Maven安装最佳实践

1) 设置MAVEN\_OPTS环境变量

mvn命令实际是执行了Java命令，因此Java命令可用的参数在运行mvn时也可用。Java默认的最大可用内存往往不能够满足Maven运行的需求，一般将MAVEN\_OPTS的值设置为-Xms128m –Xmx512m,并且设置到环境变量，而不建议修改mvn.bat或mvn，升级的时候会被覆盖，需要重新配置，麻烦并且容易忘记。

2) 配置用户范围的settings.xml

MAVEN有两个配置文件${m2\_HOME}/conf/settings.xml和${user}/.m2/settings.xml前者是全局范围的，整台机器上所有用户都会受该配置影响，而后者是用户范围的，只会影响当前用户。

推荐配置用户范围的配置，可以避免无意识影响其他用户，也可以避免maven升级时，配置被覆盖。

3) 不要使用IDE内嵌的Maven

内嵌的Maven版本通常会比较新，但不一定稳定，而且往往与命令行使用的Maven不是同个版本，因此由于不稳定，可能会出现一些难以理解的问题，其次版本不一致可能导致构建行为不一致。

1、资源排除

<resources>

<!-- 启动过滤，包含的文件会被过滤掉 -->

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

<includes>

<include>src/main/resources/dev/\*.\*</include>

<include>src/main/resources/local/\*.\*</include>

</includes>

</resource>

<!-- 不启动过滤，包含的文件会被打包 -->

<resource>

<directory>src/main/resources</directory>

<filtering>false</filtering>

<includes>

<include>\*.xml</include>

<include>\*.properties</include>

</includes>

</resource>

</resources>

2、环境区分

<!-- 配置 -->

<profiles>

<!-- 开发环境 -->

<profile>

<id>dev</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.1</version>

<executions>

<execution>

<phase>test</phase>

<goals>

<goal>run</goal>

</goals>

<configuration>

<tasks>

<echo>Using dev</echo>

<copy file="src/main/resources/dev/systemParm.properties"

tofile="${project.build.outputDirectory}/systemParm.properties" />

</tasks>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</profile>

<!-- 本地环境 -->

<profile>

<id>local</id>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-antrun-plugin</artifactId>

<version>1.1</version>

<executions>

<execution>

<phase>test</phase>

<goals>

<goal>run</goal>

</goals>

<configuration>

<tasks>

<echo>Using local</echo>

<copy file="src/main/resources/local/systemParm.properties"

tofile="${project.build.outputDirectory}/systemParm.properties" />

</tasks>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</profile>

</profiles>

3.配置jdk版本—maven全局配置

<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>

4.POM文件

Maven服务与Java平台的项目构建、依赖管理和项目信息管理。Maven依赖于JDK。

POM：Project Object Model，项目对象模型，定义项目基本信息，描述项目如何构建，声明项目依赖等。

5.坐标和依赖

Maven通过下面几个元素定义坐标:groupId、artifactId、version、packaging、classifier.

groupId:定义当前Maven项目隶属的实际项目。不应该是组织或公司(一个公司有很多项目，每个项目有很多模块)。

artifactId:定义实际项目的一个Maven模块，推荐使用实际项目名称作为前缀。

version:定义Maven项目当前所处的版本。

packaging:定义Maven项目的打包方式，可选的有：jar(默认)、war、pom，不同的打包方式会影响构建的生命周期。

classifier:定义构建输出的一些附属构建，如javadoc、sources。

5.1 依赖的配置

一个依赖声明可以包含下面元素：

<dependencies>

<dependency>

<groupId></groupId>

<artifactId></artifactId>

<version></version>

<type></type>

<scope></scope>

<optional></optional>

<exclusions>

<exclusion>

</exclusion>

</exclusions>

</dependency>

</dependencies>

**Remark**

groupId、artifactId、version依赖的基本坐标。

type：依赖的类型，对应于项目坐标定义的packaging，默认：jar。

scope：依赖的范围，见5.2 依赖的范围。

optional：标志依赖是否可选，true/false。

exclusions：用来排除传递性依赖。

5.2 依赖范围

依赖范围是用来控制依赖于三种classpath(编译classpath、测试classpath、运行classpath)的关系。

Maven的依赖范围有如下几种：

compile:编译依赖范围，默认值，对三种classpath都有效。

test:测试依赖范围，只对测试classpath有效，典型例子如：Junit

provided:已提供依赖范围，对编译和测试classpath有效，但在运行时无效，典型例子如：servlet-api，运行时由容器提供。

runtime:运行时依赖范围，对测试和运行classpath有效，编译主代码时无效，典型例子如：JDBC驱动实现，编译时只需要JDK提供的JDBC接口，运行才需要具体的实现。

system:系统依赖范围，对编译和测试classpath有效，但在运行时无效使用该范围时，必须通过systemPath元素指定依赖的路径。

<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:导入依赖范围，该范围不会对三种classpath产生实际应用，会将目标POM中的dependencyManagement配置导入合并到当前POM的dependencyManagement元素中。

