# Maven概述

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的九个核心概念分别为POM、约定目录结构、坐标、仓库、依赖、生命周期、插件和目标、继承和聚合。

**Maven构建环节**

清理：删除以前的编译结果，为重新编译做好准备。

编译：将Java源程序编译为字节码文件。

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。

报告：在每一次测试后以标准的格式记录和展示测试结果。

打包：将一个包含诸多文件和目录的工程封装为一个可用于安装或部署的压缩文件。Java工程对应jar包，Web工程对应war包。

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。

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。

**Maven和ANT有什么区别?**

虽然两者都是构建工具，都用于创建 Java 应用，但是 Maven 做的事情更多，在基于“约定优于配置”的概念下，提供标准的Java 项目结构，同时能为应用自动管理依赖（应用中所依赖的JAR 文件）。

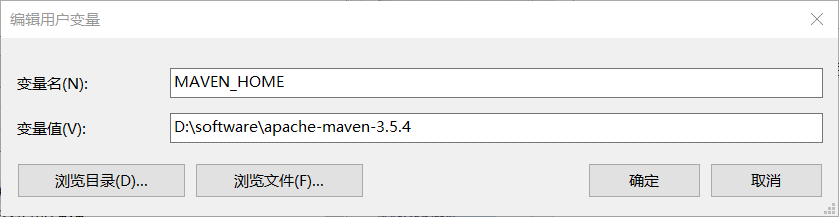
# Maven操作

## 安装并配置

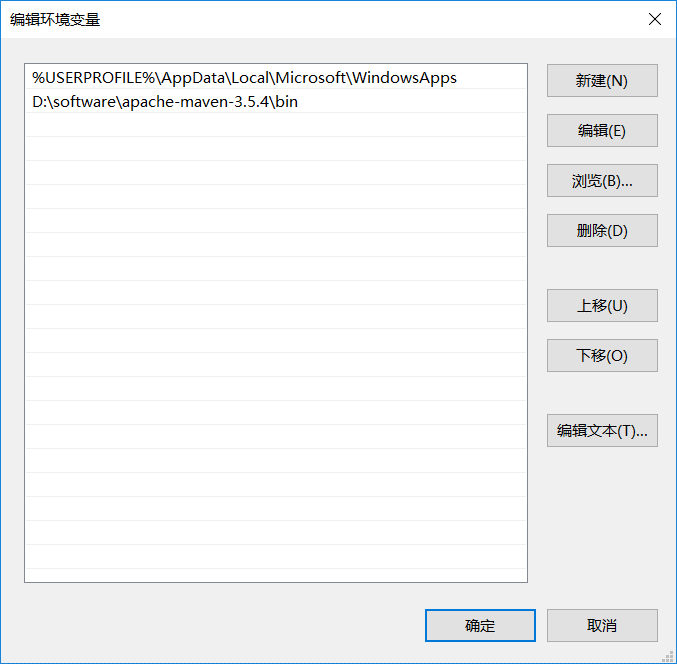
maven的安装只要解压即可，解压到安装目录后进行配置maven核心程序，要求解压到非中文且没有空格的目录。

