python 高级

7.1 迭代器

迭代是访问集合元素的一种方式。迭代器是一个可以记住遍历的位置的对象。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

1. 可迭代对象

我们已经知道可以对 list、tuple、str 等类型的数据使用 for...in...的循环语法从其中依次拿到数据进行使用，我们把这样的过程称为遍历，也叫**迭代**。

但是，是否所有的数据类型都可以放到 for...in...的语句中，然后让 for...in...每次从中取出一条数据供我们使用，即供我们迭代吗？

```
>>> for i in 100:
...     print(i)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
# int 整型不是 iterable，即 int 整型不是可以迭代的

# 我们自定义一个容器 MyList 用来存放数据，可以通过 add 方法向其中添加数据
>>> class MyList(object):
...     def __init__(self):
...             self.container = []
...     def add(self, item):
...             self.container.append(item)
...

按 ctrl+d 结束函数输入
>>> mylist = MyList()
>>> mylist.add(1)
>>> mylist.add(2)
>>> mylist.add(3)
>>> for num in mylist:
...     print(num)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'MyList' object is not iterable
>>>
# MyList 容器的对象也是不能迭代的
```

我们自定义了一个容器类型 MyList，在将一个存放了多个数据的 MyList 对象放到 for...in...的语句中，发现 for...in...并不能从中依次取出一条数据返回给我们，也就说我们随便封装了一个可以存放多条数据的类型却并不能被迭代使用。

我们把可以通过 for...in...这类语句迭代读取一条数据供我们使用的对象称之为可迭代对象（Iterable）**。

2. 如何判断一个对象是否可以迭代

可以使用 isinstance() 判断一个对象是否是 Iterable 对象：

```
In [50]: from collections import Iterable

In [51]: isinstance([], Iterable)
Out[51]: True

In [52]: isinstance({}, Iterable)
Out[52]: True

In [53]: isinstance('abc', Iterable)
Out[53]: True

In [54]: isinstance(mylist, Iterable)
Out[54]: False

In [55]: isinstance(100, Iterable)
Out[55]: False
```

3. 可迭代对象的本质

我们分析对可迭代对象进行迭代使用的过程，发现每迭代一次（即在 for...in...中每循环一次）都会返回对象中的下一条数据，一直向后读取数据直到迭代了所有数据后结束。那么，在这个过程中就应该有一个“人”去记录每次访问到了第几条数据，以便每次迭代都可以返回下一条数据。我们把这个能帮助我们进行数据迭代的“人”称为**迭代器(Iterator)**。

可迭代对象的本质就是可以向我们提供一个这样的中间“人”即迭代器帮助我们对其进行迭代遍历使用。

可迭代对象通过`__iter__`方法向我们提供一个迭代器，我们在迭代一个可迭代对象的时候，实际上就是先获取该对象提供的一个迭代器，然后通过这个迭代器来依次获取对象中的每一个数据.

那么也就是说，一个具备了`__iter__`方法的对象，就是一个可迭代对象。

```
>>> class MyList(object):
...     def __init__(self):
```

```
...             self.container = []
...     def add(self, item):
...             self.container.append(item)
...     def __iter__(self):
...             """返回一个迭代器"""
...             # 我们暂时忽略如何构造一个迭代器对象
...             pass
...
>>> mylist = MyList()
>>> from collections import Iterable
>>> isinstance(mylist, Iterable)
True
>>>
# 这回测试发现添加了__iter__方法的 mylist 对象已经是一个可迭代对象了
```

4. iter()函数与 next()函数

list、tuple 等都是可迭代对象，我们可以通过 iter()函数获取这些可迭代对象的迭代器。然后我们可以对获取到的迭代器不断使用 next()函数来获取下一条数据。
iter()函数实际上就是调用了可迭代对象的__iter__方法。

```
>>> li = [11, 22, 33, 44, 55]
>>> li_iter = iter(li)
>>> next(li_iter)
11
>>> next(li_iter)
22
>>> next(li_iter)
33
>>> next(li_iter)
44
>>> next(li_iter)
55
>>> next(li_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

注意，当我们已经迭代完最后一个数据之后，再次调用 next()函数会抛出 StopIteration 的异常，来告诉我们所有数据都已迭代完成，不用再执行 next()函数了。

5. 如何判断一个对象是否是迭代器

可以使用 isinstance() 判断一个对象是否是 Iterator 对象：

