本文来自https://github.com/Serein207/C-Cpp-Study-Notes

更细致地学习协程请前往www.cppreference.com

协程

1 概述

协程 是C++20更新的的一大特性。顾名思义，它是用来协同运行各个程序的。我们可以把协程简单概括为： **协程是一个可以在执行时暂停（挂起）和恢复（继续）的函数**。C++的协程是无栈的，具有内存占用小的优点，允许数以亿计的协程同步运行。

协程支持库定义在 `<coroutine>`。

首先，我们要明确一些概念。协程不共享堆栈，所以可以随时暂停执行，跳转到其他协程中。像使用函数一样调用协程的函数/协程称为 **调用方**，恢复暂停的协程的函数/协程称为 **恢复方**。

如何在协程间跳转呢？C++为我们提供了一个新的运算符： `co_await`。它可以跳转回到调用方/恢复方，并保证当前协程中的局部变量都被临时保存起来，等待下次恢复时使用。该运算符的操作数是一元的，操作数可以是表达式，同时必须满足一定的条件。它可以将操作数传递回调用方/恢复方。

最后，我们强调一遍协程的概念，这些关键字我们在之后都会讲到： **当函数体包含以下关键字之一时，该函数是协程**：

- `co_await`：在等待一个计算完成时挂起一个协程的执行。当计算完成后，继续执行。
- `co_return`：从协程返回。在此之后，协程无法恢复。
- `co_yield`：从协程返回一个值给调用者，并挂起协程，随后再次调用协程，在它被挂起的地方继续执行。

2 协程句柄

在 `co_await` 被执行时，会在堆区创建一个可调用对象，该对象在被调用时将恢复执行该协程。这个可调用对象的类型为 `std::coroutine_handle<>` ——协程句柄。

协程句柄的行为很像指针，它保存协程状态，用于管理协程调度，但没有析构函数销毁协程，所以必须使用 `coroutine_handle::destroy()` 方法销毁协程状态（某些情况下协程可以在完成时自动销毁）

现在让我们看看 `co_await` 是如何执行的：

```
co_await awaitable;
```

当 *awaitable* 表达式被求值后，编译器会创建一个协程句柄并将其传递给 `awaitable.await_suspend(std::coroutine_handle<>)` , *awaitable* 必须可转化为 **可等待者**，同时该对象必须支持某些方法， `await_suspend` 只是其一。

看下面的例子，请暂时忽略 `RetObj` ，我们之后会讲到它：

```
struct RetObj {
    struct promise_type {
        RetObj get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
    };
};

struct Awaiter {
    std::coroutine_handle<>* _hp;
    bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) { *_hp = h; }
    void await_resume() const noexcept {}
};

RetObj counter(std::coroutine_handle<>* continuation_out) {
    AWaiter a { continuation_out };
    for (unsigned int i = 0; ; ++i) {
        co_await a;
        std::cout << "counter: " << i << std::endl;
    }
}

int main() {
    std::coroutine_handle<> h;
    counter(&h);
    for (int i = 0; i < 3; ++i) {
        std::cout << "main function" << std::endl;
```

```
        h.resume();
        // or
        // h();
    }
    h.destroy();
}
```

output

```
main function
counter: 0
main function
counter: 1
main function
counter: 2
```

我们发现，即使在函数之间来回切换，计数器也会保存上一次调用时的值。

在这个例子中，我们首先将协程句柄传递给 `counter`，然后将协程句柄保存到 `Awaiter` 类型中，每一次 `co_await` 时，编译器都会将生成的协程句柄使用 `await_suspend` 传递给 `a`，然后回到协程调用者 `main` 函数，直到下次 `h.resume()` 将协程恢复，将跳转到 `counter()` 协程继续循环。

当然，因为h不会改变，所以我们也没必要每次调用 `co_await` 时都更改协程句柄，只需要传递一次协程句柄给 `Awaiter` 即可。为此，我们可以这样做：

```
void Awaiter::await_suspend(std::coroutine_handle<> h) {
    if (_hp) {
        *_hp = h;
        _hp = nullptr;
    }
}
```

这种写法有点像是单例模式，只对 `h` 做一次初始化。

下面我们来看看其他两个方法，C++规定它们也必须在 `Awaiter` 类定义。因为在 `co_await` 被执行时，下面的方法也会被自动按顺序执行：

- `await_ready()` 如果返回 `false`，那么挂起协程并跳转到调用方/恢复方，`true` 就立即执行协程。
- `await_suspend()` 如果返回 `void`，那么立即将控制返回给当前协程的调用方/恢复方。
- `await_resume()` 返回的结果就是整个 `co_await expr` 的结果。

`<coroutine>` 提供了两个定义好的awaiter：`std::suspend_always` 和 `std::suspend_never`，前者的 `await_ready()` 返回 `false`，后者返回 `true`。其他的方法都为空函数体。

