adam1q84 发布于 止于至善
2017年03月24日 · 12k 次阅读

Python 的异步 IO：Asyncio 简介

所谓「异步 IO」，就是你发起一个 IO 操作，却不用等它结束，你可以继续做其他事情，当它结束时，你会得到通知。

Asyncio 是并发（concurrency）的一种方式。对 Python 来说，并发还可以通过线程（threading）和多进程（multiprocessing）来实现。

Asyncio 并不能带来真正的并行（parallelism）。当然，因为 GIL（全局解释器锁）的存在，Python 的多线程也不能带来真正的并行。

可交给 asyncio 执行的任务，称为协程（coroutine）。一个协程可以放弃执行，把机会让给其它协程（即 `yield from` 或 `await`）。`

定义协程

协程的定义，需要使用 `async def` 语句。

```
async def do_some_work(x): pass
```

`do_some_work` 便是一个协程。
准确来说， `do_some_work` 是一个协程函数，可以通过 `asyncio.iscoroutinefunction` 来验证：

```
print(asyncio.iscoroutinefunction(do_some_work)) # True
```

这个协程什么都没做，我们让它睡眠几秒，以模拟实际的工作量：

```
async def do_some_work(x):
    print("Waiting " + str(x))
    await asyncio.sleep(x)
```

在解释 `await` 之前，有必要说明一下协程可以做哪些事。协程可以：

* 等待一个 future 结束
* 等待另一个协程（产生一个结果，或引发一个异常）
* 产生一个结果给正在等它的协程
* 引发一个异常给正在等它的协程

`asyncio.sleep` 也是一个协程，所以 `await asyncio.sleep(x)` 就是等待另一个协程。可参见 `asyncio.sleep` 的文档：

```
sleep(delay, result=None, *, loop=None)
Coroutine that completes after a given time (in seconds).
```

运行协程

调用协程函数，协程并不会开始运行，只是返回一个协程对象，可以通过 `asyncio.iscoroutine` 来验证：

```
print(asyncio.iscoroutine(do_some_work(3))) # True
```

此处还会引发一条警告：

```
async1.py:16: RuntimeWarning: coroutine 'do_some_work' was never awaited
  print(asyncio.iscoroutine(do_some_work(3)))
```

要让这个协程对象运行的话，有两种方式：

* 在另一个已经运行的协程中用 `await` 等待它
* 通过 `ensure_future` 函数计划它的执行

简单来说，只有 loop 运行了，协程才可能运行。
下面先拿到当前线程缺省的 loop ，然后把协程对象交给 `loop.run_until_complete` ，协程对象随后会在 loop 里得到运行。

```
loop = asyncio.get_event_loop()
loop.run_until_complete(do_some_work(3))
```

`run_until_complete` 是一个阻塞（blocking）调用，直到协程运行结束，它才返回。这一点从函数名不难看出。

`run_until_complete` 的参数是一个 future，但是我们这里传给它的却是协程对象，之所以能这样，是因为它在内部做了检查，通过 `ensure_future` 函数把协程对象包装（wrap）成了 future。所以，我们可以写得更明显一些：

```
loop.run_until_complete(asyncio.ensure_future(do_some_work(3)))
```

完整代码：

```
import asyncio

async def do_some_work(x):
    print("Waiting " + str(x))
    await asyncio.sleep(x)

loop = asyncio.get_event_loop()
loop.run_until_complete(do_some_work(3))
```

运行结果：

```
Waiting 3
<三秒钟后程序结束>
```

回调

假如协程是一个 IO 的读操作，等它读完数据后，我们希望得到通知，以便下一步数据的处理。这一需求可以通过往 future 添加回调来实现。

```
def done_callback(futu):
    print('Done')

futu = asyncio.ensure_future(do_some_work(3))
futu.add_done_callback(done_callback)

loop.run_until_complete(futu)
```

多个协程

实际项目中，往往有多个协程，同时在一个 loop 里运行。为了把多个协程交给 loop，需要借助 `asyncio.gather` 函数。

```
loop.run_until_complete(asyncio.gather(do_some_work(1), do_some_work(3)))
```

或者先把协程存在列表里：

```
coros = [do_some_work(1), do_some_work(3)]
loop.run_until_complete(asyncio.gather(*coros))
```

运行结果：

```
Waiting 3
Waiting 1
<等待三秒钟>
Done
```

这两个协程是并发运行的，所以等待的时间不是 1 + 3 = 4 秒，而是以耗时较长的那个协程为准。

参考函数 `gather` 的文档：

> gather(*coros_or_futures, loop=None, return_exceptions=False)
> Return a future aggregating results from the given coroutines or futures.

