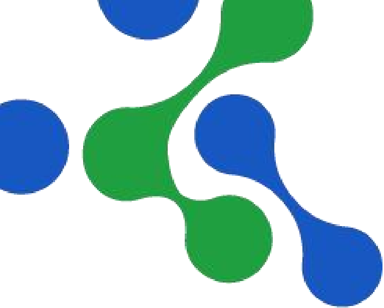
开放·连接·预见

2021 K+

全球软件研发行业创新峰会

深入解析C++20协程

主讲人：谢丙堃



目 录

Contents

01.协程的基础介绍

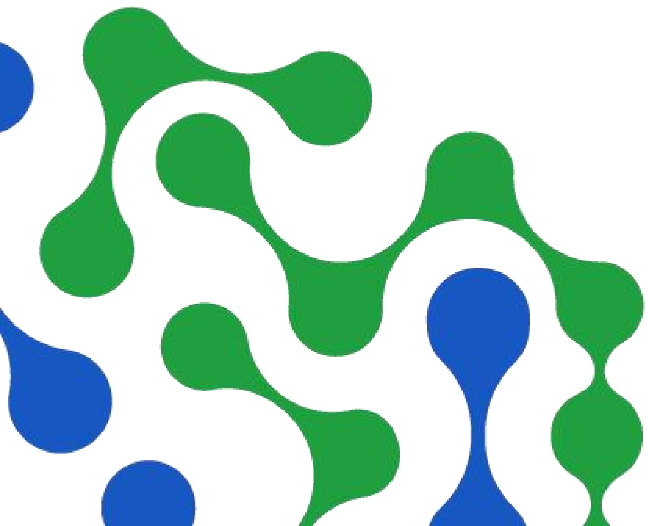
02.协程的使用方法和原理

03.stackless和stackful两种协程实现机制

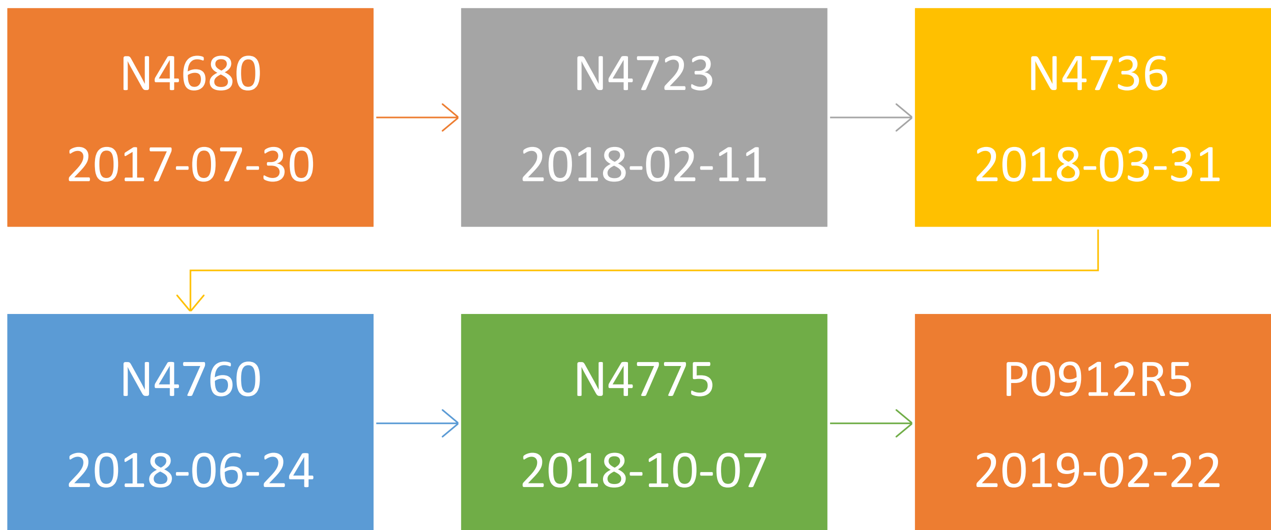
04.目前协程实现的性能问题

01

协程的基础介绍



C++协程的发展



C++协程的发展

- Visual Studio 2015 Update 2 （2016.3）
- CLANG 8 （2019.4）
- GCC 10 （2020.1）

协程的定义和动机

- 协程是一种可以被挂起和恢复的函数：
  - 先挂起执行，
  - 随后恢复执行。

协程的定义和动机

```
std::generator<int> foo()
{
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}

std::future<void> bar()
{
    auto n = co_await std::async(foo);
    co_return;
}
```

