### Maven介绍

#### 1.为什么使用maven

* 统一集中管理好所有的依赖包：jar包之间会有关联，在使用一个依赖之前，需要确定这个依赖所依赖的其他依赖，当项目比较大的时候，依赖管理会变得非常麻烦臃肿。
* Maven自动解决重复和冲突问题：对应第三方组件用到的共同jar。
* 提供了公共接口，方便同第三方插件集成。
* Maven处理多模块项目：如果项目比较复杂，要做成多模块项目，一般来说，多模块项目，每一个模块无法独立运行，要多个模块合在一起，项目才可以运行，借助 Maven 工具，可以实现项目的一键打包。

#### 2.maven是什么

一个项目管理工具，包含了一个项目对象模型（Project Object Model），反映在配置中，就是一个pom.xml文件。是一组标准集合，一个项目的生命周期、一个依赖管理系统，另外还包括定义在项目生命周期阶段的插件(plugin)以及目标(goal)。

Maven中的两大核心：

* 依赖管理：对jar的统一管理（maven提供了一个中央仓库，在项目中添加完依赖后，Maven会自动去中央仓库下载相关的依赖，并且解决依赖问题）。
* 项目构建：对项目进行编译、测试、打包、部署、上传到私服等。

#### 3.安装

maven是Java项目，安装必须要有Java环境。

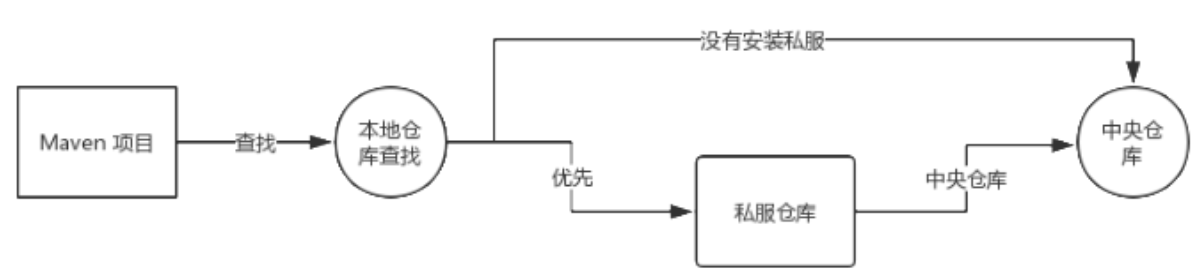
配置环境变量

#### 4.maven配置

仓库类型

|  |  |
| --- | --- |
| 仓库类型 | 说明 |
| 本地仓库 | 默认位置 当前用户名\.m2\repository |
| 私服仓库 | 公司内部搭建的Maven私服，处于局域网，访问速度较快，这个仓库中存放的jar一般就是自己公司内部开发的jar |
| 中央仓库 | 由Apache团队维护，包含了大部分的jar，早期不包含Oracle数据库驱动，2019年8月开始包含 |

存在3个仓库，jar包如何找呢？



* 本地仓库配置：默认位置即可，不建议修改
* 远程镜像配置：由于默认的中央仓库下载比较慢，因此，可以将远程仓库地址改为阿里巴巴的仓库地址：

<mirror>

<id>nexus-aliyun</id>

<mirrorOf>central</mirrorOf>

<name>Nexus aliyun</name>

<url>http://maven.aliyun.com/nexus/content/groups/public</url>

</mirror>

### 1、常见问题

#### 1.1 dependencies 和 dependencyManagement，plugins 和 pluginManagement 有什么区别?

dependencyManagement是表示依赖jar包的声明，即你在项目中的 dependencyManagement下声明了依赖，maven不会加载该依赖，dependencyManagement声明可以被继承。

dependencyManagement的一个使用案例是当有父子项目的时候，父项目中可以利用dependencyManagement声明子项目中需要用到的依赖jar包，之后，当某个或者某几个子项目需要加载该插件的时候，就可以在子项目中 dependencies 节点只配置 groupId 和 artifactId 就可以完成插件的引用。

dependencyManagement主要是为了统一管理插件，确保所有子项目使用的插件版本保持一致，类似的还有 plugins 和 pluginManagement。

#### 1.2 IDEA修改JDK版本后编译报错

修改JDK版本，指定maven-compiler-plugin的source和target为1.8。执行maven指令，报错。

错误原因：

maven的jdk源于指定的jdk编译版本不符。

排错：

1. 查看Project Settings：Project SDK是否正确；SDK路径是否正确。
2. 查看Settings>Maven的配置：JDK for importer是否正确；Runner是否正确。

#### 1.3 重复引入依赖

在idea中，选中Module，使用Ctrl + Alt + Shift + U，打开依赖图，检索是否存在重复引用的情况。如果存在重复引用，可以将多余 的引用删除。

#### 1.4 如何打包一个可以直接运行的Spring Boot jar包

可以使用spring-boot-maven-plugin插件

<build>

<plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>repackage</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

如果引入了第三方jar包，如何打包？

首先，添加依赖

<dependency>

<groupId>io.github.dunwu</groupId>  
 <artifactId>dunwu-common</artifactId>  
 <version>1.0.0</version>  
 <scope>system</scope>

<systemPath>${project.basedir}/src/main/resources/lib/dunwu-common-1.0.0.jar</systemPath>  
</dependency>

接着，需要配置spring-boot-maven-plugin插件：

<build>

<plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <executions>  
 <execution>  
 <goals>  
 <goal>repackage</goal>  
 </goals>  
 </execution>  
 </executions>  
 <configuration>  
 <includeSystemScope>true</includeSystemScope>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

#### 1.5 去哪儿找maven dependency

官方推荐的搜索 maven dependency 网址:

* <https://search.maven.org>
* <https://repository.apache.org>
* <https://mvnrepository.com>

#### 1.6 如何指定编码

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
</properties>

#### 1.7 如何指定JDK版本

1. properties方式

<properties>

<maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
</properties>

1. 使用maven-compiler-plugin插件，并指定source和target版本

<plugin>

<artifactId>maven-compiler-plugin</artifactId>  
 <version>3.8.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
</plugin>

#### 1.8 如何避免将dependency打包到构件中？

指定maven dependency的scope为provided，这意味着：依赖关系将在运行时由其容器或JDK提供。

<dependency>

<groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <scope>provided</scope>  
</dependency>

