

哈尔滨工业大学 计算学部

2024 年秋季学期《开源软件开发实践》

Lab 1: Git 实战

姓名	学号	联系方式
徐耀	2022211830	15318051170

目 录

1 实验要求	1
2 安装 Git	1
2.1 本地机器上安装 git	1
2.2 申请 github 帐号	2
3 Git 操作过程	2
3.1 实验场景(1): 仓库创建与提交	3
3.2 实验场景(2): 分支管理	10
3.3 实验场景(3): 在线 Git 练习	16
4 小结	21

1 实验要求

- (1) 了解配置管理工具 Git 及相应用环境;
- (2) 熟练掌握 Git 的基本指令和分支管理指令;
- (3) 掌握 Git 支持软件配置管理的核心机理;
- (4) 在实践项目中使用 Gitee/GitLab/GitHub 管理自己的项目源代码。

2 安装 Git

2.1 本地机器上安装 git

本人采用在 Windows 系统下安装，安装版本为 2.44.0.windows.1



```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.22631.3296]
(c) Microsoft Corporation。保留所有权利。

C:\Users\Administrator>git --version
git version 2.44.0.windows.1

C:\Users\Administrator>
```

图 2-1 Git 版本号

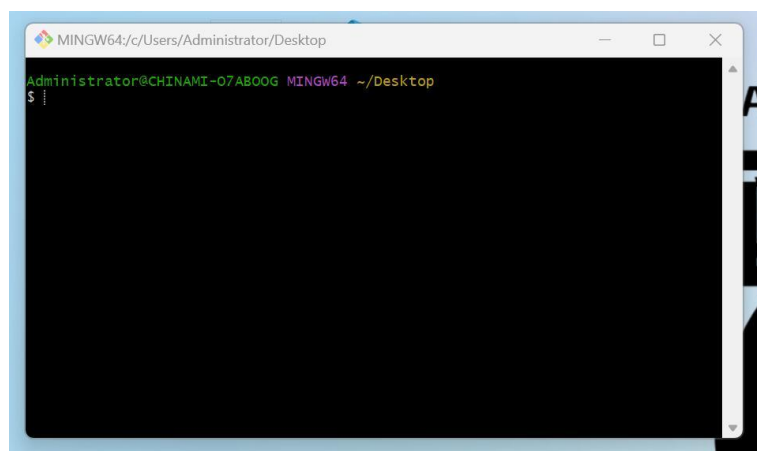


图 2-2 Git 运行界面

2.2 申请 github 帐号

本次实验本人采用 Github。

(1) 账号名称: kathy40405

(2) URL 地址: <https://github.com/OSSDP/Lab1-2022211830.git>

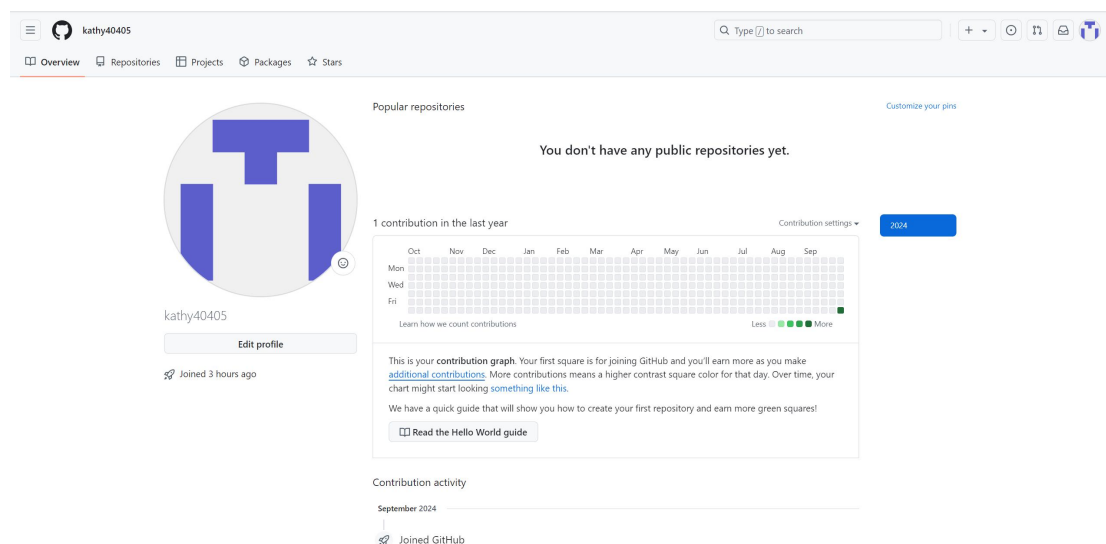


图 2-3 账号信息

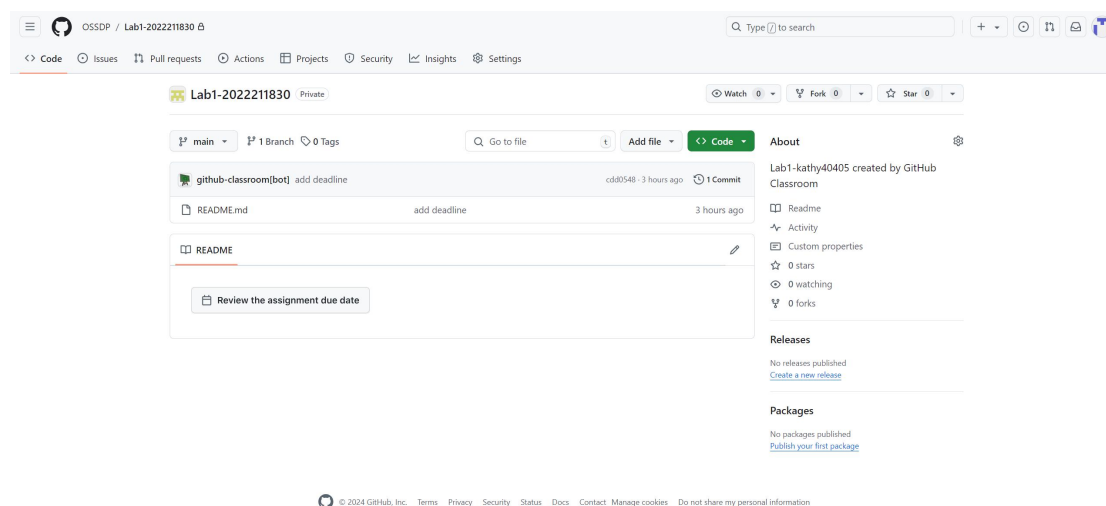


图 2-4 项目信息

3 Git 操作过程

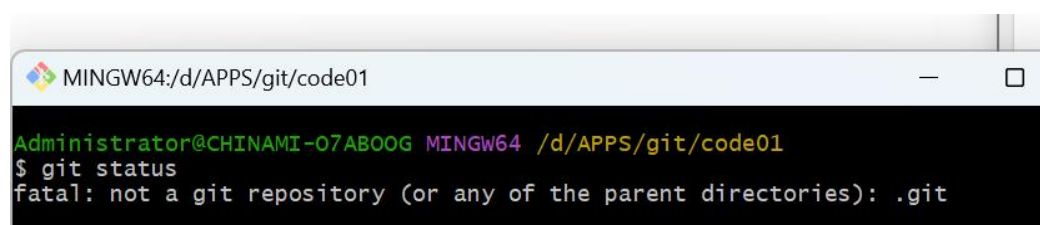
3.1 实验场景(1): 仓库创建与提交

(1) R0:

针对 R1 和 R7, 在进行每次 Git 操作之前, 随时查看工作区、暂存区、Git 仓库的状态, 确认项目里的各文件当前处于什么状态。

