Git 学习笔记

# 一、安装

下载dmg文件<http://git-scm.com/downloads>

在Mac10.4和 10.5,如果你安装了MacPorts,你可以通过 MacPorts来安装Git。当你安装好MacPorts后，你可通过下面的命令来安装:

$ sudo port install git-core

在Windows下安装Git是很简单的，你只要下载msysGit就可以了。

# 二、配置

使用Git的第一件事就是设置你的名字和email,这些就是你在提交commit时的签名。

$ git config --global user.name "Scott Chacon"

$ git config --global user.email "schacon@gmail.com"

执行了上面的命令后,会在你的主目录(home directory)建立一个叫 ~/.gitconfig 的文件. 内容一般像下面这样:

[user]

name = Scott Chacon

email = schacon@gmail.com

译者注:这样的设置是全局设置,会影响此用户建立的每个项目。

如果你想使项目里的某个值与前面的全局设置有区别(例如把私人邮箱地址改为工作邮箱);你可以在项目中使用git config 命令不带 --global 选项来设置. 这会在你项目目录下的 .git/config 文件增加一节[user]内容(如上所示)。

# 创建仓库

## 3.1本地创建

创建新文件夹，打开，然后执行git init以创建新的 git 仓库。

然后需要在服务器端新建远程仓库，再通过git add name <serverurl> 关联（当然如果不希望在远程服务器端共享，可以不push，git同样支持本地离线版本管理）

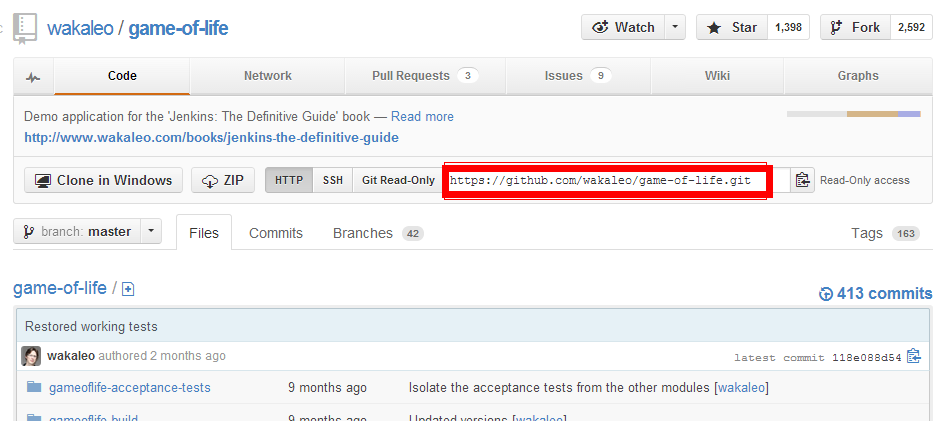
|  |
| --- |
| 1.git init  2.git remote add coder <https://github.com/tomserious/HelloWorld.git>  3. git push --set-upstream coder abc2016Coder（指定此次推送的上游分支是coder）  更好的方法是git push -u coder abc2016Coder 参数u可以把本地分支和远程分支关联，下次push直接git push coder即可 |

## 3.2 clone

git clone URL命令可以得到仓库的复制版本

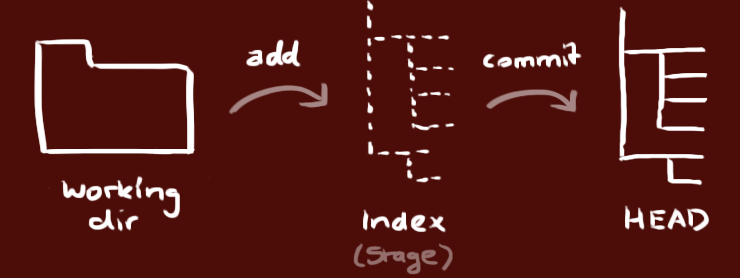
* 执行如下命令以创建一个本地仓库的克隆版本：
  + git clone /path/to/repository
* 如果是远端服务器上的仓库，你的命令会是这个样子：
  + git clone git://git.kernel.org/pub/scm/git/git.git
  + git clone <http://www.kernel.org/pub/scm/git/git.git>

