# Git简介

Git是什么？

Git是开源的分布式版本控制系统。

什么是版本控制系统？

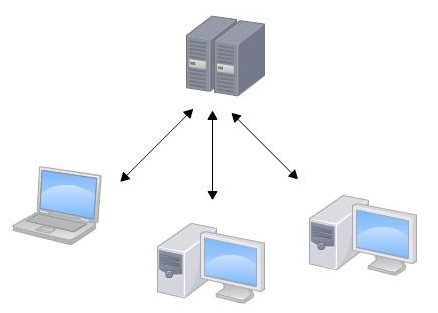
版本控制系统记录了文件的增删查改，用于管理项目或工程或文件，方便恢复至指定的版本。

## Git的诞生

2005年Linus开始Git项目，用于管理Linux代码。2008年GitHub网站上线，它为开源项目免费提供Git存储，一些著名的项目如jQuery，PHP，Ruby等等迁移到GitHub。

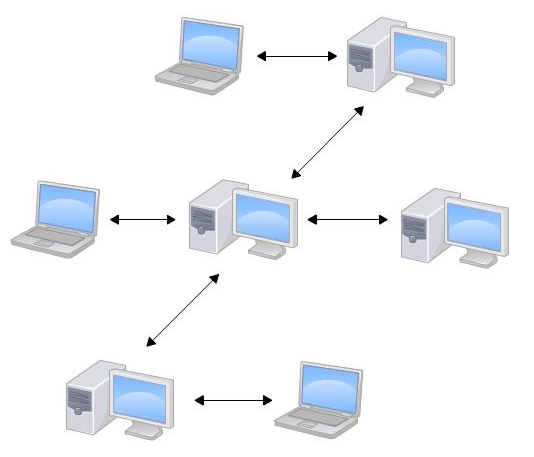
## 集中式vs分布式

SVN是著名的集中式版本控制系统，其拓扑结构如下：



版本库存储在中央服务器上。

Git是分布式版本控制系统，其拓扑结构如下：



每一台主将都有一个完整的版本库。

# 安装Git

Windows平台在<http://git-scm.com/>下载对应的版本安装。

Linux平台可以使用sudo apt-get install git命令安装。

安装完后，需要配置，

$ git config --global user.name "Your Name"

$ git config --global user.email "email@example.com"

Git config命令的—global参数，表示这台机器上所有的Git仓库都会使用这个配置，但可以对某个仓库指定不同的用户名和邮件地址。

# 创建版本库

版本库，又名仓库，英文名repository，是一个目录，里面所有的文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便在未来某个时刻还原。

创建版本库只需两步，第一步，选择一个合适的地方，创建一个空目录：

$ mkdir learngit

$ cd learngit

$ pwd

/Users/michael/learngit

pwd用于显示当前目录，第二步，

$ git init

Initialized empty Git repository **in** /Users/michael/learngit/.git/

创建了一个空仓库，多了一个.git目录，用来跟踪管理版本库的，不需手动修改。

**把文件添加到版本库**

所有的版本控制系统，只能跟踪文本文件的改动，如txt文件，网页，程序代码等，而图片、视频这些二进制文件，只能知道其大小的改动，无法知道其文件内容的改动。文本推荐使用UTF-8编码。

编写一个readme.txt文件，内容如下：

Git is a version control system.

Git is free software.

被Git管理的文件要放置仓库或其子目录下。

把文件添加到仓库，第一步，用命令git add告诉Git，把文件添加到仓库：

$ git add readme.txt

执行上面命令没有任何显示就对了，Unix的哲学是“没有消息就是好消息”，说明添加成功。

第二步，用命令git commit告诉Git，把文件提交到仓库：

$ git **commit** -m "wrote a readme file"

[master (root-**commit**) cb926e7] wrote a readme file

1 file changed, 2 insertions(+)

**create** mode 100644 readme.txt

参数-m后面输入的是本次提交的说明，git commit命令执行成功后，1个文件被改动，插入了两行内容（readme.txt有两行内容）。

可以添加多个文件，一次性提交，如：

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files."

# 时光机穿梭

修改readme.txt文件，改成如下内容：

Git is a distributed version control system.

Git is a free software.

增加了distributed，运行git status命令，查看仓库的当前状态：

$ git status

*# On branch 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.txt*

*#*

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

查看difference使用git diff命令：

$ git diff readme.txt

diff --git a/readme.txt b/readme.txt

index 46d49bf..9247db6 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,2 +1,2 @@

-Git **is** a version control system.

+Git **is** a distributed version control system.

Git **is** free software.

使用git add命令和git commit命令添加并提交修改后的readme.txt文件：

$ git add readme.txt

$ git **commit** -m "add distributed"

[master ea34578] **add** distributed

1 file changed, 1 insertion(+), 1 deletion(-)

提交后，再用git status命令查看仓库的当前状态：

$ git status

*# On branch master*

nothing to commit (working directory clean)

Git告诉我们当前没有需要提交的修改，而且工作目录是干净的(working directory clean)。

## 版本退回

再次修改readme.txt文件如下：

Git is a distributed version control system.

Git is free software distribute under the GPL.

