尚硅谷大数据技术之 Maven

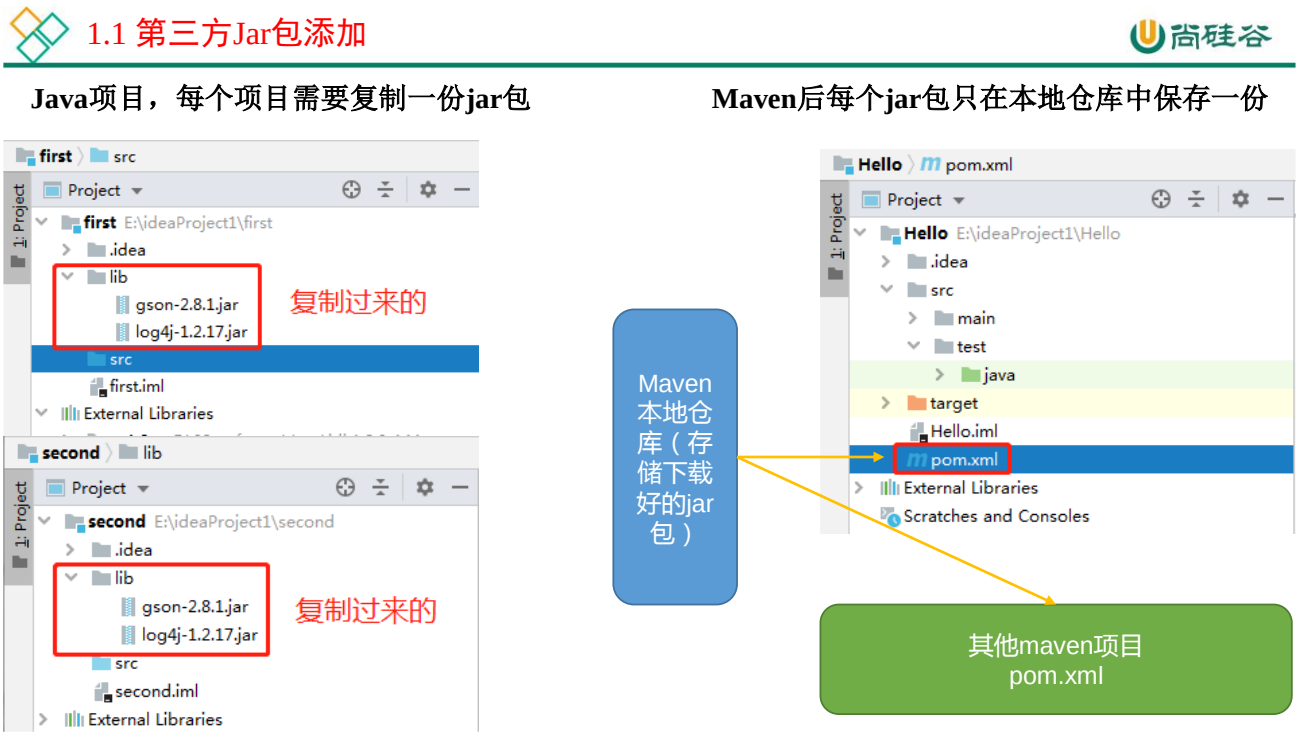
（作者：尚硅谷大数据研发部）

版本：V3.1

第 1 章 为什么要使用 Maven

Maven 是干什么用的？我们先通过企业开发中的实际需求来看一看哪些方面是我们现有技术的不足。

1.1 第三方 Jar 包添加



在今天的 JavaEE 开发领域，有大量的第三方框架和工具可以供我们使用。要使用这些 jar 包最简单的方法就是复制粘贴到 WEB-INF 目录下的 lib 目录下。但是这会导致每次创建一个新的工程就需要将 jar 包重复复制到 lib 目录下，从而造成工作区中存在大量重复的文件。

而使用 **Maven 后每个 jar 包只在本地仓库中保存一份**，需要 jar 包的工程只需要维护一个文本形式的 jar 包的引用——我们称之为“坐标”。不仅**极大的节约了存储空间**，更**避免了重复文件太多而造成的混乱。**

1.2 第三方 Jar 包获取

1.2 第三方Jar包获取

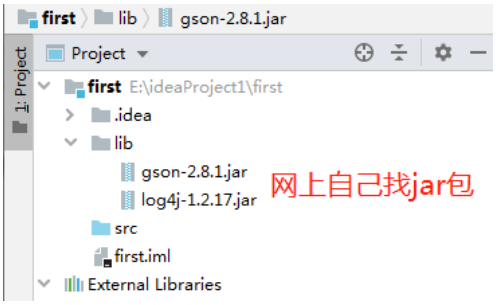
尚硅谷

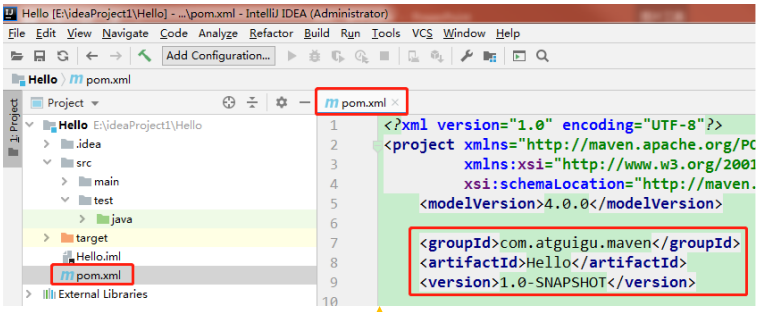
Java项目，自己去网上找jar包；

费劲心血找的jar包里有的时候并没有你需要的那个类；

Maven就会自动从仓库进行下载；

并同时下载这个jar包所依赖的其他jar包，**规范、完整、准确！**

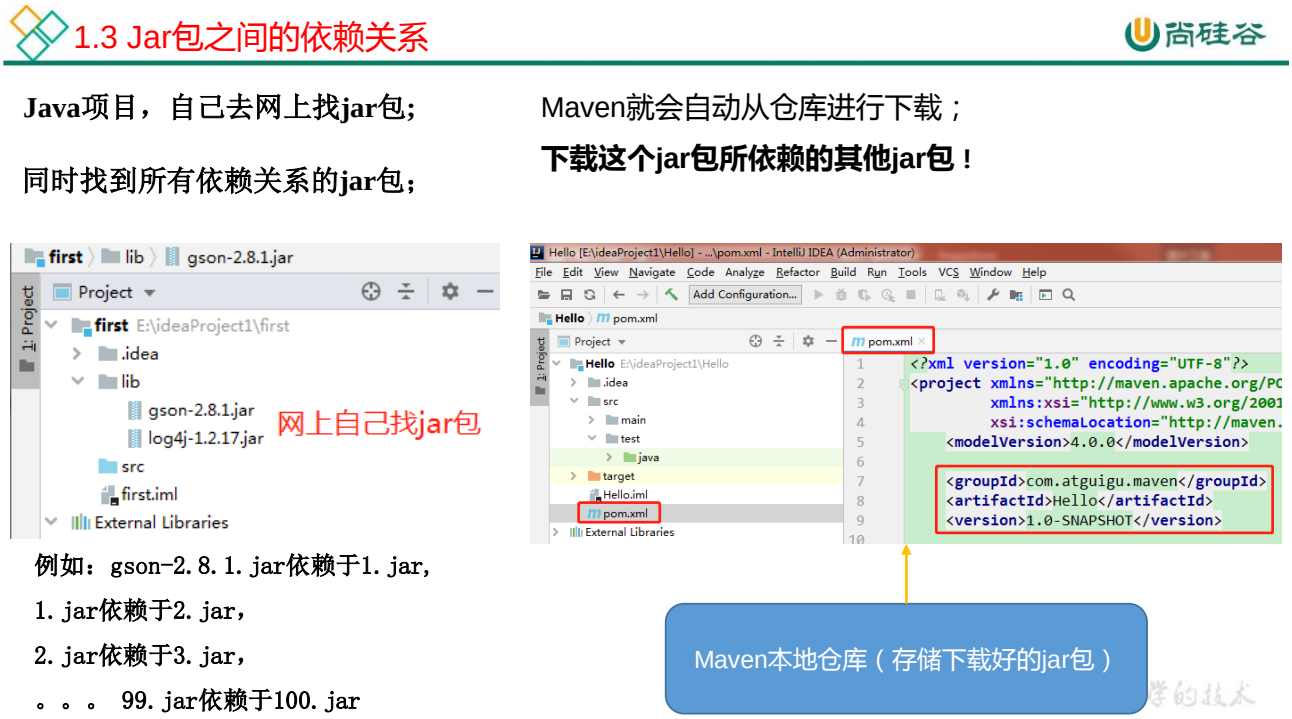




JavaEE 开发中需要使用到的 jar 包种类繁多，几乎每个 jar 包在其本身的官网上的获取方式都不尽相同。为了查找一个 jar 包找遍互联网，身心俱疲，没有经历过的人或许体会不到这种折磨。不仅如此，费劲心血找的 jar 包里有的时候并没有你需要的那个类，又或者有同名的类没有你要的方法——以不规范的方式获取的 jar 包也往往是不规范的。

使用 Maven 我们可以享受到一个完全统一规范的 jar 包管理体系。**你只需要在你的项目中以坐标的方式依赖一个 jar 包，Maven 就会自动从中央仓库进行下载，并同时下载这个 jar 包所依赖的其他 jar 包——规范、完整、准确！**一次性解决所有问题！

1.3 Jar 包之间的依赖关系



jar 包往往不是孤立存在的，很多 jar 包都需要在其他 jar 包的支持下才能够正常工作，我们称之为 jar 包之间的依赖关系。最典型的例子是：commons-fileupload-1.3.jar 依赖于 commons-io-2.0.1.jar，如果没有 IO 包，FileUpload 包就不能正常工作。

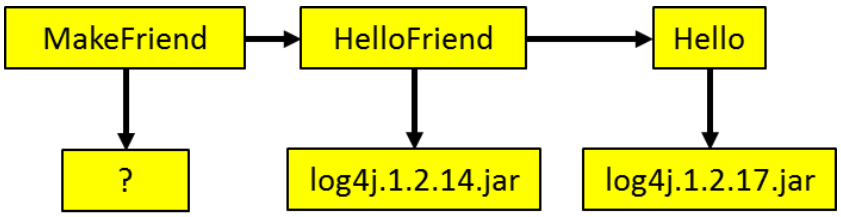
那么问题来了，你知道你所使用的所有 jar 包的依赖关系吗？当你拿到一个新的从未使用过的 jar 包，你如何得知他需要哪些 jar 包的支持呢？如果不了解这个情况，导入的 jar 包不够，那么现有的程序将不能正常工作。再进一步，当你的项目中需要用到上百个 jar 包时，你还会人为的，手工的逐一确认它们依赖的其他 jar 包吗？这简直是不可想象的。

而引入 Maven 后，**Maven 就可以替我们自动的将当前 jar 包所依赖的其他所有 jar 包全部导入进来，无需人工参与**，节约了我们大量的时间和精力。用实际例子来说明就是：通过 Maven 导入 commons-fileupload-1.3.jar 后，commons-io-2.0.1.jar 会被自动导入，程序员不必了解这个依赖关系。

1.4 Jar 包之间的冲突处理

上一点说的是 jar 包不足项目无法正常工作，但其实有的时候 jar 包多了项目仍然无法正常工作，这就是 jar 包之间的冲突。

举个例子：我们现在有三个工程 MakeFriend、HelloFriend 和 Hello。MakeFriend 依赖 HelloFriend，HelloFriend 依赖 Hello。而 Hello 依赖 log4j.1.2.17.jar，HelloFriend 依赖 log4j.1.2.14.jar。如下图所示：



