

part0:

任务一：

函数调用过程：

1. **调用函数**: 从 `main` 函数调用 `foo` 函数。
2. **保存返回地址**: 当调用一个函数时, 当前函数的返回地址 (调用该函数的指令的地址) 会被压入栈中, 以便在函数执行完毕后返回到正确的位置。这是为了保证程序执行的流畅性。
3. **保存参数**: 如果有参数传递给 `foo` 函数, 这些参数也会被压入栈中。
4. **创建局部变量**: `foo` 函数中的局部变量也会在栈上分配空间。
5. **执行函数体**: `foo` 函数的代码开始执行, 可以访问参数和局部变量。
6. **返回**: 当 `foo` 函数执行完成后, 会将返回值存放在事先约定的位置, 然后弹出栈顶的返回地址, 程序跳转到该地址, 继续执行。

从 `main` 调用 `foo` 到 `foo` 返回的栈帧变化过程:

```
1 main 函数调用 foo 函数:  
2 -----  
3 | foo 函数局部变量 | foo 函数参数 | 返回地址 | main 函数局部变量 |  
4 -----  
5  
6  
7 foo 函数执行中:  
8 -----  
9 | foo 函数局部变量 | foo 函数参数 | 返回地址 | main 函数局部变量 |  
10 -----  
11  
12  
13 foo 函数返回:  
14 -----  
15 返回地址 | main 函数局部变量 |  
16 -----  
17  
18 main 函数继续执行:  
19 -----  
20 | main 函数局部变量 |  
21 -----  
22
```

RBP 指向当前栈帧的基址, RSP 指向当前栈帧的栈顶

函数参数是如何传递的: 前6个整数和指针类型的参数 (按顺序) 会被依次放入寄存器 `rdi`, `rsi`, `rdx`, `rcx`, `r8`, 和 `r9` 中。如果参数超过6个, 额外的参数会被压入调用栈, 从左到右依次放在相对于栈顶的位置。

```

1 csharpCopy code0x55555540086f <add(int, int)+10>      mov     edx, dword ptr [rbp
- 0x14] ; 将第一个参数放入寄存器 edx
2 0x555555400872 <add(int, int)+13>      mov     eax, dword ptr [rbp - 0x18] ; 将
第二个参数放入寄存器 eax

```

函数的返回值是如何传递的：在调用 `add` 函数的地方，`foo` 函数中的 `ret` 值是通过 `EAX` 寄存器传递给 `foo` 函数调用处。接下来，`foo` 函数调用的返回值也会通过 `EAX` 寄存器返回给 `main` 函数调用处。

```

1 0x55555540087a <add(int, int)+21>      mov     eax, dword ptr [rbp - 4] ; 将返回
值放入寄存器 eax
2 0x55555540087d <add(int, int)+24>      pop    rbp
3 0x55555540087e <add(int, int)+25>      ret

```

```

RAX 0x2a
RBX 0x0
RCX 0x1a0
RDX 0x28
RDI 0x28
RSI 0x2
R8 0x7ffff796ed80 (initial) ← 0x0
R9 0x0
R10 0x55555602010 ← 0x0
R11 0x0
R12 0x55555400780 (_start) ← xor ebp, ebp
R13 0x7fffffffdd90 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffff7dc70 → 0x7fffffff7dc90 → 0x7fffffff7dc0 → 0x555555400950 (__libc_c
su_init) ← push r15
RSP 0x7fffffff7dc70 → 0x7fffffff7dc90 → 0x7fffffff7dc0 → 0x555555400950 (__libc_c
su_init) ← push r15
'RIP' 0x5555540087d (add(int, int)+24) ← pop rbp
[ DISASM / x86-64 / set emulate on ]
0x5555540086f <add(int, int)+10>      mov     edx, dword ptr [rbp - 0x14]
0x55555400872 <add(int, int)+13>      mov     eax, dword ptr [rbp - 0x18]
0x55555400875 <add(int, int)+16>      add     eax, edx
0x55555400877 <add(int, int)+18>      mov     dword ptr [rbp - 4], eax
0x5555540087a <add(int, int)+21>      mov     eax, dword ptr [rbp - 4]
0x5555540087d <add(int, int)+24>      pop    rbp
0x5555540087e <add(int, int)+25>      ret
↓
0x5555554008a4 <foo()+37>           mov     dword ptr [rbp - 4], eax
0x555554008a7 <foo()+40>           mov     eax, dword ptr [rbp - 4]
0x5555554008aa <foo()+43>           leave
0x5555554008ab <foo()+44>           ret
[ SOURCE (CODE) ]
In file: /home/fan/libco-handout/libco - TODO/1.cpp
1 #include<iostream>
2 int add(int a, int b) {
3     int ret = a + b;
4     return ret;
5 }
6
7 int foo() {
8     int a = 40;
9     int b = 2;
10    int ret = add(a, b);
[ STACK ]
00:0000 | rbp rsp 0x7fffffff7dc70 → 0x7fffffff7dc90 → 0x7fffffff7dc0 → 0x55555540095

```

任务2：

1. 一个“普通”的函数支持哪两个操作，分别承担了什么功能？

“正常”函数可以被看作具有两种操作：调用和返回

调用操作创建一个激活帧 (*activation frame*)，暂停调用函数的执行，并将执行权转移到被调用函数的开始。

返回操作将返回值传递给调用者，销毁激活帧，然后在调用函数被调用的点之后恢复调用者的执行。

2. 为什么我们说调用栈不能满足协程的要求？

Since coroutines can be suspended without destroying the activation frame, we can no longer guarantee that activation frame lifetimes will be strictly nested. This means that activation frames cannot in general be allocated using a stack data-structure and so may need to be stored on the heap instead. 协程的主要特征之一是其能够在执行过程中暂停并随后恢复。调用栈通常用于存储函数调用之间的状态，包括局部变量、返回地址等，并随着函数的调用和返回进行压栈和出栈操作。然而，对于协程来说，暂停和恢复的操作不仅仅涉及函数调用和返回，还包括部分状态的保持，以便在恢复时恢复执行。这导致了与传统调用栈模型不同的需求。另外协程可能在执行过程中多次暂停和恢复，需要保存不同暂停点的状态，这些状态不适合在传统调用栈上维护。传统的调用栈是按照严格的嵌套顺序来进行压栈和出栈的，而协程的状态在暂停时需要保持，因此无法完全使用调用栈的嵌套结构来满足其需求。而且，协程有时需要在其执行期间使用堆内存来保持状态，因为它们的生命周期不再严格嵌套，不能简单地使用调用栈进行内存分配和释放。这也是调用栈不足以满足协程需求的原因之一。

3. 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？

协程支持以下三个额外的操作：'Suspend'、'Resume' 和 'Destroy'。这些操作分别具有以下功能：

1. **Suspend**：暂停协程的执行，并将执行权转移回调用者或恢复者，同时保留激活帧，即函数执行的当前状态。在暂停点，协程中的对象继续保持活跃状态。
2. **Resume**：恢复被暂停的协程的执行，重新激活协程的激活帧，继续执行被暂停的位置。
3. **Destroy**：销毁协程的激活帧，但不会恢复协程的执行。这个操作仅在协程暂停时执行，用于释放激活帧占用的内存空间和清理被暂停点所引用的对象。

4. 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？

1. **内存**：部分协程状态和信息可以存储在堆内存中。协程可能在不同的暂停和恢复点之间切换，而传统的栈结构难以支持这种灵活性。使用堆内存可以在协程暂停时保留状态，并在稍后恢复。
2. **协程框架 (Coroutine Frames)**：协程框架是专门针对协程的概念，用于存储执行过程中所需的信息和状态。这包括局部变量、暂停点信息、恢复点等。由于协程状态可能不再严格嵌套于函数调用，这种框架可能需要存储在堆上，以便在多次暂停和恢复时保持状态。
3. **特殊寄存器/变量**：有些编程语言或平台可能提供特殊的寄存器或变量来存储协程的状态信息。这些寄存器或变量可能记录暂停点、恢复点等信息，使得在恢复时能够准确返回到之前暂停的位置。

