Maven

Maven

一、 Maven 概念

1 什么是 Maven

Maven 项目对象模型*(POM)*，可以通过一小段描述信息来管理项目的构建，报告和文档的软件项目管理工具。

Maven 除了以程序构建能力为特色之外，还提供高级项目管理工具。由于 *Maven* 的缺省构建规则有较高的可重用性，所以常常用两三行 *Maven* 构建脚本就可以构建简单的项目。由于 *Maven* 的面向项目的方法，许多 *Apache Jakarta* 项目发文时使用 *Maven*，而且公司项目采用 *Maven* 的比例在持续增长。

Maven 这个单词来自于意第绪语（犹太语），意为知识的积累，最初在 *Jakata Turbine* 项目中用来简化构建过程。当时有一些项目（有各自 *Ant build* 文件），仅有细微的差别，而 *JAR* 文件都由 *CVS* 来维护。于是希望有一种标准化的方式构建项目，一个清晰的方式定义项目的组成，一个容易的方式发布项目的信息，以及一种简单的方式在多个项目中共享 *JARs*。

2 Maven 和 ANT 的区别

那么，*Maven* 和 *Ant* 有什么不同呢？在回答这个问题以前，首先要强调一点：*Maven* 和 *Ant* 针对构建问题的两个不同方面。*Ant* 为 *Java* 技术开发项目提供跨平台构建任务。*Maven* 本身描述项目的高级方面，它从 *Ant* 借用了绝大多数构建任务。因此，由于 *Maven* 和 *Ant* 代表两个差异很大的工具，所以接下来只说明这两个工具的等同组件之间的区别，如表所示。

	Maven	*Ant*
标准构建文件	*pom.xml*	*build.xml*
特性处理顺序	*${maven.home}/bin/driver.properties* *${project.home}/project.properties* *${project.home}/build.properties* *${user.home}/build.properties* 通过 *-D* 命令行选项定义的系统特性 最后一个定义起决定作用。	通过 *-D* 命令行选项定义的系统特性 由 任务装入的特性 第一个定义最先被处理。
构建规则	构建规则更为动态（类似于编程语言）；它们是基于 *Jelly* 的可执行 *XML*。	构建规则或多或少是静态的，除非使用*<script>*任务
扩展语言	插件是用 *Jelly*（*XML*）编写的。	插件是用 *Java* 语言编写的。
构建规则可扩展性	通过定义 *<preGoal>* 和	构建规则不易扩展；

	<postGoal> 使构建 goal 可扩展。	可通过使用 <script> 任务模拟 <preGoal> 和 <postGoal> 所起的作用。

Maven 是一个项目管理工具，它包含了一个项目对象模型 (Project Object Model)，一组标准集合，一个项目生命周期(Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。当你使用 Maven 的时候，你用一个明确定义的项目对象模型来描述你的项目，然后 Maven 可以应用横切的逻辑，这些逻辑来自一组共享的（或者自定义的）插件。

Maven 有一个生命周期，当你运行 mvn install 的时候被调用。这条命令告诉 Maven 执行一系列的有序的步骤，直到到达你指定的生命周期。遍历生命周期旅途中的一个影响就是，Maven 运行了许多默认的插件目标，这些目标完成了像编译和创建一个 JAR 文件这样的工作。

此外，Maven 能够很方便的帮你管理项目报告，生成站点，管理 JAR 文件，等等。

3 Maven 仓库是什么

Maven 仓库是基于简单文件系统存储的，集中化管理 Java API 资源（构件）的一个服务。仓库中的任何一个构件都有其唯一的坐标，根据这个坐标可以定义其在仓库中的唯一存储路径。得益于 Maven 的坐标机制，任何 Maven 项目使用任何一个构件的方式都是完全相同的，Maven 可以在某个位置统一存储所有的 Maven 项目共享的构件，这个统一的位置就是仓库，项目构建完毕后生成的构件也可以安装或者部署到仓库中，供其它项目使用。

对于 Maven 来说，仓库分为两类：本地仓库和远程仓库。

3.1 远程仓库

远程仓库指通过各种协议如 file://和 http://访问的其它类型的仓库。这些仓库可能是第三方搭建的真实的远程仓库，用来提供他们的构件下载（例如 repo.maven.apache.org 和 uk.maven.org 是 Maven 的中央仓库）。其它“远程”仓库可能是你的公司拥有的建立在文件或 HTTP 服务器上的内部仓库，用来在开发团队间共享私有构件和管理发布的。

3.2 本地仓库

本地仓库指本机的一份拷贝，用来缓存远程下载，包含你尚未发布的临时构件。

4 仓库配置

Maven 官方网站： http://maven.apache.org/

Maven 工程中的默认仓库为 Apache 提供的中央仓库（repo.maven.apache.org）。中央仓库的信息在超级 Pom 中配置，所有的 maven 项目都会继承超级 POM。 超级 POM 的位置：$M2_HOME/lib/maven-model-builder-3.0.jar ，然后访问路径 org/apache/maven/model/pom-4.0.0.xml，可以看到配置：

```
<repositories>
    <repository>
        <id>central</id>
```

```
        <name>Central Repository</name>
        <url>https://repo.maven.apache.org/maven2</url>
        <layout>default</layout>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
    </repositories>
```

4.1 在 settings.xml 文件中配置本地仓库

本地仓库是开发者本地电脑中的一个目录，用于缓存从远程仓库下载的构件。默认的本地仓库是*${user.home}/.m2/repository*。用户可使用 *settings.xml* 文件修改本地仓库。具体内容如下：

```
<?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">
  <!-- 本地仓库配置 -->
  <localRepository>/put/your/local/repository/here</localRepository>
  <!-- 省略，具体信息参考后续内容。 -->
</settings>
```

