

10 | Coroutines实战（二）：异步文件操作库

2023-02-06 卢誉声 来自北京

《现代C++20实战高手课》

课程介绍 >



讲述：卢誉声

时长 11:27 大小 10.46M



你好，我是卢誉声。

今天，我们继续上一讲的工作，实现基于协程调度的异步文件系统操作库。同时，在这一讲中，我们还要探讨一个重要话题，即实现所有调度线程全异步化的理想异步模型。

上一讲的最后，我们已经实现了任务调度模块，这意味着我们搭建好了基于协程的任务调度框架。但是，目前 `task` 模块是运行在主线程上的。因此，只有当主线程没有其他任务执行时，`task` 模块才会从消息循环中获取任务执行，并唤醒协程。

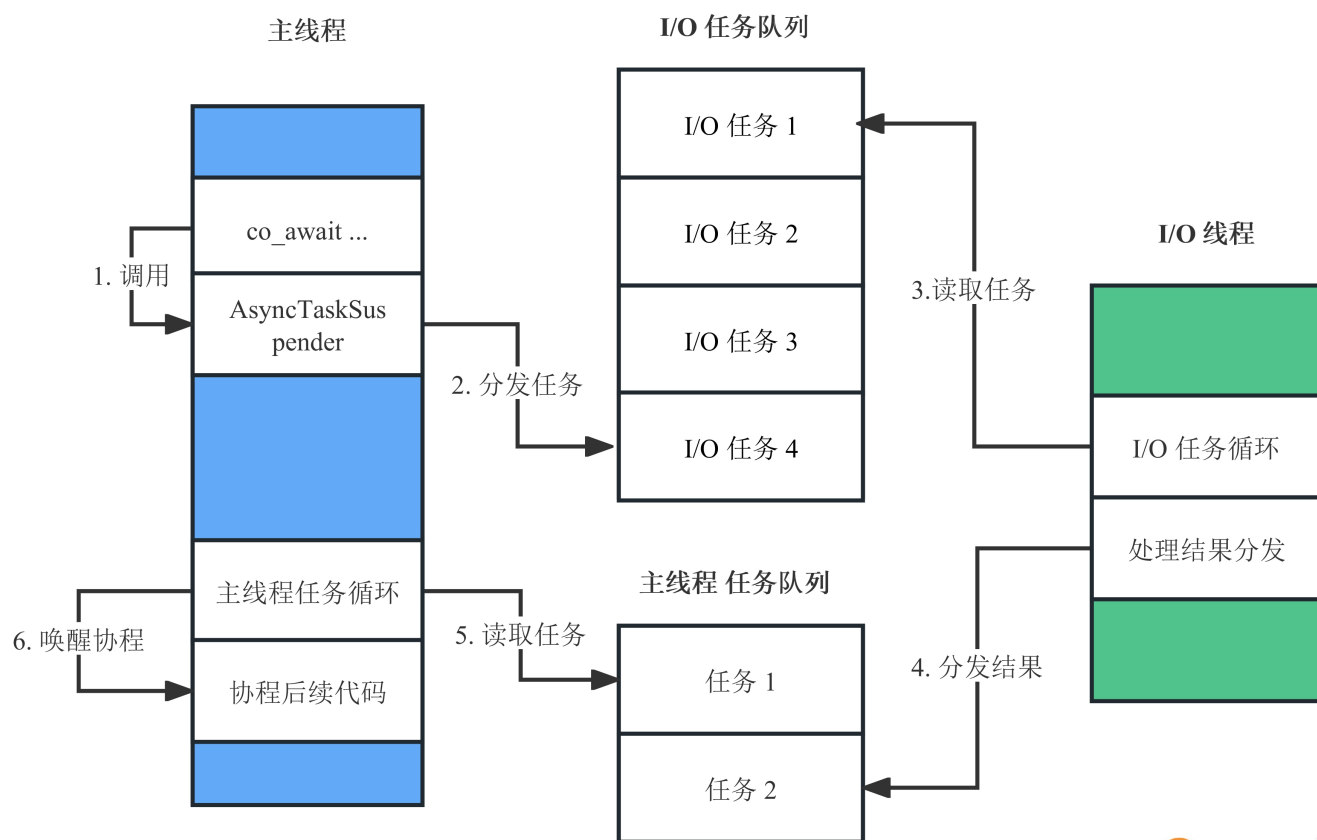
这不是一个理想的异步框架模型，我们更希望实现的是主线程和 I/O 调度全异步化。那么，这要如何实现呢？

