



git

1.Git教程

- 1.1 在Linux上安装Git
- 1.2 在Windows上安装Git

1.3 设置

2.创建版本库

- 2.1 第一步,创建
- 2.2 第二步 , git init
- 2.3 第三步 , 把文件添加到版本库
- 2.4 小结

3.时光机穿梭

- 3.1 git status 和 git diff
- 3.2 git add 和 git commit
- 3.3 小结

4.版本回退

- 4.1 查看版本与git log
- 4.2 git reset (--head)
- 4.3 git reflog
- 4.4 小结

5.工作区和暂存区

- 5.1 工作区 (Working Directory)
- 5.2 版本库 (Repository)
- 5.3 小结

6.管理修改

- 6.1 第二次的修改没有被提交
- 6.2 问题回顾与工作区和版本库区别
- 6.3 小结

7.撤销修改

- 7.1 情形一 : 犯错
- 7.2 git checkout
- 7.3 错误地git add 和 git reset HEAD file 补救
- 7.4 小结

8.删除文件

- 8.1 删除
- 8.2 恢复或删除
- 8.3 小结

9.远程仓库

- 9.1 注册GitHub
- 9.2 小结

10.添加远程序

- 10.1 创建新仓库

10.2 本地仓库与远程仓库关联

10.3 推送到远程库

10.4 SSH警告

10.5 小结

11.从远程库克隆

11.1 先创建远程库

11.2 git clone 克隆

11.3 小结

12.分支管理简介

13.创建与合并分支

13.1 创建新分支

13.2 解决冲突

13.3 小结

14.分支管理策略

14.1 分支策略

14.2 小结

15.Bug分支

15.1 小结

16.Feature分支

16.1 添加新功能

16.2 删除分支

16.3 小结

17.多人协作

17.1 查看远程库

17.2 推送分支

17.3 抓取分支

17.4 解决冲突

17.5 多人协作的工作模式

17.6 小结

18.标签管理

18.1 创建标签

18.2 忘打标签

18.3 查看PGP签名信息

18.4 小结

19.操作标签

19.1 删除标签

19.2 小结

20.使用GitHub

20.1 参与开源项目

20.2 pull request

20.3 小结

21.自定义Git

21.1 上色

21.2 忽略特殊文件

21.3 小结

22.配置别名

22.1 "git st" Vs "git status"

22.2 配置文件

22.3 小结

23.搭建Git服务器

23.1 搭建服务器

1.Git教程

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：

1.1 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：

```
1. $ git
2. 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。

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。

1.2 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<http://msysgit.github.io/>下载，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！

1.3 设置

安装完成后，还需要最后一步设置，在命令行输入：

```
1. $ git config --global user.name "Your Name"
```

```
1. $ git config --global user.name "your name"
2. $ 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。

2. 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”。

2.1 第一步，创建

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：

```
1. $ mkdir learngit
2. $ cd learngit
3. $ pwd
4. /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。

2.2 第二步，git init

通过`git init`命令把这个目录变成Git可以管理的仓库：

```
1. $ git init
2. 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。

2.3 第三步，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：

```
1. Git is a version control system.  
2. 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：

```
1. $ 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：

```
1. $ git commit -m "wrote a readme file"  
2. [master (root-commit) cb926e7] wrote a readme file  
3.   1 file changed, 2 insertions(+)  
4.    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：

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

2.4 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。

3.时光机穿梭

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：

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

现在，运行`git status`命令看看结果：

```
1. $ git status
2. # On branch master
3. # Changes not staged for commit:
4. #   (use "git add <file>..." to update what will be committed)
5. #   (use "git checkout -- <file>..." to discard changes in working directory)
6. #
7. #       modified:   readme.txt
8. #
9. no changes added to commit (use "git add" and/or "git commit -a")
```

3.1 git status 和 git diff

`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：

```
1. $ git diff readme.txt
2. diff --git a/readme.txt b/readme.txt
3. index 46d49bf..9247db6 100644
4. --- a/readme.txt
5. +++ b/readme.txt
6. @@ -1,2 +1,2 @@
7. -Git is a version control system.
8. +Git is a distributed version control system.
9.   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。

3.2 git add 和 git commit

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：

```
1. $ git status  
2. # On branch master  
3. # Changes to be committed:  
4. #   (use "git reset HEAD <file>..." to unstage)  
5. #       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：

```
1. $ git commit -m "add distributed"  
2. [master ea34578] add distributed  
3. 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：

```
1. $ git status  
2. # On branch master  
3. 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。

3.3 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. 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：

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

然后尝试提交：

```
1. $ git add readme.txt  
2. $ git commit -m "append GPL"  
3. [master 3628164] append GPL  
4. 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。

4.1 查看版本与git log

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`

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

版本2：`add distributed`

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

版本3：`append GPL`

```
1. Git is a distributed version control system.  
2. 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：

```
1. $ git log  
2. commit 3628164fb26d48395383f8f31179f24e0882e1e0  
3. Author: Michael Liao <askxuefeng@gmail.com>  
4. Date: Tue Aug 20 15:11:49 2013 +0800  
5.  
6.     append GPL  
7.  
8. commit ea34578d5496d7dd233c827ed32a8cd576c5ee85  
9. Author: Michael Liao <askxuefeng@gmail.com>  
10. Date: Tue Aug 20 14:53:12 2013 +0800  
11.  
12.     add distributed  
13.  
14. commit cb926e7ea50ad11b8f9e909c05226233bf755030  
15. Author: Michael Liao <askxuefeng@gmail.com>  
16. Date: Mon Aug 19 17:51:55 2013 +0800  
17.  
18.     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：

```
1. $ git log --pretty=oneline  
2. 3628164fb26d48395383f8f31179f24e0882e1e0 append GPL  
3. ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed  
4. 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：

4.2 git reset (--head)

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：

```
1. $ git reset --hard HEAD^
2. 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`：

```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. 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：

```
1. $ git log
2. commit ea34578d5496d7dd233c827ed32a8cd576c5ee85
3. Author: Michael Liao <askxuefeng@gmail.com>
4. Date:   Tue Aug 20 14:53:12 2013 +0800
5.
6.     add distributed
7.
8. commit cb926e7ea50ad11b8f9e909c05226233bf755030
9. Author: Michael Liao <askxuefeng@gmail.com>
10. Date:  Mon Aug 19 17:51:55 2013 +0800
11.
12.     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：

