

林子杰_20373980_lab0

一、思考题

Thinking 0.1

第一次 `add` 之前, `README.txt` 文件虽然已经存在, 但是其从来没有被 `git add` 过, 故而不参与版本控制, 其返回信息和 `Untracked` 即"未跟踪"有关。

修改文件内容后, 由于该文件已经进入版本库接受管理, 故而不会再被标记为 `Untracked`, 而是会被标记为 `Modified`。文件入库后被修改但没有 `git add` 时, 会被标记为这种状态。

Thinking 0.2

这里只写出了指令的前半部分, 具体的文件选项不写出。

add the file:对应的是 `git add`

stage the file:对应的也是 `git add`

commit:对应的指令是 `git commit`

Thinking 0.3

- 1.使用 `git checkout -- printf.c` 可将该文件恢复, 即退回版本。
- 2.使用 `git reset HEAD printf.c` 可以恢复暂存区的内容, 退回删除操作。
- 3.使用 `git rm --cached Tucao.txt` 可将其从暂存区移除, 这样就不会在commit的时候提交了。

Thinking 0.4

在每一次执行 `git commit` 之后, 都会在最终的版本库保存一个版本。了解到, Git总是使用一个HEAD指针指向最新的版本的位置。`git reset` 的作用就是修改HEAD指向的版本, 通过版本之间的切换进行版本调用。

Thinking 0.5

- 1.错误。经查他人的操作, 只有HEAD分支被检出, 如想在其他分支上工作, 我们还需要创建一个本地分支。

一. git克隆下来只有master分支，切换其它分支

1. 当我们 `git clone + 远程仓库地址` 下来代码之后，`git branch` 发现只有master分支，而我们大多数时候都是在其它分支处理事情的，所以我们用`git branch -a` 查看所有分支

```
$ git branch -a
* dev_1.5
master
remotes/origin/HEAD -> origin/master
remotes/origin/dev
remotes/origin/dev_1.2
remotes/origin/dev_1.3
remotes/origin/dev_1.4
remotes/origin/dev_1.5
remotes/origin/master
```

2. 上图我已经切换到非master的dev_1.5分支,已经正式使用，要想达到这个最终目的，我们只需要
`git checkout -t origin/xxx` (xxx指你要切换的分支名,比如我的就是dev_1.5)
3. 现在 `git branch` 查看一下，大功告成

```
$ git branch
* dev_1.5
master
```

- 2.正确。即便在之前没有使用远程仓库时，Git版本状况都可以通过这几条指令查看。
- 3.正确。如上，仅仅是HEAD分支被检出且克隆到本地；我们虽然能查看其他分支的信息，但其并不在本地。
- 4.正确。master分支是默认的HEAD分支。

Thinking 0.6

```
echo first //在标准IO输出"first"
echo second > output.txt //重定向输出"second"至文件output.txt
echo third > output.txt //重定向输出"third"至文件output.txt,会覆盖之前的内容
echo forth >> output.txt //重定向输出"forth"至文件output.txt,在原有基础之上追加
```

Thinking 0.7

command文件内容：

```
cho 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=[extract_itex]{a+[/extract_itex]b} >> test
echo echo c = [extract_itex]c >> test
echo echo save c to ./file1 >>test
echo echo[/extract_itex]c\>file1 >>test
echo echo save b to ./file2 >>test
echo echo[/extract_itex]b\>file2 >>test
echo echo save a to ./file3 >>test
echo echo[/extract_itex]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
```

result文件内容:

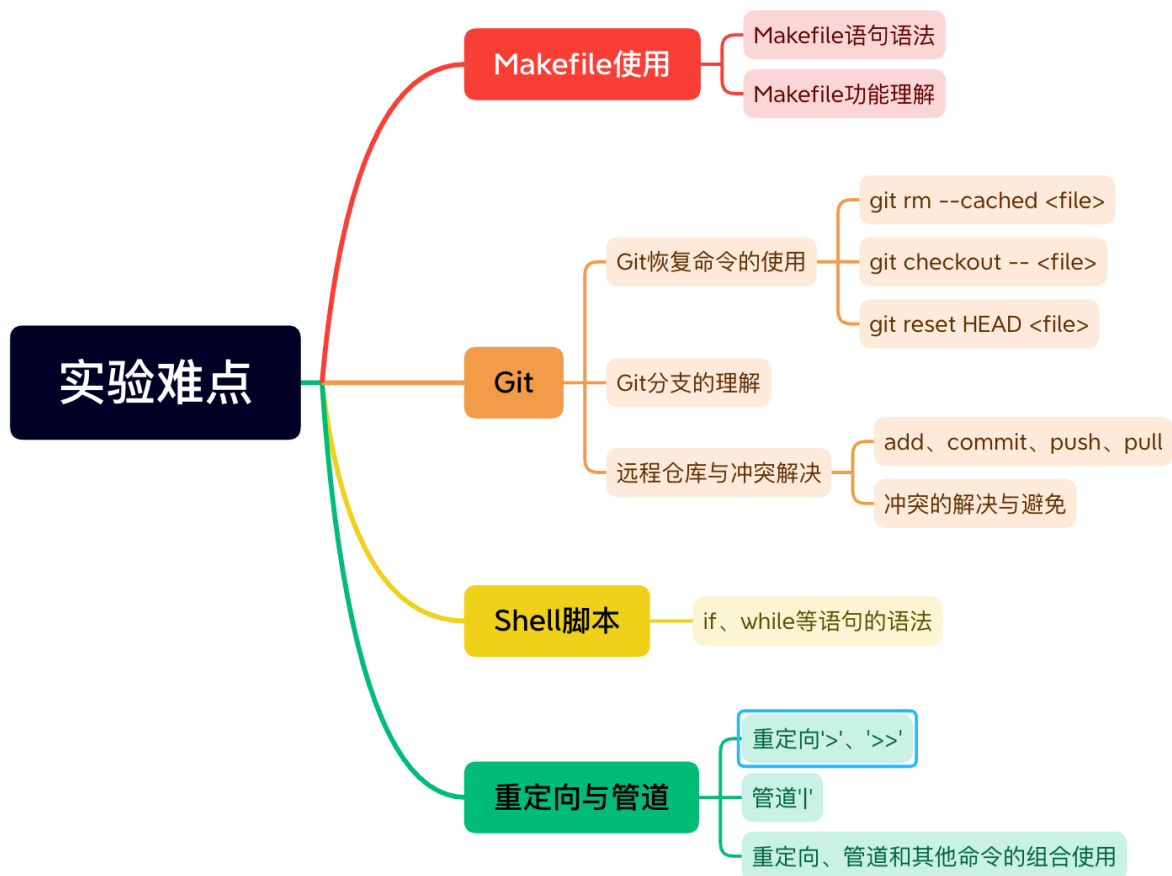
```
3
2
1
```

从command到test文件，仅仅是输出 `echo` 后的字符串，当然，需要给 `>` 添加转义符号。从test到result文件则有所不同，即 `$` 符号可以引用变量的值，而不再作为字符单独输出。

`echo echo Shell Start` 与 `echo 'echo Shell Start'` 效果没有区别；`echo echo \>file1` 与 `echo 'echo \>file1'` 效果有区别，前者将 `>` 输出到 `file1`，后者将 `\>file` 输出到 `file1`。

二、实验难点

使用思维导图归纳本次试验的难点。



Makefile: Makefile的语法比较丰富独特，我在学习的时候比较频繁地查询了语法规则，比较花费时间；另外，初次学习Makefile与Shell脚本，我还不是很能理解两者在功能上的差距——似乎他们都是一个打包脚本的工具。但经过查询资料，得知Makefile脚本与Shell脚本确实有紧密联系，Makefile包装了一些语句，使得其更便于编译连接。

Git: OO课程使得2006的同学在寒假就提前接触到了Git的基本使用，不过OS课程在Git的进阶操作上给出了更加便于理解的教程。其中，Git恢复机制、Git分支、远程仓库冲突的相关知识在初次接触时比较抽象且有难度，经过多次尝试使用，我对其也有了初步的了解。

