

第十九章：Maven

第1节：简介

Maven这个词的本意是：专家，内行。读音是[`'meɪv(ə)n`]或[`'mevn`]，不要读作“妈文”。简单来说：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。

1.1 构建

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。

1.2 构建环节

- 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。

Maven可以自动的从构建过程的起点一直执行到终点

第2节：作用

Maven是干什么用的？这是很多同学在学完这个课程后最大的问题。之所以会提出这个问题，是因为即使不使用Maven我们仍然可以进行B/S结构项目的开发。从表述层、业务逻辑层到持久化层再到数据库都有成熟的解决方案——不使用Maven我们一样可以开发项目啊？所以我们有必要通过企业开发中的实际需求来看一看哪些方面是我们现有技术的不足。

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。

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。

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Demo1、Demo2、和Demo3。Demo1依赖Demo2，Demo2依赖Demo3。而Demo3依赖`junit.4.12.jar`，Demo2依赖`junit.4.11.jar`。

那么Demo1工程的运行时环境中该导入`junit.4.12.jar`呢？还是`junit.4.11.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Demo1工程会自动使用`junit.4.11.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！

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Java工程，有的是web工程。

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。

第3节：使用

3.1 下载安装

- 1) 下载路径: <http://maven.apache.org/download.cgi>
- 2) 解压: 要求目录不要过深、不要有中文和特殊符号
- 3) 必须配置JAVA_HOME 环境变量
- 4) 配置环境变量
 - 1) M2_HOME 值为Maven的解压路径
 - 2) 在path中配置 %M2_HOME%\bin
- 5) 查看安装是否正确: 在doc中 通过命令: `mvn -v`

3.2 配置本地仓库

- 1) Maven的核心程序并不包含具体功能，仅负责宏观调度。具体功能由插件来完成。Maven核心程序会到本地仓库中查找插件。如果本地仓库中没有就会从远程中央仓库下载。此时如果不能上网则无法执行Maven的具体功能。为了解决这个问题，我们可以将Maven的本地仓库指向一个在联网情况下下载好的目录。
- 2) Maven默认的本地仓库
C:\Users\ljw\.m2\repository目录。
- 3) Maven的核心配置文件位置
D:\javaTools3\apache-maven-3.5.2\conf\settings.xml
- 4) 在配置文件中设置本地仓库
`<localRepository>D:\javaTools3\RepMaven</localRepository>`
- 5) 在配置文件的<servers>中添加镜像服务名称
`<server>`
`<id>huaweicloud</id>`
`<username>anonymous</username>`
`<password>devcloud</password>`
`</server>`
- 6) 在配置文件的<mirrors> 添加镜像地址
`<mirror>`
`<id>huaweicloud</id>`

```

        <mirrorOf>*</mirrorOf>
        <url>https://mirrors.huaweicloud.com/repository/maven/</url>
    </mirror>
7) 在配置文件的<profiles>中添加JDK
    <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>

```

3.3 Maven程序

3.3.1 创建约定的目录结构

```

mvn_pro1
  src
    —main
    —java
    —resources
    —test
    —java
    —resources
  pom.xml

```

main目录用于存放主程序。

test目录用于存放测试程序。

java目录用于存放源代码文件。

resources目录用于存放配置文件和资源文件。

3.3.2 创建Maven的核心配置文件pom.xml

```

<?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.it</groupId>
    <artifactId>Demol</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>

```

```
<groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
</dependencies>
</project>
```

3.3.3 编写主代码

在src/main/java/com/it/maven目录下新建文件Demo1.java

```
package com.it.maven;
public class Demo1 {
    public String sayHello(String name){
        return "Hello "+name+"!";
    }
}
```

3.3.4 编写测试代码

在/src/test/java/com/it/maven目录下新建测试文件Demo1Test.java

```
package com.it.maven;
import org.junit.Test;
import static junit.framework.Assert.*;
public class Demo1Test {
    @Test
    public void testHello(){
        Demo1 Demo1 = new Demo1();
        String results = Demo1.sayHello("ljw");
        assertEquals("Hello ljw!", results);
    }
}
```

