

# 基于Maven的包依赖管理与仓库配置实验

## 实验原理

### Maven基本命令

Maven运行命令格式如下:

```
mvn [options] [<goal(s)>] [<phase(s)>]
```

使用`mvn -h`可以查看到Maven所以可选项. Maven三个标准生命周期常用的阶段包括:

- `clean` - `clean`
- `default` - `validate`, `compile`, `test`, `package`, `verify`, `install`, `deploy`
- `site` - `site`, `site-deploy`

例如, 打包项目、生成所有文件站点信息并将其部署至仓库的命令为:

```
mvn clean deploy site-deploy
```

常用IAR打包命令为:

```
mvn clean package -Dmaven.test.skip=true # 跳过测试打包
mvn clean install -Dmaven.test.skip=true # 跳过测试打包, 并把打好的包上传到本地仓库
mvn clean deploy -Dmaven.test.skip=true # 跳过测试打包, 并把打好的包上传到远程仓库
```

Maven是一款项目管理工具, 通过POM(Project Object Model)文件来定义项目的结构、依赖和构建过程. 实验涉及两个方面的内容:

1. 包依赖管理: Maven使用POM文件中的`<dependencies>`元素来管理项目的依赖关系. 通过配置依赖项, Maven能够自动下载并集成项目所需的第三方库.
2. 仓库配置: Maven仓库是用来存储和管理构建过程中所需的库文件的地方. 实验中将学习如何配置Maven仓库, 以便Maven能够正确地下载项目所需的依赖.

在正式开始试验之前, 需要先安装Maven工具.

## Maven Archetype

Archetype是Maven工程的模板工具包, 定义了要做的一类项目的基本模型或者架构. 通过`mvn archetype:generate`命令, 开发人员可以很方便地将一类项目的最佳实现 应用 到自己的项目中. 在Maven项目中, 开发者可以通过archetype提供的范例快速入门并了解该项目的结构与特点. 一些常见的Archetype如下:

- `maven-archetype-quickstart`: 默认的archetype, Maven工程项目的样例原型;
- `maven-archetype-j2ee-simple`: 简单的J2EE应用程序样例;

- `maven-archetype-mojo`: Maven 插件项目的示例样例;
- `maven-archetype-webapp`: Maven 的 Webapp 工程样例.

`mvn archetype:generate` 命令执行示例如下:

```
mvn archetype:generate \  
  -DgroupId=cc.synx \  
  -DartifactId=my-project \  
  -DarchetypeArtifactId=maven-archetype-quickstart \  
  -DinteractiveMode=false
```

值得一提的是, 与 Spring Boot 项目框架最相近的 Archetype 是 `maven-archetype-webapp`, Spring Initializr(<https://start.spring.io/>) 执行与 Archetype 相同的操作, 而且对用户更加友好, 因此推荐大家直接使用 Spring Initializr 来创建.

## Setting.xml

`setting.xml` 文件是 Maven 的配置文件之一, 它用于配置 Maven 的全局设置, 包括仓库、代码、镜像、插件等. 这个文件位于 Maven 的安装目录下的 `conf` 文件夹中, 或者在用户的 Maven 主目录下的 `.m2` 文件夹中. 该文件是可选的, 如果用户没有 Maven 中指定该文件的位置, Maven 会使用默认的设置, 以下是 `setting.xml` 中一些常见的配置选项:

- `<localRepository/>`: 指定本地仓库的路径. 默认情况下, Maven 会在用户主目录下的 `.m2` 文件夹中创建一个 `repository` 文件夹来存储下载的依赖.
- `<mirrors/>`: 用于配置镜像, 可以提高构建速度并减轻官方仓库的负载. 镜像可以指向其他 Maven 仓库, 以加速依赖的下载.
- `<proxies/>`: 用于配置代理, 网络环境需要使用代理来连接 Maven 中央仓库或其他远程仓库时, 可以在这里配置代理信息.
- `<profiles/>`: 定义一系列的配置文件激活条件, 比如根据环境、操作系统等等条件来加载不同的配置文件.
- `<servers/>`: 用于配置远程仓库的身份验证信息, 包括用户名、密码等. 这些信息在访问受保护的仓库时需要用到.

## Maven 私有库

### 软件组件

在软件工程中, "软件组件"通常指的是构成软件系统的独立并且可以重用的模块或部分. 这些组件可以是库、框架、模块、插件等, 用于实现特定的功能或者服务.

- 库(library): 一个库是一组已经编写好并被打包成可重用形式的代码, 提供一组特定的功能. 开发者可以再起英语程序中使用这些库, 而不必重新实现相同的功能.
- 框架(framework): 框架是一个更大范围的组件, 通常包含一组库、工具和标准, 用于帮助开发者构建特定类型的应用程序. 框架提供了应用程序的基本结构, 开发者通过扩展或配置框架来实现具体的业务逻辑.
- 模块(module): 模块是软件系统中的一个独立单元, 具有特定的功能. 模块可以包含代码、数据、配置等等, 并且通常设计为可独立开发和测试的单元.
- 插件(plugin): 插件是一种可动态加载到应用程序中以扩展器功能的组件. 插件通常是独立开发的, 可以根据需要添加到主应用程序中.

## Maven仓库管理工具

Maven 仓库管理工具主要用于管理 Maven 仓库中的各种构件(如 JAR 文件、WAR文件等)和元数据。以下是一些常见的 Maven 仓库管理工具:

- **Nexus Repository Manager**: Nexus 是由 Sonatype 提供的强大的仓库管理和仓库代理工具。它支持 Maven、Docker、npm、NuGet 等多种仓库格式。Nexus 有开源版本 (Nexus Repository OSS) 和专业版 (Nexus Repository Pro) 。截至 2024 年 1 月, Nexus 运行环境**仅支持 JDK 8**。
- **JFrog Artifactory**: Artifactory 是由 JFrog 提供的仓库管理工具, 支持 Maven、Gradle、Ivy、Docker、npm、NuGet 等多种仓库格式。Artifactory 有开源版 (Artifactory OSS) 和付费版本 (Artifactory Pro、Artifactory Enterprise) 。
- **Apache Archiva**: Archiva 是 Apache 基金会提供的开源 Maven 仓库管理工具。它支持 Maven 仓库的基本功能, 适用于小型项目和团队。
- **JitPack**: JitPack 是一个基于 Git 的仓库管理服务, 它可以将 GitHub 上的项目直接转换为 Maven 仓库。你可以通过 JitPack 将 GitHub 项目发布到 Maven 仓库, 以便其他项目可以依赖这些构建产物。

