[1. maven 环境搭建 2](#_Toc24067)

[1.1. 下载maven 2](#_Toc9879)

[1.2. 配置环境 2](#_Toc20866)

[1.3. 修改仓库位置 3](#_Toc3196)

[2. maven 文件结构 3](#_Toc2891)

[2.1. 文件结构 3](#_Toc5850)

[2.2. 命令行 4](#_Toc22184)

[3. maven 坐标 4](#_Toc16507)

[4. maven 依赖 4](#_Toc21825)

[4.1. 概述 4](#_Toc30302)

[4.2. 依赖范围 4](#_Toc12838)

[4.3. 传递性依赖 5](#_Toc10533)

[4.4. 依赖调解 5](#_Toc28182)

[4.5. 可选依赖 6](#_Toc26278)

[4.6. 排除依赖 7](#_Toc5953)

[4.7. 归类依赖 7](#_Toc9533)

[4.8. 优化依赖 8](#_Toc4052)

[5. maven 仓库 8](#_Toc30366)

[5.1. 何为仓库 8](#_Toc25093)

[5.2. 分类 8](#_Toc17965)

[5.3. 本地仓库 8](#_Toc21897)

[5.4. 远程仓库 9](#_Toc10029)

[5.4.1. 配置 9](#_Toc22923)

[5.4.2. 认证 10](#_Toc6464)

[5.4.3. 部署 10](#_Toc12199)

[5.5. 中央仓库 10](#_Toc7693)

[5.6. 私服 10](#_Toc26488)

[5.7. 镜像 11](#_Toc4369)

[6. 生命周期 11](#_Toc30635)

[6.1. 概念 11](#_Toc31451)

[6.2. 三套生命周期 11](#_Toc26822)

[6.3. clean 12](#_Toc9028)

[6.4. default 12](#_Toc26365)

[6.5. site 12](#_Toc618)

[6.6. 命令行与生命周期 13](#_Toc31687)

[7. 插件 13](#_Toc28275)

[7.1. 目标 13](#_Toc29975)

[7.2. 绑定 13](#_Toc5213)

[7.2.1. 内部绑定 13](#_Toc28518)

[7.2.2. 自定义绑定 14](#_Toc31851)

[7.3. 配置 15](#_Toc24810)

[7.3.1. 命令行方式 15](#_Toc7925)

[7.3.2. pom全局配置 15](#_Toc14257)

[7.3.3. pom中插件任务配置 15](#_Toc29849)

[7.3.4. 16](#_Toc27005)

[7.4. 16](#_Toc1319)

[8. 聚合 16](#_Toc2964)

[9. 继承 16](#_Toc21183)

[10. maven 测试 16](#_Toc31192)

[10.1. 职责 16](#_Toc10994)

[10.2. maven-surefire-plugin 测试运行器 16](#_Toc4071)

[10.3. 跳过测试 16](#_Toc30187)

[10.3.1. 跳过测试运行 16](#_Toc19488)

[10.3.2. 跳过测试代码的编译和运行（不推荐） 17](#_Toc7151)

[10.4. 18](#_Toc13228)

[11. 与Hudson集成 18](#_Toc7082)

[12. 构建web应用 18](#_Toc18435)

[12.1. 与jetty集成 18](#_Toc42)

[12.1.1. 目的 18](#_Toc8593)

[12.1.2. 在pom.xml 18](#_Toc7033)

[12.1.3. setting.xml文件 18](#_Toc8279)

[12.1.4. 运行jetty 19](#_Toc21296)

[12.2. 与Cargo集成 19](#_Toc27325)

[12.2.1. 目的 19](#_Toc30332)

[12.2.2. 部署到本地web容器 20](#_Toc29693)

[12.2.3. 部署到远程web容器 22](#_Toc4457)

[13. 版本管理 24](#_Toc20168)

[13.1. 概述 24](#_Toc3955)

[13.2. 版本管理VS版本控制 24](#_Toc30220)

[13.3. 快照VS发布 24](#_Toc20392)

[13.4. 版本转换 24](#_Toc17675)

[13.5. 升级条件 25](#_Toc7913)

[13.6. 发布流程 25](#_Toc24875)

[13.7. 版本号含义 25](#_Toc28929)

[13.8. 版本号作用 25](#_Toc11035)

[13.9. 主干、标签与分支 25](#_Toc14254)

[13.10. 私塾 25](#_Toc14911)

[14. 灵活的构建 25](#_Toc28379)

[14.1. 背景 25](#_Toc15504)

[14.2. Maven属性 26](#_Toc28552)

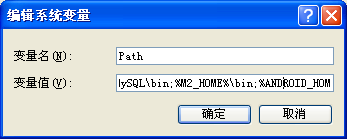
[14.3. Profile 26](#_Toc9799)

[14.4. 资源过滤 26](#_Toc26437)

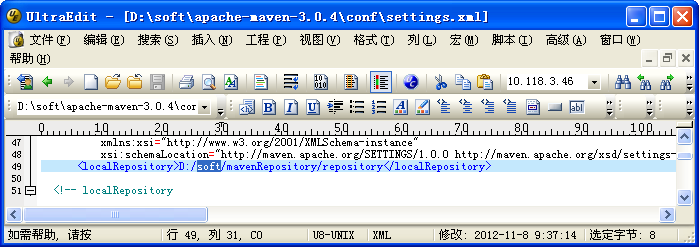
[15. 生成项目站点 26](#_Toc14347)

