# 学习笔记1——Git简介

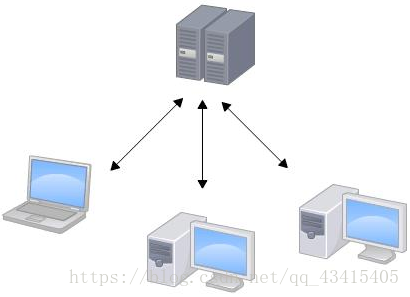
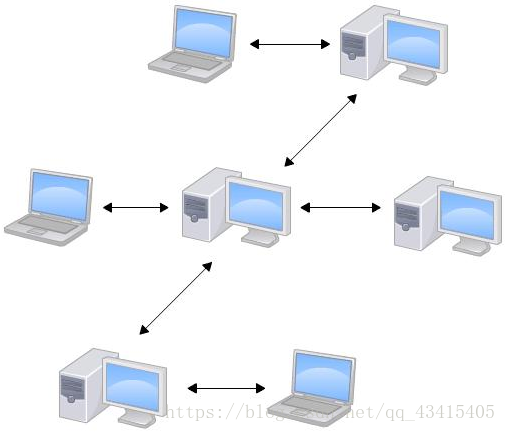
# Git是什么？

Git是目前世界上最先进的**分布式版本控制系统**（没有之一）。可以有效、高速的处理从很小到非常大的**项目版本管理**。

## 什么是版本控制系统？

举个例子，如果有一个软件，不但能自动帮你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，只需要在软件里瞄一眼就可以。这个软件用起来就应该像这个样子，能记录每次文件的改动：  
于是你可以从手动管理多个“版本”里解放出来，这就是版本控制。

## 集中式vs分布式

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，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，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。  
  
集中式版本控制系统**最大的毛病**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，这还不得把人给憋死啊。  
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。  
  
当然，Git的优势不单是不必联网这么简单，后面我们还会看到Git极其强大的**分支管理**，把SVN等远远抛在了后面。

# Git的特点

分布式相比于集中式的最大区别在于开发者可以提交到本地，每个开发者通过克隆（**git clone**），在本地机器上**拷贝一个完整的Git仓库**。

## Git的功能特性

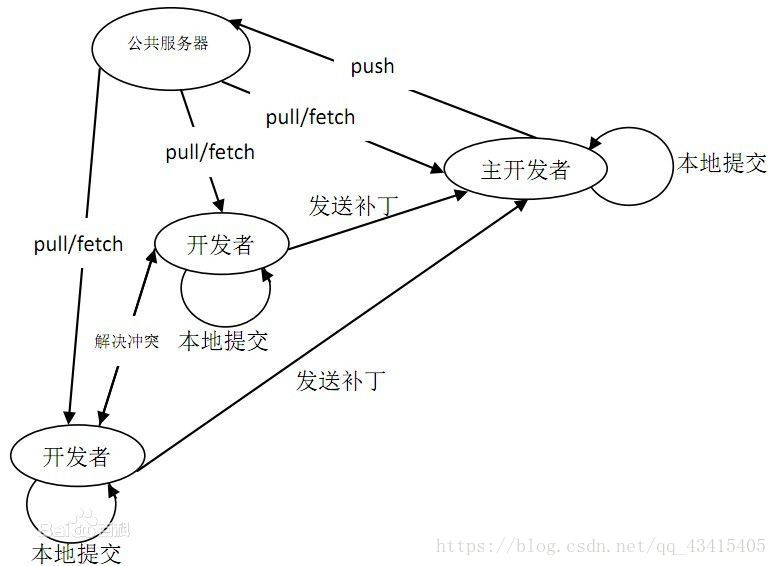
从**一般开发者**的角度来看，git有以下功能：

1. 从服务器上克隆完整的Git仓库（包括代码和版本信息）到单机上。
2. 在自己的机器上根据不同的开发目的，创建分支，修改代码。
3. 在单机上自己创建的分支上提交代码。
4. 在单机上合并分支。
5. 把服务器上最新版的代码fetch下来，然后跟自己的主分支合并。
6. 生成补丁（patch），把补丁发送给主开发者。
7. 看主开发者的反馈，如果主开发者发现两个一般开发者之间有冲突（他们之间可以合作解决的冲突），就会要求他们先解决冲突，然后再由其中一个人提交。如果主开发者可以自己解决，或者没有冲突，就通过。
8. 一般开发者之间解决冲突的方法，开发者之间可以使用pull 命令解决冲突，解决完冲突之后再向主开发者提交补丁。

从**主开发者**的角度（假设主开发者不用开发代码）看，git有以下功能：

1. 查看邮件或者通过其它方式查看一般开发者的提交状态。
2. 打上补丁，解决冲突（可以自己解决，也可以要求开发者之间解决以后再重新提交，如果是开源项目，还要决定哪些补丁有用，哪些不用）。
3. 向公共服务器提交结果，然后通知所有开发人员。

# 经典的Git开发过程



# 学习笔记2——版本库创建

# 什么是版本库？

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”。

# 创建版本库

**第一步**

选择一个合适的地方，创建一个空目录：

$ mkdir learngit（文件名）

$ cd learngit

$ pwd

/e/workspace/workspace-study/learngit

**第二步**

通过git init命令把这个目录变成Git可以管理的仓库：

$ git init

Initialized empty Git repository in E:/workspace/workspace-study/learngit/.git/

瞬间Git就把仓库建好了，而且告诉你是一个**空的仓库（empty Git repository）**，可以发现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，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

* git init 命令的作用是初始化一个 Git 仓库（repository），即把当前所在目录变成 Git 可以管理的仓库。
* git init [project-name] 命令可以新建一个目录，将其初始化为Git仓库。

如果你没有看到.git目录，那是因为这个目录默认是**隐藏**的，用ls -ah命令就可以看见。

$ ls -ah

./ ../ .git/

也不一定必须在空目录下创建Git仓库，选择一个已经有东西的目录也是可以的。

# 把文件添加到版本库

## 事先说明

所有的版本控制系统，其实只能跟踪**文本文件**的改动，比如TXT文件，网页，所有的程序代码等等，Git也不例外。版本控制系统可以告诉你每次的改动，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、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从100KB改成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，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，强烈建议使用标准的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。

## 准备工作

编写一个readme.txt文件，内容如下：

Git is a version control system.

Git is a free software.

一定要放到learngit目录下（子目录也行），因为这是一个Git仓库，如果放到其他地方，那么Git再厉害也找不到这个文件。

## 第一步

用命令git add告诉Git，把文件（这里是上面创建的文件readme.txt）添加到仓库：

$ git add readme.txt

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

* git add [file1] [file2] ... 命令的作用是把文件添加到**暂存区（stage）**，可被 track 追踪纪录下来。可多次使用来添加多个文件。
* git add [dir] 命令的作用是添加指定目录到暂存区，包括子目录。
* git add . 命令的作用是暂存新增加的和修改的文件，不包括已删除的文件。即当前目录下所有文件。

## 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"

[master (root-commit) 62c3c5e] wrote a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

* git commit -m [message] 命令的作用是一次性把暂存区所有文件修改提交到仓库的当前分支，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录。

git commit**命令执行成功后**会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：插入了两行内容（readme.txt有两行内容）。

## 为什么Git添加文件需要add，commit一共两步？

因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

$ git add file1.txt

$ git add file2.txt file3.txt

$ git commit -m "add 3 files"

[master 1e15953] add 3 files

3 files changed, 6 insertions(+)

create mode 100644 file1.txt

create mode 100644 file2.txt

create mode 100644 file3.txt

# 小结

* **初始化**一个Git仓库，使用git init命令。
* **添加文件**到Git仓库，分两步：

1. 使用命令git add [file]，注意，**可反复多次使用**，添加多个文件；
2. 使用命令git commit -m [message]，完成。

# 学习笔记3——修改文件、查看、提交修改

# 修改文件

我们已经成功地添加并提交了一个readme.txt文件，现在，是时候继续工作了，于是，我们继续修改readme.txt文件，改成如下内容：

Git is a distributed version control system.

Git is a free software.

# 显示修改文件

现在，运行git status 命令看看结果：

$ 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 命令的作用是显示当前仓库的最新状态。提交之后，工作区就是“干净的”，即没有新的修改；有未提交文件时，最上面显示的是在 staging area，即将被 commit 的文件；中间显示没有 stage 的修改了的文件，最下面是新的还没有被 Git track 的文件。

上面的命令输出告诉我们，readme.txt被修改过了，但还没有准备提交的修改。

# 显示修改内容

虽然Git告诉我们readme.txt被修改了，但如果能看看**具体修改了什么内容**，自然是很好的。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的readme.txt，所以，需要用git diff 命令看看：

$ git diff readme.txt

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

index 0065b6f..0a30af0 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 a free software.

