第14章. 异步编程

本书截止上一章都只在讨论“同步编程（synchronous programming）”，然而绝大部分复杂程序是以异步方式编写的。因此本章是用Python编写复杂程序的基础，同时也是本书最长的一章。

14-1. 异步编程的哲学

（标准库：asyncio）

到目前为止，我们在编写Python脚本时都假设了这样一种情况：计算机系统的全部资源都被用于执行该Python脚本，使其按照预设好的步骤顺序执行。然而真实情况是，计算机系统中有成千上万的进程在同时运行，其中一部分构成了操作系统，其余属于应用程序，而执行我们编写的Python脚本的进程只是后者中之一。操作系统内核实现了对进程的动态调度，将硬件资源合理地分配给所有这些进程。由于进程调度中进程切换的速度是很快的（相对于人类的感觉），所以形成了Python脚本一直在被执行的假象。

操作系统实现进程调度的初衷是为了不浪费硬件资源。第7章已经提到了I/O操作的速度比CPU运算和内存访问的速度慢几个数量级。这意味着执行一个I/O操作所需的时间足够执行成百上千条机器指令。如果全部硬件资源都被用于执行一个程序，例如我们编写的Python脚本，那么CPU很可能在99%的时间内都被闲置。而如果有大量进程在同时运行，那么CPU可以在其中一个进程执行I/O操作期间转而执行属于其他进程的机器指令，这能显著缩短完成所有这些进程的总时间。

为了达到上述目的，操作系统需要在一个进程开始执行I/O操作时将其设置为阻塞状态，在I/O操作执行完成时将其设置为就绪状态。然而问题来了：I/O操作是通过系统调用实现的，开始I/O操作必然会调用某个系统调用，同时也就通知了操作系统；但I/O操作结束不会再次调用某个系统调用，那么此时又该如何通知操作系统呢？为解决这一问题，计算机系统的外围设备需要在I/O操作结束时通过硬件触发一个状态变化提示，而操作系统每执行完一条指令都会检查一次所有外围设备的状态变化，以了解是否有I/O操作完成。

上述机制成为了实现进程调度的基础。操作系统在每执行完一条指令之后都能对新发生的外围设备状态变化做出响应。由于这些状态变化的发生是不可预测的，所以我们用“异步（asynchronous）”这个术语来描述操作系统内核进程的运行方式：它并非严格按照预设好的步骤顺序执行代码，而是根据外围设备状态变化跳跃性地执行代码。可以说，编写操作系统是“异步编程（asynchronous programming）”最早的范例。

异步编程最初仅被用于提高CPU的使用率，但后来人们逐渐发掘了它的潜力。只要让计算机的时钟以一个固定的时间间隔（例如1ms）周期性地触发状态变化提示，那么即便没有任何进程执行I/O操作也能进行进程调度，使操作系统可以创造出所有进程都在同时运行的假象，即实现所谓的“伪并行（pseudo parallel）”。操作系统还可以基于伪并行达到随时响应用户的目的，即实现所谓的“交互式操作（interactive operations）”。而“图形用户界面（graphical user interface，GUI）”其实只是对交互式操作的进一步开发利用。

然而上述技术只是操作系统对异步编程的使用。开发应用程序时是否能使用异步编程呢？答案是取决于操作系统是否提供了访问外围设备状态变化提示的系统调用。在没有提供这类系统调用的操作系统上，应用程序是无法使用异步编程的，如果其进程是单线程的，则执行一个I/O操作必然会被阻塞。为了抢占更多的计算资源，这些应用程序可以使用多线程技术，使得即便它的一个线程因执行I/O操作被阻塞，它的其他线程依然可以运行。然而线程调度同样是由操作系统进行的（在支持线程的操作系统上进程调度以线程为单位），每个线程都需要参与线程调度的竞争。因此多线程只能提高竞争获胜的概率，并不能真正实现异步编程。

大部分现代操作系统提供了访问外围设备状态变化提示的系统调用，即所谓“异步I/O（asynchronous I/O）”，例如Unix中的kqueue，Linux中的epoll和Windows中的IOCP。当然这些系统调用并非直接将外围设备的状态变化提示暴露给应用程序，而是将其包装成所谓的“事件（events）”。这些事件会按照发生的顺序被插入一个队列，而应用程序则需要利用这些系统调用不断从该队列中按顺序取出事件并进行处理，这一机制被称为“事件循环（event loop）”。判断一个应用程序是否使用了异步编程，只需要观察它的代码中是否存在事件循环。

用编译语言编程时，编程者需要使用异步I/O系统调用自己实现事件循环。解释器通常会利用异步I/O系统调用构建更容易使用的异步编程接口，例如Java中的java.awt.event包，以及JavaScript中的Event对象和Promise对象。Python则通过标准库中的selectors模块封装了异步I/O系统调用，然后通过构建在其上的asyncio模块提供了异步编程框架。本章不讨论selectors模块，只讨论asyncio模块。

asyncio是个相当复杂的模块，这从Python官方手册中关于asyncio的部分就可以看出来。概括地说，asyncio模块提供了两层API。低层级API以loop对象代表事件循环，使我们可以通过调用loop对象的属性进行异步编程。在这个层级需要采用异步编程的经典范式：为事件注册回调函数。然而这种编程方法较难掌握，尤其是当处理一个事件的回调函数会导致产生其他事件时，代码将变得迂回曲折难以理解。

为了降低异步编程的难度，asyncio模块提供了高层级API。高层级API建立在低层级API之上，使我们不需要直接与事件和事件循环打交道，通过较容易理解的async/await语法就能进行异步编程。本章只讨论asyncio模块的高层级API，以及与它们配合使用的“协程（coroutines）”。如果你对asyncio模块的低层级API感兴趣，请自行查阅Python官方手册。

14-2. 协程

（语言参考手册：3.2、3.4.1、3.4.2、6.4、8.9.1）
（标准库：asyncio、sys）

如果说事件循环是异步程序的核心驱动器，那么协程就是异步程序的构建模块。异步程序也可以包含函数，但必然额外包含一些协程。协程不是函数，它们的运行机制接近于生成器。事实上，在Python 3.5之前没有专门实现协程的语法，Python程序员是通过生成器来手工实现协程的。

现在，我们应使用“协程函数（coroutine functions）”创建协程。当在函数定义语句的def关键字之前添加async关键字时，该语句就变成了一条协程函数定义语句：

```
async def funcname([parameter_list]):
    suite
```

调用协程函数与调用生成器函数类似，不会执行函数体，而会返回一个协程。

表14-1列出了协程的属性。此外，协程也通过特殊属性__name__、__qualname__和__doc__记录了创建它的协程函数的相关信息。对比表14-1和表12-1，你会发现协程和生成器非常相似。

表14-1. 协程的属性

属性	说明
`coroutine.__await__()`	返回一个特殊的迭代器，使协程可被视为可等待对象。
`coroutine.send()`	手工启动协程，并传入一个值。
`coroutine.throw()`	手工启动协程，并触发一个异常。
`coroutine.close()`	手工关闭协程。
`coroutine.cr_code`	引用创建该协程的协程函数的函数体对应的代码对象。
`coroutine.cr_frame`	引用执行协程函数的函数体的过程中创建的帧对象。
`coroutine.cr_await`	引用一个可等待对象，表明该协程正在等待该对象执行完成。
`coroutine.cr_running`	引用一个布尔值，表明协程是否在运行。
`coroutine.cr_suspended`	引用一个布尔值，表明协程是否被挂起。
`coroutine.cr_origin`	引用创建该协程的代码，以帮助调试。

协程与生成器最大的区别在于协程不具有魔术属性__iter__和__next__，因此不能被直接迭代。作为替代，协程具有__await__属性，调用该属性会返回一个能通过await表达式迭代的特殊迭代器。协程的执行是通过迭代该特殊迭代器实现的。

协程因为具有__await__而可被视为“可等待对象（awaitables）”。理论上，任何具有__await__并通过它返回特殊迭代器的对象都可被视为一个可等待对象。然而Python官方手册并没有提供关于__await__属性的详细说明，使我们无法在自定义类中实现该属性。除了协程，只有asyncio低层级API使用的Future对象和asyncio高层级API使用的Task对象属于可等待对象，而调用它们的__await__最终会转变为调用与它们关联的协程的__await__。

可等待对象总是通过await表达式来执行。await表达式的语法为：

```
await expr
```

其中expr的求值结果必须是一个可等待对象。await表达式的含义是调用对expr求值得到的可等待对象的__await__属性，然后迭代返回的特殊迭代器直到StopIteration异常被抛出，

最后将该异常的value属性引用的对象作为对该await表达式求值得到的对象。表2-2已经说明，await表达式的优先级相当高，甚至高于幂运算。

迭代__await__返回的特殊迭代器的过程等价于执行相应协程函数的函数体的过程，也就是执行相应协程的过程。当协程返回时，其返回值就将被StopIteration异常的value属性引用。await表达式会自动处理StopIteration异常，所以该异常不会被抛出。但如果在执行协程的过程中抛出了其他异常，那么该异常不会被await表达式自动处理，而会被视为await表达式抛出的异常。

需要强调，await表达式只能出现在协程函数的函数体中，因为它代表着一次异步操作，对其求值意味着一次协程切换。下面通过一个例子来说明：

```
#!/usr/bin/env python3

import asyncio
import sys


#该协程函数创建的协程代表秒表。    sec参数传入需计时的秒数。
async def stopwatch(sec):
    n = 0
    while n < sec:
        #输出当前剩余秒数并立即显示。
        sys.stdout.write(str(sec - n) + ' ')
        sys.stdout.flush()
        #等待一个耗时1秒的协程执行完成。
        n = await asyncio.sleep(1, n + 1)
    #输出剩余秒数0并立即显示。
    sys.stdout.write('0\n')
    sys.stdout.flush()


#主协程函数将第一个命令行参数解读为需计时的秒数。
async def main():
    if len(sys.argv) < 2:
        sys.exit(1)
    try:
        sec = int(sys.argv[1])
        #等待stopwatch()返回的协程执行完成。
        await stopwatch(sec)
        sys.exit(0)
    except Exception:
        sys.exit(1)


#执行主协程函数创建的主协程。
if __name__ == '__main__':
    asyncio.run(main())
```

请将上述代码保存为coroutine1.py，然后通过如下命令行验证（在Unix和类Unix操作系统上还可以通过修改其文件模式将其当成shell脚本来使用）：

```
$ python3 coroutine1.py 10
10 9 8 7 6 5 4 3 2 1 0
$ python3 coroutine1.py 5
5 4 3 2 1 0
$
```

执行coroutine1.py并给出了命令行参数n后，将以1秒为间隔从n倒数到0，正如一个秒表所表现的那样。

coroutine1.py虽然简单，但完整体现了异步编程的逻辑，可作为异步Python脚本的规范格式。与第7章给出的同步Python脚本规范格式相比，异步Python脚本具有如下不同点：

- 除了sys模块之外，asyncio模块也必须被导入。
- main()不再是一个函数，而是一个协程函数。sys.exit()应在main()的函数体中被调用。
- 当该脚本被创建为主模块时，通过asyncio.run()来执行调用main()创建的协程。

需要详细说明的是asyncio.run()函数，其语法为：

```
asyncio.run(coro, *, debug=False)
```

其coro参数需被传入一个协程，而debug参数则用于控制是否启用调试模式。该函数的作用是创建一个事件循环并自动对其进行管理，通过该事件循环来执行coro参数传入的协程，当该协程执行完毕时该事件循环也会自动关闭。asyncio.run()总是被作为异步Python脚本的入口点来使用，在整个脚本运行过程中只应被调用一次，而通过它执行的协程就是所谓的“主协程（main coroutine）”。

主协程会通过await表达式执行其他协程，在上面的例子中是调用stopwatch()创建的协程；而这些被主协程执行的协程也会通过await表达式执行更多协程，在上面的例子中是调用asyncio.sleep()创建的协程。所有这些协程其实都是由调用asyncio.run()创建的事件循环执行的。

下面让我们跟踪一下coroutine1.py的执行流程。在执行到调用asyncio.run()之前，协程函数stopwatch()和main()就已经存在于内存中了，但此时还没有创建任何协程。然后，调用main()创建的协程被作为参数传递给asyncio.run()，该协程会立刻被执行，导致main()的函数体被执行。而当执行到“await stopwatch(sec)”时，会调用stopwatch()创建一个协程，然后main()创建的协程被挂起，stopwatch()创建的协程开始运行。这一操作是通过对await表达式求值进行的，导致一次协程切换，使主线程的指令流从主协程转移到了stopwatch()创建的协程；而当stopwatch()创建的协程执行完后，会再次进行协程切换，其返回值会变成await表达式的求值结果（该结果不一定会被使用）。任何协程都只能被执行一次，当执行完成后它的执行者（通常是另一个协程，但也可能是asyncio.run()）将不再引用它，如果它没有被其他变量引用的话，就会被垃圾回收机制销毁。

接下来让我们深入stopwatch()的函数体。它通过while语句循环了sec次，每次都需要执行“await asyncio.sleep(1, n + 1)”。asyncio.sleep()是一个协程函数，其语法为：

```
coroutine asyncio.sleep(delay, result=None)
```

其创建的协程的行为相当简单：让自身被挂起delay秒，然后返回result。如果delay被传入0，则该协程的作用是触发一次协程切换。在stopwatch()中，每次循环都用await表达式的值更新了n，而await表达式的值是n + 1，这就保证了不会陷入死循环。

从上面的例子可以看出，调用协程的__await__返回的特殊可迭代对象其实是在以await表达式为标志划分协程函数的函数体，每次迭代都从当前位置执行到下一个await表达式，只有最后一次迭代会执行到函数体的末尾。在上面的例子中，主协程的__await__返回的特殊可迭代对象可以迭代2次；stopwatch()创建的协程的__await__返回的特殊可迭代对象可以迭代sec+1次。值得说明的是，协程函数的函数体并非必须包含await表达式，只要在def关键字前添加了async关键字就会被视为协程函数。然而不包含await表达式的协程函数创建的协程不会执行任何异步操作，也就失去了存在的意义。

在理解了协程的执行方式后，就不难理解其send()、throw()和close()属性的功能了。它们的语法与在生成器中相同，分别用于指定上一次await表达式的求值结果、注入一个指定的异常、以及抛出GeneratorExit异常。注意在协程上触发GeneratorExit异常同样会让协程完成对自身的清理，效果与协程函数的函数体执行完成相同，但不会抛出StopIteration异常。同样，协程具有__del__属性，在它们被销毁时会自动调用其close()属性。

通过await表达式划分协程函数的函数体，类似于通过yield表达式划分生成器函数的函数体。但这里要强调，协程函数的函数体中不能包含yield表达式，否则将被视为语法错误。由于用await表达式替换了yield表达式，所以协程没有cr_yieldfrom属性，但具有cr_await属性。本节例子中的所有协程的cr_await属性都引用None。下一节会说明cr_await属性的作用。

显然，协程切换的本质只是一个协程被挂起，另一个协程开始运行，两者的cr_frame属性的变化与生成器的gi_frame的变化相同，而cr_running和cr_suspended同样反映协程的状态变化。cr_code属性则永远引用相应协程函数的函数体对应的代码对象。这意味着与进程切换和线程切换相比，协程切换的代价小到几乎可以忽略不计。这就是在高并发环境中使用协程能显著提升性能的原因。

协程的cr_origin属性是用来支持调试的，将在第16章讨论。这里只需要知道默认情况下cr_origin会引用None。

至此我们已经明白了协程的工作原理。从另一个角度，通过await表达式等待一个协程相当于进行了一次函数调用，只不过受影响的只是两个协程引用的帧对象，函数栈并不受影响。既然如此，协程函数也可以实现递归，即在其函数体内通过await表达式等待调用自身创建的协程。请将coroutine1.py拷贝到coroutine2.py，然后将stopwatch()修改为：

```
#该协程函数使用了递归。
async def stopwatch(sec):
    await asyncio.sleep(1)
    if sec <= 0:
        sys.stdout.write('0\n')
        sys.stdout.flush()
        return
    else:
        sys.stdout.write(str(sec) + ' ')
        sys.stdout.flush()
        await(stopwatch(sec -1))
```

可以验证coroutine2.py的功能与coroutine1.py完全相同。但要注意，递归协程函数的代价比递归函数要大，因为每次调用需创建一个协程而不仅仅是一个帧对象。

最后需要说明，asyncio.run()函数限制Python异步程序只能有一个主协程。在大部分情况下，这是符合逻辑的。但有时候我们需要通过同一事件循环执行多个主协程，此时必须使用asyncio定义的Runner类，其实例化语法为：

```
class asyncio.Runner(*, debug=False, loop_factory=None)
```

其中debug参数同样用于控制是否启用调试模式；loop_factory则用于控制创建事件循环的方式，但通常都会使用默认值None，以使用默认的方式创建事件循环。注意Runner类是Python3.11引入的。

Runner对象本身并不对应任何主协程，而应被视为一个主协程的容器。为了简化对这些主协程的控制，Runner对象被实现为一个上下文管理器，其__enter__属性完成了一些初始化操作，而__exit__属性则包含了对异常的处理。Runner对象的使用方法是作为with语句的参数，在进入了相应上下文的情况下，通过表14-2列出的属性来控制主协程和事件循环。

表14-2. Runner的属性

属性	说明
get_loop()	取得该Runner对象关联的事件循环。
run()	创建一个主协程。
close()	关闭该Runner对象。

get_loop()是通向低层级API的接口，本书不详细讨论。

run()的语法为：

```
asyncio.Runner.run(coro, *, context=None)
```

它是asyncio.run()函数的替代，表示将通过coro参数传入的协程作为主协程执行，context参数则用于自定义协程的上下文，传入None则表示使用默认上下文。

下面的例子说明了如何用Runner对象代替asyncio.run()。请将coroutine1.py拷贝到coroutine3.py，然后将最后的一段代码修改为：

```
#使用Runner对象代替asyncio.run()函数。
if __name__ == '__main__':
    with asyncio.Runner() as runner:
        runner.run(main())
```

可以验证coroutine3.py的功能与coroutine1.py完全相同。但要注意，此时我们可以在with语句的代码块中添加额外的run()调用，即（假设main1()、main2()等都是已经定义好的协程函数）：

```
runner.run(main())
runner.run(main1())
runner.run(main2())
...
```

这会使得该脚本具有多个主协程，且它们都通过同一个事件循环执行。

必须要强调的是，Runner对象被创建后，并不会自动创建事件循环，而是要等到run()或get_loop()被第一次调用时才会创建事件循环。Runner对象的__exit__会自动调用close()属性，其语法为：

```
asyncio.Runner.close()
```

这会导致事件循环被关闭，并释放主协程的上下文。

14-3. 任务

（标准库：asyncio）

至此我们已经学会了创建和使用协程，并基于协程进行异步编程。然而如果我们只使用协程，则可以实现异步，却无法实现并行。

请重新思考coroutine1.py。该脚本创建的所有协程是按照如下顺序被执行的：main()创建的协程 → stopwatch()创建的协程 → asyncio.sleep()创建的第1个协程 → asyncio.sleep()创

建的第2个协程 → ... → asyncio.sleep()创建的第sec个协程 → stopwatch()创建的协程 → main()创建的协程。这个执行流程与传统的函数调用没有本质区别。我们完全可以基于time模块提供的sleep()函数以同步编程的方式达到同样的效果。请看下面的代码：

```
#!/usr/bin/env python3
#
#本脚本以同步编程的方式实现coroutine1.py的功能。

import time
import sys


#该函数代表秒表。    sec参数传入需计时的秒数。
def stopwatch(sec):
    n = 0
    while n < sec:
        sys.stdout.write(str(sec - n) + ' ')
        sys.stdout.flush()
        #强制等待1秒钟。
        time.sleep(1)
        n = n + 1
    sys.stdout.write('0\n')
    sys.stdout.flush()


#主函数将第一个命令行参数解读为需计时的秒数。
def main():
    if len(sys.argv) < 2:
        return 1
    try:
        sec = int(sys.argv[1])
        #stopwatch()是以同步方式运行的。
        stopwatch(sec)
        return 0
    except Exception:
        return 1


#调用主函数。
if __name__ == '__main__':
    sys.exit(main())
```

请将上述代码保存为coroutine4.py，然后验证它与coroutine1.py的功能相同。

asyncio.sleep()与time.sleep()的区别在于：asyncio.sleep()是以非阻塞的方式实现等待的，只会让其本身创建的协程被挂起指定的时间，执行它的线程可以转而执行其他协程；而time.sleep()是以阻塞的方式实现等待的，执行它的线程被强制阻塞指定的时间。然而对于coroutine1.py来说，asyncio.sleep()创建的协程被stopwatch()创建的协程等待，后者又被主协程等待。在asyncio.sleep()创建的协程被挂起期间，该程序的所有协程都无法被执行，因此执行这些协程的事件循环也只能选择交出CPU的使用权，让操作系统调度运行其他线程，这与该线程被阻塞没有区别。

要想让异步编程发挥作用，就必须并行执行多个协程，使得当其中某个协程因执行I/O操作或调用了asyncio.sleep()而被挂起时，线程可以执行其他协程，不需要被阻塞。为了达到这一目的，我们需要将协程转化为“任务（tasks）”。

显式将协程转化为任务的方法是调用asyncio.create_task()，其语法为：

```
asyncio.create_task(coro, *, name=None)
```

其中coro参数需被传入一个协程，而name参数被用来设置任务的名称。注意这不是一个协程函数，而是一个函数，其功能是将通过coro参数传入的协程包装为任务。该函数返回一个代表任务的Task对象。Task类是asyncio的高层级API的一部分，但我们不应手工实例化它，而应通过调用asyncio.create_task()或别的API来获得任务。

由于任务是对协程的包装，所以也是可等待对象，能被用于await表达式。但需要强调，执行协程和执行任务的底层机制是有很大区别的。为了便于说明，先给出一个例子：

```
#!/usr/bin/env python3

import asyncio
import time
import sys

#引用主协程的全局变量。
coro_main = None


#该协程函数创建的协程会显示自身的执行过程。    它会睡眠1秒，在睡眠前后分别输出提示信息。
#   它还会显示主协程的cr_await属性的当前值。
async def self_explain(tag):
    print(f"{tag}: sleep at {time.strftime('%X')}")
    await asyncio.sleep(1)
    print(f"{tag}: awake at {time.strftime('%X')}")
    print(f"    {tag}: " + str(coro_main.cr_await))


#主协程函数混合使用了协程和任务。
async def main():
    await self_explain("coro1")
    task1 = asyncio.create_task(self_explain("task1"))
    task2 = asyncio.create_task(self_explain("task2"))
    await task2
    await self_explain("coro2")
    task3 = asyncio.create_task(self_explain("task3"))
    await task1
    await self_explain("coro3")
    await task3
    sys.exit(0)


if __name__ == '__main__':
    coro_main = main()
    asyncio.run(coro_main)
```

请将上述代码保存为coroutine5.py，然后通过如下命令行验证：

```
$ python3 coroutine5.py
coro1: sleep at 14:53:35
coro1: awake at 14:53:36
    coro1: None
task1: sleep at 14:53:36
task2: sleep at 14:53:36
task1: awake at 14:53:37
    task1: <_asyncio.FutureIter object at 0x10ed71ba0>
task2: awake at 14:53:37
    task2: <_asyncio.FutureIter object at 0x10ed71ba0>
coro2: sleep at 14:53:37
coro2: awake at 14:53:38
    coro2: None
coro3: sleep at 14:53:38
task3: sleep at 14:53:38
coro3: awake at 14:53:39
    coro3: None
task3: awake at 14:53:39
    task3: <_asyncio.FutureIter object at 0x10ed71ba0>
```

如果你初次接触异步编程的话，那么这个例子的结果肯定会让你感到迷惑。要想解释这个结果，就必须深刻理解事件循环是如何工作的。

事实上，事件循环动态维护着一个任务队列。每次调用asyncio.create_task()，都会将新创建的任务注册到这个任务队列中。而每次对await表达式求值时，首先会检查它所指定的任务是否已经执行完成，如果不是则会按顺序执行完任务队列中的所有任务，并非仅指定的任务本身。这意味着当await表达式的操作数是一个任务时，该任务必须已经存在，因此调用asyncio.create_task()的语句必须在对await表达式求值的语句之前。

更准确地说，当一个任务被注册到任务队列时，实际执行的操作是这样的：先为该任务创建一个Future对象来代表该任务的执行结果（由于该结果在未来才能得到所以此类对象所属类得名“Future”），然后让任务队列引用这个Future对象。需要强调的是，任务引用了相应Future对象，而Future对象则弱引用（会在第15章讨论）了任务以避免形成“循环引用（cyclic reference）”，无法增加任务的引用数。所以在调用asyncio.create_task()时必须将其返回的任务赋值给一个变量，以避免该任务在执行完之前就被垃圾回收机制销毁。

当通过await表达式等待一个任务时，实际上等待的是相应的Future对象。调用Future对象的__await__返回的特殊可迭代对象代表的是被该任务包装的协程对应的协程函数的函数体，在其被迭代期间会被该协程的cr_await属性引用。如果该协程执行完成，则会以返回值作为Future对象的结果；如果该协程抛出了异常，则会以异常作为Future对象的结果。当Future对象有了结果时，该结果会被传递给await表达式，Future对象自身则从任务队列中被删除。作为asyncio低层级API的一部分，Future对象是被Python解释器自动维护的，不应被手动创建。

上述操作仅当以任务的形式使用协程时才会被执行。当await表达式的操作数是一个协程时，会自动为该协程创建一个临时任务，但并不会为该任务创建Future对象，也不会将该任务注册到任务队列。事件循环会立刻调用该临时任务的__await__，而其返回的特殊可迭代对象同样代表被包装协程对应的协程函数的函数体。由于不存在Future对象，所以在该协程被执行期间等待它的协程的cr_await属性依然引用None。当该协程返回或抛出异常后，包装

它的临时任务会被垃圾回收机制销毁，如果协程本身没有被其他变量引用也会被销毁。而当该协程被挂起后，事件循环会继续按顺序执行完任务队列中的所有任务。

下面让我们回到coroutine5.py对应的例子，并梳理它的执行流程：

➤ 步骤一：等待协程coro1，此时任务队列是空的，因此线程在输出“coro1: sleep at 14:53:35”之后会被阻塞1秒钟，然后输出“coro1: awake at 14:53:36”和“coro1: None”，完成coro1的执行。
➤ 步骤二：创建任务task1和task2，使这两个变量引用各自的Task对象，而相应的Future对象则被添加到任务队列中。
➤ 步骤三：等待任务task2，这会导致清空任务队列，即先执行task1，输出“task1: sleep at 14:53:36”，然后任务被挂起，而此时线程可以执行task2，输出“task2: sleep at 14:53:36”。这两个操作是连续执行的，故记录的时间是同一秒。线程被阻塞1秒钟后，task1先被执行，先后输出“task1: awake at 14:53:37”和“task1: <_asyncio.FutureIter object at 0x10ed71ba0>”，最后弹出任务队列；task2后被执行，先后输出“task2: awake at 14:53:37”和“task2: <_asyncio.FutureIter object at 0x10ed71ba0>”，最后也弹出任务队列，使任务队列变空。这四个操作也是连续执行的，故记录的时间也是同一秒。
➤ 步骤四：等待协程coro2，与步骤一完全相同。
➤ 步骤五：创建任务task3，使任务队列再次非空。
➤ 步骤六：等待任务task1，但由于task1引用的任务已经处于“已完成”状态，所以什么也不做。
➤ 步骤七：等待协程coro3，此时任务队列非空，因此线程在输出“coro3: sleep at 14:53:38”后不会被阻塞，而是继续执行task3，输出“task3: sleep at 14:53:38”。这两个操作同样是连续执行的，所以记录的时间是同一秒。然后线程被阻塞1秒钟，协程coro3先被执行，先后输出“coro3: awake at 14:53:39”和“coro3: None”；task3后被执行，先后输出“task3: awake at 14:53:39”和“task3: <_asyncio.FutureIter object at 0x10ed71ba0>”，最后弹出任务队列，使任务队列再次变空。这四个操作也是连续执行的，故记录的时间也是同一秒。
➤ 步骤八：等待任务task3，但由于task3引用的任务已经处于“已完成”状态，所以什么也不做。
➤ 步骤九：退出并返回0。

需要强调的是，虽然“await task1”和“await task3”这两条语句什么也没做，但依然是必要的，因为它们保证了相关任务被完成。你可以做一个实验，删除“await task3”后，将只输出“task3: sleep at 14:53:38”，而不会输出后两条提示信息。在异步编程时必须坚持这样一个原则：只要创建了一个可等待对象（包括协程和任务），就必须通过某个await表达式来等待它。

至此相信你已经弄明白了任务的内在逻辑。实际上线程在执行任务时依然是串行的，但由于一个任务包装的协程被挂起后，线程可以执行其他的任务，所以这实现了“伪并行”。除了能以更复杂的方式被调度外，任务还为我们操控它自身提供了接口，即表14-3列出的属性。

表14-3. 任务的属性

属性	说明
`cancel()`	取消任务。
`cancelled()`	判断任务是否被取消。
`done()`	判断任务是否已完成。
`result()`	获得任务的结果。
`exception()`	获得任务的异常。
`add_done_callback()`	添加一个任务完成时被执行的回调函数。
`remove_done_callback()`	删除一个任务完成时被执行的回调函数。
`get_coro()`	获得任务包装的协程。
`get_stack()`	获得任务包装的协程的栈框架或回溯框架。
`print_stack()`	显示任务包装的协程的栈框架或回溯框架的相关信息。
`get_name()`	获得任务的名字。
`set_name()`	设置任务的名字。

这些属性中，add_done_callback()和remove_done_callback()属于低层级API，本书不讨论；get_coro()、get_stack()和print_stack()主要用于调试，将在第16章讨论。值得一提的是，这些属性在底层都是通过调用相应Future对象的同名属性实现其功能的。

