|  |
| --- |
| 西安艾科特声学科技有限公司 |
| Git - Manual |
| ——Bash命令语句 |

|  |
| --- |
| 李荣  2018.06 |

目 录

[1. Git简介 1](#_Toc516753120)

[2. 安装Git (Windows) 5](#_Toc516753121)

[3. 创建版本库 5](#_Toc516753122)

[4. 时光穿梭机 7](#_Toc516753123)

[4.1 版本回退 9](#_Toc516753124)

[4.2 工作区和暂存区 13](#_Toc516753125)

[4.3 管理修改 16](#_Toc516753126)

[4.4 撤销修改 18](#_Toc516753127)

[4.5 删除文件 20](#_Toc516753128)

[5. 远程仓库 21](#_Toc516753129)

[5.1 添加远程库 22](#_Toc516753130)

[5.2 从远程库克隆 24](#_Toc516753131)

[6. 分支管理 25](#_Toc516753132)

[6.1 创建与合并分支 25](#_Toc516753133)

[6.2 解决冲突 29](#_Toc516753134)

[6.3 分支管理策略 32](#_Toc516753135)

[6.4 Bug分支 34](#_Toc516753136)

[6.5 Feature分支 36](#_Toc516753137)

[6.6 多人协作 37](#_Toc516753138)

[6.7 Rebase 41](#_Toc516753139)

[7. 标签管理 46](#_Toc516753140)

[7.1 创建标签 46](#_Toc516753141)

[7.2 操作标签 48](#_Toc516753142)

[8. 自定义Git 49](#_Toc516753143)

[8.1 忽略特殊文件 50](#_Toc516753144)

[8.2 配置别名 52](#_Toc516753145)

# Git简介

Git 是用于 Linux内核开发的版本控制工具。与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持（注：这得分是用什么样的服务端，使用http协议或者git协议等不太一样。并且在push和pull的时候和服务器端还是有交互的），使源代码的发布和交流极其方便。 Git 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。 Git 最为出色的是它的合并跟踪（merge tracing）能力。

实际上内核开发团队决定开始开发和使用 Git 来作为内核开发的版本控制系统的时候，世界开源社群的反对声音不少，最大的理由是 Git 太艰涩难懂，从 Git 的内部工作机制来说，的确是这样。但是随着开发的深入，Git 的正常使用都由一些友好的脚本命令来执行，使 Git 变得非常好用，即使是用来管理自己的开发项目，Git 都是一个友好，有力的工具。现在，越来越多的著名项目采用 Git 来管理项目开发.

作为开源自由原教旨主义项目，Git 没有对版本库的浏览和修改做任何的权限限制。

简单来说，如果用Microsoft Word写过长篇大论，一定会有这种情况：想删除一个段落，又怕将来想恢复找不回来怎么办？有办法，先把当前文件“另存为……”一个新的Word文档，再接着改，改到一定程度，再“另存为……”一个新文件，这样一直改下去，最后的Word文档如图1-1所示。



图1-1 多版本Word文档

这样的Word文档存在的问题主要有以下几个方面：

一段时间后，如果想找回被删除的文字，但是已经记不清删除前保存在哪个文件里了，只好一个一个文件去找，比较麻烦。

看着一堆乱七八糟的文件，想保留最新的一个，然后把其他的删掉，又怕哪天会用上，不能轻易删除。

更不便的是，有些部分需要其他同事帮助填写，于是把文件Copy到U盘里给他（也可能通过Email发送一份给他），然后，自己继续修改Word文件。一天后，同事再把Word文件传回，此时，必须清楚地知道发给他之后到收到他的文件期间，自己作了哪些改动，然后把自己的改动和她的部分合并，比较困难。

而Git不但能自动记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆类似的文件了，也不需要把文件传来传去。如果想查看某次改动，只需要在软件Git里查看即可，会很方便。如图1-2所示。



图1-2 文件改动记录

这样，就结束了手动管理多个“版本”的史前时代，进入到版本控制的20世纪。

**集中式VS分布式**

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，而干活的时候，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，完成后再把自己的工作内容推送给中央服务器。中央服务器就好比是一个图书馆，如果要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。



图1-3 集中式版本控制系统

集中式版本控制系统最大的弊端就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，非常影响工作效率。

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，工作的时候，就不需要联网了，因为版本库就在自己的电脑上。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？比如在自己电脑上改了文件A，同事也在他的电脑上改了文件A，这时，俩人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，如果某一个人的电脑坏掉了，可以任意从其他人那里进行复制。而如果集中式版本控制系统的中央服务器出了问题，所有人都无法继续工作。

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，因为俩人可能不在一个局域网内，两台电脑不能互相访问，也可能今天同事病了，他的电脑没有开机。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，没有它大家也可以工作，只是交换修改不方便而已。



图1-4 分布式版本控制系统

当然，Git的优势不单是不必联网这么简单，后面还会看到Git极其强大的分支管理，把SVN等远远抛在了后面。

CVS作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人在用。由于CVS自身设计的问题，会造成提交文件不完整，版本库莫名其妙损坏的情况。同样是开源而且免费的SVN修正了CVS的一些稳定性问题，是目前用得最多的集中式版本库控制系统。

除了免费的外，还有收费的集中式版本控制系统，比如IBM的ClearCase（以前是Rational公司的，被IBM收购了），特点是安装文件很大，运行速度很慢。

微软自己也有一个集中式版本控制系统叫VSS，集成在Visual Studio中，但使用不便。

分布式版本控制系统除了Git以及促使Git诞生的BitKeeper外，还有类似Git的Mercurial和Bazaar等。这些分布式版本控制系统各有特点，但最快、最简单也最流行的依然是Git！

# 安装Git (Windows)

在Windows上使用Git，可以从Git官网 (<https://git-scm.com/downloads>) 直接下载安装程序，然后按默认选项安装即可。

安装完成后，在开始菜单里找到“Git”→“Git Bash”，弹出一个类似命令行窗口的东西，就说明Git安装成功！

安装完成后，还需要最后一步设置，在命令行输入：

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

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

因为Git是分布式版本控制系统，所以每个机器都必须设置自己的名称和Email地址。

※--global参数是全局参数，表示本机所有Git仓库都会使用该配置。

# 创建版本库

版本库又名仓库，英文名repository，可以简单理解成一个目录，这个目录里面的所有文件都可以被Git管理起来，每个文件的修改、删除，Git都能跟踪，以便任何时刻都可以追踪历史，或者在将来某个时刻可以“还原”。

