实现简易的协程库（面向C或C++），除了支持基本的**创建协程（co_create）、启动协程（co_resume）、协程挂起（co_yield）**功能，还需要实现能够用于**协程间通信的队列类型（包括匹配的共享资源加锁、解锁功能）**，实现能够在协程中调用的**异步文件I/O** API。最后编写基于该协程库的**生产者**（从文件中读数据并通过队列传送给消费者）**消费者**（从队列中取出数据进行处理，并将处理结果写入另一个文件）示例程序。

程序模块设计及实现

协程和协程调度器数据结构

协程的实现主要依赖于Linux 下提供的一套函数，叫做 **ucontext 簇函数**，可以用来获取和设置当前线程的上下文内容。

在头文件< ucontext.h > 中定义了两个结构类型 mcontext_ t 和 ucontext_t 及四个函数 getcontext(), setcontext(), makecontext(), swapcontext(). 利用它们可以在一个进程中实现用户级的线程切换。

ucontext_t结构体:

```
typedef struct ucontext {
    struct ucontext *uc_link;       //uc_link指向当前的上下文结束时要恢复到的上下文
    sigset_t         uc_sigmask;    //该上下文中的阻塞信号集合
    stack_t          uc_stack;      //该上下文中使用的栈
    mcontext_t       uc_mcontext;   //保存具体的程序执行上下文，如PC值，堆栈指针以及寄存器值等信息。它的实现依赖于底层，是平台硬件相关的。
    ...
} ucontext_t;
```

- `int getcontext(ucontext_t *ucp);//获取调用方当前的上下文，并保存到ucp中`
- `int setcontext(const ucontext_t *ucp);//设置当前的上下文为ucp`
- `void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);//初始化一个ucontext_t, func参数指明了该context的入口函数，argc为入口参数的个数，每个参数的类型必须是int类型。`
- `int swapcontext(ucontext_t *oucp, ucontext_t *ucp);//保存当前上下文到oucp结构体中，然后激活ucp上下文`

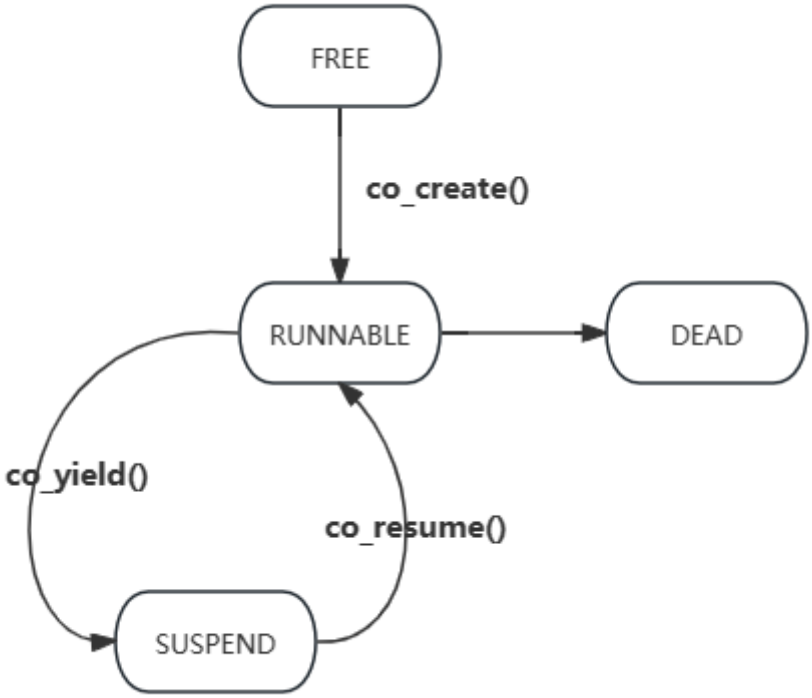
所以我们设计自己的协程和协程调度器数据结构为：

```
//协程的定义
typedef struct Co
{
    ucontext_t ctx;          //协程上下文
    Fun func;                //协程用户执行的函数
    void *arg;               //函数参数
    enum CoState state;      //协程状态
    char stack[STACK_SZIE]; //协程的栈
}co;
```

```
//协程调度器
class Schedule
{
    private:
        ucontext_t main;     //当前运行协程的上下文
        int running_co;      //正在运行协程的id
        int cap;             //调度器容量
        co *cos;             //存放协程
        std::queue<Data> q; //协程间通信的队列
        int fdr;             //读文件操作符
        int fdw;             //写文件操作符
        int offset;          //异步IO的偏移量
        std::mutex mutex_;  //互斥锁
        std::condition_variable condition_;//条件变量
};
```

