

# 1.项目中的jar包如何管理?

---

## 项目开发中遇到的问题

- 1、都是同样的代码，为什么在我的机器上可以编译执行，而其他的机器上就不行？
- 2、为什么在我的机器上可以正常打包，而配置管理员却打包不成功？
- 3、项目组加入了新的人员，我要给他说明编译环境如何设置，但是让我挠头的是，有些细节我也记不清楚了。
- 4、我的项目依赖一些jar包，我应该把他们放哪里？放源码库里？
- 5、这是我开发的第二个项目，还是需要上面的那些jar包，再把它们复制到我当前项目的svn库里，那第三个、第四个项目呢？
- 6、我写了一个数据库相关的通用类，并且推荐给了其他项目组，现在已经有五个项目组在使用它了，今天我发现了一个bug，并修正了它，我要逐个通知各个小组去修改吗？

# 2.Maven模型介绍及原理分析

---

## 2.1 Maven介绍

---

Maven(麦文)项目对象模型(POM)，可以通过一小段描述信息来管理项目的构建，报告和文档的项目管理工具软件。

Maven 除了以程序构建能力为特色之外，还提供高级项目管理工具。由于 Maven 的缺省构建规则有较高的可重用性，所以常常用两三行 Maven 构建脚本就可以构建简单的项目。由于 Maven 的面向项目的方法，许多 Apache Jakarta 项目发文时使用 Maven，而且公司项目采用 Maven 的比例在持续增长。

Maven这个单词来自于意第绪语（犹太语），意为知识的积累，最初在Jakarta Turbine项目中用来简化构建过程。当时有一些项目（有各自Ant build文件），仅有细微的差别，而JAR文件都由[CVS来维护。于是希望有一种标准化的方式构建项目，一个清晰的方式定义项目的组成，一个容易的方式发布项目的信息，以及一种简单的方式在多个项目中共享JARs。

