高级技能:

## 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、创建新的空分支

### 1.创建新的空分支

在偶尔的情况下，你可能会想要保留那些与你的代码没有共同祖先的分支。例如在这些分支上保留生成的文档或者其他一些东西。如果你需要创建一个不使用当前代码库作为父提交的分支，你可以用如下的方法创建一个空分支：

git symbolic-ref HEAD refs/heads/newbranch

rm .git/index

git clean -fdx

<do work>

git add your files

git commit -m 'Initial commit'

## 三、修改你的历史

### 1.修改你的历史

交互式洐合是修改单个提交的好方法。 git filter-branch是修改大量提交的好方法。

## 四、高级分支与合并

### 1.在合并过程中得到解决冲突的协助

git会把所有可以自动合并的修改加入到索引中去, 所以git diff只会显示有冲突的部分. 它使用了一种不常见的语法:

$ git diff

回忆一下, 在我们解决冲突之后, 得到的提交会有两个而不是一个父提交: 一个父提交会成为HEAD, 也就是当前分支的tip; 另外一个父提交会成为另一分支的tip, 被暂时存在MERGE\_HEAD. 在合并过程中, 索引中保存着每个文件的三个版本. 三个"文件暂存(file stage)"中的每一个都代表了文件的不同版本:

$ git show :1:file.txt

$ git show :2:file.txt

$ git show :3:file.txt

当你使用git diff去显示冲突时, 它在工作树(work tree), 暂存2(stage 2)和暂存3(stage 3)之间执行三路diff操作, 只显示那些两方都有的块(换句话说, 当一个块的合并结果只从暂存2中得到时, 是不会被显示出来的; 对于暂存3来说也是一样). 上面的diff结果显示了file.txt在工作树, 暂存2和暂存3中的差异. git不在每行前面加上单个'+'或者'-', 相反地, 它使用两栏去显示差异: 第一栏用于显示第一个父提交与工作目录文件拷贝的差异, 第二栏用于显示第二个父提交与工作文件拷贝的差异. (参见git diff-files中的"COMBINED DIFF FORMAT"取得此格式详细信息.) 在用直观的方法解决冲突之后(但是在更新索引之前), diff输出会变成下面的样子:

$ git diff

diff --cc file.txt

index 802992c,2b60207..0000000--- a/file.txt+++ b/file.txt

@@@ -1,1 -1,1 +1,1 @@@- Hello world-Goodbye++Goodbye world

上面的输出显示了解决冲突后的版本删除了第一个父版本提供的"Hello world"和第二个父版本提供的"Goodbye", 然后加入了两个父版本中都没有的"Goodbye world". 一些特别diff选项允许你对比工作目录和三个暂存中任何一个的差异:

$ git diff -1 file.txt

$ git diff --base file.txt

$ git diff -2 file.txt

$ git diff --ours file.txt

$ git diff -3 file.txt

$ git diff --theirs file.txt

git log和gitk命令也为合并操作提供了特别的协助:

$ git log --merge

$ gitk --merge

这会显示所有那些只在HEAD或者只在MERGE\_HEAD中存在的提交, 还有那些更新(touch)了未合并文件的提交. 你也可以使用git mergetool, 它允许你使用外部工具如emacs或kdiff3去合并文件. 每次你解决冲突之后, 应该更新索引:

$ git add file.txt

完成索引更新之后, git-diff(缺省地)不再显示那个文件的差异, 所以那个文件的不同暂存版本会被"折叠"起来.

### 2.多路合并

你可以一次合并多个头, 只需简单地把它们作为git merge的参数列出. 例如,

$ git merge scott/master rick/master tom/master

相当于

$ git merge scott/master

$ git merge rick/master

$ git merge tom/master

### 3.子树

有时会出现你想在自己项目中引入其他独立开发项目的内容的情况. 在没有路径冲突的前提下, 你只需要简单地从其他项目拉取内容即可. 如果有冲突的文件, 那么就会出现问题. 可能的例子包括Makefile和其他一些标准文件名. 你可以选择合并这些冲突的文件, 但是更多的情况是你不愿意把它们合并. 一个更好解决方案是把外部项目作为一个子目录进行合并. 这种情况不被递归合并策略所支持, 所以简单的拉取是无用的. 在这种情况下, 你需要的是子树合并策略. 这下面例子中, 我们设定你有一个仓库位于/path/to/B (如果你需要的话, 也可以是一个URL). 你想要合并那个仓库的master分支到你当前仓库的dir-B子目录下. 下面就是你所需要的命令序列:

$ git remote add -f Bproject /path/to/B (1)

$ git merge -s ours --no-commit Bproject/master (2)

$ git read-tree --prefix=dir-B/ -u Bproject/master (3)

$ git commit -m "Merge B project as our subdirectory" (4)

$ git pull -s subtree Bproject master (5)

子树合并的好处就是它并没有给你仓库的用户增加太多的管理负担. 它兼容于较老(版本号小于1.5.2)的客户端, 克隆完成之后马上可以得到代码. 然而, 如果你使用子模块(submodule), 你可以选择不传输这些子模块对象. 这可能在子树合并过程中造成问题. 另外, 若你需要修改内嵌外部项目的内容, 使用子模块方式可以更容易地提交你的修改.