项目的完整代码，你可以 [🔗 这里](#) 获取。

I/O 调度模块

其实，task 模块中预留的 `AsyncTaskSuspend` 函数，就是为了实现自定义任务的处理与唤醒机制。为此，我们继续讨论异步 I/O 的实现——基于 task 模块的任务调度框架，实现基于协程的异步 I/O 调度。

我们的基本思路是下图这样。



首先，我们要为 I/O 任务创建独立的任务队列。然后，`AsyncTaskSuspend` 中的主线程，负责将任务与协程的唤醒函数分发到 I/O 任务队列中。

接下来还要创建一个有独立任务循环的新线程，读取 I/O 任务队列，用于处理 I/O 任务。最后，处理完 I/O 任务后，将任务的返回值和协程唤醒函数分发到主线程的任务队列中。根据主线程的任务循环机制，当主线程空闲时，唤醒协程。

接下来，看一下这个思路的具体实现，我们从 task 分区的实现开始。

task 分区

第一步，我们来看看 io 模块的 task 分区 task/AsyncIoTask.cpp。该分区实现了 I/O 任务队列，后面是具体代码。

 复制代码

```
1  export module asyncpp.io:task;
2
3  import asyncpp.core;
4  import asyncpp.task;
5  import <functional>;
6  import <vector>;
7  import <mutex>;
8
9  namespace asyncpp::io {
10
11  export struct AsyncIoTask {
12      using ResumeHandler = std::function<void()>;
13      using TaskHandler = std::function<void()>;
14
15      // 协程唤醒函数
16      ResumeHandler resumeHandler;
17      // I/O任务函数
18      TaskHandler taskHandler;
19  };
20
21  export class AsyncIoTaskQueue {
22  public:
23      static AsyncIoTaskQueue& getInstance();
24
25      void enqueue(const AsyncIoTask& item) {
26          std::lock_guard<std::mutex> guard(_queueMutex);
27
28          _queue.push_back(item);
29      }
30
31      bool dequeue(AsyncIoTask* item) {
32          std::lock_guard<std::mutex> guard(_queueMutex);
33
34          if (_queue.size() == 0) {
35              return false;
36          }
37
38          *item = _queue.back();
39          _queue.pop_back();
40
41          return true;
42      }
43
44      size_t getSize() const {
45          return _queue.size();
46      }
```

```

47
48 private:
49     // I/O任务队列
50     std::vector<AsyncIoTask> _queue;
51     // I/O任务队列互斥锁，用于实现线程同步，确保队列操作的线程安全
52     std::mutex _queueMutex;
53 };
54
55 AsyncIoTaskQueue& AsyncIoTaskQueue::getInstance() {
56     static AsyncIoTaskQueue queue;
57
58     return queue;
59 }
60
61 }

```

在这段代码中，`AsyncIoTaskQueue` 的实现和 `AsyncTaskQueue` 类非常类似，不同之处就是 `AsyncIoTask` 的定义除了任务处理函数，还包含一个用于唤醒协程的处理函数 `resumeHandler`。

loop 分区

接下来，我们看一下 `io` 模块的 `loop` 分区 `task/AsyncIoLoop.cpp`。该分区定义了异步 I/O 循环的实现，代码如下。

 复制代码

```

1  export module asyncpp.io:loop;
2
3  import :task;
4  import asyncpp.task;
5
6  import <thread>;
7  import <chrono>;
8  import <thread>;
9  import <functional>;
10
11 namespace asyncpp::io {
12     export class AsyncIoLoop {
13     public:
14         static AsyncIoLoop& start();
15
16     private:
17         AsyncIoLoop() {
18             _thread = std::jthread(std::bind(&AsyncIoLoop::loopMain, this));
19         }
20
21         void loopExecution() {

