Lab0

> 班级：222111
>
> 学号：22373340
>
> 姓名：詹佳博

思考题

Thinking 0.1

```
git@22373340:~/learnGit (master)$ touch README.txt
git@22373340:~/learnGit (master)$ git status > Untracked.txt
git@22373340:~/learnGit (master)$ vim README.txt
git@22373340:~/learnGit (master)$ git add README.txt
git@22373340:~/learnGit (master)$ git status > Stage.txt
git@22373340:~/learnGit (master)$ git commit -m "22373340"
[master 20d99f6] 22373340
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
git@22373340:~/learnGit (master)$ cat Untracked.txt
位于分支 master
未跟踪的文件：
  （使用 "git add <文件>..." 以包含要提交的内容）
        README.txt
        Untracked.txt

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
git@22373340:~/learnGit (master)$ cat Stage.txt
位于分支 master
要提交的变更：
  （使用 "git restore --staged <文件>..." 以取消暂存）
        新文件：    README.txt

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

git@22373340:~/learnGit (master)$ vim README.txt
git@22373340:~/learnGit (master)$ git status > Modified.txt
git@22373340:~/learnGit (master)$ cat Modified.txt
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 restore <文件>..." 丢弃工作区的改动）
        修改：      README.txt

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

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

- 首先进行创建文件操作，对此时的 `git` 进行分析，可知 `README.txt` 文件此时处于**未跟踪状态**（`Untracked.txt`）。
- 然后进行修改文件操作，并且将修改后的文件使用 `git add` ，对此时的 `git` 进行分析，可知 `README.txt` 文件此时处于**暂存状态**（`Stage.txt`）。
- 进行提交，正常使用 `git commit (-m)` 指令，此时文件进入**未修改状态**。
- 此时执行两次 `cat` 指令来印证以上分析正确。
- 然后进行修改文件操作，区别是将**已提交的文件进行修改**，对此时的 `git` 进行分析，可知 `README.txt` 文件此时处于**修改状态**（`Modified.txt`）。

- 执行一次 `cat` 指令来印证以上分析正确。
    - 对于 `add` 之前的 `status` ，显然是不一样的。之前的并没有经过提交的文件修改，文件处于**未跟踪状态**；而现在明显是对提交的文件修改，文件就会处于**修改状态**。

Thinking 0.2

- `add the file` 对应着 `git add $filename`
- `stage the file` 对应着 `git add $filename`
- `commit` 对应着 `git commit -m $message`

Thinking 0.3

- `git restore print.c`
- `git reset HEAD print.c` 然后 `git checkout -- print.c`
- `git rm --cached hello.txt` 或者 `git restore --staged hello.txt`

Thinking 0.4

- 前四步的行为使得 `git` 上出现三次提交记录。使用 `git log` 查看后发现事实如此。

```
git@22373340:~/learnGit (master)$ git log
commit 6650df264bfb2835e9236093975186e89fe030cf (HEAD -> master)
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:26:19 2024 +0800

    3

commit cbcbd189f03c2d2c0f8cb569e79b4699f1e24046
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:26:02 2024 +0800

    2

commit 6186ecd8ec2f802f71371768ca3753ff0c088b2e
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:25:41 2024 +0800

    1
```

- 第五步进行版本回退，使得回退到 `HEAD^` 即上一个版本 (2) 。使用 `git log` 查看后发现事实如此。

```
git@22373340:~/learnGit (master)$ git reset --hard HEAD^
HEAD 现在位于 cbcbd18 2
git@22373340:~/learnGit (master)$ git log
commit cbcbd189f03c2d2c0f8cb569e79b4699f1e24046 (HEAD -> master)
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:26:02 2024 +0800

    2

commit 6186ecd8ec2f802f71371768ca3753ff0c088b2e
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:25:41 2024 +0800

    1
```

- 第六步使用哈希值进行版本回退，使得回退到版本 (1) 。使用 `git log` 查看后发现事实如此。

```
git@22373340:~/learnGit (master)$ git reset --hard 6186ecd8ec2f802f71371768ca3753ff0c088b2e
HEAD 现在位于 6186ecd 1
git@22373340:~/learnGit (master)$ git log
commit 6186ecd8ec2f802f71371768ca3753ff0c088b2e (HEAD -> master)
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:25:41 2024 +0800

    1
```

- 第七步想回到最新版本 (3)，使用 3 的哈希值。使用 `git log` 查看后发现事实如此。

```
git@22373340:~/learnGit (master)$ git reset --hard 6650df264bfb2835e9236093975186e89fe030cf
HEAD 现在位于 6650df2 3
git@22373340:~/learnGit (master)$ git log
commit 6650df264bfb2835e9236093975186e89fe030cf (HEAD -> master)
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:26:19 2024 +0800

    3

commit cbcbd189f03c2d2c0f8cb569e79b4699f1e24046
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:26:02 2024 +0800

    2

commit 6186ecd8ec2f802f71371768ca3753ff0c088b2e
Author: 詹佳博 <22373340@buaa.edu.cn>
Date:   Fri Mar 8 01:25:41 2024 +0800

