Maven 学习笔记

概述

Apache Maven 是一个软件项目管理和综合工具，基于项目对象模型（POM）的概念，Maven 可以从一个中心资料片管理项目构建，报告和文件。

Maven 主要目标是提供给开发人员：

- 项目是可重复使用，易维护，更容易理解的一个综合模型。
- 插件或交互的工具，这种声明性的模式

Maven 项目的结构和内容在一个 XML 文件中声明，pom.xml 项目对象模型（POM），这是整个 Maven 系统的基本单元

配置环境变量

添加 M2_HOME、MAVEN_HOME
更新 path：添加%M2_HOME%\bin\

```
mvn help:system
```

该命令会打印出所有的Java系统属性和环境变量，这些信息对我们日常的编程工作很有帮助。运行这条命令的目的是为了让Maven执行一个真正的任务。我们可以从命令行输出看到Maven会下载maven-help-plugin，包括POM文件和JAR文件。这些文件都被下载到了Maven本地仓库中。

也就是说这个命令不会整个仓库拖下来，而是把这个命令所需的JAR包下回来本地而已。

GroupID & artfactID

groupId
定义了项目属于哪个组，举个例子，如果你的公司是 mycom，有一个项目为 myapp，那么 groupId 就应该是 com.mycom.myapp.
groupId 一般分为多个段，这里我只说两段，第一段为域，第二段为公司名称。域又分为 org、com、cn 等等许多，其中 org 为非营利组织，com 为商业组织。举个 apache 公司的 tomcat 项目例子：这个项目的 groupId 是 org.apache，它的域是 org（因为 tomcat 是非营利项目），公司名称是 apache，artigactId 是 tomcat。
artifacted
定义了当前 maven 项目在组中唯一的 ID，比如，myapp-util，myapp-domain，myapp-web 等。

常见命令

- **validate**: 验证项目是否正确
- **compile**: 编译项目的源代码
- **test**: 使用合适的单元测试框架测试编译的源代码。这些测试不应该要求代码被打包或部署
- **package**: 打包项目，生成 jar 包 或 war 包
- **verify**: 运行所有检查，检测 package 是否有效
- **install**: 将项目打成 maven 放入本地 maven 仓库
- **deploy**: 将项目发布到远程 maven 仓库
- **clean**: 清空 target 目录
- **site**: 生成项目文档，运行 mvn site 命令会在 target/site/目录下生成文件
- **注**: maven 命令可顺序执行，如下

官方文档学习

Site: https://maven.apache.org/guides

安装

- 安装
- 配置环境变量
- 安装校验: mvn -version
- Create a project

The pom:

Maven 核心配置文件，pom 和复杂，但是一般只需要知道部分配置。

```
<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.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

生命周期概述

- validate
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile

默认的完整声明周期:

- validate 校验: validate the project is correct and all necessary information is available
- compile 编译: compile the source code of the project
- test 测试: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- package 打包: take the compiled code and package it in its distributable format, such as a JAR.
- integration-test 集成测试: process and deploy the package if necessary into an environment where integration tests can be run
- verify 验证: run any checks to verify the package is valid and meets quality criteria
- install 安装: install the package into the local repository, for use as a dependency in other projects locally
- deploy 部署: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

还有两个不在上述默认声明周期中:

- clean: cleans up artifacts created by prior builds
  mvn clean dependency:copy-dependencies package
- site: generates site documentation for this project

Maven 概述

Maven is essentially a project management 项目管理 and comprehension tool and as such provides a way to help with managing:

- Builds
- Documentation
- Reporting
- Dependencies

- SCMs
- Releases
- Distribution

Pom 标签

1. Project: This is the top-level element in all Maven pom.xml files.
2. modelVersion This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
3. groupId : This element indicates the unique identifier of the organization or group that created the project. The groupId is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example org.apache.maven.plugins is the designated groupId for all Maven plugins.
4. artifactId : This element indicates the unique base name of the primary artifact being generated by this project. The primary artifact for a project is typically a JAR file. Secondary artifacts like source bundles also use the artifactId as part of their final name. A typical artifact produced by Maven would have the form <artifactId>-<version>.<extension> (for example, myapp-1.0.jar).
5. version : This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development. We will discuss the use of snapshots and how they work further on in this guide.
6. Name: This element indicates the display name used for the project. This is often used in Maven's generated documentation.
7. url: This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
8. properties This element contains value placeholders accessible anywhere within a POM.
9. dependencies This element's children list dependencies. The cornerstone of the POM.
10. build This element handles things like declaring your project's directory structure and managing plugins.

常用命令

- mvn compile: 编译源代码，生成的类文件地址为${basedir}/target/classes
- mvn test: 编译并执行测试
- mvn test-compile: 只是编译测试代码

- mvn package: 打包，生成 jar(${basedir}/target)，并安装。 安装是把 jar 安装到本地仓库(${user.home}/.m2/repository).
- Mvn install: setting up, building, testing, packaging, and install,走遍流程并安装。
- Mvn site: setting up, building, testing, packaging, and install
- Mvn clean: **remove the target directory** with all the build data before starting so that it is fresh.
-

SNAPSHOT version

