Maven实战

第2章 maven的安装和配置

2.3 安装目录分析

2.3.1 M2_HOME

- bin：该目录包含了mvn运行的脚本，还有一个mvnDebug，比mvn多了一台哦MAVEN_DEBUG_OPTS配置
- conf：包含一个非常重要的文件settings.xml，可以把它复制到~/.m2/目录下，在用户范围内定制Maven行为

2.4 设置HTTP代理

先运行命令：`ping repol.maven.org`，确认是否可以访问公共Maven中央仓库

检查代理服务器是否畅通：`telnet 218.14.227.197 3228`，假设代理IP是218.14.227.197，端口是3228

编辑settings.xml文件，添加代理配置如下：

```
<settings>
    ...
    <proxies>
        <proxy>
            <id>my-proxy</id>
            <active>true</active>
            <protocol>http</protocol>
            <host>218.14.227.197</host>
            <port>3228</port>
            <!--
           <username>***</username>
            <password>***</password>
            <nonProxyHosts>repository.mycom.com|*.google.com</nonProxyHosts>
            -->
        </proxy>
    </proxies>
</settings>
```

proxies下可以有多个proxy元素，如果声明了多个proxy元素，则默认情况下第一个被激活的proxy会生效。当代理服务需要认证时就需要配置username和password。nonProxyHost元素用来指定哪些主机名不需要代理。可以使用“|”符号来分割多个主机名。该配置页支持通配符，如*.google.com表示所有以google.com结尾的域名访问都不需要通过代理。

第3章 Maven使用入门

3.2 编写主代码

项目的主代码会被打包到最终的构件中（如jar），而测试代码只在运行测试时用到。默认情况下，Maven假设项目主代码位于src/main/java目录。

```
mvn clean compile
```

这个命令会执行`clean:clean`，`resources:rescources`，`compiler:compile`

clean告诉Maven清理输出目录target/，resources执行项目资源，compile告诉Maven编译项目主代码。

`clean:clean`，`resources:rescources`和`compiler:compile`分别对应了一些Maven插件及插件目标，比如`clean:clean`对应了clean插件的clean目标。

3.3 编写测试代码

Maven中默认的测试代码目录是src/test/java。

pom.xml中scope为依赖范围，若依赖范围为test，则表示该依赖只对测试有效，换句话说，测试代码中的`import JUnit`代码是没有问题的，但是如果在主代码中用`import JUnit`代码，就会造成编译错误。如果不生命依赖范围，***默认值就是compile，表示该依赖对主代码和测试代码都有效。***

测试用例编写完就可以调用Maven执行测试：：`mvn clean test`

上述命令会执行`clean:clean`，`resources:resources`，`compiler:compile`，`resources:testResources`，`compiler:testCompile`和`surefire:test`。在Maven执行测试之前，它会先自动执行项目主资源处理，主代码编译，测试资源处理，测试代码编译等工作，这是Maven生命周期的一个特性。surefire是Maven中负责执行测试的插件。

