Git 全局设置

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：

1. `/etc/gitconfig` 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果在执行 `git config` 时带上 `--system` 选项，那么它就会读写该文件中的配置变量。 （由于它是系统配置文件，因此你需要管理员或超级用户权限来修改它。）
2. `~/.gitconfig` 或 `~/.config/git/config` 文件：只针对当前用户。 你可以传递 `--global` 选项让 Git 读写此文件，这会对你系统上 **所有** 的仓库生效。
3. 当前使用仓库的 Git 目录中的 `config` 文件（即 `.git/config`）：针对该仓库。 你可以传递 `--local` 选项让 Git 强制读写此文件，虽然默认情况下用的就是它。。 （当然，你需要进入某个 Git 仓库中才能让该选项生效。）

```
git config --global user.name "x"
git config --global user.email "xxx@qq.com"

# 查看
git config --list
```

获取帮助

若你使用 Git 时需要获取帮助，有三种等价的方法可以找到 Git 命令的综合手册（manpage）：

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

例如，要想获得 `git config` 命令的手册，执行

```
$ git help config
```

这些命令很棒，因为你随时随地可以使用而无需联网。 如果你觉得手册或者本书的内容还不够用，你可以尝试在 Freenode IRC 服务器 https://freenode.net 上的 `#git` 或 `#github` 频道寻求帮助。 这些频道经常有上百人在线，他们都精通 Git 并且乐于助人。

此外，如果你不需要全面的手册，只需要可用选项的快速参考，那么可以用 `-h` 选项获得更简明的 “help” 输出：

忽略文件

```
在GIT项目中可以在项目根目录添加**.gitignore**文件的方式，规定相应的忽略规则，用来管理当前项目中的文件的忽略行为。.gitignore 文件是可以提交到公有仓库中，这就为该项目下的所有开发者都共享一套定义好的忽略规则。在.gitignore 文件中，遵循相应的语法，在每一行指定一个忽略规则。
.gitignore忽略规则简单说明
file            表示忽略file文件
*.a             表示忽略所有 .a 结尾的文件
!lib.a          表示但lib.a除外
build/          表示忽略 build/目录下的所有文件，过滤整个build文件夹;
```

创建创库

```
当着手于一个新的仓库时，你只需创建一次。要么在本地创建，然后推送到 GitHub；要么通过 clone 一个现有仓库。
$ git init
git init [name]
在使用过 git init 命令后，使用以下命令将本地仓库与一个 GitHub 上的空仓库连接起来：
$ git remote add origin <url>
将现有目录转换为一个 Git 仓库

$ git clone <url> [name]
Clone（下载）一个已存在于 GitHub 上的仓库，包括所有的文件、分支和提交(commits)

# 创建 git 仓库
mkdir gittest01
cd gittest01
git init
touch README.md
git add README.md
git commit -m "first commit"
git remote add origin https://gitee.com/zmjzhong/gittest01.git
git push -u origin "master"

# 已有仓库?
cd existing_git_repo
git remote add origin https://gitee.com/zmjzhong/gittest01.git
git push -u origin "master"
```

基本操作

git status

```
# 状态
git status
```

git add

```
# 添加到暂存区
use "git add <file>..." to include in what will be committed
git add <[file]|.|*>
```

git commit

```
git commit
git commit -a -m ''
# 最终你只会有一个提交—第二次提交将代替第一次提交的结果。
git commit --amend
```

git rm

```
# 从工作区和暂存区删除
git rm
git rm -f  <file>...

# 从暂存区删除
# use "git rm --cached <file>..." to unstage
git rm --cached <file>...

