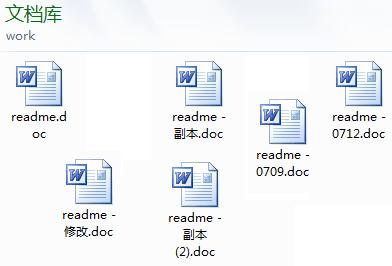
# Git简介

Git是目前世界上最先进的分布式版本控制系统(没有之一)。那什么是版本控制系统？

如果用Microsoft Word写过长篇大论，那你一定有这样的经历：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文件，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，最后你的Word文档变成了这样：



过了一周，你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，真麻烦。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，还不敢删，真郁闷。更要命的是，有些部分需要你的财务同事帮助填写，于是你把文件Copy到U盘里给她(也可能通过Email发送一份给她)，然后，你继续修改Word文件。一天后，同事再把Word文件传给你，此时，你必须想想，发给她之后到你收到她的文件期间，你作了哪些改动，得把你的改动和她的部分合并，真困难。

于是你想，如果有一个软件，不但能自动帮我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，岂不是很方便？

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：



## 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。

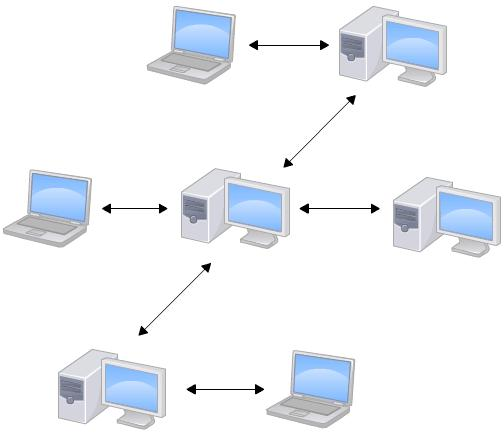


集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

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。

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。

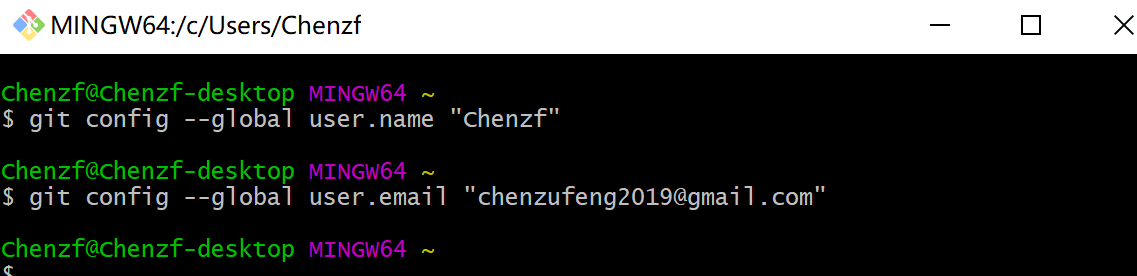
除了免费的外，还有收费的集中式版本控制系统，比如IBM的ClearCase(以前是Rational公司的，被IBM收购了)，特点是安装比Windows还大，运行比蜗牛还慢，能用ClearCase的一般是世界500强，他们有个共同的特点是财大气粗，或者人傻钱多。

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。由于其反人类的设计，连微软自己都不好意思用了。

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

在Windows上使用Git，可以从Git官网直接下载安装程序，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”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！



注意git config命令的--global参数，用了这个参数，表示你这台机器上所有的Git仓库都会使用这个配置，当然也可以对某个仓库指定不同的用户名和Email地址。

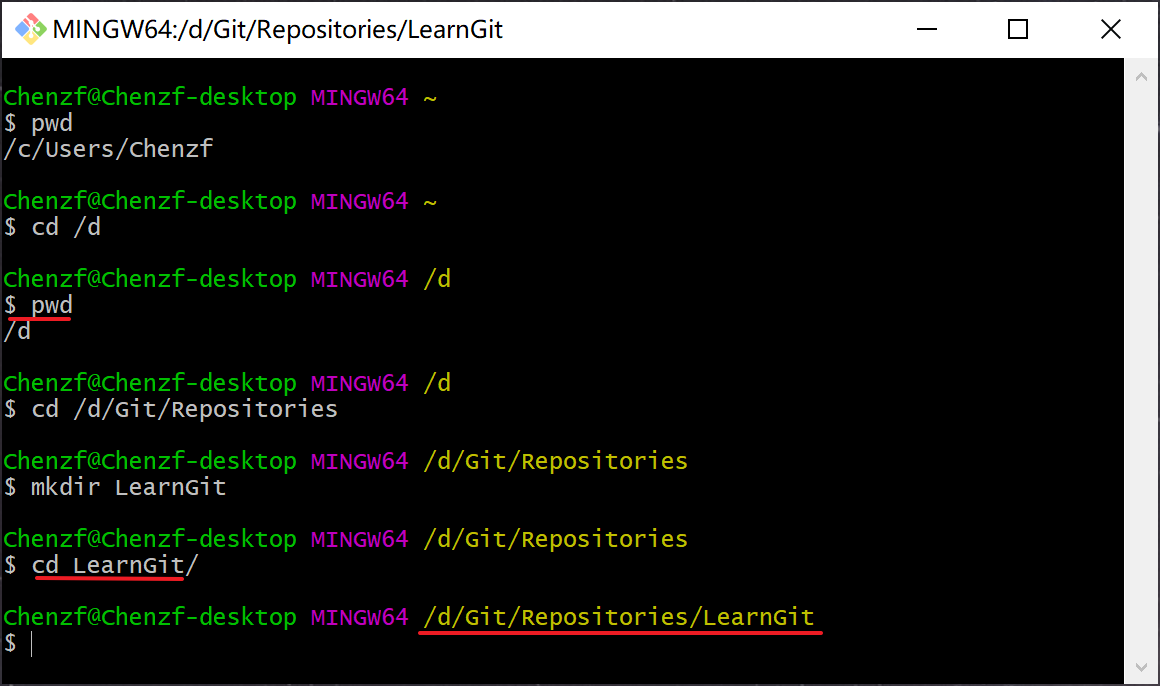
# Git命令

学习一些命令完成初始化一个仓库(repository)、开始或停止跟踪(track)文件、暂存(stage)或提交(commit)更改。

## 创建版本库

版本库又名仓库，英文名repository，你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：



第二步，通过*git init*命令把这个目录变成Git可以管理的仓库：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories

$ pwd

/d/Git/Repositories

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories

$ git init

Initialized empty Git repository in D:/Git/Repositories/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$

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)，细心的读者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。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。

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ ls

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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

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。

使用Windows要特别注意：

千万不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf(十六进制)的字符，你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“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可。

现在我们编写一个readme.txt文件。一定要放到*Repositories*目录下(子目录也行)，因为这是一个Git仓库，放到其他地方Git再厉害也找不到这个文件。

把一个文件放到Git仓库只需要两步。

第一步，用命令*git add*告诉Git，把文件添加到仓库：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

第二步，用命令*git commit*告诉Git，把文件提交到仓库：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "Write a readme file"

[master (root-commit) eba2181] Write a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

简单解释一下*git commit*命令，*-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(新添加的readme.txt文件)；2 insertions：插入了两行内容(readme.txt有两行内容)。

## 修改文件内容

修改了内容后，运行*git status*命令看看结果：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

(use "git restore <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*命令可以让我们时刻掌握仓库当前的状态，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。

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。需要用*git diff*这个命令：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git diff readme.txt

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

**index d8036c1..013b5bc 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.

\ No newline at end of file

*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，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，提交修改和提交新文件是一样的两步，第一步是git add：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

同样没有任何输出。在执行第二步git commit之前，我们再运行*git status*看看当前仓库的状态：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: readme.txt

*git status*告诉我们，将要被提交的修改包括readme.txt，下一步，就可以放心地提交了：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "add "distributed""

[master 515ff80] 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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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。

### 总结

修改了内容，但并未使用git add：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

modified: readme.txt

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

使用了git add，但并未使用git commit -m “…”

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: readme.txt

使用git commit -m “…”后：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "Conclusion of modify a file"

[master ec6754a] Conclusion of modify a file

1 file changed, 2 insertions(+), 1 deletion(-)

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

nothing to commit, working tree clean

## 版本回退

再次更改后：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

modified: readme.txt

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git diff

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

**index 013b5bc..99e0e11 100644**

**--- a/readme.txt**

**+++ b/readme.txt**

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

Git is a distributed version control system.

-Git is free software.

\ No newline at end of file

+Git is free software distributed under the GPL.

\ No newline at end of file

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "append GPL"

[master f930506] 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，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，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文件已经一共有三个版本被提交到Git仓库里了。在实际工作中，版本控制系统有命令可以告诉我们历史记录，在Git中，用*git log*命令查看：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git log

commit f930506d046721112c5602ffa474b2be3069df9e (HEAD -> master)

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Thu Feb 6 17:15:35 2020 +0800

append GPL

commit 515ff802e4aa02d93cf2fd0eb080e37f0491beea

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Thu Feb 6 17:08:02 2020 +0800

add distributed

commit eba2181c29099c70ca01603b1ebe2925fe32aee3

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Thu Feb 6 16:52:15 2020 +0800

Write a readme file

*git log*命令显示从最近到最远的提交日志，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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git log --pretty=oneline

f930506d046721112c5602ffa474b2be3069df9e (HEAD -> master) append GPL

515ff802e4aa02d93cf2fd0eb080e37f0491beea add distributed

eba2181c29099c70ca01603b1ebe2925fe32aee3 Write a readme file

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，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示。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。每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线。

如果准备把readme.txt回退到上一个版本，也就是add distributed的那个版本，怎么做呢？

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...，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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git reset --hard HEAD~1

HEAD is now at 515ff80 add distributed

看看readme.txt的内容是不是版本add distributed：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ cat readme.txt

Git is a distributed version control system.

Git is free software.

我们用*git log*再看看现在版本库的状态：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git log

commit 515ff802e4aa02d93cf2fd0eb080e37f0491beea (HEAD -> master)

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Thu Feb 6 17:08:02 2020 +0800

add distributed

commit eba2181c29099c70ca01603b1ebe2925fe32aee3

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Thu Feb 6 16:52:15 2020 +0800

Write 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，怎么办？

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到那个*append GPL*的commit id，于是就可以指定回到未来的某个版本：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git reset --hard f93050

HEAD is now at f930506 append GPL

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

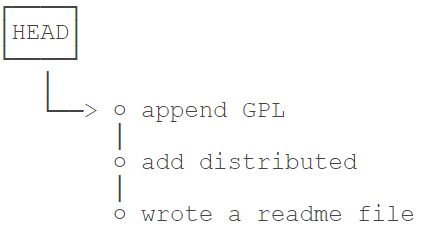
$ cat readme.txt

Git is a distributed version control system.

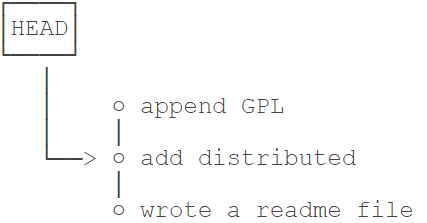
Git is free software distributed under the GPL.

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git reflog

f930506 (HEAD -> master) HEAD@{0}: reset: moving to f93050

515ff80 HEAD@{1}: reset: moving to HEAD~1

f930506 (HEAD -> master) HEAD@{2}: commit: append GPL

515ff80 HEAD@{3}: commit: add distributed

eba2181 HEAD@{4}: commit (initial): Write a readme file

从输出可知，*append GPL*的commit id是f930506。

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*。

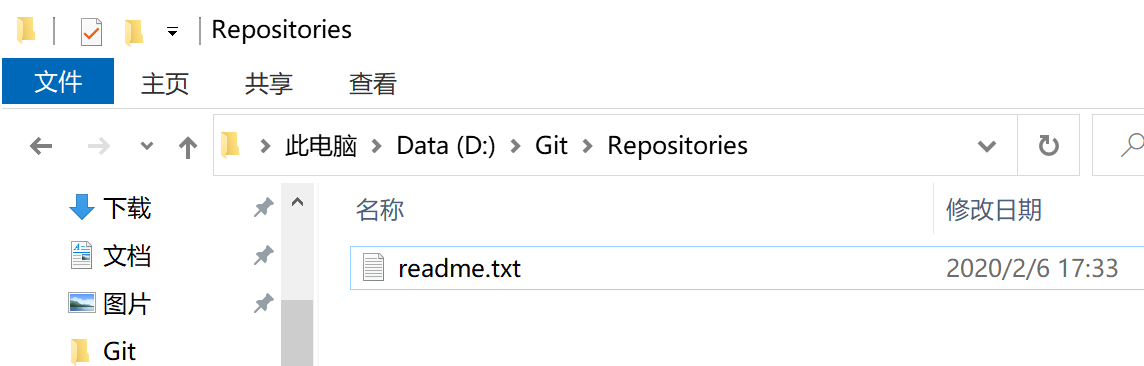
穿梭前，用*git log*可以查看提交历史，以便确定要回退到哪个版本。

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。

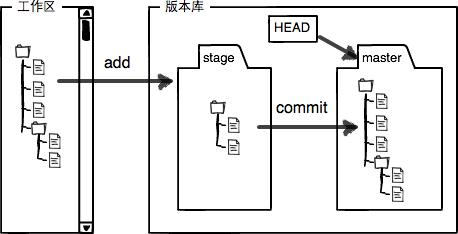
## 工作区和暂存区

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)：就是你在电脑里能看到的目录，比如Repositories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。



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。

先对readme.txt做个修改，比如加上一行内容：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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

(use "git restore <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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt LICENSE

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

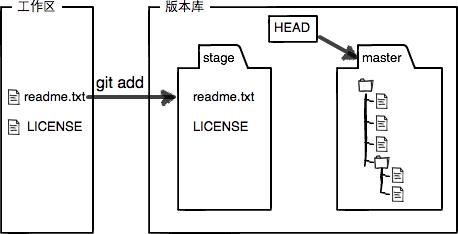
Changes to be committed:

(use "git restore --staged <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。

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "modify: readme.txt and add LICENSE to understand how stage works"

[master 11fe655] modify: readme.txt and add LICENSE to understand how stage works

2 files changed, 2 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：

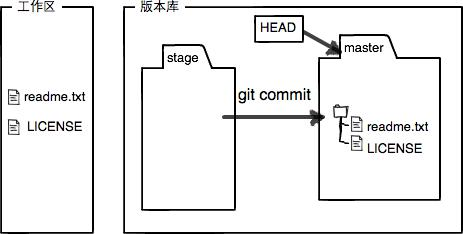
Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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：

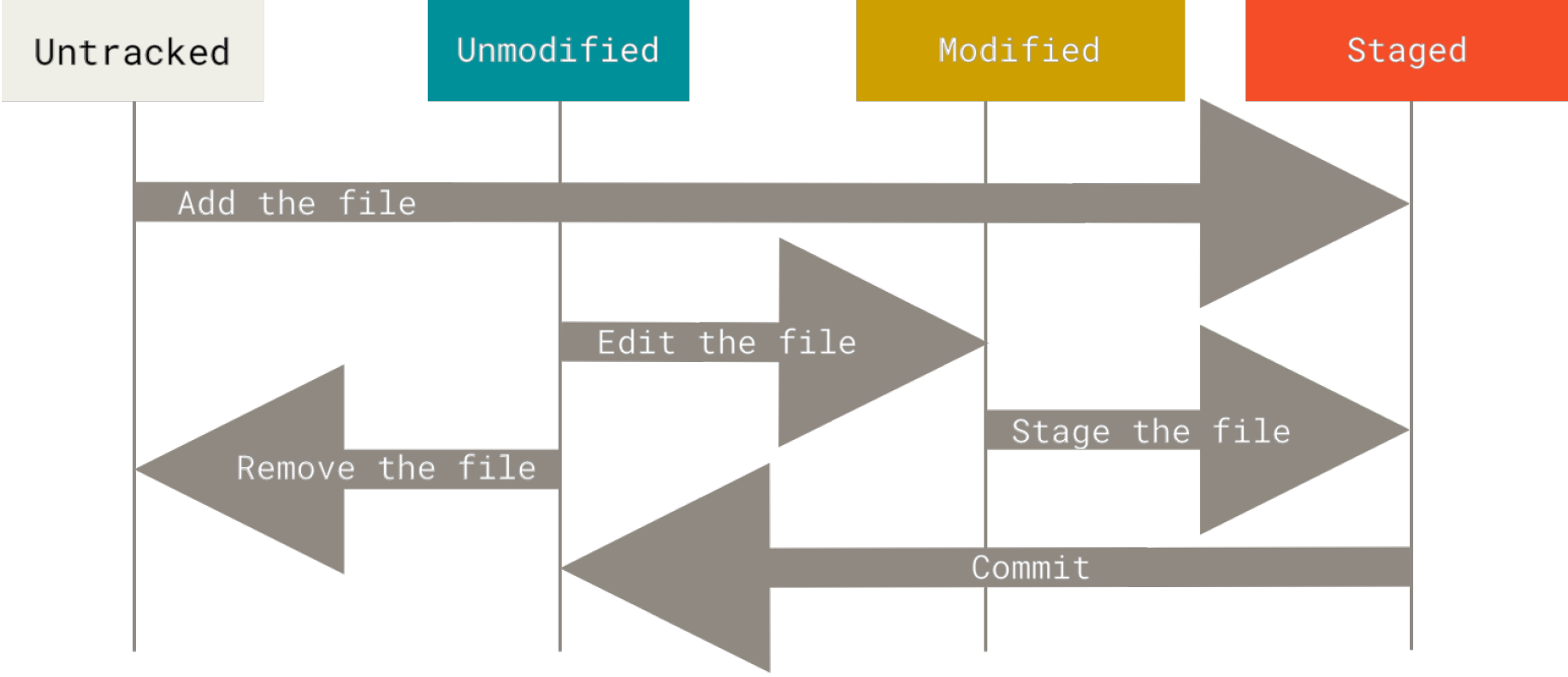


### 记录每次更新到仓库

工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。

已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能是未修改，已修改或已放入暂存区。

工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有被放入暂存区。初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。



#### 检查当前文件状态

可以用 git status 命令查看哪些文件处于什么状态：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git status

On branch master

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

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

nothing to commit, working tree clean

在项目下创建一个新的README文件。如果之前并不存在这个文件，使用git status命令，你将看到一个新的未跟踪文件：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ echo 'My Project' > README

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git status

On branch master

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

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

**Untracked** files:

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

README

nothing added to **commit** but untracked files present (use "git add" to track)

未跟踪的文件意味着 Git 在之前的快照(提交)中没有这些文件。

#### 跟踪新文件

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git add README

warning: LF will be replaced by CRLF in README.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git status

On branch master

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

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

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

说明是已暂存(staged)状态。 如果此时提交，那么该文件在你运行 git add 时的版本将被留存在历史记录中。

#### 暂存已修改文件

修改一个已被跟踪的文件。如果你修改了一个名为readme.txt的已被跟踪的文件，然后运行git status命令，会看到下面内容：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git status

On branch master

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

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

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: README

Changes not staged for commit:

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

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

modified: readme.txt

已跟踪文件的内容发生了变化，但还没有放到暂存区。要暂存这次更新，需要运行git add命令。

## 管理修改

为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

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。

为什么说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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: readme.txt

然后，再修改readme.txt：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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.

提交：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "git tracks changes"

[master b7e2add] git tracks changes

1 file changed, 2 insertions(+), 1 deletion(-)

提交后，再看看状态：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

modified: readme.txt

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

第二次的修改没有被提交！

回顾一下操作过程：第一次修改🡪 git add 🡪第二次修改🡪git commit

Git管理的是修改，当用*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，在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 -- readme.txt*命令可以查看工作区和版本库里面最新版本的区别：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git diff HEAD --readme.txt

usage: git diff [<options>] [<commit> [<commit>]] [--] [<path>...]

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git diff HEAD -- readme.txt

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

**index db28b2c..9a8b341 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.

\ No newline at end of file

+Git tracks changes of files.

\ No newline at end of file

那怎么提交第二次修改呢？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。

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

### git add到暂存区之前：丢弃工作区修改

如果你在readme.txt中添加了一行：My stupid boss still prefers SVN.。在你准备提交前，你猛然发现了stupid boss可能会让你丢掉这个月的奖金！既然错误发现得很及时，就可以很容易地纠正它。

你可以删掉最后一行，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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

modified: readme.txt

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

或者使用git checkout -- file可以丢弃工作区的修改：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git checkout -- readme.txt

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：一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

### git add到暂存区之后：撤销暂存区的修改

现在假定是凌晨3点，你不但写了一些胡话，还git add到暂存区了：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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.

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git checkout -- readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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.

在commit之前，你发现了这个问题。用git status查看一下，修改只是添加到了暂存区，还没有提交：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

modified: readme.txt

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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。

再用git status查看一下，现在暂存区是干净的，工作区有修改：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

modified: readme.txt

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

丢弃工作区的修改：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git checkout -- readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

nothing to commit, working tree clean

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ 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.

