

第三章 使用 Maven：命令行环境

第一节 实验一：根据坐标创建 Maven 工程

1、Maven 核心概念：坐标

① 数学中的坐标

images

使用 x、y、z 三个『向量』作为空间的坐标系，可以在『空间』中唯一的定位到一个『点』。

② Maven 中的坐标

[1] 向量说明

使用三个『向量』在『Maven 的仓库』中唯一的定位到一个『jar』包。

- groupId：公司或组织的 id
- artifactId：一个项目或者是项目中的一个模块的 id
- version：版本号

[2] 三个向量的取值方式

- groupId：公司或组织域名的倒序，通常也会加上项目名称
 - 例如：com.atguigu.maven
- artifactId：模块的名称，将来作为 Maven 工程的工程名
- version：模块的版本号，根据自己的需要设定
 - 例如：SNAPSHOT 表示快照版本，正在迭代过程中，不稳定的版本
 - 例如：RELEASE 表示正式版本

举例：

- groupId：com.atguigu.maven
- artifactId：pro01-atguigu-maven
- version：1.0-SNAPSHOT

③ 坐标和仓库中 jar 包的存储路径之间的对应关系

坐标：

```
<groupId>javax.servlet</groupId>  
<artifactId>servlet-api</artifactId>  
<version>2.5</version>
```

上面坐标对应的 jar 包在 Maven 本地仓库中的位置：

```
Maven本地仓库根目录\javax\servlet\servlet-api\2.5\servlet-api-2.5.jar
```

一定要学会根据坐标到本地仓库中找到对应的 jar 包。

2、实验操作

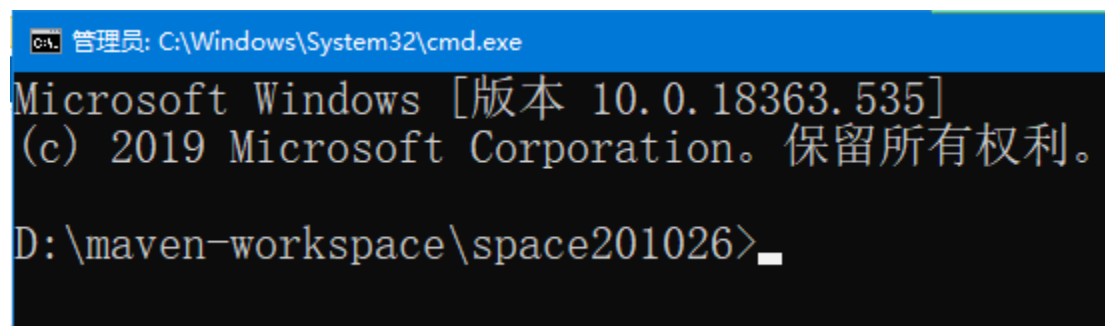
① 创建目录作为后面操作的工作空间

例如：D:\maven-workspace\space201026

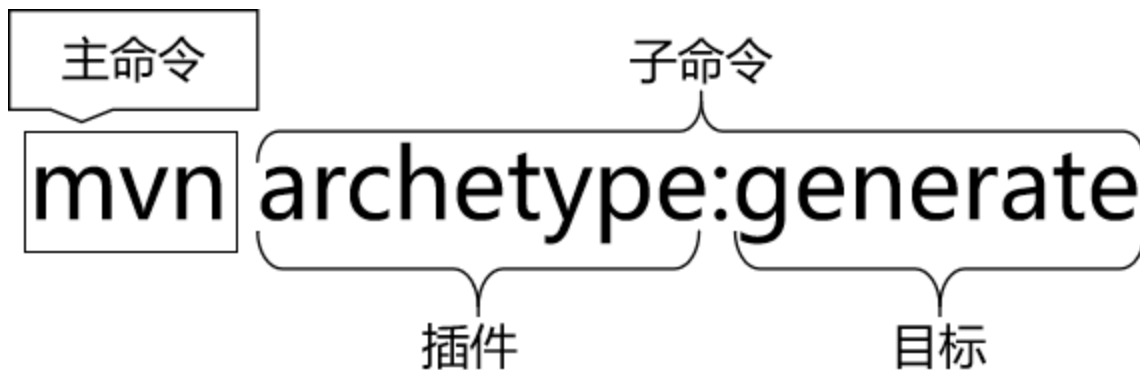
此时我们已经有了三个目录，分别是：

- Maven 核心程序：中军大帐
- Maven 本地仓库：兵营
- 本地工作空间：战场

② 在工作空间目录下打开命令行窗口



③ 使用命令生成 Maven 工程



运行 `mvn archetype:generate` 命令

下面根据提示操作

Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 7: 【直接回车，使用默认值】

Define value for property 'groupId' : com.atguigu.maven

Define value for property 'artifactId' : pro01-maven-java

Define value for property 'version' 1.0-SNAPSHOT: : 【直接回车，使用默认值】

Define value for property 'package' com.atguigu.maven: : 【直接回车，使用默认值】

Confirm properties configuration: groupId: com.atguigu.maven artifactId: pro01-maven-java version: 1.0-SNAPSHOT package: com.atguigu.maven Y: : 【直接回车，表示确认。如果前面有输入错误，想要重新输入，则输入 N 再回车。】

④ 调整

Maven 默认生成的工程，对 junit 依赖的是较低的 3.8.1 版本，我们可以改成较适合的 4.12 版本。

自动生成的 `App.java` 和 `AppTest.java` 可以删除。

```
<!-- 依赖信息配置 -->
<!-- dependencies复数标签: 里面包含dependency单数标签 -->
<dependencies>
  <!-- dependency单数标签: 配置一个具体的依赖 -->
  <dependency>
    <!-- 通过坐标来依赖其他jar包 -->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>

    <!-- 依赖的范围 -->
    <scope>test</scope>
  </dependency>
</dependencies>
```

⑤ 自动生成的 pom.xml 解读

```
<!-- 当前Maven工程的坐标 -->
<groupId>com.atguigu.maven</groupId>
<artifactId>pro01-maven-java</artifactId>
<version>1.0-SNAPSHOT</version>

<!-- 当前Maven工程的打包方式，可选值有下面三种： -->
<!-- jar：表示这个工程是一个Java工程 -->
<!-- war：表示这个工程是一个Web工程 -->
<!-- pom：表示这个工程是“管理其他工程”的工程 -->
<packaging>jar</packaging>

<name>pro01-maven-java</name>
<url>http://maven.apache.org</url>

<properties>
  <!-- 工程构建过程中读取源码时使用的字符集 -->
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<!-- 当前工程所依赖的jar包 -->
<dependencies>
  <!-- 使用dependency配置一个具体的依赖 -->
  <dependency>

    <!-- 在dependency标签内使用具体的坐标依赖我们需要的一个jar包 -->
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>

    <!-- scope标签配置依赖的范围 -->
    <scope>test</scope>
  </dependency>
</dependencies>
```

3、Maven 核心概念：POM

① 含义

POM: Project Object Model, 项目对象模型, 和 POM 类似的是: DOM (Document Object Model), 文档对象模型。它们都是模型化思想的具体体现。