#### 1.9 如何跳过单元测试

执行mvn package或mvn install时，会自动编译所有单元测试（src/test/java目录下的代码），如何跳过这一步？

在执行命令的后面，添加参数-Dmaven.test.skip=true或者-DskipTests=true

#### 1.10 如何引入本地jar

有时候，需要引入在中央仓库找不到的jar，但又想通过maven进行管理，那么应该如何做到呢？

* 将私有jar放置在resources/lib下，然后以如下方式添加依赖：
* groupId和artifactId可以按照jar包中的package设置，只要和其他jar不冲突即可。

<dependency>

<groupId>xxx</groupId>

<artifactId>xxx</artifactId>

<version>1.0.0</version>

<scope>system</scope>

<systemPath>${project.basedir}/src/main/resources/lib/xxx-6.0.0.jar</systemPath>

</dependency>

#### 1.11 如何排除依赖

比方项目中使用的libA依赖某个库的1.0版本。libB依赖某个库的2.0版本。如今想统一使用2.0版，怎么去掉1.0版的依赖？

通过exclusion排除指定依赖即可。

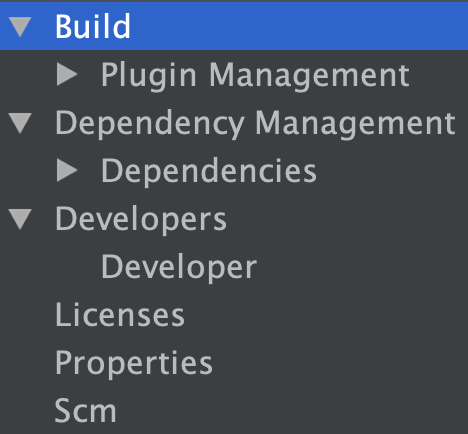
<dependency>

<groupId>org.apache.zookeeper</groupId>  
 <artifactId>zookeeper</artifactId>  
 <version>3.4.12</version>  
 <optional>true</optional>  
 <exclusions>  
 <exclusion>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>

### 2、最佳实践

#### 2.1 通过pom统一管理版本

采用类似spring-boot-dependencies的方式统一管理依赖版本。spring-boot-dependencies的pom.xml形式：



<?xml version="1.0" encoding="utf-8"?>

<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>org.springframework.boot</groupId>  
 <artifactId>spring-boot-dependencies</artifactId>  
 <version>2.1.3.RELEASE</version>  
 <packaging>pom</packaging>

<!--省略-->

<!--依赖包版本管理-->

<dependencyManagement>  
 <dependencies>

<!--省略-->

</dependencies>

</dependencyManagement>

<build>

<!--插件版本管理-->  
 <pluginManagement>  
 <plugins>

<!--省略-->

</plugins>

</pluginManagement>  
</build>

</project>

其他项目引入spring-boot-dependencies来管理版本的方式：

<parent>

<groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.3.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
</parent>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies </artifactId>

<version>${spring-boot.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

### 3、常用命令

|  |  |  |
| --- | --- | --- |
| mvn clean | 清理 | 清理已经编译好的文件 |
| mvn compile | 编译 | 将Java代码编译成Class文件 |
| mvn test | 测试 | 项目测试 |
| mvn package | 打包 | 根据用户的配置，将项目打成jar包或war包 |
| mvn install | 安装 | 手动向本地仓库安装一个jar |
| mvn deploy | 上传 | 将jar上传到私服 |

这些命令都不是独立运行的，它有一个顺序。

例如：

想将jar上传到私服，那么就要构建jar，就需要执行mvn package命令，要打包，当然也需要测试，那就要走mvn test命令，要测试就要先编译。因此，最终所有的命令都会执行一遍。不过，开发者也可以手动配置不执行某一个命令，这就是跳过。一般来说，除了测试，其他步骤都不建议跳过。

### 4、通过命令来构建项目

可以通过命令来构建一个maven项目：

mvn archetype:generate

-DgroupId=org.javaboy

-DartifactId=firstapp

-DarchetypeArtifactId=maven-archetype-quickstart

-DinteractiveMode=false

创建出成功后，就两个文件：



pom.xml中定义了所有的项目配置。

### 5、对项目进行打包

通过mvn package命令可以将创建的项目打成一个jar包。

在打包之前，需要配置JDK的版本至少为7以上，因此，我们还需要手动修改一下pom.xml文件，即添加如下配置：

<properties>

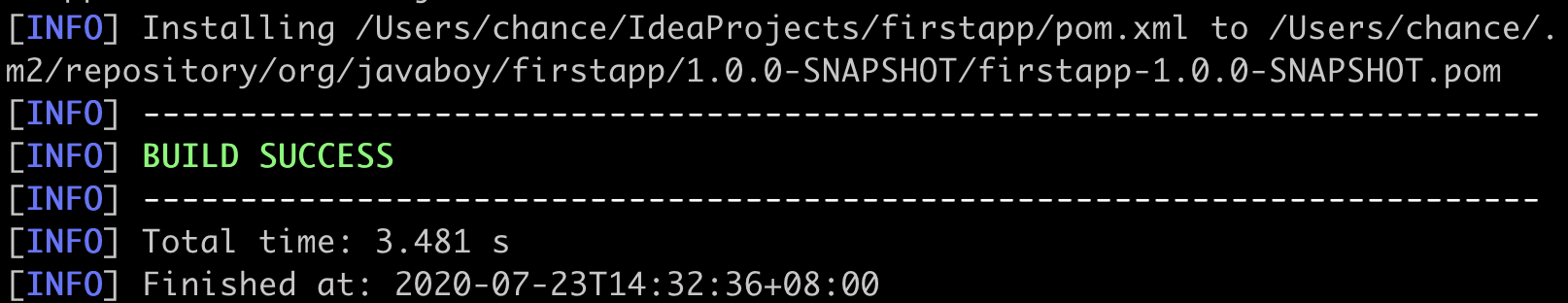
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <maven.compiler.encoding>UTF-8</maven.compiler.encoding>  
 <java.version>1.8</java.version>  
 <maven.compiler.source>1.8</maven.compiler.source>  
 <maven.compiler.target>1.8</maven.compiler.target>  
</properties>