3.3.5 运行Maven命令

打开cmd命令行，进入Hello项目根目录(pom.xml文件所在目录)执行mvn compile命令，查看根目录变化

cmd 中继续录入mvn clean命令，然后再次查看根目录变化

cmd 中录入 mvn compile命令，查看根目录变化

cmd 中录入 mvn test-compile命令，查看target目录的变化

cmd 中录入 mvn test命令，查看target目录变化

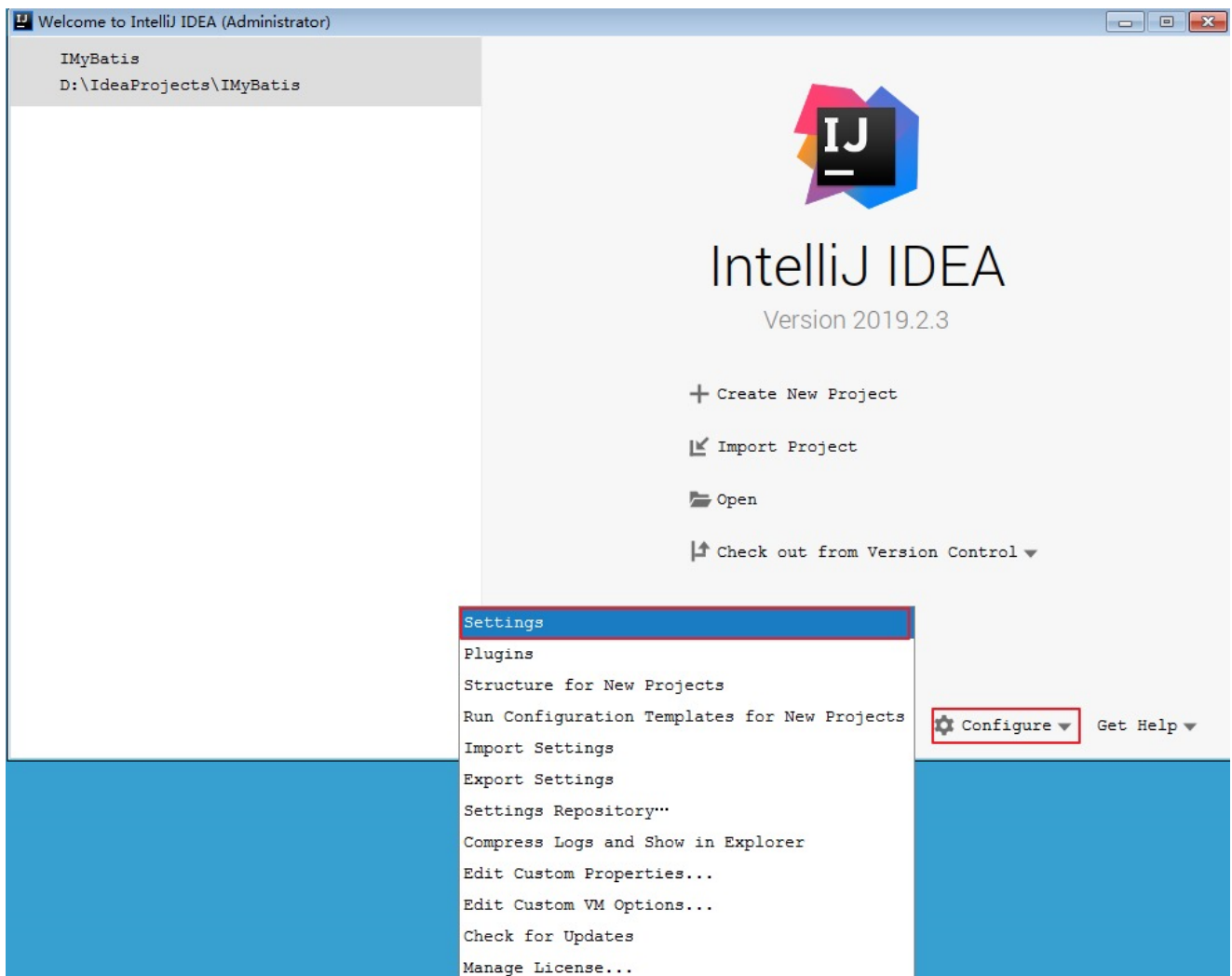
cmd 中录入 mvn package命令，查看target目录变化

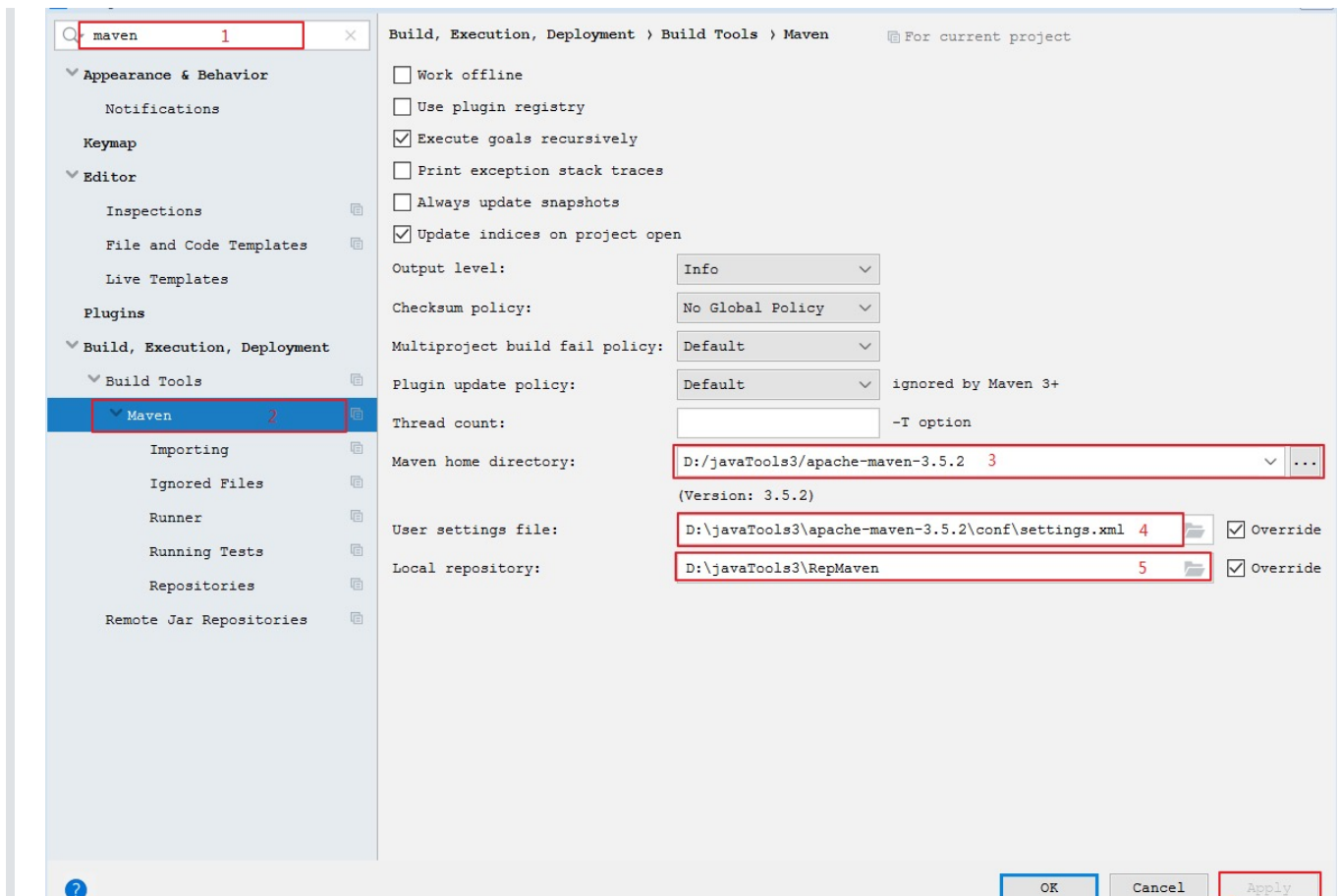
cmd 中录入 mvn install命令，查看本地仓库的目录变化

注意：运行Maven命令时一定要进入pom.xml文件所在的目录！

3.4 Idea整合Maven

1. 在idea中配置Maven





3.5 idea创建maven项目


3.5.1 创建Maven的Java项目





IntelliJ IDEA

Version 2019.2.3

+ Create New Project

 Import Project

 Open

 Check out from Version Control ▼

 Configure ▼ Get Help ▼

New Project

Project SDK: **1.8 (java version "1.8.0_131")** New...

☐ Create from archetype Add Archetype...

- > com.atlassian.maven.archetypes:bamboo-plugin-archetype
- > com.atlassian.maven.archetypes:confluence-plugin-archetype
- > com.atlassian.maven.archetypes:jira-plugin-archetype
- > com.rfc.maven.archetypes:jpa-maven-archetype
- > de.akquinet.jbosscc:jbosscc-seam-archetype
- > net.databinder:data-app
- > net.liftweb:lift-archetype-basic
- > net.liftweb:lift-archetype-blank
