中级技能:

## 一、实验说明

从本节开始，我们会介绍一些中级和高级的用法，这些用法很少用到，前面三节的内容已经满足了日常工作需要，从本节开始的内容可以简单了解，需要的时候再详细查看。

### 1.1 下载测试项目环境

通过下列命令获得gitproject项目环境，该项目默认只有一个文件README.md，可以用来进行后续git实验

$ git clone http://git.shiyanlou.com/shiyanlou/gitproject

## 二、忽略某些文件

### 1.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]

## 三、rebase

### 1.rebase

假设你现在基于远程分支"origin"，创建一个叫"mywork"的分支。

$ git checkout -b mywork origin

现在我们在这个分支做一些修改，然后生成两个提交(commit)。

$ vi file.txt

$ git commit

$ vi otherfile.txt

$ git commit

但是与此同时，有些人也在"origin"分支上做了一些修改并且做了提交了。这就意味着"origin"和"mywork"这两个分支各自"前进"了，它们之间"分叉"了。 在这里，你可以用"pull"命令把"origin"分支上的修改拉下来并且和你的修改合并； 结果看起来就像一个新的"合并的提交"(merge commit): 但是，如果你想让"mywork"分支历史看起来像没有经过任何合并一样，你也许可以用git rebase:

$ git checkout mywork

$ git rebase origin

这些命令会把你的"mywork"分支里的每个提交(commit)取消掉，并且把它们临时保存为补丁(patch)(这些补丁放到".git/rebase"目录中),然后把"mywork"分支更新 到最新的"origin"分支，最后把保存的这些补丁应用到"mywork"分支上。 当'mywork'分支更新之后，它会指向这些新创建的提交(commit),而那些老的提交会被丢弃。 如果运行垃圾收集命令(pruning garbage collection), 这些被丢弃的提交就会删除。 在rebase的过程中，也许会出现冲突(conflict). 在这种情况，Git会停止rebase并会让你去解决冲突；在解决完冲突后，用"git-add"命令去更新这些内容的索引(index), 然后，你无需执行 git-commit,只要执行:

$ git rebase --continue

这样git会继续应用(apply)余下的补丁。 在任何时候，你可以用--abort参数来终止rebase的行动，并且"mywork" 分支会回到rebase开始前的状态。

$ git rebase --abort

## 四、交互式rebase

### 1.交互式rebase

你亦可以选择进行交互式的rebase。这种方法通常用于在向别处推送提交之前对它们进行重写。交互式rebase提供了一个简单易用的途径让你在和别人分享提交之前对你的提交进行分割、合并或者重排序。在把从其他开发者处拉取的提交应用到本地时，你也可以使用交互式rebase对它们进行清理。 如果你想在rebase的过程中对一部分提交进行修改，你可以在'git rebase'命令中加入'-i'或'--interactive'参数去调用交互模式。

$ git rebase -i origin/master

这个命令会执行交互式rebase操作，操作对象是那些自最后一次从origin仓库拉取或者向origin推送之后的所有提交。 若想查看一下将被rebase的提交，可以用如下的log命令：

$ git log github/master..

一旦你完成对提交信息的编辑并且退出编辑器，这个新的提交及提交信息会被保存起来。 如果指定进行'edit'操作，git会完成同样的工作，但是在对下一提交进行操作之前，它会返回到命令行让你对提交进行修正，或者对提交内容进行修改。 例如你想要分割一个提交，你需要对那个提交指定'edit'操作： 你会进入到命令行，撤销（revert）该提交，然后创建两个（或者更多个）新提交。假设提交21d80a5修改了两个文件，file1和file2，你想把这两个修改放到不同的提交里。你可以在进入命令行之后进行如下的操作：

$ git reset HEAD

$ git add file1

$ git commit -m 'first part of split commit'

$ git add file2

$ git commit -m 'second part of split commit'

$ git rebase --continue

交互式rebase的最后一个作用是丢弃提交。如果把一行删除而不是指定'pick'、'squash'和'edit'中的任何一个，git会从历史中移除该提交

## 五、交互式添加

### 1.交互式添加

交互式添加提供友好的界面去操作Git索引（index），同时亦提供了可视化索引的能力。只需简单键入'git add -i'，即可使用此功能。Git会列出所有修改过的文件及它们的状态。

