# GIT 使用体会

v3

Yubao.Liu@gmail.com

2008-10-27

# Contents

1 前言 ..... 1
2 版本控制的基本概念 ..... 2
3 GIT 里的术语定义 ..... 5
4 版本库 ..... 10
5 对象记法 ..... 13
6 合并之 fast forward ..... 14
7 混乱之源——index ..... 17
8 工作流程 ..... 20
9 常用命令简介 ..... 22
10 GIT 的模块功能 ..... 28

# Figures


1 一个简单的版本演化图 4
2 Fast forward 14
3 Fast forward 的结果 14
4 --first-parent 的作用 16
5 --first-parent 的失效 16

# 1 前言

本文面向有一定版本控制经验的人群，对 GIT[1] 有基本了解，如果有 Subversion[2]、Mercurial[3] 等使用经验更好。文中从工具的原理和设计乃至实现出发，讲述了 GIT的用法，并以个人愚见展示了一些 GIT 不完美的方面。文章名字本来想戏谑的起名《深入浅出 GIT》，意指原理讲的较深(但愿)，而具体单个命令用法讲的很浅，因为后者有手册可查，但后来还是觉得有辱没类似名字命名的著作且混乱成语用法的嫌疑，遂作罢。

以前初学 GIT 不久写过一篇《GIT 之五分钟教程》，当时以为自己对 GIT 了解的比较好，但后来在工作中真正的用上 GIT 后，才发觉还是学习的很肤浅，遇到问题经常觉得无所适从。GIT 秉承了 UNIX 的优良传统——小工具堆叠，“条条道路通罗马”，可惜我常常不知道哪条路是最好走的。经过近一年的实战使用，我仍觉得 GIT 这玩意还是没能运用自如，但总算有些心得体会，记录成文，希望对 GIT 用户有所帮助。

GIT 的命令行界面我觉得至今还是不能让人满意，我比较喜欢 Subversion、Mercurial 的命令行界面：一致、简洁。GIT 属于那种每个人都想拼命往里面塞功能，每个人都想让 GIT 具备自己喜欢特性的工具，结果就导致 GIT 如同 Shell 编程一般，“一切皆有可能”，虽然是颇有恶趣味，但也常让人厌烦。这绝不会只是我个人的感受，遍观繁多的 GIT 包装工具就知道了，而且有一些 GIT 包装工具的做法被 GIT 吸收，可以说 GIT 正在成为一个怪物，一个让人又爱又恨的怪物[4]，它的运行速度非常快——真的是非常快，它的设计思想非常简洁有效。

如果你要纳入版本控制的文件树规模不大[5]，如果你不关心分布式开发，那么对于版本控制工具，我向你推荐Subversion，集中式版本控制工具的佼佼者，有着

---

[1] http://git.or.cz

[2] http://subversion.tigris.org/

[3] http://www.selenic.com/mercurial

[4] 我身边这种怪物还有 Perl、VIM。

[5] 大于 500 MB 你就要慎重考虑了，Subversion 没有文件复制自动探测，用户如果粗心的用 svn add 来替代 svn copy，那么会导致版本库迅速膨胀，而且 Subversion 的工作拷贝实现原理比较低效，类似操作比起 GIT 慢得多，著名的开源项目 FreeBSD 从 CVS 转向 Subversion，我是觉得很可惜的。

非常友好的命令行和图形使用界面，否则，如果你不是必须使用 GIT，那么我推荐Mercurial，目前分布式版本控制工具里唯一可以跟 GIT 抗衡的选手[6]，Sun的OpenJDK、OpenSolaris 这样的大规模项目都是使用Mercurial 管理的。这两者都已经被用在大量开源项目里，而且可移植性非常好。

啰嗦了这么多，该说感谢致辞了，首先要感谢 NewSMTH BBS 的 donated 和 garfield。大概是一年前，donated 非常耐心的给我介绍了 ConTEXt，但那时能方便使用新字体的 ConTEXt MkIV 还没有成熟，我也就没提起兴趣继续学习。不久前我阅读了 garfield 的《ConTEXt 学习笔记》[7]，才知道ConTEXt MkIV 居然已堪实用，蒙这篇笔记的引导才开始慢慢学习 ConTEXt，本文就是我正式用 ConTEXt 排版的第一篇文章[8]。其次我要感谢我的同事，我靦颜作为他们的 GIT 传教者以及版本库维护者，正是由于跟他们的合作促使我认真思索应该如何使用 GIT— — 从“怎么用”到“怎么用好”[9]是个很大的转变。

## 2 版本控制的基本概念

版本控制(version control)的目的就是保存一个文件树在不同时刻的状态，其基本概念是相当直白的：

**文件(file)**

普通的包含数据的文件。

用途：保存数据。

**目录(directory)**

文件名、目录名的集合，目录的这种递归包含结构构成了整个文件树。

---

[6] 虽然我欣赏 Canonical 的Ubuntu Linux 发行版，但是对它家的 Bazaar 并没有多大好感，从几次使用印象来看这东西效率太太太低了，表面友好的使用界面掩盖不住其繁复的设计。

[7] http://code.google.com/p/ctxnotes/

[8] 就这版面设计还很“憨厚”，另外 ConTEXt MkIV 中文排版还有些问题，有待改进。

[9] 希望以后能明白“怎么用最好”:-)

用途: 保存文件路径清单。

**版本(version, revision)**

记录某一时刻整个文件树的状态, 并且会包含一些额外信息, 比如时间、作者、备注、从哪个版本演化而来的。版本的记法有很多种, 比如用递增的数字标记(Subversion), 用点分数字如1.2.3 标记(CVS, RCS), 用版本信息的摘要值标记(GIT, Mercurial)等等。