添加完成后，执行打包命令，注意助兴所有命令时，命令行要定位到pom.xml文件所在的目录，看到如下提示，表示项目打包成功。

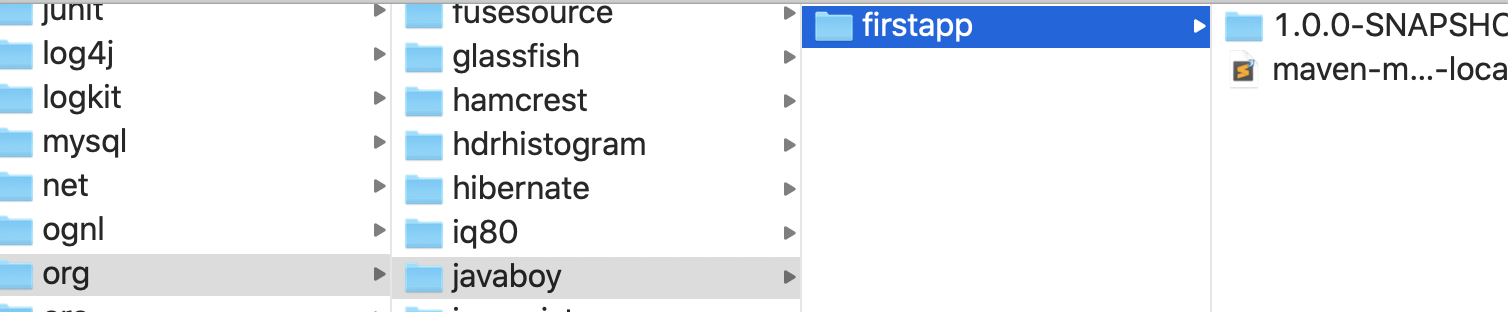


### 6、将项目安装到本地仓库

直接执行mvn install命令，注意，mvn package命令会包含上面的打包过程（mvn package）。



安装到本地仓库之后，点开自己的本地仓库，就可以看到相关的jar了。



### 7、依赖管理

Maven项目，如果需要使用第三方的控件，都是通过依赖管理来完成的。这里用到的一个东西就是pom.xml文件，概念叫做项目对象模型(POM，Project Object Model)，我们在 pom.xml 中定义了 Maven 项目的形式，所以pom.xml相当于是Maven项目的一个地图。就类似于web.xml文件用来描述三大web组件一样。

这个地图涉及哪些东西呢？

#### 7.1 Maven坐标

<dependencies>

<dependency> <!--表示一个坐标-->

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope> <!--表示依赖范围-->

</dependency>

</dependencies>

groupId

团体、公司、组织机构等等的唯一标识。团体标识的约定是它以创建这个项目的组织名称的逆向域名(例如 org.javaboy)开头。一个Maven坐标必须要包含 groupId。一些典型的groupId如apache的groupId是org.apache。

artifactId

artifactId 相当于在一个组织中项目的唯一标识符。

scope

表示依赖范围。



不同依赖的使用范围是不一样的。最典型的有两个，一个是数据库驱动，另一个是单元测试。

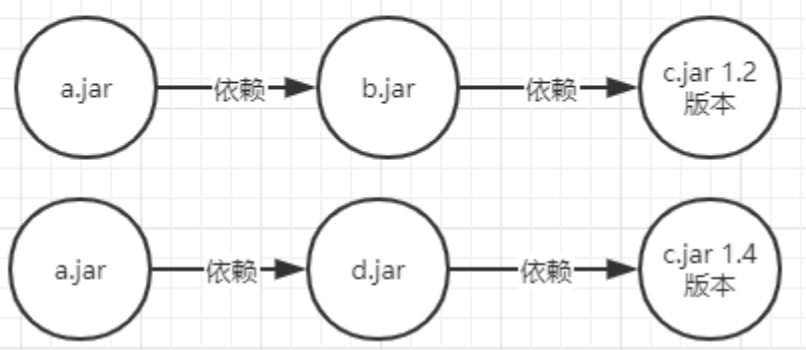
数据库驱动，在使用的过程中，我们自己写代码，写的是JDBC代码，只有在项

目运行时，才需要执行MySQL驱动中的代码。所以，MySQL驱动这个依赖在添加到项目中之后，可以设置它的scope为runtime，编译的时候不生效。

单元测试，只在测试的时候生效，所以可以设置它的scope为test，这样，当项目打包发布时，单元测试的依赖就不会跟着发布。

#### 7.2 依赖冲突

产生原因：



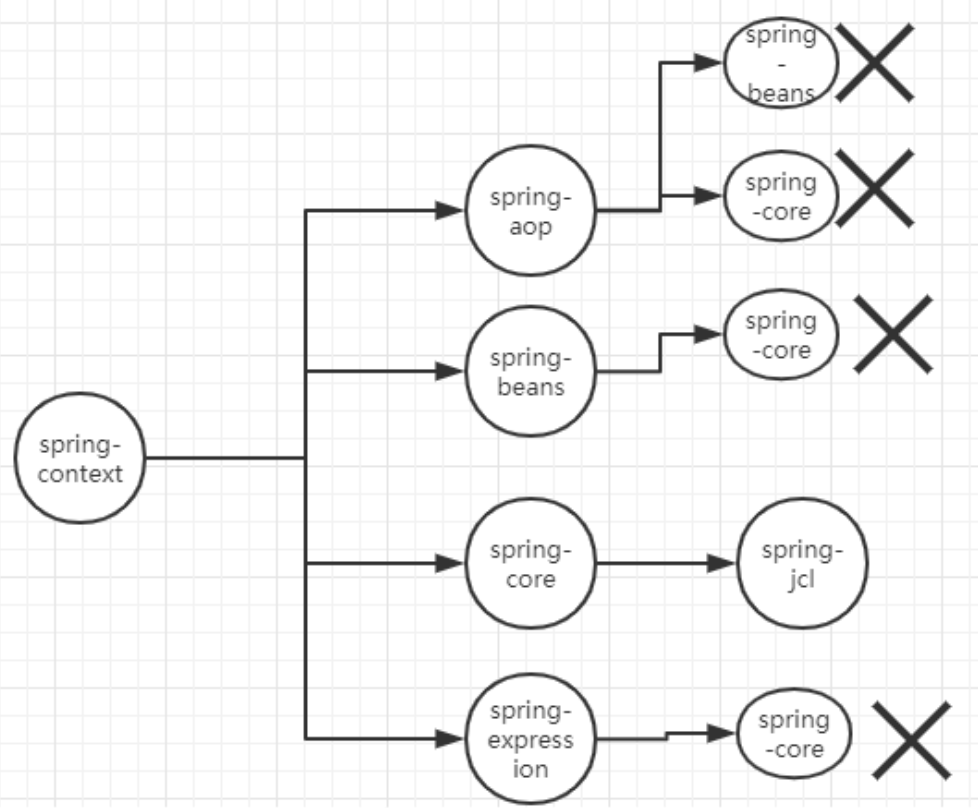
在图中，a.jar依赖b.jar，同时a.jar依赖d.jar，这个时候，a和b、d的关系是直接依赖的关系，a和c的关系是间接依赖的关系。