* git diff [file]命令的作用是查看指定文件具体修改了哪些内容。
* git diff命令的作用是查看工作区中的修改。
* git diff --staged 或 git diff --cached命令的作用是查看暂存区中的修改。
* git diff [commit id1] [commit id2]命令的作用是比较两次 commit 之间的差异。
* git diff [branch1] [branch2]命令的作用是在两个 branch 之间比较。

可以从上面的命令输出看到，我们在第一行添加了一个distributed单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了。

# 提交修改

## 第一步

git add [file]：

$ git add readme.txt

* 1

同样没有任何输出，这时可以运行git status看看当前仓库的状态：

$ git status

On branch master

Changes to be committed:

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

modified: readme.txt

git status告诉我们，将要被提交的修改包括readme.txt。

## 第二步

git commit -m [message]：

$ git commit -m "add distributed"

[master e30c756] add distributed

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

提交后，我们再用git status命令看看仓库的当前状态：

$ git status

On branch master

nothing to commit, working tree 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（working tree clean）的。

# 小结

* 提交修改与添加文件到Git仓库的步骤一样：

1. git add [file]
2. git commit -m [message]

* 要随时掌握工作区的状态，使用git status命令。
* 如果git status告诉你有文件被修改过，用git diff可以查看修改内容。

# 学习笔记4——版本回退

# 背景

再一次**修改**readme.txt文件如下：

Git is a distributed version control system.

Git is a free software distributed under the GPL.

然后**提交**：

$ git add readme.txt

$ git commit -m "append GPL"

[master 4c24228] append GPL

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

像这样，你不断对文件进行修改，然后不断提交修改到版本库里。每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

现在readme.txt文件一共有3个版本被提交到Git仓库里了：  
版本1：wrote a readme file

Git is a version control system.

Git is a free software.

版本2：add distributed

Git is a distributed version control system.

Git is a free software.

版本3：append GPL

Git is a distributed version control system.

Git is a free software distributed under the GPL.

# 查看提交历史

当然了，在实际工作中，我们脑子里怎么可能记得一个几千行的文件每次都改了什么内容，不然要版本控制系统干什么。版本控制系统肯定有某个命令可以告诉我们历史记录，在Git中，我们用git log 命令查看：

$ git log

commit 4c24228bc498e7238e7979ff0c64168c0bae6968 (HEAD -> master)

Author: XXX <XXX@qq.com>

Date: Mon Oct 15 14:55:52 2018 +0800

append GPL

commit e30c75691fe7e0c7038734d61f050c75c45aa316

Author: XXX <XXX@qq.com>

Date: Mon Oct 15 14:37:31 2018 +0800

add distributed

commit 62c3c5e0391cc2ab6f9d552a45bf6d182c7d47ea

Author: XXX <XXX@qq.com>

Date: Mon Oct 15 13:58:55 2018 +0800

wrote a readme file

我们可以看到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。

如果嫌输出信息太多，看得眼花缭乱的，可以试试加上--pretty=oneline参数：

$ git log --pretty=oneline

4c24228bc498e7238e7979ff0c64168c0bae6968 (HEAD -> master) append GPL

e30c75691fe7e0c7038734d61f050c75c45aa316 add distributed

62c3c5e0391cc2ab6f9d552a45bf6d182c7d47ea wrote a readme file

这里你看到的一大串类似4c24228...的是commit id（版本号），是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

* git log 命令的作用是显示从最近到最远的提交日志，包括每个提交的 SHA-1 校验和、作者的名字和电子邮件地址、提交时间以及提交说明等基本信息。
* git log --graph命令的作用是查看分支合并图。
* git log --pretty=oneline命令的作用是简化日志信息，将每个提交放在一行显示，查看的提交数很大时非常有用，也可带有--graph 参数，效果同git config format.pretty oneline。
* git log --graph --pretty=oneline --abbrev-commit命令的作用是查看分支的合并情况，包括分支合并图、一行显示、提交校验码缩略显示。

# 版本回退

现在我们准备把readme.txt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4c24228...（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

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

$ git reset --hard HEAD^

HEAD is now at e30c756 add distributed

* git reset --hard HEAD^命令的作用是回退到上一个版本。同理，回退到上上个版本为：HEAD^^， 回退到上100个版本为：HEAD-100，波浪号 ~ 也可以，变成倒数第101个。

看看readme.txt的内容是不是版本add distributed：

$ cat readme.txt

Git is a distributed version control system.

Git is a free software.

* cat [file]命令的作用是一次显示整个文件。

果然被还原了。我们用git log再看看现在版本库的状态：

$ git log

commit e30c75691fe7e0c7038734d61f050c75c45aa316 (HEAD -> master)

Author: XXX <XXX@qq.com>

Date: Mon Oct 15 14:37:31 2018 +0800

add distributed

commit 62c3c5e0391cc2ab6f9d552a45bf6d182c7d47ea

Author: XXX <XXX@qq.com>

Date: Mon Oct 15 13:58:55 2018 +0800

wrote a readme file

最新的那个版本append GPL已经看不到了！好比你从21世纪坐时光穿梭机来到了19世纪，想再回去已经回不去了，肿么办？

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个append GPL的commit id是4c24228...，于是就可以指定回到未来的某个版本：

$ git reset --hard 4c24228

HEAD is now at 4c24228 append GPL

* git reset --hard [commit id]命令的作用是回退到某一提交过的版本，如果已经 push，则回退的意义不大了。

**版本号没必要写全**，前几位就可以了，Git会自动去找。当然也**不能只写前一两位**，因为Git可能会找到多个版本号，就无法确定是哪一个了。

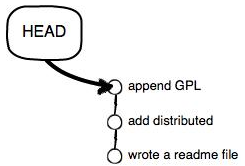
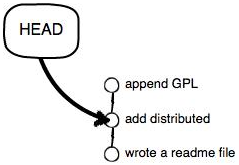
再看看readme.txt的内容：

$ cat readme.txt

Git is a distributed version control system.

Git is a free software distributed under the GPL.

果然，又回来了。

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**，当你回退版本的时候，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。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

现在，你回退到了某个版本，关掉了电脑，第二天早上就后悔了，想恢复到新版本怎么办？**找不到新版本的**commit id怎么办？

在Git中，总是有后悔药可以吃的。当你用$ git reset --hard HEAD^回退到add distributed版本时，再想恢复到append GPL，就必须找到append GPL的commit id。Git提供了一个命令git reflog用来记录你的每一次命令：

$ git reflog

e30c756 HEAD@{1}: reset: moving to HEAD^

4c24228 (HEAD -> master) HEAD@{2}: commit: append GPL

e30c756 HEAD@{3}: commit: add distributed

62c3c5e HEAD@{4}: commit (initial): wrote a readme file

从输出可知，append GPL的commit id是4c24228，现在，你又可以乘坐时光机回到未来了。

* git reflog命令的作用是显示当前分支的最近几次提交。它纪录每一次命令，可用于查找某一提交版本的 commit id。

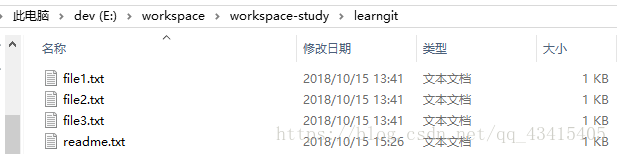
# 小结

* HEAD指向的版本就是当前版本，因此，Git允许我们在版本的历史之间穿梭，使用命令git reset --hard commit\_id。
* 穿梭前，用git log可以查看提交历史，以便确定要回退到哪个版本。
* 要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

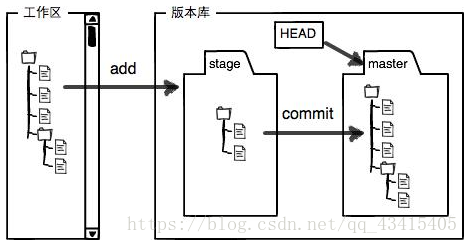
# 学习笔记5——工作区和暂存区

# 名词解释

## 工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个**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为我们自动创建的第一个**分支**master，以及指向master的一个**指针**叫HEAD。  
  
前面讲了我们把文件往Git版本库里添加的时候，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。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个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。

# 工作原理

现在再对readme.txt做个修改，比如加上一行内容：

Git is a distributed version control system.

