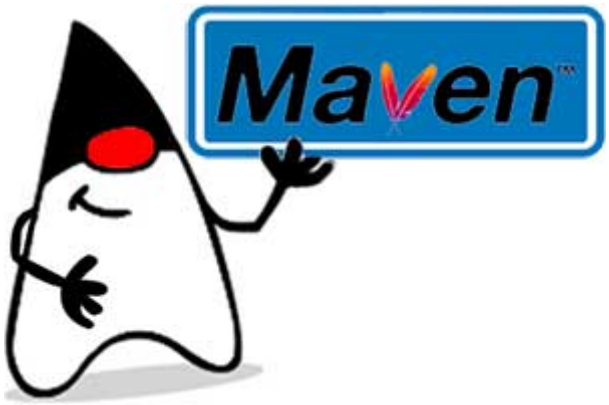
14 Maven基础

Maven是一个Java项目管理和构建工具，它可以定义项目结构、项目依赖，并使用统一的方式进行自动化构建，是Java项目不可缺少的工具。

本章我们详细介绍如何使用Maven。



Maven介绍

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
```

项目的根目录`a-maven-project`是项目名，它有一个项目描述文件`pom.xml`，存放Java源码的目录是`src/main/java`，存放资源文件的目录是`src/main/resources`，存放测试源码的目录是`src/test/java`，存放测试资源的目录是`src/test/resources`，最后，所有编译、打包生成的文件都放在`target`目录里。这些就是一个Maven项目的标准目录结构。

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。

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>
        ...
    </properties>
    <dependencies>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.2</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`commons-logging`：

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

使用``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。

安装Maven

要安装Maven，可以从Maven官网下载最新的Maven 3.6.x，然后在本地解压，设置几个环境变量：

```
M2_HOME=/path/to/maven-3.6.x
PATH=$PATH:$M2_HOME/bin
```

Windows可以把`%M2_HOME%\bin`添加到系统Path变量中。

然后，打开命令行窗口，输入`mvn -version`，应该看到Maven的版本信息：

```
┌─────────────────────────────────────────────────────┐
|Command Prompt                               - □ x |
├─────────────────────────────────────────────────────┤
|Microsoft Windows [Version 10.0.0]                   |
|(c) 2015 Microsoft Corporation. All rights reserved. |
|                                                     |
|C:\> mvn -version                                    |
|Apache Maven 3.6.0 (97c98ec64a1fdfee7767ce5ffb20918...) |
|Maven home: C:\Users\liaoxuefeng\maven               |
|Java version: ...                                    |
|...                                                  |
|C:\> _                                               |
|                                                     |
|                                                     |
└─────────────────────────────────────────────────────┘
```

如果提示命令未找到，说明系统PATH路径有误，需要修复后再运行。

小结

Maven是一个Java项目的管理和构建工具：

- Maven使用`pom.xml`定义项目内容，并使用预设的目录结构；
- 在Maven中声明一个依赖项可以自动下载并导入classpath；
- Maven使用`groupId`，`artifactId`和`version`唯一定位一个依赖。

依赖管理

如果我们的项目依赖第三方的jar包，例如commons logging，那么问题来了：commons logging发布的jar包在哪下载？

如果我们还希望依赖log4j，那么使用log4j需要哪些jar包？

类似的依赖还包括：JUnit，JavaMail，MySQL驱动等等，一个可行的方法是通过搜索引擎搜索到项目的官网，然后手动下载zip包，解压，放入classpath。但是，这个过程非常繁琐。

Maven解决了依赖管理问题。例如，我们的项目依赖`abc`这个jar包，而`abc`又依赖`xyz`这个jar包：

```
┌──────────────┐
│Sample Project│
└──────────────┘
        │
        ▼
┌──────────────┐
│     abc      │
└──────────────┘
        │
        ▼
┌──────────────┐
│     xyz      │
└──────────────┘
```

当我们声明了`abc`的依赖时，Maven自动把`abc`和`xyz`都加入了我们的项目依赖，不需要我们自己去研究`abc`是否需要依赖`xyz`。

因此，Maven的第一个作用就是解决依赖管理。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。

我们来看一个复杂依赖示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.4.2.RELEASE</version>
</dependency>
```