## Sonatype Nexus

Sonatype Nexus 是一个用于管理和组织软件构件(例如 JAR 包、WAR 包等)的开源仓库管理系统。它是一个用于构建、部署和管理软件组件的仓库管理器, 主要用于帮助团队有效地共享和管理软件构件。Sonatype Nexus 中的一些常见术语包括:

- **Repository (仓库)**: 仓库是 Nexus 存储软件/组件的地方。Nexus 支持多种类型的仓库, 包括 Maven、npm、NuGet、Docker 等。
- **Proxy Repository (代理仓库)**: 代理仓库是一个与远程仓库关联的本地缓存。当从远程仓库下载构件时, 代理仓库会缓存这些构件, 以便在将来的构建中快速访问。
- **Group Repository (组合仓库)**: 组合仓库允许将多个仓库组合在一起, 并将它们看作一个单独的仓库。这使得在构建中引用多个仓库变得更加简单, 同时也提供了对不同类型仓库的统一访问。
- **Lifecycle Management (生命周期管理)**: Nexus 提供了丰富的生命周期管理功能, 包括对构件的版本控制、审计、发布、存储等管理。

## 雪花算法

雪花算法(Snowflake Algorithm)是由Twitter开源的64位分布式ID生成算法。雪花算法的名称来源于其生成的唯一标识的形状, 看起来像雪花的晶体结构。雪花算法的核心思想是利用一个64位的整数, 其中包含了时间戳、数据中心的标识、机器的标识和一些序列号, 以保证在分布式系统中生成的每个ID都是唯一的。

- **符号位**: 1位, 通常为0。
- **时间戳**: 41位, 毫秒级时间戳, 通过时间戳来保证 ID 的递增性。
- **机器ID**: 10位, 存储机器码, 5位Data Center ID + 5位Worker ID, 最多可标识集群内1024台机器。
- **序列号**: 12位, 单台机器每毫秒可生成的 ID 数。当同一毫秒内生成的 ID 数量超过4096, 会等待下一毫秒再生成。

雪花算法的优点是简单且高效, 能够在分布式环境中生成唯一的ID。然而, 雪花算法依赖于系统的时钟, 如果系统时钟不稳定或发生回拨, 可能会导致 ID 生成重复或不连续。在实际应用中, 需要根据系统的特点和需求来选择合适的ID生成算法。

## 实验要求

- 要求学生掌握在 Maven 项目中管理依赖项的方式。
- 要求学生理解并熟练使用 Maven 命令执行 Spring Boot 项目构建打包。
- 要求学生学会安装和配置 Maven 仓库管理工具 Sonatype Nexus 并成功应用。

## 实验环境

- JDK17
- Maven 3.9
- Docker
- Ubuntu 22.04 LTS(用咱们之前git虚拟机就可以吧)
- Sonatype Nexus 3.x

## 实验步骤

### Maven安装

#### JDK17下载安装

Maven编译运行离不开JDK, 在安装Apache Maven之前, 需要先在系统中安装JDK, 并设置java环境变量. Maven 3.9+ 需要 JDK8及以上版本支持, 本节介绍在Ubuntu 22.04 LTS上安装JDK17.

1. 在Oracle官网或使用wget在当前目录获取jdk-17\_linux-x64\_bin.tar.gz 压缩包,然后解压后移至/usr/local 文件夹下.

输入一下指令

```
# 下载JDK17
wget https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz
# 查看
ls
# 解压
sudo tar -xzf jdk-17_linux-x64_bin.tar.gz
# 再次查看
ls
# 转移到/usr/local/jdk-17
sudo mv jdk-17.0.12 /usr/local/jdk-17

# 查看是不是成功转移
cd /usr/local
ls
```

```

beryl@beryl-virtual-machine:~/桌面$ wget https://download.oracle.com/java/17/archive/jdk-17_linux-x64_bin.tar.gz
--2024-11-11 15:46:46-- https://download.oracle.com/java/17/archive/jdk-17_linux-x64_bin.tar.gz
正在解析主机 download.oracle.com (download.oracle.com)... 23.48.228.98
正在连接 download.oracle.com (download.oracle.com)|23.48.228.98|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度： 180555480 (172M) [application/x-gzip]
正在保存至：‘jdk-17_linux-x64_bin.tar.gz’

jdk-17_linux-x64_bi 100%[=====>] 172.19M 14.1MB/s 用时 16s

2024-11-11 15:47:03 (10.7 MB/s) - 已保存 ‘jdk-17_linux-x64_bin.tar.gz’ [180555480/180555480])

beryl@beryl-virtual-machine:~/桌面$ ls
jdk-17_linux-x64_bin.tar.gz
jdk-17.0.12/man/man1/klist.1
jdk-17.0.12/man/man1/ktab.1
jdk-17.0.12/man/man1/rmiregistry.1
jdk-17.0.12/man/man1/serialver.1
jdk-17.0.12/release
beryl@beryl-virtual-machine:~/桌面$ ls
jdk-17 jdk-17.0.12 jdk-17.0.12_linux-aarch64_bin.tar.gz
beryl@beryl-virtual-machine:~/桌面$ sudo mv jdk-17.0.12 /usr/local/jdk-17
beryl@beryl-virtual-machine:~/桌面$ cd /usr/local
beryl@beryl-virtual-machine:/usr/local$ ls
bin etc games include jdk-17 lib man sbin share src
beryl@beryl-virtual-machine:/usr/local$

```

2. 在配置文件 `/etc/profile` 中配置环境变量 `JAVA_HOME`、`CLASSPATH`。配置完成后执行 `java -version` 查看是否正确安装了 Java。

- `JAVA_HOME`: JAVA的安装目录, 配置后可通过`$JAVA_HOME`直接获取
- `CLASSPATH`: 告诉JVM在哪些目录下可以找到JAVA执行程序所需要的类或者包, 供类加载时使用。

#### # 配置

```

sudo echo "export JAVA_HOME=/usr/local/jdk-17" >> /etc/profile
sudo echo "export PATH=$PATH:$JAVA_HOME/bin" >> /etc/profile
sudo echo "export CLASSPATH=.:$JAVA_HOME/lib" >> /etc/profile

