Maven

尚硅谷 Java

版本 V1.0

# 一、Maven简介

## 1. 简介

Maven是用java编写的跨平台的项目管理工具。作为Apache组织提供的一个颇为成功的开源项目。

Maven主要服务于基于Java平台的项目构建，依赖管理和项目信息管理。

地址：<https://maven.apache.org/>

## 2. 功能

### 2.1 添加第三方jar包

在今天的JavaEE开发领域，有大量的第三方框架和工具可以供我们使用。要使用这些jar包最简单的方法就是复制粘贴到WEB-INF目录下的lib目录下。但是这会导致每次创建一个新的工程就需要将jar包重复复制到lib目录下，从而造成工作区中存在大量重复的文件。

而使用Maven后每个jar包只在本地仓库中保存一份，需要jar包的工程只需要维护一个文本形式的jar包的引用——我们称之为“坐标”。不仅极大的节约了存储空间，让项目更轻巧，更避免了重复文件太多而造成的混乱。

### 2.2 jar包之间的依赖关系

jar包往往不是孤立存在的，很多jar包都需要在其他jar包的支持下才能够正常工作，我们称之为jar包之间的依赖关系。最典型的例子是：commons-fileupload-1.3.jar依赖于commons-io-2.0.1.jar，如果没有IO包，FileUpload包就不能正常工作。

那么问题来了，你知道你所使用的所有jar包的依赖关系吗？当你拿到一个新的从未使用过的jar包，你如何得知他需要哪些jar包的支持呢？如果不了解这个情况，导入的jar包不够，那么现有的程序将不能正常工作。再进一步，当你的项目中需要用到上百个jar包时，你还会人为的，手工的逐一确认它们依赖的其他jar包吗？这简直是不可想象的。

而引入Maven后，Maven就可以替我们自动的将当前jar包所依赖的其他所有jar包全部导入进来，无需人工参与，节约了我们大量的时间和精力。用实际例子来说明就是：通过Maven导入commons-fileupload-1.3.jar后，commons-io-2.0.1.jar会被自动导入，程序员不必了解这个依赖关系。

### 2.3 处理jar包之间的冲突

上一点说的是jar包不足项目无法正常工作，但其实有的时候jar包多了项目仍然无法正常工作，这就是jar包之间的冲突。

举个例子：我们现在有三个工程MakeFriend、HelloFriend、和Hello。MakeFriend依赖HelloFriend，HelloFriend依赖Hello。而Hello依赖log4j.1.2.17.jar，HelloFriend依赖log4j.1.2.14.jar。如下图所示：



那么MakeFriend工程的运行时环境中该导入log4j.1.2.14.jar呢还是log4j.1.2.17.jar呢？

这样的问题一个两个还可以手工解决，但如果系统中存在几十上百的jar包，他们之间的依赖关系会非常复杂，几乎不可能手工实现依赖关系的梳理。

使用Maven就可以自动的处理jar包之间的冲突问题。因为Maven中内置了两条依赖原则：最短路径者优先和先声明者优先，上述问题MakeFriend工程会自动使用log4j.1.2.14.jar。

### 2.4 获取第三方jar包

JavaEE开发中需要使用到的jar包种类繁多，几乎每个jar包在其本身的官网上的获取方式都不尽相同。为了查找一个jar包找遍互联网，身心俱疲，没有经历过的人或许体会不到这种折磨。不仅如此，费劲心血找的jar包里有的时候并没有你需要的那个类，又或者又同名的类没有你要的方法——以不规范的方式获取的jar包也往往是不规范的。

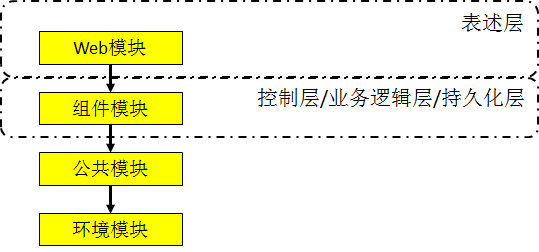
使用Maven我们可以享受到一个完全统一规范的jar包管理体系。你只需要在你的项目中以坐标的方式依赖一个jar包，Maven就会自动从中央仓库进行下载，并同时下载这个jar包所依赖的其他jar包——规范、完整、准确！一次性解决所有问题！

Tips：在这里我们顺便说一下，统一的规范几乎可以说成是程序员的最高信仰。如果没有统一的规范，就意味着每个具体的技术都各自为政，需要以诸多不同的特殊的方式加入到项目中；好不容易加入进来还会和其他技术格格不入，最终受苦的是我们。而任何一个领域的统一规范都能够极大的降低程序员的工作难度，减少工作量。例如：USB接口可以外接各种设备，如果每个设备都有自己独特的接口，那么不仅制造商需要维护各个接口的设计方案，使用者也需要详细了解每个设备对应的接口，无疑是非常繁琐的。

### 2.5 将项目拆分成多个工程模块

随着JavaEE项目的规模越来越庞大，开发团队的规模也与日俱增。一个项目上千人的团队持续开发很多年对于JavaEE项目来说再正常不过。那么我们想象一下：几百上千的人开发的项目是同一个Web工程。那么架构师、项目经理该如何划分项目的模块、如何分工呢？这么大的项目已经不可能通过package结构来划分模块，必须将项目拆分成多个工程协同开发。多个模块工程中有的是Java工程，有的是Web工程。

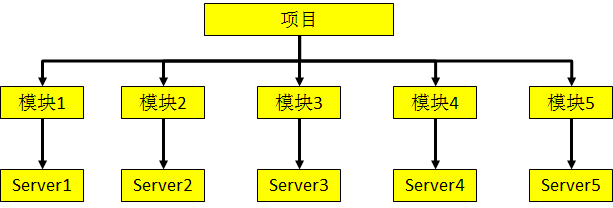
那么工程拆分后又如何进行互相调用和访问呢？这就需要用到Maven的依赖管理机制。大家请看我们的Survey调查项目拆分的情况：



上层模块依赖下层，所以下层模块中定义的API都可以为上层所调用和访问。

### 2.6实现项目的分布式部署

在实际生产环境中，项目规模增加到一定程度后，可能每个模块都需要运行在独立的服务器上，我们称之为分布式部署，这里同样需要用到Maven。



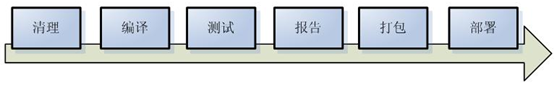
## 3. 项目构建

项目构建指项目从开发到打包发布的整个过程。

Maven是一款自动化构建工具，专注服务于Java平台的项目构建和依赖管理。在JavaEE开发的历

史上构建工具的发展也经历了一系列的演化和变迁：

Make→Ant→Maven→Gradle→其他……



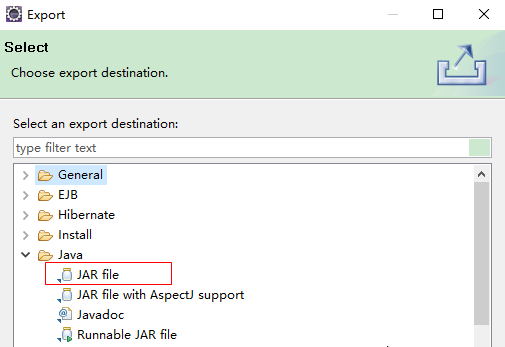
1. 清理：删除以前的编译结果，为重新编译做好准备。
2. 编译：将Java源程序编译为字节码文件。
3. 测试：针对项目中的关键点进行测试，确保项目在迭代开发过程中关键点的正确性。
4. 报告：在每一次测试后以标准的格式记录和展示测试结果。
5. 打包：将一个包含诸多文件的工程封装为一个压缩文件用于安装或部署。Java工程对应jar包，Web工程对应war包。
6. 安装：在Maven环境下特指将打包的结果——jar包或war包安装到本地仓库中。
7. 部署：将打包的结果部署到远程仓库或将war包部署到服务器上运行。