$ git add -i

在这个例子中，我们可以看到有5个修改过的文件还没有被加入到索引中（unstaged），甚至可以看到每个文件增加和减少的行数。紧接着是一个交互式的菜单，列出了我们可以在此模式中使用的命令。 如果我们想要暂存（stage）这些文件，我们可以键入'2'或者'u'进入更新（update）模式。然后我们可以通过键入文件的范围（本例中是1-4）来决定把哪些文件加入到索引之中。

What now> 2 staged unstaged path

1: unchanged +4/-0 assets/stylesheets/style.css

2: unchanged +23/-11 layout/book\_index\_template.html

3: unchanged +7/-7 layout/chapter\_template.html

4: unchanged +3/-3 script/pdf.rb

5: unchanged +121/-0 text/14\_Interactive\_Rebasing/0\_ Interactive\_Rebasing.markdown

Update>> 1-4 staged unstaged path\* 1: unchanged +4/-0 assets/stylesheets/style.css\* 2: unchanged +23/-11 layout/book\_index\_template.html\* 3: unchanged +7/-7 layout/chapter\_template.html\* 4: unchanged +3/-3 script/pdf.rb

5: unchanged +121/-0 text/14\_Interactive\_Rebasing/0\_ Interactive\_Rebasing.markdown

Update>>

如果键入回车，我会回到主菜单中，同时可以看到那些指定文件的状态已经发生了改变：

What now> status

staged unstaged path

1: +4/-0 nothing assets/stylesheets/style.css

2: +23/-11 nothing layout/book\_index\_template.html

3: +7/-7 nothing layout/chapter\_template.html

4: +3/-3 nothing script/pdf.rb

5: unchanged +121/-0 text/14\_Interactive\_Rebasing/0\_ Interactive\_Rebasing.markdown

现在我们可以看到前4个文件已经被暂存，但是最后一个没有。基本上，这是一个更加紧凑的查看状态的方式，实质上的信息与我们在命令行中运行'git status'是一致的：

$ git status

### 六、储藏

### 1.储藏

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 save "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。

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

当你修复完bug后, 你可以用git stash apply来回复到以前的工作状态。

$ git stash apply

### 2.储藏队列

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

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。

## 七、Git树名

### 1.Git树名

不用40个字节长的SHA串来表示一个提交(commit)或是其它git对象,有很多种名字表示方法。在Git里,这些名字就叫'树名'(treeish)

### 2.Sha短名

如果你的一个提交(commit)的sha名字是 '980e3ccdaac54a0d4de358f3fe5d718027d96aae', git会把下面的串视为等价的:

980e3ccdaac54a0d4de358f3fe5d718027d96aae

980e3ccdaac54a0d4

980e3cc

只要你的‘sha短名’(Partial Sha)是不重复的(unique)，它就不会和其它名字冲突(如果你使用了5个字节以上那是很难重复的)，git也会把‘sha短名’(Partial Sha)自动补全。

### 3.分支, Remote 或 标签

你可以使用分支,remote或标签名来代替SHA串名, 它们只是指向某个对象的指针。假设你的master分支目前在提交(commit):'980e3'上, 现在把它推送(push)到origin上并把它命名为标签'v1.0', 那么下面的串都会被git视为等价的:

980e3ccdaac54a0d4de358f3fe5d718027d96aae

origin/master

refs/remotes/origin/master

master

refs/heads/master

v1.0

refs/tags/v1.0

这意味着你执行下面的两条命令会有同样的输出:

$ git log master

$ git log refs/tags/v1.0

### 4.日期标识符

Git的引用日志(Ref Log)可以让你做一些‘相对'查询操作

master@{yesterday}master@{1 month ago}:

上面的第一条命令是:'master分支的昨天状态(head)的缩写‘。注意: 即使在两个有相同master分支指向的仓库上执行这条命令, 但是如果这个两个仓库在不同机器上,　那么执行结果也很可能会不一样。

### 5.顺序标识符

这种格式用来表达某点前面的第N个提交(ref)。

master@{5}

上面的表达式代表着master前面的第5个提交(ref)。

### 6.多个父对象

这能告诉你某个提交的第N个直接父提交(parent)。这种格式在合并提交(merge commits)时特别有用, 这样就可以使提交对象(commit object)有多于一个直接父对象(direct parent)。

master^2

### 7.波浪号

波浪号用来标识一个提交对象(commit object)的第N级嫡(祖)父对象(Nth grandparent)。 例如:

master~2

就代表master所指向的提交对象的第一个父对象的第一个父对象(译者:你可以理解成是嫡系爷爷:))。 它和下面的这个表达式是等价的:

master^^

你也可以把这些‘标识符'(spec)叠加起来, 下面这个3个表达式都是指向同一个提交(commit):

