# 目录



# 常见缩写

```
-d  --delete：删除

-D  --delete --force的快捷键

-f  --force：强制

-m  --move：移动或重命名

-M  --move --force的快捷键

-r  --remote：远程

-a  --all：所有
```

# 安装 git

> 已有？请跳过

- git 文档地址
- git 下载地址
- 查看 git 版本

```
git --version
```

# 初始化

## 基本配置

### 配置帐号信息

```
# 设置 [本仓库 | 全局 | 系统 ] 的 用户名称
$ git config [--local | --global | --system] user.name 'Your name'
# 设置 [本仓库 | 全局 | 系统 ] 的 用户邮箱
$ git config [--local | --global | --system] user.email 'Your email'
```

> 当然你也可以通过修改 .git 文件目录下面的 config 文件进行修改

### 配置别名

```
# 配置别名
$ git config --global alias.st status # git st
$ git config --global alias.co checkout # git co
$ git config --global alias.br branch # git br
$ git config --global alias.ci commit # git ci
```

### 查看配置

```
# 查看配置
git config --list [--local | --global | --system]
# 根据git哈希值 查看内容 / 查看类型
git cat-flie [-p / -t]
```

### 参数说明

- local：区域为本仓库
- global: 当前用户的所有仓库
- system: 本系统的所有用户

> 缺省等同于 local
> 优先级：local > global > system

## 生成 shh 密钥

> 如果你想通过 https 的方式 每次提交输入密码，当然可以跳过此节

- 检查是否已经存在公私钥

```
# 查看.ssh目录下是否有密钥 如列表中包含 id_rsa和id_rsa.pub 说明电脑中已经存在公私钥
ls -al ~/.ssh

# cat ~/.ssh/id_rsa.pub
```

- 生成密钥

```
# 生成秘钥（根据电脑主机）
$ ssh-keygen -t rsa

# 根据邮箱生成 密钥
$ ssh-keygen -t rsa -C "youremail"
# 执行后 ： 建议不要输入，一路回车，
```

> 生成的 id_rsa 是私钥 生成的 id_rsa.pub 是公钥

- 添加 密钥

在对应服务器增加密钥。【github -> setting -> SSH and GPG keys】

> 如果公司是自己搭建的 git 服务（未使用 gitlab），则交给管理员添加。如果使用 gitlab 在设置增加密钥

# 获取 Git 仓库

## 在现有目录中初始化仓库

```
# 初始化本地仓库 缺省 为当前目录
$ git init ['your_project'/缺省]
# git init --bare 建立裸仓库【中心仓库】
# 获取状态
$ git status
# .或*代表全部添加
$ git add [file1] [file2] ...
# 提交
$ git commit -m "提交message"
# 添加远程源
$ git remote add origin [远程地址]
# push 同时 设置默认跟踪分支
$ git push -u origin master
```

## 克隆现有的仓库

```
# 克隆远程仓库到本地  fileName 的文件夹内 【缺省-默认远端名称】
$ git clone [url] <fileName/缺省>
# 克隆 为 裸仓库【中心仓库】
$ git clone --bare [ 连接地址 ] < file bf名称 >
```

> 从裸仓库 clone 下来的本地仓库可以进行正常的 push 操作， 但是从一般仓库 clone 下来的本地仓库却不行。 这也正是裸仓库存在的意义。 裸仓库一般情况下是作为远端的中心仓库而存在的。

### 本地已有构建的项目

```
$ git remote -v #查看远程版本库信息
$ git remote add github <url> #添加github远程版本库
$ git fetch github #拉取远程版本库
$ git merge -h #查看合并帮助信息
$ git merge --allow-unrelated-histories github/master
# or git pull origin master --allow-unrelated-histories
# 对github上的master分支与本地master分支进行合并（两分支不是父子关系，所以合并需要添加 --allow-unr
$ git push github #推送同步到github仓库
```

## 不进行版本控制

### 添加不进行版本控制的文件目录

> 项目目录下 新建 .gitignore 文件， 写入不需要进行版本控制的文件名或文件夹

```
# 文件示例
node_modules # 对 node_modules文件夹及其文件 及 node_modules文件 不进行版本控制

doc    # 对 [doc文件夹及其子文件 和 名称为doc的文件] 不进行版本控制
*doc # 不允许 任何 包含doc字符的 文件夹及其子文件 和 文件（包括.扩展名）不进行版本控制

doc/   # 对 [doc文件夹及其下文件] 不进行版本控制
*doc/  # 对 [任何 包含doc字符的文件夹及其下文件] 不进行版本控制

*.md // 对 .md结尾的文件 不进行版本控制

# /* 复合示例 */
doc
!doc/* # git管doc文件夹，不管doc文件
```

