Maven 学习

Maven 简介

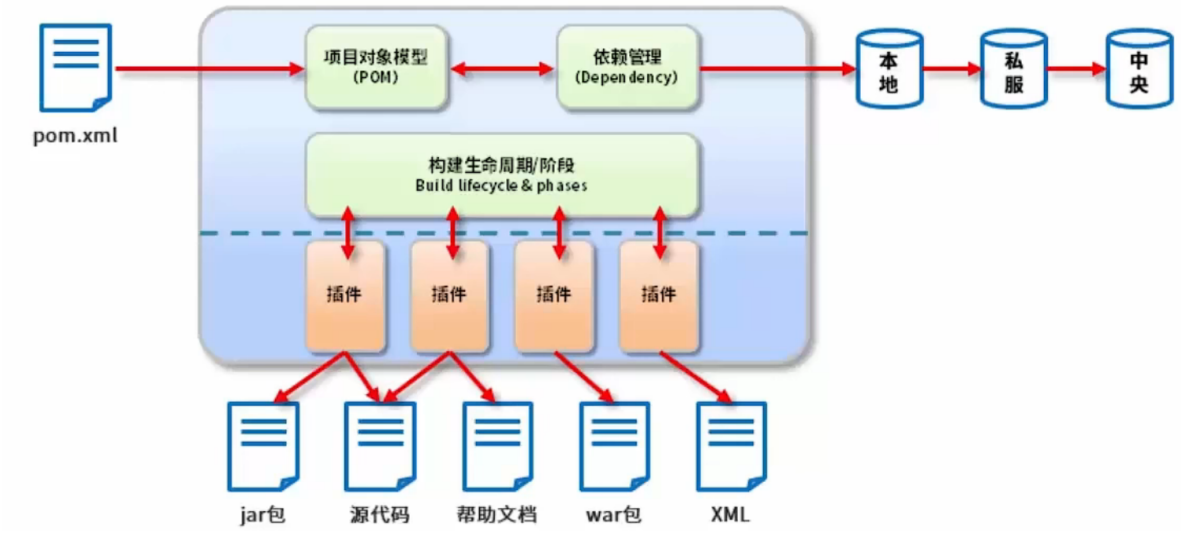
Maven 是什么?

传统项目管理状态分析

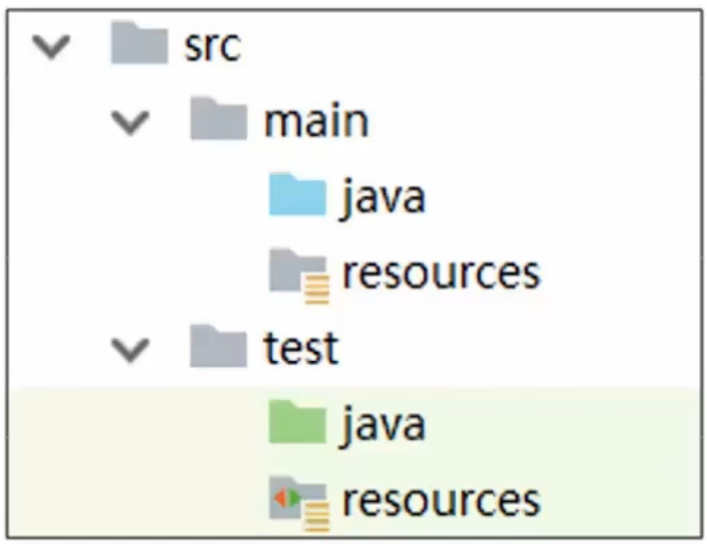
- jar 包不统一, jar 包不兼容
- 工程升级维护过程操作繁琐

Maven 是什么?

- Maven 本质是一个项目管理工具, 将项目开发和管理过程抽象成一个对象模型(POM);
- POM(Project Object Model) : 项目对象模型;



Maven 的作用?