冲突解决：

mvn dependency:tree命令检查版本冲突。

1. 先定义先使用
2. 路径最近原则（直接声明使用）

以spring-context为例，下图中x表示失效的依赖（优先级低的依赖，即路径近的依赖优先使用）：



上面两条是默认行为，也可以手动控制，主要通过排除依赖来实现：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.9.RELEASE</version>

<exclusions>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

</exclusion>

</exclusions>

</dependency>

表示从spring-context中排除spring-core依赖。

### 8、Maven私服

Maven仓库管理也叫Maven私服或者代理仓库。使用Maven私服有两个目的：

1. 私服是一个介于开发者和远程仓库之间的代理。
2. 私服可以用来部署公司自己的jar。

#### 8.1 Nexus介绍

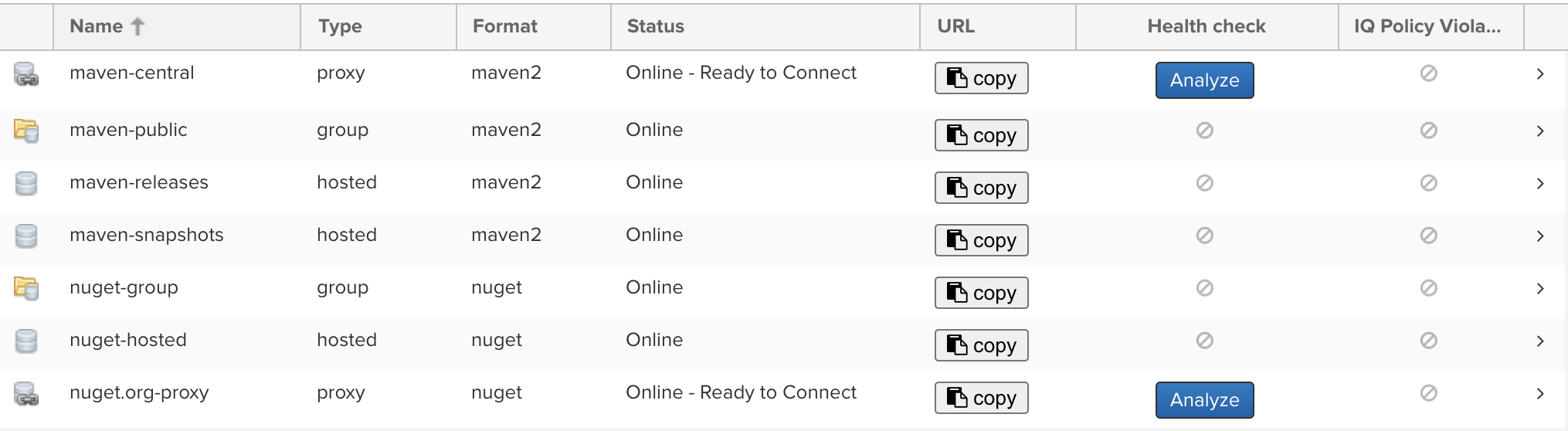
Nexus是一个强大的Maven仓库管理工具，使用Nexus可以方便的管理内部仓库同时简化外部仓库的访问。

安装之后进入bin目录，执行nexus start，启动成功后，打开管理页面

localhost:8081/nexus

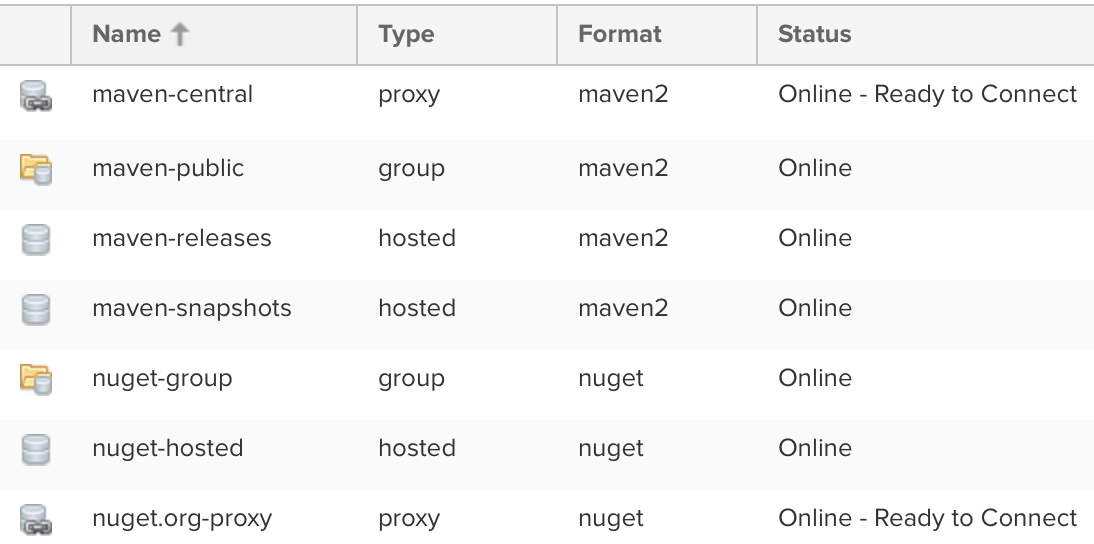
默认用户名/密码：admin/admin123

点击Repositories可以查看仓库详细信息：



* maven-releases (Version policy=Release)默认只允许上传不带SNAPSHOT版本尾缀的包，默认部署策略是Disable redeploy不允许重复上传相同版本号信息的jar，避免包版本更新以后使用方无法获取到最新的包。
* maven-snapshots (Version policy=Snapshot)只允许上传带SNAPSHOT版本尾缀的包，默认部署策略是Allow redeploy，允许重复上传相同版本号信息的jar，每次上传的时候会在jar的版本号上面增加时间后缀信息。
* maven-central中央仓库的拷贝，如果环境可以访问中央仓库，则可以获取到相关的包，否则没用
* maven-public仓库组，不是实际个一个仓库地址，只是将现有的组合到一起，可以通过它看到所属组内全部仓库的jar信息