### 提交 **commit** 后，想再忽略一些已经提交的文件

1. 把忽略的文件添加到 .gitignore;
2. 通过 git rm -- cached < file > 的方式删除掉 git 仓库里面无需跟踪的文件。

## 常见操作

## 查看信息

## 查看帮助 [ git help ]

```
# 所有可用的命令都将打印在标准输出上
$ git help [--all/-a]

# 在标准输出中也会列出有用的Git指南
$ git help [--guide/-g]

# 显示 git 手册页
$ git help git
```

## 查看状态 [ git status ]

> git status 命令用于显示工作目录和暂存区的状态。git status 不显示已经 commit 到项目历史中去的信息。看项目历史的信息要使用 git log

常用于 git commit 之前, 这样能防止你不小心提交了您不想提交的东西。

```
# 显示工作目录和暂存区的状态
$ git status

# 只列出所有已经被git管理的且被修改但没提交的文件
$ git status -uno
# 紧凑的格式输出
$ git status -s # or $ git status --short
```

## 查看变化 [ git diff ]

> 对比 修改之后还没有暂存起来的内容变化

```
# 工作树中的更改尚未分段进行下一次提交
$ git diff
# 比较当前文件和暂存区文件差异 git diff
$ git diff <file>

$ git diff <id1><id1><id2> # 比较两次提交之间的差异

$ git diff <branch1> <branch2> # 在两个分支之间比较
$ git diff --staged # 比较暂存区和版本库差异

$ git diff --cached # 比较暂存区和版本库差异

$ git diff --stat # 仅仅比较统计信息

$ git diff HEAD # 显示工作版本(Working tree)和HEAD的差别

$ git diff test # 查看当前目录和另外一个分支(test)的差别


# 假定：HEAD、缓存区、工作区中的readme.md文件内容均不相同。
# 比较 工作区 <===> HEAD
$ git diff HEAD -- readme.md
#  比较 工作区 <===> 缓存区
$ git diff -- readme.md
# 比较 缓存区 <===> HEAD
$ git diff --cached -- readme.md
# 比较两个提交或分支的差异
$ gti diff [分支名称/commit-id] [分支名称/commit-id] --< file >
# 对比两个版本的不同
$ git diff [commit号] [commit号]

# 对比 当前提交版本与上次提交
$ git diff HEAD HEAD^[同 HEAD～1]

# 对比当前提交版本与上上次（前两次）提交
$ git diff HEAD HEAD^^[同HEAD～2]
```

补充说明：

1. 一个节点，可以包含多个子节点（checkout 出多个分支）
2. 一个节点可以有多个父节点（多个分支合并）
3. 是~都是父节点，区别是跟随数字时候，2 是第二个父节点，而~2 是父节点的父节点
4. ^和~可以组合使用,例如 HEAD~2^2

## 查看日志 [ git log ]

```
# 查看所有分支的历史 git log 等同于它
$ git log --all
# 查看图形化的 log 地址
$ git log --all --graph
# 查看单行的简洁历史。
$ git log --oneline
# 查看最近的四条简洁历史。
$ git log --oneline -n4
#查看所有分支最近 4 条单行的图形化历史。
$ git log --oneline --all -n4 --graph
# 跳转到git log 的帮助文档网页
$ git help --web log

$ git log -1 // 最近一次提交信息
```

## 查看某个文件的版本历史 **[ git show ]**

```
// 先查看文件提交历史
git log --pretty=oneline [文件名或文件路径] // 例如src/AfterView/common/commonStream.vue

git show [版本号] // 显示具体的某次的改动的修改
```

## 图形化查看提交内容 **[ gitk ]**

```
# 在当前目录下输入，弹出图形化界面
$ gitk

$ gitk --all
```

> 定制化图形界面： view --> new view [勾选 all refs] # 显示全部分支

# 更新

- 使用 git pull 进行更新

```
# 按照git branch 设置的默认跟踪的服务器和分支来拉取
$ git pull
# 实例：拉取远程服务器origin的master分支
$ git pull origin master
```

- 使用 git fetch 进行更新

```
# 将远程仓库的master分支下载到本地当前branch中
$ git fetch orgin master

# 比较本地的master分支和origin/master分支的差别
$ git log -p master  ..origin/master

# 进行合并
$ git merge origin/master
```