    1
```

Thinking 0.5

- `echo` 使得后面的参数回显。
  - 在 `first` 时，回显到屏幕；
  - 在 `second` 时，重定向输入到 `output.txt`；
  - 在 `third` 时，`>` 重定向输入到 `output.txt` 并且覆盖原文件；
  - 在 `forth` 时，`>>` 重定向输入到 `output.txt` 并且追加原文件；

```
git@22373340:~/learnOSome $ echo first
first
git@22373340:~/learnOSome $ echo second > output.txt
git@22373340:~/learnOSome $ echo third > output.txt
git@22373340:~/learnOSome $ echo forth >> output.txt
git@22373340:~/learnOSome $
```

```
1 third
2 forth
```

Thinking 0.6

- `command` 文件内容如下：

```
1 echo echo Shell Start... > test
2 echo echo set a = 1 >> test
3 echo a=1 >> test
4 echo echo set b = 2 >> test
5 echo b=2 >> test
6 echo echo set c = a+b >> test
7 echo 'c=$[$a+$b]' >> test
8 echo 'echo c = $c' >> test
9 echo echo save c to ./file1 >> test
10 echo 'echo $c>file1' >> test
11 echo echo save b to ./file2 >> test
12 echo 'echo $b>file2' >> test
13 echo echo save a to ./file3 >> test
14 echo 'echo $a>file3' >> test
15 echo echo save file1 file2 file3 to file4 >> test
16 echo 'cat file1>file4' >> test
17 echo 'cat file2>>file4' >> test
18 echo 'cat file3>>file4' >> test
19 echo echo save file4 to ./result >> test
20 echo 'cat file4>>result' >> test
```

- 运行 `command`。

- 运行 `test`。
- `result` 文件内容如下：

```
1 3
2 2
3 1
```

分析 `test` 代码（直接影响 `result` ）：

- 首先，有一些 `echo` 指令在屏幕上输出伪代码；但是，有一些 `echo` 指令包含重定向符号，意味着将其后的内容**输入到重定向文件**中，如果还包含 `$` 符号，则就是将它的 *值* 代入。
- 而没有 `echo` 的指令，紧跟在 `echo` 后，是真正的在**shell**中执行了、赋值了的指令；如果还包含 `$` 符号，则就是将它的 *值* 代入。
- 以上分析即包含 `test` 所有行分析。以下为影响 `result` 的代码。

```
a=1
b=2
c=$[$a+$b]
echo $c>file1
echo $b>file2
echo $a>file3
cat file1>file4
cat file2>>file4
cat file3>>file4
cat file4>>result
```

分析 `command` 代码（直接影响 `test`）：

- 所有的 `echo` 只是为了将后面所有东西输入到 `test` 文件，故而为字符串形式；需要区分的是，如果有一些包含了**重定向符号**及 `$` 符号等，会真实改变输出字符串形式，则用单引号括起来，保证其为原字符串输出。

echo echo Shell Start 与 echo `echo Shell Start` 效果有区别，前者是`echo Shell Start`，后者是`Shell Start`；

echo echo $c>file1 与 echo `echo $c>file1` 效果有区别，前者是将**echo <c的值>**输入到 `file1` 中，后者是将**<c的值>**输入到 `file1` 中。

难点分析

1. vim 太难用了，但是通过调整 .vimrc 文件使得显示变得让自己舒适了一些；在这几次使用中也慢慢稍微习惯了 vim 编写代码的用法。
2. 编写 Makefile 有时候逻辑顺序和依赖关系理不清，包括对 shell 命令的不熟悉也使得有时候错误较多；通过画依赖图来辅助编写 Makefile。
3. shell 编程有点像之前学的 python ，但是比python更难用。有时候对单引号、双引号和反引号的用法容易混淆。还有一些美元符和重定向符号的影响。
4. shell 命令太多，实在记不清楚；有时候必须得查阅手册来明确用法。
5. Makefile 有更加高级的 make 用法，目前还不能完全理解和掌握；同时，对 gcc 编译时，若已有 .o 文件，此时用 ld 链接好似还需要很复杂的一些参数，使用 gcc 则可以省一大笔事。这一点在课上完成 Exercise 0.4 困惑了很久很久。
6. git 操作的时候，由于不熟悉，有时候会忽略自己之前的改动，导致保留了自己不想改动的操作。
7. **上机**：如果循环较为复杂，涉及多管道和复杂指令合成，需要逐步写代码并仔细分析。

实验体会

1. 学习了基本的 shell 用法，会通过各种指令对文件进行操作。
2. 学习了基本的 gcc 编译，会进行编译、链接来构建可执行文件
3. 后悔没有好好的学习 man 的用法，只知道用教材上仅有的一些指令参数进行编程。考试还剩3分钟时通过 man 指令查询了grep的更多的一些用法，总算通过循环和查找完成了 extra 测试。以后要熟练学习使用 man！