```

```

22     AsyncIoTask opItem;
23     if (!AsyncIoTaskQueue::getInstance().dequeue(&opItem)) {
24         return;
25     }
26
27     opItem.taskHandler();
28
29     auto& asyncEventQueue = asyncpp::task::AsyncTaskQueue::getInstance(
30     asyncEventQueue.enqueue({
31         .handler = opItem.resumeHandler
32     }));
33 }
34
35 void loopMain() {
36     while (true) {
37         loopExecution();
38         std::this_thread::sleep_for(std::chrono::milliseconds(1000));
39     }
40 }
41
42 std::jthread _thread;
43 };
44
45 AsyncIoLoop& AsyncIoLoop::start() {
46     static AsyncIoLoop ioLoop;
47
48     return ioLoop;
49 }
50 }

```

在这段代码中，**AsyncIoLoop** 的主体实现和之前的 **AsyncTaskLoop** 非常类似，所以这里我们只讨论两个特别之处。

从代码里可以看到 **AsyncTaskLoop** 是直接在调用线程里执行的，而 **AsyncIoLoop** 类包含一个 **std::jthread** 对象（我们会在第十五讲中详细介绍 **jthread**）。构造函数中会创建线程对象，并将 **loopMain** 作为线程的入口函数，用于启动一个线程来处理消息循环。

另一个特别的地方是，在任务循环的处理中，**taskHandler** 执行结束之后，会将任务的 **resumeHandler** 添加到主线程 **AsyncTaskQueue** 中。根据主线程的任务循环机制，在主线程空闲之后，就会立刻执行 **resumeHandler** 唤醒协程。

