C++ Summit 2019

吴咏炜
wuyongwei@gmail.com

深入浅出
C++20 协程

什么是协程？

协程是计算机程序的一类组件，推广了协作式多任务的子程序，允许执行被挂起与被恢复。相对子例程而言，协程更为一般和灵活……

——维基百科

简单示例：Python Generator

```python
def fibonacci():
    a = 0
    b = 1
    while True:
        yield b
        a, b = b, a + b
```

Python 里的可能用法

```python
# 打印头 20 项
for i in islice(fibonacci(),
                20):
    print(i)
```

```python
# 打印小于 10000 的数列项
for i in takewhile(
        lambda x: x < 10000,
        fibonacci()):
    print(i)
```

执行过程

```
a = 0; b = 1; yield b

a, b = 1, 0 + 1; yield b

a, b = 1, 1 + 1; yield b

a, b = 2, 1 + 2; yield b

a, b = 3, 2 + 3; yield b

…
```

```
i = 1; print(i)

i = 1; print(i)

i = 2; print(i)

i = 3; print(i)

i = 5; print(i)

…
```

在 C++ 里如何写出等价的代码？

过程分解

```python
def fibonacci():
    a = 0
    b = 1
    while True:
        yield b
        a, b = b, a + b
```

初始化

提供结果

计算下一项

等价的 C++ 代码 I

```cpp
class fibonacci {
public:
    class sentinel;
    class iterator;
    iterator begin() noexcept;
    sentinel end() noexcept;
};
```

```cpp
fibonacci::iterator fibonacci::begin()
    noexcept
{
    return iterator();
}

fibonacci::sentinel fibonacci::end()
    noexcept
{
    return sentinel();
}
```

等价的 C++ 代码 II（不完整）

```cpp
class fibonacci::sentinel {};

class fibonacci::iterator {
    uint64_t a_{0};
    uint64_t b_{1};
public:
    uint64_t operator*() const
    {
        return b_;
    }
    iterator& operator++()
    {
        auto tmp = a_;
        a_ = b_;
        b_ += tmp;
        return *this;
    }
```

初始化

提供结果

计算下一项

```cpp
    bool operator==(const iterator& rhs) const
    {
        return b_ == rhs.b_;
    }
    bool operator!=(const iterator& rhs) const
    {
        return !operator==(rhs);
    }
    bool operator==(const sentinel&) const
    {
        return false;
    }
    bool operator!=(const sentinel&) const
    {
        return true;
    }
};
```

使用 fibonacci

```cpp
for (auto i : fibonacci()) {
    if (i >= 10000) {
        break;
    }
    cout << i << endl;
}
```

```cpp
int count = 0;
for (auto i : fibonacci()) {
    cout << i << endl;
    if (++count == 20) {
        break;
    }
}
```

使用 fibonacci (利用 ranges)

```
for (auto i : fibonacci() |
        ranges::views::take_while([](uint64_t x) {
            return x < 10000;
        })) {
    cout << i << endl;
}
```

使用 fibonacci (利用 ranges) II

```
for (auto i : fibonacci() | ranges::views::take(20)) {
    cout << i << endl;
}
```

实现复杂度比较

- Python fibonacci：6 行
- C++ fibonacci（无需支持 ranges）：31 行
- C++ fibonacci（完整版）：48 行

C++ 的协程

- Boost.Coroutine
  - 早期实现，非现代 C++，已废弃
- Boost.Coroutine2
  - 需要 C++11
- CO2
  - 使用宏技巧实现
- C++ Coroutines TS
  - 目前被 MSVC 和 Clang 支持
  - 2019 年 7 月批准加入 C++20 标准草案

协程的常见用途

- 生成器（generator）
- 异步 I/O
- 惰性求值
- 事件驱动应用

Coroutines TS

- 新关键字：co_await, co_yield, co_return
- std::experimental 名空间中的新类型
    - coroutine_handle 模板
    - coroutine_traits 模板
    - suspend_always
    - suspend_never
- 与协程进行交互及定制其行为的底层机制
- 目前尚未标准化直接用户可用的上层封装

