# Git使用手册

目录：



## Git是什么

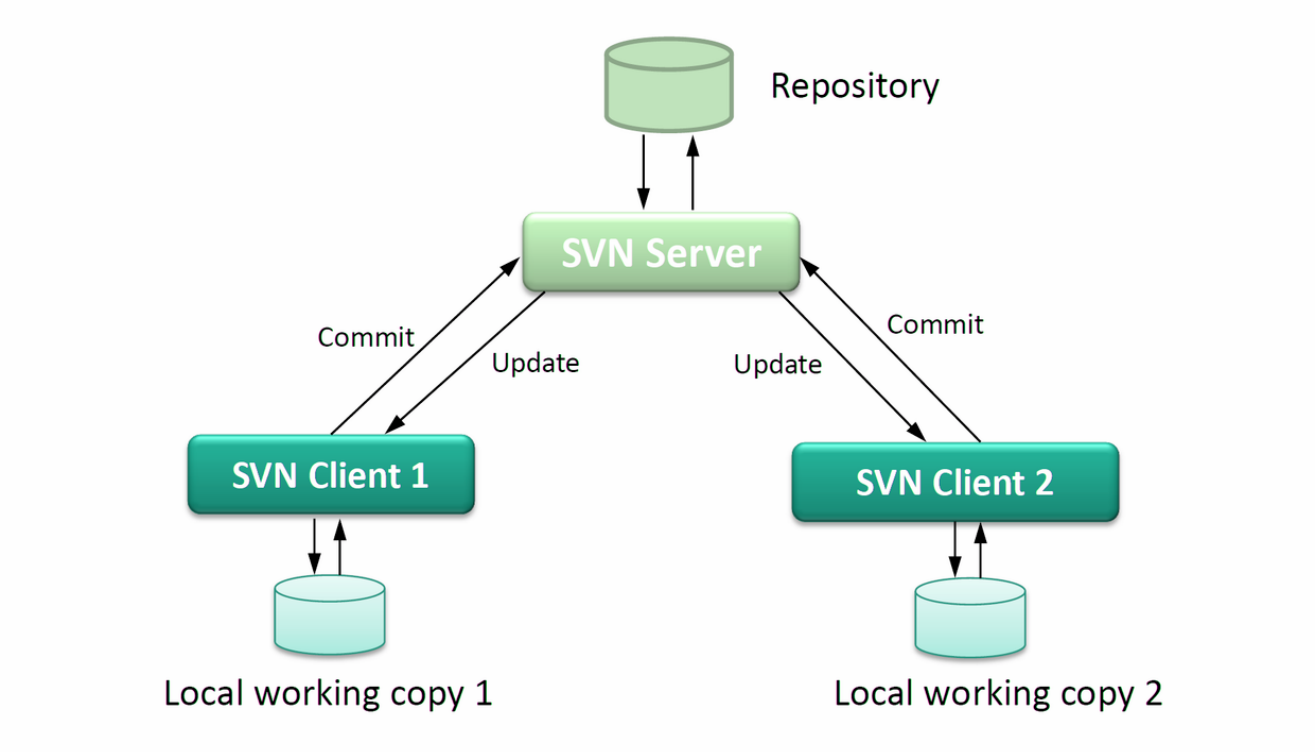
Git是目前世界上最先进的分布式版本控制系统（没有之一）。

对于文件操作，长期使用电脑办公的人想必深有体会。对于学生而言，当我们对报告、论文、工作进展等等一改再改、一版再版时，我们需要这样一个工具来帮我们记录我们对文件执行过的增删修改操作，甚至可以对版本进行回滚。这个时候，`git`就派上大用场啦。

### 基本概念

版本库(Repository)是版本控制系统用来存放所有历史数据的地方，主要存放各个文件的当前状态，历史修改时间，谁做的修改，以及修改的原因。举个简单的例子，就好比银行的保险箱，每次往里存钱，都会记录谁，什么时间，存放多少钱，存入的原因等。对应的版本库（Repository）主要存放代码（文档，数据，图标等），并且每一次更新都要记录谁，什么时间，提交了什么更新，以及更新的原因是什么。

git就是管理我们这个版本库的管家，相当于银行保险箱的管理人员。以前的版本控制入CVS，SVN等都是集中控制管理的，也就是有一个中央服务器，大家都把代码提交到中心节点(入下图)，而git是分布式的版本控制工具，也就是说没有中央服务器，每个节点的地位平等，有点P2P的味道，众生平等，谁也别瞧不起谁！^_^




