# Maven安装

1. 检查JDK安装的情况(需要1.6以上的版本):

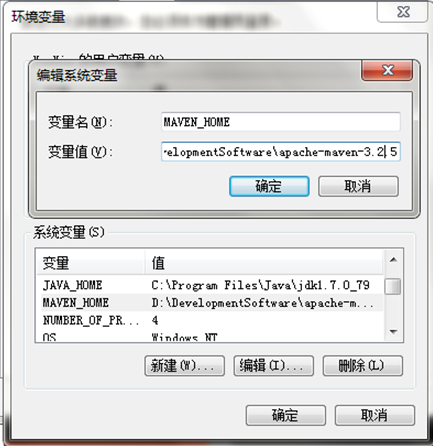
添加JAVA\_HOME，需要指向JDK安装目录；

添加PATH，添加%JAVA\_HOME%\BIN；

1. 安装Maven：

apache-maven-3.2.5-bin.zip解压到目录中（目录路径最好不要有空格和中文）；

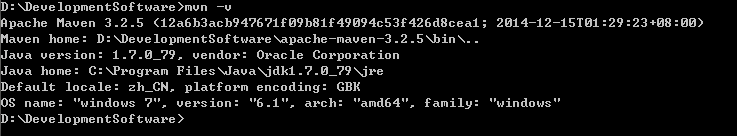
设置系统环境变量，MAVEN\_HOME，指向maven的根目录



设置环境变量Path，将%MAVEN\_HOME%\bin加入Path中;

1. 验证Maven安装

打开cmd窗口，敲入mvn –v 查看



4. 额外的配置：

Maven在构建大型业务项目的时候，十分消耗内存，可以为Maven的运行配置JVM内存：设置MAVEN\_OPTS环境变量（中间是空格）

-Xms128m -Xmx512m

1、Maven规定了一套默认的项目格式：

src/main/java —— 存放项目的.java文件

src/main/resources —— 存放项目资源文件，如spring、struts2配置文件，db.properties

src/main/webapp —— 存放jsp，css，image等文件

src/test/java —— 存放所有测试.java文件，如JUnit测试类

src/test/resources —— 测试资源文件

**pom.xml ——主要要写的maven配置文件**

target —— 项目由maven自动输出位置

2、目录结构：

--src

-----main

----------java

-----test

----------java

--pom.xml

1. 切换到项目 根目录下，分别执行下列命令
2. 执行 mvn compile命令 （编译）

先根据pom文件在本地仓库查找架包，没有就到远程仓库（如中央仓库）查找下载。然后进行编译成class文件放到 target目录下classes目录下 （compile工具第一次也是要下载的）

1. 执行mvn clean命令 （清空项目target目录）

先根据pom文件在本地仓库查找架包，没有就到远程仓库（如中央仓库）查找下载。然后清空项目target目录

（clean工具第一次也是要下载的）

1. 执行mvn clean compile命令

先删除target目录（减少以前残留的影响）， 再进行编译程序，架包没有的话，还是要等待下载之后进行

1. 执行mvn clean test命令

先删除target目录（减少以前残留的影响）， 再对程序进行单元测试，架包没有的话，还是要等待下载之后进行

1. 执行mvn clean package命令

先删除target目录（减少以前残留的影响）， 再对程序进行编译后 打成架包放到target目录下，

如果clean package或者pom文件中的架包没有的话，还是要等待下载之后进行

1. 执行mvn clean site命令

先删除target目录（减少以前残留的影响）， 再对程序进行编译后 打成架包放到target目录下，最后生成站点

如果clean package site或者pom文件中的架包没有的话，还是要等待下载之后进行

1. 执行mvn install命令

将架包放到（上传到） 本地仓库

如果 install架包没有的话，还是要等待下载之后进行

1. 执行mvn clean deploy命令

先删除target目录（减少以前残留的影响）， 再对程序进行编译后 打成架包放到target目录下，上传到设置好的远程仓库

如果clean package deploy或者pom文件中的架包没有的话，还是要等待下载之后进行

扩展：

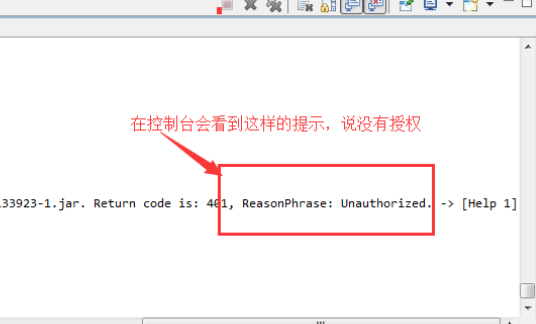
要想发布项目到nexus里，必须通过<distributionManagement>标签来进行配置。hosted里的Releases、Snapshots.