使用 C++ 协程的 fibonacci

```cpp
uint64_resumable fibonacci()
{
    uint64_t a = 0;
    uint64_t b = 1;
    while (true) {
        co_yield b;
        auto tmp = a;
        a = b;
        b += tmp;
    }
}
```

```cpp
uint64_resumable res = fibonacci();
while (res.resume()) {
    auto i = res.get();
    if (i >= 10000) {
        break;
    }
    cout << i << endl;
}
```

co_await

```
auto result = co_await expression;
```

```
auto&& __a = expression;
if (!__a.await_ready()) {
    __a.await_suspend(coroutine_handle);
    // 挂起/恢复点
}

auto result = __a.await_resume();
```

Awaitable

```cpp
template <typename T>
struct awaitable_concept {
    bool await_ready();
    void await_suspend(coroutine_handle<>);
    T await_resume();
};
```

标准类型 suspend_always

```cpp
struct suspend_always {
    bool await_ready() const noexcept
    {
        return false;
    }
    void await_suspend(coroutine_handle<>) const noexcept {}
    void await_resume() const noexcept {}
};
```

标准类型 suspend_never

```cpp
struct suspend_never {
    bool await_ready() const noexcept
    {
        return true;
    }
    void await_suspend(coroutine_handle<>) const noexcept {}
    void await_resume() const noexcept {}
};
```

协程的执行

```
frame = operator new(…);  // 分配协程帧，含 promise_type、参数、变量、状态等
promise_type& promise = frame->promise;
auto return_value = promise.get_return_object();  // 在初次挂起时返回给调用者

co_await promise.initial_suspsend();
try {
    执行协程体
    可能被 co_wait、co_yield 挂起
    恢复后继续执行
}
catch (...) {
    promise.unhandled_exception();
}

final_suspend:
    co_await promise.final_suspsend();
```

co_yield 和 co_return

co_yield

```
co_await promise.yield_value(表达式);
```

co_return

```
promise.return_value(表达式);
goto final_suspend;
```

或

```
promise.return_void();
goto final_suspend;
```

定义 uint64_resumable

```
class uint64_resumable {
public:
    struct promise_type {…};

    using coro_handle = coroutine_handle<promise_type>;
    explicit uint64_resumable(coro_handle handle) : handle_(handle) {}
    ~uint64_resumable() { handle_.destroy(); }
    uint64_resumable(const uint64_resumable&) = delete;
    uint64_resumable(uint64_resumable&&) = default;
    bool resume();
    uint64_t get();

private:
    coro_handle handle_;
};
```

resume 和 get

```cpp
bool uint64_resumable::resume()
{
    if (!handle_.done())
        handle_.resume();
    return !handle_.done();
}
```

```cpp
uint64_t uint64_resumable::get()
{
    return handle_.promise().value_;
}
```

promise_type

```
struct promise_type {
    uint64_t value_;
    using coro_handle = coroutine_handle<promise_type>;
    auto get_return_object() {
        return uint64_resumable{coro_handle::from_promise(*this)};
    }
    constexpr auto initial_suspend() { return suspend_always(); }
    constexpr auto final_suspend() { return suspend_always(); }
    auto yield_value(uint64_t value) {
        value_ = value;
        return suspend_always();
    }
    void return_void() {}
    void unhandled_exception() { std::terminate(); }
};
```

协程的生命周期

协程在下列情况之一较早发生时销毁：

- 从 `final_suspend` 恢复
- `coroutine_handle<>::destroy` 被调用

协程销毁时，已初始化的本地变量和 promise 也同时被析构；协程帧内存被释放。

```cpp
class uint64_resumable {
public:
    struct promise_type {
        uint64_t value_;
        using coro_handle = coroutine_handle<promise_type>;
        auto get_return_object()
        {
            return uint64_resumable{coro_handle::from_promise(*this)};
        }
        constexpr auto initial_suspend() { return suspend_always(); }
        constexpr auto final_suspend() { return suspend_always(); }
        auto yield_value(uint64_t value)
        {
            value_ = value;
            return suspend_always();
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    using coro_handle = coroutine_handle<promise_type>;
    explicit uint64_resumable(coro_handle handle) : handle_(handle) {}
    ~uint64_resumable() { handle_.destroy(); }
    uint64_resumable(const uint64_resumable&) = delete;
    uint64_resumable(uint64_resumable&&) = default;
    bool resume();
    uint64_t get();

private:
    coro_handle handle_;
};

bool uint64_resumable::resume()
{
    if (!handle_.done())
        handle_.resume();
    return !handle_.done();
}

uint64_t uint64_resumable::get()
{
    return handle_.promise().value_;
}
```

```cpp
class uint64_resumable {
public:
    struct promise_type {
        uint64_t value_;
        using coro_handle = coroutine_handle<promise_type>;
        auto get_return_object()
        {
            return uint64_resumable{coro_handle::from_promise(*this)};
        }
        constexpr auto initial_suspend() { return suspend_always(); }
        constexpr auto final_suspend() { return suspend_always(); }
        auto yield_value(uint64_t value)
        {
            value_ = value;
            return suspend_always();
        }
        void return_void() {}
        void unhandled_exception() { std::terminate(); }
    };

    using coro_handle = coroutine_handle<promise_type>;
    explicit uint64_resumable(coro_handle handle) : handle_(handle) {}
    ~uint64_resumable() { handle_.destroy(); }
    uint64_resumable(const uint64_resumable&) = delete;
    uint64_resumable(uint64_resumable&&) = default;
    bool resume();
    uint64_t get();

private:
    coro_handle handle_;
};

bool uint64_resumable::resume()
{
    if (!handle_.done())
        handle_.resume();
    return !handle_.done();
}

uint64_t uint64_resumable::get()
{
    return handle_.promise().value_;
}
```

generator 示例（CppCoro 和 MSVC）

```
generator<const uint64_t> fibonacci()
{
    uint64_t a = 0;
    uint64_t b = 1;
    while (true) {
        co_yield b;
        auto tmp = a;
        a = b;
        b += tmp;
    }
}
```

future 示例（仅 MSVC）

```cpp
future<int> compute_value()
{
    int result = co_await async([] { return 42; });
    co_return result;
}

int main()
{
    auto value = compute_value();
    cout << value.get() << endl;
}
```

Subroutine vs Coroutine

	Subroutine	**Coroutine**
调用	`foo(…);`	`foo(…);`
返回	`return …;`	`co_return …;`
挂起		`co_await …;`
恢复		`coroutine_handle<>::resume()`

代码

https://github.com/adah1972/cpp_summit_2019

- fibonacci.py, pp. 5–6
- fibonacci.hpp, pp. 10–11
- fibonacci.cpp, p. 12
- fibonacci_range_v3.cpp, pp. 13–14
- fibonacci_ranges.cpp, pp. 13–14
- fibonacci_coroutines_ts.cpp, pp. 19, 26–28
- fibonacci_cppcoro_generator.cpp, p. 32
- fibonacci_msvc_generator.cpp, p. 32
- await_msvc_future.cpp, p. 33

参考资料

- Gor Nishanov, “Working draft, C++ extensions for coroutines”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf
- Lewis Baker, CppCoro. https://github.com/lewissbaker/cppcoro
- Lewis Baker, “Coroutine theory”. https://lewissbaker.github.io/2017/09/25/coroutine-theory
- Lewis Baker, “C++ coroutines: understanding operator `co_await`”. https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await
- Dawid Pilarski, “Coroutines introduction”. https://blog.panicsoftware.com/coroutines-introduction/
- 吴咏炜, “Python `yield` and C++ coroutines”. https://yongweiwu.wordpress.com/2016/08/16/python-yield-and-cplusplus-coroutines/
- Oliver Kowalke, Boost.Coroutine2. http://www.boost.org/doc/libs/release/libs/coroutine2/
- Jamboree, CO2. https://github.com/jamboree/co2

备用材料

Boost.Coroutine2 示例

```
typedef boost::coroutines2::coroutine<const uint64_t> coro_t;

void fibonacci(coro_t::push_type& yield)
{
    uint64_t a = 0;
    uint64_t b = 1;
    while (true) {
        yield(b);      } 挂起点
        auto tmp = a;
        a = b;
        b += tmp;
    }
}
```

使用 fibonacci

```
for (auto i : coro_t::pull_type(
            boost::coroutines2::fixedsize_stack(),
            fibonacci)) {
    if (i >= 10000) {
        break;
    }
    std::cout << i << std::endl;
}
```

从进程到“纤程”

- 系统中一般同时存在多个进程（process）
  - 每个进程可以有多个线程（thread）
    - 每个线程可以有多个纤程（fiber）
    - 每个纤程有自己的栈（stack）
    - Stackful coroutine = goroutine = fiber