然后尝试提交：

$ git add readme.txt

$ git commit -m "append GPL"

[master 3628164] append GPL

1 file changed, 1 insertion(+), 1 deletion(-)

使用git log命令可以查看历史记录：

$ git log

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

git log命令从最近到最远的日志，可以看到提交了3次，最近一次是append GPL，上一次是add distributed，最早一次是wrote a readme file。试试加上—pretty=oneline参数：

$ git log --pretty=oneline

3628164fb26d48395383f8f31179f24e0882e1e0 append GPL

ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed

cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file

3628164…882ele0这一串是commit id（版本号），由SHA1计算出来，用十六进制表示。

在Git中，HEAD表示当前版本，HEAD^表示上一个版本，HEAD^^表示上上一个版本，HEAD~100表示往上100个版本。

如果我们要把当前版本”append GPL”回退到上一个版本”add distributed”，可以使用git reset命令：

$ git re**set** --hard HEAD^

HEAD **is** now **at** ea34578 **add** distributed

再使用git log查看现在版本库的状态：

$ git log

commit ea34578d5496d7dd233c827ed32a8cd576c5ee85

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 14:53:12 2013 +0800

add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030

Author: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

git reflog命令用来记录操作的每一次命令：

$ git reflog

ea34578 HEAD@{0}: reset: moving to HEAD^

3628164 HEAD@{1}: commit: append GPL

ea34578 HEAD@{2}: commit: add distributed

cb926e7 HEAD@{3}: commit (initial): wrote a readme file

可以根据commit id 回到未来版本：

$ git re**set** --hard 3628164

HEAD **is** now **at** 3628164 append GPL

版本号可以不用写全，前几位就可以了，只要无异议，Git能自动找到。

Git的版本回退速度非常快，因为Git内部有个指向当前版本的HEAD指针，当退回版本时，Git仅仅是把HEAD从指向append GPL改为指向add distributed



然后顺便把工作区的文件更新了。

## 工作区和暂存区

**工作区(Working Directory)**

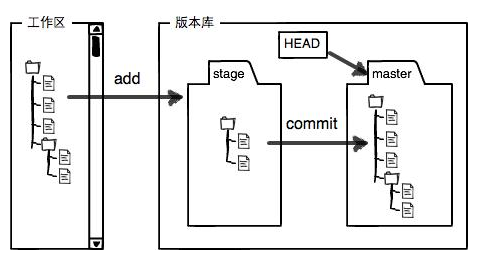
就是在电脑里能看到的目录，如learngit就是一个工作区。

**版本库(Repository)**

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存放了很多东西，其中最重要的就是称为stage(或者叫index)的暂存区，还有Git为我们创建的一个分支master，以及指向master的一个指针叫HEAD。

git add把文件添加到暂存区；git commit把暂存区的所有内容提交到当前分支，默认情况下往master分支上提交更改。过程如下：



先对readme.txt文件做修改，加上一行内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

然后，在工作区新增一个LICENSE文本文件（内容随便写）。先用git status查看一下状态：

$ git status

*# On branch 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.txt*

*#*

*# Untracked files:*

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

*# LICENSE*

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

Git告诉我们，readme.txt被修改了，而LICENSE还没被添加过，它的状态是Untracked。使用git add命令把readme.txt和LICENSE都添加后，再查看状态：

$ git status

*# On branch master*

*# Changes to be committed:*

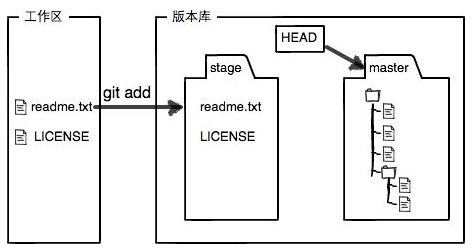
*# (use "git reset HEAD <file>..." to unstage)*

*#*

*# new file: LICENSE*

*# modified: readme.txt*

现在，暂存区的状态就变成这样了：



然后，再执行git commit命令一次性把暂存区的所有修改提交到分支：

$ git **commit** -m "understand how stage works"

[master 27c9860] understand how stage works

2 files changed, 675 insertions(+)

**create** mode 100644 LICENSE

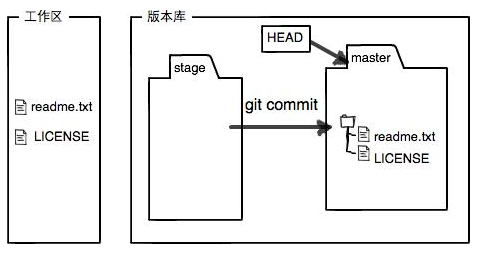
一旦提交后，如果没有对工作区做任何修改，那么工作区就是干净的：

$ git status

*# On branch master*

nothing to commit (working directory clean)

现在，版本库变成这样，暂存区就没有任何内容了：



## 管理修改

什么是修改？

新增一行，删除一行，更改了某个字符，创建一个新文件等等，都是一个修改。

Git跟踪并管理的是修改，而非文件。更确切地说，Git管理的是暂存区的修改，工作区的修改只有被git add添加到暂存区后，再通过git commit命令提交才能被Git管理。

## 撤销修改

如果在readme.txt中添加了一行：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

My stupid boss still prefers SVN.

