协程

维基百科，自由的百科全书

协程（英語：coroutine）是计算机程序的一类组件，推广了协作式多任务的子例程，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，但在实践中使用没有子例程那样广泛。协程更适合于用来实现彼此熟悉的程序组件，如协作式多任务、异常处理、事件循环、迭代器、无限列表和管道。

根据高德纳的说法，马尔文·康威于1958年发明了术语“coroutine”并用于构建汇编程序[1]，关于协程的最初解说在1963年发表[2]。

目录

同子例程的比较

协程可以通过yield（取其“让步”之义而非“出产”）来调用其它协程，接下来的每次协程被调用时，从协程上次yield返回的位置接着执行，通过yield方式转移执行权的协程之间不是调用者与被调用者的关系，而是彼此对称、平等的。由于协程不如子例程那样被普遍所知，下面对它们作简要比较：

- 子例程可以调用其他子例程，调用者等待被调用者结束后继续执行，故而子例程的生命期遵循后进先出，即最后一个被调用的子例程最先结束返回。协程的生命期完全由对它们的使用需要来决定。
- 子例程的起始处是惟一的入口点，每当子例程被调用时，执行都从被调用子例程的起始处开始。协程可以有多个入口点，协程的起始处是第一个入口点，每个yield返回出口点都是再次被调用执行时的入口点。
- 子例程只在结束时一次性的返回全部结果值。协程可以在yield时不调用其他协程，而是每次返回一部份的结果值，这种协程常称为生成器或迭代器。
- 现代的指令集架构通常提供对调用栈的指令支持，便于实现可递归调用的子例程。在以Scheme为代表的提供续体的语言环境下[3]，恰好可用此控制状态抽象表示来实现协程。

子例程可以看作是特定状况的协程[4]，任何子例程都可转写为不调用yield的协程[5]。

示例

这里是一个简单的例子证明协程的实用性。假设这样一种生产者－消费者的关系，一个协程生产产品并将它们加入队列，另一个协程从队列中取出产品并消费它们。伪码表示如下：

```
var q := 新建队列

coroutine 生产者
    loop
        while q 不满载
            建立某些新产品
            向 q 增加这些产品
        yield 消费者

coroutine 消费者
    loop
        while q 不空载
            从 q 移除某些产品
            使用这些产品
        yield 生产者
```

队列用来存放产品的空间有限，同时制约生产者和消费者：为了提高效率，生产者协程要在一次执行中尽量向队列多增加产品，然后再放弃控制使得消费者协程开始运行；同样消费者协程也要在一次执行中尽量从队列多取出产品，从而倒出更多的存放产品空间，然后再放弃控制使得生产者协程开始运行。尽管这个例子常用来介绍多线程，实际上简单明了的使用协程的yield即可实现这种协作关系。

同线程的比较

协程非常类似于线程。但是协程是协作式多任务的，而线程典型是抢占式多任务的。这意味着协程提供并发性而非并行性。协程超过线程的好处是它们可以用于硬性实时的语境（在协程之间的切换不需要涉及任何系统调用或任何阻塞调用），这里不需要用来守卫关键区段的同步性原语

（primitive）比如互斥锁、信号量等，并且不需要来自操作系统的支持。有可能以一种对调用代码透明的方式，使用抢占式调度的线程实现协程，但是会失去某些利益（特别是对硬性实时操作的适合性和相对廉价的相互之间切换）。

线程是协作式多任务的轻量级线程，本质上描述了同协程一样的概念。其区别，如果一定要说有的话，是协程是语言层级的构造，可看作一种形式的控制流，而线程是系统层级的构造，可看作恰巧没有并行运行的线程。这两个概念谁有优先权是争议性的：线程可看作为协程的一种实现[6]，也可看作实现协程的基底[7]。

生成器

生成器，也叫作“半协程”[8]，是协程的子集。尽管二者都可以`yield`多次，挂起（`suspend`）自身的执行，并允许在多个入口点重新进入，但它们特别差异在于，协程有能力控制在它让位之后哪个协程立即接续它来执行，而生成器不能，它只能把控制权转交给调用生成器的调用者[9]。在生成器中的`yield`语句不指定要跳转到的协程，而是向父例程传递返回值。

