## 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. 课程来源

感谢译者@liuhui998授权，本课程为[《Git Community Book 中文版》](http://gitbook.liuhui998.com/" \t "/home/jason/文档\\x/_blank)提供配套实验。

### 4.课程学习说明

学习本课程需要有一定的Linux基础，熟悉Linux的常用命令。

**建议初学者只需要学习前三个实验即可，基本用法已经涵盖了最常用的git操作。中级与高级用法使用很少，可以简单了解，用到时再查详细文档。**

## 二、git诞生

同生活中的许多伟大事件一样，Git 诞生于一个极富纷争大举创新的年代。1991年，Linus创建了开源的Linux，并且有着为数众多的参与者。虽然有世界各地的志愿者为Linux编写代码，但是绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991－2002年间）。在这期间，所有的源代码都是由Linus手工合并。因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统（集中式和分布式我们会在接下来的内容讲解）不但速度慢，而且必须联网才能使用。虽然有一些商用的版本控制系统，比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002 年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是整个项目组启用了一个商业版本的分布式版本控制系统 BitKeeper 来管理和维护代码。BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。安定团结的大好局面在2005年被打破，开发BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，原因是Linux社区牛人聚集，开发Samba的Andrew试图破解BitKeeper的协议，这么干的其实也不只他一个，但是被BitMover公司发现了，于是BitMover公司收回了Linux社区的免费使用权。这就迫使Linux开源社区（ 特别是Linux的缔造者 Linus Torvalds ）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。

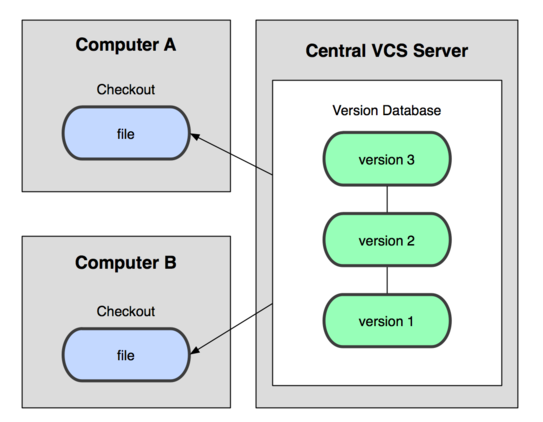
他们对新的系统制订了若干目标：速度 、 简单的设计 、 对非线性开发模式的强力支持（允许上千个并行开发的分支）、完全分布式、有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）。自诞生于 2005 年以来，Git 日臻成熟完善，迅速成为最流行的分布式版本控制系统，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，它还有着令人难以置信的非线性分支管理系统，可以应付各种复杂的项目开发需求。2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

## 三、版本控制系统

Linus一直痛恨的CVS及SVN都是集中式的版本控制系统，而Git是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

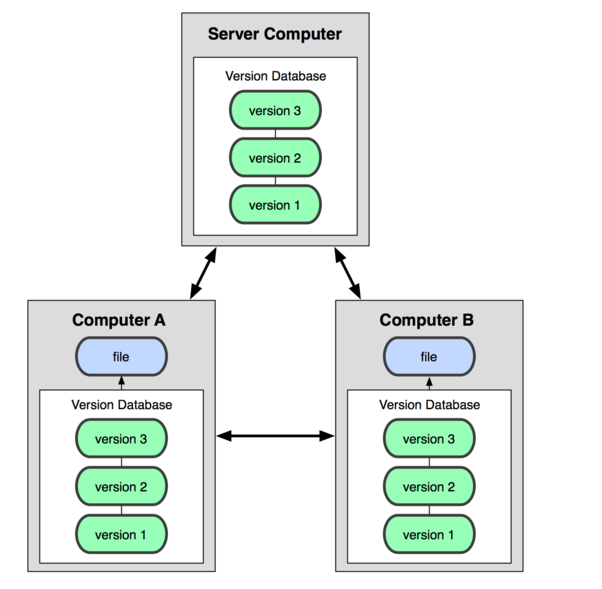
先说集中式版本控制系统，版本库是集中存放在中央服务器的，而大家工作的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始工作，工作完成，再把自己的修订推送给中央服务器。这类系统，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。



那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。



许多这类系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

基本用法一:

## 一、实验说明

本节实验为 Git 入门第一个实验，可以帮助大家熟悉如何创建和使用 git 仓库。

## 二、git的初始化

在使用git进行代码管理之前，我们首先要对git进行初始化。

### 1.Git 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，每次提交记录里都会包含这些信息。使用git config命令进行配置：

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

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

执行了上面的命令后,会在家目录(/home/shiyanlou)下建立一个叫.gitconfig 的文件（该文件为隐藏文件，需要使用ls -al查看到）. 内容一般像下面这样，可以使用vim或cat查看文件内容:

$ cat ~/.gitconfig

[user]

email = schacon@gmail.com

name = Scott Chacon

上面的配置文件就是Git全局配置的文件，一般配置方法是git config --global <配置名称> <配置的值>。

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. 这会在你当前的项目目录下创建 .git/config，从而使用针对当前项目的配置。

## 三、获得一个Git仓库

既然我们现在把一切都设置好了，那么我们需要一个Git仓库。有两种方法可以得到它：一种是从已有的Git仓库中clone (克隆，复制)；还有一种是新建一个仓库，把未进行版本控制的文件进行版本控制。

### 1.Clone一个仓库

为了得一个项目的拷贝(copy),我们需要知道这个项目仓库的地址(Git URL). Git能在许多协议下使用，所以Git URL可能以ssh://, http(s)://, git://. 有些仓库可以通过不只一种协议来访问。

我们在git.shiyanlou.com上提供了一个名字为gitproject的供大家测试的公有仓库，这个仓库可以使用下面方式进行clone：

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

clone操作完成后，会发现当前目录下多了一个gitproject文件夹，这个文件夹里的内容就是我们刚刚clone下来的代码。由于当前`gitproject仅是测试项目，里面仅有一个README.md文件。

$ cd gitproject/

(master)$ ls

README.md

细心的同学可以发现在命令提示符$前面多了个(master)。这是由于实验楼的Linux使用的是zsh Shell，zsh会判断当前的目录是否有Git仓库，如果是的话则自动把目前所在的git分支显示在提示符中。Git 分支的概念我们会在稍后介绍。

### 2.初始化一个新的仓库

可以对一个已存在的文件夹用下面的命令让它置于Git的版本控制管理之下。

创建代码目录project：

$ cd /home/shiyanlou/

$ mkdir project

进入到代码目录，创建并初始化Git仓库：

$ cd project

$ git init

Git会输出:

Initialized empty Git repository in /home/shiyanlou/project/.git/

通过ls -la命令会发现project目录下会有一个名叫.git 的目录被创建，这意味着一个仓库被初始化了。可以进入到.git目录查看下有哪些内容。

## 四、正常的工作流程

### 1. 正常的工作流程

git的基本流程如下：

1. 创建或修改文件
2. 使用git add命令添加新创建或修改的文件到本地的缓存区（Index）
3. 使用git commit命令提交到本地代码库
4. （可选，有的时候并没有可以同步的远端代码库）使用git push命令将本地代码库同步到远端代码库

进入我们刚才建立的project目录，分别创建文件file1，file2，file3：

$ cd project

$ touch file1 file2 file3

修改文件，可以使用vim编辑内容，也可以直接echo添加测试内容。

$ echo "test" >> file1

$ echo "test" >> file2

$ echo "test" >> file3

此时可以使用git status命令查看当前git仓库的状态：

$ git status

On branch master

Initial commit

Untracked files:

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

file1

file2

file3nothing added to commit but untracked files present (use "git add" to track)

可以发现，有三个文件处于untracked状态，下一步我们就需要用git add命令将他们加入到缓存区（Index）。

使用git add命令将新建的文件添加到：

$ git add file1 file2 file3

然后再次执行git status就会发现新的变化：

$ git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: file1

new file: file2

new file: file3

你现在为commit做好了准备，你可以使用 git diff 命令再加上 --cached 参数，看看缓存区中哪些文件被修改了。进入到git diff --cached界面后需要输入q才可以退出：

$ git diff --cached

如果没有--cached参数，git diff 会显示当前你所有已做的但没有加入到索引里的修改。

如果你要做进一步的修改, 那就继续做, 做完后就把新修改的文件加入到缓存区中。

当所有新建，修改的文件都被添加到了缓存区，我们就要使用git commit提交到本地仓库：

$ git commit -m "add 3 files"

需要使用-m添加本次修改的注释，完成后就会记录一个新的项目版本。除了用git add 命令，我们还可以用下面的命令将所有没有加到缓存区的修改也一起提交，但-a命令不会添加新建的文件。

$ git commit -a -m "add 3 files"

