Git 教程

讲解 Git

版本控制的基本概念与常用操作：提交、分支、合并与协作流程。适合日常开发上手，也可当作速查。

- Git
- 分支
- 内部实现
- Git的操作
- 参考链接
- 标签
- git add
- git branch
- git cat-file
- git checkout
- git cherry-pick
- git clone
- git commit-tree
- git commit
- git diff
- git hash-object
- git init
- git log
- git ls-files
- git merge
- git pull
- git rebase
- git remote
- git reset
- git restore
- git rev-parse

- git revert
- git rm
- git show
- git stash
- Git switch
- git tag
- git update-index
- git update-ref
- git write-tree

Git

git是一种源码管理系统（ source code management，缩写为 SCM
）。它对当前文件提供版本管理功能，核心思想是对当前文件建立一个对象数据库（ object database ），将历史版本信息存放在这个数据库中。

git操作流程
安装 git
提交用户名和电子邮件

$ git config --global user.name "Some One"
$ git config --global user.email "someone@gmail.com"

1. git init：新建一个 git库
2. git status：查看目前状态
3. git add <文件名 >：添加文件从工作区到暂存区
4. git commit -m "提示信息 "：从暂存区提交到代码仓库
5. git log：查看提交 commit的信息
6. git remote add origin https://github.com/try-git/try_git.git : 添加远程指针
7. git push -u origin master：将本地的 master分支推送到远程 origin主机， -u
参数表示记住对应关系，下次可以直接 git push推送。
8. git pull origin master：将远程主机 origin的代码取回本地，与本地的 master分支合并
9. git diff HEAD：查看与上一次 commit的区别

发布一个版本

为当前分支打上版本号。

$ git tag -a [VERSION] -m "released [VERSION]"
$ git push origin [VERSION]

git对象
对象数据库包含四类对象。

Blob：包含二进制数据，它们是文件内容。只要文件内容改变，就会在对象数据库中生成一个 blob 对象。注意， blob
对象只保存文件内容，不含文件名和文件存储位置等信息。如果文件名改变，或者文件储存位置改变
，不会生成新的 blob对象。
Tree： blob对象的集合，以及它们的文件名和权限。一个 tree对象描述一个时点上的一个目录。
Commit：描述一个时点上的项目状态，包含一条 log信息，一个 tree对象和指向父节点（ parent
commits）的指针。第一个 commit对象没有父节点。
紀錄 root tree SHA1
紀錄 parent commit SHA1
紀錄作者、時間和 commit message 資訊
tag

对象数据库依赖 SHA哈希函数。当一个对象加入数据库，它会被 SHA 函数处理，得到的结果就是该对象在数据库中的名字（前两个字节被当作目录名，用来提高效率）。

git命令基本上是图数据库操作命令，用来删除 /操作节点、移动指针等等。

$ git init
$ echo hello > hello.txt
$ git add .
$ tree .git
# 存在 .git/objects/ce/013625030ba8dba906f756967f9e9ca394464a # 這是 hello 內容的 SHA1
$ printf "blob 6\x00hello\n" | shasum
$ echo "hello" | git hash-object --stdin
$ git cat-file -p ce0136

上面代码有几点需要注意。

git add命令就会生成二进制对象。
shasum命令返回字符串的 SHA哈希函数结果。
git hash-object命令计算一个文件的 git对象 ID，stdin
参数表示从标准输入读取，而不是从本地文件读取。
git cat-file命令显示 git对象文件的内容和大小信息， p参数表示以易于阅读的格式显示。

树对象保存当前目录的快照。

040000 tree 0eed1217a2947f4930583229987d90fe5e8e0b74 data
100664 blob 5e40c0877058c504203932e5136051cf3cd3519b letter.txt 100664 blob 274c0052dd5408f8ae2bc8440029ff67d79bc5c3 number.txt

commit（快照）对象也保存在 .git/objects目录。

```
tree ffe298c3ce8bb07326f888907996eaa48d266db4
author Mary Rose Cook <mary@maryrosecook.com> 1424798436 -0500
committer Mary Rose Cook <mary@maryrosecook.com> 1424798436 -0500

a1
```

配置

指定全局的 .gitignore文件。

```
$ git config --global core.excludesfile=/Users/flores/.gitignore
```

目录结构

- .git/refs/heads: 保存各个分支的指针
- .git/HEAD 文件，保存 HEAD指针

```
ref: refs/heads/master
```

上面代码说明 HEAD指向 `.git/refs/heads/master` 文件，该文件是一个 Hash值。

```
a87cc0f39d12e51be8d68eab5cef1d31e8807a1c
```

- .git/refs/tags: 保存 tag指针

缓冲区域（ index）

Index区域（ .git/index）是一个二进制文件，用来保存当前目录在某个时点的状态。

`git init` 命令用来创建 index区域，以及对象数据库（ .dircache/objects）。

100644 為檔案模式 ,表示這是一個普通檔案; 100755 表示可執行檔 ,120000 表示 symbolic link。

`.git/index` 文件，保存暂存区的文件名和对应的 Hash值，每行对应一个文件。下面是一个例子。

```
data/letter.txt 5e40c0877058c504203932e5136051cf3cd3519b
data/number.txt 274c0052dd5408f8ae2bc8440029ff67d79bc5c3
```

