

一、Git 的诞生

Linux 在 1991 年创建了开源的 Linux，从此，Linux 系统不断发展，已经成为最大的服务器系统软件了。Linux 虽然创建了 Linux，但 Linux 的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为 Linux 编写代码，那 Linux 的代码是如何管理的呢？

在 2002 年以前，世界各地的志愿者把源代码文件通过 diff 的方式发给 Linux，然后由 Linux 本人通过手工方式合并代码！虽然有免费的 CVS、SVN 这些免费的版本管理系统，但 Linux 坚定地反对 CVS 和 SVN，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比 CVS、SVN 好用，但那是付费的，和 Linux 的开源精神不符。

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

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

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

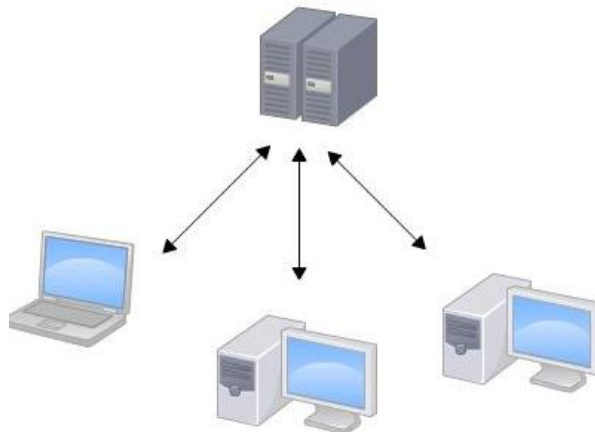
Linux 花了两周时间自己用 C 写了一个分布式版本控制系统，这就是 Git！一个月之内，Linux 系统的源码已经由 Git 管理了！牛是怎么定义的呢？大家可以体会一下：Git 迅速成为最流行的分布式版本控制系统，尤其是 2008 年，GitHub 网站上线了，它为开源项目免费提供 Git 存储，无数开源项目开始迁移至 GitHub，包括 jQuery，PHP，Ruby 等等。

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

二、集中式 vs 分布式

集中式和分布式版本控制系统有什么区别呢？

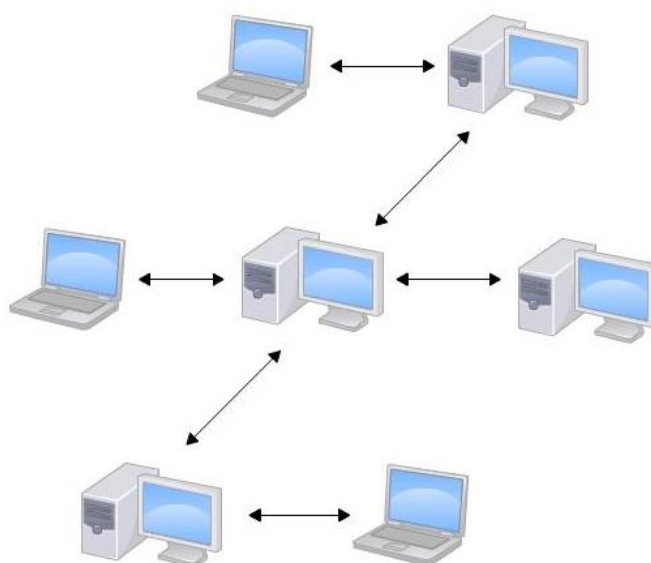
集中式版本控制系统，版本库是几种存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。中央服务器好比一个图书馆，你要改一本书，必须先把书从图书馆借出来，然后回家自己修改，改完后，再放回图书馆。



集中式版本最大的毛病就是必须要联网才能工作。如果在局域网内还好，如果在互联网上，遇到网速慢的话，可能提交一个 10M 的文件就需要 5 分钟，不得把人熬死。

而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，工作的时候就不需要联网了。应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？比方说你在自己的电脑上改了文件 A，你的同事也在他的电脑上改了文件 A，这是，你们俩只需要把各自的修改推送给对方，就可以互相看到对方的修改了。

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



【局域网内两个 IP 之间推送：一般选用 SSH 协议。考虑到通信安全问题，需要进行身份验证，即需要在 linux 上开启 SSH 服务，允许另一端连接。简单讲，就是把一端的 ssh-key 加入到 linux 端的 know-host 列表中，然后输入：\$ git clone [user@192.168.xxx.xxx:/home/test/test.code](http://192.168.xxx.xxx:/home/test/test.code)】

【创建本地服务器：<http://blog.csdn.net/moqiang02/article/details/39609311>】

三、安装

1. 在 Linux 上安装 Git:

- Ubuntu、Debian:
较新一点的 Linux: \$ sudo apt-get install git
较老一点的 Linux: \$ sudo apt-get install git-core
- 其他版本的 Linux: 先从 Git 官网下载源码，然后解压，依次输入: ./config ; make ; sudo make install

以 CentOS 为例:

Git 官网下载最新版本源代码: <https://www.kernel.org/pub/software/scm/git/>

然后编译并安装:

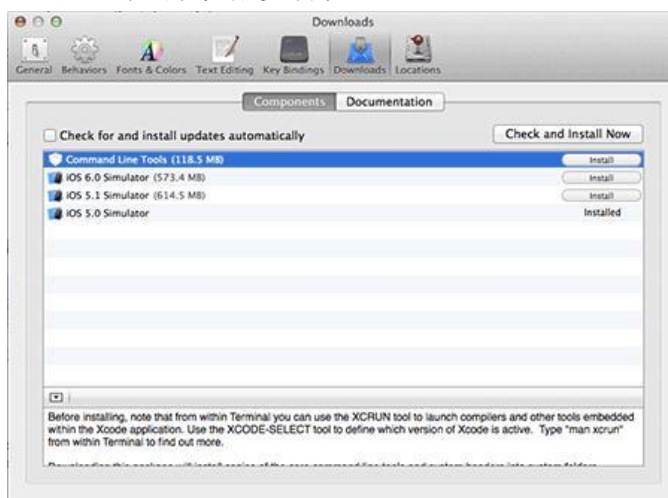
```
$ xz -d git-2.9.5.tar.xz
$ tar -xvf git-2.9.5.tar
$ cd git-2.9.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

现在已经可以用 git 命令了, 用 git 把 Git 项目仓库克隆到本地, 以便日后随时更新: <http://chenxi@bit.d.com:7990/scm/f17002/recorder.git>

2. 在 Mac OS X 上安装 Git:

一是安装 homebrew, 然后通过 homebrew 安装 Git, 具体方法请参考 homebrew 的文档: <http://brew.sh/>。

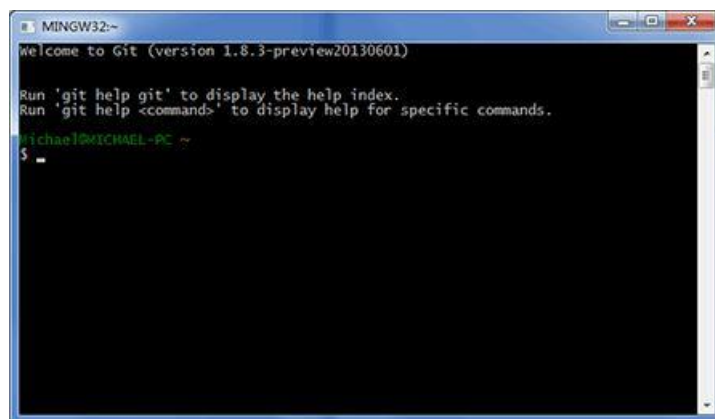
第二种方法更简单, 也是推荐的方法, 就是直接从 AppStore 安装 Xcode, Xcode 集成了 Git, 不过默认没有安装, 你需要运行 Xcode, 选择菜单 “Xcode” -> “Preferences”, 在弹出窗口中找到 “Downloads”, 选择 “Command Line Tools”, 点 “Install” 就可以完成安装了。



3. 在 Windows 上安装 Git:

msysgit 是 Windows 版的 Git, 从 <http://msysgit.github.io/> 下载, 然后按默认选项安装即可。

安装完成后, 在开始菜单里找到 “Git” -> “Git Bash”, 蹦出一个类似命令行窗口的东西, 就说明 Git 安装成功!



安装完成后，还需要最后一步设置，在命令行输入：

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

因为 Git 是分布式版本控制系统，所以，每个机器都必须自报家门：你的名字和 Email 地址

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。

`git config --list` 查看配置信息。

配置文件位置：

<code>--global</code>	使用全局配置文件
<code>--system</code>	使用系统级配置文件
<code>--local</code>	使用版本库级配置文件
<code>-f, --file <文件></code>	使用指定的配置文件

四、创建版本库（仓库）：

英文名 **repository**，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，以便任何时候都可以追踪历史，或者在将来某个时刻可以“还原”

1. 选择一个合适的地方，创建空目录：

```
$ mkdir learngit
$ cd learngit
$ pwd
/Users/michael/learngit
```

2. 通过 `git init` 命令把这个目录变成 Git 可以管理的仓库：

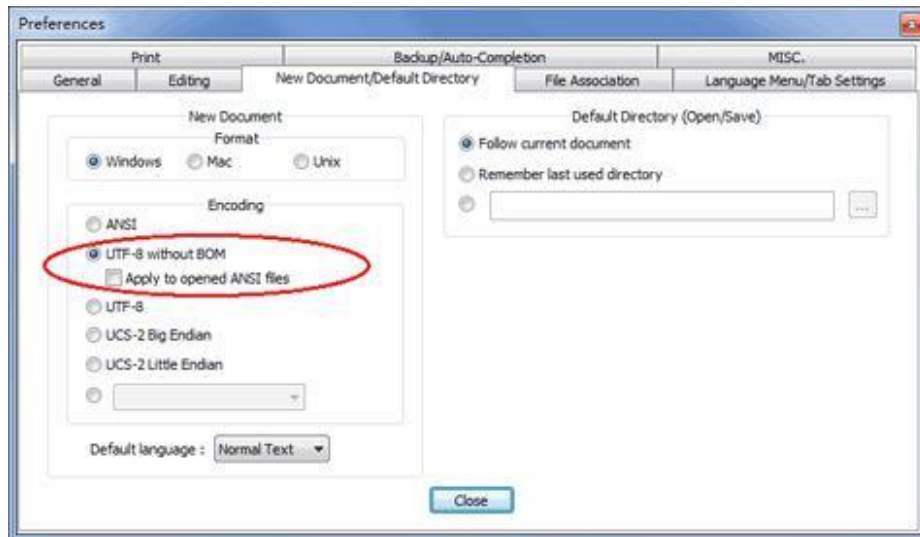
```
$ git init
Initialized empty Git repository in /Users/michael/learngit/.git/
```

此时，Git 仓库已建好，且为一个空仓库，该目录下多了一个 `.git` 的目录，这个目录是 Git 来跟踪管理版本库的，不要手动修改这个目录里的文件，不然 Git 仓库会被破坏。若没有看到 `.git` 目录，那是因为这个目录是隐藏的，用 `ls -ah` 命令可以看见。

五、把文件添加到版本库：

所有版本控制系统，只能跟踪文本文件的改动，如 `txt`，网页，代码等。**Microsoft word** 是二进制格式，无法被版本控制系统跟踪。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。

使用 Windows 时，千万不要使用 Windows 自带的记事本编辑任何文本文件。原因是 Microsoft 开发记事本的团队使用了一个非常弱智的行为来保存 UTF-8 编码的文件，他们自作聪明地在每个文件开头添加了 `0xefbbbf`（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“？”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载 **Notepad++** 代替记事本，不但功能强大，而且免费！记得把 **Notepad++** 的默认编码设置为 **UTF-8 without BOM** 即可：



以将 readme.txt 放入仓库为例：

readme 中内容如下：

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

1. 用命令 `git add` 告诉 Git，把文件添加到仓库

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

2. 用命令 `git commit` 告诉 Git，把文件提交到仓库：

```
$ git commit -m "wrote a readme file"  
[master (root-commit) cb926e7] wrote a readme file  
1 file changed, 2 insertions(+)  
create mode 100644 readme.txt
```

`git commit` 命令中，`-m` 后面输入的是本次提交的说明
`commit` 可以一次提交很多文件，可以多次 `add` 不同的文件：

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

`git commit -a`：联合了两句

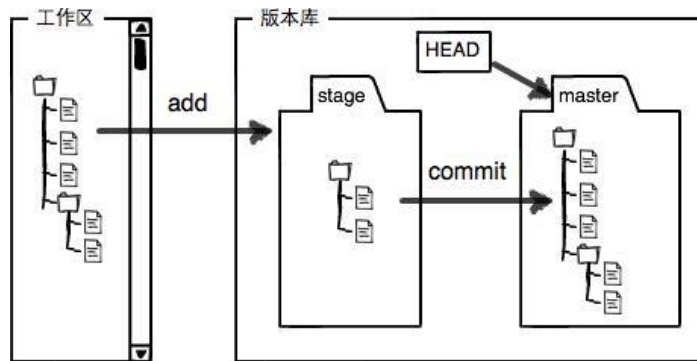
六、工作区和暂存区

电脑中能够看到的目录（如当前文件夹）就是一个工作区。

版本库：

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 版本库中存了很多东西，其中最重要的就是称为 `stage` 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` 的一个指针叫做 `HEAD` 需要提交的文件修改统统放到暂存区，然后一次性提交暂存区的所有修改。



七、库中文件修改

1. 修改库中文件：

将 readme 的内容修改为：

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

运行仓库当前状态命令 `git status` 查看结果：

```
$ 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")
```

上面命令说明 `readme.txt` 已被修改过，但是没有提交。

查看修改前后文件的不同：`git diff` 命令

```
$ git diff readme.txt
diff --git a/readme.txt b/readme.txt
index 46d49bf..9247db6 100644
--- a/readme.txt
+++ b/readme.txt
@@ -1,2 +1,2 @@
-Git is a version control system.
+Git is a distributed version control system.
 Git is free software.
```

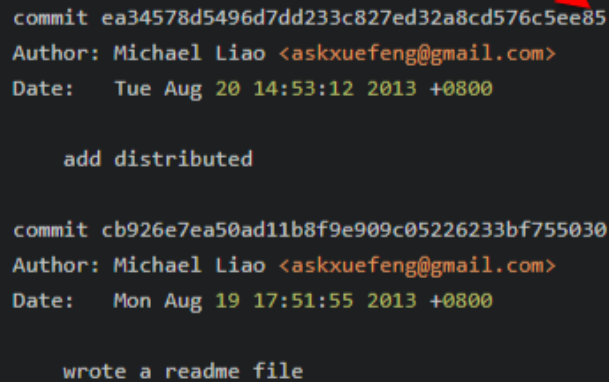
提交仓库：

`git add readme.txt`

`git commit -m "add distributed"`

查看历史 `git log`

id号，提交后由Git自动串成一条时间线



```
commit ea34578d5496d7dd233c827ed32a8cd576c5ee85
Author: Michael Liao <askxuefeng@gmail.com>
Date: Tue Aug 20 14:53:12 2013 +0800

    add distributed

commit cb926e7ea50ad11b8f9e909c05226233bf755030
Author: Michael Liao <askxuefeng@gmail.com>
Date: Mon Aug 19 17:51:55 2013 +0800

    wrote a readme file
```

`git log --pretty=oneline` 输出简单历史信息 每条一行

```
ea34578d5496d7dd233c827ed32a8cd576c5ee85 add distributed
cb926e7ea50ad11b8f9e909c05226233bf755030 wrote a readme file
```

在 Git 中，用 HEAD 表示当前版本，也就是最新的提交 ea3478.....6c5ee85，上一个版本就是 HEAD^，上上个版本就是 HEAD^^，往上 100 个可以写成：HEAD~100

我们把当前版本回退到上一个版本，可是有 `git reset` 命令：

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

```
HEAD is now at cb926e wrote a readme file
```

```
$ cat readme.txt //查看 readme 内容
```

命令窗口还未关闭的情况下，还可以追回新版本：(id 号写前几位就好，Git 会自动去找，不要只写一两位)

```
$ git reset --hard ea34578
```

```
HEAD is now at ea34578 add distributed
```

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，回退版本的时候，Git 仅仅是把 HEAD 从指向 add distributed 改为指向 wrote a readme file
`git reflog` 命令查看每一次版本更新信息及其 id 号，可以有选择版本跳转。

【慎用

```
git reset - -hard/soft/mix:
```

<http://blog.csdn.net/carolzhang8406/article/details/49761927>】

2. 修改管理：

Git 比其他版本控制系统设计的优秀，是因为 Git 跟踪并管理的是修改，而非文件。

若操作为：第一次修改-> `git add` -> 第二次修改 -> `git commit` 则只是将第一次修改的版本提交到版本库，而第二次修改的文件并未放到暂存区。

每次修改，如果不 `add` 到暂存区，就不会加入到 `commit`。

`git checkout -- file` 可以丢弃工作区的修改：

`git checkout -- readme.txt` 可以把 `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，又做了修改，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` 或者 `git add` 时的状态，前提是还没有把自己的本地版本推送到远程。

【注：该命令中的 `--` 很重要，没有 `--` 就变成了“切换到另一个分支”的命令，分支管理中会用到 `git checkout`】

3. 删除文件

在工作区新建 `test.txt` 文件，并提交到版本库：

```
$ git add test.txt
$ git commit -m "add test.txt"
[master 94cdc44] add test.txt
1 file changed, 1 insertion(+)
create mode 100644 test.txt
```

删除工作区中的 `test.txt` 文件：

```
$ rm test.txt
```

`git status` 命令可获取被删除的文件：

```
$ 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")
```

- 用 `git rm` 命令从版本库中删除该文件，并且 `git commit`：

```
$ git rm test.txt
rm 'test.txt'
$ git commit -m "remove test.txt"
[master d17efd8] remove test.txt
1 file changed, 1 deletion(-)
delete mode 100644 test.txt
```

- 误删，需要恢复工作区的原文件：

```
$ 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”。

八、远程仓库

GitHub 网站就是提供 Git 仓库托管服务的，只要注册一个 GitHub 账号，就可以免费获得 Git 远程仓库。首先自行注册 GitHub 账号，由于本地 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：

- 创建 SSH Key

若用户主目录下，已经存在 .ssh 目录，且这个目录下存在 id_rsa 和 id_rsa.pub 这两个文件，则跳到下一步。

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"
```

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。

此时用户目录里可以找到 .ssh 目录，里面有 id_rsa 和 id_rsa.pub 两个文件，这两个就是 SSH Key 的密钥对，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。


- 登录 GitHub

打开“Account settings”“SSH Key”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：

SSH keys

[New SSH key](#)

This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.

 **git_test**
Fingerprint: 23:5d:e1:b5:cb:2b:91:f5:70:d3:8a:98:dd:82:87:2c
Added on 19 Oct 2017
Never used — Read/write

Delete

通过 SSH Key，GitHub 可以识别出是你推送的提交确实是你推送的，而不是别人冒充。GitHub 允许添加多个 Key。假定你有若干个电脑，在不同地点，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。

【在 GitHub 上免费托管的 Git 仓库，任何人都可以看得到，不要把敏感信息放进去】

如果不想让别人看到 Git 库，可以交保护费，让 GitHub 把公开仓库变成私有的；或者自己动手搭一个 Git 服务器，公司内部开发必备。

1. 添加远程库

若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，又可以让他通过该仓库来协作。

首先，登录 GitHub，然后右上角找到“new repository”按钮，创建一个新的仓库：

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: xiaodouzi0411 / Repository name: git_test ✓

Great repository names are short and memorable. Need inspiration? How about [studious-goggles](#).

Description (optional)

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None Add a license: None ⓘ

Create repository

目前，新建的 `git_test` 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。

根据提示，在本地 `git_test` 目录下，运行：

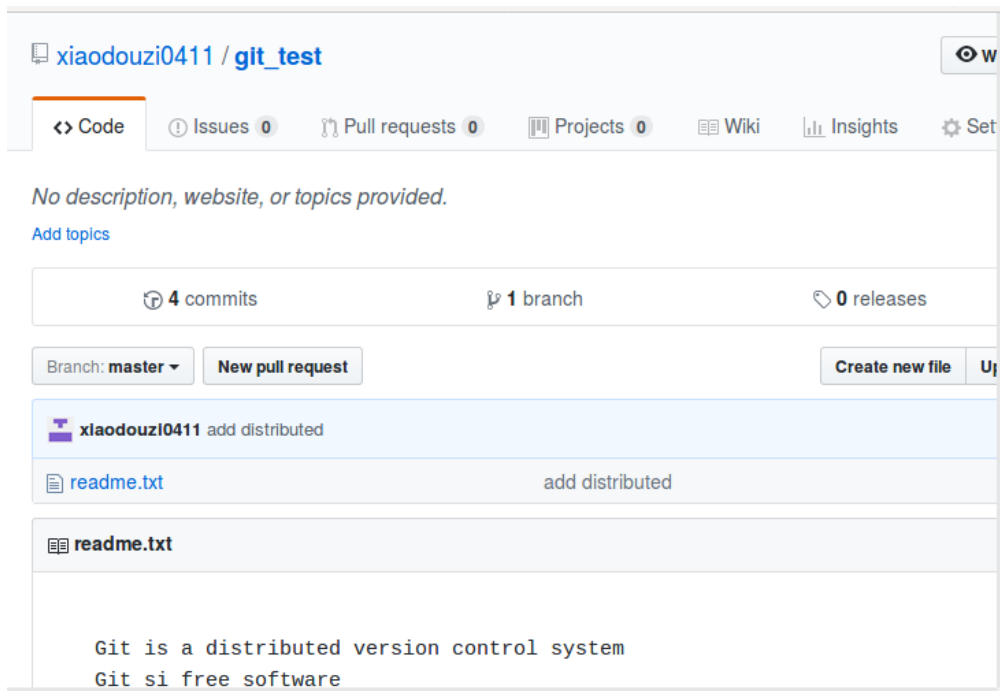
```
$ git remote add origin https://github.com/xiaodouzi0411/git_test.git
```

```
$ git push -u origin master
```

```
xiaodouzi@ubuntu:~/git_test$ git remote add origin https://github.com/xiaodouzi0411/git_test.git
xiaodouzi@ubuntu:~/git_test$ git push -u origin master
Username for 'https://github.com': xiaodouzi92
Password for 'https://xiaodouzi92@github.com':
xiaodouzi@ubuntu:~/git_test$ git push -u origin master
Username for 'https://github.com': xiaodouzi0411
Password for 'https://xiaodouzi0411@github.com':
Counting objects: 11, done.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (11/11), 966 bytes | 0 bytes/s, done.
Total 11 (delta 0), reused 0 (delta 0)
To https://github.com/xiaodouzi0411/git_test.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

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：



现在只要在本地做了修改，就可以通过

```
$ 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

【git add -> git commit -> git push】

2. 从远程克隆

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，登录 GitHub，创建一个新的仓库，名字叫 gitskills:

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

xiaodouzi0411

Repository name

gitskills

Great repository names are short and memorable. Need inspiration? How about **glowing-guide**.

Description (optional)



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

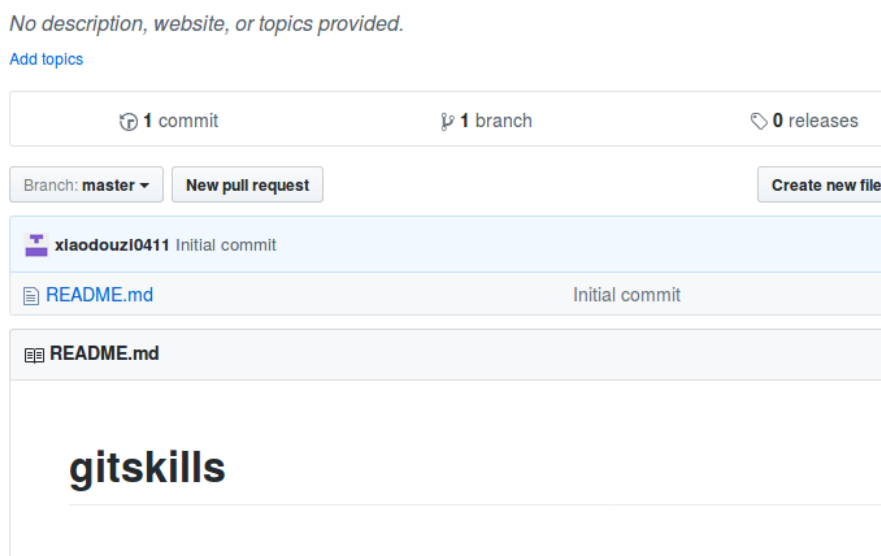
☒ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add a license: [None](#)

Add a README: [None](#)

最终显示：



远程库已备好，下一步是用命令 `git clone` 克隆一个本地库

```
xiaodouzi@ubuntu:~$ git clone git@github.com:xiaodouzi0411/gitskills.git
Cloning into 'gitskills'...
Warning: Permanently added the RSA host key for IP address '192.30.255.113' to the list of known hosts.
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
```

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。

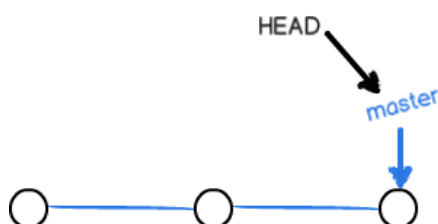
ps: 执行如下命令以创建一个本地仓库的克隆版本

九、分支管理

1. 创建与合并分支

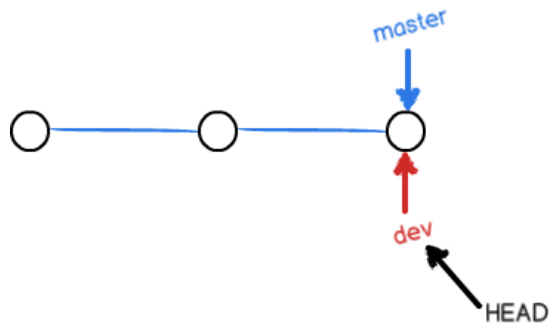
每个版本的 `commit id` 是 Git 将他们串成的一条时间线，这条时间线就是一个分支。截止到目前，只有一条时间线，在 Git 里，这个分支叫主分支，即 `master` 分支。`HEAD` 严格来说不是指向提交，而是指向 `master`，`master` 才指向提交的，所以 `HEAD` 指向的就是当前分支。

开始时，`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：



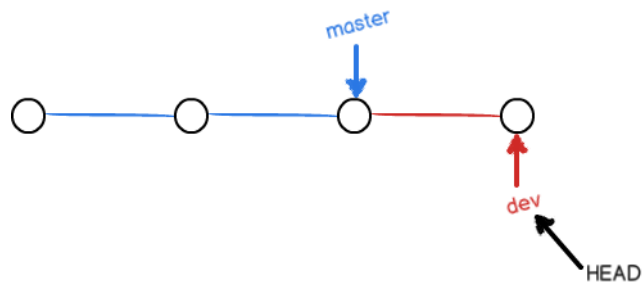
每次提交，`master` 分支都会向前移一步，这样随着提交，`master` 分支也越来越长：

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：

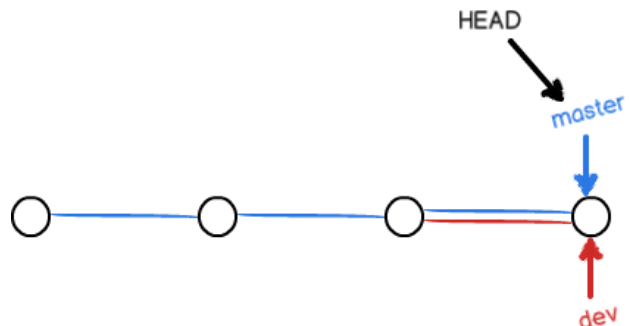


Git 创建一个分支很快，因为除了增加一个 dev 指针，改改 HEAD 的指向，工作区的文件都没有任何变化。

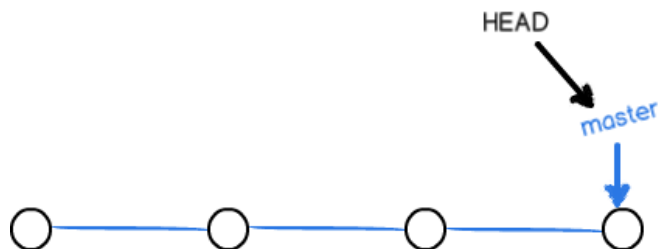
不过从现在开始，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后，dev 指针往前移动一步，而 master 指针不变：



当 dev 上的工作完成了，就可以把 dev 合并到 master 上。Git 合并的简单方法：直接把 master 指向 dev 的当前提交，就完成了合并。



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，就只剩下一条 master 分支：



首先，创建 dev 分支并切换到 dev 分支：

```
xiaodouzi@ubuntu:~/git_test$ git checkout -b dev
Switched to a new branch 'dev'
```

\$ git checkout -b dev 相当于两句： \$ git branch dev \$ git checkout dev

git branch 命令查看当前分支:

```
xiaodouzi@ubuntu:~/git_test$ git branch
* dev
master
```

***表示当前分支**

在 readme.txt 的最后添加一行;

Creating a new branch is quickk

然后提交;

git add readme.txt

git commit -m "branch test"

现在, dev 分支工作完成, 可以切换到 master 分支

\$ git checkout master

此时 readme.txt 中的内容还是未更新前, 需要进行分支合并

```
xiaodouzi@ubuntu:~/git_test$ git merge dev
Updating 2581c20..b6133c9
Fast-forward
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

合并完后, 可以放心的删除 dev 分支了:

```
xiaodouzi@ubuntu:~/git_test$ git branch -d dev
Deleted branch dev (was b6133c9).
```

此时, git branch 查看分支:

```
xiaodouzi@ubuntu:~/git_test$ git branch
* master
```

2. 解决冲突

发生冲突: 两个分支分别同步进行不同的修改, 合并时会发生冲突

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。

准备新的 feature1 分支:

\$ git checkout -b 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 "quick AND simple"

切换到 master 分支:

\$ git checkout master

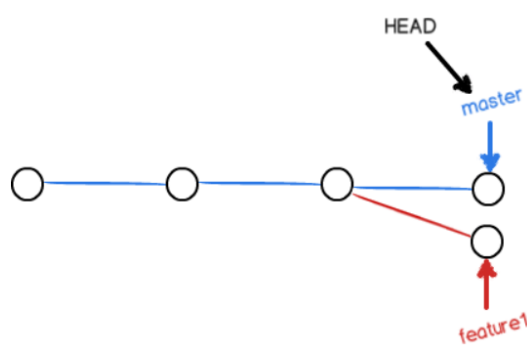
在 master 分支上把 readme.txt 文件的最后一行改为:

Creating a new branch is quick & simple

提交:

\$ git add readme.txt

现在 master 分支和 feature1 分支各自都分别有新的提交，如下



```
$ git merge feature1
```

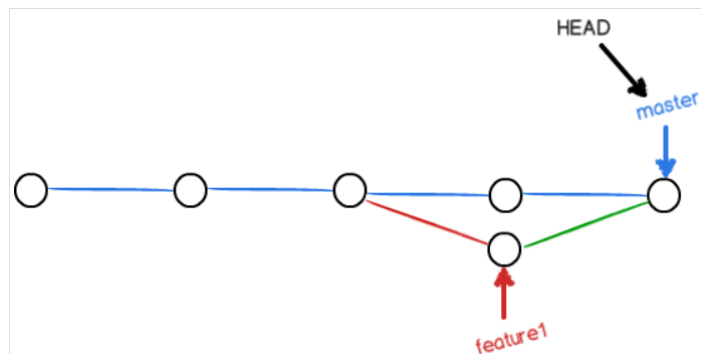
```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.

<<<<<< HEAD
Creating a new branch is quick & simple.
=====
Creating a new branch is quick AND simple.
>>>>>> feature1
```

Creating a new branch is quick and simple.

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

现在，master 和 feature1 分支变成了下图所示;



git log 可以看到分支合并情况：

```
$ git log --graph --pretty=oneline --abbrev-commit
* 59bc1cb conflict fixed
|\
| * 75a857c AND simple
* | 400b400 & simple
|/
* fec145a branch test
...
```

最后删除 feature1 分支：

```
$ git branch -d feature1
```

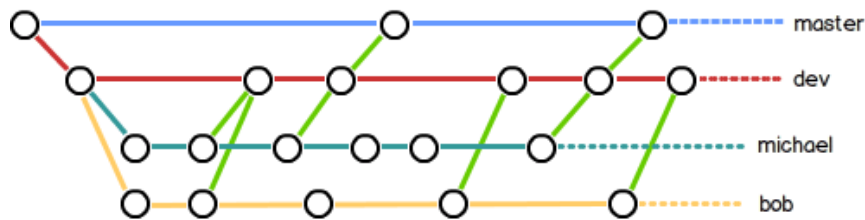
3. 分支管理策略

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。

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。

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。



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** 合并就看不出曾经做过合并

4. Bug 分支

情景：当接到一个修复代号 **101** 的 bug 任务时，很自然的想创建一个分支 **issue-101** 来修复它，但当前正在 **dev** 上进行的工作还未完成没有提交：【可添加场景编号】

