

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	文件名	用户	说明
1	service.doc	张三	删除了软件服务条款 5
2	service.doc	张三	增加了 License 人数限制
3	service.doc	李四	财务部门调整了合同金额
4	service.doc	张三	延长了免费升级周期

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。

Git 的诞生

很多人都知道，Linux 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。

Linux 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 Linux，然后由 Linux 本人通过手工方式合并代码！

你也许会想，为什么 Linux 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 Linux 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 Linux 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 Linux 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。

Linux 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：

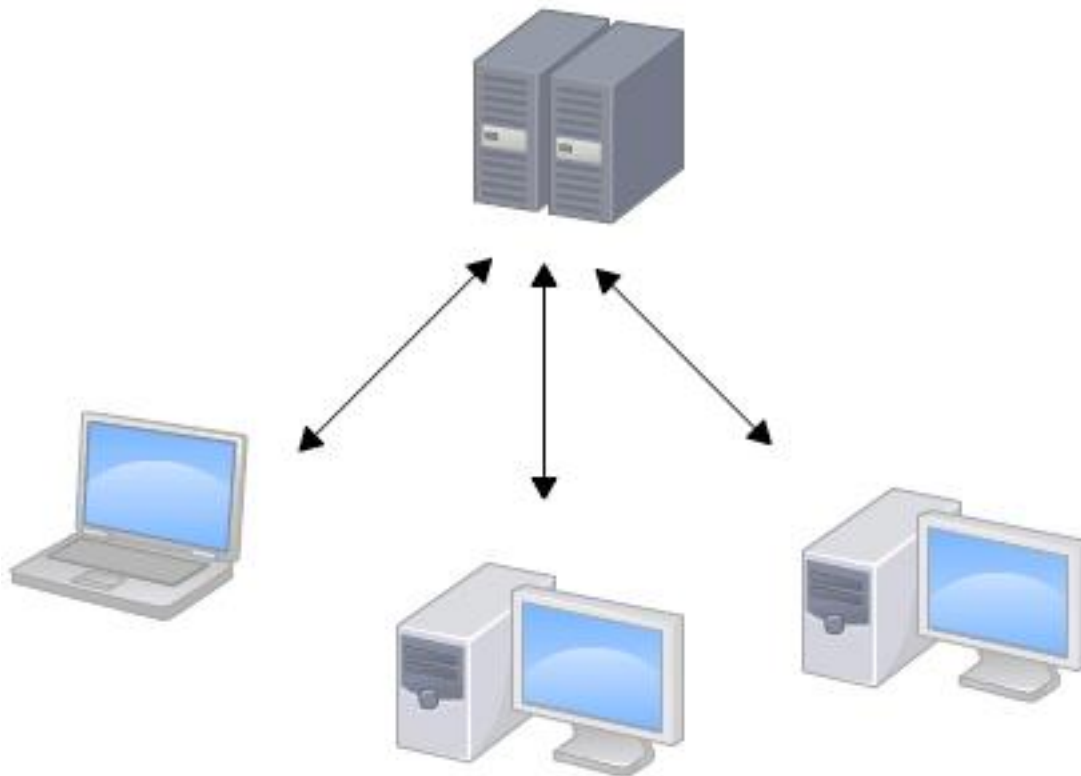
Linux 花了两周时间自己用 C 写了一个分布式版本控制系统，这就是 Git！一个月之内，Linux 系统的源码已经由 Git 管理了！牛是怎么定义的呢？大家可以体会一下。

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。

Linux 一直痛恨的 CVS 及 SVN 都是集中式的版本控制系统，而 Git 是分布式版本控制系统，集中式和分布式版本控制系统有什么区别呢？

先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆。

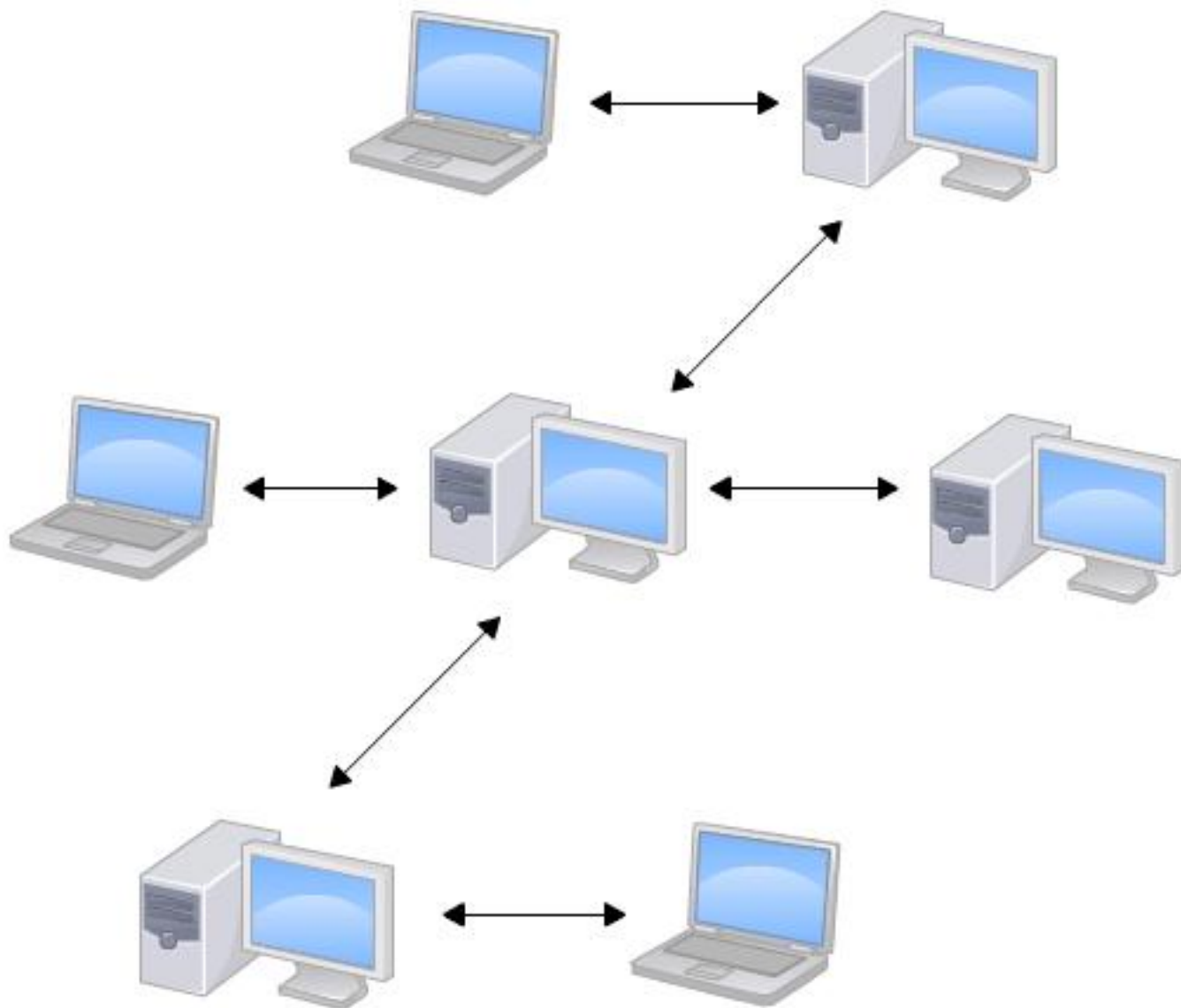


集中式版本控制系统最大的毛病就是必须联网才能工作，如果在局域网内还好，带宽够大，速度够快，可如果在互联网上，遇到网速慢的话，可能提交一个10M的文件就需要5分钟，这还不得把人给憋死啊。

那分布式版本控制系统与集中式版本控制系统有何不同呢？首先，分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。既然每个人电脑上都有一个完整的版本库，那多个人如何协作呢？比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

和集中式版本控制系统相比，分布式版本控制系统的安全性要高很多，因为每个人电脑里都有完整的版本库，某一个人的电脑坏掉了不要紧，随便从其他人那里复制一个就可以了。而集中式版本控制系统的中央服务器要是出了问题，所有人都没法干活了。

在实际使用分布式版本控制系统的时候，其实很少在两人之间的电脑上推送版本库的修改，因为可能你们俩不在一个局域网内，两台电脑互相访问不了，也可能今天你的同事病了，他的电脑压根没有开机。因此，分布式版本控制系统通常也有一台充当“中央服务器”的电脑，但这个服务器的作用仅仅是用来方便“交换”大家的修改，没有它大家也一样干活，只是交换修改不方便而已。



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

CVS作为最早的开源而且免费的集中式版本控制系统，直到现在还有不少人用。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**！