```
<version>1.0-SNAPSHOT</version>
```

The SNAPSHOT value refers to the 'latest' code along a development branch, and provides **no guarantee the code is stable or unchanging**. Conversely, the code in a 'release' version (any version value without the suffix SNAPSHOT) is unchanging.

a SNAPSHOT version is the '**development' version** before the final 'release' version.

Plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Mvn 工程结构

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           `-- application.properties
    `-- test
```

```
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

any directories or files placed within the **${basedir}/src/main/resources** directory are packaged in your JAR with the exact same structure starting at the base of the JAR.
Resource 资源文件会被直接复制进 jar

Jar 的结构:

```
|-- META-INF
|   |-- MANIFEST.MF
|   |-- application.properties
|   `-- maven
|       `-- com.mycompany.app
|           `-- my-app
|               |-- pom.properties
|               `-- pom.xml
`-- com
    `-- mycompany
        `-- app
            `-- App.class
```

有官方文档，还需要其他什么乱七八糟的东西？
明天看

知乎文章

Why maven 构建工具

◆ 一个项目就是一个工程？
如果项目非常**庞大**，就**不适合使用 package 来划分模块**，最好是**每一个模块对应一个工程**，利于分工协作。**借助于 maven 就可以将一个项目拆分成多个工程。**

◆ 项目中使用 jar 包，需要“复制”、“粘贴”项目的 lib 中
同样的 jar 包重复的出现在不同的项目工程中，你需要做不停的复制粘贴的重复工作。借助于 maven，可以将 jar 包保存在“仓库”中，不管在哪个项目只要使用引用即可就行。

◆ jar 包需要的时候每次都要自己准备好或到官网下载
借助于 maven 我们可以使用统一的规范方式下载 jar 包，规范

◆ jar 包版本不一致的风险
不同的项目在使用 jar 包的时候，有可能会导致各个项目的 jar 包版本不一致，导致未执行错

误。借助于 maven，所有的 jar 包都放在“仓库”中，所有的项目都使用仓库的一份 jar 包。

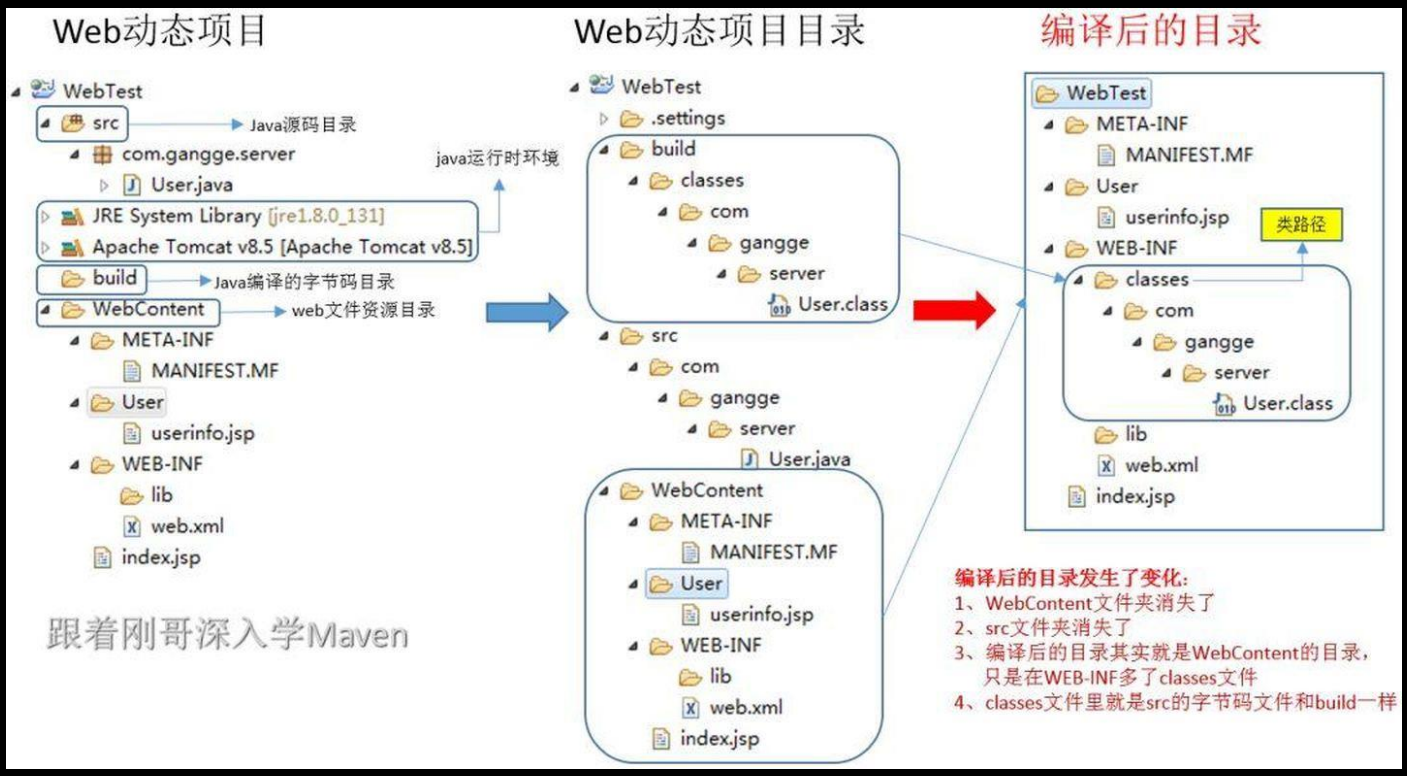
- 一个 jar 包依赖其他的 jar 包需要自己手动的加入到项目中
  借助于 maven，它会自动的将依赖的 jar 包导入进来。

maven 是什么

maven 是一款服务于 java 平台的**自动化构建工具**

make->Ant->Maven->Gradle

1. **构义**：把动态的 Web 工程经过编译得到的**编译结果部署到服务器上的整个过程。**
2. 编译：java 源文件[.java]->编译->Classz 字节码文件[.class]
3. 部署：最终在 sevlet 容器中部署的不是动态 web 工程，而是编译后的文件



构建的各个环节：

1. **清理 clean**：将**以前编译得到的旧文件 class 字节码文件删除**
2. **编译 compile**：将 java 源程序编译成 class 字节码文件
3. 测试 test：自动测试，自动调用 junit 程序
4. 报告 report：测试程序执行的结果
5. 打包 package：**动态 Web 工程打 War 包，java 工程打 jar 包**
6. 安装 install：**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**
7. 部署 deploy：将动态 Web 工程生成的 war 包复制到 Servlet 容器下，使其可以运行

第一个 maven 项目

创建约定的目录结构（maven 工程必须按照约定的目录结构创建）

根目录：工程名
|---src：源码
|---|---main:存放主程序
|---|---|---java：java 源码文件
|---|---|---resource：存放框架的配置文件
|---|---test：存放测试程序
|---pop.xml：maven 的核心配置文件

常用 maven 命令：

1. **mvn clean：清理**
2. **mvn compile：编译主程序**
3. **mvn test-compile：编译测试程序**
4. **mvn test：执行测试**
5. **mvn package：打包**
6. **mvn install：安装**

执行 maven 命令必须进入到 pom.xml 的目录中进行执行

仓库的默认位置：**c:Usrs[登录当前系统的用户名].m2repository**

仓库和坐标

pom.xml：**Project Object Model** 项目对象模型。它是 maven 的核心配置文件，所有的构建的配置都在这里设置。

坐标：使用下面的三个向量在仓库中唯一的定位一个 maven 工程

