C++20协程入门教程

原文链接：https://zplutor.github.io/2022/03/25/cpp-coroutine-beginner/

1. 前言

随着Visual Studio 2022的发布，C++20终于来到了我们的眼前，在这个标准的新特性之中，最吸引人之一的是协程，对于饱受异步调用之繁琐写法的人来说，协程似乎是解决异步问题的灵丹妙药。

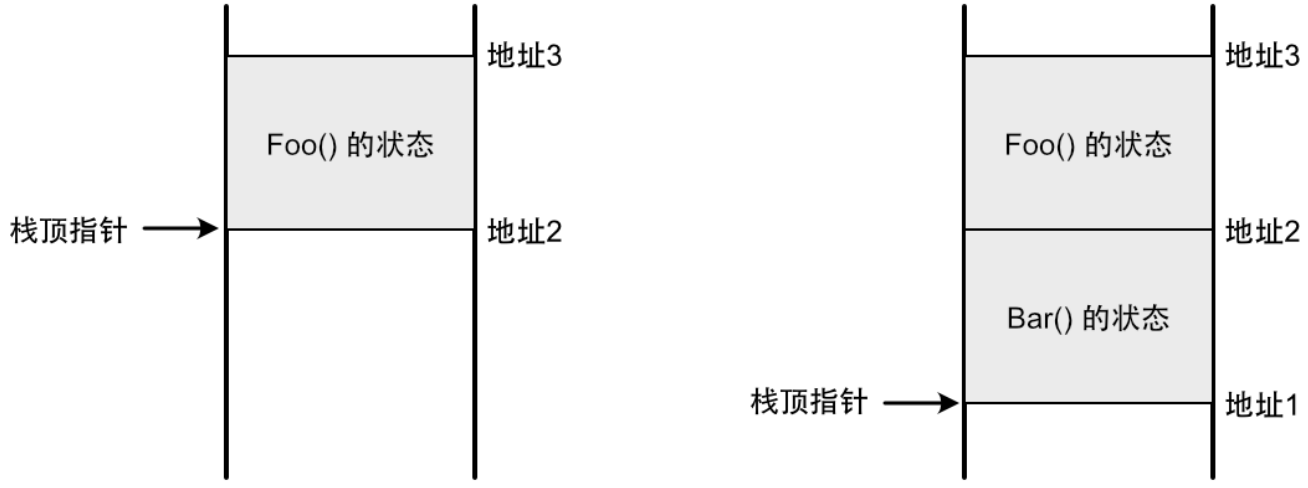
最初我对协程一无所知，当我开始学习它的时候，才发现它的复杂程度超出了我的预期。在网络上关于C++协程的文章有很多，但能够从初学者的角度把协程的基础原理讲清楚的寥寥无几。而且在 cppreference.com 上关于协程的页面仍然是未完成的状态，不少组件的文档仍然是空缺的。

在这样的背景下，学习过程是比较曲折的。我阅读了各种不同的文章，运行并调试其中的示例代码，最终才理解了C++的协程是怎么回事。在这篇文章中，我将自己的学习经历作为参考，从入门的角度来介绍C++协程。希望这篇文章能帮助大家更容易学习和理解协程。

2. 什么是协程

学习协程遇到的第一个问题是：什么是协程？一个简短的回答是：协程是一个函数，它可以暂停以及恢复执行。按照我们对普通函数的理解，函数暂停意味着线程停止运行了（就像命中了断点一样），那协程的不同之处在哪里呢？区别在于，普通函数是线程相关的，函数的状态跟线程紧密关联；而协程是线程无关的，它的状态与任何线程都没有关系。