在执行`compiler:testCompile`时候失败，Maven输出提示我们需要使用-source 5或更高版本以启动`@Test`注释，这是因为由于历史原因，***Maven的核心插件之一compiler插件默认只支持编译ava1.3***，因此需要配置该插件使其支持java 5：

```
<project>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

3.4 打包和运行

将项目编译，测试之后，下一个重要步骤就是打包：`mvn clean package`。

Maven会在打包之前执行编译、测试等操作。`jar:jar`负责打包。

如果让其他的Maven项目直接引用当前jar包，需要执行：`mvn clean install`。打包之后，执行安装任务`install:install`，该任务将项目输出的jar安装到了Maven本地仓库。***只有构建被下载（安装）到本地仓库后，才能由所有Maven项目使用。***

执行test之前会先执行compile，执行package之前会先执行test，类似地，install之前会先执行package。

默认打包生成的kar是不能够直接运行的，因为带有main方法的类信息不会添加到manifest中（打开jar文件中的META-INF/MANIFEST.MF文件，将无法看到Main-Class一行）。为了生成可执行的jar文件，需要借助maven-shade-plugin，配置该插件如下：

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>1.2.1</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>xxx</mainClass>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

现在执行`mvn clean install`，待构建完成后打开target/目录，可以看到aa.jar和original-aa.jar，前者是带有Main-Class信息的可运行的jar，后者是原始的jar。

第5章 坐标和依赖

5.2 坐标详解

- groupId：定义当前Maven项目隶属的实际项目
- artifactId：定义实际项目中的一个Maven项目（模块），推荐的做法是使用实际项目名称作为artifactId的前缀。
- version：定义Maven项目当前所处的版本。
- packaging：定义Maven项目的打包方式。打包方式会影响到构建的生命周期，比如jar和war打包会使用不同的命令。
- classifier：定义构件输出的一些附属构件。不能直接定义项目的classifier，因为附属构件不是项目直接默认生成的，而是由附加的插件帮助生成。

5.5 依赖范围

Maven有以下几种依赖范围：

- compile：默认依赖范围，对于编译，测试，运行都有效
- test：只对测试有效，在编译主代码或者运行项目的使用时无法使用此类依赖
- provided：对于编译和测试有效，但在运行时无效。典型例子时servlet-api，编译和测试项目的时候需要该依赖，但在运行项目的时候，由于容器已经提供，就不需要Maven重复地引入一遍。
- runtime：对于测试和运行有效，但在编译主代码时无效。典型例子是JDBC驱动实现，项目主代码的编译只需要JDK提供的JDBC接口，只有在执行测试或者运行项目的时候才需要实现上述接口的具体JDBC驱动。
- system：对于编译和测试有效，但是使用system范围的依赖时必须通过systemPath元素显式地指定依赖文件的路径。由于此类依赖不是通过Maven仓库解析的，而且往往与本机系统绑定，可能造成构建的不可移植。systemPath可以引用环境变量。

```
<dependency>
    <groupId>javax.sql</groupId>
    <artifactId>jdbc-stdext</artifactId>
    <version>2.0</version>
    <scope>system</scope>
    <systemPath>${java.home}/lib/rt.jar</systemPath>
</dependency>
```

- import：依赖范围：导入

依赖范围	对于编译classpath有效	对于测试classpath有效	对于运行时classpath有效	例子
compile	Y	Y	Y	spring-core
test	-	Y	-	JUnit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC驱动实现
system	Y	Y	-	本地的，Maven仓库之外的类库文件

5.6 传递性依赖

5.6.2 传递性依赖和依赖范围

依赖范围不仅可以控制依赖与三种classpath的关系，还对传递性依赖产生影响。假设A依赖B，B依赖C，我们说A对于B是第一直接依赖，B对于C是第二直接依赖，A对于C是传递性依赖。第一直接依赖的范围和第二直接依赖的范围决定了传递性依赖的范围。

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

如上图，最左边一行表示第一直接依赖范围，最上面一行表示第二直接依赖范围，中间的交叉单元格表示传递性依赖范围。

5.7 依赖调解

加入项目A有这样的依赖关系：A->B->C->X(1.0)、A->D->X(2.0)，X是A的传递性 依赖，但是有两个版本的X。Maven依赖调解的第一原则是：***路径最近者优先***。因此X(2.0)会被解析使用。Maven依赖调解的第二原则是：***第一声明者优先***。

5.8 可选依赖

假设A->B，B->X(可选)、b->Y(可选)。由于这里X，Y是可选依赖，依赖不会传递。

可选依赖就是在`<dependency></dependency>`中加入`<optional>true</optional>`

第6章 仓库

6.3 仓库的分类

对于Maven来说，仓库分为两类：本地仓库和远程仓库。当Maven根据坐标寻找构建的时候，它首先会查看本地仓库，如果本地仓库存在此构件，则直接使用；如果本地仓库不存在此构件，或者需要查看是否有更新的构件版本，就会去远程仓库查找。

6.4 远程仓库的配置

在pom中配置远程仓库

