

1.1

问题

1.2

阅读进展

中文版：38,40,48，61，75，83，93，96，98，103，106[ch6],181,199，206,213

1.3

Chapter 1:Getting Started

1.3.1

1 关于版本控制

版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统

版本控制系统缩写是VCS

本地版本控制系统，集中化的版本控制系统，分布式的版本控制系统

1.3.2

2 Git简史

2002年，使用BitKeeper（分布式的）。

2005年，开发公司不许linux 内核社区免费使用，故开发自己的版本控制系统，这就是Git,2005年就投入使用。.

1.3.3

3 Git基础

- (1) 直接记录快照
- (2) 几乎所有操作在本地执行
- (3) 保证完整性（哈希值）

(4) 三种状态：已提交，已修改，已暂存，

1.3.3.1

三个 区域：工作目录，暂存区，Git库

已提交，在Git库中。

已修改，在工作目录中。

已暂存，在暂存区中。

从Git库中检出，用check out命令

将文件暂存，用add命令

将文件提交，用commit命令。

基本的 Git 工作流程如下：

- 1.在工作目录中修改文件。
 - 2.暂存文件，将文件的快照放入暂存区域。
 - 3.提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目 录。
-

1.3.4

4 命令行

1.3.5

5 安装

1.3.5.1

1 linux

如果你想在 Linux 上用二进制安装程序来安装 Git，可以使用发行版包含的 基础软件包管理工具来安装。

如果以 Fedora 上为例，你可以使用 yum：

```
$ sudo yum install git
```

如果你在基于 Debian 的发行版上，请尝试用 apt-get：

```
$ sudo apt-get install git
```

要了解更多选择，Git 官方网站上有在各种 Unix 风格的系统上安装步 骤，

<http://git-scm.com/download/linux>。

1.3.5.2

2 Mac

在 Mac 上安装 在 Mac 上安装 Git 有多种方式。

- (1) 最简单的方法是安装 Xcode Command Line Tools。

Mavericks (10.9) 或更高版本的系统中，在 Terminal 里尝试 首次运行 git 命令即可。如果没有安装过命令行开发者工具，将会提示你安装。

- (2) 如果你想安装更新的版本，可以使用二进制安装程序。

官方维护的 OSXGit 安装程序可以在 Git 官方网站下载，网址为

<http://git-scm.com/download/mac>

- (3) 你也可以将它作为 GitHub for Mac 的一部分来安装。

GitHub for Mac 的图形化 Git 工具有一个安装命令行工具的选项。

可以从 GitHub for Mac 网站下载该 工具，网址

<http://mac.github.com> 。

1.3.5.3

3 windows

下列两个方法都可以

- (1) Git for Windows的项目（也叫做 msysGit）,打开：

<http://git-scm.com/download/win> ,

下载会自动开始。

要注意 它和 Git 是分别独立的项目；

更多信息请访问 <http://msysgit.github.io/> 。

- (2) 安装 GitHub for Windows

该安装程序包含图形化和命令行版本的 Git。 它也能支持 Powershell，提供了稳定的凭证

在 GitHub for Windows 网站下载，网址

<http://windows.github.com>

1.3.5.4

4 从源代码安装

1.3.5.5

5 设置

1.3.5.6

1 linux

1.3.5.7

1 linux上安装

如果你想在 Linux 上用二进制安装程序来安装 Git，可以使用发行版包含的 基础软件包管理工具来安装。

如果以 Fedora 上为例，你可以使用 yum：

```
$ sudo yum install git
```

如果你在基于 Debian 的发行版上，请尝试用 apt-get：

```
$ sudo apt-get install git
```

更多选择：

<http://git-scm.com/download/linux>。

1.3.6

6 设置

(1) 目录结构 (windows下默认安装，其它的可能相似)

在program files下，有Git目录

下面有目录bin,cmd,dev,etc,mingw64,tmp,usr

(2) 配置文件

Git 自带一个 git config 的工具来帮助设置控制 Git 外观和行为的配置 变量。

这些变量存储在三个不同的位置：

A. /etc/gitconfig 文件: 包含系统上每一个用户及他们仓库的通用配置。如果使用带有 --system 选项的 git config 时，它会从此文件 读写配置变量。

【自己机器上没有看到这个文件，可能是没有使用相应的git.config命令

也可能是版本升级了，作了修改，因为看到B里面的文件包含了这些信息】

B. ~/.gitconfig 或 ~/.config/git/config 文件：

只针对当前用户。可以传递 --global 选项让 Git 读写此文件。

【这个看到了，在C盘的user目录下，有当前用户的目录，里面有.gitconfig文件。】

C. 当前使用仓库的 Git 目录中的 config 文件（就是 .git/config）：针对该仓库。

【这个文件看到了，.git目录是隐藏项目】

(3) 下级配置覆盖上级配置

(4) 设置用户信息

例如：

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email johndoe@example.com
```

如果使用了 --global 选项，那么该命令只需要运行一次。

这两个命令是改变C盘上用户目录中的配置文件。

(5) 配置文本编辑器

在win10必须配置，否则在以后执行commit命令会出问题，系统将把命令窗口作为向文本编辑器输出的地方。

自己执行的是：

```
$ git config --global core.editor notepad
```

(6) 查看配置

查看全部：git config --list

查看单个：git config <key>

(7) 获得帮助

```
$ git help <verb>
```

```
$ git <verb> --help
```

```
$ man git-<verb> 【这个windows上不行，可能是linux上的命令】
```

1.4

Chapter 2:Git Basics

1.4.1

本章概述

1.4.1.1

1 配置并初始化一个仓库 (repository)

初始化命令 git init

或 git clone

2 开始或停止跟踪 (track) 文件

开始跟踪命令 git add

停止跟踪命令 git rm --cached

3 暂存 (stage) 或提交 (commit)更改。

```
git add
```

```
git commit
```

4 如何配置 Git 来忽略指定的文件和文件模式

写文件.gitignore

5 如何迅速而简单地撤销错误操作

```
git commit --amend
```

```
git reset HEAD <文件名>
```

```
git checkout -- <文件名>
```

6 如何浏览你的项目的历史版本以及不同提交 (commits) 间的差异、

```
git log
```

7 如何向你的远程仓库推送 (push)

```
git push
```

8 如何从你的远程仓库拉取 (pull)

```
git pull
```

1.4.2

1 Getting a Git Repository

1.4.2.1

1 Initializing a Repository in an Existing Directory

执行命令：

```
git init
```

如果是空目录，就可以了，如果不是，需要要将文件加入跟踪：

```
$git add *.c
```

```
$git add LICENSE
```

```
$git commit -m 'initial project version'
```

【上面的例子，可能是假定项目是C语言】

1.4.2.2

2 Cloning an Existing Repository

在一个目录中执行克隆命令，则这个命令会在该目录中建立一个子目录，这个子目录就是新仓库。