[16. maven web应用 26](#_Toc38)

1. maven 环境搭建
   1. 下载maven
   2. 配置环境



* 1. 修改仓库位置



1. maven 文件结构
   1. 文件结构

如com.st.maven 项目 blog

根目录

pom.xml:

<modelVersion>4.0.0</modelVersion>

<groupId>com.st.maven</groupId>

<artifactId>blog</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifaceId>junit</artifactId>

<version>4.10</version>

</dependency>

</dependencies>

src/main/java:主代码

src/test/java:测试代码

target:编译后放置在该文件夹中。

* 1. 命令行

编译 mvn clean compile

测试 mvn clean test

打包 mvn clean package

给其他项目引用 mvn clean install

1. maven 坐标

groupId：隶属的实际项目

必填

artifactId：实际项目的一个maven项目（模块名称）

version：版本

packing：打包方式，默认为jar

classifier：附加属性，不能直接定义 如blog-1.0-SANPSHOT.jar，blog-blog-1.0-SANPSHOT-source.jar，blog-1.0-SANPSHOT-doc.jar。source,doc就是附加属性。

1. maven 依赖
   1. 概述

……

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifaceId>junit</artifactId>

<version>4.10</version>

</dependency>

</dependencies>

……

groupId,artifactId,version

type:对应项目的packaging

scope:依赖的范围

optional:依赖是否可选

exclusions:排除传递性依赖依赖

* 1. 依赖范围

compile、test、provided、runtime、system、import

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 依赖范围 | 对于编译有效 | 对于测试有效 | 对于运行有效 | 例子 |
| compile | Y | Y | Y | Spring-core |
| test |  | Y |  | Junit |
| provided | Y | Y |  | Servlet |
| runtime |  | Y | Y | Jdbc驱动 |
| system | Y | Y |  | 本地maven库 |

其中：

compile:scope没有指定，默认使用该范围。

system：与provided依赖范围相同，必须通过指定<systemPath>，往往与本机系统绑定，不可移植。

<dependency>

<groupId>javax.sql</groupId>

<artifactId>jbdc-stdext</artifactId>

<version>2.0</version>

<scope>system</scope>

<systemPath>${java.hom}/lib/rt.jar</systemPath>

</dependency>

import：对三种classpath没有实际影响。

* 1. 传递性依赖

背景：比如，blog依赖spring-core，而spring-core又依赖其他jar包，如commons-logging，这样手动下载很麻烦。而传递性依赖就很好的解决了这一问题，可以以传递性的方式引入这些jar包。

blog

spring-core

commons-logging

blog有一个compile范围的依赖spring-core，spring-core有一个compile范围的依赖commons-logging，所以，commons-logging是blog的传递性依赖，范围为compile。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | compile | test | provided | runtime |
| compile | compile |  |  | runtime |
| test | test |  |  | test |
| provided | provided |  | provided | provided |
| runtime | runtime |  |  | runtime |

* 1. 依赖调解

背景：

如项目A有这样的依赖关系：

a）路径最近者优先：

A---->B---- > C----->X(1.0)

A---- >D---🡪X(2.0)

则X(2.0)是A的传递性依赖。

b）第一声明者优先：

A---->B----->X(1.0)

A---- >D---🡪X(2.0)

则X(1.0)是A的传递性依赖。

* 1. 可选依赖

在理想情况下，是不使用可选依赖的。

背景：

如项目A

A🡪B->X(可选)

A->B->Y(可选)

项目B的依赖声明如下：

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.st.maven<groupId>

<artifactId>project-b</artifactId>

<version>1.0.0</version>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.10</version>

<optional>true</optional>

</dependency>

<dependency>

<groupId>postgresql</groupId>

<artifactId> postgresql </artifactId>

<version>8.4-701.jdbc3</version>

<optional>true</optional>

</dependency>

</dependencies>

</project>

可选依赖不被传递，因此，A中需要显示声明

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.st.maven<groupId>

<artifactId>project-a</artifactId>

<version>1.0.0</version>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.10</version>

</dependency>

<dependency>

<groupId>com.st.maven</groupId>

<artifactId> project-b </artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

</project>

* 1. 排除依赖

背景：传递性依赖会给项目隐式性的引入很多依赖，有些不稳定的依赖会直接影响到当前的项目，因此要在这个项目中排除这些不稳定的依赖，并声明稳定的依赖。

即：A->B->C(?)

A->B，A->C。

解决方法：

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.st.maven<groupId>

<artifactId>project-a</artifactId>

<version>1.0.0</version>

<dependencies>

<dependency>

<groupId>com.st.maven</groupId>

<artifactId> project-b </artifactId>

<version>1.0.0</version>

<exclusions>

<exclusion>

<groupId>com.st.maven</groupId>

<artifactId>project-c</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>com.st.maven</groupId>

<artifactId> project-c </artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

</project>

* 1. 归类依赖

背景：在A项目中有很多关于spring-framewoke的依赖，如spring-core-2.5.6、srping-beans-2.5.6、spring-context-2.5.6。当升级spring-framework的时候2.5.6版本都得改掉，比较麻烦。

解决方法：

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.st.maven<groupId>

<artifactId>project-a</artifactId>

<version>1.0.0</version>

<properties>

<springframework.version>2.5.6</springframework.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId> spring-core </artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId> spring-beans </artifactId>

<version>${springframework.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId> spring-cotext </artifactId>