挂起点

挂起点

VS stl::experimental

协程的动机

- 协程提供这些方法：
  - 便捷的创建异步函数；
  - 创建生成器；
  - 懒惰计算；
  - 事件驱动程序。

co_yield 执行流

```
int main()
{
    for (auto i : foo()) {
        std::cout << i << std::endl;
        // ...
    }
}
```

```
std::generator<int> foo()
{
    std::cout << "begin" << std::endl;
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
    std::cout << "end" << std::endl;
}
```

co_yield 执行流

```cpp
int main()
{
    for (auto i : foo()) {
        std::cout << i << std::endl;
        // ...
    }
}
```

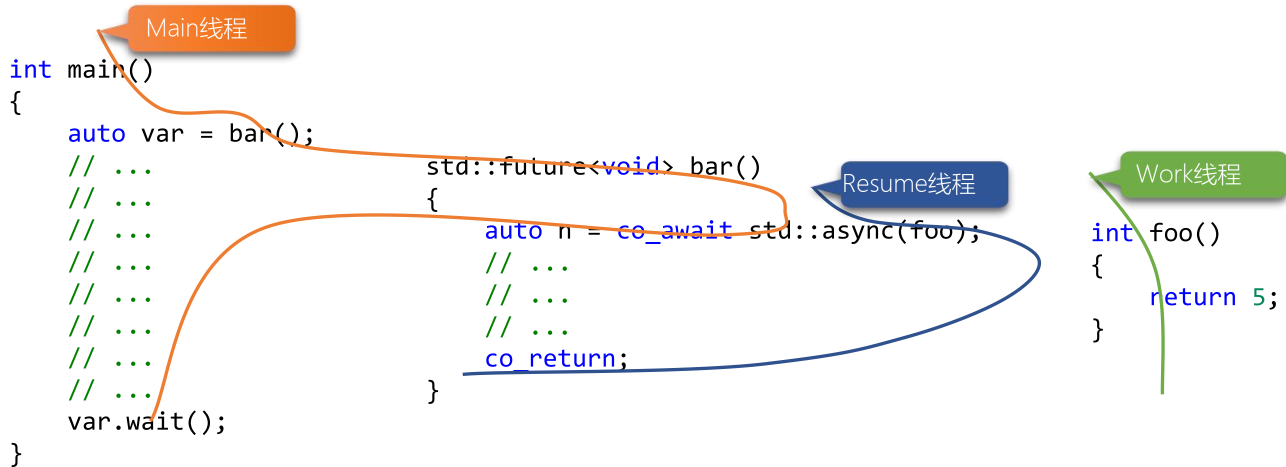
```cpp
std::generator<int> foo()
{
    std::cout << "begin" << std::endl;
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
    std::cout << "end" << std::endl;
}
```

co_yield 执行流

```
int main()
{
    for (auto i : foo()) {
        std::cout << i << std::endl;
        // ...
    }
}
```

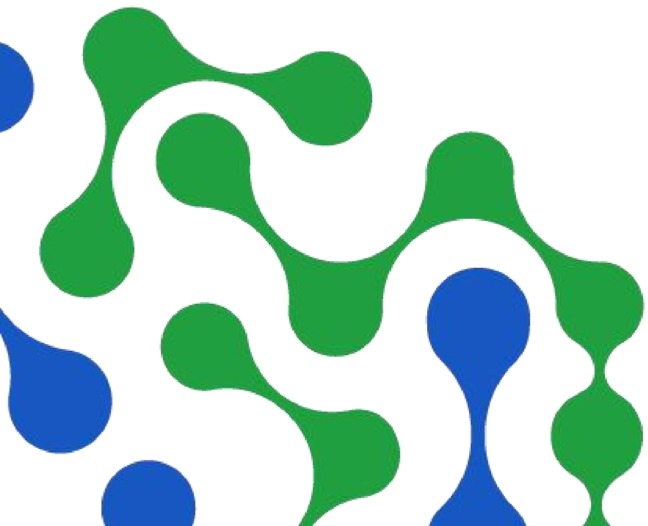
```
std::generator<int> foo()
{
    std::cout << "begin" << std::endl;
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
    std::cout << "end" << std::endl;
}
```

co_await 执行流



02

协程的使用方法和原理



可等待体和等待器

- 并不是所有对象都可等待：

```
co_await std::string{ "hello" };
```

```
error C3312: no callable 'await_resume' function found for type 'std::string'
error C3312: no callable 'await_ready' function found for type 'std::string'
error C3312: no callable 'await_suspend' function found for type 'std::string'
```