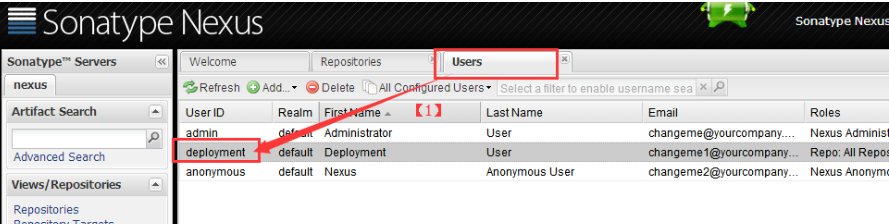


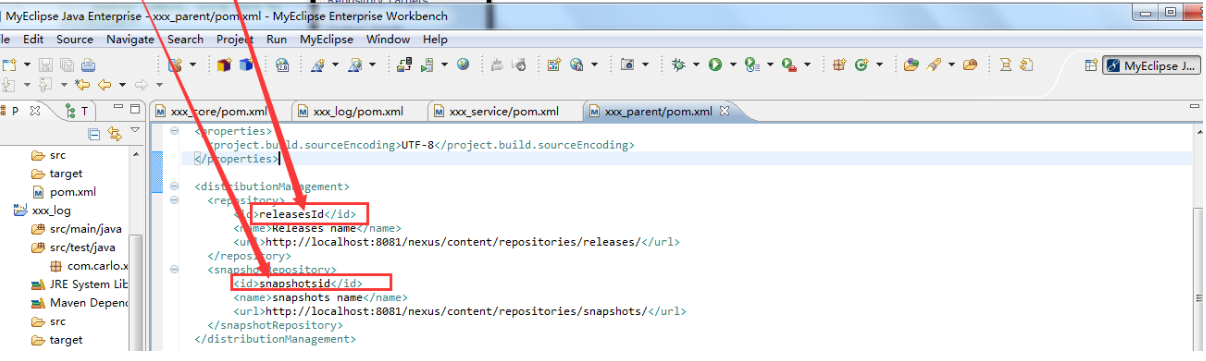
当我们发布项目到nexus里时，如果项目版本是x.x.x-Releases，则会发布到Releases工厂中；而项目版本是x.x.x-SNAPSHOTS则发布到Snapshots工厂中。

<http://blog.csdn.net/cwh056056/article/details/49667971> (参考)

很有可能提示 Unauthorized （未授权）



【1】：去到nexus管理界面 --- > 左侧菜单栏“Security”  --> “Users” ，右侧所列出的用户中，只有deployment用户才有发布项目到nexus的权限。

【2】：在setting.xml（Maven）里使用<server>标签进行授权。server里的id对应<distributionManagement>里设置的id。

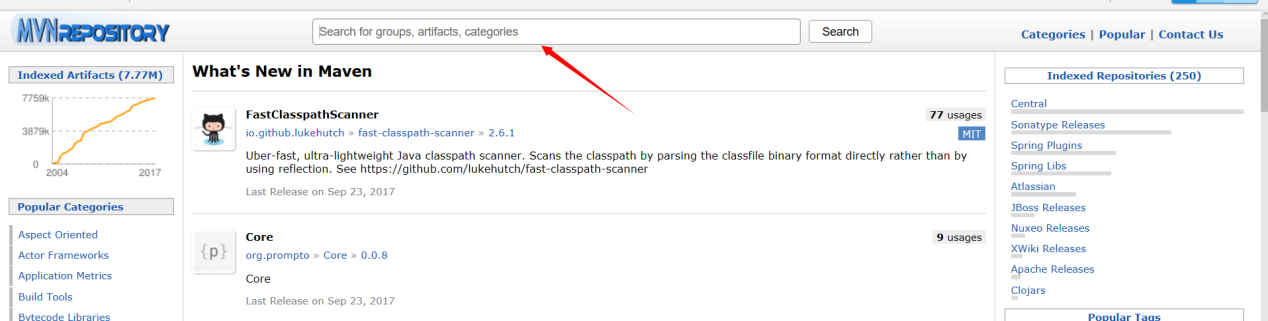
流程是：当执行clean deploy命令进行发布时，首先会找到<distributionManagement>的配置，获取配置信息。

