Lab0 实验报告

零、实验目的

1. 认识操作系统实验环境。
2. 掌握操作系统实验所需的基本工具。

在本实验中，需要了解实验环境，熟悉 Linux 操作系统（Ubuntu），了解控制终端，掌握一些常用工具并能脱离可视化界面进行工作。本实验难度不大，重点是熟悉操作系统实验环境的各类工具，为后续实验的开展奠定基础。

一、思考题

Thinking 0.1 - Git 的使用 1

> 思考下列有关 Git 的问题：
> • 在前述已初始化的 `~/learnGit` 目录下，创建一个名为 README.txt 的文件。执行命令 `git status > Untracked.txt`（其中的 `>` 为输出重定向，我们将在 0.6.3 中详细介绍）。

前述操作：在主目录 `~` 下，使用 `mkdir learnGit` 命令创建一个名为 `learnGit` 的目录（`touch` 是创建新文件）；执行 `cd learnGit` 进入该目录；输入 `git init` 初始化 Git 仓库；使用命令 `ls -a` 可以看到该目录下有一个名为 `.git` 的隐藏目录，这个目录就是 Git 版本库，称为仓库（repository）。现在 `learnGit` 目录就是 Git 里的工作区。
创建 README 文件：在 `learnGit` 目录下，执行命令 `touch README.txt`。
查看未跟踪文件状态：执行命令 `git status > Untracked.txt`，现在该目录下增加了 Untracked.txt 文件。

> • 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`。

修改 README 文件：使用命令 `vim README.txt` 进入 vim 界面，按 `i` 键进入插入模式，添加字符串 `hello world!`，按 `Esc` 键退出插入模式，按 `Shift` + `:`，再输入 `wq` 后保存并退出。
跟踪文件：执行命令 `git add README.txt`，现在该文件从未跟踪（Untracked）变成了已暂存（Staged）。
查看已暂存文件状态：执行命令 `git status > Stage.txt`，现在该目录下增加了 Stage.txt 文件。

> • 提交 README.txt，并在提交说明里写入自己的学号。

提交文件：执行命令 `git commit -m "23373179" README.txt`，提交该文件，提交说明为学号。

```
git@23373179:~/learnGit $ git commit -m "23373179" README.txt
[master （根提交） 511250b] 23373179
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
git@23373179:~/learnGit (master)$
```

> • 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。

```
git@23373179:~/learnGit (master)$ cat Untracked.txt
位于分支 master

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）
```

```
git@23373179:~/learnGit (master)$ cat Stage.txt
位于分支 master

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）
        Stage.txt
        Untracked.txt
```

可以看到，执行 `add` 命令后，README.txt 文件从未跟踪（Untracked）变成了已暂存（Staged）。此时 README.txt 处于一种等待被提交的状态。

> • 修改 README.txt 文件，再执行命令 `git status > Modified.txt` 。

修改文件：使用同样的方法，我在文件第二行加入了字符串 `hello os!` 。
查看已修改文件状态：执行命令 `git status ? Modified.txt` ，现在该目录下增加了 Modified.txt 文件。

> • 执行命令 `cat Modified.txt` ，观察其结果和第一次执行 `add` 命令之前的 status 是否一样，并思考原因。

```
git@23373179:~/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"）
```

不一样。在第一次 `add` 之前，README.txt 是未跟踪的；在 `add` 后，即使 README.txt 已被更改，但依然是被跟踪的，处于已修改（Modified）状态，但未被加入暂存区，需要重新 `add` 和 `commit` 。

Thinking 0.2 - 箭头与命令

> 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?

