# maven概述

## 引入

### 为什么要使用Maven?它能帮助我们解决什么问题？

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/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，让我们的工程显得很臃肿。而使用Maven后每个jar包本身只在本地仓库中保存一份，需要jar包的工程只需要以坐标的方式简单的引用一下就可以了。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.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-fileuplpad-1.3.jar依赖于commons-io-2.0.1.jar，如果没有10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。

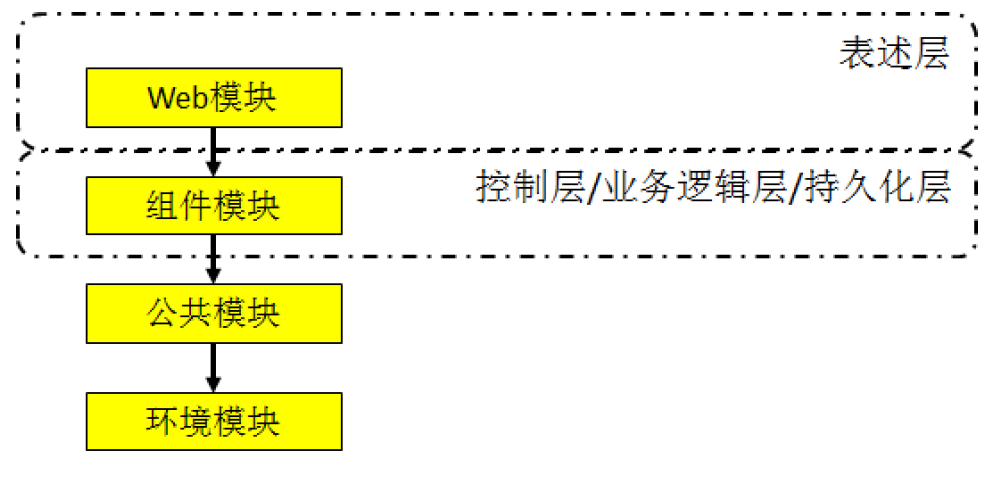
3.获取第三方jar包

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。

4.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。

## 定义

### Maven是什么?

Maven是Apache 软件基金会组织维护的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。Maven这个单词的本意是：专家，内行。读音是[merv(e)n]域['mevn]。

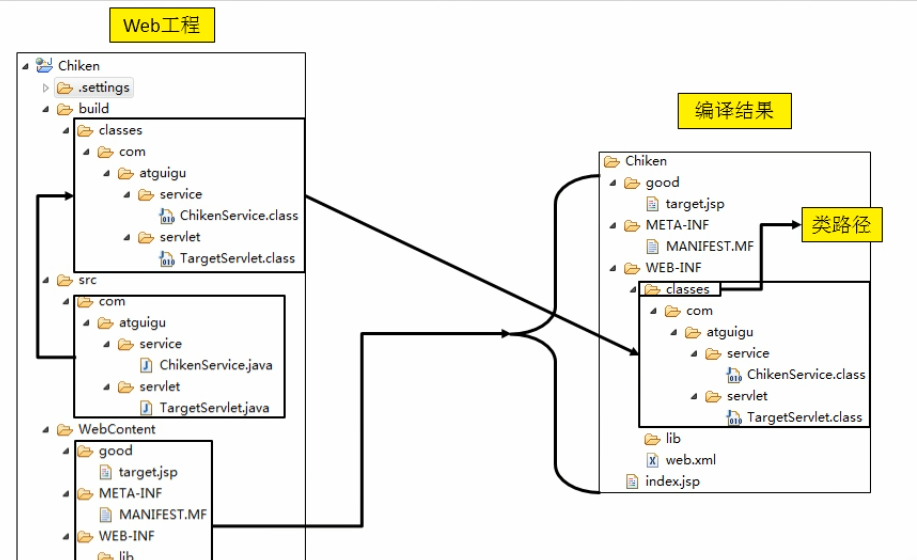
### 构建

概念：以"Java源文件”、“框架配置文件”、“JSP”、"HTML"、“图片”等资源为“原材

料”，去“生产”一个可以运行的项目的过程。

编译：java源文件[User.java]>编译>Class字节码文件[.class文件]>交给JVM运行。

部署：一个BS项目最终运行的并不是动态Web工程本身，而是这个动态Web工程“编译的结果"。



开发过程中，所有的路径或配置文件中配置的类路径等都是以编译结果的目录结构为标准的。

在实际项目中整合第三方框架，Web工程中除了Java程序和JSP页面、图片等静态资源之外，还包括第三方框架的jar包以及各种各样的配置文件。所有这些资源都必须按照正确的目录结构部署到服务器上，项目才可以运行。

所以综上所述：构建就是以我们编写的Java代码、框架配置文件、国际化等其他资源文件、JSP页面和图片等静态资源作为“原材料”，去“生产”出一个可以运行的项目的过程。

3，Maven 作为 Apache 的一个开源项目，旨在给项目管理提供更多的支持，它最早的意图只是为了给 apache 组织的几个项目提供统一的开发、测试、打包和部署，能让开发者在多个项目中方便的切换。

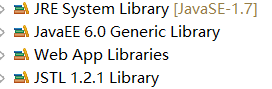
4，Maven 中最值得称赞的地方就是使用了标准的目录结构和部署。

5，在多个开发团队环境的情况下，Maven可以设置在上班的路上在很短的时间内为标准。由于大部分的项目设置简单可重复使用，Maven的生活的开发容易，而创建报告，检查，生产和测试的自动化设置。

6，maven是一个项目构建和管理的工具，提供了帮助管理 构建、文档、报告、依赖、scms、发布、分发的方法。可以方便的编译代码、进行依赖管理、管理二进制库等等。

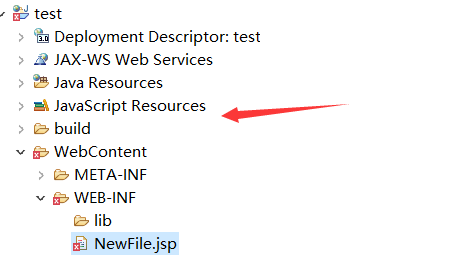
7，maven的好处在于可以将项目过程规范化、自动化、高效化以及强大的可扩展性，利用maven自身及其插件还可以获得代码检查报告、单元测试覆盖率、实现持续集成等等。

8、运行时环境



其实是一组jar包的引用，并没有把jar包本身复制到工程中，所以并不是目录。

例如：



这里没有添加tomcat的运行时环境，jsp会报错。

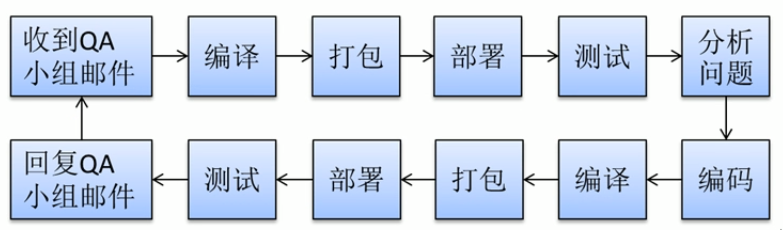
### 自动化构建

其实上述环节我们在Eclipse中都可以找到对应的操作，只是不太标准。那么既然IDE已经可以进行构建了我们为什么还要使用Maven这样的构建工具呢？我们来看一个小故事：