Maven官方地址:<http://maven.apache.org/>

## 2.2 Maven主要有两个功能

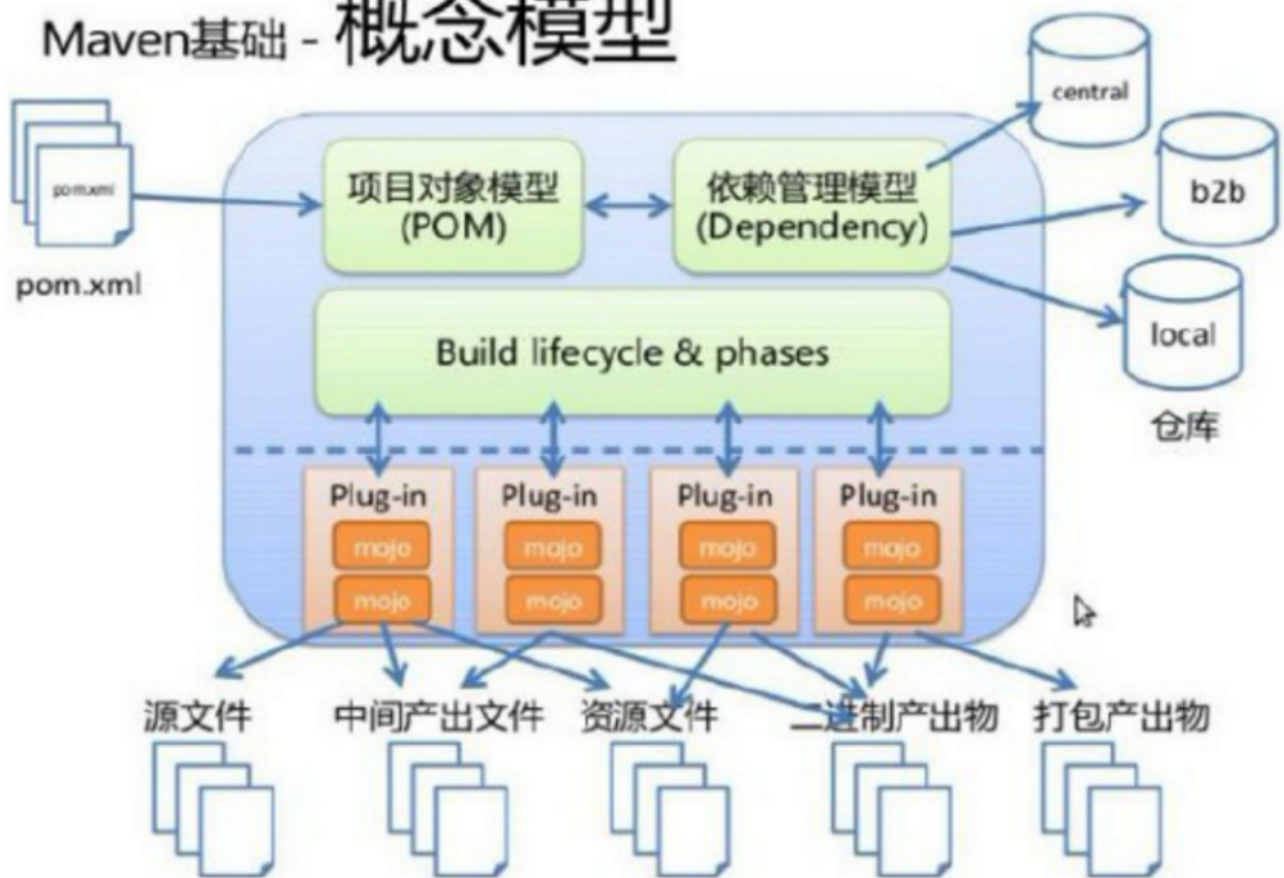
---

- 1、依赖管理-jar包管理
- 2、项目构建-代码编译

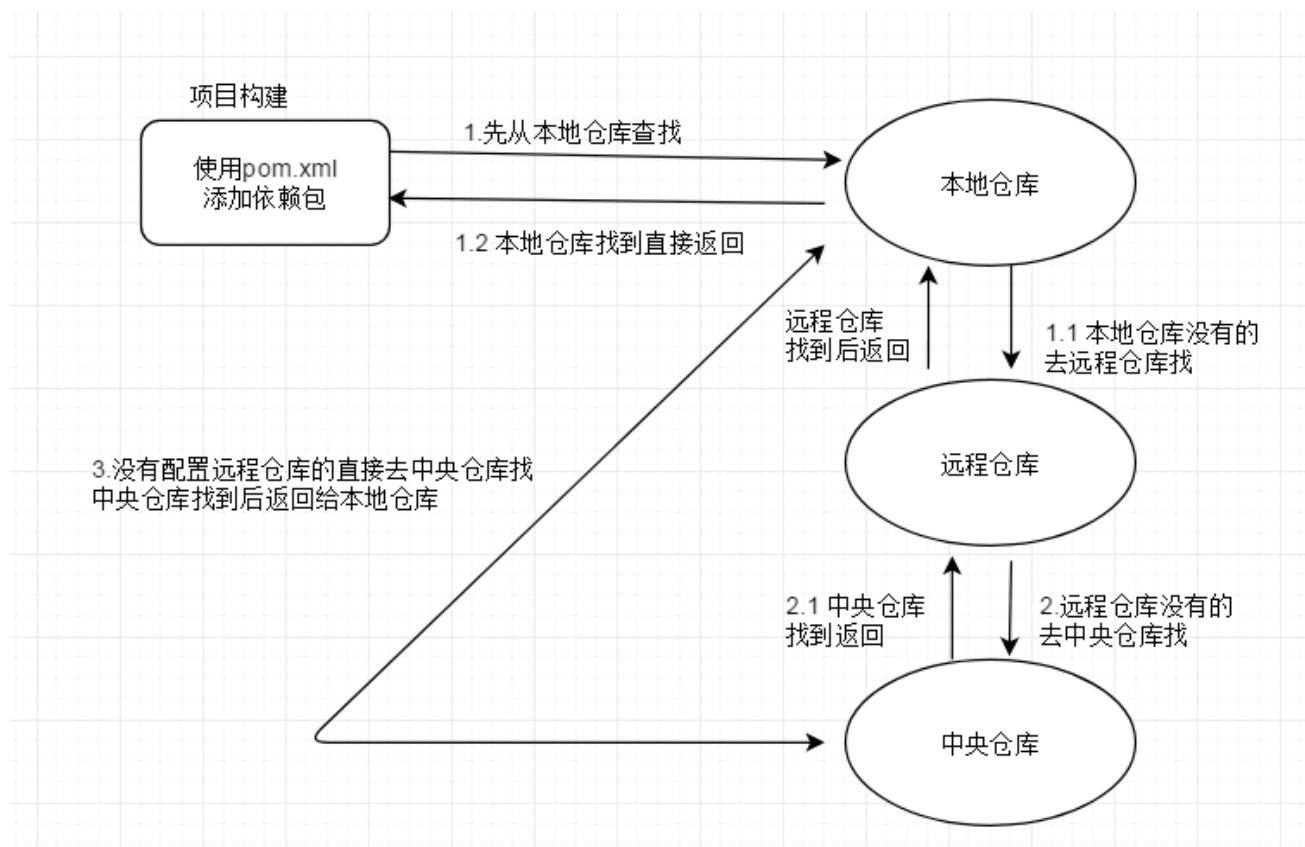
## 2.3 Maven模型介绍

---

## Maven基础 - 概念模型

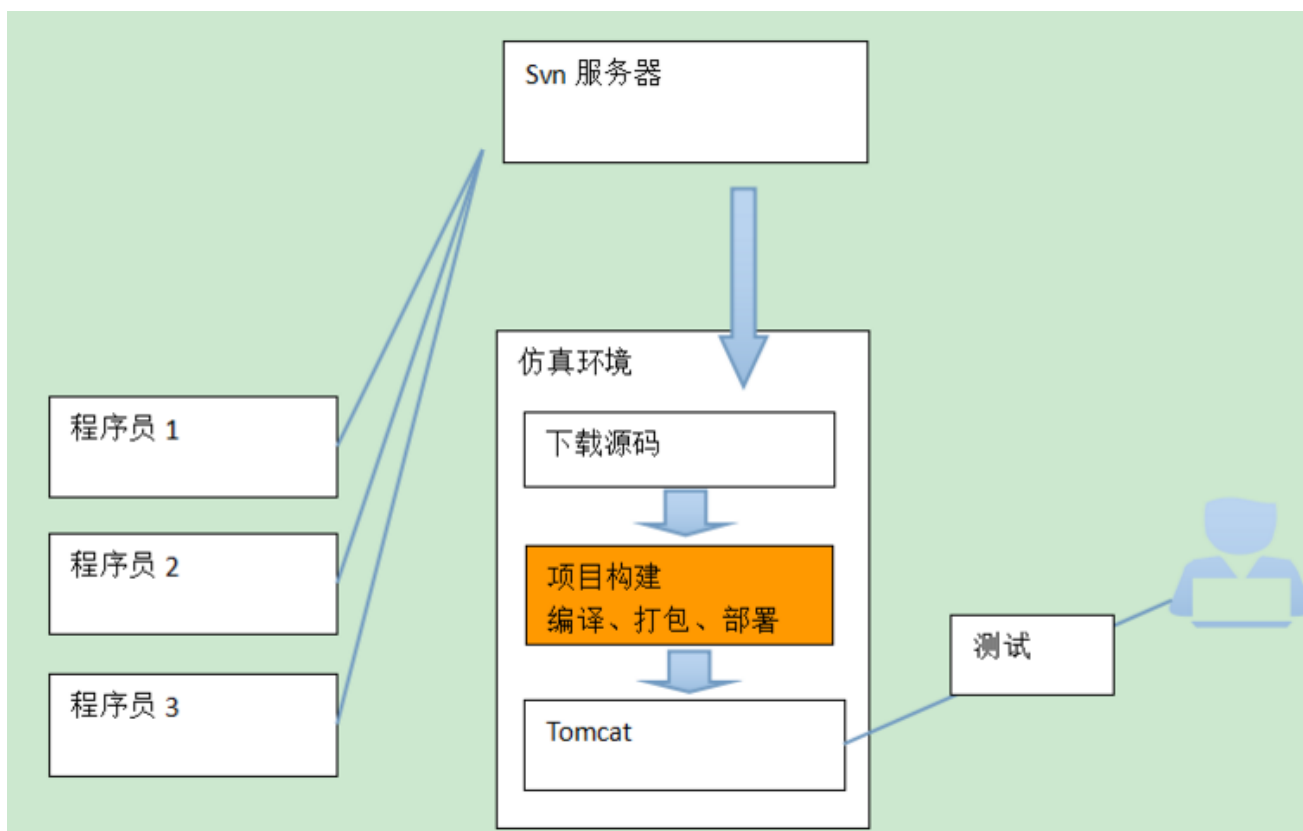


### 2.4 Maven依赖管理原理分析

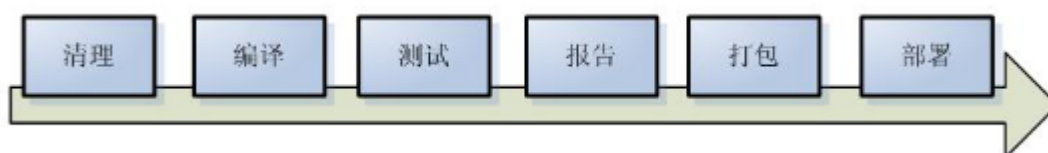


## 3.项目构建与Maven的关系

### 3.1 构建的概念



## 构建过程



## 项目构建方式

### 1、Ant

Ant只是一个项目构建工具，它没有集成依赖管理。

Ant在进行项目构建时，它没有对项目目录结构进行约定，需要手动指定源文件、类文件等目录地址。同时它执行task时，需要显示指定依赖的task，这样会造成大量的代码重复。

### 2、Maven

Maven不仅是一个项目构建工具，更是一个项目管理工具。它在项目构建工程中，比ant更全面，更灵活。Maven在进行项目构建时，它对项目目录结构拥有约定，知道你的源代码在哪里，类文件应该放到哪里去。它拥有生命周期的概念，maven的生命周期是有顺序的，在执行后面的生命周期的任务时，不需要显示的配置前面任务的生命周期。例如执行 `mvn install` 就可以自动执行编译，测试，打包等构建过程

### 3、Gradle

一个开源的自动化构建系统，建立在Apache Ant和Maven Apache概念的基础上，并引入了基于Groovy的特定领域语言（DSL），而不是使用Apache Maven宣布的项目配置XML形式。

## 4.Maven的安装与配置

Maven下载: <http://maven.apache.org/download.cgi>

Maven使用特点:约定大于配置

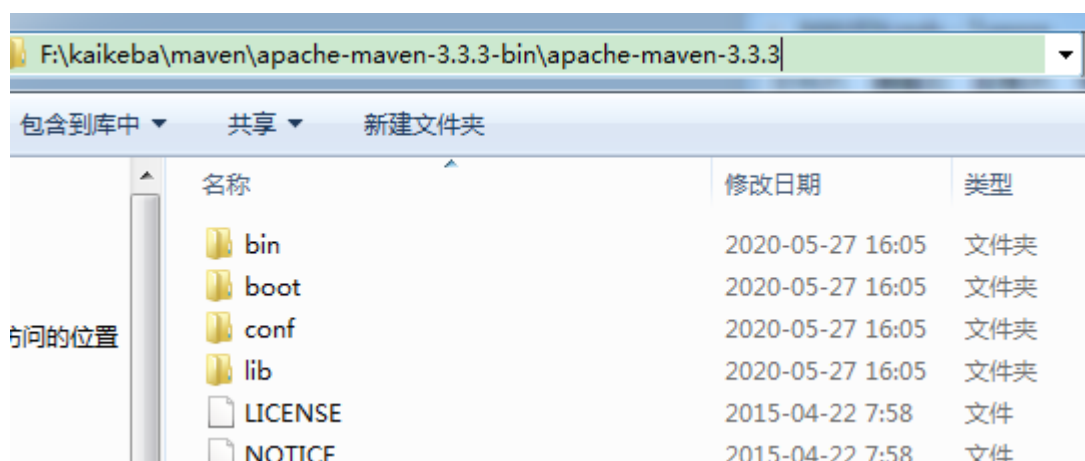
Maven中央仓库地址: <http://mvnrepository.com/>

配置步骤:

1.确定jdk 已经安装和配置(要求jdk版本最低1.8)

```
C:\Users\Administrator>java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

2.把maven3.3.3解压到放到任意磁盘以及任意文件夹, 注意:解压路径不要有中文



这里我使用的idea版本



兼容maven3.3.3。如果是其他版本的同学可以替换其他版本的maven,但是无论哪个版本一定要注意兼容性,避免浪费时间在兼容性上。可以提前上网查询你自己版本的idea对应哪个版本的maven.

3.配置maven环境变量:

以window7系统为例:"计算机"右键->"属性"->"高级系统设置"->"高级"->"环境变量"->"系统变量"->"新建"

变量名:M2\_HOME (固定值)

变量值:F:\kaikeba\maven\apache-maven-3.3.3-bin\apache-maven-3.3.3 (maven中bin文件夹的全路径)

4.配置系统变量中的path变量值,在变量值的最前面添加下面代码:%M2\_HOME%\bin

5.测试:

打开cmd运行命令: mvn -version

```
C:\Users\Administrator>mvn -version
Apache Maven 3.3.3 (7994120775791599e205a5524ec3e0dfe41d4a06; 2015-04-2
Maven home: F:\1000phone\BJ-J1903\maven\apache-maven-3.3.3
Java version: 1.8.0_131, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_131\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"
```

## 5.Maven核心文件Pom.xml

Pom.xml文件说明:

```
<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">
    <!-- 版本: 4.0.0 -->
    <modelVersion>4.0.0</modelVersion>
    <!-- 组织名称: 暂时使用 组织名称+项目名称 作为组织名称 -->
    <!-- 组织名称: 实际名称 按照访问路径规范设置, 通常以功能作为名称:
        eg: junit spring -->
    <groupId>com.kkb.maven</groupId>
    <!-- 项目名称 -->
    <artifactId>HelloWorld</artifactId>
    <!-- 当前项目版本号: 同一个项目开发过程中可以发布多个版本, 此处标示0.0.1版 -->
    <!-- 当前项目版本号: 每个工程发布后可以发布多个版本, 依赖时调取不同的版本, 使用不同的版本号 -->
    <version>0.0.1</version>
    <!-- 名称: 可省略 -->
    <name>Hello</name>

    <!-- 依赖关系 -->
    <dependencies>
        <!-- 依赖设置 -->
        <dependency>
            <!-- 依赖组织名称 -->
            <groupId>junit</groupId>
            <!-- 依赖项目名称 -->
            <artifactId>junit</artifactId>
            <!-- 依赖版本名称 -->
            <version>4.12</version>
            <!-- 依赖范围: test包下依赖该设置 -->
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

## 6.搭建原生Maven项目

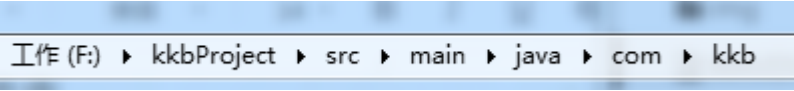
## 6.1 通过文件夹的方式创建java项目

在某个磁盘的根目录按照下面结构创建文件夹

ProjectName

- | -src
- | -main
- | -java — 存放项目的.java文件
- | -resources — 存放项目资源文件
- | -test
- | -java — 存放所有测试.java文件, 如JUnit测试类
- | -target — 目标文件输出位置例如.class、.jar、.war文件 (不需要创建, 会自动生成)
- | -pom.xml — maven项目核心配置文件

