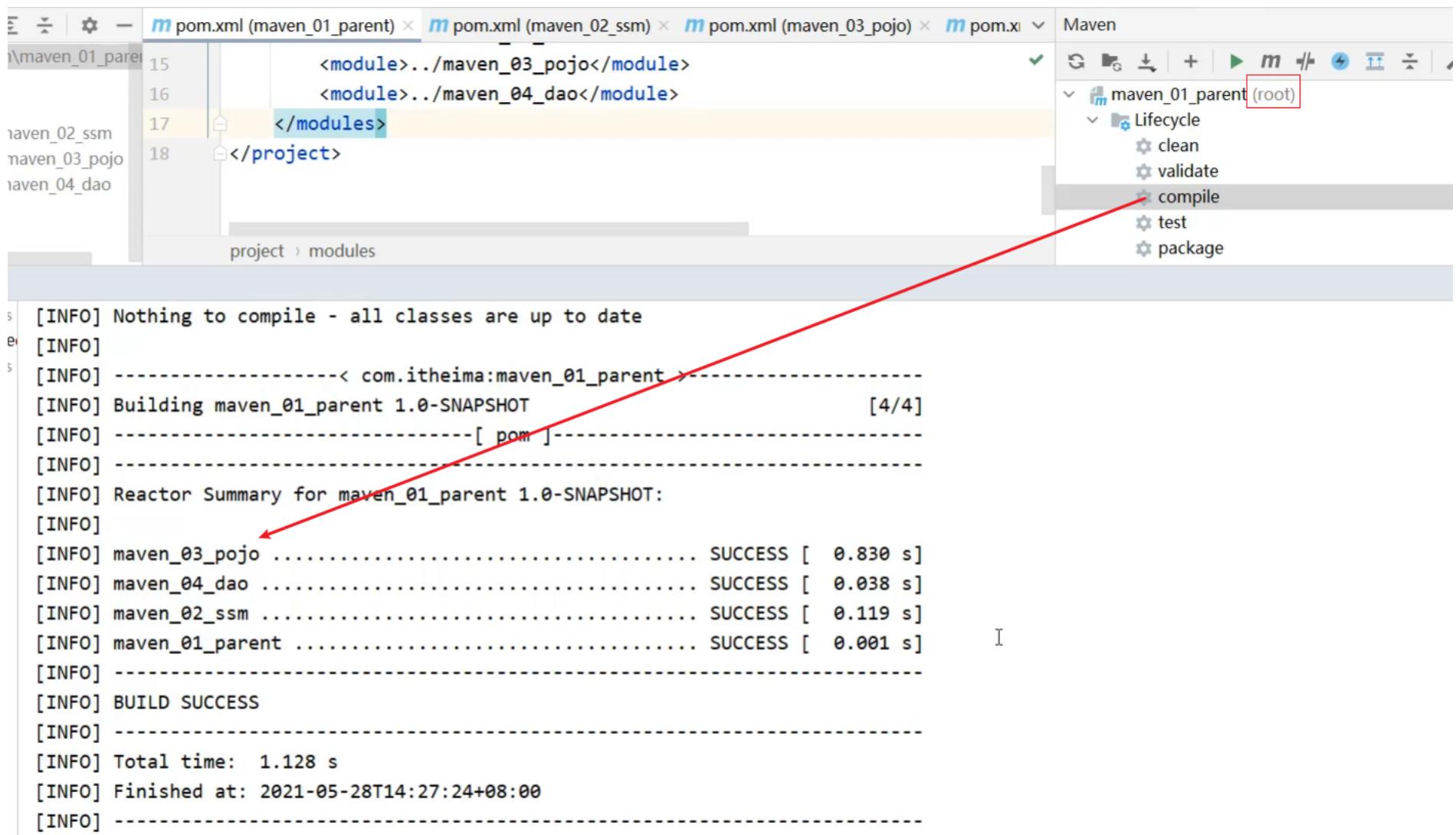
说明：项目的打包方式，我们接触到的有三种，分别是

- jar：默认情况，说明该项目为java项目
- war：说明该项目为web项目
- pom：说明该项目为聚合或继承（后面会讲）项目

步骤3:pom.xml添加所要管理的项目

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5           http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7
8     <groupId>com.itheima</groupId>
9     <artifactId>maven_01_parent</artifactId>
10    <version>1.0-RELEASE</version>
11    <packaging>pom</packaging>
12
13    <!--设置管理的模块名称-->
14    <modules>
15      <module>../maven_02_ssm</module>
16      <module>../maven_03_pojo</module>
17      <module>../maven_04_dao</module>
18    </modules>
19  </project>
```

步骤4:使用聚合统一管理项目



The screenshot shows the IntelliJ IDEA interface with the Maven tool window open. The left pane displays the parent POM file (`pom.xml (maven_01_parent)`) containing module definitions. The right pane shows the Maven tool window with the `maven_01_parent` project selected as the root. A red arrow points from the `compile` option in the tool window to the terminal output below, which shows a successful build process for all modules.

```
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] -----
[INFO] -----< com.itheima:maven_01_parent >-----
[INFO] Building maven_01_parent 1.0-SNAPSHOT [4/4]
[INFO] -----[ pom ]-----
[INFO]
[INFO] Reactor Summary for maven_01_parent 1.0-SNAPSHOT:
[INFO]
[INFO] maven_03_pojo ..... SUCCESS [ 0.830 s]
[INFO] maven_04_dao ..... SUCCESS [ 0.038 s]
[INFO] maven_02_ssm ..... SUCCESS [ 0.119 s]
[INFO] maven_01_parent ..... SUCCESS [ 0.001 s]
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.128 s
[INFO] Finished at: 2021-05-28T14:27:24+08:00
[INFO] -----
```

测试发现，当 `maven_01_parent` 的 `compile` 被点击后，所有被其管理的项目都会被执行编译操作。这就是聚合工程的作用。

说明：聚合工程管理的项目在进行运行的时候，会按照项目与项目之间的依赖关系来自动决定执行的顺序和配置的顺序无关。

聚合的知识我们就讲解完了，最后总结一句话就是，**聚合工程主要是用来管理项目。**

3.2 继承

我们已经完成了使用聚合工程去管理项目，聚合工程进行某一个构建操作，其他被其管理的项目也会执行相同的构建操作。那么接下来，我们再来分析下，多模块开发存在的另外一个问题，**重复配置的问题**，我们先来看张图：



- spring-webmvc、spring-jdbc在三个项目模块中都有出现，这样就出现了重复的内容
- spring-test只在ssm.crm和ssm.goods中出现，而在ssm.order中没有，这里是部分重复的内容
- 我们使用的spring版本目前是5.2.10.RELEASE，假如后期要想升级spring版本，所有跟Spring相关jar包都得被修改，涉及到的项目越多，维护成本越高

面对上面的这些问题，我们就得用到接下来要学习的**继承**

- 所谓继承：描述的是两个工程间的关系，与java中的继承相似，子工程可以继承父工程中的配置信息，常见于依赖关系的继承。
- 作用：
 - 简化配置
 - 减少版本冲突

接下来，我们到程序中去看看继承该如何实现？

步骤1：创建一个空的Maven项目并将其打包方式设置为pom

因为这一步和前面maven创建聚合工程的方式是一摸一样，所以我们可以单独创建一个新的工程，也可以直接和聚合公用一个工程。实际开发中，聚合和继承一般也都放在同一个项目中，但是这两个的功能是不一样的。

步骤2：在子项目中设置其父工程

分别在maven_02_ssm, maven_03_pojo, maven_04_dao的pom.xml中添加其父项目为maven_01_parent

```
1 <!--配置当前工程继承自parent工程-->
2 <parent>
3   <groupId>com.itheima</groupId>
4   <artifactId>maven_01_parent</artifactId>
5   <version>1.0-RELEASE</version>
6   <!--设置父项目pom.xml位置路径-->
7   <relativePath>../maven_01_parent/pom.xml</relativePath>
8 </parent>
```

步骤3：优化子项目共有依赖导入问题

- 将子项目共同使用的jar包都抽取出来，维护在父项目的pom.xml中

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5 http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>com.itheima</groupId>
9   <artifactId>maven_01_parent</artifactId>
10  <version>1.0-RELEASE</version>
11  <packaging>pom</packaging>
12
13  <!--设置管理的模块名称-->
14  <modules>
15    <module>../maven_02_ssm</module>
16    <module>../maven_03_pojo</module>
17    <module>../maven_04_dao</module>
18  </modules>
19  <dependencies>
20    <dependency>
21      <groupId>org.springframework</groupId>
22      <artifactId>spring-core</artifactId>
23      <version>5.2.10.RELEASE</version>
24    </dependency>
25
26    <dependency>
27      <groupId>org.springframework</groupId>
28      <artifactId>spring-webmvc</artifactId>
29      <version>5.2.10.RELEASE</version>
30    </dependency>
31
32    <dependency>
33      <groupId>org.springframework</groupId>
34      <artifactId>spring-jdbc</artifactId>
```

```
34      <version>5.2.10.RELEASE</version>
35    </dependency>
36
37    <dependency>
38      <groupId>org.springframework</groupId>
39      <artifactId>spring-test</artifactId>
40      <version>5.2.10.RELEASE</version>
41    </dependency>
42
43    <dependency>
44      <groupId>org.mybatis</groupId>
45      <artifactId>mybatis</artifactId>
46      <version>3.5.6</version>
47    </dependency>
48
49    <dependency>
50      <groupId>org.mybatis</groupId>
51      <artifactId>mybatis-spring</artifactId>
52      <version>1.3.0</version>
53    </dependency>
54
55    <dependency>
56      <groupId>mysql</groupId>
57      <artifactId>mysql-connector-java</artifactId>
58      <version>5.1.47</version>
59    </dependency>
60
61    <dependency>
62      <groupId>com.alibaba</groupId>
63      <artifactId>druid</artifactId>
64      <version>1.1.16</version>
65    </dependency>
66
67    <dependency>
68      <groupId>javax.servlet</groupId>
69      <artifactId>javax.servlet-api</artifactId>
70      <version>3.1.0</version>
71      <scope>provided</scope>
72    </dependency>
73
74    <dependency>
75      <groupId>com.fasterxml.jackson.core</groupId>
76      <artifactId>jackson-databind</artifactId>
77      <version>2.9.0</version>
78    </dependency>
79  </dependencies>
80 </project>
```

2. 删除子项目中已经被抽取到父项目的pom.xml中的jar包，如在maven_02_ssm的pom.xml中将已经出现在父项目的jar包删除掉

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6   http://maven.apache.org/xsd/maven-4.0.0.xsd">
7   <modelVersion>4.0.0</modelVersion>
8
9   <groupId>com.itheima</groupId>
10  <artifactId>maven_02_ssm</artifactId>
11  <version>1.0-SNAPSHOT</version>
12  <packaging>war</packaging>
13
14  <!--配置当前工程继承自parent工程-->
15  <parent>
16    <groupId>com.itheima</groupId>
17    <artifactId>maven_01_parent</artifactId>
18    <version>1.0-RELEASE</version>
19    <relativePath>../maven_01_parent/pom.xml</relativePath>
20  </parent>
21  <dependencies>
22    <dependency>
23      <groupId>junit</groupId>
24      <artifactId>junit</artifactId>
25      <version>4.12</version>
26      <scope>test</scope>
27    </dependency>
28    <dependency>
29      <groupId>junit</groupId>
30      <artifactId>junit</artifactId>
31      <version>4.11</version>
32      <scope>test</scope>
33    </dependency>
34    <dependency>
35      <groupId>com.itheima</groupId>
36      <artifactId>maven_04_dao</artifactId>
37      <version>1.0-SNAPSHOT</version>
38      <!--排除依赖是隐藏当前资源对应的依赖关系-->
39      <exclusions>
40        <exclusion>
41          <groupId>log4j</groupId>
42          <artifactId>log4j</artifactId>
```

