Maven使用

1、目录结构

使用 maven 创建的工程我们称为 maven 工程，maven 工程具有一定的目录规范，如下：

src/main/java ------存放项目的 .java 文件

src/main/resources ------存放项目资源的文件，如 spring 配置文件

src/test/java ------存放所有单元测试 .java 文件，如 JUnit 测试类

src/test/resources ------存放测试资源文件

target ------项目输出位置，编译后的 class 文件会输出到此目录

pom.xml ------maven 项目核心配置文件

2、常用命令

1、compile

是 maven 工程的编译命令，作用是将 src/main/java 下的文件编译为 class 文件输出到 target 目录下。

在 cmd 进入命令状态，执行：`mvn compile`

2、test

是 maven 工程的测试命令，会执行 src/test/java 下的单元测试类。

在 cmd 进入命令状态，执行：`mvn test`

3、clean

是 maven 工程的清理命令，执行 clean 会删除 target 目录的内容。

在 cmd 进入命令状态，执行：`mvn clean`

4、package

是 maven 工程的打包命令，对 java 工程执行 package 打成 jar 包，对于 web 工程打成 war 包。

在 cmd 进入命令状态，执行：`mvn package`

5、install

是 maven 工程的安装命令，执行 install 将 maven 打成 jar 包 或 war 包发布到本地仓库。

在 cmd 进入命令状态，执行：`mvn install`

3、生命周期（了解）

构建生命周期阶段的目标是执行顺序是一个良好定义的序列。 一个典型的 Maven 构建生命周期是由下列顺序的阶段：

阶段	处理	描述
准备资源	资源复制	资源复制可以进行定制
编译	执行编译	源代码编译在此阶段完成
包装	打包	创建JAR/WAR包如在 pom.xml 中定义提及的包
安装	安装	这一阶段在本地/远程Maven仓库安装程序包

可用于注册必须执行一个特定的阶段之前或之后的目标，有之前处理和之后阶段。 当 Maven 开始建立一个项目，它通过定义序列阶段步骤和执行注册的每个阶段的目标。 Maven有以下三种标准的生命周期：

- clean：在进行真正的构建之前进行一些清理工作
- default(或 build)：构建的核心部分，编译、测试、打包、部署等
- site：生成项目报告，站点，发布站点。

目标代表一个特定的任务，它有助于项目的建设和管理。可以被绑定到零个或多个生成阶段。一个没有绑定到任何构建阶段的目标，它的构建生命周期可以直接调用执行。 执行的顺序取决于目标和构建阶段折调用顺序。例如，考虑下面的命令。清理和打包（mvn clean）参数的构建阶段，而 dependency:copy-dependencies package 是一个目标。

```
mvn clean dependency:copy-dependencies package
```

在这里，清洁的阶段，将首先执行，然后是依赖关系：复制依赖性的目标将被执行，并终于将执行包阶段。

1、clean 的生命周期

当我们执行命令 mvn clean 命令后，Maven 调用 clean 的生命周期由以下几个阶段组成：

- pre-clean：执行一些需要在 clean 之前完成的工作
- clean：移除所有上一次构造生成的文件
- post-clean：执行一些需要在 clean 之后离开完成的工作

Maven 清洁目标（clean:clean）被绑定清洁干净的生命周期阶段。clean:clean 目标删除 build 目录下的构建输出。因此，当 mvn clean 命令执行时，Maven会删除编译目录。

2、default(或 build) 的生命周期

这是 Maven 主要的生命周期，用于构建应用程序。它有以下 23 个阶段。

生命周期阶段	描述
validate	验证项目是否正确，并且所有必要的信息可用于完成构建过程
initialize	建立初始化状态，例如设置属性
generate-sources	产生任何的源代码包含在编译阶段
process-sources	处理源代码，例如，过滤器值
generate-resources	包含在包中产生的资源
process-resources	复制和处理资源到目标目录，准备打包阶段
compile	编译该项目的源代码
process-classes	从编译生成的文件提交处理，例如：Java类的字节码增强/优化
generate-test-sources	生成任何测试的源代码包含在编译阶段
process-test-sources	处理测试源代码，例如，过滤器任何值
test-compile	编译测试源代码到测试目标目录
process-test-classes	处理测试代码文件编译生成的文件
test	运行测试使用合适的单元测试框架（JUnit）
prepare-package	执行必要的任何操作的实际打包之前准备一个包
package	提取编译后的代码，并在其分发格式打包，如JAR，WAR或EAR文件
pre-integration-test	完成执行集成测试之前所需操作。例如，设置所需的环境
integration-test	处理并在必要时部署软件包到集成测试可以运行的环境
pre-integration-test	完成集成测试已全部执行后所需操作。例如，清理环境
verify	运行任何检查，验证包是有效的，符合质量审核规定
install	将包安装到本地存储库，它可以用作当地其他项目的依赖
deploy	复制最终的包到远程仓库与其他开发者和项目共享

有涉及到Maven 生命周期值得一提几个重要概念：

- 当一个阶段是通过 Maven命令调用，例如：mvn compile，只有阶段到达并包括这个阶段才会被执行。
- 不同的 Maven 目标绑定到 Maven生命周期的不同阶段这是这取决于包类型(JAR/WAR/EAR)。

3、site 的生命周期

Maven的 site 插件通常用于创建新的文档，创建报告，部署网站等。

阶段

- pre-site：执行一些需要在生成站点文档之前完成的工作
- site：生成项目的站点文档
- post-site：执行一些需要在生成站点文档之后完成的工作，并且为部署做准备
- site-deploy：将生成的站点文档部署到特定的服务器上

