

# 实验思考题

## 1.Thinking0.1

不一样，Modified.txt中README.txt在Changes not staged for commit下，而第一次add前README.txt在Untracked files下

## 2.Thinking0.2

add the file: `git add <filename>`

stage the file: `git add \<filename>`

commit: `git commit -m "xxx"`

## 3.Thinking0.3

1. 使用git checkout -- printf.c
2. 使用git checkout HEAD printf.c
3. 使用git rm --cached Tucao.txt

## 4.Thinking0.4

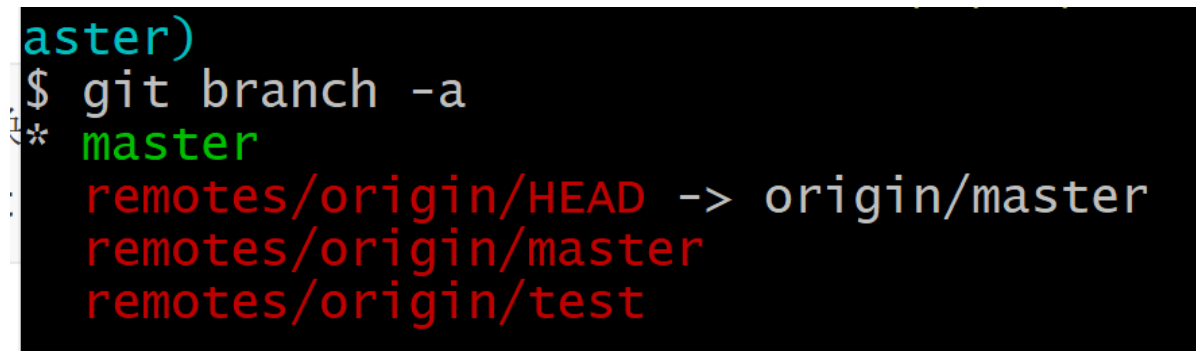
第3次hash:946ddcbf6932f4122f5a6cd484c564506b5d47e8

第一次: 8c6ea602a48f8667887e6b719b25208f7868135f

使用git reset --hard <HashCode>可以使特定hash值对应的版本库中文件覆盖工作区中的文件，实现版本后退

## 5.Thinking0.5

1. 正确，克隆时所有分支都被克隆，但只有HEAD指向的master分支被checkout。克隆后使用git branch -a可以查看所有分支，可以看到，test分支没有被检出

A terminal window with a black background and white text. The command '\$ git branch -a' has been executed. The output shows three branches: '1 \* master' (where '1' is in green and '\*' is in red), 'remotes/origin/HEAD -> origin/master' (in red), 'remotes/origin/master' (in red), and 'remotes/origin/test' (in red).

```
aster)
$ git branch -a
1 * master
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/test
```

2. 正确，这些操作都是针对本地库进行操作
3. 错误，所有分支都被克隆，但只有远程库HEAD指向的master库被检出（如上图）
4. 正确，如图

## 6.Thinking0.6

```

git@20373159:~/newdir$ echo first
first
git@20373159:~/newdir$ echo second >output.txt
git@20373159:~/newdir$ cat output.txt
second
git@20373159:~/newdir$ echo third >output.txt
git@20373159:~/newdir$ cat output.txt
third
git@20373159:~/newdir$ echo forth >>output.txt
git@20373159:~/newdir$ cat output.txt
third
forth

```

原因：重定向中，>是覆盖文件内容，>>是追加到文件

## 7.Thinking0.7

command文件内容：

```

1  #!/bin/bash
2  echo 'echo shell start...' > test
3  echo 'echo set a = 1' >> test
4  echo 'a=1' >> test
5  echo 'echo set b = 2' >> test
6  echo 'b=2' >> test
7  echo 'echo set c = a+b' >> test
8  echo 'c=${a+$b}' >> test
9  echo 'echo c = $c' >> test
10 echo 'echo save c to ./file1' >> test
11 echo 'echo $c>file1' >> test
12 echo 'echo save b to ./file2' >> test
13 echo 'echo $b>file2' >> test
14 echo 'echo save a to ./file3' >> test
15 echo 'echo $a>file3' >> test
16 echo 'echo save file1 file2 file3 to file4' >> test
17 echo 'cat file1>file4' >> test
18 echo 'cat file2>>file4' >> test
19 echo 'cat file3>>file4' >> test
20 echo 'echo save file4 to ./result' >> test
21 echo 'cat file4>>result' >> test

```

result文件内容：

```

1  3
2  2
3  1

```

解释：echo进行打印输出时，字符串会原样输出，但是遇到变量会进行值的替换（单引号内部的变量会原样输出，不会替换）。因此执行test文件会在屏幕上输出Shell Start 等，但对于echo \$c>file1, \$c首先会被替换为3，之后输出重定向到file1，同理\$b,\$a也是被替换为2和1输出到file2与file3.之后用cat指令将file1内容覆盖file4,file2和file3追加到file4末尾，最后file4内容追加到result中。