<version>${springframework.version}</version>

</dependency>

</dependencies>

</project>

* 1. 优化依赖

查看当前项目已解析依赖 mvn dependency:list

查看某个依赖通过哪条路径引入 mvn dependency:tree

分析当前项目的依赖 mvn dependency:analyse

1. maven 仓库
   1. 何为仓库

存储所有maven项目共享构件的位置

* 1. 分类

分为本地仓库和远程仓库。

Maven根据坐标查找构件，先查找本地仓库，如不存在或查看有无更新的构件版本，则查找远程仓库。

* 1. 本地仓库

默认情况下，不管是window还是linux系统，每个用户在自己的用户目录下都有一个路径为.m2/repository/的仓库目录。如用户名为XX的用户，在window上的本地仓库地址为C:\Users\XX\.m2\repository\。而linux上的为/home/XX/.m2/repository/。

默认情况下，~/.m2/setting.xml文件是不存在的。用户可以从$M2\_HOME/conf/settings.xml拷贝编辑。

* 1. 远程仓库

如果不执行任何的Maven命令，本地仓库目录是不存在的。

每个用户只有一个本地仓库，但可以配置访问多个远程仓库。

* + 1. 配置

很多情况下，默认的中央仓库不能满足项目的需求，可能项目需要的构件在另外一个远程仓库中，如JBOSS Maven仓库。可以在pom.xml中配置repository信息。



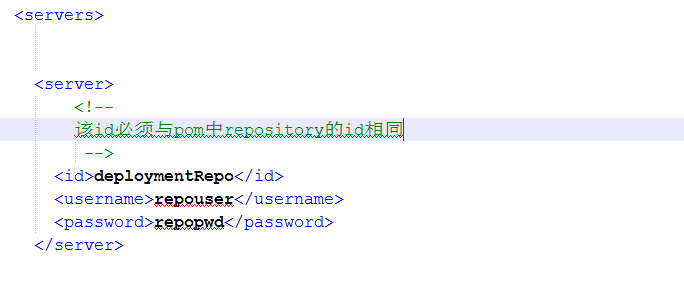
用户可以使用mvn clean install-U强制检查更新。

* + 1. 认证

大部分远程仓库无须认证就可以访问，但出于安全考虑，需要提供认证信息才能访问远程仓库。如组织内部的maven仓库，为了防止非法访问，管理员为每个仓库设置了用户名和密码。此时，为了访问远程仓库，就得配置信息。

仓库信息可以配置在pom文件中，而认证信息必须配置在settings.xml文件中。因为pom文件需提交到代码仓库中供团队成员使用，而setttings.xml文件一般放在本机中，这样，比较安全。

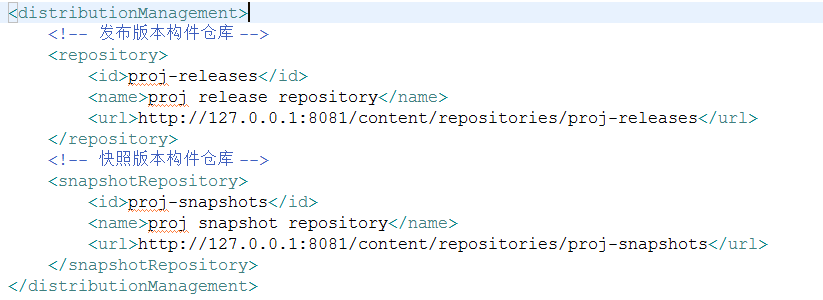
如需要一个id为deploymentRepo的认证信息，在setttings.xml文件中配置server信息



* + 1. 部署

无论是日常生活中使用的构件还是正式版本发布的构件，都需要部署到仓库中，供团队其他成员使用。

首先，编辑pom.xml文件，配置distributionManagement信息



往远处仓库部署构件的时候，往往需要认证（参照5.4.2）。

不管从远程仓库下载构件还是部署构件到远程仓库，配置认证的信息都一样。

执行 mvn clean deploy进行部署。

* 1. 中央仓库

中央仓库是默认的远程仓库。M2\_HOME\lib\maven-model-builder-3.2.1.jar\org\apache\maven\model下的pom.xml自带仓库位置。