4.2 在 settings.xml 文件中配置镜像仓库

如果仓库 *A* 可以提供仓库 *B* 存储的所有内容，那么就可以认为 *A* 是 *B* 的一个镜像。例如：在国内直接连接中央仓库下载依赖，由于一些特殊原因下载速度非常慢。这时，我们可以使用阿里云提供的镜像 *http://maven.aliyun.com/nexus/content/groups/public/*来替换中央仓库 *http://repol.maven.org/maven2/*。修改 *maven* 的 *setting.xml* 文件，具体内容如下：

```
<?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">
  <!-- 本地仓库配置
  <localRepository>${user.home}/.m2/repository</localRepository>
  -->

  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
    <pluginGroup>org.jenkins-ci.tools</pluginGroup>
```

```
    </pluginGroups>

    <proxies>
    </proxies>

    <servers>
    </servers>

    <!-- 镜像仓库配置 -->
    <mirrors>
      <!-- 配置具体镜像仓库 -->
      <mirror>
        <!-- 指定镜像 ID -->
        <id>alimaven</id>
        <!-- 指定镜像名称 -->
        <name>aliyun maven</name>
        <!-- 指定镜像路径 -->
        <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
        <!-- 指定此镜像替代中央仓库
          <mirrorOf>central</mirrorOf> ： 匹配中央仓库。
          <mirrorOf>*</mirrorOf> ： 匹配所有远程仓库。
          <mirrorOf>repo1,repo2</mirrorOf> ： 匹配仓库 repo1,repo2，多个使用逗号分
隔。
          <mirrorOf>*,!repo1</mirrorOf> ： 匹配所有远程仓库，repo1 除外。
        -->
        <mirrorOf>central</mirrorOf>
      </mirror>
    </mirrors>

    <profiles>
    </profiles>
</settings>
```

4.3 在 pom.xml 文件中指定远程仓库

如果默认的中央仓库无法满足项目需求，可能需要的构件在另外一个远程仓库，如 *JBoss Maven* 仓库，可以 *POM* 中配置该仓库。

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>

  <repositories>
    <repository>
```

```
        <!--  仓库 id，注意 id 要唯一，如果出现重复会覆盖掉之前的 -->
        <id>jboss</id>
        <!--  仓库名称 -->
        <name>JBoss Repository</name>
        <!-- 仓库地址 -->
        <url>https://repository.jboss.com/maven2/</url>
        <!-- 仓库布局方式为默认 -->
        <layout>default</layout>
        <!-- 是否从此仓库下载快照版本资源 -->
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
        <!-- 是否从此仓库下载发布版本资源 -->
        <releases>
          <enabled>true</enabled>
        </releases>
      </repository>
    </repositories>

</project>
```

4.4 远程仓库的认证

有时候处于安全考虑，需要提供认证信息才能访问一些远程仓库。为了能让 *maven* 访问仓库内容，就需要配置认证信息，认证信息的配置不会在 *pom.xml* 配置，而是在 *settings.xml* 中配置，因为 *pom* 会被提交到代码仓库中供所有成员访问，而 *settings.xml* 一般只放在本机。（此配置常用于私服应用，后续会有详细讲解）假设我在 *pom.xml* 中配置 *id=my-proj* 的远程仓库，需要认证信息，则在 *settings.xml* 中配置如下：

```
<settings>
...
  <servers>
    <server>
        <id>my-proj</id>
        <username>repo-user</username>
        <password>repo-pwd</password>
      </server>
  </servers>
...
</settings>
```

这里的 id=my-proj 一定要和 pom.xml 中仓库的 id 一致，这是它们之间唯一的联系。

settings.xml 的 servers 中就是用来配服务器授权信息的，当然不仅可以配置仓库服务器认证信息，还可以配置其它的比如 tomcat 服务器授权信息也可以在这里配置。

5 仓库优先级问题

本地仓库，镜像仓库，中央仓库，*pom* 文件中指定的远程仓库。
镜像仓库=中央仓库。 镜像仓库是用于替代中央仓库的。
仓库访问优先级：

5.1 本地仓库

第一访问本地仓库。

5.2 指定仓库

如果本地仓库不存在对应信息，访问 *pom* 文件中指定的远程仓库。
这个远程仓库是第二优先级。

5.3 远程仓库

5.3.1 镜像仓库

镜像仓库是 *Maven* 开发过程中的首选远程仓库，在本地仓库和指定仓库无法获取资源的时候，直接访问镜像仓库。

5.3.2 中央仓库

如果镜像仓库不是中央仓库，则会在访问镜像仓库无法获取资源后，访问中央仓库。

二、 Maven 工程

1 工程种类

1.1 POM 工程

用在父级工程或聚合工程中。用来做 *jar* 包的版本控制。常见 *pom.xml* 配置如下：

```
<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>
  <!-- 组名称 -->
  <groupId>groupName</groupId>
  <!-- 工程名称 -->
```

```
    <artifactId>artifactName</artifactId>
    <!-- 版本 -->
    <version>1.0</version>
    <!-- 工程类型 -->
    <packaging>pom</packaging>

    <!-- 定义一个 properties 配置信息。 定义配置信息，为了统一管理。 -->
    <properties>
      <!-- 如果需要使用 properties 配置信息内容。 可以通过表达式${junit.version}访问
4.12 -->
      <junit.version>4.12</junit.version>
    </properties>

    <!-- 定义当前工程都管理了什么 jar，都是什么版本。并非依赖。 -->
    <dependencyManagement>
      <dependencies>
        <!-- 单元测试 -->
        <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>${junit.version}</version>
          <!-- 有效范围， 作用域
           compile - 编译中有效
           runtime - 运行中有效
           system - 全部中有效[默认]
           provided - 当前工程中有效.
           test - 只在测试有效 。
           -->
          <scope>test</scope>
        </dependency>
      </dependencies>
    </dependencyManagement>
    <!-- 定义当前工程依赖了什么 jar 包，版本是什么。 -->
    <dependencies>
      <!-- 单元测试 -->
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
        <!-- 有效范围， 作用域。 -->
        <scope>test</scope>
      </dependency>
    </dependencies>
  </project>
```

1.2 JAR 工程

将会打包成 *jar* 用作 *jar* 包使用。即常见的本地工程 - *Java Project*。常见 *pom.xml* 配置如下：

