

1. git 简介

1.1 产生历史

git 是目前世界上最先进的分布式版本控制系统。

Linus 在 1991 年创建了开源的 Linux，从此，Linux 系统不断发展，已经成为最大的服务器系统软件了。Linus 虽然创建了 Linux，但 Linux 的壮大是靠全世界热心的志愿者参与的，这么多人在世界各地为 Linux 编写代码，那 Linux 的代码是如何管理的呢？事实是，在 2002 年以前，世界各地的志愿者把源代码文件通过 diff 的方式发给 Linus，然后由 Linus 本人通过手工方式合并代码！你也许会想，为什么 Linus 不把 Linux 代码放到版本控制系统里呢？不是有 CVS、SVN 这些免费的版本控制系统吗？因为 Linus 坚定地反对 CVS 和 SVN，这些集中的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然比 CVS、SVN 好用，但那是付费的，和 Linux 的开源精神不符。不过，到了 2002 年，Linux 系统已经发展了十年了，代码库之大让 Linus 很难继续通过手工方式管理了，社区的弟兄们也对这种方式表达了强烈不满，于是 Linus 选择了一个商业的版本控制系统 BitKeeper，BitKeeper 的东家 BitMover 公司出于人道主义精神，授权 Linux 社区免费使用这个版本控制系统。安定团结的大好局面在 2005 年就被打破了，原因是 Linux 社区牛人聚集，不免沾染了一些梁山好汉的江湖习气。开发 Samba 的 Andrew 试图破解 BitKeeper 的协议（这么干的其实也不只他一个），被 BitMover 公司发现了（监控工作做得不错！），于是 BitMover 公司怒了，要收回 Linux 社区的使用权。Linus 可以向 BitMover 公司道

个歉，保证以后严格管教弟兄们，嗯，这是不可能的。实际情况是这样的：
Linus 花了两周时间自己用 c 写了一个分布式版本控制系统，这就是 Git！一
个月之内，Linux 系统的源码已经由 Git 管理了！牛是怎么定义的呢？大家可
以体会一下。Git 迅速成为最流行的分布式版本控制系统，尤其是 2008 年，
GitHub 网站上线了，它为开源项目免费提供 Git 存储，无数开源项目开始迁
移至 GitHub，包括 jQuery，PHP，Ruby 等等。历史就是这么偶然，如果不
是当年 BitMover 公司威胁 Linux 社区，可能现在我们就没有免费而超级好
用的 Git 了。

1.2 git 的两大特点

- 版本控制：可以解决多人同时开发的代码问题，也可以解决找回历史代码
的问题。
- 分布式：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。首先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。可以自
己搭建这台服务器，也可以使用 GitHub 网站。

2. 安装与配置

(1) 安装命令如下：

```
sudo apt-get install git

python@ubuntu:~$ sudo apt-get install git
[sudo] python 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
```

(2) 安装成功后，运行如下命令：

```
git  
python@ubuntu:~$ git  
usage: git [--version] [--help] [-C <path>] [-c name=value]  
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]  
          [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]  
          [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]  
          <command> [<args>]
```

这些是各种场合常见的 git 命令：

3. 创建一个版本库

(1) 新建一个目录 git_test，在 git_test 目录下创建一个版本库，命令如下：

```
git init  
python@ubuntu:~$ mkdir git_test  
python@ubuntu:~$ cd git_test/  
python@ubuntu:~/git_test$ ls -al  
总用量 8  
drwxrwxr-x  2 python python 4096 9月 18 13:01 .  
drwxr-xr-x 64 python python 4096 9月 18 13:01 ..  
python@ubuntu:~/git_test$ git init  
初始化空的 Git 仓库于 /home/python/git_test/.git/  
python@ubuntu:~/git_test$ ls -al  
总用量 12  
drwxrwxr-x  3 python python 4096 9月 18 13:02 .  
drwxr-xr-x 64 python python 4096 9月 18 13:01 ..  
drwxrwxr-x  7 python python 4096 9月 18 13:02 .git  
python@ubuntu:~/git_test$ █
```

可以看到在 git_test 目录下创建了一个.git 隐藏目录，这就是版本库目录。

4. 版本创建与回退

4.1 使用

(1) 在 git_test 目录下创建一个文件 code.txt，编辑内容如下：

```
python@ubuntu:~/git_test$ touch code.txt
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
```

(2) 使用如下两条命令可以创建一个版本:

```
git add code.txt
git commit -m '版本 1'

python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '版本1'
[master (根提交) 8b40219] [版本1]
 1 file changed, 1 insertion(+)
 create mode 100644 code.txt
```

(3) 使用如下命令可以查看版本记录:

```
git log

python@ubuntu:~/git_test$ git log
commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800
      | 
      版本1
python@ubuntu:~/git_test$ █
```

(4) 继续编辑 code.txt, 在里面增加一行。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
python@ubuntu:~/git_test$ █
```

(5) 使用如下命令再创建一个版本并查看版本记录:

```
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '版本2'
[master ba37943] 版本2
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$ git log
commit ba37943ec8b588b3c121f79028ce373cd57b7f1e
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:12:39 2017 +0800

    版本2
    I
commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
python@ubuntu:~/git_test$
```

(6) 现在若想回到某一个版本，可以使用如下命令：

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

其中 HEAD 表示当前最新版本，HEAD[^]表示当前版本的前一个版本，HEAD^{^^}表示当前版本的前两个版本，也可以使用 HEAD~1 表示当前版本的前一个版本，HEAD~100 表示当前版本的前 100 版本。

现在若觉得想回到版本 1，可以使用如下命令：

```
python@ubuntu:~/git_test$ git reset --hard HEAD^
HEAD 现在位于 8b40219 版本1
python@ubuntu:~/git_test$ git log
commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
    I
python@ubuntu:~/git_test$ cat code.txt
this is the first line
```

执行命令后使用 git log 查看版本记录，发现现在只能看到版本 1 的记录，cat code.txt 查看文件内容，现在只有一行，也就是第一个版本中 code.txt 的内容。

(7) 假如我们现在又想回到版本 2，这个时候怎么办？

可以使用如下命令：

```
git reset --hard 版本号
```

从上面可以看到版本 2 的版本号为：

```
python@ubuntu:~/git_test$ git log          版本编号
commit ba37943ec8b588b3c121f79028ce373cd57b7f1e
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:12:39 2017 +0800

    版本2

commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
```

(8) 在终端执行如下命令：

```
python@ubuntu:~/git_test$ git reset --hard ba37943e
HEAD 现在位于 ba37943 版本2
python@ubuntu:~/git_test$ git log
commit ba37943ec8b588b3c121f79028ce373cd57b7f1e
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:12:39 2017 +0800

    版本2

commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
python@ubuntu:~/git_test$ █
```

现在发现版本 2 有回来了。可以 cat code.txt 查看其里面的内容如下：

```
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
python@ubuntu:~/git_test$ █
```

(9) 假如说上面的终端已经关了改怎么回退版本。

我们在执行如下命令将版本回退到版本 1。

```
python@ubuntu:~/git_test$ git reset --hard HEAD^
HEAD 现在位于 8b40219 版本1
python@ubuntu:~/git_test$ git log
commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
python@ubuntu:~/git_test$
```

下面把终端关了，然后再打开终端，发现之前版本 2 的版本号看不到了。

那么怎么再回到版本 2 呢？git reflog 命令可以查看我们的操作记录。

```
git reflog

python@ubuntu:~/git_test$ git reflog
8b40219 HEAD@{0}: reset: moving to HEAD^
ba37943 HEAD@{1}: reset: moving to ba37943e
8b40219 HEAD@{2}: reset: moving to HEAD^
ba37943 HEAD@{3}: commit: 版本2
8b40219 HEAD@{4}: commit (initial): 版本1
python@ubuntu:~/git_test$
```

可以看到版本 2 的版本号，我们再使用如下命令进行版本回退，版本重新回到

了版本 2。

```
python@ubuntu:~/git_test$ git reset --hard ba37943
HEAD 现在位于 ba37943 版本2
python@ubuntu:~/git_test$ git log
commit ba37943ec8b588b3c121f79028ce373cd57b7f1e
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:12:39 2017 +0800

    版本2

commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
python@ubuntu:~/git_test$
```

4.2 工作区和暂存区

4.2.1 工作区(Working Directory)

电脑中的目录，比如我们的 `git_test`，就是一个工作区。

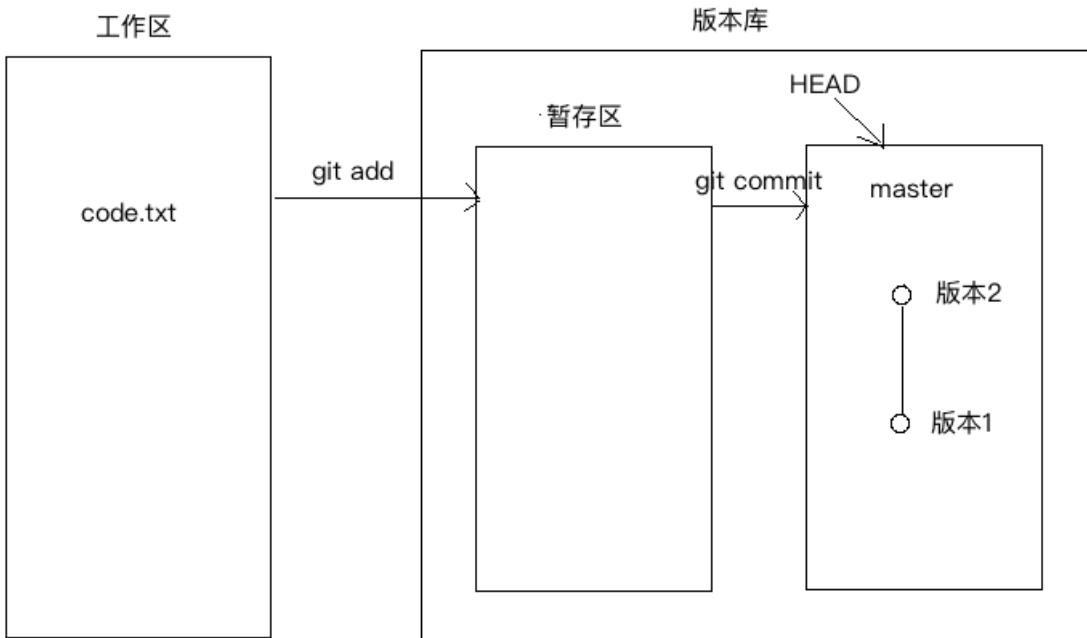
4.2.2 版本库(Repository)

工作区有一个隐藏目录`.git`，这个不是工作区，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`。

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。



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。

- (1) 下面在 `git_test` 目录下再创建一个文件 `code2.txt`，然后编辑内容
如下：