Git commit的全过程

1. 用內容產生 blob object
2. 寫入 file mode, blob SHA1, file name 到 staging area
3. 根據 staging area 產生 Tree object
4. 用 root tree SHA1 和 parent commit SHA1 產生 commit object
5. 用 commit SHA1 更新 master 參考

如何不用 `git add` 和 `git commit` 指令完成 commit 動作 ?

```
# git add的部分

$ echo "hola" | git hash-object -w --stdin
$ git update-index --add --cacheinfo \
100644 5c1b14949828006ed75a3e8858957f86a2f7e2eb hola.txt

# git commit的部分

$ git write-tree
$ git commit-tree 27b9d5 -m "Second commit" -p 30b060
$ git update-ref refs/heads/master 97b806c9e5561a08e0df1f1a60857baad3a1f02e
```

父节点

合并产生的新节点，会有两个父节点。第一个是当前所在分支的父节点，第二个合并进来的那个分支的父节点
。

Tag对象

Tag 分兩種 :annotated tag 才會產生 object。

```
$ git tag -a release
$ git rev-parse release
$ git cat-file -p 2450f3
```

tag对象的内容。

```
object 309be0
type commit
tag release
tagger ihower 1375383070 +0800
Release!
```

Reference (指针)

所谓指针（reference），只是一个链接，用来指向其他物体，方便引用。Git 有三种指针，但是所有指针归根结底都是指向某个 commit。

- Tag指针：指向某个 commit，或者指向某个 tag对象。保存位置在 .git/refs/tags/目录，文件名为 tag 名，内容为某个 commit或 ref object的 SHA1哈希。
- Branch指针：指向某个 commit。每次该分支有新的 commit，指针就会变动。
- HEAD指针：指向目前所在的 Branch，用来区分目前在哪个分支。比如，内容为 ref: refs/heads/master。

团队开发模式

集中式工作流程：團隊內部私有專案，大家都有權限 Push 到共用的 Repository

管理員工作流程：適合一般 Open Source 專案，只有少部分人有權限可以 Push到 Repository,其他開發者用用 request pull 請求合併。例如 GitHub 提供的 Fork 和 Pull Request 功能。

分支管理策略

github flow

- master 是 stable/production 可佈署的版本
- 任何開發從 master branch 分支出 feature branch

送 pull request 開始進行行討論、 code review和測試
最後合併回 master 代表可以佈署了

pros and cons

簡單、清楚、容易了解
搭配 Github 的 Pull Request 介面
沒有 release branch,東⻄一進 master 就上 production

Ruby on Rails
master 是開發版本
feature branches 審核完後,合併進 master
maintenance branches,用 cherry-pick 做 backporting
基本上就是 Github flow 加上 maintenance branches 維護舊版的設計
版本號 (Tag)打在 master 上,透過 preview 和 beta 的版本號提前釋出

CMake

master 預備釋出的版本 ,feature branches 從這裡分支出去
feature branch 完成後,合併進 next
next 整合版本,完成的 feature branch 先合併到這裡進行測試
在 next 測好的 feature branch,才合併進 master
可以將 master 合併進 next,減少之後的 code conflicts
不會將 next 合併進 master
nightly 每天 1:00 UTC 自動從 next branch 分支支出來跑自動測試

Git远程操作
Git的repo一般是用来指本地库，远程库（ remote）主要用来存档、合作、分享和触发持续集成。

参考链接

corbet, The guts of git: 最早的一篇介绍 Git的文章，可以了解 Git的总体设计思路

张文钿, git从微观到宏观

分支
分支（ branch）是 Git 最重要的概念之一，几乎所有 Git 操作流程都离不开分支。
本质上，分支只是一个动态指针（或者称为标签），指向某个提交（ commit）。所谓 “动态 ”
，指的是当你切换到当前分支，生成新的 commit 时，分支就会自动更新，指向这个新的 commit。

查看分支
git status命令查看当前是哪一个分支。
$ git status
git branch命令可以列出所有的本地分支。
$ git branch
- r 参数查看所有的远程分支。
$ git branch -r
- a 参数查看所有分支（包含本地和远程）。
$ git branch -a

创建分支
git branch
命令紧跟一个分支名，就可以创建该分支。注意，该命令不会切换分支，创建后依然停留在当前分支。
$ git branch MyBranch
上面命令创建了一个 MyBranch分支，创建后并不会切换到 MyBranch分支。 git checkout命令也可以创建分支，并且创建后会切换到该分支。

```
$ git checkout -b MyBranch
```

