

迭代器与异常

金融科技协会 2020 年 11 月 26 日

目录

1 异常.....	2
1.1 定义.....	2
1.2 raise 故意引发指定异常.....	2
1.3 常见的内置异常类.....	3
1.3.1 AttributeError.....	3
1.3.2 IndexError.....	3
1.3.3 KeyError.....	3
1.3.4 NameError.....	4
1.3.5 SyntaxError.....	4
1.3.6 TypeError.....	5
1.3.7 ZeroDivisionError.....	5
1.3.8 ValueError.....	5
1.4 捕获异常.....	6
1.4.1 捕获单种指定异常.....	6
1.4.2 多个 except 子句.....	6
1.4.3 一个 except 捕获多种异常.....	6
1.4.4 捕获任何异常，并对异常进行识别.....	7
1.4.5 代码捕获异常后，仍报错.....	7
1.4.6 代码捕获异常后，报指定错误.....	8
1.4.7 else 子句.....	8
1.4.8 finally 子句.....	9
2 迭代器和生成器.....	10
2.1 迭代器、可迭代对象.....	10
2.2 初始化一个迭代器.....	11
2.3 使迭代器输出下一个迭代值.....	11
2.4 迭代器转序列.....	11
2.5 生成器.....	11
2.6 生成器的使用.....	13
2.6.1 for.....	13
2.6.2 next().....	13
2.6.3 _next_().....	14
2.6.4 next()与 send().....	14
2.6.5 throw 和 close.....	15

1 异常

1.1 定义

Python 使用**异常对象**来表示异常状态，并在遇到错误时引发异常。异常对象未被处理（或**捕获**）时，程序将终止并显示一条错误消息（**traceback**）。

每个异常都是某个类（这里是 `ZeroDivisionError`）的实例。你能以各种方式引发和捕获这些实例，从而**逮住错误并采取措施**，而不是放任整个程序失败。

```
In [1]: 1/0

ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-1-9e1622b385b6> in <module>
----> 1 1/0

ZeroDivisionError: division by zero
```

图 1: `ZeroDivisionError` 错误示例

1.2 `raise` 故意引发指定异常

要引发异常，可使用 `raise` 语句，并将一个类（必须是 `Exception` 的子类）或实例作为参数。将类作为参数时，将自动创建一个实例。下面的示例使用的是内置异常类 `Exception`：

```
In [2]: raise Exception

Exception                                Traceback (most recent call last)
<ipython-input-2-1aaab48c2748> in <module>
----> 1 raise Exception

Exception:
```

图 2: `raise` 引发内置异常类 `Exception` 示例

在第一个示例（`raise Exception`）中，引发的是通用异常，没有指出出现了什么错误。在第二个示例中，添加了错误消息 `hyperdrive overload`。

```
In [7]: raise Exception('hyperdrive overload')

Exception                                Traceback (most recent call last)
<ipython-input-7-5731b36f442d> in <module>
----> 1 raise Exception('hyperdrive overload')

Exception: hyperdrive overload

In [8]: raise Exception('hhhhhh')

Exception                                Traceback (most recent call last)
<ipython-input-8-76dc6fc3951e> in <module>
----> 1 raise Exception('hhhhhh')

Exception: hhhhhh
```

图 3: `raise` 引发错误并添加错误消息

1.3 常见的内置异常类

- **Exception** 几乎所有的异常类都是从它派生而来的
- **AttributeError** 引用属性或给它赋值失败时引发
- **OSError** 操作系统不能执行指定的任务（如打开文件）时引发，有多个子类
- **IndexError** 使用序列中不存在的索引时引发，为 **LookupError** 的子类
- **KeyError** 使用映射中不存在的键时引发，为 **LookupError** 的子类
- **NameError** 找不到名称（变量）时引发
- **SyntaxError** 代码不正确时引发
- **TypeError** 将内置操作或函数用于类型不正确的对象时引发
- **ValueError** 将内置操作或函数用于这样的对象时引发：其类型正确但包含的值不合适
- **ZeroDivisionError** 在除法或求模运算的第二个参数为零时引发

