License：

GNU General Public License （GPL）的可能是开源项目最常用的许可证。由于GPL的规定，使用到部分GPL代码的软件，必须也使用GPL。GPL赋予和保证了开源项目开发者广泛的权利。基本上，它允许用户合法复制，分发和修改软件。**如果你使用并且修改了我的GPL软件，那么你的软件也必须要开源，否则就不能使用我的软件，你是否把你的软件商用和我没关系**，你必须给客户提供一个GPL副本，告诉他们可以在其他地方得到免费版本。当然最好在此之前，告诉客户，你为什么收费。评价：这是一份促进开源的协议。

EPL是一个与GPL相类似的许可证，任何扩展自Eclipse源码的代码也必须是开源的。

**GNU Lesser General Public License(**LGPL**)**协议的代码或者衍生，则所有修改的代码，涉及修改部分的额外代码和衍生的代码都必须采用LGPL协议。因 此LGPL协议的开源 代码很适合作为第三方类库被商业软件引用，但不适合希望以LGPL协议代码为基础，通过修改和衍生的方式做二次开发的商业软件采用。评价：和GPL协议相比，开源的范围更小了。

GPL/LGPL都保障原作者的知识产权，避免有人利用开源代码复制并开发类似的商用产品。咨询官方，购买授权例外，比如不开放自己的源码。

BSD是"*Berkeley* Software Distribution"的缩写，意思是"伯克利软件发行版"。显然，BSD这个名称并不是我们现在所理解的操作系统。如果再发布的产品中包含源代码，则在源代码中必须带有原来代码中的BSD协议。如果再发布的只是二进制类库软件，则需要在类库软件的文档和版权声明中包含原来代码中的BSD协议。不可以用开源代码的作者机构名字和原来产品的名字做市场推广。

Apache Licence、MIT是和BSD一样宽松的许可协议,作者只想保留版权,而无任何其他了限制.也就是说,你必须在你的发行版里包含原许可协议的声明,无论你是以二进制发布的还是以源代码发布的。

**Mozilla Public License 1.1 (MPL)。**MPL协议允许免费重发布、免费修改，但要求修改后的代码版权归软件的发起者。这种授权维护了商业软件的利益，，它要求基于这种软件得修改无偿贡献版权给该软件。这样，围绕该软件得所有代码得版权都集中在发起开发人得手中。

Creative Commons（CC）的许可证不太开放源代码授权，它们通常用于设计项目。 CC许可证有着宽广的定义，每个定义都会授予一定的权利。它有四个基本部分，可以单独或者组合使用。下面是部分概述：  
\* 署名  
作者必须是作品的原创者。 除此之外，作品可以修改，分发，复制和以其他方式使用。  
\* 相同方式共享  
工作可以修改，分发等等，但必须在一个许可证下。  
\* 非商业  
可以修改，分发等，但不用于商业目的。 关于什么是“商业”，说法比较含糊（没有提供明确的定义），因此您可能需要在自己的项目中澄清这一点。  
\* 禁止修改  
这意味着您可以复制和分发许可工作，但你不能以任何方式修改，或在原有的基础开发。  
商业软件的使用要遵从CC协议的具体规定，最严格的许可证将是“署名，非商业，不能修改”的授权。这意味着你可以自由共享的工作，但不能改变它，你必须把它归功于原创者。

协议一直在处理的就是作者的版权、署名以及是否用于商业。可能会有是否允许修改。

git是分布式版本控制系统+内容可寻址文件系统

Git（同docker的storage driver） 是一个被广泛使用的版本控制系统，但在规模扩展上有些不尽如人意。**随着项目和代码库的增长，其性能也会受到很大的影响，一个常见的小任务，都有可能耗费数小时去执行。不过今天，微软已经给出了解决这个问题的答案 —— Git 虚拟文件系统GVFS。在大项目的时候整个项目的操作都会比git快很多。**

Info: <https://www.git-tower.com/blog/git-cheat-sheet/>

Help: git help <command>

<http://rogerdudler.github.io/git-guide/index.zh.html>

<http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>

稍微熟悉下面命令以后的简单指令使用平台：<https://try.github.io/levels/1/challenges/1>

<http://blog.csdn.net/express_yourself/article/details/55051681> 中文解释

<http://learngitbranching.js.org>



git checkout HEAD file



commit -am 勉强可以作为workspace->Repository

* Workspace：工作区
* Index / Stage：暂存区
* Repository：仓库区（或本地仓库）
* Remote：远程仓库

GitHub ([www.github.com](http://www.github.com))

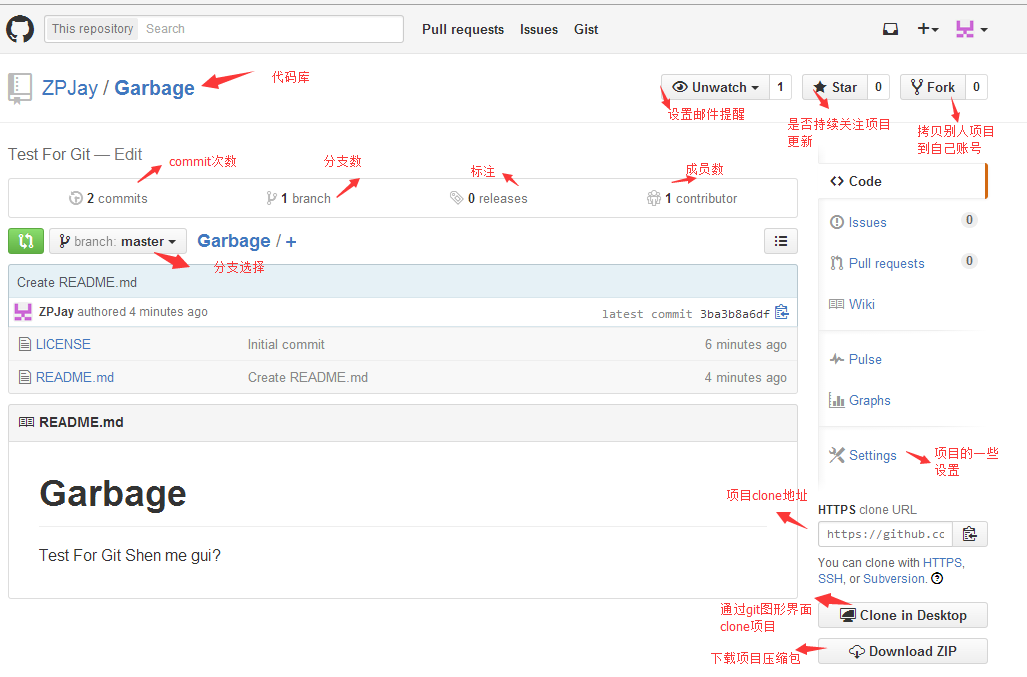
Beanstalk ([www.beanstalkapp.com](http://www.beanstalkapp.com)) 豆茎