当我们声明一个`spring-boot-starter-web`依赖时，Maven会自动解析并判断最终需要大概二三十个其他依赖：

```
spring-boot-starter-web
  spring-boot-starter
    spring-boot
    sprint-boot-autoconfigure
    spring-boot-starter-logging
      logback-classic
        logback-core
        slf4j-api
      jcl-over-slf4j
        slf4j-api
      jul-to-slf4j
        slf4j-api
```

```
      log4j-over-slf4j
        slf4j-api
    spring-core
    snakeyaml
  spring-boot-starter-tomcat
    tomcat-embed-core
    tomcat-embed-el
    tomcat-embed-websocket
      tomcat-embed-core
  jackson-databind
  ...
```

如果我们自己去手动管理这些依赖是非常费时费力的，而且出错的概率很大。

依赖关系

Maven定义了几种依赖关系，分别是`compile`、`test`、`runtime`和`provided`：

SCOPE	说明	示例
compile	编译时需要用到该jar包（默认）	commons-logging
test	编译Test时需要用到该jar包	junit
runtime	编译时不需要，但运行时需要用到	mysql
provided	编译时需要用到，但运行时由JDK或某个服务器提供	servlet-api

其中，默认的`compile`是最常用的，Maven会把这种类型的依赖直接放入classpath。

`test`依赖表示仅在测试时使用，正常运行时并不需要。最常用的`test`依赖就是JUnit：

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.3.2</version>
    <scope>test</scope>
</dependency>
```

`runtime`依赖表示编译时不需要，但运行时需要。最典型的`runtime`依赖是JDBC驱动，例如MySQL驱动：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.48</version>
    <scope>runtime</scope>
</dependency>
```

`provided`依赖表示编译时需要，但运行时不需要。最典型的`provided`依赖是Servlet API，编译的时候需要，但是运行时，Servlet服务器内置了相关的jar，所以运行期不需要：

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.0</version>
    <scope>provided</scope>
</dependency>
```

最后一个问题是，Maven如何知道从何处下载所需的依赖？也就是相关的jar包？答案是Maven维护了一个中央仓库（repo1.maven.org），所有第三方库将自身的jar以及相关信息上传至中央仓库，Maven就可以从中央仓库把所需依赖下载到本地。

Maven并不会每次都从中央仓库下载jar包。一个jar包一旦被下载过，就会被Maven自动缓存在本地目录（用户主目录的`.m2`目录），所以，除了第一次编译时因为下载需要时间会比较慢，后续过程因为有本地缓存，并不会重复下载相同的jar包。

唯一ID

对于某个依赖，Maven只需要3个变量即可唯一确定某个jar包：

- groupId：属于组织的名称，类似Java的包名；
- artifactId：该jar包自身的名称，类似Java的类名；
- version：该jar包的版本。

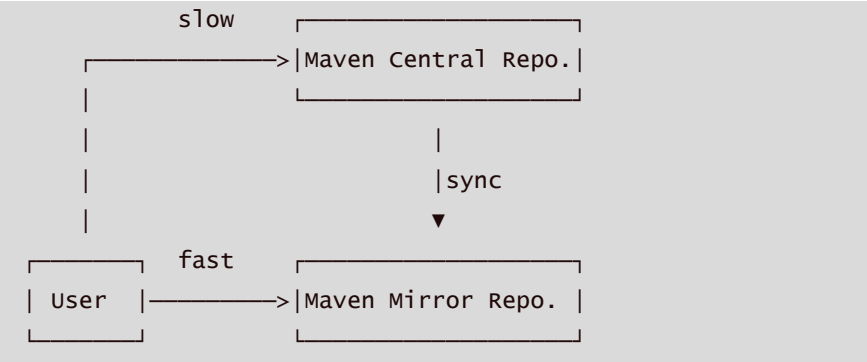
通过上述3个变量，即可唯一确定某个jar包。Maven通过对jar包进行PGP签名确保任何一个jar包一经发布就无法修改。修改已发布jar包的唯一方法是发布一个新版本。

因此，某个jar包一旦被Maven下载过，即可永久地安全缓存在本地。

注：只有以`SNAPSHOT-`开头的版本号会被Maven视为开发版本，开发版本每次都会重复下载，这种SNAPSHOT版本只能用于内部私有的Maven repo，公开发布的版本不允许出现SNAPSHOT。

Maven镜像

除了可以从Maven的中央仓库下载外，还可以从Maven的镜像仓库下载。如果访问Maven的中央仓库非常慢，我们可以选择一个速度较快的Maven的镜像仓库。Maven镜像仓库定期从中央仓库同步：



中国区用户可以使用阿里云提供的Maven镜像仓库。使用Maven镜像仓库需要一个配置，在用户主目录下进入`.m2`目录，创建一个`settings.xml`配置文件，内容如下：