创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录。

$ *mkdir learngit*

$ *cd learngit*

$ *pwd*

*/ D: / Git/ usr/ learngit*

pwd命令用于显示当前目录。Windows系统中，为避免遇到各种莫名其妙的问题，必须确保目录名（包括父目录）不包含中文。

然后，通过git init命令把这个目录变成Git可以管理的仓库。

$ *git init*  初始化Git仓库。

*Initialized empty Git repository in /* *D: /* *Git/* *usr/* *learngit/.git/*

至此建成一个空的仓库（empty Git repository），此时可以看到当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，如果手动修改这个目录里面的文件，就会破坏Git仓库，带来不必要的麻烦。

**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更改，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以记录每次的更改，比如在第5行加了一个单词“Linux”，在第8行删了一个单词“Windows”。而图片、视频等二进制文件，虽然也能由版本控制系统管理，但无法跟踪文件的变化，只能把二进制文件每次的更改串起来，也就是说，只知道图片从100KB改成了120KB，但不知道更改的具体内容。

而Microsoft Word文档是二进制格式的，因此，版本控制系统无法跟踪Word文件的更改，前面的例子只是为了演示，如果要真正使用版本控制系统记录每次的更改内容，就要以纯文本方式编写文件。

因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

另外，Windows系统中，不要使用其自带的记事本编辑文本文件。原因是Microsoft开发记事本的团队在每个文件开头添加了0xefbbbf（十六进制）的字符，来保存UTF-8编码的文件，会遇到很多不可思议的问题。比如，网页第一行可能会显示一个“?”，正确的程序一编译就报语法错误，等等。建议下载Notepad++代替记事本，不但功能强大，而且免费。安装后需要把Notepad++的默认编码设置为UTF-8 without BOM。

现在编写一个readme.txt文件，内容如下：

*Git is a version control system.*

*Git is free software.*

将文件放至learngit目录下（或其子目录下），也就是放在创建的版本库文件夹中，然后再将其放到Git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*  可反复多次使用，添加多个文件。

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"*  可以一次提交多个文件，引号中内容用于自己记录本次提交的用意。

*[master (root-commit) eaadf4e] wrote a readme file*

*1 file changed, 2 insertions(+)*

*create mode 100644 readme.txt*

git commit命令执行成功后会显示：1 file changed，即1个文件被改动（新添加的readme.txt文件）；2 insertions (+)，插入了两行内容（readme.txt有两行内容）。

Git添加文件需要add，commit两步是因为commit可以一次提交很多文件，所以可以多次add不同的文件。

$ *git add file1.txt*

$ *git add file2.txt file3.txt*

$ *git commit -m "add 3 files."*

# 时光穿梭机

现在已经成功地添加并提交了一个readme.txt文件，然后继续修改readme.txt文件，内容如下：

*Git is a distributed version control system.*

*Git is free software.*

然后，可以运行git status命令查看仓库当前的状态。

$ *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")*

git status命令可以时刻掌握仓库当前的状态，上面的命令输出显示，readme.txt被修改过了，但还没有准备提交的修改。

看到此次修改记录后，如果想知道具体修改内容，可以用git diff命令查看。

$ *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.*

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，在第一行添加了一个distributed单词。

知道readme.txt的具体修改内容后，就可以放心的向仓库提交，提交修改和提交新文件是一样的两步。

此时，运行git status查看当前仓库的状态。

$ *git status*

*On branch master*

*Changes to be committed:*

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

*modified: readme.txt*

git status显示将要被提交的修改包括readme.txt，然后提交此次修改。

$ *git commit -m "add distributed"*

*[master e475afc] 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 tree clean*

Git显示当前没有需要提交的修改，而且工作目录干净（working tree clean）。

## 4.1 版本回退

为了更好的演示各版本之间的转换，对readme.txt文件再进行一次修改，修改如下：

*Git is a distributed version control system.*

*Git is free software distributed under the GPL.*

然后提交。

$ *git add readme.txt*

$ *git commit -m "append GPL"*

*[master 1094adb] append GPL*

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

像这样，不断对文件进行修改，然后不断提交修改到版本库，就像玩RPG游戏时，每通过一关就会自动存盘游戏状态，如果某关没通过，还可以选择读取前一关状态。有时在打Boss之前，会手动存盘，如果打Boss失败，可以从最近的地方重新开始。Git也一样，如果觉得文件修改到一定程度时，就可以“保存一个快照”，这个快照在Git中被称为commit。如果不慎改乱文件，或者误删文件，还可以从最近的一个commit恢复，然后继续工作，不会丢失长时间的工作成果。

目前，一共有3个版本的readme.txt文件被提交到Git仓库里。

版本1：*wrote a readme file*

*Git is a version control system.*

*Git is free software.*

版本2：*add distributed*

*Git is a distributed version control system.*

*Git is free software.*

版本3：*append GPL*

*Git is a distributed version control system.*

*Git is free software distributed under the GPL.*

当然，实际工作中，自己无法准确记忆一个文件每次的修改内容，而版本控制系统可以有历史记录，在Git中，用git log命令查看。

$ *git log*

*commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (HEAD -> master)*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 21:06:15 2018 +0800*

*append GPL*

*commit e475afc93c209a690c39c13a46716e8fa000c366*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 21:03:36 2018 +0800*

*add distributed*

*commit eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 20:59:18 2018 +0800*

*wrote a readme file*

git log命令显示从最近到最远的提交日志，可以看到3次提交，最近的一次是append GPL，上一次是add distributed，最早的一次是wrote a readme file。