```
<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>groupName</groupId>
  <artifactId>artifactName</artifactId>
  <version>1.0</version>
  <!-- jar 工程为默认工程类型，可以省略配置 -->
  <!-- <packaging>jar</packaging> -->
</project>
```

1.3 WAR 工程

将会打包成 *war*，发布在服务器上的工程。如网站或服务。即常见的网络工程 - *Dynamic Web Project*。*war* 工程默认没有 *WEB-INF* 目录及 *web.xml* 配置文件，*IDE* 通常会显示工程错误，提供完整工程结构可以解决。常见 *pom.xml* 配置如下：

```
<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>groupName</groupId>
  <artifactId>artifactName</artifactId>
  <version>1.0</version>
  <packaging>war</packaging>
</project>
```

2 工程关系

2.1 依赖

即 A 工程开发或运行过程中需要 B 工程提供支持，则代表 A 工程依赖 B 工程。在这种情况下，需要在 pom.xml 文件中增加下属配置定义依赖关系：

```
<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>
  <!-- 本工程所在组名称 -->
  <groupId>localPrjectGroupName</groupId>
  <!-- 本工程名称 -->
  <artifactId>localPrjectArtifactName</artifactId>
  <!-- 本工程版本 -->
  <version>localPrjectVersionNo</version>
  <!-- 本工程类型 -->
  <packaging>packagingTypeName</packaging>
  <!-- 依赖信息定义 -->
  <dependencies>
    <!-- 定义一个具体的依赖 -->
    <dependency>
      <!-- 依赖的工程所在组名 -->
      <groupId>groupName</groupId>
      <!-- 依赖的工程名 -->
      <artifactId>artifactName</artifactId>
      <!-- 依赖的工程版本 -->
      <version>versionNo</version>
      <!-- 依赖的工程有效范围，其可选值有：
        compile - 编译中有效
        runtime - 运行中有效
        system - 全部中有效[默认]
        provided - 当前工程中有效.
        test - 只在测试有效
      -->
      <scope>system</scope>
    </dependency>
  </dependencies>
</project>
```

2.2 继承

如果 A 工程继承 B 工程，则代表 A 工程默认依赖 B 工程依赖的所有资源，且可以应用 B 工程中定义的所有资源信息。被继承的工程（B 工程）只能是 POM 工程。具体工程 pom.xml 文件配置如下：

2.2.1 父工程配置

```
<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>parentGroup</groupId>
  <artifactId>parentProject</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
</project>
```

2.2.2 子工程配置

```
<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>
  <parent>
    <groupId>parentGroup</groupId>
    <artifactId>parentProject</artifactId>
    <version>1.0</version>
  </parent>
  <!-- 若子工程所在组及版本与父工程一致，可以省略 groupId 和 version 标签配置 -->
  <groupId>childGroup</groupId>
  <artifactId>childProject</artifactId>
  <version>1.0</version>
</project>
```

2.3 聚合

当我们开发的工程拥有 *2* 个以上模块的时候，每个模块都是一个独立的功能集合。比如某大学系统中拥有搜索平台，学习平台，考试平台等。开发的时候每个平台都可以独立编译，测试，运行。这个时候我们就需要一个聚合工程。

在创建聚合工程的过程中，总的工程必须是一个 *POM* 工程（*Maven Project*），各子模块可以是任意类型模块（*Maven Module*）。所有聚合工程和聚合模块必须处于同一个组（*groupId*）中，且聚合工程可以嵌套。

具体 *pom.xml* 文件配置如下：

2.3.1 总工程配置

```
<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>superGroup</groupId>
  <artifactId>superProject</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <!-- 当前聚合工程中的子模块定义，引用的是子模块的 artifactId。 -->
  <modules>
    <module>subProject1</module>
    <module>subProject2</module>
  </modules>
</project>
```

2.3.2 子模块配置

```
<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>
  <parent>
    <groupId>superGroup</groupId>
    <artifactId>superProject</artifactId>
```

```
        <version>1.0</version>
    </parent>
    <artifactId>subProject1</artifactId>
</project>
```

```
<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>
    <parent>
        <groupId>superGroup</groupId>
        <artifactId>superProject</artifactId>
        <version>1.0</version>
    </parent>
    <artifactId>subProject2</artifactId>
</project>
```

三、 Maven 中的常见插件

我们都知道 *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。进一步说，每个任务对应了一个插件目标（*goal*），每个插件会有一个或者多个目标，例如 *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。

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。多年来 *Maven* 社区积累了大量的经验，并随之形成了一个成熟的插件生态圈。*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*。

下面介绍两种简单的常用插件配置。

1 编译器插件

1.1 pom.xml 配置片段

```
    <build>
        <plugins>
        <!-- java 编译插件 -->
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.2</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

1.2 settings.xml 文件中配置全局编译器插件

```
        <profile>
            <!-- 定义的编译器插件 ID，全局唯一 -->
            <id>jdk-1.7</id>
            <!-- 插件标记，activeByDefault 默认编译器，jdk 提供编译器版本 -->
            <activation>
                <activeByDefault>true</activeByDefault>
                <jdk>1.7</jdk>
            </activation>
            <!-- 配置信息 source-源信息，target-字节码信息，compilerVersion-编译过程版
本 -->
            <properties>
                <maven.compiler.source>1.7</maven.compiler.source>
                <maven.compiler.target>1.7</maven.compiler.target>
                <maven.compiler.compilerVersion>1.7</maven.compiler.compilerVersion>
            </properties>
        </profile>
```

2 Tomcat 管理插件

2.1 本地应用

使用 *Tomcat* 插件发布部署并执行 *war* 工程的时候，使用 *maven build* 功能实现。应用启动命令为： *tomcat7:run*。命令中的 *tomcat7* 是插件命名，由插件提供商决定。*run* 为插件中的具体功能。具体 *pom.xml* 文件的配置如下：

```
    <build>
        <plugins>
```

```
        <!-- 配置 Tomcat 插件 -->
        <plugin>
          <groupId>org.apache.tomcat.maven</groupId>
          <artifactId>tomcat7-maven-plugin</artifactId>
          <version>2.2</version>
          <configuration>
            <port>8080</port>
            <path>/</path>
          </configuration>
        </plugin>
      </plugins>
    </build>
```