Git is a 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非常清楚地告诉我们，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再查看一下：

$ git status

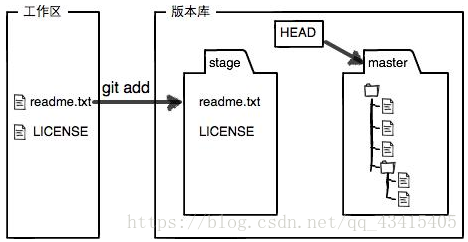
On branch master

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：  
  
所以，git add命令实际上就是把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 0de6e49] understand how stage works

2 files changed, 4 insertions(+), 1 deletion(-)

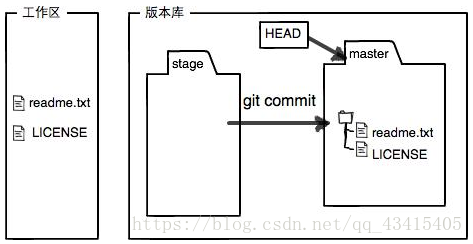
create mode 100644 LICENSE

一旦提交后，如果你又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 tree clean

现在版本库变成了这样，暂存区就没有任何内容了：  


# 小结

**暂存区**是Git非常重要的概念，弄明白了暂存区，就弄明白了Git的很多操作到底干了什么。

# 学习笔记6——管理修改

# 什么是修改？

比如你新增了一行，这就是一个修改，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。

# Git跟踪并管理的是修改，而不是文件

当你用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；如果你又进行了第二次修改，之后没有使用git add命令，那么第二次修改并不会被放入暂存区，而git commit只负责把暂存区的修改提交，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 -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：

$ 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.

* git diff HEAD -- [file]命令的作用是查看版本库最新版本和工作区之间的区别。

可见，第二次修改确实没有被提交。

那怎么提交第二次修改呢？你可以继续git add再git commit，也可以别着急提交第一次修改，先git add第二次修改，再git commit，就相当于把两次修改合并后一块提交了：

第一次修改 -> git add -> 第二次修改 -> git add -> git commit

# 撤销修改

## 尚未git add

如果错误发现得很及时，就可以很容易地纠正它。你可以手动把文件恢复到上一个版本的状态。如果用git status查看一下：

$ 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")

### 使用git checkout -- [file]

你可以发现，Git会告诉你，git checkout -- [file]可以丢弃工作区的修改：

$ git checkout -- readme.txt

* git checkout -- [file]命令的作用是丢弃工作区的修改，包括**修改后还没有放到暂存区**和**添加到暂存区后又作了修改**两种情况。总之，让该文件回到最近一次git commit 或git add 之后的状态。

git checkout -- [file]命令中的--很重要，没有--，就变成了“**切换到另一个分支**”的命令。  
命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

* 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。
* readme.txt已经添加到暂存区后，又作了修改，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。

## 已git add，尚未git commit

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

### 使用git reset HEAD [file]

Git同样告诉我们，用命令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 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。当我们用HEAD时，表示最新的版本。
* git reset HEAD [file]命令的作用是把暂存区的修改撤销（unstage），回退到工作区。注意：在 Git 中任何已提交的东西几乎总是可以恢复的。甚至那些被删除的分支中的提交或使用 –amend 选项覆盖的提交也可以恢复。然而，任何未提交的东西丢失后很可能再也找不到了。

再用git status查看一下，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

### 再使用git checkout -- [file]

用git checkout -- [file]丢弃工作区的修改：

$ git checkout -- readme.txt

$ git status

On branch master

nothing to commit, working tree clean

## 已提交到版本库

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？可以通过[版本回退](https://blog.csdn.net/qq_43415405/article/details/83059677)来回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。

# 删除文件

在Git中，**删除也是一个修改操作**。

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，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

$ rm test.txt

* git rm [file1] [file2] ...命令的作用是删除工作区文件，并且将这次删除放入暂存区。

这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

$ 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")

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm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

现在，文件就从版本库中被删除了。

**先手动删除文件，然后使用git rm [file]和git add [file]效果是一样的**。

另一种情况是删错了，因为版本库里还有，所以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”。

# 小结

* 每次修改，如果不用git add到暂存区，那就不会加入到commit中。
* 撤销修改的不同场景：

1. 场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- [file]。
2. 场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD [file]，就回到了场景1，第二步按场景1操作。
3. 场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考[**版本回退**](https://blog.csdn.net/qq_43415405/article/details/83059677)，不过前提是没有推送到远程库。

* 命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你**只能恢复文件到最新版本**，你**会丢失最近一次提交后你修改的内容**。

# 学习笔记7——远程仓库

# 介绍

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。怎么分布呢？最早，肯定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。

实际情况往往是这样，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，并且各自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。

[GitHub网站](https://github.com/)就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得Git远程仓库。

# 设置

由于你的**本地Git仓库**和**GitHub仓库**之间的传输是通过SSH加密的，所以，需要一点设置：

## 第一步

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Git Bash，创建SSH Key：

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

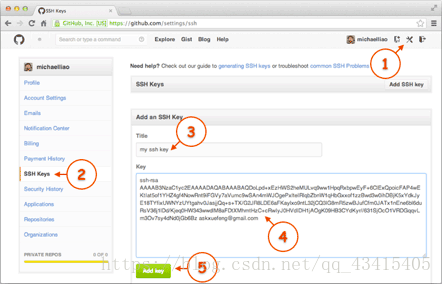
你需要把邮件地址换成你自己的邮件地址，然后一路回车，使用默认值即可，由于这个Key也不是用于军事目的，所以也无需设置密码。

如果一切顺利的话，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的密钥对，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。

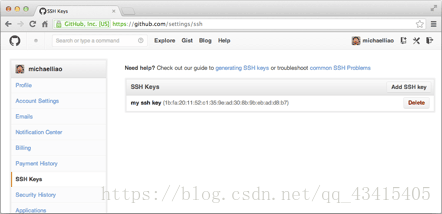
## 第二步

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“Add Key”，你就应该看到已经添加的Key：



为什么GitHub需要SSH Key？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

当然，**GitHub允许你添加多个Key**。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

**在GitHub上免费托管的Git仓库，任何人都可以看到（但只有你自己才能改）。所以，不要把敏感信息放进去。**

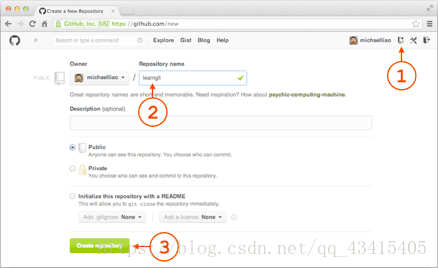
如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。

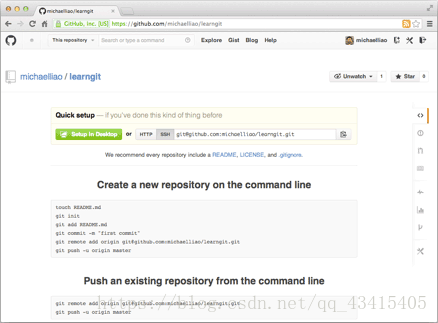
# 添加远程仓库

你**已经在本地创建了一个Git仓库**后，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。

## 第一步

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：



## 第二步

目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，**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。

现在，我们根据GitHub的提示，在**本地的learngit仓库**下运行命令：

$ git remote add origin git@github.com:example/learngit.git

记得把上面的example替换成你自己的**GitHub账户名**。

* git remote add [shortname] [url]命令的作用是添加并关联一个远程库。其中，shortname 一般是 origin，也可以是其他字符串，用来代替整个 url。

添加后，远程库的名字就是origin，这是Git默认的叫法，也可以改成别的，但是origin这个名字一看就知道是远程库。

## 第三步

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: 14, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (13/13), done.

Writing objects: 100% (14/14), 1.18 KiB | 605.00 KiB/s, done.