然后如果setting.xml里有配置server，对比id值，如果匹配的上，就验证server里的用户是否拥有发布的权限，有权限就把项目发布到对应的仓库里。

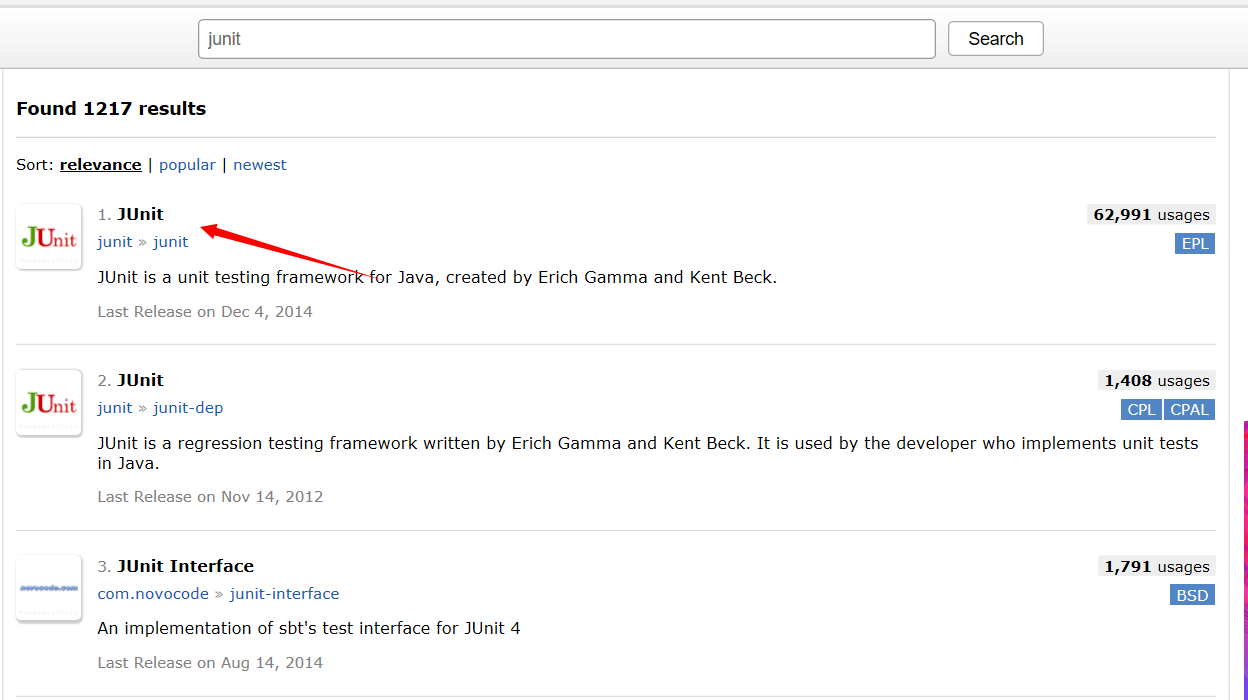
至此，发布的配置就完成了，执行clean deploy命令后，就会在nexus的Releases或Snapshots仓库中找到发布的项目了。

1. 如何查找架包坐标

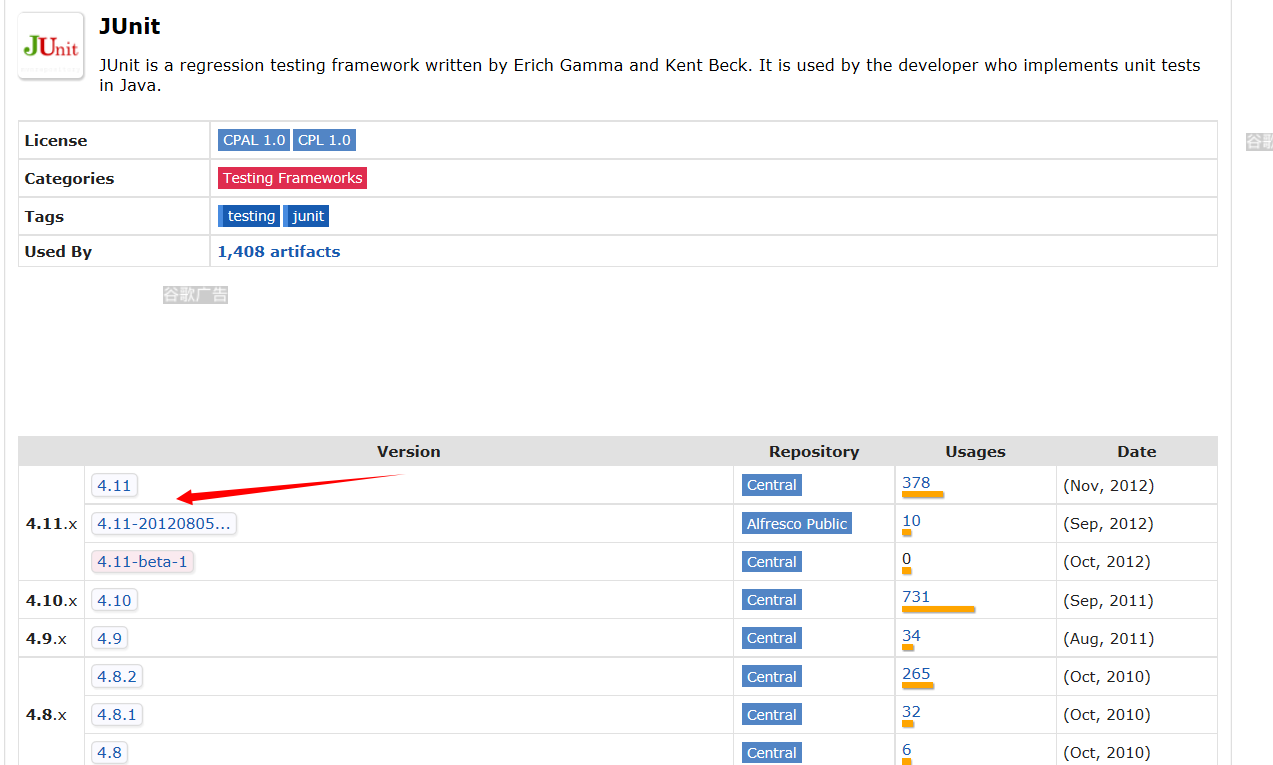
1、<https://kyfw.12306.cn/otn/queryOrder/init#nogo>



