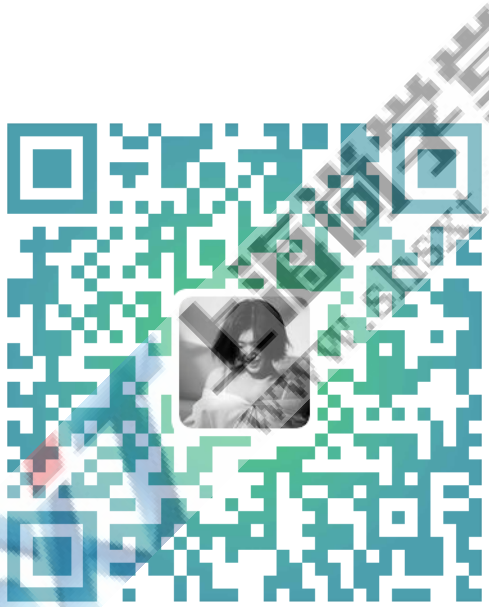
上海尚学堂高级架构课程

正所谓“授人以鱼不如授人以渔”，你们想要的 Java 学习资料来啦！

不管你是学生，还是已经步入职场的同行，希望你们都要珍惜眼前的学习机会，奋斗没有终点，知识永不过时。

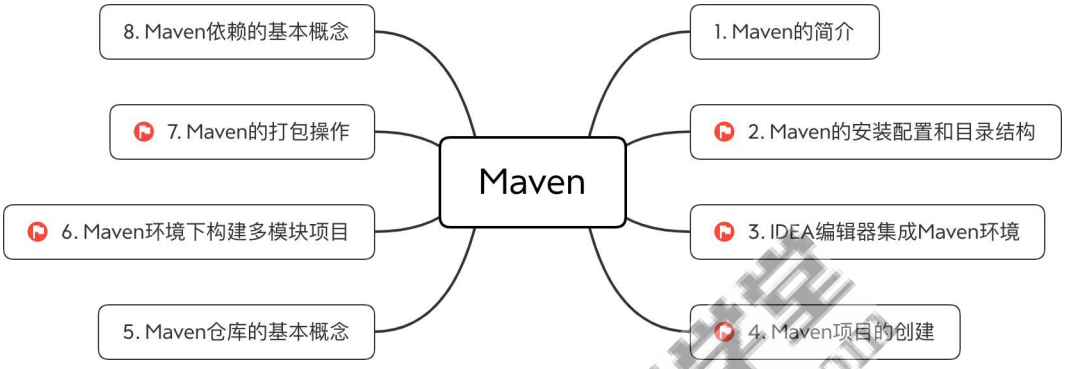
扫描下方二维码即可领取



乐字节晓啡

Maven

1. 主要内容



2. Maven的简介

2.1. 简介

Maven【['mevən]】这个词可以翻译为"专家","内行"。 作为Apache组织中的一个颇为成功的开源项目，Maven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。

无论是小型的开源类库项目，还是大型的企业级应用； 无论是传统的瀑布式开发，还是流行的敏捷开发，Maven都能大显身手。

2.2. 项目构建

不管你是否意识到，构建（build）是每一位程序员每天都在做的工作。早上来到公司，我们做的第一件事就是从源码库签出最新的代码，然后进行单元测试，如果测试失败，会找相关的同事一起调试，修复错误代码。 接着回到自己的工作上来，编写自己的单元测试及产品代码。

仔细总结一下，我们会发现，除了编写源代码，我们每天有相当一部分时间花在了编译，运行单元测试，生成文档，打包和部署等繁琐且不起眼的工作上，这就是构建。　如果我们现在还手工这样做，那成本也太高了，于是有人用软件的方法让这一系列工作完全自动化，使得软件的构建可以像全自动流水线一样，只需要一条简单的命令，所有繁琐的步骤都能够自动完成，很快就能得到最终结果。

2.3. 项目构建工具

Ant构建

最早的构建工具，基于IDE,　大概是2000年有的，当时是最流行java构建工具，不过它的XML脚本编写格式让XML文件特别大。对工程构建过程中的过程控制特别好

Maven【JAVA】

项目对象模型，通过其描述信息来管理项目的构建，报告和文档的软件项目管理工具。它填补了Ant缺点，Maven第一次支持了从网络上下载的功能，仍然采用xml作为配置文件格式。Maven专注的是依赖管理，使用Java编写。

Gradle

属于结合以上两个的优点，它继承了Ant的灵活和Maven的生命周期管理，它最后被google作为了Android御用管理工具。它最大的区别是不用XML作为配置文件格式，采用了DSL格式，使得脚本更加简洁。

目前市面上Ant比较老, 所以一般是一些比较传统的软件企业公司使用, Maven使用Java编写, 是当下大多数互联网公司会使用的一个构建工具, 中文文档也比较齐全, gradle是用groovy编写, 目前比较新型的构建工具一些初创互联网公司会使用, 以后会有很大的使用空间.

2.4. Maven的四大特性

2.4.1. 依赖管理系统

Maven为Java世界引入了一个新的依赖管理系统jar包管理 jar 升级时修改配置文件即可。在Java世界中，可以用groupId、artifactId、version组成的Coordination（坐标）唯一标识一个依赖。

任何基于Maven构建的项目自身也必须定义这三项属性，生成的包可以是Jar包，也可以是war包或者jar包。一个典型的依赖引用如下所示：

```
<dependency>
  <groupId>javax.servlet</groupId>   com.baidu
  <artifactId>javax.servlet-api</artifactId>  ueditor echarts
  <version>3.1.0</version>
</dependency>
```

坐标属性的理解

Maven坐标为各种组件引入了秩序，任何一个组件都必须明确定义自己的坐标。

groupId

定义当前Maven项目隶属的实际项目-公司名称。（jar包所在仓库路径） 由于Maven中模块的概念，因此一个实际项目往往会被划分为很多模块。 比如spring是一个实际项目，其对应的Maven模块会有很多，如spring-core,spring-webmvc等。

artifactId

该元素定义实际项目中的一个Maven模块-项目名， 推荐的做法是使用实际项目名称作为artifactId的前缀。比如： spring-bean, spring-webmvc等。

version

该元素定义Maven项目当前所处的版本。

2.4.2. 多模块构建

项目复查时 dao service controller 层分离将一个项目分解为多个模块已经是很通用的一种方式。

在Maven中需要定义一个parent POM作为一组module的聚合POM。在该POM中可以使用 标签来定义一组子模块。parent POM不会有什么实际构建产出。而parent POM中的build配置以及依赖配置都会自动继承给子module。

2.4.3. 一致的项目结构

Ant时代大家创建Java项目目录时比较随意，然后通过Ant配置指定哪些属于source，那些属于testSource等。而Maven在设计之初的理念就是Conversion over configuration（约定大于配置）。其制定了一套项目目录结构作为标准的Java项目结构,解决不同ide 带来的文件目录不一致问题。

2.4.4. 一致的构建模型和插件机制

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.25</version>
  <configuration>
     <scanIntervalSeconds>10</scanIntervalSeconds>
     <contextPath>/test</contextPath>
  </configuration>
</plugin>
```

3. Maven的安装配置和目录结构

3.1. Maven的安装配置

3.1.1. 检查JDK的版本

JDK版本1.7及以上版本

3.1.2. 下载Maven

下载地址： http://maven.apache.org/download.html

3.1.3. 配置Maven环境变量

解压后把Maven的根目录配置到系统环境变量中MAVEN_HOME，将bin目录配置到path变量中。

注：maven解压后存放的目录不要包含中文和空格

3.1.4. 检查Maven是否安装成功

打开dos窗口，执行 mvn -v

3.2. 认识Maven目录结构

Maven项目目录结构

目录	目的
${basedir}	存放 pom.xml和所有的子目录
${basedir}/src/main/java	项目的 java源代码
${basedir}/src/main/resources	项目的资源，比如说 property文件
${basedir}/src/test/java	项目的测试类，比如说 JUnit代码
${basedir}/src/test/resources	测试使用的资源

任务:手动创建一个Maven项目，并编译运行成功！

3.2.1. 创建一个文件夹作为项目的根目录

在根目录中创建一个pom.xml文件，内容如下

```
<?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.xxxx</groupId>
  <artifactId>maven01</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>maven01</name>
  <url>http://maven.apache.org</url>

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

   <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>3.8.1</version>
       <scope>test</scope>
     </dependency>
   </dependencies>

</project>
```

Ps：标签定义解释

根目录下的第一个子元素 ModelVersion指定当前Pom模型的版本，对于Maven3来说，它只能是4.0.0 。指定了当前Maven模型的版本号，对于Maven2和Maven3来说，它只能是4.0.0

groupId定义了项目属于哪个组， 这个组往往和项目所在的组织和公司存在关联。
比如： com.xxxx

artifactId 定义了当前Maven项目在组中唯一的ID。

Version X.X.X-里程碑
比如：1.0.0-SNAPSHOT
第一个X 大版本 有重大变革
第二个X 小版本 修复bug，增加功能
第三个X 更新

里程碑版本：
SNAPSHOT （快照，开发版）
alpha（内部测试）beta（公开测试）Release | RC（发布版）GA（正常版本）

使用name标签声明一个对于用户更为友好的项目名称，虽然不是必须的，但还是推荐为每个Pom声明name，以方便信息交流。

3.2.2. 编写主函数

```
package com.xxxx.demo;
public class Hello{
    public static void main(String[] args)
        { System.out.println("hello
        maven");
    }
```

3.2.3. cmd 下编译并运行

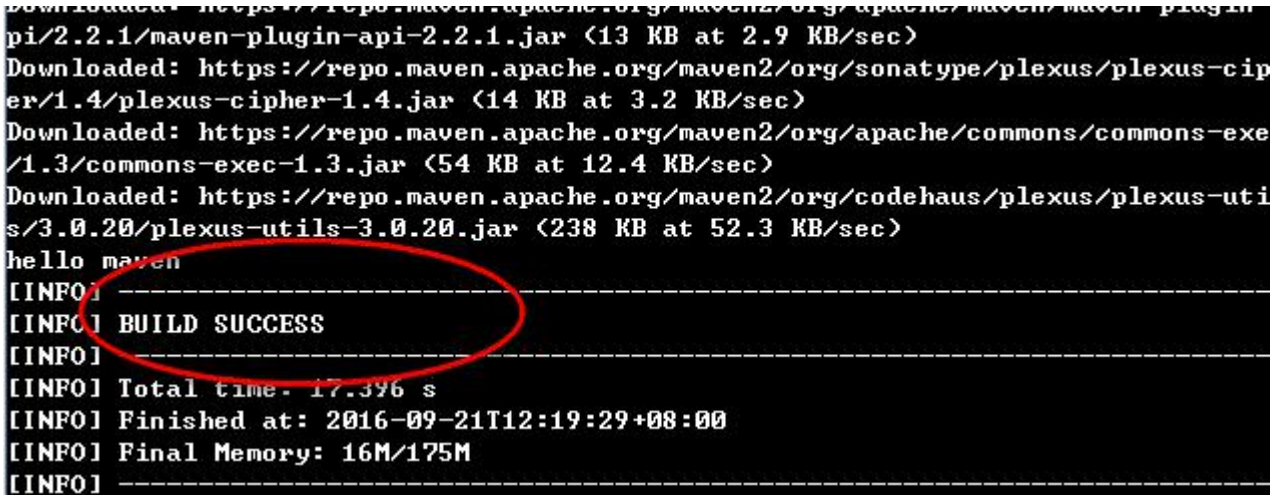
cmd下面，进入项目的根目录

1. 编译java文件

   mvn compile

2. 执行main 方法

mvn exec:java -Dexec.mainClass="com.xxxx.demo.Hello"



注：第一次下载会比较慢，要修改maven解压之后的conf目录下的settings.xml。

```
1.1. 修改默认仓库位置
打开maven目录 -> conf -> settings.xml
添加仓库位置配置
<localRepository>F:/m2/repository</localRepository>
注：仓库位置改为自己本机的指定目录，"/"不要写反

1.2. 更换阿里镜像,加快依赖下载
<mirror>
    <id>nexus-aliyun</id>
    <mirrorOf>central</mirrorOf>
    <name>Nexus aliyun</name>
    <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

如果编译不成功，可能出现的问题

```
1. 不是使用管理员权限执行dos命令
2. JDK环境配置有问题，重装JDK
3. 代码编写时，类里面没设置包名（如果编译时类没加包名，执行时也不需要加包名）
```

4. Maven命令

作为开发利器的maven，为我们提供了十分丰富的命令，了解maven的命令行操作并熟练运用常见的maven命令还是十分必要的，即使譬如IDEA等工具给我提供了图形界面化工具，但其底层还是依靠maven命令来驱动的。

*Maven*的命令格式如下：

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

命令代表的含义：执行`plugin-name`插件的`goal-name`目标

4.1. 常用命令

命令	描述
mvn clean	清理项目生产的临时文件,一般是模块下的target目录
mvn compile	编译源代码，一般编译模块下的src/main/java目录
mvn package	项目打包工具,会在模块下的target目录生成jar或war等文件
mvn test	测试命令,或执行src/test/java/下junit的测试用例.
mvn –version	显示版本信息
mvn install	将打包的jar/war文件复制到你的本地仓库中,供其他模块使用
mvn deploy	将打包的文件发布到远程参考,提供其他人员进行下载依赖
mvn site	生成项目相关信息的网站
mvn eclipse:eclipse	将项目转化为Eclipse项目
mvn dependency:tree	打印出项目的整个依赖树
mvn archetype:generate	创建Maven的普通java项目
mvn tomcat7:run	在tomcat容器中运行web应用
mvn jetty:run	调用 Jetty 插件的 Run 目标在 Jetty Servlet 容器中启动 web 应用

注意：运行maven命令的时候，首先需要定位到maven项目的目录，也就是项目的pom.xml文件所在的目录。否则，必以通过参数来指定项目的目录。

4.2. 命令参数

上面列举的只是比较通用的命令，其实很多命令都可以携带参数以执行更精准的任务。

4.2.1. -D 传入属性参数

例如：

```
mvn package -Dmaven.test.skip=true
```

以 `-D` 开头，将 `maven.test.skip` 的值设为 `true` ,就是告诉maven打包的时候跳过单元测试。同理， `mvn deploy-Dmaven.test.skip=true` 代表部署项目并跳过单元测试。

4.2.2. -P 使用指定的Profile配置

比如项目开发需要有多个环境，一般为开发，测试，预发，正式4个环境，在pom.xml中的配置如下：

```
<profiles>
      <profile>
```

```
            <id>dev</id>
            <properties>
                <env>dev</env>
            </properties>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
        </profile>
        <profile>
            <id>qa</id>
            <properties>
                <env>qa</env>
            </properties>
        </profile>
        <profile>
            <id>pre</id>
            <properties>
                <env>pre</env>
            </properties>
        </profile>
        <profile>
            <id>prod</id>
            <properties>
                <env>prod</env>
            </properties>
        </profile>
    </profiles>

    ......

    <build>
        <filters>
            <filter>config/${env}.properties</filter>
        </filters>
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>

        ......

    </build>
```

`profiles`定义了各个环境的变量`id`，`filters`中定义了变量配置文件的地址，其中地址中的环境变量就是上面`profile`中定义的值，`resources`中是定义哪些目录下的文件会被配置文件中定义的变量替换。

通过maven可以实现按不同环境进行打包部署，例如：

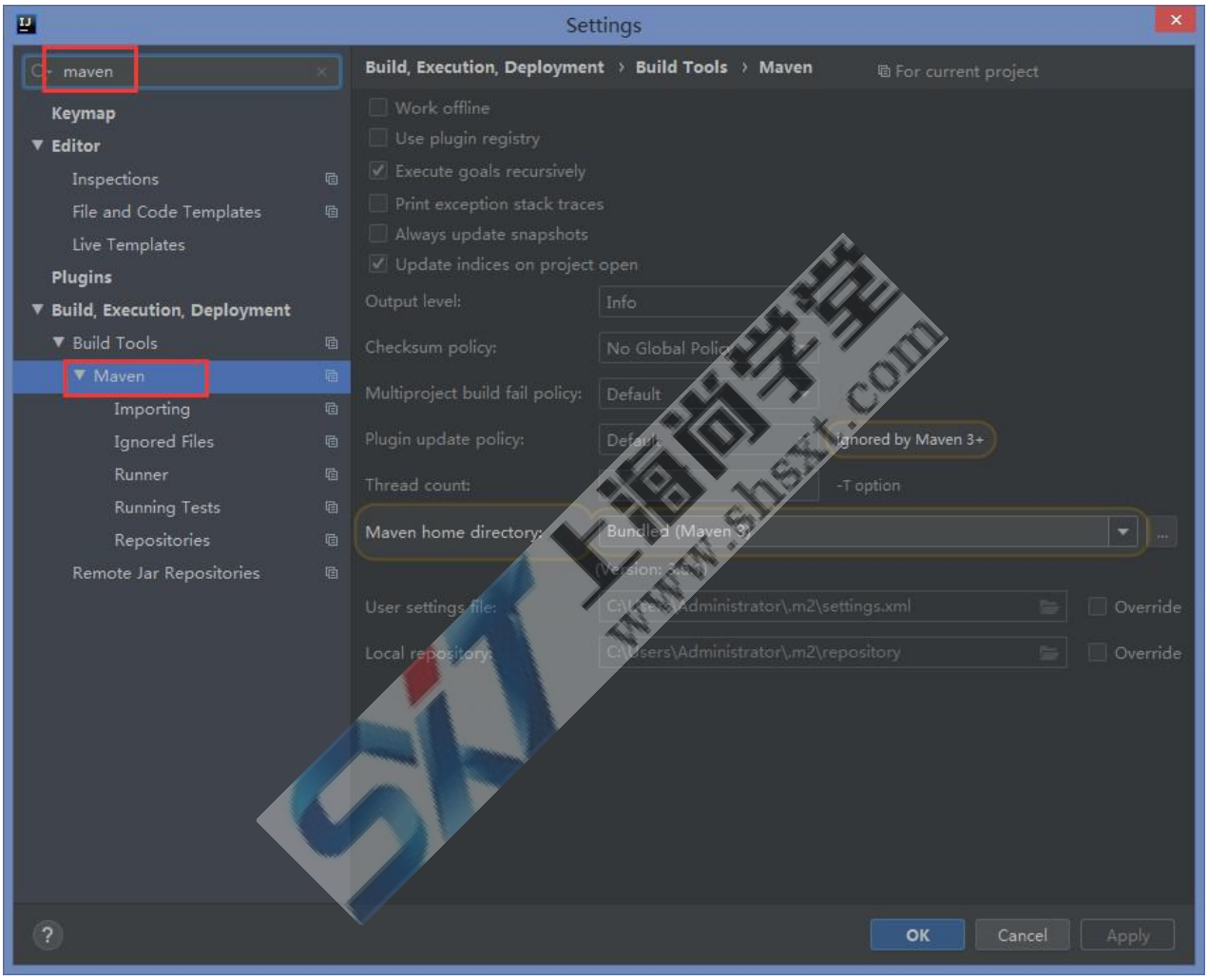
```
mvn package -Pdev  -Dmaven.test.skip=true
```