```
In [56]: from collections import Iterator

In [57]: isinstance([], Iterator)
Out[57]: False

In [58]: isinstance(iter([]), Iterator)
Out[58]: True

In [59]: isinstance(iter("abc"), Iterator)
Out[59]: True
```

6. 迭代器 Iterator

通过上面的分析，我们已经知道，迭代器是用来帮助我们记录每次迭代访问到的位置，当我们对迭代器使用 next()函数的时候，迭代器会向我们返回它所记录位置的下一个位置的数据。实际上，在使用 next()函数的时候，调用的就是迭代器对象的__next__方法（Python3 中是对象的__next__方法）。**所以，我们要想构造一个迭代器，就要实现它的__next__方法**。但这还不够，python 要求迭代器本身也是可迭代的，所以我们还要为迭代器实现__iter__方法，而__iter__方法要返回一个迭代器，迭代器自身正是一个迭代器，所以迭代器的__iter__方法返回自身即可。

一个实现了__iter__方法和__next__方法的对象，就是迭代器。

```
#coding=utf-8
from collections.abc import Iterator
class MyList(object):
    """自定义的一个可迭代对象"""
    def __init__(self):
        self.items = []

    def add(self, val):
        self.items.append(val)

    def __iter__(self):
        myiterator = MyIterator(self)
        return myiterator


class MyIterator(object):
    """自定义的供上面可迭代对象使用的一个迭代器"""
    def __init__(self, mylist):
        self.mylist = mylist
        # current 用来记录当前访问到的位置
      self.current = 0
```

```
    def __next__(self):
        if self.current < len(self.mylist.items):
            item = self.mylist.items[self.current]
            self.current += 1
            return item
        else:
            raise StopIteration

    def __iter__(self):
        return self


if __name__ == '__main__':
    mylist = MyList()
    print(isinstance(mylist, Iterator))
    mylist.add(1)
    mylist.add(2)
    mylist.add(3)
    mylist.add(4)
    mylist.add(5)
    # print(next(mylist)) #为啥不行
    for num in mylist:
        print(num)
    myIterator = MyIterator(mylist)
    print(isinstance(myIterator, Iterator))
    print(next(myIterator)) #打印结果是多少
```

7. for...in...循环的本质

for **item** in **Iterable** 循环的本质就是先通过 iter()函数获取可迭代对象 Iterable 的迭代器，然后对获取到的迭代器不断调用 next()方法来获取下一个值并将其赋值给 item，当遇到 StopIteration 的异常后循环结束。

8. 迭代器的应用场景

我们发现迭代器最核心的功能就是可以通过 next()函数的调用来返回下一个数据值。如果每次返回的数据值不是在一个已有的数据集合中读取的，而是通过程序按照一定的规律计算生成的，那么也就意味着可以不用再依赖一个已有的数据集合，也就是说不用再将所有要迭代的数据都一次性缓存下来供后续依次读取，这样可以节省大量的存储（内存）空间。

举个例子，比如，数学中有个著名的斐波拉契数列（Fibonacci），数列中第一个数为 0，第二个数为 1，其后的每一个数都可由前两个数相加得到：

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

现在我们想要通过 for...in...循环来遍历迭代斐波那契数列中的前 n 个数。那么这个斐波那契数列我们就可以用迭代器来实现，每次迭代都通过数学计算来生成下一个数。

```
class FibIterator(object):
    """斐波那契数列迭代器"""
    def __init__(self, n):
        """
        :param n: int, 指明生成数列的前 n 个数
        """
        self.n = n
        # current 用来保存当前生成到数列中的第几个数了
        self.current = 0
        # num1 用来保存前前一个数，初始值为数列中的第一个数 0
        self.num1 = 0
        # num2 用来保存前一个数，初始值为数列中的第二个数 1
        self.num2 = 1

    def __next__(self):
        """被 next()函数调用来获取下一个数"""
        if self.current < self.n:
            num = self.num1
            self.num1, self.num2 = self.num2, self.num1+self.num2
            self.current += 1
            return num
        else:
            raise StopIteration

    def __iter__(self):
        """迭代器的__iter__返回自身即可"""
        return self


if __name__ == '__main__':
    fib = FibIterator(10)
    for num in fib:
        print(num, end=" ")
```

9. 并不是只有 for 循环能接收可迭代对象

除了 for 循环能接收可迭代对象，list、tuple 等也能接收。

