一、引言

1.1 现有问题?

1、开发到底使用eclipse还是idea工具进行开发?

2、项目组加入了新的人员，我要给他说明编译环境如何设置，但是让我挠头的是，有些细节我也记不清楚了。

3、一个项目需要好几百个jar包，每次都拷入都太费力了!

4、项目想临时打包一个测试版本，还需要使用大量的大包命令!

二、Maven简介

2.1 概念

https://maven.apache.org/what-is-maven.html

Maven是基于项目对象模型(POM)，可以通过一小段描述信息来管理项目的构建，报告和文档的软件项目管理工具。

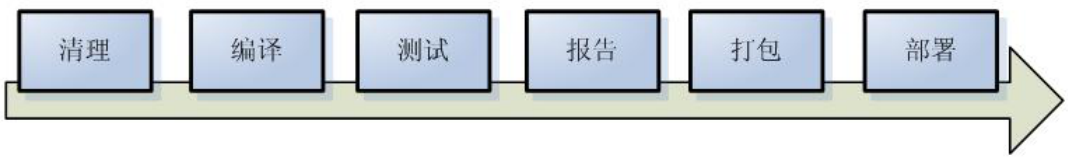
Maven主要服务于基于Java平台的**项目构建，依赖管理（jar）**和**项目信息管理。**

Maven主要有两个功能:

1、项目构建(项目基本结构创建、编译、打包)

2、依赖管理(做好配置，自定下载依赖)

项目构建过程



常见项目构建工具

eclipse：eclipse可以说是项目编码工具，也可以说他是一个项目构建工具，他可以创建自己格式的项目结构！并且会编译项目代码！但是它的构建过程是不完整，他无法直接进行打包部署等流程！还需要额外配置！

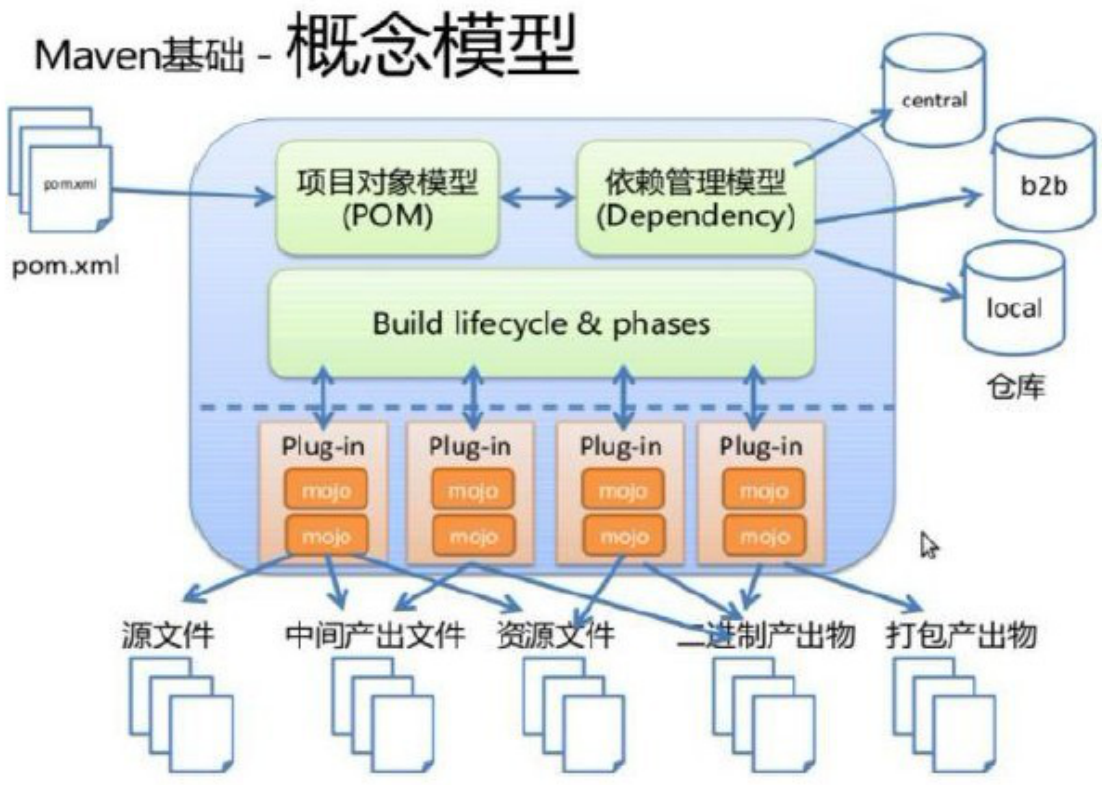
Idea:也是一款非常优秀的项目编码工具，同时也是构建工具，但是我们要知道，这个工具构建的项目格式是不同于eclipse的，但是他的自动化编译过程更加优秀，但是它也需要外部配置服务器信息，进行项目打包，并部署！

对比：从编码工具角度上来说，idea更加的优秀，但是从项目构建的角度来说，二者皆有问题！因为他们每个创建的项目的格式都不同，如果你使用eclipse开发的项目。就很难倒入到idea,反之依然！

maven:并不是编码工具，而是一款编译工具，他能创建出maven特有的项目结构，并能支持全套的项目构建过程，而且是生命周期性质的，使用maven编译的项目，我们可以在任何的开发工具打开！

但是我们还需要明确的是，maven是项目构建工具，以及项目依赖管理工具，所以他不能单独使用！我们正确的使用路线应该是eclipse+maven或者idea+maven,eclipse和idea负责编码提示，maven负责项目的构建和依赖管理！