Shell脚本: 基本语句的使用并不困难，不过其中的for语句、while语句的格式比较"奇特"，想要熟练使用需要适应一段时间。

重定向与管道: 重定向与管道在本次试验中属于比较灵活、比较复杂的一个部分，在课下实验与上机考试都给我造成了不小的麻烦。其设计的语句多、用法灵活、文件操作多，使得想要熟练使用比较困难。

三、体会与感想

总体来讲，lab0的课下实验与上机考试都比较友好，我在课下大约花费了6-8个小时学习。lab0很好地向我们普及了Linux命令行及基础操作、Linux下的实用工具以及Git的基本使用。教程的指导书写的非常详细且生动，并且提供了很多课外的学习资料，总体体验不错；当然，个人认为有些地方写的还是比较马虎，比如Git部分对三个目录的解释、部分的错别字等。总的来说，OS的初体验还是很不错的，相比OO来说的确是更好上手。本次实验报告没有写的太详细，就将lab0的学习笔记附上。

Lab0实验笔记

一、Git

Git基础操作

Git是一个版本控制工具，能够通过简单的命令记录、查看、删除、恢复文件版本。当电脑上安装好Git Bash后（Mac可以直接通过终端安装），我们就可以开始使用Git了。

Git能够将本地目录下的某个文件转化为能够管理的Git文件。当我们想使用Git管理一个文件及其子目录时，我们可以在终端（Windows为Git Bash，Mac为终端，统一称作终端）进入我们想要管理的文件后，输入下面的指令：

```
git init
```

使用这条命令后，我们的本地文件夹会生成一个隐藏的`.git`文件，这时版本控制工具Git已经接手这个文件夹了。

接下来，我们可以尝试在这个目录下做一些修改，比如创建一个`.txt`文件并且在里面写入`Hello World!`。每次作出修改，我们都可以使用下面两条指令进行版本修改提交。

```
git add .  
git commit -m "message"
```

注意，第二条指令的`message`处一定要有信息，否则会提交不成功。在第一次使用Git的时候如果输入这两条命令，可能会遇到ssh配置等相关问题。因为在OS之前的OO就已经记录过配置ssh密钥的方法，再次就不多赘述。

Git主要通过一个对象库和三个目录来进行版本控制与维护。我们所需要维护的每一个文件的内容都会被压缩成一个二进制文件，即一个Git对象，其文件名为一个长度为40个字符串的哈希值。

对象库只保存了文件信息，没有目录结构，Git通过三个目录来管理不同版本的文件。这三个目录分别叫做工作区、暂存区、版本库。工作区即是本地的文件夹目录；暂存区是一个目录，可以索引到某次修改的文件；版本库则存放了整个文件的某个版本。

我们执行 `git add .` 操作时，就是将所有修改过的文件送入暂存区（`.` 即是操作所有文件，我们也可以改为某个文件的名称，用来add我们想add的文件）。而执行 `git commit` 时，即是根据暂存区的目录树，将此次文件版本送入版本库。下面介绍几个Git指令。

```
git rm --cached <file>      //从暂存区删除指定文件，工作区不做修改
git reset HEAD              //将暂存区的目录替换为版本库当前最新的目录，工作区不做修改
git checkout --<file>       //使用暂存区的文件替换工作区的文件，这会将工作区未保存的修改覆盖！！
```

Thinking 0.1

第一次 `add` 之前，`README.txt` 文件虽然已经存在，但是其从来没有被 `git add` 过，故而不参与版本控制，其返回信息和 `Untracked` 即"未跟踪"有关。

修改文件内容后，由于该文件已经进入版本库接受管理，故而不会再被标记为 `Untracked`，而是会被标记为 `Modified`。文件入库后被修改但没有 `git add` 时，会被标记为这种状态。

