

目录

前言	1.1
第1章 Git简介	1.2
1.1 Git的诞生	1.2.1
1.2 集中式vs分布式	1.2.2
第2章 安装Git	1.3
第3章 创建版本库	1.4
第4章 时光机穿梭	1.5
4.1 版本回退	1.5.1
4.2 工作区和暂存区	1.5.2
4.3 管理修改	1.5.3
4.4 撤销修改	1.5.4
4.5 删除文件	1.5.5
4.6 让GitBisect帮助你	1.5.6
第5章 远程仓库	1.6
5.1 添加远程仓库	1.6.1
5.2 从远程库克隆	1.6.2
5.3 Git push与pull的默认行为	1.6.3
第6章 分支管理	1.7
6.1 创建与合并分支	1.7.1
6.2 解决冲突	1.7.2
6.3 分支管理策略	1.7.3
6.4 Bug分支	1.7.4
6.5 Feature分支	1.7.5
6.6 多人协作	1.7.6
第7章 标签管理	1.8
7.1 创建标签	1.8.1
7.2 操作标签	1.8.2
第8章 使用GitHub	1.9
第9章 自定义Git	1.10
9.1 忽略特殊文件	1.10.1
9.2 配置别名	1.10.2
9.3 搭建Git服务器	1.10.3
第10章 期末总结	1.11
第11章 从0开始学习GitHub	1.12
01.初识 GitHub	1.12.1

02.加入 GitHub	1.12.2
03.Git 速成	1.12.3
04.向GitHub 提交代码	1.12.4
05.Git 进阶	1.12.5
06.团队合作利器 Branch	1.12.6
07.GitHub 常见的几种操作	1.12.7
08.如何发现优秀的开源项目	1.12.8
使用GIT FLOW管理开发流程	1.12.9
2016 Top 10 Android Library	1.12.10
GIT常用命令备忘	1.12.11
2016 年最受欢迎的编程语言是什么？	1.12.12
GitHub秘籍	1.12.13



史上最浅显易懂的Git教程！

为什么要编写这个教程？因为我在学习Git的过程中，买过书，也在网上Google了一堆Git相关的文章和教程，但令人失望的是，这些教程不是难得令人发指，就是简单得一笔带过，或者，只支离破碎地介绍Git的某几个命令，还有直接从Git手册粘贴帮助文档的，总之，初学者很难找到一个由浅入深，学完后能立刻上手的Git教程。

既然号称史上最浅显易懂的Git教程，那这个教程有什么让你怦然心动的特点呢？

首先，本教程绝对面向初学者，没有接触过版本控制概念的读者也可以轻松入门，不必担心起步难度；

其次，本教程实用性超强，边学边练，一点也不觉得枯燥。而且，你所学的Git命令是“充分且必要”的，掌握了这些东西，你就可以通过Git轻松地完成你的工作。

文字+图片还看不明白？有视频！！！

本教程只会让你成为Git用户，不会让你成为Git专家。很多Git命令只有那些专家才明白（事实上我也不明白，因为我不是Git专家），但我保证这些命令可能你一辈子都不会用到。既然Git是一个工具，就没必要把时间浪费在那些“高级”但几乎永远不会用到的命令上。一旦你真的非用不可了，到时候再自行Google或者请教专家也未迟。

如果你是一个开发人员，想用上这个世界上目前最先进的分布式版本控制系统，那么，赶快开始学习吧！

关于作者

Git教程原文 - 廖雪峰的官方网站

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Visual Basic/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub

更多Git教程

- [Pro Git 中文版](#)
- [Pro Git 第二版 简体中文 1](#)
- [Pro Git 第二版 简体中文 2](#)
- [从 0 开始学习GitHub](#)
- [github-cheat-sheet GitHub秘籍](#)

Github托管主页

<https://github.com/JackChen1999/Git-Course>

请读者点击Star进行关注并支持！

GitBook在线阅读

在线阅读，PDF、ePub、Mobi电子书下载

<https://alleniverson.gitbooks.io/gitcourse/content/>

目录

- 序言
- 第1章 Git简介
 - 1.1 Git的诞生
 - 1.2 集中式vs分布式
- 第2章 创建版本库
- 第3章 安装Git
- 第4章 时光机穿梭
 - 4.1 版本回退
 - 4.2 工作区和暂存区
 - 4.3 管理修改
 - 4.4 撤销修改
 - 4.5 删除文件
 - 4.6 让GitBisect帮助你
- 第5章 远程仓库
 - 5.1 添加远程仓库
 - 5.2 从远端库克隆
 - 5.3 Git push与pull的默认行为
- 第6章 分支管理
 - 6.1 创建与合并分支
 - 6.2 解决冲突
 - 6.3 分支管理策略
 - 6.4 Bug分支
 - 6.5 Feature分支
 - 6.6 多人协作
- 第7章 标签管理
 - 7.1 创建标签
 - 7.2 操作标签
- 第8章 使用GitHub
- 第9章 自定义Git
 - 9.1 忽略特殊文件
 - 9.2 配置别名
 - 9.3 搭建Git服务器
- 第10章 期末总结

- 第11章 从0开始学习GitHub
 - 01.初识 GitHub
 - 02.加入 GitHub
 - 03.Git 速成
 - 04.向GitHub%20提交代码
 - 05.Git 进阶
 - 06.团队合作利器 Branch
 - 07.GitHub 常见的几种操作
 - 08.如何发现优秀的开源项目
 - 使用GIT FLOW管理开发流程
 - 2016 Top 10 Android Library
 - GIT常用命令备忘
 - 2016 年最受欢迎的编程语言是什么？
 - GitHub秘籍

关注我

- Email : 619888095@qq.com
- CSDN博客 : [Allen Iverson](#)
- 新浪微博 : [AndroidDeveloper](#)
- GitHub : [JackChan1999](#)
- GitBook : [alleniverson](#)
- 个人博客 : [JackChan](#)

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！



第1章 Git简介

Git是什么？

Git是目前世界上最先进的分布式版本控制系统（没有之一）。

Git有什么特点？简单来说就是：高端大气上档次！

那什么是版本控制系统？

如果你用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：

版本	用户	说明	日期
1	张三	删除了软件服务条款5	7/12 10:38
2	张三	增加了License人数限制	7/12 18:09
3	李四	财务部门调整了合同金额	7/13 9:51
4	张三	延长了免费升级周期	7/14 15:17

这样，你就结束了手动管理多个“版本”的史前时代，进入到版本控制的20世纪。

1.1 Git的诞生

很多人都知道，Linus在1991年创建了开源的Linux，从此，Linux系统不断发展，已经成为最大的服务器系统软件了。

Linus虽然创建了Linux，但Linux的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为Linux编写代码，那Linux的代码是如何管理的呢？

事实是，在2002年以前，世界各地的志愿者把源代码文件通过diff的方式发给Linus，然后由Linus本人通过手工方式合并代码！

你也许会想，为什么Linus不把Linux代码放到版本控制系统里呢？不是有CVS、SVN这些免费的版本控制系统吗？因为Linus坚定地反对CVS和SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比CVS、SVN好用，但那是付费的，和Linux的开源精神不符。

不过，到了2002年，Linux系统已经发展了十年了，代码库之大让Linus很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是Linus选择了一个商业的版本控制系统BitKeeper，BitKeeper的东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。

安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

Linus可以向BitMover公司道个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：

Linus花了两周时间自己用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了！牛是怎么定义的呢？大家可以体会一下。

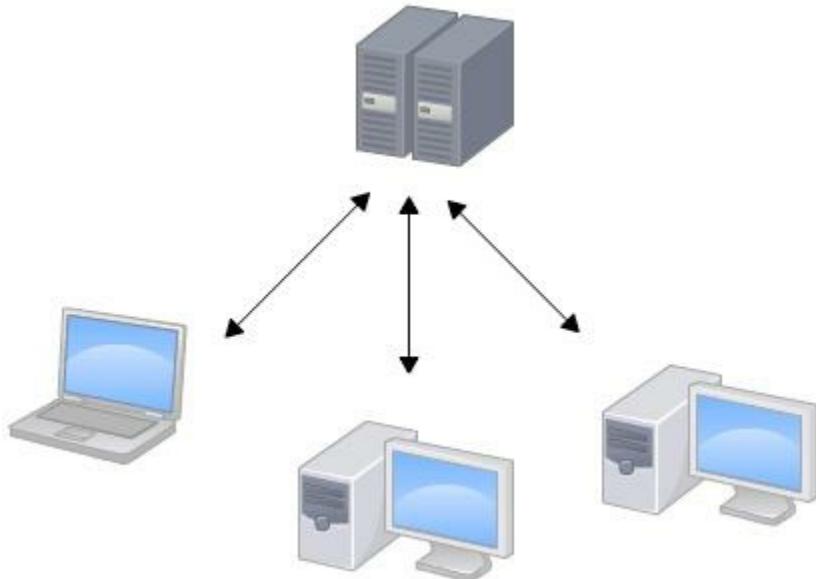
Git迅速成为最流行的分布式版本控制系统，尤其是2008年，GitHub网站上线了，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

历史就是这么偶然，如果不是当年BitMover公司威胁Linux社区，可能现在我们就没有免费而超级好用的Git了。

1.2 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。

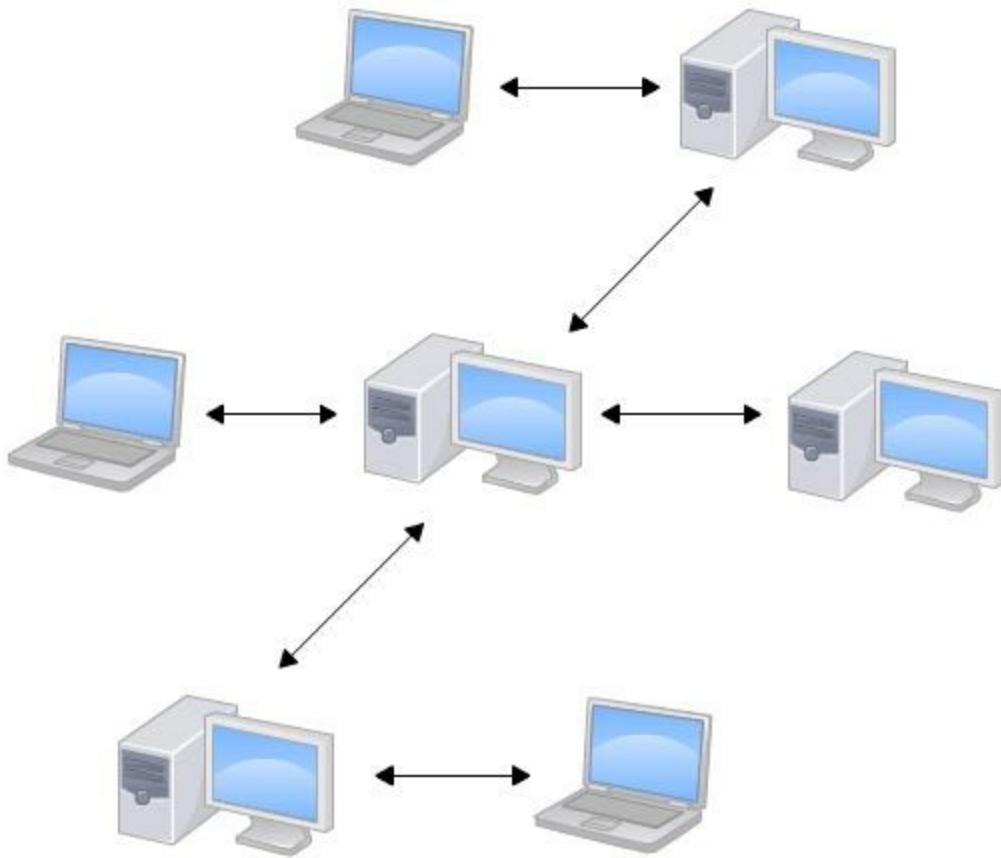


集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

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！

第2章 安装Git

最早Git是在Linux上开发的，很长一段时间内，Git也只能在Linux和Unix系统上跑。不过，慢慢地有人把它移植到了Windows上。现在，Git可以在Linux、Unix、Mac和Windows这几大平台上正常运行了。

要使用Git，第一步当然是安装Git了。根据你当前使用的平台来阅读下面的文字：

在Linux上安装Git

首先，你可以试着输入git，看看系统有没有安装Git：

```
$ git  
The program 'git' is currently not installed. You can install it by typing:  
sudo apt-get install git
```

像上面的命令，有很多Linux会友好地告诉你Git没有安装，还会告诉你如何安装Git。

如果你碰巧用Debian或Ubuntu Linux，通过一条sudo apt-get install git就可以直接完成Git的安装，非常简单。

video：<http://michaelliao.gitcafe.io/video/git-apt-install.mp4>

老一点的Debian或Ubuntu Linux，要把命令改为sudo apt-get install git-core，因为以前有个软件也叫GIT（GNU Interactive Tools），结果Git就只能叫git-core了。由于Git名气实在太大，后来就把GNU Interactive Tools改成gnuit，git-core正式改为git。

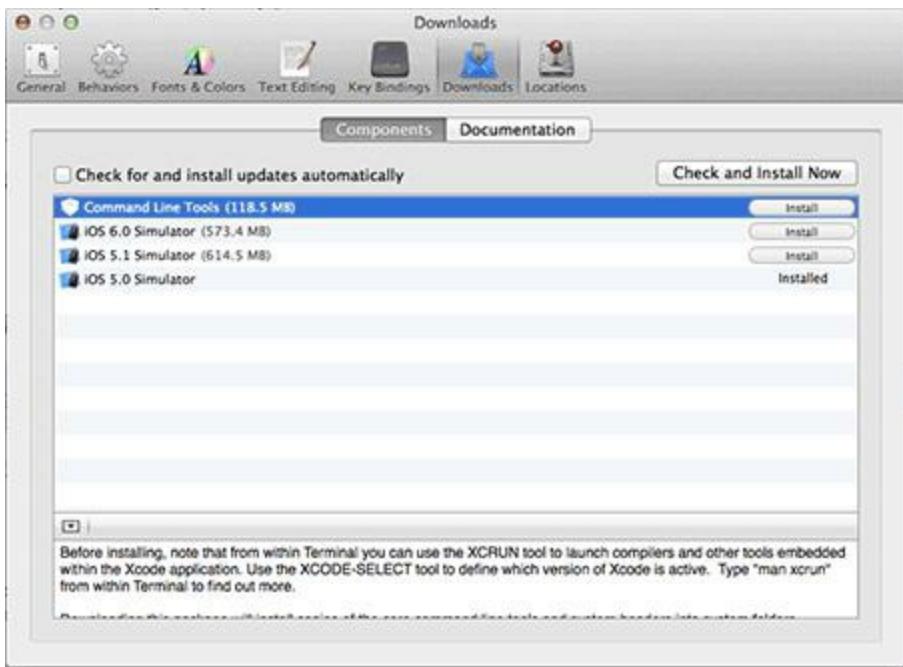
如果是其他Linux版本，可以直接通过源码安装。先从Git官网下载源码，然后解压，依次输入：./config，make，sudo make install这几个命令安装就好了。

在Mac OS X上安装Git

如果你正在使用Mac做开发，有两种安装Git的方法。

一是安装homebrew，然后通过homebrew安装Git，具体方法请参考homebrew的文档：<http://brew.sh/>。

第二种方法更简单，也是推荐的方法，就是直接从AppStore安装Xcode，Xcode集成了Git，不过默认没有安装，你需要运行Xcode，选择菜单“Xcode”->“Preferences”，在弹出窗口中找到“Downloads”，选择“Command Line Tools”，点“Install”就可以完成安装了。



Xcode是Apple官方IDE，功能非常强大，是开发Mac和iOS App的必选装备，而且是免费的！

在Windows上安装Git

实话实说，Windows是最烂的开发平台，如果不是开发Windows游戏或者在IE里调试页面，一般不推荐用Windows。不过，既然已经上了微软的贼船，也是有办法安装Git的。

Windows下要使用很多Linux/Unix的工具时，需要Cygwin这样的模拟环境，Git也一样。Cygwin的安装和配置都比较复杂，就不建议你折腾了。不过，有高人已经把模拟环境和Git都打包好了，名叫msysgit，只需要下载一个单独的exe安装程序，其他什么也不用装，绝对好用。

msysgit是Windows版的Git，从<https://git-for-windows.github.io>下载（网速慢的同学请移步国内镜像），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！



安装完成后，还需要最后一步设置，在命令行输入：

```
$ git config --global user.name "Your Name"  
$ git config --global user.email "email@example.com"
```

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和Email地址。你也许会担心，如果有人故意冒充别人怎么办？这个不必担心，首先我们相信大家都是善良无知的群众，其次，真的有冒充的也是有办法可查的。

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。

第3章 创建版本库

什么是版本库呢？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”。

所以，创建一个版本库非常简单，首先，选择一个合适的地方，创建一个空目录：

```
$ mkdir learngit  
$ cd learngit  
$ pwd  
/Users/michael/learngit
```

pwd命令用于显示当前目录。在我的Mac上，这个仓库位于/Users/michael/learngit。

如果你使用Windows系统，为了避免遇到各种莫名其妙的问题，请确保目录名（包括父目录）不包含中文。

第二步，通过git init命令把这个目录变成Git可以管理的仓库：

```
$ git init  
Initialized empty Git repository in /Users/michael/learngit/.git/
```

瞬间Git就把仓库建好了，而且告诉你是一个空的仓库（empty Git repository），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。

video：<http://michaelliao.gitcafe.io/video/git-init.mp4>

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。不过，不建议你使用自己正在开发的公司项目来学习Git，否则造成的一切后果概不负责。

把文件添加到版本库

首先这里再明确一下，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变成了120KB，但到底改了啥，版本控制系统不知道，也没法知道。

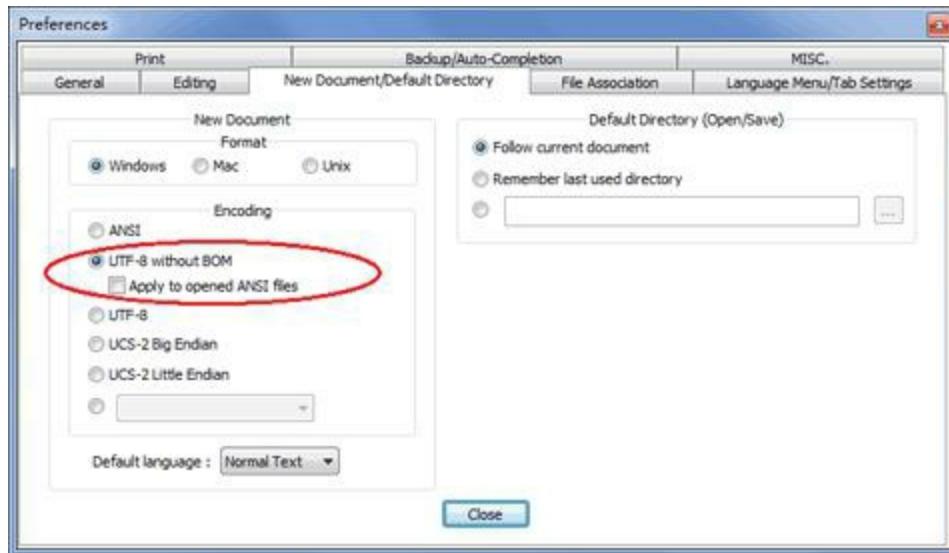
不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

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的童鞋要特别注意：

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0xefbbbb（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：



言归正传，现在我们编写一个readme.txt文件，内容如下：

```
Git is a version control system.  
Git is free software.
```

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

和把大象放到冰箱需要3步相比，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：

```
$ git add readme.txt
```

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

第二步，用命令git commit告诉Git，把文件提交到仓库：

```
$ git commit -m "wrote a readme file"  
[master (root-commit) cb926e7] wrote a readme file  
1 file changed, 2 insertions(+)  
create mode 100644 readme.txt
```

video：<http://github.liaoxuefeng.com/sinaweibopy/video/add-and-commit.mp4>

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。

嫌麻烦不想输入-m "xxx"行不行？确实有办法可以这么干，但是强烈不建议你这么干，因为输入说明对自己对别人阅读都很重要。实在不想输入说明的童鞋请自行Google，我不告诉你这个参数。

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

为什么Git添加文件需要add, commit一共两步呢？因为commit可以一次提交很多文件，所以你可以多次add不同的文件，比如：

```
$ git add file1.txt  
$ git add file2.txt file3.txt  
$ git commit -m "add 3 files."
```

小结

现在总结一下今天学的两点内容：

初始化一个Git仓库，使用git init命令。

添加文件到Git仓库，分两步：

第一步，使用命令git add < file>，注意，可反复多次使用，添加多个文件；

第二步，使用命令git commit，完成。

第4章 时光穿梭机

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

```
Git is a distributed version control system.  
Git is free software.
```

现在，运行git status命令看看结果：

```
$ git status  
# On branch master  
# Changes not staged for commit:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working directory)  
#  
#       modified:   readme.txt  
#  
no changes added to commit (use "git add" and/or "git commit -a")
```

git status命令可以让我们时刻掌握仓库当前的状态，上面的命令告诉我们，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。比如你休假两周从国外回来，第一天上班时，已经记不清上次怎么修改的readme.txt，所以，需要用git diff这个命令看看：

```
$ git diff readme.txt  
diff --git a/readme.txt b/readme.txt  
index 46d49bf..9247db6 100644  
--- a/readme.txt  
+++ b/readme.txt  
@@ -1,2 +1,2 @@  
-Git is a version control system.  
+Git is a distributed version control system.  
 Git is free software.
```

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，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：

```
$ 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：

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

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

```
$ git commit -m "add distributed"
[master ea34578] add distributed
 1 file changed, 1 insertion(+), 1 deletion(-)
```

提交后，我们再用git status命令看看仓库的当前状态：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

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

video：<http://michaelliao.gitcafe.io/video/git-diff-status.mp4>

小结

要随时掌握工作区的状态，使用git status命令。

如果git status告诉你有文件被修改过，用git diff可以查看修改内容。

4.1 版本回退

现在，你已经学会了修改文件，然后把修改提交到Git版本库，现在，再练习一次，修改readme.txt文件如下：

```
Git is a distributed version control system.  
Git is free software distributed under the GPL.
```

然后尝试提交：

```
$ git add readme.txt  
$ git commit -m "append GPL"  
[master 3628164] append GPL  
1 file changed, 1 insertion(+), 1 deletion(-)
```

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，我们回顾一下readme.txt文件一共有几个版本被提交到Git仓库里了：

版本1：wrote a readme file

```
Git is a version control system.  
Git is free software.
```

版本2：add distributed

```
Git is a distributed version control system.  
Git is free software.
```

版本3：append GPL

```
Git is a distributed version control system.  
Git is free software distributed under the GPL.
```

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

```
$ git log  
commit 3628164fb26d48395383f8f31179f24e0882e1e0  
Author: Michael Liao <askxuefeng@gmail.com>  
Date: Tue Aug 20 15:11:49 2013 +0800  
  
append GPL  
  
commit ea34578d5496d7dd233c827ed32a8cd576c5ee85  
Author: Michael Liao <askxuefeng@gmail.com>  
Date: Tue Aug 20 14:53:12 2013 +0800  
  
add distributed
```

```

commit cb926e7ea50ad11b8f9e909c05226233bf755030
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 19 17:51:55 2013 +0800

wrote a readme file

```

git log命令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：

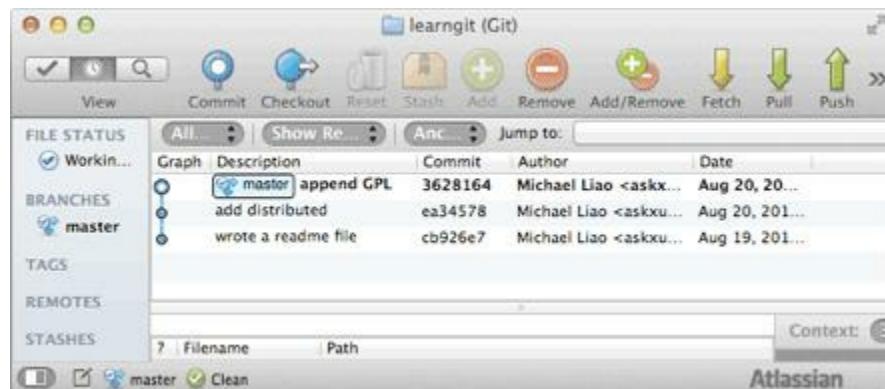
```

$ git log --pretty=oneline
3628164fb26d48395383f8f31179f24e0882e1e0 append GPL
ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed
cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file

```

需要友情提示的是，你看到的一大串类似3628164...882e1e0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，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1, 2, 3……作为版本号，那肯定就冲突了。

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：



好了，现在我们启动时光穿梭机，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3628164...882e1e0（注意我的提交ID和你的肯定不一样），上一个版本就是HEAD^，上上一个版本就是HEAD^^，当然往上100个版本写100个^比较容易数不过来，所以写成HEAD~100。

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

```

$ git reset --hard HEAD^
HEAD is now at ea34578 add distributed

```

--hard参数有啥意义？这个后面再讲，现在你先放心使用。

看看readme.txt的内容是不是版本add distributed：

```

$ cat readme.txt
Git is a distributed version control system.

```

```
Git is free software.
```

果然。

还可以继续回退到上一个版本wrote a readme file，不过且慢，然我们用git log再看看现在版本库的状态：

```
$ git log
commit ea34578d5496d7dd233c827ed32a8cd576c5ee85
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 14:53:12 2013 +0800

    add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 19 17:51:55 2013 +0800

    wrote a readme file
```

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

办法其实还是有的，只要上面的命令行窗口还没有被关掉，你就可以顺着往上找啊找啊，找到那个append GPL的commit id是3628164...，于是就可以指定回到未来的某个版本：

```
$ git reset --hard 3628164
HEAD is now at 3628164 append GPL
```

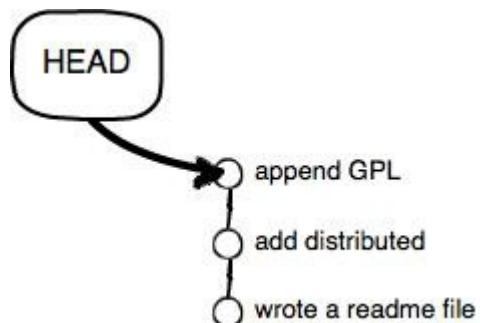
版本号没必要写全，前几位就可以了，Git会自动去找。当然也不能只写前一两位，因为Git可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看readme.txt的内容：

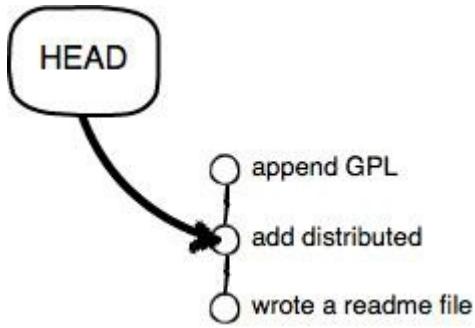
```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
```

果然，我胡汉三又回来了。

Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD从指向append GPL：



改为指向add distributed：



然后顺便把工作区的文件更新了。所以你让HEAD指向哪个版本号，你就把当前版本定位在哪。

video : <http://michaelliao.gitcafe.io/video/git-reset.mp4>

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

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

```
$ git reflog
ea34578 HEAD@{0}: reset: moving to HEAD^
3628164 HEAD@{1}: commit: append GPL
ea34578 HEAD@{2}: commit: add distributed
cb926e7 HEAD@{3}: commit (initial): wrote a readme file
```

终于舒了口气，第二行显示append GPL的commit id是3628164，现在，你又可以乘坐时光机回到未来了。

video : <http://michaelliao.gitcafe.io/video/git-reflog-reset.mp4>

小结

现在总结一下：

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。

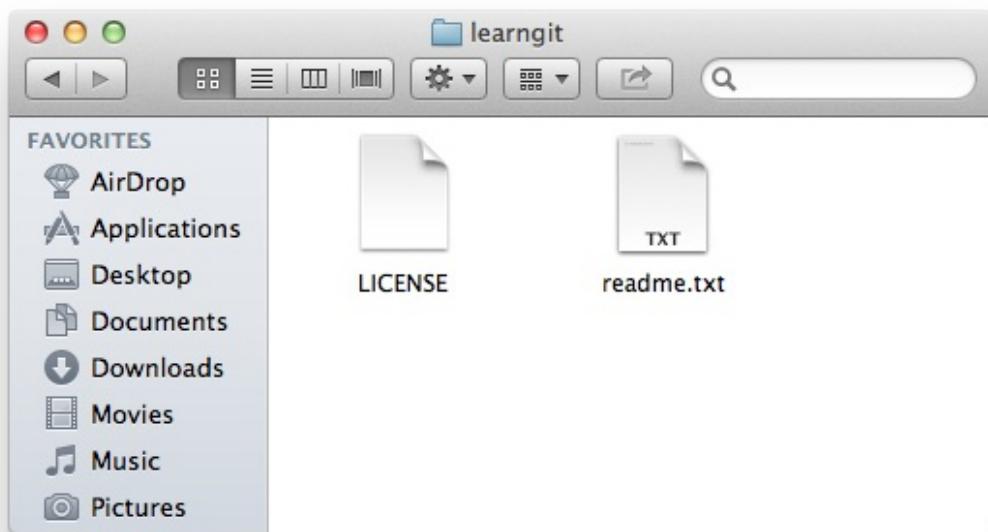
4.2 工作区和暂存区

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

先来看名词解释。

工作区 (Working Directory)

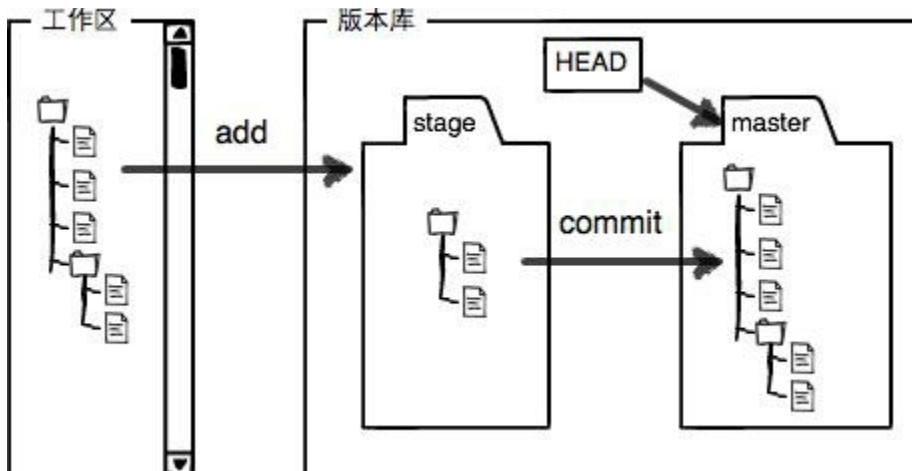
就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



版本库 (Repository)

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



分支和HEAD的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

第二步是用git commit提交更改，实际上就是把暂存区的所有内容提交到当前分支。

因为我们创建Git版本库时，Git自动为我们创建了唯一一个master分支，所以，现在，git commit就是往master分支上提交更改。

你可以简单理解为，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。

俗话说，实践出真知。现在，我们再练习一遍，先对readme.txt做个修改，比如加上一行内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
```

然后，在工作区新增一个LICENSE文本文件（内容随便写）。

先用git status查看一下状态：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       LICENSE
no changes added to commit (use "git add" and/or "git commit -a")
```

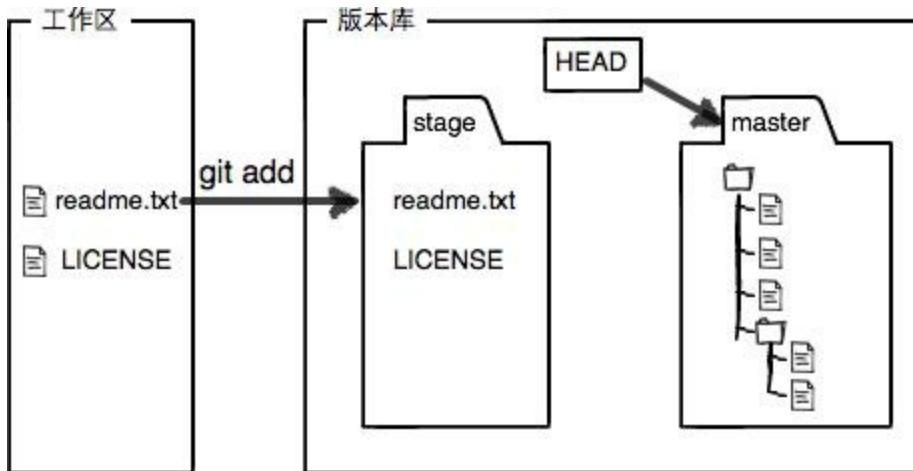
Git非常清楚地告诉我们，readme.txt被修改了，而LICENSE还从来没有被添加过，所以它的状态是Untracked。

现在，使用两次命令git add，把readme.txt和LICENSE都添加后，用git status再查看一下：

```
$ git status
# On branch master
# Changes to be committed:
```

```
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   LICENSE
#       modified:  readme.txt
```

现在，暂存区的状态就变成这样了：



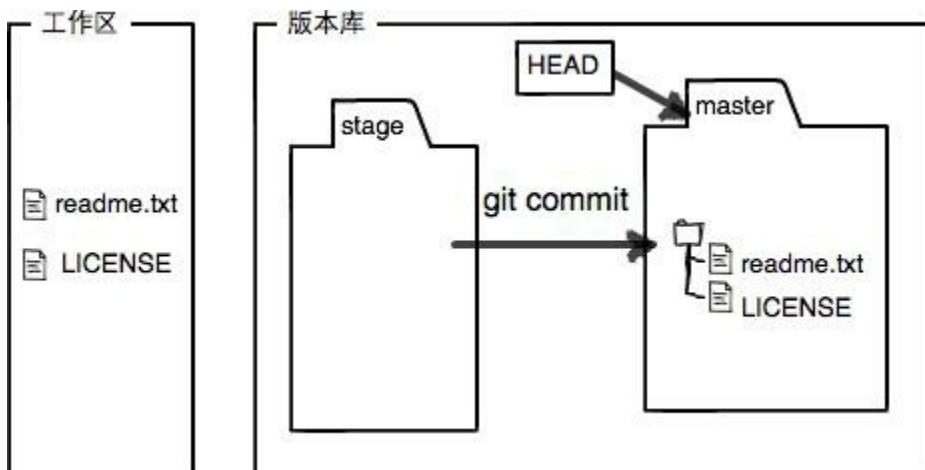
所以，`git add`命令实际上就是把要提交的所有修改放到暂存区（Stage），然后，执行`git commit`就可以一次性把暂存区的所有修改提交到分支。

```
$ git commit -m "understand how stage works"
[master 27c9860] understand how stage works
 2 files changed, 675 insertions(+)
 create mode 100644 LICENSE
```

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

现在版本库变成了这样，暂存区就没有任何内容了：



小结

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

没弄明白暂存区是怎么回事的童鞋，请向上滚动页面，再看一次。

4.3 管理修改

现在，假定你已经完全掌握了暂存区的概念。下面，我们要讨论的就是，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。

你会问，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？我们还是做实验。第一步，对readme.txt做一个修改，比如加一行内容：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes.
```

然后，添加：

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

然后，再修改readme.txt：

```
$ 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：

```
$ git commit -m "git tracks changes"
[master d4f25b6] git tracks changes
 1 file changed, 1 insertion(+)
```

提交后，再看看状态：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

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

你看，我们前面讲了，Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，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：

```
$ git diff HEAD -- readme.txt
diff --git a/readme.txt b/readme.txt
index 76d770f..a9c5755 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,4 +1,4 @@
 Git is a distributed version control system.
 Git is free software distributed under the GPL.
 Git has a mutable index called stage.
-Git tracks changes.
+Git tracks changes of files.
```

可见，第二次修改确实没有被提交。

video：<http://michaelliao.gitcafe.io/video/git-add-changes.mp4>

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

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

好，现在，把第二次修改提交了，然后开始小结。

小结

现在，你又理解了Git是如何跟踪修改的，每次修改，如果不add到暂存区，那就不会加入到commit中。

4.4 撤销修改

自然，你是不会犯错的。不过现在是凌晨两点，你正在赶一份工作报告，你在readme.txt中添加了一行：

```
$ 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.
```

在你准备提交前，一杯咖啡起了作用，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：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

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

```
$ git checkout -- readme.txt
```

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。

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

现在，看看readme.txt的文件内容：

```
$ 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.
```

文件内容果然复原了。

git checkout -- file命令中的--很重要，没有--，就变成了“切换到另一个分支”的命令，我们在后面的分支管理中会再次遇到git checkout命令。

video：<http://michaelliao.gitcafe.io/video/discard-changes-of-staged.mp4>

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：

```
$ cat readme.txt
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
My stupid boss still prefers SVN.

