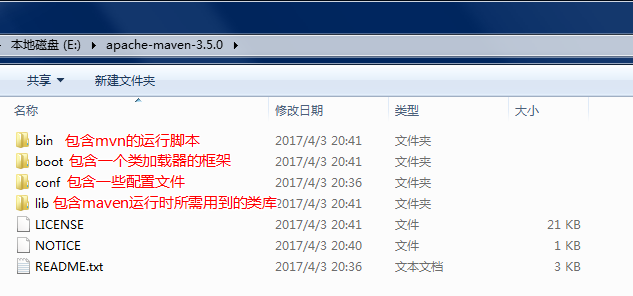
1. **maven介绍及环境搭建**
2. maven下载后的目录介绍



1. 配置环境变量

将下载好的maven中bin文件的位置添加到Path中。即：E:\apache-maven-3.5.0\bin。然后，在命令行中用mvn –v来检验是否配置好。

1. maven项目的文件结构

src

-main

-java

-resources

-test

-java

-resources

1. 第一个maven程序

（1）在code文件夹下面建立一个maven01文件夹，maven01文件夹用于存放本次程序的代码。

1. 先建立src目录
2. src目录下面有main和test两个目录
3. main目录下根据包名com.hik.maven01.model建立文件夹，main中的Java程序为HelloWorld.java。
4. test目录下根据包名com.hik.maven01.model建立文件夹，test中的测试程序为HelloWorldTest.java。
5. 在src根目录下创建pom.xml文件。

（2）下面是HelloWorld.java程序：

package com.hik.maven01.model;

public class HelloWorld {

public String sayHello() {

return “Hello World”;

}

}

（3）下面是HelloWorldTest.java的程序：

Package com.hik.maven01.model;

import org.junit.\*;

import org.junit.Assert.\*;

public class HelloWorldTest {

@Test

public void testHello() {

Assert.assertEquals(“Hello World”, new HelloWorld().sayHello());

}

}

（4）下面是pom.xml的程序：

<?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">

<!-- maven的版本 -->

<modelVersion>4.0.0</modelVersion>

<groupId>com.hik.maven01</groupId>

<artifactId>maven01-model</actifactid>

<version>0.0.1SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</actifactid>

<version>4.11</version>

<depdendency>

</dependencies>

</project>

（5）程序写好了之后，在工程根目录maven01打开命令行：

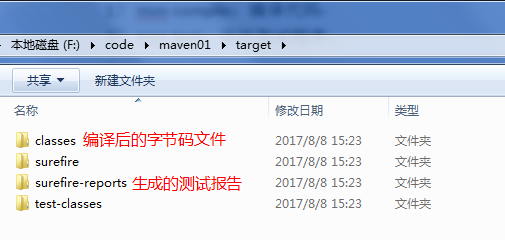
1. mvn compile：编译代码
2. mvn test：运行测试程序

**PS：在公司的电脑上，去网上下载的maven3.5.0不能完成mvn test命令，但是在hik网上下载的maven3.0.5就能顺利完成测试。**

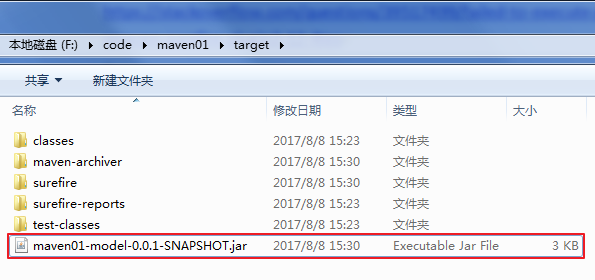
**报错为：应该是相关jar包没下载完成。垃圾网速。**<https://stackoverflow.com/questions/39517499/failed-to-execute-goal-org-apache-maven-pluginsmaven-surefire-plugin2-12-4tes>

<https://stackoverflow.com/questions/13170860/failed-to-execute-goal-org-apache-maven-pluginsmaven-surefire-plugin2-10test>