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？还记得版本回退一节吗？可以回退到上一个版本。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。还记得Git是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把stupid boss提交推送到远程版本库，你就真的惨了！

### 总结

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*。

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。

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，参考版本回退一节，不过前提是没有推送到远程库。

最新版本的git已经使用*git restore*代替了原来的reset和checkout命令了，如下：

*git resotre readme* //(使用 "git restore <文件>..." 丢弃工作区的改动)

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

*git restore --staged readme* //(使用 "git restore --staged <文件>..." 以取消暂存)

(use "git restore --staged <file>..." to unstage)

## 删除文件

先添加一个新文件：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git add TestDelete.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "add TestDelete.txt"

[master 3b70a2c] add TestDelete.txt

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 TestDelete.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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ rm TestDelete.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ ls

LICENSE readme.txt

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

Changes not staged for commit:

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

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

deleted: TestDelete.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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git rm TestDelete.txt

rm 'TestDelete.txt'

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git commit -m "remove TestDelete.txt"

[master a6df2bf] remove TestDelete.txt

1 file changed, 0 insertions(+), 0 deletions(-)

delete mode 100644 TestDelete.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git status

On branch master

nothing to commit, working tree clean

现在，文件就从版本库中被删除了。

另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git checkout -- TestDelete.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ ls

LICENSE readme.txt TestDelete.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”。从来没有被添加到版本库就被删除的文件，是无法恢复的！

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。

# 远程仓库

到目前为止，我们已经掌握的命令可以使你不用担心文件备份或者丢失的问题了。如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。本章开始介绍Git的杀手级功能之一：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。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。

你肯定会想，至少需要两台机器才能玩远程库不是？但是我只有一台电脑，怎么玩？其实一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。不过，现实生活中一台电脑上搞几个远程库完全没有意义，而且硬盘挂了会导致所有库都挂掉。

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。

完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫GitHub的神奇的网站，从名字就可以看出，这个网站就是提供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。

## SSH Key

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ ssh-keygen -t rsa -C "chenzufeng2019@gmail.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/Chenzf/.ssh/id\_rsa):

Created directory '/c/Users/Chenzf/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/Chenzf/.ssh/id\_rsa.

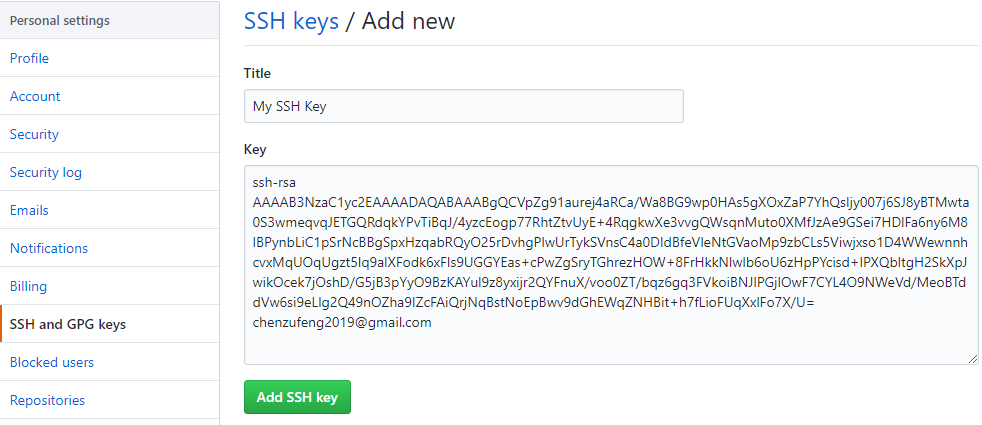
Your public key has been saved in /c/Users/Chenzf/.ssh/id\_rsa.pub.

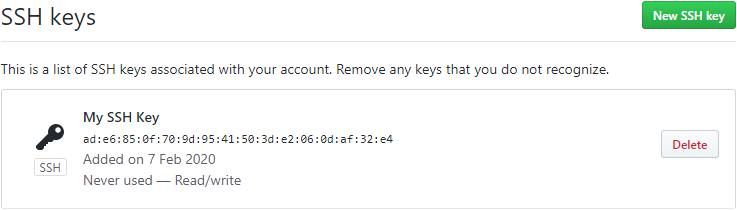
The key fingerprint is:

SHA256:QjgSEnOQWYaD4EGykO80ncaTCR5b/aGYrstL9uaKY8o chenzufeng2019@gmail.com

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：





为什么GitHub需要SSH Key呢？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。

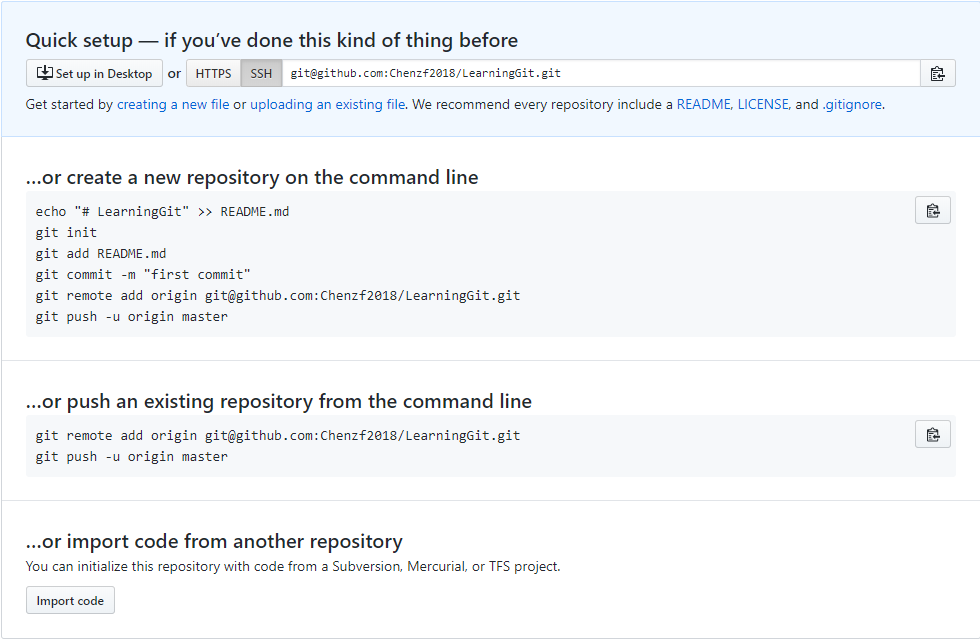
当然，GitHub允许你添加多个Key。假定你有若干电脑，你一会儿在公司提交，一会儿在家里提交，只要把每台电脑的Key都添加到GitHub，就可以在每台电脑上往GitHub推送了。

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。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

## 添加远程库

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。

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。目前，在GitHub上的这个LearningGit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。



*git remote add [shortname] [url]*

在本地的Repositories仓库下运行命令：

Chenzf@Chenzf-desktop MINGW64 /d/Git/Repositories (master)

$ git remote add origin git@github.com:Chenzf2018/learningGit.git

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。

### 推送到远程仓库

下一步，就可以把本地库的所有内容推送到远程库上：

*git push [remote-name] [branchname]*

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git push -u origin master

The authenticity of host 'github.com (13.250.177.223)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

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

Warning: Permanently added 'github.com,13.250.177.223' (RSA) to the list of known hosts.

Enumerating objects: 27, done.

Counting objects: 100% (27/27), done.

Delta compression using up to 8 threads

Compressing objects: 100% (23/23), done.

Writing objects: 100% (27/27), 2.42 KiB | 495.00 KiB/s, done.

Total 27 (delta 8), reused 0 (delta 0)

remote: Resolving deltas: 100% (8/8), done.

remote: This repository moved. Please use the new location:

remote: git@github.com:Chenzf2018/LearningGit.git

To github.com:Chenzf2018/learningGit.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。

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

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 (13.250.177.223)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

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

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,13.250.177.223' (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。如果你实在担心有人冒充GitHub服务器，输入yes前可以对照GitHub的RSA Key的指纹信息是否与SSH连接给出的一致。

### 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；

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git remote add origin git@github.com:Chenzf2018/Test.git

fatal: remote origin already exists.

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，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步！

### 新建另一个远程仓库

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit

$ git init

Initialized empty Git repository in D:/Git/TestGit/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git commit -m "add readme.txt"

[master (root-commit) 7af77b8] add readme.txt

1 file changed, 5 insertions(+)

create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ ~~git remote add origin git@github.com:Chenzf2018/TestGit.git~~

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git push origin master

Enumerating objects: 3, done.

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

Delta compression using up to 8 threads

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

Writing objects: 100% (3/3), 353 bytes | 353.00 KiB/s, done.

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

To github.com:Chenzf2018/TestGit.git

\* [new branch] master -> master

更改origin名字：

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit

$ git init

Initialized empty Git repository in D:/Git/TestGit/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git status

On branch master

No commits yet

Untracked files:

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

"Git\347\254\224\350\256\260.docx"

nothing added to commit but untracked files present (use "git add" to track)

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git add Git笔记.docx

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git commit -m "add Git笔记"

[master (root-commit) 455e419] add Git笔记

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 "Git\347\254\224\350\256\260.docx"

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git status

On branch master

nothing to commit, working tree clean

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ ~~git remote add origin git@github.com:Chenzf2018/TestGit.git~~

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git push -u orgin master

fatal: 'orgin' does not appear to be a git repository

fatal: Could not read from remote repository.

Please make sure you have the correct access rights

and the repository exists.

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git remote add TestGit git@github.com:Chenzf2018/TestGit.git

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git push -u TestGit master

Warning: Permanently added the RSA host key for IP address '52.74.223.119' to the list of known hosts.

Enumerating objects: 3, done.

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

Delta compression using up to 8 threads

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

Writing objects: 100% (3/3), 661.13 KiB | 1.90 MiB/s, done.

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

To github.com:Chenzf2018/TestGit.git

\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'TestGit'.

### 删除和重命名远程仓库

*git remote rm [别名]*

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git remote

TestGit

origin

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git remote rm TestGit

Chenzf@Chenzf-desktop MINGW64 /d/Git/TestGit (master)

$ git remote

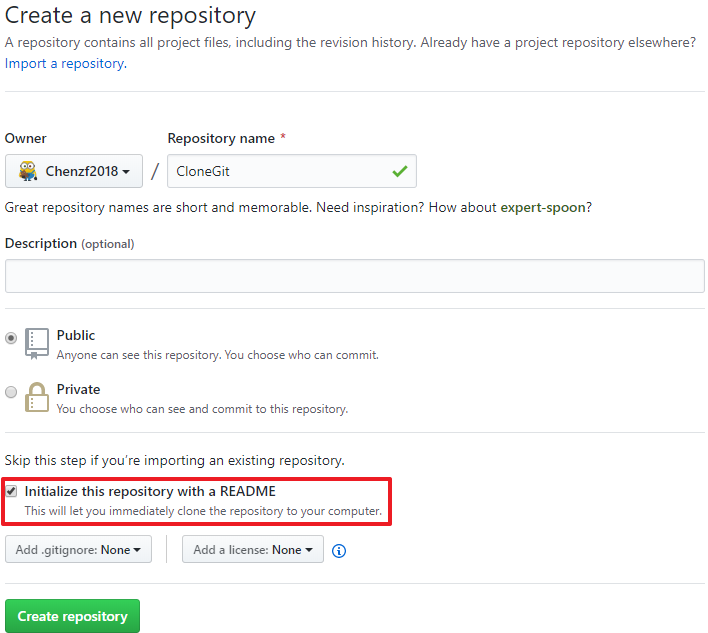
origin

如果想要重命名引用的名字可以运行*git remote rename*去修改一个远程仓库的简写名。这同样也会修改你的远程分支名字。那些过去引用oldname/master 的现在会引用rename/master。

## 从远程库克隆

上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。现在，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。

首先，登陆GitHub，创建一个新的仓库，名字叫TestGit：



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。

克隆仓库的命令是 *git clone [url]*。git clone命令会自动设置本地master分支跟踪克隆的远程仓库的master分支。

现在，远程库已经准备好了，下一步是用命令*git clone*克隆一个本地库：

Chenzf@Chenzf-desktop MINGW64 /d/Git

$ git clone git@github.com:Chenzf2018/CloneGit.git

Cloning into 'CloneGit'...

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

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

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

Chenzf@Chenzf-desktop MINGW64 /d/Git

$ cd CloneGit/

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ ls

README.md

### 克隆后更改内容

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ git add Git笔记.docx

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ git commit -m "add Git笔记.docx"

[master 7649ef3] add Git笔记.docx

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 "Git\347\254\224\350\256\260.docx"

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ ~~git push CloneGit master~~

fatal: 'CloneGit' does not appear to be a git repository

fatal: Could not read from remote repository.

Please make sure you have the correct access rights

and the repository exists.

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ git remote

origin

Chenzf@Chenzf-desktop MINGW64 /d/Git/CloneGit (master)

$ git push origin master

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 8 threads

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

Writing objects: 100% (3/3), 712.20 KiB | 938.00 KiB/s, done.

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

To github.com:Chenzf2018/CloneGit.git

6f109ed..7649ef3 master -> master

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。

GitHub给出的地址不止一个，还可以用*https://github.com/michaelliao/gitskills.gi*t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。

## 查看远程仓库

如果想查看你已经配置的远程仓库服务器，可以运行git remote命令。 它会列出你指定的每一个远程服务器的简写。如果你已经克隆了自己的仓库，那么至少应该能看到origin——这是Git给你克隆的仓库服务器的默认名字：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git remote

origin

可以指定选项-v，会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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git remote -v

origin git@github.com:Chenzf2018/learningGit.git (fetch)

origin git@github.com:Chenzf2018/learningGit.git (push)

# 分支管理

分支在实际中有什么用呢？

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。可以把你的工作从开发主线上分离开来，以免影响开发主线。

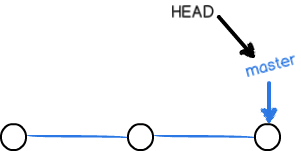
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

## 创建与合并分支

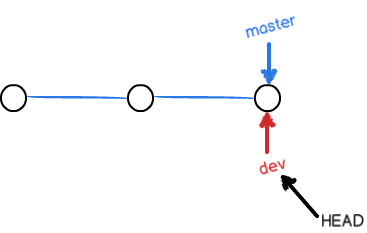
在版本回退里，已经知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支(当前分支的别名)。

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：



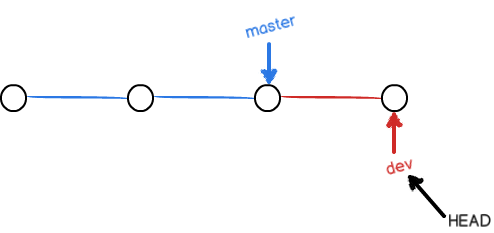
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

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！

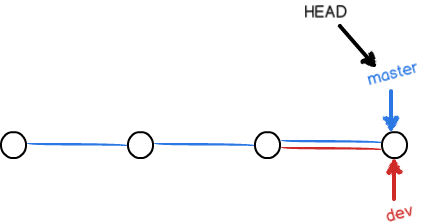
从现在开始，对工作区的修改和提交就是针对dev分支了，比如新提交一次后，dev指针往前移动一步，而master指针不变：



假如我们在dev上的工作完成了，就可以把dev合并到master上。

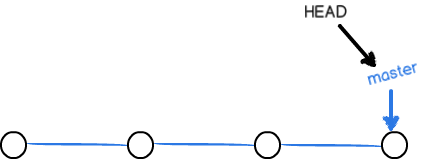
Git怎么合并呢？

最简单的方法，就是直接把master指向dev的当前提交，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：



### 创建分支dev

首先，创建dev分支，然后切换到dev分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git checkout -b dev

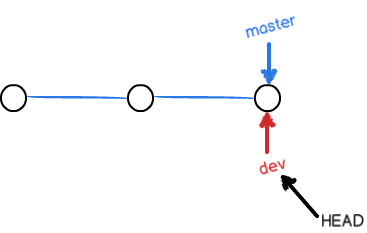
Switched to a new branch 'dev'

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'



可以使用*git log --oneline --decorate*查看各分支当前所指的对象：

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (master)

$ git log --oneline --decorate

804a711 (HEAD -> master) add hello.py in master branch

7466e4d merge bug fix 101

e97502a fix bug 101: Git is a...

c3bce63 merge with no-ff

9945119 add 分支管理策略

48e80dc conflict fixed

794f0b4 simple & quick

ab499f9 modify readme.txt

e23054d Creating dev1 branch

9bb5d48 restore read.txt

642dc37 Creating dev1 branch

5f6b7b1 合并失败

......

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (master)

$ git switch mywork

Switched to branch 'mywork'

Your branch and 'origin/master' have diverged,

and have 2 and 10 different commits each, respectively.

(use "git pull" to merge the remote branch into yours)

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (mywork)

$ git log --oneline --decorate

e27ea14 (HEAD -> mywork) add otherfile.txt

49171cc add file.txt

804a711 (master) add hello.py in master branch

然后，用git branch命令查看当前分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ 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，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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git commit -m "branch test"

[dev 8ba6421] branch test

1 file changed, 2 insertions(+), 1 deletion(-)

现在，dev分支的工作完成，我们就可以切换回master分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git checkout master

Switched to branch 'master'

Your branch is up to date with 'origin/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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ 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.

Creating a new branch is quick.

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git checkout master

Switched to branch 'master'

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ cat readme.txt

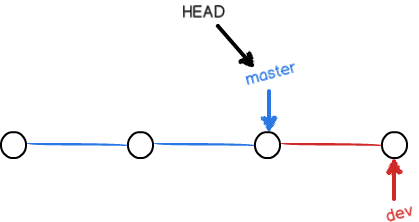
Git is a distributed version control system.

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.



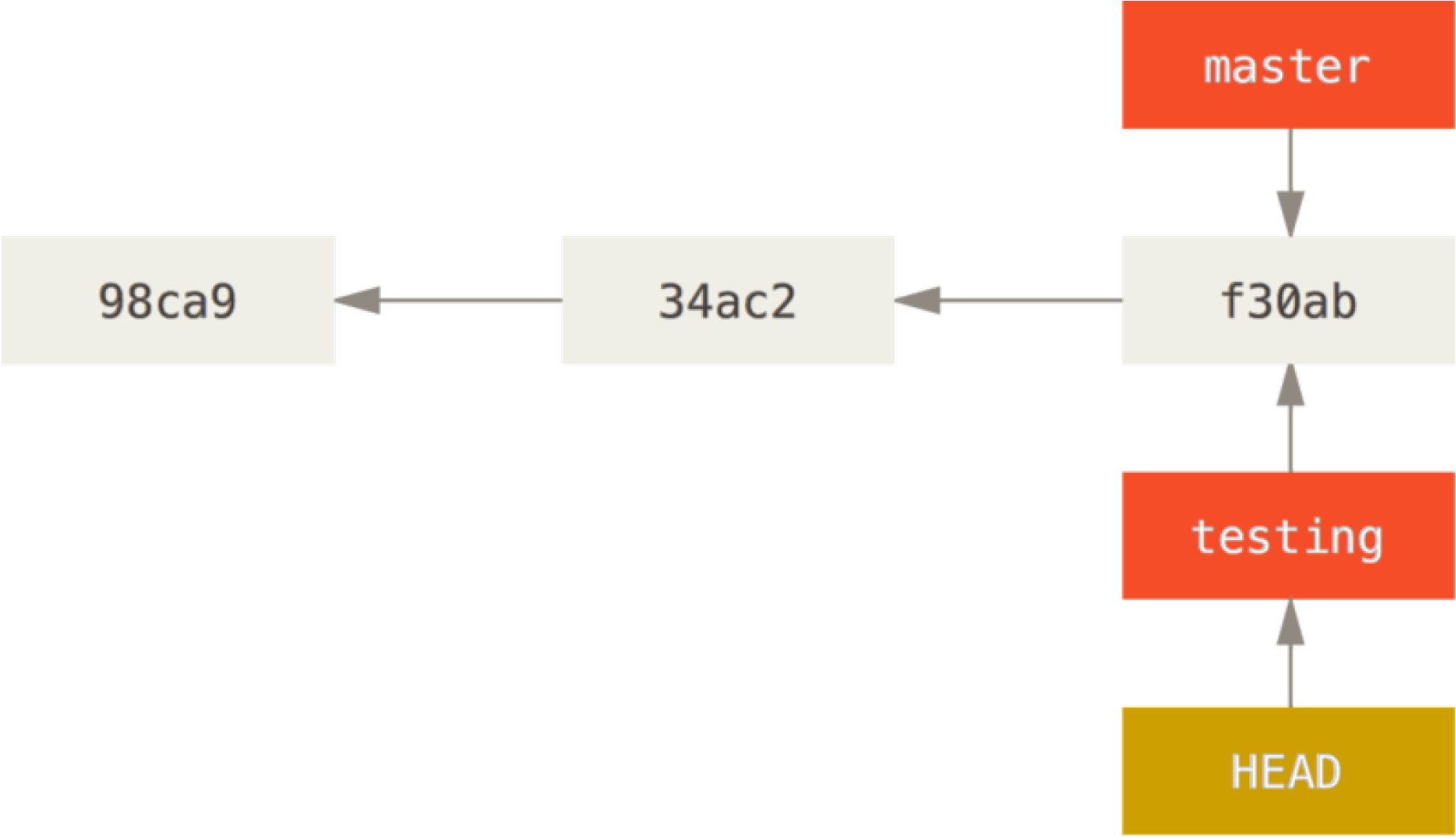
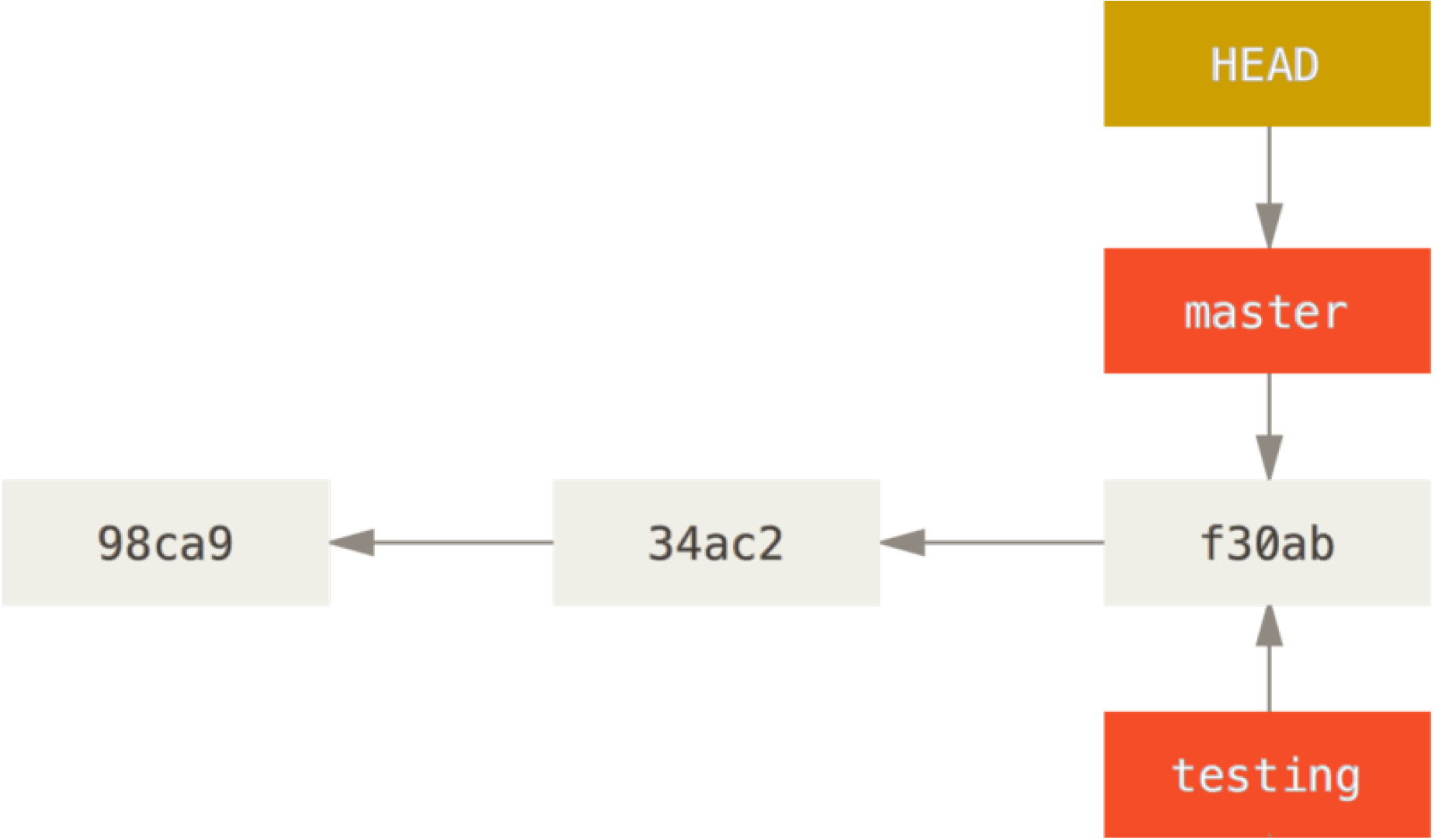
### switch