## Git的诞生

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

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

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

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

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

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

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

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

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

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

# Git的安装与配置

在Ubuntu上，直接通过命令

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

即可安装完成。

其他系统安装参考http://www.liaoxuefeng.com/wiki/001373951630592960dd18361248578c67b8067c8c017b000/001373962877033 54d8c6c01c904c7d9ff056ae23da865a000。

配置：
安装完成后，在命令行输入

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

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。

检查配置是否成功：`git config --global --list` 查看设置的用户名和Email。

如果上面和下面要讲的任何命令有疑问，使用命令`git help <command>`查看command的用法。

# 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。例如在我的电脑，这个仓库位于`/Users/michael/learngit`。

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。

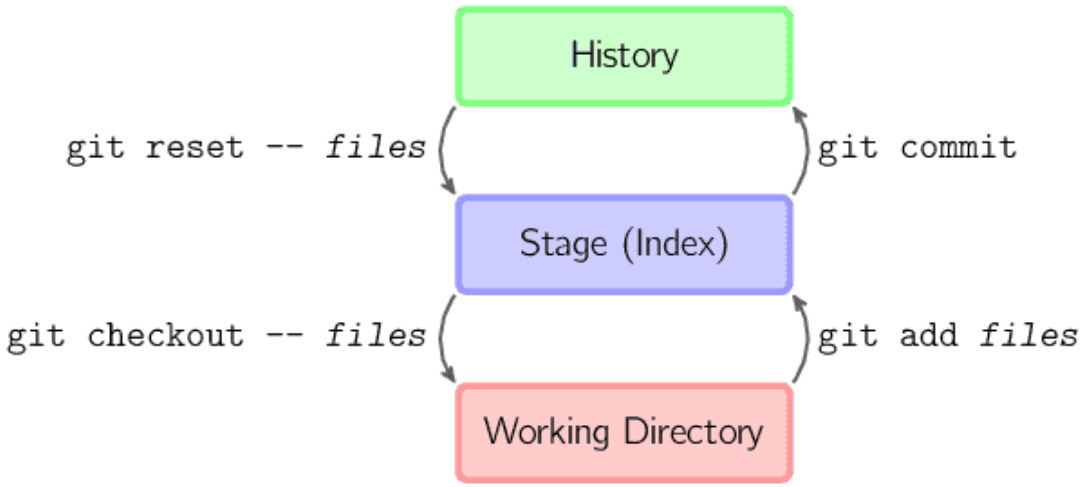
如果你没有看到`.git`目录，那是因为这个目录默认是隐藏的，用`ls -ah`命令就可以看见。

# Git操作略览

`git status`命令可以让我们时刻掌握仓库当前的状态。

`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。

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`，在执行第二步`git commit`之前，我们可以再运行git status看看当前仓库的状态。




对上图的说明：`working directory`是当前的工作目录，而`stage`是暂存区也称索引区存放工作目录中那些你打算提交到版本库的变更，`git add`只是将文件的索引提交的版本库，而真正的内容并没有进入版本库，`History`就是版本库，需要注意的是这是本地的版本库，存在于本地的电脑中，相当于你电脑上一个你的私人钱财管理员。

两个常用操作经常一块使用

```
    git add some-file      # 添加文件到暂存区
    git commit -m "some changes to some-file"   # 提交修改到版本库，并可以添加信息以便于区分和管理
```

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`命令显示从最近到最远的提交日志，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。

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，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`。

所以查看文件改动，我们可以通过

```
git diff            比较working directory 和stage的差别

git diff --cached   比较stage和history的差别


git diff HEAD        直接比较working directory  和history的区别
```

如果要回退，可以使用`git reset`命令

```
$ git reset --hard HEAD^
```

如果这个时候你想要回退到最新版本，可以通过`commit-id`找回，如果关闭了终端，可能就得另寻他法了——Git提供了一个命令`git reflog`用来记录你的每一次命令，可以通过这个`id`进行找回。

```
$ git reset --hart commit-id
```

