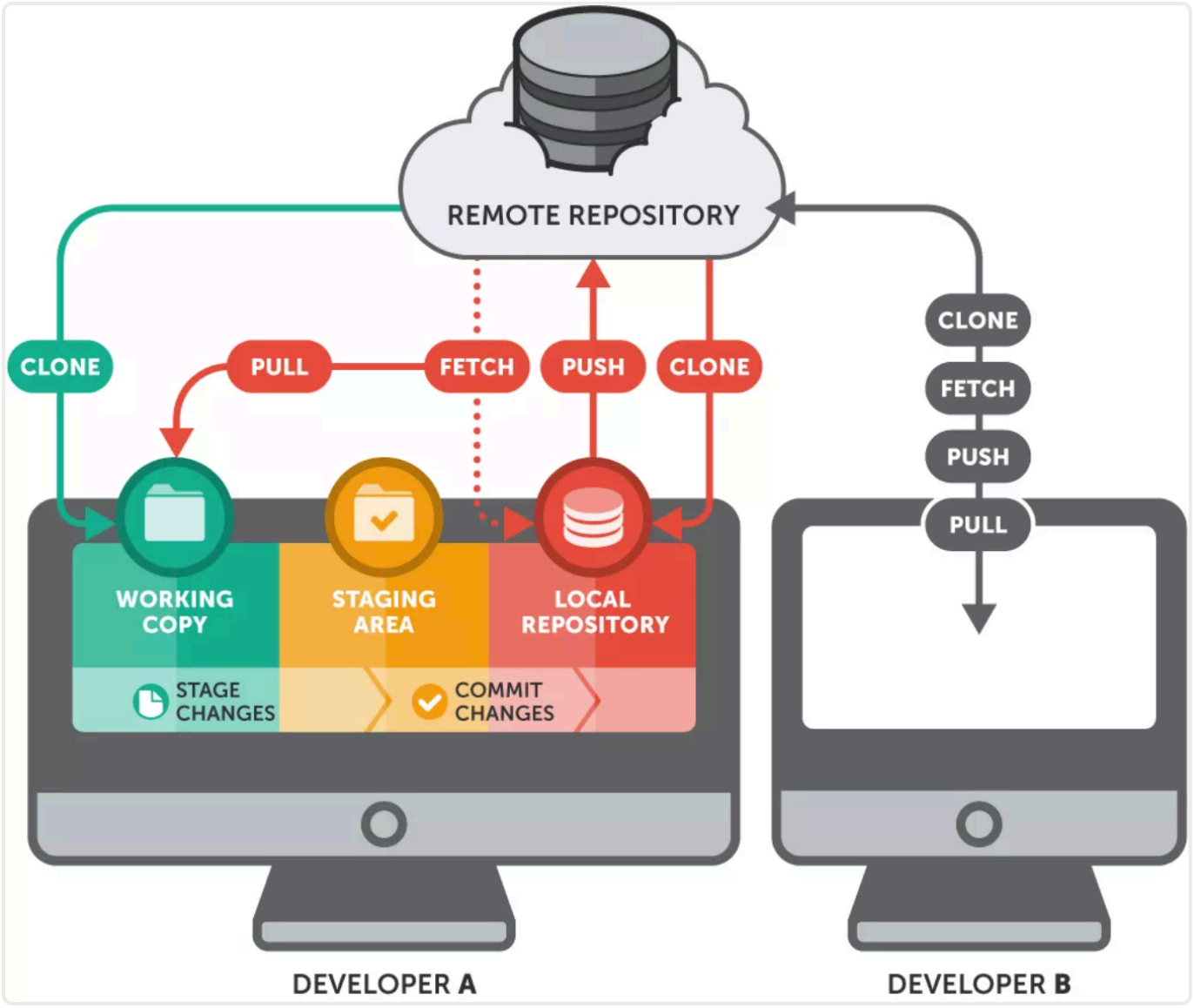
- 概述
- 版本
- 介绍
  - init
  - clone
  - config
  - remote
  - add
  - rm
  - mv
  - reset
  - clean
  - stash
  - status
  - log
  - show
  - shortlog
  - diff
  - branch
  - checkout
  - commit
  - revert
  - merge
  - rebase
  - cherry-pick
  - tag
  - push
  - fetch
  - pull
  - bisect
  - reflog
  - fsck
  - grep
  - blame
  - gc
  - submodule
- 配置
- .gitignore
- 我的 alias 设置
- 相关资料