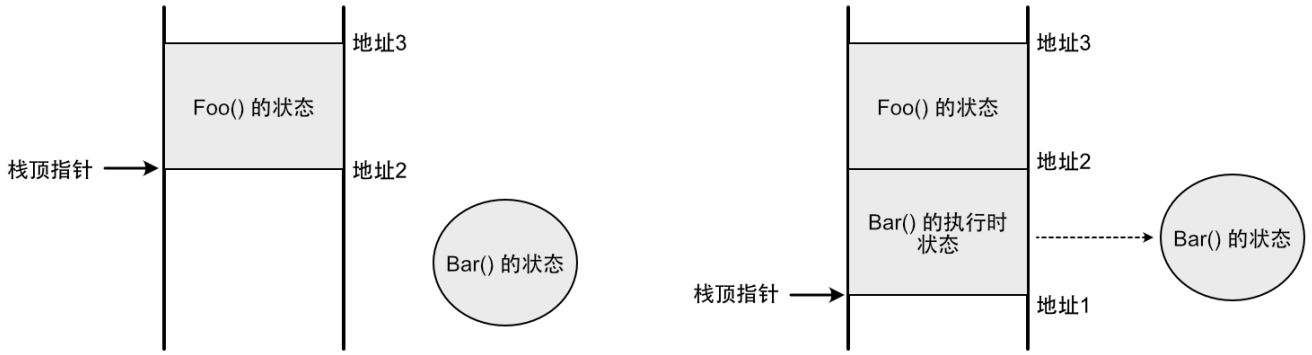
这个解释比较抽象，为了更好地理解，我们先来回顾一下函数的调用机制。在调用一个函数的时候，线程的栈上会记录这个函数的状态（参数、局部变量等），这是通过移动栈顶指针来完成的。例如，函数 `Foo()` 调用 `Bar()` 的过程如下所示：



首先，“地址3”到“地址2”这段空间，是分配给 `Foo()` 保存状态使用的，栈顶指针指向“地址2”；当调用 `Bar()` 的时候，栈顶指针移动到“地址1”，此时“地址2”到“地址1”这段空间是分配给 `Bar()` 保存状态使用的。当 `Bar()` 执行完毕，栈顶指针移动回“地址2”，`Bar()` 的状态被销毁，内存空间被回收。

由此可见，函数状态的维护完全依赖于线程栈，脱离了线程，函数就不复存在，所以说函数是线程相关的。

而协程不一样，协程的状态是保存在堆内存上的。假设 `Bar()` 是一个协程，那么调用它的过程如下所示：



首先， `Bar()` 的状态所需的内存会在堆上分配，独立于线程栈而存在。传递给它的参数都会复制到这个状态中，而局部变量会直接在这个状态中创建。调用 `Bar()` 的时候，由于本质上还是一个函数调用，所以栈顶指针也会往下移动，在栈上给执行 `Bar()` 所需的状态分配空间，其中会有一个引用指向在堆上的状态，这样一来， `Bar()` 就可以像一个普通函数那样执行了，线程也可以访问到位于堆上的状态。

如果协程需要暂停，那么当前执行到的代码位置会记录到堆的状态中。然后栈上的执行时状态被销毁，栈顶指针移动以回收空间，就像普通函数结束时那样。在下一次恢复执行时，堆状态中记录的暂停位置会读取出来，从这个位置接着执行。这样就实现了一个可暂停和恢复执行的函数。

由此可见，当协程执行的时候，它跟普通函数一样，也是需要依赖线程栈；但是，一旦它暂停了，它的状态就会独立保存在堆中，此时它跟任何线程都没有关系，调用它的线程可以继续去做其它事情而不会停止。在下一次恢复执行时，协程可以由上次执行的线程来执行，也可以由另外一个完全不同的线程来执行。所以说协程是线程无关的。

3. 协程的优点

协程的优点主要在于，它能优化异步逻辑的代码，使代码可读性更高。举个例子，假设我们有一个组件叫 `IntReader` ，它的功能是从一个访问速度很慢的设备上读取一个整数值，因此它提供的接口是异步的，如下所示：

```
class IntReader {
public:
    void BeginRead() {

        std::thread thread([]() {

            std::srand(static_cast<unsigned int>(std::time(nullptr)));
            int value = std::rand();


        });

        thread.detach();
```

```
    }
};
```