$ git add readme.txt
```

庆幸的是，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：

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

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

```
$ git reset HEAD readme.txt
Unstaged changes after reset:
M       readme.txt
```

git reset命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用HEAD时，表示最新的版本。

再用git status查看一下，现在暂存区是干净的，工作区有修改：

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

还记得如何丢弃工作区的修改吗？

```
$ git checkout -- readme.txt

$ git status
# On branch master
nothing to commit (working directory clean)
```

整个世界终于清静了！

video：<http://michaelliao.gitcafe.io/video/discard-changes-of-staged.mp4>

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……

小结

又到了小结时间。

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，不过前提是沒有推送到远程序库。

4.5 删除文件

在Git中，删除也是一个修改操作，我们实战一下，先添加一个新文件test.txt到Git并且提交：

```
$ git add test.txt
$ git commit -m "add test.txt"
[master 94cdc44] add test.txt
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

```
$ rm test.txt
```

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

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    test.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

```
$ git rm test.txt
rm 'test.txt'
$ git commit -m "remove test.txt"
[master d17efd8] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

现在，文件就从版本库中被删除了。

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：

```
$ git checkout -- test.txt
```

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

小结

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。

4.6 让GitBisect帮助你

Git 提供来很多的工具来帮助我们改进工作流程。 **bisect** 命令就是其中之一，虽然由于使用得不多而不广为人知，但是当你想知道一个本来好的分支从什么时候开始变坏时，它就能派上用场了。到底是哪一次提交把事情搞砸了呢，让 bisect 来告诉你吧。

Bisect 基于[二分查找算法](#)。给定一个有序的元素序列，它会返回你要找的元素的序号（或者告诉你该元素是否在序列中）。它基于如下的不变式：当你对比完一个元素，你就能根据比较的结果抛弃掉它之前或者之后的所有元素（从而大大缩小下次查找的范围）。

如果你有去过图书馆，那你可能注意到了每个专区内的书籍都是按作者姓名来排序的。比方说你要找的一本书的作者的姓为 Martin，而你刚刚看到某个书架上的最后一本书的作者的姓是 F 开头的，那么你就能确定你要找的书一定放在后面的某个书架上。依此方法继续下次，你就能快速的缩小搜索范围直到找到你要的书为止。

二分查找法亦是如此。如果你想在有 n 个元素的序列（有序的）中查找元素 x ，你挑出第 $n/2$ 个元素并将其与元素 x 比较。如果 x 大，那么就对从 $n/2+1$ 到 n 的子序列重复上述步骤，反之，就对从 1 到 $n/2-1$ 的子序列重复上述步骤，这样一直递归下去。这里就有一个[关于此算法的有趣的演示](#)。

二分查找算法所需的比较次数通常都比你傻傻的去拿序列中的每个元素与 x 比较所需的比较次数少得多。事实证明在有序序列中查找元素，二分查找更快更有用。

Bisect

Bisect 就是利用二分查找发来查找在你的某一分支中到底是哪一次提交引入了特定的变更。首先，也是最起码的，你得有办法检测出一个给定的提交点是不是有 bug。通常你可以写个测试代码，如果这也不可能的话，那至少你得有一套步骤来使 bug 再现出来。

有了这套检测方法，你就可以开始用 bisect 查找你的提交历史来以最快的速度发现引入 bug 的时间点了。等等，你还得准备好两个提交点：一个是你确定 bug 还没有被引入的提交点，另一个则是确定 bug 已经引入了的提交点（这样就缩小了初始的查找范围）。在 bisect 命令上，你可以用哈希值（hash）或者标签（tag）来引用这两个提交点。bisect 查找时，查找范围里的提交点要么被标记为好的（通过测试，没有 bug 的），要么被标记为坏的（通过测试，有 bug 的）。

上图演示了 bisect 的执行步骤（绿点是好的提交点，红点的就是坏的）：bisect 被要求在提交点 1 到 8 这个范围内查找首次引入 bug 的提交点（看图一目了然提交点 6），这里提交点 1 和 8 就是上段中提及的，我们需要首先准备好给 bisect 命令的两个（一好一坏）提交点。bisect 首先用调用者提供的检测方法来测试 1 到 8 中间的提交点 4，如果它是好的（图上的情况），那么 bisect 就缩小范围为它右边的区域，重复上述步骤，反之，则选择其左边的区域为新的搜索范围。如此递归下去，在上图的例子中，只需要 3 步就确定了罪魁祸首是提交点 6.

我这里创建了一个[git 仓库](#)，里面共有 1024 个提交记录，每个提交都往一个文本文件后面添加一个递增的数字，从 1 到 1024，一行一个数字。我们的任务是要找出是哪个提交点向文本文件附加了 1013 这个数字（我们假定它就是一个 bug）。

首先，我们定出一个方法来判断文件里面有没这个数字。这就是一个测试了。方法很简单：

```
$ grep 1013 file.txt
```

有了这条命令，我们就可以开始 bisect 了：

```
$ git bisect start
```

对 master 分支（的头部提交点）运行这个测试，没有得到想要的结果（就是没有任何输出），所以我们把它标记为坏的。

```
$ git bisect bad
```

现在让我们指定一个好的提交点：假设第一个提交点（7c0dcfa）是没有 bug 的。我们可以检出（check out）这个提交点并标记它为好的提交点（git bisect good + 该提交点的哈希值）。

```
$ git bisect good 7c0dcfa
Bisecting: 511 revisions left to test after this (roughly 9 steps)
[8950f7db7e7cad0b2dc394ff9b75fc3d38c9d72a] added 512
```

也可以用下面的命令，完成上面的所有步骤：

```
$ git bisect start master 7c0dcfa
```

git bisect start 命令可以接受两个参数，第一个是坏的提交点，第二是好的提交点。

很好，git bisect 自动检出了正中间的提交点，运行我们的检测命令，没有任何的输出（没有 bug），因此我们把该提交点标记为好的。

```
$ grep 1013 file.txt
$ git bisect good
Bisecting: 剩余 255 个修订待测试 (大概还需要 8 步)
[a01ba83f3500b48da97c5f5c33052623aaa4161a] added 768
```

译者注：按照二分查找算法，bisect 根据你标记的结果，决定是检出左半边的中间提交点（如果你标记为 bad）还是右半边的中间提交点（如果你标记为 good）。在 bisect 新检出的提交点上，我们再次运行测试命令。这次依然是个好的提交点。

```
$ grep 1013 file.txt
$ git bisect good
Bisecting: 剩余 127 个修订待测试 (大概还需要 7 步)
[4a4a668bf3363d09af5fd1906bc4272aacdb4495] added 896
```

再次检测，还是好的提交点。

```
$ grep 1013 file.txt
$ git bisect good
Bisecting: 剩余 63 个修订待测试 (大概还需要 6 步)
[9059c5b8b898159e8d1d797bff3b1febd1fd6a1c] added 960
```

让我们看看这些消息：除了告诉你当前新检出的提交点以及新的搜索范围内有多少个待测试的提交点外，它还估计出你最多还需要重复多少次测试命令就能找到你要的提交点。这次又是一个好的提交点。

```
$ grep 1013 file.txt
$ git bisect good
Bisecting: 剩余 31 个修订待测试 (大概还需要 5 步)
[0c844d0b33ef297b742206ebc293f4925705b083] added 992
```

继续，依然是一个好的提交点。

```
$ grep 1013 file.txt  
  
$ git bisect good  
Bisecting: 剩余 15 个修订待测试 (大概还需要 4 步)  
[0ee17eb17bd96b321a01c73eb13a8929a68b1239] added 1008
```

还有 4 步，这次依然 good。

```
$ grep 1013 file.txt  
  
$ git bisect good  
Bisecting: 剩余 7 个修订待测试 (大概还需要 3 步)  
[dfb1e71736dcfffa2a30aec7299f45f757c057e] added 1016
```

这次测试命令终于有了输出，bug 现身了！我们把这个提交点标记为坏的。

```
$ grep 1013 file.txt  
1013  
$ git bisect bad  
Bisecting: 剩余 3 个修订待测试 (大概还需要 2 步)  
[6e6d08c374df5162fed65fed82859b69f86b936e] added 1012
```

终点近在咫尺，测试通过，我们把它标记为好的。

```
$ grep 1013 file.txt  
  
$ git bisect bad  
Bisecting: 剩余 1 个修订待测试 (大概还需要 1 步)  
254efa859d7fc66f1f58a59f0] added 1014
```

坏的提交点。

```
$ grep 1013 file.txt  
1013  
$ git bisect bad  
Bisecting: 剩余 0 个修订待测试 (大概还需要 0 步)  
8e16d98ec7039a7c53855dd9ed6] added 1013
```

最后一步，这次是坏的。

```
$ git bisect bad  
458eab0eb8d808e16d98ec7039a7c53855dd9ed6 is the first bad commit  
commit 458eab0eb8d808e16d98ec7039a7c53855dd9ed6  
Author: Rodrigo Flores <mail@rodrigoflores.org>  
Date: Tue Oct 21 22:31:05 2014 -0200  
  
        added 1013  
  
:100644 100644 7bc3db7f48a43ccf1a8cc7c26146912cc88c1009 b393a2138a96c1530f41f70  
1ab43cca893226976 M file.txt
```

我们终于得到了那个引入 1013 数字的提交点。命令 git bisect log 可以回放整个过程。

```

$ git bisect start
# bad: [740cdf012013dc41a39b41d4b09b57a970bfe38f] added 1024
git bisect bad 740cdf012013dc41a39b41d4b09b57a970bfe38f
# good: [7c0dcfa7514379151e0d83ffbf805850d2093538] added 1
git bisect good 7c0dcfa7514379151e0d83ffbf805850d2093538
# good: [8950f7db7e7cad0b2dc394ff9b75fc3d38c9d72a] added 512
git bisect good 8950f7db7e7cad0b2dc394ff9b75fc3d38c9d72a
# good: [a01ba83f3500b48da97c5f5c33052623aaa4161a] added 768
git bisect good a01ba83f3500b48da97c5f5c33052623aaa4161a
# good: [4a4a668bf3363d09af5fd1906bc4272aacdb4495] added 896
git bisect good 4a4a668bf3363d09af5fd1906bc4272aacdb4495
# good: [9059c5b8b898159e8d1d797bff3b1febd1fd6a1c] added 960
git bisect good 9059c5b8b898159e8d1d797bff3b1febd1fd6a1c
# good: [0c844d0b33ef297b742206ebc293f4925705b083] added 992
git bisect good 0c844d0b33ef297b742206ebc293f4925705b083
# good: [0ee17eb17bd96b321a01c73eb13a8929a68b1239] added 1008
git bisect good 0ee17eb17bd96b321a01c73eb13a8929a68b1239
# bad: [dfb1e71736dcffffa2a30aec7299f45f757c057e] added 1016
git bisect bad dfb1e71736dcffffa2a30aec7299f45f757c057e
# good: [6e6d08c374df5162fed65fed82859b69f86b936e] added 1012
git bisect good 6e6d08c374df5162fed65fed82859b69f86b936e
# bad: [1d23b7045a8accd254efa859d7fc66f1f58a59f0] added 1014
git bisect bad 1d23b7045a8accd254efa859d7fc66f1f58a59f0
# bad: [458eab0eb8d808e16d98ec7039a7c53855dd9ed6] added 1013
git bisect bad 458eab0eb8d808e16d98ec7039a7c53855dd9ed6
# first bad commit: [458eab0eb8d808e16d98ec7039a7c53855dd9ed6] added 1013

```

这个例子里一共有 1024 个提交点，遍历他们我们只用了 10 步。如果提交点数量再多一倍变成 2048 个，根据二分查找算法，我们仅仅需要多加一步就能找到想要的提交点，因为二分查找算法的时间复杂度为 $O(\log n)$ 。

尽管已经如此高效，一遍又一遍的运行测试命令还是很枯燥的。因此，让我们再进一步，将这个过程自动化吧。

自动化

Git bisect 能接受一个脚本文件名作为命令参数，通过它的返回代码判断当前提交点的好坏。如果返回 0，就是好的提交点，返回其他值就是坏的提交点。凑巧的是，grep 命令如果找到了匹配行就会返回 0，反之返回 1。你可以使用 echo \$? 命令来检查测试命令的返回值。

grep 的返回值正好和我们的测试标准相反，所以我们需要写一个脚本来反转它的值（我不太习惯写 shell 脚本，如果大家有更好的方法，感谢告知）。

```

#!/bin/sh

if [[ `grep 1013 file.txt` ]]; then
    exit 1
else
    exit 0
fi

```

保存上述代码为 test.sh，并赋予它执行权限。

接着在给 bisect 命令指定了初始的好/坏提交点之后，你只需要运行：

```

git bisect run ./test.sh
running ./ola.sh
Bisecting: 255 revisions left to test after this (roughly 8 steps)
[a01ba83f3500b48da97c5f5c33052623aaa4161a] added 768
running ./ola.sh

```

```
Bisecting: 127 revisions left to test after this (roughly 7 steps)
[4a4a668bf3363d09af5fd1906bc4272aacdb4495] added 896
running ./ola.sh
Bisecting: 63 revisions left to test after this (roughly 6 steps)
[9059c5b8b898159e8d1d797bff3b1febd1fd6a1c] added 960
running ./ola.sh
Bisecting: 31 revisions left to test after this (roughly 5 steps)
[0c844d0b33ef297b742206ebc293f4925705b083] added 992
running ./ola.sh
Bisecting: 15 revisions left to test after this (roughly 4 steps)
[0ee17eb17bd96b321a01c73eb13a8929a68b1239] added 1008
running ./ola.sh
Bisecting: 7 revisions left to test after this (roughly 3 steps)
[dfb1e71736dcffffa2a30aec7299f45f757c057e] added 1016
running ./ola.sh
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[6e6d08c374df5162fed65fed82859b69f86b936e] added 1012
running ./ola.sh
Bisecting: 1 revision left to test after this (roughly 1 step)
[1d23b7045a8accd254efa859d7Fc66f1f58a59f0] added 1014
running ./ola.sh
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[458eab0eb8d808e16d98ec7039a7c53855dd9ed6] added 1013
running ./ola.sh
458eab0eb8d808e16d98ec7039a7c53855dd9ed6 is the first bad commit
commit 458eab0eb8d808e16d98ec7039a7c53855dd9ed6
Author: Rodrigo Flores <mail@rodrigoflores.org>
Date: Tue Oct 21 22:31:05 2014 -0200

    added 1013

:100644 100644 7bc3db7f48a43ccf1a8cc7c26146912cc88c1009 b393a2138a96c15
30f41f701ab43cca893226976 M file.txt
bisect run success
```

不费吹灰之力，你就得到了出问题的提交点。

结论

你很可能不会每天都用到 bisect。也许一周用一次，甚至一个月只用到一次。但是当你想要排查出带入问题的提交点时，bisect 确实能帮你一把。尽管并非必须，控制好每次提交的改动规模不要太大将对 bisect 排查大有帮助。如果每个提交都包含了大量的改动，那么就算 bisect 帮你找到这个提交，你也不得不埋头于大量的改动中苦苦搜寻 bug 的踪迹。因此，我推荐的提交策略是嫌大不嫌多。

你呢？你经常使用 bisect 吗？

第5章 远程仓库

到目前为止，我们已经掌握了如何在Git仓库里对一个文件进行时光穿梭，你再也不用担心文件备份或者丢失的问题了。

可是有用过集中式版本控制系统SVN的童鞋会站出来说，这些功能在SVN里早就有了，没看出Git有什么特别的地方。

没错，如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。为了保证你现在所学的Git物超所值，将来绝对不会后悔，同时为了打击已经不幸学了SVN的童鞋，本章开始介绍Git的杀手级功能之一（注意是之一，也就是后面还有之二，之三……）：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，现实生活中是不会有人这么傻的在一台电脑上搞几个远程库玩，因为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，所以我也不告诉你在一台电脑上怎么克隆多个仓库。

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。

在继续阅读后续内容前，请自行注册GitHub账号。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：

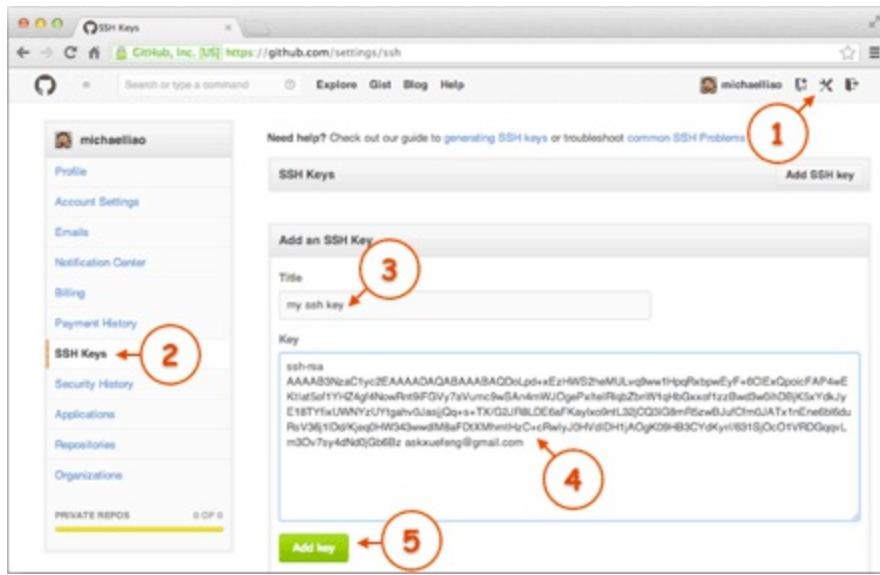
```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

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

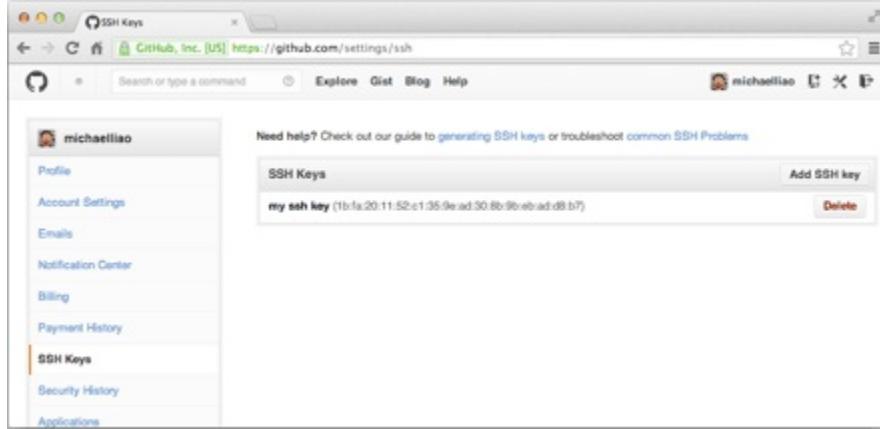
如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有id_rsa和id_rsa.pub两个文件，这两个就是SSH Key的秘钥对，id_rsa是私钥，不能泄露出去，id_rsa.pub是公钥，可以放心地告诉任何人。

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

然后，点“Add SSH Key”，填上任意Title，在Key文本框里粘贴id_rsa.pub文件的内容：



点“Add Key”，你就应该看到已经添加的Key：



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自己推送的，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。

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

如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。

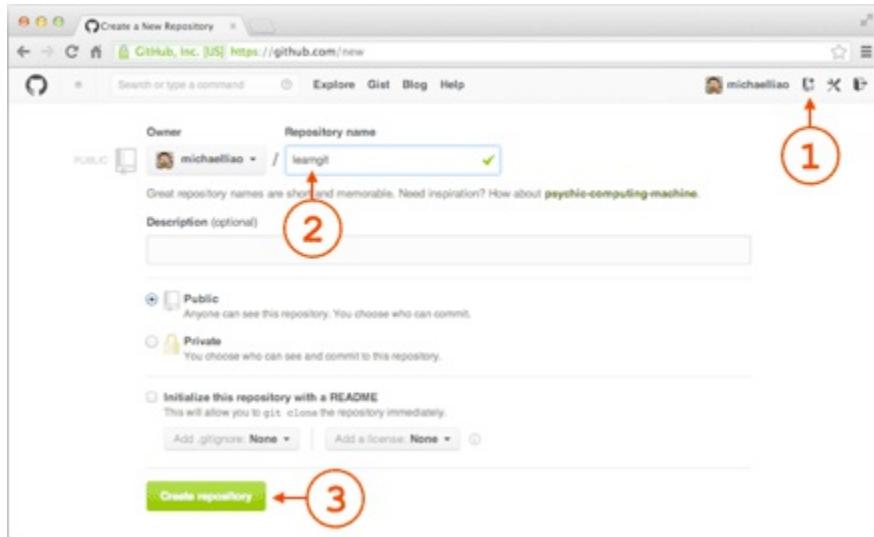
小结

“有了远程仓库，妈妈再也不用担心我的硬盘了。” ——Git点读机

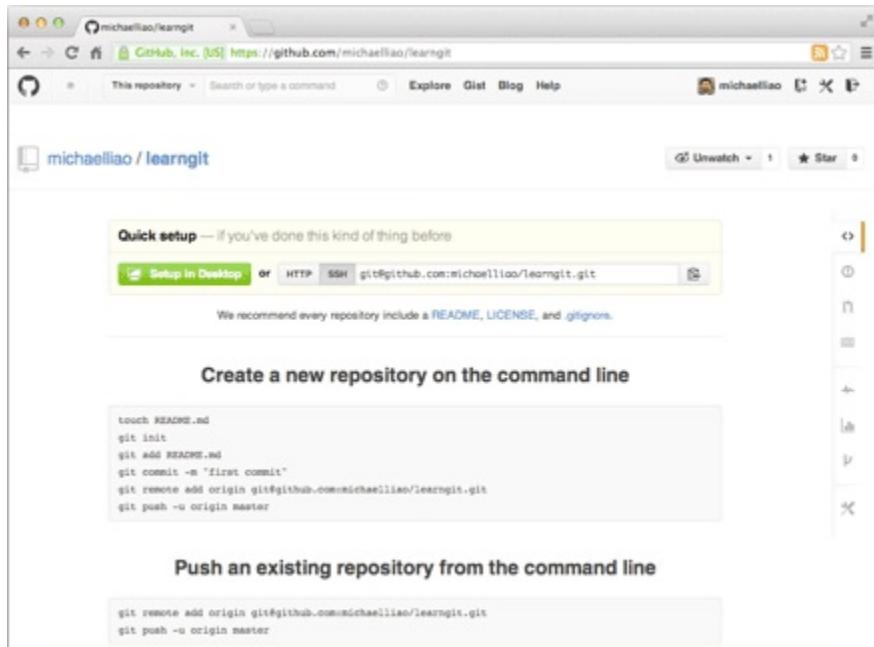
5.1 添加远程仓库

现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

首先，登陆GitHub，然后，在右上角找到“Create a new repo”按钮，创建一个新的仓库：



在Repository name填入learngit，其他保持默认设置，点击“Create repository”按钮，就成功地创建了一个新的Git仓库：



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

现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

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

请千万注意，把上面的michaelliao替换成你自己的GitHub账户名，否则，你在本地关联的就是我的远程库，关联没有问题，但是你以后推送是推不上去的，因为你的SSH Key公钥不在我的账户列表中。

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。

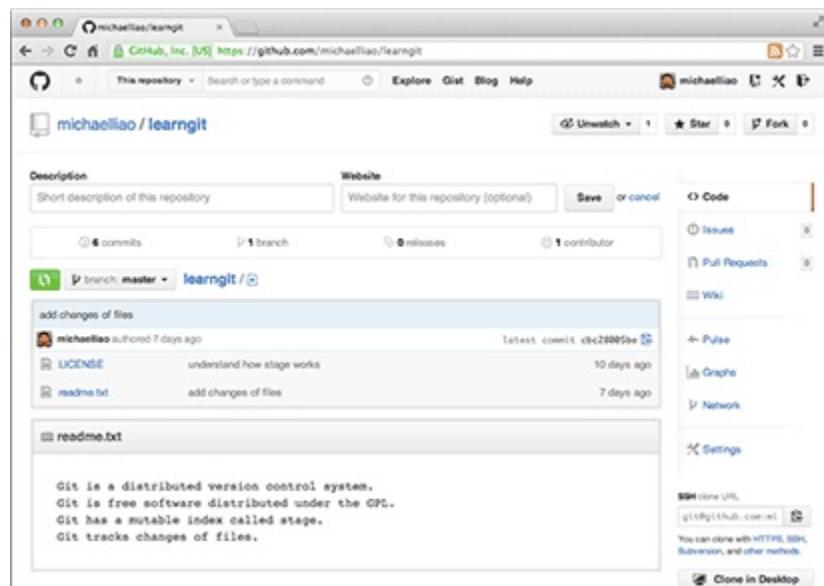
下一步，就可以把本地库的所有内容推送到远程库上：

```
$ git push -u origin master
Counting objects: 19, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (19/19), done.
Writing objects: 100% (19/19), 13.73 KiB, done.
Total 23 (delta 6), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

由于远程库是空的，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分支内容推送到远程新的master分支，还会把本地的master分支和远程的master分支关联起来，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 (xx.xx.xx.xx)' can't be established.  
RSA key fingerprint is xx.xx.xx.xx.xx.  
Are you sure you want to continue connecting (yes/no)?
```

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

Git会输出一个警告，告诉你已经把GitHub的Key添加到本机的一个信任列表里了：

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

这个警告只会出现一次，后面的操作就不会有任何警告了。

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；

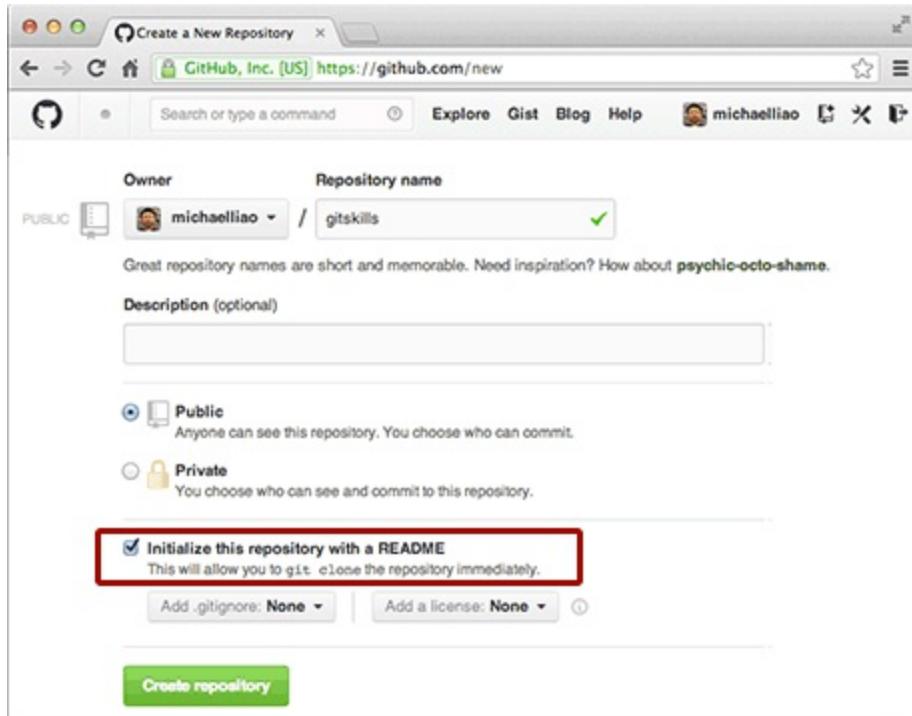
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

5.2 从远程仓库克隆

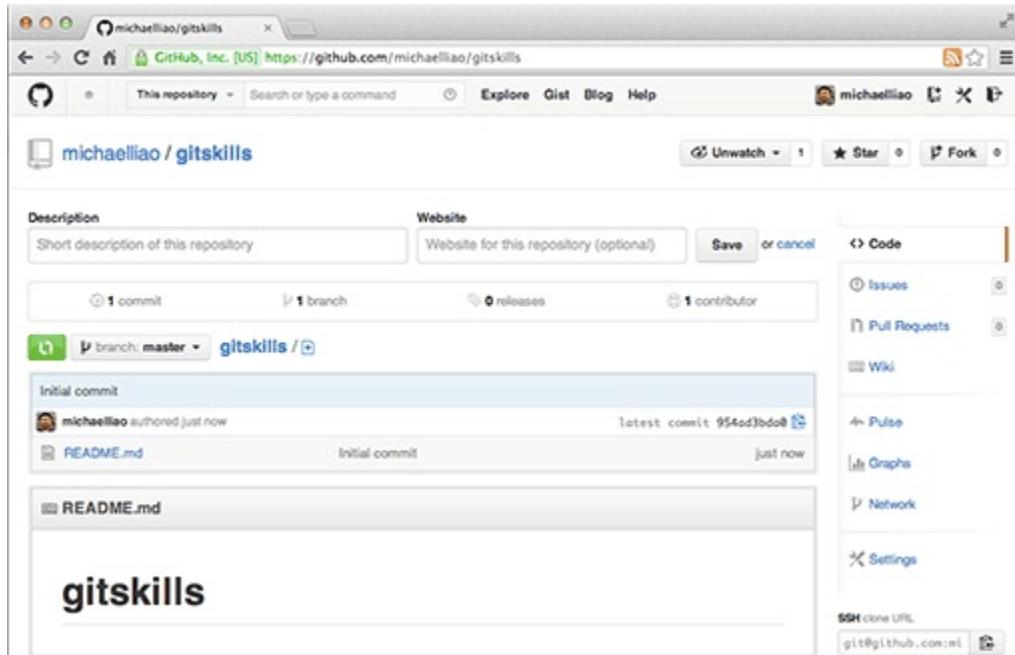
上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。

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gitskills：



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



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：

```
$ git clone git@github.com:michaelliao/gitskills.git
Cloning into 'gitskills'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.

$ cd gitskills
$ ls
README.md
```

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

video：<http://michaelliao.gitcafe.io/video/git-clone.mp4>

如果有多人协作开发，那么每个人各自从远程克隆一份就可以了。

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

使用https除了速度慢以外，还有个最大的麻烦是每次推送都必须输入口令，但是在某些只开放http端口的公司内部就无法使用ssh协议而只能用https。

小结

要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

5.3 Git push与pull的默认行为

一直以来对 `git push` 与 `git pull` 命令的默认行为感觉混乱，今天抽空总结下。

git push

通常对于一个本地的新建分支，例如 `git checkout -b develop`，在 `develop` 分支 commit 了代码之后，如果直接执行 `git push` 命令，`develop` 分支将不会被 push 到远程仓库（但此时 `git push` 操作有可能会推送一些代码到远程仓库，这取决于我们本地 `git config` 配置中的 `push.default` 默认行为，下文将会逐一详解）。

因此我们至少需要显式指定将要推送的分支名，例如 `git push origin develop`，才能将本地新分支推送到远程仓库。

当我们通过显式指定分支名进行初次 push 操作后，本地有了新的 commit，此时执行 `git push` 命令会有什么效果呢？

如果你未曾改动过 `git config` 中的 `push.default` 属性，根据我们使用的 git 不同版本（Git 2.0 之前或之后），`git push` 通常会有两种截然不同的行为：

1. `develop` 分支中本地新增的 commit 被 push 到远程仓库
2. push 失败，并收到 git 如下的警告

```
fatal: The current branch new has no upstream branch.  
To push the current branch and set the remote as upstream, use  
  
    git push --set-upstream origin develop
```

为什么 git 版本不同会有两种不同的 push 行为？

因为在 [git 的全局配置中](#)，有一个 `push.default` 属性，其决定了 `git push` 操作的默认行为。在 Git 2.0 之前，这个属性的默认被设为 `matching`，2.0 之后则被更改为了 `simple`。

我们可以通过 `git version` 确定当前的 git 版本（如果小于 2.0，更新是个更好的选择），通过 `git config --global push.default 'option'` 改变 `push.default` 的默认行为（或者也可直接编辑 `~/.gitconfig` 文件）。

`push.default` 有以下几个可选值：`nothing`, `current`, `upstream`, `simple`, `matching`

其用途分别为：

- **nothing** - push 操作无效，除非显式指定远程分支，例如 `git push origin develop`（我觉得。。。可以给那些不愿学 git 的同事配上此项）。
- **current** - push 当前分支到远程同名分支，如果远程同名分支不存在则自动创建同名分支。
- **upstream** - push 当前分支到它的 upstream 分支上（这一项其实用于经常从本地分支 push/pull 到同一远程仓库的情景，这种模式叫做 central workflow）。
- **simple** - simple 和 upstream 是相似的，只有一点不同，simple 必须保证本地分支和它的远程 upstream 分支同名，否则会拒绝 push 操作。
- **matching** - push 所有本地和远程两端都存在的同名分支。

因此如果我们使用了 git 2.0 之前的版本，`push.default = matching`，`git push` 后则会推送当前分支代码到远程分支，而 2.0 之后，`push.default = simple`，如果没有指定当前分支的 upstream 分支，就会收到上文的 fatal 提示。

upstream & downstream

说到这里，需要解释一下git中的upstream到底是什么：

git中存在upstream和downstream，简言之，当我们把仓库A中某分支x的代码push到仓库B分支y，此时仓库B的这个分支y就叫做A中x分支的upstream，而x则被称作y的downstream，这是一个相对关系，每一个本地分支都相对地可以有一个远程的upstream分支（注意这个upstream分支可以不同名，但通常我们都会使用同名分支作为upstream）。

初次提交本地分支，例如`git push origin develop`操作，并不会定义当前本地分支的upstream分支，我们可以通过`git push --set-upstream origin develop`，关联本地develop分支的upstream分支，另一个更为简洁的方式是初次push时，加入-u参数，例如`git push -u origin develop`，这个操作在push的同时会指定当前分支的upstream。

注意`push.default = current`可以在远程同名分支不存在的情况下自动创建同名分支，有些时候这也是个极其方便的模式，比如初次push你可以直接输入`git push`而不必显示指定远程分支。

git pull

弄清楚`git push`的默认行为后，再来看看`git pull`。

当我们未指定当前分支的upstream时，通常`git pull`操作会得到如下的提示：

```
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> new1
```

`git pull`的默认行为和`git push`完全不同。当我们执行`git pull`的时候，实际上是做了`git fetch + git merge`操作，fetch操作将会更新本地仓库的remote tracking，也就是refs/remotes中的代码，并不会对refs/heads中本地当前的代码造成影响。

当我们进行pull的第二个行为merge时，对git来说，如果我们没有设定当前分支的upstream，它并不知道我们要合并哪个分支到当前分支，所以我们需要通过下面的代码指定当前分支的upstream：

```
git branch --set-upstream-to=origin/<branch> develop  
// 或者git push --set-upstream origin develop
```

实际上，如果我们没有指定upstream，git在merge时会访问git config中当前分支(develop)merge的默认配置，我们可以通过配置下面的内容指定某个分支的默认merge操作

```
[branch "develop"]  
remote = origin  
merge = refs/heads/develop // [1]为什么不是refs/remotes/develop?
```

或者通过command-line直接设置：

```
git config branch.develop.merge refs/heads/develop
```

这样当我们在develop分支git pull时，如果没有指定upstream分支，git将根据我们的config文件去 `merge origin/develop`；如果指定了upstream分支，则会忽略config中的merge默认配置。

以上就是git push和git pull操作的全部默认行为，如有错误，欢迎斧正

[1] 为什么`merge = refs/heads/develop` 而不是`refs/remotes/develop`？

因为这里merge指代的是我们想要merge的远程分支，是remote上的`refs/heads/develop`，文中即是origin上的`refs/heads/develop`，这和我们在本地直接执行 `git merge` 是不同的(本地执行 `git merge origin/develop` 则是直接merge `refs/remotes/develop`)。

Git 2.0 更改 push default 为 simple

如果你最近更新了 Git，你可能会在执行 git push 时看到如下消息：

```
Allen@Android MINGW64 /d/GitBook/alleniverson/javaweb (master)
$ git push
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master
```

warning: push.default is unset; its implicit value is changing in
Git 2.0 from 'matching' to 'simple'. To squelch this message
and maintain the current behavior after the default changes, use:

`git config --global push.default matching`

To squelch this message and adopt the new behavior now, use:

`git config --global push.default simple`

Matching

'matching' 参数是 Git 1.x 的默认行为，其意是如果你执行 git push 但没有指定分支，它将 push 所有你本地的分支到远程仓库中对应匹配的分支。

Simple

而 Git 2.x 默认的是 simple，意味着执行 git push 没有指定分支时，只有当前分支会被 push 到你使用 git pull 获取的代码。

修改默认设置

从上述消息提示中的解释，我们可以修改全局配置，使之不会每次 push 的时候都进行提示。对于 matching 输入如下命令即可：

```
git config --global push.default matching
```

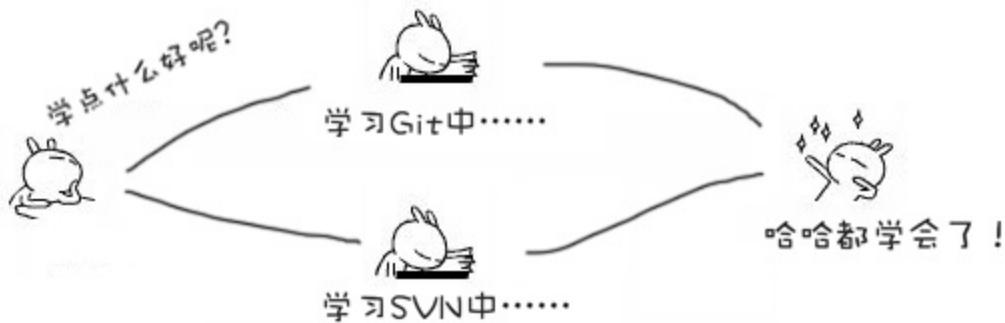
而对于 simple , 请输入 :

```
git config --global push.default simple
```

第6章 分支管理

分支就是科幻电影里面的平行宇宙，当你正在电脑前努力学习Git的时候，另一个你正在另一个平行宇宙里努力学习SVN。

如果两个平行宇宙互不干扰，那对现在的你也没啥影响。不过，在某个时间点，两个平行宇宙合并了，结果，你既学会了Git又学会了SVN！



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。

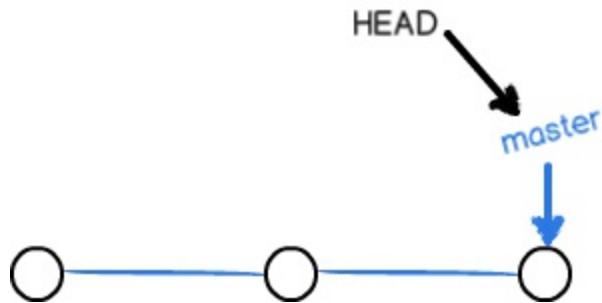
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

但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。

6.1 创建与合并分支

在版本回退里，你已经知道，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。

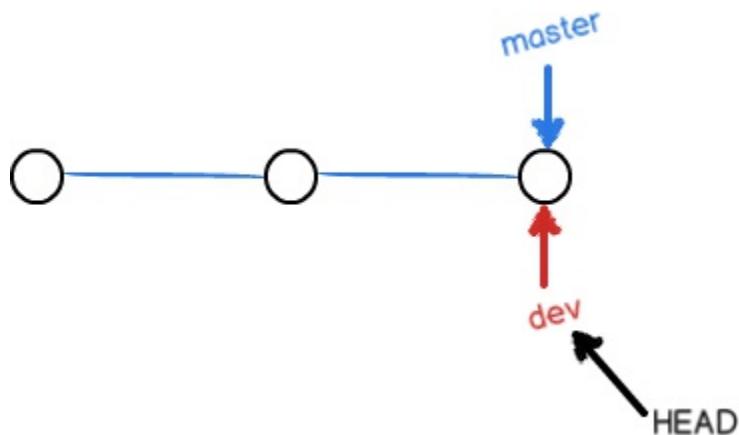
一开始的时候，master分支是一条线，Git用master指向最新的提交，再用HEAD指向master，就能确定当前分支，以及当前分支的提交点：



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：

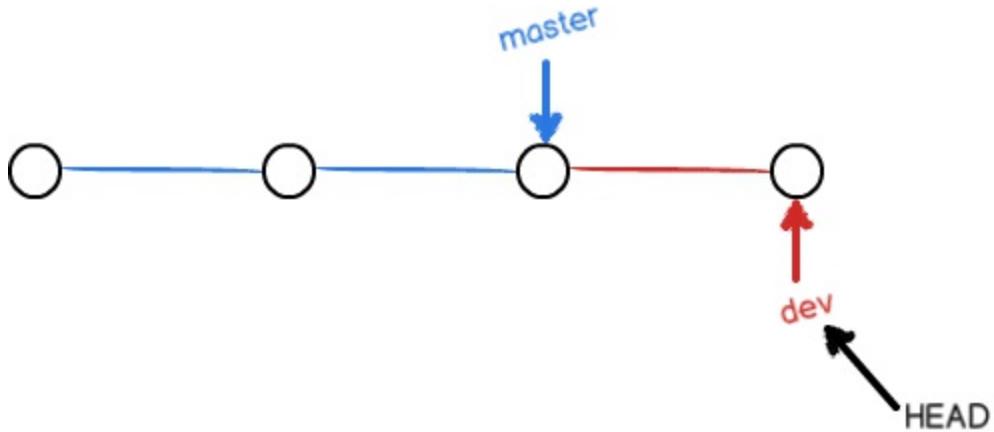
video : <http://github.liaoxuefeng.com/sinaweibopy/video/master-branch-forward.mp4>

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：

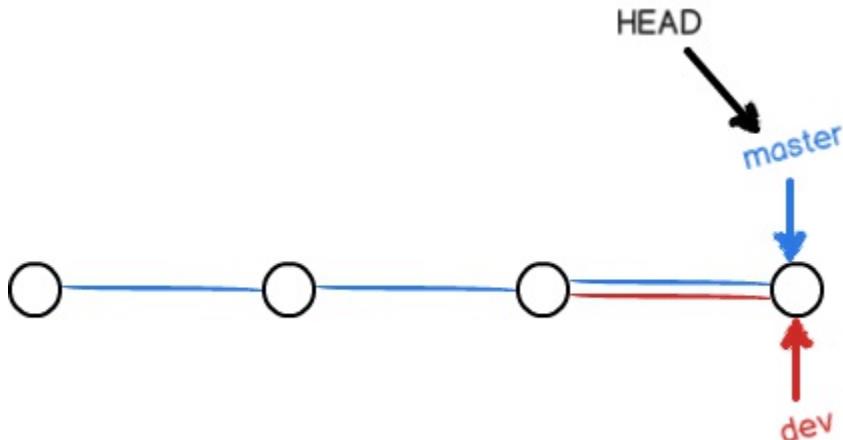


你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

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

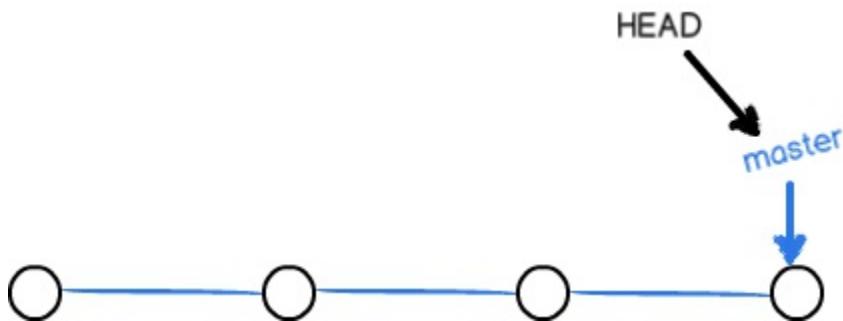


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



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：



真是太神奇了，你看得出来有些提交是通过分支完成的吗？

video：<http://michaelliao.gitcafe.io/video/master-and-dev-ff.mp4>

下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

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

```
$ git branch dev
$ git checkout dev
Switched to branch 'dev'
```

然后，用git branch命令查看当前分支：

```
$ git branch
* dev
  master
```

git branch命令会列出所有分支，当前分支前面会标一个*号。

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

```
Creating a new branch is quick.
```

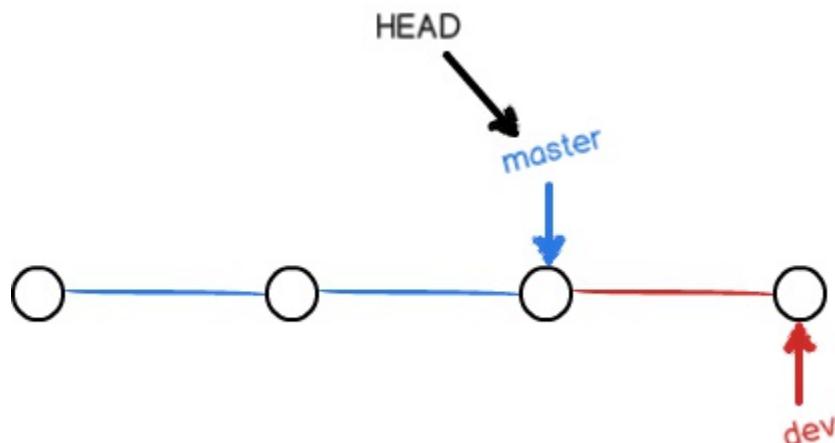
然后提交：

```
$ git add readme.txt
$ git commit -m "branch test"
[dev fec145a] branch test
 1 file changed, 1 insertion(+)
```

现在，dev分支的工作完成，我们就可以切换回master分支：

```
$ git checkout master
Switched to branch 'master'
```

切换回master分支后，再查看一个readme.txt文件，刚才添加的内容不见了！因为那个提交是在dev分支上，而master分支此刻的提交点并没有变：



现在，我们把dev分支的工作成果合并到master分支上：

```
$ git merge dev
Updating d17efd8..fec145a
Fast-forward
 readme.txt |    1 +
 1 file changed, 1 insertion(+)
```

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。

合并完成后，就可以放心地删除dev分支了：

```
$ git branch -d dev  
Deleted branch dev (was fec145a).
```

删除后，查看branch，就只剩下master分支了：

```
$ git branch  
* master
```

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

video：<http://michaelliao.gitcafe.io/video/create-dev-merge-delete.mp4>

小结

Git鼓励大量使用分支：

- 查看分支：git branch
- 创建分支：git branch
- 切换分支：git checkout
- 创建+切换分支：git checkout -b
- 合并某分支到当前分支：git merge
- 删除分支：git branch -d

6.2 解决冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

```
$ git checkout -b feature1
Switched to a new branch 'feature1'
```

修改readme.txt最后一行，改为：

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

在feature1分支上提交：

```
$ git add readme.txt
$ git commit -m "AND simple"
[feature1 75a857c] AND simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到master分支：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
```

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

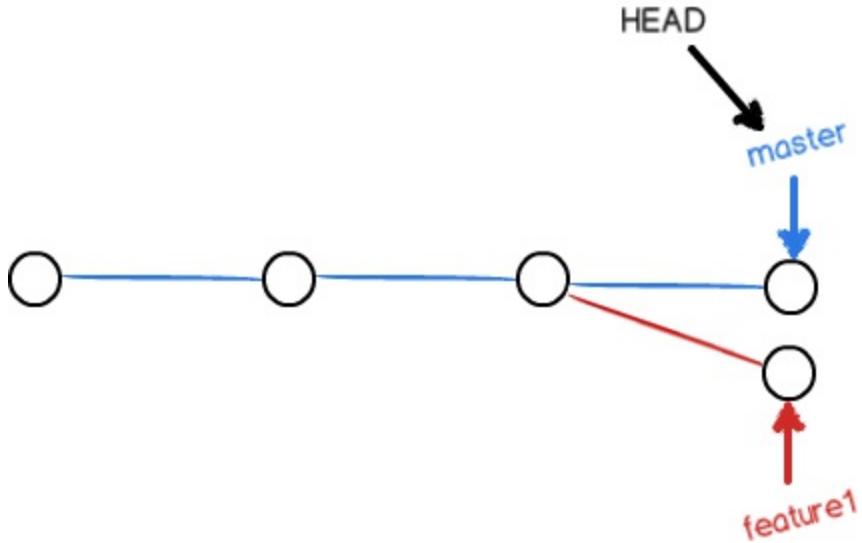
在master分支上把readme.txt文件的最后一行改为：

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

提交：

```
$ git add readme.txt
$ git commit -m "& simple"
[master 400b400] & simple
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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



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

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the result.
```

