30讲Coroutines：协作式的交叉调度执行

你好，我是吴咏炜。

今天是我们未来篇的最后一讲，也是这个专栏正文内容的最后一篇了。我们讨论 C++20 里的又一个非常重要的新功能——协程 Coroutines。

什么是协程?

协程是一个很早就被提出的编程概念。根据高德纳的描述，协程的概念在 1958 年就被提出了。不过，它在主流编程语言中得到的支持不那么好，因而你很可能对它并不熟悉吧。

如果查阅维基百科，你可以看到下面这样的定义 [1]：

> 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……

等学完了这一讲，也许你可以明白这段话的意思。但对不了解协程的人来说，估计只能吐槽一句了，这是什么鬼?

很遗憾，在 C++ 里的标准协程有点小复杂。我们还是从……Python 开始。

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

即使你没学过 Python，上面这个生成斐波那契数列的代码应该也不难理解。唯一看起来让人会觉得有点奇怪的应该就是那个 `yield` 了。这种写法在 Python 里叫做“生成器”（generator），返回的是一个可迭代的对象，每次迭代就能得到一个 yield 出来的结果。这就是一种很常见的协程形式了。

如何使用这个生成器，请看下面的代码：

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

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

这些代码很容易理解：`islice` 相当于 [第 29 讲] 中的 `take`，取一个范围的头若干项；`takewhile` 则在范围中逐项取出内容，直到第一个参数的条件不能被满足。两个函数的结果都可以被看作是 C++ 中的视图。

我们唯一需要提的是，在代码的执行过程中，`fibonacci` 和它的调用代码是交叉执行的。下面我们用代码行加注释的方式标一下：

```
a = 0  # fibonacci()
b = 0  # fibonacci()
yield b  # fibonacci()
print(i)  # 调用者
a, b = 1, 0 + 1  # fibonacci()
yield b  # fibonacci()
print(i)  # 调用者
a, b = 1, 1 + 1  # fibonacci()
yield b  # fibonacci()
print(i)  # 调用者
a, b = 2, 1 + 2  # fibonacci()
yield b  # fibonacci()
print(i)  # 调用者
…
```

学到这儿的同学应该都知道我们在 C++ 里怎么完成类似的功能吧？我就不讲解了，直接给出可工作的代码。这是对应的 `fibonacci` 的定义：

```
#include <iterator>
#include <stddef.h>
#include <stdint.h>

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

class fibonacci::sentinel {};

class fibonacci::iterator {
public:
  // Required to satisfy iterator
  // concept
  typedef ptrdiff_t difference_type;
  typedef uint64_t value_type;
  typedef const uint64_t* pointer;
  typedef const uint64_t& reference;
  typedef std::input_iterator_tag
    iterator_category;

  value_type operator*() const
  {
    return b_;
  }
  pointer operator->() const
  {
    return &b_;
  }
  iterator& operator++()
  {
    auto tmp = a_;
    a_ = b_;
    b_ += tmp;
    return *this;
  }
```

```
  iterator operator++(int)
  {
    auto tmp = *this;
    ++*this;
    return tmp;
  }
  bool
  operator==(const sentinel&) const
  {
    return false;
  }
  bool
  operator!=(const sentinel&) const
  {
    return true;
  }

private:
  uint64_t a_{0};
  uint64_t b_{1};
};

// sentinel needs to be
// equality_comparable_with iterator
bool operator==(
  const fibonacci::sentinel& lhs,
  const fibonacci::iterator& rhs)
{
  return rhs == lhs;
}
bool operator!=(
  const fibonacci::sentinel& lhs,
  const fibonacci::iterator& rhs)
{
  return rhs != lhs;
}

inline fibonacci::iterator
fibonacci::begin() noexcept
{
  return iterator();
}

inline fibonacci::sentinel
```

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

调用代码跟 Python 的相似：

```
// 打印头 20 项
for (auto i :
     fibonacci() | take(20)) {
  cout << i << endl;
}

// 打印小于 10000 的数列项
for (auto i :
     fibonacci() |
       take_while([](uint64_t x) {
         return x < 10000;
       })) {
  cout << i << endl;
}
```

这似乎还行。但 `fibonacci` 的定义差异就大了：在 Python 里是 6 行有效代码，在 C++ 里是 53 行。C++ 的生产率似乎有点低啊……

C++20 协程

C++20 协程的基础是微软提出的 Coroutines TS（可查看工作草案[2]），它在 2019 年 7 月被批准加入到 C++20 草案中。目前，MSVC 和 Clang 已经支持协程。不过，需要提一下的是，目前被标准化的只是协程的底层语言支持，而不是上层的高级封装；稍后，我们会回到这个话题。