可等待体和等待器

- 可被等待的对象称为可等待(awaitable)体或者表达式；
- co_await运算符必须返回一个等待器(awaiter)：
  - 可等待体和等待器可以是同一个类型；
  - std::future（实验）是可等待体。
  - co_await运算符返回等待器_Future_awaiter

可等待体和等待器

```
template <class _Ty>
auto operator co_await(future<_Ty>&& _Fut) {
    return experimental::_Future_awaiter<_Ty>{_Fut};
}


template <class _Ty>
auto operator co_await(future<_Ty>& _Fut) {
    return experimental::_Future_awaiter<_Ty>{_Fut};
}
```

可等待体和等待器

```
template <class _Ty>
struct _Future_awaiter {
    future<_Ty>& _Fut;

    bool await_ready() const {
        return _Fut._Is_ready();
    }

    void await_suspend(experimental::coroutine_handle<> _ResumeCb) {
        // TRANSITION, change to .then if and when future gets .then
        thread _WaitingThread([&_Fut = _Fut, _ResumeCb]() mutable {
            _Fut.wait();
            _ResumeCb();
        });
        _WaitingThread.detach();
    }

    decltype(auto) await_resume() {
        return _Fut.get();
    }
};
```

判定可等待体是否已经准备好

调度协程的执行流程

接收异步执行结果

await_suspend的三种形式

1. 返回void类型表示协程需要将执行流的控制权交给调用者，协程保持挂起状态。
2. 返回bool类型则又会出现两种情况：
   - 当返回true时效果和返回类型为void相同
   - 当返回false时则恢复当前协程运行。
3. 返回coroutine_handle类型的时候则会恢复该句柄对应的协程：
   - coroutine_handle是协程的句柄，可以用于控制协程的运行流程；
   - 由编译期构造，通过调用编译内置函数__builtin_coro_resume恢复协程执行。

协程的返回类型

- std::future和std::generator都可以作为返回类型，但是有所不同：
  - std::future可以等待，可以从协程返回，但是无法恢复协程；
  - Std::generator可以从协程返回，可以恢复协程，但是无法等待。

协程的返回类型

- 协程对返回类型有特殊要求：

```
struct my_int_generator {};

my_int_generator foo()
{
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}
```

```
error C2039: 'promise_type': is not a member of 'std::experimental::coroutine_traits<my_int_generator>'
```

promise_type

- 返回类型具有内嵌类型promise_type；
- 或者有特化版本std::experimental::coroutine_traits<T>具有内嵌类型promise_type。

```
template <class... _ArgTypes>
struct coroutine_traits<future<void>, _ArgTypes...> {
    struct promise_type {
        // ...
    };
};
```

promise_type

```
struct my_int_generator {
    struct promise_type {
        int* value_ = nullptr;
        my_int_generator get_return_object() {
            return my_int_generator{ *this };
        }
        // ...
    };

    explicit my_int_generator(promise_type& p)
        : handle_(coroutine_handle<promise_type>::from_promise(p)) {}
    // ...
    coroutine_handle<promise_type> handle_;
};

my_int_generator foo()
{
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}
```

promise_type

```
struct my_int_generator {
    struct promise_type {
        int* value_ = nullptr;
        // ...
        auto initial_suspend() const noexcept {
            return suspend_always{};
        }
        auto final_suspend() const noexcept {
            return suspend_always{};
        }
        auto yield_value(int& value) {
            value_ = &value;
            return suspend_always{};
        }
    };

    // ...
};
```

协程执行前后的挂起机会，相当于：
co_await promist_type.initial_suspend();
…
co_await promist_type.final_suspend();

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

保存co_yield操作数的值并且返回等待器

完成my_int_generator

```
struct my_int_generator {
    ...
    int next() {
        if (!handle_ || handle_.done()) {
            return -1;
        }
        handle_();
        return handle_.promise().value_;
    }
    ...
};

int main()
{
    auto obj = foo();
    std::cout << obj.next() << std::endl;
    std::cout << obj.next() << std::endl;
    std::cout << obj.next() << std::endl;
}
```

也可以实现一个迭代器，具体参考
experimental/generator

再谈promise_type

```
struct promise_type {
    int value_ = 0;
    // ...

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

    void return_void() {}
}
```

co_return调用return_value或者return_void

再谈promise_type

```
struct promise_type {
    // ...
    awaitable await_transform(expr e) {
        return awaitable(e);
    }
    // ...
    void unhandled_exception() {
        eptr_ = std::current_exception();
    }
};
```