表示打包本地环境，并跳过单元测试

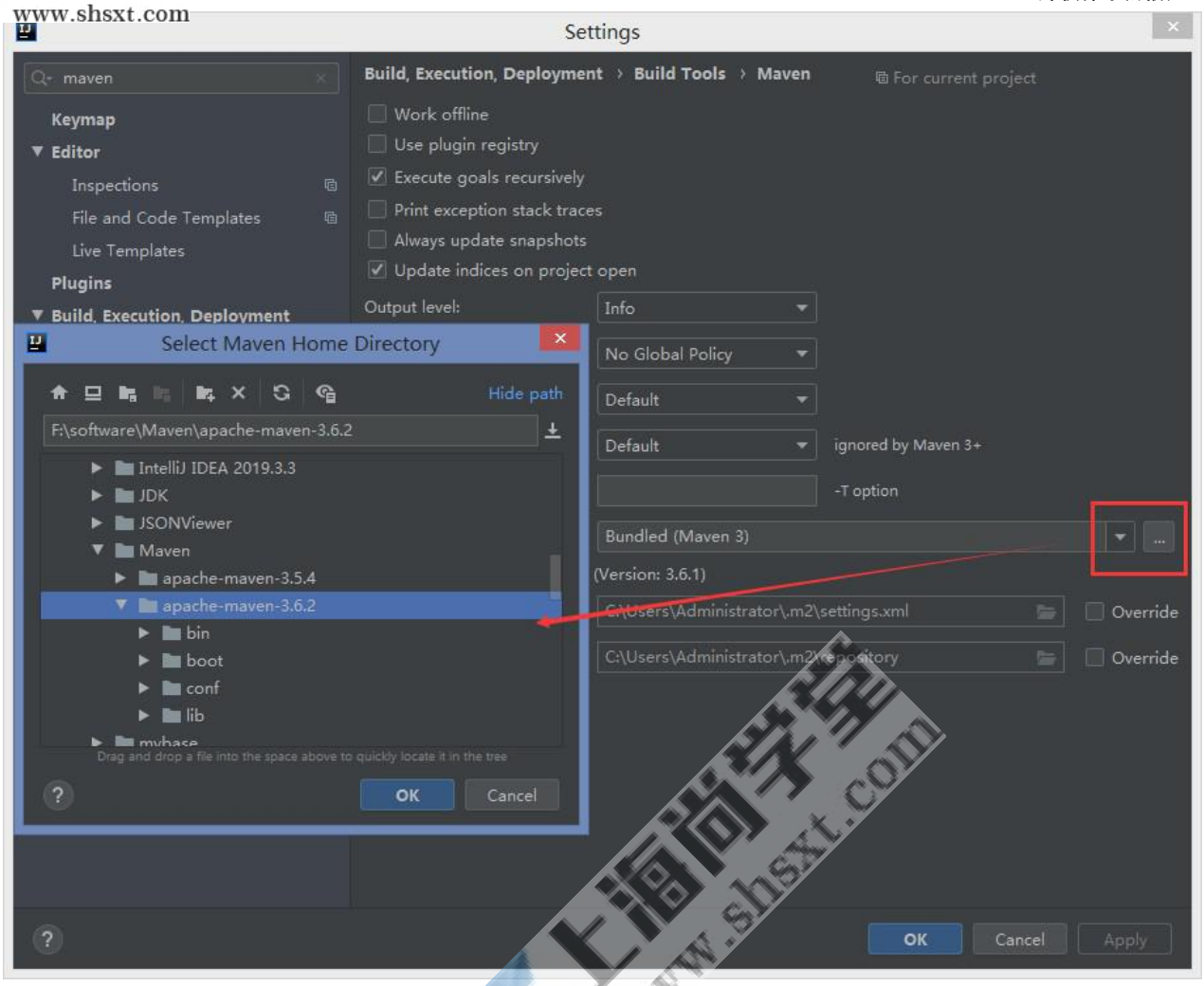
5. IDEA编辑器集成Maven环境

5.1. 设置Maven版本

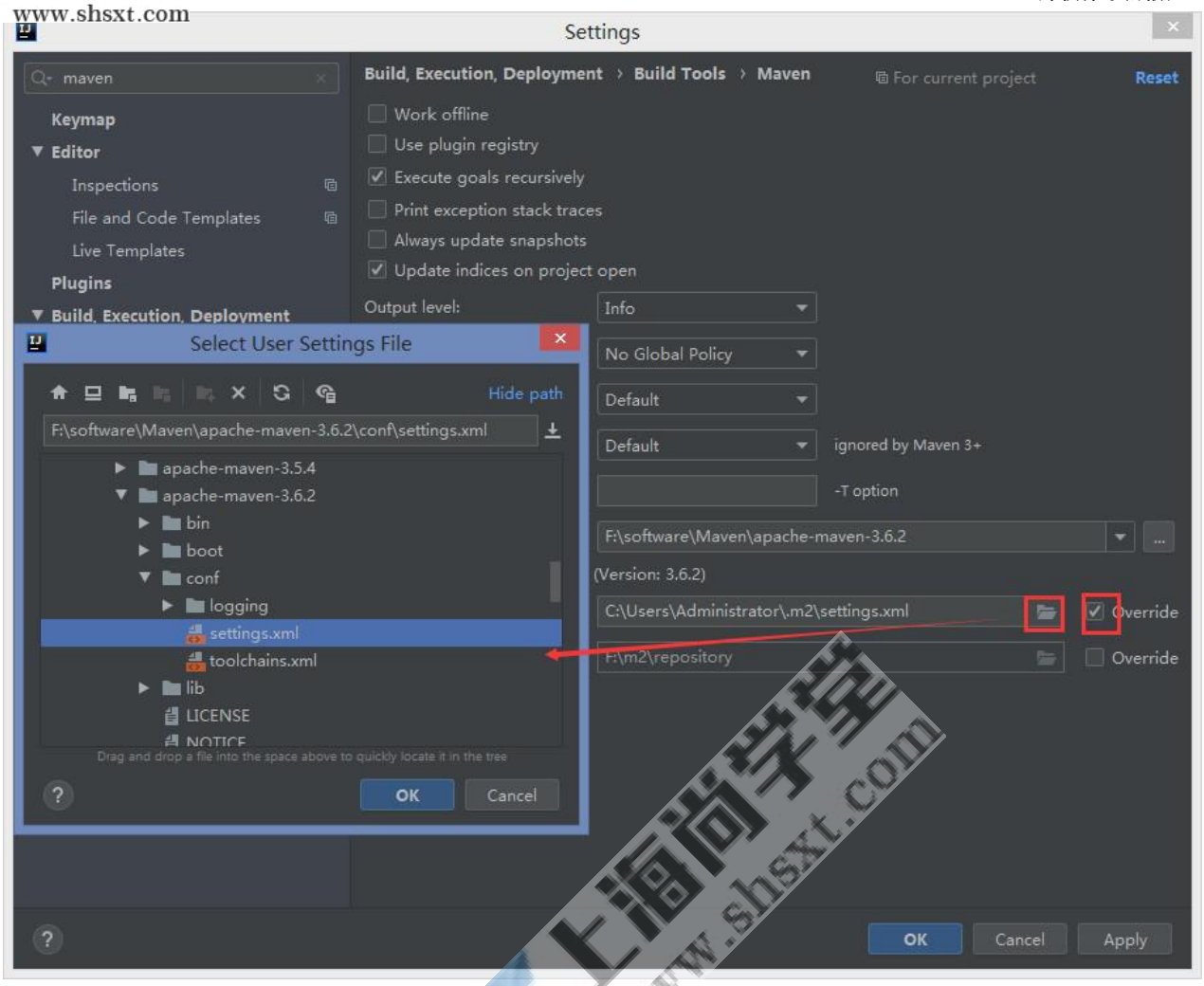
选 择 "File" —> "Other Settings" —> "Settings for New Projects..." —> 搜 索 "Maven"



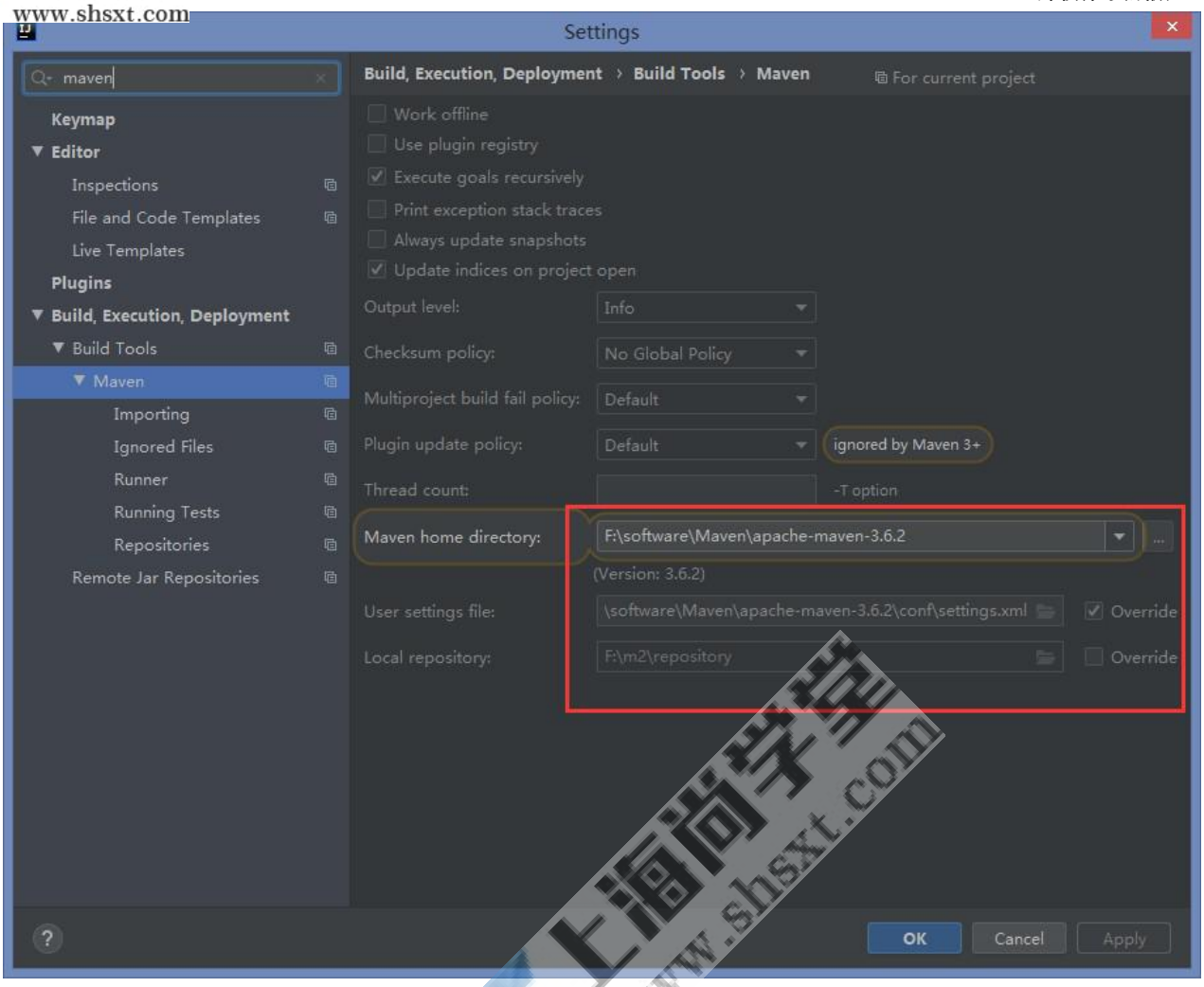
选择下载好的maven版本（目录选到bin目录的上一级目录）



设置settings.xml文件



设置好之后，选择 "Apply" 或者 "OK"

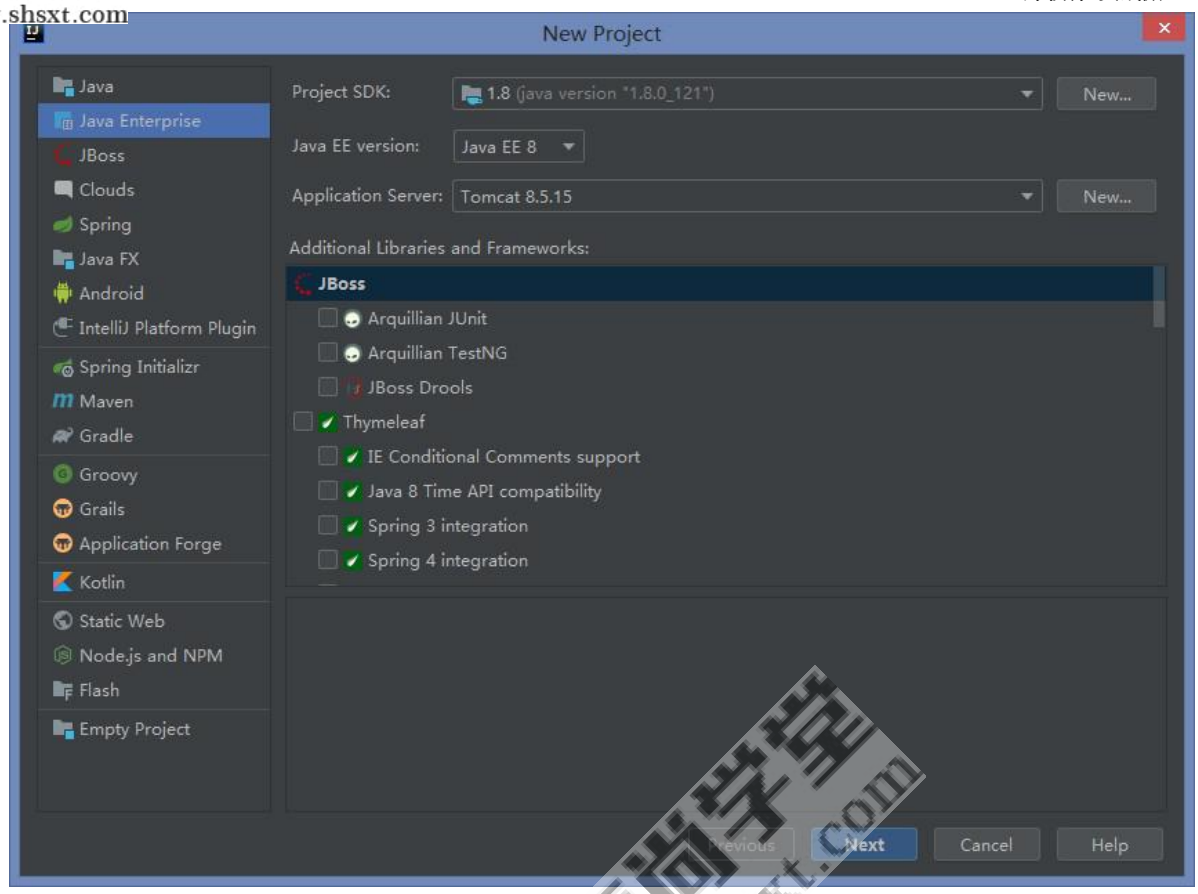


6. Maven项目的创建

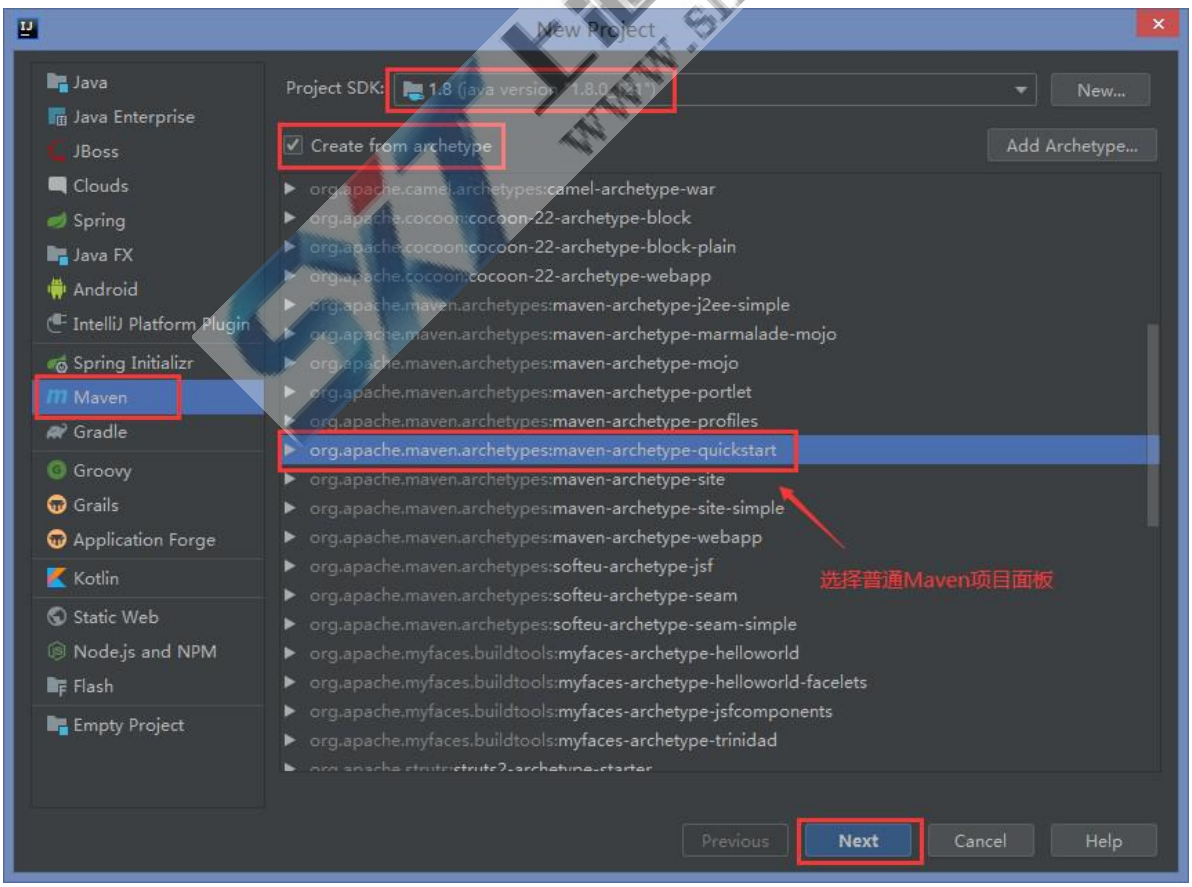
6.1. 创建 Java项目

6.1.1. 新建项目

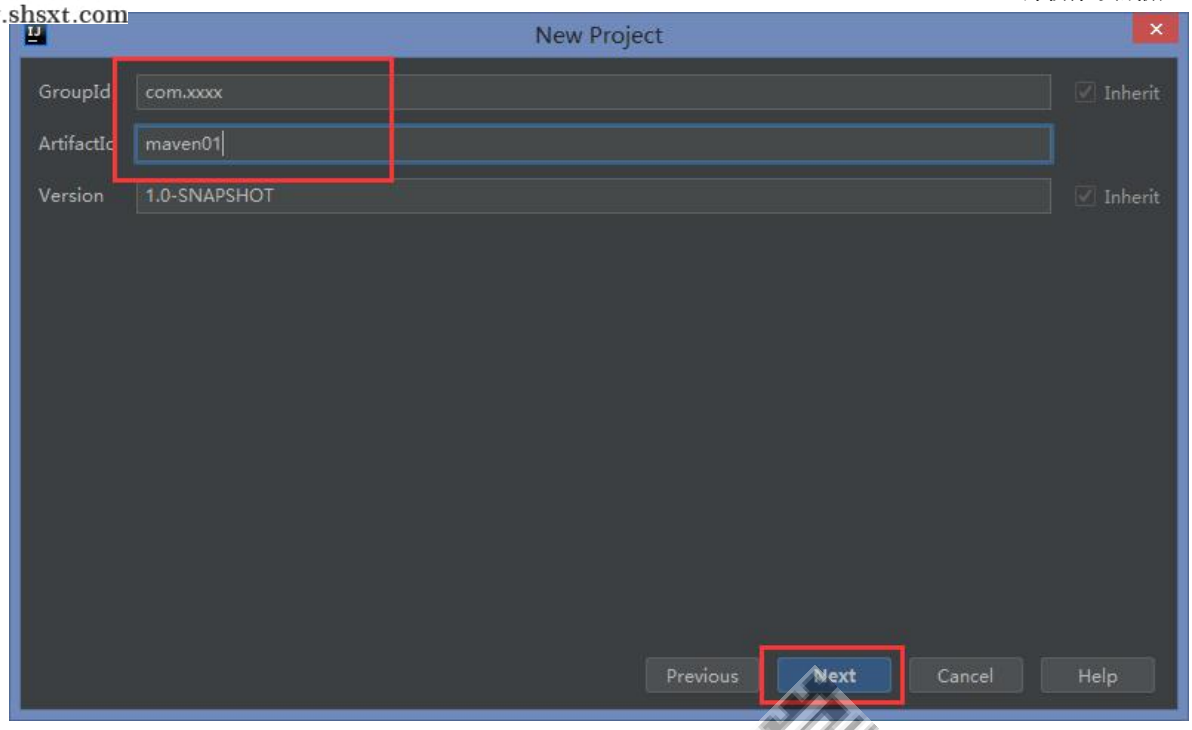
1. 选择 "File" —> "New" —> "Project"