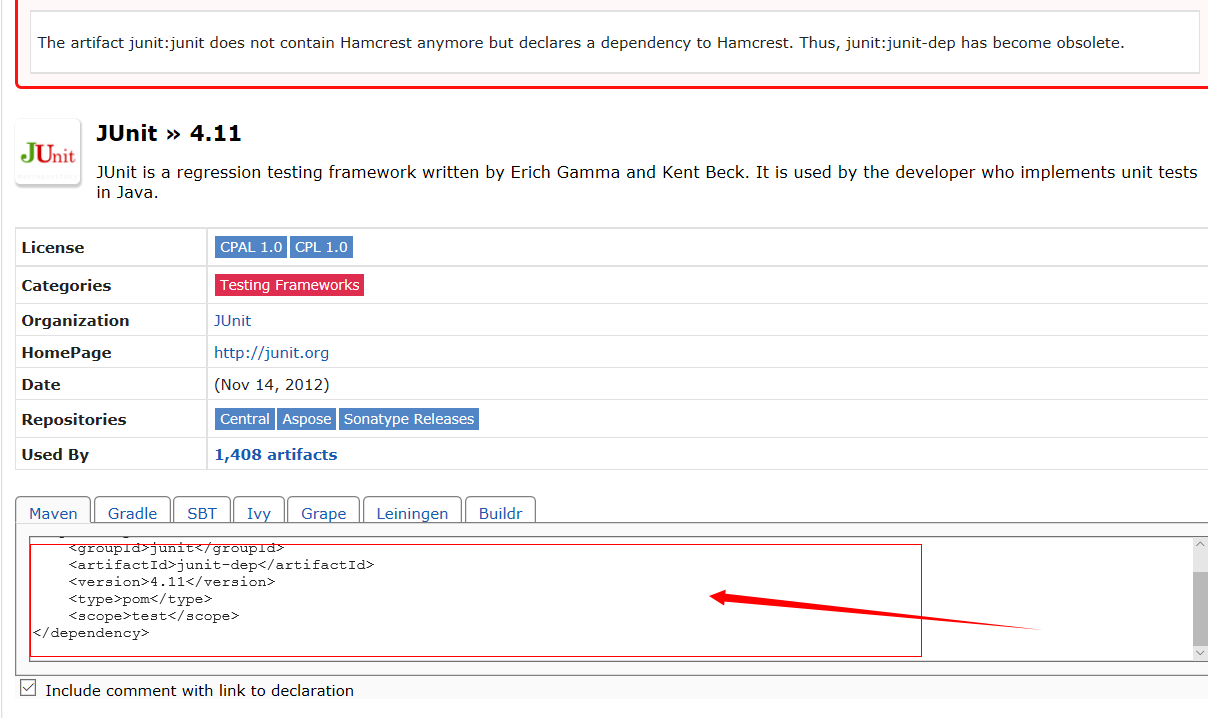
2、输入架包名字 ，点击进去



3、找到相应的版本，点进去

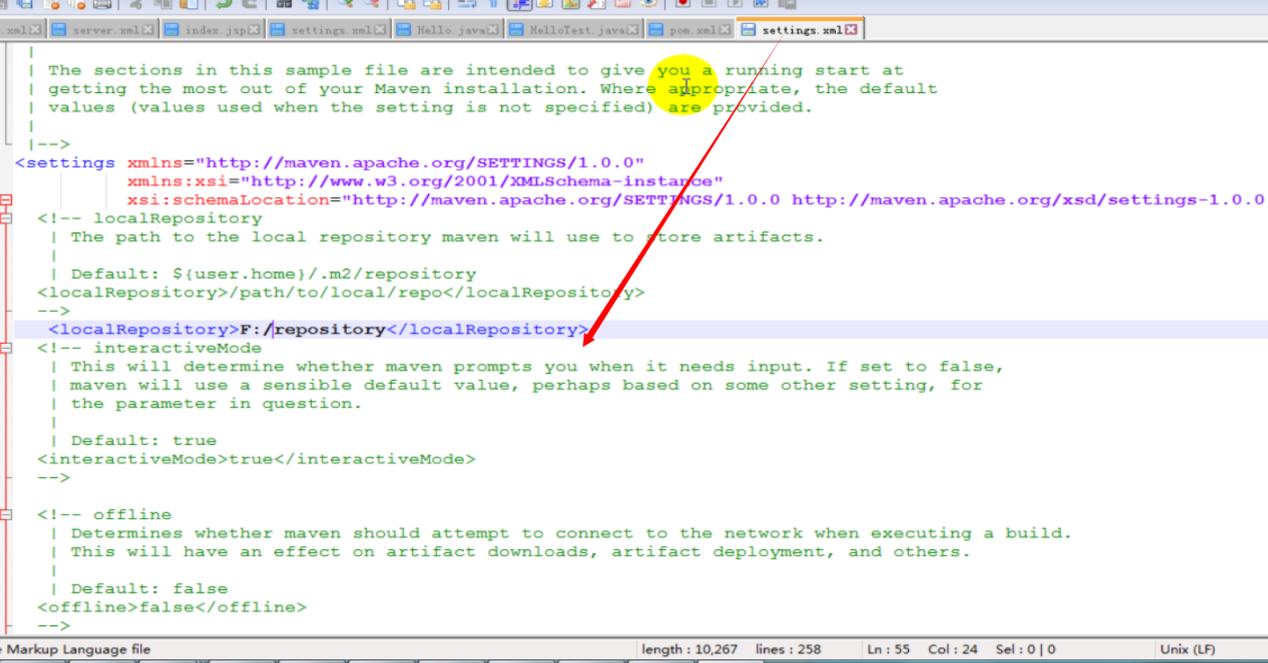


4、在下面就可以找到坐标，复制到pom.xml



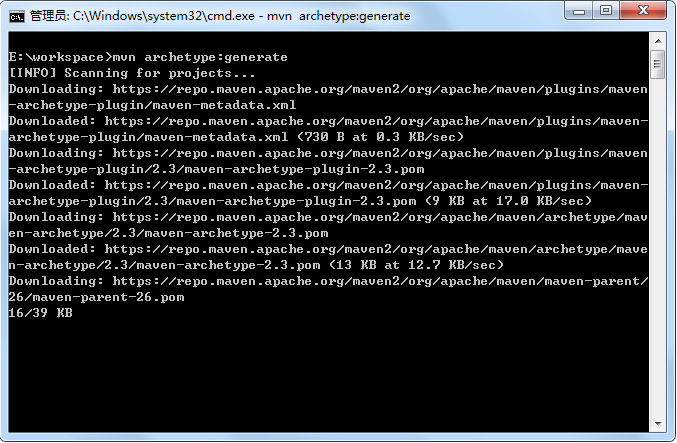
1. 如何改变本地仓库

改变localRepository的值（打开注释后，写入路径）

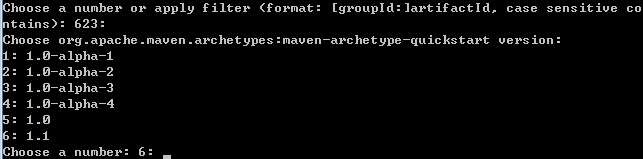


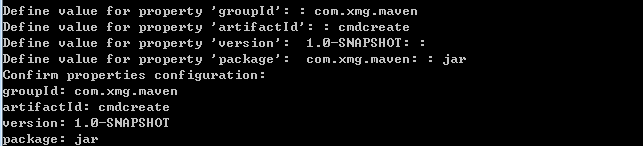
可以通过MAVEN的命令来生成Maven项目骨架：

执行命令mvn archetype:generate

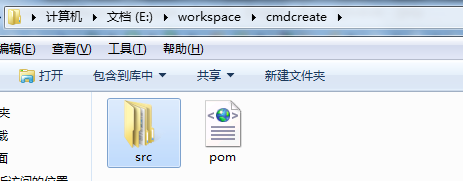


可以看到有很多的archetype，每个前面有一个序号，输入序号（或者直接回车，选的就是quick-start）：





一次输入需要的内容，

项目创建成功

2，一次性生成

mvn archetype:create -DgroupId=[your group id] -DartifactId=[your archetype id] -DarchetypeArtifactId=maven-archetype-webapp