3 协程返回对象

现在，让我们接着讨论 `RetObj` 的实现。C++限制协程返回的类型。我们暂且叫这个类型 `R`。首先，该类型必须可使用 `R::promise_type`（内部类或别名）。`R::promise_type` 必须包含返回 `R` 实例的方法 `R get_return_object()`。通常情况下，像 `R` 这样协程返回的类型被叫做 **future**。

将协程句柄传递到 `counter()` 参数中未免有些不优雅，我们可以试着改为从 `counter()` 返回句柄。这需要我们将协程句柄放在返回对象中，恰好 `promise_type::get_return_object()` 方法能够定义返回对象，所以我们只需要将协程句柄拷贝进返回对象中即可。

到目前为止，我们一直都没有使用 `std::coroutine_handle<>` 的模板参数，我们来看看它的声明：

```
template <class Promise = void> struct coroutine_handle;
```

任何类型 `T` 的 `std::coroutine_handle<T>` 都可以隐式转换为 `std::coroutine_handle<void>`。可以调用任一类型来恢复具有相同效果的协程。

当 `Promise` 类型参数不是 `void` 时，允许在协程句柄和 `T` 之间相互转换。可以使用静态方法 `coroutine_handle::from_promise()` 将 `T` 转换为协程句柄：

```
std::coroutine_handle<promise_type>::from_promise(*this);
```

现在，让我们重写刚刚的示例代码：

```
struct RetObj2 {
    struct promise_type {
        RetObj2 get_return_obj() {
            return {std::coroutine_handle<promise_type>::from_promise(*this
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
    };

    std::coroutine_handle<promise_type> _hp;
    operator std::coroutine_handle<promise_type>() const { return _hp; }
    // A coroutine_handle<promise_type> converts to coroutine_handle<>
    operator std::coroutine_handle<>() const { return _hp; }
```

```
};

RetObj2 counter2() {
    for (unsigned int i = 0; ; ++i) {
        co_await std::suspend_always{};
        std::cout << "counter: " << i << std::endl;
    }
}

int main() {
    std::coroutine_handle<> h = counter2();
    for (int i = 0; i < 3; ++i) {
        std::cout << "main function" << std::endl;
        h.resume();
        // or
        // h();
    }
    h.destroy();
}
```

output

```
main function
counter2: 0
main function
counter2: 1
main function
counter2: 2
```

因为我们不再需要我们的awaiter保存协程句柄（因为我们已经将句柄放入返回对象中），我们只是执行 `co_await std::suspend_always{}` 。注意，如果任何代码调用 `counter2()` 并忽略返回值（或否无法销毁 `RetObj2` 对象中的句柄），会产生内存泄漏。

`RetObj2` 被称为 **协程状态**。

4 promise对象

如何在协程之间传递数据？我们可以在协程状态的 `promise_type` 中添加 `_value` 数据成员来传递数据。在 `main()` 函数中，可以通过 `std::coroutine_handle<RetObj3::promise_type>` 对象的 `promise()` 方法获得 `promise_type&` 对象，从而访问到 `_value` 数据成员。

在 `counter()` 中，回忆第一个示例，我们通过自定义 `Awaiter` 类来获取协程句柄。

在执行 `co_await` 时，编译器首先调用 `awaiter.await_ready()`（这是当已知结果就绪或可以同步完成时，用以避免暂停开销的快捷方式）。如果返回 `false`，那么：

- 暂停协程（以各局部变量和当前暂停点填充其协程状态）。
- 调用 `awaiter.await_suspend()`
  - 如果 `await_suspend` 返回 `void`，那么立即将控制返回给当前协程的调用方/恢复方（此协程保持暂停），否则
  - 如果 `await_suspend` 返回 `bool`，那么：
    - 值为 `true` 时将控制返回给当前协程的调用方/恢复方
    - 值为 `false` 时恢复当前协程。
  - 如果 `await_suspend` 返回某个其他协程的协程句柄，那么（通过调用 `handle.resume()`）恢复该句柄（注意这可以连锁进行，并最终导致当前协程恢复）。
- 调用 `awaiter.await_resume()`，它的结果就是整个 `co_await expr` 表达式的结果

换句话说，**一个协程不会暂停，除非 `await_ready` 返回 `false`，然后 `await_suspend` 返回 `true` 或 `void`。**

现在让我们新定义一个 `GetPromise` 的awaiter：

```
template <typename PromiseType>
struct GetPromise {
    PromiseType* _p;
    bool await_ready() { return false; } // says yes call await_suspend()
    bool await_suspend(std::coroutine_handle<PromiseType> h) {
        _p = &h.promise();
        return false;
    }
    PromiseType* await_resume() { return _p; }
};
```

除了 `void` 和 `bool` 之外，`await_suspend` 还可能返回一个 `coroutine_handle`，在这种情况下，将立即恢复返回的句柄。`GetPromise::await_suspend` 也可以返回句柄 `h` 以恢复协程，但大概这会降低效率。