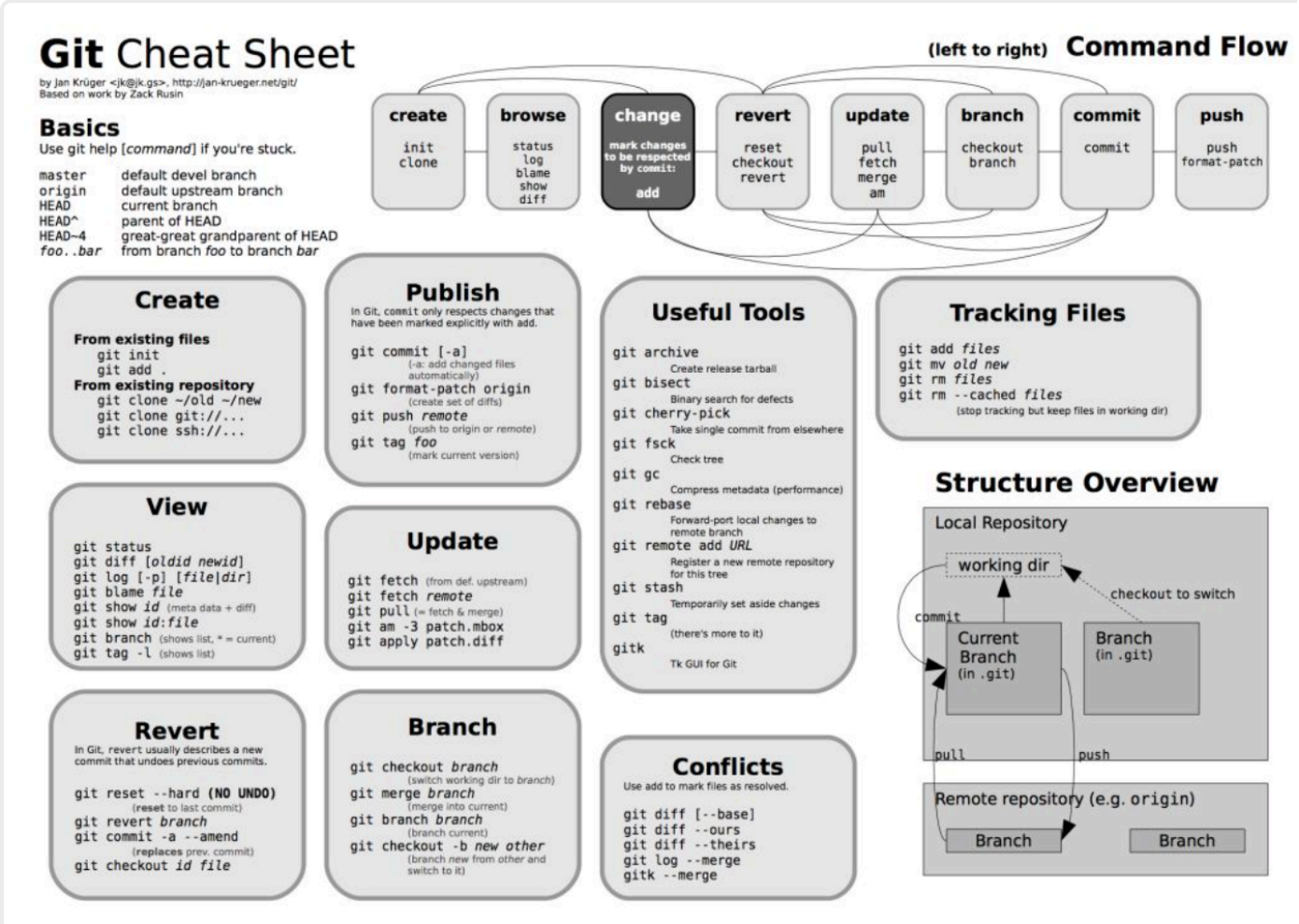
概述

git 是一个开源的分布式版本控制系统，也是计算机专业找工作的必备技能之一，其最初由 Linux Torvalds (Linux 之父) 创造，于 2005 年发布。