协程创建、启动、挂起

协程定义了三种状态，整个运行期间，根据这三种状态进行轮转，分别是 FREE：创建态，RUNNABLE：运行态，SUSPEND：挂起态



协程创建（FREE ->RUNNABLE）

协程创建通过协程调度器的 `int co_create(Fun func, void *arg)` 函数实现，创建后由协程调度器统一保存和管理，创建协程可以像线程一样指定执行函数和参数，返回协程id。实现步骤为：

1. 协程调度器中找到一个空闲位置放入协程。如若预先定义的协程数量不足以满足需求，还可以自动2倍扩容。
2. 创建协程后使用 `getcontext()` 获取当前的上下文设置对应参数，然后保存上下文
3. 使用 `makecontext()` 设置对应 ucontext 的执行函数和参数
4. 使用 `swapcontext()` 保存上下文并切换协程

```
getcontext(&(t->ctx)); //获取调用方当前的上下文，并保存到协程的ctx中
t->state = RUNNABLE;//变换状态
t->func = func;//绑定函数
t->arg = arg;//绑定参数

t->ctx.uc_stack.ss_sp = t->stack;
t->ctx.uc_stack.ss_size = STACK_SZIE;
t->ctx.uc_stack.ss_flags = 0;
t->ctx.uc_link = &main;

running_co = id;
// 用来设置对应 ucontext 的执行函数和参数
makecontext(&(t->ctx),(void (*)(void))(co_func), 1, this);
swapcontext(&main, &(t->ctx));//保存当前上下文到main中，然后激活t->ctx上下文， 切换协程
```

协程启动（SUSPEND -> RUNNABLE）

对挂起的协程调用 `co_resume()` 会再次唤醒协程，主要使用 `swapcontext(&main,&(t->ctx))` 保存当前上下文到main中，然后激活该协程的上下文启动协程。

```
running_co = id;
t->state = RUNNABLE;
swapcontext(&main,&(t->ctx)); //保存当前上下文到main中，然后激活t->ctx上下文
```

协程挂起（RUNNABLE -> SUSPEND）

调用 `co_yield()` 将当前的协程挂起，此时程序会回到主协程，不支持协程嵌套。协程挂起都会回到主协程，同样的协程启动也是从主协程切换过来的。

```
co *t = &cos[running_co];//找到该协程
t->state = SUSPEND;//挂起
//回到主协程
running_co = -1;
swapcontext(&(t->ctx),&main);//保存当前上下文到t->ctx，然后激活main上下文， 切换到主协程
```

协程间通信

协程简单通信简单的使用一个队列进行，该队列由协程调度器管理，通信数据类型定义在`Data`结构体中。

```
std::queue<Data> q; //协程间通信的队列
```

通信队列的读写使用了互斥锁和条件变量来避免安全问题，同时也会对后面实现消费者从队列中取数据在写入文件有帮助，不需要再单独处理队列中拿不到数据写空的情况。

```
std::mutex mutex_;  //互斥锁
std::condition_variable condition_;//条件变量
```

使用`push()`向队列中放入数据，放入数据前会加锁，放入成功后解锁并通知等待的协程

```
std::unique_lock<std::mutex> lock(mutex_);
q.push(Data(x));
lock.unlock();
condition_.notify_one(); // 通知等待的协程
```

使用`pop()`从队列中取数据，如果队列为空则会阻塞等待。

```
std::unique_lock<std::mutex> lock(mutex_);
// 队列空 pop 操作会阻塞等待
while (q.empty()) {
    condition_.wait(lock);
}
Data value = q.front();
q.pop();
```

异步文件I/O

所谓异步I/O即我们在调用I/O操作时(读或写)我们的程序不会阻塞在当前位置，而是在继续往下执行。

异步I/O的实现采用Linux下的aio异步读写实现

API函数	说明
aio_read	异步读操作
aio_write	异步写操作
aio_error	检查异步请求的状态
aio_return	获得异步请求完成时的返回值