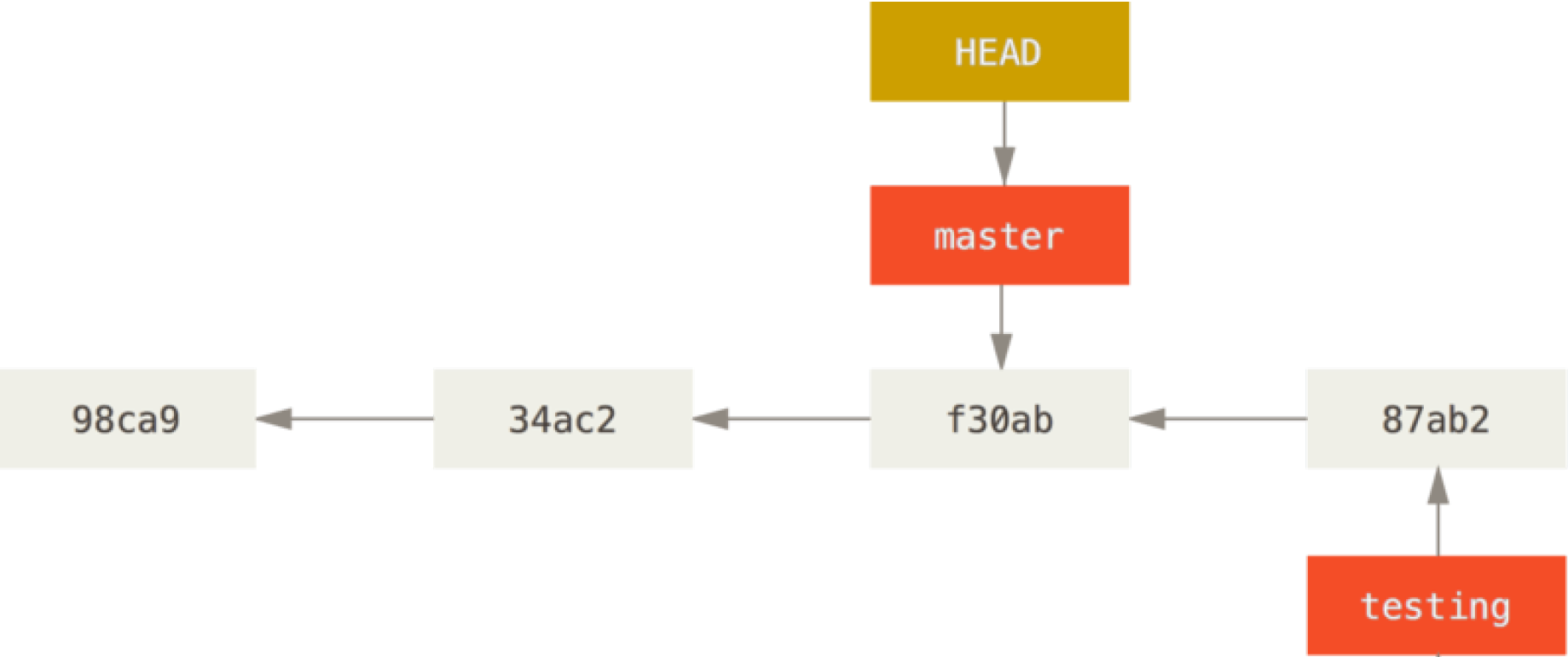
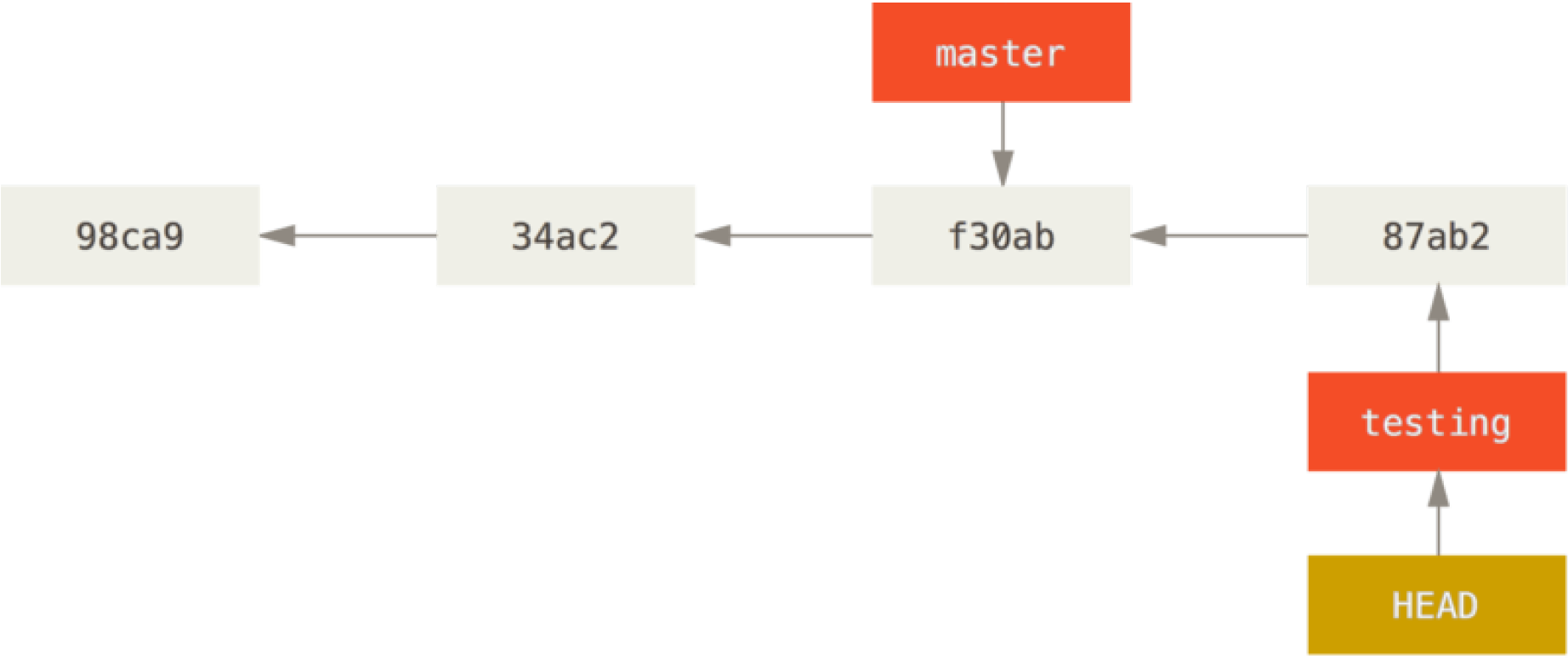
切换分支使用*git checkout <branch>*，而前面讲过的撤销修改则是*git checkout -- <file>*，同一个命令，有两种作用，确实有点令人迷惑。

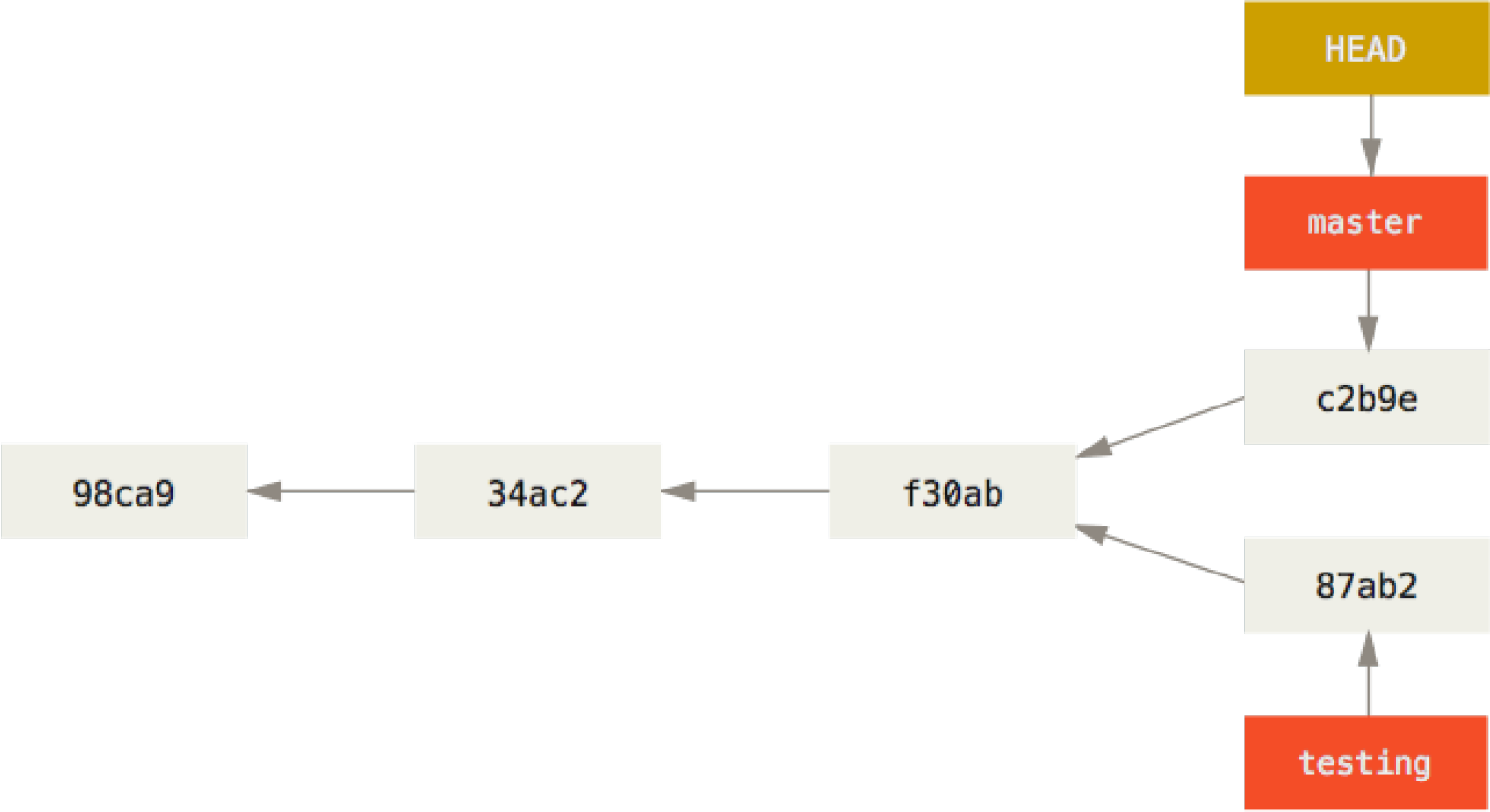
实际上，切换分支这个动作，用switch更科学。因此，最新版本的Git提供了新的git switch命令来切换分支：

创建并切换到新的dev分支，可以使用：$ git switch -c dev

直接切换到已有的master分支，可以使用：$ git switch master







$ git log --oneline --decorate --graph --all

\* c2b9e (HEAD, master) made other changes

| \* 87ab2 (testing) made a change

|/

\* f30ab add feature #32 - ability to add new formats to the

\* 34ac2 fixed bug #1328 - stack overflow under certain conditions

\* 98ca9 initial commit of my project

由于 Git 的分支实质上仅是包含所指对象校验和(长度为40的SHA-1值字符串)的文件，所以它的创建和销毁都异常高效。创建一个新分支就相当于往一个文件中写入41个字节(40个字符和1个换行符)。

### 合并分支

把dev分支的工作成果合并到master分支上：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git merge dev1

Updating 5f6b7b1..642dc37

Fast-forward

readme.txt | 3 ++-

1 file changed, 2 insertions(+), 1 deletion(-)

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。当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

### 删除分支

合并完成后，就可以放心地删除dev分支了：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git branch -d dev1

Deleted branch dev1 (was e23054d).

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ 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。

### 小结

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>或者git switch <name>

创建+切换分支：git checkout -b <name>或者git switch -c <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

## 解决冲突

合并分支往往也不是一帆风顺的。准备新的feature1分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch -c feature1

Switched to a new branch 'feature1'

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature1)

修改readme.txt最后一行，改为：Creating a new branch: feature1 is quick AND simple.。在feature1分支上提交：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature1)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature1)

$ git commit -m "modify readme.txt"

[feature1 ab499f9] modify readme.txt

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

切换到master分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature1)

$ git switch master

Switched to branch 'master'

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

(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分支上把readme.txt文件的最后一行改为：Creating a new branch: feature1 is simple & quick.。提交：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git add readme.txt

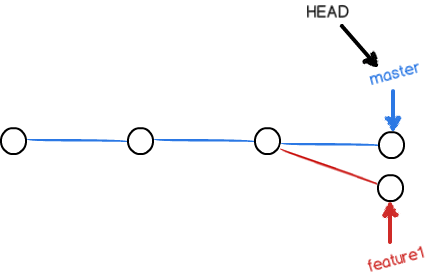
Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git commit -m "simple & quick"

[master 794f0b4] simple & quick

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ 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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master|MERGING)

$ git status

On branch master

Your branch is ahead of 'origin/master' by 4 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：

<<<<<<< HEAD  
Creating a new branch: feature1 is simple & quick.  
=======  
Creating a new branch: feature1 is quick AND simple.  
>>>>>>> feature1

Git用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，我们修改如下后保存：Creating a new branch: feature1 is simple and quick.。然后提交：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master|MERGING)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master|MERGING)

$ git commit -m "conflict fixed"

[master 48e80dc] conflict fixed

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git merge feature1

Already up to date.

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch feature1

Switched to branch 'feature1'

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature1)

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

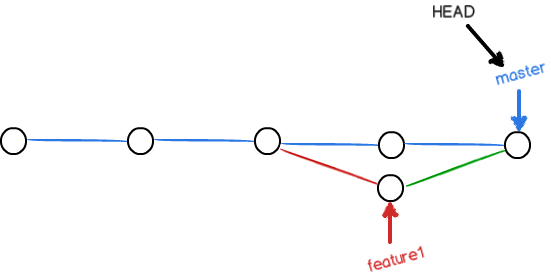
Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

Creating a new branch: feature1 is quick AND simple.

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

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

\* 48e80dc (HEAD -> master) conflict fixed

|\

| \* ab499f9 (feature1) modify readme.txt

\* | 794f0b4 simple & quick

|/

\* e23054d Creating dev1 branch

\* 9bb5d48 restore read.txt

\* 642dc37 Creating dev1 branch

\* 5f6b7b1 (origin/master) 合并失败

|\

| \* 656bbe6 add branch dev1

\* | 8ba6421 branch test

|/

\* 97bc46d add TestDelete.txt again

\* a6df2bf remove TestDelete.txt

\* 3b70a2c add TestDelete.txt

\* ec6754a Conclusion of modify a file

\* faa9fdf delete My stupid boss still prefers SVN.

\* 1a197d7 add of files.

\* b7e2add git tracks changes

\* 11fe655 modify: readme.txt and add LICENSE to understand how stage works

\* f930506 append GPL

\* 515ff80 add distributed

\* eba2181 Write a readme file

最后，删除feature1分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git branch -d feature1

Deleted branch feature1 (was ab499f9).

### 小结

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。

## 再探合并分支与解决冲突

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge

$ git init

Initialized empty Git repository in D:/Git/GitMerge/.git/

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --oneline --decorate --graph --all

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ echo 'My project' > README

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git add README

warning: LF will be replaced by CRLF in README.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git commit -m "add README"

[master (root-commit) c0be1c4] add README

1 file changed, 1 insertion(+)

create mode 100644 README

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --graph --oneline --all --decorate

\* c0be1c4 (HEAD -> master) add README

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git commit -m "add readme.txt"

[master 05ee3e4] add readme.txt

1 file changed, 1 insertion(+)

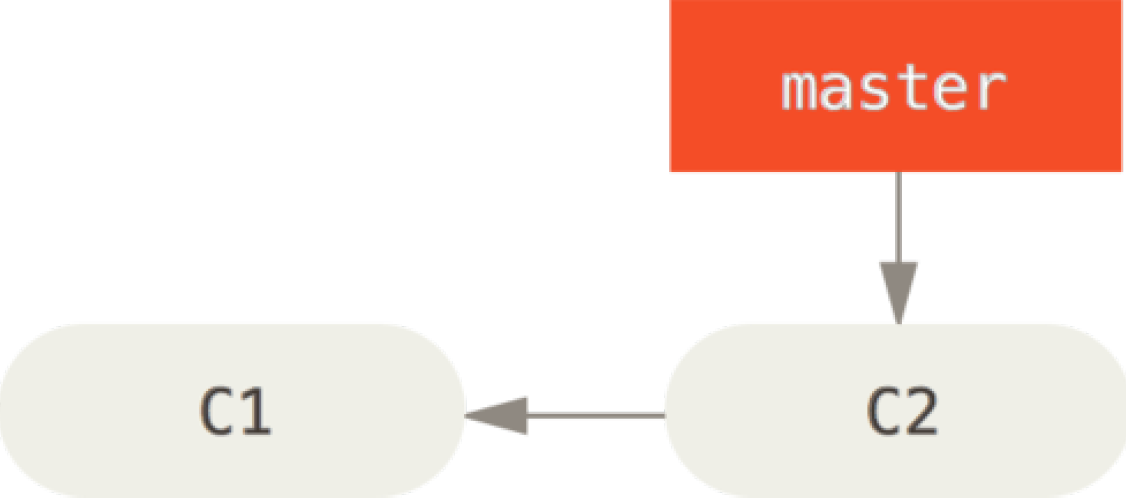
create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --oneline --graph --all --decorate