Total 14 (delta 5), reused 0 (delta 0)

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

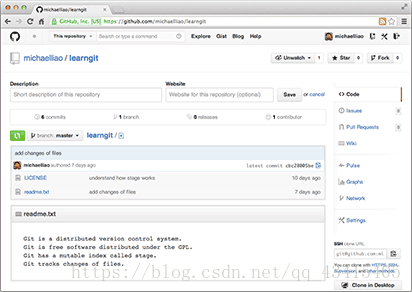
To github.com:example/learngit.git

\* [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。

* git push -u origin master命令：关联后，由于远程库是空的，使用该命令第一次推送 master 分支的所有内容，Git不但会把本地的master分支内容推送到远程新的master分支，还会把本地的master分支和远程的master分支关联起来，后续再推送的时候就可以省略后面三个参数了，其中参数 -u 代表上游（upstream）的意思。
* git push [remote] [branch]命令的作用是上传本地指定分支到远程仓库。

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模一样：  
  
从现在起，只要本地作了提交，就可以通过命令：

$ git push origin master

把本地master分支的最新修改推送至GitHub，现在，你就拥有了真正的分布式版本库！

# SSH警告

当你第一次使用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使用SSH连接，而SSH连接在第一次验证GitHub服务器的Key时，需要你确认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.

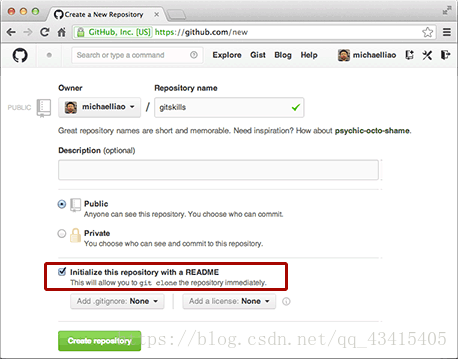
这个警告只会出现一次，后面的操作就不会有任何警告了。

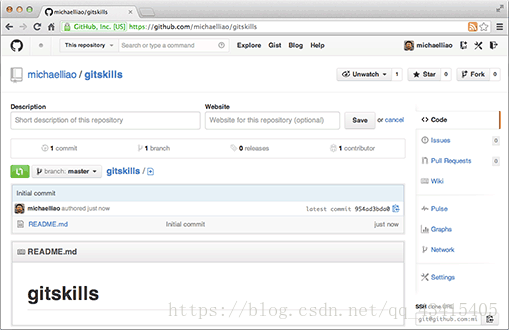
# 从远程仓库克隆

假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

## 第一步

登陆GitHub，创建一个新的仓库，名字叫gitskills：

  
我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：



## 第二步

用命令git clone克隆一个本地库：

$ git clone git@github.com:example/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.

* git clone [url]命令的作用是从远程库（origin）克隆一份到本地，仓库名同远程仓库名。

注意把Git库的地址换成你自己的，然后进入gitskills目录看看，已经有README.md文件了：

$ cd gitskills

$ ls

README.md

* ls命令用于显示文件目录列表。

如果有**多个人协作开发**，那么每个人各自从远程克隆一份就可以了。

你也许还注意到，GitHub给出的地址不止一个，还可以用https://github.com/example/gitskills.git这样的地址。实际上，Git支持多种协议，默认的git://使用ssh，但也可以使用https等其他协议。

使用https除了速度慢以外，还有个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协议而只能用https。

# 小结

* 要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git。
* 关联后，使用命令git push -u origin master第一次推送master分支的所有内容。
* 此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改。
* **分布式版本系统的最大好处之一**是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，当有网络的时候，再把本地提交推送一下就完成了同步。
* 要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。
* Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

# 学习笔记8——分支管理

# 分支在实际中有什么用？

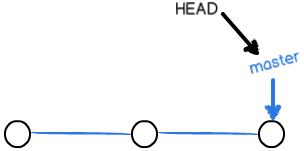
假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

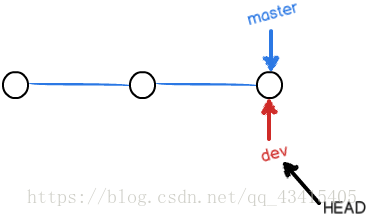
现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

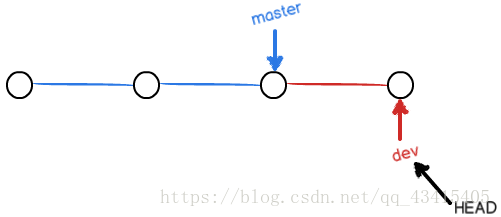
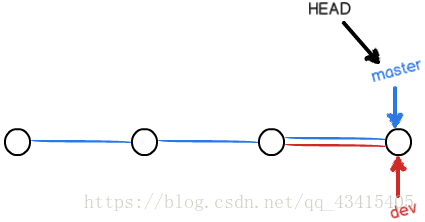
# 创建与合并分支

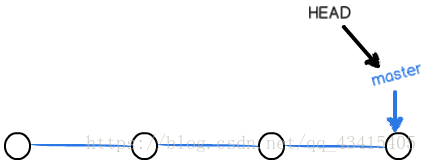
每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

## 过程图示

一开始的时候，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，以及当前分支的提交点：  
  
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

当我们创建新的分支，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：  
  
Git创建一个分支很快，因为除了增加一个dev指针，改改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分支了，比如新提交一次后，dev指针往前移动一步，而master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。最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：  
  
所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：  


## 代码实现

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

* git checkout -b [branch]命令的作用是创建并切换到新的分支。

git checkout命令加上-b参数，相当于以下两条命令：

$ git branch dev

$ git checkout dev

Switched to branch 'dev'

* git branch [branch-name]命令的作用是新建一个分支，但依然停留在当前分支。
* git checkout [branch-name]命令的作用是切换到指定分支，并更新工作区。

然后，用git branch命令查看当前分支：

$ git branch

\* dev

master

* git branch命令的作用是列出本地当前所有分支，方便查看。当前分支前面会标有一个 \* 号。

然后，我们就可以在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"

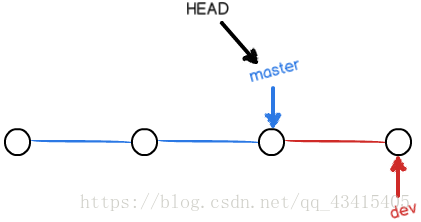
[dev b17d20e] branch test

1 file changed, 1 insertion(+)

现在，dev分支的工作完成，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，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：  
  
现在，我们把dev分支的工作成果合并到master分支上：

$ git merge dev

Updating d46f35e..b17d20e

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

* git merge [branch]命令用于合并指定分支到当前分支。

合并后，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，所以合并速度非常快。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was b17d20e).

* git branch -d [branch]命令的作用是**普通删除分支**（相对强制删除而言）。一般情况下，先合并完分支，然后再删除，否则会删除失败，除非使用 -D 参数强制删除。

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

因为**创建、合并和删除分支非常快**，所以Git鼓励你**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。

# 解决冲突

合并分支往往并不是一帆风顺的。

准备新的feature1分支，继续新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

修改readme.txt。

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(-)

切换到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还会自动提示我们当前master分支比远程的master分支要超前1个提交。

在master分支上修改readme.txt文件（与前一次修改不同）。

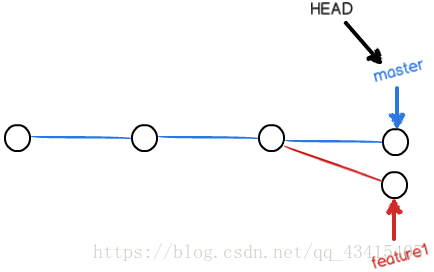
提交：

$ 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，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，但这种合并就可能会有冲突：

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict in readme.txt

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

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.

(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")

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

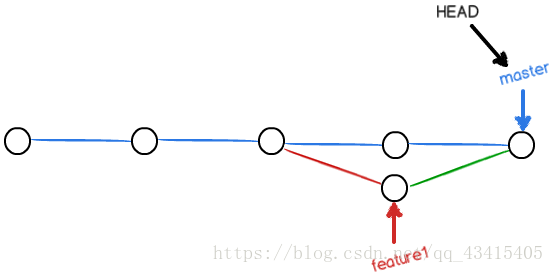
Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改后保存。

再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master cf810e4] 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).

git branch -d [branch-name]命令的作用是删除分支。

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。

## git merge --no-ff

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：

$ git add readme.txt

$ git commit -m "add merge"

[dev f52c633] 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。

* git merge --no-ff -m "message" [branch]命令：参数 --no-ff 表示禁用 Fast forward 快进模式，用普通模式合并，这样合并后的历史有分支，能看出来曾经做过合并，而 Fast forwad 合并就看不出来曾经做过合并。

合并后，我们用git log看看分支历史：

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

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

|\

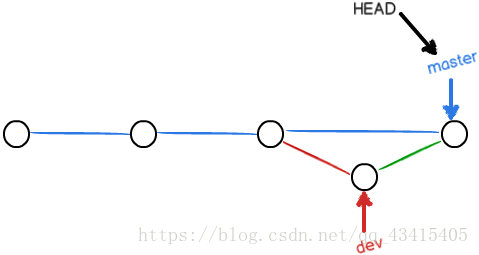
| \* f52c633 (dev) add merge

|/

\* cf810e4 conflict fixed

...