尽管如此，仍可以在生成器设施之上实现协程，这需要通过顶层的分派器（dispatcher）例程（实质上是trampoline）的援助，它显式的把控制权传递给由生成器传回的令牌（token）所标识出的子生成器：

```
var q := 建立新队列

generator 生产者
    loop
        while q 不满载
            建立某些新产品
            向 q 增加这些产品
        yield 消费者

generator 消费者
    loop
        while q 不空载
            从 q 移除某些产品
            使用这些产品
        yield 生产者

subroutine 分派器
    var d := 创建新字典(generator → iterator)
    d[生产者] := start 生产者
    d[消费者] := start 消费者
    var 当前 := 生产者
    loop
        call 当前
        当前 := next d[当前]

call 分派器
```

在不同作者和语言之间，术语“生成器”和“迭代器”的用法有着微妙的差异[10]。有人说所有生成器都是迭代器[11]，生成器看起来像函数而表现得像迭代器。在Python中，生成器是迭代器构造子：它是返回迭代器的函数。

同尾调用互递归的比较

使用协程用于状态机或并发运行类似于使用经由尾调用的互递归，在二者情况下控制权都变更给一组例程中的另一个不同例程。但是，协程更灵活并且一般而言更有效率。因为协程是`yield`而非`return`返回，接着恢复执行而非在起点重新开始，它们有能力保持状态，包括变量（同于闭

包）和执行点二者，并且yield不限于位于尾部位置；互递归子例程必须要么使用共享变量，要么把状态作为参数传递。进一步的说，每一次子例程的互递归调用都需要一个新的栈帧（除非实现了尾调用消去），而在协程之间传递控制权使用现存上下文并可简单地通过跳转来实现。

协程之常见用例

协程有助于实现：

- 状态机：在一个子例程里实现状态机，这里状态由该过程当前的出口/入口点确定；这可以产生可读性更高的代码。
- 演员模型：并发的演员模型，例如计算机游戏。每个演员有自己的过程（这又在逻辑上分离了代码），但他们自愿地向顺序执行各演员过程的中央调度器交出控制（这是合作式多任务的一种形式）。
- 生成器：可用于串流，特别是输入/输出流，和对数据结构的通用遍历。
- 通信顺序进程：这里每个子进程都是协程。通道输入/输出和阻塞操作会yield协程，并由调度器在有完成事件时对其解除阻塞（unblock）。可作为替代的方式是，每个子进程可以是在数据管道中位于其后的子进程的父进程（或是位于其前者之父，这种情况下此模式可以表达为嵌套的生成器）。

支持协程的编程语言

协程起源于一种汇编语言方法，但有一些高级编程语言支持它。早期的例子包括Simula[12]、Smalltalk和Modula-2。更新近的例子是Ruby、Lua、Julia和Go。

- Aikido
- AngelScript
- Ballerina
- BCPL
- Pascal（Borland Turbo Pascal 7.0带有uThreads模块）
- BETA
- BLISS
- C++（自从C++20）
- C#（自从2.0）
- ChucK
- CLU
- D
- Dynamic C
- Erlang
- F#
- Factor
- GameMonkey Script
- GDScript（Godot的脚本语言）
- Go
- Haskell[13][14]
- 高级汇编语言[15]
- Icon
- Io
- JavaScript（自从1.7，标准化于ECMAScript 6[16]，ECMAScript 2017还包括async/await支持）
- Julia[17]
- Kotlin（自从1.1）[18]
- Limbo
- Lua[19]
- Lucid
- µC++
- Modula-2
- Nemerle
- Perl 5（使用Coro模块[20]）

- PHP（带有hiphop-php[21]，原生支持自从PHP 5.5）
- Picolisp
- Prolog
- Python（自从2.5[22]，带有改进支持自从3.3，带有显式语法自从3.5[23]）
- Raku[24]
- Ruby
- Sather
- Scheme
- Self
- Simula 67
- Smalltalk
- Squirrel
- Stackless Python
- SuperCollider[25]
- Tcl（自从8.6）
- urbiscript

