2017-05-19 Fri 17:23, by 田宇伟

生成器、协程和asyncio

目录

1 生成器

生成器是一个函数，它生成一个值的序列，可以在迭代中使用。

1.1 生成器表达式

语法： `(expr for item_var in iterable if cond_expr)`

```
num = (x for x in range(10000) if x * x == (x - 3) * (x + 4))
print(repr(type(num)))
print(next(num))
```

```
<class 'generator'>
12
```

1.2 定义生成器

使用 `yield` 关键字来定义生成器对象：

```
def countdown(n):
    print('Count down from %d' % n)
    while n > 0:
        yield n
        n -= 1
    return

c = countdown(10)
print(type(c))
print(c.__next__())
print(next(c))
```

```
<class 'generator'>
Count down from 10
10
9
```

调用生成器对象的 `__next__()` 方法或者使用内建函数 `next()` 时，生成器函数将开始执行语句，直到遇到 `yield` 返回， `yield` 语句在函数执行停止的地方生成一个结果，直到下次调用 `__next__()` （或 `next()` ），之后继续执行 `yield` 之后的语句。

```
for n in countdown(10):
    print(n)

a = sum(countdown(10))
print(a)
```

```
Count down from 10
10
9
8
7
6
5
4
3
2
1
Count down from 10
55
```

生成器函数完成的标志是返回或引发 StopIteration 异常。

1.3 关闭生成器对象

```
for n in countdown(10):
    if n == 2:
        break
    print(n)
```

```
c = countdown(10)
c.__next__()
c.close()     # will close the generator
c.__next__()  # StopIteration
```

对于不再使用或删除生成器时，可以调用 close() 方法。

另外，在生成器函数内部，当生成器已通过 close() 关闭时，在 yield 语句上出现 GeneratorExit 异常，可以选择捕捉这个异常，以便执行清理工作。

```
def countdown(n):
    print('Counting down from %d' % n)
    try:
        while n > 0:
            yield n
            n -= 1
    except GeneratorExit:
        print('Only made it to %d' % n)

c = countdown(5)
for num in c:
    print(num)
    if num == 3:
        c.close()
```

```
Counting down from 5
5
4
3
Only made it to 3
```

一般情况，对于生成器函数，虽然可以捕捉 GeneratorExit 异常，但是通过 yield 语句处理异常并返回另一个输出值时不合法的。另外，如果程序当前正在对生成器进行迭代，不应通过另一个执行线程或异步调用该生成器的 close() 方法。

2 协程与 yield 表达式

```
def receiver():
    print('Ready to receive')
    while 1:
        n = yield
        print('Got %s' % n)

r = receiver()
r.__next__()  # 向前执行到第一条 yield 语句
```

```
r.send(1)
r.send(2)
r.send('hello')
```

```
Ready to receive
Got 1
Got 2
Got hello
```

以这种方式使用 `yield` 语句的函数称为 **协程** 。向函数发送值时函数将执行，传递给 `send()` 的值由协程中的 `yield` 表达式返回。接收到值时，协程就会执行语句，直至遇到下一条 `yield` 语句。

不要忽略首先调用 `__next__()` 。 `__next__()` 和 `send()` 在一定意义上作用是相似的，区别是 `send()` 可以传递 `yield` 表达式的值进去，而 `__next__()` 不能传递特定的值，只能传递 `None` 进去。因此， `__next__()` 和 `send(None)` 作用是一样的。

```
def coroutine(func):
    def wrapper(*args, **kwargs):
        r = func(*args, **kwargs)
        r.__next__()
        return r
    return wrapper

@coroutine
def receiver():
    print('Ready to receive')
    while 1:
        n = yield
        print('Got %s' % n)

r = receiver()
r.send('hello, world!')  # 无需初始调用 __next__()
```

```
Ready to receive
Got hello, world!
```

协程一般会不断地执行下去，除非被显式关闭或者自己退出。

```
r.close()
r.send(4)  # StopIteration
```

跟前面生成器一样， `close()` 操作将在协程内部引发 `GeneratorExit` 异常。

```
@coroutine
def receiver():
    print('Ready to receive')
    try:
        while 1:
            n = yield
            print('Got %s' % n)
    except GeneratorExit:
        print('Receive done')


r = receiver()
r.send('hello, world')
r.close()
```

```
Ready to receive
Got hello, world
Receive done
```

可以使用 `throw(type[, value[, traceback]])` 方法在协程内部引发异常。

```
r = receiver()
r.send('hello, world')
r.throw(RuntimeError, "You're hosed!")  # RuntimeError: You're hosed!
```

协程可以选择捕捉异常并以正确的方式处理它们。使用 `throw()` 方法作为给协程的异步信号并不安全——永远都不应该通过单独的执行线程或以异步方式调用这个方法。

