Exercise 2.4 阅读相关代码，并补全 lib/print.c 中 lp_Print() 函数中缺失的部分来实现字符输出。 ■

当你刚看到 lp_Print() 的代码时，也许会手忙脚乱。不知如何读起，更不清楚该如何补全它。这主要是由于，我们对于 printf 要实现的具体功能没有完整的认识。这里我们给出一些提示，printf 的 format specifier(也就是它的第一个参数) 的格式如下：

$$\%[flags][width][.precision][length]specifier$$

更详尽的描述，请大家查询 C 语言标准库的相关文档[8]。完整的描述较为复杂，我们只需实现我们所有能够支持的部分即可。例如，内核中是无法使用浮点数的，因此所有和浮点输出相关的东西都不必要实现。

2.6 Git——轻松维护和提交代码

在完成了这次实验后，你刚刚伸了个懒腰。突然，一个问题浮现在了你的脑海中：如何提交刚刚完成的实验代码呢？整个实验的代码采用了 git 版本控制系统进行管理，所有与代码管理相关的操作全部通过 git 完成。在本章最后的部分，我们就来了解一下 git 相关的内容。

2.6.1 手工的版本控制

最原始的版本控制是纯手工的版本控制：修改文件，保存文件副本。有时候偷懒省事，保存副本时命名比较随意，时间长了就不知道哪个是新的，哪个是老的了，即使知道新旧，可能也不知道每个版本是什么内容，相对上一版作了什么修改了，当几个版本过去后，很可能就是下面的样子了：

Figure 2.6: 手工版本控制效果图

当然，有些时候，我们不仅是一个人写论文，很可能是多个人同时写一篇 paper。

[8]推荐 cplusplus，这个网站给出了很多的样例 http://www.cplusplus.com/reference/cstdio/printf/

分工，制定计划，埋头苦干，看起来一切都井然有序，后来却只会让人蛋疼不已。本质原因在于每个人都会对书的内容进行改动，结果最后成了这样的情形：我把我修订的最新版的电子版发邮件给他，然后，我继续修改书的内容。一天后，他再把电子书传给我。每到此时，我就必须想清楚，发给他之后到我收到他的文件期间，我在哪里作了哪些改动，还得把我的改动和他的部分合并，真困难。

到这时我们发现一个无法避免的事实：如果每一次小小的改动都要通知对方，那么一些错误的改动将会令我们付出很大的代价：一个错误的改动要频繁在两方同时通知纠正。但如果一次性改动大幅度的内容，尤其是在不连续的段落中删删改改，将变成只有阅读完整篇论文才能知道对方在哪里改动过，才能把两个人的劳动成果合并。论文只有 10 页时还可以接受，但如果变成 20 页，50 页，100 页呢？

后来我们就想如果能有这样一个软件：

- 自动帮我记录每次文件的改动，而且最好是有后悔药的功能，改错了一个东西，我可以轻松撤销。
- 还得有多人协作编辑不费力的好处，有着简洁的指令与操作。
- 最好能像时光机一样穿越回以前，而且不但能穿越回去，还能在不满意的时候穿越回来！
- 如果想查看某次改动，只需要在软件里瞄一眼就可以。

那岂不是十分方便！

版本控制系统就是这样一种神奇的系统。而 git，则是目前世界上最先进的分布式版本控制系统，没有之一。

> **Note 2.6.1** 版本控制是一种记录若干文件内容变化, 以便将来查阅特定版本修订情况的系统。

2.6.2 Git 基础指引

看过那个小故事后，相信大家对 git 这个世界上目前最先进的分布式版本控制系统已经产生了很强的兴趣。简单地说,Git 究竟是怎样的一个系统呢? 别着急，我们慢慢来。

先从 git 的最基础的指令讲起

```
$ git init
```

使用命令

```
$ sudo mkdir learnGit
```

使用 cd learnGit 进入后，执行 git init 创建新的 git 仓库。这时候细心的同学会发现在我们的新文件夹下其实已经有东西了，我们可以使用

```
$ ls -a
```

来观察，我们看到我们的文件夹下多出一个名叫.git 的目录，这个隐藏的.git 目录就是 Git 版本库，更多时候我们称之为仓库 (repository)。仓库是用于跟踪管理版本库的，**在我们的实验中不会对.git 文件夹下的文件进行任何操作，所以不要对该文件夹中的任何文件进行任何手工修改！**

在刚刚我们提到的 **init** 命令完成之后，我们就已经有了一个仓库。我们之前所建立的 learnGit 这个文件夹就是 Git 里的工作区。目前我们的工作区除了包含一个隐藏的.git 版本库目录外空无一物。

