# Git 手册

2018-10-01

朱林峰

# 目录

# 1 Git 原理

## 1.1 简介

Git 是一款代码版本控制工具，其诞生便标志着代码版本管理步入一个新阶段，从一开始的手工维护、到集中式的版本管理工具、再到现在的分布式版本管理。Git 有着优良的设计、有如许的优秀特性，使得其在版本管理领域势如破竹，独树一帜，除去少许传统软件公司还在坚持着用 SVN，绝大多数公司以及个体开发者，都已采用 git 进行版本管理。从 Linux、Android 这样的超大型工程，到个人 demo 这样的微型工程，无不在 git 的管理下得心应手，何论 GitHub 上还有着成千上万的项目。

Git 有三大特性：

1) 分布式：git 的分布式特性，一是使得多人协同开发变得轻松，二是代码多处备份难以丢失，三是各开发者可以随时随地开发并本地提交。

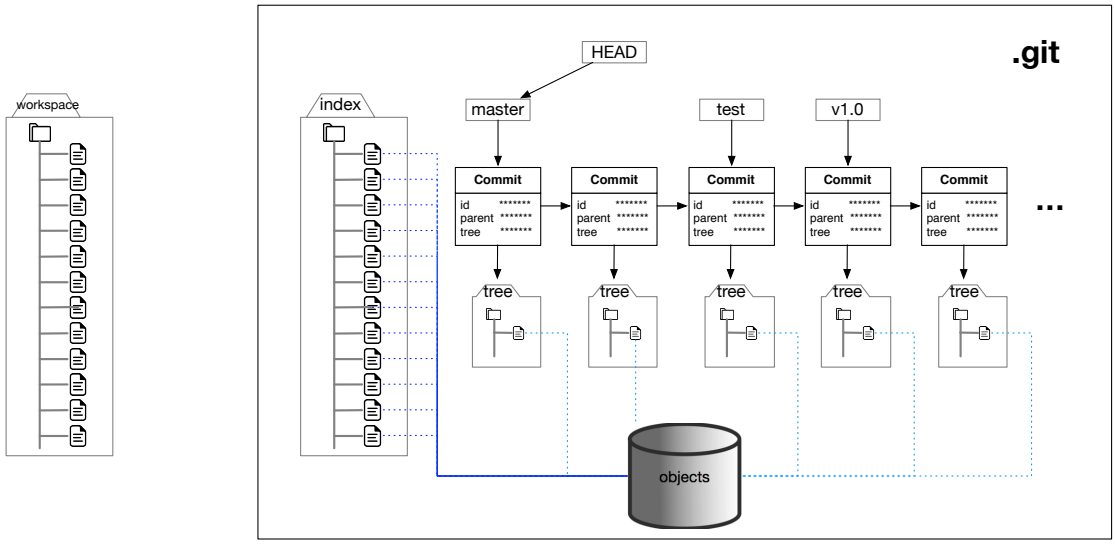
2) 灵活的分支：使用 git 可以灵活的根据需要创建分支，达到多人协同。一是各开发者可以创建自己的分支，二是可以创建特性分支，三是可以创建流程分支用于测试发布。

3) 简单快速：使用 git，掌握十条以内的命令即可上手使用，同时不管多大的项目，git 都能快速响应。

## 1.2 原理

想要掌握好 git，了解其工作原理是必经之路。同时 git 的设计思想，也值得我们欣赏借鉴；一个优秀的系统，其设计原理应该是简单，带有数学美感的。

### 1.2.1 概览




上图为一个本地 git 仓库概览图，分为两部分，一是工作区，二是版本库。工作区就是用户编写代码所在目录，版本库是 .git 目录下的全部内容。

其中版本库存储四类对象，分别为块(blob)、树(tree)、提交(commit)、标签(tag)。

1) blob：一份文件(任意类型文件)存入版本库，就是一个 blob；并且 tree 对象、commit 对象、tag 对象，也是作为 blob 存储的。blob 只记录文件的二进制内容，而忽略文件的其他属性，如文件名、读写权限、修改时间等。每个 blob 都会被计算出 sha1 哈希值，作为该 blob 的唯一索引。所有的 blob 都存储在 .git/objects 目录下，blob sha1 值的前两位作为目录名进行分目录存储。不同的文件，添加进版本库后，必然是不同的 blob，哪怕只是改动了一个字节。如果对一份文件分别进行 100 次不同的修改并分别添加进版本库，那么版本库中也应有 100 个对应的 blob。

2) tree：一个 tree 就是一份版本，是该版本下所有文件内容的一份快照，下面是真实项目中一个 tree 的部分内容:

```
100644 blob a5f9c354ef36afb0446e91051e7bec4b64fe48d0     943    .gitignore
100644 blob 4467f78d70b95670393d9646784cd7cd84a35c4b     280    Makefile
100755 blob ec02909bfac03db24b360bb2f8a458e2d6ac6ec6   94456    cseallib/lib/libsm.so
100755 blob dfdd54fd977dfd79499db7073d04d3d00f79ae00  235085    cseallib/lib/libsmwsta.so
100755 blob b7bff7ac6458f54fe4cef4977d8eb6b263fa26fe  212004    cseallib/lib/libsmwstb.so
100644 blob a7ecc523fab7051af13ad786f1addf4643ee090a    2403    cseallib/makefile
100644 blob 578aa9ac3aa0b1bd9fa1dcac25b08967ed978d02    9577    cseallib/sm_test/sm_api_test.c
100644 blob 7493b62167803f26510c8b324989eaeca5a6407a   13134    cseallib/sm_test/sm_test.c
100644 blob 22990f30a37d8dbd190408d9f7f6435a3cc072c9     756    cseallib/src/include/api.h
100644 blob 41c44957ba53545e18f59b9de1bf5edd9dbaa3dc    4238    cseallib/src/include/base64.h
```

tree 的每一项，都代表一份文件，都包含如下属性:

| 属性 | 说明 |
|---|---|
| blob id | 文件内容的索引 |
| file path | 完整的文件路径(相对于 git 仓库根目录) |
| size | 文件大小 |
| mode | 文件读写权限 |
| timestamp | 文件更新时间 |