## 6.2 在src/main/java下创建包目录, 用来存放java代码



工作 (F:) > kkbProject > src > main > java > com > kkb

注意:包目录是:com.kkb,而不包含前面的路径

## 6.3 新建java文件,编写java代码,这里输出的内容最好是英文, 中文会出乱码

```
package com.kkb; //注意这里的包路径
public class Demo{
    public void test1(){
        System.out.println("maven test success");
    }
}
```

## 6.4 在test/java文件夹下创建包结构以及测试类

示例的包路径是:com.testkkb,测试类文件名:TestDemo.java,这里输出的内容最好是英文, 中文会出乱码

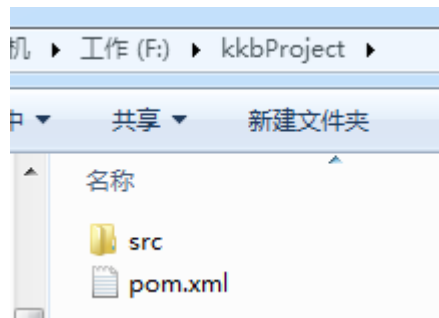
```

package com.testkkb;
import com.kkb.Demo;//这里导入Demo类所在包，注意包路径
import org.junit.Test;//这里导入junit测试包，注意我们还没有给项目中添加依赖包

public class TestDemo{
    @Test
    public void testa(){
        System.out.println("I want get Demo Class");
        new Demo().test1();
    }
}

```

6.5 在src同级文件夹下存放pom.xml文件，这里开始引入Junit工具包



```

<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.kkb.maven</groupId>
    <artifactId>HelloWorld</artifactId>
    <version>0.0.1</version>

    <!-- 依赖关系 -->
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>

```

6.6 通过maven命令行的方式运行项目

打开cmd后，进入到项目文件夹，运行mvn test命令进行junit测试



```
C:\Users\Administrator>f:
F:\>cd kkbProject
F:\kkbProject>mvn test
```

命令运行完后，程序会自动下载所需的依赖包，注意这里不仅仅是junit测试包，还有maven环境的依赖包，所以需要耐心等待一段时间。这里依赖包会下载到本地磁盘，后续再使用的时候就不会重新下载了。

6.7 运行结果:会在src同级目录下生成target文件夹，在cmd命令界面可以看到运行结果，说明maven项目结构运行成功。

```
-----
T E S T S
-----
Running com.testkkb.TestDemo
I want get Demo Class
maven test success
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] BUILD SUCCESS
-----
```

测试结果显示

说明项目构建成功

## 7.Maven常用指令

前提说明:需要在pom.xml所在目录中执行以下命令。

### 7.1 Mvn compile

执行 mvn compile命令，完成编译操作

执行完毕后，会生成target目录，该目录中存放了编译后的字节码文件。

### 7.2 Mvn clean

执行 mvn clean命令,执行完毕后，会将target目录删除。

### 7.3 Mvn test

执行 mvn test命令，完成单元测试操作。执行完毕后，会在target目录中生成三个文件夹：surefire、surefire-reports（测试报告）、test-classes（测试的字节码文件）

### 7.4 Maven package

执行 mvn package命令，完成打包操作

执行完毕后，会在target目录中生成一个文件，该文件可能是jar、war

### 7.5 Mvn install

执行 mvn install命令，完成将打好的jar包安装到本地仓库的操作

执行完毕后，会在本地仓库中出现安装后的jar包，方便其他工程引用

#### 7.6 mvn clean compile 命令

cmd 中录入 mvn clean compile 命令

组合指令，先执行clean，再执行compile，通常应用于上线前执行，清除测试类

#### 7.7 mvn clean test 命令

cmd 中录入 mvn clean test 命令

组合指令，先执行clean，再执行test，通常应用于测试环节

#### 7.8 mvn clean package 命令

cmd 中录入 mvn clean package命令

组合指令，先执行clean，再执行package，将项目打包，通常应用于发布前

执行过程：

清理—————清空环境

编译—————编译源码

测试—————测试源码

打包—————将编译的非测试类打包

#### 7.9 mvn clean install 命令

cmd 中录入 mvn clean install 查看仓库，当前项目被发布到仓库中

组合指令，先执行clean，再执行install，将项目打包，通常应用于发布前

执行过程：

清理—————清空环境

编译—————编译源码

测试—————测试源码

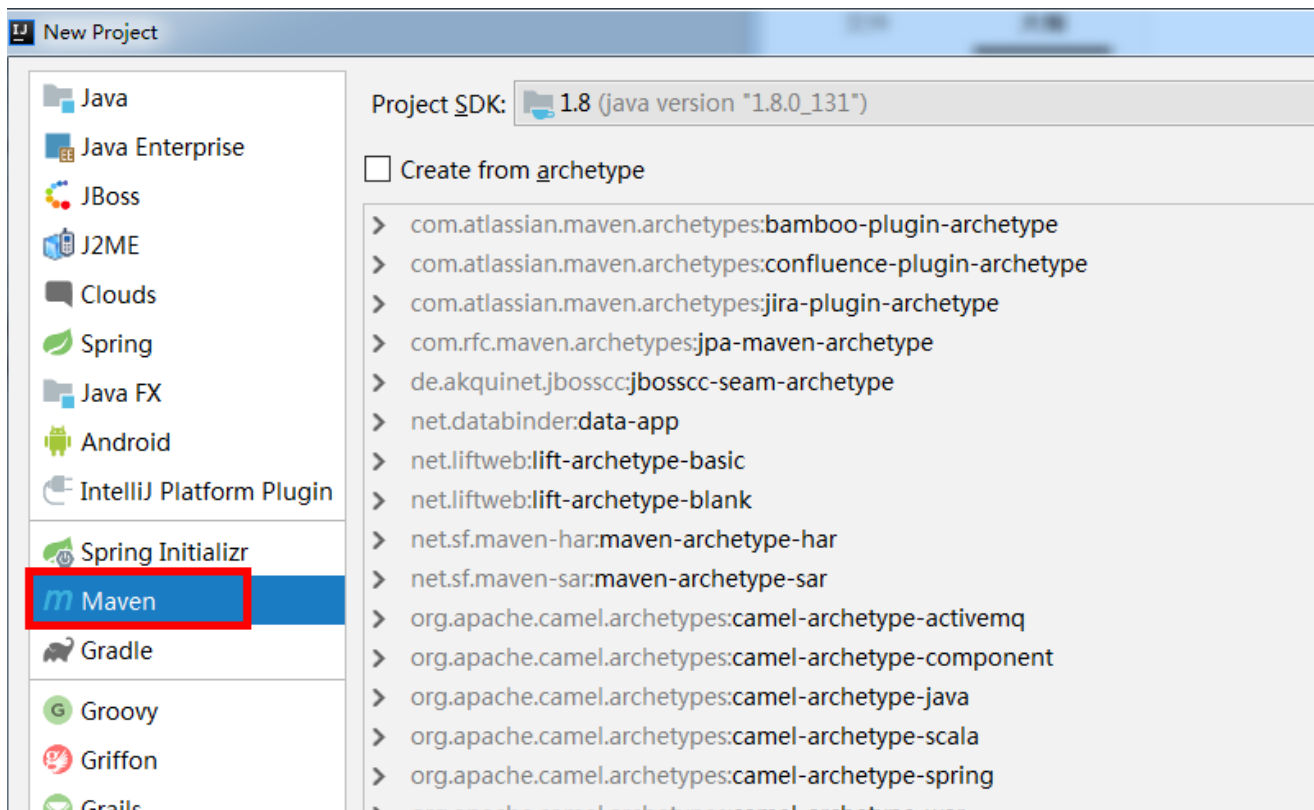
打包—————将编译的非测试类打包

部署—————将打好的包发布到资源仓库中

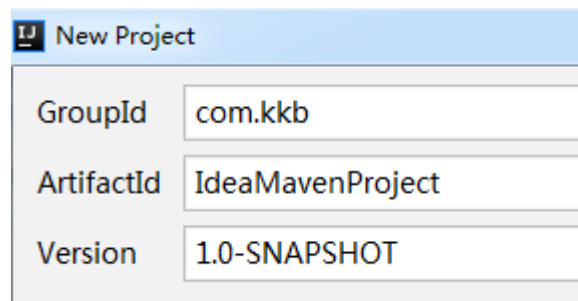
## 8.IDEA搭建Maven项目

---

1.主界面中选择Create New Project,然后在左侧项目类型中选择Maven



2.创建java类型的项目，直接点击右下角的“Next”，填写GroupId,ArtifactId两项内容，填写完毕后，后续一直点击Next即可，最后点击Finsh。