asyncify 分区

接下来，看一下 **io** 模块的 **asyncify** 分区 **task/AsyncIoAsyncify.cpp**。代码实现如下。

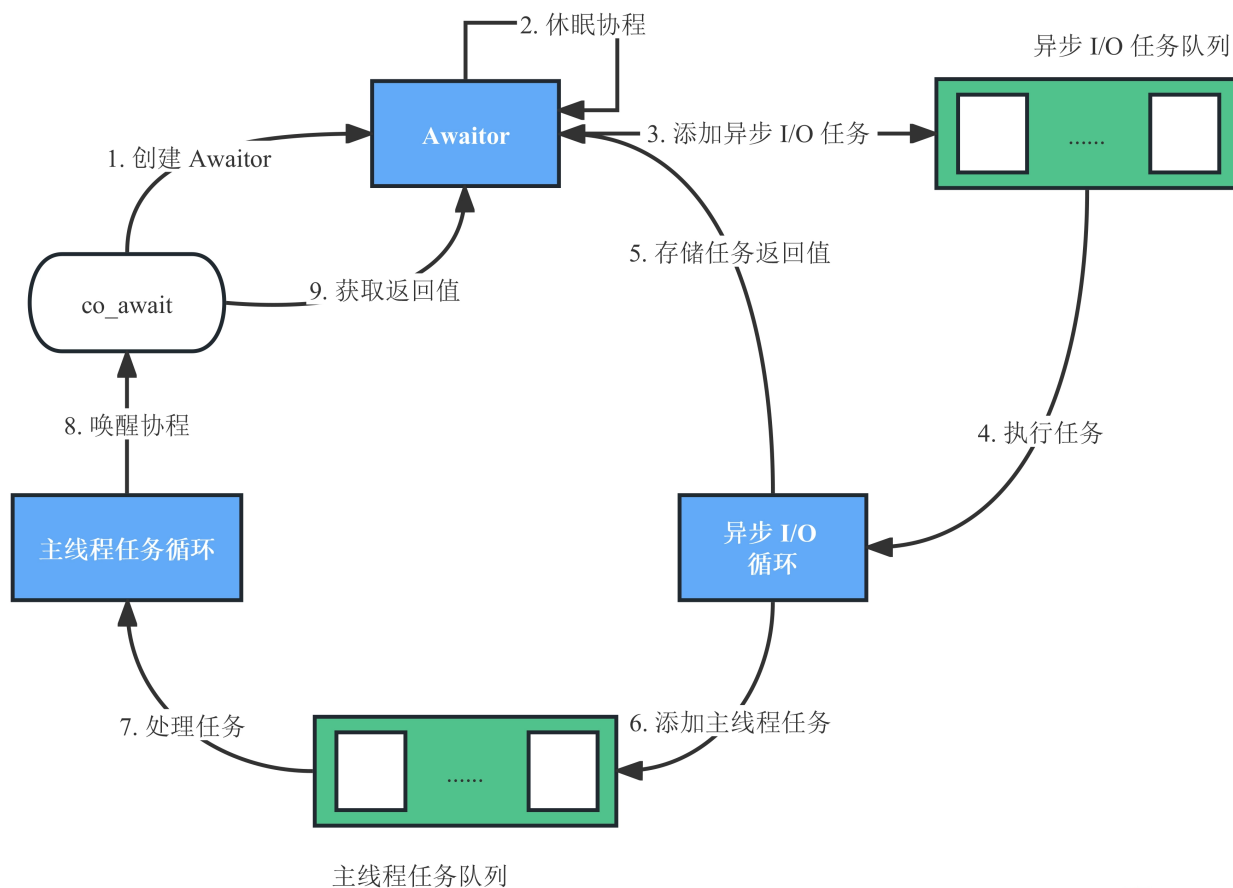
```
1 export module asyncpp.io:asyncify;
2
3 import <coroutine>;
4 import <type_traits>;
5 import asyncpp.core;
6 import asyncpp.task;
7 import :task;
8
9 namespace asyncpp::io {
10     using asyncpp::core::Invocable;
11     using asyncpp::task::Awaitable;
12     using asyncpp::task::AsyncTaskResumer;
13     using asyncpp::task::variantAsyncify;
14     using asyncpp::task::AsyncTaskSuspend;
15     using asyncpp::task::CoroutineHandle;
16
17     template <typename ResultType>
18     void ioAsyncAwaitableSuspend(
19         Awaitable<ResultType>* awaitable,
20         AsyncTaskResumer resumer,
21         CoroutineHandle& h
22     ) {
23         asyncpp::io::AsyncIoTask operationItem{
24             .resumeHandler = [h] {
25                 h.resume();
26             },
27             .taskHandler = [awaitable]() {
28                 awaitable->_taskResult = awaitable->_taskHandler();
29             }
30         };
31
32         asyncpp::io::AsyncIoTaskQueue::getInstance().enqueue(operationItem);
33     }
34
35     template <>
36     void ioAsyncAwaitableSuspend<void>(
37         Awaitable<void>* awaitable,
38         AsyncTaskResumer resumer,
39         CoroutineHandle& h
40     ) {
41         asyncpp::io::AsyncIoTask operationItem{
42             .resumeHandler = [h] {
43                 h.resume();
44             },
45             .taskHandler = [awaitable]() {
46                 awaitable->_taskHandler();
47             }
48         };
49
50         asyncpp::io::AsyncIoTaskQueue::getInstance().enqueue(operationItem);
51     }
```

```

52
53     export template <Invocable T>
54     auto ioAsyncify(T taskHandler) {
55         using ResultType = std::invoke_result_t<T>;
56
57         AsyncTaskSuspender<ResultType> suspender = ioAsyncAwaitableSuspend<Resu
58         return variantAsyncify(taskHandler, suspender);
59     }
60 }

```

在这段代码中，我们调用 `asyncpp.task` 中的 `asyncify`，将用户传递的 `taskHandler` 作为任务处理函数，将 `ioAsyncAwaitableSuspend` 作为 `suspend` 处理函数，这样我们就可以实现后面这样的异步 I/O 处理流程。



1. 在 `co_await` 时将当前协程休眠，并在异步 I/O 任务队列中添加一个任务。
2. 异步 I/O 任务循环获取任务，处理任务后将任务的返回值记录到 `Awaiter` 对象中，并将协程唤醒作为任务函数，添加到主线程的任务队列中。
3. 主线程的任务循环在空闲时获取异步 I/O 任务的协程唤醒任务，执行后唤醒休眠的协程。