* 1. 私服

私服是特殊的远程仓库。

* 1. 镜像

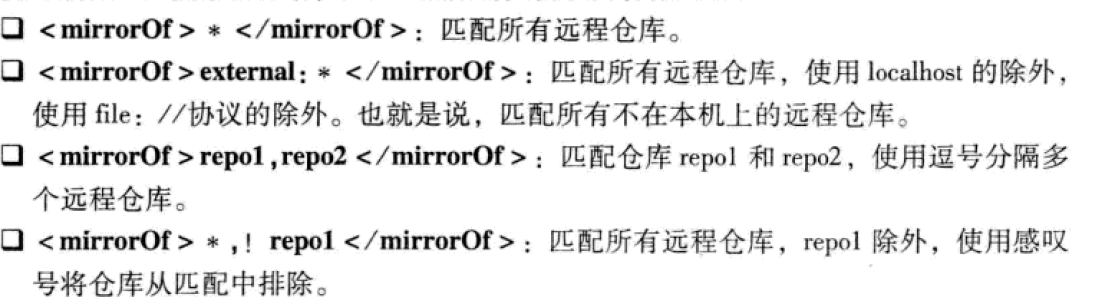
如果仓库X可以提供仓库存储的所有内容，可以认为X是Y的镜像。

如http://maven.oschina.net/content/groups/public/是中央仓库的镜像，

可以在settings.xml中配置mirror信息



为了满足一些复杂的需求，支持复杂的镜像配置



1. 生命周期
   1. 概念

对所有的构建过程进行抽象和统一。

Maven的生命周期是抽象的，本身不做任何事情，都交由插件完成。

* 1. 三套生命周期

Maven拥有3套独立的生命周期。分别为clean、default和site。

每个生命周期包含一些阶段，这些阶段都是有顺序的，并且后面的阶段依赖前面的阶段。

* 1. clean

目的：清理项目

阶段：

1. pre-clean:执行一些清理前需要完成的工作。
2. clean:清理上一次构建生成的文件。
3. post-clean:执行以下清理后需要完成的工作。
   1. default

真正构建时所需要执行的所有步骤。

所有生命周期中最核心的部分

阶段：

1. validate
2. Initialize
3. Generate-sources:
4. Process-sources:处理项目主资源文件。一般来说，是对src/main/resoureces目录的内容进行变量替换等工作后，复制到项目输出的主classpath目录中。
5. Generate-resources
6. Process-resources
7. Compile:编译项目的主源码。一般来说，是编译src/main/java目录下的java文件到主classpath目录中。
8. Process-classes
9. Generate-test-sources
10. Process-test-sources：处理项目测试资源文件。
11. Generate-test-resources
12. Process-test-resources
13. Test-compile：编译项目的测试代码。
14. Process-test-classes
15. Test：使用单元测试框架进行测试。
16. Prepare-package
17. Package：接受编译好的代码，打发成可发布的格式
18. Pre-integration-test
19. Integration-test
20. Post-integration-test
21. Verify
22. Install：将包安装到maven本地仓库，共
23. Deploy：将最终的包复制到远程仓库，
    1. site

目的：建立和发布项目站点

阶段：

1. pre-site:执行一些在生成项目站点之前需要完成的工作
2. site:生成项目站点文档
3. post-site:执行一些在生成项目站点之后需要完成的工作
4. site-deploy:将生成的项目站点分布到服务器上
   1. 命令行与生命周期

从命令行执行maven任务的目的是调用生命周期阶段。

mvn clean：该命令调用clean生命周期的clean阶段。

mvn test：该命令调用default生命周期的test阶段。

mvn clean install：该命令调用clean生命周期的clean阶段及default生命周期的test阶段。

mvn clean deploy site-deploy：该命令调用clean生命周期的clean阶段、default生命周期的deploy阶段，以及site生命周期的site-deploy阶段。

1. 插件
   1. 目标

如maven-dependency-plugin，它能够分析项目依赖，帮助找出潜在的无用依赖等功能，

这些功能聚集在一个插件里，每个功能就是一个插件目标。

maven-dependency-plugin有十多个目标，每个目标对应了一个功能，对应的插件目标为dependency:analyze、dependency:tree等。

冒号前面是插件前缀，后面是插件目标。

* 1. 绑定

生命周期的阶段与插件的目标相互绑定，以完成某个具体的构建任务。

* + 1. 内部绑定

为了能让用户几乎不用任何配置就能构建maven项目，maven在核心为一些主要的生命周期阶段绑定了很多插件的目标。

|  |  |
| --- | --- |
| 生命周期阶段 | 插件目标 |
| pre-clean |  |
| clean | maven-clean-plugin:clean |
| post-clean |  |

1、clean生命周期与插件目标的内置绑定关系

|  |  |
| --- | --- |
| 生命周期阶段 | 插件目标 |
| pre-site |  |
| site | maven-site-plugin:site |
| post-site |  |
| site-deploy | maven-site-plugin:deploy |

2、site生命周期与插件目标的内置绑定关系

default生命周期的阶段和插件目标的绑定关系由项目打包类型所决定。

|  |  |  |
| --- | --- | --- |
| 生命周期阶段 | 插件目标 | 执行任务 |
| process-resources | maven-resources-plugin:resources | 复制主资源文件到主输出目录 |
| compile | maven-compile-plugin:compile | 编译主代码到主输出目录 |
| process-test-resources | maven-resources-plugin:testResources | 复制测试资源文件到测试输出目录 |
| test-compile | maven-resources-plugin:testCompile | 编译测试代码到测试输出目录 |
| test | maven-surefire-plugin:test | 执行测试用例 |
| package | maven-jar-plugin:package | 创建项目jar包 |
| install | maven-install-plugin:install | 将项目输出构件安装到本地仓库 |
| deploy | maven-deploy-plugin:deploy | 将项目输出构件部署到远程仓库 |

3、default生命周期与插件目标的内置绑定关系及任务（打包类型：jar）

* + 1. 自定义绑定

用户可以自己选择将某个插件目标绑定到生命周期的某个阶段上。

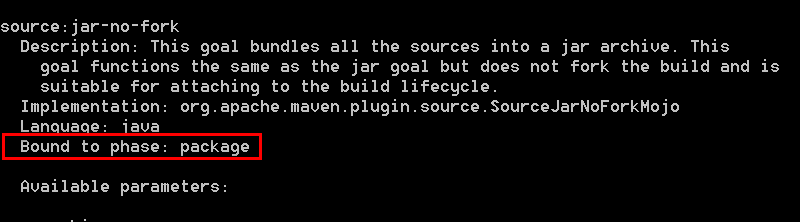
一个常见的例子，打包项目的源码。可以使用maven-source-plugin实现该任务。它的jar-no-fork目标能够将项目的主代码打包成jar文件，可以将其绑定到default生命周期的verify阶段上，在执行完集成测试后和安装构件前创建源码jar包。