- 项目构建: 提供标准的, 跨平台的自动化项目构建方式;
- 依赖管理: 方便快捷的管理项目的资源(Jar包), 避免资源间的版本冲突问题.
- 统一开发结构: 提供标准的, 统一的项目结构.
  - 

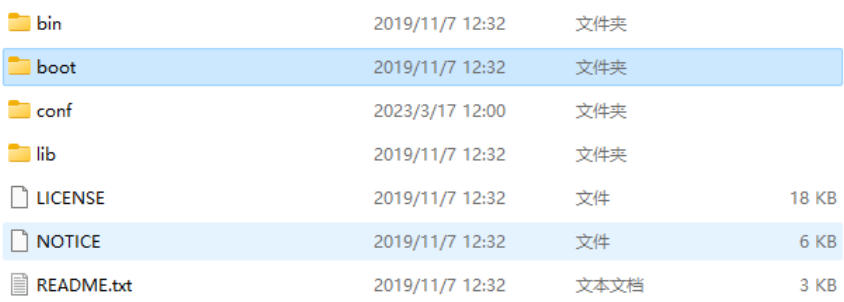
下载与安装

- 官网: https://maven.apache.org/
- 下载地址: https://maven.apache.org/download.cgi

安装

> 下载后解压可以用

目录结构



bin 目录

Mavne 中所有的可运行指令.

boot 目录

一些启动项, 里面主要就是Maven的类加载器.

conf 目录

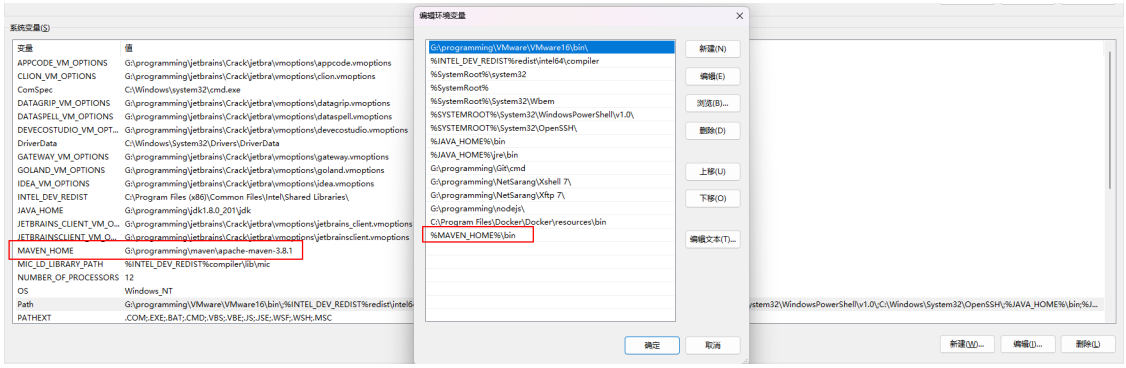
里面主要就是 Maven 的配置文件和日志文件. 其中最主要的配置文件就是settings.xml.

settings 常见配置项

1. localRepository：指定本地仓库的路径。本地仓库是Maven用来存储项目依赖的地方。默认情况下，本地仓库位于用户主目录下的.m2文件夹中。
2. mirrors：用于配置镜像仓库。镜像仓库是指在下载依赖时，从镜像仓库获取而不是从中央仓库获取。可以配置多个镜像仓库，每个镜像仓库包含id、url和mirrorOf等属性。
3. proxies：用于配置代理服务器。如果你在使用Maven时需要通过代理服务器访问外部资源，可以在这里配置代理服务器的相关信息，如主机名、端口号、用户名和密码等。
4. servers：用于配置服务器凭据。如果你需要访问需要身份验证的远程仓库或者发布到远程仓库，可以在这里配置服务器的凭据，如id、用户名和密码等。
5. profiles：用于配置Maven的配置文件激活条件。可以根据不同的环境或需求，定义不同的profile，并在这里配置profile的激活条件，如激活的操作系统、Java版本等。
6. pluginGroups：用于配置插件组。插件组是一组相关的插件，可以在pom.xml文件中直接使用其插件ID而无需指定插件的完整坐标。

Maven 环境变量配置

- 依赖 Java, 需要配置 JAVA_HOME
- 设置 Maven 自身的运行环境, 需要配置 MAVEN_HOME

- 

Maven 基础概念

仓库

> 用于存储资源, 包含各种 jar 包.