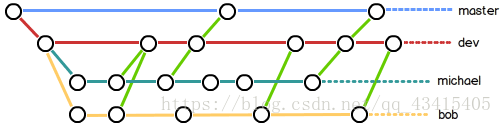
* git log --graph --pretty=oneline --abbrev-commit命令的作用是查看分支的合并情况，包括分支合并图、一行显示、提交校验码缩略显示。

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**：

* master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；
* 干活都在dev分支上，也就是说，dev分支是不稳定的，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。

当你接到一个修复一个代号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，当前正在dev上进行的工作还没有提交，并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时内修复该bug，怎么办？

## git stash

幸好，Git还提供了一个stash功能，可以把当前工作现场“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

* git stash命令的作用是把当前分支的工作现场储存起来，等以后恢复现场后继续工作。一般适用于**还没有 commit** 的分支代码。

现在，用git status查看工作区，就是干净的（除非有没有被Git管理的文件），因此可以放心地创建分支来修复bug。

## 修复bug

首先确定要在哪个分支上修复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.

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

现在，是时候接着回到dev分支干活了！

$ git checkout dev

Switched to branch 'dev'

$ git status

On branch dev

nothing to commit, working tree clean

## git stash list

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

$ git stash list

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

* git stash list命令的作用是查看储存的工作现场纪录列表。

## 工作现场恢复

工作现场还在，Git把stash内容存在某个地方了，但是需要恢复一下，有两个办法：

* 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；
* 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} (5d677e2ee266f39ea296182fb2354265b91b3b2a)

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，总有无穷无尽的新的功能要不断添加进来。

添加一个新功能时，你肯定不希望因为一些实验性质的代码，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。

当你接到一个新任务开发某新功能，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，但是这个包含机密资料的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友情提醒，feature-vulcan分支还没有被合并，如果删除，将丢失掉修改，如果要强行删除，需要使用大写的-D参数。

现在我们**强行删除**：

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 287773e).

* git branch -D [branch]命令的作用是强行删除分支，尤其适用分支内容**有了新的修改但还没有被合并**的情况。

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。

## 查看远程仓库

要查看远程库的信息，用git remote：

$ git remote

origin

* git remote命令的作用是查看已经配置的远程仓库服务器，效果同 git remote show。

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)

* git remote -v命令的作用是显示需要读写远程仓库使用的 Git 保存的简写与其对应的 URL。

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

如果要推送其他分支，比如dev，就改成：

$ git push origin dev

* git push origin [branch]命令的作用是推送最新修改。

但是，并不是一定要把本地分支往远程推送，那么，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。

现在，模拟一个你的小伙伴，可以在另一台电脑（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.

当你的小伙伴从远程库clone时，默认情况下，你的小伙伴只能看到本地的master分支。可以用git branch命令看看：

$ 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 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

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.

**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，解决办法也很简单，Git已经提示我们，先用git pull把最新的提交从origin/dev抓下来，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命令的作用是抓取远程库最新提交，拉取并合并。

git pull也失败了，原因是没有指定本地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：

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

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

* git branch --set-upstream [branch] origin/[branch]命令的作用是建立本地分支和远程分支的关联。

再pull：

$ git pull

Auto-merging env.txt

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

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

这回git pull成功，但是合并有冲突，需要手动解决，解决的方法和分支管理中的解决冲突完全一样。解决后，提交，再push：

$ 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

因此，**多人协作的工作模式**通常是这样：

1. 首先，可以试图用git push origin [branch]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]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-to [branch] origin/[branch]。

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

# Rebase

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的童鞋不得不先pull，在本地合并，然后才能push成功。

每次合并再push后，分支变成了这样：

$ 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

总之看上去很乱，有强迫症的童鞋会问：为什么Git的提交历史不能是一条干净的直线？其实是可以做到的！

Git有一种称为rebase的操作，有人把它翻译成“变基”。

在和远程分支同步后，我们对hello.py这个文件做了两次提交。用git log命令看看：

$ 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快两个提交。

现在我们尝试推送本地分支：

$ 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.

很不幸，失败了，这说明有人先于我们推送了远程分支。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看看状态：

$ 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，现在我们本地分支比远程分支超前3个提交。

用git log看看：

$ 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

...

这个时候，rebase就派上了用场。我们输入命令git rebase试试：

$ 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

输出了一大堆操作，到底是啥效果？再用git log看看：

$ 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了！这种神奇的操作是怎么实现的？其实原理非常简单。我们注意观察，发现Git把我们本地的提交“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，我们本地的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已经被修改过了。

rebase 会把提交到某一分支（当前分支）上的所有修改都转移至另一分支（目标分支）上，就好像“重新播放”一样。

**变基**是将一系列提交按照原有次序依次应用到另一分支上，而**合并**是把最终结果合在一起。简言之：这两种整合方法的**最终结果没有任何区别**，但是**变基使得提交历史更加整洁**。

**变基有风险**，需要遵守的准则是：不要对在你的仓库外有副本的分支执行变基。否则，会导致混乱。总的原则是，只对**尚未推送或分享给别人的本地修改**执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样才能享受到两种方式带来的便利。

最后，通过push操作把本地分支推送到远程：

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看看效果：

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

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

\* 3611cfe add comment

\* f005ed4 set exit=1

\* d1be385 init hello

...

远程分支的提交历史也是一条直线。

# 小结

* Git鼓励大量使用分支：

1. 查看分支：git branch
2. 创建分支：git branch <name>
3. 切换分支：git checkout <name>
4. 创建+切换分支：git checkout -b <name>
5. 合并某分支到当前分支：git merge <name>
6. 删除分支：git branch -d <name>

* 当Git**无法自动合并分支**时，就必须首先解决冲突。解决冲突后，再提交，合并完成。
* 解决冲突就是把Git合并失败的文件**手动编辑**为我们希望的内容，再提交。
* 用git log --graph命令可以看到分支合并图。
* Git分支十分强大，在团队开发中应该充分应用。
* 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。
* 修复bug时，我们会通过创建新的bug分支进行修复，然后合并，最后删除。
* 当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场。
* 开发一个新feature，最好新建一个分支。
* 如果要**丢弃一个没有被合并过的分支**，可以通过git branch -D <name>强行删除。
* 查看远程库信息，使用git remote -v。
* 本地新建的分支如果不推送到远程，对其他人就是不可见的。
* 从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交。
* 在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致。
* 建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name。
* 从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。
* rebase操作可以把本地未push的分叉提交历史整理成直线。
* rebase的目的是使得我们在查看历史提交的变化时更容易，因为分叉的提交需要三方对比。

# 学习笔记9——标签管理

# 标签介绍

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就**唯一确定**了打标签时刻的版本。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。

Git的标签虽然是版本库的快照，但其实它就是指向某个commit的**指针**（跟分支很像，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**。

Git有commit，为什么还要引入tag？tag就是一个让人容易记住的有意义的名字，它跟某个commit绑在一起。

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。

# 创建标签

在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，敲命令git tag [name]就可以打一个新标签：

$ git tag v1.0

* git tag [tag name] [commit id]命令的作用是在需要打标签的分支上创建一个轻量标签（lightweight），默认为 HEAD，也可以指定一个 commit id。

可以用命令git tag查看所有标签：

$ git tag

v1.0

* git tag命令的作用是查看所有标签。这里标签不是按照时间列出，而是按照字母排序。

默认标签是打在**最新提交**的commit上的。有时候，如果忘了打标签，比如，现在已经是周五了，但应该在周一打的标签没有打，怎么办？

方法是找到历史提交的commit id，然后打上就可以了：

$ 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

比方说要对add merge这次提交打标签，它对应的commit id是f52c633，敲入命令：

$ git tag v0.9 f52c633

再用命令git tag查看标签：

$ git tag

v0.9

v1.0

可以用git show [tag name]查看标签信息：

$ git show v0.9

commit f52c63349bc3c1593499807e5c8e972b82c8f286 (tag: v0.9)

Author: Michael Ligit有ao <askxuefeng@gmail.com>

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

add merge

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

...

