**Maven**

## 基础概念

**1 Maven 是一个项目管理工具，它负责管理项目开发过程中的几乎所有的东西。**

**（1）版本**

maven有自己的版本定义和规则

**（2）构建**

maven支持许多种的应用程序类型，对于每一种支持的应用程序类型都定义好了一组构建规则和工具集。

**（3）输出物管理**

maven可以管理项目构建的产物，并将其加入到用户库中。这个功能可以用于项目组和其他部门之间的交付行为

**（4）依赖关系**

maven对依赖关系的特性进行细致的分析和划分，避免开发过程中的依赖混乱和相互污染行为

**（5）文档和构建结果**

maven的site命令支持各种文档信息的发布，包括构建过程的各种输出，javadoc，产品文档等。

**（6）项目关系**

一个大型的项目通常有几个小项目或者模块组成，用maven可以很方便地管理

**（7）移植性管理**

maven可以针对不同的开发场景，输出不同种类的输出结果

我们在开发项目的过程中，会使用一些开源框架、第三方的工具等等，这些都是以jar包的方式被项目所引用，并且有些jar包还会依赖其他的jar包，我们同样需要添加到项目中，所有这些相关的jar包都会作为项目的依赖。通常，一个Java EE项目所依赖的jar包会有很多。然而，这还并非是主要问题，**在管理这些jar包过程中，jar包的版本往往是最令人头疼的问题。选择一个jar包的版本，需要考虑它所依赖的jar包是否支持这个版本，并且还需要确认依赖它的jar包能不能对这个版本兼容。**所以，在过去的构建项目过程中，整理出这么一堆能让项目正常运行的jar包之后，这个lib目录就会变为禁区。jar包版本更新了，我们也很少会自找麻烦去触碰它。至于是不是存在冗余的jar包？能正常运行就好了嘛。。。。

Maven的出现，解决了开发过程中的这一难题。它可以对项目依赖的jar包进行管理，可以让你的项目保持基本的依赖，排除冗余jar包，并且可以让你非常轻松的对依赖的jar包进行版本升级。而这些仅仅是Maven最基本的功能，它可以在这基础上对项目进行清理、编译、测试、打包、发布等等构建项目的工作。

**Maven是现在Java社区中最强大的项目管理和项目构建工具**，而更加值得庆幸的是，这样一个强大的工具，它的使用也是非常简单的。现在，JavaEE项目使用的开源软件都可以通过Maven来获取，并且，越来越多的公司也开始使用Maven来管理构建项目了

**2 Maven的生命周期**

maven把项目的构建划分为不同的生命周期(lifecycle)。粗略一点的话，它这个过程(phase)包括：编译、测试、打包、集成测试、验证、部署。maven中所有的执行动作(goal)都需要指明自己在这个过程中的执行位置，然后maven执行的时候，就依照过程的发展依次调用这些goal进行各种处理。这个也是maven的一个基本调度机制。一般来说，位置稍后的过程都会依赖于之前的过程。当然，maven同样提供了配置文件，可以依照用户要求，跳过某些阶段。

**Maven(翻译为"专家"，"内行")是跨平台的项目管理工具。主要服务于基于Java平台的项目构建，依赖管理和项目信息管理。**

（1）项目构建

项目构建过程包括**【清理项目】→【编译项目】→【测试项目】→【生成测试报告】→【打包项目】→【部署项目】**这几个步骤，这六个步骤就是一个项目的完整构建过程。

理想的项目构建是高度自动化，跨平台，可重用的组件，标准化的，使用maven就可以帮我们完成上述所说的项目构建过程。

（2）依赖管理

依赖指的是jar包之间的相互依赖，比如我们搭建一个Struts2的开发框架时，光光有struts2-core-2.3.16.3.jar这个jar包是不行的，struts2-core-2.3.16.3.jar还依赖其它的jar包，依赖管理指的就是使用Maven来管理项目中使用到的jar包，Maven管理的方式就是“自动下载项目所需要的jar包，统一管理jar包之间的依赖关系”。

（3）使用Maven的好处

@Maven中使用约定，约定java源代码代码必须放在哪个目录下，编译好的java代码又必须放到哪个目录下，这些目录都有明确的约定。

@Maven的每一个动作都拥有一个生命周期，例如执行 mvn install 就可以自动执行编译，测试，打包等构建过程

@只需要定义一个pom.xml,然后把源码放到默认的目录，Maven帮我们处理其他事情

使用Maven可以进行项目高度自动化构建，依赖管理(这是使用Maven最大的好处)，仓库管理。

**3 Maven的标准工程结构**

Maven的标准工程结构如下

|-- pom.xml(maven的核心配置文件)

|-- src

|-- main

|   `-- java(java源代码目录)

|   `-- resources(资源文件目录)

|-- test

        `-- java(单元测试代码目录)

|-- target(输出目录，所有的输出物都存放在这个目录下)

    |--classes(编译后的class文件存放处)

**maven profile：根据需要，可以在以下文件声明profile。**

1、pom.xml 针对当前项目

2、用户 settings.xml 用户目录下的.m2/settings.xml， 对当前用户的所有项目有效。

3、全局 settings.xml 即maven安装目录下的conf/settings.xml。对本机上的所有项目有效。

**4 Maven的约定优于配置**

所谓的"约定优于配置"，在maven中并不是完全不可以修改的，他们只是一些配置的默认值而已。但是除非必要，并不需要去修改那些约定内容。maven默认的文件存放结构如下：

每一个阶段的任务都知道怎么正确完成自己的工作，比如compile任务就知道从src/main/java下编译所有的java文件，并把它的输出class文件存放到target/classes中。

对maven来说，采用"约定优于配置"的策略可以减少修改配置的工作量，也可以降低学习成本，更重要的是，给项目引入了统一的规范。

**5 Maven的版本规范**

maven使用如下几个要素来唯一定位某一个输出物

@groudId

团体、组织的标识符。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(reverse domain name)开头。一般对应着JAVA的包的结构。例如org.apache

@artifactId

单独项目的唯一标识符。比如我们的tomcat, commons等。不要在artifactId中包含点号(.)。

@version

一个项目的特定版本。

@packaging

项目的类型，默认是jar，描述了项目打包后的输出。类型为jar的项目产生一个JAR文件，类型为war的项目产生一个web应用。

maven有自己的版本规范，一般是如下定义

<major version>.<minor version>.<incremental version>-<qualifier>

比如1.2.3-beta-01。要说明的是，maven自己判断版本的算法是major,minor,incremental部分用数字比较，qualifier部分用字符串比较，所以要小心 alpha-2和alpha-15的比较关系，最好用 alpha-02的格式。

**maven在版本管理时候可以使用几个特殊的字符串 SNAPSHOT，LATEST，RELEASE。比如"1.0-SNAPSHOT"**。各个部分的含义和处理逻辑如下说明

**SNAPSHOT**

这个版本一般用于开发过程中，表示不稳定的版本。

**LATEST**

指某个特定构件的最新发布，这个发布可能是一个发布版，也可能是一个snapshot版，具体看哪个时间最后。

**RELEASE**

指最后一个发布版。

## 二、Maven安装

**1 官网下载地址**

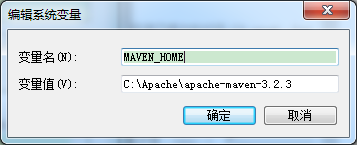
<http://maven.apache.org/download.cgi>

**2 配置环境变量**

注意：安装maven之前，必须先确保你的机器中已经安装了JDK。

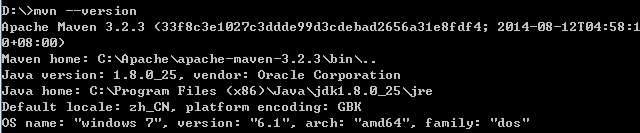
（1）．解压压缩包（以apache-maven-3.3.9-bin.zip为例）

（2）．添加环境变量MAVEN\_HOME，值为apache-maven-3.3.9的安装路径



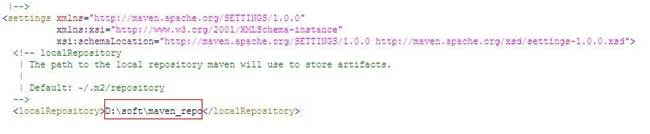
（3）．在Path环境变量的变量值末尾添加%MAVEN\_HOME%\bin

（4）．在cmd输入mvn –version，如果出现maven的版本信息，说明配置成功。



**3 本地仓储配置**

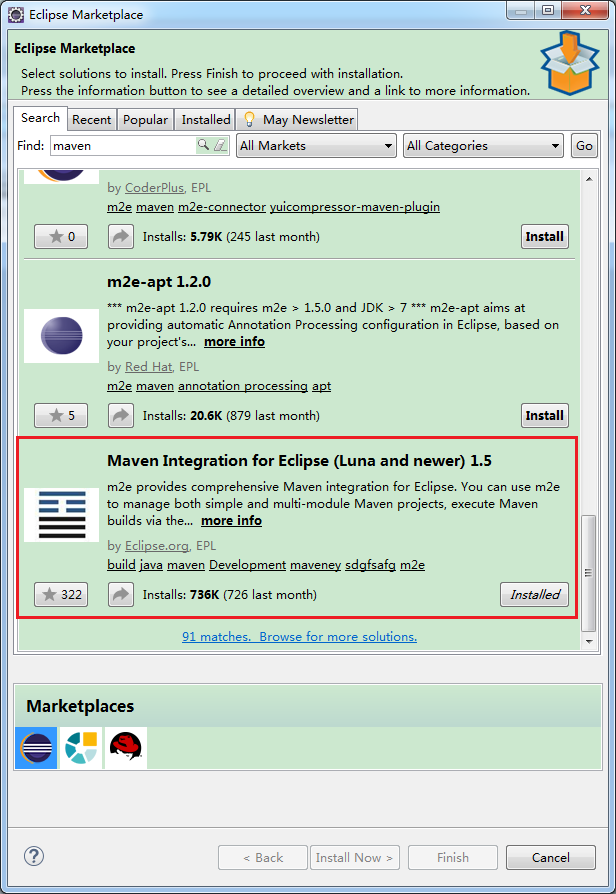
从中央仓库下载的jar包，都会统一存放到本地仓库中。我们需要配置本地仓库的位置。打开maven安装目录，打开conf目录下的**setting.xml**文件。可以参照下图配置本地仓储位置。

[](http://images2015.cnblogs.com/blog/318837/201606/318837-20160616161804495-2096239249.png)

**4 第一个Maven工程**

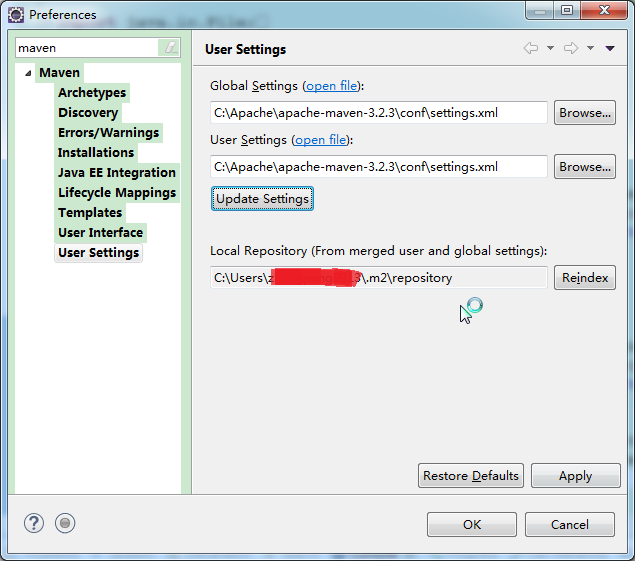
（1）Eclipse中创建Maven工程，需要安装Maven插件。

一般较新版本的Eclipse都会带有Maven插件，如果你的Eclipse中已经有Maven插件，可以跳过这一步骤。点击Help -> Eclipse Marketplace，搜索maven关键字，选择安装红框对应的Maven插件。