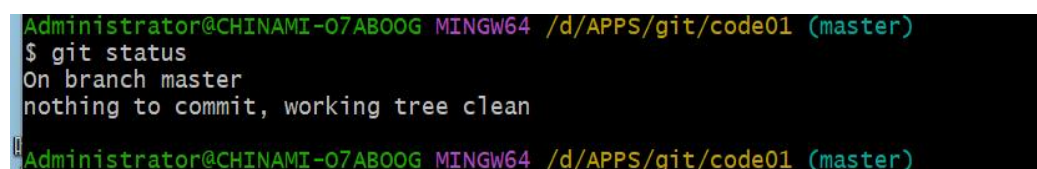
指令: `git status`

说明: 这个指令在未进行本地仓库初始化前是无法生效的, 下面分别是本人在初始化前和初始化后执行该指令得到的结果。



```
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01
$ git status
fatal: not a git repository (or any of the parent directories): .git
```

图 3-1-1 初始化前使用指令



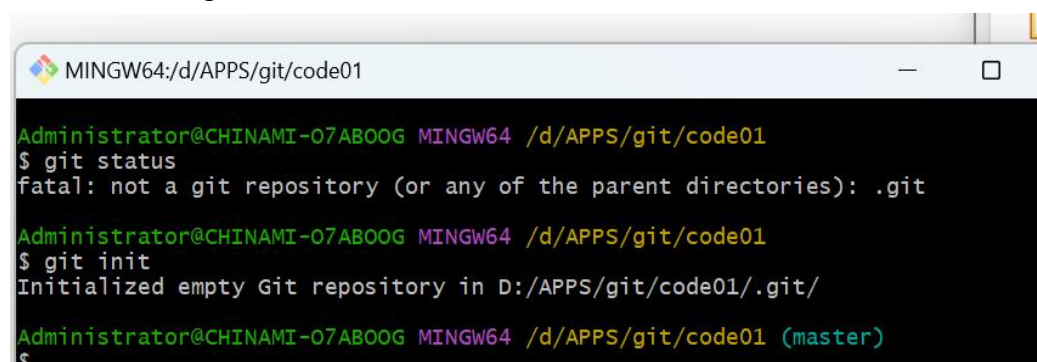
```
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master
nothing to commit, working tree clean
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (master)
```

图 3-1-2 初始化后使用指令

(2) R1:

本地初始化一个 Git 仓库, 将自己在 Lab1 中所创建项目的全部源文件加入进去, 纳入 Git 管理。

初始化指令: `git init`



```
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01
$ git status
fatal: not a git repository (or any of the parent directories): .git

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01
$ git init
Initialized empty Git repository in D:/APPS/git/code01/.git/

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (master)
$
```

图 3-1-3 初始化

执行结果:在项目文件夹下生成一个名为.git 的隐藏文件夹,需要将 Windows 资源管理器中的“显示隐藏的项目”勾选才能生效。



图 3-1-4 .git 文件

(3) R2:

提交（需要先将文件加入暂存区，用 `git add` 命令）

将文件加入暂存区：`git add .`

将暂存区的文件提交到本地仓库：`git commit-m"当次提交的备注信息"`

```
MINGW64:/d/APPS/git/code01
Reinitialized existing Git repository in D:/APPS/git/code01/.git/
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git add .
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   01.c
        new file:   02.c
        new file:   03.c
        new file:   04.c
        new file:   05.c
        new file:   06.c
        new file:   hhh.bmp
        new file:   txt.txt
        new file:   x.txt
        new file:   xy.txt
        new file:   y.txt
        new file:   z.txt
        new file:   zh.txt

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git commit -m"my first commit"
[master (root-commit) cab91a6] my first commit
13 files changed, 376 insertions(+)
create mode 100644 01.c
create mode 100644 02.c
create mode 100644 03.c
create mode 100644 04.c
create mode 100644 05.c
create mode 100644 06.c
create mode 100644 hhh.bmp
create mode 100644 txt.txt
create mode 100644 x.txt
create mode 100644 xy.txt
create mode 100644 y.txt
create mode 100644 z.txt
create mode 100644 zh.txt

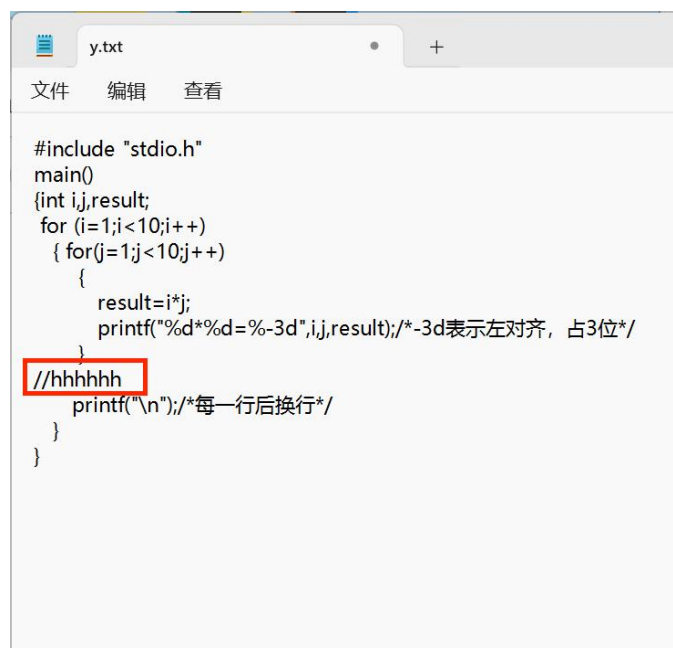
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master
nothing to commit, working tree clean
```

图 3-1-5 提交过程及结果

(4) R3:

手工对某个文件进行修改，查看上次提交之后都有哪些文件修改、具体修改内容是什么（查看修改后的文件和暂存区域中相应文件的差别）。

首先手动对文件进行修改，这里为了不对原有的代码进行改动，本人只在原来的 y.txt 文件中添加了一行注释://hhhhhh



```

y.txt
文件 编辑 查看

#include "stdio.h"
main()
{int i,j,result;
  for (i=1;i<10;i++)
  { for(j=1;j<10;j++)
    {
      result=i*j;
      printf("%d*%d=%-3d",i,j,result);/*-3d表示左对齐, 占3位*/
    }
    //hhhhhh
    printf("\n");/*每一行后换行*/
  }
}

```

图 3-1-6 修改后文件内容

指令:

查看文件修改情况: `git status`

查看具体变动内容: `git diff`

执行结果: 可以发现使用 `git status` 指令后结果中将"y.txt"文件被修改的结果标红了, 随后使用 `git diff` 指令, 结果中标记了"+ //hhhhhh"的字样, 说明前面的修改被 Git 工具发现。

```

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   y.txt

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git diff
diff --git a/y.txt b/y.txt
index abf9e4e..690781d 100644
--- a/y.txt
+++ b/y.txt
@@ -7,6 +7,7 @@ main()
     result=i*j;
     printf("%d*%d=%-3d",i,j,result);/*-3d表示左对齐, 占3位*/
+    //hhhhhh
     printf("\n");/*每一行后换行*/
 }
}
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)

```

图 3-1-7 修改后查找

(5) R4:

重新提交。

这一步的指令同 R2:

将文件加入暂存区: `git add` .

将暂存区的文件提交到本地仓库: `git commit-m"2nd"`

```
Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (master)
$ git add .

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   y.txt

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (master)
$ git commit -m"2nd"
[master 36fa031] 2nd
 1 file changed, 1 insertion(+)

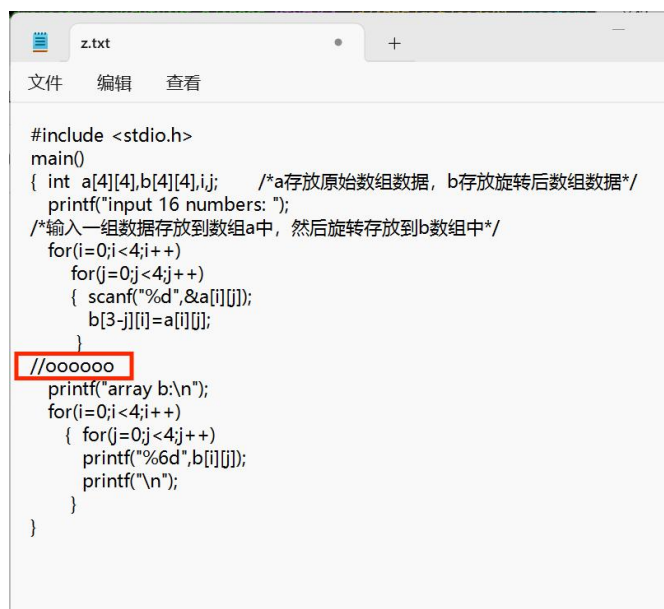
Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (master)
```

图 3-1-8 重新提交过程

(6) R5:

再次对项目中的某个文件进行修改，并重新提交。

本次对 `z.txt` 文件中添加注释 `//oooooo`



```
z.txt
文件 编辑 查看

#include <stdio.h>
main()
{ int a[4][4],b[4][4],i,j; /*a存放原始数组数据, b存放旋转后数组数据*/
  printf("input 16 numbers: ");
  /*输入一组数据存放到数组a中, 然后旋转存放到b数组中*/
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
    { scanf("%d",&a[i][j]);
      b[3-j][i]=a[i][j];
    }
  //oooooo
  printf("array b:\n");
  for(i=0;i<4;i++)
  { for(j=0;j<4;j++)
    { printf("%6d",b[i][j]);
      printf("\n");
    }
  }
}
```

图 3-1-9 修改后的文件内容

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git add .