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

5.3 传递性依赖和依赖范围

Maven的传递性依赖是指不需要考虑你依赖的库文件所需要依赖的库文件，能够将依赖模块的依赖自动的引入。

依赖的范围不仅可以控制依赖与三种classpath的关系，还会对传递性依赖产生影响。假设A依赖于B，B依赖于C，则说A对于B是第一直接依赖，B对C是第二直接依赖，A对于C是传递依赖。第一直接依赖范围和第二直接依赖范围决定了传递性依赖的范围，其结果如下：

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

第二直接依赖范围是compile时，传递性依赖范围与第一直接依赖范围一致；

第二直接依赖范围是test时，依赖不会得以传递；

第二直接依赖范围是provided时，只传递第一直接依赖范围也为provided的；

第二直接依赖范围是runtime时，传递性依赖的范围与第一直接依赖范围一致，compile例如，此时的传递性依赖范围为runtime。

5.4 依赖调解

一般情况下，只关心项目的直接依赖，而不关心直接依赖引入的传递性依赖，但当传递性依赖出现问题时，需要知道该传递性依赖是怎么引进来的。

Maven依赖调解第一原则:路径最近者优先，如：A->B->C->X(1.0)、A->D-X(2.0)，则X的2.0版本会被解析使用；

Maven依赖调解第二原则:第一声明者优先，如:A->B->X(1.0)、A->D->X(2.0),若B的依赖声明在D之前，则使用X的1.0版本，否则使用X的2.0版本。

5.5 可选依赖

假设有下面的依赖关系:A->B、B->X(可选)、B->Y(可选)，由于X和Y是可选的，所以依赖不会传递，X和Y不会对A有任何影响。

可选依赖的必要性:项目B实现2种特性，特性一依赖于X，特性二依赖于Y，而且这两个特性是互斥的，用户不可能同时适用这两个特性，这时候可选依赖就有用了。

原则上说，是不应该使用可选依赖的，根据面向对象的单一职责性原则，该原则同样适用于Maven项目的规划。

5.6 最佳实践

1）排除依赖

传递性依赖会给项目隐式的引入很多依赖，这极大的简化了项目依赖的管理，但是有时某些依赖会带来问题，这时需要把带来问题的依赖排除掉。

2）归类依赖

来自同一个项目的不同模块，其版本号应该是相同的，如springframework项目有spring-core、spring-beans模块，对这些模块的版本号通过属性定义，再进行引用，这样可以进行版本的整体升级:

<properties>

<springframework.version>4.3.13.RELEASE</springframework.version>

</properties>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${springframework.version}</version>

</dependency>

3）优化依赖

去掉多余的依赖，显示声明某些必要的依赖。

通过mvn dependency:list 查看项目已解析的依赖

通过mvn dependency:tree 查看项目的依赖树

6. 仓库

Maven坐标和依赖时任何一个构件的逻辑表示，构件的物理表示是文件，Maven通过仓库来统一管理这些文件。

得益于坐标机制，Maven项目能够以统一的方式来使用任何构件，在此基础上，Maven可以在某个位置统一存储所有Maven项目共享的构建，这个统一位置就是仓库。

6.1 仓库的布局

Maven中的仓库分为：本地仓库和远程仓库。

Maven根据坐标寻找构件时，先查看本地仓库是否存在该构件，存在则直接使用；否则就查找远程仓库，找到之后就下载到本地仓库；本地和远程都没找到，直接报错。



中央仓库是Maven核心自带的远程仓库，含绝大多数开源的构件；

私服是在局域网搭建的仓库服务器，用于代理外部的远程仓库，可以节省带宽和时间，内部的项目还能部署到私服供其他项目使用；使用私服可以加速Maven构建以及提高稳定性，内网访问不需要依赖于网络。

其他公共服，如阿里云等。

本地仓库：配置路径${M2\_HOME}/conf/settings.xml,默认是${user\_Home}/.m2/repository.