```
<settings>
    <mirrors>
        <mirror>
            <id>aliyun</id>
            <name>aliyun</name>
            <mirrorOf>central</mirrorOf>
            <!-- 国内推荐阿里云的Maven镜像 -->

 <url>http://maven.aliyun.com/nexus/content/groups/public/
</url>
        </mirror>
    </mirrors>
</settings>
```

配置镜像仓库后，Maven的下载速度就会非常快。

搜索第三方组件

最后一个问题：如果我们要引用一个第三方组件，比如`okhttp`，如何确切地获得它的`groupId`、`artifactId`和`version`？方法是通过search.maven.org搜索关键字，找到对应的组件后，直接复制：

练习

下载练习：使用Maven编译hello项目 （推荐使用IDE练习插件快速下载）

小结

- Maven通过解析依赖关系确定项目所需的jar包，常用的4种`scope`有：`compile`（默认），`test`，`runtime`和`provided`；
- Maven从中央仓库下载所需的jar包并缓存在本地；
  -可以通过镜像仓库加速下载。

构建流程

构建流程

Maven不但有标准化的项目结构，而且还有一套标准化的构建流程，可以自动化实现编译，打包，发布，等等。

Lifecycle和Phase

使用Maven时，我们首先要了解什么是Maven的生命周期（lifecycle）。

Maven的生命周期由一系列阶段（phase）构成，以内置的生命周期`default`为例，它包含以下phase：

- validate
- initialize
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile
- process-classes
- generate-test-sources
- process-test-sources
- generate-test-resources
- process-test-resources
- test-compile
- process-test-classes
- test
- prepare-package
- package
- pre-integration-test
- integration-test
- post-integration-test
- verify
- install
- deploy

如果我们运行`mvn package`，Maven就会执行`default`生命周期，它会从开始一直运行到`package`这个phase为止：

- validate

- ...
- package

如果我们运行`mvn compile`，Maven也会执行`default`生命周期，但这次它只会运行到`compile`，即以下几个phase：

- validate
- ...
- compile

Maven另一个常用的生命周期是`clean`，它会执行3个phase：

- pre-clean
- clean （注意这个clean不是lifecycle而是phase）
- post-clean

所以，我们使用`mvn`这个命令时，后面的参数是phase，Maven自动根据生命周期运行到指定的phase。

更复杂的例子是指定多个phase，例如，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，实际执行的phase如下：

- pre-clean
- clean （注意这个clean是phase）
- validate
- ...
- package

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

Goal

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 surefile:test

goal的命名总是`abc:xyz`这种形式。

看到这里，相信大家对lifecycle、phase和goal已经明白了吧？

其实我们类比一下就明白了：

- 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。

大多数情况，我们只要指定phase，就默认执行这些phase默认绑定的goal，只有少数情况，我们可以直接指定运行一个goal，例如，启动Tomcat服务器：