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

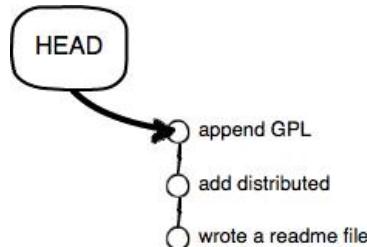
版本号没必要写全，前几位就可以了，**Git**会自动去找。当然也不能只写前一两位，因为**Git**可能会找到多个版本号，就无法确定是哪一个了。

再小心翼翼地看看 `readme.txt` 的内容：

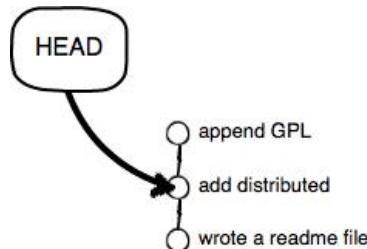
```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. Git is free software distributed under the GPL.
```

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。

4.3 git reflog

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：

```
1. $ git reflog
2. ea34578 HEAD@{0}: reset: moving to HEAD^
3. 3628164 HEAD@{1}: commit: append GPL
4. ea34578 HEAD@{2}: commit: add distributed
5. 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。

4.4 小结

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

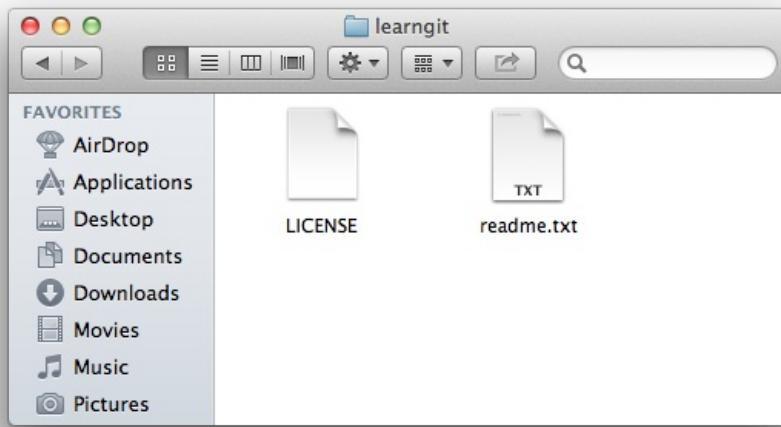
5.工作区和暂存区

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。

5.1 工作区 (Working Directory)

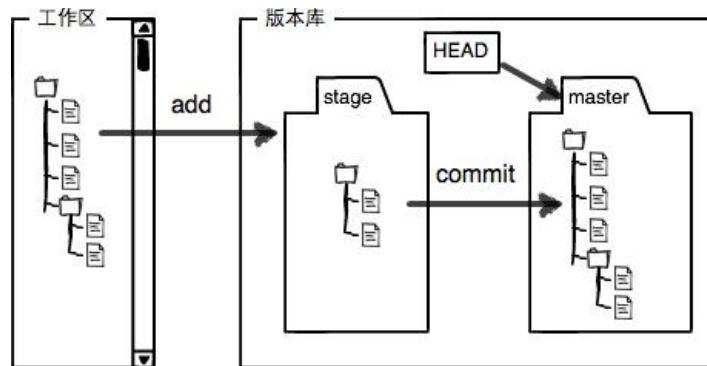
就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



5.2 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：

```
1. Git is a distributed version control system.  
2. Git is free software distributed under the GPL.  
3. 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：

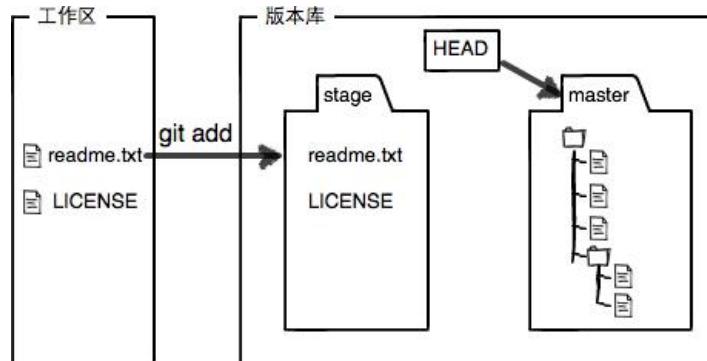
```
1. $ git status  
2. # On branch master  
3. # Changes not staged for commit:  
4. #   (use "git add <file>..." to update what will be committed)  
5. #   (use "git checkout -- <file>..." to discard changes in working directory)  
6. #  
7. #       modified:   readme.txt  
8. #  
9. # Untracked files:  
10. #   (use "git add <file>..." to include in what will be committed)  
11. #  
12. #       LICENSE  
13. 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：

```
1. $ git status  
2. # On branch master  
3. # Changes to be committed:  
4. #   (use "git reset HEAD <file>..." to unstage)  
5. #  
6. #       new file:   LICENSE  
7. #       modified:   readme.txt  
8. #
```

现在，暂存区的状态就变成这样了：



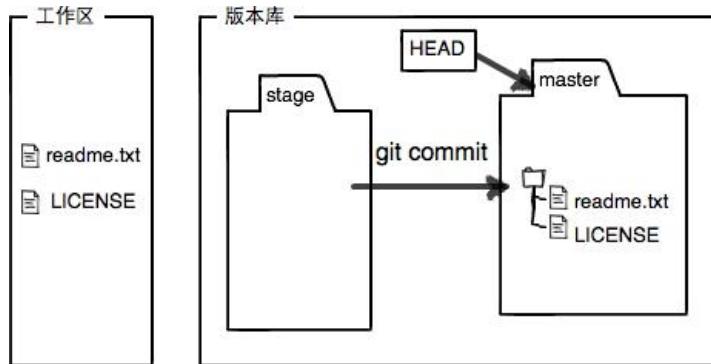
所以，`git add` 命令实际上就是把要提交的所有修改放到暂存区(Stage)，然后，执行 `git commit` 就可以一次性把暂存区的所有修改提交到分支。

```
1. $ git commit -m "understand how stage works"
2. [master 27c9860] understand how stage works
3.   2 files changed, 675 insertions(+)
4.    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：

```
1. $ git status
2. # On branch master
3. 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：



5.3 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。

6. 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。

6.1 第二次的修改没有被提交

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：

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

然后，添加：

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

```
2. $ git status
3. # On branch master
4. # Changes to be committed:
5. #   (use "git reset HEAD <file>..." to unstage)
6. #
7. #       modified:   readme.txt
8. #
```

然后，再修改readme.txt：