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

```
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:    readme.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

我们可以直接查看readme.txt的内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<< HEAD
Creating a new branch is quick & simple.
=====
Creating a new branch is quick AND simple.
>>>>> feature1
```

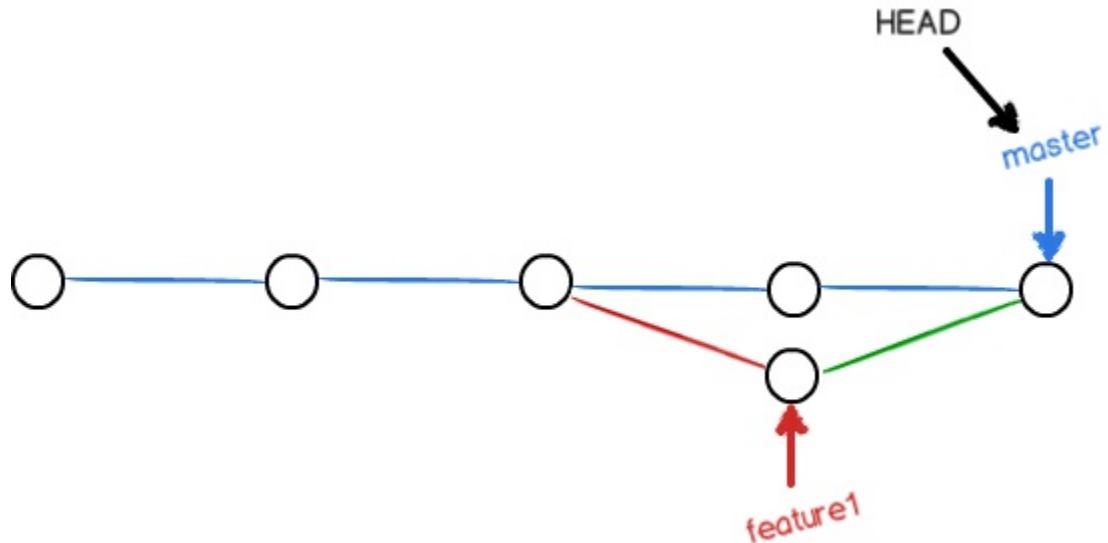
Git用<<<<<，=====，>>>>>标记出不同分支的内容，我们修改如下后保存：

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

再提交：

```
$ git add readme.txt  
$ git commit -m "conflict fixed"  
[master 59bc1cb] conflict fixed
```

现在， master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：

```
$ git log --graph --pretty=oneline --abbrev-commit  
* 59bc1cb conflict fixed  
|\  
| * 75a857c AND simple  
| * 400b400 & simple  
|/  
* fec145a branch test  
...
```

最后，删除feature1分支：

```
$ git branch -d feature1  
Deleted branch feature1 (was 75a857c).
```

工作完成。

video : <http://michaelliao.gitcafe.io/video/resolv-conflix-on-merge.mp4>

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。

用git log --graph命令可以看到分支合并图。

6.3 分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。

如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。

下面我们实战一下--no-ff方式的git merge：

首先，仍然创建并切换dev分支：

```
$ git checkout -b dev
Switched to a new branch 'dev'
```

修改readme.txt文件，并提交一个新的commit：

```
$ git add readme.txt
$ git commit -m "add merge"
[dev 6224937] add merge
 1 file changed, 1 insertion(+)
```

现在，我们切换回master：

```
$ git checkout master
Switched to branch 'master'
```

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

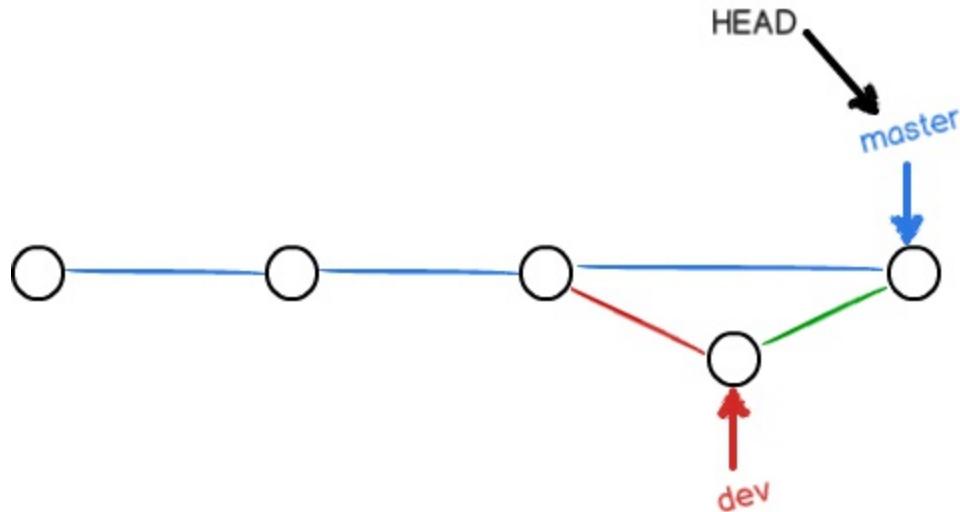
```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt |    1 +
 1 file changed, 1 insertion(+)
```

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

合并后，我们用git log看看分支历史：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   7825a50 merge with no-ff
|\ 
| * 6224937 add merge
|/
*   59bc1cb conflict fixed
...
```

可以看到，不使用Fast forward模式，merge后就像这样：



video : <http://michaelliao.gitcafe.io/video/merge-with-no-ff.mp4>

分支策略

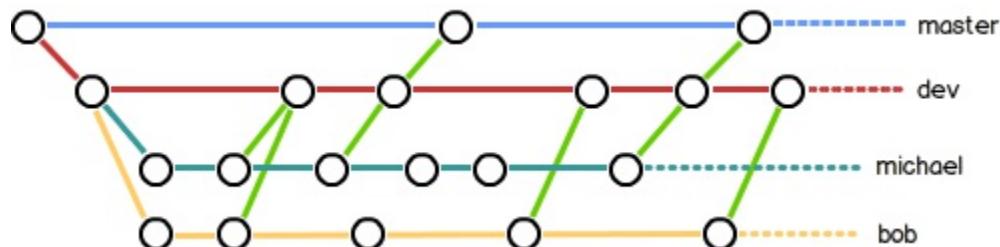
在实际开发中，我们应该按照几个基本原则进行分支管理：

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：



小结

Git分支十分强大，在团队开发中应该充分应用。

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

6.4 Bug分支

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

当你接到一个修复一个代号101的bug的任务时，很自然地，你想创建一个分支issue-101来修复它，但是，等等，当前正在dev上进行的工作还没有提交：

```
$ git status
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hello.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
```

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需1天时间。但是，必须在两个小时內修复该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：

```
$ git stash
Saved working directory and index state WIP on dev: 6224937 add merge
HEAD is now at 6224937 add merge
```

现在，用git status查看工作区，就是干净的（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：

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 6 commits.
$ git checkout -b issue-101
Switched to a new branch 'issue-101'
```

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

```
$ git add readme.txt
$ git commit -m "fix bug 101"
[issue-101 cc17032] fix bug 101
 1 file changed, 1 insertion(+), 1 deletion(-)
```

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

```
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 2 commits.
```

```
$ git merge --no-ff -m "merged bug fix 101" issue-101
Merge made by the 'recursive' strategy.
 README.txt |    2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git branch -d issue-101
Deleted branch issue-101 (was cc17032).
```

太棒了，原计划两个小时的bug修复只花了5分钟！现在，是时候接着回到dev分支干活了！

```
$ git checkout dev
Switched to branch 'dev'
$ git status
# On branch dev
nothing to commit (working directory 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：

```
$ git stash list
stash@{0}: WIP on dev: 6224937 add merge
```

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：

```
$ git stash pop
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hello.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.txt
#
Dropped refs/stash@{0} {f624f8e5f082f2df2bed8a4e09c12fd2943bdd40}
```

再用git stash list查看，就看不到任何stash内容了：

```
$ git stash list
```

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

```
$ git stash apply stash@{0}
```

video：<http://michaelliao.gitcafe.io/video/stash-fix-bug.mp4>

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。

6.5 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。

现在，你终于接到了一个新任务：开发代号为Vulcan的新功能，该功能计划用于下一代星际飞船。

于是准备开发：

```
$ git checkout -b feature-vulcan
Switched to a new branch 'feature-vulcan'
```

5分钟后，开发完毕：

```
$ git add vulcan.py
$ git status
# On branch feature-vulcan
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   vulcan.py
#
$ git commit -m "add feature vulcan"
[feature-vulcan 756d4af] add feature vulcan
 1 file changed, 2 insertions(+)
 create mode 100644 vulcan.py
```

切回dev，准备合并：

```
$ git checkout dev
```

一切顺利的话，feature分支和bug分支是类似的，合并，然后删除。但是，就在此时，接到上级命令，因经费不足，新功能必须取消！虽然白干了，但是这个分支还是必须就地销毁：

```
$ git branch -d feature-vulcan
error: The branch 'feature-vulcan' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-vulcan'.
```

销毁失败。Git友情提醒，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，需要使用命令git branch -D feature-vulcan。

现在我们强行删除：

```
$ git branch -D feature-vulcan
Deleted branch feature-vulcan (was 756d4af).
```

终于删除成功！

video：<http://github.liaoxuefeng.com/sinaweibopy/video/force-delete-br.mp4>

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 强行删除。

6.6 多人协作

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

要查看远程库的信息，用git remote：

```
$ git remote  
origin
```

或者，用git remote -v显示更详细的信息：

```
$ git remote -v  
origin  git@github.com:michaelliao/learngit.git (fetch)  
origin  git@github.com:michaelliao/learngit.git (push)
```

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

推送分支

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

```
$ git push origin master
```

如果要推送其他分支，比如dev，就改成：

```
$ git push origin 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，分支完全可以在本地自己藏着玩，是否推送，视你的心情而定！

video：<http://michaelliao.gitcafe.io/video/git-push-origin.mp4>

抓取分支

多人协作时，大家都会往master和dev分支上推送各自的修改。

现在，模拟一个你的小伙伴，可以在另一台电脑（注意要把SSH Key添加到GitHub）或者同一台电脑的另一个目录下克隆：

```
$ git clone git@github.com:michaelliao/learngit.git
Cloning into 'learngit'...
remote: Counting objects: 46, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 46 (delta 16), reused 45 (delta 15)
Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.
Resolving deltas: 100% (16/16), done.
```

当你的小伙伴从远程库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：

```
$ git branch
* master
```

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

```
$ git checkout -b dev origin/dev
```

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

```
$ git commit -m "add /usr/bin/env"
[dev 291bea8] add /usr/bin/env
 1 file changed, 1 insertion(+)
$ git push origin dev
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 349 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 fc38031..291bea8  dev -> dev
```

video : <http://michaelliao.gitcafe.io/video/git-push-by-xiaohuoban.mp4>

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：

```
$ git add hello.py
$ git commit -m "add coding: utf-8"
[dev bd6ae48] add coding: utf-8
 1 file changed, 1 insertion(+)
$ git push origin dev
To git@github.com:michaelliao/learngit.git
 ! [rejected]      dev -> dev (non-fast-forward)
error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

推送失败， 因为你的小伙伴的最新提交和你试图推送的提交有冲突， 解决办法也很简单， Git已经提示我们， 先用git pull把最新的提交从origin/dev抓下来， 然后，在本地合并， 解决冲突， 再推送：

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
```

```

Unpacking objects: 100% (3/3), done.
From github.com:michaelliao/learngit
  fc38031..291bea8  dev      -> origin/dev
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details

git pull <remote> <branch>

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

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

```

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

```

$ git branch --set-upstream dev origin/dev
Branch dev set up to track remote branch dev from origin.

```

再pull：

```

$ git pull
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Automatic merge failed; fix conflicts and then commit the result.

```

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

```

$ git commit -m "merge & fix hello.py"
[dev adca45d] merge & fix hello.py
$ git push origin dev
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 747 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
  291bea8..adca45d  dev -> dev

```

video：<http://michaelliao.gitcafe.io/video/git-pull-push-fix.mp4>

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 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；

本地新建的分支如果不推送到远程，对其他人就是不可见的；

从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交；

在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地和远程分支的名称最好一致；

建立本地分支和远程分支的关联，使用git branch --set-upstream branch-name origin/branch-name；

从远程抓取分支，使用git pull，如果有冲突，要先处理冲突。

第7章 标签管理

发布一个版本时，我们通常先在版本库中打一个标签（tag），这样，就唯一确定了打标签时刻的版本。将来无论什么时候，取某个标签的版本，就是把那个打标签的时刻的历史版本取出来。所以，标签也是版本库的一个快照。

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

7.1 创建标签

在Git中打标签非常简单，首先，切换到需要打标签的分支上：

```
$ git branch
* dev
  master
$ git checkout master
Switched to branch 'master'
```

然后，敲命令git tag 就可以打一个新标签：

```
$ git tag v1.0
```

可以用命令git tag查看所有标签：

```
$ git tag
v1.0
```

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

方法是找到历史提交的commit id，然后打上就可以了：

```
$ git log --pretty=oneline --abbrev-commit
6a5819e merged bug fix 101
cc17032 fix bug 101
7825a50 merge with no-ff
6224937 add merge
59bc1cb conflict fixed
400b400 & simple
75a857c AND simple
fec145a branch test
d17efd8 remove test.txt
...
```

比方说要对add merge这次提交打标签，它对应的commit id是6224937，敲入命令：

```
$ git tag v0.9 6224937
```

再用命令git tag查看标签：

```
$ git tag
v0.9
v1.0
```

注意，标签不是按时间顺序列出，而是按字母排序的。可以用git show 查看标签信息：

```
$ git show v0.9
commit 622493706ab447b6bb37e4e2a2f276a20fed2ab4
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Thu Aug 22 11:22:08 2013 +0800
```

```
add merge
```

```
...
```

可以看到，v0.9确实打在add merge这次提交上。

还可以创建带有说明的标签，用-a指定标签名，-m指定说明文字：

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

用命令git show 可以看到说明文字：

```
$ git show v0.1
tag v0.1
Tagger: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 26 07:28:11 2013 +0800

version 0.1 released

commit 3628164fb26d48395383f8f31179f24e0882e1e0
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Tue Aug 20 15:11:49 2013 +0800

append GPL
```

还可以通过-s用私钥签名一个标签：

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

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

```
gpg: signing failed: secret key not available
error: gpg failed to sign the data
error: unable to sign the tag
```

如果报错，请参考GnuPG帮助文档配置Key。

用命令git show <tagname>可以看到PGP签名信息：

```
$ git show v0.2
tag v0.2
Tagger: Michael Liao <askxuefeng@gmail.com>
Date:   Mon Aug 26 07:28:33 2013 +0800

signed version 0.2 released
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.12 (Darwin)

iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...
-----END PGP SIGNATURE-----

commit fec145accd63cdc9ed95a2f557ea0658a2a6537f
Author: Michael Liao <askxuefeng@gmail.com>
Date:   Thu Aug 22 10:37:30 2013 +0800

branch test
```

用PGP签名的标签是不可伪造的，因为可以验证PGP签名。验证签名的方法比较复杂，这里就不介绍了。

video : <http://michaelliao.gitcafe.io/video/git-tags.mp4>

小结

- 命令git tag <name>用于新建一个标签， 默认为HEAD， 也可以指定一个commit id；
- git tag -a <tagname> -m "blablabla..."可以指定标签信息；
- git tag -s <tagname> -m "blablabla..."可以用PGP签名标签；
- 命令git tag可以查看所有标签。

7.2 操作标签

如果标签打错了，也可以删除：

```
$ git tag -d v0.1
Deleted tag 'v0.1' (was e078af9)
```

因为创建的标签都只存储在本地，不会自动推送到远程。所以，打错的标签可以在本地安全删除。

如果要推送某个标签到远程，使用命令git push origin：

```
$ git push origin v1.0
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new tag]      v1.0 -> v1.0
```

或者，一次性推送全部尚未推送到远程的本地标签：

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 554 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:michaelliao/learngit.git
 * [new tag]      v0.2 -> v0.2
 * [new tag]      v0.9 -> v0.9
```

如果标签已经推送到远程，要删除远程标签就麻烦一点，先从本地删除：

```
$ git tag -d v0.9
Deleted tag 'v0.9' (was 6224937)
```

然后，从远程删除。删除命令也是push，但是格式如下：

```
$ git push origin :refs/tags/v0.9
To git@github.com:michaelliao/learngit.git
 - [deleted]      v0.9
```

要看看是否真的从远水库删除了标签，可以登陆GitHub查看。

video：<http://github.liaoxuefeng.com/sinaweibopy/video/git-tag-d.mp4>

小结

- 命令git push origin 可以推送一个本地标签；
- 命令git push origin --tags可以推送全部未推送过的本地标签；
- 命令git tag -d 可以删除一个本地标签；
- 命令git push origin :refs/tags/可以删除一个远程标签。

第8章 使用GitHub

我们一直用GitHub作为免费的远程仓库，如果是个人的开源项目，放到GitHub上是完全没有问题的。其实GitHub还是一个开源协作社区，通过GitHub，既可以让别人参与你的开源项目，也可以参与别人的开源项目。

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

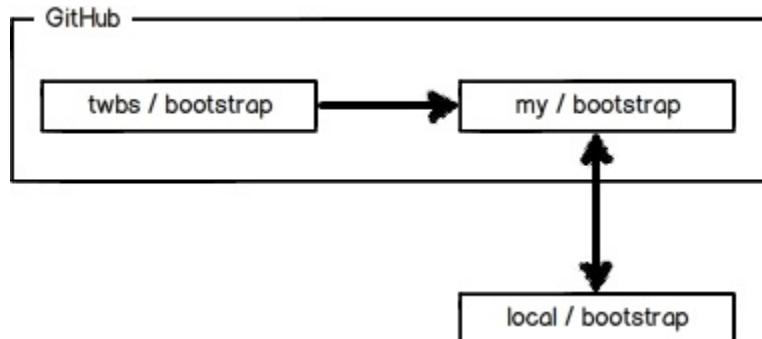
但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

如何参与一个开源项目呢？比如人气极高的bootstrap项目，这是一个非常强大的CSS框架，你可以访问它的项目主页<https://github.com/twbs/bootstrap>，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

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

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

Bootstrap的官方仓库twbs/bootstrap、你在GitHub上克隆的仓库my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：



如果你想修复bootstrap的一个bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送。

如果你希望bootstrap的官方库能接受你的修改，你就可以在GitHub上发起一个pull request。当然，对方是否接受你的pull request就不一定了。

如果你没能力修改bootstrap，但又想要试一把pull request，那就Fork一下我的仓库：

<https://github.com/michaelliao/learngit>，创建一个your-github-id.txt的文本文件，写点自己学习Git的心得，然后推送一个pull request给我，我会视心情而定是否接受。

小结

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

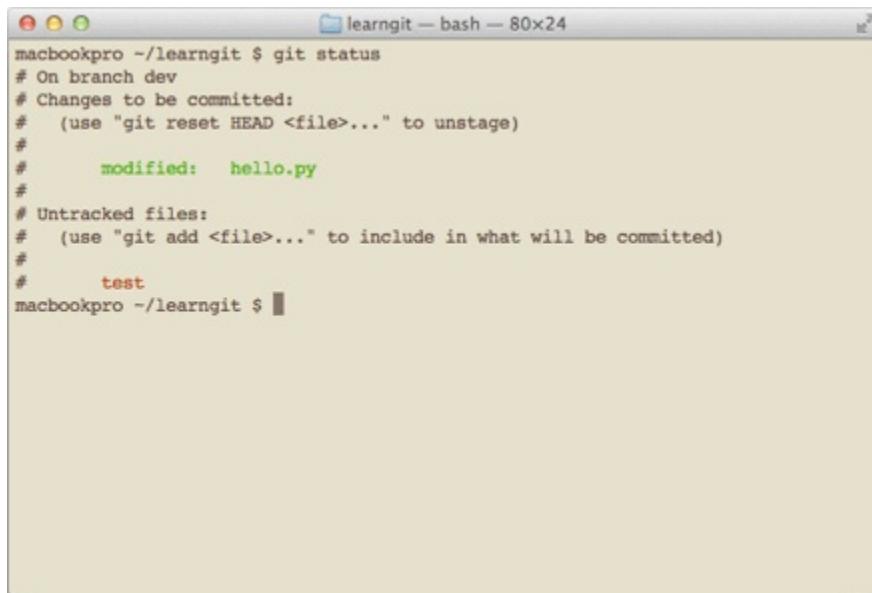
第9章 自定义Git

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

比如，让Git显示颜色，会让命令输出看起来更醒目：

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

这样，Git会适当地显示不同的颜色，比如git status命令：



```
macbookpro ~/learngit $ git status
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   hello.py
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       test
macbookpro ~/learngit $
```

文件名就会标上颜色。

我们在后面还会介绍如何更好地配置Git，以便让你的工作更高效。

9.1 忽略特殊文件

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

好在Git考虑到了大家的感受，这个问题解决起来也很简单，在Git工作区的根目录下创建一个特殊的.gitignore文件，然后把要忽略的文件名填进去，Git就会自动忽略这些文件。

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

忽略文件的原则是：

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

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

```
# Windows:  
Thumbs.db  
ehthumbs.db  
Desktop.ini
```

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

```
# Python:  
*.py[cod]  
*.so  
*.egg  
.egg-info  
dist  
build
```

加上你自己定义的文件，最终得到一个完整的.gitignore文件，内容如下：

```
# Windows:  
Thumbs.db  
ehthumbs.db  
Desktop.ini  
  
# Python:  
*.py[cod]  
*.so  
*.egg  
.egg-info  
dist  
build  
  
# My configurations:  
db.ini  
deploy_key_rsa
```

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

使用Windows的童鞋注意了，如果你在资源管理器里新建一个.gitignore文件，它会非常弱智地提示你必须输入文件名，但是在文本编辑器里“保存”或者“另存为”就可以把文件保存为.gitignore了。

有些时候，你想添加一个文件到Git，但发现添加不了，原因是这个文件被.gitignore忽略了：

```
$ git add App.class
The following paths are ignored by one of your .gitignore files:
App.class
Use -f if you really want to add them.
```

如果你确实想添加该文件，可以用-f强制添加到Git：

```
$ git add -f App.class
```

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

```
$ git check-ignore -v App.class
.gitignore:3:*.class    App.class
```

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

小结

- 忽略某些文件时，需要编写.gitignore；
- .gitignore文件本身要放到版本库里，并且可以对.gitignore做版本管理！

9.2 配置别名

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

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

我们只需要敲一行命令，告诉Git，以后st就表示status：

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

好了，现在敲git st看看效果。

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

```
$ git config --global alias.co checkout  
$ git config --global alias.ci commit  
$ git config --global alias.br branch
```

以后提交就可以简写成：

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

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

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

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

当你敲入命令：

```
$ git unstage test.py
```

实际上Git执行的是：

```
$ git reset HEAD test.py
```

配置一个git last，让其显示最后一次提交信息：

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

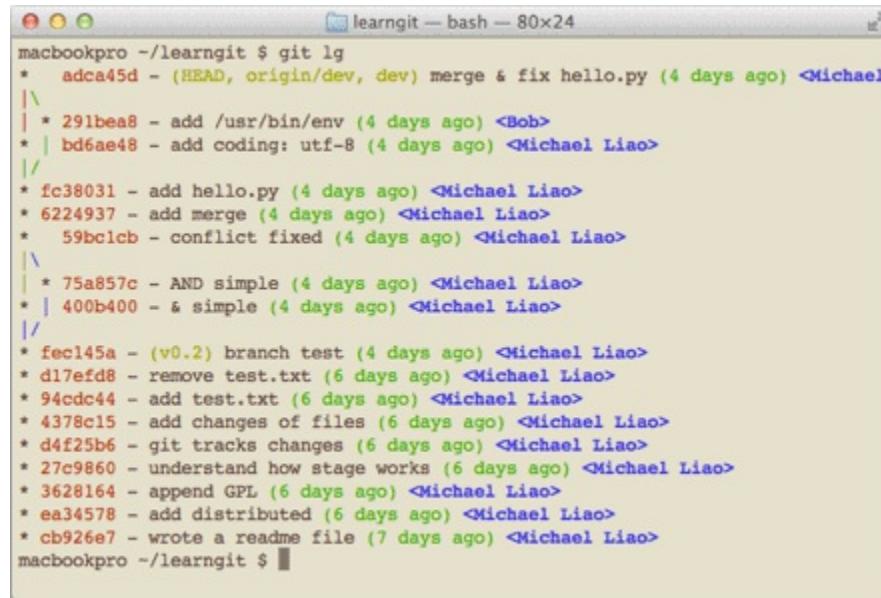
这样，用git last就能显示最近一次的提交：

```
$ git last  
commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2  
Merge: bd6ae48 291bea8  
Author: Michael Liao <askxuefeng@gmail.com>  
Date: Thu Aug 22 22:49:22 2013 +0800  
  
merge & fix hello.py
```

甚至还有人丧心病狂地把lg配置成了：

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

来看看git lg的效果：



```
macbookpro ~/learngit $ git lg
* adca45d - (HEAD, origin/dev, dev) merge & fix hello.py (4 days ago) <Michael
|\ 
| * 291bea8 - add /usr/bin/env (4 days ago) <Bob>
| | bd6ae48 - add coding: utf-8 (4 days ago) <Michael Liao>
| |
| * fc38031 - add hello.py (4 days ago) <Michael Liao>
* 6224937 - add merge (4 days ago) <Michael Liao>
* 59bc1cb - conflict fixed (4 days ago) <Michael Liao>
|\ 
| * 75a857c - AND simple (4 days ago) <Michael Liao>
| | 400b400 - & simple (4 days ago) <Michael Liao>
| |
* fec145a - (v0.2) branch test (4 days ago) <Michael Liao>
* d17efd8 - remove test.txt (6 days ago) <Michael Liao>
* 94cdc44 - add test.txt (6 days ago) <Michael Liao>
* 4378c15 - add changes of files (6 days ago) <Michael Liao>
* d4f25b6 - git tracks changes (6 days ago) <Michael Liao>
* 27c9860 - understand how stage works (6 days ago) <Michael Liao>
* 3628164 - append GPL (6 days ago) <Michael Liao>
* ea34578 - add distributed (6 days ago) <Michael Liao>
* cb926e7 - wrote a readme file (7 days ago) <Michael Liao>
macbookpro ~/learngit $
```

为什么不早点告诉我？别激动，咱不是为了多记几个英文单词嘛！

配置文件

配置Git的时候，加上--global是针对当前用户起作用的，如果不加，那只针对当前的仓库起作用。

配置文件放哪了？每个仓库的Git配置文件都放在.git/config文件中：

```
$ cat .git/config
[core]
repositoryformatversion = 0
filemode = true
bare = false
logallrefupdates = true
ignorecase = true
precomposeunicode = true
[remote "origin"]
url = git@github.com:michaelliao/learngit.git
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
remote = origin
merge = refs/heads/master
[alias]
last = log -1
```

别名就在[alias]后面，要删除别名，直接把对应的行删掉即可。

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

```
$ cat .gitconfig
```

```
[alias]
co = checkout
ci = commit
br = branch
st = status
[user]
name = Your Name
email = your@email.com
```

配置别名也可以直接修改这个文件，如果改错了，可以删掉文件重新通过命令配置。

小结

给Git配置好别名，就可以输入命令时偷个懒。我们鼓励偷懒。

9.3 搭建Git服务器

在远程仓库一节中，我们讲了远程仓库实际上和本地仓库没啥不同，纯粹为了7x24小时开机并交换大家的修改。

GitHub就是一个免费托管开源代码的远程仓库。但是对于某些视源代码如生命的商业公司来说，既不想公开源代码，又舍不得给GitHub交保护费，那就只能自己搭建一台Git服务器作为私有仓库使用。

搭建Git服务器需要准备一台运行Linux的机器，强烈推荐用Ubuntu或Debian，这样，通过几条简单的apt命令就可以完成安装。

假设你已经有sudo权限的用户账号，下面，正式开始安装。

第一步，安装git：

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

第二步，创建一个git用户，用来运行git服务：

```
$ sudo adduser git
```

第三步，创建证书登录：

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

第四步，初始化Git仓库：

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

```
$ sudo git init --bare sample.git
```

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

```
$ sudo chown -R git:git sample.git
```

第五步，禁用shell登录：

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

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

改为：

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

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

第六步，克隆远程仓库：

现在，可以通过git clone命令克隆远程仓库了，在各自的电脑上运行：

