# maven学习

## 下载maven并解压

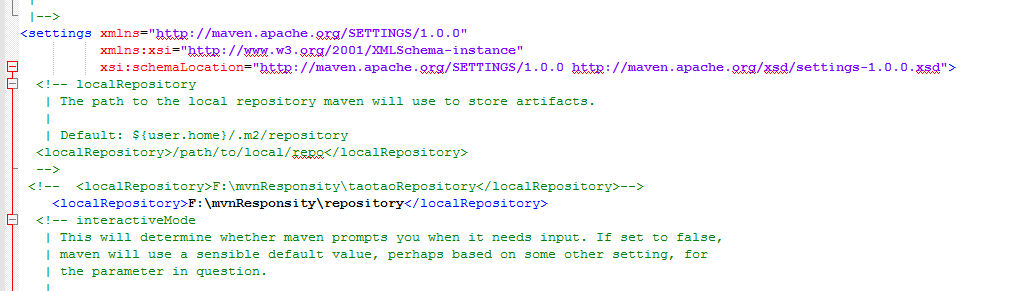
设置maven仓库地址进入$MAVEN\_HOME%\conf目录下,编辑

settings.xml修改

<localRepository>F:\mvnResponsity\repository</localRepository>

默认地址是

${user.home}/.m2/repository



## 配置maven环境变量

MAVEN\_HOME=F:\bag\maven\apache-maven-3.3.9

%MAVEN\_HOME %\bin

验证是否安装成功

mvn –version



## 创建maven-demo

### 创建pom.xml

在D:\mavenStudy\01下创建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/maven-v4\_0\_0.xsd">  <!--描述这个POM文件是遵从哪个版本的项目描述符-->  <modelVersion>4.0.0</modelVersion>  <!--创建groupid,其实就是项目的包名-->  <groupId>com.rkg</groupId>  <!--项目名称(项目中的哪个模块)-->  <artifactId>mavenStudy</artifactId>  <!--版本(SNAPSHOT快照)-->  <version>0.0.1-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  </dependencies>    </project> |

### 创建java文件

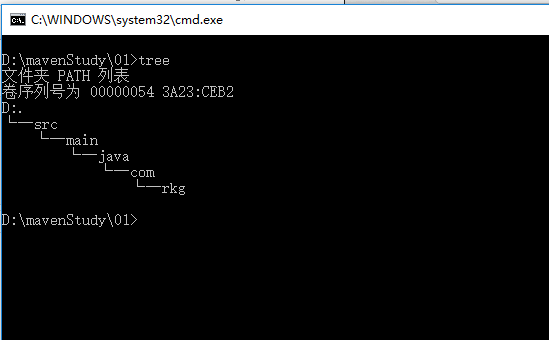
在D:\mavenStudy\01目录下创建src/main/java文件夹(固定写法,mavan中java文件需要放在这个目录下)

在D:\mavenStudy\01\src\main\java创建Hello.java内容如下

|  |
| --- |
| package com.rkg;  public class Hello{  public static void main(String[] args) {  System.out.println("hello maven");  }    public String sayHello(String name){  return "hello:"+name;  }  } |