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。

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

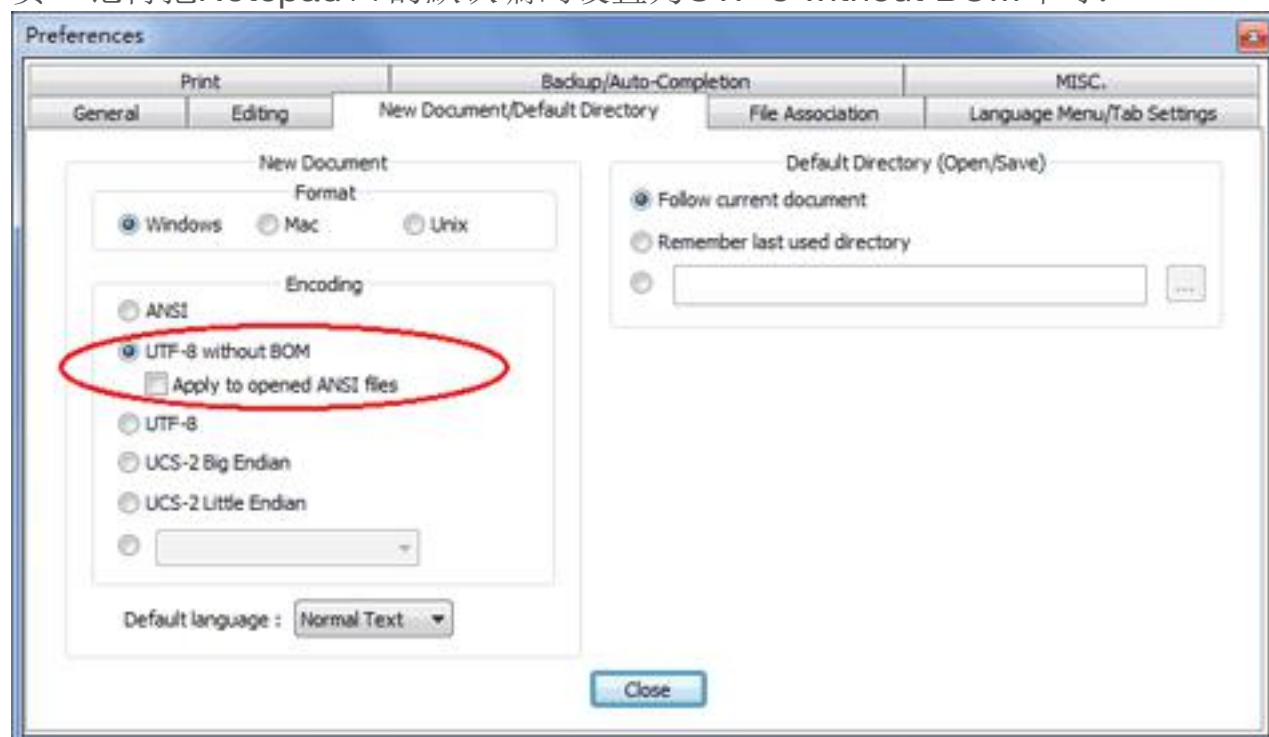
首先这里再明确一下，所有的版本控制系统，其实只能跟踪文本文件的改动，比如**TXT**文件，网页，所有的程序代码等等，**Git**也不例外。版本控制系统可以告诉你每次的改动，比如在第**5**行加了一个单词“**Linux**”，在第**8**行删了一个单词“**Windows**”。而图片、视频这些二进制文件，虽然也能由版本控制系统管理，但没法跟踪文件的变化，只能把二进制文件每次改动串起来，也就是只知道图片从**100KB**改成了**120KB**，但到底改了啥，版本控制系统不知道，也没法知道。

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

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

使用Windows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0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“？”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：



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learn git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
```

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。

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。

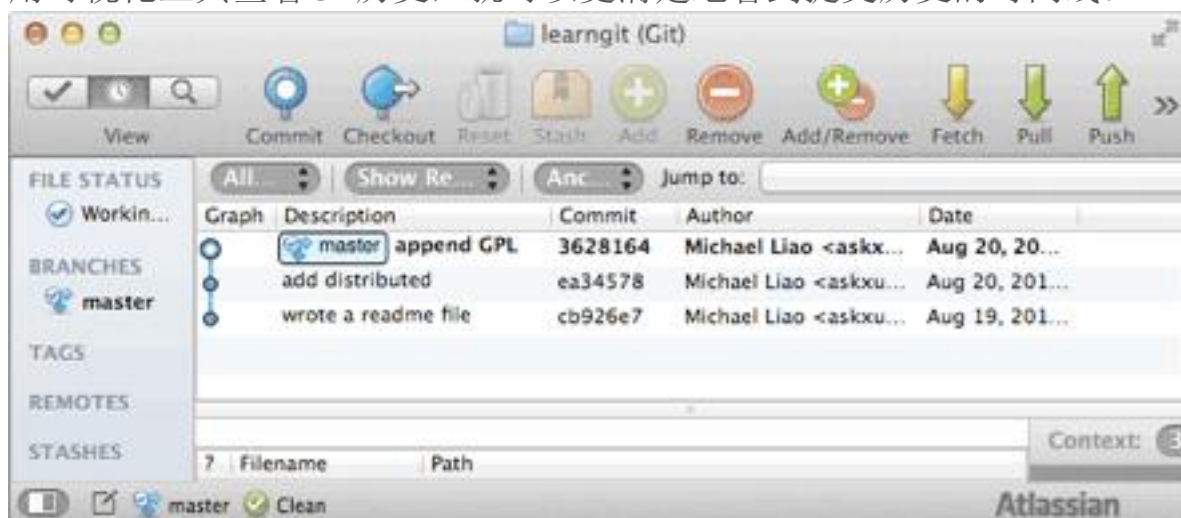
如果嫌输出信息太多，看得眼花缭乱的，可以试试加上`--pretty=oneline`参数：

```
$ 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 you 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：

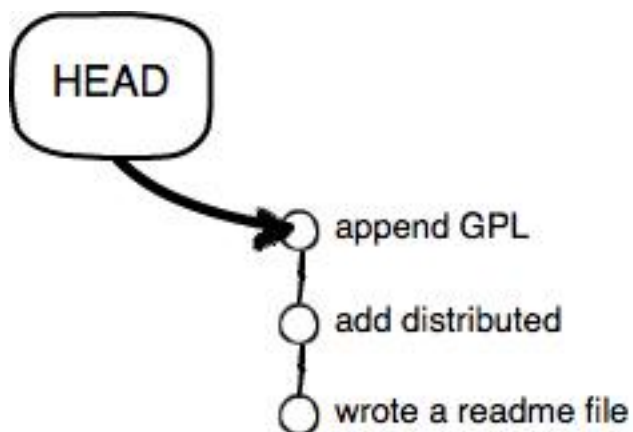
```
$ cat readme.txt
```

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

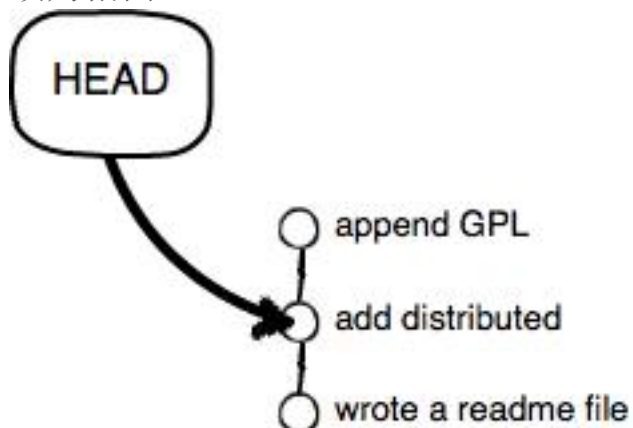
```
Git is free software distributed under the GPL.
```