关于 git 的教程网上已经有很多很多，其大致工作流程如下图：



其大致命令如下图：



本文主要介绍常用的 33 条 git 命令，希望能够大家使用 git 带来一些帮助。

由于作者能力有限，描述必然会有纰漏，欢迎提交 PR、创建 Issue 进一步交流。

如果看完之后有所收获，求求给个 Star 以表支持。😊

版本

git version 2.17.2 (Apple Git-113)

介绍

以下每条命令也可使用`git <command> -h`来查看对应官方文档。

init

可以使用`git init`在当前目录创建 git 仓库，也可使用`git init <path>`在 path 路径下创建目录并创建 git 仓库。注意有个`--bare`参数可以生成一般作为远程仓库的裸仓库，其不包含工作区，具体可参考此博客。

clone

可以直接使用`git clone <url>`来克隆对应仓库代码，本地默认存储目录为当前目录下的仓库同名目录，也可在 url 之后指定本地的存储目录名。例如：

```
git clone <url>
git clone <url> <dir_name>
```

具体的 url 有四种协议格式，分别是 local，git，ssh，https 协议，具体区别可参考此博客和博客。

此外在克隆某些存在较大文件的项目时，可以使用`git clone --depth=n <url>`来下载该项目 master 分支最近 n 次 commit 的相关文件和引用；使用`git clone --single-branch --branch=dev <url>`来下载该项目 dev 分支的相关文件和引用。这样可以大大加快克隆的速度。

config

通常只用来设置邮箱和用户名。注意有三个级别`local`，`global`和`system`，分别对应项目级别，用户级别和机器级别。默认不指定是`local`级别，具体区别可参考此博客。

```
git config user.email "TXYPotato@gmail.com"
git config user.name "LebronAl"
```

此外，在单机配置多个 git 账号时，除了在目录`~/.ssh/`下生成 config 文件外，一般也需要在对应仓库重新设置非全局的正确用户名和邮箱以避免身份混淆，具体配置方式可参考此博客。

remote

通常用来管理上游仓库，可以通过`git remote -v`来查看相关信息，其相关命令如下：

```
// 添加远程仓库关联
git remote add <name> <url>
// 删除远程仓库关联
git remote remove <name>
// 更名远程仓库关联
git remote rename <old_name> <new_name>
// 显示某个远程仓库的信息
git remote show <name>
// 更新远程仓库 url
git remote set-url <name> <new_url>
```

此外，remote 命令也可以为一个远程仓库设置多个 url 地址，这样一次 push 就可以同时对该远程仓库的所有 url 进行推送，比如同时往 github 和 gitee 上推送，具体可参考此博客，相关命令如下：

```
git remote set-url --add <name> <url>
git remote set-url --delete <name> <url>
```

此外，remote 命令也可以跟踪远程的所有分支和清理无效的远程分支跟踪，具体可参考此博客：

```
// 本地创建远程追踪分支
git remote update <name>
// 本地清理无效的远程追踪分支
git remote prune <name>
// 查看无效的远程追踪分支
git remote prune --dry-run <name>
```

add

用来从工作区向暂存区添加变更。可以使用`git add -e <file>`或者`git add -p <file>`来交互性的添加工作区文件的部分变更到暂存区中去，具体可查看此博客。

add 命令支持添加单个文件，带有通配符路径的所有对应文件和所有文件。添加所有文件对应三种命令，有细微的区别，介绍如下：

```
// 提交被修改(modified)和被删除(deleted)文件，不包括新文件(new)
git add -u
// 提交当前目录下的所有变化
git add .
// 提交所有变化
git add -A
```

rm

用于删除工作区文件，并将此次删除放入到暂存区。（注：要删除的文件没有修改过，就是说和当前版本库文件的内容相同。）

```
git rm <file>
```

用于删除工作区和暂存区文件，并将此次删除放入暂存区。（注：要删除的文件已经修改过，就是说和当前版本库文件的内容不同）

```
git rm -f <file>
// 对所有文件进行操作
git rm -f -r .
```