其中远程服务器端URL为图中红框部分，可以是HTTP格式也可以是SSH格式，推荐使用SSH格式地址



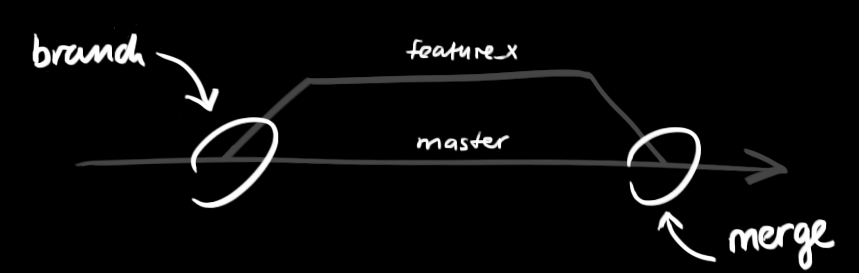
# 四、工作流

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 **缓存区（Index**），它像个缓存区域，临时保存你的改动；最后是 **HEAD**，指向你最近一次提交后的结果。



# 五、分支

分支是用来将特性开发绝缘开来的。在你创建仓库的时候，master 是“默认的”。在其他分支上进行开发，完成后再将它们合并到主分支上。



**查看**当前仓库中存在的所有分支列表：

$ git branch

git branch -a 查看全部分支

“master”分支是Git系统默认创建的主分支。星号(“\*”)标识了你当工作在哪个分支下

**创建**一个叫做“feature\_x”的分支，并切换过去：

git checkout -b feature\_x

**切换**回主分支：

git checkout master

再把新建的分支**删掉**：

git branch -d feature\_x

git branch -d只能删除那些已经被当前分支合并的分支. 如果你要强制删除某个分支的话就用git branch –D

除非你将分支**推送**到远端仓库，不然该分支就是 不为他人所见的：

git push origin <branch>

**追踪**分支：

你可以在使用'git branch'命令时加上'--track'参数, 来手动创建一个'追踪分支'.

git branch --track experimental origin/experimental

当你运行下命令时:

$ git pull experimental

它会自动从‘origin'抓取(fetch)内容，再把远程的'origin/experimental'分支合并进(merge)本地的'experimental'分支.

当要把修改推送(push)到origin时, 它会将你本地的'experimental'分支中的修改推送到origin的‘experimental'分支里,　而无需指定它(origin).

git checkout –b ipad origin/master –no-track 从master分支checkout到本地建立一个名叫ipad的分支

# 六、添加与提交

## 6.1本地提交

执行git status可以查看本地操作对原始仓库做了哪些更改, 即: 哪些文件被暂存了(就是在你的Git索引index中), 哪些文件被修改了但是没有暂存, 还有哪些文件没有被跟踪(untracked).

你可以计划改动（把它们添加到index缓存区），使用如下命令：

git add <filename>

git add \*

这是 git 基本工作流程的第一步；使用如下命令以实际提交改动：

git commit -m "代码提交信息"

现在，你的改动已经提交到了 HEAD，但是还没到你的远端仓库。

除了用git add 命令，我还可以用

$ git commit –a

这会自动把所有内容被修改的文件(不包括新创建的文件)都添加到索引index中，并且同时把它们提交。

## 6.2推送到远程

你的改动现在已经在本地仓库的 HEAD 中了。执行如下命令以将这些改动提交到远端仓库：

git push origin master

可以把 master 换成你想要推送的任何分支。

如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：

git remote add origin <server>

如此你就能够将你的改动推送到所添加的服务器上去了。

**推送失败**：

如果推送(push)结果不是"快速向前"(fast forward),那么它 可能会报像下面一样的错误：

error: remote 'refs/heads/master' is not an ancestor of

local 'refs/heads/master'.

Maybe you are not up-to-date and need to pull first?

error: failed to push to 'ssh://yourserver.com/~you/proj.git'

这种情况通常由以下的原因产生：

- 用 `git-reset --hard` 删除了一个**已经发布**了的一个提交，或是

- 用 `git-commit --amend` 去替换一个**已经发布**的提交，或是

- 用 `git-rebase` 去rebase一个**已经发布**的提交.

