宏天软件X5基于Maven的框架使用约束

Winston Yan

2013-11-17

# 前言

本文档只是针对基于Maven构建的X5开发框架进行阐述，对Maven的基本原理和知识如项目涉及会做些说明，否则不做冗述。

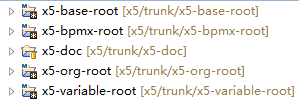
# 多模块项目说明

## 项目划分

项目的基本结构已经搭设好，一般情况下，项目和模块均由项目经理或架构师负责创建。

目前整个X5产品划分为5个子项目，除去x5-doc是用于文档，所以代码开发用的项目是4个。

项目命名如下图：



## 多模块项目的好处

传统的方式，是在一个项目中，通过java包或者目录的方式来划分代码，但是由于代码依然是归属一个项目下面，所以无论编译和发布均需要整体进行。

随着开发的进行，由于进度、功能复杂度、开发人员水平不一等各种原因，代码之间的调用纵横交错，导致耦合越来越高。

比如要进行一个简单的单元测试，由于需要启动Spring容器而不得不同时启动工作流引擎（尽管该测试和工作流无关），这样到时开发效率低下，重用和重构困难。

而用多模块项目替代包层次的划分能给我们带来如下好处：

1. **方便重用**。

如果你有一个新的项目需要用到x5-base-core和x5-base-api，添加对它们的依赖即可，你不再需要去依赖整个x5项目（即使其中一些功能该项目根本不需要）。

x5-base-core包含一些基础的工具类，所有项目均可使用。通过模块的开发方式，这部分可以独立开发、维护、编译和生成jar包，这也是模块化最重要的一个目的。

1. **按需依赖**

对于前面提到的单元测试，只有在需要工作流时，才需要依赖x5-bpmx-api和x5-bpmx-activiti。否则只需要依赖最基础的模块（x5-base-core)即可。

1. **优化pom.xml文件，防止过于庞大**

由于多模块可以采用逐层继承的方式，所以管理上可以持续优化。

每个模块的配置都在各自的pom.xml里，不用再到一个混乱的纷繁复杂的总的POM中寻找自己的配置，而且通过继承，模块的pom.xml将会非常简洁。

1. **模块的独立编译**

如果你只是在x5-base-core上工作，你不再需要build整个项目，只要在x5-base-core目录运行mvn命令进行build即可，这样可以节省时间，尤其是当项目越来越复杂，build越来越耗时。

1. **模块的独立发布**

对于只负责某个模块开发的技术人员，在修改了该模块后，只需要对该模块进行独立的编译和发布即可。这样其它模块的开发人员只需要调整依赖该模块的版本号即可完成更新。

1. **项目的限制性修改**

某些项目和模块，如x5-base-root下面的模块被所有项目依赖，但不可以给所有人修改，通过多模块项目的开发方式，现在我们将这些底层代码分离出来，作为独立的项目，这样svn只给特定的人访问，但仍可以通过发布jar方式给其它各个项目使用。

而传统的结构，由于开发人员可以修改所有的代码，只能通过项目管理规范禁止开发人员修改底层代码，这无疑是增加了管理成本。

1. **代码保护**

由于代码分布在多个项目中，所以svn权限自然可以分别管控。

当新启动项目时，只需按需引入已有的项目，开发时根据api文档即可，而无需必须对开发人员开放所有源码。

1. **Maven对多模块项目的友好支持。**

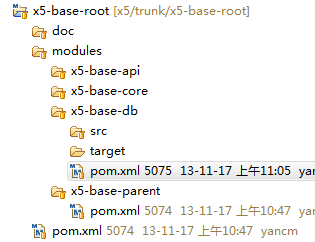
如parent、DepencencyManagement、PluginManagement等。

## x5-base-root 说明

x5-base-root是最基础的项目，下面将以该项目作为例子进行阐述。

### 项目结构

整个项目结构如下图：



项目的根目录有：

* doc：放置该项目的一些文档、数据库设计、脚本文件等。
* modules：放置该项目包括若干个模块，每个模块有独立的src和pom.xml

modules下分为多个模块，其中parent后缀的模块为其它模块的父模块，只有 pom.xml文件，用于维护项目的个性配置。

* pom.xml：项目pom文件。用于定义常用的依赖、发布管理和插件。

### Pom划分思路

采用3层pom.xml文件的结构：

1. 项目pom：维护项目和用户无关的依赖、插件和其它，用于优化配置。

新建项目时只需要复制过去，更名和修改modules里面的包含模块即可。

1. parent模块的pom：维护该项目的公共依赖引入、插件声明和配置以及项目或用户相关的一些参数配置。
2. 其它模块的pom：维护模块本身的依赖引入，注意只能引入在上层（项目pom或公共pom）已经定义好的依赖。

经过这样三层划分后，模块就可以继承公共和项目的pom的设置，在一个项目内，对同一个依赖包的引入，就可以保证同样的版本号scope了。

而且由于模块自行维护依赖引入，那么要知道该模块依赖了那些包也就很清晰了，这样调整起来也比较容易。如果没有使用继承，要在一个共同的pom.xml中找到某个模块的依赖那是一件困难的事情。