`BeginRead()` 开启了一个新的线程来生成一个随机的整数，模拟异步操作。作为一个仅用于示范的代码，这里尽量保持精简。

为了获取到 `IntReader` 的结果，传统的做法是提供一个回调函数，当操作完成的时候，通过回调函数告知使用者。如下所示：

```
class IntReader {
public:
    void BeginRead(const std::function<void(int)>& callback) {

        std::thread thread([callback]() {

            std::srand(static_cast<unsigned int>(std::time(nullptr)));
            int value = std::rand();

            callback(value);
        });

        thread.detach();
    }
};

void PrintInt() {

    IntReader reader;
    reader.BeginRead([](int result) {

        std::cout << result < std::endl;
    });
}
```

假如我们需要调用多个 `IntReader` ，把它们的结果加起来再输出，那么基于回调的代码就会很难看了：

```
void PrintInt() {

    IntReader reader1;
    reader1.BeginRead([](int result1) {

        int total = result1;

        IntReader reader2;
        reader2.BeginRead([total](int result2) {

            total += result2;

```

```
            IntReader reader3;
            reader3.BeginRead([total](int result3) {

                total += result3;
                std::cout << total << std::endl;
            });
        });
    });
}
```

代码不仅需要一层套一层，还要在每层回调之间传递结果，这就是俗称的“回调地狱”。而有了协程，这些问题都迎刃而解，我们可以这样来调用 `IntReader`：

```
Task PrintInt() {

    IntReader reader1;
    int total = co_await reader1;

    IntReader reader2;
    total += co_await reader2;

    IntReader reader3;
    total += co_await reader3;

    std::cout << total << std::endl;
}
```

代码逻辑顿时清晰了不少，看上去就像同步调用那样。在每一个调用 `co_await` 的地方，协程都会暂停下来，等 `IntReader` 操作完成之后再从这个地方恢复执行。接下来，我们来看下如何实现这种效果。

4. 实现一个协程

在C++中，只要在函数体内出现了 `co_await` 、`co_return` 和 `co_yield` 这三个操作符中的其中一个，这个函数就成为了协程。我们先来关注一下 `co_await` 操作符。

4.1. co_await 和 Awaitable

`co_await` 的作用是让协程暂停下来，等待某个操作完成之后再恢复执行。在上面的协程示例中，我们对 `IntReader` 调用了 `co_await` 操作符，目前这是不可行的，因为 `IntReader` 是我们自定义的类型，编译器不理解它，不知道它什么时候操作完成，不知道如何获取操作结果。为了让编译器理解我们的类型，C++定义了一个协议规范，只要我们的类型按照这个规范实现好，就可以在 `co_await` 使用了。

这个规范称作 `Awaitable`，它定义了若干个函数，传给 `co_await` 操作符的对象必须实现这些函数。这些函数包括：

- `await_ready()`，返回类型是 `bool`。协程在执行 `co_await` 的时候，会先调用 `await_ready()` 来询问“操作是否已完成”，如果函数返回了 `true` ，协程就不会暂停，

而是继续往下执行。实现这个函数的原因是，异步调用的时序是不确定的，如果在执行 `co_await` 之前就已经启动了异步操作，那么在执行 `co_await` 的时候异步操作有可能已经完成了，在这种情况下就不需要暂停，通过 `await_ready()` 就可以到达到这个目的。

- `await_suspend()`，有一个类型为 `std::coroutine_handle<>` 的参数，返回类型可以是 `void` 或者 `bool` 。如果 `await_ready()` 返回了 `false` ，意味着协程要暂停，那么紧接着会调用这个函数。该函数的目的是用来接收协程句柄（也就是 `std::coroutine_handle<>` 参数），并在异步操作完成的时候通过这个句柄让协程恢复执行。协程句柄类似于函数指针，它表示一个协程实例，调用句柄上的对应函数，可以让这个协程恢复执行。