2.2 远程热部署

远程热部署是指，在 *Tomcat* 容器运行过程中，动态实现 *war* 工程的部署，重新部署功能。使用 *maven build* 功能实现，具体命令为： *tomcat7:deploy* 或 *tomcat7:redeploy*。其中 *deploy* 代表第一次部署 *war* 工程；*redeploy* 代表 *Tomcat* 容器中已有同名应用，本次操作为重新部署同名 *war* 工程。

实现热部署需要远程访问 *Tomcat* 容器，所以 *Tomcat* 容器需要提供合适的访问方式和验证方式。

实现热部署，需要访问 *Tomcat* 容器提供的原始应用 *manager*，并提供有效有权限的访问用户，所以在 *Tomcat* 中也需提供部分配置。具体配置内容如下：

2.2.1 Tomcat 中的 conf/tomcat-users.xml 文件的配置

```
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<user username="tomcatUsername" password="tomcatPassword"
  roles="manager-gui,manager-script"/>
```

2.2.2 pom.xml 文件中的配置

```
    <build>
      <plugins>
        <!-- 配置 Tomcat 插件 -->
        <plugin>
          <groupId>org.apache.tomcat.maven</groupId>
          <artifactId>tomcat7-maven-plugin</artifactId>
          <version>2.2</version>
          <configuration>
            <!-- path: 上传的 war 包解压后的路径命名 -->
```

```
            <path>/ROOT</path>
            <!-- url：上传 war 包到什么位置,除 IP 和端口可以修改外其他不变 -->
            <url>http://ip:port/manager/text</url>
            <!-- 为 tomcat 配置的管理用户名和密码. -->
            <username>tomcatUsername</username>
            <password>tomcatPassword</password>
          </configuration>
        </plugin>
      </plugins>
    </build>
```

四、 Maven 常用命令

1 install

本地安装， 包含编译，打包，安装到本地仓库
编译 -javac
打包 -jar， 将 java 代码打包为 jar 文件
安装到本地仓库 - 将打包的 jar 文件，保存到本地仓库目录中。

2 clean

清除已编译信息。
删除工程中的 target 目录。

3 compile

只编译。 javac 命令

4 deploy

部署。 常见于结合私服使用的命令。
相当于是 install+上传 jar 到私服。
包含编译，打包，安装到本地仓库，上传到私服仓库。

5 package

打包。 包含编译，打包两个功能。

五、 私服

1 什么是私服

私服是一种特殊的远程仓库，它是架设在局域网的仓库服务，私服代理广域网上的远程仓库，供局域网使用。

在企业开发中，私服的建设是有必要的，其好处如下：

1.1 节省资金的外网带宽

利用私服代理外部仓库之后，对外的重复构件下载便得以简化，降低外网带宽压力。

1.2 加速 Maven 构建

不停地连接请求外部仓库是相当耗时的，但是 *maven* 的一些内部机制（如快照更新检查）要求 *Maven* 在执行构建的时候不停地检查远程仓库数据。因此，当项目配置了很多外部远程仓库的时候，构建速度会降低。使用私服解决这问题，因为 *Maven* 只需要检查局域网内私服的数据时，构建速度便有明显提高。

1.3 部署第三方构件

当某个构件无法从任何一个远程仓库获取怎么办？比如 *Oracle* 的 *JDBC* 驱动由于版权原因不能发布到公共仓库中。建立私服后，便可以将这些构件部署到这个内部仓库中，供内部 *Maven* 项目使用。

1.4 提高稳定性，增强控制

对于远程仓库来说，当外网不可用时，*Maven* 构建有可能因为依赖没有下载而不可行，搭建并应用私服后，即使没有外网，如果该构件之前被其它人下载过就会存在私服上，此时再次依赖该构件就可以不用连接外网直接就可以从私服上下载到。同时私服软件（*nexus)*还提供了额外的管理功能。

1.5 降低中央仓库的负荷

中央仓库是有限的。如果所有的 *Maven* 工程开发过程中，都通过中央仓库实现构件的依赖和管理，那么中央仓库的负荷过高，也会严重影响工程构建的效率。如果使用私服，可以分散中央仓库的负荷，只有在私服中没有需要依赖的构件时才会去连接中央仓库。

2 nexus 私服搭建

2.1 搭建环境

环境：*CentOS6.5*、*JDK7*、*Sonatype Nexus*、*Maven*（*Eclipse* 或 *MyEclipse*）

2.2 搭建步骤

2.2.1 安装 JDK

2.2.1.1 JDK 资源包下载

JDK 官方下载地址为：

http://www.oracle.com/technetwork/java/javase/downloads/index.html

本课程使用版本为 *JDK1.7*。（*jdk-7u80-linux-x64.tar.gz*）

2.2.1.2 JDK 资源包解压

在 *Linux* 中安装应用的常用目录为：*/opt* 或*/usr/local* 目录。本课件将 *JDK* 安装到 */usr/local/java* 目录中。

解压 *JDK* 到指定目录：

```
tar -zxvf jdk-7u80-linux-x64.tar.gz -C /usr/local
```

重命名 *JDK* 目录：

```
mv /usr/local/jdk1.7.0_80 /usr/local/java
```

2.2.1.3 环境变量配置

修改*/etc/profile* 配置文件，增加环境变量配置。

```
export JAVA_HOME=/usr/local/java
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$JAVA_HOME/bin:$PATH
```