2.2 maven项目模型图



2.3 访问和下载

> 官方网站：http://maven.apache.org
>
> 下载地址：http://maven.apache.org/download.cgi

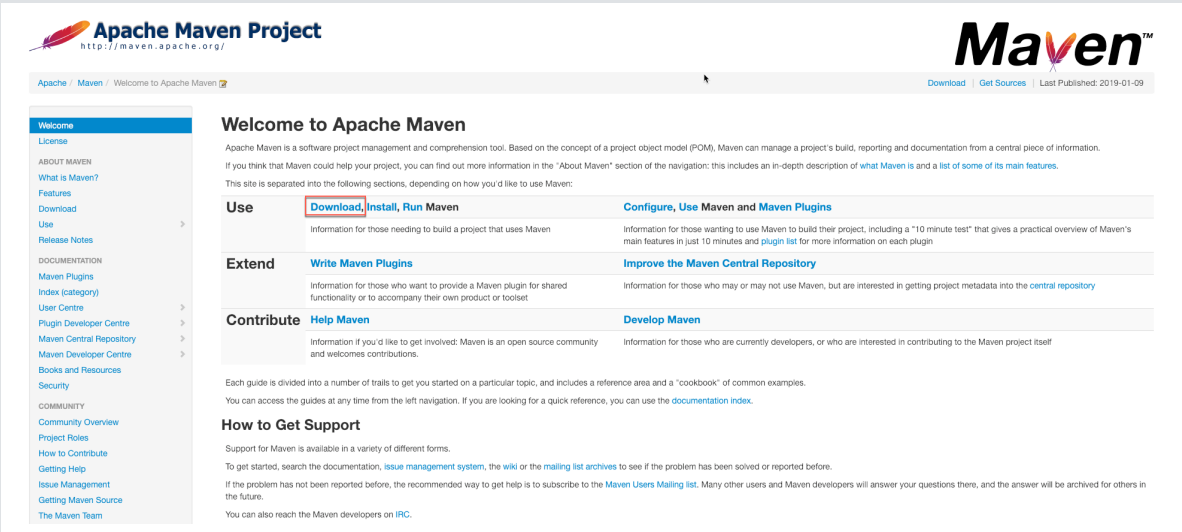
- 注意：Maven 3.3+ 需要执行JDK1.7或更高版本

三、Maven的安装

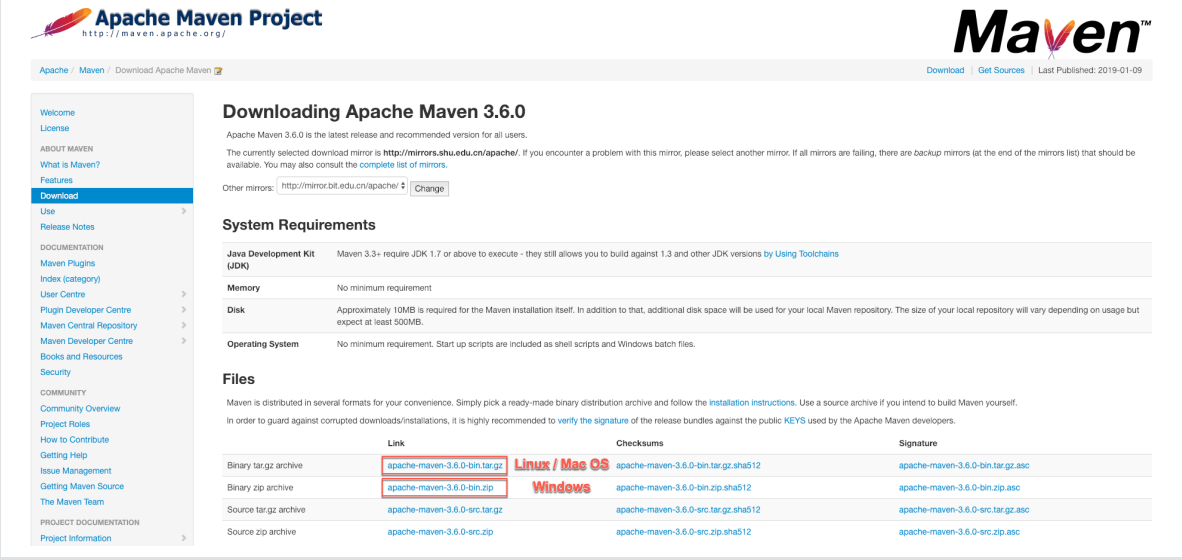
3.1 Maven的下载

> 下载地址：http://maven.apache.org
>
> 注意事项：Maven 3.3+ 需要执行JDK1.7或更高版本

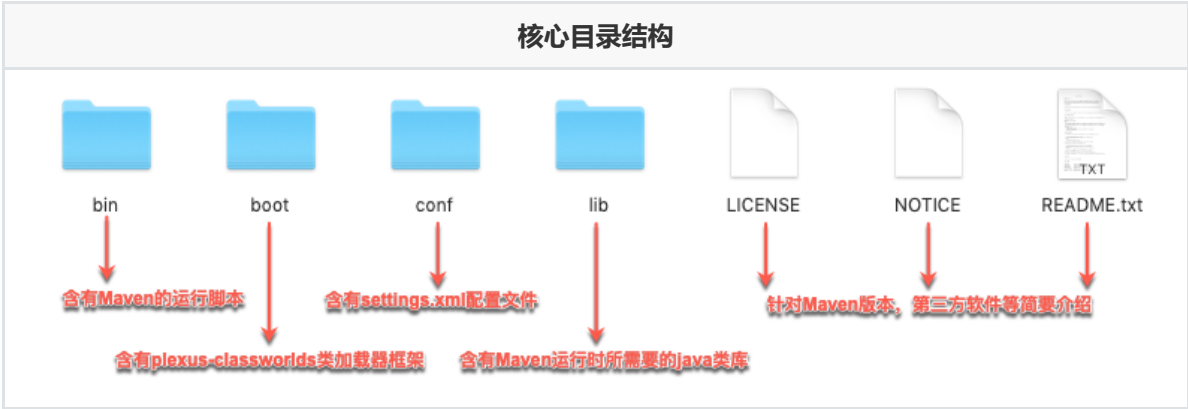
Maven官网下载位置



版本选择



3.2 Maven 目录结构



3.3 配置Maven安装流程

第一步：安装jdk，要求1.6或以上版本。

第二步：把maven解压缩，解压目录最好不要有中文。

第三步：配置环境变量MAVEN_HOME 【注意：Maven需要JAVA_HOME变量支持】

第四步：配置环境变量PATH，将%MAVEN_HOME%\bin加入Path中，在Windows中一定要注意要用分号；与其他值隔开。

第五步：验证是否安装成功，打开cmd窗口，输入mvn –v

3.4 修改settings.xml

1.修改本地仓库

我们需要修改一下配置文件，因为maven是依赖管理工具，其中依赖占有很大一部分，所谓的依赖就是通过你的pom.xml文件描写，自动去仓库下载jar并在项目中依赖，maven有一个非常好的特性，他会自动缓存下载过的jar,不用每个项目都进行网络下载！那么在本地缓存jar的文件夹，就叫做maven的本地仓库！默认位置：C:\Users\Administrator.m2\repository!

因为默认的本地仓库在c盘！我们也可以指定本地仓库位置！如果想指定我们需要修改settings配置文件!

文件位置： apache-maven-3.x.x / conf / settings.xml

```
<!--第一步：在根结点下添加本地仓库位置-->
<localRepository>F:\repository\maven</localRepository>
```

2.修改远程仓库

maven会默认指定一个远程仓库（http://repo1.maven.org/maven2/）！如果本地查找不到数据，会到远程仓库查找，考虑到远程仓库的服务器在国外，虽然可以访问，但是速度相对较慢!