```xml
<dependencies>
    <dependency>
        <!-- 1、公司或组织域名倒序+项目名 -->
        <groupId>org.springframework</groupId>
        <!-- 2、模块名 -->
        <artifactId>spring-core</artifactId>
        <!-- 3、版本 -->
        <version>4.3.4.RELEASE</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Maven的坐标

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.hzg.maven</groupId>
    <artifactId>Hello</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>Hello</name>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.3.4.RELEASE</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</project>
```

本项目的坐标

依赖的项目坐标

maven 坐标和仓库对应的映射关系：

[groupId][artifactId][version][artifactId]-[version].jar

仓库的分类：

1、本地仓库：当前电脑上的仓库，路径上已经说过了哦

2、远程仓库：

- **私服**：搭建在局域网中，一般公司都会有私服，私服一般使用 **nexus** 来搭建。具体搭建过程可以查询其他资料
- **中央仓库**：架设在 Internet 上，像刚才的 springframework 就是在中央仓库上

依赖

maven 解析依赖信息时会到本地仓库中取查找被依赖的 jar 包

1. 对于本地仓库中没有的会去中央仓库去查找 maven 坐标来获取 jar 包，获取到 jar 之后会下载到本地仓库
2. 对于中央仓库也找不到依赖的 jar 包的时候，就会编译失败了

如果依赖的是自己或者团队开发的 maven 工程，需要先使用 install 命令把被依赖的 maven 工程的 jar 包导入到本地仓库中

依赖范围：

```
<groupId>org.springframework</groupId>
<artifactId>spring-core</artifactId>
<version>4.3.4.RELEASE</version>
<scope>compile</scope>
```

scope 就是依赖的范围

- **compile**，默认值，适用于所有阶段（开发、测试、部署、运行），本 jar 会一直存在所有阶段。
- **provided**，只在**开发、测试阶段**使用，目的是不让 Servlet 容器和你本地仓库的 jar 包冲突 。如 servlet.jar。
- **runtime**，**只在运行时使用，如 JDBC 驱动**，适用运行和测试阶段。
- **test**，**只在测试时使用**，用于编译和运行测试代码。不会随项目发布。
- **system**，类似 provided，**需要显式提供包含依赖的 jar，Maven 不会在 Repository 中查找它。**

生命周期

Maven 有**三套相互独立的生命周期**，请注意这里说的是“三套”，而且“**相互独立**”，初学者容易将 Maven 的生命周期看成一个整体，其实不然。这三套生命周期分别是：

1. Clean Lifecycle
   在进行真正的**构建之前进行一些清理工作**。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
2. Default Lifecycle
   构建的核心部分，编译，测试，打包，部署等等。
   - 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

执行 mvn install 命令，通过日志看看中间经历了什么：