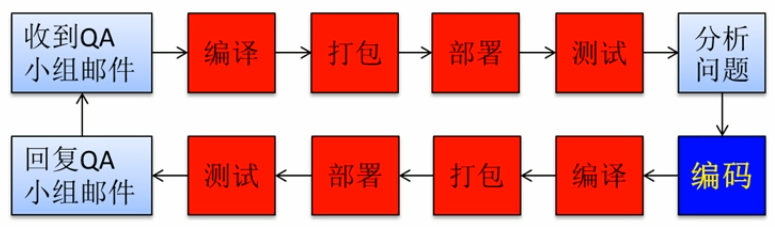
这是阳光明媚的一天。托马斯向往常一样早早的来到了公司，冲好一杯咖啡，进入了自己的邮箱一—很不幸，QA小组发来了一封邮件，报告了他昨天提交的模块的测试结果——有BUG。“好吧，反正也不是第一次”，托马斯摇摇头，进入IDE，运行自己的程序，编译、打包、部署到服务器上，然后按照邮件中的操作路径进行测试。“嗯，没错，这个地方确实有问题”，托马斯说道。于是托马斯开始尝试修复这个BUG，当他|差不多有眉目的时候已经到了午饭时间。

下午继续工作。BUG很快被修正了，接着托马斯对模块重新进行了编译、打包、部署，测试之后确认没有问题了，回复了QA小组的邮件。

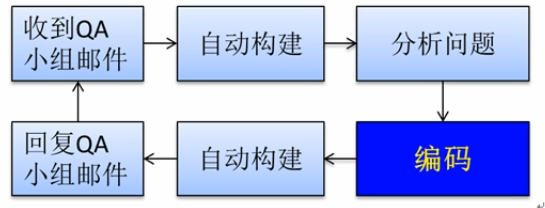
一天就这样过去了，明媚的阳光化作了美丽的晚霞，托马斯却觉得生活并不像晚霞那样美好啊。



从中我们发现，托马斯的很大一部分时间花在了“编译、打包、部署、测试”这些程式化的工作上面，而真正需要由“人”的智慧实现的分析问题和编码却只占了很少一部分。



能否将这些程式化的工作交给机器自动完成呢?——当然可以！这就是自动化构建。



此时Maven的意义就体现出来了，它可以自动的从构建过程的起点一直执行到终点：

### 构建环节

清理：将以前编译得到的旧的class字节码文件删除，为下一次编译做准备。

编译：将Java源程序编程成class字节码文件。

测试：自动测试，自动调用junit程序。编写代码时，maven会自动调用测试类，不需要手动调用。

报告：测试程序执行的结果。

打包：动态Web工程打war包，Java工程打jar包。

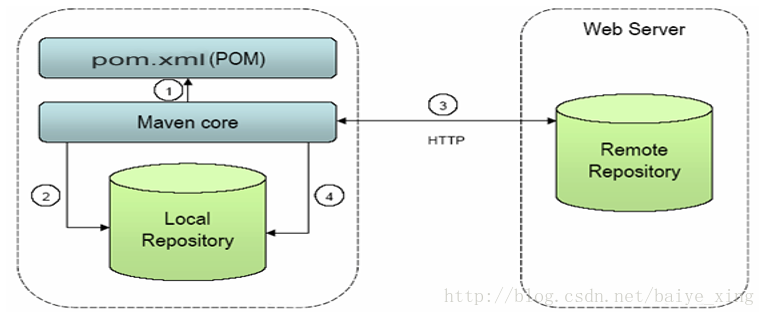
安装：Maven特定的概念——将打包得到的文件复制到“仓库”中的指定位置。

部署：将动态Web工程生成的war包复制到Servlet容器的指定目录下，使其可以运行。

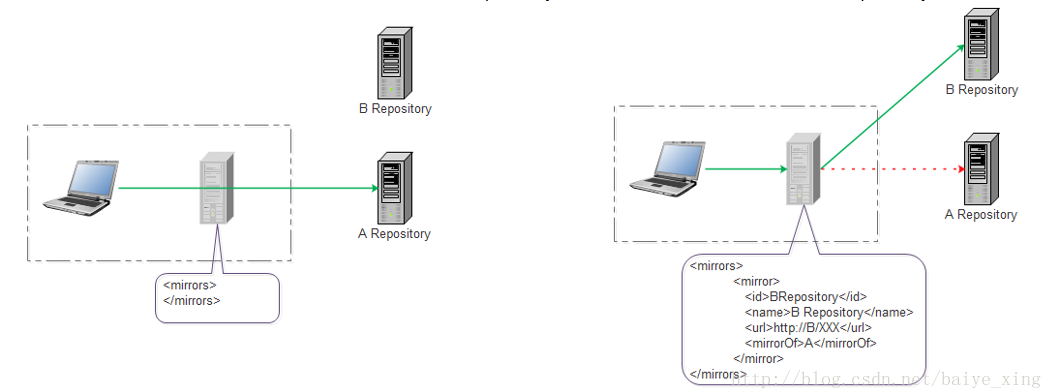
## 基本原理

Maven 的基本原理很简单，采用远程仓库和本地仓库以及一个类似 build.xml 的 pom.xml ，将 pom.xml 中定义的 jar 文件从远程仓库下载到本地仓库，各个应用使用同一个本地仓库的 jar ，同一个版本的 jar 只需下载一次，而且避免每个应用都去拷贝 jar 。

同时它采用了现在流行的插件体系架构，只保留最小的核心，其余功能都通过插件的形式提供，所以 maven 下载很小，在执行 maven 任务时，才会自动下载需要的插件。



mirror相当于一个拦截器，它会拦截maven对remote repository的相关请求，把请求里的remote repository地址，重定向到mirror里配置的地址。



此时，B Repository被称为A Repository的镜像。如果仓库X可以提供仓库Y存储的所有内容，那么就可以认为X是Y的一个镜像。换句话说，任何一个可以从仓库Y获得的构件，都能够从它的镜像中获取。

< mirrors/>是镜像列表，是maven从远程仓库里下载构件的一组服务器镜像。镜像能减轻中央maven库的负载，也能突破代理等的网络环境的限制，每个仓库都有一个ID，而mirror需要和仓库的ID对应。