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。

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。

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。

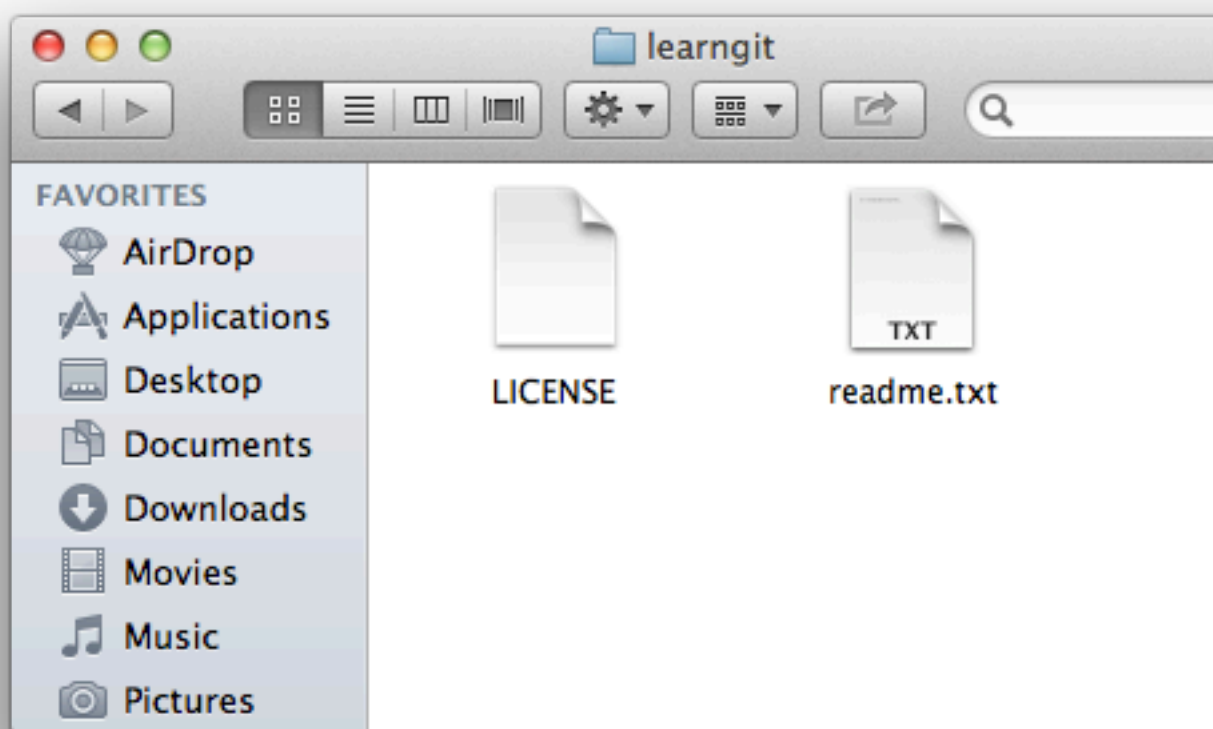
工作区和暂存区

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）

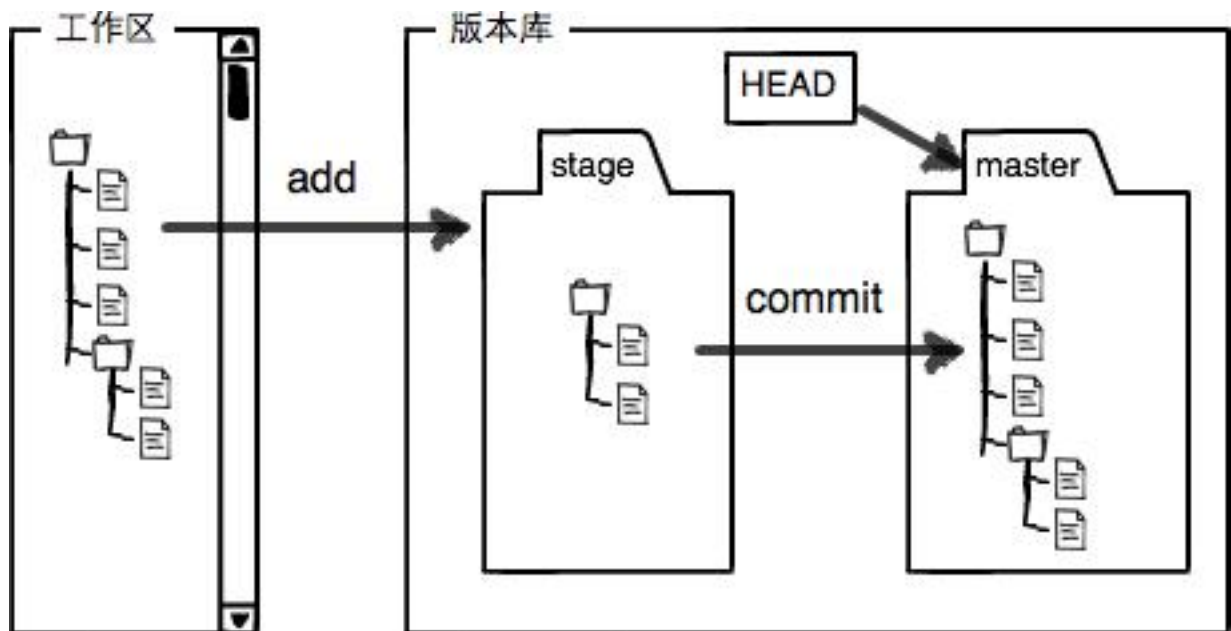
就是你在电脑里能看到的目录，比如我的`learngit`文件夹就是一个工作区：



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自动为我们创建了一个唯一的`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
```

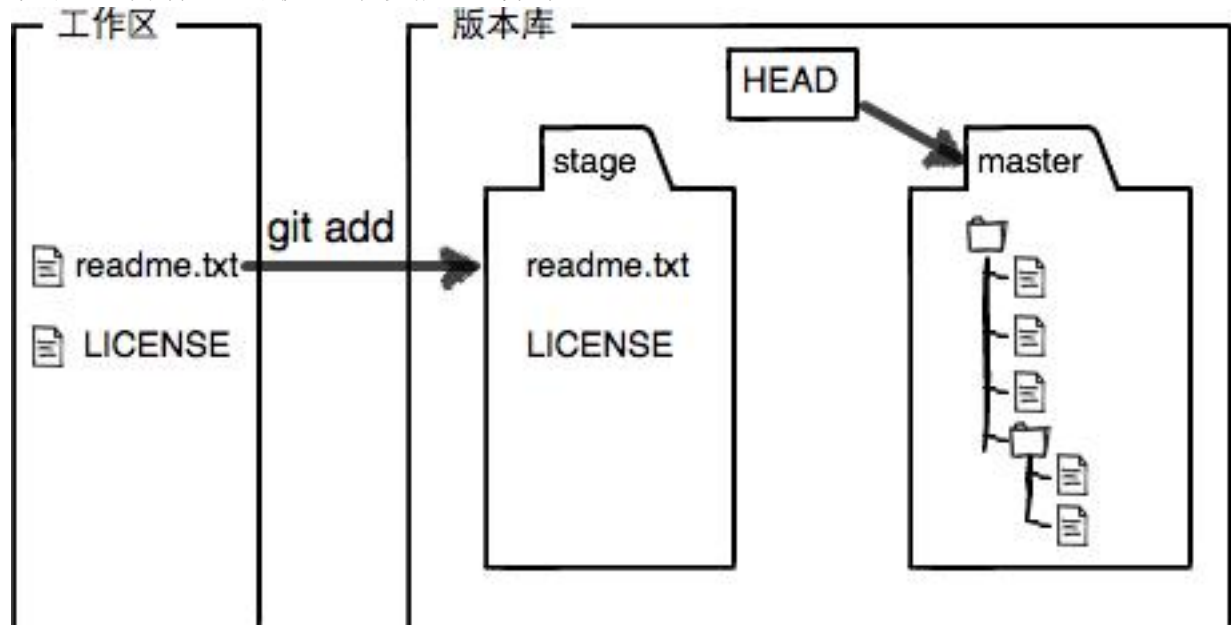

no changes added to commit (use "git add" and/or "git commit -a")

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

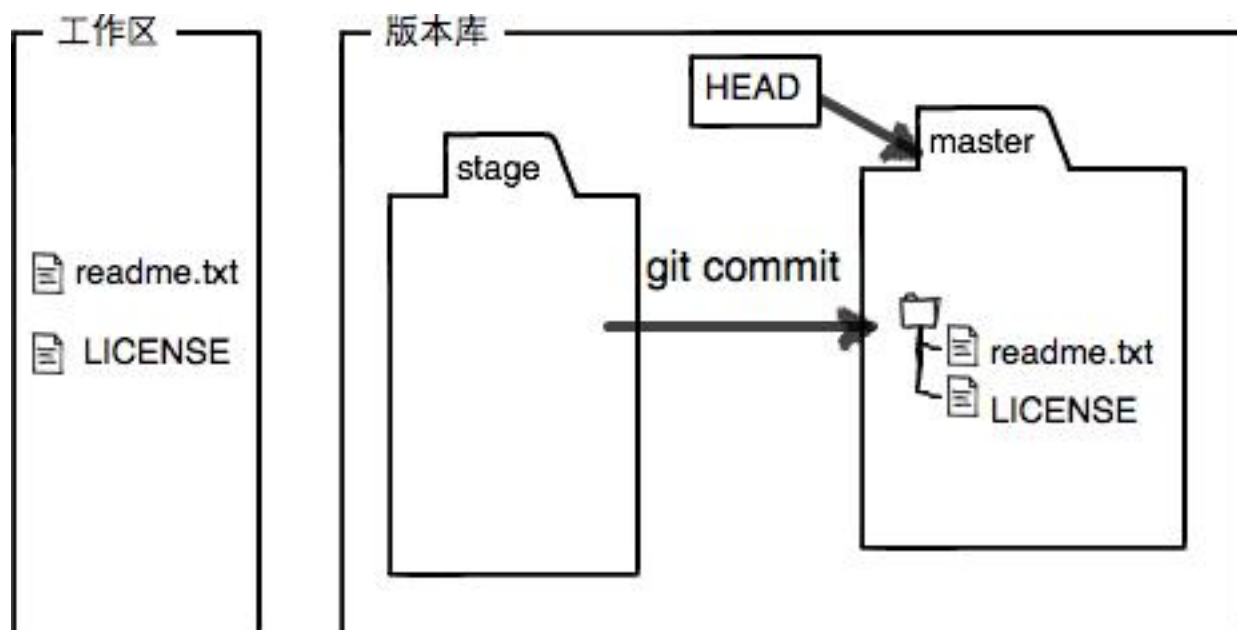
(`Stage`)，然后，执行`git commit`就可以一次性把暂存区的所有修改提交到分支。

```
$ 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。

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 director
y)

