Lab0

实验指导思考题

Thinking0.1

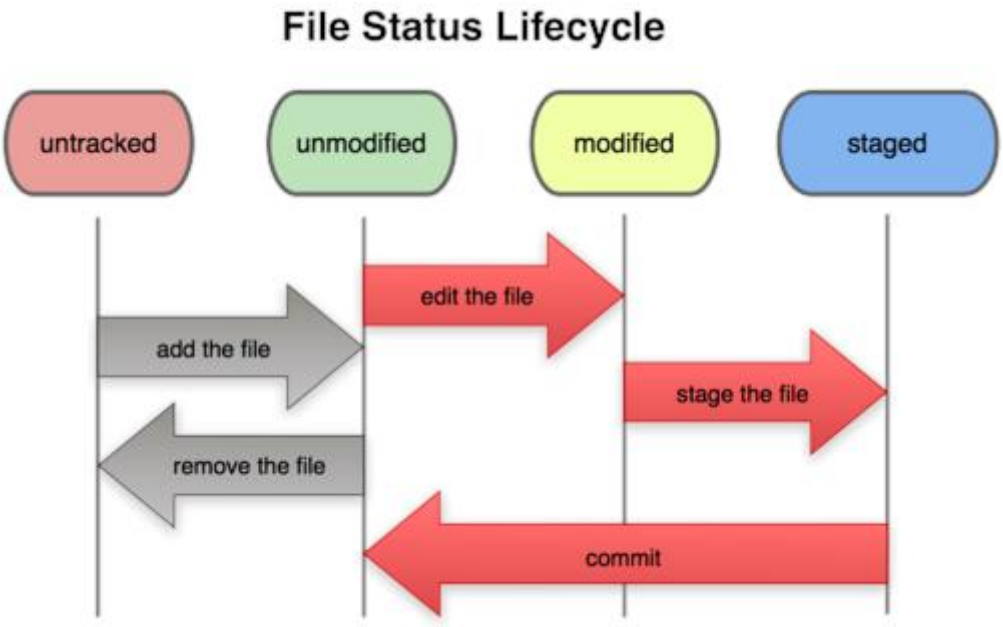
操作

- 在/home/20xxxxxx/learnGit目录下创建一个名为README.txt的文件。这时使用 git status > Untracked.txt 。
- 在 README.txt 文件中随便写点什么，然后使用刚刚学到的 add 命令，再使用 git status > Stage.txt 。
- 之后使用上面学到的 Git 提交有关的知识把 README.txt 提交，并在提交说明里写入自己的学号。
- 使用 cat Untracked.txt 和 cat Stage.txt，对比一下两次的结果，体会一下README.txt 两次所处位置的不同。
- 修改 README.txt 文件，再使用 git status > Modified.txt 。
- 使用 cat Modified.txt ，观察它和第一次 add 之前的 status 一样吗，思考一 下为什么？ (对于 Thinking 0.1，只有这一小问需要写到课后的实验报告中)

分析

新建README.txt文件后，由于此时该文件并没有被添加到缓存区，因此其处于Untracked的状态，在使用add命令后，该文件被添加到了缓存区，但是并未提交到版本库，因此此时git status后会提示使用commit命令，commit后，此时工作区，暂存区和版本库中版本统一，所有文件内容均相同，在更改了README.txt文件后，由于工作区文件与暂存区、版本库中不同，此时有两个选项，1.放弃这次的的更改，使用git ckeckout README.txt来回退到上次提交后的README.txt；2.保留此次修改，把README.txt使用add命令添加到暂存区，并commit至版本库。

Thinking0.2



操作	命令
add the file	git add
stage the file	git add
commit	git checkout --
edit the file	git add
remove the file	git rm --cached

Thinking0.3

操作

- **深夜，小明在做操作系统实验。困意一阵阵袭来，小明睡倒在了键盘上。等到小明早上醒来的时候，他惊恐地发现，他把一个重要的代码文件printf.c删除掉了。苦恼的小明向你求助，你该怎样帮他把代码文件恢复呢?**
- **正在小明苦恼的时候，小红主动请缨帮小明解决问题。小红很爽快地在键盘上敲下了git rm printf.c，这下事情更复杂了，现在你又该如何处理才能弥补小红的过错呢?**
- **处理完代码文件，你正打算去找小明说他的文件已经恢复了，但突然发现小明的仓库里有一个叫Tucao.txt，你好奇地打开一看，发现是吐槽操作系统实验的，且该文件已经被添加到暂存区了，面对这样的情况，你该如何设置才能使Tucao.txt在不从工作区删除的情况下不会被git commit指令提交到版本库?**