然后配置环境变量，如下：

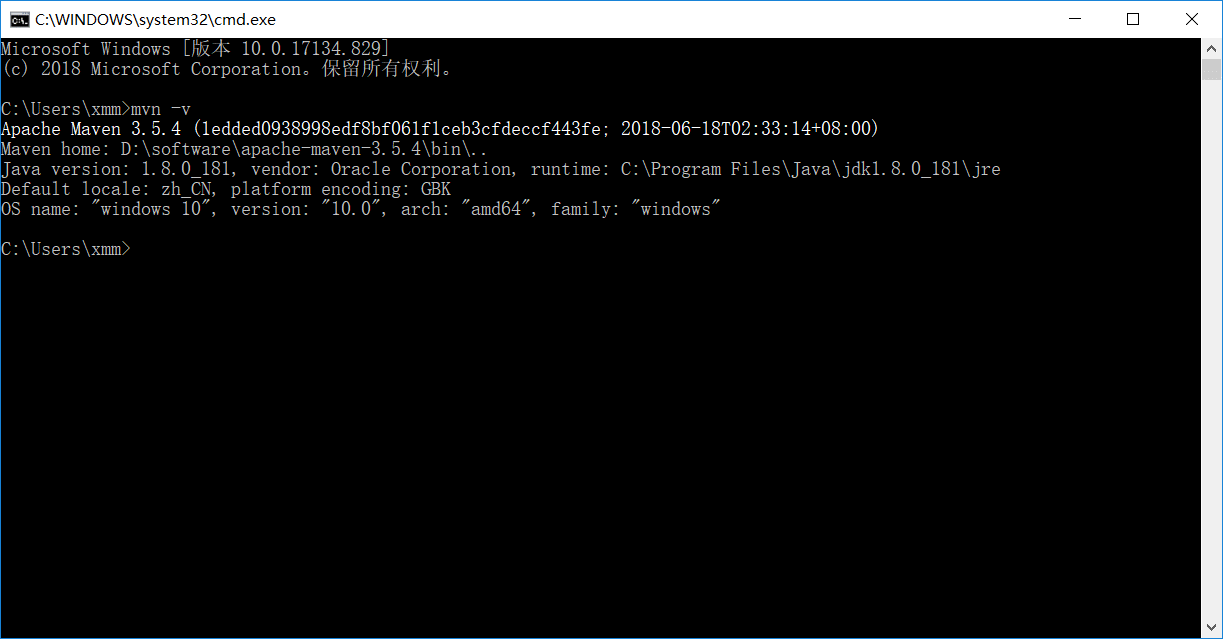
新建环境变量MAVEN\_HOME



在Path中添加maven的安装路径，截至到bin目录



在Dos操作系统中输入mvn –v查看是否配置成功

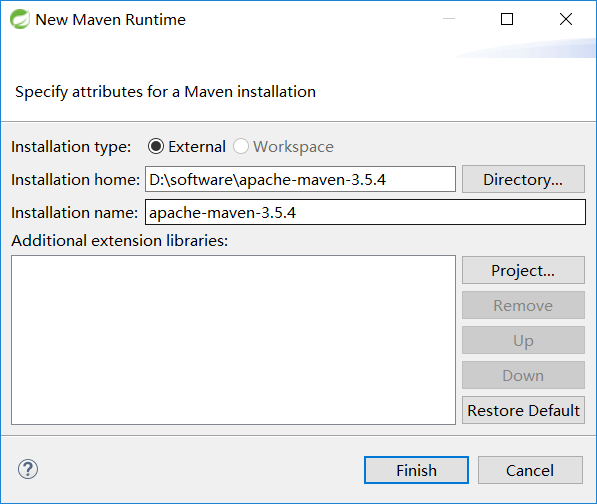


安装完成后还需要对位于安装目录下的conf目录下的settings.xml配置文件进行配置。其配置文件内容如下：

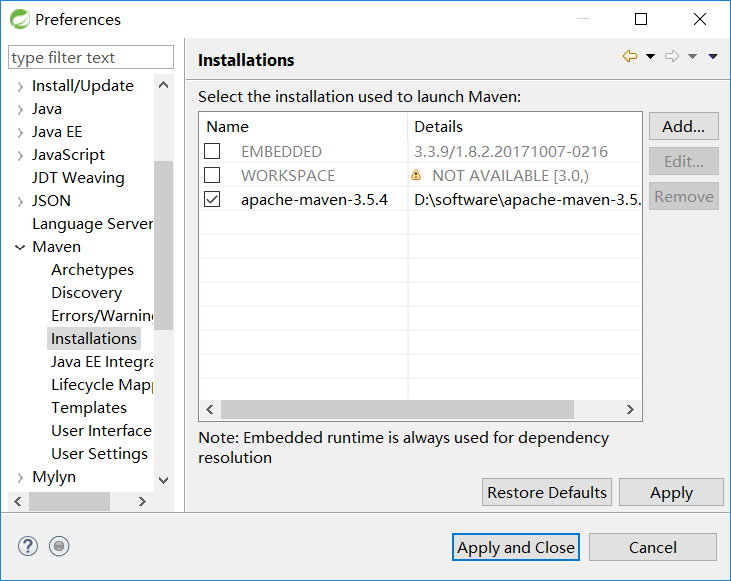
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">  <!--配置maven的本地仓库位置，即存放管理依赖jar包的地址-->  <localRepository>D:\software\apache-maven-3.5.4\mylib</localRepository>  <pluginGroups></pluginGroups>  <proxies></proxies>  <servers></servers>  <!--配置maven的远程仓库位置，这里配置的是阿里云的地址-->  <mirrors>  <mirror>  <id>nexus-aliyun</id>  <mirrorOf>central</mirrorOf>  <name>Nexus aliyun</name>  <url>http://maven.aliyun.com/nexus/content/groups/public</url>  </mirror>  </mirrors>  <!--配置创建的maven工程的jdk版本-->  <profiles>  <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>  </profiles> </settings> |