#

#       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差别:

```
$ 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。

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。

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 director
y)

#

#       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。

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！

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，不过前提是没有推送到远程库。

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 director
y)

#

#       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。

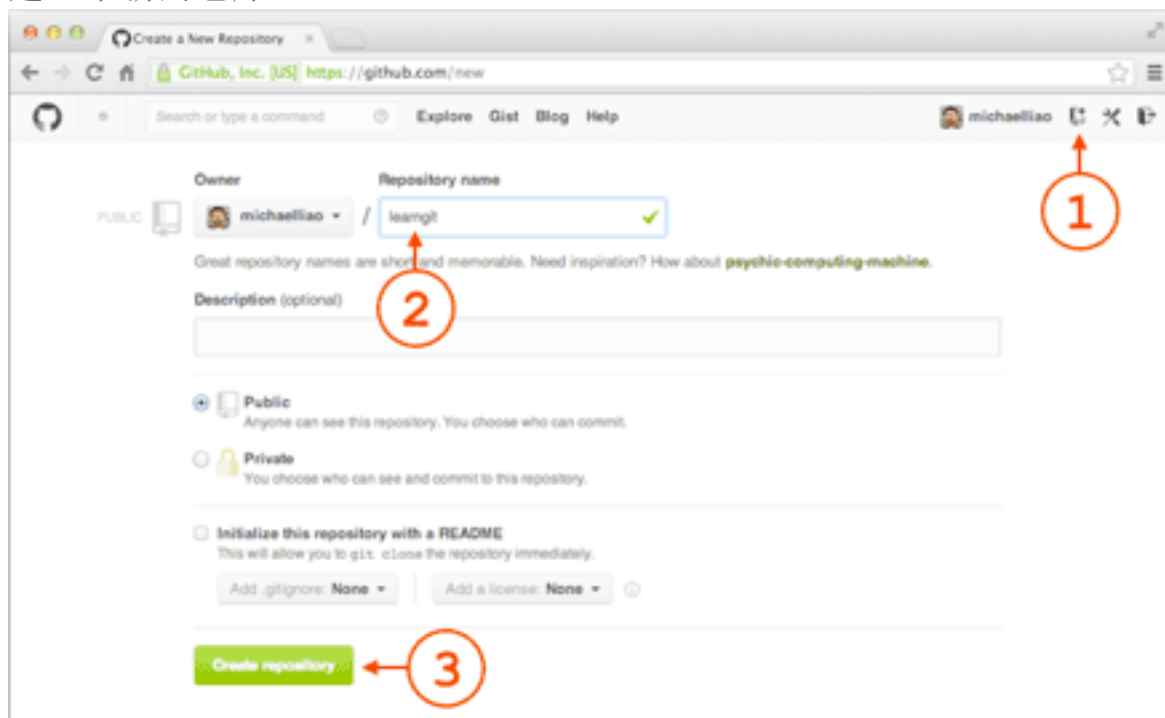
另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本：

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

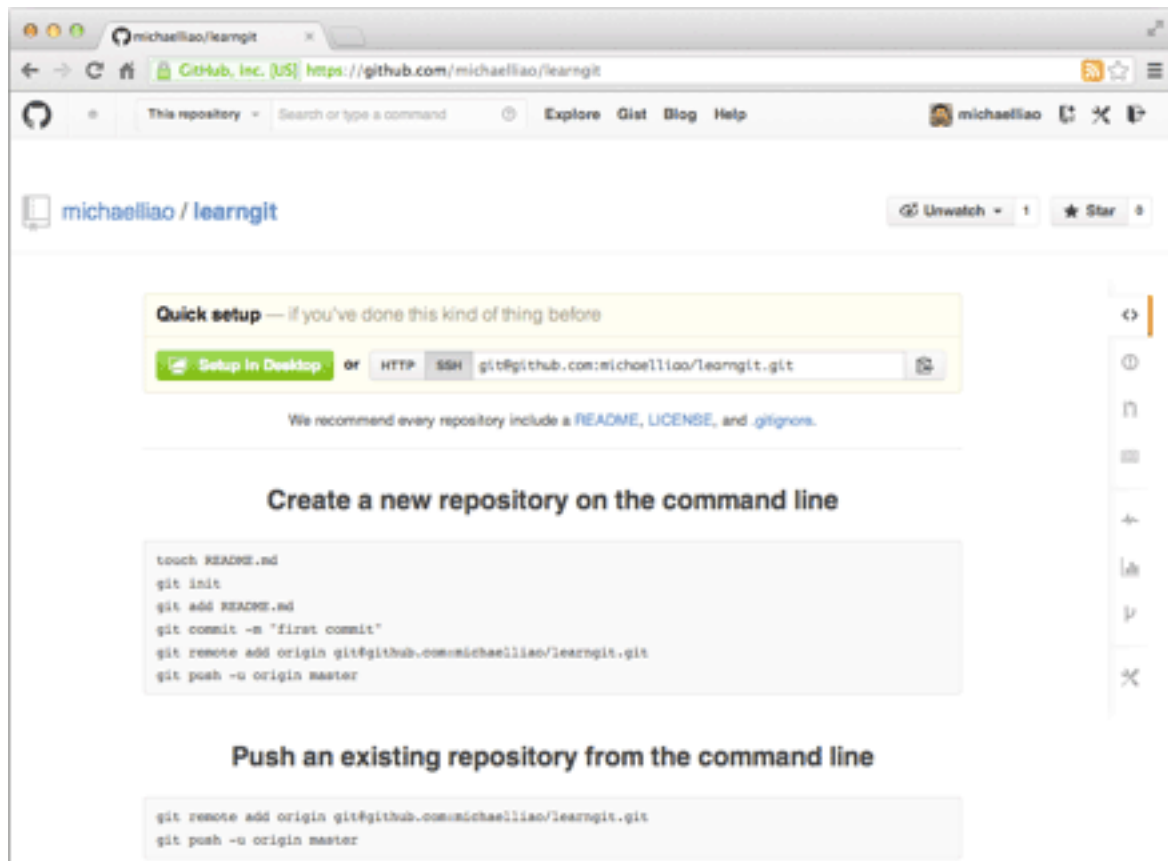
`git checkout` 其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

添加远程库

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learnngit，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：

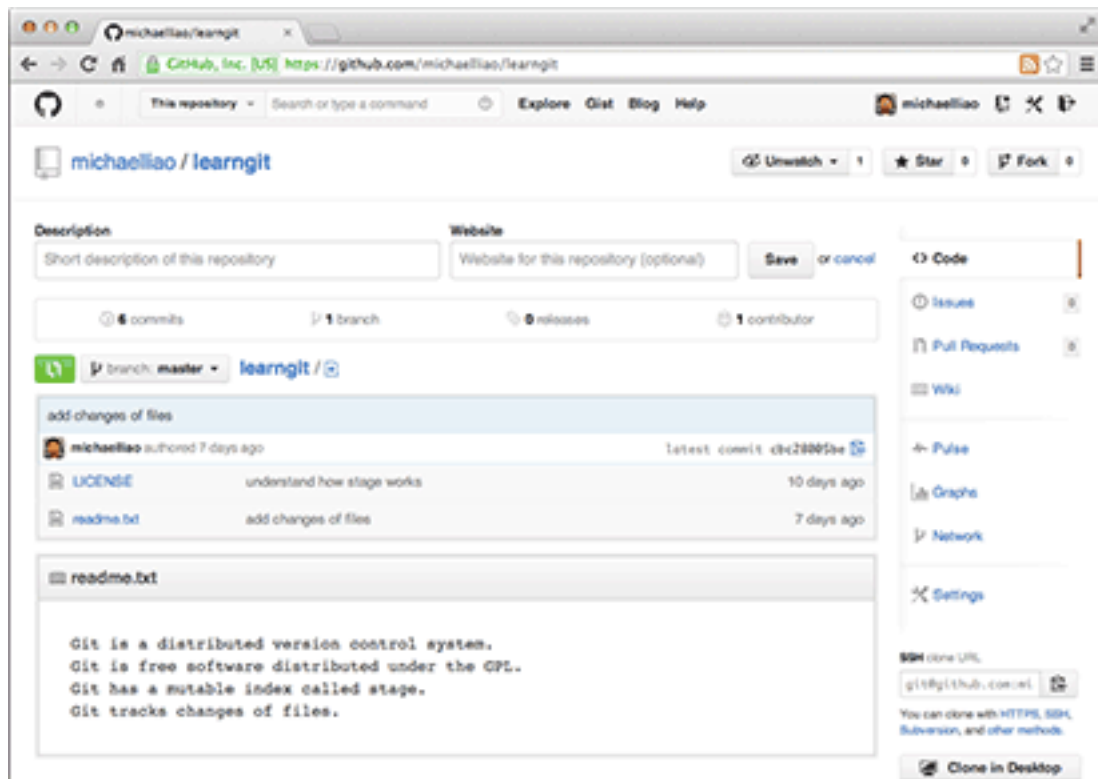
```
$ git push -u origin master  
Counting objects: 19, done.  
Delta compression using up to 4 threads.  
Compressing objects: 100% (19/19), done.  
Writing objects: 100% (19/19), 13.73 KiB, done.  
Total 23 (delta 6), reused 0 (delta 0)  
To git@github.com:michaelliao/learngit.git  
* [new branch]      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`分支内容推送的远程新的`master`分支，还会把本地的`master`分支和远程的`master`分支关联起来，在以后的推送或者拉取时就可以简化命令。

