配置管理

源码管理

王金涛

2024 年 1 月 16 日

主要内容

重要概念

配置管理 通过技术或行政手段对软件产品及其开发过程和生命周期进行控制、规范的一系列措施。

配置项 指纳入配置管理范畴的所有项目 (item) 。

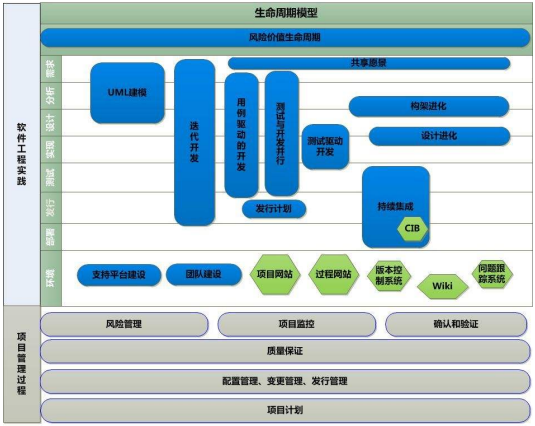
开发库 开发人员的工作空间，开发人员在配置项写入时，必须填写相关信息以标识配置项，配置项支持 Checkout 和 Checkin 能力。

受控库 保存已被批准的配置项（包括基线），由配置管理员管理与维护。信息分两类：受控基线和受控配置项。

产品库 作为最终产品存放在产品库，等待交付客户使用，出入库要严格办理手续。

软件管理中的位置

内容概览



开发运维一体化中的位置

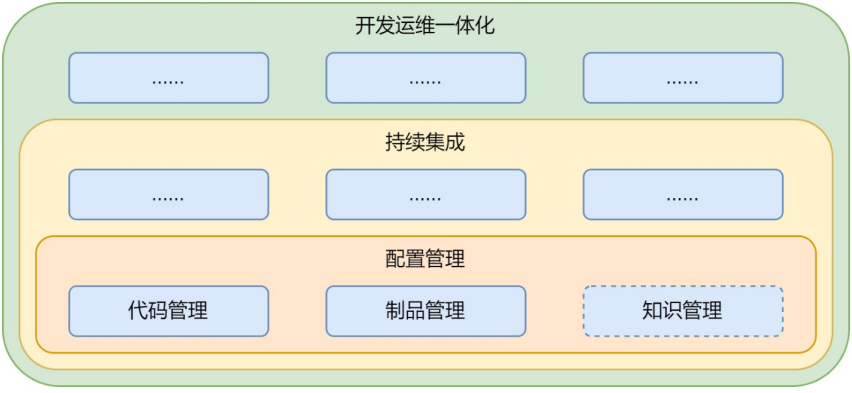


图: 配置管理在 devops 中的位置

CMMI3 对配置管理的要求

- 执行版本管理
- 识别配置项并建立标识
- 识别基线并建立标识
- 控制变更
- 管理变更
- 记录变更
- 执行审计

问题

VCS 出现之前，程序员是如何协作的?

描述

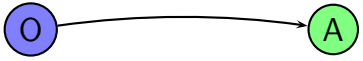


图: 版本差异

版本差异

1. 记状态 O,A; 则向量 $\overrightarrow{OA}$ 是状态 O 到 A 的版本差异。

描述

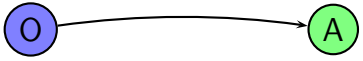


图: 版本差异

版本差异

1. 记状态 O,A; 则向量 $\overrightarrow{OA}$ 是状态 O 到 A 的版本差异。
2. 已知 O,A，求 $\overrightarrow{OA}$ 的运算叫 $diff$ ，记 $\overrightarrow{OA} = diff(O, A)$。

描述

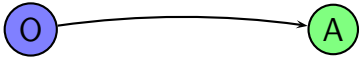


图: 版本差异

版本差异

1. 记状态 O,A; 则向量 $\overrightarrow{OA}$ 是状态 O 到 A 的版本差异。
2. 已知 O,A，求 $\overrightarrow{OA}$ 的运算叫 $diff$，记 $\overrightarrow{OA} = diff(O, A)$。
3. 已知 O, $\overrightarrow{OA}$，求 A 的运算叫 $patch$，记 $A = patch(O, \overrightarrow{OA})$。