Git 的 **stash** 功能，可以把当前的工作现场储藏起来，等以后恢复现场后继续工作：

```
$ git status
# On branch dev
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   hello.py
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   readme.txt
#
```

```
$ git stash
Saved working directory and index state WIP on dev: 6224937 add merge
HEAD is now at 6224937 add merge
```

再用 `git status` 查看工作区，就是干净的，因此可以放心地创建分支来修复 bug
首先确定在哪个分支上修复 bug，假定需要在 master 分支上修复，就从 master 创建临时分支：

```
$ git checkout master
$ git checkout -b issue-101
修改 readme.txt 后提交
$ git add readme.txt
$ git commit -m "fix bug 101"
```

修复完切换到 master 分支，并完成合并，最后删除 issue-101 分支：

```
$ git checkout master
$ git merge --no-ff -m "merge bug fix 101" issue-101
$ git branch -d issue-101
```

回到 dev 分支干活

```
$ git checkout dev
$ git status
```

恢复工作现场：

```
$ git stash list      //查看工作现场存放地址
$ git stash apply     //恢复
$ git stash drop      //删除
$ git stash pop       //恢复并删除
```

5. 推送分支：

Git 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：

```
$ git push origin dev
```

- 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。

抓取分支：

多人协作时，大家都会往 master 和 dev 分支上推送各自的修改。

当你的小伙伴从远程库 clone 时，默认他只能看到本地的 master 分支

你的小伙伴要在 dev 分支上开发，就必须创建远程 origin 的 dev 分支到本地，于是他用这个命令创建本地 dev 分支：

```
$ git checkout -b dev origin/dev
```

现在他就可以在 dev 上继续修改，然后把 dev 分支 push 到远程：

多人协作的工作模式通常是这样：

1. 首先，可以试图用 `git push origin branch-name` 推送自己的修改

```
$ git add hello.py
$ git commit -m "add coding: utf-8"
[dev bd6ae48] add coding: utf-8
1 file changed, 1 insertion(+)
$ git push origin dev
To git@github.com:michaelliao/learngit.git
! [rejected]        dev -> dev (non-fast-forward)
error: failed to push some refs to 'git@github.com:michaelliao/learngit.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

2. 如果推送失败，则因为远程分支比你的本地更新，需要先用 `git pull` 试图合并

```
$ git branch --set-upstream dev origin/dev 指定本地dev与远程origin/dev分支的链接
Branch dev set up to track remote branch dev from origin.
```

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

3. 如果合并有冲突，则解决冲突，并在本地提交
4. 没有冲突或者解决掉冲突后，再用 `git push origin branch-name` 推送就能成功

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

十、标签管理

1. 创建标签：

- 切换到需要打标签的分支上：

```
$ git branch
* dev
  master
$ git checkout master
Switched to branch 'master'
```

- 敲命令 `git tag` 就可以打一个新标签

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

- 查看所有标签

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

给之前的提交的版本号 commit id

eg: `$ git tag v0.9 6224937`

查看标签信息: `git show v0.9`

创建带有说明的标签 `-a` 指定标签名 `-m` 指定说明文字

```
$ git tag -a v0.1 -m "version 0.1 released" 3628164
```

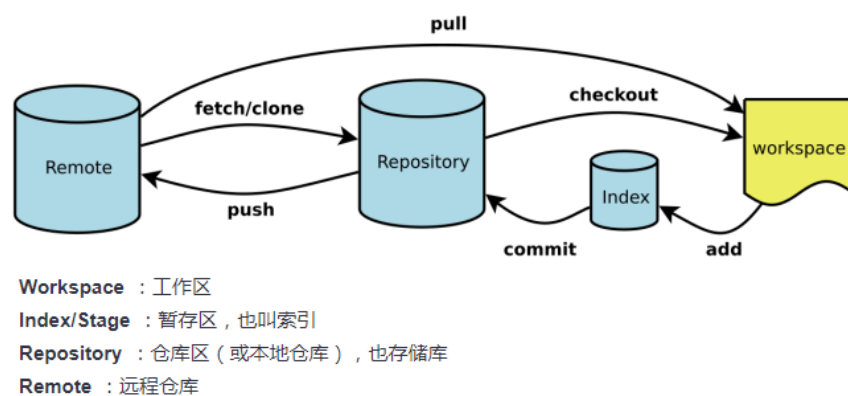
2. 操作标签

本地删除标签: `$ git tag -d v1.0`

推送标签到远程: `$ git push origin v1.0`

一次性推送全部本地标签: `$ git push origin --tags`

删除远程标签: 1. 本地删除 `$ git tag -d v1.0` 2. `git push origin :refs/tags/ v1.0`



`git fetch/git clone`:

<http://blog.csdn.net/u012575819/article/details/50553501>