- > net.sf.maven-har:maven-archetype-har
- > net.sf.maven-sar:maven-archetype-sar
- > org.apache.camel.archetypes:camel-archetype-activemq
- > org.apache.camel.archetypes:camel-archetype-component
- > org.apache.camel.archetypes:camel-archetype-java
- > org.apache.camel.archetypes:camel-archetype-scala
- > org.apache.camel.archetypes:camel-archetype-spring
- > org.apache.camel.archetypes:camel-archetype-war
- > org.apache.cocoon:cocoon-22-archetype-block
- > org.apache.cocoon:cocoon-22-archetype-block-plain
- > org.apache.cocoon:cocoon-22-archetype-webapp
- > org.apache.maven.archetypes:maven-archetype-j2ee-simple
- > org.apache.maven.archetypes:maven-archetype-marmalade-mojo
- > org.apache.maven.archetypes:maven-archetype-mojo
- > org.apache.maven.archetypes:maven-archetype-portlet
- > org.apache.maven.archetypes:maven-archetype-profiles

Previous **Next 2** Cancel Help

New Project

GroupId: com.ujiuye.maven 组Id ☒ Inherit

ArtifactId: Demo4 项目名

Version: 1.0-SNAPSHOT ☒ Inherit

Previous Next Cancel Help

New Project

Project name: Demo4

Project location: D:\IdeaProjects\Demo4 ...