```

43     </exclusion>
44     <exclusion>
45         <groupId>org.mybatis</groupId>
46         <artifactId>mybatis</artifactId>
47     </exclusion>
48   </exclusions>
49 </dependency>
50
51 </dependencies>
52
53 <build>
54   <plugins>
55     <plugin>
56       <groupId>org.apache.tomcat.maven</groupId>
57       <artifactId>tomcat7-maven-plugin</artifactId>
58       <version>2.1</version>
59       <configuration>
60         <port>80</port>
61         <path>/</path>
62       </configuration>
63     </plugin>
64   </plugins>
65 </build>
66 </project>
67
68

```

删除完后，你会发现父项目中有依赖对应的jar包，子项目虽然已经将重复的依赖删除掉了，但是刷新的时候，子项目中所需要的jar包依然存在。

当项目的`<parent>`标签被移除掉，会发现多出来的jar包依赖也会随之消失。

3. 将maven_04_dao项目的pom.xml中的所有依赖删除，然后添加上maven_01_parent的父项目坐标

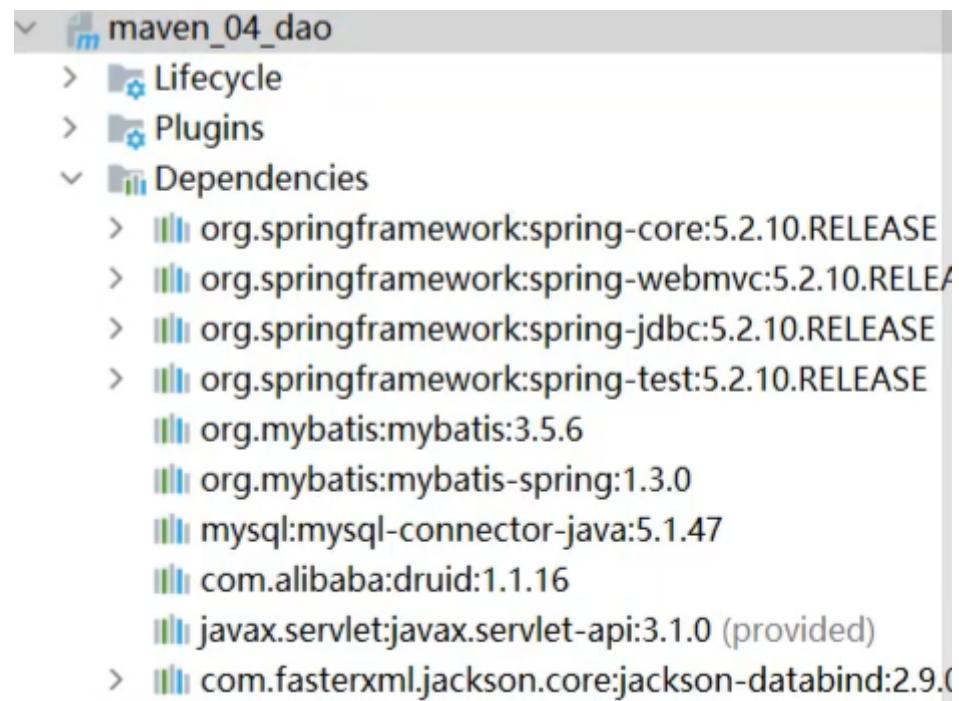
```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>com.itheima</groupId>
9   <artifactId>maven_04_dao</artifactId>
10  <version>1.0-SNAPSHOT</version>
11
12  <!--配置当前工程继承自parent工程-->
13  <parent>

```

```
13      <groupId>com.itheima</groupId>
14      <artifactId>maven_01_parent</artifactId>
15      <version>1.0-RELEASE</version>
16      <relativePath>../maven_01_parent/pom.xml</relativePath>
17  </parent>
18 </project>
```

刷新并查看Maven的面板，会发现maven_04_dao同样引入了父项目中的所有依赖。



这样我们就可以解决刚才提到的第一个问题，将子项目中的公共jar包抽取到父工程中进行统一添加依赖，这样做的可以简化配置，并且当父工程中所依赖的jar包版本发生变化，所有子项目中对应的jar包版本也会跟着更新。

```

16     <module>../maven_04_dao</module>
17   </modules>
18
19   <dependencies>
20
21     <dependency>
22       <groupId>org.springframework</groupId>
23       <artifactId>spring-core</artifactId>
24       <version>5.1.9.RELEASE</version>*
25     </dependency>
26
27     <dependency>
28       <groupId>org.springframework</groupId>
29       <artifactId>spring-webmvc</artifactId>
30       <version>5.2.10.RELEASE</version>
31     </dependency>
32
33     <dependency>
34       <groupId>org.springframework</groupId>
35       <artifactId>spring-jdbc</artifactId>
36       <version>5.2.10.RELEASE</version>
37     </dependency>
38
39     <dependency>
40       <groupId>org.springframework</groupId>
41       <artifactId>spring-test</artifactId>
42       <version>5.2.10.RELEASE</version>
43     </dependency>

```

Maven Dependencies tree:

- maven_01_parent (root)
 - Lifecycle
 - Plugins
 - Dependencies
 - org.springframework:spring-core:5.1.9.RELEASE
 - org.springframework:spring-webmvc:5.2.10.RELEASE
 - org.springframework:spring-jdbc:5.2.10.RELEASE
 - org.springframework:spring-test:5.2.10.RELEASE
 - org.mybatis:mybatis:3.5.6
 - org.mybatis:mybatis-spring:1.3.0
 - mysql:mysql-connector-java:5.1.47
 - com.alibaba:druid:1.1.16
 - javax.servlet:javax.servlet-api:3.1.0 (provided)
 - com.fasterxml.jackson.core:jackson-databind:2.9.0
- maven_02_ssm
 - Lifecycle
 - Plugins
 - Run Configurations
 - Dependencies
 - junit:junit:4.11 (test)
 - com.itheima:maven_04_dao:1.0-SNAPSHOT
 - org.springframework:spring-core:5.1.9.RELEASE
 - org.springframework:spring-webmvc:5.2.10.RELEASE
 - org.springframework:spring-jdbc:5.2.10.RELEASE
 - org.springframework:spring-test:5.2.10.RELEASE
 - org.mybatis:mybatis:3.5.6
 - org.mybatis:mybatis-spring:1.3.0
 - mysql:mysql-connector-java:5.1.47
 - com.alibaba:druid:1.1.16
 - javax.servlet:javax.servlet-api:3.1.0 (provided)
 - com.fasterxml.jackson.core:jackson-databind:2.9.0
- maven_03_pojo
 - Lifecycle
 - Plugins
 - Dependencies
 - org.springframework:spring-core:5.1.9.RELEASE
 - org.springframework:spring-webmvc:5.2.10.RELEASE
 - org.springframework:spring-jdbc:5.2.10.RELEASE
 - org.springframework:spring-test:5.2.10.RELEASE
 - org.mybatis:mybatis:3.5.6
 - org.mybatis:mybatis-spring:1.3.0
 - mysql:mysql-connector-java:5.1.47
 - com.alibaba:druid:1.1.16
 - javax.servlet:javax.servlet-api:3.1.0 (provided)
 - com.fasterxml.jackson.core:jackson-databind:2.9.0
- maven_04_dao
 - Lifecycle
 - Plugins
 - Dependencies
 - org.springframework:spring-core:5.1.9.RELEASE
 - org.springframework:spring-webmvc:5.2.10.RELEASE
 - org.springframework:spring-jdbc:5.2.10.RELEASE
 - org.springframework:spring-test:5.2.10.RELEASE
 - org.mybatis:mybatis:3.5.6
 - org.mybatis:mybatis-spring:1.3.0
 - mysql:mysql-connector-java:5.1.47
 - com.alibaba:druid:1.1.16
 - javax.servlet:javax.servlet-api:3.1.0 (provided)
 - com.fasterxml.jackson.core:jackson-databind:2.9.0

步骤4：优化子项目依赖版本问题

如果把所有用到的jar包都管理在父项目的pom.xml，看上去更简单些，但是这样就会导致有很多项目引入了过多自己不需要的jar包。如上面看到的这张图：

ssm.crm

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

```

ssm.goods

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

```

ssm.order

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.2.10.RELEASE</version>
</dependency>

<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.16</version>
</dependency>

```

如果把所有的依赖都放在了父工程中进行统一维护，就会导致ssm_order项目中多引入了spring-test的jar包，如果这样的jar包过多的话，对于ssm_order来说也是一种“负担”。

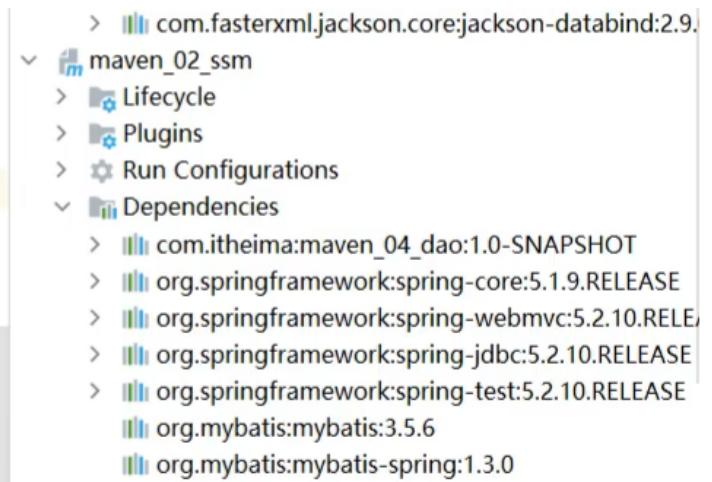
那针对这种部分项目有的jar包，我们该如何管理优化呢？

1. 在父工程mavne_01_parent的pom.xml来定义依赖管理

```
1 <!--定义依赖管理-->
2 <dependencyManagement>
3   <dependencies>
4     <dependency>
5       <groupId>junit</groupId>
6       <artifactId>junit</artifactId>
7       <version>4.12</version>
8       <scope>test</scope>
9     </dependency>
10    </dependencies>
11 </dependencyManagement>
```

2. 将maven_02_ssm的pom.xml中的junit依赖删除掉，刷新Maven

```
<!-- 定义依赖管理-->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```



刷新完会发现，在maven_02_ssm项目中的junit依赖并没有出现，所以我们得到一个结论：

<dependencyManagement>标签不真正引入jar包，而是配置可供子项目选择的jar包依赖

子项目要想使用它所提供的这些jar包，需要自己添加依赖，并且不需要指定<version>

3. 在maven_02_ssm的pom.xml添加junit的依赖

```
1 <dependency>
2   <groupId>junit</groupId>
3   <artifactId>junit</artifactId>
4   <scope>test</scope>
5 </dependency>
```

注意：这里就不需要添加版本了，这样做好处就是当父工程dependencyManagement标签中的版本发生变化后，子项目中的依赖版本也会跟着发生变化

4. 在maven_04_dao的pom.xml添加junit的依赖

```
1 <dependency>
2   <groupId>junit</groupId>
3   <artifactId>junit</artifactId>
4   <scope>test</scope>
5 </dependency>
```

这个时候，`maven_02_ssm`和`maven_04_dao`这两个项目中的`junit`版本就会跟随着父项目中的标签`dependencyManagement`中`junit`的版本发生变化而变化。不需要`junit`的项目就不需要添加对应的依赖即可。

至此继承就已经学习完了，总结来说，继承可以帮助做两件事

- 将所有项目公共的jar包依赖提取到父工程的`pom.xml`中，子项目就可以不用重复编写，简化开发
- 将所有项目的jar包配置到父工程的`dependencyManagement`标签下，实现版本管理，方便维护
 - **dependencyManagement标签不真正引入jar包，只是管理jar包的版本**
 - 子项目在引入的时候，只需要指定`groupId`和`artifactId`，不需要加`version`
 - 当`dependencyManagement`标签中jar包版本发生变化，所有子项目中有用到该jar包的地方对应的版本会自动随之更新

最后总结一句话就是，**父工程主要是用来快速配置依赖jar包和管理项目中所使用的资源。**

小结

继承的实现步骤：

- 创建Maven模块，设置打包类型为`pom`

```
1 <packaging>pom</packaging>
```

- 在父工程的`pom`文件中配置依赖关系（子工程将沿用父工程中的依赖关系），一般只抽取子项目中公有的jar包

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework</groupId>
4     <artifactId>spring-webmvc</artifactId>
5     <version>5.2.10.RELEASE</version>
6   </dependency>
7   ...
8 </dependencies>
```

- 在父工程中配置子工程中可选的依赖关系

```
1 <dependencyManagement>
2   <dependencies>
3     <dependency>
4       <groupId>com.alibaba</groupId>
5       <artifactId>druid</artifactId>
6       <version>1.1.16</version>
7     </dependency>
8   </dependencies>
9   ...
10 </dependencyManagement>
```

- 在子工程中配置当前工程所继承的父工程

```
1 <!--定义该工程的父工程-->
2 <parent>
3   <groupId>com.itheima</groupId>
4   <artifactId>maven_01_parent</artifactId>
5   <version>1.0-RELEASE</version>
6   <!--填写父工程的pom文件,可以不写-->
7   <relativePath>../maven_01_parent/pom.xml</relativePath>
8 </parent>
```

- 在子工程中配置使用父工程中可选依赖的坐标

```
1 <dependencies>
2   <dependency>
3     <groupId>com.alibaba</groupId>
4     <artifactId>druid</artifactId>
5   </dependency>
6 </dependencies>
```

注意事项：

- 子工程中使用父工程中的可选依赖时，仅需要提供群组id和项目id，无需提供版本，版本由父工程统一提供，避免版本冲突
- 子工程中还可以定义父工程中没有定义的依赖关系，只不过不能被父工程进行版本统一管理。