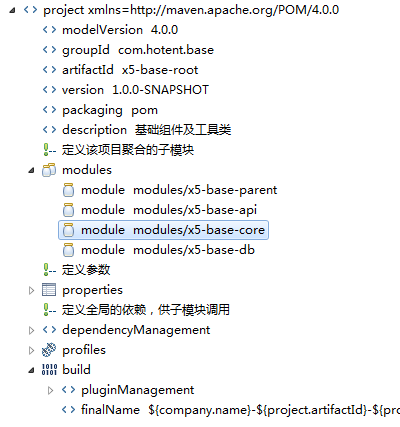
### 项目POM

#### 结构说明

项目根目录的pom.xml大部分内容均是项目无关的，所以用户相关或项目相关的配置尽量不要放在此文件中管理，而是应该放在parent模块下进行维护。

**每新建一个项目都可以从其它项目中复制该pom.xml文件过来，然后做适当调整，具体会在下面章节说明。**

该文件结构如下：



1. 命名

该pom的artifactid命名为x5-base-root（请和项目名一致，以便区分），表示是项目pom。

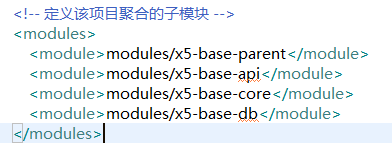
1. packaging=pom

表示其它的pom可以继承。

1. modules

配置包含那些子模块。

内容是模块相对根目录的相对路径。



其中x5-base-parent仅是维护一个pom.xml文件，没有其它文件。

1. properties

定义参数，主要是在一个位置配置所有依赖包、插件的版本号等，方便修改。

1. dependencyManagement

用于配置供子模块引入的依赖包的相同因素，保证每个子模块引入依赖包时，版本和scope保持一致。

1. profiles-->profile-->distributionManagement

配置分发管理，定义发布仓库和镜像仓库。

根据需要这里可以按需管理多个profile。

1. build\pluginManagement

配置供子模块引入的插件的相同因素，包括版本和其它的配置参数。

#### 使用规范

##### 维护规范

* 限定的维护人员

项目pom.xml 由项目经理或架构师等负责统一维护。

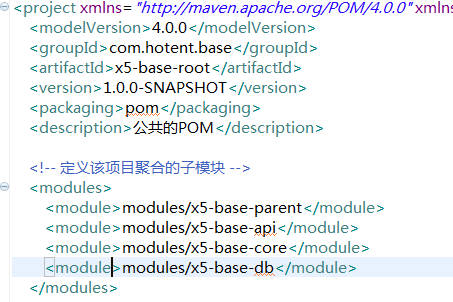
开发人员如需增加新的jar调用，则需要提出申请，由项目经理统一在pom.xml中增加。然后开发人员在相应项目的子模块下的pom.xml中调用。

* 在该文件维护modules。
* 尽量在该文件维护对外的依赖（dependencyManagement），以便保持整个项目的依赖的版本和scope一致。
* 在该文件中引入插件（pluginManagement），但对于有项目个性化设置的插件，请放在parent模块下的pom.xml，尽量降低本pom.xml的项目相关性。

##### 项目相关的内容

虽然大部分项目相关的内容都移到了parent模块下的pom.xml中，但是项目根目录的pom.xml还是有少量项目相关的内容的。

如图：



包括groupId、artifactId、version、description和modules的子项。

除此之外，其它的项目相关的内容请不要放在此文件中。

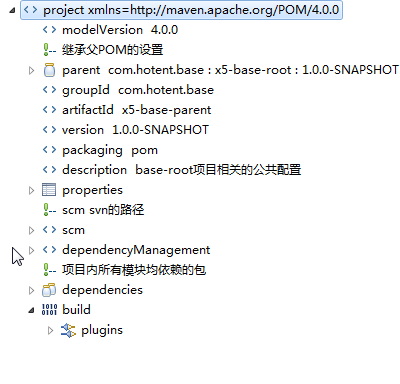
### parent模块的pom.xml

#### 结构说明

位置：每个项目均有一个后缀为parent的模块，如x5-base-parent。

模块说明：表示该项目其它模块的父pom.xml，继承项目pom.xml，设置项目相关的内容。包括其它子模块默认引入的依赖和插件，以及其它的配置。

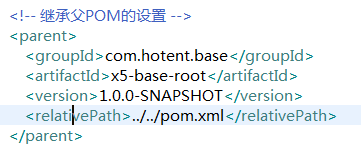
结构图如下：



从结构图可以看到：

1. parent

配置对项目pom.xml的继承。



1. dependencyManagement



这里包括两类的引入：

* 第三方包的引入：一般这些第三方包应该在项目的pom.xml中引入，而不是在这里，但是可以在这里作为过渡。
* 本项目或其它项目的模块jar的引入：在这里定义的好处是统一版本号。

1. scm

配置svn路径，供项目和svn集成使用。

1. dependencies

配置项目内的子模块默认引入的依赖包。

1. build

配置项目内的子模块默认引入的插件。

#### 使用规范

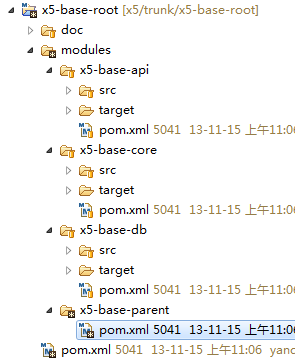
xxx-parent模块的pom.xml只允许项目经理、组长或架构师维护，开发人员不能越权修改。

开发人员在所负责的模块中，如果需要增加新的依赖时，需要告知上级，由上级在xxx-parent的pom.xml中增加（dependencyManagement中，版本号通过参数管理），然后开发人员在模块的pom.xml中引入（无需版本号）