那么 MakeFriend 工程的运行时环境中该导入 log4j.1.2.14.jar 呢还是 log4j.1.2.17.jar 呢？

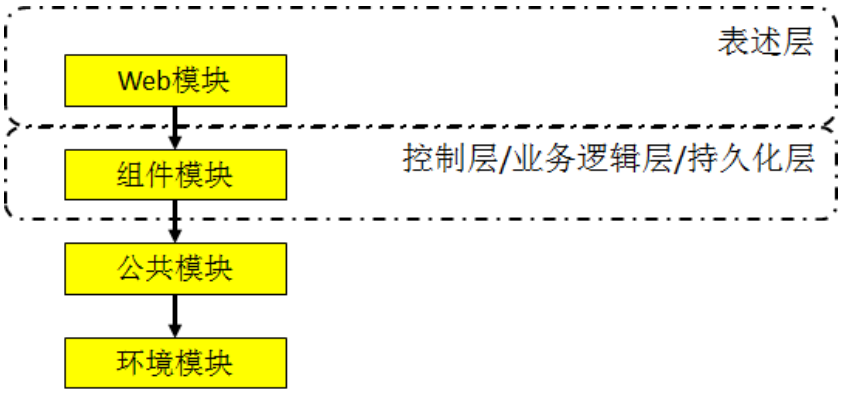
这样的问题一个两个还可以手工解决，但如果系统中存在几十上百的 jar 包，他们之间的依赖关系会非常复杂，几乎不可能手工实现依赖关系的梳理。

使用 Maven 就可以自动的处理 jar 包之间的冲突问题。因为 Maven 中内置了两条依赖原则：**最短路径者优先和先声明者优先**，上述问题 MakeFriend 工程会自动使用 log4j.1.2.14.jar。

1.5 将项目拆分成多个工程模块（了解）

随着 JavaEE 项目的规模越来越庞大，开发团队的规模也与日俱增。一个项目上千人的团队持续开发很多年对于 JavaEE 项目来说再正常不过。那么我们想象一下：几百上千的人开发的项目是同一个 Web 工程。那么架构师、项目经理该如何划分项目的模块、如何分工呢？这么大的项目已经不可能通过 package 结构来划分模块，必须将项目拆分成多个工程协同开发。多个模块工程中有的是 Java 工程，有的是 Web 工程。

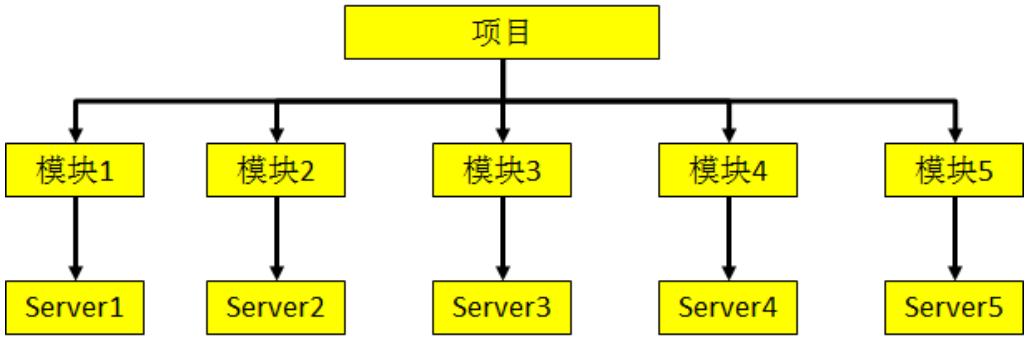
那么工程拆分后又如何进行互相调用和访问呢？这就需要用到 Maven 的依赖管理机制。例如：某项目拆分的情况如下。



上层模块依赖下层，所以下层模块中定义的 API 都可以为上层所调用和访问。

1.6 实现项目的分布式部署（了解）

在实际生产环境中，项目规模增加到一定程度后，可能每个模块都需要运行在独立的服务器上，我们称之为分布式部署，这里同样需要用到 Maven。



第 2 章 Maven 是什么？

如果上面的描述能够使你认识到使用 Maven 是多么的重要，我们下面就来介绍一下 Maven 是什么。

2.1 自动化构建工具

（1）Maven 这个单词的本意是：专家，内行。读音是['meɪv(ə)n]或['mevn]，不要读作“妈文”。

（2）Maven 是一款自动化构建工具，专注服务于 Java 平台的项目构建和依赖管理。在 JavaEE 开发的历史上构建工具的发展也经历了一系列的演化和变迁：Make→Ant→Maven→Gradle→其他……

2.2 构建的概念

构建并不是创建，创建一个工程并不等于构建一个项目。要了解构建的含义我们应该由浅入深的从以下三个层面来看：

（1）纯 Java 代码

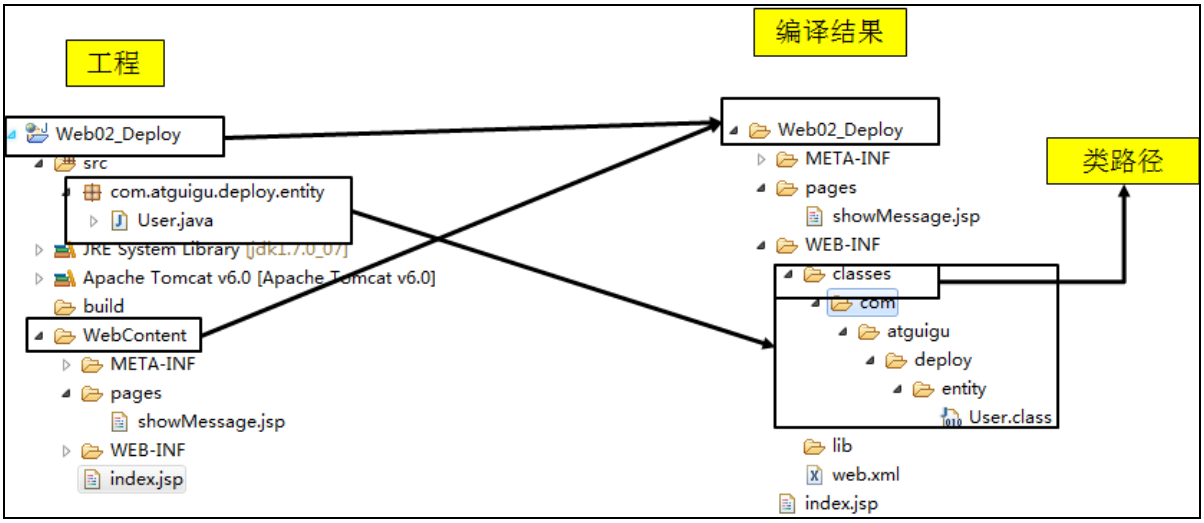
大家都知道，我们 Java 是一门编译型语言，.java 扩展名的源文件需要编译成.class 扩展名的字节码文件才能够执行。所以编写任何 Java 代码想要执行的话就必须经过**编译**得到对应的.class 文件。

（2）Web 工程

当我们需要通过浏览器访问 Java 程序时就必须将包含 Java 程序的 Web 工程编译的结果“拿”到服务器上的指定目录下，并启动服务器才行。这个“拿”的过程我们叫**部署**。

我们可以将未编译的 Web 工程比喻为一只生的鸡，编译好的 Web 工程是一只煮熟的鸡，编译部署的过程就是将鸡炖熟。

Web 工程和其编译结果的目录结构对比见下图：



（3）实际项目

在实际项目中整合第三方框架，Web 工程中除了 Java 程序和 JSP 页面、图片等静态资源之外，还包括第三方框架的 jar 包以及各种各样的配置文件。所有这些资源都必须按照正确的目录结构部署到服务器上，项目才可以运行。

所以综上所述：构建就是以我们编写的 Java 代码、框架配置文件、国际化等其他资源文件、JSP 页面和图片等静态资源作为“原材料”，去“生产”出一个可以运行的项目的过程。

那么项目构建的全过程中都包含哪些环节呢？

2.3 构建环节

（1）清理：删除以前的编译结果，为重新编译做好准备。

（2）编译：将 Java 源程序编译为字节码文件。

（3）测试：针对项目中的关键点进行测试，确保项目在迭代开发过程中关键点的正确性。

（4）报告：在每一次测试后以标准的格式记录和展示测试结果。

（5）打包：将一个包含诸多文件的工程封装为一个压缩文件用于安装或部署。Java 工程对应 jar 包，Web 工程对应 war 包。

（6）安装：在 Maven 环境下特指将打包的结果——jar 包或 war 包安装到本地仓库中。

（7）部署：将打包的结果部署到远程仓库或将 war 包部署到服务器上运行。

2.4 自动化构建

其实上述环节我们在 IDEA 中都可以找到对应的操作，只是不太标准。那么既然 IDE 已经可以进行构建了我们为什么还要使用 Maven 这样的构建工具呢？我们来看一个小故事：

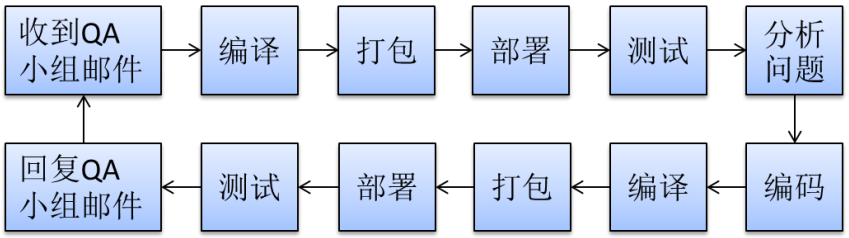
这是阳光明媚的一天。托马斯向往常一样早早的来到了公司，冲好一杯咖啡，进入了自己的邮箱——很不幸，QA 小组发来了一封邮件，报告了他昨天提交的模块的测试结果——有 BUG。“好吧，反正也不是

第一次”，托马斯摇摇头，进入 IDE，运行自己的程序，编译、打包、部署到服务器上，然后按照邮件中的操作路径进行测试。“嗯，没错，这个地方确实有问题”，托马斯说道。于是托马斯开始尝试修复这个 BUG，当他差不多有眉目的时候已经到了午饭时间。

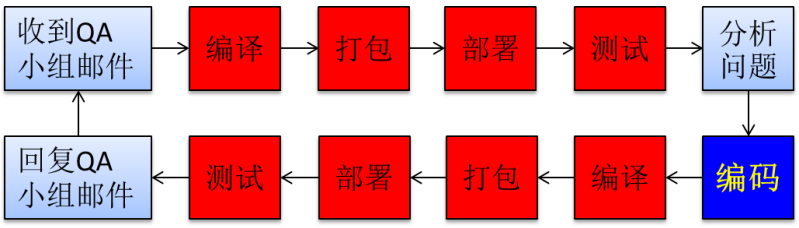
下午继续工作。BUG 很快被修正了，接着托马斯对模块重新进行了编译、打包、部署，测试之后确认没有问题了，回复了 QA 小组的邮件。

一天就这样过去了，明媚的阳光化作了美丽的晚霞，托马斯却觉得生活并不像晚霞那样美好啊。

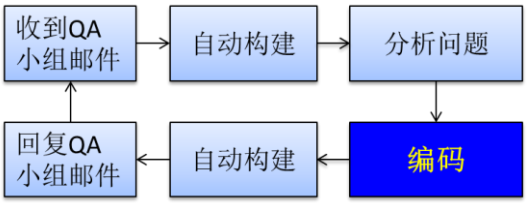
让我们来梳理一下托马斯这一天中的工作内容



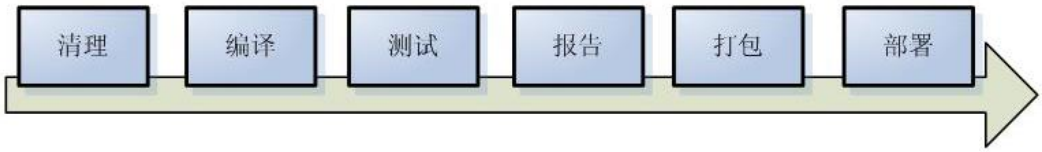
从中我们发现，托马斯的很大一部分时间花在了“编译、打包、部署、测试”这些程式化的工作上面，而真正需要由“人”的智慧实现的分析问题和编码却只占了很少一部分。