协程可以有很多不同的用途，下面列举了几种常见情况：

- 生成器
- 异步 I/O
- 惰性求值
- 事件驱动应用

这一讲中，我们主要还是沿用生成器的例子，向你展示协程的基本用法。异步 I/O 应当在协程得到广泛采用之后，成为最能有明显收益的使用场景；但目前，就我看到的，只有 Windows 平台上有较好的支持——微软目前还是做了很多努力的。

回到 Coroutines。我们今天采用 Coroutines TS 中的写法，包括`std::experimental` 名空间，以确保你可以在 MSVC 和 Clang 下编译代码。首先，我们看一下协程相关的新关键字，有下面三个：

- `co_await`
- `co_yield`

- co_return

这三个关键字最初是没有 co_ 前缀的，但考虑到 await、yield 已经在很多代码里出现，就改成了目前这个样子。同时，return 和 co_return 也作出了明确的区分：一个协程里只能使用 co_return，不能使用 return。这三个关键字只要有一个出现在函数中，这个函数就是一个协程了——从外部则看不出来，没有用其他语言常用的 async 关键字来标记（async 也已经有其他用途了，见 [第 19 讲]）。C++ 认为一个函数是否是一个协程是一个实现细节，不是对外接口的一部分。

我们看一下用协程实现的 fibonacci 长什么样子：

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

这个形式跟 Python 的非常相似了吧，也非常简洁。我们稍后再讨论 uint64_resumable 的定义，先看一下调用代码的样子：

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

这个代码也非常简单，但我们需要留意 resume 和 get 两个函数调用——这就是我们的 uint64_resumable 类型需要提供的接口了。

co_await、co_yield、co_return 和协程控制

在讨论该如何定义 uint64_resumable 之前，我们需要先讨论一下协程的这三个新关键字。

首先是 co_await。对于下面这样一个表达式：

```
auto result = co_await 表达式;
```

编译器会把它理解为：

```
auto&& __a = 表达式;
if (!__a.await_ready()) {
  __a.await_suspend(协程句柄);
  // 挂起/恢复点
}
auto result = __a.await_resume();
```

也就是说，“表达式”需要支持 `await_ready`、`await_suspend` 和 `await_resume` 三个接口。如果 `await_ready()` 返回真，就代表不需要真正挂起，直接返回后面的结果就可以；否则，执行 `await_suspend` 之后即挂起协程，等待协程被唤醒之后再返回 `await_resume()` 的结果。这样一个表达式被称作是个 awaitable。

标准里定义了两个 awaitable，如下所示：

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

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

也就是说，`suspend_always` 永远告诉调用者需要挂起，而 `suspend_never` 则永远告诉调用者不需要挂起。两者的 `await_suspend` 和 `await_resume` 都是平凡实现，不做任何实际的事情。一个 awaitable 可以自行实现这些接口，以定制挂起之前和恢复之后需要执行的操作。

上面的 `coroutine_handle` 是 C++ 标准库提供的类模板。这个类是用户代码跟系统协程调度真正交互的地方，有下面这些

成员函数我们等会就会用到：

- `destroy`：销毁协程
- `done`：判断协程是否已经执行完成
- `resume`：让协程恢复执行
- `promise`：获得协程相关的 promise 对象（和 [第 19 讲] 中的“承诺量”有点相似，是协程和调用者的主要交互对象；一般类型名称为 `promise_type`）
- `from_promise`（静态）：通过 promise 对象的引用来生成一个协程句柄

协程的执行过程大致是这个样子的：

1. 为协程调用分配一个协程帧，含协程调用的参数、变量、状态、promise 对象等所需的空间。
2. 调用 `promise.get_return_object()`，返回值会在协程第一次挂起时返回给协程的调用者。
3. 执行 `co_await promise.initial_suspsend()`；根据上面对 `co_await` 语义的描述，协程可能在此第一次挂起（但也可能此时不挂起，在后面的协程体执行过程中挂起）。
4. 执行协程体中的语句，中间可能有挂起和恢复；如果期间发生异常没有在协程体中处理，则调用 `promise.unhandled_exception()`。
5. 当协程执行到底，或者执行到 `co_return` 语句时，会根据是否有非 void 的返回值，调用 `promise.return_value(…)` 或 `promise.return_void()`，然后执行 `co_await promise.final_suspsend()`。