国内的很多大厂都配置了自己的服务器，并且是外网，而且免费开放！那么我们就可以使用国内的远程仓库！下面我们推荐阿里远程服务器镜像!

```
<mirrors>
     <!--在mirrors节点下添加中央仓库镜像-->
    <mirror>
        <id>alimaven</id>
        <name>aliyun maven</name>
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
        <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
```

3.修改默认jdk编译版本

maven进行jar项目构建，默认选择的jdk的版本是1.5!现在开发jdk版本通常是1.7 1.8！那我们有两种方式修改使用maven编译项目的jdk版本！

修改maven的配置文件，全局修改的好处就是，修改一次，后面在使用maven进行构建项目的版本都是修改后的！

mavenrootdir/config/settings.xml

```
 <profiles>
   <profile>
     <id>jdk8</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>
```

3.5 使用maven手动创建工程

Project

|-src

| |-main

| | |-java —— 存放项目的.java文件

| | |-resources —— 存放项目资源文件，如spring, hibernate配置文件

| | |-webapp ———存放jsp，html，web.xml等网页文件 WebContent/WebRoot

| |-test 测试代码

| |-java ——存放所有测试.java文件，如JUnit测试类

| |-resources —— 测试资源文件

|-target —— 目标文件输出位置例如.jar、.war文件,该文件夹是自动创建的

|-pom.xml　　——maven项目核心配置文件,依赖管理，插件管理

1. 创建符合maven工程文件夹

   main/java:com.atguigu.demo.Hello.java

```
package com.atguigu.demo;

/**
 * author: 赵伟风
 * description: 准备maven的编译类
 */
public class Hello {

    public void eat(){
        System.out.println("吃的方法");
    }

}
```

   Test/java:com.atguigu.demo.HelloTest.java

```
package com.atguigu.demo;


import org.junit.Test;

/**
 * author: 赵伟风
 * description:
 */
public class HelloTest {
    @Test
    public void test1(){
        new Hello().eat();
    }
}
```

   pom.xml

   project object model:项目对象模型!

   这个配置文件中，配置了项目所需要的所有依赖以及项目一些基本信息!

```
<?xml version="1.0" encoding="UTF-8"?>
<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</groupId>
    <artifactId>testversionold</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
```

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

2. 使用命令进行编译

   mvn compile

3. 使用命令进行打包

   mvn package

需要了解两点：

maven不是编码工具，而是一个项目构建工具,依赖管理工具

maven项目配置核心是pom.xml文件，内部标明了项目参数，同时maven提供了一套项目构建命令！

3.6 IDEA中配置Maven

我们要跳过繁琐的手动创建maven工程步骤，因为单独是用maven进行项目编译和开发是不现实的！

我们绝大部分开发模式都是开发工具+maven!

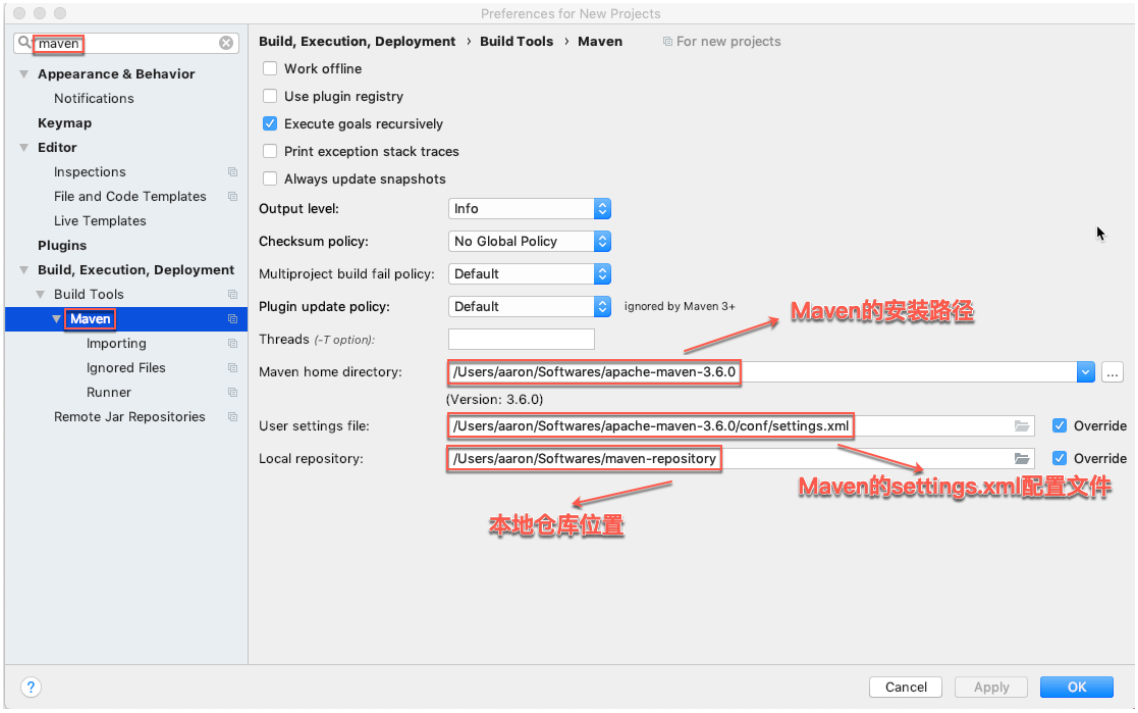
开发工具负责编码！

meven负责项目构建和依赖管理！

下面课程，我们都是使用idea+maven的模式！

注意：开发工具中绝大部分情况都自带maven工具！我们需要做的是将我们本地的maven配置到idea开发工具中，替换他们自带的！当然真正开发中，我们可以直接使用默认的！