再次输入git status查看状态，会发现当前的代码库已经没有待提交的文件了，缓存区已经被清空。

至此，我们完成了第一次代码提交，这次提交的代码中我们创建了三个新文件。需要注意的是如果是修改文件，也需要使用git add命令添加到缓存区才可以提交。如果是删除文件，则直接使用git rm命令删除后会自动将已删除文件的信息添加到缓存区，git commit提交后就会将本地仓库中的对应文件删除。

这个时候如果本地的仓库连接到了远程Git服务器，可以使用下面的命令将本地仓库同步到远端服务器：

$ git push origin master

这时候可能需要你输入在Git服务器上的用户名和密码。可以参考[实验楼的代码库](https://www.shiyanlou.com/questions/360" \t "/home/jason/文档\\x/_blank)中的说明练习这个命令。

## 五、分支与合并

Git的分支可以让你在主线（master分支）之外进行代码提交，同时又不会影响代码库主线。分支的作用体现在多人协作开发中，比如一个团队开发软件，你负责独立的一个功能需要一个月的时间来完成，你就可以创建一个分支，只把该功能的代码提交到这个分支，而其他同事仍然可以继续使用主线开发，你每天的提交不会对他们造成任何影响。当你完成功能后，测试通过再把你的功能分支合并到主线。

### 1.分支

一个Git仓库可以维护很多开发分支。现在我们来创建一个新的叫 experimental的分支：

$ git branch experimental

运行git branch命令可以查看当前的分支列表，已经目前的开发环境处在哪个分支上：

$ git branch

experimental\* master

experimental 分支是你刚才创建的，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，输入git checkout 分支名可以切换到其他分支：

$ git checkout experimental

Switched to branch 'experimental'

切换到experimental分支，切换完成后，先编辑里面的一个文件，再提交(commit)改动，最后切换回 “master”分支：

# 修改文件file1

$ echo "update" >> file1# 查看当前状态

$ git status# 添加并提交file1的修改

$ git add file1

$ git commit -m "update file1"# 查看file1的内容

$ cat file1

test

update# 切换到master分支

$ git checkout master

查看下file1中的内容会发现刚才做的修改已经看不到了。因为刚才的修改时在experimental分支下，现在切换回了master分支，目录下的文件都是master分支上的文件了。

现在可以在master分支下再作一些不同的修改:

# 修改文件file2

$ echo "update again" >> file2# 查看当前状态

$ git status# 添加并提交file2的修改

$ git add file2

$ git commit -m "update file2 on master"# 查看file2的内容

$ cat file2

test

update again

这时，两个分支就有了各自不同的修改，分支的内容都已经不同，如何将多个分支进行合并呢？

可以通过下面的git merge命令来合并experimental到主线分支master:

# 切换到master分支

$ git checkout master# 将experimental分支合并到master

$ git merge -m 'merge experimental branch' experimental

-m参数仍然是需要填写合并的注释信息。

由于两个branch修改了两个不同的文件，所以合并时不会有冲突，执行上面的命令后合并就完成了。

如果有冲突，比如两个分支都改了一个文件file3，则合并时会失败。首先我们在master分支上修改file3文件并提交：

# 切换到master分支

$ git checkout master# 修改file3文件

$ echo "master: update file3" >> file3# 提交到master分支

$ git commit -a -m 'update file3 on master'

然后切换到experimental，修改file3并提交：

# 切换到experimental分支

$ git checkout experimental# 修改file3文件

$ echo "experimental: update file3" >> file3# 提交到master分支

$ git commit -a -m 'update file3 on experimental'

切换到master进行合并：

$ git checkout master

$ git merge experimentalAuto-merging file3

CONFLICT (content): Merge conflict in file3Automatic merge failed; fix conflicts and then commit the result.

合并失败后先用git status查看状态，会发现file3显示为both modified，查看file3内容会发现：

$ cat file3

test

<<<<<<< HEADmaster: update file3

=======experimental: update file3>>>>>>> experimental

上面的内容也可以使用git diff查看，先前已经提到git diff不加参数可以显示未提交到缓存区中的修改内容。

可以看到冲突的内容都被添加到了file3中，我们使用vim编辑这个文件，去掉git自动产生标志冲突的<<<<<<等符号后，根据需要只保留我们需要的内容后保存，然后使用git add file3和git commit命令来提交合并后的file3内容，这个过程是手动解决冲突的流程。