More Settings

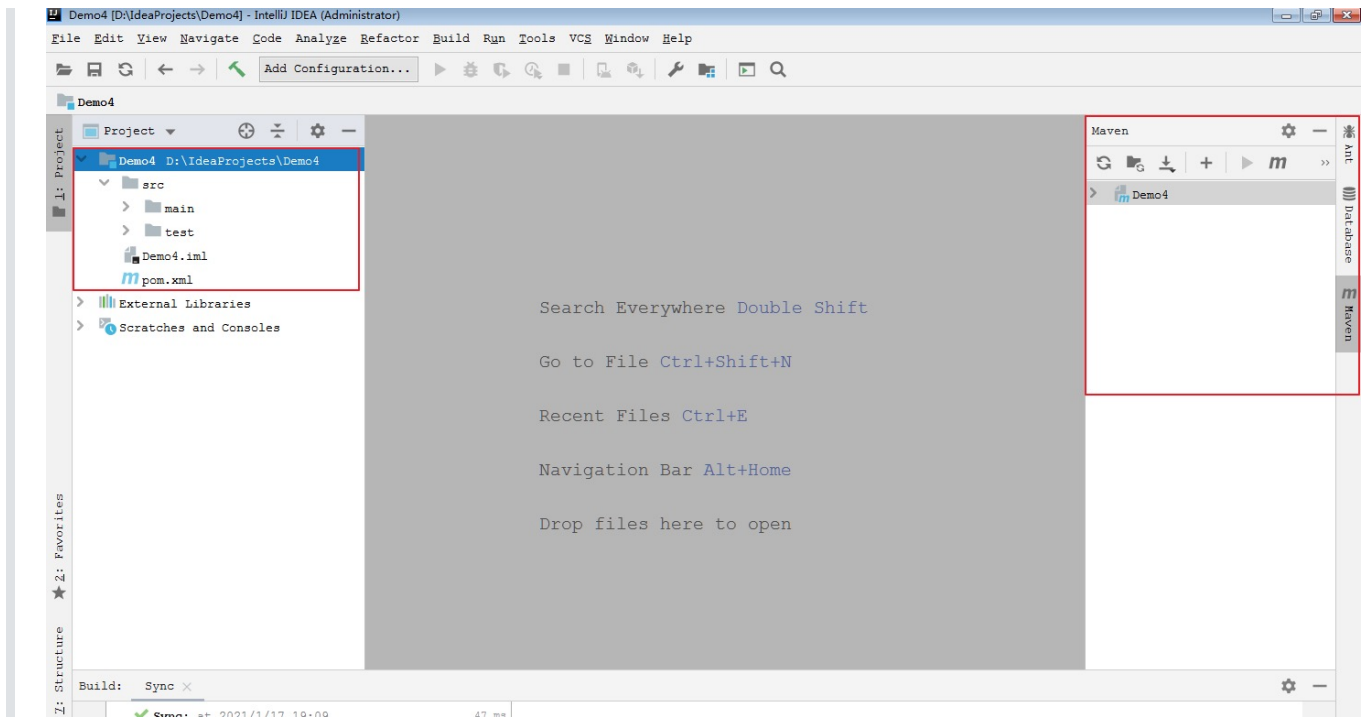
Module name: Demo4

Content root: D:\IdeaProjects\Demo4

Module file location: D:\IdeaProjects\Demo4

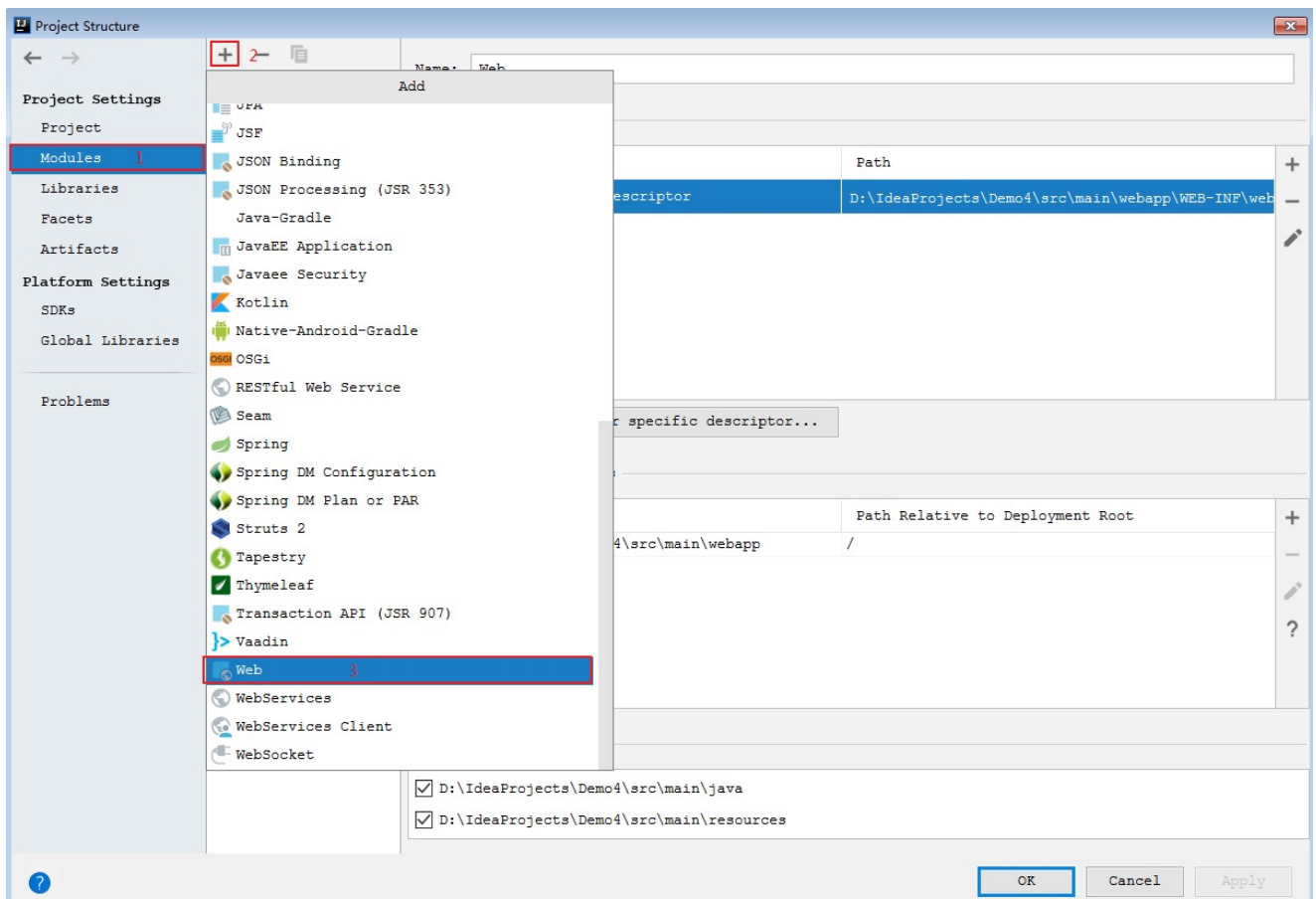
Project format: .idea (directory based)