② 模型化思想

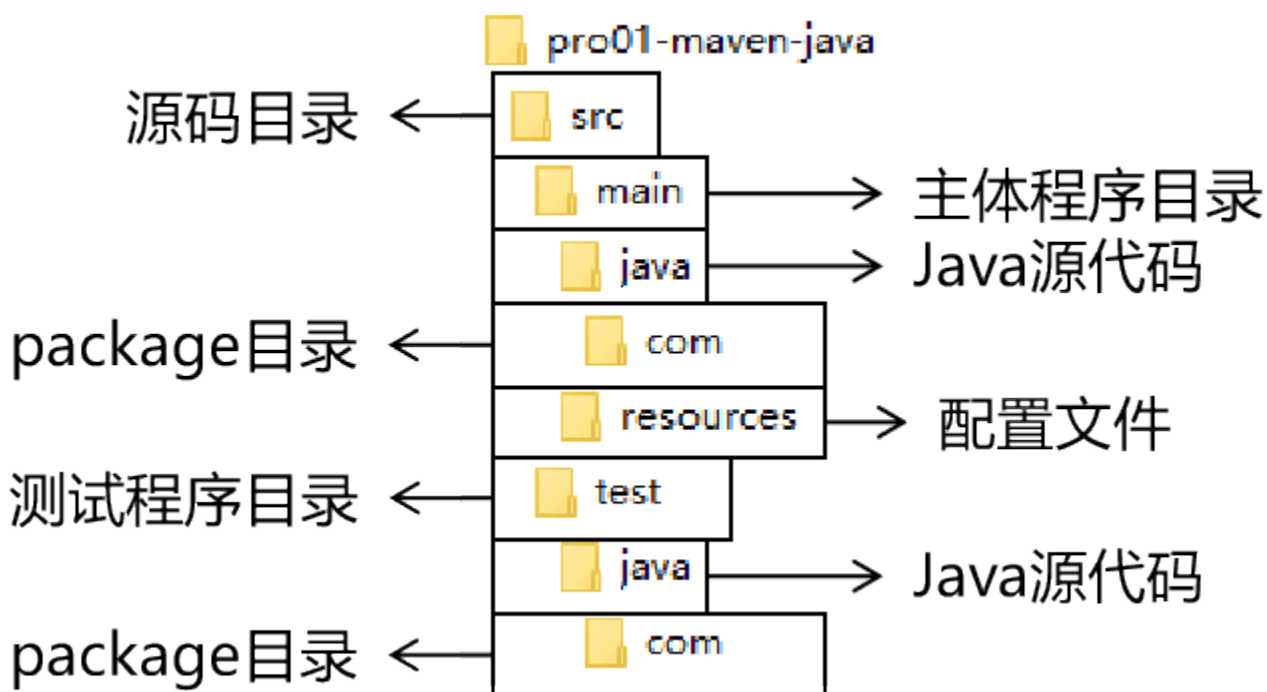
POM 表示将工程抽象为一个模型，再用程序中的对象来描述这个模型。这样我们就可以用程序来管理项目了。我们在开发过程中，最基本的做法就是将现实生活中的事物抽象为模型，然后封装模型相关的数据作为一个对象，这样就可以在程序中计算与现实事物相关的数据。

③ 对应的配置文件

POM 理念集中体现在 Maven 工程根目录下 pom.xml 这个配置文件中。所以这个 pom.xml 配置文件就是 Maven 工程的核心配置文件。其实学习 Maven 就是学这个文件怎么配置，各个配置有什么用。

4、Maven 核心概念：约定的目录结构

① 各个目录的作用



另外还有一个 `target` 目录专门存放构建操作输出的结果。

② 约定目录结构的意义

Maven 为了让构建过程能够尽可能自动化完成，所以必须约定目录结构的作用。例如：Maven 执行编译操作，必须先去 Java 源程序目录读取 Java 源代码，然后执行编译，最后把编译结果存放在 `target` 目录。

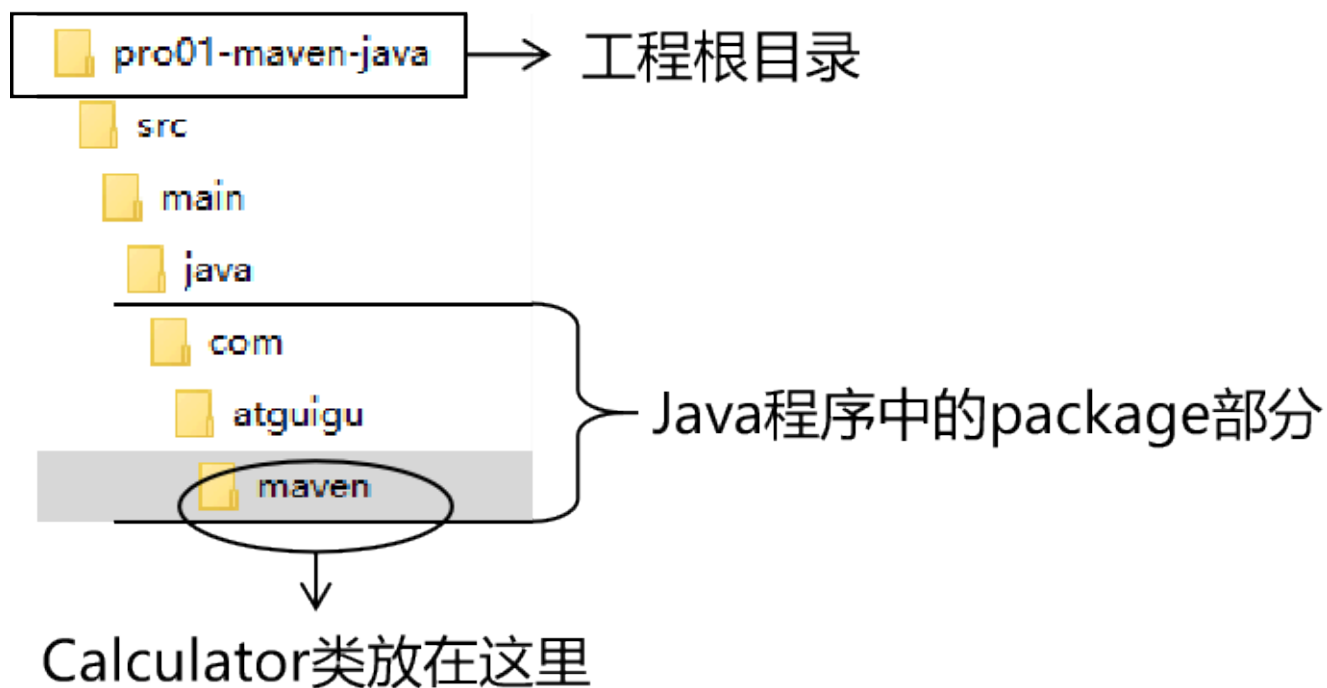
③ 约定大于配置

Maven 对于目录结构这个问题，没有采用配置的方式，而是基于约定。这样会让我们在开发过程中非常方便。如果每次创建 Maven 工程后，还需要针对各个目录的位置进行详细的配置，那肯定非常麻烦。