```
python@ubuntu:~/git_test$ touch code2.txt
python@ubuntu:~/git_test$ vi code2.txt
python@ubuntu:~/git_test$ cat code2.txt
the code2 first line
python@ubuntu:~/git_test$ █
```

- (2) 然后再次编辑 `code.txt` 内容，在其中加入一行，编辑后内容如下：

```
python@ubuntu:~/git_test$ vi code.tx
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
python@ubuntu:~/git_test$
```

(3) 使用如下命令查看当前工作树的状态:

```
git status
```

```
python@ubuntu:~/git_test$ git status
位于分支 master
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改:      code.txt

未跟踪的文件:
  (使用 "git add <文件>..." 以包含要提交的内容)

    code2.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
python@ubuntu:~/git_test$
```

上面提示我们 code.txt 被修改，而 code2.txt 没有被跟踪。

(4) 我们使用如下命令把 code.txt 和 code2.txt 加入到暂存区，然后再

执行 git status 命令，结果如下:

```
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git add code2.txt
python@ubuntu:~/git_test$ git status
位于分支 master
要提交的变更:
  (使用 "git reset HEAD <文件>..." 以取消暂存)

    修改:      code.txt
    新文件:    code2.txt

python@ubuntu:~/git_test$
```

所有 git add 命令是把所有提交的修改存放到暂存区。

- (5) 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
创建一个版本。

```
python@ubuntu:~/git_test$ git commit -m '版本3'
[master b89bb1a] 版本3
 2 files changed, 2 insertions(+)
   create mode 100644 code2.txt
python@ubuntu:~/git_test$ git log
commit b89bb1abc733ed09d41c5c43b8dd14b54fde5bc8
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 19:22:26 2017 +0800

    版本3

commit ba37943ec8b588b3c121f79028ce373cd57b7f1e
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:12:39 2017 +0800

    版本2

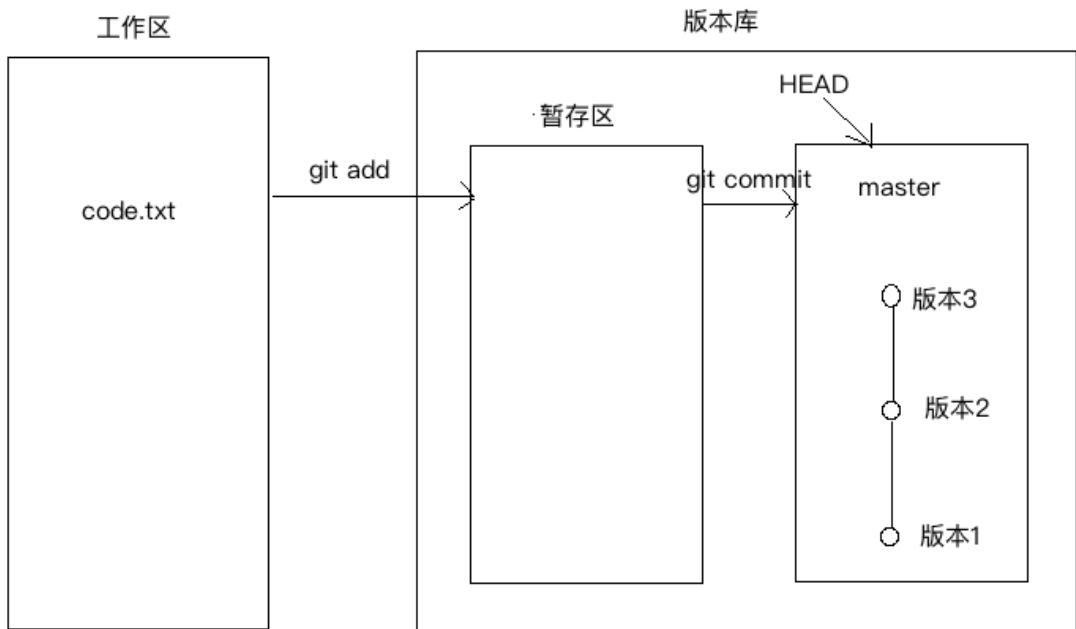
commit 8b40219e98faf639ba7583d37908abdb55751f6d
Author: smartliit <smartli_it@163.com>
Date:   Mon Sep 18 13:07:22 2017 +0800

    版本1
python@ubuntu:~/git_test$
```

- (6) 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。执行如下命令可以发现：

```
python@ubuntu:~/git_test$ git status
位于分支 master
无文件要提交，干净的工作区
python@ubuntu:~/git_test$
```

现在我们的版本库变成了这样：



4.3 管理修改

git 管理的文件的修改，它只会提交暂存区的修改来创建版本。

- (1) 编辑 code.txt，并使用 git add 命令将其添加到暂存区中。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
I
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
```

- (2) 继续编辑 code.txt，并在其中添加一行。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
this is the new line
python@ubuntu:~/git_test$
```

(3) git commit 创建一个版本，并使用 git status 查看，发现第二次修改 code.txt 内容之后，并没有将其添加的工作区，所以创建版本的时候并没有被提交。

```
python@ubuntu:~/git_test$ git commit -m '版本4'
[master d6cb90d] 版本4
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$ git status
位于分支 master
尚未暂存以备提交的变更：
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改：      code.txt

修改尚未加入提交（使用 "git add" 和/或 "git commit -a"）
python@ubuntu:~/git_test$
```

4.4 撤销修改

(1) 继续上面的操作，提示我们可以使用 git checkout -- <文件> 来丢弃工作区的改动。执行如下命令，发现工作区干净了，第二次的改动内容也没了。

```
python@ubuntu:~/git_test$ git checkout -- code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
python@ubuntu:~/git_test$ git status
位于分支 master
无文件要提交，干净的工作区
python@ubuntu:~/git_test$
```

(2) 我们继续编辑 code.txt，并在其中添加如下内容，并将其添加的暂存区。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
the new line
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git status
位于分支 master
要提交的变更：
    (使用 "git reset HEAD <文件>..." 以取消暂存)

修改：      code.txt

python@ubuntu:~/git_test$
```

(3) 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，重新放回工作区。

```
python@ubuntu:~/git_test$ git reset HEAD code.txt
重置后取消暂存的变更:
M       code.txt
python@ubuntu:~/git_test$ git status
位于分支 master
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改:      code.txt
修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
python@ubuntu:~/git_test$
```

(4) 现在若想丢弃 code.txt 的修改, 执行如下命令即可。

```
python@ubuntu:~/git_test$ git checkout -- code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
python@ubuntu:~/git_test$ git status
位于分支 master
无文件要提交, 干净的工作区
python@ubuntu:~/git_test$
```

现在, 如果你不但改错了东西, 还从暂存区提交到了版本库, 则需要进行版本回退。

小结:

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

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

场景 3: 已经提交了不合适的修改到版本库时, 想要撤销本次提交, 参考版本回退一节。

4.5 对比文件的不同

对比工作区和某个版本中文件的不同：

- (1) 继续编辑文件 code.txt，在其中添加一行内容。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
the new line
python@ubuntu:~/git_test$
```

- (2) 现在要对比工作区中 code.txt 和 HEAD 版本中 code.txt 的不同。使用如下命令：

```
python@ubuntu:~/git_test$ git diff HEAD -- code.txt
diff --git a/code.txt b/code.txt
index 66f9219..324317f 100644
@@ -2,3 +2,4 @@ this is the first line
      this is the second line
      this is the third line      工作区code.txt比
      this is the forth line     HEAD版本code.txt多
+the new line
python@ubuntu:~/git_test$
```

- (3) 使用如下命令丢弃工作区的改动。

```
python@ubuntu:~/git_test$ git checkout -- code.txt
python@ubuntu:~/git_test$ git status
位于分支 master
无文件要提交，干净的工作区
```

对比两个版本间文件的不同：

- (1) 现在要对比 HEAD 和 HEAD^ 版本中 code.txt 的不同，使用如下命令：

```
python@ubuntu:~/git_test$ git diff HEAD HEAD^ -- code.txt
diff --git a/code.txt b/code.txt
index 66f9219..01e1274 100644
--- a/code.txt      -代表HEAD版本      +代表HEAD^版本中
+++ b/code.txt      code.txt内容      code.txt内容
@@ -1,4 +1,3 @@
 this is the first line
 this is the second line      HEAD版本code.txt比
 this is the third line      HEAD^版本code.txt
+this is the forth line
python@ubuntu:~/git_test$ 多了一行
```

4.6 删除文件

(1) 我们把目录中的 code2.txt 删除。

```
python@ubuntu:~/git_test$ rm code2.txt
python@ubuntu:~/git_test$
```

这个时候，git 知道删除了文件，因此，工作区和版本库就不一致了，git status 命令会立刻提示哪些文件被删除了。

```
python@ubuntu:~/git_test$ git status
位于分支 master
尚未暂存以备提交的变更：
  (使用 "git add/rm <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)
```

删除： code2.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")

```
python@ubuntu:~/git_test$
```

(2) 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：

```
python@ubuntu:~/git_test$ git rm code2.txt
rm 'code2.txt'
python@ubuntu:~/git_test$ git status
位于分支 master
要提交的变更：
  (使用 "git reset HEAD <文件>..." 以取消暂存)

    删除：      code2.txt

python@ubuntu:~/git_test$ git commit -m '删除文件cod2.txt'
[master 48aeb0c] 删除文件cod2.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 code2.txt
python@ubuntu:~/git_test$ git status
位于分支 master
无文件要提交，干净的工作区
python@ubuntu:~/git_test$
```

另一种情况是删错了，可以直接使用 `git checkout - code2.txt`, 这样文件 `code2.txt` 又回来了。

小结：

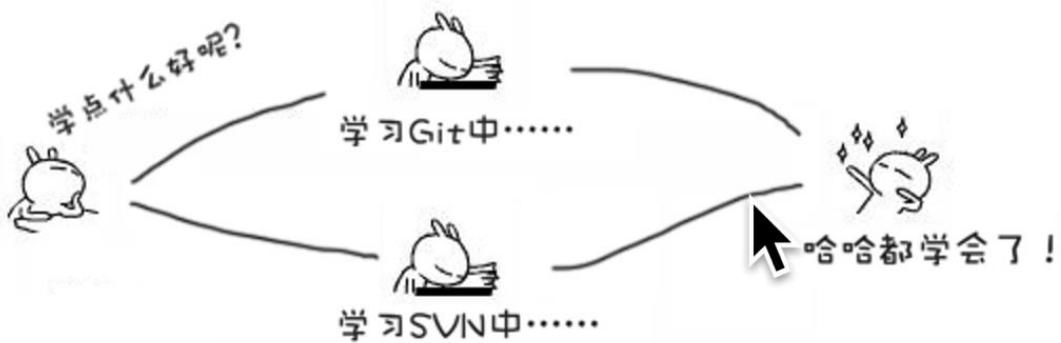
命令 `git rm` 用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失**最近一次提交后你修改的内容**。

5. 分支管理

5.1 概念

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！



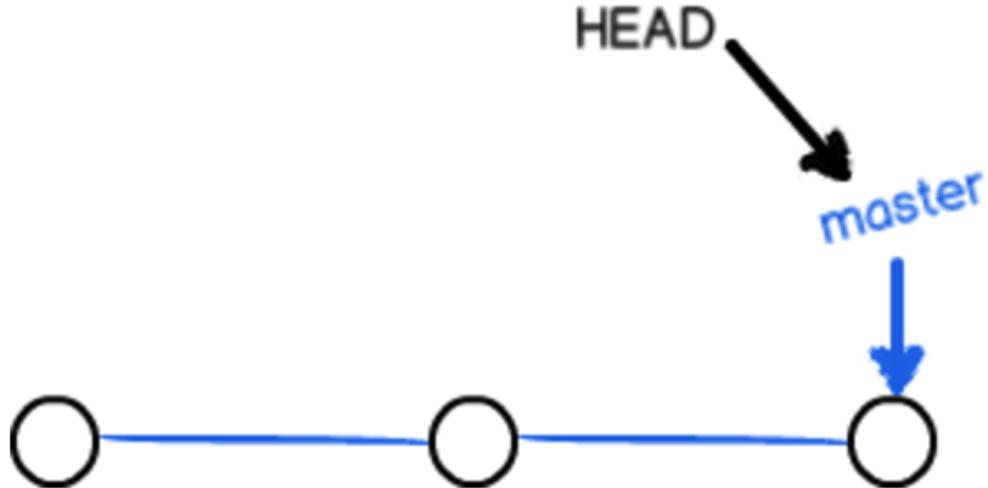
分支在实际中有什么用呢？假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了 50% 的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

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。

5.2 创建与合并分支

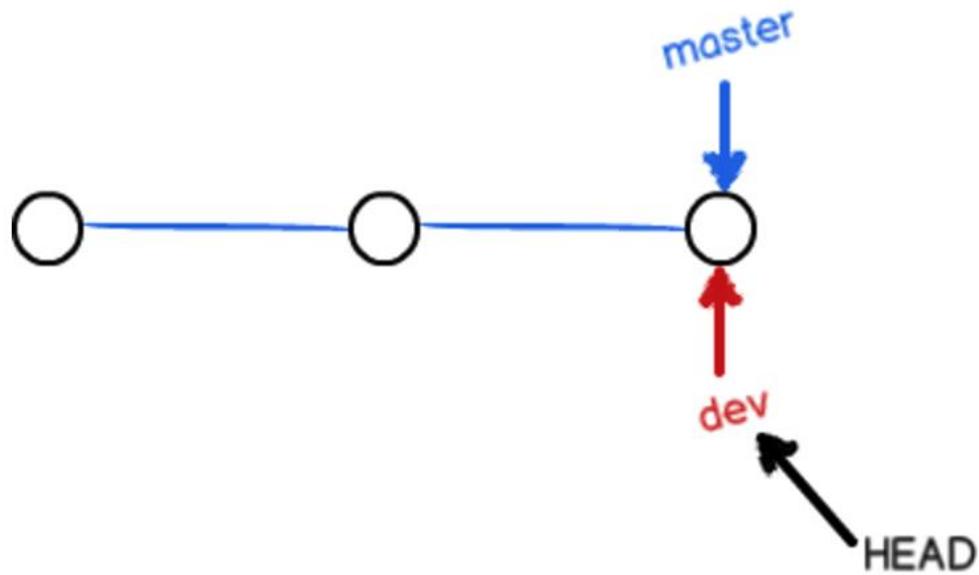
git 把我们之前每次提交的版本串成一条时间线，这条时间线就是一个分支。截止到目前只有一条时间线，在 git 里，这个分支叫**主分支**，即 `master 分支`。HEAD 严格来说不是指向提交，而是指向 `master`，`master` 才是指向提交的，所以，HEAD 指向的就是当前分支。

(1) 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：



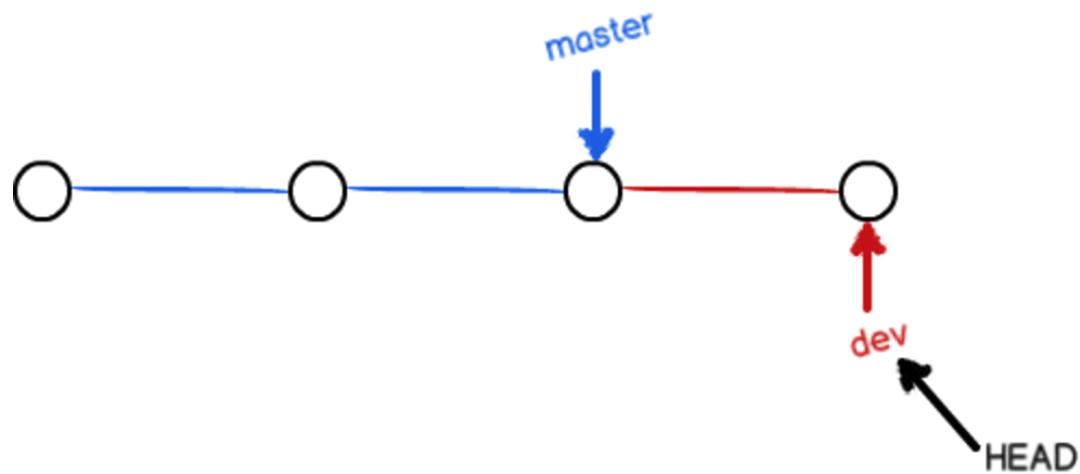
每次提交，`master` 分支都会向前移动一步，这样，随着你不断提交，`master` 分支的线也越来越长。

(2) 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：

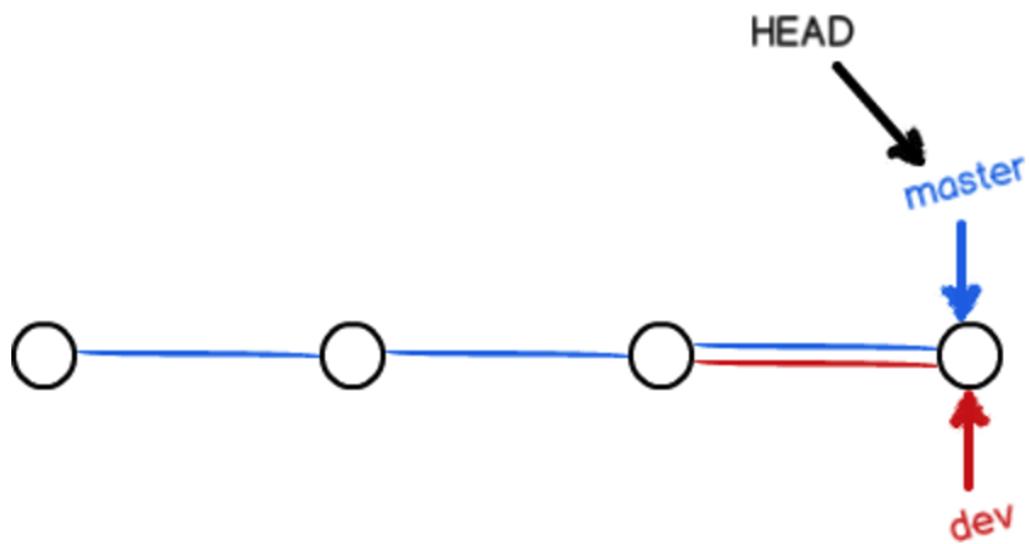


git 创建一个分支很快，因为除了增加一个 `dev` 指针，改变 `HEAD` 的指向，工作区的文件都没有任何变化。

(3) 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：

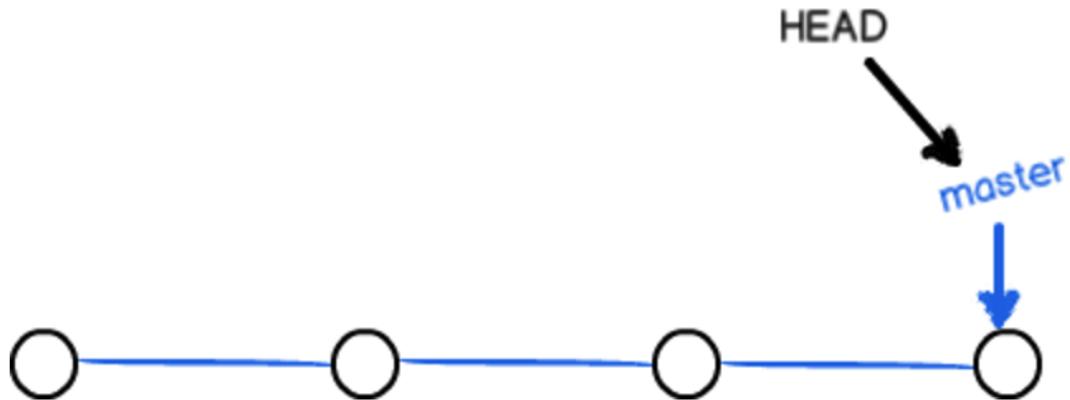


(4) 假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。git 怎么合并呢？最简单的方法，就是直接把 master 指向 dev 的当前提交，就完成了合并：



git 合并分支也很快，就改改指针，工作区内容也不变。

(5) 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：



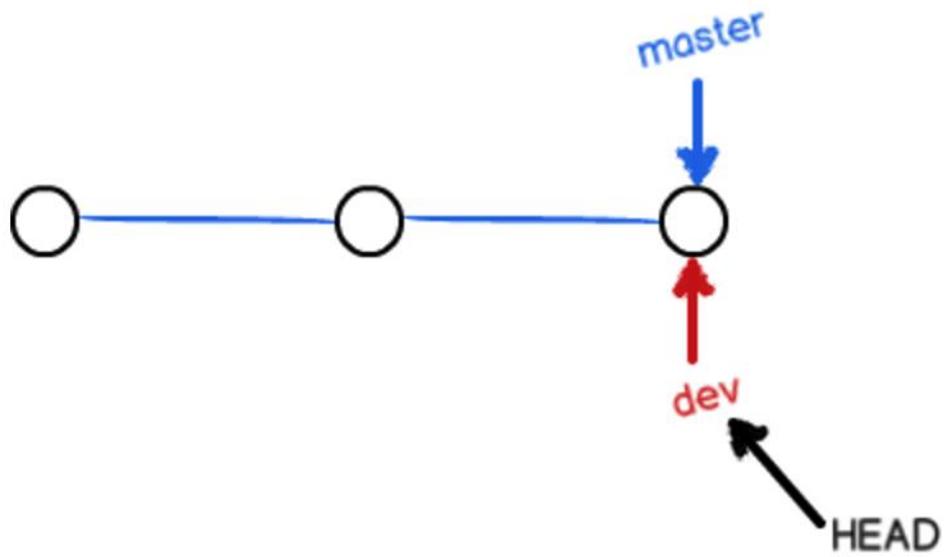
案例：

(1) 执行如下命令可以查看当前有几个分支并且看到在哪个分支下工作。

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

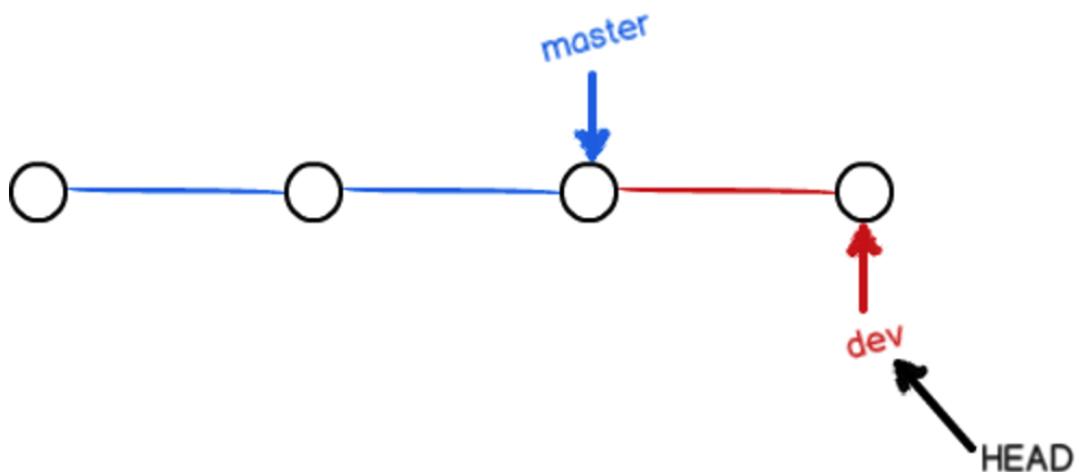
(2) 下面创建一个分支 dev 并切换到其上进行工作。

```
python@ubuntu:~/git_test$ git checkout -b dev
切换到一个新分支 'dev'
python@ubuntu:~/git_test$ git branch
* dev
  master
```



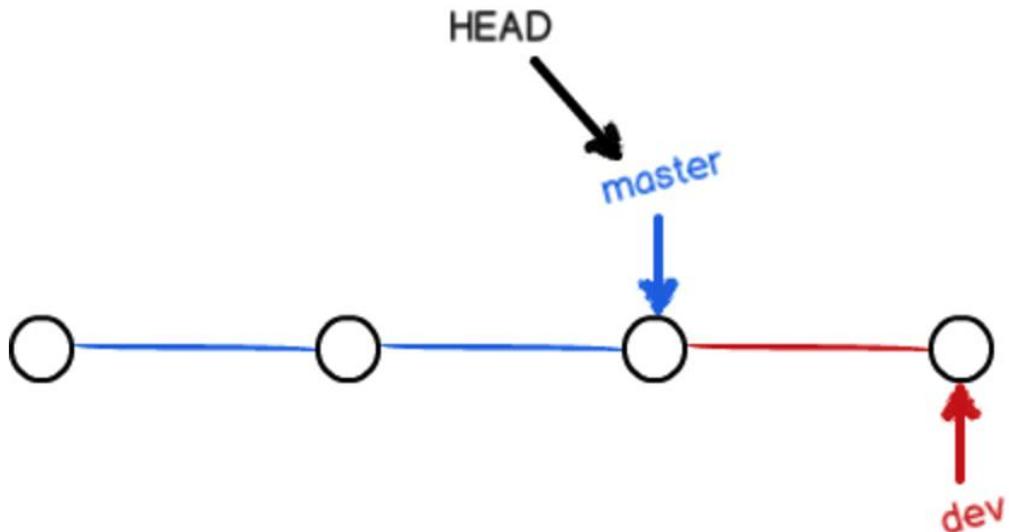
(3) 下面我们修改 code.txt 内容，在里面添加一行，并进行提交。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m 'dev分支提交'
[dev b031cbf] dev分支提交
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```



(4) dev 分支的工作完成，我们就可以切换回 master 分支：

```
python@ubuntu:~/git_test$ git checkout master
切换到分支 'master'
python@ubuntu:~/git_test$ git branch
  dev
* master
python@ubuntu:~/git_test$
```

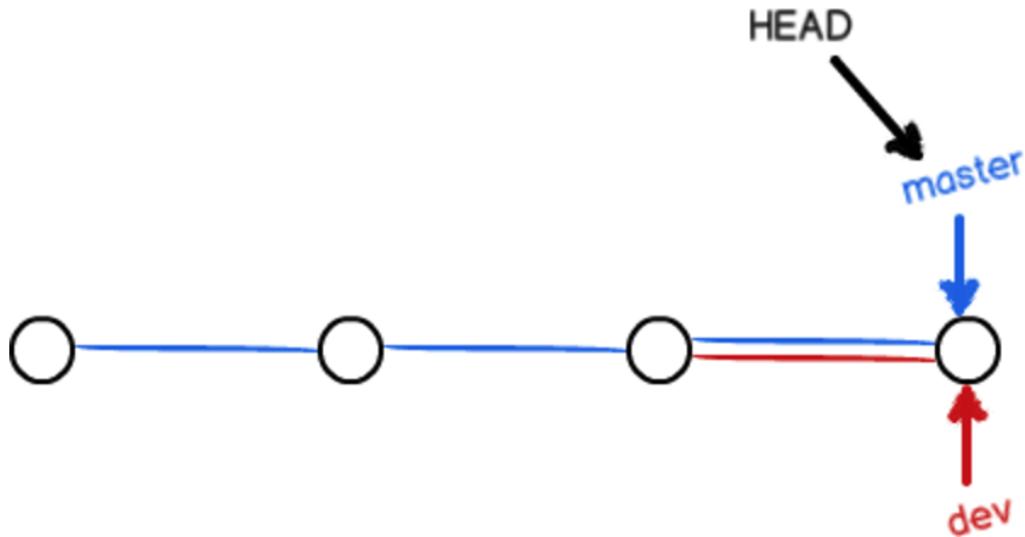


查看 code.txt，发现添加的内容没有了。因为那个提交是在 dev 分支上，而 master 分支此刻的提交点并没有变：

(5) 现在，我们把 dev 分支的工作成果合并到 master 分支上：

```
python@ubuntu:~/git_test$ git merge dev
更新 48aeb0c..b031cbf
Fast-forward
 code.txt | 1 +
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```

