《源码探秘 CPython》65. 生成器的实现原理（上）

原创 古明地觉 古明地觉的编程教室 2022-04-11 09:00

楔子

本次我们来聊一聊Python的生成器，它是我们后续理解协程的基础。生成器的话，估计大部分人在写程序的时候都不怎么用，但其实生成器一旦用好了，确实能给程序带来性能上的提升，那么我们就来看一看吧。

生成器的基础知识

我们知道，一个函数如果它的内部出现了yield关键字，那么它就不再是普通的函数了，而是一个生成器函数。当我们调用的时候，就会创建一个生成器对象。

生成器对象一般用于处理循环结构，应用得当的话可以极大优化内存使用率。比如：我们读取一个大文件。

```
def read_file(file):
    return open(file, encoding="utf-8").readlines()

print(read_file("假装是大文件.txt"))
# ['人生是什么?\n', '大概是闪闪发光的同时\n', '又让人感到痛苦的东西吧']
```

这个版本的函数，直接将里面的内容全部读取出来了，返回了一个列表。如果文件非常大，那么内存的开销可想而知。于是我们可以通过yield关键字，将普通函数变成一个生成器函数。

```
def read_file(file):
    with open(file, encoding="utf-8") as f:
        for line in f:
            yield line

data = read_file("假装是大文件.txt")
print(data)  # <generator object read_file at 0x0000019B4FA8BAC0>

# 这里返回了一个生成器对象，我们可以使用 for 循环遍历
for line in data:
    # 文件每一行自带换行符，所以这里的 print 就不用换行符了
    print(line, end="")
"""
人生是什么?
大概是闪闪发光的同时
又让人感到痛苦的东西吧
"""
```

由于生成器是一种特殊的迭代器，那么我们也可以使用它的 __next__ 方法。

```
def gen():
    yield 123
    yield 456
```

```
    yield 789
    return "result"

# 调用生成器函数时，会创建一个生成器
# 生成器虽然创建了，但是里面的代码并没有执行
g = gen()

# 调用__next__方法时才会执行
# 当遇到 yield，会将生成器暂停、并返回yield后面的值
print(g.__next__())  # 123

# 此时生成器处于暂停状态，如果我们不驱动它的话，它是不会前进的
# 再次执行__next__，生成器恢复执行，并在下一个yield处暂停
print(g.__next__())  # 456

# 生成器会记住自己的执行进度，它总是在遇到yield时暂停
# 调用 __next__ 时恢复执行，直到遇见下一个 yield
print(g.__next__())  # 789

# 显然再调用 __next__ 时，已经找不到下一个 yield 了
# 那么生成器会抛出 StopIteration，并将返回值设置在里面
try:
    g.__next__()
except StopIteration as e:
    print(f"返回值:{e.value}")  # result
```

可以看到，基于生成器，我们能够实现惰性求值。

当然啦，生成器不仅仅有 __next__ 方法，它还有 send 和 throw 方法，我们先来说一说 send。

```
def gen():
    res1 = yield "yield 1"
    print(f"res = {res1}")
    res2 = yield "yield 2"
    return res2

g = gen()

# 此时程序在第一个 yield 处暂停
print(g.__next__())
"""
yield 1
"""

# 调用 g.send(val) 依旧可以驱动生成器执行
# 同时还可以传递一个值，交给第一个 yield 左边的 res1
# 然后寻找第二个 yield
print(g.send("嘿嘿"))
"""
res = 嘿嘿
yield 2
"""
# 上面输出了两行，第一行是生成器里面的 print 打印的

try:
    # 此时生成器在第二个 yield 处暂停，调用 g.send 驱动执行
    # 同时传递一个值交给第二个 yield 左边的 res2，然后寻找第三个 yield
    # 但是生成器里面没有第三个 yield 了，于是抛出 StopIteration
    g.send("蛤蛤")
except StopIteration as e:
    print(f"返回值:{e.value}")
"""
返回值:蛤蛤
```

```
34  """
```

生成器永远在 yield 处暂停，并且会将 yield 后面的值返回。如果想驱动生成器继续执行，可以调用 __next__、send，会去寻找下一个 yield，然后在下一个 yield 处暂停。依次往复，直到找不到 yield 时，抛出 StopIteration，并将返回值包在里面。

但是这两者的不同之处在于，send 可以接收参数，比如 **res = yield 123**，假设生成器在 **yield 123** 这里停下来了。

当调用 __next__ 和 send 的时候，都可以驱动执行，但调用 send 时可以传递一个 value，并将 value 赋值给变量 res。而 __next__ 没有这个功能，如果是调用 __next__ 的话，那么 res 得到的就是一个 None。