## 配置Eclipse中的maven插件

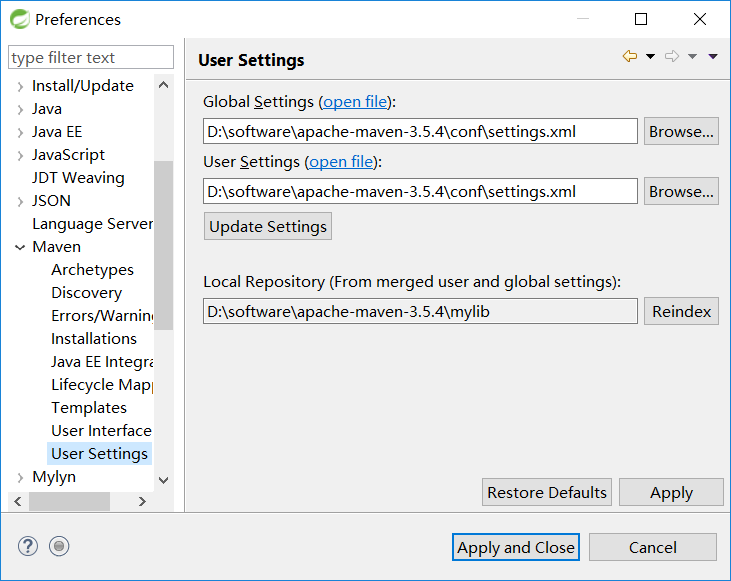
在Eclipse中的Preferences菜单🡪Maven🡪Installations🡪Add，在Installation home选择maven的安装目录。