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

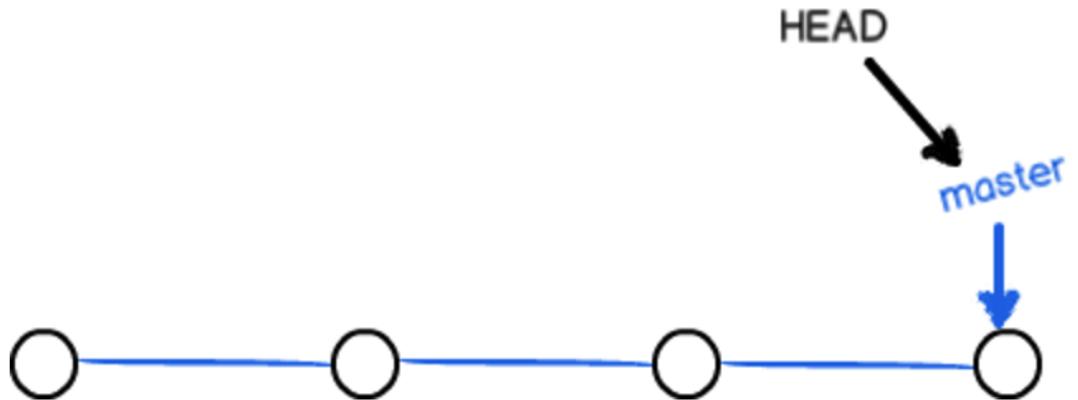


注意到上面的 `Fast-forward` 信息，Git 告诉我们，这次合并是“快进模式”，也就是直接把 `master` 指向 `dev` 的当前提交，所以合并速度非常快。

```
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
python@ubuntu:~/git_test$
```

(6) 合并完成后，就可以放心地删除 `dev` 分支了，删除后，查看 `branch`，就只剩下 `master` 分支了。

```
python@ubuntu:~/git_test$ git branch -d dev
已删除分支 dev (曾为 b031cbf)。
python@ubuntu:~/git_test$ git branch
* master
python@ubuntu:~/git_test$
```



小结：

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>`

5.3 解决冲突

合并分支往往也不是一帆风顺的。

(1) 再创建一个新分支 dev。

```
python@ubuntu:~/git_test$ git checkout -b dev
切换到一个新分支 'dev'
python@ubuntu:~/git_test$
```

(2) 修改 code.txt 内容，并进行提交。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
add one more line
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m 'dev分支提交2'
[dev 70d52e0] dev分支提交2
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```

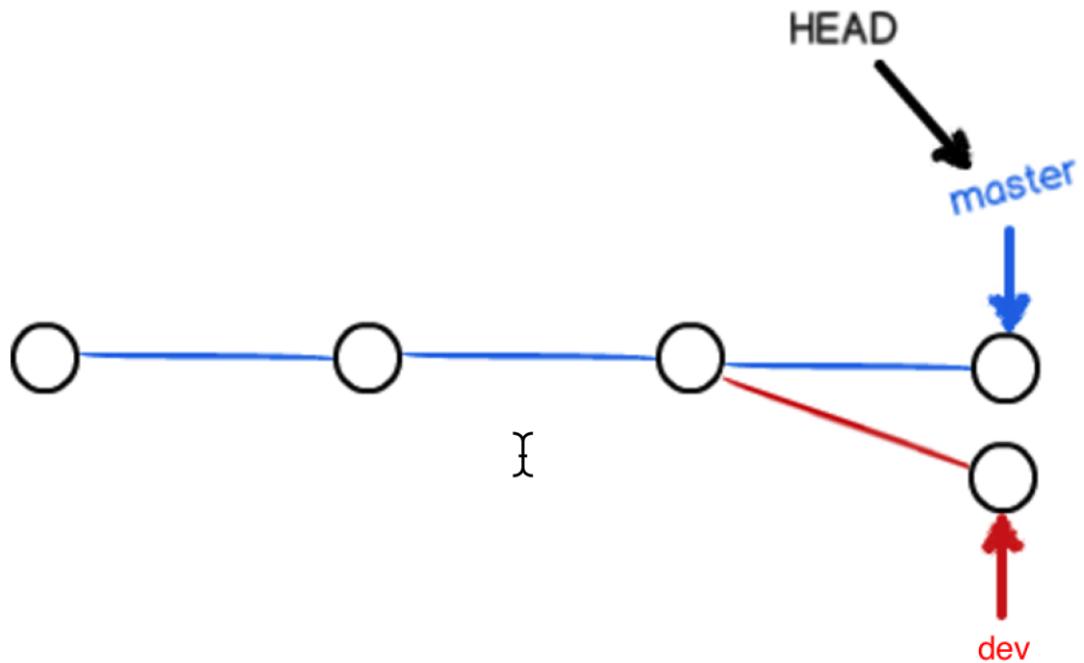
(3) 切换回 master 分支。

```
python@ubuntu:~/git_test$ git checkout master
切换到分支 'master'
python@ubuntu:~/git_test$
```

(4) 在 master 的 code.txt 添加一行内容并进行提交。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
one more line in master
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m 'master提交'
[master b8893cc] master提交
 1 file changed, 1 insertion(+)
python@ubuntu:~/git test$
```

现在，`master` 分支和 `dev` 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，但这种合并就可能不会有冲突。

(5) 执行如下命令尝试将 dev 分支合并到 master 分支上来。

```
python@ubuntu:~/git_test$ git merge dev
自动合并 code.txt
冲突 (内容)： 合并冲突于 code.txt
自动合并失败，修正冲突然后提交修正的结果。
python@ubuntu:~/git_test$
```

git 告诉我们，code.txt 文件存在冲突，必须手动解决冲突后再提交。

(6) git status 也可以告诉我们冲突的文件：

```
python@ubuntu:~/git_test$ git status
位于分支 master
您有尚未合并的路径。
  (解决冲突并运行 "git commit")

未合并的路径：
  (使用 "git add <文件>..." 标记解决方案)

    双方修改： code.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
python@ubuntu:~/git_test$
```

(7) 查看 code.txt 的内容。

```
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
<<<<<< HEAD
one more line in master
=====
add one more line
>>>>> dev
python@ubuntu:~/git_test$
```

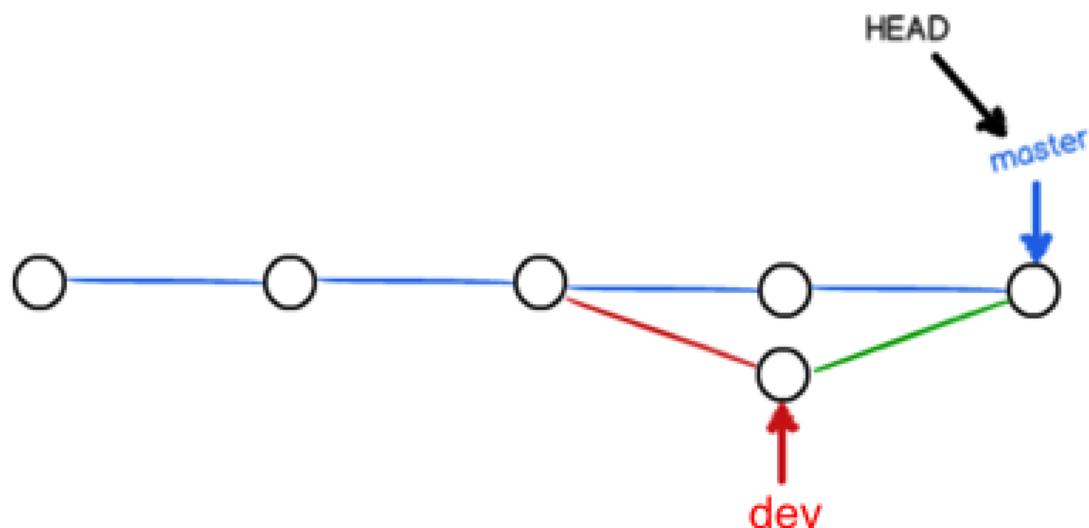
(8) git 用<<<<<, =====, >>>>>标记出不同分支的内容，我们修改如下后保存：

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
one more line in master
add one more line
python@ubuntu:~/git_test$
```

(9) 再提交。

```
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '解决冲突'
[master 1950eff] 解决冲突
python@ubuntu:~/git_test$
```

(10) 现在，master 分支和 dev 分支变成了下图所示：



(11) 用带参数的 git log 也可以看到分支的合并情况：

```
python@ubuntu:~/git_test$ git log --graph --pretty=oneline
*   1950effb646fa132d85b083409914466996a8707 解决冲突
|\ \
| * 70d52e021ddb1a398fd5e78cee1b710d3d235c4 dev分支提交2
| * | b8893cc7bcda54eff14aee0d1560834180eabf3c master提交
|| \
* b031cbfedfe137804bd15527498e39b53b1a59a3 dev分支提交
* 48aeb0c4fd4a7f5b5c4fdd6fd017388589c74aa1 删除文件cod2.txt
* d6cb90d1df0f01fd7d30c7f38a832ed83810982 [ 版本4
* b89bb1abc733ed09d41c5c43b8dd14b54fde5bc8 版本3
* ba37943ec8b588b3c121f79028ce373cd57b7f1e 版本2
* 8b40219e98faf639ba7583d37908abdb55751f6d 版本1
python@ubuntu:~/git_test$
```

(12) 最后工作完成，可以删除 dev 分支。

```
python@ubuntu:~/git_test$ git branch -d dev
已删除分支 dev (曾为 70d52e0)。
python@ubuntu:~/git_test$ git branch
* master
python@ubuntu:~/git_test$
```

5.4 分支管理策略

通常，合并分支时，如果可能，git 会用 fast forward 模式，但是有些快速合并不能成而且合并时没有冲突，这个时候会合并之后并做一次新的提交。但这种模式下，删除分支后，会丢掉分支信息。

(1) 创建切换到 dev 分支下。

```
python@ubuntu:~/git_test$ git checkout -b dev
切换到一个新分支 'dev'
```

(2) 新建一个文件 code3.txt 编辑内容如下，并提交一个 commit。

```
python@ubuntu:~/git_test$ git commit -m '创建文件code3.txt'
[dev 251eff3] 创建文件code3.txt
 1 file changed, 1 insertion(+)
  create mode 100644 code3.txt
python@ubuntu:~/git_test$
```

(3) 切换回 master 分支，编辑 code.txt 并进行一个提交。

```
CREATE MODE 100644 code.txt
python@ubuntu:~/git_test$ git checkout master
切换到分支 'master'
```

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '添加新行'
[master 2325000] 添加新行
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```

(4) 合并 dev 分支的内容到 master 分支。

```
1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$ git merge dev
```

(5) 出现如下提示，这是因为这次不能进行快速合并，所以 git 提示输入合并说明信息，输入之后合并内容之后 git 会自动创建一次新的提交。

```
GNU nano 2.5.3      文件: /home/python/git_test/.git/MERGE_MSG
Merge branch 'dev'

# 请输入一个提交信息以解释此合并的必要性，尤其是将一个更新后的上游分支
# 合并到主题分支。
#
# 以 '#' 开头的行将被忽略，而且空提交说明将会终止提交。

[ 已读取6行 ]
```

^G 求助 ^O Write Out ^W 搜索 ^K 剪切文字 ^J 对齐 ^C 游标位置
^X 离开 ^R 读档 ^\ 替换 ^U Uncut Text ^T 拼写检查 ^_ 跳行

```
GNU nano 2.5.3      文件: /home/python/git_test/.git/MERGE_MSG      已更改
合并分支dev

# 请输入一个提交信息以解释此合并的必要性，尤其是将一个更新后的上游分支
# 合并到主题分支。
#
# 以 '#' 开头的行将被忽略，而且空提交说明将会终止提交。



^G 求助      ^O Write Out  ^W 搜索      ^K 剪切文字  ^J 对齐      ^C 游标位置
^X 离开      ^R 读档      ^\ 替换      ^U Uncut Text  ^T 拼写检查  ^L 跳行
```

(6) 使用分支命令查看分支信息。

```
python@ubuntu:~/git_test$ git log --pretty=oneline --graph
*   270e1f6797df1c09a6f2d0610a628acfdce3794b 合并dev分支
|\ \
| * a0209a3be12792ad097d9caaad0f6122be8844f2 创建文件code3.txt
| * | 3875348dadf3cea18632c46c3581aafbd0696804 添加新行
| |
| * 7d7dabb6e95dbe296476af46ae564257ee2a530a 解决冲突
| \
| * d1445e739218d44819e32e190b50e95bcd868268 dev分支提交?
| * | 49f09b06d91109194de16cc30952e36f4ce35ba2 master提交
| /
* ef58df8070a0071eea3bdb0336365a086e8203c6 dev分支提交
* 630aa12c42e87a420c1716e5f12b448cb2b7b684 删除文件code2.txt
* 5fc405a1dba8662449cc4f4c44adc5918193efc 版本4
* 29ca1f22d349f8427efdaa804c18b03c02cc4d58 版本3
* 51e2efeda548a395fdb18b6be5c346d276b86f1e 版本2
* a3edeb83649f5b254c005a1de3b636fda29ff18b 版本1
python@ubuntu:~/git_test$
```

(7) 删除 dev 分支。

```
python@ubuntu:~/git_test$ git branch -d dev
已删除分支 dev (曾为 a0209a3) 。
python@ubuntu:~/git_test$
```

如果要强制禁用 fast forward 模式，git 就会在 merge 时生成一个新的 commit，这样，从分支历史上就可以看出分支信息。

(1) 创建并切换到 dev 分支。

```
python@ubuntu:~/git_test$ git checkout -b dev
切换到一个新分支 'dev'
python@ubuntu:~/git_test$
```

(2) 修改 code.txt 内容，并提交一个 commit。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '添加新行'
[master 358278c] 添加新行
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```

(3) 切换回 master 分支。

```
python@ubuntu:~/git_test$ git checkout master
M       code.txt
切换到分支 'master'
python@ubuntu:~/git_test$
```

(4) 准备合并 dev 分支，请注意--no-ff 参数，表示禁用 Fast forward:

```
python@ubuntu:~/git_test$ git merge --no-ff -m '禁用fast-forward合并' dev
Merge made by the 'recursive' strategy.
 code.txt | 1 +
 1 file changed, 1 insertion(+)
python@ubuntu:~/git_test$
```

因为本次合并要创建一个新的 commit，所以加上-m 参数，把 commit 描述写进去。

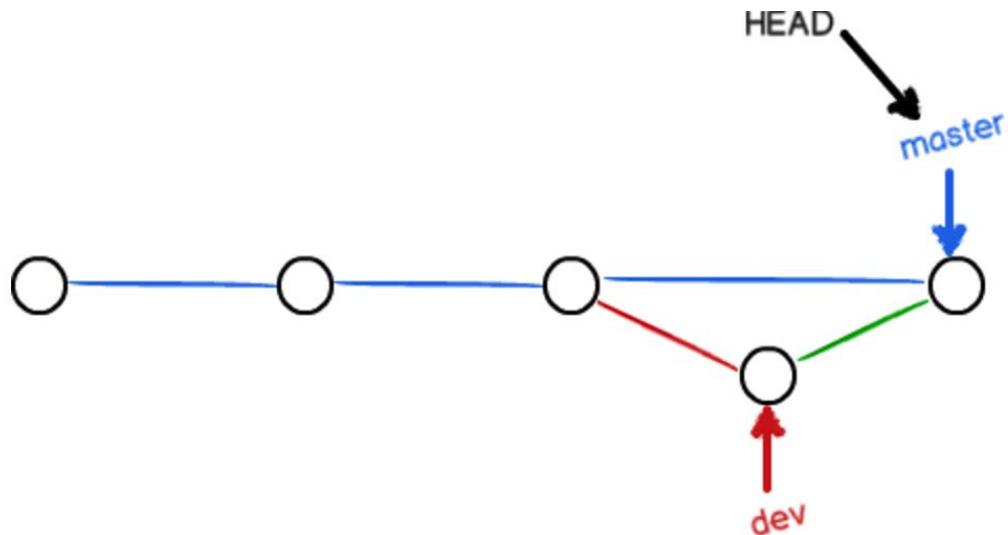
(5) 合并后，我们用 git log 看看分支历史：

可以看到，不使用 Fast forward 模式，merge 后就像这样：

```

python@ubuntu:~/git_test$ git log --pretty=oneline --graph
*   f1f947dfc1fe1d7e078d8f67144da1c94857c60b 禁用fast-forward合并
|\ \
| * 2f997e9be6bfa377cf17bf855ceb45d628bf13c6 添加新行
| / \
| * 270e1f6797df1c09a6f2d0610a628acfde3794b 合并dev分支
|\ \
| * a0209a3be12792ad097d9caaad0f6122be8844f2 创建文件code3.txt
| * 3875348dadf3cea18632c46c3581aafbd0696804 添加新行
| / \
| * 7d7dabb6e95dbe296476af46ae564257ee2a530a 解决冲突
|\ \
| * d1445e739218d44819e32e190b50e95bcd868268 dev分支提交2
| * 49f09b06d91109194de16cc30952e36f4ce35ba2 master提交
| / \
| * ef58df8070a0071eea3bdb0336365a036e8203c6 dev分支提交
| * 630aa12c42e87a420c1716e5f12b448cb2b7b684 删除文件code2.txt
| * 5fc405a1ddba8662449cc4f4c44adc5918193efc 版本4
| * 29ca1f22d349f8427efdaa804c18b03c02cc4d58 版本3
| * 51e2efeda548a395fdb18b6be5c346d276b86f1e 版本2
| * a3edeb83649f5b254c005a1de3b636fda29ff18b 版本1
python@ubuntu:~/git_test$

```



5.5 Bug 分支

软件开发中，bug 就像家常便饭一样。有了 bug 就需要修复，在 git 中，由于分支是如此的强大，所以，每个 bug 都可以通过一个新的临时分支来修复，修复后，合并分支，然后将临时分支删除。

(1) 当你接到一个修复一个代号 001 的 bug 的任务时，很自然地，你想创建一个分支 bug-001 来修复它，但是，等等，当前正在 dev 上进行的工作还没有提交：

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ git status
位于分支 dev
尚未暂存以备提交的变更：
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改：      code.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
python@ubuntu:~/git_test$
```

并不是你不想提交，而是工作只进行到一半，还没法提交，预计完成还需 1 天时间。但是，必须在两个小时内修复该 bug，怎么办？

(2) git 还提供了一个 stash 功能，可以把当前工作现场“储藏”起来，等以后恢复现场后继续工作：

```
python@ubuntu:~/git_test$ git stash
Saved working directory and index state WIP on dev: ae35f2d 禁用fast-forward合并
HEAD 现在位于 ae35f2d 禁用fast-forward合并
python@ubuntu:~/git_test$
```

(3) 首先确定要在哪个分支上修复 bug，假定需要在 master 分支上修复，就从 master 创建临时分支：

```
python@ubuntu:~/git_test$ git checkout master
切换到分支 'master'
python@ubuntu:~/git_test$ git checkout [b bug-001
切换到一个新分支 'bug-001'
python@ubuntu:~/git_test$
```

(4) 现在修复 bug，把 the new line 删掉，然后提交。

```
python@ubuntu:~/git_test$ vi code.txt
python@ubuntu:~/git_test$ cat code.txt
this is the first line
this is the second line
this is the third line
this is the forth line
add one line
one more line in master
add one more line
new line
python@ubuntu:~/git_test$ git add code.txt
python@ubuntu:~/git_test$ git commit -m '修复bug-001'
[bug-001 d63018d] 修复bug-001
 1 file changed, 1 deletion(-)
python@ubuntu:~/git_test$
```

(5) 修复完成后，切换到 master 分支，并完成合并，最后删除 bug-001 分支。

```
python@ubuntu:~/git_test$ git checkout master
切换到分支 'master'
python@ubuntu:~/git_test$ git merge --no-ff -m '修复bug-001' bug-001
Merge made by the 'recursive' strategy.
 code.txt | 1 -
 1 file changed, 1 deletion(-)
python@ubuntu:~/git_test$ git branch -d bug-001
已删除分支 bug-001 (曾为 ac73fec)。
python@ubuntu:~/git_test$
```

(6) 现在 bug-001 修复完成，是时候接着回到 dev 分支干活了！

```
python@ubuntu:~/git_test$ git checkout dev
切换到分支 'dev'
python@ubuntu:~/git_test$ git status
位于分支 dev
无文件要提交，干净的工作区
python@ubuntu:~/git_test$
```

(7) 工作区是干净的，刚才的工作现场存到哪去了？用 git stash list 命令看看：

```
python@ubuntu:~/git_test$ git stash list
stash@{0}: WIP on dev: ae35f2d 禁用fast-forward并
python@ubuntu:~/git_test$
```

工作现场还在，git 把 stash 内容存在某个地方了，但是需要恢复一下。

```

python@ubuntu:~/git_test$ git stash pop
位于分支 dev
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改:      code.txt

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
丢弃了 refs/stash@{0} (f2b6457aaf479c562013c26b5c3570ff97d6
python@ubuntu:~/git_test$ git status
位于分支 dev
尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git checkout -- <文件>..." 丢弃工作区的改动)

    修改:      code.txt

I
修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
python@ubuntu:~/git_test$ 

```

小结:

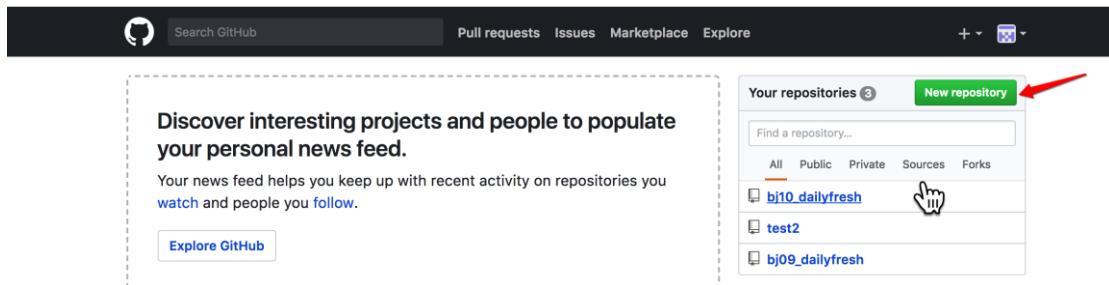
修复 bug 时，我们会通过创建新的 bug 分支进行修复，然后合并，最后删除；

当手头工作没有完成时，先把工作现场 `git stash` 一下，然后去修复 bug，修复后，再 `git stash pop`，恢复工作现场。

6. 使用 github

6.1 创建仓库

(1) 注册 github 账户，登录后，点击 "New repository"



(2) 在新页面中，输入项目的名称，勾选'readme.md'，点击'create repository'

Create a new repository

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

Owner / Repository name 

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

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: Python | Add a license: None | ⓘ

Create repository

(3) 添加成功后，转到文件列表页面。

smartliit / test

Code Issues 0 Pull requests 0 Projects 0 Wiki Settings Insights

No description, website, or topics provided. [Edit](#)

1 commit 1 branch 0 releases 1 contributor

Branch: master | New pull request Create new file Upload files Find file Clone or download

smartliit committed on GitHub Initial commit Latest commit ac0599d just now

.gitignore Initial commit just now
README.md Initial commit just now

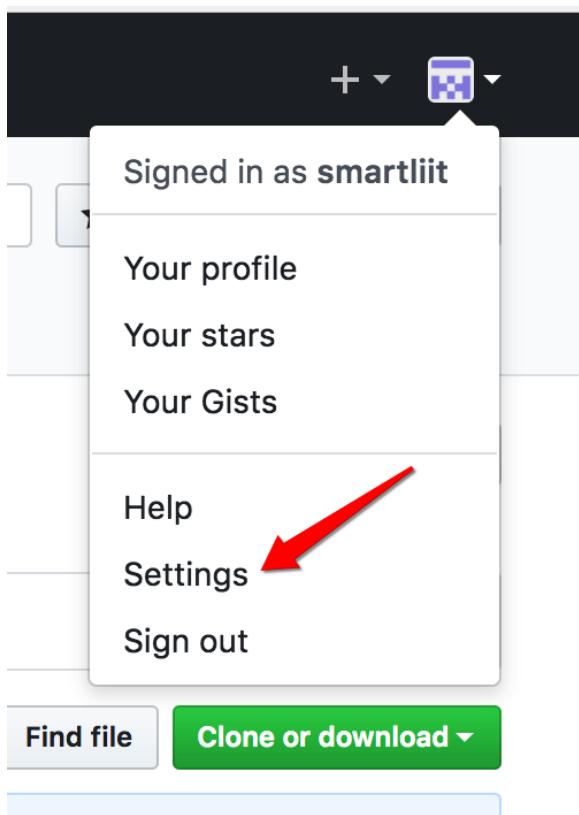
README.md

test

6.2 添加 ssh 账户

(1) 点击账户头像后的下拉三角，选择 'settings'

如果某台机器需要与 github 上的仓库交互，那么就要把这台机器的 ssh 公钥添加到这个 github 账户上



点击 'SSH and GPG keys'，添加 ssh 公钥。

A screenshot of the GitHub 'Personal settings' page. On the left is a sidebar with links: Personal settings, Profile, Account, Emails, Notifications, Billing, SSH and GPG keys (which has a red arrow pointing to it), Security, and Blocked users. The main content area shows the 'SSH keys' section with a message: 'There are no SSH keys with access to your account.' It includes a link to 'generating SSH keys' and 'troubleshoot common SSH Problems'. At the bottom right of this section is a green 'New SSH key' button. Below it is the 'GPG keys' section with a similar message and a 'New GPG key' button.

(2) 在 ubuntu 的命令行中，回到用户的主目录下，编辑文件.gitconfig，

修改某台机器的 git 配置。

```
python@ubuntu:~/git_test$ cd  
python@ubuntu:~$ vi .gitconfig  
python@ubuntu:~$ █
```

(3) 修改为注册 github 时的邮箱，填写用户名。

```
1 [user]  
2     email = smartli_it@163.com  
3     name = smartliit
```

(4) 使用如下命令生成 ssh 密钥。

```
ssh-keygen -t rsa -C "邮箱地址"  
  
python@ubuntu:~$ ssh-keygen -t rsa -C 'smartli_it@163.com'  
ssh-key: 未找到命令  
python@ubuntu:~$ ssh-keygen -t rsa -C 'smartli_it@163.com'  
Generating public/private rsa key pair.  
Enter file in which to save the key (/home/python/.ssh/id_rsa):  
/home/python/.ssh/id_rsa already exists.  
Overwrite (y/n)? y  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:  
Your identification has been saved in /home/python/.ssh/id_rsa.  
Your public key has been saved in /home/python/.ssh/id_rsa.pub.  
The key fingerprint is:  
SHA256:4hPYbI5kd39WzG2fvjeTPyr0YrAVsqR+XxdD3oG[;154 smartli_it@163.com  
The key's randomart image is:  
+---[RSA 2048]---+  
|  
|  
|  
|  
|  
|  
|  
|  
|  
|
```

(5) 进入主目录下的.ssh 文件件，下面有两个文件。

公钥为 id_rsa.pub

私钥为 id_rsa

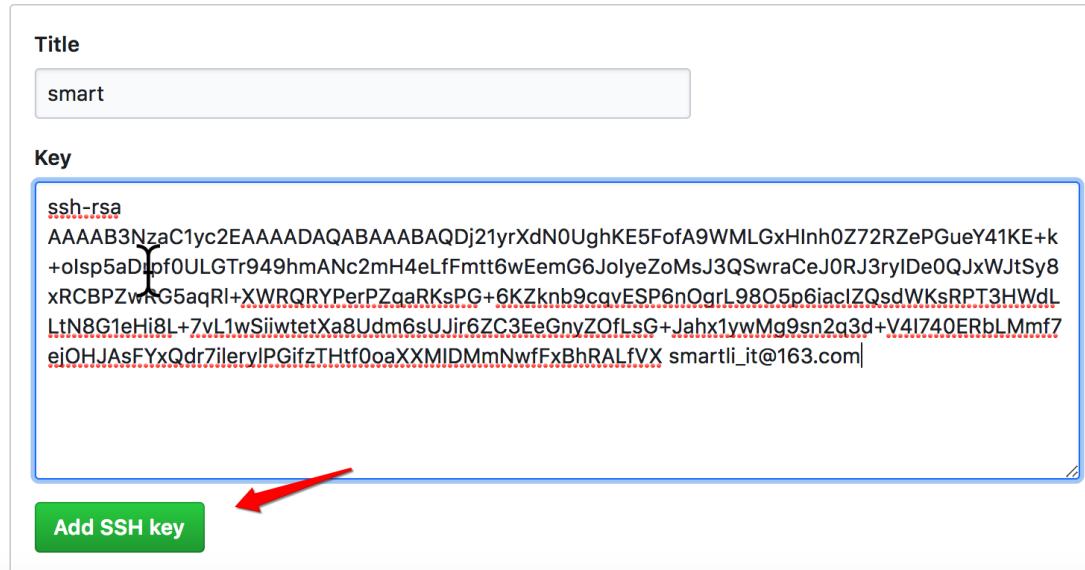
查看公钥内容，复制此内容