Git文件状态

关于Git下的文件状态，第一个思考题给了我们一些启发。下面我们一一介绍Git中的四种文件状态。

未跟踪：某个文件没有执行过 `git add` 操作，不受Git版本管理。反之，则文件已经被跟踪。

未修改：某个被跟踪后文件从未被修改过，或是修改已经被提交。

已修改：某个文件被修改，但还未 `git add` 加入暂存区。

已暂存：某个文件已经放在下次 `git commit` 的清单中。

网络上有不少流程图可以用来描述不同文件状态之间、Git指令之间的关系。

Thinking 0.2

这里只写出了指令的前半部分，具体的文件选项不写出。

add the file:对应的是 `git add`

stage the file:对应的也是 `git add`

commit:对应的指令是 `git commit`

Git恢复机制

有时候，我们可能会错误地修改、删除一些文件。Git能够通过版本控制帮我们恢复这些文件（前提是文件已被跟踪）。下面先介绍四条相关的指令。

```
git rm --cached <file>      //从暂存区删除我们不想跟踪的文件
git checkout -- <file>      //将文件退回最近一次add的版本
git reset HEAD <file>      //将暂存区的顶部文件移除
git clean <file> -f         //可以移除工作区不需要的文件
```

经过尝试，可以更深刻地体会这四条指令的作用。

Thinking 0.3

- 1.使用 `git checkout -- printf.c` 可将该文件恢复，即退回版本。
- 2.使用 `git reset HEAD printf.c` 可以恢复暂存区的内容，退回删除操作。
- 3.使用 `git rm --cached Tucao.txt` 可将其从暂存区移除，这样就不会在commit的时候提交了。

刚刚介绍的指令是Git最基础的撤销指令，不过我们在使用的时候也需要小心——如何撤销撤销？如何撤销撤销撤销？……于是，在应对版本修改的时候，Git还提供了一个更加强大的指令：

```
git reset --hard
```

配合下面两条指令，我们可以做到"往返自然"。

```
git log      //查看git日志
git status   //查看git状态
```

在 `git log` 反馈的信息中，我们可以看到一串字母数字字符，这就是我们前面提过的哈希值。有了某一版本文件的哈希值，并且配合 `HEAD` 选项，就可以很好地进行版本控制。

```
git reset --hard HEAD^      //退回上一个版本
git reset --hard HEAD~$n    //退回上n个版本
git reset --hard <hash-code> //退回该哈希值对应的版本
```

Thinking 0.4

在每一次执行 `git commit` 之后，都会在最终的版本库保存一个版本。了解到，Git总是使用一个HEAD指针指向最新的版本的位置。`git reset` 的作用就是修改HEAD指向的版本，通过版本之间的切换进行版本调用。

Git分支

分支是Git的一大特性，在不同的分支上操作文件互不影响。这使得我们可以进行多线开发，而不必每次想要测试文件时就要新建目录来保存。通常情况下，我们都会有一个基础分支，我们可以基于这个基础分支创建一个新的分支。

```
git branch <branch-name>
```

新分支拷贝了一份版本文件，而不是和当前分支共用一套版本文件。我们有一些其他的分支操作。

```
git branch -d <branch-name>    //删除某个分支
git branch -a                  //查看所有的远程分支与本地分支
git checkout <branch-name>      //切换到某个分支
```

分支操作在实验中是经常使用的操作。

Git远程仓库与本地

远程仓库的工作原理和基本操作在我的另一篇OO笔记已有记录，这里不再多做介绍，只记录思考题。

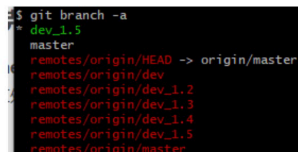
Thinking 0.5

1.错误。经查他人的操作，只有HEAD分支被检出，如想在其他分支上工作，我们还需要创建一个本地分支。

一. git克隆下来只有master分支，切换其它分支

1. 当我们 `git clone + 远程仓库地址` 下来代码之后，`git branch` 发现只有master分支，而我们大

多时候都是在其它分支处理事情的，所以我们用`git branch -a` 查看所有分支

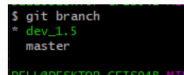


```
$ git branch -a
* dev_1.5
  master
remotes/origin/HEAD -> origin/master
remotes/origin/dev
remotes/origin/dev_1.2
remotes/origin/dev_1.3
remotes/origin/dev_1.4
remotes/origin/dev_1.5
remotes/origin/master
```

2. 上图我已经切换到非master的dev_1.5分支,已经正式使用，要想达到这个最终目的，我们只需要

`git checkout -t origin/xxx` (xxx指你要切换的分支名,比如我的就是dev_1.5)

