实验指导书

# 实验一 环境架设

|  |  |  |  |
| --- | --- | --- | --- |
| 实验项目 | 实验人员 | 地点 | 日期 |
| Maven环境架设 | 李勇 | 4302教室 | 2018/8/31 |

* 1. 实验目的：

熟悉Maven在Windows或Unix下的安装流程和配置方式；学会根据遇到的状况排查问题的方法。

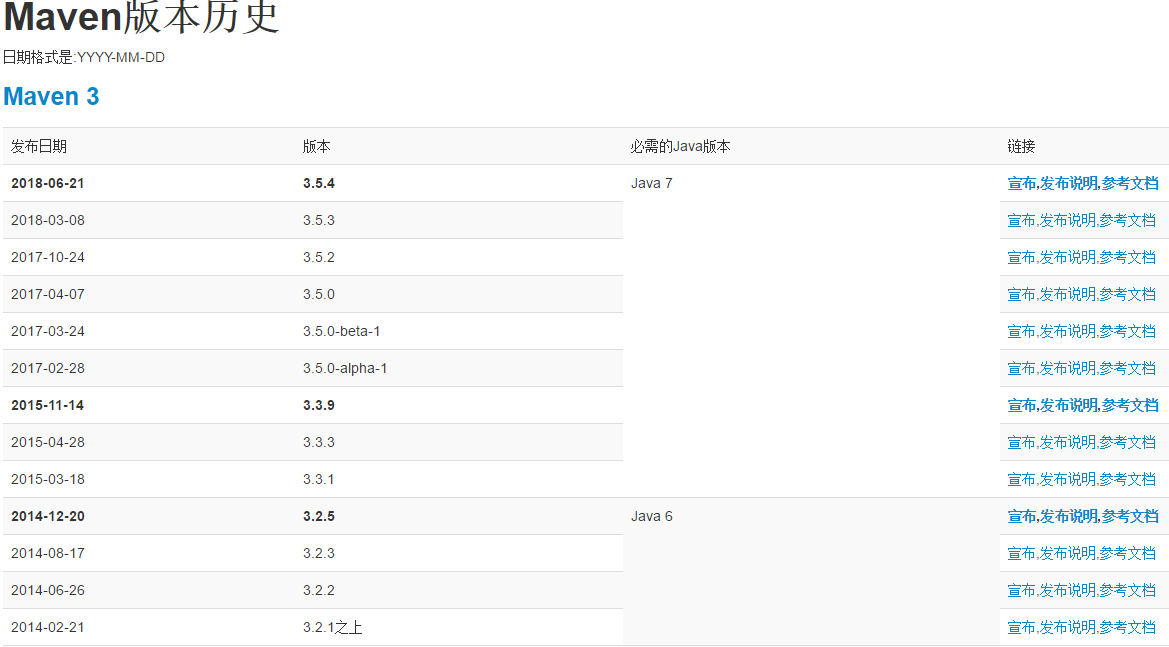
## 1.2 实验内容：

完成基础的Maven在Windows环境下的安装、M2eclipse插件的安装；有余力的学员可尝试在Unix环境下进行Maven的安装。

## 1.3 实验步骤：

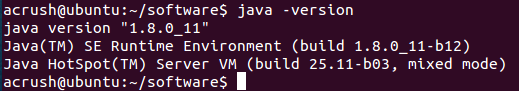
### 1.3.1安装maven

Maven运行于JDK之上，所以安装maven之前需要安装好JDK。本次实验我们选用的maven版本是3.5.4，在maven官网上可以得知对应支持的JDK版本为JDK1.7。maven历史版本及对应必须的JDK版本如下图：



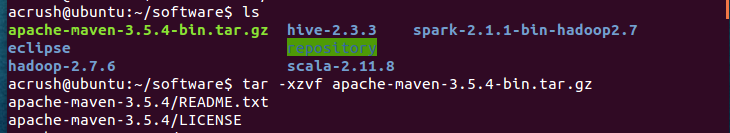
Maven历史版本及所需的JDK版本 图

实验中我使用的是linux，如下图查看java版本是否满足JDK1.7及以上。（如果不满足可以去甲骨文官网下载安装）



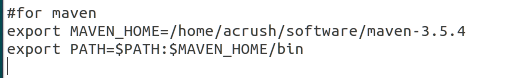
查看系统安装的JDK版本 图

接下来就可以安装maven，首先解压安装包至指定的一个目录，命令是tar -xzvf apache-maven-3.5.4-bin.tar.gz，修改目录名称如下图：



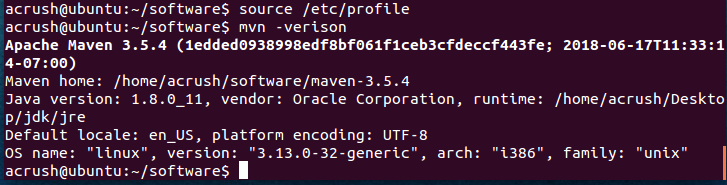
解压安装包 图

接下来是将maven的可执行命令所在的位置添加到环境变量中，使得我们在使用maven时可以直接使用，不用输入完整的路径。打开/etc/profile文件，添加maven的解压目录，添加maven解压目录的bin子目录到path中，如下图所示：