## maven核心概念

### maven的核心概念

POM

约定的目录结构

坐标

依赖管理

仓库管理

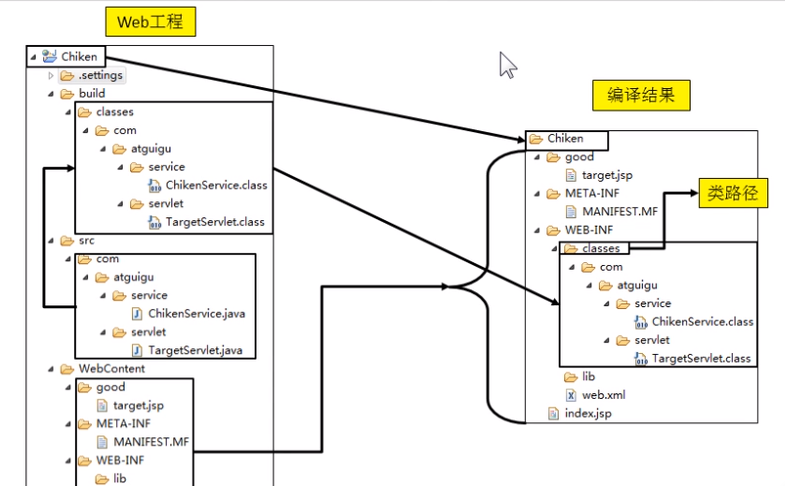
生命周期

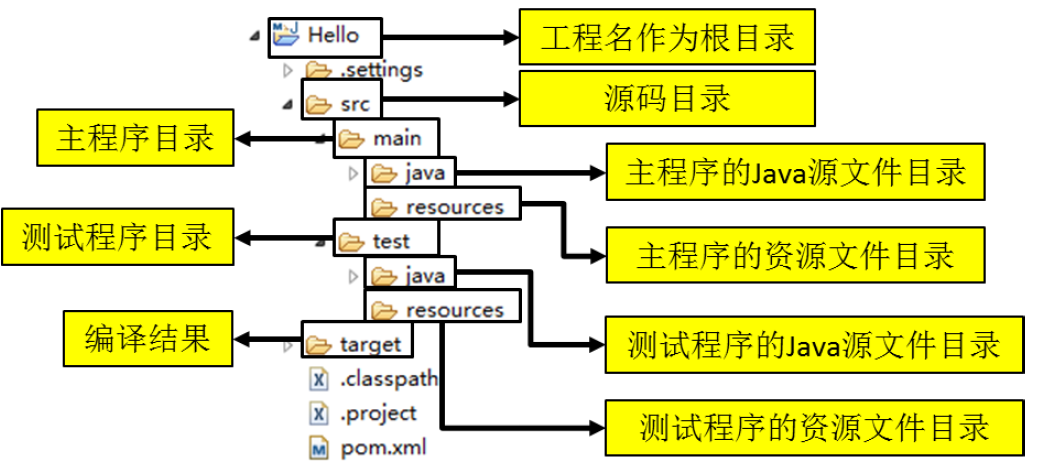
插件和目标

继承

聚合

### 约定的目录结构





根目录：工程名

src目录：源码

pom.xml文件：Maven工程的核心配置文件

main目录：存放主程序

test目录：存放测试程序

java目录：存放Java源文件

resources目录：存放框架或其他工具的配置文件

为什么要遵守约定的目录结构呢？

Maven要负责我们这个项目的自动化构建，以编译为例，Maven要想自动进行编译，那么它必须知道Java源文件保存在哪里。如果我们自己自定义的东西想要让框架或工具知道，有两种办法：

1、以配置的方式明确告诉框架。

2、遵守框架内部已经存在的约定。

开发过程中，所有的路径或配置文件中配置的类路径等都是以编译结勇的目录结构为标准的。

### pom

Project Object Model:项目对象模型。将Java工程的相关信息封装为对象作为便于操作和管理的模型。Maven工程的核心配置。可以说学习Maven就是学习pom.xml文件中的配置。

### 常用Maven命令

①注意：执行与构建过程相关的Maven命令，必须进入pom.xml所在的目录。

与构建过程相关：编译、测试、打包、….

命令：[1]mvn clean：清理，将以前编译得到的旧的 class 字节码文件删除

[2]mvn compile：编译主程序，会生成一个target目录，存放编译的程序

[3]mvn test-compile；编译测试程序，会生成一个target目录，存放编译的程序

[4]mvn test:执行测式

[5]mvn package:打包

### Maven联网的问题

①Maven的核心程序中仅仅定义了抽象的生命周期，但是具体的工作必须由特定的插件来完成。而插件本身并不包含在Maven的核心程序中。

②当我们执行的Maven命令需要用到某些插件时，Maven核心程序会首先到本地仓库中查找。

③本地仓库的默认位置：[系统中当前用户的家目录]\.m2\\repository

C:\Users\jj\.m2\repository

④Maven核心程序如果在本地仓库中找不到需要的插件，那么它会自动连接外网，到中央仓库下载。

⑤如果此时无法连接外网，则构建失败。

⑥修改默认本地仓库的位置可以让Maven核心程序到我们事先准备好的目录下查找插件

[1]找到Maven解压目录\conf\settings.xml

[2]在settings.xml文件中找到localRepository标签

[3]找到第53行，把注释去掉，修改成已准备好的仓库目录：

<localRepository>仓库路径</localRepository>

### POM

Project object Model：项目对象模型。将Java工程的相关信息封装为对象作为便于操作和管理的模型。Maven工程的核心配置。可以说学习Maven就是学习pom.xml文件中的配置。

①含义：Project Object Model项目对象模型

DOM Document Object Model 文档对象模型

②pom.xml对于Maven工程是核心配置文件，与构建过程相关的一切设置都在这个文件中进行配置。重要程度相当于web.xml对于动态Web工程。

### 坐标

几何中的坐标

[1]在一个平面中使用x、y两个向量可以唯一的确定平面中的一个点。

[2]在空间中使用x、y、z三个向量可以唯一的确定空间中的一个点。

Maven(GAV)的坐标: 使用如下三个向量在Maven的仓库中唯一的确定一个Maven工程。

[1]groupid:公司或组织的域名倒序+当前项目名称。

|  |
| --- |
| <groupld>com.sx.maven</groupld> |

[2]artifactld:当前项目的模块名称。

|  |
| --- |
| <artifactld>Hello</artifactld> |

[3]version:当前模块的版本。

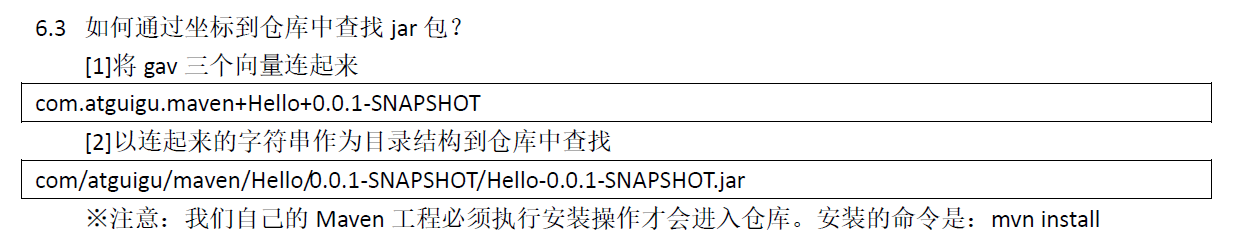
|  |
| --- |
| <version>1.1.0</versioh> |

管理jar包统一的方式

定位: groupld+ artifactld+ version+ artifactld- version.jar

com.sx.maven.Hello. 1.1.0. Hello-1.1.0.jar

如果通过坐标找到查找jar包？



### 仓库

1、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

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。

2、仓库中的文件：maven工程文件。

[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。

## 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：

①Clean Lifecycle在进行真正的构建之前进行一些清理工作。

②Default Lifecycle构建的核心部分，编译，测试，打包，安装，部署等等。

③Site Lifecycle生成项目报告，站点，发布站点。

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。

Clean生命周期

Clean生命周期一共包含了三个阶段：

①pre-clean 执行一些需要在clean之前完成的工作

②clean 移除所有上一次构建生成的文件

③post-clean执行一些需要在clean之后立刻完成的工作

Site生命周期

①pre-site执行一些需要在生成站点文档之前完成的工作

②site生成项目的站点文档

③post-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

④site-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。

Default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：

Maven核心程序为了更好的实现自动化构建，按照这一的特点执行生命周期中的各个阶段：不论现在要执行生命周期中的哪一个阶段，都是从这个生命周期最初的位置开始执行。

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。

插件和目标

[1]生命周期的各个阶段仅仅定义了要执行的任务是什么。

[2]各个阶段和插件的目标是对应的。

[3]相似的目标由特定的插件来完成。

## 坐标

### 定义

Maven的一个核心的作用就是管理项目的依赖，引入我们所需的各种jar包等。为了能自动化的解析任何一个Java构件，Maven必须将这些Jar包或者其他资源进行唯一标识，这是管理项目的依赖的基础，也就是我们要说的坐标。包括我们自己开发的项目，也是要通过坐标进行唯一标识的，这样才能才其它项目中进行依赖引用。

几乎任何Java应用都会借用一些第三方的开源类库，这些类库都可通过依赖的方式引入到项目中来。随着依赖的增多，版本不一致、版本冲突、依赖雕肿等问题都会接踵而来。手工解决这些问题是十分枯燥的，幸运的是Maven提供了一个优秀的解决方案，它通过一个坐标系统准确地定位每一个构件（artifact)，也就是通过一组坐标Maven能够找到任何一个Java类库（如jar文件）。Maven给这个类库世界引入了经纬，让它们变得有秩序，于是我们可以借助它来有序地管理依赖，轻松地解决那些繁杂的依赖问题。