由于续体可被用来实现协程，支持续体的编程语言也非常容易就支持协程。

实现

到2003年，很多最流行的编程语言，包括C语言和它的后继者，都未在语言内或其标准库中直接支持协程。（这在很大程度上是受基于堆栈的子例程实现的限制）。C++的Boost.Context[26]库是个例外，它是Boost C++ Libraries的一部分，它在POSIX、Mac OS X和Windows上支持ARM、MIPS、PowerPC、SPAR和x86的上下文切换。可以在Boost.Context之上建造协程。

在协程是某种机制的最自然的实现方式，却不能获得可用协程的情况下，典型的解决方法是使用闭包，它是用状态变量（静态变量常为布尔标志值）来在调用之间维持内部状态，并转移控制权至正确地点的子例程。基于这些状态变量的值，在代码中的条件语句导致在后续调用时有着不同代码路径的执行。另一种典型的解决方法实现一个显式状态机，采用某种形式的大量而复杂的switch语句或goto语句特别是“计算goto”。这种实现被认为难于理解和维护，更是想要有协程支持的动机。

在当今的主流编程环境里，协程的合适的替代者是线程和适用范围较小的线程。线程提供了用来管理“同时”执行的代码段实时协作交互的功能，在支持C语言的环境中，线程是广泛有效的，POSIX.1c（IEEE Std 1003.1c-1995）规定了被称为pthreads的一个标准线程API，它在类Unix系统中被普遍实现。线程被很好地实现、文档化和支持，很多程序员对其也比较熟悉。但是，线程包括了许多强大和复杂的功能用以解决大量困难的问题，这导致了困难的学习曲线，当任务仅需要协程就可完成时，使用线程似乎就是用力过猛了。GNU Pth可被视为类Unix系统上线程的代表，有人尝试过用Windows的线程机制实现协程[7]。

C语言实现

C标准库里有“非局部跳转”函数setjmp和longjmp，它们分别保存和恢复：栈指针、程序计数器、被调用者保存的寄存器和ABI要求的任何其他内部状态。在C99标准中，跳转到已经用`return`或`longjmp`终止的函数是未定义的[27]，但是大多数`longjmp`实现在跳转时不专门销毁调用栈中的局部变量，在被后续的函数调用等覆写之前跳转回来恢复时仍是原样，这允许在实现协程时谨慎的用到它们。

POSIX.1-2001/SUSv3和此前的SUSv2进一步提供了操纵上下文的强力设施：setcontext、getcontext、makecontext和swapcontext，可方便地用来实现协程，但是由于`makecontext`的参数定义不符合C99标准要求，这些函数在POSIX.1-2004中被废弃，并在POSIX.1-2008中被删除[28]。POSIX.1-2001/SUSv3和此前的SUSv2定义了`sigaltstack`，可用来在不能获得`makecontext`的情况下稍微迂回的实现协程[29]。极简实现不采用有关的标准API函数进行上下文交换，而是写一小块内联汇编只对换栈指针和程序计数器故而速度明显的要更快。

由于缺乏直接的语言支持，很多作者写了自己的含藏上述技术细节的协程库，以Russ Cox的libtask协程库为代表[30]，它自称能够“写事件驱动程序而没有麻烦的事件”，并可用在FreeBSD、Linux、Mac OS X和SunOS之上。知名的实现还有：libpcl[31]、lthread[32]、libCoroutine[33]、libconcurrency[34]、libcoro[35]、ribs2[36]、libdill[37]、libaco[38]、libco[39]等等。

此外人们做了用C语言的子例程和宏实现协程的大量尝试，并取得了不同程度的成功。Simon Tatham作出的贡献是这一方法的很好示例[40]，它受到了达夫设备利用swtich语句“掉落”特性的启发，并且是Protothreads和类似实现的基础[41]。这种方法的确可以提高代码段的可写性、可读性，但可维护性是存在争议的[42]。这种不为每个协程维护独立的栈帧的实现方式主要缺点是，局部变量在经过从函数`yield`之后是不保存的，控制权只能从顶层例程`yield`[43]。

