

GIT——分布式版本管理系统

前言

本资料以《Git Community Book》为基础，综合个人的理解编写。本资料有误或待讨论之处，欢迎联系作者：chenj@lemote.com。

第一章 基本概念.....	4
1.1 GIT 对象模型.....	4
1.1.1 SHA 1 命名对象.....	4
1.1.2 对象.....	4
1.1.3 与 SVN 的差异.....	4
1.1.4 各个对象图示.....	6
1.1.5 指定 GIT 对象.....	7
1.1.6 指定仓库.....	8
1.2 .git 目录和工作目录.....	9
1.2.1 .git 目录.....	9
1.2.2 工作目录.....	9
1.2.3 GIT 索引 (.git/index)	10
第二章 初级用法.....	11
2.1 初始化.....	11
2.2 获得 GIT 仓库.....	11
2.3 一般的工作流程.....	11
2.4 基本的分支与合并.....	12
2.4.1 如何合并.....	13
2.4.2 撤销一次合并.....	14
2.4.3 Fast-forward (快速向前) 合并.....	15
2.5 回顾历史——git log.....	15
2.5.1 显示统计日志.....	16
2.5.2 格式化日志.....	17
2.5.3 排序显示日志.....	18
2.6 比较 commits——git diff	18
2.7 分布式的工作流程	19
2.7.1 公开 git 仓库.....	20
2.7.2 push 失败了怎么办？	21

2.8 git tag.....	22
2.8.1 轻量级的 tag.....	22
2.8.2 Tag 对象.....	22
2.8.3 签名的标签 (tags)	22
第三章 中级用法.....	23
3.1 忽略 (不追踪) 文件.....	23
3.2 REBASING	23
3.3 交互式 REBASING	26
3.4 交互式的 git add.....	28
3.5 STASHING (暂存工作目录中的未提交的修改)	29
3.5.1 Stash 队列.....	29
3.6 追踪分支.....	30
3.7 使用 git grep 进行查找.....	30
3.8 GIT 中的撤销的功能——reset , checkout 和 revert.....	32
3.8.1 修正某个未提交的失误	32
3.8.2 修正已提交了的错误.....	33
3.9 维护 GIT	33
3.9.1 维护 GIT 的高效率.....	33
3.9.2 确保可靠性.....	34
3.10 设置一个公开的仓库.....	34
第四章 高级用法.....	35
4.1 创建新的空分支 (无祖先分支)	35
4.2 修改历史.....	35
4.3 分支与合并的高级用法.....	35
4.3.1 合并时获取解决冲突的帮助.....	35
4.3.2 多路合并.....	36
4.3.3 Subtree (合并他人的分支到子目录下)	36
4.4 SUBMODULES (把他人的项目纳为你的项目的一个子模块)	36
4.5 git bisect (定位某个引入问题的 commit)	39
4.6 git blame (查找问题)	40
4.7 GIT 和 email.....	40
4.8 GIT 的钩子.....	41
4.8.1 服务端的钩子.....	41
4.8.2 客户端钩子.....	41

4.9 定制 GIT.....	41
4.9.1 git config	41
4.10 修复损坏的 GIT 对象.....	42
附录.....	43
gpg 的使用.....	43
基本使用.....	43
密钥管理.....	43
签名密钥（用于认证公钥）	43
加密与解密	44
签名与验证	44
gitosis 的使用.....	44
安装.....	44
配置.....	45
运行 git-daemon.....	46
配置 gitweb，允许 web 访问 GIT 仓库.....	46

第一章 基本概念

1.1 GIT 对象模型

1.1.1 SHA 1 命名对象

GIT 使用 40 位的十六进制的 SHA1 码来标明其对象，例如：

6ff87c4664981e4397625791c8ea3bbb5f2279a3

该 SHA1 码是通过计算对象**内容**的 SHA1 哈希值获得的。这样作有如下好处：

- GIT 可以迅速决定两个对象是否一致（只需比较对象名字）。
- 两个仓库中相同的内容相同将存储于同一对象名下。
- GIT 可以通过再次计算对象内容的 SHA1 哈希值与该对象名相比，来侦测出否出错。

1.1.2 对象

每个对象包括三域：

- 对象的类型（blob、tree、commit 还是 tag）
- 内容的长度
- 内容

blob 类型对象用来存储文件内容——类似文件。以下简称“blob”。

tree 类型对象类似目录，包含了其他的 tree 和/或 blob 类型的对象（即子目录和/或文件）。以下简称“tree”。

commit 类型对象指向单个 tree（项目某个时刻的状态），还包含了相关的元数据，比如时间戳、该次改动的作者、指向前次 commit 的指针等等。以下简称“commit”。

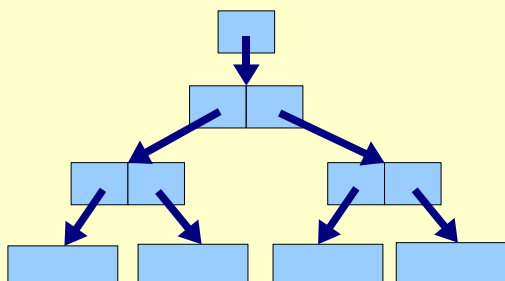
tag 类型对象，用来标记某个 commit（或 tree 等）是特殊的。通常用来标记某个 commit 作为该软件的一个发布。

1.1.3 与 SVN 的差异

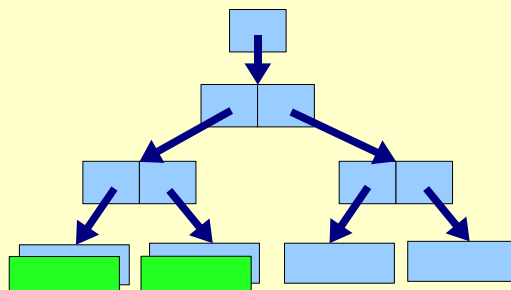
GIT 与大多数的 SCM 系统不同。大多数 SCM 是差分存储系统（Delta Storage systems）——存储某个和 commit 和下一个 commit 的不同之处。GIT 不是这样的，每次提交时，它用 tree 中来存储项目中所有文件的快照。

i 讨论： GIT 每次提交保存项目所有文件的快照做法和现在 COW (Copy On Write) 文件系统和相似。

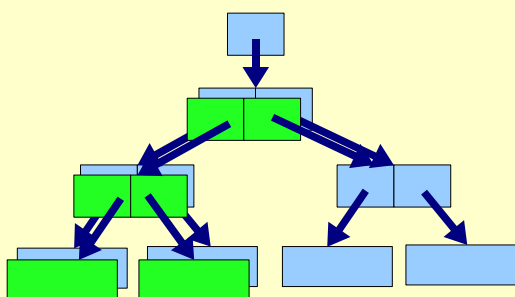
1. 未 commit 前的 tree



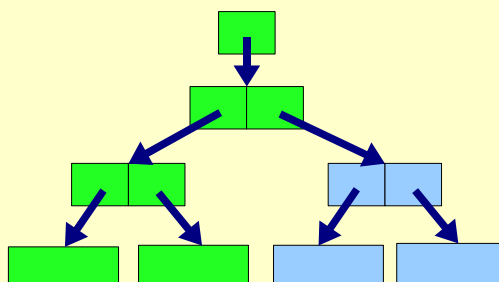
2. COW 修改 blob——
拷贝原有 blob (浅绿色块) 并修改



3. COW 修改中间的 tree 对象，
使之指向新的 blob 和 tree



4. COW 修改“根” tree 对象
commit 指向这个新的“ tree”



上图显示了猜想的进行 commit 的过程，修改 blob 对象——先拷贝出原始 blob 对象再进行修改。接着修改直接包含该的 blob 的 tree 对象，使之指向新的 blob 对象——同样是拷贝出原 tree 对象再进行修改，...，最后产生新的“根” tree (并在该次 commit 中指向该 “根” tree)。