\* 05ee3e4 (HEAD -> master) add readme.txt

\* c0be1c4 add README



Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git checkout -b iss53

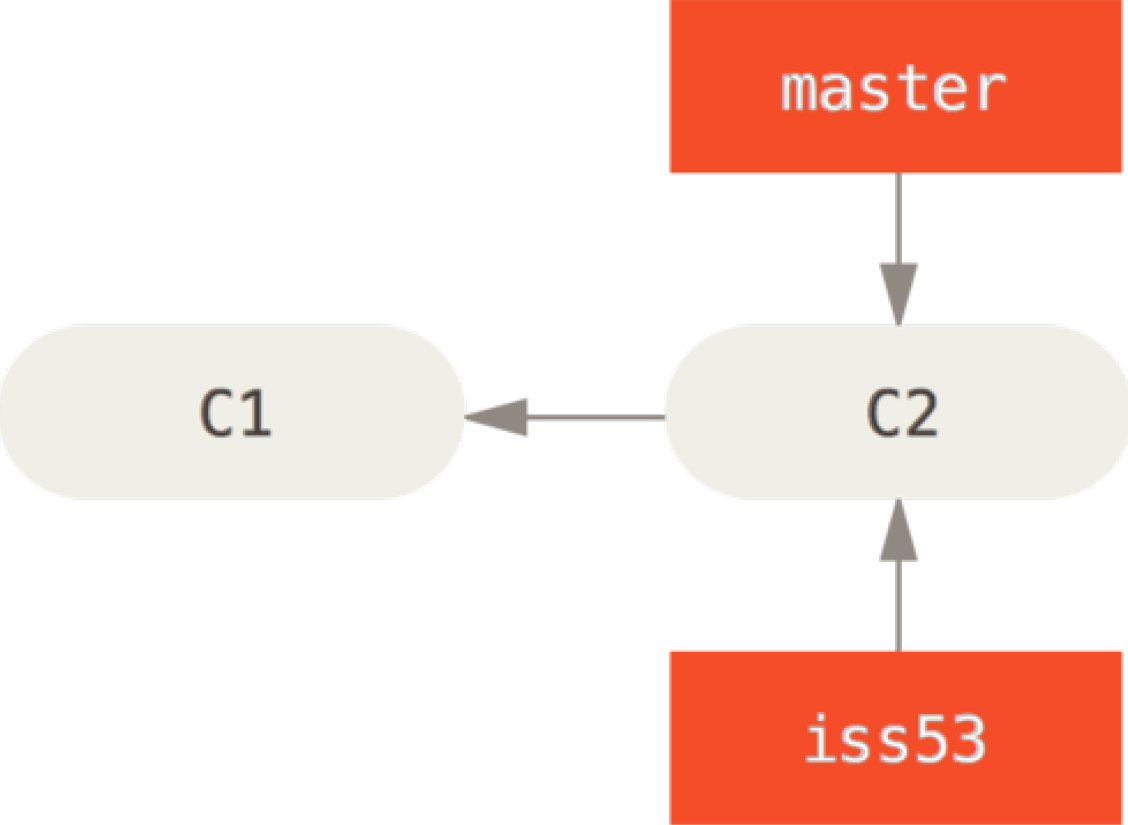
Switched to a new branch 'iss53'

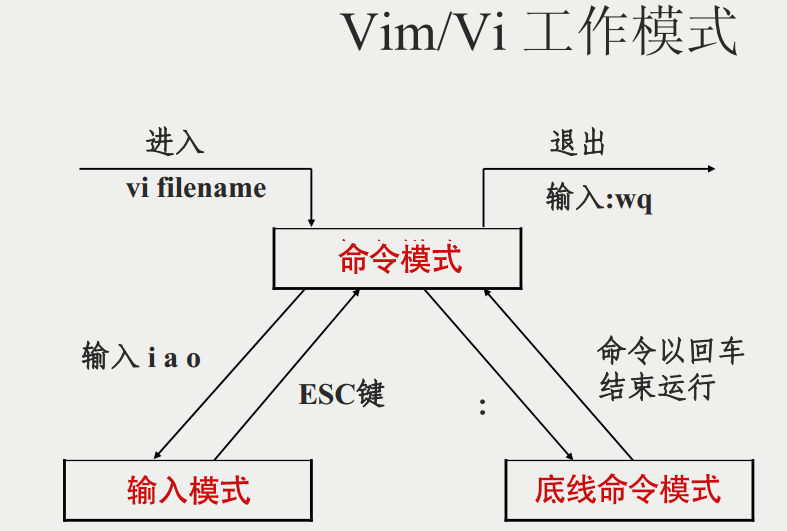
Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git log --graph --all --decorate --oneline

\* 05ee3e4 (HEAD -> iss53, master) add readme.txt

\* c0be1c4 add README





Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ vim index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git add index.html

warning: LF will be replaced by CRLF in index.html.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git commit -m "add index.html"

[iss53 05e8c2e] add index.html

1 file changed, 1 insertion(+)

create mode 100644 index.html

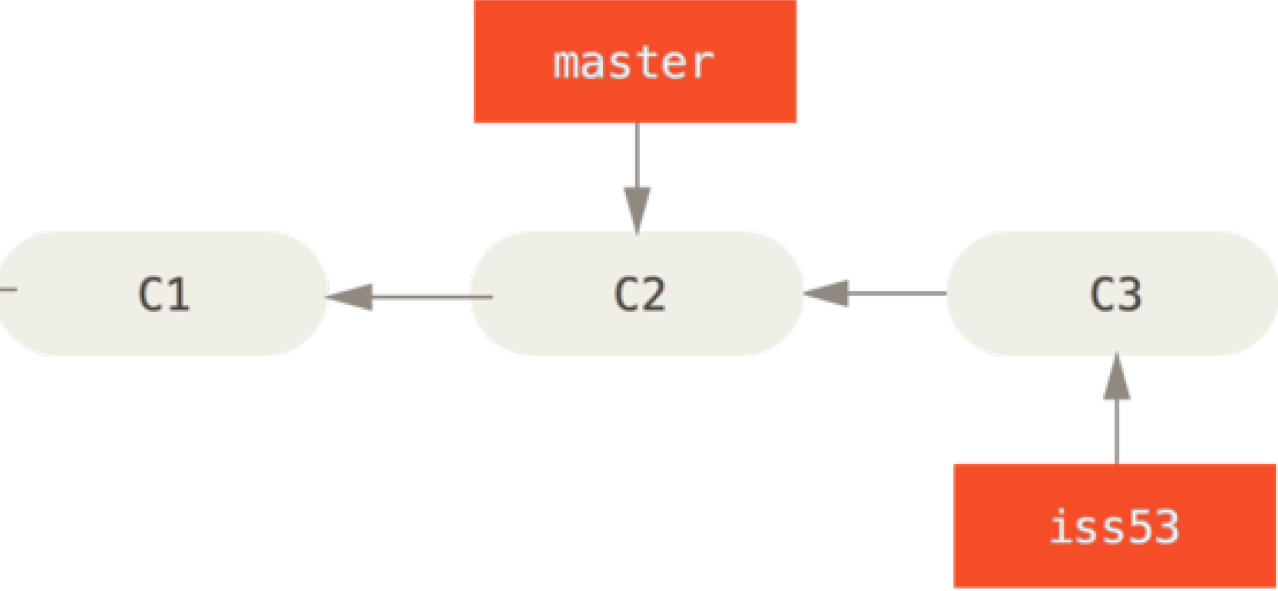
Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git log --graph --oneline --decorate --all

\* 05e8c2e (HEAD -> iss53) add index.html

\* 05ee3e4 (master) add readme.txt

\* c0be1c4 add README



Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git checkout master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git checkout -b hotfix

Switched to a new branch 'hotfix'

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (hotfix)

$ vim index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (hotfix)

$ git add index.html

warning: LF will be replaced by CRLF in index.html.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (hotfix)

$ git commit -m "add a heading in index.html"

[hotfix 812f2a2] add a heading in index.html

1 file changed, 1 insertion(+)

create mode 100644 index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (hotfix)

$ git log --graph --oneline --decorate --all

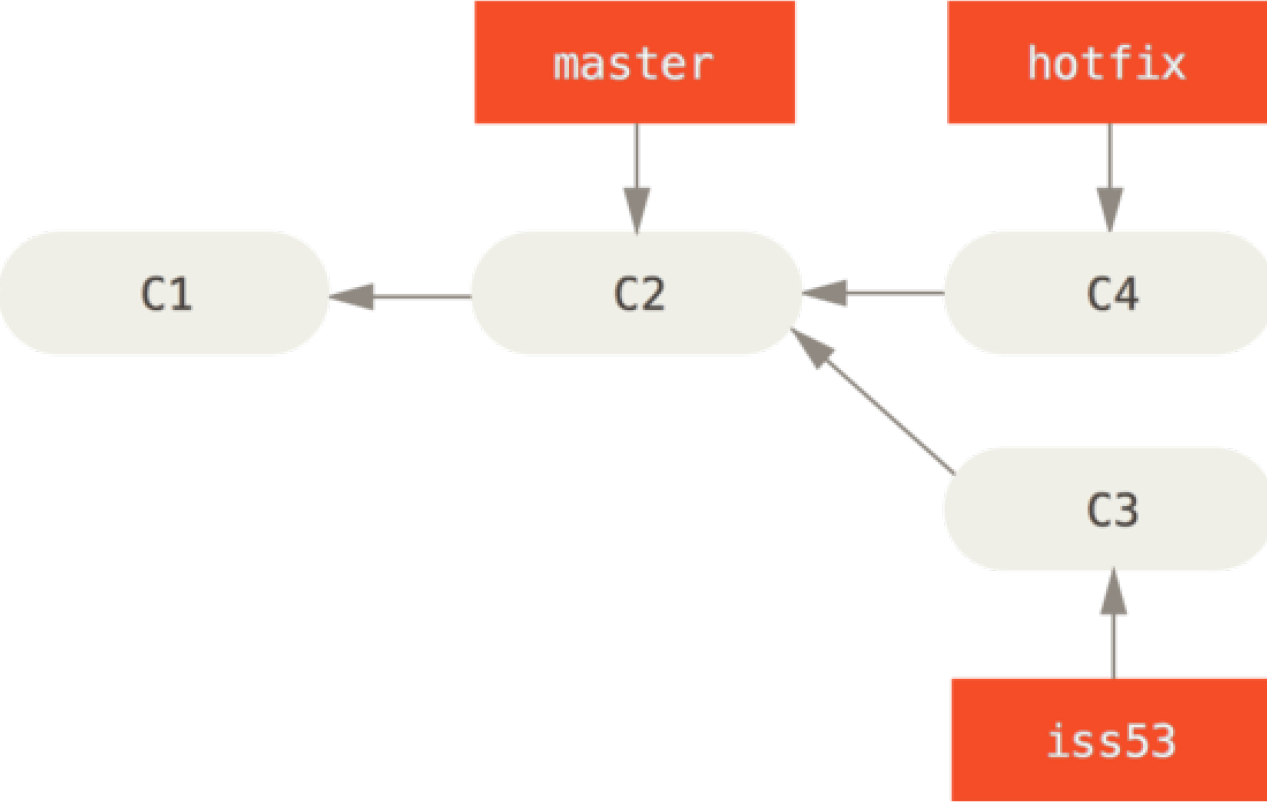
\* 812f2a2 (HEAD -> hotfix) add a heading in index.html

| \* 05e8c2e (iss53) add index.html

|/

\* 05ee3e4 (master) add readme.txt

\* c0be1c4 add README



将hotfix合并回你的 master 分支来部署到线上：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (hotfix)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --graph --oneline --decorate --all

\* 812f2a2 (hotfix) add a heading in index.html

| \* 05e8c2e (iss53) add index.html

|/

\* 05ee3e4 (HEAD -> master) add readme.txt

\* c0be1c4 add README

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git merge hotfix

Updating 05ee3e4..812f2a2

Fast-forward

index.html | 1 +

1 file changed, 1 insertion(+)

create mode 100644 index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --graph --oneline --decorate --all

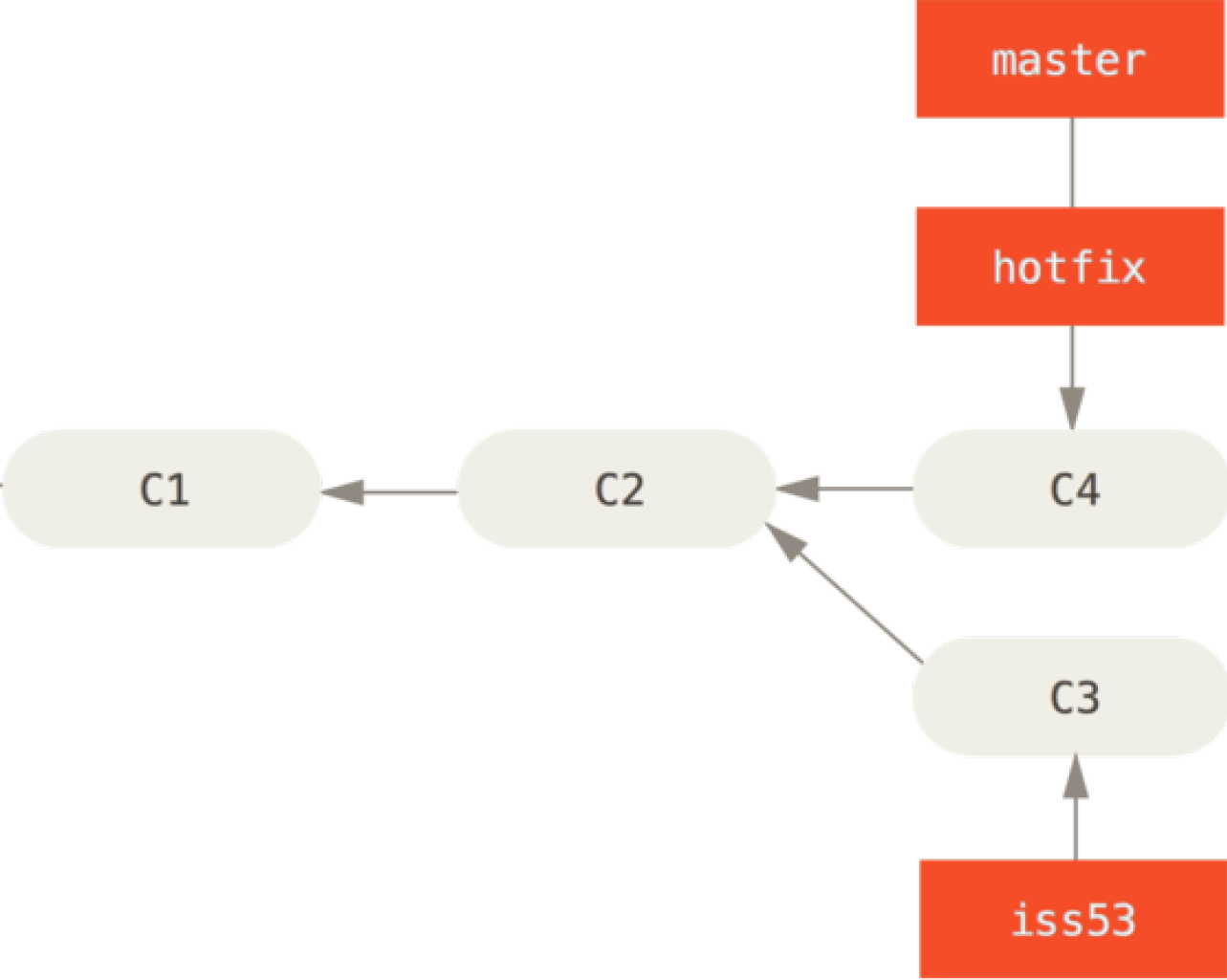
\* 812f2a2 (HEAD -> master, hotfix) add a heading in index.html

| \* 05e8c2e (iss53) add index.html

|/

\* 05ee3e4 add readme.txt

\* c0be1c4 add README



由于当前master分支所指向的提交是你当前提交(有关hotfix的提交)的直接上游，所以Git只是简单的将指针向前移动。换句话说，当你试图合并两个分支时，如果顺着一个分支走下去能够到达另一个分支，那么Git在合并两者的时候，只会简单的将指针向前推进(指针右移)，因为这种情况下的合并操作没有需要解决的分歧——这就叫做“快进(fast-forward)”。

删除分支：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git branch -d hotfix

Deleted branch hotfix (was 812f2a2).

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --graph --all --decorate --oneline

\* 812f2a2 (HEAD -> master) add a heading in index.html

| \* 05e8c2e (iss53) add index.html

|/

\* 05ee3e4 add readme.txt

\* c0be1c4 add README

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git checkout iss53

Switched to branch 'iss53'

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ vim index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ cat index.html

<h2>This is a heading</h2>

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git add index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git commit -m "add heading2 in index"

[iss53 706e482] add heading2 in index

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

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (iss53)

$ git log --graph --oneline --all --decorate

\* 706e482 (HEAD -> iss53) add heading2 in index

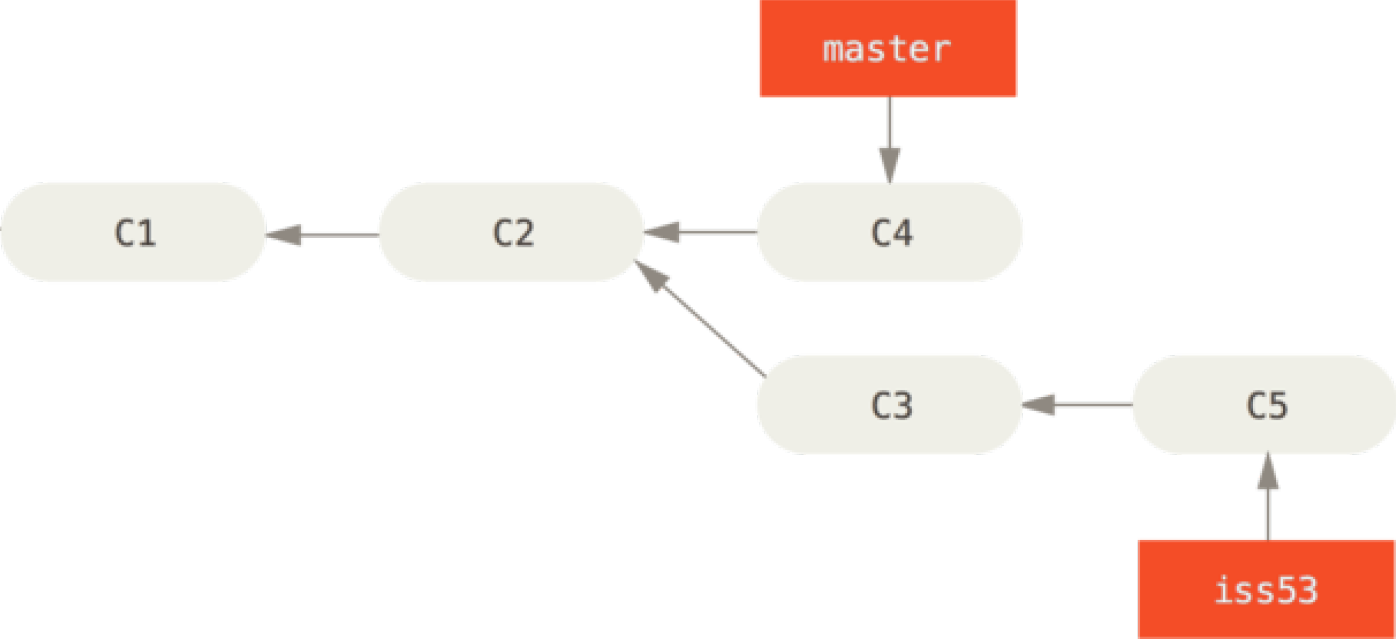
\* 05e8c2e add index.html

| \* 812f2a2 (master) add a heading in index.html

|/

\* 05ee3e4 add readme.txt

\* c0be1c4 add README



### 遇到冲突时的分支合并

如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，Git 就没法干净的合并它们。如果你对#53问题的修改和有关hotfix的修改都涉及到同一个文件的同一处，在合并它们的时候就会产生合并冲突：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git merge iss53

CONFLICT (add/add): Merge conflict in index.html

Auto-merging index.html

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

此时Git做了合并，但是没有自动地创建一个新的合并提交。Git会暂停下来，等待你去解决合并产生的冲突。你可以在合并冲突后的任意时刻使用git status命令来查看那些因包含合并冲突而处于未合并(unmerged)状态的文件：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master|MERGING)

$ git status

On branch master

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 added: index.html

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

任何因包含合并冲突而有待解决的文件，都会以未合并状态标识出来。Git会在有冲突的文件中加入标准的冲突解决标记，这样你可以打开这些包含冲突的文件然后手动解决冲突。出现冲突的文件会包含一些特殊区段，看起来像下面这个样子：

<<<<<<< HEAD

<h1>This is a heading</h1>

=======

<h2>This is a heading</h2>

>>>>>>> iss53

这表示HEAD所指示的版本(也就是你的master分支所在的位置，因为你在运行merge命令的时候已经检出到了这个分支)在这个区段的上半部分(======= 的上半部分)，而iss53分支所指示的版本在=======的下半部分。为了解决冲突，你必须选择使用由 =======分割的两部分中的一个，或者你也可以自行合并这些内容：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master|MERGING)

$ vim index.html

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master|MERGING)

$ cat index.html

<h1>This is a heading</h1>

<h2>This is a heading</h2>

解决了所有文件里的冲突之后，对每个文件使用git add命令来将其标记为冲突已解决。 一旦暂存这些原本有冲突的文件，Git 就会将它们标记为冲突已解决。

### 三方合并

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git merge iss53

Already up to date.