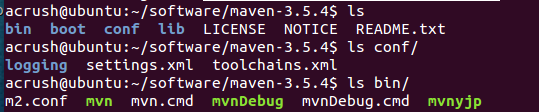
添加maven的环境变量 图

之后在使用maven命令时系统可以快速找到命令，但之前使用source命令使得配置生效。对于windows系统来说，修改环境变量的方式可以在我的电脑中找到。



查看maven版本 图

至此maven已经安装完毕，但是为了更好地使用maven，我们需要对maven的默认配置进行一些修改。首先maven的目录结构如下图所示：

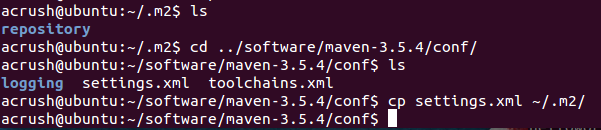


Maven目录结构 图

bin目录下都是一些可执行的脚本，其中cmd格式的是windows环境下的可执行文件，其他的是linux环境下的可执行文件。Lib目录包含了所有的conf目录下的文件都是一些配置文件，其中最为重要的一个配置为文件是settings.xml，修改该文件是在此台电脑上全局地修改maven配置，所以最好不要擅自修改此文件。

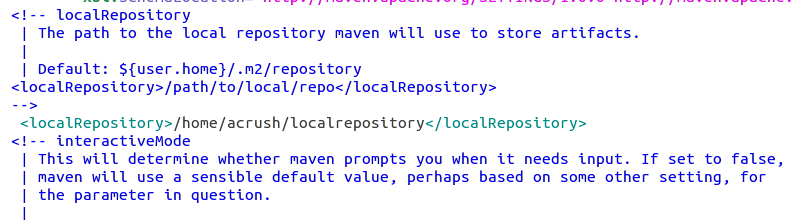
### 1.3.2 修改配置

可以在~/.m2目录下的settings文件中设置我们想要的结果，具体的做法就是复制maven/conf/settings到~/.m2目录下，‘~’是用户家目录，linux中是/home/用户名,在windows中是c\\用户\\用户名。在linux系统中如下复制文件：

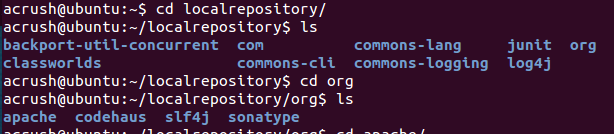


复制配置文件 图

本地仓库。当我们使用maven来构建一个工程时，maven会在本地维护一个本地仓库，如果本次所需的构件在本地仓库中没有找到，maven便会从远程的中央仓库或者私服下载构建到本地仓库中以便下次使用该构件。该本地仓库的默认地址是~/.m2/repository,但是该地址可以在settings中进行修改（使用本用户的settings修改）。修改settings文件实例如下图：

设置本地仓库地址 图

保存设置。在maven的conf目录下的settings是全局地settings，他对于某一项的设置是针对所有用户的，但是每个用户的settings对于同一项的设置能够覆盖。查看本地目录的结构，上面已经将本地仓库的位置设置为自己新建的目录了，那么在首次查看目录时是没有东西的，但是在使用mvn compile命令后，可以查看这个目录的变化，如下图是本地仓库的目录：

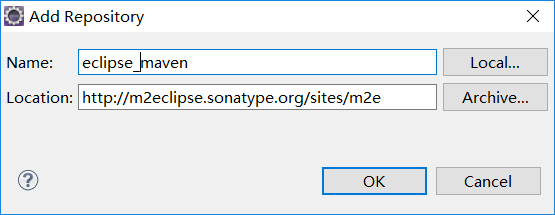


本地仓库的结构 图9

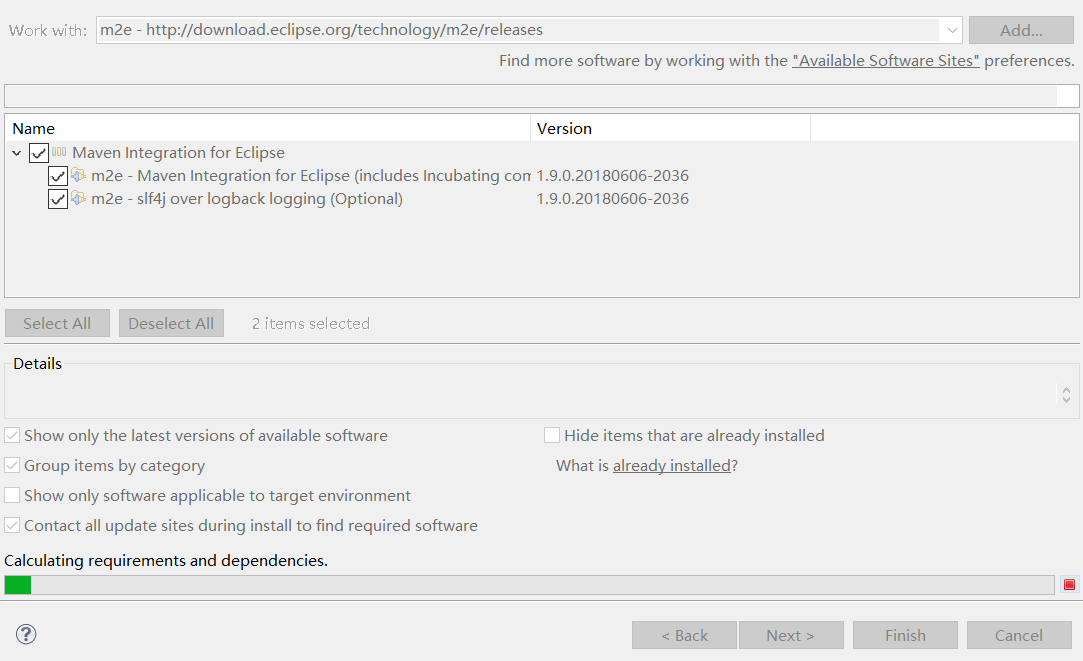
可以看到maven的目录结构是由下载的构件的坐标来创建的，也就是使用groupid+artifactId来确定的

