Git Pocket Guide 阅读笔记

——wonderflow

目录

Git 是什么？

简单来说，git 是一个版本控制（version control）管理工具。但是它的作用并不仅仅是你所理解的程序员用来管理它代码的工具，尽管它最初就是这么被 Linus 这么设计的。但是，所有涉及到文件管理的工作，或者说工程，我觉得都需要用到 git 这么一款工具。
它有如下几个作用：

1、 检查你的工程在前面某个时间点的状态
2、 比较在不同状态时工程有哪些变化
3、 把你的工程开发分隔成不同独立的生产线，这里，我们使用“分支”（branch）这个学名。
4、 周期性的把不同的分支归并（merge）到一起，通常是把两个分支归并，但 git 也支持两个以上的操作。
5、 允许多人同时从事同一个项目的工作，并且在需要的时候随时提交他们的工作。

为什么用 Git？

说到 git，就不得不提它的设计者，同时也是 linux 系统的设计开发者——linus；也不得不提目前风靡全球的 github（https://github.com）。
让我们简单的罗列一下 git 的好处：

1、 Git 的每一个用户都有一个完整的备份，只需要偶尔使用网络连接进行远程的同步。这里每个用户的完整备份我们称之为仓库(repository)。
2、 Git 的分支(branching)和归并(merging)操作效率都很高。
3、 Git 的一次变更提交分为 2 步，第一步，把变更提交到暂存区（stage eara）；第二步，把暂存区的内容提交到 git 的本地仓库(repository)。这样的两步操作有利于选择性的提交变动，而非强制性的一次把所有变动都提交。
4、 由于每人都有一个代码仓库(repository)，所以每个人都可以根据自己喜欢的方式工作，最后通过 git 的远程仓库一起归并到一起，这样的方式相当灵活和自然。
5、 Git 有一个风靡全球的网站，GitHub，上面有数量众多的世界知名的开源项目，学会了 git，将相当于开出了通向这些项目的大门。

手册的使用说明

默认情况下，我们的手册是针对 linux 系统，shell 脚本使用者的，所以我们只保证你使用的 linux，在 shell 脚本上能正确运行我们所提到的命令。

概览 Git

术语说明

Repository：一个 git 的完整项目，我们称之为“repository”，在 gitlab 上，称之为“project”，包括这个项目的所有历史版本。

Commits：一次 git 的提交，就是一个 commit，它会保存一份完整的项目。

Tree：工程目录结构快照，包括文件和文件夹，用以表示项目的完整状态。

Author：一个版本的负责人，包括姓名、邮件地址以及项目最后更改的时间戳。

Commiter：单次提交者，包括与 author 相同类目的信息。

Commit message：本次提交的原因，以及更改内容的说明性文字。

Parent commits：一个 commit 可能存在 0 个或多个 parent commit。如果是初始项目，就是 0 个，如果只是对某一个版本进行一些修改，那就是 1 个，如果是对多个项目的归并(merge)，那么就是多个了。

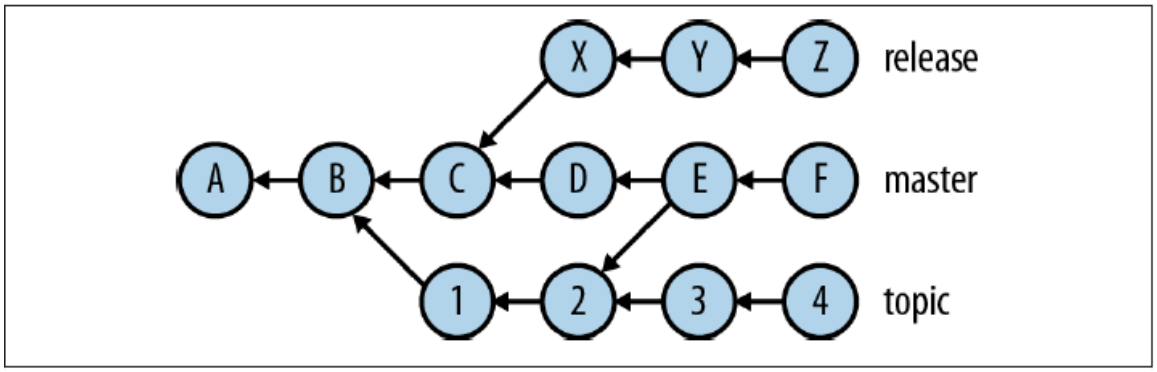


Figure 1 repository commit graph

图 1 是一个 repository 的提交图表，commitA 就是根版本，是初始提交。然后中间出现分支，如 commit1 和 commitX，而 commitE 就是 commit2 和 commitD 的一次归并

存储方式

Git 的数据库存储四个项，分别是：blobs，trees，commits 和 tags。

Blob：从字面上理解就是一大堆二进制数据，实际也是 git 对每一个版本的压缩存储文件。同时，git 使用 SHA-1 进行 hash 命名，并且用这个 hash 后的名字表示这个文件的唯一性，使得存储比对更加高效。Git 保证了你每一个版本都有一份完整的拷贝，同时也保证了仅有一份。

Tree：上面说了，tree 是一个目录结构，所以它包含 2 部分，一是文件名，包括相关信息（如用于权限控制的 mode bits 等），二是指向另一个对象的指针，指向的对象可能是 blob 或者另一个 tree

Commit：这里，commit 可以理解为版本，就是每一次改变的基础。一个 commit 就是整个项目内容的一次快照，包括辨识信息和与不同 commit 之间的关联。

Tag： tag 就是字面意思，是人为给定的标记，对于每一个版本都可以人为给一个可读性强的标记。Git 默认的不同版本的标示都是自己用 SHA-1 生成的 120hash 码。

你可以在你 repository 目录下的.git 文件中找到不同版本的包。使用如下命令：

```
$ find .git/objects –type f
```

引用（Refs）

Git 定义了两种类型的引用，或者说指针。一种就是常规的，指向对象 ID 的（commit 或 tag）；另一种是指向引用的引用。有点像 linux 系统中常说的硬连接和软连接。

Git 通过 refs 来对一些东西进行命名，包括 commits，branches 和 tags。（之前说了，原本的名字都是用 SHA-1 进行的 hash 码）。你可以使用如下命令，显示引用和对象各自的关联

```
$ git show-ref
$ git symbolic-ref #用于获取个别特殊的引用
```

分支（Branches）

对于分支最简单的解释其实可以理解为一次更改了引用(ref)的提交(commit)。具体的说，就是你和你的同事对同一个版本同时进行了修改，并且你在你本地重新命名了这个版本的名字，再上传上去的时候，你就相当于创建了一个分支。分支在你每次修改的时候都有可能产生，但是大多数时候，你在产生分支以后都进行了合并(merge)操作。

来看一下有关分支的常用命令：

```
$ git symbolic-ref HEAD
# 查看我当前分支是什么，HEAD 就是指向当前分支的引用
$ git branch
# 查看我的分支情况，这个命令会把所有的分支罗列，
##并在你当前所在分支前打*表示
$ git checkout mybranch
# checkout 命令可以让你进入到不同的分支
# 同时会检查你目前所在的分支有没有修改需要提交
# 检查通过后你才能顺利改变分支
```

合并（Merging）

顾名思义，合并就是把两个或多个分支合并到一起。大致有两种场景需要 merge 操作：

1、 你在自己的项目上编码，忽然想到一个实现方案，但是不一定成功，这个时候你创建了一个新的分支，然后开始实验你的想法，结果发现，这个想法相当好。为了应用到你原本的项目上，你可以使用 merge 把新写的东西与原来的主项目合并。

2、 你在一个项目的 master 分支上编码，这个时候别人也同时在这样操作。可能一两天下来你进行了很多次 commit，但是一直没有和远程仓库进行同步，这个时候别人也进行了很多次改动。为了把你针对老版本的改动提交到新版本上去，就需要使用 merge 功能把你的改动合并到新的版本中去。

PUSH 和 PULL

push 和 pull 将会是你最常使用的两个命令。Push 用来把你的代码提交到远程仓库，而 pull 则用于把远程仓库的代码同步到本地仓库。

具体来说，**git pull** 命令的执行包含这几个步骤：

1、首先执行 git fetch 命令，把远程仓库的代码抓取下来，包括 commit 历史等等信息。然后与你本地的信息进行对比。如果本地版本并没有做出改动，远程仓库中包含了本地的当前版本，那么这就是一个简单的增量更新。git 会帮你把版本一步步推进到与远程仓库同步的版本。

2、如果本地进行的更改，那么 git 会先对远程仓库和本地仓库的两个版本进行 merge 操作，产生出一个新的版本。

对于 **git push** 来说，其实进行的也是一样的内容，只是这是一次换位的操作。但是，如果你的版本不是对远程仓库的一次简单增量式更新，那么远程仓库就要求你先进行 git pull 在本地进行同步，然后才能把新的版本提交。git pull 操作实际上就是让你的版本与远程仓库的版本合并，生成了一个相当于对远程仓库进行增量式更新的版本。