## 查找问题的利器 - Git Bisect

### 4.查找问题的利器 - Git Bisect

假设你在项目的'2.6.18'版上面工作, 但是你当前的代码(master)崩溃(crash)了. 有时解决这种问题的最好办法是: 手工逐步恢复(brute-force regression)项目历史,　找出是哪个提交(commit)导致了这个问题. 但是 linkgit:git-bisect1 可以更好帮你解决这个问题:

$ git bisect start

$ git bisect good v2.6.18

$ git bisect bad master

如果你现在运行"git branch",　会发现你现在所在的是"no branch"(译者注:这是进行git bisect的一种状态). 这时分支指向提交（commit):"69543", 此提交刚好是在"v2.6.18"和“master"中间的位置. 现在在这个分支里,　编译并测试项目代码, 查看它是否崩溃(crash). 假设它这次崩溃了, 那么运行下面的命令:

$ git bisect bad

现在git自动签出(checkout)一个更老的版本. 继续这样做, 用"git bisect good","git bisect bad"告诉git每次签出的版本是否没有问题; 你现在可以注意一下当前的签出的版本, 你会发现git在用"二分查找(binary search)方法"签出"bad"和"good"之间的一个版本(commit or revison). 在这个项目(case)中, 经过13次尝试, 找出了导致问题的提交(guilty commit). 你可以用 git show 命令查看这个提交(commit), 找出是谁做的修改，然后写邮件给TA. 最后, 运行:

$ git bisect reset

这会到你之前(执行git bisect start之前)的状态. 注意: git-bisect 每次所选择签出的版本, 只是一个建议; 如果你有更好的想法, 也可以去试试手工选择一个不同的版本. 运行:

$ git bisect visualize

这会运行gitk, 界面上会标识出"git bisect"命令自动选择的提交(commit). 你可以选择一个相邻的提交(commit), 记住它的SHA串值, 用下面的命令把它签出来:

$ git reset --hard fb47ddb2db...

然后进行测试, 再根据测试结果执行”bisect good"或是"bisect bad"; 就这样反复执行, 直到找出问题为止.

## 五、查找问题的利器 - Git Blame

### 1.查找问题的利器 - Git Blame

如果你要查看文件的每个部分是谁修改的, 那么 git blame 就是不二选择. 只要运行'git blame [filename]', 你就会得到整个文件的每一行的详细修改信息:包括SHA串,日期和作者:

$ git blame sha1\_file.c

如果文件被修改了(reverted),或是编译(build)失败了; 这个命令就可以大展身手了. 你也可以用"-L"参数在命令(blame)中指定开始和结束行:

$ git blame -L 160,+10 sha1\_file.c

## 六、Git和Email

### 1.向一个项目提交补丁

如果你只做了少量的改动, 最简单的提交方法就是把它们做成补丁(patch)用邮件发出去: 首先, 使用git format-patch; 例如:

$ git format-patch origin

这会在当前目录生成一系统编号的补丁文件, 每一个补丁文件都包含了当前分支和origin/HEAD之间的差异内容. 然后你可以手工把这些文件导入你的Email客户端. 但是如果你需要一次发送很多补丁, 你可能会更喜欢使用git send-email脚本去自动完成这个工作. 在发送之前, 应当先到项目的邮件列表上咨询一下项目管理者, 了解他们管理这些补丁的方式.

### 2.向一个项目中导入补丁

Git也提供了一个名为git am的工具(am是"apply mailbox"的缩写)去应用那些通过Email寄来的系列补丁. 你只需要按顺序把所有包含补丁的消息存入单个的mailbox文件, 比如说"patches.mbox", 然后运行

$ git am -3 patches.mbox

Git会按照顺序应用每一个补丁; 如果发生了冲突, git会停下来让你手工解决冲突从而完成合并. ("-3"选项会让git执行合并操作; 如果你更喜欢中止并且不改动你的工作树和索引, 你可以省略"-3"选项.) 在解决冲突和更新索引之后, 你不需要再创建一个新提交, 只需要运行

$ git am --resolved

这时git会为你创建一个提交, 然后继续应用mailbox中余下的补丁. 最后的效果是, git产生了一系列提交, 每个提交是原来mailbox中的一个补丁, 补丁中的作者信息和提交日志也一并被记录下来.