```

git mv

其实，运行 `git mv` 就相当于运行了下面三条命令：

```
$ mv README.md README
$ git rm README.md
$ git add README
```

git log

```
# 查看提交记录
git log
git log --oneline
git log --pretty=oneline
git reflog
```

git rest

```
# 取消暂存
# use "git reset HEAD <file>..." to unstage
git reset head <file>

# git reset 命令用于回退版本，可以指定退回某一次提交的版本。
# 命令语法格式如下：
git reset [--soft | --mixed | --hard] [HEAD]
--soft
#当你将它 reset 回 HEAD~（HEAD 的父结点）时，其实就是把该分支移动回原来的位置，而不会改变索引和工作目录。
$ git reset --soft HEAD~3 # 回退上上上一个版本

--mixed
# 为默认，可以不用带该参数，用于重置暂存区的文件与上一次的提交(commit)保持一致，工作区文件内容保持不变。
git reset [HEAD]
# 实例：
$ git reset HEAD^              # 回退所有内容到上一个版本
$ git reset HEAD^ hello.php  # 回退 hello.php 文件的版本到上一个版本
$ git reset 052e            # 回退到指定版本
git reset file.txt # (这其实是 git reset --mixed HEAD file.txt 的简写形式)

--hard
# 参数撤销工作区中所有未提交的修改内容，将暂存区与工作区都回到上一次版本，并删除之前的所有信息提交：
git reset --hard HEAD
# 实例：
$ git reset -hard HEAD~3  # 回退上上上一个版本
$ git reset -hard bae128  # 回退到某个版本回退点之前的所有信息。
$ git reset --hard origin/master     # 将本地的状态回退到和远程的一样

注意：谨慎使用 -hard 参数，它会删除回退点之前的所有信息。
HEAD 说明：
HEAD 表示当前版本
HEAD^ 上一个版本
HEAD^^ 上上一个版本
HEAD^^^ 上上上一个版本
以此类推...
可以使用 ~数字表示
HEAD~0 表示当前版本
HEAD~1 上一个版本
HEAD^2 上上一个版本
HEAD^3 上上上一个版本
以此类推...
```

git checkout

```
# 当执行 git checkout . 或者 git checkout -- <file> 命令时，会用暂存区全部或指定的文件替换工作
```

```
区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。撤消对文件的修改
# 当执行 git checkout HEAD . 或者 git checkout HEAD <file> 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

# checkout将暂存区或者某个commit点文件恢复到工作区
git checkout [commit] -- [file]
# e.g. 将a.jpg文件恢复,不写commit表示恢复最新保存的文件内容
git checkout  --  a.jpg

$ git checkout -b iss53
Switched to a new branch "iss53"
它是下面两条命令的简写:
$ git branch iss53
$ git checkout iss53
```

总结

希望你现在熟悉并理解了 `reset` 命令，不过关于它和 `checkout` 之间的区别，你可能还是会有点困惑，毕竟不太可能记住不同调用的所有规则。

下面的速查表列出了命令对树的影响。 “HEAD” 一列中的 “REF” 表示该命令移动了 HEAD 指向的分支引用，而 “HEAD” 则表示只移动了 HEAD 自身。 特别注意 *WD Safe?* 一列——如果它标记为 **NO**，那么运行该命令之前请考虑一下。

	HEAD	Index	Workdir	WD Safe?
Commit Level				
`reset --soft [commit]`	REF	NO	NO	YES
`reset [commit]`	REF	YES	NO	YES
`reset --hard [commit]`	REF	YES	YES	**NO**
`checkout <commit>`	HEAD	YES	YES	YES
File Level				
`reset [commit] <paths>`	NO	YES	NO	YES
`checkout [commit] <paths>`	NO	YES	YES	**NO**

git remote

```
$ git remote

origin
```

```
$ git remote -v
origin  https://github.com/schacon/ticgit (fetch)
origin  https://github.com/schacon/ticgit (push)
```