能否将这些程式化的工作交给机器自动完成呢？——当然可以！这就是自动化构建。



那么 Maven 又是如何实现自动化构建的呢？简单的说来就是它可以自动的从构建过程的起点一直执行到终点：



第 3 章 Maven 如何使用

在这一节中，我们来看看 Maven 核心程序的安装和本地仓库的必要设置。然后我们就可以编写第一个 Maven 程序了。

3.1 安装 Maven 核心程序

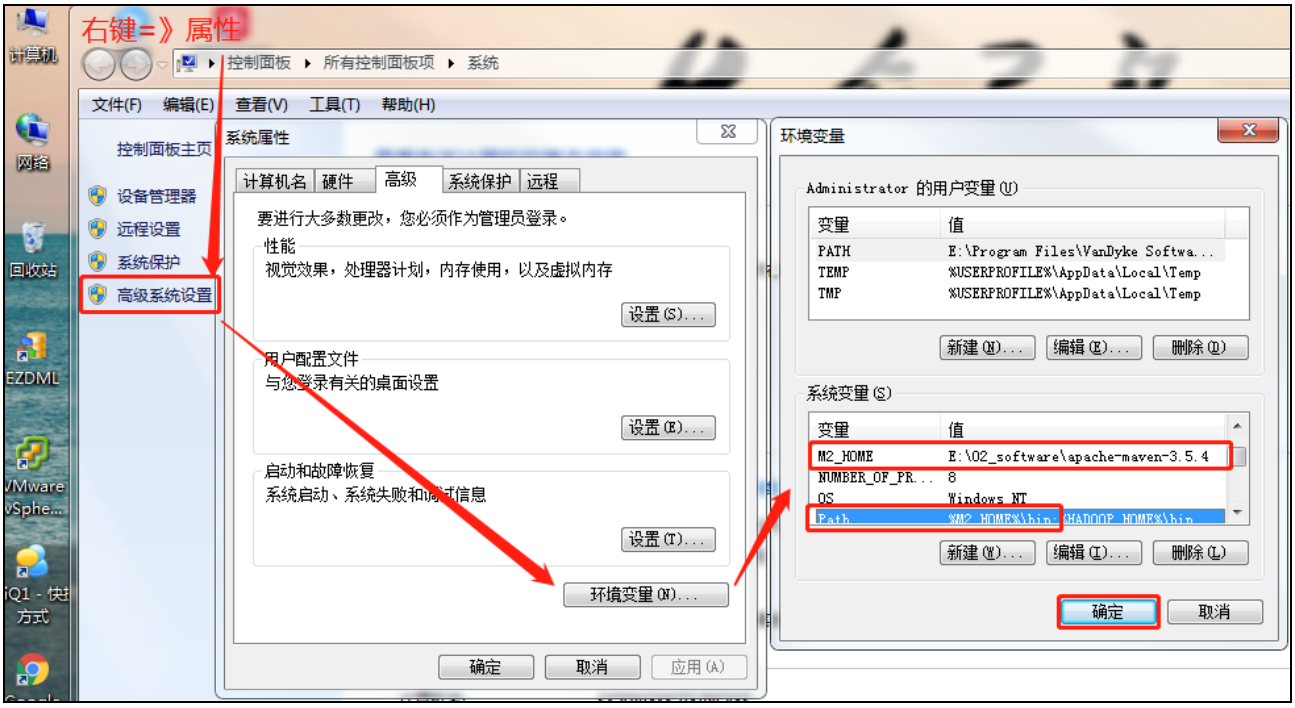
1）检查 JAVA_HOME 环境变量。Maven 是使用 Java 开发的，所以必须知道当前系统环境中 JDK 的安装目录。

```
C:\Windows\System32>echo %JAVA_HOME%
D:\MyWork\Program\jdk1.8.0_211
```

2）解压 Maven 的核心程序。将 apache-maven-3.5.4-bin.zip 解压到一个非中文无空格的目录下。例如：

```
D:\MyWork\Program\apache-maven-3.5.4
```

3）配置环境变量。



（1）在系统变量里面创建 M2_HOME 变量，并赋值

```
变量：M2_HOME
值：D:\MyWork\Program\apache-maven-3.5.4
```

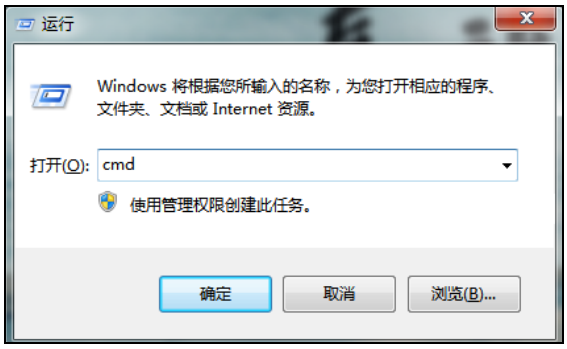
（2）在 Path 变量中，添加 maven 环境变量

```
变量：Path
值：%M2_HOME%\bin 或 D:\MyWork\Program\apache-maven-3.5.4\bin
```

4）查看 Maven 版本信息验证安装是否正确

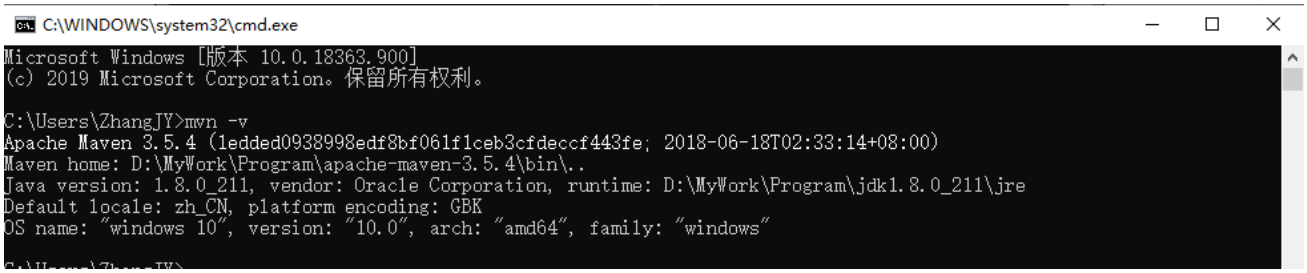
（1）按 Win +r，进入电脑运行模式；

（2）在打开里面输入：cmd



（3）在管理员窗口输入

```
C:\Users\ZhangJY>mvn -v
```



3.2 Maven 联网问题

（1）Maven 默认的本地仓库：~\.m2\repository 目录。

说明：~表示当前用户的家目录。

（2）Maven 的核心配置文件位置：

```
解压目录 D:\MyWork\Program\apache-maven-3.5.4\conf\settings.xml
```

（3）本地仓库地址更改到 D:\MyWork\Program\RepMaven，默认在 C:\Users\Administrator\.m2\repository

```
<localRepository>D:\MyWork\Program\RepMaven</localRepository>
```