#### 8.2 仓库类型



|  |  |
| --- | --- |
| 名称 | 说明 |
| proxy | 表示这个仓库是一个远程仓库的代理，最典型的就是代理Maven中央仓库 |
| hosted | hosted宿主仓库，公司自己开发的一些jar存放在宿主仓库中，以及一些在Maven中央仓库上没有的jar |
| group | 仓库组,包含代理仓库和宿主仓库 |
| virtual | 虚拟仓库 |

#### 8.3 上传jar

Maven的conf/settings文件配置：

<server>

<id>maven-releases</id>

<username>admin</username>

<password>admin123</password>

</server>

<server>

<id>maven-snapshots</id>

<username>admin</username>

<password>admin123</password>

</server>

要上传jar的项目的pom.xml文件中，配置上传路径：

<!--配置上传路径，用于上传jar到私服-->

<distributionManagement>  
 <repository>  
 <id>maven-releases</id>  
 <url>http://localhost:8081/repository/maven-releases/</url>  
 </repository>  
 <snapshotRepository>  
 <id>maven-snapshots</id>  
 <url>http://localhost:8081/repository/maven-snapshots/</url>  
 </snapshotRepository>  
</distributionManagement>

配置完成后，点击deploy按钮，或者执行mvn deploy命令就可以将jar上传到私服上。

#### 8.4 下载私服上的jar

直接在项目中添加依赖，添加完成后，额外增加私服地址即可：

<!--私服地址，用于下载私服上的jar-->

<repositories>  
 <repository>  
 <id>maven-public</id>  
 <url>http://localhost:8081/repository/maven-public/</url>  
 <releases>  
 <enabled>true</enabled>  
 </releases>  
 <snapshots>  
 <enabled>true</enabled>  
 </snapshots>  
 </repository>  
</repositories>

### 9、聚合工程

实际上也就是多模块项目。在一个比较大的互联网项目中，项目需要拆分成多个模块进 行开发，比如订单模块、VIP 模块、支付模块、内容管理模块、CMS、CRM 等等。这种拆分方式，实际上更 接近于微服务的思想。在一个模块中，还可以继续进行拆分，例如分成 dao、service、controller等。

有人可能会说，这个分包不就行了吗?

小项目当然可以分包，大项目就没法分包了。比如，在一个大的电商系统中，有一个子模块叫做用戶管理、还有一个子模块叫做订单管理，这两个子模块都涉及到用戶，像这种情况，我们就需要将用戶类单独提取出来，做成单独的模块，供其他模块调用。