# 文件重命名

- 先删除文件再添加文件

```
# 重命名3步骤    eg: 重命名 readme 为 readme.md
$ mv [文件名1] [文件名2]  # 重命名文件名1为文件名2
$ git add ['文件名'] # 添加新文件 eg: git add readme.md
$ git rm ['文件名'] # 删除旧文件 eg: git rm readme

# 就可以正常commit
```

- 直接使用 git 命令

```
$ git mv ['旧文件名'] ['新文件名'] # git重命名（相当于上面三个步骤）

# 就可以正常commit


# 回滚到对版本号的应提交状态【缺省回滚到最近的一次提交】
$ git reset --hard [提交版本号|缺省]

# 记录所有HEAD的历史，也就是说当你做reset，checkout等操作的时候，这些操作会被记录在reflog中
$ git reflog
```

# 文件删除

- 先删除文件再添加文件

```
$ rm <file> # 删除文件
$ git add ['文件名'] # 添加新文件 eg: git add readme.md

# 就可以正常commit
```

- 直接使用 git 命令

```
# 删除工作区，暂存区 的文件
$ git rm <file>
```

> 如果删除出错 借助 git reset --hard 进行撤销

# 提交

```
# 添加新文件
$ git add ['文件名' | '.'代表全部]

# 查看提交状态【是否有未提交】（不必的） but 能防止你不小心提交了您不想提交的东西
# $ git status # 没有未提交的显示 nothing to commit

# 提交填写日志
$ git commit -m'这里填写提交日志'

# 如果发现点问题，那么继续进行了修改
# 继续执行 git add . 重新提交，会覆盖上次提交，只以当前为准
# $ git commit --amend

# 查看日志(不必的)
# $ git log

# 更新 先更新避免发生冲突
$ git pull
# 推送到 orgin 远端服务器
$ git push
```

# 撤销&回滚

## 简单总结

1. 修改了工作区，恢复：git checkout
2. add 后，想撤销： git reset HEAD
3. commit 后，想撤销： git reset --hard [需要回退的 commit id]

## 撤销

### 修改文件 尚未提交

- 未执行 git add 操作

```
# 从暂存区（index） 恢复 文件
$ git checkout <fileName>
$ git checkout .
```

- 文件执行了 git add 操作，恢复 文件（index 内回滚）

```
# 取消暂存
$ git reset HEAD fileName
# 撤销修改
$ git checkout < fileName >

# 暂存区的 commitid 覆盖工作区修改
$ git checkout [commitid] -- <file>

# git checkout 5384b04 -- src/pages/AdScreen/components/commonStream.vue // 回退 5384b04 的文件
```

- 同时对多个文件执行了 git add 操作，本次只想提交其中一部分文件

```
$ git add *
$ git status
# 取消暂存
$ git reset HEAD <filename>
```

### 修改文件 已经提交（**git commit**）到 本地仓库

- 修改 git commit 不再产生新的 Commit[ 只能修改最近一次 ]

```
# 修改最后一次提交
$ git add sample.txt
# 代替（或这说修改）上一次提交，不只是修改message。
$ git commit --amend -m"说明"
```

- 多次 git commit 撤销到其中某次 Commit

```
# 修改最后一次提交
$ git reset --[soft/hard/mixed] [commit|HEAD]

# 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]
```

## 回滚

> 已进行 git push，即已推送到“远程仓库”中。我们将已被提交到“远程仓库”的代码还原操作叫做“回滚”！
> 注意：对远程仓库做回滚操作是有风险的，需提前做好备份和通知其他团队成员！

### 还原 远端服务器 提交的代码

```
# 根据 tag 还原工作区代码
$ git checkout <tag>

# 回滚到指定commitID
$ git checkout <commitID> <filename>
```

### 删除最后一次远程提交

- 使用 revert

```
$ git revert HEAD
$ git push origin master
```

- 使用 reset

```
$ git reset --hard HEAD^
$ git push origin master -f
```

二者区别：

> revert 是放弃指定提交的修改，但是会生成一次新的提交，需要填写提交注释，以前的历史记录都在；
> reset 是指将 HEAD 指针指到指定提交，历史记录中不会出现放弃的提交记录。

### 回滚某次提交

```
# 找到要回滚的 commitID
$ git log
# 根据 commitID 进行回滚
$ git revert commitID
```

### 删除某次提交

```
# 找到要回滚的 commitID
$ git log --oneline -n5
# 注意：最后的^号，意思是commit id的前一次提交
$ git rebase -i "commit id"^ # git rebase -i "5b3ba7a"^

# 根据交互输入命令修改信息
# 合并多个历史版本commit合并
$ git rebase -i [父级commit id ]

# 合并多个历史版本commit合并
$ git rebase -i [开始commit 的父级id] [结束commit id]
```