```
D:\WorkSpace\Hello>mvn install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Hello 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ Hello ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.
. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ Hello ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ H
llo ---
[WARNING] Using platform encoding (GBK actually) to copy filtered resources, i.
. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ Hello
--
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ Hello ---
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ Hello ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ Hello ---
[INFO] Installing D:\WorkSpace\Hello\target\Hello-0.0.1-SNAPSHOT.jar to C:\Prog
am Files\Java\repository\com\hzg\maven\Hello\0.0.1-SNAPSHOT\Hello-0.0.1-SNAPSHO
.jar
[INFO] Installing D:\WorkSpace\Hello\pom.xml to C:\Program Files\Java\repositor
\com\hzg\maven\Hello\0.0.1-SNAPSHOT\Hello-0.0.1-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.318 s
[INFO] Finished at: 2017-06-04T11:12:01+08:00
[INFO] Final Memory: 11M/155M
```

执行 mvn install，其中已经执行了 compile 和 test

总结：不论你要执行生命周期的哪一个阶段，maven 都是从这个生命周期的开始执行

插件：每个阶段都有插件（plugin），看上面标红的。插件的职责就是执行它对应的命令。

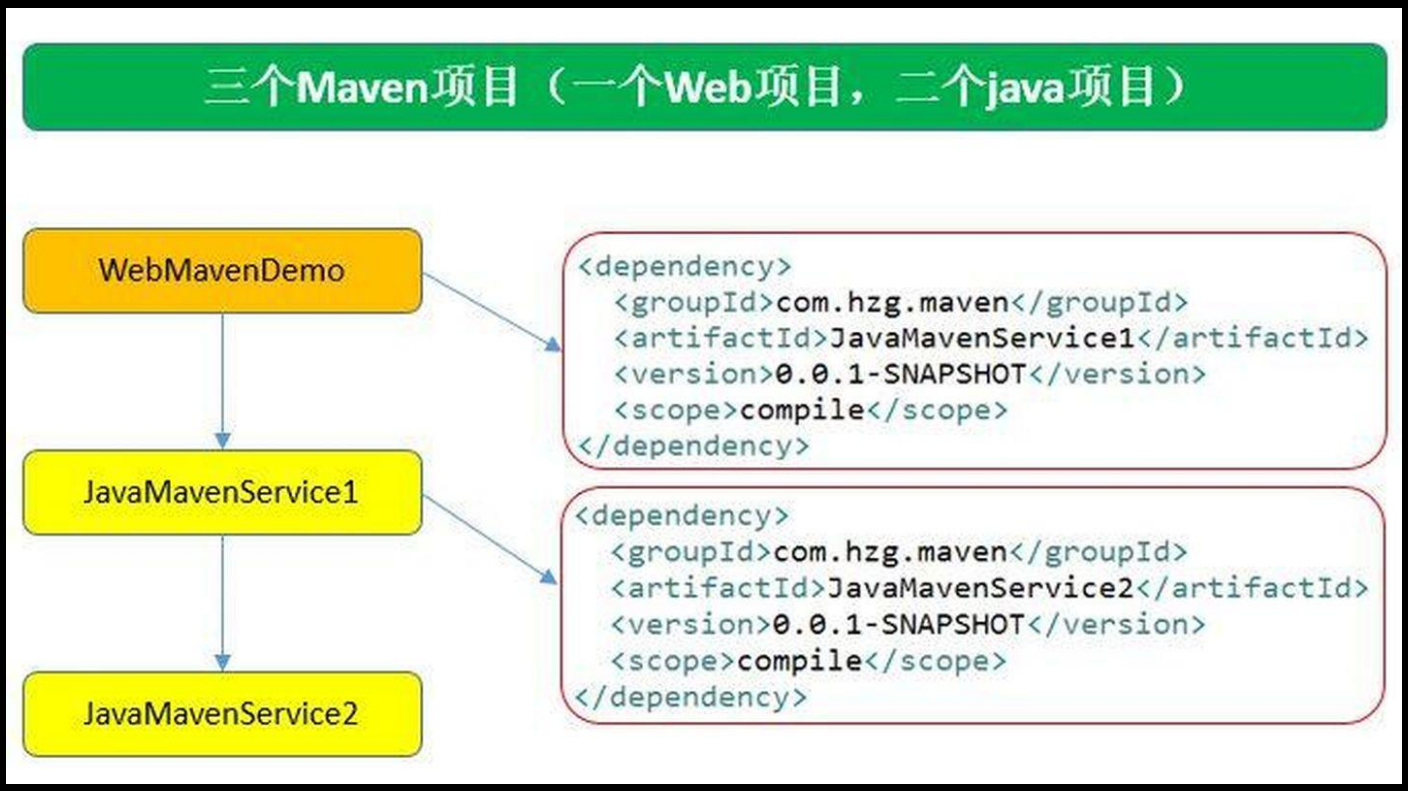
3. Site Lifecycle

生成项目报告，站点，发布站点。

- 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

maven 工程的依赖高级特性

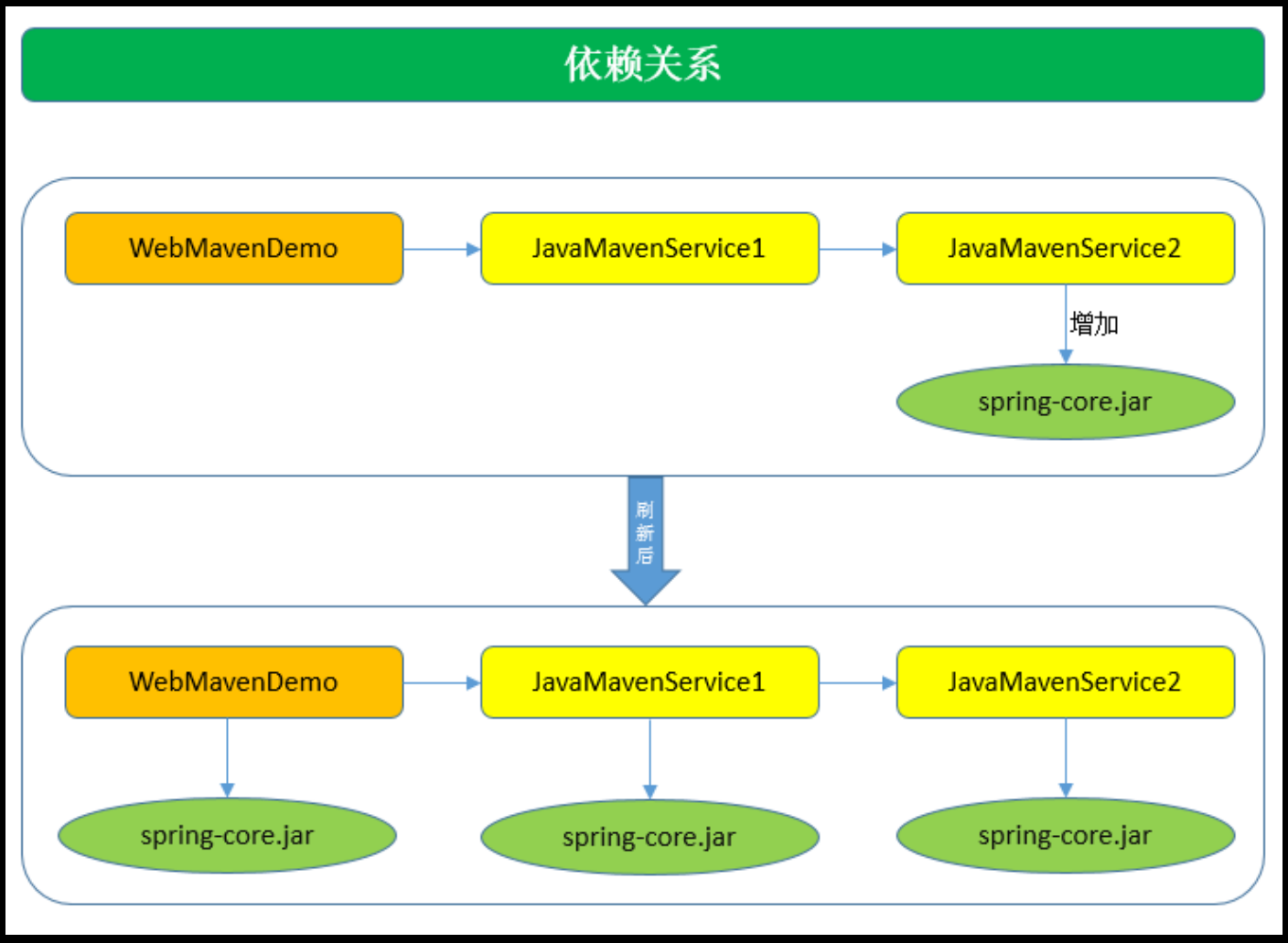
- 依赖的传递性



pom.xml 文件配置好依赖关系后，必须首先 mvn install 后，依赖的 jar 包才能使用。

- WebMavenDemo 的 pom.xml 文件想能编译通过，JavaMavenService1 必须 mvn install
- JavaMavenService 的 pom.xml 文件想能编译通过，JavaMavenService2 必须 mvn install

- 传递性：

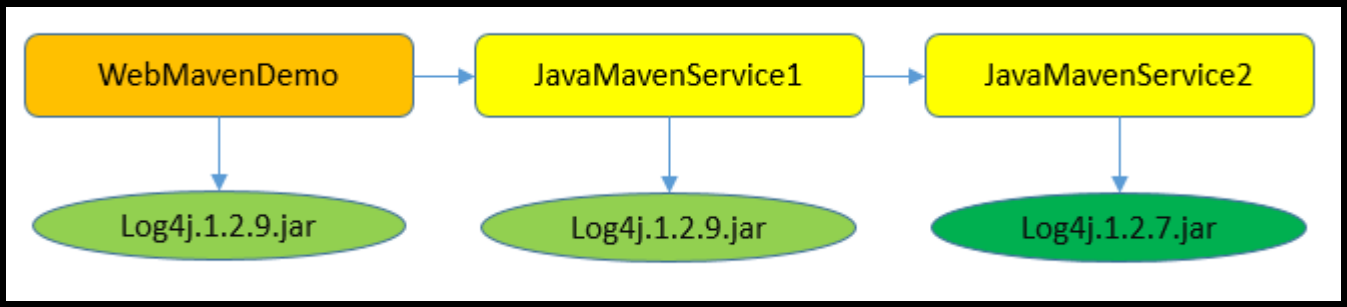


为 JavaMavenService2 中增加了一个 spring-core.jar 包后，会惊喜的发现依赖的两个项目都自动的增加了这个 jar 包，这就是依赖的传递性。

注意：**非 compile 范围的依赖是不能传递的。**

- 依赖版本的原则

1. 路径最短者优先原则



Service2 的 log4j 的版本是 1.2.7 版本，Service1 排除了此包的依赖，自己加了一个 Log4j 的 1.2.9 的版本，那么 WebMavenDemo 项目遵守路径最短优先原则，Log4j 的版本和 Sercive1 的版本一致。

- 路径相同先声明优先原则

统一管理依赖的版本：