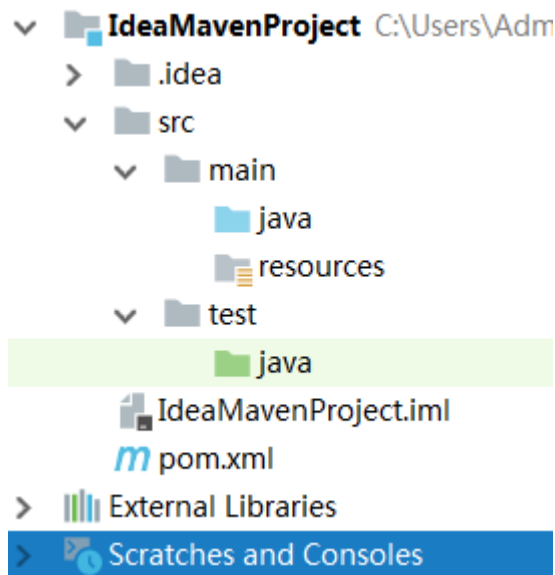


补充: maven中的仓库分为两种，**snapshot快照仓库**和**release发布仓库**。snapshot快照仓库用于保存开发过程中的不稳定版本，release正式仓库则是用来保存稳定的发行版本。定义一个组件/模块为快照版本，只需要在pom文件中在该模块的版本号后加上**-SNAPSHOT**即可(注意这里必须是大写)。

maven2会根据模块的版本号(pom文件中的version)中是否带有-SNAPSHOT来判断是快照版本还是正式版本。如果是快照版本，那么在mvn deploy时会自动发布到快照版本库中，而使用快照版本的模块，在不更改版本号的情况下，直接编译打包时，maven会自动从镜像服务器上下载最新的快照版本。如果是正式发布版本，那么在mvn deploy时会自动发布到正式版本库中，而使用正式版本的模块，在不更改版本号的情况下，编译打包时如果本地已经存在该版本的模块则不会主动去镜像服务器上下载。

所以，我们在开发阶段，可以将公用库的版本设置为快照版本，而被依赖组件则引用快照版本进行开发，在公用库的快照版本更新后，我们也不需要修改pom文件提示版本号来下载新的版本，直接mvn执行相关编译、打包命令即可重新下载最新的快照库了，从而也方便了我们进行开发。

### 3.Idea中Maven项目结构



4.在src/main/java下编写java代码，在src/test/java下编写测试代码

java代码:

```
package com.kkb;

/**
 * Administrator
 * IdeaMavenProject
 * 面向对象面向君 不负代码不负卿
 */
public class Demo1 {

    public void test1(){
        System.out.println("我是Demo1的test1方法");
    }
}
```

测试代码:

```
package com.kkbtest;

/**
 * Administrator
 * IdeaMavenProject
 * 面向对象面向君 不负代码不负卿
 */
import com.kkb.Demo1;
import org.junit.Test; //这里报错是正常的，现在还没有添加依赖包
public class TestDemo1 {

    @Test
    public void testDemo(){
        new Demo1().test1();
    }
}
```

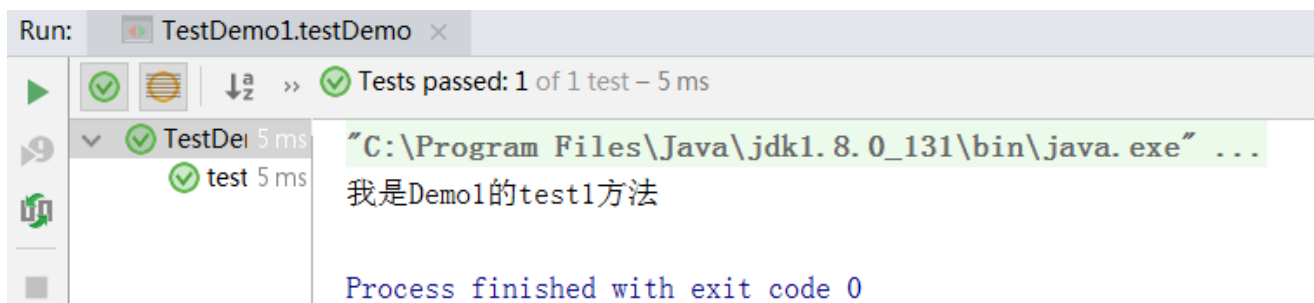
```
}
```

##### 5.在pom.xml文件中添加junit依赖包

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
</dependencies>
```

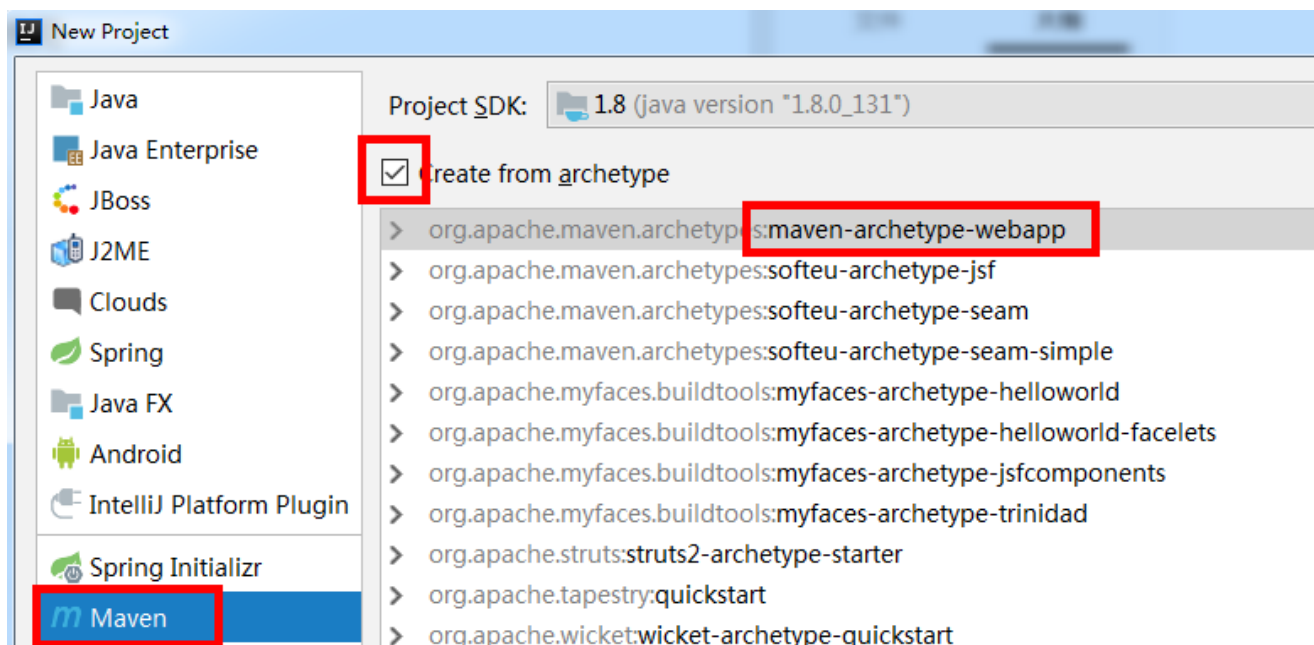
添加完依赖包后，可以在左侧External Libraries目录下查看依赖包，此时TestDemo1.java文件不再报错

##### 6.运行TestDemo1.java文件，在控制台查看运行结果



下面演示如何搭建web项目

1.选择create New Project,左侧选择Maven,勾选右侧Create from archetype选项，选择下面区域中的maven-archetype-webapp类型。

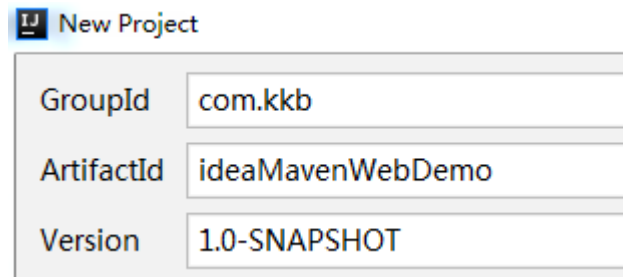


补充:archetype的意思就是模板原型的意思，原型是一个Maven项目模板工具包。

开发中常用的archetype类型:

1:maven-archetype-quickstart(简单的java类型项目,除了pom.xml外, 没有其他的xml了, 但是有main、test两个包, 包里放了一个App、AppTest类) 2:maven-archetype-webapp(一个简单的Java Web应用程序)

2.填写GroupId,ArtifactId的值, 然后点击Next



New Project	
GroupId	com.kkb
ArtifactId	ideaMavenWebDemo
Version	1.0-SNAPSHOT

3.在Properties选项卡中, 点击“+”号, 添加新属性

Name: archetypeCatalog

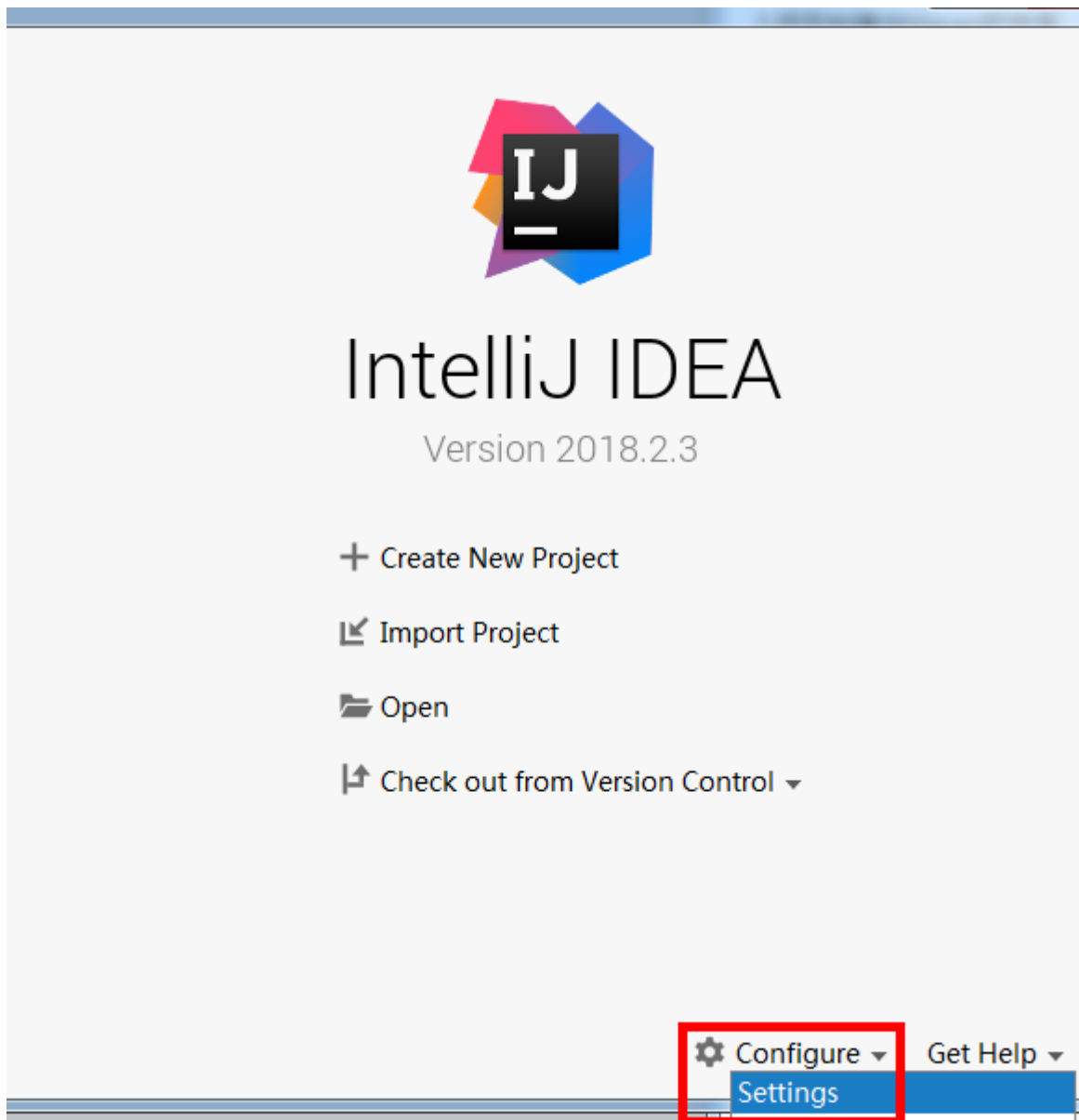
Value :internal

说明:这里是固定值, 注意大小写。后续过程和创建java项目时相同, 这里就不再赘述。

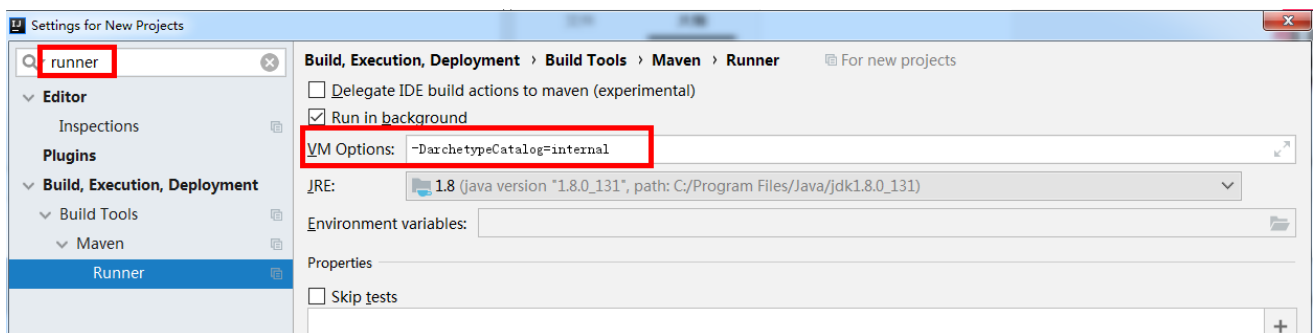
缺点:这种方式需要在每次创建web项目时添加新属性, 太麻烦了。

解决方案:一次配置, 永久使用

1.在idea的主界面选择configure->settings



2.在搜索框中搜索:runner,在VM Option选项卡中添加配置代码。保存退出后,重新启动软件。

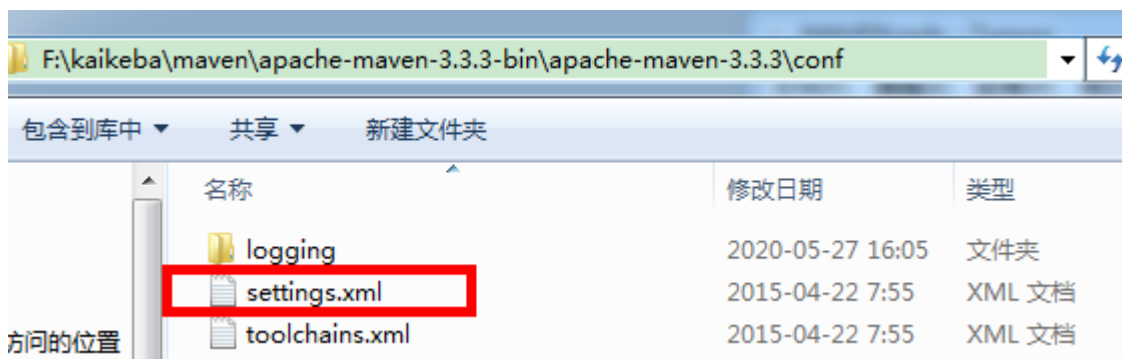


## 9.修改本地仓库

maven下载的依赖包都存在磁盘的哪里了呢?

其实maven设置了自身的默认磁盘路径,当然我们可以通过修改setting.xml文件的内容来修改仓库路径。

setting文件路径(maven路径的conf文件夹下):



默认仓库路径:

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
    | The path to the local repository maven will use to store
    | artifacts.
    |
    | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
-->
```