环境变量修改后，可以重启 *Linux* 实现永久生效；或执行 *source /etc/profile* 命令，让新修改的环境变量在当前终端中生效。

2.2.2 安装私服 Nexus

2.2.2.1 Nexus 资源包下载

Nexus 官方下载地址为：

https://www.sonatype.com/nexus-repository-oss

本课件应用版本为：*nexus-2.11.2-03*

2.2.2.2 Nexus 资源包解压

在*/usr/local* 目录中创建子目录 *nexus*：

```
mkdir /usr/local/nexus
```

解压 *Nexus* 到指定目录：

```
tar -zxvf nexus-2.11.2-03-bundle.tar.gz -C /usr/local/nexus
```

Nexus 压缩包中包含两个子目录*: nexus-2.11.2-03* 和 *sonatype-work*

nexus-2.11.2-03 是具体的私服应用内容，*sonatype-work* 是 *Nexus* 私服下载的构件存放工作目录。

2.2.2.3 检查私服端口和工作目录

在 *nexus-2.11.2-03* 目录中有子目录 *conf*，其中保存私服应用的配置信息。查看 *nexus.properties* 文件，确定私服访问端口和工作目录。此操作可不做任何内容修改。配置文件内容如下：

```
# Jetty section，Nexus 私服应用是使用 Jetty 提供 web 服务的，下述内容为 Jetty 配置。
application-port=8081    # 私服访问端口
application-host=0.0.0.0
nexus-webapp=${bundleBasedir}/nexus    # Nexus 私服 WEB 应用所在位置
nexus-webapp-context-path=/nexus    # Nexus 私服 WEB 应用 contextPath

# Nexus section Nexus 私服配置信息
nexus-work=${bundleBasedir}/../sonatype-work/nexus    # 私服工作目录，即构件保存目录
runtime=${bundleBasedir}/nexus/WEB-INF    # 私服 WEB 应用运行目录
```

2.2.2.4 修改 Nexus 运行用户

Nexus 私服在启动后，私服应用需要访问 *Linux* 的文件系统，所以需要有足够的权限。*Nexus* 的启动脚本文件中，可以指定私服应用的访问用户，此信息在 *nexus-2.11.2-03/bin/nexus* 脚本文件中定义。需要修改的信息如下：

```
# NOTE - This will set the user which is used to run the Wrapper as well as
#    the JVM and is not useful in situations where a privileged resource or
#    port needs to be allocated prior to the user being changed.
#RUN_AS_USER=    #原内容
RUN_AS_USER=root    #修改后的内容，代表 Nexus 私服使用 root 用户权限。
```

2.2.2.5 修改防火墙，开放 Nexus 私服端口访问

修改防火墙配置文件，开放 *Nexus* 私服的访问端口 ***8081***（*2.2.2.3* 章节内容）。

vi /etc/sysconfig/iptables

增加下述内容：

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 8081 -j ACCEPT
```

重新启动防火墙：

service iptables restart

2.2.2.6 启动并测试访问

启动 *Nexus* 私服：

/usr/local/nexus/nexus-2.11.2-03/bin/nexus start

成功启动后，控制台输出内容如下：

```
**********************************************
```

```
WARNING - NOT RECOMMENDED TO RUN AS ROOT
**********************************************
Starting Nexus OSS...
Started Nexus OSS.
```

可通过命令检查私服运行状态：

/usr/local/nexus/nexus-2.11.2-03/bin/nexus status

内容如下为私服运行中：

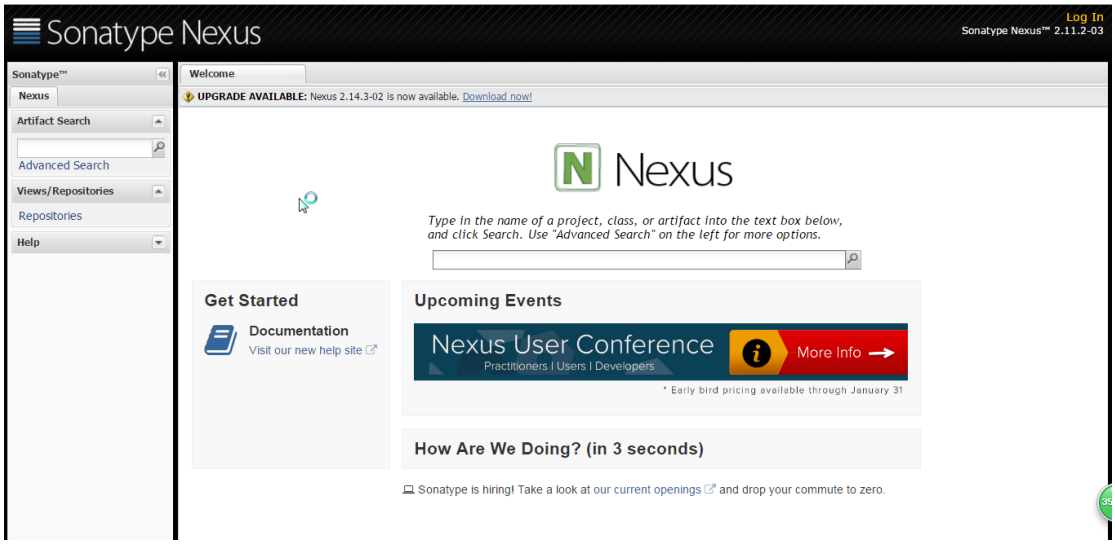
```
**********************************************
WARNING - NOT RECOMMENDED TO RUN AS ROOT
**********************************************
Nexus OSS is running (3883).
```

内容如下为私服未运行：