```
li = list(FibIterator(15))
print(li)
tp = tuple(FibIterator(6))
print(tp)
```

7.2 生成器

1. 生成器

利用迭代器，我们可以在每次迭代获取数据（通过 next()方法）时按照特定的规律进行生成。但是我们在实现一个迭代器时，**关于当前迭代到的状态需要我们自己记录，进而才能根据当前状态生成下一个数据**。为了达到记录当前状态，并配合 next()函数进行迭代使用，我们可以采用更简便的语法，即**生成器(generator)。生成器是一类特殊的迭代器**。

2. 创建生成器方法 1

要创建一个生成器，有很多种方法。**第一种方法很简单，只要把一个列表生成式的 [ ] 改成 ( )**

```
In [15]: L = [ x*2 for x in range(5)]

In [16]: L
Out[16]: [0, 2, 4, 6, 8]

In [17]: G = ( x*2 for x in range(5))

In [18]: G
Out[18]: <generator object <genexpr> at 0x7f626c132db0>

In [19]:
```

创建 L 和 G 的区别仅在于最外层的 [ ] 和 ( )， L 是一个列表，而 G 是一个生成器。我们可以直接打印出列表 L 的每一个元素，而对于生成器 G，我们可以**按照迭代器的使用方法来使用**，即可以通过 next()函数、for 循环、list()等方法使用。

```
In [19]: next(G)
Out[19]: 0

In [20]: next(G)
Out[20]: 2

In [21]: next(G)
Out[21]: 4

In [22]: next(G)
Out[22]: 6

In [23]: next(G)
Out[23]: 8
```

```
In [24]: next(G)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-24-380e167d6934> in <module>()
----> 1 next(G)

StopIteration:

In [25]:

In [26]: G = ( x*2 for x in range(5))

In [27]: for x in G:
   ....:     print(x)
   ....:
0
2
4
6
8

In [28]:
```

3. 创建生成器方法 2

generator 非常强大。如果推算的算法比较复杂，用类似列表生成式的 for 循环无法实现的时候，**还可以用函数来实现**。

我们仍然用上一节提到的斐波那契数列来举例，回想我们在上一节用迭代器的实现方式：

```
class FibIterator(object):
    """斐波那契数列迭代器"""
    def __init__(self, n):
        """
        :param n: int, 指明生成数列的前 n 个数
        """
        self.n = n
        # current 用来保存当前生成到数列中的第几个数了
        self.current = 0
        # num1 用来保存前前一个数，初始值为数列中的第一个数 0
        self.num1 = 0
        # num2 用来保存前一个数，初始值为数列中的第二个数 1
        self.num2 = 1

    def __next__(self):
```

```
        """被 next()函数调用来获取下一个数"""
        if self.current < self.n:
            num = self.num1
            self.num1, self.num2 = self.num2, self.num1+self.num2
            self.current += 1
            return num
        else:
            raise StopIteration

    def __iter__(self):
        """迭代器的__iter__返回自身即可"""
        return self
```

注意，在用迭代器实现的方式中，我们要借助几个变量(n、current、num1、num2)来保存迭代的状态。现在我们用生成器来实现一下。

```
In [30]: def fib(n):
   ....:     current = 0
   ....:     num1, num2 = 0, 1
   ....:     while current < n:
   ....:         num = num1
   ....:         num1, num2 = num2, num1+num2
   ....:         current += 1
   ....:         yield num
   ....:     return 'done'
   ....:

In [31]: F = fib(5)

In [32]: next(F)
Out[32]: 1

In [33]: next(F)
Out[33]: 1

In [34]: next(F)
Out[34]: 2

In [35]: next(F)
Out[35]: 3

In [36]: next(F)
Out[36]: 5

In [37]: next(F)
---------------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
```

```
ast)
<ipython-input-37-8c2b02b4361a> in <module>()
----> 1 next(F)