Python实现

Python 2.5基于扩展的生成器实现对类似协程功能的更好支持[44]。Python 3.3通过支持委托给子生成器增进了这个能力[45]。Python 3.4介入了综合性的异步I/O框架标准化，在其中扩展了利用子生成器委托的协程[46]，这个扩展在Python 3.8中被弃用[47]。Python 3.5通过async/await语法介入了对协程的显式支持[48]。从Python 3.7开始async/await成为保留关键字[49]。例如：

```
import asyncio
import random

async def produce(queue, n):
    for item in range(n):
        # 生产一个项目，使用sleep模拟I/O操作
        print('producing item {} ->'.format(item))
        await asyncio.sleep(random.random())
        # 将项目放入队列
        await queue.put(item)
    # 指示生产完毕
    await queue.put(None)

async def consume(queue):
    while True:
        # 等待来自生产者的项目
        item = await queue.get()
        if item is None:
            break
        # 消费这个项目，使用sleep模拟I/O操作
        print('consuming item {} <-'.format(item))
        await asyncio.sleep(random.random())

async def main():
    queue = asyncio.Queue()
    task1 = asyncio.create_task(produce(queue, 10))
    task2 = asyncio.create_task(consume(queue))
    await task1
    await task2

asyncio.run(main())
```

实现协程的第三方库：

- Eventlet[50]
- Greenlet[51]
- gevent[52]
- Stackless Python[53]

Perl实现

- Coro[54]，它是Perl5中的一种协程实现，使用C作为底层，所以具有良好的执行性能，而且可以配合AnyEvent共同使用，极大的弥补了Perl在线程上劣势。

Scheme实现

Scheme提供了对续体的完全支持，实现协程是很轻易的，只需维护一个续体的队列。在续体条目中有使用续体将协程实现为独立线程的一个用例[55]。

Smalltalk实现

在大多数Smalltalk环境中，执行堆栈是头等公民，实现协程不需要额外的库或VM支持。

Tcl实现

从 Tcl 8.6 开始，Tcl 核心内置协程支持，成为了继事件循环、线程后的另一种内置的强大功能。

引用

1. Knuth, Donald Ervin. Fundamental Algorithms (PDF). The Art of Computer Programming **1** 3rd. Addison-Wesley. 1997. Section 1.4.5: History and Bibliography, pp. 229 [2019-11-23]. ISBN 978-0-201-89683-1. （原始内容存档 (PDF)于2019-10-21）.
2. Conway, Melvin E. Design of a Separable Transition-diagram Compiler (PDF). Communications of the ACM (ACM). July 1963, **6** (7): 396–408. ISSN 0001-0782. doi:10.1145/366663.366704 –通过ACM Digital Library.
3. call-with-current-continuation for C programmers. http://community.schemewiki.org/. [2019-11-27]. （原始内容存档于2008-12-16）. “If you're a C programmer then you've probably read the various introductions and tutorials on call-with-current-continuation (call/cc) and come out not much wiser about what it all really means. Here's the secret: it's setjmp/longjmp. But on no account say that to any Scheme programmers you know, it'll send them into paroxysms of rage as they tell you you don't know what you're talking about.”
4. Knuth, Donald Ervin. Fundamental Algorithms. The Art of Computer Programming **1** 3rd. Addison-Wesley. 1997. Section 1.4.2: Coroutines, pp. 193–200. ISBN 978-0-201-89683-1.
5. Perlis, Alan J. Epigrams on programming. ACM SIGPLAN Notices. September 1982, **17** (9): 7–13 [2019-11-23]. doi:10.1145/947955.1083808. （原始内容存档于1999-01-17）. “6. Symmetry is a complexity reducing concept (co-routines include sub-routines); seek it everywhere”
6. A Fiber Class. [2019-11-22]. （原始内容存档于2017-10-23）.
7. Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API. 2005-09-16 [2019-11-26]. （原始内容存档于2020-06-13）.Ajai Shankar, MSDN Magazine
8. Anthony Ralston. Encyclopedia of computer science. Nature Pub. Group. 2000 [11 May 2013]. ISBN 978-1-56159-248-7. （原始内容存档于2019-06-08）.