上面命令创建 `MyBranch` 分支，然后切换到该分支。

`git push` 命令可以创建远程分支，并且与本地同名分支建立对应关系。

```
$ git push -u origin MyBranch
```

上面命令在远程 `origin` 仓库创建了 `MyBranch` 分支，并且与本地的 `MyBranch` 分支建立对应关系。

创建远程的同名当前分支，可以使用下面的命令。

```
$ git push -u origin HEAD
```

上面命令中， `HEAD` 是 Git 自带的动态指针，总是指向当前的本地分支。

切换分支

`git checkout` 命令用来切换分支。

```
$ git checkout MyBranch
```

上面命令会切换到一个现有分支 `MyBranch` ，当前工作区会变成 `MyBranch` 的内容。

`git checkout` 命令也可以切换到远程分支。

```
$ git checkout --track origin/MyBranch
```

上面命令切换到远程的 `MyBranch` 分支。执行该命令之前，最好先执行 `git pull` 命令，将远程的内容都拉取到本地。

分支改名

`git branch` 命令的 `-m` 参数可以为当前分支改名。

```
$ git branch -m MyBranch
```

上面命令将当前分支改名为 `MyBranch` 。

删除分支

`git branch` 的 `-d` 参数可以删除分支。

```
$ git branch -d MyBranch
```

上面命令删除 `MyBranch` 分支，前提是该分支没有未合并的变动。

`-D` 参数会强制删除分支，不管有没有未合并变动。

```
$ git branch -D MyBranch
```

`git push` 命令可以删除远程分支。

```
$ git push origin --delete MyBranch
```

上面命令删除了远程的 `MyBranch` 分支。

内部实现

.git 目录
版本管理的所有信息，保存在项目的 .git目录之中。如果没有这个目录， Git
就会认为这个仓库没有建立版本管理，必须进行初始化。 git init命令的主要作用，就是建立 .git目录。
. git/config文件保存仓库的设置。
. git/HEAD文件保存当前工作分支的引用。
. git/hooks目录保存各种事件挂钩的脚本。
. git/objects目录保存文件的内容，格式是二进制 blob。

哈希签名
每次提交的时候，每个文件、每个树节点（目录）和提交本身，都会生成一个 40个字符长的 SHA-1 哈希。
哈希作为文件名，文件本身保留在 .git/objects目录之中。

git add 命令
git add会将变动的文件写入 .git/objects目录。它内部调用的是 git hash-object命令,该命令会计算 SHA-
1 哈希，并将 blob 文件放入 .git/objects目录。
$ git hash-object -w myfile.txt
git cat-file命令可以查看 blob 格式对应的原始文件内容。
$ git cat-file -p e69de29bb2d1d6434b8b29ae775ad8c2e48c5391

git commit 命令

`git commit`命令用于有变动的、并已提交到 `.git/objects`的文件写入历史。

`git write-tree`为当前仓库创建树节点。

```
$ git write-tree
```

`git commit-tree`命令将指定的树节点写入历史。

```
$ echo“first commit”| git commit-tree \ 6e9432aeedbad83fbffb7f8aae4a5d1ab50b7fdf
```

Git的操作

新建代码库

```
# 在当前目录新建一个Git代码库
$ git init

# 新建一个目录，将其初始化为Git代码库
$ git init [project-name]

# 下载一个项目和它的整个代码历史
$ git clone [url]
```

配置

Git的设置文件为 `.gitconfig`，它可以在用户主目录下，也可以在项目目录下。

```
# 显示当前的Git配置
$ git config --list

# 编辑Git配置文件
$ git config -e [--global]

# 设置提交代码时的用户信息
$ git config [--global] user.name "[name]"
$ git config [--global] user.email "[email address]"
```

增加 /删除文件

```
# 添加指定文件到暂存区
$ git add [file1] [file2] ...
```

```
# 添加指定目录到暂存区，包括子目录
$ git add [dir]

# 添加当前目录的所有文件到暂存区
$ git add .

# 删除工作区文件，并且将这次删除放入暂存区
$ git rm [file1] [file2] ...

# 停止追踪指定文件，但该文件会保留在工作区
$ git rm --cached [file]

# 改名文件，并且将这个改名放入暂存区
$ git mv [file-original] [file-renamed]
```

代码提交

```
# 提交暂存区到仓库区
$ git commit -m [message]

# 提交暂存区的指定文件到仓库区
$ git commit [file1] [file2] ... -m [message]

# 提交工作区自上次commit之后的变化，直接到仓库区
$ git commit -a

# 提交时显示所有diff信息
$ git commit -v

# 使用一次新的commit，替代上一次提交
# 如果代码没有任何新变化，则用来改写上一次commit的提交信息
$ git commit --amend -m [message]

# 重做上一次commit，并包括指定文件的新变化
$ git commit --amend <file1> <file2> ...
```

分支

```
# 列出所有本地分支
$ git branch

# 列出所有远程分支
$ git branch -r

# 列出所有本地分支和远程分支
$ git branch -a

# 新建一个分支，但依然停留在当前分支
$ git branch [branch-name]

# 新建一个分支，并切换到该分支
$ git checkout -b [branch]

# 新建一个分支，指向指定commit
$ git branch [branch] [commit]

# 新建一个分支，与指定的远程分支建立追踪关系
$ git branch --track [branch] [remote-branch]

# 切换到指定分支，并更新工作区
$ git checkout [branch-name]

# 建立追踪关系，在现有分支与指定的远程分支之间
$ git branch --set-upstream [branch] [remote-branch]

# 合并指定分支到当前分支
$ git merge [branch]

# 选择一个commit，合并进当前分支
$ git cherry-pick [commit]

# 删除分支
$ git branch -d [branch-name]
```

```
# 删除远程分支
$ git push origin --delete <branch-name>
$ git branch -dr <remote/branch>
```