（4）配置阿里云镜像（下载速度快）

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

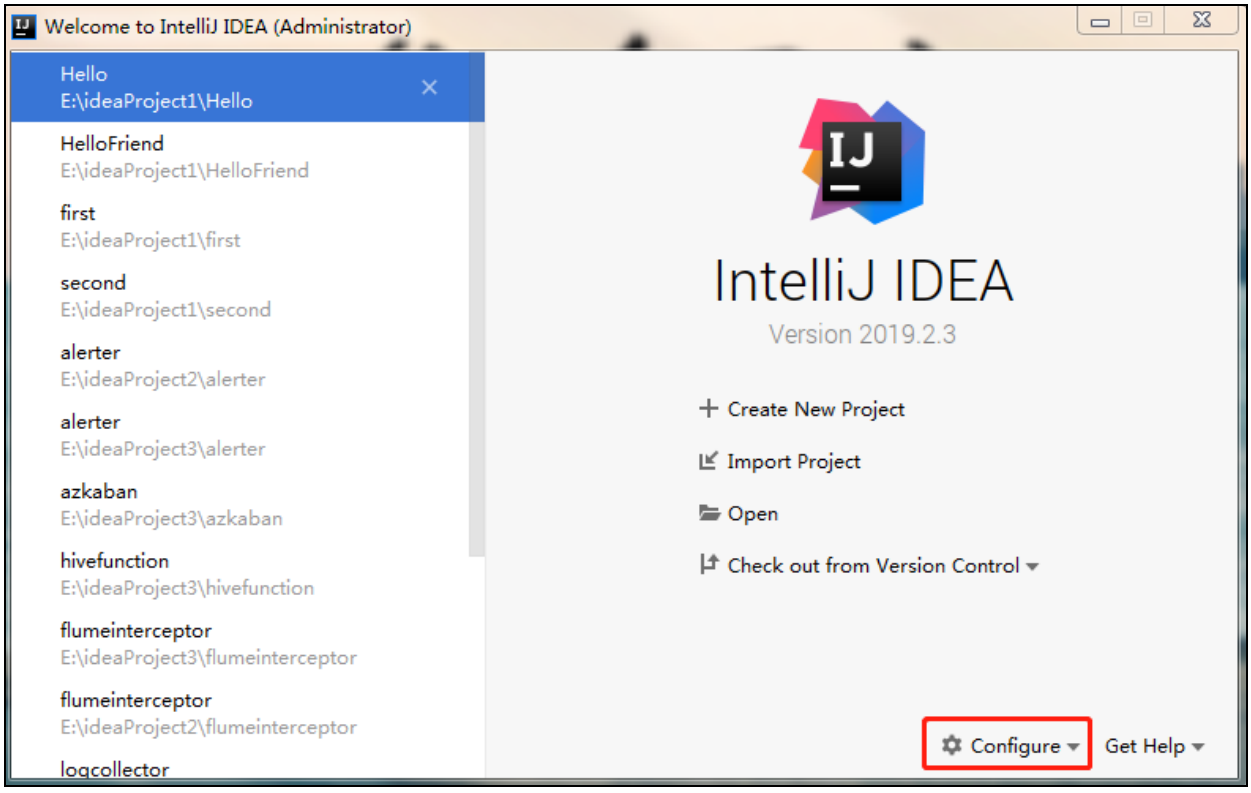
3.3 Maven 编译版本

在 settings.xml 中的<profiles></profiles>标签中加入如下内容

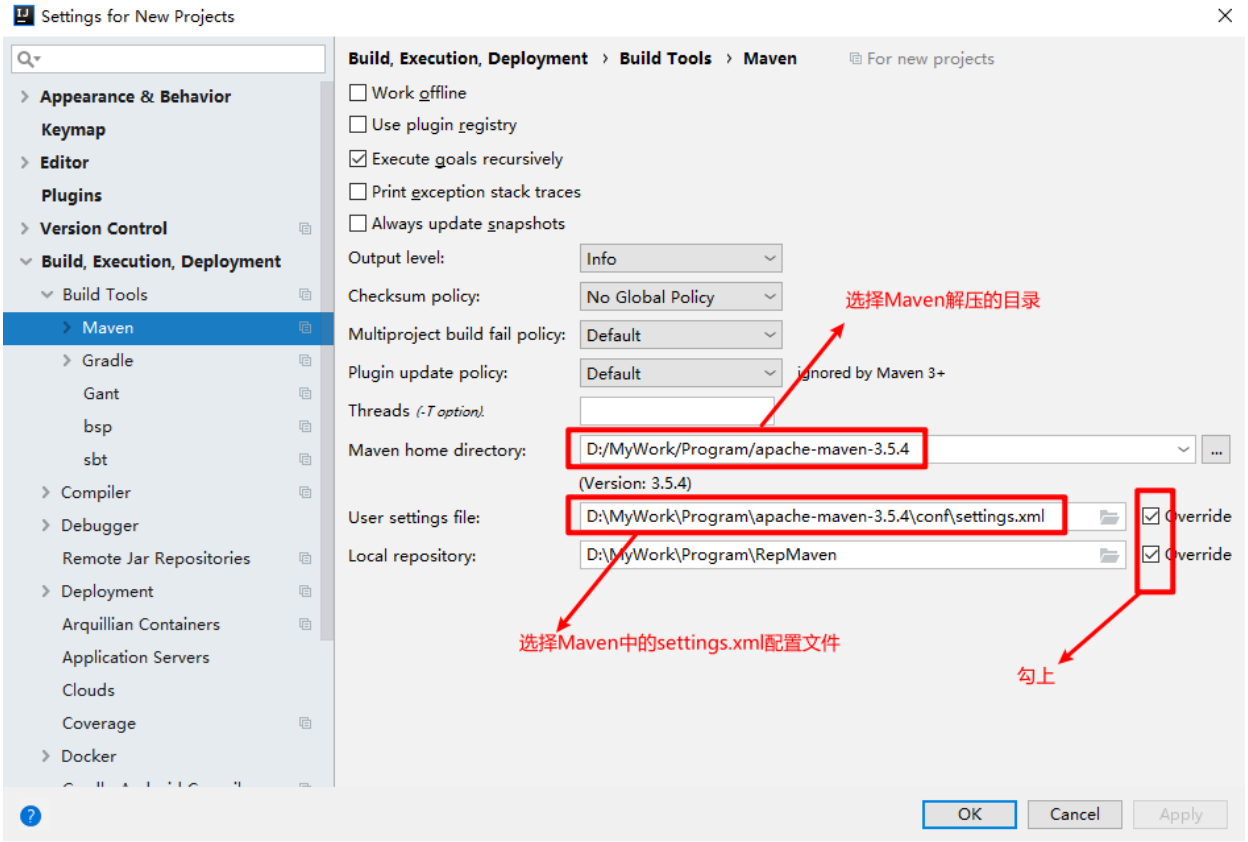
```
<profile>
  <id>jdk-1.8</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>1.8</jdk>
  </activation>
  <properties>
     <maven.compiler.source>1.8</maven.compiler.source>
     <maven.compiler.target>1.8</maven.compiler.target>
     <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>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

3.4 在 Idea 中配置 Maven

1）close project 所有项目后，回到如下页面，点击右下角的 Configure --->点击 setting

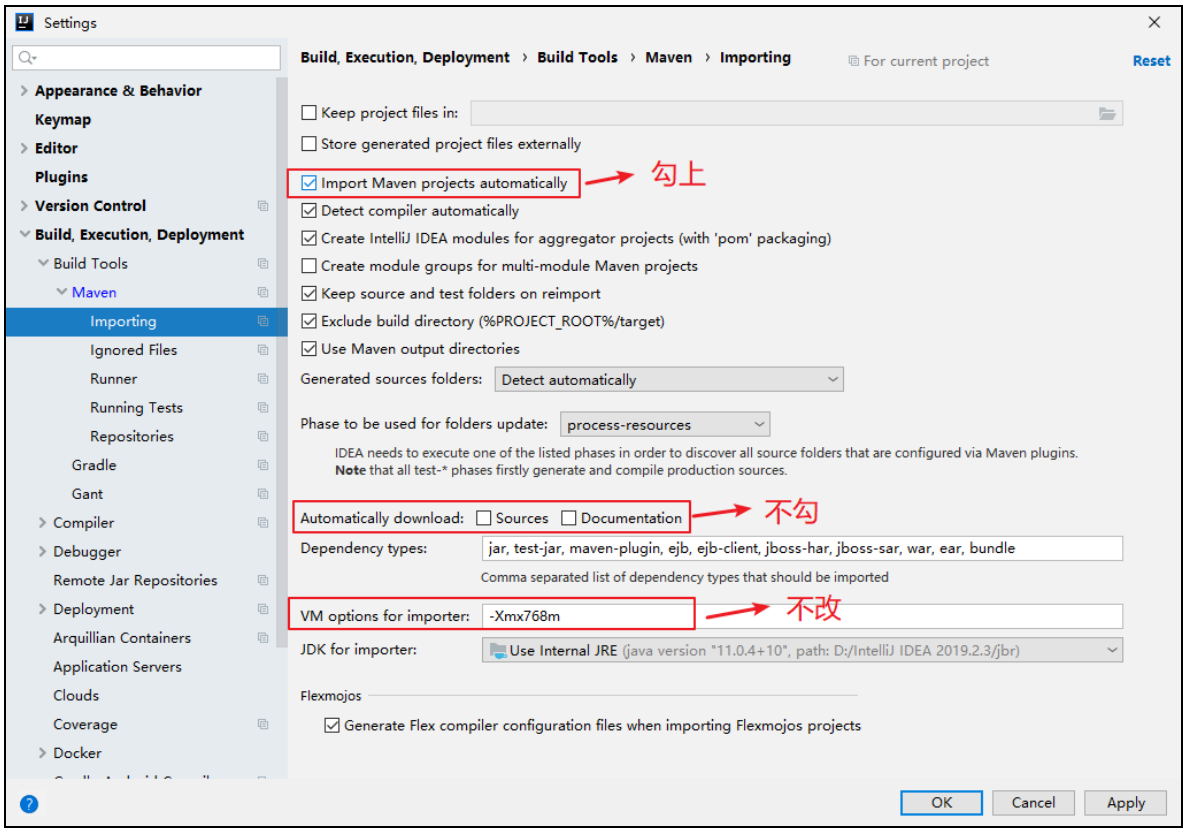


2）设置 Maven 的安装目录及本地仓库



- Maven home directory：可以指定本地 Maven 的安装目录所在，因为我已经配置了 M2_HOME 系统参数，所以直接这样配置 IntelliJ IDEA 是可以找到的。但是假如你没有配置的话，这里可以选择你的 Maven 安装目录。此外，这里不建议使用 IDEA 默认的。
- User settings file / Local repository：我们还可以指定 Maven 的 settings.xml 位置和本地仓库位置。

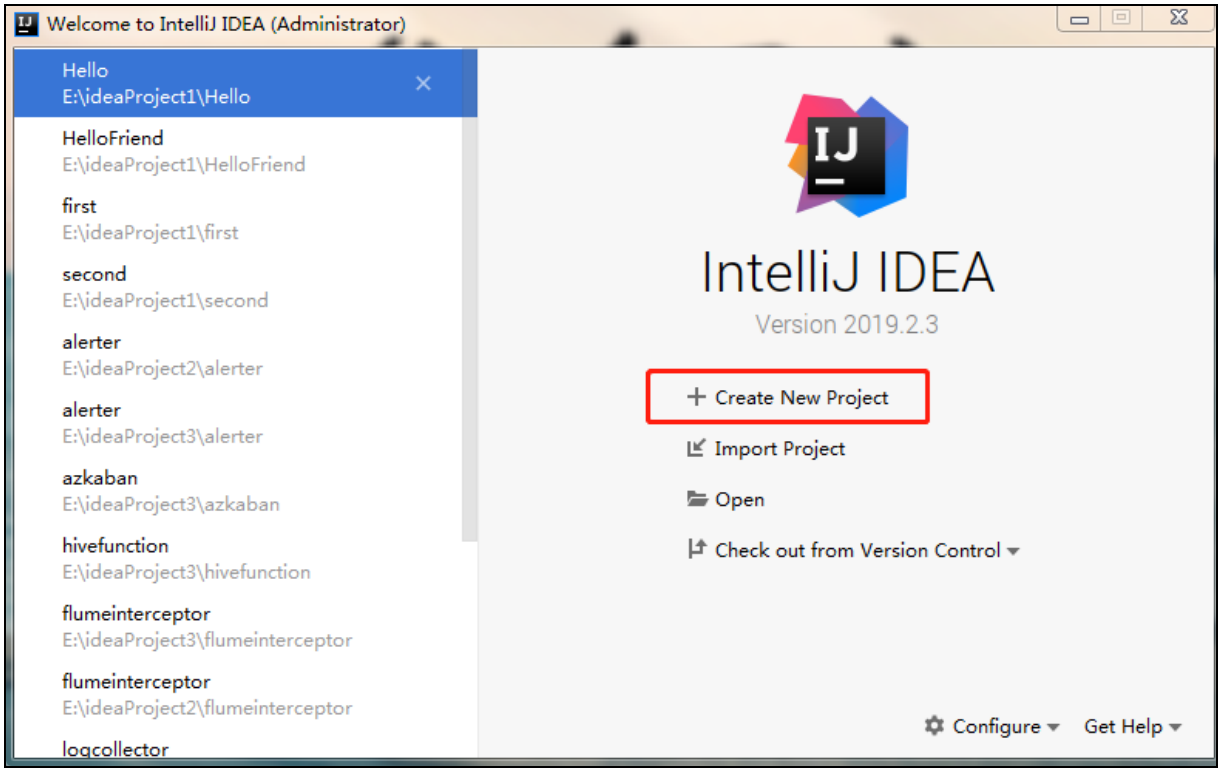
3）配置 Maven 自动导入依赖的 jar 包



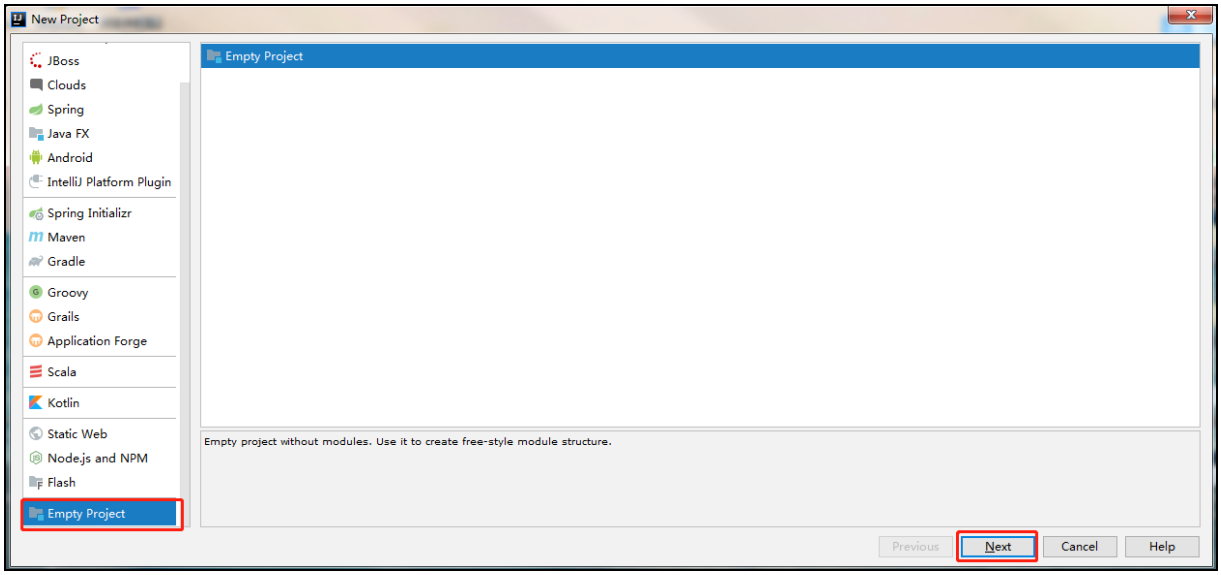
- Import Maven projects automatically：表示 IntelliJ IDEA 会实时监控项目的 pom.xml 文件，进行项目变动设置，勾选上。
- Automatically download：在 Maven 导入依赖包的时候是否自动下载源码和文档。默认是没有勾选的，也不建议勾选，原因是这样可以加快项目从外网导入依赖包的速度，如果我们需要源码和文档的时候我们到时候再针对某个依赖包进行联网下载即可。IntelliJ IDEA 支持直接从公网下载源码和文档的。
- VM options for importer：可以设置导入的 VM 参数。一般这个都不需要主动改，除非项目真的导入太慢了我们再增大此参数。

3.5 第一个 Maven 程序

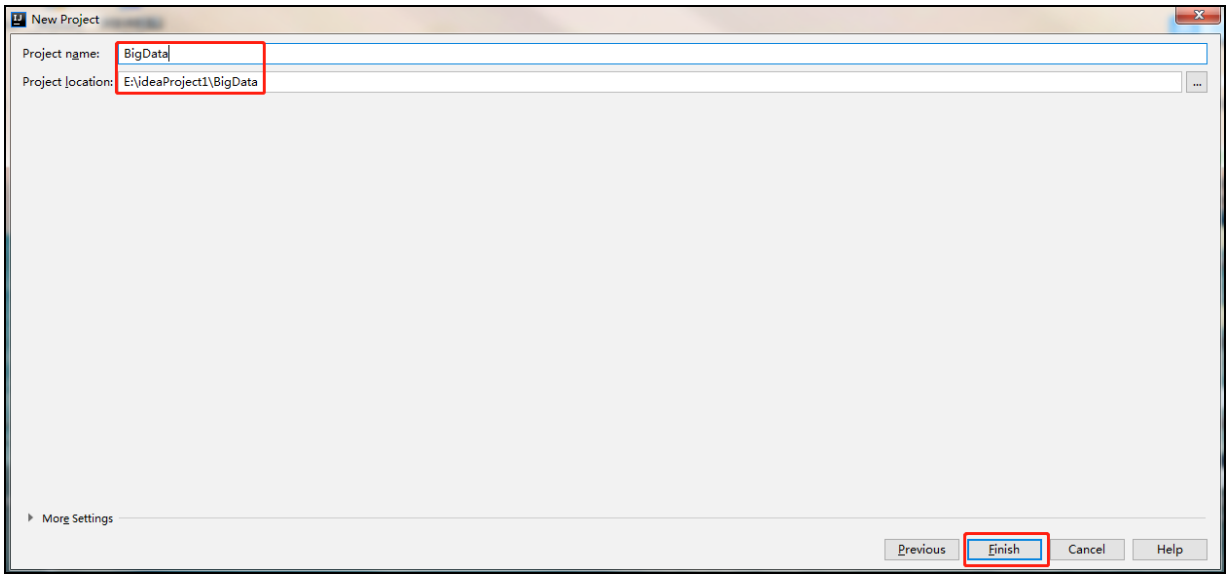
1）创建 Project

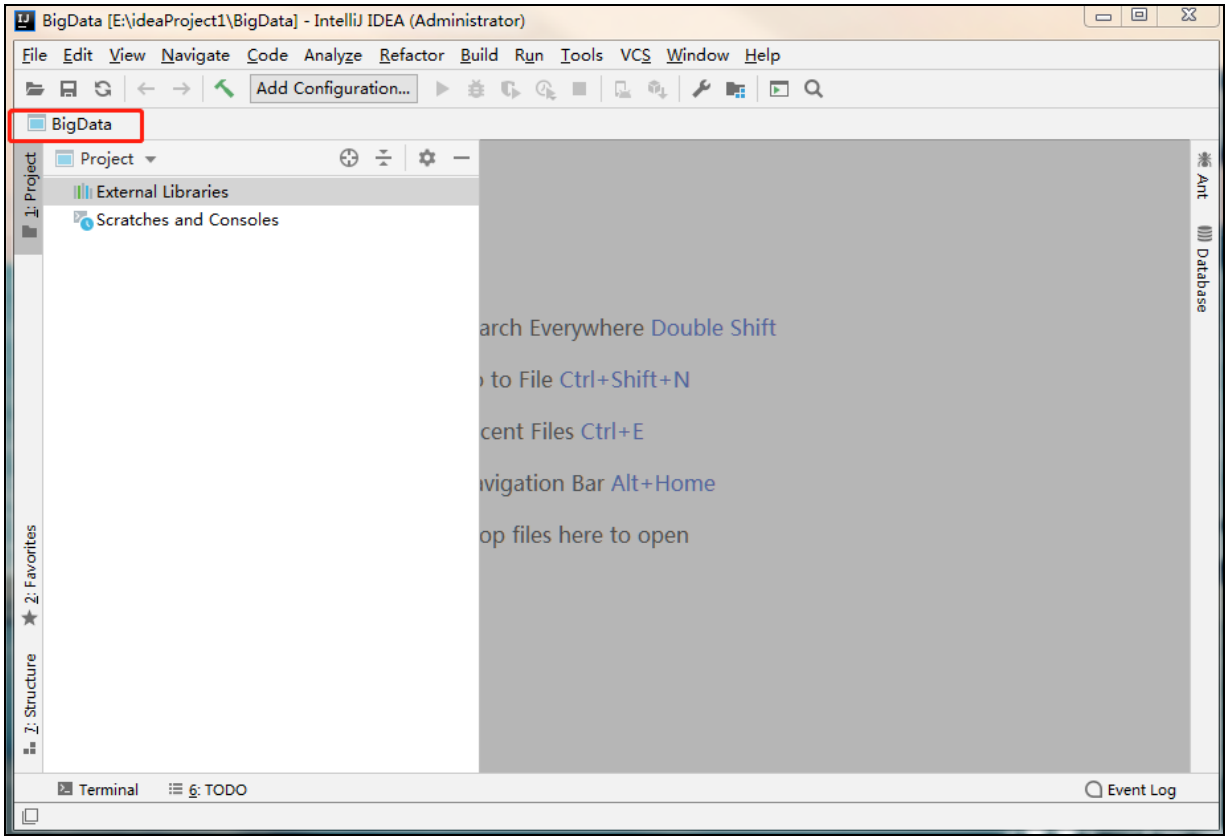


2）创建一个空的 Project

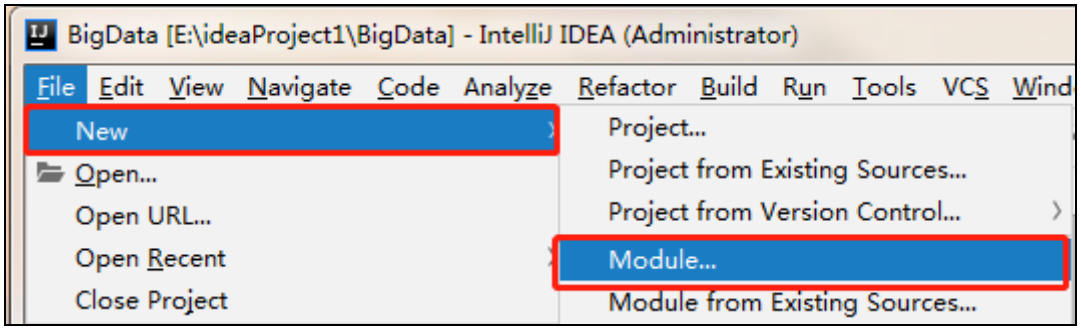


3）创建一个空的 Project

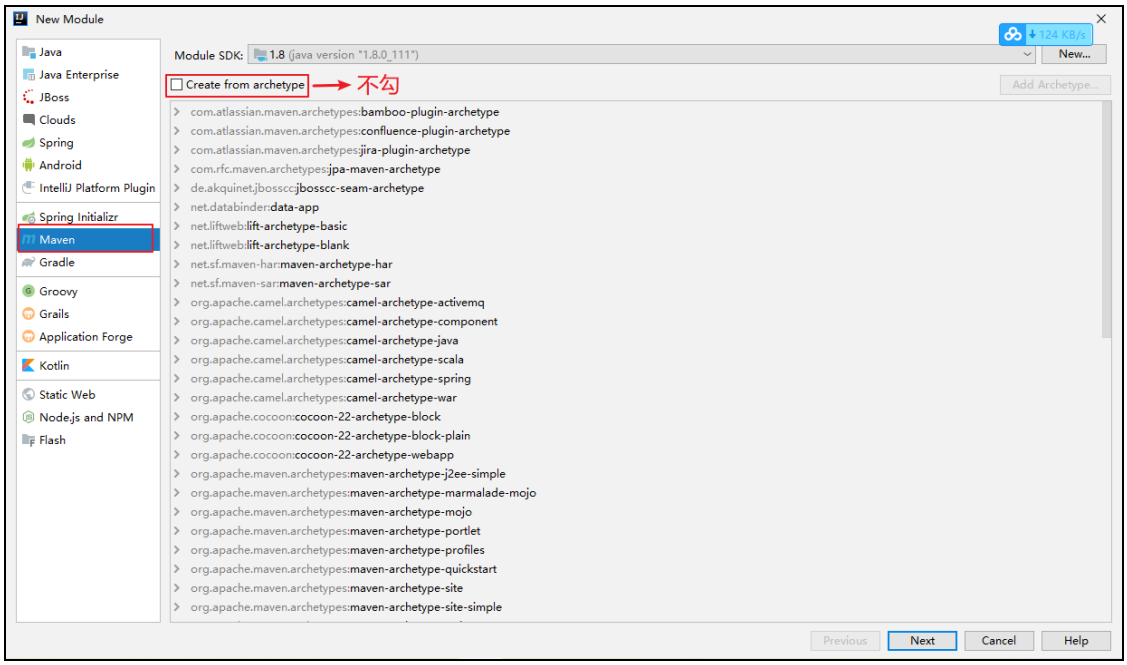




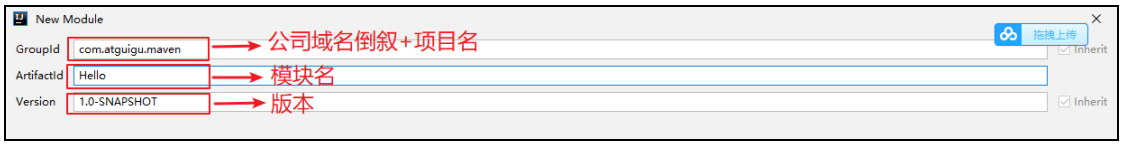
4）创建一个 module



5）右键→new Module→Maven

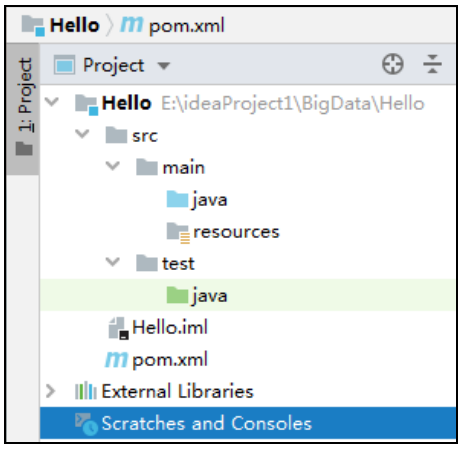


6）点击 Next，配置坐标



7）点击 Next，给 Module 命名

目录结构及说明



- main 目录用于存放主程序。
- java 目录用于存放源代码文件。
- resources 目录用于存放配置文件和资源文件。
- test 目录用于存放测试程序。

4）配置 Maven 的核心配置文件 pom.xml