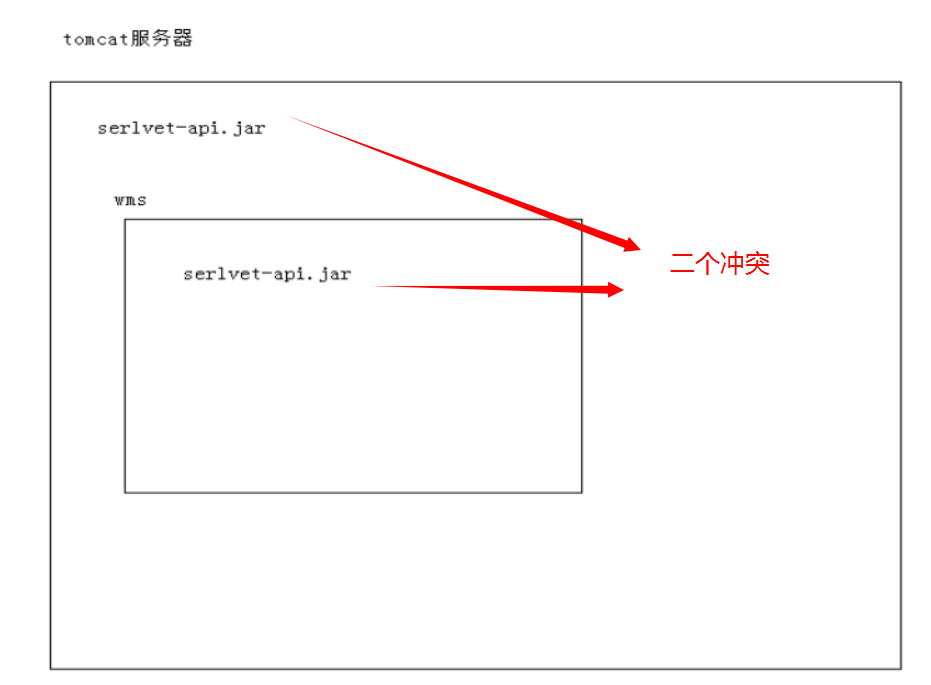
3，如果mvn archetype:generate列出内容过多，或者下载过慢，运行：

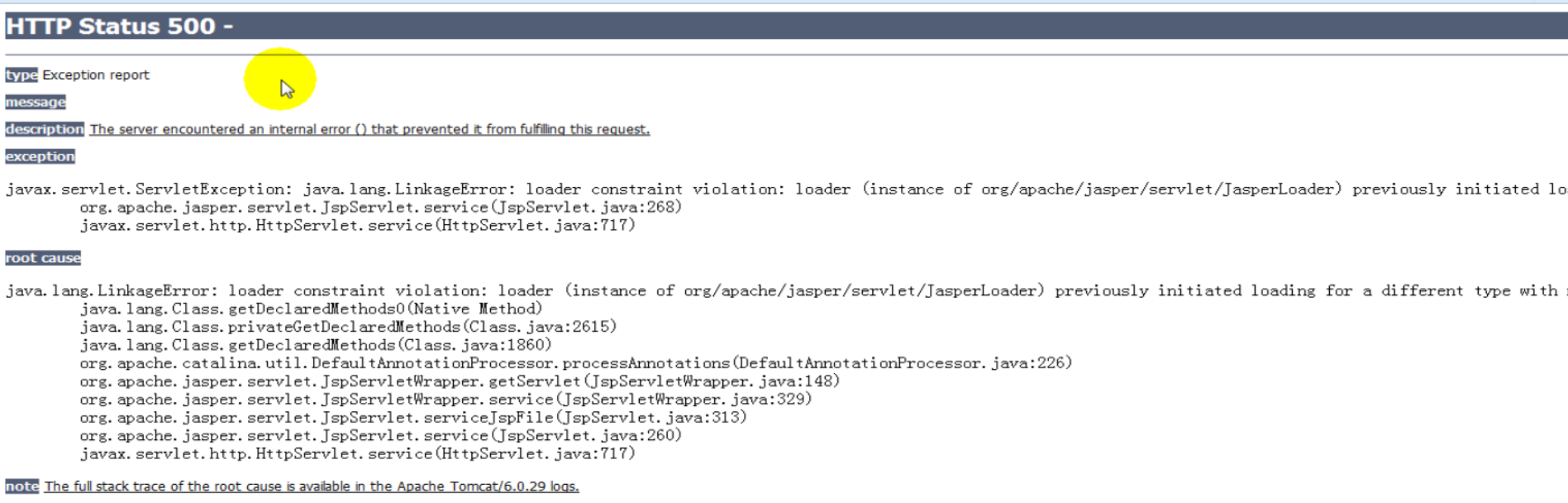
mvn archetype:generate -DarchetypeCatalog=internal

只列出系统内部有的archetype；

1. 依赖问题

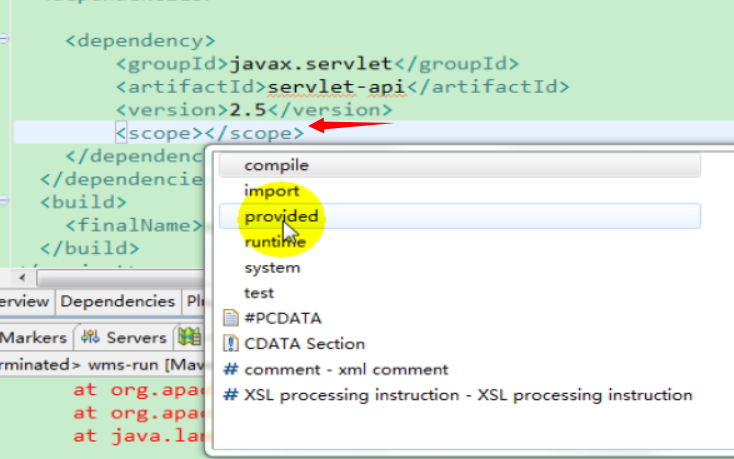
在pom引用servlet架包，但是部署到tomcat的时候，由于tomcat是有这些架包的，所以冲突了，会有下面的错误