下面是新的代码：

```
struct RetObj3 {
    struct promise_type {
        unsigned _value;

        RetObj3 get_return_object() {
            return {std::coroutine_handle<promise_type>::from_promise(*this
        }
```

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

    std::coroutine_handle<promise_type> _h;
    operator std::coroutine_handle<promise_type>() const { return _h; }
};

RetObj3 counter3() {
    auto pp = co_await GetPromise<RetObj3::promise_type>{};
    for (unsigned i = 0; ; ++i) {
        pp->_value = i;
        co_await std::suspend_always{};
    }
}

int main() {
    std::coroutine_handle<RetObj3::promise_type> h = counter3();
    RetObj3::promise_type& promise = h.promise();
    for (int i = 0; i < 3; ++i) {
        std::cout << "counter3: " << promise._value << std::endl;
        h();
    }
    h.destroy();
}
```

output

```
counter3: 0
counter3: 1
counter3: 2
```

5 co_yield

从协程传递一个值还有一种更优雅的手段——`co_yield`。我们可以在`promise_type`中添加`yield_value`方法来简化前面的示例：

```
struct ReObj4 {
    struct promise_type {
        unsigned _value;

        RetObj4 get_return_object() {
            return {std::coroutine_handle<promise_type>::from_promise(*this
        }
        std::suspend_never initial_suspend() { return {}; }
```

```
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
        std::suspend_always yield_value(unsigned value) {
            _value = value;
            return {};
        }
    };

    std::coroutine_handle<promise_type> _h;
};

RetObj4 counter4() {
    for (unsigned i = 0;; ++i)
        co_yield i;       // co yield i => co_await promise.yield_value(i)
}

int main() {
    auto h = counter4()._h;
    auto &promise = h.promise();
    for (int i = 0; i < 3; ++i) {
        std::cout << "counter4: " << _promise.value << std::endl;
        h();
    }
    h.destroy();
}
```

6 `co_return`

为了发出协程结束的信号，C++添加了一个新的 `co_return` 运算符。协程有三种方式来表示它已经结束：

1. 协程可以使用 `co_return e;` 返回最终值 `e`
2. 协程可以使用不带值的 `co_return;`（或使用 `void` 表达式）结束协程
3. 协程交还控制权，这与之前的情况类似

当协程执行到 `co_return` 时，进行以下操作：

- 对于 `void` 调用 `promise.return_void()`
- 或对于 `co_return expr`；调用 `promise.return_value(expr)`，其中 `expr` 具有非 `void` 类型
- 以创建顺序的逆序销毁所有具有自动存储期的变量。
- 调用 `promise.final_suspend()` 并 `co_await` 它的结果。

我们需要在 `promise_type` 中添加 `return_void()` 或者 `return_value()` 方法。如果协程已经执行结束，则 `coroutine_handle::done()` 返回 `true` 。注意 `coroutine_handle::operator bool()` 只是检查协程句柄是否包含指向协程的空指针，而不是检查是否完成。

```
struct RetObj5 {
    struct promise_type {
        unsigned _value;

        ~promise_type() {
            std::cout << "promise_type destroyed" << std::endl;
        }
        RetObj5 get_return_object() {
            return {std::coroutine_handle<promise_type>::from_promise(*this
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
        std::suspend_always yield_value(unsigned value) {
            _value = value;
            return {};
        }
        void return_void() {}
    };

    std::coroutine_handle<promise_type> _h;
};

RetObj5 counter5() {
    for (unsigned i = 0; i < 3; ++i)
    co_yield i;
    // falling off end of function or co_return; => promise.return_void();
    // (co_return value; => promise.return_value(value);)
}

int main() {
    auto h = counter5()._h;
    auto& promise = h.promise();
    while (!h.done()) { // Do NOT use while(h) (which checks h non-NULL)
        std::cout << "counter5: " << promise.value_ << std::endl;
        h();
    }
    h.destroy();
}
```

output

```
counter5: 0
counter5: 1
counter5: 2
promise_type destroyed
```

请注意，在前面的示例中，我们的 `promise` 对象上没有 `return_void()` 方法。没关系，只要我们没有使用 `co_return` 。否则，如果使用 `co_return` 但没有适当的 `return_void` 或 `return_value` 方法，将无法编译。然而，如果从协程交还控制权，并且 `promise_type` 缺少 `return_void` 方法，将是未定义行为。

7 UB

> 协程只保存函数栈对象，不保存 `this` 。

这会造成引用空悬的问题，对于最开始的样例代码，如果我们写下：

```
struct S {
    int i;
    promise function() {
        std::cout << "function start" << i << std::endl;
        co_return;
    }
};

int main() {
    promise p = S{0}.f(); // S{0} has been destroyed
    std::cout << "main start" << std::endl;
    p._h.resume(); // UB
}
```