如果 `yield` 表达式中提供了值，协程可以使用 `yield` 语句接收和发出返回值。

```
def line_splitter(delimiter=None):
    print('Ready to split')
    result = None
    while 1:
        line = yield result
        print(line)
        result = line.split(delimiter)
        print('result: ', result)


s = line_splitter(',')
print(s.__next__())  # 返回 result 的初始值 None
print('begin splitting')
print('send: ', s.send('A,B,C'))
print('send: ', s.send('100, 200, 300'))
```

```
Ready to split
None
begin splitting
A,B,C
result:  ['A', 'B', 'C']
send:  ['A', 'B', 'C']
100, 200, 300
result:  ['100', ' 200', ' 300']
send:  ['100', ' 200', ' 300']
```

3 生成器示例

```
import os
import fnmatch
import gzip
import bz2


def find_files(topdir, pattern, recursive=False):
    if recursive:
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name, pattern):
                    yield os.path.join(path, name)
    else:
        for name in os.listdir(topdir):
            file_path = os.path.join(topdir, name)
            if os.path.isfile(file_path):
                if fnmatch.fnmatch(file_path, pattern):
                    yield file_path


def opener(filenames):
    for name in filenames:
        if name.endswith('.gz'):
            f = gzip.open(name)
        elif name.endswith('.bz2'):
            f = bz2.BZ2File(name)
        else:
            f = open(name)
        yield name, f


def cat(filelist):
    for name, f in filelist:
        for line in f:
            yield name, line


def grep(pattern, lines):
    for name, line in lines:
        if pattern in line:
            yield name, line
```

```
import os

home = os.path.expanduser('~')
bash_files = find_files(home, '*bash*')
files = opener(bash_files)
lines = cat(files)
pip_lines = grep('brew', lines)
for i, line in enumerate(pip_lines):
    if 15 < i < 21:
        print(*line, sep=': ')
```

```
/Users/fishtai0/.bash_history: brew info git

/Users/fishtai0/.bash_history: brew list

/Users/fishtai0/.bash_history: brew install libxml2

/Users/fishtai0/.bash_history: brew update

/Users/fishtai0/.bash_history: brew outdated
```

3.1 示例2

```
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
        print('Blast off!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1


from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        self._task_queue.append(task)

    def run(self):
        while self._task_queue:
            task = self._task_queue.popleft()
            try:
                next(task)  # activate generator
                self._task_queue.append(task)
            except StopIteration:
                pass


sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()
```

```
T-minus 10
T-minus 5
Counting up 0
Blast off!
T-minus 9
Blast off!
T-minus 4
Counting up 1
Blast off!
T-minus 8
Blast off!
T-minus 3
Counting up 2
Blast off!
T-minus 7
Blast off!
T-minus 2
Counting up 3
Blast off!
T-minus 6
Blast off!
T-minus 1
Counting up 4
Blast off!
T-minus 5
Blast off!
Counting up 5
Blast off!
T-minus 4
Counting up 6
Blast off!
T-minus 3
Counting up 7
Blast off!
T-minus 2
```

```
Counting up 8
Blast off!
T-minus 1
Counting up 9
Blast off!
Counting up 10
Counting up 11
Counting up 12
Counting up 13
Counting up 14
```

4 协程示例

协程可用于编写数据流处理程序。

```
import os
import fnmatch
import gzip
import bz2


@coroutine
def find_files(target, recursive=False):
    while 1:
        topdir, pattern = yield
        if recursive:
            for path, dirname, filelist in os.walk(topdir):
                for name in filelist:
                    if fnmatch.fnmatch(name, pattern):
                        target.send(os.path.join(path, name))
        else:
            for name in os.listdir(topdir):
                file_path = os.path.join(topdir, name)
                if os.path.isfile(file_path):
                    if fnmatch.fnmatch(file_path, pattern):
                        target.send(file_path)


@coroutine
def opener(target):
    while 1:
        name = yield
        if name.endswith('.gz'):
            f = gzip.open(name)
        elif name.endswith('.bz2'):
            f = bz2.BZ2File(name)
        else:
            f = open(name)
        target.send((name, f))


@coroutine
def cat(target):
    while 1:
        name, f = yield
        for line in f:
            target.send((name, line))


@coroutine
def grep(pattern, target):
    while 1:
        name, line = yield
        if pattern in line:
            target.send((name, line))

count = 0
@coroutine
def printer():
    global count
    while 1:
        name, line = yield
        if 15 < count < 21:
            print(name, line, sep=': ')
        count += 1
```

```
finder = find_files(opener(cat(grep('brew', printer()))))

home = os.path.expanduser('~')
finder.send((home, '*bash*'))
```

```
/Users/fishtai0/.bash_history: brew info git

/Users/fishtai0/.bash_history: brew list
```

```
/Users/fishtai0/.bash_history: brew install libxml2

/Users/fishtai0/.bash_history: brew update

/Users/fishtai0/.bash_history: brew outdated
```