添加完成后单击Finish进入Installations界面，勾选刚刚添加的apache-maven-3.5.4，单击Apply保存配置。



然后在Maven选项的User Settings子选项中配置修改后的settings.xml配置文件和本地仓库Local Repository。



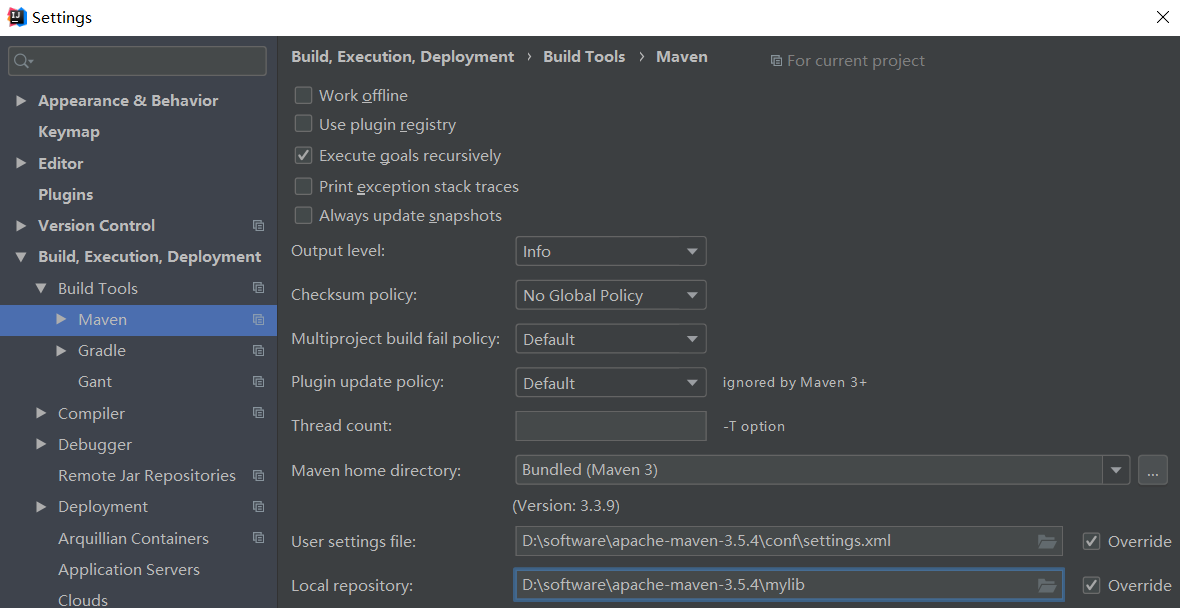
在每一次新建workspace后都需要重新进行maven的配置。

配置完成后创建maven工程测试是否配置成功，选择New选项的Maven Project自选项🡪Create a simple project🡪next🡪Group Id填写com | org.公司或组织名.总项目名；Artifact Id中填写的是创建的maven模块名；Packaging中如果创建的是Java工程则是jar，如果创建的是web工程则是war。

创建web工程时会缺少web.xml配置文件，使用eclipse中的右键菜单中Java EE Tools选项的子选项Generate Deployment Descriptor Stub自动配置web.xml配置文件。

## 配置IDEA中的maven插件

在Maven菜单中选择自定义的maven，将Maven home directory中填写为maven的安装目录，User settings file中选择自定义的settings.xml配置文件，Local repository中选择自定义的本地仓库。



## maven常见命令

执行和构建相关的命令一定要在pom.xml所在的目录执行，而像mvn –v这种命令就和构建无关，可以不再pom.xml所在的目录执行。

编译：mvn compile

清理：mvn clean

测试：mvn test

|  |
| --- |
| // 静态导入，让当前类可以直接使用Assert中的静态资源 import static org.junit.Assert.*assertEquals*; …… @Test public void testSalary() {  Employee employee = new Employee();  int expected = 12000;  int actual = employee.getEmployeeSalaryPerYear(1000);  // 断言一个值应该是多少，和实际值相比较，如果不一致会抛出异常，测试失败  assertEquals(expected, actual);  System.out.println("Maven Test Output ☆☆☆☆☆☆☆");  } |

打包：mvn package

安装：mvn install

部署：mvn deploy

在eclipse中执行maven命令可以在pom.xml配置文件上右键选择run as选项选择需要执行的maven命令。如果选项中没有目标命令则可以选择Maven build...选项并在弹出的新框体中的Goals栏填写需要执行的命令。

# Maven概念

## POM

POM(**Project Object Model**)：项目对象模型，pom体现了一种编程思想，将现实生活中的具体业务需求设计为一种模型，用程序代码中的对象来表示，这样就可以使用程序来解决现实生活中的问题。与之类时的还有文档对象模型DOM(**Document Object Model**)和浏览器对象模型BOM(**Browser Object Model**)。