目前开发领域的技术发展趋势就是：约定大于配置，配置大于编码。

第二节 实验二：在 Maven 工程中编写代码

1、主体程序

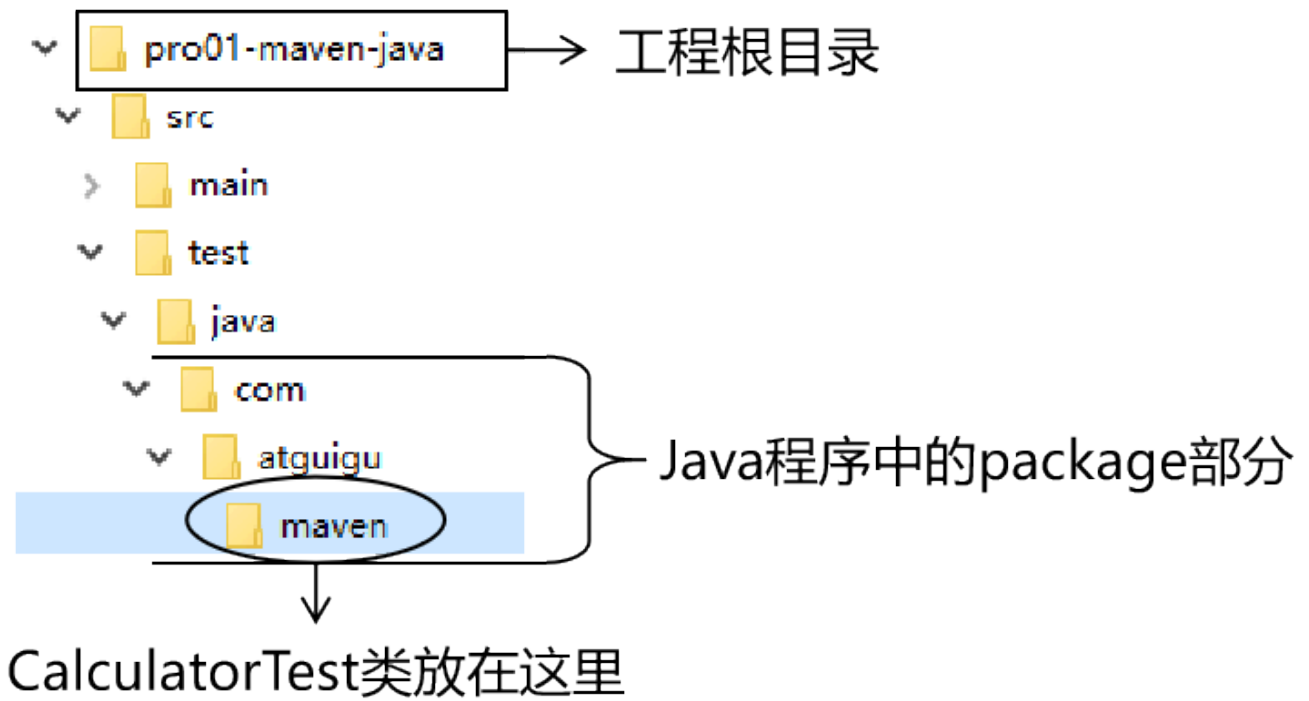


主体程序指的是被测试的程序，同时也是将来在项目中真正要使用的程序。

```
package com.atguigu.maven;

public class Calculator {
    public int sum(int i, int j){
        return i + j;
    }
}
```

2、测试程序




```

package com.atguigu.maven;

import org.junit.Test;
import com.atguigu.maven.Calculator;

// 静态导入的效果是将Assert类中的静态资源导入当前类
// 这样一来，在当前类中就可以直接使用Assert类中的静态资源，不需要写类名
import static org.junit.Assert.*;

public class CalculatorTest{
    @Test
    public void testSum(){
        // 1.创建Calculator对象
        Calculator calculator = new Calculator();

        // 2.调用Calculator对象的方法，获取到程序运行实际的结果
        int actualResult = calculator.sum(5, 3);

        // 3.声明一个变量，表示程序运行期待的结果
        int expectedResult = 8;

        // 4.使用断言来判断实际结果和期待结果是否一致
        // 如果一致：测试通过，不会抛出异常
        // 如果不一致：抛出异常，测试失败
        assertEquals(expectedResult, actualResult);
    }
}

```

第三节 实验三：执行 Maven 的构建命令

1、要求

运行 Maven 中和构建操作相关的命令时，必须进入到 pom.xml 所在的目录。如果没有在 pom.xml 所在的目录运行 Maven 的构建命令，那么会看到下面的错误信息：

```
The goal you specified requires a project to execute but there is no POM in this directory
```

mvn -v 命令和构建操作无关，只要正确配置了 PATH，在任何目录下执行都可以。而构建相关的命令要在 pom.xml 所在目录下运行——操作哪个工程，就进入这个工程的 pom.xml 目录。

2、清理操作

`mvn clean`

效果：删除 target 目录

3、编译操作

主程序编译：`mvn compile`

测试程序编译：`mvn test-compile`

主体程序编译结果存放的目录：`target/classes`

测试程序编译结果存放的目录：`target/test-classes`

4、测试操作

`mvn test`

测试的报告存放的目录：`target/surefire-reports`

5、打包操作

`mvn package`

打包的结果——jar 包，存放的目录：`target`

6、安装操作

`mvn install`

```
[INFO] Installing D:\maven-workspace\space201026\pro01-maven-java\target\pro01-maven-java-1.0-SNAPSHOT.jar to D:\maven-repository\space201026\pro01-maven-java\pro01-maven-java-1.0-SNAPSHOT.jar
[INFO] Installing D:\maven-workspace\space201026\pro01-maven-java\pom.xml to D:\maven-repository\space201026\pro01-maven-java\pro01-maven-java-1.0-SNAPSHOT.pom
```

安装的效果是将本地构建过程中生成的 jar 包存入 Maven 本地仓库。这个 jar 包在 Maven 仓库中的路径是根据它的坐标生成的。

坐标信息如下：

```
<groupId>com.atguigu.maven</groupId>
<artifactId>pro01-maven-java</artifactId>
<version>1.0-SNAPSHOT</version>
```

在 Maven 仓库中生成的路径如下：

```
D:\maven-rep1026\com\atguigu\maven\pro01-maven-java\1.0-SNAPSHOT\pro01-maven-java-1.0-SNAPSHOT.jar
```

另外，安装操作还会将 pom.xml 文件转换为 XXX.pom 文件一起存入本地仓库。所以我们在 Maven 的本地仓库中想看一个 jar 包原始的 pom.xml 文件时，查看对应 XXX.pom 文件即可，它们是名字发生了改变，本质上是同一个文件。

第四节 实验四：创建 Maven 版的 Web 工程

1、说明

使用 mvn archetype:generate 命令生成 Web 工程时，需要使用一个专门的 archetype。这个专门生成 Web 工程骨架的 archetype 可以参照官网看到它的用法：