```
$ git clone git@server:/srv/sample.git
Cloning into 'sample'...
warning: You appear to have cloned an empty repository.
```

剩下的推送就简单了。

管理公钥

如果团队很小，把每个人的公钥收集起来放到服务器的/home/git/.ssh/authorized_keys文件里就是可行的。如果团队有几百号人，就没法这么玩了，这时，可以用Gitosis来管理公钥。

这里我们不介绍怎么玩Gitosis了，几百号人的团队基本都在500强了，相信找个高水平的Linux管理员问题不大。

管理权限

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

这里我们也不介绍Gitolite了，不要把有限的生命浪费到权限斗争中。

小结

- 搭建Git服务器非常简单，通常10分钟即可完成；
- 要方便管理公钥，用Gitosis；
- 要像SVN那样变态地控制权限，用Gitolite。

第10章 期末总结

终于到了期末总结的时刻了！

经过几天的学习，相信你对Git已经初步掌握。一开始，可能觉得Git上手比较困难，尤其是已经熟悉SVN的童鞋，没关系，多操练几次，就会越用越顺手。

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

友情附赠国外网友制作的Git Cheat Sheet，建议打印出来备用：

[Git Cheat Sheet](#)

[github-cheat-sheet GitHub秘籍](#)

现在告诉你Git的官方网站：<http://git-scm.com>，英文自我感觉不错的童鞋，可以经常去官网看看。什么，打不开网站？相信我，我给出的绝对是官网地址，而且，Git官网决没有那么容易宕机，可能是你的人品问题，赶紧面壁思过，好好想想原因。

如果你学了Git后，工作效率大增，有更多的空闲时间健身看电影，那我的教学目标就达到了。

谢谢观看！

序言

我自己接触 GitHub 较早，可以说在 GitHub 在国内还没怎么普及、流行的时候就接触了，之后对我的工作以及开放的思维方式产生的很大的影响，也大大提升了自己的开发效率与个人能力。从第一个使用的第三方库，到自己的第一篇博客，再到后面自己的第一个开源项目，都享受着 GitHub 带来的好处。

后面渐渐的自己也热衷于分享，拥抱开源，从博客，到公众号都在坚持写文章，分享自己过来人的技术积累、职场经验、人生记录等，后面也慢慢的有点影响力了，直到有一天我突然发现，关注我公众号的读者们很多竟然没听说过 GitHub，或者即使听说过也没怎么使用过，这真的是巨大的一个损失啊，于是，应读者要求，我准备自己从 0 开始，写一篇针对初学者的 GitHub 教程，没想到，利用自己业余时间，持续了几个月，竟然形成了一个系列，评价也相当不错。借这个机会，把这个教程整理出来，造福更多的人。

这个教程包括如下内容：

1. 初识 GitHub
2. 加入 GitHub
3. Git 速成
4. 向 GitHub 提交代码
5. Git 进阶
6. 团队合作利器：Git 分支详解
7. GitHub 常见的几种操作
8. 发现好用的开源项目

希望通过这个教程，人人可以很方便的掌握 Git/GitHub 的使用。

最后，你还可以通过以下其他方式找到我：

GitHub：<https://github.com/stormzhang>

个人博客：<http://stormzhang.com>

如果想获取其他更多原创分享，欢迎关注我的微信公众号 AndroidDeveloper。

从0开始学习 GitHub 系列之「01.初识 GitHub」

1. 写在前面

我一直认为 GitHub 是程序员必备技能，程序员应该没有不知道 GitHub 的才对，没想到这两天留言里给我留言最多的就是想让我写关于 GitHub 的教程，说看了不少资料还是一头雾水，我转念一想，我当初接触 GitHub 也大概工作了一年多才开始学习使用，我读者里很多是初学者，而且还有很多是在校大学生，所以不会用 GitHub 也就不奇怪了，所以我觉得写一写关于 GitHub 的教程就非常有必要了！

2. 为什么还要造轮子

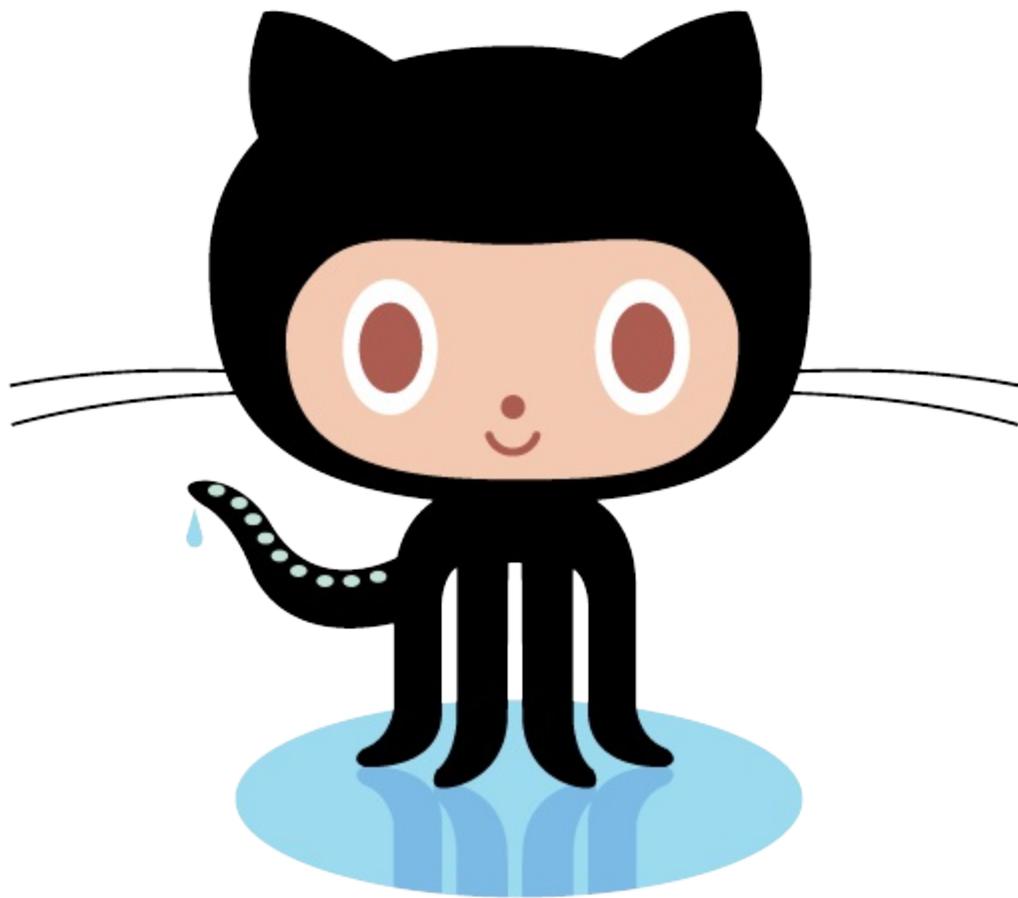
很多人难免要问这个问题，说网上关于 GitHub 的资料很多，为什么还要写呢？讲真，网上关于 [Android](#) 的资料更多，为什么你们还喜欢看我写的文章呢？是因为哪怕同样的内容，我写出来之后就有了我的风格，除了我的幽默以及我的帅，关键的是我有办法让你们看的轻松易懂，并且还有我个人的一些见解与指导，这大概是一种特殊的魅力吧！

我是从小白一路过来的，很能理解你们内心的感受与困惑，因为这些阶段都是我自己亲身经历过的，所以我写的文章都会从你们的角度去出发，并且我对文章高要求，除了排版、配图很用心外，文章的内容每次写完我都会亲自看三四遍，确保不会出现误导以及你们理解不了的情况，你们看的很轻松易懂的文章其实因为我背后做了很多的功课。

所以，为了你们，我觉得有必要用我的风格去教你们如何从0开始，跟着我一步步学习 GitHub！

3. 什么是 GitHub

确切的说 GitHub 是一家公司，位于旧金山，由 Chris Wanstrath, PJ Hyett 与 Tom Preston-Werner 三位开发者在 2008 年 4 月创办。这是它的 Logo：



2008年4月10日，GitHub正式成立，地址：[How people build software · GitHub](#)，主要提供基于[Git](#)的版本托管服务。一经上线，它的发展速度惊为天人，截止目前，GitHub 已经发展成全球最大的开（同）源（性）社区。

4. GitHub 与 Git 的关系

这个我还专门在群里调查过，很多人以为 GitHub 就是 Git，其实这是一个理解误区。

Git 是一款免费、开源的分布式[版本控制](#)系统，他是著名的 [Linux](#) 发明者 Linus Torvalds 开发的。说到版本控制系统，估计很多人都用过 SVN，只不过 Git 是新时代的产物，如果你还在用 SVN 来管理你的代码，那就真的有些落伍了。不管是学习 GitHub，还是以后想从事编程行业，Git 都可以算是必备技能了，所以从现在开始建议你先去学习熟悉下 Git，后面我会有关于一些适合新手的 Git 学习资料给你们。

而 GitHub 上面说了，主要提供基于 git 的版本托管服务。也就是说现在 GitHub 上托管的所有项目代码都是基于 Git 来进行版本控制的，所以 Git 只是 GitHub 上用来管理项目的一个工具而已，GitHub 的功能可远不止于此！

5. GitHub 的影响力

上面我说了 GitHub 现在毫无疑问基本是全球最大的开源社区了，这样说你们可能认为未免有点浮夸，且听我一一举证：

全球顶级科技公司纷纷加入 GitHub，并贡献他们自己的项目代码

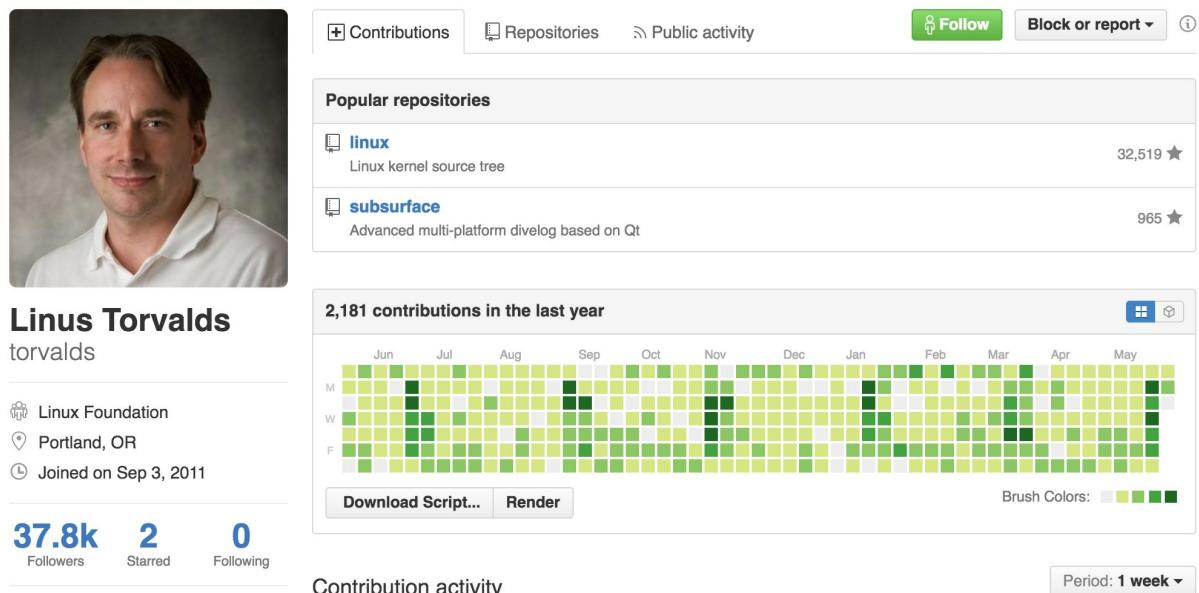
- Google: <https://github.com/google>
- 苹果: <https://github.com/apple>
- Facebook: <https://github.com/facebook>
- Twitter : <https://github.com/twitter>
- 微软 : <https://github.com/microsoft>
- Square : <https://github.com/square>
- 阿里 : <https://github.com/alibaba>
- ...

全球顶级开源项目都优先选择在 GitHub 上开源

- Linux : <https://github.com/torvalds/linux>
- Rails : <https://github.com/rails/rails>
- Nodejs : <https://github.com/nodejs/node>
- Swift : <https://github.com/apple/swift>
- CoffeeScript : <https://github.com/jashkenas/coffeescript>
- Ruby : <https://github.com/ruby/ruby>
- ...

全球顶级编程大牛加入GitHub

- Linux 发明者 Linus Torvalds : <https://github.com/torvalds>



- Rails 创始人 DHH : <https://github.com/dhh>



David Heinemeier Hansson
dhh

Basecamp
Chicago, USA
david@basecamp.com
<http://david.heinemeierhansson.org>
Joined on Mar 11, 2008

8.2k Followers **28** Starred **0** Following

- [Contributions](#)
- [Repositories](#)
- [Public activity](#)
- [Follow](#)
- [Block or report](#)

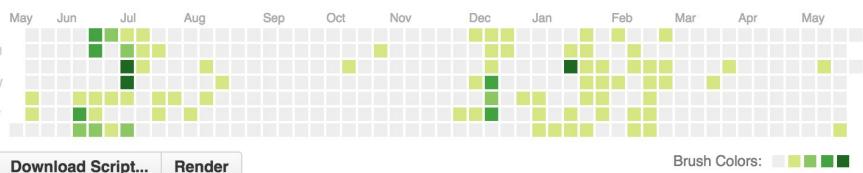
Popular repositories

 tolk	514 ★
Tolk is a web interface for doing i18n translation...	
 custom_configuration	204 ★
Custom configuration storage for Rails	
 asset-hosting-with-minimum-ssl	78 ★
Rails plugin for picking a non-ssl asset host as...	
 conductor	65 ★
 delayed_job	32 ★
Database based asynchronously priority queue...	

Repositories contributed to

 rails/rails	31,271 ★
Ruby on Rails	
 rails/actionable	1,120 ★
Framework for real-time communication over w...	
 rails/homepage	16 ★
rubyonrails.org site anno 2015	
 rails/weblog	18 ★
 rails/actionable-examples	303 ★
Action Cable Examples	

585 contributions in the last year



Download Script... Render

Brush Colors: ■ ■ ■

Organizations

- 被称为「Android之神」的 JakeWharton：<https://github.com/JakeWharton>，你们用的很多开源库如 ButterKnife、OkHttp、Retrofit、Picasso、ViewPagerIndicator 等都是出自他之手！



Jake Wharton
JakeWharton

Square, Inc.
Pittsburgh, PA, USA
jakewharton@gmail.com
<http://jakewharton.com>
Joined on Mar 25, 2009

21.8k Followers **198** Starred **13** Following

- [Contributions](#)
- [Repositories](#)
- [Public activity](#)
- [Unfollow](#)
- [Block or report](#)

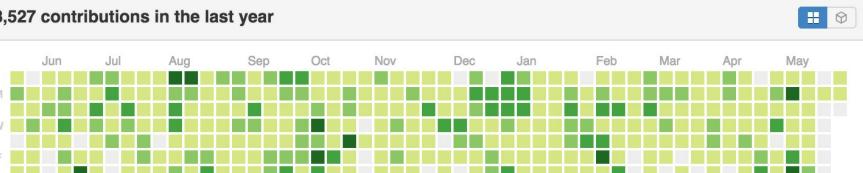
Popular repositories

 butterknife	9,271 ★
Bind Android views and callbacks to fields and...	
 ViewPagerIndicator	7,482 ★
Paging indicator widgets compatible with the V...	
 ActionBarSherlock	7,277 ★
[DEPRECATED] Action bar implementation wh...	
 u2020	3,703 ★
A sample Android app which showcases adva...	
 NineOldAndroids	3,504 ★
[DEPRECATED] Android library for using the H...	

Repositories contributed to

 square/retrofit	12,641 ★
Type-safe HTTP client for Android and Java by...	
 square/wire	1,482 ★
Clean, lightweight protocol buffers for Android ...	
 square/sqlDelight	665 ★
Generates Java models from CREATE TABLE...	
 square/okhttp	11,429 ★
An HTTP+HTTP/2 client for Android and Java ...	
 square/sqlBrite	2,591 ★
A lightweight wrapper around SQLiteOpenHelper...	

3,527 contributions in the last year



Download Script... Render

Brush Colors: ■ ■ ■

其他就不一一列举了，GitHub 上活跃的很多是 Google、Square、阿里等公司的员工，有些甚至是 Google Android Team 组的，所以在这里你可以接触到全球顶级编程大牛！

6. GitHub 有什么用

- 学习优秀的开源项目

开源社区一直有一句流行的话叫「不要重复发明轮子」，某种意义上正是因为开源社区的贡献，我们的软件开发才能变得越来越容易，越来越快速。试想你在做项目时，如果每一模块都要自己去写，如网络库、图片加载库、ORM库等等，自己写的好不好是一回事，时间与资源是很大的成本。对于大公司可能会有人力与资源去发明一套自己的轮子，但是对于大部分互联网创业公司来说时间就是一切。而且你在使用开源项目的过程也可以学习他们优秀的设计思想、实现方式，这是最好的学习资料，也是一份提升自己能力的绝佳方式！

- 多人协作

如果你想发起一个项目，比如翻译一份不错的英文文档，觉得一个人的精力不够，所以你需要更多的人参与进来，这时候 GitHub 是你的最佳选择，感兴趣的人可以参与进来，利用业余时间对这个项目做贡献，然后可以互相审核、合并，简直不要太棒！

- 搭建博客、个人网站或者公司官网

这个就不用多说了，现在越来越多的博客都是基于 GitHub Pages 来搭建的了，你可以随心所欲的定制自己的样式，可以给你博客买个逼格高的域名，再也不用忍受各大博客网站的约束与各式各样的广告了！

- 写作

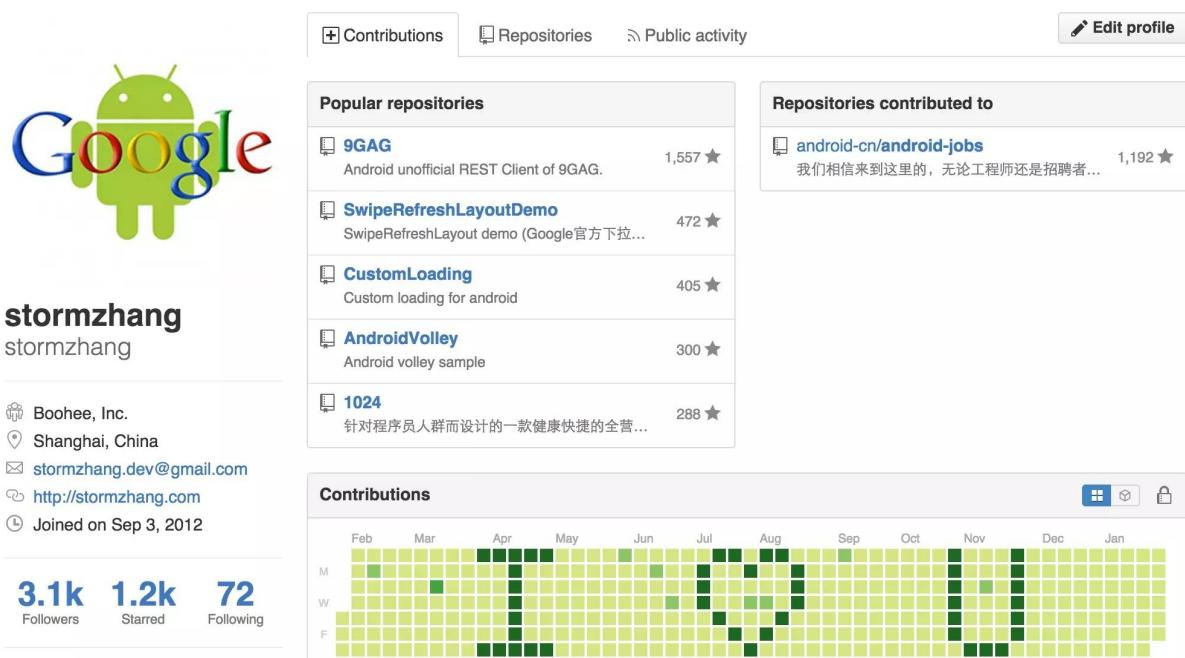
如果你喜欢写作，而且基于 Markdown，并准备出版书籍，那么推荐你用 Gitbook，技术写作人的最爱！

- 个人简历

如果你有一个活跃的 GitHub 账号，上面有自己不错的开源项目，还经常给别的开源项目提问题，push 代码，那么你找工作将是一个非常大的优势，现在程序员的招聘很多公司都很看中你 GitHub 账号，某种意义上 GitHub 就可以算是你的简历了。而且不仅国内，很多国外的科技公司都会通过 GitHub 来寻找优秀的人才，比如我甚至通过 GitHub 收到过 Facebook 的邀请邮件！

- 其他

当然 GitHub 能做的远不止这些，我见过很多在 GitHub 搞的一些有意思的项目，有找男朋友的，甚至还有利用 GitHub 的 commit 丧心病狂的秀恩爱的，没错，那个丧心病狂的人就是我，如果你前段时间关注了我的 GitHub，那么能看到这么一个壮观的景象：



7. 加入 GitHub

读完我的文章，我相信你已经蠢蠢欲动了，从现在开始，立刻、马上去注册个 GitHub 「<https://github.com/>」，去体验一番，不会用不要紧，接下来我会有一系列详细的文章，来教你学会使用 GitHub！

但是为了保证文章的质量，我要做很多准备工作，我没法保证每天都会连载，但是我会尽力尽快更新这个系列，让你们从0开始一步步一起来学习，如果周围有同学或者朋友想要学习的，那赶紧转发或者推荐他关注这个系列的文章，毕竟有个小伙伴一起学会更有氛围，后续除了理论我还会考虑结合实践，我不信你学不会！

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处。

从0开始学习 GitHub 系列之「02.加入 GitHub」

看完昨天的文章「[从0开始学习 GitHub 系列之「初识 GitHub」](#)」估计不少人已经开始期待我继续更新了，这不赶紧马不停蹄，加班加点给你们更新了第二篇。在更新本篇文章之前先回答昨天大家留言的两个问题：

- GitHub 需要翻墙么？

印象中 GitHub 之前确实总是断断续续的访问不了，不过在13年初的时候有段时间最严重，一度被封了，当时李开复老师再也忍无可忍，公开发了一条抗议 GitHub 被封的微博，这事我印象很深，因为我是12年底加入的 GitHub，那时候简直像遇到世外桃源一般，但是也深受老是访问不了的困扰，很多人早就对这件事怨声载道了，加上李开复老师的声讨，这一下就炸开了锅，微博上纷纷转发谴责，算的上是整个IT界的大新闻，后来因为这事影响太大了，没过几天 GitHub 就可以正常访问了，这里真的要感谢李开复老师敢于站出来的勇气，可以这么说，如果没有 GitHub，中国的编程水平起码要倒退好多年！

因为 GitHub 的影响力太大，基本上是各种黑客攻击的对象，所以现在偶尔也会有宕机访问不了的情况，但是好在不会被封，所以大家不用担心，访问 GitHub 不用翻墙，只是可能访问速度稍慢些，另外为了维护一个和谐的环境，这里也呼吁大家不要在 GitHub 上发表任何关于政治的言论与文章，在 GitHub 上我们只是单纯的技术交流，无关政治，在复杂的大环境下，希望 GitHub 永远是我们程序员的一片净土！

- 英语差、0基础学得会么？

这个也是不少人问我的，GitHub 虽然都是英文，但是对英语水平的要求不是那么高，都是些简单的单词，遇到不会的查一下就行了，你觉得很难只是你对英文网站反射性的抵触而已，相信我，跟着我的详细教程，我的文章面向从没有接触过甚至没有听过 GitHub 的同学，一步步教你由浅入深。如果你学不会，那么来打我，不过我这么帅，你也不忍心！

好了，废话不多说，咱们进入正文！

1. 注册 GitHub

先去 GitHub 官网「[How people build software · GitHub](#)」注册「Sign Up」个账号，注册页面如下：

Join GitHub

The best way to design, build, and ship software.



Step 1:
Set up a personal account



Step 2:
Choose your plan



Step 3:
Go to your dashboard

Create your personal account

Username

This will be your username — you can enter your organization's username next.

Email Address

You will occasionally receive account related emails. We promise not to share your email with anyone.

Password

Use at least one lowercase letter, one numeral, and seven characters.

By clicking on "Create an account" below, you are agreeing to the [Terms of Service](#) and the [Privacy Policy](#).

Create an account

这个应该没啥说的，需要填用户名、邮箱、密码，值得一提的用户名请不要那么随便，最好取的这个名字就是你以后常用的用户名了，也强烈建议你各大社交账号都用一样的用户名，这样识别度较高，比如我的博客域名、GitHub、知乎等其他社交账号 ID 都是 stormzhang，微博是因为被占用了，无奈换了个id，而且这个用户名以后在 GitHub 搭建博客的时候默认给你生成的博客地址就是 <http://username.github.io>，所以给自己取个好点的用户名吧。

填好用户名、邮箱、密码紧接着到这一步：

You'll love GitHub

Unlimited collaborators

Unlimited public repositories

- ✓ Great communication
- ✓ Friction-less development
- ✓ Open source community

Welcome to GitHub

You've taken your first step into a larger world, @googdev.

<p> Completed Set up a personal account</p>	<p> Step 2: Choose your plan</p>	<p> Step 3: Go to your dashboard</p>
---	--	---

Choose your personal plan

Unlimited public repositories for free.

Unlimited private repositories for \$7/month. ([view in CNY](#))

Don't worry, you can cancel or upgrade at any time.

Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
[Learn more about organizations.](#)

Both plans include:

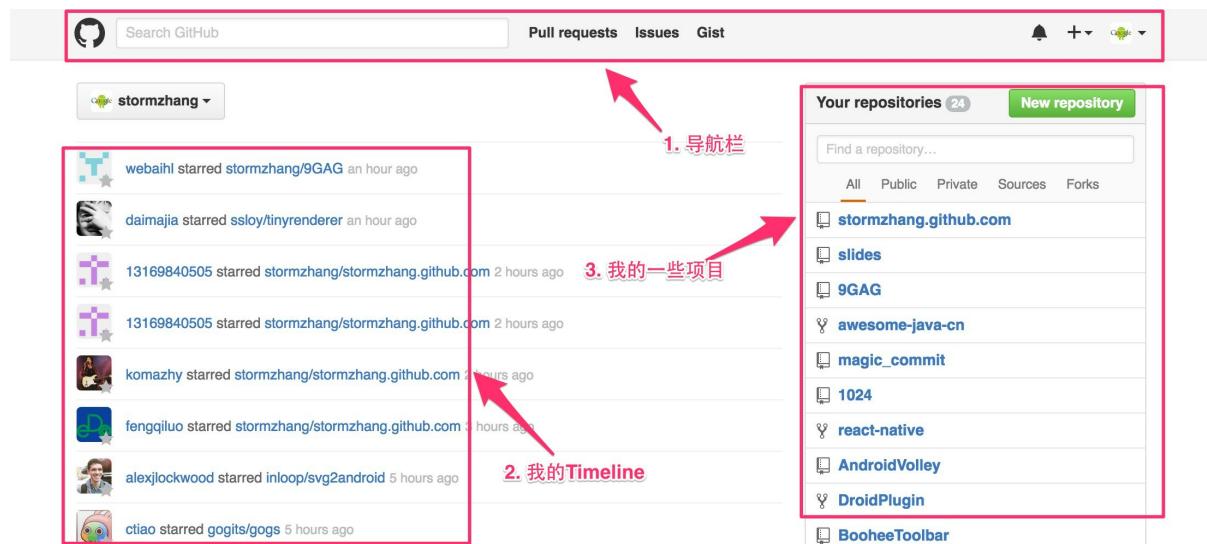
- ✓ Collaborative code review
- ✓ Issue tracking
- ✓ Open source community
- ✓ Unlimited public repositories
- ✓ Join any organization

Finish sign up

这个是什么意思呢？GitHub 有两种，一种是公开，这种是免费的，就是你创建的项目是开放的，所有人都能看到；另一种是私有，这种是收费的，这种一般是很多企业在使用 GitHub 的私有仓库在托管自己的项目，这也是 GitHub 的一种盈利模式对于个人你就直接默认选择公开的就行了。

2. 认识 GitHub

注册成功之后你会到 GitHub 的主页面来：



The screenshot shows the GitHub homepage with several annotated sections:

- 1. 导航栏**: Points to the top navigation bar which includes the GitHub logo, a search bar, and links for Pull requests, Issues, and Gist.
- 2. 我的Timeline**: Points to the left sidebar where a list of recent starred projects is displayed.
- 3. 我的一些项目**: Points to the main repository list on the right side, which shows the user's public repositories.

The repository list on the right includes:

- Your repositories 24
- New repository
- Find a repository...
- All Public Private Sources Forks
- stormzhang.github.com
- slides
- 9GAG
- awesome-java-cn
- magic_commit
- 1024
- react-native
- AndroidVolley
- DroidPlugin
- BooheeToolbar

你如果是新注册的可能看到的跟我不一样，因为你们新用户，没有自己的项目，没有关注的人，所以只有一个导航栏。

导航栏，从左到右依次是 GitHub 主页按钮、搜索框、PR、Issues、Gist（这些概念后面会讲的）、消息提醒、创建项目按钮、我的账号相关。

我的 Timeline，这部分你可以理解成微博，就是你关注的一些人的活动会出现在这里，比如如果你们关注我了，那么以后我 star、fork 了某些项目就会出现在你的时间线里。

我的项目，这部分就不用说了，如果你创建了项目，就里就可以快捷访问。

3. GitHub 主页

点击下图的 Your profile 菜单进入到你的个人 GitHub 主页。

The screenshot shows the GitHub homepage for the user 'stormzhang'. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Gist', and a dropdown menu. Below the navigation is a sidebar with a profile picture and the user's name 'stormzhang'. The main area displays a timeline of recent activity, including stars from other users like 'webaihl' and 'daimajia'. To the right is a list of the user's repositories, with '9GAG' being the most prominent. A red box highlights the 'Your profile' link in the sidebar.

还是以我的 GitHub 主页为例：

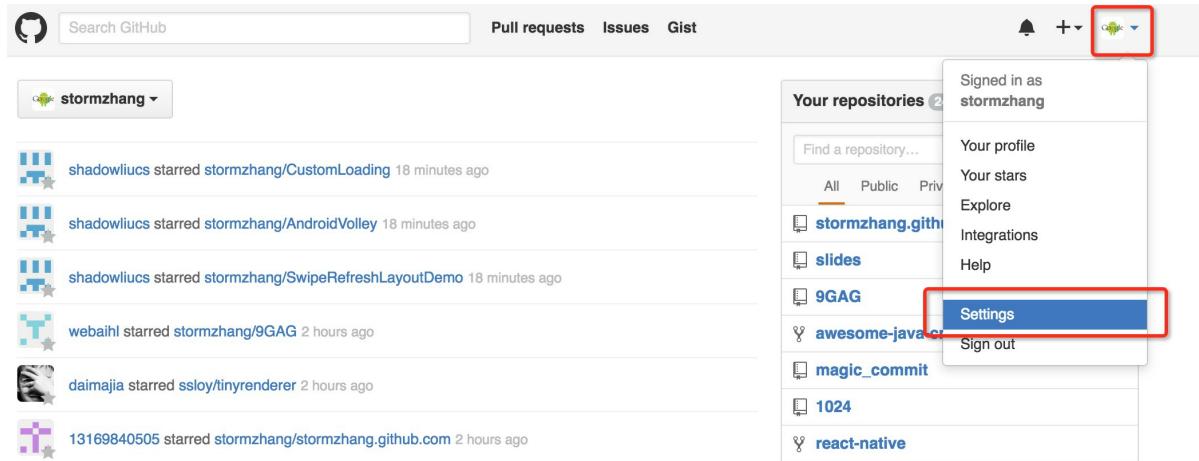
This screenshot shows the author's GitHub profile page with various sections annotated:

- Popular repositories**: Points to a list of repositories like '9GAG', 'SwipeRefreshLayoutDemo', 'CustomLoading', 'AndroidVolley', and '1024'. The '9GAG' repository has 1,557 stars.
- Repositories contributed to**: Points to a section for 'android-cn/android-jobs' with 1,192 stars.
- 我在 GitHub 上提交代码的记录**: Points to a 'Contributions' grid showing activity over time.
- 关注我的人** and **我关注的人**: Points to the 'Followers', 'Starred', and 'Following' counts at the bottom left.
- 我送出的 star 数**: Points to the total stars given by the user.
- 我在 GitHub 上提交代码的记录**: Points to the 'Contributions' grid.

这么详细应该不会看不懂吧？只不过你的账号可能没有这么丰富，因为你可能啥也没做过，但是如果做全了基本上就会看到跟我一样的了。

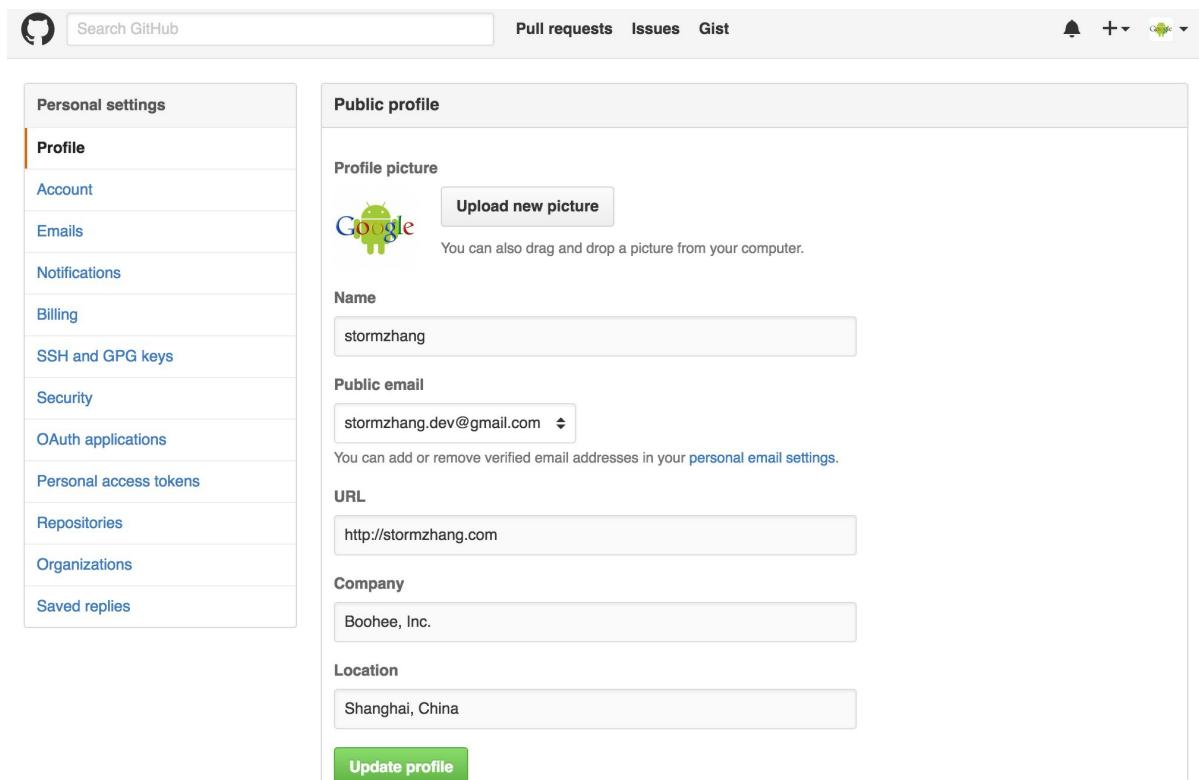
4. 设置你的 GitHub

如果你是新注册的 GitHub 账号，是不是觉得很简陋？虽然你没有自己的项目，但是第一步起码要先完善自己的信息，点击如下的 Settings 菜单：



The screenshot shows the GitHub homepage for the user 'stormzhang'. On the left, there's a sidebar with recent activity: 'shadowliucs starred stormzhang/CustomLoading 18 minutes ago', 'shadowliucs starred stormzhang/AndroidVolley 18 minutes ago', 'shadowliucs starred stormzhang/SwipeRefreshLayoutDemo 18 minutes ago', 'webahl starred stormzhang/9GAG 2 hours ago', 'daimajia starred ssloy/tinyrenderer 2 hours ago', and '13169840505 starred stormzhang/stormzhang.github.com 2 hours ago'. At the top right, there's a dropdown menu for the user 'stormzhang'. This menu includes options like 'Your repositories', 'Find a repository...', 'All', 'Public', 'Private', 'stormzhang.github.com', 'slides', '9GAG', 'awesome-javacode', 'magic_commit', '1024', 'react-native', 'Settings' (which is highlighted with a red box), and 'Sign out'. The 'Settings' option is the one we're focusing on.

到设置页面来设置一些基本信息：



The screenshot shows the 'Personal settings' page for the 'Profile' section. On the left, there's a sidebar with various options: Account, Emails, Notifications, Billing, SSH and GPG keys, Security, OAuth applications, Personal access tokens, Repositories, Organizations, and Saved replies. The 'Profile' section is currently selected. On the right, there's a 'Public profile' section. It includes fields for 'Profile picture' (with a placeholder for 'Upload new picture' and a note about dragging and dropping), 'Name' (set to 'stormzhang'), 'Public email' (set to 'stormzhang.dev@gmail.com'), 'URL' (set to 'http://stormzhang.com'), 'Company' (set to 'Boohee, Inc.'), and 'Location' (set to 'Shanghai, China'). At the bottom right of this section is a green 'Update profile' button.

像头像、Name 建议要设置一个常用的，这两个很有识别性，公开的邮箱也要设置一个，这样那些企业啊、猎头啊就通过这个公开邮箱去联系你，友情提醒：别在 GitHub 把自己的 QQ 邮箱放上去，不显得太 low 了么？没有 gmail 邮箱，起码也得注册个 foxmail、163 邮箱之类的吧。

5. GitHub 基本概念

上面认识了 GitHub 的基本面貌之后，你需要了解一些 GitHub 的基本概念，这些概念是你经常会接触并遇到的。

- Repository

仓库的意思，即你的项目，你想在 GitHub 上开源一个项目，那就必须要新建一个 Repository，如果你开源的项目多了，你就拥有了多个 Repositories。

- Issue

问题的意思，举个例子，就是你开源了一个项目，别人发现你的项目中有bug，或者哪些地方做的不够好，他就能够给你提个 Issue，即问题，提的问题多了，也就是 Issues，然后你看到了这些问题就可以去逐个修复，修复ok了就可以一个个的 Close 掉。

- Star

这个好理解，就是给项目点赞，但是在 GitHub 上的点赞远比微博、知乎点赞难的多，如果你有一个项目获得100个 star都算很不容易了！

- Fork

这个不好翻译，如果实在要翻译我把他翻译成分叉，什么意思呢？你开源了一个项目，别人想在你这个项目的基础上做些改进，然后应用到自己的项目中，这个时候他就可以 Fork 你的项目，这个时候他的 GitHub 主页上就多了一个项目，只不过这个项目是基于你的项目基础（本质上是在原有项目的基础上新建了一个分支，分支的概念后面会在讲解[Git](#)的时候说到），他就可以随心所欲的去改进，但是丝毫不会影响原有项目的代码与结构。

- Pull Request

发起请求，这个其实是基于 Fork 的，还是上面那个例子，如果别人在你基础上做了改进，后来觉得改进的很不错，应该要把这些改进让更多的人收益，于是就想把自己的改进合并到原有项目里，这个时候他就可以发起一个 Pull Request（简称PR），原有项目创建人就可以收到这个请求，这个时候他会仔细review你的代码，并且[测试](#)觉得OK了，就会接受你的PR，这个时候你做的改进原有项目就会拥有了。

- Watch

这个也好理解就是观察，如果你 Watch 了某个项目，那么以后只要这个项目有任何更新，你都会第一时间收到关于这个项目的通知提醒。

- Gist

有些时候你没有项目可以开源，只是单纯的想分享一些代码片段，那这个时候 Gist 就派上用场了！

6. 创建自己的项目

点击顶部导航栏的 + 可以快速创建一个项目，如下图：

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: stormzhang

Repository name: test

Description (optional): test for AndroidDeveloper users.

Visibility: Public (Anyone can see this repository. You choose who can commit.)

Initialize this repository with a README (checked): This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

Create repository

创建一个项目需要填写如上的几部分：项目名、项目描述与简单的介绍，你不付费没法选择私有的，所以接着只能选择 public 的，之后勾选「Initialize this repository with a README」，这样你就拥有了你的第一个 GitHub 项目：

stormzhang / test

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

test for AndroidDeveloper users. — Edit

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

stormzhang Initial commit README.md Initial commit README.md

Latest commit ce8c73e just now

test

test for AndroidDeveloper users.

可以看到这个项目只包含了一个 README.md 文件，但是它已经是一个完整的 Git 仓库了，你可以通过对它进行一些操作，如watch、star、fork，还可以 clone 或者下载下来。

这里提一下 README.md，GitHub 上所有关于项目的详细介绍以及 Wiki 都是基于 Markdown 的，甚至之后在 GitHub 上搭建博客，写博客也是如此，所以如果还不懂 Markdown 语法的，建议先去学习下。推荐一篇学习 Markdown 的文章给你们：

[献给写作者的 Markdown 新手指南](#)

7. 总结

相信看完以上文章你已经基本算是了解 GitHub 的基本概念并且正式加入 GitHub 这个大家庭了，之后会有更深入的文章介绍 Git、介绍对项目的常用操作、介绍如何给开源项目提交代码、介绍如何协同合作甚至怎么搭建博客等，敬请期待吧！

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「03.Git 速成」

前面的 GitHub 系列文章介绍过，GitHub 是基于 [Git](#) 的，所以也就意味着 Git 是基础，如果你不会 Git，那么接下来你完全继续不下去，所以今天的教程就来说说 Git，当然关于 Git 的知识单凭一篇文章肯定说不完的，我这篇文章先介绍一些最基本的、最常用的一些 Git 知识，争取让你们 Git 速成。

1. 什么是Git？

Git 是 [Linux](#) 发明者 Linus 开发的一款新时代的[版本控制](#)系统，那什么是版本控制系统呢？怎么理解？网上一大堆详细的介绍，但是大多枯燥乏味，对于新手也很难理解，这里我只举几个例子来帮助你们理解。

熟悉编程的知道，我们在软件开发中源代码其实是最重要的，那么对源代码的管理变得异常重要：

比如为了防止代码的丢失，肯定本地机器与远程服务器都要存放一份，而且还需要有一套机制让本地可以跟远程同步；

又比如我们经常是好几个人做同一个项目，都要对一份代码做更改，这个时候需要大家互不影响，又需要各自可以同步别人的代码；

又比如我们开发的时候免不了有bug，有时候刚发布的功能就出现了严重的bug，这个时候需要紧急对代码进行还原；

又比如随着我们版本迭代的功能越来越多，但是我们需要清楚的知道历史每一个版本的代码更改记录，甚至知道每个人历史提交代码的情况；

等等等类似以上的情况，这些都是版本控制系统能解决的问题。所以说，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，对于软件开发领域来说版本控制是最重要的一环，而 Git 毫无疑问是当下最流行、最好用的版本控制系统。

2. Git 安装

上面说了，Git 是一个版本控制系统，你也可以理解成是一个工具，跟 [Java](#) 类似，使用之前必须得先下载安装，所以第一步必须要安装，我用的是 Mac，Mac 上其实系统自带 Git 的，不过这里统一提供一下各平台的安装方式，这部分就不过多介绍，相信大家这里搞的定。

- Mac : <https://sourceforge.net/projects/git-osx-installer/>
- Windows : <https://git-for-windows.github.io/>
- Linux : apt-get install git

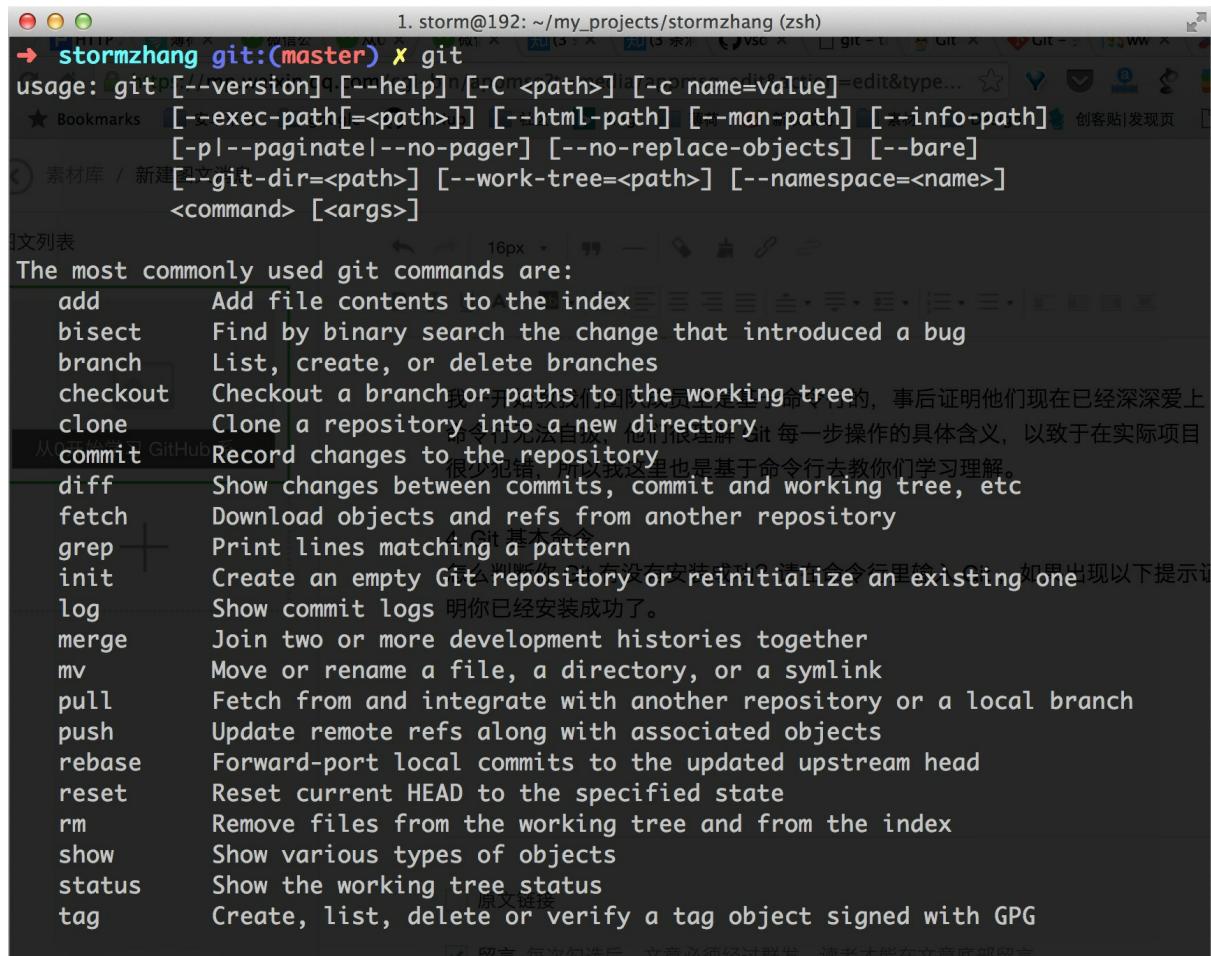
3. 如何学习 Git？

安装好 Git 之后，怎么学习是个问题，其实关于 Git 有很多图形化的软件可以操作，但是我强烈建议大家从命令行开始学习理解，我知道没接触过命令行的人可能会很抵触，但是我的亲身实践证明，只有一开始学习命令行，之后你对 Git 的每一步操作才能理解其意义，而等你熟练之后你想用任何的图形化的软件去操作完全没问题。

我一开始教我们团队成员全是基于命令行的，事后证明他们现在已经深深爱上命令行无法自拔，他们很理解 Git 每一步操作的具体含义，以致于在实际项目很少犯错，所以我这里也是基于命令行去教你们学习理解。

4. Git 命令列表

怎么判断你 Git 有没有安装成功？请在命令行里输入 git，如果出现以下提示证明你已经安装成功了。



```
1. storm@192: ~/my_projects/stormzhang (zsh)
→ stormzhang git:(master) x git
usage: git [--version] [--help] [-C <path>] [-c name=value] <command> [<args>]
★ Bookmarks [-exec-path[=<path>]] [-html-path] [-man-path] [-info-path] [创客贴|发现页]
[-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
[--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
<command> [<args>]

The most commonly used git commands are:
add      Add file contents to the index
bisect   Find by binary search the change that introduced a bug
branch   List, create, or delete branches
checkout  Checkout a branch or paths to the working tree
clone    Clone a repository into a new directory
commit   Record changes to the repository
diff     Show changes between commits, commit and working tree, etc
fetch   Download objects and refs from another repository
grep    Print lines matching a pattern
init    Create an empty Git repository or reinitialize an existing one
log     Show commit logs
merge   Join two or more development histories together
mv      Move or rename a file, a directory, or a symlink
pull   Fetch from and integrate with another repository or a local branch
push    Update remote refs along with associated objects
rebase  Forward-port local commits to the updated upstream head
reset   Reset current HEAD to the specified state
rm      Remove files from the working tree and from the index
show    Show various types of objects
status  Show the working tree status
tag     Create, list, delete or verify a tag object signed with GPG
```

Git 所有的操作命令开头都要以 git 开头，上面列举了最常用的一些 Git 命令，紧接着会有一句英文解释这个命令的意义，都不是很难的单词，不妨试着看一下，不过没有实际操作你仍然不好理解，下面我们来以一个实际的操作来介绍下一些常用命令的含义。

5. Git 具体命令

第一步，我们先新建一个文件夹，在文件夹里新建一个文件（我是用 Linux 命令去新建的，Windows 用户可以自己手动新建）

```
mkdir test (创建文件夹test) cd test (切换到test目录) touch a.md (新建a.md文件)
```

这里提醒下：在进行任何 Git 操作之前，都要先切换到 Git 仓库目录，也就是先要先切换到项目的文件夹目录下。

这个时候我们先随便操作一个命令，比如 git status，可以看到如下提示（别纠结颜色之类的，配置与主题不一样而已）：



```
→ test git status
fatal: Not a git repository (or any of the parent directories): .git
→ test
```

意思就是当前目录还不是一个 Git 仓库。

git init

这个时候用到了第一个命令，代表初始化 git 仓库，输入 git init 之后会提示：

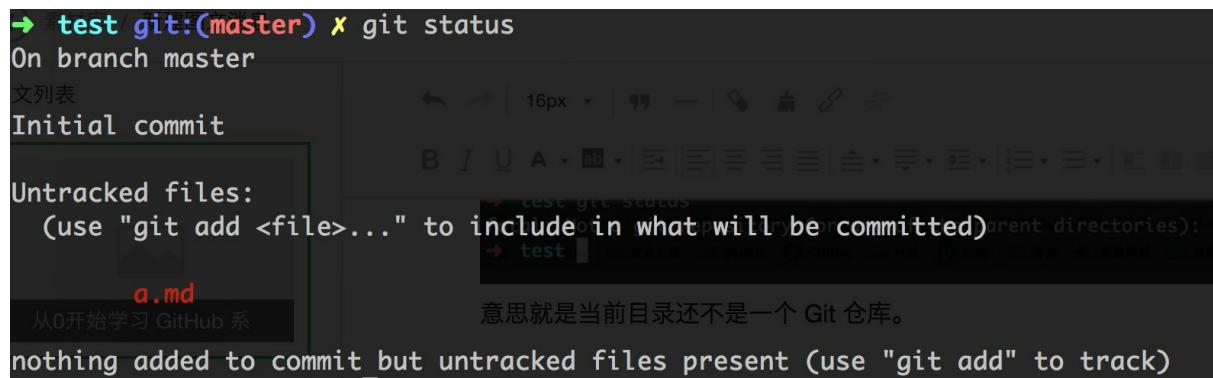


```
→ test git init
Initialized empty Git repository in /Users/storm/test/.git/
→ test git:(master) x
```

可以看到初始化成了，至此 test 目录已经是一个 git 仓库了。

git status

紧接着我们输入 git status 命令，会有如下提示：



```
→ test git:(master) x git status
On branch master
文列表
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    a.md
从0开始学习 GitHub 系
nothing added to commit but untracked files present (use "git add" to track)
```

默认就直接在 master 分支，关于分支的概念后面会提，这时最主要的是提示 a.md 文件 Untracked files，就是说 a.md 这个文件还没有被跟踪，还没有提交在 git 仓库里呢，而且提示你可以使用 git add 去操作你想要提交的文件。

git status 这个命令顾名思义就是查看状态，这个命令可以算是使用最频繁的一个命令了，建议大家没事就输入下这个命令，来查看你当前 git 仓库的一些状态。

git add

上面提示 a.md 文件还没有提交到 git 仓库里，这个时候我们可以随便编辑下 a.md 文件，然后输入 git add a.md，然后再输入 git status：



```
→ test git:(master) x git add a.md
→ test git:(master) x git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   a.md
```

此时提示以下文件 Changes to be committed， 意思就是 a.md 文件等待被提交，当然你可以使用 git rm –cached 这个命令去移除这个缓存。

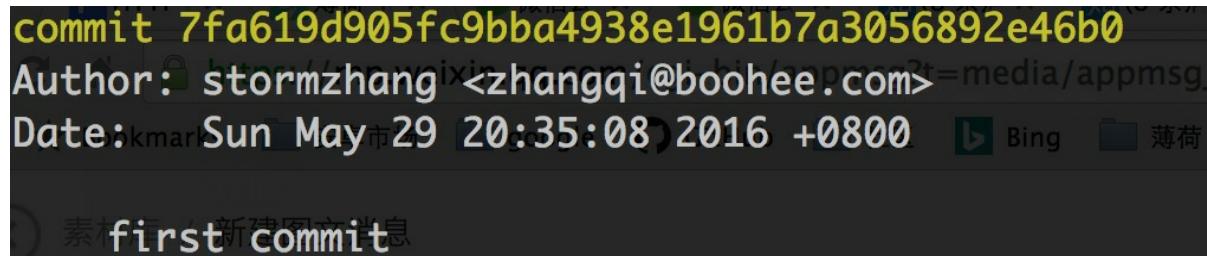
git commit

接着我们输入 git commit -m ‘first commit’ ，这个命令什么意思呢？commit 是提交的意思， -m 代表是提交信息，执行了以上命令代表我们已经正式进行了第一次提交。

这个时候再输入 git status，会提示 nothing to commit。

git log

这个时候我们输入 git log 命令，会看到如下：



```
commit 7fa619d905fc9bba4938e1961b7a3056892e46b0
Author: stormzhang <zhangqi@boohoo.com>
Date:   Sun May 29 20:35:08 2016 +0800

素木 first commit
```

git log 命令可以查看所有产生的 commit 记录，所以可以看到已经产生了一条 commit 记录，而提交时候的附带信息叫 ‘first commit’ 。

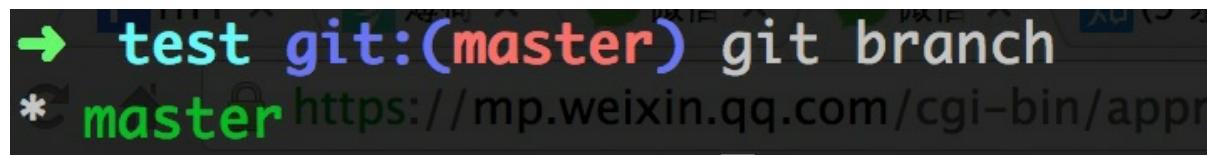
git add & git commit

看到这里估计很多人会有疑问，我想要提交直接进行 commit 不就行了么，为什么先要再 add 一次呢？首先 git add 是先把改动添加到一个「暂存区」，你可以理解成是一个缓存区域，临时保存你的改动，而 git commit 才是最后的真正的提交。这样做好处就是防止误提交，当然也有办法把这两步合并成一步，不过后面再介绍，建议新手先按部就班的一步步来。

git branch

branch 即分支的意思，分支的概念很重要，尤其是团队协作的时候，假设两个人都在做同一个项目，这个时候分支就是保证两人能协同合作的最大利器了。举个例子，A, B俩人都在做同一个项目，但是不同的模块，这个时候A新建了一个分支叫a，B新建了一个分支叫b，这样A、B做的所有代码改动都各自在各自的分支，互不影响，等到俩人都把各自的模块都做完了，最后再统一把分支合并起来。

执行 git init 初始化git仓库之后会默认生成一个主分支 master，也是你所在的默认分支，也基本是实际开发正式环境下的分支，一般情况下 master 分支不会轻易直接在上面操作的，你们可以输入 git branch 查看下当前分支情况：



```
→ test git:(master) git branch
* master https://mp.weixin.qq.com/cgi-bin/appn
```

如果我们想在此基础上新建一个分支呢，很简单，执行 git branch a 就新建了一个名字叫 a 的分支，这时候分支 a 跟分支 master 是一模一样的内容，我们再输入 git branch 查看的当前分支情况：