4.1 示例2

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')

def averager():
    total = 0.0
    count = 0
    average = None
    while 1:
        term = yield
        if term is None:
            break
        total += term
        count += 1
        average = total / count
    return Result(count, average)

coro_avg = averager()
next(coro_avg)  # 激活协程
coro_avg.send(10)
coro_avg.send(30)
coro_avg.send(6.5)
coro_avg.send(None)  # StopIteration: Result(count=3, average=15.5)
```

`return` 表达式的值会偷偷传给调用方，赋值给 `StopIteration` 异常的一个属性。

获取协程返回的值：

```
coro_avg = averager()
next(coro_avg)
coro_avg.send(10)
coro_avg.send(30)
coro_avg.send(6.5)
try:
    coro_avg.send(None)
except StopIteration as exc:
    result = exc.value

print(result)
```

```
Result(count=3, average=15.5)
```

5 yield from

在生成器 `gen` 中使用 `yield from subgen()` 时， `subgen` 会获得控制权，把产出的值传给 `gen` 的调用方，即调用方可直接控制 `subgen` 。与此同时， `gen` 会阻塞，等待 `subgen` 终止。

```
def chain(*iterables):
    for it in iterables:
        for i in it:
            yield i

def chain2(*iterables):
    for it in iterables:
        yield from it

s = 'ABC'
t = tuple(range(3))
print(list(chain(s, t)))
print(list(chain2(s, t)))
```

```
['A', 'B', 'C', 0, 1, 2]
['A', 'B', 'C', 0, 1, 2]
```

`yield from` 可用于简化 `for` 循环中的 `yield` 表达式。 `yield from x` 表达式对 `x` 对象所做的第一件事是，调用 `iter(x)` ，从中获取迭代器。 `x` 可以是任何可迭代的对象。

将一个多层嵌套的序列展开称一个单层列表：

```
from collections import Iterable


def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)  # for i in flatten(x): yield i
        else:
            yield x


items = [1, 2, [3, 4, [5, 6], 7], 8, 'abc', 'defg', ['hij', 'klmn']]
for x in flatten(items):
    print(x)
```

```
1
2
3
4
5
6
7
8
abc
defg
hij
klmn
```

上面示例中， `yield from` 会返回所有子生成器的值。

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')

def averager():
    total = 0.0
    count = 0
    average = None
    while 1:
        term = yield
        if term is None:
            break
        total += term
        count += 1
        average = total / count
    return Result(count, average)


# 委派生成器
def grouper(results, key):
    while 1:
        results[key] = yield from averager()


# 客户端代码，调用方
def main(data):
    results = {}
    for key, values in data.items():
        group = grouper(results, key)
        next(group)
        for value in values:
            group.send(value)
        group.send(None)

    # print(results)
    report(results)

def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(
            result.count, group, result.average, unit))

data = {
    'girls;kg':
    [40.9, 38.5, 44.3, 42.2, 45.2, 41.7, 44.5, 38.0, 40.6, 44.5],
    'girls;m':
    [1.6, 1.51, 1.4, 1.3, 1.41, 1.39, 1.33, 1.46, 1.45, 1.43],
    'boys;kg':
    [39.0, 40.8, 43.2, 40.8, 43.1, 38.6, 41.4, 40.6, 36.3],
    'boys;m':
    [1.38, 1.5, 1.32, 1.25, 1.37, 1.48, 1.25, 1.49, 1.46],
}

main(data)
```

```
 9 boys  averaging 40.42kg
 9 boys  averaging 1.39m
10 girls averaging 42.04kg
10 girls averaging 1.43m
```

因为委派生成器相当于管道，所以可以把任意数量个委派生成器链接在一起：一个委派生成器使用 `yield from` 调用一个子生成器，而子生成器本身也是委派生成器，使用 `yield from`=调用另一个子生成器，以此类推。最终，这个链条要以一个只使用 `=yield` 表达式的简单生成器（或者任何可迭代的对象）结束。

任何 `yield from` 链条都必须由客户驱动，在最外层委派生成器上调用 `next(...)` 函数或者 `.send(...)` 方法。

6 使用协程进行并发编程

对 Python 来说，并发（concurrency）不仅可以通过多线程（threading）和多进程（multiprocessing）来实现，还可以使用 **asyncio** 来编写异步程序。