使用git status查看：

$ git status

*# On branch 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.txt*

*#*

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

Git告诉我们，git checkout – file可以丢弃工作区的修改：

$ git checkout -- readme.txt

git checkout – readme.txt意思是，把readme.txt工作区的修改全部撤销，这里有两种情况：

一种是readme.txt修改后还没有被放到暂存区，现在撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区，又做了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次git commit或git add时的状态。

git checkout – file命令中的—很重要，没有它，就变成“切换到另一个分支”的命令。

假设修改后的readme.txt被git add到暂存区了：

$ git add readme.txt

用git status查看，修改只是被添加到了暂存区，还没有提交到版本库：

$ git status

*# On branch master*

*# Changes to be committed:*

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

*#*

*# modified: readme.txt*

Git告诉我们，用git reset HEAD file可以把暂存区的修改撤销掉(unstage)，重新放回工作区：

$ git re**set** HEAD readme.txt

Unstaged changes after reset:

M readme.txt

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区，用HEAD表示最新版本。

## 删除文件

添加一个新文件test.txt并且提交:

$ git add test.txt

$ git **commit** -m "add test.txt"

[master 94cdc44] **add** test.txt

1 file changed, 1 insertion(+)

**create** mode 100644 test.txt

一般情况下，在文件管理器或者用rm命令直接把文件删了：

$ rm test.txt

这个时候，Git知道你把文件删除了，因此，工作区和版本就不一致了，git status命令可以知道哪些文件被删除了：

$ git status

*# On branch master*

*# Changes not staged for commit:*

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

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

*#*

*# deleted: test.txt*

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

现在有两种情况，一是确实要从版本库中删除该文件，那就用git rm命令删掉，并且git commit：

$ git rm test.txt

rm 'test.txt'

$ git **commit** -m "remove test.txt"

[master d17efd8] remove test.txt

1 file changed, 1 deletion(-)

**delete** mode 100644 test.txt

另一种情况是，工作区的文件被误删了，因为版本库里还有，可以很轻松地把误删的文件恢复到最新版本：

$ git checkout -- test.txt

git checkout其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

# 远程仓库

实际情况往往是这样的，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。通常GitHub便是这样的服务器仓库。

在GitHub网站注册账号，本地仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

创建SSH Key，在用户主目录下，输入命令：

$ ssh-keygen -t rsa -C "youremail@example.com"

一路回车，使用默认值即可。主目录会多一个.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，分别是私钥和公钥。

登录GitHub，settings -> SSH Keys -> Add SSH Key，填上Title，在Key文本框里粘贴id\_rsa.pub文件的内容。GitHub允许添加多个Key，在GitHub上免费托管的Git仓库，任何人都可以看到，但只有自己能修改。

## 添加远程库

在GitHub创建一个仓库，名为learngit。

在本地learngit仓库下运行命令：

$ git remote add origin git@github.com:yun-w/learngit.git

origin是远程库的名字，默认的叫法，下一步就可以把本地库的所有内容推送到远程库上：

$ git push -u origin master

Counting objects: 19, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (19/19), done.

Writing objects: 100% (19/19), 13.73 KiB, done.

Total 23 (delta 6), reused 0 (delta 0)

To git@github.com:yun-w/learngit.git

\* [new branch] master -> master

Branch master **set** up **to** track remote branch master **from** origin.

把本地的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。由于远程库是空的，第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送到远程的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

从现在起，同过以下命令就可以把本地的master分支的最新修改推送至GitHub：

$ git push origin master

**SSH警告**

当第一次使用Git的clone或者push命令连接GitHub时，会得到一个警告：

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

这是因为Git第一次使用SSH连接时，需要你确认GitHub的Key的指纹信息是否真的来自GitHub服务器，输入yes即可。此时，Git会发出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

Warning: Permanently added 'github.com' (RSA) to the **list** of known hosts.

这个警告只会出现一次，后面的操作便不会再出现了。

## 从远程库克隆

登录GitHub，创建一个新的仓库，名字叫gitskills，勾选Initialize this repository with a README，这样GitHub会为我们自动创建一个README.md文件，使用命令git clone克隆一个本地库：

$ git clone git@github.com:michaelliao/gitskills.git

Cloning into 'gitskills'...

remote: Counting objects: 3, done.

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

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

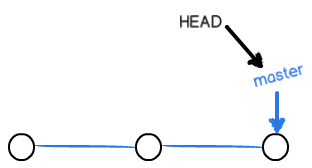
GitHub还可以使用<https://github.com/yun-w/gitskills>这样的地址，实际上，Git支持多种协议，使用https除了速度慢以外，最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能使用https。

# 分支管理

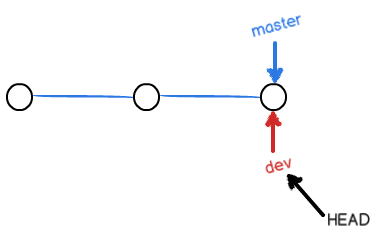
在多人协作开发时，每个人可以创建自己的分支，互不干扰，开发完毕后再合并到原来的分支上。

## 创建与合并分支

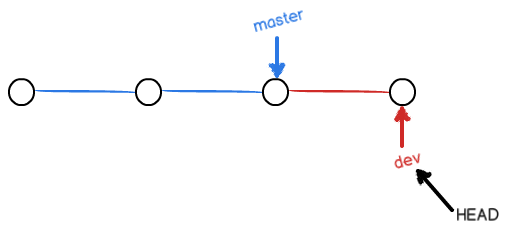
在版本退回里，已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。master分支是Git的主分支，HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前的分支。每次提交，master分支都会向前移动一步，这样，随着不断提交，master分支线也越来越长：