  `await_suspend()` 的返回类型一般为 `void`，但也可以是 `bool` ，这时候的返回值用来控制协程是否真的要暂停，这里是第二次可以阻止协程暂停的机会。如果该函数返回了 `false` ，协程就不会暂停（注意返回值的含义跟 `await_ready()` 是相反的）。

- `await_resume()`，返回类型可以是 `void` ，也可以是其它类型，它的返回值就是 `co_await` 操作符的返回值。当协程恢复执行，或者不需要暂停的时候，会调用这个函数。

接下来，我们修改一下 `IntReader` ，让它符合 `Awaitable` 规范。下面是完整的示例代码：

```
1
2
3
4
5  class IntReader {
6  public:
7      bool await_ready() {
8          return false;
9      }
10
11     void await_suspend(std::coroutine_handle<> handle) {
12
13         std::thread thread([this, handle]() {
14
15             std::srand(static_cast<unsigned int>(std::time(nullptr)));
16             value_ = std::rand();
17
18             handle.resume();
19         });
20
21         thread.detach();
22     }
23
24     int await_resume() {
25         return value_;
26     }
27
28 private:
29     int value_{};
30 };
31
```

```
class Task {
public:
    class promise_type {
    public:
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
        void return_void() {}
    };
};

Task PrintInt() {

    IntReader reader1;
    int total = co_await reader1;

    IntReader reader2;
    total += co_await reader2;

    IntReader reader3;
    total += co_await reader3;

    std::cout << total << std::endl;
}

int main() {

    PrintInt();

    std::string line;
    while (std::cin >> line) { }
    return 0;
}
```

我们先忽略返回类型 `Task` ，下文会专门介绍协程的返回类型。在这个例子中，对于 `await_ready()` 函数，我们总是返回 `false`，即协程总是要暂停。然后我们把子线程改成在 `await_supsend()` 中启动，也就是在协程暂停的时候来启动，因此不再需要 `BeginRead()` 函数了。子线程生成随机数之后，保存在 `value_` 成员变量中，然后调用协程句柄的 `resume()` 函数来恢复协程执行。最后通过 `await_resume()` 函数把结果返回给调用者。

值得关注的地方是线程的执行。在 `main()` 函数中，主线程调用了 `PrintInt()` ，执行到 `co_await reader1` 这一行，协程暂停了，于是主线程退出 `PrintInt()` ，返回到 `main()` 继续往下执行，最后在 `while` 循环中等待用户输入。接下来，在 `reader1` 中启动的子线程调用了协程句柄的 `resume()` ，所以从 `co_await reader1` 中恢复执行的是这个子线程，直到 `co_await reader2` ，协程再次暂停，子线程退出。以此类推，后面的流程分别由 `reader2` 和 `reader3` 中启动的子线程来继续执行。所以，在 `PrintInt()` 这个协程内，总共有四个线程参与了执行。这里的关键点是：哪个线程调用协程句柄的 `resume()` ，就由哪个线程恢复协程执行。建议在IDE中设置断点来观察这个示例程序的执行流程，以便更好地理解。

4.2. 预定义的 Awaitable 类型

C++预定义了两个符合 Awaitable 规范的类型： std::suspend_never 和 std::suspend_always 。顾名思义，这两个类型分别表示“不暂停”和“要暂停”，实际上它们的区别仅在于 await_ready() 函数的返回值， std::suspend_never 会返回 true，而 std::suspend_always 会返回 false。除此之外，这两个类型的 await_supsend() 和 await_resume() 函数实现都是空的。