## 4. 打包

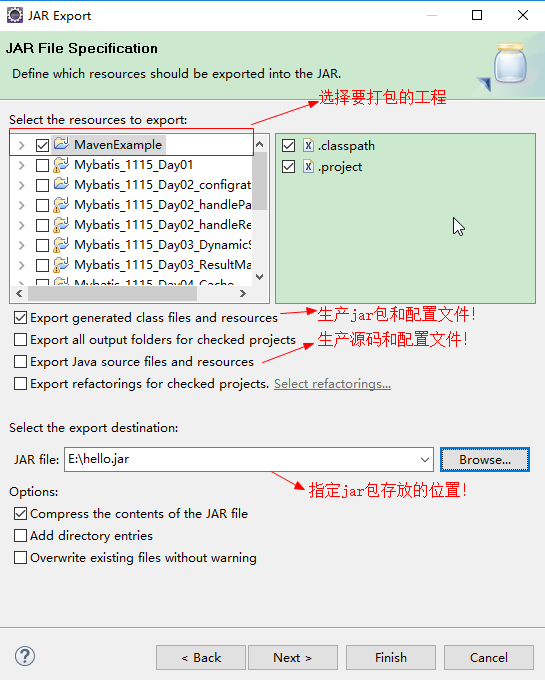
打包即将开发出来的功能，作为一个通用的jar，发布出去

### 4.1 使用Eclipse打jar包

①选择工程，点击右键，选择Export🡪jar file



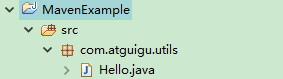
②选择jar包生成的位置：



jar包的文件结构：



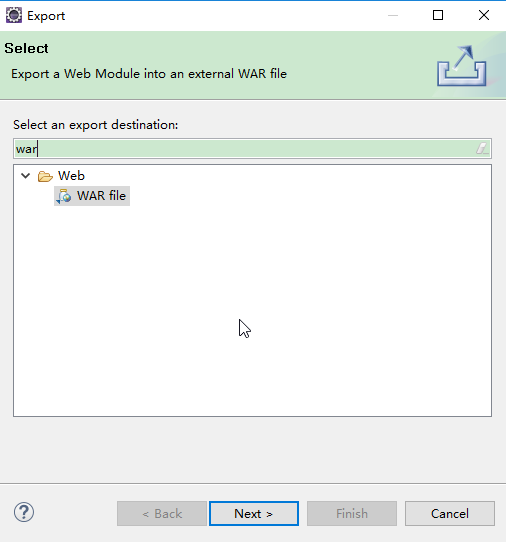
工程的视图：



工程中源码文件夹中的java文件，会被编译后，生成class文件，保存在jar包的根路径（类路径）。

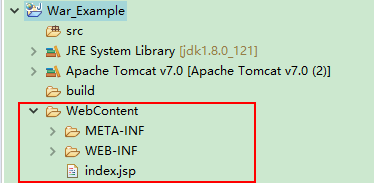
### 4.2 使用Eclipse打war包

选择web工程，右键选择Export🡪WAR file



打包后的目录结构：





①web项目WebContent下面的所有内容都在war包的根目录下；

②源码文件夹（src）目录的内容都会放在 WEB-INF/classes下；

将war包放入tomcat的webapps中，每次启动tomcat，tomcat会自动解压webapps中的war包为文件夹。我们就可以通过localhost:8080/项目名进行访问！

### 4.3 总结

项目构建中的打包，就是将java代码，配置文件，有组织有结构的打包成指定的目录格式；

使用Eclipse，可以将我们工程中所有的java文件编译成class文件，按照指定的格式放在指定的位置，压缩成指定的包；

  如果没有IDE呢？那就需要手动完成这一些列的操作，使用MAVEN可以达到项目构建的高度自动化。

## 5. 依赖管理

在学习新的框架或者工具之前，我们都需要导入相应的jar包，那么问题来了！以SpringMVC为例：

①使用springmvc该导入导入哪些包？

②jar包去哪里下载？

③如果出现jar包冲突怎么办？（不同版本的jar中类名一致，加入虚拟机时就会报错）

④怎么知道包之间的依赖关系（spring依赖commons-logging）

所谓依赖，就是指项目运行必须用到的jar包！

使用Maven可以帮助我们自动下载所有jar包，自动解决依赖关系（包括解决jar包冲突），自动关联源码！

## 6.项目信息管理

Maven能够提供高质量的项目信息，包括依赖关系、测试报告、站点信息等详细内容，并自动生成静态页面，拷贝到服务器上就可以直接使用。

通过静态页面，可以知道项目用到多少jar包，有哪些类，哪些测试通过，哪些没有测试通过！

# 二、Maven的安装

## 1. 安装

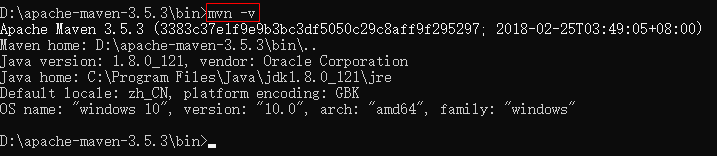
下载地址[https://maven.apache.org/download.cgi#](https://maven.apache.org/download.cgi)

将下载后的文件解压到一个非中文无空格目录，即可使用。



注意： Maven是一个java编写的项目，因此运行的前提是必须有java环境（即有JAVA\_HOME）。

解压后，可以使用mvn –v（查看maven的版本）命令来验证mvn命令是否可以执行，即安装是否成功！



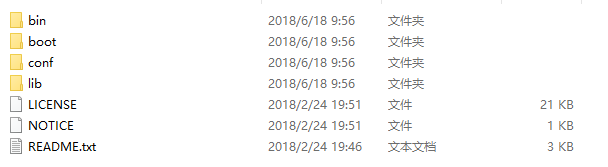
## 2. 配置环境变量(可选)

如果需要经常使用maven命令，可以将maven的bin目录添加到环境变量中！



# 三、Maven初体验

## 1. Maven的目录结构



maven目录结构：

bin：maven的可执行程序，命令

boot：引导启动；

conf：配置文件；

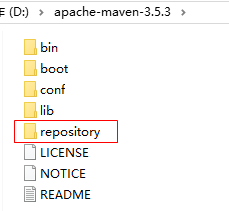
lib：maven运行期间需要的jar；

## 2. 配置Maven的本地仓库路径

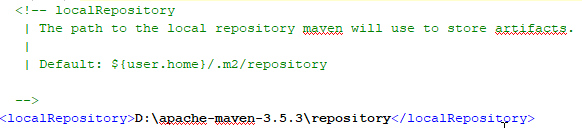
  使用Maven做项目的依赖管理时，maven会自动的下载项目依赖的一些jar，这些jar包默认会保存在本地:   C:\Users\用户目录\.m2\repository；

我们可以修改maven的配置，将下载的Jar包放入到我们自定义的目录中!