```
$ git clone https://github.com/libgit2/libgit2
```

上面的命令建立的子目录名用原来的名字，就是libgit2。

也可以取新的名字，如：

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

mylibgit就是新的名字。

【中文版p97上说了参数-o

原文：

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`。

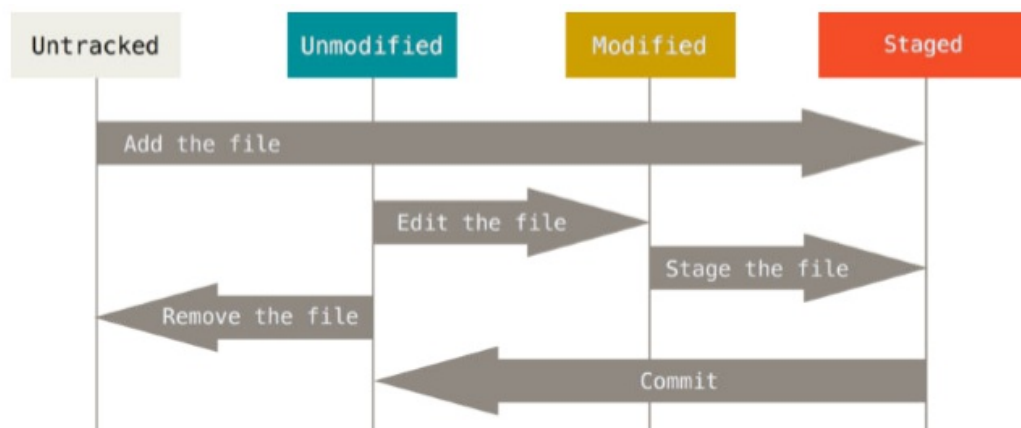
】

1.4.3

2 Recording Changes to the Repository

如果通过初始化一个现存的目录建立仓库，则目录的所有文件都没有被跟踪，需要通过命令来加入跟踪。

如果通过克隆已有仓库建立仓库，则生成的目录中，所有文件都被跟踪。



untracked到staged的命令是add(比如将新建的文件加入)

unmodified到modified，用的是编辑软件

staged到unmodified的命令是commit

问题1：modified到staged的命令是什么？（也是add，修改一个文件后，须使用add，才能对修改后的内容跟踪，否则跟踪的是原来的内容，文件虽然被跟踪，但并不反映修改的情况）

问题2:从unmodified到untracked的命令是什么？

用 `git rm <文件名> --cached`

1.4.3.1

1 Checking the Status of Your Files

用命令：

```
$ git status
```

要显示当前分支。

是否有要提交的，如果没有，会显示

```
nothing to commit
```

如果修改过但没有add的文件，会显示 working directory clean

```
nothing to commit, working directory clean
```

经常连起来出来，如果上次提交会没有任何改动。

如果有新文件建立但没有add，则可能这样显示：

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

这里同样也是没有要提交的，但说有未跟踪的文件存在。

1.4.3.2

2 Tracking New Files

```
$ git add <文件名>
```

可以跟多个文件名

例如：`$ git add README`

这是把untracked加入到staged

这里如果执行git status，就会是下面这个样子：

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Changes to be committed:下面是add了但没有commit的文件。

同时也告知了取消暂存的命令（可以后悔）

书上有一句：`git add` 命令使用文件或目录的路径作为参数；如果参数是目录的路径，该命令将递归地跟踪该目录下的所有文件。

对于文件，这句话是清楚的。

对于目录，它的意思是跟踪该目录下现在的所有文件，并不包括以后加入该目录的文件。

就是说，并不是这个目录被纳入跟踪，只是其当前的文件被跟踪。如果这个目录以后建立文件，它们是不会因现在执行的命令而被跟踪的。

这也说明一个事实：如果在本地建立目录，并把这个目录作为git add的参数，再commit，然后push到远程仓库（比如github上），并不会在远程仓库中建立这个目录。必须在目录中建立文件才可以。

1.4.3.3

3 Staging Modified Files

如果文件修改了，不管以前是否add了，都必须add。

就是说，无论是新建的文件，还是修改已有文件，要加入跟踪，都必须add。

原来add过的文件，修改后必须再次add，这是将新内容加入跟踪。

注意git总是保留完整的文件快照，可以认为修改后的文件已经是与原来文件不同的文件，其实相当于新建文件。git不是保存文件差异，因此新建文件与修改后的文件，本质上一样。

1.4.3.4

4 Short Status

```
$ git status -s
```

参数-s表示short

简化的状态显示形式。

A等级最高，表示放到了暂存区中，A是add之略。

M分左右，左边的等级高，表示修改后放进暂存区。右边的表示修改后还未放进暂存区。

M是modify之略，表示文件被修改了的事实。

有可能有两个M同时出现的情形。

另外还有??，表示未加入跟踪的文件。前面的文件都是被跟踪的，只是可能有又被修改了的，与??表示的不同。

所以共有四个符号：A,左M，右M，??

1.4.3.5

5 Ignoring Files

书中举的命令

```
$ cat .gitignore
```

中的cat不能在windows中使用，可能是linux的命令，不管。

这节讲的是文件.gitignore的内容怎么写。

这个文件，在本地仓库中目录中，可以手工建立。

gitHub有个.gitignore文件列表，可以参考。

<https://github.com/github/gitignore>

它其实是个github上的仓库。

1.4.3.6

6 Viewing Your Staged and Unstaged Changes

说的两件事，

一是查看未暂存的修改内容（修改后未执行add命令）

命令是：`$ git diff`

二是查看已暂存的修改内容（修改后已执行add命令）

命令是：`$ git diff --cached`

或 `$ git diff --staged`

1.4.3.7

7 Committing Your Changes

`$ git commit`

这个命令会打开一个编辑器（在前面设置的）

其中最上一行的空白是留来写提交说明的。后面都是注释，以后不会保留，但现在可以参考。

我的实践中，编辑器一定要有前面设置（win10环境），否则它会将命令窗口用作编辑器，就没有办法往下走了。

如果不想出来编辑器，使用：

`$ git commit -m <提交说明字符串>`

1.4.3.8

8 Skipping the Staging Area

1.4.3.9

9 Removing Files

使用命令 `git rm <文件名>`

首先，这里讨论的文件，不是未被跟踪的文件。

如果是这种情况，命令不会被执行，会出现如下的信息

`fatal: pathspec 'a.txt' did not match any files`

其中a.txt是想删除的文件，它没有被跟踪

被跟踪的文件，就有三种情况：

（1）未被修改（前面至少提交过一次）

命令会被执行

（2）被修改，但未被add（就是未被staged）

命令不能执行。

一定要执行，可以加-f，或者加--cached保留文件

（3）新建或修改后，staged（使用了add命令）

必须使用参数 -f（强行删除）或加--cached（将其从跟踪项中去除，文件保留）

加了这个参数，删除非常彻底。以后使用git status，都看不出任何痕迹。估计不能恢复。

如果不使用参数-f，则成功执行rm命令，应该执行git commit，来提交这次删除。

删除文件是一种改变，改变就需要提交，与修改文件是一样的。

另外看出，修改后或新建，不管是不是执行了add命令，都不能执行rm，一定要执行，必须加-f或--cached参数。估计是为了怕新作的工作被无意抛弃不能找回。

1.4.3.10

10 Moving Files