### 其它模块pom.xml

#### 结构说明

所有的模块均放在modules下面，在每个模块下均有一个pom.xml文件，如下图：



下面看看x5-base-core的pom.xml文件



1. 设置parent

继承“项目pom.xml”，通过artifactId、groupId、version和relativePath进行定位。

1. dependency

根据需要引入，这些依赖包均在项目pom.xml或者xxx-parent的pom.xml中定义了，在这里只需要引入，而不需要设置版本号和scope。

对于对其它模块的依赖引入，请放在最上面，方便查看。

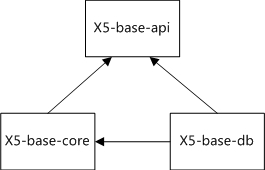
1. 插件

一般在parent模块的pom.xml中统一引入，这里无需设置，的确有必要时请联系项目经理。

#### 使用规范

## 项目内模块之间的依赖

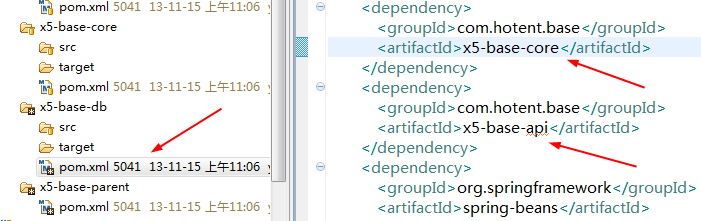
在x5-base-root中有三个模块，它们的依赖关系如下图：



注：箭头表示依赖的方向。

从图中所示，**x5-base-db**依赖另外两个模块。

所以在该模块的pom.xml中，存在这样的依赖配置：

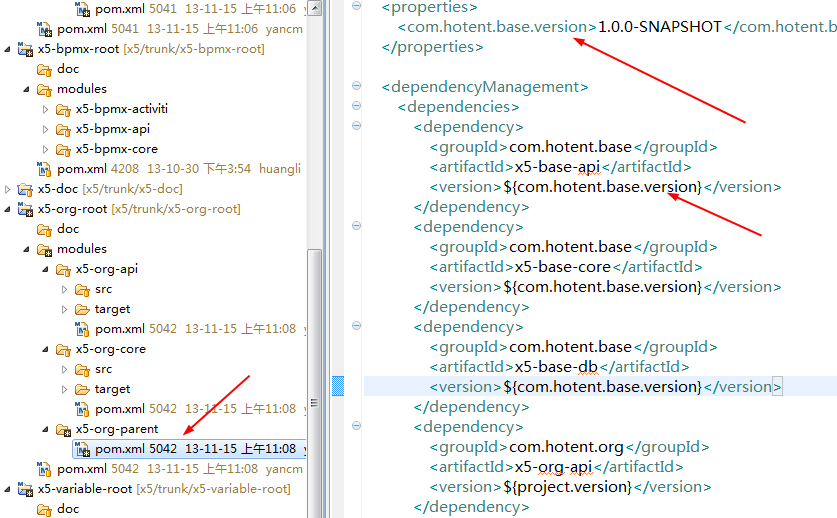


也就是说，如同引入其它的第三方依赖包一样的引入其它模块的依赖。因为这些模块编译后也同样是jar包。

## 项目之间的依赖

项目之间的依赖也是依赖到其它项目的模块，而不是直接项目本身。

比如x5-org-root的x5-org-parent模块的pom.xml文件：

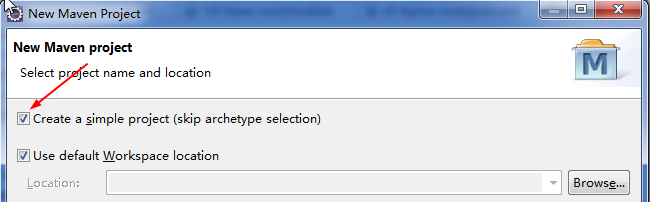


1. 在x5-org-root项目中，由于需要依赖x5-base-root项目下的模块，所以在x5-org-parent模块的pom.xml中统一引入依赖，由于该项目下的几个模块的版本是一致的，所以将版本号作为参数抽离。
2. 对于本模块的其它模块的引入，则使用project.version参数。