在讨论表14-3中的属性之前，我们还需要知道一个任务的状态有3种：

- “未完成”：表明该任务尚未被执行，或正在执行中。
- “已完成”：表明该任务的执行过程已经停止，或者给出了返回值，或者抛出了某个以Exception类为基类的异常。
- “已取消”：表明该任务的执行过程已经停止，并抛出了asyncio.CancelledError异常。

从上面的讨论可知，由asyncio模块定义的asyncio.CancelledError异常是较特殊的。该异常是BaseException的直接子类，而非Exception的直接子类。asyncio.CancelledError异常总是通过调用任务的cancel()属性被抛出，其语法为：

```
Task.cancel(msg=None)
```

该函数的效果是这样的：该任务会立即抛出asyncio.CancelledError异常，并以msg参数传入的对象作为该异常的第一个参数。在此期间任务包装协程所对应协程函数的函数体可以捕获并处理该异常，这样该异常就不会被传递出该任务；否则要等到事件循环在处理下一个事件时，该异常才会被传递出该任务。

cancelled()属性返回一个布尔值，其语法为：

```
Task.cancelled()
```

当其返回True时表明该任务的状态是“已取消”，返回False时表明该任务的状态是“已完成”或“未完成”。

done()属性同样返回一个布尔值，其语法为：

```
Task.done()
```

当其返回True时表明该任务的状态是“已完成”或“已取消”，返回False时表明该任务的状态是“未完成”。

表14-4显示了如何根据cancelled()和done()的返回值组合准确判断一个任务的状态。

表14-4. 判断任务状态

cancelled()的返回值	done()的返回值	
	True	False
True	“已取消”	不可能
False	“已完成”	“未完成”

result()属性的功能是查看任务的结果，其语法为：

```
Task.result()
```

该函数根据任务的状态得到不同的结果：
“已完成”：返回被包装协程的返回值，或者重新抛出被包装协程抛出的异常。
“已取消”：重新抛出asyncio.CancelledError异常。
“未完成”：抛出asyncio.InvalidStateError异常。

exception()属性则专门用于取得任务抛出的异常，其语法为：

```
Task.exception()
```

该函数同样根据任务的状态得到不同的结果：

“已完成”：如果被包装协程给出了返回值，则返回None；否则重新抛出被包装协程抛出的异常。

“已取消”：重新抛出asyncio.CancelledError异常。

“未完成”：抛出asyncio.InvalidStateError异常。

上述属性是使用任务时较常用到的。下面通过几个例子来说明。第一个例子说明了如何通过cancel()属性取消一个正在执行的任务：

```
#!/usr/bin/env python3

import asyncio
import time
import sys


#该任务将通过msg传入的消息在延迟sec秒后写入标准输出。
async def delayed_sender(sec, msg, tag):
    try:
        await asyncio.sleep(sec)
        print(f"{tag}: {msg} at {time.strftime('%X')}")
    except asyncio.CancelledError as e:
        print(f"{tag}: Sending message is cancelled at
{time.strftime('%X')}.")
        print(e.args)
        #raise


async def main():
    task1 = asyncio.create_task(delayed_sender(1, "Message1", "sender1"))
    task2 = asyncio.create_task(delayed_sender(2, "Message2", "sender2"))
    task3 = asyncio.create_task(delayed_sender(3, "Message3", "sender3"))
    #开始并行执行任务队列中的三个任务。
    await asyncio.sleep(0.5)
    #取消task2。
    task2.cancel("Cancelling task2 ...")
    await task1
    await task2
    await task3
    sys.exit(0)


if __name__ == '__main__':
    asyncio.run(main())
```

请将上述代码保存为coroutine6.py，然后通过如下命令行验证：

```
$ python3 coroutine6.py
sender2: Sending message is cancelled at 21:57:45.
('Cancelling task2 ...',)
sender1: Message1 at 21:57:45
sender3: Message3 at 21:57:47
```

梳理一下这个程序的运行流程：在main()中的“await asyncio.sleep(0.5)”被执行后其创建的协程立刻被挂起。任务队列中的三个任务因此启动，但也都立刻被挂起，使得事件循环选择执行主协程。因此“task2.cancel("Cancelling task2 ...")”被执行，使task2被取消，这导致task2立刻抛出asyncio.CancelledError异常。然而由于delayed_sender()在函数体中通过try语句自行截获并处理了asyncio.CancelledError异常，所以该异常并不会被传递到task2之外，task2最后的状态是“已完成”，调用其result()属性的结果是None。因此这不会影响到task1和task3的运行。

然而只要稍微改动一下delayed_sender()，去掉“#raise”注释中的“#”，就会得到不一样的运行结果：

```
$ python3 coroutine6.py
sender2: Sending message is cancelled at 22:09:47.
('Cancelling task2 ...',)
sender1: Message1 at 22:09:48
sender3: Sending message is cancelled at 22:09:48.
()
... much information about handling asyncio.CancelledError ...
```

注意task1的执行没受影响，但task3却在执行过程中被取消。这是因为task2在自行截获并处理了asyncio.CancelledError异常后，选择将其再次抛出。而前面已经提到，该异常会在事件循环处理下一次事件时被抛出，而这会导致task1执行完成（该事件就是task1等待的asyncio.sleep()创建的协程返回）。然而asyncio.CancelledError异常被抛出后进入异常处理默认流程，导致程序终止，所有任务都会被销毁。这会导致task3自动被取消，但因取消它而抛出的asyncio.CancelledError异常会被自动处理，不会传递到task3之外。

第二个例子说明如何通过cancelled()、done()、result()和exception()查看一个任务的相关信息。该例子让用户输入一个正整数，然后按照推导公式a(n)=int(a(n-1)/7) - 1生成后续的数列项。而该生成过程会持续到下面三种情况之一：

- 下一项不再是正整数。数列被完整生成。任务返回下一项，状态为“已完成”。
- 下一项是19的倍数。数列被截断。任务抛出RuntimeError异常，状态为“已完成”。
- 下一项是11的倍数。数列被截断。任务被取消，状态为“已取消”。

请将下列代码保存为coroutine7.py：

```
#!/usr/bin/env python3

import asyncio
import sys


#该任务基于正整数n生成一个数列并写入标准输出。
async def seq_gen(n):
    try:
```

```
            #终止条件是n小于等于0。
            while n > 0:
                sys.stdout.write(str(n) + " ")
                sys.stdout.flush()
                #如果n是19的倍数，则抛出RuntimeError异常。
                if n % 19 == 0:
                    raise RuntimeError()
                #如果n是11的倍数，则取消该任务。
                if n % 11 == 0:
                    #调用自身的cancel()。
                    asyncio.current_task().cancel()
                #数列的推导公式是a(n) = int(a(n-1)/7) - 1。
                n = n // 7 - 1
                await asyncio.sleep(1)
        finally:
            print("\n")
        return n


    #该函数通过访问任务的属性获得任务相关信息。
    def show_task_result(task):
        #确定任务的当前状态。
        print("task.cancelled() ==", task.cancelled())
        print("task.done() ==", task.done())
        try:
            #取得任务的结果，注意这可能导致抛出异常。
            print("task.result() ==", task.result())
            print("task.exception() ==", task.exception())
        except BaseException as e:
            print("task.result() ==", repr(e))
            print("task.exception() ==", repr(e))


    #主协程函数将第一个命令行参数视为数列的起始项。
    async def main():
        if len(sys.argv) < 2:
            sys.exit(1)
        try:
            n = int(sys.argv[1])
            task = asyncio.create_task(seq_gen(n))
            await task
        except (asyncio.CancelledError, RuntimeError):
            #数列未完全生成。
            print("The sequence is truncated!")
            show_task_result(task)
        except Exception:
            sys.exit(1)
        else:
            #数列完全生成。
            print("The sequence is fully generated.")
            show_task_result(task)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

下面的命令行验证了任务给出返回值的情况：

```
$ python3 coroutine7.py 79
79 10
```

```
The sequence is fully generated.
task.cancelled() == False
task.done() == True
task.result() == 0
task.exception() == None
```

下面的命令行验证了任务抛出RuntimeError异常的情况：

```
$ python3 coroutine7.py 17697
17697 2527

The sequence is truncated!
task.cancelled() == False
task.done() == True
task.result() == RuntimeError()
task.exception() == RuntimeError()
```

下面的命令行验证了任务被取消的情况：

```
$ python3 coroutine7.py 79873
79873 11409 1628

The sequence is truncated!
task.cancelled() == True
task.done() == True
task.result() == CancelledError()
task.exception() == CancelledError()
```

上面的例子中还用到了一个技巧，即通过调用asyncio.current_task()使一个任务可以访问到自身，进而调用自身的cancel()取消自己。显然这一技巧也可被任务用于调用自身的其他属性。asyncio.current_task()的语法为：

```
asyncio.current_task(loop=None)
```

其中loop参数用于指定事件循环，但对于高层级API来说总是会让其使用默认实参值None。如果指定的事件循环不存在，则asyncio.current_task()会返回None。

接下来让我们思考这样一个问题：如果通过调用cancel()取消一个任务，而该任务正在等待其他任务，那么cancel()是否会自动取消这些被等待的子任务呢？很遗憾，cancel()只会按照该任务中的await表达式的顺序寻找第一个可被取消的子任务，而其他的子任务则不受影响。下面的例子证明了这点：

```
#!/usr/bin/env python3

import asyncio
import sys

#引用两个子任务的全局变量。
task1 = None
task2 = None


#该协程函数用于输出平方表的自变量。
async def n(bound):
    for i in range(bound):
        print('n:', i)
        await asyncio.sleep(1)


#该协程函数用于输出平方表的因变量。
async def n_square(bound):
    for i in range(bound):
        print('n square:', i * i)
        await asyncio.sleep(1)


#该协程函数输出平方表，并将前两个协程函数作为子任务。
async def print_square_table(bound):
    global task1, task2
    task1 = asyncio.create_task(n(bound))
    task2 = asyncio.create_task(n_square(bound))
    await task1
    await task2


#该协程函数在2.9秒后取消输出平方表的任务。
async def cancel_print(task):
    await asyncio.sleep(2.9)
    task.cancel()


#主协程函数将第一个命令行参数解读为输出到平方表的第几项。
async def main():
    global task1, task2
    if len(sys.argv) < 2:
        sys.exit(1)
    try:
        bound = int(sys.argv[1])
        #创建输出平方表的任务。
        task = asyncio.create_task(print_square_table(bound))
        #创建取消输出平方表的任务。
        cancel_task = asyncio.create_task(cancel_print(task))
        await task
        await cancel_task
    except asyncio.CancelledError:
        pass
    #在接收到asyncio.CancelledError后显示输出平方表的任务和它的两个子任务的当前
    # 状态。
    print("task.cancelled():", task.cancelled())
    print("task.done():", task.done())
    print("task1.cancelled():", task1.cancelled())
    print("task1.done():", task1.done())
    print("task2.cancelled():", task2.cancelled())
    print("task2.done():", task2.done())
    #让主协程持续运行到task1和task2的状态都变成“已完成”或“已取消”。
```

```
        while not task1.done() or not task2.done():
            await asyncio.sleep(1)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine8.py，然后通过如下命令行验证：

```
$ python3 coroutine8.py 5
n: 0
n square: 0
n: 1
n square: 1
n: 2
n square: 4
task.cancelled(): True
task.done(): True
task1.cancelled(): True
task1.done(): True
task2.cancelled(): False
task2.done(): False
n square: 9
n square: 16
```

该结果证明task被取消后，task1自动被取消，但task2依然在运行。

有些时候，我们想保证一个任务在等待它的任务被取消后依然能继续运行，此时应使用asyncio.shield()，其语法为：

```
awaitable asyncio.shield(aw)
```

注意该函数的aw参数可被传入任意可等待对象；而其返回值也是一个可等待对象。特别的，如果通过aw参数传入协程，则asyncio.shield()返回的可等待对象自动变成包装它的任务。请将coroutine8.py中的print_square_table()协程函数修改为：

```
async def print_square_table(bound):
    global task1, task2
    task1 = asyncio.create_task(n(bound))
    task2 = asyncio.create_task(n_square(bound))
    await task2
    await asyncio.shield(task1)
```

再次通过如下命令行验证：

```
$ python3 coroutine8.py 5
n: 0
n square: 0
n: 1
n square: 1
n: 2
n square: 4
task.cancelled(): True
task.done(): True
task1.cancelled(): False
task1.done(): False
task2.cancelled(): True
task2.done(): True
n: 3
n: 4
```

该结果证明task被取消后，task1由于受到保护依然运行，而task2被自动取消。

显然，我们也可以通过asyncio.shield()同时保护task1和task2。但如果想让取消task的操作能同时自动取消task1和task2该怎么做呢？一个较好的解决方案是在task的函数体中通过try语句截获并处理asyncio.CancelledError异常，手工调用task1和task2的cancel()，然后再将asyncio.CancelledError异常抛出。请将print_square_table()协程函数修改为：

```
async def print_square_table(bound):
    global task1, task2
    try:
        task1 = asyncio.create_task(n(bound))
        task2 = asyncio.create_task(n_square(bound))
        await task1
        await task2
    except asyncio.CancelledError:
        task1.cancel()
        task2.cancel()
        raise
```

然后第三次通过如下命令行验证：

```
$ python3 coroutine8.py 5
n: 0
n square: 0
n: 1
n square: 1
n: 2
n square: 4
task.cancelled(): True
task.done(): True
task1.cancelled(): True
task1.done(): True
task2.cancelled(): True
task2.done(): True
```

该结果正是我们想要的。

上面给出的解决方案是具有通用性的。在实际的异步脚本中，经常会存在这样的情况：任务A等待了任务B1、B2、……，任务B1、B2、……又分别等待了任务C1、C2、……，依此类推，形成了一个树状结构。如果想使得调用A的cancel()能自动取消整棵树中的所有任务，那么所有任务都需要使用上述解决方案，递归地显式调用所有子任务的cancel()。这里需要注意，被asyncio.shield()保护的任务也可以通过显式调用cancel()属性来取消，该保护只针对其父任务的cancel()被调用而其自身的cancel()未被调用的情况。

类似的，如果任务A等待了任务B1和B2，B1抛出的异常会传递给A，导致A也抛出异常，但该异常不会影响到B2。这会造成A和B1进入“已完成”状态而B2继续运行。因此如果一个任务想保证不论被取消还是抛出了其他异常都自动取消所有子任务，被其包装的协程对应的协程函数的函数体的结构应该是：

```
async def some_coroutine(args...):
    try:
        ... any code ...
    except Exception:
        ... codes to cancel all sub-coroutines ...
        raise
    except asyncio.CancelledError:
        ... codes to cancel all sub-coroutines ...
        raise
```

现在让我们把关注点转向上面例子中让task在3秒后自动取消的办法。为了达到这一目的，该例子创建了一个任务cancel_task，使得它在3秒后调用task的cancel()。事实上我们还可以做得更漂亮，即使用asyncio.wait_for()来精简代码。该函数的语法为：

```
awaitable asyncio.wait_for(aw, timeout)
```

其中aw参数可传入任意可等待对象，而timeout参数用于指定等待该对象完成的最长时间。与asyncio.shield()类似，当aw参数被传入协程时，asyncio.wait_for()会自动返回包装该协程的任务。

当被asyncio.wait_for()限制的任务超时，会抛出asyncio.CancelledError异常，然而该异常在被该任务抛出后，会被asyncio.wait_for()截获并转换成asyncio.TimeoutError异常。下面的代码是对coroutine8.py的修改，注意它不再包含cancel_print()协程函数，而main()的函数体中则使用了asyncio.wait_for()来限制task的执行时间：

```
#!/usr/bin/env python3

import asyncio
import sys

task1 = None
```

```
    task2 = None


    async def n(bound):
        for i in range(bound):
            print('n:', i)
            await asyncio.sleep(1)


    async def n_square(bound):
        for i in range(bound):
            print('n square:', i * i)
            await asyncio.sleep(1)


    async def print_square_table(bound):
        global task1, task2
        try:
            task1 = asyncio.create_task(n(bound))
            task2 = asyncio.create_task(n_square(bound))
            await task1
            await task2
        except asyncio.CancelledError:
            task1.cancel()
            task2.cancel()
            raise


    async def main():
        global task1, task2
        if len(sys.argv) < 2:
            sys.exit(1)
        try:
            bound = int(sys.argv[1])
            task = asyncio.create_task(print_square_table(bound))
            #使得等待task完成的时间不超过2.9秒。
            waited_task = asyncio.wait_for(task, 2.9)
            await waited_task
        #这里需要处理的是asyncio.TimeoutError异常而非asyncio.CancelledError异常。
        except asyncio.TimeoutError:
            pass
        print("task.cancelled():", task.cancelled())
        print("task.done():", task.done())
        print("task1.cancelled():", task1.cancelled())
        print("task1.done():", task1.done())
        print("task2.cancelled():", task2.cancelled())
        print("task2.done():", task2.done())
        while not task1.done() or not task2.done():
            await asyncio.sleep(1)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine9.py，然后通过如下语句验证：

```
    $ python3 coroutine9.py 5
    n: 0
    n square: 0
    n: 1
    n square: 1
```

```
n: 2
n square: 4
task.cancelled(): True
task.done(): True
task1.cancelled(): True
task1.done(): True
task2.cancelled(): True
task2.done(): True
```

---

get_name()和set_name()的功能显而易见，分别用于取得和设置一个任务的名字，其语法为：

```
Task.get_name()
Task.set_name(value)
```

这两个函数在某些无法获得引用任务的变量的情况下被用来区分多个任务。需要注意的是，asyncio.create_task()可以直接通过name参数设置任务的名字，因此很少需要使用set_name()。在后面会给出需要给任务命名的例子。

最后，从上面的例子可以看出，“协程是轻量级的线程”这句话并不完全正确。并行执行多个线程时，它们相互间可以合作，即每个线程都是“非抢占式（non-preemptive）”的；但更多的情况是相互间只有竞争，即每个线程都是“抢占式（preemptive）”的。而并行执行多个协程时，它们相互间必须配合，因此每个协程都必然是非抢占式的。

14-4. 任务聚集工具

（标准库：asyncio）

上一节已经说明，如果一个任务通过await表达式并行执行了多个子任务，当该任务被取消时这些子任务中只有一个会被自动取消。对于只有一个主协程的异步脚本来说，如果主协程被取消则意味着程序终止，所有其他协程都会因__del__被调用而自动关闭，因此不论并行执行了多少个子任务都没有关系。但其他任务必须通过try语句自行处理子任务抛出的异常，并手工调用每个子任务的cancel()，这对于并行执行大量子任务的任务来说过于麻烦。asyncio模块提供了三个（协程）函数来将一个任务的所有子任务打包成一个子任务，这样不需要使用try语句就可以实现自动取消子任务。

第一个函数是asyncio.gather()，其语法为：

```
awaitable asyncio.gather(*aws, return_exceptions=False)
```

该函数的功能是将多个任务聚集在一起，包装成一个任务。注意aws参数可以被传入任意多个（但不能是0个）可等待对象。如果一个协程被传入aws参数，则它会被自动被包装成一个任务。

asyncio.gather()能够自动处理聚集的任务抛出的异常，其行为规则是：

➤ 如果所有这些任务都执行完成，没有抛出任何异常，那么asyncio.gather()创建的任务也执行完成，结果是一个列表，列表中的第i项是传入aws参数的第i个任务的执行结果。
➤ 如果return_exceptions参数被传入False，而这些任务中至少有一个抛出了异常，那么被抛出的第一个异常会立即成为asyncio.gather()创建的任务的执行结果，并被它抛出；但其余任务会继续被执行，除非被抛出的异常导致程序终止。
➤ 如果return_exceptions参数被传入True，而这些任务中至少有一个抛出了异常，那么所有抛出的异常（包括asyncio.CancelledError）都会被当成相应任务的执行结果，最终成为asyncio.gather()创建的任务的执行结果列表中的项。

此外，如果asyncio.gather()创建的任务的cancel()被调用，那么它聚集的所有任务的cancel()都会自动被调用。

下面验证asyncio.gather()的功能。请将coroutine8.py拷贝到coroutine10.py，然后将协程函数print_square_table()修改为：

```
async def print_square_table(bound):
    global task1, task2
    task1 = asyncio.create_task(n(bound))
    task2 = asyncio.create_task(n_square(bound))
    #将task1和task2聚集成一个任务。
    await asyncio.gather(task1, task2)
```

可以通过如下命令行验证：

```
$ python3 coroutine10.py 5
n: 0
n square: 0
n: 1
n square: 1
n: 2
n square: 4
task.cancelled(): True
task.done(): True
task1.cancelled(): True
task1.done(): True
task2.cancelled(): True
task2.done(): True
```

该结果说明，cancel_task被执行导致task被取消后，由于task没有直接以task1和task2为子任务，而是以“asyncio.gather(task1, task2)”的返回值为子任务，所以后者的cancel()会被自动调用，导致前两者的cancel()也自动被调用。

下面的例子说明了asyncio.gather()聚集的任务抛出异常时的行为：

```
#!/usr/bin/env python3

import asyncio
import sys


#foo()创建的协程会以1秒为间隔打印0~4。
async def foo():
    for i in range(5):
        print("foo:", i)
        await asyncio.sleep(1)
    return i


#bar()创建的协程会以1秒为间隔打印0~2，然后被取消。
async def bar():
    for i in range(5):
        print("bar:", i)
        if i > 1:
            asyncio.current_task().cancel()
        await asyncio.sleep(1)


#baz()创建的协程会以1秒为间隔打印0~3，然后抛出RuntimeError异常。
async def baz():
    for i in range(5):
        print("baz:", i)
        if i > 2:
            raise RuntimeError()
        await asyncio.sleep(1)


async def main():
    result = None
    try:
        #聚集foo()、bar()和baz()。
        task = asyncio.gather(foo(), bar(), baz())
        #task = asyncio.gather(foo(), bar(), baz(), return_exceptions=True)
        await task
    except RuntimeError as e:
        print("RuntimeError")
    except asyncio.CancelledError as e:
        print("CancelledError")
    #等待3秒钟，以确保所有被聚集的任务执行完。
    await asyncio.sleep(3)
    #检查task的状态。
    print("task.cancelled() ==", task.cancelled())
    print("task.done() ==", task.done())
    try:
        print("task.result() ==", task.result())
    except BaseException as e:
        print("task.result() ==", repr(e))
    sys.exit(0)
```

```
    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine11.py，然后通过如下命令行验证：

```
    python3 coroutine11.py
    foo: 0
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    foo: 2
    bar: 2
    baz: 2
    CancelledError
    foo: 3
    baz: 3
    foo: 4
    task.cancelled() == False
    task.done() == True
    task.result() == CancelledError()
```

该结果说明，asyncio.gather()的return_exceptions参数被传入False时，bar()被取消后抛出的asyncio.CancelledError会立即使asyncio.gather()创建的任务执行完成，但该任务虽然以asyncio.CancelledError为结果但并没有被取消，而baz()会在1秒后抛出RuntimeError异常，foo()则会运行到返回4。

请将上面例子的main()中包含asyncio.gather()的语句修改为：

```
    task = asyncio.gather(foo(), bar(), baz(), return_exceptions=True)
```

然后再次通过如下命令行验证：

```
    python3 coroutine11.py
    foo: 0
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    foo: 2
    bar: 2
    baz: 2
    foo: 3
    baz: 3
    foo: 4
    task.cancelled() == False
    task.done() == True
    task.result() == [4, CancelledError(''), RuntimeError()]
```

该结果说明，asyncio.gather()的return_exceptions参数被传入True时，不论是bar()被取消还是baz()抛出RuntimeError异常都不会使asyncio.gather()创建的任务执行完成，它会一直等到foo()执行完成，并以一个列表为结果，该列表聚集了正常的返回值和异常。

第二个是协程函数asyncio.wait()，其语法为：

```
    coroutine asyncio.wait(aws, *, timeout=None,
return_when=asyncio.ALL_COMPLETED)
```

其中aws参数需被传入一个包含任务的可迭代对象（必须保证非空且元素都是任务）；timeout用于控制等待的最长时间，传入的整数或浮点数被解读为秒数；return_when参数则用于设置返回条件。注意这是一个协程函数，如果不将其创建的协程转化为任务，直接等待它将导致线程被挂起。下面介绍的其他协程函数也是这样。

asyncio.wait()创建的协程的具体行为通过timeout参数和return_when参数来控制。return_when接受的值被总结在表14-5中。如果timeout被传入了一个数字，那么即便没有任务被取消或抛出异常，该函数在超时后也会返回，但不会抛出asyncio.TimeoutError异常。需要注意的是，asyncio.wait()创建的协程返回后，尚未完成的任务会继续运行。在任何情况下，聚集的任务抛出的异常都不会传递到asyncio.wait()之外。

表14-5. return_when参数可取值

值	说明
asyncio.ALL_COMPLETED	所有任务都执行完成或被取消时返回。
asyncio.FIRST_COMPLETED	只要有一个任务执行完成或被取消时就返回。
asyncio.FIRST_EXCEPTION	只要有一个任务抛出以Exception为基类的异常时就返回，否则在所有任务都执行完成或被取消时返回。

asyncio.wait()会返回一个二元组(done, pending)，其两个元素都是集合，其中done包含了所有已完成或已取消的任务，而pending包含了所有未完成的任务。由于done和pending都是集合，其内的元素没有固定顺序，所以当我们通过迭代这两个集合分别取得已完成或已取消任务和未完成任务时，无法判断取得的任务被哪个变量引用，此时就需要通过任务的名字来区分它们。

下面通过例子说明asyncio.wait()的用法。请将coroutine11.py拷贝到coroutine12.py，然后将main()修改为：

```
    async def main():
        #创建任务时设置了任务名字。
        task1 = asyncio.create_task(foo(), name="foo")
        task2 = asyncio.create_task(bar(), name="bar")
        task3 = asyncio.create_task(baz(), name="baz")
```

```
        #将三个任务合并成一个协程来运行。
        coro = asyncio.wait([task1, task2, task3])
        #coro = asyncio.wait([task1, task2, task3],
return_when=asyncio.FIRST_COMPLETED)
        #coro = asyncio.wait([task1, task2, task3],
return_when=asyncio.FIRST_EXCEPTION)
        #coro = asyncio.wait([task1, task2, task3], timeout=1.5)
        done, pending = await coro
        #查看已完成或已取消任务。
        print(f"\n{len(done)} tasks done:")
        for task in done:
            print(f"{task.get_name()}.cancelled() == {task.cancelled()}")
            print(f"{task.get_name()}.done() == {task.done()}")
            try:
                print(f"{task.get_name()}.result() == {task.result()}\n")
            except BaseException as e:
                print(f"{task.get_name()}.result() == {repr(e)}\n")
        #查看未完成任务。
        print(f"\n{len(pending)} tasks pending:")
        for task in pending:
           print(f"{task.get_name()}.cancelled() == {task.cancelled()}")
            print(f"{task.get_name()}.done() == {task.done()}")
            try:
                print(f"{task.get_name()}.result() == {task.result()}\n")
            except BaseException as e:
                print(f"{task.get_name()}.result() == {repr(e)}\n")
        sys.exit(0)
```

通过如下命令行来验证：

```
    $ python3 coroutine12.py
    foo: 0
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    foo: 2
    bar: 2
    baz: 2
    foo: 3
    baz: 3
    foo: 4

    3 tasks done:
    foo.cancelled() ==  False
    foo.done() ==  True
    foo.result() ==  4

    baz.cancelled() ==  False
    baz.done() ==  True
    baz.result() ==  RuntimeError()

    bar.cancelled() ==  True
    bar.done() ==  True
    bar.result() ==  CancelledError()


    0 tasks pending:
```

该结果说明foo()、bar()和baz()都执行完成后，asyncio.wait()创建的协程才返回。

然后将上面例子中包含asyncio.wait()的语句修改成：

```
    coro = asyncio.wait([task1, task2, task3],
return_when=asyncio.FIRST_COMPLETED)
```

同样通过如下命令行验证：

```
    $ python3 coroutine12.py
    foo: 0
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    foo: 2
    bar: 2
    baz: 2

    1 tasks done:
    bar.cancelled() == True
    bar.done() == True
    bar.result() == CancelledError()


    2 tasks pending:
    foo.cancelled() == False
    foo.done() == False
    foo.result() == InvalidStateError('Result is not set.')

    baz.cancelled() == False
    baz.done() == False
    baz.result() == InvalidStateError('Result is not set.')
```

该结果说明bar()被取消后asyncio.wait()创建的协程就已经返回，而此时foo()和baz()尚未完成，会继续运行下去。

再然后将上面例子中包含asyncio.wait()的语句修改成：

```
    coro = asyncio.wait([task1, task2, task3],
return_when=asyncio.FIRST_EXCEPTION)
```

再次验证：

```
    $ python3 coroutine12.py
    foo: 0
    bar: 0
    baz: 0
```

```
foo: 1
bar: 1
baz: 1
foo: 2
bar: 2
baz: 2
foo: 3
baz: 3

2 tasks done:
baz.cancelled() == False
baz.done() == True
baz.result() == RuntimeError()

bar.cancelled() == True
bar.done() == True
bar.result() == CancelledError()


1 tasks pending:
foo.cancelled() == False
foo.done() == False
foo.result() == InvalidStateError('Result is not set.')
```

该结果说明bar()被取消并没有让asyncio.wait()创建的协程返回，该协程运行到baz()抛出RuntimeError异常，而此时foo()尚未完成，会继续运行下去。

最后将上面例子中包含asyncio.wait()的语句修改成：

```
coro = asyncio.wait([task1, task2, task3], timeout=1.5)
```