- **仓库分类**:
  - **本地仓库**: 自己电脑上存储资源的仓库, 连接远程仓库获取资源.
  - **远程仓库**: 非本机电脑上的仓库, 为本地仓库提供资源.
    - **中央仓库**: Maven团队维护, 存储所有资源的仓库.
    - **私服仓库**: 部门/公司范围内存储的仓库, 从中央仓库获取资源.
  - **私服的作用**:
    - 保存具有版权的资源, 包含购买或者自主研发的 jar
      - 中央仓库中的 jar 都是开源的, 不能存出具有版权的资源.
    - 一定范围内共享资源, 仅对内部开发, 不对外共享.

坐标

什么是坐标?

> Maven 中的坐标用于描述仓库中资源的位置.

Mavne 坐标的主要组成

- groupId(组织id): 定义当前 Maven 项目隶属于组织的名称(通常域名反写)
- artifactId(项目id): 定义当前 Maven 项目名称,(通常是模块名称)

- version(版本号): 定义当前项目的版本号
- packaging: 定义该项目的打包方式.

Maven 坐标的作用

使用唯一标识, 唯一性定位资源位置, 通过该标识可以将资源的识别与下载工作交由机器完成.

Maven 项目构建命令

Maven 常用命令

- Maven 构建命令使用 mvn 开头, 后面添加功能参数, 可以一次执行多个命令, 使用空格分隔.

```
mvn compile          # 编译
mvn clean            # 清理
mvn test             # 测试
mvn package          # 打包
mvn install          # 安装到本地仓库

mvn clean：清理项目，删除生成的目标文件和构建产物。
mvn compile：编译项目的源代码。
mvn test：运行项目的单元测试。
mvn package：将项目打包为可分发的格式，如JAR、WAR或EAR。
mvn install：将项目构建产物安装到本地Maven仓库，以供其他项目使用。
mvn deploy：将项目构建产物部署到远程Maven仓库，以供其他开发人员或团队使用。
mvn clean install：清理项目并将构建产物安装到本地Maven仓库。
mvn clean package：清理项目并将项目打包为可分发的格式。
mvn clean test：清理项目并运行项目的单元测试。
mvn clean compile：清理项目并编译项目的源代码。
```

插件创建工程

- 创建工程

```
mvn archetype:generate
    -DgroupId={project-packaging}
    -DartifactId={project-name}
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
```

- 创建 java 工程

```
mvn archetype:generate
    -DgroupId=cn.luxh.app
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-quickstart
    -Dversion=0.0.1-snapshot
    -DinteractiveMode=false
```

- 创建 web 工程

```
mvn archetype:generate
    -DgroupId=cn.luxh.app
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-webapp
    -Dversion=0.0.1-snapshot
    -DinteractiveMode=false
```