> **Note 2.6.2** 在我们的小操作系统实验中我们不需要使用到 git init 命令，每个人一开始就都有一个名为 1406xxxx-lab 的版本库，包含了 lab1 的实验内容。

我们的工作区现在空荡荡的，我们来为它加点料。我们使用下面的指令创建一个新的文本文件

```
$ echo "BUAA_OSLAB" > readme.txt
```

这一步只是创建一个文本文件而已，为了将它添加到版本库，我们需要执行下面的命令

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

注意，到这里还没有结束，你可能会想，那我既然都把 readme.txt 加入了，难道不是已经提交到版本库了吗？但事实就是这样，Git——同其他大多数版本控制系统一样，add 之后需要再执行一次提交操作，提交操作的命令如下

```
$ git commit
```

如果不带任何附加选项的话，git commit 后会弹出一个说明窗口，如下所示

```
GNU nano 2.2.6    文件： /home/13061193/13061193-lab/.git/COMMIT_EDITMSG

Notes to test.
# 请为您的变更输入提交说明。以 '#' 开始的行将被忽略，而一个空的提交
# 说明将会终止提交。
# 位于分支 master
# 您的分支与上游分支 'origin/master' 一致。
#
# 要提交的变更：
#       修改：         readme.txt
#


                                [ 已读取 9 行 ]
^G 求助      ^O 写入      ^R 读档      ^Y 上页      ^K 剪切文字  ^C 游标位置
^X 离开      ^J 对齐      ^W 搜索      ^V 下页      ^U 还原剪切  ^T 拼写检查
```

在上面里书写的 **Notes to test.** 是我们本次提交所附加的说明。注意，弹出的窗口中我们**必须**得添加本次 commit 的说明，这意味着我们不能提交空白说明，否则我们的提交不会成功。而且在添加评论之后，可以按提示按键来成功保存。

Note 2.6.3 初学者一般不太重视 git commit 内容的有效性，总是使用无意义的字符串作为说明提交。但以后你可能就会发现自己写了一个自己看得懂，别人也能看得懂提交说明是多么庆幸。所以尽量让你的每次提交显得有意义，比如“fixed a bug in ...” 这样的描述，顺便推荐一条命令：git commit –amend，这条命令可以重新书写你最后一次 commit 的说明。

这样窗口提交的方式比较繁琐，我们可以采取一种较为简洁的方式

```
$ git commit -m [comments]
```

[comments] 格式为“**评论内容**”，上述的提交过程我们可以简化为下面一条指令

```
$ git commit -m "Notes to test."
```

如果我们在提交后看到类似提示就说明我们提交成功了

```
[master 955db52] Notes to test.
 1 file changed, 1 insertion(+), 1 deletion(-)
```

从我们本次提交中我们可以得到以下信息，可能现在你还不能完全理解这些信息代表的意思，但是没关系，之后我们会讲解

- 本次提交的分支是 master
- 本次提交的 ID 是 955db52
- 提交说明是 Notes to test
- 共有 1 个文件相比之前发生了变化：1 行的添加与 1 行的删除行为

但是在我们实验中，第一次提交可不会这么一帆风顺，我们第一次提交往往会出现下面的提示

```
*** Please tell me who you are.

Run

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

# to set your account's default identity.
# Omit --global to set the identity only in this repository.
```

相信大家从第一句也能推测出，这是要求我们设置提交者身份的。我们设置身份有什么作用呢？别急，等你设置成功了我们再详谈。

Note 2.6.4 从上面我们也知道了，我们可以用

git config –global user.email “you@example.com”

git config –global user.name “Your Name “

这两条命令设置我们的名字和邮箱，在我们的实验中对这两个没有什么要求，大家随性设置就好，给个示例：

git config –global user.email “qianlxc@126.com”

git config –global user.name “Qian”

现在你已设置了提交者的信息，那么做一下这个小练习来快速上手 Git 的使用吧

Exercise 2.5 • 在/home/1406xxxx/ 目录下创建一个名为 README.txt 的文件。这时使用 git status > Untracked.txt 。

- 在 README.txt 文件中随便写点什么，然后使用刚刚学到的 add 命令，再使用 git status > Stage.txt 。
- 之后使用上面学到的 Git 提交有关的知识把 README.txt 提交，并在提交说明里写入自己的学号。
- 使用 cat Untracked.txt 和 cat Stage.txt，对比一下两次的结果，体会一下 README.txt 两次所处位置的不同。
- 修改 README.txt 文件，再使用 git status > Modified.txt 。
- 使用 cat Modified.txt ，观察它和第一次 add 之前的 status 一样吗，思考一下为什么？

■

Note 2.6.5 git status 是一个查看当前文件状态的有效指令，而 git log 则是提交日志，每 commit 一次，Git 会在提交日志中记录一次。git log 将在我们后面乘坐时光机时发挥很大的作用。