> 如果没有指定 结束 commit,那么结束 commit 默认为当前分支最新的 commit，那么 rebase 结束后会自动更新当前分支指向的 commit,
> 如果指定了结束 commit，而且结束 commit 不是当前分支最新的 commit，那么 rebase 后会有生成一个 游离的 head,，而且当前分支指向的 commit 不会更新

具体还可参考：
Deleting a Git commit
在 Git 中，如何『删除』commit?

# 操作标签

> 标签操作允许为存储库中的特定版本提供有意义的名称。

## 创建标签

```
# 创建标签
$ git tag -a 'tag1' -m 'tag1的说明' [HEAD / commit id / 缺省]
# -a选项的 标签名称，-m选项的 标签消息。
# 缺省：HEAD 要标记特定提交，则使用相应的COMMIT ID而不是HEAD指针

# 将 tag1 推送到 origin 远端
$ git push origin <tagname>

# 一次性推送全部尚未推送到远程的本地标签
$ git push origin --tags
```

## 查看标签

```
$ git tag
# 查看 所有可用的标签
$ git tag -l

# 查看 tagName标签的 详细信息
$ git show tagName
```

## 删除标签

```
# 查看 所有可用的标签
$ git tag -l

# 删除 tagName标签
$ git tag -d <tagName>

# 删除 远端 tag 标签
$ git push origin --delete tag <tagname>

# 删除 远端 tag 标签 类似 删除远端分支的做法
$ git push origin :refs/tags/标签名
```

# 操作分支

## 查看分支

```
# 查看分支
git branch -v
# 查看本地分支对应的远程分支
$ git branch -av
```

## 新建分支

```
# 新建一个名字为 dev2 的分支 (不切换到新分支)
$ git branch dev2

# 新建一个名字为 dev2 的分支 并切换到该分支
$ git checkout -b <branchname>

# 从某个版本创建分支 并切换到该分支
$ git checkout -b <branchname> [commit号/分支名称]

# 创建并切换分支
$ git checkout -b <branchname> [commit版本号]
# 基于 远端分支 建立 本地分支（远程分支名x） 采用此种方法建立的本地分支会和远程分支建立映射关系
$ git checkout -b <本地分支名x> <origin/远程分支名x>

/* 相当于分别执行了下面两条命令 */
# git branch <branchname>
# git checkout <branchname>
```

## 提交到远程分支

```
$ git push [origin] <分支名称>:<分支名称>
# 本地分支push到远程服务器，远程分支与本地分支同名（当然可以随意起名）
$ git pull [origin] <分支名称> 更新远程分支到本地

# /* 远程分支关联 */
$ git branch --set-upstream-to=[origin]/<分支名称>
# 接下来就 可以直接pull 或者push 了

# 查看关联的分支名称
$ git branch -vv
```

> [origin] 代指 添加远程裸仓库地址时候，创建的名称

## 删除分支

```
#  删除分支 -d是删除  -D 是强制删除
$ git branch [-D/-d] <branchname>

# 删除远程分支
$ git branch -d -r <branchname>
$ git push origin :<branchname>
# OR
$ git branch # 查看分支名称
$ git push origin --delete <branchname>
```

## 合并分支

fast-forward

- fast-forward 方式 合并
  > 这种方法相当于直接把 master 分支移动到 test 分支所在的地方，并移动 HEAD 指针

```
# 先切换到主分支，为合并分支准备
$ git checkout master
# 进行合并
$ git merge dev
```

- no-fast-forward 方式 合并
  > 这种合并方法会在 master 分支上新建一个提交节点，从而完成合并

```
# 先切换到主分支，为合并分支准备
$ git checkout master
# 进行合并
$ git merge -no-ff dev
```

- squash 方式 合并
  > squash 和 no-ff 非常类似，区别只有一点不会保留对合入分支的引用

```
$ git checkout master
$ git merge -squash dev
```

- rebase 方式 合并
  > rebase 与 merge 不同，rebase 会将合入分支上超前的节点在待合入分支上重新提交一遍，变成线性历史

```
# 先切换到主分支，为合并分支准备
$ git checkout master
# 进行合并
$ git rebase dev
```

- cherry-pick 挑拣 合并
  对已经存在的 commit 进行 再次提交 [选择某些节点进行合并]

```
$ git cherry-pick <commit id>
```