Maven的pom.xml就是对maven的构建过程和依赖管理等各个方面进行统一配置的配置文件，是maven工程的核心配置文件。

## 约定目录结构

约定目录结构是为了使maven实现自动化构建时知道构建的源文件和目标文件应该存放的目录。

**目录结构说明**

src目录：总体上用于存放源码

target目录：存放构建过程得到的产品

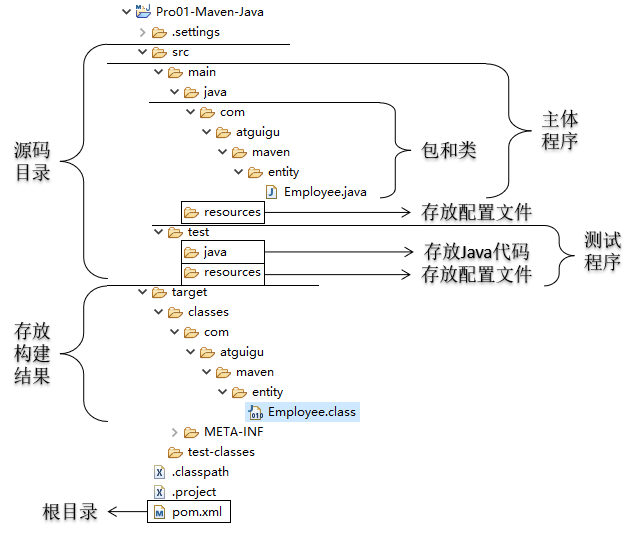
main目录：存放主程序相关

test目录：存放测试程序相关

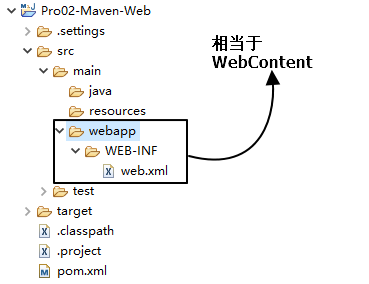
java目录：存放Java源程序

resources目录：存放配置文件或资源文件

**Java工程目录结构**



**Web工程目录结构**



## 坐标

**Maven坐标**

maven坐标用于在maven仓库中唯一定位到一个maven工程（jar包）

groupId：公司或组织域名+项目名称

artifactId：模块名称

version：版本

Maven坐标和仓库中jar包存储位置之间的对应关系

|  |
| --- |
| <groupId>com.google.code.findbugs</groupId> <artifactId>jsr305</artifactId> <version>2.0.1</version> |

Maven本地仓库与之对应的jar包根目录为：仓库根目录/com/google/code/findbugs /jsr305/2.0.1/ jsr305-2.0.1.jar

## 仓库

maven中需要配置本地仓库和远程仓库，本地仓库用于存储jar包，maven项目导入jar包时会优先在本地仓库中寻找，如果没有找到才会再到远程仓库中寻找相关jar包。

本地仓库和远程仓库都需要在settings.xml中进行配置。仓库中存放的是maven自身工作时所需要的插件、第三方框架或工具的jar包以及自己开发的maven工程。不管是什么jar包，在maven仓库中的管理方式是一样的。

