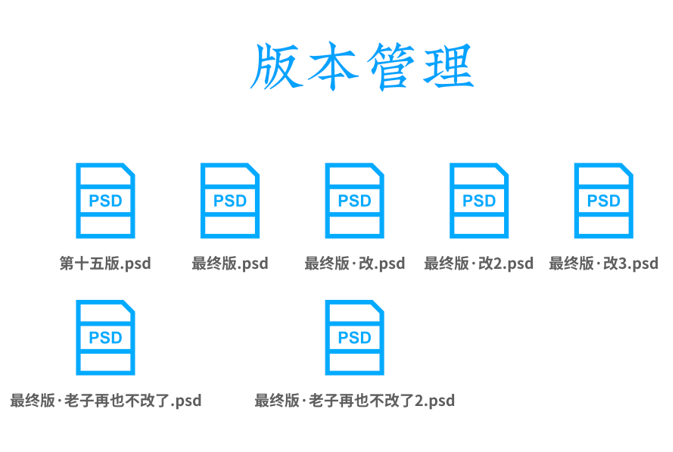


<https://lab.github.com/githubtraining>

# Git简介

Git是目前世界上最先进的分布式版本控制系统。

据说每个设计师都会有一个文件夹，里边放满了各种版本的「最终版」设计稿。



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文档变成了上面这样。

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，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。

这其实就是「人肉版」的「版本管理」。

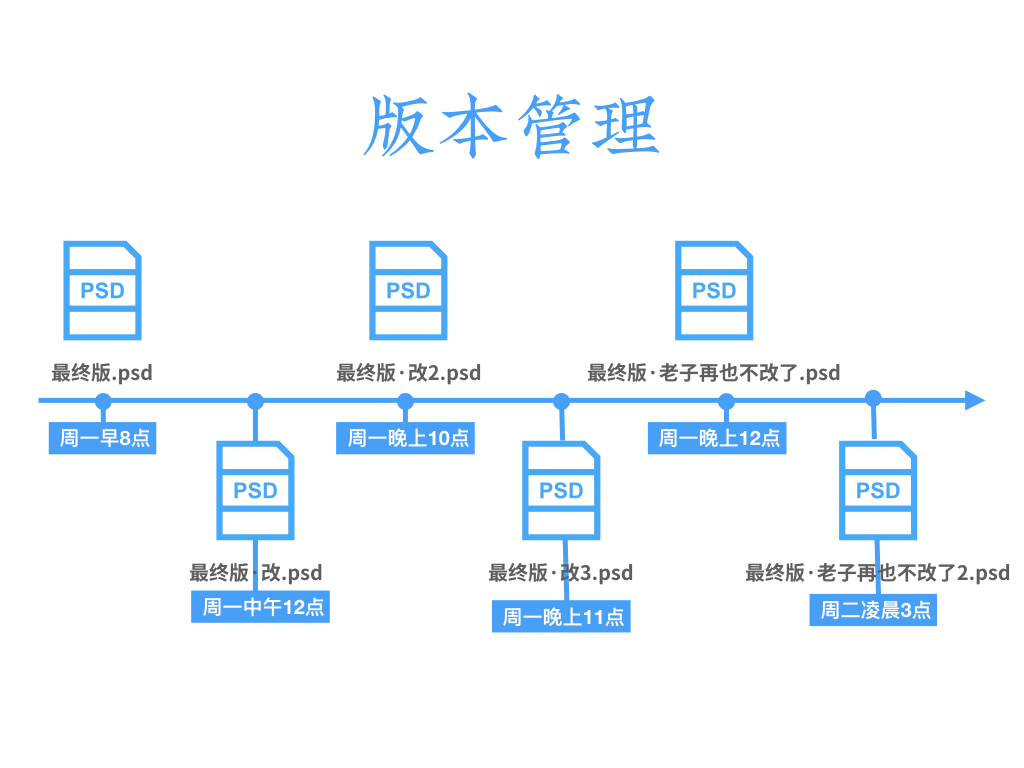
我们需要专门的软件来对代码做版本管理。

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。

版本管理最大的意义在于，它可以让我们随时「回溯」到之前某个时刻已经被覆盖过的版本去。



# Git的诞生

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。

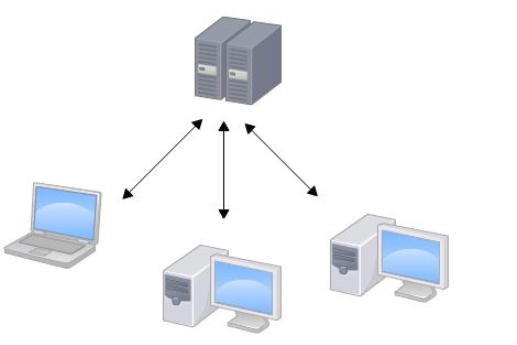
安定团结的大好局面在2005年就被打破了，原因是Linux社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发Samba的Andrew试图破解BitKeeper的协议（这么干的其实也不只他一个），被BitMover公司发现了（监控工作做得不错！），于是BitMover公司怒了，要收回Linux社区的免费使用权。

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。

# 集中式 vs 分布式

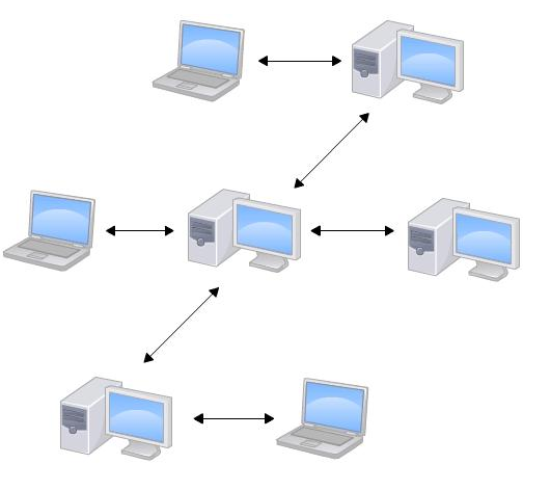
先说集中式版本控制系统，版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器就好比是一个图书馆，你要改一本书，必须先从图书馆借出来，然后回到家自己改，改完了，再放回图书馆



缺点：要求联网、服务器崩溃

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。



我们就针对「可回溯」这个特性，来想想应该如何设计一个「版本管理系统」。



要想能随时回到某个版本，我们肯定要将这些版本对应的数据存储起来才行，这里就有两种存储方案了：一种是全量存储，将每一个版本的数据都完整的保存下来，使用的时候，直接取用即可。



这个方案的优点是简单和快速。但缺点是存储了大量的重复内容，占用了磁盘空间。

另一种方案是增量存储。



这种方案存储一个基础版本，然后就只存储新的版本和它之间的补丁数据。需要使用的时候，再通过拼合来获得对应的版本。

它的优点是存储小了，因为大部分的版本都只有增量数据。缺点是使用时需要运算，降低了系统的性能、增加了系统的复杂性。

在了解了这两种方案以后，我们就需要根据场景来进行取舍。

对于编程的场景来讲，我们绝大部分时间都是在处理纯文本文件，这些文件本身很小，即使存储多份，也不会占太多的空间。 如果遇到特别大的文件，可以专门对其进行特殊处理。

相反，版本管理工具要被团队里每一个程序员使用，保持它的简单性可能比少占一点磁盘更重要。

正是因为Git采用了全量存储，所以它本质上就变成了一个「Content-addressable 的文件系统」。

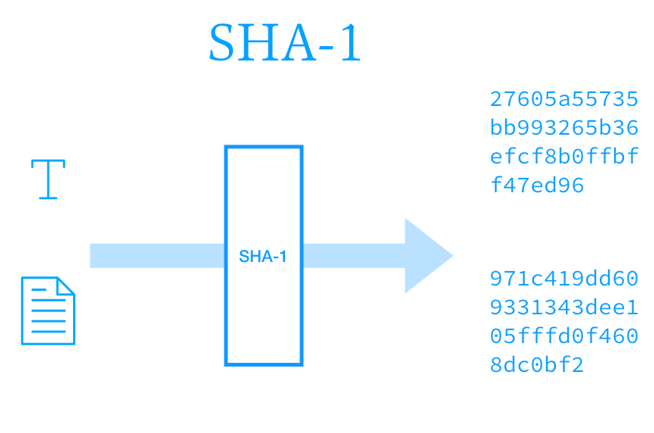
这是Git在设计上和其他那些版本管理系统最大的不同。它直接导致了其他系统的概念在Git这里是找不到对应的。这也是我们强调要回到需求层去理解Git的原因。

那么这些不同版本的文件存放在什么地方呢。

答案是，它们都在我们项目的根目录下的一个叫做.git的目录里边，这使得Git管理的项目很干净。当我们不想要版本管理数据时，直接把这个目录删除就好了。

再来说说版本库里边文件的命名。因为是在自己的版本库里边，所以这里的文件名不一定非要和原始文件一致，当然也不可能一致，因为我们要存放多份。

Git采用了SHA1函数对文件内容进行计算，然后将值作为文件名。SHA1函数和MD5函数类似，都是常用的哈希函数。

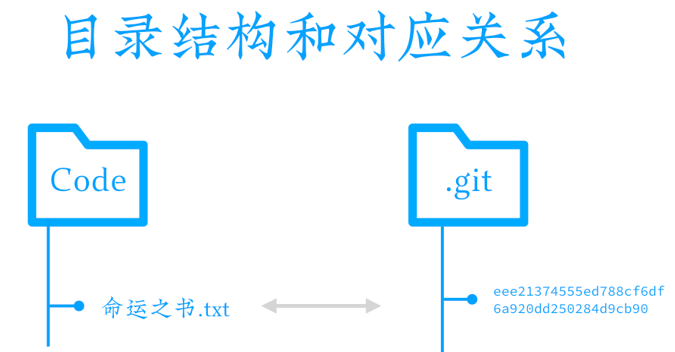


所谓哈希函数，是为数据创建「指纹」的方法。比如SHA1，它能将所有传入的内容都转换为一个40位长的数字串。当传入同一份数据时，每次会得到相同的输出；而传入不同的数据，则会得到不同的结果。

严格的讲，也会存在多份数据对应的SHA1值相同的情况，但由于概率非常小，所以我们一般简单认为它是唯一的。

而采用这种唯一哈希值作为文件名的好处就是，它保证了相同内容的文件在系统中天然就只会存在一份，不用再费心思去做排重。

然后我们需要有一种机制来将代码目录下的文件和版本库中的文件对应起来。

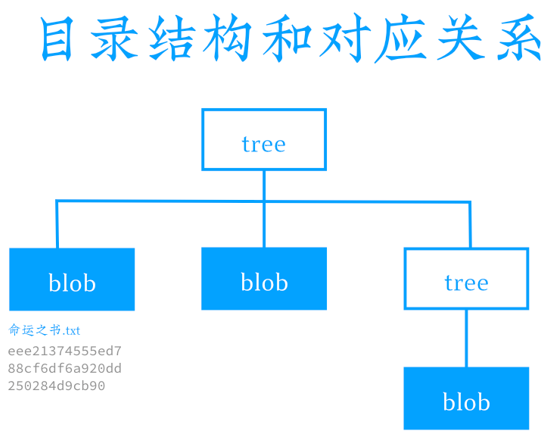


虽然当前文件的SHA值可以计算出来，但如果文件发生过修改，那我们只能计算出修改后的值，修改之前的SHA值我们是不知道的。这就需要额外的文件来描述了。

这只是文件的情况，而通常我们的代码目录下除了文件还有子目录。所以这个结构还必须支持目录。

# Git的核心设计

我们来看Git是怎么设计的。

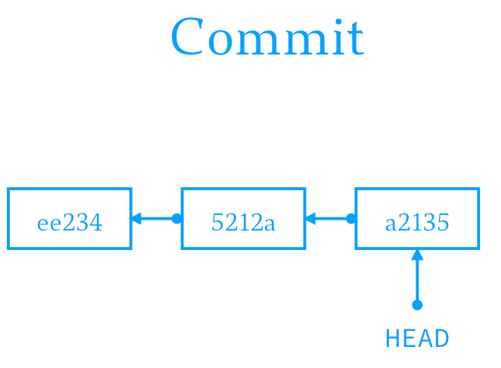


它采用了树状结构来描述这种关系。树结构里边只有两种节点，一种是叶子节点、里边直接是二进制内容。

另一种是子树节点，它表示下边还有其他的树或者叶子节点。

这种结构其实我们非常熟悉，你可以把「树节点」想成是文件夹；把「叶子节点」想成是文件。

树结构在Git中非常的常用。



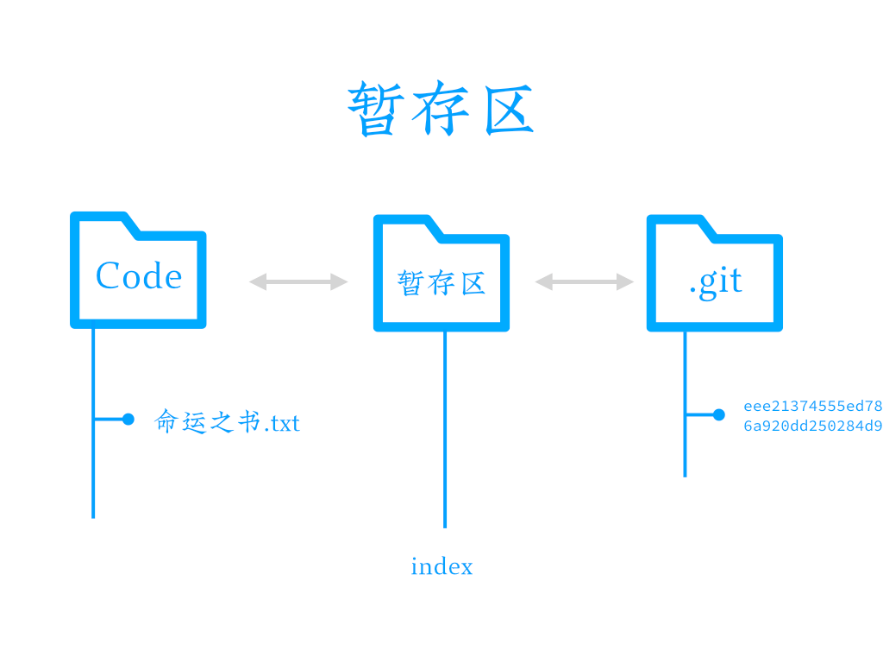
由于它同时包含有「目录结构」、「文件名称」和「它们对应的SHA值」，所以本质上来讲，它其实完整描述了一个目录或文件在某一个时间上的全部信息，如同一个「快照」一般。

在Git中我们一般通过Commit（提交）来创建这种「快照」。它除了会记录前边提到的、关于目录和文件的tree，还会记录提交人信息和上一次提交的SHA1。

这就将这些「提交」像链条般的串了起来，从而形成了完整的修改历史。而通过在「提交」之间的切换，我们就实现了复杂结构的目录在任何存储过的版本上的「回溯」。

这种切换，在Git中是由HEAD（头）文件的指向来实现的。

为了进一步提升使用上的便利性，Git在「代码目录」和「版本库」之间加入了「暂存区」。



假设我们要添加一个小功能，它需要修改10个文件。我们并不希望每修改一个文件就在「版本库」中进行一次「提交」。所以我们需要一个缓冲区域，我们可以将这些文件加入到里边，等修改完了，再一次性的放到「版本库」里边，并把缓冲清空，以便下次使用。

这个缓冲区域就是「暂存区」。

在实现上，「暂存区」其实并不是一个文件夹，只是一个名为index的文件。它甚至是放到.git目录下的。

「暂存区」还可以帮助Git识别文件状态，从而更好的生成「快照」。



通过对比「代码目录」和「暂存区」，我们可以知道一个文件是否是新增、删除或者被修改。

而通过对比「暂存区」和「版本库」，我们可以知道一个文件是否丢失、修改或者未被追踪。



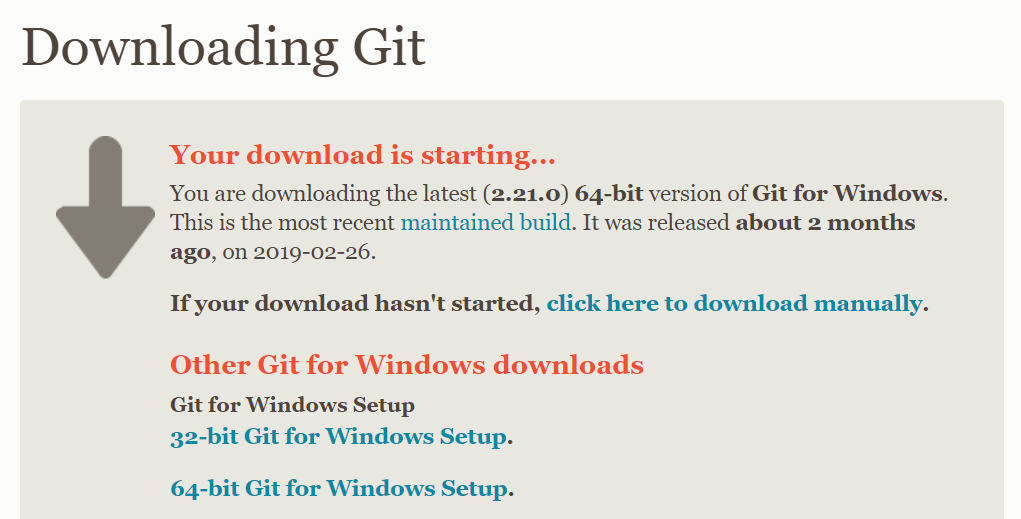
有些同学有时候理解不了Git的命令，主要就是由于无视了「暂存区」的存在而导致的。

到这里「回溯」的功能，已经被实现，我们可以很好的管理单独工作时的版本了。

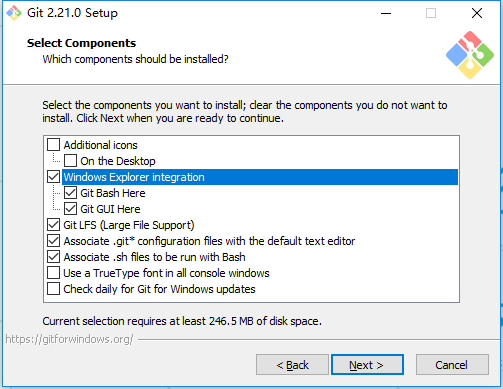
# git的安装

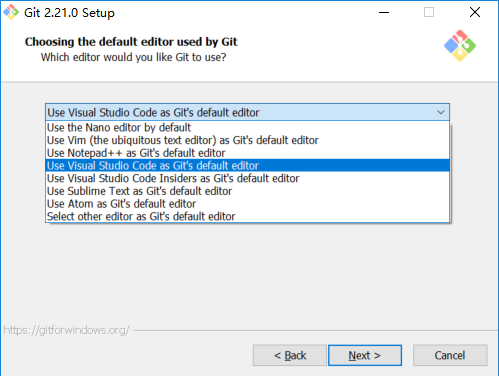
地址：

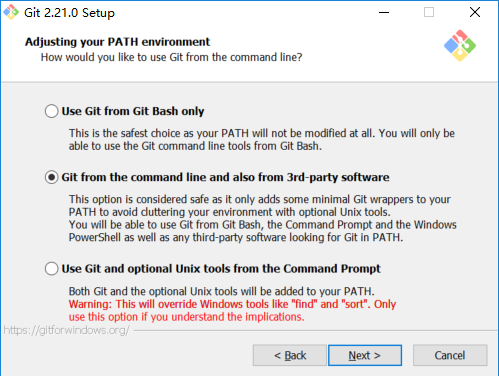
<https://git-scm.com/downloads>

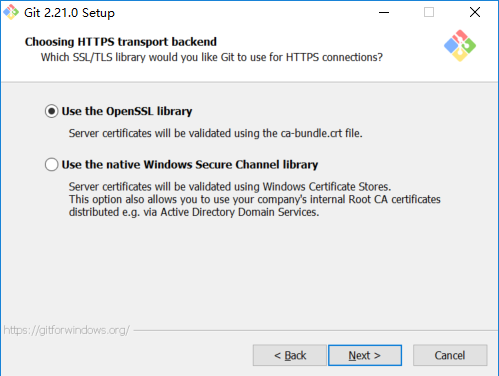


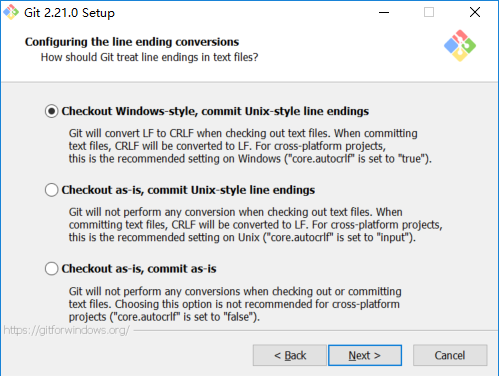
默认选项安装

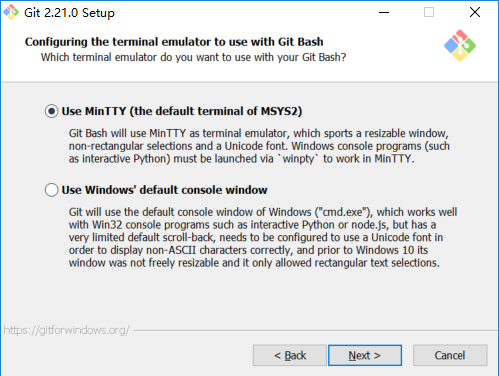


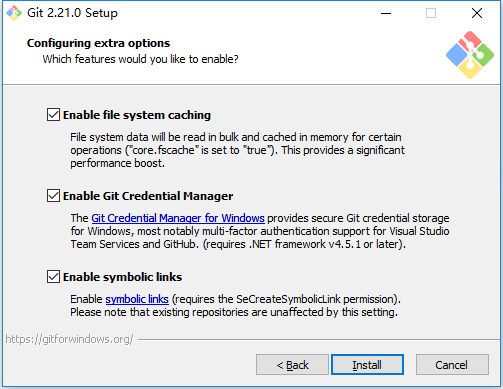


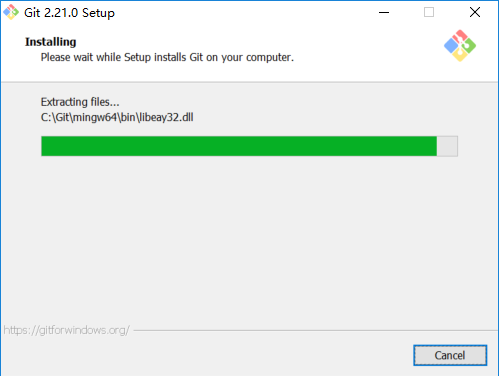








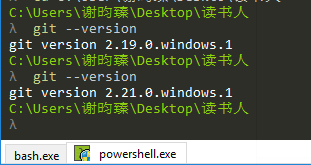




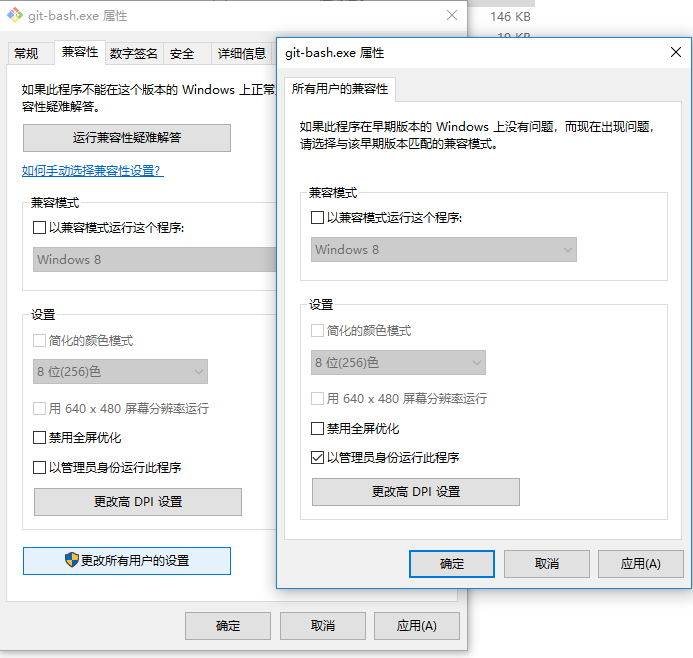
## 测试

右键，有git bash here 和 git GUI here选项则说明安装成功。

可以在命令行中输入git --version查看当前git版本



安装完成后，还需要一步设置：在git的安装路径将git-bash设置为以管理员身份允许。

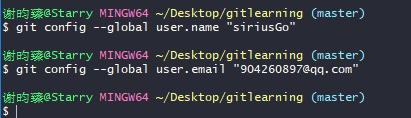


接下来要对本地的git的账号进行设置。

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。

# 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，右键，git bash here,在跳出的命令行中创建一个空目录（也可以手动创建）:

$ mkdir learngit

$ cd learngit

$ pwd



mkdir 创建文件夹

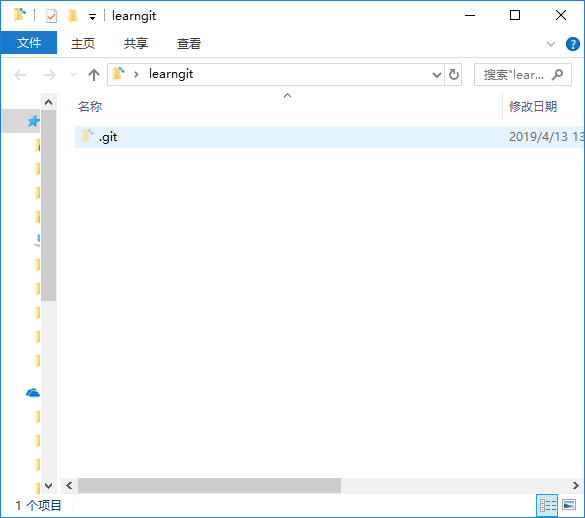
cd 转换目录

pwd命令用于显示当前目录。

Tab键可以自动补全文件名

第二步，通过git init命令把这个目录变成Git可以管理的仓库：





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。

如果没看到是因为生成的文件是隐藏文件夹，在文件属性里选择显示隐藏文件夹即可。

也不一定必须在空目录下创建Git仓库，选择一个已经有文件的目录也是可以的。

然后就可以把需要版本管理的文件放入这个文件夹，与.git同一个目录。

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。

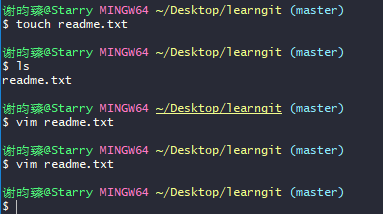
不幸的是，Microsoft的Word格式是二进制格式，因此，版本控制系统是没法跟踪Word文件的改动的，前面我们举的例子只是为了演示，如果要真正使用版本控制系统，就要以纯文本方式编写文件。

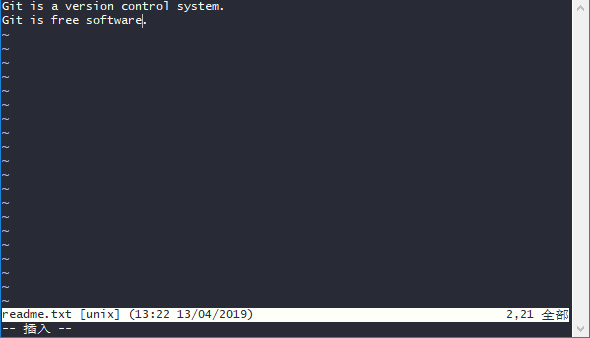
因为文本是有编码的，比如中文有常用的GBK编码，日文有Shift\_JIS编码，如果没有历史遗留问题，强烈建议使用标准的UTF-8编码，所有语言使用同一种编码，既没有冲突，又被所有平台所支持。

现在放了一个readme.txt文件进去 内容为：

Git is a version control system.

Git is free software.

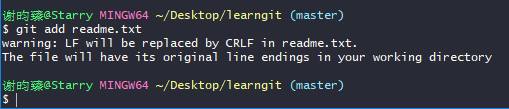




和把大象放到冰箱需要3步相比，把一个文件放到Git仓库只需要两步。

第一步，用命令git add告诉Git，把文件添加到git暂存区里：

$ git add readme.txt



（我们可以运行 git reset 命令来将暂存区里边的某个文件移出来。

$ git reset HEAD-- file 从暂存区移除某文件或某目录

$ git status 查看当前有.git文件的文件夹下的修改状态 ）

(.git文件夹下的内容

其中objects目录就是用来存放文件的全量历史数据的。

进入objects目录，我们可以发现下边有一个61目录，里边有一个以ac开头的文件。

这里将61和这个文件名拼接起来就可以得到我们刚才add的文件的SHA1。

把前两位拿出来作为目录名，这是为了避免在同一个目录里存放太多文件，导致文件索引过大而降低性能。

我们来确认下文件内容，通过git的cat-file命令，我们可以查看存储到objects目录下的所有对象。

而这个「index」文件就是我们之前提到的「暂存区」。

我们可以通过「git ls-files --stage」命令来查看它的内容。

可以看到，a.txt这个文件名和它当前对应的SHA1值都被记录在了暂存区中。

「HEAD」文件描述了当前分支上Head的位置。

通过「cat」命令我们可以查看到HEAD文件里边其实放的是一个引用地址。

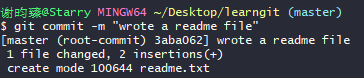
而这个引用指向的地方是不存在的。

这是因为，我们还没有进行过commit。

)

第二步，用命令git commit告诉Git，将暂存区的文件提交到本地仓库：

$ git commit -m "wrote a readme file"



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。

git commit命令执行成功后会告诉你，1 file changed：1个文件被改动（我们新添加的readme.txt文件）；2 insertions：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."

这个时候我们再来看一下HEAD里边的那个引用。

可以看到，现在这个引用就指向了一个提交。

而这个提交里边，正是我们刚才的内容。

有时候我们修改的文件比较多，可能会忘了哪些需要add。

这时候，我们可以通过git status命令来查看。

$ git status 查看本地文件的修改和暂存区情况

比如我们修改了b.txt，但是并没有add它，那么我们运行 git status 时就能看到上图的提示。▼

Untracked files:

(use “git ad <file>. . .” to include in what will be committed) . . .

千万不要使用Windows自带的**记事本**编辑任何文本文件。

# 添加远程库

## 注册github账号

<https://github.com/>

## ssh key 权限

先在bash里看看自己的身份的名字和邮箱

1 git config user.name

2 git config user.email

如果不是之前设置的就重新设置一遍

找到这个文件夹

C:\Users\(计算机用户名)\.ssh

3 git 输入命令

ssh-keygen

接着出现：

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/（计算机用户名）/.ssh/id\_rsa):

请直接按下回车，即将ssh文件下载在默认文件夹（/Users/（计算机用户名）/.ssh/）中

再回车 将密码设置成空白 以便之后登陆不需要再输入密码

再回车，确认密码

然后系统会自动在.ssh文件夹下生成两个文件，id\_rsa和id\_rsa.pub，用记事本打开id\_rsa.pub

（id\_rsa.pub 为公钥 另一个是私钥）

将全部的内容复制

打开https://github.com/，登陆你的账户，进入设置

进入ssh设置

New SSH key

把刚才复制的粘贴进去

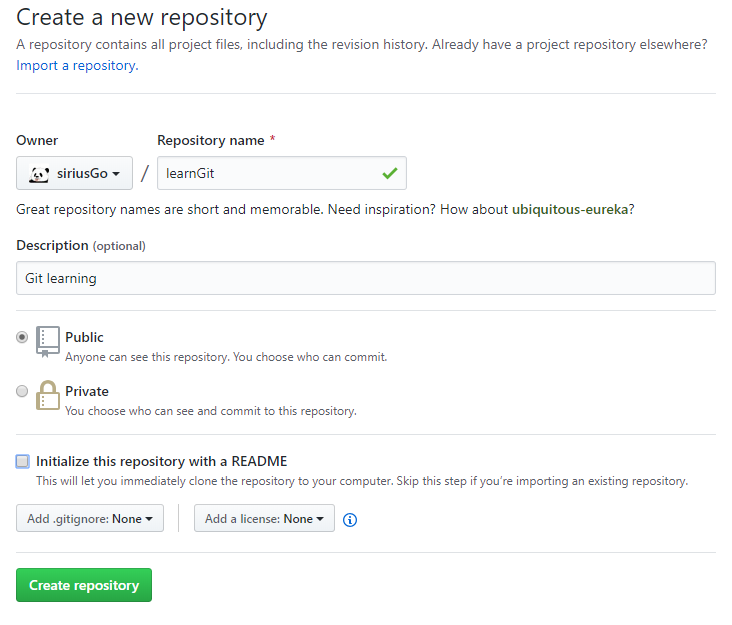
点击 add ssh key

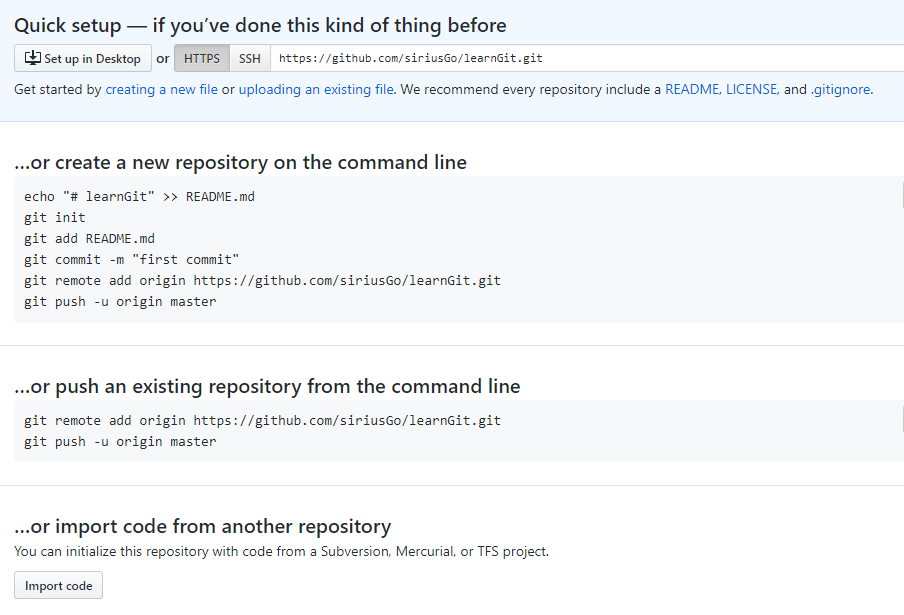
回车

提示成功

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关联、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：



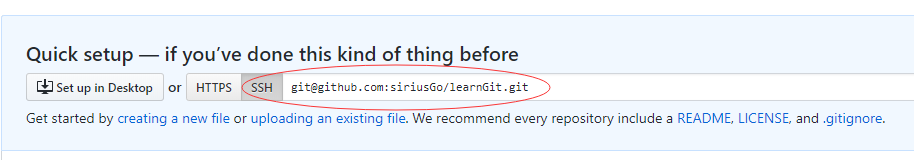


目前，在GitHub上的这个learngit仓库还是空的，GitHub告诉我们，可以从这个仓库克隆出新的仓库，也可以把一个已有的本地仓库与之关联，然后，把本地仓库的内容推送到GitHub仓库。

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:siriusGo/learnGit.git

这里 origin 后面的地址在这里



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

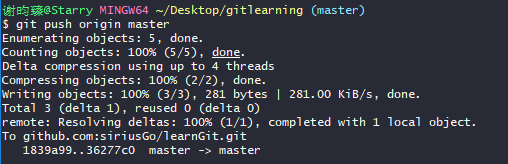
把本地库的内容推送到远程，用git push命令，实际上是把当前分支master推送到远程。

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。

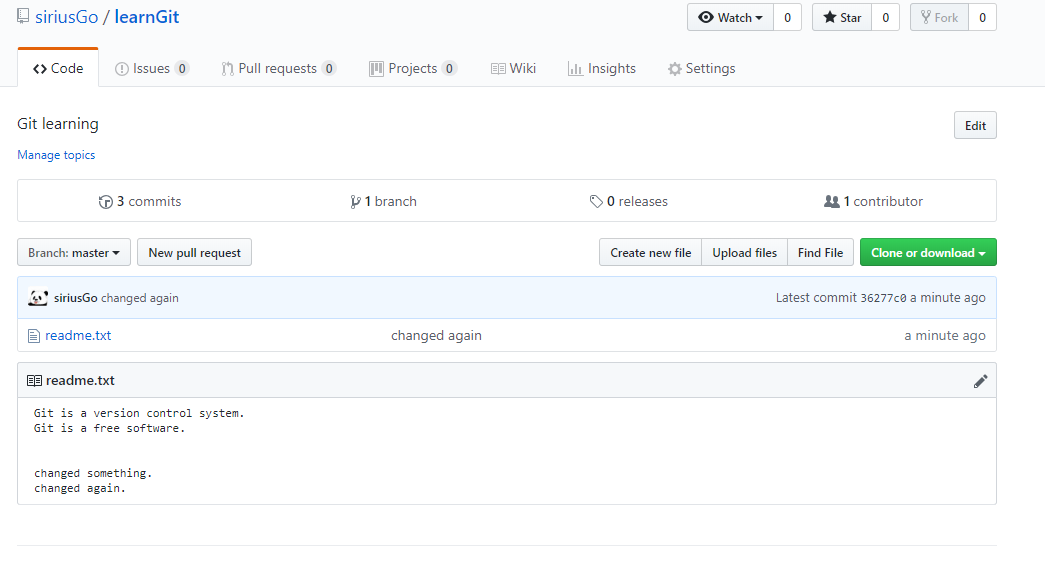
推送成功后，可以立刻在GitHub页面中看到远程库的内容已经和本地一模一样：

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！



那这个本地仓库是否和服务器端的仓库关联呢，我们来看看。

通过 git remote 命令，我们可以查看到所有关联到当前仓库的远程仓库。

$ git remote 远程仓库列表

orgin

可以看到，本地仓库和一个叫做「origin」的远程仓库关联。

而通过 git remote show 命令，我们可以查看某一个远程仓库的详细信息。

$ git remote show name 显示远程仓库信息

可以看到，这些信息包括了Fetch URL 和 Push URL。

通过 git push 和 git pull 命令，我们可以从服务器端推送和拉取代码。

### 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；

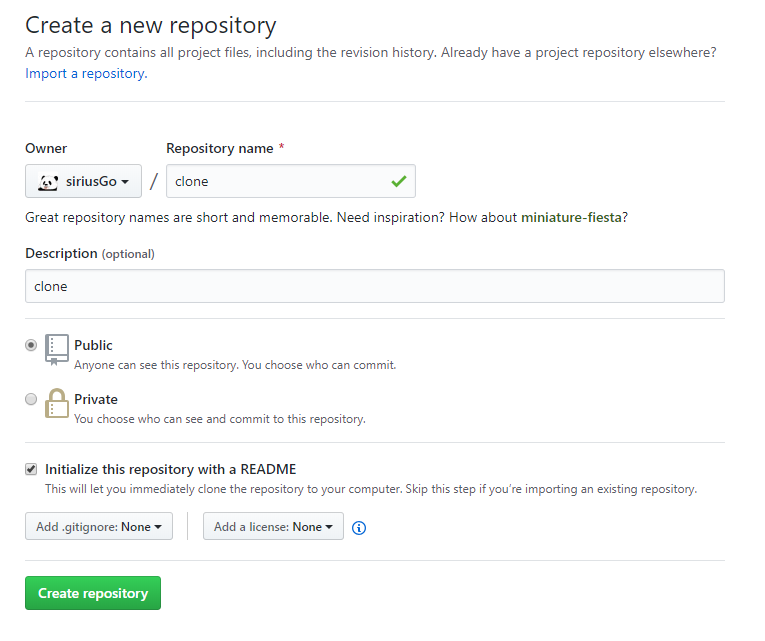
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

# 从远程库克隆

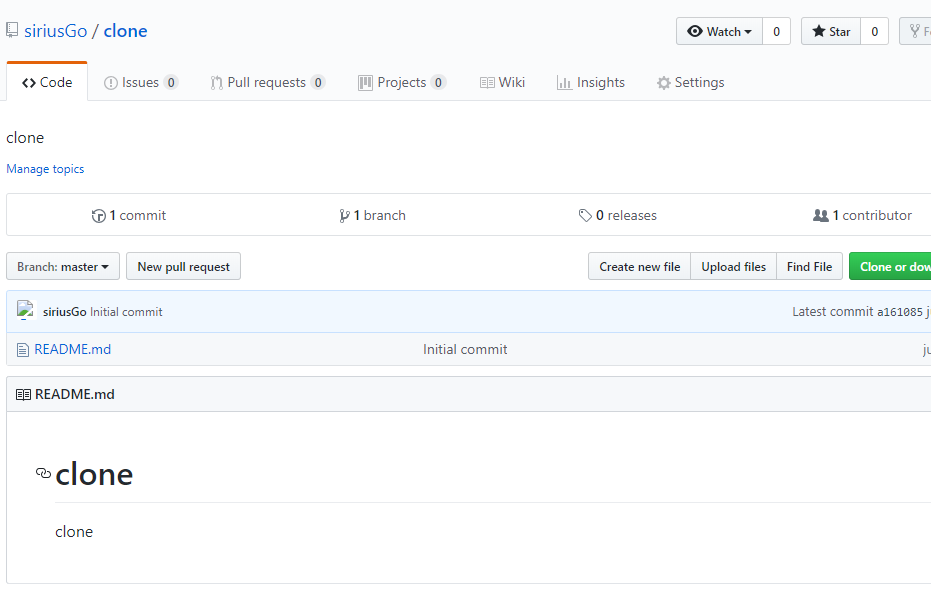
上次我们讲了先有本地库，后有远程库的时候，如何关联远程库。

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。

创建一个新的仓库

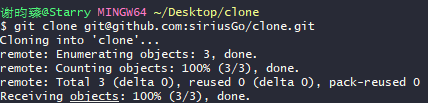


我们勾选Initialize this repository with a README，这样GitHub会自动为我们创建一个README.md文件。创建完毕后，可以看到README.md文件：



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，在你想要克隆的目录下创建文件夹并打开bash输入：

$ git clone git@github.com:siriusGo/clone.git



注意把Git库的地址换成你自己的，然后进入目录看看，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，还可以用仓库地址+.git <https://github.com/siriusGo/clone>.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

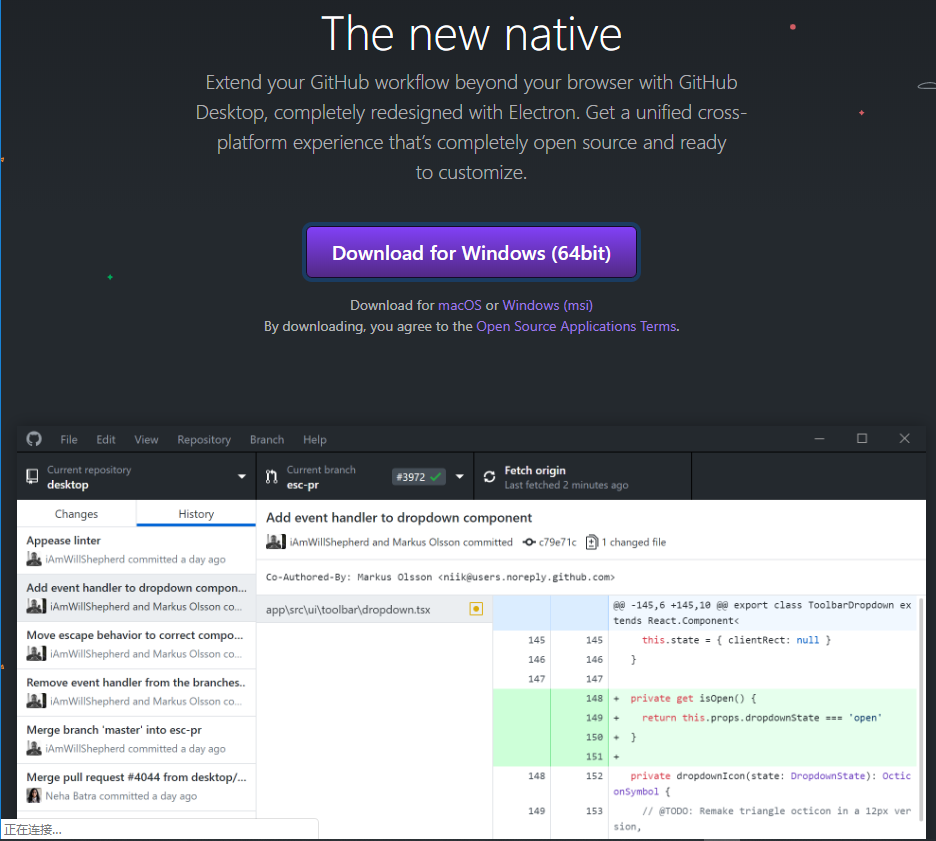
要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆。

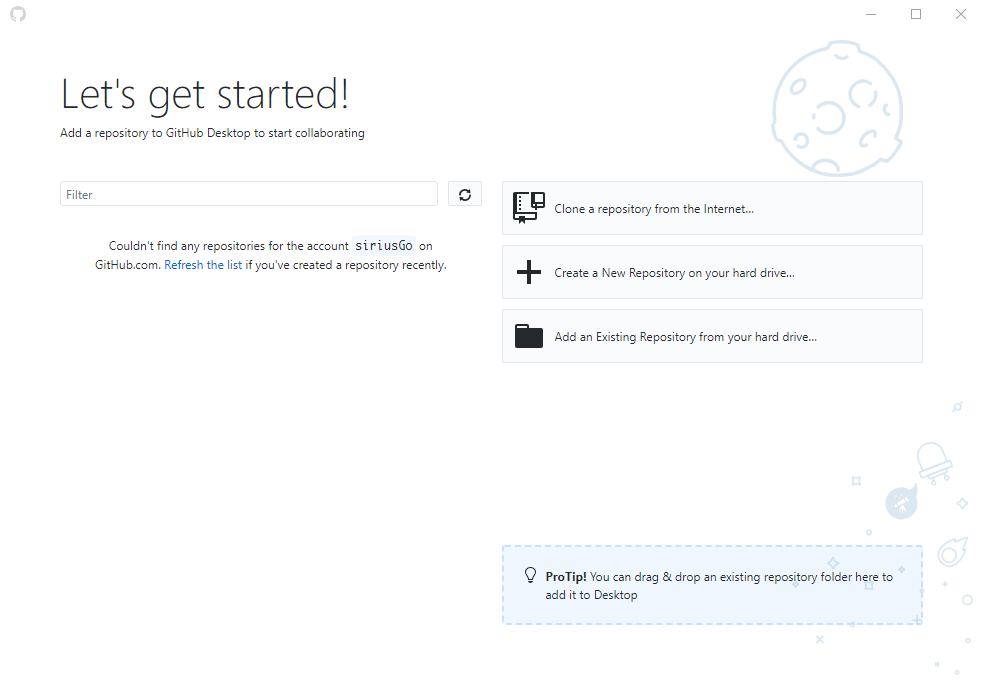
Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快。

# GitHub Desktop

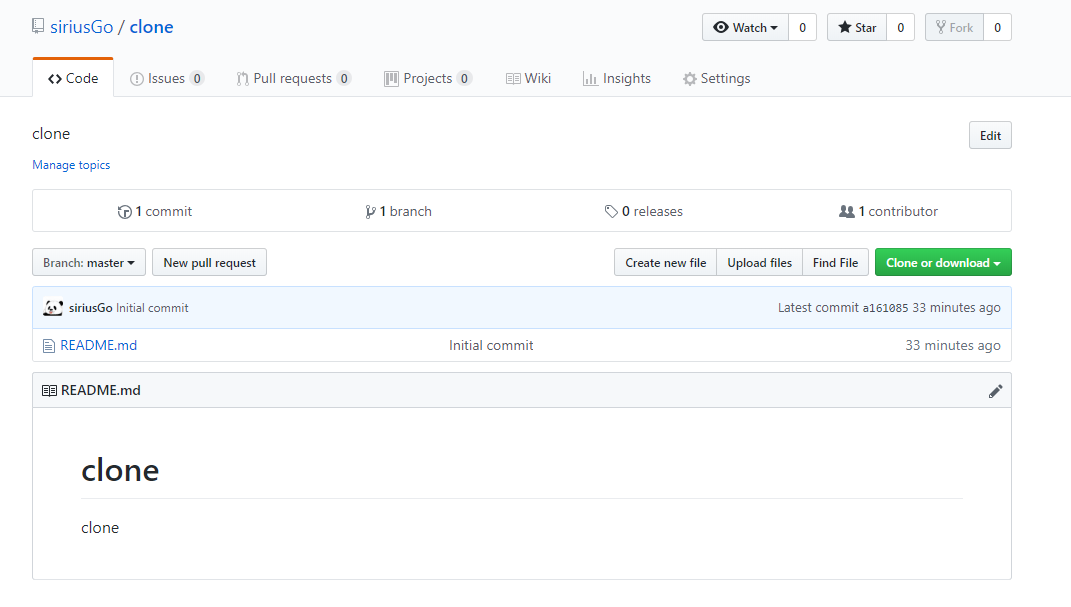
另外可以使用Github的桌面GUI来进行版本管理

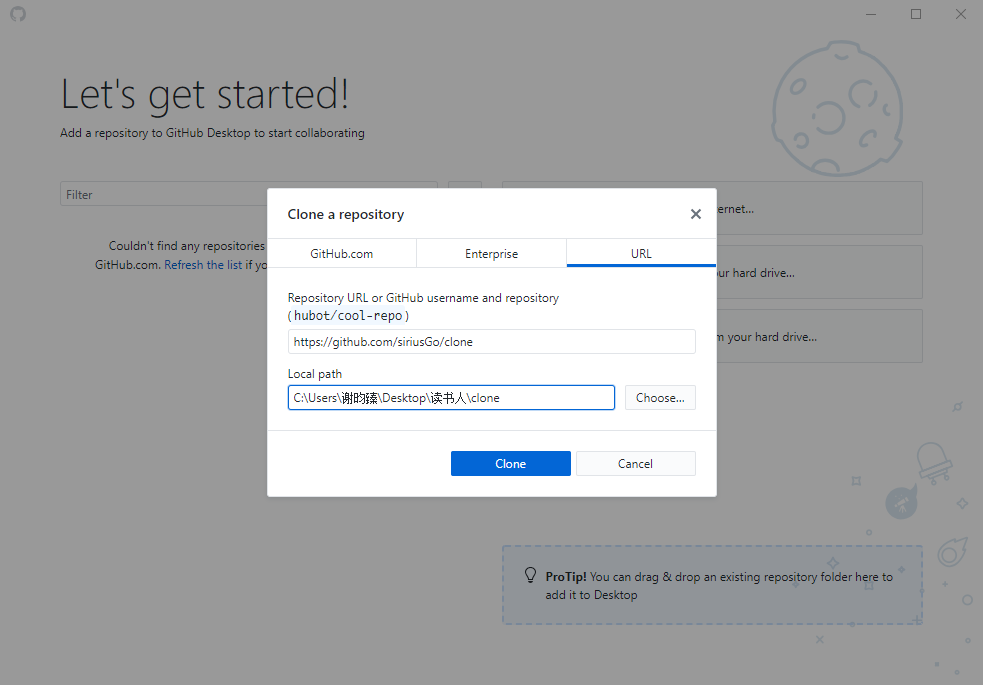
Github destop下载地址<https://desktop.github.com/>



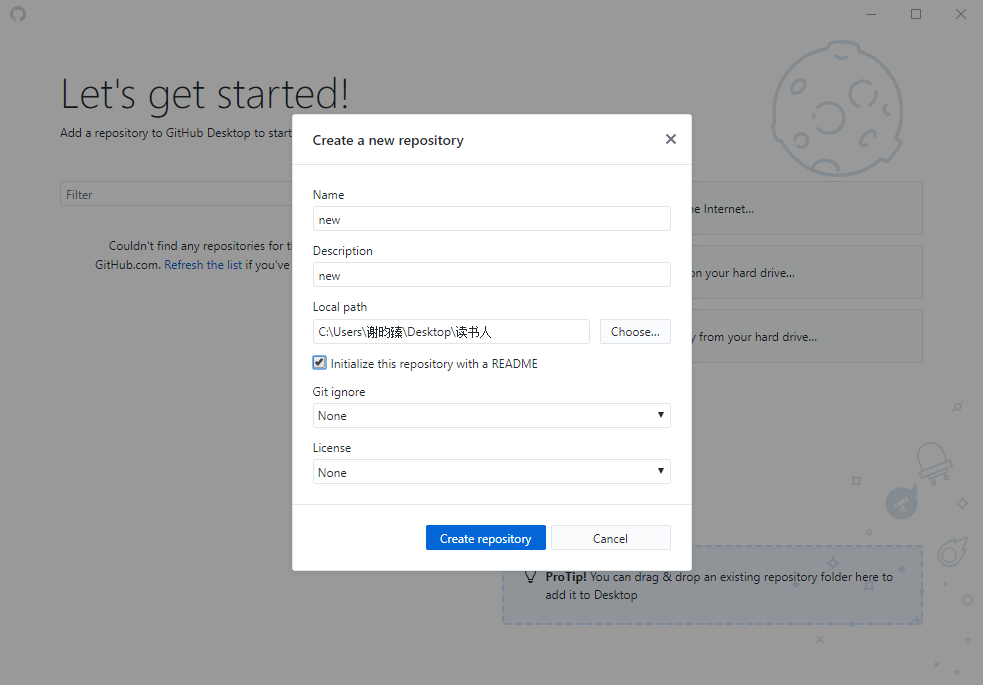


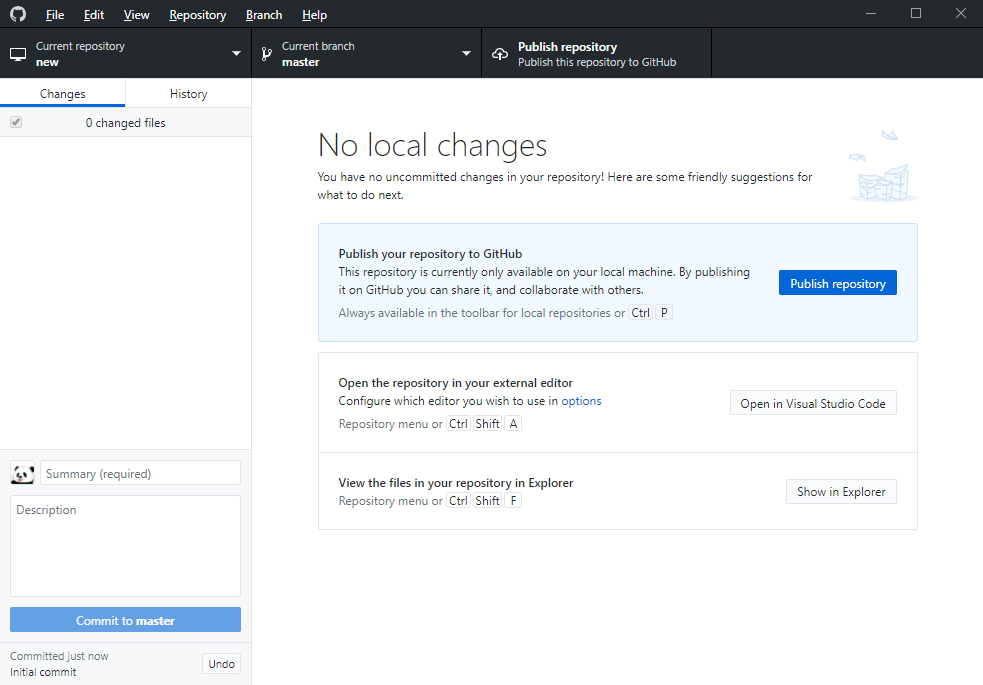
1. 从github.com上克隆下一个仓库

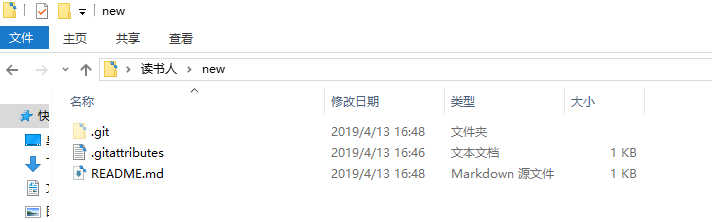




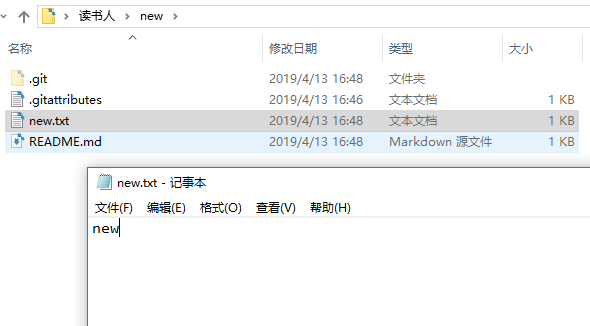
1. 在本地创建一个仓库

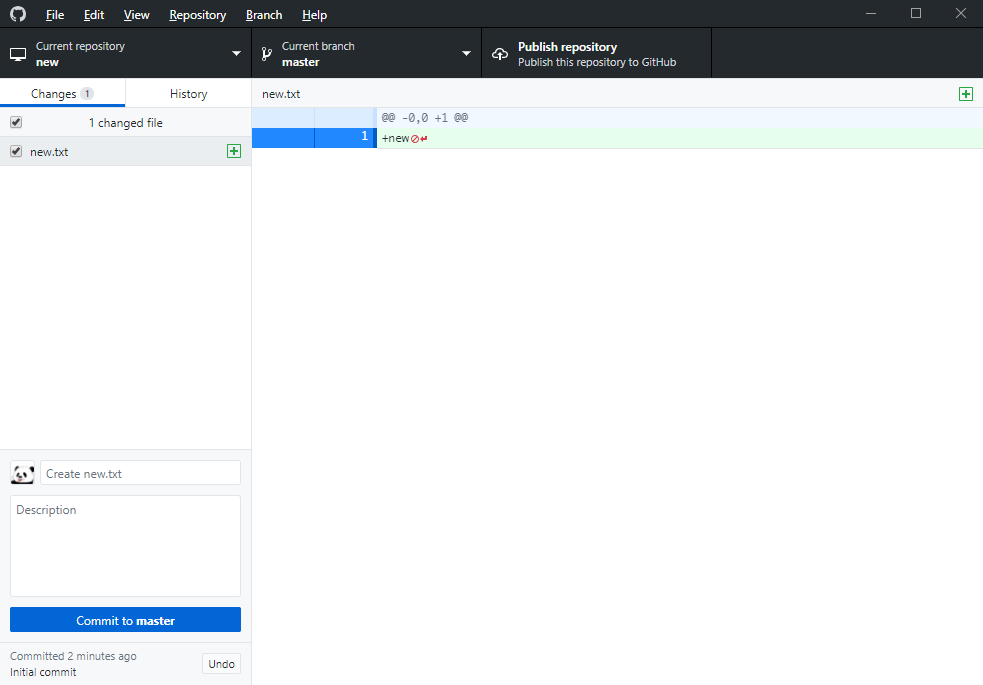


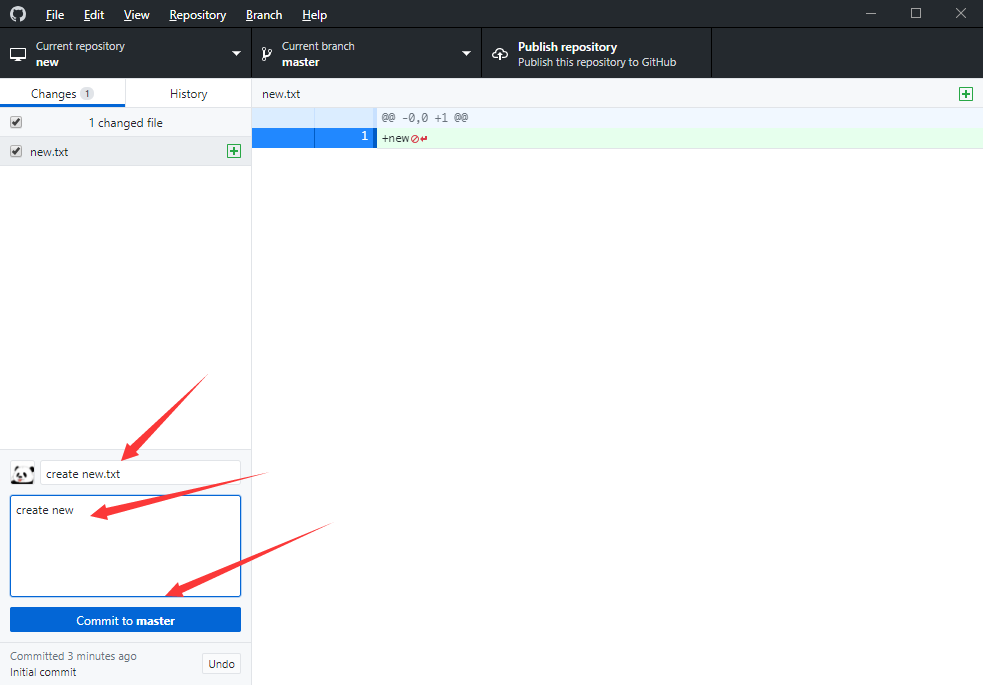


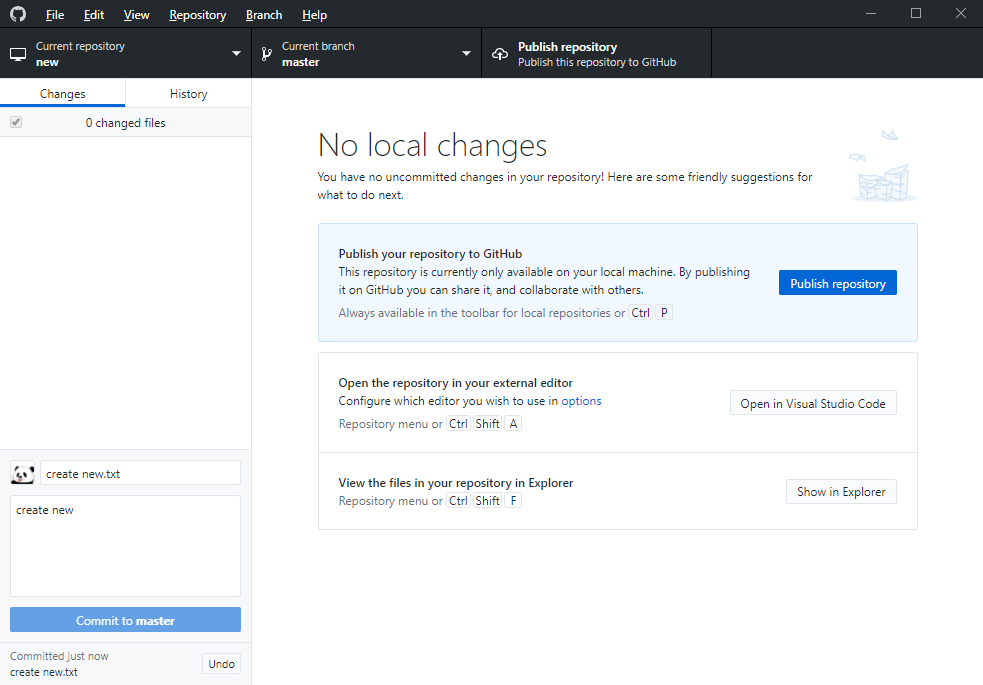


添加一个文件new.txt



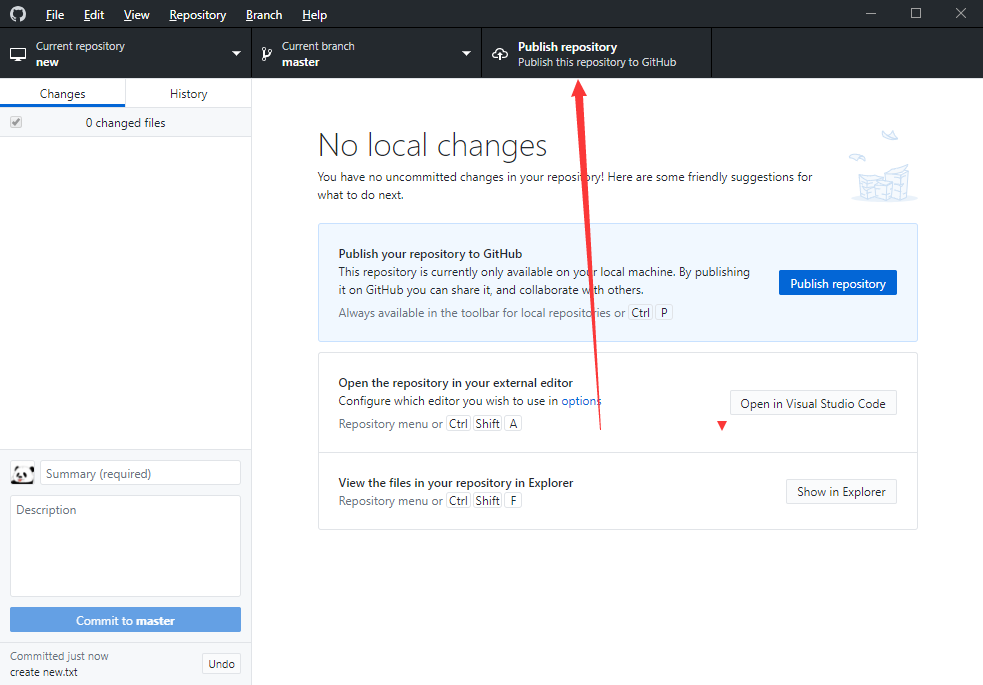






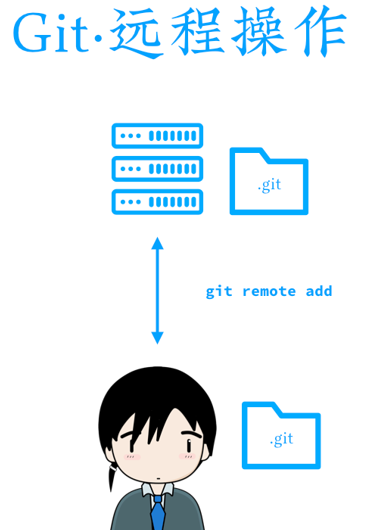
提交成功

提交到远程仓库



1. 添加一个本地的仓库

还有一种经常遇到的情况是，我们在本地已经有代码仓库了，想要把它发布到github之类的平台上。



我们会先在github上创建一个空的项目，得到远程仓库地址。

然后把它和本地仓库关联起来。

这时候我们会用到 git remote add 命令。

$ git remote add

关联远程仓库

这个命令接收两个参数。

一个是别名，另一个是远程仓库URL。

我们用git remote remove 把origin给删掉，然后将其重新添加为origin2。

$ git remote remove origin

$ git remote add origin2 <http://github.com/easychen/>...

然后我们通过 git pull 将远程仓库的代码拉取到本地。

$ git pull name branch 拉取远程仓库到本地

pull 命令接收两个参数，一个是我们remote add 时指定的name，一个是远程仓库的branch。

于是我们输入 git pull origin2 master ，可以看到，本地的代码是最新的。

$ git pull origin2 master

当我们在本地完成修改后，可以通过git push 命令来推送到远程仓库。

$ git push name branch 推送本地变更到远程仓库

注意push是写入操作，需要对远程仓库有写入权限，这里我们就不演示了。

最后补充一个课程发布后在微博上呼声最高的、没有讲到的命令，那就是git stash。

$ git stash 暂存当前修改到储藏区

这个命令可以将你当前进行到一半的工作保存到一个暂存区域，然后将当前目录回滚到上一次提交。

下边来看一个具体例子，我们往a.txt中写入一串字符，然后提交它。

$ echo “G000000000d” > a.txt

$ git add a.txt

$ git commit -m ‘init’

$ echo “BAAAAAD” > a.txt

然后我们好像把a.txt改坏了。

这时候我们当然可以checkout 前一个commit，但这样当前这个可能是坏的的修改就没有了。

这时候我们运行git stash，就会自动切回到上一个提交。

$ git stash

可以看到a.txt的内容已经恢复了。

但突然我们又觉得之前那个修改挺好的，想把它找回来。

这时候我们运行git stash apply，git就会自动把之前放到储藏区的最新的那个修改切回来。

$ git stash apply

运行git stash list，可以把所有放到储藏区的修改都列出来。

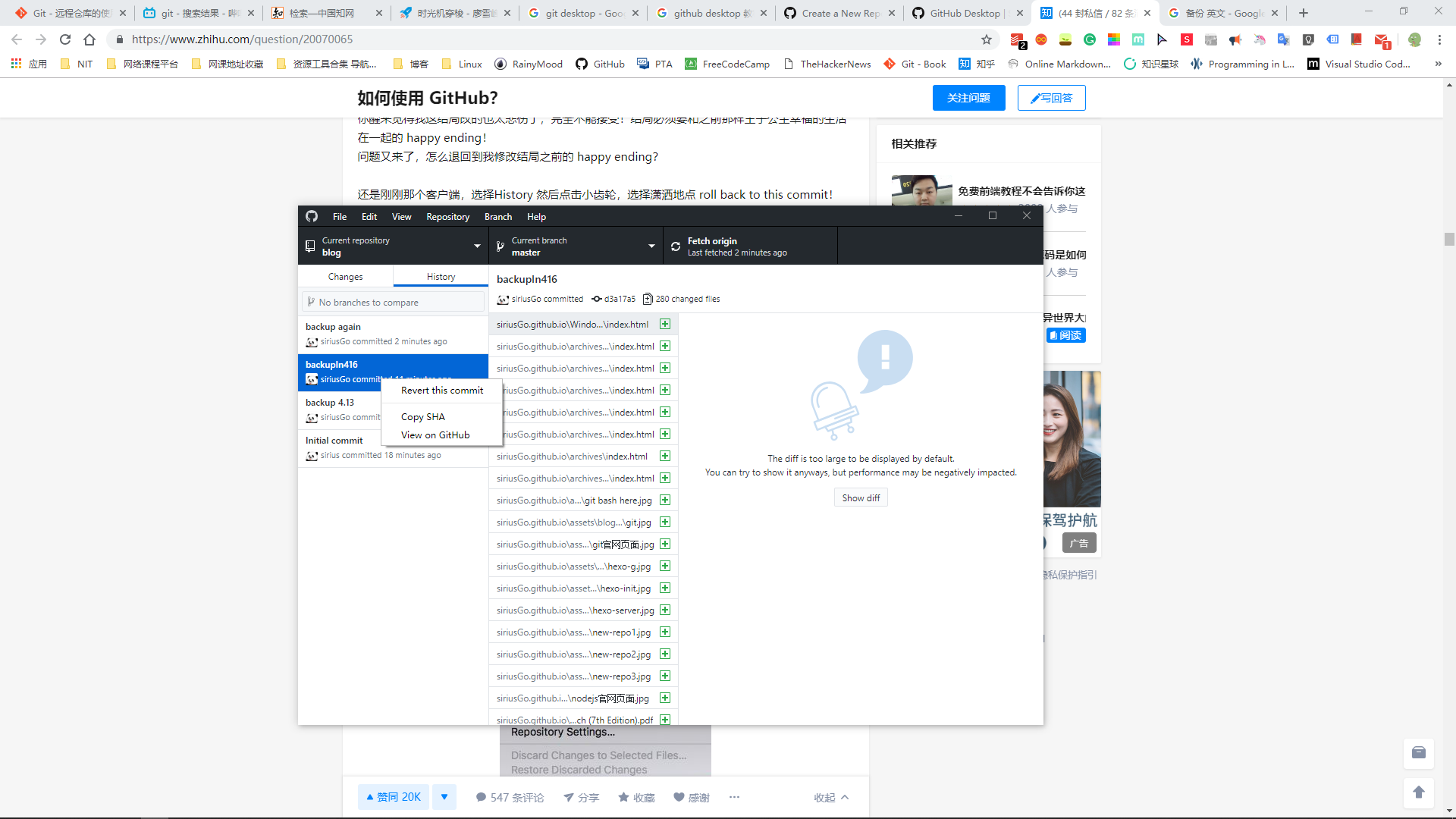
$ git stash list

stash@{0} WIP on master : ………

注意这里有个编号，通过指定编号，我们就可以对其中一个修改进行显示（show）、删除（drop）。

也可以用clear命令将储藏区全部清除。

# 回退



Revert this commit

\

Master 是主分支的意思

# 更新你的本地仓库为 最新改动 执行

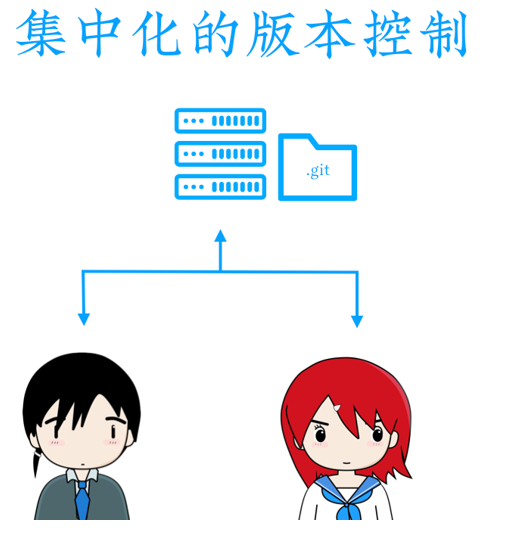
Git pull 拉取

# 分支

但绝大部分的工作，光靠一个人是做不完的。



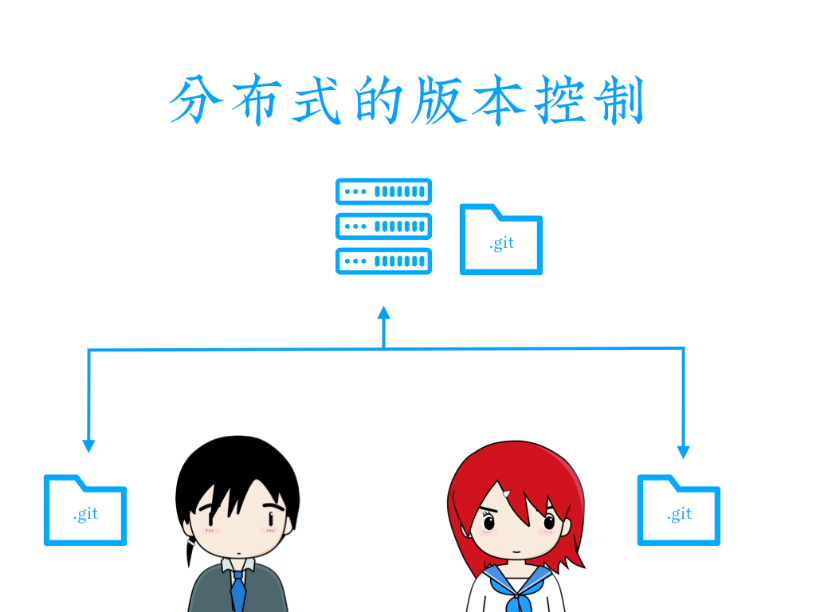
当有多个人参与的时候，就会出现很多文件被不同的人同时修改的问题。这种情况下，我们就需要维护一份公有代码库了。集中化的版本控制正是按照这个思路来做的。



它将版本库给挪到一台共用的版本管理服务器上去。每个人开始工作之前，先从服务器去获取最新的代码，进行自己需要的修改，然后再将修改后的代码提交回去。这时候服务器会检测提交的代码是否有冲突，如果其他人先提交了修改，它就会尝试自动合并冲突。

但这个合并不太智能，大部分情况下都搞不定。所以它会把两个版本的修改都写入，然后要求提交人手工处理冲突。

而Git采用的是分布式的版本控制。



虽然也放了一个版本库在服务器端，但它依然保留了本地的版本库。这样在没有网络的情况下，我们仍然可以通过本地版本库来工作。等到有了网络以后，再来处理本地版本库和服务器端版本库的同步问题。

具体而言，我们会通过「push」将本地版本库的代码推送到服务器；通过「pull」将服务器端的代码拉取到本地。现在我们已经可以处理多人一起开发一个功能的情况了。



但我们要面对的现实要更为复杂一些。比如说，一个产品下边可能会有多个工程师，每个人负责不同的功能分支，而他们之间的功能又可能有依赖。这就要求我们为不同的功能记录不同的修改历史，最后来决定是否将这些修改应用到之前的代码上。

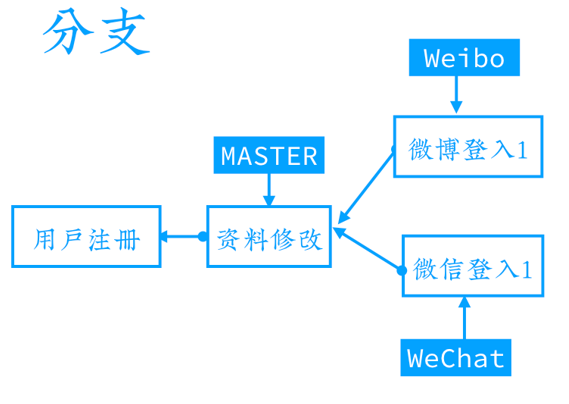
Git采用「分支」的概念来支持多种修改历史。

其实在我们初始化Git的时候，它就会为我们创建一个默认分支——「master」。

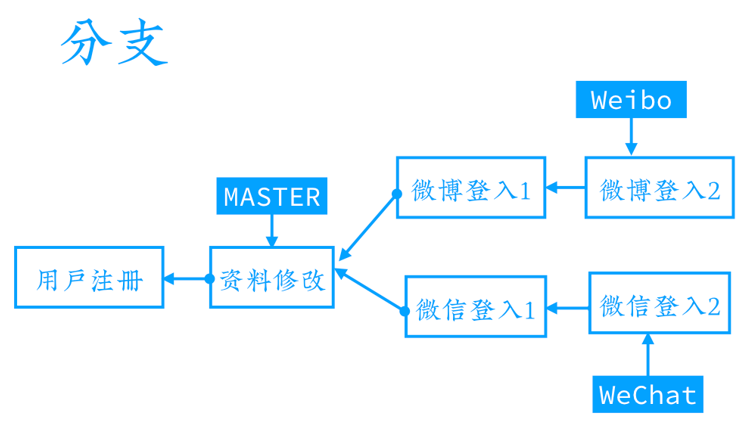
我们来看一个例子。假设我们在开发一个网站，现在已经完成了用户注册和资料修改功能。这时候，我们工作在默认的Master分支下。接下来，我们分别交给两位同学去开发登入功能，一位开发微博登入，另一位开发微信登入。

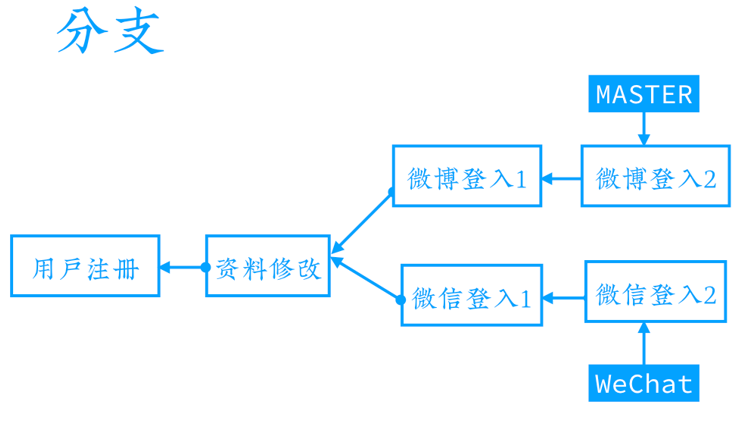
这时候他们就会分别创建一个分支。分支本身很简单，只是一个指向提交的指针。

因为这时候他们都还没有新的提交，所以这两个分支的指针和Master都是指到同一个地方的。然后他们分别在自己的分支上进行开发，产生了新的提交。



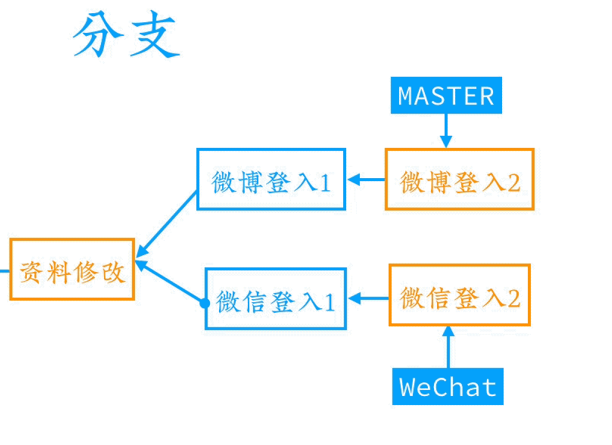
等功能开发完成、并且测试没问题后，就该把在分支里的修改「合并」回主干了。

我们先来看微博功能的合并。由于Weibo分支是直接在Master下继续开发的，所以这里的合并非常简单。就是简单的将Master指向Weibo即可。一般在完成功能后，我们会删掉这分支。



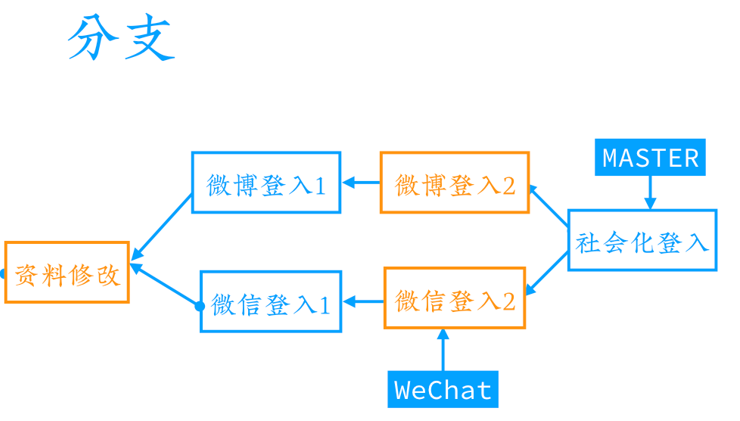
这里微博登入的功能已经合并了，所以我们会删掉Weibo分支。于是就变成这个样子。

接下来，就是要处理WeChat分支和Master分支的合并了。注意WeChat分支已经不再是Master分支的后续修改了。这里我们有两个合并方式，一种是merge合并。

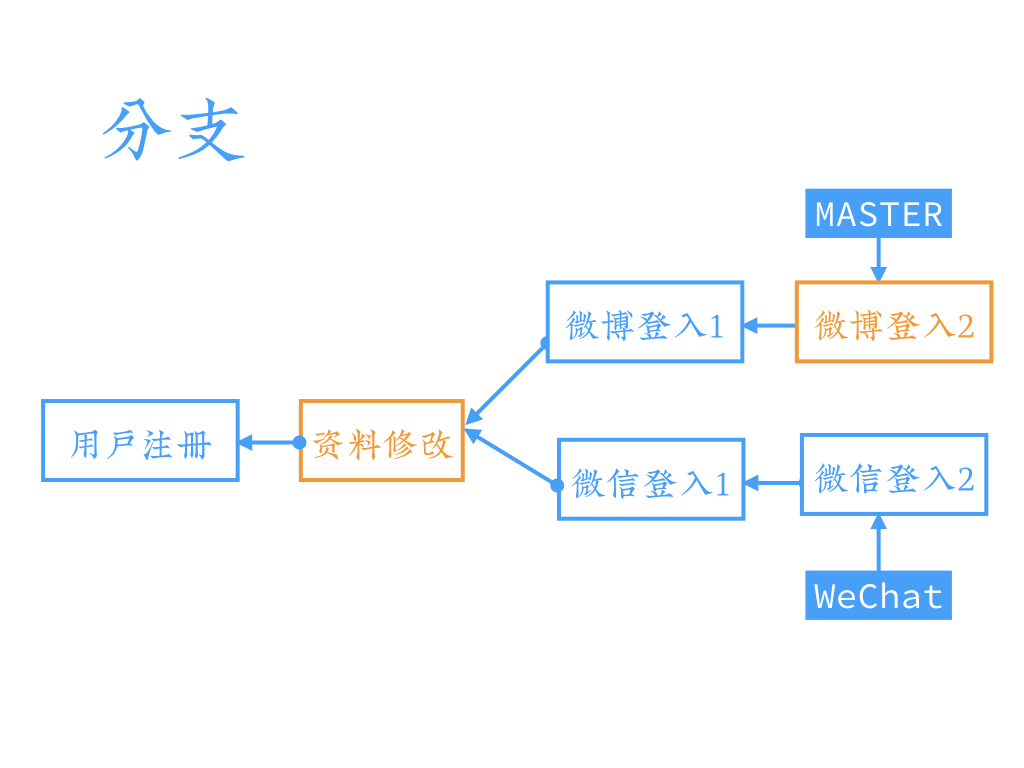


在这个命令下，Git会找到Master分支和Wechat分支的共同祖先——也就是「资料修改」，将这三者进行一个三方合并。

这种三方合并会产生一个新的提交，Master会指向这个新的提交。这样我们的分支功能合并就完成了。



而另一种合并方式则不采用三方合并，它会比较「微博登入2」和「资料修改」这两个提交，将其中的修改提取成一个补丁。

然后在「微信登入2」的提交上应用这个补丁，从而生成一个新的「提交」。

这个「提交」也同时包含了微博登入和微信登入的功能，和我们之前的merge有同样的效果。

这种方式被叫做「rebase」，中文叫「变基」。

这样，从版本管理需求一路下来，不知不觉中我们已经讲完了Git的核心设计。

下边我们来看看分支命令。▼

首先我们从master里边分支出dev分支和ops分支，分别为它们添加两句不同的话。▼

Make git great(master 分支)

Make dev great again (dev分支)

Make ops great again (ops 分支)

最后我们再把这两个分支合并到主干上。▼

首先我们通过 git branch 来创建新分支。▼

$ git branch name 创建新分支

$ git branch dev && git checkout dev

↑ 此处代码可在右侧「代码」标签页中查看和复制 ▼

然后再通过 git checkout 来切换到这个新分支上。▼

Switched to branch ‘dev’

切换过来以后，我们就可以进行编辑了。▼

这里我使用vim给a.txt原文之上加上了「make dev great again」。▼

如果你不会用vim，可以点击「Applications」菜单下的「Development」、「Visual Studio Code」，通过VSCode打开 /headless/Code/a.txt 来进行编辑。▼

编辑完成以后，我们先通过add将其加入暂存区，再通过commit放入仓库。▼

$ git add a.txt

$ git commit -m “dev”

然后，我们checkout回master。▼

$ git checkout master

再按之前的操作，创建一个ops分支，并对a.txt进行修改。▼

$ git branch ops

$ git checkout ops

$ vim a.txt

$ cat a.txt // cat是显示文件的内容

$ git add a.txt

$ git commit -m ‘ops’

现在我们两个分支就准备好了。▼

接下来我们就可以来合并分支了。▼

合并分支是通过 git merge 来完成的。▼

$ git merge branch 合并某分支

先checkout回master，然后运行merge命令。▼

$ git merge dev

可以看到，这次合并很成功，它是一次「快速合并」，它在我们的a.txt中添加了一行。▼

再次运行merge命令，将ops分支合并进来。▼

git merge ops  
↑ 此处代码可在右侧「代码」标签页中查看和复制 ▼

Auto-merging a.txt

Merge made by the ‘recursive’ strategy

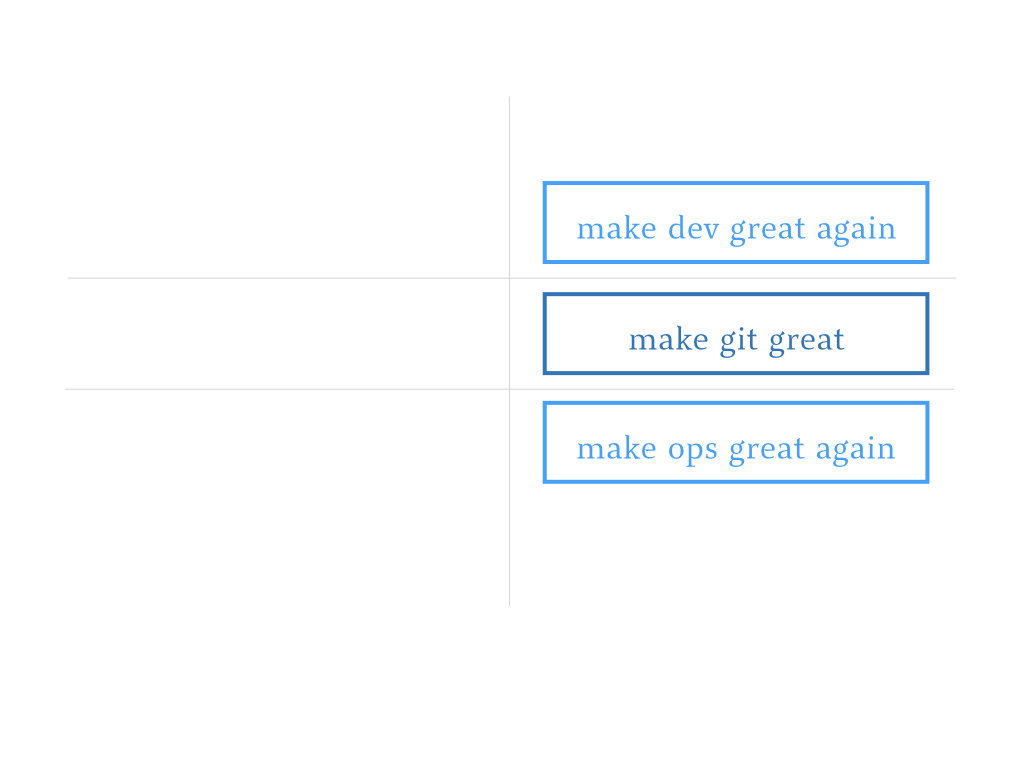
而且很幸运的，a.txt完成了自动合并。▼

通过cat命令查看a.txt的内容，可以看到dev和ops分支里边添加的内容都被加进来了。▼

之所以能自动合并，这是因为a.txt在dev分支和ops分支的新增内容分别在「make git great」的上边和下边。▼



通过简单的算法就可以进行合并。

那如果a.txt在dev和ops分支的新增内容，都在「make git great」之下会怎么样呢。

我们来合并试试。

可以看到，这里自动合并就失败了。

Auto-merging a.txt

CONFLICT(content):Merge conflict in a.txt

Automatic merge failed; fix conflicts and then commit the result.

这时候我们有两个选择。

一是放弃合并，这时候我们运行「 git merge --abort 」，就可以恢复到merge之前的状态。

另一个选择就是我们手工来处理冲突。

通过vim打开a.txt，可以看到在有冲突的第二句处，Git已经帮我们把两个版本的内容都写入进来了。

箭头两边分别标注了分支，中间用一排等号隔开。

我们把这些标记删除，把两个分支的第二句合并成「make devops great again」，然后保存。

这样我们就手工解决了冲突。

接下来，我们git add 刚才解决冲突的a.txt，然后进行commit提交。

$ git commit -m ‘fixed’

查看日志可以看到，刚才的merge提交已经完成了。

这就是我们手工解决冲突的过程。

刚才我们是通过git log 命令来查看日志的，它可以查到每次提交的记录。

$ git log 显示提交记录

在commit字样后边，有这次提交的SHA1值。

运行 git checkout SHA1值，我们就可以回到那次提交时的历史。

你也可以在checkout 后边加上 -b 参数来将其检出为一个全新的分支。

# 使用VSCode使用Git

除了命令行，Git还有成千上万的客户端。

其中最省事的是，VSCode直接就集成了git接口。

用VSCode打开目录后，点击侧边栏上的类似Y字形的按钮，它代表的是源码控制面板。

点击以后，如果命令行环境已经安装了Git，这个面板就可以直接使用了

在面板里，CHANGES下边显示的就是被修改的文件，在其上的文本框输入提交说明后，点击上边的√，就可以进行提交。

而在编辑器左下角，则显示着当前分支的名称。

点击这个名称，可以查看分支列表和创建新的分支。

注意在分支名称后边，还有一个云状的图标。

这个就是用来操作远程仓库的了。

点击这个图标，可以把本地代码推送到服务器端。

如果你的本地仓库没有和远程仓库关联，那么是没有这个图标的。

点击源码控制面板右侧的省略号，可以看到大部分常用的git命令。

这些命令足够我们日常使用了。

如果你觉得不够用，还可以通过debug面板切换到终端terminal，直接运行git命令。

# 总结

Git核心设计

首先，Git采用全量存储，记录了项目下文件历次修改的完整信息。

然后，Git计算这些文件的SHA1作为文件名，从而构造了一个天然唯一的、内容寻址的文件系统。

再接着，Git定义了Tree这种数据结构来容纳和描述目录及其下的文件的名称和SHA1值，从而具备了「目录快照」的能力。

并通过Commit记录了每次快照的上级快照，从而构造出了一个链表式的修改记录。

通过在这个链表上游动，对应的目录下的文件就可以在各个提交版本之间切换。

为了方便一次提交多个文件，Git引入了「暂存区」的概念，它和「commit」配合使用，可以很好的管理有一堆文件要修改时的中间状态。

为了实现分布式的版本管理，Git并没有直接把本地的仓库移动到服务器端去，而是在服务器端新加了一个仓库。▼

这样即使没有网络的时候，我们依然可以通过本地的仓库进行版本管理。▼

最后为了能支持多个功能并行开发，Git提供了分支的概念。▼

分支本身只是指向不同链表的指针而已。▼

分支的合并有两种方式，一种是merge，另一种是rebase。▼