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。

在Git中，删除也是一个修改操作。在你删除文件后，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。

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`。

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 -- your_file
```

`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”。

# 远程仓库： git的杀招

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。

Git是分布式版本控制系统，同一个Git仓库，可以分布到不同的机器上。怎么分布呢？最早，肯定只有一台机器有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分。

实际情况往往是这样，找一台电脑充当服务器的角色，每天24小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交。

完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫GitHub的神奇的网站，从名字就可以看出，这个网站就是提供Git仓库托管服务的，所以，只要注册一个GitHub账号，就可以免费获得Git远程仓库。

在继续阅读后续内容前，请自行注册GitHub账号。由于你的本地Git仓库和GitHub仓库之间的传输是通过SSH加密的，所以，需要一点设置：

第1步：创建**SSH Key**。在用户主目录下，看看有没有.ssh目录，如果有，再看看这个目录下有没有`id_rsa`和`id_rsa.pub`这两个文件，如果已经有了，可直接跳到下一步。如果没有，打开Shell（Windows下打开Git Bash），创建SSH Key：

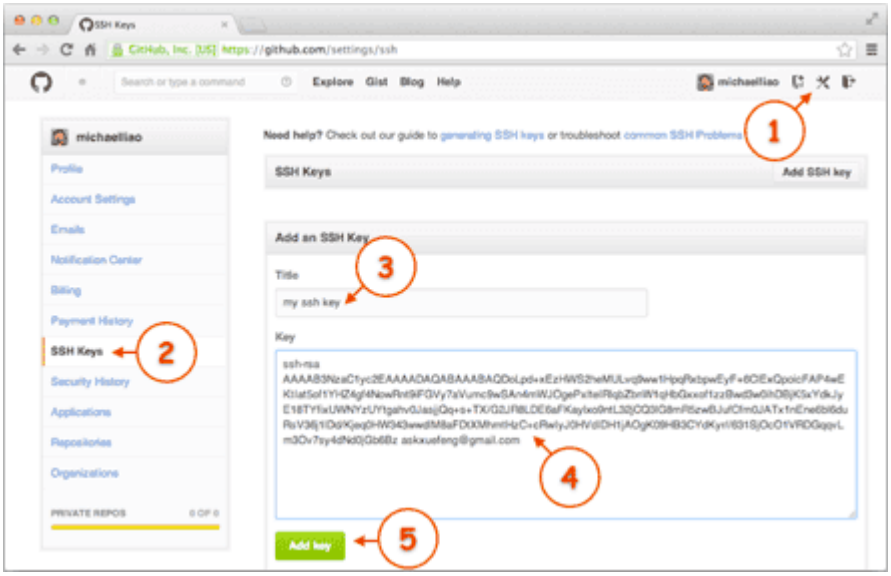
```
$ ssh-keygen -t rsa -C "youremail@example.com"
```

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

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

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

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




为什么GitHub需要SSH Key呢？因为GitHub需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而Git支持SSH协议，所以，GitHub只要知道了你的公钥，就可以确认只有你自己才能推送。

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

最后友情提示，在GitHub上免费托管的Git仓库，任何人都可以看到喔（但只有你自己才能改）。所以，不要把敏感信息放进去（这个确实需要注意，可以共享的东西随便存储大家看看也没什么，关键的信息和资料还是要注意些）。

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

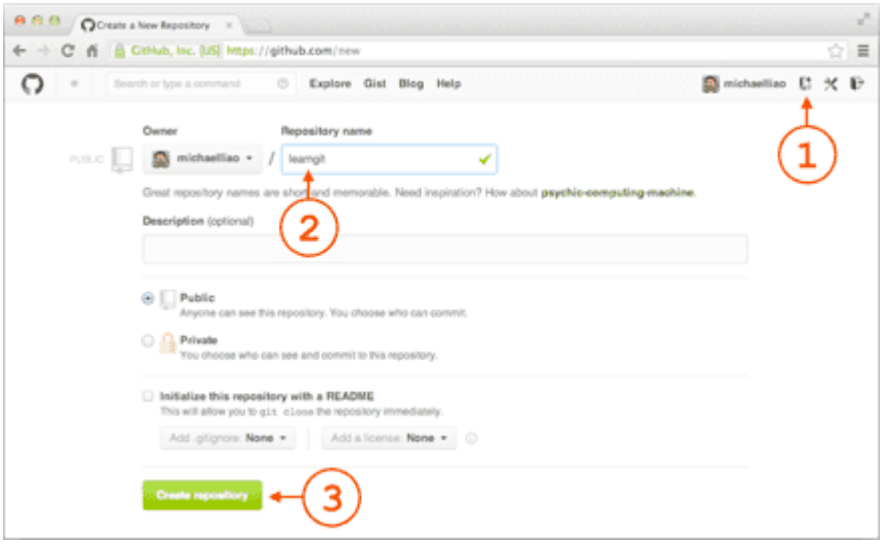
确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。

## 添加远程库

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。

（下面我自己参考博文构建一个新的远程库）

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：




因为我最近想要学习`bioconductor`，所以添加一个记录库，方便记录、查找、更新。

在Repository name填入`bioconductor_learn`，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。