相信你做过上述实验后，心里还是会有些疑惑，没关系，我们来一起看一下我们刚才得到的 Untracked.txt ，Stage.txt 和 Modified.txt 的内容

```
1  Untracked.txt 的内容如下
2
3  # On branch master
4  # Untracked files:
5  #   (use "git add <file>..." to include in what will be committed)
6  #
7  #       README.txt
8  nothing added to commit but untracked files present (use "git add" to track)
9
10 Stage.txt 的内容如下
11
12 # On branch master
13 # Changes to be committed:
14 #   (use "git reset HEAD <file>..." to unstage)
15 #
16 #       new file:   README.txt
17 #
18
19 Modified.txt 的内容如下
20
21 # On branch master
22 # 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")
```

通过仔细观察，我们看到第一个文本文件中第 2 行是：Untraced files，而第二个文本文件中第二行内容是：Changes to be committed，而第三个则是 Changes not staged for commit。这三种不同的提示意味着什么，需要你通过后面的学习找到答案，答案就在不远处。

我们开始时已经介绍了 Git 中的工作区的概念，接下来的内容就是 Git 中的最核心的概念，为了能自如运用 Git 中的命令，你一定要仔细学习。

2.6.3 Git 文件四状态

首先对于任何一个文件, 在 Git 内都只有四种状态: 未跟踪 (untracked)、未修改 (unmodified)、已修改 (modified)、已暂存 (staged)

未跟踪 表示没有跟踪 (add) 某个文件的变化，使用 git add 即可跟踪文件

未修改 表示某文件在跟踪后一直没有改动过或者改动已经被提交

已修改 表示修改了某个文件, 但还没有加入 (add) 到暂存区中

已暂存 表示把已修改的文件放在下次提交 (commit) 时要保存的清单中

> **Note 2.6.6** 关于刚才的 exercise 中的思考，实际上是因为 git add 指令本身是有多义性的，虽然差别较小但是不同情境下使用依然是有区别。我们现在只需要记住：新建文件后要 git add，修改文件后也需要 git add。

我们使用一张图来说明文件的四种状态的转换关系

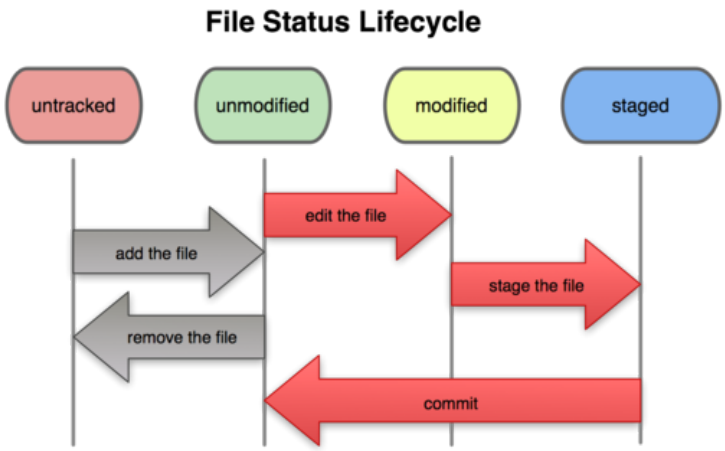

Figure 2.7: Git 中的四种状态转换关系

Exercise 2.6 仔细看看这张图，思考一下红箭头里的 add the file 、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢？ ■

看到这里，相信你对 Git 的设计有了初步的认识。下一步我们就来深入理解一下 Git 里的一些机制，从而让我们可以一次上手，终身难忘。

2.6.4 Git 三棵树

我们的本地仓库由 git 维护的三棵“树”组成。第一个是我们的工作区，它持有实际文件；第二个是暂存区（Index 有时也称 Stage），它像个暂时存放的区域，临时保存你的改动；最后是 HEAD，指向你最近一次提交后的结果。

在我们的.git 目录中，文件.git/index 实际上就是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等），但是**文件的内容并不存储其中，而是保存在 Git 对象库（.git/objects）中**，文件索引建立了文件和对象库中对象实体之间的对应。下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系[9]，希望你能耐着性子仔细理解这张图和不同操作所带来的不同影响。