Python 3.4 标准库引入了 `asyncio` ，使得可以利用协程编写单线程并发代码，通过I/O多路复用访问套接字和其他资源。

注意： `asyncio` 并不能带来真正的并行（parallelism）。

可以将协程任务交给 asyncio 执行。一个协程可以放弃执行，把机会让给其它协程（即 `yield from` 或 `await` ）。

Python 3.5 之前协程的写法：

```python
# old_coroutine.py
import asyncio

@asyncio.coroutine
def slow_operation(n):
    yield from asyncio.sleep(1)
    print('Slow operation {} complete'.format(n))


@asyncio.coroutine
def main():
    yield from asyncio.wait([
        slow_operation(1),
        slow_operation(2),
        slow_operation(3),
    ])


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

```
Slow operation 2 complete
Slow operation 1 complete
Slow operation 3 complete
```

调用 `get_event_loop` 将返回默认的事件循环，用于负责所有协程的调度。在大量协程并发执行的过程中，除了在协程中主动使用 `await` ，当本地协程发生I/O等待时，调用 `asyncio.sleep` ，程序的控制权也会在不同的协程间切换，从而在GIL的限制下实现最大程度的并发执行，不会由于等待I/O等原因导致程序阻塞，达到较高的性能。

Python 3.5 添加了 `async` 和 `await` 这两个关键字，使得协程成为新的语法。

使用 `async` / `await` 关键词之后：

```python
# new_coroutine.py
import asyncio

async def slow_operation(n):
    await asyncio.sleep(1)
    print('Slow operation {} complete'.format(n))


async def main():
    await asyncio.wait([
        slow_operation(1),
        slow_operation(2),
        slow_operation(3),
    ])


loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

`async` 简化了 `asyncio.coroutine` ， `await` 简化了 `yield from` 。

上面的例子中， `async` 用于声明一个协程：

```
import asyncio


async def coro(wt):
    print('Waiting ' + str(wt))
    await asyncio.sleep(wt)

print(asyncio.iscoroutinefunction(coro))
```

```
True
```

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

协程可以做如下事情：

- 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

`await` 表示等待另一个协程执行完返回，获取协程执行结果，它必须在协程内才能使用。

`asyncio.sleep` 也是一个协程，所以 `await asyncio.sleep(wt)` 就是等待另一个协程。

6.1 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(coro(3)))
```

要让协程对象运行的话，有两种方式：

1. 在另一个已经运行的协程中用 `await` 等待它
2. 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。

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(coro(3))
```

```
Waiting 3
```

`run_until_complete` 是一个阻塞（blocking）调用，直到协程运行结束，它才返回。

`run_until_complete` 的参数是一个future，但是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。可以写得更明显一些：

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

完整的代码：

```
import asyncio

async def coro(wt):
    print('Waiting ' + str(wt))
    await asyncio.sleep(wt)

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

```
Waiting 3
```

7 回调

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

```
def callback(ft):
    print('Done')


ft = asyncio.ensure_future(coro(3))
ft.add_done_callback(callback)

loop.run_until_complete(ft)
```

```
Waiting 3
Done
```

8 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(coro(3), coro(2)))
```

```
Waiting 2
Waiting 3
```

或者先把协程存在列表里：

```
coros = [coro(3), coro(2)]
loop.run_until_complete(asyncio.gather(*coros))
```

```
Waiting 2
Waiting 3
```

这两个协程是并发运行的，所以等待的时间不是 3 + 2 = 5 秒，而是以耗时较长的那个协程为准。

`gather` 函数也接受 future 对象：

```
fts = [asyncio.ensure_future(coro(3)),
       asyncio.ensure_future(coro(2))]

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

```
Waiting 3
Waiting 2
```

`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。

9 run_until_complete / run_forever

通过 `run_until_complete` 来运行loop，等到future完成， `run_until_complete` 也就返回了。

改用 `run_forever` ：

```
import asyncio

async def coro(wt):
    print('Waiting ' + str(wt))
    await asyncio.sleep(wt)
    print('Done')

loop = asyncio.get_event_loop()
asyncio.ensure_future(coro(3))
```

```
loop.run_forever()
```

```
Waiting 3
Done
# 程序没有退出
```

3秒之后，future结束，但程序并不会退出。 `run_forever` 会一直运行，直到 `stop` 被调用，但是不能像下面这样调 `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 coro(loop, wt):
    print('Waiting ' + str(wt))
    await asyncio.sleep(wt)
    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(coro(loop, 3))
asyncio.ensure_future(coro(loop, 2))

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。

```
import asyncio
import functools


async def coro(loop, wt):
    print('Waiting ' + str(wt))
    await asyncio.sleep(wt)
    print('Done')

def callback(loop, ft):
    loop.stop()

loop = asyncio.get_event_loop()

ft = asyncio.gather(coro(loop, 3), coro(loop, 2))
ft.add_done_callback(functools.partial(callback, loop))

loop.run_forever()
```

```
Waiting 2
Waiting 3
Done
Done
```

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` 。