4. 休眠的协程被唤醒后，通过 `co_await` 和 `Awaiter` 对象获取到任务处理的返回结果，协程继续执行。

这样，我们就可以在协程中实现 I/O 任务处理的异步化，同时也屏蔽了所有的实现细节。用户可以简单地将普通函数变为支持在协程中异步执行的函数。

多么美妙啊！我们在几乎不增加任何运行时开销的前提下，通过协程实现了异步 I/O 的异步处理与任务调度。

文件系统模块

在完成任务调度模块和 I/O 调度模块后，我们来简单看一下文件系统操作模块 `fs/FileSystem.cpp`。代码是后面这样。

 复制代码

```
1  export module asyncpp.fs;
2
3  import asyncpp.io;
4  import <string>;
5  import <filesystem>;
6  import <functional>;
7  import <iostream>;
8
9  namespace asyncpp::fs {
10     using asyncpp::io::ioAsyncify;
11     namespace fs = std::filesystem;
12
13     export auto createDirectories(const std::string& directoryPath) {
14         return ioAsyncify([directoryPath]() {
15             return fs::create_directories(directoryPath);
16         });
17     }
18
19     export auto exists(const std::string& directoryPath) {
20         return ioAsyncify([directoryPath]() {
21             return fs::exists(directoryPath);
22         });
23     }
24
25     export auto removeAll(const std::string& directoryPath) {
26         return ioAsyncify([directoryPath]() {
27             return fs::remove_all(directoryPath);
28         });
29     }
30 }
```


这段代码的封装方法非常简单，调用 `ioAsyncify` 将一个普通函数转换成“可以通过 `co_await` 调用”的异步任务函数，这跟 ES6 中的 `promisify` 一样简单！

调用示例

现在我们终于实现了所有关键模块。最后，我们来看看如何定义协程，并在协程中使用我们封装的函数。

我们还是对照代码来理解。

 复制代码

```
1  import asyncpp.core;
2  import asyncpp.task;
3  import asyncpp.io;
4  import asyncpp.fs;
5
6  #include <iostream>
7
8  using asyncpp::task::asyncify;
9  using asyncpp::task::AsyncTaskLoop;
10 using asyncpp::task::Coroutine;
11
12 using asyncpp::fs::createDirectories;
13 using asyncpp::fs::exists;
14 using asyncpp::fs::removeAll;
15 using asyncpp::fs::voidFsFunction;
16
17 using asyncpp::io::AsyncIoLoop;
18
19 /*
20  * 用于演示如何在协程中通过co_await调用异步化的文件系统操作函数
21  *  - co_await会自动控制协程的休眠和唤醒，调用者无需关心其实现细节
22  */
23 Coroutine asyncF() {
24     std::string dirPath = "dir1/a/b/c";
25
26     // 创建目录
27     std::string cmd = "createDirectories";
28     std::cout << "[AWAIT] Before: " << cmd << std::endl;
29     auto createResult = co_await createDirectories(dirPath);
30     std::cout << "[AWAIT] After: " << cmd << ": " << std::boolalpha << createRe
31
32     // 判断路径是否存在
33     cmd = "exists1";
34     std::cout << "[AWAIT] Before: " << cmd << std::endl;
35     auto existsResult1 = co_await exists(dirPath);
```

```

36     std::cout << "[AWAIT] After: " << cmd << ": " << std::boolalpha << existsRe
37
38     // 删除目录
39     cmd = "removeAll";
40     std::cout << "[AWAIT] Before: " << cmd << std::endl;
41     auto removeResult = co_await removeAll(dirPath);
42     std::cout << "[AWAIT] After: " << cmd << ": " << std::boolalpha << removeRe
43
44     // 判断路径是否存在
45     cmd = "exists2";
46     std::cout << "[AWAIT] Before: " << cmd << std::endl;
47     auto existsResult2 = co_await exists(dirPath);
48     std::cout << "[AWAIT] After: " << cmd << ": " << std::boolalpha << existsRe
49 }
50
51 void hello() {
52     std::cout << "<HELLO>" << std::endl;
53 }
54
55 auto asyncHello() {
56     return asyncify(hello);
57 }
58
59 /*
60  * 用于演示如何调用asyncify来将一个普通的void函数异步化
61  * - asyncHello的返回值推荐使用auto让编译器诊断其类型，
62  *。 如果不使用auto，这里需要写明其返回类型为asyncpp::task::Awaitable<void>
63  * - 在协程中就可以直接通过co_await调用asyncHello即可
64  */
65 asyncpp::task::Coroutine testVoid() {
66     // void函数封装示例
67     co_await asyncHello();
68 }
69
70 int main() {
71     // 启动异步I/O任务线程
72     AsyncIoLoop::start();
73
74     // 调用协程（协程会并发执行）
75     asyncF();
76     testVoid();
77
78     // 启动主线程任务循环（一定要最后调用，这里会阻塞）
79     AsyncTaskLoop::start();
80
81     return 0;
82 }