上述的每个API都要用 aiocb 结构体赖进行操作，aiocb的结构中常用的成员有：

```
struct aiocb
{
    //要异步操作的文件描述符
    int aio_fildes;
    //用于lio操作时选择操作何种异步I/O类型
    int aio_lio_opcode;
    //异步读或写的缓冲区的缓冲区
    volatile void *aio_buf;
    //异步读或写的字节数
    size_t aio_nbytes;
    //异步通知的结构体
    struct sigevent aio_sigevent;
}
```

- `int aio_read(struct aiocb *paiocb);//该函数请求对文件进行异步读操作，若请求失败返回-1，成功则返回0，并将该请求进行排队，然后就开始对文件的异步读操作.需要先对aiocb结构体进行必要的初始化`
- `int aio_write(struct aiocb *paiocb);`
- `int aio_error(struct aiocb *aiopcb);//检查当前AIO的状态，可用于查看请求是否成功，返回0(成功)EINPROGRESS(正在读取)`
- `ssize_t aio_return(struct aiocb *paiocb);//查看一个异步请求的返回值，如果成功则返回读取字节数，否则返回-1`

在协程调度器中声明读/写文件操作符，在协程进行读写操作时，都是使用同一个文件操作符。偏移量方便对文件连续读取。

```
int fdr;            //读文件操作符
int fdw;            //写文件操作符
int offset;         //异步IO的偏移量
```

同时还在协程调度器中声明了三个public的静态变量，分别用于表示文件所有内容读取完毕、单次读写完成

```
static int flag;            //文件已读完标志
static int readed;          //文件读完通知信号
static int writed;          //文件写完通知队列
```

异步读

异步读调用 `void read(const char* filename, char* data);` 函数，结果由参数 `data` 带回。实现步骤为：

1. 对 aiocb 结构体进行必要的初始化
2. open 文件
3. 设置信号通知相关结构和处理函数，在处理函数中完成异步读完成后的逻辑，这里简单的设置一个读完成的标志

4. 异步读取文件

```
// 1. aiocb 结构体进行必要的初始化
struct aiocb cb;
cb.aio_fildes = fdr;
cb.aio_buf = data;
cb.aio_nbytes = DATA_SIZE;
cb.aio_offset = offset;
// 偏移量增加，往后读取文件内容
offset+=DATA_SIZE;

// 信号通知相关结构
cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
cb.aio_sigevent.sigev_signo = SIGUSR1;
cb.aio_sigevent.sigev_value.sival_ptr = &cb;

// 2. open
fdr = open(filename, O_RDONLY);

// 3. 设置信号处理函数
struct sigaction sa;
sa.sa_flags = SA_SIGINFO;
sa.sa_sigaction = aio_readed;
sigemptyset(&sa.sa_mask);
sigaction(SIGUSR1, &sa, nullptr);

// 4. 异步读取文件
int ret = aio_read(&cb);
```

信号处理函数 `aio_readed()` 会接收 `SIGUSR1` 信号，文件所有内容读取完成后将flag设置为0，异步读结束后将readed置为1。

```
// 读完所有文件设置标志
if(aio_return(cb)==0) {
    Schedule::flag = 0;
    return;
}
// 异步读结束标志
if (info->si_signo == SIGUSR1) {
    Schedule::readed = 1;
}
```

异步写

异步写和异步读类似，微小的区别在于打开文件时，flags设置了O_APPEND，则不需要设置偏移量，会以追加的方式写入。

```
// 1. aiocb 初始化
struct aiocb cb;
// 2. open
fdw = open(filename, O_WRONLY | O_APPEND);
// 3. 设置信号处理函数
struct sigaction sa;
// 4. 异步写文件
int ret = aio_write(&cb);
```

信号处理函数 `aio_writed()` 会接收 `SIGUSR2` 信号，异步写结束后将writed置为1。

```
if (info->si_signo == SIGUSR2) {
    Schedule::writed = 1;
}
```

问题总结

1. `swapcontext(&(t->ctx),&main);` **这行代码出现不能往下执行的情况**

   通过网上查阅资料，似乎协程正在执行异步读/写的过程中，不能切换。

   只能在生产者消费者中阻塞等待单次异步读写完成，任务完成后处于可切换状态再主动让出

2. `makecontext(ucontext_t *ucp, void (*func)(), int argc, ...)` **的入口函数参数不能传入一个类内函数**

   因为对于 `makecontext()` 函数，直接使用成员函数指针是不够的，因为成员函数指针需要实例才能被正确调用。

   将该函数定义为一个类外普通函数即可。

3. **使用** `makecontext()` **初始化一个** `ucontext_t` **时报段错误**

   `makecontext` 之前必须调用 `getcontext` 初始化 `ucontext_t`。

   如果上下文是 `getcontext()` 产生的，切换到该上下文，程序的执行在 `getcontext()` 后继续执行。
   如果上下文是 `makecontext()` 产生的，切换到该上下文，程序的执行切换到 `makecontext()` 调用所指定的第二个参数的函数上。当该函数返回时，继续 `makecontext()` 中的第一个参数的上下文中 `uc_link` 所指向的上下文。

   不产生新函数的上下文切换用 `getcontext()` 和 `setcontext()`

   产生新函数的上下文切换用 `getcontext()`， `makecontext()` 和 `swapcontext()`

4. **只使用** `aio_read();` **函数异步读时，读至文件末尾继续调用该函数偏移量无意义，会读取空值**

   配合 `aio_return()` 函数进行判断。该函数的功能是查看一个异步请求的返回值，如果成功则返回读取字节数，否则返回-1

运行结果和使用说明

首先，`schedule_test()`函数创建一个协程调度器，然后创建生产者协程，生产者对文件异步读，读取完成后写入队列，然后，生产者协程主动让出回到主协程，主协程创建消费者协程，消费者从队列中取出数据异步写到对应文件，消费者再主动让出。一直重复直到文件内容全部读取完成。

```
// 生产者
// 从producer.txt文件中读取数据，写入队列
void producer(void * arg)
{
    Schedule* s = (Schedule*)arg;
    char data[DATA_SIZE];
    while(Schedule::flag) {
        // 异步读
        s->read(PRODUCER,data);
        // 阻塞等待读完再写入队列
        while(!Schedule::readed);
        // 将异步读完标记清零方便下次读
        Schedule::readed = 0;
        printf("read---%s\n",data);
        // 写入队列
        s->push(data);
        printf("push---%s\n",data);

        // 主动让出
        s->co_yield();
    }

}
// 消费者
// 从队列中获取数据，写入consumer.txt文件
void consumer(void *arg)
{
    Schedule* s = (Schedule*)arg;
    char data[DATA_SIZE];
    while(1) {
        // 从队列中获取数据
        strncpy(data, s->pop(), DATA_SIZE);
        printf("pop---%s\n",data);
        // 异步写操作
        s->write(CONSUMER,data);
        // 阻塞等待异步写完成后将标记清零
        while(!Schedule::writed);
        Schedule::writed = 0;
        printf("write---%s\n",data);
        // 主动让出
        s->co_yield();
    }

}

void schedule_test()
{
    // 协程调度器
    Schedule s;
    // 生产者协程
    int id1 = s.co_create(producer, &s);
```

```
    // 消费者协程
    int id2 = s.co_create(consumer, &s);

    // 对未执行完的协程反复唤醒执行
    while(!s.co_finished()){
        s.co_resume(id1);
        s.co_resume(id2);
    }

}
int main()
{
    schedule_test();

    return 0;
}
```

使用：

1. 环境 `ubuntu 64位`
2. 创建`producer.txt`和`consumer.txt`文件，`producer.txt`文件要先写入需要读取的文件内容，例如：

   ```
   abcdefghijklmnopq1234567890
   ```

3. 执行 `g++ main.cpp co.cpp -o co` 编译链接
4. 运行 `./co`

`DATA_SIZE`在程序中设置为2，每次读取2字节的数据，控制台输出日志如图，`consumer.txt`中也会写入同样的内容

```
wang@wang:~$ ./co
read---ab
push---ab
pop---ab
write---ab
read---cd
push---cd
pop---cd
write---cd
read---ef
push---ef
pop---ef
write---ef
read---gh
push---gh
pop---gh
write---gh
read---ij
push---ij
pop---ij
write---ij
read---kl
push---kl
pop---kl
```