* git show [tag name]命令的作用是查看标签信息。

可以看到，v0.9确实打在add merge这次提交上。

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

* git tag -a [tag name] -m "message" [commit id]命令的作用是创建附注标签（annotated），用 -a 指定标签名，-m 指定说明文字，也可以指定一个 commit id。

# 操作标签

## 删除本地标签

如果标签打错了，也可以删除：

$ git tag -d v0.1

Deleted tag 'v0.1' (was f15b0dd)

* git tag -d [tag name]命令的作用是删除一个本地标签。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。

## 推送标签

如果要推送**某个标签**到远程，使用命令git push origin [tag name]：

$ 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

* git push origin [tag name]命令的作用是推送本地某个标签到远程，默认情况下，git push 命令并不会推送标签到远程，必须显式推送。

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

* git push origin --tags命令：参数 –tags 表示一次性推送全部未推送到远程的本地标签，当其他人从仓库中克隆或拉取，他们也能得到那些标签。

## 删除远程标签

如果标签已经推送到远程，要删除远程标签就麻烦一点，**先从本地删除**：

$ git tag -d v0.9

Deleted tag 'v0.9' (was f52c633)

然后，**从远程删除**。删除命令也是push，但是格式如下：

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

To github.com:michaelliao/learngit.git

- [deleted] v0.9

* git push origin :refs/tags/[tag name]命令的作用是删除一个远程标签。要先从本地删除，再用该命令从远程删除。

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

# 小结

* 命令git tag [tag name]用于新建一个标签，默认为HEAD，也可以指定一个commit id。
* 命令git tag -a [tag name] -m "message"可以指定标签信息。
* 命令git tag可以查看所有标签。
* 命令git push origin [tag name]可以推送一个本地标签。
* 命令git push origin --tags可以推送全部未推送过的本地标签。
* 命令git tag -d [tag name]可以删除一个本地标签。
* 命令git push origin :refs/tags/[tag name]可以删除一个远程标签。

# 学习笔记10——使用GitHub

我们一直用GitHub作为免费的远程仓库，如果是个人的开源项目，放到GitHub上是完全没有问题的。

其实GitHub还是一个**开源协作社区**，通过GitHub，既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。

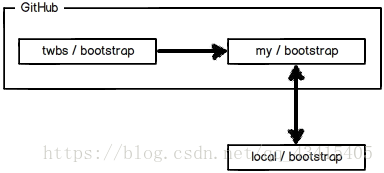
在GitHub出现以前，开源项目开源容易，但让广大人民群众参与进来比较困难，因为要参与，就要提交代码，而给每个想提交代码的群众都开一个账号那是不现实的，因此，群众也仅限于报个bug，即使能改掉bug，也只能把diff文件用邮件发过去，很不方便。

但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页<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:example/bootstrap.git

一定要从自己的账号下clone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，你将不能推送修改。

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：  
  
如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，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，对方是否接受你的pull request就不一定了。

# 小结

* 在GitHub上，可以任意Fork开源仓库。
* 自己拥有Fork后的仓库的读写权限。
* 可以推送pull request给官方仓库来贡献代码。

# 学习笔记11——配置Git、搭建Git服务器

# 忽略特殊文件

有些时候，你**必须把某些文件放到Git工作目录中**，**但又不能提交它们**，比如保存了数据库密码的配置文件啦，等等，每次git status都会显示Untracked files ...，有强迫症的童鞋心里肯定不爽。

好在Git考虑到了大家的感受，这个问题解决起来也很简单，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。

不需要从头写.gitignore文件，GitHub已经为我们准备了各种配置文件，只需要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：

* 忽略**操作系统自动生成的文件**，比如缩略图等。
* 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。
* 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，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

加上你自己定义的文件，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！当然检验.gitignore的标准是git status命令是不是说working directory clean。

有些时候，你想添加一个文件到Git，但发现添加不了，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.

如果你确实想添加该文件，可以用-f强制添加到Git：

$ git add -f App.class

* git add -f [file]命令的作用是强制添加被 .gitignore 忽略的文件到 Git。

或者你发现，可能是.gitignore写得有问题，需要找出来到底哪个规则写错了，可以用git check -ignore命令检查：

$ git check-ignore -v App.class

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

* git check-ignore -v [file]命令：可能是 .gitignore 写得有问题时，使用该命令找出到底哪个命令写错了。

Git会告诉我们，.gitignore的第3行规则忽略了该文件，于是我们就可以知道应该修订哪个规则。

# 配置别名

有没有经常**敲错命令**？比如git status？status这个单词真心不好记。

如果敲git st就能表示git status，那就简单多了，当然这种偷懒的办法我们是极力赞成的。

我们只需要敲一行命令，告诉Git，以后st就表示status：

$ git config --global alias.st status

* git config --global alias.st status命令的作用是使用 git st 代替 git status 命令。

当然还有别的命令可以简写，很多人都用co表示checkout，ci表示commit，br表示branch：

$ 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 "message"

--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](https://blog.csdn.net/qq_43415405/article/details/83060512)中，我们知道，命令git reset HEAD [file]可以把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'

当你敲入命令：

$ git unstage test.py

实际上Git执行的是：

$ git reset 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

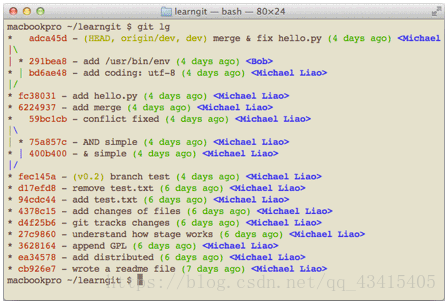
Author: Michael Liao <askxuefeng@gmail.com>

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

merge & fix hello.py

还可以丧心病狂地配置 git lg 命令，让显示 log 更加优雅，逼格更高：

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"

此时git lg效果如下：  


# 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。

配置文件放哪了？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，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，可以删掉文件重新通过命令配置。

# 搭建Git服务器

在[远程仓库](https://blog.csdn.net/qq_43415405/article/details/83061512)中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

GitHub就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。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

## 第一步

安装git：

$ sudo apt-get install git

## 第二步

创建一个git用户，用来运行git服务：

$ sudo 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：

$ sudo 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owner改为git：

$ sudo 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，这可以通过编辑/etc/passwd文件完成。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.

剩下的推送就简单了。

## 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。

## 管理权限

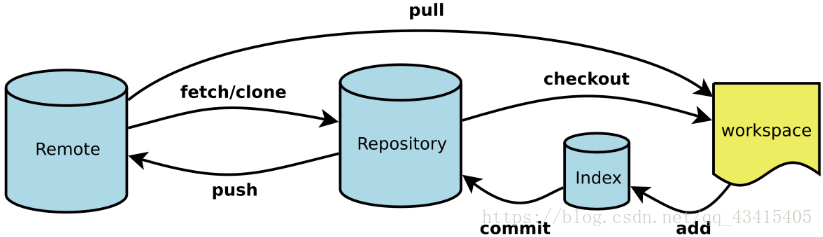
有很多不但视源代码如生命，而且视员工为窃贼的公司，会在版本控制系统里设置一套完善的权限控制，每个人是否有读写权限会精确到每个分支甚至每个目录下。因为Git是为Linux源代码托管而开发的，所以Git也继承了开源社区的精神，不支持权限控制。不过，因为Git支持钩子（hook），所以，可以在服务器端编写一系列脚本来控制提交等操作，达到权限控制的目的。Gitolite就是这个工具。

# 小结

* 忽略某些文件时，需要编写.gitignore。
* .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理。
* 给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。
* 要方便管理公钥，用[Gitosis](https://github.com/sitaramc/gitolite)。
* 要像SVN那样变态地控制权限，用[Gitolite](https://github.com/sitaramc/gitolite)。

# Git学习笔记12——命令总结

# 日常使用命令



workspace：工作区

index/stage：暂存区

repository：本地仓库

remote：远程仓库

在交互式操作中常用命令：

* p，pick = use commit
* r，reword = use commit，but edit the commit message
* e，edit = use commit，but stop for amending
* s，squash = use commit，but meld into previous commit
* f，fixup = like “squash”，but discard this commit’s log message
* x，exec = run command（the rest of the line）using shell
* d，drop = remove commit

# 新建版本库

# 在本地初始化一个 Git 仓库（repository），即把当前所在目录变成 Git 可以管理的仓库。

$ git init

# 在本地新建一个目录，将其初始化为Git仓库。

$ git init [project-name]

# 从远程库（origin）克隆一个仓库到本地，仓库名同远程仓库名.

$ git clone [url]

# 添加、提交文件

# 把文件添加到暂存区（stage），可被 track 追踪纪录下来。可多次使用来添加多个文件。

$ git add [file1] [file2] ...