由于包名是com.rkg所以需要将Hello.java放到D:\mavenStudy\01\src\main\java\com\rkg目录下

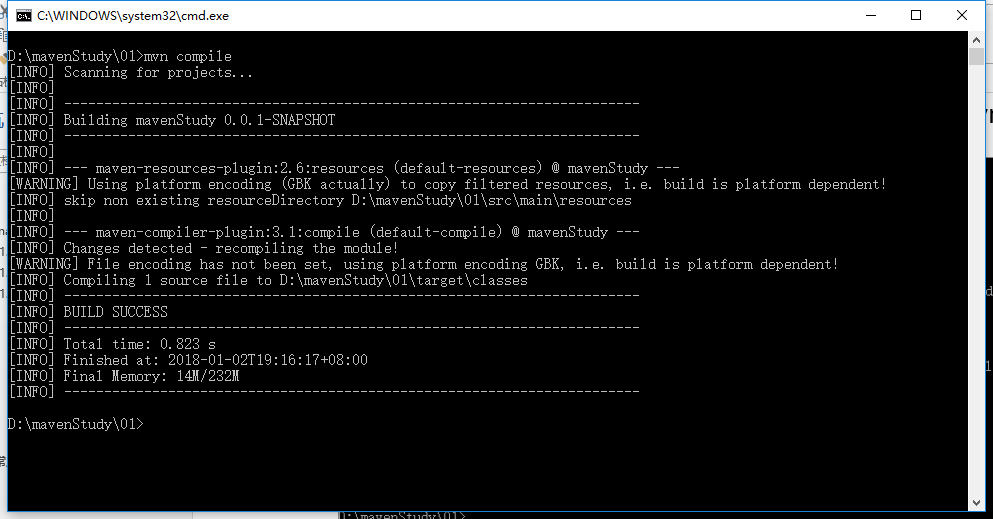
目录结构



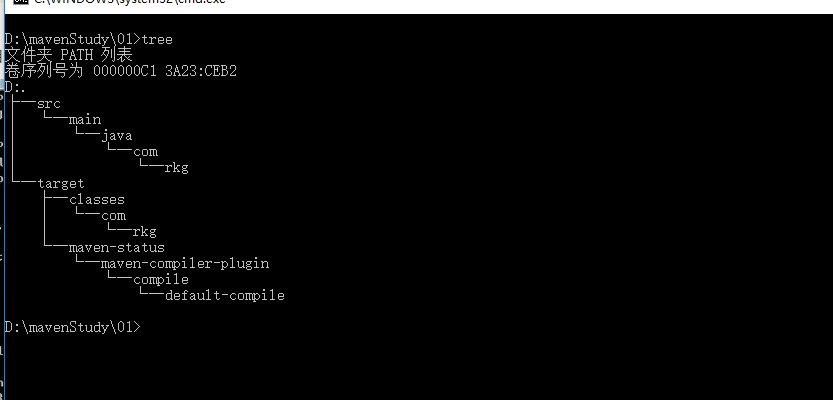
### 编译java文件mvn compile

打开命令行进入D:\mavenStudy\01目录下执行mvn compile

在target目录下生成class文件



此时查看目录结构

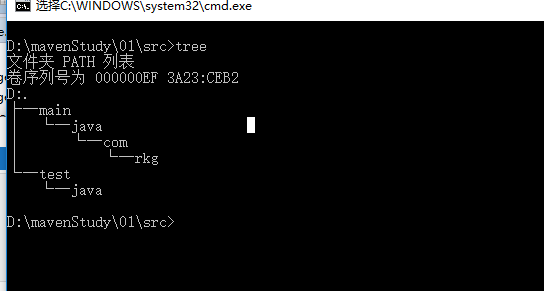


### 创建测试文件

在D:\mavenStudy\01\目录下创建src\test\java目录(固定写法,maven中的测试文件放在这里)并在D:\mavenStudy\01\src\test\java目录下创建TestHello.java文件,内容如下

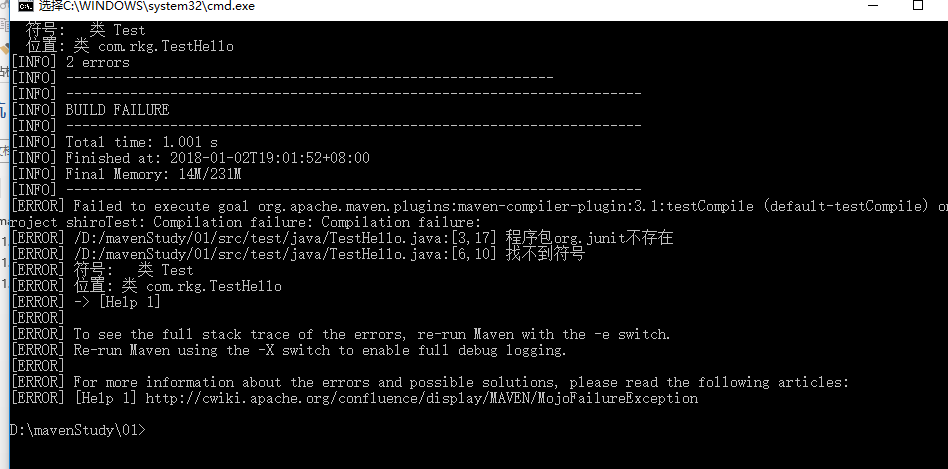
|  |
| --- |
| package com.rkg;  import org.junit.Test;  public class TestHello{  @Test  public void testHello(){  Hello h=new Hello();  System.out.println(h.sayHello("zhangsan"));  }  } |