```
<properties>
    <hzg.junit.version>3.8.1</hzg.junit.version>
    <hzg.spring.version>4.3.4.RELEASE</hzg.spring.version>
</properties>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${hzg.junit.version}</version>
    <scope>test</scope>
```

build 配置

```
<build>
    <!-- 项目的名字 -->
    <finalName>WebMavenDemo</finalName>
    <!-- 描述项目中资源的位置 -->
    <resources>
        <!-- 自定义资源 1 -->
        <resource>
            <!-- 资源目录 -->
            <directory>src/main/java</directory>
            <!-- 包括哪些文件参与打包 -->
            <includes>
                <include>**/*.xml</include>
            </includes>
            <!-- 排除哪些文件不参与打包 -->
            <excludes>
                <exclude>**/*.txt</exclude>
                    <exclude>**/*.doc</exclude>
            </excludes>
        </resource>
    </resources>
    <!-- 设置构建时候的插件 -->
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <!-- 源代码编译版本 -->
                <source>1.8</source>
                <!-- 目标平台编译版本 -->
                <target>1.8</target>
            </configuration>
        </plugin>
        <!-- 资源插件（资源的插件） -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-resources-plugin</artifactId>
            <version>2.1</version>
            <executions>
                <execution>
                    <phase>compile</phase>
                </execution>
```