Stackful vs Stackless

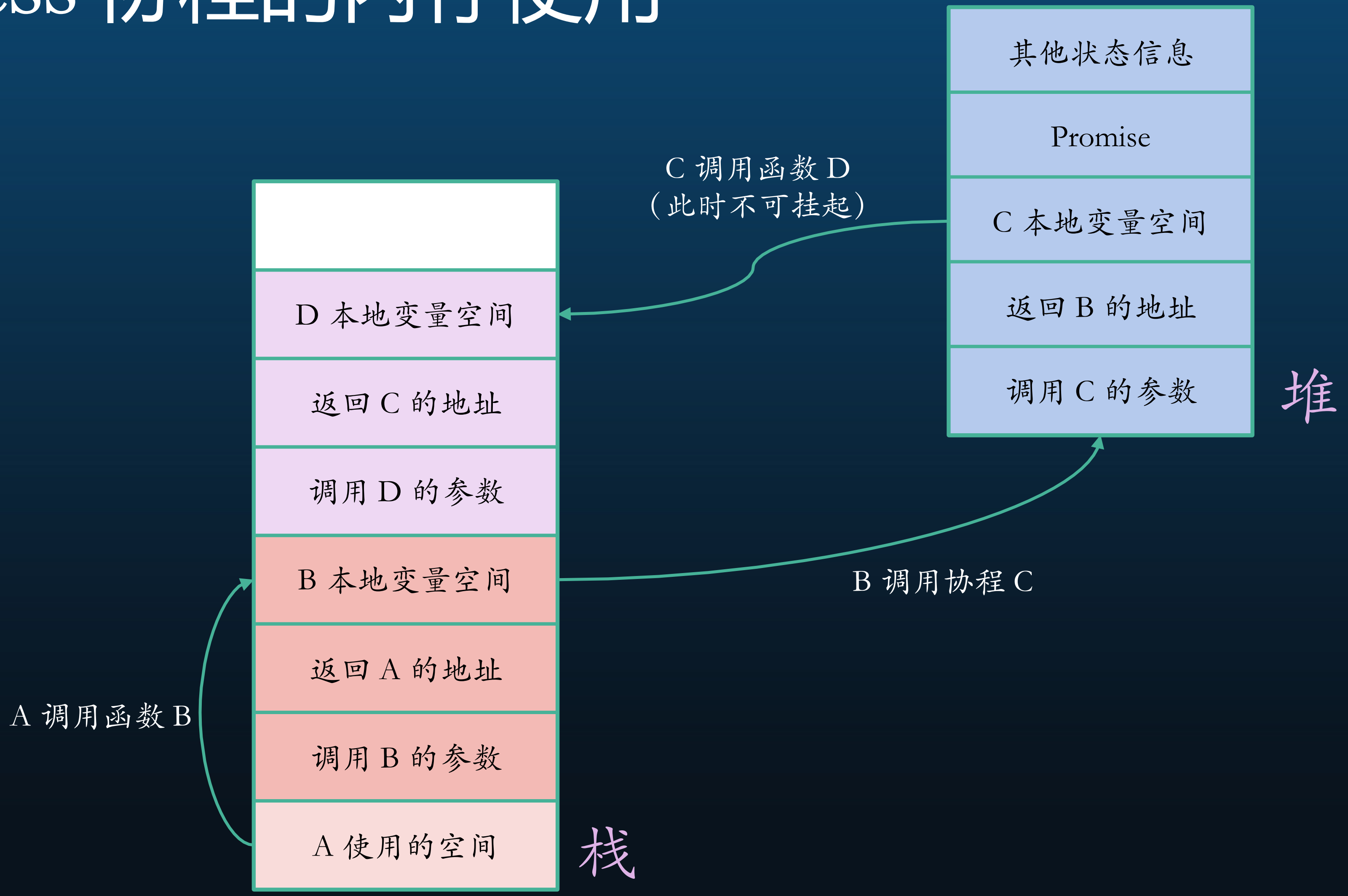
Stackful

- 创建协程需要分配栈帧（stack frame）
- 协程挂起/恢复需要切换栈帧（较慢）
- 使用内存较多（可能只能起数千个）
- 允许从任意栈帧挂起

Stackless

- 创建协程需要分配协程帧（coroutine frame）
- 协程挂起/恢复相当于正常函数调用
- 使用内存较少，允许海量协程（可达亿级）
- 只能从协程的顶层函数挂起

Stackless 协程的内存使用



CppCoro 和 Ranges

```
auto&& fib = fibonacci();
for (auto i : fib | ranges::views::take(20)) {
    cout << i << endl;
}
```

Ranges 的管道符的左侧需要满足 View 概念；或者是左值并满足 Range。