```
mvn tomcat:run
```

小结

Maven通过lifecycle、phase和goal来提供标准的构建流程。

最常用的构建命令是指定phase，然后让Maven执行到指定的phase：

- mvn clean
- mvn clean compile
- mvn clean test
- mvn clean package

通常情况，我们总是执行phase默认绑定的goal，因此不必指定goal。

使用插件

我们在前面介绍了Maven的lifecycle，phase和goal：使用Maven构建项目就是执行lifecycle，执行到指定的phase为止。每个phase会执行自己默认的一个或多个goal。goal是最小任务单元。

我们以`compile`这个phase为例，如果执行：

```
mvn compile
```

Maven将执行`compile`这个phase，这个phase会调用`compiler`插件执行关联的`compiler:compile`这个goal。

实际上，执行每个phase，都是通过某个插件（plugin）来执行的，Maven本身其实并不知道如何执行`compile`，它只是负责找到对应的`compiler`插件，然后执行默认的`compiler:compile`这个goal来完成编译。

所以，使用Maven，实际上就是配置好需要使用的插件，然后通过phase调用它们。

Maven已经内置了一些常用的标准插件：

插件名称	对应执行的**PHASE**
clean	clean
compiler	compile
surefire	test
jar	package

如果标准插件无法满足需求，我们还可以使用自定义插件。使用自定义插件的时候，需要声明。例如，使用`maven-shade-plugin`可以创建一个可执行的jar，要使用这个插件，需要在`pom.xml`中声明它：

```
<project>
    ...
    <build>
        <plugins>
            <plugin>

<groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-
plugin</artifactId>
                <version>3.2.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            ...
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

自定义插件往往需要一些配置，例如，`maven-shade-plugin`需要指定Java程序的入口，它的配置是：

```
<configuration>
    <transformers>
        <transformer
implementation="org.apache.maven.plugins.shade.resource.Ma
nifestResourceTransformer">

 <mainClass>com.itranswarp.learnjava.Main</mainClass>
        </transformer>
    </transformers>
</configuration>
```

注意，Maven自带的标准插件例如`compiler`是无需声明的，只有引入其它的插件才需要声明。

下面列举了一些常用的插件：

- maven-shade-plugin：打包所有依赖包并生成可执行jar；
- cobertura-maven-plugin：生成单元测试覆盖率报告；
- findbugs-maven-plugin：对Java源码进行静态分析以找出潜在问题。

练习

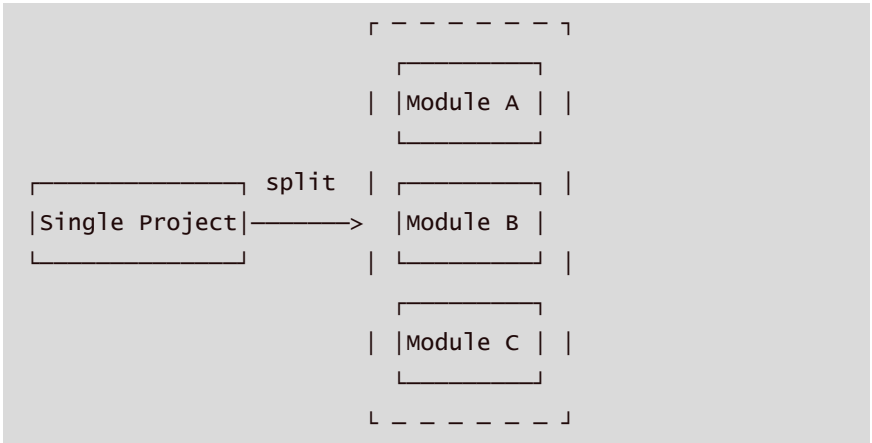
下载练习：使用maven-shade-plugin创建可执行jar（推荐使用IDE练习插件快速下载）

小结

- Maven通过自定义插件可以执行项目构建时需要的额外功能，使用自定义插件必须在pom.xml中声明插件及配置；
- 插件会在某个phase被执行时执行；
- 插件的配置和用法需参考插件的官方文档。