推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：



从现在起，只要本地作了提交，就可以通过命令：

```
$ git push origin master
```

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

SSH警告

当你第一次使用Git的`clone`或者`push`命令连接GitHub时，会得到一个警告：

```
The authenticity of host 'github.com (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；

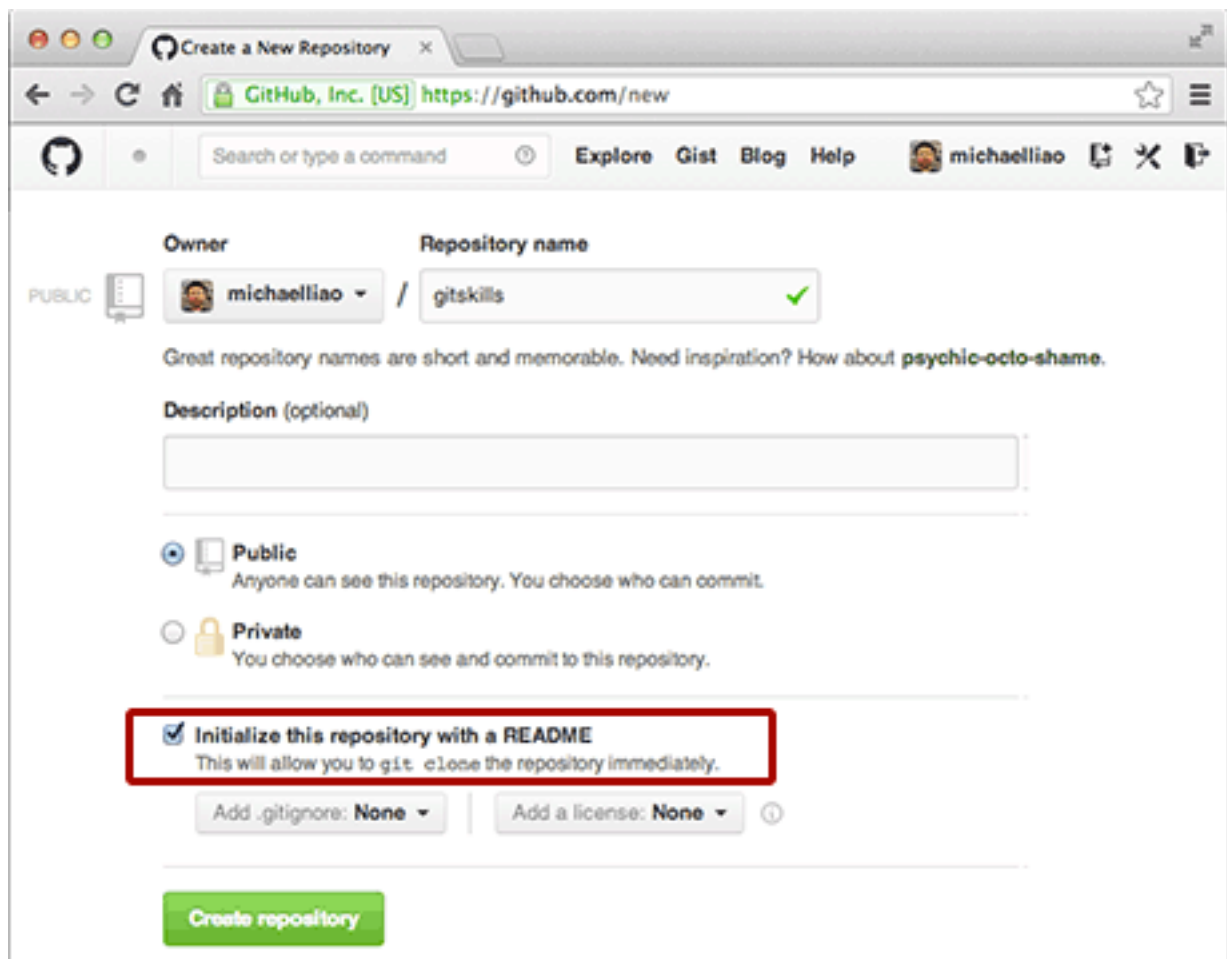
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

从远程库克隆

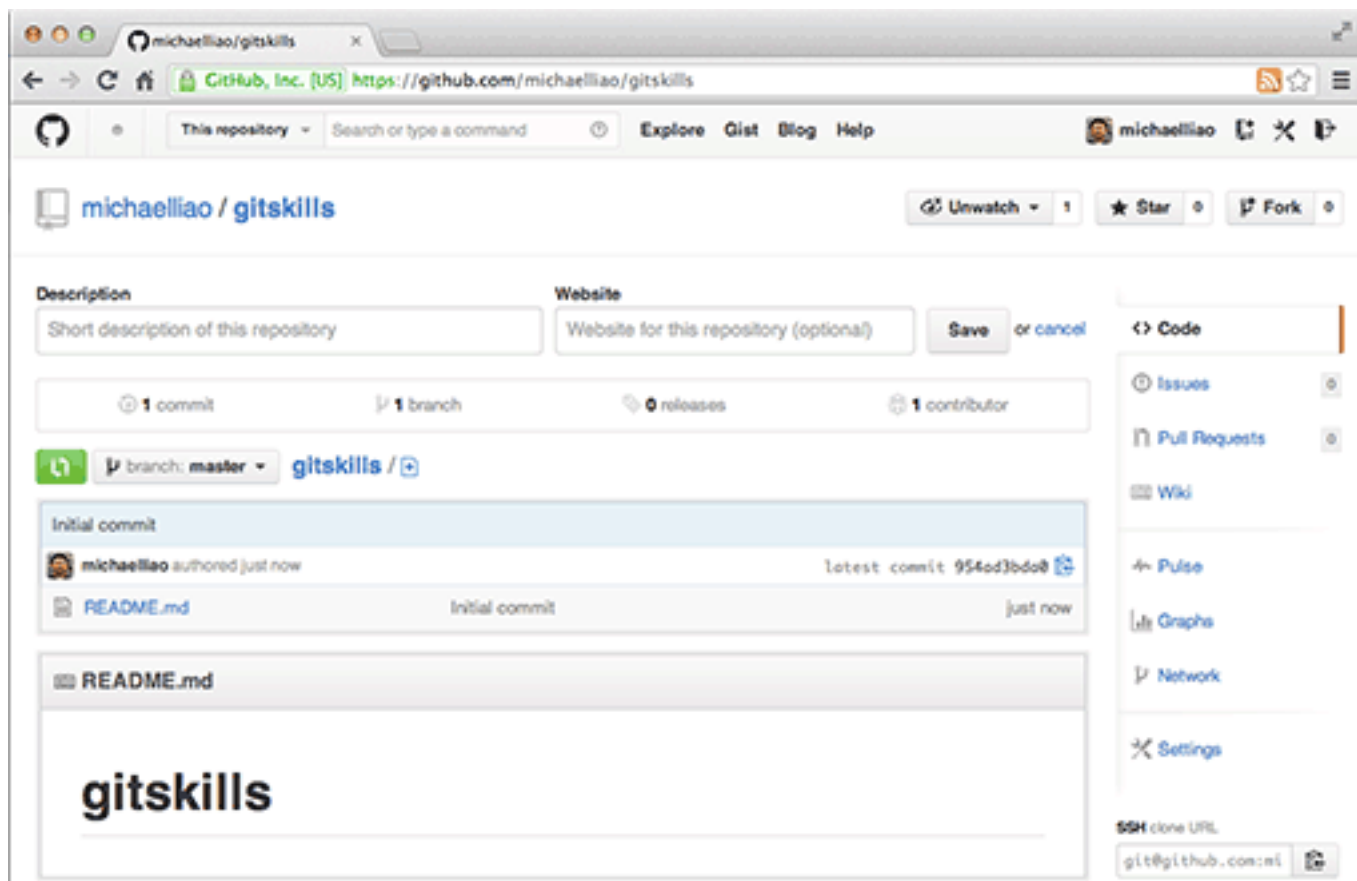
上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。

现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登陆GitHub，创建一个新的仓库，名字叫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。

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

你也许还注意到，GitHub给出的地址不止一个，还可以用

`https://github.com/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

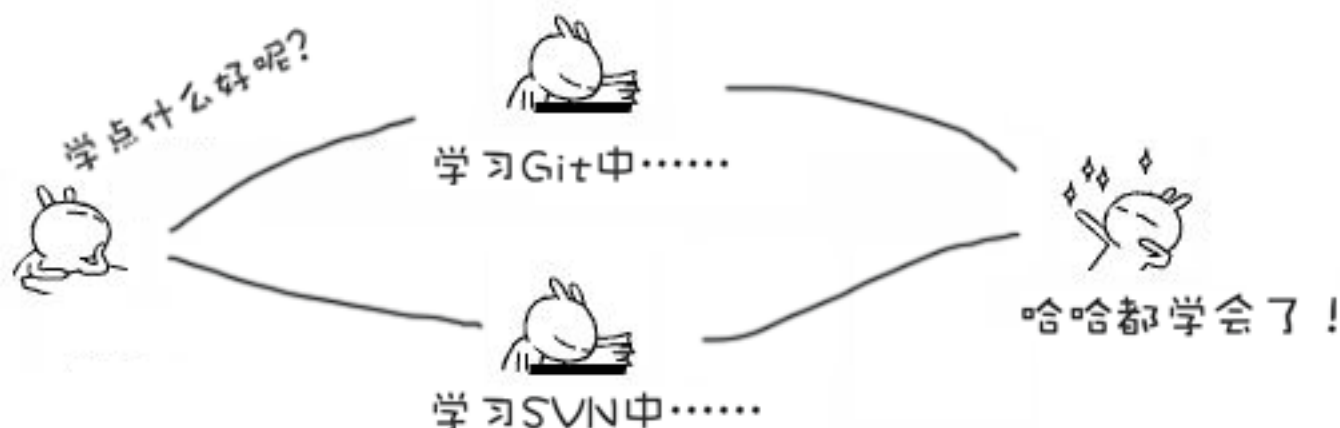
要克隆一个仓库，首先必须知道仓库的地址，然后使用`git clone`命令克隆。