#### 9.1 多模块项目展示

|--javaboy-parent

|-- javaboy-cms

|-- javaboy-crm

|-- javaboy-manger

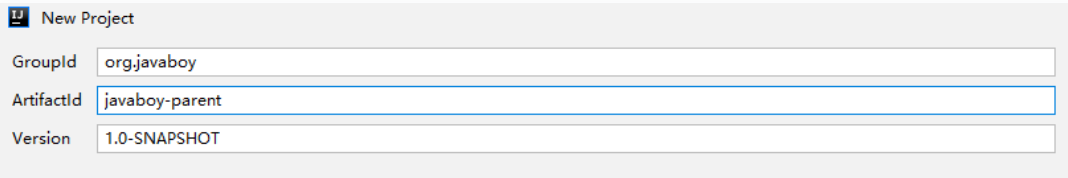
|-- javaboy-manager-model

|-- javaboy-manager-dao

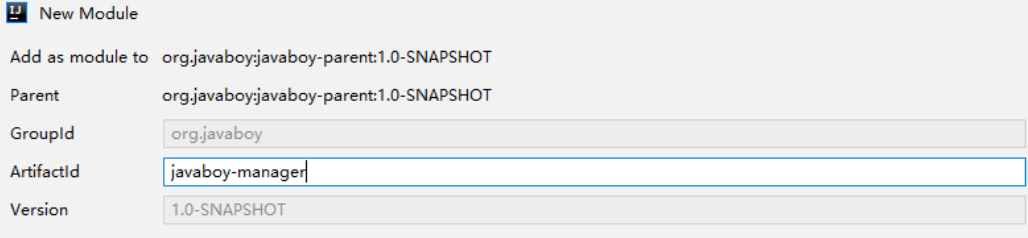
|-- javaboy-manager-service

|-- javaboy-manager-web

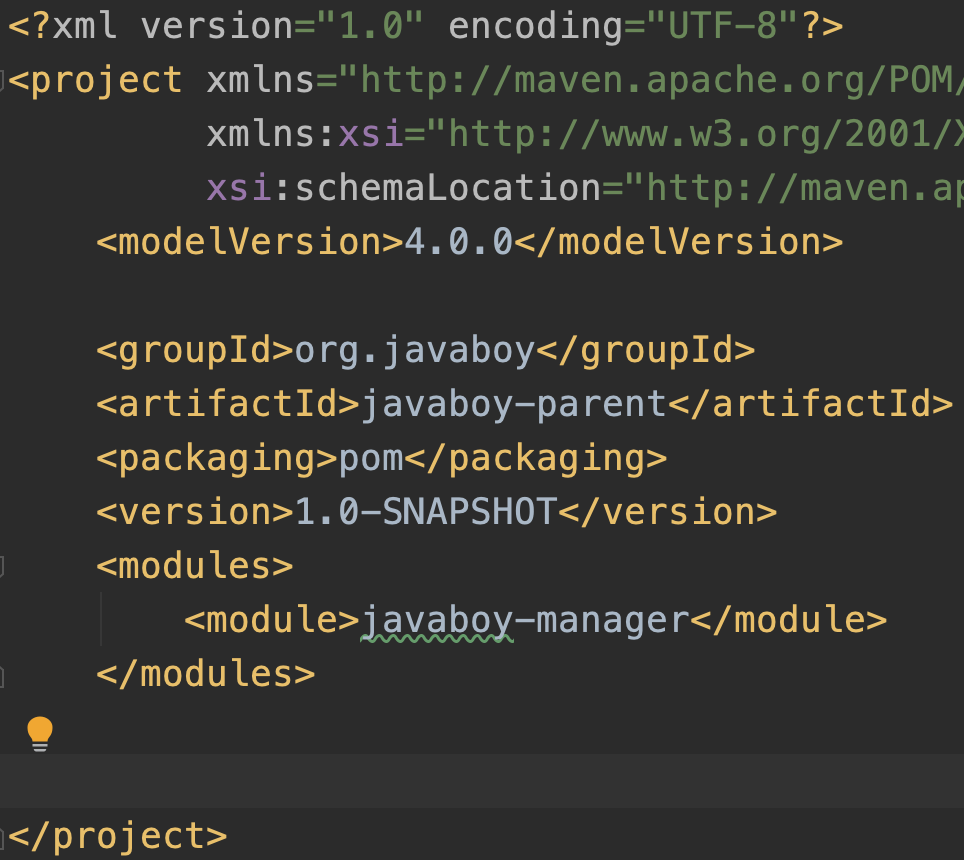
#### 9.2 IDEA创建聚合工程



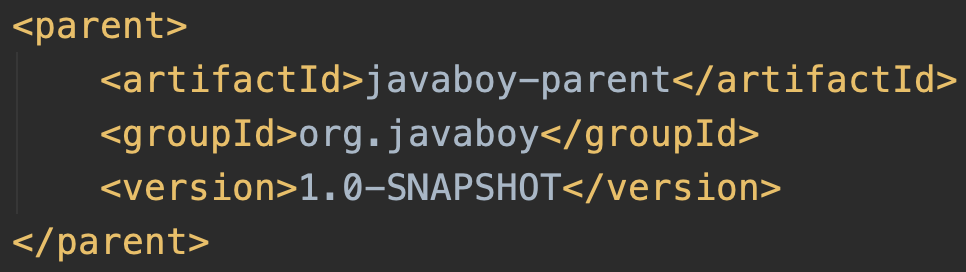
1. 创建一个空的maven项目，由于parent并不参与业务的实现，只是用来管理它的子模块，因此，src目录可以将其删除。
2. 选中当前工程，新建Module。在IDEA中已经默认指定了当前Module的parent，开发者只需要填入当前Module的artifactId即可：



javaboy-manager创建完成后，此时，观察javaboy-parent的pom.xml文件，发现它自动加上了packing属性:

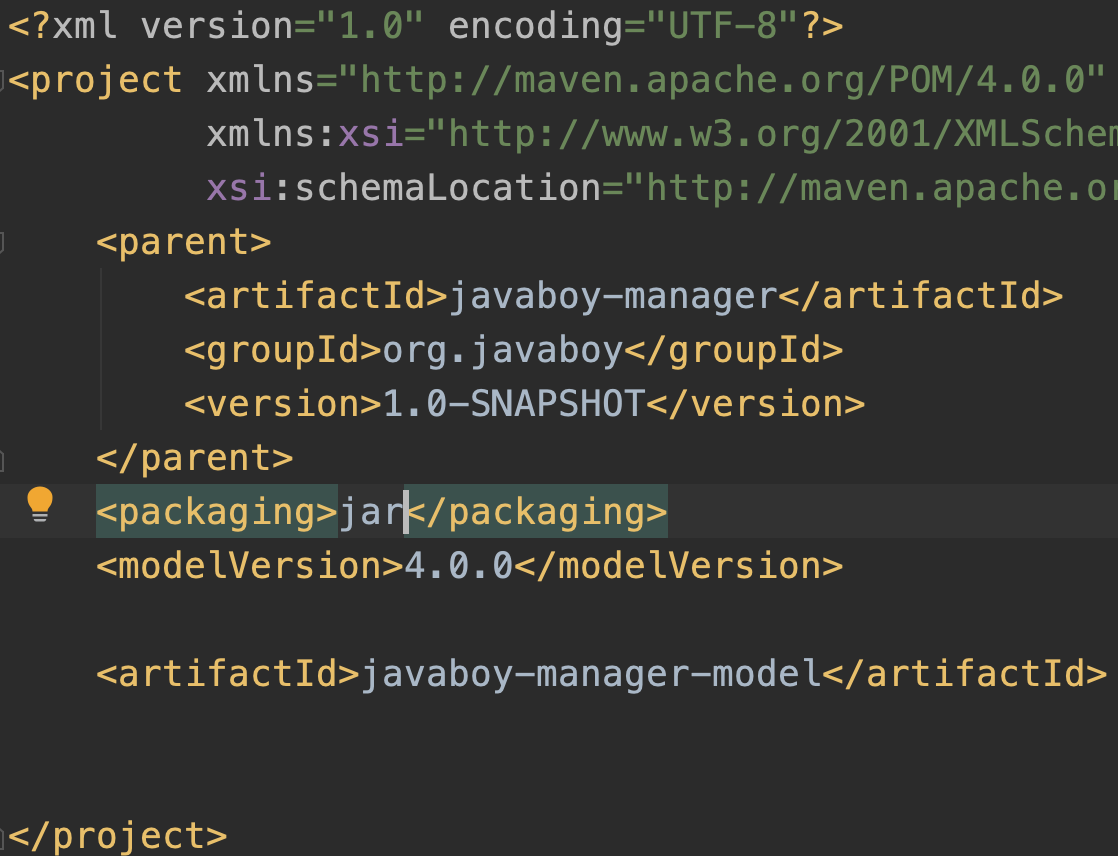


其中，它的packaging属性值为pom，这表示它是一个聚合工程，同时，他还多了modules节点，指明了它自己的子模块。同时，注意 javaboy-manager，它自身多了一个parent节点，这个parent节点描述了它的父模块的属性值：



这个parent不仅仅是一个简单的父子关系描述，它存在继承关系，一般我们可以在parent中统一定义依赖或者插件的版本号。

1. 由于javaboy-manager本身也是一个聚合工程，因此，javaboy-manager的src也可以删除。
2. 选中javaboy-manager，右键单击，New->Module创建一个新的Maven模块出来。这里，新的javaboy-manager-model创建成功后，我们手动配置它的packaging属性值为jar。



1. 依照第4）步，再分别创建javaboy-manager-service以及javaboy-manager-dao
2. 继续创建javaboy-manager-web模块，不同于其他模块，web模块需要打包成war。web模块创建可以。
3. web工程创建完成后，完善模块之间的继承关系。

javaboy-manager-web 依赖 javaboy-manager-service javaboy-manager-service 依赖 javaboy-manager-dao javaboy-manager-dao 依赖 javaboy-manager-model

注意，依赖默认是有传递性的，即在 javaboy-manager-dao 中依赖了 javaboy-manager- model，在 javaboy-manager-service 也能访问到。

有一个需要注意的地方，在多模块项目中，web 项目打包需要注意以下问题:

1. 不可以直接单独打包

2. 如果要打包，有两种方式：

* 第一种就是先手动挨个将model、dao、service安装到本地仓库(mvn install)
* 从聚合工程处打包，即从web的parent处打包。

### 10、常用Maven插件

Maven本质上是一个插件框架，它的核心并不执行任何具体的构建任务，所有这些任务都交给插件来完成，例如编译源代码是由maven-compiler-plugin完成的。maven-compiler-plugin的compile目标用来编译位于src/main/java/目录下的主源码，testCompile目标用来编译位于src/test/java/目录下的测试源码。

用戶可以通过两种方式调用Maven插件目标。

* 第一种方式是将插件目标与生命周期阶段(lifecycle phase)绑定，这样用户在命令行只是输入生命周期阶段而已，例如Maven默认将maven-compiler-plugin的compile目标与compile生命周期阶段绑定，因此命令mvn compile实际上是先定位到compile这一生命周期阶段，然后再根据绑定关系调用maven-compiler-plugin的compile目标。
* 第二种方式是直接在命令行指定要执行的插件目标，例如mvn archetype:generate就表示调用maven-archetype-plugin的generate目标，这种带冒号的调用方式与生命周期无关。

认识上述Maven插件的基本概念能帮助你理解Maven的工作机制，不过要想更高效率地使用Maven，了解一些常用的插件还是很有必要的，这可以帮助你避免一不小心重新发明轮子。Maven官方有两个插件列表，第一个列表的GroupId为org.apache.maven.plugins，这里的插件最为成熟，具体地址为: http://maven.apache.org/plugins/index.html。第二个列表的GroupId为org.codehaus.mojo，这里的插件没有那么核心， 但也有不少十分有用，其地址为:http://mojo.codehaus.org/plugins.html。

#### 10.1 maven-antrun-plugin

maven-antrun-plugin能让用戶在Maven项目中运行Ant任务。用戶可以直接在该插件的配置以Ant的方式编写Target，然后交给该插件的run目标去执行。在一些由Ant往Maven迁移的项目中，该插件尤其有用。此外当你发现需要编写一些自定义程度很高的任务，同时又觉得Maven不够灵活时，也可以以Ant的方式实现之。maven-antrun-plugin的run目标通常与生命周期绑定运行。

#### 10.2 maven-archetype-plugin

archetype指项目的骨架，Maven初学者最开始执行的Maven命令可能就是mvn archetype:generate，就是让maven-archetype-plugin生成一个很简单的项目骨架。generate目标使用交互式的方式提示用戶输入必要的信息以创建项 目。maven-archetype-plugin还有一些其他目标帮助用戶自己定义项目原型，例如你由一个产品需要交付给很多客户进行二次开发，你就可以为他们提供一个archetype，帮助他们快速上手。

#### 10.3 maven-assembly-plugin

maven-assembly-plugin的用途是制作项目分发包，该分发包可能包含了项目的可执行文件、源代码、readme、平台脚本等等。maven-assembly-plugin支持各种主流的格式如zip、tar.gz、jar和war等，具体打包哪些文件是高度可控的，例如用户可以按文件级别的粒度、文件集级别的粒度、模块级别的粒度、以及依赖级别的粒度控制打包，此外，包含和排除配置也是支持的。maven-assembly-plugin要求用户使用一个名为assembly.xml的元数据文件来表述打包，它的single目标可以直接在命令行调用，也可以被绑定至生命周期。

#### 10.4 maven-dependency-plugin

maven-dependency-plugin最大的用途是帮助分析项目依赖，dependency:list能够列出项目最终解析到的依赖列表，dependency:tree能进一步的描绘项目依赖树，dependency:analyze可以告诉你项目依赖潜在的问题，如果你有直接使用到的却未声明的依赖，该目标就会发出警告。maven-dependency-plugin还有很多目标帮助你操作依赖文件，例如dependency:copy-dependencies能将项目依赖从本地Maven仓库复制到某个特定的文件夹下面。

#### 10.5 maven-enforcer-plugin

在一个稍大一点的组织或团队中，你无法保证所有成员都熟悉Maven，那他们做一些比较愚蠢的事情就会变得很正常，例如给项目引入了外部的SNAPSHOT依赖而导致构建不稳定，使用了一个与大家不一致的Maven版本而经常抱怨构建出现诡异问题。maven-enforcer-plugin允许你创建一系列规则强制大家遵守，包括设定Java版本、设定Maven 版本、禁止某些依赖、禁止SNAPSHOT依赖。只要在一个父POM配置规则，然后让大家继承，当规则遭到破坏的时候，Maven就会报错。除了标准的规则之外，你还可以扩展该插件，编写自己的规则。maven-enforcer-plugin的enforce目标负责检查规则，它默认绑定到生命周期的validate阶段。

#### 10.6 maven-release-plugin

maven-release-plugin的用途是帮助自动化项目版本发布，它依赖于POM中的SCM信息。release:prepare用来准备版本发布，具体的工作包括检查是否有未提交代码、检查是否有SNAPSHOT依赖、升级项目的SNAPSHOT版本至RELEASE版本、为项目打标签等等。release:perform则是签出标签中的RELEASE源码，构建并发布。版本发布是非常琐碎的工作，它涉及了各种检查，而且由于该工作仅仅是偶尔需要，因此手动操作很容易遗漏一些细节。maven-release-plugin的各种目标通常直接在命令行调用，因为版本发布显然不是日常构建生命周期的一部分。

#### 10.7 maven-resources-plugin

为了使项目结构更为清晰，Maven区别对待Java代码文件和资源文件，maven-compiler-plugin用来编译Java代码，maven- resources-plugin则用来处理资源文件。默认的主资源文件目录是src/main/resources，很多用户会需要添加额外的资源文件目录，这个时候就可以通过配置maven-resources-plugin来实现。此外，资源文件过滤也是Maven的一大特性，你可以在资源文件中使用${propertyName}形式的Maven属性，然后配置maven-resources-plugin开启对资源文件的过滤，之后就可以针对不同环境通过命令行或者Profile传入属性的值，以实现更为灵活的构建。

#### 10.8 maven-surefire-plugin

可能是由于历史的原因，Maven 2/3中用于执行测试的插件不是maven-test-plugin，而是maven-surefire-plugin。其实大部分时间内，只要你的测试类遵循通用的命令约定(以Test结尾、以TestCase结尾、或者以Test开头)，就几乎不用知晓该插件的存在。然而在当你想要跳过测试、排除某些测试类、或者使用一些TestNG特性的时候，了解maven-surefire-plugin的一些配置选项就很有用了。例如mvn test -Dtest=FooTest 这样一条命令的效果是仅运行FooTest测试类，这是通过控制maven-surefire-plugin的test参数实现的。