> 当执行完 cherry-pick 以后，将会 生成一个新的提交；这个新的提交的哈希值和原来的不同，但标识名 一样；

> 图解 4 种 git 合并分支方法 > git cherry-pick 的使用

# 暂存

当收到紧急任务，手里又存在未完成的模块可以先放到暂存区

```
git stash # 暂存

git stash list # 查看暂存列表

git stash apply [序号/缺省为stash@{0}] # 恢复暂存进度到工作区

git stash pop [序号/缺省为stash@{0}] # 恢复暂存进度到工作区并删除
# git stash pop = git stash pop stash@{0}
```

# 分离头【detached HEAD】

执行 git checkout [commit 版本号]，git 会提示显示处于分离头（**无分支状态**），在此状态下，进行的提交操作不挂到在分支下，直接切换分支，会导致分离头的提交丢失。

错误示范：

```
git checkout 15a6686b624 # 检出15a6686b624的提交，到分离头情况

git commit -am"错误示例" # 创建一次提交
git branch -av # 查看分支

git checkout marst # 切换到marst 分支  此时git 会提示你，对分离头提交

# 最后一次补救
git branch [分支名称] [commit版本号]
```

正常使用

```
# 使用某个提交创建分支
git checkout -b [分支名称] [commit版本号] // 创建并切换分支

git checkout -b 本地分支名x origin/远程分支名x // 基于 远端分支 建立 本地分支（远程分支名x） 采
```

# 操作子仓库

```
# 添加子仓库
$ git submodule add <仓库地址> <本地路径>

# 检出子仓库

# 初始化本地配置文件
$ git submodule init

# 检出父仓库列出的commit
$ git submodule update

# 或者使用组合指令。
$ git submodule update --init --recursive

# 删除子仓库
# 1删除.gitsubmodule里相关部分
# 2删除.git/config 文件里相关字段
# 3删除子仓库目录。
$ git rm -r --cached <本地路径>
```

git submodule 使用小结

# 常用命令详解

## git pull 详解

> git pull 相当于从远程获取最新版本并 merge 到本地

```
# 按照git branch 设置的默认跟踪的服务器和分支来拉取
$ git pull
# 实例：拉取远程服务器origin的master分支
$ git pull origin master
```

git pull 详解

```
$ git pull [<options/缺省>] [<远端仓库名称> [<分支名称>…]]
# options : -allow-unrelated-histories 允许无关的历史，这个选项，更多是在更改远程仓库的时候用到
# options : -ff      开启fast-forward
# options : -no-ff   强行关闭fast-forward方式
# options : -ff-only
```

## git fetch 详解

> git fetch 相当于是从远程获取最新到本地，不会自动 merge
> 在实际使用中，git fetch 更安全一些

使用 git fetch 进行更新

```
# 将远程仓库的master分支下载到本地当前branch中
$ git fetch orgin master

# 比较本地的master分支和origin/master分支的差别
$ git log -p master  ..origin/master

# 进行合并
$ git merge origin/master
```

OR

```
# 从远程仓库master分支获取最新，在本地建立tmp分支
$ git fetch origin master:tmp
#將當前分支和tmp進行對比
$ git diff tmp
#合并tmp分支到当前分支
$ git merge tmp
```

> git pull = git fetch + git merge

# git add 详解

> 将修改添加到暂存区

```
# 将所有修改添加到暂存区
$ git add .
# Ant风格添加修改
$ git add *

# 将文件的修改、文件的删除，添加到暂存区。
$ git add -u
# 将文件的修改，文件的删除，文件的新建，添加到暂存区
$ git add -A

# 将以Controller结尾的文件的所有修改添加到暂存区
$ git add *Controller
# 将所有以Hello开头的文件的修改添加到暂存区 例如:HelloWorld.txt,Hello.java,HelloGit.txt ...
$ git add Hello*
# 将以Hello开头后面只有一位的文件的修改提交到暂存区 例如:Hello1.txt,HelloA.java 如果是HelloGit.tx
$ git add Hello?
```

# git commit 详解

> 用于将更改记录(提交)到存储库

```
# 普通提交填写提交信息
$ git commit -m "the commit message"

# 将所有已跟踪文件中的执行修改或删除操作的文件都提交到本地仓库，即使它们没有经过git add添加到暂存区，
# 然后提交(有点像svn的一次提交,不用先暂存)。
# 对于没有track的文件,还是需要执行`git add <file>` 命令。
$ git commit -a
$ git commit -a -m "the commit message"

# 增补提交（修改上次提交），会使用与当前提交节点相同的父节点进行一次新的提交，旧的提交将会被取消。
$ git commit --amend
```