3.3 聚合与继承的区别

3.3.1 聚合与继承的区别

两种之间的作用：

- 聚合用于快速构建项目，对项目进行管理
- 继承用于快速配置和管理子项目中所使用jar包的版本

聚合和继承的相同点：

- 聚合与继承的pom.xml文件打包方式均为pom，可以将两种关系制作到同一个pom文件中
- 聚合与继承均属于设计型模块，并无实际的模块内容

聚合和继承的不同点：

- 聚合是在当前模块中配置关系，聚合可以感知到参与聚合的模块有哪些
- 继承是在子模块中配置关系，父模块无法感知哪些子模块继承了自己

相信到这里，大家已经能区分开什么是聚合和继承，但是有一个稍微麻烦的地方就是聚合和继承的工程构建，需要在聚合项目中手动添加modules标签，需要在所有的子项目中添加parent标签，万一写错了咋办？

3.3.2 IDEA构建聚合与继承工程

其实对于聚合和继承工程的创建，IDEA已经能帮助我们快速构建，具体的实现步骤为：

步骤1：创建一个Maven项目

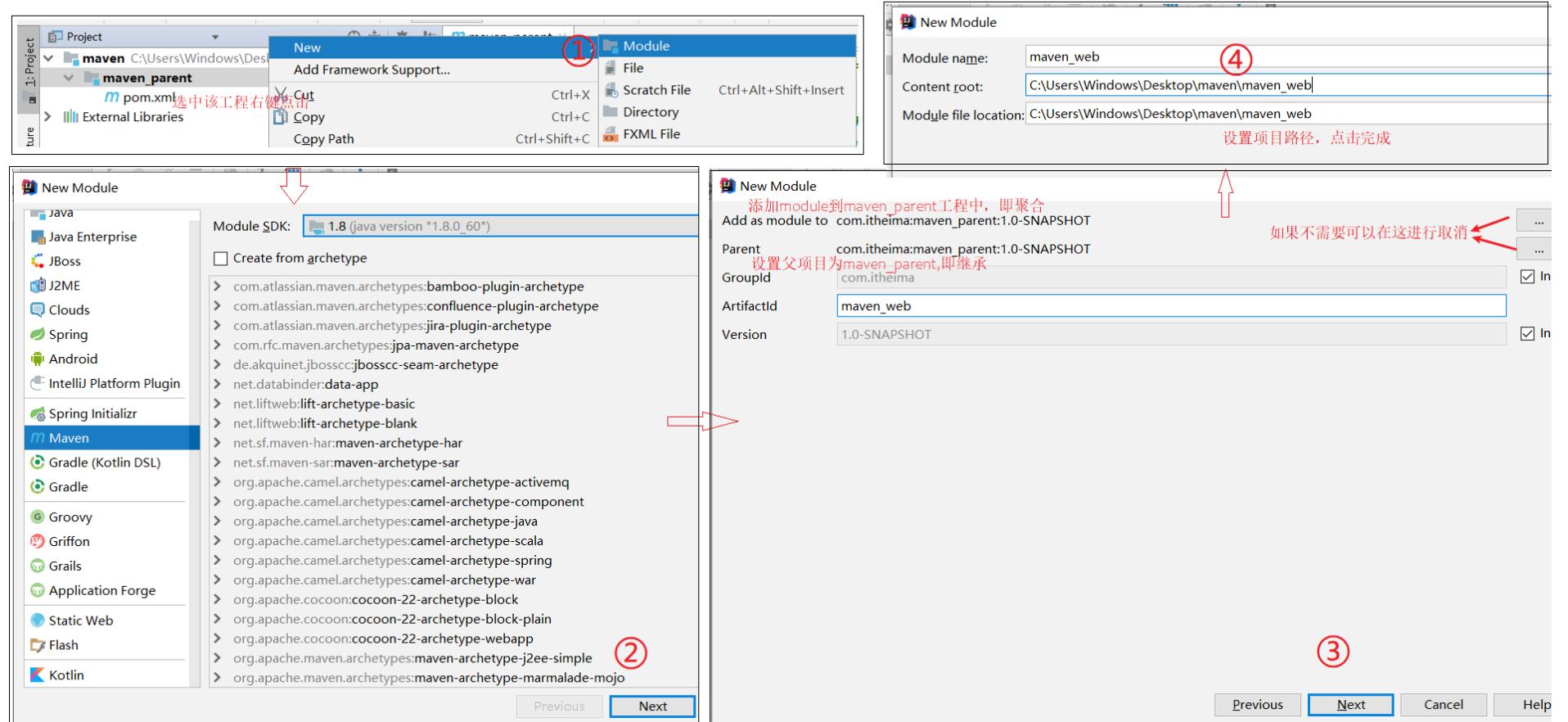
创建一个空的Maven项目，可以将项目中的src目录删除掉，这个项目作为聚合工程和父工程。

The screenshot shows the IntelliJ IDEA interface with a Maven project named 'maven' at the root. Inside it is a 'maven_parent' module containing a 'pom.xml' file. The code editor displays the XML configuration for this parent POM:

```
<?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>
    <groupId>com.itheima</groupId>
    <artifactId>maven_parent</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>
```

步骤2：创建子项目

该项目可以被聚合工程管理，同时会继承父工程。



创建成功后，maven_parent即是聚合工程又是父工程，maven_web中也有parent标签，继承的就是maven_parent，对于难以配置的内容都自动生成。

按照上面这种方式，大家就可以根据自己的需要来构建分模块项目。

4. 属性

在这一章节内容中，我们将学习两个内容，分别是

- 属性
- 版本管理

属性中会继续解决分模块开发项目存在的问题，版本管理主要是认识下当前主流的版本定义方式。

4.1 属性

4.1.1 问题分析

讲解内容之前，我们还是先来分析问题：

前面我们已经在父工程中的`dependencyManagement`标签中对项目中所使用的jar包版本进行了统一的管理，但是如果在标签中有如下的内容：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.9.RELEASE</version>
</dependency>
```

你会发现，如果我们现在想更新Spring的版本，你会发现我们依然需要更新多个jar包的版本，这样的话还是有可能出现漏改导致程序出问题，而且改起来也是比较麻烦。

问题清楚后，我们需要解决的话，就可以参考咱们java基础所学习的变量，声明一个变量，在其他地方使用该变量，当变量的值发生变化后，所有使用变量的地方，就会跟着修改，即：

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>【spring_version】</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>【spring_version】</version>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>【spring_version】</version>
</dependency>

```

```

String spring_version = "5.2.10.RELEASE";

```

4.1.2 解决步骤

步骤1：父工程中定义属性

```

1 <properties>
2     <spring.version>5.2.10.RELEASE</spring.version>
3     <junit.version>4.12</junit.version>
4     <mybatis-spring.version>1.3.0</mybatis-spring.version>
5 </properties>

```

步骤2：修改依赖的version

```

1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-core</artifactId>
4     <version>${spring.version}</version>
5 </dependency>
6 <dependency>
7     <groupId>org.springframework</groupId>
8     <artifactId>spring-webmvc</artifactId>
9     <version>${spring.version}</version>
10 </dependency>
11 <dependency>
12     <groupId>org.springframework</groupId>
13     <artifactId>spring-jdbc</artifactId>
14     <version>${spring.version}</version>
15 </dependency>

```

此时，我们只需要更新父工程中`properties`标签中所维护的jar包版本，所有子项目中的版本也就跟着更新。当然除了将spring相关版本进行维护，我们可以将其他的jar包版本也进行抽取，这样就可以对项目中所有jar包的版本进行统一维护，如：

```
1 <!--定义属性-->
2 <properties>
3   <spring.version>5.2.10.RELEASE</spring.version>
4   <junit.version>4.12</junit.version>
5   <mybatis-spring.version>1.3.0</mybatis-spring.version>
6 </properties>
```

4.2 配置文件加载属性

Maven中的属性我们已经介绍过了，现在也已经能够通过Maven来集中管理Maven中依赖jar包的版本。但是又有新的需求，就是想让Maven对于属性的管理范围能更大些，比如我们之前项目中的`jdbc.properties`，这个配置文件中的属性，能不能也来让Maven进行管理呢？

答案是肯定的，具体的实现步骤为：

步骤1：父工程定义属性

```
1 <properties>
2   <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
3 </properties>
```

步骤2：`jdbc.properties`文件中引用属性

在`jdbc.properties`，将`jdbc.url`的值直接获取Maven配置的属性

```
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=${jdbc.url}
3 jdbc.username=root
4 jdbc.password=root
```

步骤3：设置maven过滤文件范围

Maven在默认情况下是从当前项目的`src\main\resources`下读取文件进行打包。现在我们需要打包的资源文件是在`maven_02_ssm`下，需要我们通过配置来指定下具体的资源目录。

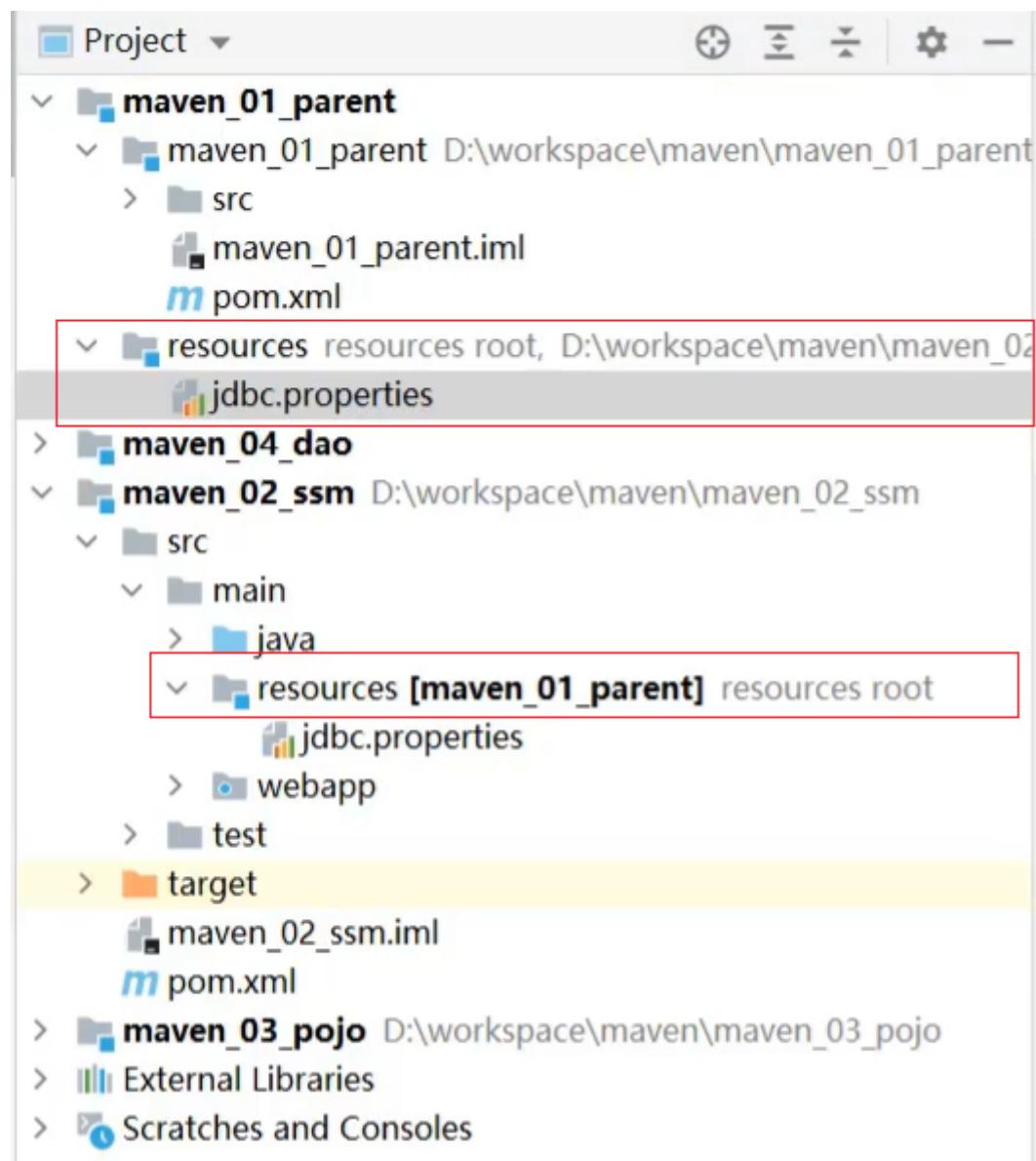
```

1 <build>
2   <resources>
3     <!--设置资源目录-->
4     <resource>
5       <directory>../maven_02_ssm/src/main/resources</directory>
6       <!--设置能够解析${}，默认是false -->
7       <filtering>true</filtering>
8     </resource>
9   </resources>
10 </build>