Bitbucket ([www.bitbucket.com](http://www.bitbucket.com)) 位市场

Plan.io ([www.plan.io](http://www.plan.io))

如果访问过慢是DNS原因，可以nslookup后写死host解析。<https://blog.csdn.net/GentleCP/article/details/86764068>

一般提供两种方式访问https、ssh，如果将远程仓库origin设置为https，那么每一次push的时候会需要http身份验证，一般会要求输入账号密码；避免方式是改成ssh绑定key。



Git文件系统中：你的本地仓库由 git 维护的三棵“树”组成。第一个是你的工作目录，它持有实际文件；第二个是暂存区（Index/Stage），它像个缓存区域，临时保存你的改动；最后是HEAD，它表示现在你的workspace处于仓库中的哪个节点（也就是现在工作区的状态）。

推荐的UI操作软件：gitkraken、sourcetree

内建的图形化 git：gitk  
彩色的 git 输出：git config color.ui true

在绝大多数场合我们取origin这个名字指代远程的代码仓库，也就是图片上的remote。

Tag标签可以作为branch的别名给当前分支打上标签（git tag）

git config –glocal user.name “xxx” 自然没有空格的时候不需要引号

git config –global user.email “xx“

或者直接在编辑器里面编辑git config --global --edit

如果不设置，就会基于登录名和主机名进行了自动设置

git --amend --reset-author 通过这个命令来更改上次提交的作者

配置文件位置

--global 使用全局配置文件

--system 使用系统级配置文件

--local 使用仓库级配置文件

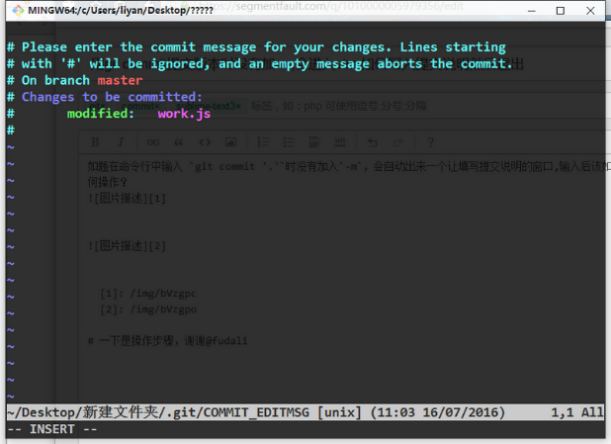
-f, --file <文件> 使用指定的配置文件

--blob <数据对象 ID> 从给定的数据对象读取配置”

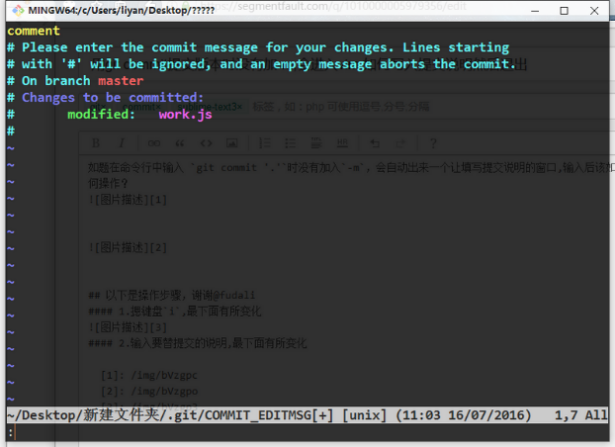
--edit 缩写也就是-e 编辑配置文件（会成为vim那样打开）

修改~/.gitconfig文件,至少要执行一次命令才会自动创建这个文件

如果你没有指定-m参数就会进入vi文本编辑器，在这里指定消息名称：



你只需要加上一行黄色的字在开头来表示你希望的message名字并退出保存：



git init还在当前代码目录下产生一个.git文件夹作为仓库信息存储

C:\Users\Battery\AppData\Roaming\Tencent\Users\641370196\QQ\WinTemp\RichOle\06X1UFLH%XH6EZX{Z}S56_B.png

HEAD文件存放当前HEAD的指向

objects文件夹中存储着所谓的组件：commmit、tree、blob，相关网址介绍：<https://www.jianshu.com/p/8659c9ae00cb>

这时候属于刚刚创建，里面就只有一条master分支。每一个分支表示进行中过程,每一个commit就是一个个节点.

git add且commit之后的文件上的修改就会被git系统跟踪，被跟踪就不能被.gitignore

git add \* <=> git add ./ 目录会自带递归

Git对于hash的寻找只需要提供能够唯一标识一个Node的前几个字符就好.

git log --help

git log --author=bob通过一些参数可以限制来查看

git log -p <filename> 查看指定文件的历史变化，但是如果没有--all的话，单独一个分支中的文件可能最新的几次改动没有显示出来。git log --all --graph -p <filename>

--name-status 增加对每次commit中文件的详细变化显示，A（append） M（modify）

git log --graph 树状图的显示 --oneline --decorate --all 能够让显示更加简洁

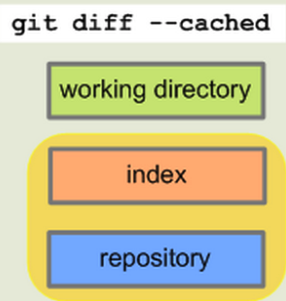
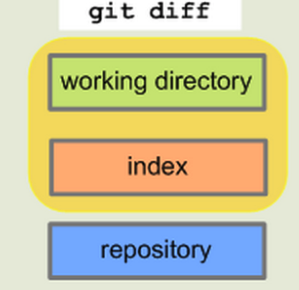
git checkout --track origin/dev 切换到远程dev分支

git diff 默认是比较当前数据与本分支的最后一次提交，需要文件已经被跟踪

Git diff [file]

Git diff <branch1> <branch2>

git diff --cached(git diff --staged) 查看index与当前的差别（前提是两个比较对象内都存在这个文件）



如果创建git init之后，新建了一个文件，index为空，working directory 中存在一个新文件，但是git diff没有任何显示。git status倒是可以看到文件没有被staged。如果修改文件内容，这时候，git diff能够看到文件内修改的内容。其实我觉得应该将status 与 diff合并到一起。