```
<project>
    ...
    <repositories>
        <repository>
            <id>jboss</id>
            <name>JBoss Repository</name>
            <url>http://repository.jboss.com/maven2</url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>

            </snapshots>
```

```
            <layout>default</layout>
        </repository>
    </repositories>
    ...
</project>
```

在repositories元素中，可以使用repository子元素声明一个或多个远程仓库。任何一个仓库声明的id必须是唯一的，尤其需要注意的是，***Maven自带的中央仓库使用的id是central***，如果其他仓库也声明该id，就会覆盖中央仓库的配置。

该例配置中的release和snapshots元素比较中通，它们用来控制Maven对于发布版构件和快照版构件的下载。release的enable为true，表示JBoss仓库的发布颁布下载支持；snapshots的enable为false，表示关闭该仓库的快照版本的下载支持。因为Maven只会从该仓库下载发布版的构件，而不会下载快照版的构件。

layout元素值default表示仓库的布局是Maven2及Maven3的默认布局，而不是Maven1的布局。

对于release和snapshots来说，除了enabled，还包含另外两个子元素：updatePolicy和checksumPolicy。

6.4.1 远程仓库的认证

如果远程仓库需要用户名和密码，需要在settings.xml中配置

```
<settings>
    ...
    <servers>
        <server>
            <id>my-proj</id>
            <username>repo-user</username>
            <password>repo-pwd</password>
        </server>
    </servers>
    ...
</settings>
```

> settings.xml中server元素的id必须与pom中需要认证的repository元素的id完全一致。

6.4.2 部署至远程仓库

```
<project>
    ...
    <distributionManagement>
        <repository>
            <id>release</id>
            <name>release repository</name>
            <url>http://ip:port/xxx</url>
        </repository>
        <repository>
            <id>snapshots</id>
            <name>snapshots repository</name>
            <url>http://ip:port/yyy</url>
        </repository>

    </distributionManagement>
```

```
    ...
</project>
```

id是该远程仓库的唯一标识，name是为了方便人阅读，关键的url表示该仓库的地址。往远程仓库部署构件的时候，往往需要认证。，配置认证的方式在6.4.1节。配置正确后，运行`mvn clean deploy`。如果项目当前的版本是快照版本，则部署到快照版本仓库地址，否则部署到发布版本仓库地址。

6.6 从仓库解析依赖的机制

1. 当依赖范围是system的时候，Maven直接从本地文件系统解析构件。
2. 根据依赖坐标计算仓库路径后，尝试直接从本地仓库寻找构件，如果发现相应构件，则解析成功。
3. 在本地仓库不存在相应构件的情况下，如果依赖的版本是显式地发布版本构件，如1.2、2.1-beta-1等，则遍历所有的远程仓库，发现后，下载并解析使用。
4. 如果依赖的版本是RELEASE或者LATEST，则基于远程仓库更新策略读取所有远程仓库的元数据，将其与本地仓库的对应元数据合并后，计算出RELEASE或者LATEST真实的值，然后基于这个真实的值检查本地和远程仓库，如步骤(2)和(3)。
5. 如果依赖的版本是SNAPSHOT，则基于远程仓库更新策略读取所有远程仓库的元数据，将其与本地仓库的对应元数据合并后，得到最新快照版本的值，然后基于该值检查本地仓库，或者从远程仓库下载。
6. 如果最后解析得到的构件版本是时间戳格式的快照，如1.4.1-20091104.121450-121，则复制其时间戳格式的文件至非时间戳格式，如SNAPSHOT，并使用该非时间戳格式的构件。

不推荐在依赖声明中使用LATEST和RELEASE，因为Maven随时都可能解析到不同的构件，可能今天LATEST是1.3.6，明天就是1.4.0-SNAPSHOT了。

6.7 镜像

任何一个从仓库Y获得的构件，都能够从它的镜像中获取。在settings.xml中配置镜像。