（6）编译以及运行后，会在根目录maven01下增多一个target目录。



1. mvn package命令：将项目打包成jar包。



1. **maven常用构建命令**

mvn –v：检查版本号

mvn compile：编译

mvn test：测试

mvn package：打包

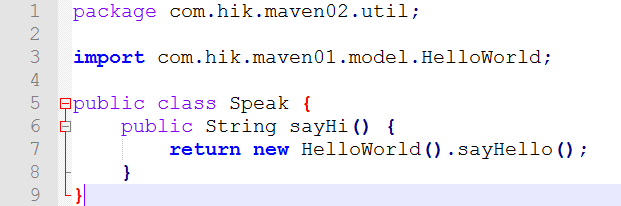
mvn clean：删除target目录

mvn install：安装jar包到本地仓库中

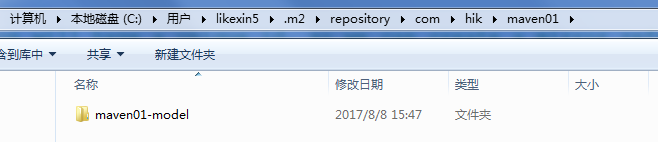
下面重点介绍mvn install命令：

新建一个maven02项目，其中代码引用到了maven01这个工程，这个时候直接编译maven02是行不通的，必须将maven01引入到maven02中。

下面是maven02的java代码：



1. 首先在maven01的目录下执行mvn install命令将其安装到本地仓库。本地仓库默认的位置如下图所示。



1. 然后在maven02项目的pom.xml中添加maven01的坐标。



1. 最后在maven02项目中执行mvn compile命令，编译成功。

**对于maven来说引入构建包的流程总结：**

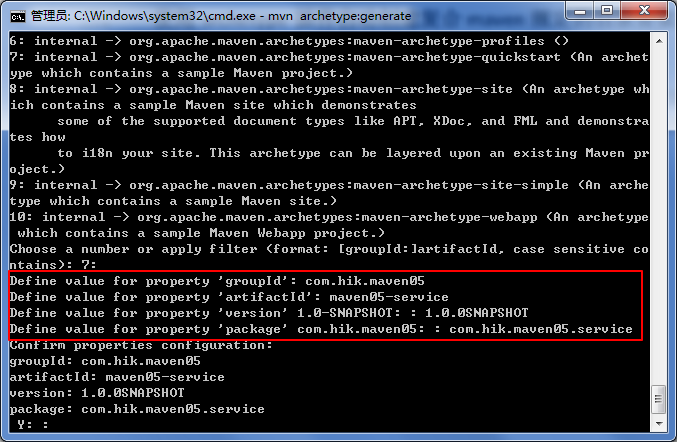
1. 首先通过mvn compile来对源代码进行编译时，如果在编译的过程中程序用到了其他的依赖包，那么它会去pom.xml中查找，是否有该依赖包的坐标。
2. 引入了坐标之后，它将会去本地仓库中查找，如果有这些jar包的话，会将该jar包加入到classpath中。
3. 如果本地仓库中没有这些jar包的话，将会去maven的中央仓库中下载并将其放入到本地仓库中，供我们的项目使用。
4. **使用archetype插件自动创建复合maven规定的目录骨架**

创建目录的两种方式：

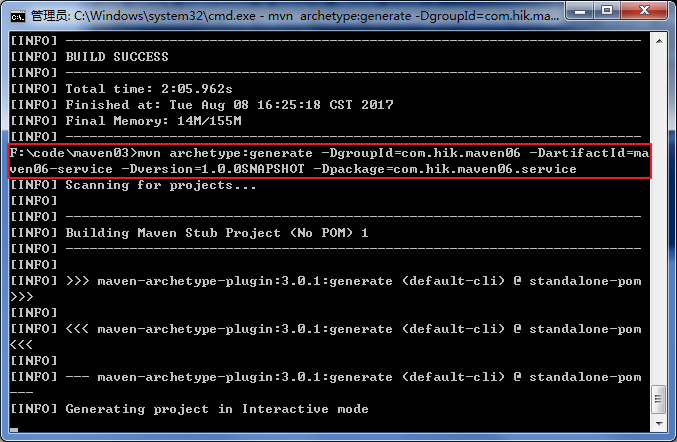
1. mvn archetype:generate：按照提示进行选择
2. mvn archetype:generate –DgroupId=组织名，公司网址反写+项目名 –DartifactId=项目名-模块名 –Dversion=版本号 –Dpackage=代码所在的包

例如：

使用第一种方式mvn archetype:generate创建:



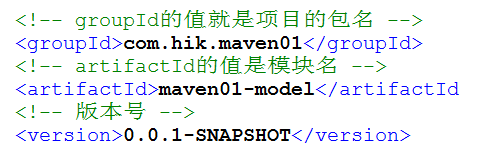
使用第二种方式创建：



1. **maven中的坐标和仓库**
2. 坐标

在maven的世界中，任何一个依赖、插件、项目构建的输出都被称为构件，所有构件都通过坐标作为其唯一标识。

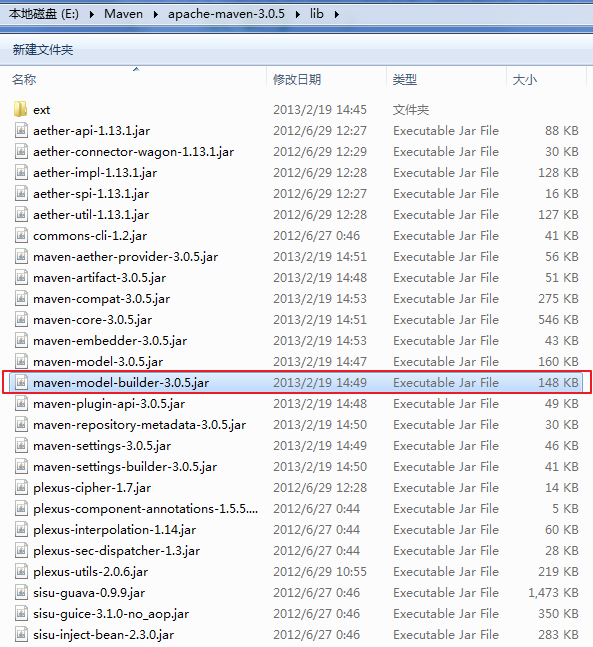
坐标的构成：groupId+artifactId+version。

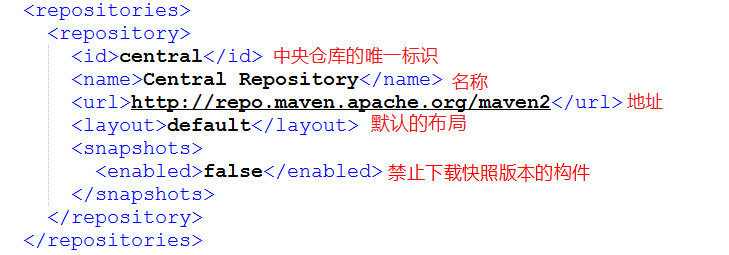


1. 仓库

仓库是用来管理项目依赖的，仓库分为本地仓库和远程仓库。如果在本地仓库不能找到我们需要的构件，那么maven将会去远程的中央仓库( <http://repo.maven.apache.org/maven2/> )中查找，查找到了之后会下载到我们的本地仓库中。如果中央仓库中也找不到构件，就将报错。

中央仓库的地址：在maven的lib目录下，maven-model-builder-3.0.5.jar包中的pom-4.0.0.xml文件。这个pom文件是所有maven项目的pom文件都继承的pom。





1. 镜像仓库
2. 更改仓库位置

默认的本地仓库位于C盘中，但不推荐这么做。因为如果遇到重装系统什么的，这些数据就没了。

更改方法：

1. 在maven下载目录的conf目录下，编辑setting.xml，将localRepository这一项改为自己仓库的位置。记得将斜杠改反一下。(E:\repo –> E:/repo)
2. 在自己仓库的位置再复制一份setting.xml过去。