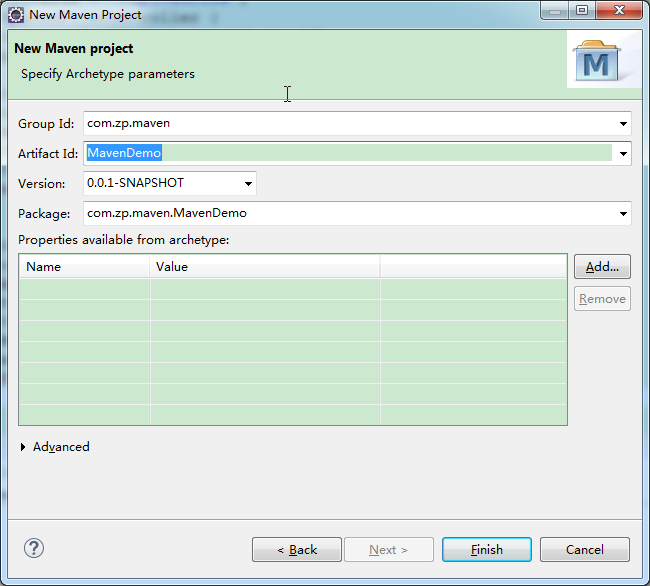
（2）Maven环境配置

点击Window -> Preferences，如下图所示，配置settings.xml文件的位置



（3）创建Maven工程

File -> New -> Maven Project -> Next，在接下来的窗口中会看到一大堆的项目模板，选择合适的模板。接下来设置项目的参数，如下：

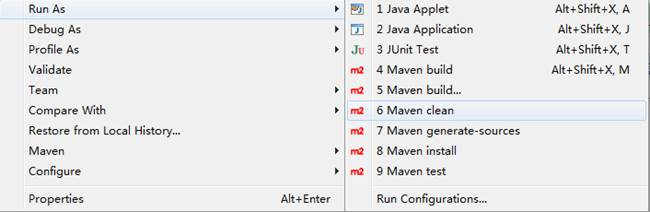
[](http://images2015.cnblogs.com/blog/318837/201606/318837-20160616161807151-1146450061.png)

groupId是项目组织唯一的标识符，实际对应JAVA的包的结构，是main目录里java的目录结构。artifactId就是项目的唯一的标识符，实际对应项目的名称，就是项目根目录的名称。点击Finish，Eclipse会创建一个Maven工程。

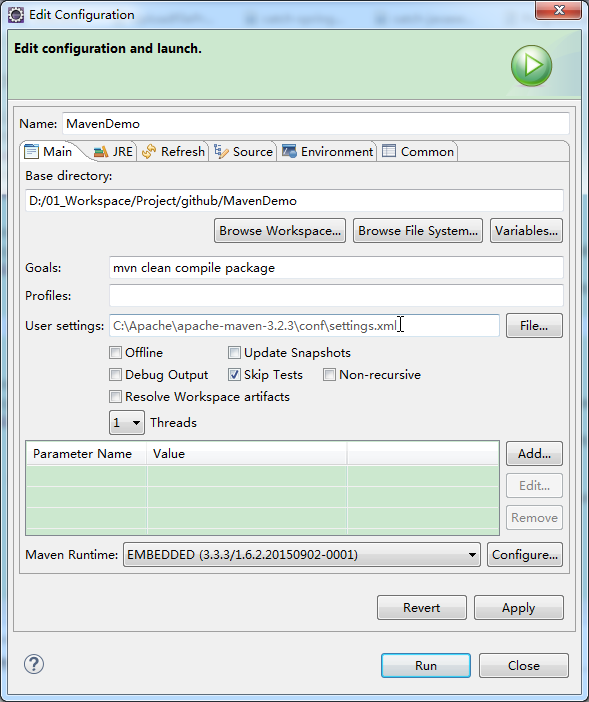
**5 使用Maven进行构建**

（1）Eclipse中构建方式

在Elipse项目上右击 -> Run As 就能看到很多Maven操作。这些操作和maven命令是等效的。例如Maven clean，等同于mvn clean命令。

[](http://images2015.cnblogs.com/blog/318837/201606/318837-20160616161808151-462649213.png)

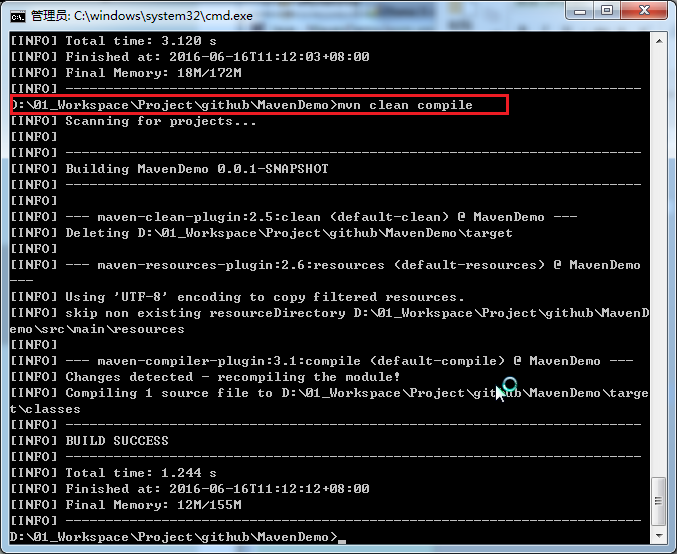
你也可以点击Maven build，输入组合命令，并保存下来。如下图：



（2）Maven命令构建方式

当然，你也可以直接使用maven命令进行构建。

进入工程所在目录，输入maven命令就可以了。

[](http://images2015.cnblogs.com/blog/318837/201606/318837-20160616161810010-960277526.png)

## 三、使用指导

**1 如何添加外部依赖jar包**

在Maven工程中添加依赖jar包，很简单，只要在POM文件中引入对应的<dependency>标签即可。参考下例：

|  |
| --- |
| <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>com.zp.maven</groupId>    <artifactId>MavenDemo</artifactId>    <version>0.0.1-SNAPSHOT</version>    <packaging>jar</packaging>    <name>MavenDemo</name>    <url>http://maven.apache.org</url>      <properties>      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>      <junit.version>3.8.1</junit.version>    </properties>      <dependencies>      <dependency>        <groupId>junit</groupId>        <artifactId>junit</artifactId>        <version>${junit.version}</version>        <scope>test</scope>      </dependency>      <dependency>        <groupId>log4j</groupId>        <artifactId>log4j</artifactId>        <version>1.2.12</version>        <scope>compile</scope>      </dependency>    </dependencies>  </project> |

<dependency>标签最常用的四个属性标签：

groupId：项目组织唯一的标识符，实际对应JAVA的包的结构。

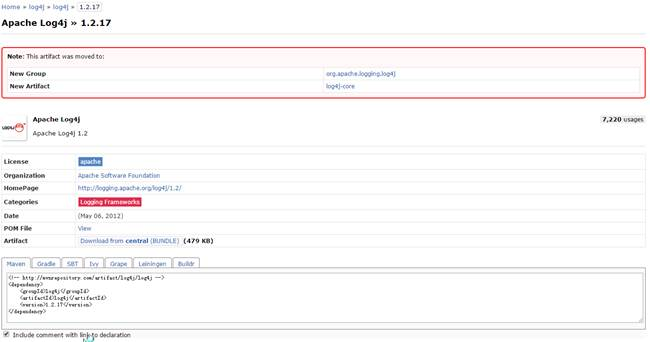
artifactId：项目唯一的标识符，实际对应项目的名称，就是项目根目录的名称。

version：jar包的版本号。可以直接填版本数字，也可以在properties标签中设置属性值。

scope：jar包的作用范围。可以填写compile、runtime、test、system和provided。用来在编译、测试等场景下选择对应的classpath。

**2 如何寻找jar包**

可以在<http://mvnrepository.com/>站点搜寻你想要的jar包版本。例如，想要使用log4j，可以找到需要的版本号，然后拷贝对应的maven标签信息，将其添加到pom .xml文件中。

[](http://images2015.cnblogs.com/blog/318837/201606/318837-20160616161810807-1939284959.png)

**3 如何使用Maven插件(Plugin)**

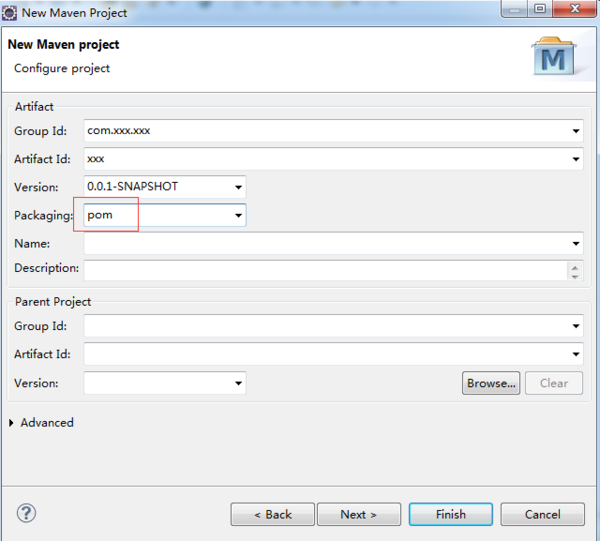
要添加Maven插件，可以在pom.xml文件中添加<plugin>标签。

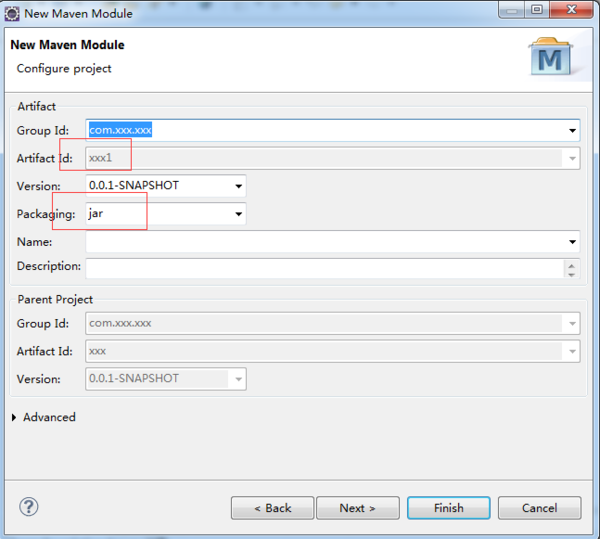
|  |
| --- |
| <build>    <plugins>      <plugin>        <groupId>org.apache.maven.plugins</groupId>        <artifactId>maven-compiler-plugin</artifactId>        <version>3.3</version>  <configuration>          <source>1.7</source>          <target>1.7</target>        </configuration>      </plugin>    </plugins>  </build> |

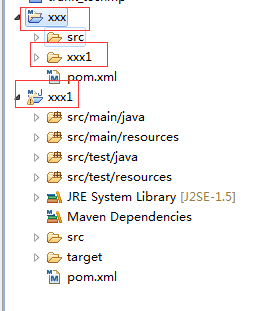
<configuration>标签用来配置插件的一些使用参数。

**4 如何一次编译多个工程**

在Maven中，允许一个Maven Project中有多个Maven Module

（1）创建maven父工程步骤：new-->other-->选择maven project-->next-->勾选create a simple project-->next-->填写Group Id、Artifact Id、Version --> packaging选择pom-->finish。  
[](http://d.hiphotos.baidu.com/zhidao/pic/item/6609c93d70cf3bc71422ca23d200baa1cc112ada.jpg" \o "点击查看大图" \t "https://www.cnblogs.com/jingmoxukong/p/_blank)