依赖管理

依赖配置

- 依赖指当前项目运行所需的 jar,一个项目可以设置多个依赖
- 格式:

```
<!-- 设置当前项目所依赖的所有 jar -->
<dependencies>
    <!-- 设置具体依赖 -->
    <dependency>
        <!-- 依赖所属群组 id -->
        <groupId>junit</groupId>
        <!-- 依赖所属项目 id -->
        <artifactId>junit</artifactId>
        <!-- 依赖版本号 -->
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

依赖传递

依赖具有传递性

- 直接依赖: 在当前项目中通过依赖配置建立的依赖关系.
- 间接依赖: 被资源的资源如果依赖其他资源, 当前项目间接依赖其他资源.
-

依赖传递冲突问题

- 路径优先: 当依赖中出现相同的资源时, 层级越深, 优先级越低, 层级越浅, 优先级越高.
- 声明优先: 当资源在相同层级被依赖时, 配置顺序靠前的覆盖配置顺序在后的.
- 特殊优先: 当同级配置了相同资源的不同版本, 后配置的覆盖先配置的.

可选依赖

- 对外隐藏当前所依赖的资源--不透明

- true 默认为 false 不隐藏, 为 true 则对外隐藏.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
        <optional>true</optional>
    </dependency>
</dependencies>
```

排除依赖

- 主动断开依赖的资源, 被排除的资源无需指定版本 -- 不需要
- exclusions

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
        <optional>true</optional>
        <exclusions>
            <exclusion>
                <groupId>org.hamcrest</groupId>
                <artifactId>hamcrest-core</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

依赖范围

- 依赖的 jar 默认情况可以在任何地方使用, 可以通过 scope 标签设定其作用范围.
- 作用范围:
  - 主程序范围有效 (main 文件夹范围内)
  - 测试程序范围有效 (test 文件夹范围内)
  - 是否参与打包 (package 指令范围内)
- compile（默认）：这是默认的依赖范围。compile作用域的依赖项在所有情况下都有效，包括编译、运行和测试。
- provided：这个依赖范围表示该依赖项由JDK或运行容器在运行时提供。也就是说，该依赖项在测试和编译阶段需要，但在运行时由容器提供。
- runtime：这个依赖范围表示该依赖项在运行时需要，但在编译阶段不需要。在测试阶段，该依赖项也会被使用。
- test：这个依赖范围表示该依赖项只在测试阶段有用，在编译和运行阶段不会被使用。
- system：这个依赖范围表示该依赖项是由我们提供的，不需要从Maven仓库中获取。使用该范围时，需要与systemPath标签配合使用，指定该依赖项在系统中的位置。
- import：这个依赖范围用于在子项目中引入父项目的dependencyManagement中定义的依赖项。它允许子项目继承父项目的依赖项，而不需要直接继承父项目。

依赖范围传递

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

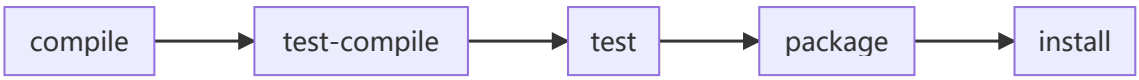
直接依赖

间接依赖

生命周期与插件

构建生命周期

> mavne 构建生命周期描述的试一次构建过程经历了多少事件.



- maven 对象吗构建的生命周期划分为 3 套
  - clean: 清理工作
  - default: 核心工作, 例如: 编译, 测试, 打包, 部署.
  - site: 产生报告, 发布站点等,

clean 生命周期

- pre-clean　　执行一些需要在 clean 之前完成的工作.
- clean　　移除所有上一次构建生成的文件.
- post-clean　　执行一些需要在 clean 之后立刻完成的工作.

default 生命周期

1. validate：验证项目的正确性，检查项目是否正确配置。
2. initialize：初始化构建环境，设置构建相关的属性。
3. generate-sources：生成源代码，例如使用Annotation Processor生成的代码。
4. process-sources：处理源代码，例如编译源代码。
5. generate-resources：生成资源文件，例如复制资源文件到目标目录。
6. process-resources：处理资源文件，例如过滤资源文件。
7. compile：编译源代码。
8. process-classes：处理编译后的类文件，例如生成额外的文件。
9. generate-test-sources：生成测试源代码。
10. process-test-sources：处理测试源代码。
11. generate-test-resources：生成测试资源文件。
12. process-test-resources：处理测试资源文件。
13. test-compile：编译测试源代码。
14. process-test-classes：处理测试编译后的类文件。
15. test：运行测试。
16. prepare-package：准备打包，例如生成额外的文件。