描述

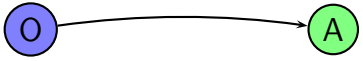


图: 版本差异

版本差异

1. 记状态 O,A; 则向量 $\overrightarrow{OA}$ 是状态 O 到 A 的版本差异。
2. 已知 O,A，求 $\overrightarrow{OA}$ 的运算叫 $diff$ ，记 $\overrightarrow{OA} = diff(O, A)$。
3. 已知 O, $\overrightarrow{OA}$ ，求 A 的运算叫 $patch$，记 $A = patch(O, \overrightarrow{OA})$。
4. 状态 O,A 的本质也是向量。$\overrightarrow{OA}$ 中不记未改变的部分 (sector)。

状态 O,A

```
1 class HelloWorld{
2     public static void main(String[] args){
3         System.out.println("hello world!!");
4     }
5 }
```

状态 O

```
1 class HelloWorld{
2     public static void main(String[] args){
3         System.out.println("==========");
4         System.out.println("hello world!!");
5     }
```

unified 格式差异

```
diff -r -U 1 0/HelloWorld.java a/HelloWorld.java
--- 0/HelloWorld.java    2019-08-26 14:15:08.070270000 +0800
+++ a/HelloWorld.java    2019-08-26 14:20:10.809870900 +0800
@@ -2,2 +2,3 @@
     public static void main(String[] args){
+        System.out.println("==========");
         System.out.println("hello world!!");
```

差异 $\overrightarrow{OA}$

```
diff -r -U 1 a/HelloWorld.java 0/HelloWorld.java
--- a/HelloWorld.java    2019-08-26 14:20:10.809870900 +0800
+++ 0/HelloWorld.java    2019-08-26 14:15:08.070270000 +0800
@@ -2,3 +2,2 @@
     public static void main(String[] args){
-        System.out.println("==========");
         System.out.println("hello world!!");
```

差异 $\overrightarrow{AO}$

context 格式差异

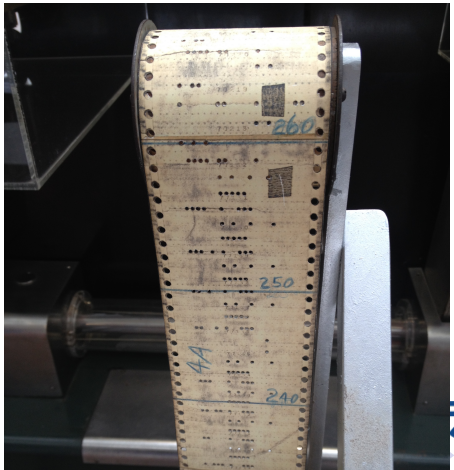
```
diff -r -C 1 0/HelloWorld.java a/HelloWorld.java
*** 0/HelloWorld.java    2019-08-26 14:15:08.070270000 +0800
--- a/HelloWorld.java    2019-08-26 14:20:10.809870900 +0800
***************
*** 2,3 ****
--- 2,4 ----
      public static void main(String[] args){
+         System.out.println("==========");
          System.out.println("hello world!!");
```

```
diff -r -C 1 a/HelloWorld.java 0/HelloWorld.java
*** a/HelloWorld.java    2019-08-26 14:20:10.809870900 +0800
--- 0/HelloWorld.java    2019-08-26 14:15:08.070270000 +0800
***************
*** 2,4 ****
      public static void main(String[] args){
-         System.out.println("==========");
          System.out.println("hello world!!");
--- 2,3 ----
```

差异 $\vec{a}$(上图) $\vec{a}$(下图)