1.1.4 各个对象图示

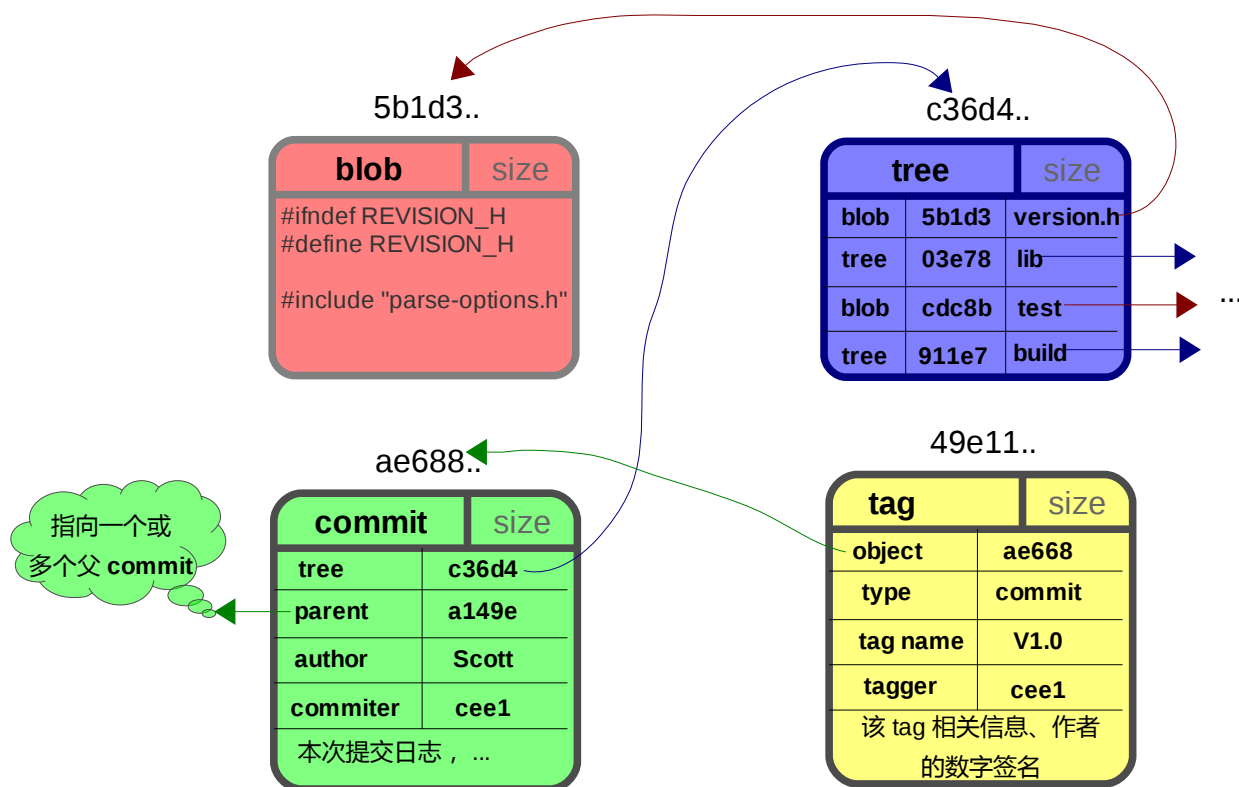


图 1：GIT 的四种基本对象的结构

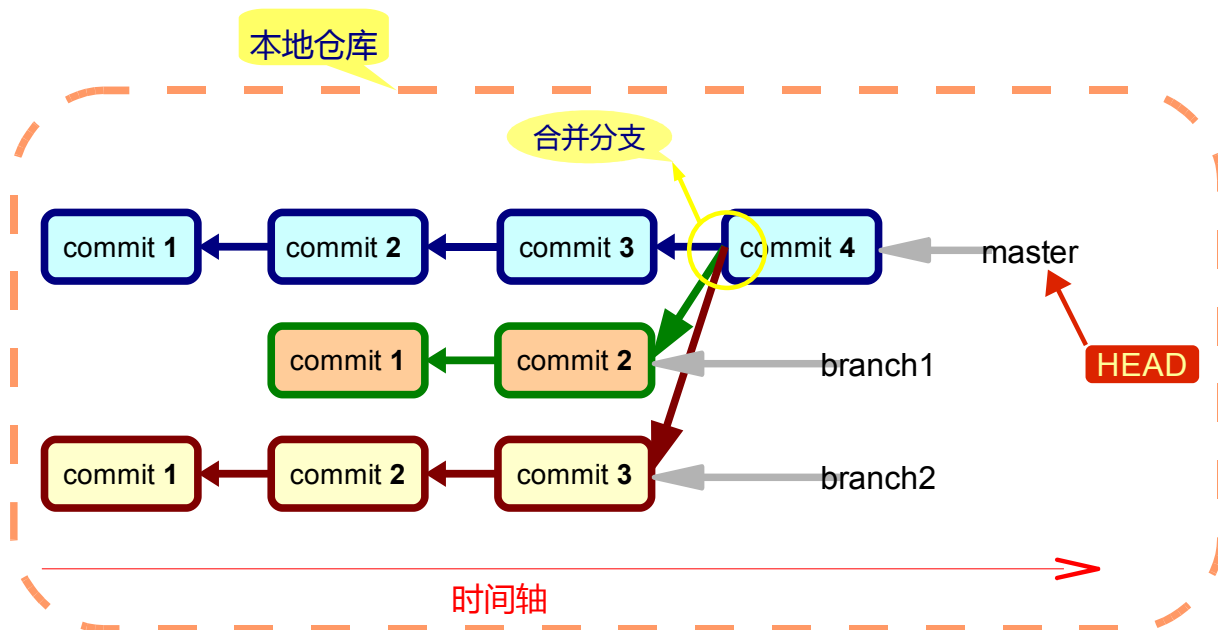


图 2：commit 对象与分支、仓库

图 2，commit 对象形成的单向链，而分支则是指向该链中**最新**的 commit。这个最新 commit 称为该分支的 tip。从图中可见：

- ➔ 何为分支？分支就是表示某个开发线（development line）的 commit 链，分支头指向**最新**commit（通常情况下，见 git reset）的指针。

➔ 何为仓库？仓库就是多个分支的集合。

1.1.5 指定 GIT 对象

- 使用对象的 SHA1 码。注意可以输入 40 位的十六进制 SHA1 码的前几位，能区别即可。
- 使用 .git/refs 目录下的“指针”进行引用（然而，出于效率的考虑，这些引用也可能打包到单个文件。见 git pack-refs），例如：

```
# 查看 master 分支（查看该分支最新的 commit 对象）
$ git show master

...

# “指针”内容实际就是对象的 SHA1 码
$ cat .git/refs/heads/master
c11d54bf82e376ce58a88c129198d10a409d2fb3

$ cat .git/refs/remotes/origin/
b634ef1e76f14b2767ce821de2cb554fc1bc7a3f
```

- 使用 .git 下的 HEAD 等的引用对象。

```
# HEAD 实际指向了当前仓库激活的分支
$ cat .git/HEAD
ref: refs/heads/master
```

- 使用 git rev-parse 支持的语法指定 commit 对象。
 - 使用日期指定，例如 **master@{yesterday}**、**master@{1 month ago}**。
 - 使用顺序指定，例如 **master@{5}**，距今第五次提交的内容，当前提交为 master@{0}
 - 例如，对于 commit 对象 A
 - **A^n** 代表 A 的第 n 个**直接父** commit（在不合并的情形下，n=1）。
 - **A^** 相当于 A^1，A^0 则等于其自身。
 - A^^ 可简写为 **A~2**，代表 A 的**第一个直接父** commit 的**第一个直接父** commit。
 - 图 2 的一些例子：

master 分支的 commit4 master、master^0

master 分支的 commit3 master^1（简写为 master^）、master~1

master 分支的 commit2	master ^{1^1} 、master ^{^^} (可写为 master~2)
master 分支的 commit1	master ^{^^^} 、master~3
branch1 分支的 commit2	master ^{^2}
branch1 分支的 commit1	master ^{2^1} 、master ^{2^} 、master ^{2~1}
branch2 分支的 commit3	master ^{^3}
branch2 分支的 commit2	master ^{3^1} 、master ^{3^} 、master ^{3~1}
branch2 分支的 commit1	master ^{3^^} 、master ^{3~2}

■ 使用对象中含有的引用。例如：

- commit 对象中的 *tree* 域，指向一个 tree 对象，因此在需要 tree 对象的命令中，可以使用 commit 对象代替（这被称为 tree-ish）。
- commit 对象转换为 tree 对象，在 commit 对象的引用之后加 “^{tree}”，例如 **master^{tree}**
- tag 对象中的 *object* 域，指向一个 commit 对象（通常这么作），因此可以使用 **tag 对象^{^0}**，来转换为 commit 对象。
- 指定 Tree 对象指向中的 blob 对象，例如 **master:path/to/file**（这里采用了 tree-ish，因为 master 指向 commit 对象）

1.1.6 指定仓库

GIT 是个分布式的版本管理系统，因此 GIT 不存在中央仓库，取而代之的是每个开发者的一个本地仓库与其他开发者的远程仓库。

仓库在如下情况下需要指定：

- 某些 git 命令中，需要指定仓库。“.”代表本地仓库。
- 使用引用（“指针”），指定其他仓库中的 git 对象。
- 某些 git 命令，需要指定分支，可以用指定仓库代替。这时指定的分支是该仓库的激活的分支（由.git/HEAD 指明）。

仓库可以直接用文件系统路径、URL（目前支持 http、git、ssh 或 rsync 协议）指定，也可以用 git remote 命令添加仓库名：

```
# git remote add <仓库名> <URL 或文件系统路径>
git remote add origin git://dev.lemote.com/demo.git
```


这时会在.git/refs/remotes/<仓库名>出现远程仓库的分支，相关对象被缓冲下来。这些分支可以使用 <仓库名>/<远程分支名> 来引用，例如：

```
git show origin/master
```

这些分支不能直接 checkout 出来进行开发，而要与本地分支合并后进行开发，例如

```
# 新建本地分支 new_branch，用之合并远程 master 分支的内容。
git checkout origin/master -b new_branch
```



把分支命名为 master、远程仓库命名为 origin，是 git 的一些习俗。

1.2 .git 目录和工作目录

1.2.1 .git 目录

.git 目录存储了关于你的项目所有的历史和元信息。包括：

- GIT 对象 (commits、trees、blobs、tags)。
- 所有指向不同分支所在的“指针”（分支头）。
- ...

一个项目下只有一个.git 目录（位于项目根目录下，而不像 SVN 或 CVS 那样，项目每个子目录下还有一个）。在你项目的根目录下，浏览.git 目录的内容：

```
.
|-- HEAD          # 指向当前的分支
|-- config        # 该项目的 GIT 相关配置
|-- description   # 该项目的描述（gitweb 使用）
|-- hooks/        # 发生某些事件之前/之后的运行的钩子
|-- index         # 索引文件，见下
|-- logs/         # 关于分支头指向过哪些 commit 的历史
|-- objects/      # 存储 GIT 对象 (commits, trees, blobs, tags)
`-- refs/         # 指向一些 commit 对象、tag 对象的“指针”（分支头、标签）
```

(也许还有其他一些文件/目录，但这里不过多谈及)

1.2.2 工作目录

GIT 中，*工作目录*指存有那些检出 (checkout) 的、当前你正在进行工作的文件。工作目录中的文件和目录通常在切换分支时，被 GIT 移除或替代。所有历史存储在.git 目录下，工作目录仅是一个临时的、存放检出 (checkout) 文件和目录的、供你进行修改的场所。

1.2.3 GIT 索引 (.git/index)

GIT 中，需要进行三种“同步”（确切的讲是合并）操作：

- 工作目录与本地分支的同步（git commit、git reset、git checkout、...）
- 本地分支间的同步（git merge、...）
- 本地分支与远程分支间的同步（git pull、git push、...）

其中，工作目录与当前分支同步操作并不是直接进行，两者同步需要通过索引来进行。索引可以用来构建，一次提交的变更集。当你进行提交时，实际提交的是索引中的内容，而不是工作目录的。



图 3：常见的开发线（当前分支）同步工作目录的过程

查看索引的最简单的方式是使用 git status 命令，运行该命令后，可见哪些文件被进入索引/（be staged），哪些文件在工作目录中修改但没有进入索引/，哪些文件不被 GIT 所追踪。

```
$ git status
```

```
# On branch master
# Your branch is behind 'origin/master' by 11 commits, and can be fast-forwarded.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   daemon.c
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   grep.c
#   modified:   grep.h
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   blametree
#   blametree-init
#   git-gui/git-citool
```

第二章 初级用法

2.1 初始化

使用 git config 来设置网名和电子邮件地址：

```
$ git config --global user.name "Scott Chacon"  
$ git config --global user.email "schacon@gmail.com"
```

上述命令将写入 git 全局配置。该配置文件默认在 ~/.gitconfig，其内容将增加下述几行：

```
[user]  
    name = Scott Chacon  
    email = schacon@gmail.com
```

如果只要设置某个特定项目的上述选项（例如，使用某个工作用的电子邮件地址），可以运行 git config 命令，不带 --global 参数。这样做会在 .git/config 加入 [user] 段（并且该处配置优先于用户目录下的 .gitconfig）。

2.2 获得 GIT 仓库

a. 通过克隆远程的仓库：

```
# 通过 git 协议，结果在当前目录的名为 "git" 的目录下  
git clone git://git.kernel.org/pub/scm/git/git.git  
# 或通过 http 协议  
git clone http://www.kernel.org/pub/scm/git/git.git
```

b. 或通过初始化得到一个 git 仓库

```
$ tar xzf project.tar.gz  
$ cd project  
$ git init
```

显示为：Initialized empty Git repository in .git/

2.3 一般的工作流程

1. 把一些文件纳入 git 的管理：

```
$ git add file1 file2 file3
```

💡 若 git add 之后的参数中有目录，则 git 会递归地把该目录下的所有文件纳入管理。

💡 对已经通过 git add 加入 git 管理的文件，之后可再次用 git add <该文件>。此时代表更新该文件的索引，作用同 git-update-index <该文件>。这点意义上来说，git 追踪的是内容而非文件。

2. 使用 git diff 查看待提交 (commit) 的内容

```
# 查看索引与最近一次 commit 的差异
$ git diff --cached

# 或查看工作目录与索引的差异 ( 工作目录中, 没有加入到索引的改变 )
$ git diff
```

也可以使用 git status 获得更改的简报 :

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   file1
#   modified:   file2
#   modified:   file3
#
```

3. 使用 git commit 把修改提交到当前分支中。