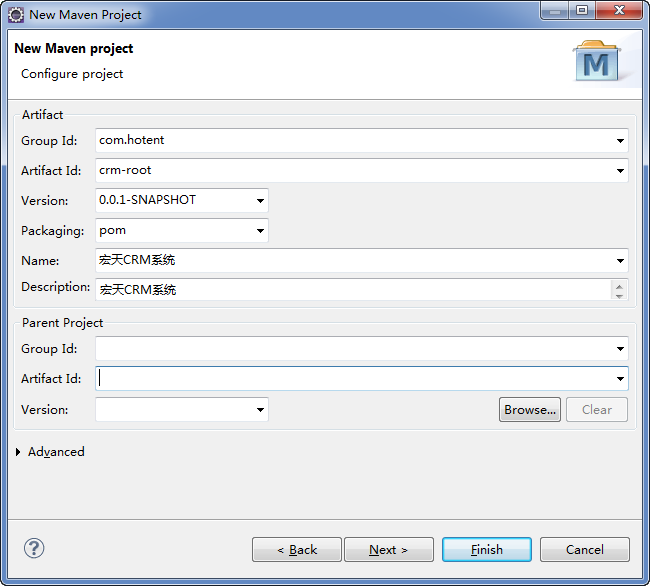
# 项目和模块的日常开发

## 新建项目

通过eclipse的向导新建一个maven project：



点击Next



注意：

Artifact Id：增加root后缀（因为是多模块项目，这样容易区分）

packaging：pom

点击next --> Finish 后，生成crm-root项目，接下来做3步。

1. 打开x5-base-root的项目pom.xml

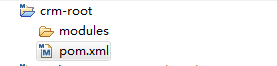
（或其它更加新的项目的pom.xml）

将<modules>以下到</project>之前的内容全部复制。

1. 打开crm项目的pom.xml，将复制的内容粘贴到其中。
2. 将crm\pom.xml中<modules></modules>里面的内容删除。

1. 删除crm\src目录，并在项目中增加doc和modules目录。

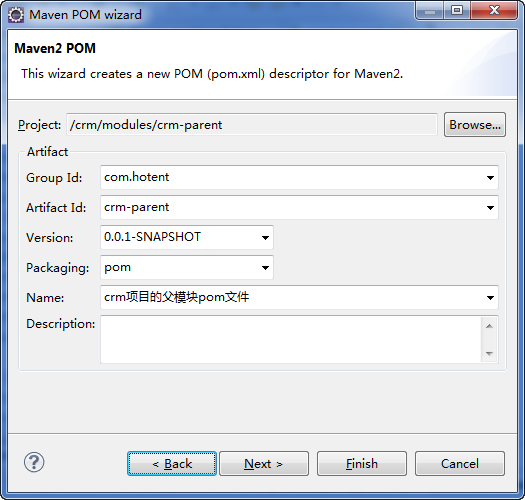
完成项目新建工作，结果如下图：



## 新建parent模块

1. 在crm-root\modules 目录下，创建以parent作为后缀的目录，如crm-parent。

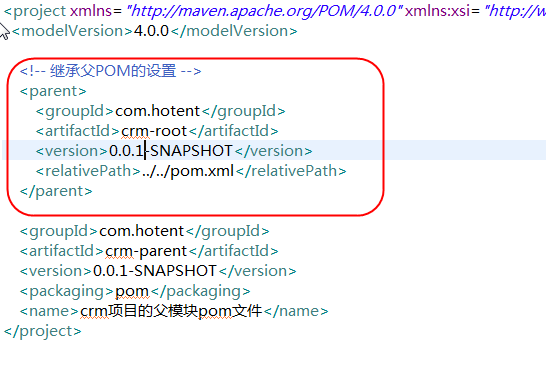
1. 右键点击crm-parent，点击new--> other-->maven-->Maven POM file，通过向导创建pom.xml。



点击Finish，完成创建。

1. 打开modules\crm-parent\pom.xml，切换到xml视图。

增加parent部分的内容：

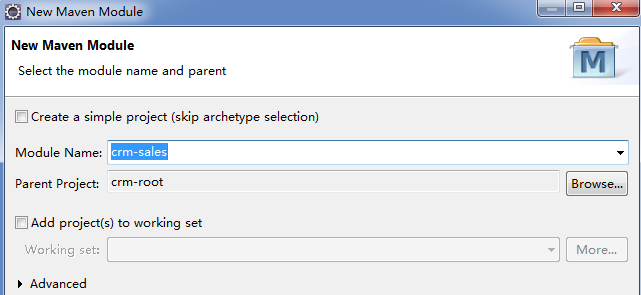


1. 打开crm-root\pom.xml，在modules中，增加该模块的关联



## 新建其它模块

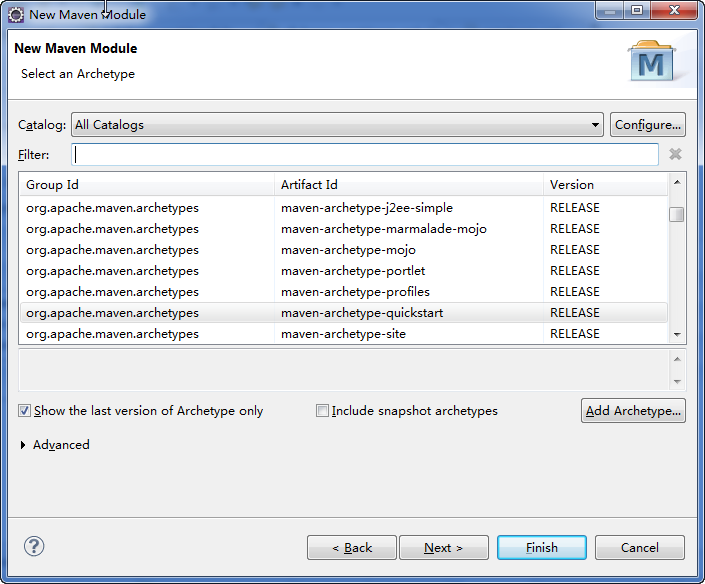
1. 右键点击父工程项目（在本例即是crm项目），选择New -> Other -> Maven -> Maven Module。