```

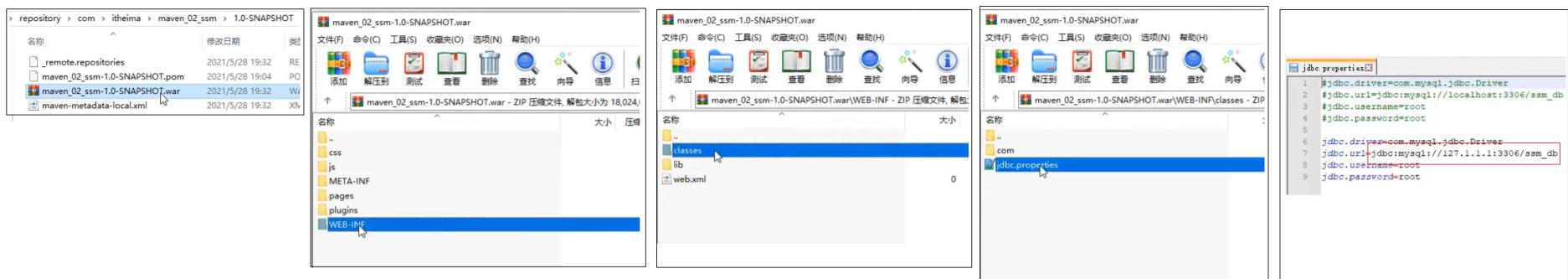
说明: directory路径前要添加`../`的原因是`maven_02_ssm`相对于父工程的`pom.xml`路径是在其上一层的目录中，所以需要添加。

修改完后，注意`maven_02_ssm`项目的`resources`目录就多了些东西，如下：



步骤4：测试是否生效

测试的时候，只需要将`maven_02_ssm`项目进行打包，然后观察打包结果中最终生成的内容是否为 Maven 中配置的内容。



上面的属性管理就已经完成，但是有一个问题没有解决，因为不只是maven_02_ssm项目需要有属性被父工程管理，如果有多个项目需要配置，该如何实现呢？

方式一：

```

1 <build>
2   <resources>
3     <!--设置资源目录，并设置能够解析${}-->
4     <resource>
5       <directory>../maven_02_ssm/src/main/resources</directory>
6       <filtering>true</filtering>
7     </resource>
8     <resource>
9       <directory>../maven_03_pojo/src/main/resources</directory>
10      <filtering>true</filtering>
11    </resource>
12    ...
13  </resources>
14 </build>

```

可以配，但是如果项目够多的话，这个配置也是比较繁琐

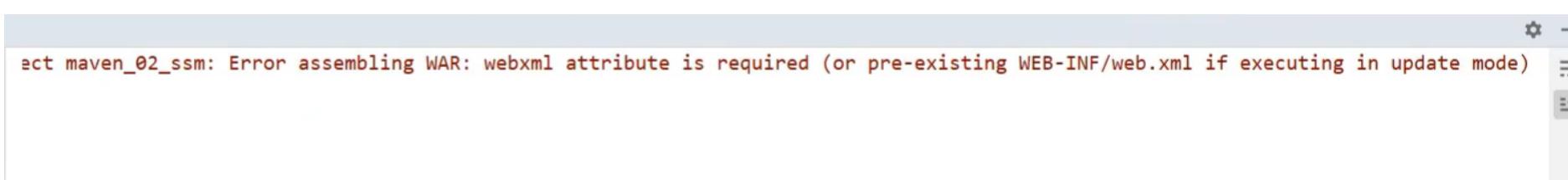
方式二：

```

1 <build>
2   <resources>
3     <!--
4       ${project.basedir}: 当前项目所在目录，子项目继承了父项目，
5       相当于所有的子项目都添加了资源目录的过滤
6     -->
7     <resource>
8       <directory>${project.basedir}/src/main/resources</directory>
9       <filtering>true</filtering>
10    </resource>
11  </resources>
12 </build>

```

说明：打包的过程中如果报如下错误：



原因就是Maven发现你的项目为web项目，就会去找web项目的入口web.xml [配置文件配置的方式]，发现没有找到，就会报错。

解决方案1：在maven_02_ssm项目的src\main\webapp\WEB-INF\添加一个web.xml文件

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
6     version="3.1">
7 </web-app>
```

解决方案2：配置maven打包war时，忽略web.xml检查

```
1 <build>
2     <plugins>
3         <plugin>
4             <groupId>org.apache.maven.plugins</groupId>
5             <artifactId>maven-war-plugin</artifactId>
6             <version>3.2.3</version>
7             <configuration>
8                 <failOnMissingwebXml>false</failOnMissingwebXml>
9             </configuration>
10            </plugin>
11        </plugins>
12    </build>
```

上面我们所使用的都是Maven的自定义属性，除了\${project.basedir}，它属于Maven的内置系统属性。

在Maven中的属性分为：

- 自定义属性（常用）
- 内置属性
- Setting属性
- Java系统属性
- 环境变量属性

属性分类	引用格式	示例
自定义属性	\${自定义属性名}	\${spring.version}
内置属性	\${内置属性名}	\${basedir} \${version}
Setting属性	\${setting.属性名}	\${settings.localRepository}
Java系统属性	\${系统属性分类.系统属性名}	\${user.home}
环境变量属性	\${env.环境变量属性名}	\${env.JAVA_HOME}

具体如何查看这些属性：

在cmd命令行中输入 `mvn help:system`

```
C:\Users\Windows>mvn help:system
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO] --- maven-help-plugin:3.2.0:system (default-cli) @ standalone-p
[INFO]
=====
===== Platform Properties Details =====
=====

System Properties
=====

java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jdk1.8.0_60\jre\bin
java.vm.version=25.60-b23
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator;
guice.disable.misplaced.annotation.check=true
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
file.encoding.pkg=sun.io
user.script=
user.country=CN
sun.java.launcher=SUN_STANDARD
sun.os.patch.level=
java.vm.specification.name=Java Virtual Machine Specification
user.dir=C:\Users\Windows
java.runtime.version=1.8.0_60-b27
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\Program Files\Java\jdk1.8.0_60\jre\lib\endorsed
os.arch=amd64
```

具体使用，就是使用 `${key}` 来获取， key 为等号左边的，值为等号右边的，比如获取红线的值，对应的写法为 `${java.runtime.name}`。

4.3 版本管理

关于这个版本管理解决的问题是，在 Maven 创建项目和引用别人项目的时候，我们都看到过如下内容：

```
<groupId>com.itheima</groupId>
<artifactId>spring_demo</artifactId>
<version>1.0-SNAPSHOT</version>
```

```
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.2.10.RELEASE</version>
```

这里面有两个单词， `SNAPSHOT` 和 `RELEASE`，它们所代表的含义是什么呢？

我们打开 Maven 仓库地址 <https://mvnrepository.com/>

spring-context	junit	mysql-jdbc
5.2.1.RELEASE	4.13-rc-2	6.0.6
5.2.0.RELEASE	4.13-rc-1	6.0.5
5.1.20.RELEASE	4.13-beta-3	6.0.x
5.1.19.RELEASE	4.13-beta-2	6.0.4
5.1.18.RELEASE	4.13-beta-1	6.0.3
5.1.17.RELEASE	4.12	6.0.2
5.1.16.RELEASE	4.12-beta-3	5.1.49
5.1.15.RELEASE	4.12-beta-2	5.1.48
5.1.14.RELEASE	4.12-beta-1	5.1.47
5.1.13.RELEASE	4.11	5.1.46
5.1.12.RELEASE	4.11-beta-1	5.1.45
5.1.11.RELEASE	4.10	5.1.44
5.1.x	4.9	5.1.43
5.1.10.RELEASE	4.8.2	5.1.42
5.1.9.RELEASE	4.8.1	5.1.41
5.1.8.RELEASE		5.1.40

在我们jar包的版本定义中，有两个工程版本用的比较多：

- SNAPSHOT (快照版本)
 - 项目开发过程中临时输出的版本，称为快照版本
 - 快照版本会随着开发的进展不断更新
- RELEASE (发布版本)
 - 项目开发到一定阶段里程碑后，向团队外部发布较为稳定的版本，这种版本所对应的构件文件是稳定的
 - 即便进行功能的后续开发，也不会改变当前发布版本内容，这种版本称为发布版本

除了上面的工程版本，我们还经常能看到一些发布版本：

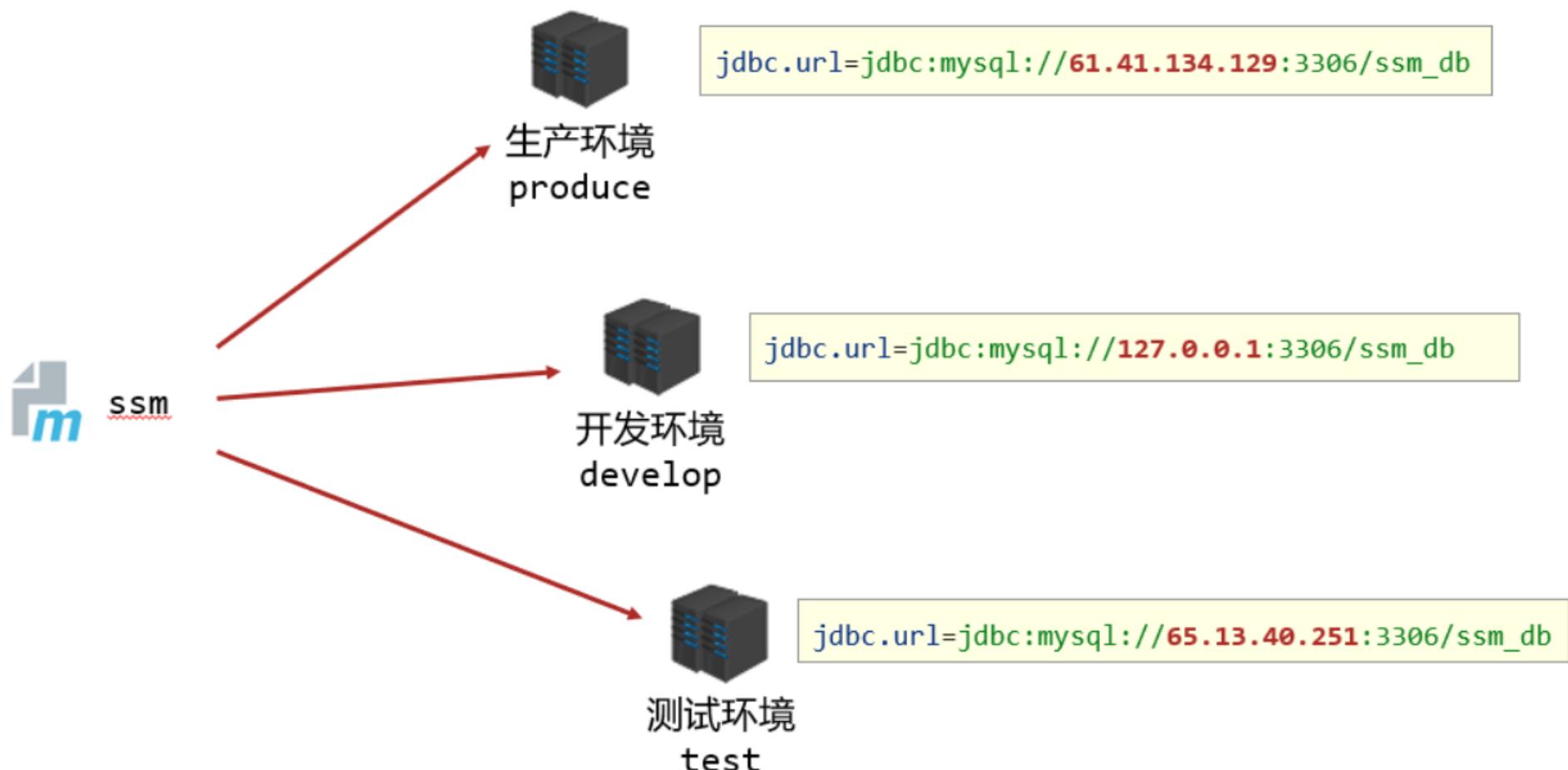
- alpha版：内测版，bug多不稳定内部版本不断添加新功能
- beta版：公测版，不稳定(比alpha稳定些)，bug相对较多不断添加新功能
- 纯数字版

对于这些版本，大家只需要简单认识下即可。

5，多环境配置与应用

这一节中，我们会讲两个内容，分别是多环境开发和跳过测试

5.1 多环境开发



- 我们平常都是在自己的开发环境进行开发，
- 当开发完成后，需要把开发的功能部署到测试环境供测试人员进行测试使用，
- 等测试人员测试通过后，我们会将项目部署到生成环境上线使用。
- 这个时候就有一个问题是，不同环境的配置是不相同的，如不可能让三个环境都用一个数据库，所以就会有三个数据库的url配置，
- 我们在项目中如何配置？
- 要想实现不同环境之间的配置切换又该如何来实现呢？

maven提供配置多种环境的设定，帮助开发者在使用过程中快速切换环境。具体实现步骤：

步骤1：父工程配置多个环境，并指定默认激活环境