分析

- 暂存区文件仍存在，但工作区文件已删除，使用git checkout printf.c
- 暂存区与工作区中printf.c被删除，因此需要从版本库中获取printf.c，使用git checkout HEAD printf.c
- 从暂存区中删去Tucao.txt，即将其变为untracked，使用git rm --cached Tucao.txt

Thinking0.4

操作

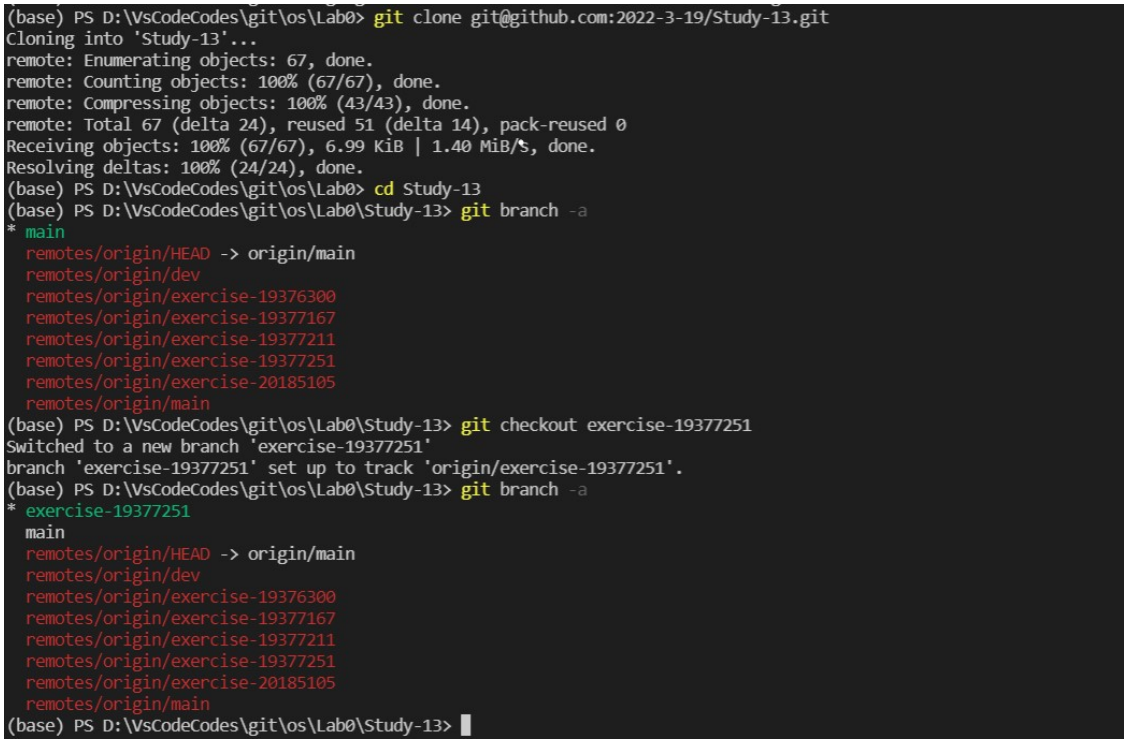
- **找到我们在/home/20xxxxxx/learnGit下刚刚创建的README.txt，没有的话就新建一个。**
- **在文件里加入Testing 1，add，commit，提交说明写 1。**
- **模仿上述做法，把1分别改为 2 和 3，再提交两次。**
- **使用 git log命令查看一下提交日志，看是否已经有三次提交了？记下提交说明为 3 的哈希值**
- **开动时光机！使用 git reset --hard HEAD^ ，现在再使用git log，看看什么没了?**
- **找到提交说明为1的哈希值，使用 git reset --hard ，再使用git log，看看什么没了?**
- **现在我们已经回到过去了，为了再次回到未来，使用 git reset --hard ，再使用git log，我胡汉三又回来了!**

分析

- git reset --hard 撤销工作区中所有未提交的修改内容，将暂存区与工作区都回到上一次版本，并删除之前的所有信息提交
- 可选参数 HEAD~n 为回退的版本次数； HEAD^ 有几个^代表回退几个版本；或者 给i他
- git reset --hard <hash值> 回退/前进至hash值对应的版本中