（2）.创建maven子工程步骤：选中刚才创建的父工程右键-->new-->other-->选择maven module-->next-->勾选create a simple project-->填写module name（其实就是artifact id）-->next-->GAV继承父工程-->packaging选择你需要的-->finish。  
[](http://a.hiphotos.baidu.com/zhidao/pic/item/d31b0ef41bd5ad6e7dae622582cb39dbb7fd3cad.jpg" \o "点击查看大图" \t "https://www.cnblogs.com/jingmoxukong/p/_blank)

（3）完成，刷新父工程；如有多个子工程，继续按照第二步骤创建。  
[](http://e.hiphotos.baidu.com/zhidao/pic/item/2fdda3cc7cd98d1045d10afe223fb80e7aec90f3.jpg)

这时打开XXX中的pom.xml可以看到其中有以下标签

<modules>

  <module>xxx1</module>

</modules>

选择编译XXX时，会依次对它的所有Module执行相同操作。

**5 常用Maven插件**

**maven-antrun-plugin**

<http://maven.apache.org/plugins/maven-antrun-plugin/>

maven-antrun-plugin能让用户在Maven项目中运行Ant任务。用户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。

**maven-archetype-plugin**

<http://maven.apache.org/archetype/maven-archetype-plugin/>

Archtype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，这实际上就是让maven-archetype-plugin生成一个很简单的项目骨架，帮助开发者快速上手。可能也有人看到一些文档写了mvn archetype:create，但实际上create目标已经被弃用了，取而代之的是generate目标，该目标使用交互式的方式提示用户输入必要的信息以创建项目，体验更好。 maven-archetype-plugin还有一些其他目标帮助用户自己定义项目原型，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Archtype，帮助他们快速上手。

**maven-assembly-plugin**

<http://maven.apache.org/plugins/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。

**maven-dependency-plugin**

<http://maven.apache.org/plugins/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。

**maven-enforcer-plugin**

<http://maven.apache.org/plugins/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能够帮助你避免之类问题，它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。

**maven-help-plugin**

<http://maven.apache.org/plugins/maven-help-plugin/>

maven-help-plugin是一个小巧的辅助工具，最简单的help:system可以打印所有可用的环境变量和Java系统属性。help:effective-pom和help:effective-settings最为有用，它们分别打印项目的有效POM和有效settings，有效POM是指合并了所有父POM（包括Super POM）后的XML，当你不确定POM的某些信息从何而来时，就可以查看有效POM。有效settings同理，特别是当你发现自己配置的 settings.xml没有生效时，就可以用help:effective-settings来验证。此外，maven-help-plugin的describe目标可以帮助你描述任何一个Maven插件的信息，还有all-profiles目标和active-profiles目标帮助查看项目的Profile。

**maven-release-plugin**

<http://maven.apache.org/plugins/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让该工作变得非常快速简便，不易出错。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。

**maven-resources-plugin**

<http://maven.apache.org/plugins/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。

**maven-surefire-plugin**

<http://maven.apache.org/plugins/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。

**build-helper-maven-plugin**

<http://mojo.codehaus.org/build-helper-maven-plugin/>

Maven默认只允许指定一个主Java代码目录和一个测试Java代码目录，虽然这其实是个应当尽量遵守的约定，但偶尔你还是会希望能够指定多个源码目录（例如为了应对遗留项目），build-helper-maven-plugin的add-source目标就是服务于这个目的，通常它被绑定到默认生命周期的generate-sources阶段以添加额外的源码目录。需要强调的是，这种做法还是不推荐的，因为它破坏了 Maven的约定，而且可能会遇到其他严格遵守约定的插件工具无法正确识别额外的源码目录。

build-helper-maven-plugin的另一个非常有用的目标是attach-artifact，使用该目标你可以以classifier的形式选取部分项目文件生成附属构件，并同时install到本地仓库，也可以deploy到远程仓库。

**exec-maven-plugin**

<http://mojo.codehaus.org/exec-maven-plugin/>

exec-maven-plugin很好理解，顾名思义，它能让你运行任何本地的系统程序，在某些特定情况下，运行一个Maven外部的程序可能就是最简单的问题解决方案，这就是exec:exec的用途，当然，该插件还允许你配置相关的程序运行参数。除了exec目标之外，exec-maven-plugin还提供了一个java目标，该目标要求你提供一个mainClass参数，然后它能够利用当前项目的依赖作为classpath，在同一个JVM中运行该mainClass。有时候，为了简单的演示一个命令行Java程序，你可以在POM中配置好exec-maven-plugin的相关运行参数，然后直接在命令运行mvn exec:java 以查看运行效果。

**jetty-maven-plugin**

<http://wiki.eclipse.org/Jetty/Feature/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。

**versions-maven-plugin**

<http://mojo.codehaus.org/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，不过不在本文讨论范围）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。

**6 常用Maven命令**

|  |  |
| --- | --- |
| 生命周期 | 阶段描述 |
| **mvn validate** | 验证项目是否正确，以及所有为了完整构建必要的信息是否可用 |
| **mvn generate-sources** | 生成所有需要包含在编译过程中的源代码 |
| **mvn process-sources** | 处理源代码，比如过滤一些值 |
| **mvn generate-resources** | 生成所有需要包含在打包过程中的资源文件 |
| **mvn process-resources** | 复制并处理资源文件至目标目录，准备打包 |
| **mvn compile** | 编译项目的源代码 |
| **mvn process-classes** | 后处理编译生成的文件，例如对Java类进行字节码增强（bytecode enhancement） |
| **mvn generate-test-sources** | 生成所有包含在测试编译过程中的测试源码 |
| **mvn process-test-sources** | 处理测试源码，比如过滤一些值 |
| **mvn generate-test-resources** | 生成测试需要的资源文件 |
| **mvn process-test-resources** | 复制并处理测试资源文件至测试目标目录 |
| **mvn test-compile** | 编译测试源码至测试目标目录 |
| **mvn test** | 使用合适的单元测试框架运行测试。这些测试应该不需要代码被打包或发布 |
| **mvn prepare-package** | 在真正的打包之前，执行一些准备打包必要的操作。这通常会产生一个包的展开的处理过的版本（将会在Maven 2.1+中实现） |
| **mvn package** | 将编译好的代码打包成可分发的格式，如JAR，WAR，或者EAR |
| **mvn pre-integration-test** | 执行一些在集成测试运行之前需要的动作。如建立集成测试需要的环境 |
| **mvn integration-test** | 如果有必要的话，处理包并发布至集成测试可以运行的环境 |
| **mvn post-integration-test** | 执行一些在集成测试运行之后需要的动作。如清理集成测试环境。 |
| **mvn verify** | 执行所有检查，验证包是有效的，符合质量规范 |
| **mvn install** | 安装包至本地仓库，以备本地的其它项目作为依赖使用 |
| **mvn deploy** | 复制最终的包至远程仓库，共享给其它开发人员和项目（通常和一次正式的发布相关） |

**使用参数**

**-Dmaven.test.skip=true: 跳过单元测试**

**(eg: mcn clean package -Dmaven.test.skip=true)**

**1 创建Maven的普通java项目：**

mvn archetype:create

-DgroupId=packageName

-DartifactId=projectName

如：mvn archetype:create -DgroupId=com.test.mvn -DartifactId=testmvn

**2. 创建Maven的Web项目：**

mvn archetype:create

-DgroupId=packageName

-DartifactId=webappName

-DarchetypeArtifactId=maven-archetype-webapp

如： mvn archetype:create -DgroupId=com.test.mvn -DartifactId=webappName -DarchetypeArtifactId=maven-archetype-webapp

**3. 编译源代码： mvn compile**

**4. 编译测试代码：mvn test-compile**

**5. 运行测试：mvn test**

**6. 产生site：mvn site**

**7. 打包：mvn package**

**8. 在本地Repository中安装jar：mvn install**

**9. 清除产生的项目：mvn clean**

**10. 生成eclipse项目：mvn eclipse:eclipse**

**11. 生成idea项目：mvn idea:idea**

**12. 组合使用goal命令，如只打包不测试：mvn -Dtest package**

**13. 编译测试的内容：mvn test-compile**

**14. 只打jar包: mvn jar:jar**

**15. 只测试而不编译，也不测试编译**：mvn test -skipping compile -skipping test-compile

( -skipping 的灵活运用，当然也可以用于其他组合命令)

**16. 清除eclipse的一些系统设置**:mvn eclipse:clean

**17 其他命令**

mvn -version/-v 显示版本信息

mvn archetype:generate 创建mvn项目

mvn archetype:create -DgroupId=com.oreilly -DartifactId=my-app 创建mvn项目

mvn package 生成target目录，编译、测试代码，生成测试报告，生成jar/war文件

mvn jetty:run 运行项目于jetty上,

mvn compile 编译

mvn test 编译并测试

mvn clean 清空生成的文件

mvn site 生成项目相关信息的网站

mvn -Dwtpversion=1.0 eclipse:eclipse 生成Wtp插件的Web项目

mvn -Dwtpversion=1.0 eclipse:clean 清除Eclipse项目的配置信息(Web项目)

mvn eclipse:eclipse 将项目转化为Eclipse项目

mvn -e 显示详细错误 信息.

mvn validate 验证工程是否正确，所有需要的资源是否可用。

mvn test-compile 编译项目测试代码。 。

mvn integration-test 在集成测试可以运行的环境中处理和发布包。

mvn verify 运行任何检查，验证包是否有效且达到质量标准。

mvn generate-sources 产生应用需要的任何额外的源代码，如xdoclet。

mvn -v 显示版本

mvn help:describe -Dplugin=help 使用help插件的describe 目标来输Help 插件的信息。

mvn help:describe -Dplugin=help -Dfull 使用Help 插件输出完整的带有参数的目标列

mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull 获取单个目标的信息,设置 mojo 参数和 plugin 参数。此命令列出了Compiler 插件的compile 目标的所有信息

mvn help:describe -Dplugin=exec -Dfull 列出所有 Maven Exec 插件可用的目标

mvn help:effective-pom 看这个“有效的 (effective)”POM，它暴露了 Maven的默认设置

mvn archetype:create -DgroupId=org.sonatype.mavenbook.ch03 -DartifactId=simple -DpackageName=org.sonatype.mavenbook 创建Maven的普通java项目，在命令行使用Maven Archetype 插件

mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main Exec 插件让我们能够在不往 classpath 载入适当的依赖的情况下，运行这个程序

mvn dependency:resolve 打印出已解决依赖的列表

mvn dependency:tree 打印整个依赖树

mvn install -X 想要查看完整的依赖踪迹，包含那些因为冲突或者其它原因而被拒绝引入的构件，打开 Maven 的调试标记运行

mvn install -Dmaven.test.skip=true 给任何目标添加maven.test.skip 属性就能跳过测试

mvn install assembly:assembly 构建装配Maven Assembly 插件是一个用来创建你应用程序特有分发包的插件

mvn jetty:run 调用 Jetty 插件的 Run 目标在 Jetty Servlet 容器中启动 web 应用

mvn compile 编译你的项目

mvn clean install 删除再编译

mvn hibernate3:hbm2ddl 使用 Hibernate3 插件构造数据库

**18 mvn deploy命令**

mvn deploy:deploy-file -DgroupId=com -DartifactId=client -Dversion=0.1.0 -Dpackaging=jar -Dfile=d:client-0.1.0.jar -DrepositoryId=maven-repository-inner -Durl=ftp://xxxxxxx/opt/maven/repository/

**19 mvn install命令：发布第三方Jar到本地库中：**

mvn install:install-file -DgroupId=com -DartifactId=client -Dversion=0.1.0 -Dpackaging=jar -Dfile=d:client-0.1.0.jar

**7 常见问题**

（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，类似的还是plugins和pluginManagement。

