姓名：谢志康
学号：22307110187
内容：lab3-coroutine_lab
目录：part0, part1, part3, part4, part5, part2/2.5, 猫猫
耗时：>60h
建议：希望以后尽量简单点（）要做破防了

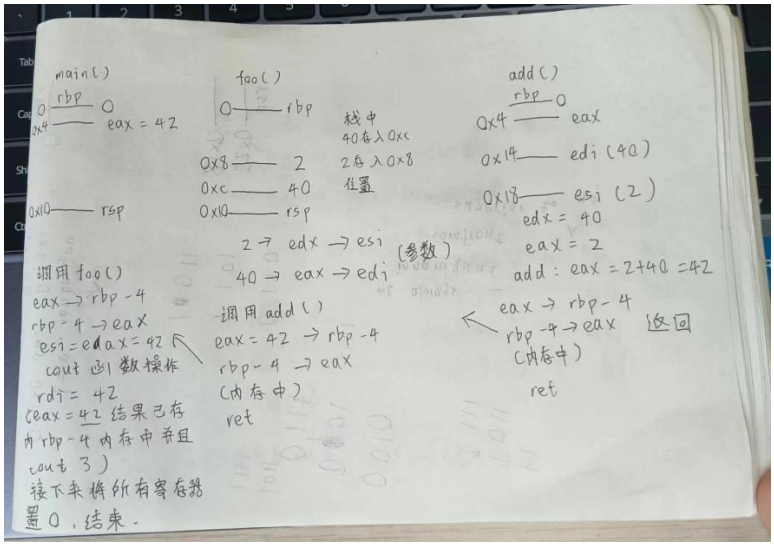
Part0：
任务 1：在 linux 中执行：g++ -g -o func func1.cpp （func1.cpp 是 cpp 文件）之后 gdb func Disas 命令查看三个函数，截图如下：

```
Dump of assembler code for function main():
   0x00000000000011f8 <+0>:     endbr64
   0x00000000000011fc <+4>:     push   %rbp
   0x00000000000011fd <+5>:     mov    %rsp,%rbp
   0x0000000000001200 <+8>:     sub    $0x10,%rsp
   0x0000000000001204 <+12>:    call   0x11c7 <_Z3foov>
   0x0000000000001209 <+17>:    mov    %eax,-0x4(%rbp)
   0x000000000000120c <+20>:    mov    -0x4(%rbp),%eax
   0x000000000000120f <+23>:    mov    %eax,%esi
   0x0000000000001211 <+25>:    lea    0x2e28(%rip),%rax        # 0x4040
<_ZSt4cout@GLIBCXX_3.4>
   0x0000000000001218 <+32>:    mov    %rax,%rdi
   0x000000000000121b <+35>:    call   0x10b0 <_ZNSolsEi@plt>
   0x0000000000001220 <+40>:    mov    0x2da9(%rip),%rdx        # 0x3fd0
   0x0000000000001227 <+47>:    mov    %rdx,%rsi
   0x000000000000122a <+50>:    mov    %rax,%rdi
   0x000000000000122d <+53>:    call   0x1090 <_ZNSolsEPFRSoS_E@plt>
   0x0000000000001232 <+58>:    mov    $0x0,%eax
   0x0000000000001237 <+63>:    leave
   0x0000000000001238 <+64>:    ret
End of assembler dump.
```

```
(gdb) disas foo
Dump of assembler code for function _Z3foov:
   0x00000000000011c7 <+0>:     endbr64
   0x00000000000011cb <+4>:     push   %rbp
   0x00000000000011cc <+5>:     mov    %rsp,%rbp
   0x00000000000011cf <+8>:     sub    $0x10,%rsp
   0x00000000000011d3 <+12>:    movl   $0x28,-0xc(%rbp)
   0x00000000000011da <+19>:    movl   $0x2,-0x8(%rbp)
   0x00000000000011e1 <+26>:    mov    -0x8(%rbp),%edx
   0x00000000000011e4 <+29>:    mov    -0xc(%rbp),%eax
   0x00000000000011e7 <+32>:    mov    %edx,%esi
   0x00000000000011e9 <+34>:    mov    %eax,%edi
   0x00000000000011eb <+36>:    call   0x11a9 <_Z3addii>
   0x00000000000011f0 <+41>:    mov    %eax,-0x4(%rbp)
   0x00000000000011f3 <+44>:    mov    -0x4(%rbp),%eax
   0x00000000000011f6 <+47>:    leave
   0x00000000000011f7 <+48>:    ret
End of assembler dump.
```

```
(gdb) disas add
Dump of assembler code for function _Z3addii:
   0x00000000000011a9 <+0>:     endbr64
   0x00000000000011ad <+4>:     push   %rbp
   0x00000000000011ae <+5>:     mov    %rsp,%rbp
   0x00000000000011b1 <+8>:     mov    %edi,-0x14(%rbp)
   0x00000000000011b4 <+11>:    mov    %esi,-0x18(%rbp)
   0x00000000000011b7 <+14>:    mov    -0x14(%rbp),%edx
   0x00000000000011ba <+17>:    mov    -0x18(%rbp),%eax
   0x00000000000011bd <+20>:    add    %edx,%eax
   0x00000000000011bf <+22>:    mov    %eax,-0x4(%rbp)
   0x00000000000011c2 <+25>:    mov    -0x4(%rbp),%eax
   0x00000000000011c5 <+28>:    pop    %rbp
   0x00000000000011c6 <+29>:    ret
End of assembler dump.
```

画栈分析如下：



Quest：“如果，我们希望一个函数能够在运行过程中暂停，然后再恢复运行（这就是我们本次实验中将要实现的协程），此时栈并不能满足我们的需求，这是为什么？”

Ans：在协程中需要能够保存函数的执行状态，包括局部变量和执行位置，以便恢复后能够继续执行。栈不提供这种能力，因为一旦函数执行完毕从栈中弹出，其状态就丢失了。

任务 2：

Q1：一个“普通”的函数支持哪两个操作，分别承担了什么功能？

A1：调用函数和返回值（**Call** and **Return**）。Call 创造一个激活帧（activation stack）（接下来用英文写……翻成 cn 太麻烦了）

Call: 1. creates an activation frame. 2. suspends execution of the calling function. 3. transfers execution to the start of the function being called.

Return: 1. passes the return-value to the caller. 2. destroys the activation frame. 3. resumes execution of the caller.

Q2：为什么我们说调用栈不能满足协程的要求？

A2：

1. 因为栈是静态分配的，而协程的实现需要更加灵活的控制流程。每次栈底 rbp 和栈顶 rsp 压入后，rsp 都下移一个**具体固定的**大小，给栈静态分配了一块内存；但协程需要在运行时动态的保存各个参数变量的状态，这点栈无法满足。
2. 栈对于每个函数有一个明确的入口点和出口点，但协程需要做到在函数的任意位置上暂停和恢复，这点栈也做不到。
3. 一旦函数返回时，函数栈中存储的变量都会被销毁，但协程需要保持状态的连续性，即使在函数返回后也要保留状态。
4. 栈的大小有限，可能无法存储协程所需的大量的状态信息。

Q3：协程作为一种泛化的函数，支持了哪几个操作，分别承担了什么功能？

A3：

1. Call: is as same as normal functions.
2. Suspend: allows the coroutine to suspend execution in the middle of the function and transfer execution back to the caller or resumer of the coroutine.
3. Resume: can be performed on a coroutine that is currently in the

'suspended' state. It is used when a function want to resume a coroutine.

4. Destroy: it destroys the coroutine frame without resuming execution of the coroutine.
5. Return: it stores the return-value somewhere, then destructs any in-scope local variables .

Q4：如果不能使用栈来实现协程，那么我们可以将函数运行时所需的信息存储在哪里？
A4：Coroutine frame, generally have to allocate memory in heap to store value.