任务三：

CPU的状态包括哪些？一个显然的问题是，我们不能一下保存所有的状态，结合函数调用约定，以及任务二中第四个问题的回答，你认为我们需要保存哪些状态用于暂停并恢复一个函数的运行？

CPU的状态包括众多寄存器中的内容，这些寄存器通常涵盖了程序执行所需的各种信息。

在实际操作中，并非需要保存和恢复所有的状态，因为一些状态可以在程序运行时被重建；我认为比较重要的几个内容：

1. **程序计数器**：指示了程序执行的位置，即当前指令的地址。

2. **栈指针**: 指向当前栈帧的栈顶地址。

3. **栈框架**: 包括局部变量、返回地址和其他调用相关信息。

4. **寄存器内容**: 一些特定寄存器可能包含重要的临时数据或计算中间状态。

考虑到函数调用约定，暂停和恢复函数的运行需要保存这些状态。当函数被暂停时，这些状态需要被存储以便稍后恢复函数执行。典型的做法是保存这些状态到一个指定的存储区域（例如堆内存），然后在恢复时重新加载这些状态。这些状态的保存和恢复使得函数在恢复执行时可以准确地返回到暂停的地方，并继续执行，同时保持原本函数调用的上下文不受影响。

part 1:

```
test-1 passed
test-2 passed
test-3 passed
Congratulations! You have passed all the tests of libco-v1!
```

函数实现：

creat函数：

`create` 函数用于创建一个协程，并返回该协程的指针。

```
coroutine* create(func_t func, void* args):
```

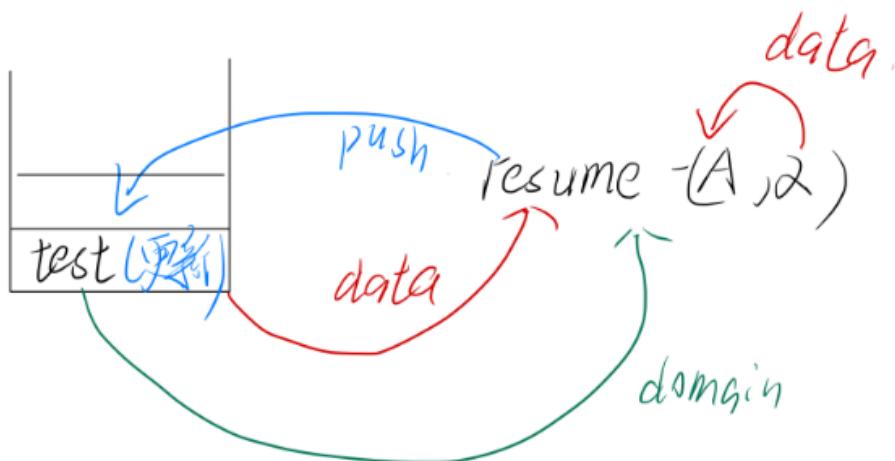
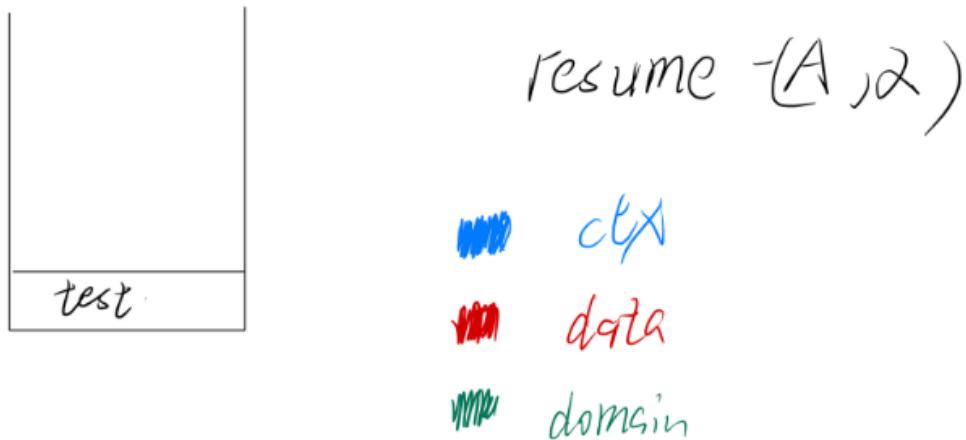
- 设置协程对象的上下文链接字段为 `nullptr`，表示在当前上下文执行完毕后，不切换到其他上下文，即结束执行。
- 使用 `makecontext` 函数设置协程对象的上下文，指定了入口函数为 `func_wrap`，参数为协程对象的指针。
- 返回创建的协程对象的指针。

```
coroutine(func_t func, void* args):
```

- 初始化协程对象的上下文。
- 设置上下文的栈起始地址为协程对象的 `stack` 数组。
- 设置上下文的栈大小为协程对象的 `stack` 数组大小。
- 设置上下文的链接字段为 `nullptr`，表示在当前上下文执行完毕后，不切换到其他上下文，即结束执行。

创建一个协程对象，初始化其上下文，为其分配栈空间，设置入口函数和参数，最后返回协程对象的指针。

env-stack

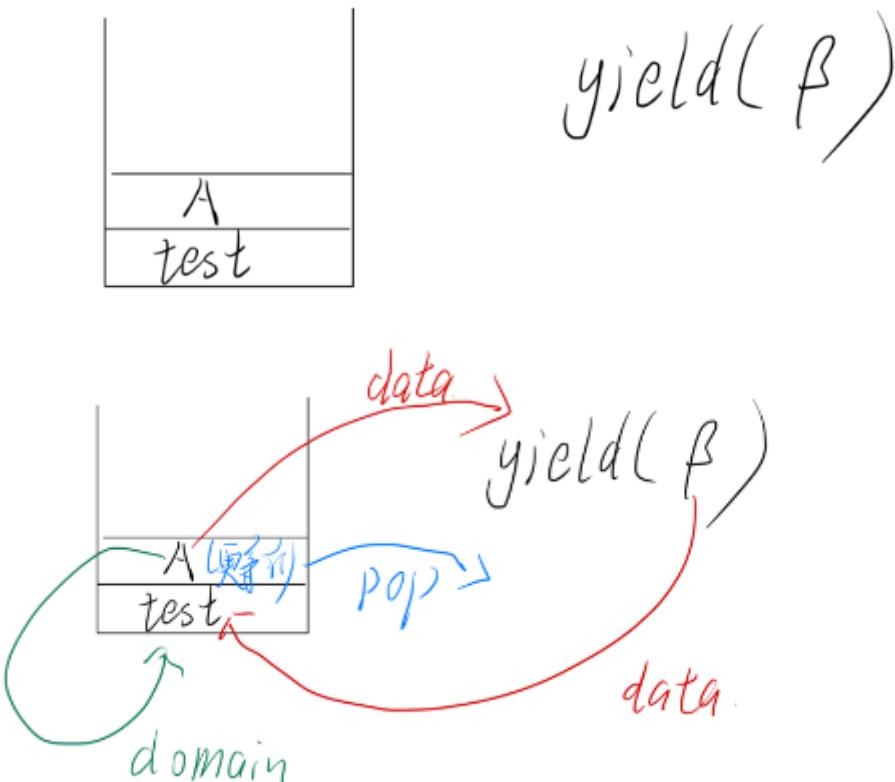


resume函数：

在第一次 resume 执行协程时进行检测，如果没有进行初始化需要进行初始化以及封装

```
1 int resume(coroutine* co, int param) {
2
3     co->data = param; // 设置传入参数
4
5     if(co->started == false){
6
7         co->started = true;
8
9         makecontext(&(co->ctx), (void(*)())func_wrap, 1, co);
10    }
11
12    g_coro_env.push(co); // 将协程推入栈中
13
14    swapcontext(&(g_coro_env.get_coro(0)->ctx), &(co->ctx));
15
16    return g_coro_env.get_coro(0)->data; // 返回yield时的数据
17
18}
```

```
19 | }  
20 |
```