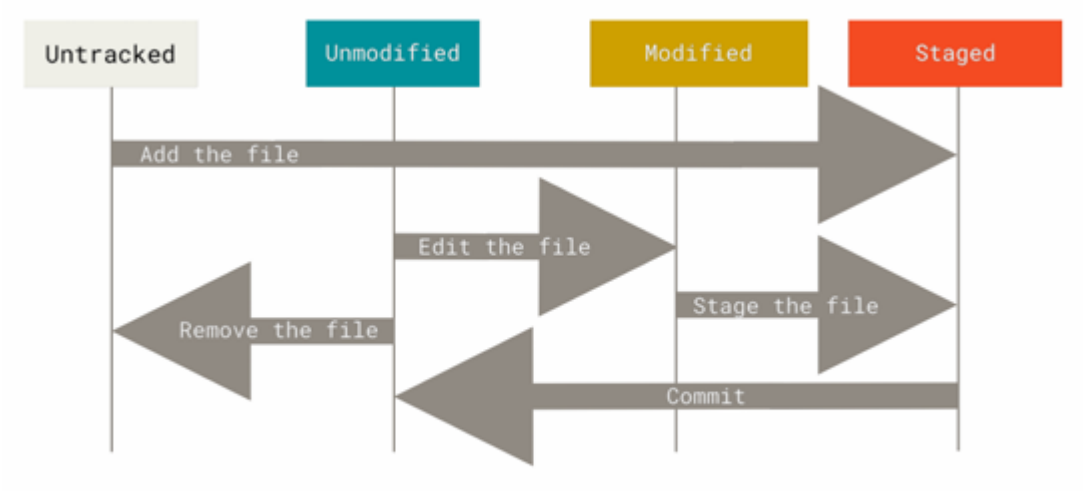

图 0.10: Git 中的四种状态转换关系

- add the file: `git add <file>`
- stage the file: `git add <file>`
- commit: `git commit <file>`

Thinking 0.3 - Git 的一些场景

> 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？

`git checkout -- print.c` ：该命令可以将暂存区的文件恢复到工作区中。

> 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？

`git reset HEAD print.c` ：该命令撤销了暂存区的更改（即 `git rm` 操作），但文件不在工作区中。

> 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？

`git reset HEAD hello.txt` ：同上，撤销了暂存区的更改。

Thinking 0.4 - Git 的使用 2

> 思考下列有关Git的问题：
> • 找到在 `/home/22xxxxxx/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。

（略）。

> • 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。

```
git@23373179:~/learnGit (master)$ git log
commit e1e1df7629b9add1297209975f46c94e11580096 (HEAD -> master)
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:15:06 2025 +0800

    3

commit 6483b07ed120d8a11f494b0542a781cf5221eadd
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:53 2025 +0800

    2

commit ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:01 2025 +0800

    1
```

提交说明为 3 的哈希值为：`e1e1df7629b9add1297209975f46c94e11580096`

> • 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。

```
git@23373179:~/learnGit (master)$ git reset --hard HEAD^
HEAD 现在位于 6483b07 2
git@23373179:~/learnGit (master)$ git log
commit 6483b07ed120d8a11f494b0542a781cf5221eadd (HEAD -> master)
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:53 2025 +0800

    2

commit ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:01 2025 +0800

    1
```

发现已不存在提交说明为 3 的提交。（`HEAD^` 表示上个版本，`HEAD^^` 是上上个版本，`HEAD~50` 是前 50 个版本）

> • 找到提交说明为 1 的哈希值，执行命令 `git reset --hard <hash>` 后，再执 行 `git log`，观察其变化。

提交说明为 1 的哈希值为： `ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27`
执行命令 `git reset --hard ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27`

```
git@23373179:~/learnGit (master)$ git reset --hard ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27
HEAD 现在位于 ba583f5 1
git@23373179:~/learnGit (master)$ git log
commit ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27 (HEAD -> master)
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:01 2025 +0800

    1
```

发现现在仅存在提交说明为 1 （及之前）的提交。

> • 现在已经回到了旧版本，为了再次回到新版本，执行 `git reset --hard <hash>` ，再执行 `git log`，观察其变化。

使用提交说明为 3 的哈希值，发现提交 2 与 3 都再次出现。

```
git@23373179:~/learnGit (master)$ git reset --hard e1e1df7629b9add1297209975f46c94e11580096
HEAD 现在位于 e1e1df7 3
git@23373179:~/learnGit (master)$ git log
commit e1e1df7629b9add1297209975f46c94e11580096 (HEAD -> master)
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:15:06 2025 +0800

    3

commit 6483b07ed120d8a11f494b0542a781cf5221eadd
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:53 2025 +0800

    2

commit ba583f536ac0fa9ccb3f0e4b2c98a9466cf77e27
Author: 周子强 <23373179@buaa.edu.cn>
Date:   Mon Mar 10 22:14:01 2025 +0800

    1
```