## 依赖

依赖的本质就是A工程的运行需要使用B工程中的Java类、接口或配置文件等。

**传统添加jar包的缺点**

 传统方式添加jar包会造成当前工程臃肿，直接把jar包本身复制到当前工程，jar包如果很多，当前工程就会被撑的很大。

 jar包在很多不同工程中是重复的，浪费存储空间，还可能会导致版本不一致。

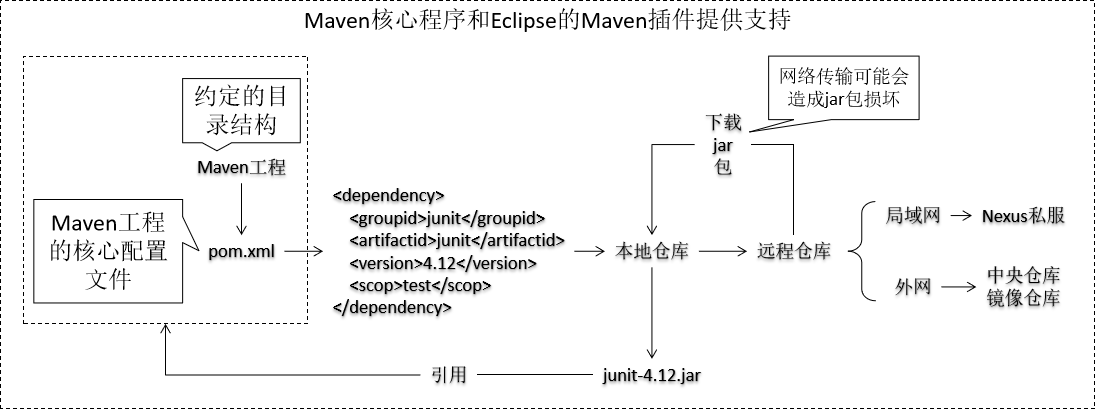
 jar包的来源不规范：

官网下载：找下载链接很麻烦

论坛/社区：版本号或许会有细微差异，名称或许有细微差异，内部或许会有缺失或篡改导致无法使用

 jar包之间存在错综复杂的依赖关系

**Maven管理jar包的工作机制**



**配置依赖关系**

在pom.xml配置文件中配置依赖：

|  |
| --- |
| <!-- dependencies中包含多个dependency --> <dependencies>  <!-- 每一个dependency对应一个具体的依赖，对应一个具体的jar包 -->  <dependency>  <!-- gav三个标签是要依赖的jar包的坐标 -->  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <!-- 依赖范围 -->  <scope>test</scope>  </dependency> </dependencies> |

在配置依赖时如果不知道具体的配置方式，可以在https://mvnrepository.com网站中查询。

**依赖的传递**

A工程→依赖→B工程→依赖C工程

C工程也会被导入到A工程中

依赖传递生效的条件：

* 依赖的范围：compile
* 没有设置optional为true

依赖无法传递的情况：

* 依赖的范围：test
* 依赖的范围：provided
* 在依赖中配置了optional：true

**依赖的范围**

maven中的依赖有三个重要的依赖范围，分别是compile、test和provided。

* compile

scope标签的默认值

如果我们需要导入Spring、MyBatis、Log4j等等这样在主体程序中正式使用的jar包，需要使用compile范围。

空间角度：对main目录、test目录下的代码都有效

时间角度：对开发阶段、在服务器运行阶段都有效

* test

用于测试的jar包，使用test范围。

空间角度：对main目录无效，对test目录有效

时间角度：对开发阶段有效，不参与服务器部署

* provided

用于在开发时使用servlet或jsp相关的API的jar包，使用provided范围。

空间角度：对main目录、test目录下的代码都有效

时间角度：对开发阶段有效，不参与服务器部署

**依赖的排除**

可以通过exclusions标签配置阻止jar包通过依赖的传递进入当前工程。有的时候如果jar包之间冲突，那么可以使用这种方式排除某一个jar包。

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>5.1.5.RELEASE</version>  <!-- 在依赖的传递中进行有针对性的排除 -->  <exclusions>  <exclusion>  <groupId>org.springframework</groupId>  <artifactId>spring-jcl</artifactId>  </exclusion>  </exclusions> </dependency> |

## 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。

三套生命周期是相互独立的，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，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。

运行某个生命周期中的任何一个阶段时，它前面的所有阶段都会被运行。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。

**Clean生命周期**

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

**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：

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。

**Site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。

## 插件和目标

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的一个目标；pre-clean是插件maven-clean-plugin的一个目标。

## 继承

依赖关系也可以继承，即A工程可以继承B工程中的依赖关系，从而实现依赖信息的统一管理，此时被继承的B工程创建时的打包方式Packaging必须设置为pom。

**子工程中配置依赖关系**

|  |
| --- |
| <!-- 当工程的坐标中相对于父工程重复的部分可以去掉 --> <!-- <groupId>com.atguigu.maven</groupId> --> <artifactId>Pro01-Maven-Java</artifactId> <!-- <version>0.0.1-SNAPSHOT</version> -->  <parent>  <!-- 父工程的坐标 -->  <groupId>com.atguigu.maven</groupId>  <artifactId>Pro03-Maven-Parent</artifactId>  <version>0.0.1-SNAPSHOT</version>  <!-- 以当前pom.xml为基准，找到父工程的pom.xml的路径 -->  <relativePath>../Pro03-Maven-Parent/pom.xml</relativePath> </parent> |

