姓名：谢志康

学号：22307110187

内容：lab3-coroutine\_lab

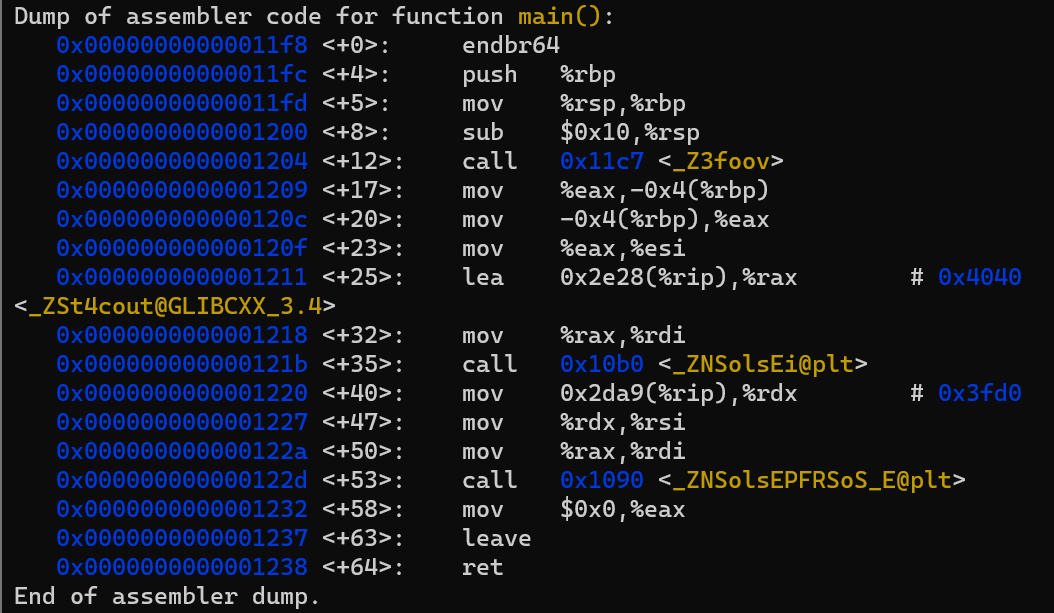
目录：part0, part1, part3, part4, part5, part2/2.5, 猫猫

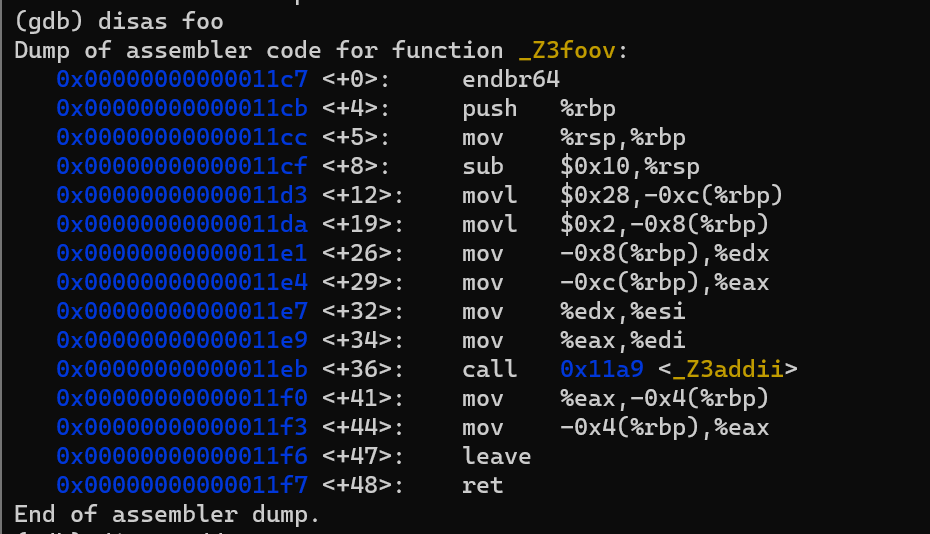
耗时：>60h

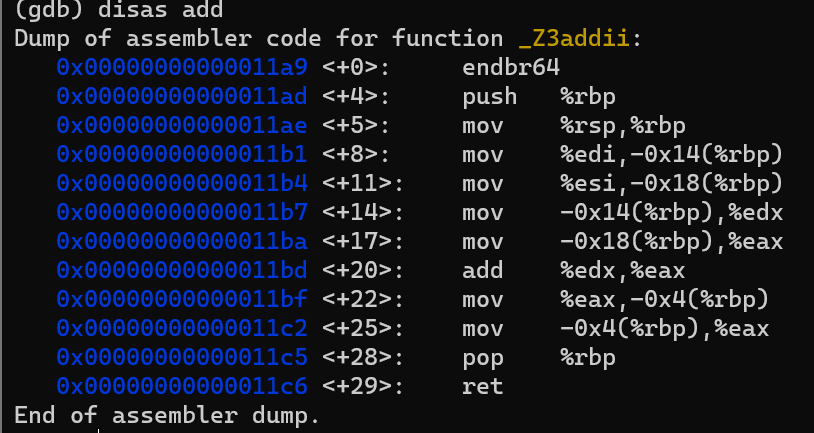
建议：希望以后尽量简单点（）要做破防了

Part0：

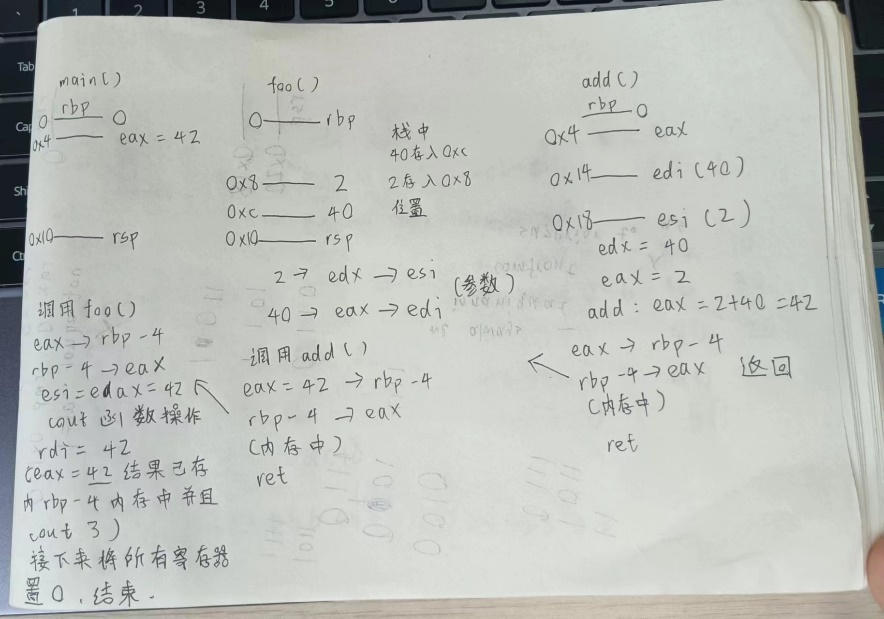
任务1：在linux中执行：g++ -g -o func func1.cpp （func1.cpp是cpp文件）之后gdb func Disas命令查看三个函数，截图如下：







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。

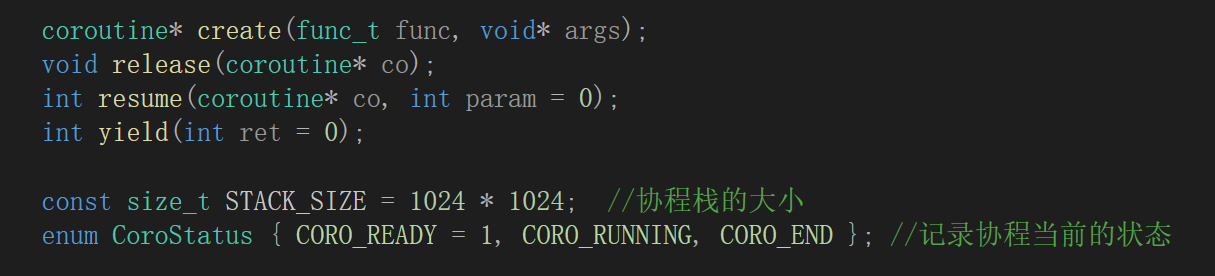
2.栈：栈用于存储函数调用的参数、局部变量以及函数的返回地址。在函数暂停时，需要保存栈的状态，使在恢复执行时能够正确返回到函数的上下文。

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。

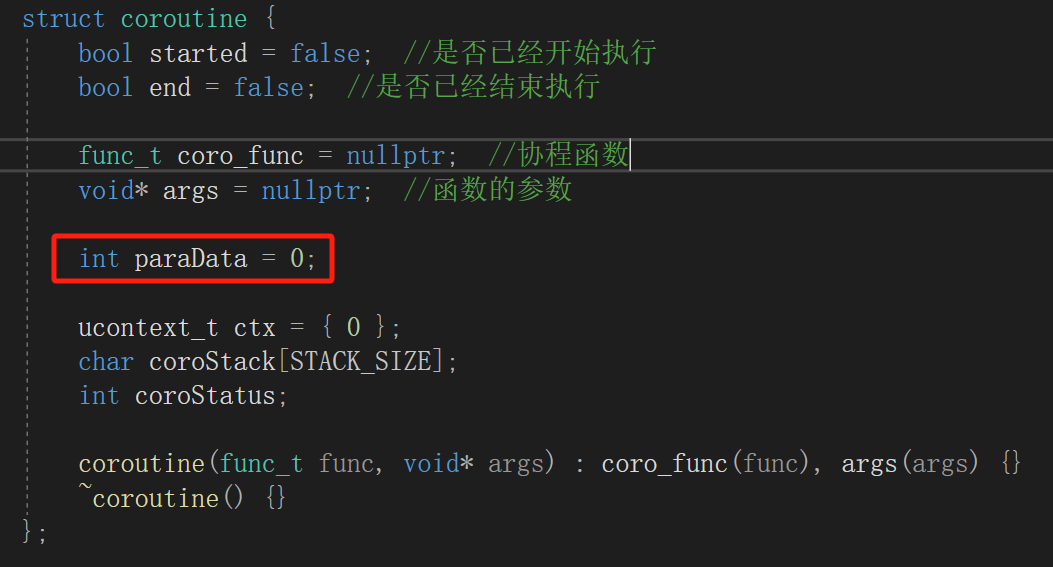
Part1：

1. 在coro.h中，补全coroutine结构体的实现。



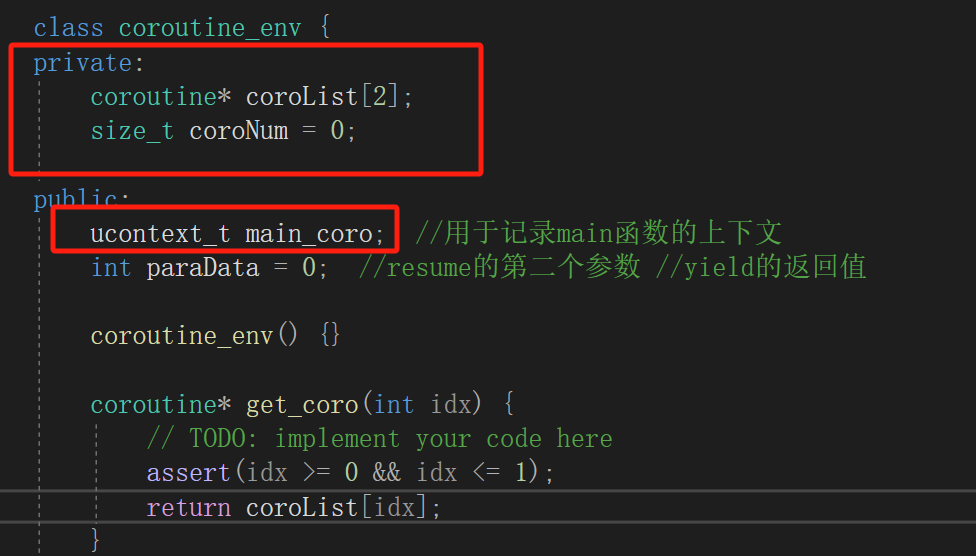
首先新开一个常量表示协程栈的大小，新开一个枚举类型变量记录一个协程的状态。

Create等四个函数在cpp中实现。



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））

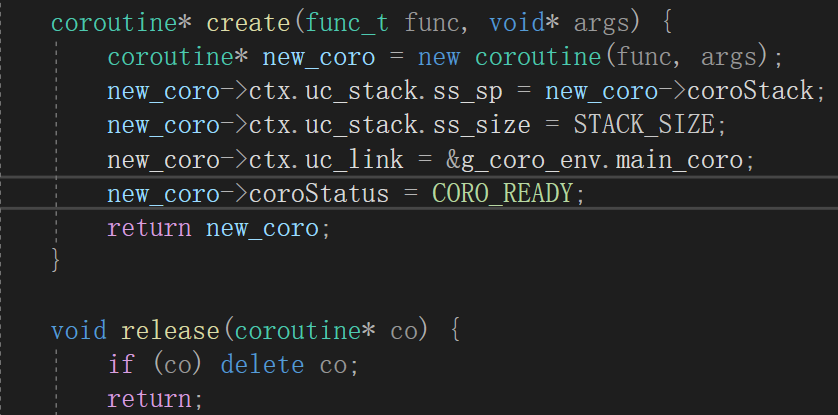
Env结构体用于存储协程的调用信息，“在目前的实验中，不会涉及到协程的递归调用，所以栈的深度可以只设置为2。” 所以开栈深度为2并给个指示记录当前存储的第几个协程。 Main\_coro用于记录main函数的上下文。 后面协程压栈出栈内容比较简单。