Maven Webapp Archetype

maven-archetype-webapp is an archetype which generates a sample Maven webapp project:

```
1. project
2. |-- pom.xml
3. |-- src
4. |   |-- main
5. |   |   |-- webapp
6. |   |   |   |-- WEB-INF
7. |   |   |   |   |-- web.xml
8. |   |   |   |   |-- index.jsp
```

Usage

To generate a new project from this archetype, type:

```
1. mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.4
```

参数 archetypeGroupId、archetypeArtifactId、archetypeVersion 用来指定现在使用的 maven-archetype-webapp 的坐标。

2、操作

注意：如果在上一个工程的目录下执行 mvn archetype:generate 命令，那么 Maven 会报错：不能在一个非 pom 的工程下再创建其他工程。所以不要再刚才创建的工程里再创建新的工

程，请回到工作空间根目录来操作。

然后运行生成工程的命令：

```
mvn archetype:generate -D archetypeGroupId=org.apache.maven.archetypes -D archetypeArtifactId=maven-archetype-webapp
```

下面的操作按照提示执行：

Define value for property 'groupId' : com.atguigu.maven Define value for property 'artifactId' : pro02-maven-web Define value for property 'version' 1.0-SNAPSHOT: : 【直接回车，使用默认值】

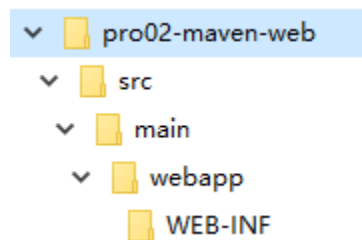
Define value for property 'package' com.atguigu.maven: : 【直接回车，使用默认值】 Confirm properties configuration: groupId: com.atguigu.maven artifactId: pro02-maven-web version: 1.0-SNAPSHOT package: com.atguigu.maven Y: : 【直接回车，表示确认】

3、生成的 pom.xml

确认打包的方式是 war 包形式

```
<packaging>war</packaging>
```

4、生成的 Web 工程的目录结构

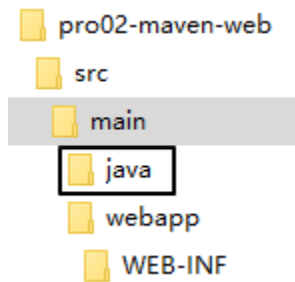


webapp 目录下有 index.jsp

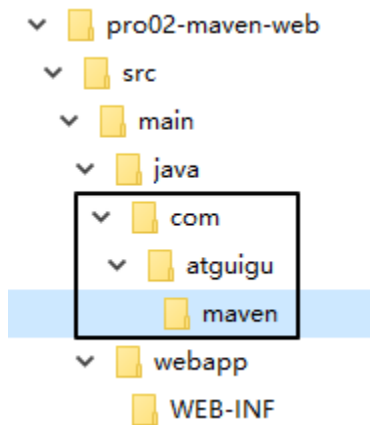
WEB-INF 目录下有 web.xml

5、创建 Servlet

① 在 main 目录下创建 java 目录



② 在 java 目录下创建 Servlet 类所在的包的目录



③ 在包下创建 Servlet 类

```
package com.atguigu.maven;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;

public class HelloServlet extends HttpServlet{
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().write("hello maven web");
    }
}
```

④ 在 web.xml 中注册 Servlet

```
<servlet>
  <servlet-name>helloServlet</servlet-name>
  <servlet-class>com.atguigu.maven.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>helloServlet</servlet-name>
  <url-pattern>/helloServlet</url-pattern>
</servlet-mapping>
```

6、在 index.jsp 页面编写超链接

```
<html>
<body>
<h2>Hello World!</h2>
<a href="helloServlet">Access Servlet</a>
</body>
</html>
```

JSP 全称是 Java Server Page，和 Thymeleaf 一样，是服务器端页面渲染技术。这里我们不必关心 JSP 语法细节，编写一个超链接标签即可。

7、编译

此时直接执行 mvn compile 命令出错：

```
DANGER

程序包 javax.servlet.http 不存在

程序包 javax.servlet 不存在

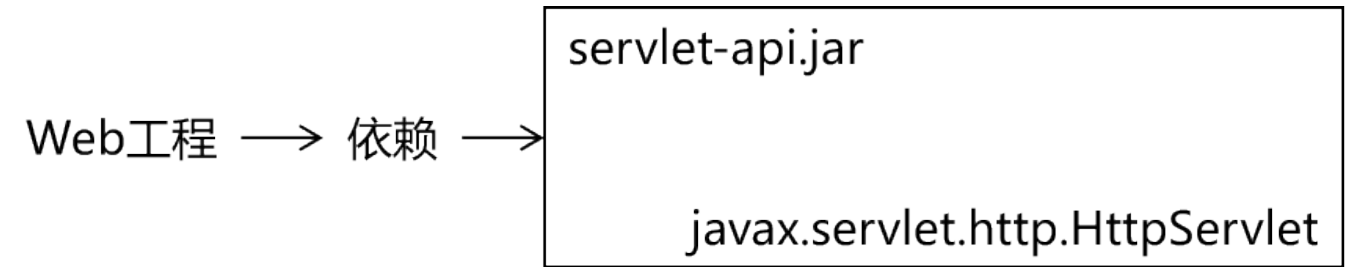
找不到符号

符号: 类 HttpServlet

.....
```

上面的错误信息说明：我们的 Web 工程用到了 HttpServlet 这个类，而 HttpServlet 这个类属

于 servlet-api.jar 这个 jar 包。此时我们说，Web 工程需要依赖 servlet-api.jar 包。




8、配置对 servlet-api.jar 包的依赖

对于不知道详细信息的依赖可以到<https://mvnrepository.com/>网站查询。使用关键词搜索，然后在搜索结果列表中选择适合的使用。

Found 35709 results

Sort: [relevance](#) | [popular](#) | [newest](#)



1. **Java Servlet API**


javax.servlet » javax.servlet-api

Java Servlet API

Last Release on Apr 20, 2018

13,968 usages

[GPL](#) [CDDL](#)



2. **JavaServlet(TM) Specification**


javax.servlet » servlet-api

JavaServlet(TM) Specification

Last Release on Apr 17, 2008

11,124 usages

[GPL](#) [GPL](#) [CDDL](#)



3. **Servlet API**


org.mortbay.jetty » servlet-api

Servlet API

Last Release on Feb 24, 2010

283 usages

[EPL](#) [Apache](#)