```

1 <profiles>
2   <!--开发环境-->
3   <profile>
4     <id>env_dep</id>
5     <properties>
6       <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
7     </properties>
8     <!--设定是否为默认启动环境-->
9     <activation>
10       <activeByDefault>true</activeByDefault>
11     </activation>
12   </profile>
13   <!--生产环境-->
14   <profile>
15     <id>env_pro</id>
16     <properties>
17       <jdbc.url>jdbc:mysql://127.2.2.2:3306/ssm_db</jdbc.url>
18     </properties>
19   </profile>

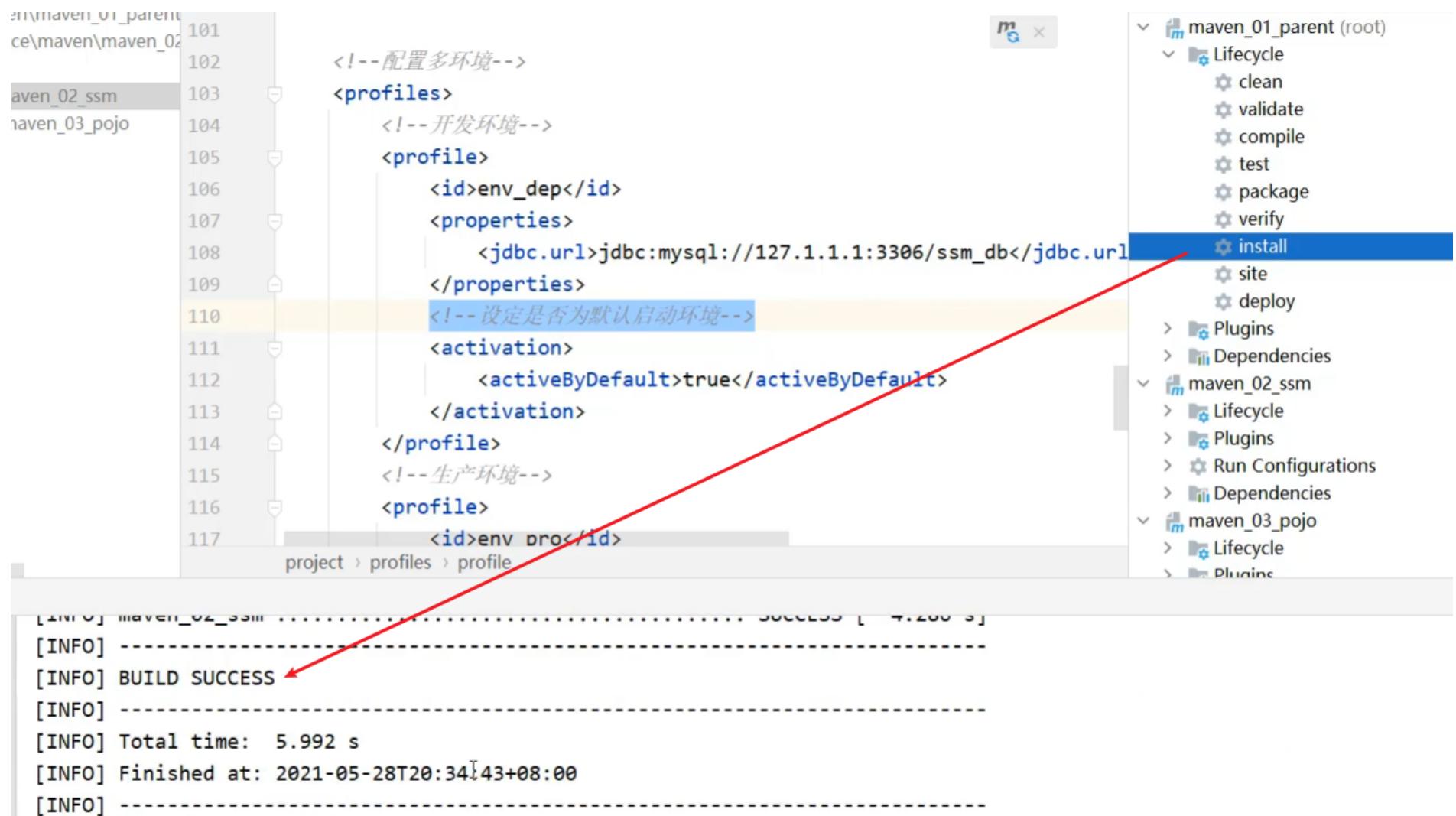
```

```

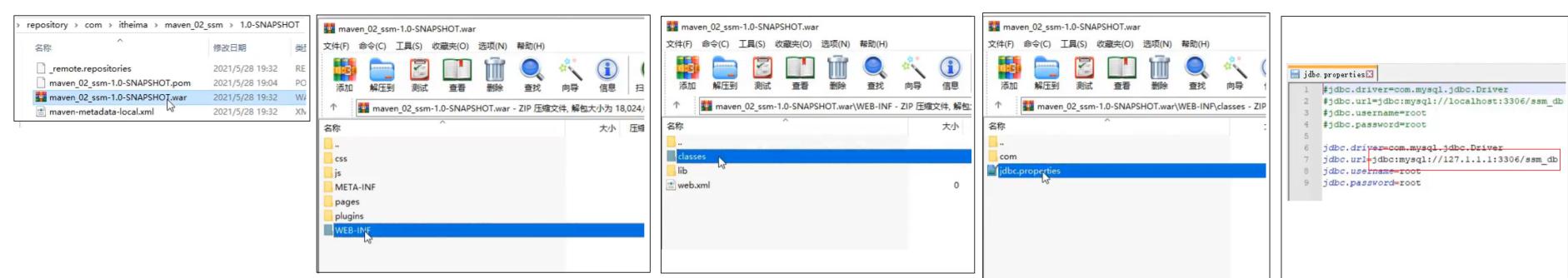
20 <!--测试环境-->
21 <profile>
22   <id>env_test</id>
23   <properties>
24     <jdbc.url>jdbc:mysql://127.3.3.3:3306/ssm_db</jdbc.url>
25   </properties>
26 </profile>
27 </profiles>

```

步骤2：执行安装查看env_dep环境是否生效



查看到的结果为：



步骤3：切换默认环境为生产环境

```

1 <profiles>
2   <!--开发环境-->
3   <profile>
4     <id>env_dep</id>
5     <properties>
6       <jdbc.url>jdbc:mysql://127.1.1.1:3306/ssm_db</jdbc.url>
7     </properties>
8   </profile>

```

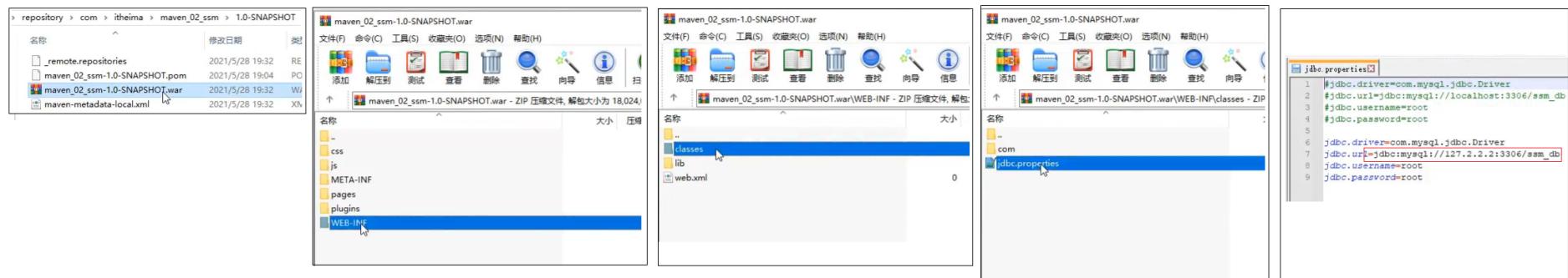
```

9   <!--生产环境-->
10  <profile>
11    <id>env_pro</id>
12    <properties>
13      <jdbc.url>jdbc:mysql://127.2.2.2:3306/ssm_db</jdbc.url>
14    </properties>
15    <!--设定是否为默认启动环境-->
16    <activation>
17      <activeByDefault>true</activeByDefault>
18    </activation>
19  </profile>
20  <!--测试环境-->
21  <profile>
22    <id>env_test</id>
23    <properties>
24      <jdbc.url>jdbc:mysql://127.3.3.3:3306/ssm_db</jdbc.url>
25    </properties>
26  </profile>
27 </profiles>

```

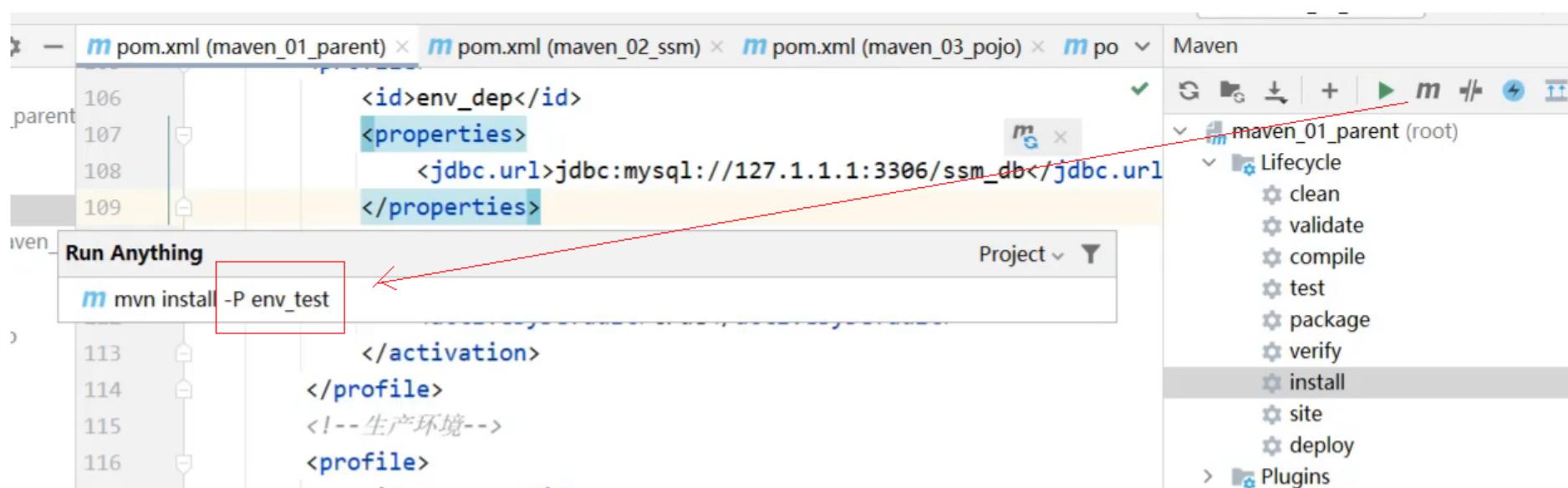
步骤4：执行安装并查看env_pro环境是否生效

查看到的结果为 `jdbc:mysql://127.2.2.2:3306/ssm_db`



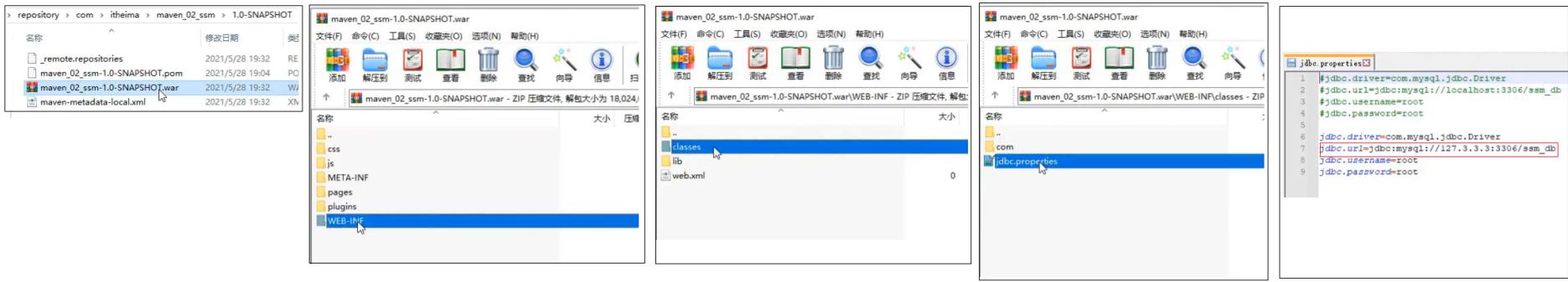
虽然已经能够实现不同环境的切换，但是每次切换都是需要手动修改，如何来实现在不改变代码的前提下完成环境的切换呢？

步骤5：命令行实现环境切换



步骤6：执行安装并查看env_test环境是否生效

查看到的结果为 `jdbc:mysql://127.3.3.3:3306/ssm_db`



所以总结来说，对于多环境切换只需要两步即可：

- 父工程中定义多环境