核心配置



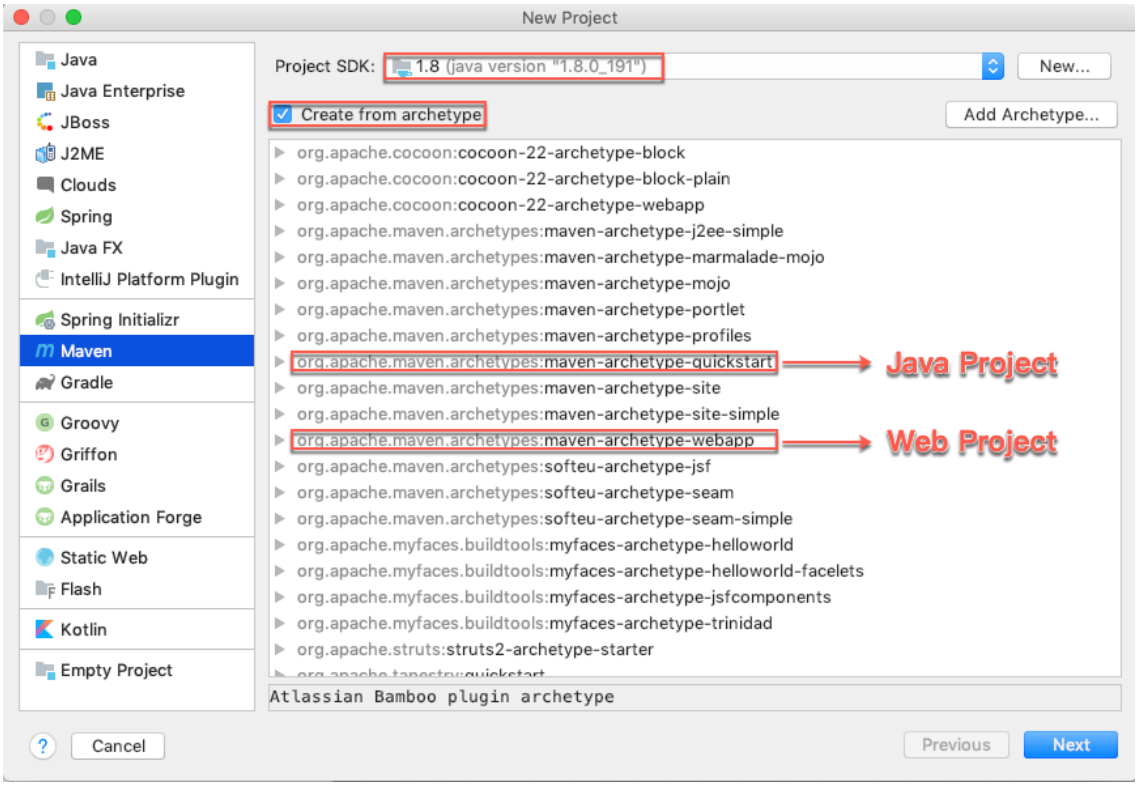
四、Mave开发项目

本一章节！我们着重介绍，idea工具下如何创建maven构建的项目！

并详细介绍maven创建的项目参数的含义以及项目结构！

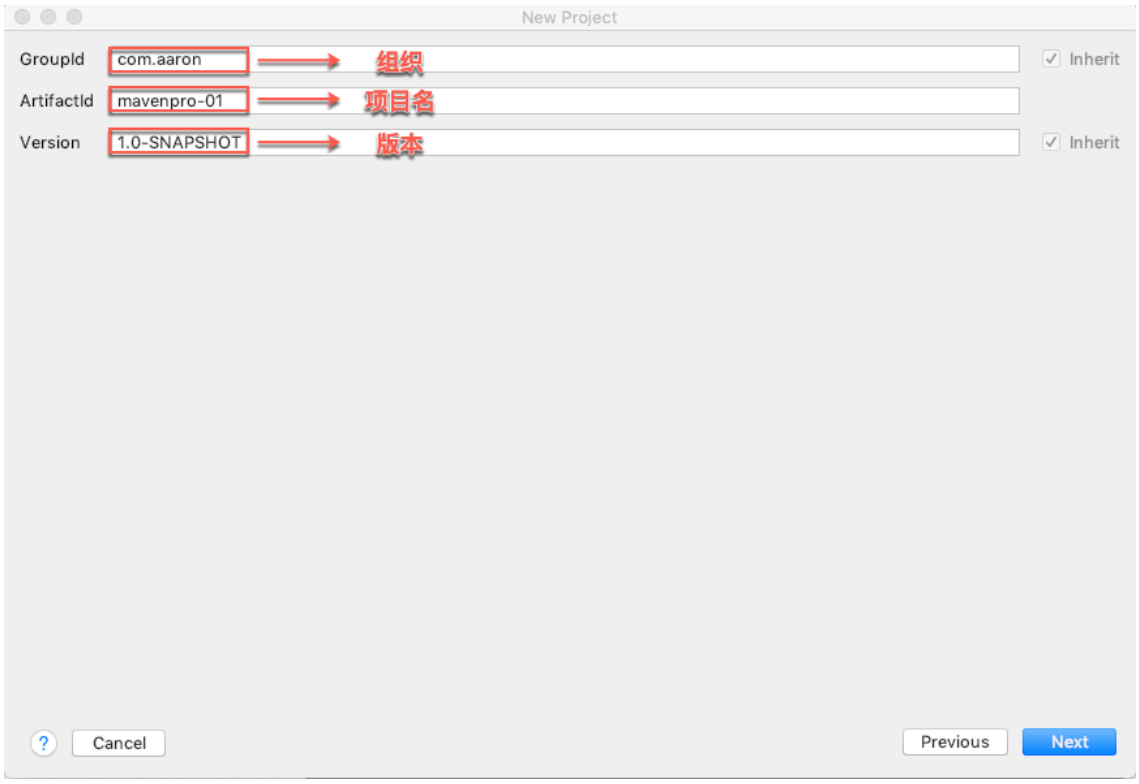
4.1 Idea创建Maven项目(普通java工程和web工程)

新建项目



4.2 GAV属性

GAV坐标



节点	详细描述
groupId	这是项目组的编号，这在组织或项目中通常是独一无二的。 例如，一家银行集 `com.company.bank` 拥有所有银行相关项目。
artifactId	这是项目的 ID。这通常是项目的名称。 例如，`consumer-banking`。 除了 groupIc artifactId 还定义了 artifact 在存储库中的位置。
version	这是项目的版本。与 groupId 一起使用，artifact 在存储库中用于将版本彼此分离。 `com.company.bank:consumer-banking:1.0`，`com.company.bank:consumer-bank`

4.3 语义化版本号

版本类型	详细描述
主要版本	当你做了不兼容的API 修改（正式版发布、架构升级）
次要版本	当你做了向下兼容的功能性新增（功能增减）
修订版本	当你做了向下兼容的问题修正（BUG修复、查缺补漏）

- 规则：正式稳定版本从v0.1.0开始，配套软件公共API。
- 注意：正式版发布后不可修改，只能在下一个版本中发布新内容。

4.4 扩展-SNAPSHOT

在使用maven过程中，我们在开发阶段经常性的会有很多公共库处于不稳定状态，随时需要修改并发布，可能一天就要发布一次，遇到bug时，甚至一天要发布N次。我们知道，maven的依赖管理是基于版本管理的，对于发布状态的artifact，如果版本号相同，即使我们内部的镜像服务器上的组件比本地新，maven也不会主动下载的。如果我们在开发阶段都是基于正式发布版本来做依赖管理，那么遇到这个问题，就需要升级组件的版本号，可这样就明显不符合要求和实际情况了。但是，如果是基于快照版本，那么问题就自热而然的解决了，而maven已经为我们准备好了这一切。

在maven的约定中，依赖的版本分为两类——SNAPSHOT和RELEASE。SNAPSHOT依赖泛指以-SNAPSHOT为结尾的版本号，例如1.0.1-SNAPSHOT。除此之外，所有非-SNAPSHOT结尾的版本号则都被认定为RELEASE版本，即正式版，虽然会有beta、rc之类说法，但是这些只是软件工程角度的测试版，对于maven而言，这些都是RELEASE版本!

4.5 项目结构介绍

Project

|-src

| |-main

| | |-java —— 存放项目的.java文件

| | |-resources —— 存放项目资源文件，如spring, hibernate配置文件

| | |-webapp ———存放jsp，html，web.xml等网页文件 WebContent/WebRoot

| |-test 测试代码

| |-java ——存放所有测试.java文件，如JUnit测试类

| |-resources —— 测试资源文件

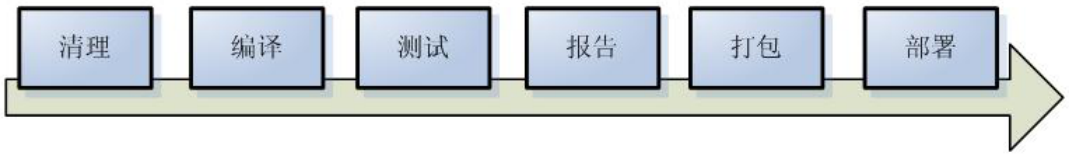
|-target —— 目标文件输出位置例如.jar、.war文件,该文件夹是自动创建的

|-pom.xml ——maven项目核心配置文件,依赖管理，插件管理

五、maven构建命令

我们应该了解！一个项目（文件夹） 要通过一系列的操作，最终才能打包编译成合适的结构 进行发布或者使用!