用于删除暂存区文件，并将此次删除放入暂存区，但会保留工作区的文件。可以理解成解除 git 对这些文件的追踪，将他们转入 untracked 状态。

```
git rm --cached <file>
// 对所有文件进行操作
git rm --cached -r .
```

更多例子可参考此博客。

mv

用于移动或重命名一个文件、目录或软链接。

```
git mv <old_file> <new_file>
```

相当于`mv old_file new_file`，`git rm old_file`，`git add new_file`这三条命令一起运行，具体可参考此博客。

新文件名已经存在，若想强制覆盖则可以使用 -f 参数：

```
git mv -f <old_file> <new_file>
```

reset

用于将指定 commit 和 branch 的文件替换暂存区的文件。有三个常用参数，分别是`--hard`，`--soft`，`--mixed`，默认是`--mixed`。具体细节和应用场景可参考此博客。

定义三种动作：

1. 替换引用的指向，指向新的提交。
2. 替换暂存区。暂存区内容将和指定的提交内容一样。
3. 替换工作区。工作区和指定的提交内容一样。

```
// 执行 1。常用于合并多个 commit，类似于 squash
git reset --soft [commitId]/<branch>
// 执行 12。常用于移出暂存区的文件以作为 add 命令的反动作
git reset (--mixed) <commitId>/<branch>
// 执行 123。常用于无条件放弃本地所有变更以向远程分支同步
git reset --hard <commitId>/<branch>
// 将暂存区的所有改动撤销到工作区
git reset (--mixed) HEAD
```

clean

用来从工作目录中删除所有没有 tracked 过的文件，git clean 经常和 git reset --hard 一起结合使用。记住 reset 只影响被 track 过的文件，所以需要 clean 来删除没有 track 过的文件。结合使用这两个命令能让工作目录完全回到一个指定 的状态，具体可参考此博客。

```
// 展示哪些文件会被删除，但不真正删除文件
git clean -n
// 删除指定目录（默认当前目录）下所有没有 track 过的文件，但不会删除 .gitignore 文件里面指定的文件夹和文件
git clean -f (<path>)
// 删除指定目录（默认当前目录）下所有没有 track 过的文件，包括 .gitignore 文件里面指定的文件夹和文件
git clean -fx (<path>)
// 删除指定目录（默认当前目录）下所有没有被 track 过的文件和文件夹，但不会删除 .gitignore 文件里面指定的文件夹和文件
git clean -df (<path>)
// 删除指定目录（默认当前目录）下所有没有被 track 过的文件和文件夹，包括 .gitignore 文件里面指定的文件夹和文件
git clean -dfx (<path>)
```

stash

常用来保存和恢复工作进度。注意该命令只对被 git 跟踪的文件有效。这是一个非常有用的命令，具体相关用法可查看此博客。

```
// 保存当前工作进度，将工作区和暂存区恢复到修改之前。默认 message 为当前分支最后一次 commit 的 message
git stash
// 作用同上，message 为此次进度保存的说明
git stash save message
// 显示此次工作进度做了哪些文件行数的变化，此命令的 stash@{num} 是可选项，不带此项则默认显示最近一次的工作进度相当于 git stash show stash@{0}
git stash show stash@{num}
// 显示此次工作进度做了哪些具体的代码改动，此命令的 stash@{num} 是可选项，不带此项则默认显示最近一次的工作进度相当于 git stash show stash@{0} -p
git stash show stash@{num} -p
// 显示保存的工作进度列表，编号越小代表保存进度的时间越近
git stash list
// 恢复工作进度到工作区，此命令的 stash@{num} 是可选项，在多个工作进度中可以选择恢复，不带此项则默认恢复最近的一次进度相当于 git stash pop stash@{0}
git stash pop stash@{num}
// 恢复工作进度到工作区且该工作进度可重复恢复，此命令的 stash@{num} 是可选项，在多个工作进度中可以选择恢复，不带此项则默认恢复最近的一次进度相当于 git stash apply stash@{0}
git stash apply stash@{num}
// 删除一条保存的工作进度，此命令的 stash@{num} 是可选项，在多个工作进度中可以选择删除，不带此项则默认删除最近的一次进度相当于 git stash drop stash@{0}
git stash drop stash@{num}
// 删除所有保存的工作进度
git stash clear
```