```
#create a new repository on the command line

echo "# bioconductor_learn"  >> README.md
  git init
  git add README.md
  git commit -m "first commit"
  git remote add origin https://github.com/ShixiangWang/bioconductor_learn.git
  git push -u origin master
```

```
#  push an existing repository from the command line

git remote add origin https://github.com/ShixiangWang/bioconductor_learn.git
  git push -u origin master
```

现在，我们根据GitHub的提示，运行相应命令

```
wsx@wsx-ubuntu:~/桌面/Bioconductor_learn$ git add README.md
wsx@wsx-ubuntu:~/桌面/Bioconductor_learn$ git commit -m "first commit"
[master （根提交） c3cb52a] first comm it
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
wsx@wsx-ubuntu:~/桌面/Bioconductor_learn$ git remote add origin
https://github.com/ShixiangWang/bioconductor_learn.git

wsx@wsx-ubuntu:~/桌面/Bioconductor_learn$ git push -u origin masterUsername for
'https://github.com' : ShixiangWang
Password for 'https://ShixiangWang@github.com' :
对象计数中: 3, 完成.
写入对象中: 100% (3/3), 243 bytes | 0 bytes/s, 完成.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ShixiangWang/bioconductor_learn.git
 * [new branch]      master -> master
分支 master 设置为跟踪来自 origin 的远程分支 master。
```

请千万注意，把上面的账户名替换成你自己的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。

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！