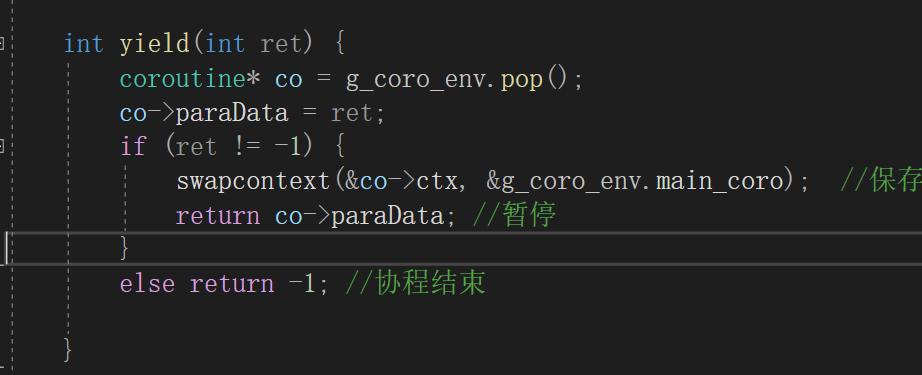
1. Coro.cpp的实现



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）



Yield

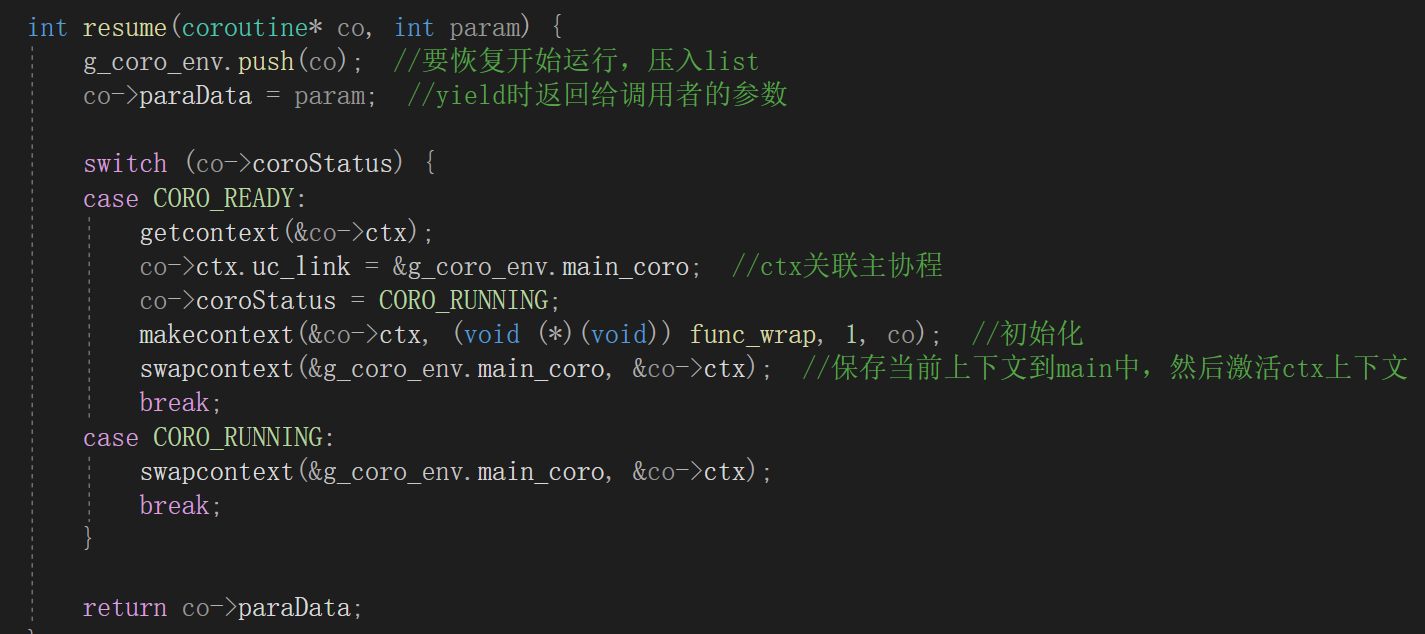


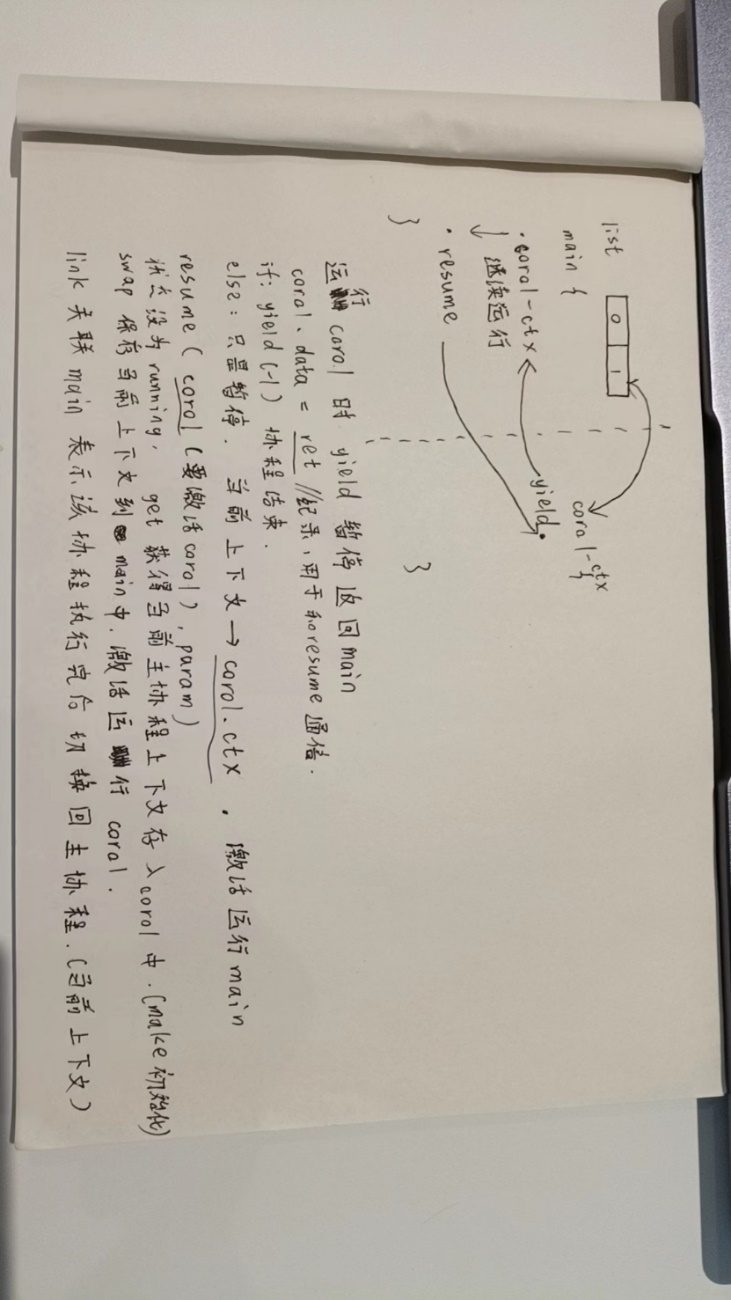
之前，协程每次开始运行了就将协程信息压入list内存储，所以list的顶部就是当前正在运行的协程，yield就是需要将这个正在运行的协程暂停同时激活主协程。

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。

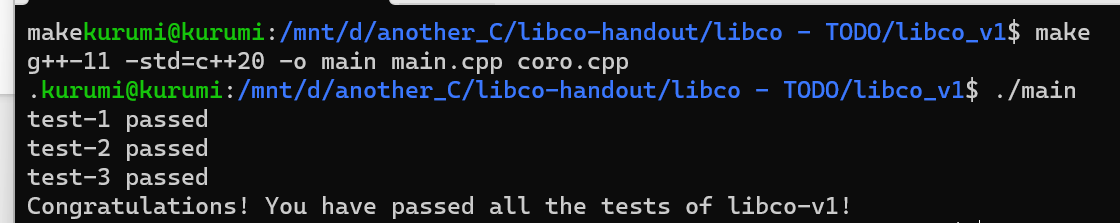
Resume（最麻烦）

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——





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（）。

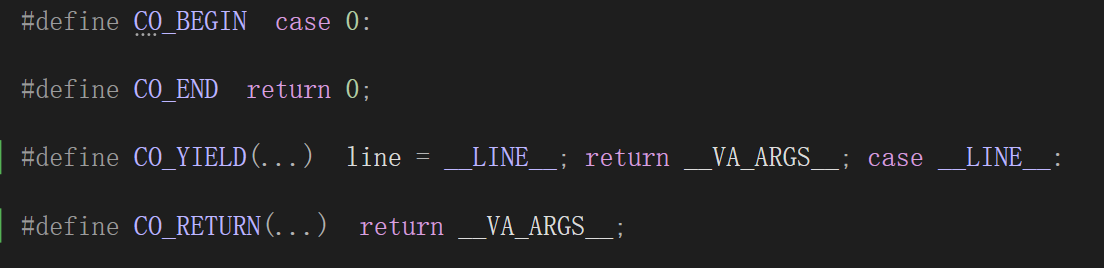


Part2，part2.5:

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。

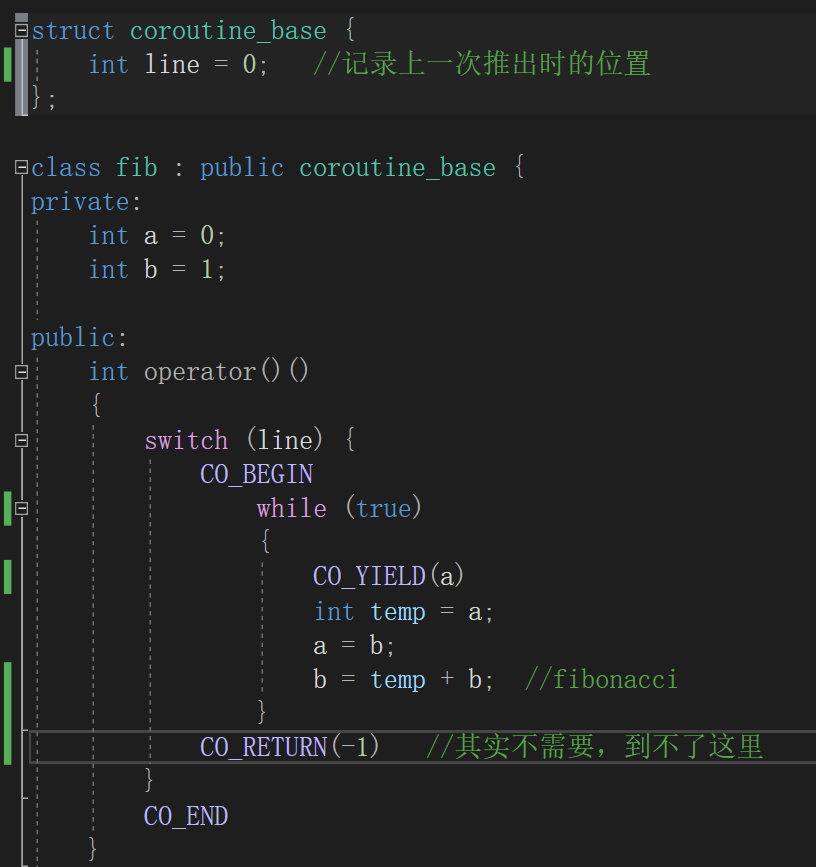
Part3:

这一part比较简单，基本看懂实验文档就能做了



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。



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。



Part4:

参考[C++ coroutine generator 实现笔记 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/590892907)学习

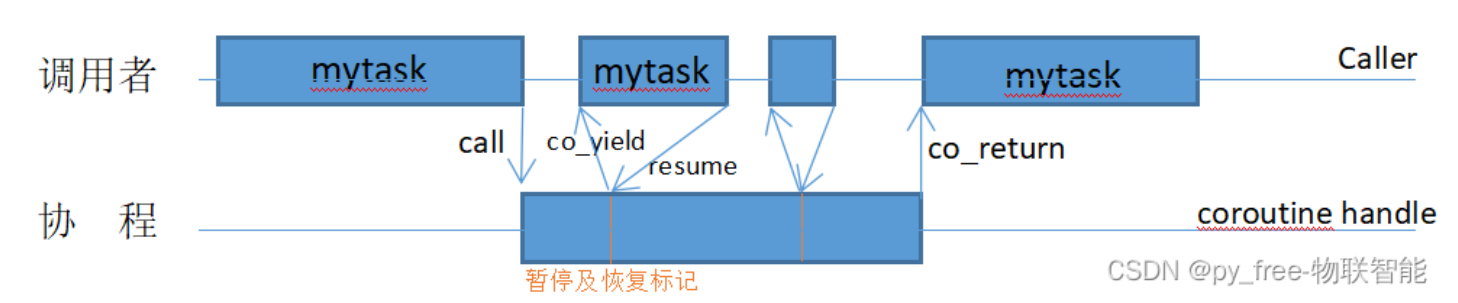
参考[C++语法糖(co\_await)详解以及示例代码 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/648821861)学习

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](https://blog.csdn.net/junxuezheng/article/details/129309912)学习

参考[C++ Coroutines: Understanding the Compiler Transform | Asymmetric Transfer (lewissbaker.github.io)](https://lewissbaker.github.io/2022/08/27/understanding-the-compiler-transform)学习

参考[Coroutines in C++20 - 简书 (jianshu.com)](https://www.jianshu.com/p/16e0a33a4f80)学习

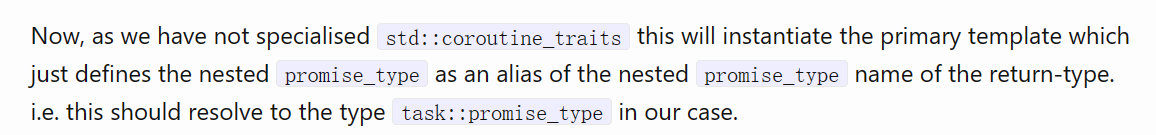
参考[C/C++开发，无可避免的多线程（篇三）.协程及其支持库\_c++协程库-CSDN博客](https://blog.csdn.net/py8105/article/details/129336892)学习



Answer：

1. 协程函数的返回值Coroutine Functor需要有哪些成员？

Promise\_type

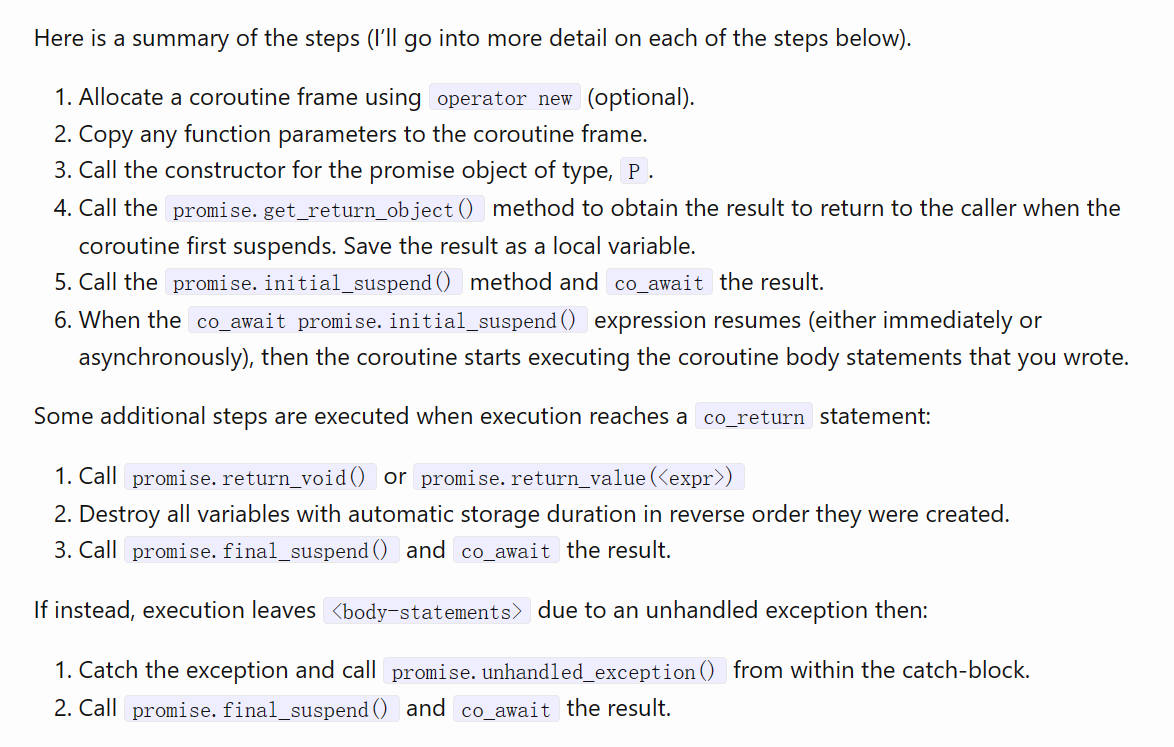


1. Promise对象需要提供哪些函数？

return\_void() return\_value(<expr>) final\_suspend()

get\_return\_object() unhandled\_exception() initial\_suspend()

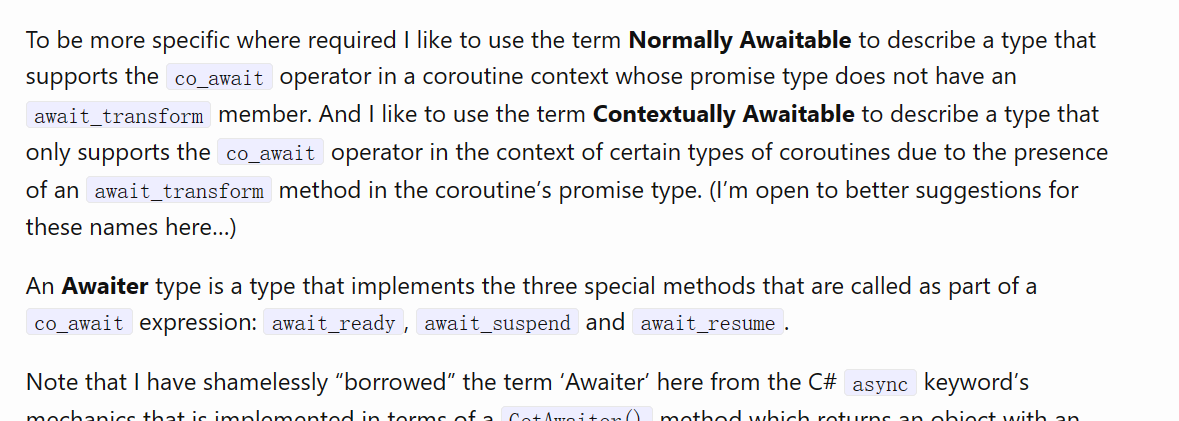
yield\_value(<expr>)



1. 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



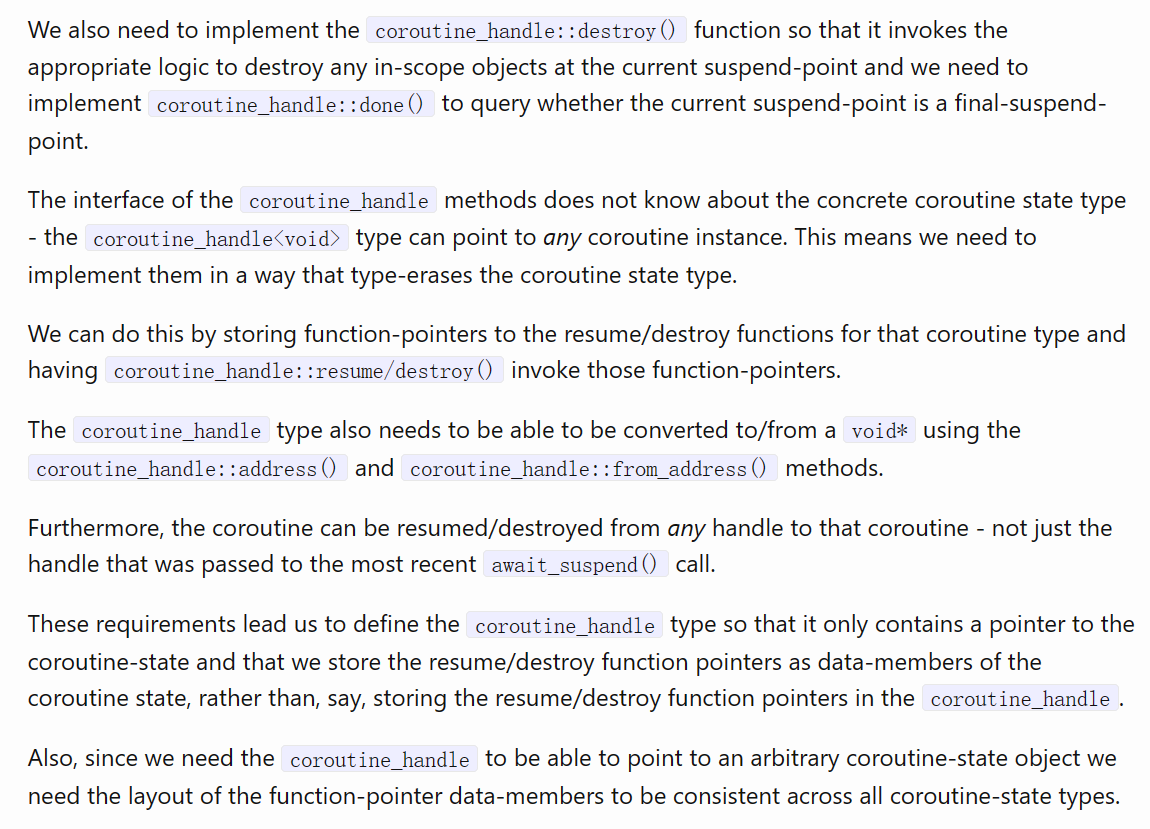
1. Coroutine handle通常需要提供哪些函数？

Coroutine\_handle:: 的namespace下：

Resume() destroy() done()

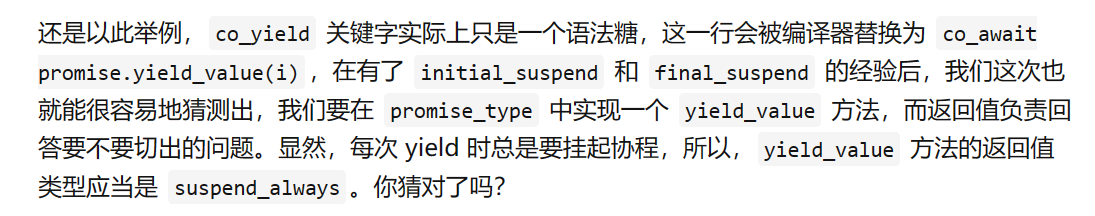
address() from\_address()