```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. Git is free software distributed under the GPL.
4. Git has a mutable index called stage.
5. Git tracks changes of files.
```

提交：

```
1. $ git commit -m "git tracks changes"
2. [master d4f25b6] git tracks changes
3.   1 file changed, 1 insertion(+)
```

提交后，再看看状态：

```
1. $ git status
2. # On branch master
3. # Changes not staged for commit:
4. #   (use "git add <file>..." to update what will be committed)
5. #   (use "git checkout -- <file>..." to discard changes in working directory)
6. #
7. #       modified:   readme.txt
8. #
9. 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？

6.2 问题回顾与工作区和版本库区别

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：

```
1. $ git diff HEAD -- readme.txt
2. diff --git a/readme.txt b/readme.txt
3. index 76d770f..a9c5755 100644
4. --- a/readme.txt
5. +++ b/readme.txt
6. @@ -1,4 +1,4 @@
7.   Git is a distributed version control system.
8.   Git is free software distributed under the GPL.
9.   Git has a mutable index called stage.
10.  -Git tracks changes.
11.  +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。

6.3 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。

7.撤销修改

7.1 情形一：犯错

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：

```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. Git is free software distributed under the GPL.
4. Git has a mutable index called stage.
5. Git tracks changes of files.
6. 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：

```
1. $ git status
2. # On branch master
3. # Changes not staged for commit:
4. #   (use "git add <file>..." to update what will be committed)
5. #   (use "git checkout -- <file>..." to discard changes in working directory)
6. #
7. #       modified:   readme.txt
8. #
9. no changes added to commit (use "git add" and/or "git commit -a")
```

7.2 git checkout

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

```
1. $ 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：

```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. Git is free software distributed under the GPL.
4. Git has a mutable index called stage.
5. 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。

7.3 错误地`git add`和`git reset HEAD file`补救

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：

```
1. $ cat readme.txt
2. Git is a distributed version control system.
3. Git is free software distributed under the GPL.
4. Git has a mutable index called stage.
5. Git tracks changes of files.
6. My stupid boss still prefers SVN.
7.
8. $ 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：

```
1. $ git status
2. # On branch master
3. # Changes to be committed:
4. #   (use "git reset HEAD <file>..." to unstage)
5. #
6. #       modified:   readme.txt
7. #
```

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：

```
1. $ git reset HEAD readme.txt
2. Unstaged changes after reset:
3. 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：

```
1. $ git status
2. # On branch master
3. # Changes not staged for commit:
4. #   (use "git add <file>..." to update what will be committed)
5. #   (use "git checkout -- <file>..." to discard changes in working directory)
```

```
6.  #
7.  #       modified:   readme.txt
8.  #
9.  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？

```
1.  $ git checkout -- readme.txt
2.
3.  $ git status
4.  # On branch master
5.  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……

7.4 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，不过前提是还没有推送到远程库。

8. 删除文件

8.1 删除

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：

```
1.  $ git add test.txt
2.  $ git commit -m "add test.txt"
3.  [master 94cdc44] add test.txt
4.    1 file changed, 1 insertion(+)
5.    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：

```
1.  $ 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：

```
1.  $ git status
2.  # On branch master
3.  # Changes not staged for commit:
4.  #   (use "git add" to update what will be committed)
5.  #   (use "git checkout --" to discard changes in working directory)
6.  #
7.  #       modified:   test.txt
```

```
4. # (use "git add/rm <file>..." to update what will be committed)
5. # (use "git checkout -- <file>..." to discard changes in working directory)
6. #
7. #       deleted:    test.txt
8. #
9. no changes added to commit (use "git add" and/or "git commit -a")
```

8.2 恢复或删除

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`：

```
1. $ git rm test.txt
2. rm 'test.txt'
3. $ git commit -m "remove test.txt"
4. [master d17efd8] remove test.txt
5.   1 file changed, 1 deletion(-)
6.   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：

```
1. $ 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”。

8.3 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。

9. 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。

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。

9.1 注册GitHub

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：

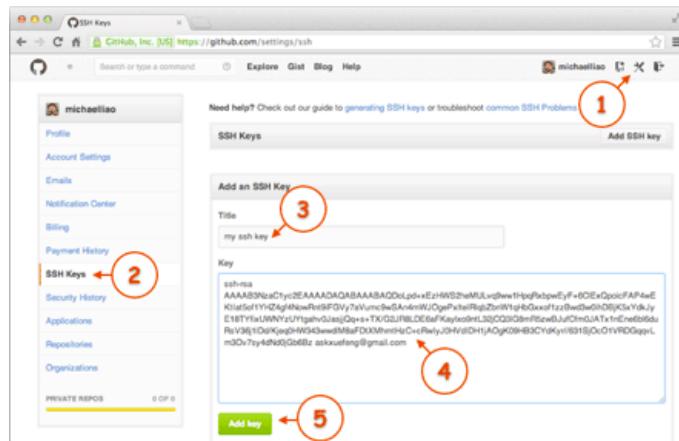
```
1. $ ssh-keygen -t rsa -C "youremail@example.com"
```

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。

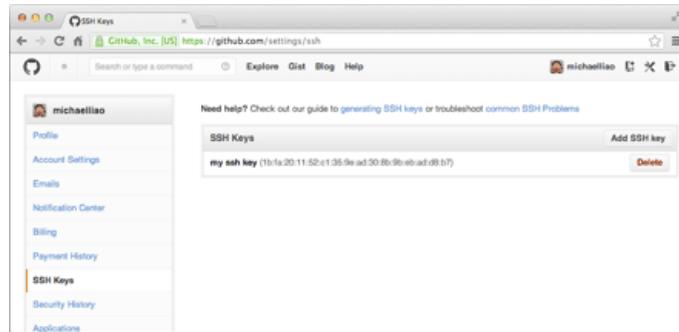
如果一切顺利的话，可以在用户主目录里找到.ssh目录，里面有 `id_rsa` 和 `id_rsa.pub` 两个文件，这两个就是**SSH Key**的秘钥对，`id_rsa`是私钥，不能泄露出去，`id_rsa.pub`是公钥，可以放心地告诉任何人。

- 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你的，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，这样别人就看不到了（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。