运行mvn verify可以看到如下结果



有时候，即使不通过phase配置生命周期阶段，也能够绑定到生命周期中去。因为很多插件的目标在编写时已经定义了默认绑定阶段。可以使用maven-help-plugin查看插件详细信息。



如果不指定phase参数，该插件目标默认绑定到package阶段。

当插件目标被绑定到不同的生命周期阶段的时候，执行顺序由生命周期阶段的先后顺序决定。

当多个插件目标被绑定到同一个生命周期阶段，执行顺序由插件目标的声明顺序决定。

* 1. 配置

完成生命周期阶段与插件目标的绑定后，还可以通过配置插件目标，调整插件模板执行的任务。可以通过命令行和pom配置等方式配置参数。

* + 1. 命令行方式

很多插件的目标都支持从命令行配置。可以在maven命令中使用-D参数，并伴有key=value(键值对)的形式，来配置插件目标的参数。

如maven-surefire-plugin提供了maven.test.skip参数。

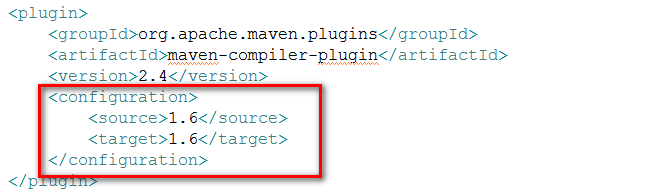


* + 1. pom全局配置

并不是所有的插件参数都适合从命令行配置，如有些差数的值不会改变或很少改变，在pom中一次性配置比重复在命令行输入方便。

用户可以在声明插件的时候，对此插件进行一个全局的配置。

如配置maven-compiler-plugin编译java 1.6版本的源代码，生成与JVM1.6 兼容的字节码文件。



这样不管是绑定到compile阶段的maven-compiler-plugin:compile任务，还是绑定到test-compiler阶段的maven-compiler-plugin:testCompiler任务，就都能使用该配置。

* + 1. pom中插件任务配置

用户可以为某个插件任务配置特定的参数。



1. 聚合
2. 继承
3. maven 测试
   1. 职责

自动运行单元测试，通过maven-surefire-plugin与主流的单元测试框架JUnit3、JUnit4以及TestNG集成，并且自动生成丰富的结果报告。

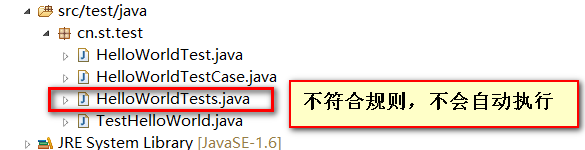
* 1. maven-surefire-plugin 测试运行器

自动执行检测源码路径(src/test/java)下所有符合一组命名模式的测试类

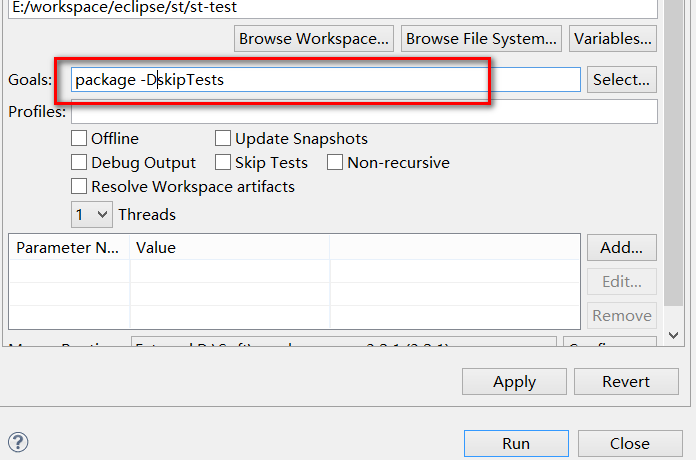
\*\*/Test\*.java

\*\*/\*Test.java

\*\*/\*TestCase.java



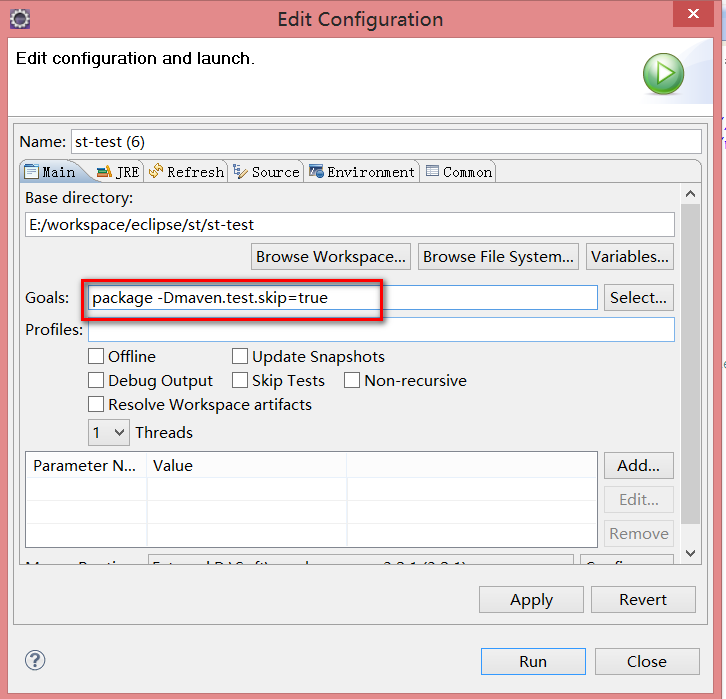
* 1. 跳过测试
     1. 跳过测试运行



也可以在pom.xml配置maven-surefire-plugin插件(不推荐)