用途: 记录文件树状态变更整个过程, 文件树状态演化形成一个单向无循环图(DAG)。

**标签(tag, label)**

用一个容易记忆的名字标记某个版本。

用途: 帮助记忆。

标签可以分成两种: 静态标签(static tag) 和浮动标签 (float tag), 在各种版本控制工具中, “标签” 一词往往指静态标签, 而将浮动标签称为“分支头(branch head)”, 或简称为“ 头(head)”, 分支头的名字(也即浮动标签的名字)命名了一个分支。需要注意的是版本控制工具中往往用大写的 HEAD 指代**当前**分支对应的头部。

所谓静态标签就是指必须显式修改标签内容(比如版本控制工具的 tag 命令), 才能让其标记另外一个版本, 浮动标签就是指从它指代的版本派生新版本时(比如版本控制工具的 commit 命令), 标签自动被修改, 以指向新版本。

而关于“ 分支(branch)”的定义, 则是各有微妙分歧, 下图中小写字母表示版本, 也就是文件树状态以及关于这个状态的备注信息, 连线表示其版本变迁关系, 时间轴从左向右:

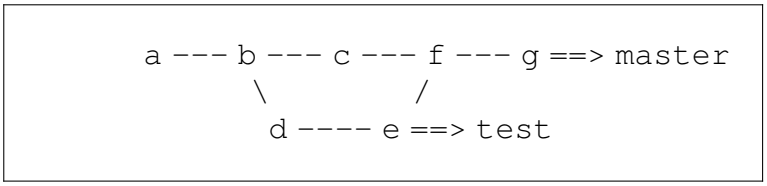


**Figure 1** 一个简单的版本演化图

这个版本图中, 文件树从起始的 a 版本, 用 master 动态标签来标记, 由于修改意图的分歧, 在 b 版本之后分成两个修改方向, b -> c 方向仍然用 master 标记, b->d 用 test 标记, 在 c 和 e 之后, 两个修改方向取长补短又合并成一个版本 f, 之后经过修改后又达到版本 g。图中最后 master 和 test 两个浮动标签分别指代 g 和 e, 它们命名了两个分支: master 分支和 test 分支。

那么“分支”指代什么呢? 比如 master 分支, 有如下几种说法:

- 从分歧点算起, master 分支指 f, g 这两个版本, 或者只算 g;
- 指 master 这个标签曾经指向的版本, 也就是 master 这个动态标签的“ 轨迹”——a, b, c, f, g;
- 指从 master 标记的版本回溯, 可以达到的版本, 也即 g, f, c, e, d, b, a (注意 f 是一次合并, 它可以回溯到 e 和 c)。

这种分支含义的分歧直接导致版本控制工具对于“ 分支历史” 定义的分歧, 比如 log 命令的输出。

由于分支头命名了分支, 所以往往将二者笼统的都称呼为分支。

关于分支还有另外一个分歧, 就是合并的对称性, 考虑图 1中的版本图, 在 e 版本的基础上合并 c(也即通常说的在 test 分支上合并分支 master), 和在 c 版本的基础上合并 e(也即通常说的在 master 分支上合并分支 test) 所带来的结果是否等价呢?

而其实“ 版本” 的定义也是有分歧的, 从上面可以看到, 标签主要是助记用的, 它的内容可以被修改, 那么“版本”记录的状态里是否要包含当时标签的状态呢?

各种术语称呼差别, 命令叫法不一, 模型定义分歧, 往往导致从一种版本控制工具切换到另一种时极为别扭, 不知所措。

## 3 GIT 里的术语定义

GIT 里用 blob, tree, commit, tag 四种存储对象来描述版本控制所需信息, 每一种存储对象都是以类型名字、内容长度、内容计算出的 SHA1 摘要值命名, 存放在 .git/objects/ 目录下的文件。

**blob**

表示第 2节中的文件, blob 只有文件数据和长度, 不保存名字、文件类型、权限等信息, 恰如其名;

**tree**

表示第 2节中的目录, 不包括像 Subversion 那样的 property，只有文件权限位（无属组和属主信息)、对象类型(blob 还是 tree)、SHA1 摘要值、文件或子目录名字。早期的 git 实现中 tree 对象保存的是整个文件树的路径名清单, 现在的实现中一个 tree 对象只保存某个目录下的一层信息;

**commit**

表示第 2节中的版本;

**tag**

表示第 2节中静态标签。

值得一提的是 GIT 自称 content tracker 而非版本控制工具。这一思想在 blob 和 tree 记录的信息上有所体现, blob 和 tree 只记录了绝对必要的**数据**(权限位虽然不必要但是在 UNIX 类系统上非常有用)，因为是 content tracker，所以 blob 和 tree 不关心作者名字、类似 Subversion 那样的额外属性, 这样两个作者声称数据内容一致(额外属性可以不一致)时, 必然可以共用 blob 和 tree 存储对象。这样做

的好处是存储模型简单, 节约空间, 有利于分布式使用, 坏处是少了 Subversion 那样方便的跟目录、文件关联的版本控制的属性功能。

content tracker 这个名字还体现在对 file 和 blob 的区分对待上: file 指 blob + file name。git 所关心的是文件内容的流向, 并不显式记录复制、移动这些信息, 这些信息是运行时根据文件内容流向动态分析出来的。这样做的好处是减轻了使用者的负担, 据个人所见, 很多 Subversion 用户在应该用 copy 时却用 add, 导致版本库迅速膨胀, 而且 svn log 里看不出来复制、移动信息。这样做的坏处自然是动态分析需要时间, 单次变更集(change set) 太大时效果不行, 这就寄希望于更好的探测算法了——起码我们将来可以很容易补救。

file 和 blob 的区别体现在文件历史上, 在 git 里, 一个文件的历史是这个文件路径曾经指代的文件内容变化, 也就是一个路径所牵涉到的历史, 而在 Subversion 里一个文件历史指的是这个文件在版本变迁中内容的改变、路径的改变历史, 因此如下操作在 git log 和 svn log 结果迥然不同:

```
git/svn add a && commit && git/svn mv a b && commit
git log -- a 会显示两次提交(一次新增, 一次删除);
git log -- b 会显示一次提交;
svn log a 会显示 a 没有纳入版本控制;
svn log b 会显示两次提交。
```