Create a simple project... 的复选框不要勾选。

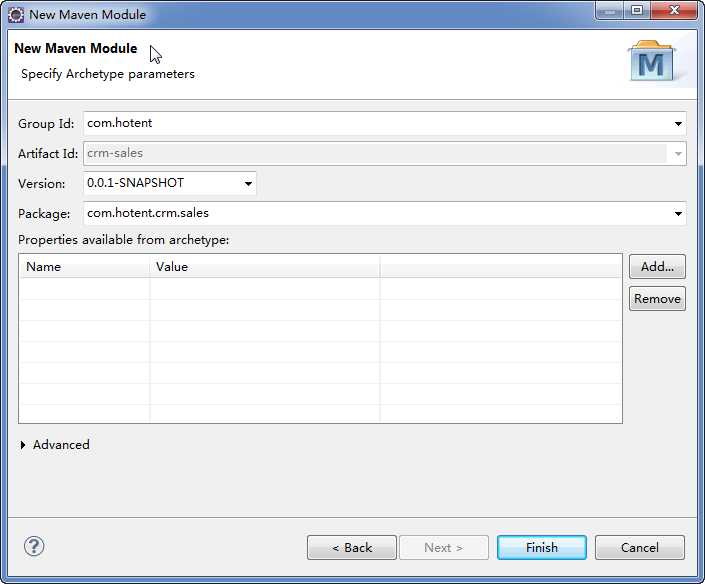
输入模块名，点击下一步。

1. 选择Archetype



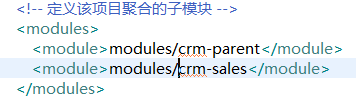
点击Next。

1. 输入Archetype参数



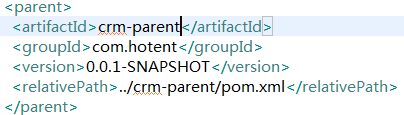
点击Finish。

1. 移动crm-sales目录到modules里面
2. 修改crm-root\pom.xml中



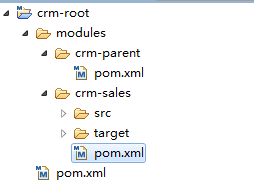
增加modules/ 路径前缀

1. 修改crm-root\modules\crm-sales\pom.xml



**重要**：

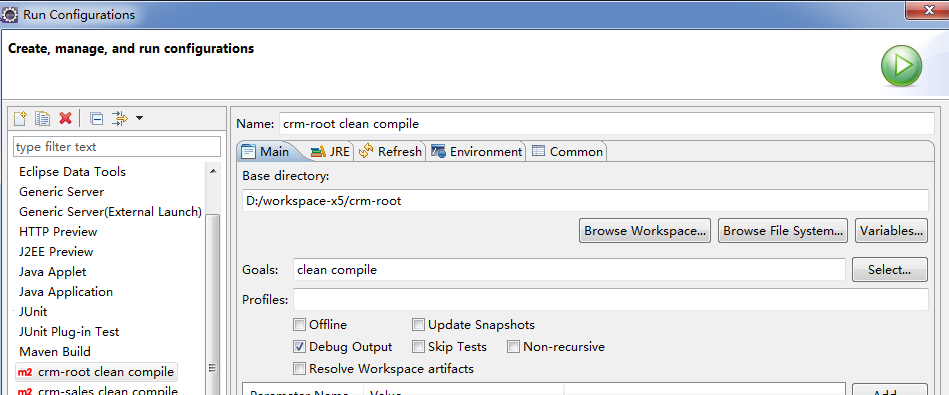
1. artifactId改为crm-parent
2. relativePath改为../crm-parent/pom.xml
3. 模块新建完成，如下图：



## 新建命令配置

### 新建项目编译命令

打开菜单Run --> Run Configurations



弹出窗口如上图：

在Maven Build栏目下，若还没有创建命令配置，则点击上面工具栏的“新建”（第一个箭头）；如果已创建，则选择该命令，点击“复制”（第二个箭头）。

在新建或复制创建后的命令设置界面：

1. Name：建议格式：

[项目名]-[模块名] [插件/命令] [目标] [参数]