```

#### # 让文件激活

```

source /etc/profile # 使配置文件生效

```

```
# 查看效果
java -version
```

如果, 被告知权限不足的话, 则改为执行下面的命令:

```
echo "export JAVA_HOME=/usr/local/jdk-17" | sudo tee -a /etc/profile
echo "export PATH=\$PATH:\$JAVA_HOME/bin" | sudo tee -a /etc/profile
echo "export CLASSPATH=.\$JAVA_HOME/lib" | sudo tee -a /etc/profile
```

```
beryl@beryl-virtual-machine:/usr/local$ echo "export JAVA_HOME=/usr/local/jdk-17"
" | sudo tee -a /etc/profile
export JAVA_HOME=/usr/local/jdk-17
beryl@beryl-virtual-machine:/usr/local$ echo "export PATH=\$PATH:\$JAVA_HOME/bin"
" | sudo tee -a /etc/profile
export PATH=$PATH:$JAVA_HOME/bin
beryl@beryl-virtual-machine:/usr/local$ echo "export CLASSPATH=.\$JAVA_HOME/lib"
" | sudo tee -a /etc/profile
export CLASSPATH=.\$JAVA_HOME/lib
beryl@beryl-virtual-machine:/usr/local$ source /etc/profile
beryl@beryl-virtual-machine:/usr/local$ java -version
java version "17.0.12" 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 17.0.12+8-LTS-286)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.12+8-LTS-286, mixed mode, sharing)
beryl@beryl-virtual-machine:/usr/local$
```

- 编辑 ~/.bashrc 文件:

```
nano ~/.bashrc
```

- 在文件末尾添加以下内容

```
export JAVA_HOME=/usr/local/jdk-17
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.\$JAVA_HOME/lib
```

- 保存文件后, 执行以下命令使更改立即生效:

```
source ~/.bashrc
```

```
beryl@beryl-virtual-machine:/usr/local$ source ~/.bashrc
beryl@beryl-virtual-machine:/usr/local$ java --version
java 17.0.12 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 17.0.12+8-LTS-286)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.12+8-LTS-286, mixed mode, sharing)
beryl@beryl-virtual-machine:/usr/local$
```

## Maven下载安装

Maven下载地址: <https://maven.apache.org/download.cgi>

1. 在终端输入 `mvn -v` 查看本地是否已存在 Maven, 若 Maven 已安装, 该命令会输出Maven 版本号, 可自行选择重装 Maven 或直接跳到下一节学习。反之, 在 Maven 官网或使用 `wget` 在当前目录下载最新版本的Maven 压缩包, 解压后移至 `/usr/local/`文件夹下。

```
mvn -v
```

```
# 查看
ls
# 解压
sudo tar -xzf apache-maven-3.9.6-bin.tar.gz
# 再次查看
ls
# 转移
sudo mv apache-maven-3.9.6 /usr/local/apache-maven-3.9
```



```
beryl@beryl-virtual-machine:~/桌面$ wget https://dlcdn.apache.org/maven/maven-3/3.9.6/binaries/
apache-maven-3.9.6-bin.tar.gz
--2024-11-11 21:19:45-- https://dlcdn.apache.org/maven/maven-3/3.9.6/binaries/apache-maven-3.9
.6-bin.tar.gz
正在解析主机 dlcdn.apache.org (dlcdn.apache.org)... 151.101.2.132, 2a04:4e42::644
正在连接 dlcdn.apache.org (dlcdn.apache.org)|151.101.2.132|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度： 9410508 (9.0M) [application/x-gzip]
正在保存至：‘apache-maven-3.9.6-bin.tar.gz’

apache-maven-3.9.6- 100%[=====>] 8.97M 12.0MB/s 用时 0.7s

2024-11-11 21:19:46 (12.0 MB/s) - 已保存 ‘apache-maven-3.9.6-bin.tar.gz’ [9410508/9410508])

beryl@beryl-virtual-machine:~/桌面$ ls
apache-maven-3.9.6-bin.tar.gz  jdk-17.0.12_linux-x64_bin.tar.gz
beryl@beryl-virtual-machine:~/桌面$ sudo tar -xzf apache-maven-3.9.6-bin.tar.gz[sudo] beryl 的
密码：
apache-maven-3.9.6/README.txt
apache-maven-3.9.6/LICENSE
apache-maven-3.9.6/NOTICE
apache-maven-3.9.6/lib/

beryl@beryl-virtual-machine:~/桌面$ ls
apache-maven-3.9.6  jdk-17.0.12_linux-x64_bin.tar.gz
beryl@beryl-virtual-machine:~/桌面$ sudo mv apache-maven-3.9.6 /usr/local/apache-maven-3.9
```

2. 在 `/etc/profile` 文件中配置 `maven` 环境变量 `M2_HOME`，配置完成后执行 `mvn -version` 查看是否正确安装了 Maven。

- 编辑 `~/.bashrc` 文件：

```
nano ~/.bashrc
```

- 在文件末尾添加以下内容

```
export M2_HOME=/usr/local/apache-maven-3.9
export PATH=$M2_HOME/bin:$PATH
```

- 保存文件后，执行以下命令使更改立即生效：

```
source ~/.bashrc
```



```
beryl@beryl-virtual-machine:~/桌面$ nano ~/.bashrc
beryl@beryl-virtual-machine:~/桌面$ source ~/.bashrc
beryl@beryl-virtual-machine:~/桌面$ mvn -v
Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dcc295161ae)
Maven home: /usr/local/apache-maven-3.9
Java version: 17.0.12, vendor: Oracle Corporation, runtime: /usr/local/jdk-17
Default locale: zh_CN, platform encoding: UTF-8
OS name: "linux", version: "6.8.0-48-generic", arch: "amd64", family: "unix"
```

## Maven仓库配置

1. 修改本地仓库地址, Maven默认仓库地址为`~/.m2/repository`。推荐将本地仓库配置至Maven安装目录下, 需要在文件`%M2_HOME/conf/setting.xml` 中配置`<localRepository>`。

在`/usr/local/apache-maven-3.9` 目录下

```
cd /usr/local/apache-maven-3.9
pwd
sudo mkdir local_repository
sudo vim conf/settings.xml
```

中间显示`sudo: vim: command not found`, 直接安装一个就解决了:

```
# 安装vim
sudo apt update
sudo apt install vim
```