Promise() from\_promise() //关联generator内部的promise\_type



1. 为什么说co\_yield和co\_return是co\_await的语法糖？

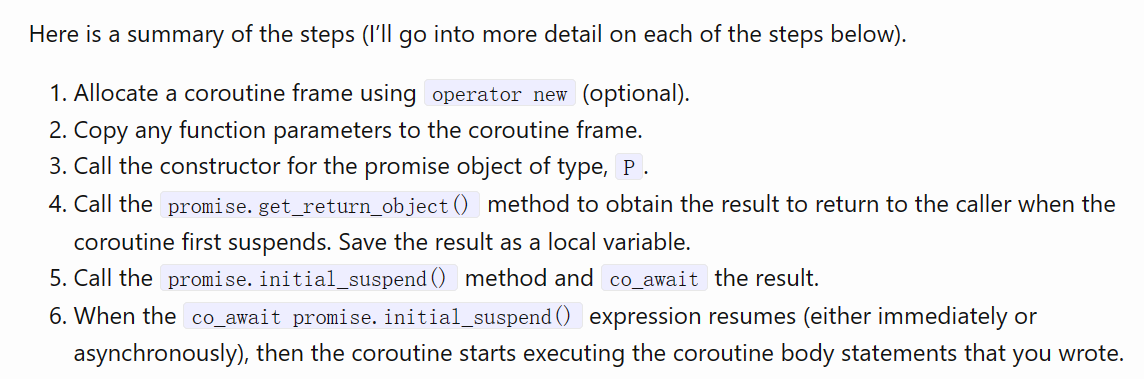
因为co\_yield实际上等价于co\_await promise.yield\_value(int value)

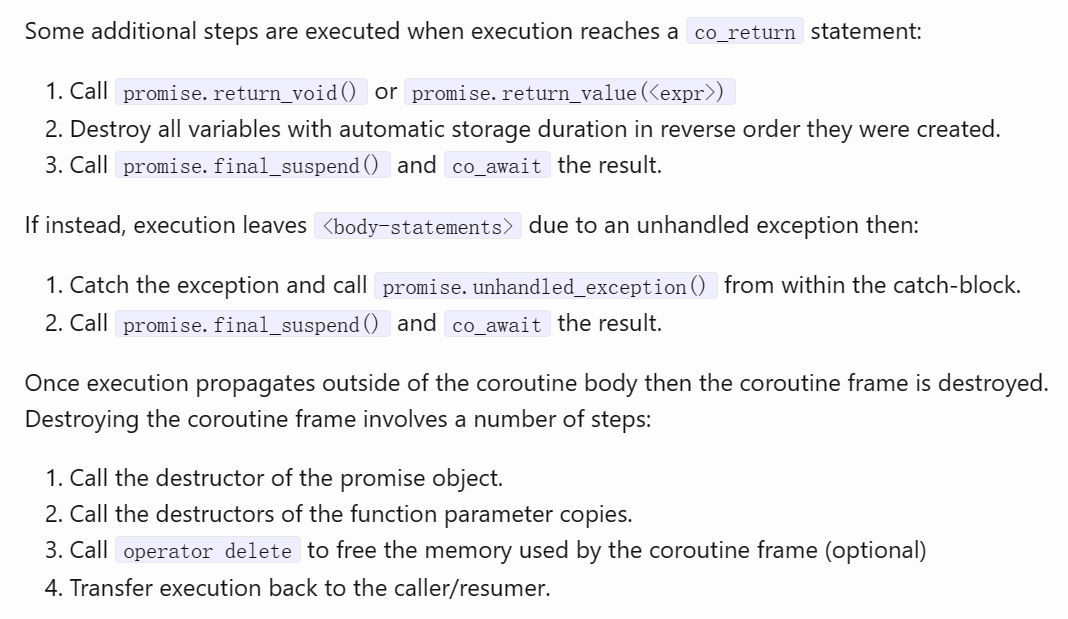


同理，co\_return实际上等价于co\_await promise.return\_value(int value)或者promise.return\_void() 且协程执行final\_suspend()（也就是执行suspend\_always（）当前协程被无条件切出去）。

1. 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。





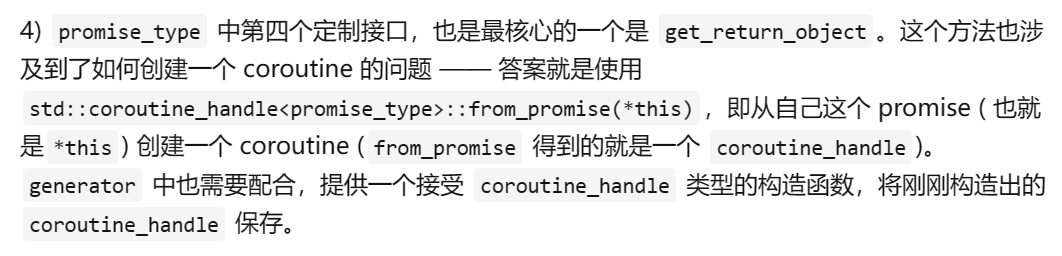
Initial\_suspend()和final\_suspend() 回答的是协程一出生时、协程最后一次时是否要挂起的问题。对于一个 generator 而言，这两个问题的回答是：初始时和最后一次始终都要挂起。因此，都直接使用suspend\_always（将他们无条件挂起）

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



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。

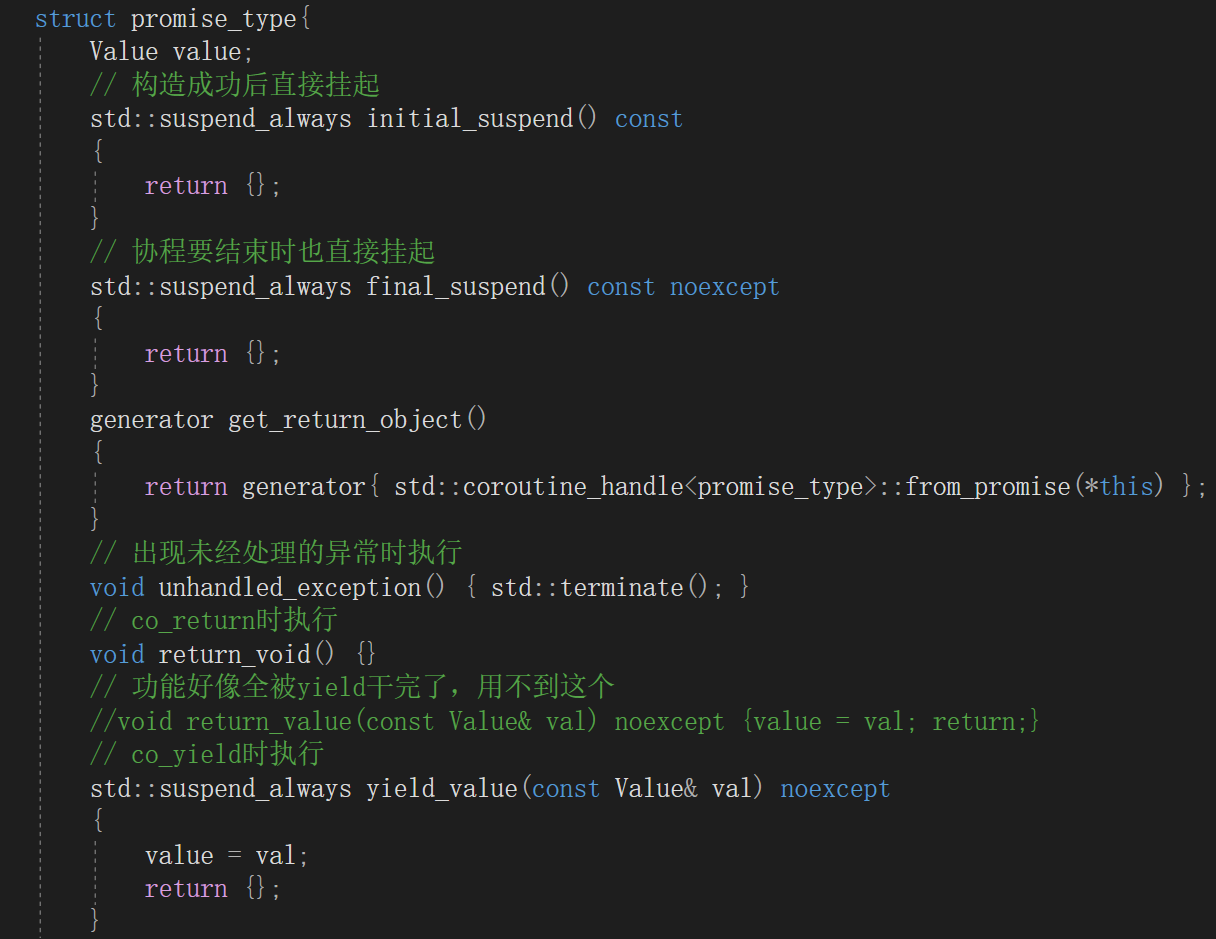
Destroy() 销毁一个协程。

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：

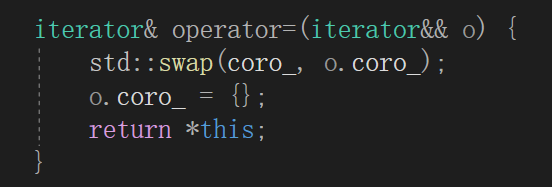
promise\_type就是承诺对象，承诺对象用于协程内外交流



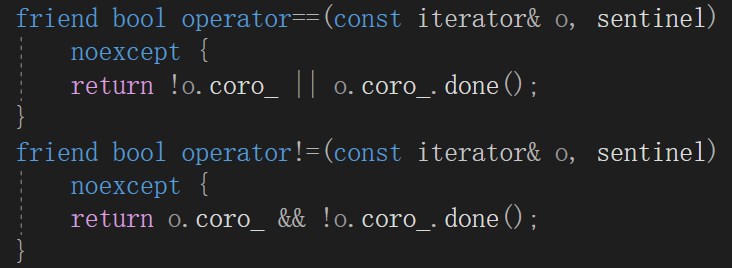
get\_return\_object，可以生成一个协程返回（借助from\_promise），会在协程正在运行前进行调用。

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。

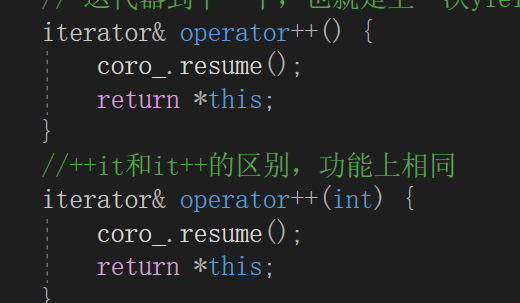


==和！=。这里==和！=都是将当前迭代器和sentinel进行比较，sentinel是边界哨兵，代表结束，也就是说，是为了判断当前迭代器是否终止。一种情况是要比较的迭代器根本不存在（null），另一种情况是该迭代器存在的，但是已经运行结束（使用done（））。 而不等（未终止）也就是==的取反。



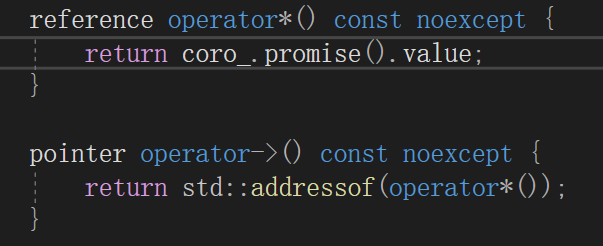
++和++（int）我的理解是++it和it++的区别。功能上是一样的（），除非写出it1 = it++和it1 = ++it这种会有区别，但是这里好像不用区分，只要简单实现功能就能过。

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。

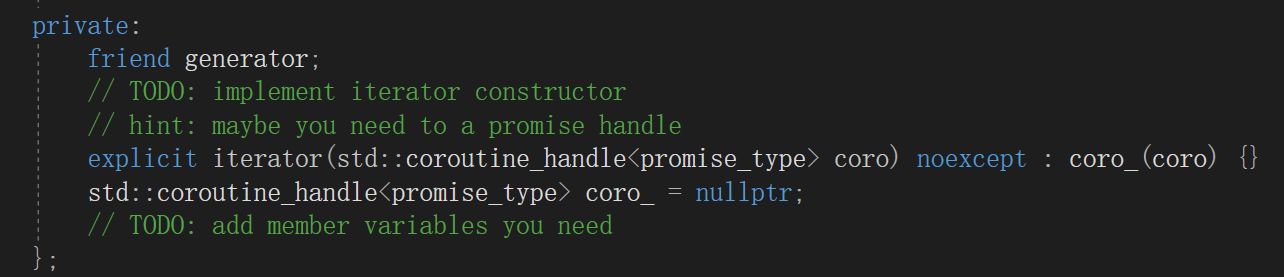


解引用操作，也就是对迭代器当前取值，使用promise()与协程内部交流，取出当前的value。

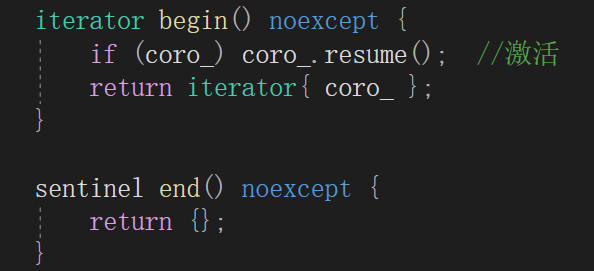
指向操作，利用addressof取地址即可。（获取一个对象的地址，与&功能相同）