# 添加指定目录到暂存区，包括子目录。

$ git add [dir]

# 暂存新增加的和修改的文件，不包括已删除的文件。即当前目录下所有文件。

$ git add .

# 强制添加被 .gitignore 忽略的文件到 Git。

$ git add -f [file]

# 一次性把暂存区所有文件修改提交到仓库的当前分支。

# 注意：提交信息可为中文也可为英文，若为英文则通常用一般现在时。

# 如果不加参数 -m 则会跳转到编辑器强制填写提交说明信息。

$ git commit -m "本次提交说明"

# 跳过暂存区，直接提交工作目录中所有改变的文件，但是不能提交工作目录中新增的文件

$ git commit -a -m "提交消息"

# 提交暂存区的文件，并重写提交作者

$ git commit --author=lavor -m "提交消息"

# 提交暂存区的文件，并重写提交日期

$ git commit --date=06.13.2016T09:00:00 -m "提交消息"

# 通过创建一个新的提交，以替换当前分支的前端。

# 所代表的含义就是在最新一次提交的基础上进行提交。

# 比如我们完成了最新一次提交，并且这次提交完成后我们对工作目录进行了一些修改，

# 但是我们发现某个文件忘了添加到暂存区并提交，

# 我们可以先添加该文件到暂存区，然后利用该命令进行提交。

$ git commit --amend -m "提交消息"

# 为文件重命名

$ git mv oldfile newfile

# 移动文件到指定目录

$ git mv files dir

# 强制为文件重命名，即使目标文件已存在

$ git mv -f oldfile newfile

# 强制移动文件到指定目录名，即使目标文件已存在

$ git mv -f files dir

# 查看状态、修改

# 显示当前仓库的最新状态。

# 提交之后，工作区就是“干净的”，即没有新的修改；

# 有未提交文件时，最上面显示的是在 staging area，即将被 commit 的文件；

# 中间显示没有 stage 的修改了的文件，

# 最下面是新的还没有被 Git track 的文件。

# Untracked files：未被跟踪的文件，表示是工作目录新增加的文件；

# Changes not staged for commit：工作目录中修改了文件，但是没有被添加到暂存区；

# Changes to be committed：添加到暂存区的文件，等待提交。

$ git status

# 查看工作目录与暂存区的差异。

$ git diff

# 查看指定文件具体修改了哪些内容。

$ git diff [file]

# 查看暂存区与指定提交（默认是HEAD）的差异。

$ git diff --cached [<commit>]

# 查看暂存区中的修改。

$ git diff --staged 或 git diff --cached

# 查看工作目录与指定提交的差异。

$ git diff <commit>

# 查看两次指定提交的差异。

$ git diff [commit id1] [commit id2]

# 查看工作目录与指定分支的差异。

$ git diff [branch]

# 查看两个指定分支间的差异。

$ git diff [branch1] [branch2]

# 查看版本库最新版本和工作区之间的区别。

$ git diff HEAD -- [file]

# 显示最近 times 次（默认是一次）提交的所有对象信息。

$ git show [-times]

# 查看日志

# 显示从最近到最远的提交日志，

# 包括每个提交的SHA-1校验和、作者的名字和电子邮件地址、提交时间以及提交说明等基本信息。

$ git log

# 查看所有提交记录。

$ git log --all

# 查看提交记录，以 oneline 形式显示，只显示一行，

# 显示的内容是提交hash的前7位与提交消息。

$ git log --oneline

# 表示查看最近 times 次提交改变的内容。

$ git log -p -times：

# 查看最近 times 次（默认是所有）提交记录，并显示文件的差异分析。

$ git log -stat [-times]

# 查看分支合并图。

$ git log --graph

# 简化日志信息，将每个提交放在一行显示，

# 查看的提交数很大时非常有用，

# 也可带有--graph 参数，效果同git config format.pretty oneline。

$ git log --pretty=oneline

# 查看分支的合并情况，包括分支合并图、一行显示、提交校验码缩略显示。

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

# 显示总提交次数与每次提交的提交消息。

$ git shortlog

# 版本回退

# 将暂存区的所有文件重置到当前分支的HEAD

$ git reset

# 将暂存区的指定文件重置到指定的 <commit> ,

# <commit> 既可以是commit的hash（或者hash前7位）也可以是 HEAD 及其祖先，

# HEAD~1 表示 HEAD 的父亲，是 HEAD 的前一次提交，

# 没有 <commit> 时默认是 HEAD。

$ git reset <commit> files

# 将当前的分支重设到指定的 <commit>，并且根据mode有可能更新暂存区和工作目录。

# mode 的取值可以是 hard、soft、mixed、merged、keep。

# hard ：重置暂存区与工作目录到指定提交，删除 <commit> 之后的所有提交并将 HEAD 指向该提交，此操作危险指数较大（应慎用）

# soft ：暂存区与工作目录不会变化，仅仅删除 <commit> 之后的所有提交并将 HEAD 指向该提交

# mixed ：默认的，重置暂存区到指定提交，删除 <commit> 之后的所有提交并将 HEAD 指向该提交

# merge与keep用的比较少

$ git reset [--hard|soft|mixed|merge|keep] [<commit>]

# 回退到上一个版本。

# 同理，回退到上上个版本为：HEAD^^，

# 回退到上100个版本为：HEAD-100，波浪号 ~ 也可以，变成倒数第101个。

$ git reset --hard HEAD^

# 回退到某一提交过的版本，如果已经 push，则回退的意义不大了。

# 版本号没必要写全，前几位就可以了，Git会自动去找。

# 当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

$ git reset --hard [commit id]

# 显示所有提交，下拉，推送，与切换分支操作。

# 显示当前分支的最近几次提交。它记录每一次命令，可用于查找某一提交版本的 commit id。

$ git reflog

# 显示所有提交，下拉，推送操作。

$ git reflog --all

# 撤销修改

# 丢弃工作区的修改，包括修改后还没有放到暂存区和添加到暂存区后又作了修改两种情况。

# 总之，让该文件回到最近一次git commit 或git add 之后的状态。

# 注意：如果没有--，就变成了切换分支的命令了。

$ git checkout -- [file]

# 既可以回退版本，也可以把暂存区的修改回退到工作区。

# 当我们用HEAD时，表示最新的版本。

$ git reset

# 把暂存区的修改撤销（unstage），回退到工作区。

# 注意：在 Git 中任何已提交的东西几乎总是可以恢复的。

# 甚至那些被删除的分支中的提交或使用 –amend 选项覆盖的提交也可以恢复。

# 然而，任何未提交的东西丢失后很可能再也找不到了。

$ git reset HEAD [file]

# 删除文件

# 删除工作区文件，并且将这次删除放入暂存区。

$ git rm [file1] [file2] ...

# 强制删除版本库中有修改的文件，不做更新检查。

$ git rm -f 文件

# 删除暂存区的文件，把文件从版本库中删除，但让文件保留在工作区且不被 Git 继续追踪（track），

# 通常适用于在 rm 之后把文件添加到 .gitignore 中的情况。

$ git rm --cached 文件

# 修补

# 在暂存区与工作目录或者暂存区（默认是工作目录）打补丁

$ git apply [--index|--cached] patchname.patch

# 反向打补丁

$ git apply --reverse|-R patchname.patch

# 打补丁，将没有冲突的文件合并，将有冲突的文件标记出来，并生成对应的 .rej 文件

$ git apply --reject patchname.patch

# 将另一个分支上面的指定提交应用到当前分支上

$ git cherry-pick <commit>

# 将指定分支上面的最后一次提交应用到当前分支上

$ git cherry-pick banchname

# 将指定分支上所有修改应用到当前分支上

$ git rebase branchname

# 将第一个指定分支上所有修改应用到第二个分支上

$ git rebase branchname branchname

在 rebase 加上 -i 会提供交互式的变基操作。

# 恢复一个指定提交

$ git revert <commit>

# 调试

# 开始二分查找。

$ git bisect start

# 设置指定提交（默认是当前分支）为bad。

$ git bisect bad [<commit>]

# 设置指定提交（默认是当前分支）为good。

$ git bisect good [<commit>]

输入了上面三个命令后就会自动开始二分查找，我们之后只需要标记当前提交时bad还是good就行了，如果当前找的的提交时bad就输入 git bisect bad ，否则输入 git bisect good 直到找到有bug的提交。