9.2 小结

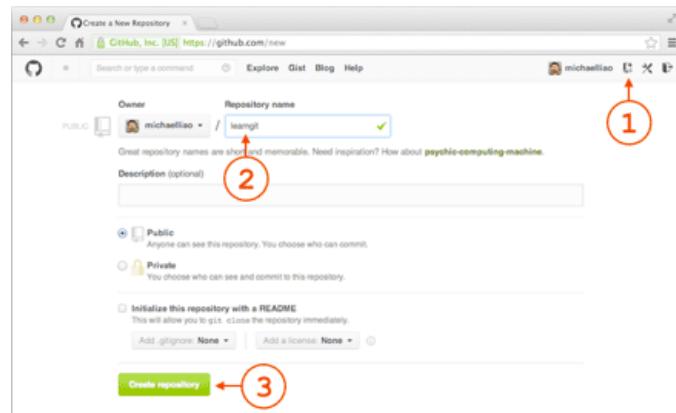
- ✓ “有了远程仓库，妈妈再也不用担心我的硬盘了。”——Git点读机

10.添加远程库

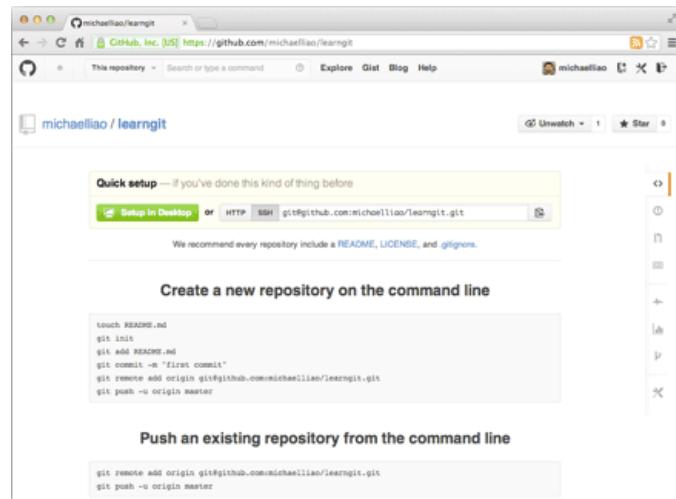
现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

10.1 创建新仓库

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：



10.2 本地仓库与远程仓库关联

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：

```
1. $ 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。

10.3 推送到远程库

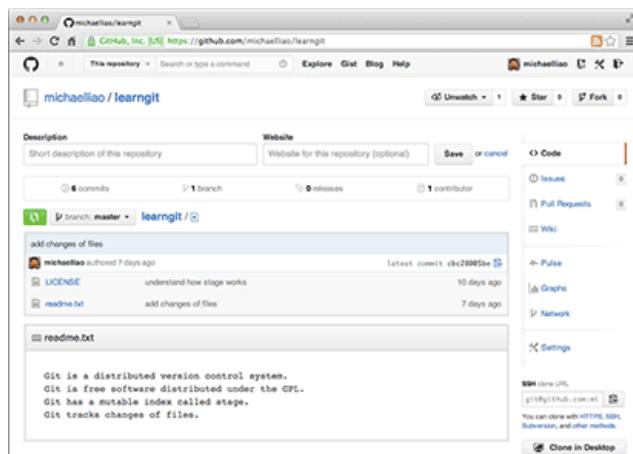
下一步，就可以把本地库的所有内容推送到远程库上：

```
1. $ git push -u origin master
2. Counting objects: 19, done.
3. Delta compression using up to 4 threads.
4. Compressing objects: 100% (19/19), done.
5. Writing objects: 100% (19/19), 13.73 KiB, done.
6. Total 23 (delta 6), reused 0 (delta 0)
7. To git@github.com:michaelliao/learngit.git
 * [new branch]      master -> master
8. 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：

```
1. $ 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！**

10.4 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：

```
1. The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.
2. RSA key fingerprint is xx.xx.xx.xx.xx.
3. 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：

```
1. 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。

10.5 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！

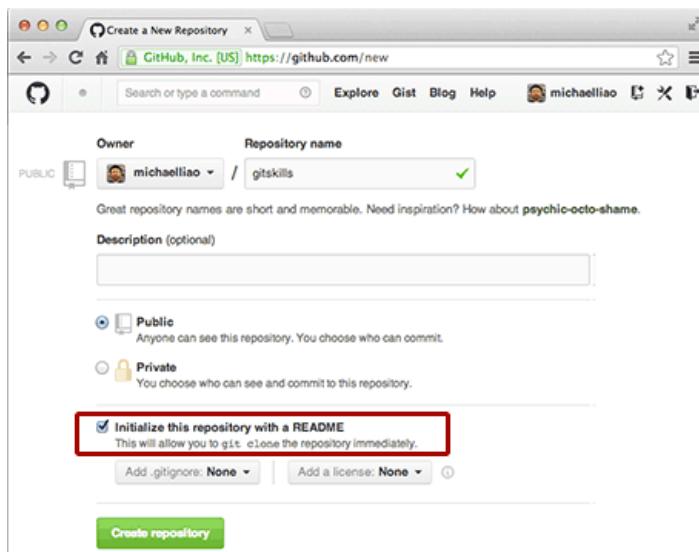
11.从远程库克隆

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。

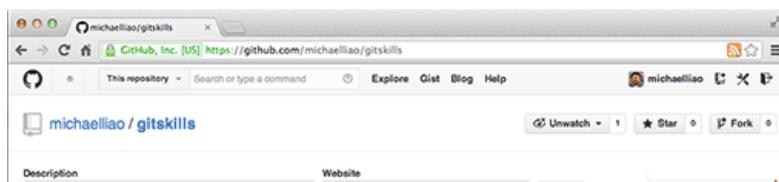
11.1 先创建远程库

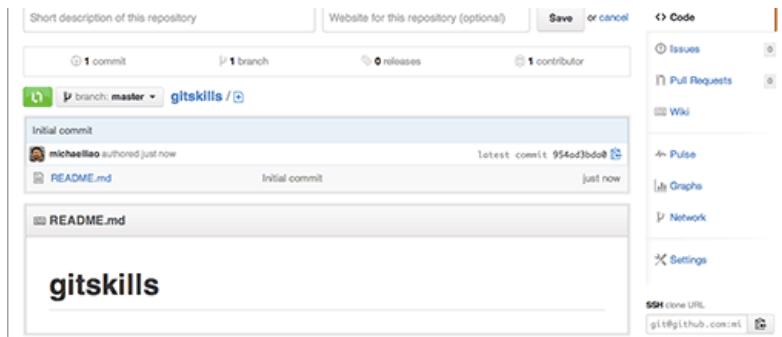
现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

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：





11.2 git clone 克隆

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：