基础配置

啰嗦了这么久，下面开始进入正题。
建立一个 git 项目最简单的操作如下:

```
$ mkdir git_project
# 新建文件夹 git_project
$ cd git_project
# 进入文件夹
$ git init
# 对 git 进行初始化
```

此时，你的项目下会自动生成一个.git 文件夹。因为 linux 下以.开头命名的文件都是隐藏文件，所以你可能需要使用下面的命令看到它。

```
$ ls -l
```

配置

Git 的配置信息分为两种，全局的和项目单独的。
全局的配置信息放置在用户的主目录下 ~/.gitconfig
项目单独的配置信息，都放在项目文件夹下的.git/config 文件中。
你可以直接在里面对已有的项目编辑。
使用类似如下的命令进行配置：

```
$ git config --global user.name "wonderflow"  #姓名
$ git config --global user.email wonderflow@wonderflow.info #邮箱
#此处–global 表示设置的是全局的配置参数，
#你可以使用–local 对当前工程进行配置，
#使用–system 对所有使用本机器的用户使用。
#--system 的配置信息可以在/etc/gitconfig 中找到。
$ git config --global core.editor vim
#设置默认使用的文本编辑器，在一些诸如提交 commit 等情况下使用
$ git config --list
#罗列出所有目前的配置清单
其他更多的 config 配置指令，请参阅：
$ git config --help
```

Git 项目

正常情况下，我们执行以下几个步骤把一个普通的项目加入 git。

```
$ cd project_folder
# 进入项目文件夹
$ git init
# 初始化 git
$ add .
```

```
# 把当前文件夹内的东西都加入到 git 的暂存区
$ git commit -m “init project”
# 把 git 暂存区的内容添加到 git 仓库，
## -m 参数即 message，快速填写 commit 信息
# 如果不加-m，会出现编辑区域让你编辑
$ git log --stat
# 显示 git 日志，查看更改的情况
$ git log --pretty=oneline
# 使用 pretty 参数是显示可读性强的 log
$ git show-ref master
# 顺带可以看一下 master 分支的 commit 编号
```

Readme

Readme 的重要性相信做过工程项目的人都明白。它是所有项目的入口，项目的制作者用它告诉大家这个项目的一些基本情况、用法以及注意点，项目的使用者则是用它来了解整个项目。

readme 文档应该包含如下几点必要信息：

- 这个项目是用来做什么的？
- 如何去使用？
- 项目中主要的文件和子目录的结构信息？

而一份合格甚至是优秀的 readme 文档应该包含如下信息：

- 整个项目的介绍（说明创建项目的意图或要解决的问题等）
- 指出编译整个项目需要的系统环境、参数等，并指出可能存在的移植性问题。
- 当前项目中的重要文件和子目录的结构信息。
- 项目相关的更多资源获取的方式信息。
- 编译、安装或使用说明。
- 项目作者及版权等相关信息。
- 项目版本更新或项目当前进展等情况。

一般 readme 文件都以 README.md 命名放在项目文件夹的根目录下。以.md 为结尾的命名方式告诉大家，一般网站都支持 markdown 语法所书写的 readme 文件。Markdown 是一种轻量级标记语言，它允许人们使用易读易写的纯文本格式编写文档。具体的语法知识，推荐大家使用搜索引擎查询一下。推荐这篇博客：http://hswg.info/blog/2013/01/01/markdown/

Ignore

Ignore 文件主要用于标注项目中的哪些文件或文件夹是可以忽略的。一般我们使用 git 只保存必须的文件和文件夹，大量生成项目所需要用到的内容是可以忽略的。所以 git 给出了 ignore 选项，来告诉项目哪些内容可以忽略。例如：

结果代码：*.o, *.so, *.a, *.dll, *.exe

二进制代码：*.jar (Java), *.elc (Emacs Lisp), *.pyc (Python)

中间过程文件：config.log, config.status, aclocal.m4, Makefile.in, config.h

通常来说，所有你的源码能自动生成的内容你都不应该加入到 git 的追踪目录中。

你有三种方式来写你的 ignore 文件。

1、 在你项目的根目录下加入.gitignore 文件，在里面加入你要 ignore 的文件名字。把.gitignore 文件加入到 git 项目中。这也是目前来说最为常用的方法

2、 在.git/info/exclude 文件中编写的内容同样也会被 git 项目所忽略。但是这只是属于你的配置，如果你上传到远程服务器，别人 clone 你项目以后，人家是不会有这份文件的，所以在人家的项目中不会忽略你在这个文件中所提及要忽略的内容。这里通常用于你觉得可以不要，但是人家未必同意的一些文件。

3、 在项目中，以任何文件名命名的一个文件，里面填写你要忽略的内容。同时执行如下命令：

```
$ git config --global core.excludesfile ~/[your_ignore_file_name]
```

这行命令就是在配置中把你的 ignore 文件加入进去。

至于 ignore 文件的语法，具体可以使用$ git ignore –help 查看。常用的有以下几种：

```
#在子目录下忽略特定文件
conf/config.h
#忽略当前目录特定文件
# 注意(不是 “./”)
## 不加斜杠的匹配会应用到当前以及所有子目录中
/super-cool-program
#忽略一个组合
# 即(*.o and *.a)写到一起
*.[oa]
# 忽略所有以特定后缀结尾的
*.so
#不要忽略以下文件，哪怕它被包含在某个匹配中
!my.so
# 忽略所有 temp 文件夹
temp/
```

值得注意的是，上述这些 ignore 文件的配置都是在你没有把你要 ignore 的文件 git add 之前有效，如果你已经把不需要的文件 git add 了 ，那么你可能需要在编写好 ignore 文件后执行

```
$ git update-index --assume-unchanged
```

进行提交

添加新文件

```
$ git add filename
```

添加对现有文件的修改

```
$ git add filename
```

是的，你没有看错，两个功能的命令是一样的。实际上这也是统一的，因为这两个操作都是 git 把文件放置到了暂存区。而放置到暂存区的命令，就是 add。

选择性添加文件

```
$ git add -p
```

你可以在这个命令后面直接添加文件名或目录，如果不添加，git 也会一一罗列出修改了的文件，给你看与前一版本比对的结果，询问你是否把他们添加进去。

```
$ git add -i
```

使用上述命令，你会得到一个交互式窗口，当然，其目的也是选择性添加文件。

```
$ git add -u
```

这个命令让你把修改过的和删除了的文件都添加到 git，但是不包括新建的文件。

删除文件

```
$ git rm filename
```

这个指令包含两个工作，从 git 中删除该文件，并在本地目录中也删除。相当于下面两步操作：

```
$ git rm --cached filename
$ rm filename
```

如果你想执行`$ git rm filename`，实际上却执行了`$ rm filename`
那也没关系，这相当于你对 filename 进行了更改，通过查看当前状态

```
$ git status
```

你会发现，git 告诉你，这次更改尚未进入暂存区。实际上你还需要执行

```
$git add filename
```

来完成这次删除操作。

重命名文件

```
$ git mv foo bar
```

这个相当于执行了如下两步操作：

```
$ mv foo bar
$ git add bar
```

顺道提一提 foo 和 bar 这两个名字，有时候还会出现 foobar，实际上这是 linux 编程文化中常用的指代变量。可以类比我们数学中常用的变量 x，y 等等，他们实际上并没有实际的意思。但经常被用来指代某个变量、函数或文件夹之类的。

取消修改

对于还没有进行提交的修改，你可以使用下面的命令进行取消：

```
$ git reset
```

你可以在该命令后面加上文件名或者目录，指定一个特殊的文件进行取消。你可以理解为这是`$ git add –p` 的一种逆操作。

提交

在之前已经提到，进行一次提交的最简单的方式如下：

```
$ git commit -m "an interesting commit message"
```

如果你不这么做，git 会开出一个文本编辑器来让你自行填写。当然，如果你做的变动比较多，你肯定需要打开编辑器来填写改动。在配置那一章节里提到怎么配置默认的编辑器：

```
$ git config --global core.editor vim
```

你只需要执行

```
$ git commit
```

Git 会用你指定的编辑器打开一个文件，你可以在这里尽情书写你所做的变动，但是我强烈建议你第一行，用简洁的话语把所做的变动整体描述一下。

你可以使用 `git log --grep` 来查看以前的变动。

如果你想要把 git 追踪到的有变动的文件一次性全部提交，就需要加入-a 参数，即：

```
$ git commit -a
```

其相当于执行了：

```
$ git add -u
$ git commit
```

小结

最后小结一下一次正常的提交流程。

1、 使用`$ git add`(添加不同的参数)来暂存你的变化。

2、 使用`$ git stash –keep-index`.这个命令把你当前工作目录中没有加入到暂存区的内

容保存到栈中，同时把这些不在暂存区的内容删除，以保证你下一步编译的是一个干净的工作目录。

3、 检查你的工作目录，确保你当前所做的变动都是正确的。比如说，你可以编译测试你的程序。

4、 运行`$ git commit` 命令。

5、 最后，使用 `$ git stash pop` 把你在第二步中保存的现场从栈中取出来，然后使用`$ git status` 确保工作目录中没有需要提交的文件了。最后回到第一步，继续接下来的工作。