标签

```
# 列出所有tag
$ git tag

# 新建一个tag在当前commit
$ git tag [tag]

# 新建一个tag在指定commit
$ git tag [tag] [commit]

# 查看tag信息
$ git show [tag]

# 提交指定tag
$ git push [remote] [tag]

# 提交所有tag
$ git push [remote] --tags

# 新建一个分支，指向某个tag
$ git checkout -b [branch] [tag]
```

查看信息

```
# 显示有变更的文件
$ git status

# 显示当前分支的版本历史
$ git log
```



```
# 显示commit历史，以及每次commit发生变更的文件
$ git log --stat

# 显示某个文件的版本历史，包括文件改名
$ git log --follow [file]
$ git whatchanged [file]

# 显示指定文件相关的每一次diff
$ git log -p [file]

# 显示指定文件是什么人在什么时间修改过
$ git blame [file]

# 显示暂存区和工作区的差异
$ git diff

# 显示暂存区和上一个commit的差异
$ git diff --cached [<file>]

# 显示工作区与当前分支最新commit之间的差异
$ git diff HEAD

# 显示两次提交之间的差异
$ git diff [first-branch]...[second-branch]

# 显示某次提交的元数据和内容变化
$ git show [commit]

# 显示某次提交发生变化的文件
$ git show --name-only [commit]

# 显示某次提交时，某个文件的内容
$ git show [commit]:[filename]

# 显示当前分支的最近几次提交
$ git reflog
```

远程同步

```
# 下载远程仓库的所有变动
$ git fetch [remote]

# 显示所有远程仓库
$ git remote -v

# 显示某个远程仓库的信息
$ git remote show [remote]

# 增加一个新的远程仓库，并命名
$ git remote add [shortname] [url]

# 取回远程仓库的变化，并与本地分支合并
$ git pull [remote] [branch]

# 上传本地指定分支到远程仓库
$ git push [remote] [branch]

# 强行推送当前分支到远程仓库，即使有冲突
$ git push [remote] --force

# 推送所有分支到远程仓库
$ git push [remote] --all
```

撤销

```
# 恢复暂存区的指定文件到工作区
$ git checkout [file]

# 恢复某个commit的指定文件到工作区
$ git checkout [commit] [file]

# 恢复上一个commit的所有文件到工作区
$ git checkout .

# 重置暂存区的指定文件，与上一次commit保持一致，但工作区不变
$ git reset [file]
```

```
# 重置暂存区与工作区，与上一次commit保持一致
$ git reset --hard

# 重置当前分支的指针为指定commit，同时重置暂存区，但工作区不变
$ git reset [commit]

# 重置当前分支的HEAD为指定commit，同时重置暂存区和工作区，与指定commit一致
$ git reset --hard [commit]

# 重置当前HEAD为指定commit，但保持暂存区和工作区不变
$ git reset --keep [commit]

# 新建一个commit，用来撤销指定commit
# 后者的所有变化都将被前者抵消，并且应用到当前分支
$ git revert [commit]
```

其他

```
# 生成一个可供发布的压缩包
# git archive
```

（完）

□□□□

- Andy Jeffries, 25 Tips for Intermediate Git Usersi- Mark Lodato, □□ Git

标签

推送

标签必须单独推送。也就是说，`git push`命令默认不会推送标签，必须使用`--tags`参数。

```
$ git push && git push --tags
```

上面的命令先推送新的 commit，成功后再单独推送标签。

`--follow-tags`参数会使得 commit 以及与之相关的标签（注意，不是所有的标签）一起推送。

```
$ git push --follow-tags
```

Git 有一个对应于`--follow-tags`的配置项，默认是关闭的。如果将它打开，以后执行`git push`的时候，默认就会带上`--follow-tags`。

```
$ git config --global push.followTags true
```

git add

概述
git add命令用于将变化的文件，从工作区提交到暂存区。它的作用就是告诉 Git
，下一次哪些变化需要保存到仓库区。用户可以使用 git status命令查看目前的暂存区放置了哪些文件。

# 将指定文件放入暂存区
$ git add <file>

# 将指定目录下所有变化的文件，放入暂存区
$ git add <directory>

# 将当前目录下所有变化的文件，放入暂存区
$ git add .

参数
- u 参数表示只添加暂存区已有的文件（包括删除操作），但不添加新增的文件。

$ git add -u

- A或者 --all参数表示追踪所有操作，包括新增、修改和删除。

$ git add -A

Git 2.0 版开始， -A参数成为默认，即 git add .等同于 git add -A。

- f 参数表示强制添加某个文件，不管 .gitignore是否包含了这个文件。

$ git add -f <fileName>

- p 参数表示进入交互模式，指定哪些修改需要添加到暂存区。即使是同一个文件，也可以只提交部分变动。

$ git add -p

注意，Git 2.0 版以前，`git add` 默认不追踪删除操作。即在工作区删除一个文件后，`git add` 命令不会将这个变化提交到暂存区，导致这个文件继续存在于历史中。Git 2.0 改变了这个行为。

实现细节

通过 `git add` 这个命令，工作区里面那些新建或修改过的文件，会加入 `.git/objects/` 目录，文件名是文件内容的 SHA1 哈希值。`git add` 命令同时还将这些文件的文件名和对应的哈希值，写入 `.git/index` 文件，每一行对应一个文件。