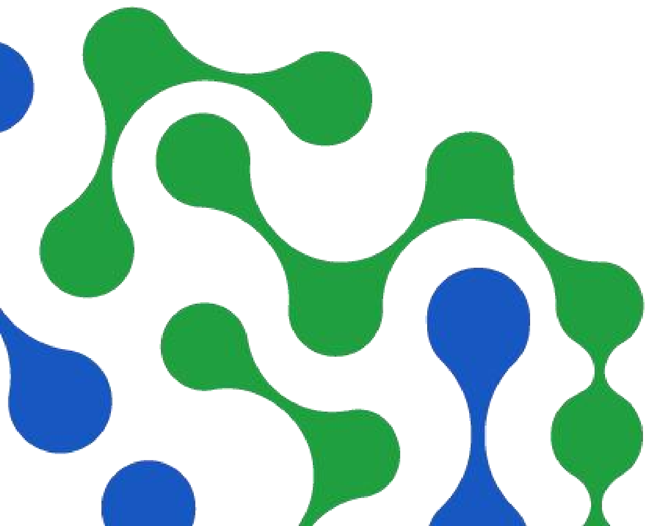
对co_await的操作数进行转换处理:
FROM:
co_await expr;
TO:
co_await promise.await_transform(expr);

对协程中的异常进行处理

```
co_await promise.initial_suspend();
Try {
    // ...
}
catch (...) {
    promise.unhandled_exception();
}
co_await promise.final_suspend();
```

03

stackless和stackful两种协程实现机制



stackless vs stackful

- 有栈协程和无栈协程并不是指协程在运行中是否有存在栈，
- 而是指的协程是否可以从嵌套的栈帧中挂起，并且在完全相同的栈帧上恢复执行。（Boost.Coroutine2中的定义）
- 当然也可以理解为是否每个协程都有独有的栈帧。
  - 显然，C++20的协程属于无栈协程，因为挂起和恢复的栈帧不同。

stackless vs stackful

- 显然，C++20协程属于无栈协程，因为挂起和恢复的栈帧不同。

挂起时	恢复时

```
coroutine_test.exe!foo$_ResumeCoro$1()
coroutine_test.exe!foo$_InitCoro$2()
coroutine_test.exe!foo() [Ramp]
coroutine_test.exe!main()
coroutine_test.exe!invoke_main()
coroutine_test.exe!__scrt_common_main_seh()
coroutine_test.exe!__scrt_common_main()
coroutine_test.exe!mainCRTStartup(void * __formal)
kernel32.dll!BaseThreadInitThunk()
ntdll.dll!RtlUserThreadStart()
```

```
coroutine_test.exe!foo$_ResumeCoro$1()
coroutine_test.exe!std::experimental::coroutine_handle<void>::re
coroutine_test.exe!std::experimental::coroutine_handle<void>::op
coroutine_test.exe!`awaitable_string::await_suspend'::`2'::<lamb
coroutine_test.exe!std::invoke<`awaitable_string::await_suspend'
mbda_1> && _Obj)
coroutine_test.exe!std::thread::_Invoke<std::tuple<`awaitable_st
ucrtbased.dll!thread_start<unsigned int (__cdecl*)(void *),1>(vo
kernel32.dll!BaseThreadInitThunk()
ntdll.dll!RtlUserThreadStart()
```

stackful

- 有栈协程的典型代表如：goroutine、fibers。
  - 有栈协程因为有自己的栈内存，所以很多时候被称为纤程；
  - 它的栈帧切换不需要进入系统内核，相对于线程非常轻量级；
  - 相对于无栈协程最大的开销也是来自于栈帧的切换。
- 如goroutine可以在32位进程上轻松跑出上百万的协程。

stackful

- 有栈协程的实现关键是：切换栈和CPU寄存器上下文：
  - 因为函数的局部变量、参数、函数调用都保持在栈上；
  - 运行的当前状态都在CPU寄存器上。
- 比较易于学习的开源代码：
  - https://swtch.com/libtask/
  - https://code.google.com/archive/p/libconcurrency/

stackful

- 有栈协程切换上下文的实现方法主要分为三种：
    - ucontext；(e.g. *boost.coroutine2)
    - setjmp/longjmp；(e.g. libconcurrency)
    - 汇编实现上下文切换。(e.g. libtask)
- 需要注意，Windows一般使用Windows Fibers API。