目录结构



### 编译测试文件mvn test

打开dos命令行在D:\mavenStudy\01目录执行mvn test

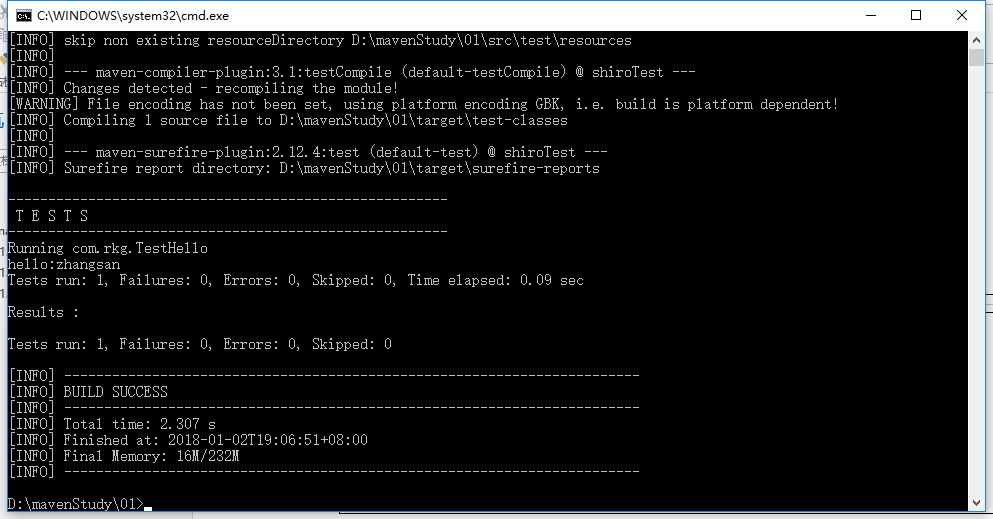


提示如下错误程序包org.junit不存在

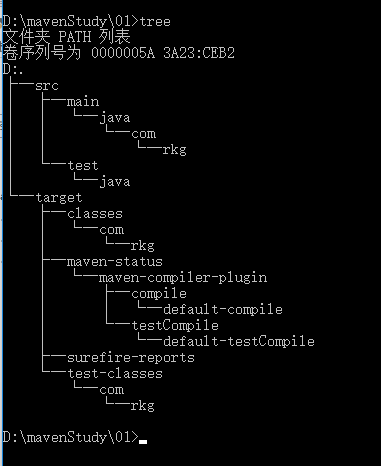
修改pom.xml文件添加junit依赖

|  |
| --- |
| <?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/maven-v4\_0\_0.xsd">  <!--描述这个POM文件是遵从哪个版本的项目描述符-->  <modelVersion>4.0.0</modelVersion>  <!--创建groupid,其实就是项目的包名-->  <groupId>com.rkg</groupId>  <!--项目名称(项目中的哪个模块)-->  <artifactId>shiroTest</artifactId>  <!--版本(SNAPSHOT快照)-->  <version>0.0.1-SNAPSHOT</version>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  </dependencies>    </project> |