```
<settings>
    ...
    <mirrors>
        <mirror>
            <id>maven.net.cn</id>
            <name>one of the central mirrors in China</name>
            <url>http://maven.net.cn/content/groups/public</url>
            <mirrorOf>central</mirrorOf>
        </mirror>
    </mirrors>
    ...
</settings>
```

`<mirrorOf>`的值为central，表示该配置为中央仓库的镜像，任何对于中央仓库的请求都会转至该镜像。

- `<mirrorOf>*</mirrorOf>`：匹配所有远程仓库
- `<mirrorOf>external：*</mirrorOf>`：匹配所有远程仓库，使用localhost的除外
- `<mirrorOf>repo1,repo2</mirrorOf>`：匹配仓库repo1和repo2，使用逗号分割多个远程仓库
- `<mirrorOf>*,!repo3</mirrorOf>`：匹配所有远程仓库，repo1除外

第7章 生命周期和插件

7.1 何为生命周期

生命周期包含了项目的清理、初始化、编译、测试、打包、集成测试、验证、部署和站点生成等几乎所有构建步骤。在Maven中，实际的任务都交由插件来完成。

生命周期抽象了构建的各个步骤，定义了它们的次序，但没有提供具体实现。每个构建步骤可以绑定一个或多个插件行为，而且Maven为大多数构建步骤编写并绑定了默认插件。例如：编译是由maven-compiler-plugin完成，测试由maven-surefire-plugin完成。

7.2 生命周期详解

7.2.1 三套生命周期

Maven拥有三套相互独立的生命周期，***它们分别是clean，default和site***。clean的目的是清理项目，default的目的是构建项目，site的目的是建立项目站点。

每个生命周期包含一些阶段，这些阶段是有顺序的，并且后面的阶段依赖与前面的阶段，用户和Maven最直接的交互方式就是调用这些生命周期阶段。例如clean包含的阶段是pre-clean，clean和post-clean。当调用pre-clean时，只有pre-clean阶段执行，调用post-clean时，pre-clean、clean和post-clean会以顺序执行。

三个生命周期本身是相互独立的。当调用clean生命周期的clean阶段时，不会触及default生命周期的任何阶段。

7.2.2 clean生命周期

1. pre-clean：执行一些清理前需要完成的工作
2. clean：清理上一次构建生成的文件
3. post-clean：执行依稀清理后需要完成的工作

7.2.3 default生命周期

1. validate
2. initialize
3. generate-sources