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git log --graph --all --decorate --oneline

\* 6dc93f9 (HEAD -> master) fix conflict

|\

| \* 706e482 (iss53) add heading2 in index

| \* 05e8c2e add index.html

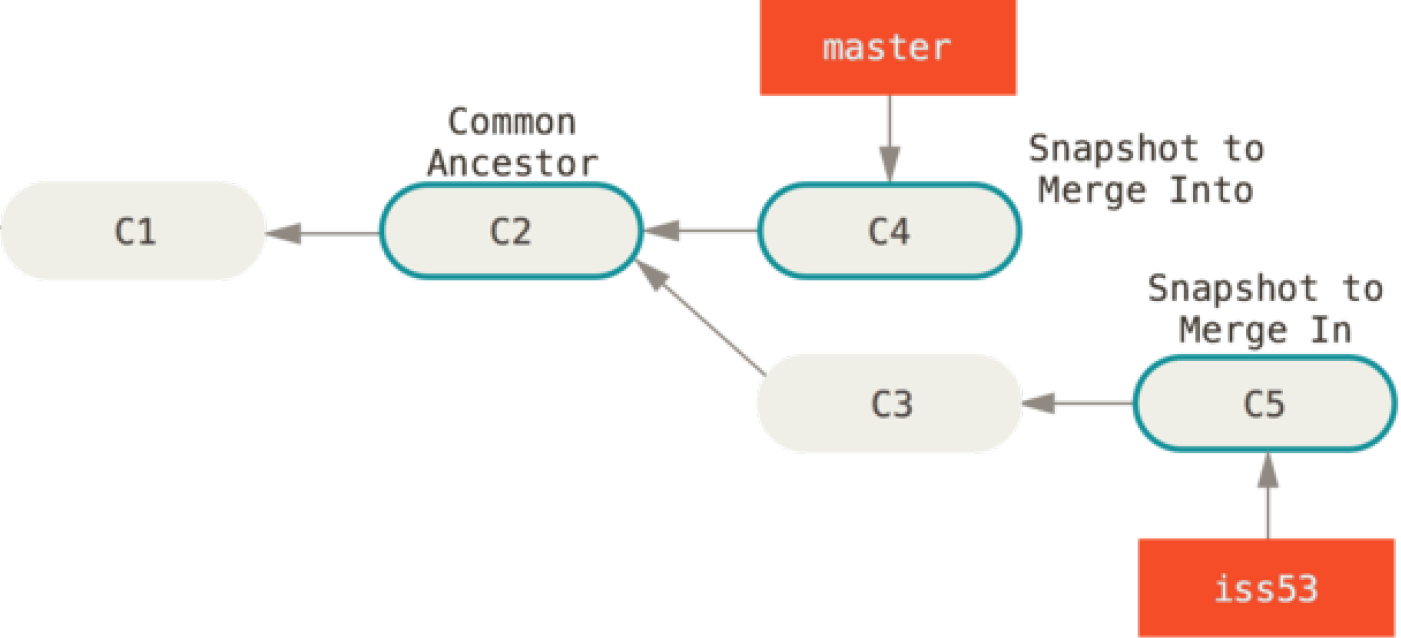
\* | 812f2a2 add a heading in index.html

|/

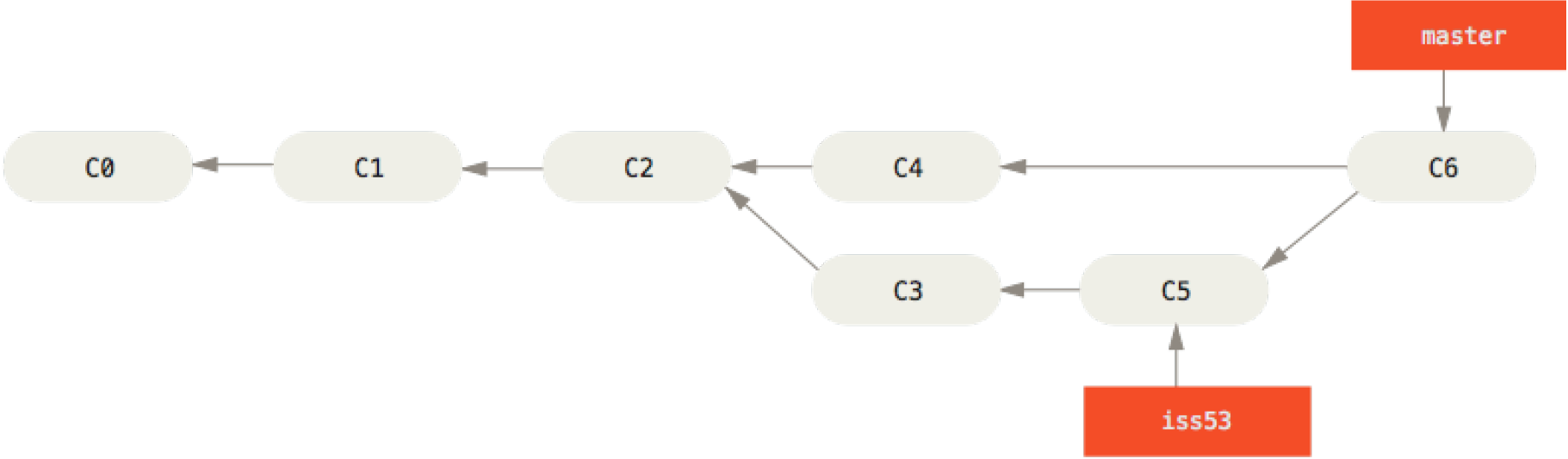
\* 05ee3e4 add readme.txt

\* c0be1c4 add README

开发历史从一个更早的地方开始分叉开来(diverged)。因为，master分支所在提交并不是iss53分支所在提交的直接祖先，Git不得不做一些额外的工作。 出现这种情况的时候，Git会使用两个分支的末端所指的快照(C4和C5)以及这两个分支的工作祖先(C2)，做一个简单的三方合并：



和之前将分支指针向前推进所不同的是，Git将此次三方合并的结果做了一个新的快照并且自动创建一个新的提交指向它。这个被称作一次合并提交，它的特别之处在于他有不止一个父提交：



Git会自行决定选取哪一个提交作为最优的共同祖先，并以此作为合并的基础；这和更加古老的CVS系统或者Subversion(1.5版本之前)不同，在这些古老的版本管理系统中，用户需要自己选择最佳的合并基础。Git的这个优势使其在合并操作上比其他系统要简单很多。

既然你的修改已经合并进来了，你已经不再需要iss53分支了。现在你可以在任务追踪系统中关闭此项任务，并删除这个分支：

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git branch -d iss53

Deleted branch iss53 (was 706e482).

Chenzf@Chenzf-desktop MINGW64 /d/git/gitmerge (master)

$ git branch

\* master

## 分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下*--no-ff*方式的git merge：

首先，仍然创建并切换dev分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch -c dev

Switched to a new branch 'dev'

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

修改readme.txt文件，并提交一个新的commit：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git commit -m "add 分支管理策略"

[dev 9945119] add 分支管理策略

1 file changed, 2 insertions(+), 1 deletion(-)

切换回master：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git switch master

Switched to branch 'master'

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

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

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

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

Merge made by the 'recursive' strategy.

readme.txt | 3 ++-

1 file changed, 2 insertions(+), 1 deletion(-)

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

合并后，我们用git log看看分支历史：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git log

commit c3bce63d56b6495c0b0ec3f02cb0a0e756d1c8f2 (HEAD -> master)

Merge: 48e80dc 9945119

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Fri Feb 7 16:48:54 2020 +0800

merge with no-ff

commit 99451191dd402eff2866e62b0eafe5fdfae7d792 (dev)

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Fri Feb 7 16:46:58 2020 +0800

add 分支管理策略

......

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

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

\* c3bce63 (HEAD -> master) merge with no-ff

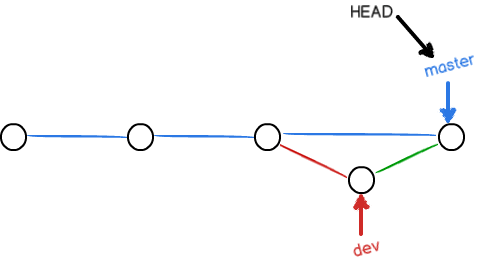
|\

| \* 9945119 (dev) add 分支管理策略

|/

\* 48e80dc conflict fixed

不使用Fast forward模式，merge后就像这样：



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。

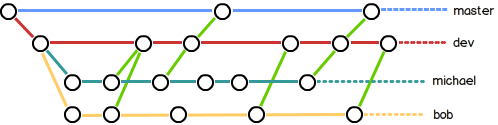
### 分支策略

在实际开发中，我们应该按照几个基本原则进行分支管理：

首先，master分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活；那在哪干活呢？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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git status

On branch dev

Changes not staged for commit:

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

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

modified: readme.txt

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git switch master

error: Your local changes to the following files would be overwritten by checkout:

readme.txt

Please commit your changes or stash them before you switch branches.

Aborting

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？

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git stash

Saved working directory and index state WIP on dev: 9945119 add 分支管理策略

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git status

On branch dev

nothing to commit, working tree clean

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。

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git switch master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 8 commits.

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch -c 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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (issue-101)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (issue-101)

$ git commit -m "fix bug 101: Git is a..."

[issue-101 e97502a] fix bug 101: Git is a...

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (issue-101)

$ git switch master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 8 commits.

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

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

Merge made by the 'recursive' strategy.

readme.txt | 2 +-

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git branch -d issue-101

Deleted branch issue-101 (was e97502a).

接着回到dev分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch dev

Switched to branch 'dev'

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git status

On branch dev

nothing to commit, working tree clean

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git stash list

stash@{0}: WIP on dev: 9945119 add 分支管理策略

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：

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

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git stash pop

On branch dev

Changes not staged for commit:

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

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

modified: readme.txt

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

Dropped refs/stash@{0} (fe1414e499dbb18f7f0be6a526e7a63694de3484)

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git stash list

再用git stash list查看，就看不到任何stash内容了。

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}*

在master分支上修复了bug后，我们要想一想，dev分支是早期从master分支分出来的，所以，这个bug其实在当前dev分支上也存在。

那怎么在dev分支上修复同样的bug？重复操作一次，提交不就行了？有木有更简单的方法？

同样的bug，要在dev上修复，我们只需要把e97502a fix bug 101这个提交所做的修改“复制”到dev分支。注意：我们只想复制e97502a fix bug 101这个提交所做的修改，并不是把整个master分支merge过来。

为了方便操作，Git专门提供了一个cherry-pick命令，让我们能复制一个特定的提交到当前分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git cherry-pick e97502a

error: Your local changes to the following files would be overwritten by merge:

readme.txt

Please commit your changes or stash them before you merge.

Aborting

fatal: cherry-pick failed

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git commit -m "Bug分支学习"

[dev 2bcd4d1] Bug分支学习

1 file changed, 2 insertions(+), 1 deletion(-)

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ 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.

Creating a new branch: feature1 is simple and quick.

分支管理策略

Bug分支

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git cherry-pick e97502a

Auto-merging readme.txt

[dev 78e969a] fix bug 101: Git is a...

Date: Fri Feb 7 17:30:00 2020 +0800

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ cat readme.txt

Git is a distributed version control system.

Git is a free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

Creating a new branch: feature1 is simple and quick.

分支管理策略

Bug分支

Git自动给dev分支做了一次提交，注意这次提交的commit是[dev 78e969a] fix bug 101: Git is a...，它并不同于master的[issue-101 e97502a] fix bug 101: Git is a...，因为这两个commit只是改动相同，但确实是两个不同的commit。用git cherry-pick，我们就不需要在dev分支上手动再把修bug的过程重复一遍。

既然可以在master分支上修复bug后，在dev分支上可以“重放”这个修复过程，那么直接在dev分支上修复bug，然后在master分支上“重放”行不行？当然可以，不过你仍然需要git stash命令保存现场，才能从dev分支切换到master分支。

### 小结

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；

在master分支上修复的bug，想要合并到当前dev分支，可以用git cherry-pick <commit>命令，把bug提交的修改“复制”到当前分支，避免重复劳动。

## 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。

准备开发：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git switch -c feature-vulcan

Switched to a new branch 'feature-vulcan'

开发完毕后：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature-vulcan)

$ git add vulcan.py

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature-vulcan)

$ git status

On branch feature-vulcan

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: vulcan.py

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature-vulcan)

$ git commit -m "add feature vulcan"

[feature-vulcan 735d77a] add feature vulcan

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 vulcan.py

切回dev，准备合并：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (feature-vulcan)

$ git switch dev

Switched to branch 'dev'

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。但是！此时新功能要被取消！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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ 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。现在强行删除：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git branch -D feature-vulcan

Deleted branch feature-vulcan (was 735d77a).

### 小结

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。

## 多人协作

当你从远程仓库克隆时，实际上Git自动把本地的master分支和远程的master分支对应起来了，并且，远程仓库的默认名称是origin。要查看远程库的信息，用git remote：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git remote

origin

或者，用*git remote -v*显示更详细的信息：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git remote -v

origin git@github.com:Chenzf2018/learningGit.git (fetch)

origin git@github.com:Chenzf2018/learningGit.git (push)

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

### 推送分支

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git push origin master

Enumerating objects: 24, done.

Counting objects: 100% (24/24), done.

Delta compression using up to 8 threads

Compressing objects: 100% (22/22), done.

Writing objects: 100% (22/22), 1.89 KiB | 643.00 KiB/s, done.

Total 22 (delta 16), reused 0 (delta 0)

remote: Resolving deltas: 100% (16/16), completed with 2 local objects.

remote: This repository moved. Please use the new location:

remote: git@github.com:Chenzf2018/LearningGit.git

To github.com:Chenzf2018/learningGit.git

5f6b7b1..7466e4d master -> master

如果要推送其他分支，比如dev，就改成：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git push origin dev

Enumerating objects: 8, done.

Counting objects: 100% (8/8), done.

Delta compression using up to 8 threads

Compressing objects: 100% (6/6), done.

Writing objects: 100% (6/6), 590 bytes | 590.00 KiB/s, done.

Total 6 (delta 4), reused 0 (delta 0)

remote: Resolving deltas: 100% (4/4), completed with 2 local objects.

remote: This repository moved. Please use the new location:

remote: git@github.com:Chenzf2018/LearningGit.git

remote:

remote: Create a pull request for 'dev' on GitHub by visiting:

remote: https://github.com/Chenzf2018/LearningGit/pull/new/dev

remote:

To github.com:Chenzf2018/learningGit.git

\* [new branch] dev -> dev

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，分支是否推送，视情况而定！

### 拉取pull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：

Chenzf@Chenzf-desktop MINGW64 /d

$ git clone git@github.com:Chenzf2018/LearningGit

Cloning into 'LearningGit'...

remote: Enumerating objects: 60, done.

remote: Counting objects: 100% (60/60), done.

remote: Compressing objects: 100% (25/25), done.

remote: Total 60 (delta 33), reused 58 (delta 31), pack-reused 0

Receiving objects: 100% (60/60), 5.03 KiB | 177.00 KiB/s, done.

Resolving deltas: 100% (33/33), 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。不信可以用git branch命令看看：

Chenzf@Chenzf-desktop MINGW64 /d

$ cd /d/LearningGit

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (master)

$ 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：

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (master)

$ git checkout -b dev origin/dev

Switched to a new branch 'dev'

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

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：

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ git add env.txt

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ git commit -m "add env.txt"

[dev 992eab8] add env.txt

1 file changed, 8 insertions(+)

create mode 100644 env.txt

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ git push origin dev

Enumerating objects: 3, done.

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

Delta compression using up to 8 threads

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

Writing objects: 100% (2/2), 254 bytes | 254.00 KiB/s, done.

Total 2 (delta 1), reused 0 (delta 0)

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

To github.com:Chenzf2018/LearningGit

78e969a..992eab8 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：

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ cd /d/Git/LearningGit/

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (master)

$ git switch dev

Switched to branch 'dev'

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ cat env.txt

env

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git add env.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git commit -m "add new env.txt"

[dev b63559b] add new env.txt

1 file changed, 1 insertion(+)

create mode 100644 env.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git push origin dev

To github.com:Chenzf2018/learningGit.git

! [rejected] dev -> dev (fetch first)

error: failed to push some refs to 'git@github.com:Chenzf2018/learningGit.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.

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git pull

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

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

remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0

Unpacking objects: 100% (2/2), 234 bytes | 33.00 KiB/s, done.

From github.com:Chenzf2018/learningGit

78e969a..992eab8 dev -> origin/dev

There is no tracking information for the current branch.

Please specify which branch you want to merge with.

See git-pull(1) for details.

git pull <remote> <branch>

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

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

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

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

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

再pull：

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

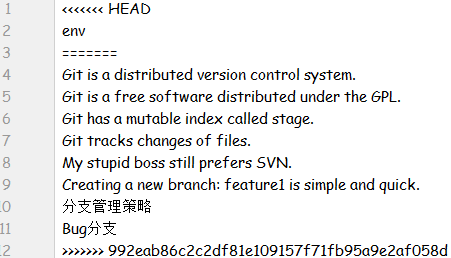
$ git pull

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

Auto-merging 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：



修改：



Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev|MERGING)

$ git commit -m "fix env conflict"

error: Committing is not possible because you have unmerged files.

hint: Fix them up in the work tree, and then use 'git add/rm <file>'

hint: as appropriate to mark resolution and make a commit.

fatal: Exiting because of an unresolved conflict.

U env.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev|MERGING)

$ git add env.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev|MERGING)

$ git commit -m "fix enc conflict"

[dev 4982115] fix enc conflict

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ git push origin dev

Enumerating objects: 7, done.

Counting objects: 100% (7/7), done.

Delta compression using up to 8 threads

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

Writing objects: 100% (4/4), 391 bytes | 391.00 KiB/s, done.

Total 4 (delta 2), reused 0 (delta 0)

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

remote: This repository moved. Please use the new location:

remote: git@github.com:Chenzf2018/LearningGit.git

To github.com:Chenzf2018/learningGit.git

992eab8..4982115 dev -> dev

#### 再探pull功能

Chenzf@Chenzf-desktop MINGW64 /d/Git/LearningGit (dev)

$ cd /d/LearningGit

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ cat env.txt

Git is a distributed version control system.

Git is a free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

Creating a new branch: feature1 is simple and quick.

分支管理策略

Bug分支

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

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

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

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ cat env.txt

Git is a distributed version control system.

Git is a free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

Creating a new branch: feature1 is simple and quick.

分支管理策略

Bug分支

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ git pull

remote: Enumerating objects: 7, done.

remote: Counting objects: 100% (7/7), done.

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

remote: Total 4 (delta 2), reused 4 (delta 2), pack-reused 0

Unpacking objects: 100% (4/4), 371 bytes | 30.00 KiB/s, done.

From github.com:Chenzf2018/LearningGit

992eab8..4982115 dev -> origin/dev

Updating 992eab8..4982115

Fast-forward

env.txt | 9 +--------

1 file changed, 1 insertion(+), 8 deletions(-)

Chenzf@Chenzf-desktop MINGW64 /d/LearningGit (dev)

$ cat env.txt

env

因此，多人协作的工作模式通常是这样：

首先，可以试图用*git push origin <branch-name>*推送自己的修改；

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；如果合并有冲突，则解决冲突，并在本地提交；没有冲突或者解决掉冲突后，再用*git push origin <branch-name>*推送就能成功！

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

这就是多人协作的工作模式，一旦熟悉了，就非常简单。

### 小结

查看远程库信息，使用git remote -v；

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；

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；

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。

当git fetch命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。它只会获取数据然后让你自己合并。然而，有一个命令叫作git pull在大多数情况下它的含义是一个git fetch紧接着一个git merge命令。

## 分支开发工作流

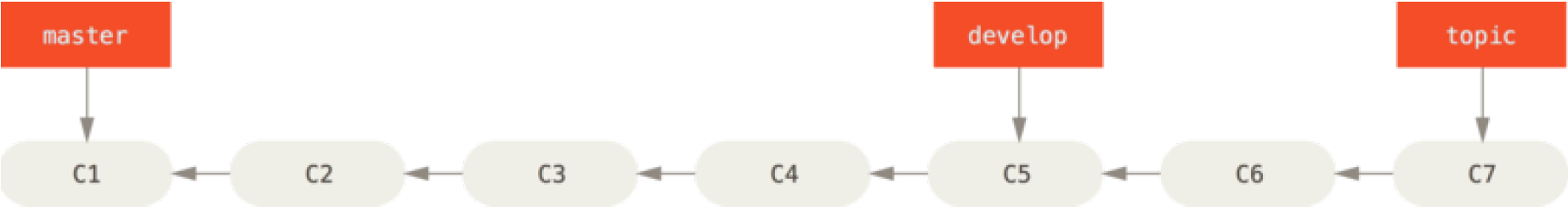
在本节，会介绍一些常见的利用分支进行开发的工作流程。而正是由于分支管理的便捷，才衍生出这些典型的工作模式。

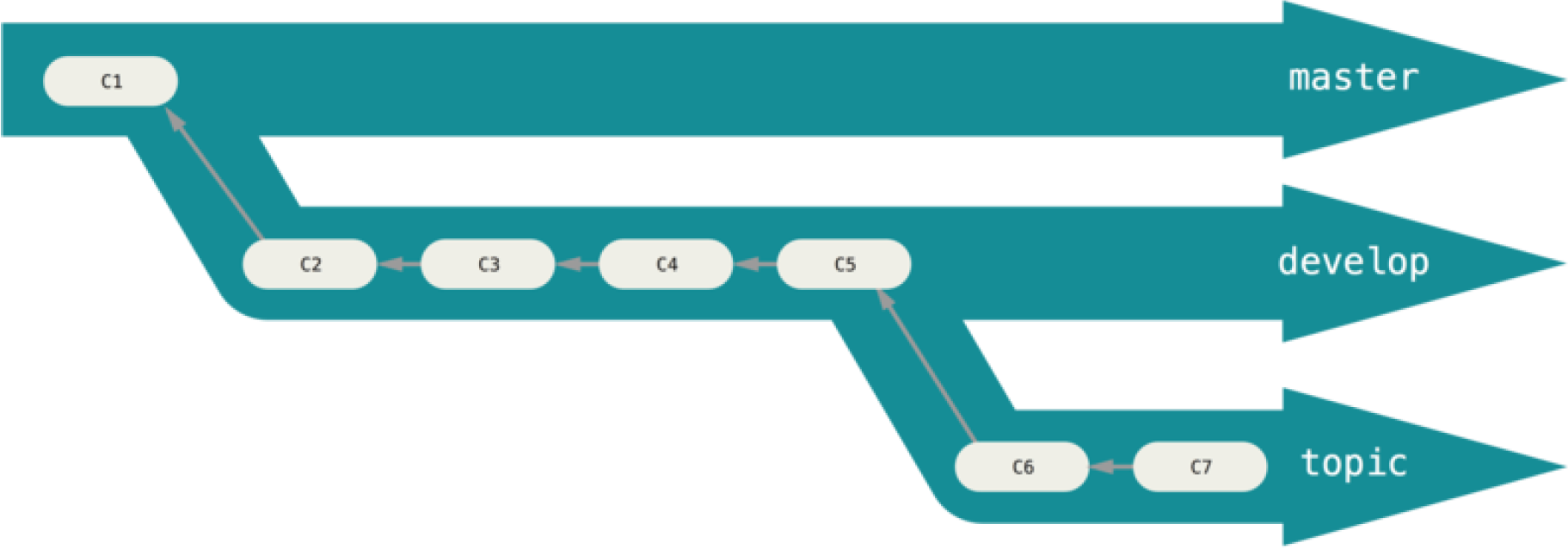
### 长期分支

因为Git使用简单的三方合并，在整个项目开发周期的不同阶段，你可以同时拥有多个开放的分支；你可以定期地把某些特性分支合并入其他分支中。

许多使用Git的开发者都喜欢使用这种方式来工作，比如只在master分支上保留完全稳定的代码——有可能仅仅是已经发布或即将发布的代码。他们还有一些平行分支，被用来做后续开发或者测试稳定性——这些分支不必保持绝对稳定，但是一旦达到稳定状态，它们就可以被合并入master分支了。这样，在确保这些已完成的特性分支(短期分支，比如之前的iss53分支)能够通过所有测试，并且不会引入更多bug之后，就可以合并入主干分支中，等待下一次的发布。