```
1 <profiles>
2   <profile>
3     <id>环境名称</id>
4     <properties>
5       <key>value</key>
6     </properties>
7     <activation>
8       <activeByDefault>true</activeByDefault>
9     </activation>
10    </profile>
11    ...
12 </profiles>
```

- 使用多环境(构建过程)

```
1 mvn 指令 -P 环境定义ID[环境定义中获取]
```

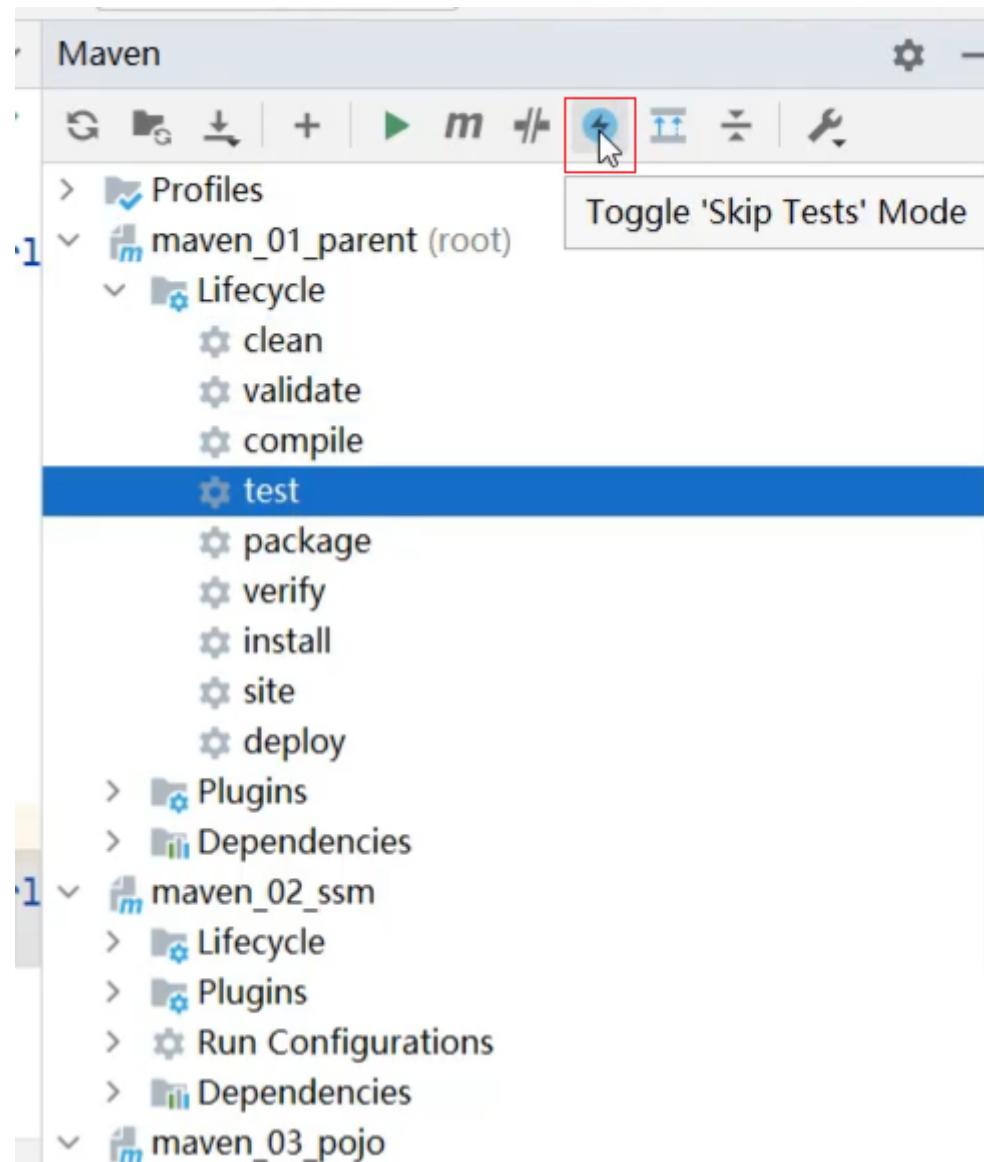
5.2 跳过测试

前面在执行 `install` 指令的时候，Maven 都会按照顺序从上往下依次执行，每次都会执行 `test`，对于 `test` 来说有它存在的意义，

- 可以确保每次打包或者安装的时候，程序的正确性，假如测试已经通过在我们没有修改程序的前提下再次执行打包或安装命令，由于顺序执行，测试会被再次执行，就有点耗费时间了。
- 功能开发过程中有部分模块还没有开发完毕，测试无法通过，但是想要把其中某一部分进行快速打包，此时由于测试环境失败就会导致打包失败。

遇到上面这些情况的时候，我们就想跳过测试执行下面的构建命令，具体实现方式有很多：

方式一：IDEA工具实现跳过测试



图中的按钮为 `Toggle 'Skip Tests' Mode`,

`Toggle` 翻译为切换的意思，也就是说在测试与不测试之间进行切换。

点击一下，出现测试画横线的图片，如下：



说明测试已经被关闭，再次点击就会恢复。

这种方式最简单，但是有点“暴力”，会把所有的测试都跳过，如果我们想更精细的控制哪些跳过哪些不跳过，就需要使用配置插件的方式。

方式二：配置插件实现跳过测试

在父工程中的 `pom.xml` 中添加测试插件配置

```
1 <build>
2   <plugins>
3     <plugin>
4       <artifactId>maven-surefire-plugin</artifactId>
5       <version>2.12.4</version>
6       <configuration>
7         <skipTests>false</skipTests>
8         <!--排除掉不参与测试的内容-->
9         <excludes>
10           <exclude>**/BookServiceTest.java</exclude>
11         </excludes>
12       </configuration>
13     </plugin>
```

```
14    </plugins>
15 </build>
```

skipTests: 如果为true, 则跳过所有测试, 如果为false, 则不跳过测试

excludes: 哪些测试类不参与测试, 即排除, 针对skipTests为false来设置的

includes: 哪些测试类要参与测试, 即包含, 针对skipTests为true来设置的

方式三: 命令行跳过测试



使用Maven的命令行, mvn 指令 -D skipTests

注意事项:

- 执行的项目构建指令必须包含测试生命周期, 否则无效果。例如执行compile生命周期, 不经过test生命周期。
- 该命令可以不借助IDEA, 直接使用cmd命令行进行跳过测试, 需要注意的是cmd要在pom.xml所在目录下进行执行。

6, 私服

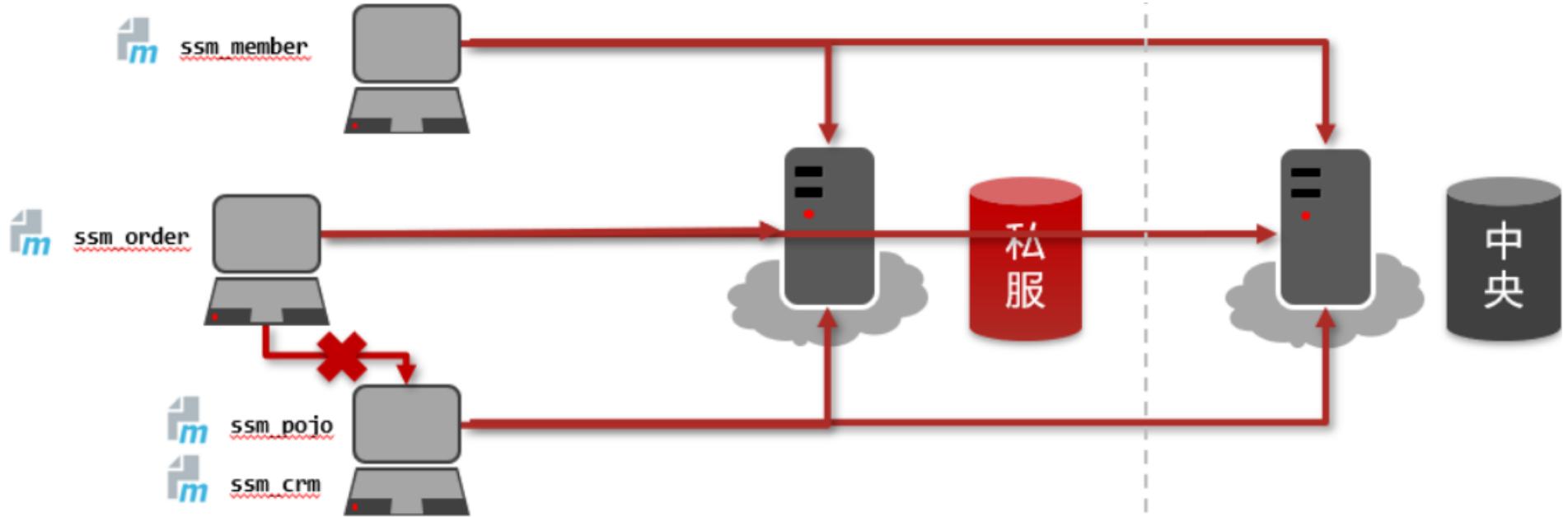
这一节, 我们主要学习的内容是:

- 私服简介
- 私服仓库分类
- 资源上传与下载

首先来说一说什么是私服?

6.1 私服简介

团队开发现状分析



(1) 张三负责ssm_crm的开发，自己写了一个ssm_pojo模块，要想使用直接将ssm_pojo安装到本地仓库即可

(2) 李四负责ssm_order的开发，需要用到张三所写的ssm_pojo模块，这个时候如何将张三写的ssm_pojo模块交给李四呢？

(3) 如果直接拷贝，那么团队之间的jar包管理会非常混乱而且容器出错，这个时候我们就能不能将写好的项目上传到中央仓库，谁想用就直接联网下载即可

(4) Maven的中央仓库不允许私人上传自己的jar包，那么我们就得换种思路，自己搭建一个类似于中央仓库的东西，把自己的内容上传上去，其他人就可以从上面下载jar包使用

(5) 这个类似于中央仓库的东西就是我们接下来要学习的**私服**

所以到这就有两个概念，一个是私服，一个是中央仓库

私服：公司内部搭建的用于存储Maven资源的服务器

远程仓库：Maven开发团队维护的用于存储Maven资源的服务器

所以说：

- 私服是一台独立的服务器，用于解决团队内部的资源共享与资源同步问题

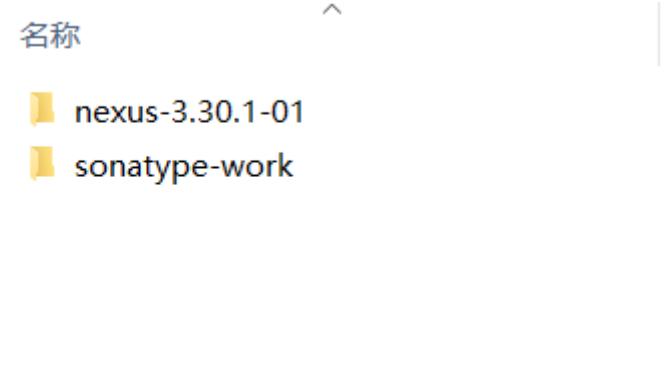
搭建Maven私服的方式有很多，我们来介绍其中一种使用量比较大的实现方式：

- Nexus
 - Sonatype公司的一款maven私服产品
 - 下载地址：<https://help.sonatype.com/repomanager3/download>

6.2 私服安装

步骤1：下载解压

将资料\latest-win64.zip解压到一个空目录下。



步骤2：启动Nexus

nexus > nexus-3.30.1-01 > bin



使用cmd进入到解压目录下的nexus-3.30.1-01\bin, 执行如下命令：