4. **Tomcat Servlet API**

org.apache.tomcat » tomcat-servlet-api

jakarta.servlet package

263 usages

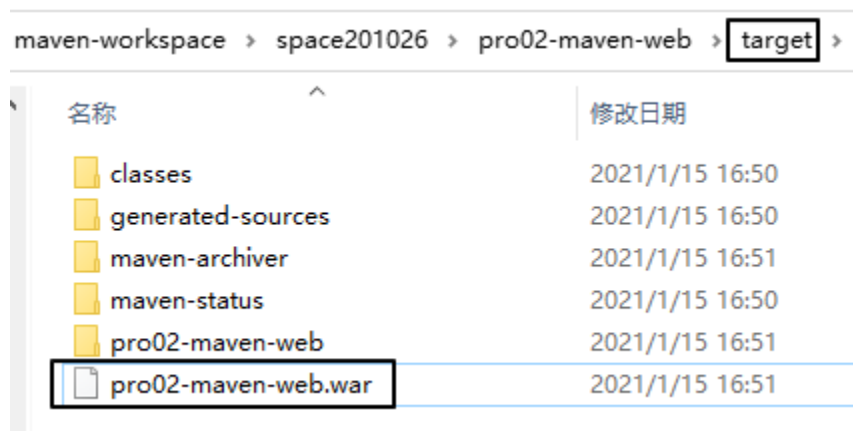
比如，我们找到的 servlet-api 的依赖信息：

```
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

这样就可以把上面的信息加入 pom.xml。重新执行 mvn compile 命令。

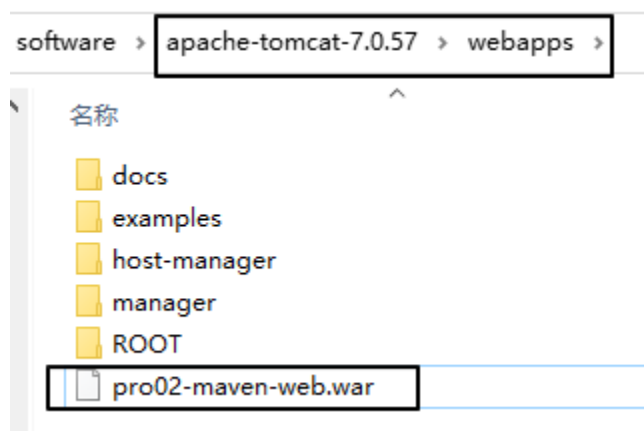
9、将 Web 工程打包为 war 包

运行 mvn package 命令，生成 war 包的位置如下图所示：

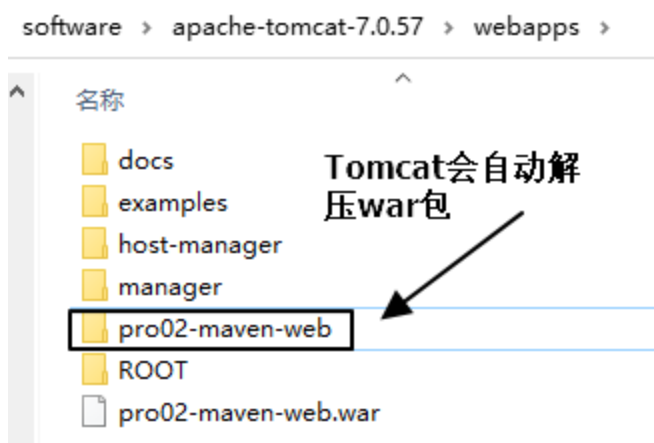
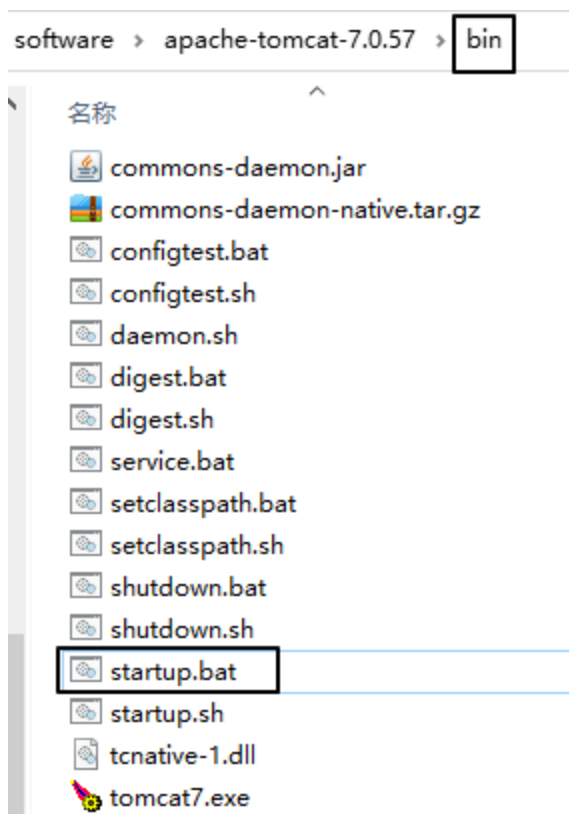


10、将 war 包部署到 Tomcat 上运行

将 war 包复制到 Tomcat/webapps 目录下



启动 Tomcat：



通过浏览器尝试访问：<http://localhost:8080/pro02-maven-web/index.jsp>

第五节 实验五：让 Web 工程依赖 Java 工程

1、观念

明确一个意识：从来只有 Web 工程依赖 Java 工程，没有反过来 Java 工程依赖 Web 工程。本质上来说，Web 工程依赖的 Java 工程其实就是 Web 工程里导入的 jar 包。最终 Java 工程会变成 jar 包，放在 Web 工程的 WEB-INF/lib 目录下。

2、操作

在 pro02-maven-web 工程的 pom.xml 中，找到 dependencies 标签，在 dependencies 标签中做如下配置：

```
<!-- 配置对Java工程pro01-maven-java的依赖 -->
<!-- 具体的配置方式：在dependency标签内使用坐标实现依赖 -->
<dependency>
    <groupId>com.atguigu.maven</groupId>
    <artifactId>pro01-maven-java</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

3、在 Web 工程中，编写测试代码

① 补充创建目录

pro02-maven-web\src\test\java\com\atguigu\maven

② 确认 Web 工程依赖了 junit

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

③ 创建测试类

把 Java 工程的 CalculatorTest.java 类复制到 pro02-maven-web\src\test\java\com\atguigu\maven 目录下

4、执行 Maven 命令

① 测试命令

mvn test

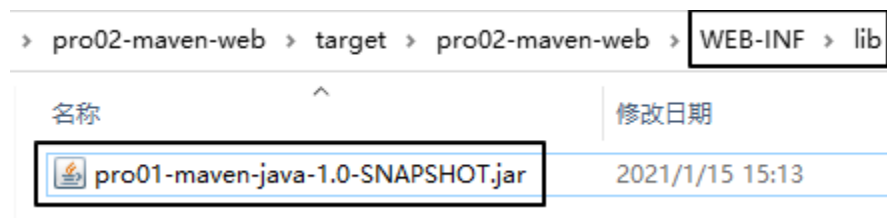
说明：测试操作中会提前自动执行编译操作，测试成功就说明编译也是成功的。

② 打包命令

mvn package



通过查看 war 包内的结构，我们看到被 Web 工程依赖的 Java 工程确实是会变成 Web 工程的 WEB-INF/lib 目录下的 jar 包。



③ 查看当前 Web 工程所依赖的 jar 包的列表

mvn dependency:list

```
[INFO] The following files have been resolved:
[INFO] org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] javax.servlet:javax.servlet-api:jar:3.1.0:provided
[INFO] com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
[INFO] junit:junit:jar:4.12:test
```

