[**Maven学习第4期---Maven简单使用**](http://www.cnblogs.com/sunddenly/p/4341542.html)

<http://www.cnblogs.com/sunddenly/p/4341542.html>

# 一、Maven常用命令

在学习使用Maven构建项目之前，先来了解一下Maven一些实用的命令。mvn的命令有很多，在项目开发中，理解了下面的几个常用命令后，运用maven就基本没有问题了。Maven常用的命令如下：

## 1.1 常用的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，使得其他的开发者或者工程可以共享。

## 1.2 一些高级功能命令

跳过测试类 ： -Dmaven.test.skip=true

下载jar包源码： -DdownloadSource=true

下载javadocs： -DdownloadJavadocs=true

例如：

mvn package -Dmaven.test.skip=true ：表示打包时忽略掉test目录，不编译。   
mvn install -DdownloadSource=true ：本地安装jar包时，下载jar的源文件包。

# 二、Maven简单使用

到目前为止，我们已经大概了解并安装好了Maven。为了进一步学习Maven，我们和其他项目同样以一个最简单的HelloWorld项目开始，探究Maven的核心概念。通过对该项目的学习，来让我们对构建生命周期 (Build Life Cycle)，Maven仓库 (Repositories)，依赖管理 (Dependency Management)和项目对象模型 (Project Object Model)有一个基本的理解。

## 2.1 编写POM

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。现在先为HelloWorld项目编写一个最简单的pom. xml，

首先创建一个名为helloworld的文件夹，打开该文件夹，新建一个名为pom.xml的文件，输入其内容，如代码清单如下所示：

<?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>org.hebut.test</groupId>

    <artifactId>helloworld</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>Maven Hello World Project</name>

</project>

**XML头**

代码的第一行是XML头，指定了该xml文档的版本和编码方式。

**project元素**

XML头之后紧接着就是project元素，project是所有pom.xml的根元素，它还声明了一些POM相关的命名空间及xsd元素，虽然这些属性不是必须的，但使用这些属性能够让第三方工具（如IDE中的XMl编辑器）帮助我们快速编辑POM 。

**modelVersion元素**

根元素下的第一个子元素modelVersion，它指定了当前POM模型的版本，对于Maven 3以及Maven 3来说，它只能是4.0.0。

**坐标**

这段代码中最重要的是：包含groupId、artifactId和version的三行。这三个元素定义了一个项目基本的坐标，在Maven的世界，任何的jar、pom或者war都是以基于这些基本的坐标进行区分的。

**groupId元素**

groupId定义了项目属于哪个组，这个组往往和项目所在的组织或公司存在关联。譬如在googlecode上建立了一个名为myapp的项目，那么groupId就应该是com.googlecod.myapp，如果你的公司是mycom，有一个项目为myapp。耶么groupId就应该是com.mycom.myapp。

**artifactId元素**

artifactId定义了当前Maven项目在组中唯一的ID，我们为这个HelloWord项目定义artifactId为hello-world。在前面的groupld为

com.googlecode.myapp的例子中，你可能会为不同的子项目（模块）分配artifactId，如myapp-util、myapp-domain、myapp-web等。

**version元素**

顾名思义，version指定了Hello World项目当前的版本0.0.1。SNAPSHOT意为快照，说明该项目还处于开发中，是不稳定的版本。随着项目的展，version会不断更新，如升级为0.0.2、0.0.3、1.0.0等。

**name元素**

最后一个name元素，声明了一个对于用户更为友好的项目名称，虽然这不是必须的，但还是推荐为每个POM声明name。以方便信息交流。

没有任何实际的Java代码，我们就能够定义一个Maven项目的POM，这体现了Maven的一大优点，它能让项目对象模型最大程度地与实际代码相独立，我们可以称之为解耦，或者正交性。这在很大程度上避免了Java代码和POM代码的相互影响：比如当项目需要升级版本时，只需要修改POM。而不需要更改Java代码；而在POM稳定之后，日常的Java代码开发工作基本不涉及POM的修改。

## 2.2 编写主代码

项目主代码和测试代码不同，项目的主代码会被打包到最终的构件中如：jar。而测试代码只在运行测试时用到，不会被打包。默认情况下，Maven假设项目主代码位于src/main/java目录，我们遵循Maven的约定，创建该目录，然后在该目录下创建文件org/hebut/test/helloworld/HelloWorld. java，其内容如代码如下所示：