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。



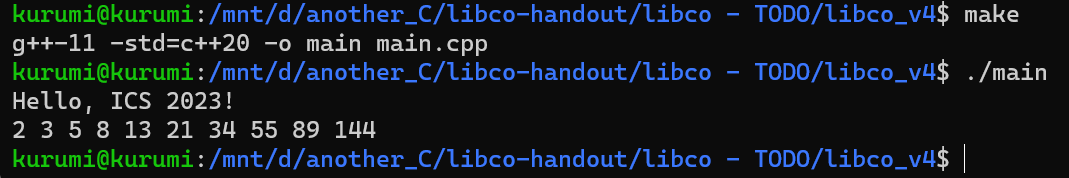
Begin（）和end（）的语法参考网上代码。功能上理解比较简单。



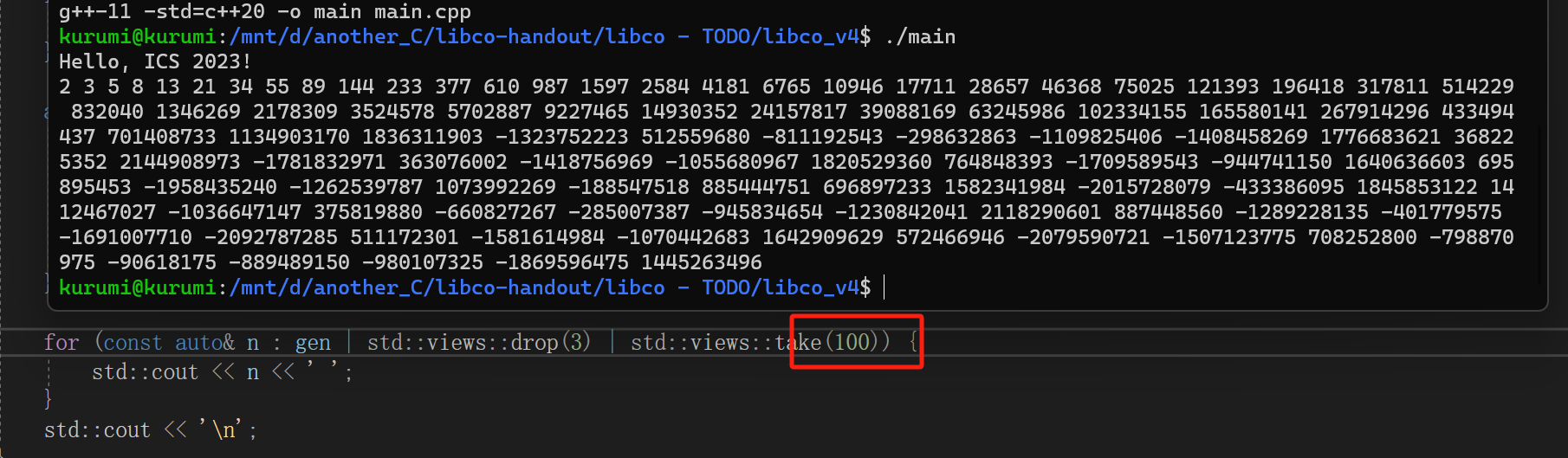
Main函数是构造了一个斐波那契数列，首先将每个元素存入创建的生成器 gen，之后忽略前三个（drop(3)）取出接下来的十个进行输出（take(10)）。

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。



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？



Part5：

参考[C++20 协程学习 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/571509492?utm_id=0)学习

参考<https://zhuanlan.zhihu.com/p/349710180>学习

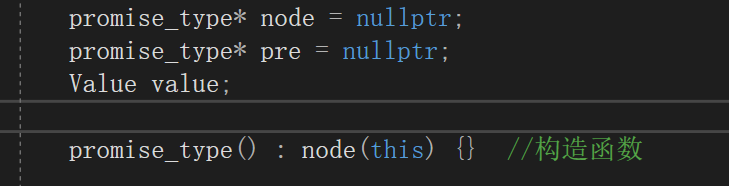
参考[C/C++开发，无可避免的多线程（篇三）.协程及其支持库\_c++协程库-CSDN博客](https://blog.csdn.net/py8105/article/details/129336892)学习

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

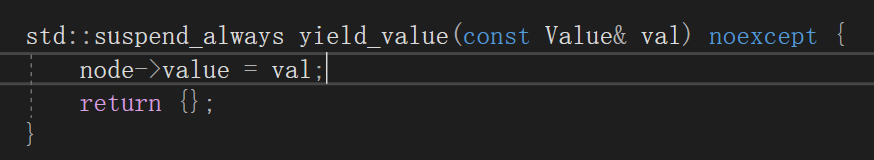
可递归相比于原本的part4多出了维护一个调用树。

先将part4的generator代码全部co过来



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。

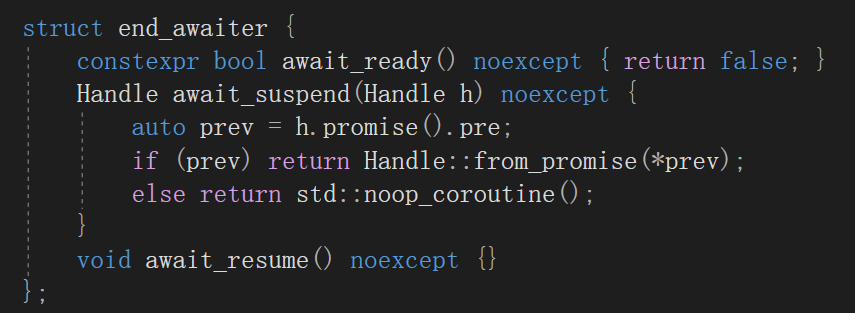


其余函数照搬part4即可，yield中注意要将赋值对象改成node所指的value（当前（this）对象中的value）。

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。

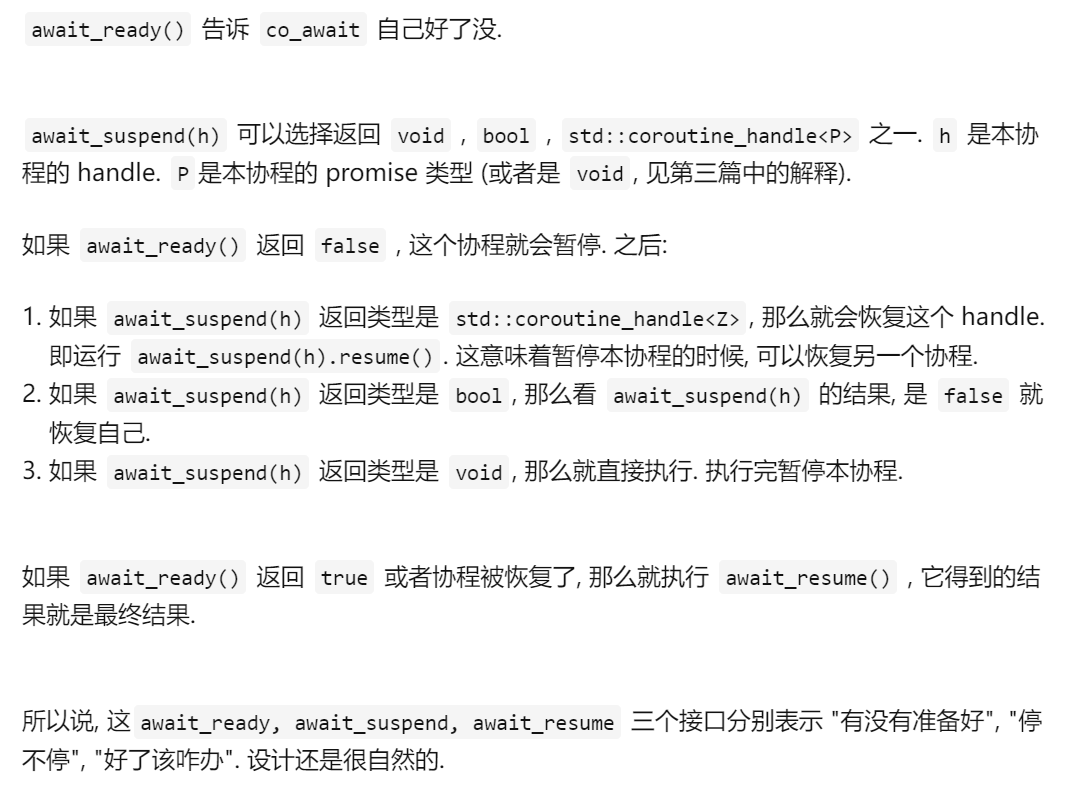
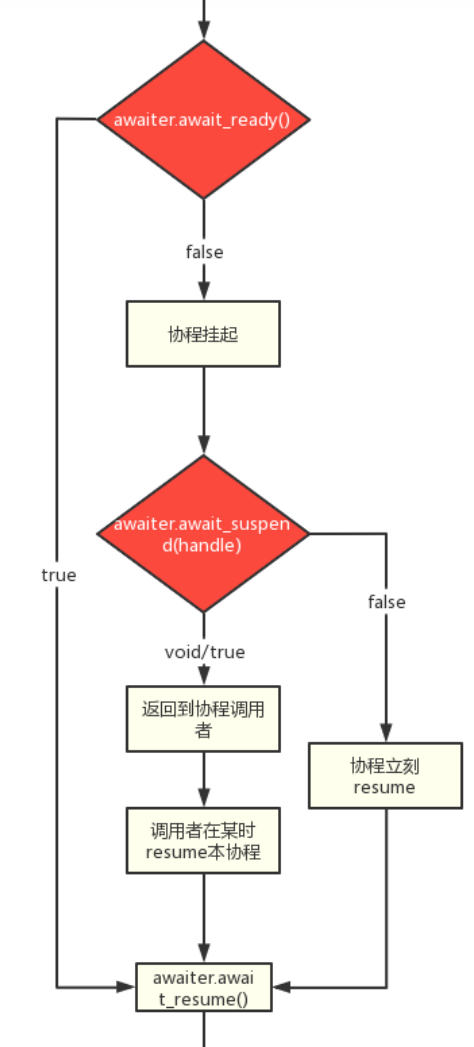
第一个结构体是做协程返回要做的操作：



（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。



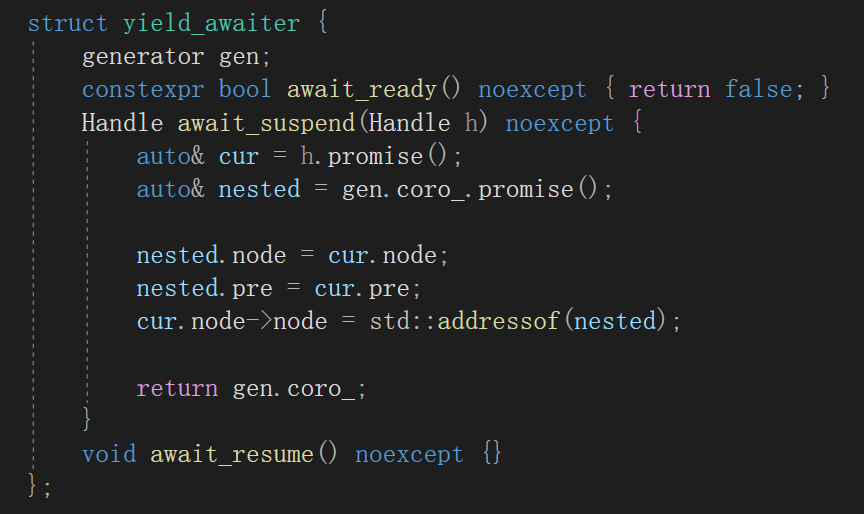
接下来是另一个结构体做协程向下递归运行要做的操作：

（参考了文章5的部分代码实现）

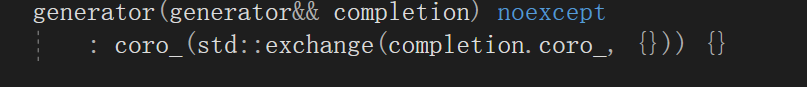
构造函数使用std的移动语义，相当于复制构造。

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）

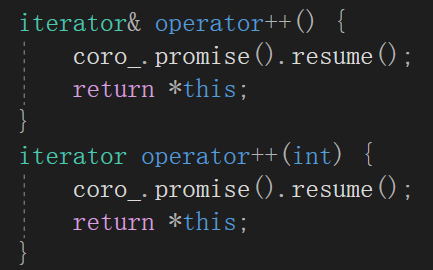


构造析构：特别的，移动构造，防止开新资源造成混乱：



（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）



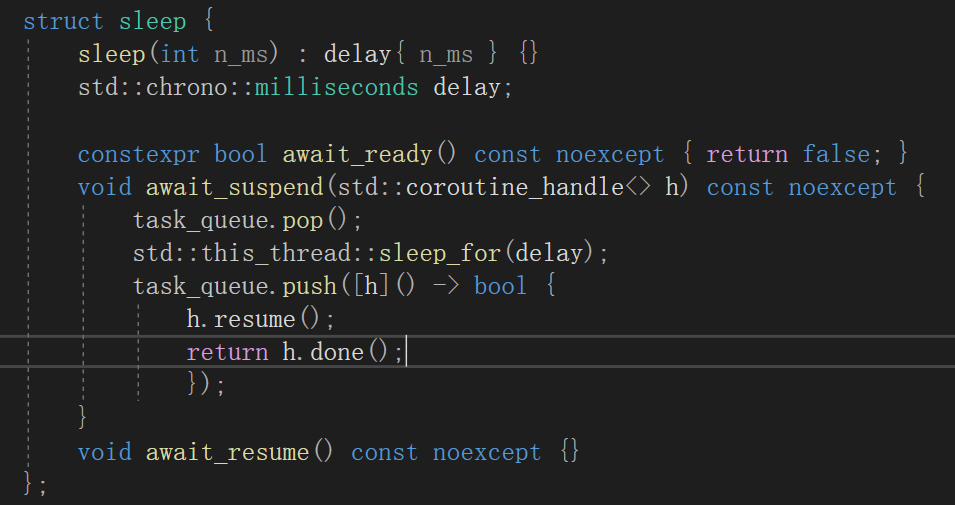
Sleep.h:

（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](https://blog.csdn.net/chengyq116/article/details/104587523/)

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](https://blog.csdn.net/m0_55875295/article/details/124679360)

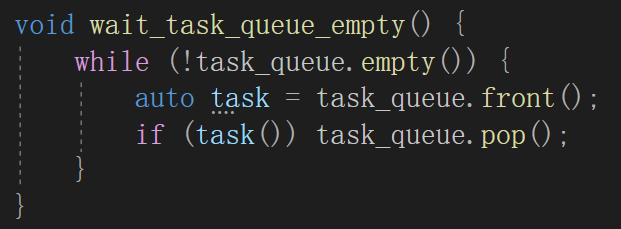
Sleep结构体代码如下：



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）

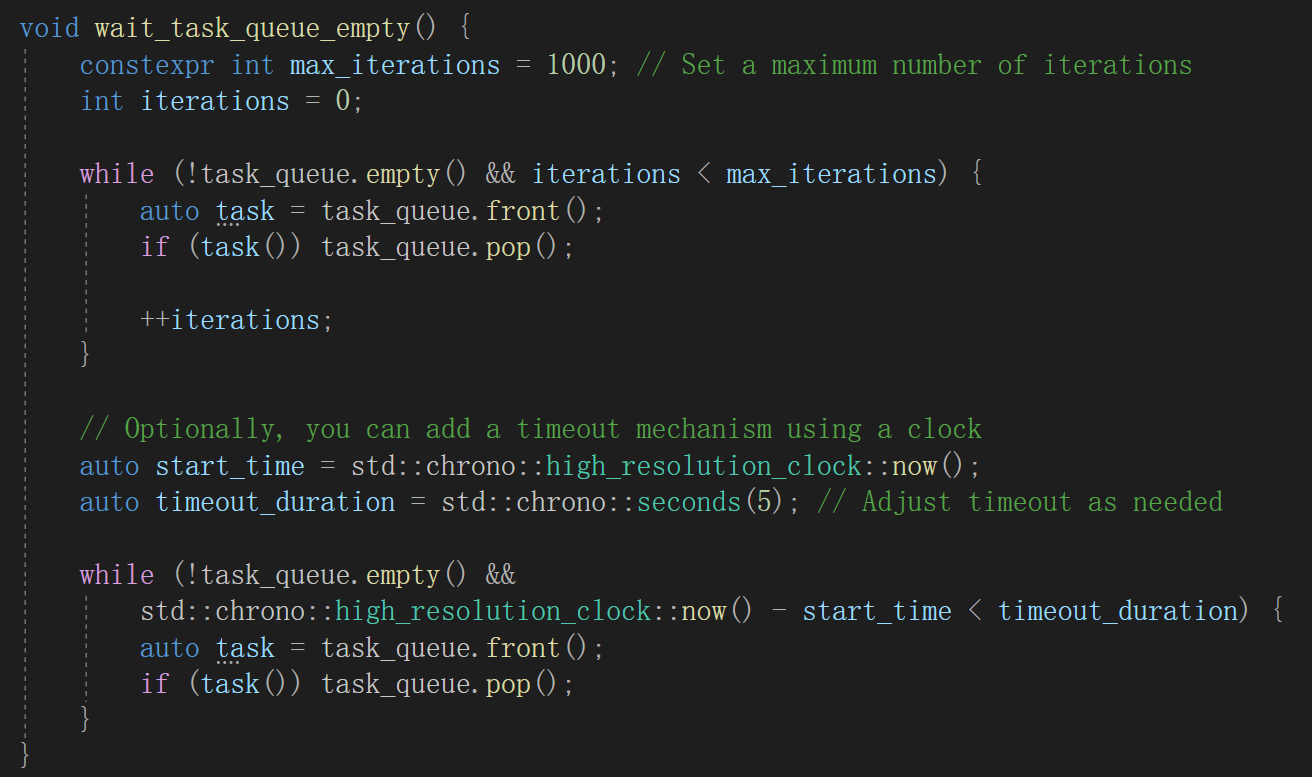


这部分有参考助教给的代码。功能上比较简单，将队列置空，即完成所有pending的工作

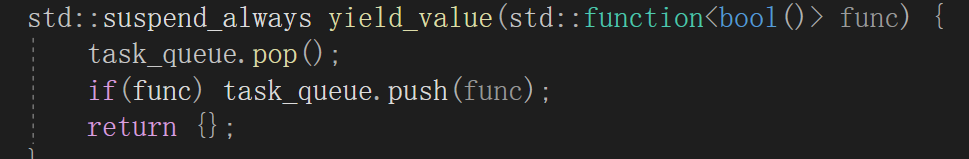
（我最开始担心这个会有死循环的风险，要是任意一个没有执行完，但好像没这问题，main函数里就是两个task交替执行输出，最后也确实都输完了，那按理说应该就是执行完毕，不存在not done的情况，事实也确实能跑（））

Attach：

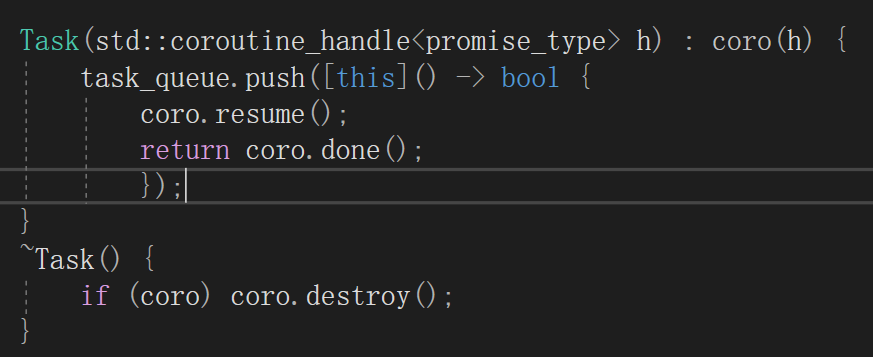
Gpt帮我写了一份避免infinite loop的代码，逻辑更加清晰，也能实现功能：



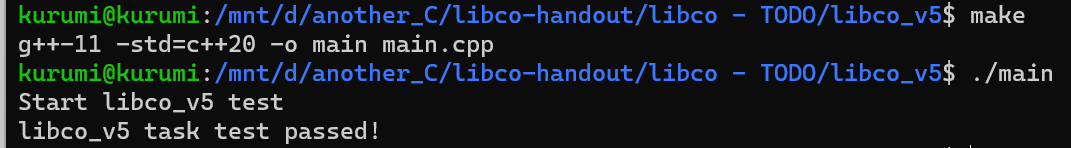
最后是Task类，promise\_type照搬generator，注意返回值的时候由于规则要suspend\_always，而没suspend我们就得扔掉一个当前执行的task（其实是暂挂），然后将给定的task加入队尾。



构造函数逻辑同理：



测试：



Part2 & part2.5:

参考[libco源码解析(1) 协程运行与基本结构-CSDN博客](https://blog.csdn.net/weixin_43705457/article/details/106863859)学习

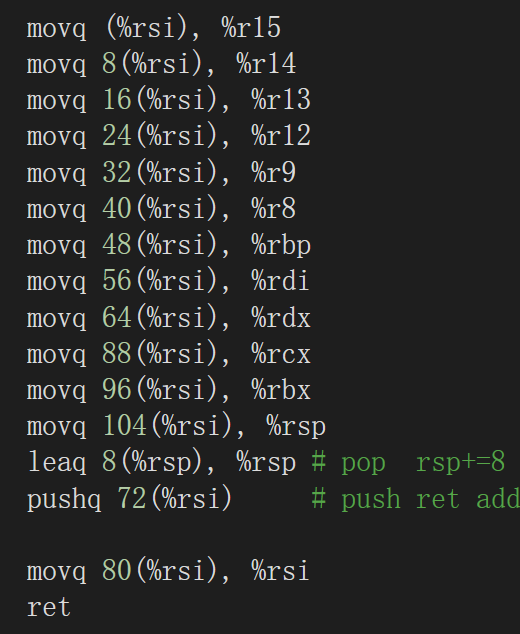
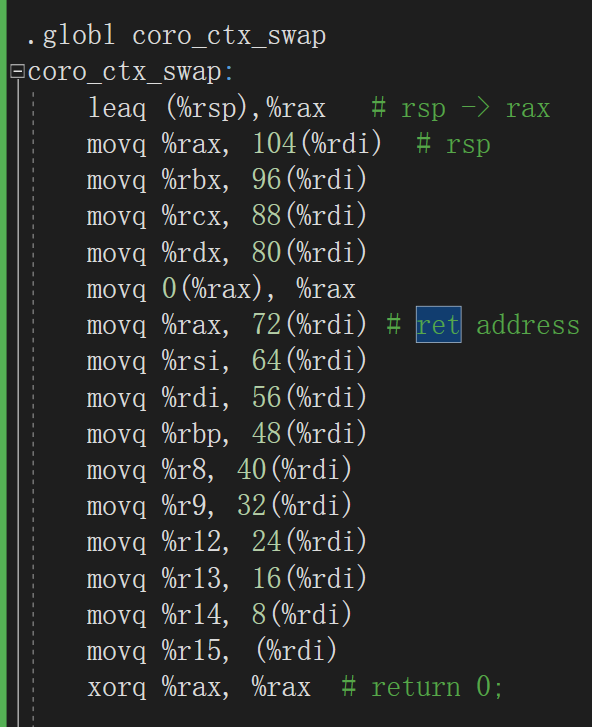
参考[libco源码解析(4) 协程切换，coctx\_make与coctx\_swap-CSDN博客](https://blog.csdn.net/weixin_43705457/article/details/106877644)学习