```
$ git mv file_from file_to
```

相当于三个命令：

```
$ mv README.md README
```

```
$ git rm README.md
```

```
$ git add README
```

其中，README.md 是源文件，README是目标文件。

mv应该是操作系统的命令，可能是linux中的。不是git的命令。

1.4.4

3 Viewing the Commit History

命令 `git log`

用参数-p，可以看到每次提交的差异。

但是，用了这个参数，就退不出去了，用p才可以退出。

这个可能是从参数来的。

即使在-p 后面加数字参数，也一样退不出去。

都必须输入p

1.4.4.1

参数汇总

- (1) -p 显示每次提交的差异
- (2) -<n> 显示最后n次提交
- (3) --stat 正常提交信息之外，还有每次提交的简略统计信息
- (4) --pretty 有下面几个子项

=oneline 每次提交显示在一行

=short

=full

=fuller

=format

例 `$ git log --pretty=format:"%h - %an, %ar : %s"`

(5) `--graph` 用一些字符显示成图形，形象显示分支及合并历史，常与`--pretty`中的`online`和`format`一起用

(6) `--shortstat` 只显示 `--stat` 中最后的行数修改添加移除统计。

(7) `--name-only` 仅在提交信息后显示已修改的文件清单。

(8) `--name-status` 显示新增、修改、删除的文件清单。

(9) `--abbrev-commit` 仅显示 SHA-1 的前几个字符，而非所有的 40 个字符。

(10) `--relative-date` 使用较短的相对时间显示（比如，“2 weeks ago”）。

以下是限制输出长度

(11) `--since`

例：`$ git log --since=2.weeks` 显示最近两周的提交

(11A) `--after`

(12) `--until`

(12A) `--after`

(13) `-S` 可以列出那些添加或移除了某些字符串的提交。

例：`$ git log -Sfunction_name`

找出添加或移除了某一个特定函数的引用的提交

(14) `--<路径>` 只显示某些文件或目录的历史提交

(15) `--author` 仅显示指定作者相关的提交。

(16) `--committer` 仅显示指定提交者相关的提交。

(17) `--grep` 仅显示含指定关键字的提交 `-S` 仅显示添加或移除了某个关键字的提交

1.4.5

4 Undoing Things

1.4.5.1

1 重新提交

重新提交用在刚提交后，发现两种情况：（1）漏掉了几个文件 （2）提交信息写错了

如果是（1），要用`add`命令把漏掉的文件加上，然后执行带参数`--amend`的提交命令

如果是（2），直接执行带参数--amend的提交命令，修改提交信息

```
$ git commit --amend
```

第二个提交将覆盖第一个提交

--amend也可以与-m一起使用

1.4.5.2

2 Unstaging a Staged File

```
$ git reset HEAD <文件名>
```

其中，文件名所指的文件之前被执行了add命令，就是加入了暂存区

【书中说加了--hard参数，这个命令就很危险，译注说是可能导致工作目录中所有当前进度丢失！但在这里，并没有说明使用这个参数是什么意思，从上下文看也没有必要在这里使用这个参数。不加这个参数，命令是安全的】

1.4.5.3

3 Unmodifying a Modified File

```
$ git checkout -- <文件名>
```

可以放弃对一个文件的修改（这个修改在工作区进行）

例如：\$ git checkout -- CONTRIBUTING.md

这个命令很危险，对该文件所作的修改会丢失，不能找回

1.4.6

5 Working with Remotes

1.4.6.1

1 Showing Your Remotes

```
$ git remote
```

会显示远程仓库的名字

可以加参数-v，这样还会显示链接地址。

链接后面的fetch和push应该是说明权限。但书中没有解释，不知为何。

1.4.6.2

2 Adding Remote Repositories

```
$ git remote add <shortname> <url>
```

1.4.6.3

3 Fetching and Pulling from Your Remotes

(1) 从远程仓库抓取数据：

```
$ git fetch [remote-name]
```

这个命令会访问远程仓库，从中拉取所有你还没有的数据。执行完成后，你将会拥有那个远程仓库中所有分支的引用，可以随时合并或查看。

(2) 从远程仓库拉取数据

```
$ git pull [remote-name] [分支名]:[本地分支]
```

fetch，并不合并到本地分支，以后需要手动。

如果一起完成，就使用git pull 命令

如果是合并到本地当前分支，则省略冒号及以后

http://www.yiibai.com/git/git_pull.html

1.4.6.4

4 Pushing to Your Remotes

```
$ git push [remote-name] [branch-name]
```

这个命令执行的条件是：对服务器有写的权限，并且之前没有人推送过。

如果是，则需要拉取数据下来，合并后才能推送。

1.4.6.5

5 Inspecting a Remote

```
git remote show [remote-name]
```

这个命令，如果不跟参数remote-name，跟前面的命令git remote的作用是一样的。

【如果只显示某一个远程仓库，就用这里的命令。前面那个命令在介绍时，还没有说到仓库的名字】

1.4.6.6

6 Removing and Renaming Remotes

(1) 移除远程仓库

```
$ git remote rm [remote-name]
```

(2) 重命名远程仓库

```
$ git remote rename <旧名> <新名>
```

1.4.7

6 Tagging

Git 可以给历史中的某一个提交打上标 签，以示重要。

Git 使用两种主要类型的标签：轻量标签（lightweight）与附注标签（annotated）

前者简明，后者完整

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

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。

【一个提交有说明，还有哈希值，标签可能主要是支持软件版本的角度引进的】

1.4.7.1

1 Listing Your Tags

```
$ git tag
```

可以有参数-l，用于限制显示的标签，如：

```
$ git tag -l 'v1.8.5*'
```

1.4.7.2

2 Creating Annotated Tags

```
$ git tag -a <标签名>
```

这个命令会开启编辑器，要求输入一条存储在标签中的信息。类似于commit命令要求输入的信息。而且也可以用-m在命令中给出。

例如：

```
$ git tag -a v1.4 -m 'my version 1.4'
```

1.4.7.3

3 Creating Lightweight Tags

```
$ git tag <标签名>
```

与建立附注标签的区别是没有-a参数。

它也不需要-m,-s参数

【这里提到-s是什么意思？因为2里面并没有说到-s】

1.4.7.4

4 Tagging Later

之后补标签，需要加上校验和（每次提交都有的），例如：

```
$ git tag -a v1.2 9fceb02
```

1.4.7.5

5 Sharing Tags

推送到远程服务器上

推送单个标签的命令是：

```
$ git push <仓库名> [tagname]
```

推送所有标签的命令是

```
$ git push <仓库名> --tags
```

1.4.7.6

6 Checking out Tags

```
$ git checkout -b <分支名> <标签>
```

例如：

```
$ git checkout -b version2 v2.0.0
```

但它不能真的检出标签，只能创建一个标签恰好在那儿的分支，但分支移动后，标签并不会跟随。

1.4.8

7 Git Aliases

这是为了减少命令输入的长度，将命令设置成较短的字符串，如：

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

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

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

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

如果要简化的是两个词组成的，外部加单引号，如：

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

如果要模拟外部命令，加!后，外面再加单引号：

```
$ git config --global alias.visual '!gitk'
```

1.4.9

8 Summary

1.5

Chapter 3:Git Branching

1.5.1

1 Branches in a Nutshell

提交后，有两个管理对象：树对象和提交对象。

树对象指向数据文件，而提交对象指向树对象。提交对象还包含提交信息。

提交对象是核心，每一次提交，会包含指向上一次提交对象的指针。

Git 的分支，其实本质上仅仅是指向提交对象的可变指针，默认名字是master。

1.5.1.1

1 Creating a New Branch

`$ git branch <分支名>`

这个分支指向当前位置（HEAD指向的位置）。

但git并不切换到这个分支。建立的时候它指向当前位置，有了新提交后，它会落后于当前位置了，因为当前位置会变。

查看分支用git log命令，但是要加参数--decorate，例如：