任务 3：
运行状态和空闲状态。通常需要保存如下状态：
1.寄存器状态：存储了函数执行过程中的临时数据和指令执行的位置。
2.栈：栈用于存储函数调用的参数、局部变量以及函数的返回地址。在函数暂停时，需要保存栈的状态，使在恢复执行时能够正确返回到函数的上下文。
3.上下文：这包括进程的当前状态。这些信息可能会在函数执行中发生改变，需要在 suspend 时保存。
4.程序计数器：保存当前执行的指令的地址，确保 resume 时能从正确位置继续执行。

Part1：
1. 在 coro.h 中，补全 coroutine 结构体的实现。

```
coroutine* create(func_t func, void* args);
void release(coroutine* co);
int resume(coroutine* co, int param = 0);
int yield(int ret = 0);

const size_t STACK_SIZE = 1024 * 1024;  //协程栈的大小
enum CoroStatus { CORO_READY = 1, CORO_RUNNING, CORO_END }; //记录协程当前的状态
```

首先新开一个常量表示协程栈的大小，新开一个枚举类型变量记录一个协程的状态。
Create 等四个函数在 cpp 中实现。

```
struct coroutine {
    bool started = false;  //是否已经开始执行
    bool end = false;  //是否已经结束执行

    func_t coro_func = nullptr;  //协程函数
    void* args = nullptr;  //函数的参数

    int paraData = 0;

    ucontext_t ctx = { 0 };
    char coroStack[STACK_SIZE];
    int coroStatus;

    coroutine(func_t func, void* args) : coro_func(func), args(args) {}
    ~coroutine() {}
};
```

Coroutine 的结构体基本由模板给出了，唯一新开一个 data 字段记录后面 resume 和 yield 之间的通信（resume 需要传入这个参数，yield 需要返回这个参数。（实验文档的 hint））

Env 结构体用于存储协程的调用信息，“在目前的实验中，不会涉及到协程的递归调用，所以栈的深度可以只设置为 2。” 所以开栈深度为 2 并给个指示记录当前存储的第几个协

程。 Main_coro 用于记录 main 函数的上下文。 后面协程压栈出栈内容比较简单。

```
class coroutine_env {
private:
    coroutine* coroList[2];
    size_t coroNum = 0;

public:
    ucontext_t main_coro;  //用于记录main函数的上下文
    int paraData = 0;  //resume的第二个参数 //yield的返回值

    coroutine_env() {}

    coroutine* get_coro(int idx) {
        // TODO: implement your code here
        assert(idx >= 0 && idx <= 1);
        return coroList[idx];
    }
```

2. Coro.cpp 的实现

2. ucontext

linux提供了ucontext库用于实现用户态线程，ucontext的意思为user context。ucontext库定义的数据结构与声明的函数在ucontext.h头文件中。

ucontext库使用结构体ucontext_t表示用户上下文，ucontext_t的定义如下：

```
typedef struct ucontext
{
 unsigned long int uc_flags;
 struct ucontext *uc_link;
 stack_t uc_stack;
 mcontext_t uc_mcontext;
 __sigset_t uc_sigmask;
} ucontext_t;
```

我们借助 ucontext 库的指针 link 来始终关联着主协程（create 的时候暂时置为 nullptr 也行，后面 resume 时候会关联）

```
coroutine* create(func_t func, void* args) {
    coroutine* new_coro = new coroutine(func, args);
    new_coro->ctx.uc_stack.ss_sp = new_coro->coroStack;
    new_coro->ctx.uc_stack.ss_size = STACK_SIZE;
    new_coro->ctx.uc_link = &g_coro_env.main_coro;
    new_coro->coroStatus = CORO_READY;
    return new_coro;
}

void release(coroutine* co) {
    if (co) delete co;
    return;
```

Yield

```
int yield(int ret) {
    coroutine* co = g_coro_env.pop();
    co->paraData = ret;
    if (ret != -1) {
        swapcontext(&co->ctx, &g_coro_env.main_coro);  //保存
        return co->paraData; //暂停
    }
    else return -1; //协程结束

}
```

之前，协程每次开始运行了就将协程信息压入 list 内存储，所以 list 的顶部就是当前正在运行的协程，yield 就是需要将这个正在运行的协程暂停同时激活主协程。

很自然的想到 swap 函数，将当前上下文记录，执行另一个上下文，规定 yield（-1）就是协程直接结束。

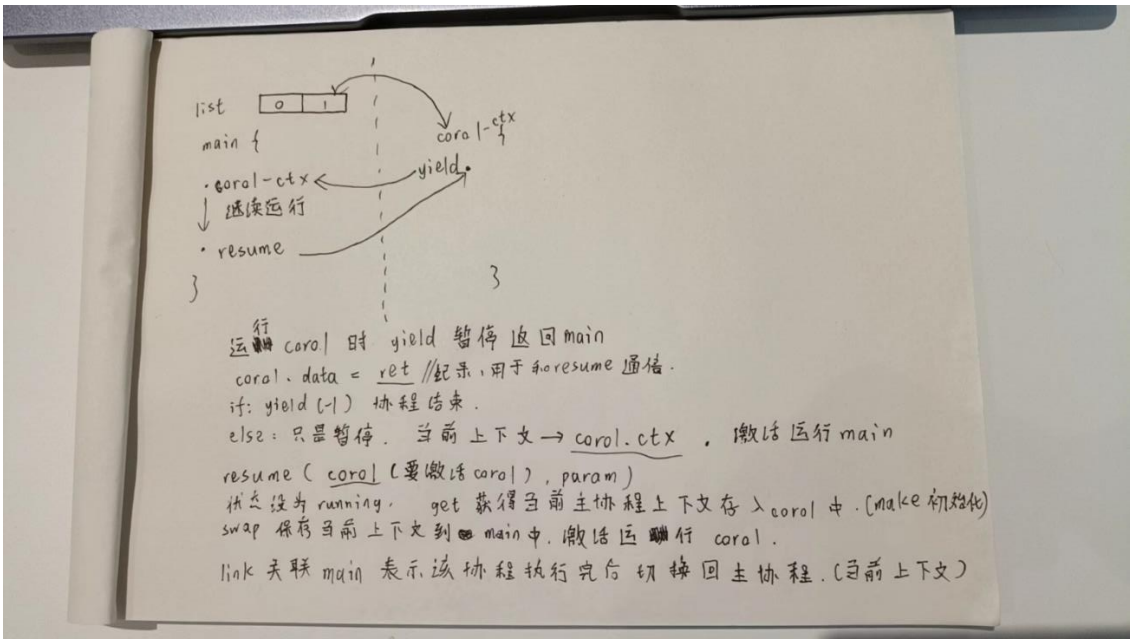
Resume（最麻烦）

首先我们全局的 env 的 list 是记录着当前正在运行的协程，所以要恢复 co，也就将 co 先 push 进这个 list。若这个 co 正在运行，直接切换上下文就好。若它只是 ready 状态（刚刚创建或者是被暂停），就要做一些初始化：get 把当前上下文写入 co 的上下文中，co 链接当前的 main 主协程（表示 co 这个协程执行完后就回到这里）状态设置为运行中，makecontext（这里确实有点蒙，参考了群里助教给的代码，我的理解应该就是把 co 初始化了），最后切换上下文　　画了一个简单的切换过程在代码后面——

```
int resume(coroutine* co, int param) {
    g_coro_env.push(co);  //要恢复开始运行，压入list
    co->paraData = param;  //yield时返回给调用者的参数

    switch (co->coroStatus) {
    case CORO_READY:
        getcontext(&co->ctx);
        co->ctx.uc_link = &g_coro_env.main_coro;  //ctx关联主协程
        co->coroStatus = CORO_RUNNING;
        makecontext(&co->ctx, (void (*)(void)) func_wrap, 1, co);  //初始化
        swapcontext(&g_coro_env.main_coro, &co->ctx);  //保存当前上下文到main中，然后激活ctx上下文
        break;
    case CORO_RUNNING:
        swapcontext(&g_coro_env.main_coro, &co->ctx);
        break;
    }

    return co->paraData;
}
```



我感觉只有 started 和 ended 总感觉说不清，于是自己定义的 ready、running、end，这个比较清晰一点，ready 和 running 应该是得分开啊（），但是这个确实有些冗余了。。。不过不会产生错误（）。

```
makekurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v1$ make
g++-11 -std=c++20 -o main main.cpp coro.cpp
.kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v1$ ./main
test-1 passed
test-2 passed
test-3 passed
Congratulations! You have passed all the tests of libco-v1!
```

Part2，part2.5:

写在最后，当时没看到腾讯的那个 libco 也没看到助教透露的代码实在做不下去了……先往后做了。

Part3:
这一 part 比较简单，基本看懂实验文档就能做了

```
#define CO_BEGIN  case 0:

#define CO_END  return 0;

#define CO_YIELD(...)  line = __LINE__; return __VA_ARGS__; case __LINE__:

#define CO_RETURN(...)  return __VA_ARGS__;
```

首先定义这四个宏（其实好像没啥必要（），在后面斐波那契那里写一样的）
而且我的感觉是只有 yield 这个是有用的，return 的功能已经完全被 yield 取代了啊（yield 记录当前行并且返回一个值（可变参数__VA_ARGS__，相当于函数参数的 . . . ）事实也确实发现，在 fib 里加不加 return 都可以过。然后 end 主要是在 fib 最后保证有个返回值编译时不会产生 warning。

```
struct coroutine_base {
    int line = 0;   //记录上一次推出时的位置
};

class fib : public coroutine_base {
private:
    int a = 0;
    int b = 1;

public:
    int operator()()
    {
        switch (line) {
            CO_BEGIN
                while (true)
                {
                    CO_YIELD(a)
                    int temp = a;
                    a = b;
                    b = temp + b;  //fibonacci
                }
            CO_RETURN(-1)   //其实不需要，到不了这里
        }
        CO_END
    }
```

公共变量 line 记录当前行，以便在下次调用时 switch 直接走到上次的行的地方继续执行，本质也就完成了 yield 和 resume（确实 6），例如，第一次 yield 0 后记录当前行后直接返回 0，第二次调用这个函数的时候，就直接是 case __LINE__（上一次记录的 line）那里继续执行，a=1，b=1+0=1 更新，while 循环 yield1……如此直到结束。
有一点可以改进的是 coroutine_base 里面或许可以加个 bool 变量记录操作是否全部完成，如果全部完成，此后再调用时相当于无效调用，直接返回。但这个测例不需要。

```
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v3$ make
g++-11 -o main main.cpp
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v3$ ./main
libco_v3 test passed!
```

Part4:

参考 C++ coroutine generator 实现笔记 - 知乎 (zhihu.com)学习

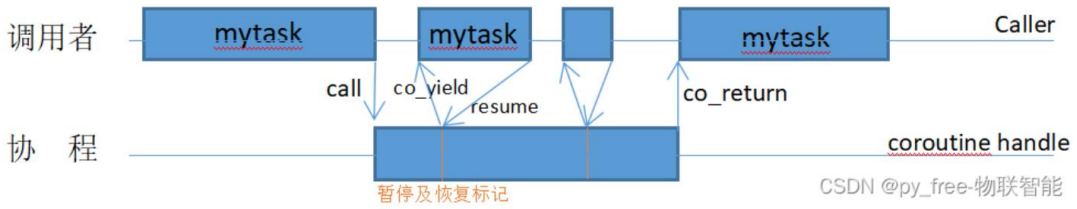
参考 C++语法糖(co await)详解以及示例代码 - 知乎 (zhihu.com)学习

参考【并发编程二十一：终章】c++20 协程(co_yield、co_return、co_await) _郑同学的笔记的博客-CSDN 博客学习

参考 C++ Coroutines: Understanding the Compiler Transform | Asymmetric Transfer (lewissbaker.github.io)学习

参考 Coroutines in C++20 - 简书 (jianshu.com)学习

参考 C/C++开发，无可避免的多线程（篇三）.协程及其支持库_c++协程库-CSDN 博客学习



Answer：

1. 协程函数的返回值 Coroutine Functor 需要有哪些成员？

Promise_type

> Now, as we have not specialised `std::coroutine_traits` this will instantiate the primary template which just defines the nested `promise_type` as an alias of the nested `promise_type` name of the return-type. i.e. this should resolve to the type `task::promise_type` in our case.

2. Promise 对象需要提供哪些函数？

return_void()　　return_value(<expr>)　　final_suspend()
get_return_object()　　unhandled_exception()　　initial_suspend()
yield_value(<expr>)

> Here is a summary of the steps (I'll go into more detail on each of the steps below).
>
> 1. Allocate a coroutine frame using `operator new` (optional).
> 2. Copy any function parameters to the coroutine frame.
> 3. Call the constructor for the promise object of type, `P`.
> 4. Call the `promise.get_return_object()` method to obtain the result to return to the caller when the coroutine first suspends. Save the result as a local variable.
> 5. Call the `promise.initial_suspend()` method and `co_await` the result.
> 6. When the `co_await promise.initial_suspend()` expression resumes (either immediately or asynchronously), then the coroutine starts executing the coroutine body statements that you wrote.
>
> Some additional steps are executed when execution reaches a `co_return` statement:
>
> 1. Call `promise.return_void()` or `promise.return_value(<expr>)`
> 2. Destroy all variables with automatic storage duration in reverse order they were created.
> 3. Call `promise.final_suspend()` and `co_await` the result.
>
> If instead, execution leaves `<body-statements>` due to an unhandled exception then:
>
> 1. Catch the exception and call `promise.unhandled_exception()` from within the catch-block.
> 2. Call `promise.final_suspend()` and `co_await` the result.

3. Awaitable object 需要提供哪些接口？

“A type that supports the co_await operator is called an Awaitable type.”

需要提供 await_ready　await_suspend　await_resume

To be more specific where required I like to use the term **Normally Awaitable** to describe a type that supports the `co_await` operator in a coroutine context whose promise type does not have an `await_transform` member. And I like to use the term **Contextually Awaitable** to describe a type that only supports the `co_await` operator in the context of certain types of coroutines due to the presence of an `await_transform` method in the coroutine's promise type. (I'm open to better suggestions for these names here...)

An **Awaiter** type is a type that implements the three special methods that are called as part of a `co_await` expression: `await_ready`, `await_suspend` and `await_resume`.

Note that I have shamelessly "borrowed" the term 'Awaiter' here from the C# `async` keyword's mechanics that is implemented in terms of a `GetAwaiter()` method which returns an object with an

4. Coroutine handle 通常需要提供哪些函数？

Coroutine_handle:: 的 namespace 下：

Resume()　destroy()　done()

address()　from_address()

Promise()　from_promise()　//关联 generator 内部的 promise_type

We also need to implement the `coroutine_handle::destroy()` function so that it invokes the appropriate logic to destroy any in-scope objects at the current suspend-point and we need to implement `coroutine_handle::done()` to query whether the current suspend-point is a final-suspend-point.

The interface of the `coroutine_handle` methods does not know about the concrete coroutine state type - the `coroutine_handle<void>` type can point to *any* coroutine instance. This means we need to implement them in a way that type-erases the coroutine state type.

We can do this by storing function-pointers to the resume/destroy functions for that coroutine type and having `coroutine_handle::resume/destroy()` invoke those function-pointers.

The `coroutine_handle` type also needs to be able to be converted to/from a `void*` using the `coroutine_handle::address()` and `coroutine_handle::from_address()` methods.

Furthermore, the coroutine can be resumed/destroyed from *any* handle to that coroutine - not just the handle that was passed to the most recent `await_suspend()` call.

These requirements lead us to define the `coroutine_handle` type so that it only contains a pointer to the coroutine-state and that we store the resume/destroy function pointers as data-members of the coroutine state, rather than, say, storing the resume/destroy function pointers in the `coroutine_handle`.

Also, since we need the `coroutine_handle` to be able to point to an arbitrary coroutine-state object we need the layout of the function-pointer data-members to be consistent across all coroutine-state types.

5. 为什么说 co_yield 和 co_return 是 co_await 的语法糖？

因为 co_yield 实际上等价于 co_await promise.yield_value(int value)

还是以此举例，`co_yield` 关键字实际上只是一个语法糖，这一行会被编译器替换为 `co_await promise.yield_value(i)`，在有了 `initial_suspend` 和 `final_suspend` 的经验后，我们这次也就能很容易地猜测出，我们要在 `promise_type` 中实现一个 `yield_value` 方法，而返回值负责回答要不要切出的问题。显然，每次 yield 时总是要挂起协程，所以，`yield_value` 方法的返回值类型应当是 `suspend_always`。你猜对了吗?