如果觉得输出信息太多，显示较乱，可以加上--pretty=oneline参数。

$ *git log --pretty=oneline*

*1094adb7b9b3807259d8cb349e7df1d4d6477073 (HEAD -> master) append GPL*

*e475afc93c209a690c39c13a46716e8fa000c366 add distributed*

*eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0 wrote a readme file*

这里显示的一大串类似1094adb...的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是SHA1计算出来的一个非常大的数字，用十六进制表示。commit id用一大串数字表示的原因是， Git是分布式的版本控制系统，很多人员都在同一个版本库里工作，如果大家都用1，2，3……作为版本号，会产生冲突。工作中，每提交一个新版本，Git就会自动把它们串成一条时间线。

现在可以启动时光穿梭机，准备把readme.txt文件回退到上一个版本，也就是add distributed的那个版本。首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交1094adb...（注意：这里的提交ID需要以自己的为准），上一个版本就是HEAD^，上上一个版本就是HEAD^^，但如果需要回退到较早的版本，^太多不易输入，所以写成HEAD~100。

现在，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 reset --hard HEAD^*  回退到上一版本。

*HEAD is now at e475afc add distributed*

然后可以使用cat命令查看readme.txt文件的内容。

$ *cat readme.txt*

*Git is a distributed version control system.*

*Git is free software.*

可以看到文件被还原至上一个版本。

当然，还可以继续回退到上一个版本wrote a readme file。这里，先用git log查看现在版本库的状态。

$ *git log*

*commit e475afc93c209a690c39c13a46716e8fa000c366 (HEAD -> master)*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 21:03:36 2018 +0800*

*add distributed*

*commit eaadf4e385e865d25c48e7ca9c8395c3f7dfaef0*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 20:59:18 2018 +0800*

*wrote a readme file*

最新的版本append GPL已经看不到了，如果还想再回到这个版本，命令窗口还未关闭时，仍然可以往上找，找到那个append GPL版本的commit id是1094adb...，于是就可以指定回到未来的某个版本。

$ *git reset --hard 1094a*  指定回到某个版本，id输入前几位即可。

*HEAD is now at 83b0afe append GPL*

此时，再查看readme.txt文件的内容。

$ *cat readme.txt*

*Git is a distributed version control system.*

*Git is free software distributed under the GPL.*

可以看到，append GPL版本又恢复了。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，回退版本的时候，Git仅仅是把HEAD从指向append GPL，改为指向add distributed（如图4-1所示），然后更新工作区的文件。

 

(a)指向append GPL (b)指向add distributed

图4-1 HEAD指针位置

所以希望HEAD指向哪个版本号，就把当前版本定位在哪。

如果已经完成了文件版本回退，关机后又想恢复到最新版本，可以通过git reflog命令查找各版本的commit id，该命令用于记录每一次命令。

$ *git reflog*

*e475afc HEAD@{1}: reset: moving to HEAD^*

*1094adb (HEAD -> master) HEAD@{2}: commit: append GPL*

*e475afc HEAD@{3}: commit: add distributed*

*eaadf4e HEAD@{4}: commit (initial): wrote a readme file*

结果显示append GPL的commit id是1094adb，现在，又可以乘坐时光机回到未来的版本。

## 4.2 工作区和暂存区

Git和其他版本控制系统（如SVN）的一个不同之处就是有暂存区的概念。

**工作区**（Working Directory）

就是能在电脑里看到的目录，比如learngit文件夹就是一个工作区。

**版本库**（Repository）

工作区中有一个隐藏目录.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自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



图4-2 工作区和版本库

前面讲到给Git版本库里添加文件时，是分两步执行的：第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为创建Git版本库时，Git自动创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

为了更好的演示，再对readme.txt做个修改，比如加上一行内容如下：

*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。用git status查看状态。

$ *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非常清楚地显示，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status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，暂存区状态如图4-3所示。



图4-3 工作区更改后的暂存区(stage)状态

所以，git add命令实际上就是把需要提交的所有修改放到暂存区(stage)，然后，执行git commit就可以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 e43a48b] understand how stage works*

*2 files changed, 2 insertions(+)*

*create mode 100644 LICENSE*

提交后，如果没有再对工作区做任何修改，那么工作区就是“干净”的。此时版本库如图4-4所示，暂存区没有任何内容。

$ *git status*

*On branch master*

*nothing to commit, working tree clean*



图4-4 提交后的暂存区(stage)状态

## 4.3 管理修改

掌握了暂存区概念后。讨论为什么Git相比于其他版本控制系统的设计更加优秀，因为Git跟踪并管理的是修改，而非文件。比如新增了一行，就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

下面通过一个例子说明为什么Git管理的是修改，而不是文件呢。第一步，对readme.txt做一个修改，比如加一行内容如下：

$ *cat readme.txt*

*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.*

然后添加。

$ *git add readme.txt*

$ *git status*

*On branch master*

*Changes to be committed:*

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

*modified: readme.txt*

然后，再修改readme.txt如下：

$ *cat readme.txt*

*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.*

再提交。

$ *git commit -m "git tracks changes"*

*[master 519219b] git tracks changes*

*1 file changed, 1 insertion(+)*

提交后，再查看状态。

$ *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")*

这里第二次的修改没有被提交。回顾操作过程：第一次修改→git add→第二次修改→git commit。

所以说，Git管理的是修改，当使用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用git diff HEAD命令可以查看工作区和版本库里最新版本的区别。

$ *git diff HEAD -- readme.txt*

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

*index 76d770f..a9c5755 100644*

*--- a/readme.txt*

*+++ b/readme.txt*

*@@ -1,4 +1,4 @@*

*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.*

*+Git tracks changes of files.*

可以看到，第二次的修改确实没有提交。要提交第二次修改可以继续git add再git commit，也可以先不提交第一次修改，先git add第二次修改，再git commit，相当于把两次修改合并后一起提交。

## 4.4 撤销修改

一般情况下很少做出错误修改，如果对文件做了错误修改，比如在readme.txt文件中加了一行内容如下：

$ *cat readme.txt*

*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.*

*My stupid boss still prefers SVN.*

如果提交前及时发现stupid boss这个错误，可以删掉最后一行，手动恢复文件到上一个版本状态。也可以通过Git进行修改，先用git status查看当前状态。