一般情况下项目都要经历一下几个过程：清理、初始化、编译、测试、打包、集成测试、验证、部署和站点生成!



清理：清理项目之前编译的内容，为再次编译等动作准备

初始化：初始化一些配置参数

编译：进行源文件的编译！java-class

测试：执行项目中的测试模块

打包：java和web项目打成不同形式的包！进行发布和使用！java项目jar包！web项目war包！

集成测试：打包后测试

验证：验证文件是否存在错误！

部署：进行项目部署工作

站点生成：生成对应的项目解释文档

但是之前我们并不是很在意！因为之前使用eclipse或者idea也好！我们创建的普通项目！过程都由工具帮我们操作！

5.1 常用的maven项目构建命令

命令	描述
mvn compile	编译项目，生成target文件
mvn package	打包项目，生成war文件
mvn clean	清理编译或打包后的项目结构
mvn install	打包后上传到Maven本地仓库
mvn deploy	只打包，不测试
mvn site	生成站点
mvn test	执行测试源码

直接在项目下面就可以直接执行！

我们比较常用的 compile-package-clean-install-deploy

5.2 maven项目生命周期

我们会发现一个问题！当我们执行package命令也会自动执行compile命令！

这种行为就是生命周期行为！也就是会自动的执行当前命令前面的命令！

1. clean生命周期

   主要是对项目编译生成文件进行清理

   我只需要记住，此阶段为清理工作

   触发命令： mvn clean

   生命周期命令：clean

```
[INFO] ------------------------------------------------------------------------
[INFO] Building ssm_integrate Maven Webapp 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ ssm_integrate ---
[INFO] Deleting /Users/zhaoweifeng/IdeaProjects/ssm_integrate/target
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 0.197 s
[INFO] Finished at: 2019-05-24T14:04:02+08:00
[INFO] Final Memory: 8M/245M
[INFO] ------------------------------------------------------------------------
```

2. default生命周期
   default生命周期定义了真正构件时所需要执行的所有步骤，它是生命周期中最核心的部分

   此生命周期的主要目的就是项目编译-测试-打包-发布！

   触发命令：mvn deploy

   声明周期命令：resources - compile - test resources - testCompile — test - war - install - deploy

```
--- maven-resources-plugin:2.6:resources (default-resources) @ ssm_integrate ---
NG] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform depende
Copying 6 resources

--- maven-compiler-plugin:3.1:compile (default-compile) @ ssm_integrate ---
Nothing to compile - all classes are up to date

--- maven-resources-plugin:2.6:testResources (default-testResources) @ ssm_integrate ---
NG] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform depende
skip non existing resourceDirectory /Users/zhaoweifeng/IdeaProjects/ssm_integrate/src/test/resources

--- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ ssm_integrate ---
No sources to compile

--- maven-surefire-plugin:2.12.4:test (default-test) @ ssm_integrate ---
No tests to run.

--- maven-war-plugin:2.2:war (default-war) @ ssm_integrate ---
Packaging webapp
Assembling webapp [ssm_integrate] in [/Users/zhaoweifeng/IdeaProjects/ssm_integrate/target/ssm]
Processing war project
Copying webapp resources [/Users/zhaoweifeng/IdeaProjects/ssm_integrate/src/main/webapp]
Webapp assembled in [400 msecs]
Building war: /Users/zhaoweifeng/IdeaProjects/ssm_integrate/target/ssm.war
WEB-INF/web.xml already added, skipping

--- maven-install-plugin:2.4:install (default-install) @ ssm_integrate ---
Installing /Users/zhaoweifeng/IdeaProjects/ssm_integrate/target/ssm.war to /Users/zhaoweifeng/.m2/repo
Installing /Users/zhaoweifeng/IdeaProjects/ssm_integrate/pom.xml to /Users/zhaoweifeng/.m2/repository/

--- maven-deploy-plugin:2.7:deploy (default-deploy) @ ssm_integrate ---
------------------------------------------------------------------------
```

3. site生命周期

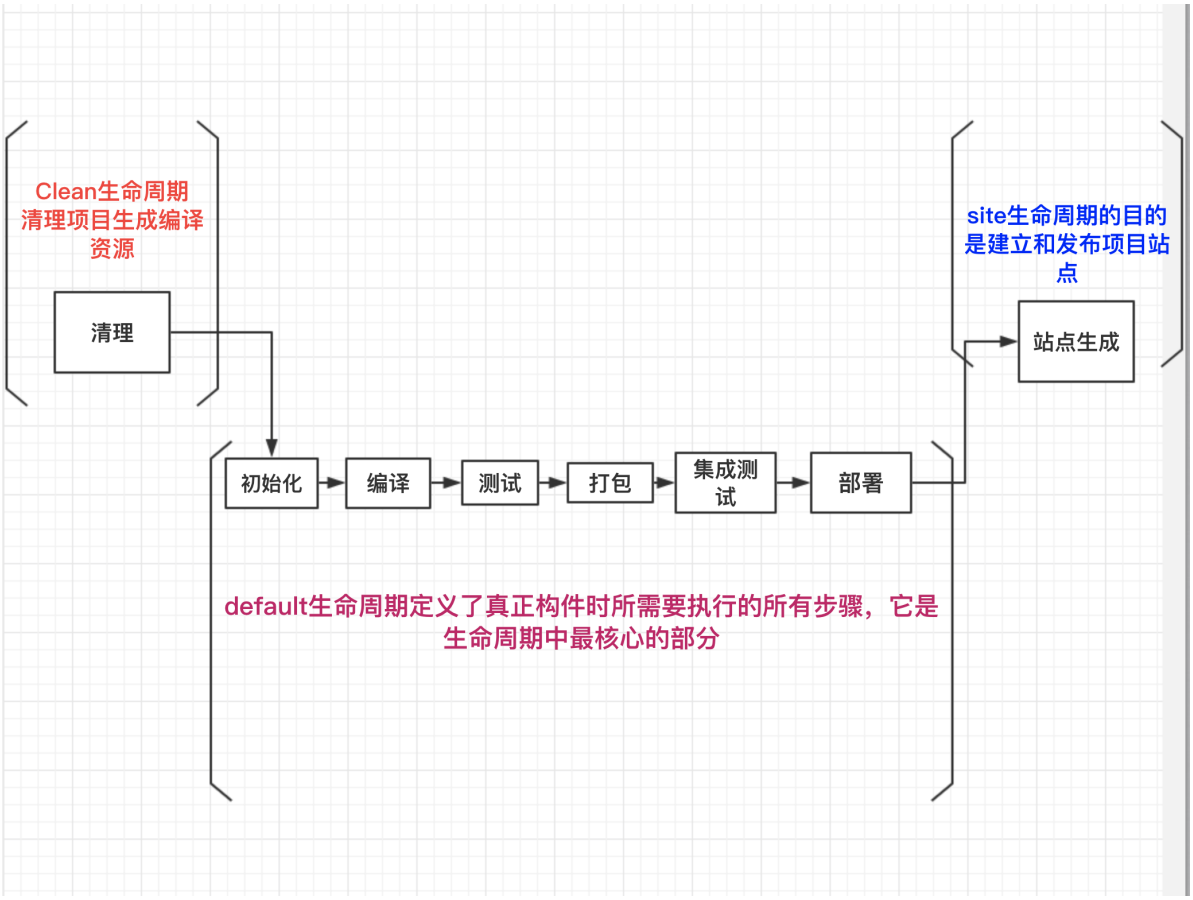
   site生命周期的目的是建立和发布项目站点，Maven能够基于POM所包含的信息，自动生成一个友好的站点