git diff HEAD 查看当前repository与working directory之间的区别

图形化的git difftool或者安装第三方的软件进行diffmerge

root@kali:~/test1# git diff HEAD^

diff --git a/file2 b/file2 #进行比较的是，a版本的f1（即变动前）和b版本的f1（即变动后）。

new file mode 100644 #对象的模式（普通文件，644权限）

index 0000000..b065a3f #index区域的6f8a38c对象，与工作目录区域的449b072对象进行比较。这些值是git hash出来的。

--- /dev/null #---表示源文件，由于我的文件是新建

+++ b/file2 #+++表示目标文件

@@ -0,0 +1 @@ #源文件第0行到第0行，新文件第1行 是发生改变的地方

+1123 #这里是改变内容

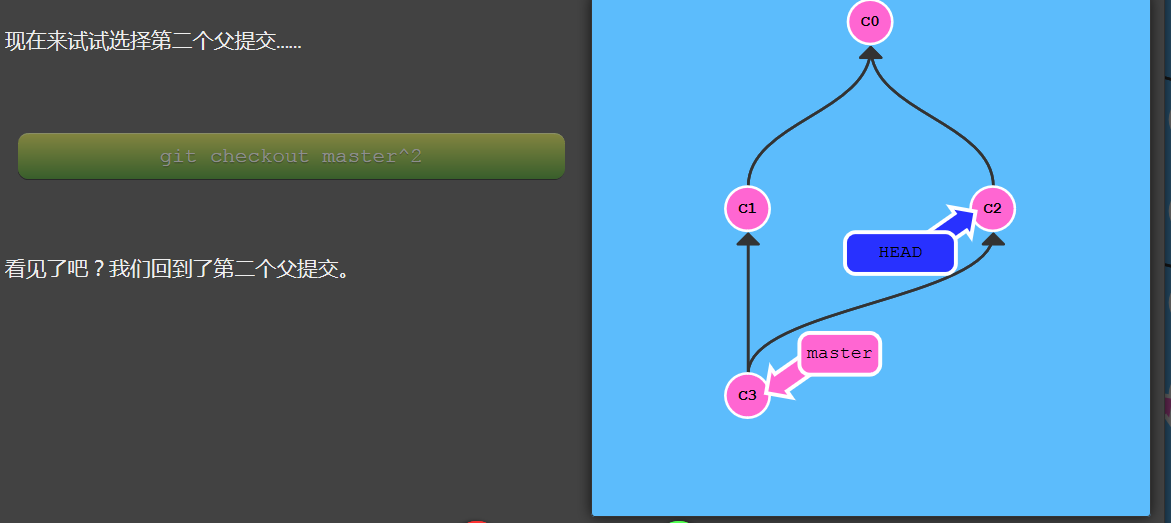
检查点叫做check point

git checkout能做的事情有1.HEAD的移动（本分支或者移动到其他分支），可以使用commitHash，可以使用tag，可以使用HEAD与^~等操作，可以使用branchName。 2.对单个指定文件的状态进行切换（也就是还原）

我们改变HEAD指向（这样git reset就会不一样）的时候使用git checkout指令：  
git checkout commitHash

git checkout HEAD^到当前HEAD的父提交处

git checkout branchName^2 就会将HEAD指针指向当前分支第二个父提交处



Git checkout master^2~1支持链式构造回溯路径（~是^的数字写法，是配合commitHash来使用的）

checkout就是从仓库中取出指定东西的状态，切换掉当前本地对应东西的。如果当前存在unstaged数据（也就是未提交请求），那么就会阻止进行分支切换，否则可能造成损失：

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test ((7850f0d...))

$ git checkout new

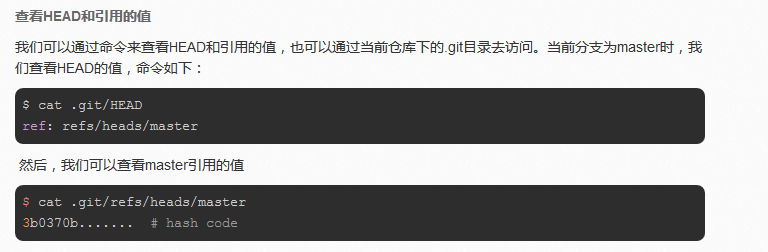
error: Your local changes to the following files would be overwritten by checkout:

test.txt

Please commit your changes or stash them before you switch branches.

Aborting

使用git checkout切换分支的时候HEAD是种会切换到当前分支的最下面，当前HEAD是在什么分支的什么位置可以使用git branch –av进行查看



要想下移HEAD，只能使用commmitHash或者tagName，又或者切换branch切过去在切回来。

git checkout -- <target> 命令文件可以回到最后一次提交的时候《=》git checkout HEAD <target>

git checkout -b feature\_x创建一个叫做“feature\_x”的分支，并切换过去

git checkout -b guowenyu origin/guowenyu 按照远程分支在本地创建一个分支

每一个分支都代表着每一条路最新的节点？不，分支名字与HEAD一样可以指向任何一个节点。每一个节点都可以用一段hash表示。

当前现实情况即由HEAD指向的节点

git branch -f master HEAD~3这条命令就会把master指向当前（HEAD指向）回溯3步的节点

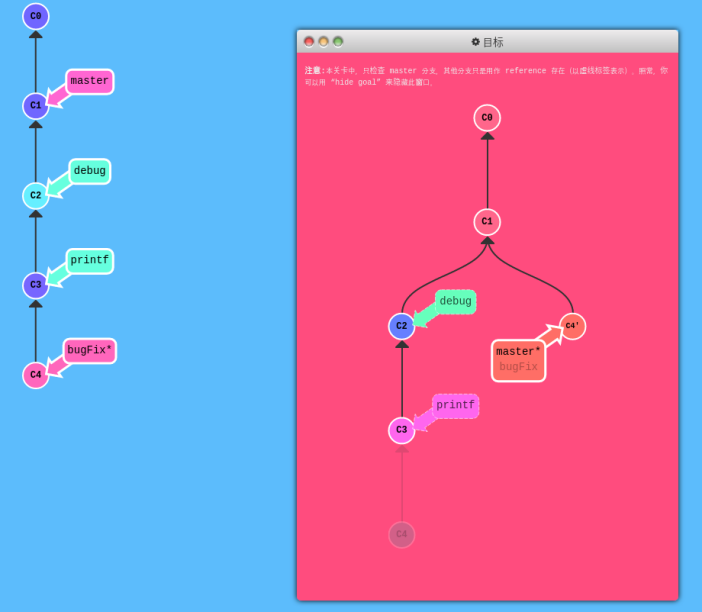
HEAD就可以当做一个commitHash或者commitTag来用，至于那些commit hash和分支名字就当做指向commit的const 指针。通过commit的hash，就可以在各个命令上直接调用，可以将别的分支上的commit拿来用。比如：

git cherry-pick <commit> <commit> …就会把指定的commits复制过来作为本分支的已commit节点。

Git rebase –i HEAD~4从HEAD~4开始移动本分支上的commit节点排列(包含对commit节点的删除)，从而重建本分支（还是图形化操作）。相当于使用了git branch –f与git

组合。

Git branch branchname 就会在当前HEAD所指的commit处创建一个分支。



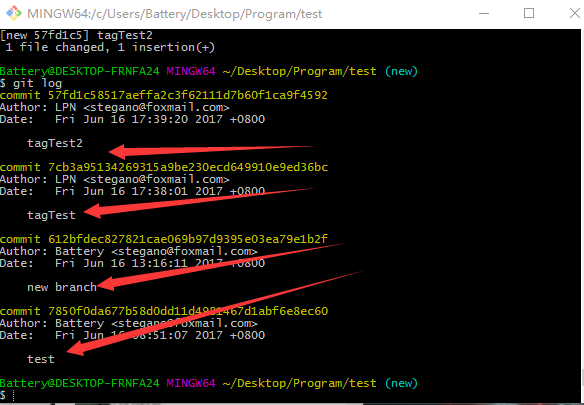
以上只能使用git rebase -i或者git cherry-pick. 使用git rebase做不到,因为认为C4已经基于C1.

git tag就是给当前HEAD所指向的commit建立一个别名。不过为了方便不让你这么频繁地移动HEAD，就有了git tag tagName <commit>

-a tagNamez add tag

-m “comment infomation” 这个也可以在提交的时候进行指定

-d tagName delete tag



Only commit the related file and often commit.