所以 **res = yield 123** 这一行语句需要两次驱动生成器才能完成，第一次驱动会让生成器执行到 yield 123，然后暂停执行，将 123 返回；第二次驱动才会给变量 res 赋值，此时会寻找下一个 yield 然后暂停。

刚创建生成器的时候，里面的代码还没有执行，它的 f_lasti 是 -1。还记得这个 f_lasti 吗？它表示上一条执行完的指令的偏移量。

```
1  def gen():
2      res1 = yield 123
3      res2 = yield 456
4      return "result"
5
6  g = gen()
7  # 生成器函数和普通函数一样，执行时也会创建栈帧
8  # 通过 g.gi_frame 可以很方便的获取
9  print(g.gi_frame.f_lasti)  # -1
```

f_lasti 是 -1，表示生成器刚被创建，还没有执行任何指令。而第一次驱动生成器的执行，叫做生成器的预激。

而在生成器还没有被预激时，我们调用 send，里面只能传递一个 None，否则报错。

```
1  def gen():
2      res1 = yield 123
3      res2 = yield 456
4      return "result"
5
6  g = gen()
7  try:
8      g.send([])
9  except TypeError as e:
10     print(e)
11 # can't send non-None value to a just-started generator
```

对于尚未被预激的生成器，我们只能传递一个 None，也就是 g.send(None)。或者调用 g.__next__()，因为不管何时它传递的都是 None。

其实也很好理解，我们之所以传值是为了赋给 yield 左边的变量，这就意味着生成器必须至少被驱动一次、在某个 yield 处停下来才可以。而未被预激的生成器，它里面的代码压根就没有执行，所以第一次驱动的时候只能传递一个 None 进去。

如果查看生成器的源代码的话，也能证明这一点：

```
192     // 如果 f_lasti 为 -1，说明生成器尚未执行
193     if (f->f_lasti == -1) {
194         // arg 就是驱动生成器执行时传递的值
195         // 如果它不为 None，报错
```

```
196         if (arg && arg != Py_None) {
197             // 报错信息和我们在 Python 里面看到的是一样的
198             const char *msg = "can't send non-None value t
199                               "just-started generator";
200             if (PyCoro_CheckExact(gen)) {
201                 msg = NON_INIT_CORO_MSG;
202             }
203             else if (PyAsyncGen_CheckExact(gen)) {
204                 msg = "can't send non-None value to a "
205                       "just-started async generator";
206             }
207             // 设置异常，类型为 TypeError，信息为 msg
208             PyErr_SetString(PyExc_TypeError, msg);
209             return NULL;
210         }
211     } else {
```


介绍完 __next__ 和 send 之后，再来看看 throw。throw 方法的作用和前两者类似，也是驱动生成器执行，并在下一个 yield 处暂停。但它在调用的时候，可以传递一个异常进去。

```python
def gen():
    try:
        yield 123
    except ValueError as e:
        print(e)
    yield 456
    return "result"

g = gen()
# 生成器在 yield 123 处暂停
g.__next__()
# 向生成器传递一个异常
# 如果当前生成器的暂停位置处无法捕获传递的异常，那么会将异常抛出来
# 如果能够捕获，那么会驱动生成器执行，并在下一个 yield 处暂停
# 然后返回 yield 后面的值
# 当前生成器在 yield 123 处暂停，而它所在位置能够捕获异常
# 所以不会报错，结果就是 456 会赋值给 val
val = g.throw(ValueError("抛出异常"))
print("------------")
print(val)
"""
抛出异常
------------
456
"""
```

以上就是 __next__、send、throw 三个方法的用法，还是比较简单的。

生成器也是可以关闭的。

```python
def gen():
    yield 123
    yield 456
    return "result"

g = gen()
# 生成器在 yield 123 处停止
print(g.__next__())
# 关闭生成器
```

```
g.close()
try:
    # 再次调用 __next__，会抛出 StopIteration
    g.__next__()
except StopIteration as e:
    # 此时返回值为 None
    print(e.value)  # None
```

这里再来说一说 GeneratorExit 这个异常，如果我们删除一个生成器（或者关闭），那么会往里面扔一个 GeneratorExit 进去。

```
def gen():
    try:
        yield 123
    except GeneratorExit as e:
        print("生成器被删除了")

g = gen()
# 生成器在 yield 123 处暂停
g.__next__()
# 生成器只持有 g 这一个引用
# 所以 del g 之后，生成器也会被删除
# 而删除生成器，会往里面扔一个 GeneratorExit
del g
"""
生成器被删除了
"""
```