所以 tree 是对一些 blob 的索引列表，一个 tree 即代表了一份版本的完整内容。不同的 tree，其对 blob 的引用，部分相同，部分不同。比如新的版本中只改动了 a, b, c 三个文件，其余文件未做变动，那么新生成的 tree，其索引列表中，也应只有 a, b, c 三个文件对应的索引值与上一次的 tree 不同。

3) commit：如果说一个 tree 就是一个版本，那么这些版本都是离散的，互不关联。commit 对象的作用，就是将这些版本组织起来。一个 commit 有如下属性：

| 属性 | 说明 |
|---|---|
| commit id | commit sha1 哈希值，根据 commit 结构计算而得 |
| tree id | 根据整棵树(索引列表)计算的 sha1 哈希值 |
| parent | 父提交。第一个提交没有父提交。合并节点有两个父提交。 |
| author | 作者 |
| commiter | 提交者 |
| timestamp | 提交时间 |
| message | 提交时说明 |

commit 对象有两大作用，一是保存提交信息，比如作者、提交时间、提交说明等，二是组织提交关系，也就是通过 parent 属性实现。parent 属性，使得所有单个的 commit 之间有了联系，形成一个偏序结构的有向无环图，这才使得 git 的高级特性，比如分支得以实现。

4) tag：如果打一个带附注的 tag，那么会生成一个 tag 对象，本质与 commit 差别不大，会记录 tag 时间、说明、commit id、甚至签名信息等。

以上四类是实体对象，版本库中还存在三类引用对象，分别是 branch、tag、HEAD。

1) branch：在了解 commit 的结构后，再来看 branch 就会很简单，一个 branch 就是对一个 commit 对象的引用，如果查看 .git/refs/heads 目录，会发现所有的本地分支，都在此有个对应文件。查看文件，会发现里面只有一行内容，是一个 sha1 值，它必然代表了某个对象的 id。使用 git cat-file -p <id> 命令查看该 id 的内容，会发现它就是一个 commit id。

```
git cat-file -p <id>
```

在一个分支，比如 master 分支上新生成一个提交，那么首先 commit 链会增长，同时 .git/refs/master 文件记录的 commit id 也更新为最新的 commit id，这样 master 分支永远会指向其最新的提交。

总结来看，branch 就是 commit 有向无环图上的一个向前移动的游标。

2) tag：tag 的原理和分支完全一样，只不过 tag 所指向的 commit id 不能变。查看 .git/refs/tags 目录，也一样发现每个 tag 都有对应的文件，文件里面也只有一行内容，是一个 sha1 值，这个值指向了某个 commit。

3) HEAD：HEAD 是一个更灵活的游标，可以指向分支、tag 或是任何一个 commit。当 HEAD 指向具体的 commit id 时(非分支)，就成为了“分离的头指针”(可以回忆下，存放分支的目录名为 head)。

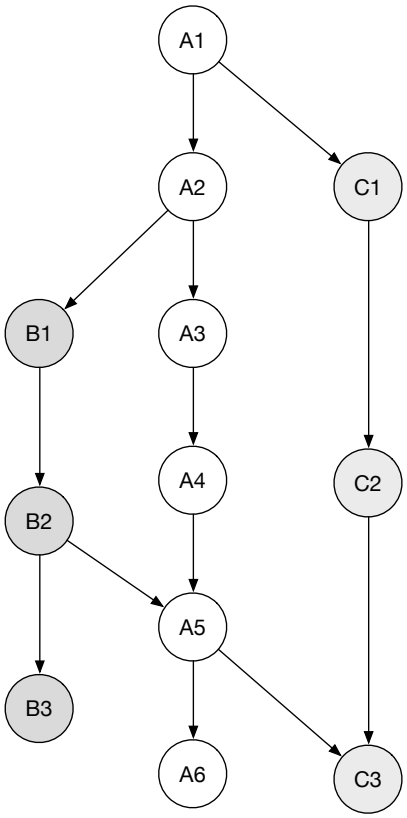
总结来说，branch、tag、HEAD 都是 commit 有向无环图上的指针，根据指针的灵活性不同，它们便有了不同的用途。其中 tag 不能移动、branch 一般只能向前移动，HEAD 能够随意

移动。补充一句，一个版本库中，可以有任意多个 branch、tag，但只有一个 HEAD。

最后说到暂存区。暂存区也是一个 tree 对象，只不过这个 tree 并未被任何 commit 引用。git 在提交时，并不是直接根据工作区生成一个 tree 以及一个 commit，而是先把工作区中需要提交的文件添加至暂存区，最后再一并把暂存区的 tree 添加至历史记录。这样做，给 git 的日常使用带来了许多的灵活性。工作区的文件，都是通过 git add 添加进暂存区时，进入到 blob 对象库的。