```
1. $ git clone git@github.com:michaelliao/gitskills.git
2. Cloning into 'gitskills'...
3. remote: Counting objects: 3, done.
4. remote: Total 3 (delta 0), reused 0 (delta 0)
5. Receiving objects: 100% (3/3), done.
6.
7. $ cd gitskills
8. $ ls
9. 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。

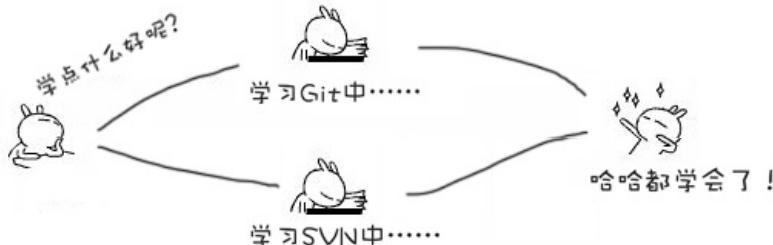
11.3 小结

- 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。

12. 分支管理简介

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。

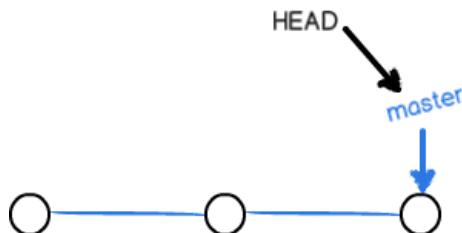
但Git的分支是与众不同的，无论创建、切换和删除分支，Git在1秒钟之内就能完成！无论你的版本库是1个文件还是1万个文件。

13. 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。

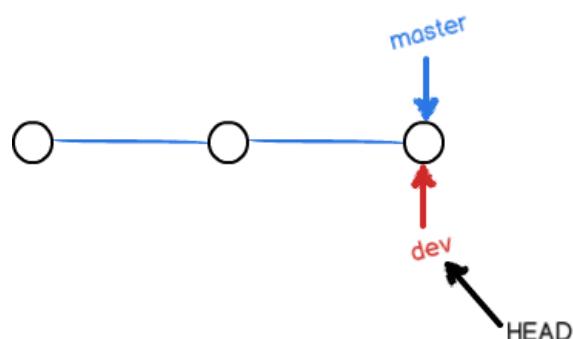
13.1 创建新分支

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：



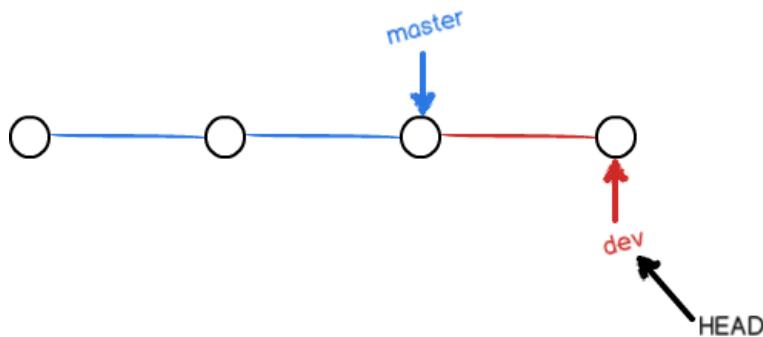
每次提交，**master** 分支都会向前移动一步，这样，随着你不断提交，**master** 分支的线也越来越长：

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：

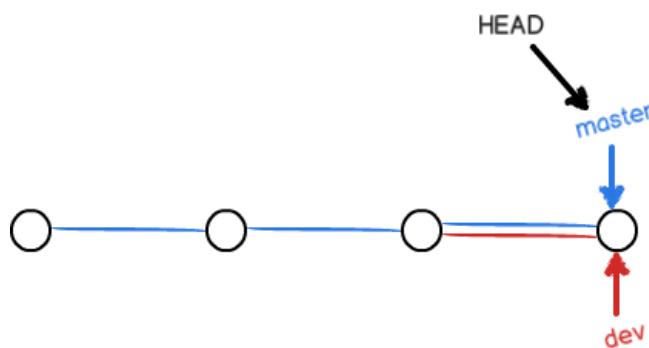


你看，**Git** 创建一个分支很快，因为除了增加一个 **dev** 指针，改改 **HEAD** 的指向，工作区的文件都没有任何变化！

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：

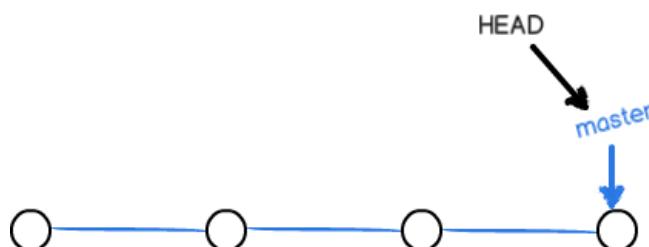


假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git 怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并：



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：



13.2 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：

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

修改 `readme.txt` 最后一行，改为：

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

在 `feature1` 分支上提交：

```
2. $ git commit -m AND simple
3. [feature1 75a857c] AND simple
4. 1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到 master 分支：

```
1. $ git checkout master
2. Switched to branch 'master'
3. 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。