这里我们捕获了传递的 GeneratorExit，所以 print 语句执行了，但如果没有捕获呢？

```
def gen():
    yield 123

g = gen()
g.__next__()
del g
```

此时无事发生，但是注意：如果是调用 throw 方法扔一个 GeneratorExit 进去，异常还是会抛出来的。

那么问题来了，生成器为什么要提供这一个机制呢？直接删就完了，干嘛还要往生成器内部丢一个异常呢？答案是为了资源的清理和释放。

在Python还未提供原生协程、或者 asyncio 还尚未流行起来的时候，很多开源的协程框架都是基于生成器实现的协程。而创建连接的逻辑，一般都会写在 yield 后面。

```
def _create_connection():
    # 一些逻辑
    yield conn
    # 一些逻辑
```

但是这些连接在不用的时候，要不要进行释放呢？答案是肯定的，所以便可以这么做：

```
def _create_connection():
    # 一些逻辑
    try:
        yield conn
    except GeneratorExit:
        conn.close()
```

```
    # 一些逻辑
```

这样当我们将生成器删除的时候，就能够自动对连接进行释放了。

但是还有一个需要注意的点，就是在捕获 GeneratorExit 之后，不可以再执行 yield，否则会抛出 RuntimeError（但不会终止程序）。

```
def gen():
    try:
        yield 123
    except GeneratorExit:
        print("生成器被删除")
        yield

g = gen()
g.__next__()
del g
print("抛出 RuntimeError，但不影响程序执行")
"""
Exception ignored in: <generator object gen at 0x000002242CADCA50>
Traceback (most recent call last):
  File "...........", line 10, in <module>
    del g
RuntimeError: generator ignored GeneratorExit
生成器被删除
抛出 RuntimeError，但不影响程序执行
"""
```

首先，如果我们没有成功捕获GeneratorExit，那么生成器会直接被删掉，不会有任何事发生；但如果我们捕获了GeneratorExit，就意味着生成器被删除了，那么就不应该再出现 yield 了。

所以这时候解释器会抛出一个 RuntimeError 以示警告，因为没捕获GeneratorExit 还好，解释器不会有什么抱怨；但如果捕获了GeneratorExit，说明我们知道生成器是被删除（或者关闭）了，既然知道，那里面还出现 yield 的意义何在呢？

所以解释器抛出 RuntimeError，并告诉我们生成器将 GeneratorExit 忽略了（因为我们捕获了），但是不应该再出现 yield 了。另外，虽然抛了异常，但是不会终止程序的执行。

> 如果将来面试官问你，能不能举出一个例子：解释器在执行过程中主动抛了异常，但是在没有异常捕获的情况下，程序依旧正常执行。
>
> 那么你就可以用这个例子回答他。

当然啦，如果出现了 yield，但是执行之前就返回了，也不会抛出 RuntimeError。

```
def gen():
    try:
        yield 123
    except GeneratorExit:
        print("生成器被删除")
        return
        yield

g = gen()
g.__next__()
del g
print("--------------")
"""
生成器被删除
--------------
"""
```

遇见 yield 之前就返回了，所以此时不会出现 RuntimeError。

> 注意：GeneratorExit 继承自 BaseException，它无法被 Exception 捕获。

yield from

当函数内部出现了 yield 关键字，那么它就是一个生成器函数，对于 yield from 而言亦是如此。那么问题来了，这两者之间有什么区别呢？

```python
from typing import Generator

def gen1():
    yield [1, 2, 3]

def gen2():
    yield from [1, 2, 3]

g1 = gen1()
g2 = gen2()
# 两者都是生成器
print(isinstance(g1, Generator))  # True
print(isinstance(g2, Generator))  # True

print(g1.__next__())  # [1, 2, 3]
print(g2.__next__())  # 1
```

结论很清晰，yield 对后面的值没有要求，直接将其返回；而 yield from 后面必须跟一个可迭代对象(否则报错)，然后每次返回可迭代对象的一个值。

```python
def gen():
    yield from [1, 2, 3]
    return "result"

g = gen()
print(g.__next__())  # 1
print(g.__next__())  # 2
print(g.__next__())  # 3
try:
    g.__next__()
except StopIteration as e:
    print(e.value)  # result
```

除了要求必须跟一个可迭代对象、然后每次只返回一个值之外，其它表现和 yield 是类似的。而且事实上，**yield from [1, 2, 3]** 类似于 **for item in [1, 2, 3]: yield item**

