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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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