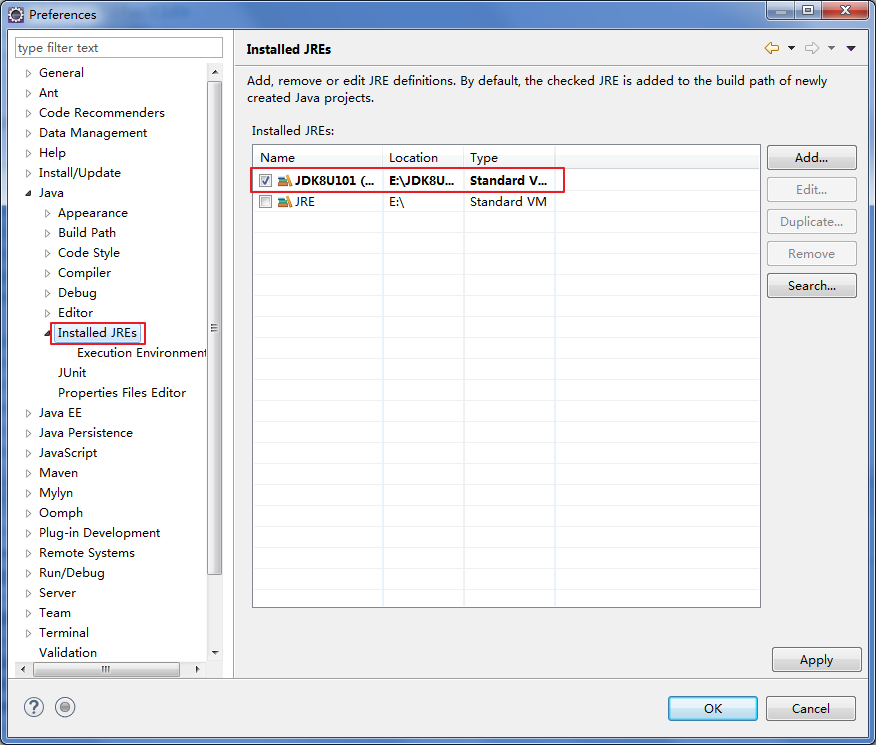
检验仓库位置是否修改成功：

检验之前随便一个工程，执行 mvn clean compile命令，看是否重新下载插件和第三方类库，并观察repo目录下是否增多了东西。

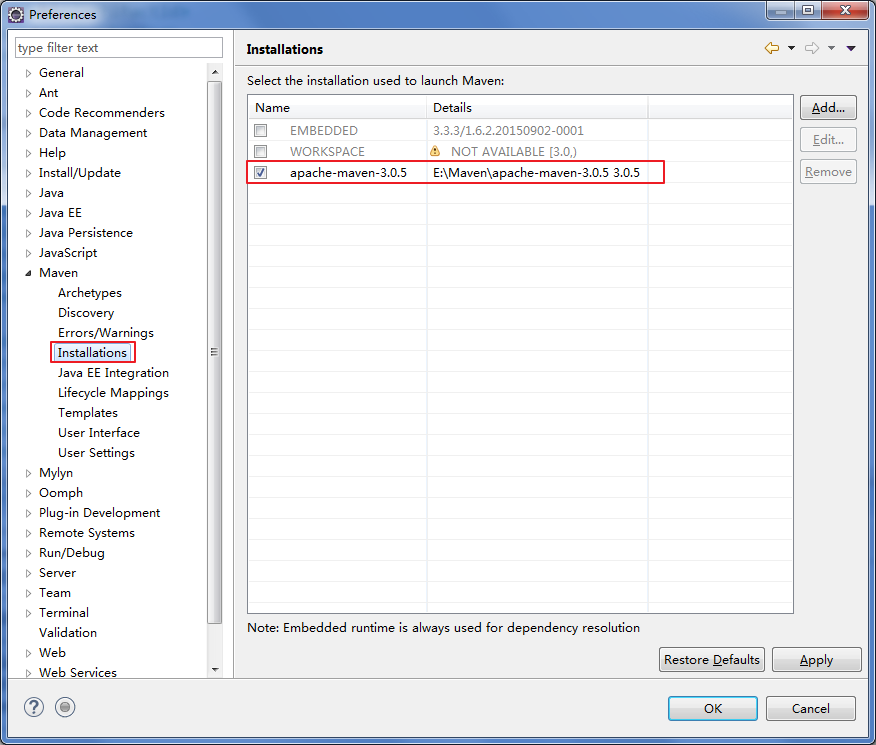
1. **在Eclipse中创建maven项目**

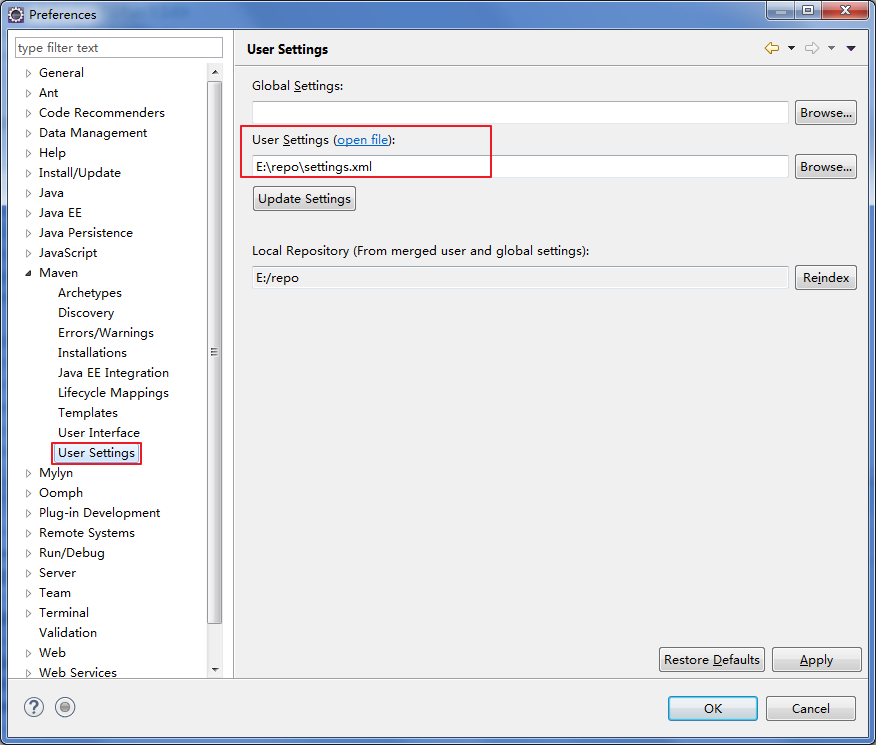
折腾了好久，总的来说做了一下三个更改：

1. 更改jdk为本地jdk1.8（也可以在pom.xml中更改



1. 更改maven为本地maven版本，并更改了setting.xml的位置。





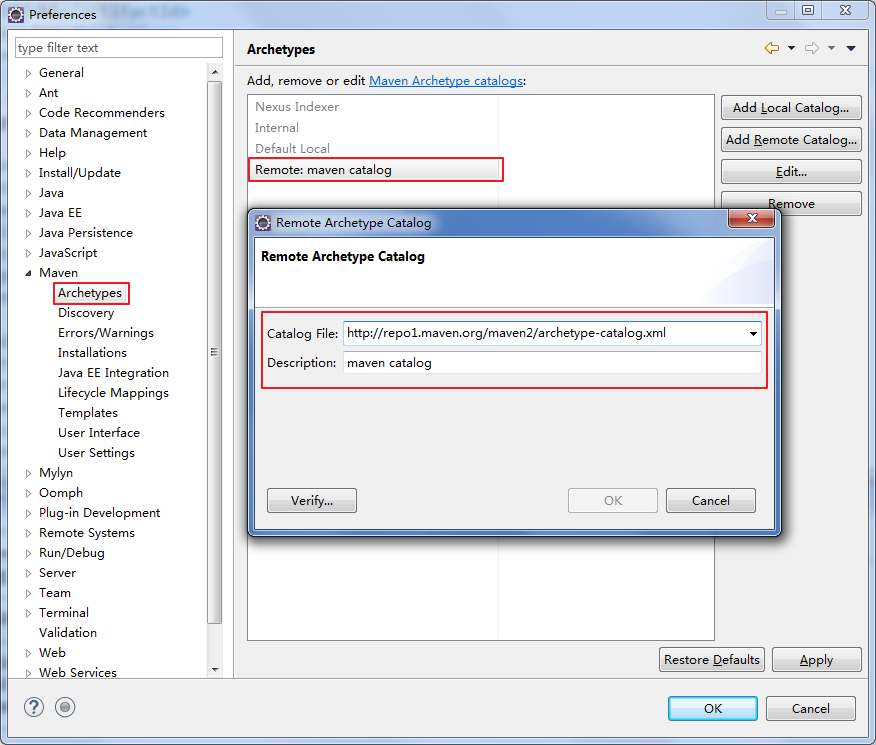
1. 更改了maven archetype为remote catalog，因为报了个错。

报错为：could not resolve archetype org.apache.maven.archetypes:maven-archetype-weba

处理方法参考于：<http://blog.csdn.net/afgasdg/article/details/12757433/>

**PS：后来没有用到这个remote catalog。**

有人说是mirror的问题~



1. **maven的生命周期和插件**

一个完整的项目构建过程包括：清理、编译、测试、打包、集成测试、验证、部署等步骤。也差不多对应了之前的clean、compile、test、package、install等命令。

maven中定义了三套相互独立的生命周期：

1. clean：清理项目
   1. pre-clean：执行清理前的工作
   2. clean：清理上一次构建生成的所有文件
   3. post-clean：执行清理后的文件
2. default：构建项目
   1. compile
   2. test
   3. package
   4. install
   5. 等等…
3. site：生成项目站点
   1. pre-site：在生成项目站点前要完成的工作
   2. site：生成项目的站点文档
   3. post-site：在生成项目站点后要完成的工作
   4. site-deploy：发布生成的站点到服务器

一个例子：在eclipse中执行source插件。



然后执行clean package命令后，在控制台将看到其完成了compile -> test -> package -> source的过程。说明了这些插件是依次执行的。