此外有时我们只想保存部分文件，可以通过 add 一些不想保存的文件到暂存区去，然后使用`git stash --keep-index`来将工作区的工作进度保存，从而保留暂存区的进度并保存工作区的进度。具体可参考此博客。

此外虽然 stash 命令默认不包含未跟踪的文件，但我们可以通过显示声明`-u`的方式来保存未被 git 追踪的文件。即`git stash -u`。

status

用于显示文件和文件夹在工作区和暂存区的状态。

```
git status
// 展示中包含被忽略的文件
git status --ignored
```

log

用来回顾提交历史。可参考此文档和此博客。以下只列出几个常用的打印格式：

```
git log
git log --oneline
git log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit --date=relative
// 从 commit 信息中查找包含指定字符串的 commit
git log --grep <key>
// 从更改内容中查找包含指定字符串的 commit ，比如想查找哪些 commit 对代码中名为 <key> 的函数进行了更改
git log -S <key>
```

show

用来显示某个 commit 的具体提交信息和元信息。

```
git show
git show <commitId>
git show <branchName>
```

shortlog

用于汇总 git 日志输出。非常人性化的一个命令。

```
// 按照用户列出其 commit 的次数以及每次 commit 的注释
git shortlog
// 按照 commit 数量从多到少的顺序列出本仓库的贡献者并省略注释
git shortlog -sn
```

diff

用来比较文件之间的不同。具体用法可参考此博客。diff 结果的格式可参考此博客了解。

```
// 查看工作区与暂存区所有文件的变更
git diff
// 查看暂存区与最后一次 commit 之间所有文件的变更
git diff --cached
// 查看工作区与最后一次 commit 之间所有文件的变更
git diff HEAD
// 查看两次 commit 之间的变动
git diff <commitId>...<commitId>
// 查看两个分支上最后一次 commit 的内容差别
git diff <branch>...<branch>
```

branch

用于分支的操作，比如创建分支，查看分支等等。

```
// 可以查看本地分支对应的远程分支
git branch -vv
// 查看远程版本库分支列表
git branch -r
// 查看所有分支列表，包括本地和远程
git branch -a
// 在当前位置新建 name 分支
git branch <name>
// 在指定 commit 上新建 name 分支
git branch <name> <commitId>
// 强制创建 name 分支，原来的分支会被新建分支覆盖，从而迁移 name 分支指针
git branch -f <name>
// 删除 dev 分支，如果在分支中有一些未 merge 的提交则会失败
git branch -d dev
// 强制删除 dev 分支
git branch -D dev
// 重命名分支
git branch -m <oldName> <newName>
```

checkout

用于切换分支或更新工作树文件以匹配索引或指定树中的版本，具体可参考此博客。

```
// 切换到指定分支
git checkout <branch>
// 在当前位置新建分支并切换
git checkout -b <branch>
// 在指定 commit 上新建分支并切换
```

```
git checkout -b <branch> <commitId>
// 切换到指定 v1.2 的标签
git checkout tags/v1.2
// 切换到指定 commit
git checkout <commitId>
// 基于当前所在分支新建一个赤裸分支，该分支没有任何的提交历史但包含当前分支的所有内容，相当于 git checkout -b <new_branch> 和 git reset --soft <firstCommitId> 两条命令
git checkout --orphan <new_branch>
// 放弃工作区单个文件的变更，默认会从暂存区检出该文件，如果暂存区为空，则该文件会回滚到最近一次的提交状态
git checkout -- <filepath>
// 放弃工作区所有文件的变更（不包含未跟踪的）
git checkout .
```