## 七、定制Git

### 1.更改你的编辑器

$ git config --global core.editor vim

### 2.添加别名

$ git config --global alias.last 'cat-file commit HEAD'

$ git last

$ git cat-file commit HEAD

### 3.添加颜色

$ git config color.branch auto

$ git config color.diff auto

$ git config color.interactive auto

$ git config color.status auto

或者你可以通过color.ui选项把颜色全部打开:

$ git config color.ui true

### 4.提交模板

$ git config commit.template '/etc/git-commit-template'

### 5.日志格式

$ git config format.pretty oneline

## 八、Git Hooks

### 1.Git Hooks

钩子(hooks)是一些在"$GIT-DIR/hooks"目录的脚本, 在被特定的事件(certain points)触发后被调用。当"git init"命令被调用后, 一些非常有用的示例钩子文件(hooks)被拷到新仓库的hooks目录中; 但是在默认情况下这些钩子(hooks)是不生效的。 把这些钩子文件(hooks)的".sample"文件名后缀去掉就可以使它们生效了。

### 2.applypatch-msg

GIT\_DIR/hooks/applypatch-msg

当'git-am'命令执行时，这个钩子就被调用。它只有一个参数：就是存有提交消息(commit log message)的文件的名字。如果钩子的执行结果是非零，那么补丁(patch)就不会被应用(apply)。

这个钩子用于在其它地方编辑提交消息，并且可以把这些消息规范成项目的标。它也可以在分析(inspect)完消息文件后拒绝此次提交(commit)。

### 3.pre-applypatch

GIT\_DIR/hooks/pre-applypatch

当'git-am'命令执行时，这个钩子就被调用。它没有参数，并且是在一个补丁(patch)被应用后还未提交(commit)前被调用。如果钩子的执行结果是非零，那么刚才应用的补丁(patch)就不会被提交。

它用于检查当前的工作树，当提交的补丁不能通过特定的测试就拒绝将它提交(commit)进仓库。

### 4.post-applypatch

GIT\_DIR/hooks/post-applypatch

当'git-am'命令执行时，这个钩子就被调用。它没有参数，并且是在一个补丁(patch)被应用且在完成提交(commit)情况下被调用。 这个钩子的主要用途是通知提示(notification)，它并不会影响'git-am'的执行和输出。

### 5.pre-commit

GIT\_DIR/hooks/pre-commit

这个钩子被 'git-commit' 命令调用, 而且可以通过在命令中添加--no-verify 参数来跳过。这个钩子没有参数，在得到提交消息和开始提交(commit)前被调用。如果钩子执行结果是非零，那么 'git-commit' 命令就会中止执行。 当默认的'pre-commit'钩子开启时，如果它发现文件尾部有空白行，那么就会中止此次提交。

### 6.prepare-commit-msg

GIT\_DIR/hooks/prepare-commit-msg

当'git-commit'命令执行时：在编辑器(editor)启动前，默认提交消息准备好后，这个钩子就被调用。 如果钩子的执行结果是非零的话，那么'git-commit'命令就会被中止执行。

### 7.commit-msg

GIT\_DIR/hooks/commit-msg

当'git-commit'命令执行时，这个钩子被调用；也可以在命令中添加--no-verify参数来跳过。这个钩子有一个参数：就是被选定的提交消息文件的名字。如这个钩子的执行结果是非零，那么'git-commit'命令就会中止执行。 这个钩子为提交消息更适当，可以用于规范提交消息使之符合项目的标准(如果有的话)；如果它检查完提交消息后，发现内容不符合某些标准，它也可以拒绝此次提交(commit)。 默认的'commit-msg'钩子启用后，它检查里面是否有重复的签名结束线(Signed-off-by lines)，如果找到它就是中止此次提交(commit)操作。

### 8.post-commit

GIT\_DIR/hooks/post-commit

当'git-commit'命令执行时，这个钩子就被调用。它没有参数，并且是在一个提交(commit)完成时被调用。 这个钩子的主要用途是通知提示(notification)，它并不会影响'git-commit'的执行和输出。

### 9.pre-rebase

GIT\_DIR/hooks/pre-rebase

当'git-base'命令执行时，这个钩子就被调用；主要目的是阻止那些不应被rebase的分支被rebase(例如，一个已经发布的分支提交就不应被rebase)。