重新执行mvn test,提示测试成功

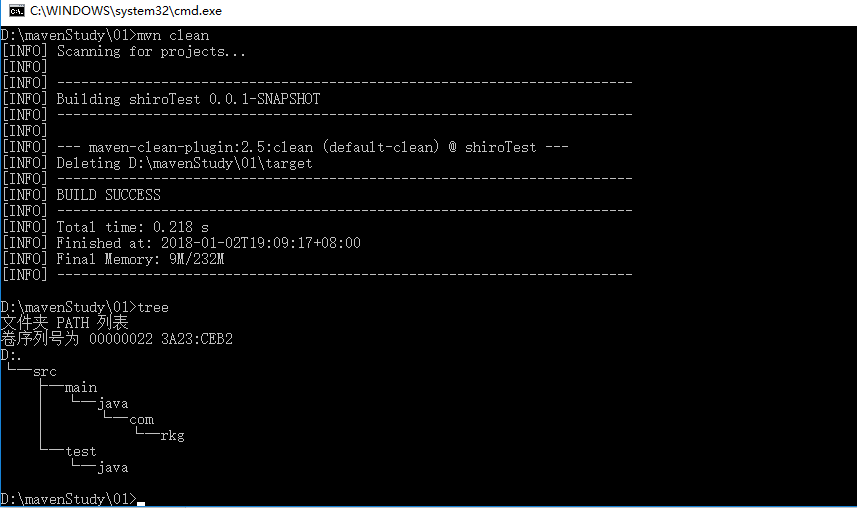


此时目录结构



### 清理编译的文件mvn clean(target目录)

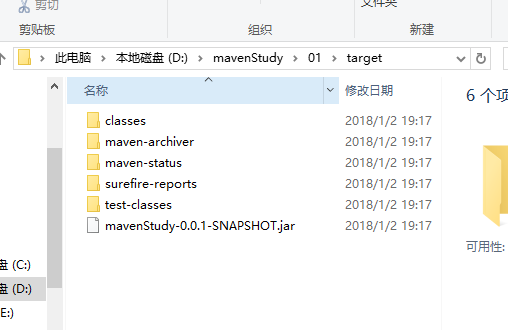
打开dos命令行在D:\mavenStudy\01目录执行mvn clean会发现之前编译的class文件都被删除了



### 清理并打包mvn clean package

将java项目打成jar文件

查看target目录发现会有一个jar文件



### 清理并将项目打包到本地仓库中mvn clean install

此时本地仓库中会生成的项目的java文件,在别的项目中就可以使用了

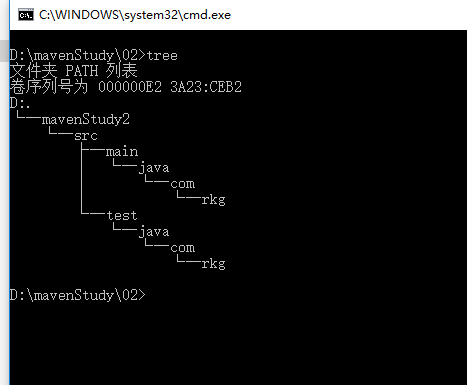


## 使用Archetype生成项目骨架

使用mvn archetype:generate命令可以自动帮我们生成项目结构

会看到长长的输出，有很多可用的Archetype供选择；每一个Archetype前面都有一个对应的编号，同时命令行会提供一个默认的编号，其对应的Archetype为maven-archetype-quickstart，回车选择该Archetype:

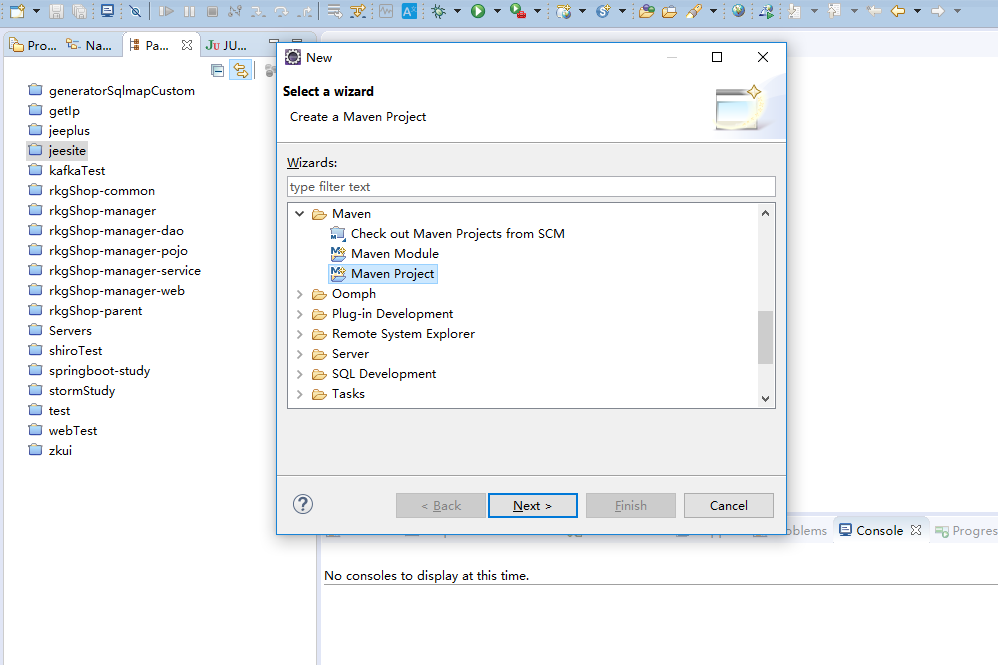
紧接着Maven会提示输入要创建项目的groupId,artifactId,version及包名package：