```
<?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.atguigu.maven</groupId>
    <artifactId>Hello</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

5）编写主代码

在 src/main/java 目录下新建文件 Hello.java

```
public class Hello {
    public String sayHello(String name){
        return "Hello "+name+"!";
    }
}
```

6）编写测试代码

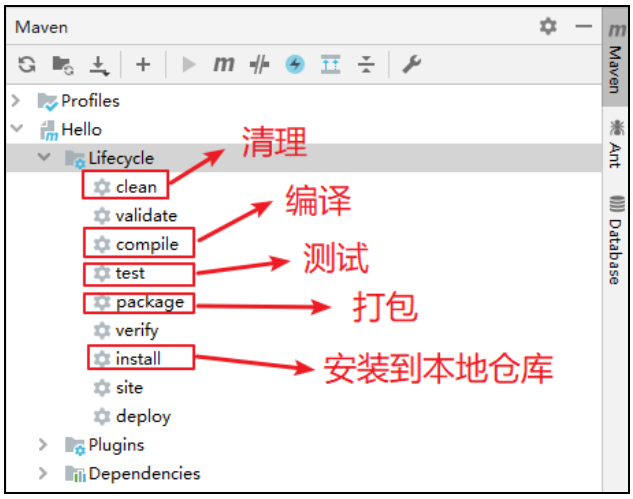
在/src/test/java 目录下新建测试文件 HelloTest.java

```
import org.junit.Test;

public class HelloTest {

    @Test
    public void testHello(){
        Hello hello = new Hello();
        String maven = hello.sayHello("Maven");
        System.out.println(maven);
    }
}
```

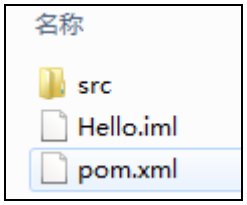
7）使用 Maven 的方式运行 Maven 工程



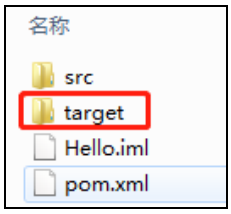
来到 BigData 项目的根目录（例如 E:\ideaProject1\BigData）。