补丁

右图中黑色的东西就是最早的 patch (补丁)。
该图为马克一号计算机 (1944 年，最早的哈弗架构)。



应用补丁

在 *O* 的副本上应用补丁。

```
patch -p1 --verbose -i ../patch
```

除了前述差异格式外，还有其他的差异描述格式，其中有的可以记录二进制文件的不同，也可以用于 patch。

三路差异

三路差异

是指当一个文件在两个独立分支中被修改后如何合并这些修改成为一个文件的差异分析问题。解决该问题的方法一般称作三路合并。

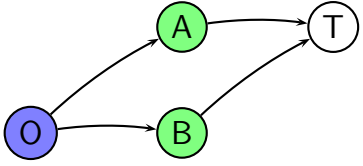


图: 三路差异/三路合并

冲突

三路合并导致冲突的原因

1. 求向量 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$, 并视为两个向量集。

冲突

三路合并导致冲突的原因

1. 求向量 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$, 并视为两个向量集。
2. 检查 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 是否有共同起点的向量 (节/sector)。

冲突

三路合并导致冲突的原因

1. 求向量 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$, 并视为两个向量集。
2. 检查 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 是否有共同起点的向量 (节/sector)。
3. 若没有, 则 T 是 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 共同作用于 O 的结果。

冲突

三路合并导致冲突的原因

1. 求向量 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$, 并视为两个向量集。
2. 检查 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 是否有共同起点的向量 (节/sector)。
3. 若没有, 则 T 是 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 共同作用于 O 的结果。
4. 若有, 则 $\overrightarrow{OA}$ 与 $\overrightarrow{OB}$ 共同起点的向量 (节/sector) 的集合就是冲突 (conflict)，需要手工解决。

优秀的三路合并算法就是最大的可能减少冲突发生。

diff3

```
diff3 [options] myFile originFile otherFile
```

diff3

diff3 直接使用了上述经典算法, 其中差异的求法依赖于 diff 并广泛直接或间接应用于各种 cvs(e.g. subversion)。

十字交叉问题

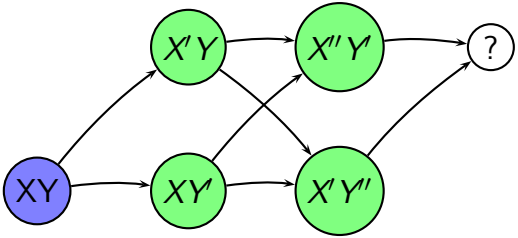


图: 十字交叉问题

十字交叉问题

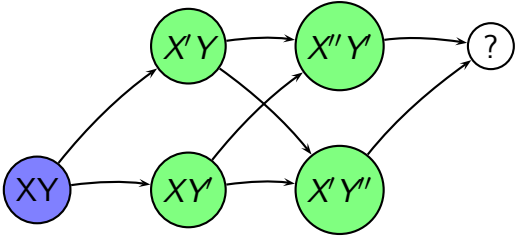


图: 十字交叉问题

说明

- 该问题出现在两个开发分支频繁 交换 特性 (feature) 的时候。
- $X''Y'$ 与 $X'Y''$ 合并时不存在唯一的最小公共祖先。
- 实践中 git 和不规范的 subversion 使用常出现上述问题。