1.3.1 AttributeError

当引用属性或给它赋值失败时引发，示例如下：

```
In [4]: s = 'THIS IS IN LOWERCASE.'
        sh = s.hhh

AttributeError                                Traceback (most recent call last)
<ipython-input-4-f7e19169c529> in <module>
      1 s = 'THIS IS IN LOWERCASE.'
--> 2 sh = s.hhh

AttributeError: 'str' object has no attribute 'hhh'
```

图 4：attribute error 示例

1.3.2 IndexError

使用序列中不存在的索引时引发，为 **LookupError** 的子类，示例如下：

```
In [10]: s = ['cat', 'dog', 'mouse']
         print(s[6])

IndexError                                Traceback (most recent call last)
<ipython-input-10-1f212d06207c> in <module>
      1 s = ['cat', 'dog', 'mouse']
--> 2 print(s[6])

IndexError: list index out of range
```

图 5：IndexError 示例

1.3.3 KeyError

使用映射中不存在的键时引发，为 **LookupError** 的子类，示例如下：

```
In [11]: s = {'cat': 'Zophie', 'dog': 'Basil', 'mouse': 'Whiskers'}
         print('The name of my pet zebra is ' + s['zebra'])

KeyError                                Traceback (most recent call last)
<ipython-input-11-cc4f4308636e> in <module>
      1 s = {'cat': 'Zophie', 'dog': 'Basil', 'mouse': 'Whiskers'}
--> 2 print('The name of my pet zebra is ' + s['zebra'])

KeyError: 'zebra'
```

图 6: KeyError 示例

1.3.4 NameError

找不到名称（变量,函数等）时引发，示例如下：

```
In [12]: s = Round(4.2)

NameError                                Traceback (most recent call last)
<ipython-input-12-f6ad29d8c0b3> in <module>
--> 1 s = Round(4.2)

NameError: name 'Round' is not defined
```

```
In [13]: s = round(4.2)
         print(s)
```

4

图 7: NameError 示例

1.3.5 SyntaxError

代码不正确时引发：

- 缩进
- 符号漏打
- 中文符号
- == 打成了 =

```
In [17]: print('hhh')

File "<ipython-input-17-ecbdb50b16cf>", line 1
      print('hhh')
      ~~~~~
SyntaxError: invalid character in identifier
```

```
In [15]: print(' hhh)

File "<ipython-input-15-7bd38f80355b>", line 1
      print(' hhh)
      ~~~~~
SyntaxError: EOL while scanning string literal
```

图 8: SyntaxError 示例

1.3.6 TypeError

将内置操作或函数用于类型不正确的对象时引发，示例如下：

```
In [16]: 1+str(2)
```

```
TypeError                                Traceback (most recent call last)
<ipython-input-16-b7904648d92b> in <module>
--> 1 1+str(2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

图 9: TypeError 示例

1.3.7 ZeroDivisionError

在除法或求模运算的第二个参数为零时引发，示例如下：

```
In [18]: 1/0
```

```
ZeroDivisionError                        Traceback (most recent call last)
<ipython-input-18-9e1622b385b6> in <module>
--> 1 1/0

ZeroDivisionError: division by zero
```

图 10: ZeroDivisionError 示例

1.3.8 ValueError

将内置操作或函数用于这样的对象时引发：其类型正确但包含的值不合适，示例如下：

```
In [14]: a, b, c=1, 2
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-14-dfc132c7dc8e> in <module>
--> 1 a, b, c=1, 2

ValueError: not enough values to unpack (expected 3, got 2)
```

```
In [15]: a, b, c=1, 2, 3, 4
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-15-23b3cf4a7041> in <module>
--> 1 a, b, c=1, 2, 3, 4

ValueError: too many values to unpack (expected 3)
```

```
In [16]: a=1, 2, 3
         a
```

```
Out[16]: (1, 2, 3)
```

```
In [17]: type(a)
```

```
Out[17]: tuple
```

图 11: ValueError 示例

1.4 捕获异常

1.4.1 捕获单种指定异常

异常比较有趣的地方是可对其进行处理,通常称之为捕获异常。为此,可使用 try...except:

```
In [32]: try:
          x = int(input('Enter the first number: '))
          y = int(input('Enter the second number: '))
          print(x / y)
        except:
          print("something wrong")

Enter the first number: 1
Enter the second number: 0
something wrong
```

```
In [33]: try:
          x = int(input('Enter the first number: '))
          y = int(input('Enter the second number: '))
          print(x / y)
        except ZeroDivisionError:
          print("The second number can't be zero!")