```
**********************************************
WARNING - NOT RECOMMENDED TO RUN AS ROOT
**********************************************
Nexus OSS is not running.
```

也可使用浏览器访问 *Nexus* 私服 *WEB* 应用， 访问地址为：

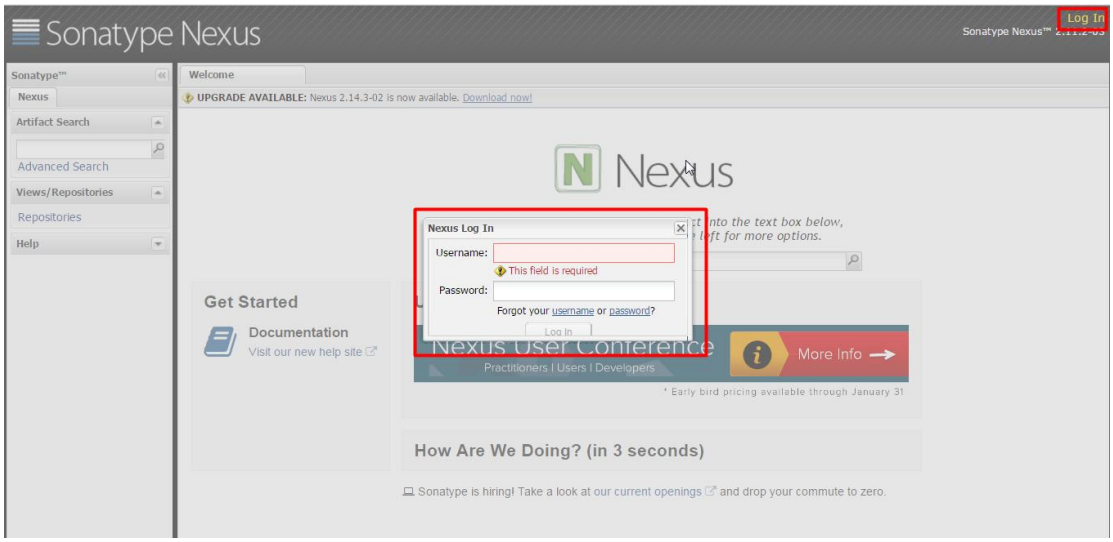
http://ip:8081/nexus （*ip* 为 *Nexus* 所在系统的访问 *IP*），访问效果如下：

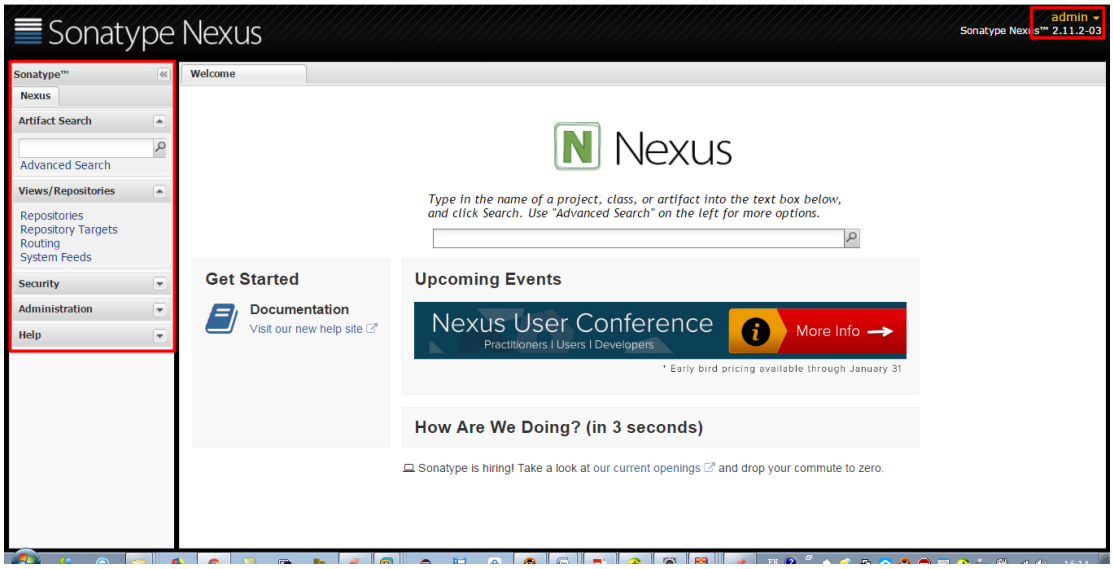


3 私服配置

3.1 登录

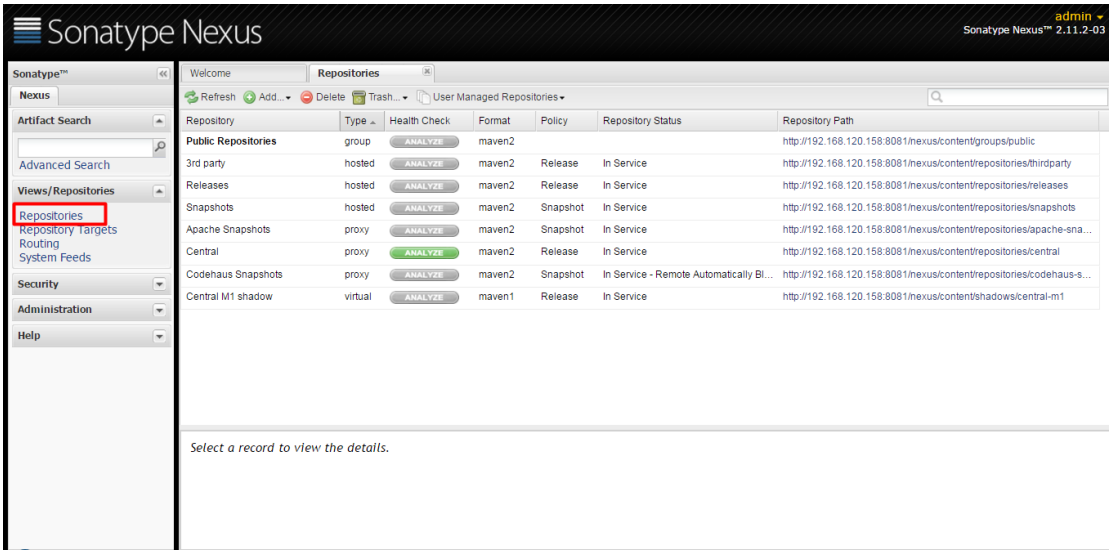
Nexus 默认提供管理用户，用户名为 *admin*，密码为 *admin123*。


Sonatype Nexus
Log In
Welcome
UPGRADE AVAILABLE: Nexus 2.14.3-02 is now available. Download now!
Artifact Search
Advanced Search
Views/Repositories
Repositories
Help
Nexus
Nexus Log In
Username:
This field is required
Password:
Forgot your username or password?
Log In
Get Started
Documentation
Visit our new help site
Nexus User Conference
Practitioners | Users | Developers
More Info
* Early bird pricing available through January 31
How Are We Doing? (in 3 seconds)
Sonatype is hiring! Take a look at our current openings and drop your commute to zero.

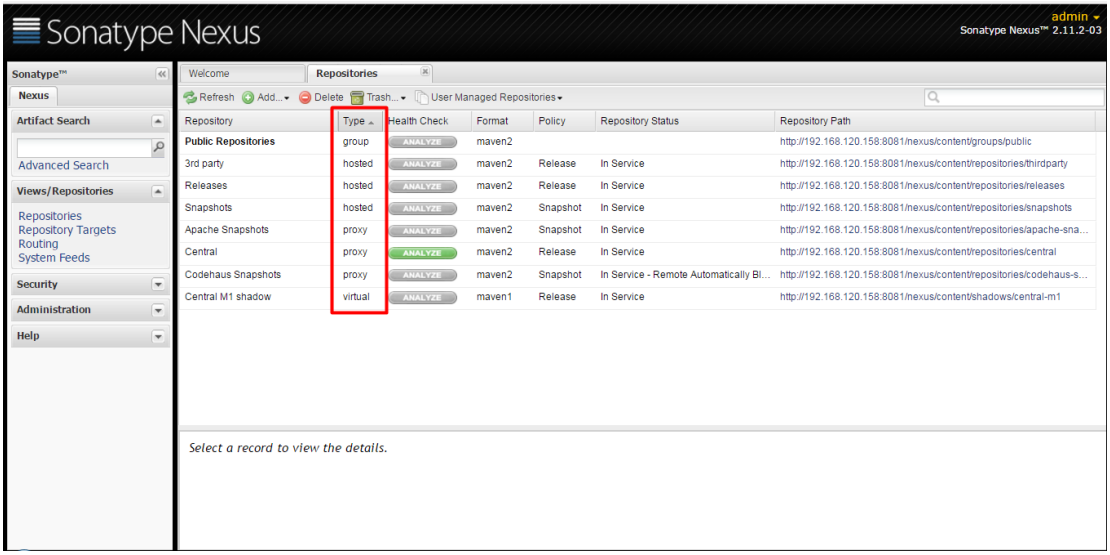

Sonatype Nexus
admin
Sonatype™
Nexus
Artifact Search
Advanced Search
Views/Repositories
Repositories
Repository Targets
Routing
System Feeds
Security