构件进入本地仓库有两种方式:Maven从远程仓库下载到本地仓库；通过在项目执行mvn install安装到本地。

对Maven而言，用户的本地仓库只有一个，但可以配置访问很多远程仓库。而中央仓库是默认的远程仓库，在$M2\_HOME/lib/maven-model-builder-{version}.jar的org/apache/maven/model/pom-4.0.0.xml文件定义了，该POM也被称为超级POM。

构件在Maven仓库里的存储路径为:{groupId}/{artifactId}/{version}/{artifactId-version.packaging}

6.2 远程仓库的配置

7. 生命周期和插件

Maven的生命周期是对所有构建过程的抽象和统一。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。

Maven的生命周期是抽象的，其实际行为是由插件来完成的，生命周期和插件两者协同合作，密不可分。

这种思想与设计模式中的模板方法非常相似。模板方法模式在父类定义算法的整体结构，子类通过实现或者重写父类的方法来控制实际行为，这样既能保证算法有足够的可扩展性，又能严格控制算法的整体结构。

7.1 生命周期

Maven拥有3套独立的生命周期:clean、default、site。clean生命周期的目的是清理项目；default生命周期的目的是构建项目；site生命周期的目的是建立项目站点。

每个生命周期包含一些阶段(phase)，这些阶段是有序的，后面的阶段会依赖于前面的阶段。

clean生命周期的3个阶段：

1) pre-clean:执行一些清理前需要完成的动作

2) clean:清理上一次构建生成的文件

3) post-clean:执行一些清理后需要完成的动作

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，复制到项目输出的主classpath目录中。

5) generate-resources

6) process-resources

7) compile 编译项目的主源码到主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，复制到项目输出的测试classpath目录中。

11) generate-test-resources

12) process-test-resources

13) test-compile编译项目的测试源码到测试classpath目录中。

14) process-test-classes

15) test 使用单元测试框架进行测试，测试代码不会被打包或部署

16) prepare-package

17) package 将编译好的代码，打包成可发布的格式，如jar

18) pre-integration-test

19) integration-test

20) post-integration-test

21) verify

22) install 将包安装到Maven本地仓库，供本地其他Maven项目使用

23) 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

site生命周期的目的是建立和发布项目站点，Maven能够基于POM所包含的信息，自动生成一个友好的站点，方便团队交流和发布项目信息，含如下阶段：

1) pre-site 执行一些在生成项目站点之前需要完成的工作

2) site 生成项目站点文档

3) post-site执行一些在生成项目站点之后需要完成的工作

4) site-deploy 将生成的项目站点发布到服务器上

7.2 插件目标

对于一个插件，为了复用代码，它往往能够完成多个任务，例如maven-dependency-plugin,能够分析项目依赖；列出项目依赖树；列出项目已解析的依赖，为这样每个功能独立编写一个插件，显然是不可取的，因为这些功能背后有相同的代码，因此将这些功能聚集在一个插件里，每个功能就是一个插件目标。

7.3 插件绑定

Maven的生命周期与插件相互绑定，用于完成实际的构建任务，具体而言，是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。

下面是一些内置的绑定：

**clean生命周期阶段与插件目标**

|  |  |
| --- | --- |
| 生命周期阶段 | 插件目标 |
| pre-clean |  |
| clean | maven-clean-plugin:clean |
| post-clean |  |

maven-clean-plugin只有一个clean目标

**site生命周期阶段与插件目标**

|  |  |
| --- | --- |
| 生命周期阶段 | 插件目标 |
| pre-site |  |
| site | maven-site-plugin:site |
| post-site |  |
| site-deploy | maven-site-plugin:deploy |

项目的打包类型会影响构建的具体过程，因此default生命周期的阶段与插件目标的绑定关系由项目的打包类型所决定，下面以jar为示例：

**default生命周期阶段与插件目标(打包类型：jar)**

|  |  |  |
| --- | --- | --- |
| 生命周期阶段 | 插件目标 | 执行任务 |
| process-resources | maven-resources-plugin:resources | 复制主资源文件至主输出目录 |
| compile | maven-compiler-plugin:compile | 编译主代码至主输出目录 |
| process-test-resources | maven-resources-plugin:testResources | 复制测试资源文件至测试输出目录 |
| test-compile | maven-compiler-plugin:testCompile | 编译测试代码至测试输出目录 |
| test | maven-surefire-plugin:test | 执行测试用例 |
| package | maven-jar-plugin:jar | 创建项目jar包 |
| install | maven-install-plugin:install | 将项目输出构件安装到本地仓库 |
| deploy | maven-deploy-plugin:deploy | 将项目输出构件部署到远程仓库 |

除了内置绑定外，用户能够自己选择将某个插件目标绑定到生命周期的某个阶段上，以便在项目构建过程中执行更丰富的任务。

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-source-plugin</artifactId>  
 <version>2.1.1</version>  
 <executions>  
 <execution>

<id>attach-source</id>

<phase>verify</phase>  
 <goals>  
 <goal>jar-no-fork</goal>  
 </goals>  
 </execution>  
 </executions>  
</plugin>

