[TOC]

(一) Git学习

1.常用git命令

设置全局用户名：git config --global user.name "Your Name"

设置全局用户邮箱：git config --global user.email "your_email@example.com"

本地生成ssh公私钥：ssh-keygen -t rsa；私钥：~/.ssh.id_rsa 公钥：~/.ssh/id_rsa.pub 远程仓库（github）添加本地公钥

本地仓库初始化为git仓库：git init

克隆远程仓库到本地：git clone "repository-url"

给本地git仓库添加远程仓库：git remote add origin url，origin为默认远程仓库

查看远程仓库：git remote get-url origin

查看工作区、暂存区、本地分支的状态：git status

工作区加载到缓存区：git add （.）

查看本地分支：git branch

创建新的本地分支：git branch "branch_name"

删除本地分支：git branch -d "branch_name"

切换本地分支：git checkout branch_name(master)

合并指定本地分支到当前所在本地分支：git merge "branch_name"

缓存区提交到本地分支：git commit -m "message"

本地分支推送到远程仓库：git -u origin my_branch，-u表示绑定远程仓库，下次push不需要写远程仓库名

将远程主机的最新内容拉到本地(注意还没有合并到本地分支)：git fetch "remote" "remote_branch" 查看最新内容的更改：git log -p FETCH_HEAD 继续使用git merge FETCH_HEAD把更新合并到当前所在本地分支

从远程仓库拉取并合并更改：git pull "remote_name" "remote_branch:local_branch" git pull相当于git fetch + git merge

查看提交记录：git log

本地分支回退到指定版本：git reset commit号

(二) Linux环境配置：wsl2 Ubuntu22.04

1.wsl2与windwos互传文件

/mnt 目录是用于访问 Windows 文件系统的挂载点。WSL 允许你在 Windows 和 Linux 之间共享文件，/mnt 目录下的子目录分别代表了 Windows 的不同驱动器。例如，/mnt/c 表示 Windows 的 C:\ 驱动器，/mnt/d 表示 D:\ 驱动器，以此类推。通过这些挂载点，你可以在 WSL 中访问和操作 Windows 文件系统中的文件和文件夹。

(三) 命令行参数解析

一、getopt()函数

1.头文件：unistd.h/getopt.h

2.原型：int getopt(int argc, char * const argv[], const char *optstring);

3.参数：argc、argv为main函数参数，optstring为指定要解析的指令选项 optstring格式：

```
"vha:b::c"
```

其中每一个字母为一个短选项(-v、-h、-a、-b、-c) 字母后面1个:代表该选项需要参数，运行格式为-a100或-a 100 2个:代表参数为可选（可带可不带），运行格式只能为-b200， 没有:代表不需要参数

4.四个全局变量：char* optarg、int optind、int opterr、int optopt optarg：若选项后面有参数，则optarg指向该参数 optind：循环使用getopt()函数扫描短选项时，为"下一个"选项在argv中的下标；扫描结束后，为第一个非选项参数的argv下标 例如：./out -h 10 -v 1 2 getopt扫描到-h时，optind为3，指向-v，所有选项扫描结束后，optind为5 opterr：出现不可识别的选项时，getopt将打印错误信息。将opterr设为0不打印错误信息 optopt：存放不可识别的选项至optopt

5.返回值：返回选项的字母的ascii值，解析完毕返回-1

6.用法：while((opt=getopt())!=-1) switch(opt)……

7.解析过程：getopt首先扫描argv[1]到argv[argc-1]，并将选项及参数依次放到argv数组的最左边，非选项参数依次放到argv的最后边，即该函数会改变argv的排列顺序。