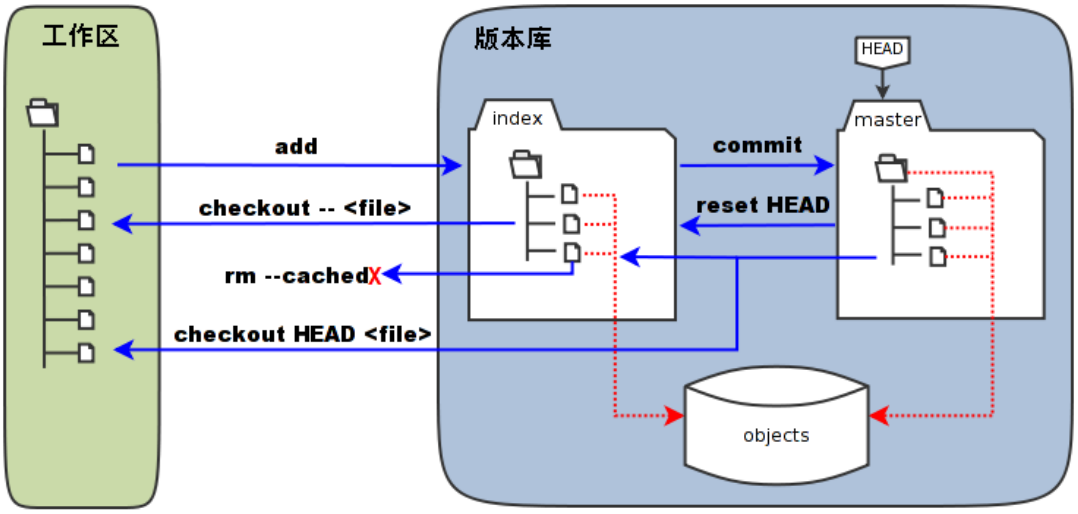

Figure 2.8: 工作区、暂存区和版本库

- 图中 objects 标识的区域为 Git 的对象库，实际位于”.git/objects” 目录下。
- 图中左侧为工作区，右侧为版本库。在版本库中标记为”index” 的区域是暂存区（stage, index），标记为”master” 的是 master 分支所代表的目录树。
- 图中我们可以看出此时”HEAD” 实际是指向 master 分支的一个“游标”。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。

[9]这张图转载自网站http://www.worldhello.net/2010/11/30/2166.html

- 当对工作区修改（或新增）的文件执行”git add” 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的 ID 被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，会将暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。
- 当执行”git rm –cached <file>” 命令时，会直接从暂存区删除文件，工作区则不做出改变。
- 当执行”git reset HEAD” 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行”git checkout – <file>” 命令时，会用暂存区指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行”git checkout HEAD <file>” 命令时，会用 HEAD 指向的 master 分支中的指定文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

我们在考虑暂存区和版本库的关系的时候，可以粗略地认为暂存区是开发版，而版本库可以认为是稳定版，而 commit 其实就是将稳定版版本升到当前开发版的一个操作。

Git 中引入的**暂存区**的概念可以说是 Git 里最难理解但是却是最有亮点的设计之一，我们在这里不再详细介绍其能快速快照与回滚的原理，如果有兴趣的同学不妨去看看Pro Git这本书。

2.6.5 Git 时光机

Figure 2.9: 多啦 A 梦的时光机

我们都知道多啦 A 梦的时光机能穿越时空回到过去，而在我们神奇的 Git 里，也有堪称时光机的指令哦！在学习之前, 我们先学习一下已经大致了解的一些伪 · 时光机指令，比如下面这些

git rm –cached <file> 如果我们不小心跟错人了，我们可以用这条指令把他从备胎中果断删掉！开个小玩笑，这个指令其实是指从暂存区中删去一些我们不想跟踪的文件，比如我们自己调试用的文件等。

git checkout – <file> 如果我们在工作区改呀改，把一堆文件改得乱七八糟的，发现编译不过了！只要我们还没 git add，我们就能使用这条命令，把它变回曾经美妙的样子。

git reset HEAD <file> 我们刚才说了，如果没有 git add 把修改放入暂存区的话，我们可以使用 checkout 其命令，那么如果我们不慎已经 git add 加入了怎么办呢？那就需要这条指令来帮助我们了！这条指令可以让我们的暂存区焕然一新。再对同一个文件使用楼上那条指令，哈哈，世界清静了。

git clean <file> -f 如果你的工作区这时候混入了奇怪的东西，你没有追踪它，但是想清除它的话就可以使用这条指令，它将帮你把奇怪的东西剔除出去。

好了，学了这么多，我们来利用自己的知识帮助小明摆脱困境吧。

Thinking 2.1

- 深夜，小明在做操作系统实验。困意一阵阵袭来，小明睡倒在了键盘上。等到小明早上醒来的时候，他惊恐地发现，他把一个重要的代码文件 printf.c 删除掉了。苦恼的小明向你求助，你觉得怎样能帮他把代码文件恢复呢？
- 正在小明苦恼的时候，小红主动请缨帮小明解决问题。小红很爽快地在键盘上敲下了 git rm printf.c，这下事情更复杂了，现在你又该如何处理才能弥补小红的过错呢？
- 处理完代码文件，你正打算去找小明说他的文件已经恢复了，但突然发现小明的仓库里有一个叫 **Tucao.txt**，你好奇地打开一看，发现是吐槽操作系统实验的，且该文件已经被添加到暂存区了，面对这样的情况，你该如何设置才能使 Tucao.txt 在不从工作区删除的情况下不会被 git commit 指令提交到版本库？