StopIteration: done
```

在使用生成器实现的方式中，我们将原本在迭代器__next__方法中实现的基本逻辑放到一个函数中来实现，但是将每次迭代返回数值的 return 换成了 yield，此时新定义的函数便不再是函数，而是一个**生成器**了。**简单来说：只要在 def 中有 yield 关键字的 就称为 生成器**

此时按照调用函数的方式(案例中为 F = fib(5))使用生成器就不再是执行函数体了，而是会返回一个生成器对象（ 案例中为 F ），**然后就可以按照使用迭代器的方式来使用生成器了。**

```
In [38]: for n in fib(5):
   ....:     print(n)
   ....:
1
1
2
3
5

In [39]:
```

但是用 for 循环调用 generator 时，发现拿不到 generator 的 return 语句的返回值。如果想要拿到返回值，必须捕获 StopIteration 错误，返回值包含在 StopIteration 的 value 中：

```
In [39]: g = fib(5)

In [40]: while True:
   ....:     try:
   ....:         x = next(g)
   ....:         print("value:%d"%x)
   ....:     except StopIteration as e:
   ....:         print("生成器返回值:%s"%e.value)
   ....:         break
   ....:
value:1
value:1
value:2
value:3
value:5
生成器返回值:done

In [41]:
```

总结

- **使用了 yield 关键字的函数不再是函数，而是生成器。（使用了 yield 的函数就是生成器）**
- **yield 关键字有两点作用：**
  - **保存当前运行状态（断点），然后暂停执行，即将生成器（函数）挂起**
  - **将 yield 关键字后面表达式的值作为返回值返回，此时可以理解为起到了 return 的作用**
- **可以使用 next()函数让生成器从断点处继续执行，即唤醒生成器（函数）**
- Python3 中的生成器可以使用 return 返回最终运行的返回值。

4. 使用 send 唤醒

我们除了可以使用 next()函数来唤醒生成器继续执行外，还可以使用 send()函数来唤醒执行。使用 send()函数的一个好处是可以在唤醒的同时向断点处传入一个附加数据。

例子：执行到 yield 时，gen 函数作用暂时保存，返回 i 的值; temp 接收下次 c.send("python")，send 发送过来的值，c.next()等价 c.send(None)

```
In [10]: def gen():
    ....:     i = 0
    ....:     while i<5:
    ....:         temp = yield i
    ....:         print(temp)
    ....:         i+=1
    ....:
```

使用 send

```
In [43]: f = gen()

In [44]: next(f)
Out[44]: 0

In [45]: f.send('haha')
haha
Out[45]: 1

In [46]: next(f)
None
Out[46]: 2

In [47]: f.send('haha')
haha
Out[47]: 3
```

```
In [48]:
```

使用 next 函数

```
In [11]: f = gen()

In [12]: next(f)
Out[12]: 0

In [13]: next(f)
None
Out[13]: 1

In [14]: next(f)
None
Out[14]: 2

In [15]: next(f)
None
Out[15]: 3

In [16]: next(f)
None
Out[16]: 4

In [17]: next(f)
None
---------------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
ast)
<ipython-input-17-468f0afdf1b9> in <module>()
----> 1 next(f)

StopIteration:
```

使用__next__()方法（不常使用）

```
In [18]: f = gen()

In [19]: f.__next__()
Out[19]: 0

In [20]: f.__next__()
None
Out[20]: 1

In [21]: f.__next__()
None
```

```
Out[21]: 2

In [22]: f.__next__()
None
Out[22]: 3

In [23]: f.__next__()
None
Out[23]: 4

In [24]: f.__next__()
None
---------------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
ast)
<ipython-input-24-39ec527346a9> in <module>()
----> 1 f.__next__()

StopIteration:
```

7.3 协程

协程，又称微线程，纤程。英文名 Coroutine。

协程是啥

协程是 python 个中另外一种实现多任务的方式，只不过比线程更小占用更小执行单元（理解为需要的资源）。 为啥说它是一个执行单元，因为它自带 CPU 上下文。这样只要在合适的时机， 我们可以把一个协程 切换到另一个协程。 只要这个过程中保存或恢复 **CPU 上下文**那么程序还是可以运行的。

通俗的理解：在一个线程中的某个函数，可以在任何地方保存当前函数的一些临时变量等信息，然后切换到另外一个函数中执行，注意不是通过调用函数的方式做到的，并且切换的次数以及什么时候再切换到原来的函数都由开发者自己确定

协程和线程差异

在实现多任务时, **线程切换从系统层面远不止保存和恢复 CPU 上下文这么简单**。操作系统为了程序运行的高效性每个线程都有**自己缓存 Cache** 等等数据，操作系统还会帮你做这些数据的恢复操作。 所以线程的切换非常耗性能。但是协程的切换只是单纯的操作 CPU 的上下文，**所以一秒钟切换个上百万次系统都抗的住**。

简单实现协程