二、getopt_long()函数

1.原型：int getopt_long(int argc, char * const argv[],const char *optstring, const struct option *longopts,int *longindex); 2.参数： longopts：结构体option的数组，每个结构体对应一个长选项（--version） longindex：一般赋为NULL；否则，它指向一个变量，这个变量在扫描选项的过程中会被赋值为寻找到的长选项在longopts 中的索引值 3.struct option { const char *name; int has_arg; int *flag; int val; }; name为长选项的名字，如version、help has_arg为长选项是否有参数，no_argument=0,required_argument=1,optional_argument=2,为了可读性一般使用符号常量 无参数：--version 需要参数：--size=100或--size 100 可选参数：--strides=10 flag ： 如果该指针为NULL，那么getopt_long返回val的值；如果该指针不为NULL，那么会使得它所指向的变量填入val的值，同时getopt_long返回0； val ： 如果flag是NULL，那么val通常是个字符常量，否则为指定填入flag的值

eg: static const struct option longopts[]={ {"version", no_argument, NULL, 'v'}, {"size", required_argument, NULL, 's'}, {"resize", optional_argument, NULL, 'r'} {NULL, 0, NULL, 0}, // 注意这个结构体必须有 } 5.getopt_long()兼容getopt(),即短选项仍然可以识别、optarg、optind、opterr、optopt也仍然可用

(四) RGB图与灰度图

一、RGB图

1.rgb值：用r、g、b表示所有颜色，值从0-255,000为黑，255,255,255为白

二、灰度图

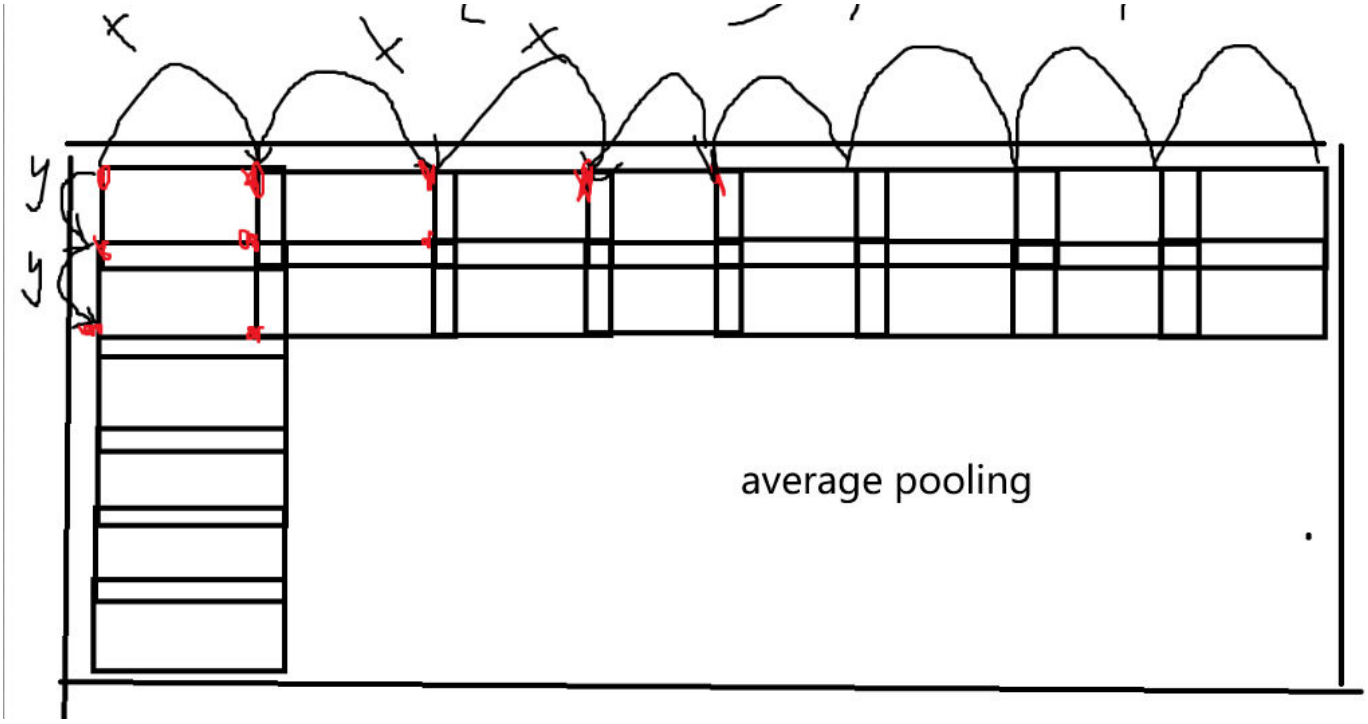
1.灰度值：一般使用加权平均，Gray = 0.299 * R + 0.587 * G + 0.114 * B

