1. Maven的概念
   1. 什么是Maven

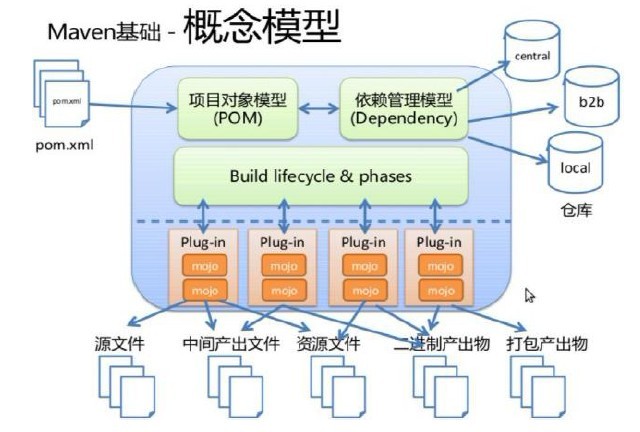
* 翻译为“专家”，“内行”
* Maven是**跨平台**的**项目管理**工具。主要服务于基于Java平台的项目构建，依赖管理和项目信息管理。
* 什么是理想的项目构建？
  + 高度自动化，跨平台，可重用的组件，标准化的
* 什么是依赖？为什么要进行依赖管理？
  + 自动下载，统一依赖管理
* 有哪些**项目信息**？
  + 项目名称描述等，开发人员信息，开发者信息等
  1. 什么是构建



* 1. Maven的应用场景



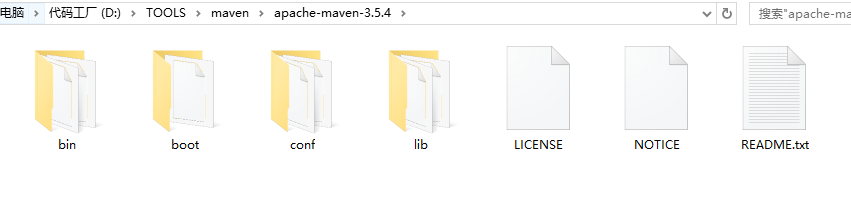
* 1. Maven模型



* 1. 为什么使用Maven
* IDE？Eclipse？IDEA？
  + 手工操作较多，编译、测试、部署等工作都是独立的，很难一步完成
  + 每个人的IDE配置都不同，很容易出现本地代码换个地方编译就出错
* Ant？
  + 没有一个约定的目录结构
  + 必须明确让ant做什么，什么时候做，然后编译，打包
  + 没有生命周期，必须定义目标及其实现的任务序列
  + 没有集成依赖管理
* Maven？
  + 拥有约定，知道你的代码在哪里，放到哪里去
  + 拥有一个生命周期，例如执行 mvn install 就可以自动执行编译，测试，打包等构建过程
  + 只需要定义一个pom.xml,然后把源码放到默认的目录，Maven帮你处理其他事情
  + 拥有依赖管理，仓库管理
  1. 依赖管理
* 对第三方依赖包的管理
  + 可以连接互联网下载项目所需第三方jar包
* 对自己开发的模块的管理
  + 可以像引用第三方依赖包一样引用自己项目的依赖包

1. 初识Maven
   1. Maven安装
      1. 安装准备

* 安装jdk1.8或以上版本
  + 略
* 下载Maven
  + - 地址：http://maven.apache.org/download.html
    - 版本：Maven3.5
* 安装Maven
  + - 解压到相应目录即可(无中文路径)
    1. Maven安装目录分析



bin：含有mvn运行的脚本

boot：含有plexus-classworlds类加载器框架

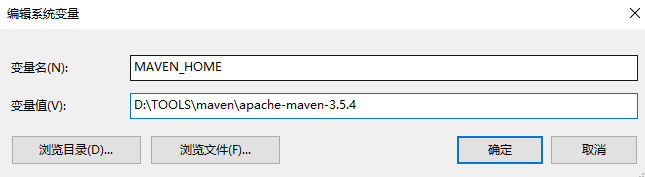
conf：含有settings.xml配置文件

lib：含有Maven运行时所需要的java类库

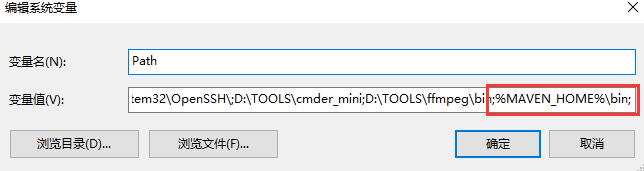
LICENSE.txt, NOTICE.txt, README.txt针对Maven版本，第三方软件等简要介绍

* + 1. Maven的环境变量
* 设置MAVEN\_HOME环境变量
* 升级时只需要下载最新版本，解压缩重新设置MAVEN\_HOME环境变量即可
* 设置MAVEN\_OPTS环境变量(可省略)
* -Xms128m -Xmx512m
* 配置用户范围的settings.xml
* %MAVEN\_HOME%/conf/settings.xml 全局的
* ~/.m2/settings.xml 单个用户的

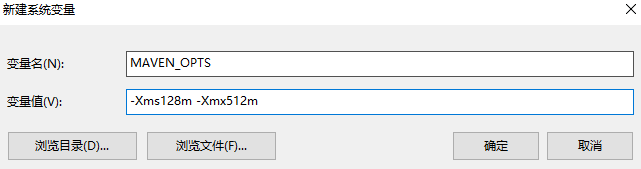
设置MAVEN\_HOME环境变量,把maven的解压后的路径添加为变量值



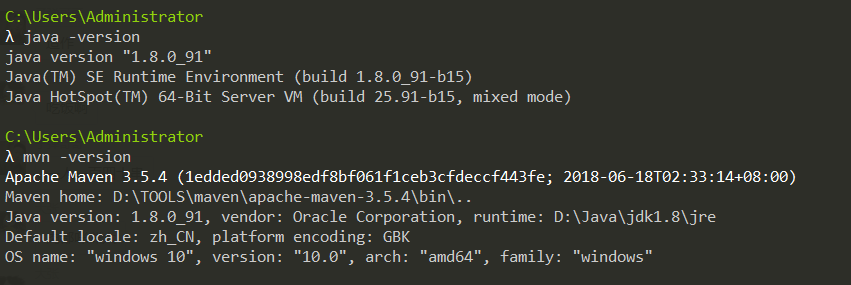
设置Path环境变量,把 %MAVEN\_HOME%\bin 添加到Path的变量值末尾,与前面的内容用 ; 隔开,如图:



设置MAVEN\_OPTS环境变量,把-Xms128m -Xmx512m添加为变量值



验证配置是否完成



* 1. Maven的第一个项目
     1. 按照约定创建Maven项目目录
* src/main/java —— 存放项目的.java文件
* src/main/resources —— 存放项目资源文件，如spring, hibernate配置文件
* src/test/java —— 存放所有测试.java文件，如JUnit测试类
* src/test/resources —— 测试资源文件
* target —— 项目输出位置
* pom.xml
  + 1. 在项目Hello根目录建立pom.xml

<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>cn.tx.maven</groupId>

<artifactId>Hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Hello</name>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.9</version>

<scope>test</scope>

</dependency>

</dependencies>

</project>

* + 1. 创建Hello.java文件

在src/main/java/cn/tx/maven目录下新建文件Hello.java

package cn.tx.maven;

public class Hello {

public String sayHello(String name){

return "Hello "+name+"!";

}

}

* + 1. 创建HelloTest.java文件

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

package cn.tx.maven;

import org.junit.Test;

import static junit.framework.Assert.\*;

public class HelloTest {

@Test

public void testHello(){

Hello hello = new Hello();

String results = hello.sayHello("maven");

assertEquals("Hello maven!",results);

}

}

* + 1. 进行maven的项目操作

1. 打开cmd命令行，进入Hello项目根目录执行 mvn compile命令，查看根目录变化
2. cmd 中继续录入mvn clean命令，然后再次查看根目录变化
3. cmd 中录入 mvn clean compile命令, 查看根目录变化
4. cmd 中录入 mvn clean test命令，查看根目录变化
5. cmd 中录入 mvn clean package命令,查看根目录变化
6. cmd 中录入 mvn clean install 查看仓库会把项目安装到仓库里
   1. Maven的第二个项目
      1. 按照约定创建Maven项目目录

HelloFriend

--src

-----main

----------java

----------resources

-----test

---------java

---------resources

--pom.xml

* + 1. 在项目HelloFriend根目录建立pom.xml

在项目HelloFriend根目录建立pom.xml

<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>cn.tx.maven</groupId>

<artifactId>HelloFriend</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>HelloFriend</name>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.9</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>cn.tx.maven</groupId>

<artifactId>Hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<scope>compile</scope>

</dependency>

</dependencies>

</project>

* + 1. 创建HelloFriend.java文件

在src/main/java/cn/tx/maven目录下新建文件HelloFriend.java

package cn.tx.maven;

import cn.tx.maven.Hello;

public class HelloFriend {

public String sayHelloToFriend(String name){

Hello hello = new Hello();

String str = hello.sayHello(name)+" I am "+this.getMyName();

System.out.println(str);

return str;

}

public String getMyName(){

return "John";

}

}

* + 1. 创建HelloFriendTest.java文件

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

package cn.tx.maven;

import static junit.framework.Assert.assertEquals;

import org.junit.Test;

import cn.tx.maven.Hello;

public class HelloFriendTest {

@Test

public void tesHelloFriend(){

HelloFriend helloFriend = new HelloFriend();

String results = helloFriend.sayHelloToFriend("zhangsan");

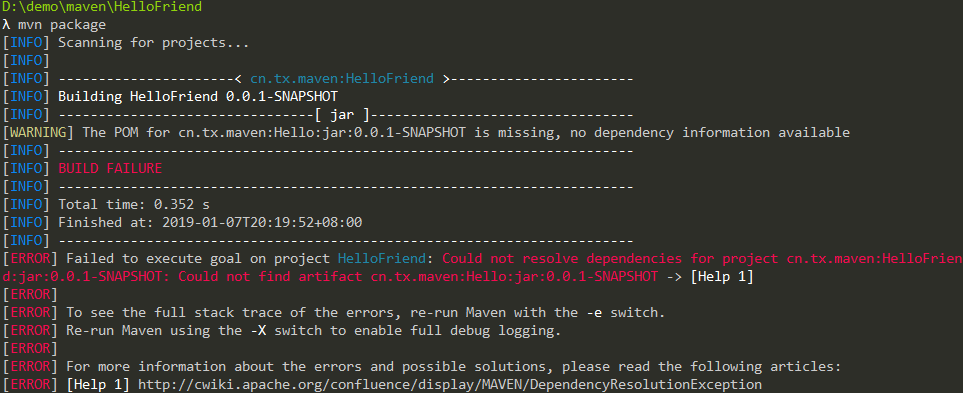
assertEquals("Hello zhangsan! I am John",results);

}

}

* + 1. 测试打包HelloFriend项目

在HelloFriend目录下执行命令mvn package

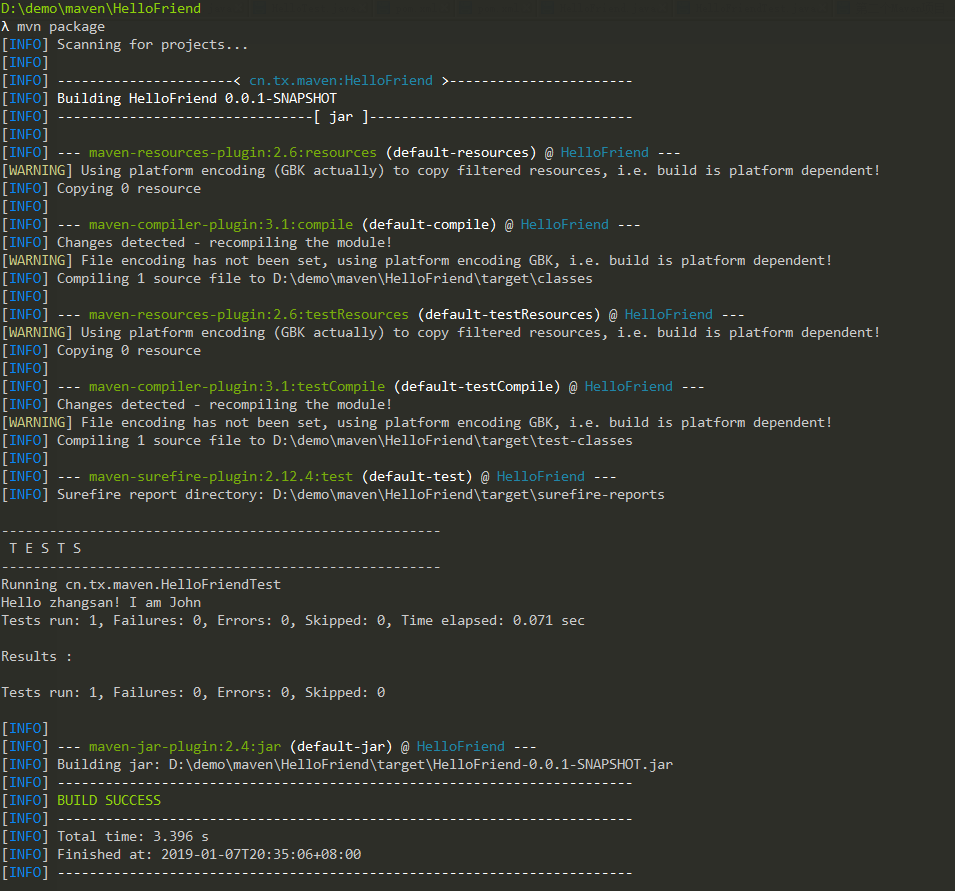


系统报错提示:Could not find artifact cn.tx.maven:Hello:jar:0.0.1-SNAPSHOT,表示无法找到HelloFriend所依赖的Hello 0.0.1-SNAPSHOT.jar

故此我们需要重新构建第一个项目Hello并安装到数据仓库，在命令行Hello根目录下执行mvn clean install



对Hello项目进行清理并安装后,对HelloFriend项目再次进行打包



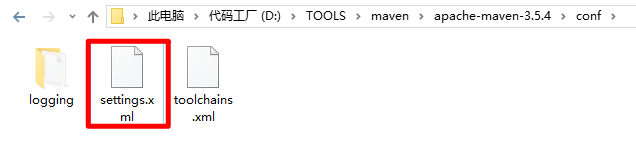
可以看到本次打包成功,说明HelloFriend项目已经可以加载Hello项目所打包的jar包

* 1. Maven仓库配置
     1. Maven仓库概念
* 何为Maven仓库？
  + 用来统一存储所有Maven共享构建的位置就是仓库
* Maven仓库布局
  + 根据Maven坐标定义每个构建在仓库中唯一存储路径
  + groupId/artifactId/version/artifactId-version.packaging
* 仓库的分类
  + **本地仓库**
    - **~/.m2/repository/ 可修改**
    - **每个用户只有一个本地仓库**
  + 远程仓库
    - 中央仓库：Maven默认的远程仓库

http://repo1.maven.org/maven2

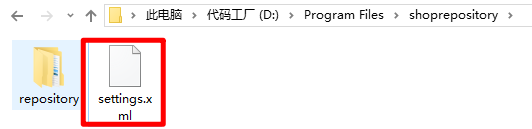
* + - 私服：是一种特殊的远程仓库，它是架设在局域网内的仓库
    1. Maven本地仓库配置
       1. 全局本地仓库(所有操作系统用户均影响)

%MAVEN\_HOME%\conf\settings.xml文件,修改该文件会影响所有使用该Maven的用户的本地仓库,例如:



* + - 1. 用户本地仓库(仅影响使用该配置文件的用户或程序)

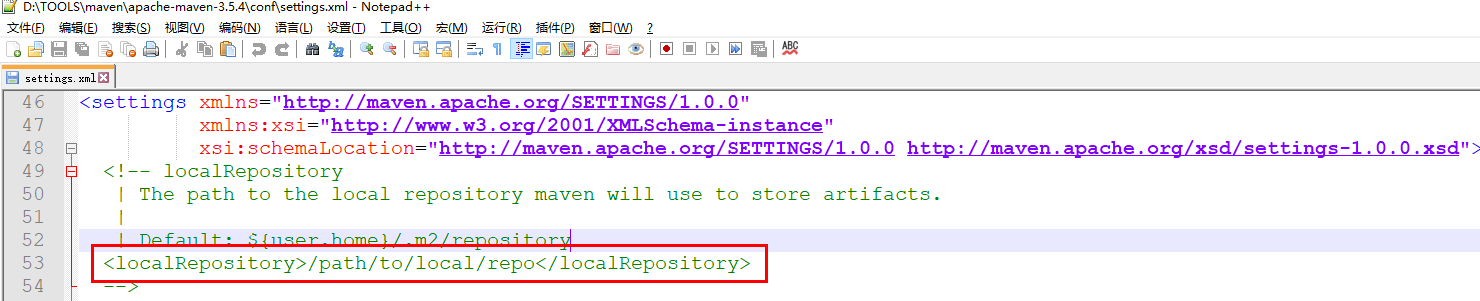
该文件可以在任意目录,例如:



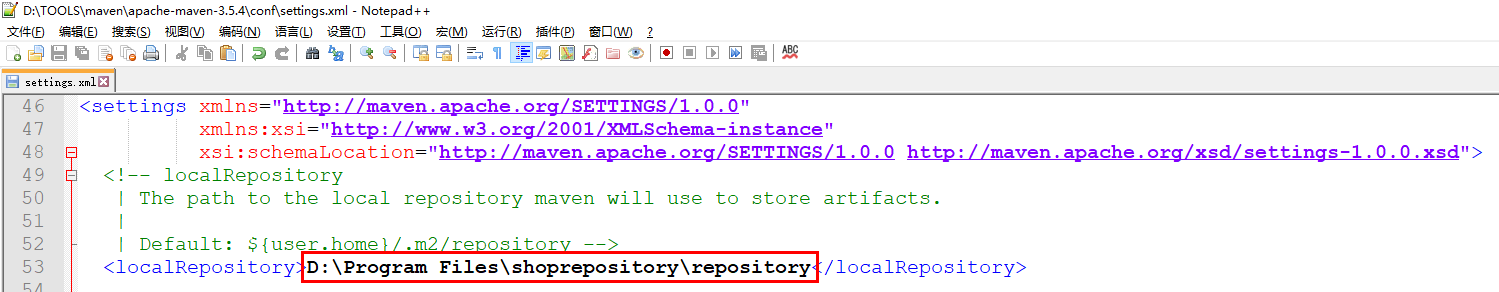
* + - 1. 修改配置文件设置Maven本地仓库(以全局设置为例)

编辑D:\TOOLS\maven\apache-maven-3.5.4\conf\settings.xml文件

找到如下位置:



修改为:



保存退出即可

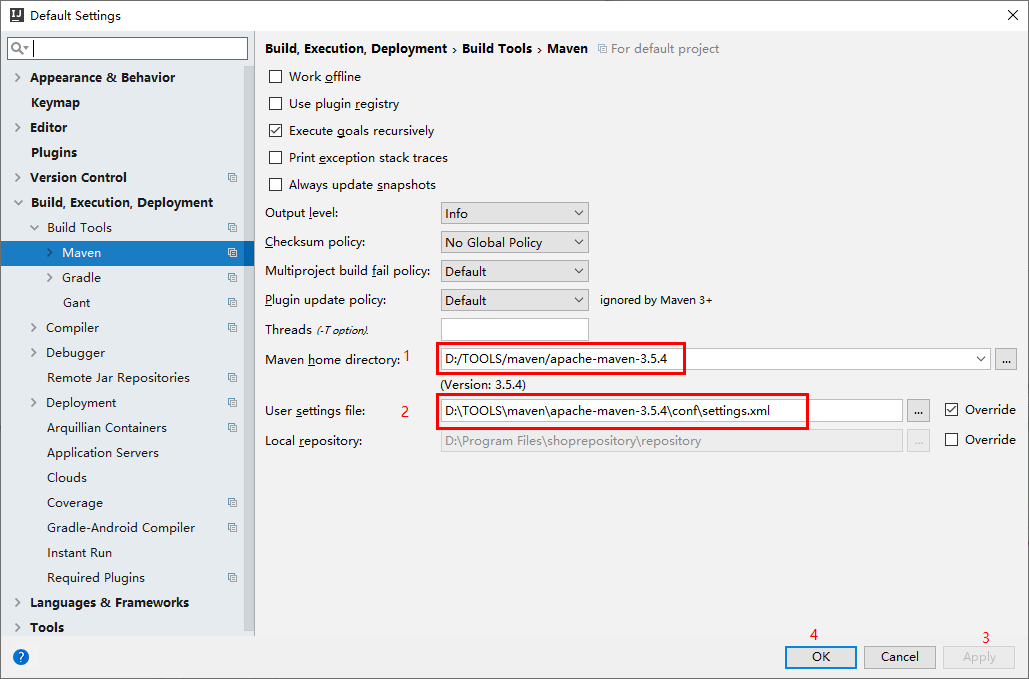
以上配置表示所有使用该Maven的程序或用户,如未单独配置,则默认加载以上目录的本地库文件;

Ps:课程提供本地库文件为shoprapository.zip,如果遇到打包不成功,jar包无法下载等情况,优先进行本地库配置,避免因网络导致的maven依赖包问题.