①新建文件夹：



②然后修改maven的配置文件：D:\apache-maven-3.5.3\conf\settings



## 3. 创建第一个Maven项目

### 3.1 Maven项目约定的目录结构

  maven有约定的项目的目录结构：

      项目

**src[源码]**

**main[主程序]**

**java[java代码存放的地方]**

**resources[配置文件存放的地方]**

**test[测试程序]**

**java[java代码存放的地方]**

**resources[配置文件存放的地方]**

          target[源码编译以后的东西]，target不是必须的

**pom.xml**[当前项目的配置文件类似于web.xml]

                    【项目的所有信息配置在pom.xml】

### 3.2 测试代码

①主程序代码：

**package** com.atguigu.maven;

**public** **class** Hello {

**public** String sayHello(String name){

**return** "hello"+name+"!";

}

}

②测试代码：

**package** com.atguigu.test;

**import** org.junit.Assert;

**import** org.junit.Test;

**import** com.atguigu.maven.Hello;

**public** **class** HelloTest {

@Test

**public** **void** testHello() {

Hello helloult=hello.sayHello("jack");

Assert.*assertEquals*("hellojack!", result);

}

}

=**new** Hello();

String res

③POM文件内容：

<?xml version="1.0" ?>

<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>com.atguigu.maven</groupId>

<artifactId>Hello</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Hello</name>

<dependencies>

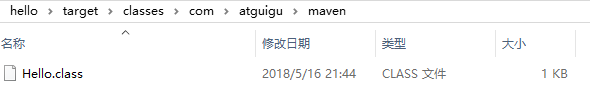
## 4. Maven命令

|  |  |
| --- | --- |
| Maven 命令 | |
| mvn compile | 编译(只编译主程序，不编译测试代码) |
| mvn clean | 清理上次的编译结果 |
| mvn test | 测试 |
| mvn package | 打包 |
| Mvn install | 将工程安装到本地仓库 |
| ※注意：运行Maven命令时一定要进入pom.xml文件所在的目录！  maven的所有功能都是网上下载相应的插件来完成的，第一次运行，maven会下载所需要的插件和当前项目需要的jar包； | |

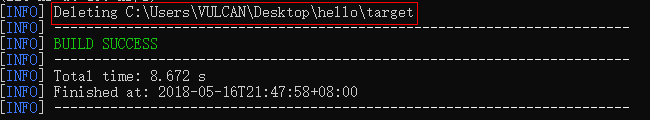
①mvn compile编译成功：



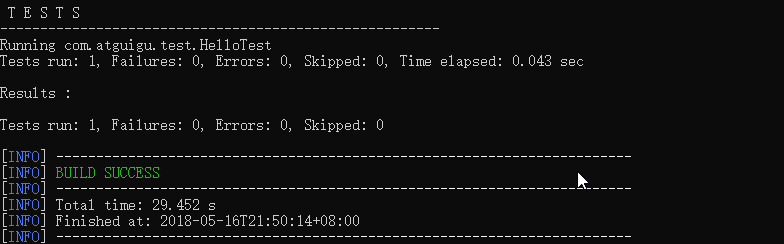
编译完的主程序代码放在target目录下：



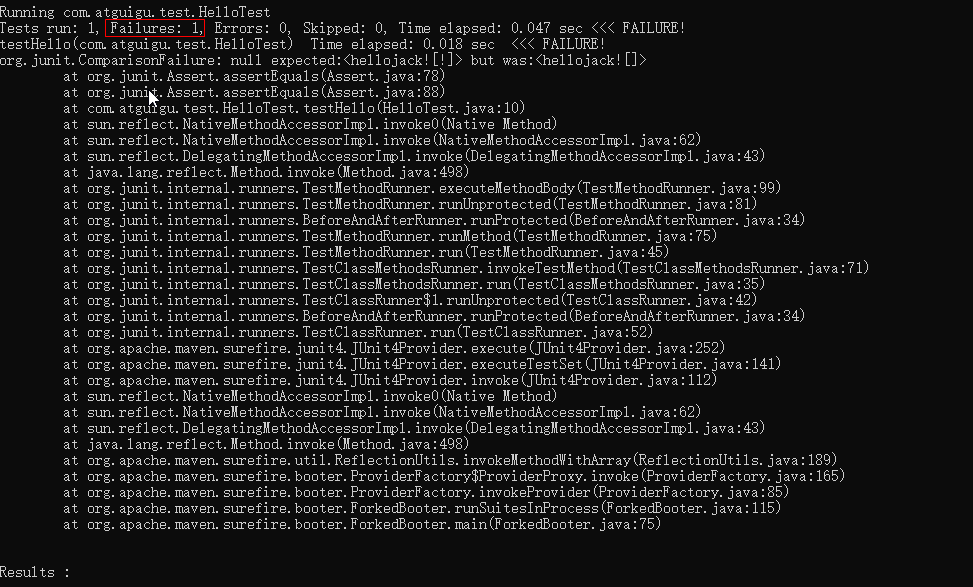
②mvn clean 清理编译结果，会删除target目录



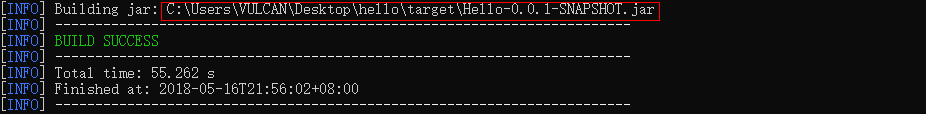
③ mvn test：测试项目（test目录下的所有测试程序都会运行）



测试失败的报告：



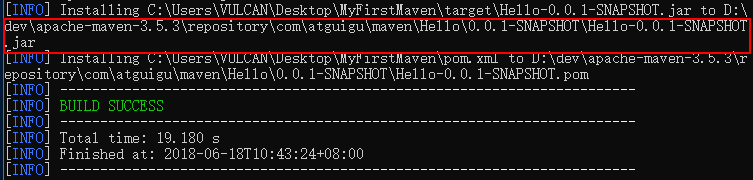
④mvn package: 项目打包(jar、war)