运行 `git remote add <shortname> <url>` 添加一个新的远程 Git 仓库

如果想要查看某一个远程仓库的更多信息，可以使用 `git remote show <remote>` 命令

删除远程仓库
`git remote remove name` 或 `git remote rm name`

修改仓库名
`git remote rename old_name new_name`

git restore

```
# 撤销工作区的更改
# use "git restore <file>..." to discard changes in working directory
git restore <file>...

# use "git restore --staged <file>..." to unstage
git restore --staged <file>...
```

git tag

```
# tag标签：在项目的重要commit位置添加快照，保存当时的工作状态，一般用于版本的迭代。
git  tag  [tag_name] [commit_id] -m  [message]
说明：commit_id可以不写则默认标签表示最新的commit_id位置，message也可以不写，但是最好添加。

e.g. 在最新的commit处添加标签v1.0
git tag v1.0 -m '版本1'

* 查看标签
 git tag  查看标签列表
 git show [tag_name]  查看标签详细信息
* 去往某个标签节点
git reset --hard [tag]
* 删除标签
git tag -d  [tag]
```

git stash

```
现在想要切换分支，但是还不想要提交之前的工作；所以贮藏修改。 将新的贮藏推送到栈上，运行 git stash 或
```

```
git stash push

此时，你可以切换分支并在其他地方工作；你的修改被存储在栈上。 要查看贮藏的东西，可以使用 git stash list：
$ git stash list 说明:最新保存的工作区在最上面
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log

应用某个工作区
选择工作区之前，必须保证当前工作区是干净的
git stash apply  [stash@{n}]

删除工作区
git stash drop [stash@{n}]  删除某一个工作区
git stash clear  删除所有保存的工作区
```

git branch

```
# branch
git branch  #列出分支
创建一个新分支
$ git branch [branch-name]
切换分支
$ git checkout [branch-name]
切换到指定分支并更新工作目录(working directory)
git checkout -b [branck-name]
$ git merge [branch]
将指定分支的历史合并到当前分支。这通常在拉取请求(PR)中完成，但也是一个重要的 Git 操作。
$ git branch -d [branch-name]
删除指定分支
git merge <branch-name> #合并到当前分支

如果想要查看设置的所有跟踪分支，可以使用 git branch 的 -vv 选项。 这会将所有的本地分支列出来并且包含更多的信息，如每一个分支正在跟踪哪个远程分支与本地分支是否是领先、落后或是都有。

$ git branch -vv
  iss53     7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master    1ae2a45 [origin/master] deploying index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing   5ea463a trying something new
这里可以看到 iss53 分支正在跟踪 origin/iss53 并且 “ahead” 是 2，意味着本地有两个提交还没有推送到服务器上。 也能看到 master 分支正在跟踪 origin/master 分支并且是最新的。 接下来可以看到 serverfix 分支正在跟踪 teamone 服务器上的 server-fix-good 分支并且领先 3 落后 1， 意味着服务器上有一次提交还没有合并入同时本地有三次提交还没有推送。 最后看到 testing 分支并没有跟踪任何远程分支。

跟踪分支
从一个远程跟踪分支检出一个本地分支会自动创建所谓的“跟踪分支”（它跟踪的分支叫做“上游分支”）。 跟踪分支是与远程分支有直接关系的本地分支。 如果在一个跟踪分支上输入 git pull，Git 能自动地识别去哪个服务器上抓取、合并到哪个分支。
```

```
当克隆一个仓库时，它通常会自动地创建一个跟踪 origin/master 的 master 分支。 然而，如果你愿意的话可以设置其他的跟踪分支，或是一个在其他远程仓库上的跟踪分支，又或者不跟踪 master 分支。 最简单的实例就是像之前看到的那样，运行 git checkout -b <branch> <remote>/<branch>。 这是一个十分常用的操作所以 Git 提供了 --track 快捷方式：

$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
由于这个操作太常用了，该捷径本身还有一个捷径。 如果你尝试检出的分支 (a) 不存在且 (b) 刚好只有一个名字与之匹配的远程分支，那么 Git 就会为你创建一个跟踪分支：

$ git checkout serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
如果想要将本地分支与远程分支设置为不同的名字，你可以轻松地使用上一个命令增加一个不同名字的本地分支：

$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
现在，本地分支 sf 会自动从 origin/serverfix 拉取。

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改正在跟踪的上游分支， 你可以在任意时间使用 -u 或 --set-upstream-to 选项运行 git branch 来显式地设置。

$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

git merge

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy

# 来简单地退出合并
git merge --abort
```