坐标用来标识时空中的某个点，方便人们找到位置，如中电信息大厦可以用经纬度坐标找到，也可以通过国家、省市区、街道、门牌组成的坐标去找。

坐标就好比每一个Java构件的身份证一样。Maven的世界是拥有非常多的Java构件的，可能是jar、可能是war、也可能是其他的一些东西。假如Maven中没有坐标的概念，我们是无法来区分这些构件的，所以我们要唯一标识每一个构件。不然就和传统的手工方式一样，你需要spring就去spring下载spring的包，需要mysql又去mysql下载mysql的包，没有统一的规范又怎么能够自动化的去依赖这些构件。

Maven便给我们制定了一套规则那就是使用坐标进行唯一标识。Maven的坐标元素包括groupId、artifactId、version、packaging、classfier。只要我们提供正确的坐标元素，Maven就能找到对应的构件，首先去你的本地仓库查找，没有的话再去远程仓库下载。如果没有配置远程仓库，会默认从中央仓库地址(http://repo1.maven.org/maven2)下载构件，该中央仓库包含了世界上大部分流行的开源项目构件，但不一定所有构件都有，我在以前的开发中就遇到过找不到oracle数据库jar包的问题，不知道现在是否依然如此。

### 坐标详细说明

groupId: 组织ID，一般是公司、团体名称

artifactId：实际项目的ID，一般是项目、模块名称

version:版本，开发中的版本一般打上 SNAPSHOT 标记

Type/packaging :包类型，如JAR,EAR,POM…

classifier:分类，如二进制包、源、文档

通过这个规则就可以定位到世界上任何一个构件。

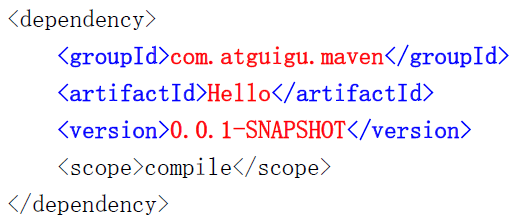
## 依赖

Maven中最关键的部分，我们使用Maven最主要的就是使用它的依赖管理功能。要理解和掌握Maven的依赖管理，我们只需要解决一下几个问题：

### 依赖的目的是什么

当Ajar包用到了Bjar包中的某些类时，A就对B产生了依赖，这是概念上的描述。那么如何在项目中以依赖的方式引入一个我们需要的jar包呢？

答案非常简单，就是使用dependency标签指定被依赖jar包的坐标就可以了。

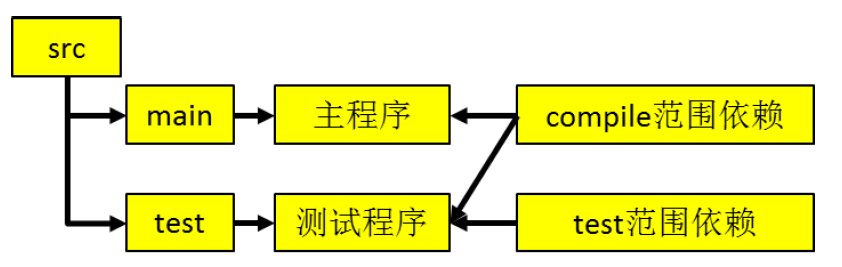




### 依赖的范围

大家注意到上面的依赖信息中除了目标jar包的坐标还有一个scope设置，这是依赖的范围。依赖的范围有几个可选值，我们用得到的是：compile、test、provided三个。

[1]从项目结构角度理解compile和test的区别

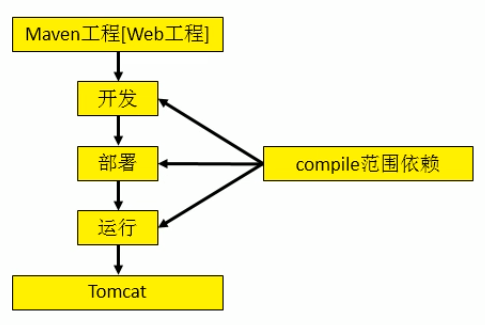


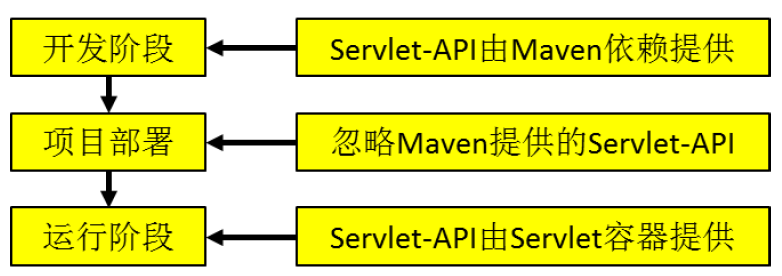
结合具体例子：对于HelloFriend来说，Hello就是服务于主程序的，junit是服务于测试程序的。

HelloFriend 主程序需要He11o是非常明显的，测试程序由于要调用主程序所以也需要Hel1o，所以compile 范围依赖对主程序和测试程序都应该有效。

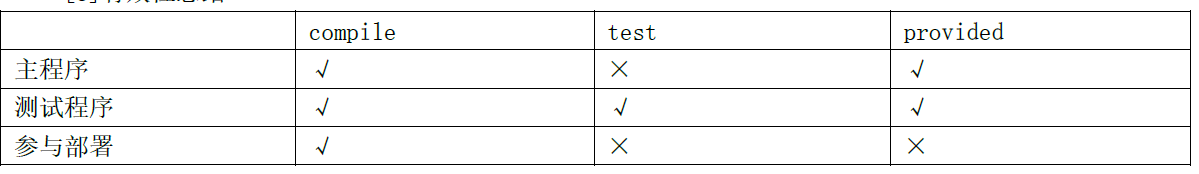
He1loFriend的测试程序部分需要junit也是非常明显的，而主程序是不需要的，所以test范围依赖

[2]从开发和运行这两个不同阶段理解compile和provided的区别





对程序影响的有效性



[1]complie范围依赖

·对主程序是否有效：有效

对测试程序是否有效：有效

·是否参与打包：参与

·是否参与部署：参与

[2]test范围依赖

·对主程序是否有效：无效

对测试程序是否有效：有效

·是否参与打包：不参与

·是否参与部署：不参与

·典型例子：junit

[3]provided范围依赖

对主程序是否有效：有效

对测试程序是否有效：有效

·是否参与打包：不参与

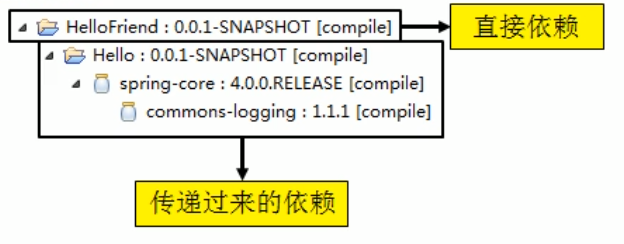
·是否参与部署：不参与

·典型例子：servlet-api.jar

例如：

|  |
| --- |
| <dependencies>  <!-- servlet -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>${servlet.version}</version>  <scope>provided</scope>//部署项目时不会复制该jar包  <scope>test</scope>//不参与打包，不参与部署，也不会参与主程序。  </dependency>  <!-- spring -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aspects</artifactId>  <version>${spring.version}</version>  //这里并没有scope，默认范围为complie，会将该jar包放到服务器项目的lib下。  </dependency> |