**@@@@@**

**setting.xml 配合 compiler 3.5 ，ALT+F5 ，java compiler自动跟着setting.xml 走。**

**<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>**

**也可以配置 pom.xml ，当配置了pom这个优先级高 。**

**<plugin>**

**<groupId>org.apache.maven.plugins</groupId>**

**<artifactId>maven-compiler-plugin</artifactId>**

**<version>3.5.1</version>**

**<configuration>**

**<source>${jdk.version}</source>**

**<target>${jdk.version}</target>**

**<showWarnings>true</showWarnings>**

**</configuration>**

**</plugin>**

**配置打包跳过测试**

**<!-- 打包跳过测试 -->**

**<plugin>**

**<groupId>org.apache.maven.plugins</groupId>**

**<artifactId>maven-surefire-plugin</artifactId>**

**<version>2.4.2</version>**

**<configuration>**

**<skipTests>true</skipTests>**

**</configuration>**

**</plugin>**

**run as maven install:在target打包并且 推送到 D:\maven\repository\yunnex**

**run as maven clean:清理class文件**

**run as maven test:自动执行单元测试，并且输出结果**

**run as maven build：**

**goals 输入 clean test 测试**

**goals 输入 clean package 打包**

**goals 输入 clean install 打包并且推送**

**goals 输入 tomcat7:run 运行Maven内置tomcat 7**

**<plugin>**

**<groupId>org.apache.tomcat.maven</groupId>**

**<artifactId>tomcat7-maven-plugin</artifactId>**

**<version>${tomcat.version}</version>**

**<configuration>**

**<port>${webserver.port}</port>**

**<path>/${project.artifactId}</path>**

**<uriEncoding>${project.build.sourceEncoding}</uriEncoding>**

**</configuration>**

**</plugin>**

**goals 输入 tomcat:reload：重新加载web war包**

**goals 输入 tomcat:start：启动Maven内置tomcat**

**goals 输入 tomcat:stop：停止Maven内置tomcat**

**goals 输入jetty:run ：运行Maven内置jetty**

**mvn package clean -Dmaven.test.skip=true**

# **（2）mirror和repository 区别**

例如， 有一个项目，需要在公司和住所都编码，并在项目pom.xml配置了A Maven库。在公司，是电信网络，访问A库很快，所以maven管理依赖和插件都从A库下载；在住所，是网通网络，访问A库很慢，但是访问B库很快。这时，在住所的setting.xml里，只要配置一下<mirrors><mirror>....</mirror></mirrors>，让B库成为A库的mirror，即可不用更改项目pom.xml里对于A库的相关配置。由于镜像仓库完全屏蔽了被镜像仓库，当镜像仓库不稳定或者停止服务的时候，Maven仍将无法访问被镜像仓库，因而将无法下载构件。如果该镜像仓库需要认证，则配置setting.xml中的<server></server>即可。

repository就是个仓库。maven里有两种仓库，本地仓库和远程仓库。远程仓库相当于公共的仓库，大家都能看到。本地仓库是你本地的一个山寨版，只有你看的到，主要起缓存作用。当你向仓库请求插件或依赖的时候，会先检查本地仓库里是否有。如果有则直接返回，否则会向远程仓库请求，并做缓存。

<mirrors>

  <mirror>

    <id>UK</id>

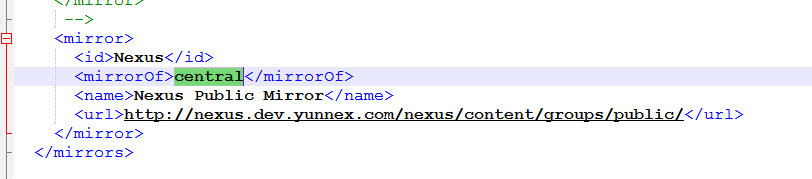
    <name>UK Central</name>

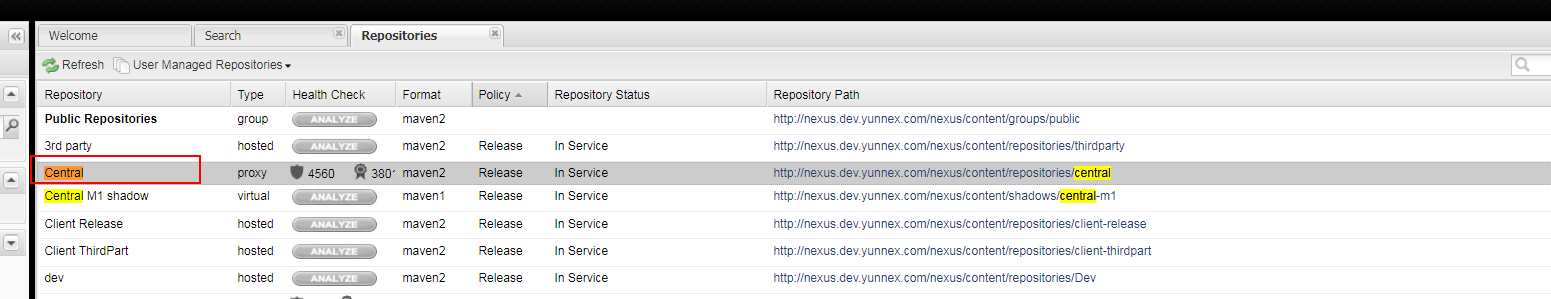
    <url>http://uk.maven.org/maven2</url>

    <mirrorOf>central</mirrorOf>

  </mirror>

</mirrors>    
这样的话，就会给上面id为central的远程仓库做了个镜像。以后向central这个仓库发的请求都会发到http://uk.maven.org/maven2而不是http://repo1.maven.org/maven2了。   
<mirrorOf>central</mirrorOf>里是要替代的仓库的id。如果填\*，就会替代所有仓库。





## 四 Maven多模块开发

**1、Parent怎么能找到childA和childB呢？**

在maven中，parent模块组织好childA和childB，叫做"聚合"，多个模块联合编译。实现起来很简单，只需要在parent的pom文件里加入以下内容。

<modules>

<module>childA</module>

<module>childB</module>

</modules>

**2、是不是这样写就完全ok了？**

这样只是告诉maven编译器，在读取parent的pom文件时去找到childA和childB，但还是会分别去编译他们引入的依赖。这样就会导致pom文件引入的包重复！！于是我们引入了"继承"的概念，也就是形成"父子"关系，子pom可以引用到父pom中引入的依赖。具体做法如下：

在parent中，写入以下内容，其中"\*"标识的行可以组成一个路径，通过这个路径可以在maven仓库中找到这个pom文件！本例中，path为M2\_Path/com/sang/main/Parent-Moduel/1.0.2/xxxx-1.0.2.pom。所以这三个标签是必须的！！！

<modelVersion>4.0.0</modelVersion>

<groupId>com.sang.main</groupId> \*

<artifactId>Parent-Moduel</artifactId> \*

<version>1.0.2</version> \*

<packaging>pom</packaging>

<name>Simple-main</name>

父pom写好了，子pom就通过<parent>标签继承父pom的依赖，如下：

<parent>

<groupId>com.sang.main</groupId>

<artifactId>Parent-Moduel</artifactId>

<version>1.0.2</version>

<relativePath>../pom.xml</relativePath> <!--本例中此处是可选的-->

</parent>

值得注意的是<relativePath>标签，如果pom的层次关系就像本例中的那样只隔一层，则可以省略这个。maven同样可以找到子pom。

子pom中引入<parent>标签后，就会从父pom继承<version>等属性了，例如childA只需要再加入如下内容即可！

<modelVersion>4.0.0</modelVersion>

<groupId>com.sang.business</groupId> <!--和artifactId一起唯一标识这个jar文件-->

<artifactId>ChildA-module</artifactId>

<packaging>jar</packaging> <!--指明打包类型-->

<name>childA</name>

**3、如何添加依赖？**

maven可以让我们方便地管理jar包依赖，具体做法如下：

<dependencies>

<dependency> <!--添加一个jar包依赖-->

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>

</dependencies>

如果不通过继承，则需要在每个pom中加入这样的依赖，这样子pom对应的模块可以引用到这个jar包。上面提到的重复引用jar包，可以通过下面的方式解决：

主pom中把依赖通过<dependecyManagement>引起来，表示子pom可能会用到的jar包依赖

<dependencyManagement>

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>

</dependencies>

</dependencyManagement>

子pom如果需要引用该jar包，则直接引用即可！不需要加入<version>，便于统一管理。此外也可以加入仅在子pom中用到的jar包，比如：

<dependencies>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId> <!--此处不再需要verison了！-->

</dependency>

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-core-lgpl</artifactId>

<version>1.9.4</version> <!--当然也可以加入只在这个子模块中用到的jar包-->

</dependency>

</dependencies>

**4、除了jar包依赖，插件也可以通过这样的方式进行管理**

<!-- mainModule -->

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-source-plugin</artifactId>

<version>2.1.1</version>

</plugin>

</plugins>

</pluginManagement>

</build>

<!-- childA -->

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-source-plugin</artifactId>

</plugin>

</plugins>

</build>

**5、如果子pom间存在引用关系，比如childA引用到了childB的jar包，该怎么做？**

<dependency>

<groupId>com.module</groupId>

<artifactId>childA</artifactId> <!--加上childA的依赖-->

<version>1.0.0</version>

</dependency>

## 五 [pom.xml文件详解](https://www.cnblogs.com/hafiz/p/5360195.html)

**1.整体概述**

pom中节点如下分布

<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>

<!-- 基本配置 -->

<groupId>...</groupId>

<artifactId>...</artifactId>

<version>...</version>

<packaging>...</packaging>

<!-- 依赖配置 -->

<dependencies>...</dependencies>

<parent>...</parent>

<dependencyManagement>...</dependencyManagement>

<modules>...</modules>

<properties>...</properties>

<!-- 构建配置 -->

<build>...</build>

<reporting>...</reporting>

<!-- 项目信息 -->

<name>...</name>

<description>...</description>

<url>...</url>

<inceptionYear>...</inceptionYear>

<licenses>...</licenses>

<organization>...</organization>

<developers>...</developers>

<contributors>...</contributors>

<!-- 环境设置 -->

<issueManagement>...</issueManagement>

<ciManagement>...</ciManagement>

<mailingLists>...</mailingLists>

<scm>...</scm>

<prerequisites>...</prerequisites>

<repositories>...</repositories>

<pluginRepositories>...</pluginRepositories>

<distributionManagement>...</distributionManagement>

<profiles>...</profiles>

</project>

**2.基本配置**

modelVersion：pom模型版本，maven2和3只能为4.0.0

groupId：组ID，maven用于定位

artifactId：在组中的唯一ID用于定位

version：项目版本

packaging：项目打包方式，有以下值：pom, jar, maven-plugin, ejb, war, ear, rar, par

**3.依赖配置**

**（1）parent：用于确定父项目的坐标。**

<parent>

<groupId>com.learnPro</groupId>

<artifactId>SIP-parent</artifactId>

<relativePath></relativePath>

<version>0.0.1-SNAPSHOT</version>

</parent>

groupId：父项目的构件标识符

artifactId：父项目的唯一标识符

relativePath：Maven首先在当前项目的找父项目的pom，然后在文件系统的这个位置（relativePath），然后在本地仓库，再在远程仓库找。

version：父项目的版本

**（2）modules：**有些maven项目会做成多模块的，这个标签用于指定当前项目所包含的所有模块。之后对这个项目进行的maven操作，会让所有子模块也进行相同操作。

<modules>

<module>com-a</>

<module>com-b</>

<module>com-c</>

</modules>

**（3）properties：用于定义pom常量**

<properties>

<java.version>1.7</java.version>

</properties>

上面这个常量可以在pom文件的任意地方通过${java.version}来引用

**（4）dependencies**

项目相关依赖配置，如果在父项目写的依赖，会被子项目引用，一般父项目会将子项目公用的依赖引入（将在之后详细讲解）

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

</dependencies>

这边依赖和中央仓库中的一致，就可以引入对应的jar

**（5）dependencyManagement**

@配置写法同dependencies

<dependencyManagement>

<dependencies>

.....