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！

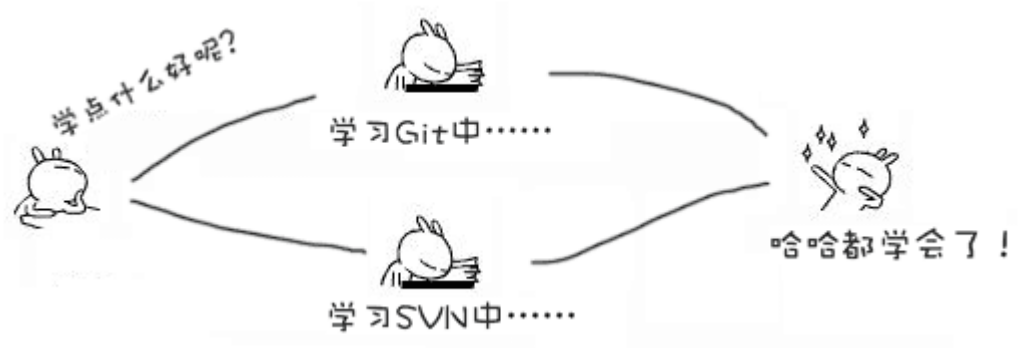
## 远程库克隆

如果是克隆别人的库，直接调用`git clone`命令后接库名地址即可。

自己建库然后克隆略显麻烦，多了前面创库的几个步骤，然后后面也是用`git clone`命令。

具体参考：从远程库克隆

# 分支管理




分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

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

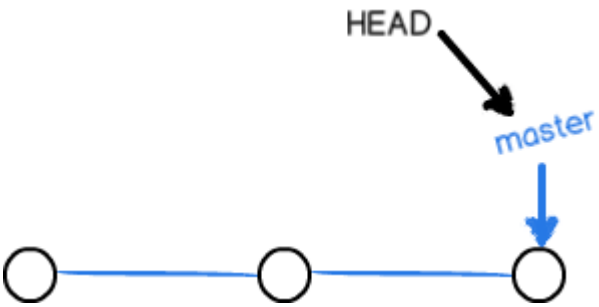
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

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。

## 创建和合并分支

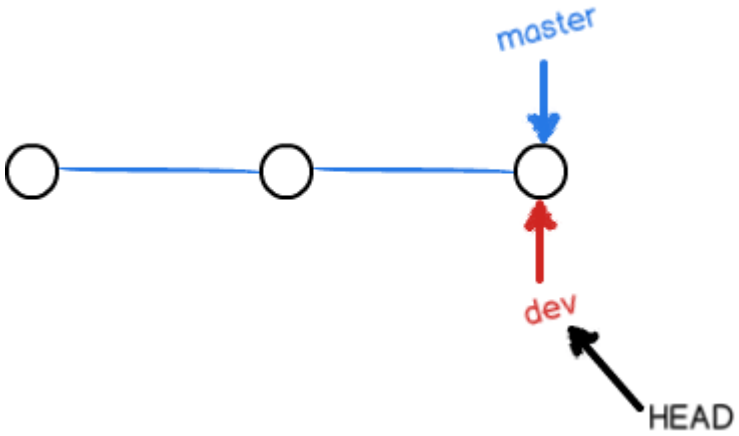
我们已经知道，git把版本串成一条时间线，这条时间线就是分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即`master`分支。`HEAD`严格来说不是指向提交，而是指向`master`，`master`才是指向提交的，所以，`HEAD`指向的就是当前分支。

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：



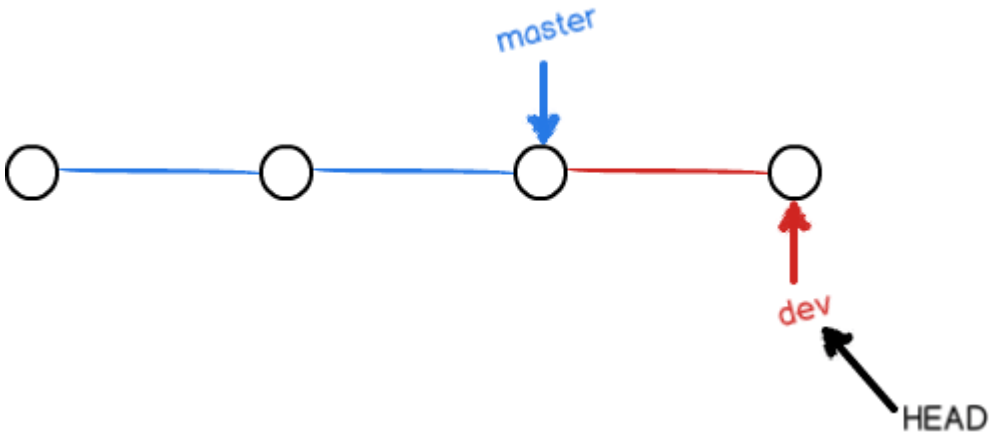

每次提交，`master`分支都会向前移动一步，这样，随着你不断提交，`master`分支的线也越来越长。

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：

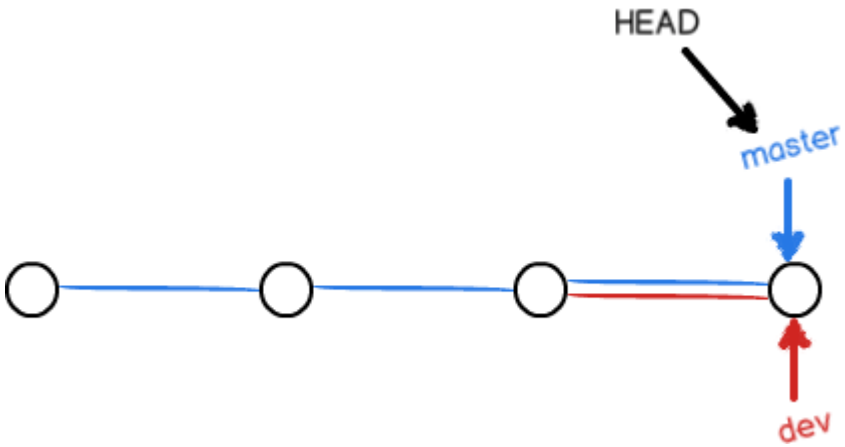



你看，Git创建一个分支很快，因为除了增加一个`dev`指针，改改`HEAD`的指向，工作区的文件都没有任何变化！

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：

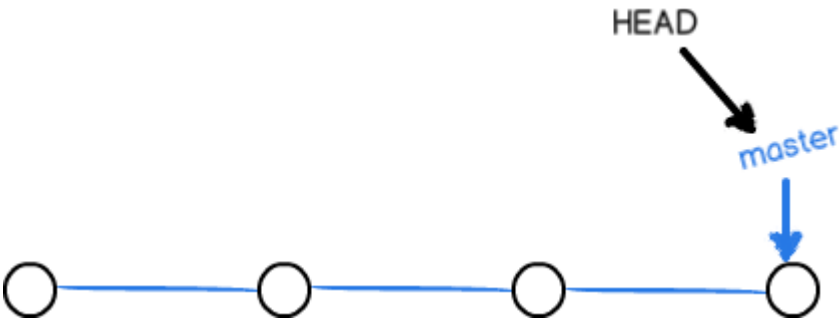



假如我们在`dev`上的工作完成了，就可以把`dev`合并到`master`上。Git怎么合并呢？最简单的方法，就是直接把`master`指向`dev`的当前提交，就完成了合并：




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：




## 分支实战

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.
```