```
1 nexus.exe /run nexus
```

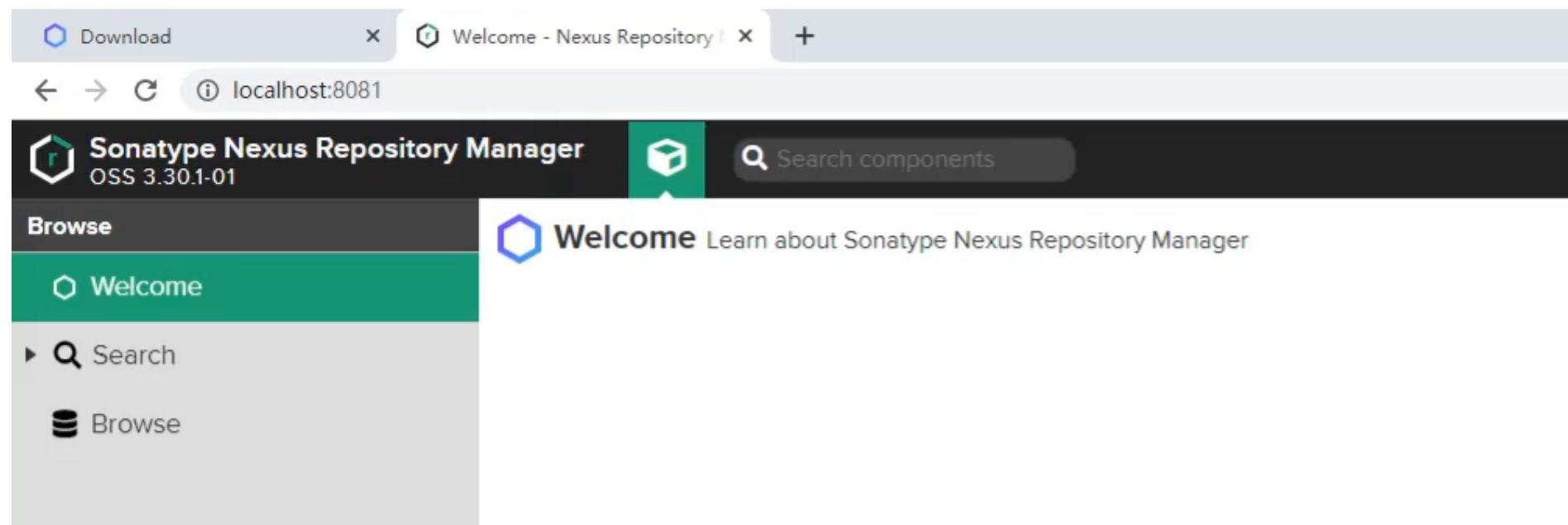
看到如下内容，说明启动成功。

```
Started Sonatype Nexus OSS 3.30.1-01
```

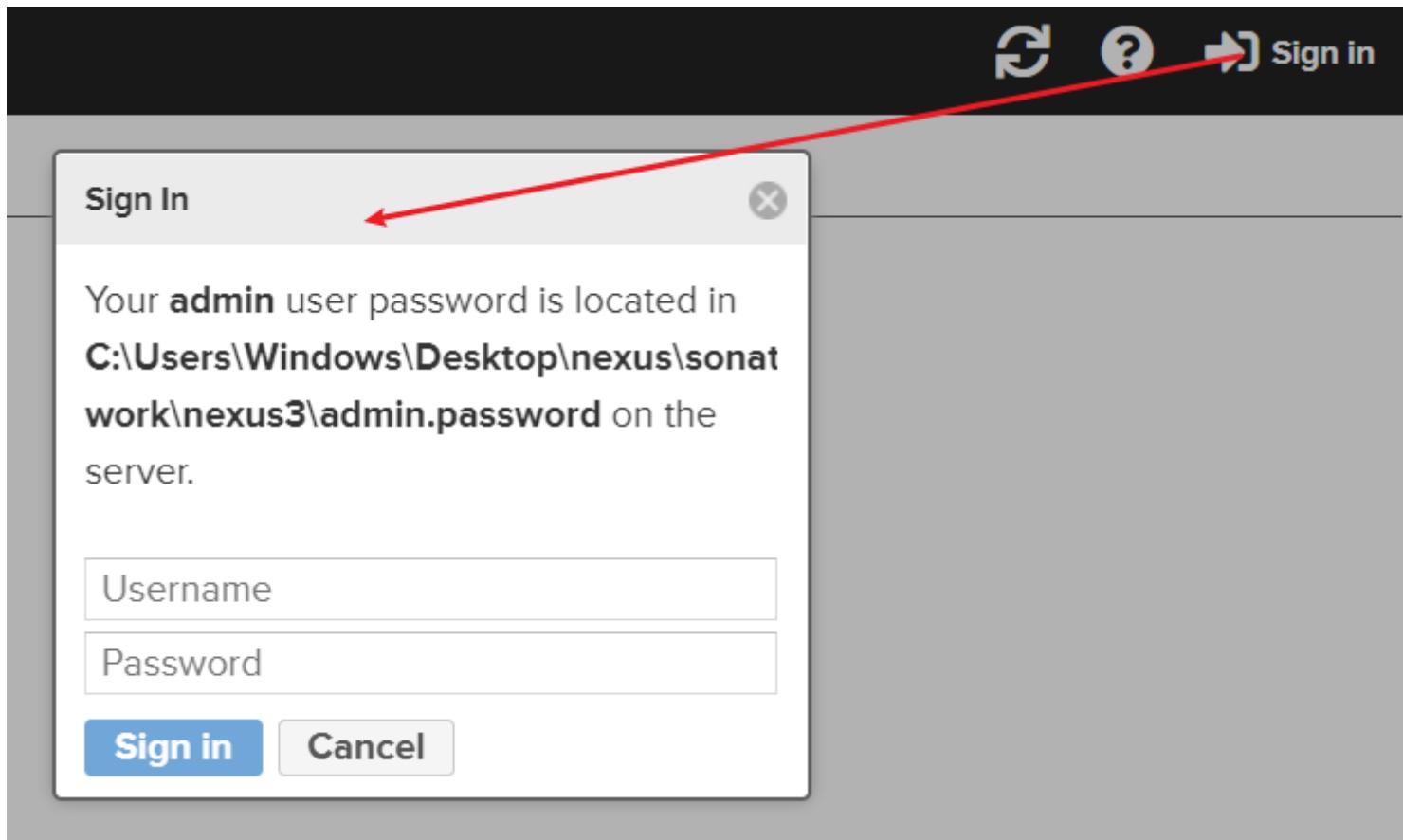
步骤3：浏览器访问

访问地址为：

```
1 http://localhost:8081
```



步骤4：首次登录重置密码



输入用户名和密码进行登录，登录成功后，出现如下页面

Setup 1 of 4

This wizard will help you complete required setup tasks.

Next

点击下一步，需要重新输入新密码，为了和后面的保持一致，密码修改为 `admin`

Please choose a password for the admin user 2 of 4

New password:
.....

Confirm password:
.....

Back **Next**

设置是否运行匿名访问

Configure Anonymous Access

3 of 4

Enable anonymous access means that by default, users can search, browse and download components from repositories without credentials. Please **consider the security implications for your organization**.

Disable anonymous access should be chosen with care, as it **will require credentials for all** users and/or build tools.

[More information](#)

- Enable anonymous access
- Disable anonymous access

[Back](#)

[Next](#)

点击完成

Complete

4 of 4

The setup tasks have been completed, enjoy using Nexus Repository Manager!

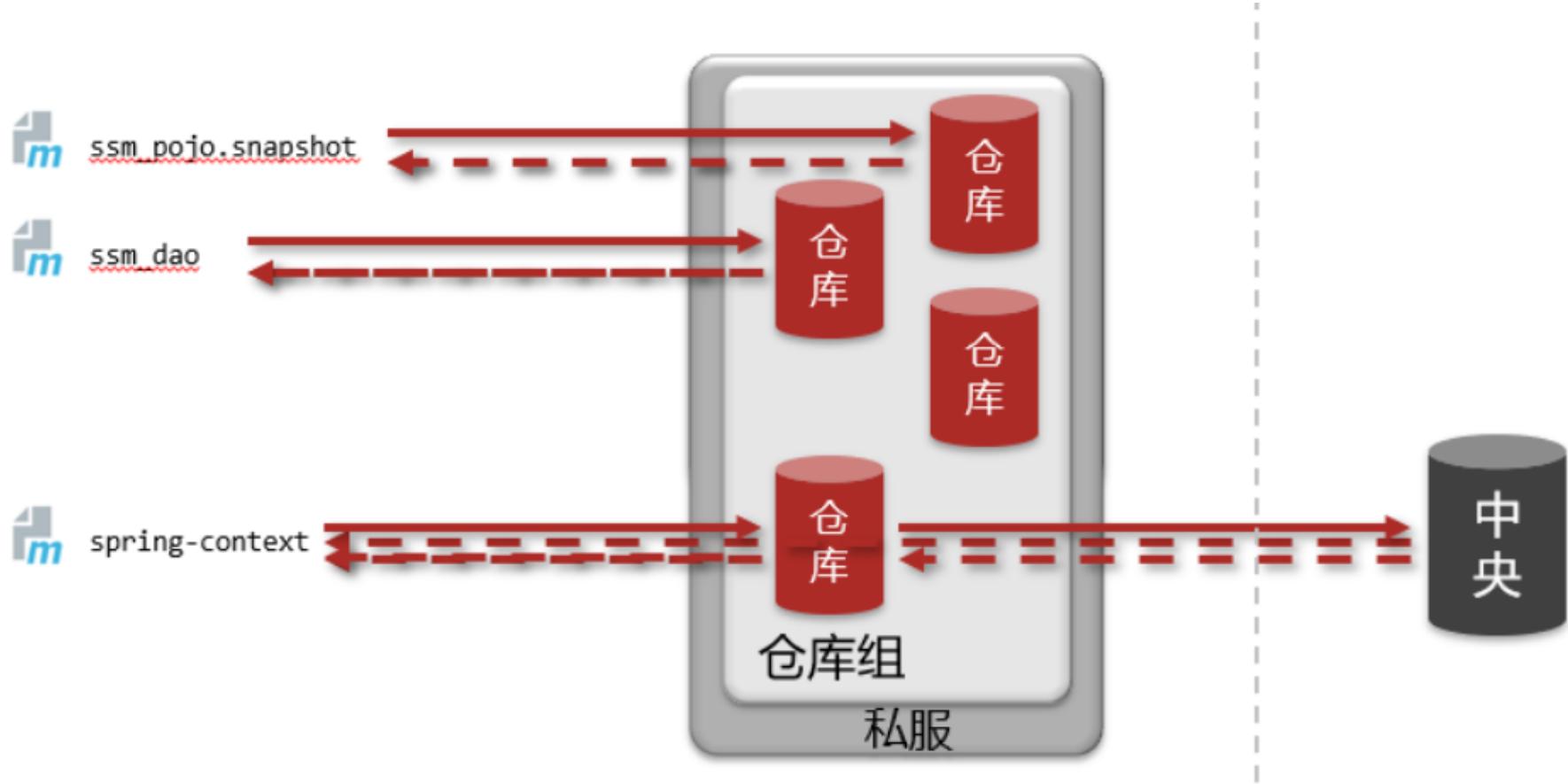
[Finish](#)

至此私服就已经安装成功。如果要想修改一些基础配置信息，可以使用：

- 修改基础配置信息
 - 安装路径下etc目录中nexus-default.properties文件保存有nexus基础配置信息，例如默认访问端口。
- 修改服务器运行配置信息
 - 安装路径下bin目录中nexus.vmoptions文件保存有nexus服务器启动对应的配置信息，例如默认占用内存空间。

6 . 3 私服仓库分类

私服资源操作流程分析：



- (1) 在没有私服的情况下，我们自己创建的服务都是安装在Maven的本地仓库中
- (2) 私服中也有仓库，我们要把自己的资源上传到私服，最终也是放在私服的仓库中
- (3) 其他人要想使用你所上传的资源，就需要从私服的仓库中获取
- (4) 当我们要使用的资源不是自己写的，是远程中央仓库有的第三方jar包，这个时候就需要从远程中央仓库下载，每个开发者都去远程中央仓库下速度比较慢（中央仓库服务器在国外）
- (5) 私服就再准备一个仓库，用来专门存储从远程中央仓库下载的第三方jar包，第一次访问没有就会去远程中央仓库下载，下次再访问就直接走私服下载
- (6) 前面在介绍版本管理的时候提到过有 `SNAPSHOT` 和 `RELEASE`，如果把这两类的都放到同一个仓库，比较混乱，所以私服就把这两个种jar包放入不同的仓库
- (7) 上面我们已经介绍了有三种仓库，一种是存放 `SNAPSHOT` 的，一种是存放 `RELEASE` 还有一种是存放从远程仓库下载的第三方jar包，那么我们在获取资源的时候要从哪个仓库种获取呢？
- (8) 为了方便获取，我们将所有的仓库编成一个组，我们只需要访问仓库组去获取资源。

所有私服仓库总共分为三大类：

宿主仓库hosted

- 保存无法从中央仓库获取的资源
 - 自主研发
 - 第三方非开源项目，比如Oracle，因为是付费产品，所以中央仓库没有

代理仓库proxy

- 代理远程仓库，通过nexus访问其他公共仓库，例如中央仓库

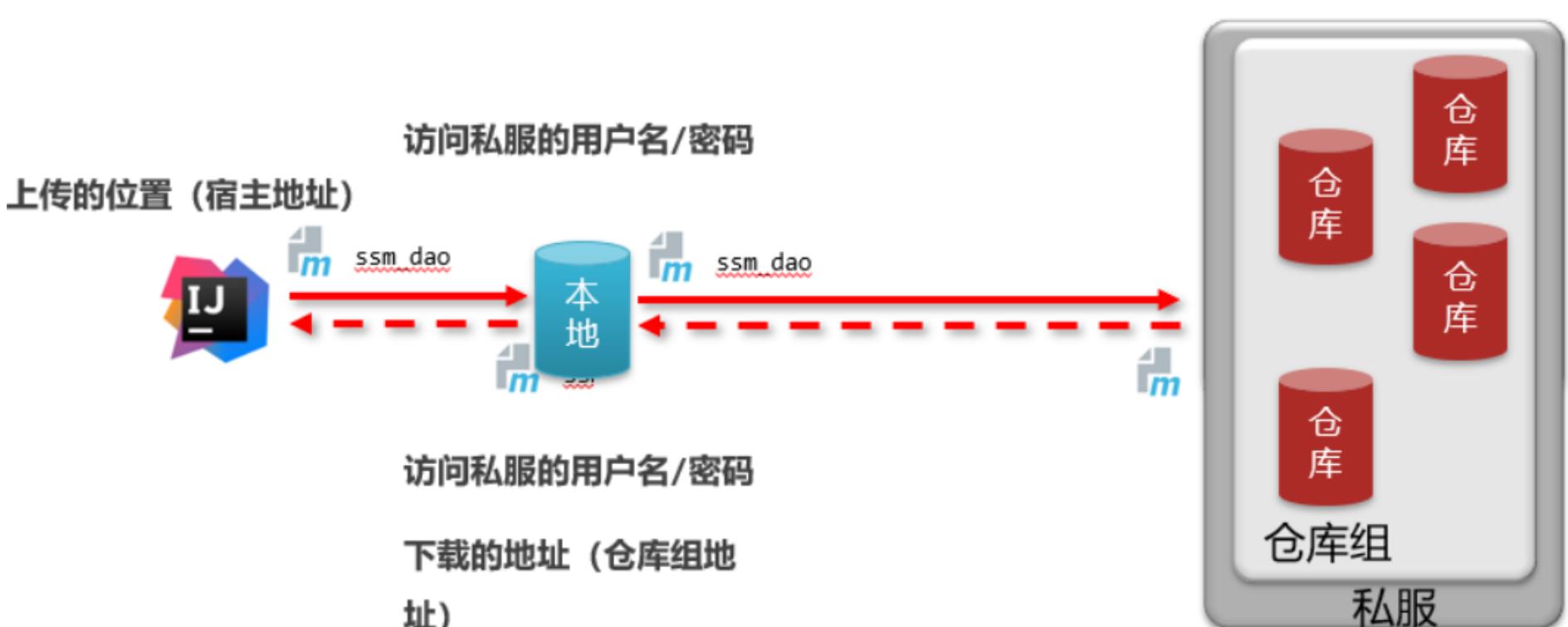
仓库组group

- 将若干个仓库组成一个群组，简化配置
- 仓库组不能保存资源，属于设计型仓库

仓库类别	英文名称	功能	关联操作
宿主仓库	hosted	保存自主研发+第三方资源	上传
代理仓库	proxy	代理连接中央仓库	下载
仓库组	group	为仓库编组简化下载操作	下载

6.4 本地仓库访问私服配置

- 我们通过IDEA将开发的模块上传到私服，中间是要经过本地Maven的
- 本地Maven需要知道私服的访问地址以及私服访问的用户名和密码
- 私服中的仓库很多，Maven最终要把资源上传到哪个仓库？
- Maven下载的时候，又需要携带用户名和密码到私服上找对应的仓库组进行下载，然后再给IDEA



上面所说的这些内容，我们需要在本地Maven的配置文件 `settings.xml` 中进行配置。

步骤1：私服上配置仓库

The screenshot shows the Sonatype Nexus Repository Manager interface with the following steps:

- Step 1: Click on "Create repository" (①).
- Step 2: Enter the name "itheima-snapshot" (②).
- Step 3: Select the "Snapshot" version policy (③).
- Step 4: Set the layout policy to "Strict" (④).
- Step 5: Set the blob store to "default" (⑤).
- Step 6: Click "Create repository" (⑥).
- Step 7: Click "Create repository" (⑦).
- Step 8: Click "Create repository" (⑧).

说明：

第5, 6步骤是创建itheima-snapshot仓库

第7, 8步骤是创建itheima-release仓库

步骤2：配置本地Maven对私服的访问权限