### 依赖的传递性（高级）



优点：可以传递的依赖不必在每个模块工程中都重复声明,在”最下面”的工程中依赖一次即可。

注：非compile的依赖不能够传递，也就是指相对于当前的maven工程，可以使用。所以在各个工程中，如果有需要就得重复声明依赖。

A 依赖 B，B依赖 C，A 能否使用C 呢？那要看B依赖 C 的范围是不是 compile，如果是则可用，否则不可用。

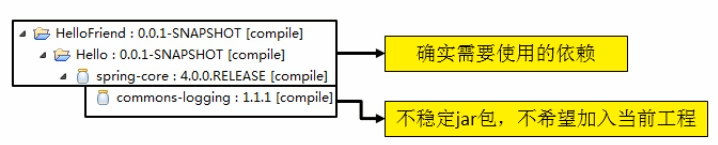


### 依赖的排除

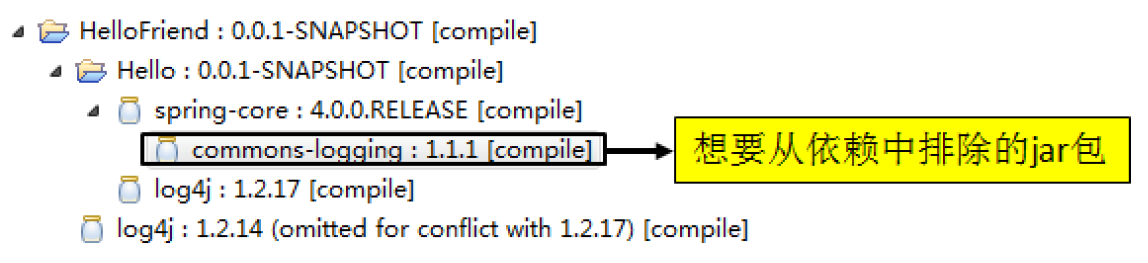
如果我们在当前工程中引入了一个依赖是A，而A又依赖了B，那么Maven会自动将A依赖的B引入当前工程，但是个别情况下B有可能是一个不稳定版，或对当前工程有不良影响。这时我们可以在引入A的时候将B排除。

注：如果C项目引入A项目,A项目被排除的包，C项目中也不会出现。

需要设置依赖排除的场合：



依赖排除的设置方式：由于commons-logging是一个不稳定的版本，想要将commons-logging排除掉。



|  |
| --- |
| <dependency>  <groupId>com.atguigu.maven</groupId>  <artifactId>HelloFriend</artifactId>//导入的maven项目  <version>0.0.1-SNAPSHOT</version〉  <type>jar</type>  <scope>compile</scope〉  <exclusions>  <exclusion>  <groupId>commons-logging</groupId>  <artifactId> commons-logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

效果：



### 依赖的原则：解决jar包冲突问题

HelloFrient和hello项目中同时导入不同版本的log4j。



传递方式：路径最短优先

MakeFriend会选择依赖路径最短的jar包，这是maven默认的方式，当然如果想要使用hello下版本的jar包，可以在pom.xml中自己声明下这个jar包。

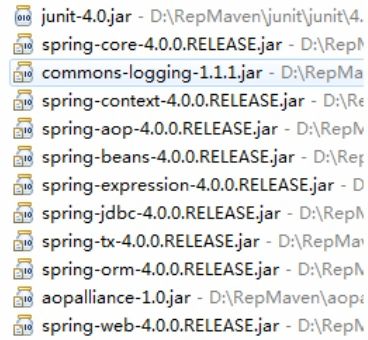
如果依赖的路径相同：



此时，先声明的dependency的标签，先被引用。

### 统一管理依赖的版本

需求：spriing的jar包都为4.0，如何统一升级版本。



如果手动去改，操作繁琐并且容易出现错误。

建议配置方式：

|  |
| --- |
| <!-- jar包版本声明 -->  <properties>  <servlet.version>3.1.0</servlet.version>  </properties>  <dependencies>  <!-- servlet -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>${servlet.version}</version>  <scope>provided</scope>  </dependency>  </dependencies |

注：其实properties标签配合自定义标签声明数据的配置并不是只能用于声明依赖的版本号。凡是需要统一声明后再引用的场合都可以使用。

## 继承

情景：

Hello依赖的junit:4.0

HelloFriend依赖的junit:4.0

MakeFriends依赖的junit:4.9

由于test范围的依赖不能传递，所以必然会分散在各个模块工程中，很容易造成版本不一致。

需求：统一管理各个模块工程中对junit依赖的版本需求：

解决思路：将junit依赖统一提取到“父”工程中，在子工程中声明junit依赖时不指定版本，以父工程中统一设定的为准。同时也便于修改。

|  |  |
| --- | --- |
| hello | <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.0</version>  <scope>test</scope>  </dependency> |
| helloFriend | <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.0</version>  <scope>test</scope>  </dependency> |
| MakeFriend | <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.0</version>  <scope>test</scope>  </dependency> |

为什么需要继承？

由于非compile范围的依赖信息是不能在“依赖链”中传递的，所以有需要的工程只能单独配置。

此时如果项目需要将各个模块的junit版本统一为4.9，那么到各个工程中手动修改无疑是非常不可取的。使用继承机制就可以将这样的依赖信息统一提取到父工程模块中进行统一管理。

操作步骤

[1]创建一个Maven工程作为父工程。注意：打包的方式pom

[2]在子工程中声明对父工程的引用

[3]将子工程的坐标中与父工程坐标中重复的内容删除

[4]在父工程中统一junit的依赖

[5]在子工程中删除junit依赖的版本号部分

### 建立maven\_parent父项目

项目的packaging：pom

Packaging为pom的项目，不用来开发java代码。用来声明整个系统中的所需要依赖的资源。

|  |
| --- |
| <modelVersion>4.0.0</modelVersion>  <groupId>com.sxt</groupId>  <artifactId>BM\_parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>pom</packaging> |

### 创建子项目，并在系项目中引用父项目

|  |
| --- |
| <modelVersion>4.0.0</modelVersion>  ~~<groupId>com.sxt</groupId>~~ //该工程的groupid与父工程相同，可以删除。  在子工程中声明父工程，配置继承后，执行安装命令后先安装父工程。  <parent>  <groupId>com.sxt</groupId>  <artifactId>BM\_parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  <!-- 指定从当前子工程的pom.xml文件出发，查找父工程的pom.xml的路径 -->  <relativePath>../BM\_parent/pom.xml</relativePath>  </parent>  <artifactId>BM\_entity</artifactId> |

### 父工程中统一配置junit依赖的管理

|  |
| --- |
| <dependencyManagement> //依赖管理  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  </dependencies>  </dependencyManagement> |

### 子工程对junit的配置

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  ~~<version>4.12</version>~~  <scope>test</scope>  </dependency> |

## 聚合

### 为什么要使用聚合？

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。而使用了聚合之后就可以批量进行Maven工程的安装、清理工作。

### 如何配置聚合？

在总的聚合工程中使用modules/module标签组合，指定模块工程的相对路径即可

|  |
| --- |
| <modules>  //指定各个自工程的相对路径  <module>BM\_mapper</module>  <module>BM\_entity</module>  <module>BM\_service</module>  <module>BM\_web</module>  </modules> |

### 操作

右键父工程的pom.xml,run as--maven install

## jar和war

### jar包

JAR（Java Archive，Java 归档文件）是与平台无关的文件格式，它允许将许多文件组合成一个压缩文件。JavaSE程序可以打包成Jar包(J其实可以理解为Java了)。

JAR 文件格式以流行的 ZIP 文件格式为基础。与 ZIP 文件不同的是，JAR 文件不仅用于压缩和发布，而且还用于部署和封装库、组件和插件程序，并可被像编译器和 JVM 这样的工具直接使用。在 JAR 中包含特殊的文件，如 manifests 和部署描述符，用来指示工具如何处理特定的 JAR。