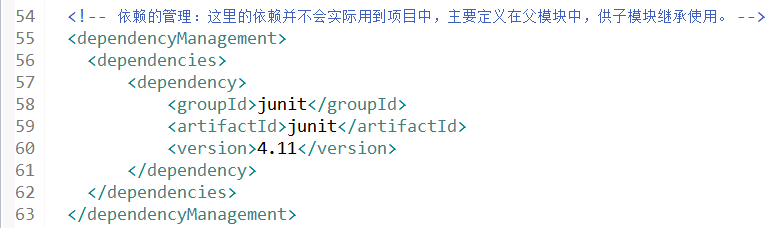
1. **pom.xml常用元素**
2. 基础配置



1. 依赖列表



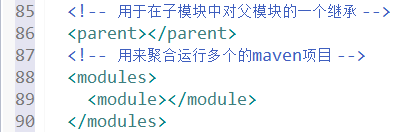
1. 依赖管理



1. 插件列表

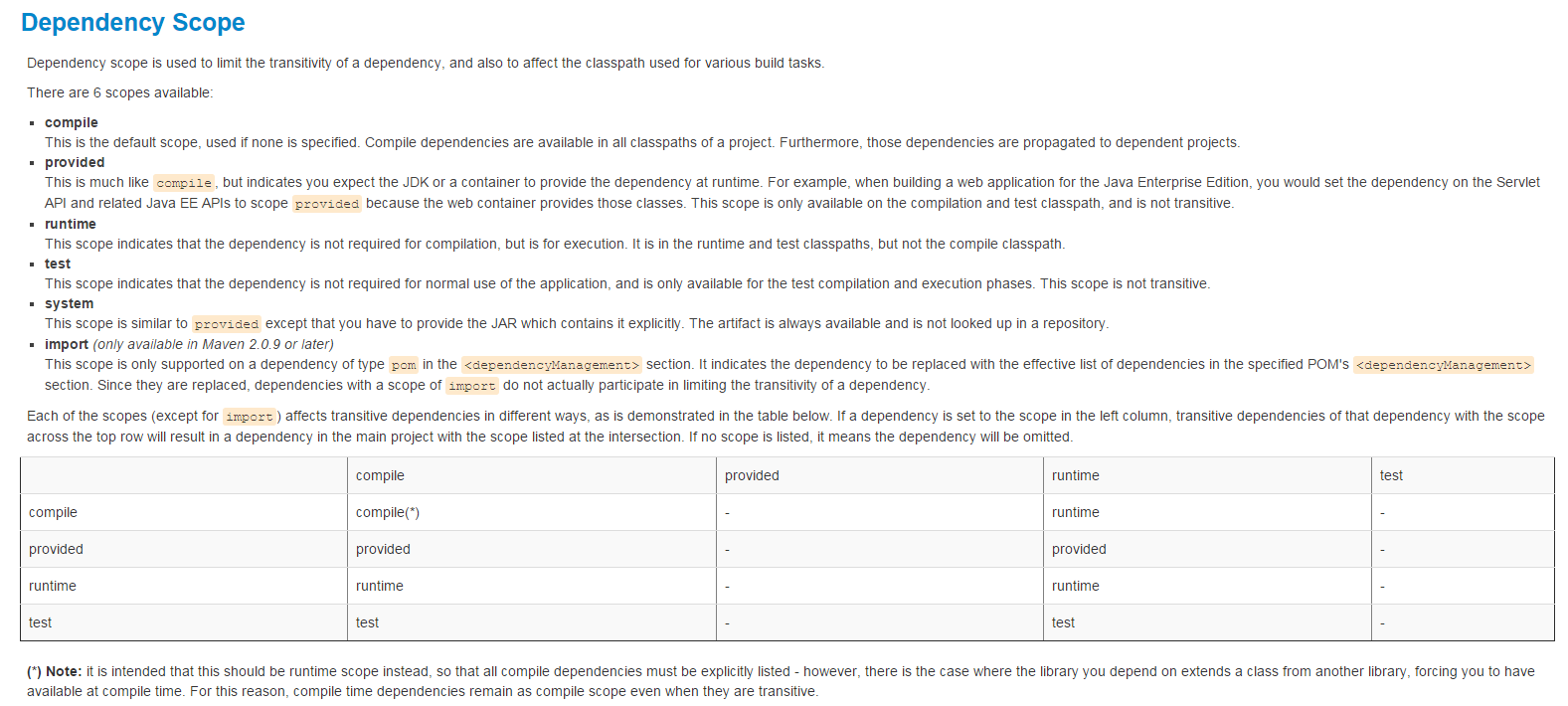


1. parent and modules



1. 依赖范围：在开发中，如果我们要使用某一个框架，一般需要将这个框架引入到classpath中，这样项目才可以使用该框架。Maven提供了三种classpath方法，分别是编译、测试、运行。因此，依赖范围是用于控制依赖与三种classpath的关系的。

查看官方文档后得知，maven的依赖范围有六种：



1. compile：默认的范围，编译测试运行都有效。
2. provided：在编译和测试有效。例如：在构建Java EE应用时，可以将servlet API设置为provided，这样servlet在编译和测试时都有效，在运行时无效。因为在运行时web容器已经包含了servlet了，这样会导致冲突。（大概是这个意思…）
3. runtime：在测试和运行时有效。比如：jdbc驱动的实现，项目的主代码的编译是需要jdk提供jdbc的API，只有在测试或者运行时才需要jdbc具体的实现。
4. test：只在测试范围有效。例如：junit。
5. system：与provided一样，在编译和测试时有效。但system与本机系统相关联，可移植性差。比如我们要引用本机的JAVA\_HOME，如果在其他的机器上就可能因为路径而导致错误。
6. import：导入的范围，它只适应在dependencyManagement中，表示从其他的pom中导入dependency。
7. **在Eclipse中创建一个Web项目**
8. 通过maven project创建一个webapp。
9. 发现index.jsp有错误，因为没有引入servlet API，遂引用之。