* + 1. 跳过测试代码的编译和运行（不推荐）



也可以在pom.xml配置maven-compiler-plugin和maven-surefire-plugin

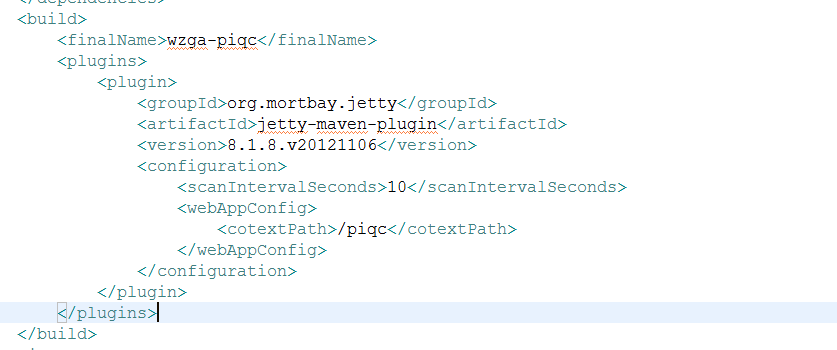




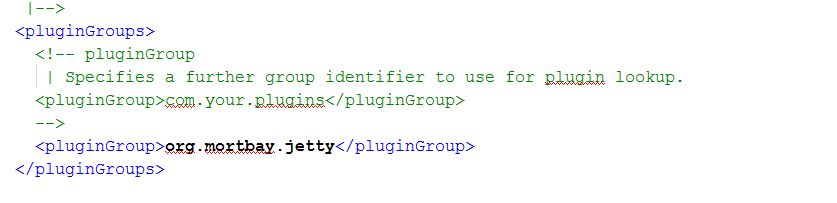

1. 与Hudson集成
2. 构建web应用
   1. 与jetty集成
      1. 目的

帮助日常的快速开发和测试

* + 1. 在pom.xml

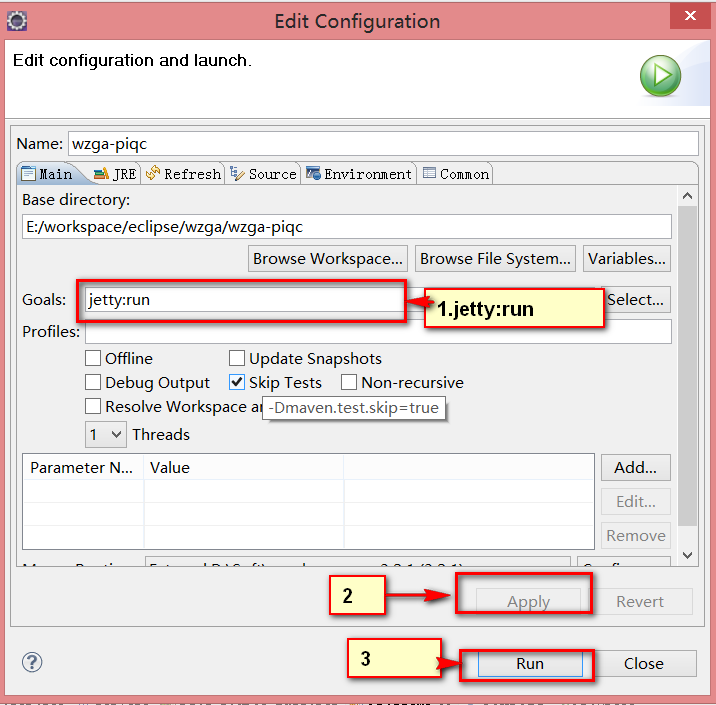
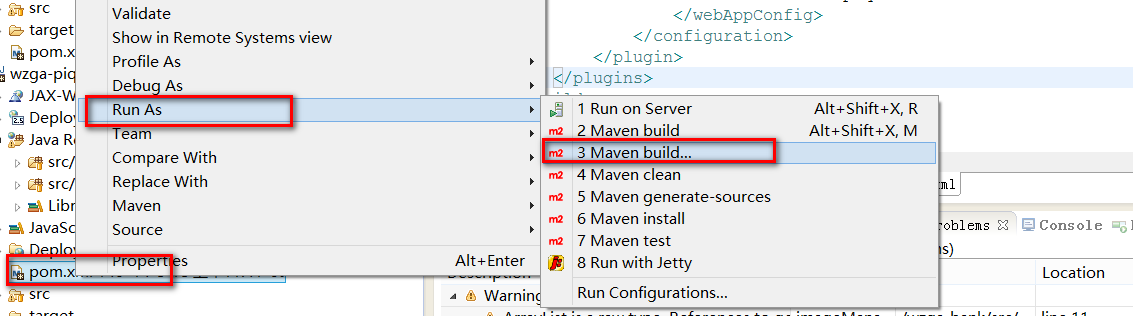