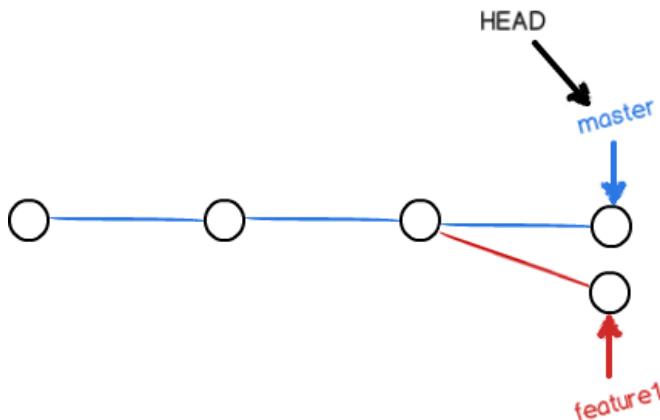
在 master 分支上把 *readme.txt* 文件的最后一行改为：

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

提交：

```
1. $ git add readme.txt
2. $ git commit -m "& simple"
3. [master 400b400] & simple
4. 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：



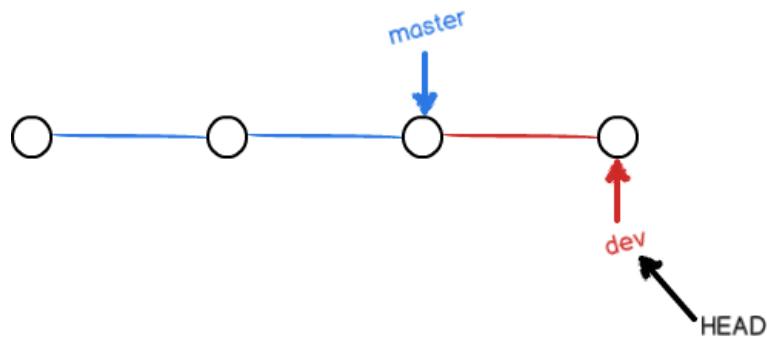
这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

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

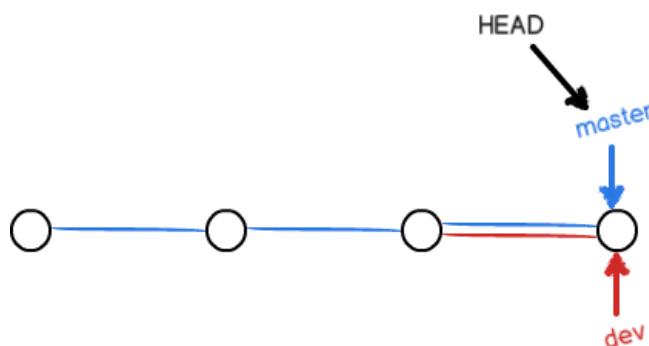
果然冲突了！Git 告诉我们，*readme.txt* 文件存在冲突，必须手动解决冲突后再提交。`git status` 也可以告诉我们冲突的文件：

```
1. $ git status
2. # On branch master
3. # Your branch is ahead of 'origin/master' by 2 commits.
4. #
5. # Unmerged paths:
6. #   (use "git add/rm <file>..." as appropriate to mark resolution)
7. #
8. #       both modified:      readme.txt
9. #
10. no changes added to commit (use "git add" and/or "git commit -a")
```

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：

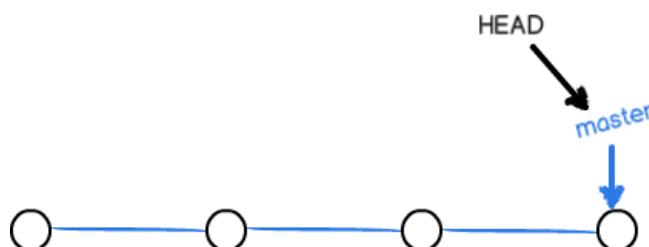


假如我们在 `dev` 上的工作完成了，就可以把 `dev` 合并到 `master` 上。Git 怎么合并呢？最简单的方法，就是直接把 `master` 指向 `dev` 的当前提交，就完成了合并：



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：



13.2 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：

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

修改 `readme.txt` 最后一行，改为：

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

在 `feature1` 分支上提交：

```
1. $ git add readme.txt
2. git commit -m "AND simple"
```

```
1. # ELL COMMIT -> AND SIMPLE
2. [feature1 75a857c] AND simple
3.   1 file changed, 1 insertion(+), 1 deletion(-)
```

切换到 master 分支：

```
1. $ git checkout master
2. Switched to branch 'master'
3. 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。

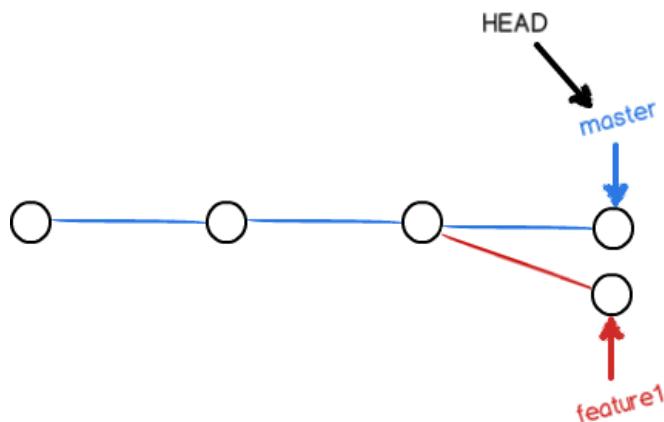
在 master 分支上把 *readme.txt*文件的最后一行改为：

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

提交：

```
1. $ git add readme.txt
2. $ git commit -m "& simple"
3. [master 400b400] & simple
4.   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：

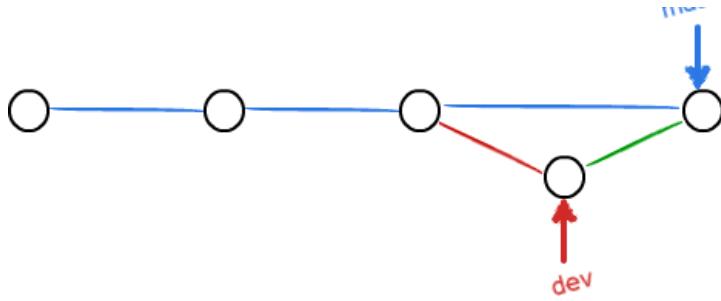


这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

```
1. $ git merge feature1
2. Auto-merging readme.txt
3. CONFLICT (content): Merge conflict in readme.txt
4. 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：

```
1. $ git status
2. # On branch master
3. # Your branch is ahead of 'origin/master' by 2 commits.
4. #
5. # Unmerged paths:
6. #   (use "git add/rm <file>..." as appropriate to mark resolution)
7. #
8. #       both modified:      readme.txt
9. #
10. no changes added to commit (use "git add" and/or "git commit -a")
```



14.1 分支策略

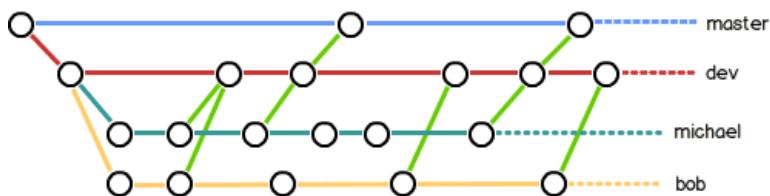
在实际开发中，我们应该按照几个基本原则进行分支管理：

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：



14.2 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。

15.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：

```

1. $ git status
2. # On branch dev
3. # Changes to be committed:
4. #   (use "git reset HEAD <file>..." to unstage)
5. #
6. #       new file:   hello.py
7. #
8. # Changes not staged for commit:

```

```
9. #   (use "git add <file>..." to update what will be committed)
10. #   (use "git checkout -- <file>..." to discard changes in working directory)
11. #
12. #       modified:   readme.txt
13. #
```

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：

```
1. $ git stash
2. Saved working directory and index state WIP on dev: 6224937 add merge
3. 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：

```
1. $ git checkout master
2. Switched to branch 'master'
3. Your branch is ahead of 'origin/master' by 6 commits.
4. $ git checkout -b issue-101
5. 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：

```
1. $ git add readme.txt
2. $ git commit -m "fix bug 101"
3. [issue-101 cc17032] fix bug 101
4. 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：

```
1. $ git checkout master
2. Switched to branch 'master'
3. Your branch is ahead of 'origin/master' by 2 commits.
4. $ git merge --no-ff -m "merged bug fix 101" issue-101
5. Merge made by the 'recursive' strategy.
6. readme.txt |    2 ++
7. 1 file changed, 1 insertion(+), 1 deletion(-)
8. $ git branch -d issue-101
9. 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！

```
1. $ git checkout dev
2. Switched to branch 'dev'
3. $ git status
4. # On branch dev
5. 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：

```
1. $ git stash list
2. 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：

```
1. $ git stash pop
2. # On branch dev
3. # Changes to be committed:
4. #   (use "git reset HEAD <file>..." to unstage)
5. #
6. #       new file:   hello.py
7. #
8. # Changes not staged for commit:
9. #   (use "git add <file>..." to update what will be committed)
10. #    (use "git checkout -- <file>..." to discard changes in working directory)
11. #
12. #       modified:   readme.txt
13. #
14. Dropped refs/stash@{0} (f624f8e5f082f2df2bed8a4e09c12fd2943bdd40)
```

再用 `git stash list` 查看，就看不到任何 **stash** 内容了：

```
1. $ 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：

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

15.1 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。

16. 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。

16.1 添加新功能

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：

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

5分钟后，开发完毕：

```
1. $ git add vulcan.py
2. $ git status
3. # On branch feature-vulcan
4. # Changes to be committed:
5. #   (use "git reset HEAD <file>..." to unstage)
6. #
7. #       new file:   vulcan.py
8. #
9. $ git commit -m "add feature vulcan"
10. [feature-vulcan 756d4af] add feature vulcan
11. 1 file changed, 2 insertions(+)
12. create mode 100644 vulcan.py
```

切回dev，准备合并：

```
1. $ 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。

16.2 删 除 分 支

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：

```
1. $ git branch -d feature-vulcan
2. error: The branch 'feature-vulcan' is not fully merged.
3. 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：

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

终于删除成功！

16.3 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。

17. 多人协作

17.1 查看远程库

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**：

```
1. $ git remote  
2. origin
```

或者，用 **git remote -v** 显示更详细的信息：

```
1. $ git remote -v  
2. origin  git@github.com:michaelliao/learngit.git (fetch)  
3. 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。

17.2 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：

```
1. $ git push origin master
```

如果要推送其他分支，比如 **dev**，就改成：

```
1. $ 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多少个 **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！

17.3 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：

```
1. $ git clone git@github.com:michaelliao/learngit.git  
2. Cloning into 'learngit'...  
3. remote: Counting objects: 46, done.  
4. remote: Compressing objects: 100% (26/26), done.  
5. remote: Total 46 (delta 16), reused 45 (delta 15)
```

```
6. Receiving objects: 100% (46/46), 15.69 KiB | 6 KiB/s, done.  
7. 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：

```
1. $ git branch  
2. * 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：

```
1. $ 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：

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

17.4 解决冲突

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：

```
1. $ git add hello.py  
2. $ git commit -m "add coding: utf-8"  
3. [dev bd6ae48] add coding: utf-8  
4.   1 file changed, 1 insertion(+)  
5. $ git push origin dev  
6. To git@github.com:michaelliao/learngit.git  
7. ! [rejected]          dev -> dev (non-fast-forward)  
8. error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'  
9. hint: Updates were rejected because the tip of your current branch is behind  
10. hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')  
11. hint: before pushing again.  
12. 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：

```
1. $ git pull  
2. remote: Counting objects: 5, done.  
3. remote: Compressing objects: 100% (2/2), done.  
4. remote: Total 3 (delta 0), reused 3 (delta 0)  
5. Unpacking objects: 100% (3/3), done.  
6. From github.com:michaelliao/learngit  
7.     fc38031..291bea8  dev      -> origin/dev  
8. There is no tracking information for the current branch.  
9. Please specify which branch you want to merge with.  
10. See git-pull(1) for details  
11.
```

```
12.     git pull <remote> <branch>
13.
14. If you wish to set tracking information for this branch you can do so with:
15.
16.     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：

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

再 **pull**：

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

这次 **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**：

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

17.5 多人协作的工作模式

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。

17.6 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。

18. 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，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拿出来。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**。

18.1 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：

```
1. $ git branch
2. * dev
3.   master
4. $ git checkout master
5. Switched to branch 'master'
```

然后，敲命令 `git tag <name>` 就可以打一个新标签：

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

可以用命令 `git tag` 查看所有标签：

```
1. $ git tag
2. v1.0
```

18.2 忘打标签

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：

```
1. $ git log --pretty=oneline --abbrev-commit
2. 6a5819e merged bug fix 101
3. cc17032 fix bug 101
4. 7825a50 merge with no-ff
5. 6224937 add merge
6. 59bc1cb conflict fixed
7. 400b400 & simple
8. 75a857c AND simple
9. fec145a branch test
10. d17efd8 remove test.txt
11. ...
```

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：

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

再用命令 `git tag` 查看标签：

```
1. $ git tag  
2. v0.9  
3. 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：

```
1. $ git show v0.9  
2. commit 622493706ab447b6bb37e4e2a2f276a20fed2ab4  
3. Author: Michael Liao <askxuefeng@gmail.com>  
4. Date: Thu Aug 22 11:22:08 2013 +0800  
5.  
6.     add merge  
7. ...
```

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：

```
1. $ 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：

```
1. $ git show v0.1  
2. tag v0.1  
3. Tagger: Michael Liao <askxuefeng@gmail.com>  
4. Date: Mon Aug 26 07:28:11 2013 +0800  
5.  
6. version 0.1 released  
7.  
8. commit 3628164fb26d48395383f8f31179f24e0882e1e0  
9. Author: Michael Liao <askxuefeng@gmail.com>  
10. Date: Tue Aug 20 15:11:49 2013 +0800  
11.  
12.     append GPL
```