■

关于上面那些撤销指令，等到你哪天突然不小心犯错的时候再来查阅即可，当然更推荐你使用 git status 来看当前状态下 Git 的推荐指令。我们现阶段先掌握好 add 和 commit 的用法即可。当然，**一定要慎用撤销指令**。虽然说 Git 理论上没有不能穿越的时空，但是需要我们功力深厚，掌握许多奇技淫巧，否则撤销之后如何撤除撤销指令将是一件难事。

介绍完上面三条撤销指令，我们来介绍真正的时光机指令

```
1  git reset --hard
```

为了体会它的作用，我们做个小练习试一下

Exercise 2.7

- 找到我们在/home/1406xxxx/下刚刚创建的 README.txt，没有的话就新建一个。
- 在文件里加入 **Testing 1**，add，commit，提交说明写 1。
- 模仿上述做法，把 1 分别改为 2 和 3，再提交两次。
- 使用 git log 命令查看一下提交日志，看是否已经有三次提交了？记下提交说明为 3 的哈希值[a]。
- 开动时光机！使用 git reset –hard HEAD^ ，现在再使用 git log，看看什么没了？
- 找到提交说明为 1 的哈希值，使用 git reset –hard <Hash-code> ，再使用 git log，看看什么没了？
- 现在我们已经回到过去了，为了再次回到未来，使用 git reset –hard <Hash-code> ，再使用 git log，我胡汉三又回来了！

■

[a]使用 git log 命令时，在 commit 标识符后的一长串数字和字母组成的字符串

这条指令就是我们可前进，可后退，还可以随意篡改“历史”的时光机是也。它有两种用法，第一种是使用 HEAD 类似形式，如果想退回上个版本就用 HEAD^，上上个的话就用 HEAD^^，当然要是退 50 次的话写不了那么多^，可以使用 HEAD~50 来代替。第二种就是使用我们神器 Hash 值，用 Hash 值不仅可以回到过去，还可以“回到未来”。Hash 值在手，天下任我走！

现在我们已经学会了一大杀器，其正式的名字其实叫做**版本回退**。我们再来学个 Git 里同样被称为**必杀级特性**的神奇性质！

2.6.6 Git 分支

如果你还有印象的话，我们之前提到过分支这个概念，那么分支是个什么东西呢？分支就是科幻电影里面的平行宇宙，不同的分支间不会互相影响。或许当你正在电脑前努力学习操作系统的时候，另一个你正在另一个平行宇宙里努力学习面向对象。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。在我们实验中也会多次使用到分支的概念。首先我们来讲一条创建分支的指令

```
# 创建一个基于当前分支产生的分支，其名字为 <branch-name>
$ git branch <branch-name>
```

这条指令往往会在我们进行周一小测的时候用到。其功能相当于把当前分支的内容拷贝一份到新的分支里去，然后我们在新的分支上做测试功能的添加即可，不会影响实验分支的效果等。假如我们当前在 master[10]分支下已经有过三次提交记录，这时我们使用 branch 命令新建了一个分支为 testing（参考图 2.10）。

[10]master 分支是我们的主分支，一个仓库初始化时自动建立的默认分支

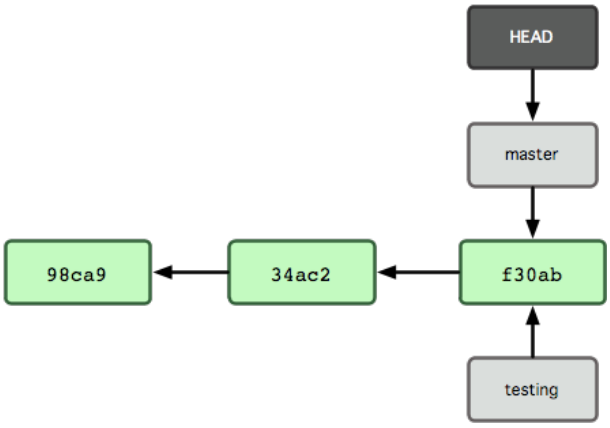

Figure 2.10: 分支建立后

删除一个分支也很简单，只要加上 -d 选项 (-D 是强制删除) 即可，就像这样

```
# 创建一个基于当前分支产生的分支，其名字为 <branch-name>
$ git branch -d(D) <branch-name>
```