```
import time

def work1():
    while True:
        print("----work1---")
        yield
        time.sleep(0.5)

def work2():
    while True:
        print("----work2---")
        yield
        time.sleep(0.5)

def main():
    w1 = work1()
    w2 = work2()
    while True:
        next(w1)
        next(w2)

if __name__ == "__main__":
    main()
```

运行结果：

```
----work1---
----work2---
----work1---
----work2---
----work1---
----work2---
----work1---
----work2---
----work1---
----work2---
----work1---
----work2---
...省略...
```

配置 pip.ini 文件

https://blog.csdn.net/sunyllove/article/details/81627281

C:\Users\Administrator\AppData\Roaming\pip

7.4 greenlet

为了更好使用协程来完成多任务，python 中的 greenlet 模块对其封装，从而使得切换任务变的更加简单

安装方式

windows 下安装

```
pip install greenlet
```

Ubuntu 下使用如下命令安装 greenlet 模块:

```
pip3 install greenlet
```

```
#coding=utf-8

from greenlet import greenlet
import time

def test1():
    while True:
        print("---A--")
        gr2.switch()
        time.sleep(0.5)

def test2():
    while True:
        print("---B--")
        gr1.switch()
        time.sleep(0.5)

gr1 = greenlet(test1)
gr2 = greenlet(test2)

#切换到 gr1 中运行
gr1.switch()
```

运行效果

```
---A--
---B--
---A--
---B--
```

```
---A--
---B--
---A--
---B--
...省略...
```

7.5 gevent

greenlet 已经实现了协程，但是这个还的人工切换，是不是觉得太麻烦了，不要捉急，python 还有一个比 greenlet 更强大的并且能够自动切换任务的模块 gevent

其原理是当一个 greenlet 遇到 IO(指的是 input output 输入输出，比如网络、文件操作等)操作时，比如访问网络，就自动切换到其他的 greenlet，等到 IO 操作完成，再在适当的时候切换回来继续执行。

由于 IO 操作非常耗时，经常使程序处于等待状态，有了 gevent 为我们自动切换协程，就保证总有协程在运行，而不是等待 IO

安装

在 Linux

pip3 install gevent

在 Windows

在 Windows 的 Pycharm 输入了 import greenlet 后，直接点击红色小灯泡进行安装即可

pip install gevent

1 gevent 的使用方法

gevent.spawn 接口使用方法，gevent.spawn(函数名，传参)

```
def f(n):
    for i in range(n):
        print(gevent.getcurrent(), i)

g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
g1.join()
g2.join()
g3.join()
```

运行结果

```
<Greenlet at 0x10e49f550: f(5)> 0
<Greenlet at 0x10e49f550: f(5)> 1
```

```
<Greenlet at 0x10e49f550: f(5)> 2
<Greenlet at 0x10e49f550: f(5)> 3
<Greenlet at 0x10e49f550: f(5)> 4
<Greenlet at 0x10e49f910: f(5)> 0
<Greenlet at 0x10e49f910: f(5)> 1
<Greenlet at 0x10e49f910: f(5)> 2
<Greenlet at 0x10e49f910: f(5)> 3
<Greenlet at 0x10e49f910: f(5)> 4
<Greenlet at 0x10e49f4b0: f(5)> 0
<Greenlet at 0x10e49f4b0: f(5)> 1
<Greenlet at 0x10e49f4b0: f(5)> 2
<Greenlet at 0x10e49f4b0: f(5)> 3
<Greenlet at 0x10e49f4b0: f(5)> 4
```

可以看到，3 个 greenlet 是依次运行而不是交替运行

2. gevent 切换执行