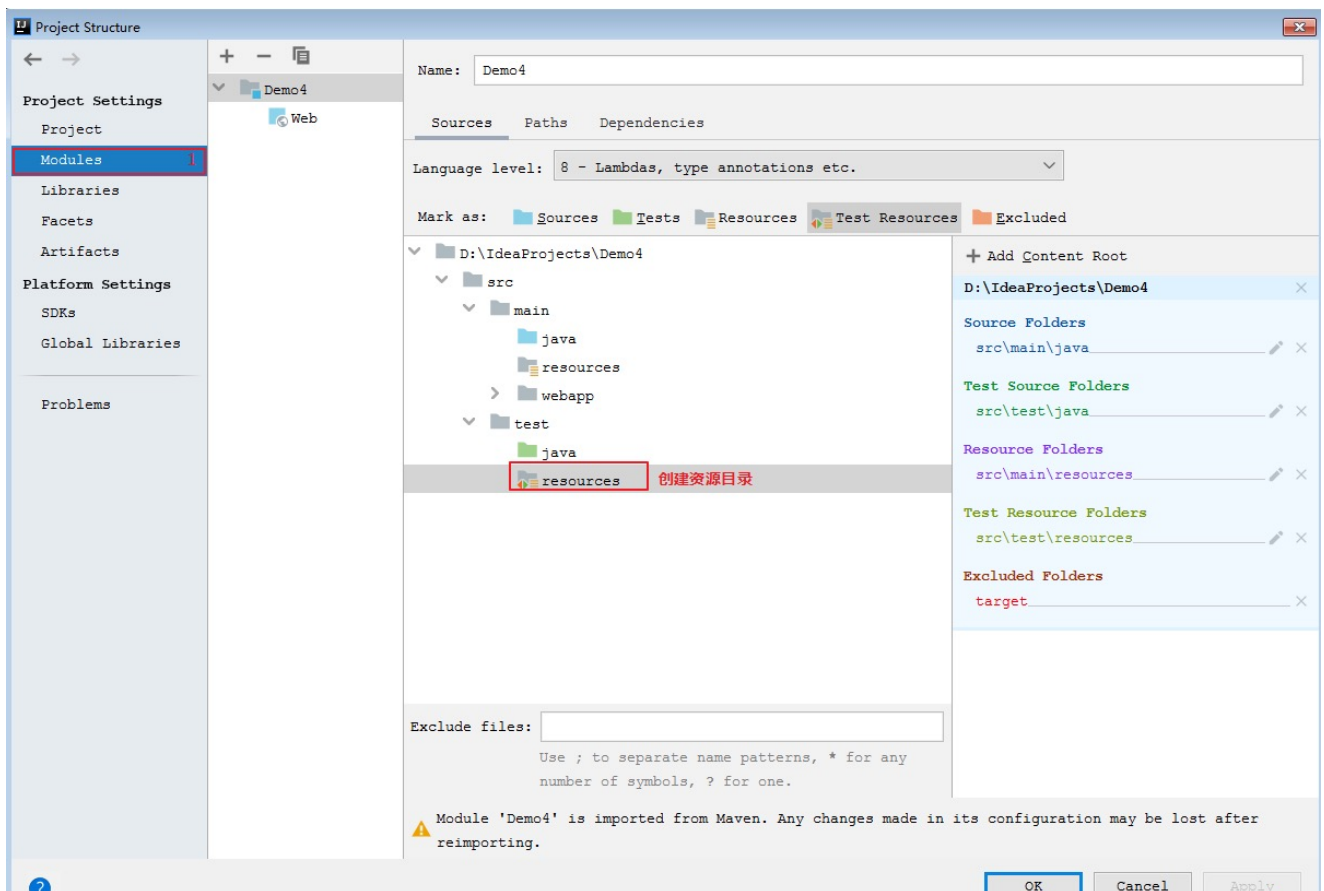