上面两个命令虽然形式类似, 但语义是相当不同的, svn 是在当前版本的文件树中找文件, 找到后回溯它的历史, 包括内容变化和名字变化, 而 git 是从当前版本回溯历史, 过滤出修改中涉及指定路径的修改(svn 和 git 各有选项能模拟彼此的效果)。

关于 file 和 blob 的区别以及 GIT 不显式记录移动、复制操作的设计, 早期在 GIT 的邮件列表上引起极大的争议, 为此爆发了 GIT 之父 Linus Torvalds 惊天地

泣鬼神前无古人后无来者先天下之忧而忧后天下之乐而乐的“ I'm right”宣言[10], 以及现任 GIT 维护者 Junio C Hamano 同样相当精彩的“read it now”警言[11]。

虽然 GIT 是 stupid content tracker，可实际上我们是拿它当 version control system 来使。GIT 包含两种命令, 底层命令和高层命令, 底层命令实现了 content tracker，高层命令封装底层命令模拟出了 version control system—— 其实语义跟通常的版本控制系统还是有差别的, 这往往让人迷惑。

在 GIT 中还有两种引用(reference, 简称 ref)类型: head 和 tag， head 就是第一节中的动态标签, tag 也表示第一节中的静态标签, 只是其存储方式比 tag 对象经济许多, 被称为轻量级 tag。

引用存放在 .git/refs/heads 和 .git/refs/tags 下(引用可以被打包集中存放在 .git/info/refs 或 .git/packed-refs 文件中)。

[ XXX: 术语跟 CVS、Subversion 很不一样, 可怜我等呆瓜脑袋]

更混乱的是还有一个 symbolic ref，保存在 .git/ 目录下的符号链接或者内容为 refs: refs/.... 的普通文件, 比如 .git/HEAD 内容一开始是 refs: refs/heads/master，表示当前 head 是 master。

需要注意的是在命令行引用分支时, 如果 symbolic ref 跟 ref 重名, 那么引用后者时要用 refs/heads/xxx 或者 heads/xxx 这样的名字。

[ XXX: 够混淆]

下面从实例看看它们各自的存储方式。

---

[10] http://thread.gmane.org/gmane.comp.version-control.git/27/focus=217

[11] http://thread.gmane.org/gmane.comp.version-control.bazaar-ng.general/18006/focus=18494

```
$ mkdir t && cd t
$ git init      # 初始化版本库, 默认放在工作目录里
$ echo hello > a
$ git add a
$ git commit -m "first"
$ git tag v0
$ ls .git/objects/??/*
.git/objects/09/76950c1fdbcb52435a433913017bf044b3a58f
.git/objects/51/ca0ad1685f36558c01ec400350d988f44176bd
.git/objects/ce/013625030ba8dba906f756967f9e9ca394464a
$ git cat-file -t 097695
tree
$ git cat-file -t 51ca0a
commit
$ git cat-file -t ce0136
blob
$ git cat-file -p 09769a
100644 blob ce013625030ba8dba906f756967f9e9ca394464a   a
$ git cat-file -p 51ca0a
tree 0976950c1fdbcb52435a433913017bf044b3a58f
author Liu Yubao <yubao.liu@gmail.com> 1224781865 +0800
committer Liu Yubao <yubao.liu@gmail.com> 1224781865 +0800

first
$ git cat-file -p ce0136
hello
$ find .git/refs/ -type f | while read f ; do echo $f; cat $f; done
.git/refs/tags/v0
51ca0ad1685f36558c01ec400350d988f44176bd
.git/refs/heads/master
51ca0ad1685f36558c01ec400350d988f44176bd
```

从上面可以看到: 我们创建了一个 blob (git add 创建)、一个 tree 和一个 commit(git commit 创建)、一个分支 master (git commit 第一次提交时创建)和一个轻量级 tag v0 (git tag 创建)，它们的关系为:

```
master 和 v0 -> commit(51ca0a) -> tree(097695) -> blob(ce0136)
```

```
$ git tag -m "this is v1" v1
```

这会新增 .git/objects/83/50f23b202c16803a7b20d2a8e37015d674babc 和 .git/refs/tags/v1。

```
$ git cat-file -p 8350f2
object 51ca0ad1685f36558c01ec400350d988f44176bd
type commit
tag v1
tagger Liu Yubao <yubao.liu@gmail.com> Fri Oct 24 01:31:01 2008
+0800

this is v1
$ cat .git/refs/tags/v1
8350f23b202c16803a7b20d2a8e37015d674babc
```

可以看到新增了一个 tag ref v1 指向 tag object 8350f2，后者指向 commit object 51ca0a。

tag 可以指向四种 object，常用的是指向 tree 和 commit。

[ XXX: tag object 本身并没有记录入 commit 或者 tree 中，而 tag object 内容包含了名字，那么 tag ref 只是为了加速访问而存在的东西了。tag 是为了助记，那么 tag object 的存在就滑稽了，其 SHA1 摘要用户一般记不住，tag object 的内容也完全可以放入 tag ref 中，即使签名也能用 ASCII 形式的签名附在 tag ref 文件内容末尾。]