# 编辑冲突文件

$ vim file3# 提交修改后的文件

$ git add file3

$ git commit -m 'merge file3'

当我们完成合并后，不再需要experimental时，可以使用下面的命令删除：

$ git branch -d experimental

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

### 2.撒销一个合并

如果你觉得你合并后的状态是一团乱麻，想把当前的修改都放弃，你可以用下面的命令回到合并之前的状态：

$ git reset --hard HEAD^# 查看file3的内容，已经恢复到合并前的master上的文件内容

$ cat file3

### 3.快速向前合并

还有一种需要特殊对待的情况，在前面没有提到。通常，一个合并会产生一个合并提交(commit), 把两个父分支里的每一行内容都合并进来。

但是，如果当前的分支和另一个分支没有内容上的差异，就是说当前分支的每一个提交(commit)都已经存在另一个分支里了，git 就会执行一个“快速向前"(fast forward)操作；git 不创建任何新的提交(commit),只是将当前分支指向合并进来的分支。

## 六、Git日志

### 1.查看日志

git log命令可以显示所有的提交(commit)：

$ git log

如果提交的历史纪录很长，回车会逐步显示，输入q可以退出。

git log有很多选项，可以使用git help log查看，例如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/

Git会根据git log命令的参数，按时间顺序显示相关的提交(commit)。

### 2.日志统计

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，这个命令相当于打印详细的提交记录：

$ git log --stat

### 3.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:

$ git log --pretty=oneline

或者你也可以使用 short 格式:

$ git log --pretty=short

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参数定义格式。

--graph 选项可以可视化你的提交图(commit graph)，会用ASCII字符来画出一个很漂亮的提交历史(commit history)线：

$ git log --graph --pretty=oneline

### 4.日志排序

日志记录可以按不同的顺序来显示。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。

按默认情况，提交会按逆时间顺序显示，可以指定--topo-order参数，让提交按拓扑顺序来显示(就是子提交在它们的父提交前显示):

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

你也可以用 --reverse参数来逆向显示所有提交日志。

基本用法二:

## 一、实验说明

本节实验为 Git 入门第二个实验，继续练习最常用的git命令。

### 1.1 实验准备

在进行该实验之前，可以先clone一个练习项目gitproject:

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

本节中的实验操作都是在该项目中完成。

## 二、比较内容

### 1.比较提交 - Git Diff

现在我们对项目做些修改：

$ cd gitproject# 向README文件添加一行

$ echo "new line" >> README.md# 添加新的文件file1

$ echo "new file" >> file1

使用git status查看当前修改的状态：

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

modified: README.md

Untracked files:

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

file1

no changes added to commit (use "git add" and/or "git commit -a")

可以看到一个文件修改了，另外一个文件添加了。如何查看修改的文件内容呢，那就需要使用git diff命令。git diff命令的作用是比较修改的或提交的文件内容。

$ git diff

diff --git a/README.md b/README.md

index 21781dd..410e719 100644--- a/README.md+++ b/README.md@@ -1,2 +1,3 @@

gitproject

==========+new line

上面的命令执行后需要使用q退出。命令输出当前工作目录中修改的内容，并不包含新加文件，请注意这些内容还没有添加到本地缓存区。

将修改内容添加到本地缓存区，通配符可以把当前目录下所有修改的新增的文件都自动添加：

$ git add \*

再执行git diff会发现没有任何内容输出，说明当前目录的修改都被添加到了缓存区，如何查看缓存区内与上次提交之间的差别呢？需要使用--cached参数：

$ git diff --cached

diff --git a/README.md b/README.md

index 21781dd..410e719 100644--- a/README.md+++ b/README.md@@ -1,2 +1,3 @@

gitproject

==========+new line

diff --git a/file1 b/file1

new file mode 100644

index 0000000..fa49b07--- /dev/null+++ b/file1

@@ -0,0 +1 @@+new file

可以看到输出中已经包含了新加文件的内容，因为file1已经添加到了缓存区。

最后我们提交代码：

$ git commit -m 'update code'

提交后git diff与git diff --cached都不会有任何输出了。

### 2.比较分支

可以用 git diff 来比较项目中任意两个分支的差异。

我们首先创建一个新的分支test，并在该分支上提交一些修改：

# 创建test分支并切换到该分支

$ git branch test

$ git checkout test# 添加新的一行到file1

$ echo "branch test" >> file1# 创建新的文件file2

$ echo "new file2" >> file2# 提交所有修改

$ git add \*

$ git commit -m 'update test branch'