4. process-sources：处理项目主资源文件，一般来说，是对src/main/resources目录的内容进行变量替换等工作后，复制到项目输出的主classpath目录中。
5. generate-resources
6. process-resources
7. compile：编译项目的主源码，一般来说，是编译src/main/java目录下的java文件至项目输出的主classpath目录中。
8. process-classes
9. generate-test-sources
10. process-test-sources：处理项目测试资源文件，一般来说，是对src/test/resources目录的内容进行变量替换等工作后，复制到项目输出的测试classpath目录中。
11. generate-test-resources
12. test-compile：编译项目的测试代码，一般来说，是编译src/test/java目录下的java文件至项目输出的测试classpath目录中。
13. process-test-classes
14. test：使用单元测试框架运行测试，测试代码不会被打包或部署。
15. prepare-package

16. package：接受编译好的代码，打包成可发布的格式，如JAR
17. pre-integration-test
18. integration-test
19. post-integration-test
20. verify
21. install：将包安装到Maven本地仓库，供本地其他Maven项目使用。
22. deploy：将最终的包复制到远程仓库，供其他开发人员和Maven项目使用。

7.2.4 site生命周期

1. pre-site
2. site
3. post-site
4. site-deploy

7.2.5 命令行与生命周期

- `mvn clean`：调用clean生命周期的clean阶段，实际执行pre-clean和clean
- `mvn test`：调用default生命周期的test阶段，实际执行validate等直到test的所有阶段。
- `mvn clean install`：调用clean生命周期的clean阶段和default生命周期的install阶段。

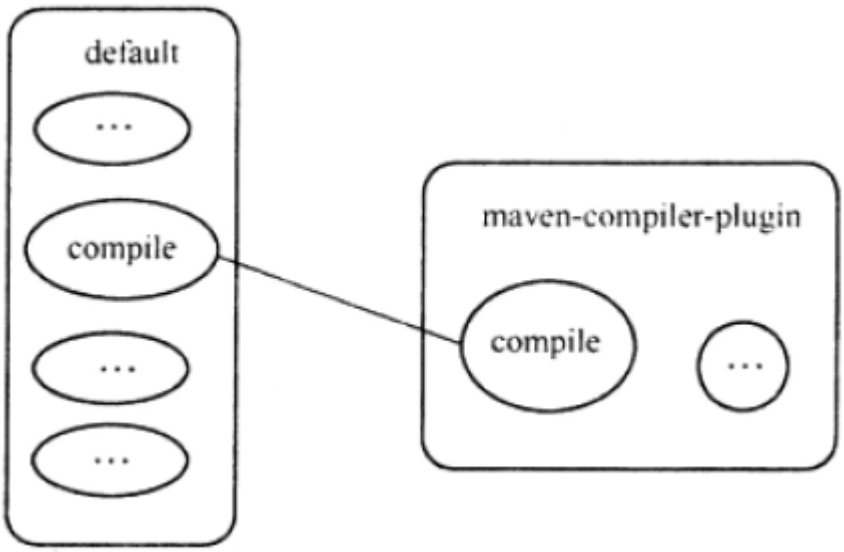
7.3 插件目标

一个插件可能有多个目标，每个目标对应了一个功能，比如maven-dependency-plugin插件：

- `dependency:analyze`：对应的目标是分析项目依赖；
- `dependency:tree`：对应的目标是列出项目的依赖树。

7.4 插件绑定

生命周期的阶段与插件的目标相互绑定，以完成某个具体的构建任务。



当插件目标被绑定到不同的生命周期阶段的时候，其执行顺序会由生命周期阶段的先后顺序决定。如果多个目标被绑定到同一个阶段的时候，这些插件声明的先后顺序决定了目标的执行顺序。

7.5 插件配置

7.5.1 命令行插件配置

例如maven-surefire-plugin提供了一个maven.test.skip参数，当其值为true时，会跳过执行测试。运输可以运行以下命令：