Enter the first number: 1
Enter the second number: 0
The second number can't be zero!
```

图 12: try...except 捕获异常示例

1.4.2 多个 except 子句

```
In [41]: try:
          x = int(input('Enter the first number: '))
          y = int(input('Enter the second number: '))
          print(x / y)
        except ZeroDivisionError:
          print("The second number can't be zero!")
        except ValueError:
          print("ValueError!!!")

Enter the first number: 1
Enter the second number: gg
ValueError!!!
```

图 13: 多个 except 子句捕获异常使用示例

1.4.3 一个 except 捕获多种异常

```
In [43]: try:
          x = int(input('Enter the first number: '))
          y = int(input('Enter the second number: '))
          print(x / y)
        except (ZeroDivisionError, ValueError, TypeError):
          print('Error!!')

Enter the first number: 1
Enter the second number: hhh
Error!!
```

图 14: 使用一个 except 捕获多种异常示例

1.4.4 捕获任何异常，并对异常进行识别

使用语句 `try...except Exception as e:`

```
In [44]: try:
          x = int(input('Enter the first number: '))
          y = int(input('Enter the second number: '))
          print(x / y)
        except (ZeroDivisionError, TypeError, ValueError) as e:
          print(e)

Enter the first number: 1
Enter the second number: 0
division by zero
```

```
In [21]: try:
          x = (input('Enter the first number: '))
          y = (input('Enter the second number: '))
          print(x / y)
        except (ZeroDivisionError, TypeError, ValueError) as e:
          print(e)

Enter the first number: 1
Enter the second number: hhh
unsupported operand type(s) for /: 'str' and 'str'
```

图 15: `try...except Exception as e` 使用示例

1.4.5 代码捕获异常后，仍报错

```
In [23]: try:
          x = 1
          y = 0
          print(x / y)
        except:
          raise ValueError

ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-23-7f001fb6bac1> in <module>
      3     y = 0
--> 4     print(x / y)
      5 except:

ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-23-7f001fb6bac1> in <module>
      4     print(x / y)
      5 except:
--> 6     raise ValueError

ValueError:
```

图 16: 代码捕获异常后仍报错处理示例

1.4.6 代码捕获异常后，报指定错误

```
In [24]: try:
        x = 1
        y = 0
        print(x / y)
    except:
        raise ValueError('ggg') from None

ValueError                                Traceback (most recent call last)
<ipython-input-24-f5c36dd87c82> in <module>
      4     print(x / y)
      5 except:
--> 6     raise ValueError('ggg') from None

ValueError: ggg
```

图 17：代码捕获异常后报指定错误处理示例

1.4.7 else 子句

在有些情况下，在没有出现异常时执行一个代码块很有用。为此，可像条件语句和循环一样，给 try/except 语句添加一个 else 子句。

```
In [46]: try:
        print('A simple task')
    except:
        print('What? Something went wrong?')
    else:
        print('Ah ... It went as planned.')

A simple task
Ah ... It went as planned.

In [25]: while True:
        try:
            x = int(input('Enter the first number: '))
            y = int(input('Enter the second number: '))
            value = x / y
            print('x / y is', value)
        except:
            print('Invalid input. Please try again.')
        else:
            break

Enter the first number: 1
Enter the second number: g
Invalid input. Please try again.
Enter the first number: 3
Enter the second number: 0
Invalid input. Please try again.
Enter the first number: 1
Enter the second number: 4
x / y is 0.25
```

图 18：else 子句使用示例

使用 `except Exception as e`，就可利用 8.3.4 节介绍的技巧在这个小型除法程序中打印更有用的错误消息


```
In [26]: while True:
        try:
            x = int(input('Enter the first number: '))
            y = int(input('Enter the second number: '))
            value = x / y
            print('x / y is', value)
        except Exception as e:
            print('Invalid input:', e)
            print('Please try again')
        else:
            break