验证：

```
$ python3 coroutine12.py
foo: 0
bar: 0
baz: 0
foo: 1
bar: 1
baz: 1

0 tasks done:

3 tasks pending:
foo.cancelled() == False
foo.done() == False
foo.result() == InvalidStateError('Result is not set.')

baz.cancelled() == False
baz.done() == False
baz.result() == InvalidStateError('Result is not set.')

bar.cancelled() == False
bar.done() == False
bar.result() == InvalidStateError('Result is not set.')
```

该结果说明asyncio.wait()创建的协程在1.5秒后返回，此时foo()、bar()和baz()都没有完成，也没有任何异常被抛出。

从上面的例子可以看出，asyncio.wait()创建的协程返回后，其聚集的未完成任务可以通过pending获得，而我们应该迭代该集合，调用所有未完成任务的cancel()，以避免浪费计算机系统的资源。

第三个函数是asyncio.as_completed()，其语法为：

```
asyncio.as_completed(aws, *, timeout=None)
```

其aws参数和timeout参数的含义与在asyncio.wait()中相同。

使用asyncio.as_completed()时的基本假设是其聚集的所有任务都会执行完成。该函数返回一个由协程构成的可迭代对象，我们需要迭代该可迭代对象，将取得的协程作为await表达式的操作数来并行执行这些任务，即：

```
for coro in asyncio.as_complete(...):
    result = await coro
    ... other code ...
```

该可迭代对象返回协程的顺序将遵循任务完成的先后顺序，因此result会依次取得每个协程的执行结果。

那么，如果某个任务被取消，或者抛出了异常会怎么样呢？这会导致可迭代对象立刻终止迭代，并传递相关异常。此外，当通过timeout参数设置了最大等待时间，超时会导致该可迭代对象终止迭代并抛出asyncio.TimeoutError异常。然而可迭代对象终止迭代并不会影响其聚集的任务继续运行，我们可以在try语句中手工调用这些任务的cancel()。

下面的代码对coroutine12.py进行较大的修改，以说明asyncio.as_complete()的用法：

```
#!/usr/bin/env python3

import asyncio
import sys


async def foo():
    for i in range(5):
        print("foo:", i)
        await asyncio.sleep(1)
    return i


async def bar():
    for i in range(3):
```

```
            print("bar:", i)
            await asyncio.sleep(1)
        return i


    async def baz():
        for i in range(4):
            print("baz:", i)
            await asyncio.sleep(1)
        return i


    async def main():
        task1 = asyncio.create_task(foo(), name = "foo")
        task2 = asyncio.create_task(bar(), name = "bar")
        task3 = asyncio.create_task(baz(), name = "baz")
        try:
            #并行运行三个任务。
            for coro in asyncio.as_completed([task1, task2, task3]):
                #取得任务执行结果并显示。
                result = await coro
                print(result)
        except (asyncio.CancelledError, asyncio.TimeoutError, RuntimeError) as
e:
            #如果抛出异常，则显示。
            print(repr(e))
        finally:
            #取得迭代停止时三个任务的信息，然后取消未完成的任务。
            for i in range(1, 4):
                codes = f"""
    print(f'\\n{{task{i}.get_name()}}.cancelled() == {{task{i}.cancelled()}}')
    print(f'{{task{i}.get_name()}}.done() == {{task{i}.done()}}')
    try:
        print(f'{{task{i}.get_name()}}.result() == {{task{i}.result()}}')
    except BaseException as e:
        print(f'{{task{i}.get_name()}}.result() == {{repr(e)}}')
    if not task{i}.cancelled():
        task{i}.cancel()"""
                exec(codes)
    sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

注意此时foo()、bar()和baz()都不会抛出异常，但分别于5、3和4秒钟后完成。请将上述代码保存为coroutine13.py，然后通过如下命令行验证：

```
    python3 coroutine13.py
    foo: 0
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    foo: 2
    bar: 2
    baz: 2
    foo: 3
    baz: 3
    2
```

```
foo: 4
3
4
foo.cancelled() == False
foo.done() == True
foo.result() == 4

bar.cancelled() == False
bar.done() == True
bar.result() == 2

baz.cancelled() == False
baz.done() == True
baz.result() == 3
```

该结果说明asyncio.as_complete()返回的可迭代对象被迭代了3次，依次获得了bar()、baz()和foo()的执行结果，而这个顺序也就是这三个任务完成的先后顺序。

下面将bar()修改为：

```
async def bar():
    for i in range(3):
        print("bar:", i)
        if i > 1:
            asyncio.current_task().cancel()
        await asyncio.sleep(1)
    return i
```

这样它会在输出了0和1之后取消自身。通过如下命令行验证：

```
python3 coroutine13.py
foo: 0
bar: 0
baz: 0
foo: 1
bar: 1
baz: 1
CancelledError()

foo.cancelled() == False
foo.done() == False
foo.result() == InvalidStateError('Result is not set.')

bar.cancelled() == True
bar.done() == True
bar.result() == CancelledError()

baz.cancelled() == False
baz.done() == False
baz.result() == InvalidStateError('Result is not set.')
```

该结果说明asyncio.as_complete()返回的可迭代对象一次都没被迭代就抛出了来自bar()的asyncio.CancelledError异常，而在它终止迭代时foo()和baz()还未完成。

请将bar()还原，然后将baz()修改为：

```
async def baz():
    for i in range(4):
        print("baz:", i)
        if i > 2:
            raise RuntimeError()
        await asyncio.sleep(1)
    return i
```

这样它会在输出了0、1、2和3后抛出RuntimeError异常。通过如下命令行验证：

```
$ python3 coroutine13.py
foo: 0
bar: 0
baz: 0
foo: 1
bar: 1
baz: 1
foo: 2
bar: 2
baz: 2
foo: 3
baz: 3
2
RuntimeError()

foo.cancelled() == False
foo.done() == False
foo.result() == InvalidStateError('Result is not set.')

bar.cancelled() == False
bar.done() == True
bar.result() == 2

baz.cancelled() == False
baz.done() == True
baz.result() == RuntimeError()
```

该结果说明asyncio.as_complete()返回的可迭代对象被迭代了一次，获得了bar()的执行结果，然后抛出了来自baz()的RuntimeError异常，此时foo()还未完成。

请将baz()还原，然后将包含asyncio.as_completed()的语句改为：

```
for coro in asyncio.as_completed([task1, task2, task3], timeout=1.5):
```

验证：

```
$ python3 coroutine13.py
foo: 0
```

```
    bar: 0
    baz: 0
    foo: 1
    bar: 1
    baz: 1
    TimeoutError()

    foo.cancelled() == False
    foo.done() == False
    foo.result() == InvalidStateError('Result is not set.')

    bar.cancelled() == False
    bar.done() == False
    bar.result() == InvalidStateError('Result is not set.')

    baz.cancelled() == False
    baz.done() == False
    baz.result() == InvalidStateError('Result is not set.')
```

该结果说明asyncio.as_complete()返回的可迭代对象在所有任务完整之前就停止了迭代。

上述三个（协程）函数都具有如下两种局限性：

➤ 父任务执行完成并不意味着其聚集的子任务执行完成，因此经常需要手工调用每个子任务的cancel()。
➤ 一旦开始执行后，就不能添加子任务。

Python 3.11引入的TaskGroup类则没有这些限制，使用起来更简便，其实例化语法为：

```
    class asyncio.TaskGroup()
```

一个TaskGroup对象就代表一个任务组，只具有一个属性create_task()作为asyncio.create_task()函数的替代，其语法为：

```
    asyncio.TaskGroup.create_task(coro, *, name=None, context=None)
```

除context参数用于自定义上下文之外，其余参数的含义与在asyncio.create_task()中相同。TaskGroup对象同时还是一个异步上下文管理器。在本章的最后会详细讨论异步上下文管理器，这里只需要知道它们的用法与上下文管理器类似，但需要将with语句替换为async with语句。

TaskGroup类的使用方法是先实例化出一个空任务组，然后在相应async with语句的代码块中调用create_task()属性以向该任务组添加任务，这些任务在创建后会立刻被执行。正常情况下，任务组会等待所有任务执行完成，在此期间随时可以通过调用相应TaskGroup对象

的create_task()属性以添加新的任务。当没有剩余的任务需要执行时，TaskGroup对象的__aexit__就会被调用，并被传入(None, None, None)，意味着async with语句执行完成。此后其create_task()属性不再能被调用，因此该对象也失去了作用，应被丢弃。

任务组中某个任务被取消，因此抛出asyncio.CancelledError异常，并不会导致相应TaskGroup对象的__aexit__被调用。但任务组中的某个任务抛出任何其他异常，都会导致相应TaskGroup对象的__aexit__被调用，并被传入该异常的相关信息。如果没有任何任务抛出异常，但async with语句的代码块中的某条语句抛出了异常，同样会导致TaskGroup对象的__aexit__被调用，并被传入该异常的相关信息。

异常导致TaskGroup对象的__aexit__被调用时，会依次调用任务组中每个未完成任务的cancel()，而这可能导致其他异常被抛出。__aexit__会将作为它的实际参数被传入的异常，以及取消所有未完成任务的过程中抛出的asyncio.CancelledError之外的异常，都打包成一个异常组，然后再传递到async with语句之外。该异常组具体是ExceptionGroup还是BaseExceptionGroup取决于收集到的异常的类型。

值得强调的是，如果导致__aexit__被调用的异常是KeyboardInterrupt或SystemExit，那么__aexit__依然会自动取消所有未完成任务，但此后会直接抛出该异常，而不把它包装成一个异常组。此外，这两种异常还会被暂时储存起来，当包含async with语句的任务执行完成时再自动抛出，即便通过try语句截获并处理了它也不影响这一行为。

下面的例子说明了如何使用TaskGroup对象：

```
#!/usr/bin/env python3

import asyncio
import sys

#用一个全局变量来引用动态增添的任务。
task3 = None


#该协程函数的参数tg需被传入一个TaskGroup对象。
async def foo(tg):
    global task3
    for i in range(5):
        #取得当前所有未完成的任务。
        unfinished_tasks = asyncio.all_tasks()
        #显示所有未完成任务的名字。
        s = "unfinished tasks: "
        for t in unfinished_tasks:
            s += t.get_name() + " "
        print(s)
        #1秒后动态给任务组增添任务baz。
        if i == 1:
            task3 = tg.create_task(baz(), name="baz")
        print("foo:", i)
        await asyncio.sleep(1)
    return i


async def bar():
    for i in range(3):
        print("bar:", i)
```

```
            #2秒后取消任务bar。
            if i == 2:
                asyncio.current_task().cancel()
            await asyncio.sleep(1)
        return i


    async def baz():
        for i in range(4):
            print("baz:", i)
            await asyncio.sleep(1)
        return i


    async def main():
        try:
            #创建一个任务组，使其包含任务foo和bar。
            async with asyncio.TaskGroup() as tg:
                task1 = tg.create_task(foo(tg), name="foo")
                task2 = tg.create_task(bar(), name="bar")
        except (KeyboardInterrupt, BaseExceptionGroup, ExceptionGroup) as e:
            #截获TaskGroup对象抛出的异常并显示。
            print(repr(e))
        finally:
            #显示任务组完成时各任务的状态。
            for i in range(1, 4):
                codes = f"""
    print(f'\\n{{task{i}.get_name()}}.cancelled() == {{task{i}.cancelled()}}')
    print(f'{{task{i}.get_name()}}.done() == {{task{i}.done()}}')
    try:
        print(f'{{task{i}.get_name()}}.result() == {{task{i}.result()}}')
    except BaseException as e:
        print(f'{{task{i}.get_name()}}.result() == {{repr(e)}}')
    if not task{i}.cancelled():
        task{i}.cancel()"""
                exec(codes)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine14.py，然后通过如下命令行验证：

```
$ python3 coroutine14.py
unfinished tasks: foo Task-1 bar
foo: 0
bar: 0
unfinished tasks: foo Task-1 bar
foo: 1
bar: 1
baz: 0
unfinished tasks: foo Task-1 bar baz
foo: 2
bar: 2
baz: 1
unfinished tasks: foo baz Task-1
foo: 3
baz: 2
unfinished tasks: foo baz Task-1
foo: 4
```

```
    baz: 3

    foo.cancelled() == False
    foo.done() == True
    foo.result() == 4

    bar.cancelled() == True
    bar.done() == True
    bar.result() == CancelledError()

    baz.cancelled() == False
    baz.done() == True
    baz.result() == 3
```

该结果证明了如下两点：

- 在TaskGroup对象代表的任务组开始运行后，依然可以动态给它增加任务。
- 任务组中的一个任务被取消，既不影响其他任务的运行，也不影响任务组本身的运行。

下面将baz()修改为：

```
async def baz():
    for i in range(4):
        print("baz:", i)
        if i == 2:
            raise RuntimeError()
        await asyncio.sleep(1)
    return i
```

再次验证：

```
$ python3 coroutine14.py
unfinished tasks: foo Task-1 bar
foo: 0
bar: 0
unfinished tasks: foo Task-1 bar
foo: 1
bar: 1
baz: 0
unfinished tasks: foo Task-1 bar baz
foo: 2
bar: 2
baz: 1
unfinished tasks: foo baz Task-1
foo: 3
baz: 2
ExceptionGroup('unhandled errors in a TaskGroup', [RuntimeError()])

foo.cancelled() == True
foo.done() == True
foo.result() == CancelledError()
```

```
    bar.cancelled() == True
    bar.done() == True
    bar.result() == CancelledError()

    baz.cancelled() == False
    baz.done() == True
    baz.result() == RuntimeError()
```

该结果证明任务组中的某个任务抛出asyncio.CancelledError之外的异常时，会导致整个任务组停止运行，其中尚未完成的任务会被自动取消，而该异常则会被整合到异常组中抛出。

最后将baz()修改为抛出KeyboardInterrupt异常而非RuntimeError异常：

```
async def baz():
    for i in range(4):
        print("baz:", i)
        if i == 2:
            raise KeyboardInterrupt()
        await asyncio.sleep(1)
    return i
```

第三次验证：

```
$ python3 coroutine14.py
unfinished tasks: foo Task-1 bar
foo: 0
bar: 0
unfinished tasks: foo Task-1 bar
foo: 1
bar: 1
baz: 0
unfinished tasks: foo Task-1 bar baz
foo: 2
bar: 2
baz: 1
unfinished tasks: foo baz Task-1
foo: 3
baz: 2
KeyboardInterrupt()

foo.cancelled() == True
foo.done() == True
foo.result() == CancelledError()

bar.cancelled() == True
bar.done() == True
bar.result() == CancelledError()

baz.cancelled() == False
baz.done() == True
baz.result() == KeyboardInterrupt()
... other information about reraised KeyboardInterrupt Exception ...
```

该结果证明KeyboardInterrupt异常会被特殊处理。

最后值得说明，在上面的例子中用到了这样一个技巧——可以通过all_tasks()函数以一个集合的形式取得指定事件循环的当前所有未完成任务，其语法为：

```
asyncio.all_tasks(loop=None)
```

其中loop参数的含义与在asyncio.current_task()中相同。

14-5. 同步原语——屏障和事件

（标准库：asyncio）

上面给出的所有例子都采用了这样的模型：主协程并行执行若干任务，这些任务独立运行，相互间没有交换数据，也没有有效的配合，唯一的交互方式是一个任务调用另一个任务的cancel()。现实中异步程序的任务会以更复杂的方式频繁地交互：或者是以设计好的方式输出内容，或者是交换数据。不论哪种情况，每次交互都意味着相关任务在某个时间点完成了同步，而接下来的几节将讨论用于实现同步的技术。这是异步编程中最难也最关键的部分，当你掌握了它们之后就能够编写出任意复杂的异步程序了。

需要强调，下面讨论的同步技术不局限于同步任务（协程），也用于同步进程、线程乃至硬件（例如多核CPU）。这些技术通过“同步原语（synchronization primitives）”来描述，属于计算机科学的理论部分。虽然这些同步原语都可被用于多种场景，但每种同步原语都是针对某种特定场景被提出的。下面将按照它们被提出时所针对的场景从简单到复杂的顺序，逐一介绍这些同步原语。

首先考虑最简单的场景：多个并行运行的任务需要在某时刻同步各自的执行进度，即只要有一个任务还没有执行到指定的进度，其余任务就必须等待。为解决这一问题而设计的同步原语是“屏障（barrier）”。

asyncio定义了Barrier类来表示屏障，其实例化语法为：

```
class asyncio.Barrier(parties)
```

其中parties参数是一个正整数，代表翻越该屏障需要的任务数。

Barrier对象具有表14-6列出的属性。需要注意的是，asyncio.Barrier是Python 3.11引入的，Python 3.10及之前的版本不支持。（当你使用这些版本时需要通过事件以更繁琐的方式实现屏障的功能。）

表14-6. 屏障的属性

属性	说明
parties	翻越该屏障所需的任务数。
n_waiting	当前等待翻越该屏障的任务数。
broken	引用一个布尔值，表明该屏障是否已经损坏。
wait()	一个协程函数，等待翻越该屏障。
abort()	一个协程函数，放弃翻越该屏障。
reset()	一个协程函数，重置该屏障。

屏障的使用方法很简单：假设有k个任务需要同步执行进度。先以k为参数实例化出一个Barrier对象，该对象的初始状态为parties引用k，n_waiting引用0，broken引用False。然后在每个任务的目标进度处等待通过调用Barrier对象的wait()创建的协程，其语法为：

```
coroutine asyncio.Barrier.wait()
```

该协程函数每被调用一次，n_waiting就会增加1。如果此时n_waiting仍然小于parties，则新建协程会被挂起，使得等待它的任务也被挂起，代表该任务被屏障阻挡；否则n_waiting会被重置为0，新建协程和之前被挂起的所有协程都执行完成，并返回其被创建时n_waiting的值，代表所有任务都翻越了屏障。所有任务都翻越屏障后，Barrier对象将回到初始状态，因此可以重复使用。

以上描述的是理想流程。有可能某个任务永远也无法运行到指定的进度，在这种情况下其他任务就会卡死在屏障前，永远得不到运行的机会。为了避免这一现象发生，可以设置一个最大等待时间，如果超时则等待通过调用Barrier对象的abort()创建的协程，其语法为：

```
coroutine asyncio.Barrier.abort()
```

该协程执行完成后会将Barrier对象的broken设置为True，然后使所有已经被该屏障阻挡的任务上抛出asyncio.BrokenBarrierError异常（该异常是RuntimeError的直接子类）。这些任务应自行截获并处理asyncio.BrokenBarrierError异常，以继续运行；否则该异常会导致任务执行完成，并被视为这些任务抛出的异常。

当一个Barrier对象的broken引用True，等待通过调用该对象的wait()创建的协程会导致立刻抛出asyncio.BrokenBarrierError异常，意味着该对象不再是一个屏障。此时如果还需要使用该屏障，只能丢弃该对象，并重新创建一个Barrier对象。如果想重用一个Barrier对象，就不要使用abort()，而是等待通过调用该对象的reset()创建的协程，相关语法为：

***coroutine* asyncio.Barrier.reset()**

该协程执行完成后同样会使所有已经被该屏障阻挡的任务抛出asyncio.BrokenBarrierError异常，但会将Barrier对象的n_waiting和broken重置到初始状态。

下面用一个很有趣的例子coroutine15.py来说明如何使用屏障来实现同步。在该例子中任务hiker1、hiker2和hiker3被抽象成三个徒步旅行者，它们需要随机花费0.5~7秒才能走到屏障前；而任务hammer则被抽象成一个锤子，它会在5秒后破坏掉屏障：

```
#!/usr/bin/env python3

import asyncio
import random
import time
import sys


#该协程函数创建的协程代表徒步旅行者。
async def hiker(name, barrier):
    try:
        #该徒步旅行者随机花费0.5~7.0秒走到屏障前。
        t = random.uniform(0.5, 7.0)
        await asyncio.sleep(t)
        #显示该徒步旅行者走到屏障前的时间，以及屏障的状态。
        print(f"{name}: I meet the barrier at {time.strftime('%X')}.")
        print(f"{name}: n_waiting == {barrier.n_waiting}, broken ==
{barrier.broken}")
        #等待所有徒步旅行者到齐。   如果屏障已经损坏，则立刻抛出
        # asyncio.BrokenBarrierError异常。
        await barrier.wait()
        #显示该徒步旅行者翻越屏障的时间，以及屏障的状态。
        print(f"{name}: I cross the barrier at {time.strftime('%X')}.")
        print(f"{name}: n_waiting == {barrier.n_waiting}, broken ==
{barrier.broken}")
    except asyncio.BrokenBarrierError:
        #显示该旅行者意识到屏障已经损坏的时间，以及屏障的状态。
        print(f"{name}: The barrier collapses before {time.strftime('%X')}.
Hooray!")
        print(f"{name}: n_waiting == {barrier.n_waiting}, broken ==
{barrier.broken}")


#该协程函数创建的协程代表锤子。   它会在5秒钟后破坏屏障。
async def hammer(barrier):
    await asyncio.sleep(5)
    await barrier.abort()
    #await barrier.reset()


async def main():
    #创建一个屏障。
    barrier = asyncio.Barrier(3)
    #创建四个任务并将它们聚集成一个任务。   只有hiker1、hiker2和hiker3需要翻越屏障。
    #  hammer0将破坏屏障。
    hiker1 = asyncio.create_task(hiker("hiker1", barrier))
    hiker2 = asyncio.create_task(hiker("hiker2", barrier))
    hiker3 = asyncio.create_task(hiker("hiker3", barrier))
```

```
        hammer0 = asyncio.create_task(hammer(barrier))
        #显示这次徒步旅行开始的时间，然后开始旅行。
        print(f"Start at {time.strftime('%X')}")
        await asyncio.gather(hiker1, hiker2, hiker3, hammer0)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

下面来验证该例子。由于三个徒步旅行者到达屏障前的时间是随机的，所以能组合出各种不同的结果，但这些结果大体上可分为两类。一类结果如下所示：

```
$ python3 coroutine15.py
Start at 23:07:18
hiker2: I meet the barrier at 23:07:18.
hiker2: n_waiting == 0, broken == False
hiker1: I meet the barrier at 23:07:19.
hiker1: n_waiting == 1, broken == False
hiker3: I meet the barrier at 23:07:20.
hiker3: n_waiting == 2, broken == False
hiker3: I cross the barrier at 23:07:20.
hiker3: n_waiting == 0, broken == False
hiker2: I cross the barrier at 23:07:20.
hiker2: n_waiting == 0, broken == False
hiker1: I cross the barrier at 23:07:20.
hiker1: n_waiting == 0, broken == False
```

该结果表明三个徒步旅行者都在5秒前抵达了屏障，并一同翻越了屏障。针对上面的结果，hiker2、hiker1和hiker3依次于不同的时间抵达了屏障，但翻越屏障在同一秒。在翻越屏障时，最晚到达的hiker3第一个翻越，然后是hiker2，最后是hiker1。这是因为当屏障被翻越的条件达成时，任务继续执行的顺序符合其等待的协程的返回值的排列顺序，亦即该协程被创建时n_waiting的值。hiker2和hiker1等待的协程的返回值分别是1和2。但hiker3等待的协程的返回值是0而不是3，因为它是最后一个到达的，会使得屏障的n_waiting被重置。

另一类结果如下所示：

```
$ python3 coroutine15.py
Start at 22:58:27
hiker2: I meet the barrier at 22:58:29.
hiker2: n_waiting == 0, broken == False
hiker2: The barrier collapses before 22:58:32. Hooray!
hiker2: n_waiting == 0, broken == True
hiker3: I meet the barrier at 22:58:33.
hiker3: n_waiting == 0, broken == True
hiker3: The barrier collapses before 22:58:33. Hooray!
hiker3: n_waiting == 0, broken == True
hiker1: I meet the barrier at 22:58:33.
hiker1: n_waiting == 0, broken == True
hiker1: The barrier collapses before 22:58:33. Hooray!
hiker1: n_waiting == 0, broken == True
```

该结果表明屏障被破坏时至少还有一个徒步旅行者没有到达屏障，这样已经到达的徒步旅行者会在屏障被破坏时通过屏障，而后来到达的徒步旅行者无需等待就可以直接通过屏障。针对上面的结果，hiker2到达屏障后，等待了3秒钟，然后屏障被破坏使其通过。而hiker3和hiker1都是在屏障被破坏后到达屏障的，因此没有等待。

下面对上面的例子做一点小小的修改，即将hammer()中的“await barrier.abort()”改成：

```
        await barrier.reset()
```

这意味着hammer0的作用是先破坏屏障，然后再让屏障恢复到初始状态。此时验证结果也可以分为两类，一类是三个徒步旅行者都在屏障被破坏前抵达了屏障，然后一起翻越屏障，与之前没有区别；另一了则是屏障被破坏时至少有一个徒步旅行者没能抵达屏障，这样已经抵达屏障的旅行者可以通过，而后抵达的徒步旅行者则会永远被屏障阻挡（因为没有新的徒步旅行者到达），导致程序无法结束。下面的结果属于后一类：

```
    $ python3 coroutine15.py
    Start at 23:07:26
    hiker2: I meet the barrier at 23:07:27.
    hiker2: n_waiting == 0, broken == False
    hiker1: I meet the barrier at 23:07:27.
    hiker1: n_waiting == 1, broken == False
    hiker2: The barrier collapses before 23:07:31. Hooray!
    hiker2: n_waiting == 0, broken == False
    hiker1: The barrier collapses before 23:07:31. Hooray!
    hiker1: n_waiting == 0, broken == False
    hiker3: I meet the barrier at 23:07:31.
    hiker3: n_waiting == 0, broken == False
    ^C
```

至此我们完成了对同步源语屏障的讨论。接下来考虑这样的场景：一个运行中的任务想通知另一个被挂起的任务，自己已经完成了某项操作，因此后者可以继续运行了。该场景关注的是如何在任务间传递消息。为解决这一问题而设计的同步原语是“事件（event）”。

asyncio定义了Event类来表示事件，其实例化语法为：

```
    class asyncio.Event()
```

Event对象具有表14-7列出的属性。

表14-7. 事件的属性

属性	说明
`wait()`	一个协程函数，等待该事件被设置。
`set()`	设置该事件。
`clear()`	取消设置该事件。
`is_set()`	判断该事件是否被设置。

wait()是一个协程函数，其语法为：

```
coroutine asyncio.Event.wait()
```

通过调用它创建的协程具有如下行为：如果该事件已经被设置，则立刻执行完成并返回True；否则挂起到该事件被设置为止。

set()函数的作用是设置该事件，其语法为：

```
asyncio.Event.set()
```

如果该事件已设置则什么也不做。

clear()函数的作用与set()正好相反，将取消该事件的设置，其语法为：

```
asyncio.Event.clear()
```

如果该事件未设置则什么也不做。事件刚被创建时是未被设置的。

is_set()函数的作用是判断该事件是否被设置，其语法为：

```
asyncio.Event.is_set()
```

该函数返回True表示事件已设置；返回False表示事件未设置。

使用Event类时，必须遵守这样的原则：先实例化一个Event对象以代表某一事件；某些任务因等待通过调用该事件的wait()创建的协程而被挂起；一个任务通过调用该事件的set()使因该事件被挂起的所有任务能够继续运行；而所有继续运行的任务都需要调用该事件的clear()，以使该事件可以重新被设置。is_set()则不一定被使用。

下面的例子说明了如何用事件实现同步。它并行执行三个任务：num_gen()产生随机整数，star()显示“*”，dollar()显示“$”。当num_gen()产生了一个偶数时，需要通过设置even_event对象来通知star()，使其显示一个“*”；当num_gen()产生了一个奇数时，需要通过设置odd_event对象来通知dollar()，使其显示一个“$”。注意为了在任务结束后调用事件的is_set()属性有可能返回True，star()和dollar()在“awiat event.wait()”和“event.clear()”之间有意插入了“await asyncio.sleep(0.4)”。而在实际应用中，事件被设置后处理它的任务第一件要做的事就是调用该事件的clear()属性。

```
#!/usr/bin/env python3

import asyncio
import random
import sys


#该协程函数创建的协程当even_event事件被设置时在0.4秒后显示“*”。
async def star(event):
    while True:
        #等待事件被设置。
        await event.wait()
        #这里等待0.4秒是人为添加的，其实不需要。
        await asyncio.sleep(0.4)
        #取消事件的设置。
        event.clear()
        #处理该事件的代码。
        sys.stdout.write("* ")
        sys.stdout.flush()


#该协程函数创建的协程当odd_event事件被设置时在0.4秒后显示“$”。
async def dollar(event):
    while True:
        #等待事件被设置。
        await event.wait()
        #这里等待0.4秒是人为添加的，其实不需要。
        await asyncio.sleep(0.4)
        #取消事件的设置。
        event.clear()
        #处理该事件的代码。
        sys.stdout.write("$ ")
        sys.stdout.flush()


#该协程函数创建的协程每隔0.5秒随机生成一个0~100内的整数。   如果该整数是偶数，则设置
# even_event事件；如果该整数是奇数，则设置odd_event事件。
async def num_gen(event1, event2):
    while True:
        n = random.randrange(100)
        if n % 2 == 0:
            #设置even_event事件。
            event1.set()
        else:
            #设置odd_event事件。
            event2.set()
        await asyncio.sleep(0.5)


async def main():
    #创建even_event事件，代表num_gen()生成了一个偶数。
```

```
        even_event = asyncio.Event()
        #创建odd_event事件，代表num_gen()生成了一个奇数。
        odd_event = asyncio.Event()
        #创建三个任务并将它们聚集成一个任务。
        task_star = asyncio.create_task(star(even_event))
        task_dollar = asyncio.create_task(dollar(odd_event))
        task_num_gen = asyncio.create_task(num_gen(even_event, odd_event))
        task = asyncio.gather(task_star, task_dollar, task_num_gen)
        try:
            #运行任务2.6秒。
            await asyncio.wait_for(task, 2.6)
        except asyncio.TimeoutError:
            print("")
            #判断任务运行完成时是否还有未处理的事件。
            if even_event.is_set():
                #even_event事件未处理。
                print("The cancelled symbol is *.")
            elif odd_event.is_set():
                #odd_event事件未处理。
                print("The cancelled symbol is $.")
            else:
                #所有事件都已经处理。
                print("No symbol is cancelled.")
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上面的代码保存为coroutine16.py，然后通过如下命令行验证：

```
$ python3 coroutine16.py
* * $ * $
The cancelled symbol is *.

$ python3 coroutine16.py
$ * * * $
The cancelled symbol is $.

...
```