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   z.txt

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git commit -m"3rd"
[master 13e0f80] 3rd
 1 file changed, 1 insertion(+)

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$
```

图 3-1-10 再次提交文件

(7) R6:

首先可以执行 `git reflog` 查看历史的提交版本。

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git reflog
13e0f80 (HEAD -> master) HEAD@{0}: commit: 3rd
36fa031 HEAD@{1}: commit: 2nd
cab91a6 HEAD@{2}: commit (initial): my first commit

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$
```

图 3-1-11 历史提交记录

可以观察到包括初始化的提交在内一共提交了三次，并且目前的 HEAD 指向的是最后一次提交。

现在执行撤销指令：`git reset --hard HEAD^`。

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git reset --hard HEAD^
HEAD is now at 36fa031 2nd

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$
```

图 3-1-12 撤销指令执行

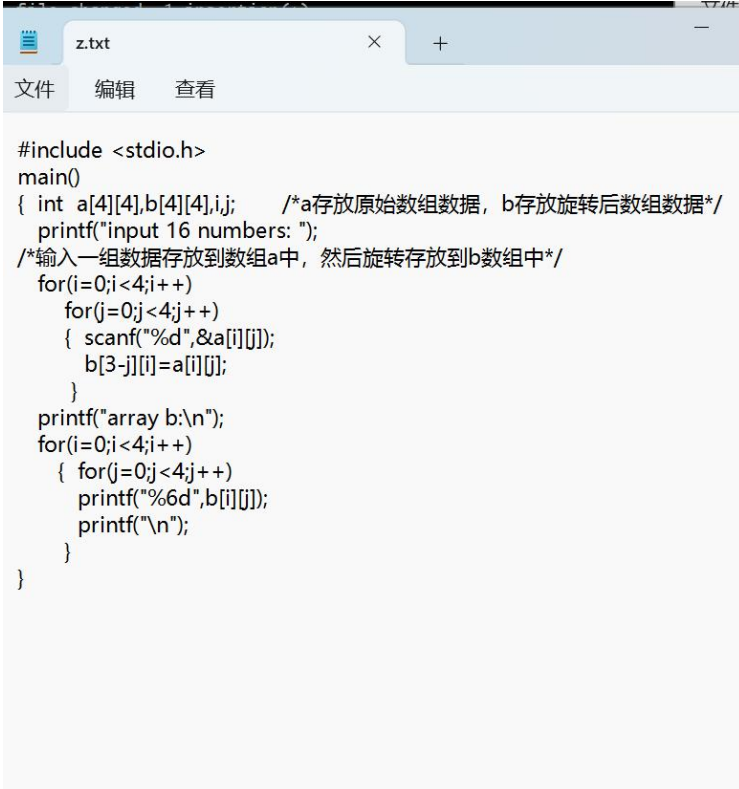
再次执行 `git reflog` 查看版本情况，特别注意版本的序列号：

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git reflog
36fa031 (HEAD -> master) HEAD@{0}: reset: moving to HEAD^
13e0f80 HEAD@{1}: commit: 3rd
36fa031 (HEAD -> master) HEAD@{2}: commit: 2nd
cab91a6 HEAD@{3}: commit (initial): my first commit

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$
```

图 3-1-13 执行撤销指令后的历史版本

观察到撤销操作后版本的序列号从"13e0f80"变成了"36fa031", 这与第二次提交的序列号一致, 说明撤销操作成功。查看对应的文件, 发现第二次修改后的结果被撤回:



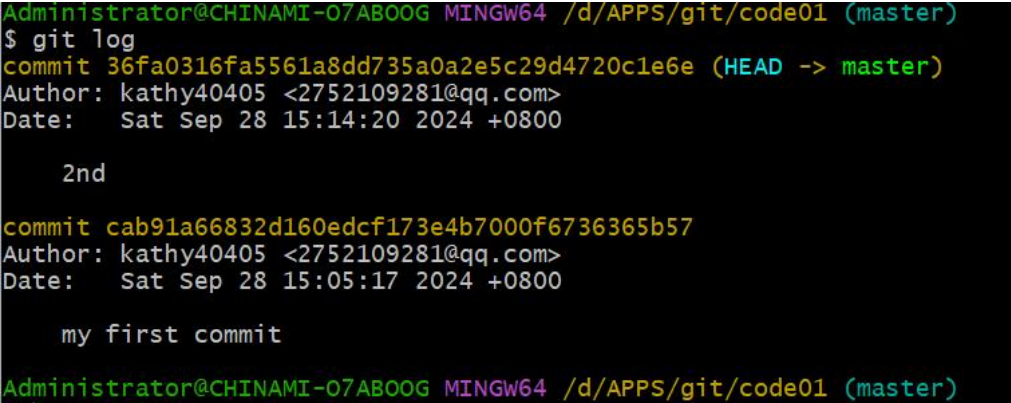
```
#include <stdio.h>
main()
{ int a[4][4],b[4][4],i,j; /*a存放原始数组数据, b存放旋转后数组数据*/
  printf("input 16 numbers: ");
  /*输入一组数据存放到数组a中, 然后旋转存放到b数组中*/
  for(i=0;i<4;i++)
    for(j=0;j<4;j++)
      { scanf("%d",&a[i][j]);
        b[3-j][i]=a[i][j];
      }
  printf("array b:\n");
  for(i=0;i<4;i++)
    { for(j=0;j<4;j++)
      { printf("%6d",b[i][j]);
        printf("\n");
      }
    }
}
```

图 3-1-14 执行撤销指令后的文件内容

(8) R7:

查询提交记录。

指令: git log



```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git log
commit 36fa0316fa5561a8dd735a0a2e5c29d4720c1e6e (HEAD -> master)
Author: kathy40405 <2752109281@qq.com>
Date: Sat Sep 28 15:14:20 2024 +0800

    2nd

commit cab91a66832d160edcf173e4b7000f6736365b57
Author: kathy40405 <2752109281@qq.com>
Date: Sat Sep 28 15:05:17 2024 +0800

    my first commit

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
```

图 3-1-15 执行查询指令的结果

可以观察到这种查询方式并不会显示撤销前的提交记录,只会显示结果到当前 HEAD 指针指向的版本, 及该版本以前的所有提交记录。

3.2 实验场景(2): 分支管理

(1) R1:

先获得本地仓库的全部分支, 指令为 `git branch`

随后切换至 `master` 分支上, 指令为 `git checkout master`, 发现提示信息为 “Already on ‘master’ ”, 说明当前分支已切换至 `master`。

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git branch
* master

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git checkout master
Already on 'master'
```

图 3-2-1 执行查看并切换分支命令

(2) R2:

在 `master` 基础上建立两个分支 `B1`、`B2`。

先在 `master` 基础上创建分支 `B1` 并切换到 `B1`, 指令为 `git checkout -b B1`

再切换回 `master` 并创建分支 `B2`, 先执行 `git checkout master` 切换回 `master` 分支, 再执行 `git checkout -b B2` 创建分支 `B2`

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git checkout -b B1
Switched to a new branch 'B1'

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git checkout master
Switched to branch 'master'

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (master)
$ git checkout -b B2
Switched to a new branch 'B2'

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B2)
$ git branch
  B1
* B2
  master

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B2)
```

图 3-2-2 创建 `B1`、`B2` 分支

可以发现已经成功创建 `B1`、`B2` 分支

(3) R3:

在 `B2` 分支基础上创建一个新分支 `C4`。

这个操作首先需要从当前的 master 分支切换到 B2 分支，再在这个分支的基础上创建一个新的分支。

切换到 B2 分支：git checkout B2

创建新的分支 C4 并切换：git checkout -b C4

```
Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (B2)
$ git checkout B2
Already on 'B2'

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (B2)
$ git checkout -b C4
Switched to a new branch 'C4'

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$
```

图 3-2-3 创建 C4

(4) R4:

在 C4 上，对 x.txt 文件进行修改并提交。

为了方便检查，本人修改的方式是简单在文件中添加一行注释。

修改的结果使用 git status 指令查看状态。

```
Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$ git status
On branch C4
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   x.txt

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$
```

图 3-2-4 使用指令查看修改成功

对于提交的过程，使用的指令仍然是：git add .和 git commit -m "2_1"，执行这两条指令后得到如下的结果：

```
Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$ git add .

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$ git status
On branch C4
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   x.txt

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$ git commit -m "2_1"
[C4 50fe24a] 2_1
1 file changed, 1 insertion(+), 1 deletion(-)

Administrator@CHINAMI-O7AB00G MINGW64 /d/APPS/git/code01 (C4)
$
```

图 3-2-5 提交修改文件

(5) R5:

在 B1 分支上对同样的文件做不同修改并提交;

首先需要切换到 B1 分支, 使用指令 `git checkout B1`

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (C4)
$ git checkout B1
Switched to branch 'B1'

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$
```

图 3-2-6 切换分支

从提示语句中可以知道已经成功切换到了 B3 分支。现在对同样的文件进行修改, 这里本人仍然采用的是在文件中增加注释的方式。在添加完注释后, 使用的指令仍然是: `git add` 文件名 和 `git commit -m "2_2"`。

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git status
On branch B1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   x.txt

no changes added to commit (use "git add" and/or "git commit -a")

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git add .

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git status
On branch B1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   x.txt

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git commit -m "2_2"
[B1 eafe36e] 2_2
1 file changed, 1 insertion(+), 1 deletion(-)

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
```

图 3-2-7 提交修改文件

(6) R6:

将 C4 合并到 B1 分支, 若有冲突, 手工消解;

合并分支指令: `git merge 分支名`

需要注意的是这个指令是将分支名指定的分支合并到当前所在的分支上, 这里本人处在的分支是上一步的 B1, 因此需要使用的指令是 `git merge C4`。执行后得到如下的结果:

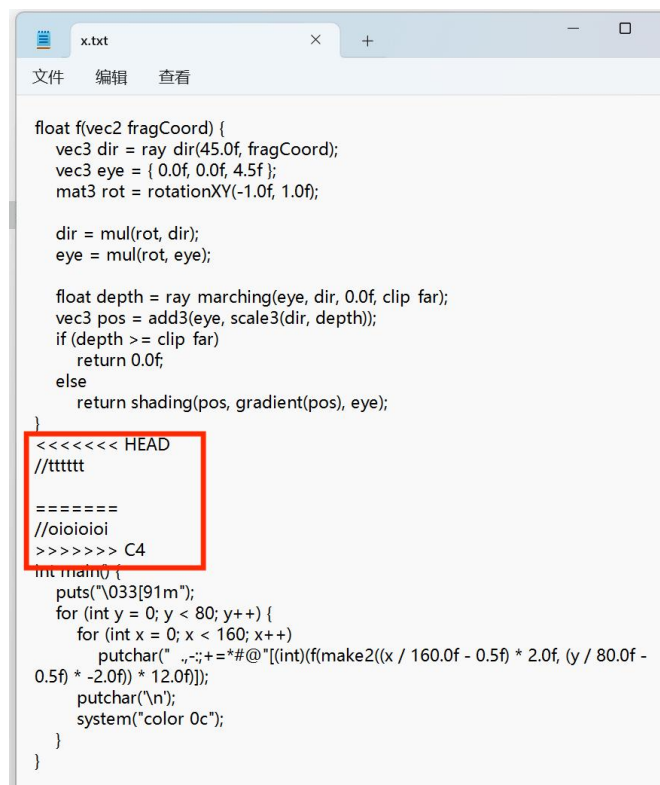
```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1)
$ git merge C4
Auto-merging x.txt
CONFLICT (content): Merge conflict in x.txt
Automatic merge failed; fix conflicts and then commit the result.

Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (B1|MERGING)
```