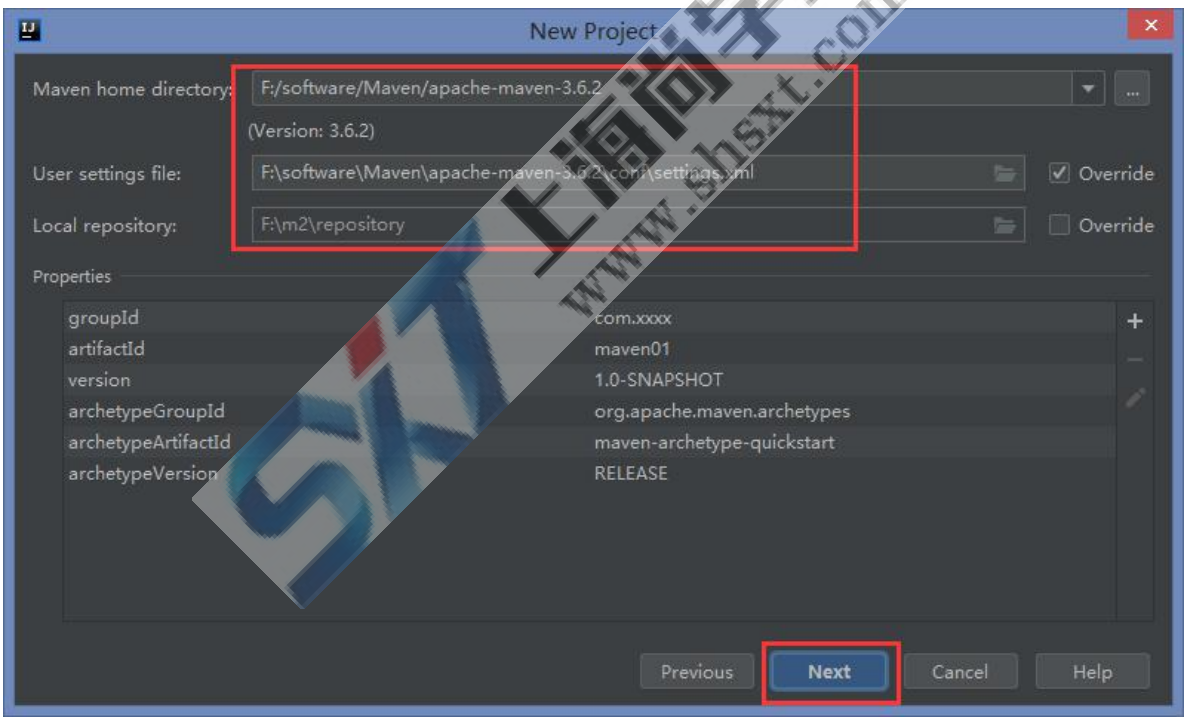
2. 选择"Maven"，设置JDK版本，选择maven项目的模板



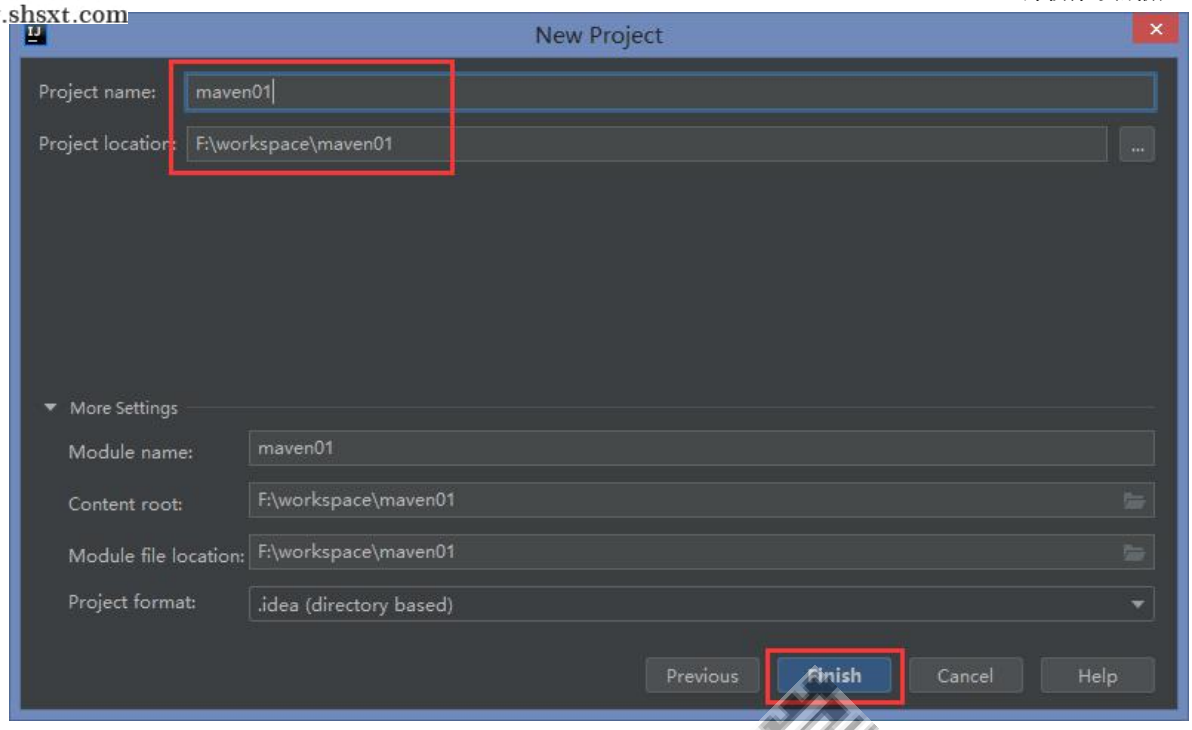
3. 设置项目的 GroupId 和 ArtifactId



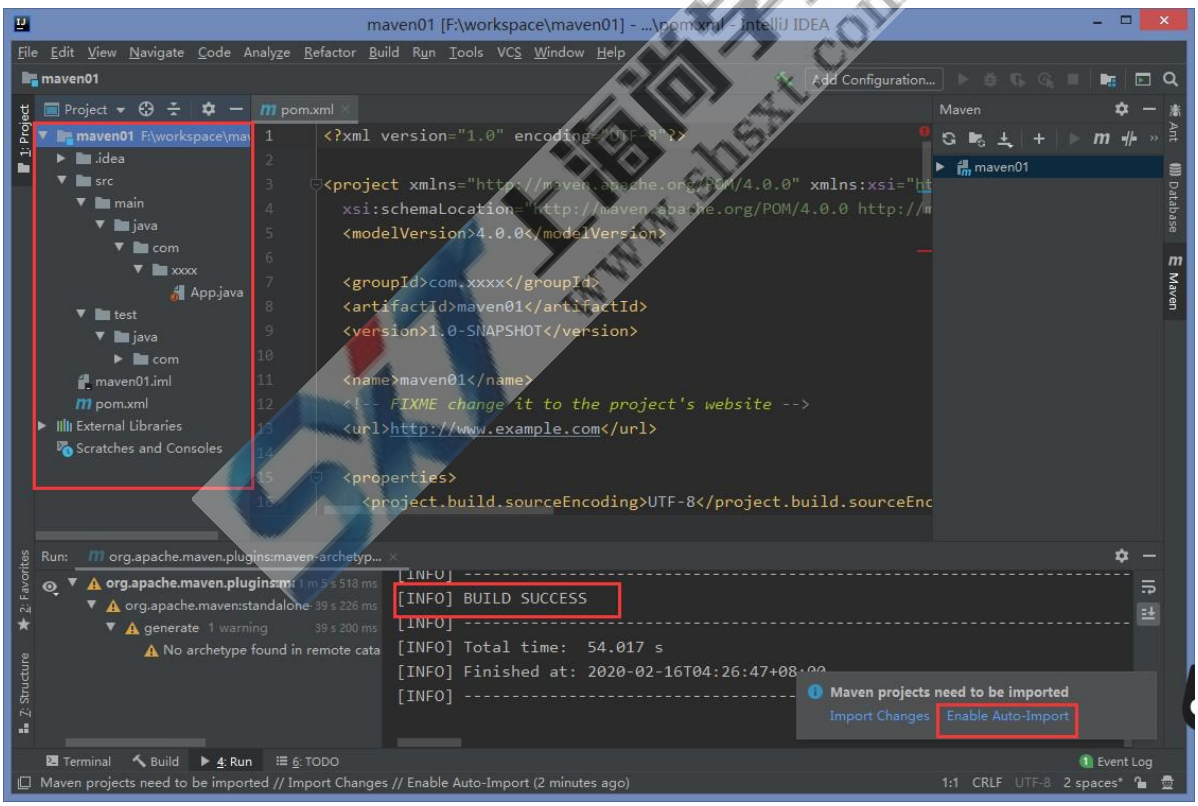
4. 检查Maven环境，选择 "Next"



5. 检查项目名和工作空间，选择 "Finish"



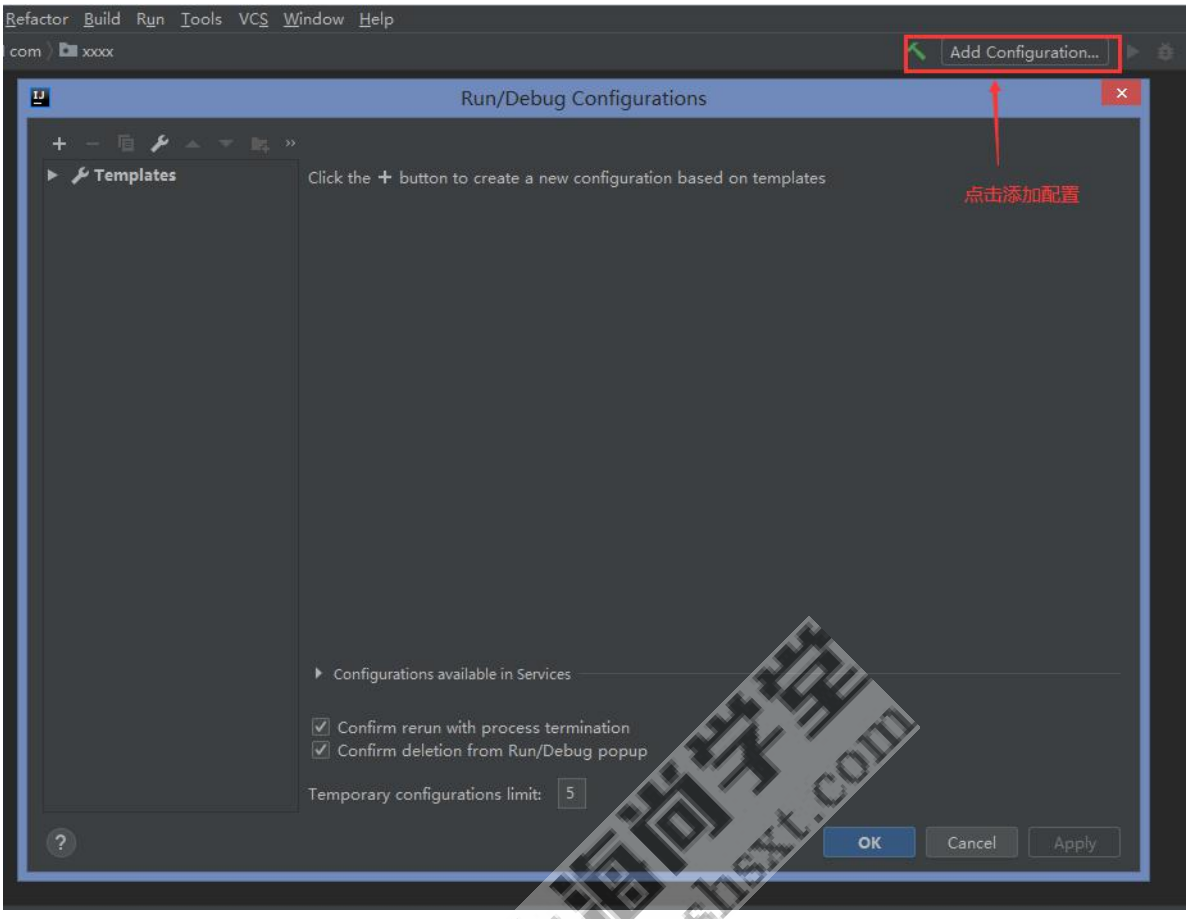
6. 等待项目创建，下载资源，创建完成后目录结构如下



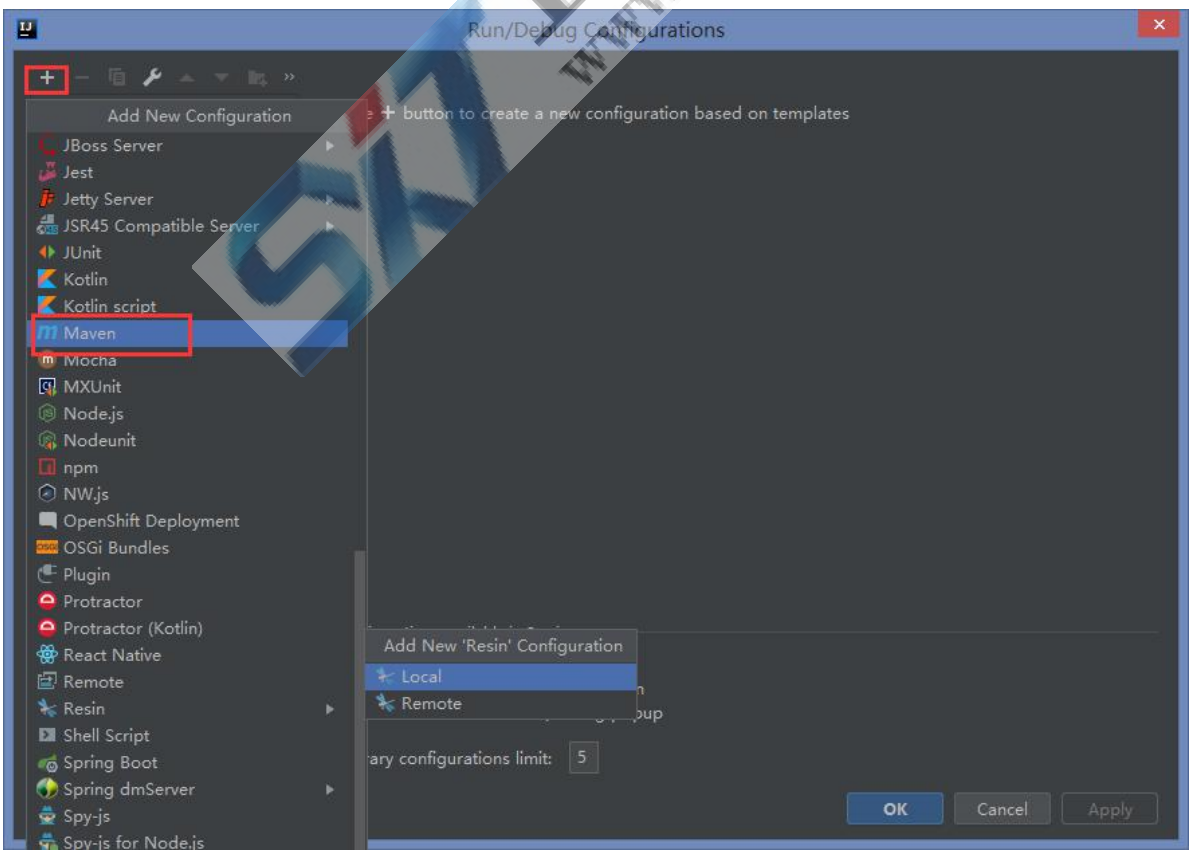
注：右下角弹出的提示框，选择 "Enable Auto-Import"（Maven启动自动导入）