从上面的 commit 对象内容可以看出 commit 区分了 author 和 committer，这对于协同开发是很方便的。

关于分支很重要的一点就是分支其实是用 head 也就是一个浮动标签表示的，我们可以很暴力的直接修改甚至删除重建这个标签，这样就能改变一个分

支名指代的分支了, 这招可以说是被 GIT 的 pull/push/fetch/merge/rebase/reset/revert 折腾的头晕目眩之余的必杀技:

```
git push -f . test:master
git fetch -f . test:master
git branch -D master && git branch master test
git update-ref refs/heads/master test
cp .git/refs/heads/test .git/refs/heads/master
```

然后执行 git reset --hard，就能恢复 master 这个 head 让它与 test 这个 head 指向同一个版本。

## 4 版本库

版本库就是 .git 目录, 用于保存**全局**版本演化图的 **子集**和标签。git clone A B，然后 A 库和 B 库内各自修改提交了, 从全局看, A 的修改者和 B 的修改者都是在扩展一个全局的版本图, 只是两个库各自只包含了这个全局版本图的子集:

1. 起始情况

   A 库版本图:

   ```
   a -- b => master
   ```

   全局版本图:

   ```
   a -- b
   ```

2. git clone A B 后, A 和 B 各提交一次

   A 库版本图:

```
a -- b -- c => master
```

B 库版本图:

```
a -- b -- c' => master (tracking branch)
      \
  origin/master (remote branch)
```

全局版本图:

```
a -- b -- c
      \
       c'
```

A 和 B 之间同步, 以完善到包含全局版本图, 就是通过 push 和 fetch 命令实现的(pull = fetch + merge)。

这里有几个需要注意的地方:

- head 和 tag ref 都是只用于在某一个版本库之内标记, 因为它们用一个字符串标记版本, 而不同库里这个字符串可以都用到, 比如都有 master 分支, 上面故意没有在全局版本图上标注 head 就是为了强调 head 和 tag ref 的私有性, 实际在表达全局版本图时往往都会标出各自的 head; 我们平时说 linux kernel-2.6 的上游分支, 其实是因为社会关系, 以 Linus Torvalds 那个公开代码库里的 master 分支为准, 而纯粹从技术上讲, 并不存在一个全局的 master 分支, 这跟集中式的版本管理工具比如 Subversion 全局主分支是很不一样的;
- origin/master 是 git clone 自动建立的分支, 用于跟踪 clone 来源的分支情况, 存放在 .git/refs/remotes/origin/ 下, 用 git fetch 或者 git remote update 更新, 在 .git/config 中 origin 定义了对方库地址;

- B 库里的 master 也是 git clone 自动建立的分支, 用于跟踪 origin/master 分支的变化(git branch --track 可以用于建立这样的分支)，在 .git/config 里定义了 master 分支的跟踪关系;
- GIT 的文档里称 origin/master 这样的为 remote branch，这个叫法相当易混淆, 从上面的图看来, 分支首先分为远端库里的分支和本地库里的分支, 之间可以重名; 其次, 每一个库里的分支又分为 remote branch, tracking branch, normal branch。 remote branch 完全是用来同步对方库里的分支的, 因此绝对不要往 remote branch 上提交; tracking branch 在无参数的 git pull 时会在同步完 remote branch 后, 将 remote branch 往 tracking branch 上合并, 在多人开发时这往往会冲突, 因此也不建议往 tracking branch 上提交。

  [ XXX: tracking branch 太无厘头了，实际上 remote branch 改叫 tracking branch 才合适, 而原来的 tracking branch 以及 pull 命令不应该存在]
- git branch -a 显示的 remote branch 列表并没有 remotes 前缀, 而在 git 里分支只是用 .git/refs/heads 下某个文件记录的, 因此可以建立一个 origin/master 分支, 这样会导致 git branch -a 的输出很迷惑人, 可以用 remotes/origin/master 来显式指代 remote branch。

  [ XXX: refs, refs/heads, refs/tags 都可以省略, 又加上 symbolic ref 添乱, 太容易弄迷糊了]

理解 GIT 里的 branch(准确说是 branch head) 只是各人在全局版本图上的动态标签, 以及 GIT repository 只是保存了这个全局版本图的子集, 对于理解分布式版本控制中的分支关系是非常重要的, 千万不要把 GIT 里的 branch 理解成像 Subversion 里那样一个线性的版本序列。GIT 里所称呼的 remote branch，不过是私有的一个标记, 用来跟踪其它代码库里的某个 head，remote branch 需要不时的用 git fetch 同步才能和对应的 head 指向**全局**版本图中的同一个版本。

# 5 对象记法

关于版本的记法在 git help rev-parse 中有详细描述, 必须学会的是 ~ 和 ^ 记法的含义。

在 git 文档里常能碰上 commit-ish 和 tree-ish 这两个词, commit-ish 的意思是能推导出版本号的字符串, 比如 SHA1 摘要, 包含 ~ 和 ^ 记法的版本号, 版本的日期记法, 标签对象的摘要, 标签名字, 分支名字, 以及这些记法的组合。tree-ish 含义以此类推。

指定某个版本的文件树中内容用: tree-ish:fullpath 语法, 比如 v2.6.27:init/main.c 就表示 linux-v2.6.27 源码中的 init/main.c 文件, 注意 init 前面没有“ /”, 而且也不能是相对于当前目录的相对路径名, 比如在 init/ 目录下也不能用 v2.6.27:./main.c。

[ XXX: 如果目录树很深, 不能用相对路径不方便; 冒号记法可能跟 Windows 盘符记法冲突, 比如有个分支 C，那么可能不小心引用了 C: 盘的文件 ]