commit

用于将暂存区里的改动给提交到本地的版本库。

```
// 提交一个描述为 message 的 commit
git commit -m "message"
// 相当于 git add -a 和 git commit -m "message" 两条命令
git commit -am "message"
// 在不增加一个新 commit 的情况下将新修改的代码追加到前一次的 commit 中，会弹出一个编辑器界面重新编辑 message 信息
git commit --amend
// 在不增加一个新 commit 的情况下将新修改的代码追加到前一次的 commit 中，不需要再修改 message 信息
git commit --amend --no-edit
// 提交一次没有任何改动的空提交，常用于触发远程 ci
git commit --allow-empty -m "message"
// 修改 commit 时间
git commit -m "message" --date=" Wed May 27 00:35:36 2020 +0800"
```

revert

撤销某个 commit 的改动。在多人协作中如果某个 commit 已经被推到了远程，此时使用 revert 相比 reset + force-update 是更优雅的撤销变更方式。可参考此博客和博客。

```
// 可以撤销指定的提交
git revert <commitId>
// 可以撤销不包括 commit1，但包括 commit2 的一串提交
git revert <commit1>...<commit2>
// revert 命令会对每个撤销的 commit 进行一次提交，--no-commit 后可以最后一起手动提交
git revert --no-commit <commit1>...<commit2>
// 退出 revert 过程，常在处理冲突出错时使用
git revert --abort
// 继续 revert 过程，常在处理完冲突时使用
git revert --continue
```

merge

用于将两个或两个以上的开发历史合并在一起的操作，具体详细命令及介绍可看此博客。

```
// 合并某个分支
git merge <branch>
// 退出 merge 过程，常在处理冲突出错时使用
git merge --abort
// 继续 merge 过程，常在处理完冲突时使用
git merge --continue
```

此外，关于在可以 fast-forward 合并时是否再提交一个 commit 有一些争议，具体可看此博客。

```
git merge --no-ff <branch>
```

此外可以在 merge 时也可以使用 --squash 参数，该命令能够将合并分支上的多个 commit 合并成一个 commit 放在当前分支上。相比 rebase 的 squash，其主要区别是该 commit 的 author 不一样。具体可查看此博客。

```
git merge --squash <branch>
```

rebase

相比 merge，合并分支历史的另一种管理方式，区别可查看此博客。

```
// 变基某个分支
git rebase <branch>
// 退出 rebase 过程，常在处理冲突出错时使用
git rebase --abort
// 继续 rebase 过程，常在处理完冲突时使用
git rebase --continue
```

此外也可以使用`git rebase -i HEAD~n`来合并 n 个 commit，具体过程可参考此博客。切记不要在共用的分支上进行 rebase，具体原因可查看此博客。

此外也可以使用`git rebase --onto <branch> <fromCommitId> <toCommitId>`来将 (fromCommitId,toCommitId] 上的所有 commit 合并到指定分支上，具体可参考此博客。其实类似于 cherry-pick 多个提交。

cherry-pick

用于将指定的 commit 应用于其他分支，具体可参考此博客。

```
// 将指定的提交应用于当前分支
git cherry-pick <commitId>
// 将指定分支的最近一次提交应用于当前分支
git cherry-pick <branch>
// 将指定的两个提交先后应用于当前分支
```

```
git cherry-pick <commitId1> <commitId2>
// 将指定范围内的多个提交(不包含 commitId1，包含 commitId2)先后应用于当前分支
git cherry-pick <commitId1>..<commitId2>
// 将指定范围内的多个提交(包含 commitId1，包含 commitId2)先后应用于当前分支
git cherry-pick <commitId1>^..<commitId2>
// 退出 cherry-pick 过程，常在处理冲突出错时使用
git cherry-pick --abort
// 继续 cherry-pick 过程，常在处理完冲突时使用
git cherry-pick --continue
```

tag

常用于发布版本的管理，是指向某个 commit 的指针，具体可查看此博客。

```
// 查看所有 tag
git tag
// 基于本地当前分支最后一个 commit 创建 tag
git tag <tagName>
// 基于指定 commit 创建 tag
git tag <tagName> <commitId>
// 基于指定 commit 创建 tag 并指定 message
git tag -a <tagName> -m "message"
// 删除本地指定 tag
git tag -d <tagName>
```

push

用于将本地版本库的分支推送到远程服务器上对应的分支，具体用法可查看此博客。

```
// 向远程推送指定分支
git push <remote> <branch>
// 向远程推送指定分支并绑定
git push -u <remote> <branch>
// 删除远程分支
git push <remote> :<branch>
// 向远程推送 tag
git push <remote> <tagName>
// 删除远程 tag
git push <remote> :<tagName>
// 向远程推送本地所有 tag
git push --tags <remote>
```

