**[yield与send实现协程操作](https://www.cnblogs.com/vipchenwei/p/7049404.html)**

**yield与send实现协程操作**

之前我们说过，在函数内部含有yield语句即称为生成器。

下面，我们来看看在函数内部含有yield语句达到的效果。首先，我们来看看以下代码：

[复制代码](javascript:void(0);)

def foo():

while True:

x = yield

print("value:",x)

g = foo() # g是一个生成器

next(g) # 程序运行到yield就停住了,等待下一个next

g.send(1) # 我们给yield发送值1,然后这个值被赋值给了x，并且打印出来,然后继续下一次循环停在yield处

g.send(2) # 同上

next(g) # 没有给x赋值，执行print语句，打印出None,继续循环停在yield处

[复制代码](javascript:void(0);)

我们都知道，程序一旦执行到yield就会停在该处,并且将其返回值进行返回。上面的例子中，我们并没有设置返回值，所有默认程序返回的是None。我们通过打印语句来查看一下第一次next的返回值：

print(next(g))

####输出结果#####

None

　　正如我们所说的，程序返回None。接着程序往下执行，但是并没有看到next()方法。为什么还会继续执行yield语句后面的代码呢？这是因为，send()方法具有两种功能：第一，传值，send()方法，将其携带的值传递给yield，注意，是传递给yield，而不是x,然后再将其赋值给x；第二，send()方法具有和next()方法一样的功能，也就是说，传值完毕后，会接着上次执行的结果继续执行，知道遇到yield停止。这也就为什么在调用g.send()方法后，还会打印出x的数值。有了上面的分析，我们可以很快知道，执行了send(1)后，函数被停止在了yield处，等待下一个next()的到来。程序往下执行，有遇到了send(2)，其执行流程与send(1)完全一样。

　　有了上述的分析，我们可以总结出send()的两个功能：1.传值；2.next()。

　　既然send()方法有和next一样的作用，那么我们可不可以这样做：

[复制代码](javascript:void(0);)

def foo():

while True:

x = yield

print("value:",x)

g = foo()

g.send(1) #执行给yield传值,这样行不行呢?

[复制代码](javascript:void(0);)

　　很明显，是不行的。

TypeError: can't send non-None value to a just-started generator

　　错误提示:不能传递一个非空值给一个未启动的生成器。

　　也就是说，在一个生成器函数未启动之前，是不能传递数值进去。必须先传递一个None进去或者调用一次next(g)方法，才能进行传值操作。至于为什么要先传递一个None进去，可以看一下官方说法。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　问题就来，既然在给yield传值过程中，会调用next()方法，那么是不是在调用一次函数的时候，是不是每次都要给它传递一个空值？有没有什么简便方法来解决這个问题呢？答案，装饰器！！看下面代码:

[复制代码](javascript:void(0);)

def deco(func): # 装饰器:用来开启协程

def wrapper():

res = func()

next(res)

return res # 返回一个已经执行了next()方法的函数对象

return wrapper

@deco

def foo():

pass

[复制代码](javascript:void(0);)

　　上面我yield是没有返回值的，下面我们看看有返回值的生成器函数。

[复制代码](javascript:void(0);)

def deco(func):

def wrapper():

res = func()

next(res)

return res

return wrapper

@deco

def foo():

food\_list = []

while True:

food = yield food\_list #返回添加food的列表

food\_list.append(food)

print("elements in foodlist are:",food)

g = foo()

print(g.send('苹果'))

print(g.send('香蕉'))

print(g.send('菠萝'))

###########输出结果为######

elements in foodlist are: 苹果

['苹果']

elements in foodlist are: 香蕉

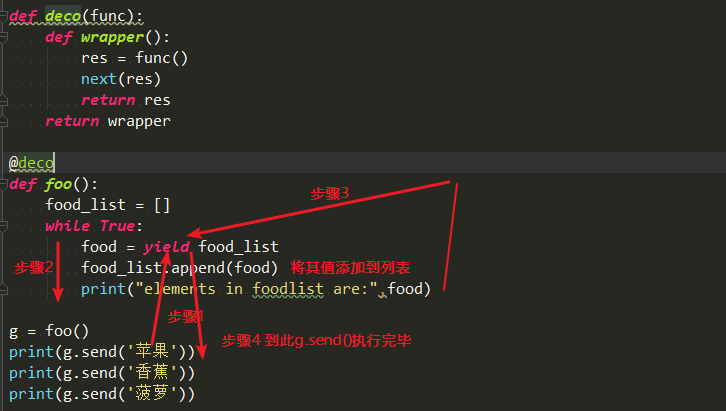
['苹果', '香蕉']

elements in foodlist are: 菠萝

['苹果', '香蕉', '菠萝']