**package** org.hebut.test.helloworld

**public** **class** HelloWorld{

**public** String sayHello(){

**return** "Hello Maven";

    }

**public** **static** **void** main(String[] args){

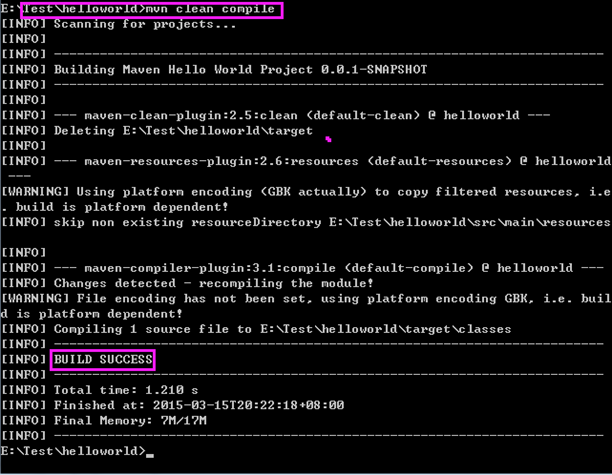
        System.out.println(**new** HelloWorld());

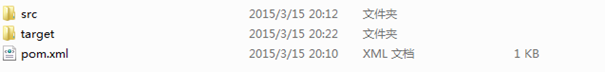
    }

}

这是一个简单的Java类，它有一个sayHello()方法，返回一个String同时这个类还带有一个main方法，创建一个HelloWorld实例，调用sayHello()方法，并将结果输出到控制台。

关于该Java代码有两点需要注意：首先，在绝大多数情况下，应该把项目主代码放到src/main/java/目录下，而无须额外的配置，Maven会自动搜寻该目录找到项目主代码。其次， 该Java类的包名是org.hebut.test.helloworld，这与之前在POM中定义的groupId和artifactld相吻合。一般 来说，项目中Java类的包都应该基于项目的groupld和anifactId。这样更加清晰，更加符合逻辑，也方便搜索构件或者Java类。

代码编写完毕后，使用Maven进行编译，在项目的根目录下运行命令mvn clean compile会得到如下输出: 执行完毕，发现项目根目录多了一个target目录：



上面使用的clean命令告诉Maven清理输出目录target/，compile告诉Maven编译项目主代码，从输出中看到Maven**首先**执行了clean：clean任务，删除target/目录。默认情况下，Maven构建的所有输出都在target/目录中；**接着**执行resources：resources任务，未定义项目资源，暂且略过；

**最后**执行compiler：compile任务，将项目主代码编译至target/classes目录，编译好的类为：

org/hebut/test/helloworld/HelloWorld.Class

上文提到的clean:clean、resources:resources和compiler:compile对应了一些Maven插件及插件目标，比如clean:clean是clean插件的clean目标，compiler:compile是compiler插件的compile目。

至此，Maven在没有任何额外的配置的情况下就执行了项目的清理和编译任务。接下来，编写一些单元测试代码并让Maven执行自动化测试。

## 2.3 编写测试代码

为了使项目结构保持清晰，主代码与测试代码应陔分别位于独立的目录中。正如上面所述，Maven项目中默认的主代码目录是src/main/java。对应地，Maven项目中默认的测试代码目录是src/test/java。因此，在编写测试用例之前，应当先创建该目录。

### 2.3.1 设置依赖

在Java世界中，由Kent Beck和Erich Gamma建立的JUnit是事实上的单元测试标准。要使用JUnit，首先需要为HelloWorld项目添加一个JUnit依赖，修改项目的POM代码如下：

<?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>org.hebut.test</groupId>

    <artifactId>helloworld</artifactId>

    <version>0.0.1-SNAPSHOT</version>

<name>Maven Hello World Project</name>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.7</version>

<scope>test</scope>

</dependency>

    </dependencies>

</project>

**dependencies元素**

代码中添加了dependencies元素，该元素下可以包含多个dependency元素以声明项目的依赖。

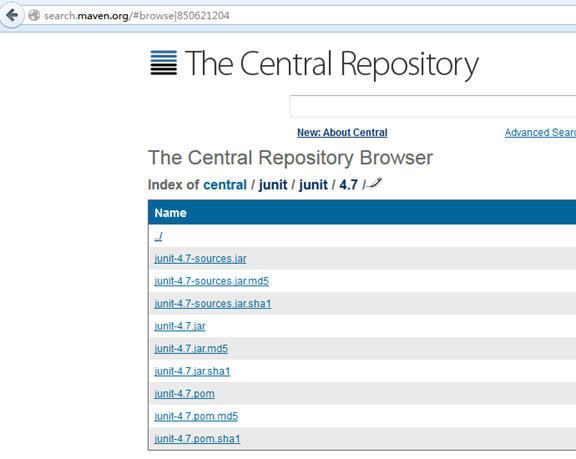
**dependency元素**

dependency元素用以声明项目的依赖，这里添加了一个依赖 groupId是junit，artifactld是junit，version是4.7。前面提到groupId、artifactId和 versIon是任何一个Maven项目最基本的坐标。JUnit也不例外，有了这段声明Maven就能够自动下载junit-4.7.jar。那 么，Maven从哪里下载这个jar 呢?