Administration
Help
Welcome
Nexus
Type in the name of a project, class, or artifact into the text box below, and click Search. Use "Advanced Search" on the left for more options.
Get Started
Documentation
Visit our new help site
Upcoming Events
Nexus User Conference
Practitioners | Users | Developers
More Info
* Early bird pricing available through January 31
How Are We Doing? (in 3 seconds)
Sonatype is hiring! Take a look at our current openings and drop your commute to zero.

3.2 仓库管理

3.2.1 检查仓库



3.2.2 仓库类型简述



常用仓库类型为：*hosted* 和 *proxy*。

3.2.2.1 group

仓库组：*Nexus* 通过仓库组来统一管理多个仓库，这样访问仓库组就相当于访问仓库组管理的多个仓库。

3.2.2.2 hosted

宿主仓库：主要用于发布内部项目构件或第三方的项目构件（如购买商业的构件）以及无法从公共仓库获取的构件（如 *oracle* 的 *JDBC* 驱动）。

3.2.2.2.1 releases

发布内部的 *releases* 模块的仓库，所有非快照版本工程都发布到此仓库中。

3.2.2.2.2 snapshots

发布内部的快照模块的仓库，所有工程版本以 *SNAPSHOT* 结尾的都发布到此仓库中。

3.2.2.2.3 3rd party

第三方依赖的仓库，这个数据通常是由内部人员自行下载之后发布上去

3.2.2.3 proxy

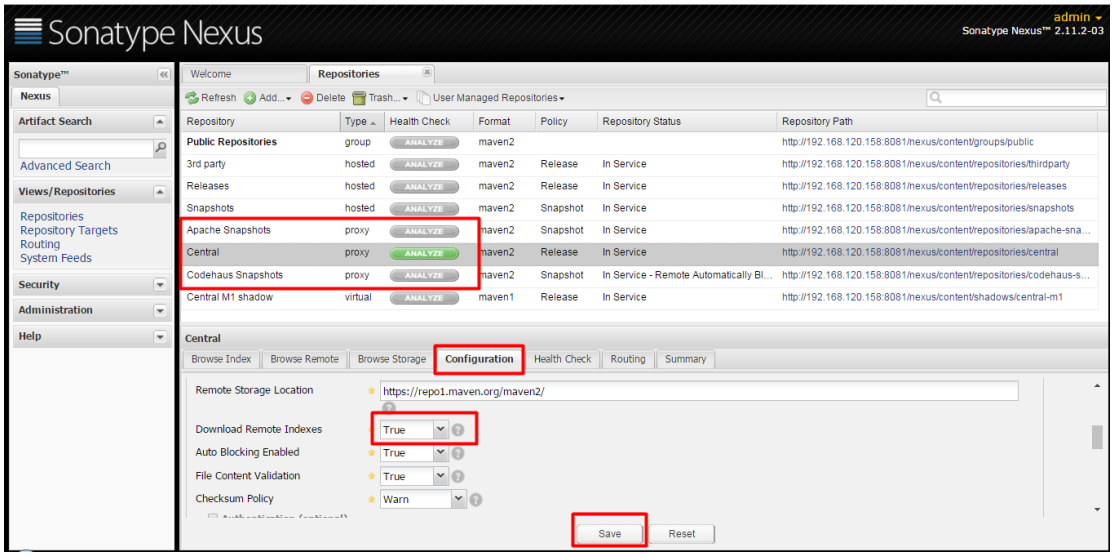
代理仓库：代理公共的远程仓库。

3.2.2.4 virtual

虚拟仓库：用于适配 *Maven 1*。

3.2.3 代理仓库配置

设置 *proxy* 代理仓库*(Apache Snapshots/Central/Codehaus Snapshots)*准许远程下载。



自此 Nexus 私服安装配置结束，私服下载中央仓库资源需要时间，一般企业都会提前搭建。

4 私服应用

要在 *Maven* 工程中使用私服，需要提供私服配置信息。

4.1 settings.xml 文件