```
git commit -a
# git 会打开一个文字编辑器, 来编辑本次提交的日志信息。
```

💡 可以通过 git commit -a -m "本次提交的日志", 来上述交互。

💡 git commit -a, 同步工作目录->索引->当前开发线。而 git commit (不带-a 参数), 仅是把索引提交到当前开发线。

💡 提交的日志的格式, 推荐为: 小于 50 字符的变更简述起头, 后为一空行, 接下来为完整的描述。

2.4 基本的分支与合并

仓库是存储多个分支的场所。创建当前分支的一个子分支, 使用 :

```
$ git branch experimental
```

现在, 运行 :

```
$ git branch
```

将会看到当前存在的本地分支 :

```
experimental
* master
```

"experimental"分支是刚刚创建的分支, 而 "master"分支是当前分支 (由前的 "*"所指示)。下面的命令, 切换到 experiment 分支 :

```
$ git checkout experimental
```

若 git 提示拒绝切换，则是你当前工作目录是“脏”的（结果没有提交到当前开发线中）。可以选择提交之后再运行上述命令，或者在 git checkout 后加 -f，来强制切换（“脏”的内容将被遗弃）。

接下来，编辑文件，提交改变，然后切回到 master 分支：

```
( 编辑文件 )  
$ git commit -a  
$ git checkout master
```

此时，你刚刚编辑的内容已经看不到了。因为你的刚才修改在 experimental 分支中，而现在已经回到 master 分支了。

在 master 分支中进行修改¹：

```
( 编辑文件 )  
$ git commit -a
```

此时，两个分支有不同的修改。为了合并 experimental 分支中的变更到 master 分支中，运行：

```
$ git merge experimental
```

若之前的两个分支中的修改不导致冲突，则合并工作顺利完成。若存在冲突，则会在有冲突的文件中留下记号，使用：

```
$ git diff  
将会看到这些冲突。你需要编辑有冲突的文件，然后运行：  
$ git commit -a
```

来提交合并的结果。最后，运行：

```
$ gitk # 需要 tk/tcl 支持
```

将会显示一个漂亮的图形界面来表示上述操作的历史。

此时，可以通过下述命令来删除 experimental 分支：

```
$ git branch -d experimental
```

此命令保证 experimental 分支中的变更已经保留在当前分支中。而使用

```
$ git branch -D experimental
```

则会无条件删除分支。

2.4.1 如何合并

```
$ git merge next
```

1 若不进行此操作，则之后的 merge 操作，git 会显示 “Fast forward” 字样。

上面一条命令将把名为 “next”分支中的变更合并到**当前**分支中。若存在冲突，例如同一个文件同一部分在两个分支中，被采用不同的方式修改，则 GIT 会警告你产生冲突：

```
$ git merge next
100% (4/4) done
Auto-merged file.txt
CONFLICT (content): Merge conflict in file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

产生冲突的文件中将会留下冲突的标记：

```
cat file.txt
<<<<<<< HEAD:file.txt
Hello world
=====
Goodbye
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

手动编辑该文件来解决冲突，然后输入：

```
# 把文件 file.txt 内容更新到索引中
$ git add file.txt
# 把索引提交到当前开发线中
$ git commit
```

2.4.2 撤销一次合并

合并中出现问题（合并并未提交），下述命令可以返回到合并前的状态：

```
$ git reset --hard HEAD2
```

若此次合并已经提交，而想撤销之：

```
$ git reset --hard ORIG_HEAD
```

使用后一个命令，**不能**用在（待撤销的）合并提交（commit），被合并到另一分支的情形下。不然会迷惑进一步的合并。

💡 git reset commit-ish，有三个选项：

--mixed：默认选项。调整当前分支头（“指针”），使之指向 commit-ish，并同步索引至 commit-ish 对应状态。但保留工作目录中的修改。

--soft：仅调整当前分支。

--hard：调整当前分支，指向 commit-ish，并同步索引和工作目录至 commit-ish 对应状态。工作目录中的修改会丢失！

² 确保合并前的工作目录是“干净”的（即提交（commit）修改后再进行合并操作），不然这些内容将丢失。

2.4.3 Fast-forward (快速向前) 合并

如下例子，branch1 分支从 master 分支的 a commit 分出，之后进行了两次提交 b 和 c

。
 --- a <--- master
 \
 b <--- c <--- branch1 *

之后，切换到 master 分支，合并 branch1 分支，则：

--- a <--- b <--- c <--- master, branch1

现在 master 和 branch1 指向同一个 commit，master^{^1} 是 b，而 b 并不在原先的 master 分支上。合并后，master 和 test 的历史是一致的，没有区别，这个 CVS 和 SVN 的做法是不同的。按照 CVS 或者 SVN 的做法，合并后 show log master 看到的应该是 a 和 d（d 的父 commit 是 a 和 c），但是 git log master 看到的是 a b c。

上述的例子，显示了 GIT 的 Fast-forward 合并：

- 通常，一次合并将导致合并的 commit 有多个父 commit（每个对应一条开发线）。
- 然而，当当前分支中的每个 commit，另一（进行合并）分支都包含。则合并那个分支，GIT 将会进行 Fast-forward 合并，当前分支的“头”将会指向合并进来的分支的“头”所指的 commit。而不创建新的合并 commit。

2.5 回顾历史——git log

git log 命令显示一系列提交。一般的 git log 显示指定的 commit 可达的所有 commits，但也可如下指定只显示某些 commits：

```
$ git log v2.5..          # 自从 v2.5 以来的 commits（不包括从 v2.5 可达的 commits）
$ git log test..master    # 从 test 到 master 的 commits（从 master 可达的 commits，但不包
                           # 括从 test 可达的）
$ git log master..test     # 从 test 可达的 commits，但不包括从 master 可达的
$ git log master...test    # 从 test 或 master 可达的 commits，但不包括两者皆可可达的
$ git log --since="2 weeks ago" # 最近两周以来的 commits
$ git log Makefile        # 显示修改了 Makefile 的 commits
$ git log fs/             # 显示修改了 fs/ 下任一文件的 commits
$ git log -S'foo()'        # 显示增加/移出文件内容，符合字符串 "foo()" 的 commits
$ git log --no-merges      # 不要显示合并 commits
```

当然，可以组合上述，例如要查找自 v2.5 以来修改 Makefile 文件或者 fs 目录下任一文件的 commits：

```
$ git log v2.5.. Makefile fs/
```

git log 将会显示按照日期，自新向旧显示符合参数的 commits：

```
commit f491239170cb1463c7c3cd970862d6de636ba787
Author: Matt McCutchen <matt@mattmccutchen.net>
Date: Thu Aug 14 13:37:41 2008 -0400
```

git format-patch documentation: clarify what --cover-letter does

```
commit 7950659dc9ef7f2b50b18010622299c508bfdfc3
Author: Eric Raible <raible@gmail.com>
Date: Thu Aug 14 10:12:54 2008 -0700
```

bash completion: 'git apply' should use 'fix' not 'strip'

Bring completion up to date with the man page.

也可要求 git log 按照补丁的形式显示：

```
$ git log -p
```

```
commit da9973c6f9600d90e64aac647f3ed22dfd692f70
Author: Robert Schiele <rschiele@gmail.com>
Date: Mon Aug 18 16:17:04 2008 +0200
```

adapt git-cvsserver manpage to dash-free syntax

```
diff --git a/Documentation/git-cvsserver.txt b/Documentation/git-cvsserver.txt
index c2d3c90..785779e 100644
--- a/Documentation/git-cvsserver.txt
+++ b/Documentation/git-cvsserver.txt
@@ -11,7 +11,7 @@ SYNOPSIS
```

SSH:

[verse]

```
-export CVS_SERVER=git-cvsserver
+export CVS_SERVER="git cvsserver"
'cvs' -d :ext:user@server/path/repo.git co <HEAD_name>
pserver (/etc/inetd.conf):
```

2.5.1 显示统计日志

```
$ git log --stat
```

```
commit dba9194a49452b5f093b96872e19c91b50e526aa
Author: Junio C Hamano <gitster@pobox.com>
Date: Sun Aug 17 15:44:11 2008 -0700
```

Start 1.6.0.X maintenance series

```
Documentation/RelNotes-1.6.0.1.txt | 15 ++++++
RelNotes                             | 2 +-
2 files changed, 16 insertions(+), 1 deletions(-)
```


2.5.2 格式化日志

```
$ git log --pretty=oneline
```

```
a6b444f570558a5f31ab508dc2a24dc34773825f  dammit, this is the second time this has reverted
49d77f72783e4e9f12d1bbcacc45e7a15c800240  modified index to create refs/heads if it is not
9764edd90cf9a423c9698a2f1e814f16f0111238  Add diff-lcs dependency
e1ba1e3ca83d53a2f16b39c453fad33380f8d1cc  Add dependency for Open4
0f87b4d9020fff756c18323106b3fd4e2f422135  merged recent changes: * accepts relative alt pat
f0ce7d5979dfb0f415799d086e14a8d2f9653300  updated the Manifest file
```

或者

```
$ git log --pretty=short
```

```
commit a6b444f570558a5f31ab508dc2a24dc34773825f
Author: Scott Chacon <schacon@gmail.com>
```

```
    dammit, this is the second time this has reverted
```

```
commit 49d77f72783e4e9f12d1bbcacc45e7a15c800240
Author: Scott Chacon <schacon@gmail.com>
```

```
    modified index to create refs/heads if it is not there
```

```
commit 9764edd90cf9a423c9698a2f1e814f16f0111238
Author: Hans Engel <engel@engel.uk.to>
```

```
    Add diff-lcs dependency
```

其他选项，如下：

```
$ git log --pretty=(oneline | short | medium | full | fuller | email | raw)
```

还可采用如下方式：

```
$ git log --pretty=format:'%h was %an, %ar, message: %s'
```

```
a6b444f was Scott Chacon, 5 days ago, message: dammit, this is the second time this has re
49d77f7 was Scott Chacon, 8 days ago, message: modified index to create refs/heads if it i
9764edd was Hans Engel, 11 days ago, message: Add diff-lcs dependency
e1ba1e3 was Hans Engel, 11 days ago, message: Add dependency for Open4
0f87b4d was Scott Chacon, 12 days ago, message: merged recent changes:
```

```
$ git log --pretty=format:'%h : %s' --graph
```

```
* 2d3acf9 : ignore errors from SIGCHLD on trap
*   5e3ee11 : Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 : Added a method for getting the current branch.
* | 30e367c : timeout code and tests
* | 5a09431 : add timeout protection to grit
* | e1193f8 : support for heads with slashes in them
|/
* d6016bc : require time for xmlschema
```

2.5.3 排序显示日志

日志显示的顺序默认为按日期降序。但也可指定：

- '--topo-order'：按照拓扑排序显示（后代 commits 先于祖先 commits 显示）。
- '--date-order'：效果同上一选项，但在平等 commits 间按照日期降序排列显示。
- '--reverse'：逆序显示。

2.6 比较 commits——git diff