[复制代码](javascript:void(0);)

　　分析：首先，我们要清楚，在函数执行之前，已经执行了一次next()(装饰器的功能)，程序停止yield。接着程序往下执行，遇到g.send()，然后将其值传递给food，然后再将获得的food添加到列表food\_list中。打印出food，再次循环程序停在yield。程序继续执行，又遇到g.send()，其过程与上面是一模一样的。看看以下的程序执行流程，你可能会更清楚。



　　这里我们要明确一点，yield的返回值和传给yield的值是两码事！！

　　yiedl的返回值就相当于return的返回值，这个值是要被传递出去的,而send()传递的值，是要被yield接受，供函数内部使用的的，明确这一点很重要的。那么上面的打印，就应该打印出yield的返回值，而传递进去的值则本保存在一个列表中。

三、实例

[复制代码](javascript:void(0);)

"""模拟:grep -rl 'root' /etc"""

import os

def deco(func): # 用来开启协程

def wrapper(\*args,\*\*kwargs):

res = func(\*args,\*\*kwargs)

next(res) # res.seng(None)

return res

return wrapper

@deco

def search(target):

while True:

PATH = yield

g = os.walk(PATH) # 获取PATH目录下的文件，文件夹

for par\_dir, \_, files in g: #迭代解包,取出当前目录路径和文件名

for file in files:

file\_path = r'%s\%s' %(par\_dir,file) # 拼接文件的绝对路径

target.send(file\_path) # 给下一个

@deco

def opener(target, pattern=None):

while True:

file\_path = yield

with open(file\_path, encoding='utf-8') as f:

target.send((file\_path, f)) # 将文件路径和文件对象一起传递给下一个函数的yield，因为在打印路径时候，需要打印出文件路径，只有从这里传递下去

@deco

def cat(target):

while True:

filepath, f = yield # 这里接收opener传递进来的路径和文件对象

for line in f:

tag = target.send((filepath, line)) # 同样，也要传递文件路径,并且获取下一个函数grep的返回值，从而判断该文件是否重复读取了

if tag: # 如果为真，说明该文件读取过了，则执行退出循环

break

@deco

def grep(target, pattern):

tag = False

while True:

filepath, line = yield tag # 接受两个值，并且设置返回值，這个返回值要传递给发送消息的send()，也就是cat()函数send

tag = False

if pattern in line: # 如果待匹配字符串在该行

target.send(filepath) # 把文件路径传递给priter

tag = True # 设置tag

@deco

def printer():

while True:

filename = yield

print(filename)

[复制代码](javascript:void(0);)

调用方式:

PATH1 = r'D:\CODE\_FILE\python\test'

search(opener(cat(grep(printer(), 'root')))).send(PATH1)

输出结果:

######找出了含有'root'的所有文件#######

D:\CODE\_FILE\python\test\a.txt

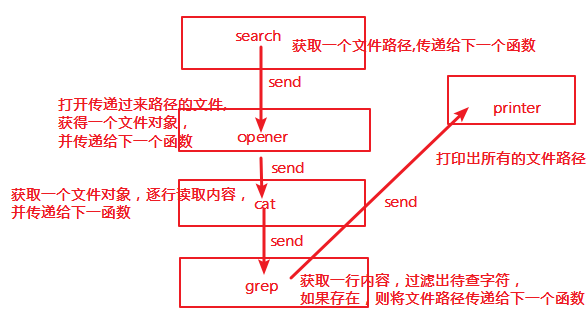
D:\CODE\_FILE\python\test\test1\c.txt

D:\CODE\_FILE\python\test\test1\test2\d.txt

程序分析:

 　　有了上面的基础，我们来分析一下上述程序的执行。

　　每一个函数之前都有一个@deco装饰器，这个装饰器用于开启协程。首先我们定义了一个search()，其内部有关键字yield，则search()是一个生成器。也就是说，我们可以通过send()给其传递一个值进去。search()函数的功能 是：获取一个文件的绝对路径，并将這个绝对路径通过send()方法，在传递给下个含有yield的函数，也就是下面的opener函数。opener的yield接受了search()传递进来的路径，并将其赋值给了file\_path，然后我们根据這个路径，打开了一个文件，获取了一个文件对象f。然后我们在将這个文件对象send()给cat()函数，這个函数功能是读取文件中的内容，我们根据逐行读取文件内容，将每次读取到的内容，在send()给下一个函数，也就是grep()，這个函数功能是实现过滤操作。我们从上一个函数cat()接受到的每一行内容，在grep()函数里面进行过滤处理，如果有匹配到的过滤内容，那么我们将就过滤到的文件传递给下一个函数print()，该函数主要是打印出文件路径。也许，上述的描述内容你没看懂，下面看看这个程序的流程图：



