# Python yield 使用浅析

廖 雪峰
2012 年 11 月 22 日发布

您可能听说过，带有 yield 的函数在 Python 中被称之为 generator（生成器），何谓 generator？

我们先抛开 generator，以一个常见的编程题目来展示 yield 的概念。

## 如何生成斐波那契數列

斐波那契（Fibonacci）數列是一个非常简单的递归数列，除第一个和第二个数外，任意一个数都可由前两个数相加得到。用计算机程序输出斐波那契數列的前 N 个数是一个非常简单的问题，许多初学者都可以轻易写出如下函数：

清单 1. 简单输出斐波那契數列前 N 个数

```
1  def fab(max):
2     n, a, b = 0, 0, 1
3     while n < max:
4         print b
5         a, b = b, a + b
6         n = n + 1
```

执行 fab(5)，我们可以得到如下输出：

```
>>> fab(5)
1
1
2
3
5
```

结果没有问题，但有经验的开发者会指出，直接在 fab 函数中用 print 打印数字会导致该函数可复用性较差，因为 fab 函数返回 None，其他函数无法获得该函数生成的数列。

要提高 fab 函数的可复用性，最好不要直接打印出数列，而是返回一个 List。以下是 fab 函数改写后的第二个版本：

清单 2. 输出斐波那契數列前 N 个数第二版

```
def fab(max):
    n, a, b = 0, 0, 1
    L = []
    while n < max:
        L.append(b)
        a, b = b, a + b
        n = n + 1
    return L
```

可以使用如下方式打印出 fab 函数返回的 List：

```
>>> for n in fab(5):
...     print n
...
1
1
2
3
5
```

改写后的 fab 函数通过返回 List 能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List

来保存中间结果，而是通过 iterable 对象来迭代。例如，在 Python2.x 中，代码：

清单 3. 通过 iterable 对象来迭代

```
1 for i in range(1000): pass
```

会导致生成一个 1000 个元素的 List，而代码：

```
1 for i in xrange(1000): pass
```

则不会生成一个 1000 个元素的 List，而是在每次迭代中返回下一个数值，内存空间占用很小。因为 xrange 不返回 List，而是返回一个 iterable 对象。

利用 iterable 我们可以把 fab 函数改写为一个支持 iterable 的 class，以下是第三个版本的 Fab：

清单 4. 第三个版本

```
 1 class Fab(object):
 2
 3     def __init__(self, max):
 4         self.max = max
 5         self.n, self.a, self.b = 0, 0, 1
 6
 7     def __iter__(self):
 8         return self
 9
10     def next(self):
11         if self.n < self.max:
12             r = self.b
13             self.a, self.b = self.b, self.a + self.b
```

```
            self.n = self.n + 1
            return r
        raise StopIteration()
```

Fab 类通过 next() 不断返回数列的下一个数，内存占用始终为常数：

```
>>> for n in Fab(5):
...     print n
...
1
1
2
3
5
```

然而，使用 class 改写的这个版本，代码远远没有第一版的 fab 函数来得简洁。如果我们想要保持第一版 fab 函数的简洁性，同时又要获得 iterable 的效果，yield 就派上用场了：

清单 5. 使用 yield 的第四版

```
def fab(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        # print b
        a, b = b, a + b
        n = n + 1

'''
```

第四个版本的 fab 和第一版相比，仅仅把 print b 改为了 yield b，就在保持简洁性的同时获得了 iterable 的效果。

调用第四版的 fab 和第二版的 fab 完全一致：

```
>>> for n in fab(5):
...     print n
...
1
1
2
3
5
```

简单地讲，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

也可以手动调用 fab(5) 的 next() 方法（因为 fab(5) 是一个 generator 对象，该对象具有 next() 方法），这样我们就可以更清楚地看到 fab 的执行流程：

清单 6. 执行流程

```
>>> f = fab(5)
>>> f.next()
1
>>> f.next()
1
>>> f.next()
2
>>> f.next()
3
>>> f.next()
5
>>> f.next()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
StopIteration
```

当函数执行结束时，generator 自动抛出 StopIteration 异常，表示迭代完成。在 for 循环里，无需处理 StopIteration 异常，循环会正常结束。

我们可以得出以下结论：

一个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

如何判断一个函数是否是一个特殊的 generator 函数？可以利用 isgeneratorfunction 判断：

清单 7. 使用 isgeneratorfunction 判断

```
>>> from inspect import isgeneratorfunction
>>> isgeneratorfunction(fab)
True
```

要注意区分 fab 和 fab(5)，fab 是一个 generator function，而 fab(5) 是调用 fab 返回的一个 generator，好比类的定义和类的实例的区别：

清单 8. 类的定义和类的实例

```
>>> import types
>>> isinstance(fab, types.GeneratorType)
False
>>> isinstance(fab(5), types.GeneratorType)
True
```

fab 是无法迭代的，而 fab(5) 是可迭代的：

```
>>> from collections import Iterable
>>> isinstance(fab, Iterable)
False
>>> isinstance(fab(5), Iterable)
True
```

每次调用 fab 函数都会生成一个新的 generator 实例，各实例互不影响：

```
>>> f1 = fab(3)
>>> f2 = fab(5)
>>> print 'f1:', f1.next()
f1: 1
>>> print 'f2:', f2.next()
f2: 1
>>> print 'f1:', f1.next()
f1: 1
>>> print 'f2:', f2.next()
f2: 1
>>> print 'f1:', f1.next()
f1: 2
>>> print 'f2:', f2.next()
f2: 2
>>> print 'f2:', f2.next()
f2: 3
>>> print 'f2:', f2.next()
f2: 5
```

# return 的作用

在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

# 另一个例子

另一个 yield 的例子来源于文件读取。如果直接对文件对象调用 read() 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。通过 yield，我们不再需要编写读文件的迭代类，就可以轻松实现文件读取：

清单 9. 另一个 yield 的例子

```
def read_file(fpath):
   BLOCK_SIZE = 1024
   with open(fpath, 'rb') as f:
       while True:
           block = f.read(BLOCK_SIZE)
           if block:
               yield block
           else:
               return
```

以上仅仅简单介绍了 yield 的基本概念和用法，yield 在 Python 3 中还有更强大的用法，我们会在后续文章中讨论。

注：本文的代码均在 Python 2.7 中调试通过

---