解决：设置idea编译加入， 打包的时候不依赖

设置架包的作用域： 设置为provided ，之前的问题就解决了



provided : 编译的时候引用架包 ， 打包不引用

五、坐标和依赖

坐标：Maven中坐标就是用来规范maven唯一标示一个构建的方法；

Maven坐标构成：

1，groupId: 项目名称

2，artifactId: 项目中Maven项目/模块名称

3，version: 项目版本

4，packing: 打包的方式，比如jar、war、pom等；

5，classifier: 附属构件，比如source，javadoc等，但是注意，一般不会直接定义classifier，附加构件不是直接定义的，是插件帮助构件的；

<groupId>svr.doushisi</groupId>  
<artifactId>doushisi-logic-refactor</artifactId>  
<packaging>jar</packaging>  
<version>1.0-SNAPSHOT</version>  
<name>doushisi-logic-refactor</name>

生成的文件名是根据坐标生成的，规则：artifactId-version[-classifier].packing

在仓库中的位置也是根据坐标生成的，后面再介绍；

配置依赖：

1，一个较完整的依赖配置包括

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<type></type>

<scope>test</scope>

<optional></optional>

<exclusions>

<exclusion></exclusion>

</exclusions>

</dependency>

groupId： 该依赖项目的groupId；

artifactId： 该依赖项目的artifactId；

version： 该依赖项目的版本；（也是通过这三个坐标找到jar包位置）

type： 依赖项目的类型，对应坐标中的packing，即文件类型，一般不填，默认为jar；

scope： 该依赖项目的依赖范围；

optional： 标记依赖是否可选；

exclusion： 设置排除依赖传递性；

（这里用于 二个架包 都依赖了 同一程序的不同版本，发生了冲突，

解决： 用exclusion 把其中一个去掉）

最佳实践

1，使用properties归类版本： 以后要改版本，就统一改这里

<properties>

<org.springframework.version>3.0.5.RELEASE</org.springframework.version>

</properties>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${org.springframework.version}</version>

<scope>test</scope>

</dependency>

2，依赖的分析：

mvn dependency:analyze

mvn dependency:tree

scope（作用域）

1，Maven在编译，测试，运行的时候，各需要一套classpath；（比如编译的时候会有编译的classpath；测试的时候会有测试的classpath；）scope表明该依赖的项目和三个classpath的关系

2，scope的可选值：

1）compile：默认，适用于所有阶段，会随着项目一起发布,在编译，测试，运行时都有效；

2）test：测试范围，不会随项目发布，如junit；

3）provided：运行时已提供；典型的如servlet-api.jar，打包时不需要，容器来提供.

4）runtime：运行时有效，用于接口和实现分离；典型的如jdbc具体驱动实现；

5）import：（maven继承中使用，不讲）

3，依赖的传递性：

1）测试junit4.8和junit4.9的区别；并查看Junit4.9依赖项目中的pom.xml文件；

2）依赖的分类；A->B（第一直接依赖）;B->C（第二直接依赖）；A->C（传递依赖）

3）依赖的传递：

（第二依赖） compile test provided runtime

(第一依赖)

compile compile N compile

test test N test

provided provided N provided provided

runtime runtime N runtime

optional： 标记依赖是否可选；

依赖的可选： 可选项目不会被传递；

如果A项目依赖B项目；B项目依赖C或者D（比如一个基础项目，兼容了MYSQL或者ORACLE，那么他可能依赖mysql-connector或者classes12.jar）

但是在A项目运行的时候，只需要C或者D其中一个就可以了。那么对于B项目来说，C和D项目就可以设置optional为true；在A项目中单独设置C或者D的依赖；

exclusions：排除一个或多个传递性依赖；

传递性依赖可能带来的问题：

A->B1；C->B2；那么项目中会有两个B项目；如果包不存在冲突，那么就可以保留一个高版本的B版本即可；在A项目中：

<dependency>

<groupId>org.xxx</groupId>

<artifactId>A</artifactId>

<version>1.1</version>

<exclusions>

<exclusion>

<groupId>org.xxx</groupId>

<artifactId>B</artifactId>

</exclusion>

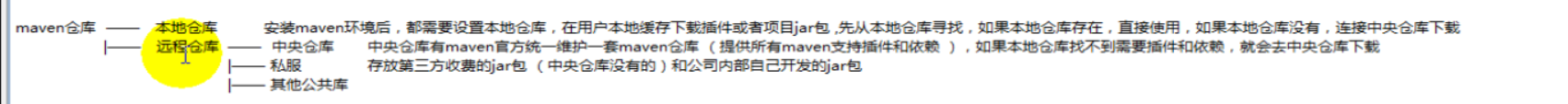
</exclusions>

</dependency>

仓库: 用于统一管理和存放依赖包的地方；

仓库中依赖包存放的方式和坐标的对应关系：groupId/artifactId/version/artifactId-version.packing

仓库的分类：本地仓库和远程仓库；组件仓库和插件仓库；



maven仓库 —— 本地仓库 安装maven环境后，都需要设置本地仓库，在用户本地缓存下载插件或者项目jar包 ,先从本地仓库寻找，如果本地仓库存在，直接使用，如果本地仓库没有，连接中央仓库下载