6.1.2. 编译项目

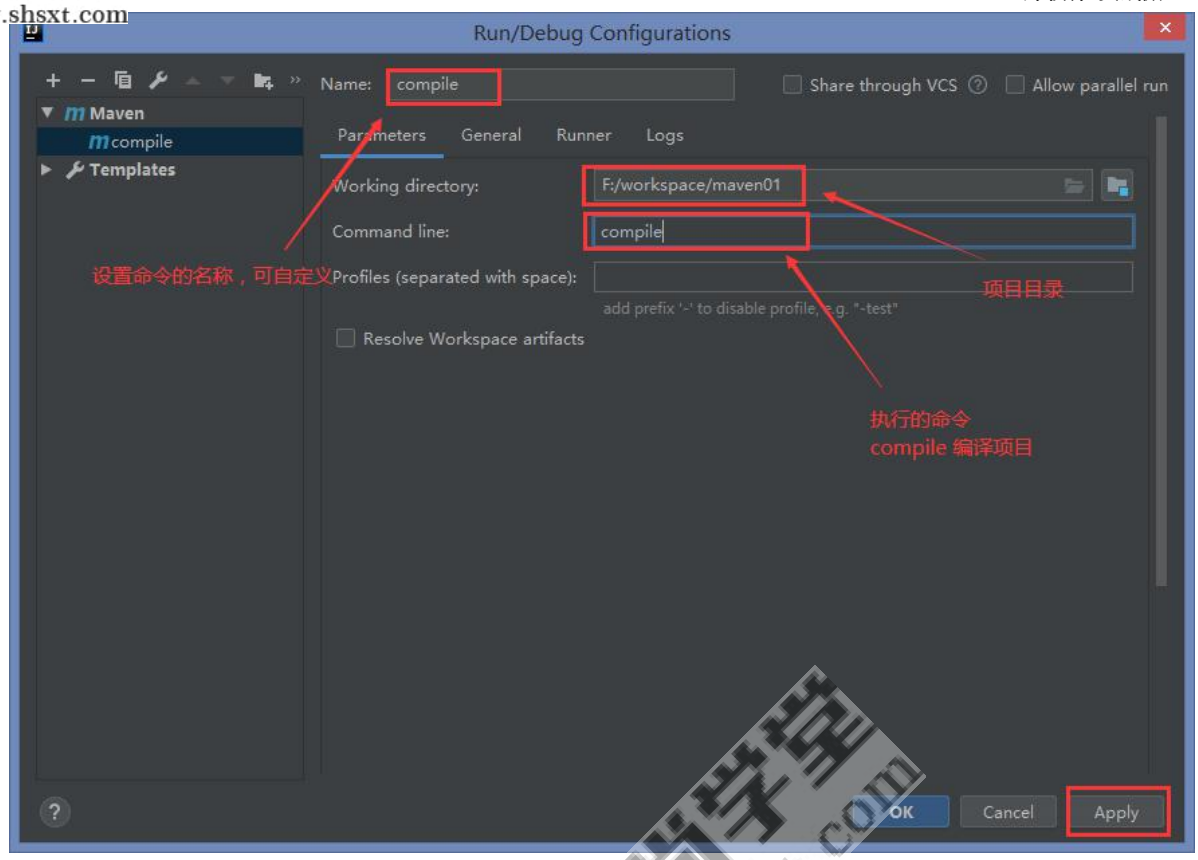
1. 点击右上角的 "Add Configurations "，打开 "Run/Debug Configurations" 窗口



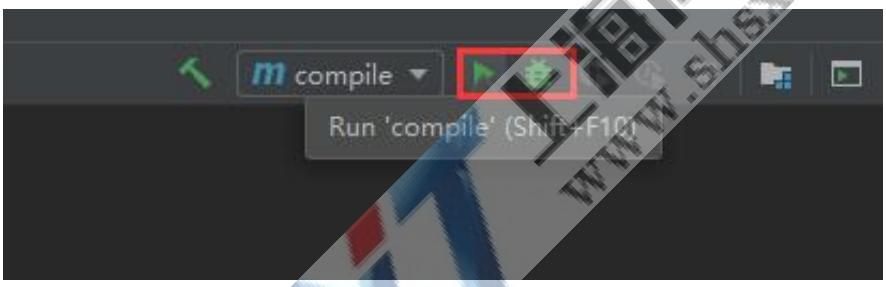
2. 点击左上角的 "+" 号，选择 "Maven"



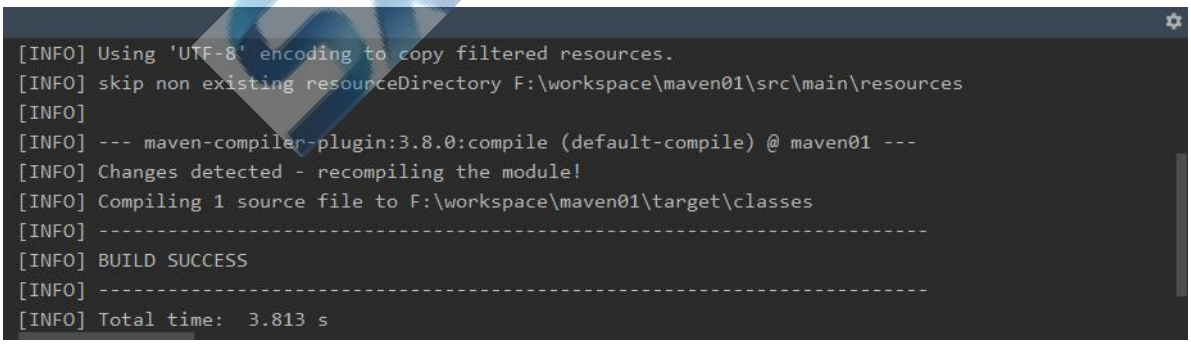
3. 设置编译项目的命令



4. 执行编译命令，两个图标分别代表"普通模式"和"调试模式"



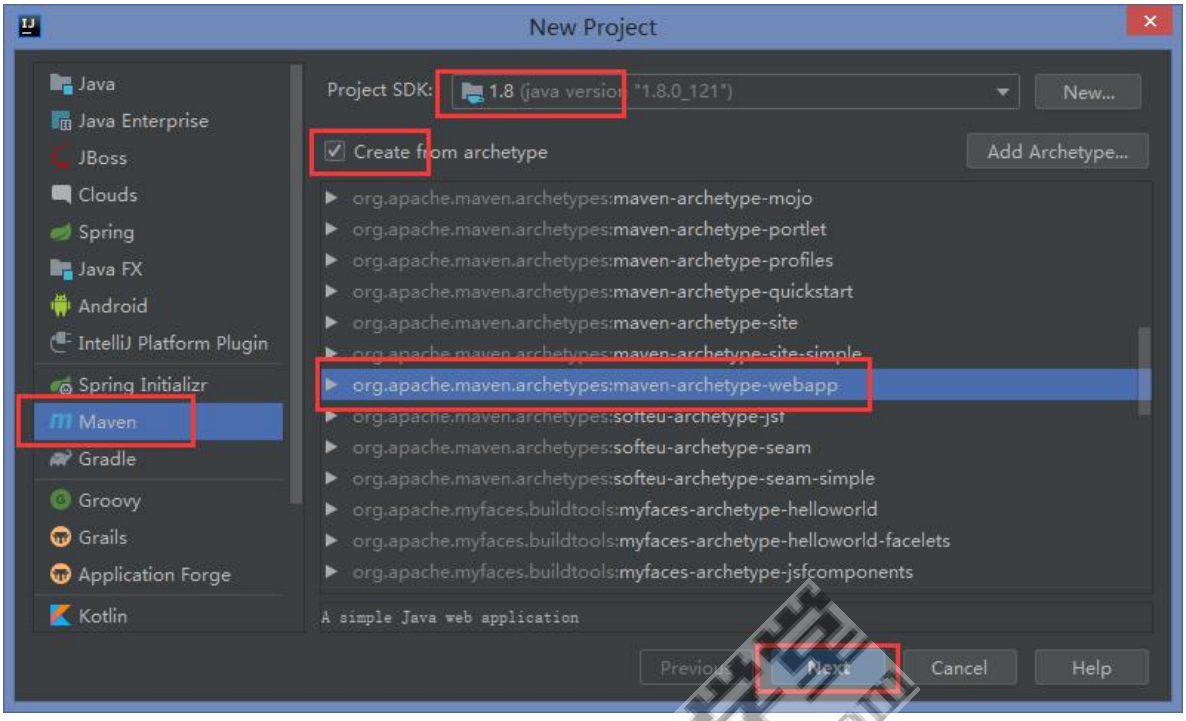
5. 编译成功



6.2. 创建 Web项目

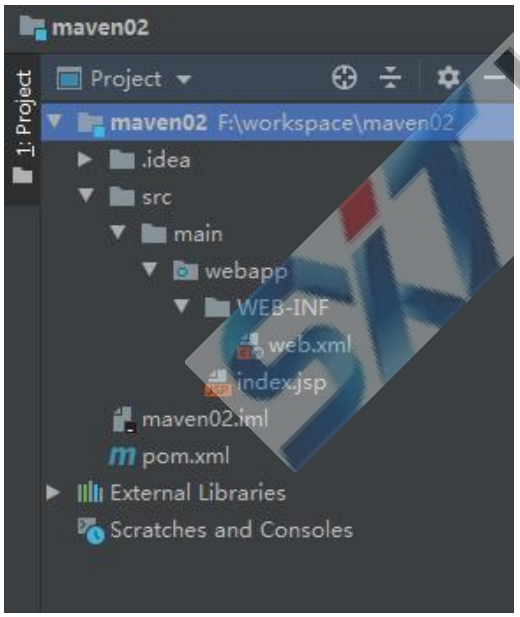
6.2.1. 创建项目

1. 创建Web项目与创建Java项目步骤基本一致，区别在于选择 Maven模板（web项目选择webapp），如图：



注：其他步骤与创建普通的Java项目相同。

2. 项目目录结构如下：



6.2.2. 启动项目

6.2.2.1. 修改 JDK 的版本

```
<!-- JDK的版本修改为1.8 -->
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

6.2.2.2. 设置单元测试的版本

```
<!-- junit的版本修改为4.12 -->
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

6.2.2.3. 删除pluginManagement标签

```
<!-- 将这个标签及标签中的内容全部删除 -->
<pluginManagement>
...
</pluginManagement>
```

6.2.2.4. 添加web部署的插件

在 build 标签中添加 plugins 标签

1. Jetty插件

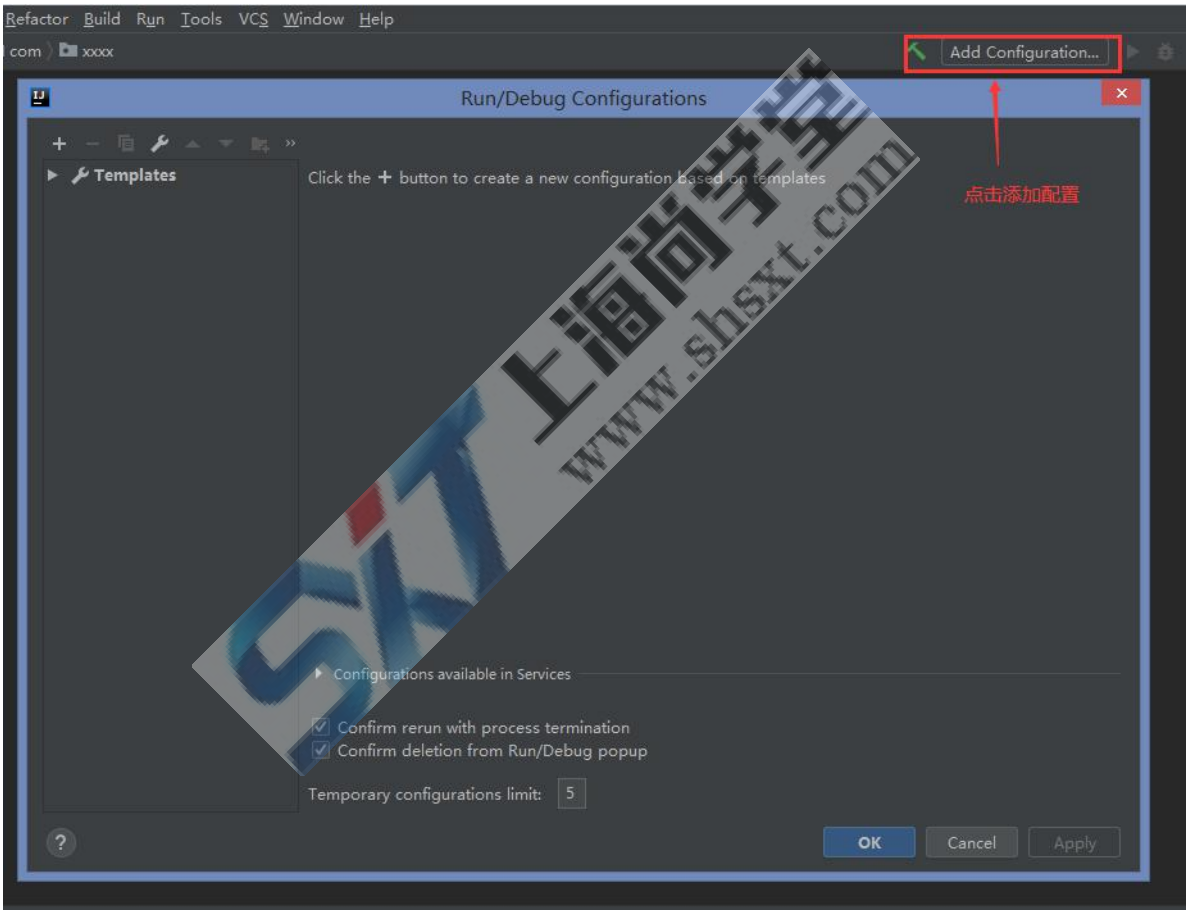
```
<!-- 设置在plugins标签中 -->
<plugin>
   <groupId>org.mortbay.jetty</groupId>
   <artifactId>maven-jetty-plugin</artifactId>
   <version>6.1.25</version>
   <configuration>
      <!-- 热部署，每10秒扫描一次 -->
      <scanIntervalSeconds>10</scanIntervalSeconds>
      <!-- 可指定当前项目的站点名 -->
      <contextPath>/test</contextPath>
      <connectors>
          <connector
implementation="org.mortbay.jetty.nio.SelectChannelConnector">
              <port>9090</port> <!-- 设置启动的端口号 -->
          </connector>
       </connectors>
    </configuration>
</plugin>
```

2. Tomcat插件

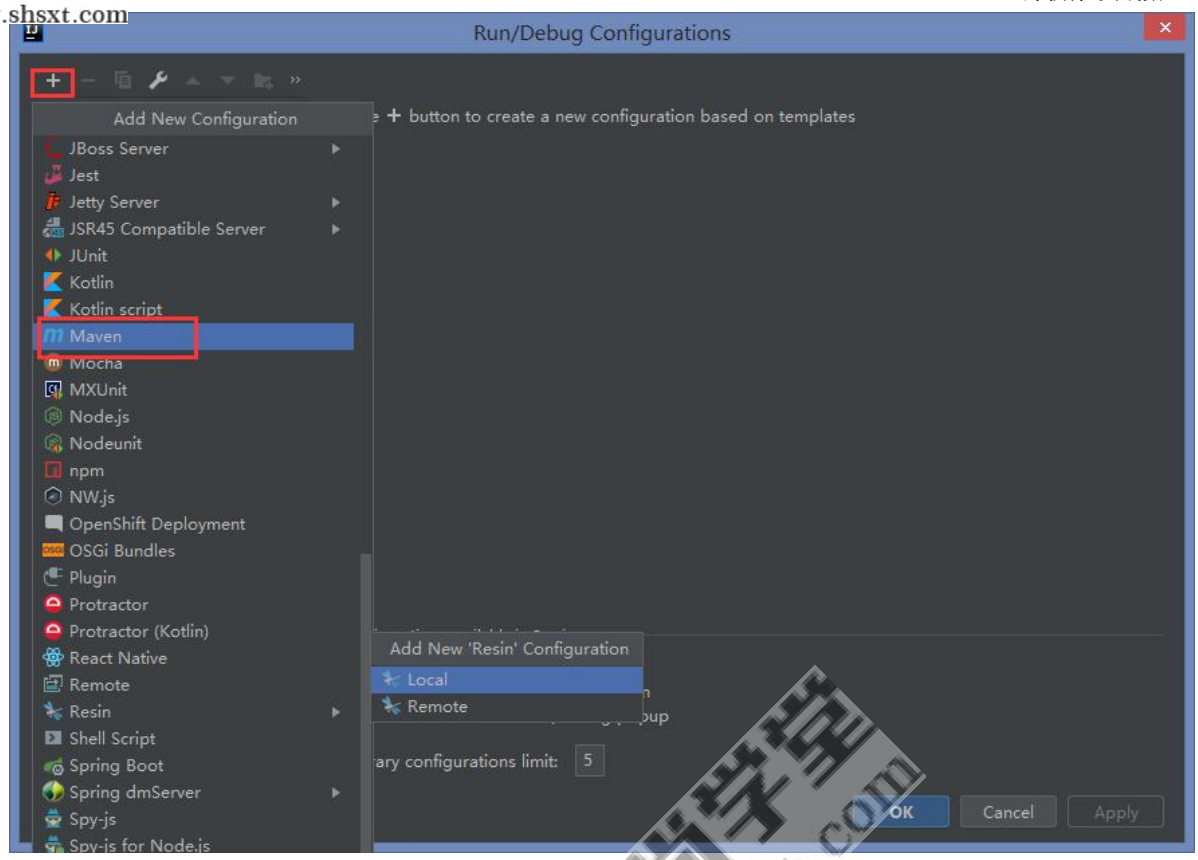
```
<!-- 设置在plugins标签中 -->
<plugin>
  <groupId>org.apache.tomcat.maven</groupId>
  <artifactId>tomcat7-maven-plugin</artifactId>
  <version>2.1</version>
  <configuration>
    <port>8081</port> <!-- 启动端口 默认:8080 -->
    <path>/test</path> <!-- 项目的站点名，即对外访问路径 -->
    <uriEncoding>UTF-8</uriEncoding> <!-- 字符集编码 默认：ISO-8859-1 -->
    <server>tomcat7</server> <!-- 服务器名称 -->
  </configuration>
</plugin>
```

6.2.2.5. 启动项目

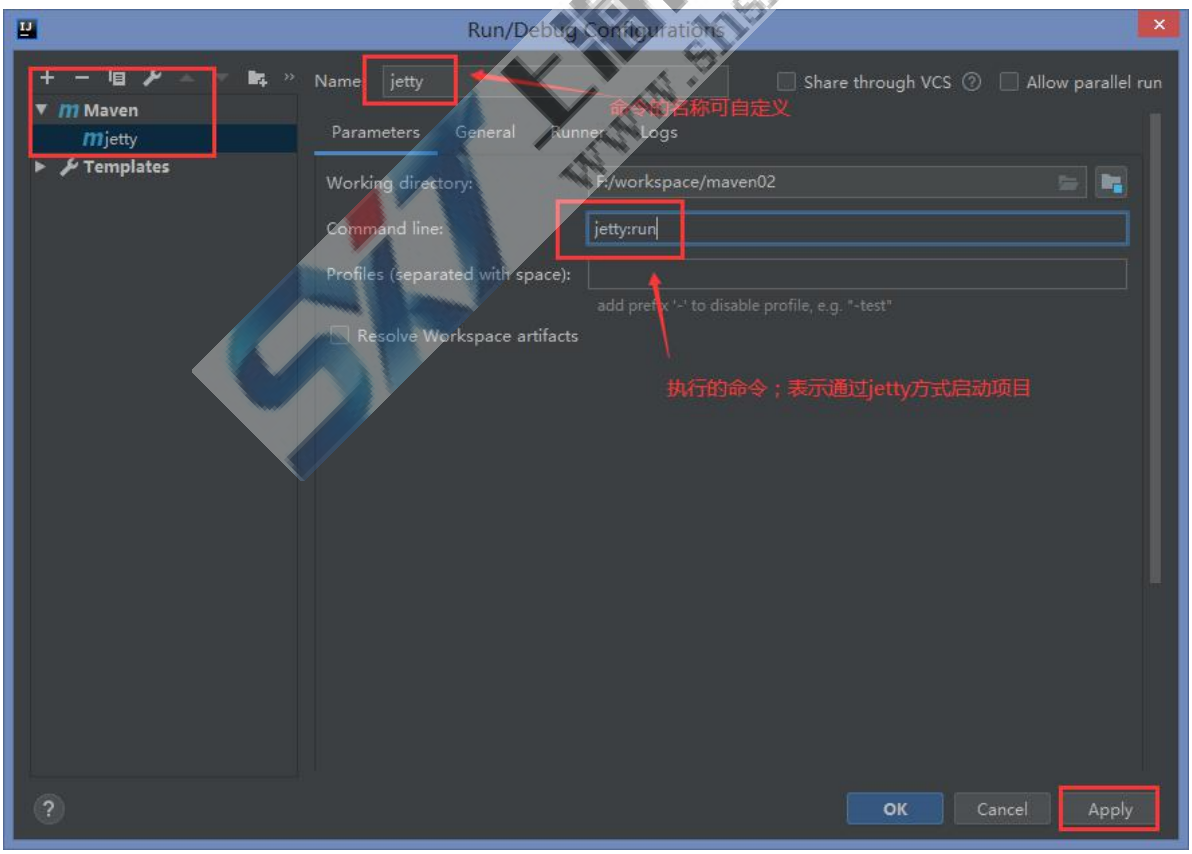
1. 点击右上角的 "Add Configurations "，打开 "Run/Debug Configurations" 窗口