14-6. 同步原语——锁和监视器

（标准库：asyncio）

在上一节考虑的两种场景中，任务间只是协调了执行的进度，并不涉及对任何共享资源的访问。当任务间存在对共享资源（例如一块内存空间或一个文件）的访问时，将使同步的难度提高一个级别。如果这些任务对该共享资源的访问不受限制，就可能产生被称为“竞争条件（rase condition）”的现象：执行结果取决于这些任务被执行的顺序，而很多时候该顺序是无法控制的（例如随机睡眠一段时间的任务），导致该结果无法预测。

事件和屏障都无助于避免竞争条件的产生，因此必须设计新的同步原语。可能产生竞争条件的场景有很多种，计算机科学家针对不同场景设计了不同的同步原语，但其设计都采用了这样的思想：程序中代码用于访问共享资源的部分被称为“临界区（critical sections）”，

只要一个任务在进入临界区时使用某种同步原语限制其他任务进入临界区，就可以避免产生竞争条件。

下面考虑这样的场景：一个共享资源在任意时刻最多只允许一个任务访问。该场景在涉及竞争条件的场景中是最简单的，本质上是将资源按某种规则分配给其中一个任务的问题。为解决该问题而设计的同步原语是“锁（lock）”，也被称为“互斥量（mutex）”。

asyncio定义了Lock类来表示锁，其实例化语法为：

```
class asyncio.Lock()
```

Lock对象具有表14-8列出的属性。

表14-8. 锁的属性

属性	说明
`acquire()`	一个协程函数，申请获得该锁。
`release()`	释放该锁。
`locked()`	判断该锁是否已被某任务获得。

锁的名字很形象地暗示了它们的用法：一个锁总是与一个资源相关联，就好像现实中的锁与一扇门相关联一样。当锁未被任何任务获得时，表示该资源可用，就好像现实中未被锁住的门。一个任务想要访问该资源，必须先获得锁，进而获得该资源的排他性访问权，就好像现实中一个人进入门内然后把门锁住一样。而当该任务不再需要访问该资源时，应释放锁，使该资源重新可用，就好像现实中进入门后的人把锁打开，退到门外。

acquire()是用于获得锁的协程函数，其语法为：

```
coroutine asyncio.Lock.acquire()
```

一个任务总是可以靠等待通过调用锁的acquire()创建的协程来申请获得该锁，不论该锁是否已经被另一个任务获得。如果被申请的锁尚未被其他任务获得，则该协程会立刻执行完成并返回True，意味着任务获得了该锁；否则，该协程会被挂起，直到获得了锁的任务将锁释放。当多个任务先后申请获得同一个锁时，第一个申请者会立刻获得锁，第二个申请者需要等到第一个申请者释放锁后才会获得锁，……依此类推。

release()是用于获得锁的函数，其语法为：

asyncio.Lock.release()

只有已经获得了锁的任务可以调用release()来释放该锁，未获得该锁的任务调用release()会导致自身抛出RuntimeError异常。此外，已经获得了锁的任务调用了release()后，其余因申请获得该锁而被挂起的任务并不会立刻被执行，直到该任务因对另一个await表达式求值而触发了任务切换后，其他任务才会获得被执行的机会。

locked()用于判断一个锁是否已被某个任务获得，其语法为：

asyncio.Lock.locked()

当该锁已经被某个任务获得时它返回True；否则返回False。

下面的例子说明了锁的作用：

```
#!/usr/bin/env python3

import asyncio
import sys


#该协程函数创建的协程以0.1秒为间隔连续显示三个“*”。
async def stars(lock):
    try:
        while True:
            #申请获得锁，进入临界区。
            await lock.acquire()
            #显示锁的当前状态。
            print(f"\nstars: locked == {lock.locked()}")
            #显示“*”。
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            #释放锁，离开临界区。
            lock.release()
    except asyncio.CancelledError:
        lock.release()
        raise


#该协程函数创建的协程以0.1秒为间隔连续显示三个“$”。
async def dollars(lock):
    try:
        while True:
            #申请获得锁，进入临界区。
            await lock.acquire()
            #显示锁的当前状态。
            print(f"\ndollars: locked == {lock.locked()}")
            #显示“$”。
            for i in range(3):
                await asyncio.sleep(0.1)
```

```
                    sys.stdout.write("$")
                    sys.stdout.flush()
                #释放锁，离开临界区。
                lock.release()
    except asyncio.CancelledError:
         lock.release()
         raise


async def main():
    #创建一个关联到标准输出的锁。
    lock = asyncio.Lock()
    #创建两个任务并将它们聚集成一个任务。
    task_stars = asyncio.create_task(stars(lock))
    task_dollars = asyncio.create_task(dollars(lock))
    task = asyncio.gather(task_stars, task_dollars)
    try:
        #显示锁的当前状态。
        print(f"\nlocked == {lock.locked()}")
        #运行任务1.1秒。
        await asyncio.wait_for(task, 1.1)
    except asyncio.TimeoutError:
        #显示锁的当前状态。
        print(f"\n\nlocked == {lock.locked()}")
    sys.exit(0)


if __name__ == '__main__':
    asyncio.run(main())
```

请将上面的代码保存为coroutine17.py，并通过如下命令行验证：

```
$ python3 coroutine17.py

locked == False

stars: locked == True
***
dollars: locked == True
$$$
stars: locked == True
***
dollars: locked == True
$

locked == False
```

然后，请将stars()和dollars()中涉及“lock”的语句都注释掉，然后通过如下命令行验证：

```
$ python3 coroutine17.py

locked == False

stars: locked == False

dollars: locked == False
*$*$*
stars: locked == False
```

```
$
dollars: locked == False
*$*$*
stars: locked == False
$
dollars: locked == False
*$*$*
stars: locked == False
$
dollars: locked == False
*$

locked == False
```

下面结合对比coroutine16.py的执行结果和coroutine17.py的两个执行结果来分析锁的功能。其实coroutine16.py中的star()和dollar()也使用了共享资源——sys.stdout代表的标准输出。前面的大量例子中也是这样。然而star()和dollar()不可能产生竞争条件，因为它们对共享资源的访问是一次“原子操作（atomic operation）”——要么全部执行，要么全部不执行，不会在中间被打断。对于单线程程序来说，在协程函数中两次await表达式求值之间的所有语句，以及最后一次await表达式求值直到代码块末尾的所有语句，都可被视为一次原子操作。而对于多线程程序来说，原子操作对应一条简单语句，复合语句都不是原子的，更不用说多条语句。

coroutine17.py中的stars()和dollars()有意在每次输出“*”和“$”后睡眠0.1秒，这样输出“***”和“$$$”就不再是原子操作，可能在中间被打断。当没有使用锁时，并行运行的任务task_stars和task_dollars对标准输出的写操作是交错的，如上面的第二个结果所示。注意该结果依然具有一定的规律性，这源自每次输出“*”或“$”后的睡眠时间固定为0.1秒。如果该睡眠时间是随机的，则输出的“*”和“$”不再是有规律的交错，而是无规律的混杂，完全不可预测，而这反而更接近现实中的异步程序。

锁的作用就是限制非原子操作的执行顺序。从上面第一个结果可以看出，由于task_stars要先于task_dollars被执行，所以task_stars第一次执行“await lock.acquire()”时会获得锁，进而能够继续执行；而task_dollars第一次执行“await lock.acquire()”时会导致它被挂起。task_stars输出“*”后睡眠0.1秒，由于task_dollars和主协程都被挂起，所以会导致整个线程被阻塞，task_dollars无法输出“$”。直到task_stars执行“lock.release()”释放了锁，然后在下一次循环中再次执行“await lock.acquire()”导致自身被挂起后，task_dollars才获得被执行的机会，而它的执行过程与task_stars完全相同，区别仅在于输出的是“$”而非“*”。

如果主协程不通过asyncio.wait_for()限制等待时间，task_stars和task_dollars会以这种方式永远地交替执行下去，而锁只会在两个任务切换的一瞬间被释放，其他时间都被其中一个任务持有。超时会导致asyncio.CancelledError异常在两个任务之一中被抛出（事实上由于超时时间是1.1秒所以总是在task_dollars中被抛出），而这两个任务会截获该异常，将锁释放，然后再将该异常抛出。保证一个任务在被取消时会释放它获得的所有锁是一个好习惯。

最后需要强调，锁并不具有限制性，它与相关资源的关联其实是逻辑上的。例如在上面的例子中创建一个锁时并没有显式声明它被关联到标准输出。锁被关联到哪个或哪些资源，其实体现在获得它的任务将访问哪个或哪些资源。而锁起作用的前提是所有访问该资源的任务都主动去获得锁，只要有一个任务在访问该资源前不去试图获得锁，就会破坏该锁的有效

性。事实上，所有同步原语（包括前面讨论的屏障和事件）都具有这种特性，即使用它们的所有任务都必须主动配合才能让它们起作用。

使用锁来分配资源的访问权时，只能严格按照任务申请获得锁的顺序。因此锁无法解决下面这种场景描述的问题：一个资源被很多任务共享，但其中部分或全部任务仅在某条件被满足时才需要访问该资源，而这些条件何时被满足是无法预知的。乍一看，我们似乎可以联合使用锁和事件来解决这一问题，即让这些任务通过申请获得锁来排队，同时等待代表相关条件被满足的事件发生，但仔细思考后就会发现可能发生这样的情况：任务A获得了锁但所需条件未被满足，而任务B所需的条件被满足但没有获得锁，假如任务A所需的条件需要通过执行任务B被满足，那么任务A和B就会永远无法被执行。上述情况是“死锁（deadlock）”的一种，在本章后面会详细说明什么是死锁。

总而言之，为了解决上述问题必须使用新的同步原语——“监视器（monitor）”。一个监视器将一个条件绑定到一个已经存在的锁。锁是监视器的底层，一个锁可以被多个监视器共享，但这些监视器必须代表不同的条件。

监视器是这样被使用的：共享同一资源的任务根据各自所需满足的条件被分组，每组任务等待一个监视器，这些监视器的底层是同一个关联到该资源上的锁。一个任务等待一个监视器需要分两步实现：第一步是获得监视器底层的锁，第二个操作是等待监视器本身，而后者会释放该监视器的底层锁。这样，该组任务可以全部处于等待该监视器的状态，但它们中没有谁获得了底层锁。当监视器代表的条件被满足时，该组任务的部分或全部会按照等待监视器的顺序依次被唤醒，然后再次尝试获得底层锁，如果成功就得到执行的机会，执行完后一定要释放底层锁（可以直接释放底层锁或者再次等待监视器）。由于所有使用该底层锁的任务未被执行时都会等待某个监视器，而在这种状态不会获得底层锁，所以不会产生死锁。

那些不需要满足任何条件，只需要获得锁就可以执行的任务，同样构成了一个分组。该分组可被视为等待一个虚拟的代表条件True的监视器，该条件总是已经被满足。因此我们可以先创建一个锁，然后再创建n个以该锁为底层的监视器，进而将所有需要获得该锁的任务分为n+1组，再套用上面的模型来设计异步程序。

asyncio定义了Condition类（而不是Monitor类）来表示监视器，其实例化语法为：

```
class asyncio.Condition(lock=None)
```

其lock参数可以是一个已经存在的锁，如果是None则自动创建一个新锁。

Condition对象具有表14-9列出的属性，注意它完全覆盖了表14-8，这意味着将一个Condition对象当成一个Lock对象来使用是完全合法的。需要强调的是，Condition类提供了两种不同的方式来实现监视器。第一种方式严格按照个上面描述的方式，每个监视器代表一种条件，因此需要为一个资源创建多个监视器。第二种方式用一个监视器代表任意多个条件，只需要为一个资源创建一个监视器，使代码更简洁。

表14-9. 监视器的属性

属性	说明
`acquire()`	一个协程函数，申请获得底层锁。
`release()`	释放底层锁。
`locked()`	判断底层锁是否已被某任务获得。
`wait()`	一个协程函数，等待该监视器，释放底层的锁。必须在已经获得底层锁的情况下调用。
`wait_for()`	一个协程函数，先验证指定的条件是否被满足，仅在不满足时等待该监视器。必须在已经获得底层锁的情况下调用。
`notify()`	唤醒指定数量的等待该监视器的任务。必须在已经获得底层锁的情况下调用。
`notify_all()`	唤醒所有等待该监视器的任务。必须在已经获得底层锁的情况下调用。

如果以第一种方式实现监视器，则需要使用协程函数wait()，其语法为：

```
coroutine asyncio.Condition.wait()
```

在调用wait()前，必须先获得底层锁。一个任务等待通过调用wait()创建的协程会自动释放底层锁，然后挂起自身。被挂起的任务仅当收到来自notify()或notify_all()的通知后才会被唤醒，然后重新申请获得底层锁，并在未能获得底层锁的情况下再次被挂起，直到重新获得底层锁为止，最后执行完成时返回True。如果一个任务在未获得底层锁的情况下直接调用一个Condition对象的wait()，会导致抛出RuntimeError异常。

如果以第二种方式实现监视器，则需要使用协程函数wait_for()，其语法为：

```
coroutine asyncio.Condition.wait_for(predicate)
```

wait_for()和wait()的区别在于前者创建的协程的行为：该协程会先调用一次通过predicate参数传入的函数，并对该函数的返回值进行逻辑值检测，如果结果为True就直接执行完成并返回True，并不需要释放底层锁；否则，它释放底层锁，并挂起自身。换句话说，wait_for()并不一定会挂起任务并释放底层锁。

notify()被运行中的任务用来唤醒等待监视器的其他任务，其语法为：

```
asyncio.Condition.notify(n=1)
```

该函数被调用后，等待该监视器的前n个任务会被唤醒，但当前任务并不会因此而停止运行，而会执行到对某个await表达式求值。同样，如果一个任务在未获得底层锁的情况下调用一个Condition对象的notify()，会抛出RuntimeError异常。

notify_all()的语法为：

```
asyncio.notify_all()
```

它与notify()的唯一区别是会唤醒所有等待该监视器的任务。

需要强调的是，如果一个任务通过wait()等待监视器，则它被唤醒后必然会开始申请获得底层锁；而如果一个任务通过wait_for()等待监视器并被挂起，则它被唤醒后会再次调用通过predicate参数传入的函数，并对其返回值进行逻辑值检测，仅当结果为True时才继续申请获得底层锁，否则会直接挂起自身。

下面用两个例子说明监视器的用法，它们都综合了coroutine16.py和coroutine17.py，使得任务task_stars和task_dollars基于任务task_num_gen随机生成的整数来调度。第一个例子以第一种方式实现监视器：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


#该协程函数创建的协程以0.1秒为间隔连续显示三个“*”。    但它等待的是监视器而非锁。
async def stars(cond):
    try:
        #申请获得底层锁，进入临界区。
        await cond.acquire()
        while True:
            #等待监视器，并释放底层锁，离开临界区。
            await cond.wait()
            #在这里会重新申请获得底层锁，进入临界区。
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            print("")
    except asyncio.CancelledError:
        #在任务被取消时释放底层锁。
        cond.release()
        raise


#该协程函数创建的协程以0.1秒为间隔连续显示三个“$”。    但它等待的是监视器而非锁。
async def dollars(cond):
    try:
        #申请获得底层锁，进入临界区。
        await cond.acquire()
        while True:
            #等待监视器，并释放底层锁，离开临界区。
            await cond.wait()
            #在这里会重新申请获得底层锁，进入临界区。
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
```

```
                    sys.stdout.flush()
                print("")
        except asyncio.CancelledError:
            #在任务被取消时释放底层锁。
            cond.release()
            raise


    #该协程函数创建的协程直接通过lock获得和释放底层锁，只要获得了底层锁就会随机生成一个
    # 0~100内的整数。   如果该整数是偶数，则通知等待cond1的任务执行。   如果该整数是奇数，
    # 则通知等待cond2的任务执行。
    async def num_gen(lock, cond1, cond2):
        while True:
            #申请获得底层锁，进入临界区。
            await lock.acquire()
            n = random.randrange(100)
            if n % 2 == 0:
                #通知等待cond1的任务。
                cond1.notify()
            else:
                #通知等待cond2的任务。
                cond2.notify()
            #释放底层锁，离开临界区。
            lock.release()
            await asyncio.sleep(0)


    async def main():
        #创建一个关联到标准输出的锁。
        lock = asyncio.Lock()
        #创建一个以lock为底层锁，代表生成了偶数的监视器。
        cond1 = asyncio.Condition(lock)
        #创建一个以lock为底层锁，代表生成了奇数的监视器。
        cond2 = asyncio.Condition(lock)
        #创建三个任务并将它们聚集成一个任务。
        task_num_gen = asyncio.create_task(num_gen(lock, cond1, cond2))
        task_stars = asyncio.create_task(stars(cond1))
        task_dollars = asyncio.create_task(dollars(cond2))
        task = asyncio.gather(task_num_gen, task_stars, task_dollars)
        try:
            #运行任务2秒。
            await asyncio.wait_for(task, 2)
        except asyncio.TimeoutError:
            pass
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

该例子首先创建了一个锁lock，然后创建了两个以lock为底层锁的监视器cond1和cond2。cond1代表的条件是task_num_gen随机生成了一个偶数，使得task_stars能够输出三个“*”。cond2代表的条件是task_num_gen随机生成了一个奇数，使得task_dollars能够输出三个“$”。

请将上述代码保存为coroutine18.py，然后通过如下命令行验证：

```
$ python3 coroutine18.py
***
$$$
***
$$$
$$$
***
*

$ python3 coroutine18.py
$$$
***
***
$$$
***
***
$

...
```

在上面的例子中，stars()和dollars()在进入无限循环前会执行“await cond.acquire()”一次，这是因为没有获得底层锁就无法执行“await cond.wait()”。而它们的循环体中的语句其实都属于临界区，但却在“await cond.wait()”处离开了临界区，然后又进入了临界区，这也是这两个任务被挂起的时间点。由于wait()创建的协程必然会挂起任务并释放底层锁，所以可以简单的以它作为分界点。

另外，虽然num_gen()的执行不需要满足任何条件，但依然直接通过lock获得/释放底层锁，这是因为它需要调用notify()函数。由于在该例子中等待监视器cond1和cond2的任务都只有1个，所以可以使用notify()，且不需要传入任务数量。本程序的运行规律其实是任务task_stars和任务task_dollars两者之一与任务task_num_gen交替运行，前两者的选择是随机的。

第二个例子则以第二种方式实现监视器：

```
#!/usr/bin/env python3

import asyncio
import random
import time
import sys

#该全局变量被用于表示该监视器涉及的两种条件。    当其引用True时，代表生成了偶数。    当其
# 引用False时，代表生成了奇数。
even_number = None


async def stars(cond):
    try:
        while True:
            #申请获得底层锁，进入临界区。
            await cond.acquire()
            #仅当event_number引用False时等待监视器，并释放底层锁，离开临界区。    否
            # 则会继续执行。
            await cond.wait_for(lambda: even_number)
            #仅当event_number引用True时申请获得底层锁，进入临界区。
```

```
                for i in range(3):
                    await asyncio.sleep(0.1)
                    sys.stdout.write("*")
                    sys.stdout.flush()
                print("")
                #释放底层锁，离开临界区。
                cond.release()
        except asyncio.CancelledError:
            cond.release()
            raise


    async def dollars(cond):
        try:
            while True:
                #申请获得底层锁，进入临界区。
                await cond.acquire()
                #仅当event_number引用True时等待监视器，并释放底层锁，离开临界区。   否则
                # 会继续执行。
                await cond.wait_for(lambda: not even_number)
                #仅当event_number引用False时申请获得底层锁，进入临界区。
                for i in range(3):
                    await asyncio.sleep(0.1)
                    sys.stdout.write("$")
                    sys.stdout.flush()
                print("")
                #释放底层锁，离开临界区。
                cond.release()
        except asyncio.CancelledError:
            cond.release()
            raise


    #该协程函数创建的协程通过监视器来获得和释放底层锁，但并不等待监视器。
    async def num_gen(cond):
        #声明全局变量。
        global even_number
        while True:
            #申请获得底层锁，进入临界区。
            await cond.acquire()
            n = random.randrange(100)
            if n % 2 == 0:
                #设置even_number引用True。
                even_number = True
            else:
                #设置even_number引用False。
                even_number = False
            #通知所有等待监视器的任务条件已经发生变化。
            cond.notify_all()
            #释放底层锁，离开临界区。
            cond.release()
            await asyncio.sleep(0)


    async def main():
        #直接创建一个监视器，并将自动创建的底层锁关联到标准输出。
        cond = asyncio.Condition()
        task_num_gen = asyncio.create_task(num_gen(cond))
        task_stars = asyncio.create_task(stars(cond))
        task_dollars = asyncio.create_task(dollars(cond))
        task = asyncio.gather(task_num_gen, task_stars, task_dollars)
        try:
           await asyncio.wait_for(task, 2)
        except asyncio.TimeoutError:
            pass
```

```
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine19.py，然后通过如下命令行验证它的功能与coroutine18.py完全相同：

```
    $ python3 coroutine19.py
    ***
    ***
    ***
    $$$
    $$$
    ***
    *

    $ python3 coroutine19.py
    ***
    $$$
    $$$
    $$$
    ***
    ***
    $

    ...
```

注意上面的例子只创建了一个监视器cond，并通过它隐式创建了一个底层锁。该监视器本身并不代表任何条件，条件是通过全局变量event_number实现的：当其引用True时表示随机生成了偶数；当其引用False时表示随机生成了奇数。

在该例子中，stars()和dollars()以“await cond.wait_for(...)”做为任务挂起的分界点，但由于它并不能保证每次都挂起任务并释放底层锁，所以必须在循环体的开始显式通过“await cond.acquire()”获得底层锁，在其末尾显式通过“cond.release()”释放底层锁。而num_gen()则将监视器当成锁来使用，在随机生成了整数后，先设置event_number，再通过“cond.notify_all()”通知所有等待该监视器的任务（task_stars和task_dollars）开始执行，但其中只有一个任务需要的条件会被满足。

14-7. 同步原语——信号量

（标准库：asyncio）

上一节考虑的两种场景虽然存在被任务共享的资源，但任务之间并没有交换数据。当任务间存在数据交换时，将使同步的难度再提高一个级别。要想解决这类场景带来的问题，必须使用同步原语“信号量（semaphore）”。

信号量最初是为了解决“生产者-消费者问题（producer-consumer problem）”而被提出的，该问题是计算机科学中的一个经典问题，描述了这样一种场景：如果任务A执行某项操作的前提是任务B执行了另一项操作，则认为任务B对于任务A来说是生产者，而任务A对于

任务B来说是消费者。生产者和消费者之间通过一个队列（栈被视为LIFO队列）来交换数据：生产者将生成的数据项插入队列，消费者则从队列取出数据项。该场景中的每个队列至少要有1个生产者和1个消费者，且生产者和消费者的数量都没有上限。

很多书籍都以类似于这样的比喻来解释生产者-消费者问题描述的场景：生产者是制造寿司的厨师，消费者是品尝寿司的食客，而队列是餐盘。这个模型的关键点在生产者生成数据项的速度和消费者取出数据项的速度不需要保持一致，甚至可以随机变化。精确描述这一模型的动力学特征的数学工具是排队论，但计算机科学家只关注产生竞争条件的时刻：当队列变空时，只允许生产者访问，消费者应被挂起；当队列变满时，只允许消费者访问，生产者应被挂起。由于竞争条件只会在队列的两端被触及时产生，所以生产者-消费者问题又被称为“有界缓存问题（bounded-buffer problem）”。

不能用锁或监视器来解决生产者-消费者问题的原因是，队列在大部分情况下是可以被所有任务访问的，而并非只能被一个任务访问。解决该问题的关键是如何判断队列的两端被触及，并采取有效的措施防止产生竞争条件。信号量与锁支持的操作完全相同，不同的地方在于如下两点：

➤ 锁的acquire()操作和release()操作必须由同一个任务先后执行，两次操作之间的代码就是临界区；而信号量的acquire()操作和release()操作可以由不同的任务执行，临界区的划分也不是那么清晰。
➤ 信号量不止有两个状态，而是在内部维护着一个计数器：当计数器的值不为0时，执行acquire()操作会让计数器减1，执行release()操作会让计数器加1；当计数器的值变成0时，后续的acquire()操作会导致执行它的任务被挂起，而后续的release()操作会使这些被挂起的任务重新被执行，仅当没有任务因该信号量被挂起时才会让计数器加1。

用计算机科学的术语来说，acquire()是P操作，release()是V操作。假设init是信号量计数器的初始值，那么信号量保证在任意时刻有：

P操作的累积执行次数 ≤ V操作的累积执行次数 + init

显然，信号量的计数器的最大值应是队列的容量。然而在生产者-消费者问题中必须允许使用具有无限容量的队列，亦即无长度限制的队列。根据队列是否有长度限制，生产者-消费者问题具有两个变体：如果队列长度没有限制，那么只可能变空不可能变满，因此只在一端有产生竞争条件的可能，使用1个信号量就可以描述；如果队列长度有限制，那么在两端都有产生竞争条件的可能，必须使用2个信号量来描述。

asyncio定义了Semaphore类来表示信号量，其实例化语法为：

```
class asyncio.Semaphore(value=1)
```

其中value参数被用于设置计数器的初始值，必须传入一个非负整数。注意Semaphore类没有对计数器的最大值做任何限制。

Semaphore对象具有表14-10列出的属性，它们与Lock对象的属性名称相同，但行为并不相同。具体来说，acquire()首先会使计数器的值减1，仅当它已经降到0时才会挂起任务；release()首先会使挂起的任务按顺序被唤醒，直到没有被挂起的任务才会使计数器的值加1；locked()返回True意味着计数器值为0，返回False意味着计数器的值大于0，但无法精确知道是哪个正整数。

表14-10. 信号量的属性

属性	说明
acquire()	一个协程函数，申请获得该信号量。
release()	释放该信号量。
locked()	判断该信号量的计数器是否为0。

下面用例子说明如何通过信号量解决生产者-消费者问题。在第一个例子中队列的长度没有限制，因此只需要使用1个信号量来限制消费者：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


#该协程函数创建的协程是生产者。    它们随机睡眠0~1秒的时间，然后随机生成一个0~100内的
# 整数n，最后将元组(name，n)插入无长度限制的LIFO队列stack。
async def producer(stack, sem, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
        n = random.randrange(100)
        #生成数据项。
        stack.append((name, n))
        print(f"append ({name}, {n})")
        print(f"  stack size: {len(stack)}")
        #释放信号量，唤醒被挂起的消费者，否则使计数器加1。
        sem.release()


#该协程函数创建的协程是消费者。    它们随机睡眠0~1秒的时间，然后从LIFO队列stack中取出
# 第一个元素是name的元组。
async def comsumer(stack, sem, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
```

```
            #申请获得信号量，当计数器值为0时被挂起，否则使计数器减1。
            await sem.acquire()
            #取出数据项。
            data = stack.pop()
            print(f"{name}: pop ({data[0]}, {data[1]})")
            print(f"  stack size: {len(stack)}")


    async def main():
        #创建生产者和消费者共享的LIFO队列。
        q = list()
        #创建信号量。
        s = asyncio.Semaphore(0)
        #创建生产者任务。
        p1 = asyncio.create_task(producer(q, s, "p1"))
        p2 = asyncio.create_task(producer(q, s, "p2"))
        #创建消费者任务。
        c1 = asyncio.create_task(comsumer(q, s, "c1"))
        c2 = asyncio.create_task(comsumer(q, s, "c2"))
        c3 = asyncio.create_task(comsumer(q, s, "c3"))
        #将多个任务聚集成一个。
        task = asyncio.gather(p1, p2, c1, c2, c3)
        #task = asyncio.gather(p1, p2, c1)
        try:
            #启动任务，运行5秒。
            await asyncio.wait_for(task, 5)
        except asyncio.TimeoutError:
            #取消任务。
            task.cancel()
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine20.py，然后通过如下命令行验证：

```
$ python3 coroutine20.py
append (p1, 55)
  stack size: 1
c1: pop (p1, 55)
  stack size: 0
append (p1, 18)
  stack size: 1
c1: pop (p1, 18)
  stack size: 0
append (p1, 34)
  stack size: 1
append (p2, 36)
  stack size: 2
c2: pop (p2, 36)
  stack size: 1
append (p2, 24)
  stack size: 2
append (p2, 25)
  stack size: 3
c3: pop (p2, 25)
  stack size: 2
c1: pop (p2, 24)
  stack size: 1
c2: pop (p1, 34)
```

```
    stack size: 0
  append (p1, 74)
    stack size: 1
  c3: pop (p1, 74)
    stack size: 0
  append (p2, 65)
    stack size: 1
  c2: pop (p2, 65)
    stack size: 0
  append (p1, 1)
    stack size: 1
  c1: pop (p1, 1)
    stack size: 0
  append (p1, 84)
    stack size: 1
  c3: pop (p1, 84)
    stack size: 0
  append (p2, 83)
    stack size: 1
  c2: pop (p2, 83)
    stack size: 0
  append (p2, 54)
    stack size: 1
  c1: pop (p2, 54)
    stack size: 0
  append (p1, 63)
    stack size: 1
  c2: pop (p1, 63)
    stack size: 0
  append (p2, 51)
    stack size: 1
  c3: pop (p2, 51)
    stack size: 0
  append (p2, 1)
    stack size: 1
  c1: pop (p2, 1)
    stack size: 0
  append (p1, 32)
    stack size: 1
  c1: pop (p1, 32)
    stack size: 0
  append (p2, 55)
    stack size: 1
  c3: pop (p2, 55)
    stack size: 0
  append (p1, 70)
    stack size: 1
  c2: pop (p1, 70)
    stack size: 0
```

从该结果可以看出，由于消费者比生产者多，而每个消费者取出数据项的平均速度等于每个生产者生成数据项的平均速度，所以队列的长度大部分时间都在0和1之间震荡。但并没有哪个消费者会对空队列调用pop()，因而没有IndexError异常被抛出，这正是信号量的作用：当队列为空时，消费者被挂起，直到某个生产者生成了新的数据项。你可以将定义消费者的协程函数consumer()中的“await sem.acquire()”注释掉，然后再次验证，可以发现很快就会抛出IndexError异常。

然而如果我们对上面的例子稍做修改，将创建消费者c2和c3的语句注释掉，然后将创建task的语句改成：