master^^^^^^

master~3^~2

master~6

### 8.树对象指针

如果大家对第一章Git对象模型还有印象的话, 就记得提交对象(commit object)是指向一个树对象(tree object)的. 假如你要得到一个提交对象(commit object)指向的树对象(tree object)的sha串名, 你就可以在‘树名'的后面加上'{tree}'来得到它:

master^{tree}

### 9.二进制标识符

如果你要某个二进制对象(blob)的sha串名,你可以在'树名'(treeish)后添加二进制对象(blob)对应的文件路径来得到它。

master:/path/to/file

### 10.区间

最后, 你可以用".."来指两个提交(commit)之间的区间. 下面的命令会给出你在"7b593b5" 和"51bea1"之间除了"7b593b5外"的所有提交(commit)(注意:51bea1是最近的提交)。

7b593b5..51bea1

这会包括所有 从 7b593b开始的提交(commit). 译者注: 相当于 7b593b..HEAD

7b593b..

## 八、小结

本节讲解了git的中级知识，在添加索引时可以通过配置.gitignore文件来忽略文件，又讲解了git rebase、git stash和git树名。

中级技能下:

## 一、实验说明

### 1. 环境登录

无需密码自动登录，系统用户名shiyanlou 若不小心登出后，直接刷新页面即可

### 2. 环境使用

实验报告可以在个人主页中查看，其中含有每次实验的截图及笔记，以及每次实验的有效学习时间（指的是在实验桌面内操作的时间，如果没有操作，系统会记录为发呆时间）。这些都是您学习的真实性证明。

### 3.下载测试项目环境

通过下列命令获得gitproject项目环境，该项目默认只有一个文件README.md，可以用来进行后续git实验

$ git clone http://git.shiyanlou.com/shiyanlou/gitproject

## 二、追踪分支

### 1.追踪分支

在Git中‘追踪分支’是用于联系本地分支和远程分支的. 如果你在’追踪分支'(Tracking Branches)上执行推送(push)或拉取(pull)时,它会自动推送(push)或拉取(pull)到关联的远程分支上. 如果你经常要从远程仓库里拉取(pull)分支到本地,并且不想很麻烦的使用"git pull "这种格式; 那么就应当使用‘追踪分支'(Tracking Branches). git clone‘命令会自动在本地建立一个'master'分支，它是'origin/master'的‘追踪分支’. 而'origin/master'就是被克隆(clone)仓库的'master'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).

## 三、使用Git Grep进行搜索

### 1.使用Git Grep进行搜索

用git grep 命令查找Git库里面的某段文字是很方便的. 当然, 你也可以用unix下的'grep'命令进行搜索, 但是'git grep'命令能让你不用签出(checkout)历史文件, 就能查找它们. 例如, 你要看 git.git这个仓库里每个使用'xmmap'函数的地方, 你可以运行下面的命令:

$ git grep xmmap

如果你要显示行号, 你可以添加'-n'选项:

$ git grep -n xmmap

如果我们想只显示文件名, 我们可以使用'--name-onley'选项:

$ git grep --name-only xmmap

我们用'-c'选项可以查看每个文件里有多少行匹配内容(line matches):

$ git grep -c xmmap

现在, 如果我们要查找git仓库里某个特定版本里的内容, 我们可以像下面一样在命令行末尾加上标签名(tag reference):

$ git grep xmmap v1.5.0

我们也可以组合一些搜索条件, 下面的命令就是查找我们在仓库的哪个地方定义了'SORT\_DIRENT'.

$ git grep -e '#define' --and -e SORT\_DIRENT

我不但可以进行“与"(both)条件搜索操作，也可以进行"或"(either)条件搜索操作.

$ git grep --all-match -e '#define' -e SORT\_DIRENT

我们也可以查找出符合一个条件(term)且符合两个条件(terms)之一的文件行.例如我们要找出名字中含有‘PATH'或是'MAX'的常量定义:

$ git grep -e '#define' --and \( -e PATH -e MAX \)

## 四、Git的撤销操作 - 重置, 签出 和 撤销

### 1.修复未提交文件中的错误(重置)

如果你现在的工作目录(work tree)里搞的一团乱麻, 但是你现在还没有把它们提交; 你可以通过下面的命令, 让工作目录回到上次提交时的状态(last committed state):

$ git reset --hard HEAD

这条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. 如果你只是要恢复一个文件,如"hello.rb", 你就要使用 git checkout

$ git checkout -- hello.rb

这条命令把hello.rb从HEAD中签出并且把它恢复成未修改时的样子.