查看对象内容可以用 git cat-file -p xxx 或者 git show xxx 。

由于 git 的命令行指定版本并没有用 -r REV 这样的方式, 而是直接用 REV 作为参数, 如果这个命令同时还能接收文件名, 那么往往就会将 REV 和 file 参数混淆, 此时需要在 file 参数前面添加“--”参数。

[ XXX: git 有时能自动分清, 有时不行]

由于 git 自己实现了命令行选项解析, 因此选项不能像 rm aa -f 那样放到后面。

[ XXX: 重造车轮]

git log 参数是个 commit-ish，其含义是说查看从这个版本能够回溯到的所有版本的日志信息, 这跟 svn log 看单个分支的直线历史是不一样的。git log 的这种行为导致看非线性历史的日志时完全不可用, 虽然 git log 后面可以用 .. 和 ^ 记号

做版本树剪枝, 但还是没有 tig 和 gitk 好用。而且由于 git merge 的 fast forward 特性, 导致即使 git log --first-parent 看的也未必是分支**头**曾经指代的那个轨迹。

由于 git log 的版本树回溯特性, 因此看某个版本之后 N 个版本的日志很不方便, 如果要查看的这 N 个版本是线性演化的, 那么可以用下面的命令 $ver 和 $ver 之后第 N 个版本的区别:

```
git diff $ver `git rev-list $ver.. | tail -$n | head -1`
```

## 6 合并之 fast forward

Fast forward 不知道怎么翻译才比较合适, 头一次在 GIT 里见到这个术语, 它指的是在合并两个分支时常常遇到的一种特殊情形, 如下图:

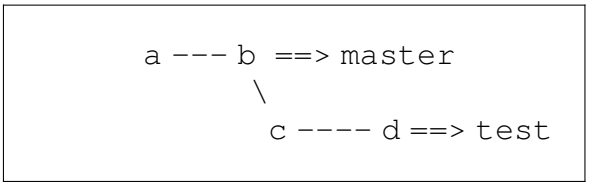


**Figure 2** Fast forward

考虑图 2中的分支情况, master 分支在 test 分支分出去之后没有任何提交, 也就是说 master 分支**真包含于** test 分支, 这时在 master 分支**上**合并 test 分支, GIT 并不会创建一个新的 commit 对象以表达两个分支的合并, 而是直接把 master 这个 head 修改成跟 test 这个 head 一致, 这种特殊的合并就称为 fast forward, 合并后的结果如图 3:

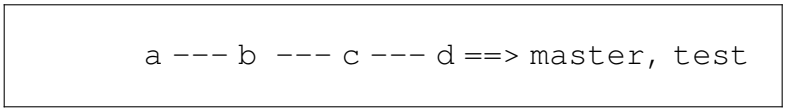


**Figure 3** Fast forward 的结果

Fast forward 的效果很类似于 CVS/Subversion 的 update 操作, 前者是一种特殊的合并, 后者是把 HEAD 版本跟工作拷贝合并。在合并的目的分支没有任何

修改时, GIT 的 fast forward 是把目的分支的状态同步到跟源分支一致; 在工作拷贝没有任何修改时, CVS/Subversion 的 update 是把工作拷贝的状态同步到跟 HEAD 指向的版本一致。

在 Subversion 里, 图 2的情况下把 test 往 master 合并会在 master 分支上创建一个新版本, 再把 master 往 test 合并又会创建一个新版本, 这种策略对于分布式版本控制是个灾难, 无法达到一个稳定的合并状态。

Fast forward 对于分布式版本控制的影响是深远的, 首先, 从图 3看, 合并后的两个分支是完全等价的, 实际上是变成了一个分支, 各自有自己的 head 做标记。这就是一种隐式的自动分支归并, 在全局版本图上, 分叉点的数目取决于实际的修改分歧, 而不是分支操作也就是 git branch 命令被执行的次数, 这种特性避免了分支数目爆炸性的增长弄得合并的工作量巨大不可收拾。考虑如果 Subversion 允许分支存放在多个服务器上, 但仍然按照 Subversion 的分支、合并策略, 那么除非我们显式的删除分支, 分支的数目是只增不减的, 这会给合并带来灾难性的困难。

其次, fast forward 实际上**从全局看**是抹除了在 Subversion 这种版本控制工具里的“ 私有分支” 。用过 Subversion 的人应该很习惯于建立一个私有分支, 然后在上面修改, 并不时的从其它分支上合并修改到私有分支, 在这个私有分支上的每次提交都是分支拥有者的。而在 GIT 里, 如果你不在自己的私有分支上提交, 那么 fast forward 后, 你所谓的“ 私有分支” 不复存在, 即使你在这个私有分支上做了些提交, 但只要别的分支合并过你的分支, 你再合并就会发生 fast forward, 之后你就没有“这个私有分支全部是我私人的提交”这个保证了。个人觉得这种特性是非常有哲学意味的: 从全局版本图看, 看不出来你的分支轨迹, 也就是找不出你建立的 head 曾指代的版本(这在 Subversion 里是可以做到的, 因为分支合并不会减少分支), 你能看到的就是版本图中哪些节点是你参与过的: 想留下痕迹就贡献, 停止贡献就不再有新的痕迹, 历史的演化是靠的所有人堆积起来的贡献, 而不是决定于单个人的痕迹。

GIT 的 fast forward，Subversion 式的总是创建新版本，两种策略的优劣也是引起很多口水的，当初我也是很不习惯没有“私有分支”，理解 fast forward 的意义后也就觉得挺好的了。