### 1.3.3 安装eclipse的maven插件

Maven提供了eclipse的插件。使得我们可以使用maven构建好一个maven工程之后可以使用eclipse继续进行该工程的开发。Eclipse安装maven的插件非常简单，在eclipse中的help-》install模块就可以安装eclipse的maven插件，m2eclipse。具体过程如下：

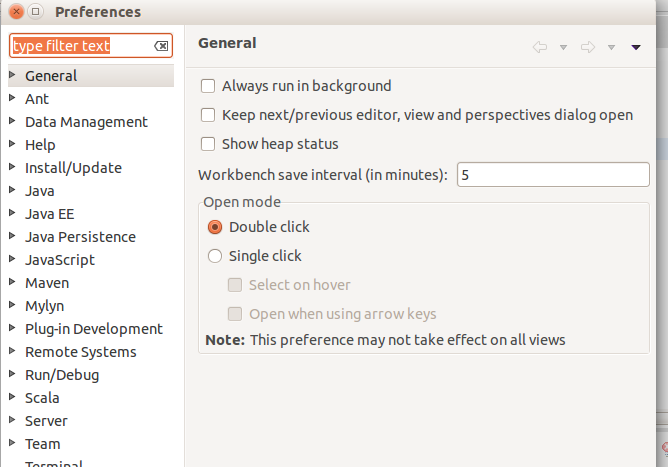


插件网址在线安装 图10



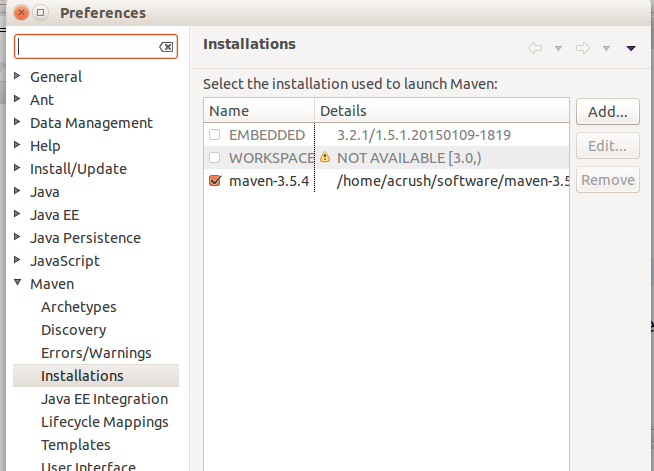
开始安装 图11

安装的下载过程可能是非常慢的，安装完成后，在eclipse的preference中可以看到有maven的这个模块，证明安装成功。如下图：



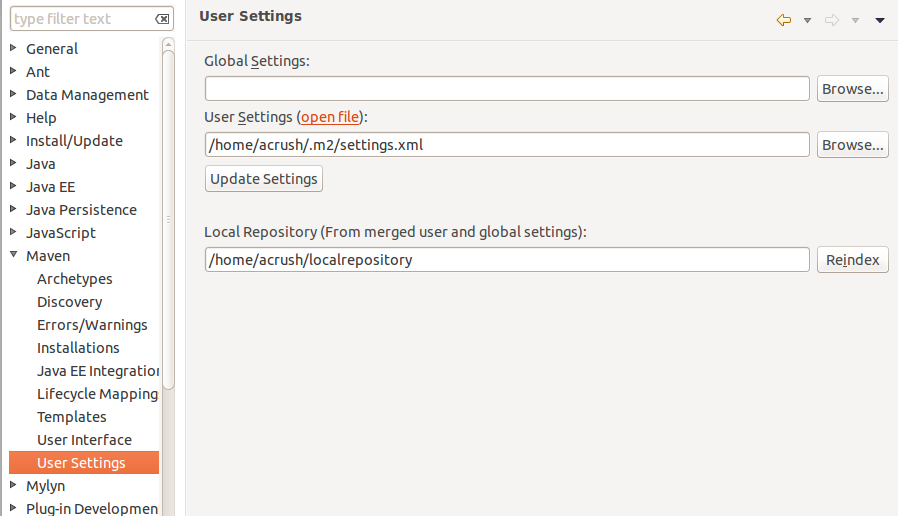
Maven模块安装成功 12

安装好maven的eclipse插件后，这个插件自带了一个maven的内核，但是这个内核是不稳定的会造成构建的不一致所以我们不使用这个maven，使用外面安装的maven，方法就是在maven插件模块中选择本地的maven，如下图：



选用maven 图

除此外，还可以选择maven的settings文件，这个就涉及到全局设置和用户设置，在eclipse中，如下图设置：



设置settings 图

## 1.4问题与分析：

### 1.4.1 关于maven的下载源

# 实验二 手动构建开源项目

|  |  |  |  |
| --- | --- | --- | --- |
| 实验项目 | 实验人员 | 地点 | 日期 |
| 手动构建开源项目 | 李勇 | 4302教室 | 2018/8/31 |