（冒号不允许，用空格分隔）

在这里输入：crm-root clean compile

1. Base directory：项目根目录路径或者模块路径

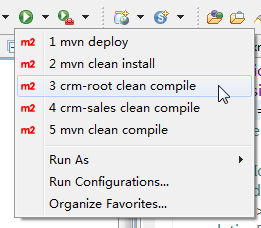
1. Goals：命令目标

在这里输入 clean compile

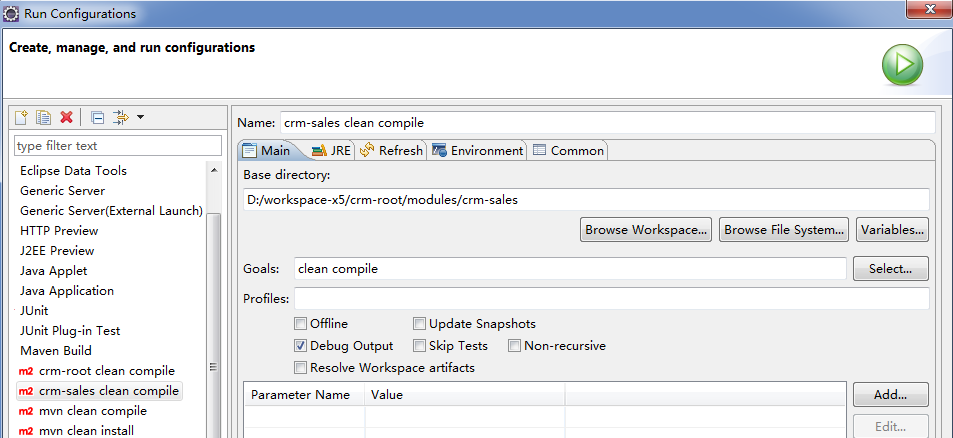
1. Debug Output：一般勾选，用于输出命令执行时更多的日志。

1. Maven Runtime：如果指向eclipse默认的maven，请点击右侧的configure设置系统安装的Maven，这样的好处时，和在cmd命令行中执行的maven版本和配置保持一致。

在配置界面点击Run，或者在外部点击：



### 新建模块编译命令



1. Name：增加模块名部分。

即crm-sales clean compile

1. Base directory：

指向模块的位置。

其它的同上一节。

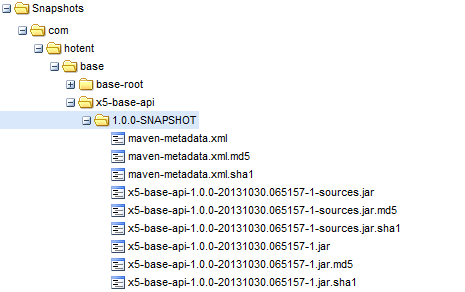
## 日常代码提交 - deploy

版本号的变更由版本管理员（可以是项目经理兼任）管理，开发人员不能变更版本号。

在日常开发过程中，开发人员所负责的模块可能会被其它的模块所依赖，那么他负责的模块有代码变更时，就需要快速的发布出去，供其它模块负责人使用。

所以在日常代码开发中，开发人员需要经常依次使用install 和 delopy将本模块的代码发布到Maven仓库服务器上。

由于使用了快照版本管理方式，所以开发人员发布的jar是不会更改版本号的，而是在Maven快照仓库中，同一个快照版本下，增加了一个新的快照版本（有时间戳后缀）。



## 依赖管理

项目的依赖分为三类：

1. 项目的公共依赖（dependencies）

指设置在xxx-parent模块的pom.xml中的公共依赖，这类依赖，它的子模块会自动继承。虽然可以减少子模块的pom.xml的配置，但是当新增模块时，你是否确认这个模块必定需要该依赖呢？

所以请谨慎使用公共依赖。

1. 项目的公共依赖管理（dependencyManagement ）

这是最为保险的，所有的项目相关的公共依赖管理均应该放在xxx-parent的pom.xml中。

如果经项目组讨论，觉得这个依赖在其它项目也会用到，那么就应该进一步的将它放在项目根目录下的pom.xml。

（可参看2.3.4）

1. 模块的依赖

开发人员在开发某模块时，如果觉得需要增加一个jar引入时，需要告知上级，由其将依赖加入到xxx-parent模块的pom.xml，即子模块不能直接声明依赖的版本和scope。

即使由于其它情况临时在子模块中加入依赖，也需要尽快的将其移到xxx-parent的pom.xml中。

## 插件依赖

1. 项目特性无关的插件

在项目根目录的pom.xml中的pluginManagement声明。

1. 项目相关，但各个模块公用的插件

在xxx-parent的pom.xml中声明，可以直接在plugins中声明，而不用

pluginManagement，当然，如果你觉得这个插件可能只是部分模块用到，那可以用

pluginManagement。

1. 模块独有的插件

这种情况比较少见，如果确实存在，那就先写在模块的pom.xml里面吧，项目组可以另行安排时间进行项目内的pom文件优化。

## 常用命令

对用户而言，主要是clean、compile、install和deploy。

1. 如果进入到指定模块下面执行这些命令，则只会对模块内的代码操作。

1. 如果在项目根目录执行这些命令，则会自动计算反应堆的依赖顺序依次执行。

（注：反应堆（Reactor）是指所有模块组成的构建结构，反应堆包含项目本身和所有模块）

1. 如果用户希望仅仅构建反应堆里面某些模块，那么就需要用到裁剪反应堆：

* -am （also make）：同时构建所列模块的依赖模块。
* -amd （also make dependents）：同时构建依赖于所列模块的模块。
* -pl （project list）：构建指定的模块，模块间用逗号分隔。
* -rf （resume from）：从指定的模块回复反应堆。

# 版本管理

## 概念

版本管理是指项目整体版本的演变过程管理，如从hotent-bpmx5-1.1.0 到 hotent-bpmx5-1.2.0 再到 hotent-bpmx5-1.3.0。版本控制是指借助版本控制工具（如SVN）追踪代码的每一个变更。

## 版本号的设计

此命名规范为公共规范，适用于所有项目。

软件版本号组成的基本规范：

[项目名称]-[主版本].[次版本].[增量]-[里程碑]

如：x5-base-root-1.0.1-release

（里程碑为可选）

（这里我将x5作为了一个新的产品，独立计算版本号，而不是bpmx3的一个延伸）

下面就每部分分别说明：

### 主版本

表示项目的重大架构变更。

如1.0、2.0、3.0等。

### 次版本

表示较大范围的功能增加和变化。可以通过次版本来规划项目的开发周期，将该主版本计划开发功能分为几期完成。

如1.0、1.1、1.2等。

**注意**：一旦一个次版本发布了正式版本（release），那么次版本将会升级。

比如xxxx-1.5.6-release是一个release版本，当升级时，将升级到xxxx-1.6.0，快照从1.6.0开始计算。

### 增量版本

增量版本主要在两种情况下使用：

1. 项目日常开发迭代。
2. 分支的bug修复。

下面分别进行详述：

#### 开发迭代

##### 说明

增量版本是用于对次版本的进一步细分。

如某个次版本需要开发20个功能，计划4周完成，那么可以分为4个增量版本，每周发布一个增量版本。

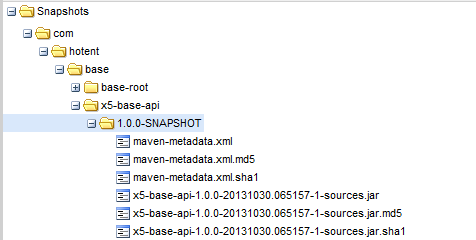
**每周迭代**

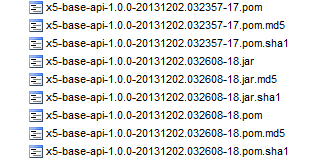
增量版本建议每周递增一次，由版本管理员通过Maven插件和自行开发的脚本，将涉及的项目的版本号批量修改（包括被依赖地方的版本号）。