Git commit –m “Fixed the bug:xxxx” –m后面要简短，每一件事情一次commit

-m后面的消息会在git log命令中被查看到。为了和自动生成的注释（比如git merge）保持一致需要使用现在时态。如果没有-m参数就会打开vi让你来输入你对本次提交的说明。

-a 将已经跟踪的文件（git所知文件）修改与删除提交到index。而git add不仅仅可以将修改内容提交到index，还可以将git 未跟踪的文件加入到index与跟踪目标。

commit是index=> repository，即便是已经被跟踪的文件修改也要add才能进入index，否则被commit

跟踪与取消跟踪：

git add（还有将本地内容提交到缓存）和git rm后面是可以跟着多个文件的，用空格隔开。

git rm <file>

-f 连着本地文件一起删除，这也就是默认的不使用参数

--cached 保留本地文件(只是删除对文件的追踪)

git merge：

If a merge conflict should occur, Git will already update the rest

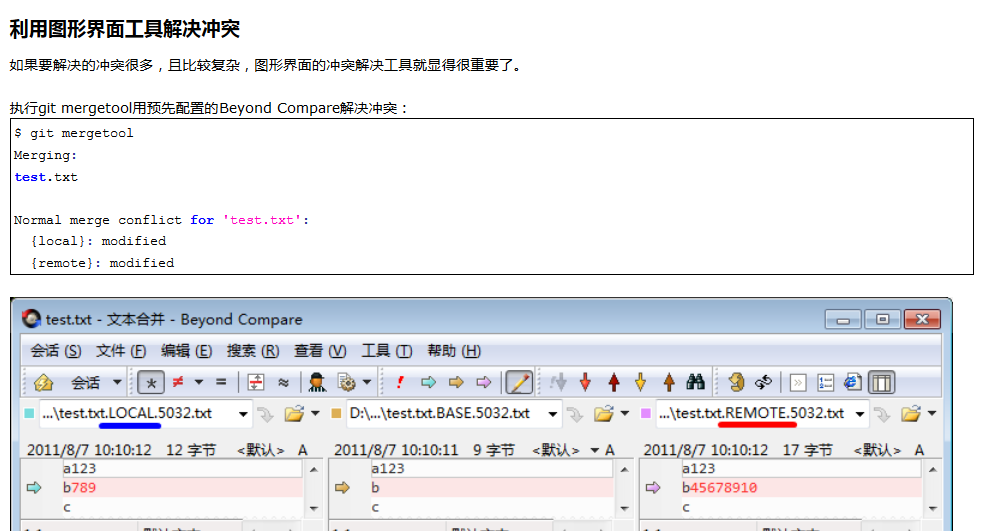
of the working copy to the new state. After resolving a conflicted

file, you can mark it using the “ git add ” command

在合并改动之前，你可以使用如下命令预览差异：  
git diff <source\_branch> <target\_branch>如果相同文件的相同行有不同的更改那么merge就会报错。默认是与当前的working directory比较。

git rm '\*.txt' #指定文件需要使用字符串符号包裹，不然命令存在不能执行的可能

在团队合作的时候，同事与我都对同一个地方进行修改，然后同事先push，我再push的时候因为自动merge，会碰到conflict的问题（后一个人push失败）。这时候就使用mergetool 整个解决过程是在交互式问答中完成的，用d 删除不要的文件，用c保留需要的文件。



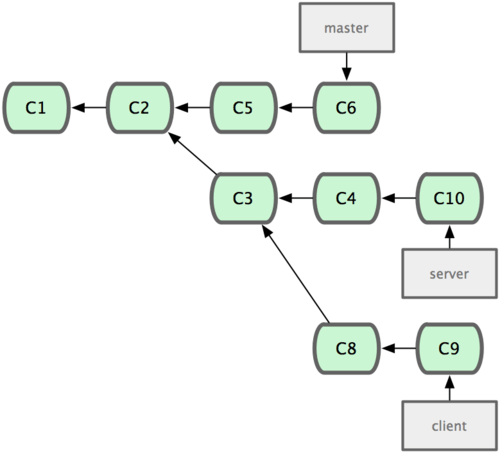
撤销提交：

git revert < commit > 是生成一个新的提交来撤销某次提交，此次提交之前的commit都会被保留

git reset <commit> 是回到某次提交，提交及之前的commit都会被保留，但是此次之后的修改都会被退回到暂存区。Reset命令只能对分支使用，不能对与分支分离的HEAD使用（HEAD没有指向任何一个分支）。

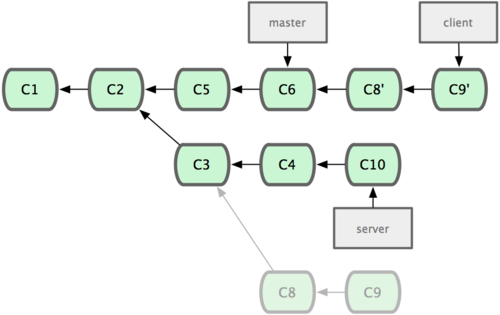
reset执行以后会改变git log，执行以后是不能被还原的。区分不使用任何参数、使用--hard参数（丢弃指定commit节点之后所有的改变）、使用—keep参数。git reset是对HEAD的引用值的更改。

变基操作的实质是丢弃一些现有的节点，然后相应地新建一些内容一样的节点。 如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 git rebase 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。



假设在接下来的一次软件发布中，我们决定先把客户端的修改并到主线中，而暂缓并入服务端软件的修改（因为还需要进一步测试）。这个时候，我们就可以把基于 client 分支而非 server 分支的改变（即 C8 和 C9），跳过 server 直接放到 master 分支中重演一遍，但这需要用 git rebase 的 --onto 选项指定新的基底分支 master：

$ git rebase --onto master server client这好比在说：“取出 client 分支，找出 client 分支和 server 分支的共同祖先之后的变化，然后把它们在 master 上重演一遍”