## 2.1 实验目的：

初步了解POM的形式和意义；了解完整的开源项目框架；了解POM中重要的要素及其含义；消除对开源软件项目管理的陌生感。

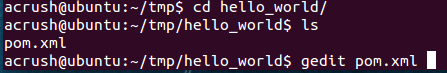
## 2.2 实验内容：

参照课本第三章提供的代码清单完成手动框架搭建。

## 2.3 实验步骤：

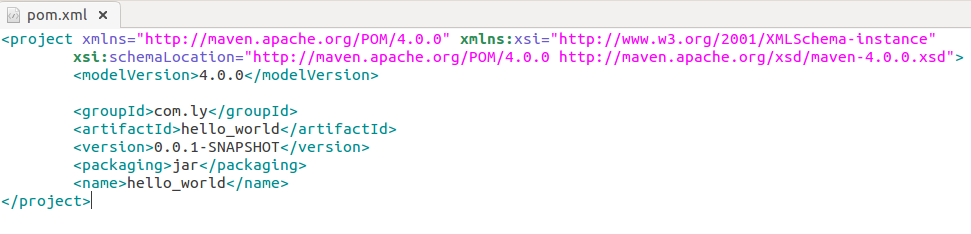
### 2.3.1编写POM

首先我们需要创建一个hello\_world的文件夹，这个文件夹用来创建一个maven工程，所以在这个新建的目录下编辑文件pom.xml，如下图：



创建pom文件 图

Pom文件中描述了文件版本和编码的方式、命名空间和xsd元素。Groupid、artifactId和version三个字段定义了一个项目的基本坐标，在Maven中jar、pom、war都是基于这些坐标区分的。注意这里的artifactId和新建的文件夹的名字要一致，也就是hello\_world。一般来说，groupid表示的是一个公司的某一个项目，groupid表示的是这个项目的工程或者说是子项目。Version指定的是当前的工程的版本，0.0.1-SNAPSHOT表示的是该项目还在开发当中，是不稳定的版本。在项目升级时，只需要修改pom文件就可以了，这保证了pom文件与java文件主代码的“解耦”。Name字段是可选的，主要是为了方便项目内部人员进行交流。具体的pom.xml文件编辑如下图：



Pom.xml文件 图

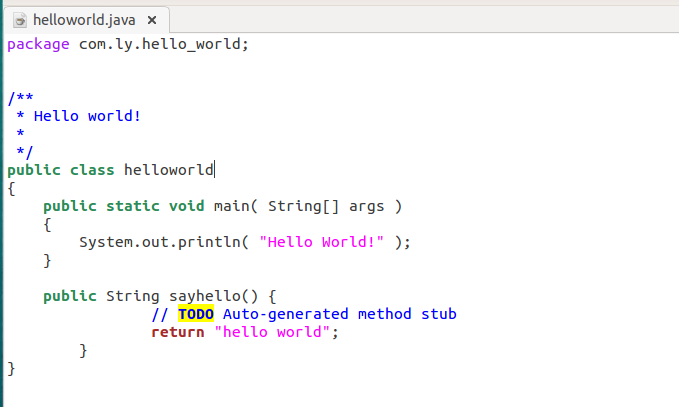
### 2.3.2 编写主代码

一个项目的主代码会被打包到最终的构件中，但是测试代码不会，maven工程的主代码默认在src/main/java目录下，所以我们首先在pom.xml文件所在目录新建目录结构src/main/java/com/ly/hello\_world，后面多加了/com/ly/hello\_world的目录结构，这是因为maven在解析主代码的位置时是根据groupid+artifactId来寻找主代码的，中间的’.’后会被替换为目录符号’\’，所以需要在这样的目录下编辑主代码，如下图所示：



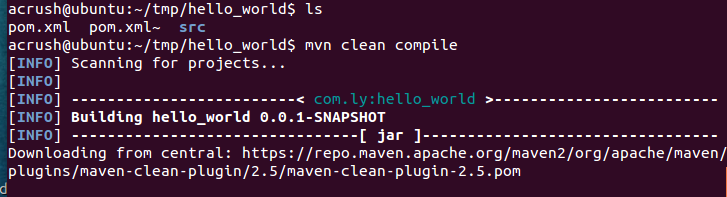
编辑主代码的位置 图

写一个简单的hello world的类，包含主方法。再写一个sayhello的方法，返回一个string。另外注意主代码的包名是groupid+artifactId。对主代码的编辑具体如下图所示：

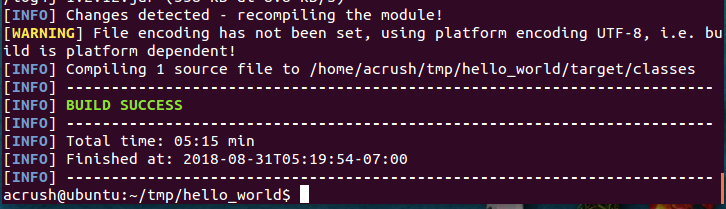


主代码编辑 图

完成编辑后，回到pom文件目录下，运行mvn clean compile。其中clean表示的是清理前一个命令生成的target目录，默认情况下maven构建的所有输出都在target目录下。compile表示的是编译主代码并放置target/classes目录下。运行的过程如下图所示：

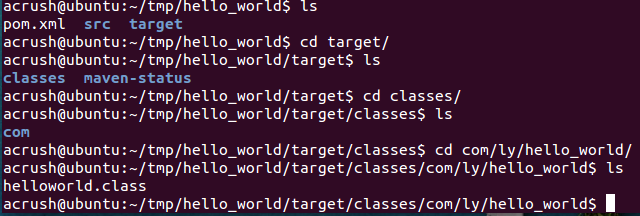


开始编译 图



编译成功 图

编译成功之后，我们来查看一下该工程目录的结构变化以及生成的文件，如下图所示：



编译结果 图

我们可以看到compile之后生成了helloworld.class文件，而在maven-status中生成了关于这次maven编译的状态的文件，包括编译的源文件位置和生成的class文件的位置。

### 2.3.3编写测试代码

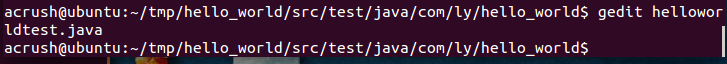
maven可以执行自动化测试，通过自动化测试我们可以知道主代码的一些问题以便做出调整。编写测试代码，需要明确测试代码的应该放置在src\test\java目录下，与主代码类似，我们需要创建的目录结构不只是这样的，我们需要创建的是src/test/java/com/ly/ hello\_world目录结构，原因和主代码类似，maven会在这样的目录下寻找测试代码，但是同时需要构建自己在pom文件中定义的groupid+artifactId标识的路径。因为涉及到测试，所以测试代码需要引用一个叫做Junit的包，我们称之为这个工程的依赖。所以在写测试代码之前先在pom文件中添加这个依赖，具体pom的修改如下图所示：