然后，我们查看test分支和master之间的差别：

$ git diff master test

diff --git a/file1 b/file1

index fa49b07..17059cd 100644--- a/file1

+++ b/file1

@@ -1 +1,2 @@

new file

+branch test

diff --git a/file2 b/file2

new file mode 100644

index 0000000..80e7991--- /dev/null

+++ b/file2

@@ -0,0 +1 @@

+new file2

git diff 是一个难以置信的有用的工具，可以找出你项目上任意两个提交点间的差异。可以使用git help diff详细查看其他参数和功能。

### 3.更多的比较选项

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:

# 切换到master

$ git checkout master

# 查看与test分支的区别

$ git diff test

diff --git a/file1 b/file1

index 17059cd..fa49b07 100644--- a/file1+++ b/file1

@@ -1,2 +1 @@

new file-branch test

diff --git a/file2 b/file2

deleted file mode 100644

index 80e7991..0000000--- a/file2+++ /dev/null

@@ -1 +0,0 @@-new file2

你也以加上路径限定符，来只比较某一个文件或目录：

$ git diff test file1

diff --git a/file1 b/file1

index 17059cd..fa49b07 100644--- a/file1

+++ b/file1

@@ -1,2 +1 @@

new file

-branch test

上面这条命令会显示你当前工作目录下的file1与test分支之间的差别。

--stat 参数可以统计一下有哪些文件被改动，有多少行被改动：

$ git diff test --stat

file1 | 1 -

file2 | 1 -

2 files changed, 2 deletions(-)

## 三、分布式的工作流程

### 1.分布式的工作流程

你目前的项目在/home/shiyanlou/gitproject目录下，这是我们的git 仓库(repository)，另一个用户也想与你协作开发。他的工作目录在这台机器上，如何让他提交代码到你的git仓库呢？

首先，我们假设另一个用户也用shiyanlou用户登录，只是工作在不同的目录下开发代码，实际工作中不太可能发生，大部分情况都是多个用户，这个假设只是为了让实验简化。

该用户需要从git仓库进行克隆：

# 进入到临时目录

$ cd /tmp# 克隆git仓库

$ git clone /home/shiyanlou/gitproject myrepo

$ ls -l myrepo

-rw-rw-r-- 1 shiyanlou shiyanlou 31 Dec 22 08:24 README.md

-rw-rw-r-- 1 shiyanlou shiyanlou 9 Dec 22 08:24 file1

这就建了一个新的叫"myrepo"的目录，这个目录里包含了一份gitproject仓库的克隆。这份克隆和原始的项目一模一样，并且拥有原始项目的历史记录。

在myrepo做了一些修改并且提交:

$ cd myrepo

# 添加新的文件newfile

$ echo "newcontent" > newfile

# 提交修改

$ git add newfile

$ git commit -m "add newfile"

myrepo修改完成后，如果我们想合并这份修改到gitproject的git仓库该如何做呢？

可以在仓库/home/shiyanlou/gitproject中把myrepo的修改给拉 (pull)下来。执行下面几条命令:

$ cd /home/shiyanlou/gitproject

$ git pull /tmp/myrepo master

remote: Counting objects: 5, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 0), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From /tmp/myrepo

\* branch master -> FETCH\_HEAD

Updating 8bb57aa..866c452

Fast-forward

newfile | 1 +

1 file changed, 1 insertion(+)

create mode 100644 newfile

# 查看当前目录文件

$ ls [8:28:02]

README.md file1 newfile

这就把myrepo的主分支合并到了gitproject的当前分支里了。

如果gitproject在myrepo修改文件内容的同时也做了修改的话，可能需要手工去修复冲突。

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 myrepo /tmp/myrepo

git pull命令执行两个操作: 它从远程分支(remote branch)抓取修改git fetch的内容，然后把它合并git merge进当前的分支。

gitproject里可以用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 myrepo

From /tmp/myrepo

\* [new branch] master -> myrepo/master

获取后，我们可以通过git log查看远程分支做的所有修改，由于我们已经合并了所有修改，所以不会有任何输出：

$ git log -p master..myrepo/master

当检查完修改后，gitproject可以把修改合并到它的主分支中：

$ git merge myrepo/master

Already up-to-date.