简单来说，jar包就是别人已经写好的一些类，然后对这些类进行打包。可以将这些jar包引入到你的项目中，可以直接使用这些jar包中的类和属性，这些jar包一般放在lib目录下。

### war包

war(Web application Archive)是一个可以直接运行的web模块，通常用于网站，打成包部署到容器中。以Tomcat来说，将war包放置在其\webapps\目录下，然后启动Tomcat，这个包就会自动解压，就相当于发布了。

war包是Sun提出的一种web应用程序格式，与jar类似，是很多文件的压缩包。war包中的文件按照一定目录结构来组织。根据其根目录下包含有html和jsp文件，或者包含有这两种文件的目录，另外还有WEB-INF目录。通常在WEB-INF目录下含有一个web.xml文件和一个classes目录，web.xml是这个应用的配置文件，而classes目录下则包含编译好的servlet类和jsp，或者servlet所依赖的其他类（如JavaBean）。通常这些所依赖的类也可以打包成jar包放在WEB-INF下的lib目录下。

简单来说，war包是JavaWeb程序打的包，war包里面包括写的代码编译成的class文件，依赖的包，配置文件，所有的网站页面，包括html，jsp等等。一个war包可以理解为是一个web项目，里面是项目的所有东西。

## 特点

1，依赖管理是maven的一大特征，对于一个简单的项目，对依赖的管理并不是什么困难的事，但是如果这个项目依赖的库文件达到几十个甚至于上百个的时候就不是一个简单的问题了。在这个时候maven对于依赖管理的作用就显露出来了。

2，传递性依赖是在maven2中添加的新特征，这个特征的作用就是你不需要考虑你依赖的库文件所需要依赖的库文件，能够将依赖模块的依赖自动的引入。

3，由于没有限制依赖的数量，如果出现循环依赖的时候会出现问题，这个时候有两种方式处理，一种是通过 build-helper-maven-plugin 插件来规避，另一种就是重构两个相互依赖的项目。

4，通过传递性依赖，项目的依赖结构能够很快生成。Maven 能够解决依赖传递

5，传递依赖中需要关注的就是依赖调解，依赖调解的两大原则是：最短路径优先和第一声明优先

6，maven有三套classpath（编译classpath，运行classpath，测试classpath）分别对应构建的三个阶段。依赖范围就是控制依赖与这三套classpath的关系。

7，依赖范围有六种：

## 在Eclipse中使用Maven

①Maven插件：Eclipse内置

②Maven插件的设置

[1]installations:指定Maven核心程序的位置。不建议使用插件自带的Maven程序，而应该使用我们自己解压的那个。

[2]user settings:指定conf/settings.xml的位置，进而获取本地仓库的位置。

③基本操作

[1]创建Maven版的Java工程

[2]创建Maven版的Web工程

[3]执行Maven命令

# Maven 安装(win)

## maven安装

### 安装jdk并配置

因为maven是java项目，首先要有jdk

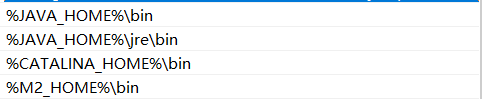
### 解压maven文件夹

### 配置maven的相关环境变量

1、配置MAVEN\_HOME 或 M2\_HOME

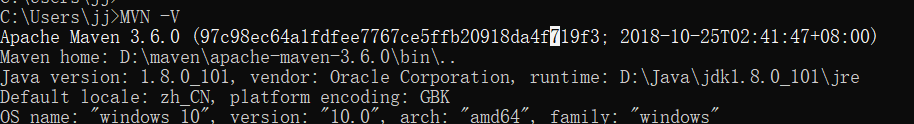


PATH：



### 验证

查看版本信息：mvn -v



### 配置仓库

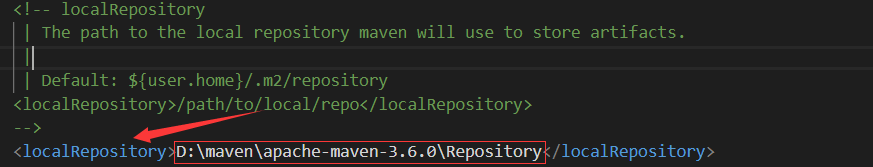
修改默认本地仓库的位置可以让Maven核心程序到我们事先准备好的目录下查找插件

[1]找到Maven解压目录\conf\settings.xml

[2]在settings.xml文件中找到localRepository标签

[3]找到第53行，把注释去掉，修改成已准备好的仓库目录：

<localRepository>仓库路径</localRepository>



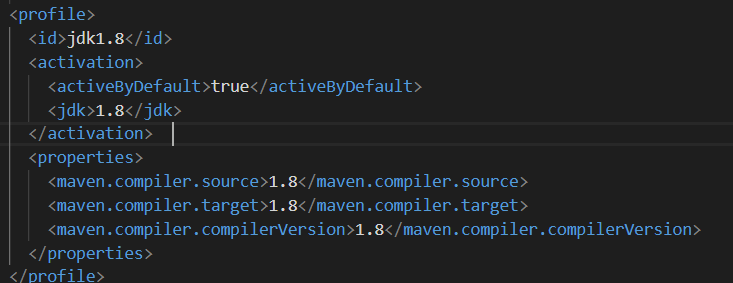
## Maven项目的JDK配置

在setting.xml中配置

目的:是让每次创建的maven项目都保持jdk1.8，否则会默认使用jdk1.5

默认使用jdk1.8

|  |
| --- |
| <profile>  <id>jdk1.8</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> |



## 下载maven服务器

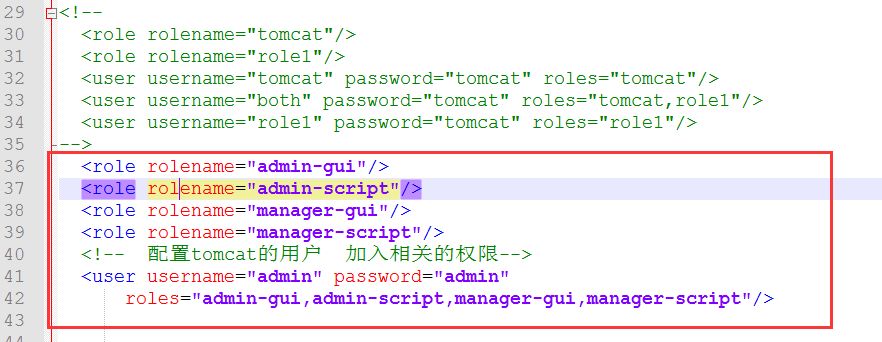
下载地址 <http://maven.apache.org/download.cgi>



## 自动部署项目

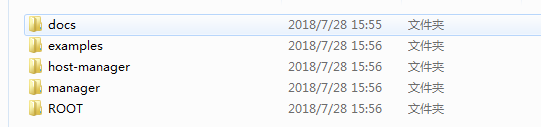
通过maven命令把项目打包，打包完成之后放到外部的tomcat里面去部署

### 配置tomcat的账户conf/ tomcat-users.xml

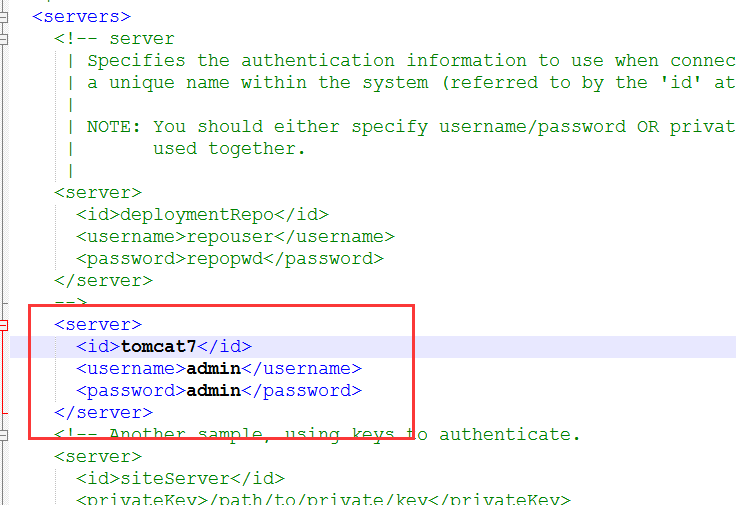


|  |
| --- |
| <role rolename="admin-gui"/>  <role rolename="admin-script"/>  <role rolename="manager-gui"/>  <role rolename="manager-script"/>  <user username="admin" password="admin"  roles="admin-gui,admin-script,manager-gui,manager-script"/> |