Pom文件添加依赖 图

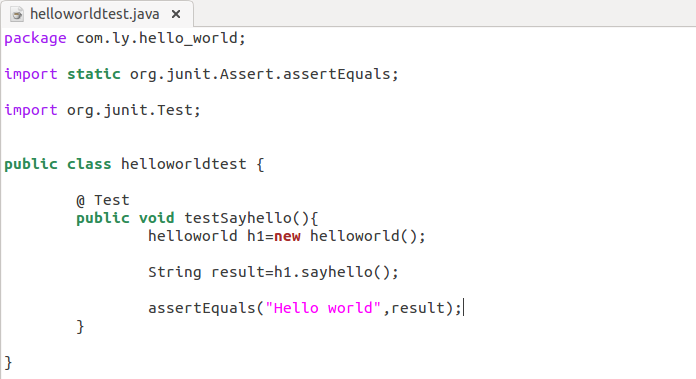
添加依赖的方式很简单，在project里面添加dependecies元素。其中每一个dependency元素可以表示一个依赖。在以后需要到别的依赖可以直接在其中按照groupid+artifactId的方式定位一个依赖，再由version指定依赖的版本，maven就可以自动访问中央仓库或者私服下载需要的文件。Scope元素指的是依赖范围，每个依赖都会有依赖范围，他通过控制classpath来控制依赖在maven的哪一个过程起作用。Scope的默认范围是compile。这里的Junit只需要在测试阶段起作用，所以scope值为test。

有了依赖接下来就可以编写测试类了，在上面新建的目录下编辑测试文件如下所示：



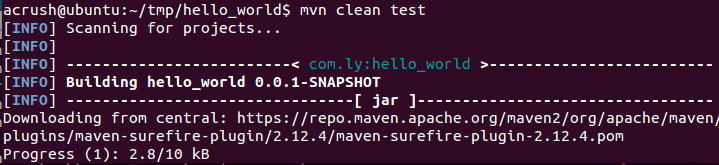
编辑测试文件 图

测试文件的内容可以自己发挥，但是一定要用到主代码并使用assertEquals提交测试的结果，具体的编辑如下所示：

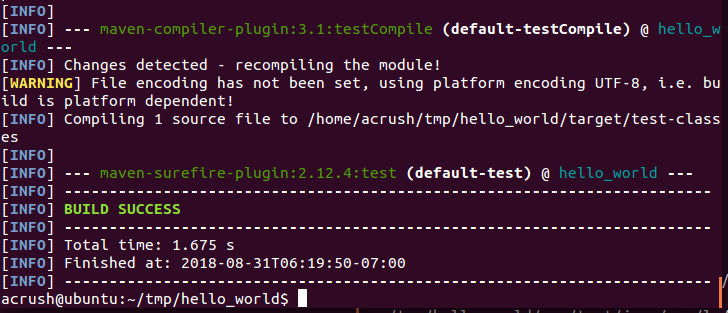


测试代码 图

我们的maven和使用的java版本是相对应的，所以不需要再添加额外的插件。在pom目录下使用mvn clean test进行测试，过程如下图：

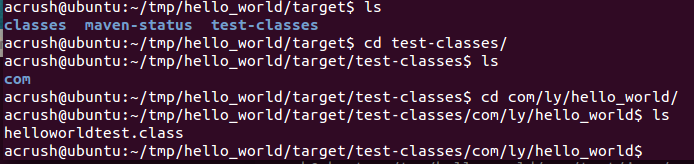


运行测试 图



测试完成 图

测试完成之后观察一下目录结构和生成文件的变化，如下图：

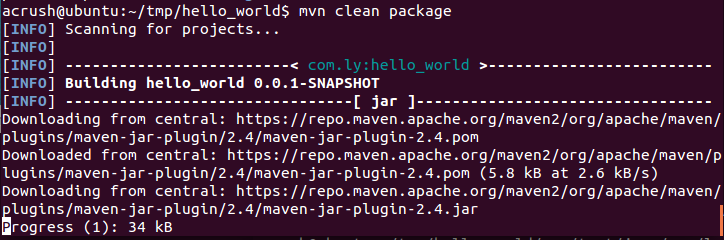


测试结果 图

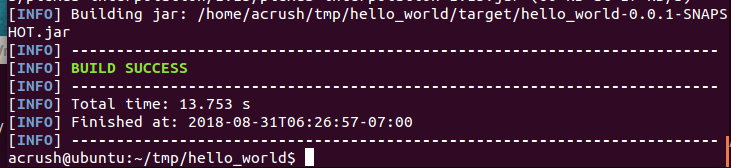
在target目录中生成了test-classes目录用于存放测试代码编译的结果，同时生成的.class文件的位置也是由groupid+artifactId标识的，这样maven可以找到编译的结果。

### 2.3.4 打包和安装

编译、测试完成之后就是打包的过程了，如果pom中没有指定打包的类型，则默认打包为jar。在pom文件所在目录使用mvn clean package 进行打包，过程如下所示:

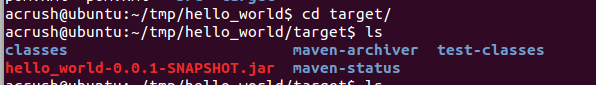


开始打包 图



打包完成 图

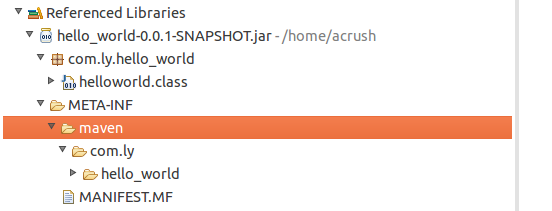
打包完成之后maven会在打包之前编译测试，虽然使用了clean将之前的东西清除了。打包的结果也是存储于target目录下，结果如下所示：



打包结果 图

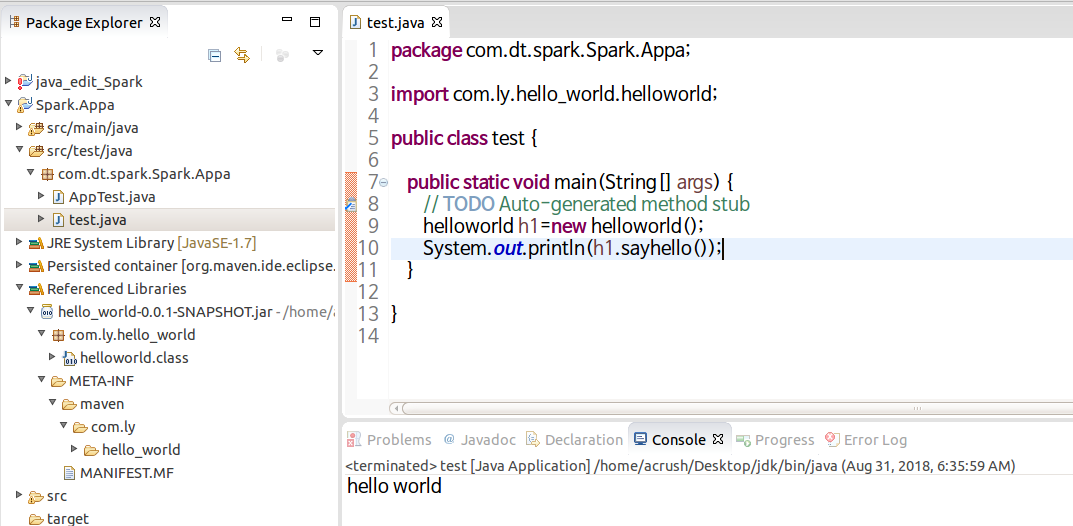
### 2.3.5 使用产生的jar包

得到了编写的主方法的jar包，现在如果要使用这个jar 包，我们可以尝试将其引入eclipse的一个工程中，可以看到jar包包含的目录结构，如下图所示：



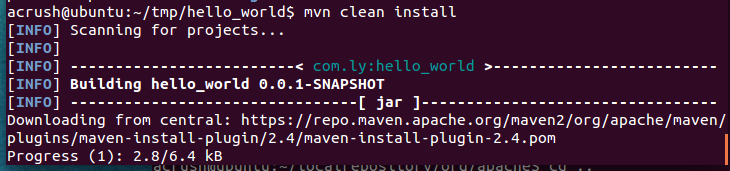
Jar包结构 图

然后可以新建一个类，在类中调用刚才在主类中实现的sayhello方法，运行结果正确。如下图所示：

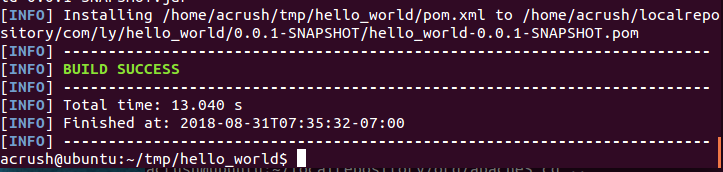


引用jar包方法 图

如果需要将自己的jar包能够被maven项目引用的话，只需要将这个jar包安装到maven中就可。如下图是安装的步骤：

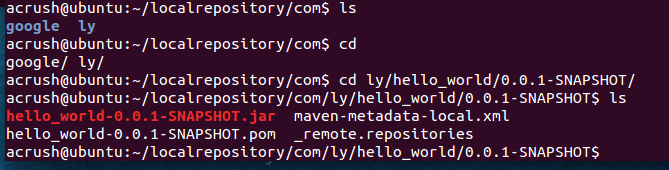


安装自己建立的项目到maven 图



安装成功 图

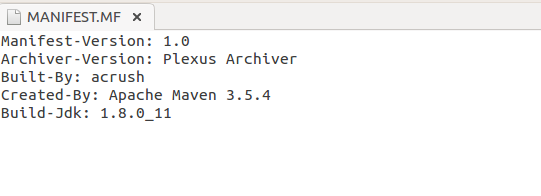
安装成功后可以查看本地的maven仓库，如下图：



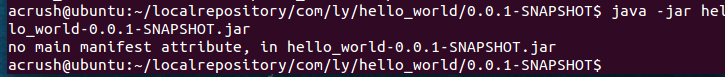
查看本地maven仓库 图

### 2.3.6 运行

虽然得到了一个maven的jar包，但是我们使用java –jar命令无法调用其中的主方法，这是因为默认打包是不会指定主方法的，自动生成的jar文件中的META-INF/MANIFEST.MF文件中没有关于主方法的信息，如下图所示：