模块管理

在软件开发中，把一个大项目分拆为多个模块是降低软件复杂度的有效方法：



对于Maven工程来说，原来是一个大项目：

```
single-project
├── pom.xml
└── src
```

现在可以分拆成3个模块：

```
single-project
├── module-a
|   ├── pom.xml
|   └── src
├── module-b
|   ├── pom.xml
|   └── src
└── module-c
    ├── pom.xml
    └── src
```

Maven可以有效地管理多个模块，我们只需要把每个模块当作一个独立的Maven项目，它们有各自独立的pom.xml。例如，模块A的pom.xml：

模块B的pom.xml：

可以看出来，模块A和模块B的pom.xml高度相似，因此，我们可以提取出共同部分作为parent：

注意到parent的packaging是pom而不是jar，因为parent本身不含任何java代码。编写parent的pom.xml只是为了在各个模块中减少重复的配置。现在我们的整个工程结构如下：

```
single-project
├── parent
|   └── pom.xml
├── module-a
|   ├── pom.xml
|   └── src
├── module-b
|   ├── pom.xml
|   └── src
└── module-c
    ├── pom.xml
    └── src
```

如果模块A依赖模块B，则模块A需要模块B的jar包才能正常编译：

中央仓库

其实我们使用的大多数第三方模块都是这个用法，例如，我们使用commons logging、log4j这些第三方模块， 就是第三方模块的开发者自己把编译好的jar包发布到maven的中央仓库中。

私有仓库

本地仓库

但是我们不推荐把自己的模块安装到maven的本地仓库，因为每次修改模块b的源码，都需要重新安装， 容易出现版本不一致的情况

推荐的做法是模块化编译，在编译的时候，告诉maven几个模块之间存在依赖关系，需要一块编译， maven就会自动按依赖顺序编译这些模块

```
<modules>
    <module>模块A</module>
    <module>模块B</module>
    <module>模块C</module>
</modules>
```

Maven支持模块化管理，可以把一个大项目拆成几个模块 可以通过继承在parent的pom.xml统一定义重复配置 可以通过``编译多个模块

使用mvnw

我们使用Maven时，基本上只会用到`mvn`这一个命令。有些童鞋可能听说过`mvnw`，这个是啥？

`mvnw`是Maven Wrapper的缩写。因为我们安装Maven时，默认情况下，系统所有项目都会使用全局安装的这个Maven版本。但是，对于某些项目来说，它可能必须使用某个特定的Maven版本，这个时候，就可以使用Maven Wrapper，它可以负责给这个特定的项目安装指定版本的Maven，而其他项目不受影响。

简单地说，Maven Wrapper就是给一个项目提供一个独立的，指定版本的Maven给它使用。

安装Maven Wrapper

安装Maven Wrapper最简单的方式是在项目的根目录（即`pom.xml`所在的目录）下运行安装命令：

```
mvn -N io.takari:maven:0.7.6:wrapper
```

它会自动使用最新版本的Maven。注意`0.7.6`是Maven Wrapper的版本。最新的Maven Wrapper版本可以去官方网站查看。

如果要指定使用的Maven版本，使用下面的安装命令指定版本，例如`3.3.3`：

```
mvn -N io.takari:maven:0.7.6:wrapper -Dmaven=3.3.3
```

安装后，查看项目结构：

```
my-project

```

```
├── .mvn
│   └── wrapper
│       ├── MavenWrapperDownloader.java
│       ├── maven-wrapper.jar
│       └── maven-wrapper.properties
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        └── resources
```

发现多了`mvnw`、`mvnw.cmd`和`.mvn`目录，我们只需要把`mvn`命令改成`mvnw`就可以使用跟项目关联的Maven。例如：

```
mvnw clean package
```

在Linux或macOS下运行时需要加上`./`：

```
./mvnw clean package
```

Maven Wrapper的另一个作用是把项目的`mvnw`、`mvnw.cmd`和`.mvn`提交到版本库中，可以使所有开发人员使用统一的Maven版本。

练习

下载练习：使用mvnw编译hello项目 （推荐使用IDE练习插件快速下载）

小结

- 使用Maven Wrapper，可以为一个项目指定特定的Maven版本。