这两个类型是工具类，用来作为 promise_type 部分函数的返回类型，以控制协程在某些时机是否要暂停。下文会详细介绍 promise_type 。

4.3. 协程的返回类型和 promise_type

现在我们把关注点聚焦在协程的返回类型上。C++对协程的返回类型只有一个要求：包含名为 promise_type 的内嵌类型。跟上文介绍的 Awaitable 一样， promise_type 需要符合C++规定的协议规范，也就是要定义几个特定的函数。 promise_type 是协程的一部分，当协程被调用，在堆上为其状态分配空间的时候，同时也会在其中创建一个对应的 promise_type 对象。通过在它上面定义的函数，我们可以与协程进行数据交互，以及控制协程的行为。

promise_type 要实现的第一个函数是 get_return_object() ，用来创建协程的返回值。在协程内，我们不需要显式地创建返回值，这是由编译器隐式调用 get_return_object() 来创建并返回的。这个关系看起来比较怪异， promise_type 是返回类型的内嵌类型，但编译器不会直接创建返回值，而是先创建一个 promise_type 对象，再通过这个对象来创建返回值。

那么协程的返回值有什么用呢？这取决于协程的设计者的意图，取决于他想要以什么样的方式来使用协程。例如，在上文的示例中， PrintInt() 这个协程只是输出一个整数，不需要与调用者有交互，所以它的返回值只是一个空壳。 假如我们想实现一个 GetInt() 协程，它会返回一个整数给调用者，由调用者来输出结果，那么就需要对协程的返回类型做一些修改了。

4.4. co_return

我们现在把 PrintInt() 修改成 GetInt() 。使用 co_return 操作符可以从协程中返回数据，如下所示：

```
1  Task GetInt() {
2
3      IntReader reader1;
4      int total = co_await reader1;
5
6      IntReader reader2;
7      total += co_await reader2;
8
9      IntReader reader3;
10     total += co_await reader3;
11
```

```
    co_return total;
}
```

`co_return total` 这个表达式等价于 `promise_type.return_value(total)` ，也就是说，返回的数据会通过 `return_value()` 函数传递给 `promise_type` 对象， `promise_type` 要实现这个函数才能接收到数据。除此之外，还要想办法让返回值 `Task` 能访问到这个数据。为了减少数据传递，我们可以在 `promise_type` 和 `Task` 之间共享同一份数据。修改之后的完整示例如下所示：

```


class IntReader {
public:
    bool await_ready() {
        return false;
    }

    void await_suspend(std::coroutine_handle<> handle) {

        std::thread thread([this, handle]() {

            std::srand(static_cast<unsigned int>(std::time(nullptr)));
            value_ = std::rand();

            handle.resume();
        });

        thread.detach();
    }

    int await_resume() {
        return value_;
    }

private:
    int value_{};
};

class Task {
public:
    class promise_type {
    public:
        promise_type() : value_(std::make_shared<int>()) {

        }

        Task get_return_object() {
            return Task{ value_ };
        }

        void return_value(int value) {
```

```
45              *value_ = value;
46          }
47
48          std::suspend_never initial_suspend() { return {}; }
49          std::suspend_never final_suspend() noexcept { return {}; }
50          void unhandled_exception() {}
51
52      private:
53          std::shared_ptr<int> value_;
54      };
55
56  public:
57      Task(const std::shared_ptr<int>& value) : value_(value) {
58
59      }
60
61      int GetValue() const {
62          return *value_;
63      }
64
65  private:
66      std::shared_ptr<int> value_;
67  };
68
69  Task GetInt() {
70
71      IntReader reader1;
72      int total = co_await reader1;
73
74      IntReader reader2;
75      total += co_await reader2;
76
77      IntReader reader3;
78      total += co_await reader3;
79
80      co_return total;
81  }
82
83  int main() {
84
85      auto task = GetInt();
86
87      std::string line;
88      while (std::cin >> line) {
89          std::cout << task.GetValue() << std::endl;
90      }
91      return 0;
92  }
```

我们在 `promise_type` 和 `Task` 之间，使用了 `std::shared_ptr<int>` 来共享数据。在 `get_return_object()` 中创建 `Task` 的时候，把 `promise_type` 里的智能指针传递了过去，这样它们就能访问到同一个数据。在 `promise_type` 的 `return_value()` 中写数据，然后在 `Task` 的 `GetValue()` 中读数据。

异步是具有传染性的，`GetInt()` 内部调用了异步操作，所以它自身实际上也是一个异步操作。为了等待它执行完成，我们把 `task.GetValue()` 的调用放在了 `while` 循环中，每当用户输入一次之后就进行输出。由于这是一个简单的示例程序，没有各种高级的异步同步机制，所以通过等待用户输入方式来变相地等待协程执行完成。

在真实的使用场景中，协程的返回类型还需要提供各种同步机制才能给调用者使用，例如加上回调、通知等，就像普通的异步操作一样。由此可见，协程的优点体现在它内部的代码逻辑上，而不是对外的使用方式上。当然，我们也可以让协程的返回类型实现 `Awaitable` 规范，让它可以被另外一个协程更好地调用。这样一来，调用协程的也必须是协程，这样层层往上传递，直到遇到不能改成协程的函数为止，例如 `main()` 函数。从这个角度来说，协程也是具有传染性的。

最后，跟普通的 `return` 一样，`co_return` 也可以不带任何参数，这时候协程以不带数据的方式返回，相当于调用了 `promise_type.return_void()`，`promise_type` 需要定义这个函数以支持不带数据的返回。如果我们在协程结束的时候没有调用任何 `co_return`，那么编译器会隐式地加上一个不带参数的 `co_return` 调用。

4.5. co_yield

当协程调用了 `co_return`，意味着协程结束了，就跟我们在普通函数中用 `return` 结束函数一样。这时候，与这个协程实例有关的内存都会被释放掉，它不能再执行了。如果需要在协程中多次返回数据而不结束协程的话，可以使用 `co_yield` 操作符。

`co_yield` 的作用是，返回一个数据，并且让协程暂停，然后等下一次机会恢复执行。`co_yield value` 这个表达式等价于 `co_await promise_type.yield_value(value)`，`co_yield` 的参数会传递给 `promise_type` 的 `yield_value()` 函数，再把这个函数的返回值传给 `co_await`。上文提到，传给 `co_await` 的参数要符合 `Awaitable` 规范，所以 `yield_value()` 的返回类型也要满足这个规范。在这里就可以使用预定义的 `std::supsend_never` 或 `std::suspend_always`，通常会使用后者来让协程每次调用 `co_yield` 的时候都暂停。

下面我们修改一下示例，当用户输入一次之后，从协程取出一个值输出，然后让它生成下一个值；用户继续输入，又取出一个值输出，再生成下一个值，如此反复循环。完整的示例代码如下：

```
1
2
3
4
5  class IntReader {
6  public:
7      bool await_ready() {
8          return false;
9      }
10
11     void await_suspend(std::coroutine_handle<> handle) {
12
13         std::thread thread([this, handle]() {
14
15             static int seed = 0;
16             value_ = ++seed;
17
18             handle.resume();
```

```
        });

        thread.detach();
    }

    int await_resume() {
        return value_;
    }

private:
    int value_{};
};

class Task {
public:
    class promise_type {
    public:
        Task get_return_object() {
            return Task{ std::coroutine_handle<promise_type>::from_promise(*t
        }

        std::suspend_always yield_value(int value) {
            value_ = value;
            return {};
        }

        void return_void() { }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}

        int GetValue() const {
            return value_;
        }

    private:
        int value_{};
    };

public:
    Task(std::coroutine_handle<promise_type> handle) : coroutine_handle_(hand

    }

    int GetValue() const {
        return coroutine_handle_.promise().GetValue();
    }

    void Next() {
        coroutine_handle_.resume();
    }

private:
    std::coroutine_handle<promise_type> coroutine_handle_;
};

Task GetInt() {

```

```
77      while (true) {
78
79          IntReader reader;
80          int value = co_await reader;
81          co_yield value;
82      }
83  }
84
85  int main() {
86
87      auto task = GetInt();
88
89      std::string line;
90      while (std::cin >> line) {
91
92          std::cout << task.GetValue() << std::endl;
93          task.Next();
94      }
95      return 0;
96  }
```