yield函数：

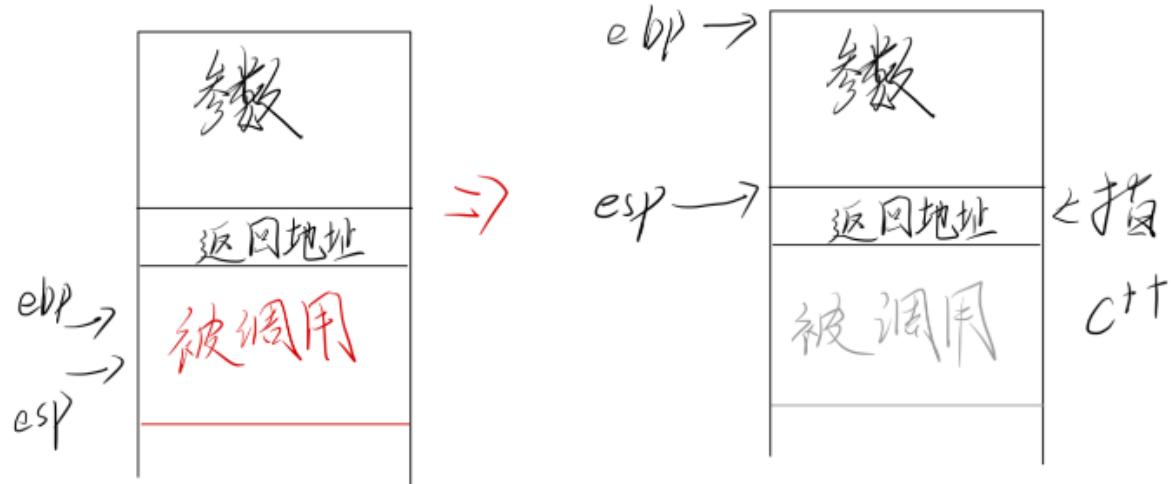
将正在运行的协程pop出来，协程恢复为resume传的数据

```
1  
2 int yield(int ret) {  
3  
4     coroutine* current = g_coro_env.get_coro(1); // 获取当前协程  
5  
6     int par = current->data;  
7  
8     g_coro_env.get_coro(0)->data = ret; // 保存返回值  
9  
10    g_coro_env.pop(); // 弹出栈顶协程  
11  
12    swapcontext(&(current->ctx), &(g_coro_env.get_coro(0)->ctx)); // 切换到调  
用者  
13  
14    return par; // 返回resume时传入的数据  
15  
16}  
17
```

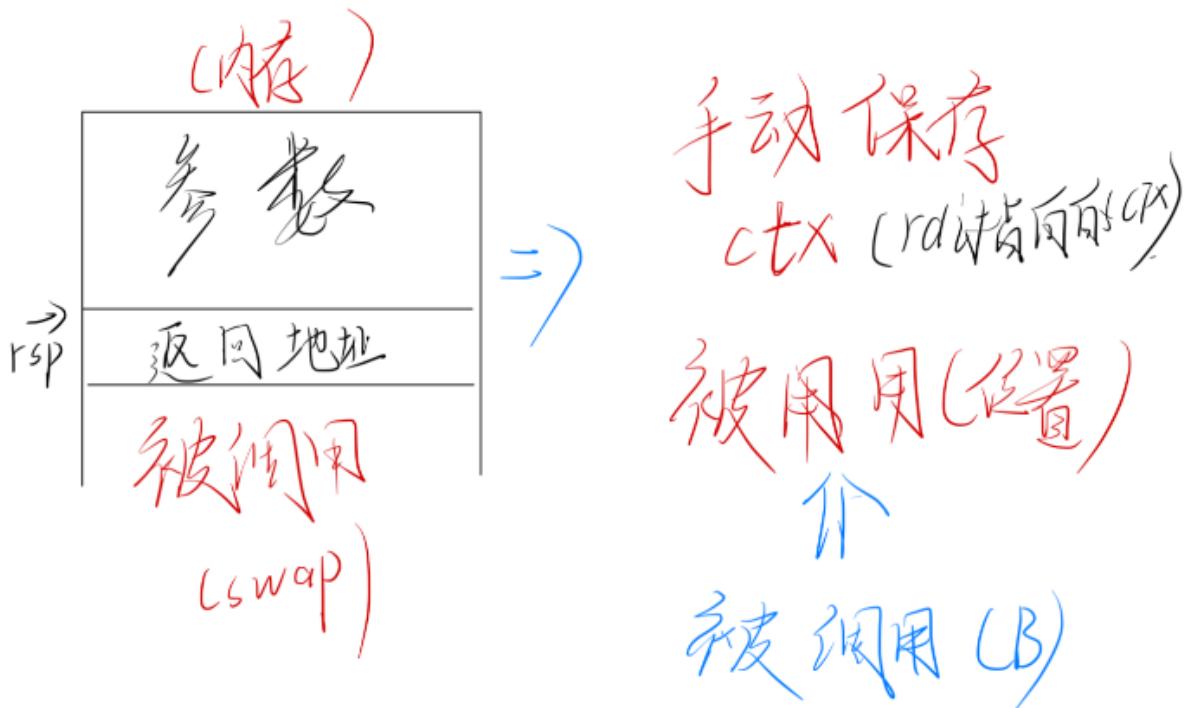
Part2:

```
test-1 passed
test-2 passed
test-3 passed
test-4 passed
Congratulations! You have passed all the tests of libco-v2!
```

coro_ctx 结构体



上面是一个正常的函数调用栈帧，但是我们需要将返回地址压栈之后，把**被调用**手动转换到另一个内存**被调用**，并且保存手动保存**ctx**，因此**context**里**模拟一个内存**，需要一个stack储参数以及一个十四位寄存器数组。其中**rsp**指向了协程函数的首地址，**rdi**指向了函数的参数。其中**stack_rsp**为栈底指针，先将**return_address**压栈，然后创建栈帧。