### 1.2.2 commit 树

前面提到了，所有的 commit 组成了一个有向无环图，也是一颗树，简单示例如下：




git 的日常使用，都是在这个图上进行游览、编辑。

1) checkout：checkout 的本质，是改变 HEAD 指针的指向，而 HEAD 指向的改变，会导致暂存区和工作区相应的被当前指向提交的内容给重置。

2) commit：提交至一个 branch，其本质就是将暂存区纳入历史提交记录，新增一个节点，branch 指向新的提交。同时 HEAD 指针也被更新。操作过程和链表新增节点一样。

3) reset：reset 操作的对象是 branch。前面提过，branch 一般只向前移动，但 reset 可以让 branch 的指针向后移动，甚至移动到任何一个历史节点。同样的，HEAD 指针也会被更新。

当一个 commit 没有被任何其他 commit 或者 tag 或者 branch 引用时，这个 commit 就是一个孤立的 commit，孤立的 commit 一方面只能通过 commit id 进行索引，另一方面 git 会定期清理版本库，孤立的 commit 最终会被清理掉。

### 1.2.3 瘦 git

git 的设计，使得其占用存储空间小而且快。

首先，不同的 tree 对象，会共享同样的 blob 对象，只要 blob 对应的文件未发生改变。也即同样的文件内容，在 git 中只会存在一份，而不是每个版本都冗余的存储一份。

其次，一份文件反复编辑并存储版本库，会导致版本库存在多个 blob，并且这些 blob 差异不大，造成额外开销。git 会定期触发对版本库的清理，一方面删除所有未被引用的对象；另一方面找到所有相似的 blob 进行压缩，压缩文件中，只有该文件的基础版本，而其他各个版本

都通过增量差异进行表示，大大减少存储开销。

只要不往 git 添加大尺寸二进制文件，git 就是高效的。

### 1.3 总结

git 的设计，分为三层。

首先，设计了基于文件内容的文件系统，所有的文件，都通过文件 Hash 进行分类和检索，同样内容的文件，也保证只存储一份。文件系统，定义好了版本系统的基石。

其次，定义了四类对象。分别是代表文件内容的 blob 对象，版本快照的 tree 对象，提交记录的 commit 对象，tag 对象。这四类对象，就定义好了版本系统的骨架，一个有向无环图。

最后，定义了分支、HEAD、tag 和暂存区，实现了对 commit 组成的有向无环图的各种编辑、游走功能，实现了灵活丰富的功能特性。

这样的设计，简单而优美，健壮而强大。

# 2 Git 命令

## 2.1 Config

配置用户名和邮箱地址，作为每次 commit 的 author。

该配置有三种选项，分别为:

1) --local: 只针对当前项目有效。

2) --global: 针对当前登录用户的所有项目有效。

3) --system: 针对系统所有用户的所有项目有效。

可以选择对三种级别的用户名和邮箱地址都进行配置，并不会发生冲突，因为这三项配置存在优先级，优先级递减。配置示例如下：

```
git config --global user.name "Max"
git config --global user.email max@example.com
```

反向操作，取消相应的配置，示例如下：

```
git config --unset --global user.name
git config --unset --global user.email
```

配置完成，可以进行查询，同样的添加不同的选项，便查询不同级别的配置，示例如下：

```
git config user.name
git config user.email
```

如果觉得一项一项的配置很繁琐，那么可以直接打开对应的 config 文件进行手动编辑：

```
git config -e               # edit local configuration
git config -e --global      # edit global configuration
git config -e --system      # edit system configuration
```

如果觉得这里提供的命令不够全面，那么建议自己输入 --help 或者 --info 选项查看手册：

```
git config --help
git config --info
```

## 2.2 Repository

### 2.2.1 本地仓库

得到本地仓库有两种途径，一是自己运行 git init 命令初始化一个仓库，二是 clone 一个现有的仓库，这两种方法都会使得你的本地文件系统得到一个 git 。而所谓 git 仓库，其实就是项目目录下的一个 .git 隐藏目录。

init 一个仓库：

```
git init <dir>            # 将某个指定目录初始化为一个 git 仓库
```

```
git init                    # 将当前目录初始化为一个 git 仓库
git init .                  # 将当前目录初始化为一个 git 仓库
```

clone 一个仓库：

```
git clone https://github.com/maxzlf/notes.git
```

clone 一个仓库时，本地会得到一个现有的 git 仓库，同时该仓库自动关联了被 clone 的远程仓库。

### 2.2.2 远程仓库

如果本地 git 仓库想要进行备份，并且与其他人一起协作开发，那么就必须关联远程仓库。远程仓库与本地仓库相同，有着 git 所有的历史提交记录，只不过远程仓库的分支与本地仓库分支不尽相同。

关联远程仓库也是有两种途径，一是通过 clone 现在的 git 仓库，自动进行了关联，二是手动通过设置 remote 进行关联。

```
git remote add <name> <url>
```

例如：

```
git remote add origin https://github.com/maxzlf/notes.git
```

远程仓库地址是一个 URL，比较长，故在设置远程仓库时，会给其取一个别名，好进行标识以及引用，这个别名只在本地仓库生效。默认的名称是 origin，当然也可以取其他名称。

一个本地仓库可以关联不止一个远程仓库。

远程仓库，可选的有三种协议，分别为 http(https)、ssh、git 协议。

如果是采用 http 协议关联远程仓库，那么每次与远程仓库交互时，都得输入账户名和密码进行认证。如果是采用 ssh 协议，则需要本地先生成 ssh 密钥对，并将公钥告诉远程 git 仓库的服务器，这样后续本地仓库与远程仓库的所有交互，都不用输入账户和密码。关于 git 协议，读者可以自行查阅文档和手册。