还可以通过 `-s` 用私钥签名一个标签：

```
1. $ 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：

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

如果报错，请参考GnuPG帮助文档配置Key。

18.3 查看PGP签名信息

用命令 `git show <tagname>` 可以看到PGP签名信息：

```
1. $ git show v0.2
```

```
1. $ git show v0.2
2. tag v0.2
3. Tagger: Michael Liao <askxuefeng@gmail.com>
4. Date: Mon Aug 26 07:28:33 2013 +0800
5.
6. signed version 0.2 released
7. -----BEGIN PGP SIGNATURE-----
8. Version: GnuPG v1.4.12 (Darwin)
9.
10. iQEcBAABAgAGBQJSGpMhAAoJEPUxHyDAhBpT4QQIAKeHfR3bo...
11. -----END PGP SIGNATURE-----
12.
13. commit fec145accd63cdc9ed95a2f557ea0658a2a6537f
14. Author: Michael Liao <askxuefeng@gmail.com>
15. Date: Thu Aug 22 10:37:30 2013 +0800
16.
17.     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。

18.4 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。

19. 操作标签

19.1 删除标签

如果标签打错了，也可以删除：

```
1. $ git tag -d v0.1
2. 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>`：

```
1. $ git push origin v1.0
2. Total 0 (delta 0), reused 0 (delta 0)
3. To git@github.com:michaelliao/learngit.git
4. * [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：

```
1. $ git push origin --tags
2. Counting objects: 1, done.
3. Writing objects: 100% (1/1), 554 bytes, done.
4. Total 1 (delta 0), reused 0 (delta 0)
```

```
5. To git@github.com:michaelliao/learngit.git
6. * [new tag]      v0.2 -> v0.2
7. * [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：

```
1. $ git tag -d v0.9
2. 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：

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

要看看是否真的从远程库删除了标签，可以登陆GitHub查看。

19.2 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。

20. 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。

20.1 参与开源项目

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。

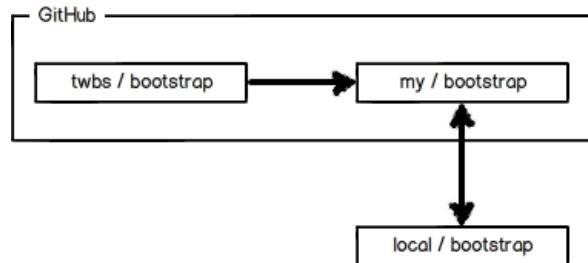
但是在GitHub上，利用Git极其强大的克隆和分支功能，广大人民群众真正可以第一次自由参与各种开源项目了。

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：

```
1. 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：



20.2 pull request

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。

20.3 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。

21.自定义Git

21.1 上色

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：

```
1. $ 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：

A screenshot of a macOS terminal window titled "learngit - bash - 80x24". The window displays the output of the command `git status`. The output shows a file named "hello.py" with a green "modified" status, and a file named "test" with a blue "untracked" status. The terminal interface includes a menu bar at the top and a scroll bar on the right side.

```
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。

21.2 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：

```
1. # Windows:  
2. Thumbs.db  
3. ehthumbs.db  
4. Desktop.ini
```

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

```
1. # Python:  
2. *.py[cod]  
3. *.so  
4. *.egg  
5. *.egg-info  
6. dist  
7. 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：

```
1. # Windows:  
2. Thumbs.db  
3. ehthumbs.db  
4. Desktop.ini  
5. # Python:  
6. *.py[cod]  
7.
```

```
8. *.so
9. *.egg
10. *.egg-info
11. dist
12. build
13.
14. # My configurations:
15. db.ini
16. 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。

21.3 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！

22.配置别名

22.1 "git st" Vs "git status"

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：

```
1. $ 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：

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

以后提交就可以简写成：

```
1. $ 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：

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

当你敲入命令：

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

实际上Git执行的是：

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

配置一个 `git last`，让其显示最后一次提交信息：

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

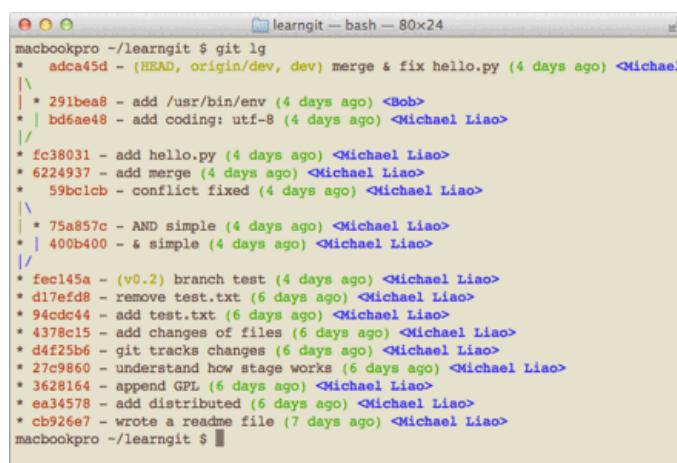
这样，用 `git last` 就能显示最近一次的提交：

```
1. $ git last
2. commit adca45d317e6d8a4b23f9811c3d7b7f0f180bfe2
3. Merge: bd6ae48 291bea8
4. Author: Michael Liao <askxuefeng@gmail.com>
5. Date:   Thu Aug 22 22:49:22 2013 +0800
6.
7.     merge & fix hello.py
```

甚至还有人丧心病狂地把 `lg` 配置成了：

```
1. 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：



```
learngit -- bash -- 80x24
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！

22.2 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：

```
1. $ cat .git/config
2. [core]
```

```
3.     repositoryformatversion = 0
4.     filemode = true
5.     bare = false
6.     logallrefupdates = true
7.     ignorecase = true
8.     precomposeunicode = true
9. [remote "origin"]
10.    url = git@github.com:michaelliao/learngit.git
11.    fetch = +refs/heads/*:refs/remotes/origin/*
12. [branch "master"]
13.    remote = origin
14.    merge = refs/heads/master
15. [alias]
16.    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：

```
1. $ cat .gitconfig
2. [alias]
3.   co = checkout
4.   ci = commit
5.   br = branch
6.   st = status
7. [user]
8.   name = Your Name
9.   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。

22.3 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。

23.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。

23.1 搭建服务器

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：

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

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

- 第二步，创建一个git用户，用来运行git服务：

```
1. $ 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：

```
1. $ 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：

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

改为：

```
1. 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：

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

剩下的推送就简单了。

23.2 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。

23.3 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。

23.4 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。

24.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！