修改Maven默认仓库:在任意磁盘下创建一个文件夹目录, 该文件夹配置成你的本地仓库路径。

将localRepository标签配置到注释代码的外面, 代码如下:

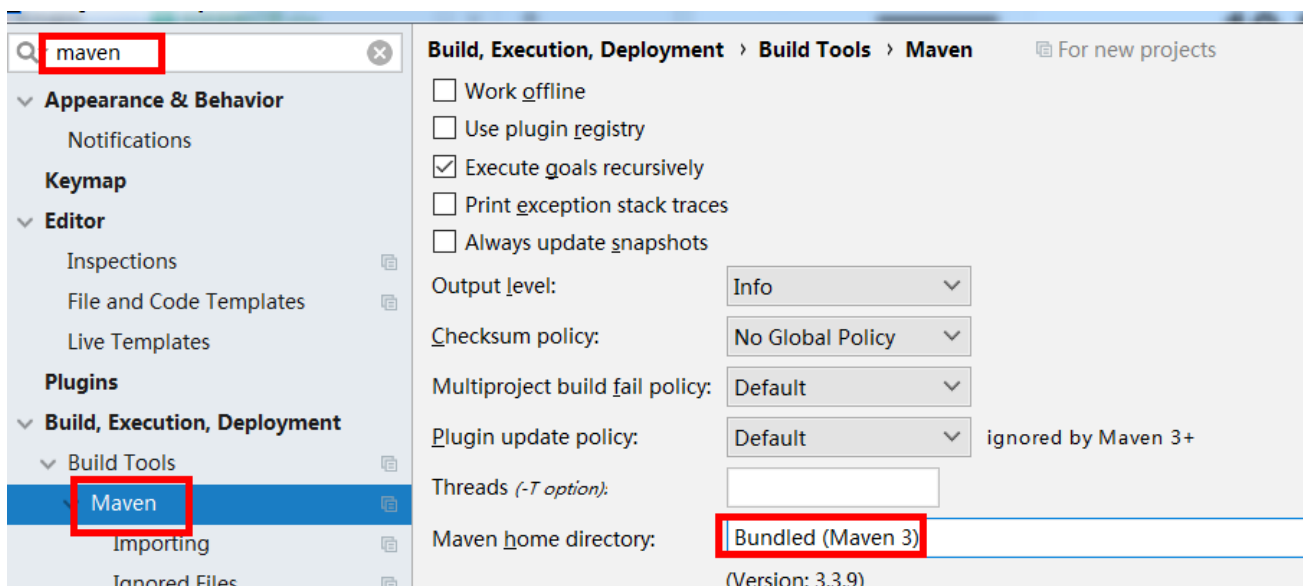
```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
    | The path to the local repository maven will use to store artifacts.
    |
    | Default: ${user.home}/.m2/repository
  -->
  <localRepository>F:\kaikeba\maven\maven_repository</localRepository>
```

## 10.自定义Idea中的Maven版本

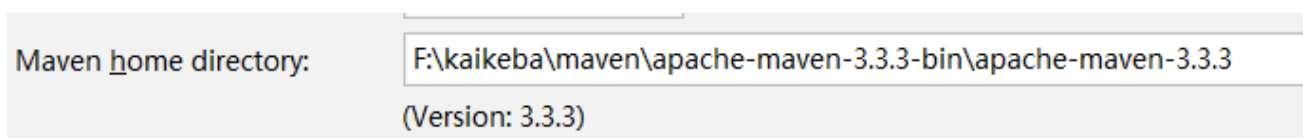
我们现在使用的idea都自带maven,但是实际开发中, 可能idea自带的版本和我们使用的框架或jdk版本不兼容, 那么就需要我们更改idea中的maven版本。

1.打开idea主界面, 选择Configure->Setting,在搜索框搜索“maven”