同理，co_return 实际上等价于 co_await promise.return_value(int value)或者 promise.return_void() 且协程执行 final_suspend()（也就是执行 suspend_always（）当前协程被无条件切出去）。

6. 简述协程函数的调用过程并阐述上述每个接口函数的功能。

调用过程：首先 new 申请空间构造一个协程框架，为这个协程框架复制任意函数参数（initialize），其次，构造一个 promise_type 对象，当协程第一次被挂起的时候通过 get_return_object 获取关联的返回对象，将结果保存为一个本地变量（协程在

initial_suspend 处挂起）。在协程体内，可以使用 co_await 来挂起协程（如果使用 co_yield，协程会挂起并返回一个值）。当这个挂起被 resume 时候，这个协程就会开始执行我们写的 body 内容。　　如果协程体执行完毕或遇到 co_return 的时候（no matter return_value(<expr>) or return_void（)），然后在 final_suspend 处挂起，同时按照变量被创造相反的顺序 destory 所有的变量，co_await 最终结果。　　如果在协程中抛出异常，调用 promise_type 的 unhandled_exception 方法。协程的调用 coroutine_handle 的 destroy 方法销毁。　　Destroy 协程框架：首先调用 promise 的 destory()对象，其次销毁函数参数的复制，释放框架内存，将执行顺序流转移回 caller 继续。

Here is a summary of the steps (I'll go into more detail on each of the steps below).

1. Allocate a coroutine frame using `operator new` (optional).
2. Copy any function parameters to the coroutine frame.
3. Call the constructor for the promise object of type, `P`.
4. Call the `promise.get_return_object()` method to obtain the result to return to the caller when the coroutine first suspends. Save the result as a local variable.
5. Call the `promise.initial_suspend()` method and `co_await` the result.
6. When the `co_await promise.initial_suspend()` expression resumes (either immediately or asynchronously), then the coroutine starts executing the coroutine body statements that you wrote.

Some additional steps are executed when execution reaches a `co_return` statement:

1. Call `promise.return_void()` or `promise.return_value(<expr>)`
2. Destroy all variables with automatic storage duration in reverse order they were created.
3. Call `promise.final_suspend()` and `co_await` the result.

If instead, execution leaves `<body-statements>` due to an unhandled exception then:

1. Catch the exception and call `promise.unhandled_exception()` from within the catch-block.
2. Call `promise.final_suspend()` and `co_await` the result.

Once execution propagates outside of the coroutine body then the coroutine frame is destroyed. Destroying the coroutine frame involves a number of steps:

1. Call the destructor of the promise object.
2. Call the destructors of the function parameter copies.
3. Call `operator delete` to free the memory used by the coroutine frame (optional)
4. Transfer execution back to the caller/resumer.

Initial_suspend()和 final_suspend() 回答的是协程一出生时、协程最后一次时是否要挂起的问题。对于一个 generator 而言，这两个问题的回答是：初始时和最后一次始终都要挂起。因此，都直接使用 suspend_always（将他们无条件挂起）

The promise() method is straight-forward -- it just returns a reference to the __promise member of the coroutine-state.

The from_promise() method requires us to calculate the address of the coroutine-state from the address of the promise object. We can do this by just subtracting the offset of the __promise member from the address of the promise object.

unhandled_exception()它回答的是协程被其里头的没有捕获的异常终止时做何处理的问题。 简单处理，调用 exit 或 terminate 提前终止程序。

get_return_object() 核心函数。可以直接调用 from_promise()实现

4) `promise_type` 中第四个定制接口，也是最核心的一个是 `get_return_object` 。这个方法也涉及到了如何创建一个 coroutine 的问题 —— 答案就是使用 `std::coroutine_handle<promise_type>::from_promise(*this)` ，即从自己这个 promise (也就是 `*this`) 创建一个 coroutine (`from_promise` 得到的就是一个 `coroutine_handle`)。`generator` 中也需要配合，提供一个接受 `coroutine_handle` 类型的构造函数，将刚刚构造出的 `coroutine_handle` 保存。

Yield_value(value) 挂起协程（直观来看直接 return 就可以实现）并且返回一个值。
Return_value(value) 在 final_suspend 处挂起，并且返回一个值（或者，当协程执行完毕后也会自动在 final_suspend 处挂起）。
Return_void() 协程在 final suspend 处挂起，但是不传回值。
等待器：await_ready() 方法：如果返回 true 则当前协程不暂停；否则协程暂停，返回到父协程，调用 await_suspend 方法。
await_suspend() 方法传入调用 co_await 的协程的 coroutine_handle，可返回以下三种类型：
void：无其他操作，父协程继续运行。
bool：true 则父协程继续运行，false 则从父协程恢复到调用 co_await 的协程。
std::coroutine_handle：从父协程恢复到 handle 对应的协程，本质上是一个指向协程框架的美化指针。
如果确定协程不暂停，或者经过暂停之后被恢复，则调用 await_resume 方法。
await_resume() 方法的返回值也成为 co_await 操作符的返回值。
Coroutine_handle 可以用于链接到协程的 promise 对象
Done() 查询一个协程是否已经结束。
Resume() 恢复一个协程的执行。
Destroy() 销毁一个协程。
from_promise [静态]从协程的承诺对象创建 coroutine_handle(公开静态成员函数)
from_address [静态]从指针导入协程(公开静态成员函数)

根据以上理解，可以先补充完 promise_type 的内容：
promise_type 就是承诺对象，承诺对象用于协程内外交流

```
struct promise_type{
    Value value;
    // 构造成功后直接挂起
    std::suspend_always initial_suspend() const
    {
        return {};
    }
    // 协程要结束时也直接挂起
    std::suspend_always final_suspend() const noexcept
    {
        return {};
    }
    generator get_return_object()
    {
        return generator{ std::coroutine_handle<promise_type>::from_promise(*this) };
    }
    // 出现未经处理的异常时执行
    void unhandled_exception() { std::terminate(); }
    // co_return时执行
    void return_void() {}
    // 功能好像全被yield干完了，用不到这个
    //void return_value(const Value& val) noexcept {value = val; return;}
    // co_yield时执行
    std::suspend_always yield_value(const Value& val) noexcept
    {
        value = val;
        return {};
    }
```

get_return_object，可以生成一个协程返回（借助 from_promise），会在协程正在运行前进行调用。
补充 iterator 中的内容：
首先=赋值语句：直接使用模板，使用 swap 交换这两个迭代器的协程，同时将原来的协程置空（补充了这一句，逻辑感觉好些，不要也行），返回当前迭代器。

```
iterator& operator=(iterator&& o) {
    std::swap(coro_, o.coro_);
    o.coro_ = {};
    return *this;
}
```

==和！=。这里==和！=都是将当前迭代器和 sentinel 进行比较，sentinel 是边界哨兵，代表结束，也就是说，是为了判断当前迭代器是否终止。一种情况是要比较的迭代器根本不存在（null），另一种情况是该迭代器存在的，但是已经运行结束（使用 done（））。
而不等（未终止）也就是==的取反。

```
friend bool operator==(const iterator& o, sentinel)
    noexcept {
    return !o.coro_ || o.coro_.done();
}
friend bool operator!=(const iterator& o, sentinel)
    noexcept {
    return o.coro_ && !o.coro_.done();
}
```

++和++（int）我的理解是++it 和 it++的区别。功能上是一样的（），除非写出 it1 = it++和 it1 = ++it 这种会有区别，但是这里好像不用区分，只要简单实现功能就能过。
需要注意的是，迭代器每次运行 yield 回一个值的时候，他都会被无条件挂起，所以迭代器++指向下一个值的时候要先将他恢复（使用 resume（））才能继续运行。

```
iterator& operator++() {
    coro_.resume();
    return *this;
}
//++it和it++的区别，功能上相同
iterator& operator++(int) {
    coro_.resume();
    return *this;
}
```

解引用操作，也就是对迭代器当前取值，使用 promise()与协程内部交流，取出当前的 value。
指向操作，利用 addressof 取地址即可。（获取一个对象的地址，与&功能相同）

```
reference operator*() const noexcept {
    return coro_.promise().value;
}

pointer operator->() const noexcept {
    return std::addressof(operator*());
}
```

接下来迭代器的 constructor 和 generator 的 constructor 类似，功能上只是创建一个，语法上很抽象参考的网上。初始化的协程置空。

```
private:
    friend generator;
    // TODO: implement iterator constructor
    // hint: maybe you need to a promise handle
    explicit iterator(std::coroutine_handle<promise_type> coro) noexcept : coro_(coro) {}
    std::coroutine_handle<promise_type> coro_ = nullptr;
    // TODO: add member variables you need
};
```

Begin（）和 end（）的语法参考网上代码。功能上理解比较简单。

```
iterator begin() noexcept {
    if (coro_) coro_.resume();  //激活
    return iterator{ coro_ };
}

sentinel end() noexcept {
    return {};
}
```