#### 10.9 jetty-maven-plugin

在进行Web开发的时候，打开浏览器对应用进行手动的测试几乎是无法避免的，这种测试方法通常就是将项目打包成war文件，然后部署到Web容器中，再启动容器进行验证，这显然十分耗时。为了帮助开发者节省时间，jetty-maven-plugin应运而生，它完全兼容 Maven项目的目录结构，能够周期性地检查源文件，一旦发现变更后自动更新到内置的Jetty Web容器中。做一些基本 配置后(例如Web应用的contextPath和自动扫描变更的时间间隔)，你只要执行mvn jetty:run ，然后在IDE中修改代码，代码经IDE自动编译后产生变更，再由jetty-maven-plugin侦测到后更新至Jetty容器，这时你就可以直接 测试Web页面了。需要注意的是，jetty-maven-plugin并不是宿主于Apache或Codehaus的官方插件，因此使用的时候需要额外的配置settings.xml的 pluginGroups元素，将org.mortbay.jetty这个pluginGroup加入。

#### 10.10 versions-maven-plugin

很多Maven用户遇到过这样一个问题，当项目包含大量模块的时候，为他们集体更新版本就变成一件烦人的事情，到底有没有自动化工具能帮助完成这件 事情呢?(当然你可以使用sed之类的文本操作工具)答案是肯定的，versions-maven-plugin提供了很多目标帮助你管理Maven项目的各种版本信息。例如最常用的，命令mvn versions:set - DnewVersion=1.1-SNAPSHOT就能帮助你把所有模块的版本更新到1.1-SNAPSHOT。该插件还提供了其他一些很有用的目 标，display-dependency- updates能告诉你项目依赖有哪些可用的更新;类似的display-plugin-updates能告诉你可用的插件

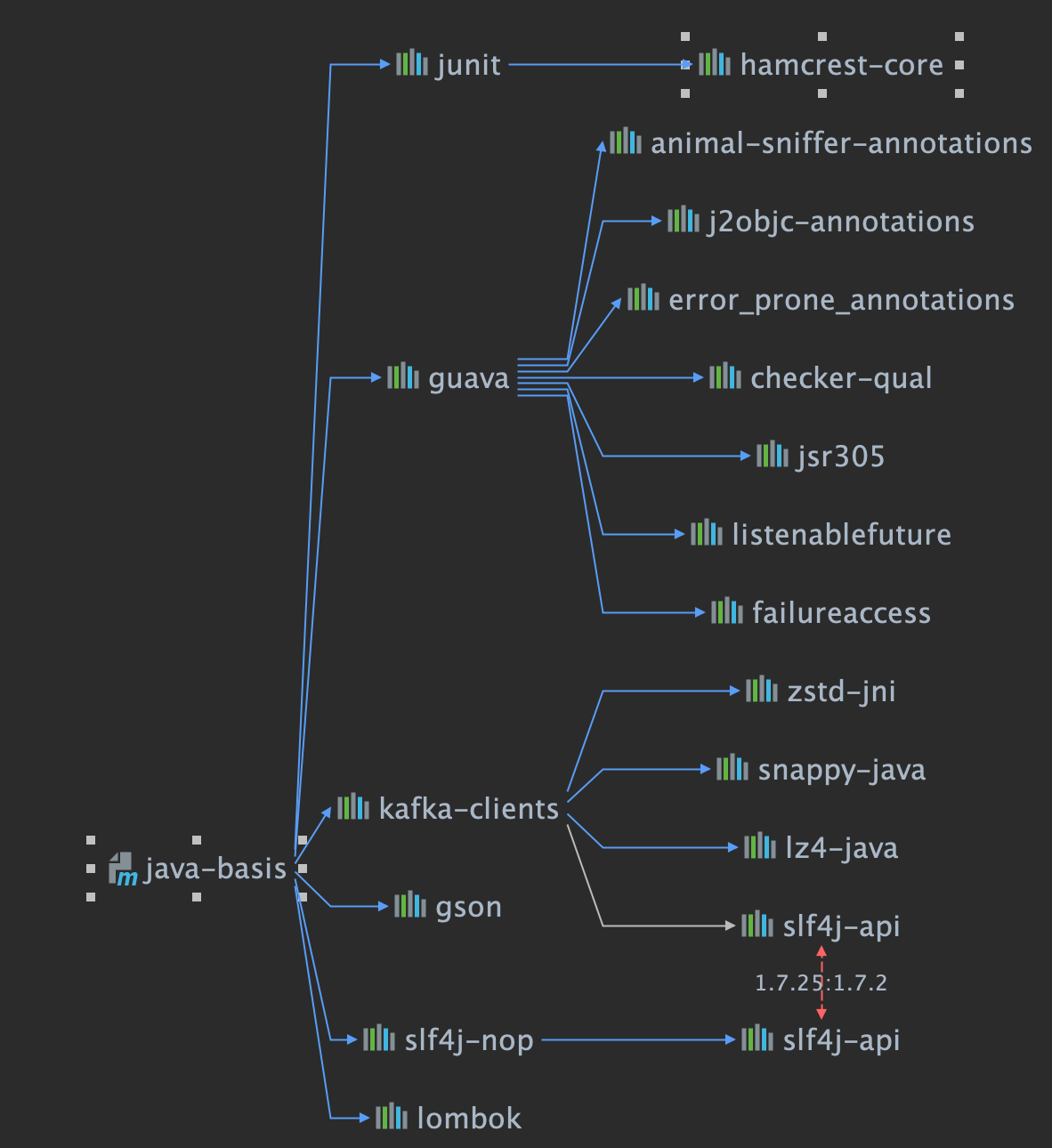
更新;然后use- latest-versions能自动帮你将所有依赖升级到最新版本。最后，如果你对所做的更改满意，则可以使用mvn versions:commit提交，不满意的话也可以使用mvn versions:revert进行撤销。

### 11、通过IDEA Maven Helper插件

在插件安装好之后，我们打开pom.xml文件，在底部会多出一个Dependency Analyzer选项。

找到冲突后，点击右键，选择我Exclude即可排除冲突版本的jar包。（同一个jar包可能需要执行多次Exclude，执行一次之后需要refresh以下，才能确定是否有新的冲突）

可以使用IDEA提供的方法——maven依赖结构图，打开maven窗口，选择Dependencies，然后点击那个图标（show dependencies），即可打开maven依赖结构图。



可以看到有一些红色的实线，这些红色实线就是依赖冲突，蓝色实线则是正常的依赖。