下面是 `.git/index` 文件的内容。

```
data/letter.txt 5e40c0877058c504203932e5136051cf3cd3519b
```

上面代码表示，`data/letter.txt` 文件的哈希值是 `5e40c087...`。可以根据这个哈希值到 `.git/objects/` 目录下找到添加后的文件。

git branch

`git branch`是分支操作命令。

```
# 列出所有本地分支
$ git branch


# 列出所有本地分支和远程分支
$ git branch -a
```

（1）新建一个分支

直接在 `git branch`后面跟上分支名，就表示新建该分支。

```
$ git branch develop
```

新建一个分支，指向当前 commit。本质是在 `refs/heads/`目录中生成一个文件，文件名为分支名，内容为当前 commit 的哈希值。

注意，创建后，还是停留在原来分支，需要用 `git checkout`切换到新建分支。

```
$ git checkout develop
```

使用 `-b`参数，可以新建的同时，切换到新分支。

```
$ git checkout -b NewBranch MyBranch
```

（2）删除分支

`-d`参数用来删除一个分支，前提是该分支没有未合并的变动。

```
$ git branch -d <分支名>
```

强制删除一个分支，不管有没有未合并变化。

```
$ git branch -D <分支名>
```

（3）分支改名

```
$ git checkout -b twitter-experiment feature132
$ git branch -d feature132
```

□□□□□

```
# □□□□□□□□
$ git branch -m twitter-experiment

# □□□□□□□□
$ git branch -m feature132 twitter-experiment

# □□□□□□□□□□□□□□
$ git branch -m feature132 twitter-experiment
```

□ 4□□□　　merge □□

```
# Shows branches that are all merged in to your current branch
$ git branch --merged

# Shows branches that are not merged in to your current branch
$ git branch --no-merged
```

□□□□

-d

`-d`□□□□□□□□□□□□□□

```
$ git branch -d <branchname>
```

-m□ -M

`-m`□ `-M`□□□□□□□

```
$ git branch (-m | -M) <oldbranch> <newbranch>
```

□□□□□　　`oldbranch`□□□　`newbranch`□

如果省略 `oldbracn`，则将当前分支改名为 `newbranch`。

```
$ git branch (-m | -M) <newbranch>
```

如果 `newbranch` 已经存在，则必须使用 `-M` 参数，进行强制改名。

git cat-file

`git cat-file`命令显示一个 Git对象文件的内容。

```
$ git cat-file -p aaa96
```

`p`参数表示以易于阅读的格式显示。

git checkout

`git checkout`命令有多种用途。

（1）用来切换分支。

```
$ git checkout
```

上面命令表示回到先前所在的分支。

```
$ git checkout develop
```

上面命令表示切换到 `develop`分支。

（2）切换到指定快照（commit）

```
$ git checkout <commitID>
```

（3）将工作区指定的文件恢复到上次提交的状态。

```
$ git checkout -- <filename>
```

上面命令将指定文件从暂存区复制到工作区，同时丢弃工作区对该文件的修改。所以，使用的时候必须小心，
它会丢失对当前文件的修改，回到上一次保存到暂存区的状态。

```
$ git checkout HEAD~ -- <filename>
```

它还可以从其他分支或提交恢复指定文件，上例是 `HEAD~`，这会同时改变暂存区和工作区。

`-p`参数表示进入交互模式，手动选择恢复哪些文件。

```
$ git checkout -p
```

（4）切换到某个 tag

```
$ git checkout tags/1.1.4
# 或者
$ git checkout 1.1.4
```

上面第二种用法的前提是，本地不能有叫做 1.1.4的分支。

（ 5）生成新的分支。

`-b`参数用于生成一个新的分支，并切换到该分支。

```
$ git checkout -b new_branch
```

git cherry-pick
对于多分支的代码库，将代码从一个分支转移到另一个分支是常见需求。

这时分两种情况。一种情况是，你需要另一个分支的所有代码变动，那么就采用合并（ git merge ）。另一种情况是，你只需要部分代码变动（某几个提交），这时可以采用 Cherry pick。

一、基本用法
git cherry-pick命令的作用，就是将指定的提交（ commit）应用于其他分支。

$ git cherry-pick <commitHash>

上面命令就会将指定的提交 commitHash
，应用于当前分支。这会在当前分支产生一个新的提交，当然它们的哈希值会不一样。

举例来说，代码仓库有 master和 feature两个分支。

a - b - c - d Master
\
e - f - g Feature

现在将提交 f应用到 master分支。

# 切换到 master 分支
$ git checkout master

# Cherry pick 操作
$ git cherry-pick f

上面的操作完成以后，代码库就变成了下面的样子。

a - b - c - d - f Master
\

```
            e - f - g Feature
```

从上面可以看到，`master`分支的末尾增加了一个提交`f`。

`git cherry-pick`命令的参数，不一定是提交的哈希值，分支名也是可以的，表示转移该分支的最新提交。

```
$ git cherry-pick feature
```