Main 函数是构造了一个斐波那契数列，首先将每个元素存入创建的生成器 gen，之后忽略前三个（drop(3)）取出接下来的十个进行输出（take(10)）。

存入迭代器的过程集中体现了 generator 类的工作原理：我的理解分析如下——

1. 初始 a=0，b=1（initialize）
2. 每次 while 循环即进行 co_yield（也就是 co_await 的 yield_value），也就是返回当前 a 的值并且将协程无条件挂起。
3. a，b = b，a+b　下一次循环。
4. 进入下一次循环时，它工作机制我感觉首先执行的是 iterator++（迭代器挪到下一个位置准备写入值），++操作首先将上一次循环挂起的协程进行恢复（resume）然后才继续执行，接下来返回第 2 步。

```
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v4$ make
g++-11 -std=c++20 -o main main.cpp
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v4$ ./main
Hello, ICS 2023!
2 3 5 8 13 21 34 55 89 144
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v4$
```

以及，它在第一步写入生成器的时候，iterator 应该是存了容量上限个数字的？

```
g++-11 -std=c++20 -o main main.cpp
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v4$ ./main
Hello, ICS 2023!
2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229
 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494
437 701408733 1134903170 1836311903 -1323752223 512559680 -811192543 -298632863 -1109825406 -1408458269 1776683621 36822
5352 2144908973 -1781832971 363076002 -1418756969 -1055680967 1820529360 764848393 -1709589543 -944741150 1640636603 695
895453 -1958435240 -1262539787 1073992269 -188547518 885444751 696897233 1582341984 -2015728079 -433386095 1845853122 14
12467027 -1036647147 375819880 -660827267 -285007387 -945834654 -1230842041 2118290601 887448560 -1289228135 -401779575
-1691007710 -2092787285 511172301 -1581614984 -1070442683 1642909629 572466946 -2079590721 -1507123775 708252800 -798870
975 -90618175 -889489150 -980107325 -1869596475 1445263496
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v4$
```

```
for (const auto& n : gen | std::views::drop(3) | std::views::take(100)) {
    std::cout << n << ' ';
}
std::cout << '\n';
```

Part5：
参考 C++20 协程学习 - 知乎 (zhihu.com)学习
参考 https://zhuanlan.zhihu.com/p/349710180 学习
参考 C/C++开发，无可避免的多线程（篇三）.协程及其支持库_c++协程库-CSDN 博客学习
参考 https://sf-zhou.github.io/coroutine/cpp_20_coroutines.html 学习
参考 https://www.bennyhuo.com/2022/03/19/cpp-coroutines-04-task/学习
可递归相比于原本的 part4 多出了维护一个调用树。
先将 part4 的 generator 代码全部 co 过来

```
    promise_type* node = nullptr;
    promise_type* pre = nullptr;
    Value value;

    promise_type() : node(this) {}  //构造函数
```

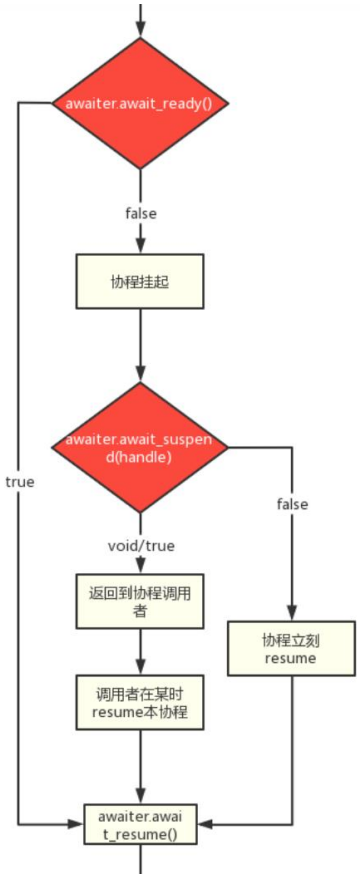
根据实验文档提示，我们要手搓一个递归调用树，其实总体逻辑并不复杂，在每个 promise_type 对象里设置一个 node（当前递归调用的结点）指向它自己，设置一个 pre（指向上一层递归调用的结点）指向他的父协程。
初始化了成员变量 node 将其地址置为当前对象 this，实现每当创建一个 promise_type 类型的对象的时候，它的 node 都指向它自己。

```
std::suspend_always yield_value(const Value& val) noexcept {
    node->value = val;
    return {};
}
```

其余函数照搬 part4 即可，yield 中注意要将赋值对象改成 node 所指的 value（当前（this）对象中的 value）。
接下来要添加其他结构实现功能：
一般来说，一个协程被调用 yield 后返回主协程，但是要实现递归的任务，也就是一个协程运行到一半 yield 后跑去递归执行另一个协程，之后还得跑回来继续执行这个协程，这也就是递归的思想。从头一个一个跑到尾，再按相反顺序返回。（所以只需要 node 和 pre 记录当前和上一个）　　我们需要两个结构分别实现协程的向下递归和向上递归运行即可。
第一个结构体是做协程返回要做的操作：

```
struct end_awaiter {
    constexpr bool await_ready() noexcept { return false; }
    Handle await_suspend(Handle h) noexcept {
        auto prev = h.promise().pre;
        if (prev) return Handle::from_promise(*prev);
        else return std::noop_coroutine();
    }
    void await_resume() noexcept {}
};
```

（这些代码实现上均有参考列举的第二篇、第五篇文章）
执行流程图如下：
首先 await_ready() 无条件返回 false，在该情况下协程直接挂起（在当前协程（以协程柄类型的 h 表示）执行即将结束时调用 end_awaiter 的 suspend），通过 promise 与协程内部交流，取出 pre（上一层递归）。如果上一层递归存在，也就是有父协程，那么返回父协程柄以恢复父协程。否则返回 noop_coroutine()，表示恢复不做任何事（递归到根协程，结束）。根据以下流程图逻辑，最后要加一个 await_resume()等待调用者在某时刻恢复本协程。



await_ready() 告诉 co_await 自己好了没.

await_suspend(h) 可以选择返回 void , bool , std::coroutine_handle<P> 之一. h 是本协程的 handle. P 是本协程的 promise 类型 (或者是 void , 见第三篇中的解释).

如果 await_ready() 返回 false , 这个协程就会暂停. 之后:

1. 如果 await_suspend(h) 返回类型是 std::coroutine_handle<Z> , 那么就会恢复这个 handle. 即运行 await_suspend(h).resume() . 这意味着暂停本协程的时候, 可以恢复另一个协程.
2. 如果 await_suspend(h) 返回类型是 bool , 那么看 await_suspend(h) 的结果, 是 false 就恢复自己.
3. 如果 await_suspend(h) 返回类型是 void , 那么就直接执行. 执行完暂停本协程.

如果 await_ready() 返回 true 或者协程被恢复了, 那么就执行 await_resume() , 它得到的结果就是最终结果.

所以说, 这 await_ready, await_suspend, await_resume 三个接口分别表示 "有没有准备好", "停不停", "好了该咋办". 设计还是很自然的.

接下来是另一个结构体做协程向下递归运行要做的操作：
（参考了文章 5 的部分代码实现）
构造函数使用 std 的移动语义，相当于复制构造。
与上同理，先无条件挂起协程，执行要做的操作。取出当前协程和嵌套协程（给定的参数 generator）的引用。之后将当前协程的 node 和 pre 信息传递给嵌套协程的 promise 对象，然后更新当前协程 promise 对象的 node 指向嵌套协程的 promise 对象（取地址）。最后，返回嵌套协程的句柄。Resume 与上流程图和上一个结构体同理，在协程被恢复时调用。
具体说来：这里将嵌套协程和当前协程关联起来了，嵌套协程是当前协程的子协程（这里是从上往下走，上一个 end 是从下往上走）

```
struct yield_awaiter {
    generator gen;
    constexpr bool await_ready() noexcept { return false; }
    Handle await_suspend(Handle h) noexcept {
        auto& cur = h.promise();
        auto& nested = gen.coro_.promise();

        nested.node = cur.node;
        nested.pre = cur.pre;
        cur.node->node = std::addressof(nested);

        return gen.coro_;
    }
    void await_resume() noexcept {}
};
```

构造析构：特别的，移动构造，防止开新资源造成混乱：

```
generator(generator&& completion) noexcept
    : coro_(std::exchange(completion.coro_, {})) {}
```