|—— 远程仓库 —— 中央仓库 中央仓库有maven官方统一维护一套maven仓库 （提供所有maven支持插件和依赖 ），如果本地仓库找不到需要插件和依赖，就会去中央仓库下载

|—— 私服 存放第三方收费的jar包 （中央仓库没有的）和公司内部自己开发的jar包

|—— 其他公共库

超级POM中：

<repositories>

<repository>

<id>central</id>

<name>Central Repository</name>

<url>https://repo.maven.apache.org/maven2</url>

<layout>default</layout>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

id：仓库的id，只有中央仓库id为central，仓库的id必须唯一，如果其他仓库id覆盖central，则相当于覆盖了中央仓库；

url：仓库的URL地址；

layout：仓库的布局，MAVEN2以上都用default；

snapshots：代表不从中央仓库中下载快照版本；

可以在POM中配置其他的公共仓库；

仓库的镜像：如果一个仓库完全和中央仓库同步，我们就可以选择本地的这种仓库作为中央仓库的镜像仓库；

在settings.xml中配置：

<mirror>

<id>maven.cn</id>

<mirrorOf>central</mirrorOf>

<name>cn</name>

<url>http://maven.net.cn/content/groups/public/</url>

</mirror>

<mirrorOf>：

\*：匹配所有的仓库（比如配置私服）

external:\*：所有远程仓库（localhost，file除外）；

rep1,rep2：仓库1和仓库2；

\*,!rep1：除了仓库1其他所有的；

仓库的搜索：

<http://www.mvnrepository.com/>

Maven的生命周期

Maven的生命周期是抽象的，比如当我们执行mvn clean实际上会执行一系列的动作，这些动作就是生命周期；

Maven的生命周期由多个阶段构成，在Maven中有以下几套生命周期：

1，clean（清理，对应clean）：

pre-clean：执行清理前要做的事情

clean：执行清理动作

post-clean：清理后要做的事情

2，default：（构建，对应install）

validate：

initialize：初始化工作

generate-sources：

process-sources： 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：

generate-test-sources：

process-test-sources：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：在测试前的工作

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：执行打包前的工作

package：接受编译好的代码，打包成可发布的格式，如 JAR 。

pre-integration-test：集成测试前工作；

integration-test：执行集成测试；

post-integration-test：集成测试后的工作；

verify：

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。

3，site（站点，对应site）

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

命令行和生命周期的对应：

1，注意，命令行不是和生命周期一一对应，可能一个命令行只执行了一个生命周期的部分阶段；

2，mvn clean：对应clean周期的clean之前的东西；

3，mvn test：对应default周期的test之前的阶段；

4，mvn clean install：对应clean+default周期的install之前的；

注意：生命周期中不是每个阶段都有事情做！！！！！

插件目标和插件

1，Maven中只定义了生命周期，而生命周期中每个步骤怎么做，是需要插件来完成的；

2，一个插件往往能执行多个功能，插件的每一个功能，就叫做插件的目标；

3，插件绑定：插件的目标需要和生命周期中的某个阶段绑定，才能完成一定的任务，插件绑定分两种：

1）内置绑定：比如，maven-clean-plugin:clean，（比如我们可以直接执行mvn org.apache.maven.plugins:maven-clean-plugin:clean插件）

2）自定义绑定：可以通过<plugin>的<execution>元素把插件目标绑定在生命周期上；

<plugin>

<groupId>org.mortbay.jetty</groupId>

<artifactId>jetty-maven-plugin</artifactId>

<executions>

<execution>

<phase>process-test-classes</phase>

<goals>

<goal>start</goal>

</goals>

</execution>

</executions>

</plugin>

4，内置绑定的执行：

1，比如mvn dependency:tree，等价于执行mvn org.apache.maven.plugins:maven-dependency-plugin:2.1:tree；在这里其实就可以看到groupId,artifactId,version都存在了；

2，怎么定位插件仓库？

1，插件是单独放在插件仓库之中的；

2，默认的插件groupId都是org.apache.maven.plugins；

3，在插件库中找到org\apache\maven\plugins，里面有一个maven-metadata-central.xml；

<plugin>

<name>Apache Maven Clean Plugin</name>

<prefix>clean</prefix>

<artifactId>maven-clean-plugin</artifactId>

</plugin>

<plugin>

<name>Apache Maven Compiler Plugin</name>

<prefix>compiler</prefix>

<artifactId>maven-compiler-plugin</artifactId>

</plugin>

在这里就能找到artifactId；