2. Maven 远程仓库默认为 Maven Central Repository, 可以在 `settings.xml` 中配置远程镜像仓库为国内镜像仓库, 提高 Jar 包下载的速度和稳定性。配置`settings.xml` 完成后保存退出。

```
<mirror>
  <id>alimaven</id>
  <mirrorOf>central</mirrorOf>
  <name>aliyun maven</name>
  <url>https://maven.aliyun.com/repository/public</url>
</mirror>
```

```
beryl@beryl-virtual-machine: /usr/local/apache-maven-3.9

<id>mirrorId</id>
<mirrorOf>repositoryId</mirrorOf>
<name>Human Readable Name for this Mirror.</name>
<url>http://my.repository.com/repo/path</url>
</mirror>
-->
<mirror>
  <id>alimaven</id>
  <name>aliyun maven</name>
  <url>https://maven.aliyun.com/repository/public</url>
  <mirrorOf>central</mirrorOf>
</mirror>
<mirror>
  <id>maven-default-http-blocker</id>
  <mirrorOf>external:http:*</mirrorOf>
  <name>Pseudo repository to mirror external repositories initially using HTTP.</name>
  <url>http://0.0.0.0/</url>
  <blocked>true</blocked>
</mirror>

160,4 60%
```

## 基于Maven构建雪花算法组件

### 创建Java项目

1. 使用 Maven Archetype 创建一个基本的 Java 项目，本节将基于该项目实现雪花算法并将其构建成组件发布。创建项目时需要配置 `groupId`、`artifactId`。

打开终端

输入以下指令:

```
cd ~
mvn archetype:generate -DgroupId=cc.synx -DartifactId=snowflake -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
ls -F # 查看信息
```

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 01:53 min
[INFO] Finished at: 2024-11-12T19:40:24+08:00
[INFO] -----
beryl@beryl-virtual-machine:~/桌面$ ls -F
apache-maven-3.9.6-bin.tar.gz  jdk-17.0.12_linux-x64_bin.tar.gz  snowflake/
```

```
cd snowflake/  
sudo apt install tree  
tree
```

```
beryl@beryl-virtual-machine:~/桌面/snowflake$ tree
```

```
├── pom.xml  
└── src  
    ├── main  
    │   ├── java  
    │   │   └── cc  
    │   │       └── synx  
    │   │           └── App.java  
    └── test  
        ├── java  
        │   └── cc  
        │       └── synx  
        │           └── AppTest.java  
  
9 directories, 3 files
```

2. pom.xml 中 package 标签为 jar，使用 Maven 可以将该项目打为 JAR 包使用。注意，当前pom.xml中没有打包插件，生成的 JAR 包需要指定主启动类方可运行。

```
# 查看  
cat pom.xml
```

```
beryl@beryl-virtual-machine:~/桌面/snowflake$ cat pom.xml  
<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/maven-v4_0_0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>cc.synx</groupId>  
  <artifactId>snowflake</artifactId>  
  <packaging>jar</packaging>  
  <version>1.0-SNAPSHOT</version>  
  <name>snowflake</name>  
  <url>http://maven.apache.org</url>  
  <dependencies>  
    <dependency>  
      <groupId>junit</groupId>  
      <artifactId>junit</artifactId>  
      <version>3.8.1</version>  
      <scope>test</scope>  
    </dependency>  
  </dependencies>  
</project>
```

## 实现雪花算法

1. 在APP.java 同级目录下创建SnowflakeUtil.java , 输入以下指令并添加代码:

```
touch SnowflakeUtil.java
```

```
package cc.synx;

public class SnowflakeUtil {
    private static final long EPOCH = 1704038400000L; // 设置起始的时间戳(2024-01-01 00:00:00)

    private long workerId;
    private long sequence = 0L;
    private long lastTimestamp = -1L;

    public SnowflakeUtil(long workerId) {
        this.workerId = workerId;
    }

    public synchronized long nextId() {
        long timestamp = System.currentTimeMillis();

        if (timestamp < lastTimestamp) {
            throw new RuntimeException("Clock moved backwards. Refusing to generate id for " + (lastTimestamp - timestamp) + " milliseconds. ");
        }

        if (lastTimestamp == timestamp) {
            sequence = (sequence + 1) & 4095;
            if (sequence == 0) {
                timestamp = tilNextMillis(lastTimestamp);
            }
        } else {
            sequence = 0L;
        }

        lastTimestamp = timestamp;

        return ((timestamp - EPOCH) << 22) | (workerId << 12) | sequence;
    }

    private long tilNextMillis(long lastTimestamp) {
        long timestamp = System.currentTimeMillis();
        while (timestamp <= lastTimestamp) {
            timestamp = System.currentTimeMillis();
        }
        return timestamp;
    }
}
```

2. 编辑APP.java的main方法, 调用SnowflakeUtil的nextId()方法.

```
package cc.synx;

public class App {
    public static void main(String[] args) {
        SnowflakeUtil snowflakeUtil = new SnowflakeUtil(1L);
        System.out.println("First snowflake id is: " + snowflakeUtil.nextId());
    }
}
```

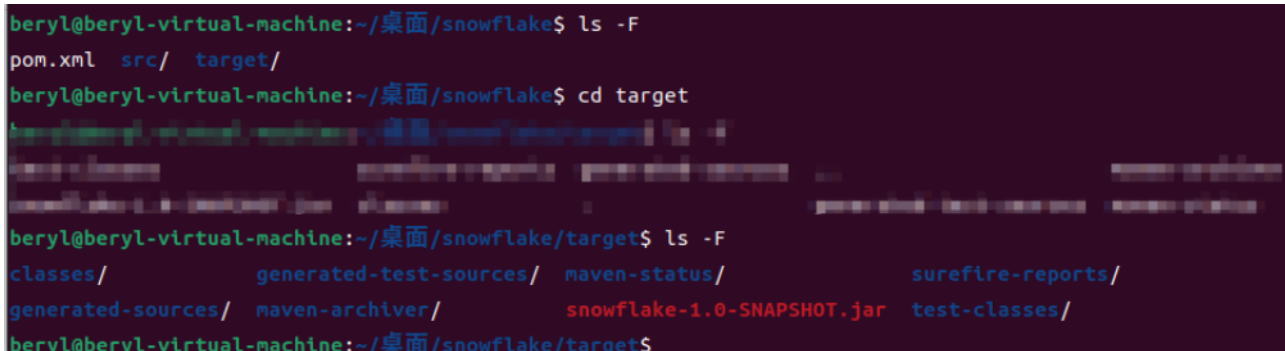
## 构建运行项目

1. pom.xml 同级目录(自己导航到对应的目录下面)下, 通过Maven工具构建项目并打包成JAR.