$ *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")*

可以看到，Git提示git checkout -- file可以丢弃工作区的修改。

$ *git checkout -- readme.txt*

命令git checkout -- readme.txt就是把readme.txt文件在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，此时，撤销修改就回到和版本库一模一样的状态；另一种是readme.txt添加到暂存区后，又作了修改，此时，撤销修改就是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“--”，就变成“切换到另一个分支”的命令。

此时，查看readme.txt的文件内容，可以看到文件内容已经复原。

$ *cat readme.txt*

*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.*

如果工作区的错误已经git add至暂存区。

$ *cat readme.txt*

*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.*

*My stupid boss still prefers SVN.*

$ *git add readme.txt*

但在commit之前，发现了这个错误。用git status查看一下，修改只是添加到了暂存区，还没有提交。

$ *git status*

*On branch master*

*Changes to be committed:*

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

*modified: readme.txt*

同样地，Git显示，用git reset HEAD <file>命令可以把暂存区的修改撤销掉（unstage），重新放回工作区。

$ *git reset 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。使用HEAD时，表示最新的版本。

再用git status查看状态，现在暂存区是干净的，工作区有修改。

$ *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*

然后丢弃工作区的修改，再查看状态。

$ *git checkout -- readme.txt*

$ *git status*

*On branch master*

*nothing to commit, working tree clean*

如果不仅对文件做了错误修改，还从暂存区提交到了版本库，可以通过版本回退的命令将文件回退至上一个版本，但前提是还未将有错误修改的本地版本库推送至远程。

## 4.5 删除文件

在Git中，删除也是一个修改操作。这里先添加一个新文件test.txt到Git并且提交。

$ *git add test.txt*

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

*[master b84166e] add test.txt*

*1 file changed, 1 insertion(+)*

*create mode 100644 test.txt*

一般情况下，可以直接在文件管理器中删除文件，或者用rm命令删除工作区文件。

$ *rm test.txt*

删除后，工作区和版本库就不一致，git status命令可以显示哪些文件被删除。

$ *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")*

工作区文件删除后，如果确实需要删除，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 d46f35e] remove test.txt*

*1 file changed, 1 deletion(-)*

*delete mode 100644 test.txt*

此时，文件从版本库中删除。

如果是误删，就用git checkout命令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，所以无论工作区是修改还是删除，都可以“一键还原”。

# 远程仓库

目前为止，已经介绍了如何在Git仓库里对一个文件进行时光穿梭，不用担心文件备份或者丢失的问题，但这些功能在集中式版本控制系统SVN里已经有了，这里开始介绍Git的其他强大功能之一：远程仓库。

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。最早，只有一台机器有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

这里注册一个GitHub账号，就可以免费获得Git远程仓库。注册网址为<https://github.com/>。由于本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要进行设置。

第1步：创建SSH Key。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有id\_rsa和id\_rsa.pub这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key。

$ *ssh-keygen -t rsa -C “youremail@example.com”*

这里，需要把邮箱地址换成自己的，然后一路回车，使用默认值即可，由于这个Key不是用于军事目的，所以无需设置密码。然后可以在用户主目录里找到.ssh目录，里面有id\_rsa和id\_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id\_rsa是私钥，不能泄露出去，id\_rsa.pub是公钥，可以放心地告诉任何人。

第2步：登陆GitHub，打开“Account settings”，“SSH Keys”页面。

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id\_rsa.pub文件的内容，点击“Add Key”，就能够看到已经添加的Key。

GitHub需要SSH Key是因为GitHub需要识别出推送的提交确实是本人推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道公钥，就可以确认只有自己才能推送。当然，GitHub允许添加多个Key。假定有若干电脑，可能在公司提交，也可能在家里提交，只需要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送。

提示：GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有自己可以修改）。如果不希望别人看到自己的Git库，有两个办法，一个是交费，让GitHub把公开的仓库变成私有的。另一个办法是自己搭建一个Git服务器，由于是自己的Git服务器，所以别人看不到。

## 5.1 添加远程库

现在，已经在本地创建了一个Git仓库，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库。在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库。

目前，在GitHub上的这个learngit仓库还是空的，GitHub显示，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

然后，根据GitHub的提示，在本地的learngit仓库下运行命令。

$ *git remote add origin git@github.com:****Your GitHub Name****/learngit.git*

添加后，远程库的名字就是origin，这是Git默认的叫法。

下一步，推送本地库的所有内容至远程库。

$ *git push -u origin master*

*Counting objects: 20, done.*

*Delta compression using up to 4 threads.*

*Compressing objects: 100% (15/15), done.*

*Writing objects: 100% (20/20), 1.64 KiB | 560.00 KiB/s, done.*

*Total 20 (delta 5), reused 0 (delta 0)*

*remote: Resolving deltas: 100% (5/5), done.*

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

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

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

把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。由于此时远程库是空的，第一次推送master分支时，加上了-u参数，Git不仅会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，以后的推送或拉取就可以简化命令。推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一样。随后，只要本地库中有更新，就可以通过git push命令把本地master分支的最新修改推送至GitHub，也就有了真正的分布式版本库。

$ *git push origin master*

**SSH警告**

第一次使用Git的clone或push命令连接GitHub时，可能会得到一个警告。

*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使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要确认GitHub的Key的指纹信息是否真的来自GitHub的服务器，输入yes回车即可。

Git会输出警告，表示已经把GitHub的Key添加到本机的一个信任列表里。

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

这个警告只会出现一次，后面的操作就不会有任何警告了。

## 5.2 从远程库克隆

现在，假设从零开发，那么最好的方式是先创建远程库，然后从远程库克隆。首先，登陆GitHub，创建一个新的仓库，名字叫gitskills。勾选Initialize this repository with a README，这样GitHub会自动创建一个README.md文件。创建完毕后，可以看到该文件。

远程库准备完成后，用git clone命令克隆一个本地库。

$ *git clone git@github.com:****Your GitHub Name****/gitskills.git*

*Cloning into 'gitskills'...*

*remote: Counting objects: 3, done.*

*remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 3*

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

然后进入gitskills目录查看，已经存在README.md文件。

$ *cd gitskills*

$ *ls*

*README.md*

如果多人协作开发某个项目，可以每人各自从远程库克隆一份。

# 分支管理

分支与科幻电影里的平行宇宙类似，假设自己在学习Git的时候，另一个自己在另一个平行宇宙里学习SVN。若两个平行宇宙互不干扰，那么两个人也没有交集。若两个平行宇宙在某个时间点合并，结果就是既学会了Git又学会了SVN。