```
task = asyncio.gather(p1, p2, c1)
```

会得到什么样的结果呢？请做出上述修改后再次通过如下命令行验证：

```
$ python3 coroutine20.py
append (p1, 26)
  stack size: 1
c1: pop (p1, 26)
  stack size: 0
append (p2, 85)
  stack size: 1
c1: pop (p2, 85)
  stack size: 0
append (p1, 99)
  stack size: 1
c1: pop (p1, 99)
  stack size: 0
append (p2, 87)
  stack size: 1
c1: pop (p2, 87)
  stack size: 0
append (p1, 21)
  stack size: 1
append (p2, 99)
  stack size: 2
c1: pop (p2, 99)
  stack size: 1
append (p2, 48)
  stack size: 2
append (p1, 10)
  stack size: 3
c1: pop (p1, 10)
  stack size: 2
append (p2, 8)
  stack size: 3
append (p1, 53)
  stack size: 4
append (p2, 47)
  stack size: 5
c1: pop (p2, 47)
  stack size: 4
append (p1, 2)
  stack size: 5
append (p2, 36)
  stack size: 6
c1: pop (p2, 36)
  stack size: 5
append (p1, 40)
  stack size: 6
append (p1, 14)
  stack size: 7
append (p2, 33)
  stack size: 8
```

该结果说明队列的长度会随着时间而增加。这样的程序只要运行足够长的时间就会耗尽全部物理内存，导致系统崩溃。

只有在能保证所有消费者取出数据项的总平均速度不低于所有生产者生成数据项的总平均速度时，才可以不限制队列的长度。对于复杂的异步程序来说，很难判断这一条件是否能被满足，所以一般而言都会使用一个有限长度的队列来解决生产者-消费者问题，而此时需要使用2个信号量分别限制生产者和消费者，如第二个例子所示：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


async def producer(stack, sem1, sem2, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
        n = random.randrange(100)
        #申请获得限制生产者的信号量。
        await sem1.acquire()
        stack.append((name, n))
        print(f"append ({name}, {n})")
        print(f"  stack size: {len(stack)}")
        #释放限制消费者的信号量。
        sem2.release()


async def comsumer(stack, sem1, sem2, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
        #申请获得限制消费者的信号量。
        await sem2.acquire()
        data = stack.pop()
        print(f"{name}: pop ({data[0]}, {data[1]})")
        print(f"  stack size: {len(stack)}")
        #释放限制生产者的信号量。
        sem1.release()


async def main():
    q = list()
    #创建限制生产者的信号量，其计数器的初始值是4。
    s1 = asyncio.Semaphore(4)
    #创建限制消费者的信号量，其计数器的初始值是0。
    s2 = asyncio.Semaphore(0)
    #创建生产者任务。
    p1 = asyncio.create_task(producer(q, s1, s2, "p1"))
    p2 = asyncio.create_task(producer(q, s1, s2, "p2"))
    #创建消费者任务。
    c1 = asyncio.create_task(comsumer(q, s1, s2, "c1"))
    task = asyncio.gather(p1, p2, c1)
    try:
        await asyncio.wait_for(task, 5)
    except asyncio.TimeoutError:
        task.cancel()
    sys.exit(0)


if __name__ == '__main__':
    asyncio.run(main())
```

注意限制生产者的信号量的计数器初始值将成为队列的长度限制，而队列本身在初始状态依然是空队列。请将上述代码保存为coroutine21.py，然后通过如下命令行验证：

```
$ python3 coroutine21.py
append (p1, 34)
  stack size: 1
c1: pop (p1, 34)
  stack size: 0
append (p2, 9)
  stack size: 1
c1: pop (p2, 9)
  stack size: 0
append (p2, 32)
  stack size: 1
c1: pop (p2, 32)
  stack size: 0
append (p1, 39)
  stack size: 1
append (p1, 40)
  stack size: 2
append (p1, 74)
  stack size: 3
c1: pop (p1, 74)
  stack size: 2
c1: pop (p1, 40)
  stack size: 1
append (p1, 64)
  stack size: 2
append (p2, 59)
  stack size: 3
append (p1, 9)
  stack size: 4
c1: pop (p1, 9)
  stack size: 3
append (p2, 92)
  stack size: 4
c1: pop (p2, 92)
  stack size: 3
append (p1, 99)
  stack size: 4
c1: pop (p1, 99)
  stack size: 3
append (p2, 12)
  stack size: 4
```

从该结果可以看出，由于生产者比消费者多，且它们产生/取出数据项的平均速度相同，所以队列的长度大部分时间都在其上限和上限减1之间震荡。限制消费者的信号量的作用与在前一个例子中相同，而限制生产者的信号量的作用是：当队列变满时，生产者被挂起，直到某个消费者取出了已有数据项。

至此我们已经讨论完了标准的信号量。然而asyncio还定义了BoundedSemaphore类来表示“受限信号量（bounded semaphores）”，即限制了计数器最大值的信号量。

该类的实例化语法为：

```
class asyncio.BoundedSemaphore(value=1)
```

其中value参数既被用于设置计数器的初始值，同时也是其最大值。BoundedSemaphore对象具有的属性与Semaphore对象相同，唯一的区别在于当计数器已经取最大值时，调用release()会导致抛出ValueError异常。

受限信号量显然不会被用于解决队列长度无限制的生产者-消费者问题。在解决队列长度有限制的生产者-消费者问题时，可以用受限信号量代替信号量来限制生产者，但这样做的意义不大。受限信号量主要被用于解决一些非典型的生产者-消费者问题，例如队列最初是满的，生产者产生的多余数据项可以被直接丢弃。下面给出的例子属于这样的场景：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


async def producer(stack, sem, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
        try:
            n = random.randrange(100)
            stack.append((name, n))
            #该操作可能导致抛出ValueError异常。
            sem.release()
            print(f"append ({name}, {n})")
        except ValueError:
            #如果抛出了ValueError异常，则丢弃生成的数据项。
            stack.pop()
            print(f"fail to append ({name}, {n})")
        finally:
            print(f"  stack size: {len(stack)}")


async def comsumer(stack, sem, name):
    while True:
        t = random.random()
        await asyncio.sleep(t)
        await sem.acquire()
        data = stack.pop()
        print(f"{name}: pop ({data[0]}, {data[1]})")
        print(f"  stack size: {len(stack)}")


async def main():
    #被生产者和消费者共享的FIFO队列初始就被填满。
    q = [("init", -1), ("init", -2), ("init", -3), ("init", -4)]
    #创建限制消费者的受限信号量，其初始值和最大值都是4。
    s = asyncio.BoundedSemaphore(4)
    #创建生产者任务。
    p1 = asyncio.create_task(producer(q, s, "p1"))
    p2 = asyncio.create_task(producer(q, s, "p2"))
```

```
        #创建消费者任务。
        c1 = asyncio.create_task(comsumer(q, s, "c1"))
        task = asyncio.gather(p1, p2, c1)
        try:
            await asyncio.wait_for(task, 5)
        except asyncio.TimeoutError:
            task.cancel()
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

该例子仅使用了一个受限信号量。请将上述代码保存为coroutine22.py，然后通过如下命令行验证：

```
$ python3 coroutine22.py
fail to append (p2, 77)
  stack size: 4
fail to append (p1, 95)
  stack size: 4
fail to append (p1, 53)
  stack size: 4
fail to append (p2, 98)
  stack size: 4
fail to append (p2, 10)
  stack size: 4
c1: pop (init, -4)
  stack size: 3
append (p1, 53)
  stack size: 4
fail to append (p2, 70)
  stack size: 4
fail to append (p2, 68)
  stack size: 4
fail to append (p1, 43)
  stack size: 4
c1: pop (p1, 53)
  stack size: 3
c1: pop (init, -3)
  stack size: 2
append (p2, 44)
  stack size: 3
append (p1, 8)
  stack size: 4
c1: pop (p1, 8)
  stack size: 3
append (p2, 27)
  stack size: 4
c1: pop (p2, 27)
  stack size: 3
append (p1, 29)
  stack size: 4
fail to append (p2, 82)
  stack size: 4
fail to append (p1, 29)
  stack size: 4
c1: pop (p1, 29)
  stack size: 3
c1: pop (p2, 44)
  stack size: 2
```

```
c1: pop (init, -2)
  stack size: 1
append (p2, 6)
  stack size: 2
c1: pop (p2, 6)
  stack size: 1
append (p1, 7)
  stack size: 2
c1: pop (p1, 7)
  stack size: 1
append (p2, 74)
  stack size: 2
```

该结果说明有相当多的数据项被丢弃，以保证队列的长度不超过其上限。

最后必须要强调，不论是信号量还是受限信号量，如果其初始值被设置为1，而所有任务在调用其release()之前都会先调用其acquire()，则该信号量/受限信号量就等同于一个锁。在计算机科学中，讨论同步相关问题时经常会用信号量来代替锁。

14-8. 复杂同步问题

（标准库：asyncio）

我们已经介绍完了所有的同步原语，但考虑的场景都相对简单。现实中的异步程序所面对的场景通常要比提出同步原语时考虑的场景要复杂得多，需要组合使用多个（甚至属于不同种类的）同步原语才能解决相关问题。关于复杂同步问题的深入讨论足以写一整本书，因此超出了本书的范围。本节仅通过对“读者-写者问题（readers-writers problem）”和“哲学家就餐问题（dining philosophers problem）”的讨论，展示解决此类复杂同步问题的思考过程，以及设计相应同步机制的要点。

读者-写者问题考虑的是这样一种场景：并行运行的多个任务共享同一资源，其中一部分将数据写入资源，另一部分从资源读取数据，而该资源并不能被简单地抽象成一个队列。异步程序需要达到的目标是：

- 当一个读操作正在执行时，允许其他读操作被执行，但不允许写操作被执行。
- 当一个写操作正在执行时，既不允许其他写操作被执行，也不允许读操作被执行。
- 必须保持读操作和写操作的相对次序，即如果程序中指定一个读操作在一个写操作之后被执行，那么实际执行时该读操作就不能在该写操作之前被执行，反之亦然。

上述场景并不符合任何一种同步原语被提出时考虑的场景，但依然会导致产生竞争条件。下面的例子说明了为什么该场景会产生竞争条件：

```
#!/usr/bin/env python3

import asyncio
import random
import sys
```

```
    #被读者和写者共享的资源。
    resource = ['*', '$', '#']


    #该协程函数创建的协程为写者。   它们会用data中的元素按顺序依次替换resource中的三个元素，
    # 每次替换前随机等待0~1秒，替换后显示操作成功相关信息。
    async def writer(data, name):
        global resource
        for i in range(3):
            t = random.random()
            await asyncio.sleep(t)
            resource[i] = data[i]
            print(f"{name}: write {data[i]}, now {resource[0] + resource[1] +
resource[2]}")


    #该协程函数创建的协程为读者。   它们会按顺序依次读取resource中的三个元素，每次读取前随机
    # 等待0~1秒，读取后显示操作成功相关信息。
    async def reader(name):
        global resource
        for i in range(3):
            t = random.random()
            await asyncio.sleep(t)
            print(f"{name}: read {resource[i]}")


    async def main():
        #定义4个读者和2个写者，按照r1，w1，r2，r3，w2，r4的顺序执行读写操作。
        r1 = asyncio.create_task(reader("r1"))
        w1 = asyncio.create_task(writer("abc", "w1"))
        r2 = asyncio.create_task(reader("r2"))
        r3 = asyncio.create_task(reader("r3"))
        w2 = asyncio.create_task(writer("def", "w2"))
        r4 = asyncio.create_task(reader("r4"))
        await asyncio.gather(r1, w1, r2, r3, w2, r4)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

注意在该例子中，读者执行的读操作和写者执行的写操作都分别由3个子原子操作构成，因此其自身不是原子操作。而按照读者-写者问题指定的目标，我们期望上述代码的行为是：

- 步骤一：读者r1依次读取“*”、“$”和“#”，即资源的初始内容。
- 步骤二：写者w1依次写入“a”、“b”和“c”，使资源的内容变成“abc”。
- 步骤三：读者r2和r3分别依次读取“a”、“b”和“c”，但它们各自的子原子操作可以交错。
- 步骤四：写者w2依次写入“d”、“e”和“f”，使资源的内容变成“def”。
- 步骤五：读者r4依次读取“d”、“e”和“f”。

然而上述代码的真实行为是怎样的呢？请将它保存为coroutine23.py，然后通过如下命令行验证：

```
$ python3 coroutine23.py
r1: read *
w2: write d, now d$#
r1: read $
w1: write a, now a$#
r3: read a
w1: write b, now ab#
r2: read a
r4: read a
w1: write c, now abc
w2: write e, now aec
r3: read e
r1: read c
r4: read e
w2: write f, now aef
r2: read e
r3: read f
r4: read f
r2: read f

$ python3 coroutine23.py
w2: write d, now d$#
r3: read d
r1: read d
w2: write e, now de#
r3: read e
r4: read d
w1: write a, now ae#
r1: read e
r2: read a
r4: read e
w2: write f, now aef
r4: read f
w1: write b, now abf
r2: read b
r3: read f
r1: read f
w1: write c, now abc
r2: read c

...
```

可见这4个读者和2个写者的操作毫无规律地混杂在一起，结果完全无法预测。这说明这些读者和写者运行的整个过程都是竞争条件。

为了解决这个问题，我们首先要判断应该使用哪些同步原语。显然该场景并不需要同步读者和写者的进度，因此不能使用屏障。使用事件可以么？比如让所有读者等待一个代表“可读”的事件发生，而所有写者等待一个代表“可写”的事件发生？这样设计的问题是，虽然可以让读操作和写操作分别在两个事件上排队，却无法记录它们相互间的相对次序。锁、监视器和信号量中，后两者都可以被当成前者使用；如果所有读操作和写操作都在某个同步原语上完成了排队，那就没必要再额外使用监视器的通知机制；而虽然读者取出的是写者写入的内容，但是一次性取出所有内容，因此没必要使用信号量的计数机制。综上所述，解决读者-写者问题只需要使用锁。

接下来开始设计同步机制。最符合直觉的设计是只使用一个“资源锁（resource lock）”来限制对共享资源的访问。当该锁被一个写者获得时，读者和其余写者都必须排队，直到该写者释放锁。当该锁被一个读者获得时，只有写者需要排队，其余读者不再需要获得该锁就可以访问共享资源，但需要用一个计数器来记录当前读者的数量，并让最后一个读者释放锁。

基于此思路设计出的方案如下面的代码所示：

```
#!/usr/bin/env python3

import asyncio
import random
import sys

resource = ['*', '$', '#']
#资源锁，在进行读操作和写操作期间都需要一直持有该锁。
resource_lock = asyncio.Lock()
#记录当前读者数量的计数器。
read_count = 0


async def writer(data, name):
    global resource, resource_lock
    #申请获得资源锁。
    await resource_lock.acquire()
    for i in range(3):
        t = random.random()
        await asyncio.sleep(t)
        resource[i] = data[i]
        print(f"{name}: write {data[i]}, now {resource[0] + resource[1] +
resource[2]}")
    #释放资源锁。
    resource_lock.release()


async def reader(name):
    global resource, resource_lock, read_count
    #读者的数量加1。
    read_count = read_count + 1
    #如果该读者是第一个，则申请获得资源锁。
    if read_count == 1:
        await resource_lock.acquire()
    for i in range(3):
        t = random.random()
        await asyncio.sleep(t)
        print(f"{name}: read {resource[i]}")
    #读者的数量减1。
    read_count = read_count - 1
    #如果该读者是最后一个，则释放资源锁。
    if read_count == 0:
        resource_lock.release()


async def main():
    r1 = asyncio.create_task(reader("r1"))
    w1 = asyncio.create_task(writer("abc", "w1"))
```

```
        r2 = asyncio.create_task(reader("r2"))
        r3 = asyncio.create_task(reader("r3"))
        w2 = asyncio.create_task(writer("def", "w2"))
        r4 = asyncio.create_task(reader("r4"))
        await asyncio.gather(r1, w1, r2, r3, w2, r4)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

该方案能够解决读者-写者问题么？请将上述代码保存为coroutine24.py，然后通过如下命令行验证：

```
    $ python3 coroutine24.py
    r3: read *
    r4: read *
    r3: read $
    r4: read $
    r2: read *
    r1: read *
    r3: read #
    r4: read #
    r2: read $
    r1: read $
    r2: read #
    r1: read #
    w1: write a, now a$#
    w1: write b, now ab#
    w1: write c, now abc
    w2: write d, now dbc
    w2: write e, now dec
    w2: write f, now def
```

该结果说明，这个方案仅实现了读者-写者问题的前2个目标。这是因为只要某个读者获得了资源锁，在锁被释放前新加入的读者立刻获得该资源的访问权，即便它们在写者之后也会插队到写者前面。由于这个特点，该方案被称为“读者优先（readers-preference）”，仅适用于读操作比较稀疏的场景。在读操作比较密集的情况下，写操作将一直得不到机会执行，这一现象在计算机科学中被称为“饥饿（starvation）”。虽然读者优先没有解决读者-写者问题，但在读操作非常紧急需要允许读者插队的场景下可应用该方案。

下面对读者优先方案进行改进。我们想到，为了避免读者插队到写者前面，可以增加一个“写保护锁（write protect lock）”，新加入的写者先获得该锁，使后续的读者无法再申请获得资源锁。

基于这一思路的方案如下面的代码所示：

```
#!/usr/bin/env python3

import asyncio
import random
import sys

resource = ['*', '$', '#']
resource_lock = asyncio.Lock()
#写保护锁，在执行写操作期间需要一直持有该锁。    而在执行读操作时先要获得该锁，在获得
# 资源锁后释放。
write_lock = asyncio.Lock()
read_count = 0
#记录当前写者数量的计数器。
write_count = 0


async def writer(data, name):
    global resource, resource_lock, write_lock, write_count
    #写者的数量加1。
    write_count = write_count + 1
    #如果该写者是第一个，则申请获得写保护锁。
    if write_count == 1:
        await write_lock.acquire()
    await resource_lock.acquire()
    for i in range(3):
        t = random.random()
        await asyncio.sleep(t)
        resource[i] = data[i]
        print(f"{name}: write {data[i]}, now {resource[0] + resource[1] +
resource[2]}")
    resource_lock.release()
    #写者的数量减1。
    write_count = write_count - 1
    #如果该写者是最后一个，则释放写保护锁。
    if write_count == 0:
        write_lock.release()


async def reader(name):
    global resource, resource_lock, read_count, write_lock
    #申请获得写保护锁。
    await write_lock.acquire()
    read_count = read_count + 1
    if read_count == 1:
        await resource_lock.acquire()
    #释放写保护锁。
    write_lock.release()
    for i in range(3):
        t = random.random()
        await asyncio.sleep(t)
        print(f"{name}: read {resource[i]}")
    read_count = read_count - 1
    if read_count == 0:
        resource_lock.release()


async def main():
    r1 = asyncio.create_task(reader("r1"))
    w1 = asyncio.create_task(writer("abc", "w1"))
    r2 = asyncio.create_task(reader("r2"))
    r3 = asyncio.create_task(reader("r3"))
    w2 = asyncio.create_task(writer("def", "w2"))
    r4 = asyncio.create_task(reader("r4"))
```

```
        await asyncio.gather(r1, w1, r2, r3, w2, r4)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine25.py，然后通过如下命令行验证：

```
$ python3 coroutine25.py
r1: read *
r1: read $
r1: read #
w1: write a, now a$#
w1: write b, now ab#
w1: write c, now abc
w2: write d, now dbc
w2: write e, now dec
w2: write f, now def
r4: read d
r2: read d
r3: read d
r2: read e
r4: read e
r4: read f
r3: read e
r2: read f
r3: read f
```

我们遗憾地发现，该方案依然未能解决读者-写者问题，因为只要某个写者获得了写保护锁，在锁被释放前新加入的写者即便在读者之后也会插队到该读者前面。该方案被称为“写者优先（writers-preference）”，有可能造成读者饥饿，因此仅适用于写操作比较稀疏的场景，并在实践中被应用于写操作非常紧急需要允许写者插队的场景。

仔细分析了读者优先方案和写者优先方案后，我们发现两者的问题都是让读者和写者分别排队，而想要解决读者-写者问题必须让它们统一排队。因此我们用一个“排队锁（queue lock）”取代写保护锁，并不再对写者计数：读者和写者在试图进入临界区之前都需要先获得排队锁，然后在获得了资源锁后将其释放。

基于这一思路设计的方案如下面的代码所示：

```
#!/usr/bin/env python3

import asyncio
import random
import sys

resource = ['*', '$', '#']
resource_lock = asyncio.Lock()
```

```
    #排队锁，不论执行读操作还是写操作都要先获得该锁，在获得资源锁后释放。
    queue_lock = asyncio.Lock()
    read_count = 0


    async def writer(data, name):
        global resource, resource_lock, queue_lock
        #申请获得排队锁。
        await queue_lock.acquire()
        await resource_lock.acquire()
        #释放排队锁。
        queue_lock.release()
        for i in range(3):
            t = random.random()
            await asyncio.sleep(t)
            resource[i] = data[i]
            print(f"{name}: write {data[i]}, now {resource[0] + resource[1] +
resource[2]}")
        resource_lock.release()


    async def reader(name):
        global resource, resource_lock, read_count, queue_lock
        #申请获得排队锁。
        await queue_lock.acquire()
        read_count = read_count + 1
        if read_count == 1:
            await resource_lock.acquire()
        #释放排队锁。
        queue_lock.release()
        for i in range(3):
            t = random.random()
            await asyncio.sleep(t)
            print(f"{name}: read {resource[i]}")
        read_count = read_count - 1
        if read_count == 0:
            resource_lock.release()


    async def main():
        r1 = asyncio.create_task(reader("r1"))
        w1 = asyncio.create_task(writer("abc", "w1"))
        r2 = asyncio.create_task(reader("r2"))
        r3 = asyncio.create_task(reader("r3"))
        w2 = asyncio.create_task(writer("def", "w2"))
        r4 = asyncio.create_task(reader("r4"))
        await asyncio.gather(r1, w1, r2, r3, w2, r4)
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine26.py，然后通过如下命令行验证：

```
    $ python3 coroutine26.py
    r1: read *
    r1: read $
    r1: read #
    w1: write a, now a$#
    w1: write b, now ab#
```

```
w1: write c, now abc
r3: read a
r2: read a
r3: read b
r2: read b
r3: read c
r2: read c
w2: write d, now dbc
w2: write e, now dec
w2: write f, now def
r4: read d
r4: read e
r4: read f
```

该结果正是我们所期待的，说明这才是解决读者-写者问题的正确方案。

最后必须要强调的是，上面的三个方案都利用了在单线程情况下协程函数中两次await表达式求值之间的所有语句可被视为一次原子操作的事实。

如果在多线程或多进程的情况下，类似下面的语句是有问题的：

```
read_count = read_count + 1
if read_count == 1:
    await resource_lock.acquire()
```

因为这其实包含三个原子操作：read_count加1、判断read_count是否等于1、申请获得资源锁。当有多个读者时，有可能出现这样的情况：两个读者先后执行read_count加1，再判断read_count是否等于1，由于此时read_count的值从0直接变成了2，所以申请获得资源锁的操作会被跳过。易知，这就是竞争条件。

事实上，除了全局变量resource之外，全局变量read_count和write_count同样是被读者和写者共享的资源。当涉及它们的操作不是一个原子操作时，依然需要通过锁来进行同步。这就是为什么在关于操作系统的书籍中给出的关于读者-写者问题的方案会更复杂一点，需要增加一个锁来保护read_count（习惯上写作“rmutex”），以及增加一个锁来保护write_count（习惯上写作“wmutex”），因此上面的语句需要被修改成：

```
await rmutex.acquire()
read_count = read_count + 1
if read_count == 1:
    await resource_lock.acquire()
rmutex.release()
```

由于本书不讨论Python的多线程/多进程编程，所以不给出具体例子了。

从上面对读者-写者问题的讨论可以看出，当用同步原语来解决复杂同步问题时，通常没有直觉告诉我们的那样简单。而哲学家就餐问题则说明了另一个设计重点——必须要时刻提防产生“死锁（deadlocks）”。

哲学家就餐问题最初是以如下形式被提出的：五位哲学家围坐在一张圆桌旁，每个人面前摆放着一盘意大利面，而每两盘意大利面之间摆放着一把餐叉。每位哲学家都会思考一段随机长度的时间，然后感到饥饿。当哲学家感到饥饿时，会先拿起身边的一把餐叉，然后拿起身边的另一把餐叉，吃意大利面，吃饱后同时放下两把餐叉。请设计一种算法，保证每个哲学家都不会永久地陷入饥饿。

哲学家就餐问题的本质是如何避免死锁。请注意如下事实：当一个哲学家成功拿到了两把餐叉开始吃意大利面时，他右边的哲学家将拿不到左手的餐叉，而他左边的哲学家将拿不到右手的餐叉，因此即便饥饿也只能在持有一把餐叉的情况下等待。那么就有可能存在这样一种情况：所有五位哲学家都先拿起了左手（或右手）的餐叉，然后他们中任何一个都无法拿到右手（或左手）的餐叉，导致他们将永远在饥饿中等待下去。

事实上，哲学家就餐问题中的哲学家代表的是任务（或线程、进程以及其他并行机制，下同），意大利面代表的是临界区，餐叉代表的是锁。产生死锁的原因是两个以上的协程需要获得两把以上的锁才能进入临界区。具体来说，当这些协程每个都持有部分锁，而无法获得剩余锁时，就产生了死锁。

下面的例子对哲学家就餐问题进行简化，只考虑两个任务foo()和bar()：

```
#!/usr/bin/env python3

import asyncio
import random
import sys

lock1 = asyncio.Lock()
lock2 = asyncio.Lock()


#该协程函数创建的协程显示“foo”之前必须同时获得lock1和lock2。
async def foo():
    #先申请获得lock1。
    await lock1.acquire()
    #挂起一瞬间。
    await asyncio.sleep(0)
    #再申请获得lock2。
    await lock2.acquire()
    print("foo")
    lock2.release()
    lock1.release()


#该协程函数创建的协程显示“bar”之前必须同时获得lock2和lock1。
async def bar():
    #先申请获得lock2。
    await lock2.acquire()
```

```
        #挂起一瞬间。
        await asyncio.sleep(0)
        #再申请获得lock1。
        await lock1.acquire()
        print("bar")
        lock1.release()
        lock2.release()


    async def main():
        print("begin")
        await asyncio.gather(foo(), bar())
        print("end")
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

请将上述代码保存为coroutine27.py，然后通过如下命令行验证：

```
$ python3 coroutine27.py
begin
^C
```

该结果说明该脚本陷入了死锁，只能通过按下Ctrl+C来强制终止。

下面分析coroutine27.py为什么会陷入死锁。任务foo()先于bar()运行，因此“await lock1.acquire()”最先执行，并立刻返回，使foo()成功取得lock1。然后，foo()会被挂起一瞬间，使bar()运行，“await lock2.acquire()”立刻返回，使bar()成功取得lock2。bar()同样会被挂起一瞬间，而foo()继续执行“await lock2.acquire()”，但由于lock2已经被bar()获得，所以foo()被挂起。bar()继续执行“await lock1.acquire()”，但由于lock1已经被foo()获得，所以bar()也被挂起。这样foo()和bar()分别持有lock1和lock2，都无法获得对方已经持有的锁，进而陷入了死锁。

那么，怎么修改coroutine27.py才能避免发生死锁呢？只需交换bar()获得两个锁的顺序，让它先获得lock1，再获得lock2；或者交换foo()获得两个锁的顺序，让它先获得lock2，再获得lock1。请做这样的修改，再次验证：

```
$ python3 coroutine27.py
begin
foo
bar
end
```

该结果说明修改后的脚本没有陷入死锁。

事实上，避免死锁并不困难，只需严格遵守这样的规则：如果有多个任务需要同时获得N个锁中的至少两个，那么让它们申请获得这些锁时都按照同一顺序：lock1，lock2，……，lockN。举例来说，成功解决读者-写者问题的方案中，每个写者和第一个读者都需要同时获

得queue_lock和resource_lock，而不论在writer()还是reader()中，都先申请获得前者再申请获得后者，这样就避免了死锁。写者优先方案中也采用了类似的处理方式。

具体到哲学家就餐问题，只需要将5把餐叉顺时针（或逆时针）编号1～5，然后规定每个哲学家必须先拿起身边的两把餐叉中数字较小者，就可以避免死锁。这是因为该规则相当于要求每个哲学家都按照1，2，……，5的顺序拿起餐叉，而只有坐在1和5之间的哲学家会先拿起左手（或右手）边的餐叉，其余哲学家都会先拿起右手（或左手）边的餐叉，因此不可能进入死锁状态。

14-9. 用异步队列解决生产者-消费者问题

（标准库：asyncio）

至此我们讨论完了异步编程的核心部分，理论上可以编写出具有任何可实现功能的异步程序了。但asyncio模块还提供了一些用于提高开发效率的高级API，下面用3节来介绍它们。

14-7节给出的例子用列表实现队列，并配合信号量解决了生产者-消费者问题。其实我们可以不显式使用信号量，而使用asyncio模块定义的如下三种类来解决该问题：

➤ asyncio.Queue：先进先出（FIFO）队列，消费者取出的数据项是所有现存数据项中最先被生产者插入的。
➤ asyncio.LifoQueue：后进先出（LIFO）队列，亦即栈，消费者取出的数据项是所有现存数据项中最后被生产者插入的。
➤ asyncio.PriorityQueue：优先级队列，每个数据项都有一个代表“优先级（priority）”的数字，消费者取出的数据项是所有现存数据项中优先级数字最小的。

这三种类适用于不同的场景，但它们提供的接口是相同的（亦即属性名保持一致，但行为上有差别），被总结在表14-11中。而它们的实例化语法也是相同的，即：