如果我们在myrepo目录下执行git pull会发生什么呢？

myrepo会从克隆的位置拉取代码并更新本地仓库，就是把gitproject上的修改同步到本地:

# 进入到gitproject

$ cd /home/shiyanlou/gitproject

# 添加一行内容到newfile

$ echo "gitproject: new line" >> newfile

# 提交修改

$ git commit -a -m 'add newline to newfile'

[master 8c31532] add newline to newfile

1 file changed, 1 insertion(+)

# 进入myrepo目录

$ cd /tmp/myrepo

# 同步gitproject的所有修改

$ git pullremote: Counting objects: 6, done.remote: Compressing objects: 100% (2/2), done.remote: Total 3 (delta 1), reused 0 (delta 0)

Unpacking objects: 100% (3/3), done.

From /home/shiyanlou/gitproject

8bb57aa..8c31532 master -> origin/master

Updating 866c452..8c31532

Fast-forward

newfile | 1 +

1 file changed, 1 insertion(+)

因为myrepo是从gitproject仓库克隆的，那么他就不需要指定gitproject仓库的地 址。因为Git把gitproject仓库的地址存储到myrepo的配置文件中，这个地址就是在git pull时默认使用的远程仓库：

$ git config --get remote.origin.url

/home/shiyanlou/gitproject

如果myrepo和gitproject在不同的主机上，可以通过ssh协议来执行clone 和pull操作：

$ git clone localhost:/home/shiyanlou/gitproject test

这个命令会提示你输入shiyanlou用户的密码，用户密码随机，可以点击屏幕上方的SSH按钮查看。

### 2.公共Git仓库

开发过程中，通常大家都会使用一个公共的仓库，并clone到自己的开发环境中，完成一个阶段的代码后可以告诉目标仓库的维护者来pull自己的代码。

如果你和维护者都在同一台机器上有帐号，那么你们可以互相从对 方的仓库目录里直接拉所作的修改，git命令里的仓库地址也可以是本地的某个目录名：

$ git clone /path/to/repository

$ git pull /path/to/other/repository

也可以是一个ssh地址：

$ git clone ssh://yourhost/~you/repository

### 3.将修改推到一个公共仓库

通过http或是git协议，其它维护者可以通过远程访问的方式抓取(fetch)你最近的修改，但是他们 没有写权限。如何将本地私有仓库的最近修改主动上传到公共仓库中呢？

最简单的办法就是用git push命令，推送本地的修改到远程Git仓库，执行下面的命令:

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

或者

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

git push命令的目地仓库可以是ssh或http/https协议访问。

### 4.当推送代码失败时要怎么办

如果推送(push)结果不是快速向前fast forward，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'

这种情况通常是因为没有使用git pull获取远端仓库的最新更新，在本地修改的同时，远端仓库已经变化了（其他协作者提交了代码），此时应该先使用git pull合并最新的修改后再执行git push：

$ git pull

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

## 四、Git标签

### 1.轻量级标签

我们可以用 git tag不带任何参数创建一个标签(tag)指定某个提交(commit):

# 进入到gitproject目录

$ cd /home/shiyanlou/gitproject

# 查看git提交记录

$ git log

# 选择其中一个记录标志位stable-1的标签，注意需要将后面的8c315325替换成仓库下的真实提交内，commit的名称很长，通常我们只需要写前面8位即可

$ git tag stable-1 8c315325

# 查看当前所有tag

$ git tag

stable-1

这样，我们可以用stable-1 作为提交 8c315325 的代称。

前面这样创建的是一个“轻量级标签”。

如果你想为一个tag添加注释，或是为它添加一个签名, 那么我们就需要创建一个 "标签对象"。

#### 标签对象

git tag中使用-a， -s 或是 -u三个参数中任意一个，都会创建一个标签对象，并且需要一个标签消息(tag message)来为tag添加注释。 如果没有-m 或是 -F 这些参数，命令执行时会启动一个编辑器来让用户输入标签消息。

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，这就是与轻量级标签的区别。

下面是一个创建标签对象的例子:

$ git tag -a stable-2 8c315325 -m "stable 2"

$ git tag

stable-1

stable-2

### 2.签名的标签

签名标签可以让提交和标签更加完整可信。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>

现在你可以在创建标签的时候使用-s 参数来创建“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

## 五、小结

本节学习了下面知识点：

* git diff
* 分布式的工作流程
* git tag

对于初学者，如果不想深入git强大的高级功能的话，学完这个实验就可以了，因为后续实验内容用到的比较少，并且理解难度大。如果仍然感兴趣，建议使用一段时间git后再仔细学习后续实验，会有更好的收获。