Jar包属性信息



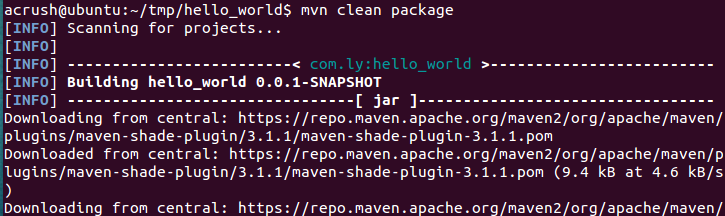
无指定主类jar包 图37

为了生成可运行的jar，需要在pom文件中添加一个插件maven-shade-plugin。插件的配置如下图所示：

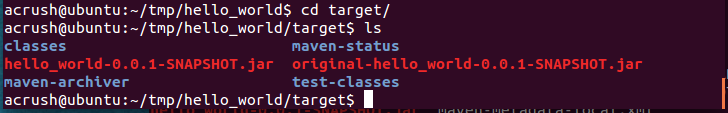


Shade插件 图38

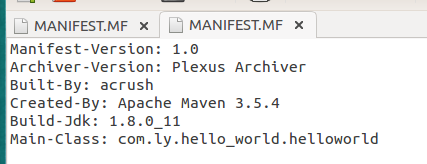
保存后，在pom文件所在目录下运行mvn clean package ，构建完毕后可以在target目录中看到original-hello\_world-0.0.1-SNAPSHOT.jar和hello\_world-0.0.1-SNAPSHOT.jar，前一个是原始的jar包，后一个是可运行的jar包。查看后一个jar包中的META-INF/MANIFEST可以发现指定了主类的完整路径，如下图所示：



构建新jar包 图

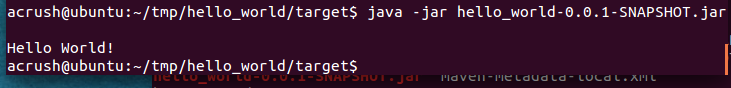


构建完成 图



Jar包信息 图41

然后，执行这个可运行的jar包，如下图所示：



运行jar包 图42

2.4问题与分析：

实验三 使用archetype生成项目骨架

|  |  |  |  |
| --- | --- | --- | --- |
| 实验项目 | 实验人员 | 地点 | 日期 |
| 使用archetype生成项目骨架 | 李勇 | 4302教室 | 2018/8/31 |

3.1 实验目的：

初步了解使用archetype生成项目骨架的流程；对比使用该方法生成的项目骨架与手动生成的项目骨架之间的差别。

3.2 实验内容：

以HelloWorld为例完成archetype项目骨架构建流程。

3.3 实验步骤：

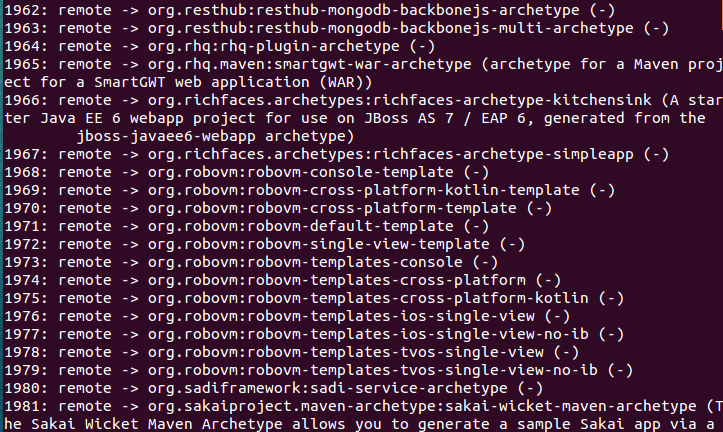
### 3.3.1 创建archetype项目骨架

在一个实验中手动创建的src/main/java…..、src/test/java….和pom文件称这些基本的目录结构和pom.xml文件为项目的骨架。但是手动创建的方式不仅慢，而且浪费了很多时间，使用archertype可以快速构建想要的项目骨架。因为使用的maven3，所以可以使用archetype:generate快速开始构建。过程如下图所示：



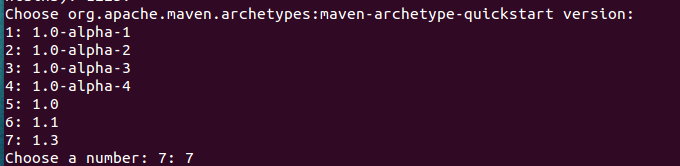
开始构建 图

这个过程可能会下载很多东西。下载完毕后会出现带有编号的一些可用的archetype供选择，如下图所示：



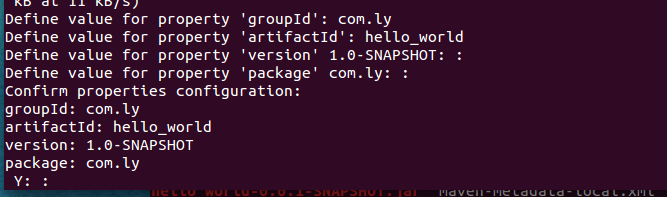
选择archetype图

输入对应的编号即可选择建立相应的archetype骨架，但是如果回车的话选择默认的maven-archetype-quickstart。接下来是选择这个archetype的版本，如下图：