此外，当本地版本库的分支和远程版本库的对应分支有冲突时，如果想要强制覆盖，千万不要用`git push -f`，而是应该用`git push --force-with-lease`，具体原因可查看此博客。

fetch

用于从远程获取对象和引用到本地，具体可参考此博客。

```
// 取回指定远程仓库的所有分支
git fetch <remote>
// 取回指定远程仓库的指定分支
git fetch <remote> <branch>
// 取回所有远程仓库的所有分支
git fetch --all
```

pull

用于从远程获取代码并合并本地的版本，其实就是`git fetch`和`git merge FETCH_HEAD`的简写。有关其与 fetch 的区别，可以参考此博客。

```
// 相当于 fetch + merge
git pull <remote> <remote_branch>:<local_branch>
// 相当于 fetch + rebase
git pull --rebase <remote> <remote_branch>:<local_branch>
```

bisect

用于在 git 历史里面二分查找有 bug 的 commit。这是找 bug 时一个非常有用的命令，可以快速定位出引入 bug 的 commit。具体使用方式可查看此博客。

```
// 开始查找
git bisect start [endCommitId] [startCommitId]
// 版本正确
git bisect good
// 版本错误
git bisect bad
// 退出查找
git bisect reset
```

reflog

常用来恢复本地错误操作。用户每次使用更新 HEAD 的 git 命令比如 commit、amend、cherry-pick、reset、revert 等都会被记录下来（不限分支），就像 shell 的 history 一样。 这样用户如果发生误操作删除了某个 commit 找不到时，调用此命令就可以查找到对应的 commit，从而回到误删前的状态了。具体使用场景可查看此博客。

```
git reflog
```

fsck

常用来恢复本地错误操作。只要是对 HEAD 进行操作的错误，一般情况下 reflog 都能够恢复，然而有些错误并不会对 HEAD 进行操作，例如将部分代码 stash 之后又不小心 drop 或 clear 掉了，那么此时 HEAD 并没有发生变化，reflog 对此类错误是无能为力的，这个时候 fsck 就可以派上用场了，该命令会更加底层，即直接检查 git 中的 blob，tree 和 commit 对象并找到悬空的对象，找到后即可以通过 `git show <commitId>` 来查看该 commit 是否是被误操作删掉的 commit，如果是的话知道这个 commitId 无论如何都可以恢复了，不论用 merge 还是 cherry-pick 还是 checkout 等，可以参考一个恢复误删 stash 数据的修复实例，也可以进一步参考官网实例。

```
git fsck
```

grep

用于检索文件中的文本内容，更多用法可参考此博客。

```
// 检索所有包含指定关键字的文件
git grep text
// 检索关键字出现在哪一行
git grep -n text
// 统计每一个文件中检索到指定关键字的行数
git grep -c text
```

blame

用于查看某个文件的每一行内容由谁所写，类似于 IDEA 的 annotate 功能，具体可查看此博客。

```
git blame <file>
```

gc

用来清理 git 目录，删除掉无效的应用和文件，节约存储空间和传输大小。具体功能可参考此博客。

```
git gc
```

submodule

用于管理多个子项目。该命令允许一个 git 仓库作为另一个 git 仓库的子目录，并且保持父项目和子项目相互独立。具体用法可参考此博客。

配置

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置变量。这些变量存储在三个不同的位置：

/etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。如果使用带有 --system 选项的 git config 时，它会从此文件读写配置变量。