Thinkinh0.5

1. **克隆时所有分支均被克隆，但只有HEAD指向的分支被检出。**



如图，在clone至本地后，HEAD指向的分支被克隆且被检出，其他分支没有被克隆，当切换分支时对应分支才被克隆并检出。

2. **克隆出的工作区中执行 git log、git status、git checkout、git commit等操作不会去访问远程版本库。**
    - git log在clone时便被clone到本地，不会访问远程版本库。
    - git status是查看本地工作区与缓存区的文件状态，不会访问远程版本库。
    - 由上题，git checkout切换至未被克隆的分支时会访问远程版本库，切换至已经克隆的分支时不会访问远程版本库。当checkout用于撤销工作区的修改时，会把暂存区的内容覆盖工作区，不会访问远程版本库。
    - git commit用于把本地的暂存区内容提交至本地的版本库，因此不会访问远程版本库。
3. **克隆时只有远程版本库HEAD指向的分支被克隆。**

    由上述及图片，正确的
4. **克隆后工作区的默认分支处于master分支。**

    由上述及图片，正确的

Thinking0.6

要求

- 执行如下命令,并查看结果
    - ```
      echo first
      echo second > output.txt
      echo third > output.txt
      echo forth >> output.txt
      ```

分析

第一句仅使用echo，即在命令行打印first

第二句使用输出重定向，且覆盖原文件的重定向，故output.txt中写入second

第三句使用输出重定向，且覆盖原文件的重定向，故output.txt中写入third

第四局使用输出重定向，在原文件后追加的重定向，故output.txt中写入third forth
      ```

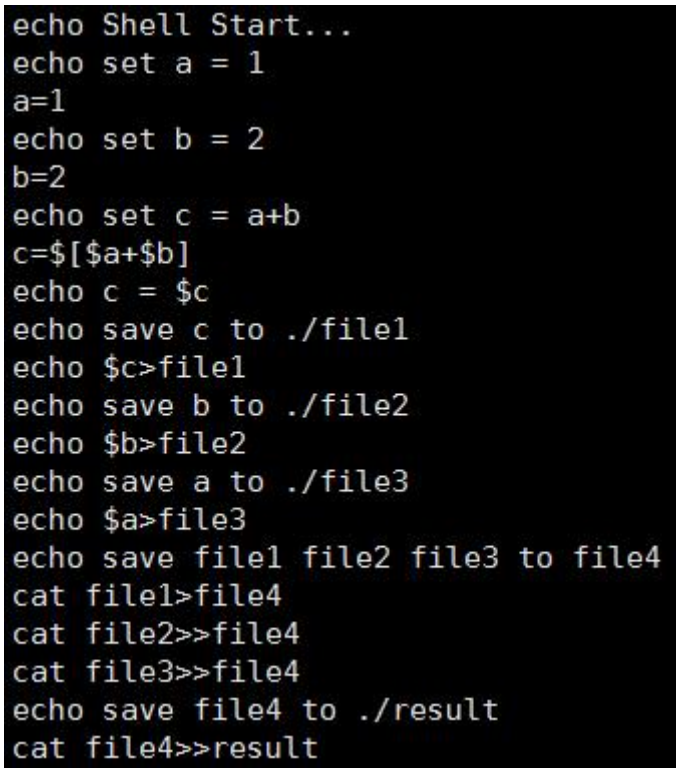
故输出为：

- 命令行
  - ```
    first
    ```
- output.txt
  - ```
    second
    ```
  - ```
    third
    forth
    ```

Thinking0.7

要求

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.



分析

command文件：

```
#!/bin/bash
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
```

生成的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
```