* + 1. setting.xml文件



默认情况下，只有org.apache.maven.plugins和org.codehaus.mojo两个groupId下的插件才支持简化的命令行调用。需要在settings.xml做如上配置，才支持命令行调用。

* + 1. 运行jetty



如果希望使用其他端口，mvn jetty:run -Djetty.port=9999

* 1. 与Cargo集成
     1. 目的

自动化部署。

* + 1. 部署到本地web容器

Cargo支持两种本地部署。分为standalone模式和existing模式。

* + - 1. standalone

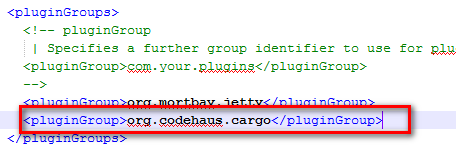
将现有的web容器拷贝到用户指定文件目录，并在该文件目录基础上部署应用。

* + - 1. existing

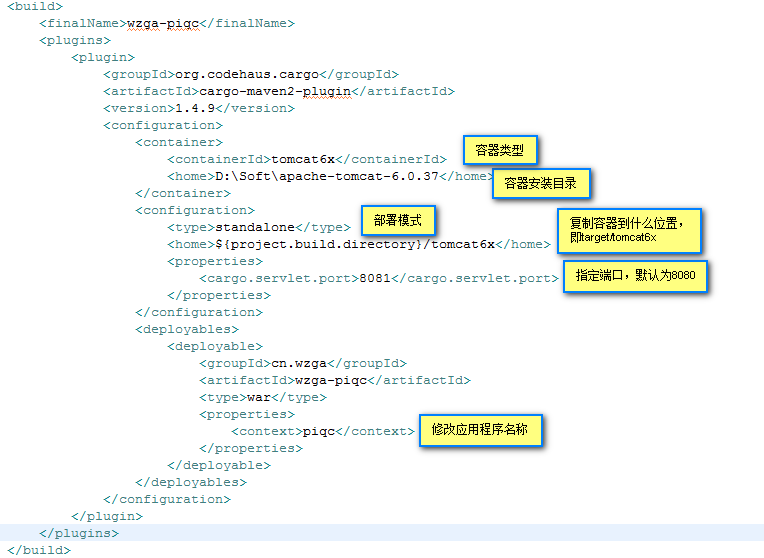
将应用直接部署到现有的web容器中

* + - 1. 配置

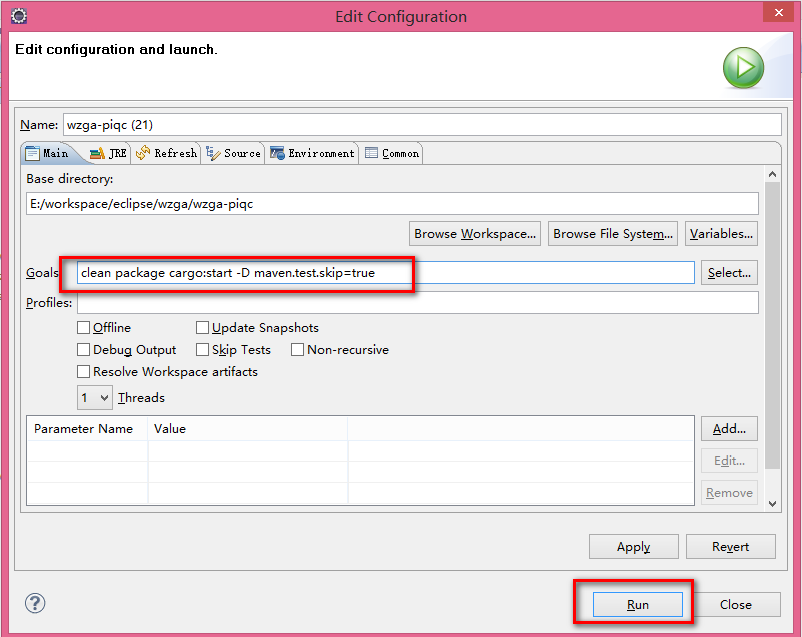
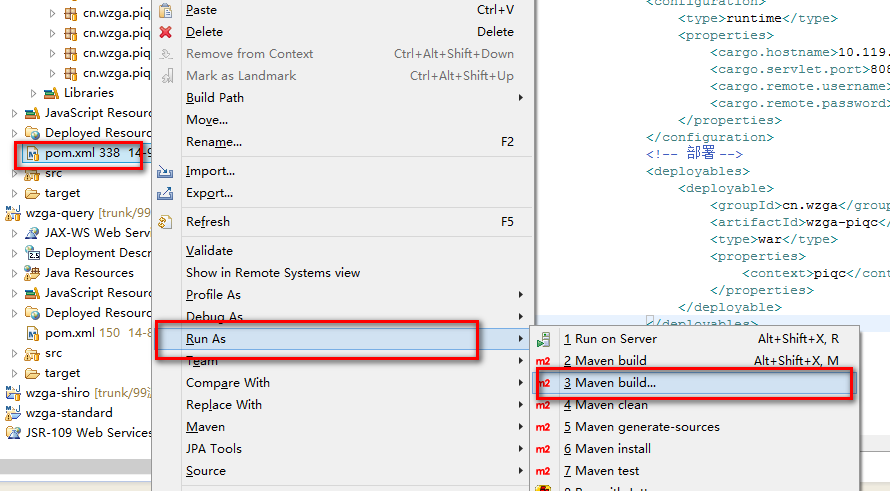
setting.xml



