简介

在了解Maven之前，我们先来看看一个Java项目需要的东西。首先，我们需要确定引入哪些依赖包。例如，如果我们需要用到commons logging，我们就必须把commons logging的jar包放入classpath。如果我们还需要log4j，就需要把log4j相关的jar包都放到classpath中。这些就是依赖包的管理。

其次，我们要确定项目的目录结构。例如，`src`目录存放Java源码，`resources`目录存放配置文件，`bin`目录存放编译生成的`.class`文件。

此外，我们还需要配置环境，例如JDK的版本，编译打包的流程，当前代码的版本号。

最后，除了使用Eclipse这样的IDE进行编译外，我们还必须能通过命令行工具进行编译，才能够让项目在一个独立的服务器上编译、测试、部署。

这些工作难度不大，但是非常琐碎且耗时。如果每一个项目都自己搞一套配置，肯定会一团糟。我们需要的是一个标准化的Java项目管理和构建工具。

Maven就是是专门为Java项目打造的管理和构建工具，它的主要功能有：

- 提供了一套标准化的项目结构；
- 提供了一套标准化的构建流程（编译，测试，打包，发布……）；
- 提供了一套依赖管理机制。

Maven项目结构

一个使用Maven管理的普通的Java项目，它的目录结构默认如下：

```
a-maven-project
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   └── test
│       ├── java
│       └── resources
└── target
```

所有的目录结构都是约定好的标准结构，我们千万不要随意修改目录结构。使用标准结构不需要做任何配置，Maven就可以正常使用。

pom.xml

我们再来看最关键的一个项目描述文件`pom.xml`，它的内容长得像下面：

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>hello</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
        <maven.compiler.release>17</maven.compiler.release>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.16</version>
        </dependency>
    </dependencies>
</project>
```

其中，`groupId`类似于Java的包名，通常是公司或组织名称，`artifactId`类似于Java的类名，通常是项目名称，再加上`version`，一个Maven工程就是由`groupId`，`artifactId`和`version`作为唯一标识。

我们在引用其他第三方库的时候，也是通过这3个变量确定。例如，依赖`org.slfj4:slf4j-simple:2.0.16`：

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>2.0.16</version>
</dependency>
```

使用`<dependency>`声明一个依赖后，Maven就会自动下载这个依赖包并把它放到classpath中。

另外，注意到`<properties>`定义了一些属性，常用的属性有：

- `project.build.sourceEncoding`：表示项目源码的字符编码，通常应设定为`UTF-8`；
- `maven.compiler.release`：表示使用的JDK版本，例如`21`；
- `maven.compiler.source`：表示Java编译器读取的源码版本；
- `maven.compiler.target`：表示Java编译器编译的Class版本。

从Java 9开始，推荐使用`maven.compiler.release`属性，保证编译时输入的源码和编译输出版本一致。如果源码和输出版本不同，则应该分别设置`maven.compiler.source`和`maven.compiler.target`。

通过`<properties>`定义的属性，就可以固定JDK版本，防止同一个项目的不同的开发者各自使用不同版本的JDK。

常用项目构建指令

```
mvn compile   编译
mvn clean     清理
mvn test      测试
mvn package   打包
mvn install   安装到本地仓库
```

依赖管理

依赖配置

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

添加依赖

依赖传递

我们声明了自己的项目需要`abc`，Maven会自动导入`abc`的jar包，再判断出`abc`需要`xyz`，又会自动导入`xyz`的jar包，这样，最终我们的项目会依赖`abc`和`xyz`两个jar包。

可选依赖

可选依赖指对外隐藏当前所依赖的资源

```
<optional>true</optional>
写在<dependency></dependency>中
```

比如将项目二添加为项目一的依赖，项目二依赖了junit，但是给junit添加了`<optional>true</optional>`,那么项目一就不知道项目二依赖了junit。

排除依赖

主动断开依赖的资源

项目一依赖了项目二，项目二依赖了junit，但是项目一不需要junit。

```
<exclusions>
    <exclusion>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </exclusion>
</exclusions>
写在<dependency></dependency>中
```

在项目一的xml中，将以上代码写入关于项目二的依赖配置

依赖范围

依赖的jar默认情况可以在任何地方使用，可以通过scope标签设定其作用范围。

作用范围：

- 主程序范围有效（main文件夹范围内）
- 测试程序范围有效（test文件夹范围内）
- 是否参与打包（package指令范围内）

scope	主代码	测试代码	打包	范例
compile(默认)	Y	Y	Y	log4j
test		Y		junit
provided	Y	Y		servlet-api
runtime			Y	jdbc

依赖范围的传递性(了解)

比如在项目二中将mybatis依赖设置一个scope，那么在项目一中把项目二添加为依赖，在项目一中mybatis的scope如下表：

- 带有依赖范围的资源在进行传递时，作用范围将受到影响

	compile	test	provided	runtime
compile	compile	test	provided	runtime
test				
provided				
runtime	runtime	test	provided	runtime