版本回退

修改上一次提交

最常见的版本回退,应该是你突然发现,自己刚刚的提交漏加了一个文件,或者写错了什么。
对于这种情况,git 人性化的提供了一个最简单的悔过方式.即 amend(悔过)参数。
在发现自己的提交出现错误，想要后悔的时候，你什么准备都不需要做。直接把你想改的地方改好，然后在下一次提交的时候，使用下面的指令：

```
$ git commit --amend
```

跟基本的 commit 操作一样，你也必须要写“commit message”。同时，它的结构就是替换之前的那个 commit 版本。如果内容什么都没有变，它依旧会进行替换，所以操作还有一个额外的作用，就是可以修改“commit message”。

如果很不巧，你刚执行了`$ git commit –amend` 然后发现，自己在刚刚被覆盖的一次 commit 里面有一些东西又进行了误操作。这时 git 提供一个命令，让你查看过去的 ref 日志信息：

```
$ git log -g
```

加上-g选项，git会把日志信息中所有的更改记录都显示给你，包括commit,pull, checkout和merge等等。有了这些记录，你就可以通过 $ git check 之类的命令恢复，这个后面马上就会说到。
让我们来看一下两个命令，感受一下`$ git log –g`的强大。

```
$ git log
commit 7f10b35eb87beba14537012015178a5b14acef7b
    add c.cpp
commit e776b4fe1f0dc9de1a4afa5698b8e0749759cbbc
    add b.cpp
commit 54fdb47e15b716d1b91c7a721c1f0a17409dfd43
    add a.cpp


$ git log -g
commit 7f10b35eb87beba14537012015178a5b14acef7b
Reflog: HEAD@{0} Reflog message: checkout: moving from master to
7f10b35eb87beba14537012015178a5b14acef7b
    add c.cpp

commit ad40a290c1603b464b9c987f7acf5452431fa1b9
Reflog: HEAD@{1}Reflog message: revert: Revert "add b.cpp"
    Revert "add b.cpp"
    This reverts commit e776b4fe1f0dc9de1a4afa5698b8e0749759cbbc.

commit 7f10b35eb87beba14537012015178a5b14acef7b
Reflog: HEAD@{2} Reflog message: commit: add c.cpp
```

```
    add c.cpp

commit e776b4fe1f0dc9de1a4afa5698b8e0749759cbbc
Reflog: HEAD@{3} Reflog message: commit: add b.cpp
    add b.cpp

commit 54fdb47e15b716d1b91c7a721c1f0a17409dfd43
Reflog: HEAD@{4} Reflog message: commit (initial): add a.cpp
    add a.cpp
```

撤销上一次提交

使用 reset 这个命令，就可以达到撤销的效果。当然 reset 是一个多功能命令，它的作用并不仅仅如此，下面我们会提到。那么撤销上一次提交的命令就是：

```
$ git reset HEAD~
```

在经过了这个操作以后，原先的更改都变成了 unstaged，即没有经过暂存的。

那么，由这个可以猜测到，撤销前面某次的 commit 只要在 `HEAD~`后面加上第几次的编号即可，确实如此。

请注意，如果你目前进行了 3 次 `git commit` 操作，然后使用如下命令对第二次 commit 进行撤销：

```
$ git reset HEAD~2
```

那么，git 就把记录指向了最初的第一次提交，即第二次以及后面所有的 commit 操作都被撤销了。当然，光执行这么一条命令，当前目录结构是不会改变的。必须再使用 git checkout 对文件进行纠正，使其和 git 里的记录相同。那样以后你就会看到撤销的效果。

还原某一次提交

可能大家觉得撤销的这个功能不够好,如果我只是希望把某一次的提交取消掉,而不希望影响到后面的提交.那么,你就需要用到 revert 参数.

同样的,使用`$ git log` 或者`$ git log -g` 查看历史的一些版本变化，然后根据你的需要执行：

```
$ git revert [HASH 值]
```

如下，假设目前我们进行了这样几步操作

```
$ git log
commit 7f10b35eb87beba14537012015178a5b14acef7b
    add c.cpp
commit e776b4fe1f0dc9de1a4afa5698b8e0749759cbbc
    add b.cpp
commit 54fdb47e15b716d1b91c7a721c1f0a17409dfd43
    add a.cpp
$ ls
```

```
a.cpp  b.cpp  c.cpp
```

分别是添加了 3 个 cpp 文件。然后我们执行：

```
$ git revert e776b4fe1f0dc9de1a4afa5698b8e0749759cbbc
[master ad40a29] Revert "add b.cpp"
 1 file changed, 1 deletion(-)
 delete mode 100644 b.cpp
$ ls
a.cpp  c.cpp
```

很明显，b.cpp 没有了，第二次的添加操作被还原了。值得注意的是，与 reset 不同，如果你要还原的某次 commit 与后面的某次 commit 都涉及到对同一个文件的修改，git 就会检测到冲突，这时需要你来进行 merge 的操作以解决冲突。

还原部分更改

可能大家会注意到，如果直接使用`$ git revert`，当前目录结构是直接被修改了的。也就是说，一次性把所有的文件更改都还原了。那么如果能不更改当前目录结构，只是把 git 的记录修改，当前目录内容保留在暂存区内，这样我们就可以进行部分修改了。实际操作就是如此，在 revert 后面使用-n 参数：

```
$ git revert -n commit
$ git reset
$ git add -p
$ git commit
$ git checkout
```

参数-n 告诉 git，应用还原操作，并把我要还原的那次修改放置到暂存区里面，然后使用 reset 命令，把当前在暂存区里的内容都放到未暂存的位置。这样，我们就能使用`$ git add -p`一个一个把不想被还原的更改再加上去。最后提交修改，使用 checkout 把工作目录清理一下，就完成了我们还原部分更改的目的。

日志信息

我想，作为一个版本控制系统，对于每个版本的说明性日志信息的分析和处理，也是相当重要的一个部分。尤其是当你知道了版本回退的方法，却不知道该回退到哪一步的时候，日志信息就显得尤为重要了。
Git 确实拥有这样强大的能力，日志系统，`$ git log`。
如你所见，`$ git log` 指令，已经把最基本的一些 commit 信息推送给你了。但是可能你需要一些更强大的功能，比如：（为了节省篇幅，显示的效果就不贴出来了。）

格式控制

每个日志都用一行来简洁的显示，同时 hash 的 commit 值也只显示前六位，相信我，这会是你最常用的命令之一：

```
$ git log --oneline
```

这里是一张表，显示的是相应格式控制下显示的信息，相应的 format 名字替换 oneline 即可：

format	author	author date	committer	commit date	subject	message
oneline					✓	
short	✓				✓	
medium	✓	✓			✓	✓
full	✓		✓		✓	✓
fuller	✓	✓	✓	✓	✓	✓
email	✓	✓			✓	✓
raw	✓	✓	✓	✓	✓	✓

选择性输出

显示某分支上的日志信息：

```
$ git log develop_branch
```

Ref 的日志

如果你要查看所有的 commit 变动，包括使用版本回退后的情况：

```
$ git log -g
```

分支变化图

查看 git 分支变化的信息的利器，在多次 git 操作以后，这个图显得很漂亮：

```
$ git log --graph --oneline
```

日志匹配查找

Git 支持正则表达式查找，在日志中进行匹配信息的查找：

```
$ git log --grep=<pattern>
```

除此之外，还有-i (--regexp-ignore-case)忽略大小写，-E (--extended-regexp)扩展正则表达式，默认的是基本的。-F (--fixed-strings)把匹配字符就作为简单字符串，不解释为正则表达式。

查找特定用户的提交：

```
$ git log --auther=<pattern>
$ git log --committer=<pattern>
```

控制输出 commit 条数

显示前 10 条 commit 信息：

```
$ git log –n 10
```

或者

```
$ git log --max-count=10
```

跳过 10 条日志信息：

```
$ git log --skip 10
```

文件变化

显示不同版本文件的变化，后面可跟单独的某个文件名字，只显示该文件的情况：

```
$ git log --name-status
```

也可以分开使用：

```
$ git log --name-only
$ git log --stat
```

显示文件变化的比例：

```
$ git log --dirstat
```

显示不同版本的变化

显示每次 commit 与上次之间的变化：

```
$ git log -p
```

显示每次 merge 后的变化：

```
$ git log -m
```

颜色显示

通过--color 选项，git 会帮你把增加的代码用绿色显示，减去的用红色显示：

```
$ git log --color
```

帮助文档

相信你会用到更多更高级的命令，那么你需要使用这个来进行查看：

```
$ git log --help
```

分支

现在大家都知道如何在一个分支上进行管理，那是时候开始学习多分支的使用了。
分支的典型应用，是你要开发一个软件的新功能的时候，为了保证主代码的安全性与简洁性。我们就会在主分支之外开出一个新的分支，我们通常称之为“feature”分支或者“topic”分支。当你在开发新功能的时候，你在 feature 分支上工作；当你对主分支进行小修改时，你切换到主分支上。一段时间过后，你把主分支的代码 merge 到 feature 分支上，对 feature 分支进行更新。当 feature 分支上的功能开发完善后，就进行相反的操作，把 feature 分支 merge 到主分支上。
另一种对分支的应用主要体现在，你需要对老版本维护，同时你还要开发新版本。这时你可能在老版本上进行 bug 的修复，同时在新分支上开发新的功能。
实际上关于分支的讨论和规范是最贴近我们项目管理的要求的，具体使用哪个工作流在后面会深入讨论。