```
$ git log --oneline --decorate
```

下面是一个输出的例子：

```
f26cb11 (HEAD -> master) dfd
```

```
6ca0d4d Merge branch 'test' 465
```

```
c99ab00 dfd
```

```
b5f8436 sf
```

```
82543d3 df
```

```
4254663 Merge branch 'test'
```

```
61aaf96 dfd
```

```
8225688 分解
```

```
8b3fe28 11 Merge branch 'test'
```

```
3e7208f 11
```

```
027559a 111
```

11eb28d 11 Merge branch 'test'

右边是提交时的信息。

【问题：可不可以建立指向任意提交的分支？上面的命令只能创建指向当前位置的分支】

1.5.1.2

2 Switching Branches

```
$ git checkout <分支名>
```

这个命令移动HEAD指针，指向分支名代表的分支。

就是将当前分支改变为参数中所指的分支。

1.5.1.3

3 合并创建与切换命令

```
$ git checkout -b <分支名>
```

这个命令新建一个分支，并切换到这个分支，相当于前面两个命令连续执行

1.5.2

2 Basic Branching and Merging

1.5.2.1

1 Basic Branching

新建与切换分支很容易

此节介绍了一个实例，涉及到分支的新建，合并，删除。

合并命令是：

```
$ git merge <分支名>
```

是将指定分支合并到当前分支，举的例子是当前分支落后于指定分支且在一条线上，成为快进。

删除是：

```
$ git branch -d <分支名>
```

先前看到，建立分支也是这个命令，只是这里加了一个参数-d

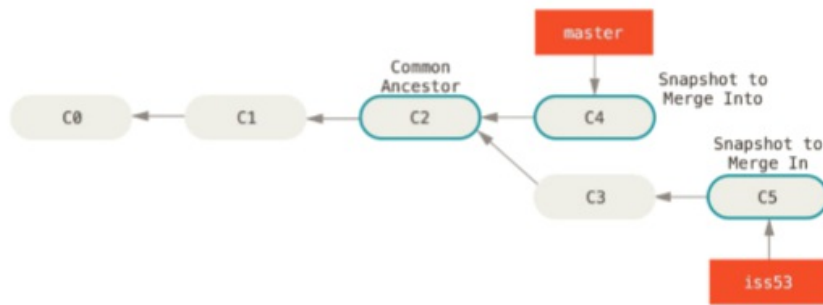
1.5.2.2

2 Basic Merging

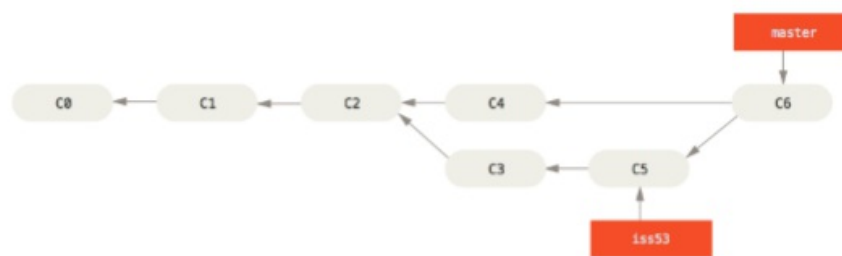
不在一条线上的合并。

```
$ git checkout master
```

```
$ git merge iss53
```



合并C4与C5得到C6：



1.5.2.3

3 Basic Merge Conflicts

如果同一个文件的同一个地方，不同的分支有不同的内容，就造成合并冲突。

此时，系统不能完成合并命令。

它将冲突的文件中冲突的地方标出。

可以手动处理，将标出的地方，保留其中一个内容。

对每个文件这样处理后，将它们add,

add完后，再commit。

不需要再执行git merge命令了。

注意对文件的手动修改，是在当前分支中进行的。

例如，执行了这两个命令：

```
$ git checkout master
```

```
$ git merge iss53
```

则是在分支master中修改文件的，它是当前分支。

那么要修改哪些文件呢？

如果合并遇到冲突，git提示合并失败。

这时执行git status,git会列出冲突文件。

这些文件就是要修改的。

手动处理后，add冲突文件。

此时可以执行git status，确认冲突是不是处理完了。

满意后，就执行git commit

1.5.3

3 Branch Management

1.5.3.1

1 显示所有分支

```
$ git branch
```

这个命令不带参数，就显示分支，但是只有名字。

前面有*表示当前分支。

1.5.3.2

2 显示分支的最后一次提交

```
$ git branch -v
```

除了1的功能外，它还显示最后一次提交的校验和以及提交的注释信息。

1.5.3.3

3 显示所有已经合并的分支

```
$ git branch --merged
```

其中，没有星号的分支，可以删除。

4 显示所有没有合并的分支

```
$ git branch --no-merged
```

显示的分支，都不能用带-d的git branch命令删除，但可以用带-D参数的git branch命令删除。

1.5.3.4

5 删除分支

\$ git branch -d <分支名> 删除已经合并的分支

\$ git branch -D <分支名> 删除没有合并的分支（将丢弃所作的工作，有危险）

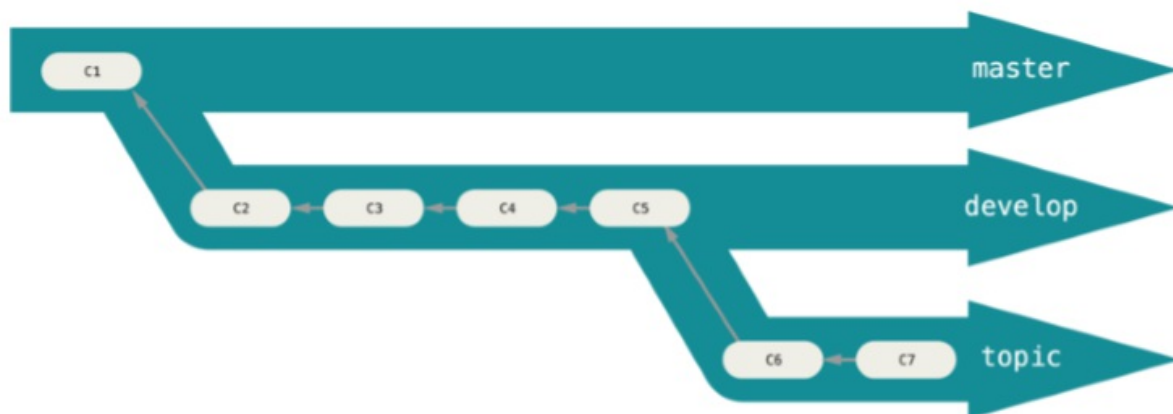
1.5.4

4 Branching Workflows

1.5.4.1

1 Long-Running Branches

可以在开发中保持多个长期分支



1.5.4.2

2 Topic Branches

特性分支对任何规模的项目都适用。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。

1.5.5

5 Remote Branches

这节的内容与第二章第五节：Working with Remotes 有相同之处。

命令

\$ git ls-remote (remote)

获取远程仓库列表

这个命令与git remote的区别是，要访问远程仓库，得到完整校验和

这是同样情况两个命令的返回信息：

git ls-remote 的