**父工程中配置依赖关系**

|  |
| --- |
| <!-- 使用DependencyManagement配置依赖的管理 --> <dependencyManagement>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>5.1.5.RELEASE</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>4.0.0</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>jsp-api</artifactId>  <version>2.1</version>  <scope>provided</scope>  </dependency>  </dependencies> </dependencyManagement> |

**在子工程中删除被管理的依赖的版本**

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

在子工程中删除依赖的版本可以达到在父工程中修改依赖的版本时，各个子工程全部被修改，实现一处修改，处处生效的效果。还可以使用properties声明属性使得该功能更强化一步：

|  |
| --- |
| <properties>  <spring.version>4.3.18.RELEASE</spring.version> </properties> <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>  </dependencies> |

## 聚合

一个项目中的多个工程想要安装到仓库中时，只能一个一个手动安装，比较繁琐，配置聚合之后就能够实现一键安装。

做法：创建一个打包方式为pom的工程作为聚合工程，和Parent可以是同一个工程，也可以不是。然后在聚合工程中配置各个模块，在聚合工程上执行install命令即可。

聚合的目的是为了呈现各个模块清单，从而实现一键安装，其实现方式是在父工程中配置modules标签。

<!-- 配置聚合 -->

|  |
| --- |
| <modules>  <module>../Pro01-Maven-Java</module>  <module>../Pro02-Maven-Web</module> </modules> |

# Maven的jar包下载失败

## 表现形式一

**表现**

jar或pom文件以lastUpdated结尾。表示文件没有下载完成。

原因

Maven正常下载文件过程中，文件会以lastUpdated结尾，成功下载完成，Maven会自动将末尾的lastUpdated扩展名删除。

但如果没有下载成功，Maven不会自动删除末尾的lastUpdated扩展名。

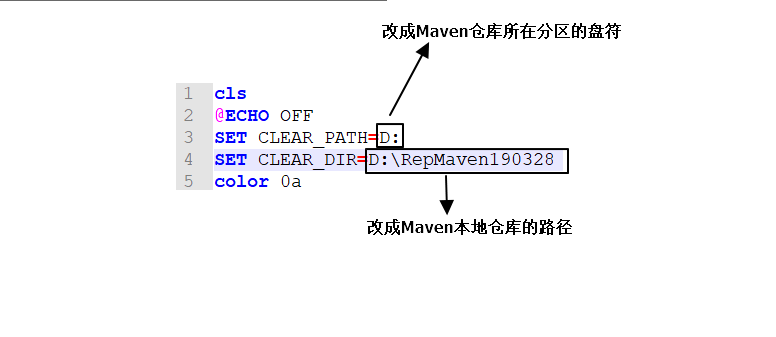
下一次重新下载时，Maven会忽略带有lastUpdated扩展名的文件。

**解决**

删除lastUpdated扩展名文件，在Eclipse中对Maven执行“Update Project”操作（快捷键是：alt+F5弹出界面然后点OK）。

但是Maven仓库使用一段时间后往往会有很多lastUpdated文件，手动删除太麻烦，所以借助clearLastUpdated.bat批处理脚本。

clearLastUpdated.bat文件使用：放在Maven本地仓库的根目录下



## 表现形式二

表面上jar包已经下载完成，但是内部损坏。

**具体表现形式**

包括但不限于以下几种：

ClassNotFoundException：对应的找不到的类是jar包中的类

NoSuchMethodException：对应的找不到的方法所在的类是jar包里面的

莫名其妙的异常，从表面看来和我们编写的代码以及配置文件没有直接关系

**解决办法**

根据抛异常的具体信息确定出问题的jar包，在Maven的本地仓库找到这个jar包，确定是不是这个jar包的问题，借助文件校验工具。

**文件校验工具原理**

本质上是一个HASH加密工具。

HASH算法有如下特点：输入数据不变，输出数据也不变；输入数据发生一丁点儿变化，输出数据就会发生天翻地覆的变化；不管输入数据体积多大，只要使用同一个算法，输出的数据长度固定；不可逆，不能根据密文反推明文。