说明：javax.servlet:javax.servlet-api:jar:3.1.0:provided 格式显示的是一个 jar 包的坐标信息。格式是：

groupId:artifactId:打包方式:version:依赖的范围

这样的格式虽然和我们 XML 配置文件中坐标的格式不同，但是本质上还是坐标信息，大家需要能够认识这样的格式，将来从 Maven 命令的日志或错误信息中看到这样格式的信息，就能够识别出来这是坐标。进而根据坐标到 Maven 仓库找到对应的 jar 包，用这样的方式解决我们遇到的报错的情况。

④ 以树形结构查看当前 Web 工程的依赖信息

mvn dependency:tree

```
[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
[INFO] \- com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
```

我们在 pom.xml 中并没有依赖 hamcrest-core，但是它却被加入了我们依赖的列表。原因是：junit 依赖了 hamcrest-core，然后基于依赖的传递性，hamcrest-core 被传递到我们的工程了。

第六节 实验六：测试依赖的范围

1、依赖范围

标签的位置：dependencies/dependency/**scope**

标签的可选值：**compile/test/provided/system/runtime/import**

① compile 和 test 对比

	main 目录 (空间)	test 目录 (空间)	开发过程 (时间)	部署到服务器 (时间)
compile	有效	有效	有效	有效
test	无效	有效	有效	无效

② compile 和 provided 对比

	main 目录 (空间)	test 目录 (空间)	开发过程 (时间)	部署到服务器 (时间)
compile	有效	有效	有效	有效
provided	有效	有效	有效	无效

③ 结论

compile: 通常使用的第三方框架的 jar 包这样在项目实际运行时真正要用到的 jar 包都是以 compile 范围进行依赖的。比如 SSM 框架所需 jar 包。

test: 测试过程中使用的 jar 包, 以 test 范围依赖进来。比如 junit。

provided: 在开发过程中需要用到的“服务器上的 jar 包”通常以 provided 范围依赖进来。比如 servlet-api、jsp-api。而这个范围的 jar 包之所以不参与部署、不放进 war 包, 就是避免和服务服务器上已有的同类 jar 包产生冲突, 同时减轻服务器的负担。说白了就是: **“服务器上已经有了, 你就别带啦!”**

2、测试

① 验证 compile 范围对 main 目录有效

main 目录下的类: HelloServlet 使用 compile 范围导入的依赖: pro01-atguigu-maven

验证: 使用 compile 范围导入的依赖对 main 目录下的类来说是有效的

有效: HelloServlet 能够使用 pro01-atguigu-maven 工程中的 Calculator 类

验证方式: 在 HelloServlet 类中导入 Calculator 类, 然后编译就说明有效。

② 验证 test 范围对 main 目录无效

测试方式: 在主体程序中导入 org.junit.Test 这个注解, 然后执行编译。

具体操作: 在 pro01-maven-java\src\main\java\com\atguigu\maven 目录下修改 Calculator.java

```

package com.atguigu.maven;

import org.junit.Test;

public class Calculator {

    public int sum(int i, int j){
        return i + j;
    }
}

```

执行 Maven 编译命令：

```
[ERROR] /D:/maven-workspace/space201026/pro01-maven-java/src/main/java/com/atguigu/maven/Calculator
```

③ 验证 test 和 provided 范围不参与服务器部署

其实就是验证：通过 compile 范围依赖的 jar 包会放入 war 包，通过 test 范围依赖的 jar 包不会放入 war 包。

```

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope> → 没有放进war包
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet-api/javax.servlet-api -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope> → 没有放进war包
</dependency>
<!-- 配置对Java工程pro01-maven-java的依赖 -->
<!-- 具体的配置方式：在dependency标签内使用坐标 -->
<dependency>
  <groupId>com.atguigu.maven</groupId>
  <artifactId>pro01-maven-java</artifactId>
  <version>1.0-SNAPSHOT</version>
  <scope>compile</scope> → 放进war包
</dependency>

```



④ 验证 provided 范围对测试程序有效

测试方式是在 pro02-maven-web 的测试程序中加入 servlet-api.jar 包中的类。

修改：pro02-maven-web\src\test\java\com\atguigu\maven\CalculatorTest.java

```

package com.atguigu.maven;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

import org.junit.Test;
import com.atguigu.maven.Calculator;

// 静态导入的效果是将Assert类中的静态资源导入当前类
// 这样一来，在当前类中就可以直接使用Assert类中的静态资源，不需要写类名
import static org.junit.Assert.*;

public class CalculatorTest{

    @Test
    public void testSum(){

        // 1.创建Calculator对象
        Calculator calculator = new Calculator();

        // 2.调用Calculator对象的方法，获取到程序运行实际的结果
        int actualResult = calculator.sum(5, 3);

        // 3.声明一个变量，表示程序运行期待的结果
        int expectedResult = 8;

        // 4.使用断言来判断实际结果和期待结果是否一致
        // 如果一致：测试通过，不会抛出异常
        // 如果不一致：抛出异常，测试失败
        assertEquals(expectedResult, actualResult);
    }
}

```

然后运行 Maven 的编译命令：mvn compile

然后看到编译成功。

第七节 实验七：测试依赖的传递性

1、依赖的传递性

① 概念

A 依赖 B，B 依赖 C，那么在 A 没有配置对 C 的依赖的情况下，A 里面能不能直接使用 C？

② 传递的原则

在 A 依赖 B，B 依赖 C 的前提下，C 是否能够传递到 A，取决于 B 依赖 C 时使用的依赖范围。

- B 依赖 C 时使用 compile 范围：可以传递
- B 依赖 C 时使用 test 或 provided 范围：不能传递，所以需要这样的 jar 包时，就必须在需要的地方明确配置依赖才可以。

2、使用 compile 范围依赖 spring-core

测试方式：让 pro01-maven-java 工程依赖 spring-core

具体操作：编辑 pro01-maven-java 工程根目录下 pom.xml

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
```

使用 mvn dependency:tree 命令查看效果：

```
[INFO] com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO]    \- commons-logging:commons-logging:jar:1.1.1:compile
```

还可以在 Web 工程中，使用 mvn dependency:tree 命令查看效果（需要重新将 pro01-maven-java 安装到仓库）：


```
[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
[INFO] \- com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
[INFO] \- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO] \- commons-logging:commons-logging:jar:1.1.1:compile
```

3、验证 test 和 provided 范围不能传递

从上面的例子已经能够看到，pro01-maven-java 依赖了 junit，但是在 pro02-maven-web 工程中查看依赖树的时候并没有看到 junit。

要验证 provided 范围不能传递，可以在 pro01-maven-java 工程中加入 servlet-api 的依赖。

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
```