这个示例修改的点比较多，我们拆解来看。首先，为了方便看出来程序的确是按照我们的预期来运行的，这里把 `IntReader::await_suspend()` 子线程内生成随机整数改成生成递增的整数。

然后，为了让使用者可以恢复协程执行， `Task` 增加了一个 `Next()` 函数，这个函数调用了作为成员变量的协程句柄来恢复执行：

```
1  void Next() {
2      coroutine_handle_.resume();
3  }
```

这意味着 `Task` 需要拿到协程的句柄，这是在 `promise_type` 的 `get_return_object()` 中通过以下方式传递过去的：

```
1  Task get_return_object() {
2      return Task{ std::coroutine_handle<promise_type>::from_promise(*this) };
3  }
```

`std::coroutine_handle` 的 `from_promise()` 函数可以通过 `promise_type` 对象获取与之关联的协程句柄，反之，协程句柄上也有一个 `promise()` 函数可以获取对应的 `promise_type` 对象，他们是可以互相转换的。所以，在 `Task` 和 `promise_type` 之间就不需要使用 `std::shared_ptr<int>` 来共享数据了， `Task` 通过协程句柄就能访问到 `promise_type` 对象，像下面这样直接取数据就可以了：

```
int GetValue() const {
    return coroutine_handle_.promise().GetValue();
}
```

这里要注意一下协程句柄 `std::coroutine_handle` 的模板类型。在前面的例子中，协程句柄的类型是 `std::coroutine_handle<>` ，不带模板参数；而在这个例子中，协程句柄的类型是 `std::coroutine_handle<promise_type>` ，模板参数中填入了 `promise_type` 类型。它们的区别类似于指针 `void*` 和 `promise_type*` 的区别，前者是无类型的，后者是强类型的。两种类型的协程句柄本质上是相同的东西，它们可以有相同的值，指向同一个协程实例，而且也都可以恢复协程执行。但只有强类型的 `std::coroutine_handle<promise_type>` 才能调用 `from_promise()` 获取到 `promise_type` 对象。

接下来，协程 `GetInt()` 的实现修改成一个无限循环，在循环内通过 `IntReader` 获取到整数，再通过 `co_yield` 把整数返回出去：

```
Task GetInt() {

    while (true) {

        IntReader reader;
        int value = co_await reader;
        co_yield value;
    }
}
```

对于协程来说，无限循环是常见的实现方式，由于它具有暂停的特性，并不会像普通函数那样让线程在里面死循环。

最后，在 `promise_type` 中定义了 `yield_value()` 函数来接收 `co_yield` 返回的数据。我们希望返回数据之后立即暂停协程，所以返回类型定义成了 `std::supsend_always` 。

4.6. 协程的生命周期

正如上文所说的，在一开始调用协程的时候，C++会在堆上为协程的状态分配内存，这块内存必须在适当的时机来释放，否则就会造成内存泄漏。释放协程的内存有两种方式：自动释放和手动释放。

当协程结束的时候，如果我们不做任何干预，那么协程的内存就会被自动释放。调用了 `co_return` 语句之后，协程就会结束，下面两个协程是自动释放的例子：

```
Task GetInt() {

    IntReader reader;
    int value = co_await reader;

    co_return value;
```

```
}


Task PrintInt() {

    IntReader reader1;
    int value = co_await reader;

    std::cout << value << std::endl;
}
```