```

我们在 main 函数中，首先调用 AsyncIoLoop::start 启动异步 I/O 任务线程，接着调用 asyncF 和 testVoid。

在调用 `asyncF` 时遇到 `co_await createDirectories` 时会先休眠，此时控制权会交还给 `main` 函数，然后 `main` 函数就会马上调用 `testVoid` 这个协程，`testVoid` 遇到 `co_await asyncHello` 后会休眠再回到 `main` 函数，然后启动主线程循环。

因此，程序会先输出 `[AWAIT] Before...`，然后输出。因为是异步的，我们其实无法准确得知运行时的具体顺序，所以，程序的控制台输出可能是后面截图里展示的这样。

```
[AWAIT] Before: createDirectories
<HELLO>
[AWAIT] After: createDirectories: false
[AWAIT] Before: exists1
[AWAIT] After: exists1: true
[AWAIT] Before: removeAll
[AWAIT] After: removeAll: 1
[AWAIT] Before: exists2
```

深入理解 Coroutines

看完编程实战后，你是不是对异步的概念和基于协程的异步实现有了新的体会。现在，我们回到 **C++ Coroutines** 的概念上，并做一些更深入的讨论，把协程调度的细节再梳理一下。

首先，协程是一个与线程独立的概念，协程的核心是让调用者和被调用的协程具备一种协同调度的能力：协程可以通过 `co_await` 暂时休眠并将控制权交还给调用者，调用者可以通过协程句柄的 `resume` 重新唤醒协程。

其次，协程通过较为复杂的约定为开发者提供了更细粒度控制协程调度的能力。我们一定要实现的类型是 **Coroutine**、**Promise**，如果想要自定义 `co_await` 的行为，还需要实现 **Awaitable** 和 **Awaiter** 类型。

- **Coroutine** 类型可以将协程的句柄作为自己的成员变量，并以协程句柄为基础为协程调用者提供调度协程的接口。
- **Promise** 类型可以在协程帧中存储更多的自定义数据，实现协程的各种元数据以及自定义状态的存储与传递。
- **Awaitable** 和 **Awaiter** 可以控制 `co_await` 的各种行为，包括 `co_await` 后协程是否休眠，休眠后何时重新唤醒协程等。

在细粒度实现协程调度时还可以细分成两种情况，让我们分别看一下。

针对调用者的协程调度，我们需要关注 **Coroutine** 和 **Promise** 的实现细节。**Promise** 中可以通过 **get_return_object** 控制调用协程的返回值类型，一般返回类型就是 **Coroutine**。而在 **Coroutine** 类型中，我们需要定义为调用者提供的各种调度控制函数，根据实际业务需求实现相关的接口。

针对协程的内部调度，**C++** 是通过 **co_await** 实现的，我们需要关注 **Awaitable** 和 **Awaiter** 的实现细节。

为了真正实现协程的异步执行，我们可以在 **Awaiter** 的 **await_suspend** 中将协程的相关信息，包括 **Awaiter** 对象、协程句柄传递给其他的线程，在其他线程中执行任务函数并恢复协程执行。为了确保线程安全，我们甚至可以在一些应用中，当任务函数执行完后，将协程的相关数据传回主线程，让主线程自己唤醒协程。

因此，只要符合与 **C++** 协程接口的约定，我们就可以根据实际需求，定义整个协程的执行与调度过程。只要了解整个协程的执行机制和线程的切换机制，就可以通过协程实现各式各样的异步任务执行与调度。

总结

虽然就目前来说，**C++20** 提供的协程看起来很粗糙——它仅提供了语言层面的支持，缺乏标准库的支持。因此，就目前来说入门门槛还相对较高，但是我们已经能够实现足够灵活的异步调度、实现我们自己的协程框架，并满足各式各样的任务调度需求。

C++ Coroutines 可以在几乎零开销的情况下，大幅降低 **C++** 中实现异步调度的复杂度。实现基于 **C++20** 中的协程，就是去实现标准中针对协程的一整套约定，包含定义 **promise_type** 类型和 **Awaitable** 类型。其中，**Awaitable** 的实现决定了协程休眠的具体行为。