```
class asyncio.Queue(maxsize=0)
class asyncio.LifoQueue(maxsize=0)
class asyncio.PriorityQueue(maxsize=0)
```

表14-11. 三种异步队列的属性

属性	说明
`maxsize`	引用一个整数，队列的容量，即最多能容纳多少个数据项。
`qsize()`	取得队列中当前元素的数量，即已经存放了多少个数据项。
`empty()`	判断队列是否是空的。
`full()`	判断队列是否是满的。
`get()`	一个协程函数，以异步方式取出一个数据项。如果队列是空的，则等待到它非空。

属性	说明
put()	一个协程函数，以异步方式插入一个数据项。如果队列是满的，则等待到它不满。
get_nowait()	以同步方式取出一个数据项。如果队列是空的，则抛出asyncio.QueueEmpty异常。
put_nowait()	以同步方式插入一个数据项。如果队列是满的，则抛出asyncio.QueueFull异常。
join()	一个协程函数，等待到队列中的所有数据项都被取出。
task_done()	表明取出的数据项已使用完毕。如果队列是空的，则抛出ValueError异常。

队列的maxsize属性引用一个整数，即该队列被创建时maxsize参数对应的数字。如果它引用的是一个正整数n，则代表该队列至多存放n个数据项，用于解决队列长度有限制的生产者-消费者问题。如果它引用的是0或一个负整数，则代表该队列的容量没有上限，用于解决队列长度没有限制的生产者-消费者问题。

qsize()的语法为：

```
asyncio.{Queue|LiFoQueue|PriorityQueue}.qsize()
```

它返回一个非负整数，代表当前该队列中有多少个数据项。

empty()的语法为：

```
asyncio.{Queue|LiFoQueue|PriorityQueue}.empty()
```

当队列中的数据项数量为0时返回True，意味着该队列是空的；否则返回False。

full()的语法为：

```
asyncio.{Queue|LiFoQueue|PriorityQueue}.full()
```

当队列中的数据项数量等于maxsize属性引用的正整数时返回True，意味着该队列是满的；否则返回False。注意对于一个容量没有上限的队列来说，该函数总是返回False。

一般而言，我们会以异步的方式读写asyncio模块提供的队列，也就是通过get()和put()访问它们。这两个属性都是协程函数。get()的语法为：

```
coroutine asyncio.{Queue|LiFoQueue|PriorityQueue}.get()
```

当等待一个通过调用get()创建的协程时，如果队列非空，则该协程立刻执行完成，并返回取出的数据项（对于优先级队列来说是(priority, item)形式的元组）；如果队列是空的，则该协程会挂起，直到队列变得非空。

put()的语法为：

```
coroutine asyncio.{Queue|LiFoQueue}.put(item)
coroutine asyncio.PriorityQueue.put(priority, item)
```

当等待一个通过调用put()创建的协程时，如果队列未满，则该协程立刻执行完成，并返回None，而数据项被插入队列；如果队列是满的，则该协程会挂起，直到队列变得未满。

注意上面两个协程函数创建的协程都会在内部访问被三种异步队列封装的信号量。

asyncio模块提供的队列也能够以同步的方式读写，即通过get_nowait()和put_nowait()访问它们。它们都是函数。get_nowait()的语法为：

```
asyncio.{Queue|LiFoQueue|PriorityQueue}.get_nowait()
```

如果队列非空则返回取出的数据项；否则抛出asyncio.QueueEmpty异常。

put_nowait()的语法为：

```
asyncio.{Queue|LiFoQueue}.put_nowait(item)
asyncio.PriorityQueue.put_nowait((priority, item))
```

如果队列未满则插入数据项；否则抛出asyncio.QueueFull异常。

task_done()和join()总是配合使用的。join()是一个协程函数，其语法为：

```
coroutine asyncio.{Queue|LiFoQueue|PriorityQueue}.join()
```

当等待通过调用join()创建的协程时，如果队列是空的，则该协程立刻执行完成，并返回None；否则该协程会被挂起，直到队列变空。join()创建的协程并不会自动调用empty()或qsize()来判断队列是否为空，而是在被创建时用队列封装的信号量的计数器记录下当时队列中数据项的个数，并在计数器的值变为0时恢复执行。

但要注意的是，异步队列每被插入一个数据项其封装的信号量的计数器就加1，但数据项被取出时该计数器不会自动减1。task_done()的作用就是让该计数器减1，其语法为：

asyncio.{Queue|LiFoQueue|PriorityQueue}.task_done()

如果调用task_done()的次数比调用get()的次数与调用get_nowait()的次数之和还要多，就会抛出ValueError异常。

下面用三个例子来说明异步队列如何使用。第一个例子以同步的方式使用三种队列，以说明它们的区别，以及maxsize、qsize()、empty()、full()、get_nowait()和put_nowait()的用法：

```
#!/usr/bin/env python3

import asyncio
import sys

#该函数创建指定类型的异步队列，然后以同步方式用指定的数据将其填满，最后再以同步方式取
#出所有数据。   队列的状态变化会被显示。
def dynamic_queue(data, priority, quetype="Queue"):
    #创建指定类型的异步队列，容量都是3。
    if quetype == "PriorityQueue":
        q = asyncio.PriorityQueue(3)
        print("\nq is a priority queue:")
    elif quetype == "LifoQueue":
        q = asyncio.LifoQueue(3)
        print("\nq is a LIFO queue (stack):")
    else:
        q = asyncio.Queue(3)
        print("\nq is a FIFO queue:")
    #显示队列的初始状态:
    print(f"q.maxsize == {q.maxsize}")
    print(f"q.empty() == {q.empty()}")
    #用指定的数据填满队列:
    n = 0
    while True:
        try:
            #对于FIFO队列和LIfO队列来说，数据项是一个元组。   对于优先级队列来说，
            # 元组的第一个元素是优先级，第二个元素才是数据项。
            q.put_nowait((priority[n], data[n]))
            print(f"q.qsize() == {q.qsize()}")
            n = n + 1
       #抛出asyncio.QueueFull异常说明队列已满。
       except asyncio.QueueFull:
            print(f"q.full() == {q.full()}")
            break
    #取出所有数据:
    while True:
        try:
            d = q.get_nowait()
            print(d)
            print(f"q.qsize() == {q.qsize()}")
        #抛出asyncio.QueueEmpty异常说明队列已空。
        except asyncio.QueueEmpty:
            print(f"q.empty() == {q.empty()}")
            break


def main():
```

```
    #准备好数据和优先级。
    data = 'abcd'
    priority = (20, 30, 10, 40)
    print(f'Data is "{data}".')
    print(f"Priorities is {priority}.")
    #验证FIFO队列的性质。
    dynamic_queue(data, priority)
    #验证LIFO队列的性质。
    dynamic_queue(data, priority, "LifoQueue")
    #验证优先级队列的性质。
    dynamic_queue(data, priority, "PriorityQueue")
    return 0


if __name__ == '__main__':
    sys.exit(main())
```

请将上述代码保存为coroutine28.py，然后通过如下命令行验证：

```
$ python3 coroutine28.py
Data is "abcd".
Priorities is (20, 30, 10, 40).

q is a FIFO queue:
q.maxsize == 3
q.empty() == True
q.qsize() == 1
q.qsize() == 2
q.qsize() == 3
q.full() == True
(20, 'a')
q.qsize() == 2
(30, 'b')
q.qsize() == 1
(10, 'c')
q.qsize() == 0
q.empty() == True

q is a LIFO queue (stack):
q.maxsize == 3
q.empty() == True
q.qsize() == 1
q.qsize() == 2
q.qsize() == 3
q.full() == True
(10, 'c')
q.qsize() == 2
(30, 'b')
q.qsize() == 1
(20, 'a')
q.qsize() == 0
q.empty() == True

q is a priority queue:
q.maxsize == 3
q.empty() == True
q.qsize() == 1
q.qsize() == 2
q.qsize() == 3
q.full() == True
(10, 'c')
q.qsize() == 2
```

```
(20, 'a')
q.qsize() == 1
(30, 'b')
q.qsize() == 0
q.empty() == True
```

第二个例子实现了这样一个模型：只有一个生产者和一个消费者，而数据交换的速度由生产者控制。在该例子中生产者生成数据的速度低于消费者，因此当队列变空时消费者被迫等待：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


#该协程函数创建的协程是生产者。    它们每隔0.5秒随机生成一个0~100内的整数，并将其插入
# 异步队列。
async def producer(queue):
    while True:
        n = random.randrange(100)
        await queue.put(n)
        await asyncio.sleep(0.5)


#该协程函数创建的协程是消费者。    它们不停地从异步队列取出整数，并检查其是否是7的倍数，
# 如果是则抛出记录有尝试次数消息的Exception异常。
async def consumer(queue):
    trial = 0
    while True:
        trial = trial + 1
        n = await queue.get()
        sys.stdout.write(str(n) + " ")
        sys.stdout.flush()
        if n % 7 == 0:
            raise Exception(f"Found after {trial} trials!")


async def main():
    #创建生产者和消费者共享的异步队列。
    q = asyncio.Queue()
    #创建生产者任务。
    p_task = asyncio.create_task(producer(q))
    #创建消费者任务。
    c_task = asyncio.create_task(consumer(q))
    try:
        #启动任务。
        await asyncio.gather(p_task, c_task)
    except Exception as e:
        #取消任务。
        p_task.cancel()
        c_task.cancel()
        #显示异常携带的消息。
        print("\n" + e.args[0])
    sys.exit(0)


if __name__ == '__main__':
```

```
    asyncio.run(main())
```

请将上述代码保存为coroutine29.py，然后通过如下命令行验证：

```
$ python3 coroutine29.py
16 22 93 80 0
Found after 5 trials!

$ python3 coroutine29.py
48 24 34 77
Found after 4 trials!

...
```

注意该例子还说明了get()和put()的用法。此外，由于该例子中队列的数据项的数量不可能多于1个，所以没必要限制其容量。

第三个例子是对第二个例子的修改，由消费者控制数据交换的速度。在该例子中生产者生成数据的速度高于消费者，因此当队列变满时生产者被迫等待：

```
#!/usr/bin/env python3

import asyncio
import random
import sys


#该协程函数创建的协程是生产者。    它们不停地随机生成一个0~100内的整数，并将其插入
# 异步队列。    当队列变满时，它会挂起自身直到队列被清空。
async def producer(queue):
    while True:
        try:
            n = random.randrange(100)
            queue.put_nowait(n)
        except asyncio.QueueFull:
            await queue.join()


#该协程函数创建的协程是消费者。    它们每隔0.5秒从异步队列取出整数，并检查其是否是7的
# 倍数，如果是则抛出记录有尝试次数消息的Exception异常。
async def comsumer(queue):
    trial = 0
    while True:
        trial = trial + 1
        n = await queue.get()
        #取出数据项后一定要通知异步队列。
        queue.task_done()
        sys.stdout.write(str(n) + " ")
        sys.stdout.flush()
        if n % 7 == 0:
            raise Exception(f"Found after {trial} trials!")
        await asyncio.sleep(0.5)
```

```
async def main():
    #创建生产者和消费者共享的异步队列，其容量是5。
    q = asyncio.LifoQueue(5)
    p_task = asyncio.create_task(producer(q))
    c_task = asyncio.create_task(comsumer(q))
    try:
        await asyncio.gather(p_task, c_task)
    except Exception as e:
        p_task.cancel()
        c_task.cancel()
        print("\n" + e.args[0])
    sys.exit(0)


if __name__ == '__main__':
    asyncio.run(main())
```

请将上述代码保存为coroutine30.py，然后验证它与coroutine29.py的功能完全相同。需要注意的是，该例子为了说明join()和task_done()的用法，特意用put_nowait()来插入数据。事实上如果直接使用put()插入数据的话，就不需要使用join()和task_done()，但两种实现方式是有区别的：前者会使队列一直是满的，后者会使队列重复被填满和清空。此外，该例子使用了栈代替队列，所以消费者取出整数的顺序与它们被生成的顺序是反过来的。

14-10. 用异步流进行网络I/O和IPC

（标准库：asyncio）

第7章讨论了如何使用流进行I/O操作，然而它们都属于同步操作，只能被用于访问本地文件。基于Internet或计算机网络的通信涉及多台主机，延时和不确定性都大幅度增加，所以必须通过异步操作来进行网络I/O。这意味着，虽然网络设备也被抽象为了流，但却不能用以IOBase为基类的类来表示。

另外，同一台主机上的两个进程之间的数据交换被称为“进程间通信（inter process communications，IPC）”。虽然IPC的延时比网络I/O低得多，接近于普通的I/O操作，但其不确定性与网络I/O是一样的，因此同样需要通过异步操作来实现。事实上，网络I/O可以被视为一种高延时的IPC。

为了便于实现进行网络I/O和IPC所需的异步操作，asyncio模块的低层级API提供了“传输（transports）”和“协议（protocols）”来抽象这种异步流。传输侧重于描述异步流中的数据交换过程，协议则侧重于描述异步流中的数据交换规则，因此总是联合使用。换句话说，一个异步流可以用一个二元组（transport, protocol）来描述，其中transport代表传输，protocol代表协议。

与IOBase派生了很多类来表示不同的流类似，用于表示传输的类形成了如图14-1所示的层级结构，其中：

- BaseTransport：所有其他传输类的基类，不对应具体的异步流。
- ReadTransport：只读传输，对应只读的单向管道。
- WriteTransport：只写传输，对应只写的单向管道。

- Transport：双向传输，对应TCP连接或Unix套接字连接。
- DatagramTransport：双向传输，对应UDP连接。
- SubprocessTransport：双向传输，对应双向管道。

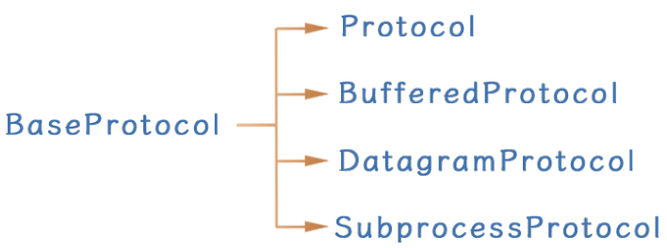

图14-1. 传输的类型

而用于表示协议的类则形成了如图14-2所示的层级结构，其中：

- BaseProtocol：所有其他协议的基类，不对应具体的异步流。
- Protocol：不使用缓冲区的流式协议，对应无缓冲区的TCP连接或Unix套接字连接。
- BufferedProtocol：使用缓冲区的流式协议，对应有缓冲区的TCP连接或Unix套接字连接。
- DatagramProtocol：数据报协议，对应UDP连接。
- SubprocessProtocol：某种IPC协议，对应管道。

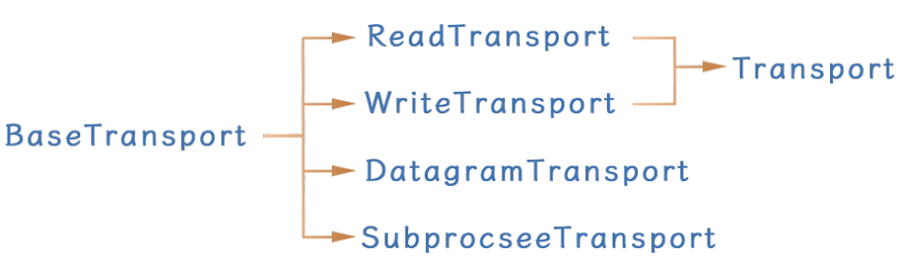

图14-2. 协议的类型

asyncio模块的高层API提供了StreamReader类和StreamWriter类来实现对异步流的读写操作。StreamReader对象是读取器，StreamWriter对象是写入器，它们总是成对出现，并被绑定到同一个(transport, protocol)元组上。由于它们可被绑定到任意传输和协议的组合，可以操作任何异步流，所以当我们使用asyncio高层API时不需要关注正在操作的异步流的种类，这甚至比第7章介绍的操作流的API更方便。

表14-12列出了StreamReader类的属性，注意其中的读操作都是异步的，因此不会造成阻塞。下面依次讨论这些属性。

表14-12. StreamReader的属性

属性	说明
`read()`	一个协程函数，从异步流中读取并返回至多指定数量的字节。

属性	说明
readline()	一个协程函数，从异步流中读取一行并返回。
readexactly()	一个协程函数，从异步流中精确读取指定数量的字节。
readuntil()	一个协程函数，从异步流中持续读取数据，直到遇到指定的分隔符。
at_eof()	判断是否已经从异步流中读到了EOF。

在讨论表14-12列出的属性之前，必须再次强调：流与异步流的本质区别在于，前者的读写操作都只涉及一个进程，因此能以同步方式完成；而后者的读写操作涉及两个进程（对于网络I/O来说分别位于两台主机上，对于IPC来说位于一台主机上），因此只能以异步的方式完成。此外，异步流处理的永远是字节，不需要考虑将其转换为字符，而不论异步流本身是否提供了缓存区，StreamReader对象都提供了自己的缓冲区。

read()的语法为：

```
coroutine asyncio.StreamReader.read(size=-1)
```

请将其与RawIOBase的read()比较，你会发现它会创建一个协程而非返回读取到的字节串。换句话说，StreamReader的read()是一个协程函数，其创建的协程在读取到size个字节或EOF后执行完成，并返回读取到的字节串（不包括EOF）。

readline()的语法为：

```
coroutine asyncio.StreamReader.readline()
```

请将其与IOBase的readline()比较，你会发现它除了创建一个协程外，还不具有size参数，因此该协程总是遇到EOL或EOF才执行完成，并返回读取到的字节串（包括EOL但不包括EOF）。

readexactly()是StreamReader特有的，其语法为：

```
coroutine asyncio.StreamReader.readexactly(n)
```

它与read()的区别在于，创建的协程必须精确读到n个字节后才会执行完成，如果在这之前读到了EOF，则会抛出asyncio.IncompleteReadError异常。该异常类型是EOFError的子类，具有如下属性：

- expected：引用一个整数，即调用readexactly()时通过参数n传入的数。
- partial：引用一个字节串，包含已经读取到的字节。

readuntil()也是StreamReader特有的，其语法为：

```
coroutine asyncio.StreamReader.readuntil(sep=b'\n')
```

它大体上可被视为readline()的扩展，可以通过sep参数指定别的字节（或字节串）作为分隔符。此外，readuntil()与readline()还存在如下两个区别：

➤ 当readuntil()创建的协程在遇到EOF之前还没有遇到指定的分隔符时，不会认为读到了完整的数据，而会抛出asyncio.IncompleteReadError异常，并通过其partial属性给出已经读取到的字节（此时expected属性无意义）。
➤ 当readuntil()创建的协程读取到的字节总数超出了StreamReader对象的缓冲区大小限制时，会抛出asyncio.LimitOverrunError异常，并通过其consumed属性给出这次操作占用了缓冲区中的多少字节。

最后，at_eof()是一个函数，其语法为：

```
asyncio.StreamReader.at_eof()
```

如果它返回True，则表明StreamReader对象已经遇到了EOF；否则返回False。

表14-13列出了StreamWriter类的属性，其中写操作也都是异步的，因此同样不会造成阻塞。下面依次讨论这些属性。

表14-13. StreamWriter的属性

属性	说明
transport	引用绑定的传输对象。
get_extra_info()	获取绑定的传输对象的指定信息。
write()	向异步流中写入指定类字节对象。
writelines()	向异步流中写入多行。
drain()	一个协程函数，等待写操作完成。
can_write_eof()	判断是否可使用write_eof()。
write_eof()	将EOF写入异步流。
close()	关闭异步流。
wait_closed()	一个协程函数，等待关闭操作完成。

属性	说明
`is_closing()`	判断异步流是否正在关闭或已经关闭。
`start_tls()`	一个协程函数，将一个TCP连接升级成TLS连接。

transport属性引用的是Reader对象和Writer对象共同绑定的传输对象，但仅Writer对象提供了该接口。get_extra_info()则用于查询该传输对象的相关信息。这两个属性是通往低层级API的接口，本书不详细讨论。

write()的语法为：

```
asyncio.StreamWriter.write(b)
```

请将其与RawIOBase的write()比较，你会发现两者的语法完全一样。但该函数返回只意味着通过b参数传入的类字节对象已经开始写入异步流，但什么时候完成尚未知。

writelines()的语法为：

```
asyncio.StreamWriter.writelines(lines)
```

请将其与IOBase的writelines()比较，你会发现两者的语法也完全一样。同样，该函数返回只意味着开始将通过lines传入的类字节对象列表写入异步流，但什么时候完成尚未可知。

不论调用write()还是writelines()，将数据写入异步流的过程都会因该异步流是否有缓冲区而存在区别。如果异步流具有缓冲区，则会设置一个低水位和一个高水位，然后按照如下规则完成写入：以一个均匀的速率将数据写入异步流的缓冲区，当缓冲区中的数据量达到高水位时暂停写入，等到数据量降到低水位时继续写入，如此反复，以确保数据不会丢失。而如果异步流不具有缓冲区，则会直接以一个均匀的速率将数据提交给异步流发送，不在乎数据是否丢失。

一般而言，在调用了write()或writelines()后，不需要等待写操作完成，可以执行其他操作，甚至可以再次调用write()或writelines()，新指定的数据会自动在原数据后排队。然而有些情况下需要确保数据完成写入后再执行其他操作，此时可以使用drain()，其语法为：

```
coroutine asyncio.StreamWriter.drain()
```

如果通过调用该协程函数创建协程时尚有写操作未完成，则该协程会被挂起，直到所有写操作都完成后才执行完成；否则，该协程立刻执行完成。

write_eof()的语法为：

```
    asyncio.StreamWriter.write_eof()
```

其作用是将EOF写入异步流，然后关闭该异步流的写入端，使得调用write()和writelines()失去作用。并非所有的异步流都支持该操作，因此在调用write_eof()前最好通过调用can_write_eof()进行判断，其语法为：

```
    asyncio.StreamWriter.can_write_eof()
```

如果该函数返回True，则说明异步流支持write_eof()；否则返回False。

close()用于关闭异步流，其语法为：

```
    asyncio.StreamWriter.close()
```

由于异步流涉及两个进程，所以一个进程调用close()后只能关闭异步流的一端，并向另一个进程发送关闭了异步流的通知。要等到后者也调用了close()后异步流才真正关闭。

close()返回只意味着己方关闭异步流的通知已经被发送，并不意味着对方关闭异步流的通知已收到。如果要确认异步流关闭，则需要使用wait_closed()，其语法为：

```
    coroutine asyncio.StreamWriter.wait_closed()
```

该协程函数创建的协程要等到收到了对方关闭异步流的通知，或等待超时后才执行完成。

is_closing()则用于判断一个异步流是否已关闭，其语法为：

```
    asyncio.StreamWriter.is_closing()
```

如果它返回True，则说明异步流正在关闭或已经关闭；否则返回False。

最后，当StreamWriter对象绑定的异步流是一个TCP连接时，可以通过调用start_tls()将其升级为一个TLS连接，以确保数据传输的安全性。start_tls()的语法为：

```
    coroutine asyncio.StreamWriter.start_tls(sslcontext, \*,
ssl_handshake_timeout=None, server_hostname=None)
```

其中sslcontext参数必须被传入一个SSLContext类的实例，而ssl_handshake_timeout参数和server_hostname参数则与在asyncio.start_server()和asyncio.open_connection()中含义相同，将在下面讨论。这里只需要知道，start_tls()是Python 3.11新增加的。

至此，我们已经知道了如何通过StreamReader对象对一个异步流执行读操作，以及通过StreamWriter对象对一个异步流执行写操作。然而我们不应直接实例化这两种对象，而应通过调用如下协程函数直接或间接地获得它们：

- asyncio.start_server()：创建一个使用TCP通信的服务器。
- asyncio.open_connection()：建立一个到使用TCP通信的服务器的连接。
- asyncio.start_unix_server()：创建一个使用Unix套接字通信的服务器。
- asyncio.open_unix_connection()：建立一个到使用Unix套接字通信的服务器的连接。
- asyncio.create_subprocess_exec()：创建一个子进程来执行指定的程序。
- asyncio.create_subprocess_shell()：创建一个子进程来执行指定的shell命令。

本节会详细讨论前4个协程函数，而后2个协程函数将放到下节讨论。

网络I/O必然绕不过“伯克利套接字（Berkeley sockets）”，所有主流操作系统都提供了伯克利套接字的某种实现。Python标准库中的socket模块封装了Unix/Linux和Windows上的伯克利套接字实现，是用Python进行网络编程的最底层API之一。

由于Internet和TCP/IP都是开放的，所以进行网络I/O时必须依赖第三方技术来构建安全的信道，而该技术就是SSL/TLS。目前所有主流操作系统都安装了openssl作为SSL/TLS的实现。Python标准库中的ssl模块封装了openssl，同样是用Python进行网络编程的最底层API之一。

理论上仅使用socket模块和ssl模块就足以编写出任何网络应用，但它们使用起来比较麻烦，开发效率不高。asyncio通过asyncio.start_server()和asyncio.open_connection()这两个函数封装了socket模块和ssl模块最主要的功能，极大地降低了网络编程的难度。我们不详细讨论socket模块和ssl模块，因为网络编程涉及的细节知识太多，超出了本书的范围。下面只讨论如何用asyncio.start_server()和asyncio.open_connection()编写简单的网络应用。

---

asyncio.start_server()用于创建基于TCP连接的服务器，其语法为：

```
    coroutine asyncio.start_server(client_connected_cb, host=None,
port=None, *, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE,
sock=None, reuse_address=None, reuse_port=None, ssl=None,
ssl_handshake_timeout=None, backlog=100, start_serving=True,
limit=None)
```

其client_connected_cb参数可以被传入一个如下格式的协程函数：

```
    async def callback(reader, writer):
        ... any code goes here ...
```

也可以被传入一个如下格式的回调函数：

```
    def callback(reader, writer):
        ... any code goes here ...
```

这两种可调用对象都是在客户与服务器建立了连接时被调用：对于前者来说，创建的协程会自动被包装成一个任务来调度；而对于后者来说，通常会利用下一节介绍的方法创建一个调度协程的子进程。不论任务还是子进程，都需要服务于该客户：Python解释器会自动将客户与服务器之间建立的连接抽象为一个异步流，并创建绑定到其上的StreamReader对象和StreamWriter对象，分别通过上述协程函数或回调函数的reader形式参数和writer形式参数传入。这样该协程函数或回调函数就可以通过这两个对象读写异步流。

asyncio.start_server()的limit参数用于设置自动创建的StreamReader对象的缓冲区的大小，如果传入None则取默认值64kB。

asyncio.start_server()的其余参数用于设置该服务器使用的伯克利套接字和SSL/TLS，以及是否自动启动。它们会被传入属于asyncio低层级API的loop.create_server()。下面依次讨论。

➤ host参数决定了该服务器将被绑定到哪个或哪些IP地址，其可取值包括：
1. 一个代表IPv4地址或IPv6地址的字符串：将服务器绑定到该IP地址。
2. 一个由代表IPv4地址或IPv6地址的字符串构成的序列：将服务器绑定到这些IP地址。
3. 空串或None：将服务器绑定到主机的所有IP地址。

➤ port参数决定了该服务器将监听哪个TCP端口，需被传入一个正整数。如果被传入0或None，则会为每个绑定的IP地址随机选择一个TCP端口。

➤ family参数用于控制如何解读host参数，其可取值为：
1. socket.AF_INET：将传入host参数的字符串强制解读为IPv4地址。
2. socket.AF_INET6：将传入host参数的字符串强制解读为IPv6地址。
3. socket.AF_UNSPEC：根据传入host参数的字符串的格式自动判断。

➤ 在指定了host、port和family这三个参数后，能够通过哪些IP地址和TCP端口的组合访问该服务器就已经确定了。然而在创建服务器时，必须先通过getaddrinfo()系统调用将这些信息转换为一个套接字列表，然后让服务器使用这些套接字。flags参数即用于控制该系统调用的行为，但由于这过于底层，本书不详细讨论，只需要知道让flags参数取默认实参值就可以了。

➤ sock参数用于传入已经创建好的伯克利套接字。如果它没有被传入None，那么就不需要使用getaddrinfo()系统调用，host、port、family和flags参数也都会被忽略。但创建伯克利套接字需要使用socket模块提供的函数，本书不进一步讨论。

➤ 如果reuse_address参数被传入True，那么操作系统可以重用已经创建好的套接字。如果给该参数传入False，则禁用这一机制。注意如果给该参数传入None，则在Unix或类Unix操作系统上等同于传入了True，而在Windows上等同于传入了False。

➤ 如果reuse_port参数被传入True，那么在Unix和类Unix操作系统上允许一个TCP端口被多个套接字共享（前提是它们绑定的IP地址不同）。如果reuse_port参数被传入False，则禁用这一机制。如果给该参数传入None，则在Unix和类Unix操作系统上等同于传入了False。Windows不支持该机制，因此在Windows上该参数会被忽略。

➤ 如果ssl被传入一个SSLContext对象，则会启用SSL/TLS来保护该TCP连接。如果ssl被传入None，则创建的是纯TCP连接。但SSLContext对象需要使用ssl模块提供的类，本书不进一步讨论。