在使用Maven之前，可以去JUnit的官方网站下载分发包。有了Maven之后，它会自动访问中央仓库：

<http://repo1.maven.org/maven2/>,下载需要的文件。

当然与可以自己访问该仓库，打开路径junit/junit/4. 7/，就能看到junit-4.7.pom和junit-4.7.jar。



**scope元素**

上述POM代码中还有一个值为test的元素scope，scope为依赖范围，若依赖范围为test则表示该依赖只对测试有效。换句话说，测试代码中的import JUnit代码是没有问题的，但是如果在主代码中用import Junit代码，就会造成编译错误。如果不声明依赖范围，那么默认值就是compile，表示该依赖对主代码和测试代码都有效。

### 2.3.2 编写测试代码

配置了测试依赖，接着就可以编写测试类。回顾一下前面的HelloWorld类，现在要测试该类的sayHello()方法，检查其返同值是否为"Hello Maven"。在src/test/java目录下创建文件，其内容代码如下所示：

**package** org.hebut.test.helloworld;

**import** org.junit.Test;

**import** **static** org.junit.Assert.assertEquals;

**public** **class** HelloWorldTest{

    @Test

**public** **void** testSayHello(){

        HelloWorld helloworld=**new** HelloWorld();

        String result=helloworld.sayHello();

        assertEquals("Hello Maven",result);

    }

}

一个典型的单元测试包含三个步骤：

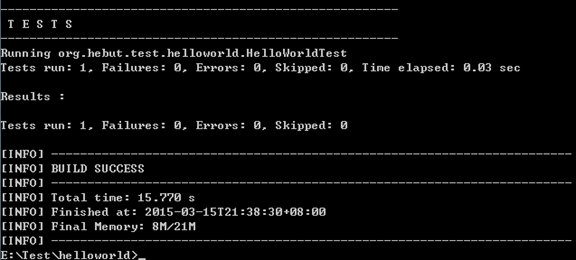
**①** 准备测试类及数据；

**②**执行要测试的行为；

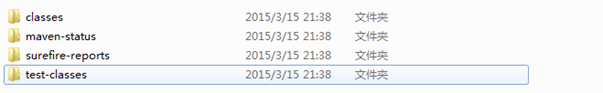
**③** 检查结果。

上述样例，**首先**初始化了一个要测试的HelloWorld实例，**接着**执行该实例的sayHello()方法并保存结果到result变量中，**最后**使用JUnit框架的Assert类检查结果是否为我们期望的"Hello Maven"。在JUnit 3中，约定所有需要执行测试的方法都以test开头，这里使用了JUnit 4，但仍然遵循这一约定。在JUnit 4中，需要执行的测试方法都应该以@Test进行标注。

测试用例编写完毕之后就可以调用Maven执行测试。运行mvn clean test：



我们看到compiler: testCompile任务执行成功了，测试代码通过编译之后在target/test-classes下生成了二进制文件:

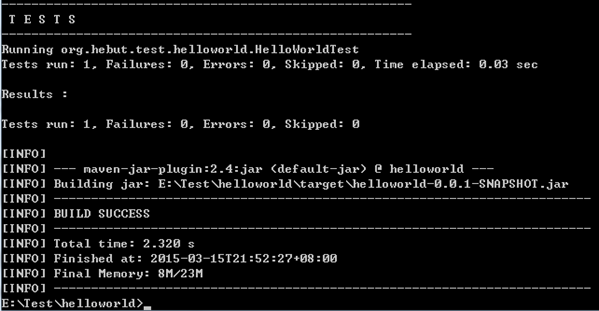


紧接着surefire：test任务运行测试，surefire是Maven中负责执行测试的插件，这里它运行测试用例HelloWorldTest，并且输出测试报告，显示一共运行了多少测试，失败了多少，出错了多少，跳过了多少二硅然，我们的测试通过了。

## 2.4 打包安装

将项目进行编译、测试之后，下一个重要步骤就是打包:package。 Hello WorldPOM中没有指定打包类型，使用默认打包类型jar。简单地执行命令：

mvn clean package进行打包，可以看到如下输出：



类似地，Maven会在打包之前执行编译、测试等操作，这里看到jar: jar任务负责打包，实际上就是jar插件的jar目标将项目主代码打包成一个名为helloworld-0.0.1-SNAPSHOT.jar的文件。该文件也位于target/输出目录中，它是根据artifact-version. jar规则进行命名的，如有需要，还可以使用finalName来自定义该文件的名称。

至此，我们得到了项目的输出，如果有需要的话，就可以复制这个jar文件到其他项目的Classpath中从而使用HelloVV orld类。但是，如何才能让其他的Maven项目直接引用这个jar呢？还需要一个安装的步骤，执行mvn clean install:

