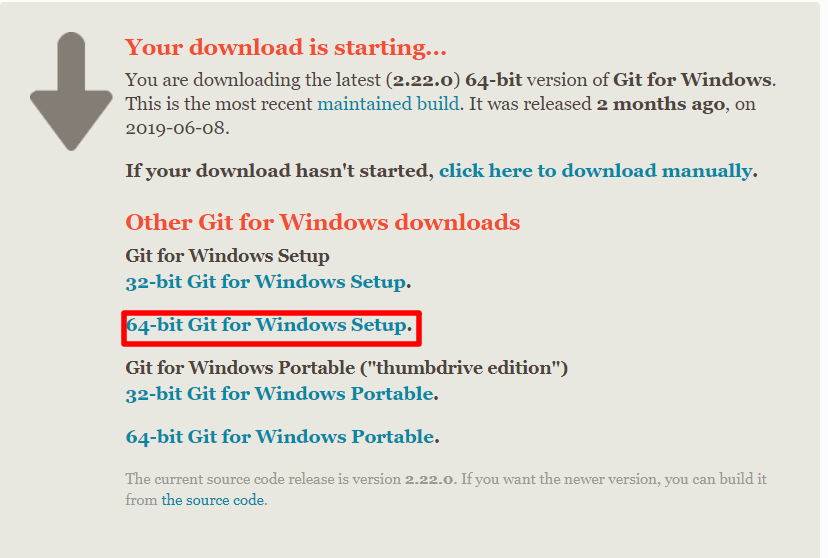
Git实践

# 搭建环境

## git官网

网址：<https://git-scm.com/downloads>下载最新版本

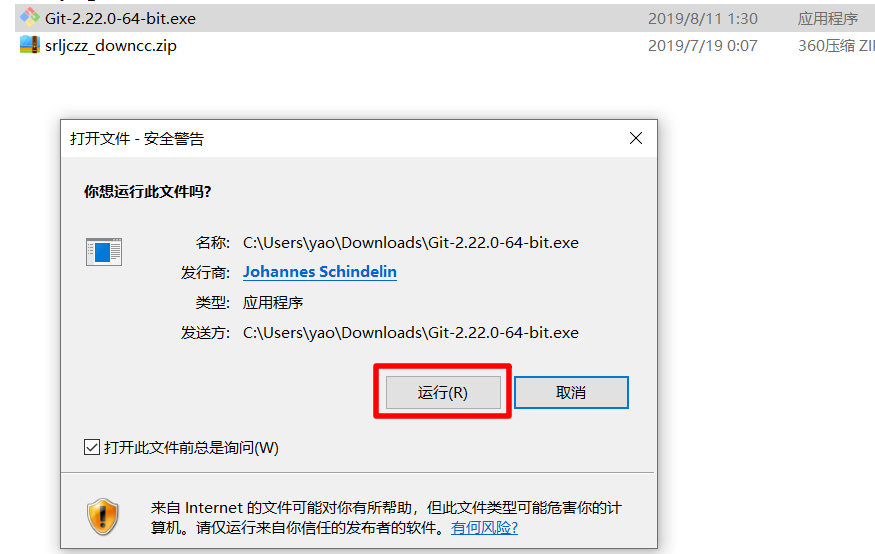


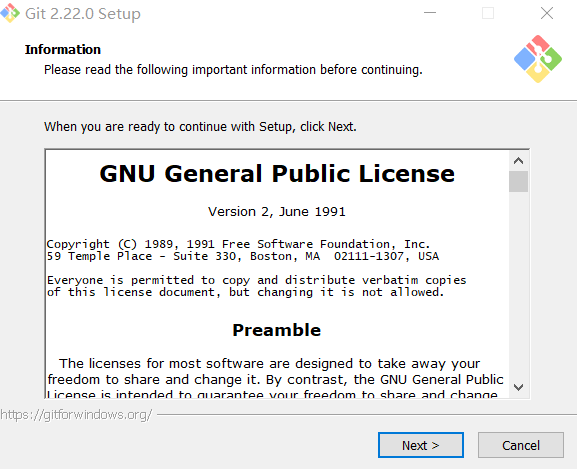
## 百度云

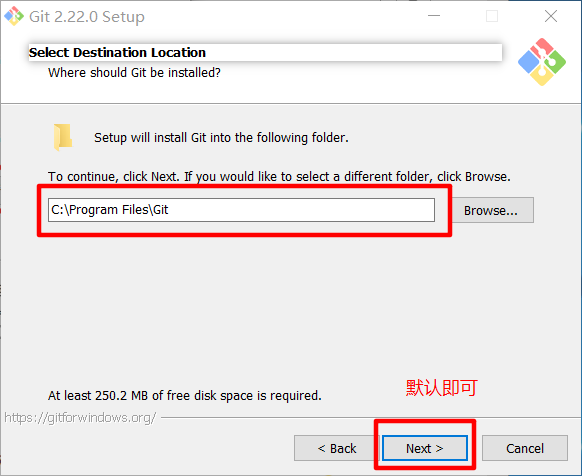
链接：https://pan.baidu.com/s/1RIfeoQnmHI923nSBFkuQTQ

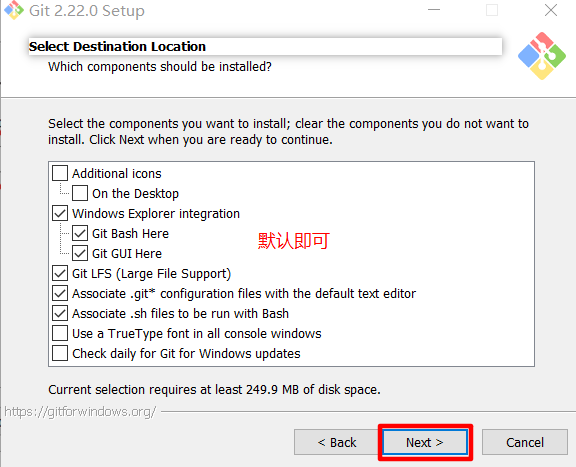
提取码：gs00

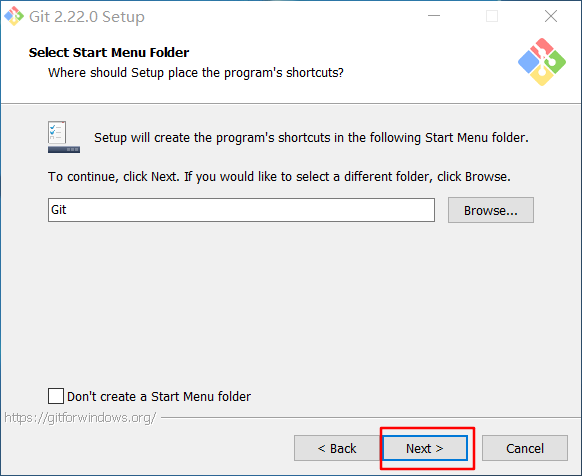
# 安装

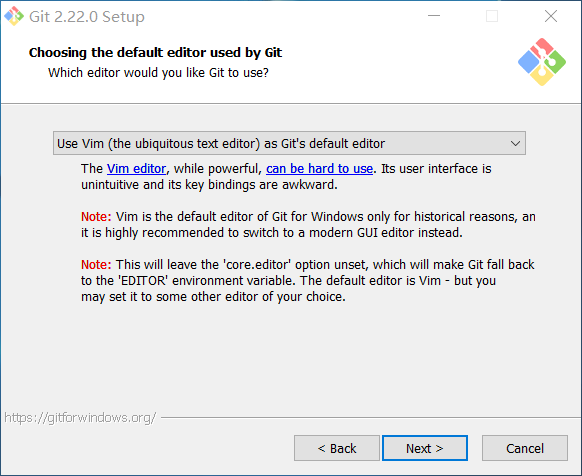


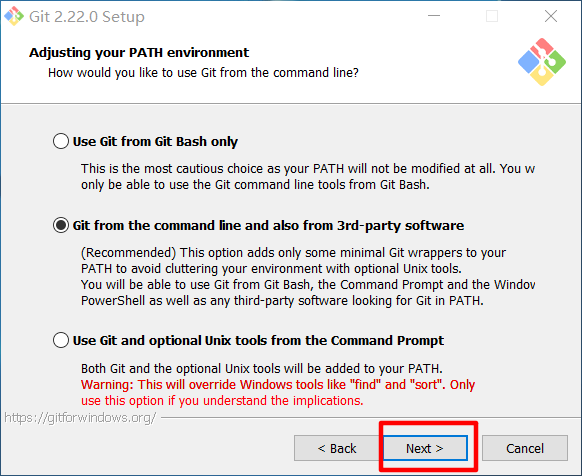


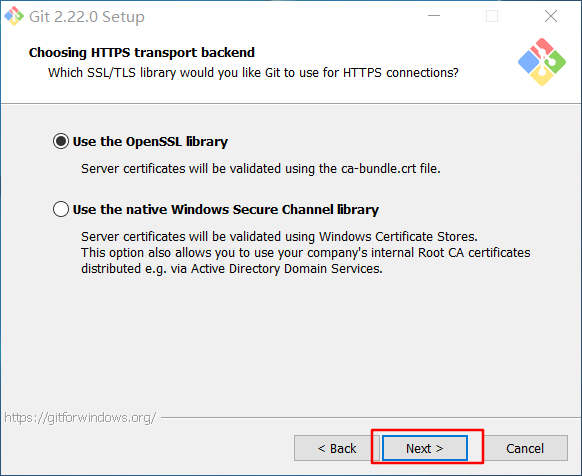


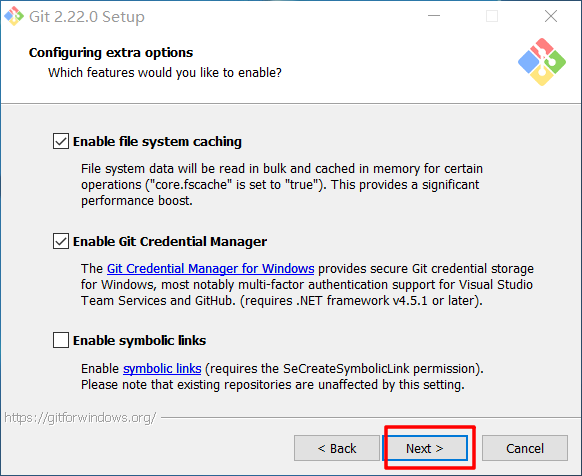


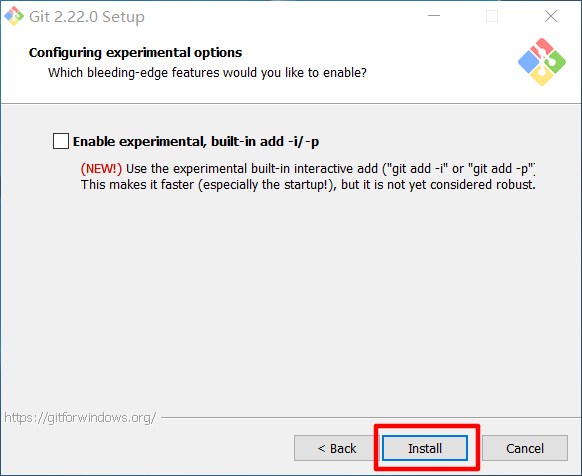




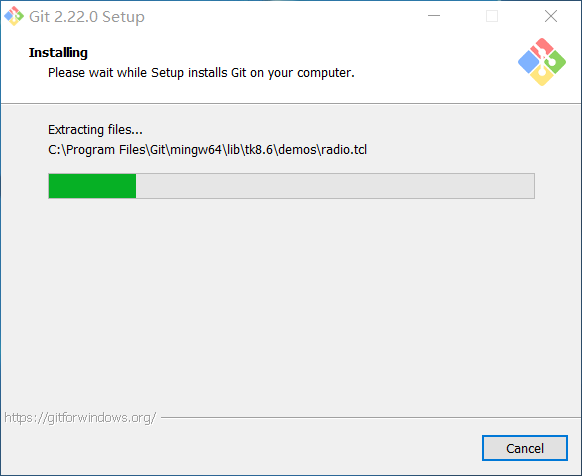




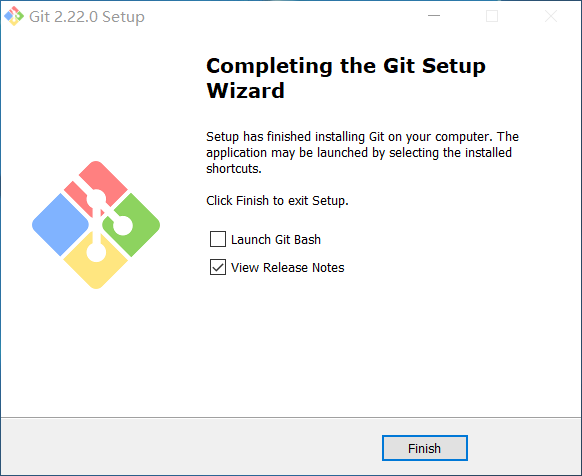




## 安装中



安装完成：



# 设置用户名和邮箱

|  |
| --- |
| yao@DESKTOP-5H1NCBV MINGW64 ~  $ git config --global user.nane "chensy"  yao@DESKTOP-5H1NCBV MINGW64 ~  $ git config --global user.email "shengyaoisme@gmail.com" |

目录

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit

$ pwd

/f/A-LearningRecord/Git/03-learngit

快捷方式

 Ctrl+ins  复制  
 Shift+ins 粘贴

# 创建一个版本库

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：

$ cd learngit

$ pwd

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit

$ pwd

/f/A-LearningRecord/Git/03-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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit

$ git init

Initialized empty Git repository in F:/A-LearningRecord/Git/03-learngit/.git/

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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。

# 把文件添加到版本库

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

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。

不要使用Windows自带的记事本编辑任何文本文件。原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：



编写一个readme.txt文件，内容如下：

Git **is** a version control system.

Git **is** free software.

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ vim readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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。

## 把文件添加到仓库：

第一步，用命令git add告诉Git

$ git add readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add readme.txt

warning: LF will be replaced by CRLF in readme.txt.

The file will have its original line endings in your working directory

出现警告的原因：

首先问题出在不同操作系统所使用的换行符是不一样的，下面罗列一下三大主流操作系统的换行符：

Uinx/Linux采用换行符LF表示下一行（LF：LineFeed，中文意思是换行）；

Dos和Windows采用回车+换行CRLF表示下一行（CRLF：CarriageReturn LineFeed，中文意思是回车换行）；

Mac OS采用回车CR表示下一行（CR：CarriageReturn，中文意思是回车）

在Git中，可以通过以下命令来显示当前你的Git中采取哪种对待换行符的方式

$ git config core.autocrlf

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git config core.autocrlf

true

此命令会有三个输出，“true”，“false”或者“input”

为true时，Git会将你add的所有文件视为文本询问你，将结尾的CRLF转换为LF，而checkout时会再将文件的LF格式转为CRLF格式。

为false时，line endings不做任何改变，文本文件保持其原来的样子。

为input时，add时Git会把CRLF转换为LF，而check时仍旧为LF，所以Windows操作系统不建议设置此值。

建议设置为flash

git config --global core.autocrlf false

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git config --global core.autocrlf false

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

再添加一次：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "wrote a readme file"

[master (root-commit) a970159] wrote a readme file

1 file changed, 2 insertions(+)

create mode 100644 readme.txt

简单解释一下git commit命令，-m后面输入的是本次提交的说明，可以输入任意内容，当然最好是有意义的，这样你就能从历史记录里方便地找到改动记录，想当于SVN的提交日志。

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 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."

## 小结：

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：

1. 使用命令git add <file>，注意，可反复多次使用，添加多个文件；
2. 使用命令git commit -m <message>，完成。

# 版本管理：

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

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

Git **is** free software.

改完之后运行git status命令查看结果：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ vim readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

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

git status命令可以让我们时刻掌握仓库当前的状态，上面的命令输出告诉我们，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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git diff

**diff --git a/readme.txt b/readme.txt**

**index 6de9768..0218057 100644**

**--- a/readme.txt**

**+++ b/readme.txt**

@@ -1,2 +1,2 @@

-Git is a version control sysytem

+Git is free softwarGit is a version control sysytem

Git is free software

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

git diff顾名思义就是查看difference，显示的格式正是Unix通用的diff格式，可以从上面的命令输出看到，我们在第一行添加了一个distributed单词。

知道了对readme.txt作了什么修改后，再把它提交到仓库就放心多了，提交修改和提交新文件是一样的两步，第一步是git add：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "add distributed"

[master a4ca0b1] add distributed

1 file changed, 1 insertion(+), 1 deletion(-)

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ 

提交后，我们再用git status命令看看仓库的当前状态：

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 tree clean）的。

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

nothing to commit, working tree clean

## 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。

# 版本回退

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

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

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

然后尝试提交：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "append GPL"

[master 9cc6bba] append GPL

1 file changed, 1 insertion(+), 1 deletion(-)

像这样，你不断对文件进行修改，然后不断提交修改到版本库里，就好比玩RPG游戏时，每通过一关就会自动把游戏状态存盘，如果某一关没过去，你还可以选择读取前一关的状态。有些时候，在打Boss之前，你会手动存盘，以便万一打Boss失败了，可以从最近的地方重新开始。Git也是一样，每当你觉得文件修改到一定程度的时候，就可以“保存一个快照”，这个快照在Git中被称为commit。一旦你把文件改乱了，或者误删了文件，还可以从最近的一个commit恢复，然后继续工作，而不是把几个月的工作成果全部丢失。

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

版本1：wrote a readme file

Git **is** a version control system.

Git **is** free software.

版本2：add distributed

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

Git **is** free software.

版本3：append GPL

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

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

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git log

commit 9cc6bba44cd1012ae82c9db39f3a190c98857c53 (HEAD -> master)

Author: shengyaoisme <shengyaoisme@gmail.com>

Date: Sun Aug 18 14:25:35 2019 +0800

append GPL

commit a4ca0b150860a3f14b2f5c2a3addb4784efc8c10

Author: shengyaoisme <shengyaoisme@gmail.com>

Date: Sun Aug 18 11:09:46 2019 +0800

add distributed

commit a970159e9f59f07262694088024846f83981daa2

Author: shengyaoisme <shengyaoisme@gmail.com>

Date: Fri Aug 16 23:35:29 2019 +0800

wrote a readme file

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

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：

需要友情提示的是，你看到的一大串类似1094adb...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。

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git log --pretty=oneline

9cc6bba44cd1012ae82c9db39f3a190c98857c53 (HEAD -> master) append GPL

a4ca0b150860a3f14b2f5c2a3addb4784efc8c10 add distributed

a970159e9f59f07262694088024846f83981daa2 wrote a readme file

需要友情提示的是，你看到的一大串类似1094adb...的是commit id（版本号），和SVN不一样，Git的commit id不是1，2，3……递增的数字，而是一个SHA1计算出来的一个非常大的数字，用十六进制表示，而且你看到的commit id和我的肯定不一样，以你自己的为准。为什么commit id需要用这么一大串数字表示呢？因为Git是分布式的版本控制系统，后面我们还要研究多人在同一个版本库里工作，如果大家都用1，2，3……作为版本号，那肯定就冲突了。

每提交一个新版本，实际上Git就会把它们自动串成一条时间线。如果使用可视化工具查看Git历史，就可以更清楚地看到提交历史的时间线：

好了，现在我们启动时光穿梭机，准备把readme.txt回退到上一个版本，也就是add distributed的那个版本，怎么做呢？

首先，Git必须知道当前版本是哪个版本，在Git中，用HEAD表示当前版本，也就是最新的提交1094adb...（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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ cat readme.txt

Git is free softwarGit is a version control sysytem

Git is free software

果然被还原了。

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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git log

commit a4ca0b150860a3f14b2f5c2a3addb4784efc8c10 (HEAD -> master)

Author: shengyaoisme <shengyaoisme@gmail.com>

Date: Sun Aug 18 11:09:46 2019 +0800

add distributed

commit a970159e9f59f07262694088024846f83981daa2

Author: shengyaoisme <shengyaoisme@gmail.com>

Date: Fri Aug 16 23:35:29 2019 +0800

wrote a readme file

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

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1094adb...，于是就可以指定回到未来的某个版本：

$ git re**set** --hard 1094a

HEAD **is** now **at** 83b0afe 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。

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。

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git reset --hard 9cc6bba

HEAD is now at 9cc6bba append GPL

再小心翼翼地看看readme.txt的内容：

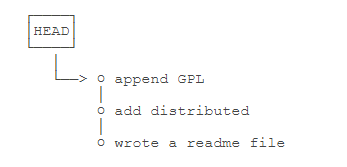
$ cat readme.txt

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

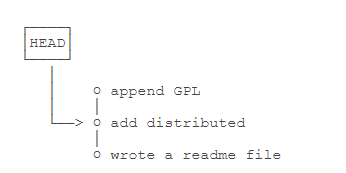
Git **is** free software distributed under the GPL.

果然，版本恢复了。

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



改为指向add distributed：



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

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

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git reflog

9cc6bba (HEAD -> master) HEAD@{0}: reset: moving to 9cc6bba

a4ca0b1 HEAD@{1}: reset: moving to HEAD^

9cc6bba (HEAD -> master) HEAD@{2}: reset: moving to HEAD

9cc6bba (HEAD -> master) HEAD@{3}: commit: append GPL

a4ca0b1 HEAD@{4}: commit: add distributed

a970159 HEAD@{5}: commit (initial): wrote a readme file

终于舒了口气，从输出可知，append GPL的commit id是9cc6bba，现在，你又可以乘坐时光机回到未来了。

## 小结

现在总结一下：

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

# 工作区和暂存区

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

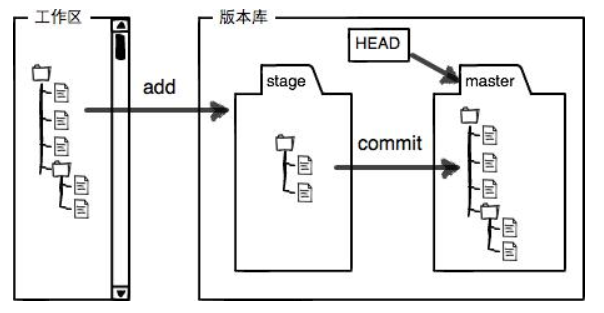
## 工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：

## 版本库（Repository）

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

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



分支和HEAD的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

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

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

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

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

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

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

Git has a mutable index called stage.

然后，在工作区新增一个LICENSE文本文件（内容随便写）。

先用git status查看一下状态：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

LICENSE

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add LICENSE

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes to be committed:

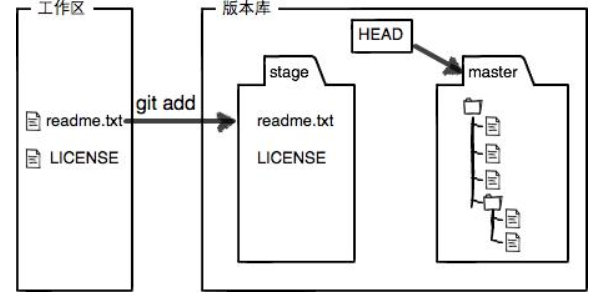
(use "git reset HEAD <file>..." to unstage)

new file: LICENSE

modified: readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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。

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "understand how stage works"

[master cc84c68] understand how stage works

2 files changed, 2 insertions(+)

create mode 100644 LICENSE

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

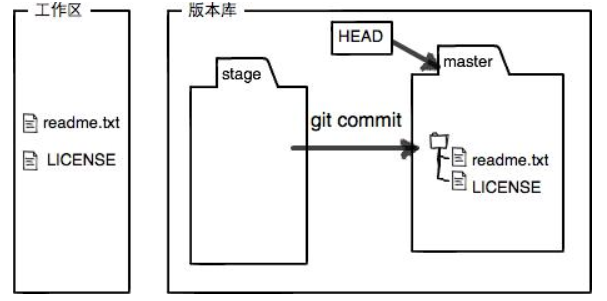
yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

nothing to commit, working tree clean

现在版本库变成了这样，暂存区就没有任何内容了：



### 工作区和暂存区

Git和其他版本控制系统如SVN的一个不同之处就是有暂存区的概念。

先来看名词解释。

### 工作区（Working Directory）

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



### 版本库（Repository）

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。

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



分支和HEAD的概念我们以后再讲。

前面讲了我们把文件往Git版本库里添加的时候，是分两步执行的：

第一步是用git add把文件添加进去，实际上就是把文件修改添加到暂存区；

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

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

你可以简单理解为，需要提交的文件修改通通放到暂存区，然后，一次性提交暂存区的所有修改。

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

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

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

Git has a mutable index called stage.

然后，在工作区新增一个LICENSE文本文件（内容随便写）。

先用git status查看一下状态：

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to **update** what will be committed)

(use "git checkout -- <file>..." **to** discard changes **in** working directory)

modified: readme.txt

Untracked files:

(use "git add <file>..." **to** include **in** what will be committed)

LICENSE

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

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

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

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: LICENSE

modified: readme.txt

现在，暂存区的状态就变成这样了：



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

$ git **commit** -m "understand how stage works"

[master e43a48b] understand how stage works

2 files changed, 2 insertions(+)

**create** mode 100644 LICENSE

一旦提交后，如果你又没有对工作区做任何修改，那么工作区就是“干净”的：

$ git status

On branch master

nothing to **commit**, working tree clean

现在版本库变成了这样，暂存区就没有任何内容了：



## 小结

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

# 管理修改

下面，我们要讨论的就是，为什么Git比其他版本控制系统设计得优秀，因为Git跟踪并管理的是修改，而非文件。

你会问，什么是修改？比如你新增了一行，这就是一个修改，删除了一行，也是一个修改，更改了某些字符，也是一个修改，删了一些又加了一些，也是一个修改，甚至创建一个新文件，也算一个修改。

为什么说Git管理的是修改，而不是文件呢？我们还是做实验。第一步，对readme.txt做一个修改，比如加一行内容：

$ cat readme.txt

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

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

Git has a mutable index called stage.

Git tracks changes.

然后再添加

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

bash: syntax error near unexpected token `('

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: readme.txt

然后，再修改readme.txt：

$ cat readme.txt

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

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

Git has a mutable index called stage.

Git tracks changes of files.

提交：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "git tracks changes"

[master 69f7e96] git tracks changes

1 file changed, 1 insertion(+)

提交后，再看看状态：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

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

咦，怎么第二次的修改没有被提交？

别激动，我们回顾一下操作过程：

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

你看，我们前面讲了，Git管理的是修改，当你用git add命令后，在工作区的第一次修改被放入暂存区，准备提交，但是，在工作区的第二次修改并没有放入暂存区，所以，git commit只负责把暂存区的修改提交了，也就是第一次的修改被提交了，第二次的修改不会被提交。

提交后，用git diff HEAD -- readme.txt命令可以查看工作区和版本库里面最新版本的区别：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git diff HEAD -- readme.txt

**diff --git a/readme.txt b/readme.txt**

**index a3bb765..fddd75f 100644**

**--- a/readme.txt**

**+++ b/readme.txt**

@@ -1,4 +1,4 @@

Git is free softwarGit is a version control sysytem

Git is free software distributed under the GPL.

Git has a mutable index called stage.

-Git tracks changes.

+Git tracks changes of files.

可见，第二次修改确实没有被提交。

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

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

好，现在，把第二次修改提交了，然后开始小结。

## 小结

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

# 撤销修改：

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

$ cat readme.txt

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

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

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

在你准备提交前，一杯咖啡起了作用，你猛然发现了stupid boss可能会让你丢掉这个月的奖金！

既然错误发现得很及时，就可以很容易地纠正它。你可以删掉最后一行，手动把文件恢复到上一个版本的状态。如果用git status查看一下：

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to **update** what will be committed)

(use "git checkout -- <file>..." **to** discard changes **in** working directory)

modified: readme.txt

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

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

$ git checkout -- readme.txt

命令git checkout -- readme.txt意思就是，把readme.txt文件在工作区的修改全部撤销，这里有两种情况：

一种是readme.txt自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是readme.txt已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

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

现在，看看readme.txt的文件内容：

$ cat readme.txt

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

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

Git has a mutable index called stage.

Git tracks changes of files.

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

现在假定是凌晨3点，你不但写了一些胡话，还git add到暂存区了：

$ cat readme.txt

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

My stupid boss still prefers SVN.

$ git add readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ vim readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add readme.txt

庆幸的是，在commit之前，你发现了这个问题。用git status查看一下，修改只是添加到了暂存区，还没有提交：

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: readme.txt

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

$ git re**set** HEAD readme.txt

Unstaged changes after reset:

M readme.txt

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

再用git status查看一下，现在暂存区是干净的，工作区有修改：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git reset HEAD readme.txt

Unstaged changes after reset:

M readme.txt

再用git status查看一下，现在暂存区是干净的，工作区有修改：

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

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

还记得如何丢弃工作区的修改吗？

$ git checkout *-- readme.txt*

$ git status

On branch master

nothing to **commit**, working tree clean

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git checkout -- readme.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

nothing to commit, working tree clean

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

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](https://www.liaoxuefeng.com/wiki/896043488029600/897013573512192)一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得Git是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把stupid boss提交推送到远程版本库，你就真的惨了……

## 小结

又到了小结时间。

场景1：当你改乱了工作区某个文件的内容，想直接丢弃工作区的修改时，用命令git checkout -- file。

场景2：当你不但改乱了工作区某个文件的内容，还添加到了暂存区时，想丢弃修改，分两步，第一步用命令git reset HEAD <file>，就回到了场景1，第二步按场景1操作。

场景3：已经提交了不合适的修改到版本库时，想要撤销本次提交，参考[版本回退](https://www.liaoxuefeng.com/wiki/896043488029600/897013573512192)一节，不过前提是没有推送到远程库。

# 删除文件

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ vim test.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git add test.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git commit -m "add test.txt"

[master 83ca168] add test.txt

1 file changed, 1 insertion(+)

create mode 100644 test.txt

一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用rm命令删了：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git status

On branch master

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: test.txt

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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

$ git rm test.txt

rm 'test.txt'

$ git **commit** -m "remove test.txt"

[master d46f35e] remove test.txt

1 file changed, 1 deletion(-)

**delete** mode 100644 test.txt

现在，文件就从版本库中被删除了。

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

$ git checkout -- test.txt

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

## 小结

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

# 远程仓库基础

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

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

没错，如果只是在一个仓库里管理文件历史，Git和SVN真没啥区别。为了保证你现在所学的Git物超所值，将来绝对不会后悔，同时为了打击已经不幸学了SVN的童鞋，本章开始介绍Git的杀手级功能之一（注意是之一，也就是后面还有之二，之三……）：远程仓库。

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

你肯定会想，至少需要两台机器才能玩远程库不是？但是我只有一台电脑，怎么玩？

其实一台电脑上也是可以克隆多个版本库的，只要不在同一个目录下。不过，现实生活中是不会有人这么傻的在一台电脑上搞几个远程库玩，因为一台电脑上搞几个远程库完全没有意义，而且硬盘挂了会导致所有库都挂掉，所以我也不告诉你在一台电脑上怎么克隆多个仓库。

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

完全可以自己搭建一台运行Git的服务器，不过现阶段，为了学Git先搭个服务器绝对是小题大作。好在这个世界上有个叫[GitHub](https://github.com/)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。

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ ssh-keygen -t rsa -C "shengyaoisme@gmail.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/c/Users/yao/.ssh/id\_rsa):

Created directory '/c/Users/yao/.ssh'.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /c/Users/yao/.ssh/id\_rsa.

Your public key has been saved in /c/Users/yao/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:3qE9bsCSCx+ep4hxkyn0ww59D6E0eevF2XX5hvFKhGs shengyaoisme@gmail.com

The key's randomart image is:

+---[RSA 3072]----+

| |

| |

| |

| . . . |

| . + ooS . o = |

| . =.\*+=o\* o + = |

| + @++=\*.= E o +|

| B \*=+...o . o |

| . o oo... . |

+----[SHA256]-----+

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

yao@DESKTOP-5H1NCBV MINGW64 ~/Desktop

$ cd /c/Users/yao/.ssh

yao@DESKTOP-5H1NCBV MINGW64 ~/.ssh

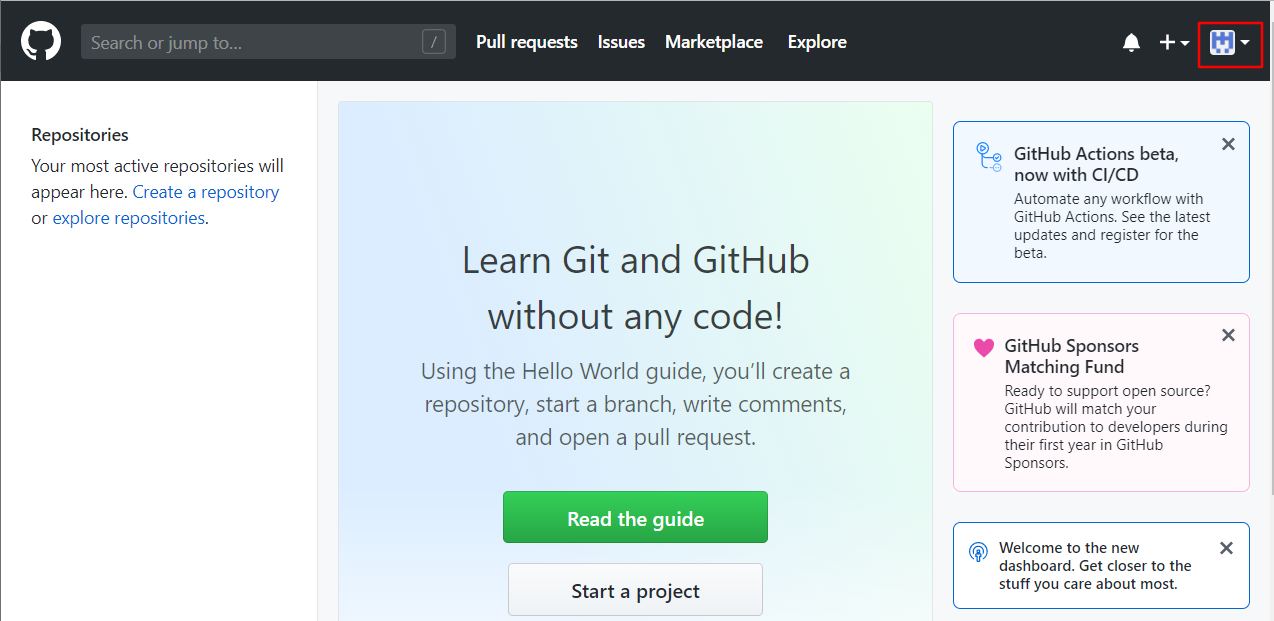
$ ls

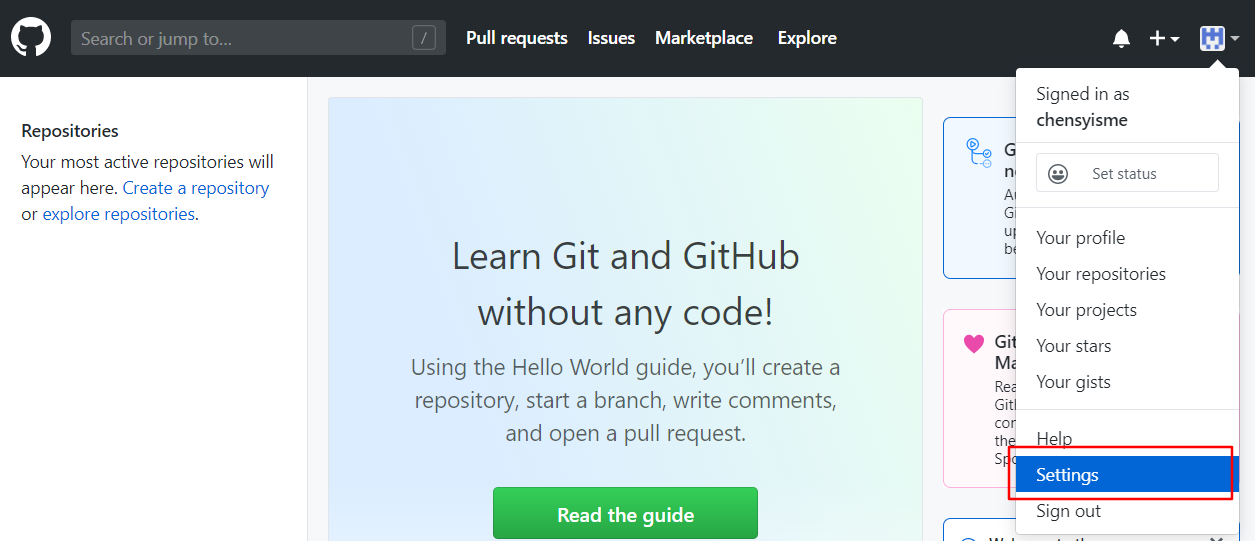
id\_rsa id\_rsa.pub

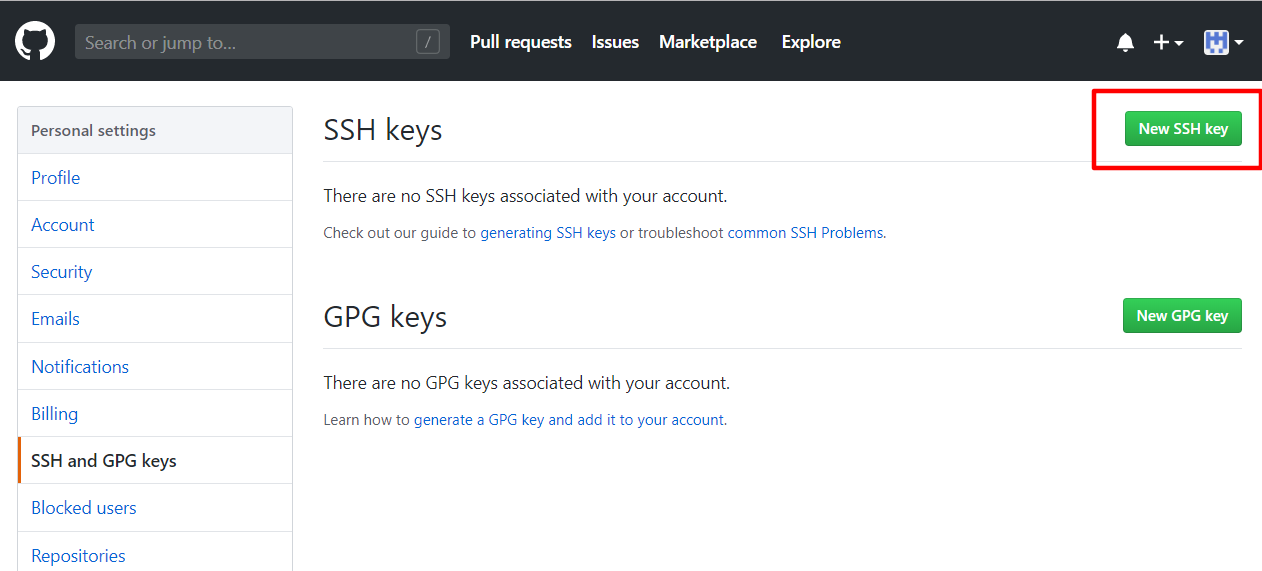
C:\Users\yao\.ssh

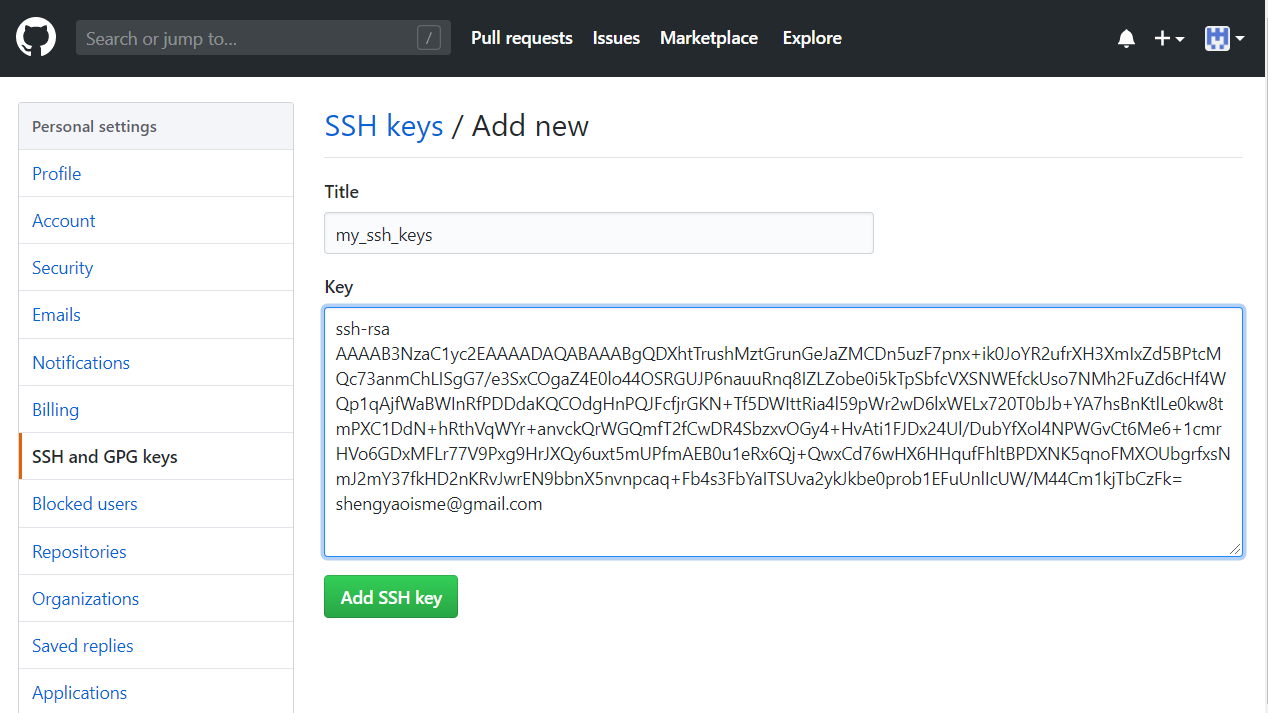
第2步：登陆GitHub

GitHub点击用户头像，选择setting









点“Add Key”，可以看到已经添加的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。

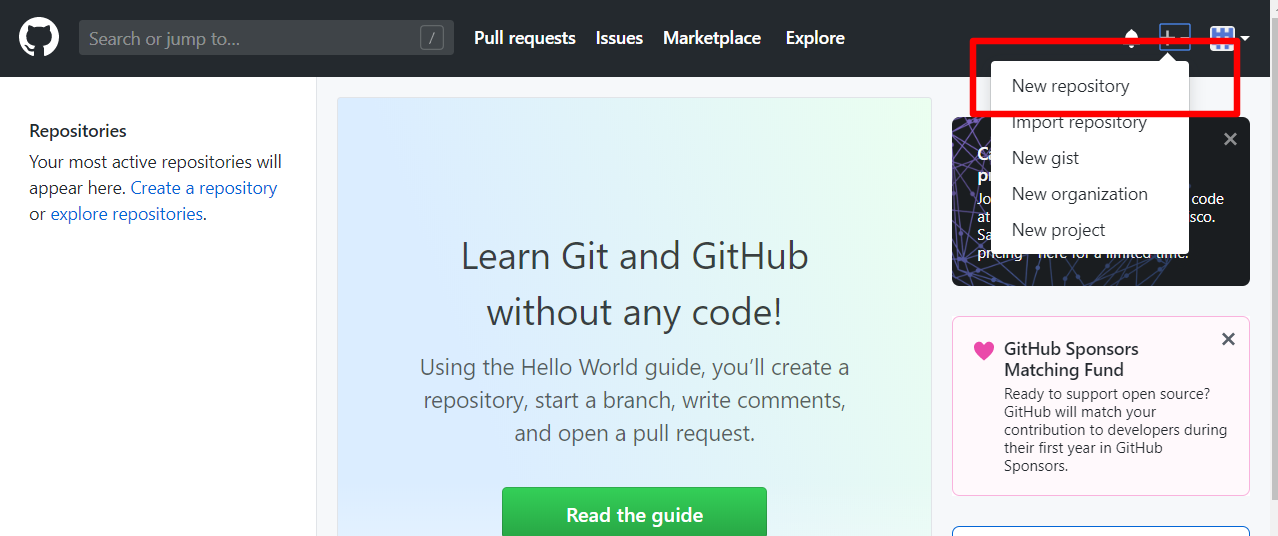
如果你不想让别人看到Git库，有两个办法，一个是交点保护费，让GitHub把公开的仓库变成私有的，这样别人就看不见了（不可读更不可写）。另一个办法是自己动手，搭一个Git服务器，因为是你自己的Git服务器，所以别人也是看不见的。这个方法我们后面会讲到的，相当简单，公司内部开发必备。

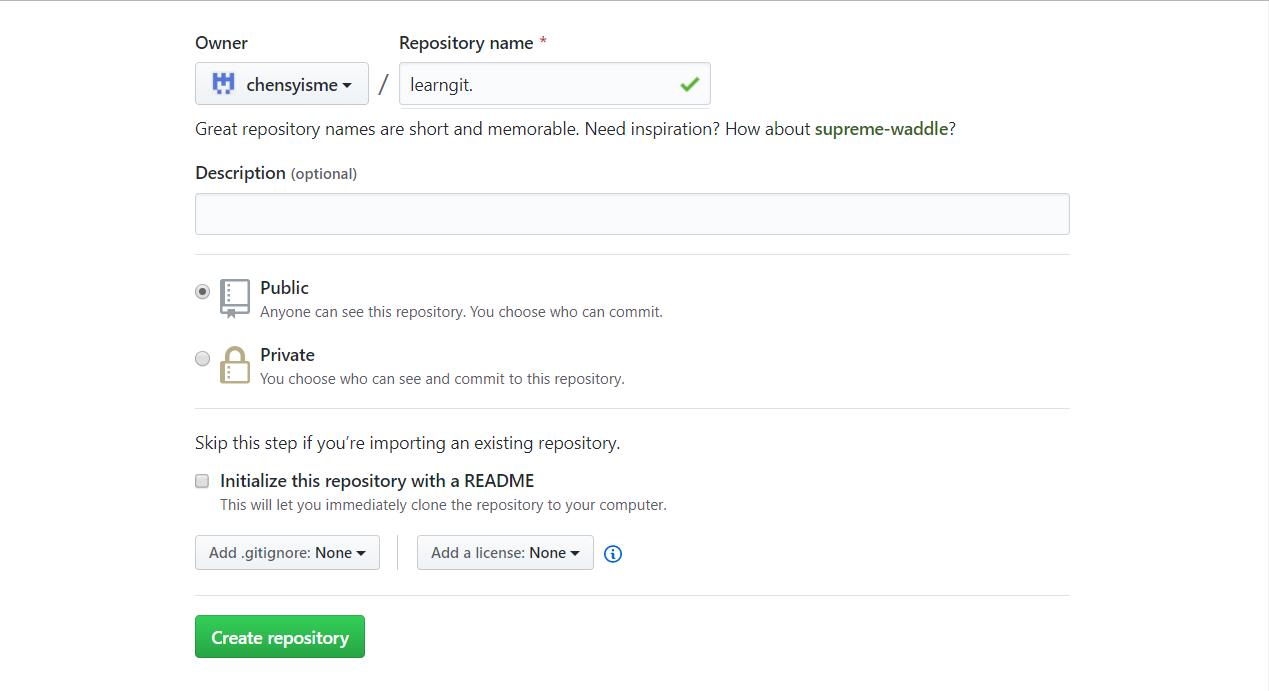
确保你拥有一个GitHub账号后，我们就即将开始远程仓库的学习。

# 添加远程仓库

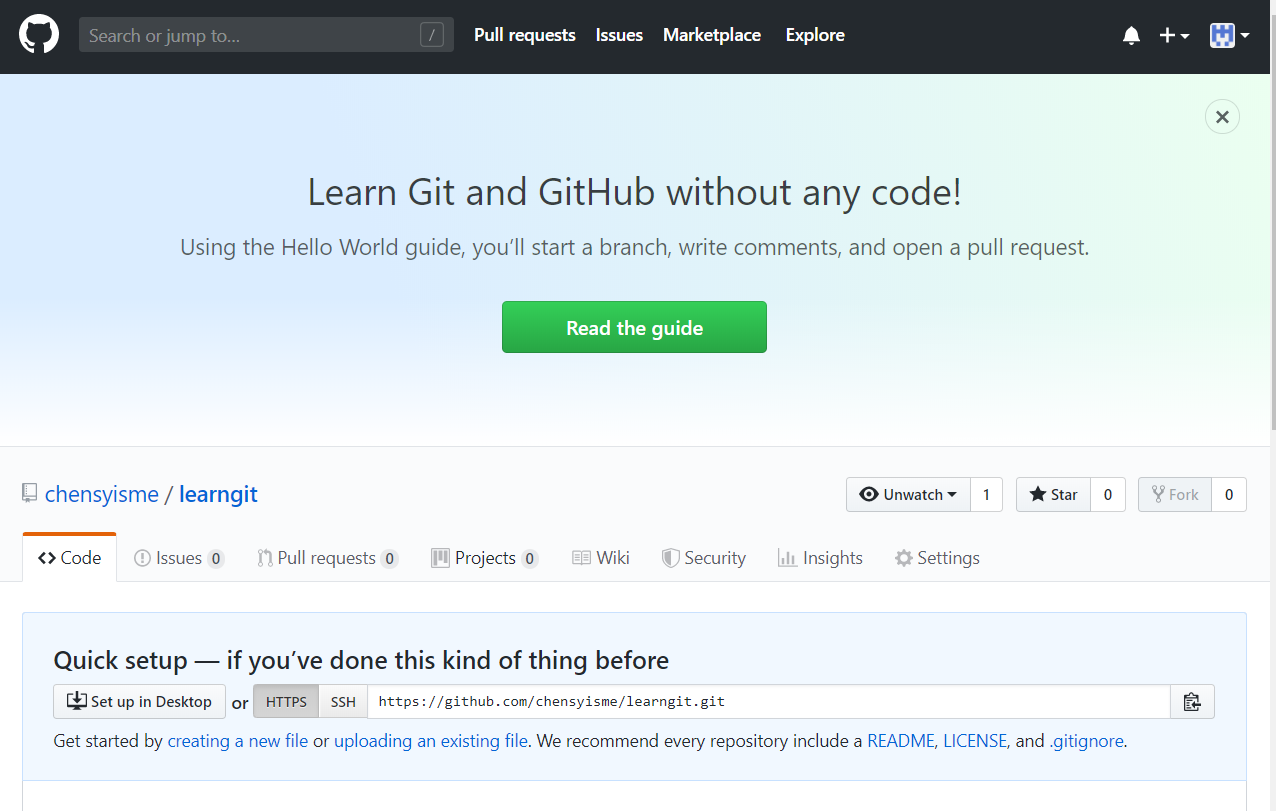
现在的情景是，你已经在本地创建了一个Git仓库后，又想在GitHub创建一个Git仓库，并且让这两个仓库进行远程同步，这样，GitHub上的仓库既可以作为备份，又可以让其他人通过该仓库来协作，真是一举多得。

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





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



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

现在，我们根据GitHub的提示，在本地的learngit仓库下运行命令：

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

错误示范：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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

usage: git remote add [<options>] <name> <url>

-f, --fetch fetch the remote branches

--tags import all tags and associated objects when fetching

or do not fetch any tag at all (--no-tags)

-t, --track <branch> branch(es) to track

-m, --master <branch>

master branch

--mirror[=(push|fetch)]

set up remote as a mirror to push to or fetch from

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

正确的方法：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git push -u origin master

The authenticity of host 'github.com (13.250.177.223)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no/[fingerprint])? y

Please type 'yes', 'no' or the fingerprint: y

Please type 'yes', 'no' or the fingerprint: y

Please type 'yes', 'no' or the fingerprint: y

Please type 'yes', 'no' or the fingerprint: yes

Warning: Permanently added 'github.com,13.250.177.223' (RSA) to the list of known hosts.

Enumerating objects: 19, done.

Counting objects: 100% (19/19), done.

Delta compression using up to 4 threads

Compressing objects: 100% (14/14), done.

Writing objects: 100% (19/19), 1.51 KiB | 49.00 KiB/s, done.

Total 19 (delta 4), reused 0 (delta 0)

remote: Resolving deltas: 100% (4/4), done.

To github.com:chensyisme/learngit.git

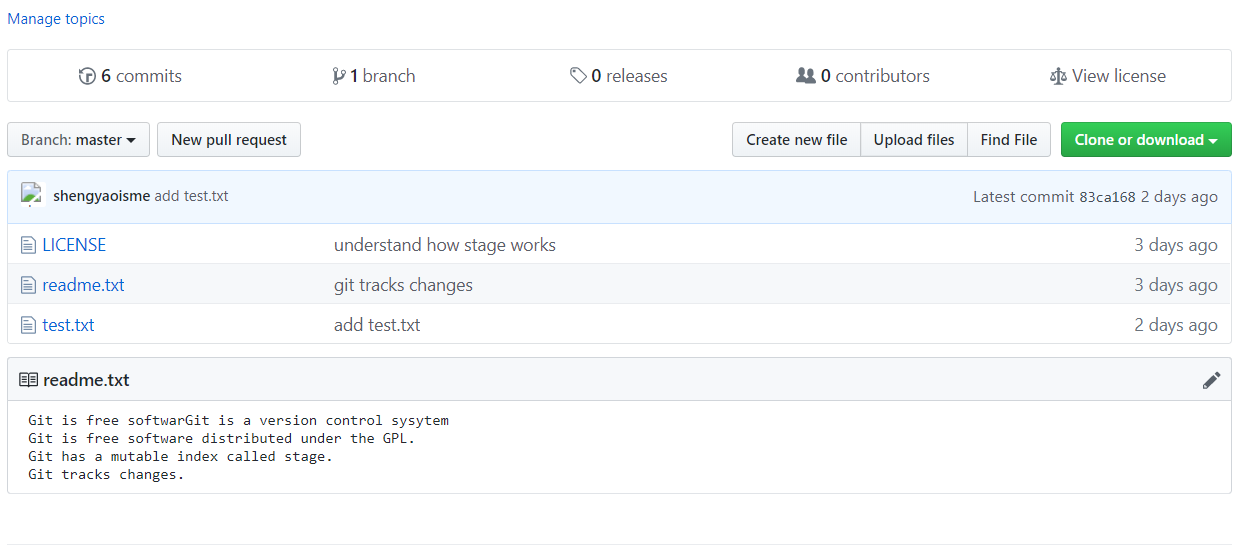
\* [new branch] master -> master

Branch 'master' set up to track remote branch 'master' from 'origin'.

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

由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令。

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



从现在起，只要本地作了提交，就可以通过命令

$ git push origin master

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

## SSH警告

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

The authenticity of host 'github.com (xx.xx.xx.xx)' can't be established.

RSA key fingerprint is xx.xx.xx.xx.xx.

Are you sure you want to continue connecting (yes/no)?

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

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

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

这个警告只会出现一次，后面的操作就不会有任何警告了。

如果你实在担心有人冒充GitHub服务器，输入yes前可以对照[GitHub的RSA Key的指纹信息](https://help.github.com/articles/what-are-github-s-ssh-key-fingerprints/)是否与SSH连接给出的一致。

## 小结

要关联一个远程库，使用命令git remote add origin git@server-name:path/repo-name.git；

关联后，使用命令git push -u origin master第一次推送master分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令git push origin master推送最新修改；

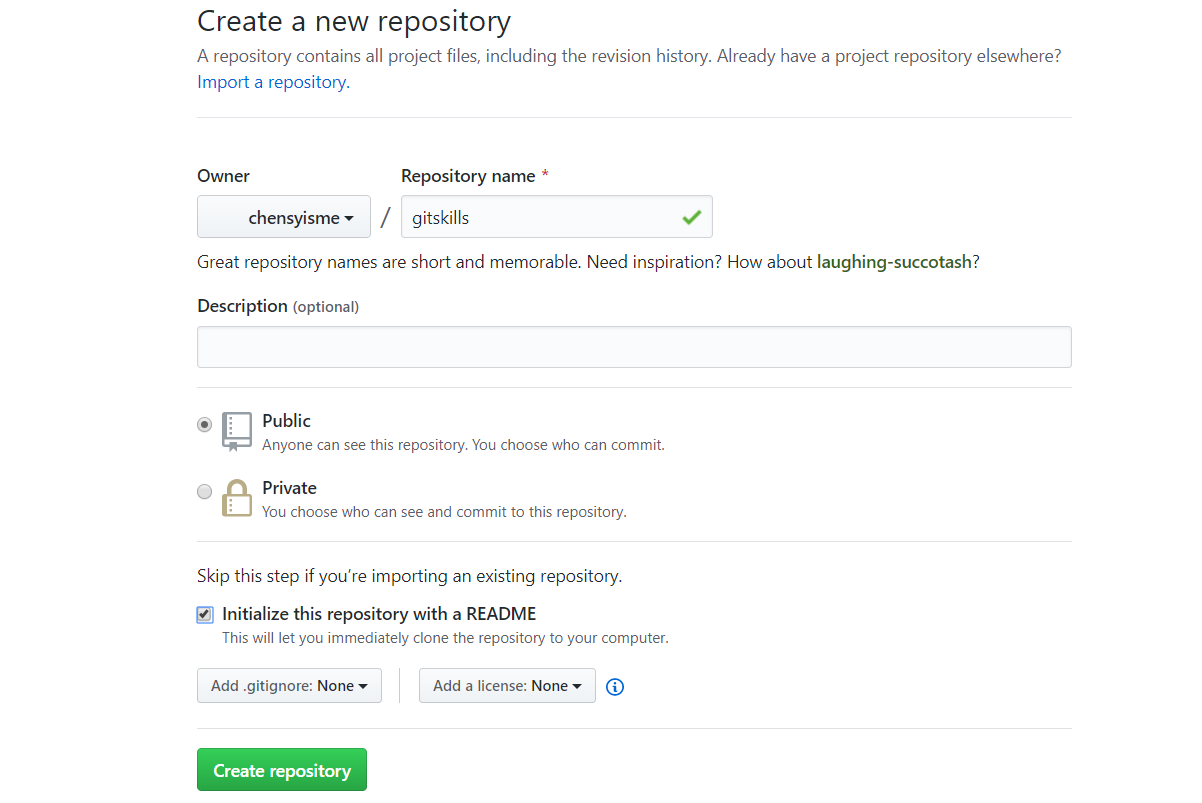
分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而SVN在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

# 从远程库克隆

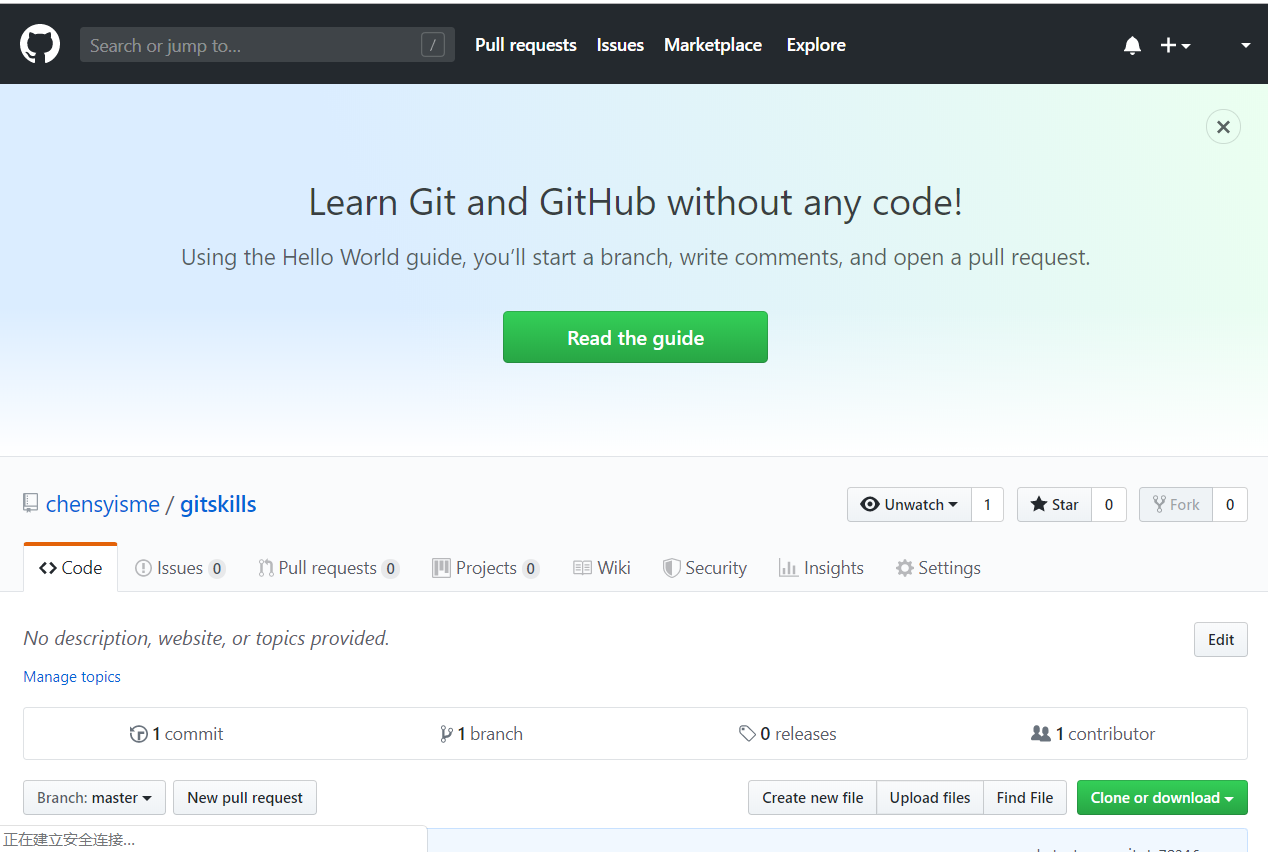
先有本地库，后有远程库的时候，如何关联远程库？

现在，假设我们从零开发，那么最好的方式是先创建远程库，然后，从远程库克隆。

首先，登陆GitHub，创建一个新的仓库，名字叫gitskills：



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



现在，远程库已经准备好了，下一步是用命令git clone克隆一个本地库：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git clone git@github.com:chensyisme/gitskills.git

Cloning into 'gitskills'...

Warning: Permanently added the RSA host key for IP address '13.229.188.59' to the list of known hosts.

remote: Enumerating objects: 3, done.

remote: Counting objects: 100% (3/3), done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Receiving objects: 100% (3/3), done.

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：

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ cd gitskills/

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit/gitskills (master)

$ ls

README.md

如果有多个人协作开发，那么每个人各自从远程克隆一份就可以了。

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

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

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

## 小结

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

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

# 分支管理

举个例子：

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

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

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

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

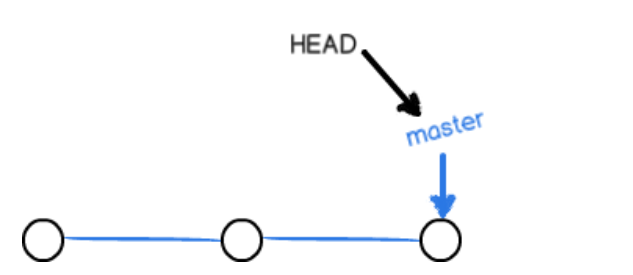
其他版本控制系统如SVN等都有分支管理，但是用过之后你会发现，这些版本控制系统创建和切换分支比蜗牛还慢，简直让人无法忍受，结果分支功能成了摆设，大家都不去用。

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

## 创建与合并分支

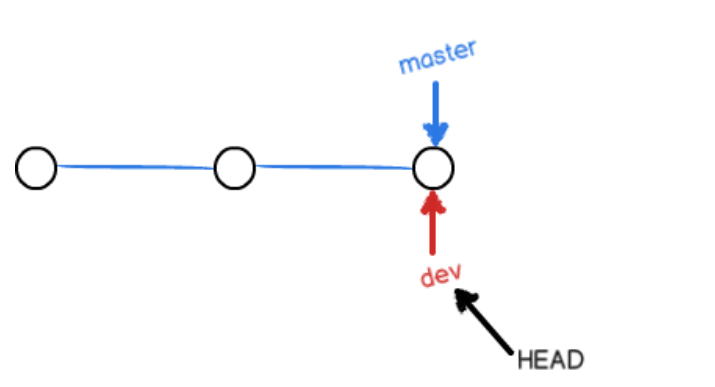
在[版本回退](https://www.liaoxuefeng.com/wiki/896043488029600/897013573512192)里，可以知道知道，每次提交，Git都把它们串成一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在Git里，这个分支叫主分支，即master分支。HEAD严格来说不是指向提交，而是指向master，master才是指向提交的，所以，HEAD指向的就是当前分支。

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



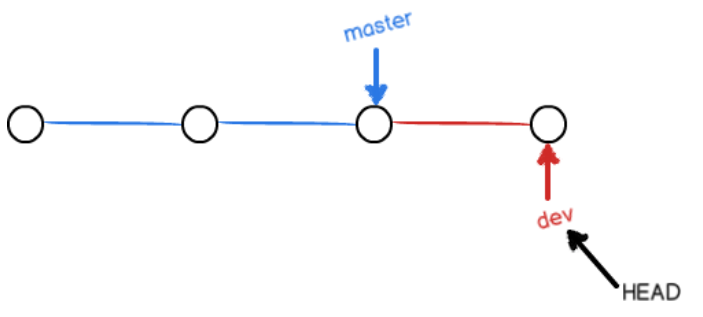
每次提交，master分支都会向前移动一步，这样，随着你不断提交，master分支的线也越来越长。

当我们创建新的分支，例如dev时，Git新建了一个指针叫dev，指向master相同的提交，再把HEAD指向dev，就表示当前分支在dev上：

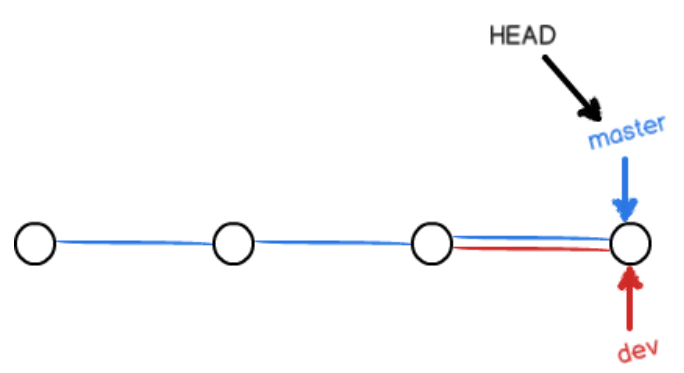


你看，Git创建一个分支很快，因为除了增加一个dev指针，改改HEAD的指向，工作区的文件都没有任何变化！

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

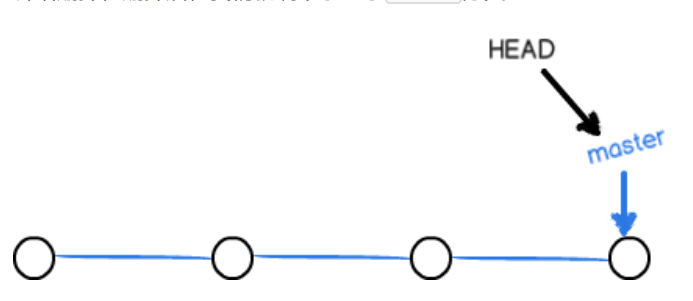


假如我们在dev上的工作完成了，就可以把dev合并到master上。Git怎么合并呢？最简单的方法，就是直接把master指向dev的当前提交，就完成了合并：



所以Git合并分支也很快！就改改指针，工作区内容也不变！

合并完分支后，甚至可以删除dev分支。删除dev分支就是把dev指针给删掉，删掉后，我们就剩下了一条master分支：



下面开始实战。

首先，我们创建dev分支，然后切换到dev分支：

$ git checkout -b dev

Switched to a new branch 'dev'

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git checkout -b dev

Switched to a new branch 'dev'

D test.txt

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (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

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (dev)

$ 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.

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (dev)

$ vim readme.txt

然后提交：

$ git add readme.txt

$ git commit -m "branch test"

[dev b17d20e] branch test

1 file changed, 1 insertion(+)

The most similar command is

commit

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (dev)

$ git commit -m "branch test"

[dev 3b15d0c] 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'

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (dev)

$ git checkout master

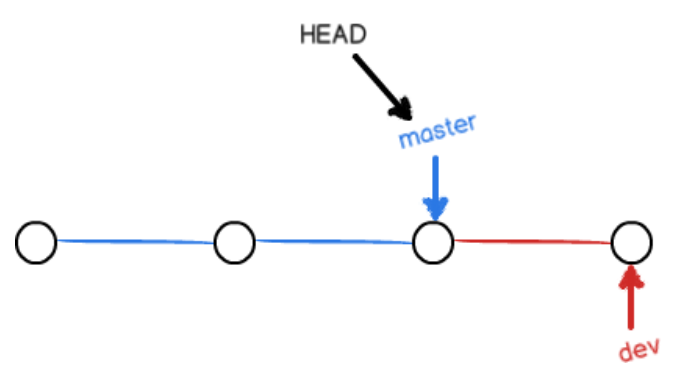
Switched to branch 'master'

D test.txt

Your branch is up to date with 'origin/master'.

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

现在，我们把dev分支的工作成果合并到master分支上：



$ git merge dev

Updating d46f35e..b17d20e

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git merge dev

Updating 83ca168..3b15d0c

Fast-forward

readme.txt | 1 +

1 file changed, 1 insertion(+)

git merge命令用于合并指定分支到当前分支。合并后，再查看readme.txt的内容，就可以看到，和dev分支的最新提交是完全一样的。

注意到上面的Fast-forward信息，Git告诉我们，这次合并是“快进模式”，也就是直接把master指向dev的当前提交，所以合并速度非常快。

当然，也不是每次合并都能Fast-forward，我们后面会讲其他方式的合并。

合并完成后，就可以放心地删除dev分支了：

$ git branch -d dev

Deleted branch dev (was b17d20e).

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git branch -d dev

Deleted branch dev (was 3b15d0c).

删除后，查看branch，就只剩下master分支了：

$ git branch

\* master

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git branch

\* master

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

### 小结

Git鼓励大量使用分支：

查看分支：git branch

创建分支：git branch <name>

切换分支：git checkout <name>

创建+切换分支：git checkout -b <name>

合并某分支到当前分支：git merge <name>

删除分支：git branch -d <name>

## 解决冲突

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

$ git checkout -b feature1

Switched to a new branch 'feature1'

M readme.txt

D test.txt

修改readme.txt最后一行，改为：

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

在feature1分支上提交：

$ git add readme.txt

$ git **commit** -m "AND simple"

[feature1 14096d0] **AND** simple

1 file changed, 1 insertion(+), 1 deletion(-)

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (feature1)

$ git commit -m "AND simple"

[feature1 3274fc7] AND simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 1 **commit**.

(use "git push" **to** publish your **local** commits)

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (feature1)

$ git checkout master

Switched to branch 'master'

D test.txt

Your branch is ahead of 'origin/master' by 1 commit.

(use "git push" to publish your local commits)

yao@DESKTOP-5H1NCBV MINGW64 /f/A-LearningRecord/Git/03-learngit (master)

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

在master分支上把readme.txt文件的最后一行改为：

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

提交：

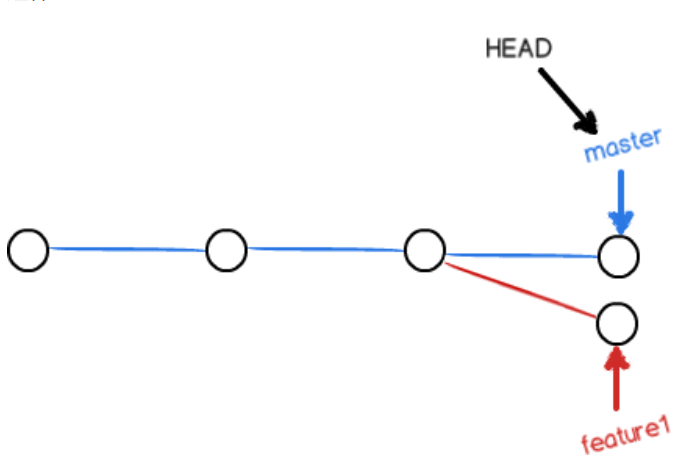
$ git add readme.txt

$ git commit -m "& simple"

[master 5dc6824] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

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



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

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict **in** readme.txt

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

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

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(**use** "git push" to publish your local commits)

You have unmerged paths.

(fix conflicts **and** run "git commit")

(**use** "git merge --abort" to abort the merge)

Unmerged paths:

(**use** "git add <file>..." to mark resolution)

both modified: readme.txt

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

我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

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

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

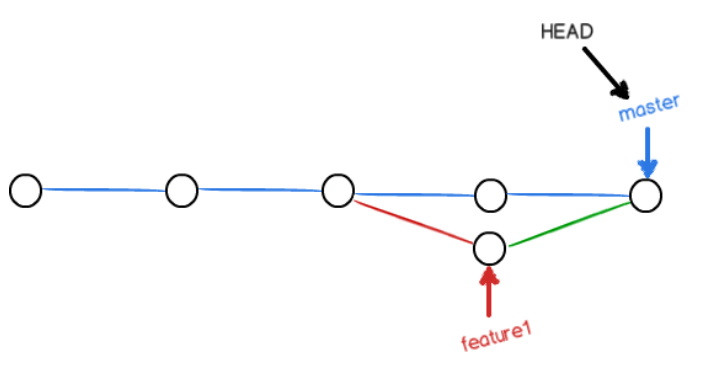
再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master cf810e4] conflict fixed

现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：

$ git log --graph --pretty=oneline --abbrev-commit

\* cf810e4 (HEAD -> master) conflict fixed

|\

| \* 14096d0 (feature1) **AND** simple

\* | 5dc6824 & simple

|/

\* b17d20e branch test

\* d46f35e (origin/master) remove test.txt

\* b84166e add test.txt

\* 519219b git tracks changes

\* e43a48b understand how stage works

\* 1094adb append GPL

\* e475afc add distributed

\* eaadf4e wrote a readme file

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 14096d0).

### 解决问题：

人生不如意之事十之八九，合并分支往往也不是一帆风顺的。

准备新的feature1分支，继续我们的新分支开发：

$ git checkout -b feature1

Switched to a new branch 'feature1'

修改readme.txt最后一行，改为：

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

在feature1分支上提交：

$ git add readme.txt

$ git **commit** -m "AND simple"

[feature1 14096d0] **AND** simple

1 file changed, 1 insertion(+), 1 deletion(-)

切换到master分支：

$ git checkout master

Switched to branch 'master'

Your branch is ahead of 'origin/master' by 1 **commit**.

(use "git push" **to** publish your **local** commits)

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

在master分支上把readme.txt文件的最后一行改为：

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

提交：

$ git add readme.txt

$ git commit -m "& simple"

[master 5dc6824] & simple

1 file changed, 1 insertion(+), 1 deletion(-)

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



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

$ git merge feature1

Auto-merging readme.txt

CONFLICT (content): Merge conflict **in** readme.txt

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

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

$ git status

On branch master

Your branch is ahead of 'origin/master' by 2 commits.

(**use** "git push" to publish your local commits)

You have unmerged paths.

(fix conflicts **and** run "git commit")

(**use** "git merge --abort" to abort the merge)

Unmerged paths:

(**use** "git add <file>..." to mark resolution)

both modified: readme.txt

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

我们可以直接查看readme.txt的内容：

Git is a distributed version control system.

Git is free software distributed under the GPL.

Git has a mutable index called stage.

Git tracks changes of files.

<<<<<<< HEAD

Creating a new branch is quick & simple.

=======

Creating a new branch is quick AND simple.

>>>>>>> feature1

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

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

再提交：

$ git add readme.txt

$ git commit -m "conflict fixed"

[master cf810e4] conflict fixed

现在，master分支和feature1分支变成了下图所示：



用带参数的git log也可以看到分支的合并情况：

$ git log --graph --pretty=oneline --abbrev-commit

\* cf810e4 (HEAD -> master) conflict fixed

|\

| \* 14096d0 (feature1) **AND** simple

\* | 5dc6824 & simple

|/

\* b17d20e branch test

\* d46f35e (origin/master) remove test.txt

\* b84166e add test.txt

\* 519219b git tracks changes

\* e43a48b understand how stage works

\* 1094adb append GPL

\* e475afc add distributed

\* eaadf4e wrote a readme file

最后，删除feature1分支：

$ git branch -d feature1

Deleted branch feature1 (was 14096d0).

工作完成。

### 小结

当Git无法自动合并分支时，就必须首先解决冲突。解决冲突后，再提交，合并完成。

解决冲突就是把Git合并失败的文件手动编辑为我们希望的内容，再提交。

用git log --graph命令可以看到分支合并图。