```
→ test git:(master) git branch
```

文死a表

```
* master
```

但是可以看到 master 分支前有个 * 号，即虽然新建了一个 a 的分支，但是当前所在的分支还是在 master 上，如果我们想在 a 分支上进行开发，首先要先切换到 a 分支上才行，所以下一步要切换分支

```
git checkout a
```

执行这个命令，然后再输入 git branch 查看下分支情况：

```
→ test git:(master) git checkout a
```

Switched to branch 'a'

```
→ test git:(a) git branch
```

文死a表

```
master
```

可以看到当前我们在的分支已经是 a 了，这个时候 A 同学就可以尽情的在他新建的 a 分支去进行代码改动了。

那有人就说了，我要先新建再切换，未免有点麻烦，有没有一步到位的，聪明：

```
git checkout -b a
```

这个命令的意思就是新建一个 a 分支，并且自动切换到 a 分支。

git merge

A 同学在 a 分支代码写的不亦乐乎，终于他的功能完工了，并且 测试也都 ok 了，准备要上线了，这个时候就需要把他的代码合并到主分支 master 上来，然后发布。git merge 就是合并分支用到的命令，针对这个情况，需要先做两步，第一步是切换到 master 分支，如果你已经在了就不用切换了，第二步执行 git merge a，意思就是把 a 分支的代码合并过来，不出意外，这个时候 a 分支的代码就顺利合并到 master 分支来了。为什么说不出意外呢？因为这个时候可能会有冲突而合并失败，留个包袱，这个到后面进阶的时候再讲。

git branch -d

有新建分支，那肯定有删除分支，假如这个分支新建错了，或者 a 分支的代码已经顺利合并到 master 分支来了，那么 a 分支没用了，需要删除，这个时候执行 git branch -d a 就可以把 a 分支删除了。

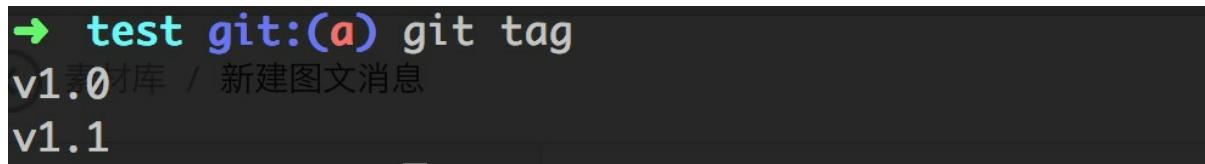
git branch -D

有些时候可能会删除失败，比如如果 a 分支的代码还没有合并到 master，你执行 git branch -d a 是删除不了的，它会智能的提示你 a 分支还有未合并的代码，但是如果你非要删除，那就执行 git branch -D a 就可以强制删除 a 分支。

git tag

我们在客户端开发的时候经常有版本的概念，比如 v1.0、v1.1 之类的，不同的版本肯定对应不同的代码，所以我一般要给我们的代码加上标签，这样假设 v1.1 版本出了一个新 bug，但是又不知道 v1.0 是否有这个 bug，有了标签就可以顺利切换到 v1.0 的代码，重新打个包测试了。

所以如果想要新建一个标签很简单，比如 git tag v1.0 就代表我在当前代码状态下新建了一个v1.0的标签，输入 git tag 可以查看历史 tag 记录。



```
→ test git:(a) git tag
v1.0 / 新建图文消息
v1.1
```

可以看到我新建了两个标签 v1.0、v1.1。

想要切换到某个tag怎么办？也很简单，执行 git checkout v1.0，这样就顺利的切换到 v1.0 tag的代码状态了。

OK，以上全是一些最基本的Git操作，而且全是在本地环境进行操作的，完全没有涉及到远程仓库，下一章节将以远程 GitHub 仓库为例，讲解下本地如何跟远程仓库一起同步协作，另外今天讲的全是最基础最简单的Git操作，一步步来，后续再继续讲解一下Git的高阶以及一些Git的酷炫操作。

另外，考虑到可能会有人嫌我讲解的太基础太慢，毕竟我是针对小白，所以得一步步来，迫不及待的想要提前自己学习的不妨在我公众号 AndroidDeveloper 回复「git」关键字，获取一份我推荐的还不错的 Git 学习资料，不谢，毕竟我这么帅！

相关文章：

[从0开始学习 GitHub 系列之「初识 GitHub」](#)

[从0开始学习 GitHub 系列之「加入 GitHub」](#)

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「04.向GitHub 提交代码」

之前的这篇文章「[从0开始学习 GitHub 系列之「Git速成」](#)」相信大家都已经对 Git 的基本操作熟悉了，但是这篇文章只介绍了对本地 Git 仓库的基本操作，今天我就来介绍下如何跟远程仓库一起协作，教你们向 GitHub 上提交你们的第一行代码！

1. SSH

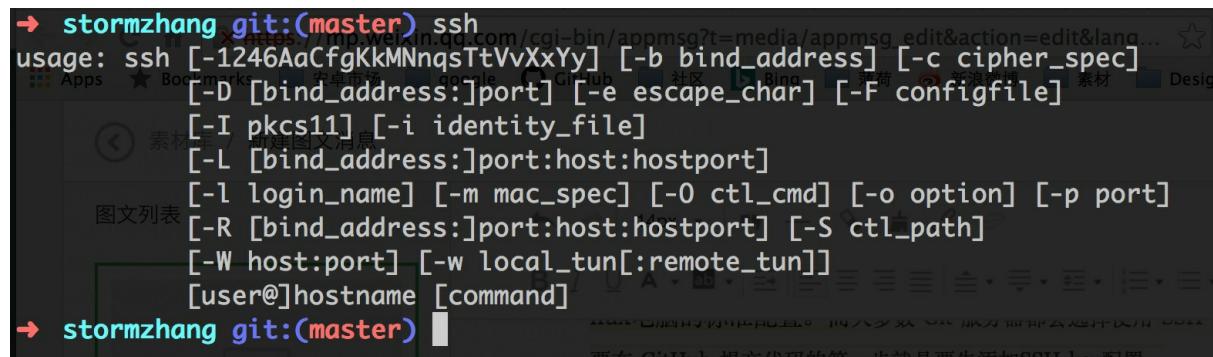
你拥有了一个 GitHub 账号之后，就可以自由的 clone 或者下载其他项目，也可以创建自己的项目，但是你没法提交代码。仔细想想也知道，肯定不可能随意就能提交代码的，如果随意可以提交代码，那么 GitHub 上的项目岂不乱了套了，所以提交代码之前一定是需要某种授权的，而 GitHub 上一般都是基于 SSH 授权的。

那么什么是 SSH 呢？

简单点说，SSH 是一种网络协议，用于计算机之间的加密登录。目前是每一台 [Linux](#) 电脑的标准配置。而大多数 Git 服务器都会选择使用 SSH 公钥来进行授权，所以想要在 GitHub 提交代码的第一步就是要先添加 SSH key 配置。

2. 生成SSH key

Linux 与 Mac 都是默认安装了 SSH，而 Windows 系统安装了 Git Bash 应该也是带了 SSH 的。大家可以在终端（win下在 Git Bash 里）输入 ssh 如果出现以下提示证明你本机已经安装 SSH，否则请搜索自行安装下。



```
→ stormzhang git:(master) ssh
usage: ssh [-1246AaCfgKkMNnqsTtVvXxYy] [-b bind_address] [-c cipher_spec]
           [-D [bind_address:]port] [-e escape_char] [-F configfile]
           [-I pkcs11] [-i identity_file]
           [-L [bind_address:]port:host:hostport]
           [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port]
           [-R [bind_address:]port:host:hostport] [-S ctl_path]
           [-W host:port] [-w local_tun[:remote_tun]]
           [user@]hostname [command]
→ stormzhang git:(master)
```

紧接着输入 `ssh-keygen -t rsa`，什么意思呢？就是指定 rsa [算法](#)生成密钥，接着连续三个回车键（不需要输入密码），然后就会生成两个文件 id_rsa 和 id_rsa.pub，而 id_rsa 是密钥，id_rsa.pub 就是公钥。这两文件默认分别在如下目录里生成：

Linux/Mac 系统在 `~/.ssh` 下，win系统在 `/c/Documents and Settings/username/.ssh` 下，都是隐藏文件，相信你们有办法查看的。

```
MINGW64:/c/Users/Allen/Desktop
Allen@Android MINGW64 ~/Desktop
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/Allen/.ssh/id_rsa): 保存路径
Created directory '/c/Users/Allen/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/Allen/.ssh/id_rsa.
Your public key has been saved in /c/Users/Allen/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:m2oQ+hrtcRhoXTPpRloUVUMRGseUD+Oajbc86RXcjWg Allen@Android
The key's randomart image is:
+---[RSA 2048]----+
|   ooo=O+
|   . . +=.
|   = .. = o o
|   ..* . + E o .
|   o.=o SB + .
|   o.o.= oo= .
|   o.+.. o. o
|   o.o.. .
|   .....
+---[SHA256]-----
```

接下来要做的是把 id_rsa.pub 的内容添加到 GitHub 上，这样你本地的 id_rsa 密钥跟 GitHub 上的 id_rsa.pub 公钥进行配对，授权成功才可以提交代码。

3. GitHub 上添加 SSH key

第一步先在 GitHub 上的设置页面，点击最左侧 SSH and GPG keys：

The screenshot shows the GitHub settings interface. On the left, there is a sidebar with various options like Profile, Account, Emails, Notifications, Billing, Security, OAuth applications, Personal access tokens, Repositories, Organizations, and Saved replies. The 'SSH and GPG keys' option is highlighted with a red box. On the right, there are two main sections: 'SSH keys' and 'GPG keys'. The 'SSH keys' section contains a list of existing keys, including one for 'storm@StormMBP.local' with a green key icon. A 'New SSH key' button is located at the top right of this section, also highlighted with a red box. The 'GPG keys' section indicates that there are no GPG keys associated with the account.

然后点击右上角的 New SSH key 按钮：

SSH keys

New SSH key

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

 storm@StormMBP.local	Fingerprint: a2:cc:81:70:ba:e4:7a:64:42:ab:e4:e0:6f:09:96:b9	Delete
SSH	Added on May 1, 2013 — Last used within the last day	
Title		
<input type="text"/>		
Key		
<pre>ssh-rsa NzaC1yc2EAAAQABAAQDLf0hHHFtW7xjLeI/rwoYPC3+MHE4HT+ATSRuhKnr19k40RdpICgP4/uxcf5Rj3gnVv XjFnwqMOiOHhFs33tRztGeU/m1DZVV0v39rLpl+0Q8cC8M1EUReLnGSLvTGfMjsUr91YiANLkGSK0dXOrR63Vu7t26IF EY4HlnfkeKdvcUCxq/63lslu/v1svVhn8Ge6U9ZQdeK+O6Htb+CJFekJeYnq9036OxG0DHGHfl7zv7sftYg/yqlIQHtmLCkv MATg2cDOZ4BzPSa/SOQ2b9PUzoLcYjOq1GP4xWKV053mWTNeGnaKgKqVb8KwJ storm@StormMBP.local</pre>		
Add SSH key		
<small>② Check out our guide to generating SSH keys or troubleshoot common SSH Problems.</small>		

需要做的只是在 Key 那栏把 id_rsa.pub 公钥文件里的内容复制粘贴进去就可以了（上述示例为了安全粘贴的公钥是无效的），Title 那栏不需要填写，点击 Add SSH key 按钮就ok了。

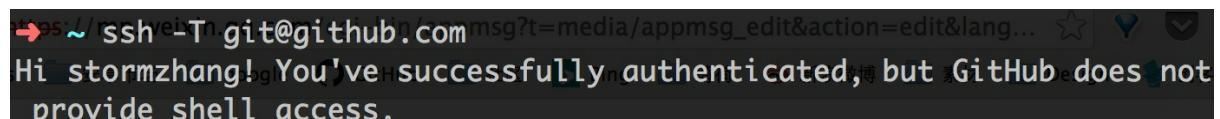
这里提醒下，怎么查看 id_rsa.pub 文件的内容？

Linux/Mac 用户执行以下命令：

```
cd ~/.ssh
cat id_rsa.pub
```

Windows用户，设置显示隐藏文件，可以使用 EditPlus 或者 Sublime 打开复制就行了。

SSH key 添加成功之后，输入 `ssh -T git@github.com` 进行测试，如果出现以下提示证明添加成功了。



→ 3 ~/ ssh -T git@github.com msg?t=media/appmsg_edit&action=edit&lang... ★ 🌐 ⚙️
 Hi stormzhang! You've successfully authenticated, but GitHub does not provide shell access.

```
Allen@Android MINGW64 ~/Desktop
$ ssh -T git@github.com
The authenticity of host 'github.com (192.30.255.113)' can't be established.
RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxrdCARLviKw6E5SY8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'github.com,192.30.255.113' (RSA) to the list of known hosts.
Hi JackChan1999! You've successfully authenticated, but GitHub does not provide shell access.
```

4. Push & Pull

在提交代码之前我们先要了解两个命令，也是上次的文章没有介绍的，因为这两个命令需要跟远程仓库配合。

Push：直译过来就是「推」的意思，什么意思呢？如果你本地代码有更新了，那么就需要把本地代码推到远程仓库，这样本地仓库跟远程仓库就可以保持同步了。

代码示例：`git push origin master`

意思就是把本地代码推到远程 master 分支。

Pull：直译过来就是「拉」的意思，如果别人提交代码到远程仓库，这个时候你需要把远程仓库的最新代码拉下来，然后保证两端代码的同步。

代码示例：`git pull origin master`

意思就是把远程最新的代码更新到本地。一般我们在 push 之前都会先 pull，这样不容易冲突。

5. 提交代码

添加 SSH key 成功之后，我们就有权限向 GitHub 上我们自己的项目提交代码了，而提交代码有两种方法：

Clone自己的项目

我以我在 GitHub 上创建的 test 项目为例，执行如下命令：

```
git clone git@github.com:stormzhang/test.git
```

这样就把 test 项目 clone 到了本地，你可以把 clone 命令理解为高级点的复制，这个时候该项目本身就已经是一个 git 仓库了，不需要执行 `git init` 进行初始化，而且甚至都已经关联好了远程仓库，我们只需要在这个 test 目录下任意修改或者添加文件，然后进行 commit，之后就可以执行：

```
git push origin master
```

进行代码提交，这种是最简单方便的一种方式。

至于怎么获取项目的仓库地址呢？如下图：

The screenshot shows the GitHub repository page for 'test'. At the top, there's a header with the repository name 'stormzhang / test', a star count (2), and a fork count (0). Below the header, there are tabs for 'Code', 'Issues 1', 'Pull requests 0', 'Wiki', 'Pulse', 'Graphs', and 'Settings'. Under the 'Code' tab, it shows '1 commit', '1 branch', '0 releases', and '1 contributor'. There are buttons for 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The 'Clone or download' button is highlighted with a red box. A tooltip for 'Clone with SSH' is visible, showing the URL 'git@github.com:stormzhang/test.git' which is also highlighted with a red box. Below the repository stats, there's a list of files: 'README.md' (Initial commit) and another 'README.md' (test). At the bottom, there's a large 'test' heading.

关联本地已有项目 如果我们本地已经有一个完整的 git 仓库，并且已经进行了很多次 commit，这个时候第一种方法就不适合了。

假设我们本地有个 test2 的项目，我们需要的是在 GitHub 上建一个 test 的项目，然后把本地 test2 上的所有代码 commit 记录提交到 GitHub 上的 test 项目。

第一步就是在 GitHub 上建一个 test 项目，这个想必大家都会了，就不用多讲了。

第二步把本地 test2 项目与 GitHub 上的 test 项目进行关联，切换到 test2 目录，执行如下命令：

```
git remote add origin git@github.com:stormzhang/test.git
```

什么意思呢？就是添加一个远程仓库，他的地址是 `git@github.com:stormzhang/test.git`，而 `origin` 是给这个项目的远程仓库起的名字，是的，名字你可以随便取，只不过大家公认的只有一个远程仓库时名字就是 `origin`，为什么要给远程仓库取名字？因为我们可能一个项目有多个远程仓库？比如 GitHub 一个，比如公司一个，这样的话提交到不同的远程仓库就需要指定不同的仓库名字了。

查看我们当前项目有哪些远程仓库可以执行如下命令：

```
git remote -v
```

接下来，我们本地的仓库就可以向远程仓库进行代码提交了：

```
git push origin master
```

就是默认向 GitHub 上的 test 目录提交了代码，而这个代码是在 master 分支。当然你可以提交到指定的分支，这个之后的文章再详细讲解。

对了，友情提醒，在提交代码之前先要设置下自己的用户名与邮箱，这些信息会出现在所有的 commit 记录里，执行以下代码就可以设置：

```
git config --global user.name "stormzhang"  
git config --global user.email "stormzhang.dev@gmail.com"
```

6. 总结

通过本文的介绍，终于大家可以成功的向 GitHub 提交代码了，但是相信大家还有很多疑问，比如关于分支的理解与使用，比如 git 的其他一些有用的配置，比如怎么向一些开源项目贡献代码，发起 Pull Request 等，之后的系列文章会逐一进行介绍，敬请期待。

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「05.Git 进阶」

关于 [Git](#) 相信大家看了之前一系列的文章已经初步会使用了，但是关于Git还有很多知识与技巧是你不知道的，今天就来给大家介绍下一些 Git 进阶的知识。

1. 用户名和邮箱

我们知道我们进行的每一次commit都会产生一条log，这条log标记了提交人的姓名与邮箱，以便其他人方便的查看与联系提交人，所以我们在进行提交代码的第一步就是要设置自己的用户名与邮箱。执行以下代码：

```
git config --global user.name "stormzhang"  
git config --global user.email "stormzhang.dev@gmail.com"
```

以上进行了全局配置，当然有些时候我们的某一个项目想要用特定的邮箱，这个时候只需切换到你的项目，以上代码把 **-global** 参数去除，再重新执行一遍就ok了。

PS：我们在 GitHub 的每次提交理论上都会在主页的下面产生一条绿色小方块的记录，如果你确认你提交了，但是没有绿色方块显示，那肯定是你提交代码配置的邮箱跟你 GitHub 上的邮箱不一致，GitHub 上的邮箱可以到 **Setting -> Emails** 里查看。

2. alias

我们知道我们执行的一些Git命令其实操作很频繁的类似有：

```
git commit  
git checkout  
git branch  
git status  
...
```

这些操作非常频繁，每次都要输入完全是不是有点麻烦，有没有一种简单的缩写输入呢？比如我对应的直接输入以下：

```
git c  
git co  
git br  
git s  
...
```

是不是很简单快捷啊？这个时候就用到了alias配置了，翻译过来就是别名的意思，输入以下命令就可以直接满足了以上的需求。

```
git config --global alias.co checkout # 别名  
git config --global alias.ci commit  
git config --global alias.st status  
git config --global alias.br branch
```

当然以上别名不是固定的，你完全可以根据自己的习惯去定制，除此之外还可以设置组合，比如：

```
git config --global alias.psm 'push origin master'  
git config --global alias.plm 'pull origin master'
```

之后经常用到的 `git push origin master` 和 `git pull origin master` 直接就用 `git psm` 和 `git plm` 代替了，是不是很方便？

另外这里给大家推荐一个很强大的 alias 命令，我们知道我们输入 `git log` 查看日志的时候是类似这样的：

```
commit 0b5e8caecc2c6ff8426079c005f3452288073463  
Author: ttdevs <...> Date: Wed May 11 13:38:35 2016 +0800  
        git diff <branch1>..<branch2> # 比较两次提交之间的差异  
        git diff --staged # 比较暂存区和版本库差异  
        git diff --cached # 比较暂存区和版本库差异  
        git log # 仅比较统计信息  
commit 63ff87d6380135258964ddf689e7b23f438e576b  
Merge: 6a5ed4c 0fd7789  
Author: loody <...>  
Date: Thu May 19 11:04:11 2016 +0800  
        git log  
        Merge branch 'daily_build' into feature/ping++  
        git log -p <file> # 查看每次详细修改内容的diff  
commit 0fd7789e824f2ca8e67ae4f2f5a2ec81b0a15973  
Merge: 48bbea3 9d10a3b  
Author: loody <...>  
Date: Thu May 19 11:03:45 2016 +0800  
        Mac上可以使用tig代替diff和log, brew install tig  
        Merge branch 'daily_build' of git.boohee.cn:android/one into daily_build  
        Git 本地分支管理
```

告诉大家一个比较屌的命令，输入

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

然后日志这样了：

```
* 0b5e8ca - (HEAD, feature/debugger) show cached ip (4 hours ago) <ttdevs>
* 63ff87d - (origin/s1, origin/feature/ping++, feature/ping++) Merge branch 'do
weeks ago) <loody>
| \
| * 0fd7789 - Merge branch 'daily_build' of git://github.com/ttdevs/daily-build into dail
| | \
| | * 9d10a3b - 修改饮食工具 (4 weeks ago) <wanglinglong>
* | | 6a5ed4c - Merge branch 'daily_build' into feature/ping++ (4 weeks ago) <1
| | \ 查看提交记录
| | /
| * | 48bbea3 - Merge branch 'daily_build' of git://github.com/ttdevs/daily-build into do
| | \ 查看该文件每次提交记录
| | / git log -p <file> # 查看每次详细修改内容的diff
| |
| * 2441d30 - 工具饮食改进 (4 weeks ago) <wanglinglong>
| * c7b8c38 - 修改评测 (4 weeks ago) <wanglinglong>
| * 6f4c304 - 评测适配小屏幕手机&优化 (4 weeks ago) <wanglinglong>
| * 8df50ef - 优化用户评测 (4 weeks ago) <wanglinglong>
| * 6b96942 - 处理BaseJsonRequest中判断是不是food逻辑错误 (4 weeks ago) <ttdevs>
| * 4a9649f - 用户在开始时设置的目标时间已到不需要再评测 (4 weeks ago) <wanglin
| * 6297df5 - 个人资料界面调整 (4 weeks ago) <wanglinglong>
| * f493984 - 刻度尺精度 (4 weeks ago) <wanglinglong>
| * c1a5489 - 修改评测相关问题 (5 weeks ago) <wanglinglong>
| * ca5320d - Merge branch 'daily_build' of git://github.com/ttdevs/daily-build into d
ong>
| | \
| | * 34c17fe - (origin/feature/init) 完善评测 (5 weeks ago) <wanglinglong>
| | * 1237856 - 修改用户评测 (5 weeks ago) <wanglinglong>
* | | 6bd2e71 - Merge branch 'master' into daily_build (5 weeks ago) <loody>
| | \
```

是不是比较清晰，整个分支的走向也很明确，但是每次都要输这么一大串是不是也很烦？这时候你就该想到 alias 啊：

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

这样以后直接输入 `git lg` 就行了。

3. 其他配置

当然还有一些其他有用的配置，默认情况下 git 用的编辑器是 vi，如果不喜欢可以改成其他编辑器，比如我习惯 vim。

```
git config --global core.editor "vim" # 设置Editor使用vim
```

你们如果喜欢其他编辑器可自行搜索配置，前提是本机有安装。

有些人纳闷我的终端怎么有各种颜色，自己却不是这样的，那是因为你们没有开启给 Git 着色，输入如下命令即可：

```
git config --global color.ui true
```

还有些其他的配置如：

```
git config --global core.quotePath false # 设置显示中文文件名
```

以上基本所有的配置就差不多了，默认这些配置都在 `~/.gitconfig` 文件下的，你可以找到这个文件查看自己的配置，也可以输入 `git config -l` 命令查看。

4. diff

diff命令算是很常用的，使用场景是我们经常在做代码改动，但是有的时候2天前的代码了，做了哪些改动都忘记了，在提交之前需要确认下，这个时候就可以用diff来查看你到底做了哪些改动，举个例子，比如我有一个 `a.md` 的文件，我现在做了一些改动，然后输入 `git diff` 就会看到如下：

```
diff --git a/a.md b/a.md
index ea8f022..d2a5109 100644
--- a/a.md
+++ b/a.md
@@ -1 +1,3 @@
-aaaaaaaaa
+bbbbbbbb
+ccccccccc
+ddddddddd
```

红色的部分前面有个 `-` 代表我删除的，绿色的部分前面有个 `+` 代表我增加的，所以从这里你们很一目了然的知道我到底对这个文件做了哪些改动。

值得一提的是直接输入 `git diff` 只能比较当前文件和暂存区文件差异，什么是暂存区？就是你还没有执行 `git add` 的文件。

当然跟暂存区做比较之外，他还可以有其他用法，如比较两次 commit 之间的差异，比较两个分支之间的差异，比较暂存区和版本库之间的差异等，具体用法如下：

```
git diff <$id1> <$id2> # 比较两次提交之间的差异
git diff <branch1>..<branch2> # 在两个分支之间比较
git diff --staged # 比较暂存区和版本库差异
```

5. checkout

我们知道 `checkout` 一般用作切换分支使用，比如切换到 `develop` 分支，可以执行：

```
git checkout develop
```

但是 `checkout` 不只用作切换分支，他可以用来切换tag，切换到某次commit，如：

```
git checkout v1.0
git checkout ffd9f2dd68f1eb21d36cee50dbdd504e95d9c8f7 # 后面的一长串是commit_id，是每次commit的SHA1值，可以根据 git log 看到。
```

除了有“切换”的意思，**checkout** 还有一个撤销的作用，举个例子，假设我们在一个分支开发一个小功能，刚写完一半，这时候需求变了，而且是大变化，之前写的代码完全用不了了，好在你刚写，甚至都没有 **git add** 进暂存区，这个时候很简单的一个操作就直接把原文件还原：

```
git checkout a.md
```

这里稍微提下，**checkout** 命令只能撤销还没有 **add** 进暂存区的文件。

6. stash

设想一个场景，假设我们正在一个新的分支做新的功能，这个时候突然有一个紧急的bug需要修复，而且修复完之后需要立即发布。当然你说我先把刚写的一点代码进行提交不就行了么？这样理论上当然是ok的，但是这会产品垃圾commit，原则上我们每次的commit都要有实际的意义，你的代码只是刚写了一半，还没有什么实际的意义是不建议就这样commit的，那么有没有一种比较好的办法，可以让我暂时切到别的分支，修复完bug再切回来，而且代码也能保留的呢？

这个时候 **stash** 命令就大有用处了，前提是我们的代码没有进行 **commit**，哪怕你执行了 **add** 也没关系，我们先执行

```
git stash
```

命令，什么意思呢？意思是把当前分支所有没有 **commit** 的代码先暂存起来，这个时候你再执行 **git status** 你会发现当前分支很干净，几乎看不到任何改动，你的代码改动也看不见了，但其实是暂存起来了。执行

```
git stash list
```

你会发现此时暂存区已经有了一条记录。

这个时候你可以切换到其他分支，赶紧把bug修复好，然后发布。之后一切都解决了，你再切换回来继续做你之前没做完的功能，但是之前的代码怎么还原呢？

```
git stash apply
```

你会发现你之前的代码全部又回来了，就好像一切都没发生过一样，紧接着你最好需要把暂存区的这次 **stash** 记录删除，执行：

```
git stash drop
```

就把最近一条的 **stash** 记录删除了，是不是很方便？其实还有更方便的，你可以使用：

```
git stash pop
```

来代替 **apply** 命令，**pop** 跟 **apply** 的唯一区别就是 **pop** 不但会帮你把代码还原，还自动帮你把这条 **stash** 记录删除，省的自己再 **drop** 一次了，为了验证你可以紧接着执行 **git stash list** 命令来确认是不是已经没有记录了。

最后还有一个命令介绍下：

```
git stash clear
```

就是清空所有暂存区的记录，**drop** 是只删除一条，当然后面可以跟 **stash_id** 参数来删除指定的某条记录，不跟参数就是删除最近的，而 **clear** 是清空。

7. merge & rebase

我们知道 **merge** 分支是合并的意思，我们在一个 **featureA** 分支开发完了一个功能，这个时候需要合并到主分支 **master** 上去，我们只需要进行如下操作：

```
git checkout master  
git merge featureA
```

其实 **rebase** 命令也是合并的意思，上面的需求我们一样可以如下操作：

```
git checkout master  
git rebase featureA
```

rebase 跟 **merge** 的区别你们可以理解成有两个书架，你需要把两个书架的书整理到一起去，第一种做法是 **merge**，比较粗鲁暴力，就直接腾出一块地方把另一个书架的书全部放进去，虽然暴力，但是这种做法你可以知道哪些书是来自另一个书架的；第二种做法就是 **rebase**，他会把两个书架的书先进行比较，按照购书的时间来给他重新排序，然后重新放置好，这样做好处就是合并之后的书架看起来很有逻辑，但是你很难清晰的知道哪些书来自哪个书架的。

只能说各有好处的，不同的团队根据不同的需要以及不同的习惯来选择就好。

8. 解决冲突

假设这样一个场景，A和B两位同学各自开了两个分支来开发不同的功能，大部分情况下都会尽量互不干扰的，但是有一个需求A需要改动一个基础库中的一个类的方法，不巧B这个时候由于业务需要也改动了基础库的这个方法，因为这种情况比较特殊，A和B都认为不会对地方造成影响，等两人各自把功能做完了，需要合并的到主分支 **master** 的时候，我们假设先合并A的分支，这个时候没问题的，之后再继续合并B的分支，这个时候想想也知道就有冲突了，因为A和B两个人同时更改了同一个地方，Git本身他没法判断你们两个谁更改的对，但是这个时候他会智能的提示有 **conflicts**，需要手动解决这个冲突之后再重新进行一次 **commit** 提交。我随便在项目搞了一个冲突做下示例：

```
diff --cc build.gradle
index f07fb5,840d50b..0000000
--- a/build.gradle
+++ b/build.gradle
@@@ -10,8 -6,7 +10,12 @@@ buildscript
    about.html jcenter()
    atom.{nl
    baidu dependencies{
++<<<< HEAD
+ bhjobs.html classpath 'com.tencent.mm:AndResGuard-gradle-plugin:1.1.5'
+ categories.html classpath 'com.android.tools.build:gradle:2.1.0'
CNAME
++=====+
+ favicon.ico classpath 'com.android.tools.build:gradle:1.3.0'
++>>>>baidu_activity
index.md
link.html
}my-progress.md
```

自强不息
熊
小小主V 啊
熊
6级可以学大
Android学习之...
陈宝石
个基础库中的一个类的方法，不巧B这个时候由于业务需要也改成影响，等两人各自把功能做完了，需要合并的到主分支 master，并B的分支，这个时候想想也知道就有冲突了，因为A和B两个人本身他没法判断你们两个谁更改的对，但是这个时候他会智能的 commit 提交。

以上截图里就是冲突的示例，冲突的地方由 **====** 分出了上下两个部分，上部分一个叫 **HEAD** 的字样代表是我当前所在分支的代码，下半部分是一个叫 **baidu_activity** 分支的代码，可以看到 **HEAD** 对 gradle 插件进行了升级，同时新增了一个插件，所以我们很容易判断哪些代码该保留，哪些代码该删除，我们只需要移除掉那些老旧代码，而且同时也需要把那些 **<<< HEAD**、**====** 以及 **>>>>baidu_activity** 这些标记符号也一并删除，最后进行一次 commit 就ok了。

我们在开发的过程中一般都会约定尽量大家写的代码不要彼此影响，以减少出现冲突的可能，但是冲突总归无法避免的，我们需要了解并掌握解决冲突的方法。

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「06.团队合作利器 Branch」

Git 相比于 SVN 最强大的一个地方就在于「分支」，Git 的分支操作简直不要太方便，而实际项目开发中团队合作最依赖的莫过于分支了，关于分支前面的系列也提到过，但是本篇会详细讲述什么是分支、分支的具体操作以及实际项目开发中到底是怎么依赖分支来进行团队合作的。

1. 什么是分支？

我知道读者中肯定有些人对分支这个概念比较模糊，其实你们可以这么理解，你们几个人一起去旅行，中间走到一个三岔口，每条路可能有不同的风景，你们约定 3 天之后在某地汇聚，然后各自出发了。而这三条分叉路就可以理解成你们各自的分支，而等你们汇聚的时候相当于把你们的分支进行了合并。

2. 分支的常用操作

通常我们默认都会有一个主分支叫 master，下面我们先来看下关于分支的一些基本操作：

- 新建一个叫 develop 的分支

```
git branch develop
```

这里稍微提醒下大家，新建分支的命令是基于当前所在分支的基础上进行的，即以上是基于 master 分支新建了一个叫做 develop 的分支，此时 develop 分支跟 master 分支的内容完全一样。如果你有 A、B、C 三个分支，三个分支是三位同学的，各分支内容不一样，如果你当前是在 B 分支，如果执行新建分支命令，则新建的分支内容跟 B 分支是一样的，同理如果当前所在是 C 分支，那就是基于 C 分支基础上新建的分支。

- 切换到 develop 分支

```
git checkout develop
```

- 如果把以上两步合并，即新建并且自动切换到 develop 分支：

```
git checkout -b develop
```

- 把 develop 分支推送到远程仓库

```
git push origin develop
```

- 如果你远程的分支想取名叫 develop2，那执行以下代码：

```
git push origin develop:develop2
```

但是强烈不建议这样，这会导致很混乱，很难管理，所以建议本地分支跟远程分支名要保持一致。

- 查看本地分支列表

```
git branch
```

- 查看远程分支列表

```
git branch -r
```

- 删除本地分支

```
git branch -d develop
```

```
git branch -D develop (强制删除)
```

- 删除远程分支

```
git push origin :develop
```

- 如果远程分支有个 develop , 而本地没有, 你想把远程的 develop 分支迁到本地 :

```
git checkout develop origin/develop
```

- 同样的把远程分支迁到本地顺便切换到该分支 :

```
git checkout -b develop origin/develop
```

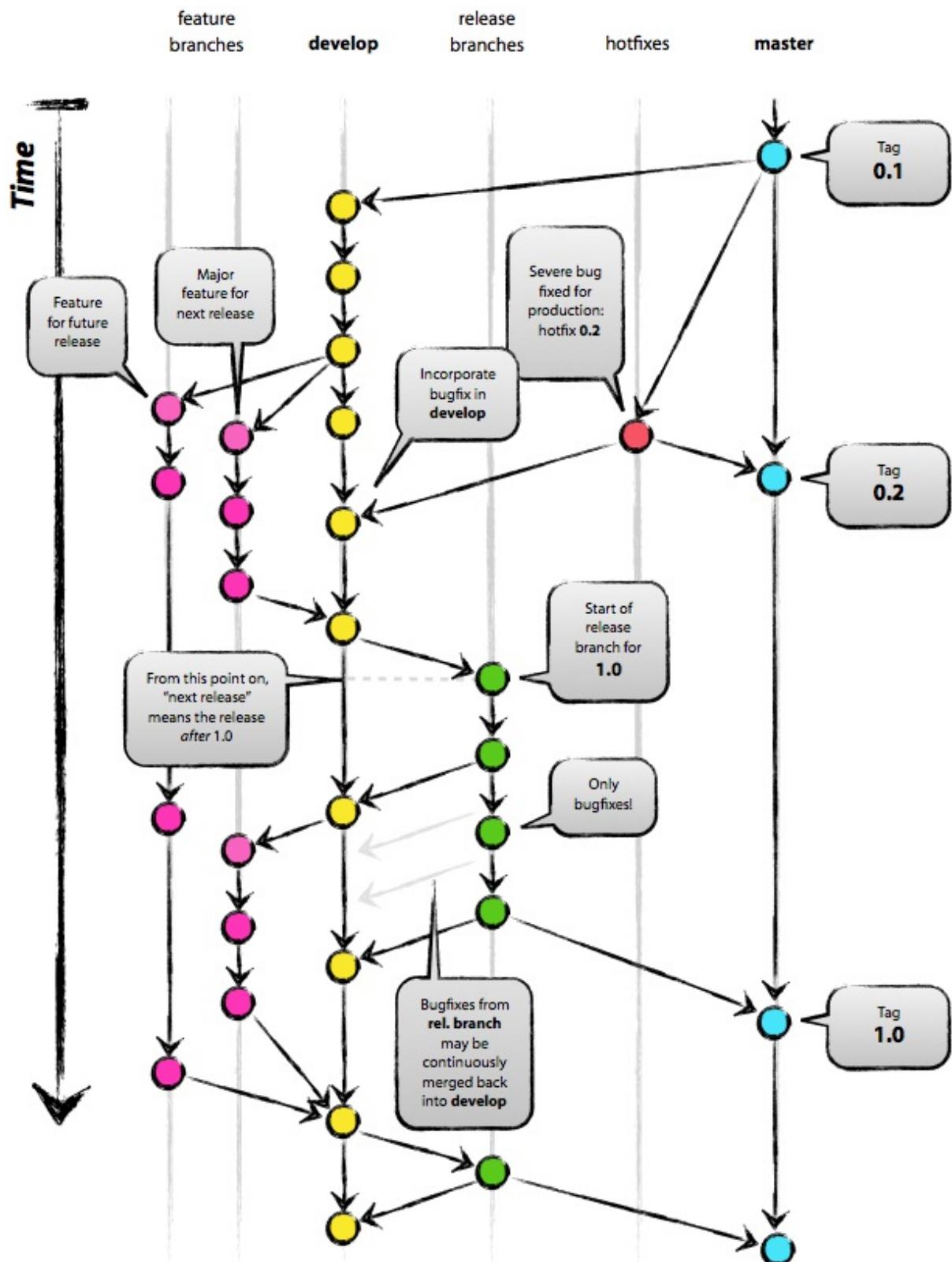
3. 基本的团队协作流程

一般来说, 如果你是一个人开发, 可能只需要 master、develop 两个分支就 ok 了, 平时开发在 develop 分支进行, 开发完成之后, 发布之前合并到 master 分支, 这个流程没啥大问题。

如果你是 3、5 个人, 那就不一样了, 有人说也没多大问题啊, 直接可以新建 A、B、C 三个人的分支啊, 每人各自开发各自的分支, 然后开发完成之后再逐步合并到 master 分支。然而现实却是, 你正在某个分支开发某个功能呢, 这时候突然发现线上有一个很严重的 bug , 不得不放下手头的工作优先处理 bug , 而且很多时候多人协作下如果没有一个规范, 很容易产生问题, 所以多人协作下的分支管理规范很重要, 就跟代码规范一样重要, 以下就跟大家推荐一种我们内部在使用的一种分支管理流程 Git Flow。

4. Git Flow

准确的说 Git Flow 是一种比较成熟的分支管理流程, 我们先看一张图能清晰的描述他整个的工作流程 :



第一次看上面那个图是不是一脸懵逼？跟我当时一样，不急，我来用简单的话给你们解释下。

一般开发来说，大部分情况下都会拥有两个分支 master 和 develop，他们的职责分别是：

- master：永远处在即将发布(production-ready)状态
- develop：最新的开发状态

确切的说 master、develop 分支大部分情况下都会保持一致，只有在上线前的测试阶段 develop 比 master 的代码要多，一旦测试没问题，准备发布了，这时候会将 develop 合并到 master 上。

但是我们发布之后又会进行下一版本的功能开发，开发中间可能又会遇到需要紧急修复 bug，一个功能开发完成之后突然需求变动了等情况，所以 Git Flow 除了以上 master 和 develop 两个主要分支以外，还提出了以下三个辅助分支：

- feature: 开发新功能的分支，基于 develop，完成后 merge 回 develop
- release: 准备要发布的分支，用来修复 bug，基于 develop，完成后 merge 回 develop 和 master
- hotfix: 修复 master 上的问题，等不及 release 版本就必须马上上线。基于 master，完成后 merge 回 master 和 develop

什么意思呢？

举个例子，假设我们已经有 master 和 develop 两个分支了，这个时候我们准备做一个功能 A，第一步我们要做的，就是基于 develop 分支新建个分支：

```
git branch feature/A
```

看到了吧，其实就是一个规范，规定了所有开发的功能分支都以 feature 为前缀。

但是这个时候做着做着发现线上有一个紧急的 bug 需要修复，那赶紧停下手头的工作，立刻切换到 master 分支，然后再此基础上新建一个分支：

```
git branch hotfix/B
```

代表新建了一个紧急修复分支，修复完成之后直接合并到 develop 和 master，然后发布。

然后再切回我们的 feature/A 分支继续着我们的开发，如果开发完了，那么合并回 develop 分支，然后在 develop 分支属于测试环境，跟后端对接并且测试的差不多了，感觉可以发布到正式环境了，这个时候再新建一个 release 分支：

```
git branch release/1.0
```

这个时候所有的 api、数据等都是正式环境，然后在这个分支上进行最后的测试，发现 bug 直接进行修改，直到测试 ok 达到了发布的标准，最后把该分支合并到 develop 和 master 然后进行发布。

以上就是 Git Flow 的概念与大概流程，看起来很复杂，但是对于人数比较多的团队协作现实开发中确实会遇到这么复杂的情况，是目前很流行的一套分支管理流程，但是有人会问每次都要各种操作，合并来合并去，有点麻烦，哈哈，这点 Git Flow 早就想到了，为此还专门推出了一个 Git Flow 的工具，并且是开源的：

GitHub 开源地址：<https://github.com/nvie/gitflow>

简单点来说，就是这个工具帮我们省下了很多步骤，比如我们当前处于 master 分支，如果想要开发一个新的功能，第一步切换到 develop 分支，第二步新建一个以 feature 开头的分支名，有了 Git Flow 直接如下操作完成了：