### 确保存webapps下面的五个项目没有被删除



### 配置maven的settings.xml



|  |
| --- |
| <server>  <id>tomcat7</id>  <username>admin</username>  <password>admin</password>  </server> |

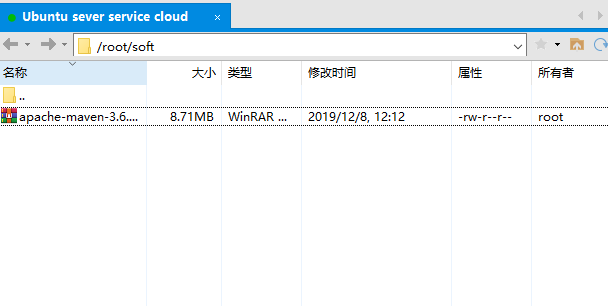
### 配置项目的pom.xml

|  |
| --- |
| <build>  <finalName>06\_carrent</finalName>  <!-- 配置插件 -->  <plugins>  <!--maven的tomcat插件 -->  <plugin> //启动servlet容器  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  <configuration>  <uriEncoding>UTF-8</uriEncoding> <!--解决页面提交数据乱码问题 -->  <port>8080</port><!-- tomcat插件的请求端口 -->  <path>/MB</path>  //使用自定义的tomcat去加载项目，必须将服务器开启。  <!-- <url>http://localhost:8080/manager/text</url>  <username>admin</username>  <password>admin</password>  <path>/aa</path> -->  </configuration>  </plugin>  <!-- 你前项目使用的JDK版本 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.7.0</version>  <configuration>  <!-- 指定source和target的版本 -->  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>  </build> |

# Maven 安装(linux)

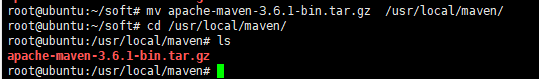
## maven下载

### 上传maven文件到服务器



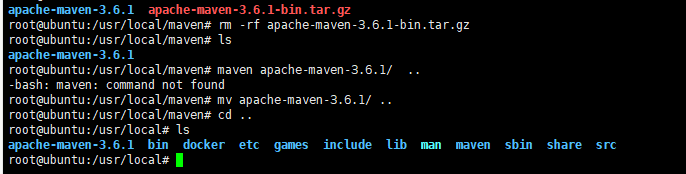
### 将当前文件复制到对应的目录中

/usr/local/maven



### 解压目录

解压目录，并复制到local中



### 配置环境环境变量

## maven安装

### 安装jdk并配置

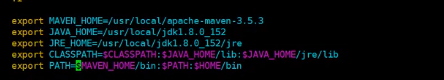
因为maven是java项目，首先要有jdk

### 配置maven的相关环境变量

vi /etc/profile 文件，加入下述代码

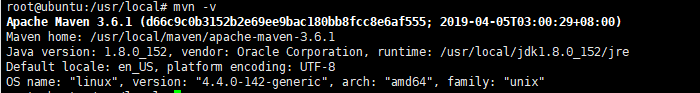
|  |
| --- |
| export MAVEN\_HOME=/usr/local/maven/apache-maven-3.6.1  export PATH=$MAVEN\_HOME/bin:$PATH:$HOME/bin |



重新加载配置文件source /etc/profile  


### 验证

查看版本信息：mvn -v



### 配置仓库

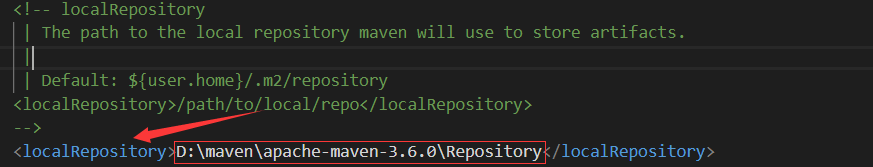
修改默认本地仓库的位置可以让Maven核心程序到我们事先准备好的目录下查找插件

[1]找到Maven解压目录\conf\settings.xml

[2]在settings.xml文件中找到localRepository标签

[3]找到第53行，把注释去掉，修改成已准备好的仓库目录：

<localRepository>仓库路径</localRepository>



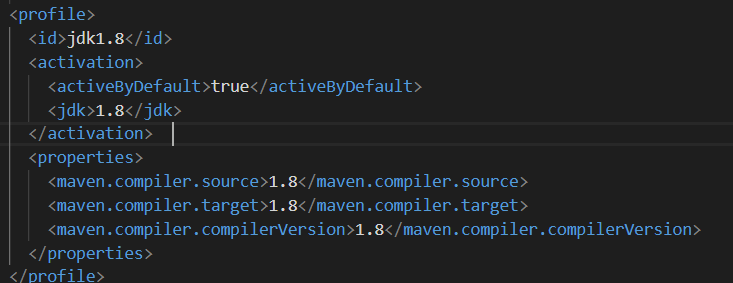
## Maven项目的JDK配置

在setting.xml中配置

目的:是让每次创建的maven项目都保持jdk1.8，否则会默认使用jdk1.5

默认使用jdk1.8

|  |
| --- |
| <profile>  <id>jdk1.8</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> |



## 下载maven服务器

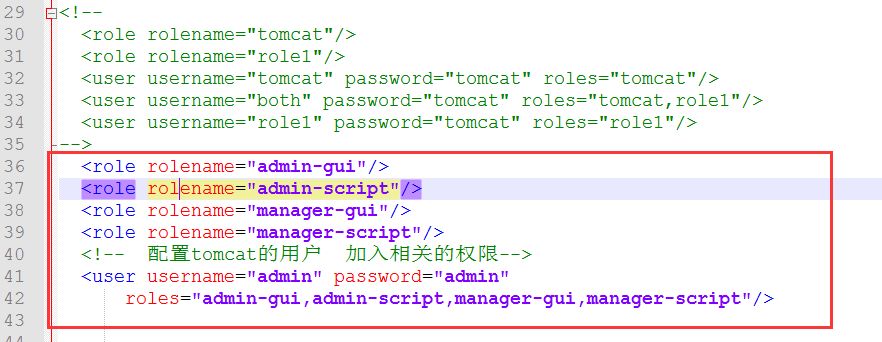
下载地址 <http://maven.apache.org/download.cgi>