9. See for example *The Python Language Reference (https://docs.python.org/reference/index.html)（页面存档备份 (https://web.archive.org/web/20121024054933/https://docs.python.org/reference/index.html)，存于互联网档案馆）*
"https://docs.python.org/reference/expressions.html#yieldexpr（页面存档备份 (https://web.archive.org/web/20121026064102/https://docs.python.org/reference/expressions.html#yieldexpr)，存于互联网档案馆） 5.2.10. Yield expressions":
"All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where should the execution continue after it yields; the control is always transferred to the generator's caller."
10. Watt, Stephen M. A Technique for Generic Iteration and Its Optimization (PDF). The University of Western Ontario, Department of Computer Science. [2012-08-08].（原始内容存档于2012-08-06）. “Some authors use the term iterator, and others the term generator. Some make subtle distinctions between the two.”
11. What is the difference between an Iterator and a Generator?. [2019-11-29].（原始内容存档于2020-06-25）.
12. Dahl, O.-J. and Hoare, C.A.R. (ed). Hierarchical Program Structures. Structured Programming. London, UK: Academic Press. 1972: 175–220. ISBN 978-0122005503.
13. Coroutine: Type-safe coroutines using lightweight session types. [2019-11-24].（原始内容存档于2013-01-20）.
14. Co-routines in Haskell. [2019-11-24].（原始内容存档于2020-01-09）.
15. The Coroutines Module (coroutines.hhf). HLA Standard Library Manual. [2019-11-24].（原始内容存档于2019-04-27）.
16. New in JavaScript 1.7. [2019-11-24].（原始内容存档于2009-03-08）.
17. Julia Manual - Control Flow - Tasks (aka Coroutines). [2019-11-24].（原始内容存档于2020-06-13）.
18. What's New in Kotlin 1.1. [2019-11-24].（原始内容存档于2019-08-11）.
19. Lua 5.2 Reference Manual. www.lua.org. [2019-11-24].（原始内容存档于2018-01-13）.
20. Coro. [2019-11-24].（原始内容存档于2019-05-29）.
21. [1] (https://github.com/facebook/hiphop-php)]（页面存档备份 (https://web.archive.org/web/20121224062933/https://github.com/facebook/hiphop-php)，存于互联网档案馆）
22. Python async/await Tutorial. Stack Abuse. December 17, 2015 [2019-11-24].（原始内容存档于2019-11-29）.
23. 8. Compound statements — Python 3.8.0 documentation. docs.python.org. [2019-11-24].（原始内容存档于2019-11-27）.
24. Gather and/or Coroutines. 2012-12-19 [2019-11-24].（原始内容存档于2020-06-13）.
25. McCartney, J. "Rethinking the Computer Music Programming Language: SuperCollider" (http://portal.acm.org/citation.cfm?id=1245228). Computer Music Journal, 26(4):61-68. MIT Press, 2002.
26. Boost.Context (http://www.boost.org/doc/libs/1_55_0/libs/context/doc/html/index.html)（页面存档备份 (https://web.archive.org/web/20190831230026/http://www.boost.org/doc/libs/1_55_0/libs/context/doc/html/index.html)，存于互联网档案馆）
27. ISO/IEC 9899:1999 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf) Archived (https://www.webcitation.org/618HUIDTw?url=http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf) 2011-08-22 at WebCite, 2005, 7.13.2.1:2 and footnote 211
28. getcontext(3) - Linux manual page. man7.org. [2019-11-21].（原始内容存档于2019-11-27）.