2. 点击左上角的 "+" 号，选择 "Maven"



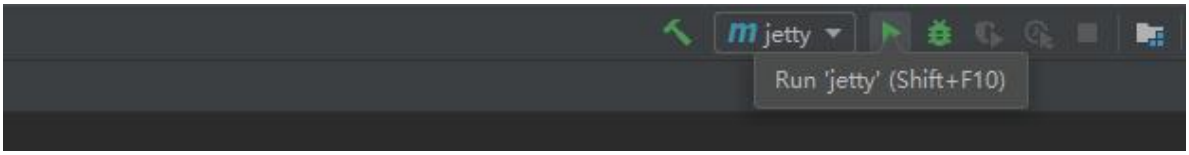
3. Jetty插件配置



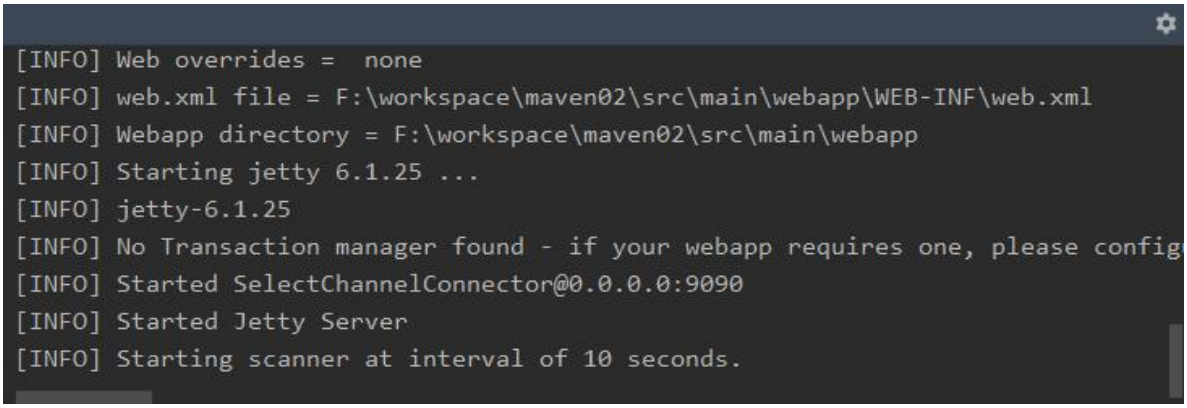
也可以输入命令指定端口启动

```
jetty:run -Djetty.port=9090  # 需要将插件配置中的port标签去掉
```

点击启动图标，启动服务

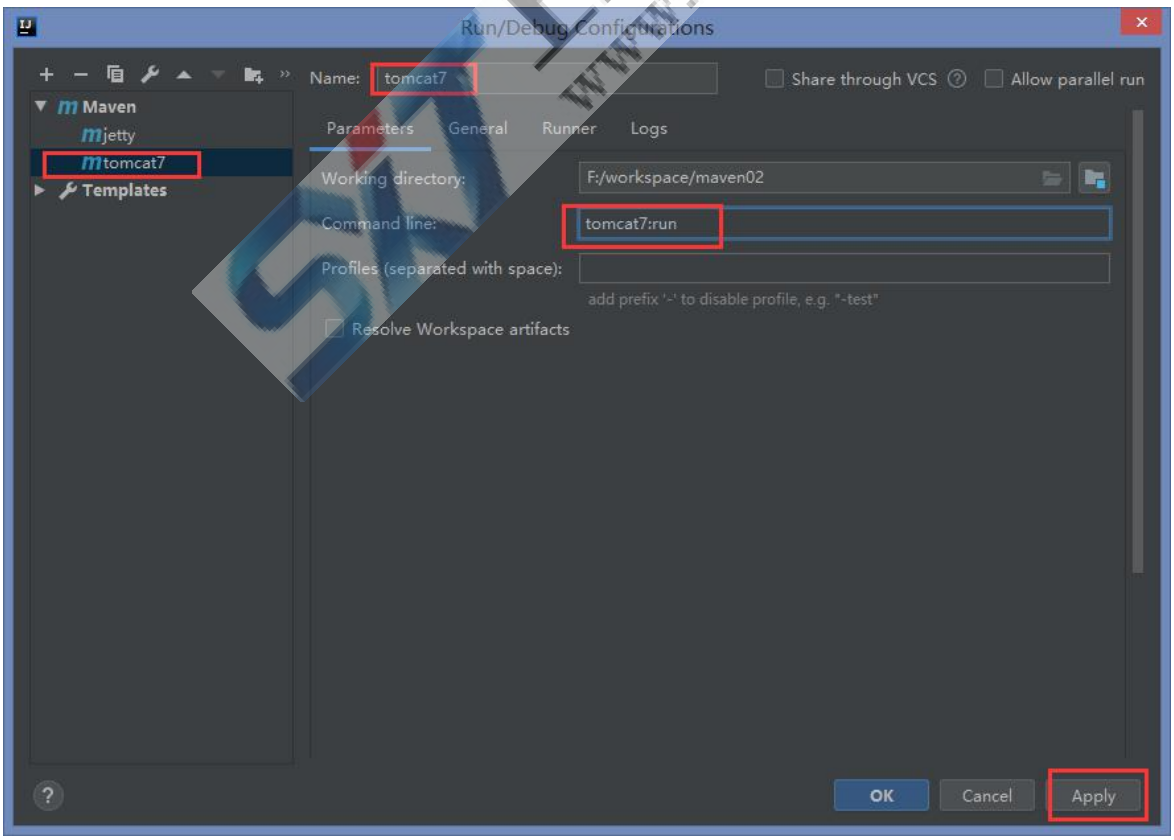


启动成功



浏览器访问效果

4. Tomcat插件配置



启动方式如上，启动成功

```
[INFO] No sources to compile
[INFO]
[INFO] <<< tomcat7-maven-plugin:2.1:run (default-cli) < process-classes @ maven0
[INFO]
[INFO]
[INFO] --- tomcat7-maven-plugin:2.1:run (default-cli) @ maven02 ---
[INFO] Running war on http://localhost:8080/test
[INFO] Using existing Tomcat server configuration at F:\workspace\maven02\target
[INFO] create webapp with contextPath: /test
```

浏览器访问 **http://localhost:8080/test**

- Maven 依 赖 仓 库 ：

  https://mvnrepository.com/

- Tomcat7插件的命令：

  https://tomcat.apache.org/maven-plugin-trunk/tomcat7-maven-plugin/plugin-info.html

7. Maven仓库的基本概念

当第一次运行Maven命令的时候， 你需要Internet链接， 因为它需要从网上下载一些文件。 那么它从哪里下载呢？ 它是从Maven默认的远程库下载的。 这个远程仓库有Maven的核心插件和可供下载的jar文件。

对于Maven来说， 仓库只分为两类： **本地仓库和远程仓库**。

当Maven根据坐标寻找构件的时候，它首先会查看本地仓库，如果本地仓库存在，则直接使用； 如果本地没有，Maven就会去远程仓库查找，发现需要的构件之后，下载到本地仓库再使用。 如果本地仓库和远程仓库都没有，Maven就会报错。

远程仓库分为三种： 中央仓库，私服， 其他公共库。

中央仓库是默认配置下，Maven下载jar包的地方。

私服是另一种特殊的远程仓库，为了节省带宽和时间，应该在局域网内架设一个私有的仓库服务器，用其代理所有外部的远程仓库。 内部的项目还能部署到私服上供其他项目使用。

一般来说，在Maven项目目录下，没有诸如lib/这样用来存放依赖文件的目录。 当Maven在执行编译或测试时，如果需要使用依赖文件，它总是基于坐标使用本地仓库的依赖文件。

默认情况下，每个用户在自己的用户目录下都有一个路径名为.m2/repository/的仓库目录。 有时候，因为某些原因（比如c盘空间不足）,需要修改本地仓库目录地址。

对于仓库路径的修改，可以通过maven 配置文件conf 目录下settings.xml来指定仓库路径

```
<!-- 设置到指定目录中，路径的斜杆不要写反 -->
<settings>
  <localRepository>D:/m2/repository</localRepository>
</settings>
```

7.1. 中央仓库

由于原始的本地仓库是空的，maven必须知道至少一个可用的远程仓库，才能执行maven命令的时候下载到需要的构件。中央仓库就是这样一个默认的远程仓库。

maven-model-builder-3.3.9.jar maven自动的 jar 中包含了一个 超级POM。定义了默认中央仓库的位置。

中央仓库包含了2000多个开源项目，接收每天1亿次以上的访问。

7.2. 私服

私服是一种特殊的远程仓库，它是架设在局域网内的仓库服务， 私服代理广域网上的远程仓库，供局域网内的maven用户使用。 当maven需要下载构件时， 它去私服当中找，如果私服没有， 则从外部远程仓库下载，并缓存在私服上， 再为maven提供。

此外，一些无法从外部仓库下载的构件也能从本地上传到私服提供局域网中其他人使用

配置方式项目pom.xml 配置

```
<repositories>
    <repository>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
        <id>public</id>
        <name>Public Repositories</name>
        <url>http://192.168.0.96:8081/content/groups/public/</url>
    </repository>
    <repository>
        <id>getui-nexus</id>
<url>http://mvn.gt.igexin.com/nexus/content/repositories/releases/</url>
    </repository>
</repositories>
```