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）

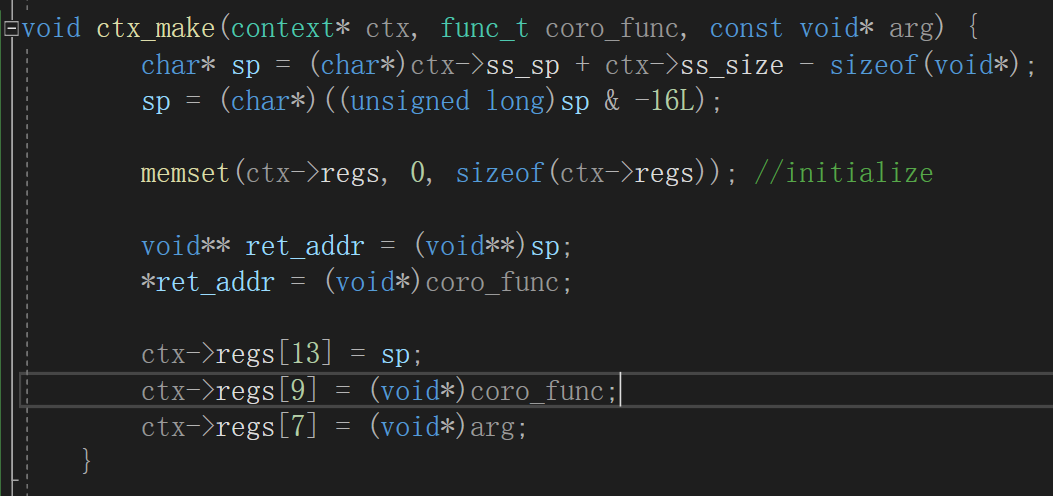


Rdi和rsi是函数的第一个和第二个参数，显然，这也就是执行协程上下文交换的功能。

1. 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](https://github.com/Tencent/libco/blob/master/coctx.cpp#L93))



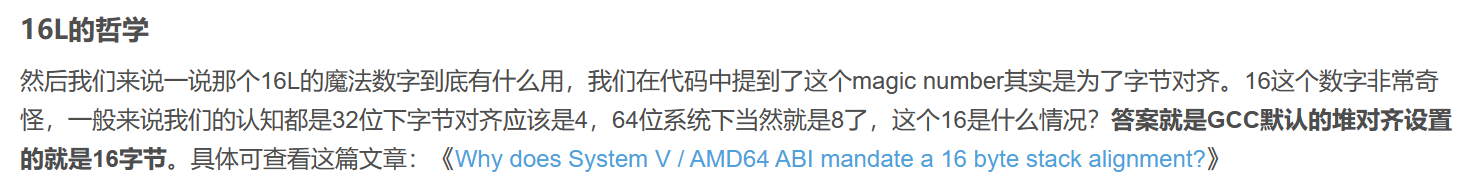
首先给出sp所指的地方，也就是esp指向的方向，ss\_size就是目前栈上剩余的空间，ctx->ss\_sp 对应的空间是在堆上分配的，地址是从低到高的增长，而堆栈是往低地址方向增长的。

*//------- 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）



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。

1. Coro\_ctx.h: （仅定义一些变量即可）

Coroutine.h部分：

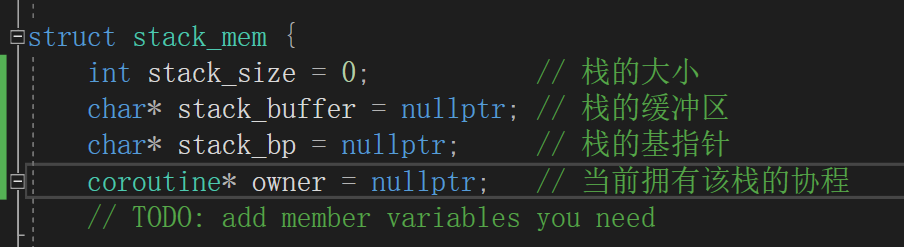
参考[libco/co\_routine.cpp at master · Tencent/libco · GitHub](https://github.com/Tencent/libco/blob/master/co_routine.cpp)代码学习

参考[微信著名libco协程库原理剖析-CSDN博客](https://blog.csdn.net/lianhunqianr1/article/details/127274878)学习

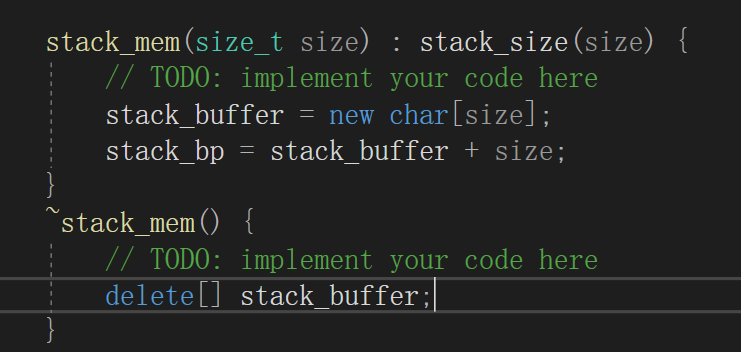
由于是共享栈，我们必须记录一下当前是谁在使用栈。

“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：



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）



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。”

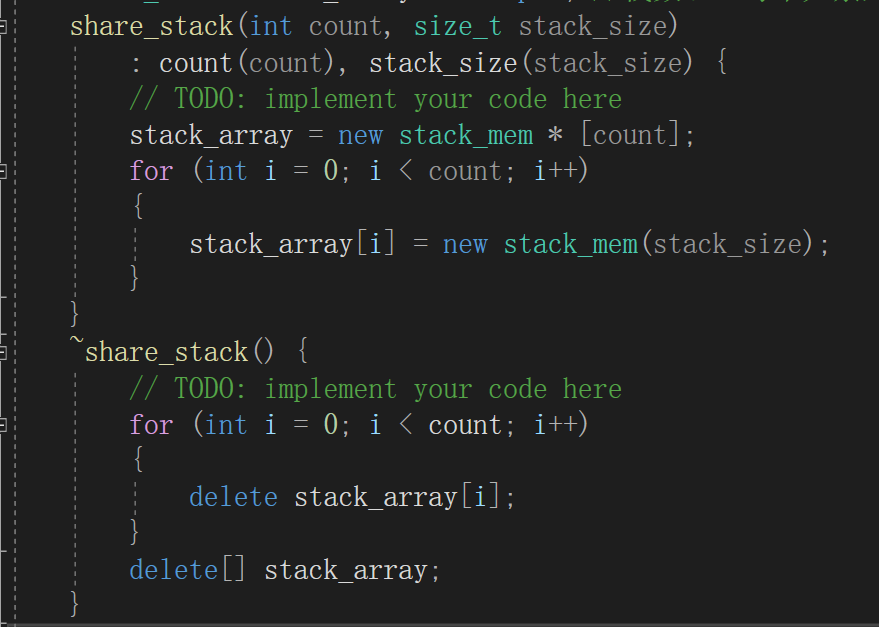
回答——“采用共享栈时，每个协程的栈从共享栈拷出时，需要分配空间存储，但按需分配空间。因为绝大部分协程的栈空间都远低于128K，因此拷出时只需分配很小的空间，相比私有栈能节省大量内存。共享栈可以只开一个，但为了避免频繁换入换出，一般开多个共享栈。每个共享栈可以申请大空间，降低栈溢出的风险。

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。“

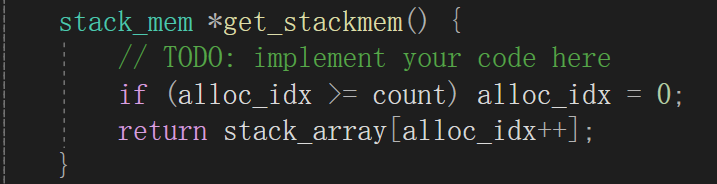
我们需要一个idx记录下次使用的栈的索引。模板中的count即表示有几个共享栈可以使用，每当idx大于count总数时，取模。



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。



得到目前使用的共享栈：



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)](https://www.zhihu.com/question/52193579/answer/2252618029)学习

参考[万字长文 | 漫谈libco协程设计及实现 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/73679393)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。

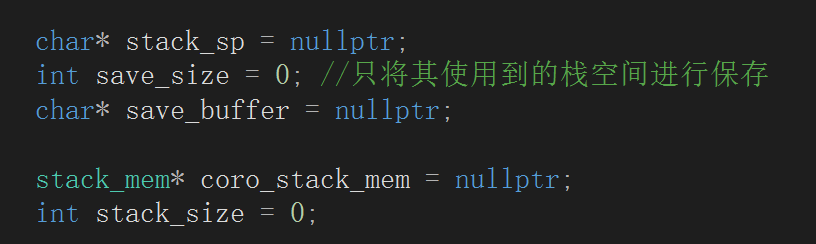
——pCallStack即我的call\_stack

“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：

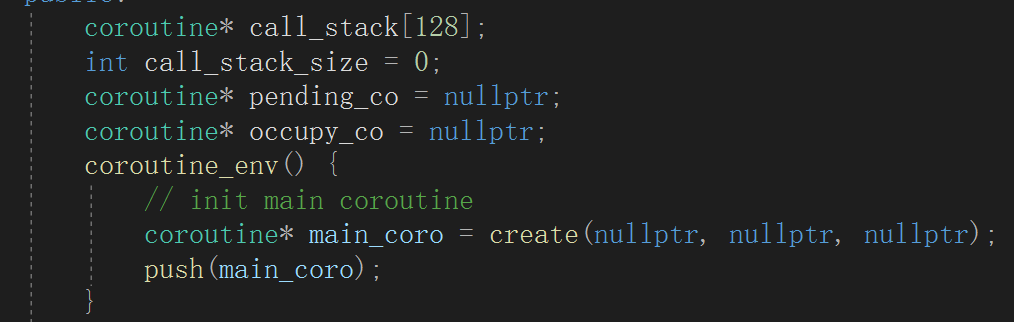
在coroutine结构体中，其余部分大致与part1相同，增加了：



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。



Push和pop模仿压栈出栈行为，对协程数组简单操作即可：



Coroutine.cpp部分：

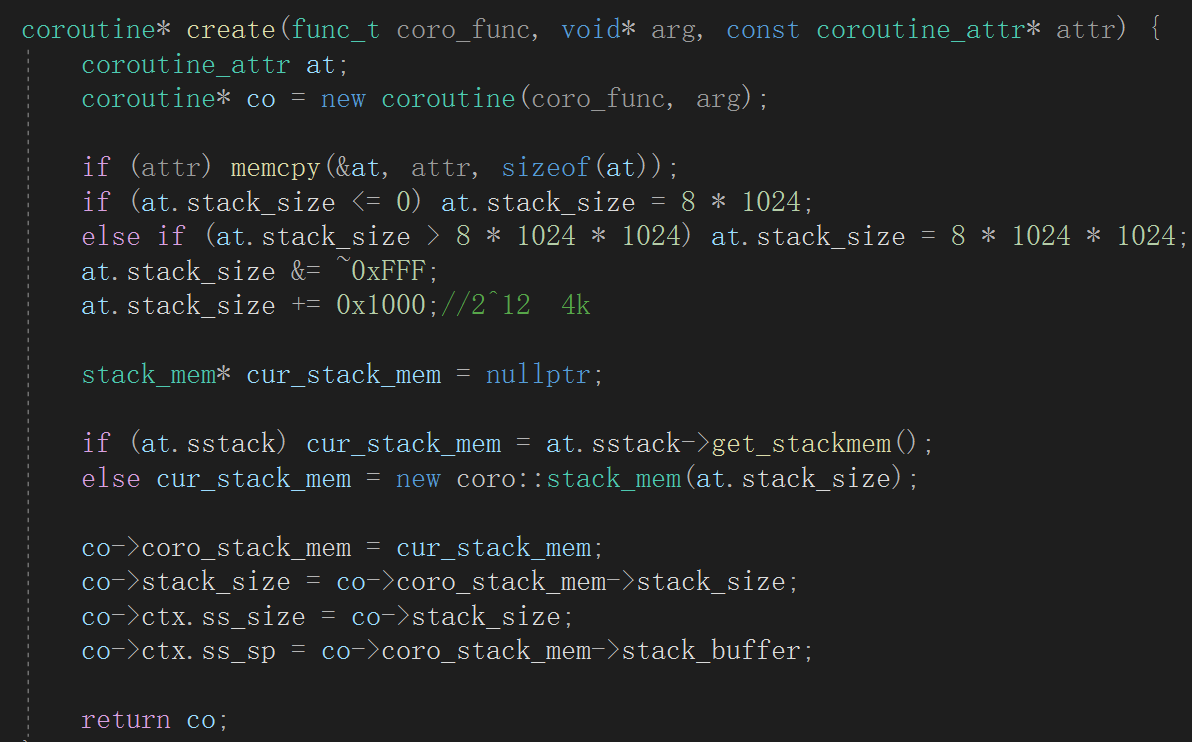
首先还是经典的create, release, resume, yield函数。新增的是将某新协程存入栈的操作和交换两协程（更换执行权，嵌套使用）的功能