### 10.post-checkout

GIT\_DIR/hooks/post-checkout

当'git-checkout'命令更新完整个工作树(worktree)后，这个钩子就会被调用。这个钩子有三个参数：前一个HEAD的 ref，新HEAD的 ref，判断一个签出是分支签出还是文件签出的标识符(分支签出＝1，文件签出＝0)。这个钩子不会影响'git-checkout'命令的输出。 这个钩子可以用于检查仓库的一致性，自动显示签出前后的代码的区别，也可以用于设置目录的元数据属性。

### 11.post-merge

GIT\_DIR/hooks/post-merge

### 12.pre-receive

当用户在本地仓库执行'git-push'命令时，服务器上远端仓库就会对应执行'git-receive-pack'命令，而'git-receive-pack'命令会调用 pre-receive 钩子。在开始更新远程仓库上的ref之前，这个钩子被调用。钩子的执行结果(exit status)决定此次更新能否成功。 每执行一个接收(receive)操作都会调用一次这个钩子。它没有命令行参数，但是它会从标准输入(standard input)读取需要更新的ref，格式如下：

SP SP LF(SP是空格，LF是回车。)

<old-value>是保存在ref里的老对象的名字，<new-value>是保存在ref里的新对象的名字，<ref-name>就是此次要更新的ref的全名。如果是创建一个新的ref，那么<old-value>就是由40个0组成的字符串表示。 如果钩子的执行结果是非零，那么没有引用(ref)会被更新。如果执行结果为零，更新操作还可以被后面的<<update,'update'>>钩子所阻止。 钩子(hook)的标准输出和标准错误输出(stdout & stderr)都会通过'git-send-pack'转发给客户端(other end)，你可以把这个信息回显(echo)给用户。 如果你用ruby,那么可以像下面的代码一样得到它们的参数。

rev\_old, rev\_new, ref = STDIN.read.split(" ")

### 13.update

GIT\_DIR/hooks/update

当用户在本地仓库执行'git-push'命令时，服务器上远端仓库就会对应执行'git-receive-pack'，而'git-receive-pack'会调用 update 钩子。在更新远程仓库上的ref之前，这个钩子被调用。钩子的执行结果(exit status)决定此次update能否成功。

### 14.post-receive

GIT\_DIR/hooks/post-receive

当用户在本地仓库执行'git-push'命令时，服务器上远端仓库就会对应执行'git-receive-pack'命令；在所有远程仓库的引用(ref)都更新后，这个钩子就会被'git-receive-pack'调用。

## 九、找回丢失的对象

### 1.准备

我们先创建一个用以实验的仓库，在里面创建了若干个提交和分支。

$ mkdir recovery

$ cd recovery

$ git init

$ touch file

$ git add file

$ git commit -m "First commit"

$ echo "Hello World" > file

$ git add .

$ git commit -m "Greetings"

$ git branch cool\_branch

$ git checkout cool\_branch

$ echo "What up world?" > cool\_file

$ git add .

$ git commit -m "Now that was cool"

$ git checkout master

$ echo "What does that mean?" >> file

### 2.恢复已删除分支提交

现在repo里有两个branch

$ git branch

cool\_branch\* master

存储当前仓库未提交的改动

$ git stash save "temp save"

删除一个分支

$ git branch -D cool\_branch

$ git branch

\* master

用git fsck --lost-found命令找出刚才删除的分支里面的提交对象。

$ git fsck --lost-found

用git show命令查看一个找到的对象的内容，看是否为我们所找的。

git show 2e43cd56ee4fb08664cd843cd32836b54fbf594a

这个提交对象确实是我们在前面删除的分支的内容；下面我们就要考虑一下要如何来恢复它了。 使用 git rebase　进行恢复

$ git rebase 2e43cd56ee4fb08664cd843cd32836b54fbf594a

现在我们用git log命令看一下，看看它有没有恢复:

$ git log

使用git merge　进行恢复,我们把刚才的恢复的提交删除

$ git reset --hard HEAD^

再把刚删的提交给找回来：

$ git fsck --lost-found

不过这回我们用是合并命令进行恢复：

$ git merge 2e43cd56ee4fb08664cd843cd32836b54fbf594a

### 3.git stash的恢复

前面我们用git stash把没有提交的内容进行了存储，如果这个存储不小心删了怎么办呢？ 当前repo里有的存储：

$ git stash list

把它们清空：

$ git stash clear

再用git fsck --lost-found找回来：

$ git fsck --lost-found

用git show看一下回来的内容对不对：

$ git show 674c0618ca7d0c251902f0953987ff71860cb067

看起来没有问题，好的，那么我就把它恢复了吧：

$ git merge 674c0618ca7d0c251902f0953987ff71860cb067

## 十、子模块

### 1.子模块

一个大项目通常由很多较小的, 自完备的模块组成. 例如, 一个嵌入式Linux发行版的代码树会包含每个进行过本地修改的软件的代码; 一个电影播放器可能需要基于一个知名解码库的特定版本完成编译; 数个独立的程序可能会共用同一个创建脚本. 在集中式版本管理系统中, 可以通过把每个模块放在一个单独的仓库中来完成上述的任务. 开发者可以把所有模块都签出(checkout), 也可以选择只签出他需要的模块. 在移动文件, 修改API和翻译时, 他们甚至可以在一个提交中跨多个模块修改文件 为说明子模块的使用方法, 创建4个用作子模块的示例仓库:

$ mkdir ~/git

$ cd ~/git

$ for i in a b c ddo

mkdir $i

cd $i

git init

echo "module $i" > $i.txt

git add $i.txt

git commit -m "Initial commit, submodule $i"

cd ..done

现在创建父项目, 加入所有的子模块

$ mkdir super

$ cd super

$ git init

$ for i in a b c ddo

git submodule add ~/git/$i $i

done

注意: 如果你想对外发布你的父项目, 请不要使用本地的地址! 列出git-submodule创建文件:

$ ls -a

git-submodule add命令进行了如下的操作:

它在当前目录下克隆各个子模块, 默认签出master分支. 它把子模块的克隆路径加入到gitmodules文件中, 然后把这个文件加入到索引, 准备进行提交. 它把子模块的当前提交ID加入到索引中, 准备进行提交. 提交父项目:

$ git commit -m "Add submodules a, b, c and d."

现在克隆父项目:

$ cd ..

$ git clone super cloned

$ cd cloned

子模块的目录创建好了, 但是它们是空的:

$ ls -a a

$ git submodule status

注意: 上面列出的提交对象的名字会和你的项目中看到的有所不同, 但是它们应该和HEAD的提交对象名字一致. 你可以运行git ls-remote ../git/a进行检验. 拉取子模块需要进行两步操作. 首先运行git submodule init, 把子模块的URL加入到.git/config:

$ git submodule init

现在使用git-submodule update去克隆子模块的仓库和签出父项目中指定的那个版本:

$ git submodule update

$ cd a

$ ls -a

git-submodule update和git-submodule add的一个主要区别就是git-submodule update签出一个指定的提交, 而不是该分支的tip. 它就像签出一个标签(tag): 头指针脱离, 你不在任何一个分支上工作.

$ git branch

如何你需要对子模块进行修改, 同时头指针又是脱离的状态, 那么你应该创建或者签出一个分支, 进行修改, 发布子模块的修改, 然后更新父项目让其引用新的提交:

$ git checkout master

或者

$ git checkout -b fix-up

然后

$ echo "adding a line again" >> a.txt

$ git commit -a -m "Updated the submodule from within the superproject."

$ git push

$ cd ..

$ git diff

$ git add a

$ git commit -m "Updated submodule a."

$ git push

如果你想要更新子模块, 你应该在git pull之后运行git submodule update.

### 2.子模块方式的陷阱

你应该总是在发布父项目的修改之前发布子模块修改. 如果你忘记发布子模块的修改, 其他人就无法克隆你的仓库了:

$ cd ~/git/super/a

$ echo i added another line to this file >> a.txt

$ git commit -a -m "doing it wrong this time"

$ cd ..

$ git add a

$ git commit -m "Updated submodule a again."

$ git push

$ cd ~/git/cloned

$ git pull

$ git submodule update

如果你暂存了一个更新过的子模块, 准备进行手工提交, 注意不要在路径后面加上斜杠. 如果加上了斜杠, git会认为你想要移除那个子模块然后签出那个目录内容到父仓库.

$ cd ~/git/super/a

$ echo i added another line to this file >> a.txt

$ git commit -a -m "doing it wrong this time"

$ cd ..

$ git add a/

$ git status

为了修正这个错误的操作, 我们应该重置(reset)这个修改, 然后在add的时候不要加上末尾斜杠.

$ git reset HEAD A

$ git add a

$ git status

## 十一、小结

本节讲解了创建新的空分支，修改历史，高级分支与合并，查找问题的利器Git Bisect和Git Blame，Git和Email，定制Git，Git Hooks，找回丢失的对象以及子模块。