这里出一道思考题：

```
# 有这样一个列表
[1, [[[[3, 3], 5]]], [[[[[[[[[[[[6]]]]]]]]]]], 8]]], "aaa"]]]], 250]]
# 编写一个函数，将列表转成扁平化
# 得到如下结果
[1, 3, 3, 5, 6, 8, "aaa", 250]
```


此时我们就可以通过 yield 和 yield from 来实现这一点：

```python
def flatten(lst):
    for item in lst:
        (yield from flatten(item)) \
```

```
            if isinstance(item, list) else (yield item)

lst = [1, [[[[3, 3], 5]]], [[[[[[[[[[[[6]]]]]]]]]]]], 8]]], "aaa"]]]], 250]]
print(list(flatten(lst)))  # [1, 3, 3, 5, 6, 8, 'aaa', 250]
```

怎么样，是不是很简单呢？

如果单从语法上来看的话，会发现yield from貌似没什么特殊的地方，但其实yield from还可以作为委托生成器。

委托生成器会在调用方和子生成器之间建立一个双向通道，什么意思呢？我们举例说明。

```
def gen():
    yield 123
    yield 456
    return "result"

def middle():
    res = yield from gen()
    print(f"接收到子生成器的返回值: {res}")

# 我们调用了 middle，此时我们便是调用方
# 然后 middle 里面 yield from gen()
# 那么 middle() 便是委托生成器，gen() 是子生成器
g = middle()

# 而 yield from 会在调用方和子生成器之间建立一个双向通道
# 两者是可以互通的
# 我们调用 g.send、g.throw 都会直接传递给子生成器
print(g.__next__())  # 123
print(g.__next__())  # 456

# 问题来了，如果再调用一次 __next__ 会有什么后果呢？
# 按照之前的理解，应该会抛出 StopIteration
g.__next__()
"""
接收到子生成器的返回值: result
Traceback (most recent call last):
  File "......", line 23, in <module>
    g.__next__()
StopIteration
"""
```

在第三次调用 __next__ 的时候，确实抛了异常，但是委托生成器收到了子生成器的返回值。也就是说，委托生成器在调用方和子生成器之间建立了双向通道，两者是直接通信的，但是当子生成器出现 StopIteration 时，委托生成器还要负责兜底。

委托生成器会将子生成器抛出的 StopIteration 里面的 value 取出来，然后赋值给左侧的变量 res，并在自己内部继续寻找 yield。

换句话说，当子生成器 return 之后，委托生成器会拿到返回值，并将子生成器抛出的异常给捕获掉。但是还没完，因为还要找到下一个 yield，那么从哪里找呢？显然是委托生成器的内部寻找，于是接下来就变成了调用方和委托生成器之间的通信。

如果在委托生成器内部能找到下一个 yield，那么会将值返回给调用方。如果找不到，那么就重新构造一个 StopIteration，将异常抛出去。此时异常的 value 属性，就是委托生成器的返回值。

```
def gen():
    yield 123
```

```
    return "result"

def middle():
    res = yield from gen()
    return "委托生成器"

g = middle()
print(g.__next__())  # 123
try:
    g.__next__()
except StopIteration as e:
    print(e.value)  # 委托生成器
```

但是大部分情况下，我们并不关注委托生成器的返回值，我们更关注的是子生成器。于是可以换种写法：

```
def gen():
    yield 123
    return "result"

def middle():
    yield (yield from gen())

g = middle()
print(g.__next__())  # 123
print(g.__next__())  # result
```

所以委托生成器是负责在调用方和子生成器之间建立一个双向通道，通道一旦建立，调用方可以和子生成器直接通信。虽然调用的是委托生成器，但调用 g 的 __next__、send、throw 等方法，影响的都是子生成器。

并且委托生成器还可以对子生成器抛出的异常进行兜底，会捕获掉里面的异常，然后拿到返回值，这样就无需手动捕获子生成器的异常了。但问题是委托生成器还要找到下一个 yield，并将值返回给调用方，此时这个重担就落在了它自己头上。

如果找不到，还是要将异常抛出来的，只不过抛出的 StopIteration 是委托生成器构建的。而子生成器抛出的 StopIteration，早就被委托生成器捕获掉了。

于是我们可以考虑在 yield from 的前面再加上一个 yield，这样就不会抛异常了。

我们上面已经见识到了委托生成器的用法，不过问题来了，这玩意为啥会存在呢？我们上面的逻辑，即便不使用 yield from 也可以完成啊。

其实是因为我们上面的示例代码比较简单（为了演示用法），当需求比较复杂时，将生成器内部的部分操作委托给另一个生成器是有必要的，这也是委托生成器的由来。

而委托生成器不仅要能保证调用方和子生成器直接通信，还要能够以一种优雅的方式获取子生成器的返回值，于是新的语法 yield from 就诞生了。

但其实 yield from 背后为我们做得事情还不止这么简单，它不单单是建立双向通道、获取子生成器的返回值，它还会处理子生成器内部出现的异常，详细内容可以查看**PEP380**。