</dependencies>

</dependencyManagement>

在父模块中定义后，子模块不会直接使用对应依赖，但是在使用相同依赖的时候可以不加版本号：

@父项目：

<dependencyManagement>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

</dependencies>

</dependencyManagement>

@子项目：

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

@这样的好处是，父项目统一了版本，而且子项目可以在需要的时候才引用对应的依赖

**4.构建配置**

build：用于配置项目构建相关信息

<build>

<!--该元素设置了项目源码目录，当构建项目的时候，构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。-->

<sourceDirectory/>

<!--该元素设置了项目脚本源码目录，该目录和源码目录不同：绝大多数情况下，该目录下的内容 会被拷贝到输出目录(因为脚本是被解释的，而不是被编译的)。-->

<scriptSourceDirectory/>

<!--该元素设置了项目单元测试使用的源码目录，当测试项目的时候，构建系统会编译目录里的源码。该路径是相对于pom.xml的相对路径。-->

<testSourceDirectory/>

<!--被编译过的应用程序class文件存放的目录。-->

<outputDirectory/>

<!--被编译过的测试class文件存放的目录。-->

<testOutputDirectory/>

<!--使用来自该项目的一系列构建扩展-->

<extensions>

<!--描述使用到的构建扩展。-->

<extension>

<!--构建扩展的groupId-->

<groupId/>

<!--构建扩展的artifactId-->

<artifactId/>

<!--构建扩展的版本-->

<version/>

</extension>

</extensions>

<!--当项目没有规定目标（Maven2 叫做阶段）时的默认值-->

<defaultGoal/>

<!--这个元素描述了项目相关的所有资源路径列表，例如和项目相关的属性文件，这些资源被包含在最终的打包文件里。-->

<resources>

<!--这个元素描述了项目相关或测试相关的所有资源路径-->

<resource>

<!-- 描述了资源的目标路径。该路径相对target/classes目录（例如${project.build.outputDirectory}）。举个例 子，如果你想资源在特定的包里(org.apache.maven.messages)，你就必须该元素设置为org/apache/maven /messages。然而，如果你只是想把资源放到源码目录结构里，就不需要该配置。-->

<targetPath/>

<!--是否使用参数值代替参数名。参数值取自properties元素或者文件里配置的属性，文件在filters元素里列出。-->

<filtering/>

<!--描述存放资源的目录，该路径相对POM路径-->

<directory/>

<!--包含的模式列表，例如\*\*/\*.xml.-->

<includes/>

<!--排除的模式列表，例如\*\*/\*.xml-->

<excludes/>

</resource>

</resources>

<!--这个元素描述了单元测试相关的所有资源路径，例如和单元测试相关的属性文件。-->

<testResources>

<!--这个元素描述了测试相关的所有资源路径，参见build/resources/resource元素的说明-->

<testResource>

<targetPath/><filtering/><directory/><includes/><excludes/>

</testResource>

</testResources>

<!--构建产生的所有文件存放的目录-->

<directory/>

<!--产生的构件的文件名，默认值是${artifactId}-${version}。-->

<finalName/>

<!--当filtering开关打开时，使用到的过滤器属性文件列表-->

<filters/>

<!--子项目可以引用的默认插件信息。该插件配置项直到被引用时才会被解析或绑定到生命周期。给定插件的任何本地配置都会覆盖这里的配置-->

<pluginManagement>

<!--使用的插件列表 。-->

<plugins>

<!--plugin元素包含描述插件所需要的信息。-->

<plugin>

<!--插件在仓库里的group ID-->

<groupId/>

<!--插件在仓库里的artifact ID-->

<artifactId/>

<!--被使用的插件的版本（或版本范围）-->

<version/>

<!--是否从该插件下载Maven扩展（例如打包和类型处理器），由于性能原因，只有在真需要下载时，该元素才被设置成enabled。-->

<extensions/>

<!--在构建生命周期中执行一组目标的配置。每个目标可能有不同的配置。-->

<executions>

<!--execution元素包含了插件执行需要的信息-->

<execution>

<!--执行目标的标识符，用于标识构建过程中的目标，或者匹配继承过程中需要合并的执行目标-->

<id/>

<!--绑定了目标的构建生命周期阶段，如果省略，目标会被绑定到源数据里配置的默认阶段-->

<phase/>

<!--配置的执行目标-->

<goals/>

<!--配置是否被传播到子POM-->

<inherited/>

<!--作为DOM对象的配置-->

<configuration/>

</execution>

</executions>

<!--项目引入插件所需要的额外依赖-->

<dependencies>

<!--参见dependencies/dependency元素-->

<dependency>

......

</dependency>

</dependencies>

<!--任何配置是否被传播到子项目-->

<inherited/>

<!--作为DOM对象的配置-->

<configuration/>

</plugin>

</plugins>

</pluginManagement>

<!--使用的插件列表-->

<plugins>

<!--参见build/pluginManagement/plugins/plugin元素-->

<plugin>

<groupId/><artifactId/><version/><extensions/>

<executions>

<execution>

<id/><phase/><goals/><inherited/><configuration/>

</execution>

</executions>

<dependencies>

<!--参见dependencies/dependency元素-->

<dependency>

......

</dependency>

</dependencies>

<goals/><inherited/><configuration/>

</plugin>

</plugins>

</build>

**5 reporting：该元素描述使用报表插件产生报表的规范。**

当用户执行“mvn site”，这些报表就会运行。 在页面导航栏能看到所有报表的链接。

<reporting>

<!--true，则，网站不包括默认的报表。这包括“项目信息”菜单中的报表。-->

<excludeDefaults/>

<!--所有产生的报表存放到哪里。默认值是${project.build.directory}/site。-->

<outputDirectory/>

<!--使用的报表插件和他们的配置。-->

<plugins>

<!--plugin元素包含描述报表插件需要的信息-->

<plugin>

<!--报表插件在仓库里的group ID-->

<groupId/>

<!--报表插件在仓库里的artifact ID-->

<artifactId/>

<!--被使用的报表插件的版本（或版本范围）-->

<version/>

<!--任何配置是否被传播到子项目-->

<inherited/>

<!--报表插件的配置-->

<configuration/>

<!--一组报表的多重规范，每个规范可能有不同的配置。一个规范（报表集）对应一个执行目标 。例如，有1，2，3，4，5，6，7，8，9个报表。1，2，5构成A报表集，对应一个执行目标。2，5，8构成B报表集，对应另一个执行目标-->

<reportSets>

<!--表示报表的一个集合，以及产生该集合的配置-->

<reportSet>

<!--报表集合的唯一标识符，POM继承时用到-->

<id/>

<!--产生报表集合时，被使用的报表的配置-->

<configuration/>

<!--配置是否被继承到子POMs-->

<inherited/>

<!--这个集合里使用到哪些报表-->

<reports/>

</reportSet>

</reportSets>

</plugin>

</plugins>

</reporting>

**6.项目信息**

**（1）license信息**

name：给用户提供更为友好的项目名

description：项目描述，maven文档中保存

url：主页的URL，maven文档中保存

inceptionYear：项目创建年份，4位数字。当产生版权信息时需要使用这个值

licenses：该元素描述了项目所有License列表。 应该只列出该项目的license列表，不要列出依赖项目的 license列表。如果列出多个license，用户可以选择它们中的一个而不是接受所有license。（如下）

<license>

<!--license用于法律上的名称-->

<name>...</name>

<!--官方的license正文页面的URL-->

<url>....</url>

<!--项目分发的主要方式：repo，可以从Maven库下载 manual， 用户必须手动下载和安装依赖-->

<distribution>repo</distribution>

<!--关于license的补充信息-->

<comments>....</comments>

</license>

**（2）developers信息**

organization：1.name 组织名 2.url 组织主页url

developers：项目开发人员列表（如下）

contributors：项目其他贡献者列表，同developers

<developers>

<!--某个开发者信息-->

<developer>

<!--开发者的唯一标识符-->

<id>....</id>

<!--开发者的全名-->

<name>...</name>

<!--开发者的email-->

<email>...</email>

<!--开发者的主页-->

<url>...<url/>

<!--开发者在项目中的角色-->

<roles>

<role>Java Dev</role>

<role>Web UI</role>

</roles>

<!--开发者所属组织-->

<organization>sun</organization>

<!--开发者所属组织的URL-->

<organizationUrl>...</organizationUrl>

<!--开发者属性，如即时消息如何处理等-->

<properties>

<!-- 和主标签中的properties一样，可以随意定义子标签 -->

</properties>

<!--开发者所在时区， -11到12范围内的整数。-->

<timezone>-5</timezone>

</developer>

</developers>

**7.环境设置**

**（1）issueManagement：目的问题管理系统(Bugzilla, Jira, Scarab)的名称和URL**

<issueManagement>

<system>Bugzilla</system>

<url>http://127.0.0.1/bugzilla/</url>

</issueManagement>

system：系统类型

url：路径

**（2）ciManagement：项目的持续集成信息**

<ciManagement>

<system>continuum</system>

<url>http://127.0.0.1:8080/continuum</url>

<notifiers>

<notifier>

<type>mail</type>

<sendOnError>true</sendOnError>

<sendOnFailure>true</sendOnFailure>

<sendOnSuccess>false</sendOnSuccess>

<sendOnWarning>false</sendOnWarning>

<address>continuum@127.0.0.1</address>

<configuration></configuration>

</notifier>

</notifiers>

</ciManagement>

system：持续集成系统的名字

url：持续集成系统的URL

notifiers：构建完成时，需要通知的开发者/用户的配置项。包括被通知者信息和通知条件（错误，失败，成功，警告）

type：通知方式

sendOnError：错误时是否通知

sendOnFailure：失败时是否通知

sendOnSuccess：成功时是否通知

sendOnWarning：警告时是否通知

address：通知发送到的地址

configuration：扩展项

**（3）mailingLists：项目相关邮件列表信息**

<mailingLists>

<mailingList>

<name>User List</name>

<subscribe>user-subscribe@127.0.0.1</subscribe>

<unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>

<post>user@127.0.0.1</post>

<archive>http://127.0.0.1/user/</archive>

<otherArchives>

<otherArchive>http://base.google.com/base/1/127.0.0.1</otherArchive>

</otherArchives>

</mailingList>

.....

</mailingLists>

subscribe, unsubscribe: 订阅邮件（取消订阅）的地址或链接，如果是邮件地址，创建文档时，mailto: 链接会被自动创建

archive：浏览邮件信息的URL

post：接收邮件的地址

**（4）scm：允许你配置你的代码库，供Maven web站点和其它插件使用**

<scm>

<connection>scm:svn:http://127.0.0.1/svn/my-project</connection>

<developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnection>

<tag>HEAD</tag>

<url>http://127.0.0.1/websvn/my-project</url>

</scm>

connection, developerConnection：这两个表示我们如何连接到maven的版本库。connection只提供读，developerConnection将提供写的请求，写法如：scm:[provider]:[provider\_specific]

如果连接到CVS仓库，可以配置如下：scm:cvs:pserver:127.0.0.1:/cvs/root:my-project

tag：项目标签，默认HEAD

url：共有仓库路径

**（5）prerequisites：项目构建的前提**

<prerequisites>

<maven>2.0.6</maven>

</prerequisites>

**（6）repositories,pluginRepositories：依赖和扩展的远程仓库列表，同上篇文章，setting.xml配置中介绍的。**

<repositories>

<repository>

<releases>

<enabled>false</enabled>

<updatePolicy>always</updatePolicy>

<checksumPolicy>warn</checksumPolicy>

</releases>

<snapshots>

<enabled>true</enabled>

<updatePolicy>never</updatePolicy>

<checksumPolicy>fail</checksumPolicy>

</snapshots>

<id>codehausSnapshots</id>

<name>Codehaus Snapshots</name>

<url>http://snapshots.maven.codehaus.org/maven2</url>

<layout>default</layout>

</repository>

</repositories>

<pluginRepositories>

...