公司内部应该建立私服：

- 节省自己的外网带宽
- 加速maven构建
- 部署第三方控件
- 提高稳定性
- 降低中央仓库的负荷

7.3. 其他公共库

常用的阿里云仓库配置

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

8. Maven环境下构建多模块项目

使用maven 提供的多模块构建的特性完成maven 环境下多个模块的项目的管理与构建。

这里以四个模块为例来搭建项目,以达到通俗易懂的初衷

模块 maven_parent -- 基模块,就是常说的parent （pom）

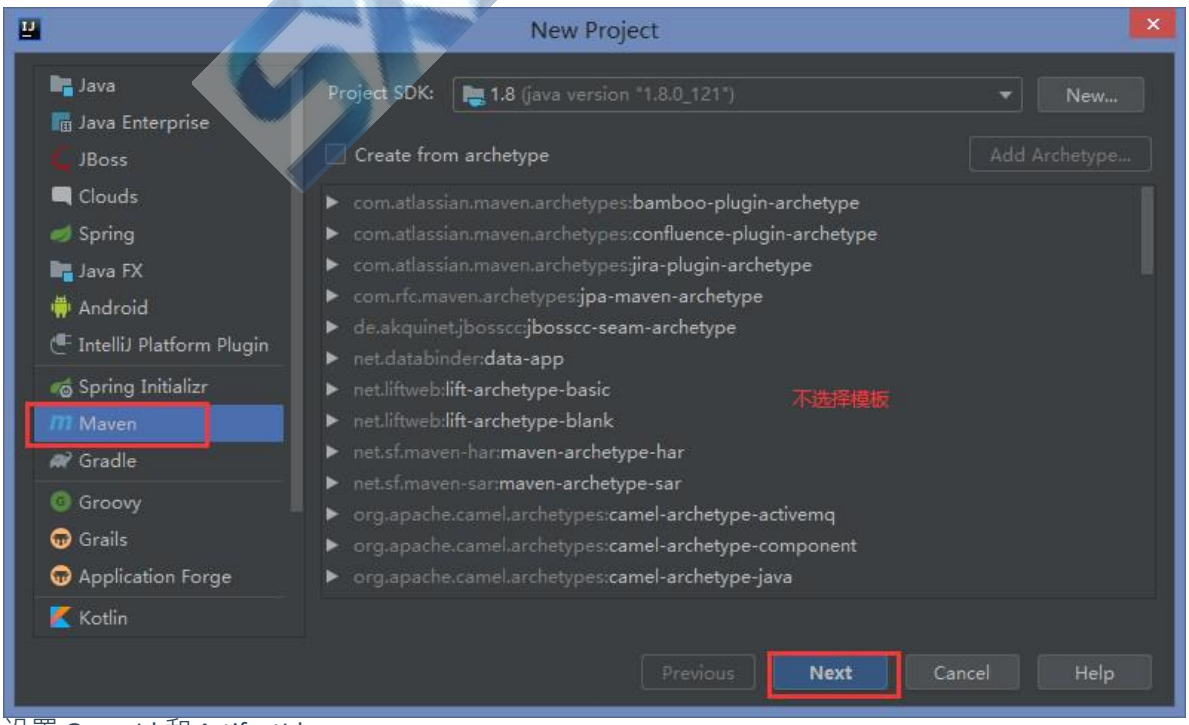
模块 maven_dao -- 数据库的访问层，例如jdbc操作（jar）

模块 maven_service -- 项目的业务逻辑层（jar）

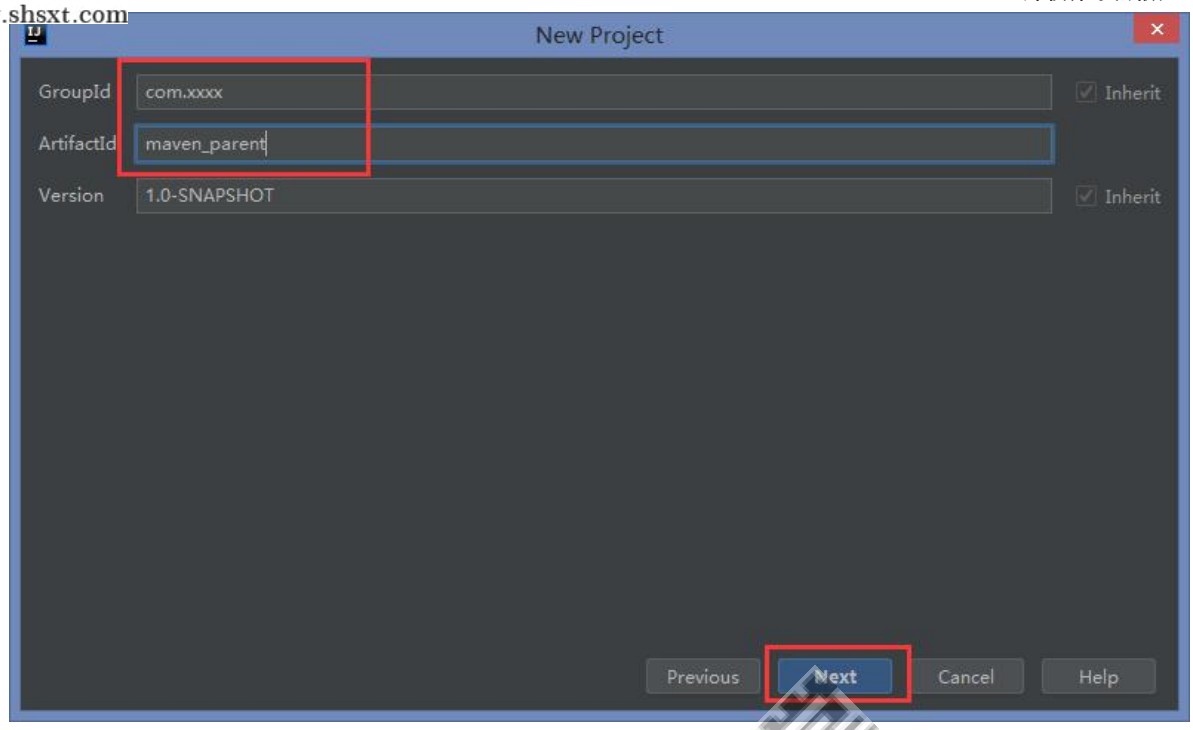
模块 maven_controller -- 用来接收请求，响应数据（war）

8.1. 创建 maven_parent 项目

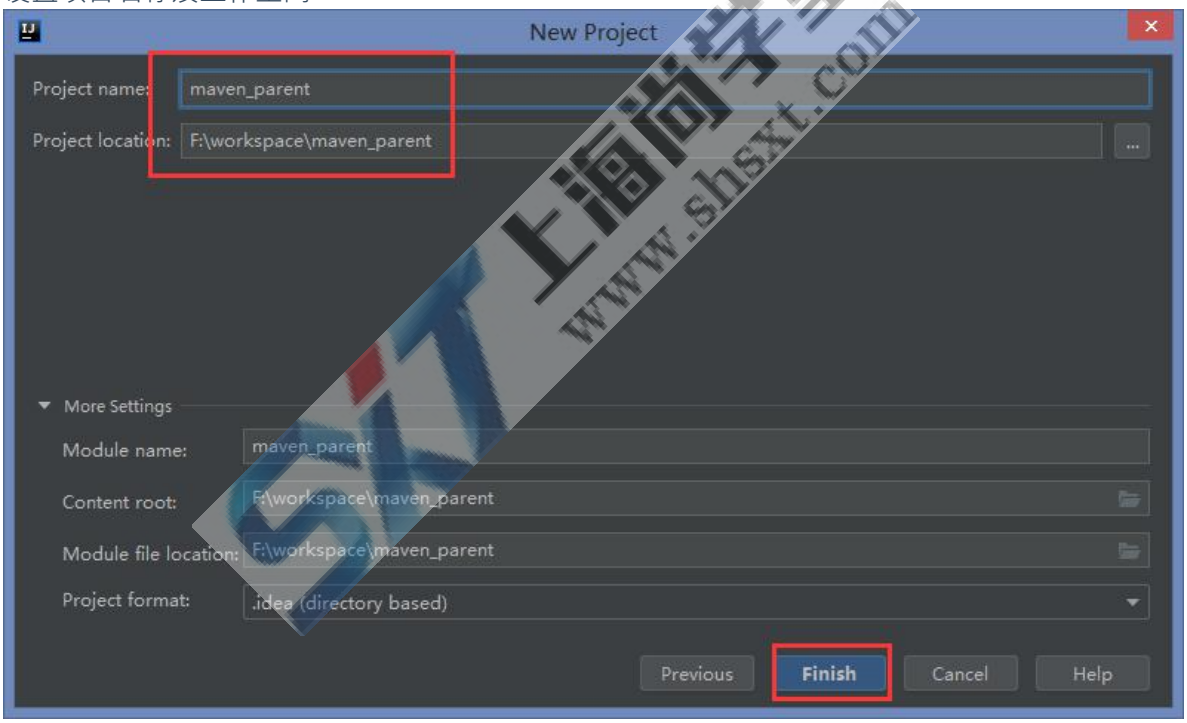
1. 选择 File —> Settings —> Project



2. 设置 GroupId 和 ArtifactId

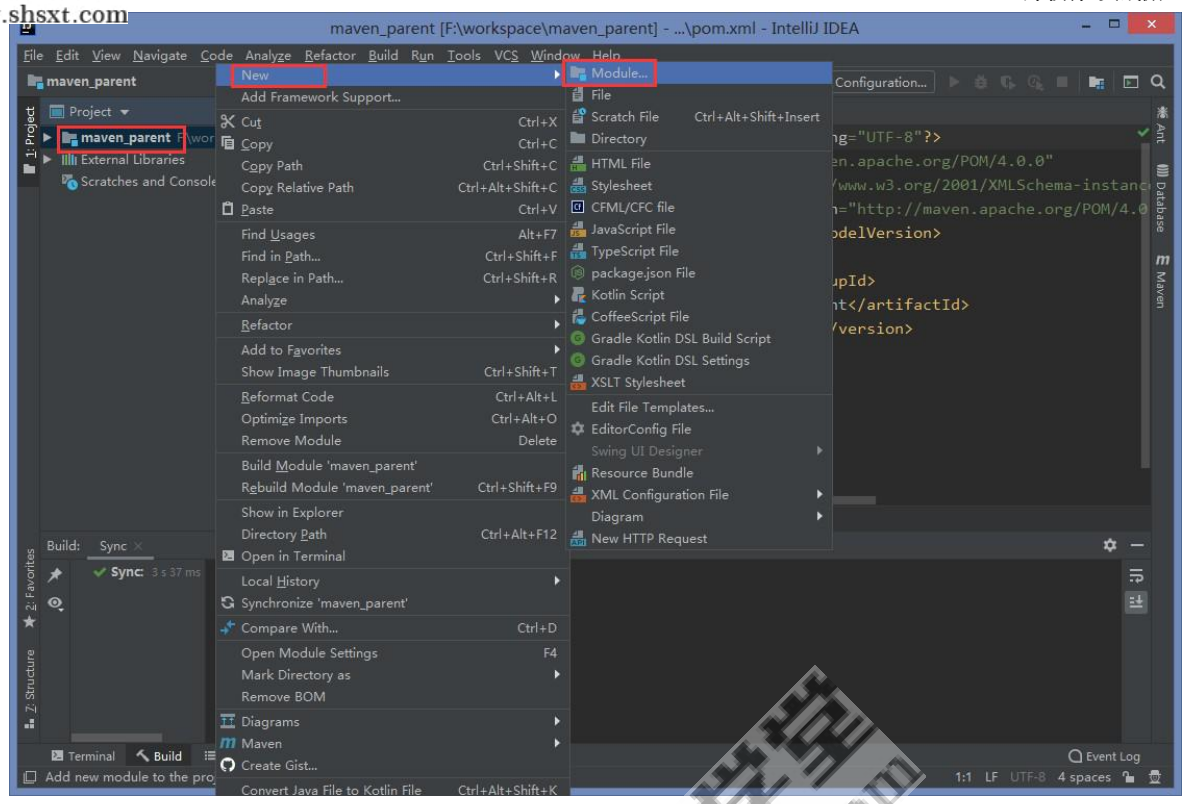


3. 设置项目名称及工作空间

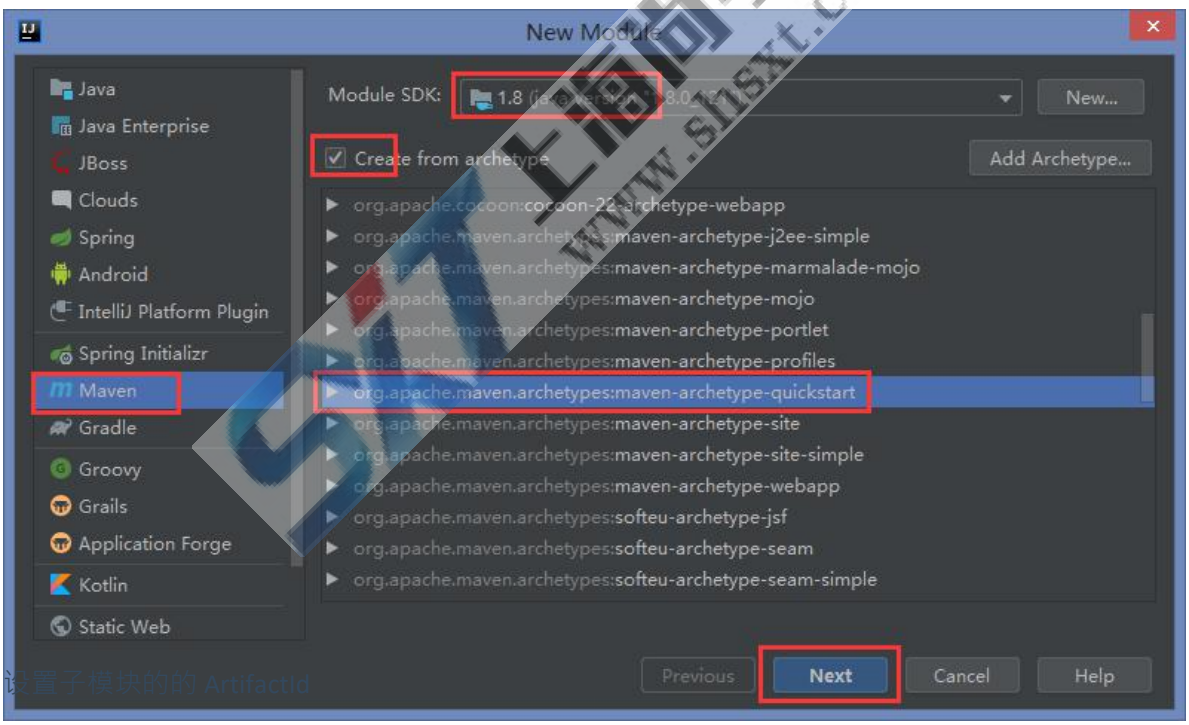


8.2. 创建 maven_dao 模块

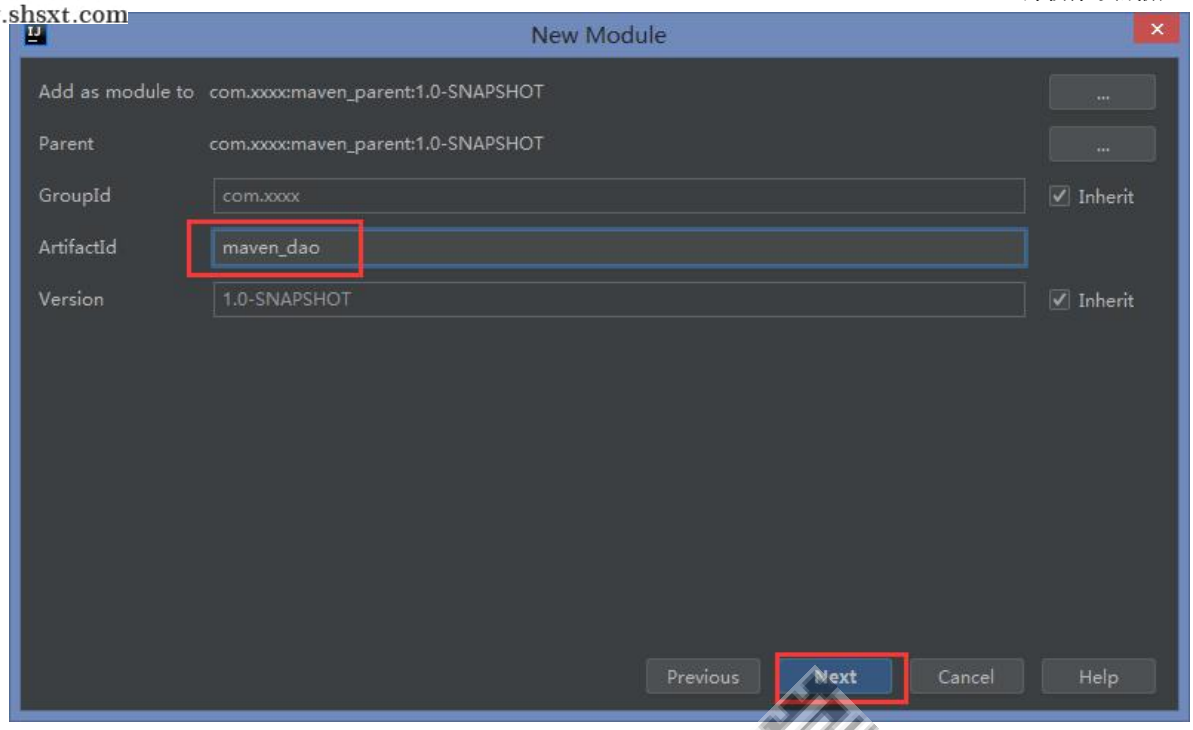
1. 选择项目maven_parent，右键选择 New ，选择Module



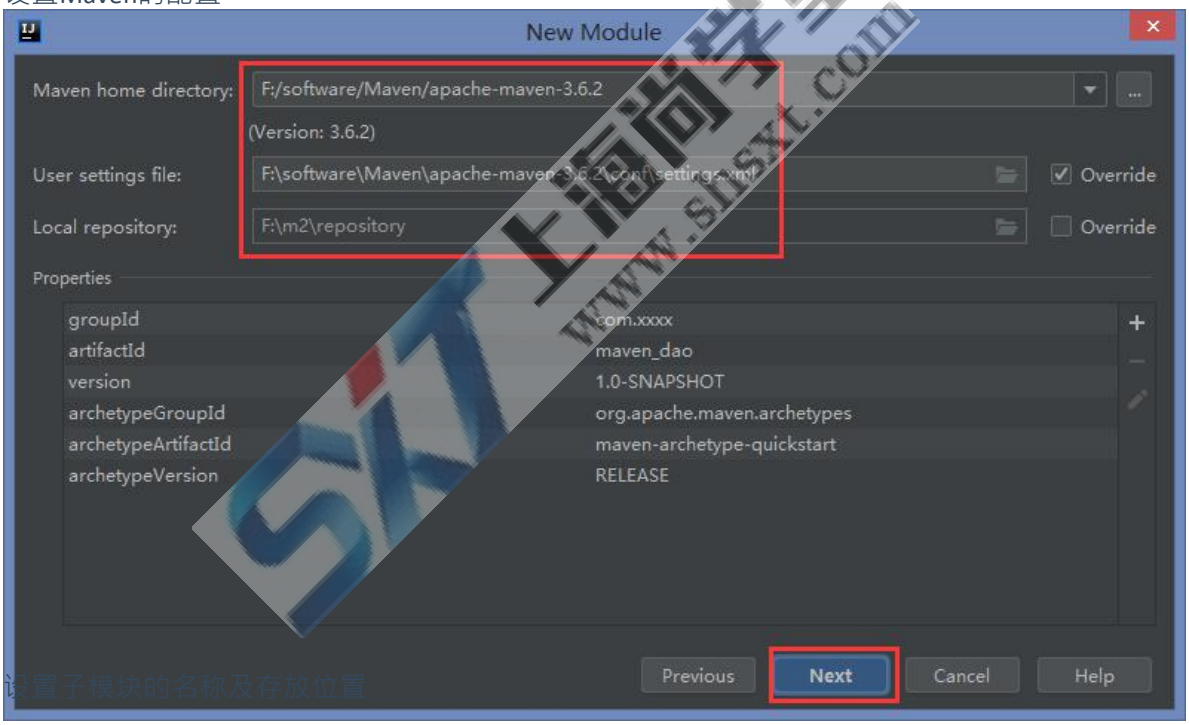
2. 选择Maven项目的模板（普通 Java 项目）



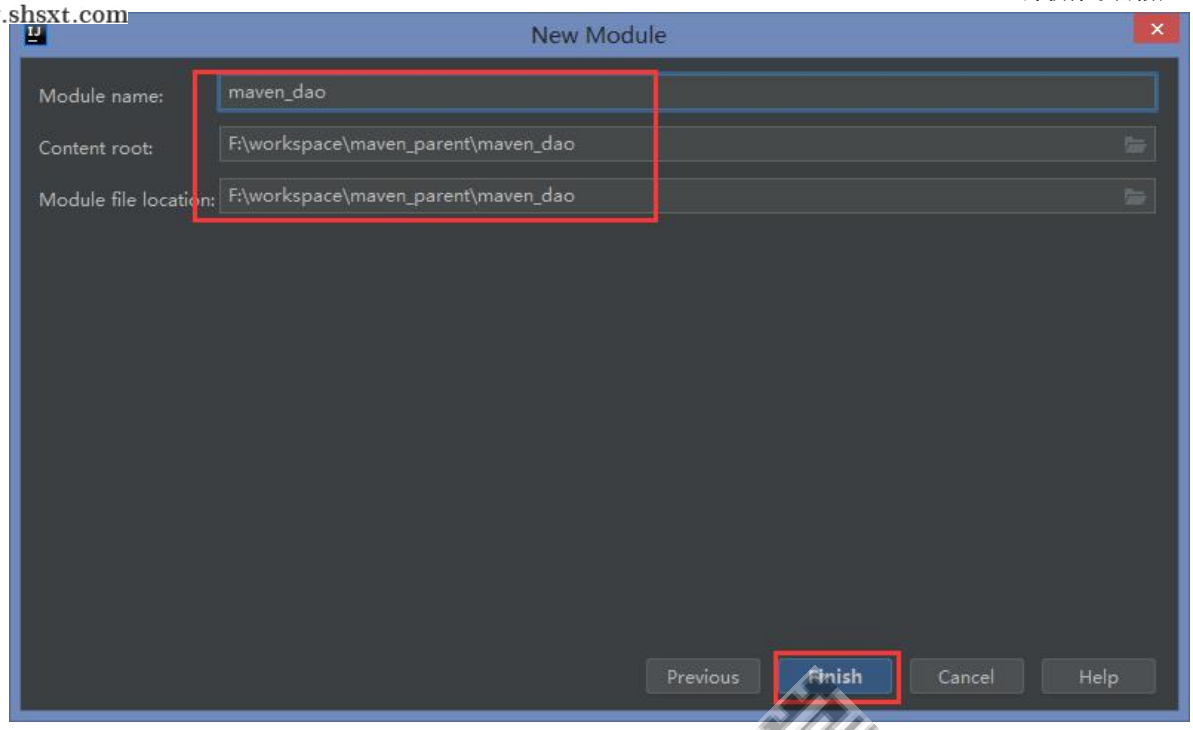
3. 设置子模块的的 ArtifactId



4. 设置Maven的配置



5. 设置子模块的名称及存放位置



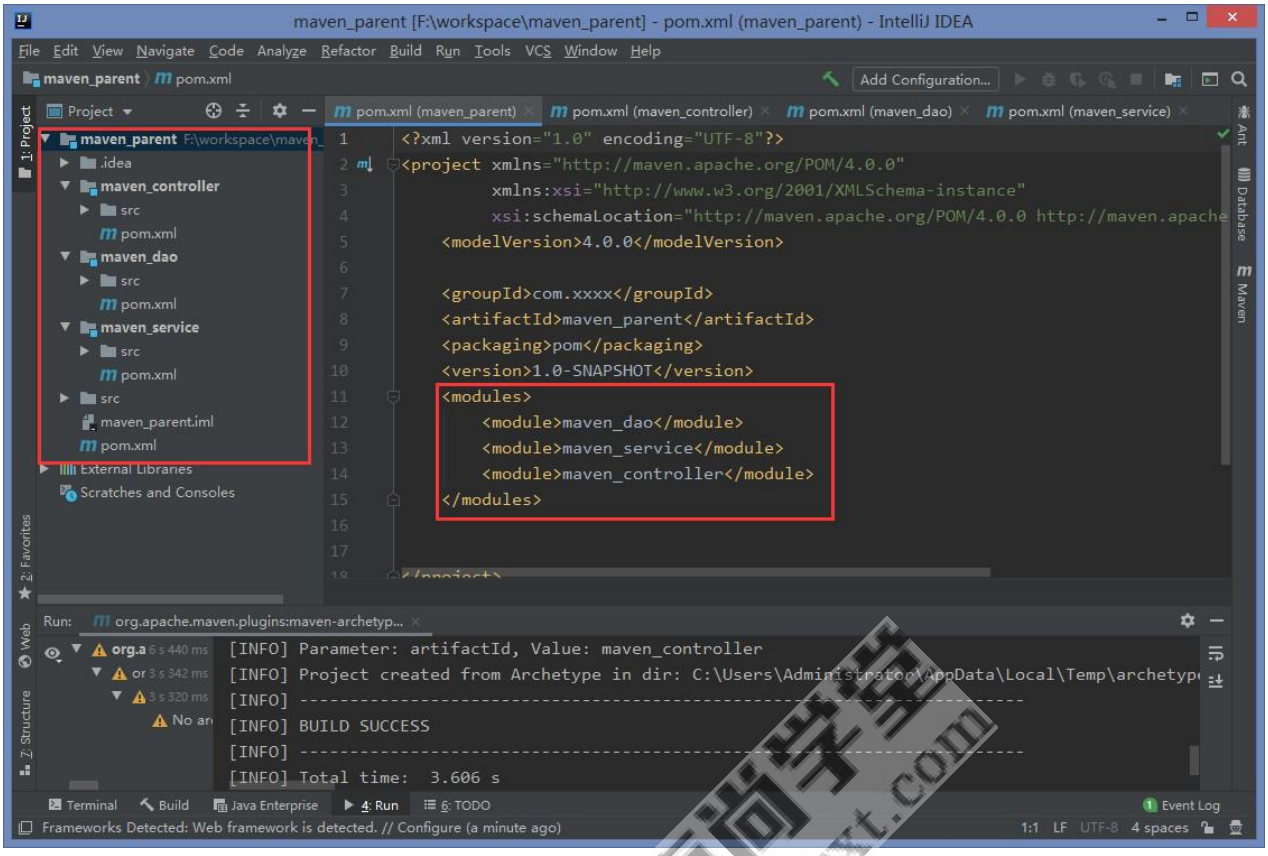
8.3. 创建 maven_service 模块

创建 maven_service 模块的步骤与 maven_dao模块一致。

8.4. 建 maven_controller 模块

创建 maven_service 模块的步骤与 maven_dao模块基本一致，只需要将第一步选择Maven模板设置为web项目即可。（模板类型：maven-archetype-webapp）