打包之前，会先进行测试，因此必须保证测试类都可以完整通过，才会打包。

打包的jar包名称和pom文件的配置有关！

⑤mvn install



# 四、整合Eclipse

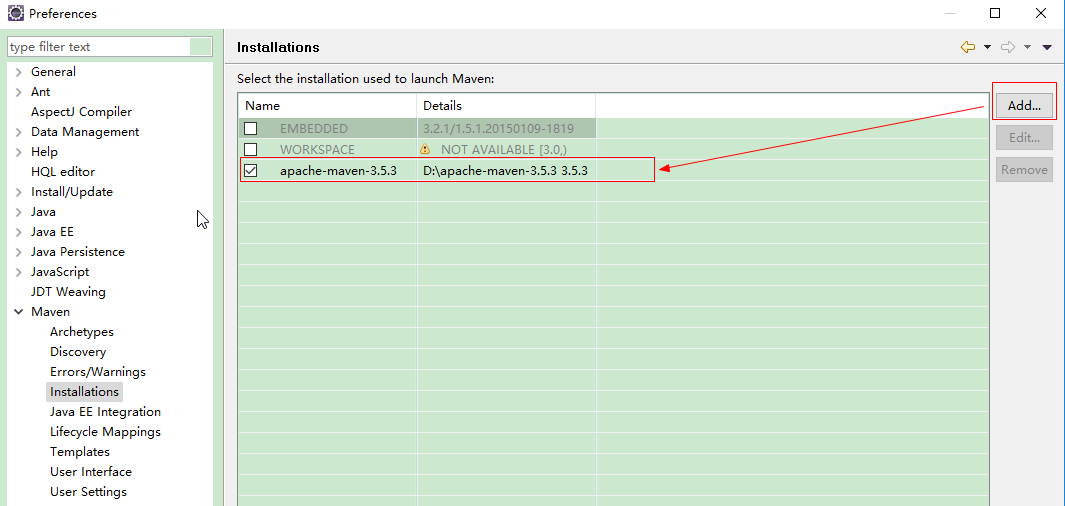
## 1. 安装Maven插件

Eclipse自Kepler版本开始内置了Maven插件，所以Maven插件不需要额外安装，使用特定版本的Eclipse即可。

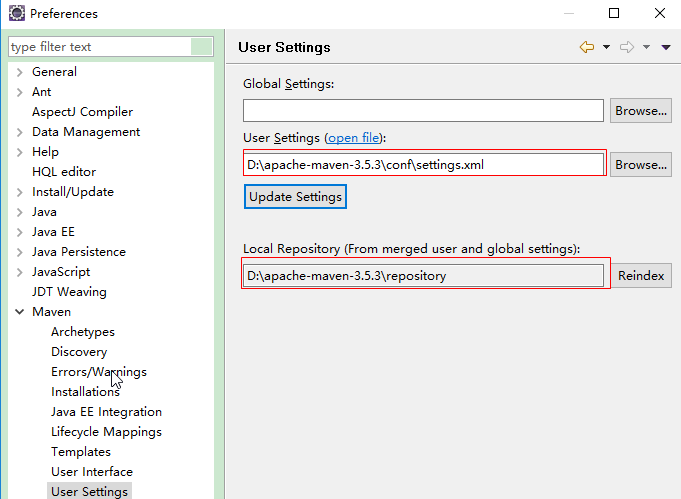
## 2. 配置Maven插件

插件的设置：

①指定Maven核心程序的位置：Window→Preferences→Maven→Installations

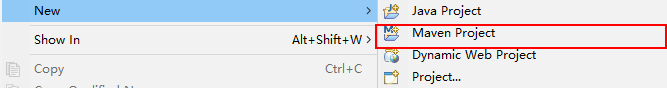


②设置Maven仓库的位置：

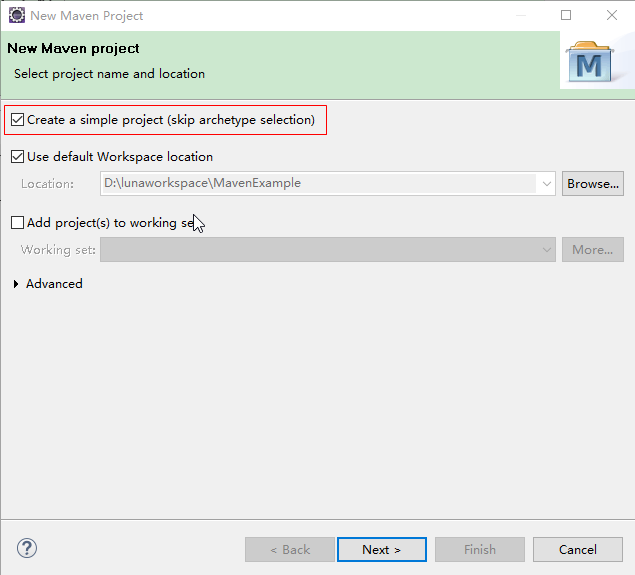


## 3. 创建一个Maven项目

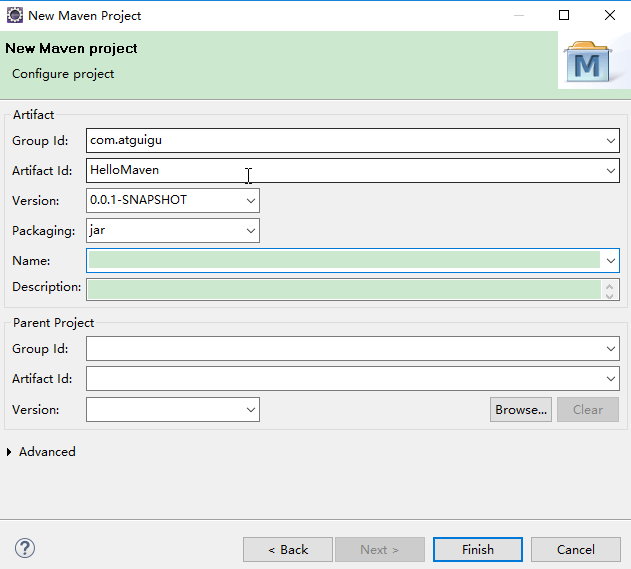
①首先选择创建一个Maven Project:



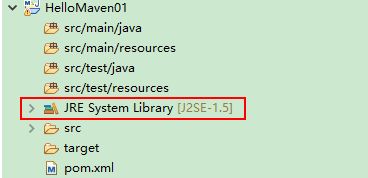
②选择create a simple project



③输入项目的GAV坐标



创建完成后的项目如图：



## 4. 修改默认的JDK版本

在Maven核心程序的settings.xml中加入如下配置

<profile>

<id>jdk-1.8</id>

<activation>

<activeByDefault>true</activeByDefault>

<jdk>1.8</jdk>

</activation>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

<maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>

</properties>

</profile>

## 5. 解决项目测试乱码问题

如果在项目测试中，使用mvn test发生了中文乱码，则可以在当前的pom.xml中增加如下配置：

<build>

<plugins>

<!-- 解决maven test命令时console出现中文乱码乱码 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.7.2</version>

<configuration>

<forkMode>once</forkMode><!--在一个进程中进行所有测试 ; 默认值:once -->

<argLine>-Dfile.encoding=UTF-8</argLine>

</configuration>

</plugin>

</plugins>

</build>

## 6. 生成web项目的web.xml

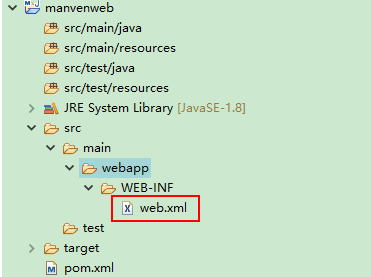
当创建一个Maven的war工程时，会发现缺少web.xml文件，从而报错：



此时，在项目上面右键，选择如下：

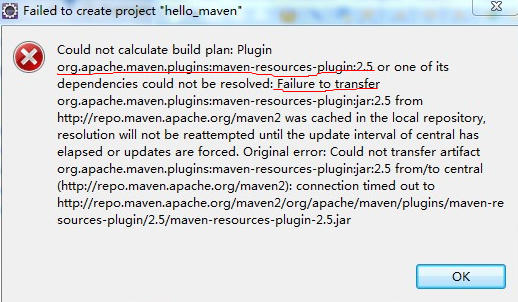


此时会自动生成web.xml文件，而webapp目录，就是之前的webcontent目录！

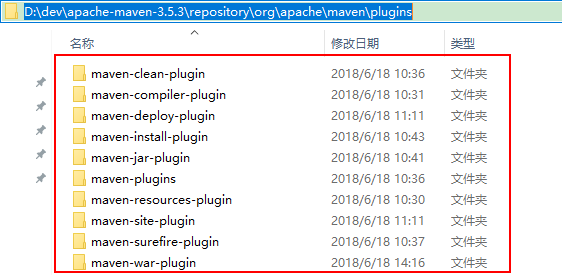


## 7. 常见下载传输错误

①Maven插件下载丢失字节错误：



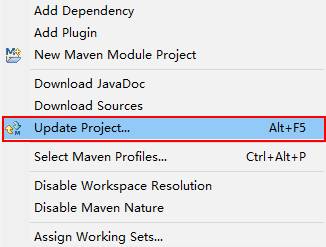
当查看错误信息，发现是Failure to transfer xxx\_plugin时，此时可以通过删除maven仓库中的对应的plugin目录，使Maven程序重新下载！