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。

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。

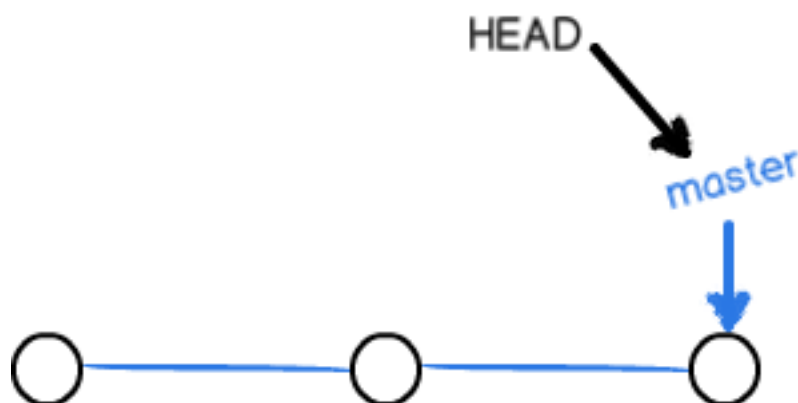
现在有了分支，就不用怕了。你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

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。

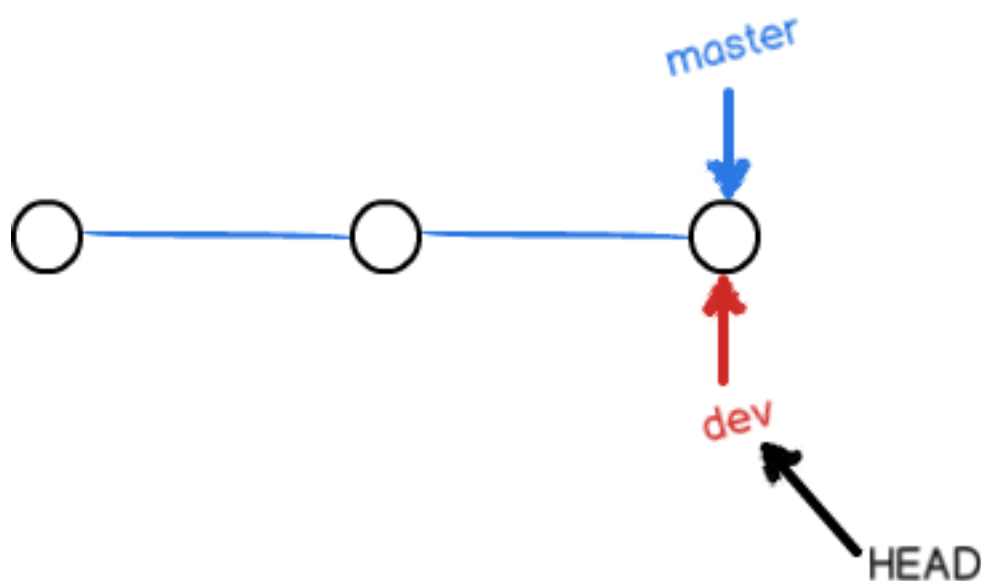
创建与合并分支

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：



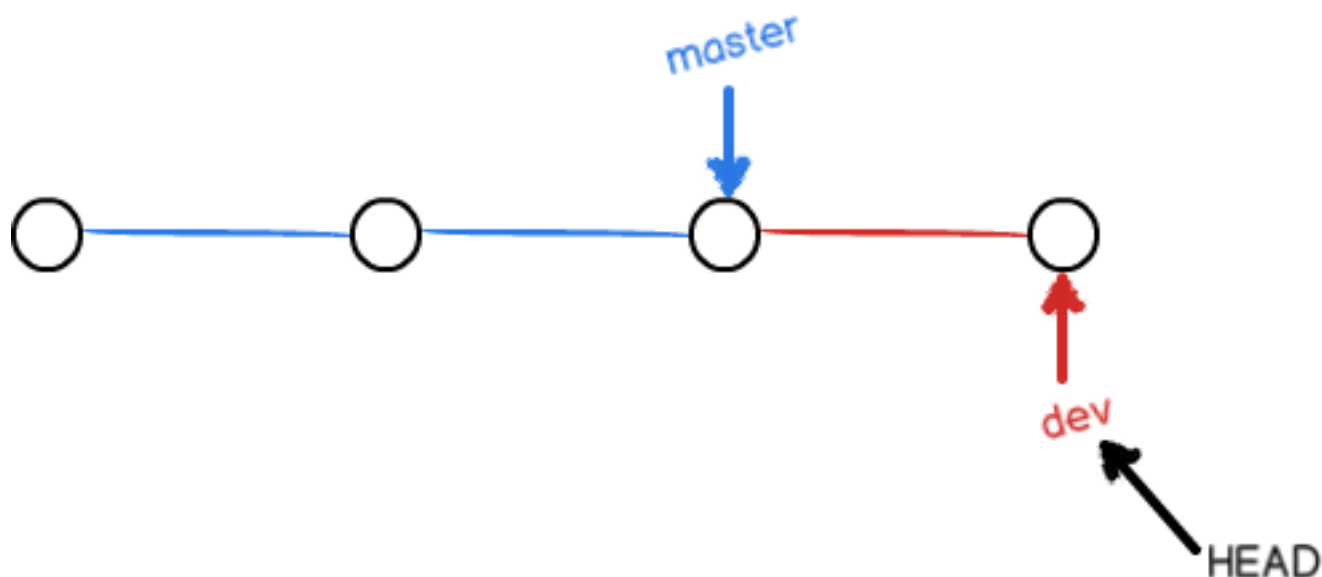
每次提交，`master`分支都会向前移动一步，这样，随着你不断提交，`master`分支的线也越来越长：

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：

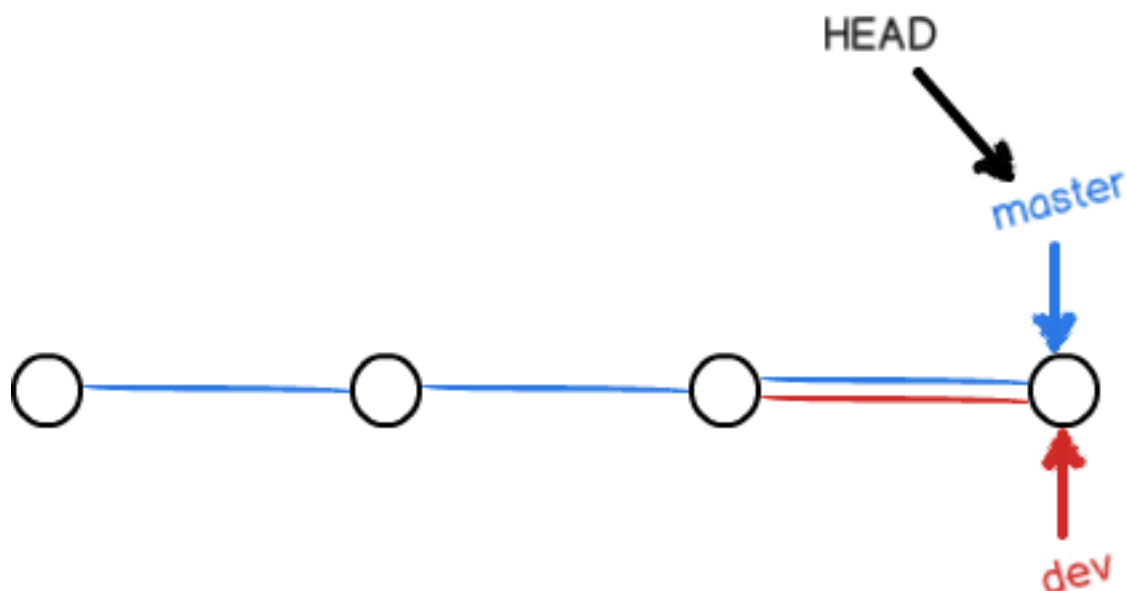


你看，Git创建一个分支很快，因为除了增加一个`dev`指针，改改`HEAD`的指向，工作区的文件都没有任何变化！

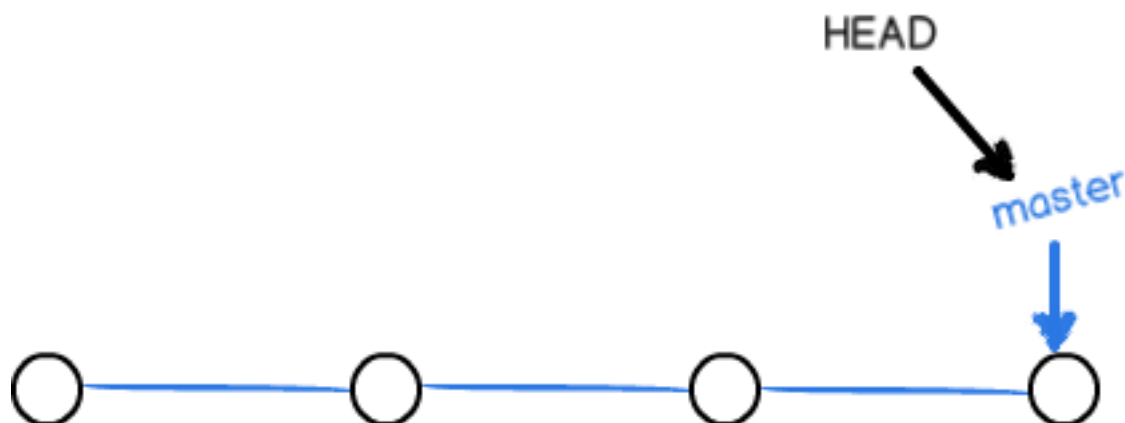
不过，从现在开始，对工作区的修改和提交就是针对`dev`分支了，比如新提交一次后，`dev`指针往前移动一步，而`master`指针不变：



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？

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
```