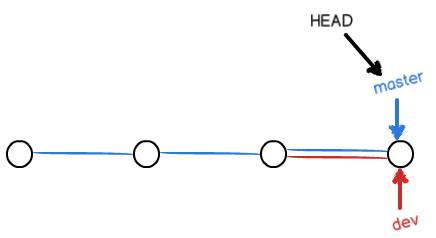
当我们创建新的分支时，例如dev分支，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：



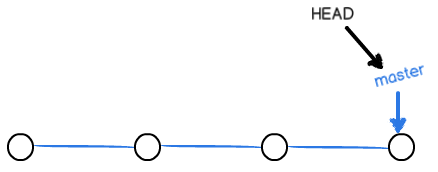
Git创建一个分支很快，增加一个dev指针，修改HEAD的指向，工作区的文件都没有任何变化。不过，从现在开始，对工作区的修改和提交就是针对dev分支，比如新提交一次后，dev指针往前移动一步，而master的指针不变：



假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，完成合并：



Git合并分支速度也很快，就修改指针，工作区内容也不变。合并分支完成后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，最后我们就剩下一条master分支：



下面开始实践，首先，创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

git checkout命令加上-b参数表示创建并切换，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

然后，用git branch命令查看所有分支，当前分支前面会标有一个\*号：

$ git branch

\* dev

master

然后，我们就可以在dev分支上正常提交，如对readme.txt做个修改，加上一行：

Creating a **new** branch is quick.

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev fec145a] branch test

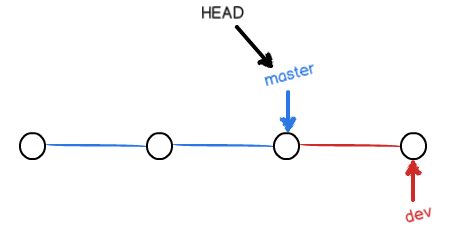
1 file changed, 1 insertion(+)

现在，dev分区工作完成，我们就可以切换回master分支：

$ git checkout master

Switched to branch 'master'

切换回master分支后，并没有刚才在readme.txt添加的内容，因为那是在dev分支上提交的，而master分支此刻的提交点并没有变化：



现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d17efd8..fec145a

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge用于合并指定分支到当前分支。和并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交。当然，也不是每次合并都能Fast-forward的。

合并完成后，就可以放心地删除dev分支了，最后再查看branch，就只剩master分支了：

$ git branch -d dev

Deleted branch dev (was fec145a).

$ git branch

\* master

因为创建、合并和删除分支非常快，所以Git鼓励使用分支完成某个任务，合并后再删掉分支，这和直接在master分支上工作效果是一样的，但过程更安全。

**小结：**

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建并合并分支：git checkout –b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch –d <name>

## 解决冲突

合并分支往往也不是一帆风顺的。准备新的feature1分支，继续我们的新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

修改readme.txt最后一行，改为：

Creating a **new** branch is quick **AND** simple.

在feature1分支上提交：

$ git add readme.txt

$ git **commit** -m "AND simple"

[feature1 75a857c] **AND** simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 1 **commit**.

Git还会自动提醒我们当前master分支比远程的master分支要超前1个提交。在master分支上把readme.txt文件的最后一行改为：

Creating a **new** branch is quick & simple.

提交：

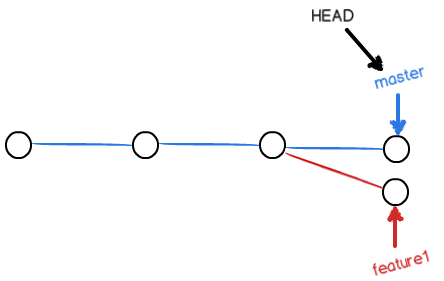
$ git add readme.txt

$ git commit -m "& simple"

[master 400b400] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

现在master分支和feature1分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就有可能会有冲突，我们试试看：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict **in** readme.txt

Automatic merge failed; fix conflicts **and** **then** commit the result.

果然冲突了！Git告诉我们，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status也可以告诉我们冲突的文件：

$ git status

*# On branch master*

*# Your branch is ahead of 'origin/master' by 2 commits.*

*#*

*# Unmerged paths:*

*# (use "git add/rm <file>..." as appropriate to mark resolution)*

*#*

*# both modified: readme.txt*

*#*

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

我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：

Creating a **new** branch is quick **and** simple.

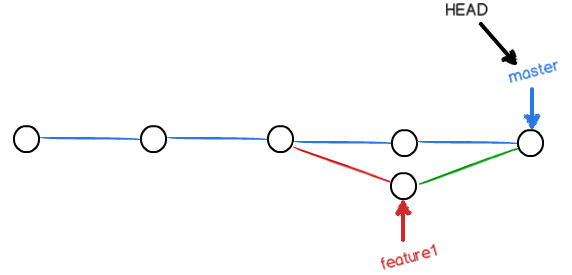
再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master 59bc1cb] conflict fixed

现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支合并的情况：

$ git log --graph --pretty=oneline --abbrev-commit

\* 59bc1cb conflict fixed

|\

| \* 75a857c **AND** simple

\* | 400b400 & simple

|/

\* fec145a branch test

...

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 75a857c).