29. GNU Pth - IMPLEMENTATION NOTES. [2019-11-27]. （原始内容存档于2019-12-19）. “Pth is very portable because it has only one part which perhaps has to be ported to new platforms (the machine context initialization). But it is written in a way which works on mostly all Unix platforms which support makecontext(2) or at least sigstack(2) or sigaltstack(2) [seepth_mctx.c for details].”
Portable multithreading: the signal stack trick for user-space thread creation. ATEC '00 Proceedings of the annual conference on USENIX Annual Technical Conference. 2000-06-18: 20–20 [2019-11-27]. （原始内容存档于2019-10-31） –通过ACM Digital Library.
30. libtask. [2019-11-21]. （原始内容存档于2019-11-15）.
31. Portable Coroutine Library (http://xmailserver.org/libpcl.html) （页面存档备份 (https://web.archive.org/web/20051214201518/http://xmailserver.org/libpcl.html)，存于互联网档案馆） - C library using POSIX/SUSv3 facilities
32. lthread: a multicore enabled coroutine library written in C (https://github.com/halayli/lthread) （页面存档备份 (https://web.archive.org/web/20200613175536/https://github.com/halayli/lthread)，存于互联网档案馆） - lthread is a multicore/multithread coroutine library written in C
33. libcoroutine: A portable coroutine implementation. [2019-11-21]. （原始内容存档于2019-11-12）. for FreeBSD, Linux, OS X PPC and x86, SunOS, Symbian and others
34. libconcurrency - A scalable concurrency library for C. a simple C library for portable stack-switching coroutines
35. libcoro: C-library that implements coroutines (cooperative multitasking) in a portable fashion. [2019-11-21]. （原始内容存档于2019-12-02）. used as the basis for the Coro perl module.
36. RIBS (Robust Infrastructure for Backend Systems) version 2. August 13, 2019 [2019-11-21]. （原始内容存档于2020-04-22） –通过GitHub.
37. libdill: Structured Concurrency for C. libdill.org. [2019-11-21]. （原始内容存档于2019-12-02）.
38. 极速的轻量级C异步协程库: hnes/libaco. October 21, 2019 [2018-10-16]. （原始内容存档于2018-11-29） –通过GitHub.
39. libco: cooperative multithreading library written in C89.. code.byuu.org.
40. Simon Tatham. Coroutines in C. 2000 [2019-11-22]. （原始内容存档于2019-11-09）.
41. Stackless coroutine implementation in C and C++: jsseldenthuis/coroutine. March 18, 2019 [2019-11-26]. （原始内容存档于2020-06-13） –通过GitHub.
42. Simon Tatham. Coroutines in C. 2000 [2019-11-22]. （原始内容存档于2019-11-09）. “这一技巧破坏了书本中的每一个编码标准……［但是］任何试图牺牲算法明晰来确保语法清晰的编码标准都应该被重写。如果你的老板因为你使用了这些技巧而解雇你，在保安把你从大楼里拖出来的同时不断地告诉他们我的这句话。”
43. Coroutines in C – brainwagon. [2019-11-22]. （原始内容存档于2019-07-23）.
44. PEP 342 -- Coroutines via Enhanced Generators. [2019-11-21]. （原始内容存档于2020-05-29）.
45. PEP 380 -- Syntax for Delegating to a Subgenerator. [2019-11-21]. （原始内容存档于2020-06-04）.
46. PEP 3156 -- Asynchronous IO Support Rebooted: the "asyncio" Module. [2019-11-21]. （原始内容存档于2019-11-14）.
47. Generator-based Coroutines. [2020-10-29]. （原始内容存档于2018-12-31）.

48. PEP 492 -- Coroutines with async and await syntax. [2019-11-21]. （原始内容存档于2019-01-05）.
49. What's New In Python 3.7. [2019-11-21]. （原始内容存档于2019-11-28）.
50. Eventlet. [2019-11-21]. （原始内容存档于2019-11-15）.
51. Greenlet. [2019-11-21]. （原始内容存档于2019-08-25）.
52. gevent. [2020-10-02]. （原始内容存档于2020-09-16）.
53. Stackless Python. [2019-11-21]. （原始内容存档于2015-12-08）.
54. Coro. [2013-06-01]. （原始内容存档于2013-06-01）.
55. Haynes, C. T., Friedman, D. P., and Wand, M. 1984. Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (Austin, Texas, United States, August 06–08, 1984). LFP '84. ACM, New York, NY, 293-298.

参见

- 生成器
- 管道 (软件)
- 通道 (编程)
- 绿色线程
- Future与promise

延伸阅读

- Ana Lucia de Moura; Roberto Ierusalimschy. Revisiting Coroutines (PDF). ACM Transactions on Programming Languages and Systems. 2004, **31** (2): 1–31 [2019-11-27]. doi:10.1145/1462166.1462167. （原始内容存档 (PDF)于2019-05-13）.

外部链接

- Softpanorama coroutine page (http://www.softpanorama.org/Lang/coroutines.shtml)（页面存档备份 (https://web.archive.org/web/20191202061946/http://www.softpanorama.org/Lang/coroutines.shtml)，存于互联网档案馆） – 包含很多汇编协程链接