上面代码表示将`feature`分支的最近一次提交，转移到当前分支。

二、转移多个提交

Cherry pick 支持一次转移多个提交。

```
$ git cherry-pick <HashA> <HashB>
```

上面的命令将 A 和 B 两个提交应用到当前分支。这会在当前分支生成两个对应的新提交。

如果想要转移一系列的连续提交，可以使用下面的简便语法。

```
$ git cherry-pick A..B
```

上面的命令可以转移从 A 到 B 的所有提交。它们必须按照正确的顺序放置：提交 A 必须早于提交 B，否则命令将失败，但不会报错。

注意，使用上面的命令，提交 A 将不会包含在 Cherry pick 中。如果要包含提交 A，可以使用下面的语法。

```
$ git cherry-pick A^..B
```

三、配置项

`git cherry-pick`命令的常用配置项如下。

（1）`-e`，`--edit`

打开外部编辑器，编辑提交信息。

（2）`-n`，`--no-commit`

只更新工作区和暂存区，不产生新的提交。

（3）`-x`

在提交信息的末尾追加一行`(cherry picked from commit ...)`，方便以后查到这个提交是如何产生的。

（4）`-s`，`--signoff`

在提交信息的末尾追加一行操作者的签名，表示是谁进行了这个操作。

（5）`-m parent-number`，`--mainline parent-number`

如果原始提交是一个合并节点，来自于两个分支的合并，那么 Cherry pick
默认将失败，因为它不知道应该采用哪个分支的代码变动。

`-m`配置项告诉 Git，应该采用哪个分支的变动。它的参数`parent-number`是一个从`1`
开始的整数，代表原始提交的父分支编号。

```
$ git cherry-pick -m 1 <commitHash>
```

上面命令表示，Cherry pick 采用提交`commitHash`来自编号 1的父分支的变动。

一般来说，1号父分支是接受变动的分支（the branch being merged into），2
号父分支是作为变动来源的分支（the branch being merged from）。

四、代码冲突

如果操作过程中发生代码冲突，Cherry pick 会停下来，让用户决定如何继续操作。

（1）`--continue`

用户解决代码冲突后，第一步将修改的文件重新加入暂存区（`git add .`），第二步使用下面的命令，让
Cherry pick 过程继续执行。

```
$ git cherry-pick --continue
```

（2）`--abort`

发生代码冲突后，放弃合并，回到操作前的样子。

（3）`--quit`

发生代码冲突后，退出 Cherry pick，但是不回到操作前的样子。

五、转移到另一个代码库
Cherry pick 也支持转移另一个代码库的提交，方法是先将该库加为远程仓库。 $ git remote add target git://gitUrl
上面命令添加了一个远程仓库 target。
然后，将远程代码抓取到本地。
$ git fetch target
上面命令将远程代码仓库抓取到本地。
接着，检查一下要从远程仓库转移的提交，获取它的哈希值。
$ git log target/master
最后，使用 git cherry-pick命令转移提交。 $ git cherry-pick <commitHash> （完）

git clone

`git clone`命令用于克隆远程分支。

```
$ git clone alpha delta --bare
```

上面命令表示将 alpha目录（必须是 git代码仓库），克隆到 delta目录。 bare参数表示 delta 目录只有仓库区，没有工作区和暂存区，即 delta目录中就是 .git目录的内容。

git commit-tree

□□□□□□□□□□□□ commit□□□

```
$ git commit-tree 16e19f -m “First commit”
```

git commit

`git commit`命令用于将暂存区中的变化提交到仓库区。

`-m`参数用于指定 commit 信息，是必需的。如果省略 `-m`参数，`git commit` 会自动打开文本编辑器，要求输入。

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

`git commit`命令可以跳过暂存区，直接将文件从工作区提交到仓库区。

```
$ git commit <filename>  -m "message"
```

上面命令会将工作区中指定文件的变化，先添加到暂存区，然后再将暂存区提交到仓库区。

命令行参数

-a

`-a`参数用于先将所有工作区的变动文件，提交到暂存区，再运行 `git commit`。用了 `-a`参数，就不用执行 `git add .`命令了。

```
$ git commit -am "message"
```

如果没有指定提交说明，运行下面的命令会直接打开默认的文本编辑器，让用户撰写提交说明。

```
$ git commit -a
```

--allow-empty

`--allow-empty`参数用于没有提交信息的 commit。

```
$ git commit --allow-empty
```

--amend

`--amend` 参数用于撤销上一次 commit，然后生成一个新的 commit。

```
$ git commit --amend - m "new commit message"
```

--fixup

`--fixup` 参数的含义是，当前添加的 commit 是以前某一个 commit 的修正。以后执行互动式的 `git rebase` 的时候，这两个 commit 将会合并成一个。

```
$ git commit --fixup <commit>
```

执行上面的命令，提交说明将自动生成，即在目标 commit 的提交说明的最前面，添加 “fixup! ”这个词。

-m

`-m` 参数用于添加提交说明。

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

--squash

`--squash` 参数的作用与 `--fixup` 类似，表示当前添加的 commit 应该与以前某一个 commit 合并成一个，以后执行互动式的 `git rebase` 的时候，这两个 commit 将会合并成一个。

```
$ git commit --squash <commit>
```

git diff

`git diff`命令用于查看文件之间的差异。

```
# 查看工作区与暂存区的差异
$ git diff

# 查看某个文件的工作区与暂存区的差异
$ git diff file.txt

# 查看暂存区与当前 commit 的差异
$ git diff --cached

# 查看两个commit的差异
$ git diff <commitBefore> <commitAfter>

# 查看暂存区与仓库区的差异
$ git diff --cached

# 查看工作区与上一次commit之间的差异
# 即如果执行 git commit -a, 将提交的文件
$ git diff HEAD

# 查看工作区与某个 commit 的差异
$ git diff <commit>

# 显示两次提交之间的差异
$ git diff [first-branch]...[second-branch]

# 查看工作区与当前分支上一次提交的差异，但是局限于test文件
$ git diff HEAD -- ./test

# 查看当前分支上一次提交与上上一次提交之间的差异
$ git diff HEAD -- ./test

# 生成patch
$ git format-patch master --stdout > mypatch.patch
```