远程仓库，其他常用相关命令如下：

1) 同步远程仓库内容，下载远程仓库所有的更新，默认是同步 origin 仓库，如果指定远程仓库名称，则会同步指定的远程仓库。

```
git fetch
git fetch origin
git remote update    # 同步所有远程仓库的更新
```

2) 显示所有关联的远程仓库，加上 -v 选项(verbose)，能显示更详细的信息。

```
git remote
$ origin
$ out

git remote -v
$ origin    ssh://git@192.168.20.131:22022/seal/pyeseal.git (fetch)
$ origin    ssh://git@192.168.20.131:22022/seal/pyeseal.git (push)
$ out       git@gitlab.yunjingit.com:eseal/pyeseal.git (fetch)
$ out       git@gitlab.yunjingit.com:eseal/pyeseal.git (push)
```

3) 显示某个远程仓库的详细信息。

```
git remote show origin
$ * remote origin
$   Fetch URL: ssh://git@192.168.20.131:22022/seal/pyeseal.git
```

```
$   Push  URL: ssh://git@192.168.20.131:22022/seal/pyeseal.git
$   HEAD branch: master
$   Remote branches:
$     lfzhu     tracked
$     master    tracked
$     sunchaoyi tracked
$     test      tracked
$     v1.1      tracked
$     zhangyu   tracked
$   Local branches configured for 'git pull':
$     lfzhu     merges with remote lfzhu
$     master    merges with remote master
$     sunchaoyi merges with remote sunchaoyi
$     test      merges with remote test
$     v1.1      merges with remote v1.1
$     zhangyu   merges with remote zhangyu
$   Local refs configured for 'git push':
$     lfzhu     pushes to lfzhu     (up to date)
$     master    pushes to master    (up to date)
$     sunchaoyi pushes to sunchaoyi (up to date)
$     test      pushes to test      (up to date)
$     v1.1      pushes to v1.1      (up to date)
$     zhangyu   pushes to zhangyu   (up to date)
```

4) 移除某个远程仓库。

```
git remote remove <name>
```

5) 修改某个远程仓库的别名

```
git remote rename <old-name> <new-name>
```

6) 修改远程仓库的 url。前面可以看到，远程仓库有 fetch 和 push 两个 url，这两个 url 一般相同，但也可以不同，通过指定 --fetch 或 --push 选项，可以分别设置 url 地址。

```
git remote set-url <name> <url>
git remote set-url --push <name> <url>
git remote set-url --fetch <name> <url>
```

最后，所有这些配置信息，都可以在 .git/config 文件中查看并编辑。

## 2.3 Workspace & Index

操作工作区和暂存区，是 git 日常使用最为频繁的。

1) 查看工作区和暂存区的状态，加上 -s 选项用于简要显示。

```
git status
git status -s
```

小提示：建议终端用户安装 oh-my-zsh，一个华丽的 shell 外壳，对 git 支持很好，能实时显示当前所处分支以及工作区状态，还有命令行补全功能。

2) 将工作区的改动添加至暂存区，工作区的改动，可以分多批次的加入暂存区，甚至一份文件的改动，也可以分多次加入暂存区。

```
git add <file-path>    # 将某个文件添加进暂存区
git add .              # 将工作区的所有改动添加进暂存区
```

#### 2.3.1 checkout

checkout 名为检出，意思是将版本库(暂存区和历史提交记录)中的 tree 同时置换进工作区和暂存区，使得工作区显示该 tree 保存的内容。

1) 工作区中进行了修改，但后来发现这些都改无用，希望一并丢弃掉，这时可以从暂存区检出，用暂存区的内容覆盖工作区的内容。

```
git checkout .
git checkout <file-path>    # 仅仅是从暂存区检出某份文件
```

2) 检出到某个分支。

```
git checkout <branch-name>
```

3) 检出到某个提交。

```
git checkout <commit-id>
```

4) 工作区或暂存区不干净，存在修改，但强制 checkout 并放弃所有修改。

```
git checkout -f <branch-name>
git checkout -f <commit-id>
```

5) 可以不用 checkout 整个 tree，而只 checkout 一份文件。比如发现之前某个历史版本文件中的代码比现在的更好，想要拿到那份文件中的部分代码，但同时工作区中该份文件也添加了其他功能，这时候直接 checkout 那份文件就很适合。单独 checkout 文件时，HEAD 指针不变，暂存区不受影响，仅工作区受影响。

```
git checkout <commit-id/branch-name> <file-path>
```

#### 2.3.2 reset

reset 名为重置，是用于重置分支并附带暂存区的，后续有专门小节。这里因为撤销暂存区有所涉及，故略加介绍。

1) 清除暂存区。工作区的改动通过 git add 添加进暂存区后，假如后面想要放弃暂存区的内容，可以使用 reset 命令，即用 HEAD 节点的 tree 替换暂存区的 tree，导致之前暂存区临时暂存的修改全部丢失，而工作区不受影响。

```
git reset HEAD
```

2) 同时清除暂存区和工作区。本来分别使用两条命令：git reset HEAD 和 git checkout . ，便可清除暂存区和工作区中的所有修改，不过这里有更快捷的方式。

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

#### 2.3.3 commit

1) 将暂存区的内容，添加进 commit 历史记录。

```
git commit       # 将暂存区的内容提交
git commit -a    # 将工作区以及暂存区的所有内容一并提交
git commit -m "some msg" # 提交并附带简要说明信息
```