```
# 比较 master 和 test 分支的 tip 之间的不同。
```

```
$ git diff master..test
```

```
# 比较 master 和 test 分支的共同祖先，与 test 的 tip 之间的不同。
```

```
$ git diff master...test
```

```
# 比较工作目录与索引的不同
```

```
$ git diff
```

```
# 比较索引与当前分支最近一次提交的不同
```

```
$ git diff --cached
```

```
# 比较工作目录与当前分支最近一次提交的不同
```

```
$ git diff HEAD
```

```
# 比较工作目录与另一分支不同之处
```

```
$ git diff test
```

```
# 通过加入路径限制的参数，指定比较特定文件或者目录的不同
```

```
# 下面命令将显示工作目录与当前分支最近一次提交，lib 子目录下的不同。
```

```
$ git diff HEAD -- ./lib
```

```
# 统计地显示不同
```

```
$>git diff --stat
```

```
layout/book_index_template.html      |  8 ++-
text/05_Installing_Git/0_Source.markdown | 14 ++++++
text/05_Installing_Git/1_Linux.markdown  | 17 +++++++
text/05_Installing_Git/2_Mac_104.markdown | 11 +++++
text/05_Installing_Git/3_Mac_105.markdown |  8 ++++
text/05_Installing_Git/4_Windows.markdown |  7 +++
.../1_Getting_a_Git_Repo.markdown       |  7 +++-
.../0_ Comparing_Commits_Git_Diff.markdown | 45 ++++++-----
.../0_ Hosting_Git_gitweb_repoorc_github.markdown |  4 +-

```

9 files changed, 115 insertions(+), 6 deletions(-)

2.7 分布式的工作流程

假设在/home/alice/project 是 Alice 一个用 GIT 管理的项目。

同机另一用户 Bob，通过

```
$ git clone /home/alice/project myrepo
```

获得了 Alice 项目仓库的一个拷贝（在目录 myrepo 中）。接下来，Bob 进行一些修改，然后提交到（他本地的）仓库中。

```
(edit files)
$ git commit -a
(repeat as necessary)
```

当 Bob 的修改完成后，他告诉 Alice，从他的仓库中，获取并合并他的修改。Alice 运行下述命令：

```
$ cd /home/alice/project
# 合并 Bob 的 master 分支中的变更到 Alice 的当前分支
$ git pull /home/bob/myrepo master
```

“pull”命令执行两个操作：

- 获取远程分支（git fetch）。
- 获取的把远程分支的内容与当前分支合并（git merge）。

为了避免每次都输入仓库的路径，使用下述命令命名远程仓库，之后使用仓库名来引用仓库：

```
$ git remote add bob /home/bob/myrepo
```

这样，上述的 “pull”命令等效于：

```
# 获取 bob 仓库中激活分支，到本地的（远程仓库的）缓冲分支中（.git/refs/remotes/bob/master）
$ git fetch bob
# 查看本地分支 master 与 bob 仓库的 master 分支的不同之处
$ git log -p master..bob/master
# 合并 bob 仓库的 master 到当前分支
$ git merge bob/master
```

上面命令还可这样写：

```
# remotes/bob 下缓冲了 bob 仓库的分支
$ git pull . remotes/bob/master
```

💡 git pull 命令：用给定参数调用 git fetch，并且调用 git merge 把取回的分支（可能不只一个）与**当前分支合并**。一般的用法为“git pull <仓库> 远程分支名”。

💡 当带有参数--rebase，则调用 git rebase 代替 git merge 操作。

💡 当使用“git pull <仓库> src:dst ...”格式时，其中仓库是远程仓库名或路径，src 是源，即远程仓库的分支名。

除了上面所述与当前分支合并，还进行如下操作：

当 dst 为非空字符串时，则与本地名为 dst 的分支进行 Fast forward 形式的合并。当 src → dst 不能构成 Fast forward 形式的合并时，可在 src:dst 之前添加“+”，来强制进行更新。

之后 Bob 可以合并 Alices 的最新变更到他仓库中：

```
$ git pull
```

注意到 Bob 不要指定仓库名、与待合并来的远程分支名，这是因为在 .git/config 文件中有记录：

```
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

仓库 origin 的路径在这个文件中记录：

```
$ git config --get remote.origin.url
/home/alice/project
```

若 Bob 之后工作于另一不同的主机上，他和 Alice 可以通过 ssh 来进行 clone 与 pull。例如：

```
$ git clone alice.org:/home/alice/project myrepo
```

2.7.1 公开git 仓库

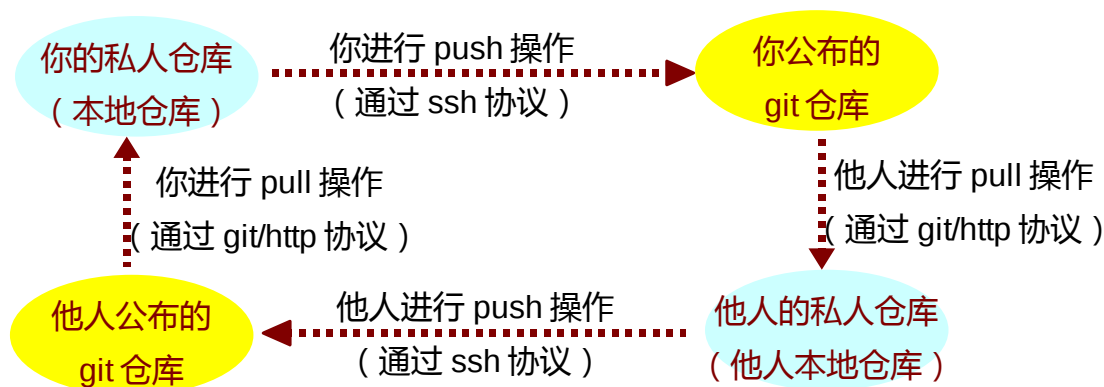


图 4：各个仓库间的“同步”流程

git/http 协议使得使用者只能下载不能上传内容，而 ssh 与 rsync 则支持双向传输。

```
# 把本地仓库的内容 “上推” 到公开的 git 仓库中的远程分支（可以不存在）
# git push <仓库> <本地分支>:<远程分支>
$ git push ssh://yourserver.com/~you/proj.git master:master
```

或者使用

```
# 上推到公开的 git 仓库中的，同名的远程分支
$ git push ssh://yourserver.com/~you/proj.git master
```

💡 如何删除公开的 git 仓库中的远程分支？

`git push <公开的 git 仓库> :<远程分支>`

💡 如同 git fetch，git push 时，若远程分支不能以 [Fast-forwards](#) 形式合并” 上推 “的内容，则 GIT 会” 抱怨 “的。

如同 git fetch/pull，也可通过命名远程的仓库来，避免输入远程仓库的 URL：

```
git remote add myrepo_pub ssh://yourserver.com/~you/proj.git
# 或者编辑配置文件
$ cat >>.git/config <<EOF
[remote "public-repo"]
  url = ssh://yourserver.com/~you/proj.git
EOF
```

2.7.2 push 失败了怎么办？

如果 push 不能导致远程分支以 [Fast-forwards](#) 形式合并” 上推 “的内容时，” 上推 “会失败，并显示如下错误：

```
error: remote 'refs/heads/master' is not an ancestor of local 'refs/heads/master'.
Maybe you are not up-to-date and need to pull first?
error: failed to push to 'ssh://yourserver.com/~you/proj.git'
```

导致发生上面错误的原因有：

- 使用” git reset --hard”，移除了已经发布的 commits。
- 使用 “git commit --amend”，代替了已经发布的 commits。
- 使用 “git rebase”来 rebase 任何已经发布的 commits。

可以通过在分支名前加 “+”来强制 “上推”：

```
$ git push ssh://yourserver.com/~you/proj.git +master
```

作为一个惯例，当公开的仓库中的分支头发生修改，该修改应该总是指向分支头前次所指 commit 的后代。而上述的，发生错误而强制 “上推” 的方式破化了该惯例。不管怎样，为了简单地发布开发中的一系列补丁，实践中常常是这么作的。只要你警告其他开发者你打

算这么作，这是个可以接受的方案。

另外，当不只一人可以对同一仓库“上推”时，也可能发生上推失败的情形。此时，正确的做法是先更新你的工作目录（`git pull` 或 `git fetch + git rebase`），然后再次尝试“上推”。

2.8 git tag

2.8.1 轻量级的tag

```
# git tag <tag 名> <git 对象>
$ git tag stable-1 1b2e1d63ff
```

轻量级的tag 仅仅是一个指向git对象的“指针”，并不创建tag对象（因此不能加入关于该tag的日志、对其进行数字签名）。

2.8.2 Tag 对象

若上述命令中，加入了-a，-s 或者-u <密钥对 ID>，则创建一个tag对象，并且会启动一个编辑器来输入日志信息（可以用参数-F <file>或者-m "日志信息"来避免交互模式）。

Tag对象会被加入GIT对象数据库，.git/refs/tags 下会新增一个指向该tag对象的tag“指针”。

可以用tag对象标记任何对象，但通常标记commit对象（在Linux内核源代码中，第一个tag对象标记一个tree对象，而非commit对象）。

2.8.3 签名的标签 (tags)

若你有一个gpg的密钥（关于gpg的用法，参见附录，[gpg的使用](#)），则可以创建签名的标签（signed tags）。首先，需要在.git/config 或者~/.gitconfig 文件中配置你的密钥ID。

```
[user]
  signingkey = <gpg-key-id>
```

或者使用下面命令：

```
$ git config [--global] user.signingkey <gpg-key-id>
```

之后加上替代“-a”参数为“-s”参数

```
$ git tag -s stable-1 1b2e1d63ff
```

还可在运行git tag时指定：

```
$ git tag -u <gpg-key-id> stable-1 1b2e1d63ff
```

第三章 中级用法

3.1 忽略（不追踪）文件

项目经常可能产生一些不需要 GIT 来追踪的文件，这些文件通常是由“构建（build）”生成的文件（例如编译产生的.o 文件），或编辑器产生的临时备份文件。

当然，只要不对这些文件使用“git add”命令，这些文件就不会被 GIT 追踪。但这样作有时很烦：

- 在项目根目录下，不能用调用“git add .”来添加文件（因为 git add 是递归添加的，这样 GIT 会追踪根目录及其子目录下的所有文件，包括不需要追踪的文件）
- 使用“git commit -a”和“git status”老是提示存在未追踪的文件。

可以告诉 git 忽略某些文件，只要在项目根目录下创建名为.gitignore 的文件，其内容例子如下：

```
# Lines starting with '#' are considered comments.
# Ignore any file named foo.txt.
foo.txt
# Ignore (generated) html files,
*.html
# except foo.html which is maintained by hand.
!foo.html
# Ignore objects and archives.
*.[oa]
```

.gitignore 也可以放置到工作目录的其他目录下。某目录下的.gitignore 只会对该目录及该目录下的文件夹生效。

💡 需要把.gitignore 文件纳入 GIT 的追踪中：git add .gitignore; git commit。

💡 对于不小心把不需要追踪的文件，纳入 GIT 的追踪。gitignore 不会生效，此时使用：

git rm --cached <不需要追踪但已纳入 GIT 追踪的文件>

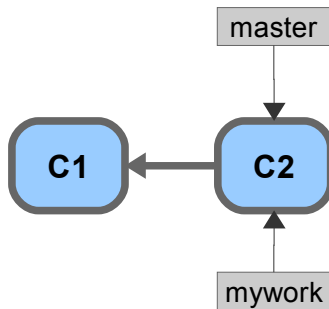
若需要上述排除追踪的文件，只在某个仓库生效（而不是某个项目的所有仓库³），可以在.git/info/exclude 文件或配置文件的 core.exclude 变量中设置排除追踪的文件（而不需要在.gitignore 文件中指明）