```
1 <servers>
2   <server>
3     <id>itheima-snapshot</id>
4     <username>admin</username>
5     <password>admin</password>
6   </server>
7   <server>
8     <id>itheima-release</id>
9     <username>admin</username>
10    <password>admin</password>
11  </server>
12 </servers>
```

步骤3：配置私服的访问路径

```
1 <mirrors>
2   <mirror>
3     <!--配置仓库组的ID-->
4     <id>maven-public</id>
5     <!--*代表所有内容都从私服获取-->
6     <mirrorof>*</mirrorof>
7     <!--私服仓库组maven-public的访问路径-->
8     <url>http://localhost:8081/repository/maven-public/</url>
9   </mirror>
10 </mirrors>
```

为了避免阿里云Maven私服地址的影响，建议先将之前配置的阿里云Maven私服镜像地址注释掉，等练习完后，再将其恢复。

Name: maven-public

Format: maven2

Type: group 私服仓库组访问地址

URL: http://localhost:8081/repository/maven-public/

Online: If checked, the repository accepts incoming requests

Storage

Blob store: default

Strict Content Type Validation:

Validate that all content uploaded to this repository is of a MIME type appropriate for the repository format

Group

Member repositories:

Select and order the repositories that are part of this group

Available	Members
Filter	maven-releases maven-snapshots maven-central itheima-release itheima-snapshot

需要将新加的两个仓库添加成仓库组的成员

Save **Discard**

至此本地仓库就能与私服进行交互了。

6.5 私服资源上传与下载

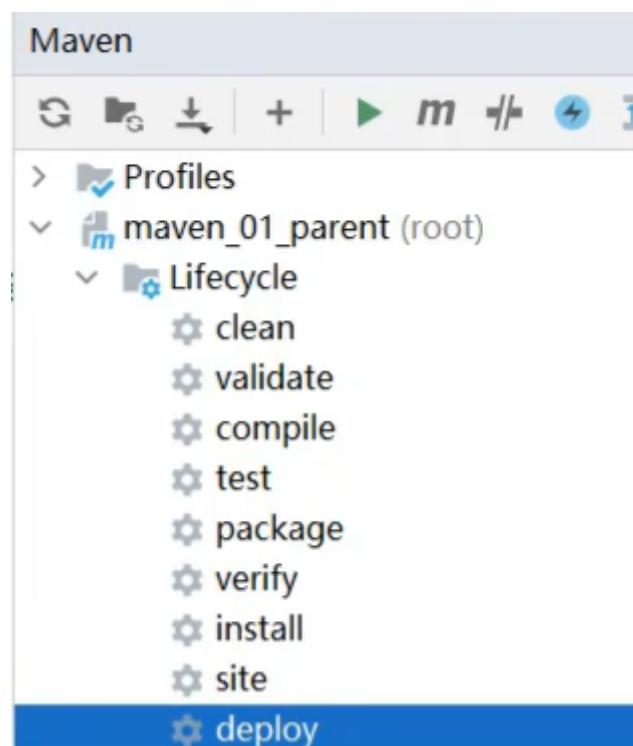
本地仓库与私服已经建立了连接，接下来我们就需要往私服上上传资源和下载资源，具体的实现步骤为：

步骤1：配置工程上传私服的具体位置

```
1 <!--配置当前工程保存在私服中的具体位置-->
2 <distributionManagement>
3   <repository>
4     <!--和maven/settings.xml中server中的id一致，表示使用该id对应的用户名和密码-->
5     <id>itheima-release</id>
6     <!--release版本上传仓库的具体地址-->
7     <url>http://localhost:8081/repository/itheima-release/</url>
```

```
8   </repository>
9   <snapshotRepository>
10    <!--和maven/settings.xml中server中的id一致，表示使用该id对应的用户名和密码-->
11    <id>itheima-snapshot</id>
12    <!--snapshot版本上传仓库的具体地址-->
13    <url>http://localhost:8081/repository/itheima-snapshot/</url>
14  </snapshotRepository>
15 </distributionManagement>
```

步骤2：发布资源到私服



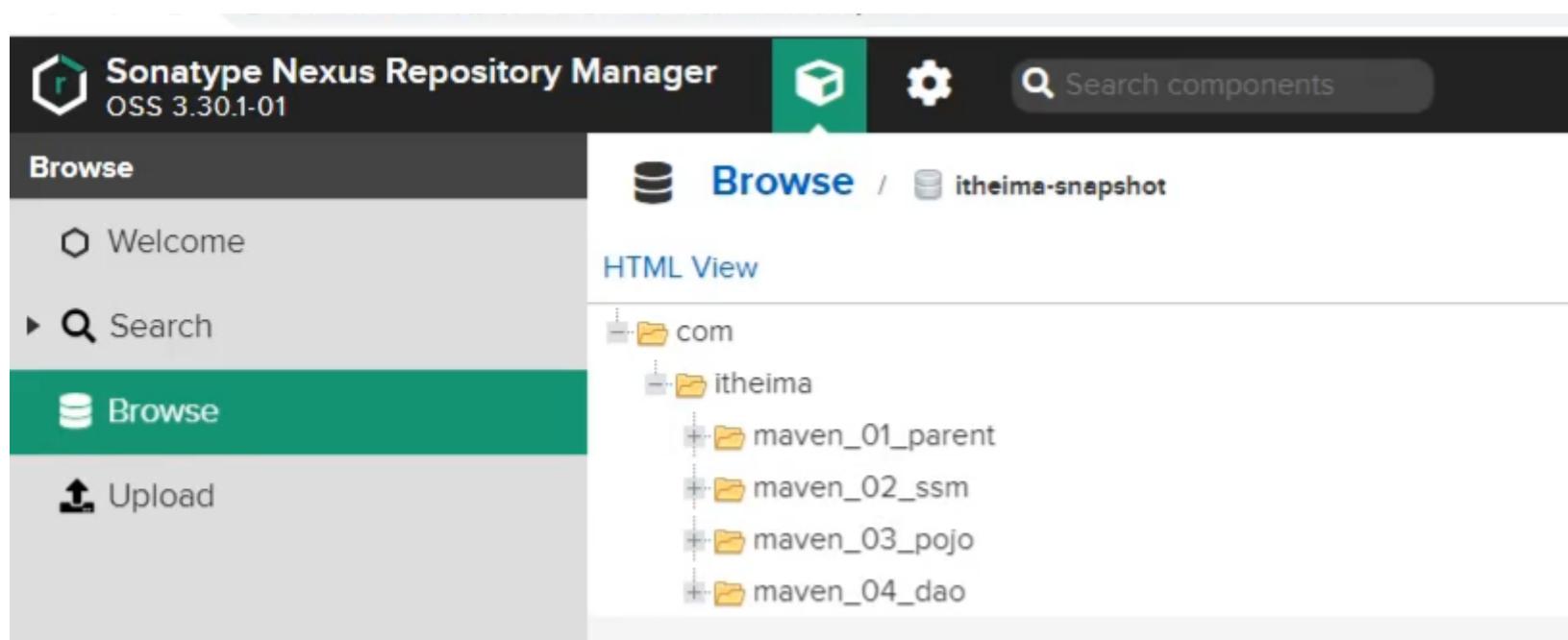
或者执行Maven命令

```
1 mvn deploy
```

注意：

要发布的项目都需要配置 `distributionManagement` 标签，要么在自己的 `pom.xml` 中配置，要么在其父项目中配置，然后子项目中继承父项目即可。

发布成功，在私服中就能看到：



现在发布是在itheima-snapshot仓库中，如果想发布到itheima-release仓库中就需要将项目pom.xml中的version修改成RELEASE即可。

如果想删除已经上传的资源，可以在界面上进行删除操作：



如果私服中没有对应的jar，会去中央仓库下载，速度很慢。可以配置让私服去阿里云中下载依赖。

This screenshot shows two parts of the Nexus Repository Manager interface. On the left, the 'Administration' section is open, with 'Repositories' highlighted. On the right, the 'Repositories' management screen is shown. It lists several repositories: 'maven-central' (selected and highlighted with a red box), 'maven-public', 'maven-releases', and 'maven-snapshots'. The 'maven-central' row has 'proxy' listed under 'Type'. A red arrow points from the 'Repositories' link in the sidebar to the 'maven-central' row. Another red arrow points from the 'Settings' tab in the top right to the 'Remote storage' field, which contains the URL 'http://maven.aliyun.com/nexus/content/groups/public'. Below this field, a note says 'Blocked: 配置成阿里云中央仓库地址，最后点击下方“保存”按钮即可' (Configure to Alipay Cloud Central Repository address, then click the 'Save' button below).

至此私服的搭建就已经完成，相对来说有点麻烦，但是步骤都比较固定，后期大家如果需要的话，就可以参考上面的步骤一步步完成搭建即可。