# git remote 详解

> git remote 命令管理一组跟踪的存储库

```
# 查看当前的远程库
# 列出已经存在的远程分支
$ git remote
#  列出详细信息，在每一个名字后面列出其远程url
$ git remote [-v | --verbose]

# 添加一个名字为 <name> 的 远程服务器
# $ git remote add <name> <url>
$ git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <
# 重命名 远程链接
$ git remote rename <old> <new>
# 删除 远程链接
$ git remote remove <name>


$ git remote set-head <name> (-a | --auto | -d | --delete | <branch>)
$ git remote set-branches [--add] <name> <branch>…

# 获取 <name> 远程服务 地址
$ git remote get-url [--push] [--all] <name>
# 设置 <name> 远程服务 地址
$ git remote set-url [--push] <name> <newurl> [<oldurl>]
# eg: git remote set-url [ 裸仓库名称/常用origin]

$ git remote set-url --add [--push] <name> <newurl>
$ git remote set-url --delete [--push] <name> <url>
$ git remote [-v | --verbose] show [-n] <name>…
$ git remote prune [-n | --dry-run] <name>…
$ git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]
```

# git push 详解

> 将修改添加到暂存区

## 语法

```
# 命令用于将本地分支的更新，推送到远程主机。与git pull命令相似。
$ git push <远程主机名/缺省> <本地分支名/缺省>:<远程分支名/缺省>
```

## 示例

> origin 可通过 .git config 查看地址

### 完整示例

```
# 将本地的master分支推送到origin主机的master分支。如果master不存在，则会被新建。
$ git push origin master
```

### 本地分支名 缺省

表示**删除指定的远程分支**，因为这等同于推送一个空的本地分支到远程分支。

```
# 表示删除origin主机的master分支
$ git push origin :master
# 等同于
$ git push origin --delete master
```

### 本地分支、远程分支都 缺省

表示推送**特定主机的对应分支**简写方式：如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

```
# 将当前分支推送到origin主机的对应分支
$ git push origin
```

### 远程主机名、本地分支、远程分支都 缺省

表示推送**origin 主机的对应分支**简写方式：如果当前分支只有一个追踪分支，那么主机名都可以省略。

```
# 将当前分支推送到 预设 主机【默认origin】的对应分支
$ git push
```

当前分支与多个主机存在追踪关系，则可以使用-u 选项指定一个默认主机，这样后面就可以不加任何参数使用 git push

```
# 将本地的master分支推送到origin主机，同时指定origin为默认主机，后面就可以不加任何参数使用git push
$ git push -u origin master
```

### simple 方式 和 matching 方式

> 不带任何参数的 git push，默认只推送当前分支，这叫做 simple 方式
> matching 方式，会推送所有有对应的远程分支的本地分支
> Git 2.0 版本之前，默认采用 matching 方法，现在改为默认采用 simple 方式

如果要修改这个设置，可以采用 git config 命令

```
$ git config --global push.default matching
# 或者
$ git config --global push.default simple


# 不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机
# 所有本地分支都推送到origin主机
$ git push --all origin

# 使用--force选项，结果导致在远程主机产生一个”非直进式”的合并(non-fast-forward merge)
$ git push --force origin

# git push不会推送标签(tag)，除非使用-tags选项
$ git push origin --tags
# 推送tag
$ git push origin tag_name
# 删除远程标签
$ git push origin :tag_name

# 将当前分支推送到远程的同名分支
$ git push origin HEAD

# 将当前所在指针分支 推送到 远程的master分支
$ git push origin HEAD:master

# 用本地分支 dev 覆盖远程分支 dev_op
$ git push -f origin dev:refs/dev_op
# or
$ git push origin :refs/dev   //删除远程的dev分支
$ git push origin dev:refs/dev_op
```

# git branch 详解

> 用于列出，创建或删除分支

## 语法

```
# 查看分支
$ git branch [-r | -a]

# 新建分支
$ git branch [-f] <branchname>

# 重命名分支 使用-M则表示强制重命名
$ git branch (-m | -M) <oldbranch> <newbranch>

# 删除分支 使用-D则表示强制删除，相当于 --delete --force
$ git branch (-d | -D) <branchname>
```

> 使用-d 在删除前 Git 会判断在该分支上开发的功能是否被 merge 的其它分支。如果没有，不能删除。如果 merge 到其它分支，但之后又在其上做了开发，使用-d 还是不能删除。-D 会强制删除