如果你想要 svn log 式的直线分支历史，有两种近似的方式，第一，可以用 git log -g xxx，这个会查看 xxx 这个 head 的 ref log，也就是 head 这个 reference 曾经指代过哪些 commit。但是因为 head 可以指向版本图中任何位置，因此 git log -g xxx 看到的并不一定是一条直线历史。第二，可以用 git log --first-parent，意思是在回溯历史遇到 commit 是合并操作时，只回溯其第一个父版本(默认是回溯所有父版本)，如图 4所示，master 曾指向 abefglm，如果只想看这些提交情况，那么默认的 git log master 是不行的，因为会从 m 回溯到图中所有版本，这时 git log --first-parent 就是合适的，因为 git merge 创建 commit 是将合并发生时 HEAD 指向的 commit 作为第一个 parent，所以就在 m 处只去回溯 l 那边，在 f 处只回溯 e 那边，最后得到 mlgfeba 这一系列版本的历史。但是 git log --first-parent 在碰到 fast forward 时会失效，如图 5所示，test 往 master 合并后，git log --first-parent master 就不是 master 曾指代的 gba，而是成了 gdcba，因为 g 的 first parent 是 d。

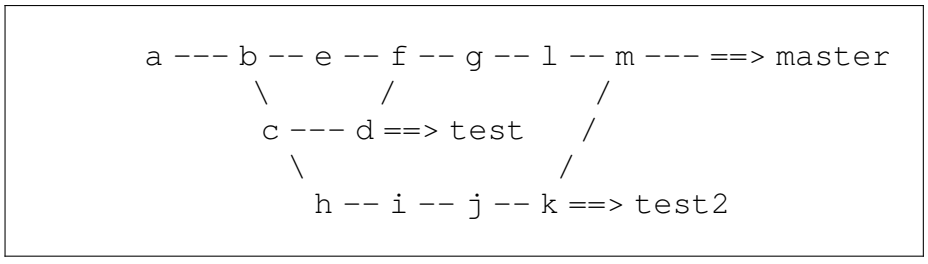


**Figure 4** --first-parent 的作用

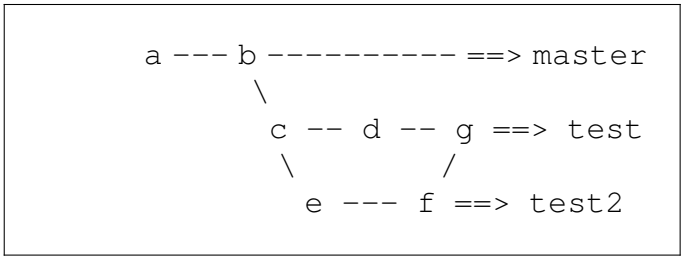


**Figure 5** --first-parent 的失效

既然 --first-parent 并不总是有效的, 与其让 git merge 操作生成的 commit 的 first parent 是当前 HEAD 而让分支合并操作不对称, 不如让 git merge 生成的 commit 的各个 parent 按照 SHA1 值排序, 这样不管 HEAD 是谁, 合并的结果都是一样的。不知道为什么 GIT 没有采用这个方案, 或许是历史原因。

## 7 混乱之源——index

在 git 里文件在逻辑上有三个地方保存: repository, index, work tree。repository 就是指 .git，这里保存了提交过的各个版本的文件。work tree 就是工作目录, index 用来暂存修改 (实际只是存了个索引 .git/index，真实数据在 .git/objects/ 目录下) 以及记录工作目录里文件状态, 其实早先 index 就被称做 cache，现在在命令行选项中也能看到 cache 的字样。

正是 index 的这个暂存修改功能, 导致 git 比起 svn 复杂多了, svn 里的 .svn 只用于记录工作目录里文件状态, 并保存一份干净的基础版本。

下面列举一下涉及 index 的几个命令, 你就能明白为什么有很多人都建议对 GIT 用户隐藏 index 缓存文件这个机制了。

**git add**

将文件从 work tree 加入 index。注意与 subversion 的 add 命令不同, git add 是把文件内容加入 index，在每次修改 work tree 中文件后, 都要 git add 才能让 git commit 把修改提交到 repository 里。

**git rm**

git rm 是从 index 和 work tree 中删除文件, git rm --cached 只从 index 中删除文件。

**git commit**

git commit 是把 index 中文件提交到 repository 里, git commit -a 是把 work tree 中所有修改过的文件都 git add 到 index 里, 然后把 index 中文件提交到 repository 里。

**git reset**

git reset <commit> -- paths... 是从 repository 中的 commit 版本取 paths 指定的文件覆盖 index 中的对应文件, 也就是撤销 index 中暂存的修改。注意输出信息中的“ needs update”是说文件目前状态没有被“ git update-index”命令更新到 index 中, 此处的“update”跟 CVS、Subversion 里的 update 完全是两码事。

git reset --hard <commit> 是取 repository 中的 commit 版本覆盖 index 和 work tree 中的所有文件, 也就是撤销 index 和 work tree 中的修改。注意这个命令不允许指定 paths。

**git checkout**

git checkout <commit-ish> 是切换分支, 取 repository 中指定版本的文件覆盖 index 中文件, 并覆盖 work tree 中文件(如果有区别要用 -f 强制覆盖, 用 -m 来与 work tree 合并)。

git checkout -- paths... 从 index 取文件覆盖 work tree 中文件。

git checkout <tree-ish> -- paths... 从 repository 中的 commit 版本取文件覆盖 index 以及 work tree 中文件。

需要注意的是与 Subversion 里的 revert 命令等价的是

```
git reset -- paths... && git checkout -- paths...
```