（语法参考文章 5 实现，功能理解简单）

Iterator 把 4 的代码 co 过来即可，注意实现递归功能，重载的++需要变成当前 coro 的 promise 对象再++（resume）

```
iterator& operator++() {
    coro_.promise().resume();
    return *this;
}
iterator operator++(int) {
    coro_.promise().resume();
    return *this;
}
```

Sleep.h:

（参考了部分周四透露的代码（））

功能上比较好理解：主要目的是允许对 task 进行执行调度，我使用一种叫 std::this_thread::sleep_for 引入延迟的机制（阻塞当前线程执行，至少经过指定的 sleep_duration）。std::this_thread::sleep_for-CSDN 博客

关于理解仿函数实现 lambda 功能：std::function 类模板_std::function 模板函数_#A#的博客-CSDN 博客

Sleep 结构体代码如下：

```cpp
struct sleep {
    sleep(int n_ms) : delay{ n_ms } {}
    std::chrono::milliseconds delay;

    constexpr bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) const noexcept {
        task_queue.pop();
        std::this_thread::sleep_for(delay);
        task_queue.push([h]() -> bool {
            h.resume();
            return h.done();
            });
    }
    void await_resume() const noexcept {}
};
```

功能上是实现延时。
首先需要理解 task_queue 的功能，这应该是存储着等待工作的协程（task 名字（）），然后工作后从队列中扔出去。
ready suspend resume 思路与 generator 一致。关键是实现 await_suspend 的内容。每当挂起一个协程，可以理解为这个协程暂时工作完成，于是我们将其（当前）pop() 掉，之后实现延迟效果，延迟前面定义的 delay 时间，再将参数给定的协程激活后放入 queue 末尾（按顺序等待执行），函数体 bool 值标记为判断当前协程是否已经完成（这点在最后 wait_task_queue_empty()用到）

```cpp
void wait_task_queue_empty() {
    while (!task_queue.empty()) {
        auto task = task_queue.front();
        if (task()) task_queue.pop();
    }
}
```

这部分有参考助教给的代码。功能上比较简单，将队列置空，即完成所有 pending 的工作（我最开始担心这个会有死循环的风险，要是任意一个没有执行完，但好像没这问题，main 函数里就是两个 task 交替执行输出，最后也确实都输完了，那按理说应该就是执行完毕，不存在 not done 的情况，事实也确实能跑（））

Attach：
Gpt 帮我写了一份避免 infinite loop 的代码，逻辑更加清晰，也能实现功能：

```cpp
void wait_task_queue_empty() {
    constexpr int max_iterations = 1000; // Set a maximum number of iterations
    int iterations = 0;

    while (!task_queue.empty() && iterations < max_iterations) {
        auto task = task_queue.front();
        if (task()) task_queue.pop();

        ++iterations;
    }

    // Optionally, you can add a timeout mechanism using a clock
    auto start_time = std::chrono::high_resolution_clock::now();
    auto timeout_duration = std::chrono::seconds(5); // Adjust timeout as needed

    while (!task_queue.empty() &&
        std::chrono::high_resolution_clock::now() - start_time < timeout_duration) {
        auto task = task_queue.front();
        if (task()) task_queue.pop();
    }
}
```

最后是 Task 类，promise_type 照搬 generator，注意返回值的时候由于规则要 suspend_always，而没 suspend 我们就得扔掉一个当前执行的 task（其实是暂挂），然后将给定的 task 加入队尾。

```cpp
std::suspend_always yield_value(std::function<bool()> func) {
    task_queue.pop();
    if(func) task_queue.push(func);
    return {};
}
```

构造函数逻辑同理：

```cpp
Task(std::coroutine_handle<promise_type> h) : coro(h) {
    task_queue.push([this]() -> bool {
        coro.resume();
        return coro.done();
        });
}
~Task() {
    if (coro) coro.destroy();
}
```

测试：

```
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v5$ make
g++-11 -std=c++20 -o main main.cpp
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v5$ ./main
Start libco_v5 test
libco_v5 task test passed!
```

Part2 & part2.5:

参考 libco 源码解析(1) 协程运行与基本结构-CSDN 博客学习

参考 libco 源码解析(4) 协程切换，coctx make 与 coctx swap-CSDN 博客学习

参考文章 https://github.com/Tencent/libco/blob/master 学习

Ctx 部分：

1. Coro_ctx_swap.S:

（汇编和 github 中 x86 体系一样）

```
.globl coro_ctx_swap
coro_ctx_swap:
    leaq (%rsp),%rax    # rsp -> rax
    movq %rax, 104(%rdi)  # rsp
    movq %rbx, 96(%rdi)
    movq %rcx, 88(%rdi)
    movq %rdx, 80(%rdi)
    movq 0(%rax), %rax
    movq %rax, 72(%rdi) # ret address
    movq %rsi, 64(%rdi)
    movq %rdi, 56(%rdi)
    movq %rbp, 48(%rdi)
    movq %r8, 40(%rdi)
    movq %r9, 32(%rdi)
    movq %r12, 24(%rdi)
    movq %r13, 16(%rdi)
    movq %r14, 8(%rdi)
    movq %r15, (%rdi)
    xorq %rax, %rax  # return 0;

    movq (%rsi), %r15
    movq 8(%rsi), %r14
    movq 16(%rsi), %r13
    movq 24(%rsi), %r12
    movq 32(%rsi), %r9
    movq 40(%rsi), %r8
    movq 48(%rsi), %rbp
    movq 56(%rsi), %rdi
    movq 64(%rsi), %rdx
    movq 88(%rsi), %rcx
    movq 96(%rsi), %rbx
    movq 104(%rsi), %rsp
    leaq 8(%rsp), %rsp # pop  rsp+=8
    pushq 72(%rsi)     # push ret add

    movq 80(%rsi), %rsi
    ret
```

Rdi 和 rsi 是函数的第一个和第二个参数，显然，这也就是执行协程上下文交换的功能。

2. Coro_ctx.cpp:

Ctx_make:

(代码语法上参考了文章 2 和 libco/coctx.cpp at master · Tencent/libco · GitHub)

```
void ctx_make(context* ctx, func_t coro_func, const void* arg) {
        char* sp = (char*)ctx->ss_sp + ctx->ss_size - sizeof(void*);
        sp = (char*)((unsigned long)sp & -16L);

        memset(ctx->regs, 0, sizeof(ctx->regs)); //initialize

        void** ret_addr = (void**)sp;
        *ret_addr = (void*)coro_func;

        ctx->regs[13] = sp;
        ctx->regs[9] = (void*)coro_func;
        ctx->regs[7] = (void*)arg;
    }
```

首先给出 sp 所指的地方，也就是 esp 指向的方向，ss_size 就是目前栈上剩余的空间，ctx->ss_sp 对应的空间是在堆上分配的，地址是从低到高的增长，而堆栈是往低地址方向增长的。

```
//------- ss_sp + ss_size
//|      |
//------- ss_sp
```

接下来需要字节对齐（GCC 默认的堆对齐设为 16 字节）

> **16L的哲学**
>
> 然后我们来说一说那个16L的魔法数字到底有什么用，我们在代码中提到了这个magic number其实是为了字节对齐。16这个数字非常奇怪，一般来说我们的认知都是32位下字节对齐应该是4，64位系统下当然就是8了，这个16是什么情况？**答案就是GCC默认的堆对齐设置的就是16字节**。具体可查看这篇文章：《Why does System V / AMD64 ABI mandate a 16 byte stack alignment?》

Ret_address 处理得到函数返回地址

Coro_func 是新协程要执行的指令函数，也即执行完这个函数要 ctx_swap 要返回的值

之后便是压栈操作，处理好地址即可。

3. Coro_ctx.h: （仅定义一些变量即可）

Coroutine.h 部分：

参考 libco/co_routine.cpp at master · Tencent/libco · GitHub 代码学习

参考微信著名 libco 协程库原理剖析-CSDN 博客学习

由于是共享栈，我们必须记录一下当前是谁在使用栈。

“共享栈模式：协程切换的时候，用来拷贝存储当前共享栈内容的 buffer，长度为实际的共享栈使用长度。通常情况下，一个协程实际占用的（从 esp 到栈底）栈空间，相比预分配的这个栈大小（比如 libco 的 128KB）会小得多；这样一来， copying stack 的实现方案所占用的内存便会少很多。当然，协程切换时拷贝内存的开销有些场景下也是很大的。因此两种方案各有利弊，默认使用前者，也允许用户在创建协程时指定使用共享栈。”