- echo echo Shell Start 与 echo 'echo Shell Start'效果是否有区别

    - 在command文件中，把第一行echo 'echo Shell Start...'更改为echo echo Shell Start...后test文件中内容不边，因此效果应该没有区别
- echo echo \$c>file1 与 echo 'echo \$c>file1'效果是否有区别.
    - 有区别，第一句将echo $c重定向输入至file1，而第二句则是把echo \$c>file1输出至命令行

linux基本操作命令

ls

```
查看当前目录的文件
用法:ls [选项]... [文件]...
选项（常用）：
        -a      不隐藏任何以"."开始的项目
        -l      每行只列出一个文件
```

touch

```
新建文件
用法:touch [选项]... [文件]...
例：touch hello_world.c
```

mkdir

```
新建文件夹
用法:mkdir [选项]... 目录...
```

cd

```
进入目录
用法:cd [选项]... 目录...
例：  cd .. 返回上一级目录
```

pwd

```
查看当前绝对路径
```

rmdir

```
删除空目录
用法:rmdir [选项]... 目录...
```

rm

```
删除目录或文件
rm - remove files or directories
用法:rm [选项]... 文件...
选项（常用）：
        -r      递归删除目录及其内容
        -f      强制删除。忽略不存在的文件，不提示确认
 rm helloworld.c
```

cp

```
拷贝
用法:cp [选项]... 源文件... 目录
选项（常用）：
        -r      递归复制目录及其子目录内的所有内容
cp h.c trashbin/
```

mv

```
移动-拷贝+删除
mv - move/rename file
用法:mv [选项]... 源文件... 目录
特殊用法 mv oldname newname 重命名
```

杂项

在多数shell中，四个方向键也是有各自特定的功能的：←和→可以控制光标的位置，↑和↓可以切换最近使用过的命令

```
- 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（若正在使用Shell，则退出当前Shell）
- Ctrl+L 清屏
```

linux进阶操作命令

vim

命令模式下

指令	作用
i	切换为插入模式
Esc	返回普通模式
u	撤销
yy	复制一行
dd	剪切一行
o	在当前行之下插入
O	在当前行之后插入
y 复制	按住y后移动方向键即可选中复制的区域
d 剪切	按住y后移动方向键即可选中剪切的区域
2yy	复制下面的两行
p	在当前位置之后粘贴
P	在当前位置之前粘贴
:q	不保存直接退出
:q!	强制不保存直接退出
:w	保存
:wq	保存后退出
:N	切换到第N行
:set nu	显示行号
/word	搜索word出现的位置，如果有多个，n/N分别切换至上/下一个

gcc

流程

.c -> 预处理（-E) .i ->编译(-S).s ->汇编(-c).o ->链接 可执行文件

ISO ESC

```
gcc -E test.c -o test.i
gcc -S test.i -o test.s
gcc -c test.s -o test.o
gcc test.o -o test
```

可选项

选项	作用
-E	仅做预处理，生成.i文件
-S	仅做编译，生成.s文件
-c	仅作汇编，生成.o文件
-Wall	显示最多警告信息
-I	指定头文件路径
-include filename	编译时用来包含头文件，功能相当于在代码中使用#include
-DCptz	定义宏Cptz，中间无空格

Makefile

格式

```
target: dependencies
    command 1
    command 2
    ...
    command n
#其中，target是我们构建(Build)的目标，而dependencies是构建该目标所需的其它文件或其他目标。之后是构建出该目标所需执行的指令。 有一点尤为需要注意：每一个指令(command)之前必须有一个TAB。这里必须使用TAB而不能是空格，否则make会报错。
```

make 只有 在依赖文件中存在文件的修改时间比目标文件的修改时间晚的时候 (也就是对依赖文件 做了改动)，shell 命令才被会执行，编译生成新的目标文件。

链接	简要介绍
http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor	详细描述了如何从一个简单的makefile入手，逐步扩充其功能，最终写出一个相对完善的makefile，同学们可以从这一网站入手，仿照其样例，写出一个自己的makefile。
http://www.gnu.org/software/make/manual/make.html#Reading-Makefiles	提供了一份完备的makefile语法说明，今后同学们在实验中遇到了未知的makefile语法，均可在这一网站上找到满意的答案。

若只输入make则默认执行第一个target

ctags

Generate tag files for source code

帮助查看多个.c文件间函数的调用关系

步骤:

1. 配置vim
   vim .vimrc,添加

```
  set tags=tags;
  set antochdir
```

2. 建立多个互相有调用关系的.c文件
3. 命令行执行 ctags -R *，然后就可以使用ctags的一些功能
4. vim查看.c 文件，光标移到函数调用上时，按下Ctrl+],即可跳转到定义处，Ctrl+o即可返回；或者vim中:进入底线命令模式，输入tag 函数名 即可跳转到函数定义的位置