git pull

```
git pull 命令用于从远程获取代码并合并本地的版本。
git pull 其实就是 git fetch 和 git merge FETCH_HEAD 的简写。 命令格式如下：
git pull <远程主机名> <远程分支名>:<本地分支名>
实例
更新操作：
$ git pull
$ git pull origin
将远程主机 origin 的 master 分支拉取过来，与本地的 brantest 分支合并。
git pull origin master:brantest
如果远程分支是与当前分支合并，则冒号后面的部分可以省略。
git pull origin master
```

git fetch

```
就如刚才所见，从远程仓库中获得数据，可以执行：
$ git fetch <remote>
这个命令会访问远程仓库，从中拉取所有你还没有的数据。 执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。
```

git push

```
git push 命用于从将本地的分支版本上传到远程并合并。
命令格式如下：
git push <远程主机名> <本地分支名>:<远程分支名>如果本地分支名与远程分支名相同，则可以省略冒号：
git push <远程主机名> <本地分支名>
实例
以下命令将本地的 master 分支推送到 origin 主机的 master 分支。
$ git push origin master
相等于：
$ git push origin master:master
如果本地版本与远程版本有差异，但又要强制推送可以使用 --force 参数：
git push --force origin master
删除主机但分支可以使用 --delete 参数，以下命令表示删除 origin 主机的 master 分支：
git push origin --delete master

git push origin [tag]  推送一个本地标签到远程
git push origin --tags  推送所有本地所有标签到远程
git push origin --delete tag  [tagname]  删除向远程仓库推送的标签
```

大文件上传

> 安装 Git LFS

根据系统环境将 `git - lfs` 安装到本机。

> Windows

```
# 下载安装 Windows installer
https://github.com/github/git-lfs/releases

# 运行 windows installer
```

> 开启 LFS 功能

首先，到达指定 `git` 项目文件夹，执行安装命令，开启 `lfs` 功能。

```
$ cd xxx

# 只需执行一次即可，即可开启 lfs 功能
$ git lfs install
复制代码
```

其次，选择您希望 `Git LFS` 管理的文件类型，默认会生成一个 `.gitattributes` 文件。

```
# 此处建议此种格式，可以统一关联 .zip 类型的文件（具体文件类型，视项目而定）
$ git lfs track "* .zip"
```

解决中文显示的问题

```
# 第一步
git config --global core.quotepath false
# 第二步
在git bash的界面中右击空白处，Options->Text->Locale改为zh_CN，Character set改为UTF-8
```

统计代码行

```
git统计代码行数命令
git log --author="zwx996578" --pretty=tformat: --numstat | gawk '{ add += $1 ; subs += $2 ; loc += $1 - $2 } END { printf "增加行数:%s;删除的行数:%s;总行数:%s\n",add,subs,loc }'
git log --author="zwx996578" --since='2023-03-01' --until='2023-12-31' --pretty=tformat: --numstat | gawk '{ add += $1 ; subs += $2 ; loc += $1 - $2 } END { printf "增加行数:%s;删除的行数:%s;总行数:%s\n",add,subs,loc }'

git统计代码行数命令
git log --author="zhongmajun WX996578" --since='2022-01-01' --until='2022-12-31'--pretty=tformat: --numstat | gawk '{ add += $1 ; subs += $2 ; loc += $1 - $2 } END { printf "增加行数:%s 删除的行数:%s 总行数: %s\n",add,subs,loc }'
git log --author="zhongmajun WX996578"  --pretty=tformat: --numstat | awk '{ add += $1 ; subs += $2 ; loc += $1 - $2 } END { printf "增加行数:%s;删除的行数:%s;总行数:%s\n",add,subs,loc }'

```