```

python@ubuntu:~$ cd .ssh/
python@ubuntu:~/ssh$ ls
id_rsa  id_rsa.pub  known_hosts
python@ubuntu:~/ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDj21yrXdN0UghKE5FofA9WMLGxHInh0Z72RZePGueY
41KE+k+oIsp5aDrpf0ULGTr949hmANc2mH4eLfFmtt6wEemG6JolyeZoMsJ3QSwraCeJ0RJ3ryIDe0QJ
xWjtSy8xRCBPZwRG5aqRl+XWRQRYPPerPZqaRKsPG+6KZknb9cqvESP6nOgrL9805p6iacIZQsdWKsRPT
3HWdLLtN8G1eHi8L+7vL1wSiiwtetXa8Udm6sUJir6ZC3EeGnyZOfLsG+Jahx1ywMg9sn2q3d+V4I740
ERbLMmf7ejOHJAsFYxQdr7ilerylPGifzThtf0oaXXMIDMmNwfFxBhRALfVX smartli_it@163.com
python@ubuntu:~/ssh$ 

```

(6) 回到浏览器中，填写标题，粘贴公钥

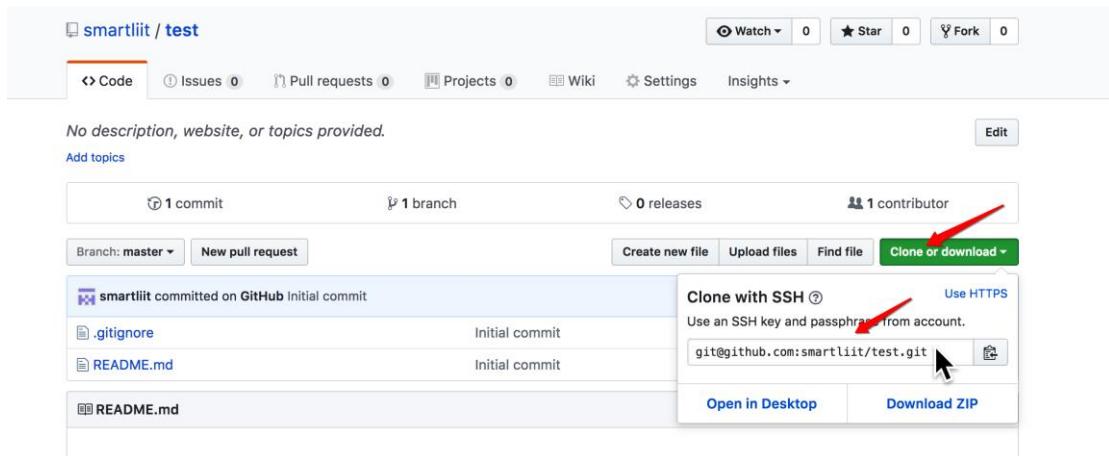


6.3 克隆项目

(1) 在浏览器中点击进入 github 首页，再进入项目仓库的页面

The screenshot shows the GitHub homepage. On the right, there is a sidebar titled 'Your repositories 4' with a 'New repository' button. Below it is a search bar and filter buttons for 'All', 'Public', 'Private', 'Sources', and 'Forks'. A red arrow points to the 'test' repository in the list.

(2) 复制 git 地址



(3) 克隆出错

```
python@ubuntu:~/bj16$ git clone git@github.com:smartliit/bj16.git
正克隆到 'bj16'...
sign_and_send_pubkey: signing failed: agent refused operation
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
python@ubuntu:~/bj16$ ls
git_test test1 test2 test3 test4 test5
python@ubuntu:~/bj16$ eval "$(ssh-agent -s)"
Agent pid 110242
python@ubuntu:~/bj16$ ssh-add
Identity added: /home/python/.ssh/id_rsa (/home/python/.ssh/id_rsa)
python@ubuntu:~/bj16$ git clone git@github.com:smartliit/bj16.git
```

(4) 在命令行中复制仓库中的内容

```
python@ubuntu:~$ git clone git@github.com:smartliit/test.git
正克隆到 'test'...
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
接收对象中: 100% (4/4), 完成.
检查连接... 完成。
python@ubuntu:~$
```

6.4 上传分支

(1) 项目克隆到本地之后，执行如下命令创建分支 smart.

```
python@ubuntu:~/test$ git checkout -b smart
切换到一个新分支 'smart'
python@ubuntu:~/test$ git branch
  master
* smart
python@ubuntu:~/test$
```

(2) 创建一个 code.txt 并提交一个版本。

```
python@ubuntu:~/test$ touch code.txt
python@ubuntu:~/test$ vi code.txt
python@ubuntu:~/test$ git add code.txt
python@ubuntu:~/test$ git commit -m '版本1'
[smart 1e78b54] 版本1
 1 file changed, 1 insertion(+)
  create mode 100644 code.txt
python@ubuntu:~/test$
```

(3) 推送前 github 上文件列表如下图

The screenshot shows a GitHub repository page for 'smartliit / test'. At the top, there are buttons for Watch (0), Star (0), and Fork (0). Below that is a navigation bar with Code, Issues (0), Pull requests (0), Projects (0), Wiki, Settings, and Insights. A red box highlights the 'Branch' section, which shows 1 commit, 1 branch, 0 releases, and 1 contributor. The 'Branch' dropdown is set to 'master'. Below this, a commit from 'smartliit' is listed: 'smartliit committed on GitHub Initial commit' (Latest commit ac0599d an hour ago). Two files are shown: '.gitignore' (Initial commit, an hour ago) and 'README.md' (Initial commit, an hour ago). A large text area at the bottom contains the word 'test'.

(4) 推送前 github 上分支列表如下图

The screenshot shows the GitHub repository page for 'smartliit/test'. At the top, there are buttons for 'Watch' (0), 'Star' (0), and 'Fork' (0). Below that is a navigation bar with links for 'Code', 'Issues 0', 'Pull requests 0', 'Projects 0', 'Wiki', 'Settings', and 'Insights'. The main navigation tabs are 'Overview' (selected), 'Yours', 'Active', 'Stale', and 'All branches'. A search bar for 'Search branches...' is also present. Below the tabs, a section titled 'Default branch' shows the 'master' branch, which is highlighted with a red border and has a note 'Updated an hour ago by smartliit'. There are buttons for 'Default' and 'Change default branch'.

(5) 推送分支，就是把该分支上的所有本地提交推送到远程库，推送时要指定本地分支，这样，git 就会把该分支推送到远程库对应的远程分支上

```
git push origin 分支名称
```

例：

```
git push origin smart
```

```
python@ubuntu:~/test$ git push origin smart
对像计数中: 3, 完成.
压缩对象中: 100% (2/2), 完成.
写入对象中: 100% (3/3), 326 bytes | 0 bytes/s, 完成.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:smartliit/test.git
 * [new branch]      smart -> smart
python@ubuntu:~/test$
```

(6) 再去 github 网站上去看分支页面，内容如下。

The screenshot shows the GitHub repository page for 'smartliit/test'. At the top, there are summary statistics: '2 commits', '2 branches' (highlighted with a red border), '0 releases', and '1 contributor'. Below that is a section for 'Your recently pushed branches:' with a yellow background. It lists the 'smart' branch, which was pushed '1 minute ago'. There is a green button 'Compare & pull request' next to it. Below this, there are buttons for 'Branch: smart' (selected), 'New pull request', and 'Clone or download'. A message says 'This branch is 1 commit ahead of master.' To the right are buttons for 'Pull request' and 'Compare'. The main content area shows a table of files in the 'smart' branch:

File	Description	Last Commit
.gitignore	Initial commit	an hour ago
README.md	Initial commit	an hour ago
code.txt	版本1	5 minutes ago
README.md		

The screenshot shows a GitHub repository page for 'smartliit / test'. At the top, there are buttons for Watch (0), Star (0), and Fork (0). Below the header, there are tabs for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Settings, and Insights (0). The 'Code' tab is selected. A navigation bar below the tabs includes Overview, Yours, Active, Stale, All branches, and a search bar for 'Search branches...'. The main content area is divided into sections: 'Default branch' (master, Default, Change default branch), 'Your branches' (smart, New pull request, Delete), and 'Active branches' (smart, New pull request, Delete). The 'smart' branch is highlighted with a red box.

6.5 将本地分支跟踪服务器分支

```
git branch --set-upstream-to=origin/远程分支名称 本地分支名称
```

例：

```
git branch --set-upstream-to=origin/smart smart
```

```
python@ubuntu:~/test$ git branch --set-upstream-to=origin/smart smart
分支 smart 设置为跟踪来自 origin 的远程分支 smart。
python@ubuntu:~/test$ git status
位于分支 smart
您的分支与上游分支 'origin/smart' 一致。
无文件要提交，干净的工作区
python@ubuntu:~/test$
```

6.6 从远程分支上拉取代码

```
git pull orgin 分支名称
```

例：

```
git pull orgin smart
```

使用上述命令会把远程分支 smart 上的代码下载并合并到本地所在分支。

```
python@ubuntu:~/bj11/bj11$ git pull origin smart
来自 github.com:smartliit/bj11
 * branch            smart      -> FETCH_HEAD
Already up-to-date.
python@ubuntu:~/bj11/bj11$
```

7. 工作使用 git

项目经理:

- (1) 项目经理搭建项目的框架。
- (2) 搭建完项目框架之后，项目经理把项目框架代码放到服务器。

普通员工:

- (1) 在自己的电脑上，生成 ssh 公钥，然后把公钥给项目经理，项目经理把它添加的服务器上面。
- (2) 项目经理会给每个组员的项目代码的地址，组员把代码下载到自己的电脑上。
- (3) 创建本地的分支 dev，在 dev 分支中进行每天的开发。
- (4) 每一个员工开发完自己的代码之后，都需要将代码发布远程的 dev 分支上。

Master: 用户保存发布的项目代码。v1.0, v2.0

Dev: 保存开发过程中的代码。