渐进稳定分支：





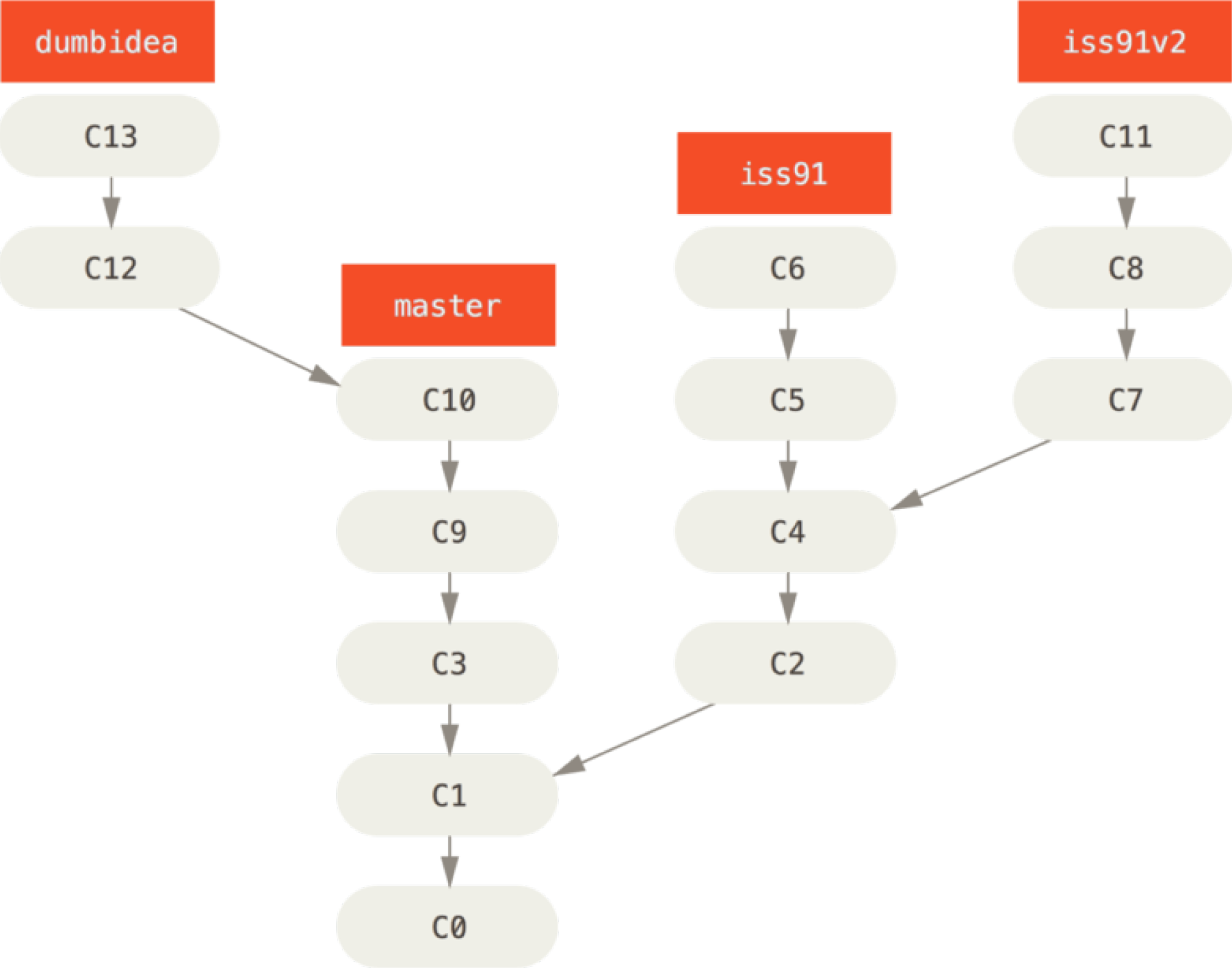
使你的分支具有不同级别的稳定性；当它们具有一定程度的稳定性后，再把它们合并入具有更高级别稳定性的分支中。

### 特性分支

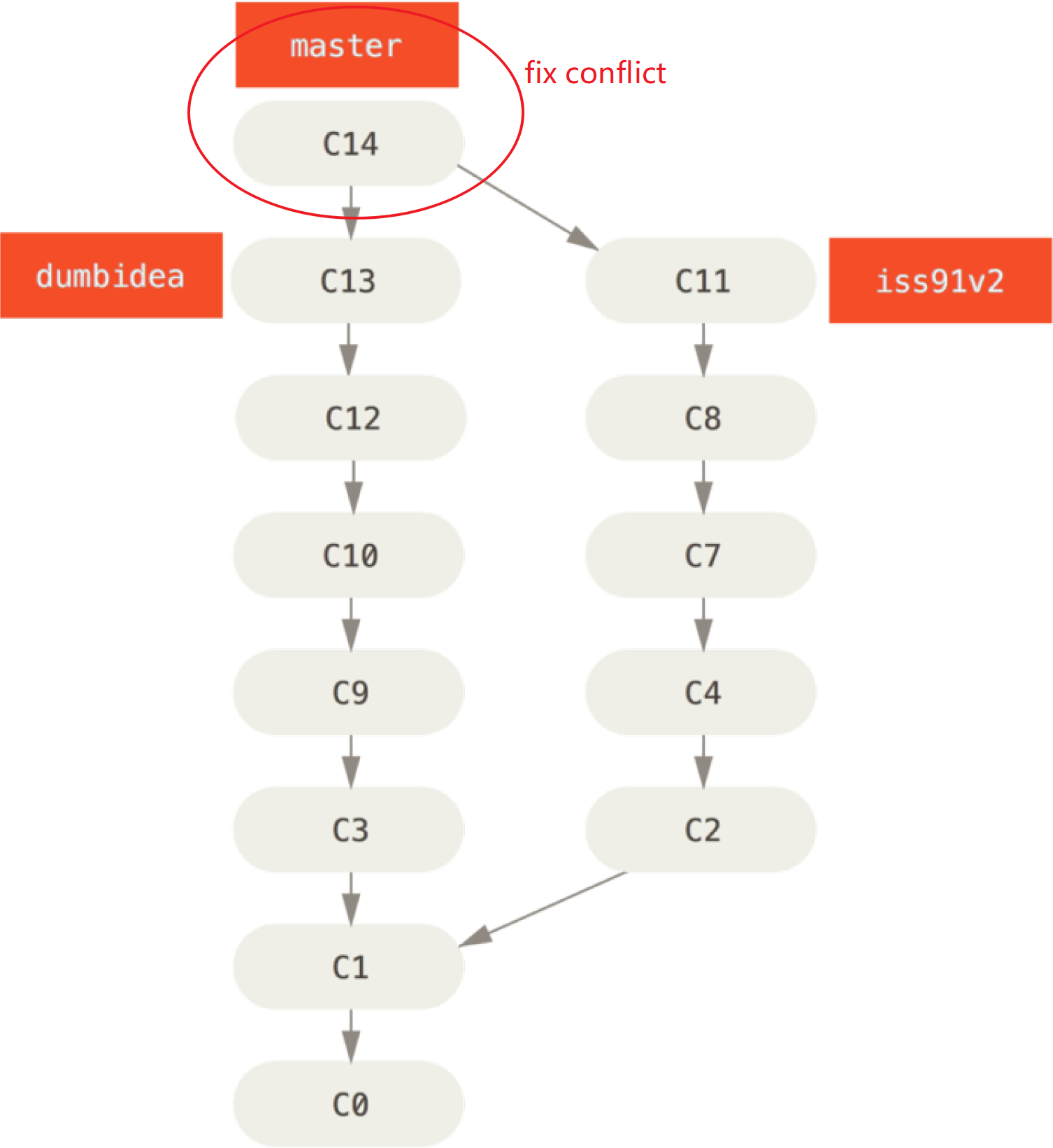
特性分支是一种短期分支，它被用来实现单一特性或其相关工作。这项技术能使你快速并且完整地进行上下文切换(context-switch)——因为你的工作被分散到不同的流水线中，在不同的流水线中每个分支都仅与其目标特性相关，因此，在做代码审查之类的工作的时候就能更加容易地看出你做了哪些改动。 你可以把做出的改动在特性分支中保留几分钟、几天甚至几个月，等它们成熟之后再合并，而不用在乎它们建立的顺序或工作进度。

考虑这样一个例子，你在master分支上工作到C1，这时为了解决一个问题而新建iss91分支，在iss91分支上工作到C4，然而对于那个问题你又有了新的想法，于是你再新建一个iss91v2分支试图用另一种方法解决那个问题，接着你回到master分支工作了一会儿，你又冒出了一个不太确定的想法，你便在C10 的时候新建一个dumbidea分支，并在上面做些实验。

你的提交历史看起来像下面这个样子：



现在，你决定使用在iss91v2分支中方案；另外，你将dumbidea分支拿给你的同事看过之后，结果发现这是个惊人之举。这时你可以抛弃iss91分支(即丢弃C5和C6提交)，然后把另外两个分支合并入主干分支。最终你的提交历史看起来像下面这个样子：



### 远程分支

远程引用是对远程仓库的引用(指针)，包括分支、标签等等。你可以通过*git ls-remote [remote]*来显式地获得远程引用的完整列表，或者通过*git remote show [remote]*获得远程分支的更多信息。 然而，一个更常见的做法是利用远程跟踪分支。

远程跟踪分支是远程分支状态的引用。它们是你不能移动的本地引用，当你做任何网络通信操作时，它们会自动移动。远程跟踪分支像是你上次连接到远程仓库时，那些分支所处状态的书签。它们以*(remote)/(branch)*形式命名。 例如，如果你想要看你最后一次与远程仓库origin通信时master分支的状态，你可以查看origin/master分支。 你与同事合作解决一个问题并且他们推送了一个iss53分支，你可能有自己的本地iss53分支；但是在服务器上的分支会指向origin/iss53的提交。

假设你的网络里有一个在git@github.com/Chenzf2018/GitClone的Git服务器。如果你从这里克隆，Git的clone命令会为你自动将其命名为origin，拉取它的所有数据，创建一个指向它的master分支的指针，并且在本地将其命名为origin/master。Git也会给你一个与origin的master分支在指向同一个地方的本地master分支，这样你就有工作的基础。

Chenzf@Chenzf-desktop MINGW64 /d/Git

$ git clone git@github.com:Chenzf2018/GitClone.git

Cloning into 'GitClone'...

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

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

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

Chenzf@Chenzf-desktop MINGW64 /d/Git

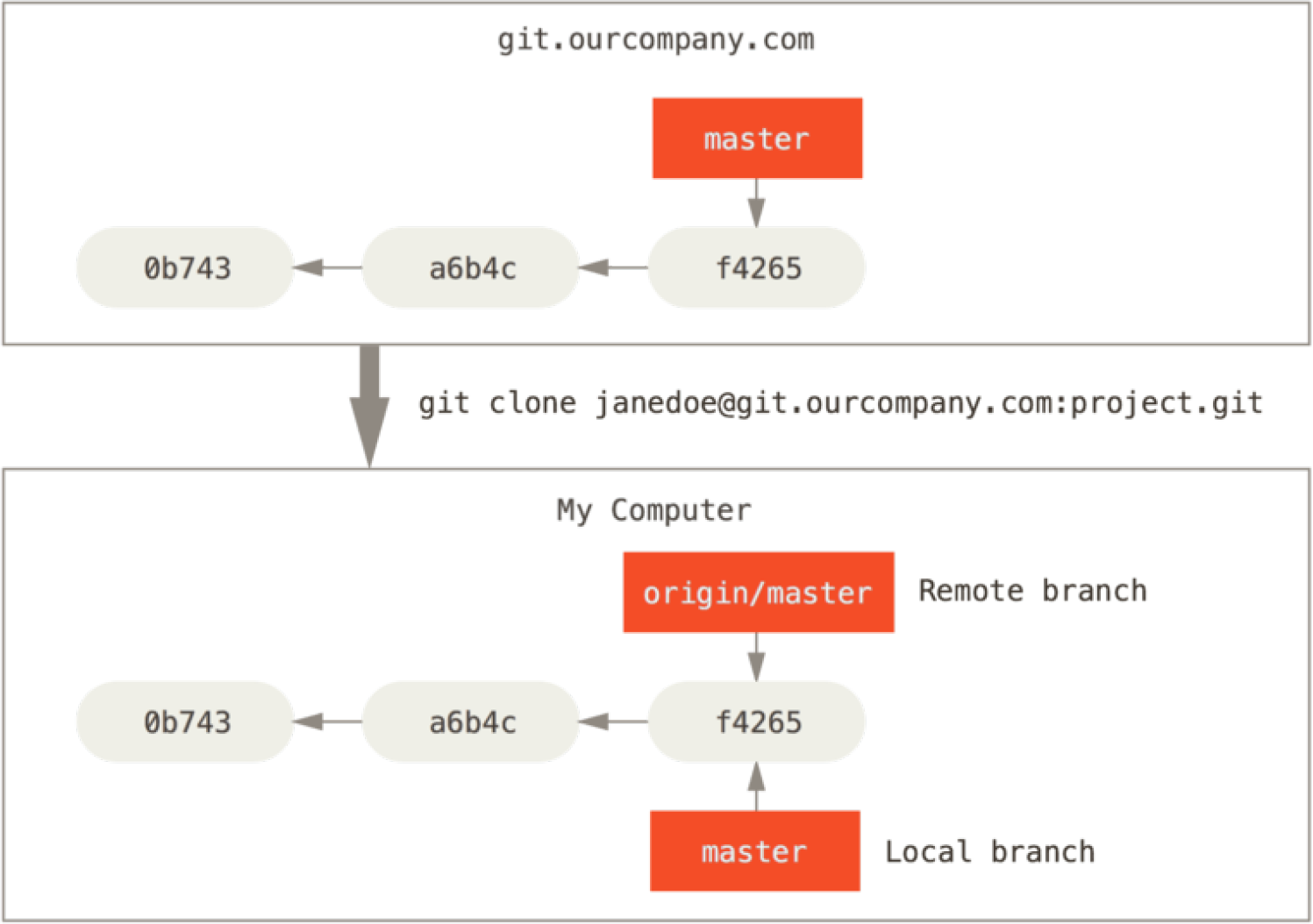
$ cd GitClone/

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git log --graph --oneline --all --decorate

\* 60f5e2c (HEAD -> master, origin/master, origin/HEAD) Initial commit

远程仓库名字“origin”与分支名字“master”一样，在Git中并没有任何特别的含义一样。同时“master”是当你运行git init时默认的起始分支名字，原因仅仅是它的广泛使用，“origin”是当你运行 git clone时默认的远程仓库名字。如果你运行 *git clone -o booyah*，那么你默认的远程分支名字将会是 booyah/master。



Chenzf@Chenzf-desktop MINGW64 /d

$ git clone git@github.com:Chenzf2018/GitClone.git

Cloning into 'GitClone'...

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

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

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

Chenzf@Chenzf-desktop MINGW64 /d

$ cd GitClone

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git log

commit 60f5e2cadb7cc71c1136edd603cdf6deaef27772 (HEAD -> master, origin/master, origin/HEAD)

Author: Chenzf2018 <40626787+Chenzf2018@users.noreply.github.com>

Date: Sat Feb 8 16:44:23 2020 +0800

Initial commit

如果你在本地的master分支做了一些工作，在同一段时间内有其他人推送提交到git@github.com:Chenzf2018/GitClone.git并且更新了它的master分支，这就是说你们的提交历史已走向不同的方向。即便这样，只要你保持不与origin服务器连接(并拉取数据)，你的origin/master指针就不会移动。

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git commit -m "add readme.txt in /d/GitClone"

[master 517eb1a] add readme.txt in /d/GitClone

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git push origin master

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 8 threads

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

Writing objects: 100% (3/3), 291 bytes | 145.00 KiB/s, done.

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

To github.com:Chenzf2018/GitClone.git

60f5e2c..517eb1a master -> master

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git log --graph --all --oneline --decorate

\* 517eb1a (HEAD -> master, origin/master, origin/HEAD) add readme.txt in /d/GitClone

\* 60f5e2c Initial commit

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ cd /d/Git/GitClone

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git commit -m "add readme.txt in /d/Git/GitClone"

[master cfc5dc4] add readme.txt in /d/Git/GitClone

1 file changed, 0 insertions(+), 0 deletions(-)

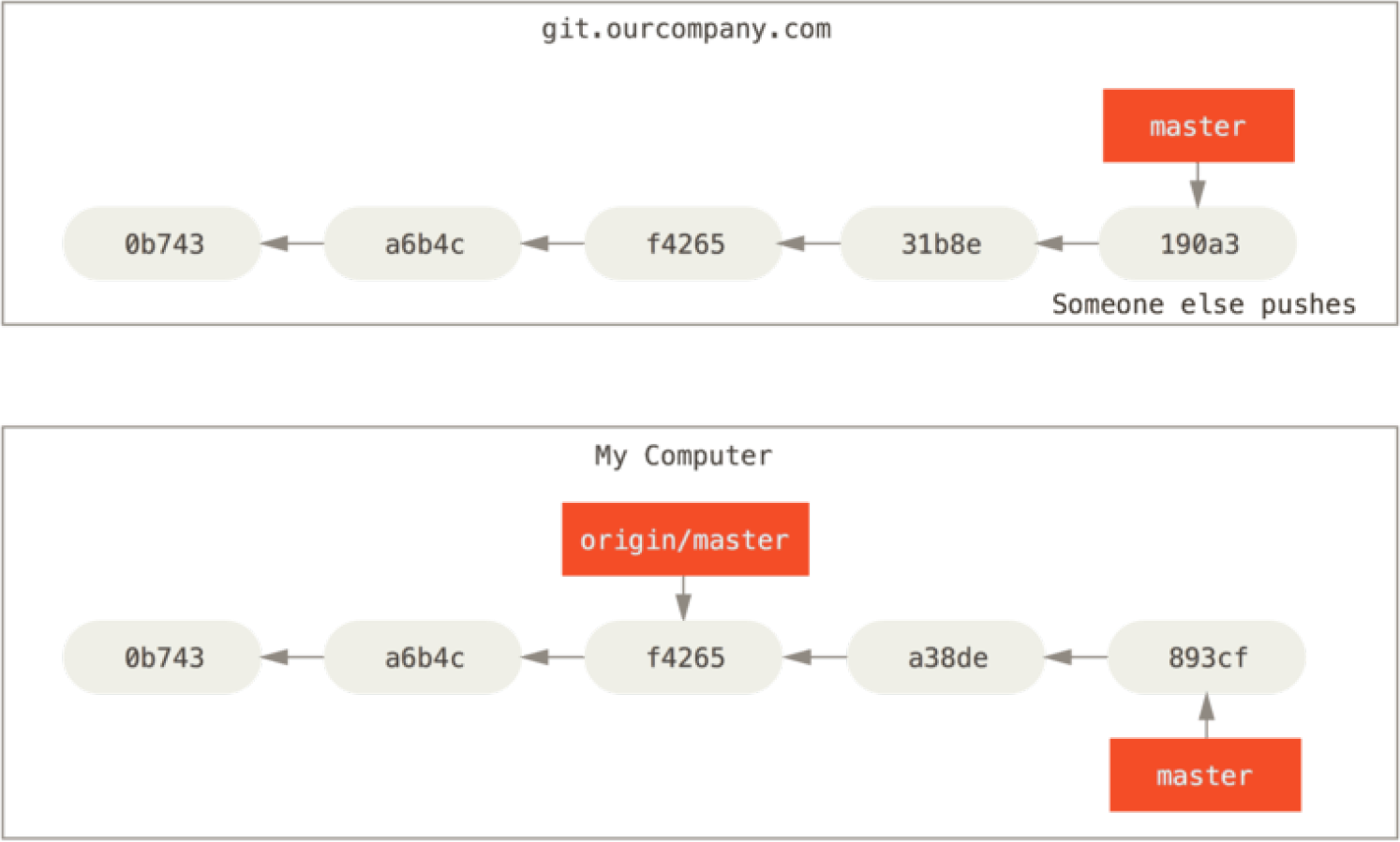
create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git log --graph --all --oneline --decorate

\* cfc5dc4 (HEAD -> master) add readme.txt in /d/Git/GitClone

\* 60f5e2c (origin/master, origin/HEAD) Initial commit



#### 从远程库fetch

如果要同步数据，运行*git fetch origin*命令。这个命令查找“origin”是哪一个服务器(在本例中，是git@github.com:Chenzf2018/GitClone.git)，从中抓取本地没有的数据，并且更新本地数据库，移动origin/master指针到更新之后的位置：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git fetch origin

remote: Enumerating objects: 4, done.

remote: Counting objects: 100% (4/4), done.