4、定义 maven 坐标

每个 maven 工程都需要定义本工程都坐标，坐标是 maven 对 jar 包的身份定义。

```
<!-- 项目名称，定义为组织名+项目名，类似包名 -->
<groupId>com.rimi</groupId>
<!-- 模块名称 -->
<artifactId>maven</artifactId>
<!-- 当前项目版本号，snapshot 为快照版本即非正式版本，release 为正式发布版本 -->
<version>1.0-SNAPSHOT</version>
<!-- 打包类型：
    jar：执行 package 会打包成 jar 包
    war：执行 package 会打包成 war 包
    pom：用于 maven 工程的继承，通常父工程设置为 pom
     -->
<packaging>jar</packaging>
```

5、构建 web 工程

1、需求

创建一个 web 工程，实现入门程序的功能。

1. 添加 index.jsp，输出 hello world
2. 添加一个servlet 转发到 jsp 页面

2、创建 maven工程

3、定义坐标

4、设置编译版本

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
                <encoding>UTF-8</encoding>
            </configuration>
        </plugin>
    </plugins>
</build>
```

5、编写 servlet

```
public class ServletTest extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        request.getRequestDispatcher("/jsp/test.jsp").forward(request,
response);
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doPost(request, response);
    }
}
```

6、编写 jsp

7、添加 jar

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
  <version>2.5</version>
    <scope>provided</scope>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.0</version>
    <scope>provided</scope>
</dependency>
```

8、配置 servlet

9、运行

在 cmd 命令行进入工程目录，运行 `tomcat:run` 命令

6、依赖管理-添加依赖

1、依赖范围

1、基本概念

当一个库 A 依赖于其他库 B，另一工程 C 想要使用库 A，那么该工程同样也需要使用到库 B。

Maven 可以避免去搜索所有需要的库资源的这种需求。通过读取工程文件（pom.xml）中的依赖项，Maven 可以找出工程之间的依赖关系。

我们只需要在每个工程的 pom 文件里去定义直接的依赖关系。Maven 则会自动的来接管后续的工作。

通过传递依赖，所有被包含的库的图形可能会快速的增长。当重复的库存在时，可能出现的情形将会持续上升。Maven 提供一些功能来控制可传递的依赖的程度。依赖范围包括：

- compile：编译范围，值 A 在编译时依赖 B，此范围为默认依赖范围。编译范围的依赖会用在编译、测试、运行，由于运行时需要所以编译范围的依赖会被打包。
- provided：provided 依赖只有在当 JDK 或者一个容器已提供该依赖之后才使用，provided 依赖在编译和测试时需要，在运行时不需要，比如：servlet api 被 tomcat 容器提供。
- runtime：runtime 依赖在运行和测试系统的时候需要，但在编译的时候不需要，比如：jdbc 的驱动包。由于运行时需要所以 runtime 范围依赖会被打包。
- test：test 范围在编译和运行时都不需要，他们只有在测试编译和测试运行阶段可用，比如：junit。由于运行时不需要所以 test 范围依赖不会被打包。
- system：system 范围依赖与 provided 类似，但是你必须显示的提供一个对于本系统中 jar 文件的路径，需要指定 systemPath 磁盘路径，system 依赖不推荐使用。
- import (only available in Maven 2.0.9 or later)：import 范围依赖只能用在类型为 pom 的依赖项 `<dependencyManagement>` 中，它表示在指定的 POM 的 `<dependencyManagement>` 中，依赖项被替换为有效的依赖项列表，从其它的pom文件中导入依赖设置。

每个作用域(除了导入)都以不同的方式影响传递依赖关系，如下表所示。如果一个依赖项被设置为左列的范围，那么依赖项的传递依赖关系将会导致在主项目中与在交集中列出的范围相关联。如果没有列出范围，则意味着依赖项将被省略。

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
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC驱动
test	-	Y	-	Junit
system	Y	Y	-	本地的maven仓库之外的类库

7、使用私服

1、从私服上下载

打开maven 的安装目录， 编辑 settings.xml

```
<!-- 配置镜像 -->
<mirrors>
  <mirror>
    <id>nexus</id>
    <mirrorOf>*</mirrorOf>
    <url>http://10.2.1.115:8081/repository/maven-public/</url>
  </mirror>
</mirrors>

<!-- 设置中央工厂 -->
<profiles>
  <profile>
        <id>nexus</id>
    <repositories>
      <repository>
        <id>central</id>
        <url>http://central</url>
        <releases>
                    <enabled>true</enabled>
                </releases>
        <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </repository>
    </repositories>
```

```
    <pluginRepositories>
      <pluginRepository>
        <id>central</id>
        <url>http://central</url>
        <releases>
                    <enabled>true</enabled>
                </releases>
        <snapshots>
                    <enabled>true</enabled>
                </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>

<!-- 激活设置 -->
<activeProfiles>
  <activeProfile>nexus</activeProfile>
</activeProfiles>
```

2、提交到私服

打开maven 的安装目录， 编辑 settings.xml

```
<!-- 私服认证 -->
<servers>
  <server>
    <id>nexus</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
</servers>
```

在 pom 文件中指定要上传私仓的名称

```
<distributionManagement>
  <repository>
    <id>nexus</id>
    <name>Releases</name>
    <url>http://10.2.1.115:8081/repository/maven-releases/</url>
  </repository>
  <snapshotRepository>
    <id>nexus</id>
    <name>Snapshot</name>
    <url>http://10.2.1.115:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```