### 2.修复已提交文件中的错误

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. 创建新提交来修复错误 创建一个新的，撤销(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销“上上次”(next-to-last)的提交:

$ git revert HEAD^

在这种情况下,git尝试去撤销老的提交,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销的修改有重叠(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), 把撤销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). 修改提交来修复错误 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). 如果你在老提交(older commit)里发现一个错误, 但是现在还没有发布到代码服务器上. 你可以使用 git rebase命令的交互模式, "git rebase -i"会提示你在编辑中做相关的修改. 这样其实就是让你在rebase的过程来修改提交.

## 五、维护Git

### 1.保证良好的性能

在大的仓库中, git靠压缩历史信息来节约磁盘和内存空间. 压缩操作并不是自动进行的, 你需要手动执行 git gc:

$ git gc

压缩操作比较耗时, 你运行git gc命令最好是在你没有其它工作的时候.

### 2.保持可靠性

git fsck 运行一些仓库的一致性检查, 如果有任何问题就会报告. 这项操作也有点耗时, 通常报的警告就是“悬空对象"(dangling objects).

$ git fsck

“悬空对象"(dangling objects)并不是问题, 最坏的情况只是它们多占了一些磁盘空间. 有时候它们是找回丢失的工作的最后一丝希望.

## 六、建立一个公共仓库

### 1.建立一个公共仓库

假设你个人的仓库在目录 ~/proj. 我们先克隆一个新的“裸仓库“,并且创建一个标志文件告诉git-daemon这是个公共仓库.

$ git clone --bare ~/proj proj.git

$ touch proj.git/git-daemon-export-ok

上面的命令创建了一个proj.git目录, 这个目录里有一个“裸git仓库" -- 即只有'.git'目录里的内容,没有任何签出(checked out)的文件. 下一步就是你把这个 proj.git　目录拷到你打算用来托管公共仓库的主机上. 你可以用scp, rsync或其它任何方式.

### 2.通过git协议导出git仓库

用git协议导出git仓库, 这是推荐的方法. 如果这台服务器上有管理员，TA们要告诉你把仓库放在哪一个目录中, 并且“git:// URL”除仓库目录部分外是什么. 你现在要做的是启动 git daemon; 它会监听在 9418端口. 默认情况下它会允许你访问所有的git目录(看目录中是否有git-daemon-export-ok文件). 如果以某些目录做为 git-daemon 的参数, 那么 git-daemon 会限制用户通过git协议只能访问这些目录. 你可以在inetd service模式下运行 git-daemon; 点击 git daemon　可以查看帮助信息.

### 3.通过http协议导出git仓库

git协议有不错的性能和可靠性, 但是如果主机上已经配好了一台web服务器,使用http协议(git over http)可能会更容易配置一些. 你需要把新建的"裸仓库"放到Web服务器的可访问目录里, 同时做一些调整,以便让web客户端获得它们所需的额外信息.

$ mv proj.git /home/you/public\_html/proj.git

$ cd proj.git

$ git --bare update-server-info

$ chmod a+x hooks/post-update

$ git clone [http://yourserver.com/~you/proj.git](http://yourserver.com/~you/proj.git" \t "/home/jason/文档\\x/_blank)

$ git clone http://yourserver.com/~you/proj.git

## 七、建立一个私有仓库

### 1.通过SSH协议来访问仓库

通常最简单的办法是通ssh协议访问Git(Git Over SSH). 如果你在一台机器上有了一个ssh帐号, 你只要把“git祼仓库"放到任何一个可以通过ssh访问的目录, 然后可以像ssh登录一样简单的使用它. 假设你现在有一个仓库，并且你要把它建成可以在网上可访问的私有仓库. 你可以用下面的命令, 导出一个"祼仓库", 然后用scp命令把它们拷到你的服务器上:

$ git clone --bare /home/user/myrepo/.git /tmp/myrepo.git

$ scp -r /tmp/myrepo.git myserver.com:/opt/git/myrepo.git

如果其它人也在 myserver.com　这台服务器上有ssh帐号，那么TA也可以从这台服务器上克隆(clone)代码:

$ git clone myserver.com:/opt/git/myrepo.git

上面的命令会提示你输入ssh密码或是使用公钥(public key).

## 八、小结

本节讲了追踪分支，使用Git Grep进行搜索，Git的撤消操作（git reset、git checkout、git revert），维护Git（git gc、git fsck）以及建立公有和私有仓库。