由上所述，我们的栈结构应该至少包含四个内容：

```
struct stack_mem {
    int stack_size = 0;             // 栈的大小
    char* stack_buffer = nullptr; // 栈的缓冲区
    char* stack_bp = nullptr;      // 栈的基指针
    coroutine* owner = nullptr;    // 当前拥有该栈的协程
    // TODO: add member variables you need
```

构造函数和析构函数如下：直接在堆上申请一段大小为 size 的空间作为该协程的使用栈，用 buffer 来记录，更新 bp（堆栈是往低地址方向走，buffer 比 bp 低）

```
stack_mem(size_t size) : stack_size(size) {
    // TODO: implement your code here
    stack_buffer = new char[size];
    stack_bp = stack_buffer + size;
}
~stack_mem() {
    // TODO: implement your code here
    delete[] stack_buffer;
}
```

在共享栈的结构中，参考助教的提示“注意：当 count 不为 1 时，你需要维护一种 stack_mem 的使用方式，以减少拷贝带来的开销。”
回答——“采用共享栈时，每个协程的栈从共享栈拷出时，需要分配空间存储，但按需分配空间。因为绝大部分协程的栈空间都远低于 128K，因此拷出时只需分配很小的空间，相比私有栈能节省大量内存。共享栈可以只开一个，但为了避免频繁换入换出，一般开多个共享栈。每个共享栈可以申请大空间，降低栈溢出的风险。
假设开 10 个共享栈，每个协程模 10 映射到对应的共享栈。假设协程调用顺序为主协程、协程 2、协程 3、协程 12。协程 2 切到协程 3 时，因为协程 2、3 使用的共享栈分别是第 2、3 个共享栈，没有冲突，所以协程 2 的栈内容仍然保留在第 2 个共享栈，并不拷出来，但协程 2 的寄存器会被 coctx_swap 保存在 regs 数组。调用到协程 12 时，协程 12 和协程 2 都映射到第 2 个共享栈，因此需要将协程 2 的栈内容拷出，并将协程 12 的栈内容拷贝到第 2 个共享栈中。所以共享栈多了拷出协程 2 的栈、拷进协程 12 的栈两个操作，即用拷贝共享栈的时间换取每个协程栈的空间。“
我们需要一个 idx 记录下次使用的栈的索引。模板中的 count 即表示有几个共享栈可以使用，每当 idx 大于 count 总数时，取模。

```
unsigned long alloc_idx = 0; //下次使用的栈的索引
```

构造函数即在堆上开新空间，语法上就是二维数组的构造。析构同理。

```
share_stack(int count, size_t stack_size)
    : count(count), stack_size(stack_size) {
    // TODO: implement your code here
    stack_array = new stack_mem * [count];
    for (int i = 0; i < count; i++)
    {
        stack_array[i] = new stack_mem(stack_size);
    }
}
~share_stack() {
    // TODO: implement your code here
    for (int i = 0; i < count; i++)
    {
        delete stack_array[i];
    }
    delete[] stack_array;
}
```

得到目前使用的共享栈：

```
stack_mem *get_stackmem() {
    // TODO: implement your code here
    if (alloc_idx >= count) alloc_idx = 0;
    return stack_array[alloc_idx++];
}
```

Share_stack 也就是 count 个 stack_mem
共享栈模式参考[腾讯开源的 libco 号称千万级协程支持，那个共享栈模式原理是什么？ - 知乎 (zhihu.com)]学习
参考[万字长文 | 漫谈 libco 协程设计及实现 - 知乎 (zhihu.com)]学习。
Coroutine_attr 指定协程的属性，文档已做说明。
Coroutine_env 存储协程的调用信息：
首先开一个大小为 • 128 的调用栈模拟实现协程的嵌套调用功能（其中每个元素均为协程）。call_stack_size 用于记录调用过程中协程的个数以及获取当前的协程 call_stack[call_stack_size-1]。获取当前协程挂起后该切到的协程 call_stack[call_stack_size - 2]。参考文章 2 解释，call_stack[0]一定是主协程（main），然后主协程调用协程 1，协程 1 调用协程 2……实现嵌套（犹如递归）调用的功能。
——pCallStack 即我的 call_stack
“这种递归关系的 k 最大为 127，调到协程 127 时，此时 pCallStack[0]存主协程，pCallStack[1]存协程 1...pCallStack[k]存协程 k..pCallStack[127]存协程 127。但递归如此之深的协程实际中不会遇到，更多的场景应该是主协程调用协程 1，协程 1 挂起切回主协程，主协程再调用协程 2，协程 2 挂起切回主协程，主协程再调用协程 3...因此主协程调到协程 k 时，pCallStack[0]是主协程，pCallStack[1]是协程 k，其他元素为空；协程 k 挂起切回主协程时，pCallStack[0]是主协程，其他元素为空。因此 128 大小的 pCallStack 足够上万甚至更多协程使用。”
——
由上解释实现 coroutine 和 coroutine_env 结构体：
在 coroutine 结构体中，其余部分大致与 part1 相同，增加了：

```
char* stack_sp = nullptr;
int save_size = 0; //只将其使用到的栈空间进行保存
char* save_buffer = nullptr;

stack_mem* coro_stack_mem = nullptr;
int stack_size = 0;
```

开栈存储当前协程，记录当前协程的实际使用大小和栈顶指针（方便按容量拷贝）
在 coroutine_env 结构体中，初始化便直接将主协程放在 call_stack[0]的位置（其实这里是抽象实现，也就是占了一位，把它当成是主协程即可）
Pending_co 和 occupy_co 分别为当前在工作的协程和即将要切到的协程。

```
    coroutine* call_stack[128];
    int call_stack_size = 0;
    coroutine* pending_co = nullptr;
    coroutine* occupy_co = nullptr;
    coroutine_env() {
        // init main coroutine
        coroutine* main_coro = create(nullptr, nullptr, nullptr);
        push(main_coro);
    }
```

Push 和 pop 模仿压栈出栈行为，对协程数组简单操作即可：

```
void pop() {
    call_stack_size--;
}
void push(coroutine* co) {
    call_stack[call_stack_size++] = co;
}
```

Coroutine.cpp 部分：
首先还是经典的 create，release，resume，yield 函数。新增的是将某新协程存入栈的操作和交换两协程（更换执行权，嵌套使用）的功能
首先用 memcpy——“memcpy 指的是 C 和 C++使用的内存拷贝函数，函数原型为 void *memcpy(void *destin, void *source, unsigned n)；函数的功能是从源内存地址的起始位置开始拷贝若干个字节到目标内存地址中，即从源 source 中拷贝 n 个字节到目标 destin 中。”——将给定的 attr 参数拷贝到我们 create 新创建的 attr 中。
栈大小应该在 8k 到 128k 之间，否则强制修改到边界值。
以及要保持 4k 对齐——理由上上页已有解释。
剩余代码也就是参考一下 libco/co_routine.cpp at master · Tencent/libco · GitHub 文章的 co_create_env 函数，将给定的参数的内容拷到我们新创建的协程中并返回，实现创建功能

```
coroutine* create(func_t coro_func, void* arg, const coroutine_attr* attr) {
    coroutine_attr at;
    coroutine* co = new coroutine(coro_func, arg);

    if (attr) memcpy(&at, attr, sizeof(at));
    if (at.stack_size <= 0) at.stack_size = 8 * 1024;
    else if (at.stack_size > 8 * 1024 * 1024) at.stack_size = 8 * 1024 * 1024;
    at.stack_size &= ~0xFFF;
    at.stack_size += 0x1000;//2^12  4k

    stack_mem* cur_stack_mem = nullptr;

    if (at.sstack) cur_stack_mem = at.sstack->get_stackmem();
    else cur_stack_mem = new coro::stack_mem(at.stack_size);

    co->coro_stack_mem = cur_stack_mem;
    co->stack_size = co->coro_stack_mem->stack_size;
    co->ctx.ss_size = co->stack_size;
    co->ctx.ss_sp = co->coro_stack_mem->stack_buffer;

    return co;
```

Release 将创建的协程删除即可。
Save_stack 存栈操作：首先取出给定协程的内部的自己的存储内容（自己的 stack_mem），计算得栈大小：|sp-bp|，实际上仅仅需要保存这部分，并不用把 128k 的大小全部拷贝

