1. 思考题

Thinking 0.1 思考下列有关 Git 的问题：

> • 在/home/21xxxxxx/learnGit（已初始化）目录下创建一个名为 README.txt的文件。执行命令 git status > Untracked.txt。 • 在 README.txt 文件中添加任意文件内容，然后使用 add 命令，再执行命令git status > Stage.txt。 • 提交 README.txt，并在提交说明里写入自己的学号。 • 执行命令 cat Untracked.txt 和 cat Stage.txt，对比两次运行的结果，体会 README.txt 两次所处位置的不同。 • 修改 README.txt 文件，再执行命令 git status > Modified.txt。 • 执行命令 cat Modified.txt，观察其结果和第一次执行 add 命令之前的 sta□tus 是否一样，并思考原因。

解答：

1. 在已初始化的目录下创建新文件 `README.txt` 后，此文件在文件夹中但并没有加入到git库，不参与版本控制，状态为 `Untracked` ，执行命令 `git status` 后显示：

```
位于分支 learnGit

尚无提交

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）
```

2. 执行命令 `git add README.txt` 后，文件变为暂存状态 `Staged`，即已经把改文件放在下次提交(commit)时要保存的清单中。执行命令 `git status` 后显示：

```
位于分支 learnGit

尚无提交

要提交的变更:
   (使用 "git rm --cached <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)
        Untracked.txt
```

3. 执行 `git commit README.txt -m "21373007"` 后，将文件同步到库中。这时库中的文件和本地文件变为一致，文件处于未修改 `Unmodify` 状态。

4. 修改 README.txt 文件，文件变为修改 `Modified` 状态，执行 `git status` 后显示：

```
位于分支 learnGit
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)
         Stage.txt
         Untracked.txt
```

Thinking 0.2 仔细看看0.10，思考一下箭头中的 add the file 、stage the file 和 commit 分别对应的是 Git 里的哪些命令呢?

1. **Add the file**: `git add <filename>`
2. **Stage the file**: `git add <filename>`
3. **Commit**: `git commit <filename> -m "xxx"`

Thinking 0.3 思考下列问题：

> 1. 代码文件 print.c 被错误删除时，应当使用什么命令将其恢复?
> 2. 代码文件 print.c 被错误删除后，执行了 git rm print.c 命令，此时应当使用什么命令将其恢复?
> 3. 无关文件 hello.txt 已经被添加到暂存区时，如何在不删除此文件的前提下将其移出暂存区?

解答：

1. 在工作区误删除了 `print.c`， 如果 `print.c` 在暂存区，则可以执行 `git restore print.c` 将其恢复（本质是用暂存区的文件替换工作区的文件）；否则执行 `git checkout -- print.c` （本质是用已修改区的文件替换工作区的文件）
2. 在工作区误删了 `print.c`，同时执行了 `git rm print.c` ，即把暂存区的文件也删除了，则可以执行 `git restore --staged print.c`将文件恢复到暂存区（本质是将暂存区的文件还原到上一次提交时的状态），然后执行 `git restore print.c`恢复文件。
3. 执行 `git rm --cached print.c`。

Thinking 0.4 思考下列有关 Git 的问题：

> • 找到在/home/21xxxxxx/learnGit 下刚刚创建的 README.txt 文件，若不存在则新建该文件。 • 在文件里加入 Testing 1，git add，git commit，提交说明记为 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 命令查看提交日志，看是否已经有三次提交，记下提交说明为 3 的哈希值a。 • 进行版本回退。执行命令 git reset --hard HEAD^后，再执行 git log，观察其变化。 • 找到提交说明为 1 的哈希值，执行命令 git reset --hard 后，再执行 git log，观察其变化。 • 现在已经回到了旧版本，为了再次回到新版本，执行 git reset --hard ，再执行 git log，观察其变化。

解答： 执行 `git log` 后显示：

```
commit 8c2535838c47edfb3eef83c12bb4ee24d6db4026 (HEAD -> learnGit)
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:12:51 2023 +0800

    3

commit 357e1c03818ba1a9cd0cdce480af2a178bc7b421
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:12:24 2023 +0800

    2

commit 614d23f3c453ea7da408853a4fe39ed5bf62e886
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:11:45 2023 +0800

    1
```

执行 `git reset --hard HEAD^` 后，日志变为

```
commit 357e1c03818ba1a9cd0cdce480af2a178bc7b421 (HEAD -> learnGit)
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:12:24 2023 +0800

    2

commit 614d23f3c453ea7da408853a4fe39ed5bf62e886
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:11:45 2023 +0800

    1
```

执行 `git reset --hard 614d23f3c453ea7da408853a4fe39ed5bf62e886` 后，日志变为

```
commit 614d23f3c453ea7da408853a4fe39ed5bf62e886 (HEAD -> learnGit)
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:11:45 2023 +0800

    1
```

执行 `git reset --hard 8c2535838c47edfb3eef83c12bb4ee24d6db4026` 后，日志变为

```
commit 8c2535838c47edfb3eef83c12bb4ee24d6db4026 (HEAD -> learnGit)
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:12:51 2023 +0800
```

```
    3

commit 357e1c03818ba1a9cd0cdce480af2a178bc7b421
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:12:24 2023 +0800

    2

commit 614d23f3c453ea7da408853a4fe39ed5bf62e886
Author: 苏云鹤 <21373007@buaa.edu.cn>
Date:   Wed Mar 1 18:11:45 2023 +0800

    1
```

Thinking 0.5 执行如下命令, 并查看结果

1. echo first shell输出：

```
first
```

2. echo second > output.txt output.txt中的内容为：

```
second
```

3. echo third > output.txt output.txt中的内容为：

```
third
```

4. echo forth >> output.txt output.txt中的内容为：

```
third
forth
```

Thinking 0.6 使用你知道的方法（包括重定向）创建下图内容的文件（文件命名为 test），将创建该文件的命令序列保存在 command 文件中，并将 test 文件作为批处理文件运行，将运行结果输出至 result 文件中。给出 command 文件和 result 文件的内容，并对最后的结果进行解释说明（可以从 test 文件的内容入手）. 具体实现的过程中思考下列问题: echo echo

Shell Start 与 echo `echo Shell Start`效果是否有区别; echo echo$c>file1 与 echo `echo $c>file1`效果是否有区别.

文件内容： command文件：

```
echo echo Shell Start...>test
echo echo 'set a = 1'>>test
echo a=1>>test
echo echo 'set b = 2'>>test
echo b=2>>test
echo echo 'set c = a+b'>>test
echo 'c=$[$a+$b]'>>test
echo echo c = '$c'>>test
echo echo save c to ./file1>>test
echo echo '$c>file1'>>test
echo echo save b to ./file2>>test
echo echo '$b>file2'>>test
echo echo save a to ./file3>>test
echo echo '$a>file3'>>test
echo echo save file1 file2 file3 to file4>>test
echo cat file1'>'file4>>test
echo cat file2'>>'file4>>test
echo cat file3'>>'file4>>test
echo echo save file4 to ./result>>test
echo cat file4'>>'result>>test
```

test文件：

```
echo Shell Start...
echo set a = 1
a=1
echo set b = 2
b=2
echo set c = a+b
c=$[$a+$b]
echo c = $c
echo save c to ./file1
echo $c>file1
echo save b to ./file2
echo $b>file2
echo save a to ./file3
echo $a>file3
echo save file1 file2 file3 to file4
cat file1>file4
cat file2>>file4
cat file3>>file4
echo save file4 to ./result
cat file4>>result
```

result文件：

```
Shell Start...
set a = 1
set b = 2
set c = a+b
c = $[1+2]
save c to ./file1
save b to ./file2
save a to ./file3
save file1 file2 file3 to file4
save file4 to ./result
$[1+2]
2
1
```

分析： 单引号：会剥夺内部所有字符的特殊含义。 反引号：含义同$()，在执行命令时，会先将其中的``中的语句执行一遍，然后再将结果加入到原命令中重新执行。

---

2. 难点分析

exercise 0.1

sed指令： sed 是一个文件处理工具，可以将数据行进行增删改查等工作。 sed的运行模式是：当处理数据时，sed 从输入源一次读入一行，并将它保存到所谓的模式空间pattern space中。所有 sed 的变换都发生在模式空间。变换都是由命令行上或外部 Sed 脚本文件提供的单字母命令来描述的。大多数 sed 命令都可以由一个地址或一个地址范围作为前导来限制它们的作用范围。

用`sed -n`指令可以控制输出的行数。

cp指令： cp 指令用于复制文件。`cp <path1> <path2>` 可以把文件从 path2 复制到 path1。

---

exercise 0.2

change.sh文件的内容为：

```
#!/bin/bash
a=1
while [ $a -le 100 ]
do
    if [ $a -gt 70 ]       #if loop variable is greater than 70
    then rm -rf file$a

    elif [ $a -gt 40 ]         # else if loop variable is great than 40
    then mv file$a newfile$a
```

```
    fi
            #don't forget change the loop variable
        a=$[$a+1]
done
```

shell脚本文件：

- 开头需要写`#!/bin/bash`
- 变量定义和赋值可以直接写，如`a=1`
- 变量自增：`$[$a+1]`

---

exercise 0.3

管道： 管道 `|`可以把前序指令的结果作为后续指令的输入。比如，对于本题的要求，需要先使用 `grep` 找到对应的行，再用 `awk` 处理输出文字的格式，这时便可以用管道连接这两个指令，具体写法为：

```
#!/bin/bash
#First you can use grep (-n) to find the number of lines of string.
#Then you can use awk to separate the answer.
grep -n $2 $1 | awk -F: '{print $1}' > $3
```

grep指令： grep是文本搜索功能，可以使用正则表达式搜索文本，并把匹配的行打印出来。用法是`grep [选项]... PATTERN [FILE]...`。-n可以显示行号，-i可以忽略大小写差异。

awk指令： awk是处理文本文件的语言，可以用`awk -F: '{print $1}'`指定分隔符为冒号，并输出被分隔的第一个元素。

exercise 0.4

make文件：

- .PHONY（伪目标）：
  - 被 .PHONY 标记的目标并不代表一个真正的文件名，被其定义的命令一定会执行。
  - 如果我们制定的目标不是创建目标文件，而是使用makefile执行一些特定的命令，比如`clean`，那么我们希望只要输入`make clean`后这条指令一定会执行。但是，如果目录存在一个和指定目标重名的文件clean时，结果便不是我们想要的，这条指令一定不会被执行。
  - 解决办法：便是把目标clean定义为伪目标。
  - 伪目标还可以提高make的执行效率，因为make的执行程序在执行的时候不会试图寻找clean的隐含规则。

> .PHONY修饰的目标就是只有规则没有依赖。

---

3. 实验体会

1. 初步掌握了linux的使用；
2. 体会到了os实验的魅力~