分支的实际应用主要在于某项工作周期较长，提前提交会影响其他人，又存在已完成内容丢失风险的时候，可以创建一个分支，别人看不到，不影响他们在原来分支上的正常工作，而自己可以在分支上继续工作，可以在任意时间提交，直至工作完成后，再一次性合并到原来的分支上，这样，既安全，又不会影响其他人的工作。

其他版本控制系统如SVN等都有分支管理，但问题是创建、切换分支的速度较慢。但是，Git的分支无论创建、切换或删除，基本都会在1s内完成，不受版本库里的文件数量影响。

## 6.1 创建与合并分支

在版本回退里提到，每次提交，Git都会把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master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指向的就是当前分支。

最初的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支上的提交点，如图6-1所示。每次提交，master分支都会向前移动一步，随着不断地提交，master分支的线也会越来越长。



图 6-1 当前分支及提交点

当创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上，如图6-2所示。



图6-2 新建分支

这里可以看到，Git创建一个分支非常迅速，因为除了增加一个dev指针，改变HEAD的指向，工作区的文件没有任何变化。

此时，对工作区的修改和提交就针对dev分支进行，例如新提交一次后，dev指针往前移动一步，而master指针不变，如图6-3所示。



图6-3 针对dev分支新增提交

如果完成了在dev上的工作，就可以把dev合并到master上。Git合并最简单的方法，就是直接把master指向dev的当前提交，完成合并，如图6-4所示。



图6-4 合并分支

所以Git合并分支也很迅速，仅改变指针，工作区内容保持不变。

分支合并完成后，就可以删除dev分支。删除dev分支实际就是删除dev指针，删除后，就只有一条master分支，如图6-5所示。



图6-5 删除dev分支

下面介绍上述操作的命令语句。

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，可以用git branch命令查看当前分支。

$ *git branch*

*\* dev*

*master*

git branch命令会列出所有分支，当前分支前面会标一个\*号。

然后，可在dev分支上正常提交。比如对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 b17d20e] branch test*

*1 file changed, 1 insertion(+)*

dev分支工作完成后，就可以切换回master分支。

$ *git checkout master*

*Switched to branch 'master'*

切换回master分支后，再查看readme.txt文件，发现并没有刚才添加的内容。这是因为提交是在dev分支上完成的，而master分支此刻的提交点还没有变，如图6-6所示。



图6-6 切换分支

所以，需要把dev分支的工作成果合并到master分支上。git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt内容，就可以看到当前分支与被合并分支的最新提交完全一样。

$ *git merge dev*

*Updating d46f35e..b17d20e*

*Fast-forward*

*readme.txt | 1 +*

*1 file changed, 1 insertion(+)*

上面的Fast-forward信息表示，Git这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

合并完成后，可以删除dev分支。

$ *git branch -d dev*

*Deleted branch dev (was b17d20e).*

删除后，查看branch，就只剩master分支。

$ *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，合并后再删除分支，这和直接在master分支上工作效果一样，但过程更安全。

## 6.2 解决冲突

为演示合并分支过程中可能出现的问题，创建新的feature1分支。

$ *git checkout -b feature1*

*Switched to a new branch 'feature1'*

修改readme.txt文件的最后一行为：

*Creating a new branch is quick AND simple.*

然后，在feature1分支上提交。

$ *git add readme.txt*

$ *git commit -m "AND simple"*

*[feature1 14096d0] AND simple*

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

再切换至master分支。

$ *git checkout master*

*Switched to branch 'master'*

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

*(use "git push" to publish your local commits)*

Git会自动提示，当前master分支比远程的master分支超前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.*

然后提交。

$ *git add readme.txt*

$ *git commit -m "& simple"*

*[master 5dc6824] & 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，如图6-7所示。



图6-7 两个分支分别新增提交

这种情况下，Git无法执行“快速合并”，只能试图把各自的修改进行合并，但这种合并就可能会产生冲突，下面尝试合并。

$ *git merge feature1*

*Auto-merging readme.txt*

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

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

结果发生冲突，Git显示，readme.txt文件存在冲突，必须手动解决冲突后再提交。git status可以显示冲突的文件。

$ *git status*

*On branch master*

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

*(use "git push" to publish your local commits)*

*You have unmerged paths.*

*(fix conflicts and run "git commit")*

*(use "git merge --abort" to abort the merge)*

*Unmerged paths:*

*(use "git add <file>..." to mark resolution)*

*both modified: readme.txt*

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

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会用<<<<<<<，=======，>>>>>>>标记不同分支内容，修改如下保存。

*Creating a new branch is quick and simple.*

然后再提交。

$ *git add readme.txt*

$ *git commit -m "conflict fixed"*

*[master cf810e4] conflict fixed*

此时，master分支和feature1分支情况如图6-8所示。



图6-8 手动解决冲突并提交

用带参数的git log可以查看分支的合并情况。

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

*\* cf810e4 (HEAD -> master) conflict fixed*

*|\*

*| \* 14096d0 (feature1) AND simple*

*\* | 5dc6824 & simple*

*|/*

*\* b17d20e branch test*

*\* d46f35e (origin/master) remove test.txt*

*\* b84166e add test.txt*

*\* 519219b git tracks changes*

*\* e43a48b understand how stage works*

*\* 1094adb append GPL*

*\* e475afc add distributed*

*\* eaadf4e wrote a readme file*

最后，删除feature1分支。

$ *git branch -d feature1*

*Deleted branch feature1 (was 14096d0).*

## 6.3 分支管理策略

通常合并分支时， Git会用Fast forward模式，但该模式下删除分支后，会丢掉分支信息。如果强制禁用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。下面详细说明如何操作。

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，并提交一个新的commit，然后再切换回master。

$ *git add readme.txt*

$ *git commit -m "add merge"*

*[dev f52c633] add merge*

*1 file changed, 1 insertion(+)*

$ *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。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。