```
git flow feature start A
```

这个分支完成之后，需要合并到 develop 分支，然而直接进行如下操作就行：

```
git flow feature finish A
```

如果是 hotfix 或者 release 分支甚至会自动帮你合并到 develop、master 两个分支。

想必大家已经了解了这个工具的具体作用，具体安装与用法我就不多提了，感兴趣的可以看我下我之前写过的一篇博客：

<http://stormzhang.com/git/2014/01/29/git-flow/>

5. 总结

以上就是我分享给你们的关于分支的所有知识，一个人你也许感受不到什么，但是实际工作中大都以团队协作为主，而分支是团队协作必备技能，而 Git Flow 是我推荐给你们的一个很流行的分支管理流程，也是我们薄荷团队内部一直在实践的一套流程，希望对你们有借鉴意义。

更多相关博客

[从0开始学习 GitHub 系列之「初识 GitHub」](#)

[从0开始学习 GitHub 系列之「加入 GitHub」](#)

[从0开始学习 GitHub 系列之「Git 速成」](#)

[从0开始学习 GitHub 系列之「向GitHub 提交代码」](#)

[从0开始学习 GitHub 系列之「Git 进阶」](#)

本文原创发布于[微信公众号 AndroidDeveloper 「googdev」](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「07.GitHub 常见的几种操作」

之前写了一个 GitHub 系列，反响很不错，突然发现竟然还落下点东西没写，前段时间 GitHub 也改版了，借此机会补充下。

我们都说开源社区最大的魅力是人人多可以参与进去，发挥众人的力量，让一个项目更完善，更强壮。那么肯定有人疑问，我自己目前还没有能力开源一个项目，但是想一起参与到别的开源项目中，该怎么操作呢？那么今天，就来给大家一起介绍下 GitHub 上的一些常见的操作，看完之后你就知道方法了。

我们姑且以 Square 公司开源的 Retrofit 为例来介绍。

打开链接：

<https://github.com/square/retrofit>

然后看到如下的项目主页：

Type-safe HTTP client for Android and Java by Square, Inc. <http://square.github.io/retrofit/>

Branch: master ▾ New pull request Create new file Upload files Find file Clone or download ▾

Author	Commit Message	Date
JakeWharton	Merge pull request #2023 from joaham/close-responsebody-204	Latest commit 78b0491 5 days ago
.buildscript	Errors fail the deploy script.	2 years ago
.github	Issue template for retrofit.	7 months ago
retrofit-adapters	Update RxJava adapter to 1.2.0.	8 days ago
retrofit-converters	Merge pull request #2017 from square/jw/generics	8 days ago
retrofit-mock	Promote adapter response body type parameter to class level.	8 days ago
retrofit	Close ResponseBody on 204/205 to avoid connection leak	5 days ago
samples	Promote adapter response body type parameter to class level.	8 days ago
website	Remove update warning from website.	2 months ago

可以看到一个项目可以进行的操作主要就是两部分，第一部分包括 Watch、Star、Fork，这三个操作之前的系列介绍过了，这里就不啰嗦了。

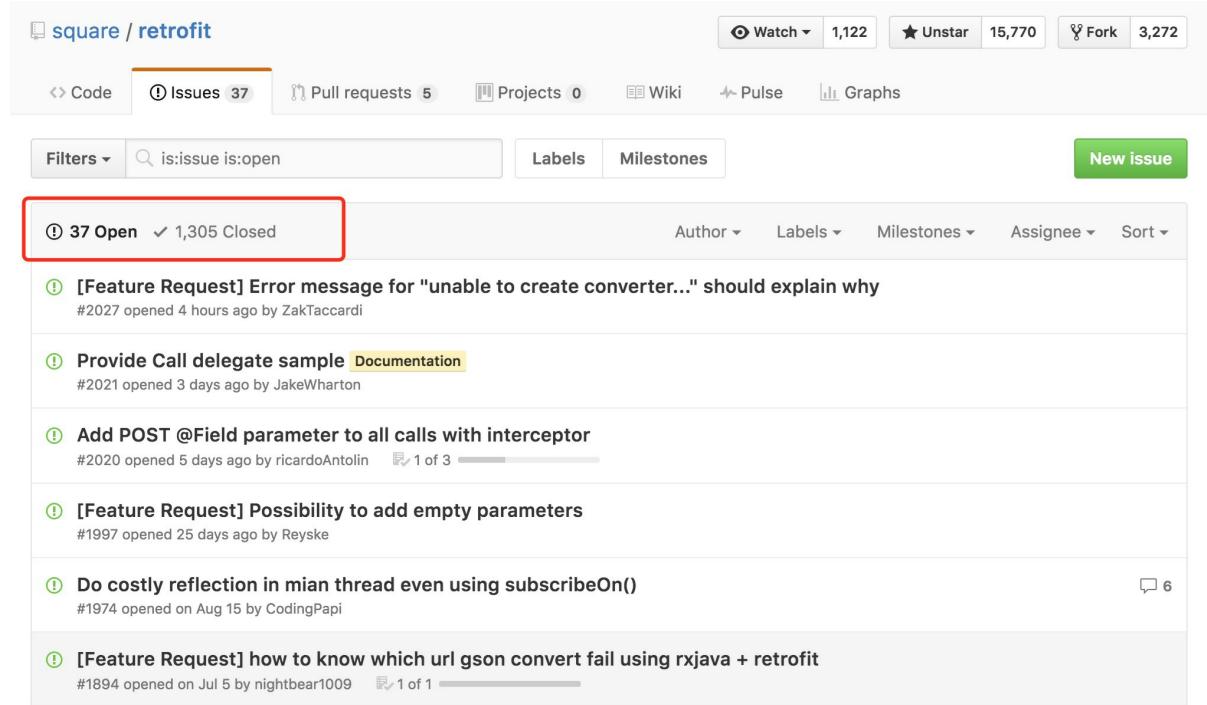
我们着重来介绍第二部分，分别包括 Code、Issues、Pull requests、Projects、Wiki、Pulse、Graphs。接下来我们来一个个解释下。

- Code

这个好理解，就是你项目的代码文件而已，这里说明一下，每个项目通常都会有对该项目的介绍，只需要在项目的根目录里添加一个 README.md 文件就可以，使用 markdown 语法，GitHub 自动会对该文件进行渲染。

- Issues

Issues 代表该项目的一些问题或者 bug，并不是说 Issues 越少越好，Issues 被解决的越多说明项目作者或者组织响应很积极，也说明该开源项目的作者很重视该项目。我们来看下 Retrofit 的 Issues 主页，截至目前 close（解决）了 1305 个 Issue，open（待解决）状态的有 37 个，这解决问题的比例与速度值得每位开源项目的作者学习。



The screenshot shows the GitHub Issues page for the Retrofit repository. At the top, there are buttons for Watch (1,122), Unstar (15,770), Fork (3,272), and a New issue button. Below the header, there are tabs for Code, Issues (37), Pull requests (5), Projects (0), Wiki, Pulse, and Graphs. A search bar with the query "is:issue is:open" is present. A red box highlights the "37 Open" count. The main list of issues includes:

- ① [Feature Request] Error message for "unable to create converter..." should explain why #2027 opened 4 hours ago by ZakTaccardi
- ① Provide Call delegate sample Documentation #2021 opened 3 days ago by JakeWharton
- ① Add POST @Field parameter to all calls with interceptor #2020 opened 5 days ago by ricardoAntolin 1 of 3
- ① [Feature Request] Possibility to add empty parameters #1997 opened 25 days ago by Reyske
- ① Do costly reflection in main thread even using subscribeOn() #1974 opened on Aug 15 by CodingPapi 6
- ① [Feature Request] how to know which url gson convert fail using rxjava + retrofit #1894 opened on Jul 5 by nightbear1009 1 of 1

同样的，大家在使用一些开源项目有问题的时候都可以提 Issue，可以通过点击右上角的 New Issue 来新建 Issue，需要添加一个标题与描述就可以了，这个操作很简单。

- Pull requests

我们都知道 GitHub 的最大魅力在于人人都可参与，比如别人开源一个项目，我们每个人都可以一起参与开发，一起来完善，而这都通过 Pull requests 来完成，简称 PR。这个没法在 Retrofit 演示，下面我就以我自己在 GitHub 上的一个项目 9GAG 来给大家详细演示下怎么给一个项目发起 PR：

提前说明下，你必须确保你可以正常向 GitHub 提交代码，如果不可以的话，请看我之前的系列文章。

第一步登录你的 GitHub 账号，然后找到你想发起 PR 的项目，这里以 9GAG 为例，点击右上角的 Fork 按钮，然后该项目就出现在了你自己账号的 Repository 里。

请注意，这个项目原本是属于 GitHub 账号 stormzhang 下的，为了演示，我自己又重新注册了另一个账号叫 googdev 单纯为了演示而用。

Fork 之后，在账号 googdev 下多了一个 9GAG 的项目，截图显示如下：

googdev / 9GAG
forked from stormzhang/9GAG

Unwatch 1 Star 0 Fork 1,030

Code Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

Android unofficial REST Client of 9GAG. <http://stormzhang.com> — Edit

100 commits 2 branches 1 release 3 contributors

Your recently pushed branches:
patch-1 (7 minutes ago) Compare & pull request

Branch: master New pull request Create new file Upload files Find file Clone or download ▾

This branch is 1 commit ahead of stormzhang:master.

Patch-1 (7 minutes ago) Pull request Compare

orangecat committed on GitHub Update README.md ... Latest commit 7a33974 16 minutes ago

orangecat app Add config.gradle 7 months ago

orangecat extras/ShimmerAndroid Add config.gradle 7 months ago

orangecat gradle/wrapper Upgrade gradle plugin 2 years ago

可以看到 Fork 过来的项目标题底部会显示一行小字：fork from stormzhang/9GAG，除此之外，项目代码跟原项目一模一样，对于原项目来说，相当于别人新建了一个分支而已。

第二步，把该项目 clone 到本地，然后修改的 bug 也好，想要新增的功能也好，总之把自己做的代码改动开发完，保存好。为了方便演示，我这里只在原项目的 README.md 文件添加了一行文字：Fork from stormzhang！

接着，把自己做的代码改动 push 到你自己的 GitHub 上去。

相信看过我前面教程的同学这一步应该都会，不会的可以滚回去看前面的教程了。

第三步，点击你 Fork 过来的项目主页的 Pull requests 页面，

googdev / 9GAG
forked from stormzhang/9GAG

Unwatch 1 Star 0 Fork 1,030

Code Pull requests 0 Projects 0 Wiki Pulse Graphs Settings

Filters is:pr is:open Labels Milestones New pull request

Welcome to Pull Requests!

Pull requests help you collaborate on code with other people. As pull requests are created, they'll appear here in a searchable and filterable list. To get started, you should [create a pull request](#).

点击 New pull request 按钮紧接着到如下页面：

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

The screenshot shows a GitHub pull request interface. At the top, there are dropdown menus for 'base fork: stormzhang/9GAG', 'base: master', '...', 'head fork: googdev/9GAG', and 'compare: master'. A red box highlights the 'Compare' section where it says 'Able to merge. These branches can be automatically merged.' Below this, a green button labeled 'Create pull request' is also highlighted with a red box. To its right, a note says 'Discuss and review the changes in this comparison with others.' and a question mark icon. Further down, summary statistics are shown: '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. A detailed commit log follows, starting with 'Commits on Sep 25, 2016' and listing a single commit from 'googdev' that updated 'README.md'. The commit details show 'Showing 1 changed file with 2 additions and 0 deletions.' Below this, a diff view of 'README.md' is presented. A red box highlights the addition of the line '+Fork from stormzhang !' at line 4. The diff view includes standard GitHub syntax highlighting and line numbers.

这个页面自动会比较该项目与原有项目的不同之处，最顶部声明了是 stormzhang/9GAG 项目的 master 分支与你 fork 过来的 googdev/9GAG 项目 master 分支所做的比较。

然后最顶部可以方便直观的看到到底代码中做了哪些改动，你们也看到我就是加了一句 Fork from stormzhang !

同样的我写好标题和描述，然后我们点击中间的 Create pull request 按钮，至此我们就成功给该项目提交了一个 PR。

然后就等着项目原作者 review 你的代码，并且决定会不会接受你的 PR，如果接受，那么恭喜你，你已经是该项目的贡献者之一了。

- Projects

这个是最新 GitHub 改版新增的一个项目，这个项目就是方便你把一些 Issues、Pull requests 进行分类，反正我觉得该功能很鸡肋，起码到目前为止基本没人用该功能，你们了解下就好。

- Wiki

一般来说，我们项目的主页有 README.me 基本就够了，但是有些时候我们项目的一些用法很复杂，需要有详细的使用说明文档给开源项目的使用者，这个时候就用到了 Wiki。

Home

Philipp Jahoda edited this page on Dec 19, 2015 · 2 revisions

[Edit](#) [New Page](#)

Welcome to the Retrofit wiki page!

▼ Pages 4

[Home](#)
[Call Adapters](#)
[Converters](#)
[Retrofit Tutorials](#)

Clone this wiki locally

<https://github.com/square/r>

[Clone in Desktop](#)

使用起来也很简单，直接 New Page，然后使用 markdown 语法即可进行编写。

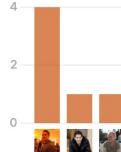
- Pulse

Pulse 可以理解成该项目的活跃汇总。包括近期该仓库创建了多少个 Pull Request 或 Issue，有多少人参与了这个仓库的开发等，都可以在这里一目了然。

根据这个页面，用户可以判断该项目受关注程度以及项目作者是否还在积极参与解决这些问题等。



Excluding merges, 3 authors have pushed 6 commits to master and 6 commits to all branches. On master, 36 files have changed and there have been 453 additions and 991 deletions.

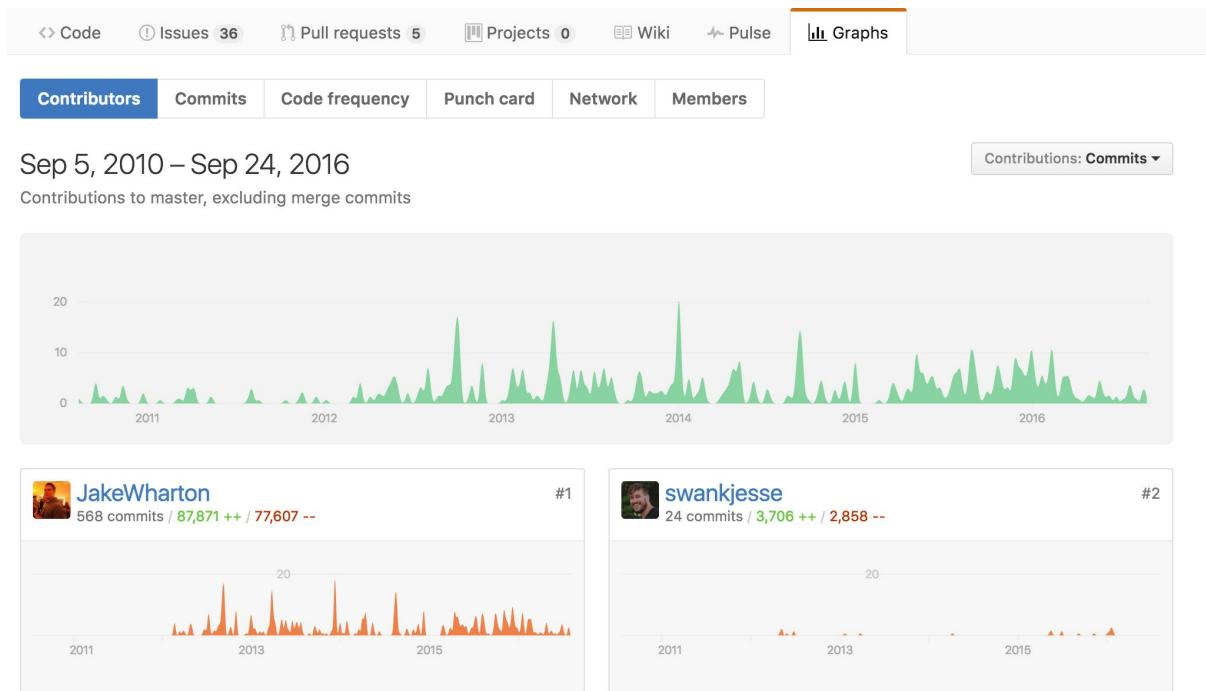


5 Pull requests merged by 3 people

- Merged** #2023 [Close ResponseBody on 204/205 to avoid connection leak](#) 5 days ago
- Merged** #2016 [Update RxJava adapter to 1.2.0.](#) 7 days ago
- Merged** #2017 [Promote adapter response body type parameter to class level.](#) 8 days ago
- Merged** #2018 [Add method for returning a Moshi converter that serializes nulls.](#) 8 days ago

- Graphs

Graphs 是以图表的形式来展示该项目的一个整体情况。比如项目的全部贡献人，比如 commits 的国度分析，比如某天代码提交的频繁程度等。



- Settings

如果一个项目是自己的，那么你会发现会多一个菜单 Settings，这里包括了你对整个项目的设置信息，比如对项目重命名，比如删除该项目，比如关闭项目的 Wiki 和 Issues 功能等，不过大部分情况下我们都不需要对这些设置做更改。感兴趣的，可以自行看下这里的设置有哪些功能。

stormzhang / 9GAG

Code Issues Pull requests Projects Wiki Pulse Graphs Settings

Options

- Collaborators
- Branches
- Webhooks
- Integrations & services
- Deploy keys

Settings

Repository name: 9GAG

Features

- Wikis
- Restrict editing to collaborators only
- Issues

以上就包含了一个 GitHub 项目的所有操作，相信大家看完之后对 GitHub 上一些常用的操作都熟悉了，从现在开始，请一起参与到开源社区中来吧，开源社区需要我们每个人都贡献一份力，这样开源社区才能越来越强大，也才能对更多的人有帮助！

相关阅读

[从0开始学习 GitHub 系列之「初识 GitHub」](#)

[从0开始学习 GitHub 系列之「加入 GitHub」](#)

[从0开始学习 GitHub 系列之「Git 速成」](#)

[从0开始学习 GitHub 系列之「向GitHub 提交代码」](#)

[从0开始学习 GitHub 系列之「Git 进阶」](#)

[从0开始学习 GitHub 系列之「团队合作利器 BRANCH」](#)

[从0开始学习 GitHub 系列之「如何发现优秀的开源项目？」](#)

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

从0开始学习 GitHub 系列之「08.如何发现优秀的开源项目」

之前发过一系列有关 GitHub 的文章，有同学问了，GitHub 我大概了解了，[Git](#) 也差不多会使用了，但是 还是搞不清 GitHub 如何帮助我的工作，怎么提升我的工作效率？

问到点子上了，GitHub 其中一个最重要的作用就是发现全世界最优秀的开源项目，你没事的时候刷刷微博、知乎，人家没事的时候刷刷 GitHub，看看最近有哪些流行的项目，久而久之，这差距就越来越大，那么如何发现优秀的开源项目呢？这篇文章我就来给大家介绍下。

1. 关注一些活跃的大牛

GitHub 主页有一个类似微博的时间线功能，所有你关注的人的动作，比如 star、fork 了某个项目都会出现在你的时间线上，这种方式适合我这种比较懒的人，不用主动去找项目，而这种基本是我每天获取信息的一个很重要的方式。不知道怎么关注这些人？那么很简单，关注我 `stormzhang`，以及我 GitHub 上关注的一些大牛，基本就差不多了。



stormzhang ▾



asLody starred p0sixspwn/p0sixspwn 39 minutes ago



baoyongzhang starred a-voyager/AutoInstaller 2 hours ago



Skykai521 starred reark/reark 2 hours ago



elezhangwen starred stormzhang/free-programming-books 2 hours ago



green robot starred markusl/GoogleTestRunner 15 days ago



chrisjenx starred xetorthio/jedis 15 days ago



johnkil starred requery/sqlite-android 15 days ago



Skykai521 starred ldk/SmallChart 15 days ago



ryancheung created repository ryancheung-wowhub 15 days ago



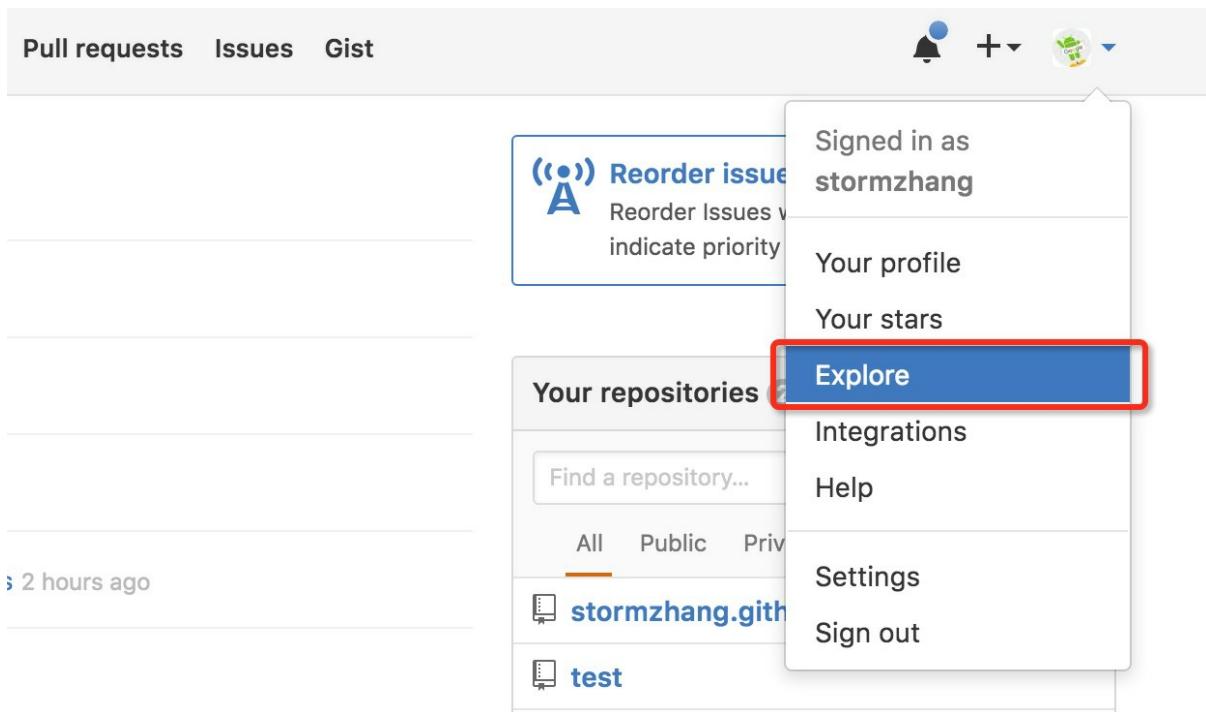
ryancheung starred AndorChen/rbenv-china-mirror 15 days ago



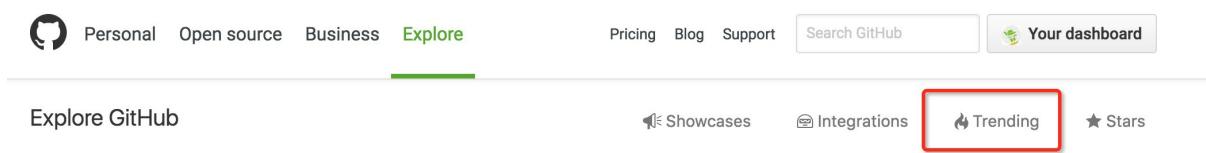
mariotaku starred kevin0571/STPopup 15 days ago

2. Trending

点击下图的 Explore 菜单到“发现”页面



紧接着点击 Trending 按钮



这个 Trending 页面是干嘛的呢？直译过来就是趋势的意思，就是说这个页面你可以看到最近一些热门的开源项目，这个页面可以算是很多人主动获取一些开源项目最好的途径，可以选择「当天热门」、「一周之内热门」和「一月之内热门」来查看，并且还可以分语言类来查看，比如你想查看最近热门的 [Android](#) 项目，那么右边就可以选择 [Java](#) 语言。

A screenshot of the 'Trending in open source' page. It shows two projects: 'FailableInc/security-guide-for-developers' and 'facebookincubator/create-react-app'. The 'Create React apps with no build configuration.' project has a 'Star' button. On the right, there are dropdown menus for 'Trending: this week' (highlighted with a red box) and 'All languages' (also highlighted with a red box). The 'All languages' menu lists 'Unknown languages', 'C', 'CSS', 'HTML', 'Java', 'JavaScript', 'Python', and 'Ruby'.

这样页面推荐大家每隔几天就去看下，主动发掘一些优秀的开源项目。

3. Search

除了 Trending，还有一种最主动的获取开源项目的方式，那就是 GitHub 的 Search 功能。

举个例子，你是做 Android 的，接触 GitHub 没多久，那么第一件事就应该输入 android 关键字进行搜索，然后右上角选择按照 star 来排序，结果如下图：

The screenshot shows the GitHub search interface with the query 'android' entered in the search bar. A red box highlights the 'Sort: Most stars ▾' button. The results page displays 373,145 repository results. The first result is 'Trinea/android-open-project', which has 17,353 stars and 9,458 forks. Below it is 'Prinzhorn/skrollr', a library for parallax scrolling, with 15,926 stars and 3,235 forks. The third result is 'wasabeef/awesome-android-ui', a curated list of Android UI/UX libraries, with 15,691 stars and 4,362 forks. The fourth result is 'square/retrofit', a type-safe HTTP client, with 14,286 stars and 2,933 forks. The fifth result is 'codepath/android_guides', extensive open-source guides for Android developers, with 13,725 stars and 3,391 forks. On the left sidebar, there are sections for 'Repositories' (373,146), 'Code' (114,026,983), 'Issues' (2,269,602), and 'Users' (4,964). A 'Languages' section lists Java (231,785), C (18,136), C++ (7,858), JavaScript (6,914), Shell (5,184), Makefile (3,840), C# (3,057), Python (2,894), HTML (2,737), and CSS (1,427).

如果你是学习 iOS 的，那么不妨同样的方法输入 iOS 关键字看看结果：

Search Search

Repositories	135,181
Code	25,987,056
Issues	475,924
Users	2,811

We've found 135,181 repository results Sort: Most stars ▾

AFNetworking/AFNetworking Objective-C ★ 26,401 ⚡ 8,376
A delightful networking framework for iOS, OS X, watchOS, and tvOS.
Updated 9 days ago

facebook/pop Objective-C++ ★ 16,019 ⚡ 2,561
An extensible iOS and OS X animation library, useful for physics-based interactions.
Updated on May 13

Prinzhorn/skrollr HTML ★ 15,927 ⚡ 3,235
Stand-alone parallax scrolling library for mobile (Android + iOS) and desktop. No jQuery. Just plain JavaScript (and some love).
Updated on May 13

[Advanced search](#) [Cheat sheet](#)

vsouza/awesome-ios Swift ★ 13,810 ⚡ 2,372
A curated list of awesome iOS ecosystem, including Objective-C and Swift Projects
Updated 5 hours ago

BradLarson/GPUImage Objective-C ★ 13,332 ⚡ 3,312
An open source iOS framework for GPU-based image and video processing
Updated 24 days ago

可以看到按照 star 数，排名靠前基本是一些比较火的项目，一定是很用，才会这么火。值得一提的是左侧依然可以选择语言进行过滤。

而对于实际项目中用到一些库，基本上都会第一时间去 GitHub 搜索下有没有类似的库，比如项目中想采用一个网络库，那么不妨输入 android http 关键字进行搜索，因为我只想找到关于 Android 的项目，所以搜索的时候都会加上 android 关键字，按照 star 数进行排序，我们来看下结果：

Search Search

Repositories	6,705
Code	71,886,340
Issues	896,804
Users	

We've found 6,705 repository results Sort: Most stars ▾

square/retrofit Java ★ 14,286 ⚡ 2,933
Type-safe HTTP client for Android and Java by Square, Inc.
Updated 3 days ago

square/okhttp Java ★ 13,091 ⚡ 3,411
An HTTP + HTTP/2 client for Android and Java applications.
Updated 3 hours ago

loop/android-async-http Java ★ 9,078 ⚡ 4,228
An Asynchronous HTTP Library for Android
Updated on Jun 27

可以看到 Retrofit、OkHttp、android-async-http 是最流行的网络库，只不过 android-async-http 的作者不维护了，之前很多人问我网络库用哪个比较好？哪怕你对每个网络库都不是很了解，那么单纯的按照这种方式你都该优先选择 Retrofit 或者 OkHttp，而目前绝大部分 Android 开发者确实也都是在用这两个网络库，当然还有部分在用 Volley 的，因为 google 没有选择在 GitHub 开源 volley，所以搜不到 volley 的上榜。

除此之外，GitHub 的 Search 还有一些小技巧，比如你想搜索的结果中 star 数大于1000的，那么可以这样搜索：

android http stars:>1000

当然还有其他小技巧，但是我觉得不是很重要，就不多说了。

有些人如果习惯用 Google 进行搜索，那么想搜索 GitHub 上的结果，不妨前面加 GitHub 关键字就ok了，比如我在 google 里输入 GitHub android http，每个关键字用空格隔开，然后搜索结果如下：

The screenshot shows a Google search results page for the query "github android http". The search bar at the top contains the query. Below it, the "All" tab is selected among other options like Images, Videos, News, Maps, More, and Search tools. The results section starts with a summary: "About 3,800,000 results (0.41 seconds)". The first result is a link to the GitHub repository for Retrofit, titled "GitHub - loopj/android-async-http: An Asynchronous HTTP Library for ...". It includes a snippet of the repository's description: "An asynchronous, callback-based Http client for Android built on top of Apache's ...". Below this, there are links to the repository's homepage, samples, issues, and releases, along with a note about the page being visited 5 times. The second result is for OkHttp, titled "GitHub - square/okhttp: An HTTP+HTTP/2 client for Android and Java ...". It includes a snippet: "okhttp - An HTTP+HTTP/2 client for Android and Java applications." and links to its wiki, samples, issues, and releases. The third result is for Volley, titled "GitHub - kevinsawicki/http-request: Java HTTP Request Library". It includes a snippet: "Contribute to http-request development by creating an account on GitHub. ... to (Android), you just want to use a good old-fashioned HttpURLConnection .". The fourth result is for Retrofit, titled "GitHub - afollestad/bridge: A simple but powerful HTTP networking ...". It includes a snippet: "A simple but powerful HTTP networking library for Android. It features a Fluent chainable API, powered by Java/Android's URLConnection classes for maximum ...". The fifth result is for OkHttp, titled "GitHub - square/retrofit: Type-safe HTTP client for Android and Java by ...". It includes a snippet: "retrofit - Type-safe HTTP client for Android and Java by Square, Inc.".

可以看到，基本也是我们想要的结果，只不过排序就不是单纯的按照 star 来排序了。

福利大放送

相信以上三种方法够大家遨游在 GitHub 的海洋了，最后给大家献上一些福利，这些项目是 GitHub 上影响力很大，同时又对你们很有用的项目：

- [free-programming-books](#)

这个项目目前 star 数排名 GitHub 第三，总 star 数超过6w，这个项目整理了所有跟编程相关的免费书籍，而且全球多国语言版的都有，中文版的在这里：[free-programming-books-zh](#)，有了这个项目，理论上你可以获取任何编程相关的学习资料，强烈推荐给你们！

- [oh-my-zsh](#)

俗话说，不会用 shell 的程序员不是真正的程序员，所以建议每个程序员都懂点 shell，有用不说，装逼利器啊！而 oh-my-zsh 毫无疑问就是目前最流行，最酷炫的 shell，不多说了，懂得自然懂，不懂的以后你们会懂的！

- [awesome](#)

GitHub 上有各种 awesome 系列，简单来说就是这个系列搜罗整理了 GitHub 上各领域的资源大汇总，比如有 awesome-android, awesome-ios, awesome-java, awesome-Python 等等等，就不截图了，你们自行去感受。

- [github-cheat-sheet](#)

GitHub 的使用有各种技巧，只不过基本的就够我们用了，但是如果你对 GitHub 超级感兴趣，想更多的了解 GitHub 的使用技巧，那么这个项目就刚好是你需要的，每个 GitHub 粉都应该知道这个项目。

- [android-open-project](#)

这个项目是我一个好朋友 Trinea 整理的一个开源项目，基本囊括了所有 GitHub 上的 Android 优秀开源项目，但是缺点就是太多了不适合快速搜索定位，但是身为 Android 开发无论如何你们应该知道这个项目。

- [awesome-android-ui](#)

这个项目跟上面的区别是，这个项目只整理了所有跟 Android UI 相关的优秀开源项目，基本你在实际开发终于到的各种效果上面都几乎能找到类似的项目，简直是开发必备。

- [Android_Data](#)

这个项目是我的邪教群的一位管理员整理的，几乎包括了国内各种学习 Android 的资料，简直太全了，我为这个项目也稍微做了点力，强烈推荐你们收藏起来。

- [AndroidInterview-Q-A](#)

这个就不多说了，之前给大家推荐过的，国内一线互联网公司内部面试题库。

- [LearningNotes](#)

这是一份非常详细的面试资料，涉及 Android、Java、设计模式、[算法](#)等等等，你能想到的，你不能想到的基本都包含了，可以说是适应于任何准备面试的 Android 开发者，看完这个之后别说你还不知道怎么面试！

总结

GitHub 上优秀开源项目真的是一大堆，就不一一推荐了，授人以鱼不如授人以渔，请大家自行主动发掘自己需要的开源项目吧，不管是应用在实际项目上，还是对源码的学习，都是提升自己工作效率与技能的很重要的一个渠道，总有一天，你会突然意识到，原来不知不觉你已经走了这么远！

觉得不错，不妨随手转发、点赞，都是对我良心张莫大的鼓励！

关注我的[微信](#)公众号 `AndroidDeveloper「googdev」`，第一时间获取文章更新以及更多原创干货分享！

使用GIT FLOW管理开发流程

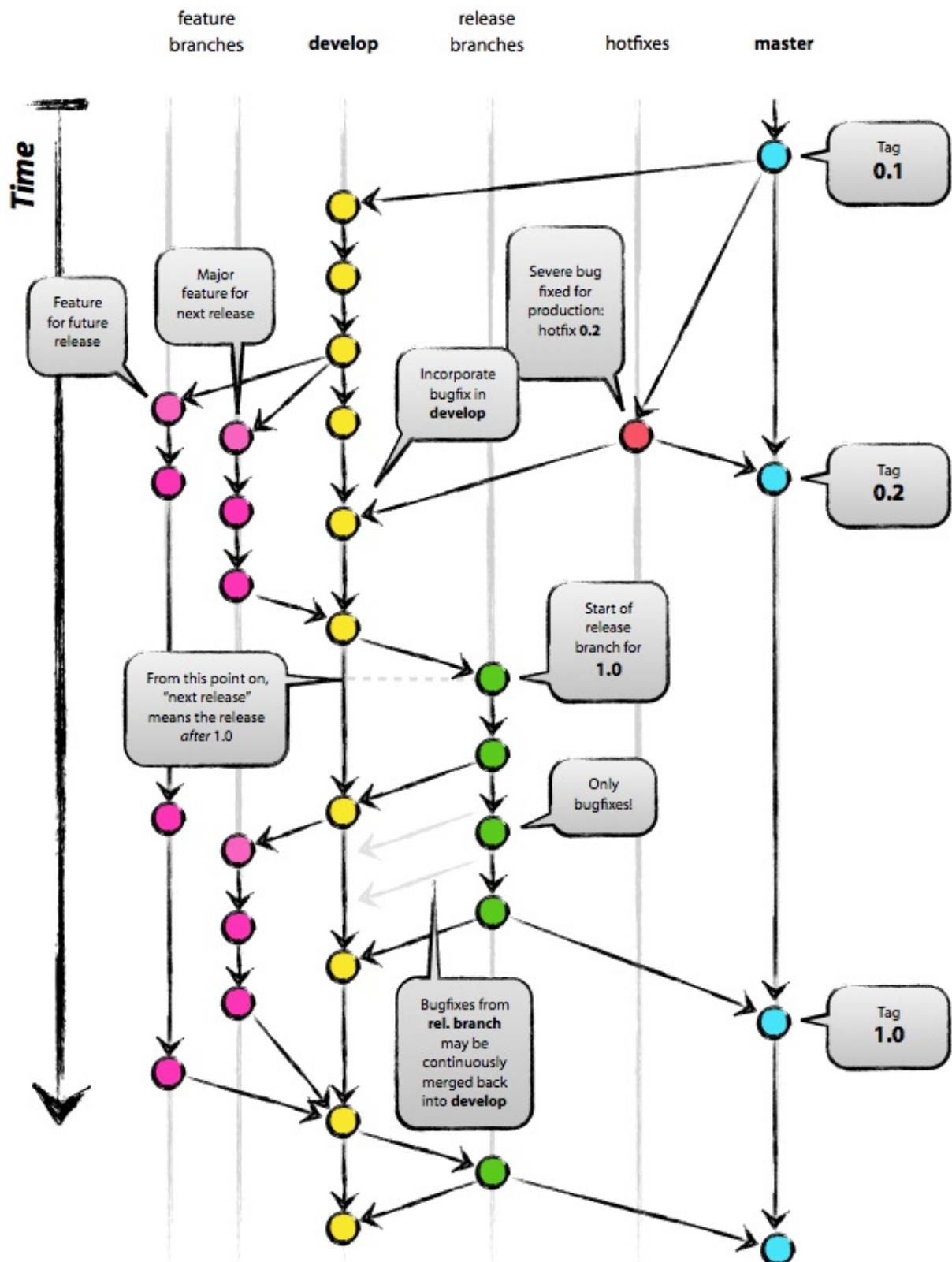
版权声明：本文为 stormzhang 原创文章，可以随意转载，但必须在明确位置注明出处！！！

我们都知道，在 git 的分支功能相对 svn 确实方便许多，而且也非常推荐使用分支来做开发。我的做法是每个项目都有2个分支，master 和 develop. master 分支是主分支，保证程序有一个 稳定版本，develop 则是开发用的分支，几乎所有的功能开发，bug 修复都在这个分支上，完成后 再合并回 master.

但是情况并不是这么简单，有时当我们正在开发一个功能，但程序突然出现 bug 需要及时去修复的时候，这时要切回 master 分支，并基于它创建一个 hotfix 分支。有时我们在开发一个功能时，需要停下来去开发另一个功能。而且所有这些问题都出现 的时候，发布也会成为比较棘手问题。

也就是说，git branch 功能很强大，但是没有一套模型告诉我们应该怎样在开发的时候善用 这些分支。于是有人就整理出了一套比较好的方案 [A successful Git branching model](#)，今天我们就一起来学习下这套方案。

简单来说，他将 branch 分成2个主要分支和3个临时的辅助分支：



主要分支

- master: 永远处在即将发布(production-ready)状态
- develop: 最新的开发状态

辅助分支

- feature: 开发新功能的分支, 基于 develop, 完成后 merge 回 develop
- release: 准备要发布版本的分支, 用来修复 bug. 基于 develop, 完成后 merge 回 develop 和 master
- hotfix: 修复 master 上的问题, 等不及 release 版本就必须马上上线. 基于 master, 完成后 merge 回 master 和 develop

作者还提供了 git-flow 命令工具:

```
$ git flow init
```

接着它会问你一系列的问题, 蛋定! 尽量使用它的默认值就好了:

```
No branches exist yet. Base branches must be created now.  
Branch name for production releases: [master]  
Branch name for "next release" development: [develop]  
How to name your supporting branch prefixes?  
Feature branches? [feature/]   
Release branches? [release/]   
Hotfix branches? [hotfix/]   
Support branches? [support/]   
Version tag prefix? []
```

完成后当前所在分支就变成 develop. 任何开发都必须从 develop 开始:

```
git flow feature start some_awesome_feature
```

完成功能开发之后:

```
git flow feature finish some_awesome_feature
```

该命令将会把feature/some_awesome_feature合并到develop分支, 然后删除功能(feature)分支。

将一个 feature 分支推到远程服务器:

```
git flow feature publish some_awesome_feature  
# 或者  
git push origin feature/some_awesome_feature
```

当你的功能点都完成时 (需要发布新版本了), 就基于develop创建一个发布(release)分支, 然后升级版本号并在最后发布日期前把Bug Fix掉吧:

```
$ git flow release start v0.1.0
```

当你在完成 (finish)一个发布分支时, 它会把你所作的修改合并到master分支, 同时合并回develop分支, 所以, 你不需要担心你的master分支比develop分支更加超前。

最后一件让git-flow显得威武的事情是它处理热修复 (即时的BugFix) 的能力, 你可以像其他分支一样地创建和完成一个热修复分支, 区别是它基于master分支, 因此你可以在产品出现问题时快速修复, 然后通过"finish"命令把修改合并回master和develop分支。

git flow on github: <https://github.com/nvie/gitflow>

推荐关注我的微信公众号 AndroidDeveloper

微信 id: googdev



不仅有技术与职场，还有生活与远方

扫码关注，听我扯淡

微博: @googdev

个人博客: <http://stormzhang.com>

stormzhang

01/29/2014

2016 Top 10 Android Library

过去的 2016 年，开源社区异常活跃，很多个人与公司争相开源自己的项目，让人眼花缭乱，然而有些项目只是昙花一现，有些项目却持久创造价值，为开发者提供了极大的便利，这些终究由时间来判断。今天，我就来整理一篇，我个人认为的 2016 年对 [Android](#) 开发有巨大帮助的，甚至改变了 [Android](#) 开发方式的开源库，但是，仅限个人认为，不具有任何权威性。

1. RxJava

地址：<https://github.com/ReactiveX/RxJava>

2016 年 [Android](#) 界最火的莫过于 RxJava 了，如果你还不知道 RxJava，你所在公司，或者你所在的项目还没有使用 RxJava，那真的是有点 out 了，RxJava 不仅大大简化了代码，甚至可以说改变了我们的开发方式。

RxJava 是一种函数式、响应式的异步操作库，它让你的代码更加简洁，真正的让你的代码写到爽！由于 RxJava 用过的都说好，基于此，GitHub 上衍生了一堆比如 RxAndroid、RxBus、RxPermission 等之类的开源库，足以说明它的影响力。

关于 RxJava 的文章网上一大堆，它的使用方法与好处我就不多说了，请自行去搜索了解，总之，身为 [Android](#) 开发者，到现在你还不知道 RxJava，简直了！

2. Retrofit

地址：<https://github.com/square/retrofit>

如果有人问我，[Android](#) 界最好用的网络请求库是什么？在之前可能会有人回答 android-async-http、Volley、OkHttp（准确说，OkHttp 是一个 http 请求客户端）之类的，但是 16 年过后，我会告诉你 Retrofit 是最好用的网络请求库。

Retrofit 完全 RESTful 风格的 api 网络请求库，解耦更彻底，源码设计超多的设计模式，值得大家学习，另外扩展性非常好，支持各种配置来满足你的需求，最重要的是，如果你的项目使用了 RxJava，那么 Retrofit 可以完美结合，我只能说 Perfect！再次验证了那句话：Square 出品，必属精品！

3. EventBus

地址：<https://github.com/greenrobot/EventBus>

试想这么一个场景，在 A 页面打开 B 页面，然后 B 页面打开了 C 页面，C 页面又打开了 D 页面，而且还需要传递参数，在 D 页面修改了一些信息，然后这些信息更新之后，A、B、C 页面很可能都需要对应的进行数据更新，碰到这种需求该怎么处理？

有人说用 `startActivityForResult()`，你可以试下，有多么难处理，还有人想到用广播，这个当然可以，因为广播是全局的，主要进行注册都可以通知到每一个页面，但是我很不喜欢用广播，每次用广播都要走那一套流程，很麻烦，而且很重。

而如果你知道 Eventbus，那么一切都非常的简单。

EventBus 是一个事件管理平台，以事件驱动的方式来简化事件传递逻辑，可以把它想象成轻量级的 BroadcastReceiver，不过，EventBus 并不是 16 年才开始进入大众视野的，很早就开源了，只是这个库太实用了，时至今日，它仍然很火，使用起来非常方便。

值得注意的是：EventBus 固然好用，但是不要过度使用，因为一旦你的代码大量使用 EventBus，会致使代码可读性稍差，而且出了问题不太好定位。所以建议只在特定的场景使用，切莫贪杯！

4. Glide、Fresco

图片加载可能跟网络请求一样，基本是所有 App 开发必备的功能，选择一款成熟稳定的图片加载库重要性不言而喻，目前主流的图片加载有 Picasso、Glide、Fresco，Glide 是 Google 员工基于 Picasso 基础上进行开发的，所以自然各方面比 Picasso 更有优势，而且支持 Gif，所以推荐大家优先选择 Glide 库，官方地址：

<https://github.com/bumptech/glide>

如果你的项目需要大量使用图片，比如是类似 Instagram 一类的图片社交 App，那么推荐使用 Fresco。Fresco 是 Facebook 作品，关于内存的占用优化更好，但是同时包也更大，门槛也更高，初级工程师不建议使用。官方地址：

<https://github.com/facebook/fresco>

这两款图片加载库，基本算是在 16 年使用最多，被认可最高的两个图片加载库了。

5. LeakCanary

地址：<https://github.com/square/leakcanary>

开发者最关心的除了完成功能外，其次就是会不会造成内存泄露了，其实检测内存泄露在 Java 领域有很多种方法与工具，但是针对 Android 都不够方便，而良心公司 Square 开源了一款针对 Android 平台的内存泄露检测工具 LeakCanary，集成简单，使用方便，平时测试的过程中就自动记录了内存泄露的位置，甚至帮你定位到代码级别，强烈推荐。

6. ButterKnife

地址：<https://github.com/JakeWharton/butterknife>

我想应该没有人没听过这个库了吧？ButterKnife 是 Android 之神 JakeWharton 的大作，已经开源了很长时间，然而在 2016 年它的使用热度依然不减，它可以让你避免无休止的 findViewById() 代码，具体用法我就不多说了，使用起来比较简单。

7. Realm

地址：<https://realm.io/>

说到 Realm 不得不提到一个 ORM 的概念。何为 ORM 呢？ORM 是 Object Relation Mapping 的缩写，翻译过来就是对象关系映射。这是相对于数据库的，我们知道 Android 中使用的数据库是 SQLite，而且 Android SDK 自带操作数据库的接口，而实际我们在使用的过程往往需要把查询的数据转换到一个 Java Object，也就是所谓的 Model，比如一般是这样：