```
-----  
From https://github.com/huangblue/hello-world  
4e4605c6ad202f817354da2b20f700d9bee42f4a HEAD  
70b90447e00a7582e0bfc473f268a8ba8efabf59 refs/heads/asdff  
4e4605c6ad202f817354da2b20f700d9bee42f4a refs/heads/master  
e5d8e1bc99a54391ebaace7cbcba7a9737b669d7 refs/pull/1/head  
70b90447e00a7582e0bfc473f268a8ba8efabf59 refs/pull/4/head
```

git remote的

```
-----  
git  
kn  
origin
```

可见，区别还是很大的

\$ git remote show (remote)

获取远程仓库完整信息（链接）

【这个与git rmote 效果一样，要得到链接要加参数-v】

问题：怎样查看远程跟踪分支？

原文：

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。

经反复测试，能执行命令：

git ls-remote 它显示所有远程仓库，它是要去读远程仓库的。

git ls-remote remote 它显示指定的远程仓库，也要读的。

你可以查看 origin/ master 分支

这句话，指的是读出远程仓库后，查看master分支，而不是只读一个分支master

这个命令是以仓库为单位来读的。

ls-remote中的ls，应该是list之略。

书中多次说到本地能得到远程分支，估计是这个命令：

\$ git fetch <远程仓库名>

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

这个命令会在本地得到指向远程分支的指针，它的功能是能合并到本地分支中：

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

这个命令的最后一个参数，就是本地的远程分支指针。

它与git ls-remote得到的不一样，那是查看远程仓库的，这是确实在本地的。只能通过fetch命令取得。

上面这个命令，得到一个本地分支，它是远程分支的副本。注意是由fetch命令先取来的。

也可以用merge命令将fetch命令取得的远程分支合并到本地分支,例如：

```
$ git merge origin/serverfix
```

1.5.5.1

总结

1.5.5.1.1

1 加入远程仓库

```
$ git remote add <shortname> <url>
```

远程仓库名在这里定义

2 移除远程仓库

```
$ git remote rm [remote-name]
```

3 重命名远程仓库

```
$ git remote rename <旧名> <新名>
```

4 clone取一个仓库的数据

```
$ git clone <远程仓库网址>
```

例

```
$ git clone https://github.com/libgit2/libgit2
```

这里用原来的仓库名字

```
$ git clone <远程仓库网址> <新名字>
```

例

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

5 fetch命令

```
$ git fetch [remote-name] [分支名]
```

完整的命令，取一个仓库中的一个分支

省略分支名，则取该仓库上的所有分支

用参数--all，不用远程仓库名，则取所有远程仓库上的所有分支

6 远程分支的指针

fetch,clone

会得到远程分支的指针：

远程仓库名/远程分支名

例如 origin/master

这些指针保存在本地。

7 设置跟踪远程分支

就是用本地分支跟踪远程分支

```
$ git checkout --track origin/serverfix (同名)
```

```
$ git checkout -b sf origin/serverfix (不同名)
```

同一个远程分支只能由一个本地分支跟踪（如果已经有分支跟踪，执行同样的命令让另一个分支跟踪会出错）

8 查看远程分支

```
$ git branch -r
```

```
$ git ls-remote [远程仓库名]
```

```
$ git remote show (remote)
```

另外，查看所有分支是

```
$ git branch -a
```

9 使用远程分支

将其合并到本地分支，用merge命令

例如：

```
$ git merge origin/master
```

10 pull命令

```
$ git pull <远程主机名> <远程分支名>:<本地分支名>
```

去掉冒号后，与当前分支合并

如：\$ git pull origin next

如果再去掉远程分支名，表示当前分支与远程仓库中的跟踪分支合并，如：

```
$ git pull origin
```

如果再去掉远程仓库名，则表示唯一跟踪分支与当前分支合并

```
$ git pull
```

1.5.5.2

1 Pushing

git push (remote) (branch)

例如：

```
$ git push origin serverfix
```

意味着，推送本地的 serverfix 分支来更新远程仓库上的 serverfix 分支。

另一个格式，

```
git push (remote) (local branch):(tele branch)
```

例如：

```
git push origin serverfix:awesomebranch
```

是把本地的serverfix推送远程仓库的awesomebranch上。

按从左到右理解

避免每次输入帐号密码：

```
git config --global credential.helper cache
```

1.5.5.3

2 Tracking Branches

(1)设置跟踪分支

```
$ git checkout --track origin/serverfix
```

```
$ git checkout -b sf origin/serverfix
```

两个命令主要目的是一样的，建立一个跟踪远程分支的本地分支。

不同的是，后一个命令另外为本地分支取一个名字，而前一个命令就用远程仓库上的分支名。

【问题：这里的远程分支，需要之前用fetch或clone命令取得吗？】

(书中下面这段,看起来是说之前拉取下来的：

设置已有的本地分支跟踪一个刚刚拉取下来的远程分支，或者想要修改 正在跟踪的上游分支，你可以在任意时间使用 -u 或 -set-upstream-to 选项运行 git branch 来显式地设置。

```
$ git branch -u origin/serverfix Branch serverfix
```

set up to track remote branch serverfix from origin.

)

当设置好跟踪分支后，可以通过 @{upstream} 或 @{u} 快捷方式来引用它。所以在 master 分支时并且它正在跟踪 origin/master 时，如果愿意的话可以使用 git merge @{u} 来取代 git merge origin/master。

(2)查看跟踪分支

```
$ git branch -vv
```

得到的返回信息里面，有方括号的是跟踪远程的分支，其它的不是。

【自己执行的命令，括号里面是空的，或者只有落后领先的数据，没有远程分支的名字，而书上的有，不知道是什么原因？】

领先是本地没有提交上去的，而落后是服务器没有拉取的。

这个数据不是最新的，这个命令也不连接服务器。

如果要看最新的，执行命令：

```
$ git fetch --all
```

```
$ git branch -vv
```

【这就说明，设置跟踪之前需要抓取，不管是用fetch,还是clone】

1.5.5.4

3 Pulling

命令 git pull

相当于fetch命令紧跟merge命令。

一般提倡分开用。

1.5.5.5

4 Deleting Remote Branches

```
$ git push <远程仓库名>--delete <分支名>
```

例如：

```
$ git push origin --delete serverfix
```

这个命令会联网，删除的是服务器上仓库的分支。

1.5.6

6 Rebasing

合并分支的另一种方法。

1.5.6.1

rebase命令格式解释：

涉及的两个最重要的分支是基底分支和变基分支。

命令中，基底分支写在前面，变基分支写在后面。

如果当前分支是变基分支，则不用写出，这样命令中只有一个基底分支。

重演的提交是两者共同祖先之后的。

如果加上参数--onto，则会出现参考分支，这时，要重演的提交是所有分支的共同祖先之后的，范围比前面的要窄。