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

Unpacking objects: 100% (3/3), 271 bytes | 90.00 KiB/s, done.

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

From github.com:Chenzf2018/GitClone

60f5e2c..517eb1a master -> origin/master

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

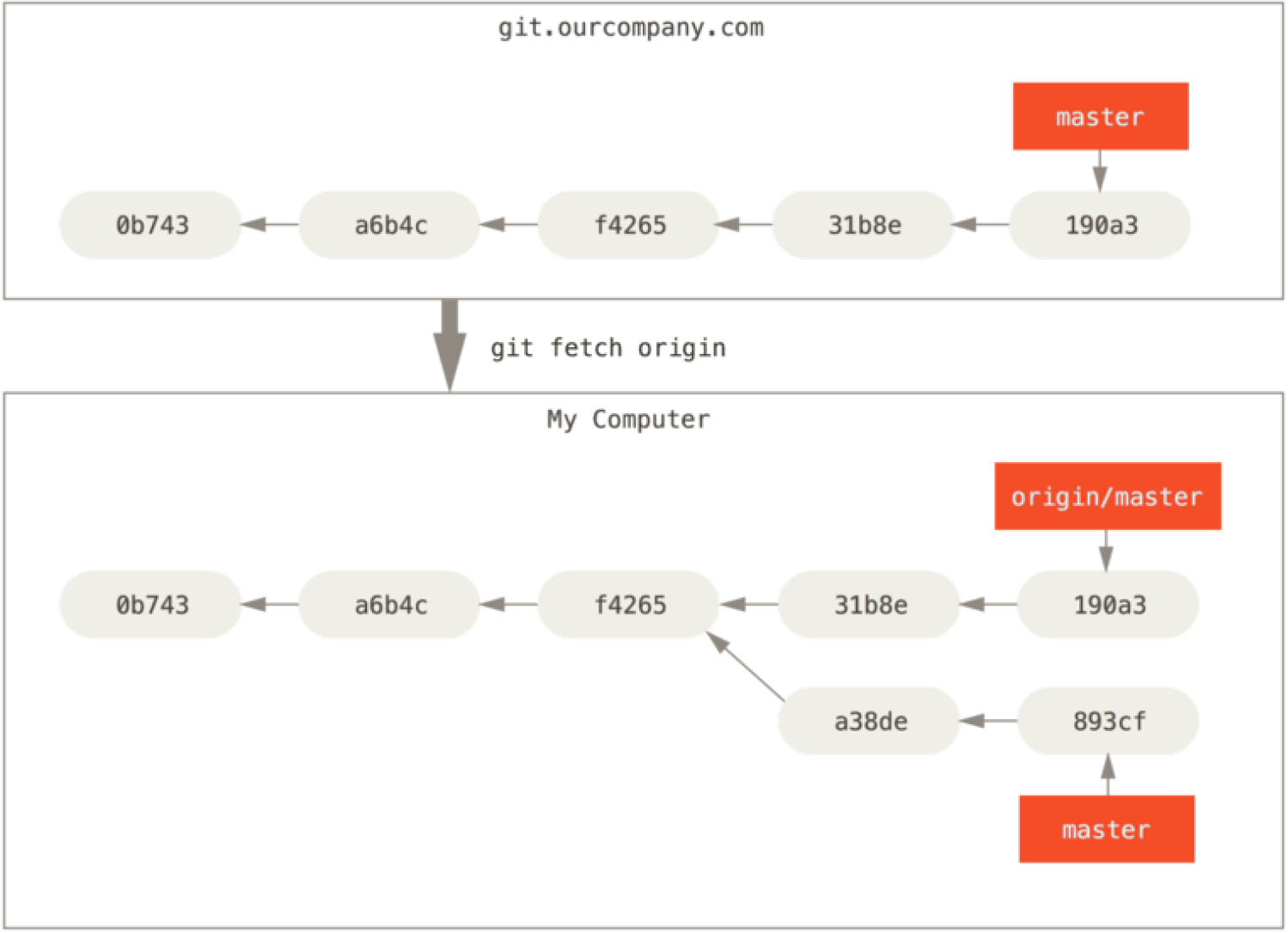
$ git log --graph --all --decorate --oneline

\* cfc5dc4 (HEAD -> master) add readme.txt in /d/Git/GitClone

| \* 517eb1a (origin/master, origin/HEAD) add readme.txt in /d/GitClone

|/

\* 60f5e2c Initial commit



当git fetch命令从服务器上抓取本地没有的数据时，它并不会修改工作目录中的内容。它只会获取数据然后让你自己合并。然而，有一个命令叫作git pull在大多数情况下它的含义是一个git fetch紧接着一个git merge命令。

### 删除远程分支

可以运行带有 --delete选项的git push命令来删除一个远程分支：

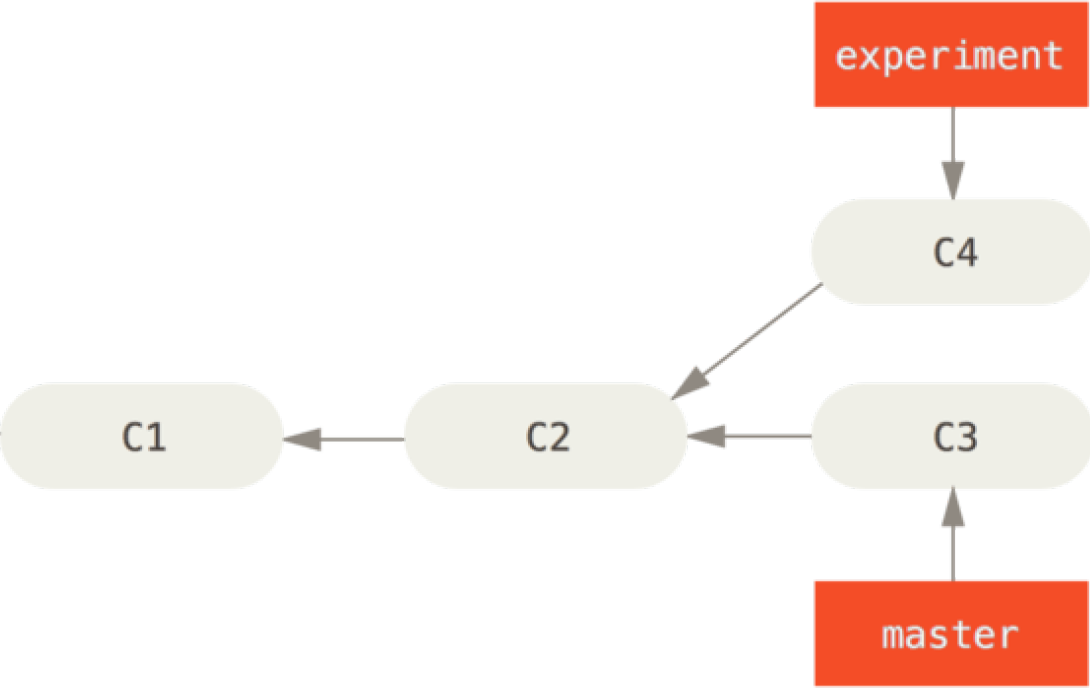
*$ git push origin --delete serverfix*

## Rebase变基

在Git中整合来自不同分支的修改主要有两种方法：merge以及rebase。

### 变基的基本操作

回顾三方合并：开发任务分叉到两个不同分支，又各自提交了更新



整合分支最容易的方法是merge命令。它会把两个分支的最新快照(C3和C4)以及二者最近的共同祖先(C2)进行三方合并，合并的结果是生成一个新的快照(并提交)。

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase

$ git init

Initialized empty Git repository in D:/Git/GitRebase/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add readme.txt"

[master (root-commit) 2e62652] add readme.txt

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add 'My Project' to readme.txt"

[master 411031d] add 'My Project' to readme.txt

1 file changed, 1 insertion(+)

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* 411031d (HEAD -> master) add 'My Project' to readme.txt

\* 2e62652 add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git checkout -b experiment

Switched to a new branch 'experiment'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ vim experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git add experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git commit -m "add experiment.txt in experiment branch"

[experiment 840f816] add experiment.txt in experiment branch

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add 'Learing rebasing' in readme.txt"

[master b26f976] add 'Learing rebasing' in readme.txt

1 file changed, 1 insertion(+)

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* b26f976 (HEAD -> master) add 'Learing rebasing' in readme.txt

| \* 840f816 (experiment) add experiment.txt in experiment branch

|/

\* 411031d add 'My Project' to readme.txt

\* 2e62652 add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git merge experiment

Merge made by the 'recursive' strategy.

experiment.txt | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* da48016 (HEAD -> master) Merge branch 'experiment'

|\

| \* 840f816 (experiment) add experiment.txt in experiment branch

\* | b26f976 add 'Learing rebasing' in readme.txt

|/

\* 411031d add 'My Project' to readme.txt

\* 2e62652 add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git branch -d experiment

Deleted branch experiment (was 840f816).

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* da48016 (HEAD -> master) Merge branch 'experiment'

|\

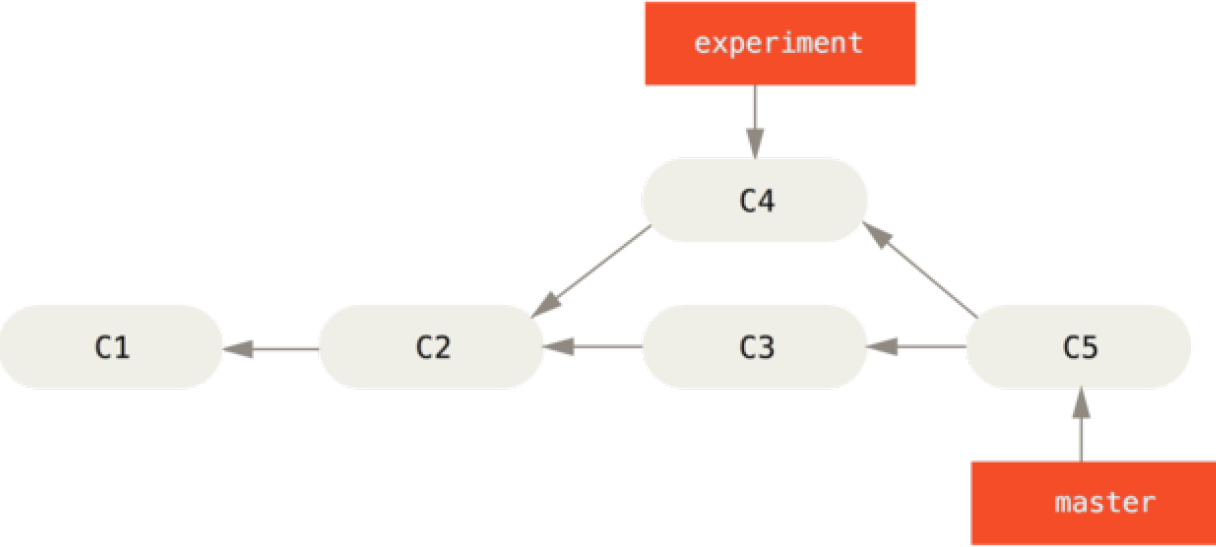
| \* 840f816 add experiment.txt in experiment branch

\* | b26f976 add 'Learing rebasing' in readme.txt

|/

\* 411031d add 'My Project' to readme.txt

\* 2e62652 add readme.txt



还有一种方法：你可以提取在C4中引入的补丁和修改，然后在C3的基础上应用一次。在Git中，这种操作就叫做变基。你可以使用rebase命令将提交到某一分支上的所有修改都移至另一分支上，就好像“重新播放”一样。

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase

$ git init

Initialized empty Git repository in D:/Git/GitRebase/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --oneline --decorate

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --oneline --decorate

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add readme.txt"

[master (root-commit) 2d0eb4e] add readme.txt

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --oneline --decorate

\* 2d0eb4e (HEAD -> master) add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add My project to readme.txt"

[master 7772aad] add My project to readme.txt

1 file changed, 1 insertion(+)

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --all --graph --oneline --decorate

\* 7772aad (HEAD -> master) add My project to readme.txt

\* 2d0eb4e add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add Learning Git rebasing to readme.txt"

[master 3ecadcc] add Learning Git rebasing to readme.txt

1 file changed, 1 insertion(+)

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --oneline --decorate --all

\* 3ecadcc (HEAD -> master) add Learning Git rebasing to readme.txt

\* 7772aad add My project to readme.txt

\* 2d0eb4e add readme.txt

#### 指定位置创建分支

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git checkout -b experiment 7772aad

Switched to a new branch 'experiment'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git log --graph --oneline --decorate --all

\* 3ecadcc (master) add Learning Git rebasing to readme.txt

\* 7772aad (HEAD -> experiment) add My project to readme.txt

\* 2d0eb4e add readme.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ vim experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git add experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git commit -m "add experiment.txt in experiment branch"

[experiment fcbec27] add experiment.txt in experiment branch

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git log --graph --oneline --decorate --all

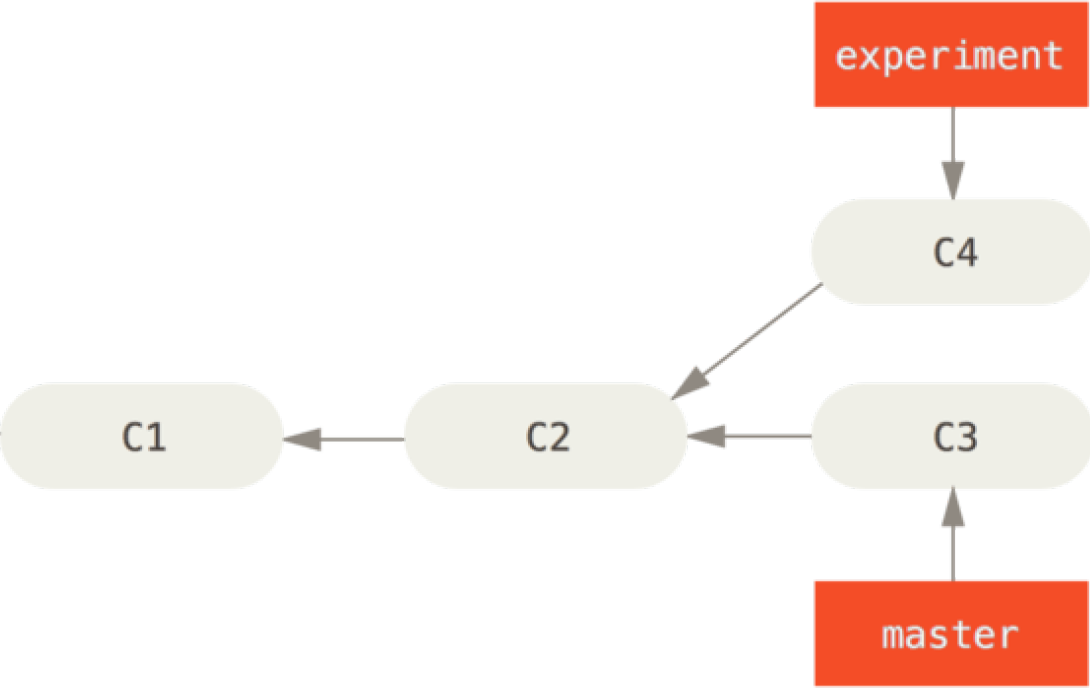
\* **fcbec27** (HEAD -> experiment) add experiment.txt in experiment branch

| \* 3ecadcc (master) add Learning Git rebasing to readme.txt

|/

\* 7772aad add My project to readme.txt

\* 2d0eb4e add readme.txt



Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git rebase master

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

Applying: add experiment.txt in experiment branch

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

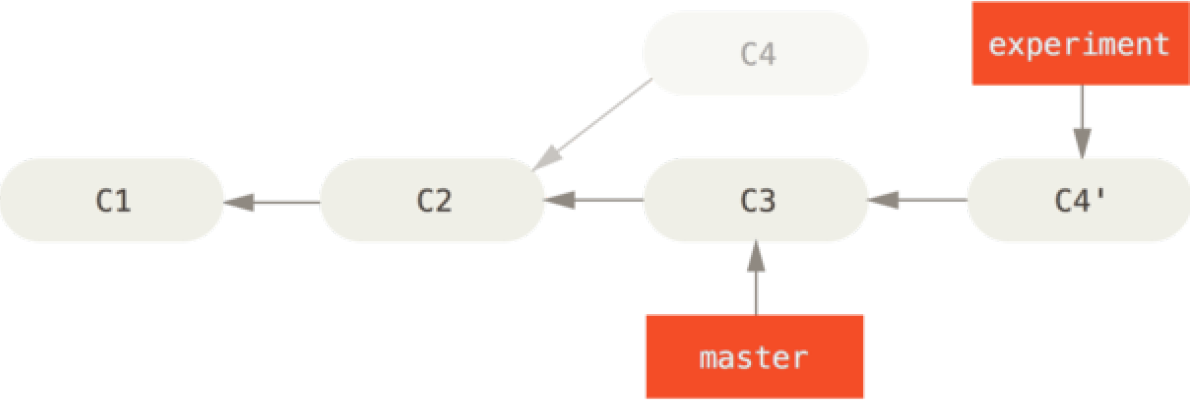
$ git log --graph --all --decorate --oneline

\* **fc72503** (HEAD -> experiment) add experiment.txt in experiment branch

\* 3ecadcc (master) add Learning Git rebasing to readme.txt

\* 7772aad add My project to readme.txt

\* 2d0eb4e add readme.txt



它的原理是首先找到这两个分支(即当前分支experiment、变基操作的目标基底分支master)的最近共同祖先C2，然后对比当前分支相对于该祖先的历次提交，提取相应的修改并存为临时文件，然后将当前分支指向目标基底C3，最后以此将之前另存为临时文件的修改依次序应用。

现在回到master分支，进行一次快进合并：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (experiment)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git merge experiment

Updating 3ecadcc..fc72503

Fast-forward

experiment.txt | 0

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 experiment.txt

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

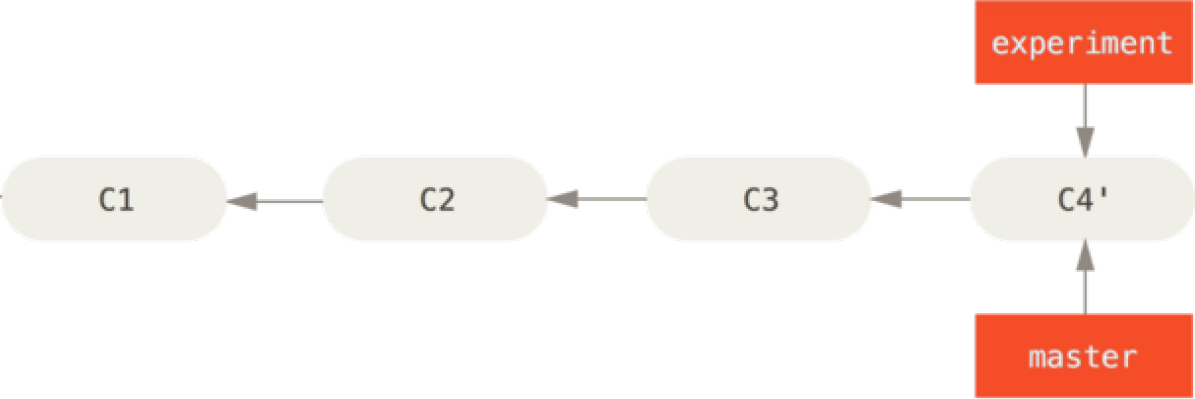
$ git log --graph --all --oneline --decorate

\* **fc72503** (HEAD -> master, experiment) add experiment.txt in experiment branch

\* 3ecadcc add Learning Git rebasing to readme.txt

\* 7772aad add My project to readme.txt

\* 2d0eb4e add readme.txt



这两种整合方法中变基使得提交历史更加整洁。你在查看一个经过变基的分支的历史记录时会发现，尽管实际的开发工作是并行的，但它们看上去就像是串行的一样，提交历史是一条直线没有分叉。

### 变基例子

在对两个分支进行变基时，所生成的“重放”并不一定要在目标分支上应用，你也可以指定另外的一个分支进行应用。

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase

$ git init

Initialized empty Git repository in D:/Git/GitRebase/.git/

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add C1"

[master (root-commit) 498984f] add C1

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim C2

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add C2

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add C2"

[master 3aaaf38] add C2

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C2

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git checkout -b server

Switched to a new branch 'server'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ vim C3

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git add C3

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git commit -m "add C3"

[server cbd2003] add C3

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C3

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ vim C4

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git add C4

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git commit -m "add C4"

[server 9c13c92] add C4

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C4

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git log --graph --oneline --all --decorate

\* 9c13c92 (HEAD -> server) add C4

\* cbd2003 add C3

\* 3aaaf38 (master) add C2

\* 498984f add C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git checkout master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim C5

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add C5

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add C5"

[master b512f11] add C5

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C5

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ vim C6

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git add C6

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git commit -m "add C6"