##### 对日常开发影响

除了每周迭代，日常对于功能开发、bug修复等是不影响版本号的变更的，由开发人员在完成后，执行deploy命令将代码提交到远程仓库中即可，此时会在Maven快照仓库的该项目的快照版本中会生成一个新的快照（但不改变快照版本号）。

如下图：





x5-base-api-1.0.0 是当前的版本，在该1.0.0-SNAPSHOT快照版本下有若干不同时间戳的版本号。

#### Bug修复

##### 说明

在产品发布了里程碑版本，这些里程碑版本会创建相应的分支（svn的branch）。

当这些分支出现bug时，如果需要修复，则根据Tag创建分支（branch），然后将分支检出，进行修复，修复完成后，将对分支打tag，而这个tag将会在branch的基础上累加增量版本号。

如xxxx-1.5.6-release分支，修复后，tag的版本号将是xxxx-1.5.7

（由于1.5已经发布，这时trunk的版本号已经升级到了1.6，所以增量版本号是不会冲突的）

【注】在发布relase版本之前，由于次版本号还未升级，所以即使alpha、beta版，这些版本发现bug，也是依然在该次版本号的基础上递增增量版本。

##### 对快速修复的影响

* **流程说明**：

1. 客服接到客户反馈的Bug。
2. 和客户沟通，分析该Bug，定位该Bug所在的项目和模块。
3. 找到相关项目（1～N个）的Tag，生成分支Branch。（由版本管理员协助）。
4. 检出这些分支。
5. 修复Bug，提交内测，反复操作。
6. 完成修复，创建Tag（增加增量版本）（由版本管理员协助）
7. 利用svn工具生成版本变更patch，或通过Maven生成的jar、war等文件。
8. 返回给客户。

* **难点说明**

1. 客户购买产品的项目不一样。

如单纯购买工作流、表单，或者购买整个产品所有项目。有些项目只有java代码，有些则有页面，涉及jsp、js或css等技术，根据不同的项目有不同的处理方式。

1. 定位Bug归属的项目。

项目之间存在依赖，在某个项目中发现的Bug，可能是项目本身的，也可能是它所依赖的项目的，也可能涉及多个项目。定位Bug的归属是难点。

1. 多个项目的修复

需要联系相应的项目的开发人员进行修复。

1. 多个项目的版本管理

修复完成后，对于各个项目的branch需要生成相应的增量版本（tag）

（由版本管理员负责）

1. 生成更新Patch

由版本管理员对各个修改过的项目的tag进行比对，生成更新patch。

（最好写成脚本形式，以减少错误，提高效率）

### 里程碑版本

基于增量版本的基础上，增加特别的标识，对该增量版本的重要声明。

如1.1.3-beta，表示该1.1.3版本是一个对外测试版（公测）。

1.1.4-release，表示该版本可用于对外发布。

里程碑版本可以发布多次，通过增加数字后缀处理。如bpmx4.1.4-release-2

里程碑版本可以对外销售，给客户使用。

当次版本出现release版本后，即可以进入下一个次版本的开发工作。

里程碑级别：alpha < beta < release。

分别代表内测、公测和发布。

## 快照版本

一般团队在项目开发的过程中会使用快照（SNAPSHOT）版本。快照版本就是不稳定的，它定位的构件随时可能发生变化。所以当项目需要对外发布时，团队就需要提供稳定的发布版。发布版的下一个就是新的快照版。就这样不断的更新。

版本管理关心的问题之一就是这种快照版和发布版之间的转换。

首先了解主干（trunk）、标签（tag）和分支（branch）的概念。

主干是项目开发代码的主体，从项目开始知道当前都处于活动的状态。

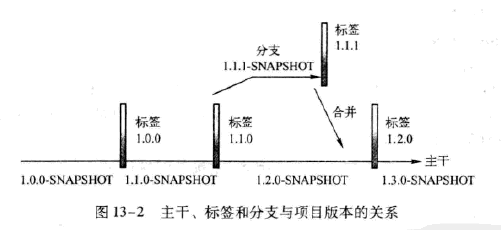
分支是从主干的某个点分离出来的代码拷贝通常进行重大的Bug的修复或做一些实验性质的开发。

标签用来标识主干或分支的某个点的状态。

下图是Maven的版本管理，在两个次版本之间是快照版本。模块和模块之间的依赖均是依赖快照号。

当某个版本正式发布后，如果需要修改（主要是修复bug），需要做的操作包括：

1. 根据标签（tag）1.1.0生成分支（branch）；
2. 将分支branch检出。
3. 配置parent模块的pom.xml文件，将scm的路径更改为该branch路径。
4. 随着分支的修改，增量版本就起到作用了。修改了bug后，增量版本发布。
5. 然后从增量版本中将修改的代码合并回开发主干。