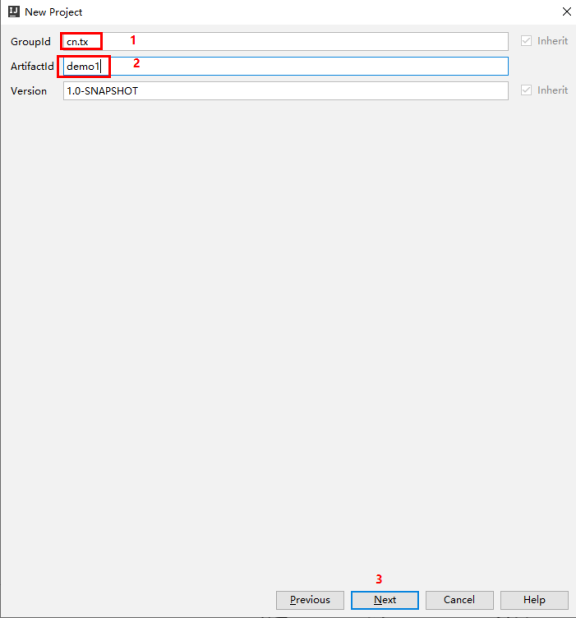
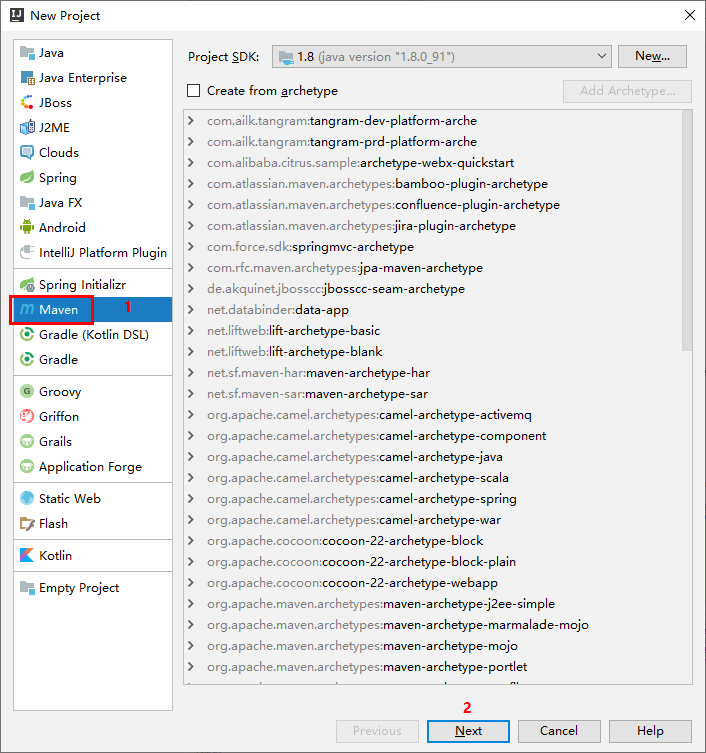
* 1. IDEA整合Maven项目
     1. IDEA的Maven全局配置

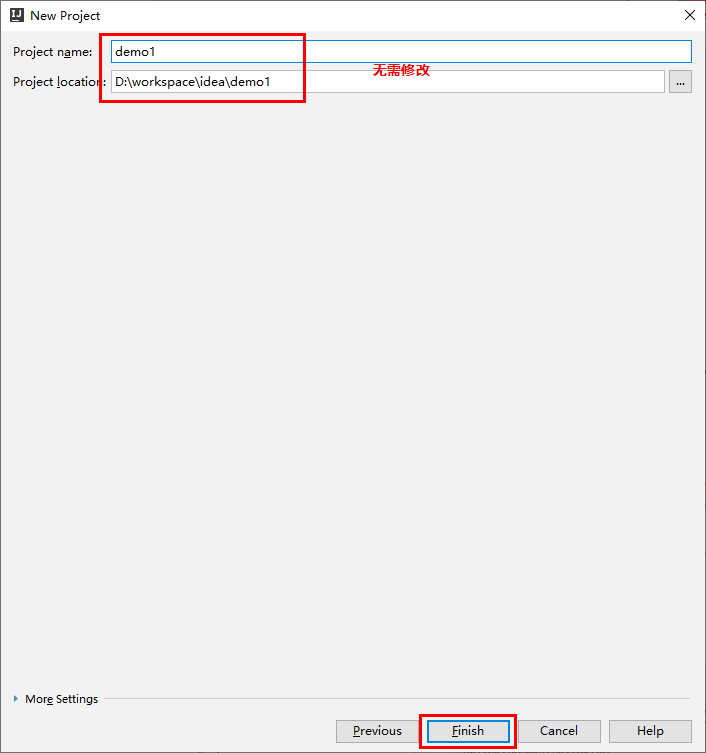
启动IDEA后找到Settings



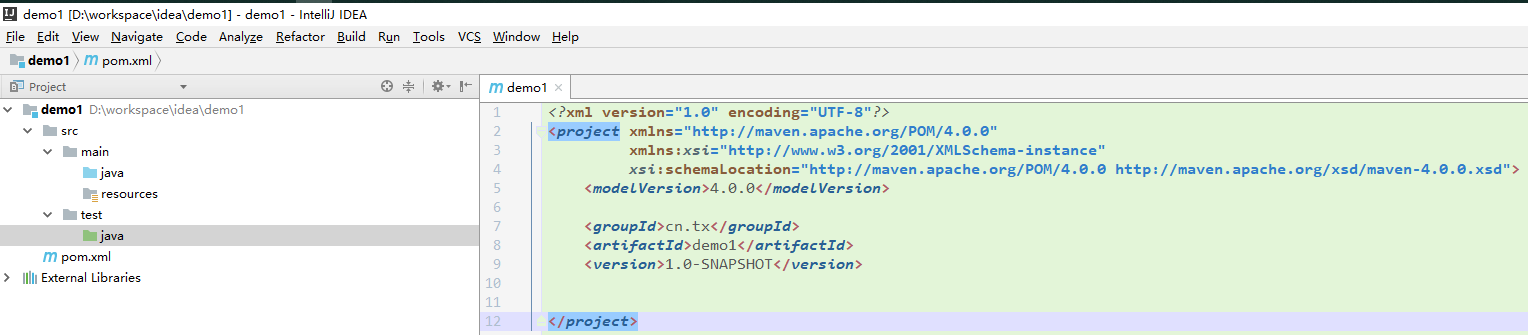
以上设置完成表示在本idea上,所有项目如未对Maven的配置进行单独配置,则默认均使用该Maven版本和本地库设置;

* + 1. IDEA创建Maven的Java项目

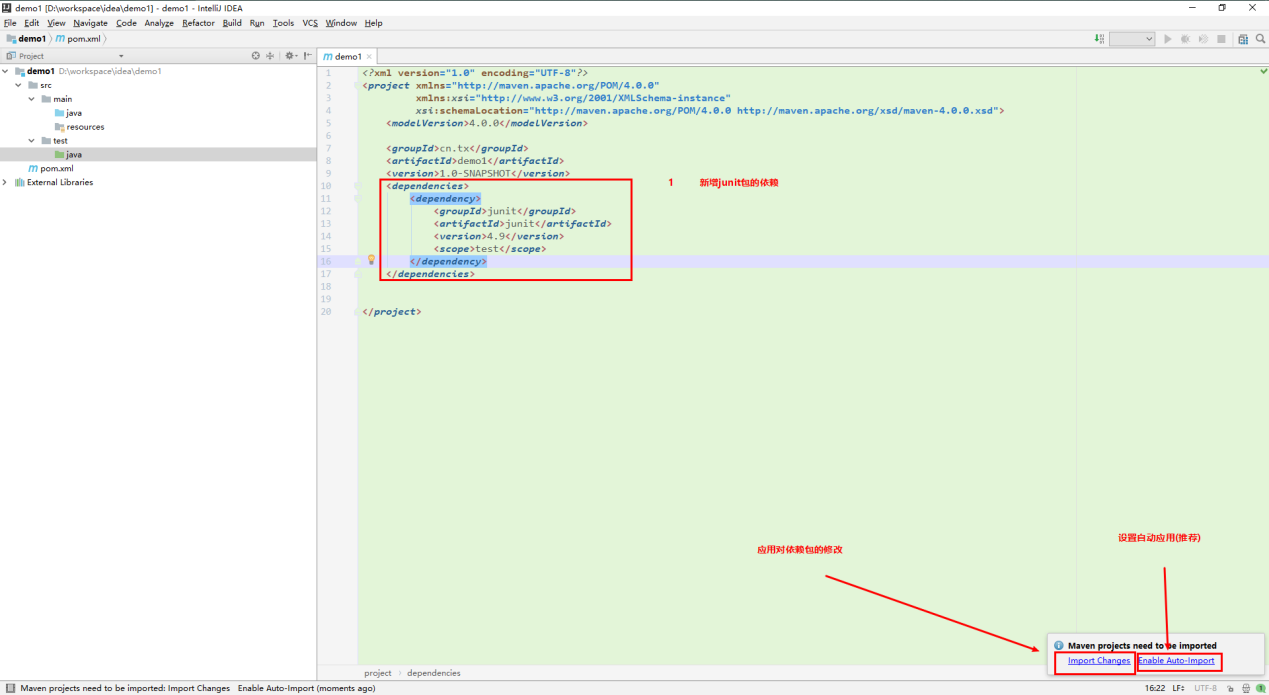


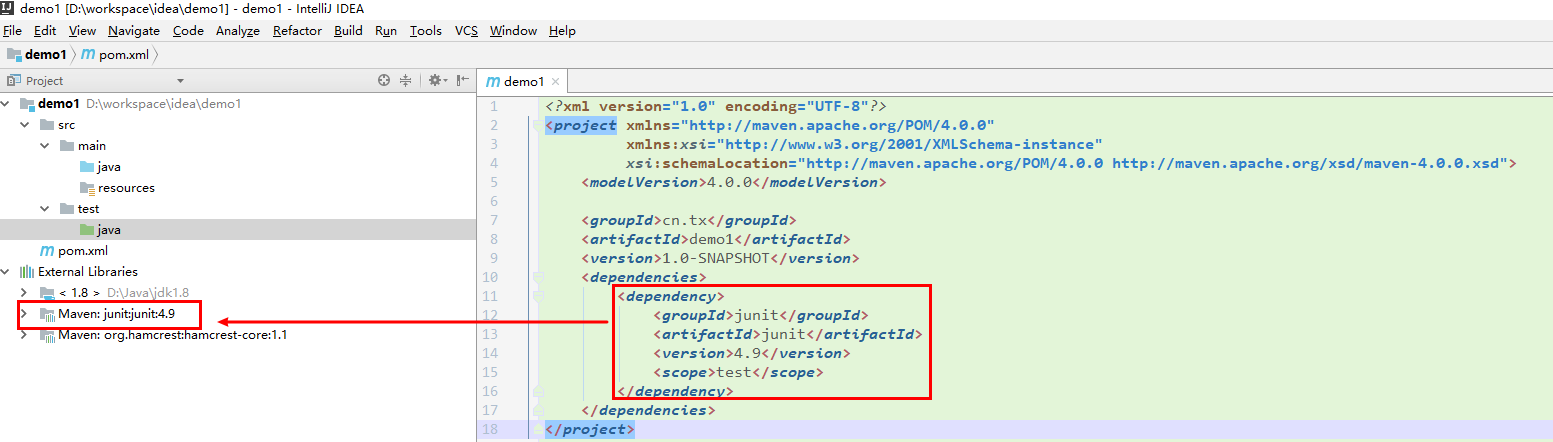


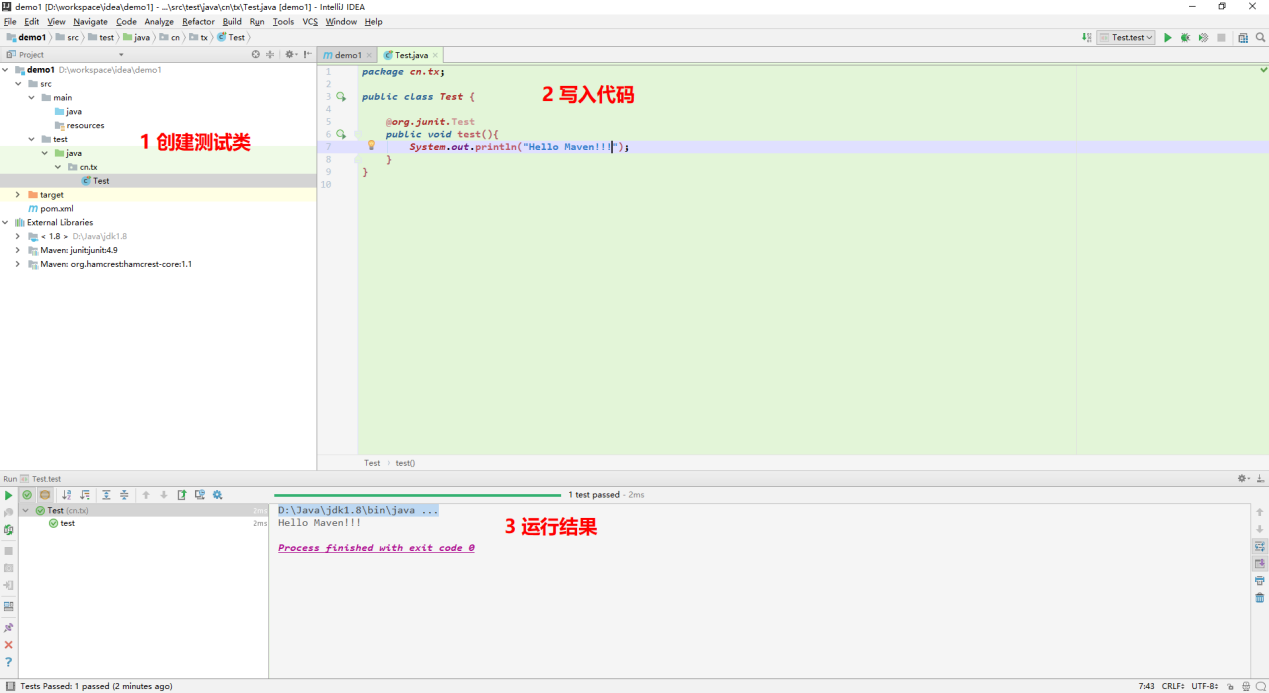
至此Maven的Java项目创建完成



测试Maven的jar包引入

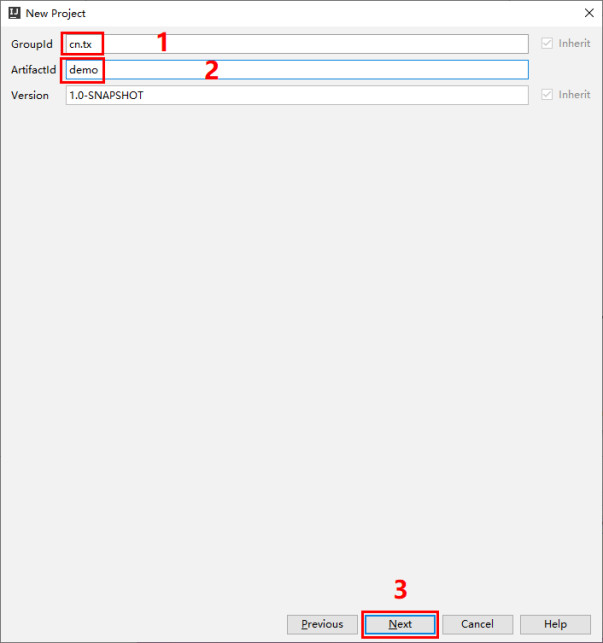
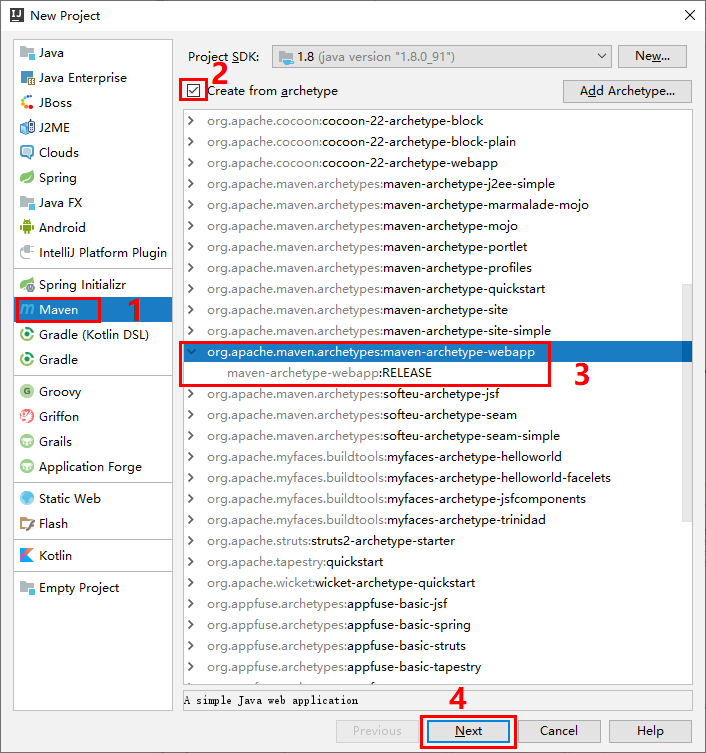