## 示例

```
# 查看分支
$ git branch
# 列出所有远程分支
$ git branch -r
# 查看本地和远程分支
$ git branch -a
# 查看本地分支对应的远程分支
$ git branch -vv

# 新建一个名字为 dev2 的分支
$ git branch dev2

# 修改分支的名字
#你需要重命名远程分支，推荐的做法是：1.删除远程待修改分支 2.push本地新分支名到远程
$ git branch -m dev2

# 删除本地分支
$ git branch -d <branchname>

# 删除远程分支
$ git branch -d -r <branchname>
$ git push origin :<branchname>
# OR
$ git branch
$ git push origin --delete dev2
```

# git checkout 详解

> 用于切换分支或恢复工作树文件。这条命令会重写工作区

```
# 切换 <branch> 分支
$ git checkout <branch>
# 从 <branch> 分支 提交中取出文件
$ git checkout <branch>  <fileName>
# 从暂存区（index） 恢复文件
$ git checkout <fileName>

# 新建 < branch > 分支 并进行切换
$ git checkout -b <branch>
$ git checkout -b|-B <new_branch> [<start point>] # 完整版
# 相当于 执行
# git branch newBranch
# git checkout newBranch

# 换到newBranch的远程分
$ git checkout -b newBranch  origin/newBranch

# 检出索引中的 fileName文件
$ git checkout -- <fileName>

# 检出索引中的 fileName文件
$ git checkout -- <fileName>

# 用于检出某一个指定文件
# 不填写commit id，则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 [-q] [<commit id>] [--] <paths>

# 当暂存区为空，如果我们想要放弃对某一个文件的修改，可以用这个命令进行撤销
$ git checkout [-q] [--] <paths>

# 新建的分支，严格意义上说，还不是一个分支，因为HEAD指向的引用中没有commit值，只有在进行一次提交后，
$ git checkout --orphan <new_branch>

# 将当前分支修改的内容一起打包带走，同步到切换的分支下 [切换到新分支后，当前分支修改过的内容就丢失了
$ git checkout --merge <branch>
$ git checkout -m <branch>
```

## 例子

```
#//取出master版本的head。
$ git checkout master
#//在当前分支上 取出 tag_name 的版本
$ git checkout tag_name
#//放弃当前对文件file_name的修改
$ git checkout  master file_name

#//取文件file_name的 在commit_id是的版本。commit_id为 git commit 时的sha值。
$ git checkout  commit_id file_name

# 从远程dev/1.5.4分支取得到本地分支/dev/1.5.4
$ git checkout -b dev/1.5.4 origin/dev/1.5.4

#这条命令把hello.rb从HEAD中签出.
$ git checkout -- hello.rb

# 检出索引中的所有C源文件
$ git checkout -- '*.c'

#这条命令把 当前目录所有修改的文件 从HEAD中签出并且把它恢复成未修改时的样子.
#注意：在使用 git checkout 时，如果其对应的文件被修改过，那么该修改会被覆盖掉。
$ git checkout .
```

# git reset 详解

```
$ git reset [ --soft | --mixed | --hard] <commit>
# git reset <commit> 的意思就是 把HEAD移到<commit>

# --soft 这个只是把 HEAD 指向的 commit 恢复到你指定的 commit，暂存区 工作区不变
# --hard 这个是 把 HEAD， 暂存区， 工作区 都修改为 你指定的 commit 的时候的文件状态
# --mixed 这个是不加时候的默认参数，把 HEAD，暂存区 修改为 你指定的 commit 的时候的文件状态，工作区

# 取消暂存区 部分文件（files 代表多个）的修改
$ git reset HEAD --< files >
```

# git cherry-pick 详解

语法

```
$ git cherry-pick [<options>] <commit-ish>...

# 常用options:
#   --quit                退出当前的chery-pick序列
#   --continue            继续当前的chery-pick序列
#   --abort               取消当前的chery-pick序列，恢复当前分支
#   -n, --no-commit       不自动提交
#   -e, --edit            编辑提交信息
```