工作完成。

## 分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。下面我们实战一下—no--ff方式的git merge。

首先，仍然创建并切换dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

修改readme.txt文件，并提交到一个新的commit：

$ git add readme.txt

$ git **commit** -m "add merge"

[dev 6224937] **add** merge

1 file changed, 1 insertion(+)

现在我们切换回master：

$ git checkout master

Switched to branch 'master'

准备合并dev分支，请注意—no--ff参数，表示禁用Fast forward：

$ git merge --no-ff -m "merge with no-ff" dev

Merge made by the 'recursive' strategy.

readme.txt | 1 +

1 file changed, 1 insertion(+)

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。合并后，我们再用git log看看分支历史：

$ git log --graph --pretty=oneline --abbrev-commit

\* 7825a50 merge **with** no-ff

|\

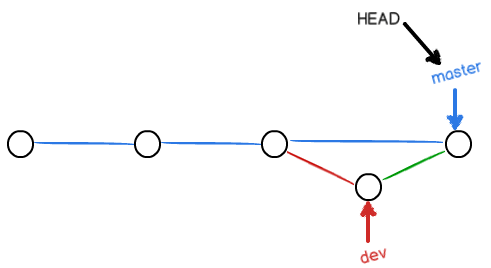
| \* 6224937 add merge

|/

\* 59bc1cb conflict fixed

...

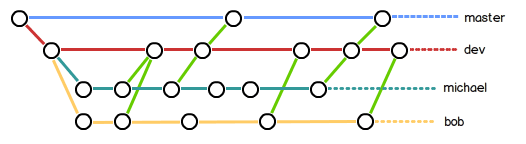
可以看到，不使用Fast forward模式，merge后就像这样：



**分支策略**

在实际开发中我们应该按照几个基本原则进行分支管理：

首先，master分支应该非常稳定，也就是仅用来发布新版本，平时不能在上面干活；那在哪干活呢？在dev上！也就是dev是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了，所以团队合作的分支看起来就像这样：



## bug分支

软件开发中，bug就像家常便饭一样。有了bug就需要修复，在Git中，由于分支是如此的强大，所以每个bug都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

当你接到一个修复代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

$ git status

*# On branch dev*

*# Changes to be committed:*

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

*#*

*# new file: hello.py*

*#*

*# 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.txt*

*#*

并不是你不想提交，而是工作只进行到一半，还没法提交，预计还需要1天时间。但是，必须在两个小时内修复该bug，怎么办呢？

幸好，Git提供了一个stash功能，可以把当前工作储藏起来，等以后恢复现场后继续工作：

$ git stash

Saved working directory **and** index state WIP on dev: 6224937 add merge

HEAD is now at 6224937 add merge

现在，用git status查看工作区，就是干净的，因此可以放心地创建新分支来修复bug。

首先确定要在那个分支上修复bug，假定需要在master分支上修复，就从master创建临时分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 6 commits.

$ git checkout -b issue-101

Switched to a new branch 'issue-101'

现在修复bug，需要把“Git is free software…”改为“Git is a free software…”,然后提交：

$ git add readme.txt

$ git commit -m "fix bug 101"

[issue-101 cc17032] fix bug 101

1 file changed, 1 insertion(+), 1 deletion(-)

修复完成后，切换到master分支，完成合并，最后删除issue-101分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 2 commits.

$ git merge --no-ff -m "merged bug fix 101" issue-101

Merge made by the 'recursive' strategy.

readme.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

$ git branch -d issue-101

Deleted branch issue-101 (was cc17032).

太棒了，原计划两小时的bug修复只花了5分钟！现在，是时候接着回来dev分支干活了：

$ git checkout dev

Switched to branch 'dev'

$ git status

*# On branch dev*

nothing to commit (working directory clean)

工作区是干净的，刚才的工作现场存到哪里去了？用git stash list命令看看：

$ git stash list

stash@{0}: WIP on dev: 6224937 add merge

工作现场还在，Git把stash内容存到某个地方去了，但是需要回复一下，有两个办法：

一是用git stash apply恢复，但是恢复后，stash内容并不删除，你需要用git stash drop来删除；

另一种方式是用git stash pop，恢复的同时把stash内容也删了：

$ git stash pop

*# On branch dev*

*# Changes to be committed:*

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

*#*

*# new file: hello.py*

*#*

*# 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.txt*

*#*

Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)

再用git stash list查看，就看不到任何stash内容了：

$ git stash list

你可以多次stash，恢复的时候，先用git stash list查看，然后恢复指定的stash，用命令：

$ git stash apply stash@{0}

## Feature分支

软件开发中，总有无穷无尽的新功能要不断添加进来。添加一个新功能时，你肯定不希望一些实验性的代码，把主分支搞乱了，所以，每添加一个新功能，最好新建一个feature分支，在上面开发，完成后，合并，最后，删除该feature分支。

现在，你终于接到一个新任务：开发代号为Vulcan的新功能，该功能计划用下一代星际飞船。于是准备开发：

$ git checkout -b feature-vulcan

Switched to a new branch 'feature-vulcan'

5分钟后，开发完毕：

$ git add vulcan.py

$ git status

*# On branch feature-vulcan*

*# Changes to be committed:*

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

*#*

*# new file: vulcan.py*

*#*

$ git commit -m "add feature vulcan"

[feature-vulcan 756d4af] add feature vulcan

1 file changed, 2 insertions(+)

create mode 100644 vulcan.py

切回dev，准备合并：

$ git checkout dev

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。但是，就在此时，接到上级命令，因经费不足，新功能必须取消！

虽然白干了，但是这个分支还是必须就地销毁：

$ git branch -d feature-vulcan

error: The branch 'feature-vulcan' is **not** fully merged.

If you are sure you want to delete it, run 'git branch -D feature-vulcan'.