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

*Merge made by the 'recursive' strategy.*

*readme.txt | 1 +*

*1 file changed, 1 insertion(+)*

合并后，用git log查看分支历史。

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

*\* e1e9c68 (HEAD -> master) merge with no-ff*

*|\*

*| \* f52c633 (dev) add merge*

*|/*

*\* cf810e4 conflict fixed*

*...*

可以看到，不使用Fast forward模式，merge后状态如图6-9所示。



图6-9 不使用Fast forward模式合并

合并分支时，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

## 6.4 Bug分支

如果工作过程中，在某分支进行一项预计1天完成的工作时，又接到另一项需要2小时内完成的任务，比如有一个Bug需要马上修复，可以创建一个新的分支来进行新任务。但是原来进行的工作还没有完成，无法提交，就可以用git stash命令把当前工作现场“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: f52c633 add merge*

“储藏”完成后，用git status查看工作区就是干净的（前提是没有未被Git管理的文件）。然后就可以创建分支来修复Bug。

首先确定要在哪个分支上修复bug，假设需要在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.*

*(use "git push" to publish your local commits)*

$ *git checkout -b issue-101*

*Switched to a new branch 'issue-101'*

然后修复bug，把*Git is free software ...*改为*Git is a free software ...*，再提交。

$ *git add readme.txt*

$ *git commit -m "fix bug 101"*

*[issue-101 4c805e2] fix bug 101*

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

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

$ *git checkout master*

*Switched to branch 'master'*

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

*(use "git push" to publish your local 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(-)*

bug修复完成后，就可以回到dev分支继续工作。

$ *git checkout dev*

*Switched to branch 'dev'*

$ *git status*

*On branch dev*

*nothing to commit, working tree clean*

这里显示工作区是干净的，这时，可以用git stash list命令查看工作区。

$ *git stash list*

*stash@{0}: WIP on dev: f52c633 add merge*

工作现场还在，Git把stash内容存储在某个地方，恢复工作现场有两个办法：一是用git stash apply恢复，但是恢复后不删除stash内容，需要用git stash drop删除；另一种方式是用git stash pop，恢复的同时删除stash内容。

$ *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} (5d677e2ee266f39ea296182fb2354265b91b3b2a)*

再用git stash list查看，就看不到任何stash内容。

$ *git stash list*

所以，可以多次stash，恢复时先用git stash list查看，然后恢复指定的stash。

$ *git stash apply stash@{0}*

## 6.5 Feature分支

软件开发中，总是需要不断添加新功能。添加新功能时，为保证主分支工作内容不受影响，可以每添加一个新功能，就新建一个feature分支，在分支上开发，开发完成后合并分支，最后删除该feature分支。

比如现在有一个开发代号为Vulcan的新功能，于是准备开发。

$ *git checkout -b feature-vulcan*

*Switched to a new branch 'feature-vulcan'*

开发完毕后提交。

$ *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 287773e] add feature vulcan*

*1 file changed, 2 insertions(+)*

*create mode 100644 vulcan.py*

然后切回dev，准备合并。

$ *git checkout dev*

如果顺利，feature分支和bug分支是类似的，合并，然后删除。但是如果某些原因导致需要取消该新功能，并且需要就地销毁这个包含机密资料的分支，就可以删除该分支。

$ *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'.*

但结果显示删除失败。Git提示feature-vulcan分支还没有被合并，如果删除，将丢失修改，如果要强行删除，需要使用大写的-D参数。然后进行强行删除。

$ *git branch -D feature-vulcan*

*Deleted branch feature-vulcan (was 287773e).*

结果显示删除成功。

## 6.6 多人协作

当从远程仓库克隆时，Git会自动把本地的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。可以用git remote命令查看远程库的信息。

$ *git remote*

*origin*

或者，用git remote –v命令显示更详细的信息。

$ *git remote -v*

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

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

结果显示可以抓取和推送的origin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，需要指定本地分支，Git会把该分支推送到远程库对应的远程分支上。

$ *git push origin master*

如果要推送其他分支，比如dev，可以更改名称。

$ *git push origin dev*

但是，并不是一定要把本地分支往远程推送。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，不必要推到远程；feature分支是否推到远程，取决于是否和同事合作在上面开发。总之在Git中，分支完全可以只在本地，是否推送，视情况而定。

**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。

现在，模拟一个同事在另一台电脑（注意要把SSH Key添加到GitHub）或者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: 40, done.*

*remote: Compressing objects: 100% (21/21), done.*

*remote: Total 40 (delta 14), reused 40 (delta 14), pack-reused 0*

*Receiving objects: 100% (40/40), done.*

*Resolving deltas: 100% (14/14), done.*

当从远程库clone时，默认情况下，同事只能看到本地的master分支。可以用git branch命令查看。

$ *git branch*

*\* master*

现在，如果要在dev分支上开发，就必须创建远程origin的dev分支到本地。

$ *git checkout -b dev origin/dev*

然后就可以在dev上继续修改，需要时再把dev分支push到远程。

$ *git add env.txt*

$ *git commit -m "add env"*

*[dev 7a5e5dd] add env*

*1 file changed, 1 insertion(+)*

*create mode 100644 env.txt*

$ *git push origin dev*

*Counting objects: 3, done.*

*Delta compression using up to 4 threads.*

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

*Writing objects: 100% (3/3), 308 bytes | 308.00 KiB/s, done.*

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

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

*f52c633..7a5e5dd dev -> dev*

如果同事已经向origin/dev分支推送了他的提交，而同时自己也对同样的文件作了修改，并试图推送。

$ *cat env.txt*

*env*

$ *git add env.txt*

$ *git commit -m "add new env"*

*[dev 7bd91f1] add new env*

*1 file changed, 1 insertion(+)*

*create mode 100644 env.txt*

$ *git push origin dev*

*To 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. Integrate the remote changes (e.g.*

*hint: 'git pull ...') before pushing again.*

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

结果显示推送失败，这是因为同事的最新提交和自己试图推送的提交发生冲突。这里，Git已经提示如何解决冲突，先用git pull把最新的提交从origin/dev上抓取下来，然后，在本地合并，解决冲突，再推送。