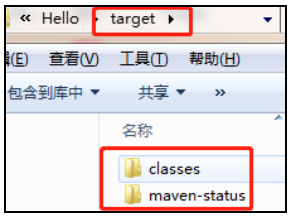
（1）compile 命令，查看 target 目录的变化

编译前=》



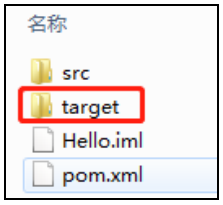
编译后=》



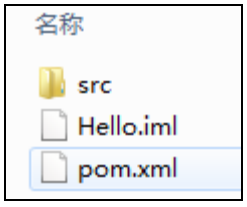


（2）clean，然后再次查看根目录变化

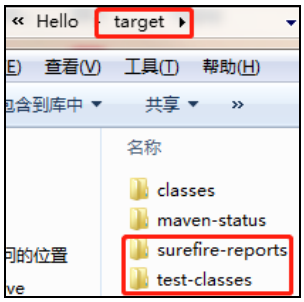
clean 前=》



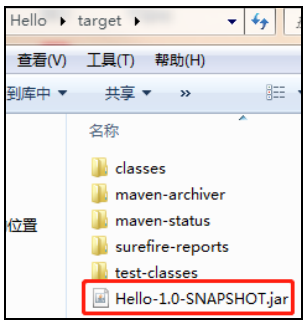
clean 后=》



（3）test 命令，查看 target 目录变化

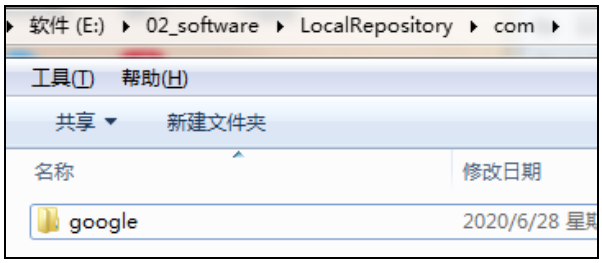


（4）package 命令，查看 target 目录变化

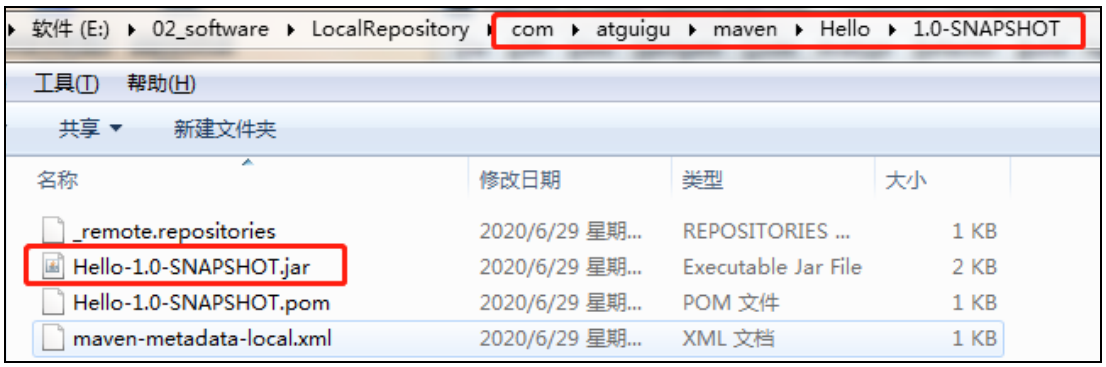


（5）install 命令，查看本地仓库的目录变化

（1）执行 install 命令前:



（2）执行命令后



3.6 Maven 打包插件

Maven 本身的打包插件不负责将依赖的 jar 包一并打入到 jar 包中。如果项目所依赖的 jar 包在服务器环境中提供了还好，如果服务器环境中没有提供，则比较悲惨，运行各种 ClassNotFound….你们懂的!

因此需要一款能够将项目所依赖的 jar 包 一并打入到 jar 中的插件来解决这些问题.

在 pom.xml 中加入如下内容:

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
              <archive>
                    <manifest>
                    <!-- 指定主类 -->
                        <mainClass>xxx.xxx.XXX</mainClass>
                    </manifest>
                </archive>
            </configuration>
            <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

第 4 章 Maven 核心概念

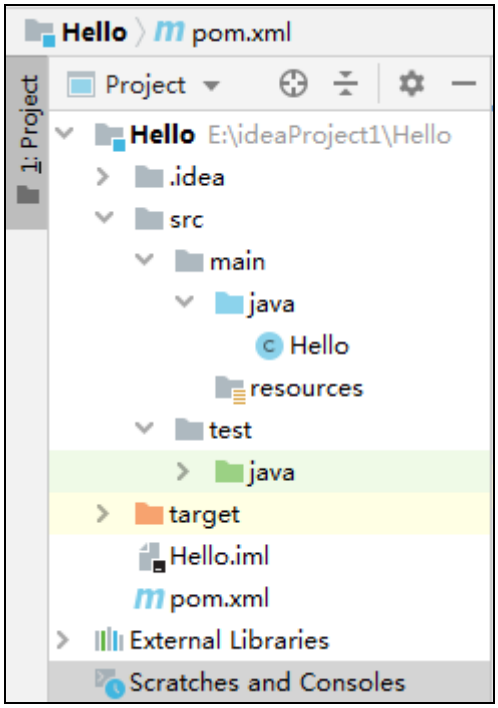
Maven 的核心概念包括：POM、约定的目录结构、坐标、依赖、仓库、生命周期、插件和目标、继承、聚合。

4.1 POM

Project Object Model：项目对象模型。**将 Java 工程的相关信息封装为对象**作为便于操作和管理的模型。Maven 工程的核心配置。可以说学习 Maven 就是学习 pom.xml 文件中的配置。

4.2 约定的目录结构

现在 JavaEE 开发领域普遍认同一个观点：约定>配置>编码。意思就是能用配置解决的问题就不编码，能基于约定的就不进行配置。而 Maven 正是因为指定了特定文件保存的目录才能够对我们的 Java 工程进行自动化构建。



4.3 坐标

1）几何中的坐标

（1）在一个平面中使用 x、y 两个向量可以唯一的确定平面中的一个点。

（2）在空间中使用 x、y、z 三个向量可以唯一的确定空间中的一个点。

2）Maven 的坐标

使用如下三个向量在 Maven 的仓库中唯一的确定一个 Maven 工程。

（1）groupId：公司或组织的域名倒序+当前项目名称

（2）artifactId：当前项目的模块名称

（3）version：当前模块的版本

在项目的 pom.xml 文件中存储坐标

```
<?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.atguigu.maven</groupId>
    <artifactId>Hello</artifactId>
    <version>1.0-SNAPSHOT</version>

</project>
```

3）如何通过坐标到仓库中查找 jar 包？

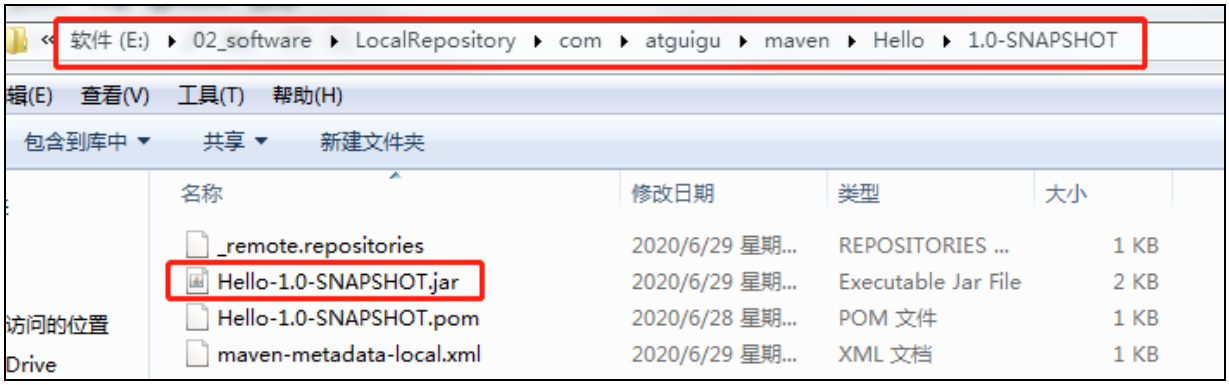
（1）将 gav 三个向量连起来

```
com.atguigu.maven + Hello + 1.0-SNAPSHOT
```

（2）以连起来的字符串作为目录结构到仓库中查找

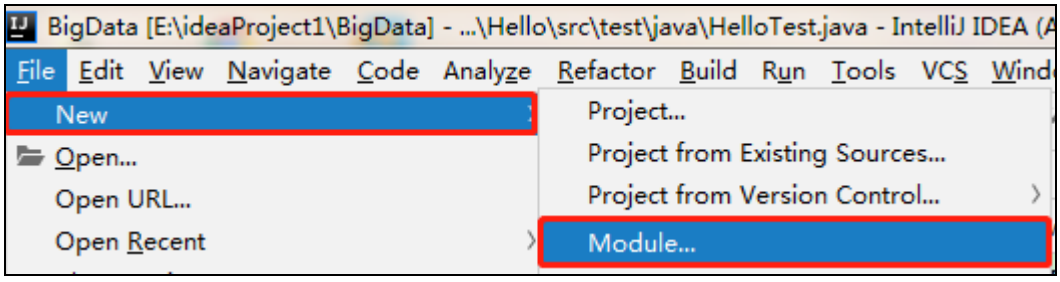
```
com/atguigu/maven/Hello/1.0-SNAPSHOT/Hello-1.0-SNAPSHOT.jar
```

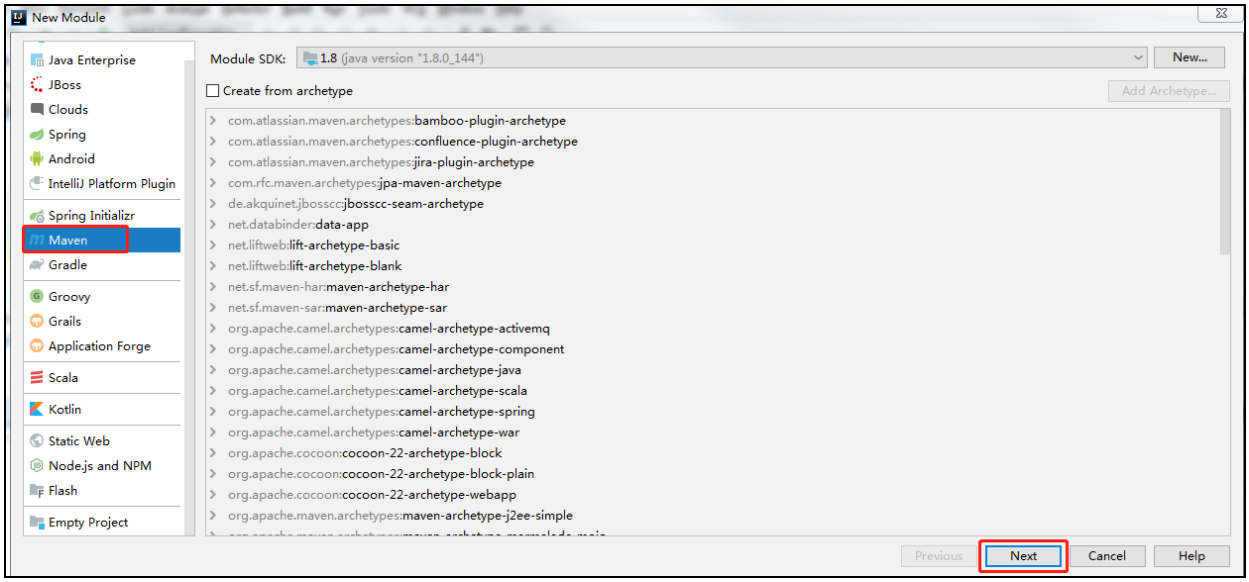
※注意：我们自己的 Maven 工程必须执行安装操作才会进入仓库。安装的命令是：mvn install