</pluginRepositories>

releases, snapshots:这是各种构件的策略，release或者snapshot。这两个集合，POM就可以根据独立仓库任意类型的依赖改变策略。如：一个人可能只激活下载snapshot用来开发。

enable：true或者false，决定仓库是否对于各自的类型激活(release 或者 snapshot)。

updatePolicy: 这个元素决定更新频率。maven将比较本地pom的时间戳（存储在仓库的maven数据文件中）和远程的. 有以下选择: always, daily (默认), interval:X (x是代表分钟的整型) ， never.

checksumPolicy：当Maven向仓库部署文件的时候，它也部署了相应的校验和文件。可选的为：ignore，fail，warn，或者不正确的校验和。

layout：在上面描述仓库的时候，提到他们有统一的布局。Maven 2有它仓库默认布局。然而，Maven 1.x有不同布局。使用这个元素来表明它是default还是legacy。

**（7）distributionManagement：它管理的分布在整个构建过程生成的工件和支持文件**

<distributionManagement>

...

<downloadUrl>http://mojo.codehaus.org/my-project</downloadUrl>

<status>deployed</status>

</distributionManagement>

downloadUrl: 其他pom可以通过此url的仓库抓取组件

status：给出该构件在远程仓库的状态

none: 默认

converted: 将被早期Maven 2 POM转换过来

partner: 这个项目会从合作者仓库同步过来

deployed: 从Maven 2或3实例部署

verified: 被核实时正确的和最终的

**（8）Repository：指定Maven pom从远程下载控件到当前项目的位置和方式，如果snapshotRepository没有被定义则使用repository相关的配置**

<distributionManagement>

<repository>

<uniqueVersion>false</uniqueVersion>

<id>corp1</id>

<name>Corporate Repository</name>

<url>scp://repo/maven2</url>

<layout>default</layout>

</repository>

<snapshotRepository>

<uniqueVersion>true</uniqueVersion>

<id>propSnap</id>

<name>Propellors Snapshots</name>

<url>sftp://propellers.net/maven</url>

<layout>legacy</layout>

</snapshotRepository>

...

</distributionManagement>

id, name：仓库的唯一标识

uniqueVersion：true或false，指明控件部署的时候是否获取独立的版本号。

url：repository元素的核心。指定位置和部署协议发布控件到仓库。

layout：布局，default或legacy

**（9）Site Distribution：多分布存储库,distributionManagement负责定义如何部署项目的网站和文档。**

<distributionManagement>

...

<site>

<id>mojo.website</id>

<name>Mojo Website</name>

<url>scp://beaver.codehaus.org/home/projects/mojo/public\_html/</url>

</site>

...

</distributionManagement>

id, name, url: 这些元素与distributionManagement repository中的相同

**（10）Relocation：**重新部署-项目不是静态的，是活的。他们需要被搬到更合适的地方。如：当你的下个成功的开源项目移到Apache下，重命名为org.apache:my-project:1.0 对你项目更有好处。

<distributionManagement>

...

<relocation>

<groupId>org.apache</groupId>

<artifactId>my-project</artifactId>

<version>1.0</version>

<message>We have moved the Project under Apache</message>

</relocation>

...

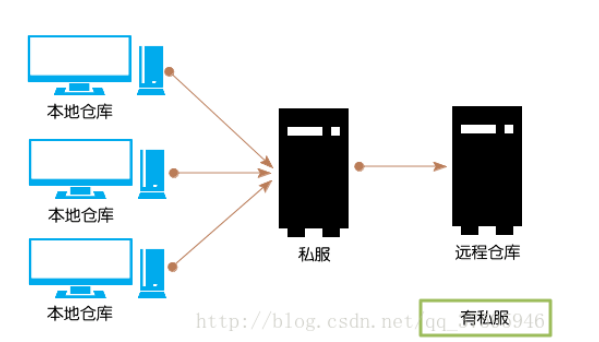
</distributionManagement>

**（11）profiles：profile**可以让我们定义一系列的配置信息（插件等），然后指定其激活条件

**Nexus私服**

## 一、私服简介

**1 Nexus 是Maven仓库管理器**，如果你使用Maven，你可以从Maven中央仓库 下载所需要的构件（artifact），但这通常不是一个好的做法，你应该在本地架设一个Maven仓库服务器，在代理远程仓库的同时维护本地仓库，以节省带宽和时间，Nexus就可以满足这样的需要。此外，他还提供了强大的仓库管理功能，构件搜索功能，它基于REST，友好的UI是一个extjs的REST客户端，它占用较少的内存，基于简单文件系统而非数据库。这些优点使其日趋成为最流行的Maven仓库管理器。



**2 私服是架设在局域网的一种特殊的远程仓库**，目的是代理远程仓库及部署第三方构件。有了私服之后，当 Maven 需要下载构件时，直接请求私服，私服上存在则下载到本地仓库；否则，私服请求外部的远程仓库，将构件下载到私服，再提供给本地仓库下载。我们可以使用专门的 Maven 仓库管理软件来搭建私服，比如：Apache Archiva，Artifactory，Sonatype Nexus。这里我们使用 Sonatype Nexus。

**3 Maven私服的特性**：

1.节省自己的外网带宽：减少重复请求造成的外网带宽消耗

2.加速Maven构件：如果项目配置了很多外部远程仓库的时候，构建速度就会大大降低

3.部署第三方构件：有些构件无法从外部仓库获得的时候，我们可以把这些构件部署到内部仓库(私服)中，供内部maven项目使用

4.提高稳定性，增强控制：Internet不稳定的时候，maven构建也会变的不稳定，一些私服软件还提供了其他的功能

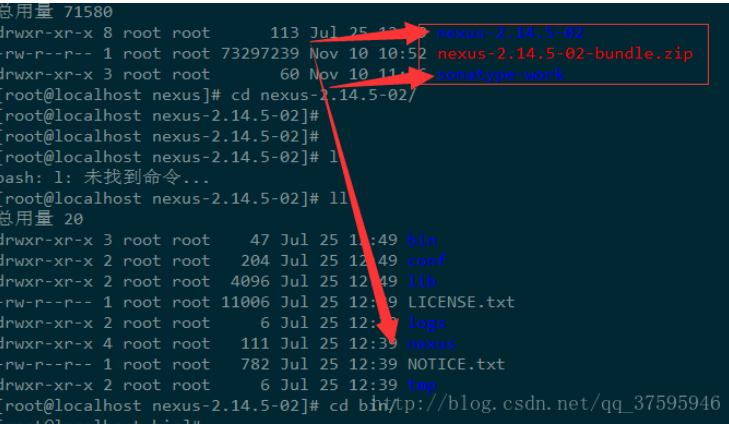
5.降低中央仓库的负荷：maven中央仓库被请求的数量是巨大的，配置私服也可以大大降低中央仓库的压力

## 二、安装Nexus

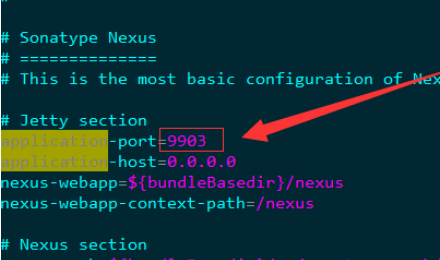
**1 Nexus 专业版是需要付费的**，这里我们下载开源版 Nexus OSS。Nexus 提供两种安装包，一种是包含 Jetty 容器的 bundle 包，另一种是不包含容器的 war 包。下载地址：http://www.sonatype.org/nexus/go。

**2 安装环境：操作系统：Linxu（CentOS）**

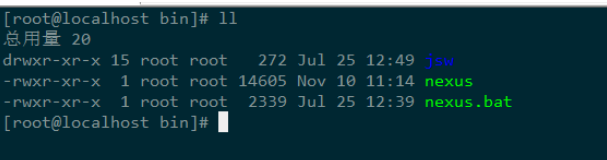
将压缩包解压到/usr/local/nexus目录下，nexus里面有2个文件夹，第一个文件夹是核心文件，第二个文件夹用来存储下载下来的jar：