## 自动部署项目

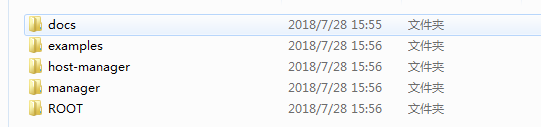
通过maven命令把项目打包，打包完成之后放到外部的tomcat里面去部署

### 配置tomcat的账户conf/ tomcat-users.xml

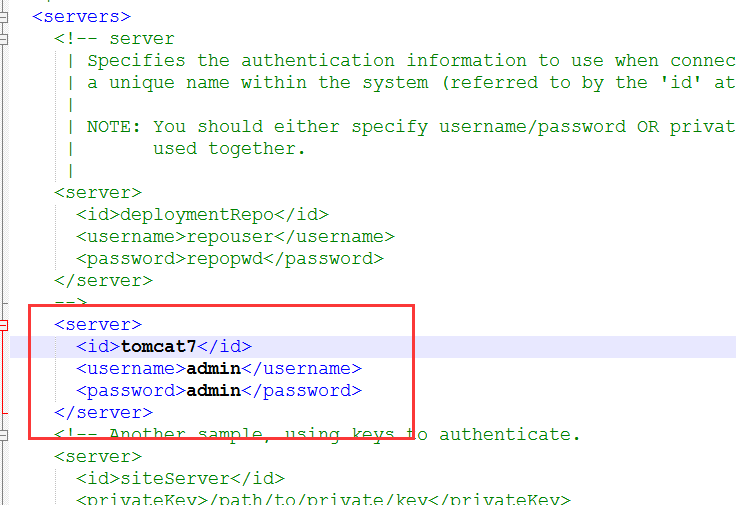


|  |
| --- |
| <role rolename="admin-gui"/>  <role rolename="admin-script"/>  <role rolename="manager-gui"/>  <role rolename="manager-script"/>  <user username="admin" password="admin"  roles="admin-gui,admin-script,manager-gui,manager-script"/> |

### 确保存webapps下面的五个项目没有被删除



### 配置maven的settings.xml



|  |
| --- |
| <server>  <id>tomcat7</id>  <username>admin</username>  <password>admin</password>  </server> |

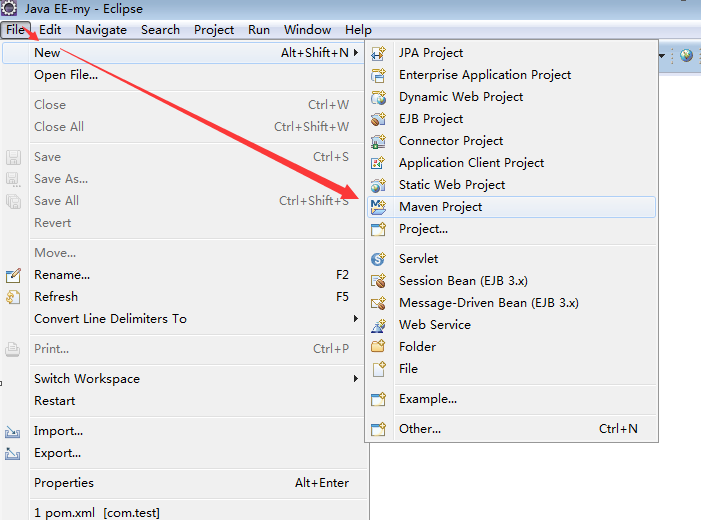
### 配置项目的pom.xml

|  |
| --- |
| <build>  <finalName>06\_carrent</finalName>  <!-- 配置插件 -->  <plugins>  <!--maven的tomcat插件 -->  <plugin> //启动servlet容器  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  <configuration>  <uriEncoding>UTF-8</uriEncoding> <!--解决页面提交数据乱码问题 -->  <port>8080</port><!-- tomcat插件的请求端口 -->  <path>/MB</path>  //使用自定义的tomcat去加载项目，必须将服务器开启。  <!-- <url>http://localhost:8080/manager/text</url>  <username>admin</username>  <password>admin</password>  <path>/aa</path> -->  </configuration>  </plugin>  <!-- 你前项目使用的JDK版本 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.7.0</version>  <configuration>  <!-- 指定source和target的版本 -->  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>  </build> |

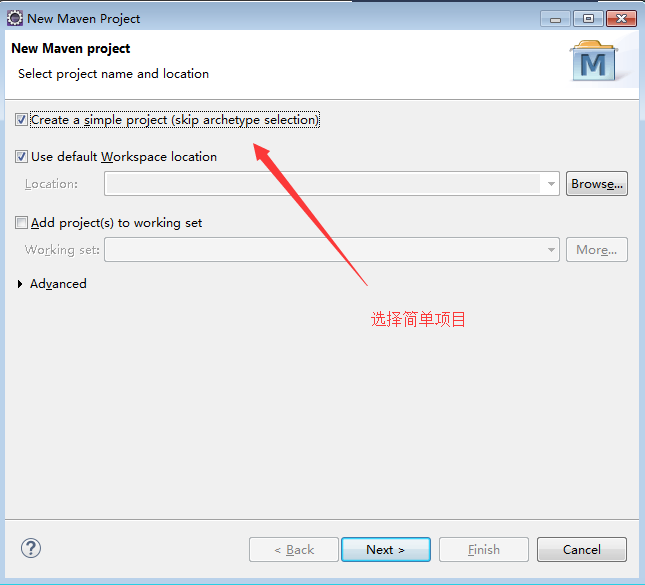
# 创建java工程

## 新建一个maven工程

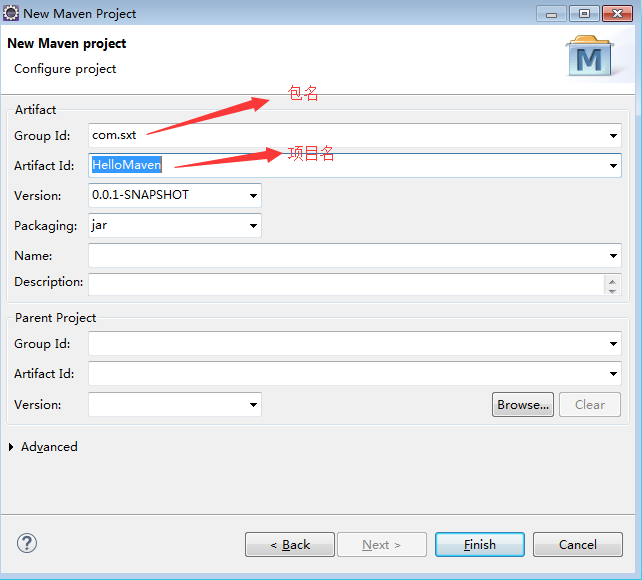
File-->new--other,然后如图所示，点击next



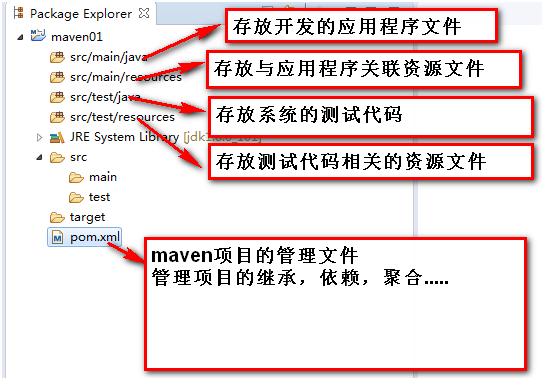
## 选择简单项目



## 填写包名和项目名



## Finish完成创建--项目目录说明

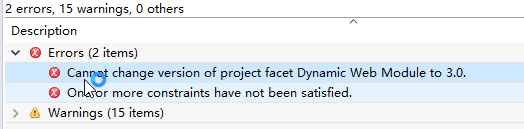


## 目录结构



# 问题

## servlet的版本和web.,xml里面所使用的配置文件的版本不一致

  
解决方法：将项目中.settingXML文件的<installed facet="jst.web" version="3.0"/>修改为与web.xml中的版本一致。

## XXXXClassNotFoundException

原因是因为jar没有完全下载

把本地仓库里面的相关jar包的文件夹删除再更新整理个项目

## 解决maven中servlet-api.jar和容器自带的servlet-api.jar的依赖冲突

web项目的serlvet api 和jsp api依赖设置为provided