你以强制git-push在上传修改时先更新，只要在分支名前面加一个加号。

$ git push ssh://yourserver.com/~you/proj.git +master

**提交范例**

|  |
| --- |
| git status //查看当前本地更改状态  git add \* //全部添加到缓冲区  git status //查看当前本地更改状态  git commit –m “add frank test cases” //提交缓冲区更改到HEAD  git push Jgeng //推送到远端仓库，因为已经设置为追踪分支，故不需要跟本地分支名 |

## 6.3忽略某些文件

项目中经常会生成一些Git系统不需要追踪(track)的文件。典型的是在编译生成过程中 产生的文件或是编程器生成的临时备份文件。当然，你不追踪(track)这些文件，可以 平时不用"git add"去把它们加到索引中。 但是这样会很快变成一件烦人的事，你发现 项目中到处有未追踪(untracked)的文件; 这样也使"git add ." 和"git commit -a" 变得实际上没有用处，同时"git status"命令的输出也会有它们。

你可以在你的顶层工作目录中添加一个叫".gitignore"的文件，来告诉Git系统要忽略 掉哪些文件，下面是文件内容的示例:

# 以'#' 开始的行，被视为注释.

# 忽略掉所有文件名是 foo.txt 的文件.

foo.txt

# 忽略所有生成的 html 文件,

\*.html

# foo.html是手工维护的，所以例外.

!foo.html

# 忽略所有.o 和 .a文件.

\*.[oa]

你也可以把".gitignore" 这个文件放到工作树(working tree)里的其它目录中，这就会在它和它的子目录起忽略(ignore) 指定文件的作用。.gitignor文件同样可以像其它文件一样加到项目仓库里( 直接用 git add .gitignore 和 git commit等命令), 这样项目里的其它开发者也能共享同一套忽略 文件规则。

如果你想忽略规则只对特定的仓库起作用,你可以把这些忽略规则写到你的仓库下 .git/info/exclude文件中，或是写在Git配置变量core.excludesfile中指定的 文件里。有些Git命令也可在命令行参数中指定忽略规则。

# 七、更新与合并

要更新你的本地仓库至最新改动，执行：git pull以在你的工作目录中 获取（fetch） 并 合并（merge） 远端的改动。

要合并其他分支到你的当前分支（例如 master），执行：

git merge <branch>

两种情况下，git 都会尝试去**自动合并改动到当前分支**。最后通过push更新到远端服务器。不幸的是，自动合并并非次次都能成功，并可能导致 冲突（conflicts）。 这时候就需要你修改这些文件来人工合并这些 冲突（conflicts） 了。改完之后，你需要执行如下命令以将它们标记为合并成功：

git add <filename>

在合并改动之前，也可以使用如下命令查看：

git diff <source\_branch> <target\_branch>

如果你要经常操作远程分支(remote branch),你可以定义它们的缩写:

$ git remote add bob /home/bob/myrepo

这样，就可以用"git fetch"" 来执行"git pull"前半部分的工作， 但是这条命令并不会把抓下来的修改合并到当前分支里。

$ git fetch bob

我们用git remote命令建立了Bob的运程仓库的缩写，用这个(缩写) 名字我从Bob那得到所有远程分支的历史记录。在这里远程分支的名 字就叫bob/master.

$ git log -p master..bob/master

上面的命令把Bob从Alice的主分支(master)中签出后所做的修改全部显示出来。

当检查完修改后,Alice就可以把修改合并到她的主分支中。

$ git merge bob/master

这种合并(merge)也可以用pull来完成，就像下面的命令一样：

$ git pull . remotes/bob/master

注意：git pull 会把远程分支合并进当前的分支里，而不管你在命令 行里指定什么。

其后，Bob可以更新它的本地仓库--把Alice做的修改拉过来(pull):

$ git pull

如果Bob从Alice的仓库克隆(clone)，那么他就不需要指定Alice仓库的地 址；因为Git把Alice仓库的地址存储到Bob的仓库配库文件，这个地址就是 在git pull时使用：

$ git config --get remote.origin.url

/home/alice/project

(如果要查看git clone创建的所有配置参数，可以使用"git config -l", git config 的帮助文件里解释了每个参数的含义.)