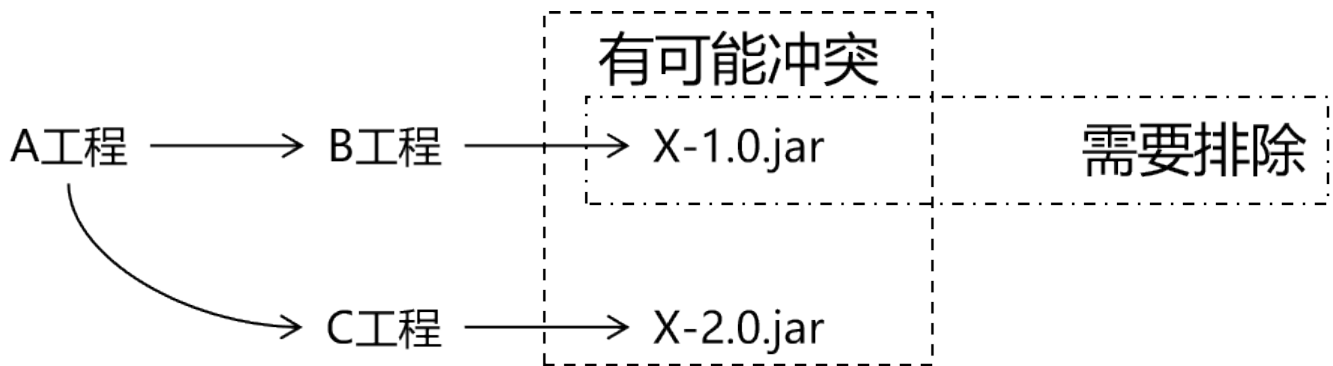
效果还是和之前一样：

```
[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
[INFO] \- com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
[INFO] \- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO] \- commons-logging:commons-logging:jar:1.1.1:compile
```

第八节 实验八：测试依赖的排除

1、概念

当 A 依赖 B，B 依赖 C 而且 C 可以传递到 A 的时候，A 不想要 C，需要在 A 里面把 C 排除掉。而往往这种情况都是为了避免 jar 包之间的冲突。



所以配置依赖的排除其实就是阻止某些 jar 包的传递。因为这样的 jar 包传递过来会和其他 jar 包冲突。

2、配置方式

```
<dependency>
  <groupId>com.atguigu.maven</groupId>
  <artifactId>pro01-maven-java</artifactId>
  <version>1.0-SNAPSHOT</version>
  <scope>compile</scope>
  <!-- 使用excludes标签配置依赖的排除 -->
  <exclusions>
    <!-- 在exclude标签中配置一个具体的排除 -->
    <exclusion>
      <!-- 指定要排除的依赖的坐标（不需要写version） -->
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

3、测试

测试的方式：在 pro02-maven-web 工程中配置对 commons-logging 的排除

```
<dependency>
  <groupId>com.atguigu.maven</groupId>
  <artifactId>pro01-maven-java</artifactId>
  <version>1.0-SNAPSHOT</version>
  <scope>compile</scope>
  <!-- 使用excludes标签配置依赖的排除 -->
  <exclusions>
    <!-- 在exclude标签中配置一个具体的排除 -->
    <exclusion>
      <!-- 指定要排除的依赖的坐标（不需要写version） -->
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

运行 mvn dependency:tree 命令查看效果：

```
[INFO] com.atguigu.maven:pro02-maven-web:war:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] | \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- javax.servlet:javax.servlet-api:jar:3.1.0:provided
[INFO] \- com.atguigu.maven:pro01-maven-java:jar:1.0-SNAPSHOT:compile
[INFO] \- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
```

发现在 spring-core 下面就没有 commons-logging 了。

第九节 实验九：继承

1、概念

Maven 工程之间，A 工程继承 B 工程

- B 工程：父工程
- A 工程：子工程

本质上是 A 工程的 pom.xml 中的配置继承了 B 工程中 pom.xml 的配置。

2、作用

在父工程中统一管理项目中的依赖信息，具体来说是管理依赖信息的版本。

它的背景是：

- 对一个比较大型的项目进行了模块拆分。
- 一个 project 下面，创建了很多个 module。
- 每一个 module 都需要配置自己的依赖信息。

它背后的需求是：

- 在每一个 module 中各自维护各自的依赖信息很容易发生出入，不易统一管理。
- 使用同一个框架内的不同 jar 包，它们应该是同一个版本，所以整个项目中使用的框架版本需要统一。
- 使用框架时所需要的 jar 包组合（或者说依赖信息组合）需要经过长期摸索和反复调试，最终确定一个可用组合。这个耗费很大精力总结出来的方案不应该在新的项目中重新摸索。

通过在父工程中为整个项目维护依赖信息的组合既**保证了整个项目使用规范、准确的 jar 包**；又能够将**以往的经验沉淀**下来，节约时间和精力。

3、举例

在一个工程中依赖多个 Spring 的 jar 包

```
[INFO] +- org.springframework:spring-core:jar:4.0.0.RELEASE:compile
[INFO] | \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.springframework:spring-beans:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-context:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-expression:jar:4.0.0.RELEASE:compile
[INFO] +- org.springframework:spring-aop:jar:4.0.0.RELEASE:compile
[INFO] | \- aopalliance:aopalliance:jar:1.0:compile
```

使用 Spring 时要求所有 Spring 自己的 jar 包版本必须一致。为了能够对这些 jar 包的版本进行统一管理，我们使用继承这个机制，将所有版本信息统一在父工程中进行管理。

4、操作

① 创建父工程

创建的过程和前面创建 pro01-maven-java 一样。

工程名称：pro03-maven-parent

工程创建好之后，要修改它的打包方式：

```
<groupId>com.atguigu.maven</groupId>
<artifactId>pro03-maven-parent</artifactId>
<version>1.0-SNAPSHOT</version>

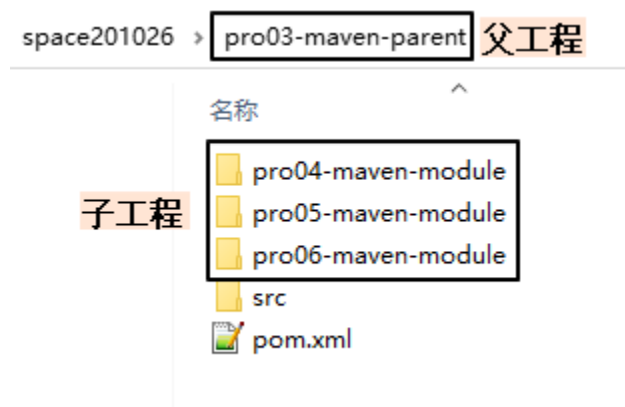
<!-- 当前工程作为父工程，它要去管理子工程，所以打包方式必须是 pom -->
<packaging>pom</packaging>
```

只有打包方式为 pom 的 Maven 工程能够管理其他 Maven 工程。打包方式为 pom 的 Maven 工程中不写业务代码，它是专门管理其他 Maven 工程的工程。

② 创建模块工程

模块工程类似于 IDEA 中的 module，所以需要**进入 pro03-maven-parent 工程的根目录**，然后运行 mvn archetype:generate 命令来创建模块工程。

假设，我们创建三个模块工程：



③ 查看被添加新内容的父工程 pom.xml

下面 modules 和 module 标签是聚合功能的配置

```
<modules>
  <module>pro04-maven-module</module>
  <module>pro05-maven-module</module>
  <module>pro06-maven-module</module>
</modules>
```