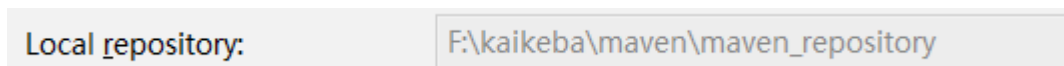




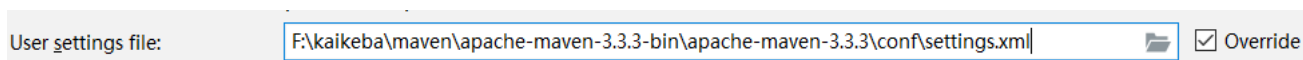
2.修改Maven home directory路径，改成自己的maven路径



此时local repository会自动更新成我们修改的本地仓库路径



3.完成上述步骤即可，当然我们也可以修改setting文件的路径，点击User settings file 选项后的override选项卡



最后保存退出。注意如果没有生效，则按照刚才的步骤重新配置一遍即可。

## 11.Maven坐标的概念

### 11.1 什么是坐标？

在平面几何中坐标 (x,y) 可以标识平面中唯一的一点。在maven中坐标就是为了定位一个唯一确定的jar包。

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

### 11.2 Maven坐标主要组成

groupId：定义当前Maven组织名称

artifactId：定义实际项目名称

version：定义当前项目的当前版本或者是所依赖的jar包的版本

```
<groupId>com.kkb</groupId>
<artifactId>demoProject</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

# 12.Maven的依赖性和传递性

## 12.1 依赖管理

就是对项目中jar 包的管理。可以在pom文件中定义jar包的GAV坐标，管理依赖。

依赖声明主要包含如下元素：

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.10</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

## 12.2 依赖范围(scope标签的取值)

依赖范围 (Scope)	对于主代码 classpath有效	对于测试代码 classpath有效	被打包，对于 运行时 classpath有效	例子
compile	Y	Y	Y	log4j
test	-	Y	-	junit
provided	Y	Y	-	servlet-api
runtime	-	-	Y	JDBC Driver Implementation

scope属性值介绍：

### 1. compile

编译范围，默认scope，在工程环境的 classpath（编译环境）和打包（如果是WAR包，会包含在WAR包中）时候都有效。

### 2.provided

容器或JDK已提供范围，表示该依赖包已经由目标容器（如tomcat）和JDK提供，只在编译的classpath中加载和使用，打包的时候不会包含在目标包中。最常见的是j2ee规范相关的servlet-api和jsp-api等jar包，一般由servlet容器提供，无需在打包到war包中，如果不配置为provided，把这些包打包到工程war包中，在tomcat6以上版本会出现冲突无法正常运行程序（版本不符的情况）。

### 3.runtime

一般是运行和测试环境使用，编译时候不用加入classpath，打包时候会打包到目标包中。一般是通过动态加载或接口反射加载的情况比较多。也就是说程序只使用了接口，具体的时候可能有多个，运行时通过配置文件或jar包扫描动态加载的情况。典型的包括：JDBC驱动等。

4.test

测试范围，一般是单元测试场景使用，在编译环境加入classpath，但打包时不会加入，如junit等。

5.system（一般不用，不同机器可能不兼容）

系统范围，与provided类似，只是标记为该scope的依赖包需要明确指定基于文件系统的jar包路径。因为需要通过systemPath指定本地jar文件路径，所以该scope是不推荐的。如果是基于组织的，一般会建立本地镜像，会把本地的或组织的基础组件加入本地镜像管理，避过使用该scope的情况。

12.3 依赖传递

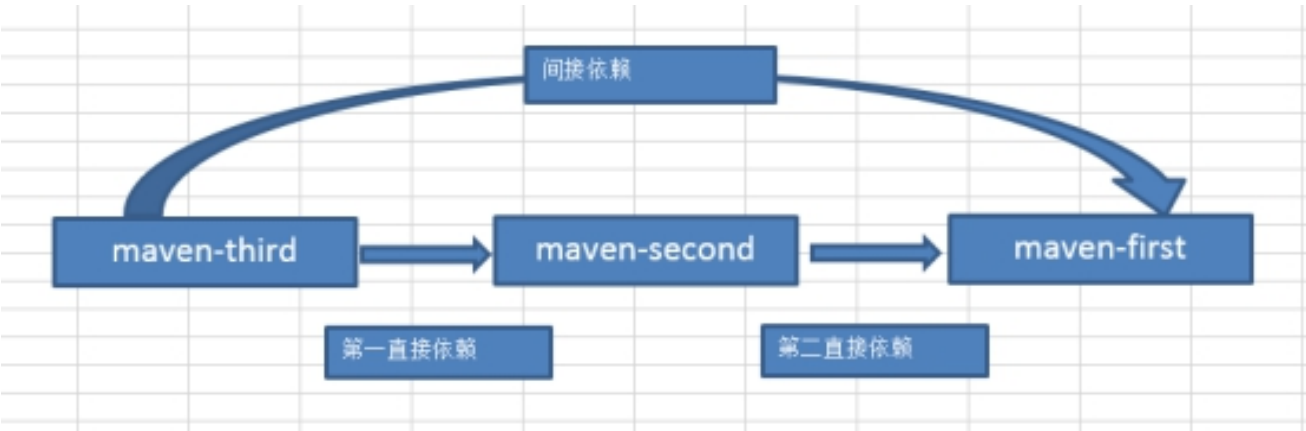
12.3.1 直接依赖和间接依赖

如果B中使用A，C中使用B，则称B是C的**直接依赖**，而称A是C的**间接依赖**。

C->B B->A

C直接依赖B

C间接依赖A



12.3.2 依赖范围对传递依赖的影响

	compile	test	provided	runtime
compile	compile	-	-	runtime
test	test	-	-	test
provided	provided	-	provided	provided
runtime	runtime	-	-	runtime

左边第一列表示第一直接依赖范围

上面第一行表示第二直接依赖范围

中间的交叉单元格表示传递性依赖范围。

定义在标签

总结：

- (1) 当第二依赖的范围是compile的时候，传递性依赖的范围与第一直接依赖的范围一致。
- (2) 当第二直接依赖的范围是test的时候，依赖不会得以传递。
- (3) 当第二依赖的范围是provided的时候，只传递第一直接依赖范围也为provided的依赖，且传递性依赖的范围同样为 provided；
- (4) 当第二直接依赖的范围是runtime的时候，传递性依赖的范围与第一直接依赖的范围一致，但compile例外，此时传递的依赖范围为runtime；

#### 12.4 依赖冲突

- (1) 如果直接与间接依赖中包含有同一个坐标不同版本的资源依赖，以直接依赖的版本为准（就近原则）
- (2) 如果直接依赖中包含有同一个坐标不同版本的资源依赖，以配置顺序下方的版本为准（就近原则）

#### 12.5 可选依赖

true/false 用于设置是否可选，也可以理解为jar包是否向下传递。

在依赖中添加optional选项决定此依赖是否向下传递，如果是true则不传递，如果是false就传递，默认为false。

#### 12.6 排除依赖

maven 的传递依赖能自动将间接依赖引入项目中来，这样极大地简化了项目中的依赖管理，但是有时候间接依赖的关联包可以因为版本或其他原因，并不是我们想要的版本，那该怎么办呢？

这种做法就是排除依赖。那怎么实现排除依赖呢？实现排除依赖还是比较简单的，在直接依赖的配置里面添加exclusions→exclusion 元素，指定要排除依赖的 groupId 和 artifactId 就行，如下面代码所示。

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-core</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

说明:排除依赖包中所包含的依赖关系，不需要添加版本号。

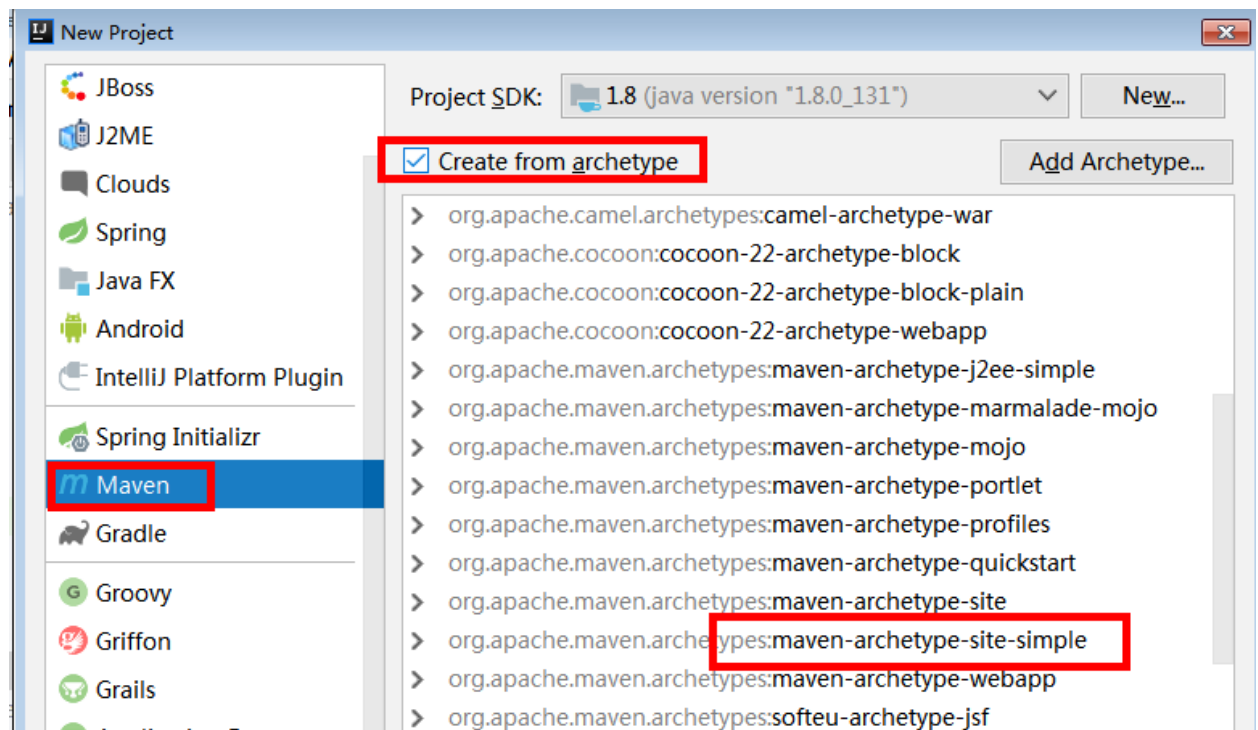
## 13.Maven的继承性

在一家开发公司里，很多项目中使用的jar包里80%是重复的，难道要每个项目都把这些pom.xml文件的依赖包复制过来吗？如果后期出现统一的版本升级，那岂不是修改起来也很麻烦？如何解决呢？这里可以使用maven的继承特性来消除重复依赖，可以把很多相同的配置提取出来。例如：groupId, version等

#### 15.1 实现maven继承性的步骤：

## 1. 创建父工程

父工程的打包类型必须是POM。步骤如图:



注意这类项目和原来项目的区别在于，打包方式是pom