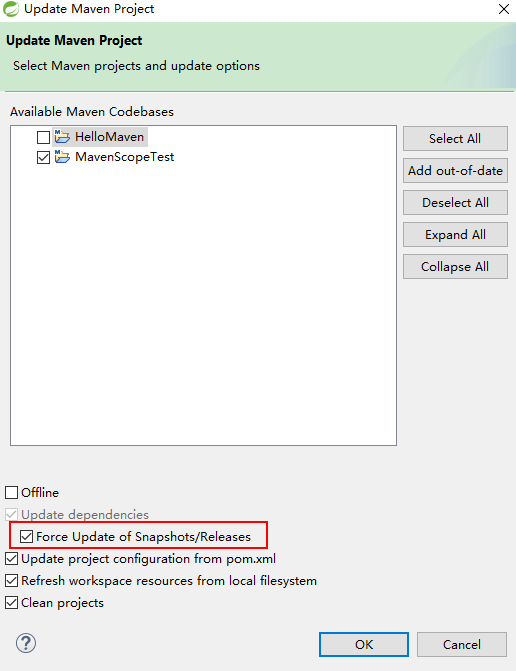


②Jar包下载丢失字节

C:\Users\VULCAN\AppData\Local\Temp\enhtmlclip\Image.png

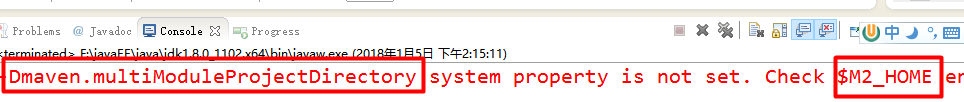
如果Jar包下载失败，会在指定的目录生成一个.lastUpdated文件。此时，删除指定的Jar包目录，然后选择强制更新项目即可！



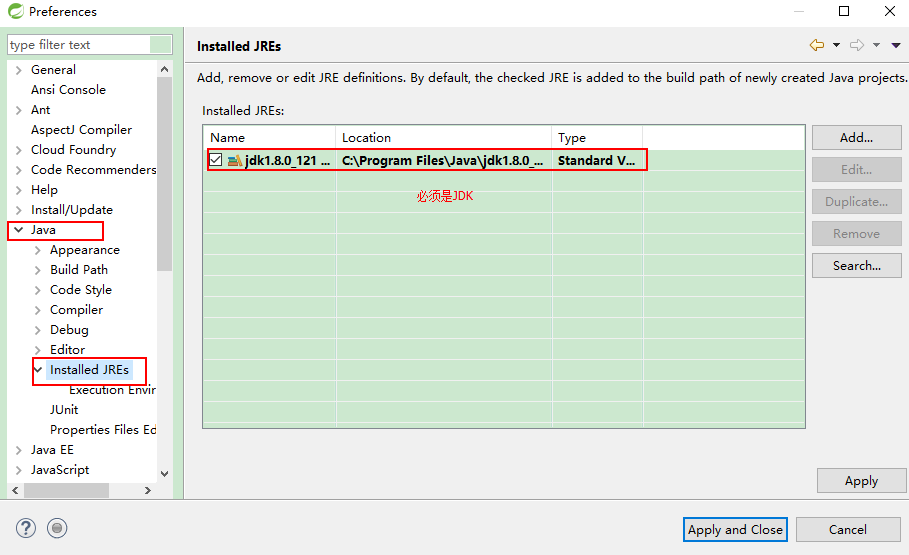


## 8. 第一次运行Maven test出错

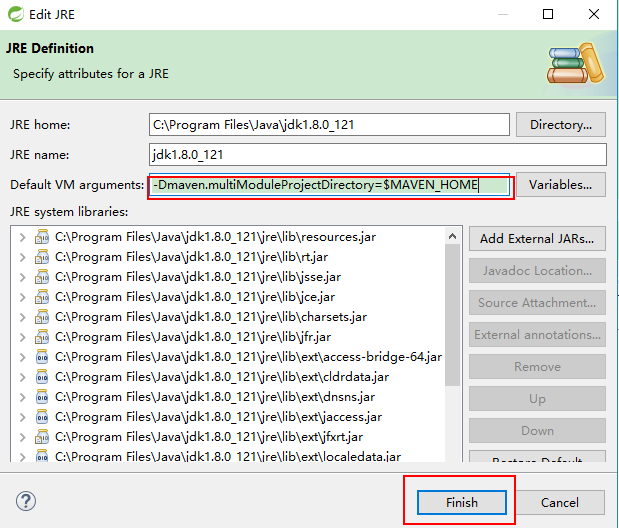
如果第一次运行mvn test出现以下错误：



解决：



选中JDK后，选择edit,:



# 五、Maven核心概念

## 1. POM（项目对象模型）

DOM（Document  Object  Model）：文档对象模型；HTML；整个HTML网页认为是一个对象，网页中的每一个标签认为是一个dom元素；

POM（Project  Object   Model）：项目对象模型；MAVEN；整个项目就是一个大的对象，项目的所有信息(（项目依赖哪些jar、项目版本号、项目怎么编译、项目该怎么测试）)就是这个对象的属性，都可以进行调整；