想查看分支情况以及当前所在分支，只需要加上 -a 选项即可

```
# 查看所有的远程与本地分支
$ git branch -a

# 使用该命令的效果如下
# 前面带 * 的分支是当前分支
  lab1
  lab1-exam
* lab1-result
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/lab1
  remotes/origin/lab1-exam
  remotes/origin/lab1-result
  remotes/origin/master
# 带 remotes 是远程分支，在后面提到远程仓库的时候我们会知道
```

我们建立了分支并不代表会自动切换到分支，那么，Git 是如何知道你当前在哪个分支上工作的呢？其实答案也很简单，它保存着一个名为 HEAD 的特别指针。在 Git 中，它是一个指向你正在工作中的本地分支的指针，可以将 HEAD 想象为当前分支的别名。运行 git branch 命令，仅仅是建立了一个新的分支，但不会自动切换到这个分支中去，所以在这个例子中，我们依然还在 master 分支里工作。

那么我们如何切换到另一个分支去呢，这时候我们就要用到这个我们在实验中更常见的分支指令了

```
# 切换到 <branch-name> 代表的分支，这时候 HEAD 游标指向新的分支
$ git checkout <branch-name>
```

比如这时候我们使用 **git checkout testing**，这样 HEAD 就指向了 testing 分支 (见图2.11)。

这时候你会发现你的工作区就是 testing 分支下的工作目录，而且在 testing 分支下的修改，添加与提交不会对 master 分支产生任何影响。

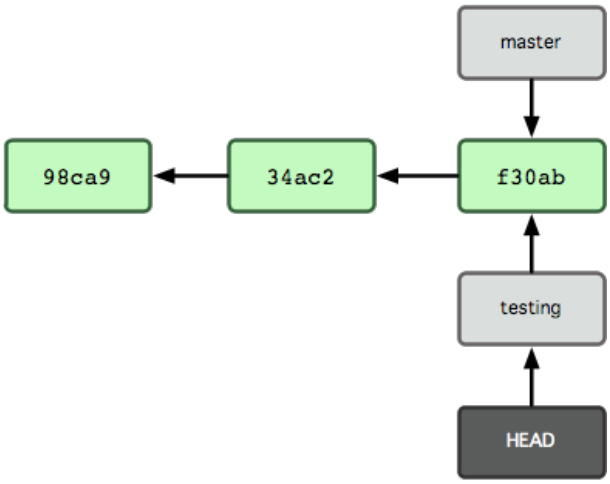

Figure 2.11: 分支切换后

在我们的操作系统实验中，有以下几种分支：

labx 这是我们提交实验代码的分支，这个分支不需要我们手动创建。当写好代码提交到服务器上后，在该次实验结束后，使用后面提到的更新指令可获取到新的实验分支，到时只需要使用 git checkout labx 即可进行新的实验。

labx-exam 这是我们周一小测实验的分支，每次需要使用 git branch 指令将刚完成的实验分支拷贝一份到 labx-exam 分支下，并进行小测代码的填写。

labx-result 这是我们每次实验结果的分支，每次的实验结果将会在该分支工作区的 log 文件夹下，数字越大代表检测的时间越近。测试下方 Summary：Number (in 100)，只要 Number >= 60 即算作通过本次实验。

> **Note 2.6.7** 每次实验虽然是 60 算实验通过，但是 Summary 最好是 100。因为每次新实验的代码是你刚完成的实验代码以及一些新的要填充的文件组成的，前面实验的错误可能会在后面的实验中变成不小的坑。当然 Summary 为 100 也不代表实验一定全部正确，尽可能多花点时间理解与修改。

我们之前所介绍的这些指令只是在本地进行操作的，其中必须掌握

1. git add
2. git commit
3. git branch
4. git checkout

其余指令可以临时查阅，当然掌握对你益处现在体会不出来，但当你们小团队哪天一起做项目的时候，你就会体会到掌握这么多 Git 的知识是件多么幸福的事情了。之前我们所有的操作都是在本地版本库上操作的，下面我们要介绍的是一组和远程仓库有关的指令。这组指令是最容易出错的，所以你一定要认真学习。

2.6.7 Git 远程仓库与本地

在我们的实验中，我们设立了几台服务器主机作为大家的远程仓库。那么远程仓库是什么呢？远程仓库其实和你本地版本库结构是一致的，只不过远程仓库是在服务器上的仓库，而本地仓库是在本地的。实验中我们每次对代码有所修改时，最后都需要在实验截止时间之前提交到服务器上，我们以服务器上的远程仓库里的代码为评测标准哦。我们先介绍一条我们实验中比较常用的一条命令

```
# git clone 用于从远程仓库克隆一份到本地版本库
$ git clone git@ip: 学号 -lab
```