```
<?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:/repositories</localRepository>
  <interactiveMode>true</interactiveMode>
  <offline>false</offline>

  <pluginGroups>
    <pluginGroup>org.mortbay.jetty</pluginGroup>
    <pluginGroup>org.jenkins-ci.tools</pluginGroup>
  </pluginGroups>

  <proxies>
  </proxies>

  <servers>
    <server>
      <!-- server 的 id 必须和 pom.xml 文件中的仓库 id 一致 -->
      <id>nexus-releases</id>
      <username>deployment</username>
      <password>deployment123</password>
    </server>
    <server>
      <id>nexus-snapshots</id>
      <username>deployment</username>
      <password>deployment123</password>
    </server>
  </servers>

  <profiles>
    <profile>
      <id>jdk-1.7</id>
```

```
    <activation>
      <activeByDefault>true</activeByDefault>
      <jdk>1.7</jdk>
    </activation>
    <properties>
      <maven.compiler.source>1.7</maven.compiler.source>
      <maven.compiler.target>1.7</maven.compiler.target>
      <maven.compiler.compilerVersion>1.7</maven.compiler.compilerVersion>
    </properties>
  </profile>
  <profile>
    <id>sxt</id>
    <activation>
      <activeByDefault>false</activeByDefault>
      <jdk>1.7</jdk>
    </activation>
    <repositories>
      <!-- 私有库配置 -->
      <repository>
        <!-- 私有库 id -->
        <id>nexus</id>
        <!-- 私有库地址 -->
        <url>http://192.168.120.158:8081/nexus/content/groups/public/</url>
        <!-- 私有库是否支持 releases 版本 -->
        <releases>
          <enabled>true</enabled>
        </releases>
        <!-- 私有库是否支持 snapshots 版本 -->
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <!-- 插件库配置，具体含义私有库配置 -->
      <pluginRepository>
        <id>nexus</id>
        <url>http://192.168.120.158:8081/nexus/content/groups/public/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
```

```
            </pluginRepository>
        </pluginRepositories>
    </profile>
</profiles>

<!-- 激活 profile -->
<activeProfiles>
    <!-- 根据 profile 的 id 标签值激活指定的内容 -->
    <activeProfile>sxt</activeProfile>
</activeProfiles>
</settings>
```

4.2 pom.xml 文件

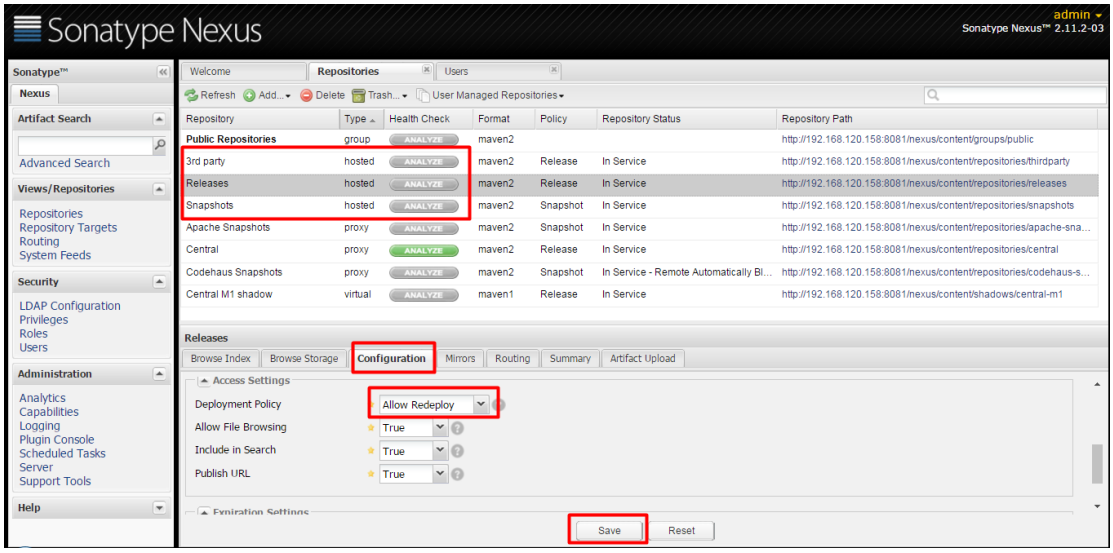
```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>group</groupId>
    <artifactId>project</artifactId>
    <version>1.0</version>

    <distributionManagement>
        <repository>
            <id>nexus-releases</id>
            <name>Nexus Release Repository</name>
            <url>http://192.168.120.158:8081/nexus/content/repositories/releases/</url>
        </repository>
        <snapshotRepository>
            <id>nexus-snapshots</id>
            <name>Nexus Snapshot Repository</name>
            <url>http://192.168.120.158:8081/nexus/content/repositories/snapshots/</url>
        </snapshotRepository>
    </distributionManagement>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-source-plugin</artifactId>
                <version>2.1.2</version>
```

```
            <executions>
                <execution>
                    <id>attach-sources</id>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
</project>
```

4.3 发布本地工程到私服

在 *Maven* 工程的 *maven build* 中，输入命令 *deploy*，即可实现发布工程信息到私服。
如果同版本工程可能多次发布，需要修改 *Nexus* 配置。



4.4 发布三方插件到私服