**pom.xml；（对这个项目的一个配置POM）；**

## 2. 坐标

**groupid**：组织，公司的标识（名）

**artifactId**：项目/模块名；

**version**：版本号；

每一个jar包都能用这个坐标唯一确定（gav坐标）；

version：版本号的命名；

          大版本（主版本）.中版本（分支版本）.小版本

**大版本**：项目的架构、设计思想、模块重组；

**中版本**：加功能；

**小版本**：改bug；

          版本说明表示：

                    RELEASE：发布版（稳定版）

                    SNAPSHOT：快照版（极不稳定版本）

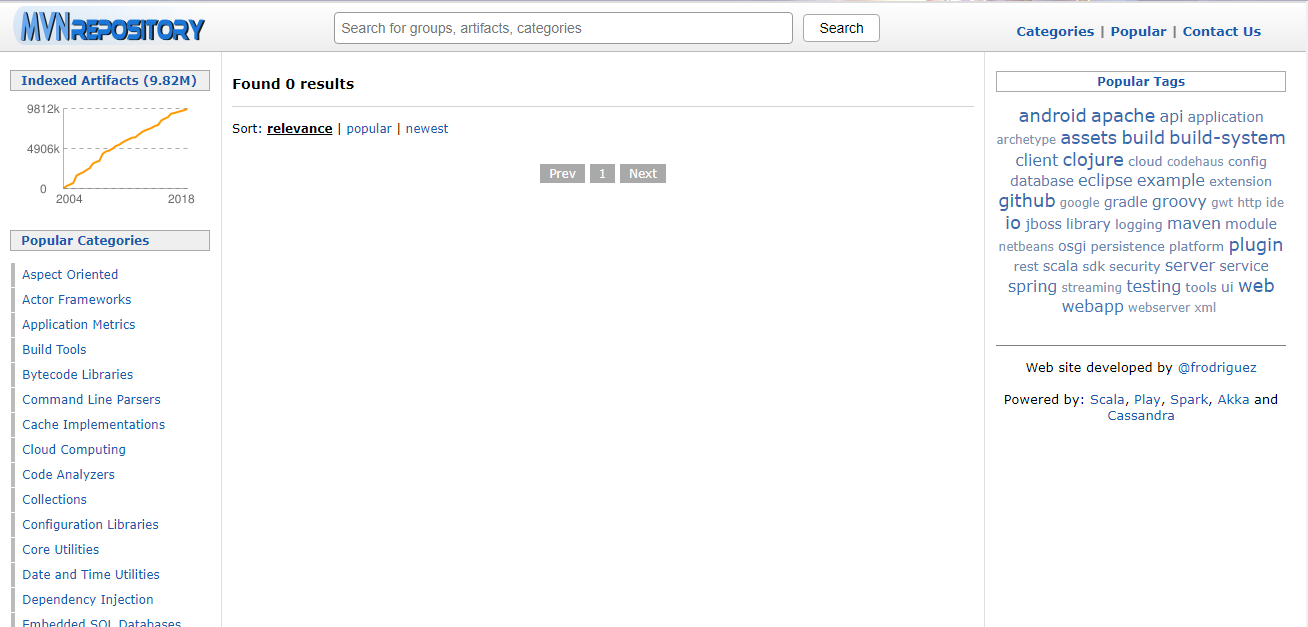
                    Alpha：内测试版

                    Beta：公测版；

## 3. Maven酷站

maven的镜像仓库：<http://mvnrepository.com/>

可以通过maven的镜像仓库来查询需要Jar包的坐标！



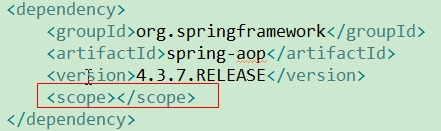
## 4. 依赖管理

所谓依赖，就是一个项目运行所需要的Jar包。

### 4.1. 依赖的Scope

要导入的jar都是依赖，使用<dependencies>管理所有依赖；

在pom.xml中，一个<dependency>代表依赖的一个jar包。每个依赖都有一个scope属性，意为依赖的作用范围！也就是jar包可以生效的范围！



常用的依赖范围如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 依赖范围(scope) | 对于主代码  classpath有效 | 对于测试  代码classpath有效 | 打包是否会携带 | 例子 |
| Compile | 是 | 是 | 是 | Log4j |
| Test | 否 | 是 | 否 | Junit |
| provided | 是 | 是 | 否 | Servlet-api |

其他：runtime、import、system等；

### 4.2 依赖的传递

假设有A工程依赖于B工程，B工程依赖于C工程，那么我们称A工程，间接依赖于C工程。

问题：A工程是否可以访问C工程中的Jar包？

结论：

①只有scope属性为compile的依赖，才可以向下传递！

②在<dependency>中，设置<optional>true</optional>的依赖也不会向下传递！

### 4.3 依赖的排除

依赖的排除使用<exclustions>标签，可以选择排除哪些依赖！

<dependency>

<groupId>com.atguigu.maven</groupId>

<artifactId>Environment</artifactId>

<version>0.0.1-SNAPSHOT</version>

<!-- 依赖排除 -->

<exclusions>

<exclusion>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

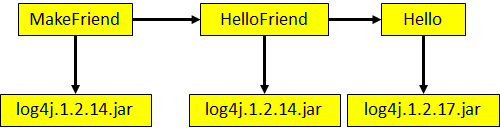
<version>1.1.2</version>

</dependency>

### 4.4 依赖的原则

依赖需要符合一定的规则，主要是为了解决jar包的冲突。

①就近原则，即路径最短者优先



②路径相同时先声明者优先



这里“声明”的先后顺序指的是dependency标签配置的先后顺序。

## 5. 统一管理目标Jar包的版本

以对Spring的jar包依赖为例：Spring的每一个版本中都包含spring-core、spring-context等jar包。我们应该导入版本一致的Spring jar包，而不是使用4.0.0的spring-core的同时使用4.1.1的spring-context。

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>4.0.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>4.0.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>4.0.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>4.0.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>4.0.0.RELEASE</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>4.0.0.RELEASE</version>  </dependency> |

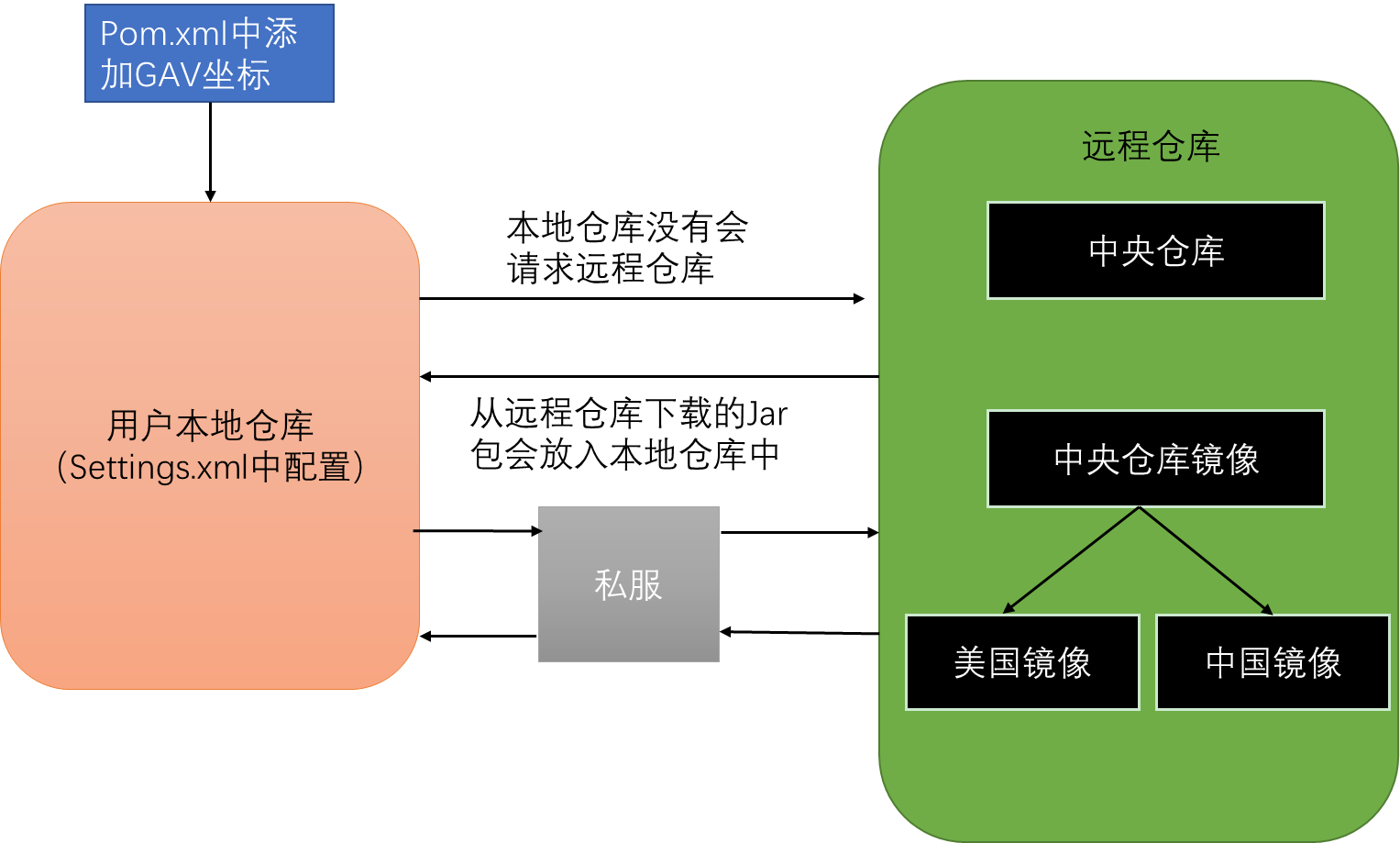
问题是如果我们想要将这些jar包的版本统一升级为4.1.1，是不是要手动一个个修改呢？显然，我们有统一配置的方式：