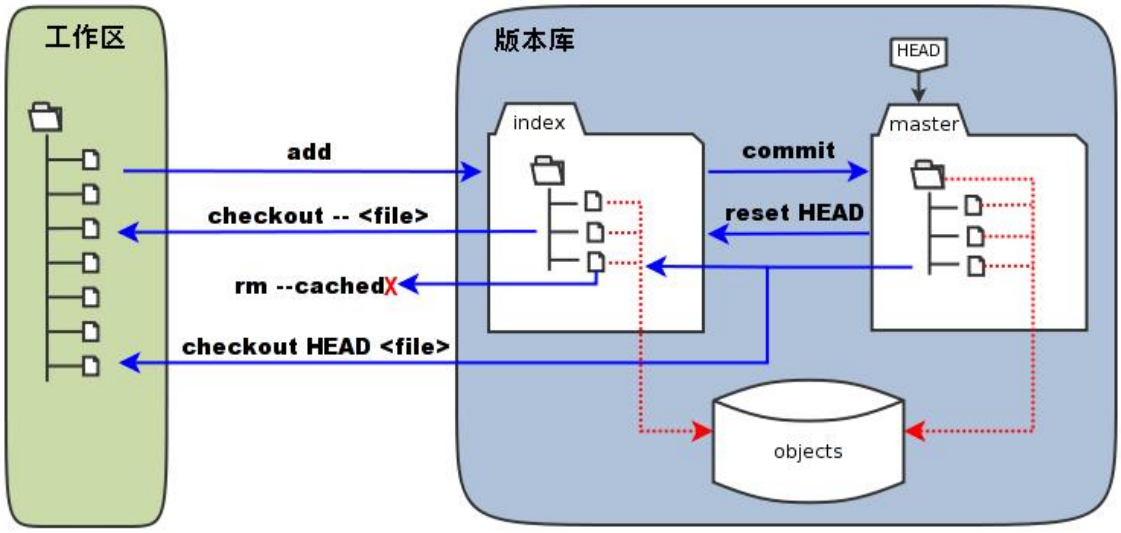
Git

用于项目**版本控制**

版本控制是一种记录若干文件内容变化，以便将来查阅特定版本修订情况的系统。

工作原理

git 的对象库只保存了文件信息，而没有目录结构，因此我们的本地仓库由 git 还维 护了三个目录。第一个目录我们称为**工作区**，在本地计算机文件系统中能看到这个目录， 它保存实际文件。第二个目录是**暂存区**（英语是 Index，有时也称 Stage），是.git/index 文件，用于暂存工作区中被追踪的文件。将工作区的文件内容放入暂存区，可理解为给 工作区做了一个快照（snapshot）。当工作区文件被破坏的时候，可以根据暂存区的快照对工作区进行恢复。最后一个目录是**版本库**，是 HEAD 指向最近一次提交（commit）后 的结果。在项目开发的一定阶段，将可以在将暂存区的内容归档，放入版本库。如下图：



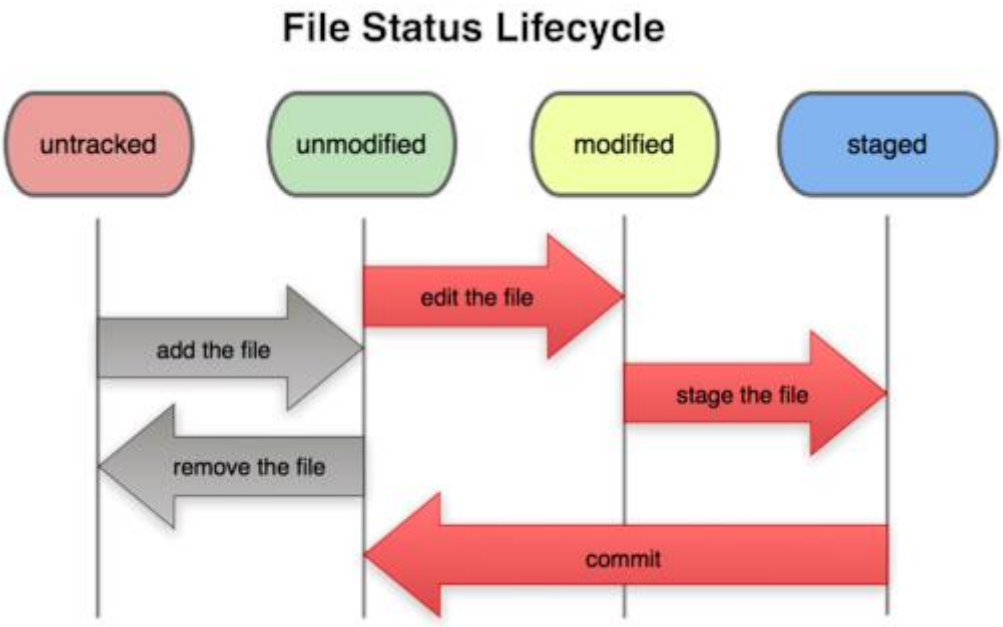
- 当对工作区修改（或新增）的文件执行 “git add” 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID 被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，会将**暂存区**的目录树写到**版本库**（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。
- 当执行 “git rm --cached ” 命令时，会直接从**暂存区**删除文件，**工作区**则不做出改变。
- 当执行 “git reset HEAD” 命令时，**暂存区**的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 “git checkout -- / .” 命令时，会用**暂存区**指定的文件替换**工作区**的文件。这个操作很危险，会清除工作区中未添加到暂存区的改动。
- 当执行 “git checkout HEAD / .” 命令时，会用 HEAD 指向的 master 分支中的指定文件**替换暂存区和以及工作区中的文件**。这个命令也是极具危险性的，因为不但**会清除工作区中未提交的改动**，也**会清除暂存区中未提交的改动**。