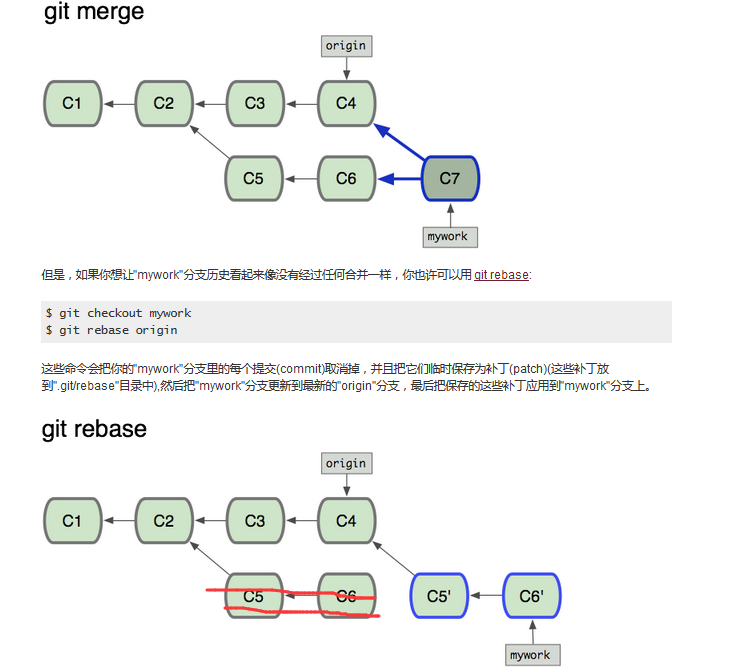


除了使用git merge来合并分支还可以使用git rebase来转义分支更改，git rebase转义当前分支上的一切更改到指定的分支，同时删除被转移的分支。

有一种观点认为，仓库的提交历史即是 记录实际发生过什么。 它是针对历史的文档，本身就有价值，不能乱改。 从这个角度看来，改变提交历史是一种亵渎，你使用\_谎言\_掩盖了实际发生过的事情。 如果由合并产生的提交历史是一团糟怎么办？ 既然事实就是如此，那么这些痕迹就应该被保留下来，让后人能够查阅。

另一种观点则正好相反，他们认为提交历史是 项目过程中发生的事。 没人会出版一本书的第一版草稿，软件维护手册也是需要反复修订才能方便使用。 持这一观点的人会使用 rebase 及 filter-branch 等工具来编写故事，怎么方便后来的读者就怎么写。

总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。



git show可以用来查看已经提交的commit详情或者是分支：

git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7

git show HEAD

git show HEAD^ # to see the parent of HEAD

git show HEAD^^ # to see the grandparent of HEAD

git show HEAD~4 # to see the great-great grandparent of HEAD #就是回溯4步

也就是使用 git show就是查看那时候的git diff

创建一个含附注类型的标签非常简单，用 -a （译注：取 annotated 的首字母）指定标签名字即可：

$ git tag -a v1.4 -m 'my version 1.4' #有没有-a参数都行

$ git tag

v0.1

v1.3

v1.4

git remote add origin <https://github.com/try-git/try_git.git> （https协议，还有git协议）建立origin远程仓库

git push -u origin master 将本地master分支的更新推送到运远程服务器的origin分支，同时记录这个命令（-u 参数），之后git push就相当于执行当前命令。

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

git merge <branch> 将其他分支合并到当前分支，如果要合并远程分支，在web界面上处理merge request或者本地merge后上传。

存档：

git archive master --format=zip --output=../website-12-10-2012.zip其实这相当于使用了tar来打包

git bundle create <file> <branch-name> bundle打包得到的文件就当做远程仓库来使用

当我们的工作还没有完成这时候还不想提交，那么就使用git stash来隐藏起来，文件也会返回到你之前什么都没有做的时候，git stash apply来恢复。

Git submodule：

Git 子模块功能允许你将一个Git仓库当作另外一个Git仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。

在项目中可能需要包含一些库文件或者一些必要的源文件以调用，使用子模块可以方便地将也是git管理的第三方库的更新合并到自己更改的库里面或者方便我们为lib贡献自己的代码。一种方式是把它们下载进来然后git add进去，但是这并不是最干净的办法，应该使用submodules。如果一个项目使用了submodules，那么git clone的时候需要使用参数—recuse-submodules或者在git clone之后再执行git submodule update --init --recursive。

mkdir lib && cd lib

git submodule add https://github.com/xxxxxxxx

git会自动创建.gitmodules，其内容：

[submodule ”lib/xxxxx”]

path=lib/xxx

url=https://github.com/xxxxxxxx

submodule的删除：

git submodule deinit lib/xxxxx

git rm lib/xxxxx

git clone会下载一个项目与它的整个代码历史。相当于.git也会被下载过来。

git flow工作流:

相对使用仅有的一个master分支，Gitflow工作流使用2个分支来记录项目的历史。master分支存储了正式发布的历史，而develop分支作为功能的集成分支。这样也方便master分支上的所有提交分配一个版本号。你可以存在功能分支，但是功能分支只能是从develop分支中拉出来，然后要回归到develop分支。维护分支或说是热修复（hotfix）分支用于生成快速给产品发布版本（production releases）打补丁，这是唯一可以直接从master分支fork出来的分支。修复完成，修改应该马上合并回master分支和develop分支（当前的发布分支），master分支应该用新的版本号打好Tag。Release这个分支的创建意味着一个发布周期的开始，也意味着本次发布不会再增加新的功能。



git flow init

git flow feature start/finish xxxx

git flow feature help

git flow release start/finish v1.0

远程操作:

在本地添加远程分支：git remote add origin https://github.com/xiaololo/myTest.git

删除远程分支：

$ git branch -a

\* master

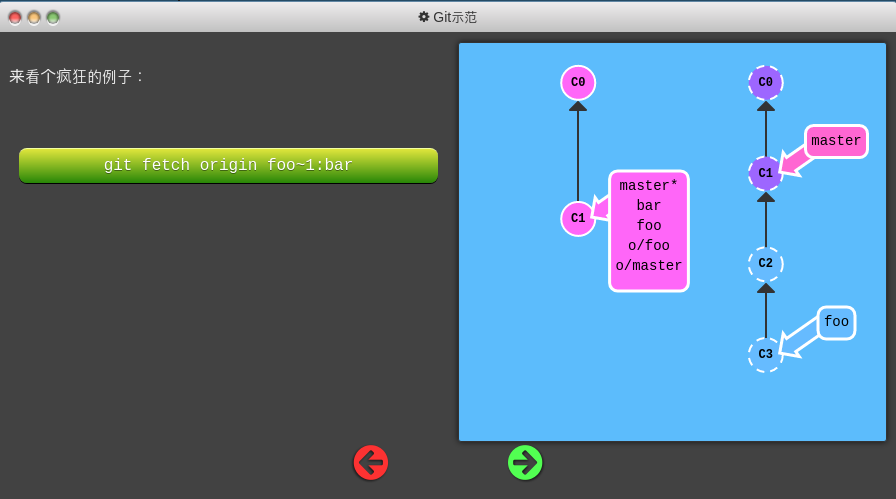
remotes/ExplosiveBattery/add-license-1

remotes/ExplosiveBattery/master

所以执行git push ExplosiveBattery --delete add-license-1，会要求输入github的账号与密码。

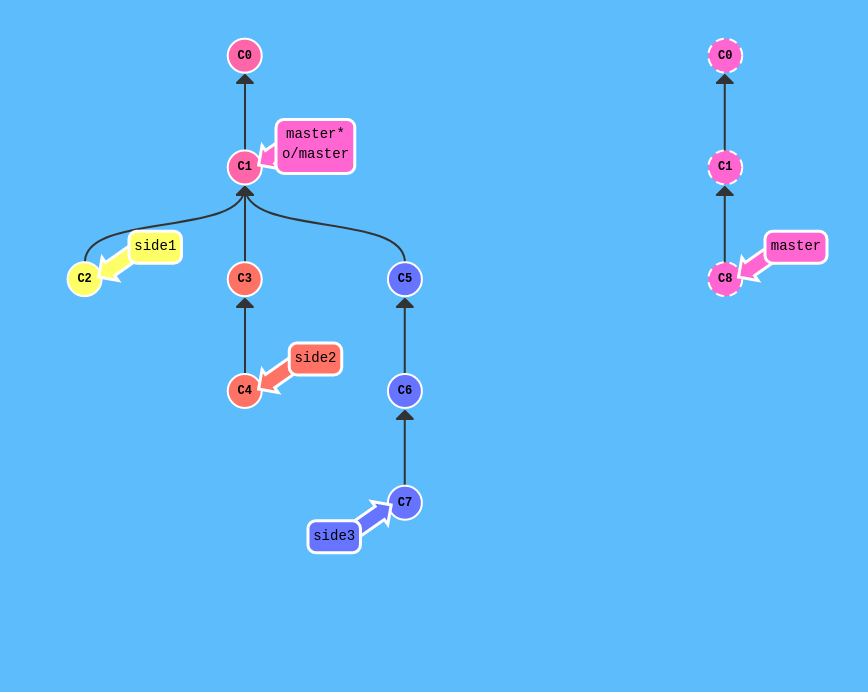
git branch -ｒ查看远程分支

git fetch 更新远程分支(下载缺失节点,并移动本地远程分支指向),但是不会对本地其余分支造成影响.git fetch没有参数的时候就会下载所有的提交记录到各个远程分支.



如果本地没有bar分支就会创建一个

git pull <=> git fetch; git merge <remote-branch-in-local> 更新远程分支并将内容合并到当前分支



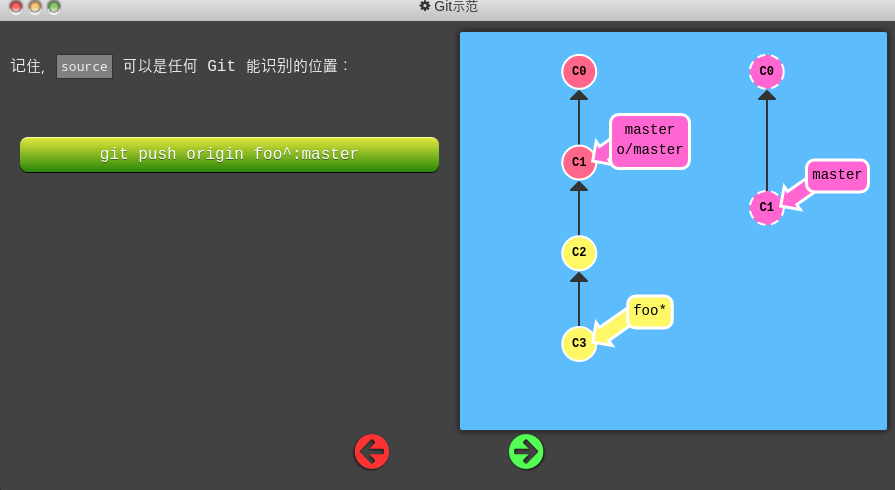
注意:远程仓库中分支的名字叫做master,所以git pull之前要先将HEAD切到本地的master,因为后面还有一步merge。但是merge不是完全替换掉本地的内容，完全替换应该是git clone。

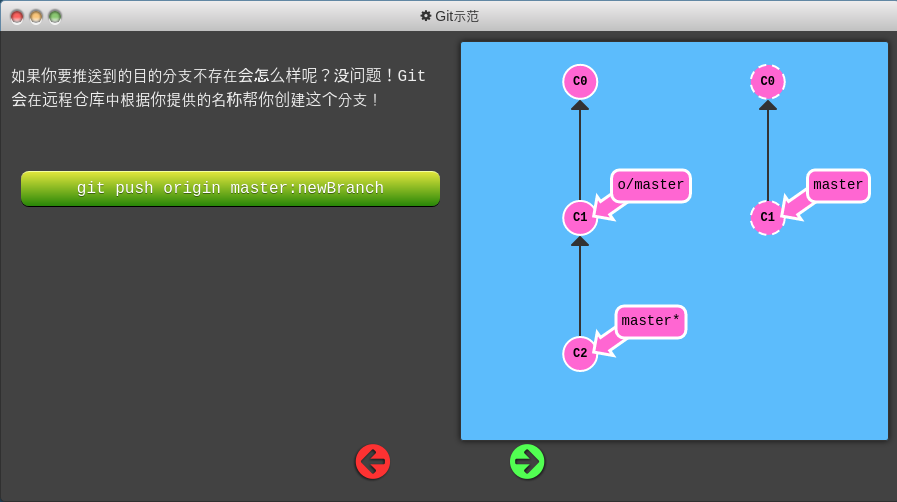
git branch --set-upstream-to=origin/master master 后git pull相当于git pull origin master

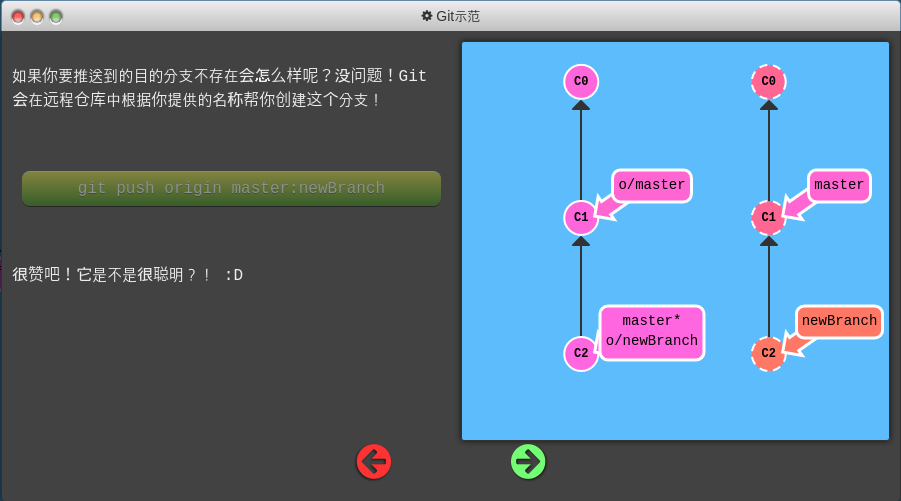
git push不带参数时的行为与Git的一个名为push.default的配置有关.它的默认值取决与你正使用的Git版本.git push之后,本地的远程分支也会移动到当前位置,因为这是远程仓库对应分支的最新状态.

git push origin master 切到本地master分支,尝试git push到origin/master.直接git push则是以当前HEAD所指分支来尝试.