17. package：打包，将编译后的代码打包成可发布的格式，例如JAR或WAR文件。
18. pre-integration-test：在集成测试之前执行的操作。
19. integration-test：执行集成测试。
20. post-integration-test：在集成测试之后执行的操作。
21. verify：验证打包的正确性，例如运行额外的检查。
22. install：安装到本地仓库，供其他项目使用。
23. deploy：部署到远程仓库，供其他开发人员使用。

Site 生命周期

- 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

插件

- 插件与生命周期内的阶段绑定, 在执行到对应生命周期时执行对应的插件功能.
- 默认 Maven 在各个生命周期上绑定有预设的功能.
- 通过插件可以自定义其它功能.

分模块开发与设计

工程模块与模块划分

在aven项目中，模块是指将一个大型项目拆分成多个独立的子项目，每个子项目都有自己的pom.xml文件。这种模块化的划分可以带来许多好处，包括更好的代码组织、更容易的维护和测试、更好的团队协作等。

在划分Maven项目模块时，可以根据功能、业务领域或技术层面进行划分。以下是一些常见的模块划分方式：

1. 功能划分：根据不同的功能将项目划分为多个模块，例如用户管理模块、订单管理模块、支付模块等。这种划分方式可以使每个模块专注于特定的功能，便于团队成员的协作和独立开发。
2. 业务领域划分：根据不同的业务领域将项目划分为多个模块，例如电商模块、金融模块、物流模块等。这种划分方式可以使每个模块独立演化，便于团队成员的专注和理解。
3. 技术层面划分：根据不同的技术层面将项目划分为多个模块，例如前端模块、后端模块、数据库模块等。这种划分方式可以使每个模块专注于特定的技术领域，便于团队成员的专长发挥和独立开发。

在划分模块时，需要考虑模块之间的依赖关系和版本管理。可以使用Maven的父子模块关系来管理模块之间的依赖关系，并在父模块的pom.xml中定义共享的依赖和插件版本。这样可以确保模块之间的一致性和版本控制。

聚合

作用

> 聚合用于快速构建 maven 工程, 一次性构建多个项目/模块

制作方式

- 创建一个父模块, 打包类型定义为 pom

```
<packaging>pom</packaging>
```

- 定义当前模块进行构建操作时关联的其他模块名称

```
<modules>
    <module>子模块的名称1</module>
    <module>子模块的名称2</module>
    <module>子模块的名称3</module>
    <module>子模块的名称4</module>
</modules>
```

注意事项

参与聚合操作的模块最终执行顺序与模块间的依赖关系有关, 与配置顺序无关.

继承

描述

在Maven中，可以使用继承来实现项目之间的代码和配置的共享。Maven继承是通过在父项目中定义一组通用的配置和依赖，然后在子项目中继承这些配置和依赖来实现的。

要使用Maven继承，需要创建一个父项目（也称为聚合项目）和一个或多个子项目。父项目中定义了一些通用的配置，例如插件版本、依赖管理等。子项目可以继承父项目的配置，并可以添加自己的特定配置。

在父项目的pom.xml文件中，使用元素列出所有的子项目。子项目的pom.xml文件中使用元素指定父项目的坐标（groupId、artifactId和version）。

实现

父项目的pom.xml：

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>parent-project</artifactId>
  <version>1.0.0</version>

  <modules>
    <module>child-project1</module>
    <module>child-project2</module>
  </modules>

  <!-- 其他通用配置 -->
</project>
```

子项目的pom.xml：

```
<project>
  <parent>
    <groupId>com.example</groupId>
    <artifactId>parent-project</artifactId>
    <version>1.0.0</version>
```

```
  </parent>

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>child-project1</artifactId>
  <version>1.0.0</version>