Enter the first number: 1
Enter the second number: g
Invalid input: invalid literal for int() with base 10: 'g'
Please try again
Enter the first number: 1
Enter the second number: 0
Invalid input: division by zero
Please try again
Enter the first number: 1
Enter the second number: 4
x / y is 0.25
```

图 19: else 子句结合 except Exception as e 使用示例

1.4.8 finally 子句

用于在发生异常时执行清理工作。这个子句是与 try 子句配套的。

不管 try 子句中发生什么异常，都将执行 finally 子句。

```
In [27]: x = None
        try:
            x = 1 / 0
        finally:
            print('Cleaning up ...')
            del x

Cleaning up ...

ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-27-297133942295> in <module>
      1 x = None
      2 try:
--> 3     x = 1 / 0
      4 finally:
      5     print('Cleaning up ...')

ZeroDivisionError: division by zero
```

```
In [30]: try:
        1 / 1
    except NameError:
        print("Unknown variable")
    else:
        print("That went well!")
    finally:
        print("Cleaning up.")
```

That went well!
Cleaning up.

图 20: finally 子句使用示例

2 迭代器和生成器

2.1 迭代器、可迭代对象

- **迭代**是重复反馈过程的活动，其目的通常是为了接近并到达所需的目标或结果。每一次对过程的重复被称为一次“迭代”，而每一次迭代得到的结果会被用来作为下一次迭代的初始值。
- 可迭代对象不一定是迭代器，迭代器一定是可迭代对象。因为迭代器一定会实现 `__iter__` 方法，而可迭代对象尽管实现了 `__iter__` 也不一定实现 `__next__` 方法
- 迭代器协议是指：对象需要提供 `next` 方法，它要么返回迭代中的下一项，要么就引起一个 `StopIteration` 异常，以终止迭代
- 可迭代对象就是：实现了迭代器协议的对象
- 协议是一种约定，可迭代对象实现迭代器协议，Python 的内置工具(如 `for` 循环，`sum`，`min`，`max` 函数等)使用迭代器协议访问对象。

iter 方法:

- 方法 `iter` 返回一个迭代器，它是包含方法 `next` 的对象，而调用这个方法时可不提供任何参数。
- 当你调用方法 `next` 时，迭代器应返回其下一个值。
- 如果迭代器没有可供返回的值，应引发 `StopIteration` 异常。

```
In [37]: class Fibs:
        def __init__(self):
            self.a = 0
            self.b = 1
        def __next__(self):
            self.a, self.b = self.b, self.a + self.b
            return self.a
        def __iter__(self):
            # print(self.b)
            return self
```

```
In [4]: fibs = Fibs()
        for f in fibs:
            print(f, end=" ")
            if f > 1000:
                print("=====")
                print(f)
                break
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 =====
1597
```

图 21: iter 使用示例

注意到这个迭代器实现了方法 `iter`，而这个方法返回迭代器本身。在很多情况下，都在另一个对象中实现返回迭代器的方法 `iter`，并在 `for` 循环中使用这个对象。但推荐在迭代器中也实现方法 `iter`（并像刚才那样让它返回 `self`），这样迭代器就可直接用于 `for` 循环中。

2.2 初始化一个迭代器

```
In [10]: a=[1, 2, 3, 5, 6, 7, 8, 9]
         b=iter(a)

In [11]: b
Out[11]: <list_iterator at 0x2d64bb030f0>

In [12]: for x in b:
         print(x, end=" ")

1 2 3 5 6 7 8 9
```

图 22：初始化迭代器示例

2.3 使迭代器输出下一个迭代值

```
In [81]: a=[1, 2, 3, 5, 6, 7, 8, 9]
         b=iter(a)

In [82]: b.__next__()
Out[82]: 1

In [83]: next(b)
Out[83]: 2
```

图 23：使迭代器输出下一个迭代值示例

2.4 迭代器转序列

```
In [84]: list(b)
Out[84]: [3, 5, 6, 7, 8, 9]

In [40]: list(range(10))
Out[40]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [41]: [x for x in range(10)]
Out[41]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