灰度值越大越白，越小越黑。

终端中使用字符点阵密度表示灰度值大小

(五) 平均池化/最大池化：

1.原理：设置pool_size池化大小和strides步长，对周围的区域进行平均化/最大化处理，通过求近似的方法实现图片大小的缩放



(六) 问题解决

Q：终端视频帧比例有问题,一般字体的长宽比为1：2 A：：(1)每个像素点输出2个字符/空格; (2)设置行步长，每2行才打印一行 Q：bad apple视频帧只能输出右边一半 A：也是比例问题

Q：如何在终端体现出灰度差异 A：设置背景为黑，字符为白色，用字符点阵密度最大的表示白色，最小的表示黑色

N：打印灰度图比打印rgb图效率高

Q：反复打印每一帧太麻烦 A：一个视频内设置取帧数步长；同一帧内设置打印字符的步长；使用更高效的输出函数，如write（）

Q：灰度图白色值大的字符不显示 A：unsigned char / 256(int)得到的结果是整数，小数部分舍弃，所以只有灰度值为255的最白色字符能显示 把256改为256.0即可

Q：dragon.mp4视频帧倾斜、rgb无颜色问题 A：把frame的linesize默认成width的3倍了，其实不是，计算字节数应该用linesize

Q：如何非阻塞读取键盘事件 A：(1)把STDIN_FILENO设置为非阻塞模式,getchar()没有字符可获取时不阻塞程序; (2)实现getch()函数，设置终端模式，键盘命令不需要按回车直接加载到标准输入缓冲区

Q：开启非阻塞读取键盘事件的设置后，rgb的图打印出现问题 A：在完成一次读取缓冲区后，要把STDIN_FILENO的模式改回阻塞模式

> 项目中，把解析视频并处理视频帧和打印视频帧分为两个线程并行执行，我的机器运行该项目，多线程比单线程速度提升7%左右

(七) 多线程

一、多线程编程

1.线程概述

（1）定义：是操作系统能够进行**运算调度**的最小单位，进程划分为两个或多个线程，一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

> [!WARNING]
>
> 进程是**资源分配**的最小单位，不要弄混

（2）进程有自己独立的地址空间, 多个线程共用同一个地址空间

（3）在一个地址空间中，

多个线程独享: 栈区, 寄存器 多个线程共享: ,堆区, 全局数据区，代码段

（4）CPU时间片：是操作系统分配给每个正在运行的进程微观上的一段CPU时间，不同进程需要抢CPU时间片才能执行程序，因此实际上不同程序并不是并行的（只有1个CPU的话）

2.创建线程

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)
```

- pthread_t类型为线程ID的类型，在linux中实际上是长整型，其他平台不一定。
- thread为线程ID类型的指针，该函数调用后该指针指向的位置被赋值为创建出的线程的ID
- attr为线程属性，默认为NULL
- start_routine为函数指针，是子线程的任务函数
- arg为传给子线程的参数，默认为NULL
- 返回值为0则创建成功，创建失败返回错误号

返回当前所在线程ID的函数：

```
pthread_t pthread_self(void);
```

3.结束线程

```
void pthread_exit(void *retval)
```

- retval为返回数据的指针，一般与pthread_join搭配使用，默认为NULL

> [!NOTE]
>
> 程序默认有一个主线程，创建出子线程后，如果主线程先结束，虚拟地址空间就会被释放，子线程也会自动结束，主线程使用pthread_exit则会退出而不导致虚拟地址空间被释放

4.回收线程

```
int pthread_join(pthread_t thread, void **retval)
```

- thread为线程ID
- retval为二级指针，即pthread_exit中retval指针的地址
- 成功回收返回0，失败返回错误值

调用该函数会使主线程阻塞（主线程调用），直到指定的子线程执行完毕主线程再解除阻塞状态，并且进行子线程资源的回收

获取子线程返回的数据常见有两种方式：

（1）使用全局变量

```
全局变量：int a;
子线程：pthread_exit(&a);
主线程：int* ret = NULL;
          pthread_join(&ret);
          printf("a=%d", *ret);
```

（2）使用主线程的栈

```
主线程：int a;
          pthread_create(id, NULL, func, &a);
          int* ret;
          pthread_join(id, &ret);
          printf("a=%d", *ret);
子线程：int* p = (int*)arg;
          *p=...
          pthread_exit(p);
```

> [!WARNING]
>
> 不能使用子线程的栈来返回数据，因为子线程结束后，其栈被释放后被分给其他线程

5.分离线程

```
int pthread_detach(pthread_t thread)
```

- thread为线程ID
- 成功分离返回0，失败返回错误值

默认情况下，子线程需要让主线程回收其资源（通过pthread_join）才能结束，但pthread_join会阻塞主线程；使用pthread_detach后子线程与主线程分离，不能被主线程的pthread_join回收资源，而是由系统自动回收其资源

6.取消（杀死）线程

```
int pthread_cancel(pthread_t thread);
```

- thread为线程ID
- 成功杀死返回0，失败返回错误值

注意使用pthread_cancel并不会立即结束子线程，只有子线程进行了系统调用时才会结束。例如子线程使用printf（）

7.比较线程PID

```
int pthread_equal(pthread_t t1, pthread_t t2)
```

- t1、t2为线程ID
- 相同返回非0，不相同返回0

考虑到不同平台pthread_t的类型可能不同，因此专门有一个判断线程是否相同的函数

> 因为两个线程，一个线程要写数据（解析视频，把视频帧处理后写入buffer），另一个线程读数据（取出buffer中的视频帧打印），自然考虑到可能产生冲突，于是学习了一下线程同步；但是后续在项目中发现，使用循环队列来作为 buffer的话，不需要线程同步也能起到多线程优化的效果，因此线程同步没有使用到。

二、线程同步

1.线程同步概述

（1）线程同步定义：同一进程的多个线程同时对它们的共享资源（堆、数据区、代码区）进行访问时，让这些线程按前后顺序对内存进行访问

（2）目的：保护共享资源，避免数据竞争，保证程序正确性

（3）同步方式：可能会发生数据竞争的共享资源被称为**临界资源**，和临界资源相关的上下文代码被称为**临界区**

2.互斥锁

3.读写锁

4.条件变量

5.信号量

在临界区代码的上边，添加加锁函数，对临界区加锁。 哪个线程调用这句代码，就会把这把锁锁上，其他线程就只能阻塞在锁上了。 在临界区代码的下边，添加解锁函数，对临界区解锁。 出临界区的线程会将锁定的那把锁打开，其他抢到锁的线程就可以进入到临界区了。 通过锁机制能保证临界区代码最多只能同时有一个线程访问，这样并行访问就变为串行访问了。

2.互斥锁

可以锁定一个代码块, 被锁定的这个代码块, 所有的线程只能顺序执行(不能并行处理)，这样多线程访问共享资源数据混乱的问题就可以被解决了

```
pthread_mutex_t  mutex;         //定义1个互斥锁变量
```

锁中保存了当前这把锁的状态信息：锁定or打开，如果是锁定状态还记录了给这把锁加锁的线程信息（线程ID）。一个互斥锁只能被一个线程锁定，被锁定之后其他线程再对互斥锁变量加锁就会被阻塞，直到这把互斥锁被解锁，被阻塞的线程才能被解除阻塞。

一般情况下，每一个共享资源对应一个把互斥锁，锁的个数和线程的个数无关。

```
// 初始化互斥锁
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);
// 释放互斥锁
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- mutex：互斥锁的地址
- attr：互斥锁的属性，默认为NULL

```
// 当前线程给互斥锁上锁，如果已经被其他线程上锁，则本线程阻塞
int pthread_mutex_lock(pthread_mutex_t *mutex);
// 当前线程给互斥锁解锁
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

> [!IMPORTANT]
>
> 死锁：所有的线程都被阻塞，并且线程的阻塞是无法解开的（可以解锁的线程也被阻塞），情况如下：
>
> - 加锁之后忘记解锁（其他线程被阻塞）
> - 重复加锁, 造成死锁（加锁的线程把自己也阻塞了）
> - 多个共享资源/互斥锁，不同线程互相阻塞（线程1给A上锁，线程2给B上锁，线程1被B阻塞，线程2被A阻塞）

3.读写锁

读写锁为互斥锁的进阶版，允许读取共享资源的读取是并行的

```
pthread_rwlock_t rwlock;
// 初始化读写锁
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
           const pthread_rwlockattr_t *restrict attr);
// 释放读写锁占用的系统资源
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

读写锁本身只是1把锁，但是有不同的上锁方式：对读取上锁/对写入上锁

读写锁记录了：锁的状态、上锁线程、上锁方式

- 不同线程都对读取上锁，则并行，相当于没有锁
- 不同线程都对写入上锁，则串行，防止竞争
- 有读有写，则写入优先级高，先执行

所以读写锁在大量读取操作的程序中有优势

```
// 在程序中对读写锁加读锁，锁定的是读操作
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
// 在程序中对读写锁加写锁，锁定的是写操作
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
// 解锁，不管锁定了读还是写都可用解锁
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

4.条件变量

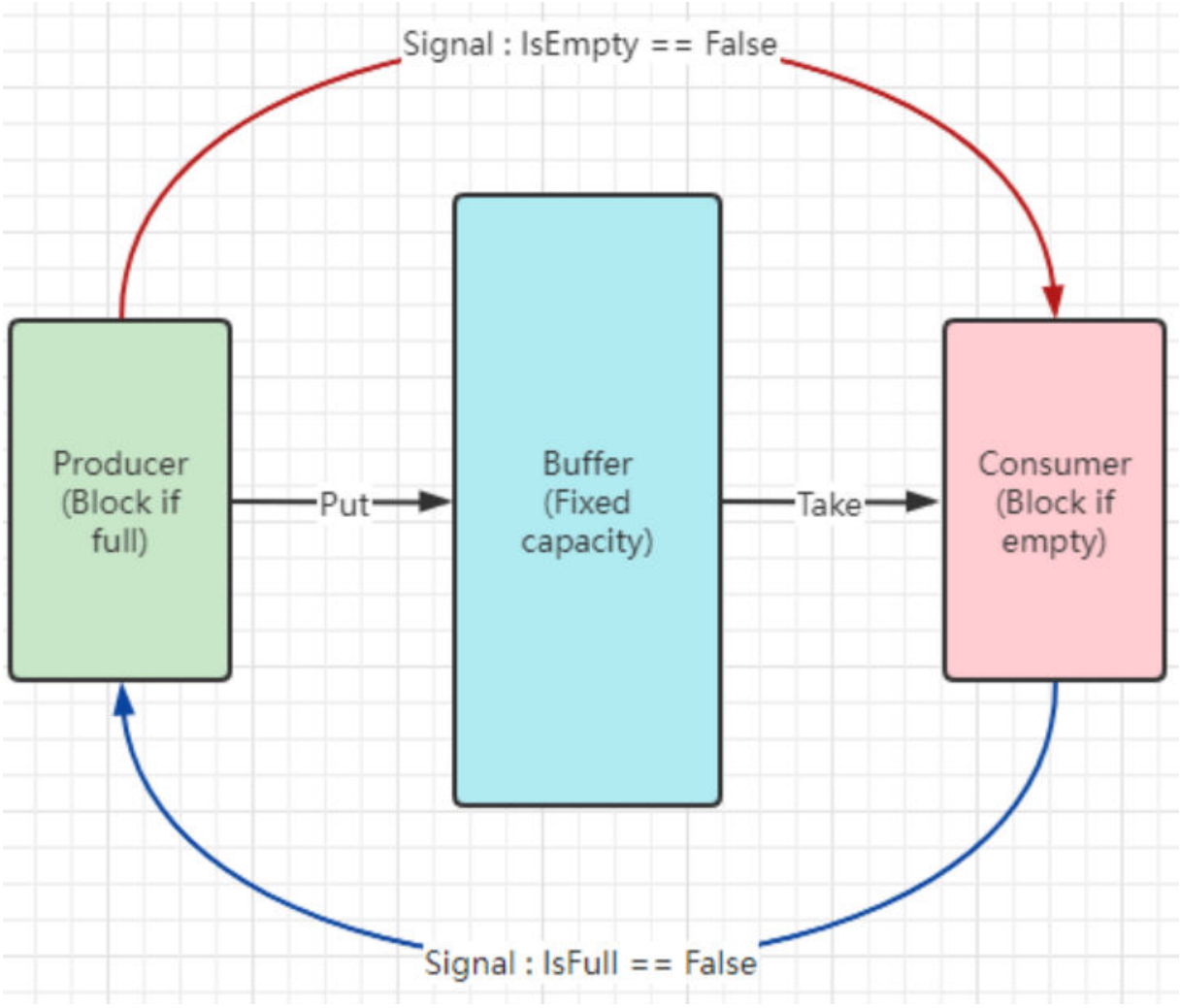
与互斥锁搭配使用，用于在一定条件下阻塞/唤醒程序