```
mvn install -Dmaven.test.skip=true
```

-D是java自带的，其功能时通过命令行设置一个java系统属性。

第8章 聚合与继承

Maven的聚合特性能够把项目的各个模块聚合在一起构建，而Maven的继承特性则能帮助抽取各模块相同的依赖和插件等配置。

8.2 聚合

```
<modules>
    <module>account-email</module>
    <module>account-persist</module>
</modules>
```

这里每个module的值都是一个***当前pom的相对目录***。

8.3 继承

```
<parent>
    <groupId>com.juvenxu.account</groupId>
    <artifactId>account-parent</artifactId>
    <version>1.5.10</version>
    <relativePath>../account-parent/pom.xml</relativePath>
</parent>
```

元素relativePath表示父POM的位置在于该项目目录平行的account-parent目录下。当项目构建时，Maven会首先根据relativePath检查父pom，如果找不到，再从本地仓库查找。relativePath的默认值时../pom.xml，也就是说，Maven默认父pom在上一层目录下。

8.3.3 依赖管理

父模块中的依赖有可能新增加的子模块中不需要，这就造成了空间的浪费，所以可以在父模块中使用：

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>1.5.10.RELEASE</version>
        </dependency>
        ...
    </dependencies>
</dependencyManagement>
```

这样在子模块中，可以使用父模块中的`<dependencyManagement></dependencyManagement>`元素中定义的依赖：

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

使用dependencyManagement声明的依赖既不会给父模块引入依赖，也不会给它的子模块引入依赖，不过这段胚子是***会被继承的***。因此子模块中就可以省去version和scope等。

父POM中使用dependencyManagement声明依赖能够统一项目范围中依赖的版本，当依赖版本在父POM中声明之后，子模块在使用依赖的时候就无需声明版本，也就不会发生多个子模块使用依赖版本不一致的情况，降低了依赖冲突的记录。

如果子模块不生命依赖的使用，即使该依赖已经在父pom的dependencyManagement中声明，也不会产生实际效果。

8.4 聚合与集成的关系

聚合是为了方便快速构建项目，继承是为了消除重复配置。

对于聚合模块来说，它知道有哪些被聚合的模块，但那些被聚合的模块不知道这个聚合模块的存在。

对于继承关系的父pom来说，它不知道有哪些子模块继承于它，但那些子模块都必须知道自己的父pom是什么。

8.5 约定优于配置

Maven中有一个超级POM，任何一个Maven项目都隐式地继承自该POM，这有点类似于任何一个Java类都隐式地继承Object类，因此大量超级POM的配置都会被所有Maven项目继承。

对于Maven3，超级POM在文件$MAVEN_HOME/lib/maven-model-builder-x.x.x.jar中的org/apache/maven/model/pom-4.0.0.xml路径下。

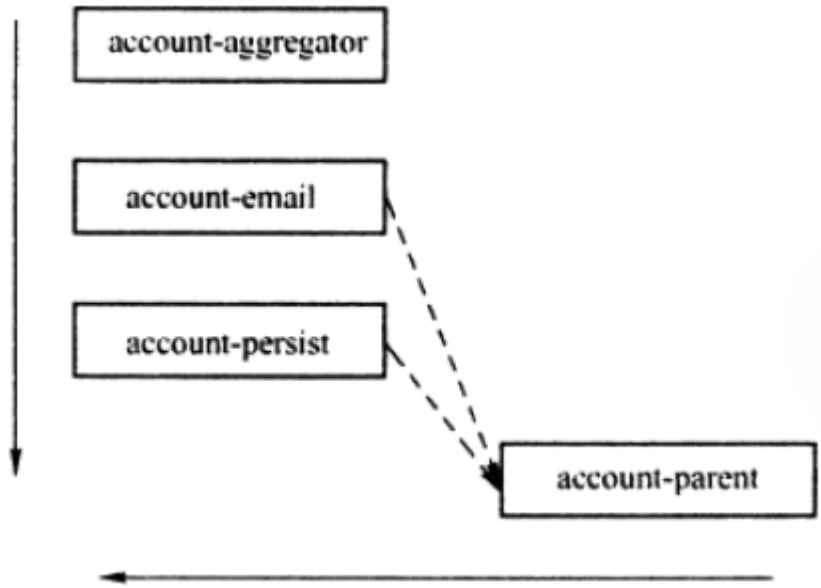
8.6 反应堆的构建顺序

在一个多模块的Maven项目中，反应堆是指所有模块组成的一个构建结构。对于单模块项目，反应堆就是该模块本身，但对于多模块项目，反应堆就包含了各模块直接继承与依赖的关系，从而能够自动计算出合理的模块构建顺序。

8.6.1 反应堆的构建顺序

聚合模块为account-aggregator

```
<modules>
    <module>account-email</module>
    <module>account-persist</module>
    <module>account-parent</module>
</modules>
```