****

**3 修改端口，进入nexus/nexubs-2.13.0-01/conf目录下，编辑nexus.properties文件，命令：vim nexus.properties**

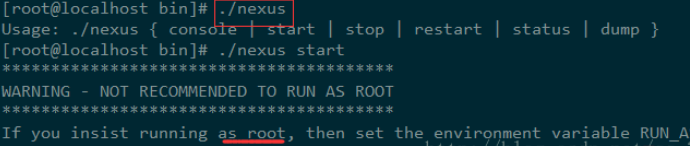
****

**4 启动nexus，进入nexus/nexubs-2.13.0-01/bin目录，**

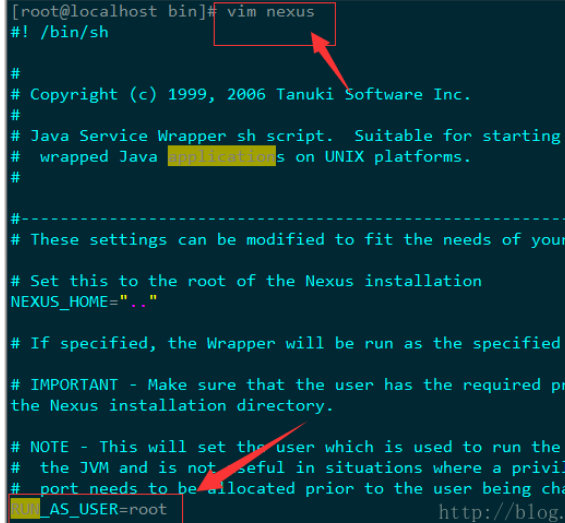
****

**5 运行命令：./nexus //可以查看有那些运行命令**

**6 运行命令：./nexus start //启动nexus**

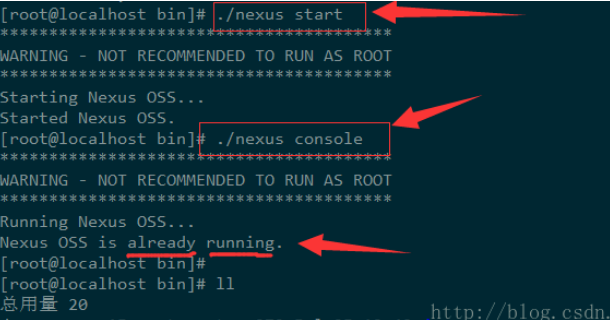
****

**7 根据错误提示：修改运行的用户，修改运行文件 nexus，命令：vim nexus，将RUN\_AS\_USER修改为root；**

****

**8 保存之后，再次启动nexus，运行命令：./nexus start**

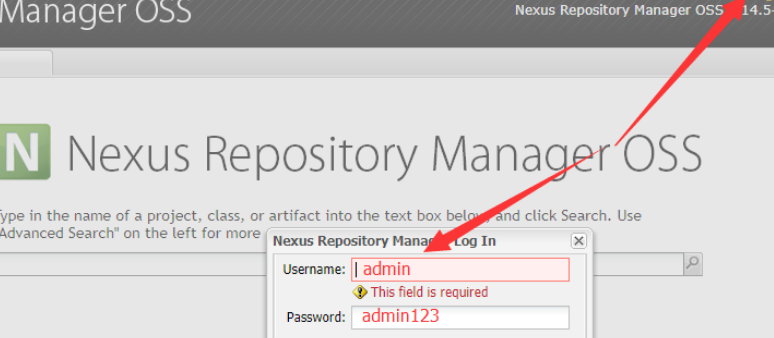
**9查看nexus 控制台，命令：./nexus console**

****

**10 在浏览器进行访问。地址：http://ip地址:端口（刚修改的9903）/nexus**

****

**11 点击右上角 Log In，使用用户名：admin ，密码：admin123 登录，可使用更多功能：**

****

**12点击左侧 Repositories 链接，查看 Nexus 内置的仓库：**

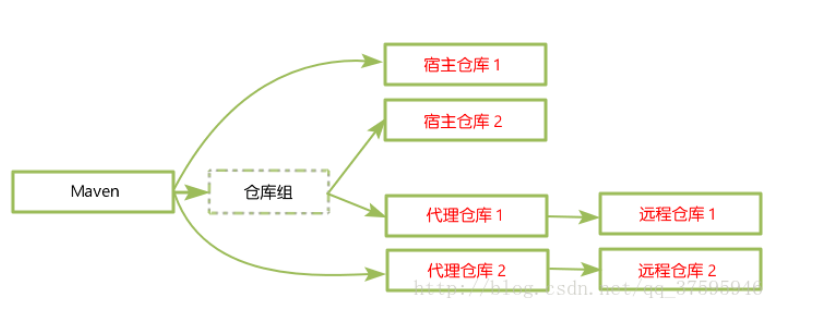
****

## 三、Nexus 的仓库分类

1 hosted 宿主仓库：主要用于部署无法从公共仓库获取的构件（如 oracle 的 JDBC 驱动）以及自己或第三方的项目构件；

2 proxy 代理仓库：代理公共的远程仓库； virtual 虚拟仓库：用于适配 Maven 1；

3 group 仓库组：Nexus 通过仓库组的概念统一管理多个仓库，这样我们在项目中直接请求仓库组即可请求到仓库组管理的多个仓库。



**我们看到仓库的类型主要有proxy、hosted、group类型，**

proxy是代理的远程仓库，

hosted是指本地或者内部项目仓库，

group只是一个仓库组，它包含其他的几个仓库，

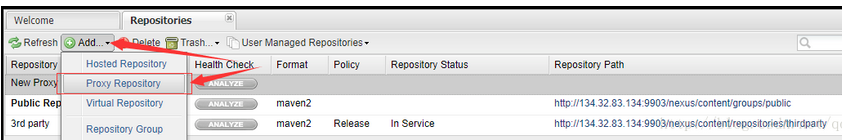
Releases是指发行版本（本地或者内部项目），

Snapshots是指正在构建的版本（本地或者内部项目），

Central是指中央仓库（远程中央仓库下载的构件放入此处）

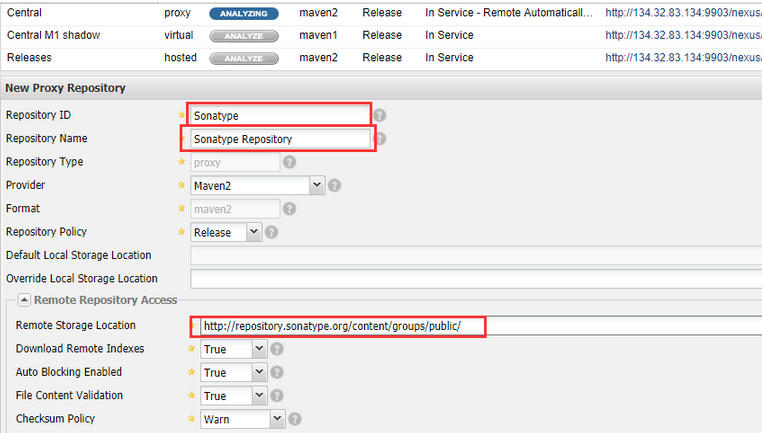
## 四、添加代理仓库

1 以 Sonatype 为例，添加一个代理仓库，用于代理 Sonatype 的公共远程仓库。点击菜单 Add - Proxy Repository ：

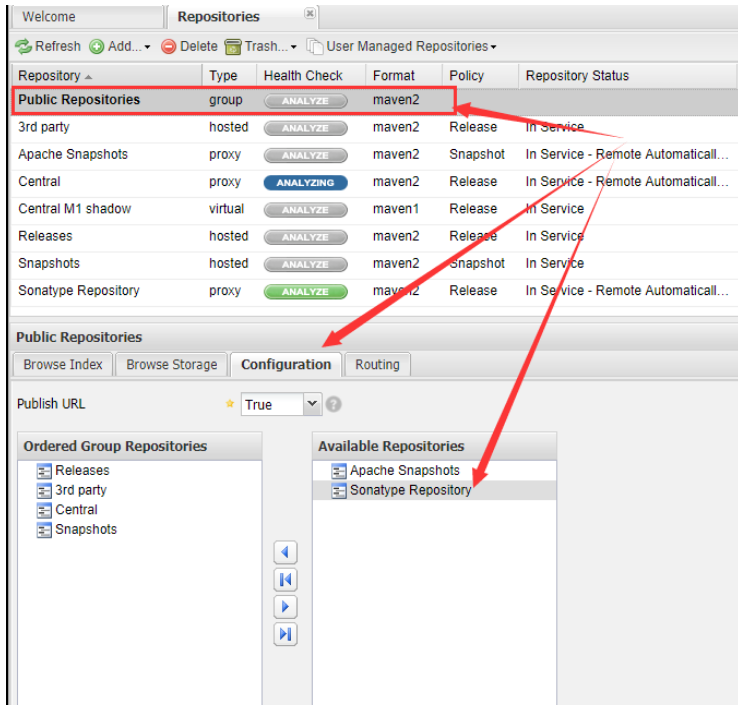


2 填写Repository ID - sonatype；Repository Name - Sonatype Repository；

Remote Storage Location - http://repository.sonatype.org/content/groups/public/ ，save 保存：

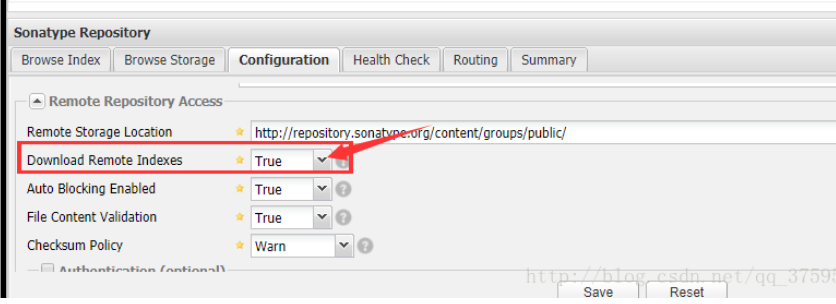


3 将添加的 Sonatype 代理仓库加入 Public Repositories 仓库组。选中 Public Repositories，在 Configuration 选项卡中，将 Sonatype Repository 从右侧 Available Repositories 移到左侧 Ordered Group Repositories，save 保存：

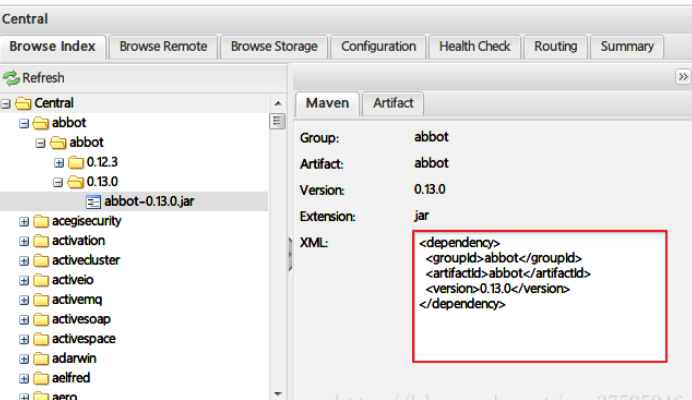


## 五、搜索构件

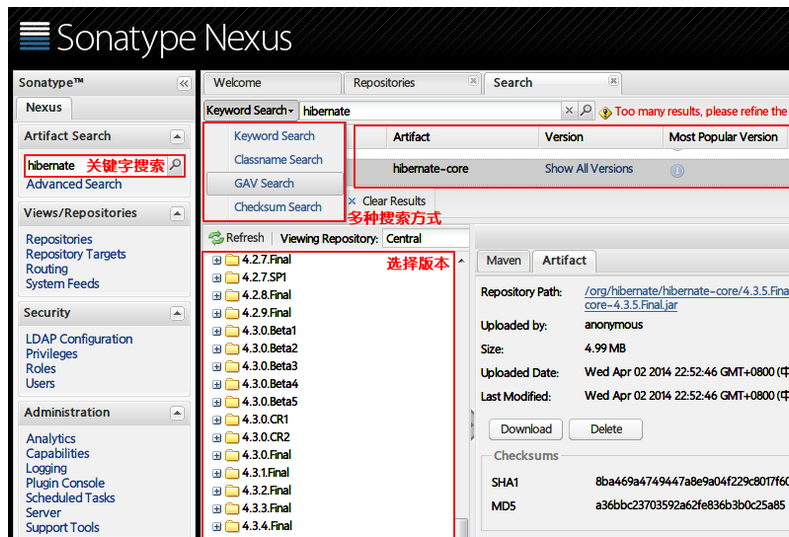
1 为了更好的使用 Nexus 的搜索，我们可以设置所有 proxy 仓库的 Download Remote Indexes 为 true，即允许下载远程仓库索引。



2 索引下载成功之后，在 Browse Index 选项卡下，可以浏览到所有已被索引的构件信息，包括坐标、格式、Maven 依赖的 xml 代码：



**3 有了索引，我们就可以搜索了**



## 六、配置Maven使用私服

私服搭建成功，我们就可以配置 Maven 使用私服，以后下载构件、部署构件，都通过私服来管理。

在 settings.xml 文件中，为所有仓库配置一个镜像仓库，镜像仓库的地址即私服的地址（这儿我们使用私服公共仓库组 Public Repositories 的地址）：



<mirrors>

<mirror>

<id>central</id>

<mirrorOf>\*</mirrorOf> <!-- \* 表示让所有仓库使用该镜像-->

<name>central-mirror</name>

<url>http://localhost:8081/nexus/content/groups/public/</url>

</mirror>

</mirrors>

<profile>

<id>dev</id>

<repositories>

<repository>

<id>dev</id>

<url>http://nexus.dev.yunnex.com/nexus/content/groups/public/</url>

<releases>

<enabled>true</enabled>

</releases>

<snapshots>

<enabled>true</enabled>

</snapshots>

</repository>

</repositories>

<properties>

<package.environment>dev</package.environment>

</properties>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<distributionManagement>

<repository>

<id>dev</id>

<name>Nexus Releases Repository</name>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/Dev</url>

</repository>

<snapshotRepository>

<id>snapshots</id>

<name>Nexus Snapshots Repository</name>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/snapshots</url>