主分支(master)

众所周知，master 是 git 默认帮你创建的分支。当你从别处`$ git clone` 一份代码的时候，git 也是默认使用 master 分支打开。但是，当你使用`$ git init` 命令对项目进行初始化的时候，如果你使用`$ git log` 命令查看，你会发现，此时尚无 master 分支。
git 就帮你默认创建 master 分支，是在你第一次 commit 的时候创建的。

创建新分支

最常用的一个创建新分支的方法如下，假设我们要创建”dev”分支。

```
$ git checkout -b dev
Switched to a new branch ‘dev’
```

这个命令创建了一个新的分支，指向当前的 commit 版本。也就是说，当你执行了这个命令以后，master 分支和你新建的 dev 分支是同一个版本，同一份代码。同时，你进入了新的分支，在这个分支对代码进行操作不会对 master 分支产生任何影响。执行`$ git checkout master` 可以让你回到主分支。你可以在 figure2 分支演化图中了解的更清晰。

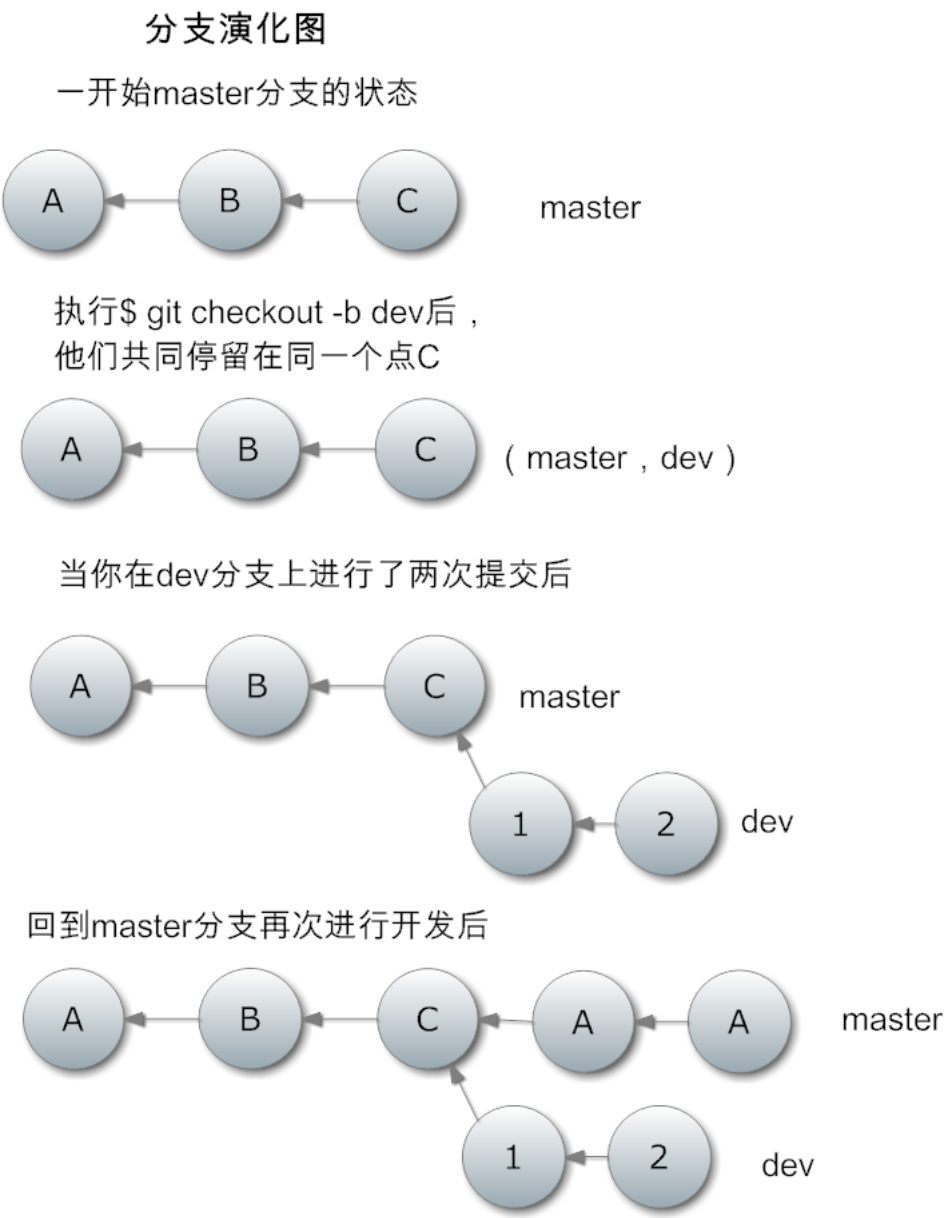


Figure 2 分支演化图

当然，你也可以把一个特定的 commit 作为分为一个分支的起点。只要使用如下命令：

```
$ git checkout -b dev [commit id]
Switched to a new branch 'dev'
```

要说明的是，如果你使用 checkout 创建新分支，在创建新分支的同时，它会把你转到新分支，所以如果你在原有的分支上有内容没有提交的话，你就需要把修改提交。

如果你只是单纯的想要创建一个新的分支，并不要跳转到新的分支上，那么你可以使用如下命令：

```
$ git branch dev
```

如果要查看当前有哪些分支，可以使用命令：

```
$ git branch --all
```

切换分支

切换分支最常用的操作就是$ git checkout。实际上，对于 git 来说，切换分支，只是把 HEAD 文件中的引用符号指向新分支的名字而已。所以，从定义上来说，HEAD 分支显示的就是你当前在哪个分支上。使用`$ git symbolic-ref HEAD` 可以显示你当前的引用（分支名）指向了哪个:

```
$ git checkout -b dev
Switched to a new branch 'dev'
$ git symbolic-ref HEAD
refs/heads/dev
$ git checkout master
Switched to branch 'master'
$ git symbolic-ref HEAD
refs/heads/master
```

那么，一条简单的`$ git checkout dev` 到底做了哪些事情呢？

1、 把 HEAD 引用符号指向 dev 分支。

2、 重置当前的 git 目录以匹配 dev 分支最新一次 commit 的情况。

3、 重置当前的工作目录，以匹配 git 目录。（这个实际上就是最纯粹的 checkout 干的事情）

值得一提的是：

如果你切换之前，有部分暂存区的文件，没有提交。那么如果你切换的分支中相关文件的情况跟当前分支是相同的，git 会帮你切换分支，并告诉你，这个文件在暂存区中等待提交。

如果文件本来就不同，那么 git 会组织你切换分支，要求你提交后再进行切换。

对于没有被 git 追踪的文件，如果目标分支中也没有该文件，那么切换分支后，那个文件就丢失了。如果目标分支中有那个文件，那么 git 会又一次组织你进行分支切换。你可以使 --merge 选项来对把文件合并到目标分支。

直接切换到一个 commit 编号的说明：

如果你直接使用`$ git checkout 3423b3a`，切换到某个特定的 commit 编号，那么 git 会告诉你，你进入了“detached HEAD state”。这个状态实际上是告诉你，你进入了一个无名分支，你一次一次的 commit 也会慢慢衍进，被记录。但是，一旦你 checkout 到别的分支的时候，你都会丢失这个 commit 记录，因为这是一个无名分支，你没有用如何名字来记录它。

防止丢失的办法很简单，你在这个分支上的如何时候，执行`$ git checkout –b new_name` 都可以把你当前这条分支记录下来。

删除分支

如果你想删除 git 的分支，那么 git 只是删除了一个指向 git 引用的一个分支名而已。它不会删除整个分支的内容，即不会从数据库里删除这些历史提交信息。因为这个要删除的分支中的某些 commit 可能被包含在其他分支里面。删除分支 dev 的命令为：

```
$ git branch -d dev
Deleted branch dev (was daef973).
```

当然，有时你无法轻易删除一个分支，如果它有新的内容没有被包含在别的分支里。（比如

有过 commit 以后没有被 merge）此时你可能需要 merge 一下，如果你非要删除整个分支，只要把参数改成大写 D 即可：

```
$ git branch -D dev
```

那么，什么时候会需要删除一个分支呢？

你要删除的一个分支，必须是你私有的。就是说，那是你在自己代码仓库中新建的，并且你不准备再用它，也不准备把它 push 到任何远程仓库。同样的，你也可以删除上游服务器的分支（如果老板允许你这么干的话!），但是你的操作并不会影响到别人。也就是说，你在服务器端删除了这个分支以后，别人之前已经下载过该分支的人如果执行`$ git pull`操作的话，并不会导致本地仓库代码被删除。Git 的宗旨就是，只有你能决定是否删除自己仓库里的分支，而你的操作也不会影响到他人的代码仓库。