# 查看指定文件所有的操作者，看看是谁错误地修改了该文件。

$ git blame filename

# 在工作目录中所有文件中搜索 keys。

$ git grep keys

# 在暂存区中所有文件中搜索 keys。

$ git grep --cached keys

# 远程仓库操作

# 查看所有与本地仓库关联的远程仓库。

$ git remote

# 查看所有与本地仓库关联的远程仓库，并显示 url。

$ git remote -v

# 参数 remote 通常都是缩写名 origin，

# 可以得到远程分支更为详细的信息以及 pull 和 push 相关提示信息。

$ git remote show [remote]

# 添加与本地仓库关联的远程仓库。

# 其中，shortname 一般是 origin，也可以是其他字符串，用来代替整个 url。

$ git remote add [shortname] [url]

如果拉的是 master 分支的代码，然后才创建自己的分支，可以使用 git remote update origin -p 更新远程分支

然后用 git checkout -b 本地分支名 origin/远程分支名 创建本地分支并且自动切换到自己的远程分支上

从master分支更新代码到本地分支:

$ git fetch origin master:temp

$ git merge temp

$ git branch -d temp

# 为远程仓库重命名。

$ git remote rename oldname newname

# 移除远程仓库。

$ git remote remove remote-name

# 推送本地修改到 origin。

$ git push

# 关联后，由于远程库是空的，使用该命令第一次推送 master 分支的所有内容，

# Git不但会把本地的master分支内容推送到远程新的master分支，

# 还会把本地的master分支和远程的master分支关联起来，

# 后续再推送的时候就可以省略后面三个参数了，其中参数 -u 代表上游（upstream）的意思。

$ git push -u origin master

# 推送本地仓库指定分支到远程仓库指定分支（默认是与本地分支同名的远程分支），

# 默认是不推送标签到远程仓库的，加上 –tags 就会推送标签。

$ git push remoterepository localbranch[:remotebranch] [--tags]

# 抓取远程库最新提交，拉取并合并。

$ git pull

# 拉取指定远程仓库指定分支到本地仓库指定分支（默认是当前分支）

$ git pull remoterepository branchname[:localbranch]

# 下载远程仓库 origin 到本地

$ git fetch origin

# 下载指定远程仓库到本地

$ git fetch [remote]

# 下载指定远程仓库指定分支到本地

$ git fetch [remote] [branch]

# 分支管理

# 列出本地当前所有分支，方便查看。当前分支前面会标有一个 \* 号。

$ git branch

# 新建一个分支，但依然停留在当前分支。

$ git branch [branch]

# 以特定提交为基创建分支。

$ git branch [branch] <commit>

# 普通删除分支（相对强制删除而言）。

# 一般情况下，先合并完分支，然后再删除，否则会删除失败，除非使用 -D 参数强制删除。

$ git branch -d [branch]

# 强行删除分支，尤其适用分支内容有了新的修改但还没有被合并的情况。

$ git branch -D [branch]

# 移动或者删除分支。

$ git branch -m [oldbranchname] newbranchname

# 强制移动或者删除分支。

$ git branch -M [oldbranchname] newbranchname

# 列出所有被跟踪的远程分支。

$ git branch -r

# 删除被跟踪的远程分支。

$ git branch -r -d [branch]

# 列出所有本地分支与被跟踪的远程分支。

$ git branch -a

# 建立本地分支和远程分支的关联。

$ git branch --set-upstream [branch] origin/[branch]

# 检查本地仓库分支与远程仓库分支的差异

$ git checkout

# 切换到指定分支，并更新工作区。

$ git checkout [branch]

# 切换到指定tag，如果存在tag与branch同名，优先切换分支。

$ git checkout [tag]

# 新建分支并切换到该分支上。

$ git checkout -b [branch]

# 强制新建分支并切换到该分支上。

$ git checkout -B [branch]

# 将指定分支合并到当前分支上(默认附带提交操作)，在 merge 后面使用 -m <message> 表示提交消息。

$ git merge [branch]

# 将指定提交合并到当前分支上

$ git merge <commit>

# 将 oldbranchname 分支合并到 newbranchname 分支上

$ git merge oldbranchname newbranchname

# 将指定分支上所有修改放入当前分支工作目录与暂存区，不提交

$ git merge --squash [branch]

# 将指定分支上所有修改放入当前分支工作目录与暂存区，并提交

$ git merge no--squash [branch]

# 参数 --no-ff 表示禁用 Fast forward 快进模式，用普通模式合并，

# 这样合并后的历史有分支，能看出来曾经做过合并，

# 而 Fast forwad 合并就看不出来曾经做过合并。

$ git merge --no-ff -m "message" [branch]

# 在 merge 命令执行之后出现冲突时执行该命令，用来解决合并的冲突，

# 冲突解决完了不要忘了提交

$ git mergetool

# 查看分支的合并情况，包括分支合并图、一行显示、提交校验码缩略显示。

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

# 储藏工作目录与暂存区的状态到堆栈中

# 把当前分支的工作现场储存起来，等以后恢复现场后继续工作。

# 一般适用于还没有 commit 的分支代码。

$ git stash

# 显示所有储藏信息。

$ git stash list

# 恢复到指定储藏的状态（默认是最近一次储藏的状态）

$ git stash apply [stash@{id}]

# 并从堆栈中移除指定储藏（默认是最近一次储藏）

$ git stash drop [stash@{id}]

# 用 git stash apply 命令恢复最近 stash 过的工作现场，

# 但是恢复后，stash 内容并不删除，用 git stash drop 命令来删除。

# apply 和 drop 后面都可以加上某一指定的 stash\_id。

$ git stash apply [stash@{id}] + git stash drop [stash@{id}]

# 恢复到指定储藏的状态（默认是最近一次储藏的状态），并从堆栈中移除该储藏

# 相当于git stash apply + git stash drop，

# 恢复回到工作现场的同时把 stash 内容也删除了。

$ git stash pop [stash@{id}]

# 在本地 master 上进行变基操作。

$ git rebase 目标分支（通常是 master）

# 标签管理

# 查看所有标签。这里标签不是按照时间列出，而是按照字母排序。

$ git tag

# 列出符合一定条件的标签

$ git tag -l 'v1.\*'

# 新建标签

$ git tag [tag]

# 在需要打标签的分支上创建一个轻量标签（lightweight），默认为 HEAD，

# 也可以指定一个 commit id。

$ git tag [tag] [commit id]

# 创建附注标签（annotated），

# 用 -a 指定标签名，-m 指定说明文字，也可以指定一个 commit id。

$ git tag -a [tag] -m "message" [commit id]

# 删除一个本地标签。

# 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。

$ git tag -d [tag]

# 查看标签信息。

$ git show [tag]

# 推送本地某个标签到远程，

# 默认情况下，git push 命令并不会推送标签到远程，必须显式推送。

$ git push origin [tag]

# 参数 –tags 表示一次性推送全部未推送到远程的本地标签，

# 当其他人从仓库中克隆或拉取，他们也能得到那些标签。

$ git push origin --tags

# 删除一个远程标签。要先从本地删除，再用该命令从远程删除。

$ git push origin :refs/tags/[tag]

# 查看指定提交或者指定标签（默认是最近一次提交）的注解标签信息。

$ git describe [<commit>|<tag>]

# 查看指定提交或者指定标签（默认是最近一次提交）的标签信息。

$ git desribe --tags [<commit>|<tag>]

# 查看指定提交或者指定标签（默认是最近一次提交）的引用信息。

$ git desribe --all [<commit>|<tag>]

# 设置

# 配置全局用户名，

# 若不要 –global ，或者将 global 改为 local ，则表示配置局部用户名。

$ git config --global user.name lavor

# 配置全局电子邮箱。

$ git config --global user.email lavor@qq.com

# 配置别名，为git命令配置别名，

# 还可以配置复合操作，比如 git config --global alias.pom 'push origin master'

# 注意复合操作中不可以带 - 与 ""

$ git config --global alias.cm commit

# 显示所有配置信息。

$ git config --list

# 显示帮助信息，会显示一些常见git命令及其意义

$ git help

# 其他

# 可能是 .gitignore 写得有问题时，使用该命令找出到底哪个命令写错了。

$ git check-ignore -v [file]

# 使用 git st 代替 git status 命令。

$ git config --global alias.st status

# 添加仓库到指定目录，使之成为本仓库的子模块。

$ git submodule add repository-url dir

# 初始化子模块。

$ git submodule init

# 更新子模块。

$ git submodule update