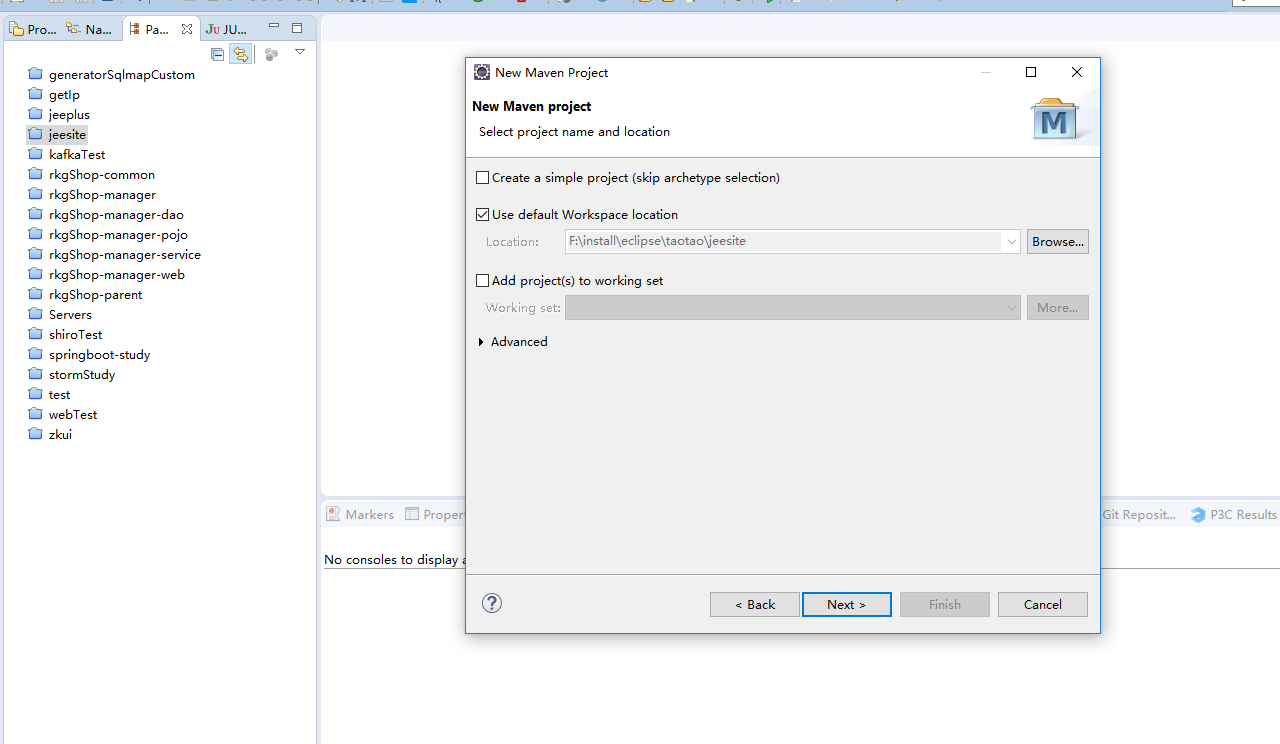
# maven依赖

## 创建maven工程

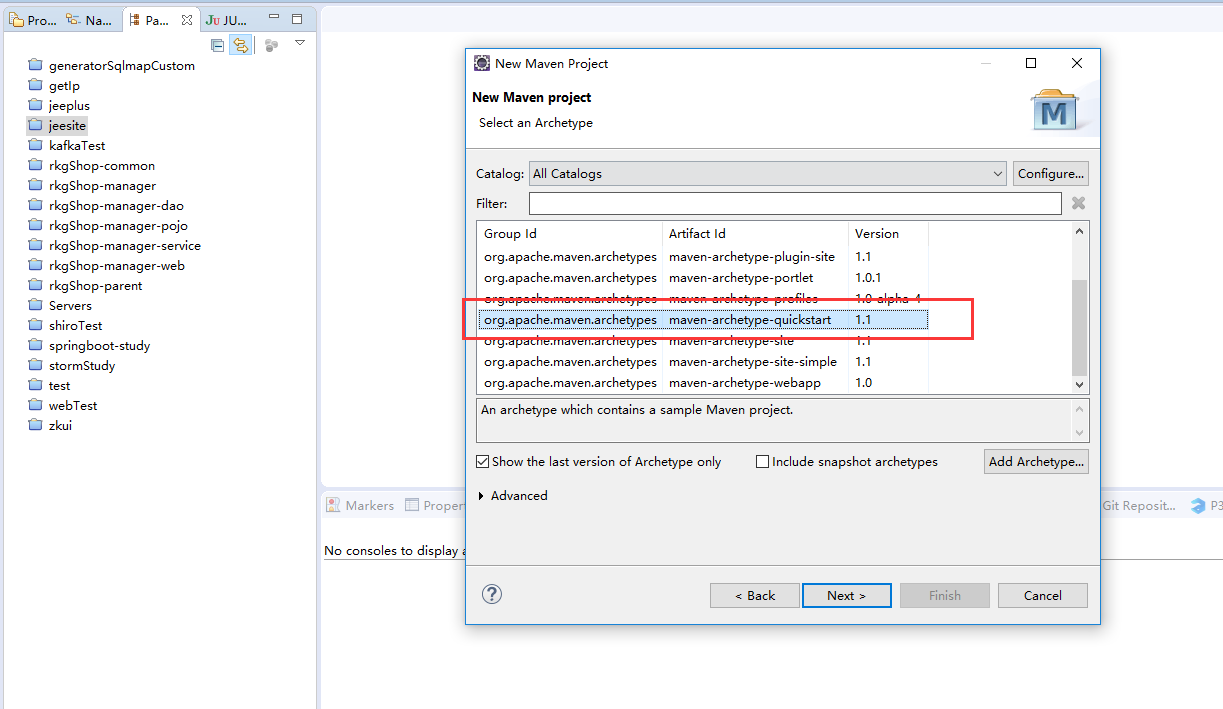
使用eclipse右键🡪new🡪maven project



## 选择工作空间,使用默认即可



## 选择默认quickstart



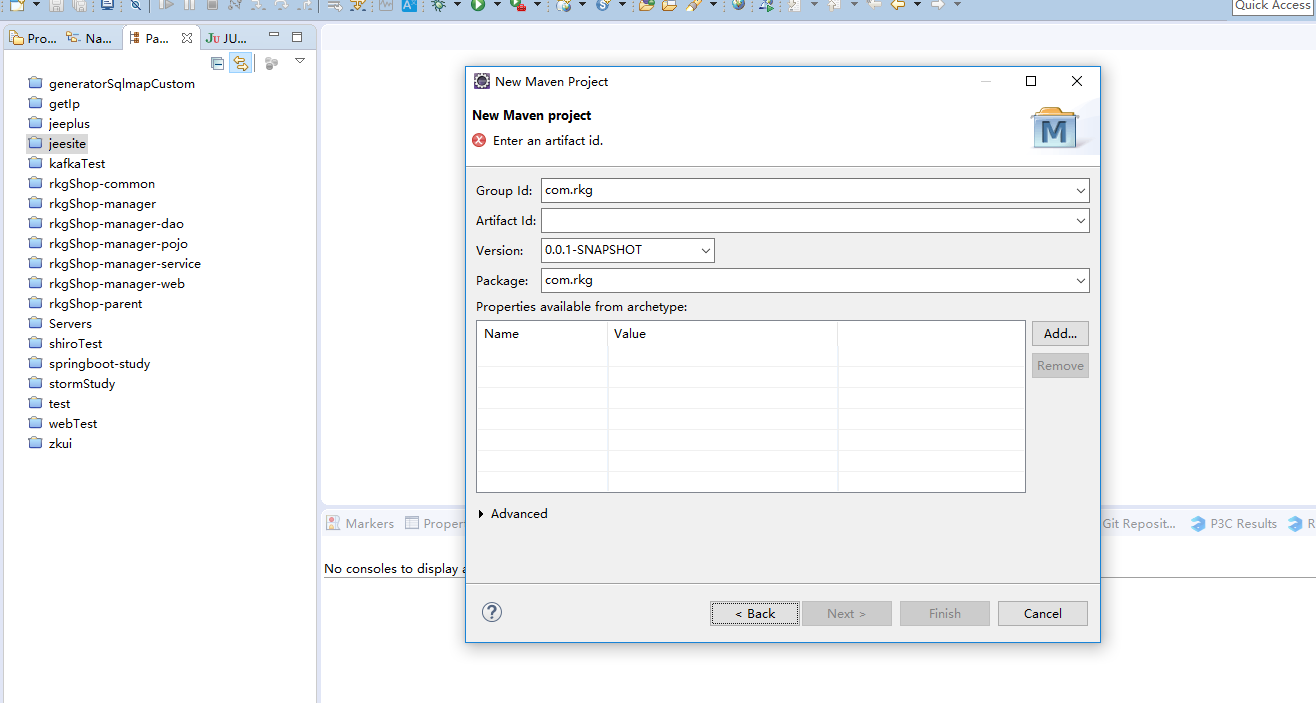
## 设置坐标

groupId: 定义当前Maven项目隶属的实际项目

artifactId: 该元素定义实际项目中的一个Maven项目或模块

version: 该元素定义Maven项目当前所处的版本

package:创建默认App.java的目录



## 说明

### maven所有的依赖都是通过坐标来进行存储的(GAV:groupId, artifactId,version)

### 有一些网上的仓库提供了坐标的查询,例如

<http://mvnrepository.com/>

### 通过dependency设置依赖



### maven是如何搜索依赖的?

首先会在本地仓库查询,如果本地仓库没有,就去中央仓库查询

### 依赖的范围

<scope>test</scope>依赖范围，默认compile

依赖是会被传递的,例如A依赖C,B依赖A,则B也依赖C(这种依赖基于compile范围)

 依赖范围控制哪些依赖在哪些classpath 中可用，哪些依赖包含在一个应用中。让我们详细看一下每一种范围：