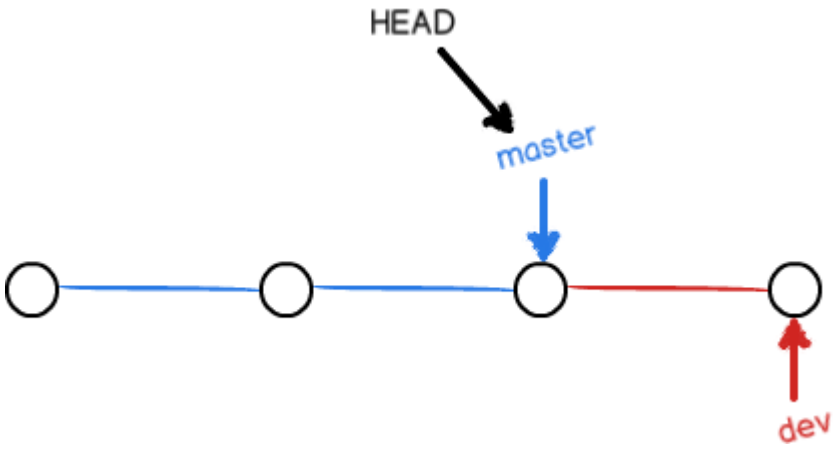
然后提交：

```
$ 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`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。(感觉这种方式在进行一些试探性的操作时非常安全有用呢)

## 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>`

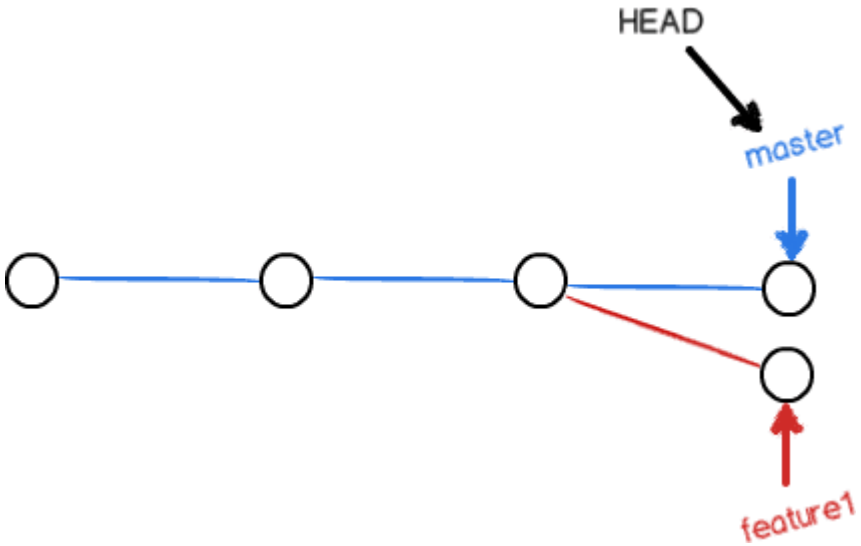
合并某分支到当前分支：`git merge <name>`

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。

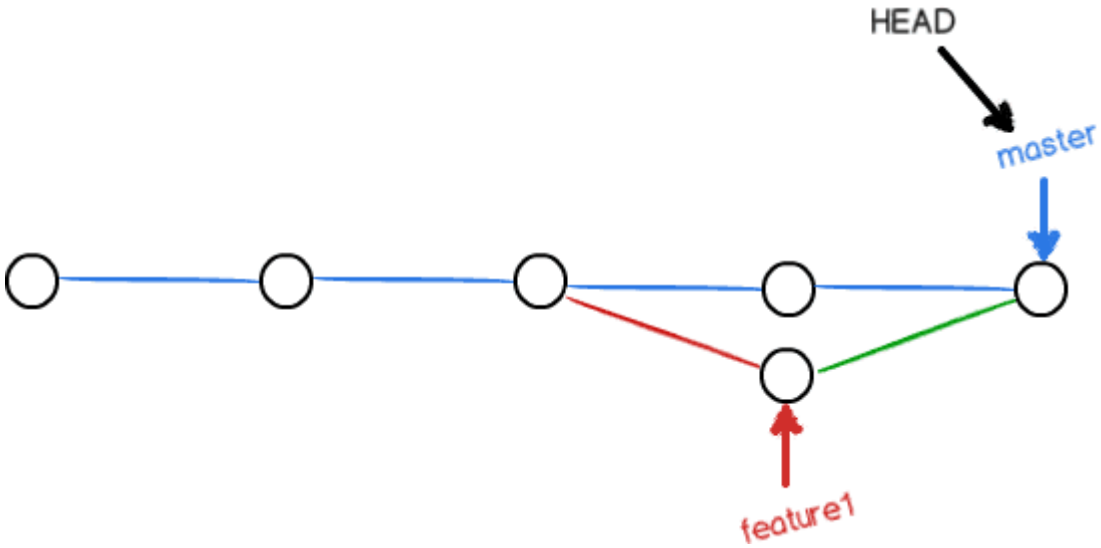
删除分支：`git branch -d <name>`

## 解决冲突

当我们在`master`和另一个分支对某个文档同时进行了修改，`git`无法对它们进行合并，导致冲突。




我们需要手动修改后提交：




具体实例参考解决冲突

## 分支策略

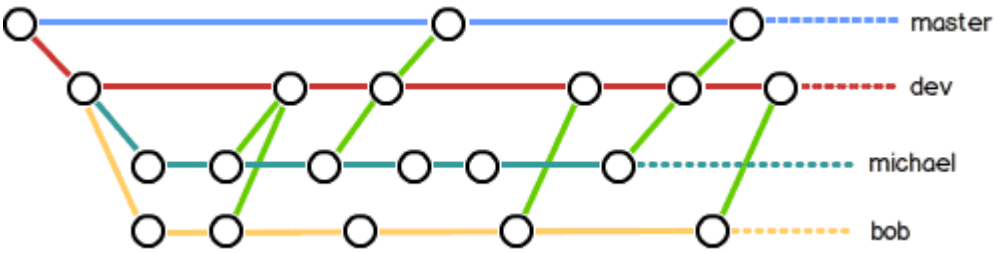
在实际开发中，我们应该按照几个基本原则进行分支管理：

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

那在哪干活呢？干活都在dev分支上，也就是说，dev分支是不稳定的，到某个时候，比如1.0版本发布时，再把dev分支合并到master上，在master分支发布1.0版本；

你和你的小伙伴们每个人都在dev分支上干活，每个人都有自己的分支，时不时地往dev分支上合并就可以了。

所以，团队合作的分支看起来就像这样：




## bug分支

- 简介：
  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。

## feature分支

- 简介：
  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。

## 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！

# 便签管理

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。

## 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
...
```

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
 - [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。
未完待续.....

### 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

这里推荐几个网址博文供大家学习使用

怎样使用github? （from 知乎）

Github GUI 基本操作教程

github release 功能的使用及问题解决

# 自定义Git

这里涉及不是常用的概念和功能，可以根据需求跳转到链接进行查看学习：

忽略特殊文件

配置别名

搭建git服务器

---

参考博文：
Git学习手册
Git教程

---

文档Github地址：https://github.com/ShixiangWang/Git-Hankbook