文件状态

一般修改或新建文件后都需要 git add

文件状态	详细信息
未跟踪 (untracked)	表示没有跟踪(add)某个文件的变化，使用git add即可跟踪文件
未修改 (unmodified)	表示某文件在跟踪后一直没有改动过或者改动已经被提交
已修改 (modified)	表示修改了某个文件,但还没有加入(add)到暂存区中
已暂存 (staged)	表示把已修改的文件放在下次提交(commit)时要保存的清单中

状态转移图谱



操作	命令
add the file	git add
stage the file	git add
commit	git checkout --
edit the file	git add
remove the file	git rm --cached

`git rm --cached <file>`这条指令是指从暂存区中删去一些我们不想跟踪的文件，比如我们自己调试用的文件等。

`git checkout -- <file>`如果我们在工作区改呀改，把一堆文件改得乱七八糟的，发现编译不过了！别急，如果我们还没`git add`，就能使用这条命令，把它变回曾经美妙的样子。

`git reset HEAD <file>`刚才提到，如果没有`git add` 把修改放入暂存区的话，我们可以使用`checkout` 命令，那么如果我们不慎已经 `git add` 加入了怎么办呢？那就需要这条指令来帮助我们了！这条指令可以让我们的暂存区焕然一新。再对同一个文件使用楼上那条指令，哈哈，世界清静了。

`git clean <file> -f`如果你的工作区这时候混入了奇怪的东西，你没有追踪它，但是想清除它的话就可以使用这条指令，它可以帮你把奇怪的东西剔除出去。

分支

创建一个基于当前分支的分支，其功能相当于把当前分支的内容拷贝一份到新的分支里去，然后我们在新的分支上做测试功能的添加即可，不会影响实验分支的效果等。

```
git branch <new-branch-name>#基于当前工作分支创建新分支，HEAD仍指向当前工作分支
git branch -b <new-branch-name>#基于当前工作分支创建新分支，HEAD指向新建的分支
git branch -d <branch-name>#删除对应分支
git branch -a#查看所有分支 包括本地与远程仓库
git checkout <branch-name>#切换分支
```

分支合并

创建分支后，总会有它合并到主分支的啥时候，使用git merge则可以合并分支，合并完成后就可以删除子分支了

合并冲突

当合并时遇到对同一文件的不同修改时，会提示用户处理两次修改的冲突

```
#提示信息
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.

有冲突的文件中往往包含一部分类似如下的奇怪代码，我们打开test.txt，发现这样一些“乱码”

a123
<<<<<<< HEAD
b789
=======
b45678910
>>>>>>> 6853e5ff961e684d3a6c02d4d06183b5ff330dcc# 修改版本的hash值
c
冲突标记<<<<<<< 与=======之间的内容是当前HEAD指向分支中的修改，=======与>>>>>>>之间的内容是在合并的另一分支的修改
```