## git submodule 详解

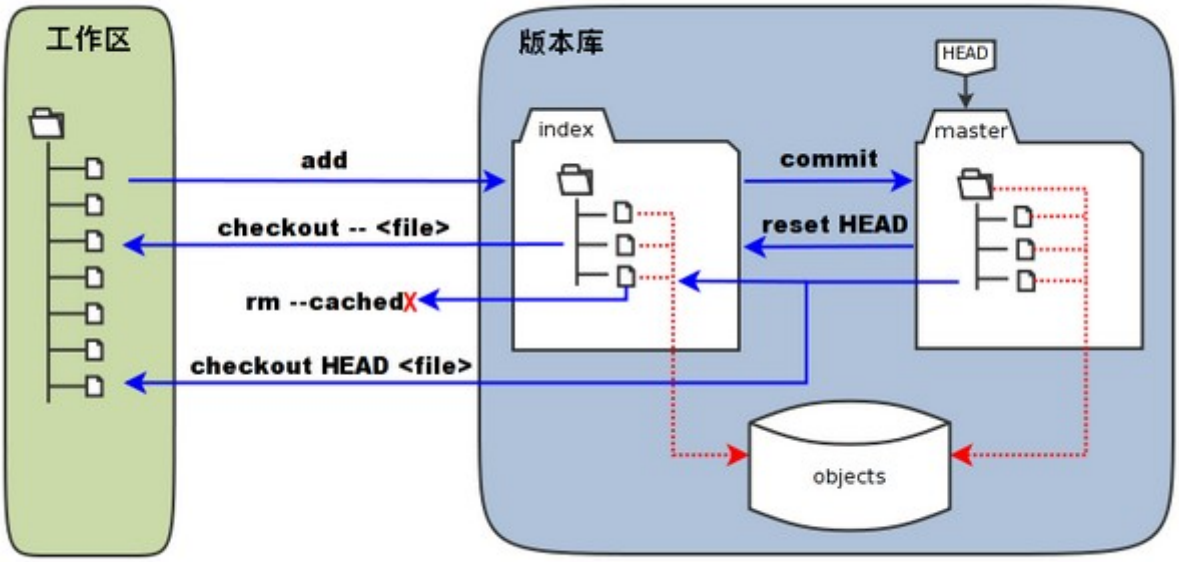
> 命令用于初始化，更新或检查子模块

```
$ git submodule status [--cached] [--recursive] [--] [<path>…]

# 添加 子模块 路径为 相关信息保存在 .gitmodules 文件
$ git submodule add <url> [<path>]


# 初始化本地配置文件
$ git submodule init [--] [<path>…]
# 检出父仓库列出的commit
$ git submodule update
# 使用组合指令
$ git submodule update --init --recursive
```

# git 原理

## git 工作解析图

下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系：




# fast-forward [快进] 与 no-fast-forward

> 当前分支合并到另一分支时，如果没有分歧解决，就会直接移动文件指针。这个过程叫做 fastforward。

具体阅读以下文章，感受下：

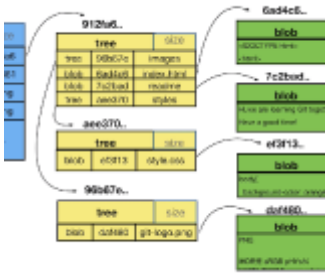
- git：到底什么是 fast-forwards
- 图解 4 种 git 合并分支方法 - 颜海镜

# git 文件目录

- COMMIT_EDITMSG
- config 当前 git 的配置文件
- description （仓库的描述信息文件）
- HEAD （指向当前所在的分支），例如当前在 develop 分支，实际指向地址是 refs/heads/develop
- hooks [文件夹]
- index
- info [文件夹]
- logs [文件夹]
- objects [文件夹] （存放所有的 git 对象，对象哈希值前 2 位作为文件夹名称，后 38 位作为对象文件名, 可通过 git cat-file -p 命令，拼接文件夹名称+文件名查看）
- ORIG_HEAD
- refs [文件夹]
  • heads （存放当前项目的所有分支）

• tags (存放的当前项目的所有标签，又叫做里程碑)

## commit 与 tree 和 blob 的关系



commit 对应一个tree

tree 包含文件[blob]，如果下级还是文件夹，则又是一个tree

blob 是tree下面具体文件 【与文件名无关具体到文件内容】

## git 注意事项

1. checkout reset 慎用
2. 禁止向集成分支[多人使用分支] 执行 push -f [强制更新到远端 可能会导致远端所在分支回退很多版本]

   > 如果单人自行分支，确认是要返回某个节点， 使用 push -f
   > 公共分支修改 commit ： git reflog 命令查找历史，然后利用 git reset --hard [提交版本号|缺省] 的方式恢复

3. 公共分支 禁止进行 rebase 变基操作

   > 对于自己在本地的多次 commit，我想把他合并成一次 commit，还没有 push 的情况下,使用 rebase