递归三路合并

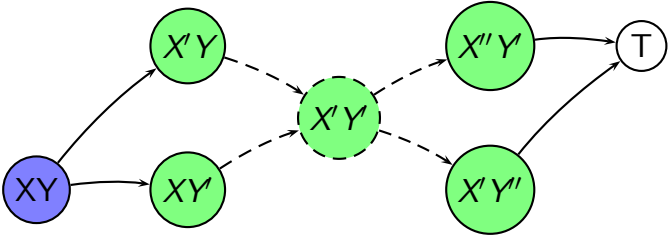


图: 递归三路合并

递归三路合并

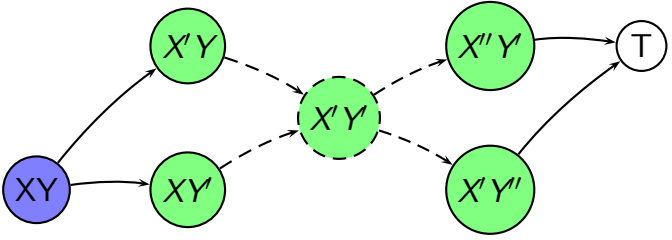


图: 递归三路合并

说明

Git 采取了递归三路合并 (Recursive three-way merge) 处理一般的三路合并问题，对没有单一最小共同祖先的一对文件递归创建虚拟祖先。

其他方法

模糊修补包算法

patch 程序可以把一个修补包用于与最初产生该包的文件不是完全相同的文件。这称作模糊修补包应用 (fuzzy patch application)。GNU arch 采用了这种方法。但模糊修补包应用是一种不太可信的办法，在上下文太少情况下可能会误用。

编织合并

编织合并 (Weave merge) 算法跟踪每行是被增加或是删除，产生结果信息。如果在一个版本中该行被删除，则结果文件就不包含该行。BitKeeper、GNU Bazaar、Codeville 采用了此方法，对三路合并出错的情形能产生正确结果。

其他方法

修补包交换

修补包交换（Patch commutation）改变修补包的应用顺序，形成一线性历史。效果上，当两个修补包产生于同一个环境，合并时，一个修补包被重写以便它可以在另一个修补包执行完毕后才使用。

Darcs 、Git (称作"rebasing") 采用了这一方法。

"patchutils" package 中的 Unix 程序 flipdiff 实现了修补包交换.

基本方案

- 文件共享

基本方案

- 文件共享
- 锁定-修改-解锁

基本方案

- 文件共享
- 锁定-修改-解锁
- 复制-修改-合并

基本方案

- 文件共享
- 锁定-修改-解锁
- 复制-修改-合并

参考书目

- 《git 权威指南》第一章
- 《svn book》第一章

只管理源码

原因

代码版本控制系统只能优雅的处理纯文本文件的版本问题。

不管理

- 编译结果与依赖 (jar,so)
- 文档（docx,pdf)
- 临时文件与环境相关文件

替代方案

- 产出物与依赖管理
- 专用协作系统、文档纯文本化或降低协作需求
- 不管理没有任何协作需求的文件

提交时先下后上

说明

代码提交前，应先将远端的代码同步至本地，解决掉冲突后再提交至远端。

规则与技巧

- 后提交者有责任处理与先提交者的代码冲突（提交竞赛）
- 管理者在工作分派时有责任减小冲突发生的概率（模块划分与工作分派）
- 变化越小，冲突越少
- 不提交无意义的代码变更（统一格式化标准，先格式化，再提交）

演变关系是核心价值

说明

版本管理系统的核心价值是保存了配置项的演变关系，任何破坏这些演变关系的行为都是不可取的。($vcs \neq ftp$)

禁止事项

- 将工作区的内容删除，再复制一套进来提交。
- 将工作区的文件重命名，提交。
- 清空一个仓库，再导入一套进去。
- 仓库迁移时只迁移最新状态。

核心概念

- 主支 (trunk/master)
- 分支 (branches)
- 标记 (里程碑/tags)