3.2 REBASING

假设在本地创建一支名为“mywork”的追踪远程分支“master”。（这样的分支称为 remote-tracking 分支）

3 GIT 是分布式版本管理系统，通常一个项目会有许多仓库。

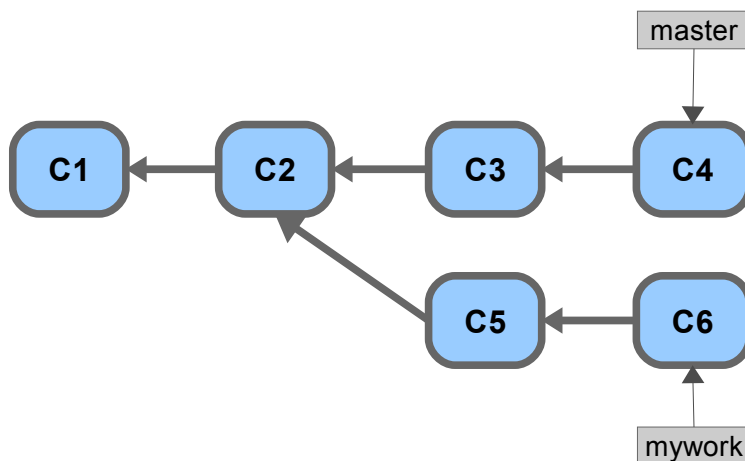
```
$ git checkout -b mywork origin/master
```



接下来，你进行了一些工作，创建了两个新的 commits：

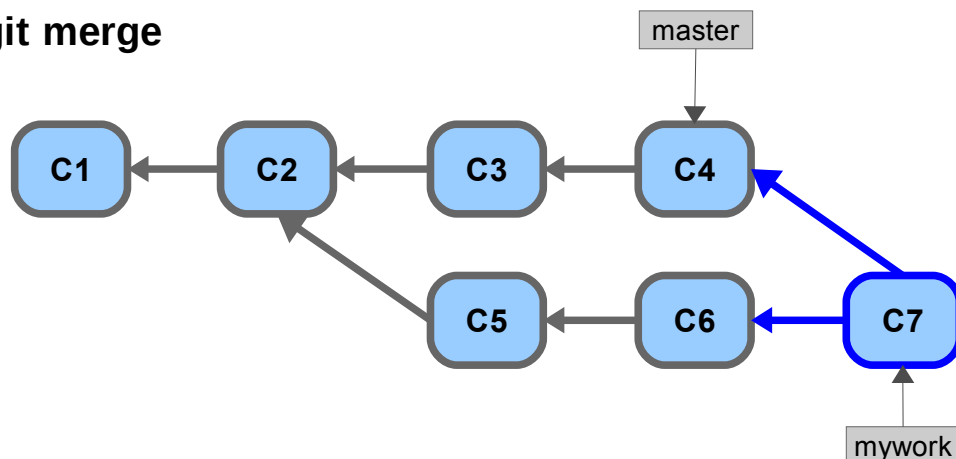
```
$ vi file.txt  
$ git commit  
$ vi otherfile.txt  
$ git commit
```

此时，他人也对远程 master 分支进行了一些工作，创建了两个新的 commits。这样，mywork 和远程 master 的开发线就分叉了。



此时，你需要使用 “pull”来合并他人的修改，其结果是创建了一个新的**合并 commit**：

git merge



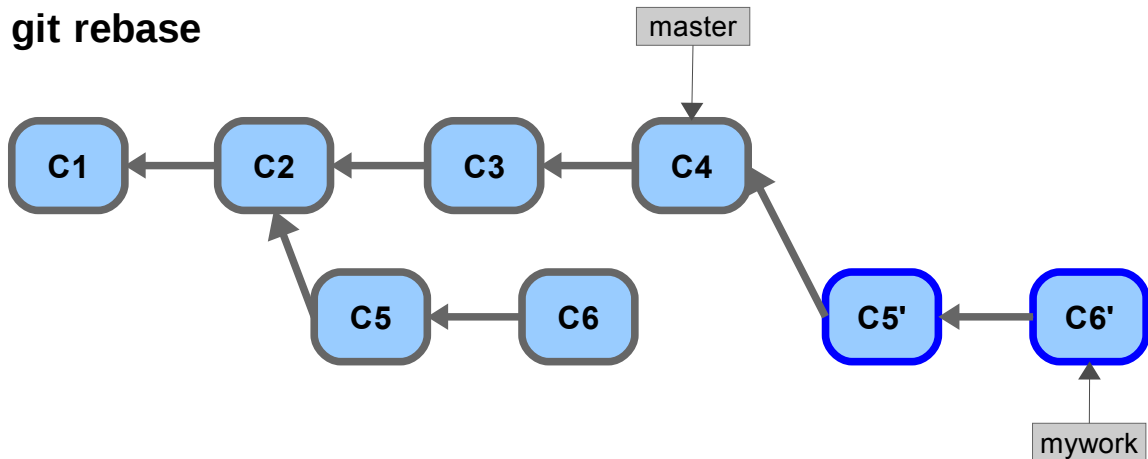
然而，若想保持 mywork 分支的历史为一系列没有合并的 commits，你可以选择 git

rebase :

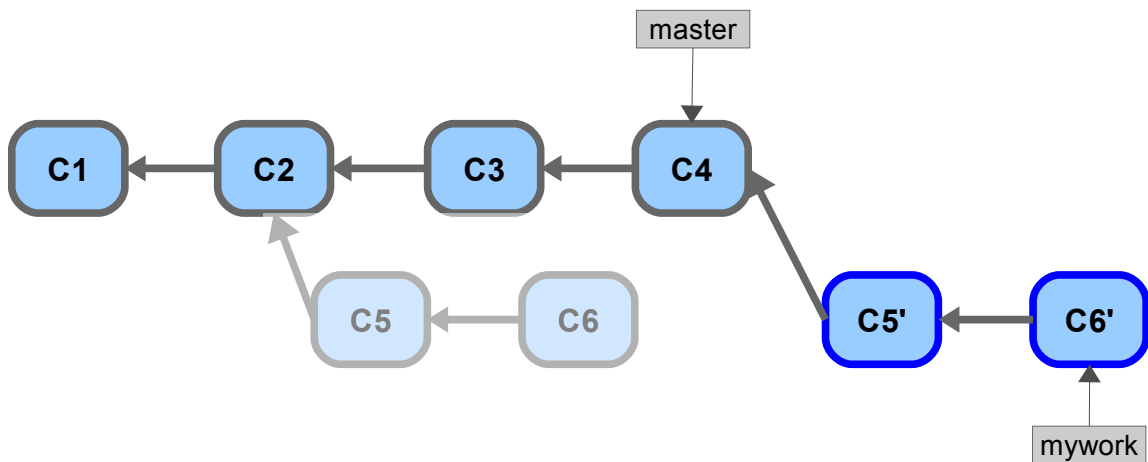
```
$ git checkout mywork  
$ git rebase origin/master
```

上述操作将会移除你在 mywork 的**每个** commits，临时保存到.git/rebase（以补丁的形式）。然后改变 mywork 分支头，使之指向远程 master 分支头所指。最后在把保存的补丁打到“新”的 mywork 分支上。

git rebase

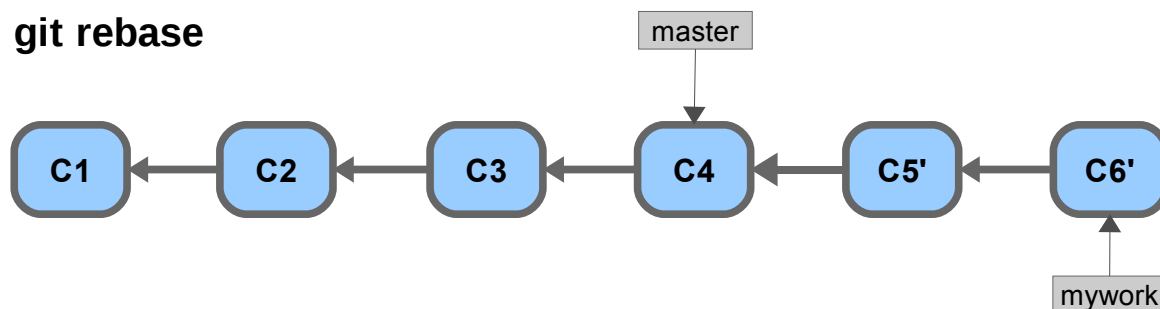


一旦分支头指向新创建的 commit 对象，那些旧的 commits 将被抛弃（即图中的 C5、C6）。例如，你运行垃圾收集时（见 git gc）。

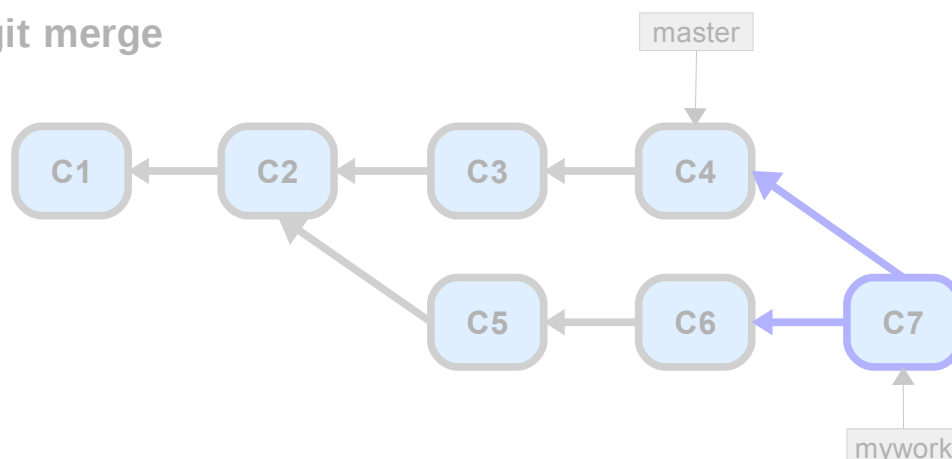


比较下使用合并与 rebase，其历史信息的区别：

git rebase



git merge



在 rebase 的过程中，可能会产生冲突。这时，rebase 过程会停止来让你解决冲突。解决冲突后，运行 “git add”来更新冲突内容的索引/，然后运行：

```
$ git rebase --continue #而不是运行 “git commit”
```

GIT 将会继续打剩余的补丁。

在任何时刻，使用：

```
$ git rebase --abort
```

来终止 rebase 过程并把 mywork **重置到 rebase 过程开始前的状态**。

3.3 交互式 REBASING

通过交互式的 REBASING，可以在“上推”到公开仓库前⁴，重写你自己的 commit（分割、合并、重新排序和去掉部分 commits）。

若你有一些 commits（在 rebase 过程中，被从分支中移出，暂存为了补丁），想在 rebase 过程中加以修改，可以使用交互模式的 rebase：

```
$ git rebase -i origin/master
```

这时，GIT 会带你进入一个编辑器，编辑器中列出将要处理的 commits（这是一个

4 重写已经公开的 commits 会造成一些文件，参见 [push 失败了怎么办](#)