分支重命名

重命名本地的一个分支很简单，执行如下命令即可：

```
$ git branch -m old new
```

但是，对于远程服务器上分支的重命名，我们就没有直接的操作了。必须先上传一个新的分支，然后删除旧的分支。

```
$ git push -u origin new
$ git push origin :old
```

是的，你没有看错，删除远程分支就是在冒号后面跟原有分支的名称。这是 gitV1.5 版本中添加的特性，可能比较难记。如果你装了 gitV1.7 或以上版本，也可以使用下面的命令删除远程仓库的分支：

```
$ git push origin –delete old
```

如果你删除了远程仓库的一个分支，请务必告诉你的同事，因为他们在`$ git pull`操作时会感到疑惑。不得不跟着你一起删除原来的分支，并且新建相同的分支名。实际上，重命名分支这样的操作不该是 git 干的活。

Clone

这个章节讲如何追踪一个远程仓库，包括克隆(clone)，上传(push)和同步(pull)。

Clone 一个远程仓库

可能大家最初接触 git 的时候，使用的第一个命令就是`$ git clone`，从 github 或者其他 git 服务器上获取一份代码。`$ git clone` 这个操作在本地初始化一个新的代码仓库，并且把所有的分支版本等信息设置到本地，这样后面就能进行 push、pull 操作进行本地和远程代码仓库的同步了。
当克隆完一个远程代码仓库以后，git 会检查远程的 HEAD 文件，如果 HEAD 文件时 master，那么它会自动帮你执行$git checkout master 命令，当然，你也可以切换到别的分支，使用-b 参数。

```
$ git clone git@github.com:wonderflow/test.git test
Cloning into 'test'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Receiving objects: 100% (4/4), done.
```

你可能注意到了，我的 git clone 命令后跟了一个参数，test。如果你不加这个参数，git 会自己创建一个文件夹，通常以远程的 git 名命名，然后把内容放到里面。如果你给了第二个参数，那么 git 会以这个参数命名创建这个文件夹，并把内容放进去。
在这里，我使用的 clone 方法是 ssh 的方式，需要预先把本地的 ssh 密码放置到 github 上，可能你需要参考这个教程：https://help.github.com/articles/generating-ssh-keys
当然，你也可以使用 http、https 等的方式，这些方式都大同小异，一般提供给你这些方式的 git 服务器都会告诉你如何使用。

使用 PUSH 和 PULL 同步

我们使用`$ git pull` 这样的命令来进行同步，git 同步的原理其实是根据一个工作流分支迭代的顺序来的，如果 git 发现，远程的最新版本（HEAD）在你当前的工作流分支的前几次迭代中出现了，那么 git 就会告诉你，你不需要使用`$ git pull` 进行更新。而当你的工作分支的最新版本(HEAD)，出现在了远程服务器最新版本的前几次迭代中，那么 git 就会按顺序把你本地的版本衍进，一次次迭代直到跟远程版本同步。
但是，如果在远程服务器中出现了不在本地仓库的最新版本，并且本地也出现了不在远程服务器中的最新版本，怎么办呢？

这个时候，就需要先进行`$ git merge`，把远程仓库的变化合并到本地，然后再使用`$ git push`把本地的上传如图 figure3 push and pull 所示。

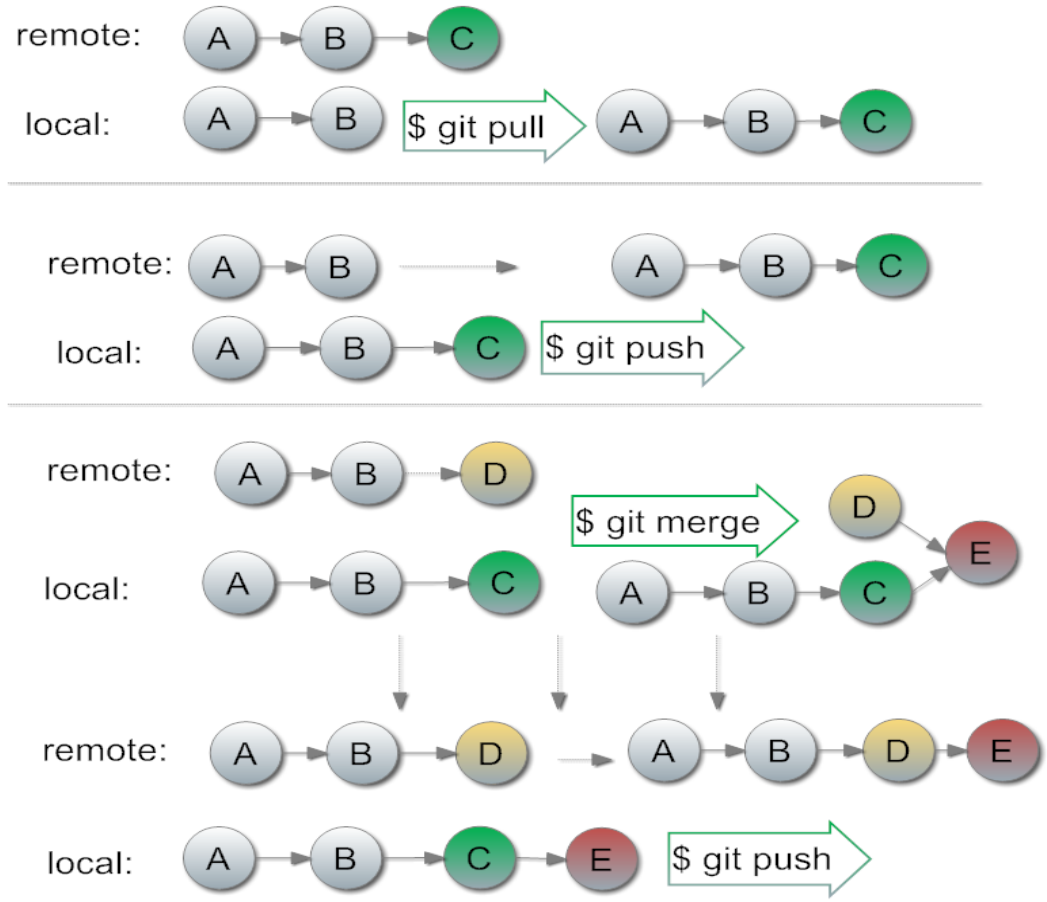


Figure 3 push and pull

Merge

简单的 merge 示范

之前已经很多次提到了 merge（合并）这个操作。最简单的 merge 操作，就是没有文件冲突的合并。

```
$ git checkout feature    #我们切换到 feature 分支
Switched to branch 'feature'
$ touch test.txt        #新建并添加 test.txt
$ git add test.txt
$ git commit -m "add test.txt"    #对本次更改进行提交
[feature b21f9e4] add test.txt
 0 files changed
 create mode 100644 test.txt
$ git checkout master         #切换到 master 分支
Switched to branch 'master'
$ git merge feature               #把 feature 分支合并到 master 分支
Merge made by the 'recursive' strategy.
 0 files changed
 create mode 100644 test.txt
```

以上，就是最简单的一个 merge 的示范。

请注意：请务必在进行 merge 操作前，把原有分支（master）的更改进行提交 commit。

Merge 工具

Merge可能是相当复杂的工作，也许对于普通的纯文本文件的对比，你可以简单的使用`$ git diff` 方法进行文件的差异性比较，但是对于大量琐事的文件，也许你需要更多更方便的工具。所以，git也支持很多种或开源或商业化的merge工具程序，包括araxis,emerge, opendiff, kdiff3, and gvimdiff。在git config的配置信息的merge.tool一栏填入相应工具程序的名字即可，当然，前提是你得要装上这个工具。

```
$ git config --global merge.tool gvimdiff
```

Merge strategy

当 git 出现冲突的时候，git 有很多种自动合并的策略，很容易联想到，要使用 git merge 的自动化策略，就是在后面加上-s 参数(strategy)

```
$ git merge –s [策略名]
```

下面开始介绍几个 git 提供的简单策略：

```
$ git merge -s ours
```

Ours 策略，是不顾所有其他分支进行了什么变化，都舍弃，只保留我当前分支的原状，然后把其他分支的 HEAD 指针拿进来，并且当我下次再进行 merge 操作时，其他已经在这次跟

我合并过的分支的那些变化不会再引进。这个有助于保证一个分支的历史不被其他分支影响。

```
$ git merge -s recursive
```

Recursive 策略是 git 在合并两个分支时使用的默认策略，也就是后面可以不写-s recursive。Recursive 策略先建立一个公共祖先树，树分叉枝干代表需要合并的不同点，git 通过这棵树，来找到需要你人工合并的地方，交给你来合并。

```
$ git merge –s octopus
```