Git同时也保存了一份最初(pristine)的Alice主分支(master)，在 "origin/master"下面。

$ git branch -r

origin/master

|  |
| --- |
| 操作范例：  git remote –v  git remote add source [git@git.corp.adobe.com:dms-mobile/mobile-mc.git](mailto:git@git.corp.adobe.com:dms-mobile/mobile-mc.git) //定义远端仓库缩写为source  git remote –v  git fetch source //获取  git merge source/master //合并  git stash |

# 八、替换本地改动

## 8.1未提交

假如你做错事，你可以使用如下命令替换掉本地改动：

git checkout -- <filename>

此命令会使用 HEAD 中的最新内容替换掉你的工作目录中的文件。已添加到缓存区的改动，以及新文件，都不受影响。

假如你想要丢弃你所有的本地改动与提交，可以到服务器上获取最新的版本并将你本地主分支指向到它：

git fetch origin

git reset --hard origin/master

或者

$ git reset --hard HEAD

这条件命令会把你工作目录中所有未提交的内容清空(当然这不包括未置于版控制下的文件 untracked files). 从另一种角度来说, 这会让"git diff" 和"git diff --cached"命令的显示法都变为空.

## 8.2已提交

或者你已经把合并后的代码提交，但还是想把它们撒销：

$ git reset --hard ORIG\_HEAD

但是刚才这条命令在某些情况会很**危险**，如果你把一个已经被另一个分支合并的分支给删了，那么 以后在合并相关的分支时会出错。

如果你已经做了一个提交(commit),但是你马上后悔了, 这里有两种截然不同的方法去处理这个问题:

* 创建一个新的提交(commit), 在新的提交里撤消老的提交所作的修改. 这种作法在你已经把代码发布的情况下十分正确.
* 你也可以去修改你的老提交(old commit). 但是如果你已经把代码发布了,那么千万别这么做; git不会处理项目的历史会改变的情况,如果一个分支的历史被改变了那以后就不能正常的合并.

### 创建新提交

创建一个新的，撤消(revert)了前期修改的提交(commit)是很容易的; 只要把出错的提交(commit)的名字(reference)做为参数传给命令: git revert就可以了; 下面这条命令就演示了如何撤消最近的一个提交:

$ git revert HEAD

这样就创建了一个撤消了上次提交(HEAD)的新提交, 你就有机会来修改新提交(new commit)里的提交注释信息.

你也可撤消更早期的修改, 下面这条命令就是撤消“上上次”(next-to-last)的提交:

$ git revert HEAD^

在这种情况下,git尝试去撤消老的提交,然后留下完整的老提交前的版本.　如果你最近的修改和要撤消的修改有重叠(overlap),那么就会被要求手工解决冲突(conflicts),　就像解决合并(merge)时出现的冲突一样.

译者注: git revert 其实不会直接创建一个提交(commit), 把撤消后的文件内容放到索引(index)里,你需要再执行git commit命令，它们才会成为真正的提交(commit).

### 修改老提交

如果你刚刚做了某个提交(commit), 但是你又想马上修改这个提交; git commit 现在支持一个叫--amend的参数，它能让你修改刚才的这个提交(HEAD commit). 这项机制能让你在代码发布前,添加一些新的文件或是修改你的提交注释(commit message).

# 九、比较

你可以用 git diff 来比较项目中任意两个版本的差异。

$ git diff master..test

上面这条命令只显示两个分支间的差异，如果你想找出‘master’,‘test’的共有 父分支和'test'分支之间的差异，你用3个‘.'来取代前面的两个'.' 。

$ git diff master...test

如果你要查看当前的工作目录与另外一个分支的差别，你可以用下面的命令执行:

$ git diff test

这会显示你当前工作目录与另外一个叫'test'分支的差别。你也以加上路径限定符，来只 比较某一个文件或目录。

$ git diff HEAD -- ./lib

上面这条命令会显示你当前工作目录下的lib目录与上次提交之间的差别(或者更准确的 说是在当前分支)。

如果不是查看每个文件的详细差别，而是统计一下有哪些文件被改动，有多少行被改 动，就可以使用‘--stat' 参数。

$ git diff fullFileName

可以查看单独文件上与上次提交的差异，不过是在git add 该文件**之前，**如果查看git add之后但未commit之前，要加--staged参数。

## 哪些内容将被提交

你通常用git diff来找你**当前工作目录**和上次提交与本地索引间的差异。

$ git diff

上面的命令会显示在当前的工作目录里的，没有 staged(添加到索引中)，且在下次提交时 **不会被提交**的修改。

如果你要看在下次提交时**要提交的内容**(staged,添加到索引中),你可以运行：

$ git diff --cached

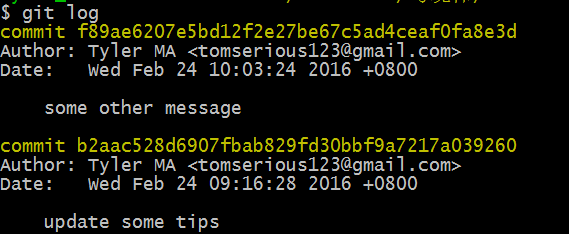
上面的命令会显示你当前的索引和上次提交间的差异；这些内容在不带"-a"参数运行 "git commit"命令时就会被提交。

$ git diff HEAD

上面这条命令会显示你工作目录与上次提交时之间的**所有差别**，这条命令所显示的 内容都会在执行"git commit -a"命令时被提交。

# 十、日志

git log命令可以显示目前所在分支上，所有的提交(commit)。



git show 接commit 后面的序列号可以查看特定提交的更改详细信息

当然你也可以组合上面的命令选项；下面的命令就是找出所有从"v2.5“开 始在fs目录下的所有Makefile的修改.

$ git log v2.5.. Makefile fs/

你也可以让git log显示补丁(patchs)，即每次提交的具体更改内容:

$ git log –p

如果用--stat选项使用'git log',它会显示在每个提交(commit)中哪些文件被修改了, 这些文件分别添加或删除了多少行内容.

$ git log --stat

使用oneline参数可以简洁形式列出所有日志，每次一行

$ git log --oneline

## 10.1格式化日志

你可以按你的要求来格式化日志输出。‘--pretty'参数可以使用若干表现格式，如‘oneline':

或者你也可以使用 'short' 格式:

你也可用‘medium','full','fuller','email' 或‘raw'. 如果这些格式不完全符合你的相求， 你也可以用‘--pretty=format'参数(参见：git log)来创建你自己的"格式“.

$ git log --pretty=format:'%h was %an, %ar, message: %s'

另一个有趣的事是：你可以用'--graph'选项来可视化你的提交图(commit graph),就像下面这样:

$ git log --pretty=format:'%h : %s' --graph

它会用ASCII字符来画出一个很漂亮的提交历史(commit history)线。

## 10.2日志排序

你也可以把日志记录按一些不同的顺序来显示。注意，git日志从最近的提交(commit)开始，并且从这里开始向它们父分支回溯。然而git历史可能包括多个互不关联的开发线路，这样有时提交(commits)显示出来就有点杂乱。

如果你要指定一个特定的顺序，可以为git log命令添加顺序参数(ordering option).

按默认情况，提交(commits)会按逆时间(reverse chronological)顺序显示。

但是你也可以指定‘--topo-order'参数，这就会让提交(commits)按拓朴顺序来显示(就是子提交在它们的父提交前显示). 如果你用git log命令按拓朴顺序来显示git仓库的提交日志，你会看到“开发线"(development lines)都会集合在一起.