```
<groupId>com.kkb</groupId>
<artifactId>father</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>pom</packaging>
```

由于pom项目一般都是用来做父项目的，所以该项目的src文件夹可以删除掉。

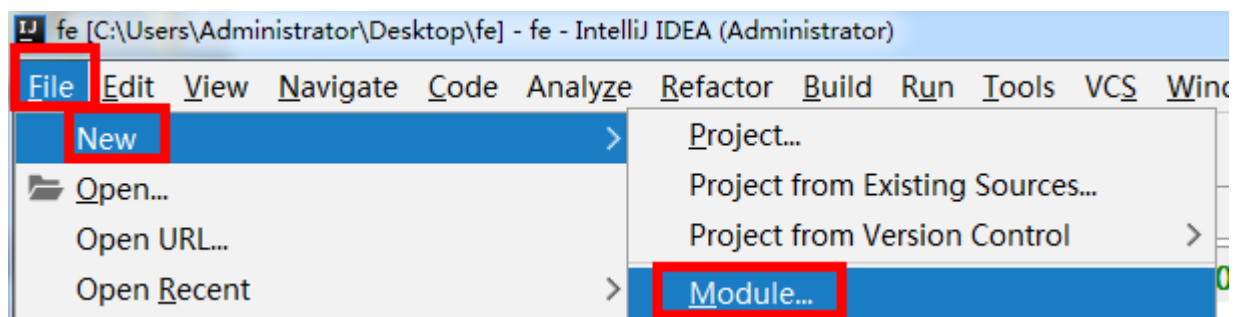
## 2. 创建子工程

创建方式有两种：

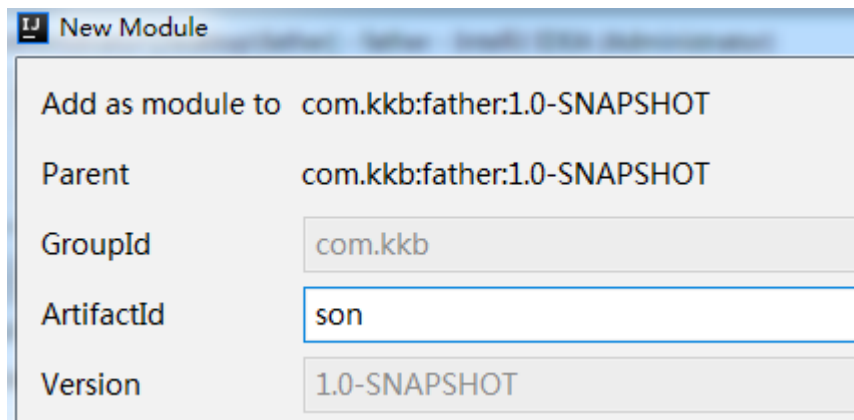
一种是创建新工程为子工程，在创建时设置父工程的GAV。

一种是修改原有的工程为子工程，在子工程的pom.xml文件中手动添加父工程的GAV

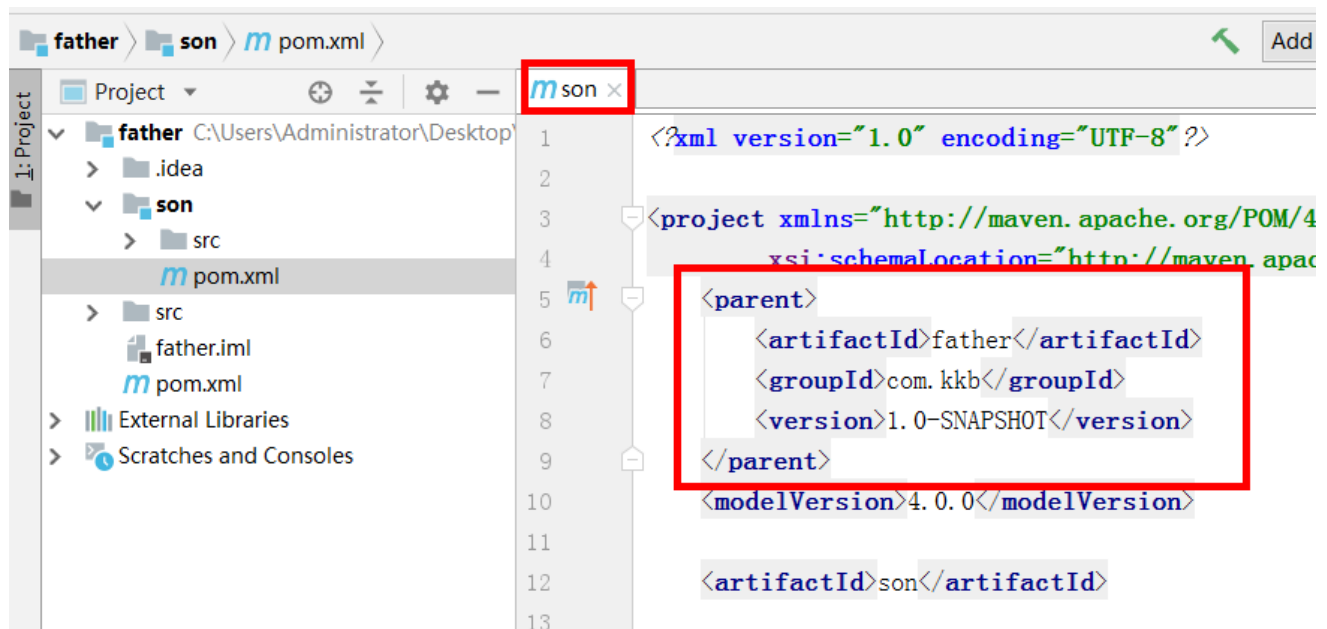
创建子项目步骤:先选中父项目的项目名，然后再按照图片步骤执行



此时只需要添加子项目名称即可,然后一直下一步:



结果:在子项目的pom文件中,默认通过parent标签来继承父项目中的依赖包。这样避免了依赖包的重复依赖



## 15.2 父工程统一管理版本号

在开发中依赖的jar包众多,对应的版本也是很杂乱,那有没有一种方式可以统一管理这么jar包的版本值呢?当然是有的,Maven使用dependencyManagement来管理依赖的版本号。

**注意: 此处只是定义依赖jar包的版本号,并不实际依赖。如果子工程中需要依赖jar包还需要添加dependency节点。**

中的jar直接加到项目中,管理的是依赖关系(如果有父pom,子pom,则子pom中只能被动接受父类的版本);

主要管理版本,对于子类继承同一个父类是很有用的,集中管理依赖版本不添加依赖关系,对于其中定义的版本,子pom不一定要继承父pom所定义的版本。

**父工程: 声明版本**

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.kkb</groupId>
      <artifactId>demo2</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

子工程：不需要指定版本号(先继承父元素)

```
<parent>
  <groupId>com.kkb</groupId>
  <artifactId>father</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>

<dependencies>
  <dependency>
    <groupId>com.kkb</groupId>
    <artifactId>demo2</artifactId> <!--这里不需要指定版本-->
  </dependency>
</dependencies>
```

### 6.5.5 父工程中版本号提取

当父工程中定义的jar包越来越多，找起来越来越麻烦，所以可以把版本号提取成一个属性集中管理。