Octopus 策略是 git 在合并三个及以上分支时使用的默认策略，这个策略的特点是，如果你的分支合并都是跟 recursive 第一步建树以后对枝干的衍生，不存在分叉即冲突，那么它就会顺利帮你合并。否则合并中途会中断，合并到一半的内容不会恢复，需要使用`$ git reset`，同时加上--hard 参数可以把工作目录也一起恢复了。

Octopus 合并策略通常被用于把多个不同主题的分支合并到主分支，用于**新发布版本**的使用。在使用 octopus 策略进行合并之前，不同的分支应该已经和主分支合并过了，所以不存在冲突，否则就如之前所说，合并会失败。Octopus 的优点在于它会把之前的合并分开来，然后一个一个合并到最后这个版本。这样做的有点相当明显，你可以在查看 commit 日志图的时候(`$ git log --graph`)看到一个相当清晰的 commit 情况。所以经常使用 Octopus 来进行 release 版本的合并。如下图 figure4 所示。

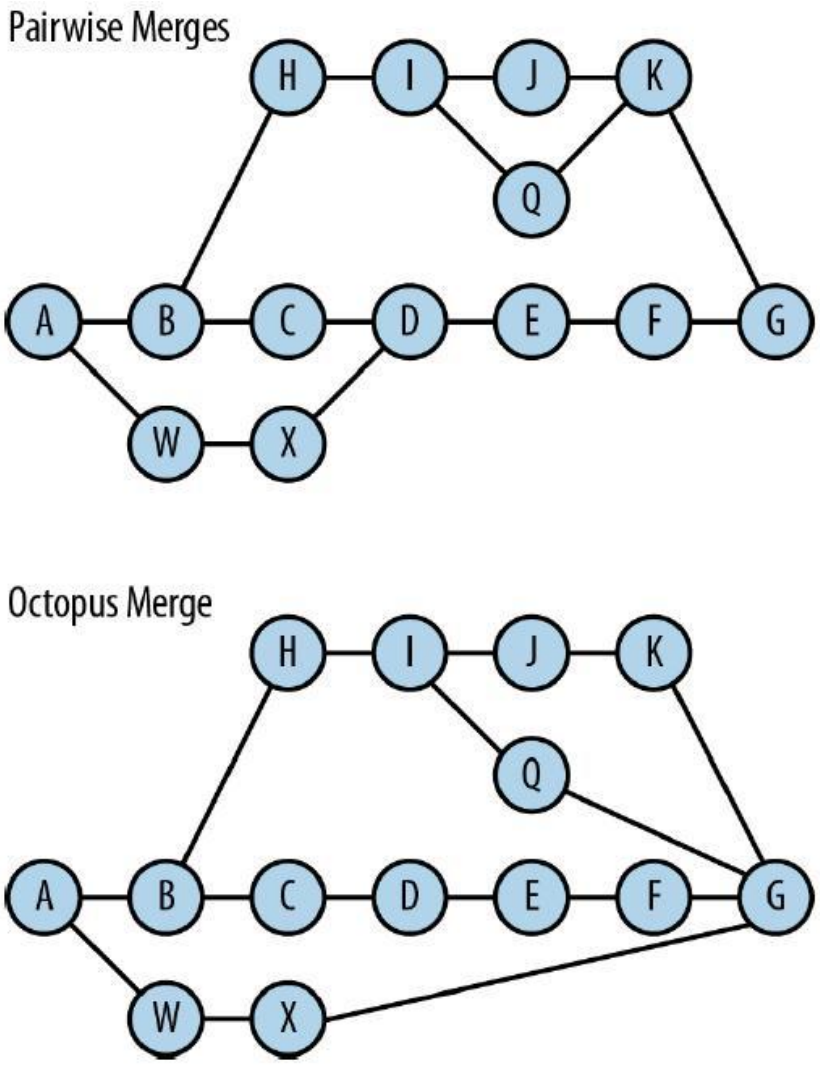


Figure 4

关于merge，最后要说的就是git还有一个功能，记录你之前作出的合并决定。如果你在做一个复杂的合并工作，不妨考虑利用一下这个功能。这个功能的名字是`$ git rerere`，名字的来源自然是缩写“reuse recorded resolution”。由于这个功能比较高级，在这个简介手册里，我们就不细说了，有兴趣的可以自己查一下。

Commit 命名规则

Git有一系列的方法让你快速定位到某个commit。

Commit ID：

首先你想到的必然是使用SHA-1进行hash的20位16进制数。

当然，还有其缩写形式，前八位16进制数，只要这个在你的代码仓库是唯一的，就可以用。

或者使用`$ git describe`命令进行commit命名的查看，当然你可能需要添加一些参数使用。

引用名：

如果你使用git通过引用（ref）来查找一个commit，假设引用名为foo，git通常通过如下顺序来查找：

1、 foo:通常这个是内置的如HEAD，MERGE_HEAD,FETCH_HEAD

2、 refs/foo

3、 refs/tags/foo:tags命名空间

4、 refs/heads/foo:本地分支中查找

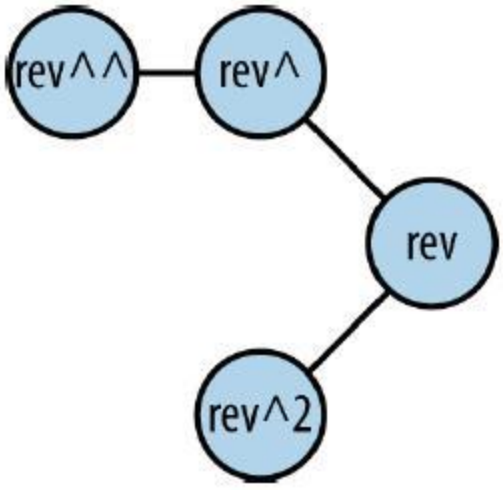
5、 refs/remotes/foo:远程分支中查找

通过和给定引用名的关系查找

Git有几个符号来专门用于定位某个ref的父commit，如“^”、“~”。在这里我们使用*rev*（revision）来指代某个commit引用名。

Rev^n:这表示rev这个commit前面一次迭代的第n个父commit节点。想想merge，你就知道为什么会出现这么一个奇怪的东西了，直接看图或许可以更快的理解。

显然rev^ == rev^1;但是rev^^不等于rev^2，这个一定要搞清楚。



那用什么符号来代替一连串的“^”呢？那就是“~”符号啦。

rev^^ == rev~2

rev~ == rev~1

rev~0 == rev

再次声明：HEAD~2 == HEAD^1^1 == HEAD^^，但绝不等于HEAD^2!

当然，这里你可以放在一起用，比如**releasev1~3^2**就表示releasev1这个tag在前面第三次合并时的第二个父节点。

Patch

通常意义上，我们把patch理解为补丁，表示两个文件之间的变化。显然，除了内容，它还包含了对于变化的记录。术语“补丁(patch)”和“比较（diff）”经常互换使用，但他们本质上还是有区别的。

Diff仅需显示两个文件之间的区别。Patch是对diff的扩展，增加上下文区别的上下文信息和文件名，这些特性使其应用更加广泛。当年，在还有git这个工具的时候，linus在写伟大的linux系统的时候，就是使用UNIX中的diff工具，来生成补丁，进行版本控制的。

这里是一个由diff程序创建的简单补丁：

```
$ diff --git a/foo.c b/foo.c
  index 30cfd169..8de130c2 100644
    --- a/foo.c
    +++ b/foo.c
    @@ -1,5 +1,5 @@
    #include <string.h>
    int check (char *string) {
  -  return !strcmp(string, "ok");
  +  return (string != NULL) && !strcmp(string, "ok");
  }
```

分析一下如上代码。Diff使用git的形式对a/foo.c和b/foo.c进行比较。值得一提的是，git diff中使用的并不是自己写的比较器，而是调用unix下的diff程序。我们比较的两个文件名字是相同的，所以用它们额外的前缀目录名a和b来区分它们。我们可以看到，不但包括代码的增减，还能看到增减代码的上下文信息，以及文件名称，这就是一个补丁。

让我们来细说上述补丁各行的内涵：

```
index 30cfd169..8de130c2 100644
```

这是一个扩展的头部行，该行给出了与这个文件相关的来自Git索引的信息：30cfd169 和 8de130c2 是文件内容A和B的commit ID；100644是“模式位”，表明了文件的读写执行权限。两个commit ID之间的“..”只是作为一个分隔符使用。

这里的commit ID是相当有用的，当git项目的merge操作存在冲突的时候，就能通过查看补丁的父节点的commit ID来和与之合并的分支中的一些历史commit版本进行对比，如果发现存在这个commit或者在merge过程中已经有过含带这样commit的版本，那么冲突就会顺理成章的被自动解决。

```
--- a/foo.c
+++ b/foo.c
```

这是diff程序统一的开头，表明被比较的文件以及改变的方向：减号表明A版本存在但B版本缺失的行；加号则正相反。有全局添加或删除文件操作的补丁文件，在其中一个文件处会有/dev/nul这个标记。

```
@@ -1,5 +1,5 @@
#include <string.h>
int check (char *string) {
- return !strcmp(string, "ok");
+ return (string != NULL) && !strcmp(string, "ok");
}
```