Thinking 0.5 - `echo` 的使用

> 执行如下命令, 并查看结果
> • `echo first`

终端直接显示出 `first`。

> • `echo second > output.txt`

当前目录下出现新的文件 `output.txt` ，其内容为 `second` 。

> - `echo third > output.txt`

`output.txt` 中的内容直接被替换成 `third` 。

> - `echo forth >> output.txt`

`output.txt` 中的内容增加了一行 `forth` ，之前的 `third` 被保留。

Thinking 0.6 - 文件的操作

> 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 <撇>echo Shell Start<撇>` 效果是否有区别; `echo echo $c>file1` 与 `echo <撇>echo $c>file1<撇>` 效果是否有区别.
>
> ```
> echo Shell Start...
> echo set a = 1
> a=1
> echo set b = 2
> b=2
> echo set c = a+b
> c=$[$a+$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
> ```
>
> 图 0.14: 文件内容

`command` 内容:

```
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
```

执行命令 `chmod +x test` 以及 `./test > result`
`result` 内容：

```
Shell Start...
set a = 1
set b = 2
set c = a+b
c = 3
save c to ./file1
save b to ./file2
save a to ./file3
save file1 file2 file3 to file4
save file4 to ./result
3
2
1
```

解释：
`a=1` 命令对 `a` 赋值； `c=$[$a+$b]` 命令将 a 和 b 的值相加赋给 `c` ； `echo c = $c` 命令输出 `c` 的值； `echo $c>file1` 将 `c` 的值重定向输入给 `file1` ； `cat` 则是将文件内容输入给其他文件；
`result` 文件最后出现的三行是将 `file4` 拼接到 `result` 末尾的结果。
加撇号：会先执行符号内部的指令。

二、难点分析

命令行界面（CLI）

在之前的学习中，用的更多的是图形用户界面（GUI），接触过少部分 CLI 如 cmd、Power Shell 等等。CLI 最大的特点是不直观，但足够简洁、直击本质，需要一些时间适应和熟练。

Linux 基本操作命令

命令	选项
`ls [选项] [文件]`	`-a` ： 不隐藏以 `.` 开始的项目 `-l` ： 列出详细信息，每行只列一个文件
`touch [文件名]`	创建文件
`mkdir 目录`	创建目录
`cd 目录`	进入目录。返回上一级目录 `cd ..`
`rmdir 目录`	删除空目录
`rm [选项] 文件`	`-r` ： 递归删除目录及其内容，删除非空目录 `-f` ： 强制删除 `-i` ： 删除前询问
`cp [选项] 源文件 目录`	`-r` ： 递归复制目录及其子目录的所有内容
`mv 源文件 目录`	移动文件
`echo`	回显
`man`	查看帮助

快捷键	说明
Ctrl+C	终止当前程序的执行
Ctrl+Z	挂起当前程序
Ctrl+D	终止输入
Ctrl+L	清屏

Vim

文本编辑器，具体用法在思考题中已经提到。在 CLI 中非常好用。

GCC

即 GNU 编译器套件，包含了 C 语言编译器 gcc。

- 将多个文件进行编译链接：`gcc a.c b.c -o test`
- 将每个文件单独编译，再进行链接：`gcc -c a.c && gcc -c b.c && gcc a.o b.o -o test`
- 执行可执行文件：`./test`

命令	选项
`gcc [选项] [C 源文件]`	`-o` ：自动生成的输出文件 `-S` ：将 C 代码转换为汇编代码 `-Wall` ：显示警告信息 `-c` ：仅执行编译操作，不进行链接操作 `-M` ：列出依赖 `-I<path>` ：编译时指定头文件目录

Makefile

格式：

```
target: dependencies
    command 1
    command 2
    ...
    command n
```

例子：

```
all: hello_world.c
    gcc -o hello_world hello_world.c
clean:
    rm -f helloworld
```

直接执行命令 `make` 或 `make all` 将 `hello_world.c` 编译成可执行文件；执行命令 `make clean` 删除 `helloworld` 可执行文件。