从名字也能很容易理解这条指令的含义所在，我们就是使用 clone 指令而把服务器上的远程仓库拷贝到本地版本库里。这是一条很重要的指令，以后我们会经常使用。包括前期检查我们是否成功地提交到服务器上，以及后期使用 Git 为开源社区做贡献时都需要。但是初学者在使用这条命令的时候可能会遇到一个问题，那么来仔细思考一下下面的问题

Thinking 2.2 思考下面四个描述，你觉得哪些正确，哪些错误，请给出你参考的资料或实验证据。

1. 克隆时所有分支均被克隆，但只有 HEAD 指向的分支被检出。
2. 克隆出的工作区中执行 git log、git status、git checkout、git commit 等操作不会去访问远程版本库。
3. 克隆时只有远程版本库 HEAD 指向的分支被克隆。
4. 克隆后工作区的默认分支处于 master 分支。

■

Note 2.6.8 检出某分支指的是在该分支有对应的本地分支，使用 git checkout 后会在本地检出一个同名分支自动跟踪远程分支。比如现在本地空无一物，远程有一个名为 os 的分支，我们使用 git checkout os 即可在本地建立一个跟远程分支同名，自动追踪远程分支的 os 分支，并且在 os 分支下 push 时会默认提交到远程分支 os 上。

初学者最容易犯的一个错误是，在检查自己是否提交到服务器上时，克隆下来就着急忙慌地编译。大侠莫慌，看清楚分支再编译。我们克隆下来时默认处于 master 分支，但很可惜实验的代码是不会在 master 分支上测试的，所以我们要先使用git checkout检出对应的 labx 分支，再进行测试。

下面再介绍两条跟远程仓库有关的指令，其作用很简单，但要用好却是比较难。

```
# git push 用于从本地版本库推送到服务器远程仓库
$ git push

# git pull 用于从服务器远程仓库抓取到本地版本库
```

```
$ git pull
```

git push 只是将本地版本库里已经 commit 的部分同步到服务器上去，不包括**暂存区**里存放的内容。在我们实验中除了还可能会加些选项使用

```
# origin 在我们实验里是固定的，以后就明白了。branch 是指本地分支的名称。
$ git push origin [branch]
```

这条指令可以将我们本地创建的分支推送到远程仓库中，在远程仓库建立一个同名的本地追踪的远程分支。比如我们实验小测时要在本地先建立一个 **labx-exam** 的分支，在提交完成后，我们要使用 **git push origin labx-exam** 在服务器上建立一个同名远程分支，这样服务器才可以通过检测该分支的代码来检测你的代码是否正确。

git pull 是条更新用的指令，如果助教老师在服务器端发布了新的分支，下发了新的代码或者进行了一些改动的话，我们就需要使用 git pull 来让本地版本库与远程仓库保持同步。

2.6.8 Git 冲突与解决冲突

这两条指令含义注释里也写得清楚，但是还是很容易出问题。新手使用 push 时，容易出现的大问题会是这样的

```
中文版：
To git@github.com:1406xxxx.git
 ! [rejected]        master -> master (non-fast-forward)
error: 无法推送一些引用到 'git@github.com:1406xxxx.git'
提示：更新被拒绝，因为您当前分支的最新提交落后于其对应的远程分支。
提示：再次推送前，先与远程变更合并（如 'git pull ...'）。详见
提示：'git push --help' 中的 'Note about fast-forwards' 小节。

英文版：
To git@github.com:1406xxxx.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'To git@github.com:1406xxxx.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

你的提示可能是英文的，但这并不妨碍问题的发生，这个问题是因为什么而产生的呢？我们来分析一下，想象你在公司和在家操作同一个分支，在公司你对一个文件进行了修改，然后进行了提交。回了家又对同样的文件做了不同的修改，在家中使用 push 同步到远程分支了。但等你回到公司再 push 的时候就会发现一个严重的问题：现在远程仓库和本地仓库已经分离开变成两条岔路了（见图2.12）。

这样的话远程仓库可就为难了，你在公司的提交有效，在家里的提交也有效，你又不想浪费劳动成果，想让远程仓库把你的提交全部接受，那么我们怎么样才能解决这个问题呢？这时候就要请出我们的 **git pull** 指令了！

你此时可能会产生一个很大的疑问，在 push 之前，使用 git pull 轻轻一挥，难道问题就能全部解决？答案当然是否定的，我们不能指望 Git 帮我们把文件中的修改全部妥善合并，但是 Git 为我们提供了另一种机制帮我们能快速定位有冲突（conflict）的文件，这时候我们使用 git pull，你可能会看到有下面这样的提示

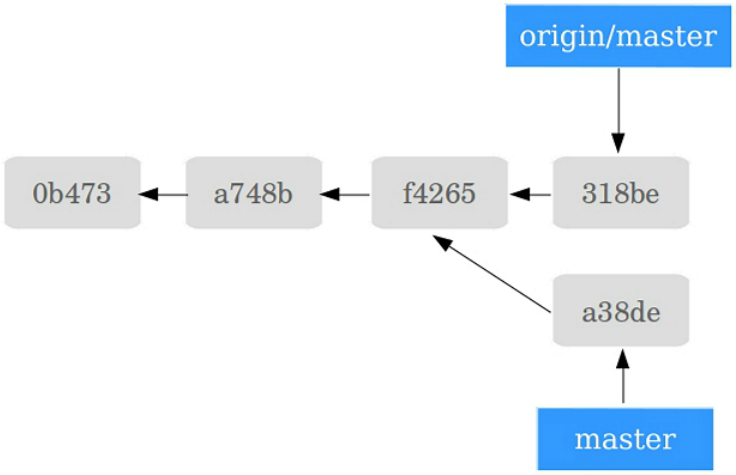

Figure 2.12: 远程仓库与本地仓库的岔路