## Svn和版本管理

### 独立的svn目录

每个项目使用独立的svn路径，这样便可以按需开放源码的读写权限了。

### 项目的版本号

* **主版本号一致**：一个产品下多个项目的主版本号一致。
* **同一项目下多个模块**
  + 次版本保持一致
  + 增量版本可以不一样。
  + 里程碑版本，项目统一，由父模块维护。
* **项目之间**
  + **次版本号独立**：每个项目采用的独立的次版本号。

原因是基础项目（如x5-base-root）更新的频率远远比业务项目低，如果次版本号一致，会相互干扰开发管理。

* + **增量版本独立**

各项目的增量版本独立维护。

* + **里程碑版本：存在等级依赖限制**。

如果打包的产品的最外层项目是beta版，那么它依赖的其它模块可以是beta版也可以是release版，但是如果最外层是release版，那么它依赖的其它模块则必须全部是release版本了。

简单而言，就是按依赖级别，层次高的模块的里程碑级别不能低于层次低的模块。

* **模块之间**

同一项目不同模块采用同样的版本号。

理由是它们从属以同一svn项目的管辖，版本号不一致无从管理。

### 项目的svn路径设计

* 一个产品一套根目录

如：[产品名]\trunk

[产品名]\branches

[产品名]\tags

* trunk：一个项目一个独立目录

[产品名]\trunk\[项目1]

[产品名]\trunk\[项目2]

* branches： 按次版本进一步划分

[产品名]\branches\[项目1]\1.1.x\[项目名]-1.1.1-beta

\[项目名]-1.1.2-release

* tags：按次版本进一步划分

[产品名]\tags\[项目1]\1.1.x\[项目名]-1.1.1

\1.1.x\[项目名]-1.1.2

\1.1.x\[项目名]-1.1.3

### 源码独立打包

子项目的源码是独立的，不存在同步其它项目源码到本项目中的情况。

销售软件产品时，也是各个子项目的源码独立提供，而不是合并到一个项目内。

# Maven的版本管理

主要是使用Maven Release Plugin来完成版本的构建，必要时会增加一些自行开发的maven插件或者脚本来完成一些辅助功能。

## Maven Release Plugin

### 插件目标介绍

插件首页：

<http://maven.apache.org/maven-release/maven-release-plugin/index.html>

1. **release:prepare**

准备版本发布，步骤包括：

1. 检查项目是否有未提交的代码；
2. 检查项目是否有快照版本依赖；
3. 根据用户的输入将快照版本升级为发布版；
4. 将POM中的SCM信息更新为标签地址；
5. 基于修改后的POM执行Maven构建。
6. 提交POM更改；
7. 将代码从发布版升级为新的快照版。
8. 提交POM变更。
9. **release:rollback**

回退release:prepare所执行的操作。

1. **release:branch**

为当前项目创建一个分支，并升级当前项目的版本（可选）。

执行的步骤和release:prepare一致，区别是插件branch而不是tag。

1. **release:perform**

执行版本发布。

1. 签出release:prepare生成的标签中的源代码；
2. 在此基础上执行mvn deploy 命令打包并 部署构件至仓库。

### 插件和版本管理

* **release:prepare**

用于生成项目的tags，并升级trunk中该项目的增量版本号。

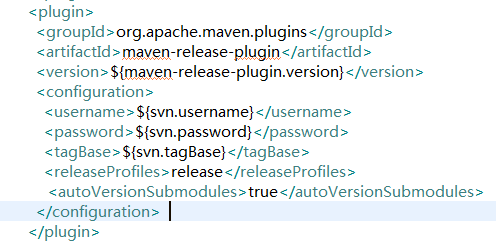
* **release:branch**

用于生成项目的branch，并升级trunk中该项目的增量版本号（可通过参数控制不升级）。

## Maven Release Plugin需要的配置

### release插件

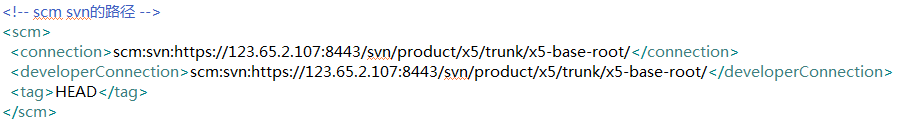
在各个项目的xxx-parent模块的pom.xml中引入：



其中各个设置通过参数进行管理，其中tagBase示例：



### scm



connection, developerConnection: 都是连接字符串，其中后者是具有write权限的scm连接，常用的scm工具包括cvs与subversion，还有其他scms，url的字符串格式是：scm:[provider]:[provider\_specific]，例如cvs的是scm:cvs:pserver:127.0.0.1:/cvs/root:my-project

tag：说明project所在的目录tag，默认是HEAD，表示根目录

### 待思考的问题

#### 快照版本号的升级

* **项目内**

通过Maven Release插件可以完成项目内根目录pom.xml版本号的升级（包括在子模块中引入了它的版本号），但插件无法同时更新这些子模块本身的版本号。

按4.4.2的设计，所有子模块均应该和项目采用同样的版本号，可以考虑通过编写一个脚本或插件，同时更新子模块的版本号。

* **项目之间**

另外，这些模块会在其它的项目的模块中被依赖，那么当这些模块版本升级了，其它项目的pom.xml也需要调整依赖的版本号。

但由于各个项目的开发进度不一致，所以有时依赖的模块升级了，但是由于开发的原因，某个项目不一定立刻进行依赖更新，即依然依赖旧的版本。

所以我觉得对于其它项目依赖的版本号的更新，通过邮件通知和日常项目管理人工沟通，由相应项目负责人决定更新时机。