比较两个分支

```
# 查看topic分支与master分支最新提交之间的差异
$ git diff topic master

# 与上一条命令相同
$ git diff topic..master

# 查看自从topic分支建立以后，master分支发生的变化
$ git diff topic...master
```

git hash-object

`git hash-object`命令计算一个文件的 git对象 ID，即 SHA1的哈希值。

```
$ echo "hello" | git hash-object --stdin
$ echo "hola" | git hash-object -w --stdin
```

参数

- w 将对象写入对象数据库
- stdin 表示从标准输入读取，而不是从本地文件读取。

git init
git init命令将当前目录转为 git仓库。
它会在当前目录下生成一个 .git子目录，在其中写入 git的配置和项目的快照。

git log

`git log` commit

```
# 
$ git log

# 
$ git log --follow [file]
```

```
$ git log remote/branch
```

log commit

```
$ git log --author=Andy
$ git log -i --grep="Something in the message"
```

`-i`

commit

```
$ git log origin/master..new
# [old]..[new] - everything you haven't pushed yet
```

```
git log --graph --decorate --pretty=oneline --abbrev-commit
```

- --graph commit
- --decorate commit
- --pretty=oneline commit
- --abbrev-commit commit SHA1 7

--oneline

`git log` □□□□□□ commit □□□□□□□□□□□□□ `--oneline` □□□□□□□□□ commit □□□□□□

```
$ git log --oneline --decorate
ccc3333 (HEAD, my-feature-branch) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 (master) Old stuff on master
```

git ls-files

```
# □□□□□.gitignore□□□□□
$ git ls-files --other --ignored --exclude-standard
```

git merge

将当前分支合并到指定分支。

```
$ git merge develop
```

将当前分支与 develop分支合并，产生的新的 commit对象有两个父节点。

如果 “指定分支 ”本身是当前分支的一个直接子节点，则会产生 fast-forward 合并，即合并不会产生新的节点，只是让当前分支指向 “指定分支 ”的最新 commit。

Git合并所采用的方法是 Three-way merge ，及合并的时候除了要合併的兩個檔案，還加上它们共同的父节点。这样可以大大減少人為處理 conflict 的情況。如果采用 two-way merge，則只用兩個檔案進行合併（ svn默认就是这种合并方法。）

git pull

```
# 合并指定分支到当前分支
$ git pull . topic/branch
```

即使当前分支有没有 commit 的变动，也可以使用 `git pull` 从远程拉取分支。

git rebase

git rebase 将当前分支移植到指定分支或指定 commit之上。

```
$ git rebase -i <commit>
```

互动的 rebase。

```
$ git rebase -i master~3
```

命令行参数

--autosquash

`--autosquash`参数用于互动模式，必须与 `-i`参数配合使用。它会使得以前通过 `git commit --fixup`和 `git commit --squash`提交的 commit，按照指定的顺序排列（实质是选择提交说明以以 `fixup!`或 `squash!`开头的 commit），即 `--fixup`的 commit 直接排在它所对应的 commit 的后面。

```
$ git rebase --interactive --autosquash <branch>
```

--continue

`--continue`参数用于解决冲突以后，继续执行 rebase。

```
$ git rebase --continue
```

-i，--interactive

`-i`参数会打开互动模式，让用户选择定制 `rebase`的行为。

```
$ git rebase -i develop
```



- Auto-squashing Git Commits, by George Brocklehurst

git remote

□□□□□□□□□□

```
$ git remote add john git@github.com:johnsomeone/someproject.git

# □□□□□□□□□□
$ git remote -v

# □□□□□□□□□□□□
$ git remote show name
```

`git remote` □□□□□□□ `.git/config` □□□□□□□□□□

```
$ git remote add bravo ../bravo
```

```
[remote "bravo"]
    url = ../bravo/
```

git reset

`git reset` 命令用于将当前分支指向另一个位置。

```
# 将当期分支的指针倒退三个 commit，
# 并且会改变暂存区
$ git reset HEAD~3

# 倒退指针的同时，不改变暂存区
$ git reset --soft HEAD~3

# 倒退指针的同时，改变工作区
$ git reset --hard HEAD~3
```

如果不指定回滚的位置，那么等同于撤销修改。

```
# 撤销上一次向暂存区添加的所有文件
$ git reset

# 无任何效果
$ git reset --soft

# 同时撤销暂存区和工作区的修改，
# 回复到上一次提交的状态
$ git reset --hard

# 撤销上一次向暂存区添加的某个指定文件，
# 不影响工作区中的该文件
$ git reset -- <filename>
```