```
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

有冲突的文件中往往包含一部分类似如下的奇怪代码，我们打开 test.txt，发现这样一些“乱码”

```
a123
<<<<<<< HEAD
b789
=======
b45678910
>>>>>>> 6853e5ff961e684d3a6c02d4d06183b5ff330dcc
c
```

冲突标记<<<<<<< 与 ======== 之间的内容是你在家里的修改，========与>>>>>>> 之间的内容是你在公司的修改。

要解决冲突也很简单：编辑冲突文件，将其中冲突的内容手工合理合并一下就可以了，当然记得在文件中解决了冲突之后要重新 add 该文件并 commit。大声告诉我，是不是非常简单？

然而世间并没有那么多简单的事情，如果你足够不幸，你可能在 git pull 的时候也会遇到不小的问题，问题可能是这样的

```
error: Your local changes to the following files would be overwritten by merge:
        1406xxxx-lab/readme.txt
Please, commit your changes or stash them before you can merge.
Aborting
```

其实提示已经比较清楚了，这里我们只需要把我们之前的所有修改全部提交(commit) 即可，提交之后再 git pull 就好。当然，有更高级的用法是这样的，不推荐大家现在学习，如果你已经熟悉了 Git 的基础操作，那么可以阅读git stash 解决 git pull 冲突

不要觉得这一节的冲突一节不需要学习，因为你可能会想：我现在怎么可能在公司和家里同时修改文件呢！但是要注意，在远程仓库编辑的不止你一个人，还有助教老师，

助教老师一旦修改一些东西都有可能产生冲突，所以你一定要认真学会这一节的内容。当然实践是最好的老师，我们再来实践一下

Exercise 2.8 仔细回顾一下上面这些指令，然后完成下面的任务

- 在 /home/1406xxxx/1406xxxx-lab 下新建分支，名字为 Test
- 切换到 Test 分支，添加一份 readme.txt，内容写入自己的学号
- 将文件提交到本地版本库，然后建立相应的远程分支。

■

到这里 Git 教程基本就算是结束了，能看完这么长的教程也真是辛苦你了，奉送一下实验代码提交流程的简明教程，希望你可以快速上手，终身难忘！

2.6.9 实验代码提交流程

modify 写代码。

git add & git commit <modified-file> 提交到本地版本库。

git pull 从服务器拉回本地版本库，并解决服务器版本库与本地代码的冲突。

git add & git commit <conflict-file> 将远程库与本地代码合并结果提交到本地版本库。

git push 将本地版本库推到服务器。

mkdir test & cd test & git clone 建立一个额外的文件夹来测试服务器上的代码是否正确。

而我们在一次实验结束，新的实验代码下发时，一般是按照以下流程的来开启新的实验之旅。

git add & git commit 如果当前分支的暂存区还有东西的话，先提交。

git pull 这一步很重要！要先确保服务器上的更新全部同步到本地版本库！

git checkout labx 检出新实验分支并进行实验。

谨记，一定要勤使用 **git pull**，这条指令很重要！有事没事，同步一下！

感谢你看这篇长长的 Git 教程到现在，希望你能快乐地使用 Git，若有不会勤查教程[11]。如果你希望能学到更厉害的技术，推荐GitHug，这是一个关于 Git 的通关小游戏。开启你快乐的实验之旅吧！^_^

[11]推荐廖雪峰老师的网站: http://www.liaoxuefeng.com/

2.7 实验思考

- 思考 -小明的困境
- 思考 -克隆命令

请认真做练习，然后把实验思考里的内容附在实验文档中一起提交！这是我们第一次交实验文档，我们的实验文档将包含以下内容：[FIXME]