   触发命令：mvn site

   问题：低版本的site插件可能引发失败现象！升级高版本site插件即可

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.7.1</version>
</plugin>
```

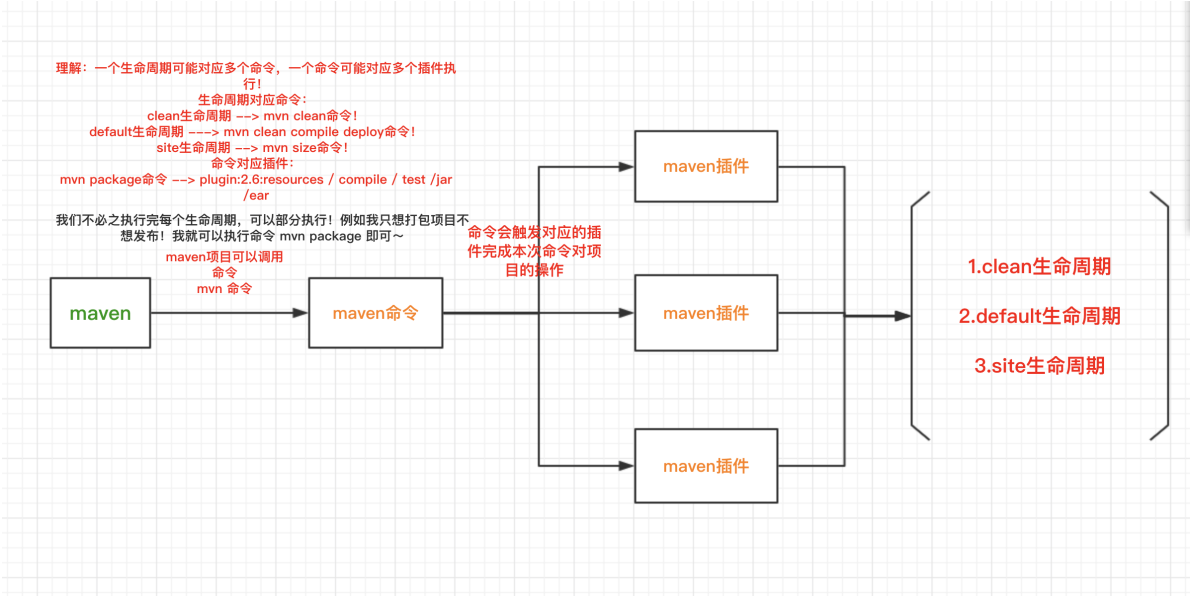
```
[INFO]
[INFO] --- maven-site-plugin:3.7.1:site (default-site) @ ssm_integrate ---
[WARNING] Input file encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
[WARNING] Report plugin org.apache.maven.plugins:maven-project-info-reports-plugin has an empty version.
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
```



生命周期理解

5.3 Maven命令与插件的关系

> maven命令是操作maven项目的重要方式！但是我们要知道，真正干活的是插件！maven内嵌了项目操作插件！
>
> 通过执行命令调用插件完成项目编译测试发布等工作！
>
> 还要注意：一个生命周期可能由多个命令组成！
>
> 执行一次个命令，可能触发多个插件操作！



六、pom配置文件讲解

> 这一章节主要讲解pom文件的写法！重点学习如何引入资源以及部分插件配置等！

POM(Project Object Model，项目对象模型) 是 Maven 工程的基本工作单元，是一个XML文件，包含了项目的基本信息，用于描述项目如何构建，声明项目依赖，等等。

执行任务或目标时，Maven 会在当前目录中查找 POM。它读取 POM，获取所需的配置信息，然后执行目标。

POM 中可以指定以下配置：

- 项目的gva
- 项目依赖
- 插件
- 执行目标
- 项目构建 profile
- 项目版本

6.1 项目核心属性

项目核心属性，就是项目打包的基本信息！

```
<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">

    <!-- 模型版本 -->
    <modelVersion>4.0.0</modelVersion>
    <!-- 公司或者组织的唯一标志，并且配置时生成的路径也是由此生成， 如
com.companyname.project-group，maven会将该项目打成的jar包放本地路
径：/com/companyname/project-group -->
    <groupId>com.companyname.project-group</groupId>

    <!-- 项目的唯一ID，一个groupId下面可能多个项目，就是靠artifactId来区分的 -->
    <artifactId>project</artifactId>

    <!-- 版本号 -->
```

```
    <version>1.0.0</version>

    <!--打包方式
          默认：jar
            jar指的是普通的java项目打包方式！ 项目打成jar包！
            war指的是web项目打包方式！项目打成war包！
              pom不会讲项目打包！这个项目作为父工程，被其他工程聚合或者继承！后面会讲解两个概念
    -->
    <packaging>jar/pom/war</packaging>
</project>
```

所有 POM 文件都需要 project 元素和三个必需字段：groupId，artifactId，version，packaging

节点	描述
project	工程的根标签。
modelVersion	模型版本需要设置为 4.0。
groupId	这是工程组的标识。它在一个组织或者项目中通常是唯一的。例如，一个银行组织 com.companyname.project-group 拥有所有的和银行相关的项目。
artifactId	这是工程的标识。它通常是工程的名称。例如，消费者银行。groupId 和 artifactId 一起定义了 artifact 在仓库中的位置。
version	这是工程的版本号。在 artifact 的仓库中，它用来区分不同的版本。例如：`com.company.bank:consumer-banking:1.0 com.company.bank:consumer-banking:1.1`

6.2 依赖管理和版本统一管理

> maven还一项比较重要的内容就是引入依赖包！
>
> 这样大大的减少了导包的繁琐程度，也让团队开发版本更加统一

依赖包引入

依赖包引用的配置也出现在pom.xml中，他的根标签是

具体引入标签！

内部包含四个子标签：

```
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
        <!--
            生效范围
            - compile ： 编译 测试 运行 打包
            - provided：编译和测试  servlet
            - runtime：测试 运行 打包  MySQL
            - test:     用于test任务时使用 junit
         -->
        <scope>runtime</scope>
```

```
----------  上面是gav  --------------

    <!-- 依赖包！如果有依赖管理器！那么此处可以不用写版本号！
         如果不写版本号，使用包管理器版本号！
         如果写版本号！则使用新版本
     -->
    <dependencies>
        <!-- 引入具体的依赖包 -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
            <!--
                生效范围
                - compile ：编译 测试 运行 打包 [默认]
                - provided：编译和测试  servlet
                - runtime：测试 运行 打包  MySQL
                - test:    用于test任务时使用 junit
             -->
            <scope>runtime</scope>
        </dependency>