<!-- servlet -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

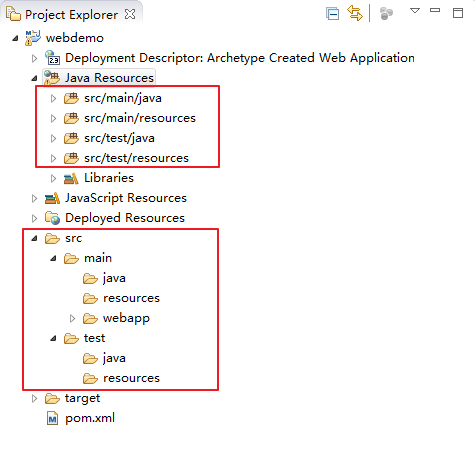
<version>3.1.0</version>

<!-- 只在编译和测试时运行 -->

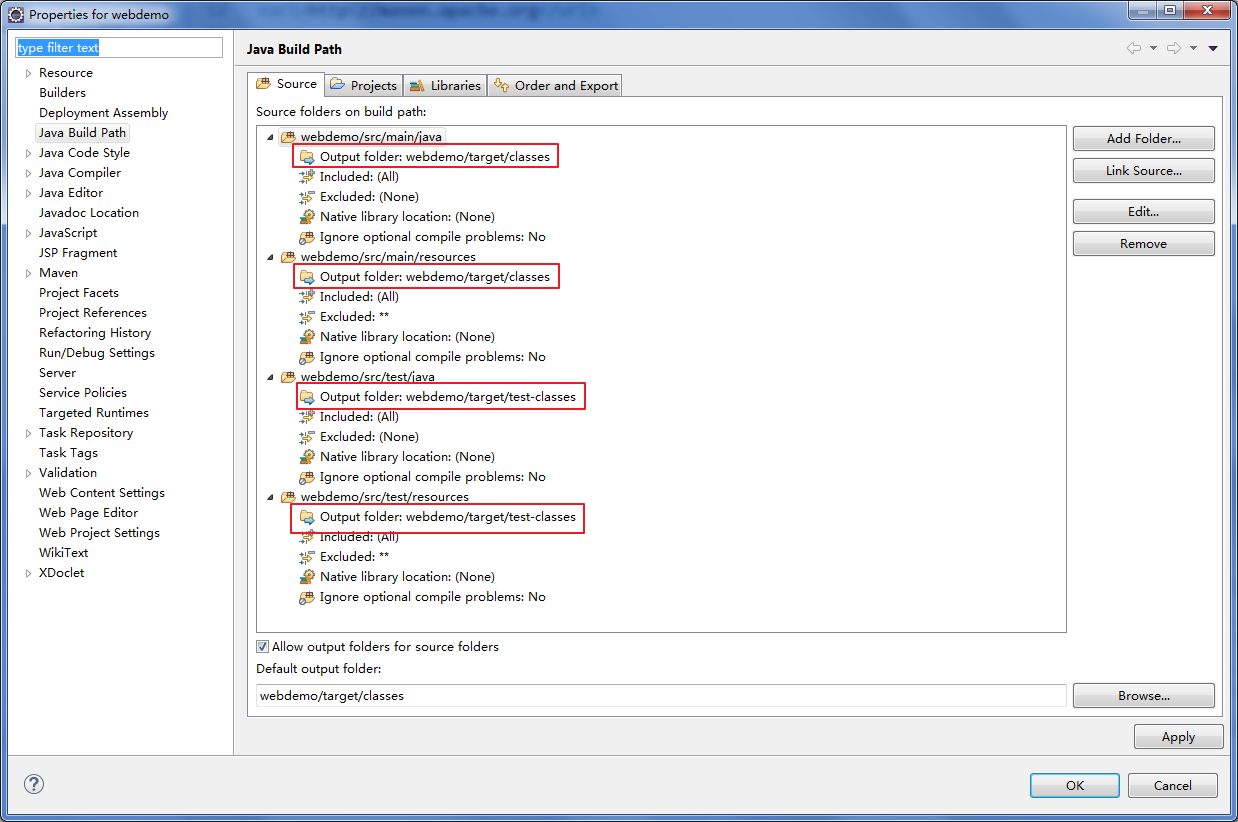
<scope>provided</scope>

</dependency>

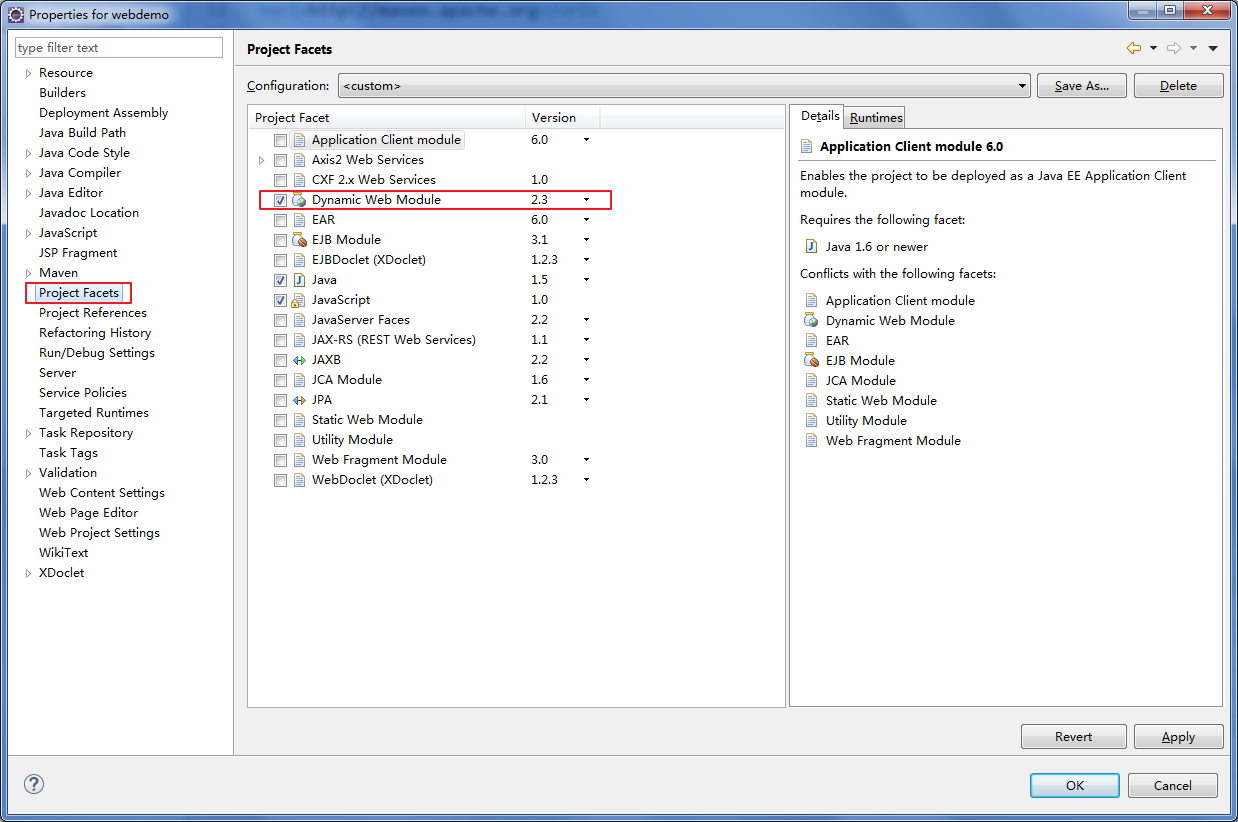
1. 检查创建的目录是否正确，并补全。