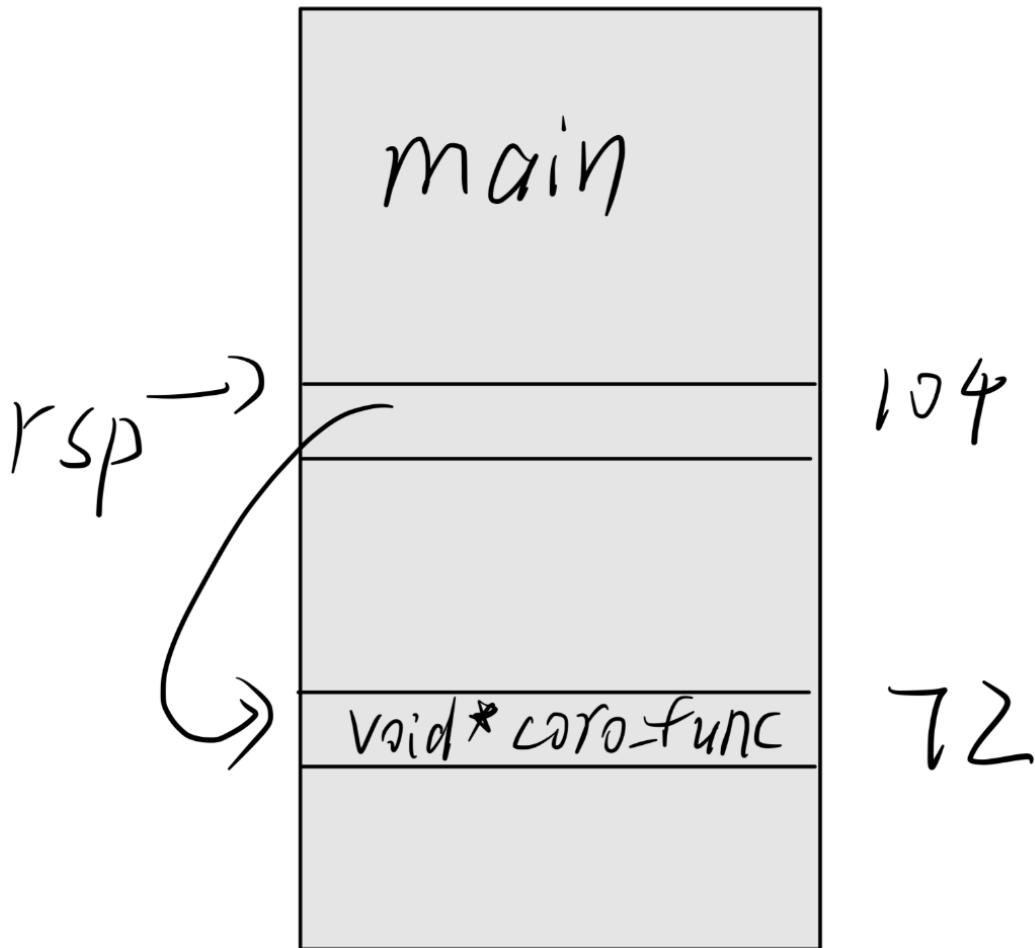
```

1 struct context{
2
3     void *regs[14];
4
5     size_t stack_size;
6
7     char *stack_rsp;
8
9 } ;mespace coro
10
11
12
13

```

ctx_make 函数

ctx_make 函数中为协程创建上下文并初始化它。



```
1 void ctx_make(context *ctx, func_t coro_func, const void *arg)
2 {
3
4     char *rsp = ctx->stack_rsp + ctx->stack_size - 8; // 栈顶指针下移，防止返回地址覆盖
5
6     memset(ctx->regs, 0, sizeof(ctx->regs)); // 初始化
7
8
9     void **return_address = (void **)(rsp); // 将72的指针赋给104
10
11    *return_address = (void *)coro_func; // 将封装函数封装到72
12
13
14
15    ctx->regs[kRSP] = (char *)rsp; // 104存到寄存器
16
17    ctx->regs[kRETURN] = (char *)coro_func; // 72存到寄存器
18
19
20    ctx->regs[kRDI] = (char *)arg;
21
```

```
22  
23  
24     }  
25
```

coro_ctx_swap

coro_ctx_swap函数首先保存上下文(六个save寄存器、六个传参寄存器、rsp、rdi)rsp->rdi

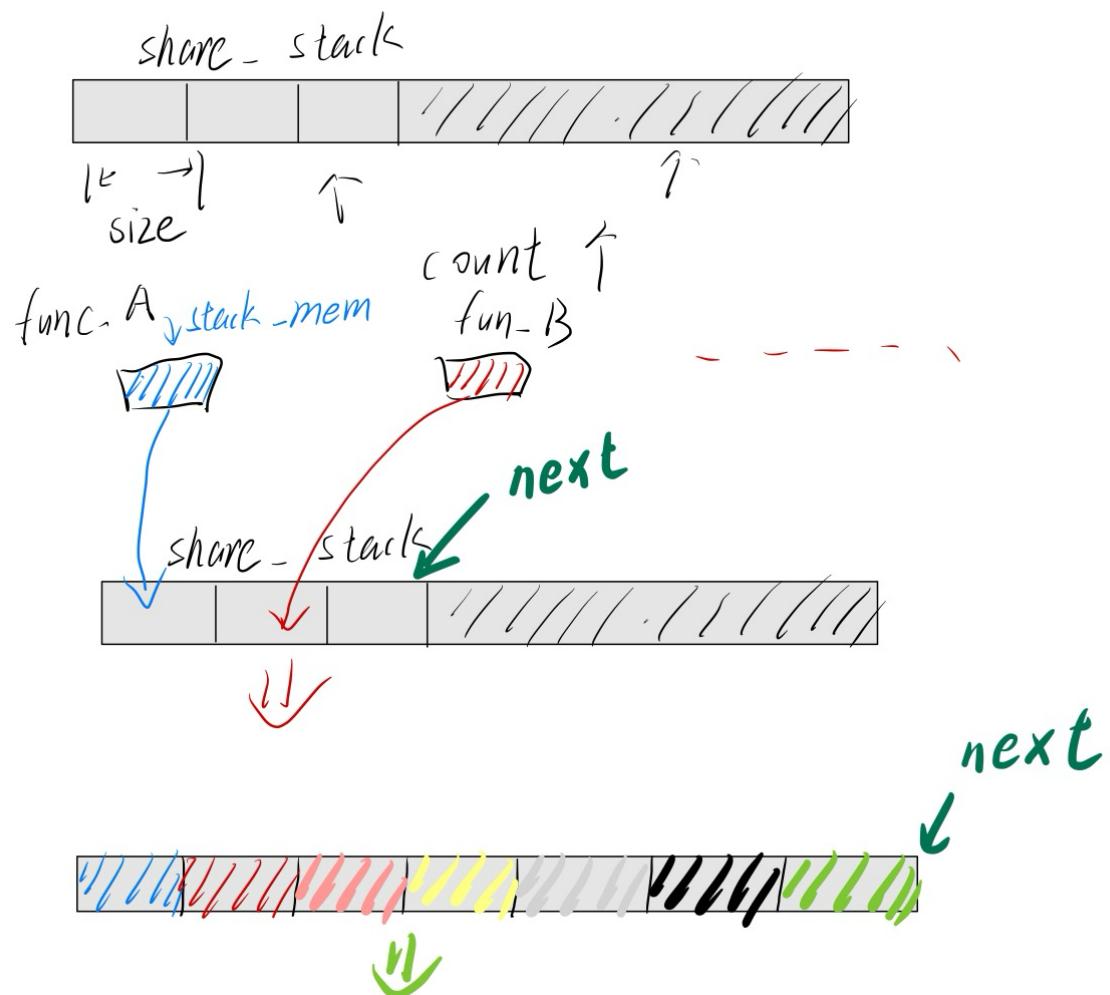
然后恢复callee的上下文rsi->%rsp(.....),并且把返回地址赋值给栈顶rsi, 然后ret栈顶 (rsi) 交出控制权
~~有好多寄存器一直是0 (比如64什么的) 也一并传过去算了。~~