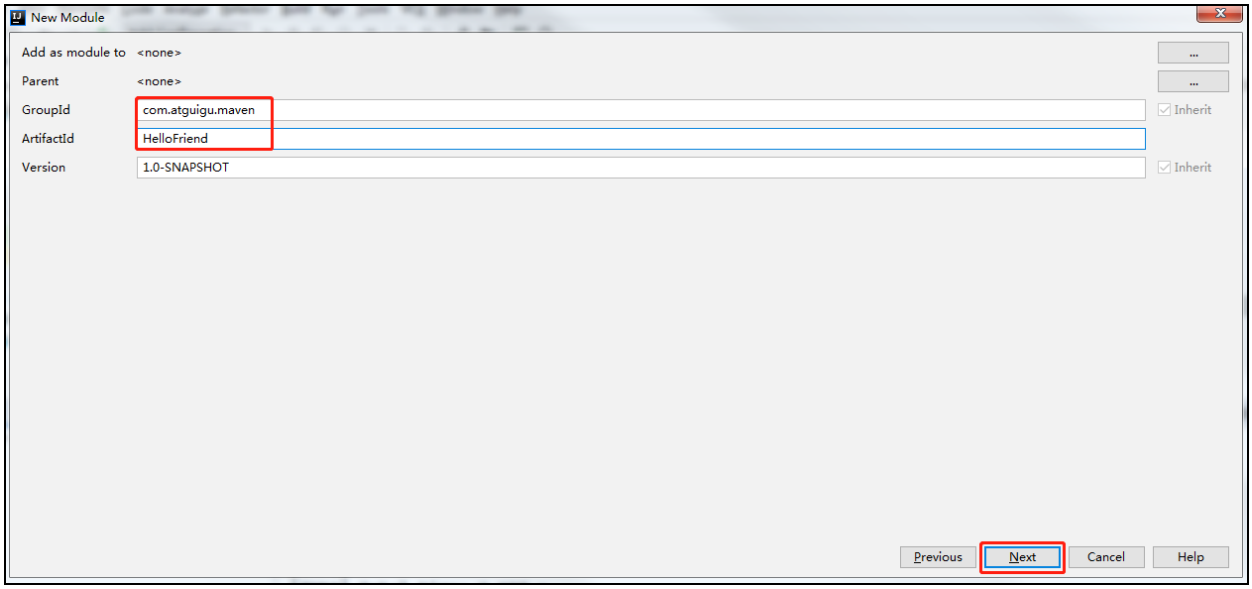


4.4 第二个 Maven 工程

1）创建 HelloFriend Module







2）在 pom.xml 配置文件中配置当前工程依赖 Hello

```
<?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.atguigu.maven</groupId>
    <artifactId>HelloFriend</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>com.atguigu.maven</groupId>
            <artifactId>Hello</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

3）主程序

在 src/**main**/java 目录下新建文件 HelloFriend.java

```
public class HelloFriend {

    public String sayHelloToFriend(String name){
        Hello hello = new Hello();
        String str = hello.sayHello(name)+" I am "+this.getMyName();
        return str;
    }

    public String getMyName(){
        return "Idea";
    }
}
```

4）测试程序

在/src/test/java 目录下新建测试文件 HelloFriendTest.java

```
import org.junit.Test;

public class HelloFriendTest {
    @Test
    public void testHelloFriend(){
        HelloFriend helloFriend = new HelloFriend();
        String results = helloFriend.sayHelloToFriend("Maven");
        System.out.println(results);
    }
}
```

5）关键：对 Hello 的依赖

这里 Hello 就是我们的第一个 Maven 工程，现在 HelloFriend 对它有依赖。那么这个依赖能否成功呢？更进一步的问题是：HelloFriend 工程会到哪里去找 Hello 呢？

答案是：本地仓库。任何一个 Maven 工程会根据坐标到本地仓库中去查找它所依赖的 jar 包。如果能够找到则可以正常工作，否则就不行。

4.5 依赖管理

1）基本概念

当 A jar 包需要用到 B jar 包中的类时，我们就说 A 对 B 有依赖。例如：HelloFriend-1.0-SNAPSHOT.jar 依赖于 Hello-1.0-SNAPSHOT.jar。

通过第二个 Maven 工程我们已经看到，当前工程会到本地仓库中根据坐标查找它所依赖的 jar 包。

配置的基本形式是使用 dependency 标签指定目标 jar 包的坐标。例如：

```
<dependencies>
    <!--坐标-->
    <dependency>
        <groupId>com.atguigu.maven</groupId>
        <artifactId>Hello</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

    <dependency>
```

```
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2）直接依赖和间接依赖

如果 A 依赖 B，B 依赖 C，那么 A→B 和 B→C 都是直接依赖，而 A→C 是间接依赖。

4.5.1 依赖的范围

1）compile（默认就是这个范围）

（1）main 目录下的 Java 代码**可以**访问这个范围的依赖

（2）test 目录下的 Java 代码**可以**访问这个范围的依赖

（3）部署到 Tomcat 服务器上运行时**要**放在 WEB-INF 的 lib 目录下

例如：对 Hello 的依赖。主程序、测试程序和服务器运行时都需要用到。

2）test

（1）main 目录下的 Java 代码**不能**访问这个范围的依赖

（2）test 目录下的 Java 代码**可以**访问这个范围的依赖

（3）部署到 Tomcat 服务器上运行时**不会**放在 WEB-INF 的 lib 目录下

例如：对 junit 的依赖。仅仅是测试程序部分需要。

3）provided

（1）main 目录下的 Java 代码**可以**访问这个范围的依赖

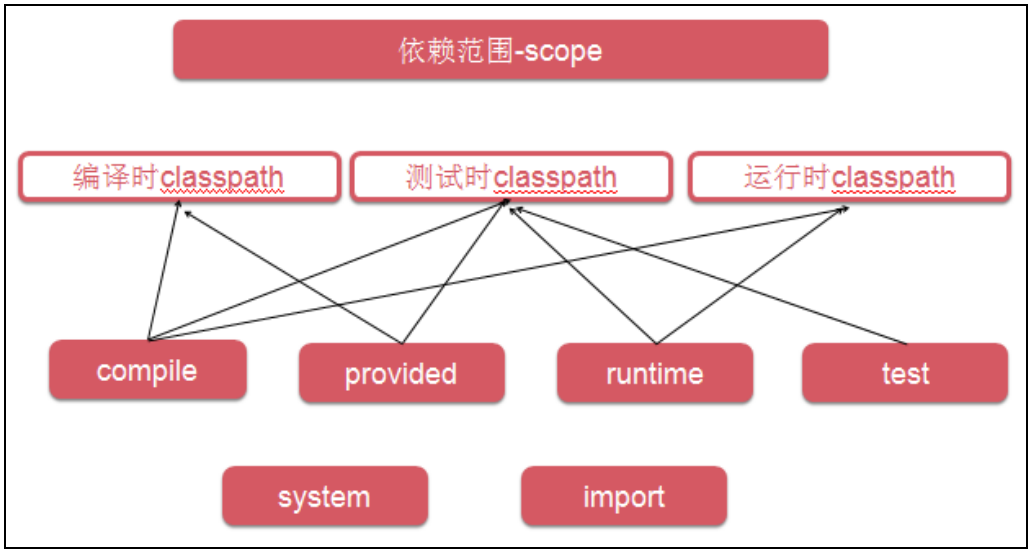
（2）test 目录下的 Java 代码**可以**访问这个范围的依赖

（3）部署到 Tomcat 服务器上运行时**不会**放在 WEB-INF 的 lib 目录下

例如：servlet-api 在服务器上运行时，Servlet 容器会提供相关 API，所以部署的时候不需要。

4）其他：runtime、import、system 等。

各个依赖范围的作用可以概括为下图：



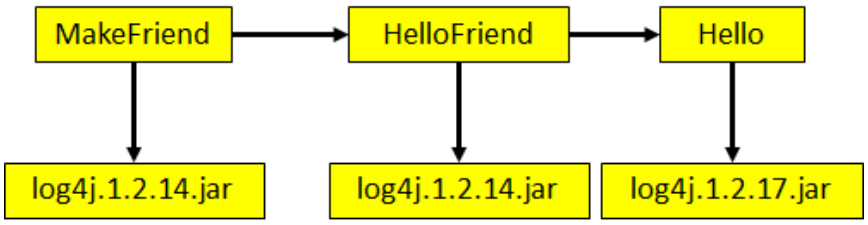
4.5.2 依赖的传递性

当存在间接依赖的情况时，主工程对间接依赖的 jar 可以访问吗？这要看间接依赖的 jar 包引入时的依赖范围——只有依赖范围为 compile 时可以访问。例如：

Maven 工程			依赖范围	对 A 的可见性
A	B	C	compile	√
		D	test	×
		E	provided	×

4.5.3 依赖的原则：解决 jar 包冲突

1）路径最短者优先



2）路径相同时先声明者优先

这里“声明”的先后顺序指的是 dependency 标签配置的先后顺序。

4.5.4 依赖的排除

有的时候为了确保程序正确可以将有可能重复的间接依赖排除。请看如下的例子：

假设当前工程为 MakeFriend，直接依赖 OurFriends。

OurFriends 依赖 commons-logging 的 1.1.1 对于 MakeFriend 来说是间接依赖。

当前工程 MakeFriend 直接依赖 commons-logging 的 1.1.2

加入 exclusions 配置后可以在依赖 OurFriends 的时候排除版本为 1.1.1 的 commons-logging 的间接依赖

```
<dependency>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>OurFriends</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!--依赖排除-->
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.2</version>
</dependency>
```

4.5.5 统一管理目标 Jar 包的版本

以对 Spring 的 jar 包依赖为例：Spring 的每一个版本中都包含 spring-context，springmvc 等 jar 包。我们应该导入版本一致的 Spring jar 包，而不是使用 4.0.0 的 spring-context 的同时使用 4.1.1 的 springmvc。

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.0.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.0.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>4.0.0.RELEASE</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.0.0.RELEASE</version>
</dependency>
```

问题是如果我们想要将这些 jar 包的版本统一升级为 4.1.1，是不是要手动一个个修改呢？显然，我们有统一配置的方式：

```
<!--统一管理当前模块的 jar 包的版本-->
<properties>
    <spring.version>4.0.0.RELEASE</spring.version>
</properties>
```