```
public User selectWithId(int id) {
```

```

User user = null;
Cursor cursor = db.rawQuery("select * from users where id = ?", new String[]{id});
if (cursor != null && cursor.moveToFirst()) {
    int age = cursor.getInt(cursor.getColumnIndex("age"));
    String userName = cursor.getString(cursor.getColumnIndex("user_name"));
    ...
    user = new User(age, userName, ...);
    cursor.close();
}
return user;
}

```

操作起来是不是很麻烦？而且可读性超差，而有了 ORM 我们写代码可能会是类似这样：

查询数据是这样：

```

public User getUserId(int id) {
    return RealmResults<User> pups = realm.where(User.class)
        .lessThan("id", 2)
        .findAll();
}

```

是不是非常方便？代码写起来更像是面向对象，而不是一个个的裸写 SQL 了，这就是所谓的 ORM。

而 Android 界的 ORM 框架有很多，比如 GreenDao、SugarORM、ActiveAndroid 等等，但是我推荐大家的 ORM 框架以上都不是，是叫做 Realm。

Realm 是一种面向移动端的新型轻量数据库，而且是开源的，跟 SQLite 完全不一样，性能上秒杀 SQLite，支持 Java、Android、iOS 各平台，我们在实际项目中采用过，体验下来各方面都很不错，所以推荐大家尝试下 Realm。

8. Dagger 2

地址：<https://github.com/google/dagger>

依赖注入的概念估计大家都听过，不理解的不妨搜索了解下，Android 领域比较著名的依赖注入库莫过于 Dagger 了，基于注解，使用起来异常方便。

Dagger 起初是 Square 开源的，后来 Google 在此技术上进行了改进与优化，去除了反射，编译时进行依赖注入，性能上有大幅提升，取名 Dagger 2，Square 之前开源的 Dagger 已不建议使用。其实之前大家对 Dagger 的关注程度没有那么高，一般都是属于中、高级工程师才会关注使用，但是 16 年 Android 的架构被提上日程，各种 MVP、MVVM、Clean 架构等讨论的较多，而 Dagger 作为承载这些架构重要的一环被越来越多的开发者使用，所以 16 年我们看到 Dagger 的身影越来越多，所以，Dagger 被我列为 16 年还算比较火，比较实用的 Top 10 Android Library。

9. android-architecture

地址：<https://github.com/googlesamples/android-architecture>

上面说了，16 年 Android 架构被越来越多的开发者关注，国内外关于架构的探讨比较活跃，大家熟知的 MVC、MVP、MVVM、Clean 等，就在大家争执哪个更好，Android 开发到底该怎样架构的时候，Google 开源了一个 Android 架构的官方指导，涉及 mvp、mvp-loaders、databinding、mvp-clean、mvp-dagger、mvp-contentproviders、mvp-rxjava 等，分别在各自指定的分支下，有非常大的参考意义，可以算是 Android 界的一大步。

10. awesome-android-ui

地址：<https://github.com/wasabeef/awesome-android-ui>

Android 开发中除了我们以上用到的各种实用库之外，我们往往还会涉及到各种 UI 效果的实现，对于移动开发，界面开发其中是很重要的一环，而 16 年针对 Android 开发有人开源整理了这么一个库，里面网罗了所有你见过的、没见过的各种 UI 效果，涉及 Material、Layout、Button、List、ViewPager、Dialog、Menu、Parallax、Progress 等等，而且有相对应的截图、gif 展示，以后应对设计师各种效果的时候有很大的参考帮助作用。

以上就是我总结的，我个人认为的，在 2016 年的 Android 开发中，比较实用的、对你的开发有很大帮助的一些 Android Library，除了对你们的开发效率有提升之外，还能够了解其原理，阅读其优秀源码，参考其代码设计，是绝佳的一份学习资料，希望每个人在 2017 年都能在技术上取得很大进步！

本文原创发布于微信公众号 AndroidDeveloper，欢迎关注，第一时间获取更多原创分享。



GIT常用命令备忘

版权声明：本文为 stormzhang 原创文章，可以随意转载，但必须在明确位置注明出处！！！

Git配置

```
git config --global user.name "storm"
git config --global user.email "stormzhang.dev@gmail.com"
git config --global color.ui true
git config --global alias.co checkout # 别名
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.br branch
git config --global core.editor "vim" # 设置Editor使用vim
git config --global core.quotepath false # 设置显示中文文件名
```

用户的git配置文件~/.gitconfig

Git常用命令

查看、添加、提交、删除、找回，重置修改文件

```
git help <command> # 显示command的help
git show # 显示某次提交的内容
git show $id

git co -- <file> # 抛弃工作区修改
git co . # 抛弃工作区修改

git add <file> # 将工作文件修改提交到本地暂存区
git add . # 将所有修改过的工作文件提交暂存区

git rm <file> # 从版本库中删除文件
git rm <file> --cached # 从版本库中删除文件，但不删除文件

git reset <file> # 从暂存区恢复到工作文件
git reset -- . # 从暂存区恢复到工作文件
git reset --hard # 恢复最近一次提交的状态，即放弃上次提交后的所有本次修改

git ci <file>
git ci .
git ci -a # 将git add, git rm和git ci等操作都合并在一起做
git ci -am "some comments"
git ci --amend # 修改最后一次提交记录

git revert <$id> # 恢复某次提交的状态，恢复动作本身也创建了一次提交对象
git revert HEAD # 恢复最后一次提交的状态
```

查看文件diff

```
git diff <file> # 比较当前文件和暂存区文件差异
git diff
git diff <$id1> <$id2> # 比较两次提交之间的差异
```

```
git diff <branch1>..<branch2> # 在两个分支之间比较  
git diff --staged # 比较暂存区和版本库差异  
git diff --cached # 比较暂存区和版本库差异  
git diff --stat # 仅仅比较统计信息
```

查看提交记录

```
git log  
git log <file> # 查看该文件每次提交记录  
git log -p <file> # 查看每次详细修改内容的diff  
git log -p -2 # 查看最近两次详细
```

tig

Mac上可以使用tig代替diff和log, brew install tig

Git 本地分支管理

查看、切换、创建和删除分支

```
git br -r # 查看远程分支  
git br <new_branch> # 创建新的分支  
git br -v # 查看各个分支最后提交信息  
git br --merged # 查看已经被合并到当前分支的分支  
git br --no-merged # 查看尚未被合并到当前分支的分支  
  
git co <branch> # 切换到某个分支  
git co -b <new_branch> # 创建新的分支, 并且切换过去  
git co -b <new_branch> <branch> # 基于branch创建新的new_branch  
  
git co $id # 把某次历史提交记录checkout出来, 但无分支信息, 切换到其他分支会自动删除  
git co $id -b <new_branch> # 把某次历史提交记录checkout出来, 创建成一个分支  
  
git br -d <branch> # 删除某个分支  
git br -D <branch> # 强制删除某个分支 (未被合并的分支被删除的时候需要强制)
```

分支合并和rebase

```
git merge <branch> # 将branch分支合并到当前分支  
git merge origin/master --no-ff # 不要Fast-Foward合并, 这样可以生成merge提交  
  
git rebase master <branch> # 将master rebase到branch, 相当于:  
git co <branch> && git rebase master && git co master && git merge <branch>
```

Git补丁管理(方便在多台机器上开发同步时用)

```
git diff > ../sync.patch # 生成补丁  
git apply ../sync.patch # 打补丁  
git apply --check ../sync.patch # 测试补丁能否成功
```

Git暂存管理

```
git stash          # 暂存  
git stash list    # 列所有stash  
git stash apply   # 恢复暂存的内容  
git stash drop    # 删除暂存区  
git stash clear
```

Git远程分支管理

```
git pull          # 抓取远程仓库所有分支更新并合并到本地  
git pull --no-ff  # 抓取远程仓库所有分支更新并合并到本地, 不要快进合并  
git fetch origin  # 抓取远程仓库更新  
git merge origin/master # 将远程主分支合并到本地当前分支  
git co --track origin/branch # 跟踪某个远程分支创建相应的本地分支  
git co -b <local_branch> origin/<remote_branch> # 基于远程分支创建本地分支, 功能同上  
  
git push          # push所有分支  
git push origin master # 将本地主分支推到远程主分支  
git push -u origin master # 将本地主分支推到远程(如无远程主分支则创建, 用于初始化远程仓库)  
git push origin <local_branch> # 创建远程分支, origin是远程仓库名  
git push origin <local_branch>:<remote_branch> # 创建远程分支  
git push origin :<remote_branch> #先删除本地分支(git br -d <branch>), 然后再push删除远程分支
```

Git远程仓库管理

```
git remote -v      # 查看远程服务器地址和仓库名称  
git remote show origin # 查看远程服务器仓库状态  
git remote add origin git@github:stormzhang/demo.git      # 添加远程仓库地址  
git remote set-url origin git@github.com:stormzhang/demo.git # 设置远程仓库地址(用于修改远程仓库地址)
```

创建远程仓库

```
git clone --bare robbin_site robbin_site.git # 用带版本的项目创建纯版本仓库  
scp -r my_project.git git@git.csdn.net:~      # 将纯仓库上传到服务器上  
  
mkdir robbin_site.git && cd robbin_site.git && git --bare init # 在服务器创建纯仓库  
git remote add origin git@github.com:robbin/robbin_site.git      # 设置远程仓库地址  
git push -u origin master                      # 客户端首次提交  
git push -u origin develop # 首次将本地develop分支提交到远程develop分支, 并且track  
  
git remote set-head origin master    # 设置远程仓库的HEAD指向master分支
```

也可以命令设置跟踪远程库和本地库

```
git branch --set-upstream master origin/master  
git branch --set-upstream develop origin/develop
```

Via robbinfan

stormzhang

01/27/2014

2016 年最受欢迎的编程语言是什么？

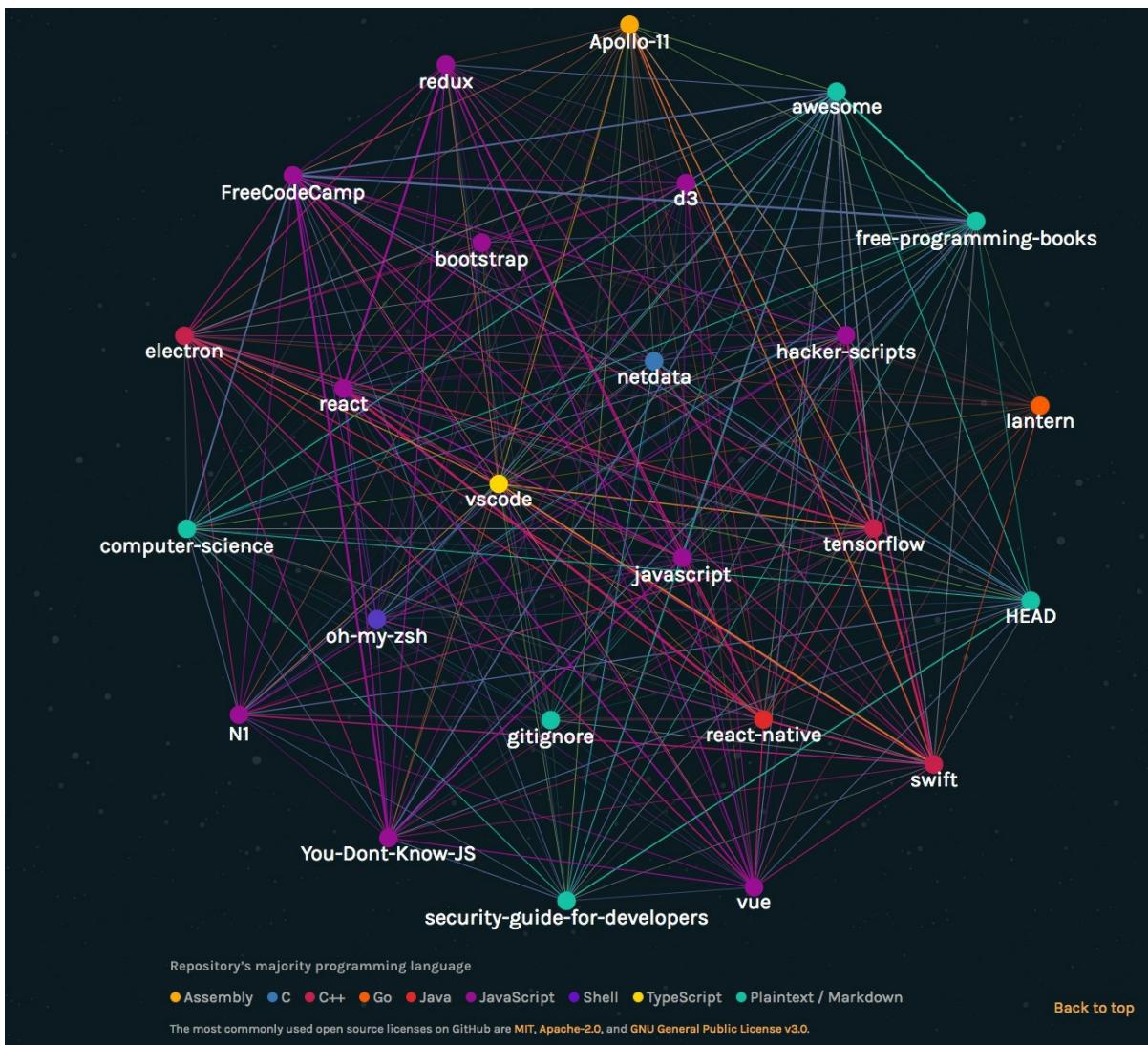
这两天 GitHub 对其官网进行了改版，紧接着又发布了一年一度的开源报告，我们程序员比较关心之后的趋势是什么，而 GitHub 毫无疑问代表了全世界编程领域的趋势，我们不妨先来解读下这份报告，然后再解答下你们关注的标题的答案。

事先声明，本篇文章的一些数据完全来自这份报告，地址在这里：

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

最流行的开源项目

首先发布的是过去一年在 GitHub 上最流行的开源项目，见下图：

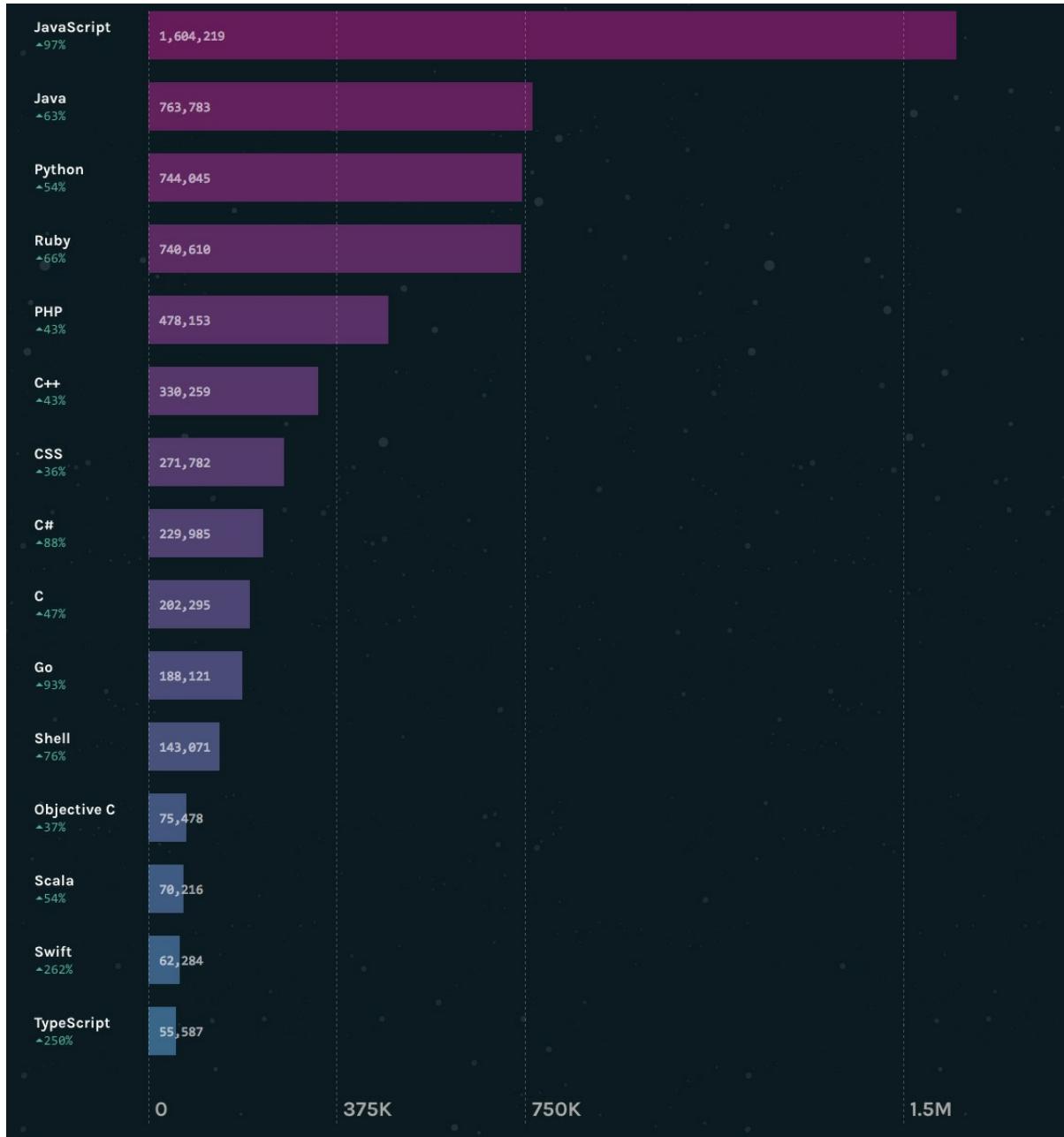


可以看到其中有不少是我在之前 GitHub 系列文章里介绍过的，如 awesome、free-program-books、React-native、on-my-zsh 等，不过令我没想到的是 lantern 竟然也入选了，足以说明全世界人们对自由上网的渴望，关于 lantern 是什么我不多说了，自己去了解吧。

最受欢迎的编程语言

这个世界有多少种编程语言你们知道么？我想没人说得清楚，GitHub 给出了答案。GitHub 上所有的开源项目包含了 316 种编程语言。不说不知道，一说吓一跳，要知道这世界上只有 226 个国家和地区，编程语言的数量超出了世界上国家的数量，有时候就在想，那么多不为人知的编程语言都是什么人在用？

要问 2016 年最受欢迎的编程语言是什么？同样 GitHub 也给出了答案。以下是 GitHub 根据过去 12 月提交的 PR 数量来排名的，虽然不完全准确，但是 PR 起码代表了项目的热度与欢迎度，还是值得可信的：



可以看到排名第一的是 [JavaScript](#)。我想有几方面的原因吧，一是本来 GitHub 上早期的一些开源项目都是 web 前端相关的，二是随着移动端各种跨平台框架的需求，js 被予以重任，如 [React Native](#)、[weex](#) 等，三是 js 领域各种框架层出不穷，如 [vue.js](#)、[angular.js](#)、[react.js](#) 等，所以 JavaScript 排名第一并不是很意外。所以有对 web 前端感兴趣的同学，js 是必备技能，想往这方面发展依然热度不减，而事实上国内需求目前对有经验的 web 前端工程师确实很缺乏，很多时候钱多活少离家近都招不到人。

另外老牌语言 Java 依然能排名第二蛮意外的，我想这其中很大部分是因为 Android 的发展让 Java 焕发了第二春。

紧接着是 Python、Ruby、PHP，这三种都是属于动态语言，对于我们 Android 开发所用的 Java 静态语言是不一样的，之前有人问过我想学习一门除了 Java 之外的语言，如果实在感兴趣的话我就建议学习下 Ruby 或者 Python，能从中了解到很多 Java 层面没接触过的知识。另外都说 PHP 是世界上最好的编程语言，这排名名不副实啊！

另外这份排名很有意思，元老级编程语言 C++、C 几乎每年都上榜，所以根本不用担心自己用的编程语言会过时，如果真那样的话 C++、C 那些程序员早都丢饭碗了。

最后一经出来就被热捧的 Swift 排名有点对不起大家对它的期待，今年仍然比不过亲兄弟的 Objective C，我觉得很大原因是因为亲爹 Apple 没有让开发者们强制使用 Swift，不过增长倒是很迅速，增长了 262%，相信这增长速度加上有个强大的爹，它的发展还是很期待的，只不过听说现在甚至还在改语法，所以还没有完全成熟，不要过于这么快就报太大的期待，不过如果 iOS 开发者们到现在还没有学习甚至了解就说不过去了。

所以，2016 年最受欢迎的编程语言是 JavaScript！

PS：作为 Android 开发者也蛮高兴的，毕竟我们所用的编程语言 Java 是 JavaScript 他哥！

开源贡献最多的组织

打死我都想不到 2016 年对开源贡献最多的竟然是微软，一向封闭为主的微软今年发力开源社区，竟然超越了 Google、Facebook，加上国内很大巨头也纷纷在开源社区发力，别的不说，就说 Android 界吧，今年包括腾讯、阿里等纷纷推出各自的开源项目，可能真的说明拥抱开源，才是王道吧！

Organizations with the most open source contributors

	Microsoft	16,419
	facebook	15,682
	docker	14,059
	angular	12,841
	google	12,140
	atom	9,698
	FontAwesome	9,617
	elastic	7,220
	Apache	6,999
	npm	6,815

GitHub 新增用户

GitHub 已经有超过 520 万的用户和超 30 万的组织。而今年，中国是新用户增长最多的国家，比 15 年增长快翻了一番，而这其中，身为一个 Google、GitHub 真爱粉，我觉得我也出了一把力(装逼完成，逃…)



当然还有很多其他有意思的数据，这里就不一一详细介绍了，感兴趣的不妨到[这里](#)去看下。

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

最后，GitHub 的这份报告代表着过去的数据，不过对于我们对未来的技术趋势判断有一定参考意义，所有编程从业者都有必要关注下这份报告，当然文中涉及到的一些观点纯属个人，不代表官方与任何组织，欢迎交流。

推荐阅读

[从0开始学习 GitHub 系列之「初识 GitHub」](#)

[从0开始学习 GitHub 系列之「加入 GitHub」](#)

[从0开始学习 GitHub 系列之「Git 速成」](#)

[从0开始学习 GitHub 系列之「向GitHub 提交代码」](#)

[从0开始学习 GitHub 系列之「Git 进阶」](#)

[从0开始学习 GitHub 系列之「团队合作利器 BRANCH」](#)

[从0开始学习 GitHub 系列之「如何发现优秀的开源项目？」](#)

本文原创发布于[微信公众号 AndroidDeveloper](#)，转载请务必注明出处！

不比较空白字符

在任意 diff 页面的 URL 后加上 `?w=1`，可以去掉那些只是空白字符的改动，使你能更专注于代码改动。

```
diff --git a/secrets.md b/secrets.md
--- a/secrets.md
+++ b/secrets.md
@@ -3,4 +3,6 @@ Over the years we've added
@@ -4,4 +4,5 @@ huge features, sometimes we
@@ -5 +6 @@ - Let's talk about some c
@@ -6 +7 @@ +Let's talk about some c
@@ -7 +8 @@ +
@@ -8 +9 @@ +### Whitespace
```

详见 [GitHub secrets](#).

调整 Tab 字符所代表的空格数

在 diff 或文件的 URL 后面加上 `?ts=4`，这样当显示 tab 字符的长度时就会是 4 个空格的长度，不再是默认的 8 个空格。`ts` 后面的数字还可以根据你个人的偏好进行修改。这个技巧不适用于 Gists，或者以 Raw 格式查看文件，但有浏览器扩展插件可以帮你自动调整: [Chrome 扩展](#)，[Opera 扩展](#)。

下面以一个 Go 语言源文件为例，看看在 URL 里添加 `?ts=4` 参数的效果。添加前：

```
file | 69 lines (57 sloc) | 1.86 kb
Open Edit Raw Blame History Delete
1 package flint
2
3 import (
4     "path/filepath"
5 )
6
7 type lintError struct {
8     Level    int
9     Message string
10 }
11
```

... 添加后的样子：

```
file | 69 lines (57 sloc) | 1.86 kb
Open Edit Raw Blame History Delete
1 package flint
2
3 import (
4     "path/filepath"
5 )
6
7 type lintError struct {
8     Level    int
9     Message string
10 }
```

查看用户的全部 Commit 历史

在 Commits 页面 URL 后加上 `?author={user}` 查看用户全部的提交。

```
https://github.com/rails/rails/commits/master?author=dhh
```

The screenshot shows the GitHub Commits page for the user dhh. At the top, there is a dropdown menu showing 'branch: master'. Below it, a section titled 'Commits on Jan 13, 2015' lists four commits:

- Merge pull request #18476 from Alamoz/scaffold_index_view_grammar ... 78a4884
- Stop promoting rack-cache usage at the moment (not so common or import... 1302edf
- Show how to change the queuing backend for ActiveJob in production 6463495
- Set all asset options together b9b28d8

Each commit includes a small profile picture of dhh, the commit message, the date it was authored, and the commit hash.

Below this, another section titled 'Commits on Jan 9, 2015' shows one commit:

- Merge pull request #18413 from brainopia/automatic_inverse_of_for_be... 6eb499f

[深入了解提交视图之间的区别](#)

仓库克隆

当克隆仓库时可以不要那个 `.git` 后缀。

```
$ git clone https://github.com/tiimgreen/github-cheat-sheet
```

[更多对 Git `clone` 命令的介绍.](#)

分支

将某个分支与其他所有分支进行对比

当你查看某个仓库的分支（Branches）页面（紧挨着 Commits 链接）时

```
https://github.com/{user}/{repo}/branches
```

你会看到一个包含所有未合并的分支的列表。

在这里你可以访问分支比较页面或删除某个分支。

The screenshot shows the GitHub repository 'rails / rails' on the 'Branches' tab. The 'Default branch' is 'master'. Below it, there are five active branches listed: '4-2-stable', '4-1-stable', 'fix_nested_transactions_for_re...', '3-2-stable', and '4-0-stable'. Each branch entry includes a 'Compare' button.

比较分支

如果要在 GitHub 上直接比较两个分支，可以使用如下形式的 URL：

```
https://github.com/{user}/{repo}/compare/{range}
```

其中 `{range} = master...4-1-stable`

例如：

```
https://github.com/rails/rails/compare/master...4-1-stable
```

The screenshot shows the GitHub comparison page for 'master...4-1-stable'. It displays commit statistics: 672 commits, 509 files changed, and 5 commit comments. A note says 'This comparison is big! We're only showing the most recent 250 commits'. The list of commits shows several merges from 'rafaelfranca' with commit IDs like 6413eb4, 0769465, 1a12bee, 8f76511, a44feed, and ae0d952.

`{range}` 参数还可以使用下面的形式：

```
https://github.com/rails/rails/compare/master@{1.day.ago}...master
https://github.com/rails/rails/compare/master@{2014-10-04}...master
```

日期格式 `YYYY-MM-DD`

The screenshot shows a GitHub commit comparison page. At the top, it shows two branches: `master@{2014-10-04}` and `master`. Below this, there are summary statistics: 130 commits, 123 files changed, 5 comments, and 39 contributors. There are tabs for `Commits`, `Files changed`, and `Commit comments`. The main area lists commits grouped by date:

- Aug 07, 2013**: emre-basala - Add tests to ActiveSupport::XmlMini `to_tag` method (green checkmark, 05d7cde)
- Nov 17, 2013**: iantropov - Fix insertion of records for `hmt` association with scope, fix #3548 (red X, ec09280)
- Jan 07, 2014**: roderickvd - Auto-generate stable fixture UUIDs on PostgreSQL. ... (green checkmark, 9330631)

在 `diff` 和 `patch` 页面里也可以比较分支：

```
https://github.com/rails/rails/compare/master...4-1-stable.diff
https://github.com/rails/rails/compare/master...4-1-stable.patch
```

[了解更多关于基于时间的 Commit 比较.](#)

比较不同派生库的分支

想要对派生仓库（Forked Repository）之间的分支进行比较，可以使用如下的 URL：

```
https://github.com/user/repo/compare/{foreign-user}:{branch}...{own-branch}
```

例如：

```
https://github.com/rails/rails/compare/byroot:master...master
```

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

[Create Pull Request](#) Open a Pull Request for this comparison to discuss and review your changes with others. [?](#)

6,802 commits 309 files changed 14 comments 64 contributors

Commits Files changed Commit comments

This comparison is big! We're only showing the most recent 250 commits

Apr 02, 2014

- kastiglione PostgreSQL, Support for materialized views. [Dave Lee & Yves Senn] ... ✓ def6071
- rajcybage We can conditional define the tests depending on the adapter or ... ✗ ee36af1
- rafaelfranca Merge pull request #14565 from rajcybage/conditional_test_cases ... ✓ c82483a
- rwz DRY AS::SafeBuffer a bit using existing helper ✓ 8482895
- alex88 Fixed small documentation typo ... ✗ 8ae3f24
- rafaelfranca Merge pull request #14568 from alex88/patch-1 ... ✓ 3bcc51a

Gists

Gists 方便我们管理代码片段，不必使用功能齐全的仓库。

GitHub Gist Search... Discover Gists rafalchmiel

PUBLIC tiimgreen / app.rb Created 4 days ago

A simple Ruby program.

Gist Detail Revisions 1

app.rb Ruby ↗ ↘

```
1 puts 'Hello World'
```

Download Gist Clone this gist https://gist.github.com/tiimgreen/10545811

Embed this gist <script src="https://gist.github.com/tiimgreen/10545811.js"></script>

Link to this gist https://gist.github.com/tiimgreen/10545811

rafalchmiel commented 2 days ago Wow, dat is some engineering.

Write Preview Comments are parsed with GitHub Flavored Markdown

Leave a comment Add Comment

Gist 的 URL 后加上 `.pibb` (像这样) 可以得到便于嵌入到其他网站的 HTML 代码。

Gists 可以像任何标准仓库一样被克隆。

```
$ git clone https://gist.github.com/tiimgreen/10545811
```

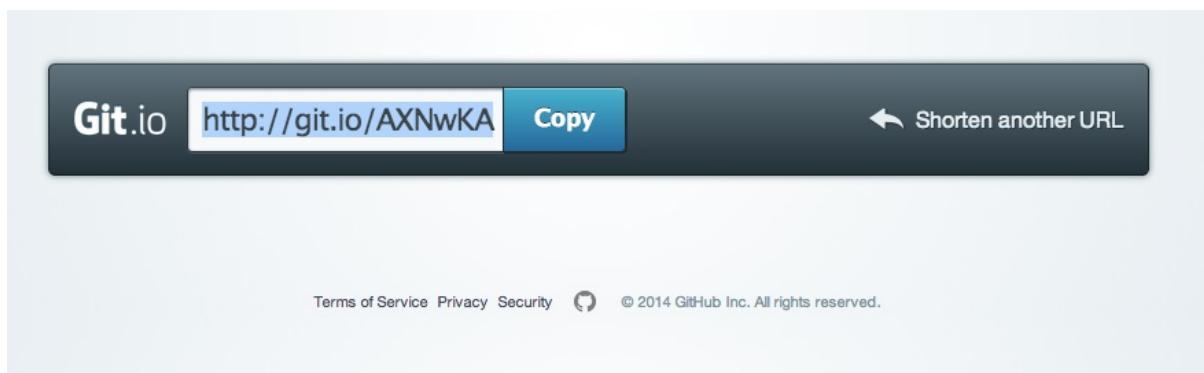
这意味着你可以像 Github 仓库一样修改和更新 Gists :

```
$ git commit  
$ git push  
Username for 'https://gist.github.com':  
Password for 'https://tiimgreen@gist.github.com':
```

但是， Gists 不支持目录。所有文件都必须添加在仓库的根目录下。 [进一步了解如何创建 Gists.](#)

Git.io

Git.io是 Github 的短网址服务。



你可以通过 Curl 命令以普通 HTTP 协议使用它：

```
$ curl -i http://git.io -F "url=https://github.com/..."  
HTTP/1.1 201 Created  
Location: http://git.io/abc123  
  
$ curl -i http://git.io/abc123
```

```
HTTP/1.1 302 Found
Location: https://github.com/...
```

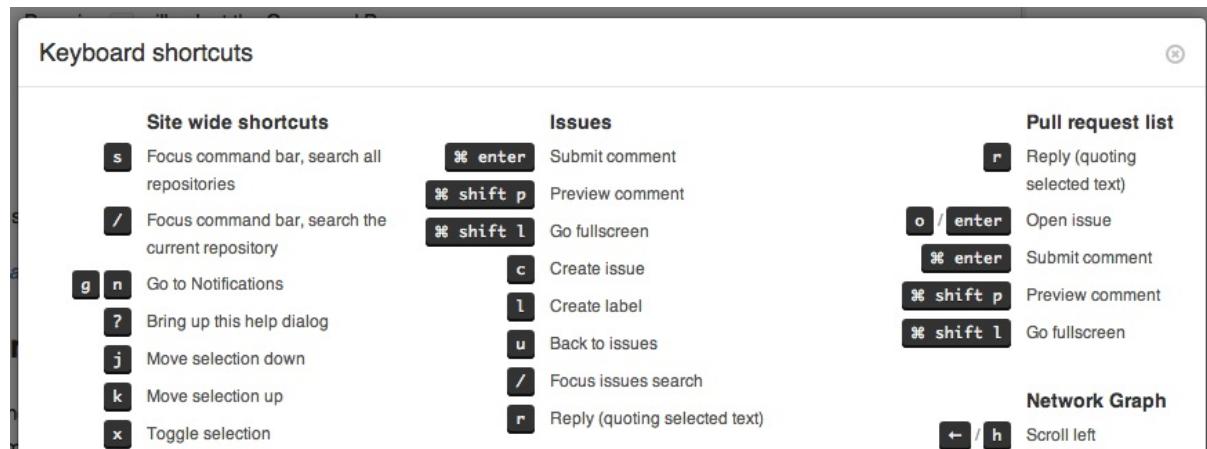
[进一步了解 Git.io.](#)

键盘快捷键

在仓库页面上提供了快捷键方便快速导航。

- 按 `t` 键打开一个文件浏览器。
- 按 `w` 键打开分支选择菜单。
- 按 `s` 键聚焦光标到当前仓库的搜索框。此时按退格键就会从搜索当前仓库切换到搜索整个 Github 网站。
- 按 `l` 键编辑 Issue 列表页的标签。
- 查看文件内容时** (如：<https://github.com/tiimgreen/github-cheat-sheet/blob/master/README.md>)，按 `y` 键将会冻结这个页面，这样就算代码被修改了也不会影响你当前看到的。

按 `?` 查看当前页面支持的快捷键列表：



[进一步了解可用的搜索语法.](#)

整行高亮

在代码文件地址 URL 后加上 `#L52` 或者单击行号 52 都会将第 52 行代码高亮显示。

多行高亮也可以，比如用 `#L53-L60` 选择范围，或者按住 `shift` 键，然后再点击选择的两行。

```
https://github.com/rails/rails/blob/master/activemodel/lib/active\_model.rb#L53-L60
```

```

43   autoload :Serialization
44   autoload :TestCase
45   autoload :Translation
46   autoload :Validations
47   autoload :Validator
48
49   eager_autoload do
50     autoload :Errors
51   end
52
53   module Serializers
54     extend ActiveSupport::Autoload
55
56     eager_autoload do
57       autoload :JSON
58       autoload :Xml
59     end
60   end
61
62   def self.eager_load!
63     super
64     ActiveModel::Serializers.eager_load!
65   end
66 end
67
68 ActiveSupport.on_load(:i18n) do
69   I18n.load_path << File.dirname(__FILE__) + '/active_model/locale/en.yml'
70 end

```

用 Commit 信息关闭 Issue

如果某个提交修复了一个 Issue，当提交到 master 分支时，提交信息里可以使用 `fix/fixes/fixed`，`close/closes/closed` 或者 `resolve/resolves/resolved` 等关键词，后面再跟上 Issue 号，这样就会关闭这个 Issue。

```
$ git commit -m "Fix screwup, fixes #12"
```

这将会关闭 Issue #12，并且在 Issue 讨论列表里关联引用这次提交。

```

$ git commit -m "Fix screwup, fixes #12"
$ git commit -m "Fix screwup, closes #12 in 36f4317"

```

这将会关闭 Issue #12，并且在 Issue 讨论列表里关联引用这次提交。

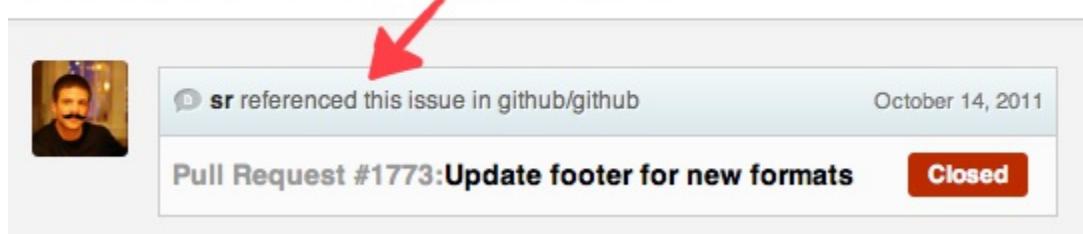
[进一步了解通过提交信息关闭 Issue.](#)

链接其他仓库的 Issue

如果你想引用到同一个仓库中的一个 Issue，只需使用井号 `#` 加上 Issue 号，这样就会自动创建到此 Issue 的链接。

要链接到其他仓库的 Issue，就使用 `{user}/{repo}#ISSUE_NUMBER` 的方式，例如 `tiimgreen/toc#12`。

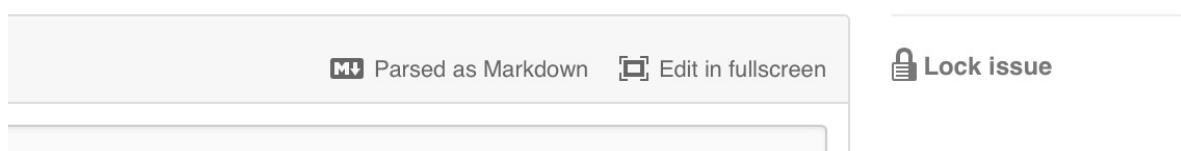
We should probably handle this with [github/enterprise#59](#)



A screenshot of a GitHub pull request page. At the top, there is a comment from a user named 'sr' with the text: "We should probably handle this with [github/enterprise#59](#)". A red arrow points to this comment. Below the comment, the pull request details are shown: "sr referenced this issue in github/github" and "October 14, 2011". The pull request itself is titled "Pull Request #1773: Update footer for new formats" and has a status of "Closed".

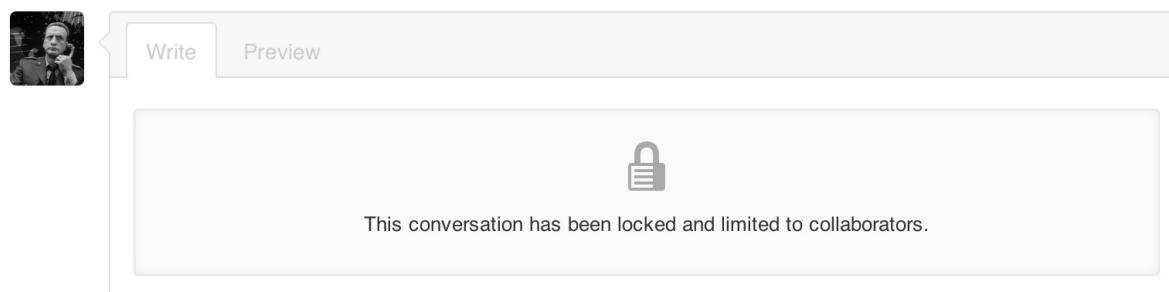
锁定项目对话功能

现在仓库的管理员和合作者可以将 Pull Requests 和 Issue 的评论功能关闭。



A screenshot of a GitHub issue comment form. At the top right, there is a button labeled "Lock issue" with a padlock icon. The rest of the interface includes a "Parsed as Markdown" button, an "Edit in fullscreen" button, and a text input area.

这样，不是项目合作者的用户就不能在这个项目上使用评论功能。



A screenshot of a GitHub issue comment form. On the left, there is a user profile picture. In the center, there are two tabs: "Write" and "Preview". Below them is a large text input area. Inside the text input area, there is a padlock icon and the text: "This conversation has been locked and limited to collaborators."

[进一步了解对话锁定功能.](#)

设置 CI 对每条 Pull Request 都进行构建

如果配置正确，[Travis CI](#) 会为每个你收到的 Pull Request 执行构建，就像每次提交也会触发构建一样。想了解更多关于 Travis CI 的信息，请参考 [Travis CI入门](#)。

The screenshot shows a GitHub pull request page for the repository 'octokit/octokit.rb'. The pull request is titled '#452 Lazy create test repos' and is merged into the 'master' branch. The commit message includes a link to '#429' and describes the changes as 'Checks for and creates test repositories before recording new cassettes.' The commit history shows two commits from 'joeyw': one adding some commits and another lazily creating test repositories. The status bar at the bottom indicates 'All is well — The Travis CI build passed'.

[进一步了解提交状态 API.](#)

Markdown 文件语法高亮

例如，可以像下面这样在你的 Markdown 文件里为 Ruby 代码添加语法高亮：