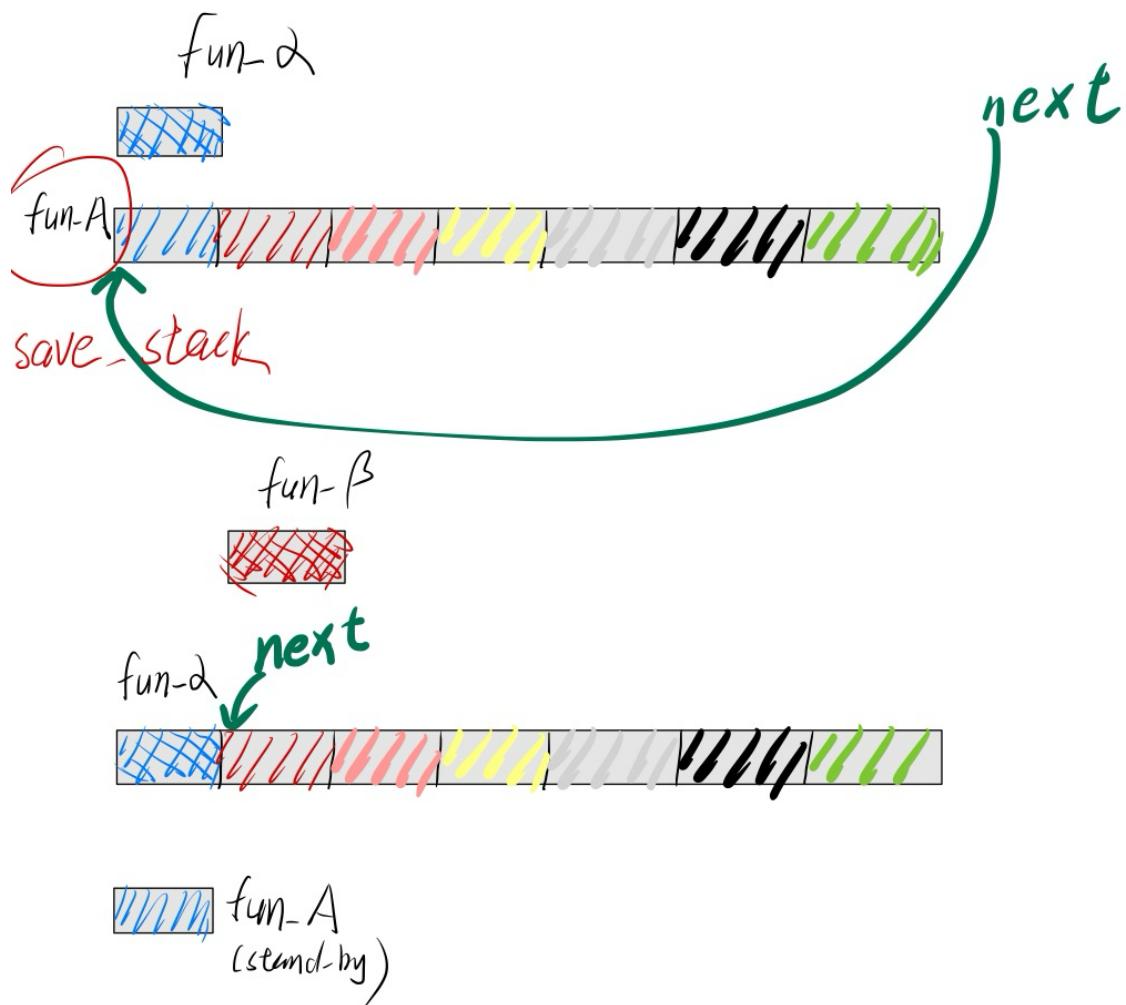
```
1      leaq (%rsp),%rax  
2  
3      movq %rax, 104(%rdi)  
4 //从104到0  
5      movq %r15, (%rdi)  
6      xorq %rax, %rax  
7  
8  
9      movq (%rsi), %r15  
10 //0到104  
11  
12      movq 104(%rsi), %rsp  
13  
14  
15      movq 72(%rsi),%rax//注意不可以两边取 ()  
16  
17      movq %rax,%(rsp)//把返回地址赋值给rsp  
18  
19      ret//交出控制权  
20  
21  
22  
23
```

coroutine与从coroutine_env与v1基本一致

part 2.5

Share_stack





在内存中圈出一块内存，作为共享栈；采用循环队列的方式，**每一个stack_member对应位置自从初始化之后就对应该位置**；一直到如果共享栈被全部占用，就重新从零号位置开始。当一个栈 α 被分配到共享栈对应位置0，但是对应位置0已经有了栈 α ；那么就把栈 α 的信息存放在私有栈里，然后将栈 α 存放进共享栈。

```

1 struct share_stack {
2     stack_mem** stack_array; //指向栈内存块的指针数组
3
4     int count = 0; //共享栈大小
5
6     size_t stack_size = 0; //每个栈的大小
7
8     int position = 0; //如果共享栈全部被占用，需要进行保存、覆盖操作的计数器
9
10
11
12     share_stack(int count, size_t stack_size) : count(count),
13         stack_size(stack_size) {
14
15         stack_array = new stack_mem*[count];
16
17         for (int i = 0; i < count; ++i) {

```

```
18     stack_array[i] = new stack_mem(stack_size);
19
20 }
21
22 }
```

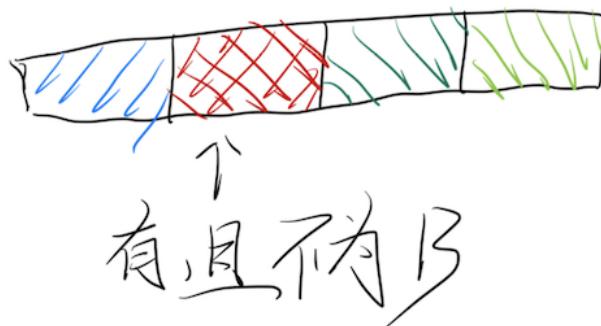
Save_stack:

```
1 void save_stack(coroutine* co) { //旧的栈被清走的时候，保存旧的结构体的信息
2
3     int len = co->stack->stack_start +co->stack->stack_size - co->stack_sp;
4     if(co-> stack_mem_sp ){//如果之前进行过save_stack，需要先把之前的free掉
5         free(co-> stack_mem_sp);
6         co->stack_mem_sp = nullptr;
7     }
8     co->stack_mem_sp = (char *)malloc(len); //开辟一块新的内存
9     co->stack_mem_size = len;
10    memcpy(co->stack_mem_sp,co->stack_sp, len);
11 }
```

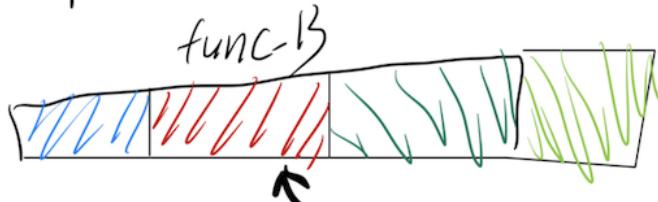
Swap:



save-stack:



如果 B 经历过 save-stack



func-B
(swap-by)代码中 stack_mem_sp



寄存器交换

ctx-swap

实现之前Share_stack所说的是否对 callee 对应栈上的 stack_mem 的 last_user 进行 save_stack 功能，以及进行 coro_ctx_swap

如果按照字面意思写，就对每一步进行判断，如果对应共享站位置已经被占用并且使用者 last_user 不是 caller，就将这个栈上内容保存在 last_user 私有栈。

如果共享栈被全部占用，就重新从零号位置开始。当一个栈 α 被分配到共享栈对应位置0，但是对应位置0已经有了栈 a ；那么就把栈 a 的信息存放在私有栈里，然后将栈 α 存放进共享栈。

```
1 void swap(coroutine *caller, coroutine *callee)
2 {
3
4     char c;
5     caller->stack_sp = &c; //通过创建在内存上新建变量，让其正好在未使用的内存部位
6     static coroutine *last_member = nullptr;
7     static coroutine *next_member = nullptr;
8     if (!pending->is_shared) //没有使用过共享栈
9     {
10
11         next_member = nullptr;
12
13         last_member = nullptr; //不需要进行save_stack以及memcpy操作
14     }
15     else //如果使用共享栈
16     {
17         next_member = callee;
18         coroutine *occupy = callee->stack->last_user; //callee所对应共享栈的
19         上一人使用者
20         callee->stack->last_user = callee; //把callee放到共享栈上
21         last_member = occupy;
22         if (occupy && occupy != callee){
23             save_stack(occupy); } //如果上一任使用者不为callee，将其save_stack
24
25     }
26
27     //callee是否进行过save_stack
28
29     if (last_member && next_member && last_member != next_member) {
30
31         if (next_member->stack_mem_sp && next_member->stack_mem_size > 0)
32     {
33
34             memcpy(next_member->stack_sp, next_member->stack_mem_sp,
35             next_member->stack_mem_size); } //如果进行过save_stack，将其私有栈中信息进行复制
36
37
38     coro_ctx_swap(&(caller->ctx), &(pending->ctx));
39
40 }
41 }
```