参数

- soft: 不改变工作区和缓存区，只移动 HEAD 到指定 commit。
- mixed: 只改变缓存区，不改变工作区。这是默认参数，通常用于撤销 `git add`。
- hard: 改变工作区和暂存区到指定 commit。该参数等同于重置，可能会引起数据损失。`git reset --hard` 等同于 `git reset --hard HEAD`。

- `-p`表示键入交互模式，指定暂存区的哪些部分需要撤销。

```
# Undo add
$ git reset

# Undo a commit，不重置工作区和缓存区
# 回到 HEAD 之前的那个 commit
$ git reset --soft HEAD^

# Undo a commit，重置工作区和缓存区
# 连续撤销三个 commit：HEAD, HEAD^, and HEAD~2
$ git reset --hard HEAD~3

# 从暂存区移除指定文件，但不改变工作区中的该文件
$ git reset -- frotz.c
```

git restore

`git restore` Git 2.23 `git checkout`

```
$ git restore -- test.txt
```

`test.txt`

git rev-parse

□□□□□□□□ SHA1□□□□

```
$ git rev-parse HEAD
```

git revert

`git revert` □□□□□□ commit□

```
$ git revert <commitID>
```

git rm

`git rm`命令用于删除文件。

解除追踪某个文件，即该文件已被 `git add`添加，然后抵消这个操作。

```
$ git rm --cached <fileName>
```

git show

`git show` □□□□□□ commit□□□

```
# □□□□□□□□□□□□□□□□□□
$ git show [commit]

$ git show 12a86bc38 # By revision
$ git show v1.0.1 # By tag
$ git show feature132 # By branch name
$ git show 12a86bc38^ # Parent of a commit
$ git show 12a86bc38~2 # Grandparent of a commit
$ git show feature132@{yesterday} # Time relative
$ git show feature132@{2.hours.ago} # Time relative
```

git stash

`git stash` commit
commit

`git reset --hard` `git commit` “
”

```
# 
$ git stash
Saved working directory and index state WIP on workbranch: 56cd5d4 Revert "update old files"
HEAD is now at 56cd5d4 Revert "update old files"

# 
$ git stash list
stash@{0}: WIP on workbranch: 56cd5d4 Revert "update old files"
stash@{1}: WIP on project1: 1dd87ea commit "fix typos and grammar"

# 
$ git stash apply stash@{1}

# stash
$ git stash pop

# stash
$ git stash drop

# stash
$ git stash clear
```

commit

```
$ git stash pop
```

git stash命令可以运行多次，保存多个未提交的修改。这些修改以 “先进后出 ”的stack结构保存。

git stash list命令查看内部保存的多次修改。

$ git stash list
stash@{0}: WIP on new-feature: 5cedccc Try something crazy
stash@{1}: WIP on new-feature: 9f44b34 Take a different direction
stash@{2}: WIP on new-feature: 5acd291 Begin new feature

上面命令假设曾经运行过 git stash命令三次。

git stash pop命令总是取出最近一次的修改，但是可以用 git stash apply指定取出某一次的修改。

$ git stash apply stash@{1}

上面命令不会自动删除取出的修改，需要手动删除。

$ git stash drop stash@{1}

git stash 子命令一览。

# 展示目前存在的stash
$ git stash show -p

# 切换回stash
$ git stash pop

# 清除stash
$ git stash clear

参考链接

Ryan Hodson, Quick Tip: Leveraging the Power of Git Stash

Git switch

`git switch`是 Git 2.23 版本新增的命名，用来切换分支，跟 `git checkout`的作用类似，但有更好的语义。

```
$ git switch develop
```

上面命令切换到 develop 分支。

如果要切换到分离状态的（ detached）的提交，需要使用 `-d`参数。

```
$ git switch -d f8c540805b7e16753c65619ca3d7514178353f39
```

上面命令切换到指定提交。

`-c`参数（ create）用于生成一个新的分支。

```
$ git switch -c new_branch
```

git tag

`git tag` □□□□□ commit □□□□ Tag □□□□□□ tag□□□ tag□□□ annotated tag □□□□ object□

```
$ git tag tmp # □□.git/refs/tags/tmp
$ git tag -a release
$ git tag -a [VERSION] -m "released [VERSION]"
```

□□□□□□□□□ commit□□□□□□□□□□□□□□□□ release□

□□□□□□□□

```
$ git tag 1.0.0
$ git push --tags

$ git tag v0.0.1
$ git push origin master --tags
```

`git tag --contains` □□□□□□□□□□□□□□

```
$ git tag --contains 4ae3003ba5
v2.36.0
v2.36.0-rc0
v2.36.0-rc1
v2.36.0-rc2
v2.36.1
v2.37.0
v2.37.0-rc0
v2.37.0-rc1
v2.37.0-rc2
```

git update-index

将工作区的文件加入缓存区域。

```
$ git update-index --add --cacheinfo \
100644 5c1b14949828006ed75a3e8858957f86a2f7e2eb hola.txt
```

直接将缓存信息插入缓存文件。

git update-ref

`git update-ref` □□□□□□□□□□□□□□□□ Git□□ ID□

```
$ git update-ref refs/heads/master 107aff
```

git write-tree

`git write-tree` □□□□□□□□□□□□□□□□□□□□□

```
$ git write-tree
```