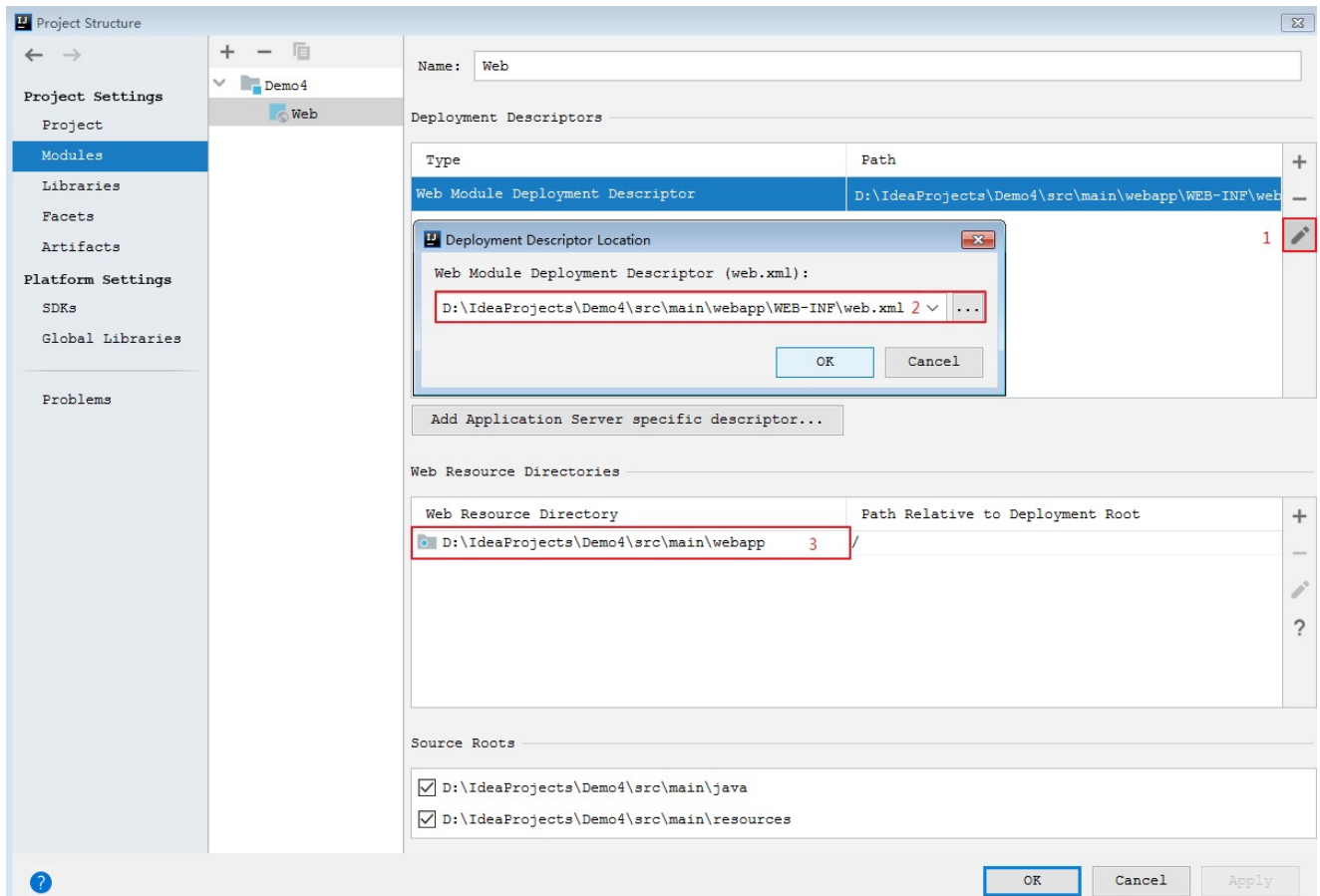
Previous Finish Cancel Help