发现也可以传 futures 给它：

```
futus = [asyncio.ensure_future(do_some_work(1)),
             asyncio.ensure_future(do_some_work(3))]

loop.run_until_complete(asyncio.gather(*futus))
```

`gather` 起聚合的作用，把多个 futures 包装成单个 future，因为 `loop.run_until_complete` 只接受单个 future。

run_until_complete 和 run_forever

我们一直通过 `run_until_complete` 来运行 loop，等到 future 完成，`run_until_complete` 也就返回了。

```
async def do_some_work(x):
    print('Waiting ' + str(x))
    await asyncio.sleep(x)
    print('Done')

loop = asyncio.get_event_loop()

coro = do_some_work(3)
loop.run_until_complete(coro)
```

输出：

```
Waiting 3
<等待三秒钟>
Done
<程序退出>
```

现在改用 `run_forever`：

```
async def do_some_work(x):
    print('Waiting ' + str(x))
    await asyncio.sleep(x)
    print('Done')

loop = asyncio.get_event_loop()

coro = do_some_work(3)
asyncio.ensure_future(coro)

loop.run_forever()
```

输出：

```
Waiting 3
<等待三秒钟>
Done
<程序没有退出>
```

三秒钟过后，future 结束，但是程序并不会退出。 `run_forever` 会一直运行，直到 `stop` 被调用，但是你不能像下面这样调 `stop`：

```
loop.run_forever()
loop.stop()
```

`run_forever` 不返回， `stop` 永远也不会被调用。所以，只能在协程中调 `stop`：

```
async def do_some_work(loop, x):
    print('Waiting ' + str(x))
    await asyncio.sleep(x)
    print('Done')
    loop.stop()
```

这样并非没有问题，假如有多个协程在 loop 里运行：

```
asyncio.ensure_future(do_some_work(loop, 1))
asyncio.ensure_future(do_some_work(loop, 3))

loop.run_forever()
```

第二个协程没结束，loop 就停止了——被先结束的那个协程给停掉的。
要解决这个问题，可以用 `gather` 把多个协程合并成一个 future，并添加回调，然后在回调里再去停止 loop。

```
async def do_some_work(loop, x):
    print('Waiting ' + str(x))
    await asyncio.sleep(x)
    print('Done')

def done_callback(loop, futu):
    loop.stop()

loop = asyncio.get_event_loop()

futus = asyncio.gather(do_some_work(loop, 1), do_some_work(loop, 3))
futus.add_done_callback(functools.partial(done_callback, loop))

loop.run_forever()
```

其实这基本上就是 `run_until_complete` 的实现了， `run_until_complete` 在内部也是调用 `run_forever` 。

Close Loop?

以上示例都没有调用 `loop.close`，好像也没有什么问题。所以到底要不要调 `loop.close` 呢?

简单来说，loop 只要不关闭，就还可以再运行。：

```
loop.run_until_complete(do_some_work(loop, 1))
loop.run_until_complete(do_some_work(loop, 3))
loop.close()
```

但是如果关闭了，就不能再运行了：

```
loop.run_until_complete(do_some_work(loop, 1))
loop.close()
loop.run_until_complete(do_some_work(loop, 3))  # 此处异常
```

建议调用 `loop.close`，以彻底清理 loop 对象防止误用。

gather vs. wait

`asyncio.gather` 和 `asyncio.wait` 功能相似。

```
coros = [do_some_work(loop, 1), do_some_work(loop, 3)]
loop.run_until_complete(asyncio.wait(coros))
```

具体差别可请参见 StackOverflow 的讨论：Asyncio.gather vs asyncio.wait。

Timer

C++ Boost.Asio 提供了 IO 对象 timer，但是 Python 并没有原生支持 timer，不过可以用 `asyncio.sleep` 模拟。

```
async def timer(x, cb):
    futu = asyncio.ensure_future(asyncio.sleep(x))
    futu.add_done_callback(cb)
    await futu

t = timer(3, lambda futu: print('Done'))
loop.run_until_complete(t)
```