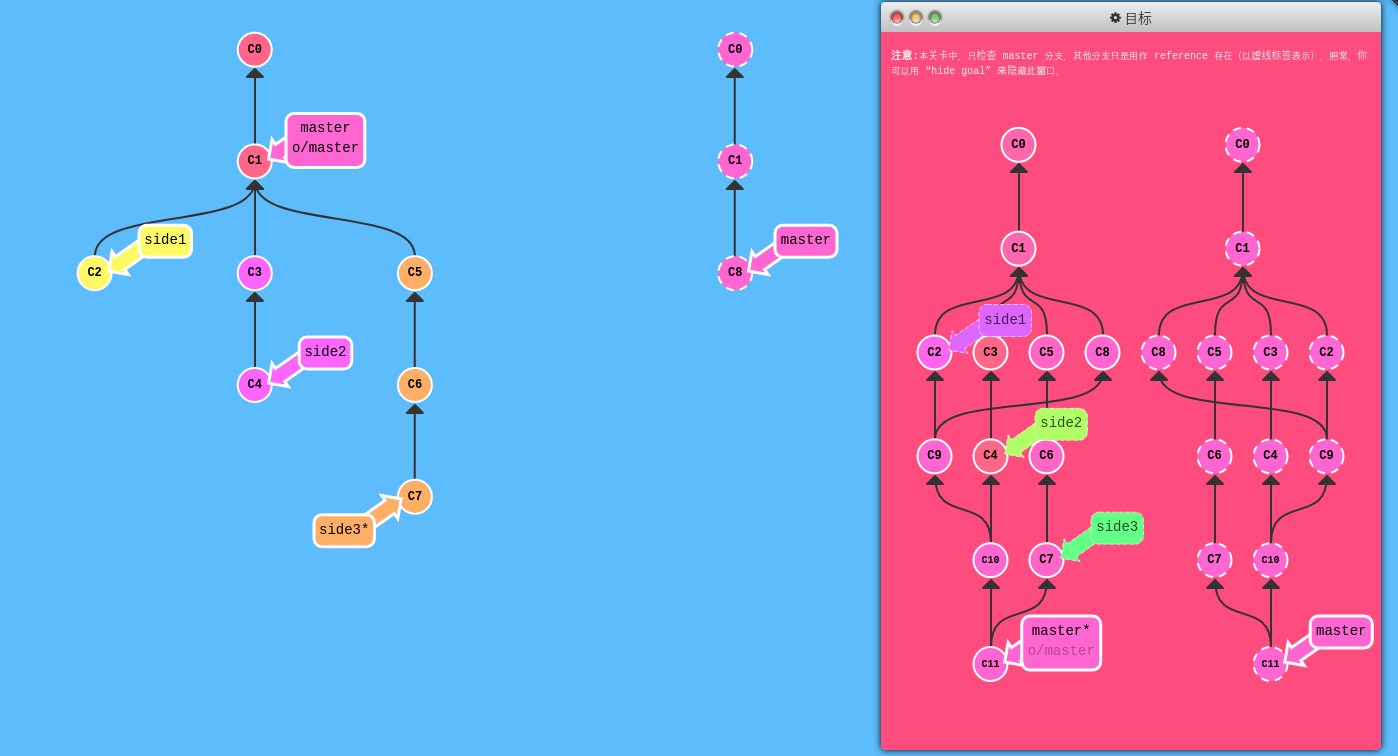
git push origin <local-branch>:<remote-branch> #下面的图片表明branch是一个Node为终点的路径 而不是 分支







如果本地的远程分支比远程仓库中的对应分支旧(少了几个新加入的节点,也就是说你还没有更新本地远程分支),git会拒绝你的push,解决办法:git fetch;git rebase origin/master(也可以使用git merge);git push=>git pull --rebase(上面git merge对应这里没有--rebase);git push



注意:git push的时候把本地的变换过程完整push过去了,所以使用git pull

--rebase 比 git pull 看起来关系图会简单很多.

<local-branch>为空时,将删除指定的<remote-branch>. 或者使用git push -d origin <remote-branch>

git lfs：large file storage

这条命令用于上传大文件

Octotree 浏览器插件在firefox上不能下载单文件，但是在chrome上加上翻墙就可以做到。下载单文件可以点到文件那一面，然后对“raw”进行另存为。

github对超过1mb的文件无法直接在浏览器中查看的限制

github有一个很有爱的项目，叫做github pages，这个项目是给开发者建立一个私人页面，上面用来分享新颖的想法和自己写的代码，而且最主要的是，这个是免费而且没有空间流量限制的。这也就是我为什么放弃了自由度很高的，却需要支付高昂的主机费的wordpress，而转投了github pages阵营（自带CDN）：

建立一个仓库并自动初始化，要求仓库名字是”用户名.github.io”，还可以设置自己的域名。我是想用blog.hellovega.cn，需要自己配置一条子域名到github的CNAME记录，然后在gihub的custom domain中告知自己的子域名（然后它会自己在仓库创建一个CNAME文件内容就是blog.hellovega.cn），否则你只有CNAME的话解析到是404，因为提交过去的请求头中host字段带有blog.hellovega.cn。这样以后这两个网址ExplosiveBattery.github.io与blog.hellovega.cn就连接在一起，最后访问的都会是blog.hellovega.cn，显示的都是ExplosiveBattery.github.io（当然域名是不会有大写字母的，浏览器会自动转化，路径中的大写字母才会被浏览器允许）的内容。如果只是制定了github上关于域名的链接而没有加上自己的cname，那么访问域名会返回502 Server dropped connection（Edge和360安全浏览器，似乎还有web代理，估计这是edge的锅；如果是chrome，应该就是DNS无法解析），