　　根据上述流程，我们很清楚知道，send()传递给下一个函数的值。但是上述代码存在一个问题：如果待过滤的字符在一个文件中存在多个，而在读取文件的时候，我们是一行一行地读取，然后再传递给下一个函数。我们的目的是：过滤出包好pattern的文件，如果一个文件存在多个同样的pattern，那么就会输出多次同样的文件名。这无疑是浪费内存，要解决这中问题，我们可以通过yield的返回值来控制，每次读取时候的条件。具体实施，看上述代码注释。

# [详解生成器、迭代器](https://www.cnblogs.com/vipchenwei/p/6991204.html)

## 本文讲述了以下几个方面:

　　1.何为迭代，何为可迭代对象，何为生成器，何为迭代器？

　　2.可迭代对象与迭代器之间的区别

　　3.生成器内部原理解析，for循环迭代内部原理解析

　　4.可迭代对象，迭代器，生成器，生成器函数之间关系

## 1.迭代

　　要搞清楚什么关于迭代器，生成器，可迭代对象，前提是我们要理解何为迭代。

　　第一，迭代需要重复进行某一操作

　　第二，本次迭代的要依赖上一次的结果继续往下做，如果中途有任何停顿，都不能算是迭代.

　　下面来看看几个例子，你就会更能理解迭代的含义。

# example1  
# 非迭代

count = 0

while count < 10:

print("hello world")

count += 1

# example2  
# 迭代

count = 0

while count < 10:

print(count)

count += 1

　　例子1，仅仅只是在重复一件事,那就是不停的打印"hello world"，并且,这个打印的结果并不依赖上一次输出的值。而例子2,就很好地说明迭代的含义，重复+继续。

## 2.可迭代对象

　　按照上面迭代的含义，我们应该能够知道何为可迭代对象。顾名思义，就是一个对象能够被迭代的使用。那么我们该如何判断一个对象是否可迭代呢？

　　Python提供了模块collections，其中有一个isinstance(obj,string)的函数，可以判断一个对象是否为可迭代对象。看下面实例:

[复制代码](javascript:void(0);)

from collections import Iterable

f = open('a.txt')

i = 1

s = '1234'

d = {'abc':1}

t = (1,2,344)

m = {1,2,34,}

print(isinstance(i, Iterable)) # 判断整型是否为可迭代对象

print(isinstance(s, Iterable)) # 判断字符串对象是否为可迭代对象

print(isinstance(d, Iterable)) # 判断字典对象是否为可迭代对象

print(isinstance(t, Iterable)) # 判断元组对象是否为可迭代对象

print(isinstance(m, Iterable)) # 判断集合对象是否为可迭代对象

print(isinstance(f, Iterable)) # 判断文件对象是否为可迭代对象

########输出结果#########

False

True

True

True

True

True

[复制代码](javascript:void(0);)

　　由上面得出，除了整型之外，python内的基本数据类型都是可迭代对象，包括文件对象。那么，python内部是如何知道一个对象是否为可迭代对象呢？答案是，在每一种数据类型对象中，都会有有一个\_\_iter\_\_()方法，正是因为这个方法，才使得这些基本数据类型变为可迭代。

　　如果不信，我们可以来看看下面代码片段:

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif 检测属性方法

　　如果大家还是不信，可以继续来测试。我们自己来写一个类，看看有\_\_iter\_\_()方法和没有此方法的区别。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　从上面，实验结果可以看出一个对象是否可迭代，关键看這个对象是否有\_\_iter\_\_()方法。

## 3.迭代器

　　在介绍迭代器之前，我们先来了解一下容器这个概念。

　　容器是一种把多个元素组织在一起的数据结构，容器中的元素可以逐个地迭代获取。简单来说，就好比一个盒子,我们可以往里面存放数据，也可以从里面一个一个地取出数据。

　　在python中，属于容器类型地有:list,dict,set,str,tuple.....。容器仅仅只是用来存放数据的，我们平常看到的 l = [1,2,3,4]等等，好像我们可以直接从列表这个容器中取出元素，但事实上容器并不提供这种能力，而是可迭代对象赋予了容器这种能力。

　　说完了容器，我们在来谈谈迭代器。迭代器与可迭代对象区别在于:\_\_next\_\_()方法。

　　我们可以采用以下方法来验证一下:

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　结果显示:除了文件对象为迭代器，其余均不是迭代器

　　下面，我们进一步来验证一下:

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　从输出结果可以表明,迭代器与可迭代对象仅仅就是\_\_next\_\_（）方法的有无。

## 4.for内部机制剖析

　　先来看看一段普通的迭代过程:

l = [1,2,3,4,5]

for i in l:

print(i)