```
1 git rebase <基底分支>
```

例如：`git rebase master`

对当前分支（没有写出）进行变基，基底指向命令所示分支。

注意命令中的分支不是当前分支。

变基后，当前分支领先于命令中所示分支，两个分支在一条线上。落后的可以快进（用merge命令）到领先分支。

例如，假定上面例子中的当前分支是experiment，则快进这样作：

`$ git checkout master`

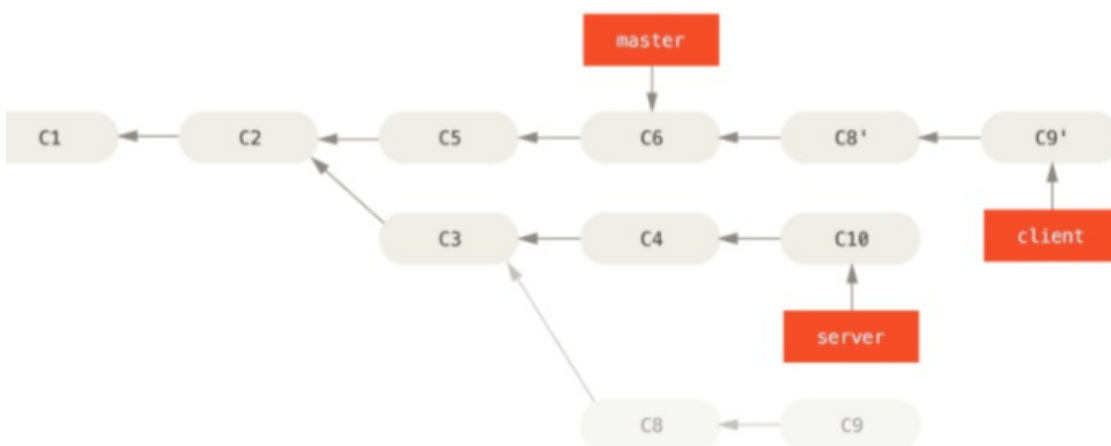
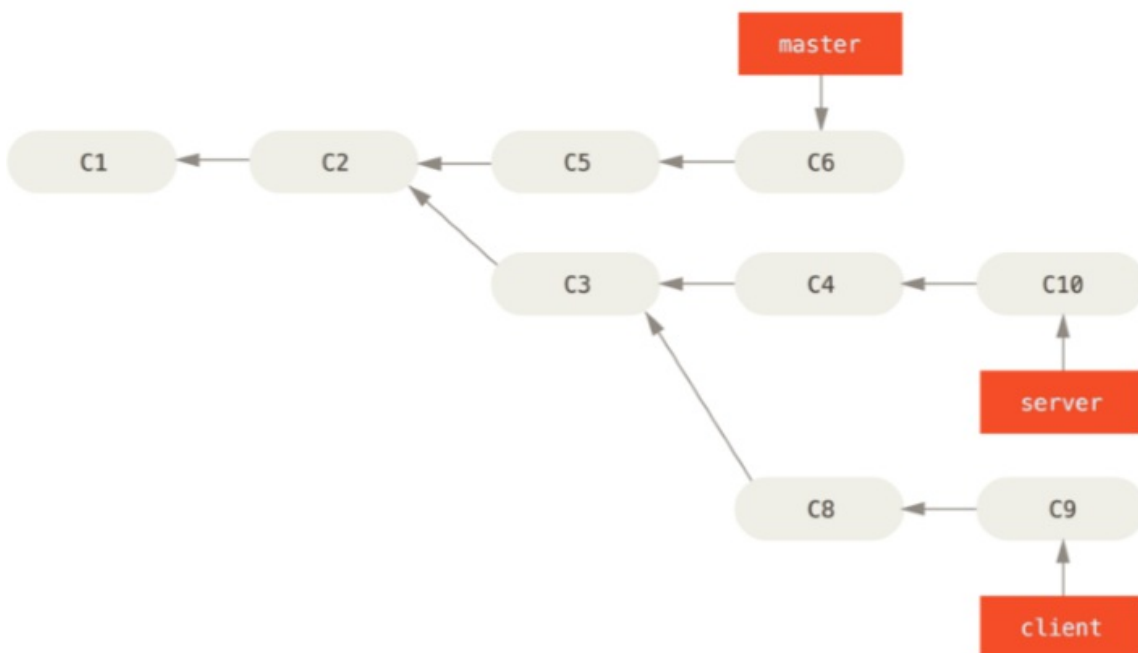
`$ git merge experiment`

2 `git rebase --onto` <基底分支> <参考分支> <变基分支>

例如：`$ git rebase --onto master server client`

找出位于第二、三分支共同祖先之后的修改，将其在第一分支上重演。

要将变基分支在共同祖先之后的指向基底分支。



`$ git rebase --onto master server client`

这个命令中，如果没有server这一项，应该是将C3,C8,C9之后的修改在C6上重演。

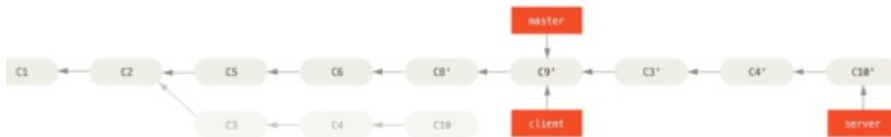
换言之，最上面一行是C6,C3,C8,C9

3 git rebase <基底分支> <变基分支>

例：\$ git rebase master server

将变基分支指向基底分支。

将两个分支共同祖先之后的修改在基底分支上重演。



1.5.6.2

1 The Basic Rebase

问题：合并两个分支，它们有一个共同的祖先

- (1) 先检出一个分支作为当前分支

\$ git checkout experiment

- (2) 将其对另一个分支变基

\$ git rebase master

这实际已经完成了合并，

- (3) 检出另一个分支，并进行快进

\$ git checkout master

\$ git merge experiment

这里，实际上是将experiment与master合并，并最后置当前分支为master.

1.5.6.3

2 More Interesting Rebases

1.5.6.4

3 The Perils of Rebasing

奇妙的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。

这个应该指的就是远程仓库拉下来的分支。这样别人可能就有了。

自己的分支，如果上推到服务器，也是这个情况。

1.5.6.5

4 Rebase When You Rebase

只要你把变基命令当作是在推送前清理提交使之整洁的工具，并且只在从未推送至共用仓库的提交上执行变基命令，你就不会有事。

假如你在那些已经被推送至共用仓库的提交上执行变基命令，并因此丢弃了一些别人的开发所基于的提交，那你就有大麻烦了，你的同事也会因此鄙视你。

如果你或你的同事在某些情形下决意要这么做，请一定要通知每个人执行 `git pull --rebase` 命令，这样尽管不能避免伤痛，但能有所缓解。

1.5.6.6

5 Rebase vs. Merge

总的原则是，只对尚未推送或分享给别人的本地修改执行变基操作清理历史，从不对已推送至别处的提交执行变基操作，这样，你才能享受到两种方式带来的便利。

1.5.7

7 Summary

1.6

Chapter 4:Git on the Server

1.7

Chapter 5:Distributed Git

1.8

Chapter 6:GitHub

1.8.1

1 Account Setup and Configuration

Octocat 是章鱼猫。

1.8.1.1

1 SSH Access

在界面上key框里的文字是：

Begins with 'ssh-rsa','ssh-dss','ssh-ed25519','ecdsa-sha2-nistp256','
'ecdsa-sha2-nistp384',or 'ecdsa-sha2-nistp521'

1.8.1.2

2 Your Avatar

1.8.1.3

3 Your Email Addresses

1.8.1.4

4 Two Factor Authentication

1.8.2

2 Contributing to a Project

1.8.2.1

1 Forking Projects

派生一个项目，就是在自己的账户中得到一个副本

合并请求Pull Request，将自己的分支向原作者推送，注意将本地将服务器推送，git用的命令是push。这里用pull这个词有点拗，别人这样规定了，就这么理解。

1.8.2.2

2 The GitHub Flow

(1)从 master 分支中创建一个新分支

【应该是fork之后克隆到本地，创建新分支是在本地】

(2)提交一些修改来改进项目【在本地上作】

(3)将这个分支推送到 GitHub 上【推送上自己账户的fork仓库中】

(4)创建一个合并请求

(5)讨论，根据实际情况继续修改

(6)项目的拥有者合并或关闭你的合并请求

1.8.2.3

3 Advanced Pull Requests

(1)与远端保持同步

前提：如果合并请求不能成功，是因为与远端不同步，需要按下面步骤作：

加远端加入为远程仓库，拉取，合并到本地，再推送到github。

之后合并请求会自动检查合并能否进行。

(2)参考

在合并请求中引用其它issue或合并请求（它们都有编号，估计是不重号的）

要引用，写在描述中。

【似乎github并不关心文字怎么写，只管里面的编号

完整地引用是： 用户名/ 版本库名#<编号>

如果是自己的，省略用户名，如果引用的是同一个版本库，省略版本库名

用户名，应该是指不同的fork用户

】

【不清楚说的版本库是什么？】

1.8.2.4

4 Markdown

轻量级格式语言

(1)任务列表（复选框）

- []

- [x]

里面是小写的x

(2) 代码

三反引号括住