  <!-- 子项目特定配置 -->
</project>
```

继承依赖定义

描述

在Maven中，可以使用继承来定义依赖关系。通过在父项目的pom.xml文件中定义依赖，子项目可以继承这些依赖，从而实现共享和统一管理。

在父项目的pom.xml文件中，使用元素来定义依赖管理。在该元素下，可以使用元素来列出所有需要管理的依赖项。每个依赖项使用元素进行定义，包括groupId、artifactId和version等信息。

子项目可以通过在其pom.xml文件中使用元素来继承父项目的依赖。当子项目中引用一个依赖时，Maven会首先查找子项目自身的依赖定义，如果找不到，则会继续查找父项目的依赖定义。

这种继承依赖定义的方式可以简化项目的管理，避免重复定义相同的依赖，并确保所有子项目使用相同的依赖版本。同时，通过在父项目中集中管理依赖，可以更方便地进行版本升级和依赖冲突的解决。

案例展示

以下是一个示例的父项目pom.xml文件中的dependencyManagement定义：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>dependency1</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>com.example</groupId>
      <artifactId>dependency2</artifactId>
      <version>2.0.0</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

子项目可以通过以下方式继承父项目的依赖：

```
<dependencies>
  <dependency>
    <groupId>com.example</groupId>
    <artifactId>dependency1</artifactId>
  </dependency>
  <dependency>
    <groupId>com.example</groupId>
    <artifactId>dependency2</artifactId>
  </dependency>
</dependencies>
```

继承的资源

- groupId：项目组ID，项目坐标的核心元素
- version：项目版本，项目坐标的核心因素
- description：项目的描述信息
- organization：项目的组织信息
- 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：项目的缺陷跟踪系统信息
- ciManagement：项目的持续集成系统信息
- scm：项目的版本控制系统西溪
- malilingLists：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：包括项目的源码目录配置、输出目录配置、插件配置、插件管理配置等
- reporting：包括项目的报告输出目录配置、报告插件配置等

属性

自定义属性

作用

等同于定义变量, 方便统一维护.

定义格式

```
<properties>
    <spring.version>5.1.9</spring.version>
</properties>
```

调用格式

```
<dependency>
    <version>${spring.version}</version>
</dependency>
```

内置属性

作用

使用 Maven 内置属性, 快速配置.

调用格式

```
${basedir}
${version}
```

Setting 属性

作用

使用 Maven 配置文件 setting.xml 中的标签属性, 用于动态配置.

调用格式

```
${setting.localRepository}
```

Java 系统属性

作用

读取 Java 属性.

调用格式

```
${user.home}
```

系统属性查询方式

```
mvn help:system
```

环境变量属性

作用

使用 Maven 配置文件 setting.xml 中的标签属性, 用于动态配置.

调用格式

```
${env.JAVA_HOME}
```

环境变量属性查询方式

```
mvn help:system
```

版本管理

简介

Maven是一个流行的构建工具，它提供了版本管理的功能。在Maven中，你可以使用版本号来管理你的项目的不同版本。Maven的版本号遵循一定的规则，通常采用三段式的形式：

- 主版本号 : 表示项目重大架构的变更, 如: spring5相较于spring4的迭代
- 次版本号 : 表示有较大的功能增加和变化, 或者全面系统地修复漏洞
- 增量版本 : 表示有重大漏洞的修复
- 里程碑版本 : 表示一个版本的里程碑(版本内部)

例如，1.0.0是一个常见的版本号格式。在Maven中，你可以在项目的pom.xml文件中指定版本号。你可以使用属性来定义版本号，以便在构建过程中灵活地修改它。

工程版本

- SNAPSHOT(快照版本)
  - 项目开发过程中, 为了方便团队成员合作, 解决模块间相互依赖和时间啊更新的问题, 开发者对每个模块进行构建的时候, 输出的临时性版本叫快照版本(测试阶段版本)
  - 快照版本会随着开发的进展不断更新
- RELEASE(发布版本)
  - 项目开发到进入阶段里程碑后, 向团队外部发布较为稳定的版本, 和这个版本所对应的构建文件是稳定的. 即便进行功能的后续开发, 也不会改变当前发布版本内容, 这种版本称之为发布版本.

资源配置

多环境开发配置

配置方式