3.5.2 创建Maven的JavaWeb项目

在创建Java项目的基础上





第4节：核心概念

相关概念

1) POM 2) 约定的目录结构 3) 坐标 4) 依赖 5) 仓库 6) 生命周期 7) 插件和目标 8) 继承 9) 聚合

4.1 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。

4.2 目录结构

现在JavaEE开发领域普遍认同一个观点：约定>配置>编码。意思就是能用配置解决的问题就不编码，能基于约定的就不进行配置。而Maven正是因为指定了特定文件保存的目录才能够对我们的Java工程进行自动化构建

4.3 坐标

1. Maven的坐标（类似几何的坐标）

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

[2] artifactId：当前项目的模块名称

[3] version：当前模块的版本

[4] 如：

```
<groupId>com.it</groupId>
<artifactId>Demo1</artifactId>
<version>1.0-SNAPSHOT</version>
```

2. 如何通过坐标到仓库中查找jar包

[1] 将gav三个向量连起来

```
com.ujiuye.maven+Demo1+1.0-SNAPSHOT
```

[2] 以连起来的字符串作为目录结构到仓库中查找

```
com/ujiuye/maven/Demo1/0.0.1-SNAPSHOT/Demo1-0.0.1-SNAPSHOT.jar
```

3. 注意

我们自己的Maven工程必须执行安装操作才会进入仓库。安装的命令是：mvn install

4.4 第二个Maven程序

1. 目录结构

```
Demo2
src
  --main
  --java
  --resources
  --test
  --java
  --resources
pom.xml
```

2. POM文件

```
<?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.ujiuye</groupId>
  <artifactId>Demo2</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.ujiuye.maven</groupId>
      <artifactId>Demo1</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

3. 主程序

在src/main/java/com/ujiuye/maven目录下新建文件Demo2.java

```
package com.ujiuye.maven;
import com.ujiuye.maven.Demo1;
public class Demo2 {
    public String sayHelloToFriend(String name){
        Demo1 demo1 = new Demo1();
```

```

        String str = demo1.sayHello(name)+" I am "+this.getMyName();
        System.out.println(str);
        return str;
    }
    public String getMyName(){
        return "John";
    }
}

```

4. 测试程序

在/src/test/java/com/ujiuye/maven目录下新建测试文件Demo2Test.java

```

package com.ujiuye.maven;
import static junit.framework.Assert.assertEquals;
import org.junit.Test;
import com.ujiuye.maven.Demo1;
public class Demo2Test {
    @Test
    public void testHelloFriend(){
        Demo2 demo2 = new Demo2();
        String results = demo2.sayHelloToFriend("ljw");
        assertEquals("Hello ljw! I am John",results);
    }
}

```

5. 关键：对Demo1的依赖

这里Demo1就是我们的第一个Maven工程，现在Demo2对它有依赖。那么这个依赖能否成功呢？更进一步的问题是：Demo2工程会到哪里去找Demo1呢？

答案是：本地仓库。任何一个Maven工程会根据坐标到本地仓库中去查找它所依赖的jar包。如果能够找到则可以正常工作，否则就不行。

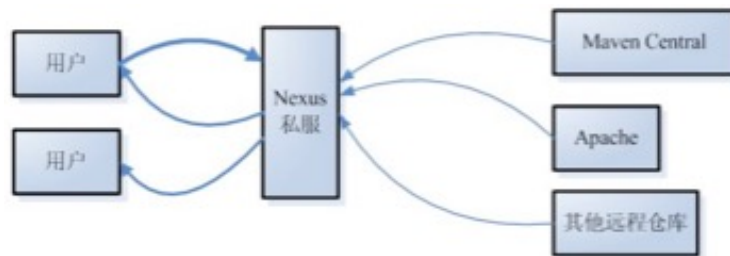
第5节：仓库

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

[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。

2. 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。

④ 手动将jar引入仓库

```
mvn install:install-file -Dfile=jar包的位置(参数一) -DgroupId=groupId(参数二) -DartifactId=artifactId(参数三) -Dversion=version(参数四) -Dpackaging=jar
```

第6节：生命周期

1. 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。

2. clean生命周期

Clean生命周期一共包含了段：

`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

3. 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。

4. 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。

5. 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。

第7节：插件和目标

1) 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。

2) 每个插件都能实现多个功能，每个功能就是一个插件目标。

3) 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。

第8节：依赖管理

1. 基本概念

当A jar包需要用到B jar包中的类时，我们就说A对B有依赖。例如：`commons-fileupload-1.3.jar`依赖于`commons-io-2.0.1.jar`。

通过第二个Maven工程我们已经看到，当前工程会到本地仓库中根据坐标查找它所依赖的jar包。

配置的基本形式是使用`dependency`标签指定目标jar包的坐标。