TODO 列表) :

```
pick fc62e55 added file_size
pick 9824bf4 fixed little thing
pick 21d80a5 added number to log
pick 76b9da6 added the apply command
pick c264051 Revert "added file_size" - not implemented correctly
# Rebase f408319..b04dc3d onto f408319
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

上述列出的 5 个待处理的 commits (每行一个) , 以下述格式显示 :

(处理方式) (部分 SHA1 码) (提交日志的简短描述)

你可以编辑各行 commits :

- 更改处理方式由默认的 pick (直接尝试应用补丁 , 应用完后提交一下 , 保持原有提交日志) , 改为 edit (编辑 commit , --amend) 或者 squash (与之前的 commit 融合为一)
- 可以更改各行顺序 , 来改变这些 commit 的顺序。
- 移出某行来移出对应的 commit。

保存退出编辑器后 , GIT 将会根据刚才编辑的内容 (TODO 列表) 来应用 commits。

这里再说明下 “edit” (编辑 commit) 的行为。当 GIT 读到上述 TODO 列表中某行 , 指定用 “edit”来处理 commit , 则暂停 , 然后进入一个命令行来编辑该 commit。例如 , 你想编辑某个修改了 file1 与 file2 的 commit , 把它分成两个 commit (每个 commit 修改一个文件) , 则在命令行中输入 :

```
$ git reset HEAD^
$ git add file1
$ git commit 'first part of split commit'
$ git add file2
$ git commit 'second part of split commit'
#继续处理下一个 commit
$ git rebase --continue
```

3.4 交互式的 git add

```
$ 'git add -i'.
```

GIT 会显示所有修改过的文件及其状态：

	staged	unstaged	path
1:	unchanged	+4/-0	assets/stylesheets/style.css
2:	unchanged	+23/-11	layout/book_index_template.html
3:	unchanged	+7/-7	layout/chapter_template.html
4:	unchanged	+3/-3	script/pdf.rb

*** Commands ***

1: status	2: update	3: revert	4: add untracked
5: patch	6: diff	7: quit	8: help

What now>

可见有 4 个修改的文件，其更改没有加入到索引中（unstaged）。

*** Commands ***下显示有可进入的模式，例如：

- 若要加入某文件的更改到索引中（stage 此文件），可以输入 “2”或 “u”，进入升级模式：然后后输入想要 stage（把更改加入索引）的文件的编号（或编号范围）。

What now> 2

	staged	unstaged	path
1:	unchanged	+4/-0	assets/stylesheets/style.css
2:	unchanged	+23/-11	layout/book_index_template.html
3:	unchanged	+7/-7	layout/chapter_template.html
4:	unchanged	+3/-3	script/pdf.rb

Update>> 1-3

	staged	unstaged	path
*1:	unchanged	+4/-0	assets/stylesheets/style.css
*2:	unchanged	+23/-11	layout/book_index_template.html
*3:	unchanged	+7/-7	layout/chapter_template.html
4:	unchanged	+3/-3	script/pdf.rb

Update>>

此时按下回车键，回到主界面，可见文件的状态已经改变了：

What now> status

	staged	unstaged	path
1:	+4/-0	nothing	assets/stylesheets/style.css
2:	+23/-11	nothing	layout/book_index_template.html
3:	+7/-7	nothing	layout/chapter_template.html
4:	unchanged	+3/-3	script/pdf.rb

- 若想加入未追踪的文件，输入 4。
- 若想查看索引和当前分支头的差异，输入 6。

- 输入 5 或 'p'，进入一个很酷的模式：输入要待处理的文件编号，回车，之后再按一个回车。git 将会显示一个个补丁，询问你如何 stage⁵（按 “?” 可以查看各个选项的含义）。这样，你可以一次 stage 文件的**部分**修改。例如，stage 第四个文件的部分修改之后：

```

      staged  unstaged  path
1:      +4/-0   nothing  assets/stylesheets/style.css
2:    +23/-10   nothing  layout/book_index_template.html
3:      +7/-7   nothing  layout/chapter_template.html
4:      +1/-1    +2/-2  script/pdf.rb

```

- 若想，回复索引/到当前分支头所指状态，则输入 3（revert）。
- 输入 7（quit）退出交互模式。之后运行 “git commit”（不要运行 “git commit -a”，不然所有更改都会被 stage 到索引中，再把索引提交——之前的工作白费了）来把索引提交。

3.5 STASHING（暂存工作目录中的未提交的修改）

当在做某些复杂的工作中，又发现了一个与当前工作**无关**但是个明显的小 bug，你也许想暂停手头的工作，来修正这个 bug。可以使用 “git stash” 来保存工作目录和索引的当前状态，然后修正这个 bug（或者 “git stash” 后切换到另一分支进行除虫工作，然后切回），最后运行 unstash 来回到先前的复杂工作中。

```
$ git stash save "在实现 xxx 特性的工作中"
```

上个命令将会保存工作目录和索引的当前状态到某处，然后设置**工作目录和索引**到当前分支头所指（通常是最近一次提交）的状态。

完成除虫的修改后，提交之：

```
$ git commit -a -m "修正 bug xxxx"
```

之后，返回先前的复杂工作中：

```
# 在当前工作目录基础上，加入其最近一次保存的内容
$ git stash apply
```

3.5.1 Stash 队列

```
# 列出 Stash 队列
$ git stash list
```

```
stash@{0}: WIP on book: 51bea1d... fixed images
stash@{1}: WIP on master: 9705ae6... changed the browse code to the official repo
```

5 虽然按下 “?” 会有很多选项的解释，但只有部分选项在当前语境下是可用的（即提示的选项）

然后，可以独立应用其中某个，例如：

```
$ git stash apply stash@{1}
```

下述命令清空 Stash 队列

```
$ git stash clear
```

3.6 追踪分支

追踪分支是指某个本地的分支关联一个远程的分支（这种关联记录在.git/config中）。当 push（“上推”）或 pull 那个分支，会与远程关联分支自动进行 push 或者 pull。

“git clone”命令会自动设置本地的 “master”分支追踪远程的 “master”分支（origin/master）。

```
# 加入 “--track”参数，创建远程 experiment 分支的追踪分支（本地的 experiment）  
$ git branch --track experimental origin/experimental
```

接下来，只要简单地运行（一般的 git pull 命令，需要带有指示仓库、远程分支等的参数）

```
$ git pull experimental
```

就会自动从 origin 仓库取 experimental 分支，并合并本地的 experimental 分支上。

同样的，push 也可如上简写。

3.7 使用 git grep 进行查找

```
$ git grep mmap
```

```
config.c:          contents = mmap(NULL, contents_sz, PROT_READ,  
diff.c:          s->data = mmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd, 0);  
git-compat-util.h:extern void *mmap(void *start, size_t length, int prot, int fla  
read-cache.c:      mmap = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_PRIVATE,  
refs.c: log_mapped = mmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);  
sha1_file.c:      map = mmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);  
sha1_file.c:      idx_map = mmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd, 0);  
sha1_file.c:          win->base = mmap(NULL, win->len,  
sha1_file.c:          map = mmap(NULL, *size, PROT_READ, MAP_PRIVATE, f  
sha1_file.c:          buf = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);  
wrapper.c:void *mmap(void *start, size_t length,
```

显示匹配内容的行号，使用 “-n”参数：

```
$>git grep -n mmap
```

```
config.c:1016:          contents = mmap(NULL, contents_sz, PROT_READ,  
diff.c:1833:          s->data = mmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,  
git-compat-util.h:291:extern void *mmap(void *start, size_t length, int prot, int
```

```

read-cache.c:1178:      mmap = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE, MAP_
refs.c:1345:      log_mapped = mmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, logfd, 0);
sha1_file.c:377:      map = mmap(NULL, mapsz, PROT_READ, MAP_PRIVATE, fd, 0);
sha1_file.c:479:      idx_map = mmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
sha1_file.c:780:      win->base = mmap(NULL, win->len,
sha1_file.c:1076:      map = mmap(NULL, *size, PROT_READ, MAP_PR
sha1_file.c:2393:      buf = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd
wrapper.c:89:void *mmap(void *start, size_t length,

```

若只对文件名感兴趣，传入 “--name-only”参数：

```
$ git grep --name-only mmap
```

```

config.c
diff.c
git-compat-util.h
read-cache.c
refs.c
sha1_file.c
wrapper.c

```

“-c”参数，计数匹配的行：

```
$ git grep -c mmap
```

```

config.c:1
diff.c:1
git-compat-util.h:1
read-cache.c:1
refs.c:1
sha1_file.c:5
wrapper.c:1

```

在指定的版本中查找，指定一个指示版本的参数：

```
$ git grep mmap v1.5.0
```

```

v1.5.0:config.c:      contents = mmap(NULL, st.st_size, PROT_READ,
v1.5.0:diff.c:      s->data = mmap(NULL, s->size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:git-compat-util.h:static inline void *mmap(void *start, size_t length,
v1.5.0:read-cache.c:      cache_mmap = mmap(NULL, cache_mmap_size,
v1.5.0:refs.c: log_mapped = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, logfd
v1.5.0:sha1_file.c:      map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:      idx_map = mmap(NULL, idx_size, PROT_READ, MAP_PRIVATE, fd
v1.5.0:sha1_file.c:      win->base = mmap(NULL, win->len,
v1.5.0:sha1_file.c:      map = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
v1.5.0:sha1_file.c:      buf = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd

```

可以看到当前内容和 v1.5.0 版本所示内容的相异之处，比如 v1.5.0 中，mmap 未在 wrapper.c 中出现。

还可以组合匹配的条件，例如想要搜索名为 SORT_DIRENT 的宏定义：

```
$ git grep -e '#define' --and -e SORT_DIRENT
```

```
builtin-fsck.c:#define SORT_DIRENT 0  
builtin-fsck.c:#define SORT_DIRENT 1
```

或者搜索所有，宏定义标记或有 SORT_DIRENT 的：

```
$ git grep --all-match -e '#define' -e SORT_DIRENT
```

```
builtin-fsck.c:#define REACHABLE 0x0001  
builtin-fsck.c:#define SEEN 0x0002  
builtin-fsck.c:#define ERROR_OBJECT 01  
builtin-fsck.c:#define ERROR_REACHABLE 02  
builtin-fsck.c:#define SORT_DIRENT 0  
builtin-fsck.c:#define DIRENT_SORT_HINT(de) 0  
builtin-fsck.c:#define SORT_DIRENT 1  
builtin-fsck.c:#define DIRENT_SORT_HINT(de) ((de)->d_ino)  
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)  
builtin-fsck.c: if (SORT_DIRENT)
```

也可以搜索必须匹配某个，但至少匹配另两个中的一个，例如查找宏定义 PATH 或 MAX：

```
$ git grep -e '#define' --and \( -e PATH -e MAX \)
```

```
abspath.c:#define MAXDEPTH 5  
builtin-blame.c:#define MORE_THAN_ONE_PATH (1u<<13)  
builtin-blame.c:#define MAXSG 16  
builtin-describe.c:#define MAX_TAGS (FLAG_BITS - 1)  
builtin-fetch-pack.c:#define MAX_IN_VAIN 256  
builtin-fsck.c:#define MAX_SHA1_ENTRIES (1024)  
...
```

3.8 GIT 中的撤销的功能——reset，checkout 和 revert

GIT 提供了许多手段，来修正开发中的失误。具体选择何种方式，取决与是否提交（commit）了该失误；若提交了，是否与他人共享了这个有错误的 commit（即是否 push 到了公开的服务器上）。

3.8.1 修正某个未提交的失误

若弄乱了工作目录，但尚未提交。可以设置工作目录的状态到分支头所指（通常是最近一次提交）的状态：

```
$ git reset --hard HEAD
```

若仅要恢复单个文件，例如 hello.rb，只要使用 git checkout 即可：

```
# 恢复 hello.rb 为索引中的版本，之后 "git diff hello.rb"会返回无差别。  
$ git checkout -- hello.rb
```