```
……
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
</dependency>
```

这样一来，进行版本调整的时候只改一改地方就行了。

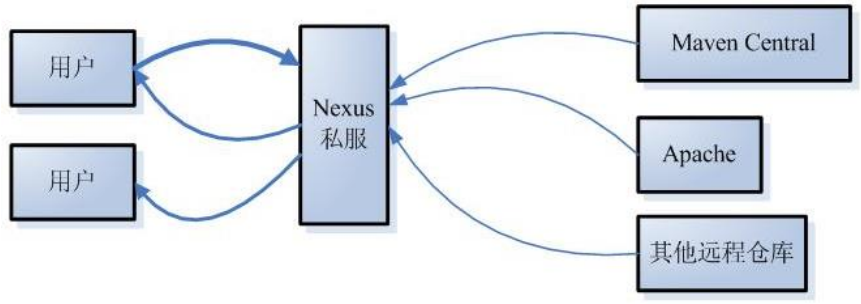
4.6 仓库

1）分类

（1）本地仓库：为当前本机电脑上的所有 Maven 工程服务。

（2）远程仓库

a）私服：架设在当前局域网环境下，为当前局域网范围内的所有 Maven 工程服务。



b）中央仓库：架设在 Internet 上，为全世界所有 Maven 工程服务。

c）中央仓库的镜像：架设在各个大洲，为中央仓库分担流量。减轻中央仓库的压力，同时更快的响应用户请求。

2）仓库中的文件

（1）Maven 的插件

（2）我们自己开发的项目的模块

（3）第三方框架或工具的 jar 包

※不管是什么样的 jar 包，在仓库中都是按照坐标生成目录结构，所以可以通过统一的方式查询或依赖。

4.7 生命周期

1）什么是 Maven 的生命周期？

Maven 生命周期定义了各个构建环节的执行顺序，有了这个清单，Maven 就可以自动化的执行构建命令了。

Maven 有三套相互独立的生命周期，分别是：

- Clean Lifecycle 在进行真正的构建之前进行一些清理工作。
- Default Lifecycle 构建的核心部分，编译，测试，打包，安装，部署等等。
- Site Lifecycle 生成项目报告，站点，发布站点。

再次强调一下它们是**相互独立的**，你可以仅仅调用 clean 来清理工作目录，仅仅调用 site 来生成站点。当然你也可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。比如，运行 mvn clean，这个 clean 是 Clean 生命周期的一个阶段。有 Clean 生命周期，也有 clean 阶段。

2）clean 生命周期

Clean 生命周期一共包含了三个阶段：

- pre-clean 执行一些需要在 clean 之前完成的工作
- clean 移除所有上一次构建生成的文件

- post-clean 执行一些需要在 clean 之后立刻完成的工作

3）Site 生命周期

- pre-site 执行一些需要在生成站点文档之前完成的工作
- site 生成项目的站点文档
- post-site 执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
- site-deploy 将生成的站点文档部署到特定的服务器上

这里经常用到的是 site 阶段和 site-deploy 阶段，用以生成和发布 Maven 站点，这可是 Maven 相当强大的功能，Manager 比较喜欢，文档及统计数据自动生成，很好看。

4）Default 生命周期

Default 生命周期是 Maven 生命周期中最重要的一个，绝大部分工作都发生在这个生命周期中。这里，只解释一些比较重要和常用的阶段：

validate
generate-sources
process-sources
generate-resources
process-resources 复制并处理资源文件，至目标目录，准备打包。
compile 编译项目的源代码。
process-classes
generate-test-sources
process-test-sources
generate-test-resources
process-test-resources 复制并处理资源文件，至目标测试目录。
test-compile 编译测试源代码。
process-test-classes
test 使用合适的单元测试框架运行测试。这些测试代码不会被打包或部署。
prepare-package
package 接受编译好的代码，打包成可发布的格式，如 JAR。
pre-integration-test
integration-test
post-integration-test
verify
install 将包安装至本地仓库，以让其它项目依赖。
deploy 将最终的包复制到远程的仓库，以让其它开发人员与项目共享或部署到服务器上运行。

5）生命周期与自动化构建

运行任何一个阶段的时候，它前面的所有阶段都会被运行，例如我们运行 mvn install 的时候，代码会被编译，测试，打包。这就是 Maven 为什么能够自动执行构建过程的各个环节的原因。此外，Maven 的插件机制是完全依赖 Maven 的生命周期的，因此理解生命周期至关重要。

4.8 插件和目标

（1）Maven 的核心仅仅定义了抽象的生命周期，具体的任务都是交由插件完成的。

（2）每个插件都能实现多个功能，每个功能就是一个插件目标。

（3）Maven 的生命周期与插件目标相互绑定，以完成某个具体的构建任务。

例如：compile 就是插件 maven-compiler-plugin 的一个功能；pre-clean 是插件 maven-clean-plugin 的一个目标。

第 5 章 继承

5.1 为什么需要继承机制？

由于非 compile 范围的依赖信息是不能在“依赖链”中传递的，所以有需要的工程只能单独配置。例如：

Hello	`<dependency>`
	`<groupId>junit</groupId>`
	`<artifactId>junit</artifactId>`
	`<version>4.0</version>`
	`<scope>test</scope>`
	`</dependency>`

HelloFriend	`<dependency>`
	`<groupId>junit</groupId>`
	`<artifactId>junit</artifactId>`
	`<version>4.0</version>`
	`<scope>test</scope>`
	`</dependency>`

MakeFriend	`<dependency>`
	`<groupId>junit</groupId>`
	`<artifactId>junit</artifactId>`
	`<version>4.0</version>`
	`<scope>test</scope>`
	`</dependency>`

此时如果项目需要将各个模块的 junit 版本统一为 4.9，那么到各个工程中手动修改无疑是非常不可取的。使用继承机制就可以将这样的依赖信息统一提取到父工程模块中进行统一管理。

5.2 创建父工程

父工程的打包方式为 pom

```
<groupId>com.atguigu.maven</groupId>
    <artifactId>Parent</artifactId>
    <packaging>pom</packaging>
<version>1.0-SNAPSHOT</version>
```

父工程只需要保留 pom.xml 文件即可

5.3 在子工程中引用父工程

```
<parent>
    <!-- 父工程坐标 -->
    <groupId>...</groupId>
        <artifactId>...</artifactId>
    <version>...</version>

    <!--指定从当前 pom.xml 文件出发寻找父工程的 pom.xml 文件的相对路径-->
    <relativePath>..</relativePath>
</parent>
```

```
<!--继承-->
<parent>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>Parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--指定从当前 pom.xml 文件出发寻找父工程的 pom.xml 文件的相对路径-->
    <relativePath>../Parent/pom.xml</relativePath>
</parent>
```

此时如果子工程的 groupId 和 version 如果和父工程重复则可以删除。

5.4 在父工程中管理依赖

将 Parent 项目中的 dependencies 标签，用 dependencyManagement 标签括起来

```
<!--依赖管理-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

在子项目中重新指定需要的依赖，删除范围和版本号

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
</dependency>
```

第 6 章 聚合

6.1 为什么要使用聚合？

将多个工程拆分为模块后，需要手动逐个安装到仓库后依赖才能够生效。修改源码后也需要逐个手动进行 clean 操作。而使用了聚合之后就可以批量进行 Maven 工程的安装、清理工作。

6.2 如何配置聚合？

在总的聚合工程中使用 modules/module 标签组合，指定模块工程的相对路径即可

```
<!--聚合-->
<modules>
    <module>../MakeFriend</module>
    <module>../OurFriends</module>
    <module>../HelloFriend</module>
    <module>../Hello</module>
</modules>
```

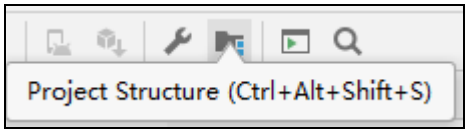
Maven 可以根据各个模块的继承和依赖关系自动选择安装的顺序

第 7 章 通过 Maven 创建 Web 工程

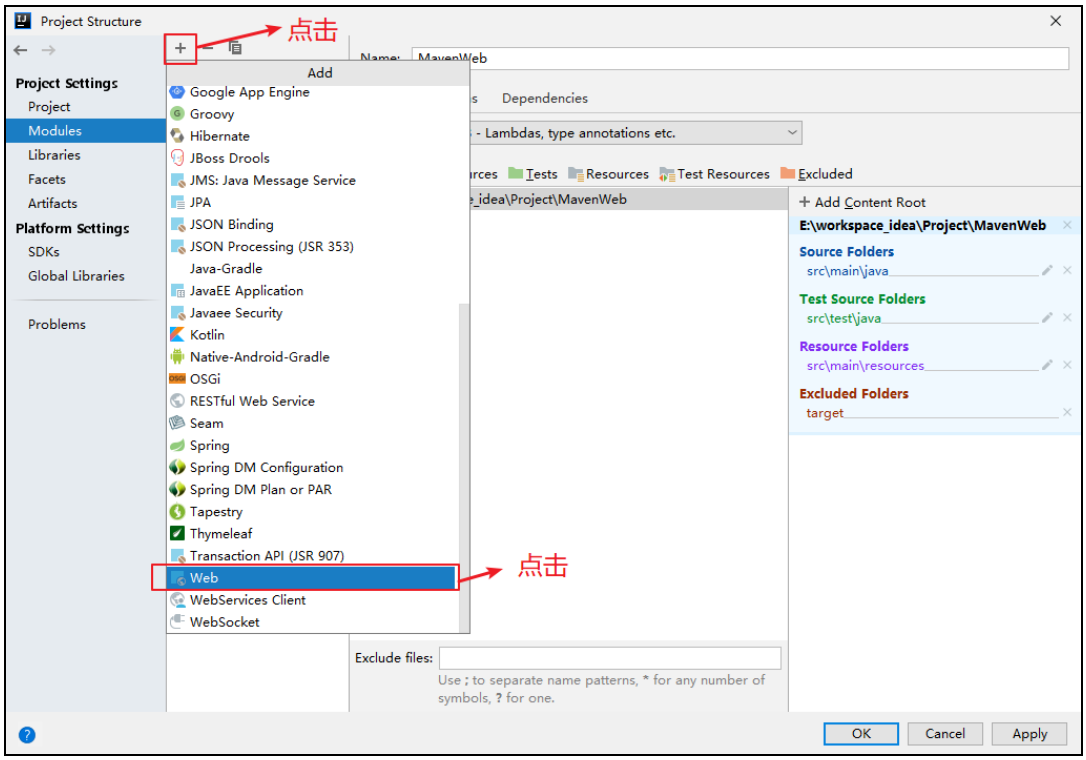
1）创建简单的 Maven 工程，打包方式为 war 包

```
<groupId>com.atguigu.maven</groupId>
<artifactId>MavenWeb</artifactId>
<packaging>war</packaging>
<version>1.0-SNAPSHOT</version>
```

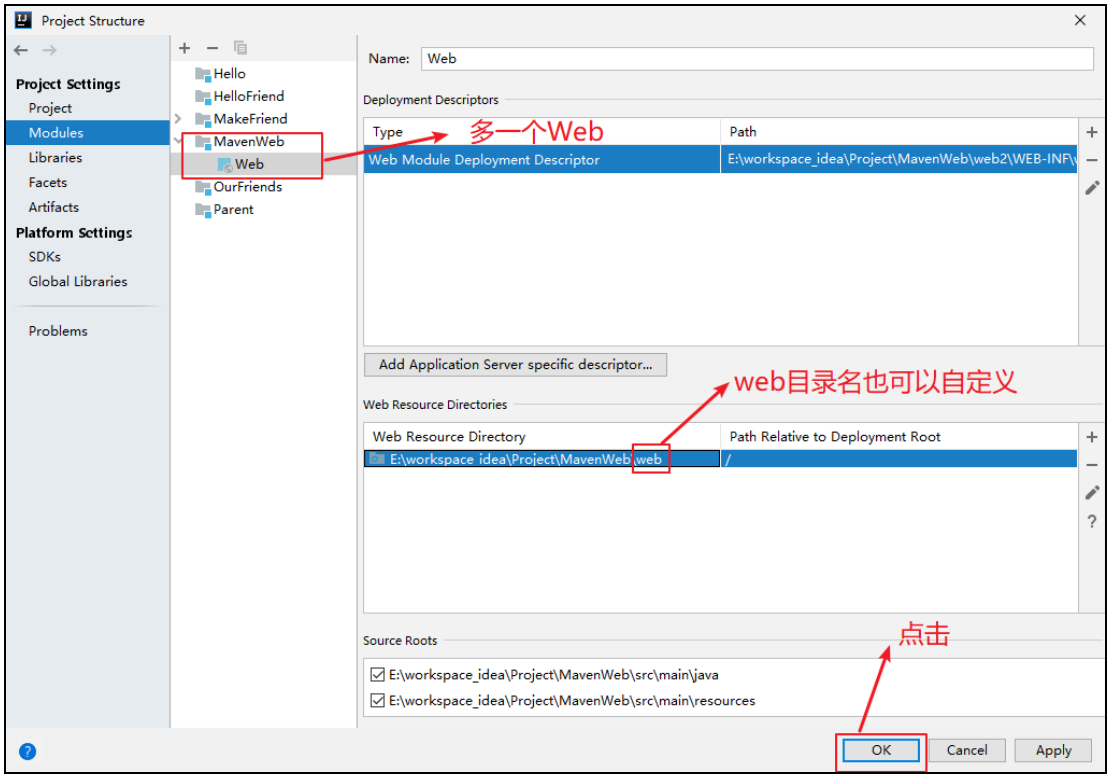
2）点击 Project Structure



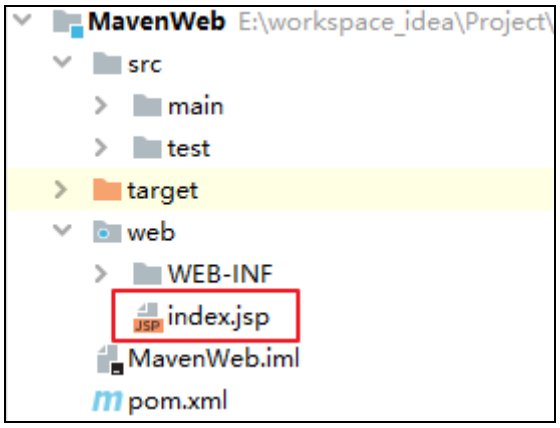
3）选择对应的 Module，添加 web 目录



4）设置目录名称



5）在 web 目录下创建 index.jsp 页面



6）部署到 Tomcat 上运行

第 8 章 Maven 酷站

我们可以到 http://mvnrepository.com/搜索需要的 jar 包的依赖信息。

http://search.maven.org/