```
require 'tabbit'
table = Tabbit.new('Name', 'Email')
table.add_row('Tim Green', 'tiimgreen@gmail.com')
puts table.to_s
```

效果如下：

```
require 'tabbit'
table = Tabbit.new('Name', 'Email')
table.add_row('Tim Green', 'tiimgreen@gmail.com')
puts table.to_s
```

GitHub 使用 [Linguist](#) 做语言识别和语法高亮。你可以仔细阅读 [Languages YAML file](#)，了解有哪些可用的关键字。

[进一步了解 GitHub Flavored Markdown.](#)

表情符

可以在 Pull Requests, Issues, 提交消息, Markdown 文件里加入表情符。使用方法 :name_of_emoji:

```
:smile:
```

将输出一个笑脸：

:smile:

Github 支持的完整表情符号列表详见emoji-cheat-sheet.com 或 [scotch.io/All-Github-Emoji-Icons](https://scotch.io>All-Github-Emoji-Icons)。

Github 上使用最多的5个表情符号是：

1. :shipit:
2. :sparkles:
3. :-1:
4. :+1:
5. :clap:

图片 / GIF 动画

注释和README等文件里也可以使用图片和 GIF 动画：

```
![Alt Text](http://www.sheawong.com/wp-content/uploads/2013/08/keephatin.gif)
```

仓库中的图片可以被直接引用：

```
![Alt Text](https://github.com/{user}/{repo}/raw/master/path/to/image.gif)
```



所有图片都缓存在 Github，不用担心你的站点不能访问时就看不到图片了。

在 GitHub Wiki 中引用图片

有多种方法可以在 Wiki 页面里嵌入图片。既可以像上一条里那样使用标准的 Markdown 语法，也可以像下面这样指定图片的高度或宽度：

```
[[ http://www.sheawong.com/wp-content/uploads/2013/08/keephatin.gif | height = 100px ]]
```

结果：

Home (Preview)



README not found

While a README isn't a required part of an open source project, it is a very good idea to have one. READMEs are a great place to describe your project or add some documentation such as how to install or use your project. You might want to include contact information - if your project becomes popular people will want to help you out. See [GitHub's article](#) on creating repositories. See [Tom Preston-Werner's blog post](#) on README driven development. Also see the [Wikipedia article](#) on READMEs.

快速引用

在主题评论中引用之前某个人所说的，只需选中文本，然后按 `r` 键，想要的就会以引用的形式复制到你的输入框里。

The screenshot shows a GitHub comment interface. At the top, a comment by user `jakeboxer` is visible, reading: "Yep, I really like the second option—It puts exactly the right amount of emphasis on the content." Below this, a reply is being composed. The reply area has tabs for "Write" and "Preview". The "Write" tab is active. A text input field contains the quoted text: " Yep, I really like the second option—It puts exactly the right amount of emphasis on the content." Below the input field, there is a note: "Comments are parsed with GitHub Flavored Markdown". At the bottom of the reply area, there is a "Comment" button.

[进一步了解快速引用.](#)

粘贴剪贴板中的图片到评论

(仅适用于 Chrome 浏览器)

当截屏图片复制到剪贴板后 (mac 上用 `cmd-ctrl-shift-4`)，你可以用(`cmd-v / ctrl-v`)把图片粘贴到评论框里，然后它就会自动上传到 Github。

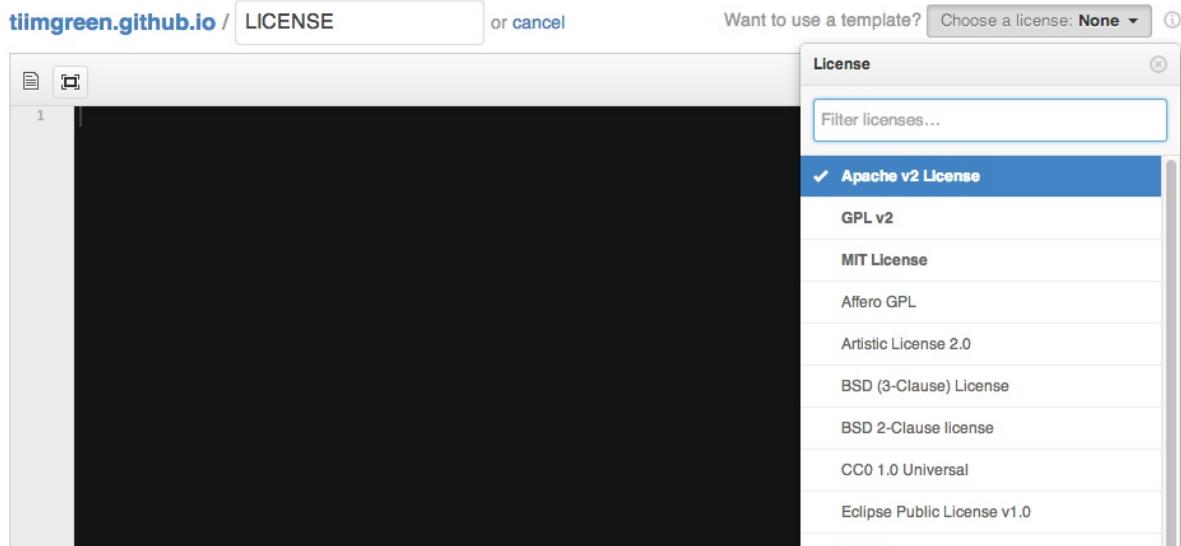


进一步了解在 issue 中使用附件

快速添加许可证文件

创建一个仓库时，Github会为你提供一个预置的软件许可列表：

对于已有的仓库，可以通过 web 界面创建文件来添加软件许可。输入 LICENSE 作为文件名后，同样可以从预置的列表中选择一个作为模板。



这个技巧也适用于 `.gitignore` 文件。

[进一步了解开源许可证](#)

任务列表

Issues 和 Pull requests 里可以添加复选框，语法如下（注意空白符）：

```
- [ ] Be awesome
- [ ] Prepare dinner
- [ ] Research recipe
- [ ] Buy ingredients
- [ ] Cook recipe
- [ ] Sleep
```

◀ TODO #1

Open AlexandreArpin opened this issue just now · 0 comments



AlexandreArpin commented just now

```
 Be awesome
 Prepare dinner
   Research recipe
   Buy ingredients
   Cook recipe
 Sleep
```

当项目被选中时，它对应的 Markdown 源码也被更新了：

```
- [x] Be awesome
- [ ] Prepare dinner
- [x] Research recipe
- [x] Buy ingredients
```

```
- [ ] Cook recipe  
- [ ] Sleep
```

[进一步了解任务列表.](#)

Markdown 文件中的任务列表

在完全适配Markdown语法的文件中可以使用以下语法加入一个只读的任务列表

```
- [ ] Mercury  
- [x] Venus  
- [x] Earth  
  - [x] Moon  
- [x] Mars  
  - [ ] Deimos  
  - [ ] Phobos
```

- [] Mercury
- [x] Venus
- [x] Earth
 - [x] Moon
- [x] Mars
 - [] Deimos
 - [] Phobos

[进一步了解 Markdown 文件中的任务列表](#)

相对链接

Markdown文件里链接到内部内容时推荐使用相对链接。

```
[Link to a header](#awesome-section)  
[Link to a file](docs/readme)
```

绝对链接会在 URL 改变时（例如重命名仓库、用户名改变，建立分支项目）被更新。使用相对链接能够保证你的文档不受此影响。

[进一步了解相对链接.](#)

GitHub Pages 的元数据与插件支持

在 Jekyll 页面和文章里，仓库信息可在 `site.github` 命名空间下找到，也可以显示出来，例如，使用 `{{ site.github.project_title }}` 显示项目标题。

Jemoji 和 jekyll-mentions 插件为你的 Jekyll 文章和页面增加了emoji和@mentions功能。

[了解更多 GitHub Pages 的元数据和插件支持.](#)

查看 YAML 格式的元数据

许多博客站点，比如基于 [Jekyll](#) 的 [GitHub Pages](#)，都依赖于一些文章头部的 YAML 格式的元数据。Github 会将其渲染成一个水平表格，方便阅读。

The screenshot shows a GitHub file page for a file named 'bare'. The file contains the following YAML configuration:

```
layout: bare
title: Git and GitHub Review for NYU
description: Git and GitHub Review for NYU
tags: notes, online, class
path: classnotes/2013-02-13-NYU-github-class.md
eventdate: 2013-02-13
```

Below the file content, there is a heading: "Your instructors for the evening are:" followed by a bulleted list:

- Matthew McCullough ([Twitter](#), [GitHub](#))
- Tim Berglund ([Twitter](#), [GitHub](#))

[进一步了解 在文档里查看 YAML 元数据.](#)

渲染表格数据

GitHub 支持将 `.csv` (逗号分隔) 和 `.tsv` (制表符分隔) 格式的文件渲染成表格数据。

The screenshot shows a GitHub file page for a file named '180.csv'. The file contains a CSV table of movie locations:

	Title	Release Year	Locations	Fun Facts	Production Company
1	180	2011	555 Market St.		SPI Cinemas
2	180	2011	Epic Roasthouse (399 Embarcadero...)		SPI Cinemas
3	180	2011	Mason & California Streets (Nob Hill)		SPI Cinemas
4	180	2011	Justin Herman Plaza		SPI Cinemas
5	180	2011	200 block Market Street		SPI Cinemas
6	180	2011	City Hall		SPI Cinemas
7	180	2011	Polk & Larkin Streets		SPI Cinemas
8	180	2011	Randall Musuem		SPI Cinemas
9	24 Hours on Craigslist	2005			Yerba Buena Productions
10	48 Hours	1982			Paramount Pictures
11	50 First Dates	2004	Rainforest Café (145 Jefferson Str...)		Columbia Pictures Corpora
12	A Jitney Elopement	1915	Golden Gate Park	During San Francisco's Gold Rush...	The Essanay Film Manufact
13	A Jitney Elopement	1915	20th and Folsom Streets		The Essanay Film Manufact
14	A Night Full of Rain	1978	San Francisco Chronicle (901 Mis...)	The San Francisco Zodiac Killer of...	Liberty Film

[进一步了解渲染表格数据.](#)

撤销 Pull Request

合并一个 Pull Request 之后，你可能会反悔：要么是这次 Pull Request 没什么用处，要么是还不到合并的时候。

此时可以通过 Pull Request 中的 Revert 按钮来撤销一个已合并的 Pull Request 中的 commit。按下按钮后将自动生成一个进行逆操作的 Pull Request。

A screenshot of a GitHub pull request interface. At the top, there's a list of commits by 'mdo and others' with their commit messages and status (green checkmark for successful, red X for failed). Below this, a commit by 'octocat' is shown, which has been merged into 'master'. A 'Revert' button is prominently displayed next to this commit, with a tooltip 'Create a new pull request to revert these changes'.

*[进一步了解“撤销”按钮](#)

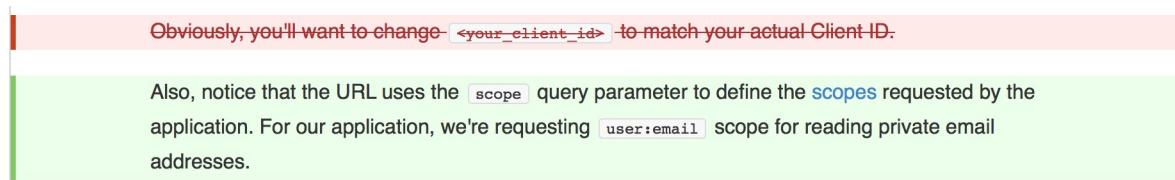
Diffs

可渲染文档的Diffs

Commit 和 Pull Request 里包含有 Github 支持的可渲染文档（比如 Markdown）会提供source 和 rendered 两个视图功能。



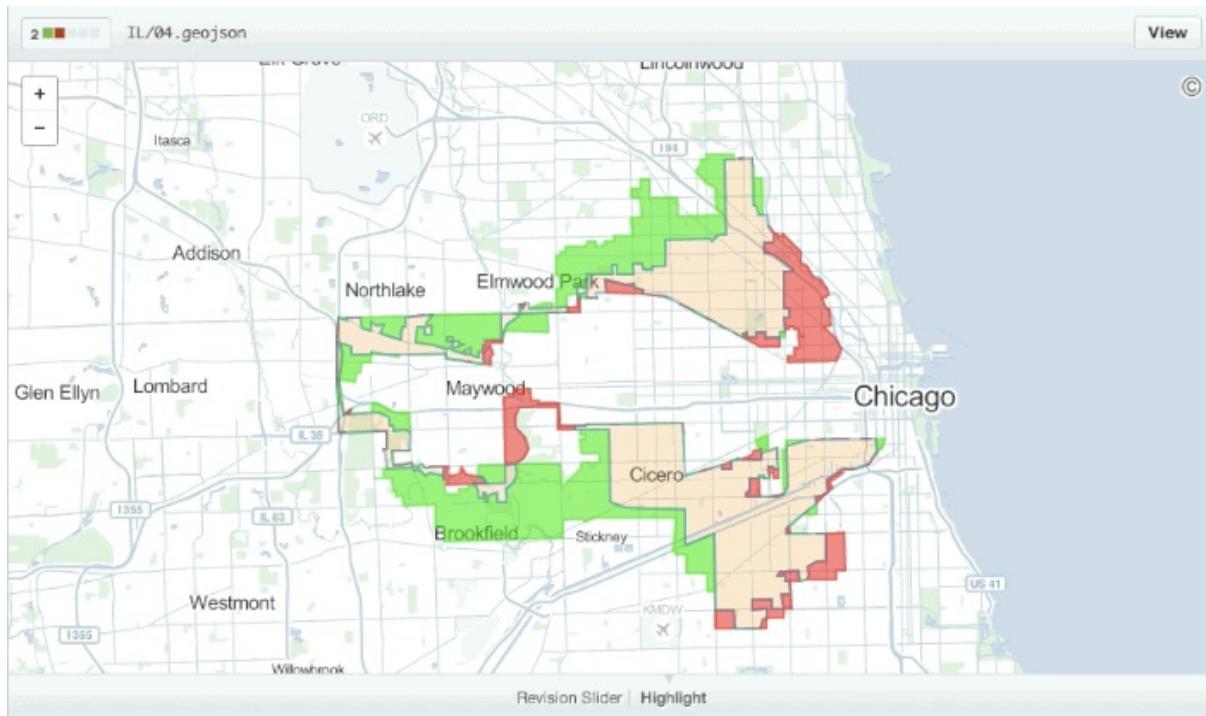
点击 "rendered" 按钮，看看改动在渲染后的显示效果。当你添加、删除或修改文本时，渲染纯文本视图非常方便。



[进一步了解渲染纯文本视图Diffs.](#)

可比较的地图数据

当你在GitHub上查看一个包含地理数据的 commit 或 pull request时，Github 将以可视化的方式对比版本之间的差异。



[进一步了解可比较的地图数据.](#)

在 Diff 中展开查看更多的上下文

你可以通过点击 diff 边栏里的 unfold 按钮来多显示几行上下文。你可以一直点击 unfold 按钮直到显示了文件的全部内容。这个功能在所有 GitHub 的 diff 功能中都可以使用。

```

94  +- (RACSignal *)enqueueRequest:(NSURLRequest *)request fetchAllPages:(BOOL)fetchAllPages;
95  +
82  96 // Enqueues a request to fetch information about the current user by accessing
83  97 // a path relative to the user object.
84  98 //
@2 -241,11 +255,13 @ - (id)initWithServer:(OCTServer *)server {
241 255     NSString *userAgent = self.class.userAgent;
242 256     if (userAgent != nil) [self setDefaultHeader:@"User-Agent" value:userAgent];
243 257     - self.parameterEncoding = AFJSONParameterEncoding;
244 258     - [self setDefaultHeader:@"Accept" value:@"application/vnd.github.beta+json"];
245 259     -
246 260     [AFHTTPRequestOperation addAcceptableStatusCodes:[NSSet indexSetWithIndex:OCTClientNotModifiedStatusCode]];
247 261     - [AFJSONRequestOperation addAcceptableContentTypes:[NSSet setWithObject:@"application/vnd.github.beta+json"]];
248 262     +
249 263     + NSString *contentType = [NSString stringWithFormat:@"application/vnd.github.%@+json", OCTClientAPIVersion];
250 264     + [self setDefaultHeader:@"Accept" value:contentType];
251 265     + [AFJSONRequestOperation addAcceptableContentTypes:[NSSet setWithObject:contentType]];
252 266     +
253 267     + self.parameterEncoding = AFJSONParameterEncoding;
254 268     [self registerHTTPOperationClass:AFJSONRequestOperation.class];
255 269     return self;
256 270 }

```

[进一步了解展开 Diff 上下文.](#)

获取 Pull Request 的 diff 或 patch 文件

在 Pull Request 的 URL 后面加上 `.diff` 或 `.patch` 的扩展名就可以得到它的 diff 或 patch 文件，例如：

```
https://github.com/tiimgreen/github-cheat-sheet/pull/15
https://github.com/tiimgreen/github-cheat-sheet/pull/15.diff
https://github.com/tiimgreen/github-cheat-sheet/pull/15.patch
```

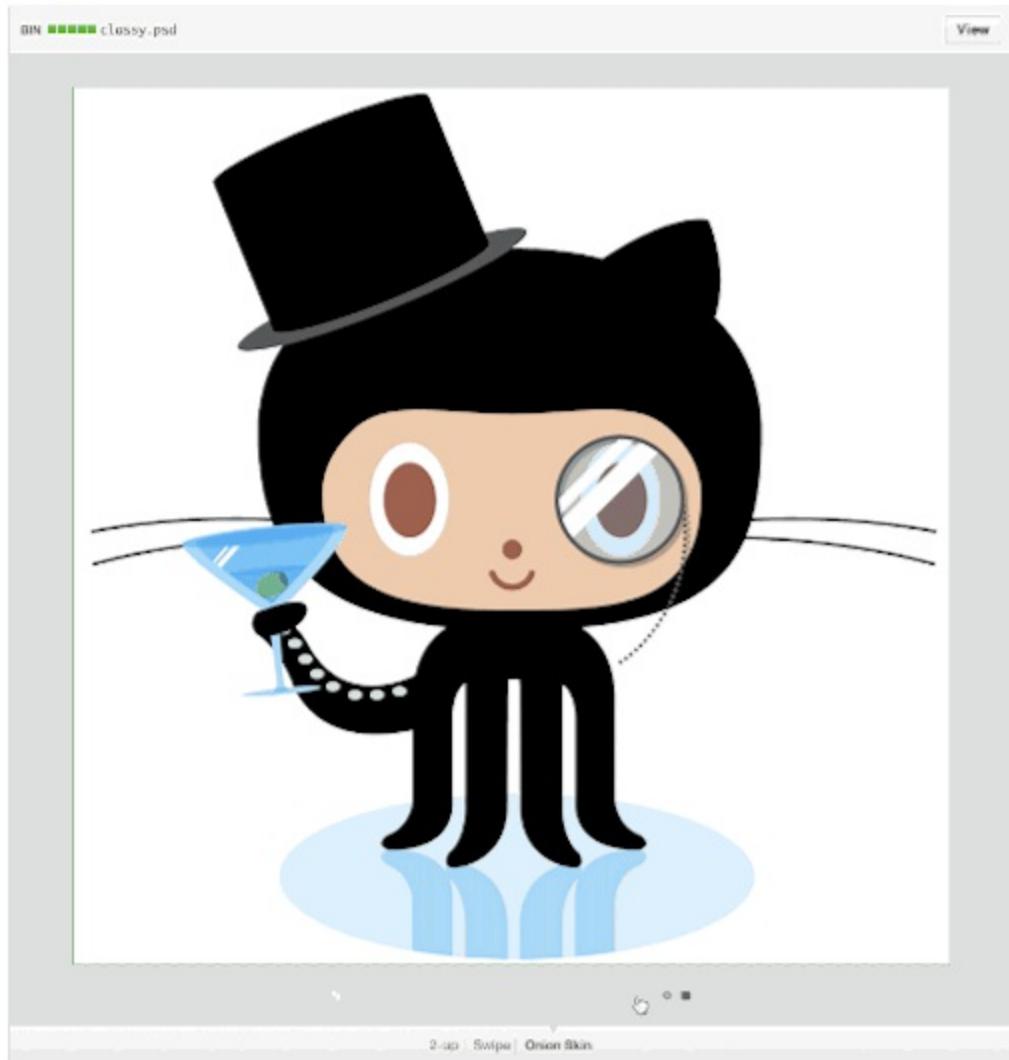
.diff 扩展会使用普通文本格式显示如下内容：

```
diff --git a/README.md b/README.md
index 88fcf69..8614873 100644
--- a/README.md
+++ b/README.md
@@ -28,6 +28,7 @@ All the hidden and not hidden features of Git and GitHub. This cheat sheet was i
- [Merged Branches](#merged-branches)
- [Quick Licensing](#quick-licensing)
- [TODO Lists](#todo-lists)
+- [Relative Links](#relative-links)
- [.gitconfig Recommendations](#gitconfig-recommendations)
  - [Aliases](#aliases)
  - [Auto-correct](#auto-correct)
@@ -381,6 +382,19 @@ When they are clicked, they will be updated in the pure Markdown:
- [ ] Sleep

(...)
```

显示图片以及比较图片

GitHub 可以显示包括 PNG、JPG、GIF、PSD 在内的多种图片格式并提供了几种方式来比较这些格式的图片文件版本间的不同。



[查看更多关于图片显示和比较](#)

Hub

[Hub](#) 是一个对 Git 进行了封装的命令行工具，可以帮助你更方便的使用 Github。

例如可以像下面这样进行克隆：

```
$ hub clone tiimgreen/toc
```

[查看更多 Hub 提供的超酷命令.](#)

贡献者指南

在仓库的根目录添加一个名为 `CONTRIBUTING` 的文件后，贡献者在新建 Issue 或 Pull Request 时会看到一个指向这个文件的链接。

Houston, we have a problem!

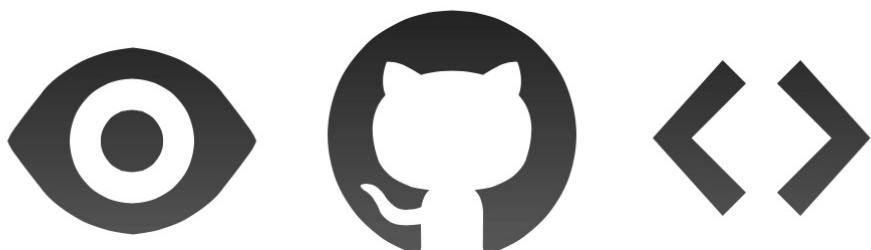
[Write](#) [Preview](#) Comments are parsed with GitHub Flavored Markdown

Leave a comment

[进一步了解贡献者指南.](#)

Octicons

GitHubs 图标库 (Octicons) 现已开源。



[进一步了解 GitHub 图标库](#)

GitHub 资源

内容	链接
探索 GitHub	https://github.com/explore
GitHub 博客	https://github.com/blog
GitHub 帮助	https://help.github.com/
GitHub 培训	http://training.github.com/
GitHub 开发者	https://developer.github.com/

GitHub 相关演讲视频

内容	链接
How GitHub Uses GitHub to Build GitHub	https://www.youtube.com/watch?v=qyz3jkOBbQY
Introduction to Git with Scott Chacon of GitHub	https://www.youtube.com/watch?v=ZDR433b0HJY
How GitHub No Longer Works	https://www.youtube.com/watch?v=gXD1lTW7iZI
Git and GitHub Secrets	https://www.youtube.com/watch?v=Foz9yvMkvIA
More Git and GitHub Secrets	https://www.youtube.com/watch?v=p50xsL-iVgU

Git

从工作区去除大量已删除文件

当用 `/bin/rm` 命令删除了大量文件之后，你可以用下面一条命令从工作区和索引中去除这些文件，以免一个一个的删除：

```
$ git rm $(git ls-files -d)
```

例如：

```
$ git status
On branch master
Changes not staged for commit:
  deleted:    a
  deleted:    c

$ git rm $(git ls-files -d)
rm 'a'
rm 'c'

$ git status
On branch master
Changes to be committed:
  deleted:    a
  deleted:    c
```

上一个分支

快速检出上一个分支：

```
$ git checkout -
# Switched to branch 'master'

$ git checkout -
# Switched to branch 'next'

$ git checkout -
# Switched to branch 'master'
```

[进一步了解 Git 分支.](#)

去除空白

Git StripSpace 命令可以:

- 去掉行尾空白符
- 多个空行压缩成一行
- 必要时在文件末尾增加一个空行

使用此命令时必须传入一个文件, 像这样 :

```
$ git strip-space < README.md
```

[进一步了解 Git `strip-space` 命令.](#)

检出 Pull Requests

对 Github 仓库来说, Pull Request 是种特殊分支, 可以通过以下多种方式取到本地:

取出某个特定的 Pull Request 并临时作为本地的 `FETCH_HEAD` 中以便进行快速查看更改(`diff`)以及合并(`merge`):

```
$ git fetch origin refs/pull/[PR-Number]/head
```

通过 refspec 获取所有的 Pull Request 为本地分支:

```
$ git fetch origin '+refs/pull/*:refs/remotes/origin/pr/*'
```

或在仓库的 `.git/config` 中加入下列设置来自动获取远程仓库中的 Pull Request

```
[remote "origin"]
fetch = +refs/heads/*:refs/remotes/origin/*
url = git@github.com:tiimgreen/github-cheat-sheet.git
```

```
[remote "origin"]
fetch = +refs/heads/*:refs/remotes/origin/*
url = git@github.com:tiimgreen/github-cheat-sheet.git
fetch = +refs/pull/*:refs/remotes/origin/pr/*
```

对基于派生库的 Pull Request, 可以通过先 `checkout` 代表此 Pull Request 的远端分支再由此分支建立一个本地分支:

```
$ git checkout pr/42 pr-42
```

操作多个仓库的时候, 可以在 Git 中设置获取 Pull Request 的全局选项。

```
git config --global --add remote.origin.fetch "+refs/pull/*:refs/remotes/origin/pr/*"
```

此时可以在任意仓库中使用以下命令：

```
git fetch origin
```

```
git checkout pr/42
```

[进一步了解如何本地检出 pull request.](#)

没有任何改动的提交

可以使用 `--allow-empty` 选项强制创建一个没有任何改动的提交：

```
$ git commit -m "Big-ass commit" --allow-empty
```

这样做在如下几种情况下是有意义的：

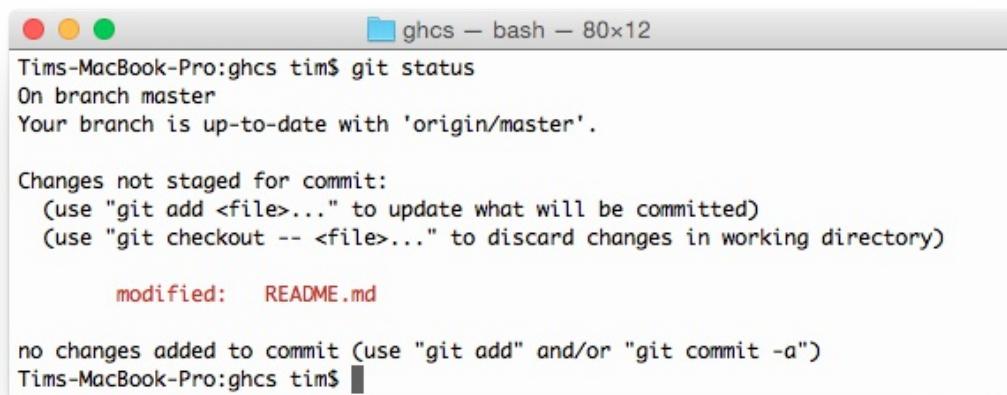
- 标记新的工作或一个新功能的开始。
- 记录对项目的跟代码无关的改动。
- 跟使用你仓库的其他人交流。
- 作为仓库的第一次提交，因为第一次提交后不能被 rebase：`git commit -m "init repo" --allow-empty` .

美化 Git Status

在命令行输入如下命令：

```
$ git status
```

可以看到：



The screenshot shows a terminal window titled "ghcs – bash – 80x12". The command "git status" was run, and the output is as follows:

```
Tims-MacBook-Pro:ghcs tim$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    (Use "git checkout -- <file>..." to discard changes in working directory)

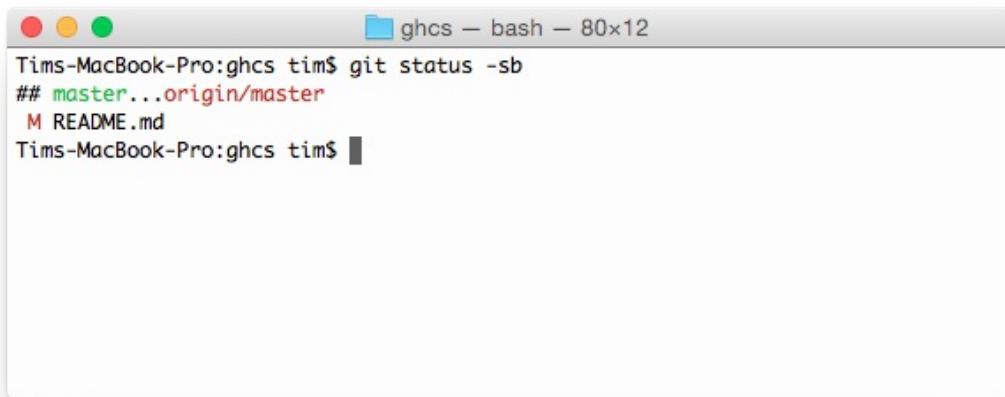
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
Tims-MacBook-Pro:ghcs tim$ █
```

加上 `-sb` 选项：

```
$ git status -sb
```

这会得到:



```
Tims-MacBook-Pro:ghcs tim$ git status -sb
## master...origin/master
 M README.md
Tims-MacBook-Pro:ghcs tim$
```

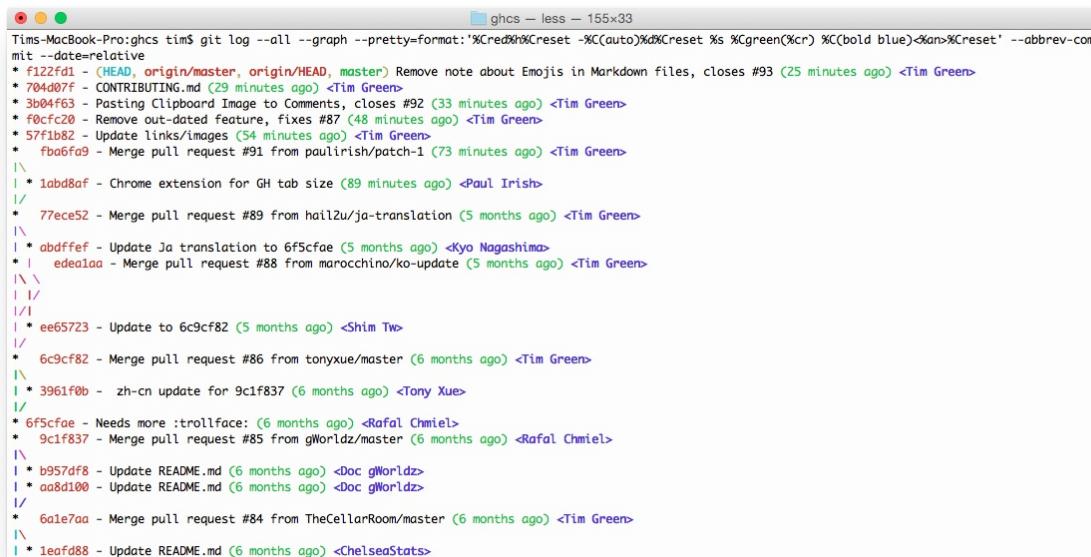
进一步了解 Git `status` 命令.

美化 Git Log

输入如下命令:

```
$ git log --all --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative
```

可以看到:



```
Tims-MacBook-Pro:ghcs tim$ git log --all --graph --pretty=format:'%Cred%h%Creset -%C(auto)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit --date=relative
* f122fd1 - (HEAD, origin/master, origin/HEAD, master) Remove note about Emojis in Markdown files, closes #93 (25 minutes ago) <Tim Green>
* 704d0f7 - CONTRIBUTING.md (29 minutes ago) <Tim Green>
* 3b00f63 - Pastin Clipboard Image to Comments, closes #92 (33 minutes ago) <Tim Green>
* f0cfc20 - Remove out-dated feature, fixes #87 (48 minutes ago) <Tim Green>
* 57fib82 - Update links/images (54 minutes ago) <Tim Green>
* fba6fd9 - Merge pull request #91 from paulirish/patch-1 (73 minutes ago) <Tim Green>
|\
| * 1abd8af - Chrome extension for GH tab size (89 minutes ago) <Paul Irish>
|/
* 77ece52 - Merge pull request #89 from hail2u/ja-translation (5 months ago) <Tim Green>
|\
| * abdffef - Update Ja translation to 6f5cfae (5 months ago) <kyo Nagashima>
* | edea1aa - Merge pull request #88 from marocchino/ko-update (5 months ago) <Tim Green>
| \
| /
|/
* ee65723 - Update to 6c9cf82 (5 months ago) <Shim Tw>
|/
* 6c9cf82 - Merge pull request #86 from tonyxue/master (6 months ago) <Tim Green>
|\
| * 3961f0b - zh-cn update for 9c1f837 (6 months ago) <Tony Xue>
|/
* 6f5cfae - Needs more :trollface: (6 months ago) <Rafal Chmiel>
* 9c1f837 - Merge pull request #85 from gWorldz/master (6 months ago) <Rafal Chmiel>
|\
| * b957df8 - Update README.md (6 months ago) <Doc gWorldz>
| * aa8d100 - Update README.md (6 months ago) <Doc gWorldz>
|/
* 6a1e7aa - Merge pull request #84 from TheCellarRoom/master (6 months ago) <Tim Green>
|\
| * 1eaf88 - Update README.md (6 months ago) <ChelseaStats>
```

这要归功于 [Palesz](#) 在 stackoverflow 的回答。

这个命令可以被用作别名，详细做法见[这里](#)。

[进一步了解 Git `Log` 命令](#).

Git 查询

Git 查询运行你在之前的所有提交信息里进行搜索，找到其中和搜索条件相匹配的最近的一条。

```
$ git show :/query
```

这里 `query` (区别大小写) 是你想要搜索的词语，这条命令会找到包含这个词语的最后那个提交并显示变动详情。

```
$ git show :/typo
```

```
Tim-MacBook-Pro:ghcs tim$ git show :/typo
commit e70c204e3d7cb154f866b575d01c1da14b68a6c
Author: Rafal Chmiel <hi@rafalchmiel.com>
Date: Mon Apr 14 12:37:57 2014 +0100

    Fix typo

diff --git a/README.md b/README.md
index 2bc494e..40022d2 100644
--- a/README.md
+++ b/README.md
@@ -118,7 +118,7 @@ To see all of the shortcuts for the current page press '?'.

## Closing Issues with Commits

-If a particular commit fixes an issue, any of the keywords 'fix/fixes/fixed' or 'close/closes/closed', followed by the issue number, will c
+If a particular commit fixes an issue, any of the keywords 'fix/fixes/fixed' or 'close/closes/closed', followed by the issue number, will c

```bash
$ git commit -m "Fix cock up, fixes #12"
(END)
```

- 按 `q` 键退出命令。\*

## 合并分支

输入命令:

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

这会显示所有已经合并到你当前分支的分支列表。

相反地：

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

会显示所有还没有合并到你当前分支的分支列表。

[进一步了解 Git `branch` 命令.](#)

## 修复有问题的提交以及自动合并

如果上一个或多个提交包含了错误，可以在你修复问题后使用下列命令处理（假设要修复的提交版本是 `abcde`）：

```
$ git commit --fixup=abcde
$ git rebase abcde^ --autosquash -i
```

[进一步了解 Git `commit` 命令.](#) [进一步了解 Git `rebase` 命令.](#)

## 以网站方式查看本地仓库

使用 Git 的 `instaweb` 可以立即在 `gitweb` 中浏览你的工作仓库。这个命令是个简单的脚本，配置了 `gitweb` 和用来浏览本地仓库的Web服务器。（译者注：默认需要lighttpd支持）

```
$ git instaweb
```

执行后打开：

The screenshot shows the gitweb interface for a local repository. At the top, there's a header bar with 'projects / .git / summary' and a search bar. Below the header, there's a summary section with repository details: 'description: Unnamed repository; edit this file 'description' to name the repository.', 'owner: Rafał Chmiel', and 'last change: Mon, 14 Apr 2014 20:04:18 +0100 (20:04 +0100)'. The main content area is divided into sections: 'shortlog', 'heads', and 'remotes'. The 'shortlog' section lists numerous commits from various authors (Rafał Chmiel, Tim Green) with their commit messages and details like author, committer, tree, and snapshot. The 'heads' section shows the master branch with its last commit. The 'remotes' section is currently empty.

[进一步了解 Git `instaweb` 命令.](#)

## Git 配置

所有 Git 配置都保存在你的 `.gitconfig` 文件中。

# Git 命令自定义别名

别名用来帮助你定义自己的 git 命令。比如你可以定义 `git a` 来运行 `git add --all`。

要添加一个别名，一种方法是打开 `~/.gitconfig` 文件并添加如下内容：

```
[alias]
co = checkout
cm = commit
p = push
Show verbose output about tags, branches or remotes
tags = tag -l
branches = branch -a
remotes = remote -v
```

...或者在命令行里键入：

```
$ git config --global alias.new_alias git function
```

例如：

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

指向多个命令的别名可以用引号来定义：

```
$ git config --global alias.ac 'add -A . && commit'
```

下面列出了一些有用的别名：

别名 Alias	命令 Command	如何设置 What to Type			
git cm	git commit	<code>git config --global alias.cm commit</code>			
git co	git checkout	<code>git config --global alias.co checkout</code>			
git ac	git add . -A git commit	<code>git config --global alias.ac '!git add -A &amp;&amp; git commit'</code>			
git st	git status -sb	<code>git config --global alias.st 'status -sb'</code>			
git tags	git tag -l	<code>git config --global alias.tags 'tag -l'</code>			
git branches	git branch -a	<code>git config --global alias.branches 'branch -a'</code>			
git cleanup	`git branch --merged	<code>grep -v '*'</code>	xargs git branch -d`	<code>`git config --global alias.cleanup "!git branch --merged`</code>	grep -v '*'
git remotes	git remote -v	<code>git config --global alias.remotes 'remote -v'</code>			
git lg	git log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)	<code>git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)"</code>			

```
%Cgreen(%cr) %C(bold blue)
<%an>%Creset' --abbrev-commit
--
```

```
%Cgreen(%cr) %C(bold blue)
<%an>%Creset' --abbrev-commit
--"
```

## 自动更正

如果键入 `git comit` 你会看到如下输出：

```
$ git comit -m "Message"
git: 'comit' is not a git command. See 'git --help'.

Did you mean this?
commit
```

为了在键入 `comit` 调用 `commit` 命令，只需启用自动纠错功能：

```
$ git config --global help.autocorrect 1
```

现在你就会看到：

```
$ git comit -m "Message"
WARNING: You called a Git command named 'comit', which does not exist.
Continuing under the assumption that you meant 'commit'
in 0.1 seconds automatically...
```

## 颜色输出

要在你的 Git 命令输出里加上颜色的话，可以用如下命令：

```
$ git config --global color.ui 1
```

进一步了解 Git `config` 命令。

## Git 资源

Title	Link
Official Git Site	<a href="http://git-scm.com/">http://git-scm.com/</a>
Official Git Video Tutorials	<a href="http://git-scm.com/videos">http://git-scm.com/videos</a>
Code School Try Git	<a href="http://try.github.com/">http://try.github.com/</a>
Introductory Reference & Tutorial for Git	<a href="http://gitref.org/">http://gitref.org/</a>
Official Git Tutorial	<a href="http://git-scm.com/docs/gittutorial">http://git-scm.com/docs/gittutorial</a>
Everyday Git	<a href="http://git-scm.com/docs/everyday">http://git-scm.com/docs/everyday</a>
Git Immersion	<a href="http://gitimmersion.com/">http://gitimmersion.com/</a>
Ry's Git Tutorial	<a href="http://rypress.com/tutorials/git/index.html">http://rypress.com/tutorials/git/index.html</a>

Git Magic	<a href="http://www-cs-students.stanford.edu/~blynn/gitmagic/">http://www-cs-students.stanford.edu/~blynn/gitmagic/</a>
GitHub Training Kit	<a href="http://training.github.com/kit">http://training.github.com/kit</a>
Git Visualization Playground	<a href="http://onlywei.github.io/explain-git-with-d3/#freestyle">http://onlywei.github.io/explain-git-with-d3/#freestyle</a>

## Git 参考书籍

Title	Link
Pragmatic Version Control Using Git	<a href="http://www.pragprog.com/titles/tsgit/pragmatic-version-control-using-git">http://www.pragprog.com/titles/tsgit/pragmatic-version-control-using-git</a>
Pro Git	<a href="http://git-scm.com/book">http://git-scm.com/book</a>
Git Internals Pluralsight	<a href="https://github.com/pluralsight/git-internals-pdf">https://github.com/pluralsight/git-internals-pdf</a>
Git in the Trenches	<a href="http://cbx33.github.com/gitt/">http://cbx33.github.com/gitt/</a>
Version Control with Git	<a href="http://www.amazon.com/Version-Control-Git-collaborative-development/dp/1449316387">http://www.amazon.com/Version-Control-Git-collaborative-development/dp/1449316387</a>
Pragmatic Guide to Git	<a href="http://www.pragprog.com/titles/pg_git/pragmatic-guide-to-git">http://www.pragprog.com/titles/pg_git/pragmatic-guide-to-git</a>
Git: Version Control for Everyone	<a href="http://www.packtpub.com/git-version-control-for-everyone/book">http://www.packtpub.com/git-version-control-for-everyone/book</a>