或者

```
git checkout HEAD -- paths...
```

而 git revert 相当于 Subversion 里撤销已经提交的修改: 提交一个新版本, 新版本做的修改等效于撤销指定版本的修改。

在用 git checkout 的第一种形式切换分支时, 如果工作目录中有修改没提交, 那么最好用 git stash 先暂存起来, 因为可能出现这样的尴尬境地: 源分支有文件 A、B、C，其中 C 有修改没提交, 目标分支中 A、B 、C 都被修改或者删除, 那么切换分支会因为 C 的修改而告失败(默认不会覆盖或者合并)，但在处理 C 时已经处理完 A 和 B，所以最后分支没切换成功, 但 git status 显示 A、B、 C 都被修改了(A、B 是中断的分支切换操作误引起的, C 是切换分支前的正常修改 )，如果文件很多, 你就分不出来哪些是切换分支前修改过的。此时的挽救办法是执行 git diff --name-status <目标分支>，输出的就是切换分支前修改的文件, 排除这些, git status 显示的就是失败的分支切换误引起的。

**git status**

git status 的输出分为三个部分, 第一部分是“Changes to be committed”, 列举的是 index 中的文件状态, 用不带 -a 的 git commit 会把这些文件提交到 repository 中; 第二部分是“Changed but not updated”, 这部分列举的是 work tree 中被 git 管理的文件状态, 用 git add 会把这些文件加入到 index 中, 这里的“update”一词来源于 GIT 底层命令 update-index；第三部分是“untracked files”, 这部分列举的是 work tree 中没有被 git 管理的文件。

在 git status 第一部分输出中有“ unstage”这个词，“ stage”这个字眼是为了避免“ index”这个太易混淆的词, stage 是指 git add/rm，unstage 指 git reset，staged files 就是指 index 中暂存的文件修改。

**git diff**

由于在 repository 中有多份文件树, 在 index 和 work tree 中各有一份文件树, 因此 diff 的功能非常丰富:

git diff 比较 index 和 work tree

git diff --cached tree-ish 比较 index 和 tree-ish，后者缺省为 HEAD

git diff tree-ish 比较 tree-ish 和 work tree

git diff tree1-ish tree2-ish 比较两个 tree

tree-ish 可以是文件树的一个部分, 比如

```
git diff v2.6.21:init v2.6.27:init
```

上面命令末尾都可以添加 -- path...，表示只输出牵涉到这些路径的结果。

[ XXX: 繁复, 无视正交设计原则。其实可以直接用一个 index 引用来标记, 限制用户不能自己建立 index 引用, 类似于 Subversion 的 -r BASE 里的 BASE，造成现在这个局面无非是懒: 懒输入 index 所以就用隐含参数和选项, 懒得输入 -r 所以要用 -- 分隔版本号和文件名 ]

## 8 工作流程

由于 GIT 太灵活了太繁复了, 所以 GIT 用户常常结合自己的使用经历总结 GIT 的工作流程, 在 GIT 源码的 Documentation/howto 和 GIT 邮件列表中有很多这样的总结。

我个人的体会有如下几点:

1. 不要往不是自己手动建立的分支上直接提交。

这里说的主要是 tracking branch，也就是 .git/config 中 [branch] 节配置的那些分支，因为 remote branch 一般都不会误提交上去。我感觉压根就不应该使用 tracking branch，因为 git pull 的合并百分之九十的情况下都不是想要的结果，而且经常发生冲突，即使确实想合并，执行下 `git merge` 也不费事，不建议使用 pull 命令。另外我常常希望上游分支的历史尽量是线性的以方便查看，所以我常用 rebase 和 cherry-pick 而不是 merge 或者 pull 来处理工作分支。

我跟踪别人分支的办法是用 `git fetch repos src:dst` 或者 `git remote update`，前者比较随意，目标分支可以不在 refs/remotes/ 这个命名空间里，后者比较正规。

有了 remote branch 后我再建立个 normal branch 就行了[12]，想提交到正式库时，先更新 remote branch，然后 rebase normal branch，然后 push 或者 fetch 到正式库中的上游分支。

这样最少可以只牵涉到三个分支：正式库里的上游分支，本地库里的 remote branch 和工作分支。而如果要用 tracking branch 的话，为了避免 pull 时冲突，我不会把 tracking branch 作为工作分支，这样 tracking branch 和 remote branch 的功能就完全重复了。实际上我很少用 pull，所以 tracking branch 对我没用，只会让人迷糊。

如果我的本地库要提供一个分支让人跟踪，那么我一般是再建立个工作分支，可以随意提交修改，改好后再建立个分支用`git merge --squash` 和 `git cherry-pick`来整理这些修改，弄完后`git rebase pub` 然后用 `git push . my:pub` 更新我的公开分支，不往公开分支上直接提交。

2. 不要往 HEAD 指向的分支 push。

   push 的目标分支应该不会被 checkout 出来，因为 push 过后，head 指向的位置变了，导致 index 跟 HEAD 版本不匹配而出现很莫名的文件被修

[12] git checkout -b my origin/master 会自动在 .git/config 中把 my 分支配置为 tracking branch，去掉这个配置就行了。

改状态, 虽然可以用 `git reset --hard` 或者更细致的 `git reset && git checkout paths...` 纠正, 但总是不便。

3. 尽量不要回滚公开分支。

   公开分支如果回滚了, 比如被 rebase，commit --amend 或者强制修改了 head 的指向, 都会给别人从这个分支上 pull 造成麻烦, 即使执行 pull 的人本地没有做任何修改也会出现很莫名的冲突。