这是一个表明两个版本文件之间差异的部分，在这个diff中也就只有这么一处，我们姑且称之为“大块”。以@@开头的行用行号和行长度来表明版本A和B中“大块”的位置。这里，两个版本的“大块”都开始于行1并且延续到行5。接下来以空格开始的行是上下文，正如他们在文件的不同版本中显示出来的那样。那些以减号和加号开始的行的含义已经讨论过了，该补丁替换了原来的一行，解决了一个常见的C语言bug。

应用 git diff 的输出

如果你将`$ git diff` 的输出保存到一个文件（例如：`$ git diff > foo.patch`），那么你可以使用`$ git apply` 把它应用到相同或相似的文件版本中，或者你可以使用其他处理 diff 格式的常规工具，例如 path，不过这些工具都不能利用 diff 中额外的 Git 特色的信息。这种做法对保存一组未提交的改变并将其应用到另外一组不同的文件中或者将任何一组改变传送给任何没有使用 Git 的用户非常有用。
你可以把`$ git show commit` 的输出作为一个补丁来代表一个未合并的 commit，也可以作为`$ git diff commit~ commit`（比较某个 commit 和它父节点区别）的一个快捷键。

远程访问

在 Git Clone 就提到过，Git 可以通过不同的网络协议访问远程版本库进行 push 和 pull 操作，最常用的协议是 HTTP(S)，SSH 和本地 Git 协议。特别是当远程版本库接受 push 请求，访问协议为了授权访问通常会要求验证你的身份。这个过程称为“认证”，你可以通过各种途径完成它，例如提供一个用户名和密码。Git 协议本身不支持认证，所以这个过程通常由 HTTP 或 SSH 完成。

SSH

关于如何添加 ssh 公钥等信息，已经在《快速开始使用 gitlab》上给大家写过。

HTTP

一个提供远程访问 Git 版本库的 web 服务器可能也被设置成需要认证。尽管存在更加复杂精妙的机制，包括 Kerberos 和公钥证书，然而最普通 HTTP 的方式仍然是一个简单的用户名和密码。这使自动认证变得复杂，但 Git 自身的框架能够来管理和提供这样的证书。
每次 push 和 pull 都需要输入用户名和密码显得令人烦躁，所以你可以使用`$ git config`来配置他们。

```
$ git config --global credential.'https://10.10.0.1/test.git'.username
dieter
```

Git 拥有一种称为“证书助手”的机制，为了方便使用，它以不同的方式存储密码。在该助手中，有一个类似于 SSH 代理的 cache 将你的密码缓存在内存中以供 Git 使用。默认情况下 cache 是关闭的，若要启用它，执行下面这条命令：

```
$ git config --global credential.helper cache
```

一旦你这样做了，Git 应该会在一个登陆会话中提示你输入一次与任何一个 URL 相关的密码。当你提供了这个密码后，Git 将它存储在运行中的认证 cache 代理中，那么接下来的命令会自动从里面获取密码。你可以看到该代理的处理过程：

```
$ ps -e | grep git-cred | grep -v grep
33078 ttys001    0:00.01 ↲
git-credential-cache--daemon ↲
/home/res/.git-credential-cache/socket
```

Git 通过上面命令行显示的套接字与代理进行通信。

其他

一些有必要提及但是跟上面哪一部分都不搭的命令：

git cherry-pick

`$ git cherry-pick` 顾名思义，把最珍贵的挑选出来，它让你把一个 commit 集合的所有信息作为一次提交应用到当前分支。什么意思呢？
假设我们项目组有个 release1.0稳定版，现在做了一个 xx 功能准备发布，分支名为 xx-develop。如果我把 xx-develop 直接 merge 到 release1.0里发布，可能会造成一些问题，稳妥起见，我在 release1.0的基础上新建一个分支：

```
$ git checkout –b release2.0-test
$ git cherry-pick xx-develop
```

这样就把所有 xx-develop 至于 release1.0的所有 commit 更改应用上去了，可以试下该版本（release2.0-test）是否稳定了。当然，cherry-pick 的过程中也是会出现冲突的，有可能需要手动解决。
如果要应用的补丁存在一些隐患，那么`$ git cherry-pick` 可能会失败，这时，git cherry-pick 会以一种普通的方式使用归并机制来记录索引和工作文件中的冲突。然后，它会提醒你使用以下选项--{continue,quit,abort}。continue 表示在解决完冲突后继续提交；quit 表示跳过当前提交；abort 表示中止整个 cherry-pick。这一切都类似于`$ git rebase`。

git notes

由于提交是不可变的，所有一旦你完成提交后你就不能为其增加提交信息，而且如果你替换你已经 push 过的提交，那么你就不可避免地会对其他人造成损失。git 注解提供了一种为你的提交生成注释并解决以上困难的方式。

你的版本库的注解集合以如下方式保存在 refs/notes/commits 分支上：为了找到提交的注解，Git 在当前注解树中查找它的40位的16进制提交 ID，如果存在，将指向一个包含注解文本的 blob。每当你增加或者移除一个注解，Git 简单地向注解分支提交相应的改变，因此你可以通过`$ git log notes/commits` 看到你的注解的历史记录。

尽管一般情况下注解用来注释提交，事实上注解可以用在任何 Git 对象上。

git notes 子命令

你可以使用-f 选项替换已经存在的注解。缺省的 object 参数默认是 HEAD：

```
$ git notes list [object]
#通过对象 ID 列出所有注解或者列出所有没有对象的注解。
$ git notes {add,append,edit} [object]
#编辑一个对象的注解，包括 add，append，edit 方法。
```

```
$ git notes copy first second
#把注解从一个对象拷贝到另一个对象
$ git notes show [object]
#在屏幕上显示对象的注解
$ git notes remove [object]
#删除对象的注解
```

你可以通过--ref 选项来指定一个注解的引用，例如`$ git notes --ref=bugs`
最初，git 注解大部分情况下用于私人用途，因为对合并其他来源的注解没有清晰支持。最新的 Git 版本已经增加了一个`$ git notes merge`命令，具体可以参考 git-notes(1)。

git grep

Git 字符串查找子命令让你可以使用正则表达式查找你的版本库内容。你甚至可以在 Git 版本库外把它当做一个加强版的 Unix grep 命令。

结合正则表达式

Git 字符串查找可以处理表达式的逻辑组合{and，or，not}，而不是一个单一的正则表达式。使用中，匹配模式前要加-e，例如：

```
$ git grep -e '^#define' ↵
--and \( -e AGE_MAX -e MAX_AGE \)
```

上面的命令查找以#define开头并且包含AGE_MAX或MAX_AGE的行。因此，它既查找#define AGE_MAX又查找#define MAX_AGE。

查找什么

默认情况下，git grep 查找在工作树中的文件或者给定的提交或者树对象。给定的对象必须单独列出，你不能使用范围表达式例如 master..topic。你可以添加 glob-style 匹配模式到搜索路径来缩小文件查找范围，例如：

```
$ git grep pattern HEAD~5 master -- '*.[ch]' README
```

其他选项：

--untracked

包含未追踪的文件；增加--no-exclude-standard 来跳过一般情况下的“忽略”规则

--cached

搜索索引

--no-index

搜索当前目录，即使它不是 Git 版本库的一部分。增加--exclude-standard 来重设一般情况下的“忽略”规则

显示什么

默认情况下，Git 字符串查找会显示所有匹配行，并酌情注释文件名和对象，其他选项包括：

--invert-match (-v)

显示没有匹配的行

-n

显示行号

-h

省略文件名

--count(-c)

显示匹配行的数目而不是匹配行本身

--file-with-matches(-l)

仅仅列出包含匹配行的文件

--file-with-matches(-l)

仅仅列出不包含匹配行的文件

--full-name

显示与工作树顶端相关的文件名，而不是当前目录

--break

将同一个文件中的匹配行整理到一起输出，并且在两个结果集中间输出一个空行

--heading

在同一个文件中所有匹配行输出之前输出该文件名，而不是每个匹配行都输出一个文件名

--all-match

对所有包含”or”的多模式匹配，只输出至少有一行匹配所有模式的文件。

如何匹配

-i(--regexp-ignore-case)

忽略大小写，例如 hello 和 HELLO 同样匹配”Hello”

-E(--extended-regexp)

使用拓展的正则表达式；默认类型是基础的正则表达式

-F(--fixed-strings)

使用固定字符串来匹配，不需要把这个字符串当做正则表达式

--perl-regexp

使用 Perl 类型的正则表达式。如果 Git 没有内置--with-libpcre 选项，则 Perl 类型的正则表达式不可用。

git rev-parse

`$ git rev-parse` 是一个管道命令，主要用于帮助其他 Git 程序解析并解释部分使用常规选项来指定修改 Git 命令行。你可以直接用它，它有一些用于显示一个版本库的不同特性的有用的选项，包括：

--git-dir

显示当前版本库的 Git 目录

--show-toplevel

显示工作树的顶端

--is-inside-git-dir

表明当前目录是否存在于 Git 版本库

--is-inside-working-tree

表明当前版本库是否是为空

git clean

`$ git clean`从工作树中移除所有未追踪的文件，由全局匹配模式来有选择地限定。例如，git clean '*~' 移除所有备份文件。选项包括：

--force (-f)