```
ls -F
mvn clean # 清理
mvn package # 打包
```

2. 成功执行mvn clean package 命令后, 在pom.xml 同级目录下生成了target目录, 包括Java字节码文件和生成的JAR包snowflake-1.0-SNAPSHOT.jar

```
ls -F
cd target
ls -F
```



```
beryl@beryl-virtual-machine:~/桌面/snowflake$ ls -F
pom.xml  src/  target/
beryl@beryl-virtual-machine:~/桌面/snowflake$ cd target
beryl@beryl-virtual-machine:~/桌面/snowflake/target$ ls -F
classes/          generated-test-sources/  maven-status/          surefire-reports/
generated-sources/  maven-archiver/         snowflake-1.0-SNAPSHOT.jar  test-classes/
```

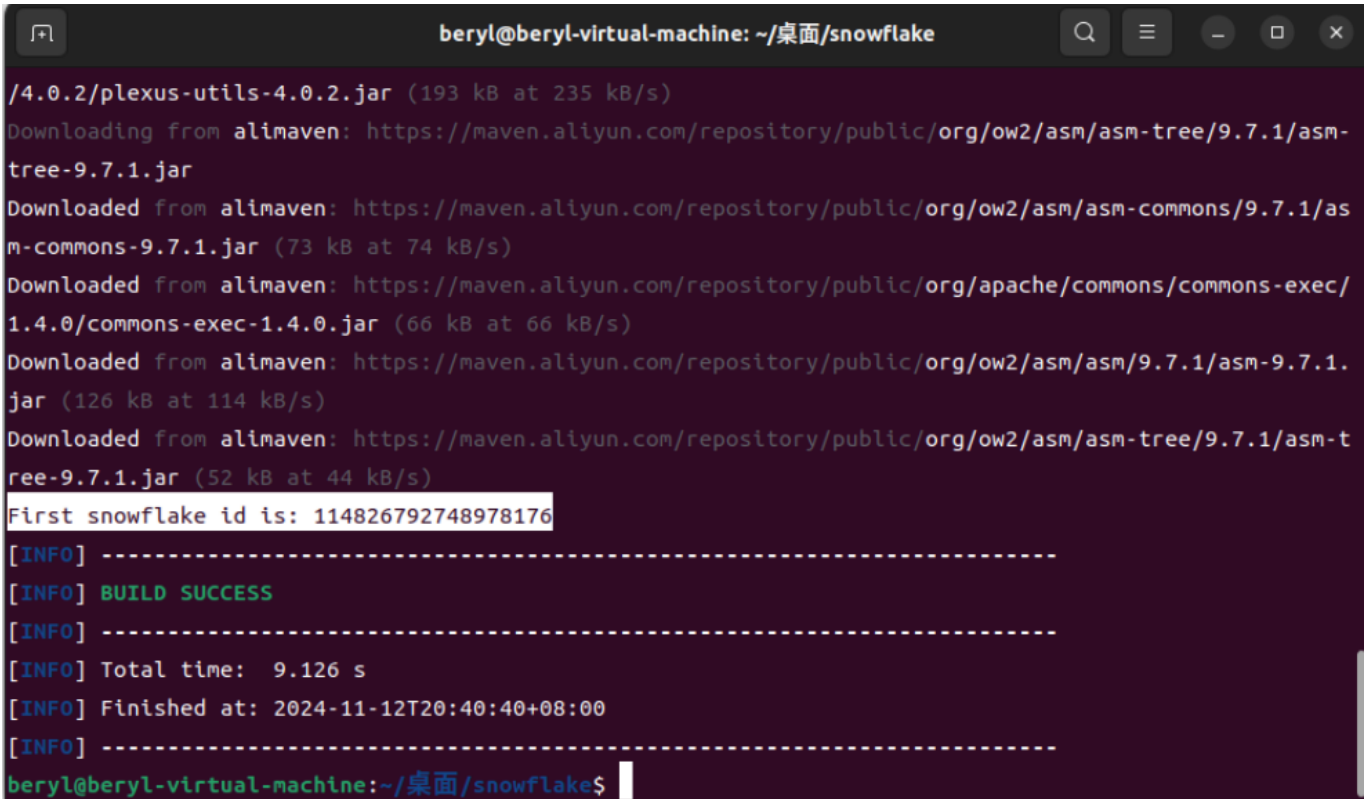
3. 使用java -jar xxx.jar 命令运行JAR包, 发现该JAR包没有指定主执行类, 因此在运行时需要指定Java MainClass.

```
java -jar snowflake-1.0-SNAPSHOT.jar
java -cp snowflake-1.0-SNAPSHOT.jar cc.synx.App
```

```
beryl@beryl-virtual-machine:~/桌面/snowflake/target$ java -jar snowflake-1.0-SNAPSHOT.jar
snowflake-1.0-SNAPSHOT.jar中没有主清单属性
beryl@beryl-virtual-machine:~/桌面/snowflake/target$ java -cp snowflake-1.0-SNAPSHOT.jar cc.synx.App
First snowflake id is: 114826112676139008
```

4. 执行如下Maven命令也可以运行Java项目(注意这条指令要在**pom.xml**同目录下的终端执行):

```
mvn exec:java -Dexec.mainClass="cc.synx.App"
```



```
beryl@beryl-virtual-machine: ~/桌面/snowflake
/4.0.2/plexus-utils-4.0.2.jar (193 kB at 235 kB/s)
Downloading from alimaven: https://maven.aliyun.com/repository/public/org/ow2/asm/asm-tree/9.7.1/asm-tree-9.7.1.jar
Downloaded from alimaven: https://maven.aliyun.com/repository/public/org/ow2/asm/asm-commons/9.7.1/asm-commons-9.7.1.jar (73 kB at 74 kB/s)
Downloaded from alimaven: https://maven.aliyun.com/repository/public/org/apache/commons/commons-exec/1.4.0/commons-exec-1.4.0.jar (66 kB at 66 kB/s)
Downloaded from alimaven: https://maven.aliyun.com/repository/public/org/ow2/asm/asm/9.7.1/asm-9.7.1.jar (126 kB at 114 kB/s)
Downloaded from alimaven: https://maven.aliyun.com/repository/public/org/ow2/asm/asm-tree/9.7.1/asm-tree-9.7.1.jar (52 kB at 44 kB/s)
First snowflake id is: 114826792748978176
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.126 s
[INFO] Finished at: 2024-11-12T20:40:40+08:00
[INFO] -----
beryl@beryl-virtual-machine:~/桌面/snowflake$
```

## 将JAR包存储至本地仓库

1. 在 **pom.xml** 同级目录下执行 **mvn install** 命令将 JAR 包安装至本地仓库中。