## 9 常用命令简介

**git add**

往 index 里添加文件内容。加上 -i 选项可以交互式的选择需要 add 哪些内容, 细到 patch hunk 级别, 非常实用。

**am**

从邮箱里提取补丁并应用。git apply-mbox 是过时的命令。

**apply**

应用补丁。

**archive**

将指定版本的文件树打包, 支持 tar 和 zip 格式。

**bisect**

对半查找版本树中引入 bug 的版本, 由于版本树往往不是线性的, 手工记录验证过的版本并且剪枝是很琐碎的。

**blame**

查看每一行是从哪个版本来的, 谁修改的。git annotate 是过时的命令。

**bundle**

将指定版本的各种对象以及 refs 打包, 方便离线开发。

**cat-file**

判断对象类型, 以及原样或者格式化输出对象内容, 注意对于 tree 对象要用 ls-tree 或者 cat-file -p 或者 show 命令查看, cat-file 的原样输出是二进制数据。

**checkout**

切换分支或者撤销修改。

**cherry-pick**

相当于将某次提交打个补丁应用到当前分支, 用来从其它分支提取所需的修改而不用合并整个分支。

**clean**

删除工作目录中没有被 git 管理的文件, 非常方便。

**clone**

复制代码库, 如果目标库和源库在同一台机器上, -s 选项能够很节约磁盘空间。另外 --depth 选项可以避免复制代码库中包含的整个版本历史。

**commit**

提交一个新版本。

**config**

配置 git 的行为, 有三种配置: system 范围的, 指 `/etc/gitconfig`, global 范围的, 指 `$HOME/.gitconfig`, repository 范围的, 指 `.git/config`, 三者的

优先级依次增高。config 命令默认操作的是 `.git/config`。repo-config 是过时的命令。

config 命令可以用来设置命令别名, 比如将 b 设置为 branch 的别名:

```
git config --global alias.b branch
```

一份示例配置 `$HOME/.gitconfig`, 具体含义参考 `git help config`:

```
[user]
    email = YourEmail
    name = YourName
[color]
    ui = auto
[alias]
    ci = commit
    st = status
    co = checkout
    b = branch
[diff]
    renames = copy
[rerere]
    enabled = true
[i18n]
    commitEncoding = utf-8
    logOutputEncoding = utf-8
```

**count-objects**

统计 loose object 个数以及所占空间。

**describe**

将某个版本用最近的 tag 加上 ~ 和 ^ 记法来显示, 方便识别, 有一点点类似 Subversion 里的整数版本号。

**diff**

查看文件内容修改情况。

**fetch**

从远端代码库同步版本图到本地代码库, 并修改本地库里指定的 head 跟远端库里对应的 head 一致, 这个修改要求本地的 head 指示的分支**真包含于** 远端库里对应 head 指示的分支, 否则要加 -f 选项。

**format-patch**

将分支上的修改生成补丁序列, 跟 am、imap-send、send-email 命令搭配构成了利用 Email 进行协作分布式开发的工作流程。

**gc**

由于 GIT 里存储对象分为 loose object 和 packed object，因此需要不时用 repack 将 loose objects 打包成 pack 格式存储的 packed objects，以节省磁盘空间并提高访问效率。此外还有一些其它需要清理的文件, gc 作为一个高层命令统管了这些杂事。

**grep**

在指定版本的文件树里搜索包含特定模式的行。

**gui**

GIT 的图形界面, 个人觉得不大好用。

**init**

新建一个 GIT 版本库。init-db 是过时的命令。

**log**

查看版本历史。

**ls-files**

显示 index 和 work tree 中的文件列表, 常用于 Shell 编程。

**merge**

分支合并。

**mergetool**

在合并冲突发生后, 此命令会对每个冲突的文件调用第三方合并工具, 比如 kdiff3(最好的图形三路合并工具)来进行可视化的合并, 非常方便。

**mv**

重命名、移动文件或者目录, 这个命令只是为了方便, GIT 并不在版本库里记录这个操作。

**pull**

pull = fetch + merge，不推荐用这个命令。

**push**

与 fetch 功能类似, 命令行格式一致, 只是操作的方向相反, 也接收 -f 选项。

**rebase**

将一个分支上的版本序列提取出来在另一个分支上重新提交一遍, 这个命令可以用来保证上游分支历史是直线的, 有一点类似于 CVS 和 Subversion 里的 update 命令, 只是更强大, 能一次处理多个版本, 在 CVS 和 Subversion 里, 在某一个分支上工作的操作序列是这样的:

```
checkout <branch>
...modify....
update
commit
```

update 操作保证了 commit 的提交总是在最新版本的基础上, 这样分支历史就是一条直线。

在 GIT 里, 类似功能的操作序列是这样的:

```
git clone <remote_repos> <local_repos>
checkout -b <my_branch> <remotes/upstream_branch>
...can modify and commit many times...
...`fetch` or `remote update` to update <remotes/upstream_branch>
rebase <remotes/upstream_branch>
commit
push remote_repos <my_branch>:<upstream_branch>
```

由于 GIT 的分布式版本控制特性, 类似功能的命令 GIT 显得要繁琐的多。

**remote**

管理 remote branches。

**reset**

撤销 index 和 work tree 中未提交的修改。

**revert**

撤销已经提交的修改。

**rm**

删除文件。

**shortlog**

统计提交情况。

**show-branch**

显示各个分支的提交情况, 方便查看是否有提交需要合并或者 cherry-pick。

**stash**

暂存 index 和 work tree 的状态, 稍候可以再恢复, 非常有用的命令。

**status**

查看 index 和 work tree 中文件状态。

**svn**

与 Subversion 的双向交互, 这样可以用 GIT 来当作更强更快的 Subversion 的客户端了。

**tag**

管理静态标签。

## 10 GIT 的模块功能

git module 我还没用过, 粗看了下貌似是比较复杂, 猜测也不大好用, 不然 Xorg 和 Android 项目不会自己写辅助脚本了。