➤ ssl_handshake_timeout参数可被传入一个正整数或正浮点数，代表的是在放弃尝试启用SSL/TLS之前最多等待的秒数。如果该参数被传入None，则默认最多等待60秒。

➤ backlog参数用于控制排队等待建立TCP连接的请求的最大数量，可被传入一个正整数。当请求数达到了这一数量时，后续的请求会被直接拒绝。如果给该参数传入None，则默认允许100个请求排队等待。

➤ 最后，如果start_serving参数被传入True，则服务器被创建后会立刻启动（相当于自动调用相应Server对象的start_serving()属性）。如果该参数被传入False，则服务器被创建后需要手工启动。

因为创建TCP服务器需要等待TCP握手以及SSL/TLS握手，可能需要花费很长的时间，所以asyncio.start_server()是一个协程函数，其创建的协程要等到服务器成功创建后才执行完成（否则会抛出异常），并返回一个Server类的实例。Server对象具有表14-14列出的属性，以控制该服务器。下面依次讨论这些属性。

表14-14. Server的属性

属性	说明
sockets	引用服务器监听的套接字列表。
get_loop()	取得与服务器关联的事件循环。
is_serving()	判断该服务器是否已经开始服务。
start_serving()	一个协程函数，启动服务器。
close()	关闭服务器。
wait_closed()	一个协程函数，等待服务器关闭。
serve_forever()	一个协程函数，使服务器运行到控制协程停止运行。

sockets属性引用了一个列表，其内的元素是该服务器使用的伯克利套接字的副本。该属性只用于查询，对其的一切修改都不会影响到服务器。

get_loop()的作用是获取该服务器关联到的事件循环。作为通向低层级API的接口，本书不详细讨论。

is_serving()的作用是判断该服务器当前有没有启动，其语法为：

```
asyncio.Server.is_serving()
```

其返回True表明服务器已经启动，可以接受客户发来的连接请求；否则返回False。

start_serving()是手工启动服务器的标准方法，其语法为：

```
coroutine asyncio.Server.start_serving()
```

该协程函数创建的协程在服务器成功启动后执行完成。如果服务器已经启动，则该协程会立刻执行完成，对服务器没有不好的影响。

close()是手工关闭服务器的标准方法，其语法为：

```
asyncio.Server.close()
```

与StreamWriter中的close()类似，调用该函数会将服务器即将关闭的消息通知所有连接到该服务器的客户，而服务器要等到收到每个客户的响应，或者等待超时，然后断开所有TCP连接，正式关闭。close()返回仅意味着通知已经被发送。

如果要确定服务器被关闭，则需要使用wait_closed()，其语法为：

```
coroutine asyncio.Server.wait_closed()
```

该协程函数创建的协程要等到服务器正式停止后才会执行完成。

最后，serve_forever()是控制服务器启动和关闭的另一个方法，其语法为：

```
coroutine asyncio.Server.serve_forever()
```

不论服务器当前有没有启动，都可以调用该协程函数以获得一个控制协程，未启动的服务器会自动启动，此后关闭该服务器的方法是停止该控制协程（通常通过取消包装它的任务）。需要强调的是，对一个服务器只能调用一次serve_forever()，即一个服务器最多只能有一个控制协程。

至此我们讨论完了如何创建基于TCP连接的服务器，下面讨论如何创建基于TCP连接的客户，即asyncio.open_connection()，其语法为：

```
coroutine asyncio.open_connection(host=None, port=None, *,
family=0, flags=0, proto=0, happy_eyeballs_delay=None,
interleave=None, local_addr=None, sock=None, ssl=None,
ssl_handshake_timeout=None, server_hostname=None, limit=None)
```

该协程函数创建的协程在TCP连接成功建立后执行完成，返回一个(reader, writer)形式的元组，其中reader为绑定到新建异步流的StreamReader对象，writer为绑定到新建异步流的StreamWriter对象。

asyncio.open_connection()的limit参数同样用于设置自动创建的StreamReader对象的缓冲区的大小，如果传入None则取默认值64kB。

asyncio.open_connection()的其余参数用于设置该客户使用的伯克利套接字和SSL/TLS。它们会被传入属于asyncio低层级API的loop.create_connection()。下面依次讨论。

➤ 首先，sock参数同样用于传入已经创建好的伯克利套接字。如果它没有被传入None，那么host、port、family、flags、proto、happy_eyeballs_delay、interleave和local_addr参数都会被忽略。

➤ 在sock参数被传入None的情况下，必须给host参数传入一个代表IPv4地址、IPv6地址或完全限定域名（FQDN）的字符串，给port传入一个代表TCP端口号的正整数，它们共同指定了向哪个服务器发起TCP连接请求。

➤ family参数同样用于指定解读host参数的方法，但当host参数被传入一个FQDN时，必须通过DNS将其转换为IP地址，此时family参数只能取None或socket.AF_UNSPEC。

➤ flags参数和proto参数都用于控制getaddrinfo()系统调用的行为，本书不详细讨论。

➤ happy_eyeballs_delay参数和interleave参数都用于控制使用DNS返回记录的方式。如果给happy_eyeballs_delay传入一个正浮点数（推荐0.25），那么当FQDN转化为多个IP地址，且其中有IPv4也有IPv6时，会以该浮点数作为间隔的秒数分别针对IPv4地址集合和IPv6地址集合逐步发起多个TCP连接请求，每个请求都对应不同的IP地址。而只要其中有一个TCP连接建立成功，其他的请求就会被主动放弃。如果该参数被传入None则禁用该机制。

➤ interleave参数则控制着是否对DNS返回记录中的IP地址集合按照地址族进行重排序。如果该参数被传入None或0，则不进行重排序；如果被传入一个正整数，则进行重排序。当happy_eyeballs_delay被传入None时，interleave参数默认被传入0；否则，interleave参数默认被传入1。

➤ local_addr参数应被传入一个(local_host, local_port)格式的二元组，以控制客户端套接字绑定的IP地址和TCP端口。如果它被传入None，则通过getaddrinfo()自动查找使用哪个IP地址和TCP端口。

➤ ssl参数和ssl_handshake_timeout参数的含义与在asyncio.start_server()中相同。而额外的server_hostname参数则需被传入一个代表服务器主机名的字符串，用于和服务器提供的x.509证书中的信息进行比对。如果server_hostname被传入None，则会使用host参数被传入的FQDN中包含的主机名。如果server_hostname被传入空串，则取消主机名比对（但这会带来安全隐患）。显然，仅当ssl被传入一个SSLContext对象时，server_hostname参数才有意义。

至此，我们终于可以给出第一个用Python进行网络编程的例子了。按照惯例，我们编写一个“反射服务器（echo server）”，其功能是将客户发送来的数据原封不动地返回。这是最简单的网络应用。

首先编写作为服务器运行的脚本coroutine31.py：

```
#!/usr/bin/env python3

import asyncio
import sys

#指定服务器绑定的IP地址。
host = '127.0.0.1'
#指定服务器监听的TCP端口。
port = 8000


#该协程函数创建的协程处理连接请求。
async def echo(reader, writer):
    #在TCP连接成功建立后，可以一直响应客户发来的消息，并将消息反射给客户，直到收到特殊
    # 消息“quit”。
    while True:
        #异步读取一行二进制数据，注意这会包含\n。
        data = await reader.readline()
        #将二进制数据转换为文本消息。
        msg = data.decode()
        #将文本消息再次转换为二进制数据。
        data = msg.encode()
        #将二进制数据原封不动地异步写入。
        writer.write(data)
        #如果读取到的文本消息是“quit”，则退出循环。
        if msg == "quit\n":
            break
    #关闭异步流。
    writer.close()


async def main():
    #创建服务器，并绑定到指定的IP，在指定的TCP端口上监听。   服务器被创建后并未启动。
    server = await asyncio.start_server(echo, host, port,
start_serving=False)
    #创建服务器的控制任务。
    controller = asyncio.create_task(server.serve_forever())
    try:
```

```
            #启动并运行服务器60秒。
            await asyncio.wait_for(controller, 60)
        except asyncio.TimeoutError:
            #60秒后自动关闭服务器。
            controller.cancel()
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

该脚本包含这些设计要点：

➤ 默认建立的TCP连接是长连接，因此在异步流被关闭之前，客户可以多次向服务器发送消息并获得反射。
➤ 消息必须是以“\n”为EOL的一行文本，转换为二进制数据后“\n”会被保留。这样可以使用readline()来读取。
➤ 文本消息转换为二进制数据可以通过字符串的encode()实现，而二进制数据转换为文本消息可以通过二进制序列的decode()实现，它们默认都采用UTF-8编码。

然后编写作为客户运行的脚本coroutine32.py：

```
    #!/usr/bin/env python3

    import asyncio
    import sys

    #指定服务器的IP地址。
    host = '127.0.0.1'
    #指定服务器的TCP端口。
    port = 8000


    async def main():
        #建立到服务器的TCP连接，并获得相关异步流的读取器和写入器。
        reader, writer = await asyncio.open_connection(host, port)
        #在TCP连接成功建立后，可以一直向服务器发送消息，并将反射来的消息通过标准输出显示，
        # 直到发送了特殊消息“quit”。
        while True:
            #让用户随意输入一行文本，注意这不会包含\n。
            text = input('You can send any message to the server, "quit" to
stop: ')
            #给文本添加\n以形成文本消息。
            msg = text + "\n"
            #将文本消息转换为二进制数据。
            data = msg.encode()
            #将二进制数据异步写入。
            writer.write(data)
            #异步读取一行二进制数据，注意这会包含\n。
            data = await reader.readline()
            #将二进制数据转换为文本消息。
            msg = data.decode()
            #显示文本消息。
            print(msg)
            #如果读取到的文本消息是“quit”，则退出循环。
```

```
            if msg == "quit\n":
                break
        #关闭异步流。
        writer.close()
        sys.exit(0)


if __name__ == '__main__':
    asyncio.run(main())
```

该脚本包含如下设计要点：通过input()获得的文本是不包含最后的\n的，必须手工添加才能形成完整的文本消息。

如果你使用的是Unix或类Unix操作系统，则先通过如下命令行让coroutine31.py在后台运行以作为服务器：

```
$ python3 coroutine31.py &
[1] 40860
```

该服务器被绑定到本地环回IPv4地址（“127.0.0.1”），并在TCP端口8000上监听。然后通过如下命令行让coroutine32.py在前台运行以作为客户：

```
$ python3 coroutine32.py
You can send any message to the server, "quit" to stop: Hello
Hello

You can send any message to the server, "quit" to stop: World!
World!

You can send any message to the server, "quit" to stop: Hello World!
Hello World!

You can send any message to the server, "quit" to stop: quit
quit
```

该客户自动建立了到本地环回的8000端口的TCP连接。用户可以输入任何文本消息，按下Enter键提交，而该文本消息会被回显。这就证明了该反射服务器程序已经编写成功。用户可以通过输入“quit”来关闭客户，这也意味着异步流被关闭。但即使没有异步流，该服务器依然存在，可以通过再次执行“python3 coroutine32.py”建立一个到服务器的新TCP连接，亦即新的异步流。服务器会在1分钟后自动关闭，这不会导致客户自动关闭，但此后客户再次提交消息会获得ConnectionResetError异常。

如果你使用的是Windows，则需要打开两个运行cmd命令的窗口，分别执行“python3 coroutine31.py”和“python3 coroutine32.py”，前者扮演服务器，后者扮演客户。后面的例子也是一样的，不再重复提示。

你也可以在两台主机上分别运行coroutine31.py和coroutine32.py，但此时服务器绑定的IP地址就不能选择本地环回，必须是可被其他主机访问的IP地址，例如“192.168.0.10”。

下面我们对反射服务器稍作修改，使得它支持如下命令：
upper：将文本消息中的英文字符全部改为大写。
lower：将文本消息中的英文字符全部改为小写。
swapcase：将文本消息中的英文字符的大小写反转。
capitalize：将文本消息中的英文首字符大写，其余小写。
title：将文本消息中的英文单词首字符大写，其余小写。
quit：关闭异步流。
这些命令必须以“（command, message）”的格式发送。

作为服务器运行的脚本coroutine33.py的内容是：

```
#!/usr/bin/env python3

import asyncio
import sys

host = '127.0.0.1'
port = 8001


async def engine(reader, writer):
    while True:
        #从异步流读取“(command, message)”格式的二进制数据。
        binary = await reader.readuntil(b")")
        #将二进制数据转换为文本数据。
        text = binary.decode().strip()
        #从文本数据中抽取命令和消息。
        command, _, message = text.partition(",")
        command = command.strip("( ")
        message = message.strip(") ")
        #根据命令对消息做不同的处理。
        match command.lower():
            case "upper":
                message = message.upper()
            case "lower":
                message = message.lower()
            case "swapcase":
                message = message.swapcase()
            case "capitalize":
                message = message.capitalize()
            case "title":
                message = message.title()
            case "quit":
                message = "quit"
            case _:
                message = "unknown command."
        #基于消息生成文本数据。
        text = message + "\n"
        #将文本数据转换为二进制数据。
        binary = text.encode()
        #将二进制数据写入异步流。
        writer.write(binary)
        if message == "quit":
            break
    writer.close()
    sys.exit(0)
```

```
    async def main():
        #创建服务器，并让其自动启动。
        server = await asyncio.start_server(engine, host, port)
        #等待60秒。
        await asyncio.sleep(60)
        #关闭服务器。
        server.close()


    if __name__ == '__main__':
        asyncio.run(main())
```

该脚本包含这些设计要点：

➤ 以readuntil()代替readline()读取数据，以“)”作为分界点。然后解析读取到的数据，从中抽取出命令和消息。
➤ 没有通过控制任务来控制服务器，而是直接通过Server对象来控制服务器。

作为客户运行的脚本coroutine34.py的内容是：

```
    #!/usr/bin/env python3

    import asyncio
    import sys

    host = '127.0.0.1'
    port = 8001


    async def main():
        reader, writer = await asyncio.open_connection(host, port)
        while True:
            text = input('Input a "(command, message)" style request, "(quit,)"
to stop: ')
            #将文本数据转换为二进制数据，不需要添加\n。
            data = text.encode()
            writer.write(data)
            data = await reader.readline()
            msg = data.decode()
            print(msg)
            if msg == "quit\n":
                break
        writer.close()
        sys.exit(0)


    if __name__ == '__main__':
        asyncio.run(main())
```

该脚本包含如下设计要点：

➤ 由于服务器通过readuntil()来读取数据，因此没有必要在数据中插入\n。

请通过如下命令行让coroutine33.py在后台运行以作为服务器：

```
$ python3 coroutine33.py &
[1] 41113
```

该服务器同样被绑定到本地环回IPv4地址，但在TCP端口8001上监听。然后通过如下命令行让coroutine34.py在前台运行以作为客户：

```
$ python3 coroutine34.py
Input a "(command, message)" style request, "(quit,)" to stop: (Upper,
Hello World!)
HELLO WORLD!

Input a "(command, message)" style request, "(quit,)" to stop: (lower,
Hello World!)
hello world!

Input a "(command, message)" style request, "(quit,)" to stop: (SWAPCASE,
Hello World!)
hELLO wORLD!

Input a "(command, message)" style request, "(quit,)" to stop: (quit,)
quit
```

本书不再给出更多的网络编程的例子，只在这里额外提一下编写网络应用时的要点：

➤ 在网络I/O中，一个进程写入EOF意味着此后不再会写入其他数据，因此另一个进程会使用read()一次性读取所有数据，然后同样一次性写入所有数据，再关闭异步流。而前者也需要使用read()一次性读取所有数据，再关闭异步流。这种建立一次TCP连接只交换一次数据的方式，典型例子是早期的HTTP协议。
➤ 如果想通过一个TCP连接多次交换数据，则不能依赖于EOF来划分数据。最方便的方式是将数据用EOL组织成行，通过readline()按行读取。但如果数据有特定的格式，则使用readuntil()或readexactly()更加方便。
➤ 在建立了TCP连接后，交换的数据可以采用任意格式。不同的网络应用通常会自己规定这些数据采用的格式，这就形成了所谓的“互联网应用协议”。

我们对用异步流进行网络I/O就讨论到这里。下面讨论用异步流进行IPC。事实上操作系统通常会提供多种IPC方法，不同的操作系统还存在差异。

Python标准库通过signal模块实现了“信号（signals）”，通过mmap模块实现了“内存映射（memory map）”，它们都属于IPC。然而对它们的讨论超出了本书的范围。

Unix和类Unix操作系统中的“Unix套接字（Unix sockets）”也属于IPC，由于它们使用起来与伯克利套接字几乎没有区别，所以深受开发者欢迎。然而需要强调，Windows不支持Unix套接字。

Unix套接字本质上是被所有相关进程共享的一个文件。创建基于Unix套接字的服务器的方法是使用asyncio.start_unix_server()，其语法为：

```
    coroutine asyncio.start_unix_server(client_connected_cb,
path=None, *, sock=None, ssl=None, ssl_handshake_timeout=None,
backlog=100, start_serving=True, limit=None)
```

除了用path参数传入的文件路径取代了host，port，family和flags参数提供的信息，且不再支持reuse_address和reuse_port参数外，该函数与asyncio.start_server()没有区别。

而创建使用Unix套接字的客户的方法则是使用asyncio.open_unix_connection()，其语法为：

```
    coroutine asyncio.open_unix_connection(path=None, *,
sock=None, ssl=None, ssl_handshake_timeout=None,
server_hostname=None, limit=None)
```

同样，除了用path参数取代host，port，family，flags，proto，happy_eyeballs_delay，interleave和local_addr参数外，该函数与asyncio.open_connection()没有区别。

注意在上述两个函数中，path参数可以被传入一个字符串、字节串或Path对象。不论哪种形式，它都应代表着一个相对路径。asyncio.start_unix_server()会自动在执行它的脚本所在目录下搜过该相对路径指定的子目录和文件，如果文件不存在则会自动创建，但如果子目录不存在则会报错。

下面通过将基于TCP连接的反射服务器该写为基于Unix套接字的反射服务器来说明上述两个函数的用法。请将coroutine31.py拷贝至coroutine35.py，将全局变量host和port去掉，添加全局变量path：

```
    #指定实现Unix套接字的文件。
    path = "unix_socks/temporary.sock"
```

然后将main()中包含asyncio.start_server()的语句替换成：

```
    #创建服务器，并指定使用的Unix套接字。
    server = await asyncio.start_unix_server(echo, path, start_serving=False)
```

类似的，将coroutine32.py拷贝至coroutine36.py，同样去掉全局变量host和port，添加全局变量path并使其指向同一个文件。然后将main()中包含asyncio.open_connection()的语句替换成：

```
    #建立到服务器的Unix套接字连接，并获得相关异步流的读取器和写入器。
    reader, writer = await asyncio.open_unix_connection(path)
```

最后通过如下命令行进行验证（你需要先在工作目录下手工创建unix_socks目录）：

```
    $ python3 coroutine35.py &
    [1] 41277

    $ python3 coroutine36.py
    You can send any message to the server, "quit" to stop:
```

可以验证它们的功能与coroutine31.py和coroutine32.py完全相同。但要注意，由于Unix套接字是通过文件实现的，所以不能将coroutine35.py和coroutine36.py分别放在两台主机上，也就是说Unix套接字是一种本地IPC技术。

14-11. 跨线程/进程调度协程

（标准库：asyncio）

Python标准库提供了threading模块以支持基于线程的并行，提供了multiprocessing模块以支持基于进程的并行，但对它们的讨论都超出了本书的范围。asyncio模块提供的绝大部分函数和类都假设Python脚本在单线程环境中运行。然而创建一个线程来执行一个阻塞I/O操作，以及创建一些子进程来执行其他程序，并通过协程来取得它们的运行结果，在有些情况下是非常有用的。asyncio提供了三个协程函数来满足此类情况下的需要。

---

asyncio.to_thread()是一个协程函数，其语法为：

```
    coroutine asyncio.to_thread(func, /, *args, **kwargs)
```

该协程函数被调用时，会自动创建一个线程，由该线程执行func(*args, **kwargs)，然后创建一个协程，该协程会挂起到func(*args, **kwargs)返回，然后相应线程被销毁，而该函数的返回值会成为该协程的返回值。

下面的例子说明了如何使用asyncio.to_thread()：

```
#!/usr/bin/env python3

import asyncio
import time
import sys


#该函数模拟需要执行阻塞I/O的函数。
def blocked_io(oprand1, oprand2, operator):
    #阻塞3秒。
    time.sleep(3)
    return eval(str(oprand1) + str(operator) + str(oprand2))


async def main():
    print(f'start at {time.strftime("%X")}.')
    #创建四个线程分别执行blocked_io()的一次调用。
    coro1 = asyncio.to_thread(blocked_io, 35, 17, "+")
    coro2 = asyncio.to_thread(blocked_io, 35, 17, "-")
    coro3 = asyncio.to_thread(blocked_io, 35, 17, "*")
    coro4 = asyncio.to_thread(blocked_io, 35, 17, "/")
    #使这四个线程并行执行。
    result = await asyncio.gather(coro1, coro2, coro3, coro4)
    print(f'{time.strftime("%X")}: 35+17={result[0]}')
    print(f'{time.strftime("%X")}: 35-17={result[1]}')
    print(f'{time.strftime("%X")}: 35*17={result[2]}')
    print(f'{time.strftime("%X")}: 35/17={result[3]}')
    sys.exit(0)

if __name__ == '__main__':
    asyncio.run(main())
```

请将上面的代码保存为coroutine37.py，然后通过如下命令行验证：

```
$ python3 coroutine37.py
start at 19:09:34.
19:09:37: 35+17=52
19:09:37: 35-17=18
19:09:37: 35*17=595
19:09:37: 35/17=2.0588235294117645
```

在上面的例子中，blocked_io()是一个普通的函数，通过调用time.sleep()模拟了执行耗时3秒的阻塞I/O操作。当Python脚本以单线程方式运行时，调用该函数4次将耗费12秒。然而通过调用asyncio.to_thread()创建4个线程分别执行一次该函数的调用，然后将返回的协程作为任务并行执行后，只花了3秒钟就得到了4次调用的结果。

需要强调的是，由于CPython使用了GIL，所以通过asyncio.to_thread()创建的线程只有在执行阻塞式I/O操作时才会被挂起，实现真正的并行，因此该技术对于因执行复杂计算而耗时很长的函数没有提速作用。但如果在没有使用GIL的Python实现（例如PyPy）上，即使不执行阻塞式I/O操作的多个线程也能被多核CPU并行执行，达到提速的目的。

asyncio.create_subprocess_exec()的功能是创建一个子进程来运行指定的程序（该程序不一定是Python脚本，可以是任意编程语言编写的可执行文件）。该协程函数的语法为：

```
    coroutine asyncio.create_subprocess_exec(program, *args,
stdin=None, stdout=None, stderr=None, limit=None, **kwds)
```

其中program参数需被传入一个代表可执行文件路径的字符串或字节串，args参数接收若干字符串或字节串。它们共同指定了执行哪个程序，以及传入哪些参数，相当于通过一个新创建的子进程执行了如下命令行：

```
    $ program *args
```

asyncio.create_subprocess_exec()创建的协程会在子进程开始运行后执行完成，并返回一个asyncio.subprocess.Process对象以使得父进程可以和子进程交互。

stdin、stdout和stderr参数用于通过管道重定向该子进程的标准输入、标准输出和标准出错。它们都可以被传入如下值之一：

➤ None：表示子进程继承父进程的标准输入、标准输出或标准出错。这是默认值。
➤ asyncio.subprocess.PIPE：表示创建一个管道连接父进程和子进程，使得只有父进程可以通过asyncio.subprocess.Process对象访问子进程的标准输入、标准输出或标准出错。
➤ 类文件对象：表示将子进程的标准输入、标准输出或标准出错重定向到该文件。
➤ asycncio.subprocess.DEVNULL：表示将子进程的标准输入、标准输出或标准出错重定向到null文件（这样无法获得输入，输出将被丢弃）。

此外，stderr参数还可以被传入asyncio.subprocess.STDOUT，表示创建一个新管道将子进程的标准出错重定向到其标准输出。

仅当stdout和/或stderr参数被传入了asyncio.subprocess.PIPE时limit参数才有意义，用于设置绑定到它们的StreamReader对象的缓冲区上限。此时如果给limit参数传入None则会取默认值64kB。

除非stdin、stdout和stderr都被传入了None，否则asyncio.create_subprocess_exec()需要创建asyncio.subprocess.Popen对象来代表所需的管道，此时可以通过kwds参数给该类传入初始化参数列表。但这些参数都是对管道进行微调的，大部分情况下保留默认值即可。

asyncio.create_subprocess_shell()的功能是创建一个子shell来运行指定的shell命令（在Windows上是DOS命令）。该协程函数的语法为：

```
coroutine asyncio.create_subprocess_shell(cmd, stdin=None,
stdout=None, stderr=None, limit=None, **kwds)
```

其中cmd参数需被传入一个代表完整命令行的字符串，包括命令名称和后续的选项和参数。除了用cmd参数代替了program参数和args参数外，该函数的其余参数以及返回值都与asyncio.create_subprocess_exec()相同。

不论asyncio.create_subprocess_exec()还是asyncio.create_subprocess_shell()，执行它们所创建的协程都会返回一个asyncio.subprocess.Process对象。表14-15总结了Process对象具有的属性。

表14-15. Process的属性

属性	说明
`pid`	引用子进程的进程ID。
`returncode`	子进程运行过程中引用None；当子进程被某信号N中断时引用-N（仅适用于Unix和类Unix操作系统）；子进程运行完成后引用退出状态。
`stdin`	引用绑定到子进程的标准输入的StreamWriter对象，或者None。
`stdout`	引用绑定到子进程的标准输出的StreamReader对象，或者None。
`stderr`	引用绑定到子进程的标准出错的StreamReader对象，或者None。
`communicate()`	一个协程函数，向子进程的标准输入写入数据，并读取其标准输出和标准出错的反馈结果。
`wait()`	一个协程函数，等待子进程运行完成，并取得退出状态。
`send_signal()`	向子进程发送指定的信号。
`terminate()`	停止子进程。
`kill()`	杀死子进程。

Process对象的属性pid和returncode提供了关于子进程的基本信息：前者为子进程运行时的进程ID，后者为子进程运行完成时的退出状态。

如果在创建子进程时给stdin参数传入了asyncio.subprocess.PIPE，那么父进程将会具有一个连接到子进程的标准输入的只写管道，亦即一个基于管道的单向异步流。此时父进程获得的Process对象的stdin属性将会引用一个StreamWriter对象，使得父进程可以通过它向子进程的标准输入写入数据。这是子进程获的标准输入获得数据的唯一方式。如果创建子进程时stdin参数传入了其他值，则Process对象的stdin属性将引用None。

类似的，如果在创建子进程时给stdout或stderr参数传入了asyncio.subprocess.PIPE，那么父进程将会具有一个连接到子进程的标准输出或标准出错的只读管道，同样是一个基于管道的单向异步流。此时父进程获得的Process对象的stdout或stderr属性将会引用一个StreamReader对象，使得父进程可以通过它从子进程的标准输出或标准出错读取数据。如果创建子进程时stdout或stderr参数传入了其他值，则Process对象的stdout或stderr属性将引用None。

在调用asyncio.create_subprocess_exec()或asyncio.create_subprocess_shell()创建的协程执行完成后，相应子进程就已经开始运行了。如果父进程和子进程之间存在上述单向管道，则可以基于StreamWriter对象和StreamReader对象进行通信。一般而言，一次通信从父进程将一些数据写入子进程的标准输入开始，到父进程从子进程的标准输出和/或标准出错读取到数据为止，形成一次对话。

如果父进程和子进程之间只进行一次对话，则可以使用Process对象的communicate()，其语法为：

```
coroutine asyncio.subprocess.Process.communicate(input=None)
```

其input参数需传入一个字符串调用encode()的结果，这会使得该结果通过StreamWriter对象写入子进程的标准输入。而其创建的协程要等到子进程终止后才会执行完成，并会返回一个“(stdout_data, stderr_data)”格式的元组，其中stdout_data为子进程写入标准输出的所有数据，stderr_data为子进程写入标准出错的所有数据。它们都是通过StreamReader对象取得的，且会一直读取到遇到EOF。显然，仅当创建子进程时给stdin、stdout和stderr参数都传入asyncio.subprocess.PIPE，才能使用这一协程函数。

父进程和子进程是独立被操作系统调度执行的，因此父进程无法预计子进程什么时候执行完成。为了实现与子进程之间的同步，父进程可以使用Process对象的wait()。wait()的语法为：

```
coroutine asyncio.subprocess.Process.wait()
```

该协程函数创建的协程要到子进程退出后才执行完成，并返回子进程的退出状态。而此时该Process对象的returncode属性也将引用子进程的退出状态。

父进程可以通过Process对象的send_signal()向子进程发送任何操作系统支持的信号，进而实现基于信号的进程控制。send_signal()的语法为：

```
asyncio.subprocess.Process.send_signal(signal)
```

其中signal参数需被传入一个正整数，代表一个信号。然而由于不同操作系统提供的信号不同，同一种信号的编码也不同，所以除非你对运行Python解释器的操作系统很熟悉，否则很难用好该函数。

Process对象的terminate()的语法为：

```
asyncio.subprocess.Process.terminate()
```

在Unix和类Unix操作系统上，调用该函数会发送SIGTERM信号给子进程；而在Windows上，调用该函数会导致调用Win32 API中的TerminateProcess()。这都会导致子进程正常终止，即子进程先递归地让它创建的后代进程终止，自己再终止。

Process对象的kill()的语法为：

```
asyncio.subprocess.Process.kill()
```

在Unix和类Unix操作系统上，调用该函数会发送SIGKILL信号给子进程，这会导致子进程立刻终止，而它创建的后代进程会成为孤儿进程，被操作系统托管。一般而言，仅当调用terminate()失败时才调用kill()。而在Windows上，该函数的作用与terminate()完全相同。

下面用两个例子说明如何使用子进程。在第一个例子中，父进程不需要和子进程交互，但需要让子进程的标准I/O使用自己的标准I/O，且需要与子进程实现同步。在这种情况下，创建子进程时需给stdin、stdout和stderr参数都应传入None，并使用Process对象的wait()来等待子进程执行完成。

首先编写由子进程执行的脚本coroutine38.py，其内容为：

```
#!/usr/bin/env python3

import io
import time
import sys


#主函数先睡眠3秒，然后将第一个命令行参数指定的文件的内容拷贝到第二个命令行参数指定的
# 文件。
def main():
    time.sleep(3)
    if len(sys.argv) < 3:
        #如果命令行参数少于2个，则通过标准出错报错。
        sys.stderr.write("Not enough arguments.\n")
        sys.stderr.flush()
        return 1
    try:
        source = io.FileIO(sys.argv[1])
        target = io.FileIO(sys.argv[2], 'w')
        pair = io.BufferedRWPair(source, target)
        content = pair.read(10)
        while content:
            pair.write(content)
            content = pair.read(10)
        pair.close()
        #如果拷贝成功，则通过标准输出给出提示信息。
        sys.stdout.write(f"Successfully copy from {sys.argv[1]} to
{sys.argv[2]}.\n")
        sys.stdout.flush()
```

```
        except Exception as e:
            #如果抛出了异常，则通过标准出错报错。
            sys.stderr.write(repr(e) + "\n")
            sys.stderr.flush()
            return 2
        return 0

    if __name__ == '__main__':
        sys.exit(main())
```

然后通过如下命令行修改该脚本的模式（假设你使用的是Unix或类Unix操作系统），使其可以被直接执行：

```
    $ chmod 755 coroutine38.py
```

再通过如下命令行创建文本文件a1.txt和a3.txt：

```
    $ cat > a1.txt
    This is the text file "a1.txt"
    ^D

    $ cat > a3.txt
    This is the text file "a3.txt"
    ^D
```

最后编写由父进程执行的脚本coroutine39.py，其内容为：

```
    #!/usr/bin/env python3

    import asyncio
    import time
    import sys


    async def main():
        #显示开始执行的时间。
        print(f'start at {time.strftime("%X")}.')
        #创建三个子进程分别以不同的参数执行coroutine38.py。   子进程proc1将a1.txt拷贝
        # 到b1.txt。
        proc1 = await asyncio.create_subprocess_exec("./coroutine38.py",
"a1.txt", "b1.txt")
        #子进程proc2将a2.txt拷贝到b2.txt。
        proc2 = await asyncio.create_subprocess_exec("./coroutine38.py",
"a2.txt", "b2.txt")
        #子进程proc3将a3.txt拷贝到b3.txt。
        proc3 = await asyncio.create_subprocess_exec("./coroutine38.py",
"a3.txt", "b3.txt")
        #等待三个子进程都执行完成。
        result = await asyncio.gather(proc1.wait(), proc2.wait(), proc3.wait(),
return_exceptions=True)
        #显示三个子进程的退出状态。
        print(result)
        #显示执行完成的时间。
```

```
        print(f'end at {time.strftime("%X")}.')
        sys.exit(0)

    if __name__ == '__main__':
        asyncio.run(main())
```

现在，请通过如下命令行验证该例子：

```
    $ python3 coroutine39.py
    start at 12:25:26.
    FileNotFoundError(2, 'No such file or directory')
    Successfully copy from a3.txt to b3.txt.
    Successfully copy from a1.txt to b1.txt.
    [0, 2, 0]
    end at 12:25:30.