销毁失败。Git友情提示，feature-vulcan分支还没有被合并，如果删除，将丢失修改，如果强行删除，需要使用命令git branch -D feature-vulcan：

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 756d4af).

终于删除成功！

## 多人协作

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。

要查看远程库的信息，用git remote：

$ git remote

origin

或者，用git remote -v显示更详细的信息：

$ git remote -v

origin git@github.com:michaelliao/learngit.git (fetch)

origin git@github.com:michaelliao/learngit.git (push)

上面显示了可以抓取和推送的origin的地址。如果没有推送权限，就看不到push的地址。

**推送分支**

推送分支，就是把该分支上的所有本地提交推送到远程库。推送时，要指定本地分支，这样，Git就会把该分支推送到远程库对应的远程分支上：

$ git push origin master

如果要推送其它分支，如dev，就改成：

$ git push origin dev

哪些分支需要推送，哪些不需要呢？

master分支是主分支，因此要时刻与远程同步；

dev分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步；

bug分支只用于在本地修复bug，就没必要推送到远程了，除非老板要看看你每周到底修复了几个bug；

feature分支是否推送到远程，取决于你是否和你的小伙伴合作在上面开发。

总之，就是在Git中，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定。

**抓取分支**

多人协作时，大家都会往master和dev分支上推送各自的修改。

模拟一个小伙伴，在同一台电脑的另一个目录下克隆：

$ git clone git@github.com:michaelliao/learngit.git

Cloning into 'learngit'...

remote: Counting objects: 46, done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 46 (delta 16), reused 45 (delta 15)

Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.

Resolving deltas: 100% (16/16), done.

当你的小伙伴从远程clone时，默认情况下，你的小伙伴只能看到本地的master分支，用git branch查看：

$ git branch

\* master

现在，你的小伙伴要在dev分支上开发，就必须创建远程origin的dev分支到本地，于是他用这个命令创建本地dev分支：

$ git checkout -b dev origin/dev

现在，他就可以在dev上继续修改，然后，时不时地把dev分支push到远程：

$ git **commit** -m "add /usr/bin/env"

[dev 291bea8] **add** /usr/bin/env

1 file changed, 1 insertion(+)

$ git push origin dev

Counting objects: 5, done.

Delta compression **using** up **to** 4 threads.

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

Writing objects: 100% (3/3), 349 bytes, done.

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

**To** git@github.com:michaelliao/learngit.git

fc38031..291bea8 dev -> dev

你的小伙伴已经向origin/dev分支推送了他的提交，而碰巧你也对同样的文件做了修改，并试图推送：

$ git add hello.py

$ git **commit** -m "add coding: utf-8"

[dev bd6ae48] **add** coding: utf-8

1 file changed, 1 insertion(+)

$ git push origin dev

**To** git@github.com:michaelliao/learngit.git

! [rejected] dev -> dev (non-fast-forward)

error: failed **to** push **some** refs **to** 'git@github.com:michaelliao/learngit.git'

hint: Updates were rejected because the tip **of** your **current** branch **is** behind

hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')

hint: before pushing again.

hint: See the 'Note about fast-forwards' **in** 'git push --help' **for** details.

推送失败，因为你的小伙伴的最新提交和你试图推送的提交有冲突。Git提示，先用git pull把最新的提交从origin/dev抓取下来，然后，在本地合并，解决冲突，再推送：

$ git pull

remote: Counting objects: 5, done.

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

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

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

From github.com:michaelliao/learngit

fc38031..291bea8 dev -> origin/dev

There is no tracking information **for** the current branch.

Please specify which branch you want to merge **with**.

See git-pull(1) **for** details

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream dev origin/<branch>

git pull也失败了，原因是没有指定本地dev分支与远程分支origin/dev分支的链接，根据提示，设置dev和origin/dev的链接：

$ git branch *--set-upstream dev origin/dev*

Branch dev **set** up **to** track remote branch dev **from** origin.

再pull：

$ git pull

Auto-merging hello.py

CONFLICT (content): Merge conflict **in** hello.py

Automatic merge failed; fix conflicts **and** **then** commit the result.

这回git pull成功，但是合并冲突需要手动解决，方法和分支管理的解决冲突完全一样。解决后，提交，再push:

$ git **commit** -m "merge & fix hello.py"

[dev adca45d] merge & fix hello.py

$ git push origin dev

Counting objects: 10, done.

Delta compression **using** up **to** 4 threads.

Compressing objects: 100% (5/5), done.

Writing objects: 100% (6/6), 747 bytes, done.

Total 6 (delta 0), reused 0 (delta 0)

**To** git@github.com:michaelliao/learngit.git

291bea8..adca45d dev -> dev

因此，多人协作的工作模式通常是这样的：

1. 首先，可以试用git push origin branch-name推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用git pull试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用git push origin branch-name推送就能成功！

如果git pull提示“no tracking information”，则说明本地分支和远程分支的链接关系没有创建，用命令git branch --set-upstream branch-name origin/branch-name。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

# 标签管理

发布一个版本时，通常先在版本库中打一个标签，这样，就唯一确定了打标签时刻的版本。将来取某个标签的版本时，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。Git的标签实际就是指向某个commit的指针，与分支很像，但是分支可以移动，标签不能移动，所以创建和删除标签都是瞬间完成的。

## 创建标签

首先，切换到需要打标签的分支上：

$ git branch

\* dev

master

$ git checkout master

Switched to branch 'master'

然后，使用命令git tag <name>打一个新标签：

$ git tag v1.0

使用git tag命令可以查看所有标签：

$ git tag

v1.0

默认标签是打在最新提交的commit上，如果需要打在过去的提交上，怎么办？找的历史提交的commit id，然后打上就可以了：

$ git log --pretty=oneline --abbrev-commit

6a5819e merged bug fix 101

cc17032 fix bug 101

7825a50 merge with no-ff

6224937 add merge

59bc1cb conflict fixed

400b400 & simple

75a857c AND simple

fec145a branch test

d17efd8 remove test.txt

...