但是我注意到，如果B因为不再共享栈上，如果经历过 `save_stack` 需要 `memcpy`，如果没经历过 `save_stack` 说明只进行过初始化，也可以进行 `memcpy`，所以可以直接把 `memcpy` 放进 `save_stack` 的条件判断里。

简化版 [↓](#)

```

1 void swap coroutine *caller, coroutine *callee)
2 {
3     if (callee->is_shared){
4         coroutine *occupy = callee->stack->last_user;
5         callee->stack->last_user = callee;//与之前一致
6         if (occupy && occupy != callee){//如果需要对上一任使用者save_stack,
7             把callee私有栈上信息赋值到共享栈
8             save_stack(occupy);
9             memcpy( callee->stack_sp, callee->stack_mem_sp, callee-
10            >stack_mem_size);
11        }
12    }
13    coro_ctx_swap(&(caller->ctx), &(callee->ctx));
14 }
15

```

part 3:

先用c++看一遍这几个宏在干什么，通过使用 `switch` 语句和 `goto` 语句来实现状态的切换。

接下来，逐步把逻辑过程用宏来替换。

对应思路写出的c++程序

```

1 class Fib : public CoroutineBase {
2 private:
3     int a = 0, b = 1;
4 public:
5     int operator()() {
6
7         int result = -1;
8         //执行CO_BEGIN
9         switch (started) {
10             case 0:
11                 //Coroutine logic
12                 while (true) {
13

```

```

14         //执行CO_YIELD(a);
15         started = __LINE__;
16         return a;
17         case __LINE__: } //执行CO_END
18     }
19 }
20
21 //执行CO_RETURN(...)
22
23 started = -1;
24
25 return result;
26
27 }
28
29 };
30
31
32

```

1. CO_BEGIN 和 CO_END:

- CO_BEGIN 和 CO_END 是函数的起始和结束位置， CO_BEGIN 中的 switch (started) 用于根据 started 的值执行不同的代码块，从而实现协程的状态切换。
- CO_END 结束 switch 语句的定义，即加上}。

2. CO_YIELD:

- 当执行到 CO_YIELD 时，将 started 设置为当前行号（__LINE__），然后返回相应的值，当下一次的时候通过这个值来进行返回以及定位。

3. CO_RETURN:

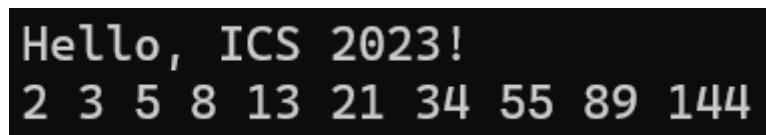
- CO_RETURN 结束协程函数的执行。 started 设置为 -1，表示协程函数已经执行完毕。

```

1 #define CO_BEGIN switch (started) { case 0:
2
3
4 #define CO_END   }
5
6 #define CO_YIELD(...) do { started = __LINE__; return __VA_ARGS__; case
7 __LINE__: } while (0)
8
9 #define CO_RETURN(...) do { started = -1; return __VA_ARGS__; } while (0)

```

Part4:



任务四：

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

- 构造函数和析构函数。
- `get_return_object()`：返回与协程相关联的对象。
- `initial_suspend()`：定义协程开始时的行为。
- `final_suspend()`：定义协程结束时的行为。
- `unhandled_exception()`：处理协程内未捕获的异常。
- `return_value()` 或 `return_void()`：用于返回一个值或指示完成而没有值

2. `Promise` 对象需要提供哪些函数？

- 构造函数，包括默认构造函数。
- `get_return_object()`：返回协程句柄或表示协程最终输出的其他对象。
- `initial_suspend()`：在协程开始时提供一个挂起点。
- `final_suspend()`：协程结束时的挂起点。
- `unhandled_exception()`：用于处理协程内的异常。
- `return_void()` 或 `return_value()`：用于返回一个值或指示完成而没有值

3. `Awaitable object` 需要提供哪些接口？

- `await_ready()`：检查可等待对象是否就绪。
- `await_suspend()`：如果不就绪，则挂起协程。
- `await_resume()`：一旦就绪，恢复协程

4. `Coroutine handle` 通常需要提供哪些函数？

- `resume()`：恢复协程的执行。
- `destroy()`：销毁协程状态。
- `from_address()`：从给定地址创建协程句柄。
- `address()`：返回协程状态的地址。
- `done()`：检查协程是否已经执行完成

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

它们被编译器转换为 `co_await` 表达式，通过提供便捷的方式在协程中返回或产生值，同时在幕后利用 `co_await` 机制进行 suspending and resuming execution。

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。

协程函数调用过程：

- 当协程函数被调用时，编译器首先根据协程的返回类型确定 `promise_type`。
- 创建协程状态（Coroutine State），其中包含 `promise` 对象、函数参数副本、挂起点信息以及跨挂起点的局部变量/临时变量的存储空间。
- 调用 `get_return_object()` 方法，从 `promise` 对象获取协程的返回值。
- 进入初始挂起点（Initial Suspend Point），通常使用 `co_await promise.initial_suspend()`。
- 执行协程体。在协程体中，可以有多个挂起点，通过 `co_await`、`co_yield` 或 `co_return` 表达式实现。
- 最终挂起点（Final Suspend Point）被执行，通常是 `co_await promise.final_suspend()`。
- 协程结束，清理协程状态。

接口函数的功能：

- `promise_type` 中的方法：
 - `get_return_object()`：创建与协程相关联的对象，通常是协程的句柄。
 - `initial_suspend()`：在协程开始时提供一个挂起点。
 - `final_suspend()`：在协程结束时提供一个挂起点。
 - `unhandled_exception()`：处理协程内部未捕获的异常。
 - `return_void()` 或 `return_value()`：返回值或表示协程完成但无返回。
- `Awaitable` 对象的接口：
 - `await_ready()`：检查对象是否已准备好。
 - `await_suspend()`：如果对象未准备好，则挂起协程。
 - `await_resume()`：一旦准备好，恢复协程执行。
- `coroutine_handle` 的功能：
 - `resume()`：恢复协程的执行。
 - `destroy()`：销毁协程状态。
 - `from_address()`：从给定地址创建协程句柄。
 - `address()`：返回协程状态的地址。

- `done()`: 检查协程是否已完成执行。

以上描述了协程函数的调用过程和相关接口函数的功能，这些功能共同管理了协程的生命周期、状态和控制流。

generator.h:

1. Promise Type:

- 实现 `promise_type` 类，它是协程的核心，用于管理协程的状态和生命周期。
- 包含方法如 `get_return_object()`, `initial_suspend()`, `final_suspend()`, `return_void()` 或 `return_value()` 以及 `yield_value()` (用于 `co_yield`) 。

2. Generator 构造函数和析构函数:

- 构造函数应该初始化协程。
- 析构函数应该确保协程被正确销毁。

3. Iterator 类:

- 实现迭代器构造函数，可能需要使用promise的句柄。
- 实现运算符 `==` 和 `!=`，以支持迭代器比较。
- 实现前缀和后缀的 `++` 运算符，用于移动迭代器。
- 实现 `*` 和 `->` 运算符，用于访问迭代器指向的元素。

4. Begin() 和 End() 成员函数:

- `begin()` 应该返回一个指向协程第一个元素的迭代器。
- `end()` 应该返回一个表示协程结束的迭代器。

5. 成员变量:

- 根据需要添加任何必要的成员变量，如协程句柄。

一眼看上去不知道从哪里开始入手，那就从上往下一步一步实现，然后再统一接口。

generator

构造函数

通过协程句柄来进行generator的构造，将句柄的所有权从一个生成器对象 `other` 转移到新创建的生成器对象；句柄放在 `private` 里。

```
1 | generator(generator&& other) : coro_(other.coro_) {  
2 |     other.coro_ = nullptr;  
3 | }
```

promise_type

从构造到暂停到恢复 (resume通过handle执行) , 需要promise有相对应的关键字函数

get_return_object

`get_return_object` 当协程函数被调用时, 编译器会根据协程的返回类型确定 `promise_type`。然后, 编译器会调用 `promise_type` 中的 `get_return_object` 函数来创建与协程相关联的 `generator` 对象。

所以我需要一个接受协程句柄作为参数, 用于创建生成器对象的`generator`的构造函数, 我把它写在了`generator`的 `private` 里面

```
1 | explicit generator(std::coroutine_handle<promise_type> coro) : coro_(coro) {}  
  
1 | generator get_return_object() {  
2 |     return  
3 |     generator{std::coroutine_handle<promise_type>::from_promise(*this)};  
3 | }
```

yield_value

`co_yield` 语句用于在协程中产生一个值。这个值会被传递给与协程相关联的 `promise_type` 结构体中的 `yield_value` 函数, 所以我需要先把 `generator` 储存这个值; 然后返回 `std::suspend_always{}`, 告诉编译器在这里挂起协程。

```
1 | std::optional<value> current_value;  
2 | std::suspend_always yield_value(value value) {  
3 |     current_value = std::move(value);  
4 |     return {};  
5 | }  
6 |
```

其他

`initial_suspend()` 控制协程开始时的挂起, `final_suspend()` 控制协程执行完成时的挂起。
`return_void()` 函数用于处理无返回值的协程 (当函数调用的时候)。当协程使用 `co_return;` 语句结束时, 会调用这个函数。`unhandled_exception()` 函数用于处理协程内部未捕获的异常。当协程内发生了未捕获的异常时, 这个函数会被调用。

iterator

`iterator` 类的基本原理是为 `generator` 类提供一个迭代器, 使得通过迭代器能够遍历生成器产生的序列。`iterator` 类通过存储协程句柄、定义迭代器特性别名以及实现迭代器的基本行为, 使得通过迭代器可以有效地访问生成器产生的序列。它是生成器与迭代器之间的桥梁, 提供了在范围表达式和标准库算法中使用生成器的便捷方式。

`iterator` 类中的两个 `operator++` 函数, 分别用于前置和后置递增; 在序列上移动到下一个元素。即从一个`co_yield`暂停点一直运行到下一个暂停点; 前置递增恢复协程, 并且返回引用; 后置递增需要先保存当前迭代器, 然后恢复协程, 返回之前保存的协程, 完成多次迭代。

```

1 iterator& operator++() {
2
3     coro_.resume(); //恢复与迭代器关联的协程的执行
4     if (coro_.done()) coro_ = nullptr; //表示迭代器已经到达末尾
5     return *this;
6 }
7
8 iterator operator++(int) {
9
10    iterator temp = *this; //代表了当前迭代器
11
12    ++(*this); //当前迭代器对象执行一次递增操作
13
14    return temp; //返回的是递增之前的迭代器状态的副本
15
16 }
17

```

`== !=` 用于比较迭代对象 `coro_`, 判断是否正确执行

```

1 bool operator==(const iterator& other) const {
2
3     return coro_ == other.coro_;
4
5 }
6
7 bool operator!=(const iterator& other) const {
8
9     return !(*this == other);
10
11 }
12

```

起始和终止迭代器:

`iterator begin():`

- `begin` 函数用于获取生成器的起始迭代器。
- 如果当前协程句柄 `coro_` 不为 `nullptr`, 表示生成器尚未结束, 调用 `coro_.resume()` 恢复协程的执行。这确保在开始迭代时生成器的协程已经开始执行。
- 返回一个迭代器对象。如果协程已经执行完成, 返回的是结束迭代器; 否则返回当前协程句柄关联的迭代器对象。

```

1 iterator begin() {
2     if (coro_) coro_.resume();
3     return coro_.done() ? end() : iterator{coro_};
4 }

```

Part 5:

```
Start libco_v5 test
libco_v5 task test passed!
```

generator.h:

递归实现思路：

1. 当前生成器通过 `await_suspend` 将协程控制权传递给嵌套生成器，同时设置嵌套生成器的 `root_` 指针为当前生成器的根协程。
2. 如果嵌套生成器内部发生了异常，异常指针会在 `await_resume` 中设置，并通过 `std::rethrow_exception` 递归传递给当前生成器。
3. 嵌套生成器执行完成后，通过 `await_suspend` 将协程控制权返回给当前生成器，同时重新设置当前生成器的一些属性（如 `leaf_` 指针等）。

需要的三个生成器：

1. **当前生成器 (Current Generator) :**
 - 当前生成器即是正在执行的生成器，它是调用嵌套生成器的生成器。表示为 `current`。
2. **嵌套生成器 (Nested Generator) :**
 - 嵌套生成器是在当前生成器内部通过 `yield_sequence_awaiter` 结构体嵌套调用的生成器。表示为 `nested`。
3. **父生成器 (Parent Generator) :**
 - 父生成器是当前生成器的直接调用者，也是嵌套生成器的外层生成器。表示为 `parent`。

调用树：

由此，我们还原调用树的建立以及维护过程：

1. **根协程的创建 (Root Coroutine Creation) :**
 - 当创建一个生成器对象时，会同时创建一个根协程。根协程是整个协程调用树的根节点。（和）
2. **嵌套生成器的调用 (Invocation of Nested Generator) :**
 - 当一个生成器在其协程体内通过 `co_await` 调用另一个生成器时，会触发嵌套生成器的执行。
 - 在 `yield_sequence_awaiter` 的 `await_suspend` 函数中，建立了嵌套生成器与当前生成器之间的关联关系。具体来说，将嵌套生成器的 `root_` 设置为当前生成器的根协程，将当前生成器的 `leaf_` 设置为嵌套生成器。
3. **异常的传递 (Exception Propagation) :**

- 在嵌套生成器执行过程中，如果发生了异常，异常会被捕获并设置在 `yield_sequence_awaiter` 的 `await_resume` 函数中。然后，通过 `std::rethrow_exception` 将异常递归传递给当前生成器。这样，异常就能够从嵌套生成器一直传递到当前生成器，形成了异常的递归传递。

4. 协程调用关系的维护 (Coroutine Relationship Maintenance) :

- 通过 `yield_sequence_awaiter` 中的操作，维护了生成器之间的调用关系。具体来说：
 - 当前生成器的 `leaf_` 指针指向了嵌套生成器。
 - 嵌套生成器的 `root_` 指针指向当前生成器的根协程。
 - 嵌套生成器的 `parent_` 指针指向当前生成器。

5. 协程的恢复和挂起 (Resumption and Suspension of Coroutines) :

- 在 `await_suspend` 函数中，通过挂起当前生成器，将控制权转交给嵌套生成器。当嵌套生成器执行完成后，再次通过 `await_suspend` 将控制权返回给当前生成器。

这样，通过以上步骤，就在协程调用链上建立了一棵树状的结构。这棵树反映了生成器之间的嵌套关系，同时通过指针关联实现了协程调用的树状结构。在每一层的生成器调用中，都会维护相应的关联关系，使得协程能够递归调用并传递异常。

代码实现：

根据搭建调用树的思路，我们进行代码的实现

创建根节点

```
1 | explicit generator(std::coroutine_handle<promise_type> coro) noexcept:
  coro_(coro) {}
```