    $ python3 coroutine39.py
    start at 12:26:25.
    Successfully copy from a1.txt to b1.txt.
    FileNotFoundError(2, 'No such file or directory')
    Successfully copy from a3.txt to b3.txt.
    [0, 2, 0]
    end at 12:26:28.

    ...
```

从该结果可以看出如下3点：

1. 三个子进程是并行执行的。
2. 三个子进程是由操作系统调度的，因此执行的顺序不被父进程控制。
3. 三个子进程的总耗时并不稳定，因为有可能有其他的进程插入了它们的运行序列中。

在第二个例子中，子进程会永远运行下去，直到父进程调用了Process对象的terminate()或kill()。而在子进程的运行期间，父进程需要多次与之交互。在这种情况下，创建子进程时需给stdin、stdout和stderr参数都应传入asyncio.subprocess.PIPE，并使用Process对象的stdin、stdout和stderr来与子进程交互。

首先编写由子进程执行的脚本coroutine40.py，其内容为：

```
    #!/usr/bin/env python3

    import sys


    #该程序判断一个正整数是否是质数。
    def main():
        while True:
            try:
                #从标准输入读取整数。
                n = int(input())
```

```
                if n < 0:
                    raise RuntimeError()
                if n == 0 or n == 1:
                    #将结果写入标准输出。
                    print(f"{n} is not a prime.")
                    continue
                for m in range(2, n):
                    if n % m == 0:
                        #将结果写入标准输出。
                        print(f"{n} is not a prime, it equals {n//m}*{m}.")
                        break
                #将结果写入标准输出。
                if m == n-1:
                    print(f"{n} is a prime.")
            except Exception:
                #将错误信息写入标准出错。
                sys.stderr.write("Only accept natural number or -1!\n")
                sys.stderr.flush()
        return 0

    if __name__ == '__main__':
        sys.exit(main())
```

然后编写由父进程执行的脚本coroutine41.py，其内容为：

```
    #!/usr/bin/env python3

    import asyncio
    import sys

    async def console(proc):
        while True:
            try:
                #从标准输入读取数据。
                data = input("Please enter a natural number, -1 means quit: ")
                #当读取到-1时终止程序。
                if data == "-1":
                    print("Closed!")
                    #向子进程的标准输入写入EOF。
                    proc.stdin.write_eof()
                    try:
                        #先尝试正常终止子进程。
                        proc.terminate()
                    except Exception:
                        #杀死子进程。
                        proc.kill()
                    #跳出循环，终止程序。
                    break
                #将读取到的数据写入子进程的标准输入。
                proc.stdin.write((data + "\n").encode())
                #创建一个任务从子进程的标准输出中读取一行。
                task1 = asyncio.create_task(proc.stdout.readline())
                #创建一个任务从子进程的标准出错中读取一行。
                task2 = asyncio.create_task(proc.stderr.readline())
                #等待两个任务之一完成。
                done, pending = await asyncio.wait([task1, task2],
return_when=asyncio.FIRST_COMPLETED)
                #输出子进程返回的信息。
                for task in done:
                    print(task.result().decode())
                #取消尚未完成的任务。
```

```
                    for task in pending:
                        task.cancel()
                except Exception as e:
                    print(repr(e))
            return 0


        async def main():
            #创建一个子进程执行coroutine40.py。
            proc = await asyncio.create_subprocess_exec("./coroutine40.py",
stdin=asyncio.subprocess.PIPE, stdout=asyncio.subprocess.PIPE,
stderr=asyncio.subprocess.PIPE)
            #将控制该子进程的Process对象传给console，创建相应协程并等待。
            await console(proc)
            sys.exit(0)

        if __name__ == '__main__':
            asyncio.run(main())
```

现在，请通过如下命令行验证该例子：

```
        $ python3 coroutine41.py
        Please enter a natural number, -1 means quit: 9
        9 is not a prime, it equals 3*3.

        Please enter a natural number, -1 means quit: 17
        17 is a prime.

        Please enter a natural number, -1 means quit: 2349
        2349 is not a prime, it equals 783*3.

        Please enter a natural number, -1 means quit: -12321
        Only accept natural number or -1!

        Please enter a natural number, -1 means quit: abc
        Only accept natural number or -1!

        Please enter a natural number, -1 means quit: -1
        Closed!
```

最后，为了说明在Unix和类Unix操作系统中，asyncio.create_subprocess_shell()和asyncio.create_subprocess_exec()没有本质区别，请将coroutine39.py中包含后者的语句修改为：

```
        proc1 = await asyncio.create_subprocess_shell("./coroutine38.py a1.txt
b1.txt")
        proc2 = await asyncio.create_subprocess_shell("./coroutine38.py a2.txt
b2.txt")
        proc3 = await asyncio.create_subprocess_shell("./coroutine38.py a3.txt
b3.txt")
```

并将coroutine41.py中包含后者语句修改为：

```
    proc = await asyncio.create_subprocess_shell("./coroutine40.py",
stdin=asyncio.subprocess.PIPE, stdout=asyncio.subprocess.PIPE,
stderr=asyncio.subprocess.PIPE)
```

可以验证，这两个修改后的脚本功能依然不变。

14-12. 扩展异步技术

（语言参考手册：3.2、3.4.3、3.4.4、6.2.4～6.2.9、7.6、8.9.2、8.9.3）
（标准库：内置函数、内置异常、asyncio）

至此我们已经讨论完了由asyncio模块实现的异步技术。而它们与其他技术融合在一起，扩展出了一些新的异步技术。本章的最后讨论这些扩展异步技术。

第8章讨论的上下文管理器与异步技术融合产生的扩展异步技术被称为“异步上下文管理器（asynchronous context managers）”。

异步上下文管理器就是实现了魔术属性__aenter__和__aexit__的类的实例。这两个魔术属性与__enter__和__exit__的不同在于它们被调用后会创建一个可等待对象，因此不能与with语句联合使用，而只能与async with语句联合使用。

async with语句语法为：

```
async with expression [as target], ... :
    suite
```

其中每个expression代表的表达式被求值后都得到一个异步上下文管理器，而该异步上下文管理器的__aenter__会被自动调用。__aenter__创建的协程执行完成后返回值在存在as子句的情况下被赋值给target标识符，而这些标识符同样可以在suite中被使用。当suite执行完成时，该异步上下文管理器的__aexit__会被自动调用，并被传入（None, None, None），其创建的协程执行完成后的返回值会被忽略。如果执行suite的过程中抛出了异常，则该异步上下文管理器的__aexit__也会被自动调用，并传入该异常的相关信息（与sys.exc_info()的返回值格式相同）。而创建的协程执行完成后如果返回True，则该异常不继续抛出；如果该协程返回False，则该异常被抛出到async with语句之外。

下面将context_manager.py中的例子改写成异步上下文管理器。需要强调的是，像await表达式一样，async with语句只能出现在异步函数（指协程函数或异步生成器函数，下同）的函数体内。下面的代码自定义了一个异步上下文管理器类MyAsyncContextManager，并使用该类：

```
import math
import asyncio
import sys


#自定义一个异步上下文管理器类。
class MyAsyncContextManager():
    async def __aenter__(self):
        await asyncio.sleep(1)
        def f(x, y):
            return math.sqrt(x)/y
        return f

    async def __aexit__(self, exc_type, exc_value, traceback):
        if exc_type == TypeError:
            print('Please enter two numbers!')
            return True
        elif exc_type == ValueError:
            print('x must be non-negative!')
            return True
        else:
            return False


#该协程函数通过async with语句使用自定义异步上下文管理器。
async def g(x, y):
    async with MyAsyncContextManager() as f:
        print(f(x, y))


async def main():
    await g(1, 2)
    await g('a', 'b')
    await g(-1, 2)
    await g(1, 0)
    sys.exit(0)


if __name__ == "__main__":
    asyncio.run(main())
```

请将上述代码保存为async_context_manager.py，然后通过如下命令行验证：

```
$ python3 async_context_manager.py
0.5
Please enter two numbers!
x must be non-negative!
Traceback (most recent call last):
  File "/Users/wwy/async_context_manager.py", line 40, in <module>
    asyncio.run(main())
    ^^^^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/runners.py", line 181, in run
    return runner.run(main)
           ^^^^^^^^^^^^^^^^
  File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/runners.py", line 115, in run
    return self._loop.run_until_complete(task)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
        File "/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/
asyncio/base_events.py", line 650, in run_until_complete
          return future.result()
                 ^^^^^^^^^^^^^^^
        File "/Users/wwy/async_context_manager.py", line 35, in main
          await g(1, 0)
          ^^^^^^^^^^^^
        File "/Users/wwy/async_context_manager.py", line 28, in g
          print(f(x, y))
          ^^^^^^^
        File "/Users/wwy/async_context_manager.py", line 11, in f
          return math.sqrt(x)/y
                 ~~~~~~~~~~~~^~
      ZeroDivisionError: float division by zero
```

本章前面介绍的用于实现同步原语的类中，除了Event类之外，其余5种类的实例都是异步上下文管理器。下面依次讨论它们。

将Barrier对象当成异步上下文管理器来使用时，其__aenter__在函数体内自动调用wait()。请将coroutine15.py拷贝到coroutine42.py，然后将协程函数hiker()中的如下语句：

```
      await barrier.wait()
      print(f"{name}: I cross the barrier at {time.strftime('%X')}.")
      print(f"{name}: n_waiting == {barrier.n_waiting}, broken ==
{barrier.broken}")
```

修改成：

```
      #以异步上下文管理器的方式使用屏障。
      async with barrier:
          print(f"{name}: I cross the barrier at {time.strftime('%X')}.")
          print(f"{name}: n_waiting == {barrier.n_waiting}, broken ==
{barrier.broken}")
```

可以验证两者的功能相同。

将Lock对象当成异步上下文管理器来使用时，其__aenter__在函数体内自动调用acquire()，其__aexit__在函数体内自动调用release()。请将coroutine17.py拷贝到coroutine43.py，然后将协程函数stars()从：

```
      async def stars(lock):
          try:
              while True:
                  await lock.acquire()
                  print(f"\nstars: locked == {lock.locked()}")
                  for i in range(3):
                      await asyncio.sleep(0.1)
                      sys.stdout.write("*")
                      sys.stdout.flush()
                  lock.release()
          except asyncio.CancelledError:
```

```
            lock.release()
            raise
```

修改为：

```
async def stars(lock):
    while True:
        #将锁当成异步上下文管理器来使用。
        async with lock:
            print(f"\nstars: locked == {lock.locked()}")
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
```

并将协程函数dollars()从：

```
async def dollars(lock):
    try:
        while True:
            await lock.acquire()
            print(f"\ndollars: locked == {lock.locked()}")
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
                sys.stdout.flush()
            lock.release()
    except asyncio.CancelledError:
        lock.release()
        raise
```

修改为：

```
async def dollars(lock):
    while True:
        #将锁当成异步上下文管理器来使用。
        async with lock:
            print(f"\ndollars: locked == {lock.locked()}")
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
                sys.stdout.flush()
```

可以验证两者的功能相同。注意将Lock对象当成异步上下文管理器使用能较明显地精简代码，尤其是可以省略任务被取消时的处理，因此是推荐的使用方式。

将Condition对象当成异步上下文管理器来使用时，本质上是将其当成了锁，因此同样只会在__aenter__中自动调用acquire()，在__aexit__中自动调用release()。请将coroutine18.py拷贝到coroutine44.py，然后将协程函数stars()从：

```
async def stars(cond):
    try:
        await cond.acquire()
        while True:
            await cond.wait()
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            print("")
    except asyncio.CancelledError:
        cond.release()
        raise
```

修改为：

```
async def stars(cond):
    #将监视器当成异步上下文管理器来使用。
    async with cond:
        while True:
            await cond.wait()
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            print("")
```

将协程函数dollars()从：

```
async def dollars(cond):
    try:
        await cond.acquire()
        while True:
            await cond.wait()
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
                sys.stdout.flush()
            print("")
    except asyncio.CancelledError:
        cond.release()
        raise
```

修改为：

```
async def dollars(cond):
    #将监视器当成异步上下文管理器来使用。
    async with cond:
        while True:
            await cond.wait()
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
```

```
                    sys.stdout.flush()
                print("")
```

将协程函数num_gen()从：

```
async def num_gen(lock, cond1, cond2):
    while True:
        await lock.acquire()
        n = random.randrange(100)
        if n % 2 == 0:
            cond1.notify()
        else:
            cond2.notify()
        lock.release()
        await asyncio.sleep(0)
```

修改为：

```
async def num_gen(lock, cond1, cond2):
    while True:
        #将锁当成异步上下文管理器来使用。
        async with lock:
            n = random.randrange(100)
            if n % 2 == 0:
                cond1.notify()
            else:
                cond2.notify()
            await asyncio.sleep(0)
```

可以验证两者的功能相同。

类似的，请将coroutine19.py拷贝到coroutine45.py，然后将协程函数stars()从：

```
async def stars(cond):
    try:
        while True:
            await cond.acquire()
            await cond.wait_for(lambda: even_number)
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            print("")
            cond.release()
    except asyncio.CancelledError:
        cond.release()
        raise
```

修改为：

```
async def stars(cond):
    while True:
        #将监视器当成异步上下文管理器来使用。
        async with cond:
            await cond.wait_for(lambda: even_number)
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("*")
                sys.stdout.flush()
            print("")
```

将协程函数dollars()从：

```
async def dollars(cond):
    try:
        while True:
            await cond.acquire()
            await cond.wait_for(lambda: not even_number)
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
                sys.stdout.flush()
            print("")
            cond.release()
    except asyncio.CancelledError:
        cond.release()
        raise
```

修改为：

```
async def dollars(cond):
     while True:
        #将监视器当成异步上下文管理器来使用。

        async with cond:
            await cond.wait_for(lambda: not even_number)
            for i in range(3):
                await asyncio.sleep(0.1)
                sys.stdout.write("$")
                sys.stdout.flush()
            print("")
```

将协程函数num_gen()从：

```
async def num_gen(cond):
    global even_number
    while True:
        await cond.acquire()
        n = random.randrange(100)
        if n % 2 == 0:
            even_number = True
```

```
            else:
                even_number = False
            cond.notify_all()
            cond.release()
            await asyncio.sleep(0)
```

修改为：

```
async def num_gen(cond):
    global even_number
    while True:
        #将监视器当成异步上下文管理器来使用。
        async with cond:
            n = random.randrange(100)
            if n % 2 == 0:
                even_number = True
            else:
                even_number = False
            cond.notify_all()
        await asyncio.sleep(0)
```

可以验证两者的功能相同。

从上面的例子可以看出，当将Condition对象当成异步上下文管理器使用时，不再需要调用acquire()和release()，但依然需要调用wait()或wait_for()，以及notify()或notify_all()。换句话说，异步上下文管理器仅涵盖了Condition对象功能中等价于锁的部分，而没有涵盖其功能中等价于事件的部分。

将Semaphore对象和BoundedSemaphore对象当成异步上下文管理器使用时，同样本质上是将其当成了锁。但当使用它们解决生产者-消费者问题时，需要在生产者或消费者中调用acquire()，而在相应的消费者或生产者中调用release()，并不等同于一个锁，因此不能使用async with语句。

除了同步原语，当通过asyncio.start_server()或asyncio.start_unix_server()创建一个服务器时，代表服务器的Server对象也可以被当成异步上下文管理器来使用，其__aenter__会自动调用start_serving()，而其__aexit__会自动调用close()。请将coroutine33.py拷贝到coroutine46.py，然后将协程函数main()从：

```
async def main():
    server = await asyncio.start_server(engine, host, port)
    await asyncio.sleep(60)
    server.close()
```

修改为：

```
async def main():
    server = await asyncio.start_server(engine, host, port)
    #将Server对象当成异步上下文管理器来使用。
```

```
        async with server:
            await asyncio.sleep(60)
```

可以验证两者的功能相同。

第12章讨论的迭代器与异步技术融合产生的扩展异步技术被称为“异步迭代器（asynchronous iterators）”。

异步迭代器就是实现了魔术属性__aiter__和__anext__的类的实例。__aiter__与__iter__的不同之处在于前者会创建一个异步迭代器。__anext__与__next__的区别在于前者会创建一个可等待对象，以该可等待对象的返回值作为本次迭代的结果，迭代结束时抛出不具有value属性的StopAsyncIteration异常。

异步迭代器不能与内置函数iter()和next()联合使用，而只能与内置函数aiter()和anext()联合使用。

aiter()只有一种语法，与iter()的第一种语法相对应：

```
aiter(async_iterable)
```

其async_iterable参数需被传入一个异步可迭代对象，而该对象的__aiter__会被调用，创建的异步迭代器会成为aiter()的返回值。

anext()的语法与next()对应：

```
awaitable anext(async_iterable[, default])
```

该函数会调用通过async_iterable参数传入的异步可迭代对象的__anext__，如果其创建了一个可等待对象，则以之作为返回值；否则StopAsyncIteration异常被抛出。在后一种情况下，如果通过default参数传入了一个对象，则该对象被作为返回值；否则，StopAsyncIteration异常会被抛出道anext()之外。

同理，异步迭代器不能与for语句联合使用，而只能与async for语句联合使用，后者的语法为：

```
async for targets in expr:
    suite1
else:
    suite2
```

其中expr是一个求值结果为异步可迭代对象的表达式。async for语句的内在执行逻辑是这样的：以对expr表达式求值得到的对象作为参数调用aiter()，进而获得一个异步迭代器；然后重复通过await表达式等待调用该异步迭代器的__anext__返回的可等待对象，将其返回值赋值给targets，然后执行suite1；直到StopAsyncIteration异常被抛出，自动将其截获，并通过执行suite2处理。如果else子句被省略，则StopAsyncIteration异常被抛出时async for语句直接执行完成。与async with语句类似，async for语句也只能出现在异步函数的函数体中。

下面的代码将iterate5.py中自定义的迭代器类修改为异步迭代器类，然后对其进行使用：

```
import asyncio
import sys


#自定义一个异步迭代器类。
class MyAsyncIterator:
    def __init__(self, n):
        self.max_n = int(n)
        self.n = 0

    #满足异步迭代器协议对__aiter__的要求。
    def __aiter__(self):
        return self

    #满足异步迭代器协议对__anext__的要求。
    async def __anext__(self):
        await asyncio.sleep(0.3)
        if self.n >= self.max_n:
            raise StopAsyncIteration()
        self.n += 1
        return self.n


async def main():
    #验证第一个异步可迭代对象。
    my_aiter1 = MyAsyncIterator(3)
    try:
        print("my_aiter1:")
        for i in range(4):
            print(await anext(my_aiter1))
    except StopAsyncIteration as e:
        print(repr(e))
    finally:
        print("")
    #验证第二个异步可迭代对象。
    my_aiter2 = MyAsyncIterator(2)
    print("my_aiter2:")
    for i in range(4):
        print(await anext(my_aiter2, -1))
    print("")
    #验证第三个异步可迭代对象。
    my_aiter3 = MyAsyncIterator(-10)
    print("my_aiter3:")
    for i in range(4):
        print(await anext(my_aiter3, False))
    print("")
    #验证async for语句。
    print("async for:")
```

```
        async for n in MyAsyncIterator(5):
            print(n)
        sys.exit(0)


    if __name__ == "__main__":
        asyncio.run(main())
```

请将上述代码保存为async_iterator.py，然后通过如下命令行验证：

```
$ python3 async_iterator.py
my_aiter1:
1
2
3
StopAsyncIteration()

my_aiter2:
1
2
-1
-1

my_aiter3:
False
False
False
False

async for:
1
2
3
4
5
```

第12章还介绍了生成器，它们与异步技术融合产生的扩展异步技术被称为“异步生成器（asynchronous generators）”。

可以通过调用“异步生成器函数（asynchronous generator functions）”来创建异步生成器。在生成器函数的定义语句的def关键字之前添加async关键字，就可以定义异步生成器函数。但异步生成器函数还具有两条限制：

- 函数体内只能使用yield表达式的第一种语法。
- 函数体内的return语句不能有任何返回值。

第一条限制的原因是，当迭代异步生成器时，每次迭代返回的可等待对象执行完成的结果即下一条yield表达式的返回值，使得同步操作变成了异步操作；而“yield from”并不支持异

步操作。第二条限制的原因是，异步生成器迭代完成时也抛出StopAsyncIteration异常，该异常不具有value属性，无法携带return语句的返回值。

也可以通过“异步生成器表达式（asynchronous generator expressions）”来创建异步生成器。当一个生成器表达式中的某个for关键字之前存在async关键字， 或包含await表达式时，就被视为一个异步生成器表达式。

异步生成器与生成器还有如下三个区别：

- 用asend()属性代替了send()属性。
- 用athrow()属性代替了throw()属性。
- 用aclose()属性代替了close()属性。

概括地说，它们都只是原属性的异步版本。

asend()的语法为：

```
coroutine async_generator.asend(value)
```

它创建的协程一旦被await表达式等待，就会启动下一次迭代，并将通过value参数传入的对象作为上次对yield表达式求值的结果。

athrow()的语法为：

```
coroutine async_generator.athrow(expression)
```

它创建的协程一旦被await表达式等待，就会启动下一次迭代，并立刻抛出通过expression参数传入的异常。

aclose()的语法为：

```
coroutine async_generator.aclose()
```

它创建的协程一旦被await表达式等待，就会启动下一次迭代，并立刻抛出GeneratorExit异常。

下面的代码将第12章中有关生成器函数的例子都改写为了异步生成器函数，并进行了验证：

```
import random
import asyncio
import sys


#定义第一个异步生成器函数。
async def async_gen1():
    await asyncio.sleep(0.3)
    print("A")
    yield 1
    await asyncio.sleep(0.3)
    print("B")
    yield 2
    await asyncio.sleep(0.3)
    print("C")
    yield 3
    await asyncio.sleep(0.3)
    print("D")
    return


#定义用于支持异步生成器表达式的异步迭代器类。
class MyAsyncIterator:
    def __init__(self, n):
        self.max_n = int(n)
        self.n = 0

    def __aiter__(self):
       return self

    async def __anext__(self):
        await asyncio.sleep(0.3)
        if self.n >= self.max_n:
            raise StopAsyncIteration()
        self.n += 1
        return self.n - 1


#定义第三个异步生成器函数。
async def async_gen3(start, upper_limit):
    n = start
    while n < upper_limit:
        await asyncio.sleep(0.3)
        print(n)
        n *= (yield n)


#定义第四个异步生成器函数。
async def async_gen4():
    try:
        while True:
            await asyncio.sleep(0.3)
            n = random.random()
            yield n
    except RuntimeError:
        return


#定义第五个异步生成器函数。
async def async_gen5():
    try:
        while True:
            await asyncio.sleep(0.3)
```

```
                n = random.random()
                yield n
        except Exception:
            print("Stop generates random numbers.")
            return


    async def main():
        #验证第一个异步生成器函数。
        ag1 = async_gen1()
        print("ag1:")
        print(type(ag1))
        try:
            for i in range(4):
                print(await anext(ag1))
        except StopAsyncIteration as e:
            print(repr(e))
        finally:
            print("")
        #验证异步生成器表达式。
        ag2 = ((x, y) async for x in MyAsyncIterator(2) for y in (x, 2))
        print("ag2:")
        print(type(ag2))
        async for coordinate in ag2:
            print(coordinate)
        print("")
        #验证第三个异步生成器函数。
        ag3 = async_gen3(2, 100000)
        print("ag3:")
        try:
            n = await ag3.asend(None)
            while True:
                n = await ag3.asend(n)
        except StopAsyncIteration:
            print("Bigger than upper limit!")
        finally:
            print("")
        #验证第四个异步生成器函数。
        ag4 = async_gen4()
        print("ag4:")
        for i in range(3):
            print(await ag4.asend(None))
        try:
            await ag4.athrow(RuntimeError())
        except Exception as e:
            print(repr(e))
        finally:
            print("")
        #验证第五个异步生成器函数。
        ag5 = async_gen5()
        print("ag5:")
        for i in range(2):
            print(await ag5.asend(None))
        await ag5.aclose()
        sys.exit(0)


    if __name__ == "__main__":
        asyncio.run(main())
```

请将上述代码保存为async_generator.py，然后通过如下命令行验证：

```
$ python3 async_generator.py
ag1:
<class 'async_generator'>
A
1
B
2
C
3
D
StopAsyncIteration()

ag2:
<class 'async_generator'>
(0, 0)
(0, 2)
(1, 1)
(1, 2)

ag3:
2
4
16
256
65536
Bigger than upper limit!

ag4:
0.5214924590168137
0.6206093848769274
0.638185225421343
StopAsyncIteration()

ag5:
0.9192493902309666
0.8496150941748964
```

最后，与异步生成器表达式类似，当推导式中的某个for关键字之前存在async关键字， 或包含await表达式时，就被称为“异步推导式（asynchronous comprehensions）”。

注意当推导式相互嵌套时，如果内层的推导式是异步推导式，则外层的推导式也自动变成异步推导式。异步推导式同样只能出现在异步函数的函数体内。下面的代码将第12章关于推导式的例子修改为异步推导式并验证：

```
import asyncio
import sys


#定义用于实现异步推导式的异步迭代器类。
class MyAsyncIterator:
    def __init__(self, n):
        self.max_n = int(n)
        self.n = 0
```

```
        def __aiter__(self):
            return self

        async def __anext__(self):
            await asyncio.sleep(0.3)
            if self.n >= self.max_n:
                raise StopAsyncIteration()
            self.n += 1
            return self.n


    async def main():
        #验证异步列表推导式。
        l = [x*y async for x in MyAsyncIterator(3) for y in 'ab' + 'c' if not
(x==2 and y=='b')]
        print(l)
        #验证异步集合推导式。
        st = {(x, y) async for x in MyAsyncIterator(9) if x%3 !=0 for y in
range(x, 10) if y*x > 50}
        print(st)
        #验证异步字典推导式。
        d = {x-1: (x-1)**2 async for x in MyAsyncIterator(10)}
        print(d)
        sys.exit(0)


    if __name__ == "__main__":
        asyncio.run(main())
```

请将上述代码保存为async_comprehension.py，然后通过如下命令行验证：

```
    $ python3 async_comprehension.py
    ['a', 'b', 'c', 'aa', 'cc', 'aaa', 'bbb', 'ccc']
    {(7, 9), (8, 8), (8, 9), (7, 8)}
    {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```