$ *git pull*

*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-to=origin/<branch> dev*

git pull也显示失败，是因为没有指定本地dev分支与远程origin/dev分支的链接，根据提示，设置dev和origin/dev的链接，然后再进行抓取。

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

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

$ *git pull*

*Auto-merging env.txt*

*CONFLICT (add/add): Merge conflict in env.txt*

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

这次git pull成功，但合并存在冲突，需要手动解决，解决方法与分支管理中解决冲突的方法完全一致。解决后，提交，再进行推送。

$ *git commit -m "fix env conflict"*

*[dev 57c53ab] fix env conflict*

$ *git push origin dev*

*Counting objects: 6, done.*

*Delta compression using up to 4 threads.*

*Compressing objects: 100% (4/4), done.*

*Writing objects: 100% (6/6), 621 bytes | 621.00 KiB/s, done.*

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

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

*7a5e5dd..57c53ab dev -> dev*

总的来说，多人协作的工作模式通常是这样的。首先，可以试图用git push origin <branch-name>推送自己的修改；若推送失败，则因为远程分支比自己的本地更新，需要先用git pull试图合并；若合并存在冲突，则解决冲突，并在本地提交；没有冲突或者解决冲突后，再用git push origin <branch-name>推送即可。

若git pull提示no tracking information，则说明本地分支和远程分支未创建链接关系，可以用git branch --set-upstream-to <branch-name> origin/<branch-name>命令进行创建。

## 6.7 Rebase

在上一节可以看到，多人在同一个分支上协作时，很容易出现冲突。即使没有冲突，后push的必须先pull，在本地合并，然后才能push成功。

每次合并再push后，分支都会发生改变。

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

*\* d1be385 (HEAD -> master, origin/master) init hello*

*\* e5e69f1 Merge branch 'dev'*

*|\*

*| \* 57c53ab (origin/dev, dev) fix env conflict*

*| |\*

*| | \* 7a5e5dd add env*

*| \* | 7bd91f1 add new env*

*| |/*

*\* | 12a631b merged bug fix 101*

*|\ \*

*| \* | 4c805e2 fix bug 101*

*|/ /*

*\* | e1e9c68 merge with no-ff*

*|\ \*

*| |/*

*| \* f52c633 add merge*

*|/*

*\* cf810e4 conflict fixed*

看上去十分乱，如果想让Git的提交历史是一条干净的直线，可以通过rebase操作，下面举例说明。

和远程分支同步后，对hello.py这个文件做了两次提交。用git log命令查看。

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

*\* 582d922 (HEAD -> master) add author*

*\* 8875536 add comment*

*\* d1be385 (origin/master) init hello*

*\* e5e69f1 Merge branch 'dev'*

*|\*

*| \* 57c53ab (origin/dev, dev) fix env conflict*

*| |\*

*| | \* 7a5e5dd add env*

*| \* | 7bd91f1 add new env*

*...*

注意到Git用(HEAD -> master)和(origin/master)标识出当前分支的HEAD和远程origin的位置分别是582d922 add author和d1be385 init hello，本地分支比远程分支超前两个提交。

现在尝试推送本地分支。

$ *git push origin master*

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

*! [rejected] master -> master (fetch first)*

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

*hint: Updates were rejected because the remote contains work that you do*

*hint: not have locally. This is usually caused by another repository pushing*

*hint: to the same ref. You may want to first integrate the remote changes*

*hint: (e.g., 'git pull ...') before pushing again.*

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

结果显示失败，这说明有人先推送了远程分支。按照经验，先pull一下。

$ *git pull*

*remote: Counting objects: 3, done.*

*remote: Compressing objects: 100% (1/1), done.*

*remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0*

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

*From github.com:michaelliao/learngit*

*d1be385..f005ed4 master -> origin/master*

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

*Auto-merging hello.py*

*Merge made by the 'recursive' strategy.*

*hello.py | 1 +*

*1 file changed, 1 insertion(+)*

再用git status查看状态。

$ *git status*

*On branch master*

*Your branch is ahead of 'origin/master' by 3 commits.*

*(use "git push" to publish your local commits)*

*nothing to commit, working tree clean*

加上刚才合并的提交，本地分支比远程分支超前3个提交。用git log查看。

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

*\* e0ea545 (HEAD -> master) Merge branch 'master' of github.com:michaelliao/learngit*

*|\*

*| \* f005ed4 (origin/master) set exit=1*

*\* | 582d922 add author*

*\* | 8875536 add comment*

*|/*

*\* d1be385 init hello*

*...*

结果显示提交历史分叉。如果现在把本地分支push到远程，会产生问题。这时，可以使用git rebase命令。

$ *git rebase*

*First, rewinding head to replay your work on top of it...*

*Applying: add comment*

*Using index info to reconstruct a base tree...*

*M hello.py*

*Falling back to patching base and 3-way merge...*

*Auto-merging hello.py*

*Applying: add author*

*Using index info to reconstruct a base tree...*

*M hello.py*

*Falling back to patching base and 3-way merge...*

*Auto-merging hello.py*

再用git log查看。

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

*\* 7e61ed4 (HEAD -> master) add author*

*\* 3611cfe add comment*

*\* f005ed4 (origin/master) set exit=1*

*\* d1be385 init hello*

*...*

原本分叉的提交现在变成一条直线，其实原理非常简单。观察发现Git把本地的提交“挪动”了位置，放到了f005ed4 (origin/master) set exit=1之后，这样，整个提交历史就成了一条直线。rebase操作前后，最终的提交内容是一致的，但是，本地的commit修改内容已经变化了，它们的修改不再基于d1be385 init hello，而是基于f005ed4 (origin/master) set exit=1，但最后的提交7e61ed4内容是一致的。