你也可以用'--date-order'参数，这样显示提交日志的顺序主要按提交日期来排序. 这个参数和'--topo-order'有一点像，没有父分支会在它们的子分支前显示，但是其它的东东还是按交时间来排序显示。你会看到"开发线"(development lines)没有集合一起，它们会像并行开发(parallel development)一样跳来跳去的

最后，你也可以用 ‘--reverse'参数来逆向显示所有日志。

# 十一、标签

在软件发布时创建标签，是被推荐的。这是个旧有概念，在 SVN 中也有。可以执行如下命令以创建一个叫做 1.0.0 的标签：

git tag 1.0.0 1b2e1d63ff

这样，我们可以用1.0.0 作为提交(commit) "1b2e1d63ff" 的代称(refer)，1b2e1d63ff 是你想要标记的提交 ID 的前 10 位字符。使用如下命令获取提交 ID：

git log

你也可以用该提交 ID 的少一些的前几位，只要它是唯一的。

如果有 "-a", "-s" 或是 "-u " 中间的一个命令参数被指定，那么就会创建 一个标签对象，并且需要一个标签消息(tag message). 如果没有"-m " 或是 "-F " 这些参数，那么就会启动一个编辑器来让用户输入标签消息(tag message).

译者注：大家觉得这个标签消息是不是提交注释(commit comment)比较像。

当这样的一条命令执行后，一个新的对象被添加到Git对象库中，并且标签引用就指向了 一个标签对象，而不是指向一个提交(commit). 这样做的好处就是：你可以为一个标签 打处签名(sign), 方便你以后来查验这是不是一个正确的提交(commit).

## 签名的标签

如果你配有GPG key,那么你就很容易创建签名的标签.首先你要在你的 .git/config 或 ~.gitconfig里配好key.

下面是示例:

[user]

signingkey = <gpg-key-id>

你也可以用命令行来配置:

$ git config (--global) user.signingkey <gpg-key-id>

现在你可以直接用"-s" 参数来创“签名的标签”。

$ git tag -s stable-1 1b2e1d63ff

如果没有在配置文件中配GPG key,你可以用"-u“ 参数直接指定。

$ git tag -u <gpg-key-id> stable-1 1b2e1d63ff

# 十二、暂存

当你正在做一项复杂的工作时, 发现了一个和当前工作不相关但是又很讨厌的bug. 你这时想先修复bug再做手头的工作, 那么就可以用 git stash 来保存当前的工作状态, 等你修复完bug后,执行'反储藏'(unstash)操作就可以回到之前的工作里.

$ git stash "work in progress for foo feature"

上面这条命令会保存你的本地修改到储藏(stash)中, 然后将你的工作目录和索引里的内容全部重置, 回到你当前所在分支的上次提交时的状态.

好了, 你现在就可以开始你的修复工作了.

... edit and test ...

$ git commit -a -m "blorpl: typofix"

当你修复完bug后, 你可以用git stash apply来恢复到以前的工作状态.

$ git stash apply

你也可多次使用'git stash'命令,　每执行一次就会把针对当前修改的‘储藏’(stash)添加到储藏队列中. 用'git stash list'命令可以查看你保存的'储藏'(stashes):

$>git stash list

stash@{0}: WIP on book: 51bea1d... fixed images

stash@{1}: WIP on master: 9705ae6... changed the browse code to the official repo

可以用类似'git stash apply stash@{1}'的命令来使用在队列中的任意一个'储藏'(stashes). 'git stash clear‘则是用来清空这个队列.

# 十三、Tips

彩色的 git 输出：git config color.ui true

交互地添加文件至缓存区：git add –i

文件名有空格的，应该把文件名放在引号内

<http://www.01happy.com/git-resolve-conflicts/>

Git pull 冲突解决办法

**方案1：**

Git stash 暂存本地内容

Git pull 更新到本地

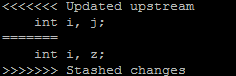
Git stash pop stash@{0} 还原暂存内容

系统提示如下类似信息

Auto-merging c/environ.c

CONFLICT (content): Merge conflict in c/environ.c

打开冲突文件，看到如下类似内容



其中Updated upstream 和=====之间的内容就是pull下来的内容，====和stashed changes之间的内容就是本地修改的内容。碰到这种情况，git也不知道哪行内容是需要的，所以要自行确定需要的内容。

方案2：

Both modified

手动解决冲突

Git add 重新添加

Git commit ,git push

参考资料：

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

社区手册：<http://gitbook.liuhui998.com/>