处理方法是，修改产生冲突的文件来处理冲突，随后git add与git commit来告诉git已经解决了冲突

远程仓库

远程仓库其实和本地版本库结构是一致的，只不过远程仓库是在服务器上的仓库，而本地仓库是在本地的。

```
git clone 用于从远程仓库克隆一份到本地版本库
git remote add [short name] [https/ssh url] #添加远程仓库
```

```
git remote [-v]查看当前配置的远程仓库 -v显示实际链接地址
```

git fetch

用于从远程获取代码库，一般fetch后使用merge来与本地库进行合并

git pull

用于从远程获取代码并合并本地的版本。**git pull** 其实就是 **git fetch** 和 **git merge FETCH_HEAD** 的简写。

git push

用于从将本地的分支版本上传到远程并合并。当版本有差异时想要强制push可以使用-f[--force]参数。push推送的是版本库。

一些补充

见文章开头的Thinking中分析部分

find

用于在当前目录下递归的**查找符合参数所示文件名**的文件，并输出文件的路径

```
find -name filename
Example: find -name *.go
```

grep

Globally search a Regular Expression and Print

grep是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。简单来说，grep命令可以从文件中查找包含pattern部分字符串的行，并将该文件的路径和该行输出至屏幕。**当你需要在整个项目目录中查找某个函数名、变量名等特定文本的时候，grep 将是你手头一个强有力的工具。**

预设 grep 指令会把含有范本样式的那一列显示出来。若不指定任何文件名称，或是所给予的文件名为 **-**，则 grep 指令会从标准输入设备读取数据。

```
grep int test.c -n
```

参数	作用
-n	列出匹配到的结果所在的行号
-a	不忽略二进制数据搜索
-i	忽略文件大小写
-r	从文件夹递归查找
-v	显示不匹配的行的数据
-An	显示匹配到的行的后n行（A after）
-Bn	显示匹配到的行的前n行（B before）
-c	显示匹配到的行数(若搜索目录是文件夹，则输出标准为 filepath:count)

tree

tree指令可以根据文件目录生成文件树，作用类似于ls。

用法：tree [选项] [目录名]

选项	作用
-a	列出全部文件
-d	只列出目录

locate

用于查找文件，速度往往快于find，但是他的查找不是实时的，因为find是直接查找对应文件，而locate是查找/var/lib/slocate的资料库中找，有的文件由于更新不及时可能未能在数据库中找到

用法 locate [选项] 文件名

chmod

change mode

Linux的文件调用权限分为三级 : 文件拥有者、群组、其他。利用 chmod 可以藉以控制文件如何被他人所调用。

用户类型

who	用户类型	说明
u	user	文件所有者
g	group	文件所有者所在组
o	others	所有其他用户
a	all	所有用户，相当于ugo

操作符

Operator	说明
+	为指定用户增加权限
-	为指定用户去除权限
=	设置指定用户的权限，将对应用户权限重新设置

权限

模式	名字	说明
r	读	设置为可读权限
w	写	设置为可写权限
x	执行权限	设置为可执行权限

举例

```
chmod a=rwx test# 把test文件设置为任何人可读写执行
chmod +x change_file.sh# 为change_file.sh增加执行权限
```

diff

用于比较两文件的差异

diff [选项] 文件1 文件2

可选项

选项	作用
-b	不检查空格字符
-B	不检查空行
-q	仅显示有无差异，不显示详细信息

sed

一个文件处理工具，可以将数据行进行替换、删除、新增、选取等特定工作

sed [选项]命令 输入文本

选项

选项	说明
-n	使用安静模式，即只输出进行处理的内容
-e	使用多项编辑，可在同一条命令中按序执行多条sed命令
-i	直接对读取的内容进行修改，而不是输出到屏幕

命令

命令	说明
a	新增
c	取代
d	删除
i	插入
p	打印，通常搭配-n
s	取代 s/old/new/g 所有old更改为new

举例

- 输出my.txt的第三行。
  - sed -n '3p' my.txt
- 删除my.txt文件的第二行到最后一行。
  - 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str1被替换成str2。
  - sed 's/str1/str2/g' my.txt
- -e选项允许在同一行里执行多条命令。例子的第一条是第四行后添加一个str，第二个命令是将str替换为aaa。命令的执行顺序对结果有影响。
  - sed -e '4a\str ' -e 's/str/aaa/' my.txt
- 对sed使用管道与重定向输出
  - sed /this/p -n conflict | sed a\778 > output

awk

- awk '$1>2 {print $1,$3}' my.txt
  - 这个命令的格式为awk 'pattern action' file，pattern为条件，action为命令，file为文件。命令中出现的$n代表每一行中用空格分隔后的第n项。所以该命令的意义是文件my.txt中所有第一项大于2的行，输出第一项和第三项。
- 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，分隔的第n项了。
- grep int file -n | awk -F: '{print $1}' >output
  - 用于提取file文件中包含int的行数