适用于产品的经典模型

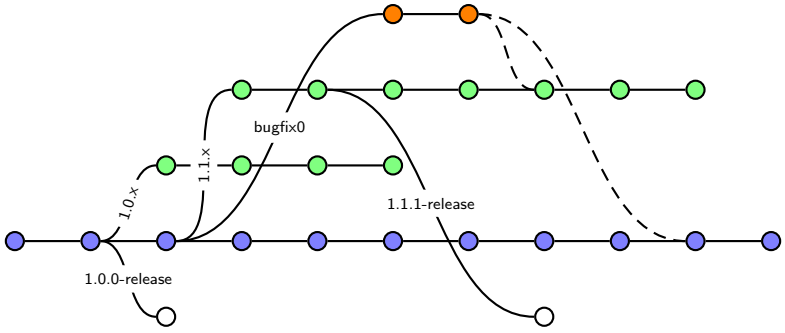


图: 语义化版本号模型

版本号的格式为 x.y.z

适用于产品的经典模型

特性

- 主支用以开发最新的 y 版本

适用于产品的经典模型

特性

- 主支用以开发最新的 y 版本
- 标记用以记录发布行为，如果 z 为 0 则来自主支，否则来自版本分支

适用于产品的经典模型

特性

- 主支用以开发最新的 y 版本
- 标记用以记录发布行为，如果 z 为 0 则来自主支，否则来自版本分支
- 分支的主要作用是维持 y 版本号的生命周期

适用于产品的经典模型

特性

- 主支用以开发最新的 y 版本
- 标记用以记录发布行为，如果 z 为 0 则来自主支，否则来自版本分支
- 分支的主要作用是维持 y 版本号的生命周期
- 对于跨越多个 y 版本的 bug（且需要在多个版本上修复的），则需要 bugfix 分支

局限性

- 路线图管理 (版本规划、发布管理，不灵活)