## 实验难点

### 1. Makefile的使用

本实验知识点较为简单，主要是应用的时候容易出bug。花时间较长的是在Exercise 0.4的第二问，最开始内层Makefile写成这样

```
1 fibo: fibo.c main.c fibo.h
2     gcc fibo.o main.o -o fibo -I../include
```

但会报错：No rule to make target 'fibo.h', needed by 'fibo'. Stop. 后来知.h文件要设置路径，于是改成了这样：

```
1 VPATH=../include
2 fibo: fibo.c main.c fibo.h
3     gcc fibo.o main.o -o fibo -I../include
```

但还是不对，原因是自己误解了一个makefile的自动推导规则，.o文件需要通过gcc -c 编译出来，因此最终改成了这样：

```
1 #以下为外层Makefile
2 DOBJ= code
3 fibo: ./include/fibo.h
4     cd code/ && make
5     gcc -o fibo $(DOBJ)/main.o $(DOBJ)/fibo.o -I../include
6 clean:
7     $(MAKE) --directory=$(DOBJ) clean
8
9 #以下为内层Makefile
10 VPATH=../include
11 fibo: fibo.c main.c fibo.h
12     gcc -c fibo.c -I../include
13     gcc -c main.c -I../include
14 clean:
15     rm fibo.o main.o
```

之所以会出现之前的错误，是自己对makefile的规则不太熟悉。

### 2. Git

Git的一些指令容易混淆。比如 git reset HEAD 只会对暂存区进行撤销，而 git checkout HEAD 则是对暂存区和工作区都进行撤销操作。

有一些组合指令应用性很强，包括：

```
1 git rm --cached <file> #从暂存区删除文件，工作区则不做出改变
2 git commit
3 git push
4 #上述3条指令连用可以将远程库的修改撤销掉
```

```

1 git clean -n #演练，告知那些文件将被删除
2 git clean -f #强力删除工作区里未被追踪（add）的文件
3 git clean -df #删除工作区里未被追踪的文件夹和文件
4 #连用可以删除切换分支后工作区里原有的未追踪文件和目录

```

### 3. 流编辑器

本次实验最难的部分大概就是流编辑器的使用了。此实验涉及了2种流编辑器。

1. **sed**，处理的是输入流，对当前文件读一行处理一行，不具备存储能力。也就是说，一旦该行流过去了，就不会再管了。所以使用这样的语句是很危险的,如 `sed -n 'xxx' file1 > file1`，很可能将file1清空。

◦ 用法总结如下：

```

1 sed [选项] '命令' 输入文本 #命令两边的'不能少，既可以是双引号和单引号，双引号是为了
   加转义\和变量。输入文本即为文件
2
3 #选项(常用):
4 -n: 只有sed处理的流会输出。否则输入文本的所有内容都会先被输出。
5 -e: 进行多项编辑，即对输入行应用多条 sed 命令时使用。可以用一个sed实现多个sed操作，
   如
6 # sed -e '3a\str' -e '2a\str' my.txt
7 # sed -e '8p' -e '32p' -n file
8 -i: 保存处理结果到文件，否则不改变文件内容
9
10 #命令(常用):
11 a : 新增， a 后紧接着\，在当前行的后面添加一行文本
12 c : 取代， c 后紧接着\，用新的文本取代本行的文本
13 i : 插入， i 后紧接着\，在当前行的上面插入一行文本
14 d : 删除，删除当前行的内容
15 p : 显示，把选择的内容输出。通常 p 会与参数 sed -n 一起使用。如sed -n '/main/p'
   my.txt,将my.txt中含有main的行打印出来
16 # 使用这些命令前，要加行号，如: sed '2,$d' my.txt, 表示剔除从第2行到最后一行所有
   内容
17 s : 取代，格式为 s/re/string, re 表示正则表达式，string 为字符串，功能为将正则表
   达式替换为字符串。例如: sed '3s/re/string/g' my.txt 将my.txt中的第三行替换，不改
   变my.txt
18 #例如sed 's/ .*//' file , 显示每一行的第一个单词

```

2. **awk**，也是流编辑器，但是具有存储能力。用法如下

```

1 格式: awk 'program' file
2 program为一段程序，形式:
3 pattern1 {action1} pattern2 {action2}....., pattern为条件，action为命令。模式
   之间是或的关系
4 pattern:ed的正则表达式，逻辑表达式
5 action:如print,printf
6 awk -F, '{print $2}' my.txt#-F选项用来指定用于分隔的字符，默认是空格。所以该命令的
   $n就是用逗号分隔的第n项了。由于没有写pattern，无条件执行
7 awk '$1>2 {print $1,$3}' my.txt #打印第一项大于2的行的第1和第3项。
8 awk '{print NR,": "$0}' my.txt# NR是行号，$0为当前行，即显示行号和当前行

```

命令中出现的\$*n*代表每一行中用空格(不加-F参数指定分隔符时)分隔后的第*n*项。

## 体验与感想

---

本次实验内容较为简单，但细节需要把握到位，总共花了一天左右的时间。完成了这次实验，对vim、Makefile、部分Linux指令、Shell脚本的编写都有了初步的认识，为之后操作系统课程的学习奠定了一部分基础，也解决了预习时部分关于Git、Shell、Linux的疑问。