[master 35c0fd7] add C6

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C6

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --oneline --all --decorate

\* 35c0fd7 (HEAD -> master) add C6

\* b512f11 add C5

| \* 9c13c92 (server) add C4

| \* cbd2003 add C3

|/

\* 3aaaf38 add C2

\* 498984f add C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git checkout -b client cbd2003

Switched to a new branch 'client'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ vim C8

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git add C8

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git commit -m "add C8"

[client 17a8bf4] add C8

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C8

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ vim C9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git add C9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git commit -m "add C9"

[client 4d8be32] add C9

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git log --graph --oneline --decorate --all

\* 4d8be32 (HEAD -> client) add C9

\* 17a8bf4 add C8

| \* 35c0fd7 (master) add C6

| \* b512f11 add C5

| | \* 9c13c92 (server) add C4

| |/

|/|

\* | cbd2003 add C3

|/

\* 3aaaf38 add C2

\* 498984f add C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git checkout server

Switched to branch 'server'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ vim C10

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git add C10

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git commit -m "add C10"

[server 1b74d18] add C10

1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C10

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git log --graph --all --decorate --oneline

\* 1b74d18 (HEAD -> server) add C10

\* 9c13c92 add C4

| \* 4d8be32 (client) add C9

| \* 17a8bf4 add C8

|/

\* cbd2003 add C3

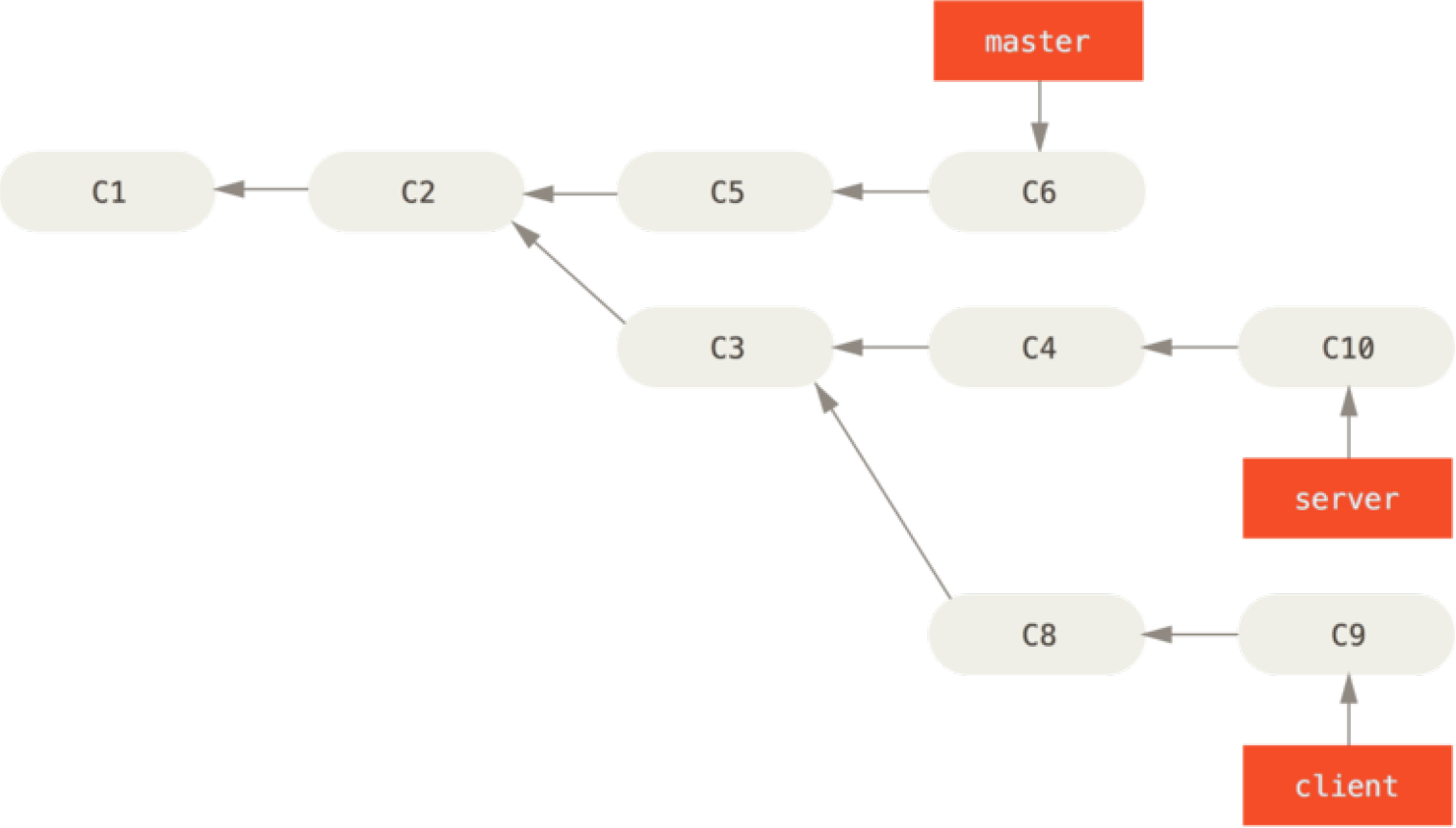
| \* 35c0fd7 (master) add C6

| \* b512f11 add C5

|/

\* 3aaaf38 add C2

\* 498984f add C1



假设你希望将client中的修改合并到主分支并发布，但暂时并不想合并server中的修改，因为它们还需要经过更全面的测试。这时，你就可以使用git rebase命令的--onto选项，**选中在client分支里但不在server分支里的修改(即C8和C9)**，将它们在master分支上重放：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git rebase --onto master server client

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

Applying: add C8

Applying: add C9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git log --graph --oneline --decorate --all

\* b7d8d8c (HEAD -> client) add C9

\* b6815ae add C8

\* 35c0fd7 (master) add C6

\* b512f11 add C5

| \* 1b74d18 (server) add C10

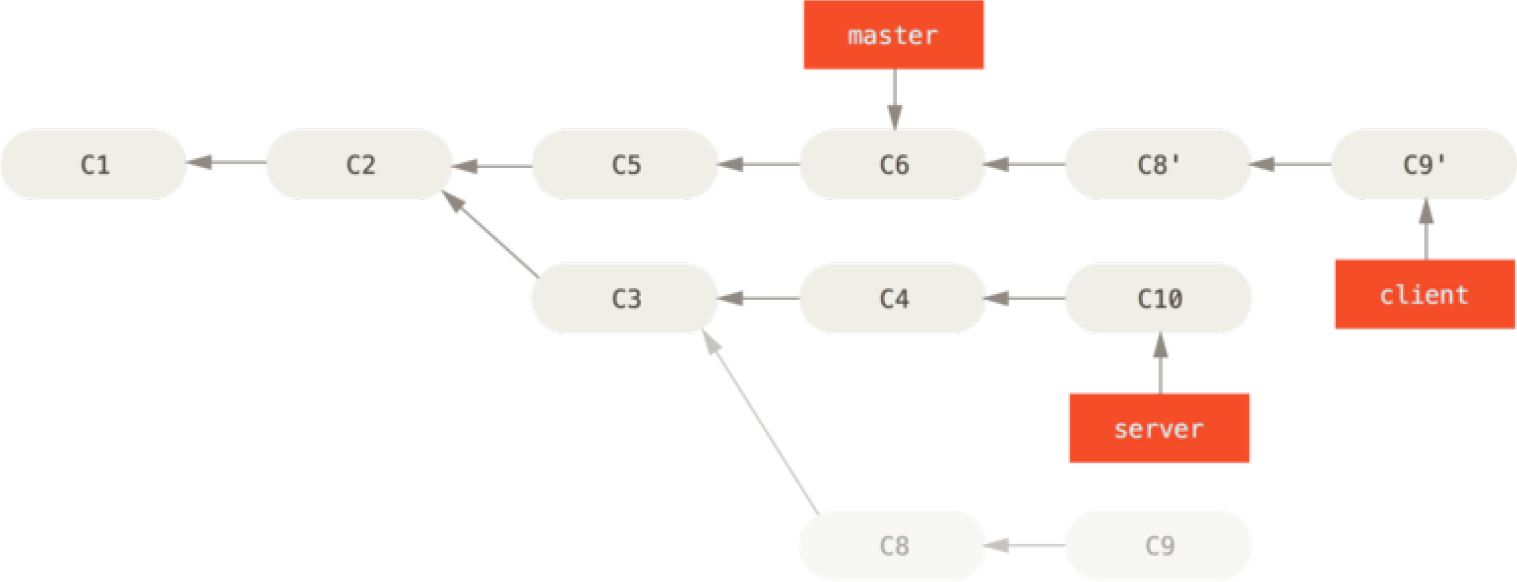
| \* 9c13c92 add C4

| \* cbd2003 add C3

|/

\* 3aaaf38 add C2

\* 498984f add C1



现在可以快进合并master分支了：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git merge client

Updating 35c0fd7..b7d8d8c

Fast-forward

C8 | 0

C9 | 0

2 files changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C8

create mode 100644 C9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --oneline --decorate --all

\* b7d8d8c (HEAD -> master, client) add C9

\* b6815ae add C8

\* 35c0fd7 add C6

\* b512f11 add C5

| \* 1b74d18 (server) add C10

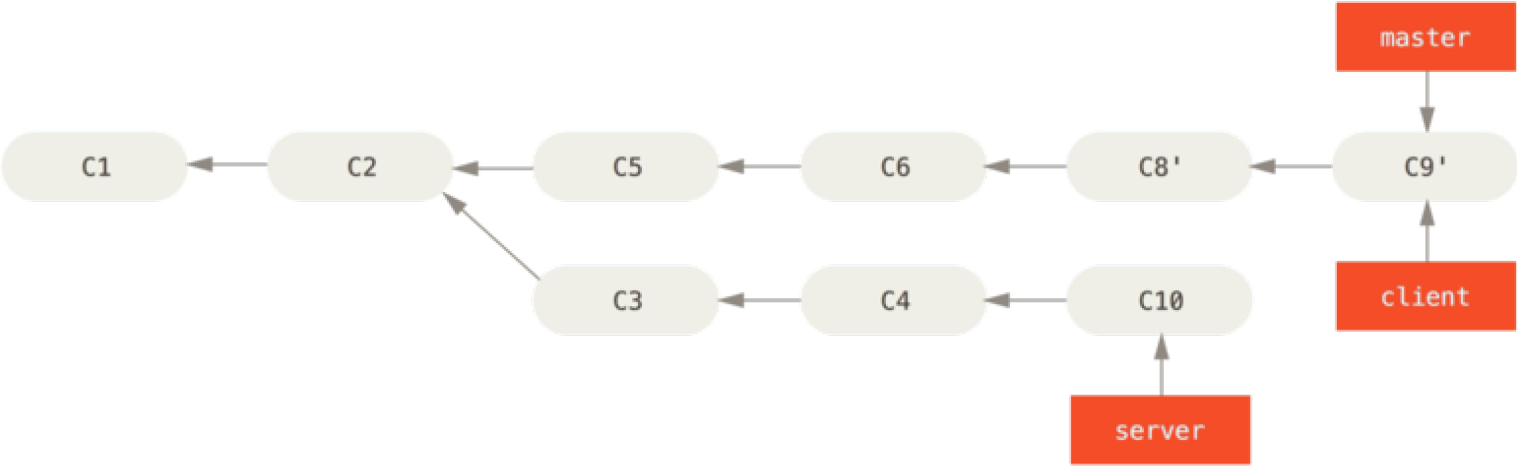
| \* 9c13c92 add C4

| \* cbd2003 add C3

|/

\* 3aaaf38 add C2

\* 498984f add C1



接下来你决定将server分支中的修改也整合进来。

使用*git rebase [basebranch] [topicbranch]*命令可以直接将特性分支(即本例中的server)变基到目标分支(即master)上。

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git rebase master server

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

Applying: add C3

Applying: add C4

Applying: add C10

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git log --graph --oneline --decorate --all

\* c2c9d53 (HEAD -> server) add C10

\* cce1ccf add C4

\* 7f3d267 add C3

\* b7d8d8c (master, client) add C9

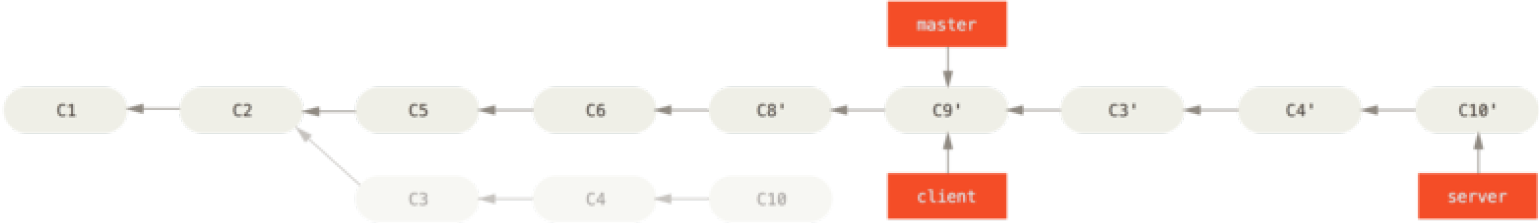
\* b6815ae add C8

\* 35c0fd7 add C6

\* b512f11 add C5

\* 3aaaf38 add C2

\* 498984f add C1



Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (server)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git merge server

Updating 4334288..a89a080

Fast-forward

C10 | 0

C3 | 0

C4 | 0

3 files changed, 0 insertions(+), 0 deletions(-)

create mode 100644 C10

create mode 100644 C3

create mode 100644 C4

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* a89a080 (HEAD -> master, server) add C10

\* e37bcdb add C4

\* f9e572e add C3

\* 4334288 (client) add C9

\* 2a17ec3 add C8

\* 3d1a697 add C6

\* c62e2da add C5

\* 3016553 add C2

\* 42a1b81 add C1

删除分支：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git branch -d client

error: Cannot delete branch 'client' checked out at 'D:/Git/GitRebase'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git branch -d server

Deleted branch server (was a89a080).

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git branch -d client

error: Cannot delete branch 'client' checked out at 'D:/Git/GitRebase'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git log --graph --all --decorate --oneline

\* a89a080 (HEAD -> client, master) add C10

\* e37bcdb add C4

\* f9e572e add C3

\* 4334288 add C9

\* 2a17ec3 add C8

\* 3d1a697 add C6

\* c62e2da add C5

\* 3016553 add C2

\* 42a1b81 add C1

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (client)

$ git switch master

Switched to branch 'master'

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git branch -d client

Deleted branch client (was a89a080).

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitRebase (master)

$ git log --graph --all --decorate --oneline

\* a89a080 (HEAD -> master) add C10

\* e37bcdb add C4

\* f9e572e add C3

\* 4334288 add C9

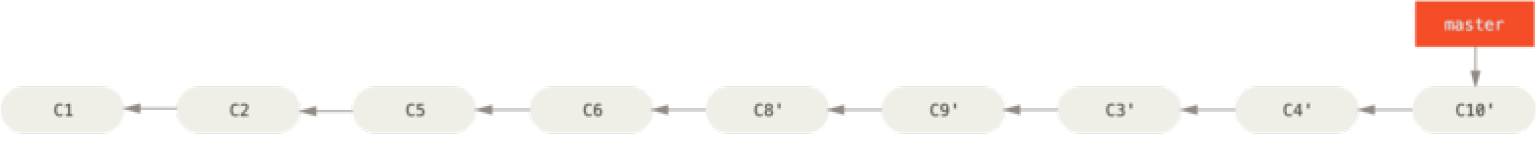
\* 2a17ec3 add C8

\* 3d1a697 add C6

\* c62e2da add C5

\* 3016553 add C2

\* 42a1b81 add C1



### 变基的风险

变基也并非完美无缺，要用它得遵守一条准则：不要对在你的仓库外有副本的分支执行变基。

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。 如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用git rebase命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

# 打标签

Git使用两种主要类型的标签：轻量标签(lightweight)与附注标签(annotated)。一个轻量标签很像一个不会改变的分支——它只是一个特定提交的引用。然而，附注标签是存储在Git数据库中的一个完整对象。 它们是可以被校验的；其中包含打标签者的名字、电子邮件地址、日期时间；还有一个标签信息；并且可以使用GNU Privacy Guard(GPG)签名与验证。通常建议创建附注标签，这样你可以拥有以上所有信息；但是如果你只是想用一个临时的标签，或者因为某些原因不想要保存那些信息，轻量标签也是可用的。

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。

Gi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？

“请把上周一的那个版本打包发布，commit号是6a5819e...”。“一串乱七八糟的数字不好找！”如果换一个办法：“请把上周一的那个版本打包发布，版本号是v1.2”“好的，按照tag v1.2查找commit就行！”所以，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。

## 创建标签

首先，切换到需要打标签的分支上：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git log --graph --oneline --decorate --all

\* 838ea72 (HEAD -> master) add 'Modify readme in master'

| \* 640766e (dev) add 'Creating a branch' in readme.txt

|/

\* 6dc93f9 fix conflict

|\

| \* 706e482 add heading2 in index

| \* 05e8c2e add index.html

\* | 812f2a2 add a heading in index.html

|/

\* 05ee3e4 add readme.txt

\* c0be1c4 add README

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag v1.0

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag

v1.0

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag v0.9 640766e

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag

v0.9

v1.0

标签不是按时间顺序列出，而是按字母排序的。可以用*git show <tagname>*查看标签信息：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git show v0.9

commit 640766e92066468098f1cfd49b221dcec0bbe85d (tag: v0.9, dev)

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Sun Feb 9 11:12:12 2020 +0800

add 'Creating a branch' in readme.txt

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

**index 74e5e69..7276bf9 100644**

**--- a/readme.txt**

**+++ b/readme.txt**

@@ -1 +1,2 @@

My project

+Creating a dev branch

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag -a v0.9 -m "add 'Creating a branch' in readme.txt" 640766e

fatal: tag 'v0.9' already exists

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag -a v0.8 -m "fix conflict" 6dc93f9

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git show v0.8

tag v0.8

Tagger: Chenzf <chenzufeng2019@gmail.com>

Date: Sun Feb 9 12:40:20 2020 +0800

fix conflict

commit 6dc93f934b165e695eff6fa2bfece0e5d84e5a89 (tag: v0.8)

Merge: 812f2a2 706e482

Author: Chenzf <chenzufeng2019@gmail.com>

Date: Sat Feb 8 12:32:49 2020 +0800

fix conflict

**diff --cc index.html**

**index d5dc639,52e2dec..51ccfa8**

**--- a/index.html**

**+++ b/index.html**

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

+<h1>This is a heading</h1>

+ <h2>This is a heading</h2>

## 删除标签

如果标签打错了，也可以删除：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag -d v0.8

Deleted tag 'v0.8' (was 3766e92)

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitMerge (master)

$ git tag

v0.9

v1.0

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。

## 推送标签到远程

如果要推送某个标签到远程，使用命令git push origin <tagname>：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git push origin v1.0

Enumerating objects: 4, done.

Counting objects: 100% (4/4), done.

Delta compression using up to 8 threads

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

Writing objects: 100% (3/3), 294 bytes | 294.00 KiB/s, done.

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

To github.com:Chenzf2018/GitClone.git

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

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git log --graph --oneline --decorate --all

\* cfc5dc4 (HEAD -> master, tag: v1.0) add readme.txt in /d/Git/GitClone

| \* 517eb1a (origin/master, origin/HEAD) add readme.txt in /d/GitClone

|/

\* 60f5e2c Initial commit

伙伴pull：

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git log --graph --oneline --decorate --all

\* 517eb1a (HEAD -> master, origin/master, origin/HEAD) add readme.txt in /d/GitClone

\* 60f5e2c Initial commit

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git pull

remote: Enumerating objects: 8, done.

remote: Counting objects: 100% (8/8), done.

remote: Compressing objects: 100% (3/3), done.

remote: Total 5 (delta 1), reused 5 (delta 1), pack-reused 0

Unpacking objects: 100% (5/5), 608 bytes | 32.00 KiB/s, done.

From github.com:Chenzf2018/GitClone

517eb1a..c89c5ef master -> origin/master

\* [new tag] v1.1 -> v1.1

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

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

Updating 517eb1a..c89c5ef

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

Chenzf@Chenzf-desktop MINGW64 /d/GitClone (master)

$ git log --graph --oneline --decorate --all

\* c89c5ef (HEAD -> master, tag: v1.1, origin/master, origin/HEAD) Merge branch 'master' of github.com:Chenzf2018/GitClone

|\

| \* 517eb1a add readme.txt in /d/GitClone

\* | c2729d2 add contents in readme.txt

\* | cfc5dc4 (tag: v1.0, tag: v0.9) add readme.txt in /d/Git/GitClone

|/

\* 60f5e2c Initial commit

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：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git tag -d v1.1

Deleted tag 'v1.1' (was c89c5ef)

然后删除远程的tag：

Chenzf@Chenzf-desktop MINGW64 /d/Git/GitClone (master)

$ git push origin :refs/tags/v1.1

To github.com:Chenzf2018/GitClone.git

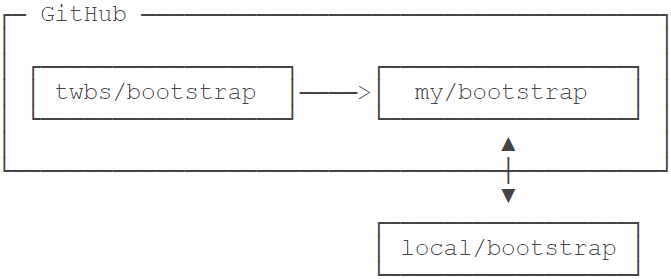
- [deleted] v1.1

# 参与开源项目

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:自己的账号/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。

# 其他问题

## Push的文件太大