`PrintInt()` 没有出现 `co_return` 语句，编译器会在末尾隐式地加上 `co_return` 。

自动释放的方式有时候并不是我们想要的，参考下面这个例子：

```
class Task {
public:
    class promise_type {
    public:
        Task get_return_object() {
            return Task{ std::coroutine_handle<promise_type>::from_promise(*t
        }

        void return_value(int value) {
            value_ = value;
        }

        int GetValue() const {
            return value_;
        }

        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}

    private:
        int value_{};
    };

public:
    Task(std::coroutine_handle<promise_type> handle) : coroutine_handle_(hand

    }

    int GetValue() const {
        return coroutine_handle_.promise().GetValue();
    }

private:
```

```
39      std::coroutine_handle<promise_type> coroutine_handle_;
40  };
41
42  Task GetInt() {
43
44      co_return 1024;
45  }
46
47  int main() {
48
49      auto task = GetInt();
50
51      std::string line;
52      while (std::cin >> line) {
53          std::cout << task.GetValue() << std::endl;
54      }
55      return 0;
56  }
```

在这个例子中， `GetInt()` 协程通过 `co_return` 返回了1024给 `promise_type` ；协程返回值 `Task` 通过协程句柄访问 `promise_type` ，从中取出这个值。随着用户的输入，把这个值输出来。运行程序，我们会发现输出的值并不是1024，而是一个随机值；也有可能会出现地址访问错误的异常。

造成这个现象的原因是，协程在返回1024之后就被自动释放了， `promise_type` 也跟着被一起释放了，此时在 `Task` 内部持有的协程句柄已经变成了野指针，指向一块已经被释放的内存。所以访问这个协程句柄的任何行为都会造成不确定的后果。

解决这个问题的方法是，修改 `promise_type` 中 `final_supsend()` 函数的返回类型，从 `std::suspend_never` 改成 `std::suspend_always` 。协程在结束的时候，会调用 `final_suspend()` 来决定是否暂停，如果这个函数返回了要暂停，那么协程不会自动释放，此时协程句柄还是有效的，可以安全访问它内部的数据。

不过，这时候释放协程就变成我们的责任了，我们必须在适当的时机调用协程句柄上的 `destroy()` 函数来手动释放这个协程。在这个例子中，可以在 `Task` 的析构函数中做这个事情：

```
1  ~Task() {
2      coroutine_handle_.destroy();
3  }
```

只要协程处于暂停状态，就可以调用协程句柄的 `destroy()` 函数来释放它，不一定要求协程结束。对于通过无限循环来实现的协程，手动释放是必需的。

与 `final_suspend()` 相对应的是 `initial_suspend()` ，在协程刚开始执行的时候，会调用这个函数来决定是否暂停。我们可以将这个函数的返回类型改成 `std::suspend_always` 来让协程一执行即暂停。这对于一些需要延迟执行的场景是有用的，例如，我们想先获取一批协程句柄，像数据那样对它们进行管理，在稍后的时机再挑选合适的协程来执行。

4.7. 异常处理

最后，我们看一下协程的异常处理。编译器生成的执行协程的伪代码大概如下所示：

```
try {

    co_await promise_type.initial_suspend();


}
catch (...) {

    promise_type.unhandled_exception();
}

co_await promise_type.final_suspend();
```

协程主要的执行代码都被 `try - catch` 包裹，假如抛出了未处理的异常， `promise_type` 的 `unhandled_exception()` 函数会被调用，我们可以在这个函数里面做对应的异常处理。由于这个函数是在 `catch` 语句中调用的，我们可以在函数内调用 `std::current_exception()` 函数获取异常对象，也可以调用 `throw` 重新抛出异常。

调用了 `unhandled_exception()` 之后，协程就结束了，接下来会继续调用 `final_suspend()` ，与正常结束协程的流程一样。C++规定 `final_suspend()` 必须定义成 `noexcept` ，也就是说它不允许抛出任何异常。

5. 后记

至此，关于C++协程的基础内容介绍完毕。从当前的使用方式来看，C++20的协程只提供了最基础的能力，为了用上它，我们需要写不少代码，将一个一个小块串联起来。所以，现在C++协程的形态是不友好的，晦涩难懂，难以学习。

要想在实际的开发中使用上C++协程，还有比较长的路。我们可以自己动手对它进行封装，或者寻求第三方库的解决方案，或者继续期待未来的C++标准带来更高层封装的协程组件。