```<语言名>

```

举了java的例子。效果是作语法渲染

(3)引用

用>号，快捷键是r。

创建comment后来到上面的评论，选择要回应的，按r键即可

(4)表情

用冒号开头，选择后按回车，或鼠标双击

(5)图片

可以拖进去，将文件夹中的图片文件拖进文本编辑框即可。

已经在issues里面试过。

从图片的链接来看，图片被上传到了github的服务器。

我的是

![psb](https://cloud.githubusercontent.com/assets/21062610/20214705/babf486c-a84a-11e6-99d3-a38f907dd2eb.jpg)

但书上的方括号中是git

【(已解决)

为什么里面是git或psd，各代表什么含义？

我的是jpg文件，书的是png文件。

答：方括号里面是表示类型。png是git，jpg是psd

另外，文档也可以，比如pdf，docx等。

文档前面没有!号。

】

1.8.3

3 Maintaining a Project

1.8.3.1

1 Creating a New Repository

最好选择要readme

1.8.3.2

2 Adding Collaborators

在设置里，左边菜单。


在第二个。

1.8.3.3

3 Managing Pull Requests

1.8.3.3.1

1 合并请求邮件

[fade] Wait longer to see the dimming effect better (#1) 

 **Scott Chacon** <notifications@github.com>
to tonychacon/fade [Unsubscribe](#) ▼

10:05 AM (0 minutes ago)

One needs to wait another 10 ms to properly see the fade.

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

<https://github.com/tonychacon/fade/pull/1>

Commit Summary

- wait longer to see the dimming effect better

File Changes

- M [fade.ino](#) (2)

Patch Links:

- <https://github.com/tonychacon/fade/pull/1.patch>
- <https://github.com/tonychacon/fade/pull/1.diff>

Reply to this email directly or [view it on GitHub](#).

【还有一些有趣的 URL，像 .diff 和 .patch，就像你猜的那样，它们提供 diff 和 patch 的标准版本。你可以技术性地用下面的方法合并“合并请求”：

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

这个curl是linux命令

】

1.8.3.3.2

2 在合并请求上进行合作

【(已解决)在任何地方都可以用 GitHub Flavored Markdown

GitHub Flavored Markdown

是什么？

就是github上用的md格式，可以简写为GFM

】

1.8.3.3.3

3 合并的第一种方法：直接用git pull

```
git pull <url> <branch>
```

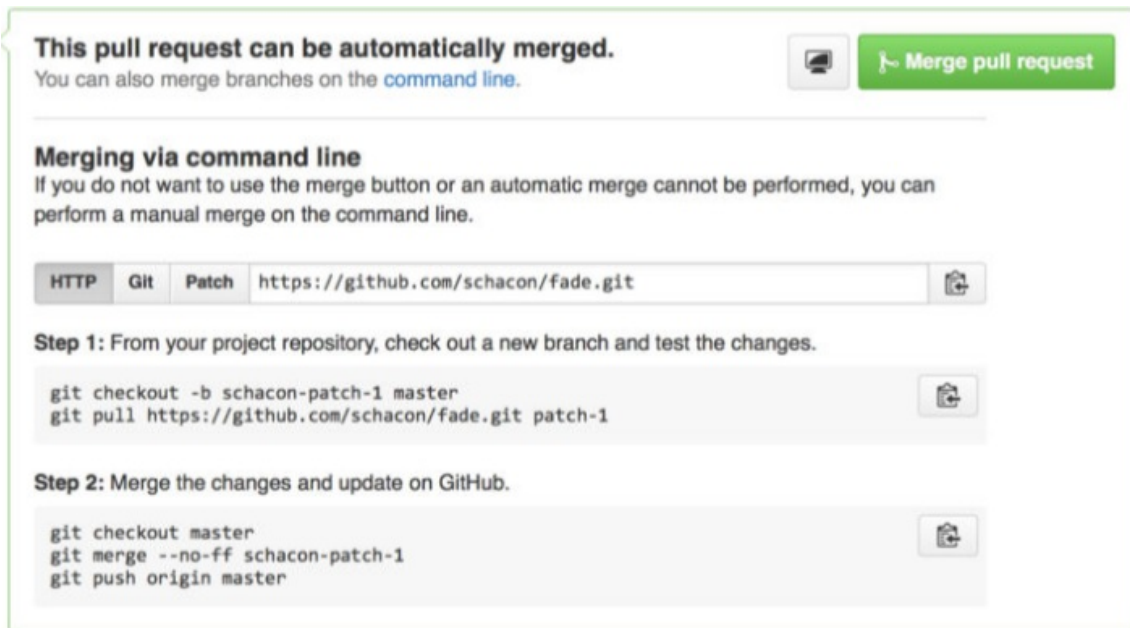
这个命令将对方的分支（远程）合并到自己当前本地分支。

1.8.3.3.4

4 合并的第二种方法：将对方的fork仓库添加为远程仓库，再抓取来合并

1.8.3.3.5

5 合并的第三种方法：用github上的merge按钮。



【merge命令的 --no-ff参数的意义是什么？

可能是不进行快进，但这意味着什么？

】

1.8.3.3.6

6 refs/heads与refs/pull

前者是自己的分支，后者是别人推送的分支。

用git ls-remote可以显示它们的全部，例如下图：

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d      HEAD
10d539600d86723087810ec636870a504f4fee4d      refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e        refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3        refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1        refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d        refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a        refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c        refs/pull/4/merge
```

refs/pull后面跟着/数字/, 代表不同的用户。

1.8.3.3.7

7 合并一个合并请求

- (1) 确定refs/pull分支，可以用git ls-remote

形如pull/<pr#>/head

- (2) 用fetch命令抓取这个分支，如：

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
* branch                refs/pull/958/head -> FETCH_HEAD
```

- (3) 用 git merge FETCH_HEAD

将其合并到自己的分支

这个方法可用于处理少数合并请求。多了麻烦。

1.8.3.3.8

8 合并多个请求

- (1) 添加refspec.行

在本地仓库目录下，找到.git\config文件，打开。

对每一个远程仓库，都有url= 和 fetch= 行，加一行refs\pull的。例：