```
# 恢复 hello.rb 为 HEAD 所指的修订版本，  
# 之后 "git diff hello.rb"和 "git diff --cached hello.rb"返回无差别。  
$ git checkout HEAD hello.rb
```

3.8.2 修正已提交了的错误

若创建了一个 commit，之后发现其存在错误。有两个根本上差异很大的方式来修正：

1. 再创建一个新的 commit，来撤销错误的 commit。这个方式适用于已公开错误 commit 的情形下。

```
# 创建一个新的 commit，来撤销 HEAD 所指的改变。  
# GIT 会启动编辑器，来让你输入新的 commit 的日志。  
# 可能会产生合并冲突  
$ git revert HEAD^
```

2. 直接修改错误的 commit。注意：若已经公开了错误的 commit，则不要这么做。

➔ 若是发现刚刚提交的 commit 有错误，使用

```
# 根据当前索引的内容创建 commit，替代 HEAD 所指的 commit  
# 可以添加忘记添加的文件或者修改提交日志  
$ git commit --amend
```

➔ 若非上面的情形，使用：

```
# 在交互模式的 rebase 中修正  
$ git rebase -i <错误 commit>^
```

3.9 维护 GIT

3.9.1 维护 GIT 的高效率

大型仓库中，GIT 通过压缩的方式，避免占用过多的磁盘/内存空间。这个压缩过程不是自动进行的，需要手工运行：

```
# 可能会运行很长时间，每隔一定时间运行此命令  
$ git gc
```

💡 GIT 一般是稀疏存储的，即在 .git/objects 文件夹下，每个以 40 位 16 进制 SHA1 码的前两位（一个字节）命名的子文件夹下。运行 git gc 后会打包文件到 .git/objects/pack 文件夹下，后缀为 pack 和 idx（索引文件，用于加快打包文件访问）的文件。

💡 git gc 还会对打包文件进行差异存储，相似的对象存储为，“**基对象**（其共同部分）的引用” + “差异部分”。

💡 git gc 还会清理，不能从当前有效引用达到的对象（称为 dangling 对象）。这正是

“gc” (garbage collection) 的字面含义。

3.9.2 确保可靠性

```
# 仓库一致性检查，可能会运行一些时间。通常警告 dangling 对象  
$ git fsck
```

3.10 设置一个公开的仓库

参见附录，[gitosis 的使用](#)、[运行 git-daemon](#)、[配置 gitweb](#)

注意以 HTTP 方式设置公开仓库，需要给予 post-update 以可执行权限

```
chmod a+x .git/hooks/post-update
```

第四章 高级用法

4.1 创建新的空分支（无祖先分支）

```
$ git symbolic-ref HEAD refs/heads/newbranch
$ rm .git/index
$ git clean -fdx
    <do work>
$ git add your files
$ git commit -m 'Initial commit'
```

4.2 修改历史

交互式的 git rebase 是一个修改少数几个 commits 的好办法。

git filter-branch 是修改全体的 commits 的好办法。

4.3 分支与合并的高级用法

4.3.1 合并时获取解决冲突的帮助

合并后遇到冲突时，所有能自动合并的变更都加到索引中了，故 git diff 仅显示冲突的内容：

```
$ git diff

diff --cc file.txt
index 802992c,2b60207..0000000
--- a/file.txt
+++ b/file.txt
@@@ -1,1 -1,1 +1,5 @@@
++<<<<<< HEAD:file.txt
+Hello world
+=====
+Goodbye
++>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:file.txt
```

上述输出结果，差异显示有两列，第一列显示的工作目录中的 file.txt 与第一父 commit 的差异。第二列线是的是工作目录中的 file.txt 与第二父 commit 的差异。

冲突解决后提交的合并 commit，有两个父 commits（假设两路合并），一个是 HEAD（当前分支头所指的）；另一个其他分支的分支头，暂存在 **MERGE_HEAD** 中。

合并过程中，索引持有每个文件的三个版本：

```
$ git show :1:file.txt    # 两个分支共同祖先中的 file.txt
$ git show :2:file.txt    # HEAD 中的 file.txt
$ git show :3:file.txt    # MERGE_HEAD 中的 file.txt
```

```
$ git diff -1 file.txt      # 与共同祖先中的 file.txt 的差异
$ git diff --base file.txt  # 同上
$ git diff -2 file.txt      # 与 HEAD 中的 file.txt 的差异
$ git diff --ours file.txt  # 同上
$ git diff -3 file.txt      # 与 MERGE_HEAD 中的 file.txt 的差异
$ git diff --theirs file.txt # 同上
```

git log 和 gitk 命令，关于合并的选项：

```
# 显示只在 HEAD 或者 MERGE_HEAD 中，触及未合并文件的 commits。
$ git log --merge
$ gitk --merge
```

4.3.2 多路合并

可以一次合并多个分支，例如

```
$ git merge scott/master rick/master tom/master
```

等同于：

```
$ git merge scott/master
$ git merge rick/master
$ git merge tom/master
```

4.3.3 Subtree (合并他人的分支到子目录下)

```
# “-f”参数确保远程仓库信息设置完成后，立刻进行 git fetch
$ git remote add -f Bproject /path/to/B

$ git merge -s ours --no-commit Bproject/master
$ git read-tree --prefix=dir-B/ -u Bproject/master
$ git commit -m "Merge B project as our subdirectory"
$ git pull -s subtree Bproject master
```

4.4 SUBMODULES (把他人的项目纳为你的项目的一个子模块)

大型的项目往往是由许多独立的小模块（即子项目）组成。例如一个 Linux 的发行版的源代码。

在“中央式”的版本管理系统中，通常采用的是单个仓库中包含每个模块的方式。开发者可以检出（checkout）他们所需要的模块进行工作。可以在单个 commit 中，修改许多个模块中的文件（例如移动文件、升级 API 和翻译）。


GIT 的 submodule，支持仓库在子目录中，含有外部项目的一个检出。Submodule 仅保存子模块的仓库位置和 commit ID。之后可以让 GIT 仅克隆感兴趣的子模块项目。

实践下 submodule，首先创建四个例子仓库（之后作为四个子项目）：

```
$ mkdir ~/git
$ cd ~/git
$ for i in a b c d
do
    mkdir $i
    cd $i
    git init
    echo "module $i" > $i.txt
    git add $i.txt
    git commit -m "Initial commit, submodule $i"
    cd ..
done
```

接下来创建总项目，加入上述项目为子模块：

```
$ mkdir super
$ cd super
$ git init
$ for i in a b c d
do
    git submodule add ~/git/$i
done
```

 注意：若想发布总项目，不要使用本地 URLs

查看下 submodule 作了哪些工作：

```
$ ls -a
. .. .git .gitmodules a b c d
```

git submodule add 命令作了一些工作：

- 在当前目录下，克隆子模块（默认检出 master 分支）
- 添加子模块的路径到.gitmodules 文件中，并把该文件加入索引。
- 添加子模块的当前 commit Id 到索引中。

在总项目中提交：

```
$ git commit -m "Add submodules a, b, c and d."
```

接下来试着克隆总项目：

```
$ cd ..
$ git clone super cloned
$ cd cloned
```

子模块的文件夹存在，但其内容为空：

```
$ ls -a a
```

. ..

```
$ git submodule status
```

```
-d266b9873ad50488163457f025db7cdd9683d88b      a
-e81d457da15309b4fef4249aba9b50187999670d      b
-c1536a972b9affea0f16e0680ba87332dc059146      c
-d96249ff5d57de5de093e6baff9e0aafa5276a74      d
```

Pull 子模块：

1. 运行 git submodule init 来把子模块仓库的 URLs 到.git/config：

```
$ git submodule init
```

2. 运行 git submodule update 来克隆仓库，并检出在总项目中指定的 commits。

```
$ git submodule update
```

```
$ cd a
```

```
$ ls -a
```

```
. .. .git a.txt
```

git submodule add 和 git submodule 的不同之处，在与 git submodule update 检出一个特定的 commit 而非分支头所指的 commit（分支的 tip），好像检出一个 tag 一样，故不是在一个分支上工作：

```
$ git branch
```

```
*(no branch)
```

```
master
```

若想对子模块进行修改，你应该创建或者检出一个分支，进行修改，在子模块内发布修改，然后升级总项目的引用到新的 commit：

```
$ git checkout master
```

```
# 或
```

```
$ git checkout -b fix-up
```

```
# 进行修改
```

```
$ echo "adding a line again" >> a.txt
```

```
# 提交修改
```

```
$ git commit -a -m "Updated the submodule from within the superproject."
```

```
$ git push
```

```
$ cd ..
```

```
$ git diff
```

```
diff --git a/a b/a
```

```
index d266b98..261dfac 160000
```

```
--- a/a
```

```
+++ b/a
```

```
@@ -1,1 @@
```

```
-Subproject commit d266b9873ad50488163457f025db7cdd9683d88b
```

```
+Subproject commit 261dfac35cb99d380eb966e102c1197139f7fa24
```

```
$ git add a
$ git commit -m "Updated submodule a."
$ git push
```

若想升级子模块，在 **git pull** 之后，运行 **git submodule update**。

💡 切记：先发布子模块的修改，再发布总项目。不然别人不能克隆仓库。

💡 不要把子模块的分支头倒回到某个在总项目中从来没有记录过的 commit。

💡 没有先检出分支，就在某个子项目中进行/提交了修改，之后运行 **git submodule update** 是不安全的。会没有任何提示被覆写。

4.5 git bisect (定位某个引入问题的 commit)

假设你的项目，v2.6.18 版本工作良好但是 “master” 检出的版本却崩溃了。此时需要定位那次提交造成了此问题。这时 **git bisect** 命令派上用场了：

```
$ git bisect start
$ git bisect good v2.6.18
$ git bisect bad master
```

```
Bisecting: 3537 revisions left to test after this
[65934a9a028b88e83e2b0f8b36618fe503349f8e] BLOCK: Make USB storage depend on SCSI rather than selecting it [try
```

若运行 “**git branch**”，将会看到 **git** 暂时移到了一个新的、名为 **bisect** 的分支上。该分支指向一个从 “master” 可达，但从 v2.6.18 版本不可达的 commit。

编译并且测试是否崩溃，若还是崩溃，则运行：

```
$ git bisect bad
```

```
Bisecting: 1769 revisions left to test after this
[7eff82c8b1511017ae605f0c99ac275a7e21b867] i2c-core: Drop useless bitmaskings
```

检出一个更老的版本，重复上述过程。告诉 **git** 是否检出的还是崩溃。**git bisect** 进行的是折半查找，最后会定位引入该问题的 commit，用 **git show** 查找作者，报告该 bug。

最后，运行：

```
$ git bisect reset
```

返回先前的（运行 **git bisect** 之前的）分支，并且删除临时的 “bisect” 分支。

注意，每次 **git bisect** 检出的版本，是个建议的版本（通过折半查找产生这个 “建议” 的版本），你也可以自由的试一个另外的更可疑的版本。运行：

```
# 运行 gitk，并且标记自动选择的 commit
$ git bisect visualize
```

然后选择某个可疑的 commit，使用 **git reset** 检出：

```
$ git reset --hard fb47ddb2db...
```

接下来测试，重复运行 “git bisect good”或 “git bisect bad”。

4.6 git blame (查找问题)

git blame 用来查找，谁改了某个文件的哪些部分。