stackful

- 有栈协程性能很好，但是很少有工业基本的应用，原因是：
  - 只是实现上下文切换并不能真正发挥异步优势；
  - 需要与系统异步IO（文件、网络等）高度整合；
  - 在现有开源库的基础上整合异步IO也并不容易。
- 工业级别的有栈协程开源代码（网络接口HOOK）：
  - https://github.com/Tencent/libco
  - https://github.com/yyzybb537/libgo

stackless

- C++20 coroutines是典型无栈协程：
    - 因为不是每个协程都独占一个栈，所以不可能做到在任何条件下挂起和恢复协程；
    - 但是由于没有栈和CPU寄存器上下文的切换，性能上要更好；
    - C++20 coroutines提供了非常灵活和系统异步IO整合的方法，未来有更多可能。
- 目前比较完善的支持库：
    - https://github.com/lewissbaker/cppcoro
    - https://think-async.com/Asio/

stackless

- 无栈协程的实现更像是一个状态机，以generator为例：

```
int main()
{
    for (auto i : foo()) {
        std::cout << i << std::endl;
        // ...
    }
}
```

```
std::generator<int> foo()
{
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}
```

stackless

- 无栈协程的实现更像是一个状态机，以generator为例：
  - 代码来自：https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

```c
int foo() {
  static int i, state = 0;
  switch (state) {
    case 0:
      goto LABEL0;
    case 1:
      goto LABEL1;
  }
LABEL0:
  for (i = 0; i < 10; i++) {
    state = 1;
    return i;
  LABEL1:;
  }
}
```

```c
int foo() {
  static int i, state = 0;
  switch (state) {
    case 0:
      for (i = 0; i < 10; i++) {
        state = 1;
        return i;
    case 1:;
      }
  }
}
```

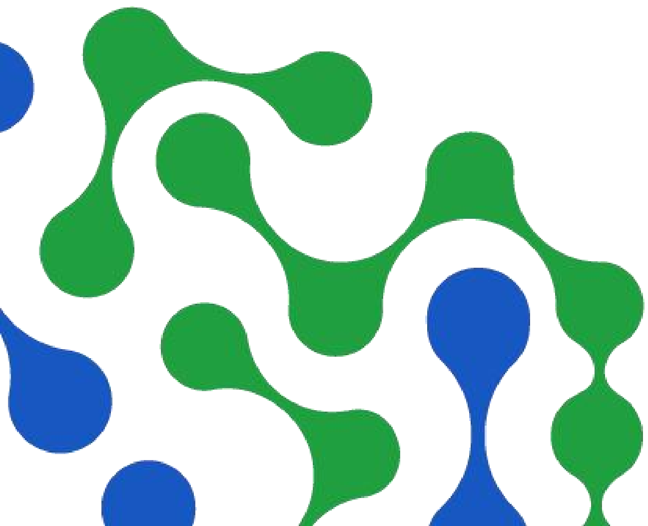
stackless

- 最终版本：
  - 简直是场灾难，但是原理上却是这样的。

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(i,x) do { state=i; return x; case i:; } while (0)
#define crFinish }
int foo() {
    static int i;
    crBegin;
    for (i = 0; i < 10; i++)
        crReturn(1, i);
    crFinish;
}
```

04

目前协程实现的性能问题



微软提供实验future的性能问题

```
template <class _Ty>
struct _Future_awaiter {
    future<_Ty>& _Fut;

    bool await_ready() const {
        return _Fut._Is_ready();
    }

    void await_suspend(experimental::coroutine_handle<> _ResumeCb) {
        // TRANSITION, change to .then if and when future gets .then
        thread _WaitingThread([&_Fut = _Fut, _ResumeCb]() mutable {
            _Fut.wait();
            _ResumeCb();
        });
        _WaitingThread.detach();
    }

    decltype(auto) await_resume() {
        return _Fut.get();
    }
};
```

std::async的返回值，微软STL使用的线程池（PPL），GCC和CLANG直接创建新线程。

除了调用线程之外，还创建了工作线程和恢复线程，创建线程性能损耗大。

性能优化的方法讨论

- 直观的优化：
  - 减少线程创建，合并工作线程和恢复线程；
  - 使用线程池，不创建额外线程。