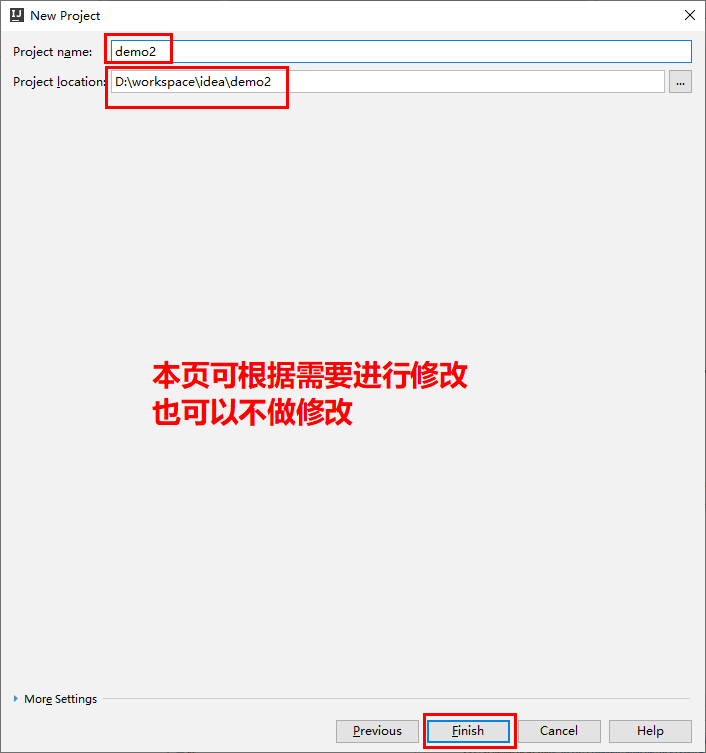
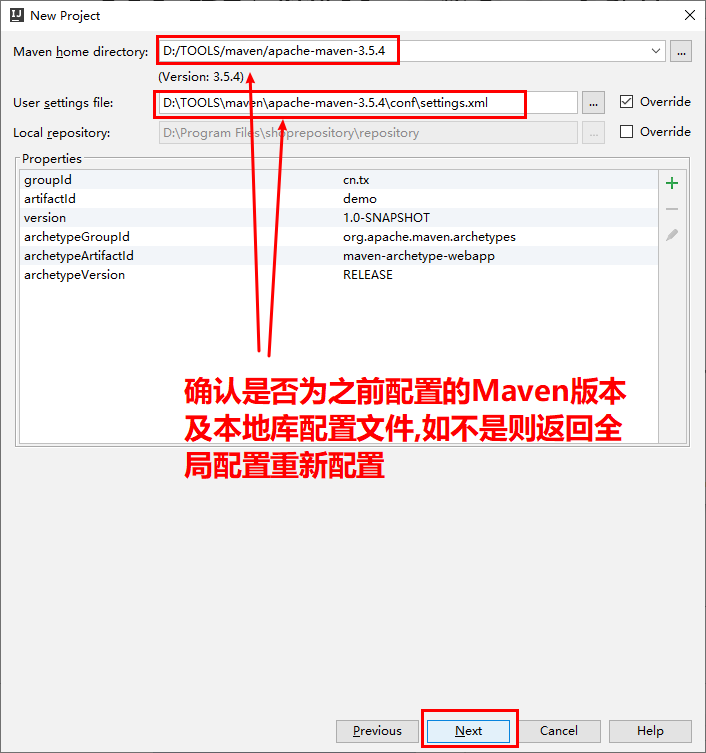




以上,在IDEA上使用Maven进行Java项目的创建及测试已完成!

* + 1. IDEA创建Maven的Java web项目

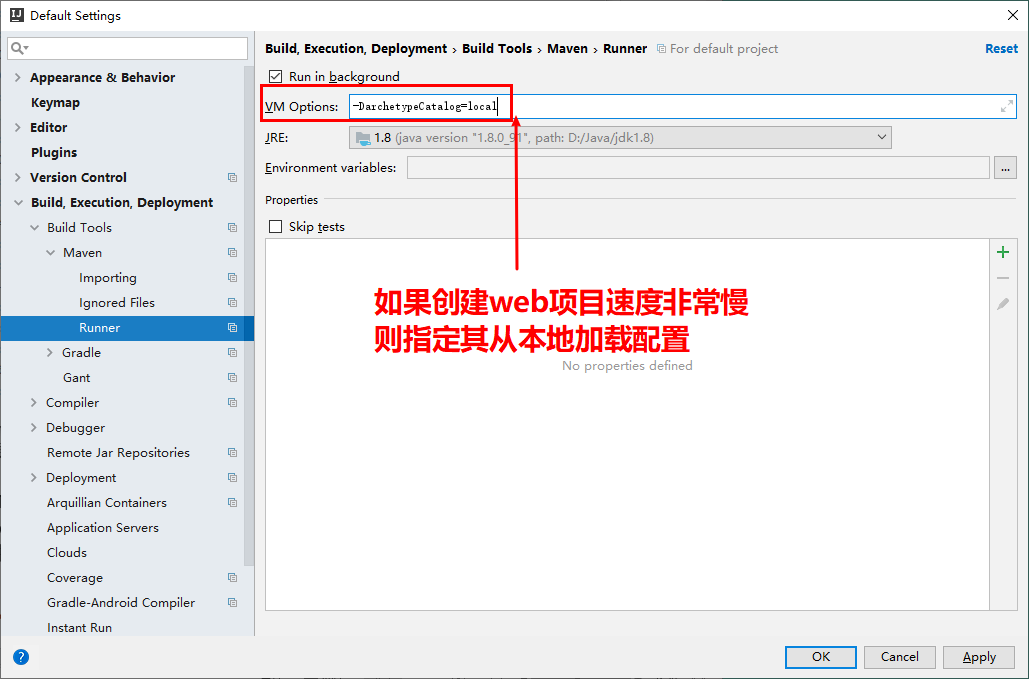


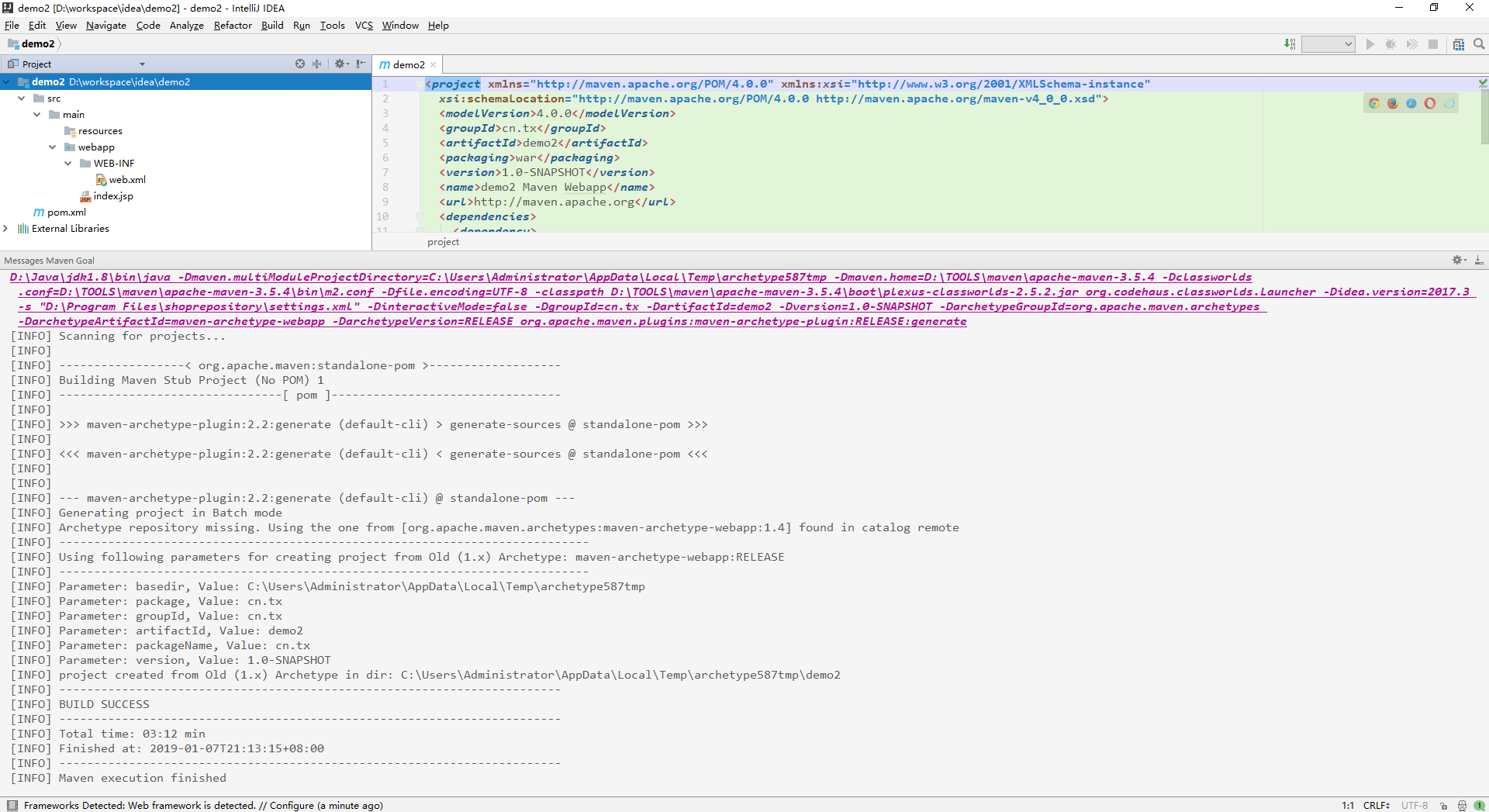


配置完毕后会进行web项目的初始化,初始化过程取决于本地库是否已生成及网络情况,推荐使用课程提供的Maven本地库

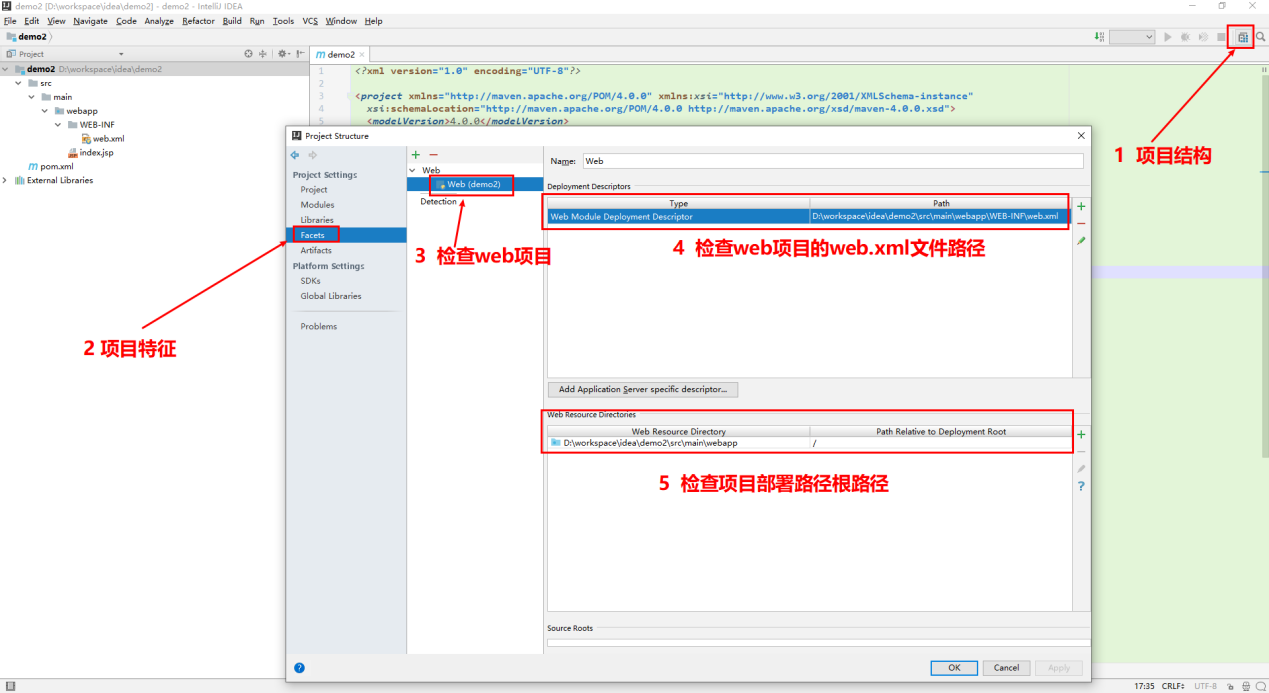
如果创建项目速度非常慢,并且在Generating project in Batch mode卡住,则进行全局设置如下:

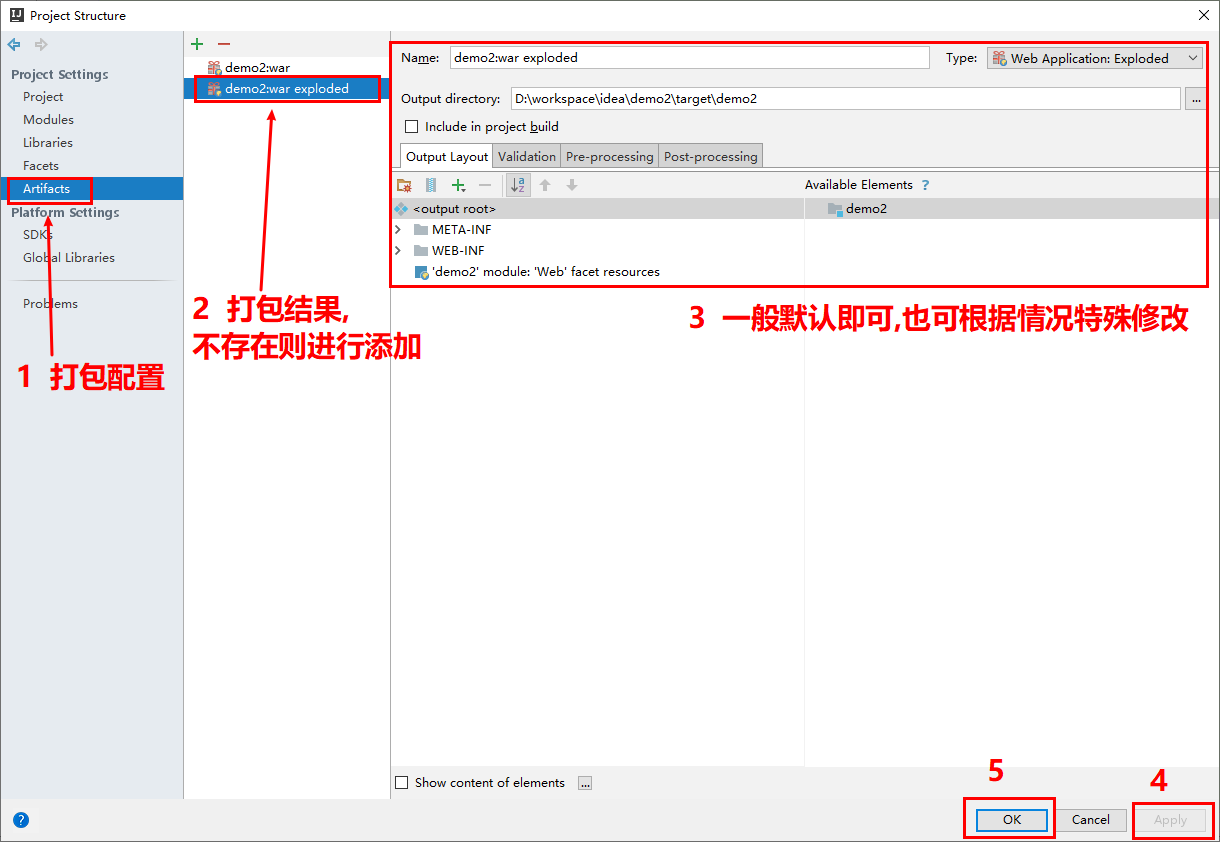
-DarchetypeCatalog=local



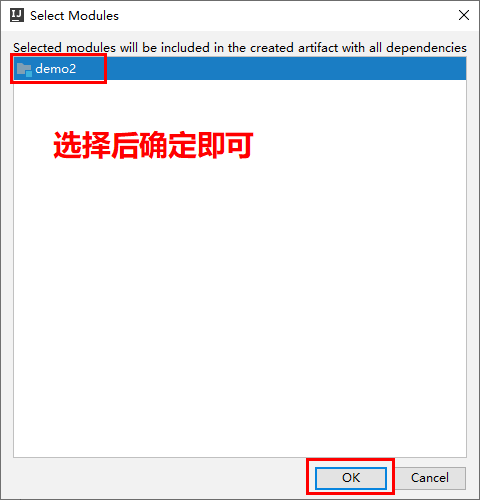
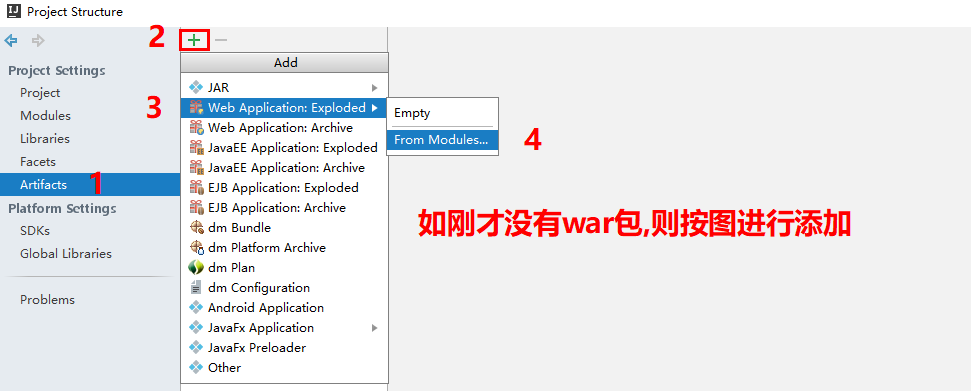


以上web项目则创建成功,接下来需要进行一些配置来确定web项目可以正常运行





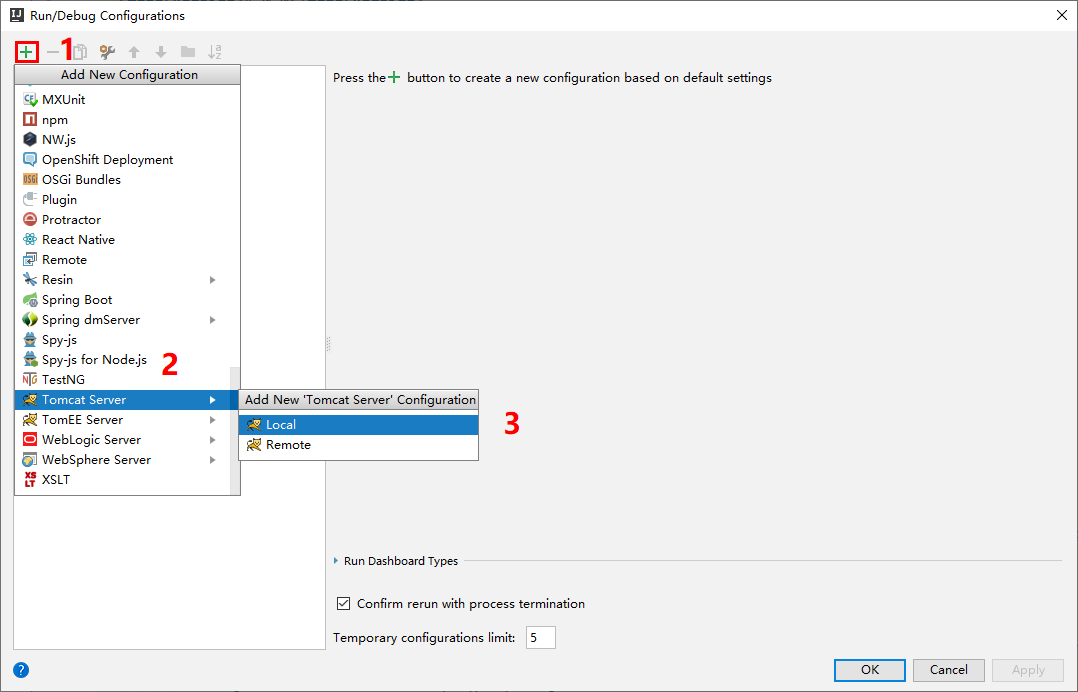
添加war包的方法:

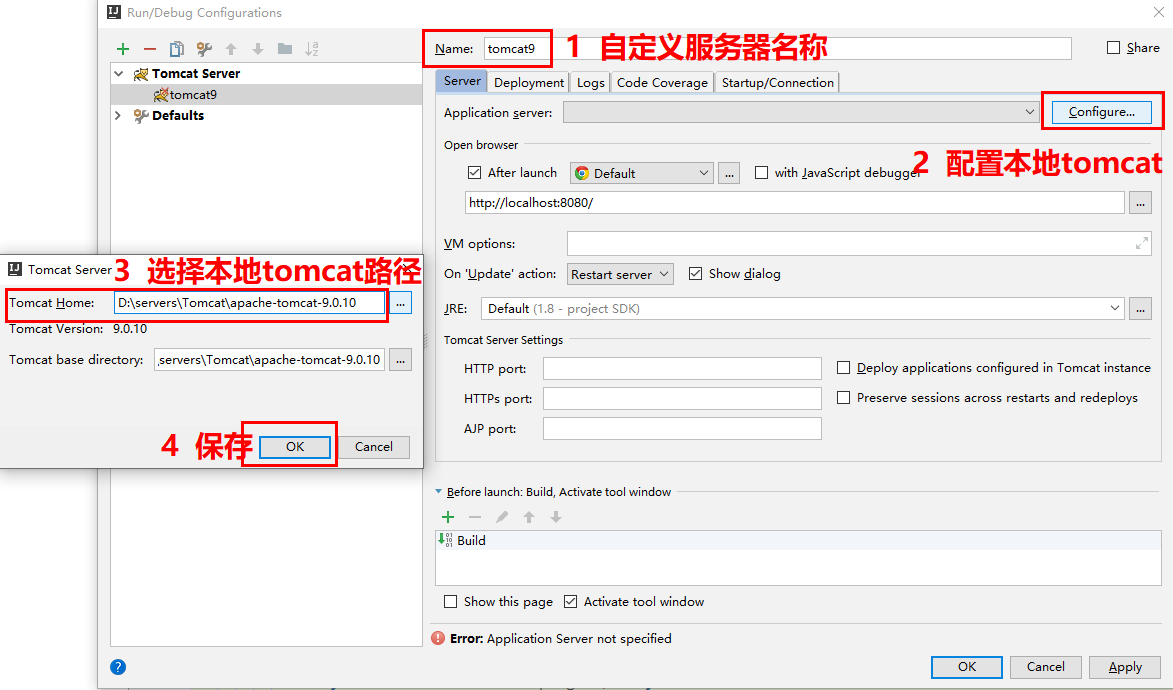


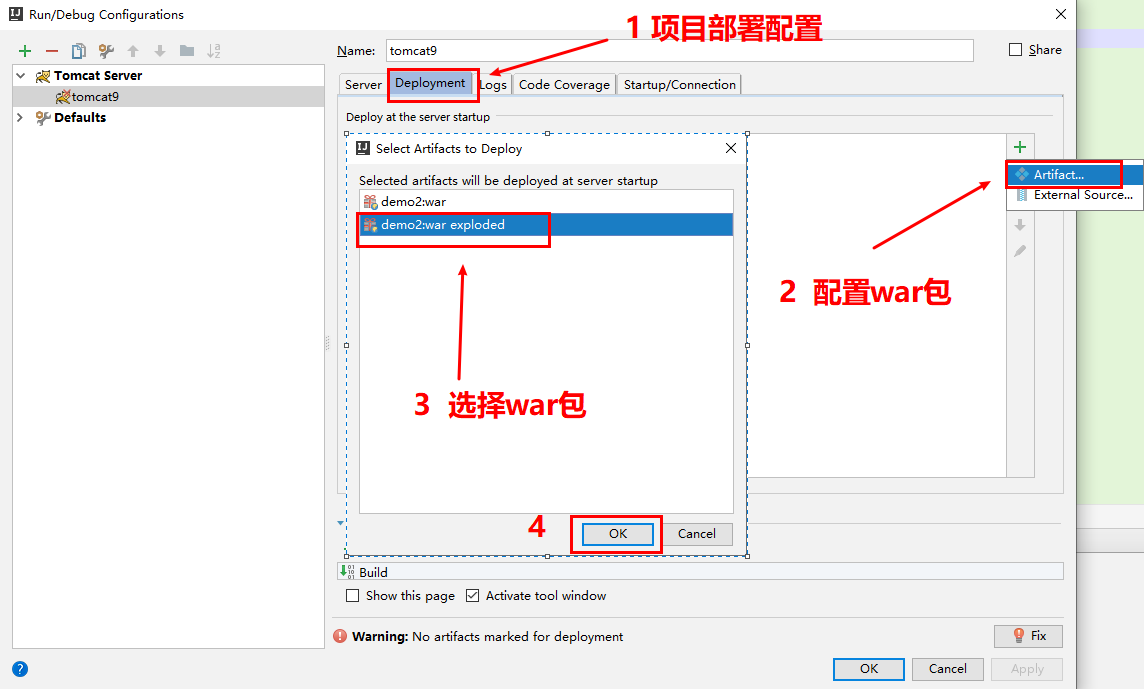
配置web服务器

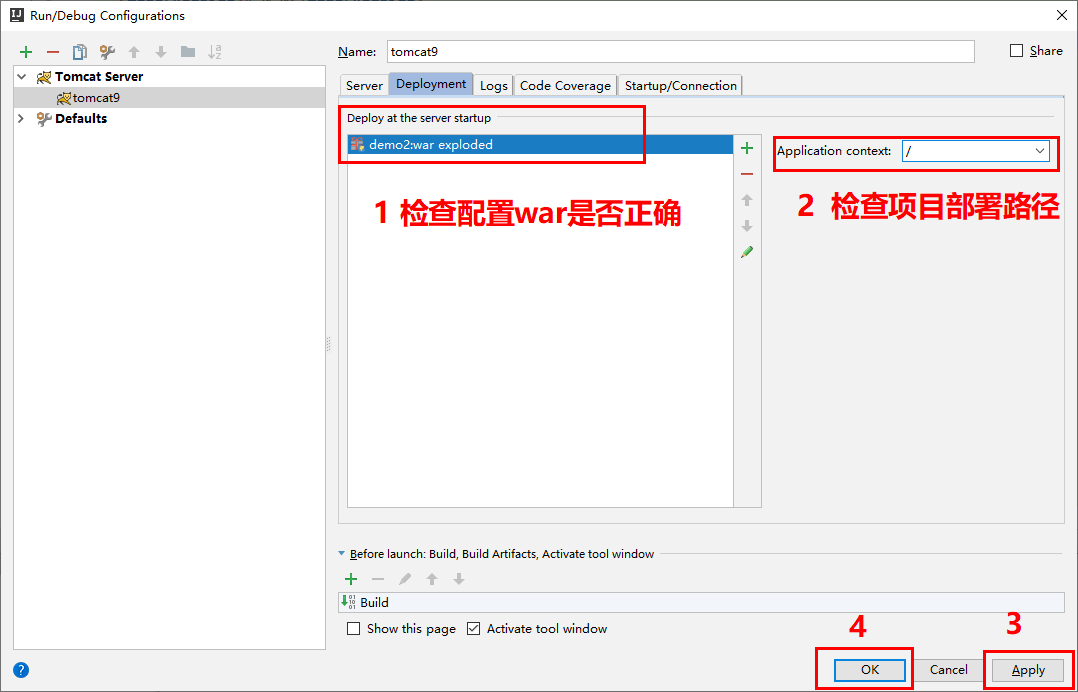


添加本地tomcat配置

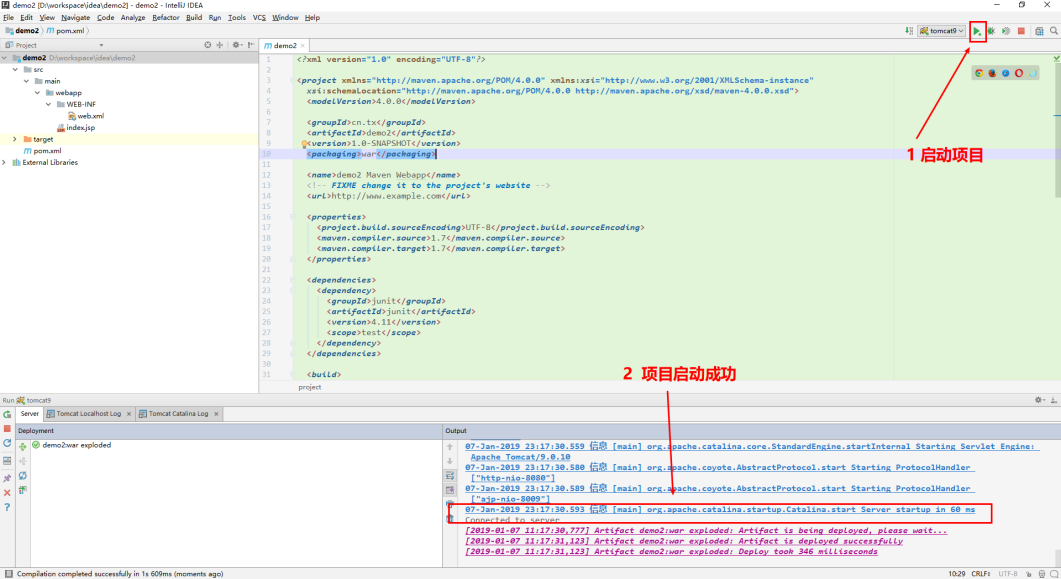


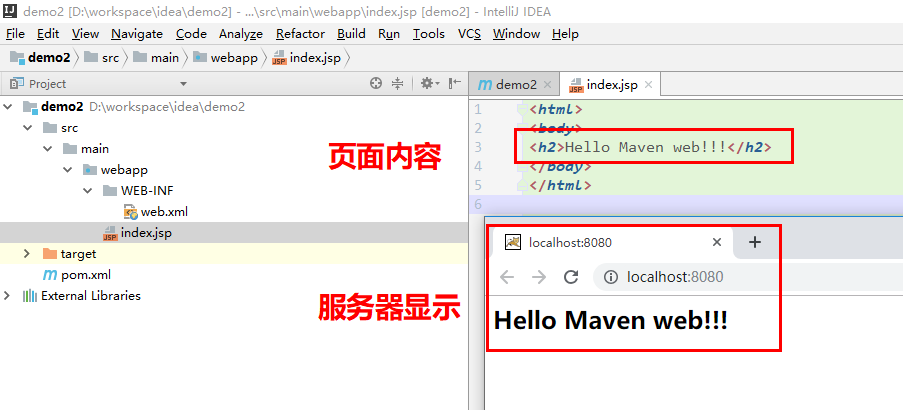






测试项目运行

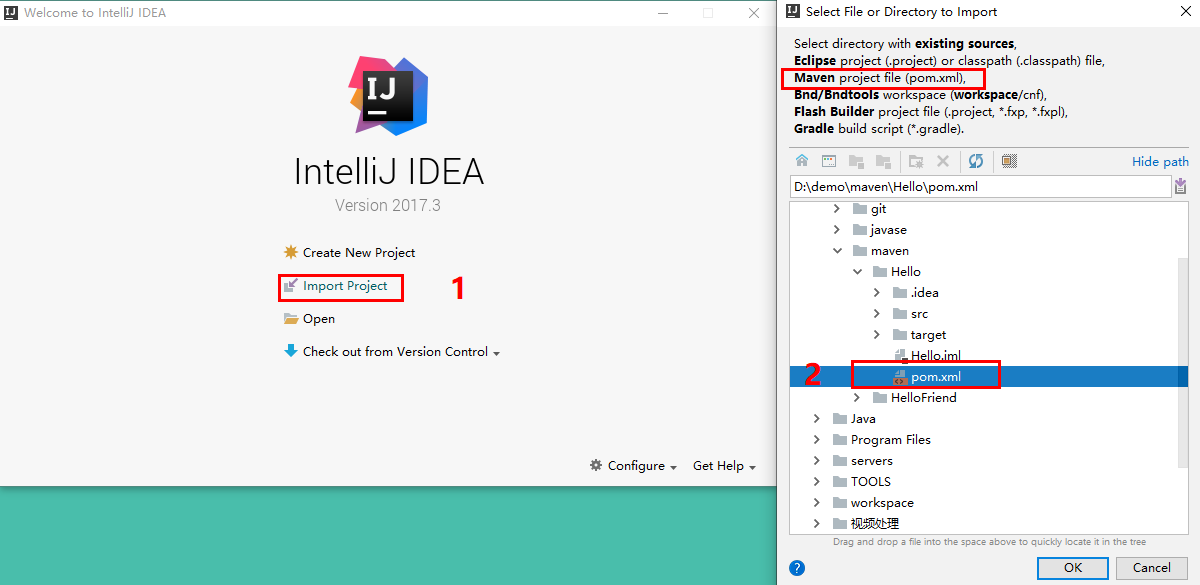


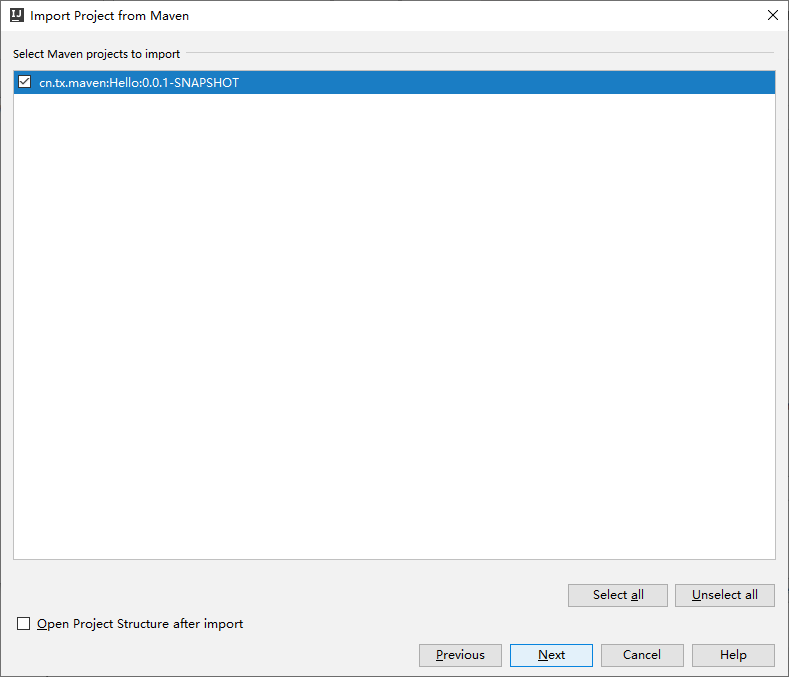
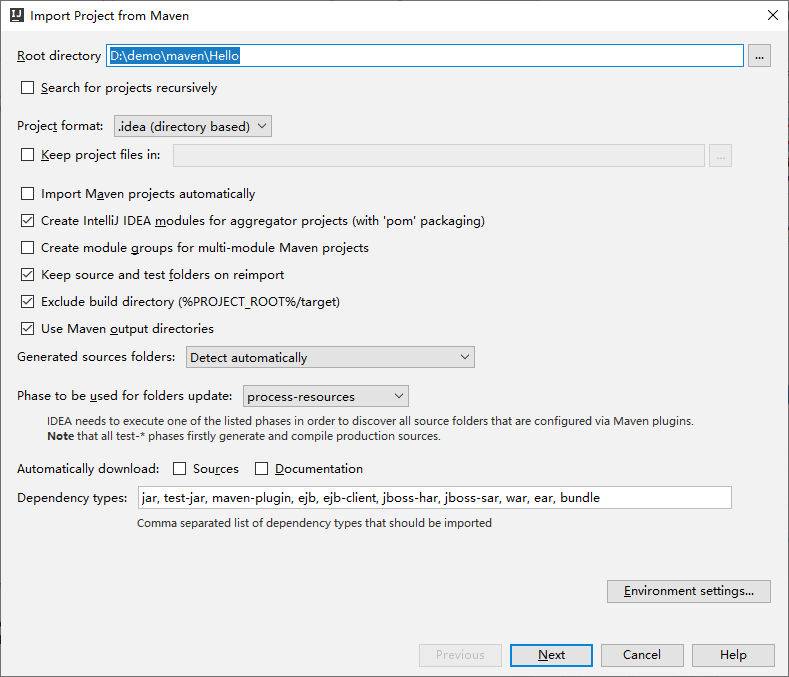


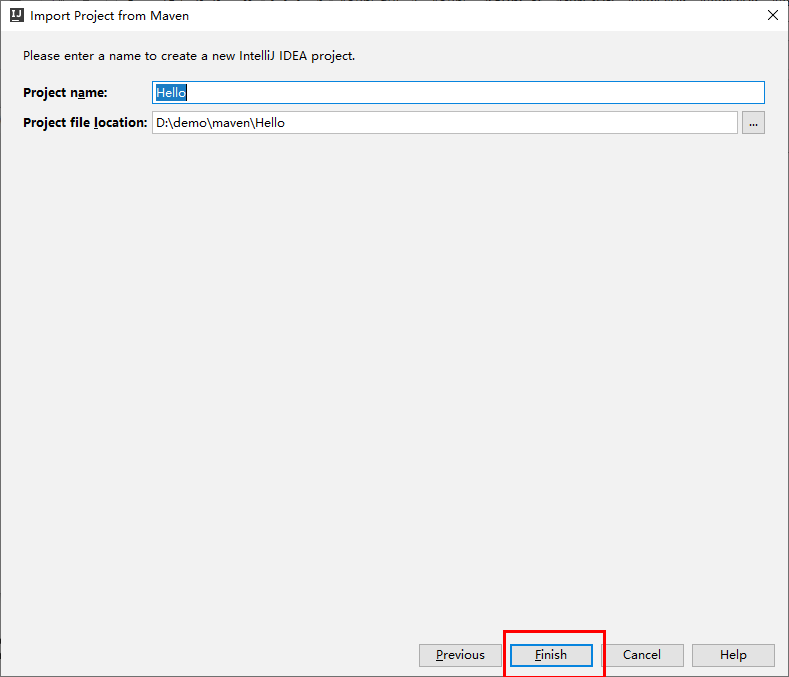
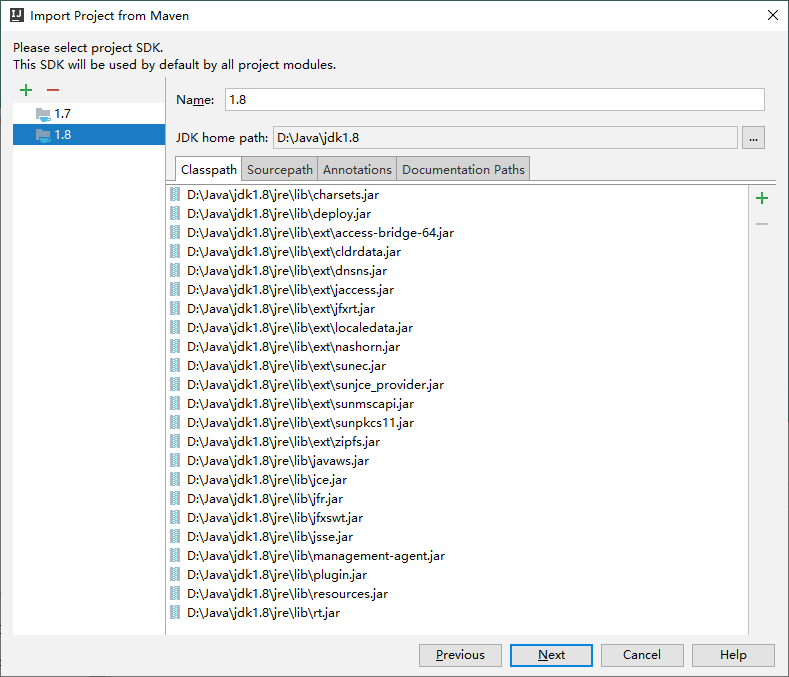
至此,在IDEA上使用Maven配置的Java web工程已经完成配置