```
import gevent

def f(n):
    for i in range(n):
        print(gevent.getcurrent(), i)
        #用来模拟一个耗时操作，注意不是 time 模块中的 sleep
        gevent.sleep(1)

g1 = gevent.spawn(f, 5)
g2 = gevent.spawn(f, 5)
g3 = gevent.spawn(f, 5)
g1.join()
g2.join()
g3.join()
```

运行结果

```
<Greenlet at 0x7fa70ffa1c30: f(5)> 0
<Greenlet at 0x7fa70ffa1870: f(5)> 0
<Greenlet at 0x7fa70ffa1eb0: f(5)> 0
<Greenlet at 0x7fa70ffa1c30: f(5)> 1
<Greenlet at 0x7fa70ffa1870: f(5)> 1
<Greenlet at 0x7fa70ffa1eb0: f(5)> 1
<Greenlet at 0x7fa70ffa1c30: f(5)> 2
<Greenlet at 0x7fa70ffa1870: f(5)> 2
<Greenlet at 0x7fa70ffa1eb0: f(5)> 2
<Greenlet at 0x7fa70ffa1c30: f(5)> 3
<Greenlet at 0x7fa70ffa1870: f(5)> 3
<Greenlet at 0x7fa70ffa1eb0: f(5)> 3
<Greenlet at 0x7fa70ffa1c30: f(5)> 4
<Greenlet at 0x7fa70ffa1870: f(5)> 4
<Greenlet at 0x7fa70ffa1eb0: f(5)> 4
```

3. 给程序打补丁

```
from gevent import monkey
import gevent
import random
import time

def coroutine_work(coroutine_name):
    for i in range(10):
        print(coroutine_name, i)
        time.sleep(random.random())

gevent.joinall([
        gevent.spawn(coroutine_work, "work1"),
        gevent.spawn(coroutine_work, "work2")
])
```

运行结果

```
work1 0
work1 1
work1 2
work1 3
work1 4
work1 5
work1 6
work1 7
work1 8
work1 9
work2 0
work2 1
work2 2
work2 3
work2 4
work2 5
work2 6
work2 7
work2 8
work2 9
```

```
from gevent import monkey
import gevent
import random
import time

# 有耗时操作时需要
monkey.patch_all()  # 将程序中用到的耗时操作的代码，换为 gevent 中自己实现的模块
```

```
def coroutine_work(coroutine_name):
    for i in range(10):
        print(coroutine_name, i)
        time.sleep(random.random())

gevent.joinall([
        gevent.spawn(coroutine_work, "work1"),
        gevent.spawn(coroutine_work, "work2")
])
```

运行结果