除了基本的插件坐标配置，<executions>下的每个<execution>用来配置执行一个任务。有时候，即使不配置<phase>阶段，插件目标也能绑定到生命周期中去，这是因为很多插件的目标在编写时已经定义了默认的绑定阶段，可以通过maven-help-plugin查看插件的详细信息：

mvn help:describe –Dplugin=org.apache.maven.plugins:maven-source-plugin:2.1.1 -Ddetail

7.4 插件解析机制

为了方便用户使用和配置插件，Maven不需要用户提供完整的插件坐标信息，就可以解析得到正确的插件。

与依赖构件一样，插件构件同样基于坐标存储在Maven仓库中，但Maven会区别对待依赖的远程仓库与插件的远程仓库。

通过<repositories>及其子元素<repository>可以配置依赖的远程仓库，插件的远程仓库需要使用<pluginRepositories>和<pluginRepository>进行配置。

默认情况下，如果该插件是Maven官方插件，则可以省略groupId(org.apache.maven.plugins),Maven在解析该插件的时候，会自动将groupId补上。

当插件没有添加版本号时，若该插件是核心插件，则在超级POM已经定义了版本号，若不是核心插件，Maven会遍历本地仓库和远程仓库，计算出latest和release的值，Maven 2使用latest，但因为latest可能是快照版本，Maven 3更改为使用release。

8.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，而继承特性则能够帮助抽取各模块相同的依赖和插件等配置，在简化POM的同时，还能促进各个模块配置的一致性。

8.1 聚合

Maven聚合也称多模块，能够一次构建多个模块。聚合模块本身是一个Maven项目，所以也有自己的POM文件，该POM文件的packaging为pom，并且含有<modules>和<module>元素，如：

<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.wangdh</groupId>  
 <artifactId>springboot.demo</artifactId>  
 <version>1.0.0-SNAPSHOT</version>  
 <packaging>pom</packaging>  
 <modules>  
 <module>springboot-mybatis</module>  
 <module>springboot-web</module>  
 <module>springboot-quickstart</module>  
 </modules>  
</project>

这里每个module的值都是一个当前POM的相对目录，一般而言，为了方便快速定位内容，模块所处的目录名称应该与其artifactId一致，不过这不是Maven的要求。

因此，聚合模块与其他模块的目录结构并非一定要父子关系，通过修改module的值，也能更改为平级关系:

<module>../springboot-quickstart</module>  
 Maven首先会解析聚合模块的POM，分析要构建的模块，并计算出一个反应堆构建顺序，然后根据这个顺序依次构建各个模块。反应堆包含了模块之间继承和依赖的关系。模块间的依赖关系会将反应堆构成一个有向非循环图。

8.2 继承

继承解决的是对重复依赖和插件配置的抽取。通过定义一个父模块，将其他模块相同的配置抽离到父模块中，然后继承父模块，并且父模块也是一个Maven项目，其POM文件的packaging为pom。

子模块需要增加<parent>元素配置：

<parent>  
 <artifactId>springboot.demo</artifactId>  
 <groupId>com.wangdh</groupId>  
 <version>1.0.0-SNAPSHOT</version>

<relativePath>../pom.xml</relativePath>  
</parent>

relativePath定义了父模块POM文件的位置。默认值为../pom.xml，Maven默认父POM在上一层目录下。

POM文件可被继承的元素有：

groupId:项目组ID，项目坐标的核心元素

version:项目版本，项目坐标的核心元素

description:项目的描述信息

organization:项目所在组织机构信息

inceptionYear:项目的创始年份

url:项目的URL地址

developers:项目的开发者信息

contributors:项目的贡献者信息

distributionManagement:项目的部署配置

issueManagement:项目的缺陷跟着系统信息

ciManagement:项目的持续集成系统信息

scm:项目的版本控制系统信息

mailingLists:项目的邮件列表信息

properties:自定义的Maven属性

dependencies:项目的依赖配置

dependencyManagement:项目的依赖管理配置

repositories:项目的仓库配置

build:项目的源码目录配置、输出目录配置、插件配置、插件配置管理等

reporting:项目的报告输出目录配置、报告插件配置等

8.3 依赖管理

8.4 聚合与继承

 

对于Maven聚合，聚合模块知道他聚合了哪些模块，但是被聚合模块不知道它自己被谁聚合了；对于Maven继承，子模块知道自己的父模块是谁，但父模块不知道自己有多少子模块。

在实际使用过程中聚合模块和父模块是同一个模块。

8.5约定大于配置

使用约定可以大量减少配置。在Maven中，默认的约定有：

1) 源码路径 src/main/java

2) 编译输出目录 target/classes

3) 打包方式 jar

4) 包输出目录 target

在Maven中，约定的项是能够通过配置项修改的，约定的项也是配置在超级POM中，可以在当前的POM文件，通过修改对应配置进行覆盖。