```
[remote "origin"]
url = https://github.com/libgit2/libgit2.git
fetch = +refs/heads/*:refs/remotes/origin/*
fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

这个设置能使一个fetch命令抓取所有形如refs/pull/*/head的合并请求

- (2) 执行 git fetch命令，不带参数，例如：

```
$ git fetch
# ...
* [new ref]          refs/pull/1/head -> origin/pr/1
* [new ref]          refs/pull/2/head -> origin/pr/2
* [new ref]          refs/pull/4/head -> origin/pr/4
# ...
```

这个命令将合并请求都抓取为本地分支，以后可以检出

(3) 检出合并请求，如：

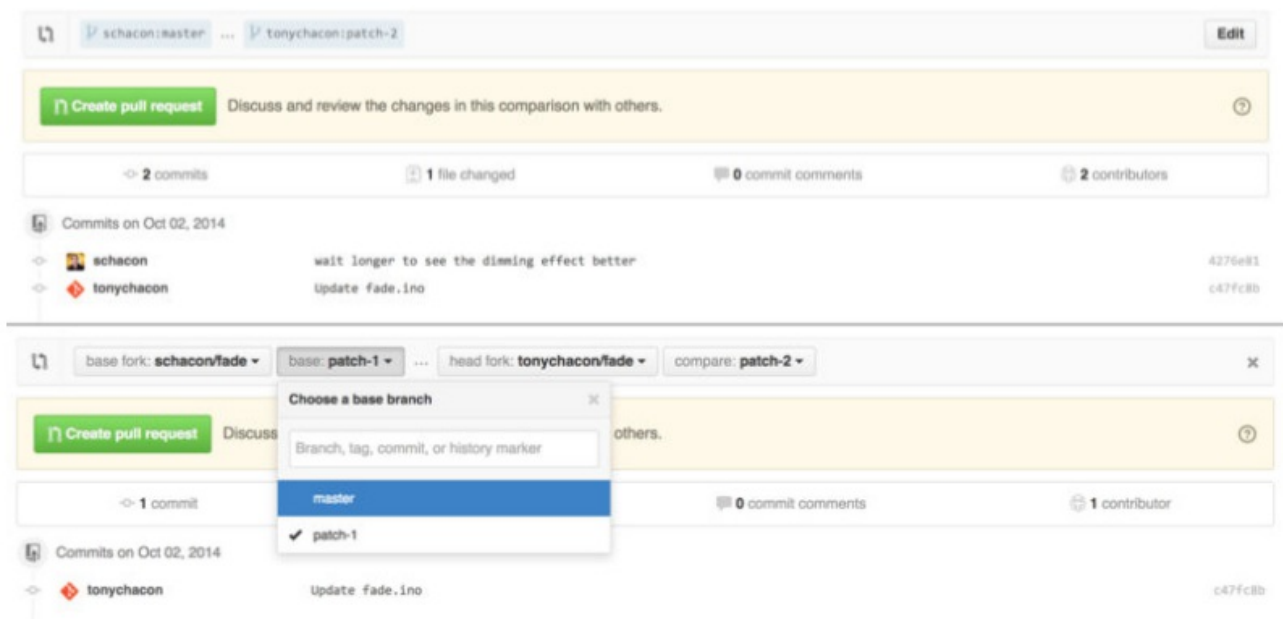
```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

之后可以按需要进行合并。

9 合并请求之上的合并请求

前面都是讲的如何处理别人发过来的合并请求。

这里讲的是在github界面上如何修改要发出的合并请求。如：



图的上部分是要发出请求。但想修改，点击edit，就成了下面的样子。

可以在界面上选择要合并到的分支（你要推送的目标）和用（自己的）哪个分支去合并（源）。

目标在左边，源在右边。

1.8.3.4

4 Mentions and Notifications

1.8.3.4.1

1 提醒用户

在issues的评论中用@,会出现一个用户列表，显示合作者和贡献者的名字和用户名。

这样可以对他们发问。也可以提醒不在列表中的用户，这样需要自己写用户名。

被提醒用户会收到评论。

1.8.3.4.2

2 通知中心

有网站通知和邮件通知两种

可以在账户中设置。

1.8.3.5

5 Special Files

1.8.3.5.1

1 readme

可以是readme,readme.md,readme.asciidoc

【readme是什么格式？txt吗？】

【asciidoc是什么格式？建立了一个，但没有看出有什么特别的地方】

1.8.3.5.2

2 CONTRIBUTING

有任何一个扩展名CONTRIBUTING的文件存在，当别人发出合并请求时，作者就会得到提醒：

Please review the [guidelines for contributing](#) to this repository.

Title

Write

Preview

Leave a comment

Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard.

🔗

We can't automatically merge these branches.

Don't worry, you can still create the pull request.

Create pull request

📄

14

🔗

0

📁

0

📖

Wiki

📶

Pulse

📊

Graphs

⚙️

Settings

🔍

Search branches...

作者可以在里面写要求

1.8.3.6

6 Project Administration

1.8.3.6.1

1 改变默认分支

按 书上，是在仓库的设置option中，如下图所示：

Options

Collaborators

Webhooks & Services

Deploy keys

Settings

Repository name

fade

Rename

Default branch

✓ master

development

issue-53

经实际查看，应该是在code的branch中

点击code,再点击branch,则得到下图：

<> Code

🔔 Issues 14

🔗 Pull requests 0

📁 Projects 0

📖 Wiki

📶 Pulse

📊 Graphs

⚙️ Settings

Overview

Yours

Active

Stale

All branches

🔍

Search branches...

Default branch

master

Updated 10 days ago by huangblue

Default

Change default branch

1.8.3.6.2

2 移交所有权

在仓库设置的options中，在Danger Zone（红色区）中。

1.8.4

4 Managing an organization

组织是多人多项目。

提供了管理成员的工具。

1.8.4.1

1 Organization Basics

1.8.4.2

2 Teams

各成员可以拥有不同的权限：只读，只写，读写和管理

可以邀请和移除成员。

1.8.4.3

3 Audit Log

可以查看各成员在世界各地对项目作的工作。

1.8.5

5 Scripting GitHub

1.8.5.1

1 Hooks

编写钩子的教程：

<https://developer.github.com/webhooks/>

1.8.5.2

2 Services

书中举了email的例子。

1.8.5.3

3The GitHub API

1.8.5.4

4 Basic Usage

1.8.5.5

5 Commenting on an Issue

1.8.5.6

6 Changing the Status of a Pull Request

1.8.5.7

7 Octokit

1.9

Chapter 7:Git Tools

1.10

Chapter 8:Customizing Git

1.11

Chapter 9:Git and Other Systems

1.12

Chapter 10:Git Internals

1.13

Appendix :Git in Other Environments

1.14

Appendix B:Embedding Git in your Applications

1.15

Appendix C:Git Commands