选项参数

参数	说明
-F	指定文件分隔符，默认为空格

tmux

tmux是一个优秀的终端复用软件，可用于在一个终端窗口中运行多个终端会话。窗格，窗口，会话是tmux的三个基本概念，一个会话可以包含多个窗口，一个窗口可以分割为多个窗格。突然中断退出后tmux仍会保持会话，通过进入会话可以直接从之前的环境开始工作。

会话操作 session

命令	解释
tmux new -s 会话名	新建一个名为..的会话
Ctrl+b d	退出会话，回到shell的环境
tmux ls	终端环境查看会话列表
tmux a -t 会话名	从终端进入对应会话
tmux kill-session -t 会话名	销毁对应会话

窗口操作 window

命令	解释
Ctrl+b c	创建一个新窗口
Ctrl+b p	切换到上一个窗口
Ctrl+b n	切换到下一个窗口
Ctrl+b 0	切换到0号窗口，数字键以此类推
Ctrl+b w	列出当前session的所有窗口，通过上下键切换窗口
Ctrl+b &	关闭当前窗口，随后按y确认即可

窗格操作 pane

命令	解释
Ctrl+b %	垂直分屏
Ctrl+b "	水平分屏
Ctrl+b o	依次切换当前窗口的窗格
Ctrl+b Up|Down|Left|Right	根据方向切换窗格
Ctrl+b PageUp|PageDown	向上与向下翻页
Ctrl+b Space	对当前窗口下窗格重新布局
Ctrl+b z	最大化当前窗格，再按一次恢复
Ctrl+b x	关闭正在使用中的窗格

Shell脚本

当有很多想要执行的Linux指令来完成复杂的工作，或者有一个或一组指令会经常执行时，我们可以通过shell脚本来完成。

新建my.sh后，第一行添加

```
#!/binbash
```

以保证脚本默认会使用bash

随后运行chmod +x my.sh为其添加运行权限

后续./my.sh即可运行

参数

参数	解释
$n	代表获取命令的第n个参数，$0为./sh
$#	获取的参数个数
$?	前一条命令的运行状态，0表示没有错误，否则有错误
$*	以一个单字符串显示所有向脚本传递的参数。如"*"用「"」括起来的情况、以"$1 $2 … $n"的形式输出所有参数。

shell中的函数也使用上述方法传参

```
fun(){
    echo $1
    echo $2
    echo "the number of parameters is $#"
}
fun 1 2
```

shell条件控制

if

```
if condition
then
    command1
    command2
fi

if condition
then
    command1
    command2
    ...
    commandN
else
    command
fi

if condition1
then
    command1
elif condition2
then
    command2
else
    commandN
fi
```

while

```
while condition
do
    commands
done
```

for

```
for var in item1 item2 ... itemN
do
    command1
    command2
    ...
    commandN
done
```

重定向与管道

shell使用三种流：

- 标准输入：stdin，由0表示
- 标准输出：stdout，由1表示
- 标准错误：stderr，由2表示

重定向和管道可以重定向以上的流。

重定向

命令	说明
command > file	重定向输出到file
command < file	重定向输入到file
command >> file	重定向输出追加到file
command 2> file	重定向stderr到file
command 2>> file	重定向stderr追加到file

管道

管道符号“|”可以连接命令：

```
command1 | command2 | command3 | ...
```

以上内容是将command1的stdout发给command2的stdin，command2的stdout发给command3的stdin，依此类推。

例如

- echo "3 5" | ./hello
  - 把3 5 作为hello可执行文件的输入
- cat my.sh | grep "Hello"
  - 上述命令的功能为将my.sh的内容输出给grep指令，grep在其中查找字符串。
- cat < my.sh | grep "Hello" > output.txt
  - 上述命令重定向和管道混合使用，功能为将my.sh的内容作为cat指令标准输入，cat指令stdout发给grep指令的stdin，grep在其中查找字符串，最后将结果输出到output.txt。