    </dependencies>
```

依赖范围

依赖范围（Scope）	对于主代码classpath有效	对于测试代码classpath有效	被打包，对于运行时classpath有效	例子
compile	Y	Y	Y	log4j
test	-	Y	-	junit
provided	Y	Y	-	servlet-api
runtime	-	-	Y	JDBC Driver Implementation

其中依赖范围**scope** 用来控制依赖和编译，测试，运行的classpath的关系. 主要的是三种依赖关系如下：

1.compile： 默认编译依赖范围。对于编译，测试，运行三种classpath都有效

2.test：测试依赖范围。只对于测试classpath有效

3.provided：已提供依赖范围。对于编译，测试的classpath都有效，但对于运行无效。因为由容器已经提供，例如servlet-api

4.runtime:运行时提供。例如:jdbc驱动

优化写法

优化写法直接在上面进行版本声明！下面引用！

```
<!--声明版本-->
<properties>
  <!--命名随便-->
  <junit.version>4.11</junit.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <!--引用 -->
    <version>${junit.version}</version>
  </dependency>
</dependencies>
```

全局编码格式设置

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <maven.compiler.encoding>UTF-8</maven.compiler.encoding>
</properties>
```

6.3 Build标签使用

> 除了依赖的加入，gav的配置，还有一些其他的内容，例如插件，资源管理，打包命名等！build标签中包含插件标签和资源管理标签，也都是pom文件中的重点内容！

1.修改默认打包名

默认的打包名称：artifactid+verson.打包方式

我们可以通过build中finalName修改！

```
<build>
  <finalName>定义打包名称</finalName>
</build>
```

2.引入插件

dependencies引入开发需要的jar包！我们有时还需要倒入一些插件，插件可以辅助我们做一些其他工作！

常用的插件：修改jdk版本，tomcat插件！后期要学的分页插件！

插件配置位置也在Build标签中

build/plugins/plugin

```
<build>
<plugins>
    <!-- java编译插件，配jdk的编译版本 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
    <!-- tomcat插件 -->
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <configuration>
        <port>8090</port>
        <path>/</path>
        <uriEncoding>UTF-8</uriEncoding>
        <server>tomcat7</server>
      </configuration>
    </plugin>
  </plugins>
</build>
```

3.控制打包资源

如果在java文件夹中添加java类，会自动打包编译到classes文件夹下！

但是xml文件默认不会被打包！需要我们自己指定！

我们可以使用resources，指定要打包资源的文件夹 要把哪些静态资源打包到 classes根目录下

```
<!--打包指定静态资源-->
<build>
<resources>
    <resource>
      <!-- 指定要打包资源的文件夹 要把哪些静态资源打包到 classes根目录下-->
      <directory>src/main/resources</directory>
      <includes>
        <include>**/*.xml</include>
        <include>**/*.properties</include>
      </includes>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
      <excludes>
        <exclude>spring/*</exclude>
      </excludes>
      <includes>
        <include>*.xml</include>
        <!--<include>*/*.properties</include>-->
      </includes>
    </resource>
  </resources>
</build>
```

七、依赖传递以及冲突解决（了解）

7.1 概念简介

1.依赖传递

假如有Maven项目A，项目B依赖A，项目C依赖B。那么我们可以说 C依赖A。也就是说，依赖的关系为：C—>B—>A， 那么我们执行项目C时，会自动把B、A都下载导入到C项目的jar包文件夹中，这就是依赖的传递性。

2.依赖冲突

当直接引用或者间接引用出现了相同的jar包，不同版本的时候，这就算作冲突！碰到引用相同包的概率很大！

但是，也不用过于担心，maven有着强大的依赖冲突解决能力！在你不知情的情况下就解决问题，同时也提供了手动解决的冲突的方式！

7.2 依赖传递演示

7.3 冲突解决

1.依赖排除（手动处理）

如果我不想在c中出现b!那么我可以主动的使用依赖排除技术，排除 b的引用！

使用的时机就是当c依赖a的时刻！

```
<dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>4.3.12.RELEASE</version>
            <!--手动排除-->
            <exclusions>
                <exclusion>
                    <groupId>commons-logging</groupId>
                    <artifactId>commons-logging</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
    </dependencies>
```

2.短路优先原则（自动处理）

A—>B—>C—>D—>E—>X(version 0.0.1)

A—>F—>X(version 0.0.2)

则A依赖于X(version 0.0.2)。

3.依赖路径长度相同情况下，则“先声明优先”（自动处理）

A—>E—>X(version 0.0.1)

A—>F—>X(version 0.0.2)

则在项目A的中，E、F那个在先则A依赖哪条路径的X。

八、继承和聚合

8.1 继承

> 继承为了消除重复，我们把很多相同的配置提取出来，例如：grouptId，version等，
>
> 最关键的是子模块能能直接得到父工程的依赖,或者父工程进行版本管理!

8.2 聚合

> Maven的聚合特性能够把项目的各个模块聚合在一起构建!
>
> 聚合通常组合继承一起使用!

父项目

```
<?xml version="1.0" encoding="UTF-8"?>
<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.qf</groupId>
    <artifactId>shop_manage</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!-- 聚合语法 -->
    <modules>
        <module>shop_common</module>
        <module>shop_web</module>
        <module>shop_service</module>
    </modules>

    <packaging>pom</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.test.skip>true</maven.test.skip>
        <fastjson.version>1.2.70</fastjson.version>
        <jackson.databind.version>2.9.8</jackson.databind.version>
        <hibernate.validator.version>6.0.15.Final</hibernate.validator.version>
        <mysql.connector.version>5.1.47</mysql.connector.version>
        <mybatis.plus.version>3.4.2</mybatis.plus.version>
        <swagger2.version>2.7.0</swagger2.version>
        <shiro.version>1.4.0</shiro.version>
        <servlet.version>3.1.0</servlet.version>
        <druid.version>1.2.6</druid.version>
        <spring.version>5.1.6.RELEASE</spring.version>
```

```
        <mybatis.spring.version>2.0.0</mybatis.spring.version>
        <lombok.version>1.18.6</lombok.version>
        <junit.version>4.12</junit.version>
        <aspectj.version>1.9.2</aspectj.version>
        <log4j.version>1.2.17</log4j.version>
        <self4j.version>1.7.21</self4j.version>
        <commons.logging.version>1.2</commons.logging.version>
        <commons.lang3.version>3.7</commons.lang3.version>
        <aliyun.oss.version>3.10.2</aliyun.oss.version>
        <commons.fileupload.version>1.3.2</commons.fileupload.version>
        <commons.io.version>2.5</commons.io.version>
    </properties>

    <!--依赖管理-->
    <dependencyManagement>
        <dependencies>
            <!--文件上传-->
            <dependency>
                <groupId>commons-fileupload</groupId>
                <artifactId>commons-fileupload</artifactId>
                <version>${commons.fileupload.version}</version>
            </dependency>
            <dependency>
                <groupId>commons-io</groupId>
                <artifactId>commons-io</artifactId>
                <version>${commons.io.version}</version>
            </dependency>
            <dependency>
                <groupId>com.aliyun.oss</groupId>
                <artifactId>aliyun-sdk-oss</artifactId>
                <version>${aliyun.oss.version}</version>
            </dependency>
            <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
                <version>${self4j.version}</version>
            </dependency>

            <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>${self4j.version}</version>
            </dependency>
      </dependencies>
 <dependencyManagement>
```