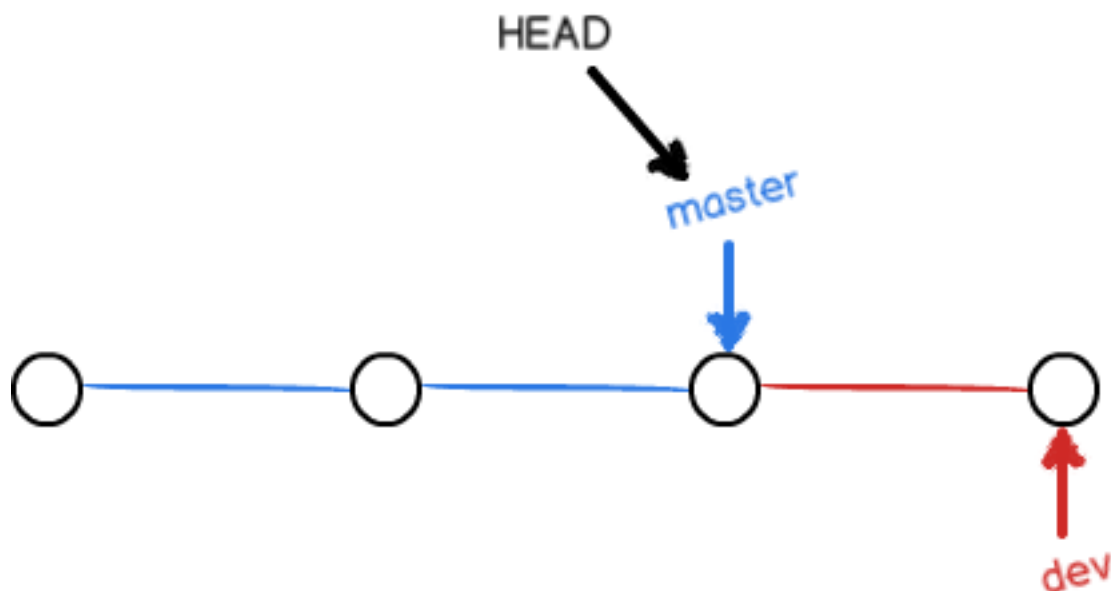
```
1 file changed, 1 insertion(+)
```

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。

小结

Git鼓励大量使用分支：

查看分支：`git branch`

创建分支：`git branch <name>`

切换分支：`git checkout <name>`

创建+切换分支：`git checkout -b <name>`

合并某分支到当前分支：`git merge <name>`

删除分支：`git branch -d <name>`

解决冲突

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。

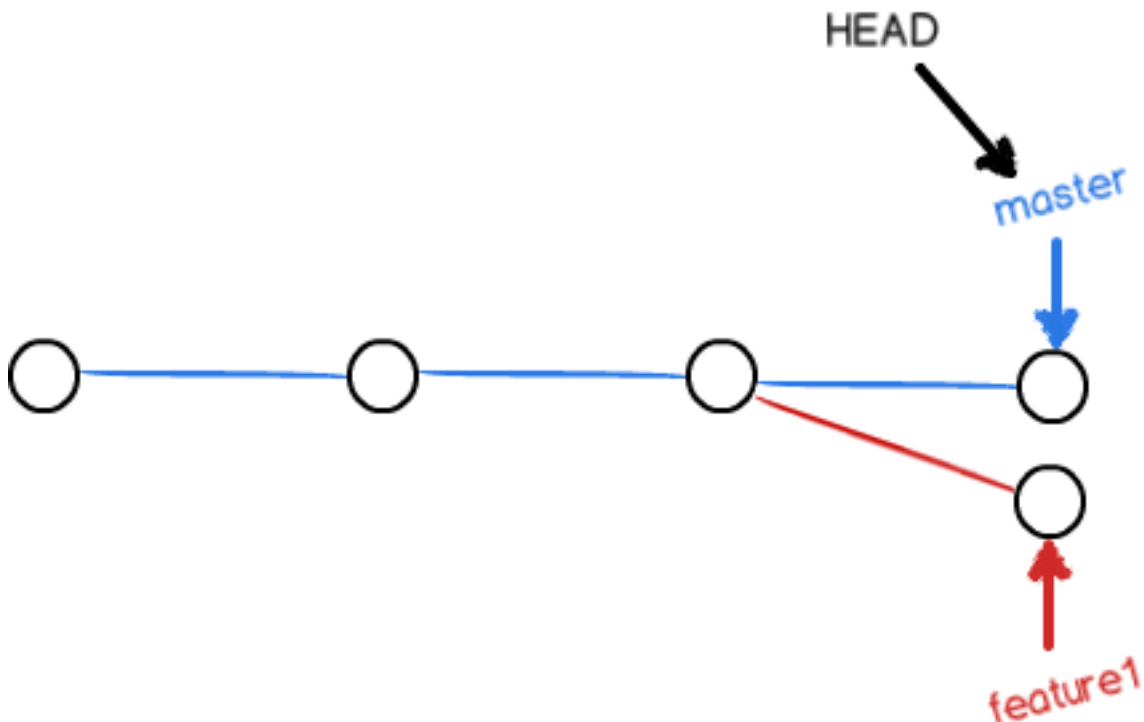
在`master`分支上把`readme.txt`文件的最后一行改为：

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
```

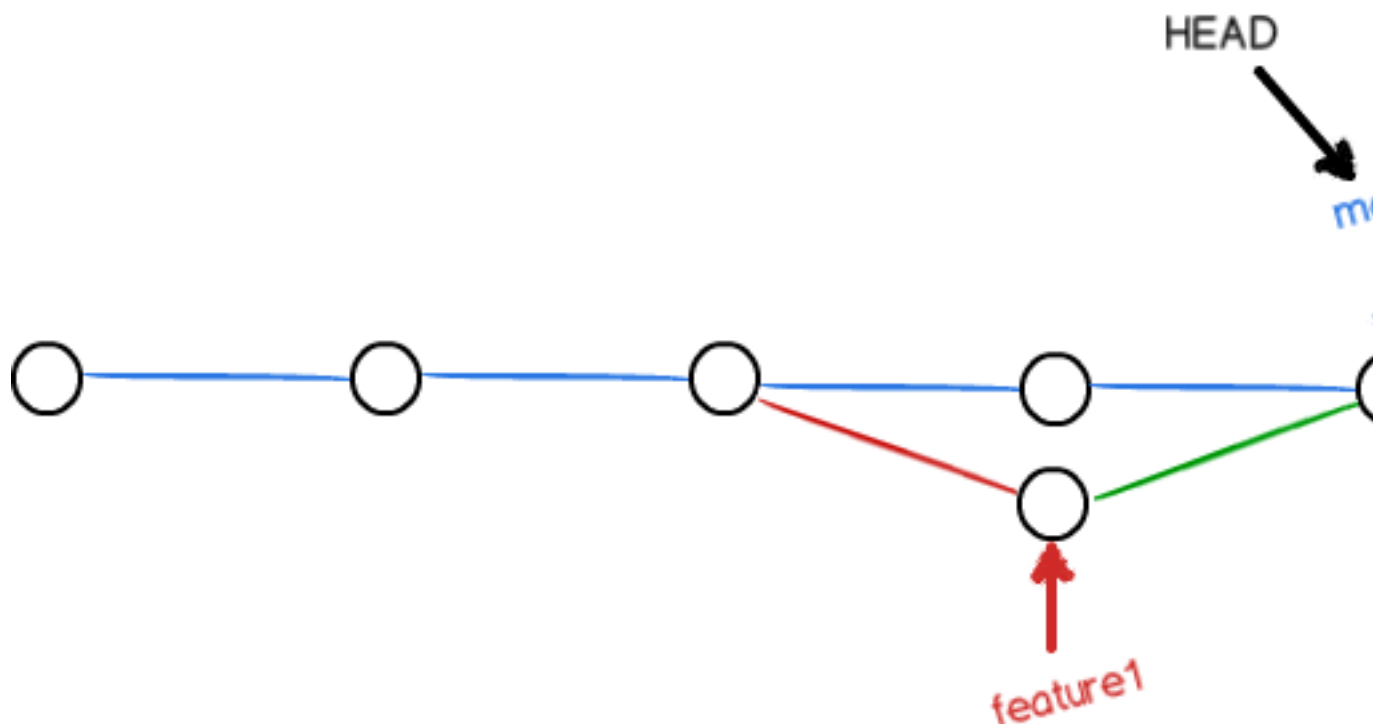
Git用<<<<<<<, =====, >>>>>>>标记出不同分支的内容，我们修改如下后保存：

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

再提交：

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

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。

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
```

```
| \
```

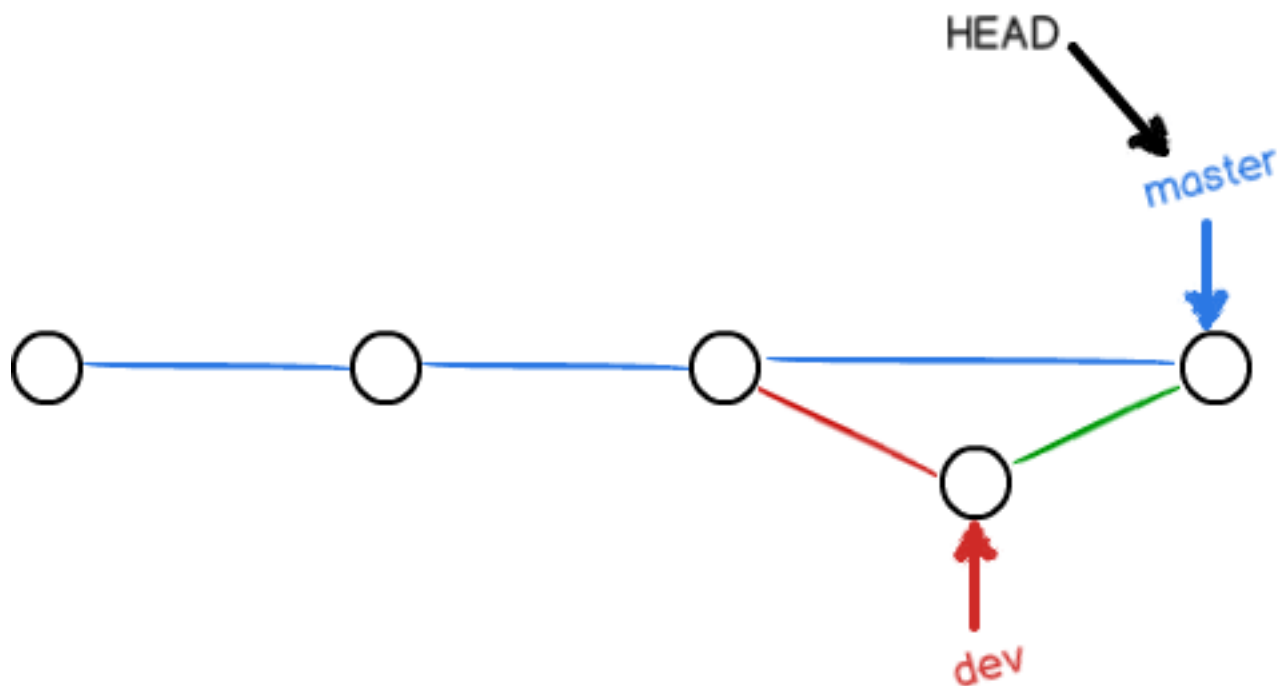
```
| * 6224937 add merge
```

```
| /
```

```
* 59bc1cb conflict fixed
```

```
...
```

可以看到，不使用Fast forward模式，merge后就像这样:



分支策略

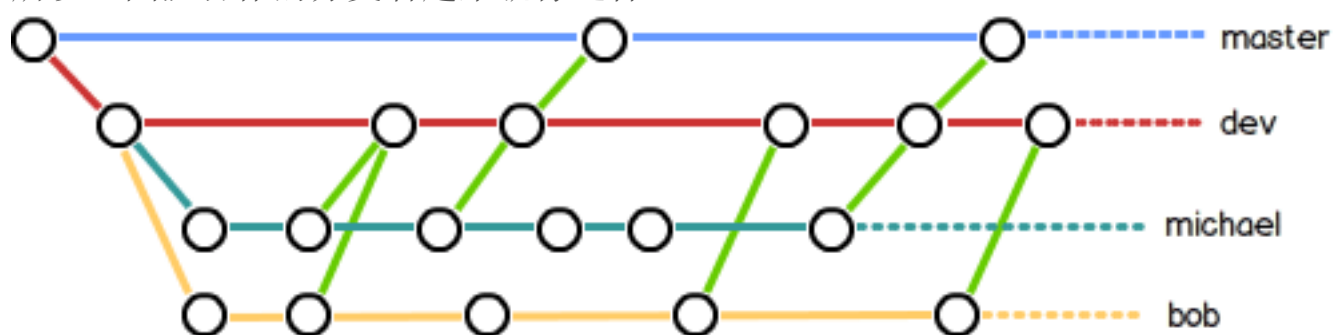
在实际开发中，我们应该按照几个基本原则进行分支管理：

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。

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 director
y)