```
template <class _Ty>
struct _Task_awaiter {
    task<_Ty>& _Task;
    // ...
    void await_suspend(experimental::coroutine_handle<> _ResumeCb) {
        threadpool::post([&_Task = _Task, _ResumeCb]() mutable {
            _Task.run();
            _ResumeCb();
        });
    }
    // ...
};
```

性能优化的方法讨论

- 使用异步IO充分利用CPU资源：
  - 对于计算密集型程序，增加协程并不能有效提高性能；
  - 对于IO密集型程序，需要协程整合异步IO来提供性能。例如：
    - 读写文件，磁盘IO使用DMA，不需要CPU参与；
    - 网络传输，同样不需要CPU参与。

Demo

- Windows系统有非常完善的异步IO系统：
  - 利用内核提供的异步IRP和APC，我们可以在单线程中完成异步读写文件的操作。
  - 我们需要用到：
    - Overlapped
    - ReadFileEx
    - Alertable wait
    - std::future

Demo

```
class AsyncFileIO {
public:
  // ...
  bool Create(const char *filename, unsigned long flag) {
    HANDLE handle =
        CreateFileA(filename, GENERIC_READ | GENERIC_WRITE, 0, nullptr, flag,
              FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, nullptr);

    bool retval = handle != INVALID_HANDLE_VALUE;
    if (retval) {
      shared_handle_.reset(handle, CloseHandle);
    }

    return retval;
  }

  ReadAwaiter Read(unsigned long offset, unsigned long length) {
    return ReadAwaiter(shared_handle_, offset, length);
  }

private:
  std::shared_ptr<void> shared_handle_;
};
```

使用异步IO打开文件。

读取时创建等待器对象，这是一个懒惰读，在co_await的时候发生读取。

Demo

```
class ReadAwaiter {
public:
  // ...
  bool await_ready() const { return false; }
  void await_suspend(std::experimental::coroutine_handle<> h) {
    OVERLAPPED o{0};
    o.hEvent = this;
    o.Offset = offset_;
    result_.resize(length_);
    coro_ = h;
    BOOL retval = ReadFileEx(shared_handle_.get(), result_.data(), length_, &o,
                             OverlappedCompletionRoutine);
    if (!retval) {
      result_.clear();
      h();
    }
  }
  std::vector<unsigned char> await_resume() const { return result_; }
  void resume() { coro_(); }
  void resize_result(const size_t real_number) {
    if (real_number != result_.size()) {
      result_.resize(real_number);
    }
  }
  // ...
};
```

使用ReadFileEx异步读取文件。

读取完成后在调用者线程插入APC回调函数。

如果调用失败，则立即恢复协程执行。

Demo

```
static void OverlappedCompletionRoutine(unsigned long err_code,
                                        unsigned long transfered_number,
                                        LPOVERLAPPED overlaped) {
  ReadAwaiter *awaiter = static_cast<ReadAwaiter *>(overlaped->hEvent);
  if (awaiter) {
    awaiter->resize_result(transfered_number);
    awaiter->resume();
  }
}
```

在回调函数中恢复协程执行

Demo

```
std::future<void> ReadTestFile() {
  AsyncFileIO reader;
  reader.Create(R"(D:\some_data.bin)", OPEN_EXISTING);
  auto vec = co_await reader.Read(0, 100 * 1024 * 1024);

  std::cout << vec.size();
}

int main() {
  auto f = ReadTestFile();

  SleepEx(INFINITE, TRUE);
  f.wait();
}
```

co_await执行文件读取。

这里直接使用f.wait();不行，需要使用alertable wait。

Windows API SleepEx能够执行警醒等待，除此之外 MsgWaitForMultipleObjectsEx， WaitForSingleObjectEx，WaitForMultipleObjectsEx 等API都可以执行警醒等待。

参考文献

1. Working Draft, Standard for Programming Language C++(N4800) [http://www2.open-std.org/JTC1/SC22/WG21/docs/papers/2019/n4800.pdf]
2. C++ compiler support [https://en.cppreference.com/w/cpp/compiler_support]
3. Coroutines in C [https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html]
4. Synchronous and Asynchronous I/O [https://docs.microsoft.com/en-us/windows/win32/fileio/synchronous-and-asynchronous-i-o]
5. Boost.Coroutine2 Introduction [https://www.boost.org/doc/libs/1_77_0/libs/coroutine2/doc/html/coroutine2/intro.html]
6. Libgo source code [https://github.com/yyzybb537/libgo]
7. Libconcurrency source code [https://code.google.com/archive/p/libconcurrency/]
8. Libtask source code [https://swtch.com/libtask/]

THANKS