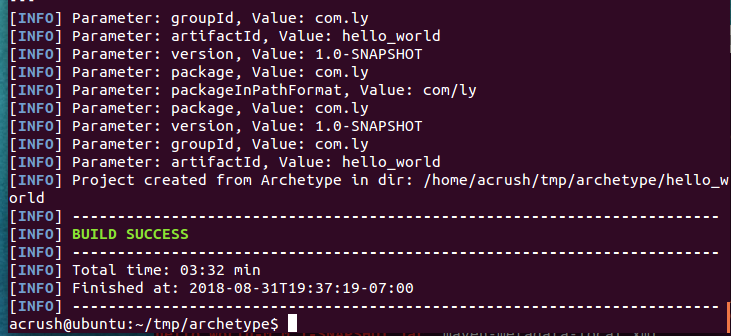
选择版本 图

然后是定义坐标元素的值，包括groupid、artifactid、version以及包名，如下图：



定义坐标元素 图

Version以及包名都可以使用默认的名称就好了，这里主要是定义基本的坐标元素。构建成功后会提示生成的目录位置，如下图：



构建完成 图47

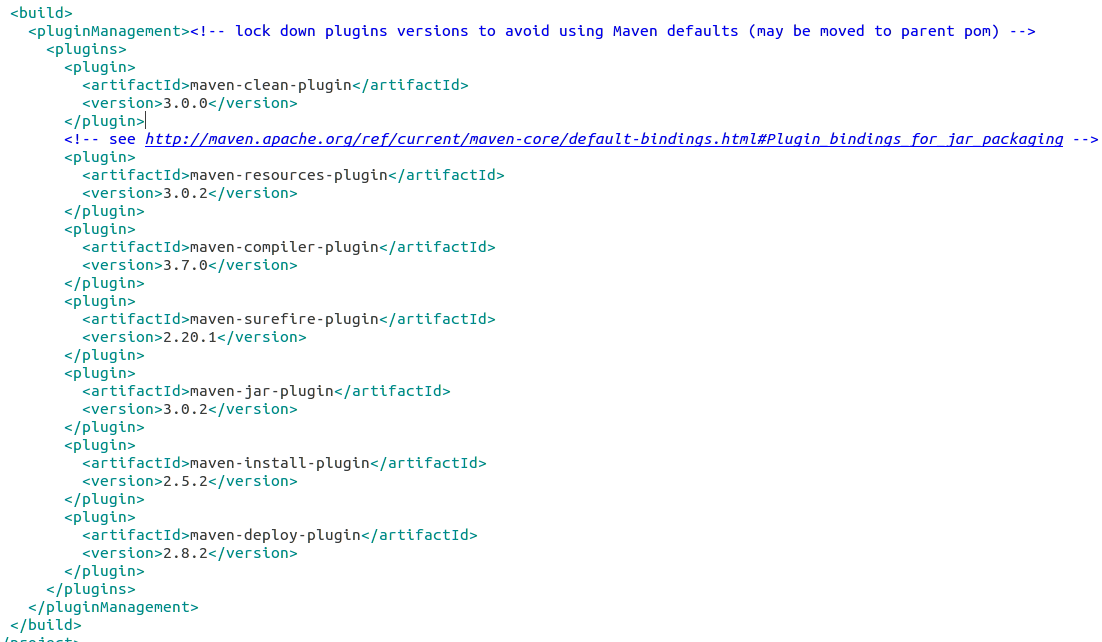
构建完成后，maven会在命令运行的路径下生成一个hello\_world的工程目录，其中还包含了pom文件，如下图：



查看生成的文件目录 图

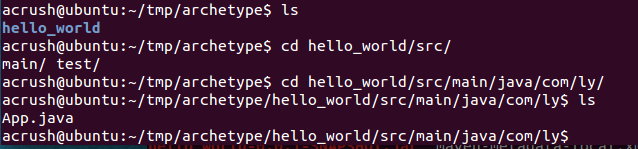
首先来查看一下生成的pom.xml文件，其内容如下图所示：





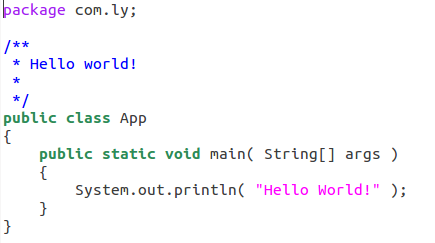
Pom文件内容 图49

在自动生成的pom文件中，定义了编码的格式是UTF-8，指定了compile的版本是1.7，其次这个pom已经定义加载了很多的依赖，包括maven编译、install、deploy、surrfire等过程所需的插件。这比我们手动创建要快的多。接下来查看src目录，如下图所示：



自动生成的主代码位置

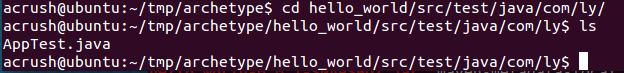
Src目录和手动创建的无异，但是自行观察可以发现在groupid路径之后没有了artifactid标识的一级目录，直接就是生成的主代码App.java，查看App.java的内容如下图所示：

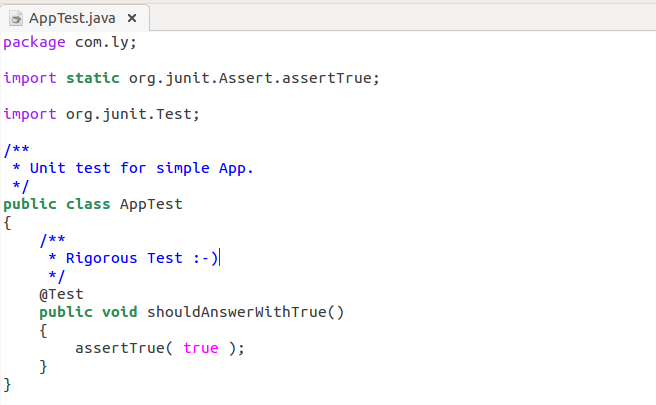


App.java内容 图

注意App.java的包是com.ly,这个包名和外面的路径也是对应起来了。因为我们在创建时选择了默认的包名com.ly，所以这里的目录也就没有涉及到了artifactid。

同样测试代码的目录位置也是没有artifactid标识的，只有com/ly目录，其内容如下所示：





Test代码内容 图

这个测试方式还没有真正实现，需要自己进行编写。

和手动构建相比，使用archetype构建maven骨架会比较快速和便捷，同时使用archetype构建的maven工程会自动添加很多需要的依赖，在自动生成的骨架上，我们可以继续开发，此外archetype提供的多达2215种maven骨架的自动建立。

3.4问题与分析：