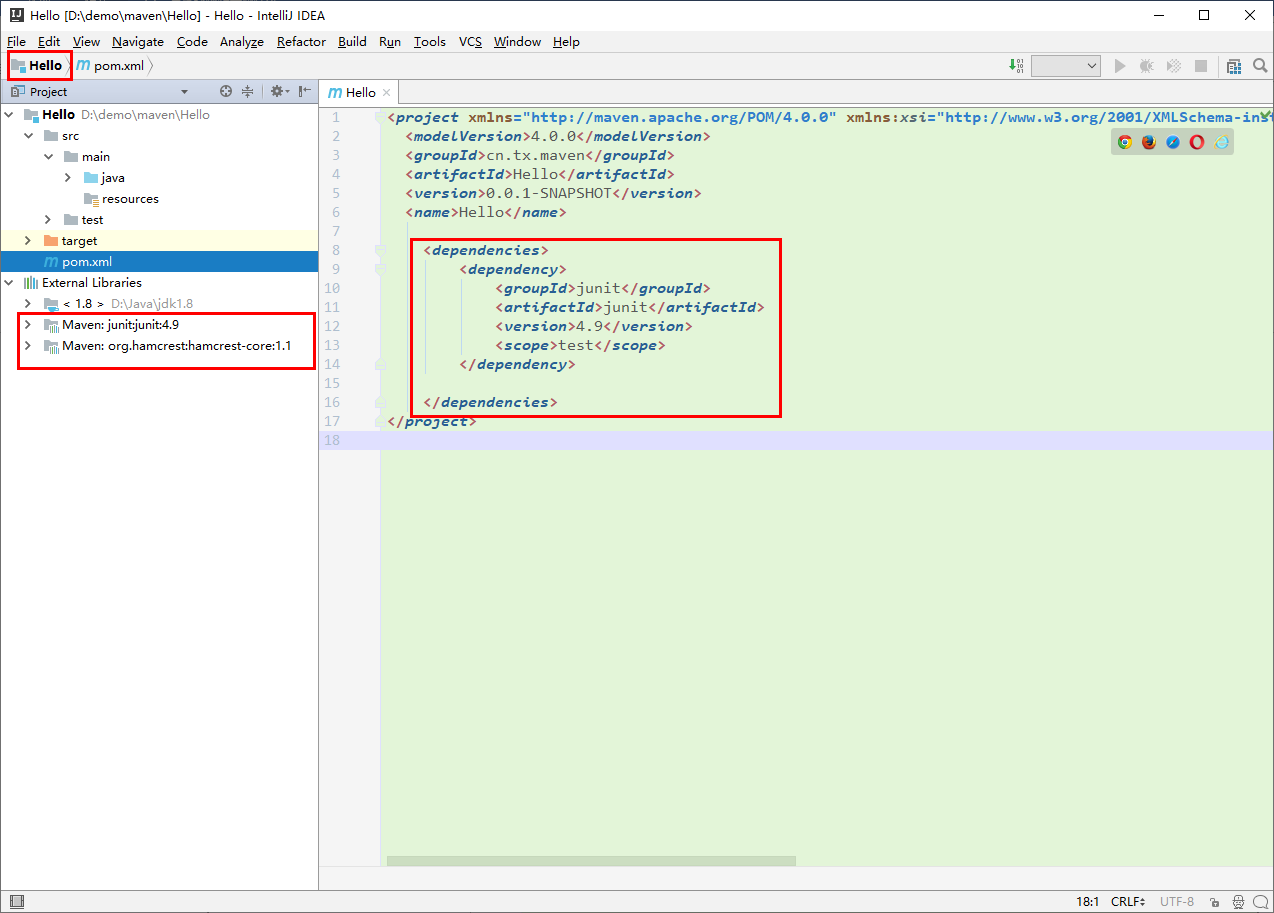
* + 1. IDEA导入既有Maven项目

以导入刚才的Maven项目的例子做示范:

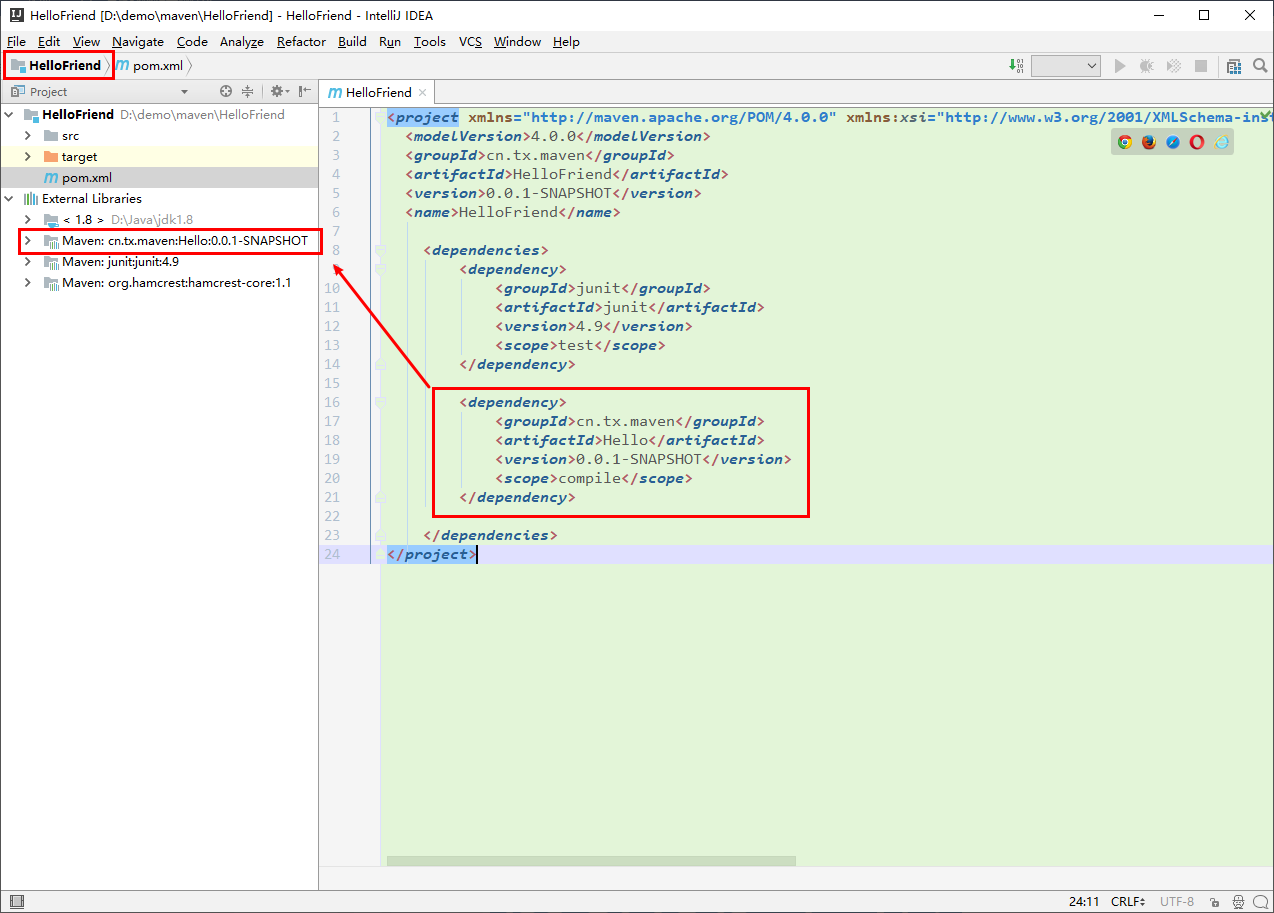






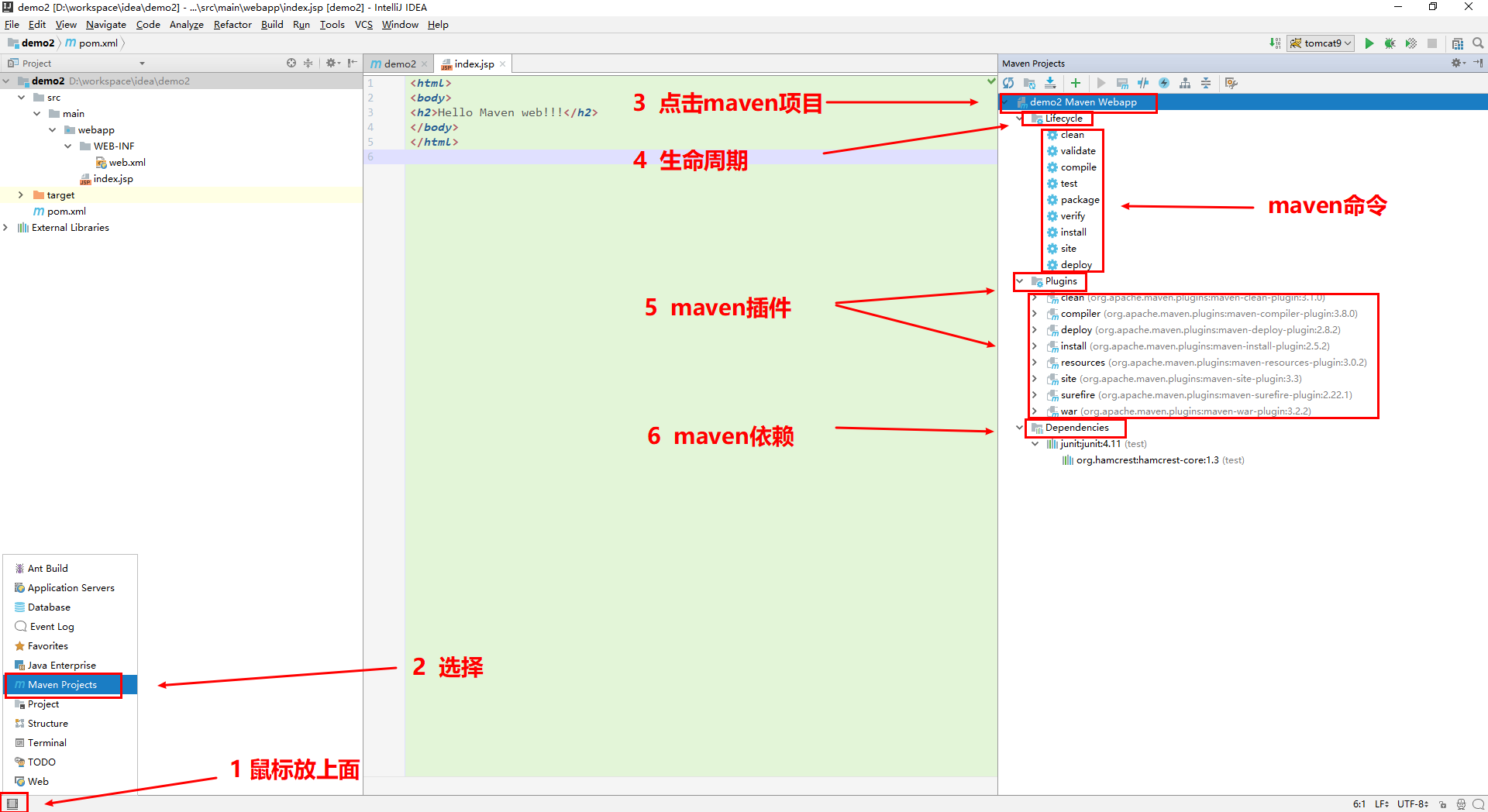


导入引入依赖Hello项目的HelloFriend项目,可以看到Hello项目已经被打成jar包引用到HelloFriend的Maven依赖中



* 1. Maven项目视图

Maven项目视图主要用于查看该maven项目的各项属性,同时也可以进行一些常见的maven的操作,比如打包,清理,测试等等;



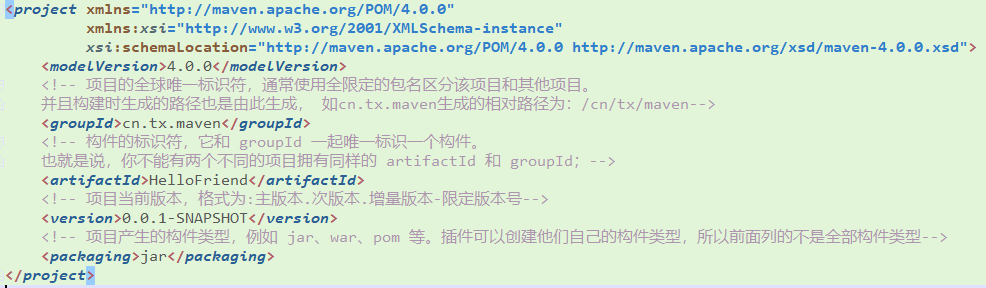
1. Maven进阶
   1. pom.xml文件

就像 Make 的 MakeFile、Ant 的 build.xml 一样，Maven 项目的核心是 pom.xml。POM( Project Object Model，项目对象模型 ) 定义了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。

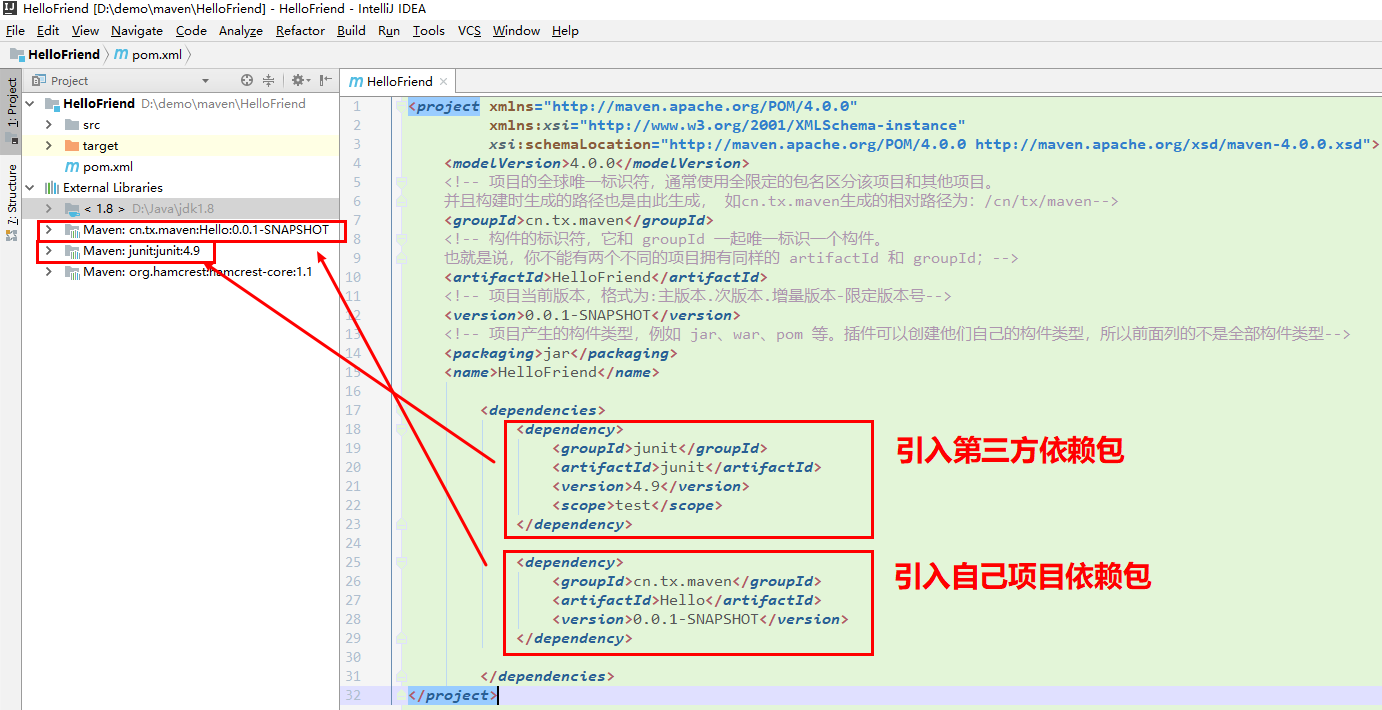
* 1. 坐标
     1. 坐标的概念

在 Maven 中坐标是构件的唯一标识，Maven 坐标的元素包括 **groupId**、**artifactId**、**version**、packaging、classifier。上述5个元素中，groupId、artifactId、version 是必须定义的，packaging 是可选的 ( 默认为 jar )。

* + 1. 坐标的意义
* Maven世界拥有大量构建，我们需要找一个用来唯一标识一个构建的统一规范
* 拥有了统一规范，就可以把查找工作交给机器
  + 1. 坐标的含义
* groupId：组织标识，一般为：公司网址的反写+项目名
* artifactId：项目名称，一般为：项目名-模块名
* version：版本号 形式为0.0.1-SNAPSHOT：
* 第一个 0 表示大版本号，第二个 0 表示分支版本号，第三个 0 表示小版本号
* SNAPSHOT -- 快照版本，ALPHA -- 内侧版本，BETA -- 公测版本，RELEASE -- 稳定版本，GA -- 正式发布
* packaging：打包的方式，如：pom, jar, maven-plugin, ejb, war, ...
* clissifier：用来帮助定义构件输出的一些附属构件。
  + 1. 自己项目的坐标



* + 1. 第三方项目坐标



* 1. 依赖
     1. 依赖的意义

当编写Java代码时，我们总是需要一些库，例如，做单元测试我们需要JUnit库。对于更大的项目，我们可能需要创建自己的库并在不同的部分使用它的项目。不同的项目需要不同版本的库。 保持项目最新的库JAR文件的正确版本不是一个容易的任务。

每个外部JAR可能还依赖于其他外部JAR文件等。以递归方式下载所有这些外部依赖JAR文件并确保下载正确的版本是一项巨大的任务。

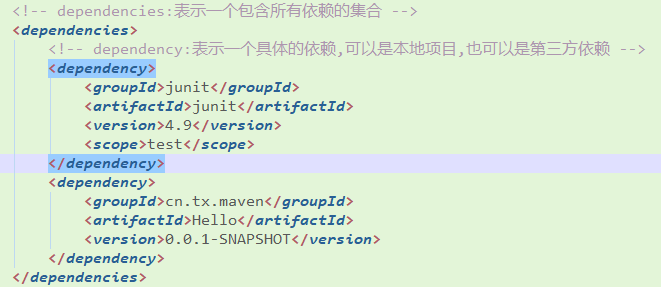
当项目越来越大，我们将需要越来越多的外部依赖。

Maven将下载它们并将它们放在您的本地Maven存储库中。

我们可以在POM文件中的dependencies元素内指定依赖关系。

* + 1. 依赖的使用

例如我们的项目需要进行单元测试,则需要使用到junit-4.9.jar包,使用maven引用该依赖的方式如下:

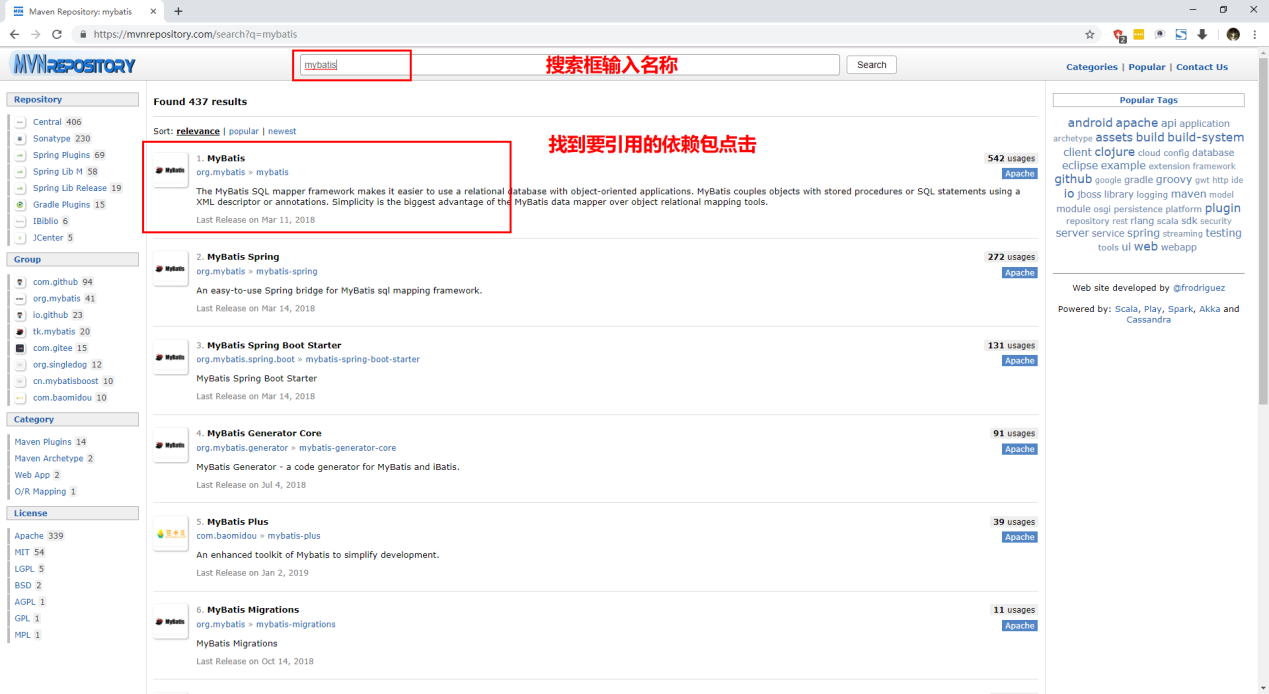


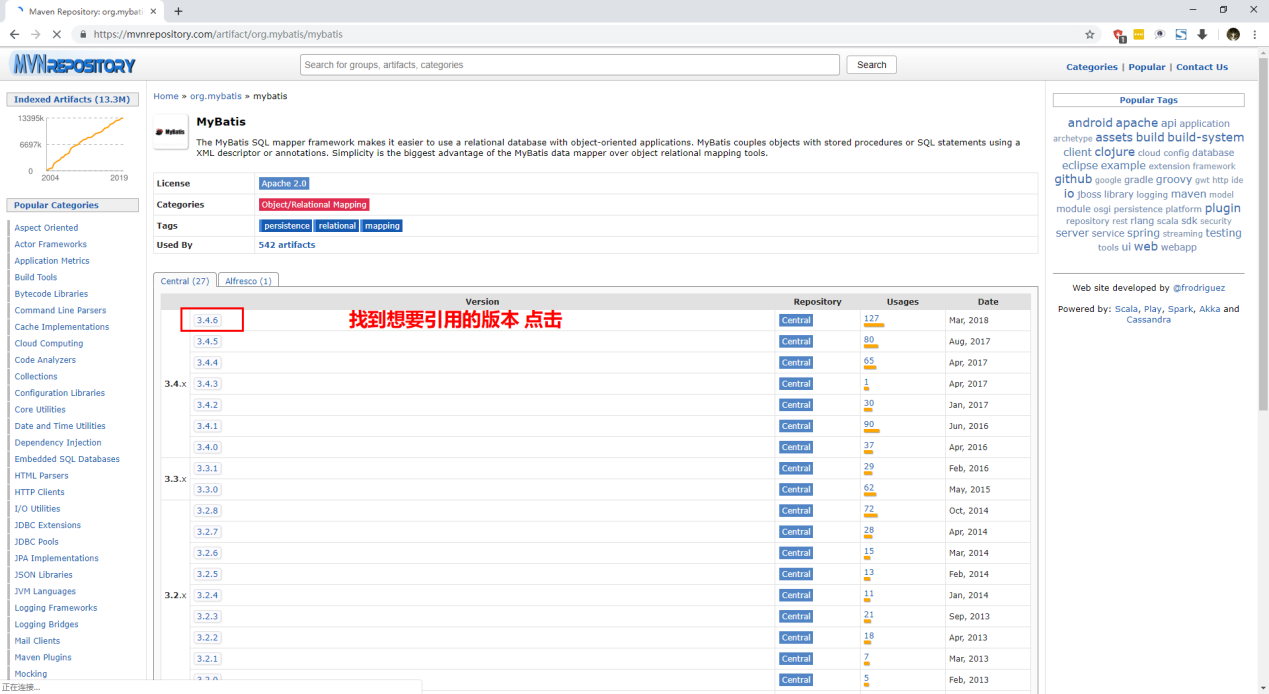
属性说明:

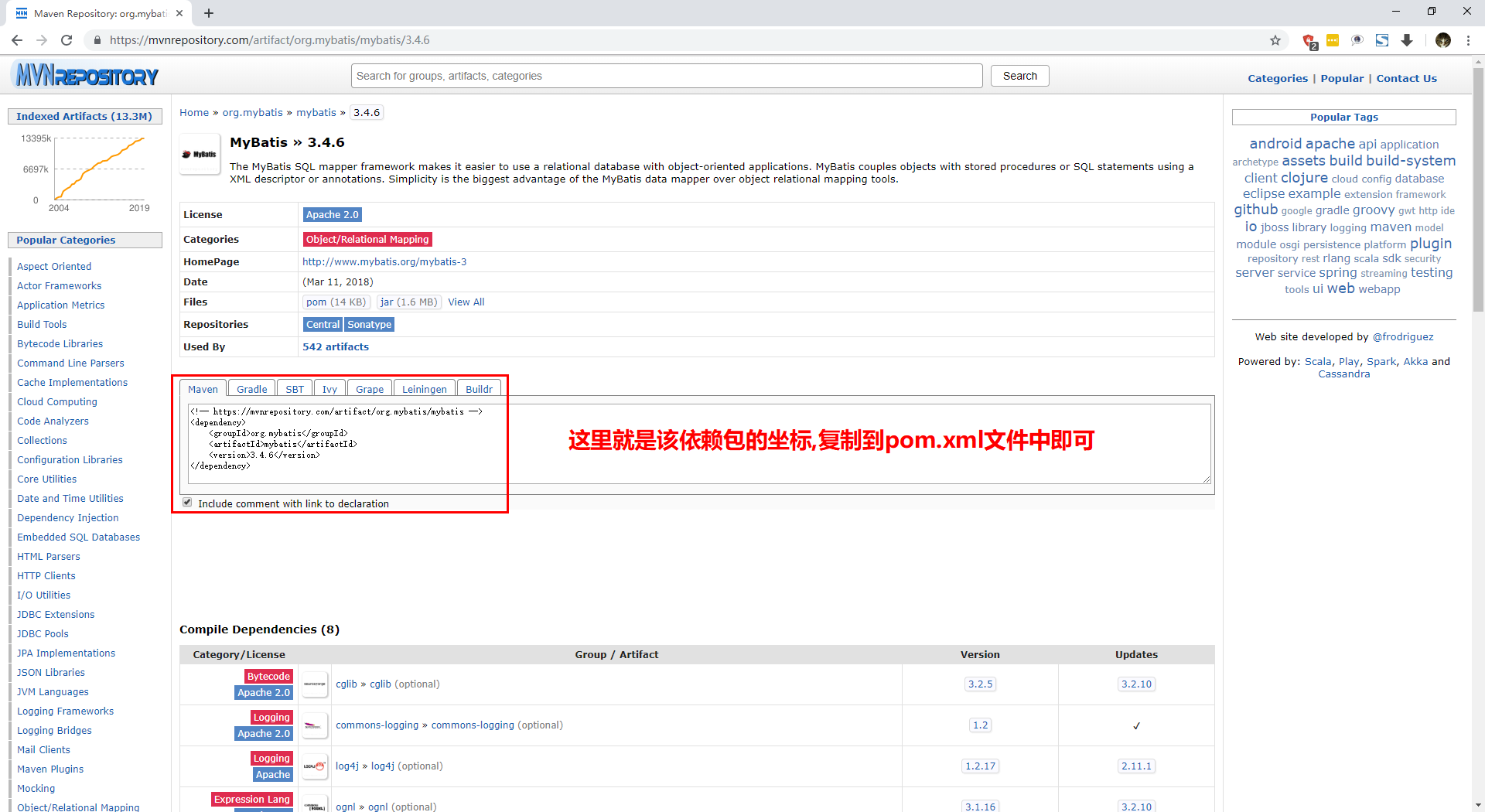
* + - 三维坐标:引用依赖包的三维坐标,用来定位该依赖包;
    - scope: 控制该依赖包在什么情况下会被加到 classpath 中;
    1. 第三方依赖的查找方法

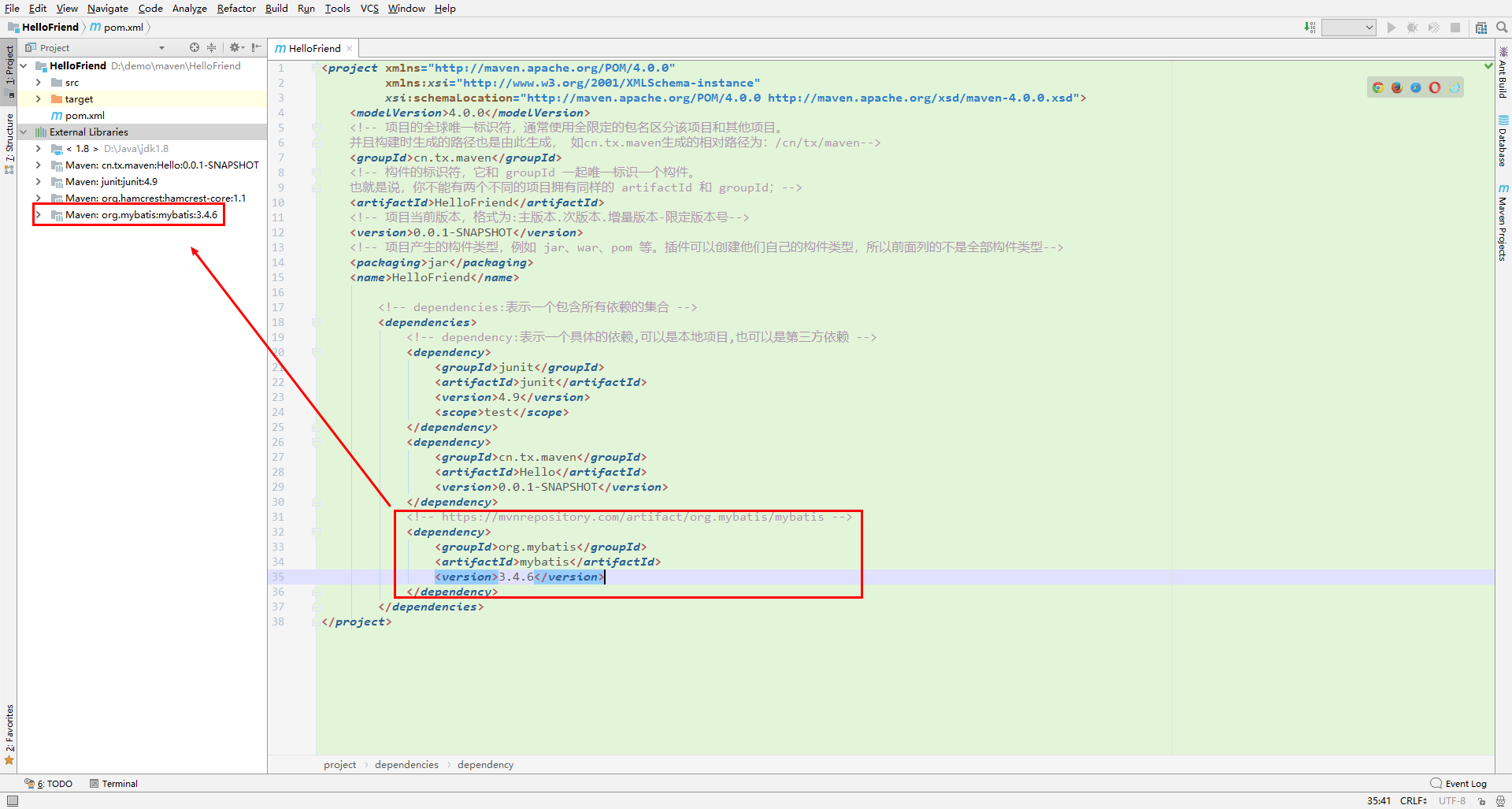
我们在不确定所需引用的第三方依赖包的坐标时,通过maven的中央仓库进行查找,网址: <https://mvnrepository.com/>;

以mybatis举例:







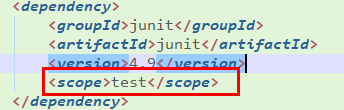


* 1. 依赖范围

Maven项目在开发工程中有三套classpath

* 主代码：main下面的都是主代码在编译的时候的依赖
* 测试代码：test下是测试代码编译的时候的依赖
* 运行时：main代码在运行的时候对包的依赖

依赖范围的使用,通过在引用第三方依赖时的<scope></scope>标签进行设置,例如:



共 6 种 scope，包括：**compile**、**provided**、**runtime**、**test**、system、import。例如上图的junit,只在测试中使用,则选择test即可,默认为**compile**

* Compile：编译依赖范围。默认使用此依赖范围，其下的maven依赖，对于编译，测试，运行classpath都有效。
* Test：测试依赖范围。只对测试classpath有效，编译主代码或运行项目时无法使用此依赖。典型例子如junit。
* Provided：已提供依赖范围。其对于编译与测试classpath有效，运行时无效。如在web开发时，只有在编译和测试时才用到servlet-api，将其设置为此范围，在运行时servlet-api由web容器提供，无须依赖。并且在打war包时，此范围的依赖不会打在WEB-INF/lib下。
* Runtime：运行时依赖范围。与provided相对，运行时classpath有效。典型例子如jdbc。



* 1. 依赖传递和可选依赖
     1. 依赖传递

应用场景:

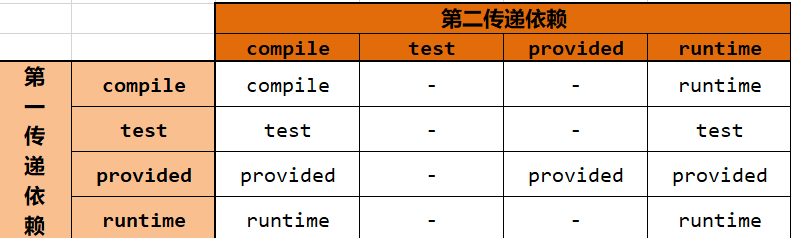
第一直接依赖: HelloFriend项目依赖Hello项目

第二直接依赖: MakeFriend项目依赖HelloFriend项目



* + 1. 依赖范围对传递依赖的影响

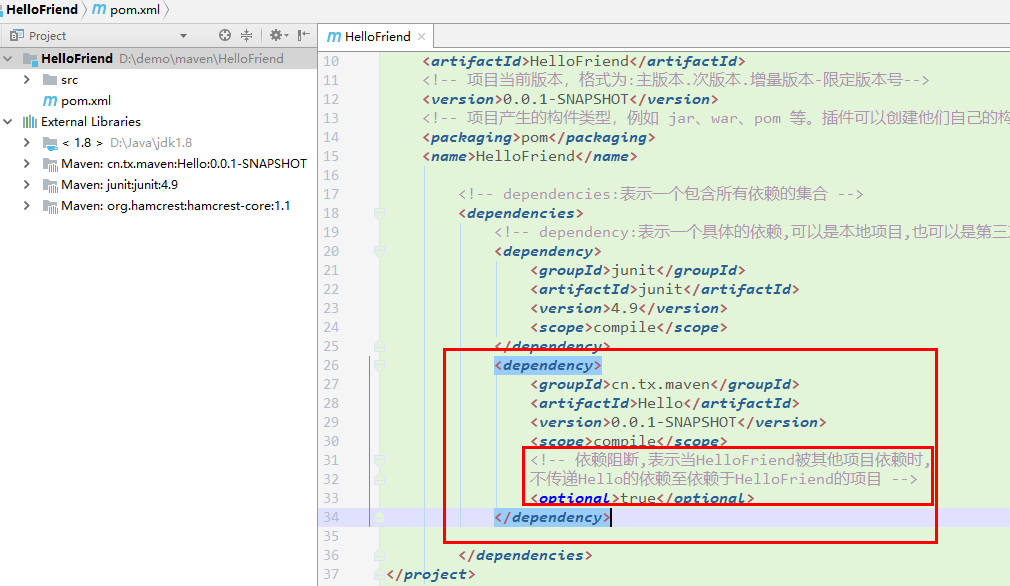
传递依赖是会受到依赖范围的影响的,具体来看结果如下:

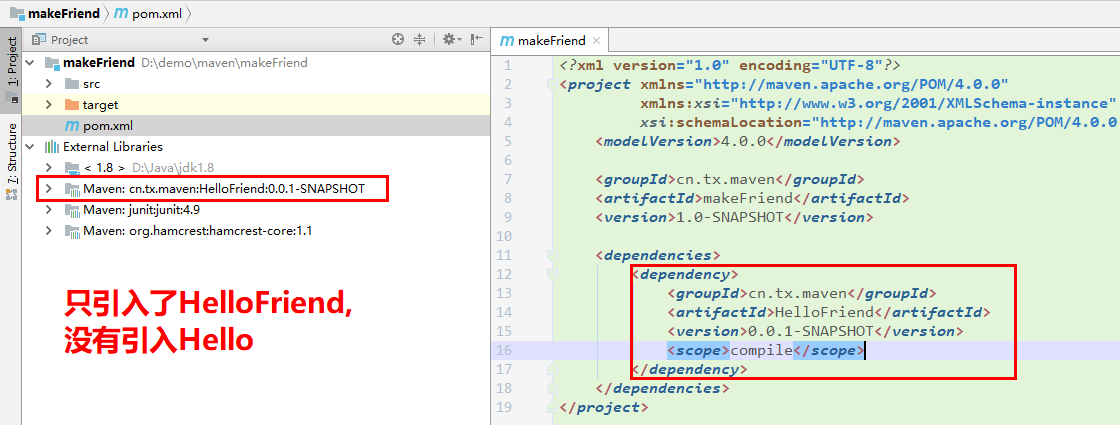


* + 1. 依赖阻断

我们使用

例如我们在HelloFriend项目里面的Hello依赖处添加该配置





则makeFriend项目里面就不会再引入Hello的依赖

* + 1. 可选依赖

如果我们需要在依赖中明确的排除掉某一依赖,则可以使用exclusion属性,排除掉引用的依赖,如图:



* 1. 仓库
     1. 仓库的概念

在 Maven 的术语中，仓库是一个位置（place）。

Maven 仓库是项目中依赖的第三方库，这个库所在的位置叫做仓库。

在 Maven 中，任何一个依赖、插件或者项目构建的输出，都可以称之为构件。

Maven 仓库能帮助我们管理构件（主要是JAR），它就是放置所有JAR文件（WAR，ZIP，POM等等）的地方。

仓库的类型有:

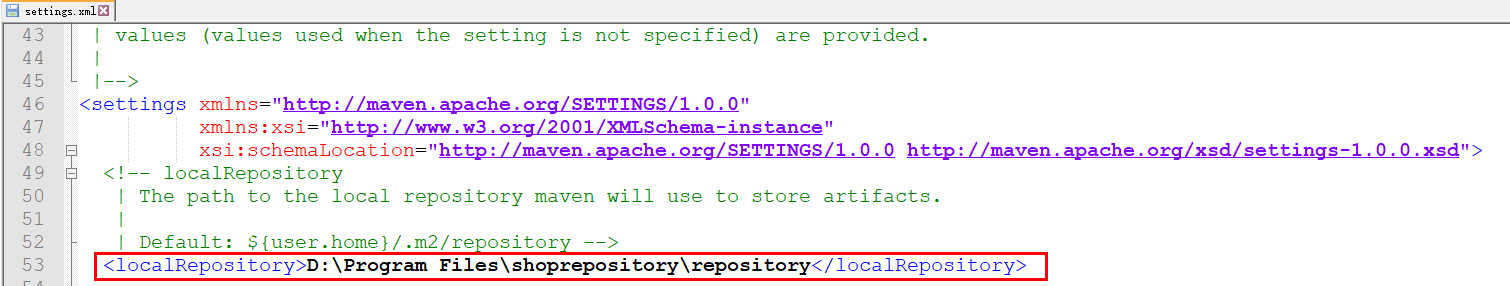
* 本地（local）
* 中央（central）
* 远程（remote）
  + 1. 本地仓库

Maven 的本地仓库，在安装 Maven 后并不会创建，它是在第一次执行 maven 命令的时候才被创建。

运行 Maven 的时候，Maven 所需要的任何构件都是直接从本地仓库获取的。如果本地仓库没有，它会首先尝试从远程仓库下载构件至本地仓库，然后再使用本地仓库的构件。

默认情况下，不管Linux还是 Windows，每个用户在自己的用户目录下都有一个路径名为 .m2/respository/ 的仓库目录

Maven 本地仓库默认被创建在 %USER\_HOME% 目录下。要修改默认位置，在 %Maven\_HOME%\conf 目录中的 Maven 的 settings.xml 文件中定义另一个路径。



* + 1. 中央仓库

Maven 中央仓库是由 Maven 社区提供的仓库，其中包含了大量常用的库。

中央仓库包含了绝大多数流行的开源Java构件，以及源码、作者信息、SCM、信息、许可证信息等。一般来说，简单的Java项目依赖的构件都可以在这里下载到。

中央仓库的关键概念：

* 这个仓库由 Maven 社区管理。
* 不需要配置。
* 需要通过网络才能访问。
  + 1. 依赖搜索顺序



* 1. 生命周期
     1. 生命周期的概念

Maven的生命周期是对所有的构建过程进行抽象和统一。Maven的生命周期是抽象的，这意味着生命周期本身不做任何实际的工作，生命周期只是定义了一系列的阶段，并确定这些阶段的执行顺序。而在执行这些阶段时，实际的工作还是由插件来完成的。这种思想与设计模式中的模板方法非常相似。

Maven有三套相互独立的生命周期:

* Clean
  + - clean生命周期的目的是清理项目
* Default
  + - default生命周期的目的是构建项目
* site
  + - site生命周期的目的是建立项目站点。
    1. 完整生命周期

|  |  |  |  |
| --- | --- | --- | --- |
| **生命周期** | **clean** | **default** | **site** |
| 阶段(phase)，执行顺序由上至下 | pre-clean | validate | pre-site |
| clean | initialize | site |
| post-clean | generate-sources | post-site |
|  | process-sources | site-deploy |
| generate-resources |  |
| process-resources |
| **compile** |
| process-classes |
| generate-test-sources |
| process-test-sources |
| generate-test-resources |
| process-test-resources |
| test-compile |
| process-test-classes |
| **test** |
| prepare-package |
| **package** |
| pre-integration-test |
| integration-test |
| post-integration-test |
| verify |
| **install** |
| deploy |

用户在mvn命令后可以指定三个生命周期中的任何阶段，则Maven会按以下逻辑执行：首先会得到该阶段所属生命周期，从该生命周期中的第一个阶段开始按顺序执行，直至该阶段本身。例如执行mvn clean命令会依次执行clean生命周期中的pre-clean阶段及clean阶段。mvn命令后可以指定多个阶段，Maven会按照输入的顺序依次执行，每次执行都会按照之前描述的逻辑执行。

之前提到实际的工作还是由插件来完成的，这意味着插件需要和阶段绑定起来。Maven已经事先将很多核心插件绑定到对应的阶段，这样用户几乎不用配置就有构建Maven项目。Maven的内置绑定如下：

|  |  |  |
| --- | --- | --- |
| **生命周期** | **阶段(phase)** | **插件目标** |
| clean | **clean** | maven-clean-plugin:clean |
| default | process-resources | maven-resources-plugin:resources |
| **compile** | maven-compiler-plugin:compile |
| generate-test-resources | maven-resources-plugin:testResouces |
| test-compile | maven-compiler-plugin:testCompile |
| **test** | maven-surefire-plugin:test |
| **package** | 打包类型是jar时：maven-jar-plugin:jar； |
| 打包类型是war时：maven-war-plugin:war |
| **install** | maven-install-plugin:install |
| deploy | maven-deploy-plugin:deploy |
| site | site | maven-site-plugin:site |
| site-deploy | maven-site-plugin:deploy |

* 1. Maven继承和聚合
     1. 继承的意义

继承就是避免重复，maven的继承也是这样，它还有一个好处就是让项目更加安全。比如我们在项目开发的过程中，可能多个模块独立开发，但是多个模块可能依赖相同的公共模块，比如说每个模块都需要javaseo-utils，在编译的时候，maven-compiler-plugin插件也要被引入，maven仓库地址以及发布目录都是相同的配置。我们可以使用Maven的继承功能，把公共的配置信息写到父模块中，子模块只要继承了该父模块，也会继承父模块的配置信息。

* + 1. 可继承的POM元素

**groupId** ：项目组 ID ，项目坐标的核心元素；

**version** ：项目版本，项目坐标的核心元素；

description ：项目的描述信息；

organization ：项目的组织信息；

inceptionYear ：项目的创始年份；

url ：项目的 url 地址

develoers ：项目的开发者信息；

contributors ：项目的贡献者信息；

distributionManagerment ：项目的部署信息；

issueManagement ：缺陷跟踪系统信息；

ciManagement ：项目的持续继承信息；

scm ：项目的版本控制信息；

mailingListserv ：项目的邮件列表信息；

**properties** ：自定义的 Maven 属性；

**dependencies** ：项目的依赖配置；

**dependencyManagement** ：醒目的依赖管理配置；

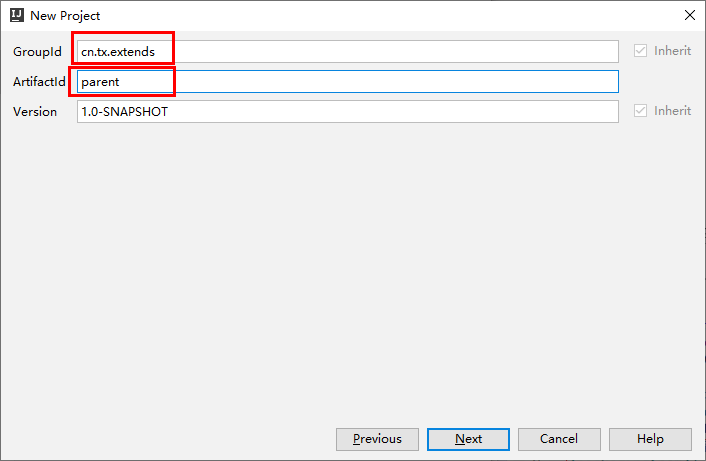
repositories ：项目的仓库配置；

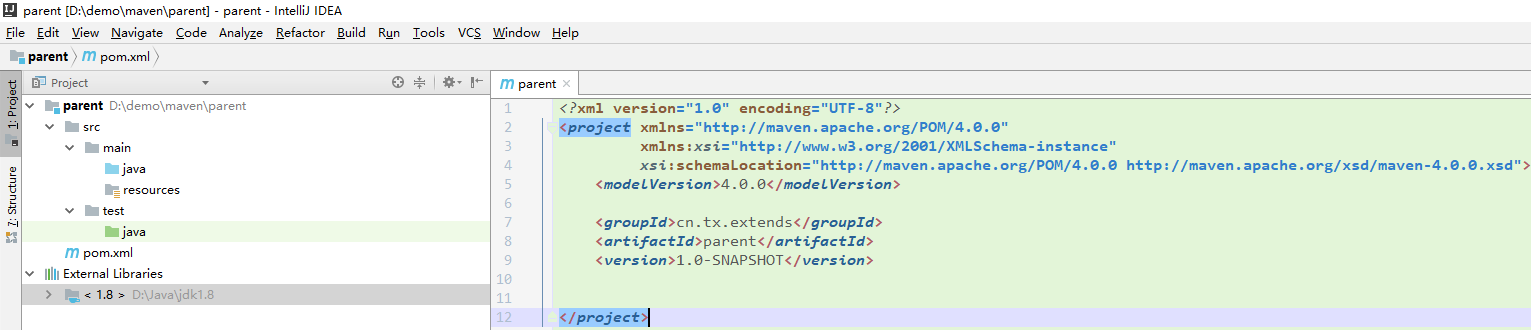
build ：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等；

reporting ：包括项目的报告输出目录配置、报告插件配置等。

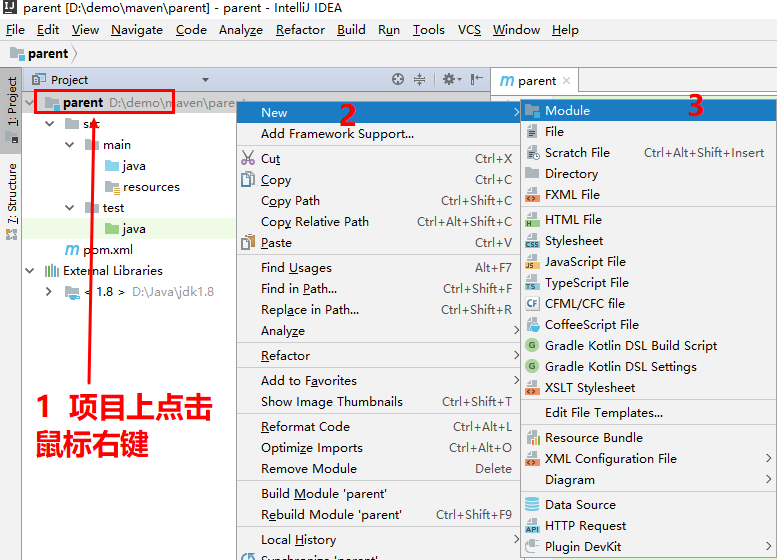
* + 1. IDEA实现Maven的继承

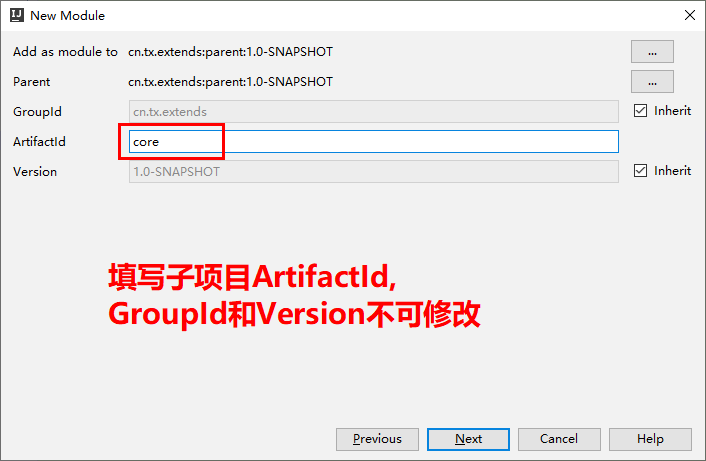
创建父类项目

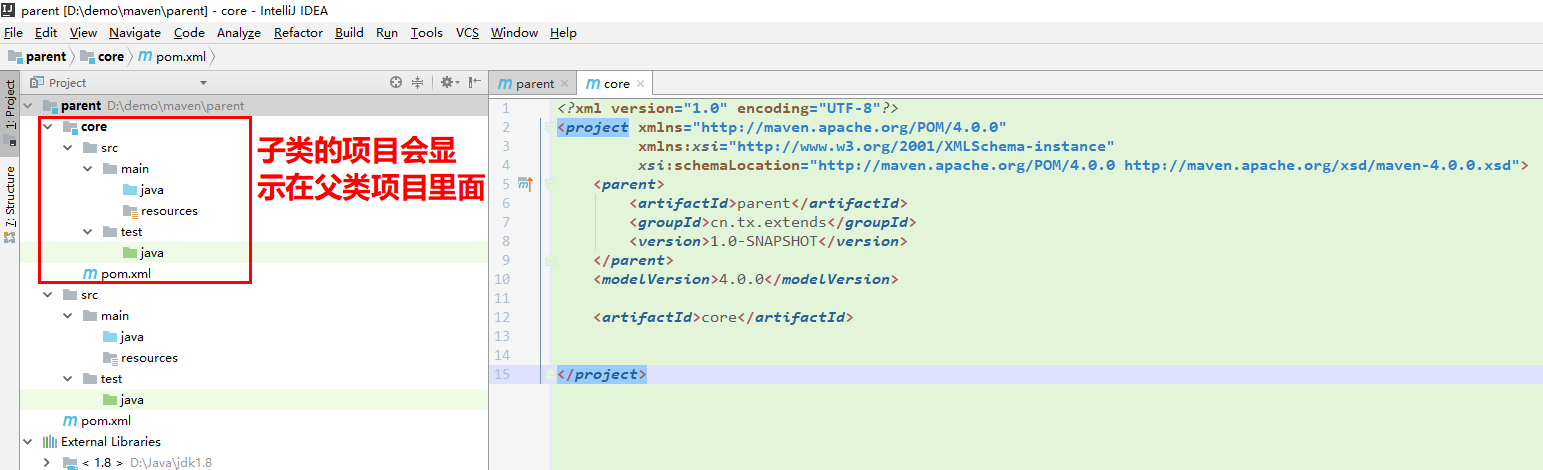




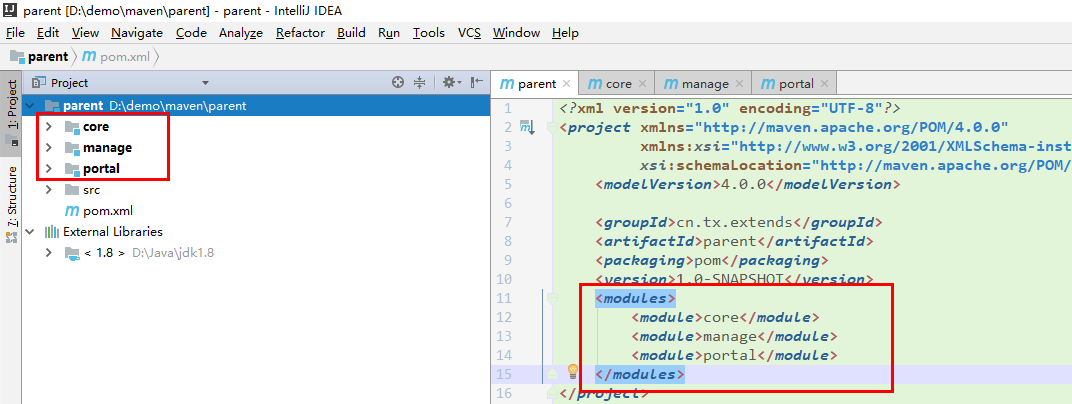
创建子类项目core





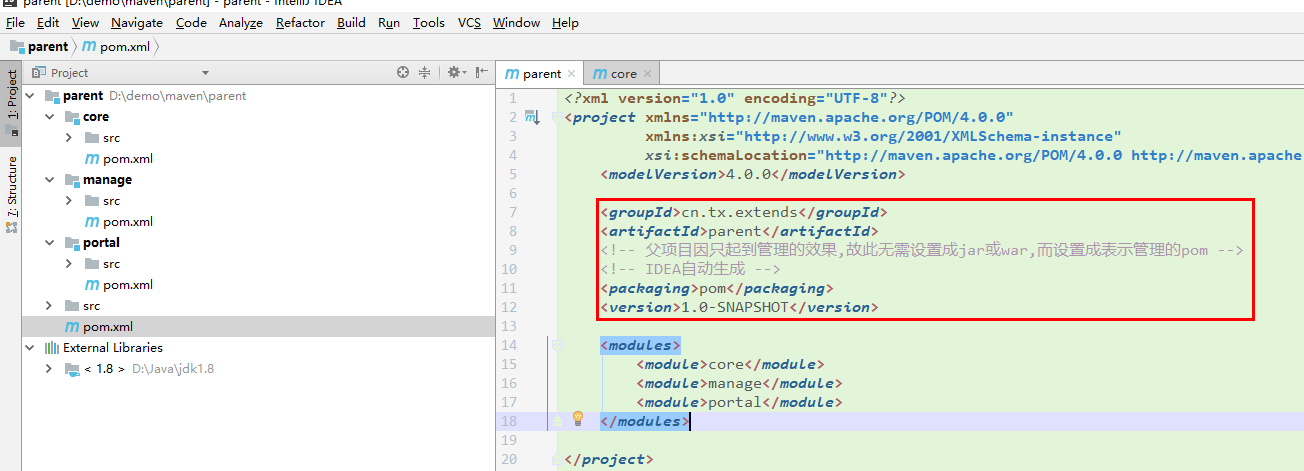


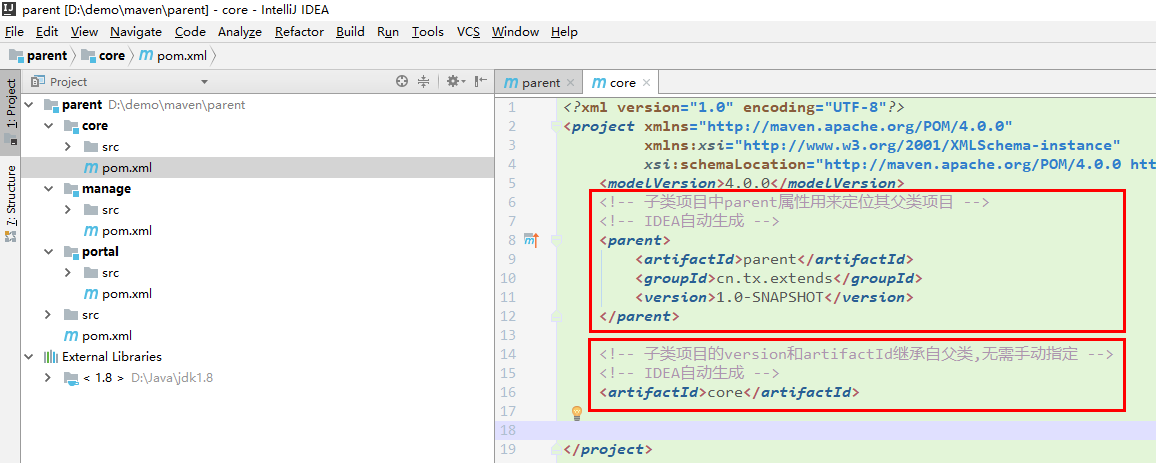
以同样的方式创建manage和portal项目:



至此,一个由parent统一管理core,manage和portal的mavan项目就创建好了;

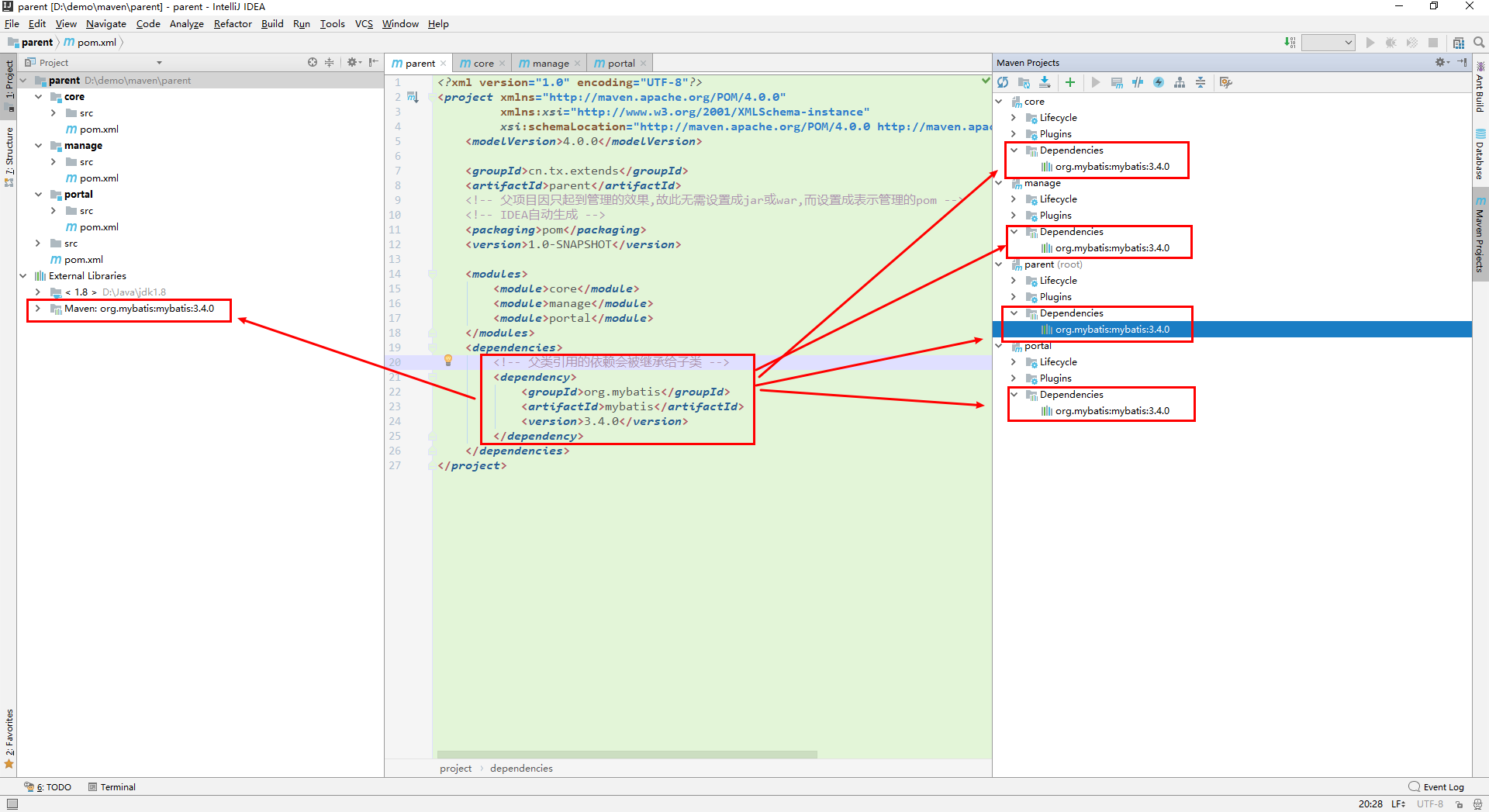
观察父子项目的pom文件配置



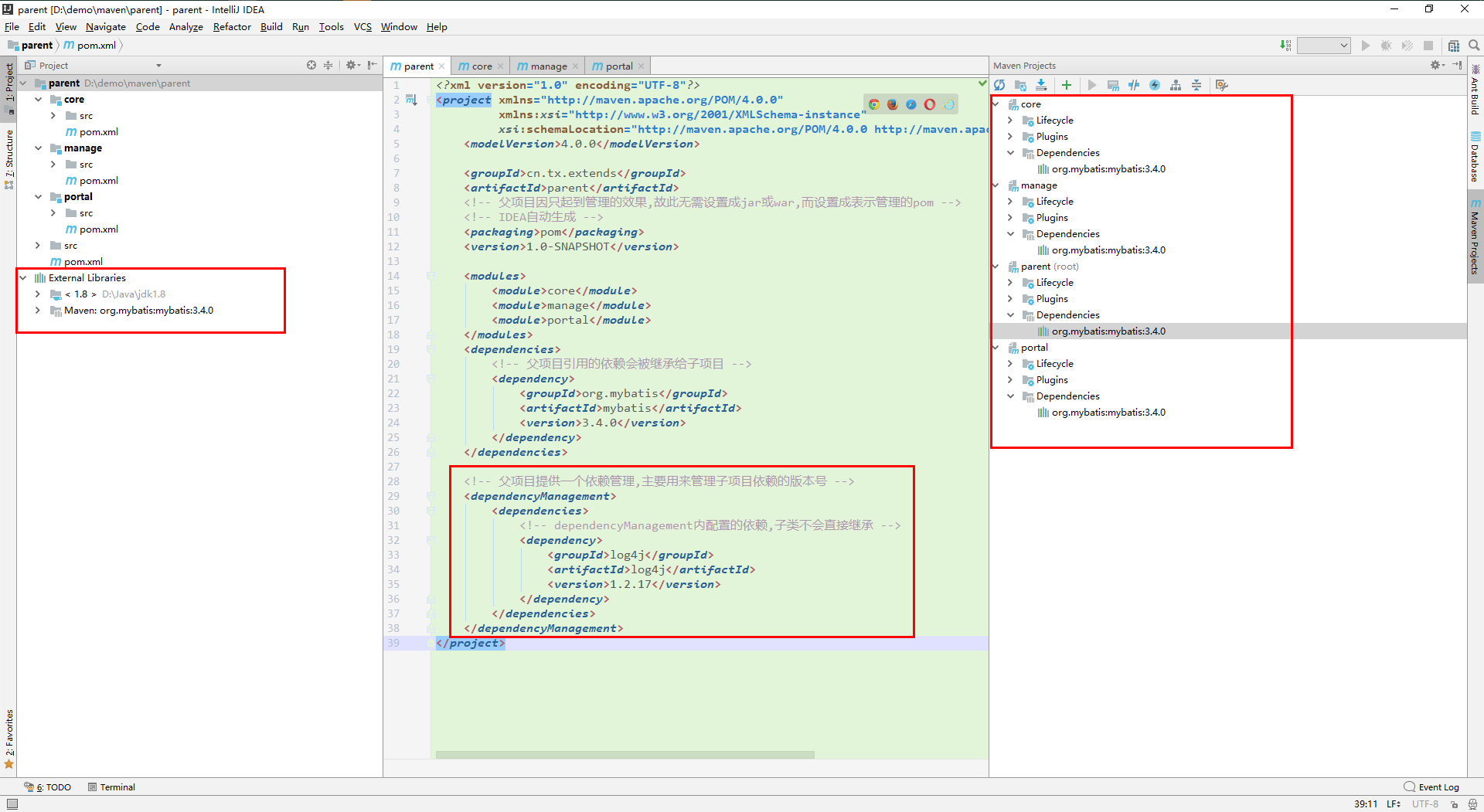


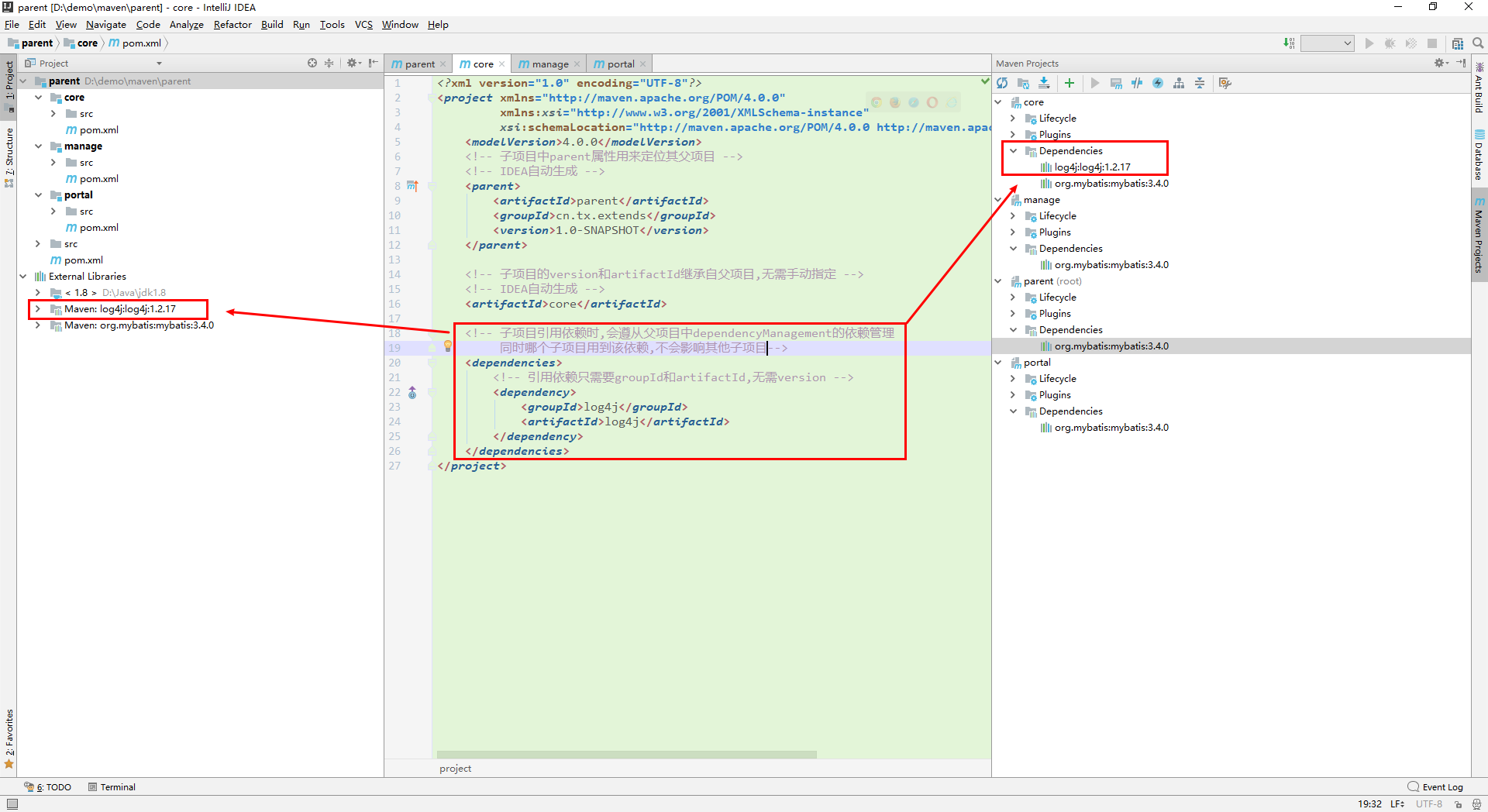
* + 1. 继承的依赖管理

父类直接引用依赖,如果在父类的dependencies内直接引用依赖,则子类都会继承该依赖,以mybatis为例:



父类通过dependencyManagement管理依赖,子类不会默认继承该依赖,但是当子类使用该依赖时无需考虑版本信息,直接继承父类dependencyManagement中设置的版本号(Version),以log4j为例:

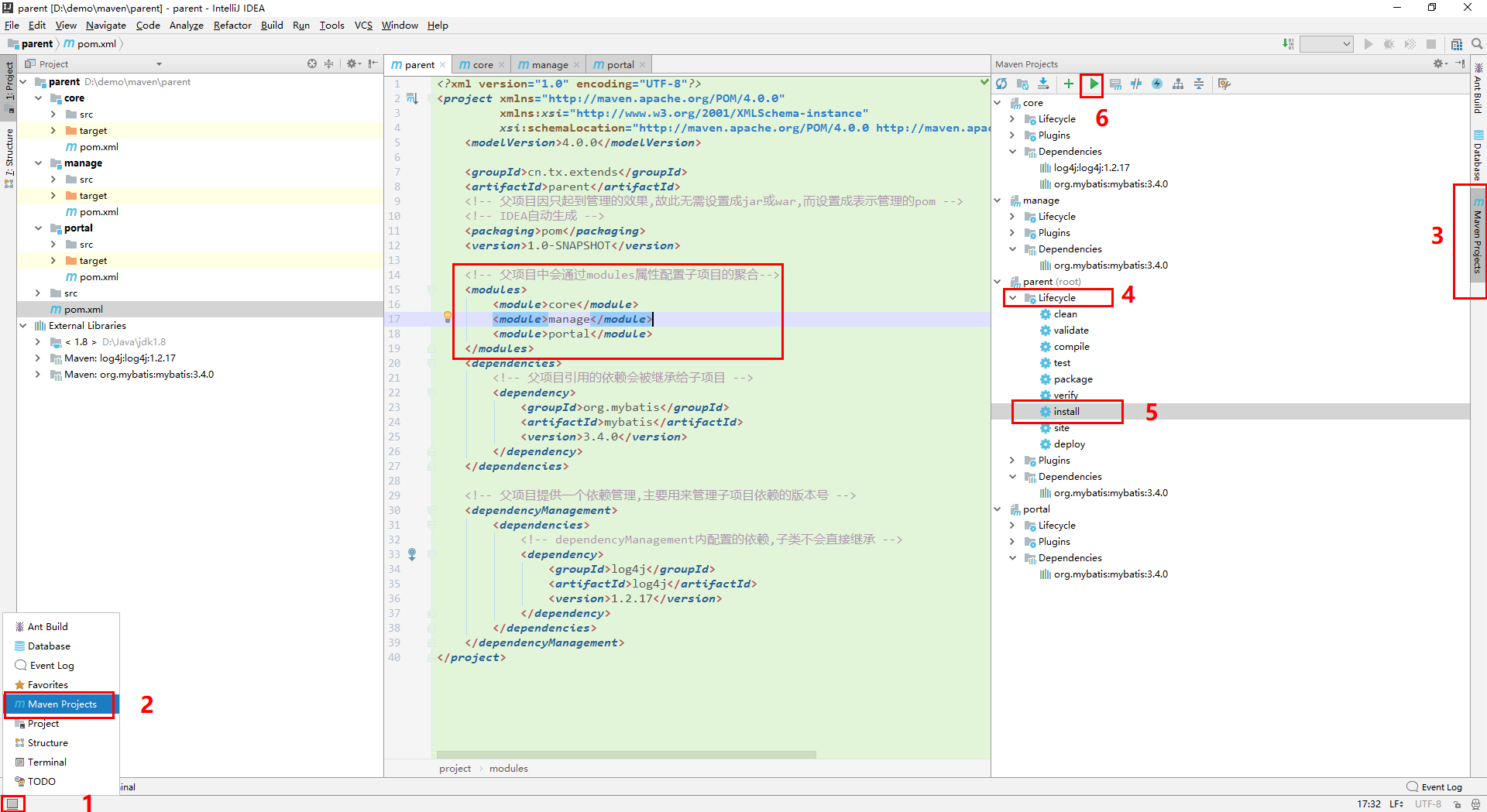


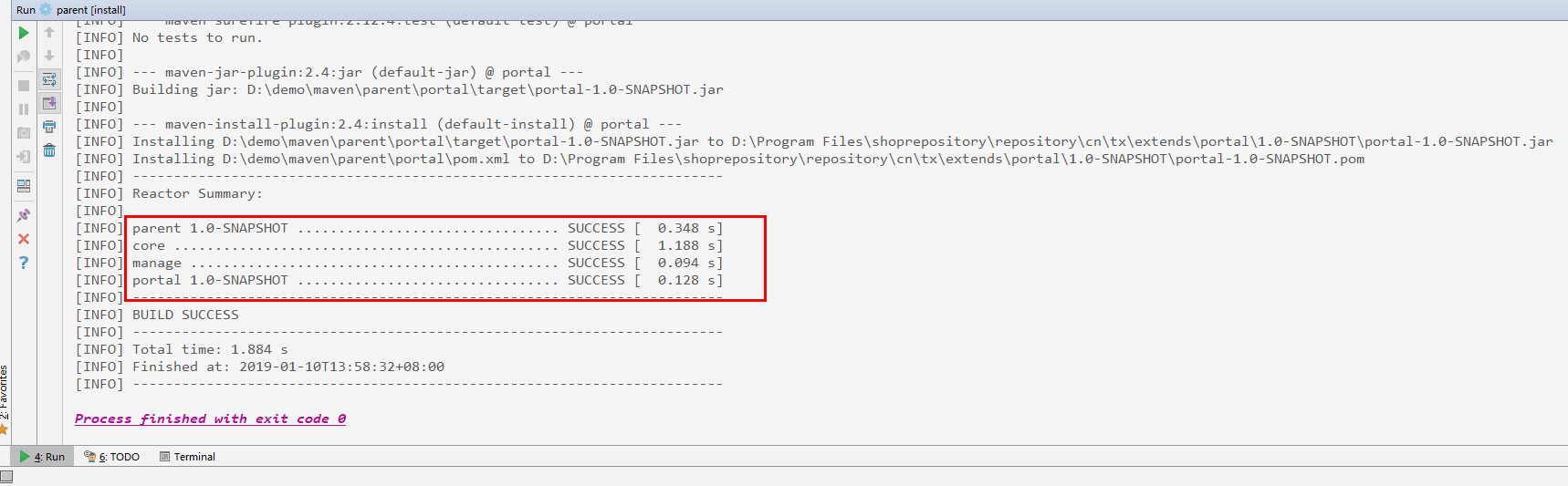


* + 1. 聚合管理

我们在平时的开发中，项目往往会被划分为好几个模块，比如common公共模块、system系统模块、log日志模块、reports统计模块、monitor监控模块等等。这时我们肯定会出现这么一个需要，我们需要一次构件多个模块，而不用每个模块都去mvn;

以上面得父子项目为例,当我们对parent进行mvn install时,会对core,manage和portal项目均进行install操作:





* + 1. properties属性的使用

通过 properties元素用户可以定义一个或多个 maven 属性，然后在 maven 的其他地方使用 ${属性名称} 的方式引用该属性，这种做法的意义在于消除重复和统一管理。比如，需要在多个地方重复声明同样的 SpringFramework 版本，现在只需要在一个地方声明就可以。