④ 解读子工程的 pom.xml

```
<!-- 使用parent标签指定当前工程的父工程 -->
<parent>
  <!-- 父工程的坐标 -->
  <groupId>com.atguigu.maven</groupId>
  <artifactId>pro03-maven-parent</artifactId>
  <version>1.0-SNAPSHOT</version>
</parent>

<!-- 子工程的坐标 -->
<!-- 如果子工程坐标中的groupId和version与父工程一致，那么可以省略 -->
<!-- <groupId>com.atguigu.maven</groupId> -->
<artifactId>pro04-maven-module</artifactId>
<!-- <version>1.0-SNAPSHOT</version> -->
```

⑤ 在父工程中配置依赖的统一管理

```
<!-- 使用dependencyManagement标签配置对依赖的管理 -->
<!-- 被管理的依赖并没有真正被引入到工程 -->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-core</artifactId>
      <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-beans</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-expression</artifactId>
      <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-aop</artifactId>
      <version>4.0.0.RELEASE</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

⑥ 子工程中引用那些被父工程管理的依赖

关键点：省略版本号

```

<!-- 子工程引用父工程中的依赖信息时，可以把版本号去掉。 -->
<!-- 把版本号去掉就表示子工程中这个依赖的版本由父工程决定。 -->
<!-- 具体来说是由父工程的dependencyManagement来决定。 -->
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-expression</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
  </dependency>
</dependencies>

```

⑦ 在父工程中升级依赖信息的版本

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>4.1.4.RELEASE</version>
</dependency>

```

然后在子工程中运行 `mvn dependency:list`，效果如下：

```

[INFO] org.springframework:spring-aop:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-core:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-context:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-beans:jar:4.1.4.RELEASE:compile
[INFO] org.springframework:spring-expression:jar:4.1.4.RELEASE:compile

```


⑧ 在父工程中声明自定义属性

```
<!-- 通过自定义属性，统一指定Spring的版本 -->
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

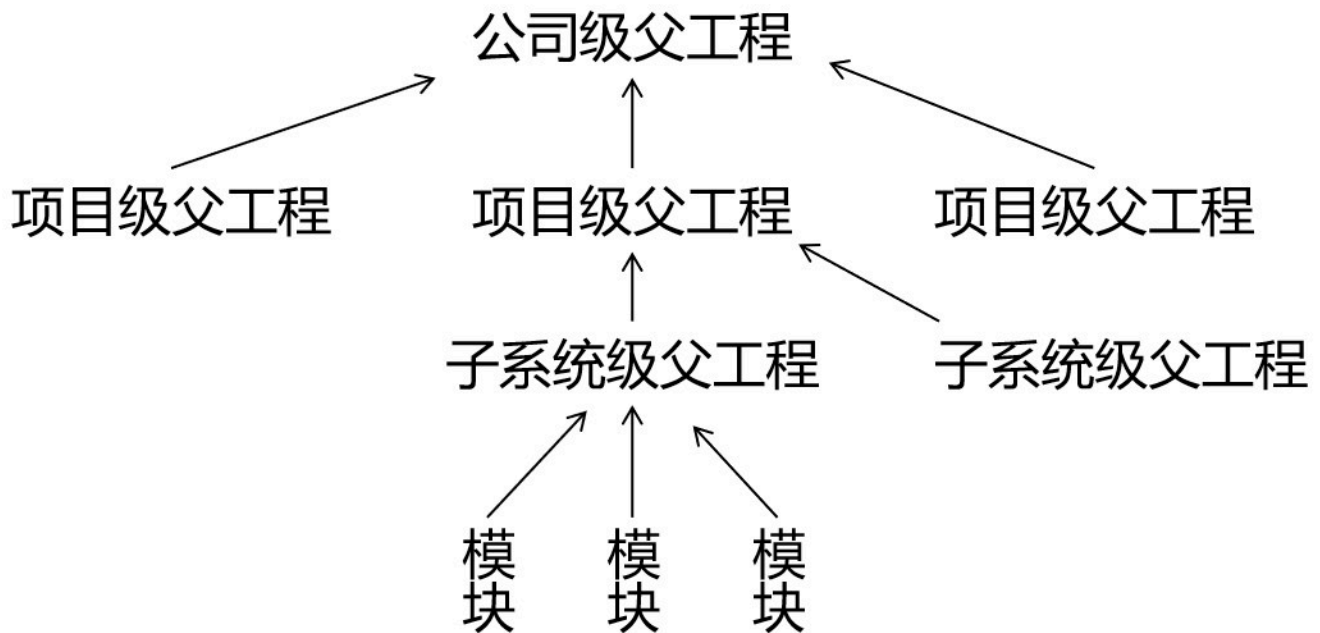
  <!-- 自定义标签，维护Spring版本数据 -->
  <atguigu.spring.version>4.3.6.RELEASE</atguigu.spring.version>
</properties>
```

在需要的地方使用\${}的形式来引用自定义的属性名：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${atguigu.spring.version}</version>
</dependency>
```

真正实现“一处修改，处处生效”。

5、实际意义



编写一套符合要求、开发各种功能都能正常工作的依赖组合并不容易。如果公司里已经有人总结了成熟的组合方案，那么再开发新项目时，如果不使用原有的积累，而是重新摸索，会浪费大量的时间。为了提高效率，我们可以使用工程继承的机制，让成熟的依赖组合方案能够保留下来。

如上图所示，公司级的父工程中管理的就是成熟的依赖组合方案，各个新项目、子系统各取所需即可。

第十节 实验十：聚合

1、聚合本身的含义

部分组成整体



动画片《战神金刚》中的经典台词：“我来组成头部！我来组成手臂！”就是聚合关系最生动的体现。

2、Maven 中的聚合

使用一个“总工程”将各个“模块工程”汇集起来，作为一个整体对应完整的项目。

- 项目：整体
- 模块：部分

概念的对应关系：

从继承关系角度来看：

- 父工程
- 子工程

从聚合关系角度来看：

- 总工程
- 模块工程

3、好处

- 一键执行 Maven 命令：很多构建命令都可以在“总工程”中一键执行。
以 mvn install 命令为例：Maven 要求有父工程时先安装父工程；有依赖的工程时，先安装被依赖的工程。我们自己考虑这些规则会很麻烦。但是工程聚合之后，在总工程执行 mvn install 可以一键完成安装，而且会自动按照正确的顺序执行。
- 配置聚合之后，各个模块工程会在总工程中展示一个列表，让项目中的各个模块一目了然。

4、聚合的配置

在总工程中配置 modules 即可：

```
<modules>
  <module>pro04-maven-module</module>
  <module>pro05-maven-module</module>
  <module>pro06-maven-module</module>
</modules>
```

5、依赖循环问题

如果 A 工程依赖 B 工程，B 工程依赖 C 工程，C 工程又反过来依赖 A 工程，那么在执行构建操作时会报下面的错误：

DANGER

```
[ERROR] [ERROR] The projects in the reactor contain a cyclic reference:
```

这个错误的含义是：循环引用。