#

#       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，必须在两个小时内修复该 bug，怎么办？

幸好，Git 还提供了一个 `stash` 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

```
$ 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 director
y)

#

#       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}
```

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。

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！

小结

开发一个新 **feature**，最好新建一个分支；

如果要丢弃一个没有被合并过的分支，可以通过 `git branch -D <name>` 强行删除。

多人协作

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

要查看远程库的信息，用 `git remote`：

```
$ 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！

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
```

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
```

因此，多人协作的工作模式通常是这样：

1. 首先，可以试图用 `git push origin branch-name` 推送自己的修改；
2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并；
3. 如果合并有冲突，则解决冲突，并在本地提交；
4. 没有冲突或者解决掉冲突后，再用 `git push origin branch-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

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。

创建标签

在 Git 中打标签非常简单，首先，切换到需要打标签的分支上：

```
$ git branch
* dev
  master

$ git checkout master

Switched to branch 'master'
```

然后，敲命令 `git tag <name>` 就可以打一个新标签：

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

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 <tagname>** 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 <tagname>` 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。

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。

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 <tagname>`：

```
$ 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
```

要看看是否真的从远程库删除了标签，可以登陆 **GitHub** 查看。

小结

- 命令 `git push origin <tagname>` 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 <tagname>` 可以删除一个本地标签；
- 命令 `git push origin :refs/tags/<tagname>` 可以删除一个远程标签。

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

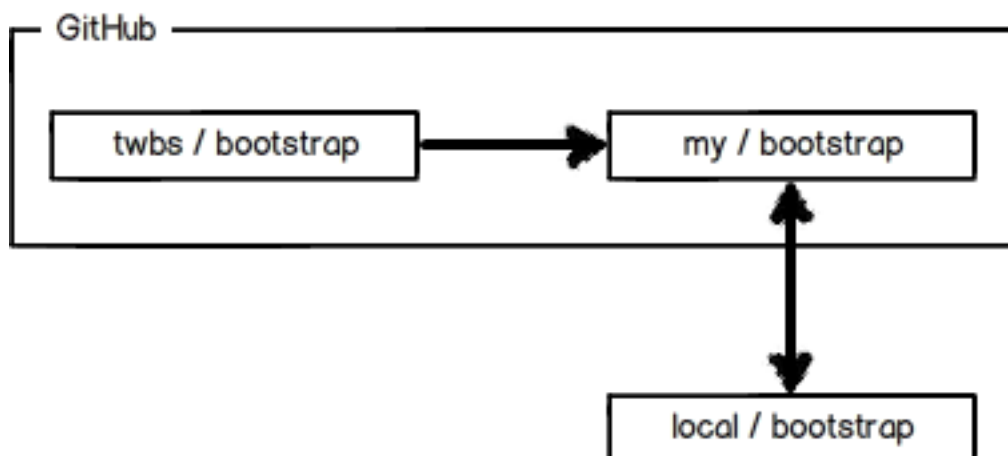
<https://github.com/twbs/bootstrap>，点“Fork”就在自己的账号下克隆了一个bootstrap仓库，然后，从自己的账号下clone：

```
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。

使用码云

使用GitHub时，国内的用户经常遇到的问题是访问速度太慢，有时候还会出现无法连接的情况（原因你懂的）。

如果我们希望体验Git飞一般的速度，可以使用国内的Git托管服务——码云（gitee.com）。

和GitHub相比，码云也提供免费的Git仓库。此外，还集成了代码质量检测、项目演示等功能。对于团队协作开发，码云还提供了项目管理、代码托管、文档管理的服务，5人以下小团队免费。

码云的免费版本也提供私有库功能，只是有5人的成员上限。

使用码云和使用GitHub类似，我们在码云上注册账号并登录后，需要先上传自己的SSH公钥。选择右上角用户头像 -> 菜单“修改资料”，然后选择“SSH公钥”，填写一个便于识别的标题，然后把用户主目录下的.ssh/id_rsa.pub文件的内容粘贴进去：

码云

开源项目

代码片段

企业版

特惠

我的码云

搜索...

廖雪峰

加入时间 2年前

个人资料

修改账户

通知方式

SSH公钥

代码风格

第三方应用

升级为组织

升级为企业

捐赠管理

SSH公钥

使用SSH公钥可以让你在你的电脑和码云通讯的时候使用安全连接 (SSH地址)

您当前的SSH公钥数: 0

你还没有添加任何SSH公钥

添加公钥

标题

Michael's SSH Key

公钥

把你的公钥粘贴到这里。查看 怎样生成公钥

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDAQDQwAYs+47hfubTfqqiGcN8LqPKwQoryeqdkOk169PW0+QCZv/MWU>ZgDB61oOcjZtgW8uNH+6n+Oscnmo3YSbaaj4UleGa+FD6Jj1G+sfGYHczJh1Gw5iHHOFC1RdEdSepKTB+mDWYP5CEwcks2fhmuKkEoNfqg4zbmFva0tPSj3MZToSI56vULfsb6otUTy9TMKI++q1I7kvXktbvZo5MzxaNh1zzFD6UOei4FSvcfZoGRLMp8m0Q4+VqMOOUgT1oQGfsAV askxuefeng@gmail.com

确定

1

2

3

4

5

点击“确定”即可完成并看到刚才添加的Key:

码云 开源项目 代码片段 企业版 特惠 我的码云 搜索...

 廖雪峰
加入时间 2年前

- 个人资料
- 修改账户
- 通知方式
- SSH公钥

SSH公钥

使用SSH公钥可以让你在你的电脑和码云通讯的时候使用安全连接 (Git的Remote)

您当前的SSH公钥数: 1

Michael's SSH Key (81:90:9b:62:10:45:fc:0b:a0:85:3f:57:a2:a2:88:af)

添加于 1分钟前

添加公钥

标题

公钥标题(key)

如果我们已经有了一个本地的git仓库（例如，一个名为learngit的本地库），如何把它关联到码云的远程库上呢？

首先，我们在码云上创建一个新的项目，选择右上角用户头像 -> 菜单“控制面板”，然后点击“创建项目”：

📁 创建项目

归属	廖雪峰		
名称	learngit		
路径	https://gitee.com/liaoxuefeng/	learngit	
介绍 非必填	Learn git		
选择语言 ▼		添加.gitignore ▼	添加开源许可
是否公开	<div>公开的 🔒</div> <div><input type="checkbox"/> 使用Readme文件初始化这个项目</div> <div><input type="checkbox"/> 使用Issue模板文件初始化这个项目 ⓘ</div> <div><input type="checkbox"/> 使用Pull Request模板文件初始化这个项目 ⓘ</div> <div> 导入已有项目</div> <div>创建</div>		

项目名称最好与本地库保持一致：

然后，我们在本地库上使用命令`git remote add`把它和码云的远程库关联：

```
git remote add origin git@gitee.com:liaoxuefeng/learngit.git
```

之后，就可以正常地用`git push`和`git pull`推送了！

如果在使用命令`git remote add`时报错：

```
git remote add origin git@gitee.com:liaoxuefeng/learngit.git
```


fatal: remote origin already exists.

这说明本地库已经关联了一个名叫**origin**的远程库，此时，可以先用**git remote -v**查看远程库信息：

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

可以看到，本地库已经关联了**origin**的远程库，并且，该远程库指向GitHub。

我们可以删除已有的GitHub远程库：

```
git remote rm origin
```

再关联码云的远程库（注意路径中需要填写正确的用户名）：

```
git remote add origin git@gitee.com:liaoxuefeng/learngit.git
```

此时，我们再查看远程库信息：

```
git remote -v
origin      git@gitee.com:liaoxuefeng/learngit.git (fetch)
origin      git@gitee.com:liaoxuefeng/learngit.git (push)
```

现在可以看到，**origin**已经被关联到码云的远程库了。通过**git push**命令

就可以把本地库推送到Gitee上。

有的小伙伴又要问了，一个本地库能不能既关联GitHub，又关联码云呢？答案是肯定的，因为git本身是分布式版本控制系统，可以同步到另外一个远程库，当然也可以同步到另外两个远程库。

使用多个远程库时，我们要注意，git给远程库起的默认名称是**origin**，如果有多个远程库，我们需要用不同的名称来标识不同的远程库。

仍然以**learngit**本地库为例，我们先删除已关联的名为**origin**的远程库：

```
git remote rm origin
```

然后，先关联GitHub的远程库：

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

注意，远程库的名称叫**github**，不叫**origin**了。

接着，再关联码云的远程库：

```
git remote add gitee git@gitee.com:liaoxuefeng/learngit.git
```

同样注意，远程库的名称叫**gitee**，不叫**origin**。

现在，我们用**git remote -v**查看远程库信息，可以看到两个远程库：

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

如果要推送到GitHub，使用命令：

```
git push github master
```

如果要推送到码云，使用命令：

```
git push gitee master
```

这样一来，我们的本地库就可以同时与多个远程库互相同步：



码云也同样提供了Pull request功能，可以让其他小伙伴参与到开源项目
中来。你可以通过Fork我的仓库：

<https://gitee.com/liaoxuefeng/learngit>，创建一个`your-gitee-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，这个仓库
会在码云和 GitHub 做双向同步。

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。

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：

1. 忽略操作系统自动生成的文件，比如缩略图等；
2. 忽略编译生成的中间文件、可执行文件等，也就是如果一个文件是通过另一个文件自动生成的，那自动生成的文件就没必要放进版本库，比如 Java 编译产生的 `.class` 文件；
3. 忽略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！

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 Liao>
| \
| * 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>
| /
* fecl45a - (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。

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](#)。

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](#)

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！