```
<!--创建多环境-->
<profiles>
    <!-- 定义具体环境:开发 -->
    <profile>
        <!--定义环境对应的唯一名称-->
        <id>dev</id>
        <!--设置默认启动-->
        <activation>
            <!--默认激活配置-->
            <activeByDefault>true</activeByDefault>
        </activation>
        <!--定义环境中专用的属性值-->
        <properties>
            <!--当前环境-->
            <profile.name>dev</profile.name>
        </properties>
    </profile>
    <!-- 内网测试 -->
    <profile>
        <id>sit</id>
        <properties>
            <!--当前环境-->
            <profile.name>sit</profile.name>
        </properties>
    </profile>
    <!-- 生产 -->
    <profile>
        <id>prod</id>
        <properties>
            <!--当前环境-->
            <profile.name>prod</profile.name>
        </properties>
    </profile>
</profiles>
```

加载指定环境

调用格式

```
mvn 指令 -P 环境定义id
```

范例

```
mvn install -P prod
```

跳过测试

在Maven中，您可以使用-DskipTests参数来跳过测试阶段。当您运行Maven命令时，可以在命令行中添加该参数，如下所示：

```
mvn clean install -DskipTests
```

上述命令将执行Maven的clean和install目标，并跳过测试阶段。这意味着在构建过程中不会执行任何测试。

另外，如果您只想跳过特定的测试类或测试方法，而不是完全跳过测试阶段，您可以使用maven-surefire-plugin插件的skip或excludes配置。在项目的pom.xml文件中，您可以添加以下配置：

```
    <build>
        <plugins>
            <plugin>
               <groupId>org.apache.maven.plugins</groupId>
               <artifactId>maven-surefire-plugin</artifactId>
               <configuration>
                    <skip>true</skip>
                    <!-- 或者使用以下配置来排除特定的测试类或测试方法 -->
                    <!--
                    <excludes>
                        <exclude>com.example.MyTest</exclude>
                    </excludes>
                    -->
                    <!-- 或者使用以下配置来包含特定的测试类或测试方法 -->
                    <!--
                    <includes>
                        <include>com.example.MyTest</include>
                    </includes>
                    -->
                </configuration>
            </plugin>
        </plugins>
    </build>
```

上述配置使maven-surefire-plugin插件跳过所有测试。如果您想要排除特定的测试类或测试方法，您可以使用配置，并指定要排除的类或方法的全限定名。

私服

Nexus

- Nexus 是 Sonatype 公司的一款 maven 私服产品
- 下载地址： https://www.sonatype.com/download-oss-sonatype

仓库分类

1）hosted：宿主仓库，即本地仓库，该仓库存放本地项目产生的构建，无论是团队内部开发了通用组件库、公共 jar 等，都是发布到这里面。

2）proxy：代理仓库，用来代理远程仓库，如代理 Maven 中央仓库等。

3）group：仓库组，可以聚合上面两者。因为在开发过程中，某些包是远端的、某些包是内部私服中的，这样就对应了两个地址，使用仓库组将 hosted 和 group 聚合，暴露为一个地址。

访问私服配置

- 配置当前项目访问私服上传资源的保存位置（pom.xml）

```
<distributionManagement>
    <repository>
        <id>test</id>
        <url>https://localhost:8081</url>
    </repository>
    <snapshotRepository>
        <id>snapshorts</id>
        <url>https://localhost:8081</url>
    </snapshotRepository>
</distributionManagement>
```

- 发布资源到私服命令

```
mvn deploy
```