> https://peps.python.org/pep-0380/

这里我们直接给出结论：

- 子生成器 yield 后面的值，会直接传给调用方；调用方 send 发送的值，也会直接传给子生成器。
- 子生成器结束时，最后的 return value 会触发一个 StopIteration(value)；然后该异常会被 yield from 捕获，并将 value 赋值给 yield from 左侧的变量。注意：生成器抛出的 StopIteration 必须是执行结束时自己抛出的，我们不可以在内部手动 raise StopIteration。

- 在拿到子生成器的返回值时，委托生成器会继续运行，寻找下一个 yield。
- 如果子生成器在执行的过程中，内部出现了异常，那么会将异常丢给委托生成器。委托生成器会检测异常是不是 GeneratorExit，如果不是，那么就调用子生成器的 throw 方法。
- 如果在委托生成器上调用 close 或者传入 GeneratorExit，那么会调用子生成器的 close。如果调用的时候出现异常，那么向上抛，否则的话，委托生成器会抛出 GeneratorExit。

yield from 算是 Python 里面特别难懂的一个语法了，但如果理解了 yield from，后续理解 await 就会简单很多。

Python 里面还有一个生成器表达式，我们来看一下：

```
from typing import Generator

g = (x for x in range(10))
print(isinstance(g, Generator))  # True
print(g)  # <generator object <genexpr> at 0x...>

print(g.__next__())  # 0
print(g.__next__())  # 1
```

如果表达式是在一个函数里面，那么生成器表达式周围的小括号可以省略掉。

```
import random

d = [random.randint(1, 10) for _ in range(100)]
# 我们想统计里面大于 5 的元素的总和
# 下面的做法都是可以的
print(
    sum((x for x in d if x > 5)),
    sum(x for x in d)
)
```

这两种做法是等价的，字节码完全一样。

生成器表达式还存在一些陷阱，一不小心就踩进去。至于是什么陷阱呢？很简单，一句话：**使用生成器表达式创建生成器的时候，in 后面的变量就已经确定好了，但其它的变量则不会。**举个栗子：

```
g = (反叒叟叕段变叛 for x in [1, 2, 3])
```

执行这段代码不会报错，尽管 for 前面那一坨我们没有定义，但不要紧，因为生成器是惰性执行的。如果我们调用了 g.__next__()，那么很明显就报错了，会抛出 NameError。

```
g = (x for x in lst)
```

但是这段代码会报错：NameError: name 'lst' is not defined，因为 in 后面变量是谁，在创建生成器的时候就已经确定好了。而在创建生成器的时候，发现 lst 没有定义，于是抛出 NameError。

所以，陷阱就来了：

```
i = 1
g = (x + i for x in [1, 2, 3])
i = 10
```

```
# 输出的不是 (2, 3, 4)
print(tuple(g))  # (11, 12, 13)
```

因为生成器只有在执行的时候，才会去确定变量 i 究竟指向谁，而调用 tuple(g) 的时候 i 已经被修改了。

```
lst = [1, 2, 3]
g = (x for x in lst)
lst = [4, 5, 6]
print(tuple(g))  # (1, 2, 3)
```

但这里输出的又是 (1, 2, 3)，因为在创建生成器的时候，in 后面的变量就已经确定了，这里会和 lst 指向同一个列表。而第三行改变的只是变量 lst 的指向，和生成器无关。

```
g = (x for x in [1, 2, 3, 4])
for i in [1, 10]:
    g = (x + i for x in g)

print(tuple(g))
```

思考一下，上面代码会打印啥？下面进行分析：

- 初始的 g，可以看成是 **(1, 2, 3, 4)**，因为 in 后面是啥，在创建生成器的时候就确定了；
- 第一次循环之后，g 就相当于 **(1+i, 2+i, 3+i, 4+i)**；
- 第二次循环之后，g 就相当于 **(1+i+i, 2+i+i, 3+i+i, 4+i+i)**；

而循环结束，i 会指向 10，所以打印结果就是 **(21, 22, 23, 24)**。

以上我们就从 Python 的角度梳理了一遍生成器的相关知识，但这只是文章的上半部分，而下半部分我们将从解释器源代码的角度来分析生成器的实现。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》66. 生成器的实现原理（下）

下一篇 ›
《源码探秘 CPython》64. 装饰器是怎么实现的?