c++20协程入门

飞鼠明天...
招聘C++/Linux开发，有意向直接私信

375 人赞同了该文章

随着coroutine ts正式进入c++20，c++已经进入协程时代了。c++20提供的无栈协程，拥有许多无与伦比的优越性，比如说没有传染性，可以与以前非协程风格的代码并存，再比如说不需要额外的调度器，总之是个好东西。

但是不幸的是c++20的协程标准只包含编译器需要实现的底层功能，并没有包含简单方便地使用协程的高级库，相关的类和函数进入std标准库估计要等到c++23。所以，在c++20中，如果要使用协程，要么等别人封装好了给你用，要么就要自己学着用底层的功能自己封装。

c++的协程功能是给库的开发者使用的，所以看起来比较复杂，但是经过库的作者封装以后用起来是非常简单的，比如说asio里面就已经封装好了，相关用法看我前面的这篇文章。另外，c++的协程性能非常之高，其作者的视频介绍里面说了，(一个进程)可以开启几十亿个协程，可以说是无出其右了。

虽然说协程主要是给库的作者使用的，但是学c++的，哪个不想写自己的库呢？所以学学肯定是有好处的。可惜现在关于c++20协程的教程很少，所以我试着写了这篇入门教程，介绍一些最简单最有用的东西，而且我希望提供一个将异步变为协程的通用套路。对于这篇教程，大家如果发现有什么错误请及时告诉我，我会第一时间修改。

协程的出现主要是为了解决异步编程的麻烦，异步编程一般是这样的：

```
async_call(input1, intput2, ..., call_back)
```

就是用一堆输入参数再加上一个回调函数作为参数， `async_call` 函数调用后立即返回，当异步操作完成时，call_back函数会被调用。

在C++中，使用异步的场景主要有两种：
1. 需要等待的结果在其它进程中提供，比如调用操作系统的异步io读写文件、通过网络发送请求到其它进程等待处理以后结果依然通过网络返回，文件读写完成或是网络请求返回时调用指定的回调函数；
2. 需要等待的结果在本进程中提供，一般就是在别的线程中，将请求发送到别的线程，别的线程操作完成以后，调用指定的回调函数。

不管是哪种情况，对于异步调用，都可以用协程改造成一个如下格式的“同步”调用：

```
co_await coro_call(input1, input2, ...)
```

其实改造起来极为简单，只要在异步函数的回调函数中调用resume()恢复协程即可。对于1和2两种情况，第一种情况更简单，因为第二种可能会需要考虑同步的问题。

协程通过Promise和Awaitable接口的15个以上的函数来提供给程序员定制协程的流程和功能，实现最简单的协程需要用到其中的8个（5个Promise的函数和3个Awaitable的函数），其中Promise的函数可以先不管它，先来看Awaitable的3个函数。

如果要实现形如 `co_await blabla;` 的协程调用格式, blabla就必须实现Awaitable。 `co_await` 是一个新的运算符。Awaitable主要有3个函数：
1. `await_ready` ：返回Awaitable实例是否已经ready。协程开始会调用此函数，如果返回true，表示你想得到的结果已经得到了，协程不需要执行了。所以大部分情况这个函数的实现是要return false。
2. `await_suspend` ：挂起awaitable。该函数会传入一个 `coroutine_handle` 类型的参数。这是一个由编译器生成的变量。在此函数中调用handle.resume()，就可以恢复协程。
3. `await_resume` ：当协程重新运行时，会调用该函数。这个函数的返回值就是 `co_await` 运算符的返回值。

既然这3个函数都搞清楚了，那我们只要自己实现这3个函数，就可以用 `co_await` 来等待结果了。 `co_await` 在协程中使用，但是协程的入口必须是在某个函数中，函数的返回值需要满足Promise的规范。最简单的Promise如下：

```
struct Task
{
    struct promise_type {
        auto get_return_object() { return Task{}; }
        auto initial_suspend() { return std::experimental::suspend_never{}; }
        auto final_suspend() { return std::experimental::suspend_never{}; }
        void unhandled_exception() { std::terminate(); }
        void return_void() {}
    };
};
```

下面来举个例子，在例子中其实这个Task除了作为函数返回值以外没其它作用。

以一个add函数为例： `int add100(int a)` ，就是将参数a加上100，假设这个add非常耗时，把它设计成一个异步的调用，异步调用的结果当然是通过回调函数进行通知，定义如下：

```
using call_back = std::function<void(int)>;
void Add100ByCallback(int init, call_back f) //init是传入的初始值，add之后的结果由回调函数
{
```

```
    std::thread t([init, f]() {
        std::this_thread::sleep_for(std::chrono::seconds(5)); // sleep一下，假装很耗时
        f(init + 100); // 耗时的计算完成了，调用回调函数
    });
    t.detach();
}
```

这个函数的调用是大家都熟悉的异步函数调用，提供一个回调函数作为参数就可以了，比如直接将得到的结果输出：

```
Add100ByCallback(5, [](int value){ std::cout<<"get result: "<<value<<"\n"; });
```

现在我们把通过回调函数的异步调用改成协程，协程和普通的回调函数不同，可以直接得到add之后的结果。普通的回调函数是不可以有返回值的。协程如下：

```
Task Add100ByCoroutine(int init, call_back f)
{
    int ret = co_await Add100AWaitable(init); //co_await可以有返回值
    f(ret);
}
```

一次co_await不能体现协程的优越性，可以连续调用几次，将多个异步调用转化成串行化的“同步”调用，像下面这样，立马有从原始社会进入到现代社会的感觉，作为一个长年写异步程序的码农，庆幸自己终于迎来了解放：

```
Task Add100ByCoroutine(int init, call_back f)
{
    int ret = co_await Add100AWaitable(init);
    ret = co_await Add100AWaitable(ret);
    ret = co_await Add100AWaitable(ret);
    f(ret);
}
```

根据前面提到的3个函数，来实现Add100AWaitable类型：

```
struct Add100AWaitable
{
    Add100AWaitable(int init):init_(init) {}
    bool await_ready() const { return false; }
    int await_resume() { return result_; }
```

```
    void await_suspend(std::experimental::coroutine_handle<> handle)
    {
        // 定义一个回调函数，在此函数中恢复协程
        auto f = [handle, this](int value) mutable {
            result_ = value;
            handle.resume(); // 这句是关键
        };
        Add100ByCallback(init_, f);
    }
    int init_; // 将参数存在这里
    int result_; // 将返回值存在这里
};
```

启动一个协程也很简单，直接调用Add100ByCoroutine即可：

```
Add100ByCoroutine(10, [](int value){ std::cout<<"get result from coroutine: "<<value<<
```

按照这个思路，可以将任意异步函数采用Awaitable包装的方法改造成协程，比如说上面例子中的异步函数Add100ByCallback。

按照本文中的例子依葫芦画瓢，然后再结合网上那些似懂非懂的文档，各位c++er都可以入门了。但是c++协程的知识非常之多，如何在需要的时候提供15个函数中的其它函数来定制不同的流程、如何减少不必要的内存分配、如何避免不必要的加锁等等，可以说是高手发挥的舞台。关于协程的进一步学习可以看这里（共有3篇）。