```
# 输出每行的最近一次提交的 commit ID、作者和日期
$ git blame sha1_file.c
```

```
...
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 8) */
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 9) #include "cache.h"
1f688557 (Junio C Hamano 2005-06-27 03:35:33 -0700 10) #include "delta.h"
a733cb60 (Linus Torvalds 2005-06-28 14:21:02 -0700 11) #include "pack.h"
8e440259 (Peter Eriksen 2006-04-02 14:44:09 +0200 12) #include "blob.h"
8e440259 (Peter Eriksen 2006-04-02 14:44:09 +0200 13) #include "commit.h"
8e440259 (Peter Eriksen 2006-04-02 14:44:09 +0200 14) #include "tag.h"
8e440259 (Peter Eriksen 2006-04-02 14:44:09 +0200 15) #include "tree.h"
f35a6d3b (Linus Torvalds 2007-04-09 21:20:29 -0700 16) #include "refs.h"
...
```

```
# 指定输出行的范围， “+”表示相对行数
```

```
$ git blame -L 160,+10 sha1_file.c
```

```
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 160)}
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 161)
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 162)/*
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 163) * NOTE! This returns a
790296fd (Jim Meyering 2008-01-03 15:18:07 +0100 164) * careful about using it. Do an
0fcfd160 (Linus Torvalds 2005-04-18 13:04:43 -0700 165) * filename.
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 166) *
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 167) * Also note that this returns
ace1534d (Junio C Hamano 2005-05-07 00:38:04 -0700 168) * SHA1 file can happen from any
d19938ab (Junio C Hamano 2005-05-09 17:57:56 -0700 169) * DB_ENVIRONMENT environment
```

4.7 GIT 和 email

在当前目录中生成一系列补丁，这些补丁包含在当前分支中，但不包含在 origin/HEAD 中。

```
$ git format-patch origin
```

之后导入这些补丁到电子邮件客户端，然后手动发送。若一次有很多需要发送，则可使用 **git send-email** 脚本来自动完成这个过程。

GIT 也提供了一个名为 **git am** (am 代表 “apply mailbox”) 的工具，用来导入电子邮件发送的一系列补丁。只要按序保存所有含补丁的信息到单个 mailbox 文件，例如

“patches.mbox”，然后运行：

```
# “-3”意味这进行合并
$ git am -3 patches.mbox
```

GIT 将会按序应用每个补丁。当发生冲突时，该过程便暂停，来让你修正冲突。修正之后，

运行：

```
# 创建一个 commit，并继续应用剩下的补丁。  
$ git am --resolved
```

4.8 GIT 的钩子

4.8.1 服务端的钩子

接收后，运行脚本\$GIT_DIR/hooks/post-receive。例如该脚本为 bash：

```
#!/bin/sh  
# <oldrev> <newrev> <refname>  
# update a blame tree  
while read oldrev newrev ref  
do  
    echo "STARTING [$oldrev $newrev $ref]"  
    for path in `git diff-tree -r $oldrev..$newrev | awk '{print $6}`  
    do  
        echo "git update-ref refs/blametree/$ref/$path $newrev"  
        `git update-ref refs/blametree/$ref/$path $newrev`  
    done  
done
```

4.8.2 客户端钩子

提交前，运行脚本\$GIT_DIR/hooks/pre-commit，例如提交前自动运行测试。

4.9 定制 GIT

4.9.1 git config

```
# 改变默认的编辑器  
$ git config --global core.editor emacs
```

```
# 添加别名  
$ git config --global alias.last 'cat-file commit HEAD'  
$ git last  
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b  
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735  
author Scott Chacon <schacon@gmail.com> 1220473867 -0700  
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700  
fixed a weird formatting problem
```

```
$ git cat-file commit HEAD
tree c85fbd1996b8e7e5eda1288b56042c0cdb91836b
parent cdc9a0a28173b6ba4aca00eb34f5aabb39980735
author Scott Chacon <schacon@gmail.com> 1220473867 -0700
committer Scott Chacon <schacon@gmail.com> 1220473867 -0700
fixed a weird formatting problem
```

添加彩显，参见所有 git config 文件中，关于 color.* 的选项。

```
$ git config color.branch auto
$ git config color.diff auto
$ git config color.interactive auto
$ git config color.status auto
```

或使用 color.ui 选项设置

```
$ git config color.ui true
```

设置提交的模板

```
$ git config commit.template '/etc/git-commit-template'
```

设置默认日志的显示格式

```
$ git config format.pretty oneline
```

还有些有趣的选项，关于：

- 打包
- gc
- 合并
- 远程仓库
- 分支
- HTTP 传输
- diffs
- 分页
- 空格
- ...

4.10 修复损坏的 GIT 对象

[Recovering Lost Commits Blog Post](#)

[Recovering Corrupted Blobs by Linus](#)

附录

gpg 的使用

- gpg 命令中，参数的顺序重要，不然 gpg 相关命令不能正确解析参数。
- 有一些强大的图形化的前端，如 seahorse。

基本使用

- i. 创建 gpg 公私密钥对

```
gpg --gen-key
```

- ii. 导出你的公钥（用于发布你的公钥）

```
# -a 参数，代表输出为 7bit 的 ASCII 文件而非二进制文件  
gpg --export [gpg user'ID] [-o file] [-a]
```

- iii. 导入（他人发布的）公钥

```
gpg --import [file]
```

密钥管理

- i. 创建吊销证书（调用者要有自己的私钥）——需要创建时输入的密码。因此创建密钥对同时创建一个吊销证书是明智的。但是该吊销证书要妥善保管，以免有人用来吊销你的公钥。

```
gpg --gen-revoke [-o file] <gpg user'ID>
```

- ii. gpg --list-keys（列出公钥）
- iii. gpg --list-sigs（同上，但还列出对公钥的签名）
- iv. gpg --fingerprint（同--list-keys，但还列出公钥的指纹）
- v. gpg --list-secret-keys
- vi. gpg --delete-key <gpg user'ID>（删除公钥）
- vii. gpg --delete-secret-key <gpg user'ID>
- viii. gpg --edit-key <gpg user'ID>（签名别人公钥...）

签名密钥（用于认证公钥）

信任级别：

1 = Don't know（不知道）

2 = I do NOT trust（我不相信）

3 = I trust marginally

4 = I trust fully

加密与解密

加密与签名前会压缩原始文件

■ 加密

```
gpg -r 接收者 -e <待加密的文件>
# 后缀名为 gpg ( gpg 二进制文件的后缀为 gpg , 用 7 位 ascii 编码的文件后缀为 asc )
```

■ 解密

```
gpg -o file -d <待解密的文件>
```

签名与验证

```
gpg -s <待签名的文件>

# 不压缩<待签名的文件>
gpg --clearsign <待签名的文件>

# 签名与原文件分离 , 签名的后缀为 sig
gpg -b 待签名的文件
```

常用的形式 :

```
gpg [-u 发送者] [-r 接收者] -a -s -e <待签名与加密的文件>
```

文件后缀名为 asc (ascii 格式 , 由 -a 参数所指定) 。

```
# 验证签名
gpg --verify 待验证的文件
```

gitosis 的使用

参见:

- <http://scie.nti.st/2007/11/14/hosting-git-repositories-the-easy-and-secure-way>
- <http://eagain.net/gitweb/?p=gitosis.git;a=blob;f=README.rst;hb=master>

安装

```
git clone git://eagain.net/gitosis.git
python setup.py install
```

配置

1. 增加用户 git

```
adduser \  
  --system \  
  --shell /bin/sh \  
  --gecos 'git version control' \  
  --group \  
  --disabled-password \  
  --home /home/git \  
git
```

2. 本地创建管理员公钥,并拷贝到服务器上

```
ssh-keygen -t rsa  
scp $HOME/.ssh/id_rsa.pub root@172.16.2.41:/home/git/
```

3. 服务器上初始化 gitosis

```
sudo -H -u git gitosis-init < /home/git/id_rsa.pub  
# 修正 post-update 权限:使其可执行  
chmod 755 /home/git/repositories/gitosis-admin.git/hooks/post-update
```

4. 本地修改 gitosis 服务器的配置

```
git clone git@172.16.2.41:gitosis-admin.git  
cd gitosis-admin
```

修改配置文件 gitosis.conf

```
[gitosis]  
# repositories = /path/to/repositories  
# 全局禁止访问 git 下的文件  
daemon = no  
gitweb = no  
# DEBUG, INFO, WARNING, ERROR, CRITICAL  
loglevel = DEBUG  
  
[group gitosis-admin]  
# 值为空格分隔的用户密钥文件的主文件名,带后缀 pub 的公钥文件放到 keydir 目录下  
# group 没有 writeable 与 readonly 时仅仅为一方便用户分组管理  
writable = gitosis-admin  
# readonly =  
members = cee1  
  
[group my_proj_participant]  
writable = my_proj
```

```
# readonly =  
members = cee1  
  
[repo my_proj]  
gitweb = yes  
daemon = yes  
owner = cee1  
description =  
  
# [gitweb]  
## Where to make gitweb link to as it's "home location".  
## NOT YET IMPLEMENTED.  
# homelink = http://example.com/
```

完成后提交修改:

```
git commit -a -m "..."  
git push
```

运行 git-daemon

git-daemon 是 git 协议的服务端程序。git 协议和 http 协议均只支持单向（下载）。相比 http 协议，git 协议更加高效，但需要配置防火墙，允许相应端口。

运行 git-daemon，只需要：

```
git-daemon --base-path=/path/to/repositories/
```

这样，git-daemon 会允许/path/to/repositories/下所有仓库（其根目录下要有名为 git-daemon-export-ok 的文件）的匿名 git 协议访问。可以添加到 local 脚本中，以便开机自动启动。

配置 gitweb，允许 web 访问 GIT 仓库

下面是一个具体部署方案。

1. 拷贝 gitweb 相关资源到/var/www/gitweb 文件夹下：

- ✓ /usr/lib/cgi-bin/: gitweb.cgi
- ✓ /usr/share/gitweb/:
 - git-favicon.png
 - git-logo.png
 - gitweb.css
- ✓ /etc/gitweb.conf

2. 修改 gitweb.conf：

```
$projectroot = "/path/to/repositories";
```

#配合 gitosis 使用，不然注释之

```
$projects_list = "/path/to/git_home/gitosis/projects.list";
```

```
@git_base_url_list = ('git://your_server_domainname_or_ip');
```

```
$stylesheet = "files/gitweb.css";
```

```
$logo = "files/git-logo.png";
```

```
$favicon = "files/git-favicon.png";
```

3. 配置 Apache :

```
Alias /files/gitweb.css /var/www/gitweb/gitweb.css
```

```
Alias /files/git-logo.png /var/www/gitweb/git-logo.png
```

```
Alias /files/git-favicon.png /var/www/gitweb/git-favicon.png
```

```
ScriptAlias /git /var/www/gitweb/gitweb.cgi
```

```
<Directory "/var/www/gitweb">
```

```
    SetEnv GITWEB_CONFIG /var/www/gitweb/gitweb.conf
```

```
    Options FollowSymLinks -SymLinksIfOwnerMatch Indexes +MultiViews
```

```
    Order allow,deny
```

```
    Allow from all
```

```
    AllowOverride None
```

```
</Directory>
```

4. 通过 `http://your_server/git` 就可访问 GIT 仓库了。