```
sudo chown -R $USER:$USER /usr/local/apache-maven-3.9/local_repository # 赋予用户访问这个文件的权利
```

```
mvn install:install-file -Dfile=target/snowflake-1.0-SNAPSHOT.jar -DgroupId=cc.synx -DartifactId=snowflake -Dversion=1.0 -Dpackaging=jar
```

上述命令参数解释如下:

- **-Dfile**: 指定要安装的 JAR 文件的路径。
- **-DgroupId**: 指定组 ID。
- **-DartifactId**: 指定artifact ID。
- **-Dversion**: 指定版本号
- **-Dpackaging**: 指定打包类型(在这种情况下是 JAR)


```
beryl@beryl-virtual-machine:~/桌面/snowflake$ mvn install:install-file -Dfile=target/snowflake-1.0-SNAPSHOT.jar
-DgroupId=cc.synx -DartifactId=snowflake -Dversion=1.0 -Dpackaging=jar
[INFO] Scanning for projects...
[INFO]
[INFO] -----< cc.synx:snowflake >-----
[INFO] Building snowflake 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- install:3.1.1:install-file (default-cli) @ snowflake ---
[INFO] Installing /home/beryl/桌面/snowflake/target/snowflake-1.0-SNAPSHOT.jar to /home/beryl/.m2/repository/cc/synx/snowflake/1.0/snowflake-1.0.jar
[INFO] Installing /tmp/snowflake-1.0-SNAPSHOT12588708899395293699.pom to /home/beryl/.m2/repository/cc/synx/snowflake/1.0/snowflake-1.0.pom
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.591 s
[INFO] Finished at: 2024-11-12T20:46:22+08:00
[INFO] -----
```

## 2. 查看Maven本地仓库是否存在Snowflake组件包

```
cd /usr/local/apache-maven-3.9/local_repository/
ls -F
```

## 3. Snowflake 组件在本地仓库安装成功后，其他Maven项目在pom.xml文件中可以通过<dependency></dependency>标签引入该组件。

```
<dependency>
  <groupId>cc.synx</groupId>
  <artifactId>snowflake</artifactId>
  <version>1.0</version>
</dependency>
```

 将 Snowflake 组件作为依赖添加到新的 Maven 项目中，可以让该项目复用 Snowflake 的功能和代码。这种做法的主要作用和优势如下：

**代码复用** 通过将 Snowflake 组件打包并安装到本地仓库，其他项目可以直接引用它，而不需要重新编写 Snowflake 中的代码。只需在 pom.xml 中添加依赖项，就可以使用 Snowflake 提供的所有类和方法。这大大提高了代码的复用性和开发效率。

**简化依赖管理** Maven 自动管理依赖关系。添加 Snowflake 组件依赖后，Maven 会自动将其下载（如果在远程仓库中）或从本地仓库中获取，并将其包含到项目的构建路径中，开发者无需手动复制文件或处理复杂的依赖关系。

**版本控制** 使用 Maven 依赖管理，可以轻松地对 Snowflake 组件进行版本控制。比如，1.0-SNAPSHOT 只是一个版本，未来可以发布更新的版本（例如 1.1 或 2.0）。项目中可以指定所需的版本，这样如果 Snowflake 组件更新了，新项目可以选择升级依赖的版本而不影响其他项目的运行。



**组件化开发** 将项目分解为多个组件（如 `Snowflake`）可以让项目更模块化、易于维护。各个组件之间是松耦合的，可以独立开发、测试和发布。团队可以专注于开发某个组件，同时其他团队可以直接使用这些组件，提高协作效率。

### 依赖传递

如果 `Snowflake` 组件依赖其他库（例如 `Junit`），Maven 会自动处理这些传递依赖。新项目无需手动添加 `Snowflake` 的所有依赖项，Maven 会自动引入并管理所需的依赖库。

## 配置Maven私有仓库

### 构建Maven私有仓库

Nexus 官网: <https://www.sonatype.com/>

sonatype/nexus 镜像网址: <https://hub.docker.com/r/sonatype/nexus3/>

1. 本节将使用 Nexus Repository Manager3 仓库管理工具来搭建独立的 Maven 私有仓库。安装 Nexus 的机器(必须有 JDK8, Nexus 3.x 只能在 JDK8 环境下运行)和构建 Maven 项目的机器不要求一致。

```
sudo apt update
sudo apt install openjdk-8-jdk
java -version # 查看版本
```

```
正在设置 libxt-dev:amd64 (1:1.2.1-1) ...
beryl@beryl-virtual-machine:~/桌面$ java -version
openjdk version "1.8.0_432"
OpenJDK Runtime Environment (build 1.8.0_432-8u432-ga~us1-0ubuntu2~22.04-ga)
OpenJDK 64-Bit Server VM (build 25.432-bga, mixed mode)
```