2) 追加，修正。本来良好的工作习惯是先在工作区进行开发工作，然后将想要提交的内容逐个添加进暂存区，最后将暂存区的内容提交。但也难免会发生这样的情况，即提交完毕，才发

现工作区中还有某处改动，理应一并提交，但给遗漏了；或者发现刚才提交时，写的提交说明不够准确。重新 commit 并添加 --amend 选项就可以对 HEAD 指向的提交进行修正。

```
git commit --amend
```

小提示：开发者日常使用中，有两个常见的坏习惯。一是喜欢每次提交时使用 -a 选项，看似十分便捷，把提交本应有的两步操作简化，一条命令就搞定了。但实际情况是，习惯使用该命令的开发者自己对这条命令并没有完全理解，仅仅是因为只知道这一条命令，或者仅仅是因为偷懒，才选择这么做。使用 -a 选项，会对工作区中所要提交的内容失去精确控制(实际情况往往是，工作区的一些改动，根本就不应该放入一次提交，而应属于不同的提交)。如果无脑的使用 -a 选项，会使得暂存区毫无价值。

二是提交时喜欢使用 -m 选项，进行简要的提交说明。使用该选项的开发者，往往也只是因为偷懒，不愿意为自己的提交写下恰如其分的说明。标准的做法不应该使用 -m 选项，而是进入 vim 编辑器，为该次提交撰写详细的说明，便于版本维护。提交信息里面，应详细的说明改动了哪些模块，可能会产生什么影响，以及其他信息。

有兴趣者，可以 clone 一下 GitHub 中的著名项目，如 linux, git 等，查阅别人的提交说明是怎么写的，向专业靠近。专业的开发者，不会对文档保持随意的态度。项目开发，重心不在于编码实现，而在于管理复杂度。

#### 2.3.4 ignore

项目中，一般都会存在一些文件，不希望被纳入版本库。比如操作系统自动给每个目录生成的文件、比如 IDE 生成的文件、比如运行产生的临时文件、以及依赖的一些大文件等。版本库，应尽量只跟踪由我们手工编辑生产出来的文件。

为了应对这项需求，每个项目下面，都应该有一个 .gitignore 的隐藏文件，该文件配置着哪些文件不应该被添加至版本库。

.gitignore 的规则如下:

1) .gitignore 的作用范围是其所在当前目录和子目录。意味着，一个项目，可以有不止一个 .gitignore 文件，可以给不同的目录配置不同的 .gitignore 文件。不过更常见的做法，还是在项目根目录下，配置一个全局的 .gitignore 文件。

2) .gitignore 只对未跟踪的文件有效，对已经添加到版本库的文件无效。即一个文件，比如 api.log，之前已经 git add api.log 并 commit 过，存在于版本库了，后续再在 .gitignore 文件里面对其进行忽略，就已经为时已晚。

3) 以 "#" 开始的行为注释行。

4) 通配符 "*" 匹配任意多字符。

5) 通配符 "?" 匹配一个字符。

6) [abc] 匹配可选字符范围。

7) 如果名称的最前面是一个路径分隔符(/)，表示要忽略的文件在此目录下，而非子目录的文件。

8) 如果名称的最后面是一个路径分隔符(/)，表示要忽略的是整个目录，同名文件不忽略，否则同名文件和目录都忽略。

9) 通过在名称前面添加感叹号(!)，代表不忽略该文件。

对于已经添加进版本库的文件，但是后来想要进行忽略，可以输入下面命令后，再在 .gitignore 文件中进行配置。

```
git rm --cached <file-path>
```

#### 2.3.5 diff

diff 命令用于比较版本库的不同。

1) 比较工作区和暂存区的不同。

```
git diff                    # 显示工作区和暂存区所有的不同
```

```
git diff <file-path>    # 单独比较某份文件在工作区和暂存区中的不同
```

2) 比较暂存区和 HEAD 的不同。

```
git diff --cached                 # 显示暂存区和 HEAD 所有的不同
git diff --cached <file-path>     # 单独比较某份文件在暂存区和 HEAD 中的不同
```

3) 比较工作区和 HEAD 的不同。

```
git diff HEAD                 # 显示工作区和 HEAD 所有的不同
git diff HEAD <file-path>     # 显示某份文件在工作区和 HEAD 中的不同
```

以上三条命令为最常使用的场景。根据 git 帮助手册，git diff 可以用于比较任意两个 tree 之间的不同，上面的三条命令，只是比较了三个 tree 之间的不同，分别为 workspace、index、HEAD，实际功能不止如此。

4) 比较工作区和任意 commit 之间的不同。

```
git diff HEAD [file-path]             # 比较工作区和 HEAD 的不同
git diff <branch-name> [file-path]    # 比较工作区和任意分支的不同
git diff <commit-id> [file-path]      # 比较工作区和任意 commit 的不同
```

小提示：本手册命令行中，尖括号 <> 表示的是必选参数，方括号 [] 表示的是可选参数。

5) 比较暂存区和任意 commit 之间的不同。

```
git diff --cached HEAD [file-path]
git diff --cached <branch-name> [file-path]
git diff --cached <commit-id> [file-path]
```

6) 比较其他任意 commit 之间的不同。

```
git diff <branch-a> <branch-b> [file-path]       # 比较两个分支间的不同
git diff <commit-a> <commit-b> [file-path]       # 比较任意 commit 间的不同
```

7) git diff 也可作为一个单纯的比较工具，比较任意文件之间的不同，甚至文件不属于 git 版本库，而是系统中的任何文件也可。

```
git diff --no-index -- <file-path> <file-path>
```