用代码可以大致表示如下：

```
  frame = operator new(…);
  promise_type& promise =
    frame->promise;

  // 在初次挂起时返回给调用者
  auto return_value =
    promise.get_return_object();

  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，直到 co_return;
  }
  catch (...) {
    promise.unhandled_exception();
  }

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

上面描述了 co_await 和 co_return，那 co_yield 呢？也很简单，co_yield 表达式 等价于：

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

定义 uint64_resumable

了解了上述知识之后，我们就可以展示一下 uint64_resumable 的定义了：

```
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_;
};
```

这个代码相当简单，我们的结构内部有个 promise_type（下面会定义），而私有成员只有一个协程句柄。协程构造需要一个协程句柄，析构时将使用协程句柄来销毁协程；为简单起见，我们允许结构被移动，但不可复制（以免重复调用 handle_.destroy()）。除此之外，我们这个结构只提供了调用者需要的 resume 和 get 成员函数，分别定义如下：

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

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

也就是说，`resume` 会判断协程是否已经结束，没结束就恢复协程的执行；当协程再次挂起时（调用者恢复执行），返回协程是否仍在执行中的状态。而 `get` 简单地返回存储在 promise 对象中的数值。

现在我们需要看一下 promise 类型了，它里面有很多协程的定制点，可以修改协程的行为：

```
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();
  }
};
```

简单解说一下：

- 结构里面只有一个数据成员 `value_`，存放供 `uint64_resumable::get` 取用的数值。
- `get_return_object` 是第一个定制点。我们前面提到过，调用协程的返回值就是 `get_return_object()` 的结果。我们这儿就是使用 promise 对象来构造一个 `uint64_resumable`。
- `initial_suspend` 是第二个定制点。我们此处返回 `suspend_always()`，即协程立即挂起，调用者马上得到 `get_return_object()` 的结果。
- `final_suspend` 是第三个定制点。我们此处返回 `suspend_always()`，即使执行到了 `co_return` 语句，协程仍处于挂起状态。如果我们返回 `suspend_never()` 的话，那一旦执行了 `co_return` 或执行到协程结束，协程就会被销毁，连同已初始化的本地变量和 promise，并释放协程帧内存。
- `yield_value` 是第四个定制点。我们这儿仅对 `value_` 进行赋值，然后让协程挂起（执行控制回到调用者）。
- `return_void` 是第五个定制点。我们的代码永不返回，这儿无事可做。
- `unhandled_exception` 是第六个定制点。我们这儿也不应该发生任何异常，所以我们简单地调用 `terminate` 来终结程

序的执行。

好了，这样，我们就完成了协程相关的所有定义。有没有觉得轻松点？

没有？那就对了。正如我在这一节开头说的，C++20 标准化的只是协程的底层语言支持（我上面还并不是一个非常完整的描述）。要用这些底层直接写应用代码，那是非常痛苦的事。这些接口的目标用户实际上也不是普通开发者，而是库的作者。

幸好，我们并不是没有任何高层抽象，虽然这些实现不“标准”。

C++20 协程的高层抽象

cppcoro

我们首先看一下跨平台的 cppcoro 库 [3]，它提供的高层接口就包含了 `generator`。如果使用 cppcoro，我们的 `fibonacci` 协程可以这样实现：

```
#include <cppcoro/generator.hpp>
using cppcoro::generator;

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

使用 `fibonacci` 也比刚才的代码要方便：

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

除了生成器，cppcoro 还支持异步任务和异步 I/O——遗憾的是，异步 I/O 目前只有 Windows 平台上有，还没人实现 Linux 或 macOS 上的支持。

MSVC

作为协程的先行者和 Coroutines TS 的提出者，微软在协程上做了很多工作。生成器当然也在其中：

```
#include <experimental/generator>
using std::experimental::generator;

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

微软还有一些有趣的私有扩展。比如，MSVC 把标准 C++ 的 `future` 改造成了 awaitable。下面的代码在 MSVC 下可以编译通过，简单地展示了基本用法：

```
future<int> compute_value()
{
  int result = co_await async([] {
    this_thread::sleep_for(1s);
    return 42;
  });
  co_return result;
}

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

代码中有一个地方我需要提醒一下：虽然上面 `async` 返回的是 `future<int>`，但 `compute_value` 的调用者得到的并不是这个 `future`——它得到的是另外一个独立的 `future`，并最终由 `co_return` 把结果数值填充了进去。

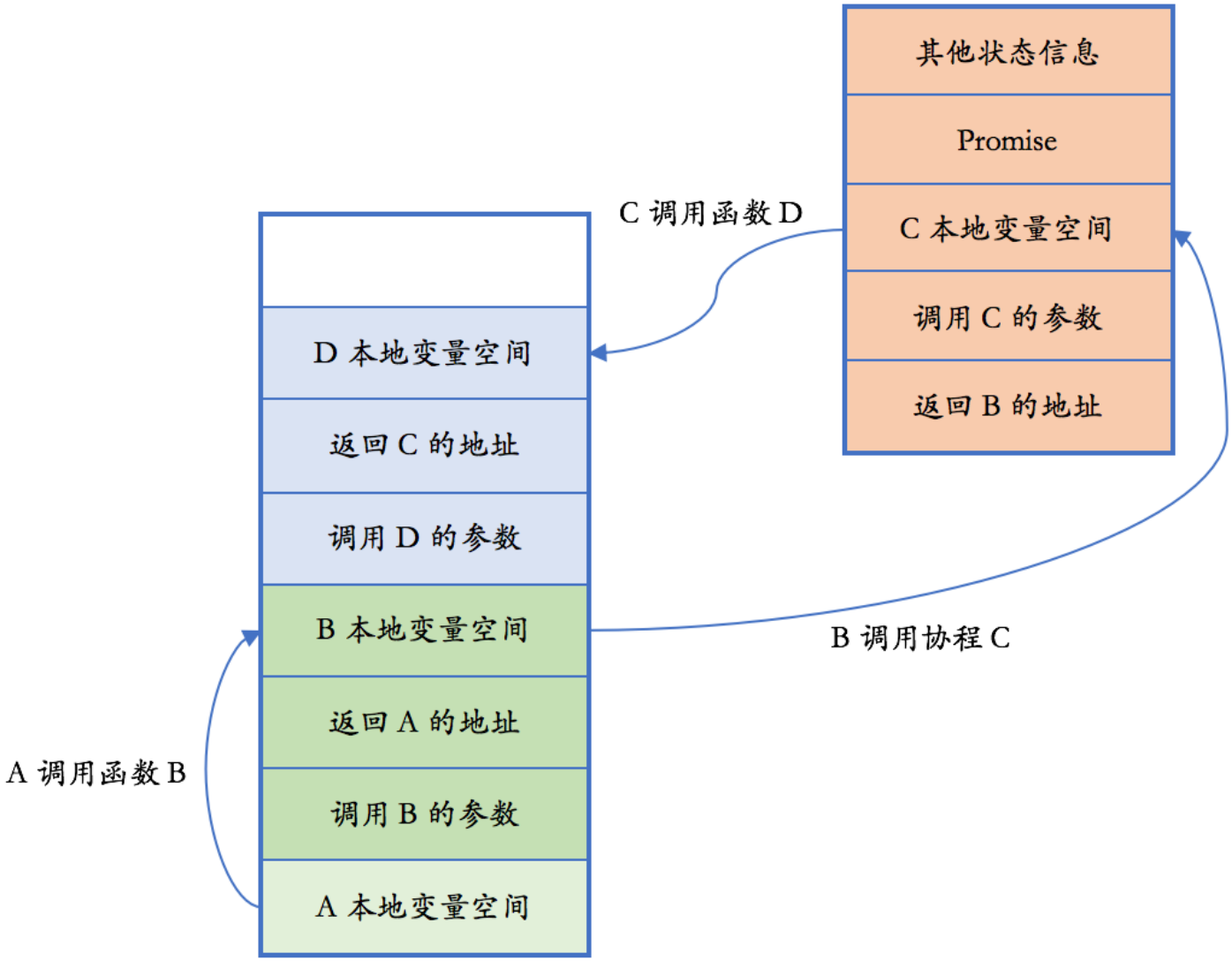
有栈协程和无栈协程

我们最后需要说一下有栈（stackful）协程和无栈（stackless）协程的区别。C++ 里很早就有了有栈的协程，概念上来讲，有栈的协程跟纤程、goroutines 基本是一个概念，都是由用户自行调度的、操作系统之外的运行单元。每个这样的运行单元都有自己独立的栈空间，缺点当然就是栈的空间占用和切换栈的开销了。而无栈的协程自己没有独立的栈空间，每个协程只需要一个很小的栈帧，空间占用小，也没有栈的切换开销。

C++20 的协程是无栈的。部分原因是有栈的协程可以使用纯库方式实现，而无栈的协程需要一点编译器魔法帮忙。毕竟，协

程里面的变量都是要放到堆上而不是栈上的。

一个简单的无栈协程调用的内存布局如下图所示：



可以看到，协程 C 本身的本地变量不占用栈，但当它调用其他函数时，它会使用线程原先的栈空间。在上面的函数 D 的执行过程中，协程是不可以挂起的——如果控制回到 B 继续，B 可能会使用目前已经被 D 使用的栈空间！