```
<dependencies>
  <dependency>
    <!--坐标 -->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <!-- 依赖的范围 -->
    <scope>test</scope>
  </dependency>
</dependencies>
```

2. 直接依赖和间接依赖

如果A依赖B，B依赖C，那么A→B和B→C都是直接依赖，而A→C是间接依赖。

3. 依赖的范围

① compile

[1] main目录下的Java代码可以访问这个范围的依赖

[2] test目录下的Java代码可以访问这个范围的依赖

[3] 部署到Tomcat服务器上运行时要放在WEB-INF的lib目录下

例如：对Demo1的依赖。主程序、测试程序和服务器运行时都需要用到。

② test

[1] main目录下的Java代码不能访问这个范围的依赖

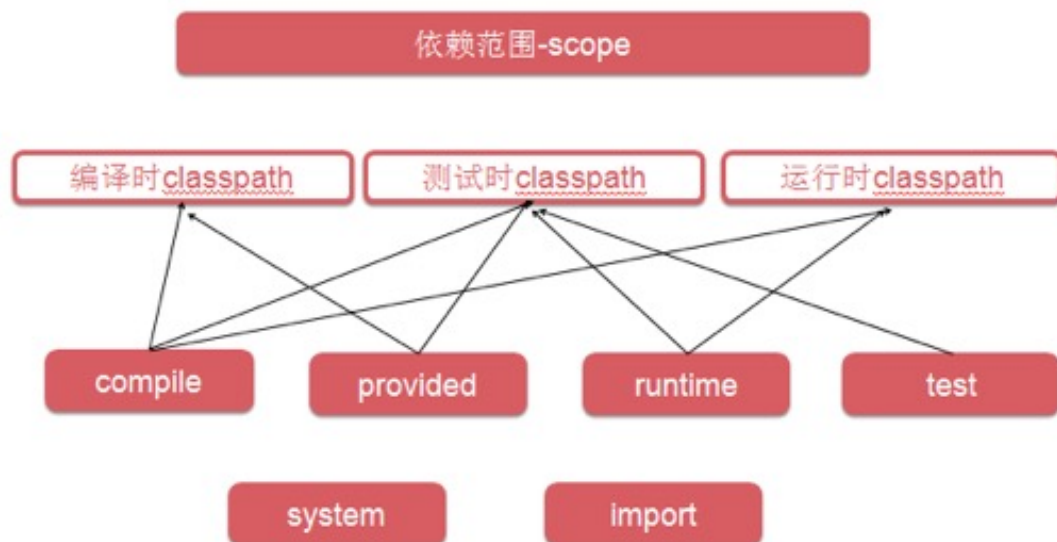
[2] test目录下的Java代码可以访问这个范围的依赖

[3] 部署到Tomcat服务器上运行时不会放在WEB-INF的lib目录下

例如：对junit的依赖。仅仅是测试程序部分需要。

③ provided

- [1] main目录下的Java代码可以访问这个范围的依赖
 - [2] test目录下的Java代码可以访问这个范围的依赖
 - [3] 部署到Tomcat服务器上运行时不会放在WEB-INF的lib目录下
- 例如：servlet-api在服务器上运行时，Servlet容器会提供相关API，所以部署的时候不需要。
- ④ 其他：runtime、import、system等。



8.1 依赖的传递性

当存在间接依赖的情况时，主工程对间接依赖的jar可以访问吗？这要看间接依赖的jar包引入时的依赖范围——只有依赖范围为compile时可以访问。

8.2 依赖的原则

- 1) 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。

8.3 依赖的排除

假设：

当前工程为public，直接依赖environment。environment依赖commons-logging的1.1.1对于public来说是间接依赖。当前工程public直接依赖commons-logging的1.1.2.加入exclusions配置后可以在依赖environment的时候排除版本为1.1.1的commons-logging的间接依赖。

```
<dependency>
  <groupId>com.ujiuye.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>

```

8.4 统一版本

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。

第9节：继承

9.1 继承机制

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。

9.2 创建父工程和子模块

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。在父工程中new - module [创建子模块](#) (Java工程或者JavaWeb工程)

9.3 在子工程中引用父工程

```
<parent>
  <!-- 父工程坐标 -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <relativePath>从当前目录到父项目的pom.xml文件的相对路径</relativePath>
</parent>
```

例如：

```
<parent>
  <artifactId>Parent</artifactId>
  <groupId>com.ujiuye.maven</groupId>
  <version>1.0-SNAPSHOT</version>
  <relativePath>../pom.xml</relativePath>
</parent>
```

注：此时如果子工程的groupId和version如果和父工程重复则可以删除。

9.4 在父工程中管理依赖

① 将Parent项目中的dependencies标签，用dependencyManagement标签括起来

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

② 在子项目中重新指定需要的依赖，删除范围和版本号

```
<dependencies>
  <dependency>
```

```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
</dependency>
</dependencies>
```

第10节：聚合

10.1 为什么使用聚合

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。

10.2 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>
  <module>son01</module>
</modules>
```

第11节：Maven酷站

- ① <https://mvnrepository.com/> 搜索需要的jar依赖的信息
- ② <https://search.maven.org/> 也可以