图 3-2-8 执行合并指令后的结果

Git 提示有 1 个文件产生冲突, 分别是之前对 C4、B3 分支分别做改动的 x.txt 文件, 这个也是符合预期的。在产生冲突时, Git 的命令行后面会有类似这里" (B1|MERGING)"的提示, 下一步需要手动进行合并。

手动合并并没有对应的指令，需要开发者手动打开发生冲突的文件修改，打开这个文件后发现 Git 已经做了基本的编辑：



```

float f(vec2 fragCoord) {
    vec3 dir = ray_dir(45.0f, fragCoord);
    vec3 eye = { 0.0f, 0.0f, 4.5f };
    mat3 rot = rotationXY(-1.0f, 1.0f);

    dir = mul(rot, dir);
    eye = mul(rot, eye);

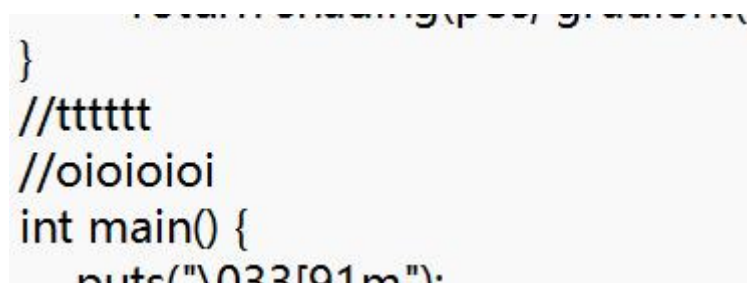
    float depth = ray_marching(eye, dir, 0.0f, clip far);
    vec3 pos = add3(eye, scale3(dir, depth));
    if (depth >= clip far)
        return 0.0f;
    else
        return shading(pos, gradient(pos), eye);
}

<<<<<< HEAD
//ttttt

=====
//oioioioi
>>>>>> C4
int main() {
    puts("\033[91m");
    for (int y = 0; y < 80; y++) {
        for (int x = 0; x < 160; x++) {
            putchar("-.-;+*#@"[((int)(f(make2((x / 160.0f - 0.5f) * 2.0f, (y / 80.0f -
0.5f) * -2.0f) * 12.0f))]);
            putchar("\n");
            system("color 0c");
        }
    }
}

```

图 3-2-9 执行完合并命令后的文件内容



```

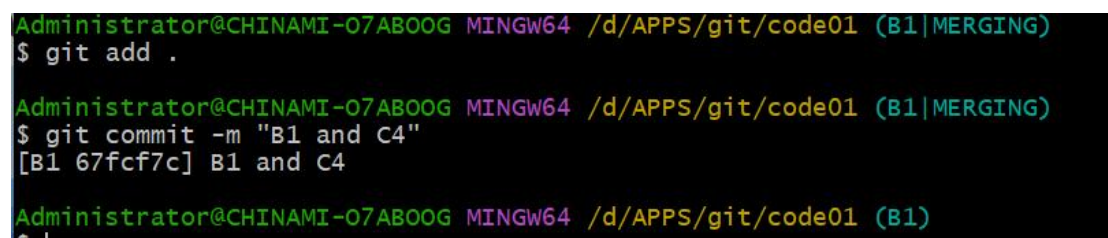
}

//ttttt
//oioioioi
int main() {
    puts("\033[91m");

```

图 3-2-10 手动消解冲突后的文件内容

这样就完成了手动解决冲突的问题。处理完冲突后提交的方式和一般提交文件到本地仓库的操作一致，使用的语句为：`git add 文件名` 和 `git commit -m "提交备注信息"`。需要注意的是一般在手动处理冲突时需要开发人员一次性处理完所有的冲突文件，也就是在 `git commit` 的时候一般不会指定特定的文件名。



```

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (B1|MERGING)
$ git add .

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (B1|MERGING)
$ git commit -m "B1 and C4"
[B1 67fcf7c] B1 and C4

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (B1)
$

```

图 3-2-11 手动消解冲突后提交文件

可以发现解决冲突后命令行的提示尾部从"(B1) |MERGING)"变成了"(B1)"说明分支合并、冲突处理完毕。

(7) R7:

在 B2 分支上对某个文件做修改并提交;

首先切换到 B2 分支,随后手动在 xy.txt 中加入一行注释进行修改,修改后通过 git add 文件名 和 git commit -m ""进行提交。

```
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B1)
$ git checkout B2
Switched to branch 'B2'

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$ git add .

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$ git status
On branch B2
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   xy.txt

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$ git commit -m "R7"
[B2 c0ba5ab] R7
1 file changed, 2 insertions(+), 1 deletion(-)

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$
```

图 3-2-12 切换到 B2 分支并修改提交文件

(8) R8:

查看目前哪些分支已经合并、哪些分支尚未合并。

查看已经合并的分支: git branch --merged

查看尚未合并的分支: git branch --no-merged

分别执行上述两条指令,得到如下的结果:

```
Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$ git branch --merged
* B2
  master

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
$ git branch --no-merged
B1
C4

Administrator@CHINAMI-07ABOOG MINGW64 /d/APPS/git/code01 (B2)
```

图 3-2-13 查看合并/未合并分支

(9) R9:

将 C4 和 B1 合并后的分支删除,将尚未合并的分支合并到一个新分支上,分支名字为你的学号。

上述全部过程中一共有 4 个分支,分别是 master 分支、B1 分支、B2 分支、C4 分支。

这一步删除了 B1 和 C4 合并后的分支,因此需要合并的分支是 master 和 B2 分支。

删除分支的指令: git branch -d 分支名 强制删除分支的指令: git branch -D 分支名。


```
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git branch -D B1
Deleted branch B1 (was 67fcf7c).
```

图 3-2-14 删除分支 B1

随后继续采用 `git merge` 合并 B2 与 master, 手动消解冲突后利用指令 `git branch -m 旧名称 新名称` 改变分支名。

```
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git merge B2
Updating 36fa031..c0ba5ab
Fast-forward
 xy.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git branch
  B2
  C4
* master

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git merge B2
Already up to date.

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (master)
$ git branch -m master 2022211830

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (2022211830)
$
```

图 3-2-15 改变分支名

发现修改后命令行的末尾变成了 (2022211830), 说明分支名修改成功。

(10) R10:

将本地以你的学号命名的分支推送到 GitHub 上自己的仓库内;

上一步结束后分支以切换至学号分支, 随后利用指令 `git remote add origin 仓库地址` 和 `git push -u origin 学号推送`。

```
Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (2022211830)
$ git remote add origin https://github.com/OSSDP/Lab1-2022211830.git

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (2022211830)
$ git push -u origin 2022211830
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 20 threads
Compressing objects: 100% (18/18), done.
Writing objects: 100% (20/20), 4.50 KiB | 4.50 MiB/s, done.
Total 20 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (4/4), done.
remote:
remote: Create a pull request for '2022211830' on GitHub by visiting:
remote:   https://github.com/OSSDP/Lab1-2022211830/pull/new/2022211830
remote:
To https://github.com/OSSDP/Lab1-2022211830.git
 * [new branch]      2022211830 -> 2022211830
branch '2022211830' set up to track 'origin/2022211830'.

Administrator@CHINAMI-O7ABOOG MINGW64 /d/APPS/git/code01 (2022211830)
$
```

图 3-2-16 推送至 GitHub

(11) R11:

查看完整的版本变迁树：git log --graph --oneline --all

```
Administrator@CHINAMI-07AB00G MINGW64 /d/APPS/git/code01 (2022211830)
$ git log --graph --oneline --all
* c0ba5ab (HEAD -> 2022211830, origin/2022211830, B2) R7
* 50fe24a (C4) 2_1
* 5dffffb5 2_1
* f2338d6 2_1
/
* 36fa031 2nd
* cab91a6 my first commit
```

图 3-2-17 版本变迁树

(12) R12:

在 Github 上以 web 页面的方式查看你的 Lab1 仓库的当前状态。

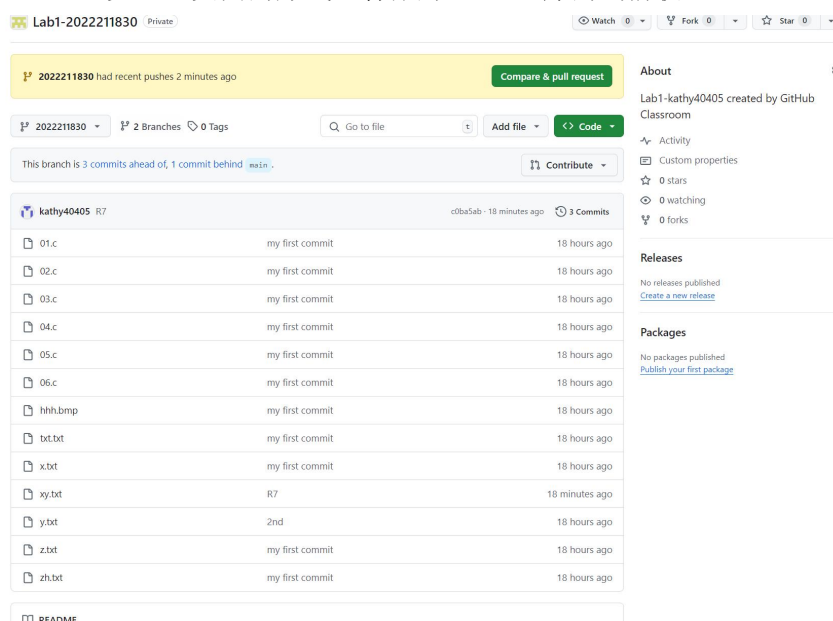


图 3-2-18 web 页面上的仓库状态

3.3 实验场景(3): 在线 Git 练习

给出完成的所有任务的命令，格式如下：

(一) 主要页面-基础篇

任务 1: Git Commit

操作命令集

```
git commit
```

```
git commit
```

任务 2: Git Branch

操作命令集

```
git branch bugFix
```

```
git checkout bugFix
```

任务 3: Git Merge**操作命令集**

```
git checkout -b bugFix
git commit
git checkout main
git commit
git merge bugFix
```

任务 4: Git Rebase**操作命令集**

```
git checkout -b bugFix
git commit
git checkout main
git commit
git checkout bugFix
git rebase main
```

(二) 主要页面-高级篇**任务 1: 分离 HEAD****操作命令集**

```
git checkout c4
```

任务 2: 相对引用 (^)**操作命令集**

```
git checkout bugFix
git checkout HEAD^
```

任务 3: 相对引用 2 (~)**操作命令集**

```
git branch -f main c6
git branch -f bugFix c0
git checkout c1
```

任务 4: 撤销变更**操作命令集**

```
git reset c1
git checkout pushed
git revert c2
```

(三) 主要页面-移动提交记录**任务 1: Git Cherry-pick****操作命令集**

```
git cherry-pick c3 c4 c7
```