#### 2.3.6 stash

存在这样的场景，你正在自己的特性分支上进行开发，尚未完成，不应该生成一个 commit，但这时测试分支测出 bug 了，需要切换至测试分支排查问题并解决。贸然 checkout 到测试分支，很可能会导致当前工作区的改动会被覆盖(checkout 失败)，或者即便 checkout 成功了，但测试分支的代码也不是干净的，你把未完成的特性代码带了过来。

又或者，你忽然发现自己在错误的分支写着代码，而且写了还不少，也不应该提交至当前分支。

这种场景，正适合 stash。

1) 保存进度，临时将工作区和暂存区的改动保存至别处。

```
git stash
```

2) 查看 stash 记录，stash 记录是整个版本库唯一的，与分支无关，所有的 stash 记录按照时间先后顺序保存，并从 0 开始编号。

```
git stash list
```

3) 查看某个 stash 的改动。

```
git stash show              # 查看最近一次 stash 的改动
git stash show stash@{0}    # 查看第 0 个 stash 的改动
```

4) 恢复进度，将临时保存的改动恢复至工作区和暂存区。添加 --index 选项会同时恢复暂存区，否者暂存区进度丢失。

```
git stash pop [--index]           # 恢复最近一次 stash
git stash pop [--index] stash@{0} # 恢复第 0 个 stash
```

5) 有时候，放在 stash 的内容不想要了，可以直接丢弃。

```
git stash drop
git stash drop stash@{0}
```

6) 甚至可以将所有的 stash 一次性丢弃。

```
git stash clear
```

回到上述场景。第一种场景，先 stash 当前进度，再切到测试分支，定位并修改 bug 并提交，工作区干净了，再切回至特性分支，从 stash 恢复进度。第二种场景，先 stash 进度，再切换至正确的分支，然后从 stash 恢复进度。

## 2.4 Branch & Tag

### 2.4.1 local branch

1) 查看所有本地分支。

```
git branch [-v]
```

2) 查看所有分支，包括本地分支和远程分支。

```
git branch -a [-v]
```

3) 切换分支。

```
git checkout <branch-name>
```

4) 从当前分支创建新分支并切换过去。

```
git checkout -b <branch-name>
```

5) 从当前分支创建新分支。

```
git branch <branch-name>
```

6) 为分支改名。

```
git branch -m <old-name> <new-name>
```

7) 删除分支。如果分支包含一些没有合并过的提交，也即删除该分支会导致其中一些提交丢失，那么删除会失败。如果确认无误，执意如此，那么请使用 -D 选项强制删除。

```
git branch -d <branch-name>
git branch -D <branch-name>    # 会丢失未合并的提交
```

### 2.4.2 remote branch

1) 显示所有远程分支。

```
git branch -a
```

2) 拉取远程分支，并在本地创建关联分支。

```
git pull <remote> <branch-name>
```

3) 设置本地分支关联远程分支。

```
git branch --set-upstream-to=<remote>/<branch> <branch-name>
```

4) 跟踪某个远程分支并在本地创建对应的分支。

```
git checkout --track <remote>/<branch-name>
```

5) 跟踪某个远程分支并在本地创建对应的分支，不过本地分支可以自己取名。

```
git checkout -b <branch-name> <remote>/<branch-name>
```

6) 将本地当前分支推送至远程版本库，并在远程版本库创建远程分支，两分支自动关联。

```
git push --set-upstream <remote> <branch-name>
```

7) 提交本地分支并创建远程分支，但分支并未进行关联。

```
git push <remote> <local-branch>:<remote-branch>
```

8) 删除某远程分支。

```
git push <remote> --delete <branch-name>
git push <remote>  :<remote-branch>
```

**2.4.3 push & pull**

前面提到了本地分支、远程分支、以及本地分支和远程分支之间的关联。本地分支与远程分支，是互相独立的，它们可以有完全不同的分支名称和分支数量，但在代码拉取时，需要确认从哪个远程分支拉取至哪个本地分支；以及在代码提交时，需要确认哪个本地分支的代码提交至哪个远程分支。在分支没有进行关联时，完全可以手动在命令行里面进行指定，但这样做一是麻烦，二是容易出错，导致分支之间的关系混乱

所谓分支关联，就是让本地的某些分支与远程的某些分支产生映射关系，之后在 git pull 或者 git push 时，省去输入参数的麻烦。

1) 拉取关联分支。

```
git pull
```

2）推送至远程分支

```
git push
```

3) 从指定远程分支拉取。

```
git pull <remote> <branch-name>
```

4) 推送至指定远程分支。

```
git push <remote> <branch-name>
```

小提示：日常使用中，建议对本地分支与远程分支设置关联，并且分支名称也保持一致，避免混乱。所有的远程分支到本地分支，应该是一一映射。禁止随意的 push 至不相关联的分支或者从无关联的分支 pull。

**2.4.4 merge**

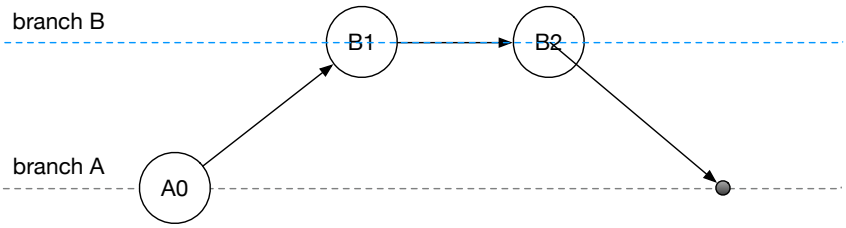
merge 名为合并，就是将两个分支的内容进行汇合。

1) 合并分支，将分支 master 的内容合并至分支 test，合并完成。

```
git checkout test    # 先切换至 test 分支
git merge master     # 合并 master 分支
```