假如需要对add merge这次提交打标签，它对应的commit id是6224937，敲入命令：

$ git tag v0.9 6224937

再用git tag查看标签：

$ git tag

v0.9

v1.0

标签不是按时间顺序列出，而是按字母排列。可以使用git show <tagname>查看标签信息：

$ git **show** v0.9

**commit** 622493706ab447b6bb37e4e2a2f276a20fed2ab4

Author: Michael Liao <askxuefeng@gmail.com>

**Date**: Thu Aug 22 11:22:08 2013 +0800

**add** merge

...

可以看到，v0.9确实打在了add merge这次提交上。

还可以创建带说明的标签，用-a指定标签名，-m指定说明文字：

$ git tag -a v0.1 -m "version 0.1 released" 3628164

用命令git show <tagname>可以看到说明文字：

$ git show v0.1

tag v0.1

Tagger: Michael Liao <askxuefeng@gmail.com>

Date: Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0

Author: Michael Liao <askxuefeng@gmail.com>

Date: Tue Aug 20 15:11:49 2013 +0800

append GPL

还可以通过-s用私钥签名一个标签：

$ git tag -s v0.2 -m "signed version 0.2 released" fec145a

签名采用PGP签名，因此，必须先安装gpg(GnuPG)，如果没有找到gpg，或者没有gpg密钥对，就会报错：

gpg: signing failed: secret key **not** available

error: gpg failed to sign the data

error: unable to sign the tag

如果报错，请参考GnuPG帮助文档配置Key。用命令git show <tagname>可以看到PGP签名信息：

$ git **show** v0.2

tag v0.2

Tagger: Michael Liao <askxuefeng@gmail.com>

**Date**: Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released

-----**BEGIN** PGP SIGNATURE-----

Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...

-----**END** PGP SIGNATURE-----

**commit** fec145accd63cdc9ed95a2f557ea0658a2a6537f

Author: Michael Liao <askxuefeng@gmail.com>

**Date**: Thu Aug 22 10:37:30 2013 +0800

branch test

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。

## 操作标签

如果标签打错了，也可以删除：

$ git tag -d v0.1

Deleted tag 'v0.1' (was e078af9)

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。如果要推送某个标签到远程，使用命令git push origin <tagname>：

$ git push origin v1.0

Total 0 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [new tag] v1.0 -> v1.0

或者，一次性推送全部尚未推送到远程的本地标签：

$ git push origin --tags

Counting objects: 1, done.

Writing objects: 100% (1/1), 554 bytes, done.

Total 1 (delta 0), reused 0 (delta 0)

To git@github.com:michaelliao/learngit.git

\* [**new** tag] v0.2 -> v0.2

\* [**new** tag] v0.9 -> v0.9

如果标签已经推送到远程，要删除远程标签，首先，先从本地删除：

$ git tag -d v0.9

Deleted tag 'v0.9' (was 6224937)

然后，从远程删除。删除命令也是push，格式如下：

$ git push origin :refs/tags/v0.9

To git@github.com:michaelliao/learngit.git

- [deleted] v0.9

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

# 使用GitHub

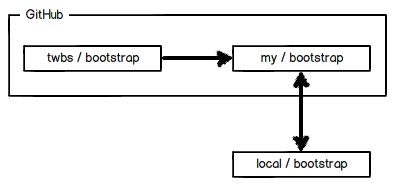
我们一直使用GitHub作为免费的远程仓库。GitHub还是一个开源协作社区，利用Git极其强大的克隆和分支功能，可以让别人参与你的开源项目，也可以参与别人的开源项目。

如何参与一个开源项目呢？比如人气极高bootstrap项目，这是一个非常强大的CSS框架，

从<https://github.com/twbs/bootstrap>点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

git clone git@github.com:michaelliao/bootstrap.git

一定要从自己的账号下克隆仓库，这样你才能推送修改。如果从bootstrap的作者的仓库地址git@github.com:twbs/bootstrap.git克隆，因为没有权限，你将不能推送。关系图如下：



如果你想修复bootstrap的一个bug，或者新增一个功能，干完活后往自己的仓库推送。如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受就不一定了。

# 自定义Git

在安装Git一节中，我们已经配置了user.name和user.email，实际上，Git还有很多配置项。比如，让Git显示颜色，会让命令输出看起来更醒目：

$ git config --global color.ui **true**

这样Git会适当显示不同的颜色。

## 忽略特殊文件

有时候，你必须把某些文件放到Git工作目录中，但又不能提交它们，比如数据库密码的配置文件等，每次git status都会显示Untracked files …，等。解决方法是在Git工作区目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需组合一下就可以使用了。所有配置文件可以直接在线浏览：<https://github.com/github/gitignore>。忽略文件的原则是：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如Java编译产生的.class文件；
3. 忽略你自己的带有敏感信息的配置文件，比如存放口令的配置文件。