前端模板：<http://jekyllthemes.org/>

Hexo我感觉是个不错的前端框架（使用主题https://github.com/iissnan/hexo-theme-next）

被推荐<http://jekyllthemes.org/themes/cool-concise-high-end/>，认为后期可塑性强。

打开存放你克隆下来仓库的文件夹，把里面的文件全部都删了（没错），除了隐藏文件夹“.git”不要删就好了，然后把模板里的东西全部拖到你的原博客仓库里

一个Jekyll规范的博客的基本内容了，想想也不难吧:

index.html：这是你博客的主页面，里面的内容就是你的主页了

\_config.yml：这是你博客的基本配置文件，里面有你博客的名字，以及存放博主的一些基本信息

\_layouts：这文件夹里面存放你每个页面的设计，一般有default.html（默认页面）和posts.html（博文页面）

\_includes：这个文件夹里的的内容将会通用到你博客每个页面，起到一种便利的作用

\_posts：这里面装的就是你的博文啦，记住，要用markdown语法写，要不上传会失败的。

跟踪分支*是一种和远程分支有直接联系的*本地分支

从git gui 可以知道git配置了一个数据库，估计是sqlite3。

git bash有点意思，支持ls、reset等linux命令。

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git blame test.txt

^7850f0d (Battery 2017-06-16 08:51:07 +0800 1) 123

最前面的^7850f0d 是一个正则表达式来表示开头。

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git log

commit 7850f0da677b58d0dd11d4981467d1abf6e8ec60

Author: Battery <stegano@foxmail.com>

Date: Fri Jun 16 08:51:07 2017 +0800

test

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git branch new

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git branch -av

\* master 7850f0d test

new 7850f0d test

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git branch

\* master

new

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git mergetool

This message is displayed because 'merge.tool' is not configured.

See 'git mergetool --tool-help' or 'git help config' for more details.

'git mergetool' will now attempt to use one of the following tools:

opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc codecompare emerge vimdiff

No files need merging

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (new)

$ git diff 我并没有删除123，我只是单纯地加上一行

**diff --git a/test.txt b/test.txt**

**index d800886..c028161 100644**

**--- a/test.txt**

**+++ b/test.txt**

@@ -1 +1,2 @@

-123

\ No newline at end of file

+123

+new branch

\ No newline at end of file

Battery@DESKTOP-FRNFA24 MINGW64 ~/Desktop/Program/test (master)

$ git status 查看当前目录中还没有被git管理的文件以及被管理以后还没有提交保存的文件

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: testCommitA.txt

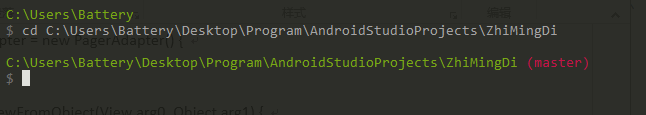
new file: testtracked.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

am.txt

cmder 还是很高级的： 可以直接识别git建立的仓库



Git ignore：

规则在.gitignore（需要自己创建，不支持正则表达式，\*.a表示所有以a结尾的文件，a/表示文件夹a下的所有文件，不要斜杠也可以但是缺乏可读性）（还有个不建议的地方就是.git/info/exclude）

只有文件名意味着所有叫这个名字的文件都会被ignore，所以指定文件的话你还需要在前面加上自己的目录。可以使用\*通配符。如果是感叹号开头，表示取反，也就是感叹号后面路径中的内容会被添加到版本管理。

Only untracker files can be ignored. untracked文件就会在git status一直提醒你，加入ignore就会被完全忽略，这就是untracked与ignore的区别。

**.gitignore文件** 只是针对远程push

如果git已经在管理文件了（之前push过文件），那么你的ignore规则不会起到作用。

1）/mtk/               过滤整个文件夹  
2）/mtk/do.c         过滤某个具体文件

3）\*.zip                过滤所有.zip文件

以斜杠“/”开头表示目录；还是表示当前文件夹  
以星号“\*”通配多个字符；  
以问号“?”通配单个字符  
以方括号“[]”包含单个字符的匹配列表；  
以叹号“!”表示不忽略(跟踪)匹配到的文件或目录； 从而选出某一个文件夹中那个未闭环不应该被ignore

如果不指定具体的路径，就会对所有的名字生效

github的Create new file输入/就会创建一个文件夹

如果一开始忘了设置License，那么创建一个这个文件就好，名字一写右边就会有一个选择模板的按钮

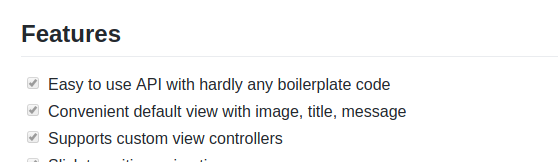
README.md(创建文件的时候不要直接README，不然少了加成，好处是普通文本模式所见即所得)：

参考：<https://github.com/meolu/walle-web>

https://github.com/Orderella/PopupDialog

icons：<https://shields.io/>

gifs:<img src="https://github.com/xxxxxxxx.gif?raw=true" width="300">



## Features

- [x] Easy to use API with hardly any boilerplate code

- [x] Convenient default view with image, title, message

- [x] Supports custom view controllers

间隔：<p>&nbsp;</p>

最后以# License结尾。

**GitHook：**

一般代码编写完git push到github，然后通过ssh登上服务器执行git pull重新部署。使用githook原理：

1. git用户执行git push操作
2. 远程仓库发现有用户执行了push操作，就会执行一个脚本post-receive（钩子）
3. 在post-receive脚本中，将git仓库的代码拷贝到web站点目录下

看网上的教程不少人会用到git init --bare 在服务器端创建一个所谓的裸仓库，实际上就是将.git中内容放到了项目根目录中。

.git有一个hooks文件夹，见https://git-scm.com/book/zh/v1/%E8%87%AA%E5%AE%9A%E4%B9%89-Git-Git%E6%8C%82%E9%92%A9，主要是其中的post-receive脚本。

**问题：**

在git add文件的时候出现warning: LF will be replaced by CRLF 。这个错误是正常的，没想到windows中的as创建的项目居然是LF作为结尾。其实git应该统一一下这个换行符问题。

在git删文件的时候出现error: the following files have changes staged in the index: 我采用的办法是加上-f参数强制删除。use --cached to keep the file, or -f to force removal

**Gist :**<https://gist.github.com/>

一种markdown类似的即时网页分享的写作平台

**1. 匿名张贴**您不需要拥有Github账号就可以使用Gist。用浏览器打开[http://gist.github.com](https://link.zhihu.com/?target=http%3A//gist.github.com)，在窗口中写下你想说的就可以创建一个Gist。您可以发布一个私密的Gist，也就是说这个Gist将不能被他人搜索到而只对直接在浏览器中输入其URL的人可见。

**2. 能像wiki一样记录历史**  
如果您修改了已经发布了的Gist的话，之前的所有版本都将被保存。您可以点击Revisions按钮按时间浏览，而且您可以通过内置的diff引擎查看任意两个版本间的差异。 这也可以用于比较文本文件。

**Devops：** Development+Operations

<https://www.cnblogs.com/liufei1983/p/7152013.html>