```
            </executions>
            <configuration>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
        <!-- war 插件(将项目打成 war 包) -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <!-- war 包名字 -->
                <warName>WebMavenDemo1</warName>
            </configuration>
        </plugin>
    </plugins>
</build>
```

如何将 maven 项目划分为多个模块

多模块拆分的必要性

使用 Java 技术开发的工程项目，无论是数据处理系统还是 Web 网站，随着项目的不断发展，需求的不断细化与添加，工程项目中的代码越来越多，包结构也越来越复杂这时候工程的进展就会遇到各种问题：

（1）不同方面的代码之间相互耦合，这时候一系统出现问题很难定位到问题的出现原因，即使定位到问题也很难修正问题，可能在修正问题的时候引入更多的问题。

（2）多方面的代码集中在一个整体结构中，新入的开发者很难对整体项目有直观的感受，增加了新手介入开发的成本，需要有一个熟悉整个项目的开发者维护整个项目的结构（通常在项目较大且开发时间较长时这是很难做到的）。

（3）开发者对自己或者他人负责的代码边界很模糊，这是复杂项目中最容易遇到的，导致的结果就是开发者很容易修改了他人负责的代码且代码负责人还不知道，责任追踪很麻烦。

将一个复杂项目拆分成多个模块是解决上述问题的一个重要方法，多模块的划分可以降低代码之间的耦合性（从类级别的耦合提升到 jar 包级别的耦合），每个模块都可以是自解释的（通过模块名或者模块文档），模块还规范了代码边界的划分，开发者很容易通过模块确定自己所负责的内容。

EG：将原来 product 项目拆分为如下 3 个模块

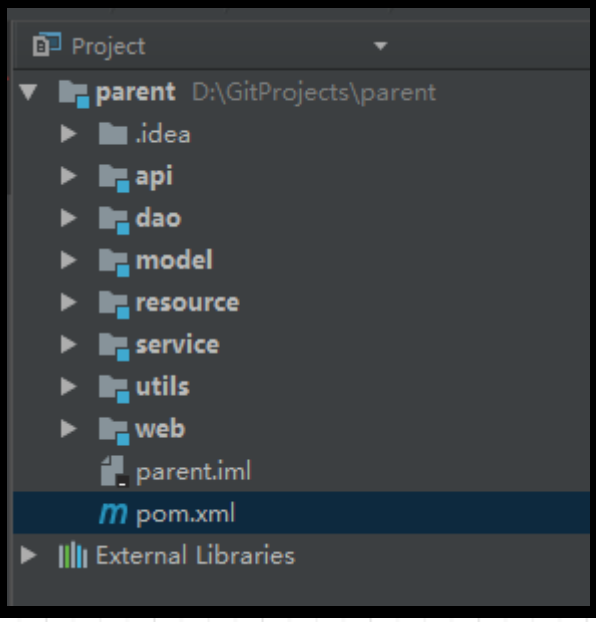
1. **product-server //所有的业务逻辑**
2. **product-client //对外暴露的接口**
3. **product-common //公用的对象**

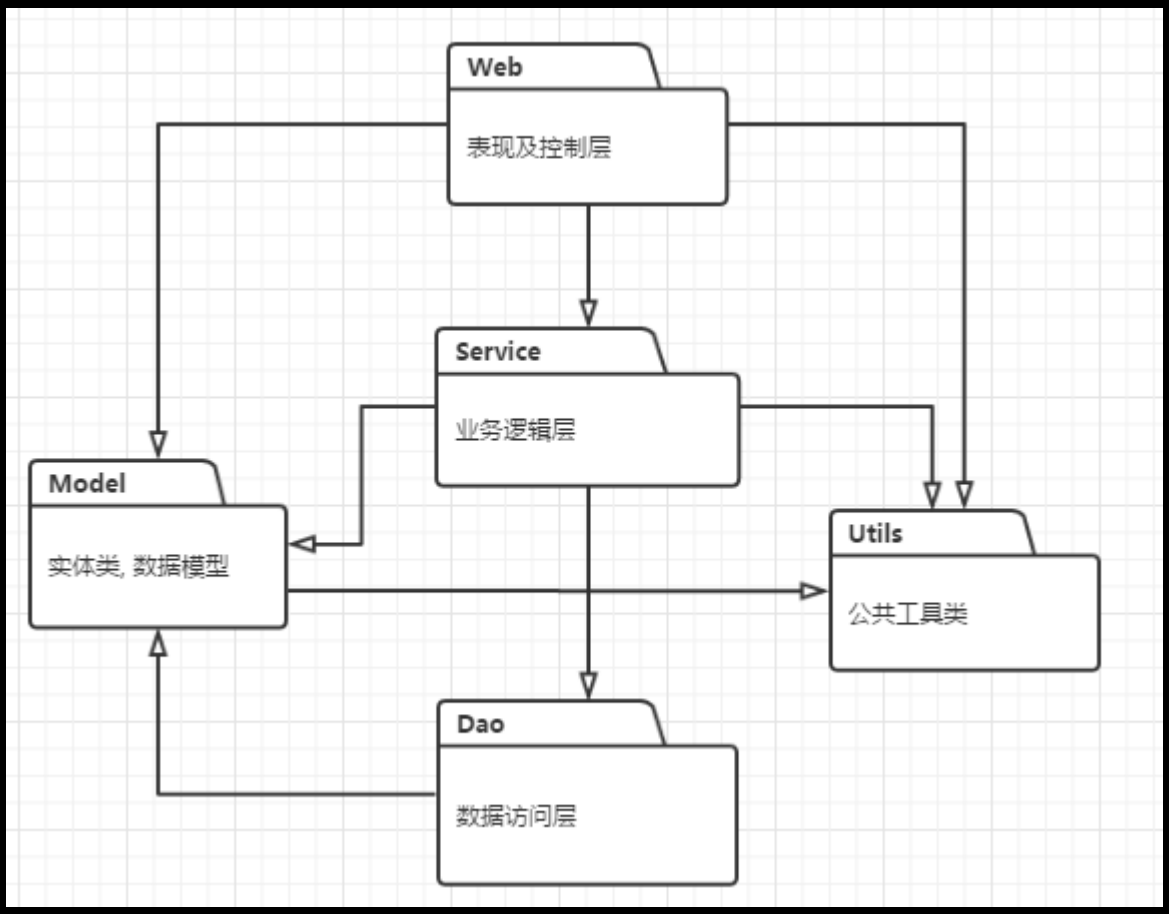
操作：

1. 在原有的项目上右击，New->Module

2. 选择 Maven 选项，archetype 可以不选择，直接 next
3. 填写对应的 GroupId 和 ArtifactId 即可，一直 next，一直到 finish
如果发现创建好的 maven 项目不能新增 class 文件，可以在对应的文件夹上右键，将其添加为 Sources Root 即可；

多模块项目结构例子：

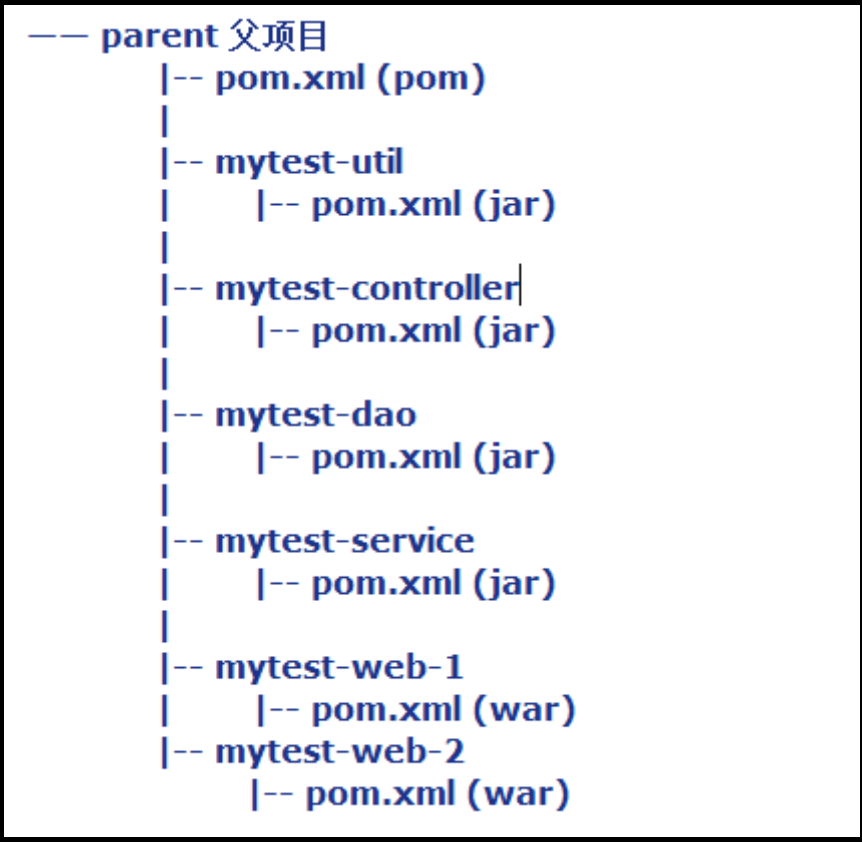




dao 层依赖 model
service 层依赖 dao、model、utils
web 层依赖 service、model、utils

Edwin Xu

例子 2:
一个简单的 Java Web 项目，Maven 模块结构是这样的



上述示意图中，有一个父项目(parent)聚合很多子项目（mytest-controller，mytest-util，mytest-dao，mytest-service，mytest-web)。每个项目，不管是父子，都含有一个 pom.xml 文件。而且要注意的是，小括号中标出了每个项目的打包类型。父项目是 pom,也只能是 pom。子项目有 jar，或者 war。根据它包含的内容具体考虑。

父项目声明打包类型等:

```
<groupId>my.test</groupId>
<artifactId>mytest-parent</artifactId>
<version>1.0</version>
<packaging>pom</packaging>
```

声明各个子模块:

```
<modules>
        <module>mytest-controller</module>
        <module>mytest-service</module>
        <module>mytest-util</module>
        <module>mytest-dao</module>
        <module>mytest-web-1</module>
        <module>mytest-web-2</module>
</modules>
```

一般来说，项目中需要的外部依赖等都在父项目中引入，这样在子项目中省去了不必要的配置。

另外，各个子项目间的依赖在单独的 pom.xml 中配置，
比如 mytest-web 项目依赖控制层的 mytest-controller，那么就在依赖中单独配置:

```
<dependency>
          <groupId>my.test<</groupId>
```

```
            <artifactId>mytest-controller</artifactId>
            <version>1.0</version>
</dependency>
```

这就需要在项目拆分和架构之前需要理清各个模块间的依赖关系。
如果是单个 War 项目，使用普通的构建方式即可，需要注意的是如果项目中包含多个 war 的子模块，需要使用 maven 的 maven-war-plugin 插件的 overlays 属性来处理

案例地址：
https://blog.csdn.net/icecoola /article/details/77717467

dependencies 与 dependencyManagement 搭配使用

1. dependencies 即使在子项目中不写该依赖项，那么子项目仍然会从父项目中继承该依赖项（全部继承）
2. **dependencyManagement 里只是声明依赖，并不实现引入**，因此子项目需要显示的声明需要用的依赖。如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，**并且没有指定具体版本**，才会从父项目中继承该项，并且 version 和 scope 都读取自父 pom；另外如果子项目中指定了版本号，那么会使用子项目中指定的 jar 版本。

Maven 错误日志

Pom.xml 报错

需要点击项目-Maven-update（最上面两个必选）
如果这也错误，报错：Could not get the value for parameter encoding for plugin execution default
解决：

01. 先关闭 eclipse
02. 找到 maven 的本地仓库路径、例如（C:\Users\YourUserName.m2）
03. 删除文件夹：repository
04. 重新打开 eclipse
05. clean 有问题的 maven 项目、点击 Project → clean...

6. 右键有问题的 maven 项目、依次点击 Maven → Update Project...
7. 关闭 eclipse
8. 打开 eclipse
9. 在 Project Explorer 中选中有问题的 project
10. 按 F5 键刷新一会
11. 大功告成