模块全部创建完毕后，效果如下：



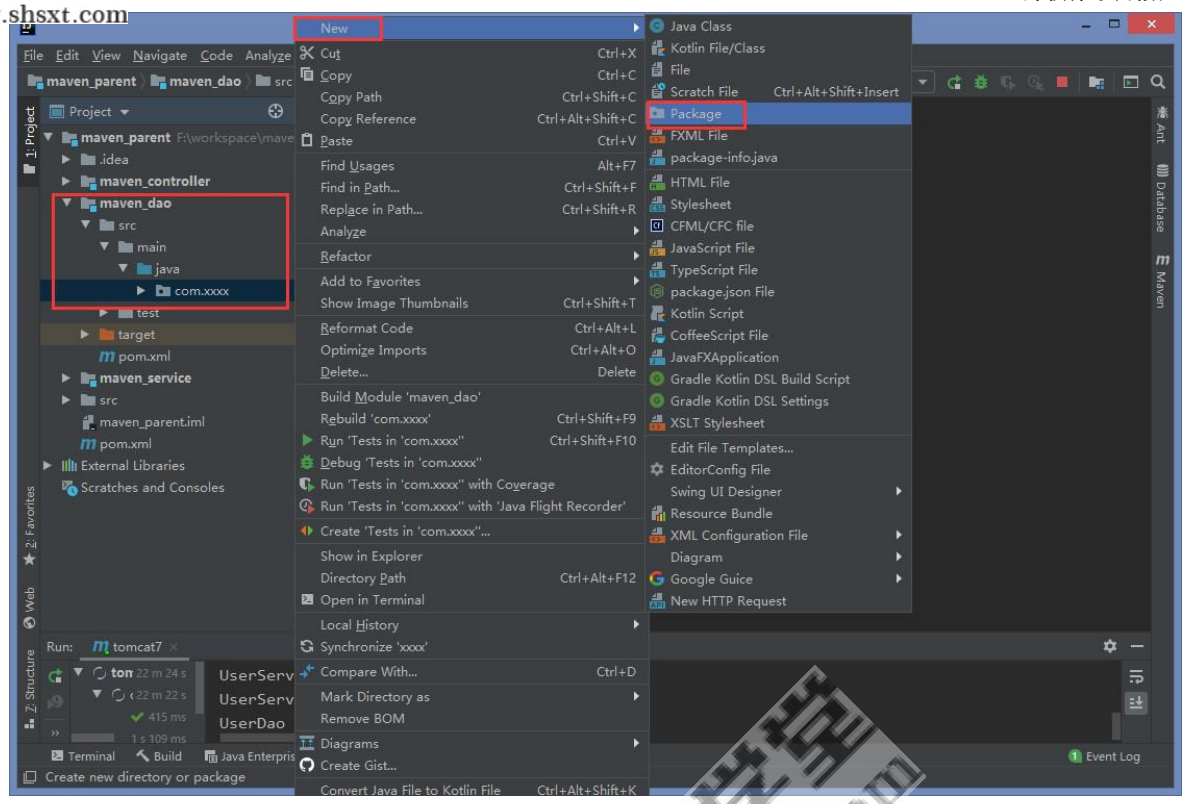
8.5. 修改模块的配置

- 设置 JDK 版本
- 单元测试 JUnit版本
- 删除多余的配置

8.6. 设置模块之间的依赖

8.6.1. maven_dao

1. 新建包



2. 在包中创建 UserDao类

3. 在类中添加方法

```
package com.xxxx.dao;

public class UserDao {

    public static void testDao()
        { System.out.println("UserDao
        Test...");
    }
```

8.6.2. maven_service

1. 添加maven_dao的依赖

```
<!-- 加入maven_dao模块的依赖 -->
<dependency>
    <groupId>com.xxxx</groupId>
    <artifactId>maven_dao</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

2. 在项目中添加UserService类，并添加方法

```
package com.xxxx.service;

import com.xxxx.dao.UserDao;

public class UserService {

    public static void testService()
        { System.out.println("UserService
        Test...");
        // 调用UserDao的方法
        UserDao.testDao();
    }
```

8.6.3. maven_controller

1. 添加 maven_service 模块的依赖

```
<!-- 加入maven_service模块的依赖 -->
<dependency>
    <groupId>com.xxxx</groupId>
    <artifactId>maven_service</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

2. 添加Servlet的依赖

```
<!-- Servlet的依赖 -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
```

3. 新建 Java 类，继承 HttpServlet 类，并重写 service方法

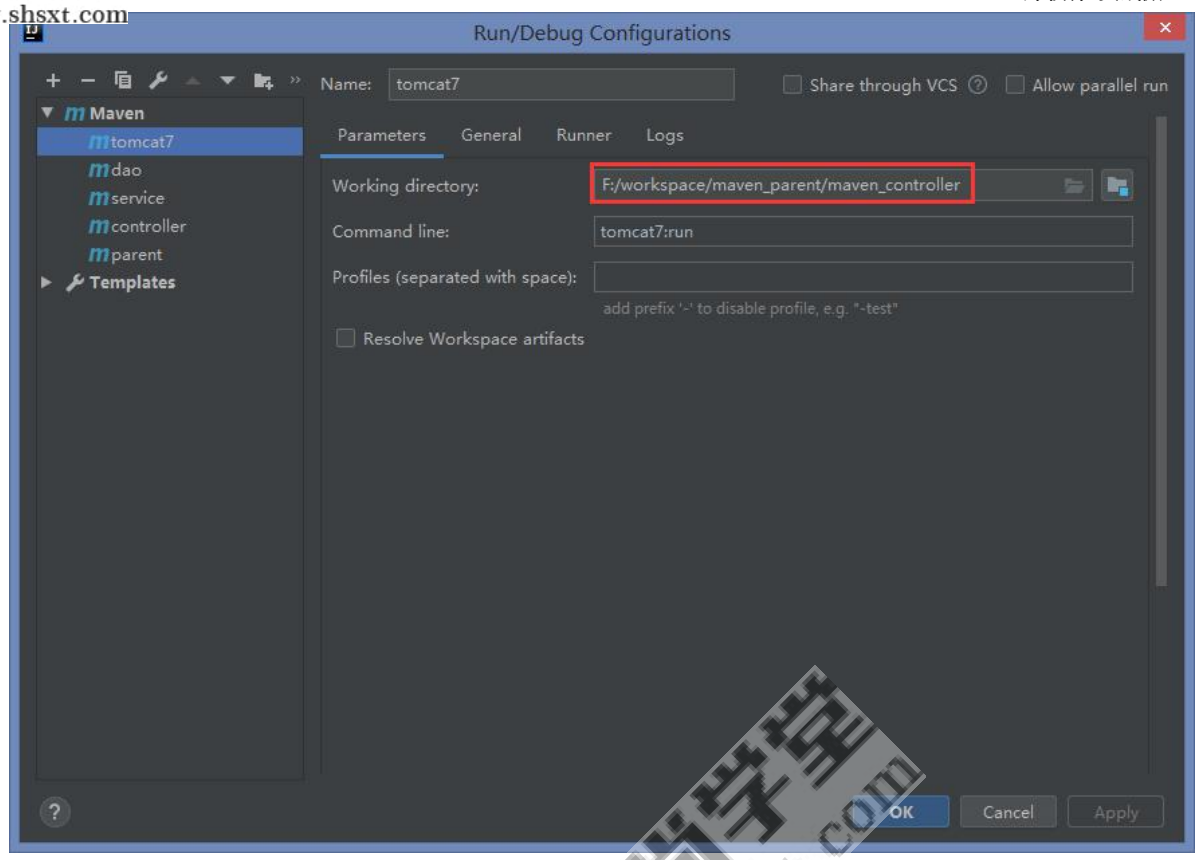
```
package com.xxxx.controller;
import com.xxxx.service.UserService;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/user")
public class UserServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("UserServlet Test...");
        // 调用UserService层的方法
        UserService.testService();
    }
}
```

4. 添加Tomcat插件

```
<!-- 添加插件 -->
<plugins>
    <!-- tomcat7插件 -->
    <plugin>
        <groupId>org.apache.tomcat.maven</groupId>
        <artifactId>tomcat7-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
            <!-- <port>8080</port> -->
            <path>/web</path>
            <uriEncoding>UTF-8</uriEncoding>
            <server>tomcat7</server>
        </configuration>
    </plugin>
</plugins>
```

5. 启动项目

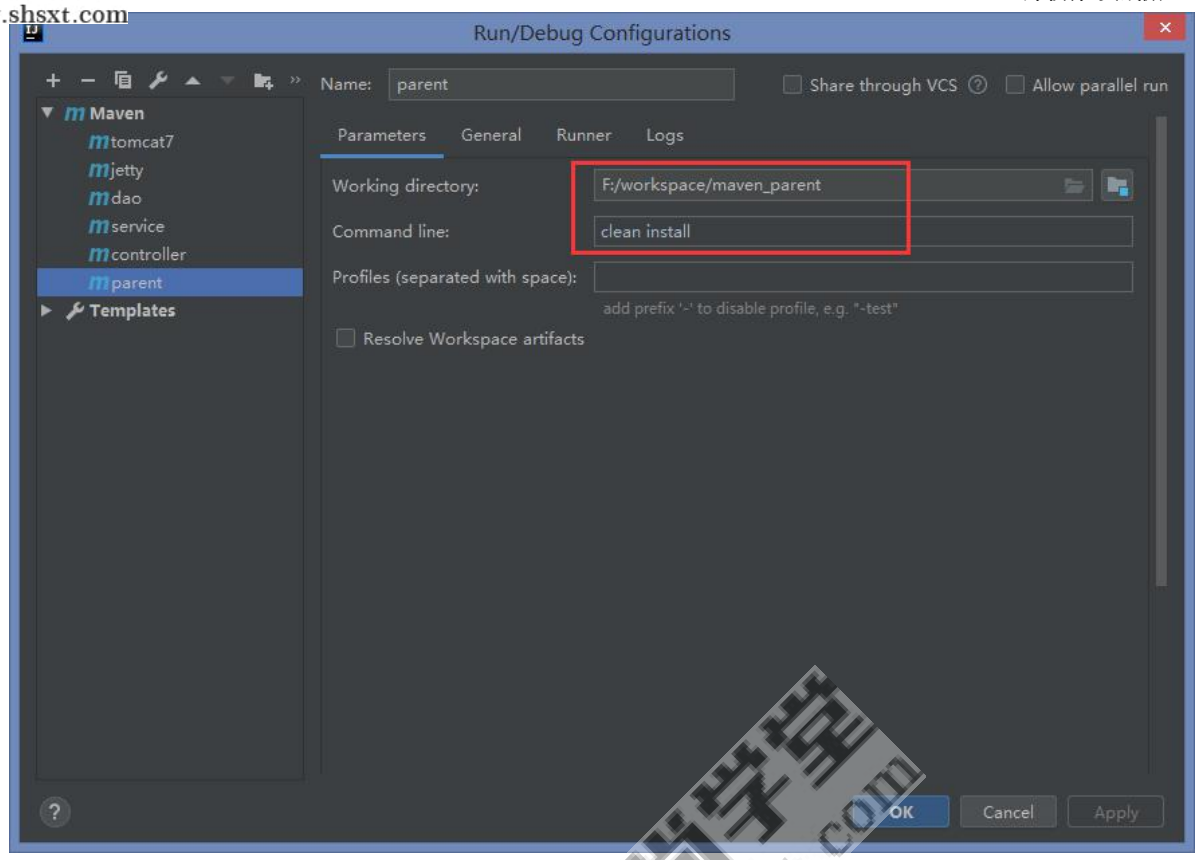


6. 访问项目

   访问地址：`http://localhost:8080/web/user`

   访问结果：

7. 如果启动失败，请先将父项目 install



注：如果父项目 install 失败，则先将所有子模块 install 成功后，再 install 父项目。

9. Maven的打包操作

对于企业级项目，无论是进行本地测试，还是测试环境测试以及最终的项目上线，都会涉及项目的打包操作，对于每个环境下项目打包时，对应的项目所有要的配置资源就会有所区别，实现打包的方式有很多种，可以通过ant,获取通过idea 自带的打包功能实现项目打包，但当项目很大并且需要的外界配置很多时，此时打包的配置就会异常复杂，对于maven 项目，我们可以用过pom.xml 配置的方式来实现打包时的环境选择，相比较其他形式打包工具，通过maven 只需要通过简单的配置，就可以轻松完成不同环境先项目的整体打包。

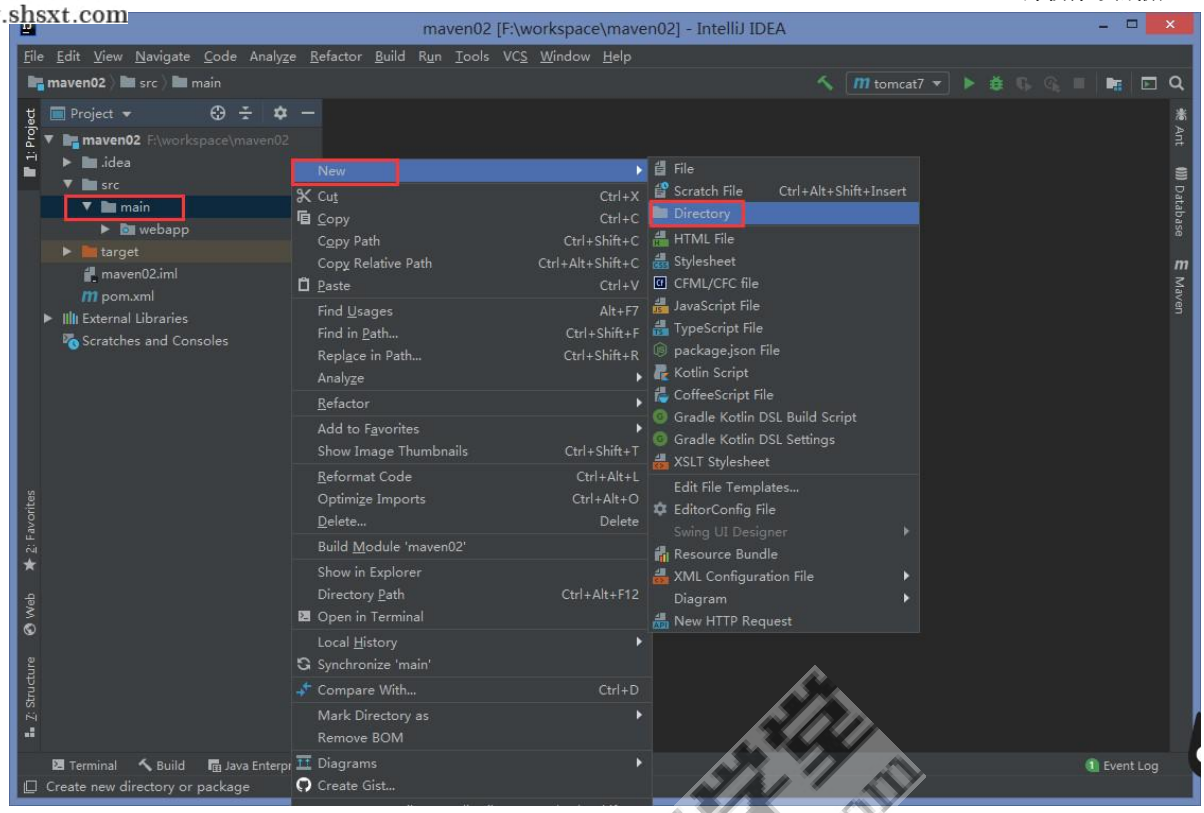
比如下面这样一个项目,项目中配置了不同环境下项目所需要的配置文件，这时候需要完成不同环境下的打包操作，此时通过修改pom.xml 如下：