pom.xml



部署

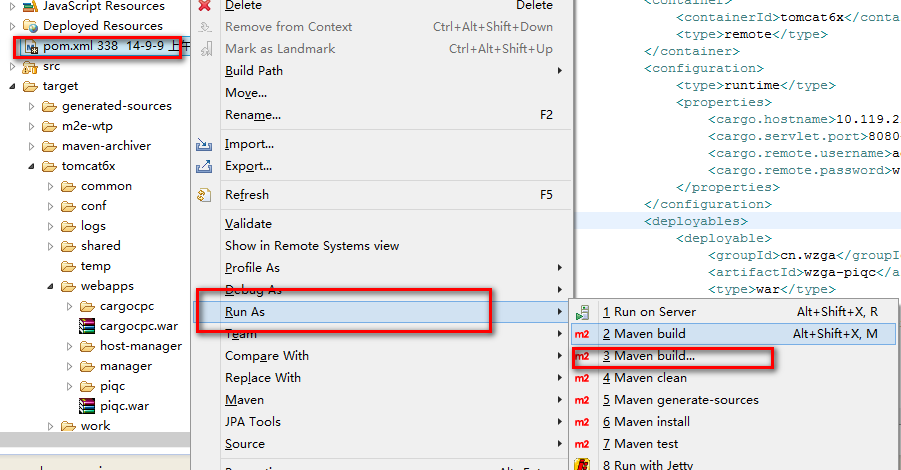


* + 1. 部署到远程web容器

pom.xml



部署



1. 版本管理
   1. 概述

一个健康的项目都有一个长期、合理的版本演变过程。如JUnit有3.7、3.8、4.0、4.1等版本。Maven本身的版本也比较多，如maven1，maven2(2.0.9，2.0.10)，maven3（3.0-alpha-1，3.0-alpha-2）。



* 1. 版本管理VS版本控制

版本管理：项目整体版本的演变过程管理，如从1.0-SNAPSHOT到1.0，再到1.1-SNAPSHOT.

版本控制：借助版本控制工具（SVN）监控代码的每一个变更。

* 1. 快照VS发布

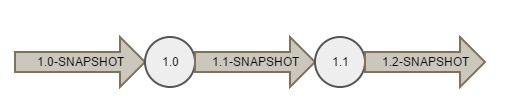
在项目开发过程中，应使用快照版本，MAVEN能解析项目各个模块最新的快照。

快照能促进团队内部交流。

但当项目对外发布时，我们需要提供非常稳定的版本。保证永远定位到唯一的构建。不能像快照版本那样，随时发生改变。

* 1. 版本转换

项目经过一段时间1.0-SNAPSHOT的开发后，在某一时刻发布1.0正式版，然后项目又进入1.1-SNAPSHOT的开发，这个版本可能添加一些有趣的特性，在某一时刻发布1.1正式版，然后项目又进入1.2-SNAPSHOT的开发。



如图所示，快照版本对应了项目的开发过程，对应了很长的时间，正式版本对应了项目的发布，代表某个时刻的状态。

* 1. 升级条件
* 所有自动化测试应当全部通过。
* 项目中没有配置任何快照版本的依赖。
* 项目没有配置任何快照版本的插件。
* 项目所包含的代码全部提交到版本控制系统中。
  1. 发布流程
* 更新项目的快照版本号为发布版本
* 执行一次maven构建（确保项目的状态是健康的）
* 提交变更到版本控制系统的主干中
* 为当前主干打上标签
  1. 版本号含义

<主版本>.<次版本>.<增量版本>-<里程碑版本>

* 主版本：表示了项目的重大架构变更。如struts 1和struts 2
* 次版本：表示了较大范围的功能添加和变化，及bug修复。如Nexus 1.5比Nexus 1.4增加了LDAP的支持。当总体架构没有什么变化。
* 增量版本：一般表示重大Bug的修复
* 里程碑版本：往往指某一版本的里程碑。往往不是很稳定，需要很多测试。

注意：并不是每个版本号都拥有这四个部分。一般来说，主版本和次版本都会声明。

* 1. 版本号作用

当用户在声明依赖或插件未声明版本是，maven会根据版本号约定解析最新版本。

规则如下：

* 对于前3个部分，比较是基于数字的(1.5>1.4>1.3.11>1.3.9)
* 对于里程碑部分，进行简单的字符串比较（1.2-beta-3>1.2-beta-11）
  1. 主干、标签与分支
* 主干：项目开发代码的主体，从项目开始到当前都处于活动状态。从这里可以获取项目的最新代码及几乎所有的变更历史。
* 分支：从主干的某个点分离出的代码拷贝。在这里，可以在不影响主干的情况下，进行bug修复或进行实验性质的开发。如果该分支到达了预期目标，通常发生在该分支的变更都会合并到主干中。
* 标签：用来标识主干或者分支的某个点的状态，代表项目的某个稳定状态。一般是版本发布时的状态。

1. 灵活的构建
   1. 背景
2. 典型的项目都会有开发、测试、产品环境，这些环境的数据库配置可能不尽相同，那么构建的时候要识别正确的环境并使用对应的数据库配置。
3. 项目开发集成了大量的测试，运行这些测试很耗费时间，这些测试并不是每次构建项目的时候都要运行。
   1. Maven属性
   2. Profile
   3. 资源过滤
4. 生成项目站点
5. maven web应用