　  根据之前的分析，我们知道 l = [1,2,3,4,5]是一个可迭代对象。而且可迭代对象是不可以直接从其中取到元素。那么为啥我们还能从列表L中取到元素呢？这一切都是因为for循环内部实现。在for循环内部，首先L会调用\_\_iter\_\_()方法，将列表L变为一个迭代器，然后这个迭代器再调用其\_\_next\_\_()方法，返回取到的第一个值,这个元素就被赋值给了i，接着就打印输出了。

　　下面，我们通过一系列的实验来证明上述所说的。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　上述实验，与我上面说明的一致。

　　下面，我们可以while循环来模拟for循环，输出列表中的元素。

[复制代码](javascript:void(0);)

l = [1,2,3,4,5]

item = l.\_\_iter\_\_() # 生成一个迭代器

while True:

try:

i = item.\_\_next\_\_()

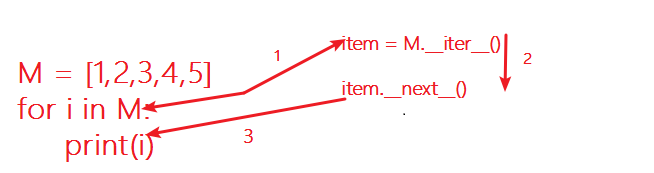
print(i)

except StopIteration: # 捕获异常，如果有异常，说明应该停止迭代

break

[复制代码](javascript:void(0);)

　　由上分析，我们可以总结出:当我们试图用for循环来迭代一个可迭代对象时候，for循环在内部进行了两步操作:第一，将可迭代对象S变为迭代器M；第二，迭代器M调用\_\_next\_\_()方法，并且返回其取出的元素给变量i。



　　你可能看见过这种写法,for i in iter(M):xxx ,其实这一步操作和我们上面没什么区别。iter()函数，就是将一个可迭代对象M变为迭代器也就是M调用\_\_iter\_\_()方法，然后内部在调用\_\_next\_\_()方法。也就是说,

[复制代码](javascript:void(0);)

M = [1,2,3,4,5]

for i in iter(M): # 等价于 M.\_\_iter()\_\_ 人为显示调用

print(i)

for i in M: # 解释器隐式调用

print(i)

#################

#

#上面输出的结果完全一样

#

#################

[复制代码](javascript:void(0);)

　　还有next(M)等价于M.\_\_next\_\_。

### 迭代器优点:

　　　　1.节约内存

　　　　2.不依赖索引取值

　　　　3.实现惰性计算(什么时候需要，在取值出来计算)

## 5.生成器(本质就是迭代器)

　　什么是生成器？可以理解为一种数据类型，这种数据类型自动实现了迭代器协议（其他的数据类型需要调用自己内置的\_\_iter\_\_方法）。

　　按照我们之前所说的，迭代器必须满足两个条件:既有\_\_iter\_\_()，又有\_\_next\_\_()方法。那么生成器是否也有这两个方法呢？答案是，YES。具体来通过以下代码来看看。

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　实验表明,生成器就是迭代器。

　　Python有两种不同的方式提供生成器:

**1.生成器函数（函数内部有yield关键字）：**常规函数定义，但是，使用yield语句而不是return语句返回结果。yield语句一次返回一个结果，在每个结果中间，挂起函数的状态，以便下次重它离开的地方继续执行

**2.生成器表达式：**类似于列表推导，但是，生成器返回按需产生结果的一个对象，而不是一次构建一个结果列表

* 列表解析  
  [expr for iter\_var in iterable if cond\_expr]
* 生成器表达式  
  (expr for iter\_var in iterable if cond\_expr)

　　既然生成器就是迭代器,那么我们是不是也可以通过for循环来遍历出生成器中的内容呢?看下面代码.

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　很显然，生成器也可以通过for循环来遍历出其中的内容。

　　下面我们来看看生成器函数执行流程：

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　每次调用g.\_\_next\_\_()就回去函数内部找yield关键字，如果找得到就输出yield后面的值并且返回；如果没有找到，就会报出异常。上述代码中如果在调用g.\_\_next\_\_()就会报错。

　　Python使用生成器对延迟操作提供了支持。所谓延迟操作，是指在需要的时候才产生结果，而不是立即产生结果。这也是生成器的主要好处。

|  |  |
| --- | --- |
| 1 | 实例:生成器模拟Linux下tail -f a.txt | grep 'error' | grep '404' |

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　生成器小结：

　　　　1.是可迭代对象

　　　　2.实现了延迟计算,省内存啊

　　　　3.生成器本质和其他的数据类型一样，都是实现了迭代器协议，只不过生成器附加了一个延迟计算省内存的好处，其余的可迭代对象可没有这点好处！

## 6.可迭代对象、迭代器、生成器关系总结