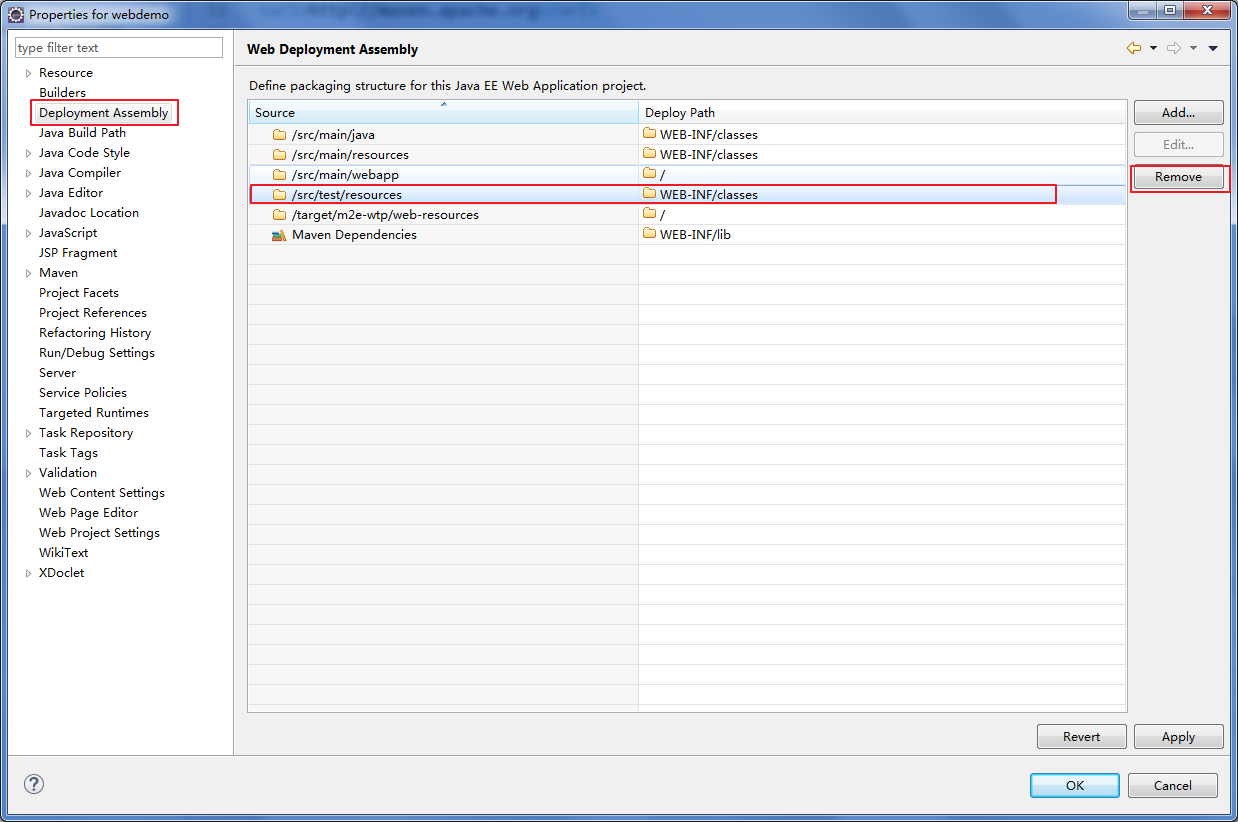
1. 检查classes文件的输出路径是否正确：确保它输出在target的目录下。



1. 将项目转化为web项目。右键项目名，点击properties -> Project Factes -> 勾选Dynamic Web Module。



1. 修改部署时的默认配置。部署时删除test目录下的文件。



1. 使用mvn package命令将其打包，发布到web容器中。首先在build标签中添加tomcat插件！



8）mvn clean package启动tomcat！！！