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。

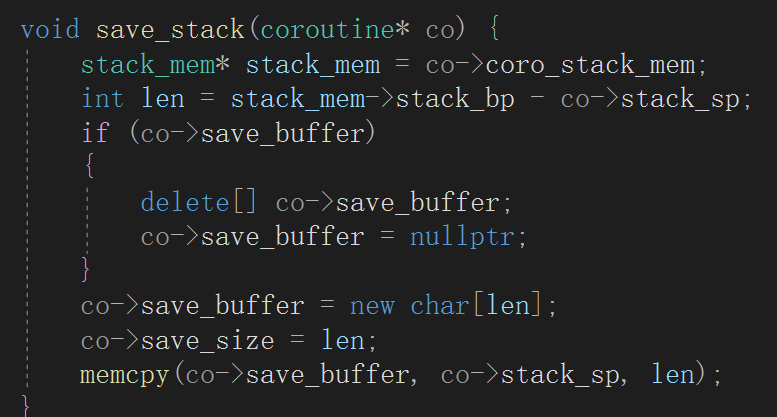
剩余代码也就是参考一下[libco/co\_routine.cpp at master · Tencent/libco · GitHub](https://github.com/Tencent/libco/blob/master/co_routine.cpp#L521)文章的co\_create\_env函数，将给定的参数的内容拷到我们新创建的协程中并返回，实现创建功能



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）



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。

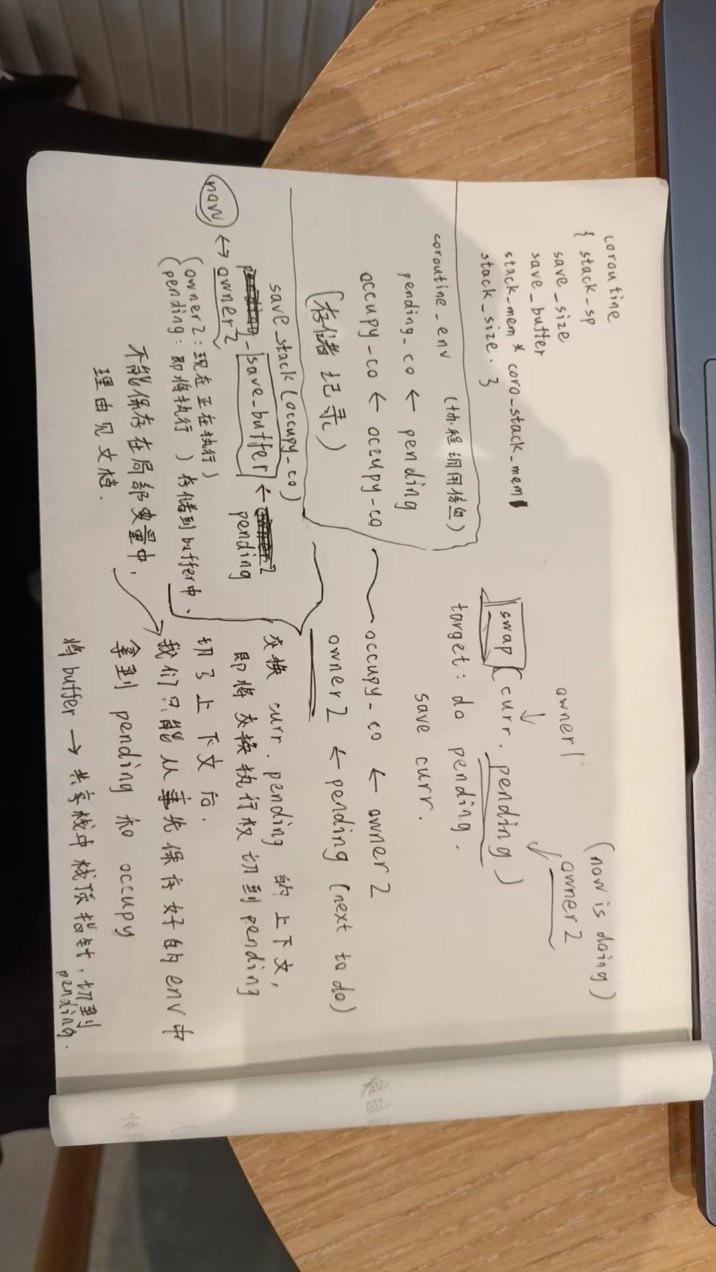
Occupy和pending记录在env里，以实现在协程栈切换后仍然能拿到这两个变量。

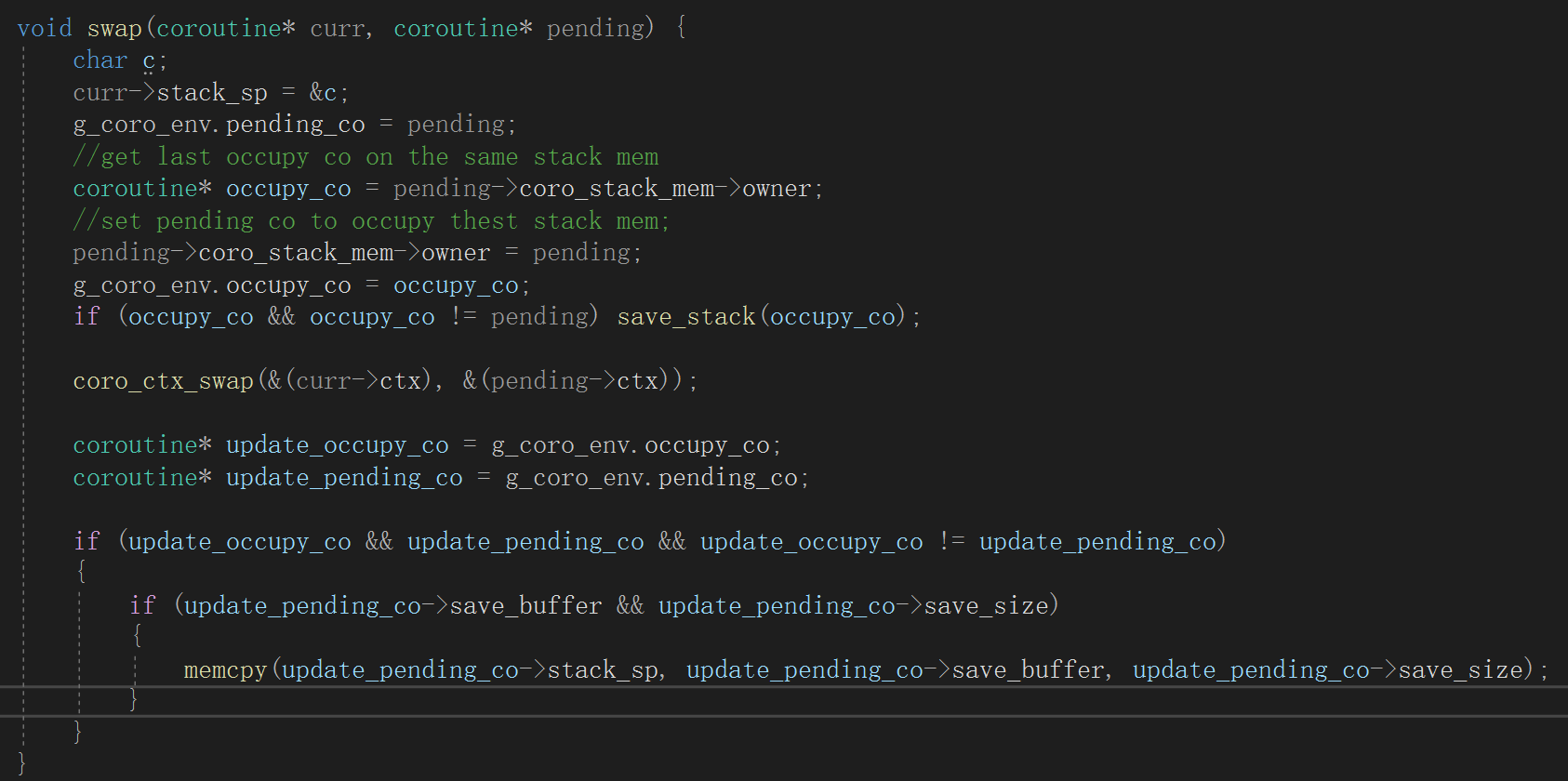
交换过程的存储重点通过env来保存，不能使用occupy的局部变量，否则切换上下文后无法再获取，没法更新共享栈切换协程。

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……

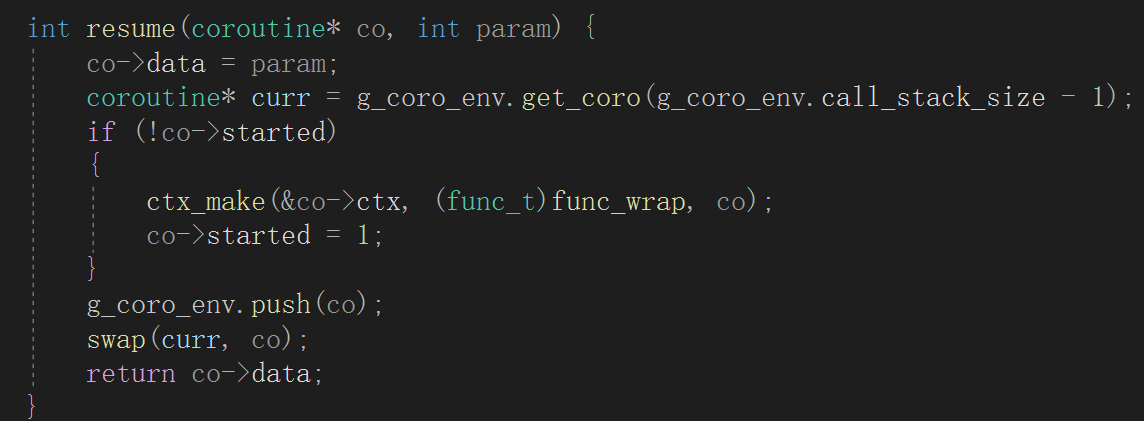
所以这一part大部分代码参考了libco的开源代码（）

过程简略示意图如下，代码如下：

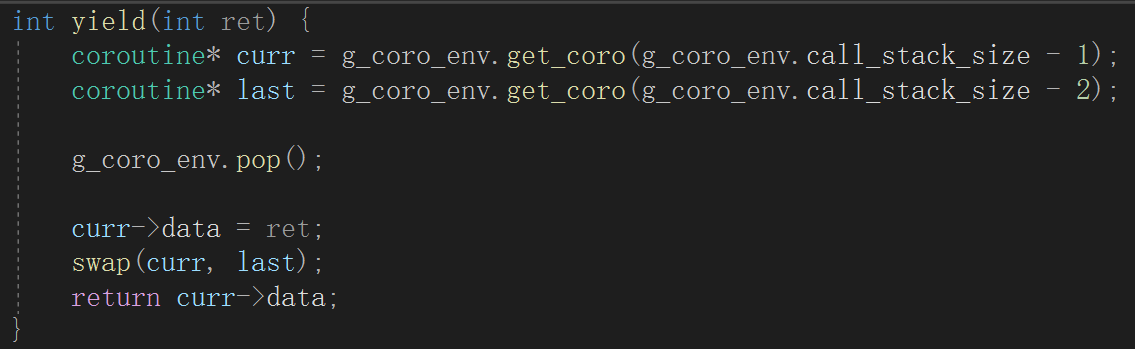




协程挂起和返回代码简单：



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。



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

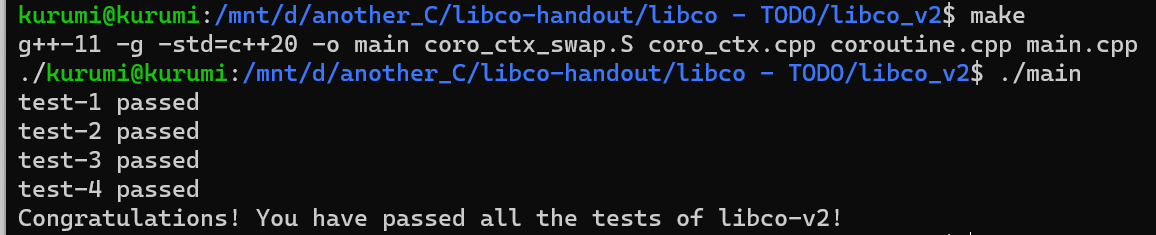
While循环：恢复co1，执行co1，恢复co2，执行co2……直到结束，测试了resume和h文件和swap的功能

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：



猫猫：