这就是rebase操作的特点：把分叉的提交历史“整理”成一条直线，看上去更直观。缺点是本地的分叉提交会被修改。

最后，通过push操作把本地分支推送到远程。

$ *git push origin master*

*Mac:~/learngit michael$ git push origin master*

*Counting objects: 6, done.*

*Delta compression using up to 4 threads.*

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

*Writing objects: 100% (6/6), 576 bytes | 576.00 KiB/s, done.*

*Total 6 (delta 2), reused 0 (delta 0)*

*remote: Resolving deltas: 100% (2/2), completed with 1 local object.*

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

*f005ed4..7e61ed4 master -> master*

再用git log查看效果。

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

*\* 7e61ed4 (HEAD -> master, origin/master) add author*

*\* 3611cfe add comment*

*\* f005ed4 set exit=1*

*\* d1be385 init hello*

*...*

可以看到，远程分支的提交历史也是一条直线。

# 标签管理

由于版本库中指向某个commit的指针号码较乱，不易记忆管理，所以需要对commit id打上标签，与某个commit绑定，方便查找。所以，tag就是一个让人容易记住的有意义的名字，跟某个commit绑定。发布一个版本时，通常会先在版本库中打一个标签（tag），这样，就可以唯一确定打标签时刻的版本。将来无论什么时候，取某个标签的版本，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。

## 7.1 创建标签

Git中打标签非常简单，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， 可以用git tag <name>命令打一个新标签。

$ *git tag v1.0*

可以用命令git tag查看所有标签。

$ *git tag*

*v1.0*

标签会默认打在最新提交的commit上。如果忘记在之前应该打标签的地方打上标签，可以先找到历史提交的commit id，然后打上标签。

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

*12a631b (HEAD -> master, tag: v1.0, origin/master) merged bug fix 101*

*4c805e2 fix bug 101*

*e1e9c68 merge with no-ff*

*f52c633 add merge*

*cf810e4 conflict fixed*

*5dc6824 & simple*

*14096d0 AND simple*

*b17d20e branch test*

*d46f35e remove test.txt*

*b84166e add test.txt*

*519219b git tracks changes*

*e43a48b understand how stage works*

*1094adb append GPL*

*e475afc add distributed*

*eaadf4e wrote a readme file*

如果需要对add merge这次提交打标签，可以用它对应的commit id—— f52c633。

$ *git tag v0.9 f52c633*

再用命令git tag查看标签。

$ *git tag*

*v0.9*

*v1.0*

注意：标签按字母排序，不是时间。git show <tagname>可以查看标签信息。

$ *git show v0.9*

*commit f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 21:56:54 2018 +0800*

*add merge*

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

*...*

可以看到，v0.9确实打在add merge这次提交上。

另外，还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字。

$ *git tag -a v0.1 -m "version 0.1 released" 1094adb*

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: Fri May 18 22:48:43 2018 +0800*

*version 0.1 released*

*commit 1094adb7b9b3807259d8cb349e7df1d4d6477073 (tag: v0.1)*

*Author: Michael Liao <askxuefeng@gmail.com>*

*Date: Fri May 18 21:06:15 2018 +0800*

*append GPL*

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

*...*

注意：标签总是和某个commit挂钩。如果这个commit既出现在master分支，又出现在dev分支，那么在这两个分支上都可以看到这个标签。

## 7.2 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 f15b0dd)*

因为创建的标签暂时都只存储在本地，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。

如果需要推送某个标签到远程，使用git push origin <tagname>命令。

$ *git push origin v1.0*

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

*To 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*

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

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

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

若标签已推送到远程，要删除远程标签就需要先从本地删除，再从远程删除。

$ *git tag -d v0.9*

*Deleted tag 'v0.9' (was f52c633)*

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

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

*- [deleted] v0.9*

如果需要确认是否真的完成远程库标签的删除，可以登陆GitHub查看。

# 自定义Git

在安装Git一节中，已经配置了user.name和user.email，实际上，Git还有很多可配置项。

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*

这时，Git会适当显示不同颜色，执行git status，可以看到文件名标上颜色。



图8-1 文件名显示颜色

## 8.1 忽略特殊文件

有些时候，必须把某些文件放到Git工作目录中，但又不能提交它们，比如保存了数据库密码的配置文件等，每次git status都会显示Untracked files ...。解决方法是在Git工作区的根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。

这里不需要从头写.gitignore文件，GitHub已经准备了各种配置文件，只需组合使用。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，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。

下面举例说明，如果在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，因此需要忽略Windows自动生成的垃圾文件。

*# Windows:*

*Thumbs.db*

*ehthumbs.db*

*Desktop.ini*

然后，继续忽略Python编译产生的.pyc、.pyo、dist等文件或目录。

*# Python:*

*\*.py[cod]*

*\*.so*

*\*.egg*

*\*.egg-info*

*dist*

*build*

加上自己定义的文件，最终得到一个完整的.gitignore文件。

*# 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。当然检验.gitignore的标准是执行git status命令后，是否显示working directory clean。

Windows系统下，如果在资源管理器里新建一个.gitignore文件，它会提示必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。有些时候，想添加一个文件到Git，但添加不了的原因是这个文件被.gitignore忽略了。

$ *git add App.class*

*The following paths are ignored by one of your .gitignore files:*

*App.class*

*Use -f if you really want to add them.*

如果确实需要添加该文件，可以用-f强制添加到Git。

$ *git add -f App.class*

或者发现，可能是.gitignore写得有问题，需要找出写错的规则，可以用git check-ignore命令检查。

$ *git check-ignore -v App.class*

*.gitignore:3:\*.class App.class*

Git会显示，.gitignore的第3行规则忽略了该文件，于是就可以知道应该修订哪个规则。

## 8.2 配置别名

由于部分命令中单词过长，容易出错，所以最好可以针对某些较长命令存在简写，方便操作，提高效率。

$ *git config --global alias.<abbreviation> <original word>*

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

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

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

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

以上四行命令表示用st就表示status，co表示checkout，ci表示commit，br表示branch。以后提交就可以简写。

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

撤销修改一节中，git reset HEAD file命令可以撤销（unstage）暂存区的修改，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'*

这时，键入git unstage test.py命令，实际上Git执行的是git reset HEAD test.py。

再配置一个git last，让其显示最后一次的提交信息：

$ *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*

还可以如下配置lg。

$ *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"*

git lg命令的效果如图8-2所示。



图8-2 git lg命令的效果

**配置文件**

配置Git的时候，加上—global的作用针对当前用户，如果不加，那作用只针对当前仓库。

每个仓库的Git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，如果要删除别名，可以直接删除对应行。

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。