因此，无栈的协程牺牲了一定的灵活性，换来了空间的节省和性能。有栈的协程你可能起几千个就占用不少内存空间，而无栈的协程可以轻轻松松起到亿级——毕竟，维持基本状态的开销我实测下来只有一百字节左右。

反过来，如果无栈的协程不满足需要——比如，你的协程里需要有递归调用，并在深层挂起——你就不得不寻找一个有栈的协程的解决方案。目前已经有一些成熟的方案，比如 Boost.Coroutine2 [4]。下面的代码展示如何在 Boost.Coroutine2 里实现 `fibonacci`，让你感受一点点小区别：

```
#include <iostream>
#include <stdint.h>
#include <boost/coroutine2/all.hpp>

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);
    auto tmp = a;
    a = b;
    b += tmp;
  }
}

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

编译器支持

前面提到了，MSVC 和 Clang 目前支持协程。不过，它们都需要特殊的命令行选项来开启协程支持：

- MSVC 需要 `/await` 命令行选项
- Clang 需要 `-fcoroutines-ts` 命令行选项

为了满足使用 CMake 的同学的要求，也为了方便大家编译，我把示例代码放到了 GitHub 上：https://github.com/adah1972/geek_time_cpp

内容小结

本讲讨论了 C++20 里的第三个重要特性：协程。协程仍然很新，但它的重要性是毋庸置疑的——尤其在生成器和异步 I/O 上。

课后思考

请仔细比较第一个 `fibonacci` 的 C++ 实现和最后使用 `generator` 的 `fibonacci` 的实现，体会协程代码如果自行用状态机的方式来实现，是一件多麻烦的事情。

如果你对协程有兴趣，可以查看参考资料 [5]，里面提供了一些较为深入的原理介绍。

参考资料

[1] 维基百科, “协程”. https://zh.wikipedia.org/zh-cn/协程

[2] Gor Nishanov, “Working draft, C++ extensions for coroutines”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf

[3] Lewis Baker, CppCoro. https://github.com/lewissbaker/cppcoro

[4] Oliver Kowalke, Boost.Coroutine2. https://www.boost.org/doc/libs/release/libs/coroutine2/doc/html/index.html

[5] Dawid Pilarski, “Coroutines introduction”. https://blog.panicsoftware.com/coroutines-introduction/

精选留言

谦谦君子

老师， 图左边“调用X的参数”在“返回Y的地址”下面， 而右边“调用X的参数”在“返回Y的地址”上面， 是画错了么， 还是协成里面就是跟栈上函数调用是反的呢?

2020-02-12 23:11

作者回复

在栈里是一个明确的压栈顺序的（x86或类似平台上不管什么编译器实现都差不多）。而协程的数据放在堆里，并没有类似的顺序惯例，实际实现的顺序可能完全不一样。

2020-02-13 23:50

晚风·和煦

老师，map<int, int>().swap(map1);
这个语句为什么不能达到真正释放map1内存的效果呢？必须得用malloc_trim

2020-02-11 02:20

作者回复

先说是不是，然后才谈得上为什么。

在大部分情况下都没有必要那么做。我从来没在代码里写过 malloc_trim。

在有虚拟内存的世界里，我看不出调用 malloc_trim 的必要性。操作系统自己能管好。

如果嵌入式开发，没有虚拟内存，你用 malloc_trim 也不见得有用。因为一旦堆的尾部有分配，你并不能释放内存回操作系统。

而且，你为什么要还内存给操作系统？以后你不用了吗？是程序要退出了吗？如果程序要退出，本来占用的资源就会被释放掉。如果程序不退出，进程管好自己的事，下次分配能不麻烦操作系统就不麻烦操作系统，应该反而更好。

我的个人见解，谁告诉你这句话的，基本上并不真懂应用开发，只是学了点 Linux 的知识，在瞎卖弄而已。

2020-02-11 13:38

gallencalade

您好，吴老师，之前在2018年cpp开发者大会上听过您讲string view和range，还巧妙的使用|管道符进行函数间对象传递，能否有幸添加一下您的微信？谢谢

2020-02-20 21:35

作者回复

有事情先发我邮件好了。应该不会找不到吧。

2020-02-21 08:35

Vackine

感觉跟python里面的async的新的标准库好像，是真的有性能上的提升么，还是只是编程魔法？

2020-02-10 09:41

作者回复

编程魔法和性能提升有矛盾么？

严肃点，协程不是为了提高代码性能，而是为了提高程序员的生产率。从这点上来说，协程仍然是一种编译器的黑魔法。

跟手写比起来，没有性能的提升。就如同除了极少数的情况（比如泛型允许内联导致C++的排序比C的qsort快），C++代码不会比C代码性能更高。

2020-02-10 19:58