|  |
| --- |
| <properties>  <spring.version>4.1.1.RELEASE</spring.version>  </properties> |
| …… |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-jdbc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-web</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency> |

这样一来，进行版本调整的时候只改一改地方就行了。

## 6. 仓库

### 6.1 仓库的相关概念



仓库的分类：

1. 本地仓库：为当前本机电脑上的所有Maven工程服务。
2. 远程仓库

[1] 私服：架设在当前局域网环境下，为当前局域网范围内的所有Maven工程服务。

[2]中央仓库：架设在Internet上，为全世界所有Maven工程服务。

[3]中央仓库的镜像：架设在各个大洲，为中央仓库分担流量。减轻中央仓库的压力，同时更快的响应用户请求。

仓库中的文件：

1. Maven的插件
2. 我们自己开发的项目的模块
3. 第三方框架或工具的jar包

※不管是什么样的jar包，在仓库中都是按照坐标生成目录结构，所以可以通过统一的方式查

询或依赖。

### 6.2 修改远程仓库为阿里云镜像

在settings.xml文件的<mirrors>标签中，添加以下内容：

 <mirror>

    <id>alimaven</id>

    <name>aliyun maven</name>

    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>

    <mirrorOf>central</mirrorOf>

  </mirror>

## 7. 生命周期

什么是Maven的生命周期？

Maven生命周期定义了各个构建环节的执行顺序，有了这个清单，Maven就可以自动化的执行构 建命令了。

Maven有三套相互独立的生命周期，分别是：

Clean Lifecycle在进行真正的构建之前进行一些清理工作。

Default Lifecycle构建的核心部分，编译，测试，打包，安装，部署等等。

Site Lifecycle生成项目报告，站点，发布站点。

再次强调一下它们是相互独立的，你可以仅仅调用clean来清理工作目录，仅仅调用site来生成站点。当然你也可以直接运行 **mvn clean install site** 运行所有这三套生命周期。

每套生命周期都由一组阶段(Phase)组成，我们平时在命令行输入的命令总会对应于一个特定的阶段。比如，运行mvn clean，这个clean是Clean生命周期的一个阶段。有Clean生命周期，也有clean阶段。

①clean生命周期

Clean生命周期一共包含了三个阶段：

pre-clean 执行一些需要在clean之前完成的工作

clean 移除所有上一次构建生成的文件

post-clean 执行一些需要在clean之后立刻完成的工作

②Site生命周期

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

这里经常用到的是site阶段和site-deploy阶段，用以生成和发布Maven站点，这可是Maven相当强大的功能，Manager比较喜欢，文档及统计数据自动生成，很好看。

③Default生命周期

Default生命周期是Maven生命周期中最重要的一个，绝大部分工作都发生在这个生命周期中。这里，只解释一些比较重要和常用的阶段：

validate

generate-sources

process-sources

generate-resources

process-resources 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

generate-test-resources

process-test-resources 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

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

**package** 接受编译好的代码，打包成可发布的格式，如JAR。

pre-integration-test

integration-test

post-integration-test

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或部署到服务器上运行。

生命周期与自动化构建：

**运行任何一个阶段的时候，它前面的所有阶段都会被运行**，例如我们运行mvn install 的时候，代码会被编译，测试，打包。这就是Maven为什么能够自动执行构建过程的各个环节的原因。此外，Maven的插件机制是完全依赖Maven的生命周期的，因此理解生命周期至关重要。

## 8. 插件和目标

### 8.1 插件的概念

Maven的核心仅仅定义了抽象的生命周期，具体的任务都是交由插件完成的。

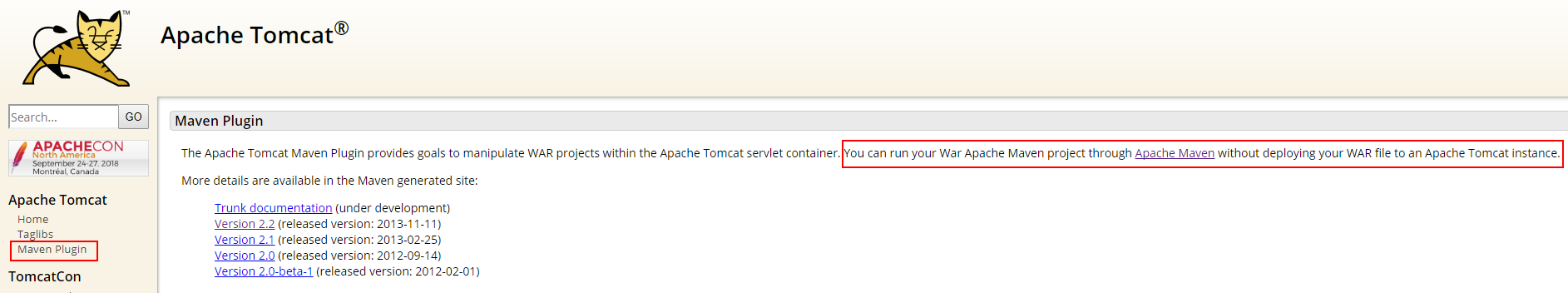
每个插件都能实现多个功能，每个功能就是一个插件目标。

Maven的生命周期与插件目标相互绑定，以完成某个具体的构建任务。

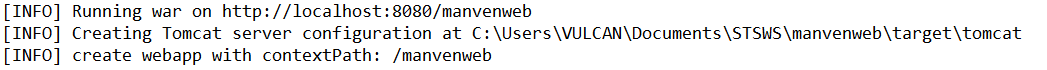
例如：compile就是插件maven-compiler-plugin的一个功能；pre-clean是插件maven-clean-plugin的一个目标。

### 8.2 使用Tomcat的Maven Plugin

Tomcat提供的Maven插件可以使用插件运行war包，而不需要启动一个Tomcat实例！



使用命令： mvn tomcat6/7:run来运行指定项目。



根据提示，访问指定路径!