```
pthread_cond_t cond;
// 初始化
int pthread_cond_init(pthread_cond_t *restrict cond,
      const pthread_condattr_t *restrict attr);
// 销毁释放资源
int pthread_cond_destroy(pthread_cond_t *cond);



// 线程阻塞函数，哪个线程调用这个函数，哪个线程就会被阻塞
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

// 唤醒阻塞在条件变量上的线程，至少有一个（>=1）被解除阻塞
int pthread_cond_signal(pthread_cond_t *cond);
// 唤醒阻塞在条件变量上的线程，被阻塞的线程全部解除阻塞
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- 线程用pthread_cond_wait阻塞自己时，对指定的互斥锁解锁，这是为了保证其他线程能够进入临界区，防止死锁，并且在一定条件下唤醒阻塞的线程
- 线程被pthread_cond_signal或pthread_cond_broadcast唤醒时，会把指定的互斥锁上锁，防止竞争

为什么要这样设计？

> [!IMPORTANT]
>
> ### 生产者——消费者模型：
>
> **生产者负责写入，消费者负责读取，因此二者需要互斥锁。**
>
> **生产者写满buffer时，生产者需要自己阻塞自己，并且让消费者取消阻塞。**
>
> **消费者读完buffer时，消费者需要自己阻塞自己，并且让生产者取消阻塞。**



条件变量在这里就发挥了它的作用

```
生产者线程：
// 进入临界区，互斥锁上锁
pthread_mutex_lock(&mutex);
// 阻塞条件：buffer为满
        pthread_cond_wait(&cond, &mutex);
// 生产了任务，通知消费者消费
pthread_cond_broadcast(&cond);
// 互斥锁解锁
pthread_mutex_unlock(&mutex);

消费者线程：
// 进入临界区，互斥锁上锁
pthread_mutex_lock(&mutex);
// 阻塞条件：buffer为空
        pthread_cond_wait(&cond, &mutex);
// 消费了任务，通知生产者生产
pthread_cond_broadcast(&cond);
// 互斥锁解锁
pthread_mutex_unlock(&mutex);
```

5.信号量

暂时还没看.....