子项目

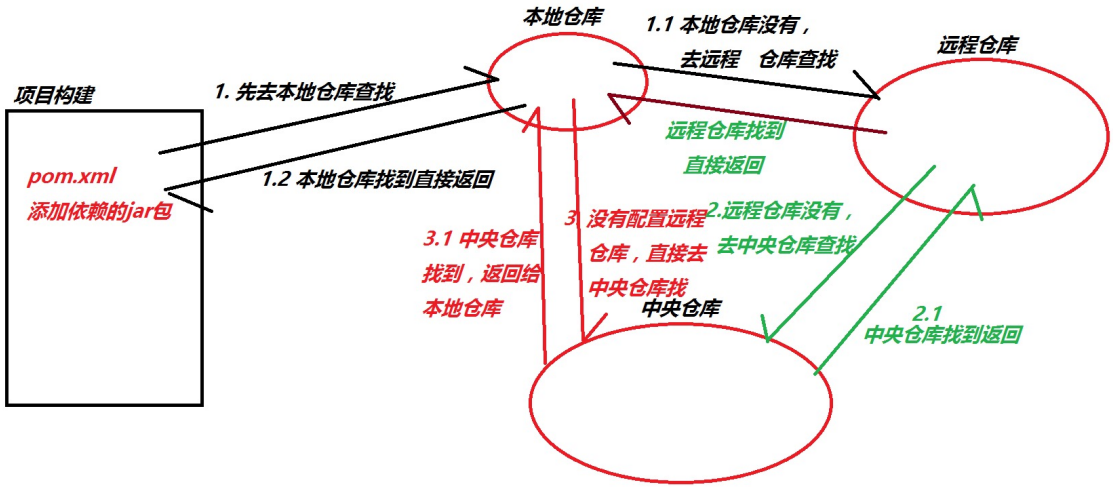
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>shop_manage</artifactId>
        <groupId>com.qf</groupId>
        <version>1.0-SNAPSHOT</version>
```

```
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>shop_web</artifactId>

    <!--表示这个是war包工程-->
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

    <dependencies>
        <!--依赖于service工程-->
        <dependency>
            <groupId>com.qf</groupId>
            <artifactId>shop_service</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>

        <!--文件上传-->
        <dependency>
            <groupId>commons-fileupload</groupId>
            <artifactId>commons-fileupload</artifactId>
        </dependency>
        <dependency>
            <groupId>commons-io</groupId>
            <artifactId>commons-io</artifactId>
        </dependency>

        <!--springmvc依赖-->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
        </dependency>



        <!--aop的第三方依赖-->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
        </dependency>

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
        </dependency>

        <!--shiro-->
        <dependency>
            <groupId>org.apache.shiro</groupId>
            <artifactId>shiro-all</artifactId>
            <type>pom</type>
        </dependency>
```

```
        <!--swagger2 依赖-->
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
        </dependency>

        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
        </dependency>

        <dependency>
            <groupId>org.hibernate.validator</groupId>
            <artifactId>hibernate-validator</artifactId>
        </dependency>
    </dependencies>

</project>
```

九、maven仓库【了解】

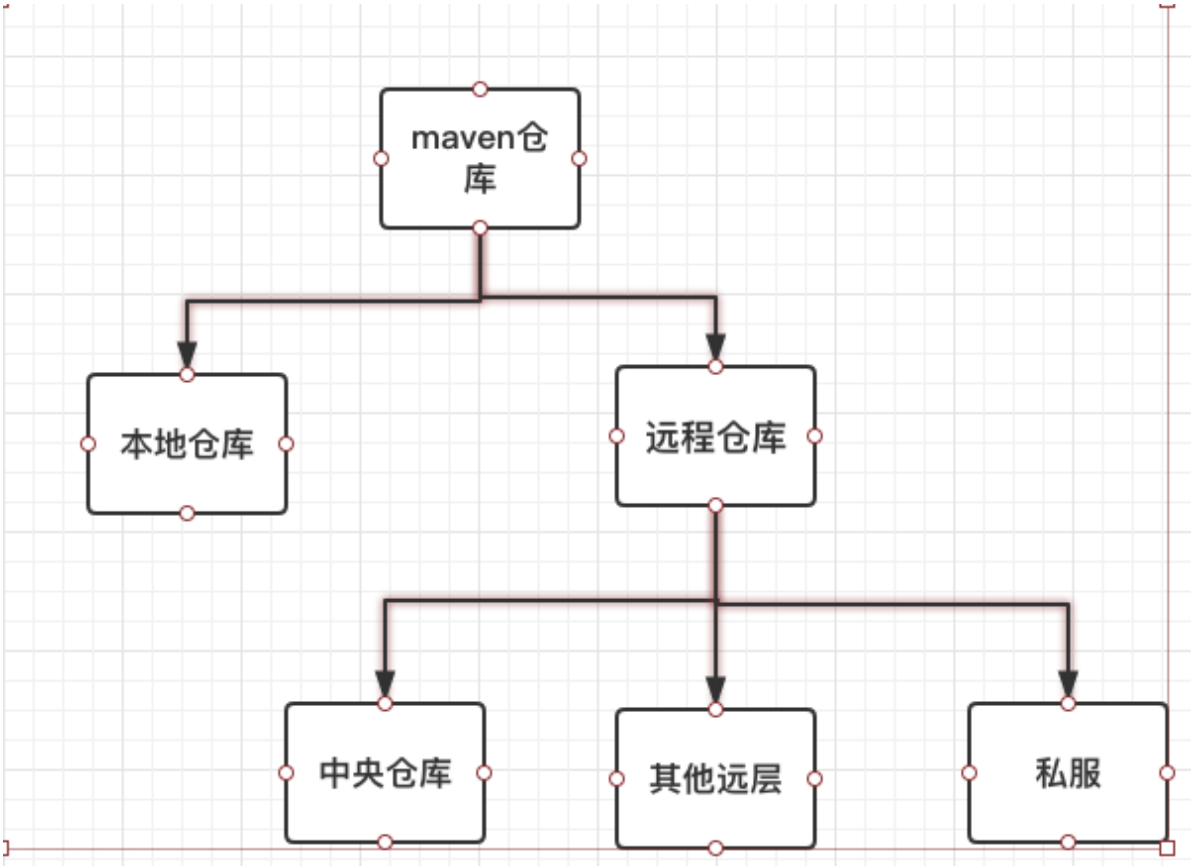
1. maven仓库图解



2.maven仓库介绍

maven仓库分为本地仓库和远程仓库，而远程仓库又包括私服和中央仓库。
本地仓库就是用户自己电脑上的仓库，直接从本地获取。
私服是一种特殊的远程仓库因为他是架设在局域网内的仓库服务，私服代理广域网上的远程仓库，供局域网内的maven用户使用。

中央仓库是maven公司提高的最大的仓库，里面拥有最全的jar包资源，所以私服上也没有的时候就会去中央仓库找作


maven仓库
本地仓库
远程仓库
中央仓库
其他远层
私服