同时，我们在代码中设计了 **asyncify** 和 **ioAsyncify** 函数，使用这两个函数可以在不修改原有接口的情况下简单包装，**以非侵入式的方式生成为协程提供的异步函数**。与调用原函数相比，在协程中调用生成的包装函数只需要加上 **co_await** 即可，其他地方没有任何区别。

我们也需要关注，协程只是一种调度框架或者说是调度机制，协程和线程分别是独立的概念，甚至在实现具体协程机制的时候，往往也离不开线程技术，就像我们的实现一样。但是一旦实现了协程框架，就能降低调用者的异步编程门槛，这正是协程的价值所在。


我们期待更加成熟的支持会在 C++26 或后续演进标准中到来。可以预见，在未来 C++ 标准支持协程是光明的。


课后思考

在目前的设计中，异步 I/O 运行在一个独立线程上。由于所有的 I/O 任务都需要顺序执行，所以效率较低。请你尝试使用 `std::thread` 和 `std::mutex`，基于线程池来实现更好性能的并发。

不妨在这里分享你的答案，我们一同交流。下一讲见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | Coroutines实战（一）：异步文件操作库

下一篇 期中周 | 扩展现有协程框架，实现高级任务调度

精选留言 (4)

 写留言



_wang

2023-04-06 来自陕西

老师，上一个问题可以忽略了，其实是我看岔了，这里是对的，最后返回的都是Awaitable，感谢老师的精彩讲解。

作者回复: :thumbup



_wang

2023-04-06 来自陕西

不知道是不是我看岔了，ioAsyncity函数返回的类型是asyncify，而这个asyncify并没有定义为一

个Awaitable,那么是如何co_await的? 反倒asyncify函数返回的类型是Awaitable这个是可以被co_await的。



猪小擎

2023-03-30 来自北京

老师的github代码有好多GBK文件，建议修改下。

携程的本质优点减少上下文切换，（中断），中断操作系统会把寄存器存在栈顶，然后内核栈和用户栈互换，这些大概会消耗1-10微秒。携程就要节约这无数的1-10微秒。可以操作系统的IO操作是默认会中断的，要把中断改成yield出让执行权需要做什么呢？底层调用的filesystem的操作本身就是中断操作吧？上层改成携程，对于进程来说，这中断的消耗并没有节省。

作者回复: 感谢反馈，源代码正在做更新。

这里我们协程是一个简单的“示例性”封装，实际要通过协程处理I/O肯定要以IOCP、epoll和kqueue等技术为基础，并利用协程作为接口封装，在避免多线程与锁的情况下实现任务上下文切换，而不是直接使用filesystem这种简单的同步I/O接口，但是如果这样就会偏离我们想要讲解的重点，所以就使用filesystem做了简单的示例。



peter

2023-02-07 来自北京

请教老师两个问题：

Q1: asyncF与testVoid定义为什么不同？

asyncF与testVoid都是协程，但testVoid的定义是：asyncpp::task::Coroutine testVoid(); 而testVoid的定义是Coroutine asyncF()。函数前面部分不同，为什么？

Q2: 协程在C++语法层面的支持就是几个关键字吗？

这两篇读下来，有点模模糊糊，总体上感觉在C++语法层面，对于协程，似乎就是Coroutine、co_await这两个关键字，是吗？（Awaitable 和 Awaiter需要自己实现，不算关键字吧）

作者回复: Q1: 代码前面已经using asyncpp::task::Coroutine，所以这里asyncpp::task::Coroutine和Coroutine是一样的，这里只是演示引用该类型的不同用法，没有其他不同。

Q2: C++的协程提供了：

- （1）关键字：协程只提供了co_await、co_yield这两个关键字。
- （2）基础类型支持：coroutine_handle是C++协程标准库内提供的类型
- （3）协程协议框架：其他部分只定义了一个协议框架，包括Coroutine、Promise、Awaitable和Awaiter都是标准提出的概念，并且定义了相关的标准协议，但是并没有提供标准实现或者工具，需要开发者熟悉这些协议然后自己实现。

所以说，就目前来说（在C++26或后续标准到来以前）实现一个C++20的协程框架确实并不容易，但是基于框架再做业务开发就会简单很多。