还可以增加配置信息，来设置端口号和项目访问路径！

使用：在pom文件中添加以下信息：

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat6-maven-plugin</artifactId>

<version>2.2</version>

</plugin>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<path>/</path>

<port>80</port>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

# 六、继承和聚合

## 1. 继承

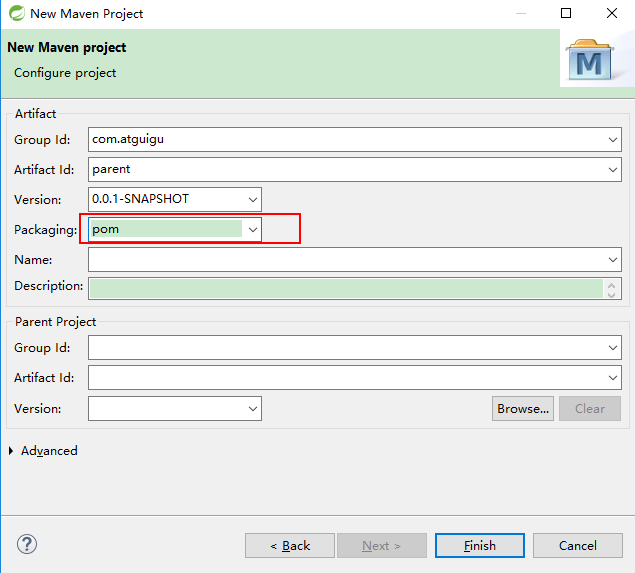
### 1.1 继承的目的

继承多用作一个项目中多个模块之间的统一Jar包版本管理。例如，一个项目中，多个模块都需要使用Junit测试Jar包，那么就分别需要在这些模块的POM文件中进行声明。随着后期项目的升级，Jar包可能需要迭代到新版本，那么就需要依次升级各个模块。

使用继承机制，可以将这样的依赖信息统一提取到父工程的模块中，进行统一管理。只需要配置子模块依赖父工程即可！

### 1.2 创建父工程

创建父工程：创建父工程和创建一般的Java工程操作一致，唯一需要注意的是：打包方式处要设置为pom。



父工程主要是做依赖的统一管理，因此不需要编写任何的java代码，只需要编写pom文件即可！



### 1.3 子工程中引用父工程

在子工程的pom文件中，填写父工程的GAV坐标：

<parent>

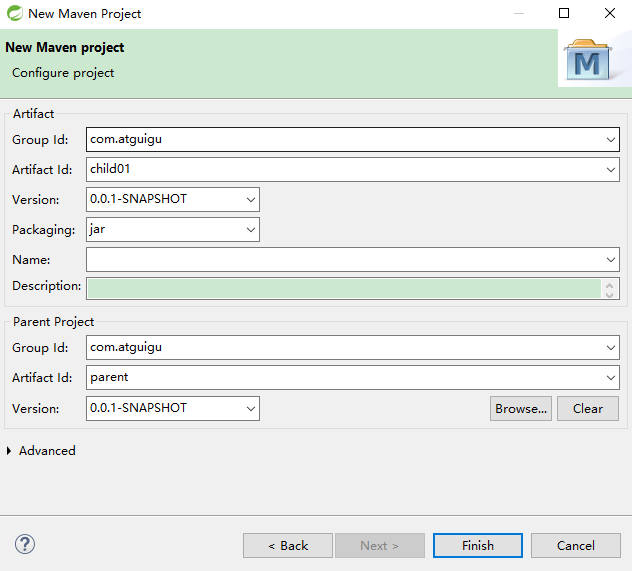
<groupId>com.atguigu</groupId>

<artifactId>parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

或者在创建向导中，直接选择父项目的GAV坐标：



如果继承了父项目，那么子项目的groupId和version可以省略！

如果当前的父工程，并没有安装在本地仓库中，那么必须指定当前父工程的<relativePath>来指定相对路径，子工程才可以运行Maven命令！

<parent>

<groupId>com.atguigu</groupId>

<artifactId>parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<relativePath>../parent/pom.xml</relativePath>

</parent>

### 1.4 父子项目的依赖管理

子父工程之间，是继承关系，而不是依赖关系。子工程可以选择父工程中的所需要的Jar包。而没有必要全部继承！

在父工程的pom文件中，将所有的依赖信息，加入到<dependencyManagement>标签中：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

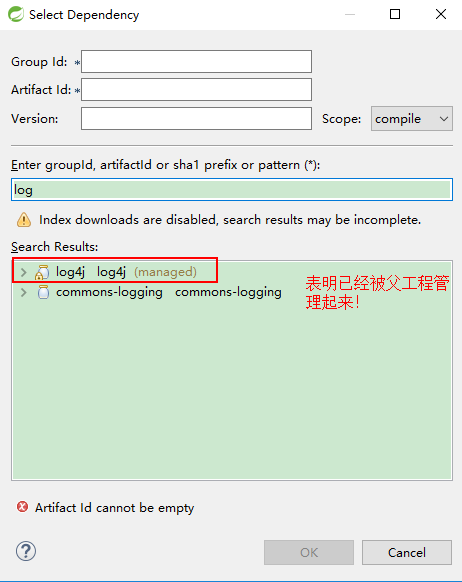
<version>${spring.verson}</version>

</dependency>

</dependencies>

</dependencyManagement>

子模块需要哪些Jar包，只需要在自己的pom文件中声明即可：



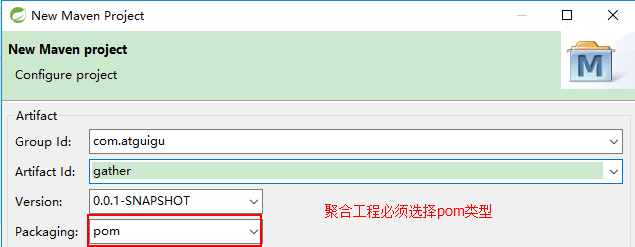
## 2. 聚合

### 2.1 聚合的目的

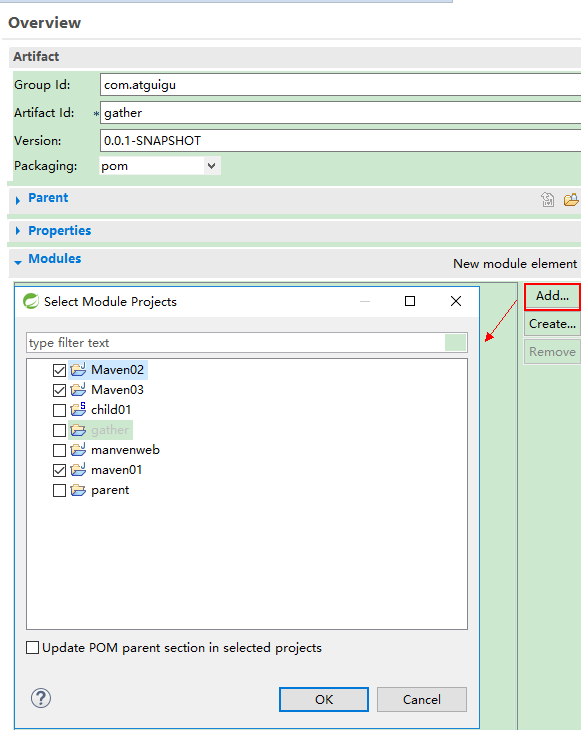
将多个工程拆分为模块后，需要手动逐个安装到仓库后依赖才能够生效。修改源码后也需要逐个手动进行clean操作。可以把依赖关系复杂的聚合在一个项目里面，统一运行聚合项目，聚合项目可以自动批量地进行清理和安装等操作！

### 2.2 如何配置聚合

①创建一个聚合工程，类型为pom



②在聚合工程的pom文件中，指定聚合的子工程：



生成的pom文件的信息如下：

<modules>

<module>../Maven02</module>

<module>../Maven03</module>

<module>../maven01</module>

</modules>

之后，在聚合工程上运行Maven命令，便可以对里面配置的模块，进行批处理，而无需关心他们之间的依赖关系！