```
sudo docker run -d -p 8081:8081 -e INSTALL4J_ADD_VM_PARAMS="-Xms128m -Xmx1024m" --name nexus sonatype/nexus3
sudo docker ps
```

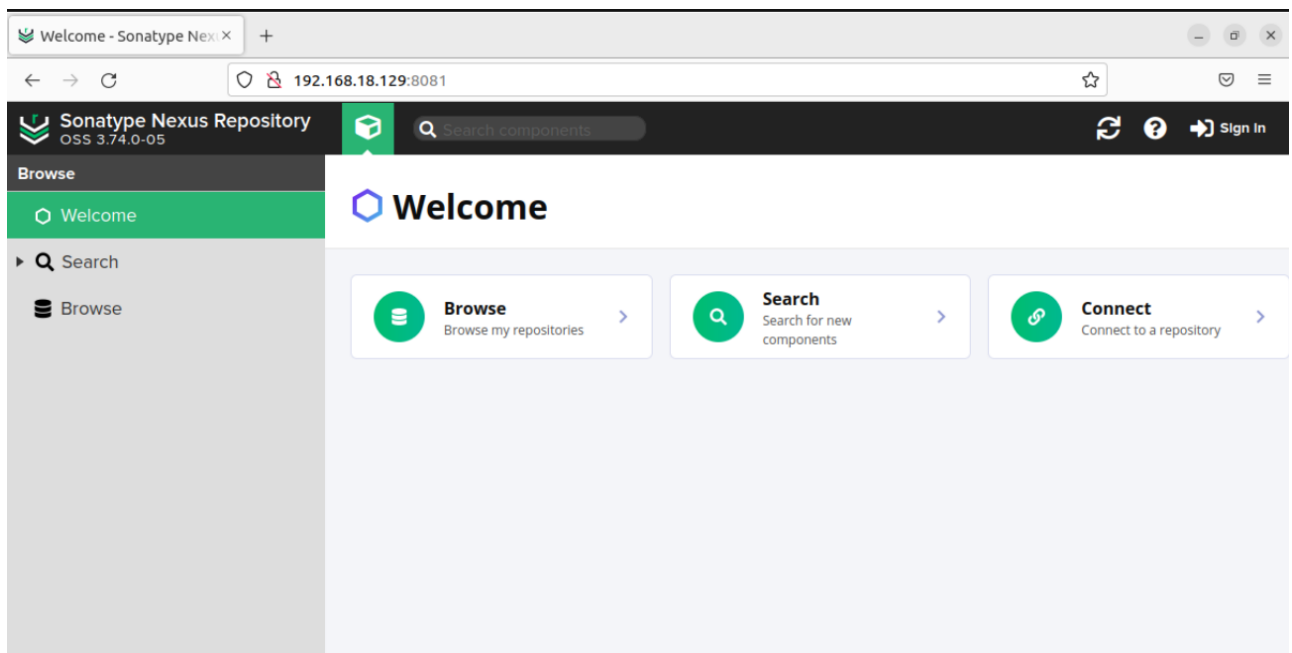
```
beryl@beryl-virtual-machine:~/桌面$ sudo docker run -d -p 8081:8081 -e INSTALL4J_ADD_VM_PARAMS="-Xms128m -Xmx1024m" --name nexus sonatype/nexus3
Unable to find image 'sonatype/nexus3:latest' locally
latest: Pulling from sonatype/nexus3
ccc2996f86eb: Pull complete
07241f9a48ef: Pull complete
309813c1cb87: Pull complete
0aa2466e3f7c: Pull complete
be16f82acac8: Pull complete
18a390d9c2b3: Pull complete
b22df4fd3341: Pull complete
Digest: sha256:ec9891ccacfc6540e0df714f02c787aeef254d7ee6e27060c2233bca9ed1d206
Status: Downloaded newer image for sonatype/nexus3:latest
ea547700841ba2a7f25c046b9feefba3d9a466f5533deb1001e54fdb5267ff9
beryl@beryl-virtual-machine:~/桌面$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
ea547700841b	sonatype/nexus3	"/opt/sonatype/nexus..."	2 minutes ago	Up 2 minutes	0.0.0.0:8081->8081/tcp, :::8081->8081/tcp

nexus

2. Nexus 默认以 8081 端口开放，访问 `http://<server-ip>:8081` 查看 Nexus 的页面，本次实验勿开启安全检查。

```
ip a # 查看ip
```

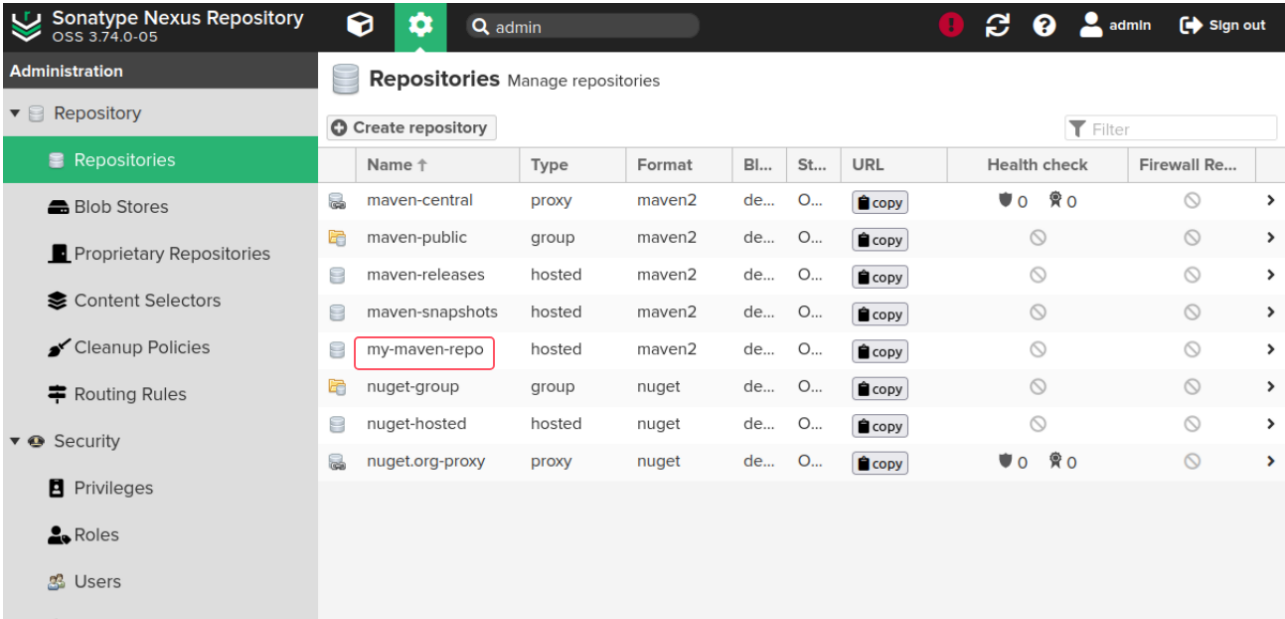


3. 点击 `sign in`，Username 输入 `admin`，password 则位于 `${nexus 安装目录}/sonatype-work/nexus/admin.password` 文件中(docker下载的看下面的指令就行)，首次登录成功后需要修改密码。

```
sudo docker exec -it nexus cat /nexus-data/admin.password
```

4. Nexus 登录成功后，点击 `create repository` 新建 Maven 私有仓库

5. 仓库类型选择 **maven2(hosted)**类型，仓库名填入 **my-maven-repo**，仓库制品类型选择 **Release**，完成创建仓库。