任务 2: 交互式的 rebase**操作命令集**

```
git rebase -i HEAD~4^
```

(四) 主要页面-杂项**任务 1: 本地栈式提交****操作命令集**

```
git rebase -i HEAD~3
```

```
git branch -f main bugFix
```

任务 2: 提交的技巧 #1

操作命令集

```
git rebase -i HEAD~2
```

```
git commit --amend
```

```
git rebase -i HEAD~2
```

```
git branch -f main
```

任务 3: 提交的技巧 #2

操作命令集

```
git checkout main
```

```
git cherry-pick newImage
```

```
git commit --amend
```

```
git cherry-pick caption
```

任务 4: Git Tags

操作命令集

```
git tag v0 c1
```

```
git tag v1 c2
```

```
git checkout c2
```

任务 5: Git Describe

操作命令集

```
git commit
```

(五) 主要页面-高级话题*

任务 1: 多分支 rebase

操作命令集

```
git rebase main bugFix
```

```
git rebase bugFix side
```

```
git rebase side another
```

```
git branch -f main another
```

任务 2: 选择 parent 提交记录

操作命令集

```
git branch bugWork HEAD~^2~
```

任务 3: 纠缠不清的分支

操作命令集

```
git checkout one
```

```
git cherry-pick c4 c3 c2
```

```
git checkout two
```

```
git cherry-pick c5 c4 c3 c2
```

```
git branch -f three c2
```

(六) 远程页面-Git 远程仓库

任务 1: Git Clone

操作命令集

```
git clone
```

任务 2: 远程分支

操作命令集

```
git commit
git checkout o/master
git commit
```

任务 3: Git Fetch**操作命令集**

```
git fetch
```

任务 4: Git Pull**操作命令集**

```
git pull
```

任务 5: 模拟团队合作**操作命令集**

```
git clone
git fakeTeamwork 2
git commit
git pull
```

任务 6: Git Push**操作命令集**

```
git commit
git commit
git push
```

任务 7: 偏离的提交历史**操作命令集**

```
git clone
git fakeTeamwork 1
git commit
git pull --rebase
git push
```

任务 8: 锁定的 Main**操作命令集**

```
git reset --hard o/main
git checkout -b feature C2
git push origin feature
```

(七) 远程页面-Git 远程仓库高级操作**任务 1: 推送主分支****操作命令集**

```
git fetch
git rebase o/master side1
git rebase side1 side2
git rebase side2 side3
git rebase side3 master
git push
```

任务 2: 合并远程仓库**操作命令集**

```
git checkout main
git pull origin main
git merge side1
git merge side2
git merge side3
git push origin main
```

任务 3: 远程追踪

操作命令集

```
git checkout -b side o/main
git commit
git pull --rebase
git push
```

任务 4: Git Push 的参数

操作命令集

```
git push origin main
git push origin foo
```

任务 5: Git Push 的参数 2

操作命令集

```
git push origin foo:main
git push origin main^:foo
```

任务 6: Git Fetch 的参数

操作命令集

```
git fetch origin c3:foo
git fetch origin c6:main
git checkout foo
git merge main
```

任务 7: 没有 Source 的 Source

操作命令集

```
git pull origin :bar
git push origin :foo
```

任务 8: Git Pull 的参数

操作命令集

```
git pull origin c3:foo
git pull origin c2:side
```

(八) 通关后的主界面截图

完成规定任务后，“主页”和“远程”截图各一张，如下图所示：



4 小结

对本次实验过程和结果的思考：

(1) 比较之前的开发经验，使用 Git 的优点？

Git 是一种分布式版本控制系统，每个开发者都可以在本地拥有完整的版本库。这意味着开发者可以在没有网络连接的情况下进行版本控制和提交更改，然后在连接到网络时与远程仓库同步。Git 提供了强大的合并工具，可以智能地解决代码冲突，并提供多种合并策略来帮助开发者有效地合并代码。Git 是开源软件，拥有庞大的社区支持和活跃的开发者社区。这意味着有大量的文档、教程和第三方工具可供使用，帮助开发者更好地利用 Git 进行版本控制。

(2) 在个人开发和团队开发中，Git 起到的作用有何主要差异？

在权限管理方面，个人开发者拥有对他们自己的代码库的完全访问权限，不需要考虑权

限管理的问题。在团队中，需要对不同的开发者分配不同的权限，以确保只有授权的成员能够对代码库进行更改。Git 提供了权限控制的机制，例如分支保护和权限管理工具。

在协作合并方面，个人开发者通常不需要频繁地进行代码合并，因为他们只需要管理自己的代码库。但是，他们可能会使用分支来管理不同的功能或实验性的更改。团队中的多个开发者会同时修改代码库中的不同部分，因此需要频繁进行代码合并和解决可能出现的冲突。Git 提供了强大的分支和合并功能，使得团队成员能够高效地协作。

(3) 之前是否用过其他的版本控制软件？如果有，同 Git 相比有哪些优缺点？如果没有查阅资料对比一下不同版本控制系统的差别。

使用过 SVN。

优点：相对简单易用，学习曲线较平缓；集中式版本控制，对于一些团队来说可能更直观，特别是那些从集中式版本控制转换过来的团队；提供了更好的文件锁定机制，避免了冲突问题。

缺点：不支持离线工作，需要与服务器连接才能执行版本控制操作；分支和合并操作相对复杂，不如 Git 灵活和高效；对于大型仓库和大量文件的处理效率不如 Git。

(4) 在什么情况下适合使用 Git、什么情况下没必要使用 Git？

Git 适合大多数团队和项目，特别是需要灵活分支管理和多人协作开发的情况下。但是，对于一些小型、个人或者对版本控制不熟悉的团队来说，并没有使用 git 的必要。