```
<properties>
  <log4j.version>1.2.9</log4j.version>
</properties>
```

//注意这里的log4j.version的值是自定义的，但命名时不要使用中文或特殊字符等，标签对中间写版本值使用:\${log4j.version}读取该变量值，代码如下

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${log4j.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

子工程的jar包版本不受影响：

## 14.Maven的生命周期

---

### 13.1 什么是生命周期?

Maven生命周期就是为了对所有的构建过程进行抽象和统一。包括项目清理、初始化、编译、打包、测试、部署等几乎所有构建步骤。生命周期可以理解为构建工程的步骤。

在Maven中有三套相互独立的生命周期，请注意这里说的是“三套”，而且“相互独立”，这三套生命周期分别是：

(1) Clean Lifecycle： 在进行真正的构建之前进行一些清理工作。 Mvn clean

(2) Default Lifecycle： 构建的核心部分，编译，测试，打包，部署等等。

Mvn compile test package install deploy

(3) 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。

### 13.2 Maven三大生命周期

#### 13.2.1 clean： 清理项目

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，这个的clean是Clean生命周期的一个阶段。有Clean生命周期，也有clean阶段。Clean生命周期一共包含了三个阶段：

(1) pre-clean:执行清理前需要完成的工作

(2) clean:清理上一次构建过程中生成的文件，比如编译后的class文件等。

(3) post-clean:执行清理后需要完成的工作

#### 13.2.2 default： 构建项目

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 验证项目结构是否正常，必要的配置文件是否存在

initialize 做构建前的初始化操作，比如初始化参数，创建必要的目录等

generate-sources 产生在编译过程中需要的源代码

process-sources 处理源代码，比如过滤值

**generate-resources** 产生主代码中的资源在classpath中的包

**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 产生编译过程中生成的文件

generate-test-sources 产生编译过程中测试相关的代码

process-test-sources 处理测试代码

**generate-test-resources** 产生测试中资源在classpath中的包

**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 产生编译测试代码过程的文件

**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 处理打包前需要初始化的准备工作

package 接受编译好的代码，打包成可发布的格式，如 JAR。

pre-integration-test 做好集成测试前的准备工作，比如集成环境的参数设置

integration-test 集成测试

post-integration-test 完成集成测试前的准备工作，比如集成环境的参数设置

verify 检测测试后的包是否完好

**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。

运行任何一个阶段的时候，它前面的所有阶段都会被运行，这也就是为什么我们运行mvn install 的时候，代码会被编译，测试，打包。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。

参考资料地址:<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html>

### 13.2.3 site：生成项目站点

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。

## 15.Maven插件

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。每个插件都能实现一个功能，每个功能就是一个插件目标。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的一个插件目标

### 14.1 Maven编译插件

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

或者直接修改maven的setting.xml文件

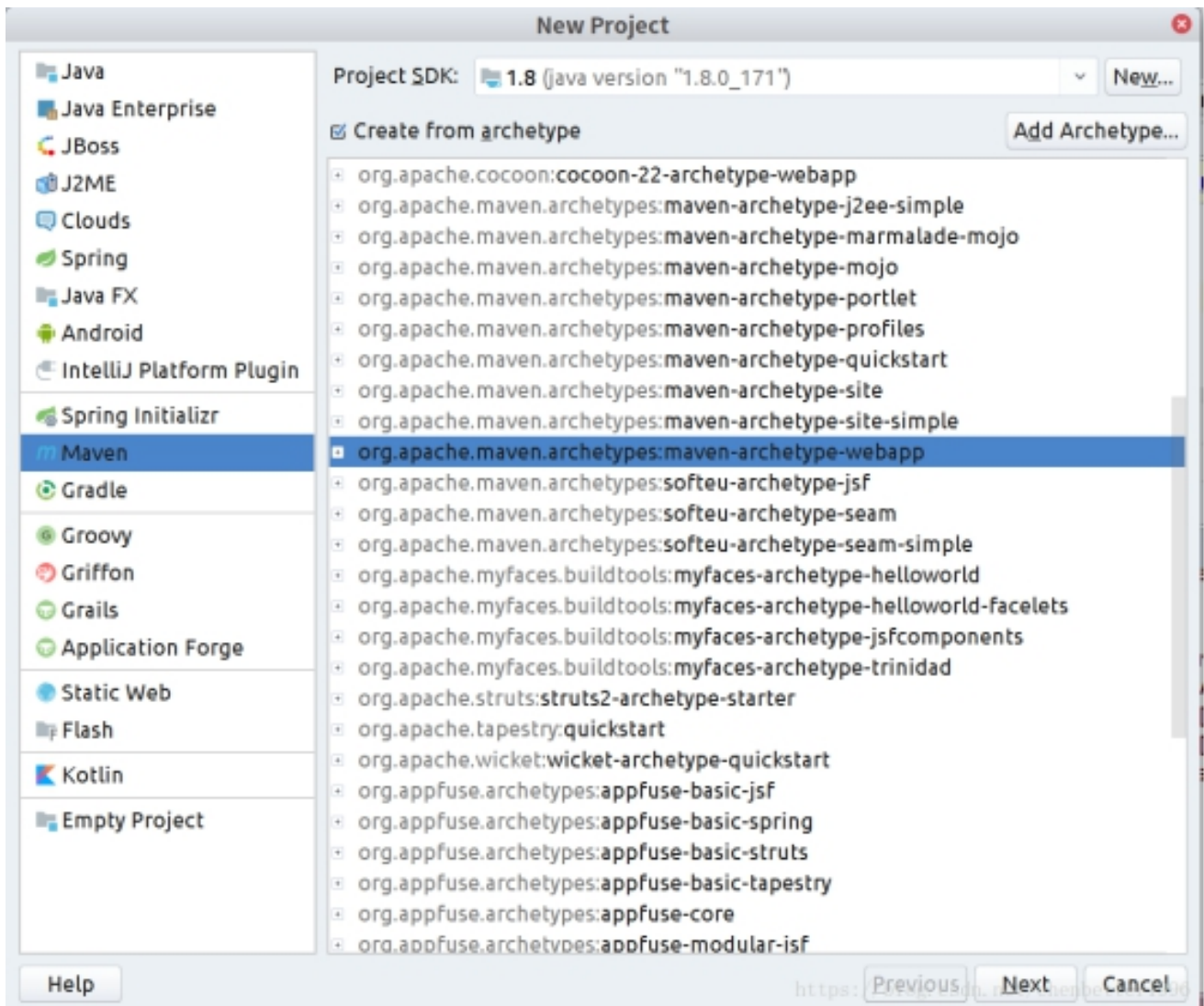
在里面添加如下内容：

```
<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>
```

## 14.2 Tomcat插件

我们之前创建的web项目都需要额外配置tomcat以后才能运行项目，现在maven提供了tomcat插件，这样我们就无需再添加额外的tomcat了。

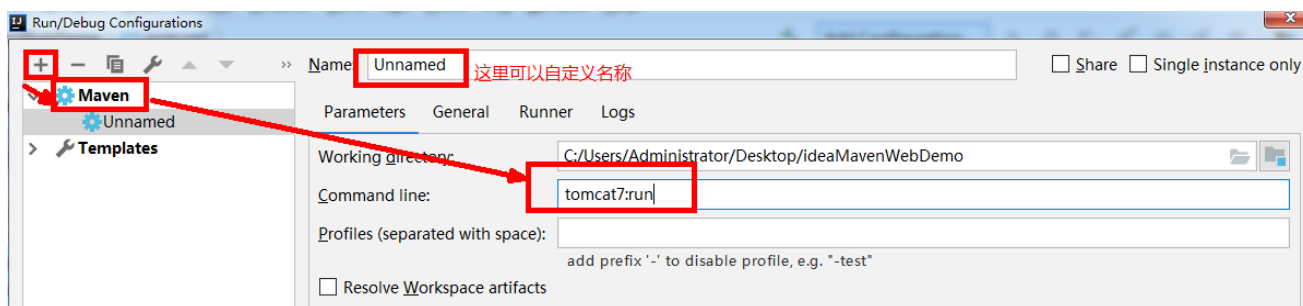
步骤1:创建maven类型的web工程



步骤2: pom.xml文件中添加插件信息

```
<plugins>
  <plugin>
    <!-- 配置插件 -->
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <configuration>
      <port>8080</port>
      <path>/</path>
    </configuration>
  </plugin>
</plugins>
```

步骤3:点击“add configuration”,按照图片中的步骤添加tomcat命令，运行tomcat插件



运行方式1:

命令:tomcat:run 运行tomcat6 (默认)

运行方式2:

命令:tomcat7:run 运行tomcat7 (推荐, 但是需要添加插件)

## 16.Maven私服介绍

### 16.1 什么是Maven仓库?

用来统一存储所有Maven共享构建的位置就是仓库。根据Maven坐标定义每个构建在仓库中唯一存储路径大致为: groupId/artifactId/version/artifactId-version.packaging

### 16.2 仓库的分类

#### 1、本地仓库

~/m2/repository

每个用户只有一个本地仓库

#### 2、远程仓库

中央仓库: Maven默认的远程仓库, 不包含版权资源

<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



## 17.Maven私服-Nexus搭建与使用

### 17.1 安装Nexus

为所有来自中央仓库的构建安装提供本地缓存。

下载网站: <https://help.sonatype.com/repomanager3/download/download-archives---repository-manager-3>

注意:安装JDK1.8以上的版本

(1) 解压nexus压缩包

(2) 打开window的命令窗口, 切换到解压的文件夹中bin目录下, 运行nexus.exe /install nexus 命令, 注册服务



(3) 启动Nexus服务:在bin文件夹下运行nexus /run命令



注意:启动命令会因为版本不同, 指令也不同

(4) 打开网站

<http://localhost:8081>



(5) 使用默认的用户名和密码登陆 (admin/admin123)

查看现有仓库:



proxy 表示远程获取

group 组

hosted 本机, 第三方jar, 如数据库驱动等, 需要手动添加

maven-central: maven中央库, 默认从<https://repo1.maven.org/maven2/>获取jar

maven-releases: 私库发行版jar

maven-snapshots: 私库快照 (调试版本) jar

maven-public: 仓库分组, 把上面三个仓库组合在一起对外提供服务, 在本地maven基础配置settings.xml中使用

17.2 配置所有构建均从私服下载

1、添加镜像配置; 将所有访问外网仓库的请求指向私服;

settings.xml文件

```
<mirror>
  <id>nexus</id>
  <mirrorOf>*</mirrorOf> <!--匹配所有的远程仓库-->
  <url>http://localhost:8081/repository/maven-public/</url>
</mirror>
```

说明:url配置, 描述的便是 上一步中提到的那个**Public Repositories**的信息, 这个地址便是他的地址, 这些写都可以再界面上看到, 这里的是localhost是在本机搭建测试用的, 如果是公司内部仓库的话, 可自行修改成公司内网ip地址

2.添加中央仓库配置

```
<profile>
  <id>nexus</id>
  <repositories>
    <repository>
      <id>central</id>
      <url>https://repo1.maven.org/maven2/</url>
      <releases><enabled>true</enabled></releases>
      <snapshots><enabled>true</enabled></snapshots>
    </repository>
  </repositories>
</profile>
```

### 3.生效配置

```
<activeProfiles>
  <activeProfile>nexus</activeProfile>
</activeProfiles>
```

### 17.3 配置maven阿里云镜像仓库

阿里云镜像仓库是国内服务器，下载速度更快。

修改setting文件:

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