~/.gitconfig 文件：只针对当前用户。可以传递 --global 选项让 Git 读写此文件。

当前使用仓库的 Git 目录中的 config 文件（就是 .git/config）：针对该仓库。

每一个级别覆盖上一级别的配置，所以 .git/config 的配置变量会覆盖 /etc/gitconfig 中的配置变量。

.gitignore

.gitignore 文件可能从字面含义也不难猜出：这个文件里配置的文件或目录，会自动被 git 所忽略，不纳入版本控制。

在日常开发中，我们的项目经常会产生一些临时文件，如编译 Java 产生的 *.class 文件，又或是 IDE 自动生成的隐藏目录（Intellij 的 .idea 目录、Eclipse 的 .settings 目录等）等等。这些文件或目录实在没必要纳入版本管理。在这种场景下，你就需要用到 .gitignore 配置来过滤这些文件或目录。

配置的规则很简单，也没什么可说的，看几个例子，自然就明白了。

这里推荐一下 Github 的开源项目。在这里，你可以找到很多常用的模板，如：Java、Nodejs、C++ 的 .gitignore 模板等等。

一般有两个级别的 .gitignore 文件：一个是全局的 ~/.gitignore 文件，另一个是仓库目录下的 .gitignore 文件。前者会覆盖后者的过滤条件，因此一个项目具体有哪些文件会被 git 忽略与两个文件都有关。我们可以将 .idea，*.class 这种常有的过滤目录添加到全局的 ~/.gitignore 中，这样子就不用在每个仓库的 .gitignore 中再次添加了。

我的 alias 设置

```
a = add
aa = add -A
au = add -u
ae = add -e
ap = add -p
bm = blame
br = branch -vv
bra = branch -a
brd = branch -D
brf = branch -f
brm = branch -m
bs = bisect
bss = bisect start
bsb = bisect bad
bsg = bisect good
bsr = bisect reset
cf = config
cfl = config --list
cfg = config --global
cfgl = config --global --list
cfs = config --system
cfsl = config --system --list
cl = clone
clb = clone --single-branch --branch
cld = clone --depth
cn = clean
```

```
cnn = clean -n
cnf = clean -f
cnfx = clean -fx
cndf = clean -df
cndfx = clean -dfx
cm = commit
cmm = commit -m
cmme = commit -m --allow-empty
cma = commit --amend
cman = commit --amend --no-edit
co = checkout
cob = checkout -b
cod = checkout --
coda = checkout .
coo = checkout --orphan
cp = cherry-pick
cpa = cherry-pick --abort
cpc = cherry-pick --continue
df = diff
dfc = diff --cached
dfh = diff HEAD
fh = fetch
fha = fetch --all
fc = fsck
ge = grep -n
gec = grep -c
it = init
lg = log
lgo = log --oneline
lgga = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
%C(bold blue)<%an>%Creset' --abbrev-commit --date=relative
lgge = log --grep
lgs = log -S
me = merge
mea = merge --abort
mec = merge --continue
men = merge --no-ff
mvf = mv -f
pl = pull
plr = pull --rebase
ps = push
psu = push -u
psf = push --force-with-lease
pst = push --tags
rb = rebase
rba = rebase --abort
rbc = rebase --continue
rbi = rebase -i
rbo = rebase --onto
```

```
rf = reflog
rmc = rm --cached
rmca = rm --cached -r .
rmf = rm -f
rmfa = rm -f -r .
rs = reset
rse = reset HEAD
rsh = reset --hard
rss = reset --soft
rv = revert
rva = revert --abort
rvc = revert --continue
rvn = revert --no-commit
rt = remote
rta = remote add
rtp = remote prune
rtpd = remote prune --dry-run
rtrm = remote remove
rtrn = remote rename
rtsu = remote set-url
rtsua = remote set-url --add
rtsud = remote set-url --delete
rts = remote show
rtu = remote update
rtv = remote -v
sh = stash
shu = stash -u
shk = stash --keep-index
sha = stash apply
shc = stash clear
shd = stash drop
shl = stash list
shp = stash pop
shsh = stash show
shshp = stash show -p
shsa = stash save
shsak = stash save --keep-index
so = show
sl = shortlog
slsn = shortlog -s -n
st = status --show-stash
sti = stash ---ignored --show-stash
tg = tag
tga = tag -a
tgd = tag -d
```

相关资料

- githug

- learnGitBranching
- Git原理入门解析
- Git 从入门到精通，这一篇就够了