将组件上传至Maven私有仓库

1. 编辑 Maven 的 **settings.xml** 文件，添加私有仓库的配置信息。在**<servers/>**标签中添加仓库名、账号密码，在**<mirrors/>**和**<profiles/>**标签中添加仓库URL、仓库ID等，最后在**<activeProfiles/>**标签中使私有仓库生效。

```
cd /usr/local/apache-maven-3.9
sudo vim conf/settings.xml
```

```
<servers>
  <server>
    <id>my-maven-repo</id>
    <username>admin</username>
    <password>"仓库密码"</password>    <!-- 仓库密码(密码部分不需要加引号) -->
  </server>
  <!-- 其他服务器 -->
</servers>

<mirrors>
  <mirror>
    <id>my-maven-repo</id>
    <url>http://<server-id>:8081/repository/my-maven-repo/</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
  <!-- 其他镜像仓库 -->
</mirrors>

<profiles>
  <profile>
    <id>my-maven-repo</id>
```

```

    <repositories>
      <repository>
        <id>my-maven-repo</id>
        <url>http://<server-id>:8081/repository/my-maven-repo/</url>
      </repository>
    </repositories>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>my-maven-repo</activeProfile>
</activeProfiles>

```

```

| Authentication profiles can be used whenever Maven must make a connection to a remote server
|-->
<!-->
<servers>
  <server>
    <id>my-maven-repo</id>
    <username>admin</username>
    <password>[REDACTED]</password>
  </server>
  <!-- 其他服务器 -->
  <!-- server
    | Specifies the authentication information to use when connecting to a particular server, id
ted by

```

2. 保存退出后，在 `snowflake` 项目的 `pom.xml` 中添加上传远程仓库的配置

修改 `pom.xml` 内容

```

<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>cc.synx</groupId>
  <artifactId>snowflake</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version> <!-- 将version中的snapshot去掉 -->

  <name>snowflake</name>
  <url>http://maven.apache.org</url>

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

```

```
</dependencies>

<distributionManagement>
  <repository>
    <id>my-maven-repo</id>
    <url>http://<server-id>:8081/repository/my-maven-repo/</url>
  </repository>
</distributionManagement>
</project>
```

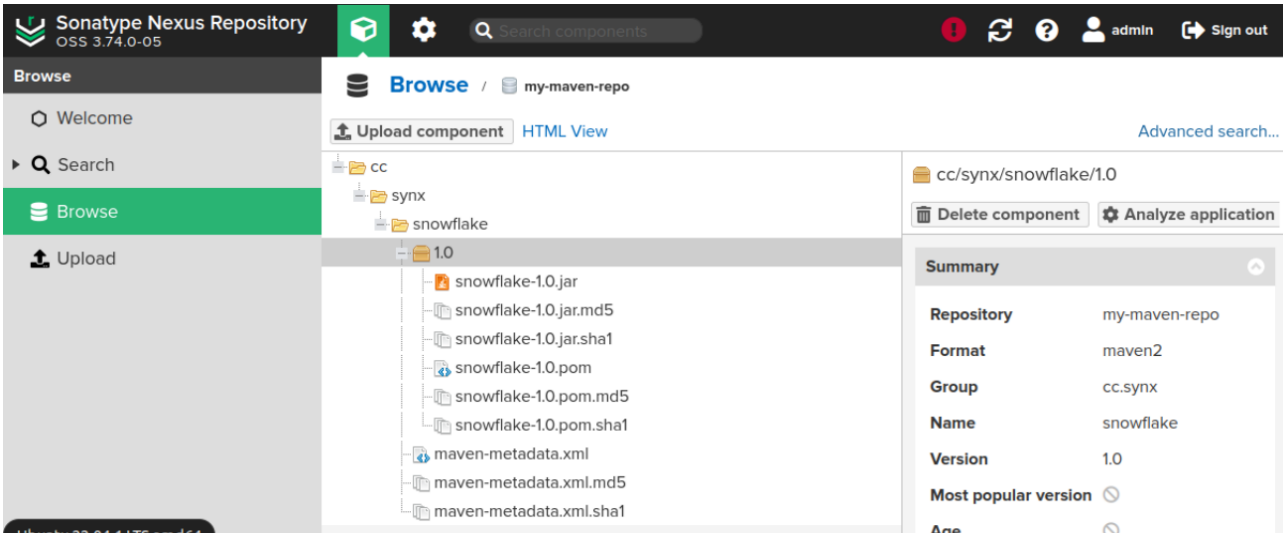
请记得根据实际的 `server-id` 替换掉 `<url>` 部分的 `server-id`

3. 将 6.2.3 节构建的 Java 组件上传至私有仓库。(这一步中的 `mvn deploy` 指令需要在包含 `pom.xml` 文件的项目根目录下执行。)

```
mvn deploy
```

```
[INFO] --- deploy:3.1.1:deploy (default-deploy) @ snowflake ---
Uploading to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/1.0/snowflake-1.0.pom
Uploaded to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/1.0/snowflake-1.0.pom (833 B at 143 B/s)
Uploading to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/1.0/snowflake-1.0.jar
Uploaded to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/1.0/snowflake-1.0.jar (3.2 kB at 7.6 kB/s)
Downloading from my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/maven-metadata.xml
Uploading to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/maven-metadata.xml
Uploaded to my-maven-repo: http://192.168.18.129:8081/repository/my-maven-repo/cc/synx/snowflake/maven-metadata.xml (292 B at 504 B/s)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 19.670 s
[INFO] Finished at: 2024-11-12T23:43:10+08:00
[INFO] -----
```

4. 上传完成后，在 Nexus 页面上可以查看到已上传的snowflake-1.0 组件



基于私有仓库构建项目

1. 参考 上一小节"基于Maven构建雪花算法组件" 创建新的Maven项目, 在pom.xml中引入snowflake依赖, 并在App.java的main方法中使用SnowflakeUtil工具类.

```
# pom.xml
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>cc.synx</groupId>
  <artifactId>snowflake</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>snowflake</name>
  <url>http://maven.apache.org</url>

  <!-- 配置私有仓库 -->
  <repositories>
    <repository>
      <id>my-maven-repo</id>
      <url>http://<192xxxxxxx>:8081/repository/my-maven-repo/</url>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>cc.synx</groupId>
      <artifactId>snowflake</artifactId>
      <version>1.0</version>

    </dependency>

    <dependency>
      <groupId>junit</groupId>
```

```
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>

</dependencies>

</project>
```

App.java的main方法内容如下:

```
public static void main(String[] args) {
    SnowflakeUtil snowflakeUtil = new SnowflakeUtil(1L);
    System.out.println("snowflakeId:" + snowflakeUtil.nextId());
}
```