合并不一定会产生新的 merge 节点，这得视情况而定。

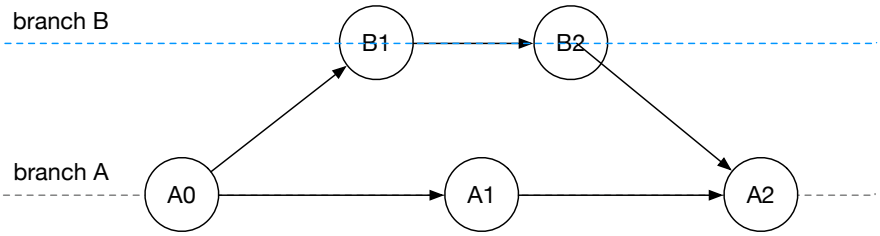
2) 不生成 merge 节点。




如图，有分支 A，在 A 的 A0 节点新建特性分支 B，之后在 B 上进行开发，并有两个提交，分别为 B1 和 B2；与此同时，分支 A 保持不变。当分支 B 上的新功能开发完毕，需要合并到 A 分支，这时候并不会产生 merge 节点。合并完成之后，分支 A 的最近三个 commit 分别为 B2, B1, A0。

因为尽管开了新的分支，但提交记录并没有分叉，仍然是线性的，分支 A 的最新提交 A0 仍然是 B2 的祖先，所以合并时，不会产生 merge 节点，也不可能发生冲突。

3) 生成 merge 节点




如图，这种情况有所不同。在 B 特性分支开发时，A 分支也在并行开发并更新，有了新的提交 A1。当 B 分支开发完毕，想要合并至 A 分支时，提交记录就有了分叉，A 分支的最新提交 A1 不是 B2 的祖先。这种情况下，合并时，有可能会有冲突，所以需要生成一个新的 merge 节点，来解决冲突，也即 A2。merge 节点 A2 与一般的 commit 节点略微不同，即它有两个 parent。

### 2.4.5 tag

tag 名为标签，也有称里程碑。当项目开发到一定阶段，比如通过 alpha 测试，通过 beta 测试等，需要做一个特殊标记以和其他 commit 进行区分。被 tag 标记的 commit，应该是一个经过充分测试的、稳定的版本，只有被 tag 标记过的版本，才应拿出来上线实用，一方面如果线上出了 bug，有个基线可以参考，另一方面如果线上崩溃，可以切换至上一次 tag 过的稳定版本。

tag 的本质，与分支无异，只不过是场景不同。分支头会随着新增提交而移动，但 tag 不会移动，它永远指向某个固定的 commit id。

1) 显示 tag 列表。

```
git tag
```

2) 创建轻量 tag，该 tag 只有一个 tag 名称。如果没有指定基于哪个 commit 创建 tag，则默认基于 HEAD 创建 tag。轻量 tag 创建容易，但是缺少相关描述信息，不知道何时和人创建的，一般不推荐使用。

```
git tag <tagname> [commit-id]
```

3) 创建带说明的 tag，会生成一个 tag 类型的 commit，记录着 tag 创建信息。

```
git tag -a <tagname>
```

```
git tag -m <message> <tagname>
```

4) 删除 tag。

```
git tag -d <tagname>
```

5) 检出 tag，tag 与分支本质无异，可以随意 checkout。

```
git checkout <tagname>
```

6) 推送 tag。

```
git push <remote> <tagname>
```

7) 拉取 tag。

```
git pull
```

8) 删除远程 tag，与删除远程分支一样。

```
git push <remote> --delete <branch-name>
git push <remote>  :<remote-branch>
```

## 2.5 **Log**

查看提交信息，主要有 git log 和 git reflog。git log 查看所有的历史提交记录，git reflog 查看 references 的变更记录，即分支、HEAD 的变更记录。git log 选项太多，下面仅列出最常用的命令，剩余的需自行查阅手册。

1) 从当前节点回溯，查看所有 commit 记录。

```
git log
```

2) 显示简要信息，仅部分提交 id 和提交说明信息。

```
git log --oneline
```

3) 过滤。

```
git log --author=<username>    # 查询某个作者的提交
git log --before <datetime>    # 显示某个时间以前的提交
git log --after  <datetime>    # 显示某个时间之后的提交
```

4) 显示某个文件的历史修改记录。

```
git log -p <file-path>
```

5) 显示每次提交的修改量。

```
git log --stat
```

6) 图形化显示。

```
git log --graph
```

7) 只显示最近 n 次提交。

```
git log -5
```

8) 根据关键字检索提交信息。

```
git log --grep <key-word>
```

9) 追责，查看某份文件每一行的修改记录。

```
git blame <file-path>
```

10) 自定义显示。

```
git log --pretty=format:"%h - %an, %ar : %s"
```

下表列出了常用的格式占位符写法及其代表的意义。

| 选项 | 说明 |
|---|---|
| %H | 提交对象（commit）的完整哈希字串 |
| %h | 提交对象的简短哈希字串 |
| %T | 树对象（tree）的完整哈希字串 |
| %t | 树对象的简短哈希字串 |
| %P | 父对象（parent）的完整哈希字串 |
| %p | 父对象的简短哈希字串 |
| %an | 作者（author）的名字 |
| %ae | 作者的电子邮件地址 |
| %ad | 作者修订日期（可以用 -date= 选项定制格式） |
| %ar | 作者修订日期，按多久以前的方式显示 |
| %cn | 提交者(committer)的名字 |
| %ce | 提交者的电子邮件地址 |
| %cd | 提交日期 |
| %cr | 提交日期，按多久以前的方式显示 |
| %s | 提交说明 |