直接依赖

间接依赖

生命周期与插件

项目构建生命周期

maven对项目构建的生命周期划分为三套：

- clean：清理工作
- default：核心工作，例如编译，测试，打包，部署等
- site：产生报告，发布站点等

```
mvn compile  编译
mvn clean    清理
mvn test     测试
mvn package  打包
mvn install  安装到本地仓库
```

当执行以上指令时，Maven就会开始生命周期，它会从开始一直运行到插件对应的phase为止。

如果运行`mvn clean package`，Maven先执行`clean`生命周期并运行到`clean`这个phase，然后执行`default`生命周期并运行到`package`这个phase。

在实际开发过程中，经常使用的命令有：

`mvn clean`：清理所有生成的class和jar；

`mvn clean compile`：先清理，再执行到`compile`；

`mvn clean test`：先清理，再执行到`test`，因为执行`test`前必须执行`compile`，所以这里不必指定`compile`；

`mvn clean package`：先清理，再执行到`package`。

大多数phase在执行过程中，因为我们通常没有在`pom.xml`中配置相关的设置，所以这些phase什么事情都不做。

经常用到的phase其实只有几个：

- clean：清理
- compile：编译
- test：运行测试
- package：打包

执行一个phase又会触发一个或多个goal：

执行的Phase	对应执行的Goal
compile	compiler:compile
test	compiler:testCompile surefire:test

- lifecycle相当于Java的package，它包含一个或多个phase；
- phase相当于Java的class，它包含一个或多个goal；
- goal相当于class的method，它其实才是真正干活的。

插件

- 插件与生命周期内的阶段绑定，在执行到对应生命周期时插件执行对应的插件功能
- 默认maven在各个生命周期上绑定有预设的功能
- 通过插件可以自定义其他的功能

分模块开发与设计

分模块将我们之前SSM中所做的项目分解为多个模块，每个模块专注于完成某一组相关的功能或职责。比如分成pojo、controller、service、dao。（仅举例）

模块间通过接口相互作用

分模块的母的是提高代码的可维护性、复用性和可测试性

特点：

- 高内聚低耦合：模块内功能紧密相关，模块间的依赖尽可能少
- 代码重用：模块可以在不同的项目中重复使用
- 易于维护：模块化的结构使得代码更容易理解和维护

聚合

如果将各个模块独立，那么在一个模块进行改动时，其他模块不一定能适应它的修改。于是我们用聚合的思想，用一个额外的模块将其余模块都管理起来，对子模块统一操作(编译、打包等)，这个模块不需要写代码，只要pom.xml。

聚合不仅可以解决上述问题，还能够快速地构建maven工程。

方法：

- 我们给这个模块的`pom.xml`文件中加上：`<packaging>pom</packaging>`，表示这个模块专门用来管理，只提供pom.xml文件。
- 再用`<module>模块名</module>`，将剩余模块管理起来。再将所有`<module></module>`用`<modules></modules>`包起来

继承

如果模块之间使用的依赖版本不同，容易出现不兼容的情况。

于是我们用一个父模块将所有模块需要的依赖管理起来：

```
<dependencyManagement>
    <dependenties>
    <dependency>
    ...
    </dependency>
    </dependencies>
<dependencyManagement>
```

即只需要在外套一层。

在父模块中写明版本，那么子模块在用的时候就只需要声名需要的依赖，而不需要具体到它的版本了。

不仅是依赖，插件也是如此：只要用标签将标签包起来即可

当然，还要让父子模块联系起来：

```
<parent>
    <groupId></groupId>
    <artifactedId></artifactedId>
    <version></version> <!--这里是父模块的version，不是依赖的version -->
    <relativePath>父模块的pom文件路径</relativePath>
</parent>
```

虽然继承和聚合的父工程可以使用两个不同的模块，但是还是用一个父模块统一管理。

继承和聚合的区别

最重要的一点：继承是继承父模块的配置信息；聚合是用来快速构建的。

属性

自定义属性

格式：

```
<properties>
    <spring.version>5.1.9</spring.version>
</properties>
```

使用：

```
<dependency>
    ...
    ...
    <version>${spring.version}</version>
</dependency>
```

内置属性

在pom.xml中有一些内置属性，不需要自定义配置

调用：

```
${version}
```

版本管理

- SNAPSHOT:快照版
- RELEASE：发布版本
- 不同企业用的可能不同

资源配置

上面的<属性>中已经讲了pom.xml调用自定义的属性。

那么现在要让properties调用pom.xml定义的属性，如下来做：（以jdbc.rul为例）

- 还是先把jdbc.url写入标签中
- 使配置文件能够加载pom属性：

```
<resources>
    <resource>
        <directory>resources文件夹所在路径</directory>
        <filtering>true</filtering>
    </resource>
</resources>
```

- 调用格式还是${jdbc.url}

私服