```
work1 0
work2 0
work1 1
work1 2
work1 3
work2 1
work1 4
work2 2
work1 5
work2 3
work1 6
work1 7
work1 8
work2 4
work2 5
work1 9
work2 6
work2 7
work2 8
work2 9
```

猴子补丁作用

monkey patch 指的是在执行时动态替换,通常是在 startup 的时候.
用过 gevent 就会知道,会在最开头的地方 gevent.monkey.patch_all();
把标准库中的 thread/socket 等给替换掉.这样我们在后面使用 **socket 的时候能够跟寻常一样使用,无需改动不论什么代码,可是它变成非堵塞的了.**
之前做的一个游戏 server,非常多地方用的 import json,后来发现 ujson 比自带 json 快了 N 倍,于是问题来了,难道几十个文件要一个个把 import json 改成 import ujson as json 吗?
事实上仅仅须要在进程 startup 的地方 monkey patch 即可了.是影响整个进程空间的.

直接参考以下实例，采用协程访问三个网站

由于 IO 操作非常耗时，程序经常会处于等待状态

比如请求多个网页有时候需要等待，gevent 可以自动切换协程

遇到阻塞自动切换协程，程序启动时执行 monkey.patch_all()解决

7.6 进程、线程、协程对比

1 请仔细理解如下的通俗描述

- 有一个老板想要开个工厂进行生产某件商品（例如剪子）
- 他需要花一些财力物力制作一条生产线，这个生产线上有很多的器件以及材料这些所有的 为了能够生产剪子而准备的资源称之为：进程
- 只有生产线是不能够进行生产的，所以老板的找个工人来进行生产，这个工人能够利用这些材料最终一步步的将剪子做出来，这个来做事情的工人称之为：线程
- 这个老板为了提高生产率，想到 3 种办法：
    a. 在这条生产线上多招些工人，一起来做剪子，这样效率是成倍增长，即单进程 多线程方式
    b. 老板发现这条生产线上的工人不是越多越好，因为一条生产线的资源以及材料毕竟有限，所以老板又花了些财力物力购置了另外一条生产线，然后再招些工人这样效率又再一步提高了，即多进程 多线程方式
    c. 老板发现，现在已经有了很多条生产线，并且每条生产线上已经有很多工人了（即程序是多进程的，每个进程中又有多个线程），为了再次提高效率，老板想了个损招，规定：如果某个员工在上班时临时没事或者再等待某些条件（比如等待另一个工人生产完谋道工序 之后他才能再次工作），那么这个员工就利用这个时间去做其它的事情，那么也就是说：如果一个线程等待某些条件，可以充分利用这个时间去做其它事情，其实这就是：协程方式

简单总结

1. 进程是资源分配的单位
2. 线程是操作系统内核调度的基本单位
3. 进程切换需要的资源很最大，效率很低
4. 线程切换需要的资源一般，效率一般（当然了在不考虑 GIL 的情况下）
5. 协程切换任务资源很小，效率高

6. 多进程、多线程根据 cpu 核数不一样可能是并行的，但是协程是在一个线程中， 所以是并发

gevent 常用方法：

gevent.spawn()	创建一个普通的 Greenlet 对象并切换
gevent.spawn_later(seconds=3)	延时创建一个普通的 Greenlet 对象并切换
gevent.spawn_raw()	创建的协程对象属于一个组
gevent.getcurrent()	返回当前正在执行的 greenlet
gevent.joinall(jobs)	将协程任务添加到事件循环，接收一个任务列表
gevent.wait()	可以替代 join 函数等待循环结束，也可以传入协程对象列表
gevent.kill()	杀死一个协程
gevent.killall()	杀死一个协程列表里的所有协程
monkey.patch_all()	非常重要，会自动将 python 的一些标准模块替换成 gevent 框架

https://blog.csdn.net/biheyu828/article/details/86593413

python 最新接口
file:///D:/7.Python%E8%AF%BE%E4%BB%B6/2.Python%E5%9F%BA%E7%A1%80-1%E5%91%A8%E5%8A%A02%E5%A4%A9/python-3.6.10-docs-html/library/asyncio-task.html?highlight=event

https://blog.csdn.net/weixin_42146296/article/details/92166245

7.7 并发下载器

并发下载原理

```
from gevent import monkey
import gevent
```

```
import urllib.request

# 有耗时操作时需要
monkey.patch_all()

def my_downLoad(url):
    print('GET: %s' % url)
    resp = urllib.request.urlopen(url)
    data = resp.read()
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(my_downLoad, 'http://www.baidu.com/'),
        gevent.spawn(my_downLoad, 'http://www.cskaoyan.com/'),
        gevent.spawn(my_downLoad, 'http://www.duolaima.com/'),
])
```

运行结果

```
GET: http://www.baidu.com/

GET: http://www.cskaoyan.com/

GET: http://www.duolaima.com/

52562 bytes received from http://www.cskaoyan.com/.

153759 bytes received from http://www.baidu.com/.

12605 bytes received from http://www.duolaima.com/.
```

从上能够看到是先发送的获取王道论坛的相关信息，然后依次是百度、多来吗，但是收到数据的先后顺序不一定与发送顺序相同，这也就体现出了异步，即不确定什么时候会收到数据，顺序不一定

实现多个图片，音乐，电影下载

```
from gevent import monkey
import gevent
import urllib.request

#有 IO 才做时需要这一句
monkey.patch_all()

def my_downLoad(file_name, url):
    print('GET: %s' % url)
    resp = urllib.request.urlopen(url)
    data = resp.read()

    with open(file_name, "wb") as f:
```

```
        f.write(data)

    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(my_downLoad,
"7a082c0dde36eac2205a088397aaf295.jpg",
'http://qzs.qq.com/qzone/v6/v6_config/upload/7a082c0dde36ea
c2205a088397aaf295.jpg'),
        gevent.spawn(my_downLoad, "da8e974dc_is.jpg",
'https://pic1.zhimg.com/da8e974dc_is.jpg'),
])
```

上面的 url 可以换为自己需要下载视频、音乐、图片等网址