## 2.6 Reset & Rebase

### 2.6.1 reset

Git 分支一般只向前推进，但偶尔也有向后移动的场景。比如有一回我调试测试环境的bug(本地无法调试，只能推送至测试环境调试)，一连做了二十几个提交，都只是微小的调整，最后 bug 解决了，但提交记录却乱了，实际上这二十几个提交都只是在干一件事，就是解决那个 bug，它们本应属于一个提交。使用 git reset 可以达成这个目的，将那二十几个提交，压缩成为一个更有意义的提交。也有情况是，最近几个提交的代码根本不想要了，可能是其中引入了乱糟糟的代码，导致了莫名的问题，想要从之前某个稳定的提交重新开始，需要把这最近的几个提交干掉。git reset 能丢弃任意不想要的提交，以及里面的代码。

git reset 的本质是将分支的指针，指向任意历史提交节点，结果是分支的指针发生改变，暂存区也被新指向的提交重置，但默认工作区不受影响。

1) 仅仅重置暂存区。

```
git reset HEAD    # 重置为 HEAD 指向的节点，也就是分支原地踏步，但暂存区被重置
```

2) 重置暂存区和工作区。

```
git reset --hard HEAD  # --hard 选项就是强制重置工作区，工作区中未提交的内容丢失
```

3) 重置为一个祖先节点。

```
git reset HEAD^    # 重置为当前节点的父节点
git reset HEAD~1   # 重置为当前节点的父节点
git reset HEAD^^^  # 重置为当前节点的前 3 个节点
git reset HEAD~3   # 重置为当前节点的前 3 个节点
```

```
git reset <commit-id> # 重置为任意一个 commit id 对应的节点
```

默认不带选项，会移动 branch 分支指针，重置暂存区，但工作区不变。所以一般重置完成，会发现工作区变得不干净，因为工作区已经与暂存区不一样了。这种情况，只会丢失提交记录，但不会丢失代码。

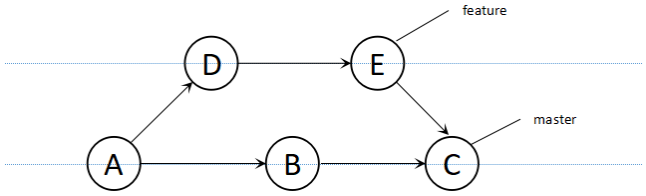
如果带 --hard 选项，则重置暂存区时同时也重置工作区，使得工作区显示的内容完全为最新 HEAD 指向的提交。那么不只是提交记录丢失了，那些提交记录里面变更的代码也一并丢失。

如果带 --soft 选项，则暂存区也不会被重置。

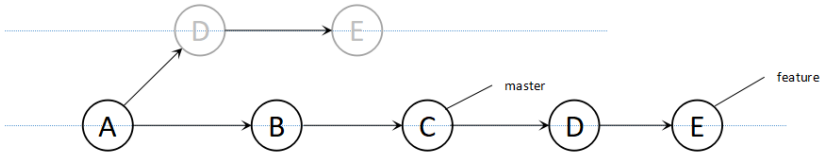
执行 git reset，并不会真的删掉提交记录，而只是使分支指针向后移动，提交记录仍然存在于版本库中。比如有分支 A-->B-->C-->D<--HEAD，reset 至节点 B，那么 C 和 D 两个节点将不被该分支索引，而如果这两个节点有被其他分支索引，或者被 tag 索引，那么这两个节点将一直存在于版本库中；否则这两个节点成为孤儿节点，指不定在什么时候就被清理掉了。

### 2.6.2 rebase

合并分支，最常使用的是 merge 命令，这也是最简单的。但 merge 也有一些弊端，首先它会自动生成大量 merge 类型的提交，这类提交与我们的手动提交不同，它没有有意义的提交说明，它增加了代码 review 的工作量；其次，让整个 commit 有向无环图变得错综复杂，提交线不够清晰明朗。所以在一些场景，比如将特性分支代码合并至主干分支，推荐使用 rebase，名为变基。




合并时，分支图如上，会生成新的 mere 节点 C，C 有两个 parent。




变基时，将 feature 分支变基至 master 分支，会先将 feature 分支 reset 至 master 分支，也即 C 节点，同时将 feature 分支中所有分叉的提交(D 和 E)临时保存。然后再按照时间先后顺序，将临时保存的分支，一个个拣选至最新的 feature 分支。如上图，rebase 完成后，feature 分支依然指向着提交节点 E，master 分支仍然指向 C 节点。整个提交线是线性的，没有分叉。

这时，如果 master 分支想要跟上最新的进度，只需与 feature 分支合并即可，这时候合并就不会产生 merge 节点了，因为两个分支的提交是线性的。

1) feature 分支 rebase 至 master 分支。

```
git rebase master    # 当前分支必须为 feature 分支
git rebase master feature   # 会先 checkout 至 feature 分支，再变基至 master
```

可见，rebase 的功能，是进行分支修剪，避免 commit 有向无环图有太多分叉，过于复杂。