3. 现在 `git branch` 查看一下，大功告成



```
$ git branch
* dev_1.5
  master
```

2.正确。即便在之前没有使用远程仓库时，Git版本状况都可以通过这几条指令查看。

3.正确。如上，仅仅是HEAD分支被检出且克隆到本地；我们虽然能查看其他分支的信息，但其并不在本地。

4.正确。master分支是默认的HEAD分支。

Git冲突与解决

假设我们在A电脑与B电脑对同一版文件进行了不同的修改，然后我们在A电脑将修改后的版本push到了远程仓库。这时候，如果你想把B电脑修改后的内容也push到远程仓库，你就会发现一件很尴尬的事情：远程仓库已经包含了一些B电脑本地仓库"不认识"的东西。这样就产生了冲突，必然是无法成功push的。我们可以使用下面命令：

```
git pull
```

拉取远程仓库的内容，可以帮助我们快速定位冲突产生的地方；之后要做的事情就是手动合并冲突内容。但这样也很麻烦，故而每次做修改之前，我们最好提前 `git pull` 一下。

Linux操作补充

常用的查找命令：find和grep

find命令：

在Linux终端下，我们在某个目录下输入如下命令，即可在当前目录及其子目录下寻找目标文件：

```
find -name <file>
```

Mac上的 `find` 命令有所不同，没有默认目录，我们需要手动输入。

```
find [catalogue] -name <file>
```

此外，我们还能对该命令使用重定向的操作，将其输出从标准IO转移。

grep命令：

`grep` 是一个强大的文本搜索命令，可以进行正则匹配。其基本用法为：

```
grep [option] PATTERN <file>
```

常用的选项有：-n，显示行号；-r，从文件中递归查找；-i，忽略大小写差异。`grep` 指令也可以重定向它的输入输出。

```
grep [option] PATTERN <file1> > <file2>  
grep [option] PATTERN <file1> >> <file2>
```

> 将抹去目标文件的内容，加载输入；>> 则将在最后一行插入。

文件树命令：tree

我们可以通过tree命令来查看当前文件下的目录树。进入某个文件，在终端输入：

```
tree
```

这样即可显示当前的文件目录。需要注意，Mac的zsh没有这一命令，你可以尝试安装brew，然后在终端输入 `brew install tree` 来加载 `tree` 命令。`tree` 命令的完整形式如下：

```
tree [option] [catalogue]
```

常用的选项有：-a，列出全部文件；-d，仅列出目录。

权限命令chmod:

权限命令的格式如下:

```
chmod [+|=][rwxX] <file>
```

我们可以给文件添加"+"、删除"-、设定唯一权限"="。"r"代表可读, "w"代表可写, "x"代表可执行。

差异比较命令diff:

diff 命令格式如下:

```
diff [option] <file1> <file2>
```

文件处理工具sed:

sed 命令的功能比较复杂, 我们只列举几个常见的用法。

```
sed -n '3p;4p' <filename>           //输出文件第3、4行的内容
sed 's/str1/str2/g' <filename>       //在整行范围内把str1替换为str2, g为全局标记
sed '2,$d' <filename>               //删除文件的第二行至最后一行
```

p、d 等是命令选项, 配合行号使用, 且中间要有分号间隔

Thinking 0.6

```
echo first                           //在标准IO输出"first"
echo second > output.txt             //重定向输出"second"至文件output.txt
echo third > output.txt               //重定向输出"third"至文件output.txt, 会覆盖之前的内容
echo forth >> output.txt              //重定向输出"forth"至文件output.txt, 在原有基础之上追加
```

Thinking 0.7

command文件内容:

```
cho 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
```

result文件内容:

```
3
2
1
```

从command到test文件，仅仅是输出 `echo` 后的字符串，当然，需要给 `>` 添加转义符号。从test到result文件则有所不同，即 `$` 符号可以引用变量的值，而不再作为字符单独输出。

`echo echo Shell Start` 与 `echo 'echo Shell Start'` 效果没有区别；`echo echo \>file1` 与 `echo 'echo \>file1'` 效果有区别，前者将 `>` 输出到 `file1`，后者将 `\>file` 输出到 `file1`。