实际输出的反应堆的构建顺序是：account-aggregator -> account-parent -> account-email -> account-persist。

account-email和account-persist依赖于account-parent，那么account-parent就必须先于另外两个模块构建。

Maven按序读取pom，如果该pom没有依赖模块，那么就构建该模块，否则就先构建其依赖模块，如果该依赖还依赖其他模块，则进一步先构建依赖的依赖。

第10章 使用maven进行测试

10.2 maven-surefire-plugin简介

Maven本身不是一个单元测试框架，Maven所做的只是在构建执行到特定生命周期阶段的时候，通过插件来执行JUnit或者TestNG的测试用例。这一插件就是maven-surefire-plugin。

maven-surefire-plugin的test目标和default生命周期的test阶段绑定。

在默认情况下，maven-surefire-plugin的test目标会自动执行测试源码路径（默认为src/test/java/）下所有符合一组命名模式的测试类，这组模式为：

- `**/Test*.java`：任何子目录下所有命名以Test开头的Java类
- `**/*Test.java`：任何子目录下所有命名以Test结尾的Java类
- `**/*TestCase.java`：任何子目录下所有命名以TestCase结尾的Java类

10.3 跳过测试

提哦过测试运行

```
mvn package-DskipTests
```

跳过测试代码的编译

```
mvn package-Dmaven.test.skip=true
```

第14章 灵活的构建

14.1 Maven属性

```
<properties>
    <fastjson.version>1.2.20</fastjson.version>
</properties>
```

这只是6类Maven属性中的一类，这6类属性分别为：

- 内置属性：主要是两个常用内置属性— `${basedir}` 表示项目根目录，即包含pom.xml文件的目录； `${version}` 表示项目版本。
- POM属性：用户可以使用该类属性引用POM文件中对应元素的值，例如：
  - `${project.build.sourceDirectory}` ：项目的主源码目录，默认为：src/main/java/。
  - `${project.build.testSourceDirectory}` ：项目的测试源码目录。
  - `${project.build.directory}` ：项目构建输出目录，默认为：target/
  - `${project.outputDirectory}` ：项目主代码编译输出目录，默认为：target/classes/
  - `${project.testOutputDirectory}` ：项目测试代码编译输出目录，默认为：target/test-classes/。
  - `${project.groupId}` ：项目的groupId。
  - `${project.artifactId}` ：项目的artifactId。
  - `${project.version}` ：项目的version，与 `${version}` 等价。
  - `${project.build.finalName}` ：项目打包输出文件的名称，默认为 `${project.artifactId}-${project.version}` 。
- 自定义属性：用户可以在POM的元素下自定义Maven属性，例如：

  ```
  <properties>
      <my.prop>hello</my.prop>
  </properties>
  ```

  然后在POM中其他地方使用 `${my.prop}` 的时候会被替换成hello。
- Settings属性：与POM属性同理，用户使用以 `settings.` 开头的属性引用settings.xml文件中XML元素的值，如常用的 `${settings.localRepository}` 指向用户本地仓库的地址。
- Java系统属性：用户可以使用 `mvn help:system` 查看所有的Java系统属性。

- 环境变量属性：所有环境变量都可以使用以`env.`开头的Maven属性引用。例如`${env.JAVA_HOME}`指代了JAVA_HOME环境变量的值。用户可以使用`mvn help:system`查看所有的环境变量。

14.3 资源过滤

为了应对环境的变化，首先需要使用Maven属性将这些将会发生变化的部分提取处理，用Maven属性取代它们：

```
database.jdbc.driverClass=${db.driver}
database.jdbc.connectionURL=${db.url}
database.jdbc.username=${db.username}
database.jdbc.password=${db.password}
```

```
<profiles>
    <profile>
        <id>dev</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
            <db.username>dev-name</db.username>
            <db.password>dev-pwd</db.password>
        </properties>
    </profile>
</profiles>
```

Maven属性默认只有在POM中才会被解析。也就是说，`${db.username}`放到POM中会变成dev-name，但是如果放到src/main/resources/目录下的文件中，构建的时候它仍然是${db.username}。***因此需要让Maven解析资源文件中的Maven属性***。