**compile （编译范围）**

compile是默认的范围；在编译,测试,打包等阶段都会被依赖进去

**provided （已提供范围）**

编译和测试的时候会被用到,但是打包的时候不会被依赖进去,例如servlet-api.jar,tomcat的lib下面已经包含了此时再在项目中打进去就会重复了

**runtime （运行时范围）**

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。**test （测试范围）**

test范围依赖指的是测试范围有效,在编译和打包时不会被使用,它们只有在测试编译和测试运行阶段可用。打包时不会被依赖进去

**system**

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。

# 注意

源代码应该放置到src/main/java中

源代码的资源文件应该放置到src/main/resources中

测试代码应该放置到src/test/java中

测试资源文件应该放置到src/test/resources中

# maven坐标

**groupId**: 定义当前Maven项目隶属的实际项目

**artifactId**: 该元素定义实际项目中的一个Maven项目或模块

**version**: 该元素定义Maven项目当前所处的版本

**packaging**: 该元素定义Maven项目的打包方式

**classifier**: 该元素用来帮助定义构建输出的一些附属构件

注：groupId、artifactId、version、packaging是必须定义的，classifier是不能被直接定义的，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成的。

# 常用mvn命令

|  |
| --- |
| **mvn archetype:create 创建** **Maven 项目** **mvn compile 编译主程序**源代码，**不会编译test目录的源代码**。第一次运行时，会下载相关的依赖包，可能会比较费时 **mvn test-compile 编译测试代码**，compile之后会生成target文件夹，**主程序**编译在classes下面，**测试程序**放在test-classes下 **mvn test 运行**应用程序中的单元测试  **mvn site 生成**项目相关信息的网站 **mvn clean 清除**目标目录中的生成结果 **mvn package 依据项目**生成 jar 文件，打包之前会进行编译，测试 **mvn install在本地** Repository 中安装 jar。 **mvn eclipse:eclipse 生成** **Eclipse 项目文件**及**包引用定义** **mvn deploy**在**整合**或者**发布**环境下执行，将最终版本的包拷贝到远程 的repository(私服nexus)，使得其他的开发者或者工程可以共享。 |