举个例子：

假设你在windows下进行Python开发，Windows会自动在有图片的目录下生成隐藏的缩略图文件，如果有自定义目录，目录下就会有Desktop.ini文件，因此你需要忽略Windows自动生成的垃圾文件：

*# Windows:*

Thumbs.db

ehthumbs.db

Desktop.ini

然后忽略Python编译产生的.pyc、.pyo、dist等文件或目录：

*# Python:*

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

加上你自己的定义文件，最终得到一个完整的.gitignore文件，内容如下：

*# Windows:*

Thumbs.db

ehthumbs.db

Desktop.ini

*# Python:*

\*.py[cod]

\*.so

\*.egg

\*.egg-info

dist

build

*# My configurations:*

db.ini

deploy\_key\_rsa

最后一步就是把.gitignore也提交到Git，就完成了。检验gitignore的标准是git status命令是不是说working directory clean。

## 配置别名

让git st就表示git status：

$ git config --global **alias**.st status

其它命令的简写：

$ git config --global **alias**.co checkout

$ git config --global **alias**.ci commit

$ git config --global **alias**.br branch

以后提交就可以简写成：

$ git ci -m "bala bala bala..."

--global参数是全局参数，也就是这些命令在这台电脑的所有Git仓库下都有用。

在撤销修改一节中，我们知道，命令git reset HEAD file可以吧暂存区的修改撤销掉(unstage)，重新放回工作区。既然是一个unstage操作，就可以配置一个unstage别名：

$ git config --global **alias**.unstage 'reset HEAD

当你敲命令：

$ git unstage test.py

实际上Git执行的是：

$ git re**set** HEAD test.py

配置一个git last，让其显示最后一次提交信息：

$ git config --global **alias**.last 'log -1'

这样，用git last就能显示最近一次的提交：

$ git last

commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2

Merge: bd6ae48 291bea8

Author: Michael Liao <askxuefeng@gmail.com>

Date: Thu Aug 22 22:49:22 2013 +0800

merge & fix hello.py

甚至还有人丧心病狂地把lg配置成了：

git config --global **alias**.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

**配置文件**

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。每个仓库的配置文件都放在.git/config文件中：

$ cat .git/config

[core]

repositoryformatversion = 0

filemode = **true**

bare = **false**

logallrefupdates = **true**

ignorecase = **true**

precomposeunicode = **true**

[remote "origin"]

url = git@github.com:michaelliao/learngit.git

fetch = +refs/heads/\*:refs/remotes/origin/\*

[branch "master"]

remote = origin

merge = refs/heads/master

[**alias**]

last = log -1

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

而当前用户的Git配置文件放在用户主目录下的一个隐藏文件.gitconfig中：

$ cat .gitconfig

[**alias**]

co = checkout

ci = commit

br = branch

st = status

[user]

name = Your Name

email = your@email.com

配置别名也可以直接修改这个文件，如果改错了，可以删除文件重新通过命令配置。

小结：给Git配置好别名，就可以输命令时偷个懒，我们鼓励偷懒。

## 搭建Git服务器

GitHub是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

第一步，安装git：

$ su**do** apt-**get** install git

第二步，创建一个git用户，用来运行git服务：

$ su**do** adduser git

第三步，创建证书登录：

收集所有需要登录的用户的公钥，就是他们自己的id\_rsa.pub文件，把所有公钥导入到/home/git/.ssh/authorized\_keys文件里，一行一个。

第四步，初始化Git仓库：

先选定一个目录作为Git仓库，假定是/srv/sample.git，在/srv目录下输入命令：

$ su**do** git init --bare sample.git

Git就会创建一个裸仓库，裸仓库没有工作区，因为服务器上的Git仓库纯粹是为了共享，所以不让用户直接登录到服务器上去改工作区，并且服务器上的Git仓库通常都以.git结尾。然后，把ower改为git：

$ su**do** chown -R git:git sample.git

第五步，禁用shell登录：

出于安全考虑，第二步创建的git用户不允许登录shell，可以通过编辑/etc/passwd文件完全。找到类似下面的一行：

git:x:1001:1001:,,,:/home/git:/bin/bash

改为：

git:x:1001:1001:,,,:/home/git:/usr/bin/git-shell

这样，git用户可以正常通过ssh使用git，但无法登录shell，因为我们为git用户指定的git-shell每次一登录就自动退出。

第六步，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

$ git clone git@server:/srv/sample.git

Cloning into 'sample'...

warning: You appear to have cloned an empty repository.

**管理公钥**

如果团队很小，把每个人的公钥收集起来放到服务器的/home/git/.ssh/authorized\_keys文件里就是可行的。如果团队有几百号人，就没法这么玩了，这时，可以用Gitosis来管理公钥。这里不介绍怎么玩Gitosis了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

**管理权限**

视源代码如生命的公司，会视员工为窃贼，会在版本控制系统里设置一套完善的权限控制，每个人是否有读写权限会精确到每个分支甚至每个目录下。因为Git是为Linux源代码托管而开发的，所以Git也继承了开源社区的精神，不支持权限控制。不过，因为Git支持钩子(hook)，所以，可以在服务器端编写一系列脚本来控制提交等操作，达到权限控制的目的。Gitolite就是这个工具。

# 总结

Git虽然极其强大，命令繁多，但常用的就那么十来个，掌握好这十几个常用命令，就可以得心应手地使用Git了。