Git

思考题中已经涉及，不再赘述。

Linux 操作补充

命令	说明
`find -name 文件名`	递归地查找符合参数所示文件名的文件，输出文件路径
`grep [选项]`	用正则表达式搜索文件所含文本，输出目标行 `-a` ：不忽略二进制数据进行搜索 `-i` ：忽略文件大小写差异

命令	说明
`PATTERN [FILE]`	`-r` ：从目录递归查找 `-n` ：显示行号
`tree [选项] [目录名]`	生成文件树 `-a` ：列出全部文件 `-d` ：只列出目录
`chmod 权限设定字符串 文件`	`+x` ：可执行
`diff [选项] file1 file2`	比较文件的差异 `-b` ：不检查空白字符 `-B` ：不检查空行 `-q` ：仅显示有无差异，不显示详细信息
`sed [选项] 命令 输入文本`	将数据行进行替换、删除、新增、选取 **选项**: `-n` ：安静模式，只显示经过处理的内容 `-i` ：直接修改读取档案内容，不输出到屏幕 **命令**: `[行号] a\ [内容]` ：新增，行号后新增一行内容 `[行号] c\ [内容]` ：取代 `[行号] i\ [内容]` ：插入，上一行插入 `[行号] d` ：删除 `[行号] p` ：输出 `s/re/string` ：将 `re` 正则表达式匹配的内容替换为 `string` **例子**: `Shell # 输出第三行 sed -n '3p' my.txt # 删除第二行 sed '2d' my.txt # 删除第二行到最后一行（$ 表示末尾） sed '2,$d' my.txt # 在整行范围内把 str1 替换为 str2 # 如果没有 g 标记，则只有每行第一个匹配的被替换 sed 's/str1/str2/g' my.txt # -e 选项允许同一行里执行多条命令 # 第四行后添加一个 str，再替换成 aaa sed -e '4a\str' -e 's/str/aaa' my.txt `
`awk`	`awk '$1>2 {print $1,$3}' my.txt` ：输出文件中所有第一项大于 2 的行的第一项和第三项 `awk -F, '{print $2}' my.txt` ：用 `,` 分割

tmux

终端复用软件。

shell 脚本

创建 `my.sh` 文件，内容例如：

```
#!/bin/bash
# say something...
echo "Hello World!"
```

运行脚本：`bash my.sh` 或 `chmod +x my.sh` 后 `./my.sh`
脚本传参：

```
#!/bin/bash
echo $1
```

执行命令 `./my2.sh msg` 。 `$n` 表示第几个参数。 `$#` 传递参数的个数， `$*` 一个字符串显示传递的全部参数， `$?` 获取前一个命令的返回值。
函数传参：

```
# 定义函数，function 与 () 可省其一
function < 函数名 > () {
    <commands>
}
# 调用函数
< 函数名 > <args>...
```

流程控制语句：
if 语句

```
# 格式 1
if <condition>
then
    <command1>
    <command2>
    # ...
fi
# 格式 2
if condition; then command1; command2; ... fi
# 例如
a=1
if [ $a -ne 1 ]; then echo ok; fi
```

`<condition>` 也是命令，返回值为 0 时成功执行，条件成立。

关系运算符	说明
`-eq`	==
`-ne`	!=
`-gt`	>
`-lt`	<
`-ge`	>=
`-le`	<=

while 语句

```
# 格式
while <condition>
do
    <commands>
done
# 例子
a=1
while [ $a -ne 10 ]
do
    mkdir file$a
    a=$[$a+1]
done
```

重定向和管道

标准输入：`stdin`，`0`
标准输出：`stdout`，`1`
标准错误：`stderr`，`2`
重定向输出：`>`，重定向输入：`<`，重定向输出追加：`>>`
管道：`|` 前者输出发给后者

三、实验体会

Lab0 的指导书所列出的知识点是庞杂的，OS 的学习不能只停留于理论，而要动手实操，这样不仅能加深对理论的记忆，也能训练动手设计能力。本实验学习的内容是非常有用的，其中 Linux、Git、Makefile、GCC 等知识应该熟记于心。
Lab0 影响较为深刻的部分：Exercise 0.4 中，可能需要嵌套调用 Makefile 命令，但我使用列出具体地址的方法只用到了外层 Makefile。同时，gcc 命令寻找头文件的方法靠网络搜索得出，即添加选项 `-I` 。