</snapshotRepository>

</distributionManagement>

</profile>

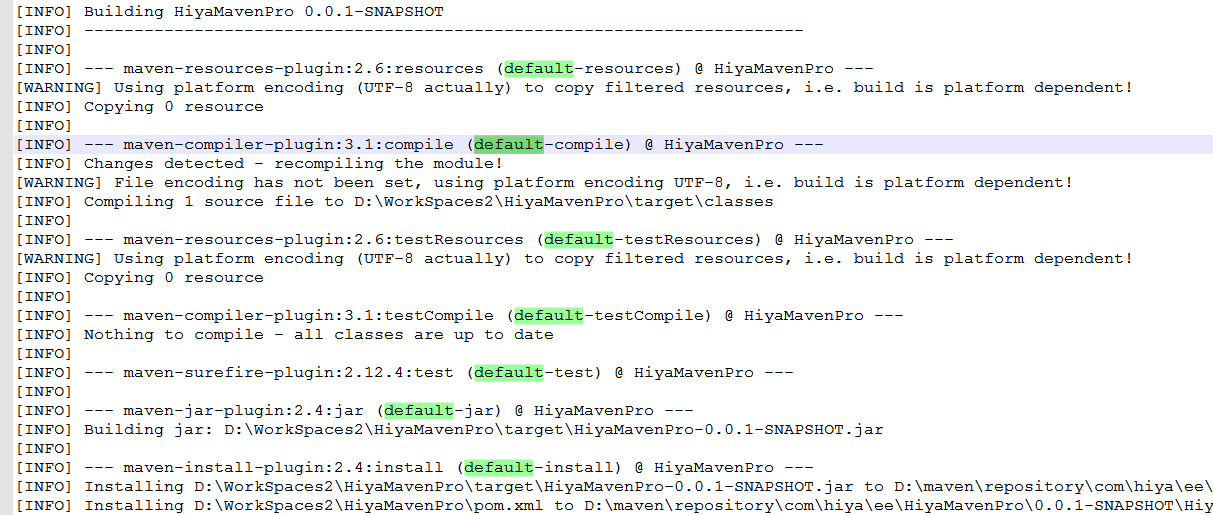
上面那个是中央仓库，下载路径，是一个仓库组包含很多仓库

下面两个是上传路径 ，附带账号密码。



**effective pom**

指的是默认一个大而全的有效的pom配置，如果指定了就用用户指定的，如果没指定的话取值默认的。如果不配置插件，使用默认的插件编译



**pluginRepositories和 Repositories**

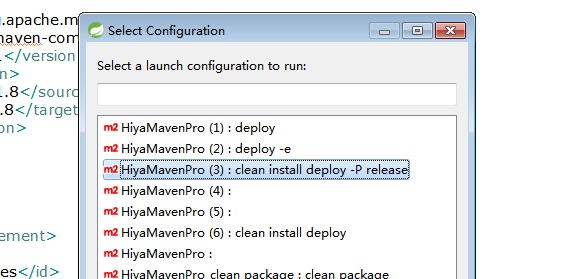


Maven中的库（repository）是构件（artifact）的集合。构件以一定的布局存储在库中。

根据构件的用途，构件可以分为两类，**一类是被其他构件依赖的构件（dependencies）**，这也是Maven库中主要的构件。**另一类是插件（plugins）**，这是一种特殊的构件。对于依赖的构件，其所在的库通过<repositories>设置。但是对于插件的构件，由于插件的特殊性，所以插件库独立于依赖库，使用<pluginRepositories>单独设置。但是由于依赖和插件都属于构件，所以<repositories>和<pluginRepositories>中的设置基本一致。

**mvn deploy命令**

mvn install 会将项目生成的构件安装到本地Maven仓库，mvn deploy 用来将项目生成的构件分发到远程Maven仓库。本地Maven仓库的构件只能供当前用户使用，在分发到远程Maven仓库之后，所有能访问该仓库的用户都能使用你的构件。



· **package命令完成了项目编译、单元测试、打包功能，但没有把打好的可执行jar包（war包或其它形式的包）布署到本地maven仓库和远程maven私服仓库**

· **install命令完成了项目编译、单元测试、打包功能，同时把打好的可执行jar包（war包或其它形式的包）布署到本地maven仓库，但没有布署到远程maven私服仓库**

· **deploy命令完成了项目编译、单元测试、打包功能，同时把打好的可执行jar包（war包或其它形式的包）布署到本地maven仓库和远程maven私服仓库**

**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/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.hiya.ee</groupId>

<artifactId>HiyaMavenPro</artifactId>

<version>0.0.1</version>

<properties>

<project.encoding>UTF-8</project.encoding>

<maven.compiler.encoding>${project.encoding}</maven.compiler.encoding>

</properties>

<dependencies>

<dependency>

<groupId>yunnex</groupId>

<artifactId>interface-base</artifactId>

<version>3.5.0</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

<distributionManagement>

<repository>

<id>releases</id>

<name>Releases Repository</name>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/releases</url>

</repository>

<snapshotRepository>

<id>snapshots</id>

<name>Snapshots Repository</name>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/snapshots</url>

</snapshotRepository>

</distributionManagement>

</project>

**Setting.xml**

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

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

<localRepository>D:\maven\repository</localRepository>

<pluginGroups> </pluginGroups>

<proxies></proxies>

<servers>

<server>

<id>releases</id>

<username>deployment</username>

<password>deployment123</password>

</server>

<server>

<id>snapshots</id>

<username>deployment</username>

<password>deployment123</password>

</server>

</servers>

<mirrors>

<mirror>

<id>Nexus</id>

<mirrorOf>central</mirrorOf>

<name>Nexus Public Mirror</name>

<url>http://nexus.dev.yunnex.com/nexus/content/groups/public/</url>

</mirror>

</mirrors>

<profiles>

<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>

<profile>

<id>nexus</id>

<repositories>

<repository>

<id>releases</id>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/releases</url>

<releases><enabled>true</enabled></releases>

</repository>

<repository>

<id>snapshots</id>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/snapshots</url>

<snapshots><enabled>true</enabled></snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>releases</id>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/releases</url>

<releases><enabled>true</enabled></releases>

</pluginRepository>

<pluginRepository>

<id>snapshots</id>

<url>http://nexus.dev.yunnex.com/nexus/content/repositories/snapshots</url>

<snapshots><enabled>true</enabled></snapshots>

</pluginRepository>

</pluginRepositories>

</profile>

<profile>

<id>disable-javadoc-doclint</id>

<activation>

<jdk>[1.8,)</jdk>

</activation>

<properties>

<additionalparam>-Xdoclint:none</additionalparam>

</properties>

</profile>

</profiles>

<activeProfiles>

<activeProfile>nexus</activeProfile>

</activeProfiles>

</settings>

**以下三个都可以**

**clean install deploy -P release**

**clean install deploy**

**deploy**

**（注意：-P release根据版本号来决定，如果版本号没有-SNAPSHOT，默认取得release）**



[INFO] Scanning for projects...

[INFO]

[INFO] ------------------------------------------------------------------------

[INFO] Building HiyaMavenPro 0.0.1

[INFO] ------------------------------------------------------------------------

[INFO]

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ HiyaMavenPro ---

[INFO] Deleting D:\WorkSpaces2\HiyaMavenPro\target

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ HiyaMavenPro ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ HiyaMavenPro ---

[INFO] Changes detected - recompiling the module!

[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!

[INFO] Compiling 1 source file to D:\WorkSpaces2\HiyaMavenPro\target\classes

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ HiyaMavenPro ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ HiyaMavenPro ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ HiyaMavenPro ---

[INFO]

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ HiyaMavenPro ---

[INFO] Building jar: D:\WorkSpaces2\HiyaMavenPro\target\HiyaMavenPro-0.0.1.jar

[INFO]

[INFO] --- maven-install-plugin:2.4:install (default-install) @ HiyaMavenPro ---

[INFO] Installing D:\WorkSpaces2\HiyaMavenPro\target\HiyaMavenPro-0.0.1.jar to D:\maven\repository\com\hiya\ee\HiyaMavenPro\0.0.1\HiyaMavenPro-0.0.1.jar

[INFO] Installing D:\WorkSpaces2\HiyaMavenPro\pom.xml to D:\maven\repository\com\hiya\ee\HiyaMavenPro\0.0.1\HiyaMavenPro-0.0.1.pom

[INFO]

[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ HiyaMavenPro ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ HiyaMavenPro ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ HiyaMavenPro ---

[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!

[INFO] Copying 0 resource

[INFO]

[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ HiyaMavenPro ---

[INFO] Nothing to compile - all classes are up to date

[INFO]

[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ HiyaMavenPro ---

[INFO] Skipping execution of surefire because it has already been run for this configuration

[INFO]

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ HiyaMavenPro ---

[INFO]

[INFO] --- maven-install-plugin:2.4:install (default-install) @ HiyaMavenPro ---

[INFO] Installing D:\WorkSpaces2\HiyaMavenPro\target\HiyaMavenPro-0.0.1.jar to D:\maven\repository\com\hiya\ee\HiyaMavenPro\0.0.1\HiyaMavenPro-0.0.1.jar

[INFO] Installing D:\WorkSpaces2\HiyaMavenPro\pom.xml to D:\maven\repository\com\hiya\ee\HiyaMavenPro\0.0.1\HiyaMavenPro-0.0.1.pom

[INFO]

[INFO] --- maven-deploy-plugin:2.7:deploy (default-deploy) @ HiyaMavenPro ---

[INFO] Uploading: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/0.0.1/HiyaMavenPro-0.0.1.jar

[INFO] Uploaded: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/0.0.1/HiyaMavenPro-0.0.1.jar (6 KB at 77.5 KB/sec)

[INFO] Uploading: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/0.0.1/HiyaMavenPro-0.0.1.pom

[INFO] Uploaded: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/0.0.1/HiyaMavenPro-0.0.1.pom (2 KB at 347.7 KB/sec)

[INFO] Downloading: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/maven-metadata.xml

[INFO] Downloaded: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/maven-metadata.xml (303 B at 26.9 KB/sec)

[INFO] Uploading: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/maven-metadata.xml

[INFO] Uploaded: http://nexus.dev.yunnex.com/nexus/content/repositories/releases/com/hiya/ee/HiyaMavenPro/maven-metadata.xml (303 B at 37.0 KB/sec)

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 2.729 s

[INFO] Finished at: 2018-08-08T11:01:48+08:00

[INFO] Final Memory: 22M/221M

[INFO] ------------------------------------------------------------------------

**中央仓库**



https://repo1.maven.org/maven2/

## 问题详解

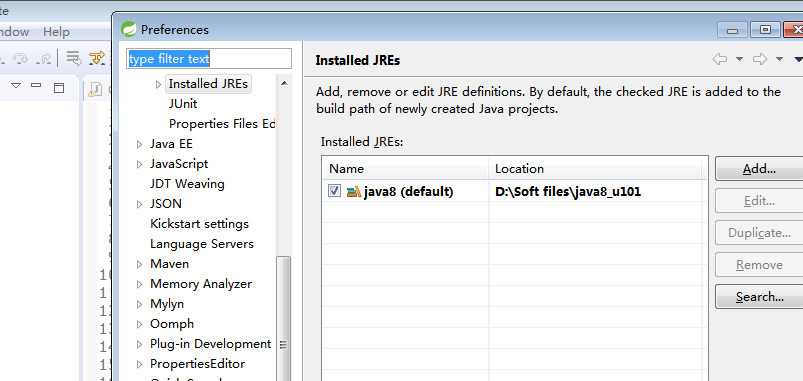
**1[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.1:compile (default-compile) on project pep-common: Compilation failure**

**[ERROR] No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?**

分析： 对这个问题其实报错中已经说的很清楚了，看第一条最后的 Perhaps you are running on a JRE rather than a JDK?

 就是说你可能编译在一个JRE而不是JDK上，maven 编译是要JDK的，可能你的编译器配置了一个jre路径。

**解决：在eclipse中，选择window→preference→java→Installed JREs 点add添加你的jdk路径，选jdk而不是jre,之前的报错原因就出在这里。**



**2 [ERROR] Failed to execute goal on project pep-generator: Could not resolve dependencies for project yunnex.pep:pep-generator:jar:0.0.1-SNAPSHOT: Failed to collect dependencies at yunnex.pep:pep-common:jar:0.0.1-SNAPSHOT: Failed to read artifact descriptor for yunnex.pep:pep-common:jar:0.0.1-SNAPSHOT: Failure to find yunnex.pep:pep:pom:${pep.version} in http://nexus.dev.yunnex.com/nexus/content/repositories/releases was cached in the local repository, resolution will not be reattempted until the update interval of releases has elapsed or updates are forced -> [Help 1]**

**解决方法：**

**将本地Maven仓库中已下载的相关依赖Jar删掉，从新build update工程即可.!**

**3[ERROR] Failed to execute goal org.apache.maven.plugins:maven-resources-plugin:2.6:resources (default-resources) on project pep-web-admin: Cannot create resource output directory: D:\git\pep\pep-web-admin\target\classes -> [Help 1]**

4

5