9.1. 建立对应的目录结构

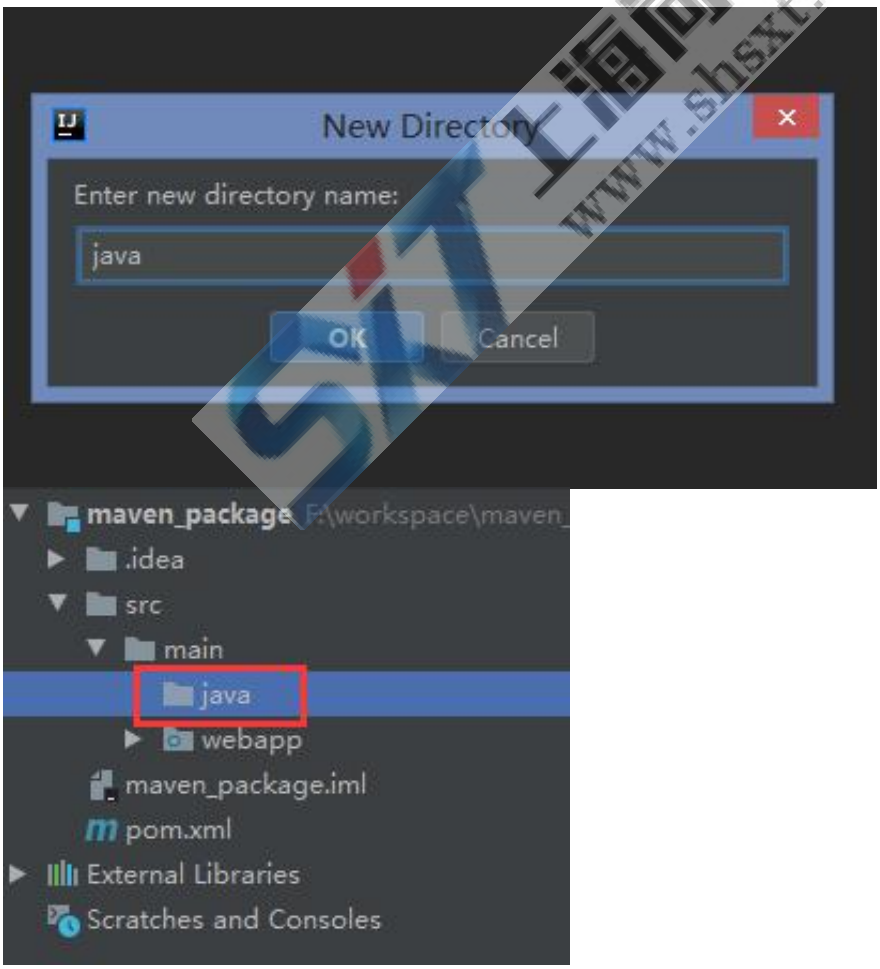
使用idea创建项目，目录结构可能会缺失，需要通过手动添加对应的目录。

1. 添加 Java 源文件夹

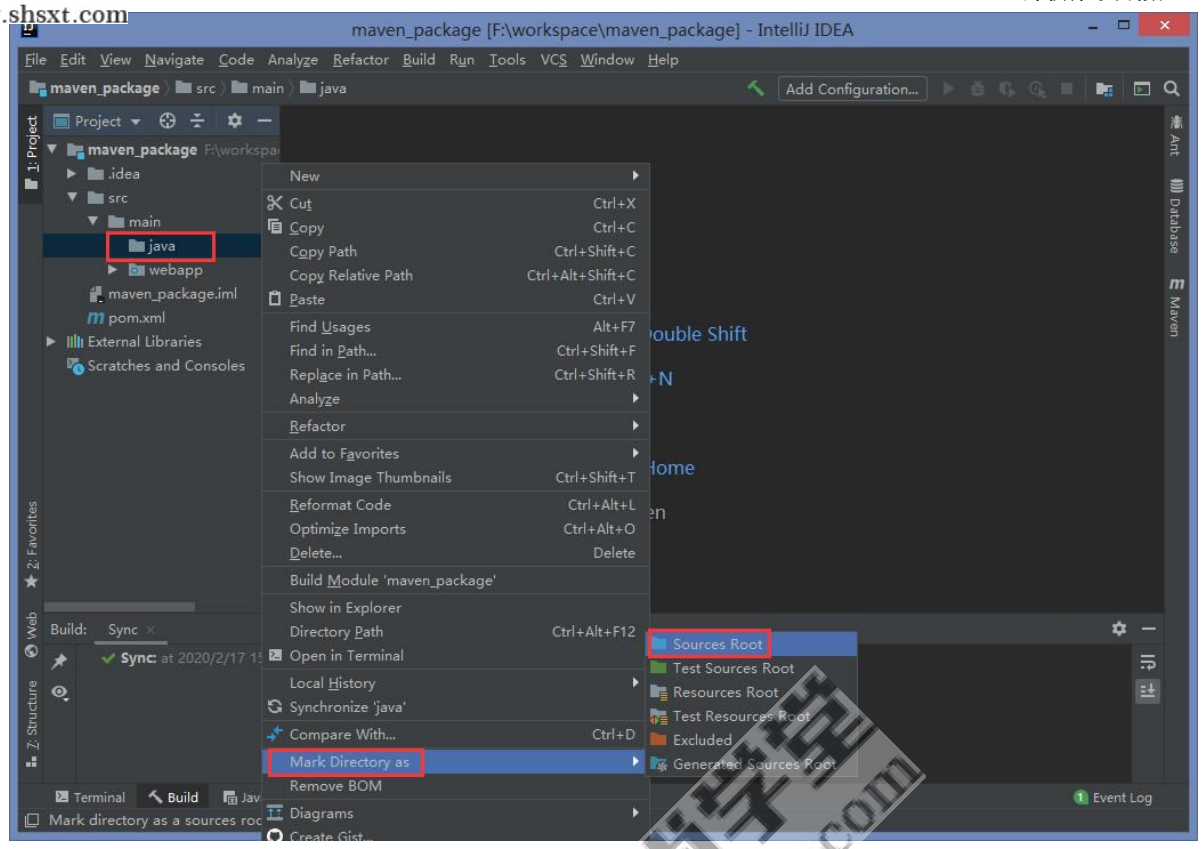
   选择项目的 main 文件夹，右键选择New，选择Directory



输入文件夹名 "Java"，如图：

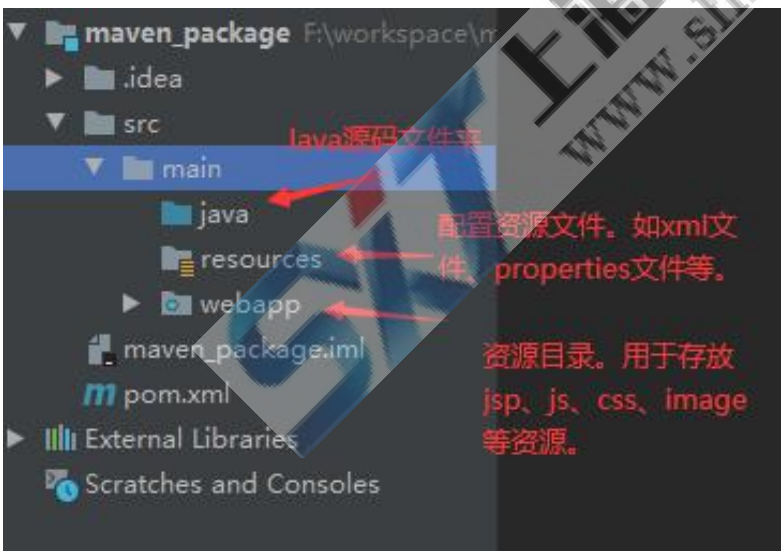


选择 java 目录，右键选择 Mark Directory as，选择 Sources Root。将文件夹标记为源文件夹。

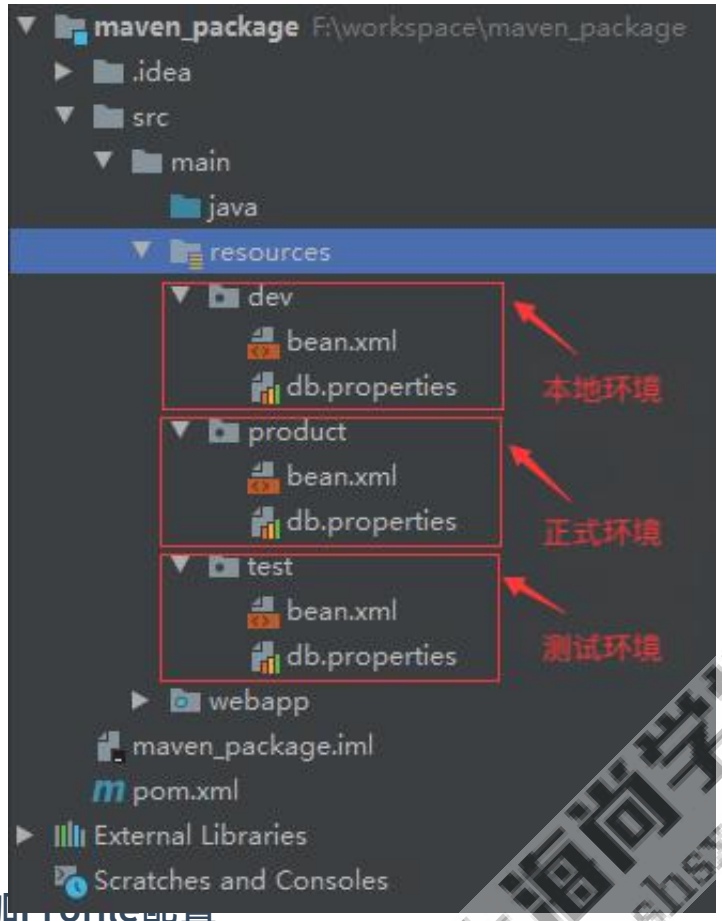


2. 添加资源文件夹

步骤如上，创建文件夹，命名为 resources，并标记为 Resources Root



3. 添加对应的文件夹目录，及添加不同环境下对应的配置文件。（本地环境、测试环境、正式环境）



9.2. 添加Profile配置

```
<!-- 打包环境配置 开发环境 测试环境 正式环境 -->
<profiles>
  <profile>
    <id>dev</id>
    <properties>
      <env>dev</env>
    </properties>
       <!-- 未指定环境时，默认打包dev环境 -->
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
  </profile>
  <profile>
    <id>test</id>
    <properties>
      <env>test</env>
    </properties>
  </profile>
  <profile>
    <id>product</id>
    <properties>
      <env>product</env>
    </properties>
  </profile>
```

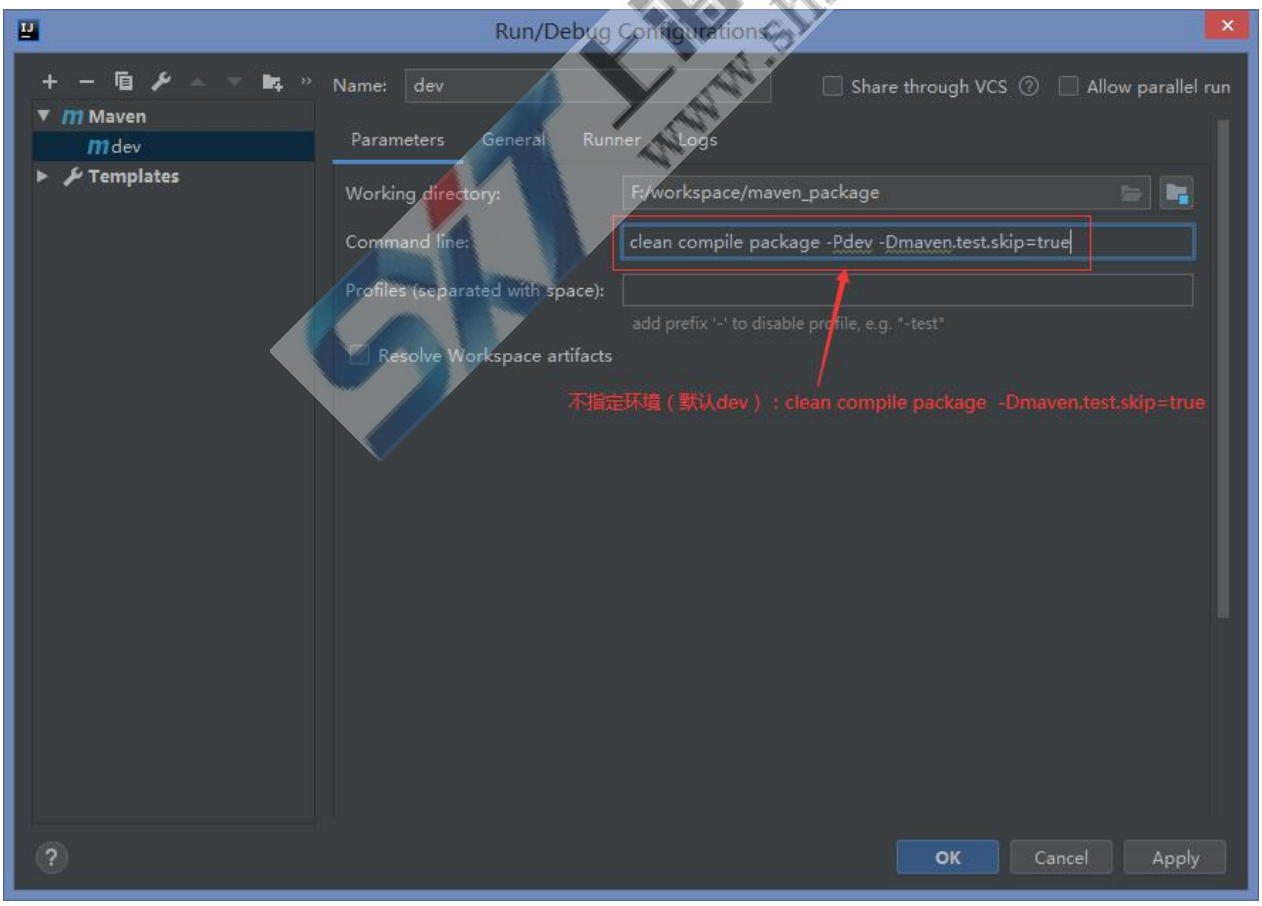
```
</profiles>
```

9.3. 设置资源文件配置

```
<!-- 对于项目资源文件的配置放在build中 -->
<resources>
  <resource>
    <directory>src/main/resources/${env}</directory>
  </resource>
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>**/*.xml</include>
      <include>**/*.properties</include>
      <include>**/*.tld</include>
    </includes>
    <filtering>false</filtering>
  </resource>
</resources>
```

9.4. 执行打包操作

打开Run/Debug Configuarations窗口，输入对应的打包命令



此时对应打包命令

1. `clean compile package -Dmaven.test.skip=true`

打包默认环境（开发环境）并且跳过maven 测试操作

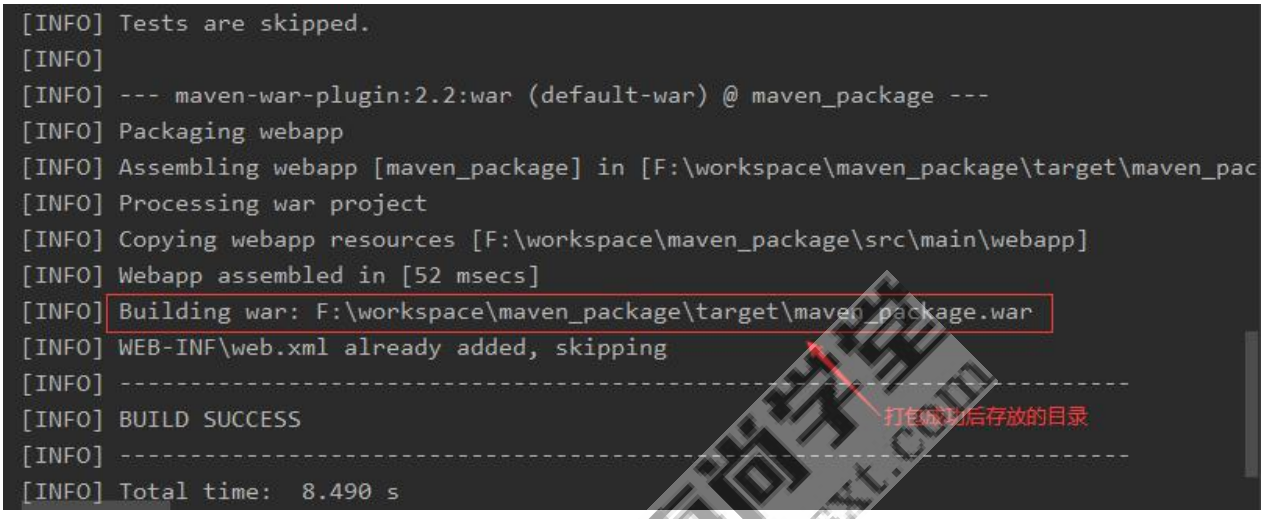
2. `clean compile package -Ptest -Dmaven.test.skip=true`

打包测试环境并且跳过maven 测试操作

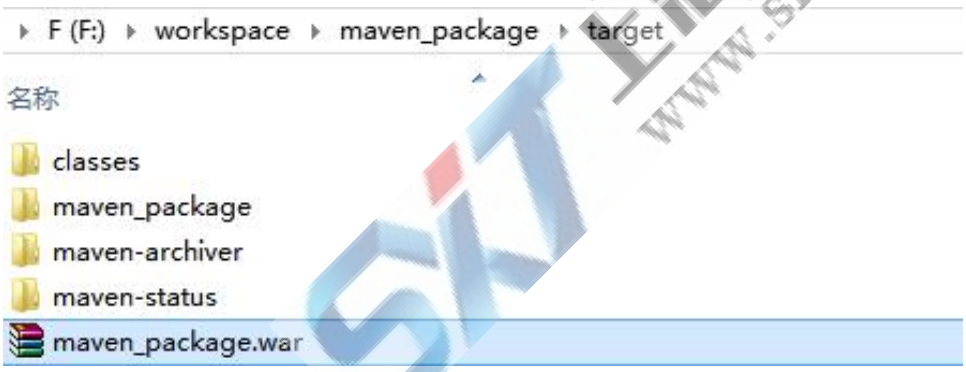
3. `clean compile package -Pproduct -Dmaven.test.skip=true`

打包生产环境并且跳过maven 测试操作

打包成功



不同的项目打包的文件不一样，一般来说，普通java项目打成jar包，web项目打成war包



10. Maven依赖的基本概念

10.1. 依赖的基本配置

根元素project下的dependencies可以包含多个 dependence元素，以声明多个依赖。每个依赖都应该包含以下元素：

1. groupId, artifactId, version：依赖的基本坐标， 对于任何一个依赖来说，基本坐标是最重要的，Maven根据坐标才能找到需要的依赖。
2. Type： 依赖的类型，大部分情况下不需要声明。 默认值为jar
3. Scope： 依赖范围（compile,test,provided,runtime,system）

- compile: 编译依赖范围。

  如果没有指定，就会默认使用该依赖范围。使用此依赖范围的Maven依赖，对于编译、测试、运行三种classpath都有效。

- test: 测试依赖范围。

  使用此依赖范围的Maven依赖，只对于测试classpath有效，在编译主代码或者运行项目的使用时将无法使用此类依赖。典型的例子就是JUnit，它只有在编译测试代码及运行测试的时候才需要。

- provided: 已提供依赖范围。

  使用此依赖范围的Maven依赖，对于编译和测试classpath有效，但在运行时无效。典型的例子是servlet-api，编译和测试项目的时候需要该依赖，但在运行项目的时候，由于容器已经提供，就不需要Maven重复地引入一遍(如：servlet-api)。

- runtime: 运行时依赖范围。

  使用此依赖范围的Maven依赖，对于测试和运行classpath有效，但在编译主代码时无效。典型的例子是JDBC驱动实现，项目主代码的编译只需要JDK提供的JDBC接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体JDBC驱动。

- system: 系统依赖范围。

  该依赖与三种classpath的关系，和provided依赖范围完全一致。但是，使用system范围依赖时必须通过systemPath元素显式地指定依赖文件的路径。由于此类依赖不是通过Maven仓库解析的，而且往往与本机系统绑定，可能造成构建的不可移植，因此应该谨慎使用。

4. Optional：标记依赖是否可选
5. Exclusions： 用来排除传递性依赖。

10.2. 依赖范围

首先需要知道，Maven在编译项目主代码的时候需要使用一套classpath。 比如：编译项目代码的时候需要用到spring-core, 该文件以依赖的方式被引入到classpath中。 其次， Maven在执行测试的时候会使用另外一套classpath。 如：junit。

最后在实际运行项目时，又会使用一套classpath， spring-core需要在该classpath中，而junit不需要。

那么依赖范围就是用来控制依赖与这三种classpath(编译classpath，测试classpath，运行时classpath)的关系， Maven有以下几种依赖范围：

- Compile 编译依赖范围。 如果没有指定，就会默认使用该依赖范围。 使用此依赖范围的Maven依赖， 对于编译，测试，运行都有效。
- Test： 测试依赖范围。 只在测试的时候需要。比如junit
- Provided： 已提供依赖范围。 使用此依赖范围的Maven依赖，对于编译和测试有效， 但在运行时无效。 典型的例子是servlet-API, 编译和测试项目的需要， 但在运行项目时， 由于容器已经提供，就不需要Maven重复地引入一遍。
- Runtime： 运行时依赖范围。 使用此依赖范围的Maven依赖，对于测试和运行有效， 但在编译代码时无效。 典型的例子是：jdbc驱动程序， 项目主代码的编译只需要jdk提供的jdbc接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体jdbc驱动。

- System： 系统依赖范围。 一般不使用。

10.3. 传递性依赖

传递依赖机制， 让我们在使用某个jar的时候就不用去考虑它依赖了什么。也不用担心引入多余的依赖。 Maven会解析各个直接依赖的POM，将那些必要的间接依赖，以传递性依赖的形式引入到当前项目中。

注意： 传递依赖有可能产生冲突！！

冲突场景：

```
A-->B--->C (2.0)

A-->E--->C (1.0)
```

如果A下同时存在两个不同version的C，冲突！！（选取同时适合A、B的版本）

```
<dependencies>
    <dependency>
        <groupId>A</groupId>
        <artifactId>A</artifactId>
        <version>xxx</version>
        <exclusions>
            <exclusion>
                <groupId>C</groupId>
                <artifactId>C</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

    <dependency>
        <groupId>B</groupId>
        <artifactId>B</artifactId>
    </dependency>
</dependencies>
```