`yield_sequence_awaiter` 结构体的实现

当嵌套的 `generator` 被 `co_await` 时，会暂停当前的协程，并通过 `await_suspend` 函数来指定接下来的执行流程。在这里，主要是将嵌套 `generator` 的执行环境（`promise_type` 的相关信息）与当前 `generator` 关联起来，并设置好执行的上下文。最后，返回嵌套 `generator` 的协程句柄，以告诉编译器将控制权转移到哪个协程。

- 在 `yield_sequence_awaiter` 结构体中，`await_suspend` 函数是嵌套生成器执行前的挂起点。在这里，当前生成器 (`current`) 将控制权挂起，将协程的执行权交给嵌套生成器 (`gen_`)。
- 同时，还设置了嵌套生成器的 `parent_` 指针指向当前生成器，建立了嵌套生成器与当前生成器的父子关系。

```

1 std::coroutine_handle<> await_suspend(std::coroutine_handle<promise_type> h)
2     noexcept {
3         auto& current = h.promise();
4         auto& nested = gen_.coro_.promise(); // 将嵌套 generator 的根 promise 设置为
5             // 当前 generator 的根 promise
6         auto& root = current.root_; // 将当前 generator 的根 promise 的 leaf 设置为嵌套
7             // generator 的 promise
8         nested.root_ = root; // 将嵌套 generator 的根 promise 设置为当前 generator 的
9             // 根 promise
10        root->leaf_ = &nested; // 将当前 generator 的根 promise 的 leaf 设置为嵌套
11        generator 的 promise
12        nested.parent_ = &current; // 将嵌套 generator 的 parent 设置为当前 generator
13        的 promise
14        nested.exception_ = &exception_; // 将嵌套 generator 的异常指针指向当前
15        generator 的异常指针
16
17        return gen_.coro_;
18    }

```

3. 在 `await_suspend` 函数中，首先获取了当前生成器的 `root_` 指针，然后将嵌套生成器的 `root_` 设置为当前生成器的根协程，建立了嵌套生成器与当前生成器之间的关联。
4. 如果在嵌套生成器的执行过程中发生了异常，异常指针将被设置，并在 `await_resume` 函数中通过 `std::rethrow_exception` 递归传递回当前生成器。

```

1 void await_resume() {
2     if (exception_)
3         std::rethrow_exception(std::move(exception_));
4 }

```

final_awaiter

协程最终挂起点实现

`final_awaiter` 实现了在协程执行完成后的最终挂起点。在协程体执行结束时，通过 `final_suspend` 返回的 `final_awaiter` 对象，触发了相应的挂起和恢复逻辑，从而完成协程的生命周期。在协程的生命周期中，最终挂起点是指协程执行完毕后的最后一次挂起。

1. `final_awaiter` 结构体

`final_awaiter` 定义了三个关键函数：`await_ready`、`await_suspend` 和 `await_resume`。

- `await_ready` 函数表示协程是否可以立即继续执行。在这里，总是返回 `false`，意味着协程需要挂起。
- `await_suspend` 函数是实际的挂起点。在协程执行到最终挂起点时，该函数将被调用。它接受当前协程的句柄 `h` 作为参数，然后通过 `h.promise()` 获取协程的 `promise` 对象，检查是否存在父协程。
 - 如果存在父协程 (`parent` 不为 `nullptr`)，则通过设置当前协程的根协程的 `leaf_` 指针为父协程，将控制权转移给父协程。

- 如果不存在父协程，表示当前协程是最顶层的协程，则返回 `std::noop_coroutine()`，表示不执行任何操作。

```

1 std::coroutine_handle<>
2 await_suspend(std::coroutine_handle<promise_type> h) noexcept {
3     auto& promise = h.promise();
4     auto parent = h.promise().parent_;
5
6     // 检查是否存在父协程
7     if (parent) {
8         // 如果存在父协程，将当前协程的根协程的 leaf_ 指针设置为父协程
9         promise.root_>leaf_ = parent;
10        // 返回 std::noop_coroutine()，表示不执行任何操作，将控制权转移给父协程
11        return std::noop_coroutine();
12    }
13
14    // 如果不存在父协程，同样返回 std::noop_coroutine()
15    return std::noop_coroutine();
16 }
```

- `await_resume` 函数在协程从挂起状态恢复时被调用。在这里，实现为空，因为最终挂起点不需要执行任何操作。

2. final_suspend 函数

`final_suspend` 函数返回一个 `final_awaiter` 对象，表示协程的最终挂起点。当协程执行到最终挂起点时，将调用 `final_awaiter` 对象的相关函数。

异常值处理：

`void unhandled_exception()` 函数用于处理协程执行过程中的未处理异常。如果协程执行过程中发生了异常且未被处理，即 `exception_` 指针不为 `nullptr`，则将当前异常对象存储在 `exception_` 指向的位置。如果 `exception_` 为 `nullptr`，则直接抛出当前异常。

```

1 // 处理未捕获异常
2 void unhandled_exception() {
3     if (exception_ == nullptr)
4         throw;
5     std::rethrow_exception(std::current_exception());
6 }
7 }
```

sleep.h

`sleep` 包括 `await_ready`、`await_suspend` 和 `await_resume` 函数。

- `await_ready` 函数返回 `false`，表示协程需要挂起。因为我们的目的是实现一个异步等待，所以始终返回 `false`。

- `await_suspend` 函数将一个带有延迟逻辑的函数 (lambda) 推送到 `task_queue` 中，然后协程挂起。该函数中的 lambda 函数会在一段时间后判断是否满足条件，如果满足，则恢复协程执行。
- `await_resume` 函数为空操作，因为在我们的实现中，协程被挂起后并没有真正的返回值。

`Task` 结构体为一个协程句柄，添加 `get_return_object`、`initial_suspend`、`final_suspend`、`return_void` 和 `unhandled_exception`。

`wait_task_queue_empty` 函数的目的是等待任务队列为空，以确保所有异步任务都得到执行，协调异步任务执行顺序

1. **循环检查任务队列：** 使用 `while (!task_queue.empty())` 循环结构，持续检查任务队列是否为空。
2. **获取并执行任务：** 通过 `auto task = task_queue.front();` 获取队列中的第一个任务，并通过 `task()` 执行任务逻辑。如果任务返回 `false`，说明任务未完成。
3. **重新推送未完成的任务：** 将未完成的任务通过 `task_queue.push(task);` 重新推送到队列的尾部。这确保了任务会在队列中等待下一次执行机会。
4. **弹出已执行或重新推送的任务：** 使用 `task_queue.pop();` 弹出队列的第一个任务，因为它已经被执行或重新推送。
5. **短暂等待：** 通过 `std::this_thread::sleep_for(std::chrono::milliseconds(10));` 进行短暂的等待，以避免过于频繁地尝试执行任务，减轻系统负担。

感受：

v1和v2写了好久好久，v1是因为学了栈帧好多的基础知识；v2是因为为了理解共享栈，生看微信的代码（可恶为什么没有参考文档QwQ）看了好久；导致我写第二部分的时候，每天看到携程给我发消息我都ptsd。整个lab写下来，最大的感受还是不要过度依赖chatgpt给你写代码，v1的时候企图用chatgpt帮我构建框架结构，然后顺着它的思路往下写，然后发现逻辑上就是错的，卡了好久好久。

还有就是，没有想到自己学完gdb之后真的使用上了，用vscode打断点，直接跳segmentation fault；最后用gdb一点一点看汇编，发现自己指针传输有问题，写bomb-lab的时候想“这辈子绝对用不上这玩意儿”，没想到打脸来得这么快。写lab还是一个痛苦痛苦痛苦并且快乐的过程，v2.5过的时候，半夜在宿舍欢呼，然后去全家吃的那一碗泡面的味道估计很长一段时间都会停留在我的记忆里。

一只可爱的猫猫