（大部分都没用到很浪费时间浪费空间）。若该协程缓冲区之前已经保存内容，更新，若没有，直接初始化该协程的缓冲区，更新 save_size（实际要保存的大小）。最后利用 memcpy 函数将 sp（低地址）开始的 len 长度（实际使用栈 size 的长度）拷贝到 buffer 中存储起来。

（参考 libco 的 save_stack_buffer 函数实现）

```
void save_stack(coroutine* co) {
    stack_mem* stack_mem = co->coro_stack_mem;
    int len = stack_mem->stack_bp - co->stack_sp;
    if (co->save_buffer)
    {
        delete[] co->save_buffer;
        co->save_buffer = nullptr;
    }
    co->save_buffer = new char[len];
    co->save_size = len;
    memcpy(co->save_buffer, co->stack_sp, len);
}
```

Swap 函数：挂起 current_coro，调用并开始执行 pending_coro。（功能简单，但代码复杂……好难理解，参考了 libco 开源代码）

思路参考：

“从 curr 协程切到 pending_co 协程时，如果是共享栈模式，先拿到 pending_co 的共享栈 stack_mem 里已有的协程 occupy_co，如果 occupy_co 非空且不是 pending_co，则保存已有的协程 save_stack_buffer(occupy_co)，将 stack_mem 指向的协程换为 pending_co。并将 pending_co 和 occupy_co 均保存在 env 里，不能用局部变量记录，因为局部变量在 coctx_swap 之前属于 curr 协程，但 coctx_swap 后协程栈已经被切换，curr 的所有局部变量无法被 pending_co 访问。如果 occupy_co 和 pending_co 不是同一个协程，需要将 occupy_co 在共享栈里的数据拷贝到 occupy_co->save_buffer。协程的数据除了在栈里还分布在寄存器，如果 occupy_co 不是 curr，则在 occupy_co 之前被切换到其他协程时寄存器已经被 coctx_swap 保存；否则，则其寄存器在本次执行 coctx_swap 被保存。”

自己实现思路如下：

首先将给定的参数 pending 赋值给记录协程调用信息（env）中的 pending 保存起来。

取出 pending_co 映射的共享栈信息中已有的协程（我们之前使用 owner 保存，也就是 pending 中的 occupy_co），在共享栈中记录给定的参数 pending。

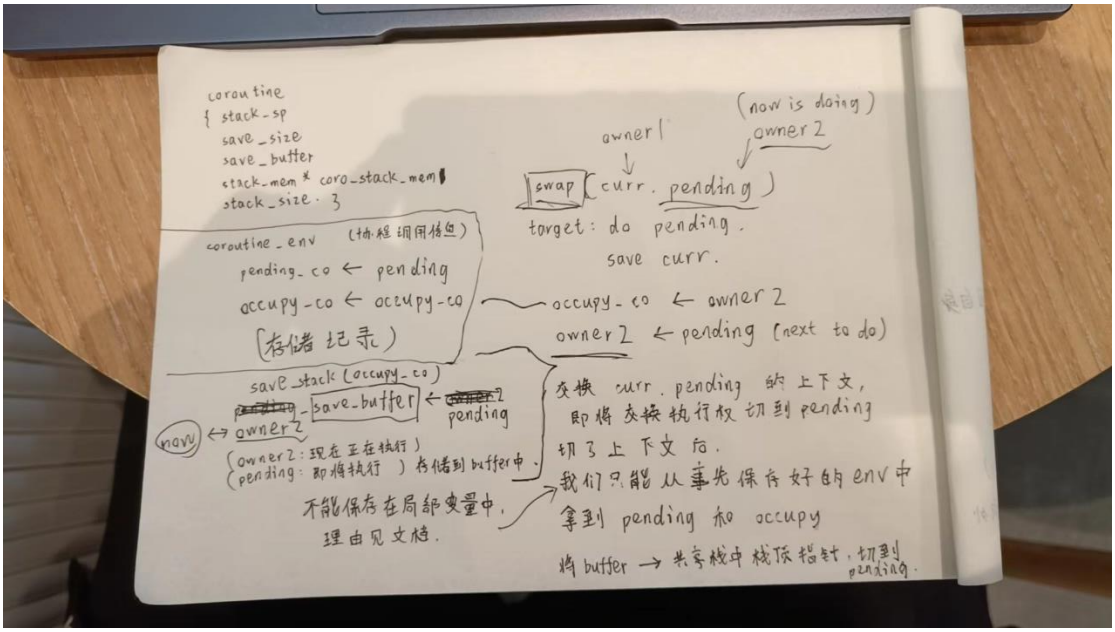
Occupy 和 pending 记录在 env 里，以实现在协程栈切换后仍然能拿到这两个变量。

交换过程的存储重点通过 env 来保存，不能使用 occupy 的局部变量，否则切换上下文后无法再获取，没法更新共享栈切换协程。

我觉得这一 part 的难度主要在代码实现上。过程虽然也很复杂，但自己画图推一下还是很好理解，主要是语法也不熟以前也没学过协程，最开始根本不知道应该要实现什么功能，也不知道该添加什么变量来进行存储……

所以这一 part 大部分代码参考了 libco 的开源代码（）

过程简略示意图如下，代码如下：




```
void swap(coroutine* curr, coroutine* pending) {
    char c;
    curr->stack_sp = &c;
    g_coro_env.pending_co = pending;
    //get last occupy co on the same stack mem
    coroutine* occupy_co = pending->coro_stack_mem->owner;
    //set pending co to occupy thest stack mem;
    pending->coro_stack_mem->owner = pending;
    g_coro_env.occupy_co = occupy_co;
    if (occupy_co && occupy_co != pending) save_stack(occupy_co);

    coro_ctx_swap(&(curr->ctx), &(pending->ctx));

    coroutine* update_occupy_co = g_coro_env.occupy_co;
    coroutine* update_pending_co = g_coro_env.pending_co;

    if (update_occupy_co && update_pending_co && update_occupy_co != update_pending_co)
    {
        if (update_pending_co->save_buffer && update_pending_co->save_size)
        {
            memcpy(update_pending_co->stack_sp, update_pending_co->save_buffer, update_pending_co->save_size);
        }
    }
}
```

协程挂起和返回代码简单：

```
int resume(coroutine* co, int param) {
    co->data = param;
    coroutine* curr = g_coro_env.get_coro(g_coro_env.call_stack_size - 1);
    if (!co->started)
    {
        ctx_make(&co->ctx, (func_t)func_wrap, co);
        co->started = 1;
    }
    g_coro_env.push(co);
    swap(curr, co);
    return co->data;
}
```

Param 的理解与 part1 相同（这里确实又发现最开始写 part1 时候的 ready 参数是冗余的了（）），用于 resume 和 yield 通信，h 文件 coroutine 用一个 data 记录即可。

Call_stack_size-1 即为共享栈中当前协程地址（索引），将其取出，即将要 resume 的 co 标记 started 为 true，压入共享栈，交换执行权即可。

```
int yield(int ret) {
    coroutine* curr = g_coro_env.get_coro(g_coro_env.call_stack_size - 1);
    coroutine* last = g_coro_env.get_coro(g_coro_env.call_stack_size - 2);

    g_coro_env.pop();

    curr->data = ret;
    swap(curr, last);
    return curr->data;
}
```

Size-1, size-2 取出当前和上次的协程，pop 即将当前协程弹出（结束执行），swap 回去执行上次的协程，参数存入当前协程的 data 用于与 resume 通信。

完成！

简单看看 main 函数：

Test1：共享栈三个元素：main，co1，co2，最初 co2 在顶层首先执行

While 循环：恢复 co1，执行 co1，恢复 co2，执行 co2……直到结束，测试了 resume 和 h 文件和 swap 的功能

Test2：也就是测试 create，resume，release 功能是否正常

Test3：测试 resume 和 yield 的通信功能是否正常，resume 中每次记录 data 字段为参数 233，yield 每次断言返回值为 233。

Test4：递归调用（嵌套调用）实现反序输出。Args 中的 n 是全局变量，每次将 n—输出 str。Resume 功能正常即可。

测试如下：

```
kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v2$ make
g++-11 -g -std=c++20 -o main coro_ctx_swap.S coro_ctx.cpp coroutine.cpp main.cpp
./kurumi@kurumi:/mnt/d/another_C/libco-handout/libco - TODO/libco_v2$ ./main
test-1 passed
test-2 passed
test-3 passed
test-4 passed
Congratulations! You have passed all the tests of libco-v2!
```

猫猫：