适用于缺乏发布版本管理的模型及变体

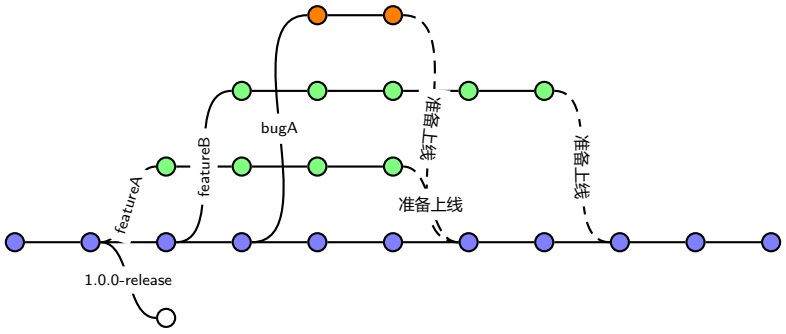


图: 无发布计划的模型

适用于缺乏发布版本管理的模型及变体

场景

需求频繁而琐碎，可以确认需求开发的开始日期，不能确认需求的上线日期（无路线图）。

- 测试环境仅能包含单一的开发中需求，难以集成测试。

适用于缺乏发布版本管理的模型及变体

场景

需求频繁而琐碎，可以确认需求开发的开始日期，不能确认需求的上线日期（无路线图）。

- 测试环境仅能包含单一的开发中需求，难以集成测试。
- 当分支的存活时间过长时，易发生合并错误。

- 主支记录了线上的最新版本或允许上线的最新版本。

适用于缺乏发布版本管理的模型及变体

场景

需求频繁而琐碎，可以确认需求开发的开始日期，不能确认需求的上线日期（无路线图）。

- 测试环境仅能包含单一的开发中需求，难以集成测试。
- 当分支的存活时间过长时，易发生合并错误。

- 主支记录了线上的最新版本或允许上线的最新版本。
- 分支对应各需求的开发，当可以上线时合并到主支。

适用于缺乏发布版本管理的模型及变体

场景

需求频繁而琐碎，可以确认需求开发的开始日期，不能确认需求的上线日期（无路线图）。

- 测试环境仅能包含单一的开发中需求，难以集成测试。
- 当分支的存活时间过长时，易发生合并错误。

- 主支记录了线上的最新版本或允许上线的最新版本。
- 分支对应各需求的开发，当可以上线时合并到主支。
- 标记记录了每次上线的里程碑，只来自主支。

适用于缺乏发布版本管理的模型及变体

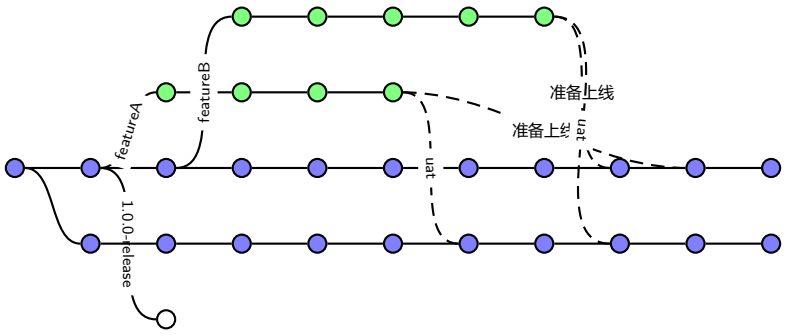


图: 适应集成 uat 的模型

适用于缺乏发布版本管理的模型及变体

场景

同上一种模型，但需求的开发和测试周期重叠严重。

- 解决了上一个模式集中进行 uat 测试的问题。

适用于缺乏发布版本管理的模型及变体

场景

同上一种模型，但需求的开发和测试周期重叠严重。

- 解决了上一个模式集中进行 uat 测试的问题。
- 增加了基线维护的难度

- 主支记录了线上的最新版本或允许上线的最新版本。

适用于缺乏发布版本管理的模型及变体

场景

同上一种模型，但需求的开发和测试周期重叠严重。

- 解决了上一个模式集中进行 uat 测试的问题。
- 增加了基线维护的难度

- 主支记录了线上的最新版本或允许上线的最新版本。
- uat 分支记录了所有可以进行 uat 测试的需求状态。

适用于缺乏发布版本管理的模型及变体

场景

同上一种模型，但需求的开发和测试周期重叠严重。

- 解决了上一个模式集中进行 uat 测试的问题。
- 增加了基线维护的难度

- 主支记录了线上的最新版本或允许上线的最新版本。
- uat 分支记录了所有可以进行 uat 测试的需求状态。
- 分支对应各需求的开发，当可以 uat 测试时合并至 uat 分支，当可以上线时合并到主支。

适用于缺乏发布版本管理的模型及变体

场景

同上一种模型，但需求的开发和测试周期重叠严重。

- 解决了上一个模式集中进行 uat 测试的问题。
- 增加了基线维护的难度

- 主支记录了线上的最新版本或允许上线的最新版本。
- uat 分支记录了所有可以进行 uat 测试的需求状态。
- 分支对应各需求的开发，当可以 uat 测试时合并至 uat 分支，当可以上线时合并到主支。
- 标记记录了每次上线的里程碑，只来自主支。

Git 的模型

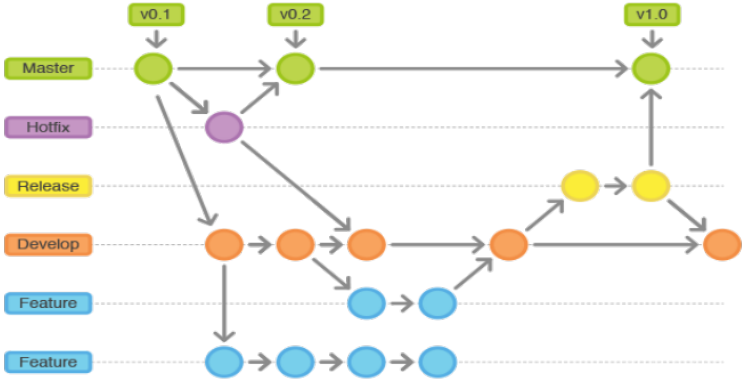


图: git flow 模型

版本控制系统使用的要求

- 尽量早的提交源码。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。
- 重视路线图管理。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。
- 重视路线图管理。
- 合理使用分支。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。
- 重视路线图管理。
- 合理使用分支。
- 警惕沉睡分支。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。
- 重视路线图管理。
- 合理使用分支。
- 警惕沉睡分支。
- 模型的选择应与项目管理相适应。

版本控制系统使用的要求

- 尽量早的提交源码。
- 尽量只管理源码。
- 系统上线一定要有标记。
- 重视路线图管理。
- 合理使用分支。
- 警惕沉睡分支。
- 模型的选择应与项目管理相适应。
- 格式统一、先格式化再提交。

思考

为什么 Git 选择使用“递归三路合并”作为其默认的合并方式?

作业

小作文

介绍一下你负责的某个项目的代码版本管理模型，阐述使用这个模型的原因，论述其优缺点以及与该项目开发过程中的契合性。

推荐绘图工具 drawio。