强制移除。除非你将 clean.requireForce 设置为 false，否则如果没有这个标记，git clean 不会做任何改变。

--dry-run (-n)

显示将要执行的动作，但不会移除任何文件。

--quiet (-q)

只报告错误，不报告移除的文件。

--exclude=pattern (-e)

使”ignore”规则起作用

-d

移除未追踪的目录和文件。存在于其他 Git 版本库的目录将不会被移除，除非添加两个 -f 标记。

-x

跳过普通”ignore”规则，但仍然遵守-e 的规则。

-X

只移除被忽视的文件。

没有那个单个的 git clean 命令是最常用的，事实上它取决于你要做的事情。例如，重构包含编译过的对象的被忽视的文件往往花费很大，所以当你清理其他累积在你的工作树中未追踪的垃圾时你并不希望移除它们。另一方面，当你切换分支后，你也许希望移除所有对象文件来确保一个正确的构建，因为由 make 或 ant 这些工具表达的复杂项目中的依赖关系可能无法正确地处理如此大量的重新布置的文件。

git stash

`$ git stash` 保存你的当前索引和工作树，然后像 git reset --hard 那样重置工作树以匹配提交头部（HEAD）。它可以让你很方便地重载工作状态以便你改变分支，pull 等其它可能会被你当前的改变阻塞的操作。

保存的状态存储在一个“栈”中，这意味着你最后放进去的状态将是被第一个取出的。也就是说，如果保存了一个状态，再做一些改变，然后再保存，当你再保存这个状态时，这将是被储存的第二个状态，而非第一个。接下来要用到的术语“push”和“pop”是计算机中传统的压入或移除栈元素操作。不过与一个单纯的栈不同的是，该命令允许你绕开栈顺序而直接处理栈中的状态。

子命令

save

这个是默认的子命令，用于保存当前的工作状态，选项包括：

--patch (-p)

交互选择要保存的块，而不是 HEAD 与当前工作树的完整的差别。该命令与 patch 模

式下的 git add 工作方式相同。

--keep-index

告诉 Git 不要恢复已经应用到索引中的改变。

--include-untracked (-u)

保存未追踪的文件（一般情况下只会保存被追踪的文件）。保存像对象文件这样经常被忽略和未被追踪的编译项目是很有用的，因为重建它们要花费一些代价。

你也可以指定一个注释参数作为代表该 stash 的提交信息，比如 git stash save “bugfix in progress”。否则，Git 会自动产生一行默认信息，例如：

WIP on master: 72e25df0 'commit subject'

--keep-index 选项对在提交前测试进行到一半的改变非常有用。如果你使用 git add -p 来将你当前工作树的改变分离到多个提交中，你可能需要先测试这些提交。git stash save --keep-index 会保存你进行中的改变并恢复剩下的，因此你可以测试该中间过程。然后你再提交，并通过 git stash pop 恢复剩下的改变，如此重复直到结束。

list

列出整个 stash 栈，你可以通过 stash@{0}，stash@{1}来象征性地引用栈中元素。

show

显示一个给定 stash 的所有改变。

pop

git stash 的反向操作：恢复一个状态并将其从 stash 链表中移除。

apply

类似于 git stash pop，但并不从 stash 链表中移除恢复的状态。

branch <branchname> [stash]

转向一个新的分支，并恢复其 stash。

drop [stash]

从 stash 链表中移除 stash，默认是移除 stash@{0}。

clear

删除整个 stash 链表。

git show

`$ git show` 以一种适合实体类型的方式显示一个给定的实体（默认是 HEAD）

commit

提交的 ID，作者，日期和 diff

tag

标记信息和被标记的实体

tree

树的路径名

blog

内容

你可以使用 git diff foo~ foo 来查看一个提交与另外一个提交的差别，不过使用 git show foo 更加简单。你也可以使用--format 来自定义输出。

git tag

一个 Git 标签可以为一个提交提供一个稳定的，可读的名字，比如：“version-1.0”或“release/2012-08-01”。存在两种类型的标签：

轻量级的标签，仅仅是指向标记过提交的 refs/tags 里面的一个引用；

注释标签，也是 refs/tags 里面的一个引用，不过指向的是一个标记类型的对象。该标记不仅仅指向标记过的提交，还记录其他信息：标签作者，时间戳，标记信息以及可选的 GnuPG 秘钥签名。

```
$ git tag [tag_name] [commit]
```

创建了一个轻量级的指向一个给定提交的标签，如果不指定一个提交，则默认指向 HEAD。选项包括：

--annotate (-a)

创建一个注释标签

--sign (-s)

建立一个签名标签，使用使用提交者邮箱地址或 user.signingkey 的值作为 GnuPG 秘钥

--local-user=key-ID (-u)

建立一个签名标签，使用指定的 GnuPG 秘钥

--force (-f)

强制替换已存在标签

--delete (-d)

删除一个标签

--verify (-v)

验证一个标签的 GnuPG 签名

--list pattern (-l)

使用模式匹配列出相应的标签名，如果不指定一个模式，则列出所有的标签。如果有多个模式则列出至少匹配一个模式的标签。

--contains commit

列出给定提交的标签。

--points-at object

列出指向

回顾

最后让我们来回顾一下一系列常用的操作该如何做：

在本地创建 git 项目，并上传。

```
$ mkdir project           #新建文件夹
$ cd project
$ git init          #初始化 git
$ git config --global user.name "jianbo"   #配置用户、邮件、默认 commit 编辑器
$ git config --global user.email 0
$ git config --global core.editor vi
$ touch README.md      #创建 readme 文件
////////
$ vi README.md         #编辑文件
////////
$ git add README.md    #把文件添加到 git
$ git commit -m “my init commit” #提交
$ git remote add origin http://10.10.103.47/gitlab/jianbosun/test.git #配置 git 远程服务地址
$ git push origin master  #远程同步，第一次要输入完整，以后可以$ git push
```

复制已有的项目到本地。

```
$ git clone http://10.10.0.1/test.git test_project_dir
$ cd test_project_dir
$ ls
```

修改本地文件

```
$ vi README.md
$ git add README.md
$ git commit –m “modify readme”
$ git push
```

删除本地文件

```
$ git rm README.md
$ git commit –m “delete readme”
```

```
$ git push
```

同步远程服务器到本地

```
$ git pull origin master
```

从当前分支创建新的分支，并切换到新分支

```
$ git checkout -b feature
```

切换分支

```
$ git checkout master
```

在 master 分支合并新分支的东西

```
$ git merge feature
```

删除分支

```
$ git branch –d feature
```

添加 ssh key

参见：https://help.github.com/articles/generating-ssh-keys

```
$ ssh-keygen -t rsa -C "your_email@example.com"
#用 email 地址创建
 Generating public/private rsa key pair.
 Enter file in which to save the key (/c/Users/you/.ssh/id_rsa): [Press enter]
 Enter passphrase (empty for no passphrase): [Type a passphrase]
 Enter same passphrase again: [Type passphrase again]
 Your identification has been saved in /c/Users/you/.ssh/id_rsa.
 Your public key has been saved in /c/Users/you/.ssh/id_rsa.pub.
 The key fingerprint is:
 01:0f:f4:3b:ca:85:d6:17:a1:7d:f0:68:9d:f0:a2:db your_email@example.com
 $ clip < ~/.ssh/id_rsa.pub
 # 使用 clip 工具把公钥复制到剪切板。如果没有这个工具，就直接打开文件全选复制。
```

修改本地 remote 服务器地址

```
$ git remote rm origin     #删除原来的地址
$ git remote add origin git@10.10.0.1/test.git  #配置新的
$ git push origin master
$ git remote -v            #查看当前远程服务器配置
```

修改上次 commit

```
$ git add [files]
$ git commit --amend
```

编辑前面的 n 个 commits

```
$ git rebase -i HEAD~n
```

重置（撤销）最后 n 个 commits

```
$ git reset HEAD~n
```

把一个已经存在的 commit 应用到另一个分支上

```
$ git cherry-pick rev
```

在 merge 的时候显示冲突文件的名字

```
$ git status
$ git diff --name-only --diff-filter=U
```

把分支都罗列出来

本地的：`$ git branch`
所有的：`$ git branch -a`
简要显示本地分支的一些细节：`$ git branch -vv`
远程分支的细节：`$ git remote show origin`

把工作目录和暂存区的情况罗列出来

```
$ git status
```

把工作现场保存在栈里，并重现出来：

```
$ git stash pop
```

添加一个新分支

```
$ git branch foo origin/foo
```

or

```
$ git checkout -b foo
```

显示一次 commit 产生的变化

```
$ git diff rev~ rev
```

罗列添加的所有远程服务器

```
$ git remote -v
```

找出我丢失的一些 commit（比如使用 amend 覆盖掉的之类）

```
$ git log -g
```

至此，谢谢你的耐心阅读，再见！

——wonderflow