资源文件的处理是maven-resources-plugin做的事情，它默认的行为只是将项目主资源文件复制到主代码编译输出目录中，将测试资源文件复制到测试代码编译输出目录中。不过通过一些POM配置，该插件就能够解析资源文件中的Maven属性，即开启资源过滤。

为主资源目录开启过滤：

```
<build>
    <resources>
        <resource>
            <directory>${project.basedir}/src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
</build>
```

为测试资源目录开启过滤：

```
<build>
    <testResources>
        <testResource>
            <directory>${project.basedir}/src/test/resources</directory>
            <filtering>true</filtering>
        </testResource>
    </testResources>
</build>
```

最后只需要在命令行激活profile：

```
mvn clean install -Pdev
```

mvn的-P参数表示在命令行激活一个profile，这里激活了id为dev的profile，构建完成后，输出目录中的数据库配置就是开发环境的配置了：

```
database.jdbc.driverClass=com.mysql.jdbc.Driver
database.jdbc.connectionURL=jdbc:mysql://192.168.1.100:3306/test
database.jdbc.username=dev-name
database.jdbc.password=dev-pwd
```

14.4 Maven Profile

不同环境的构建很可能是不同的。除此之外，有些环境可能需要配置插件使用本地文件。为了能让构建在各个环境下方便地移植，Maven引入了profile的概念。用户可以使用很多方式激活profile，以实现构建在不同环境下的移植。

14.4.1 针对不同环境的profile

```
<profiles>
    <profile>
        <id>dev</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/dev</db.url>
            <db.username>dev-name</db.username>
            <db.password>dev-pwd</db.password>
        </properties>
    </profile>
    <profile>
        <id>test</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
            <db.username>test-name</db.username>
            <db.password>test-pwd</db.password>
        </properties>
    </profile>
</profiles>
```

14.4.2 激活profile

为了尽可能方便用户，Maven支持很多种激活profile的方式

1. 命令行激活

用户可以使用mvn命令行参数-P加上profile的id来激活profile，多个id之间以逗号分隔。例如，下面的命令激活了dev-x和dev-y两个profile：

```
mvn clean install -Pdev-x,dev-y
```

2. settings文件显式激活

如果用户希望某个profile默认一直处于激活状态，就可以配置settings.xml文件的activeProfiles元素，表示其配置的profile对于所有项目都处于激活状态。

```
<settings>
    ...
    <activeProfiles>
        <activeProfile>dev-x</activeProfile>
    </activeProfiles>
    ...
</settings>
```

3. 系统属性激活

用户可以配置当某系统属性存在的时候，自动激活profile：

```
<profiles>
    <profile>
        <activation>
            <property>
                <name>abc</name>
            </property>
        </activation>
        <id>test</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/test</db.url>
            <db.username>test-name</db.username>
            <db.password>test-pwd</db.password>
        </properties>
    </profile>
</profiles>
```

当属性abc存在的时候，就自动激活id为test的profile。

可以进一步配置当某系统属性abc存在，且值为123的时候激活profile：

```
<profiles>
```

```
    <profile>
        <activation>
            <property>
                <name>abc</name>
                <value>123</value>
            </property>
        </activation>
        <id>dev</id>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/dev</db.url>
            <db.username>dev-name</db.username>
            <db.password>dev-pwd</db.password>
        </properties>
    </profile>
</profiles>
```

不要忘了，用户可以在命令行声明系统属性：

```
mvn clean install -Dabc=123
```

4. 操作系统环境激活

5. 文件存在与否激活

6. 默认激活

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <db.driver>com.mysql.jdbc.Driver</db.driver>
            <db.url>jdbc:mysql://192.168.1.100:3306/dev</db.url>
            <db.username>dev-name</db.username>
            <db.password>dev-pwd</db.password>
        </properties>
    </profile>
</profiles>
```

使用activeByDefault元素可以指定profile自动激活，不过需要注意的是，如果POM中有任何一个profile通过以上其他一种方式被激活了，所有默认激活配置都会失效。