图 24：迭代器转序列示例

2.5 生成器

生成器的主要思想：对于可以公式自动生成的数字序列，由计算机不断迭代，每次只生成一个数字，从而通过循环遍历生成序列中的所有元素。所以说，生成器产生的不是一个静态的值（比如类似字符串、元组和列表等，都是一次性生成所有值），而是一个动态的数据流。

```
In [105]: #生成器表达式
a = (x**2 for x in range(1,9))
aa = [x**2 for x in range(1,9)]
print(a)
print(aa)

<generator object <genexpr> at 0x0000013EB60A0660>
[1, 4, 9, 16, 25, 36, 49, 64]
```

```
In [98]: next(a)
```

```
Out[98]: 1
```

图 25: 生成器示例

函数体中有关键字 `yield`。`yield` 关键字类似于 `return`，当生成器被 `next()` 函数调用时，会返回其后的变量，相当于程序中断；当再次调用 `next()` 函数后，生成器会从中断的 `yield` 语句处继续执行，也就是用多少，取多少，不占内存。

```
In [42]: #生成器函数
def gen(x):
    x+=1
    yield x**2
```

```
In [43]: b=gen(0)
b
```

```
Out[43]: <generator object gen at 0x000001CD00F820B0>
```

```
In [44]: next(b)
```

```
Out[44]: 1
```

```
In [45]: next(b)
```

```
StopIteration                                Traceback (most recent call last)
<ipython-input-45-adb3e17b0219> in <module>
----> 1 next(b)

StopIteration:
```

```
In [56]: #生成器函数
def gen(x):
    while x>0:
        x+=1
        yield x**2
```

```
In [57]: b = gen(1)
b
```

```
Out[57]: <generator object gen at 0x000001CD01898660>
```

```
In [58]: next(b)
```

```
Out[58]: 4
```

```
In [59]: next(b)
```

```
Out[59]: 9
```

图 26: `yield` 使用示例

可以看到，变量 **a** 和 **b** 都是生成器，我们不能直接使用 **a**、**b**，因为它们实际上保存的是一个公式，使用时可以调用内置函数 `next()`，由 `next(a)`、`next(b)` 来动态生成序列中的下一个值。采用生成器的好处是：节省内存空间，特别是对于数据量大的序列，一次性生成所有值将会耗费大量内存，而采用生成器可以极大地节省存储空间。同时，生成器还可以处理无限长的序列。比如，上述实例中，变量 **b** 就是一个无限序列，理论上可以永远 `next(b)`，而且每次都是按顺序生成其中的一个值。

2.6 生成器的使用

2.6.1 for

可以通过 `for` 循环遍历所有值，示例如下：

```
In [13]: def flatten(nested):
        for sublist in nested:
            for element in sublist:
                yield element

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

        for x in flatten(nested):
            print(x, end=" ")

1 2 3 4 5

In [14]: flatten(nested)

Out[14]: <generator object flatten at 0x000002D64BACA480>

In [15]: list(flatten(nested))

Out[15]: [1, 2, 3, 4, 5]
```

图 27: `for` 循环遍历示例

2.6.2 next()

用内置函数 `next()` 循环生成下一个值，示例如下：

```
In [63]: # next()
        a = (x**2 for x in range(1,4))

In [64]: next(a)

Out[64]: 1

In [65]: next(a)

Out[65]: 4

In [66]: next(a)

Out[66]: 9

In [67]: next(a)

StopIteration                                Traceback (most recent call last)
<ipython-input-67-15841f3f11d4> in <module>
----> 1 next(a)

StopIteration:
```

图 28: `next()` 使用示例

2.6.3 `__next__()`

用生成器自身方法`__next__()`循环生成下一个值，示例如下：

```
In [68]: # __next__()
a = (x**2 for x in range(1,4))

In [69]: a.__next__()
Out[69]: 1

In [70]: a.__next__()
Out[70]: 4

In [71]: a.__next__()
Out[71]: 9

In [72]: a.__next__()

StopIteration                                Traceback (most recent call last)
<ipython-input-72-d34d2a8c0899> in <module>
----> 1 a.__next__()

StopIteration:
```

图 29: `__next__()`使用示例

- 通常访问生成器元素的较常用的方法就是采用 `for` 循环，`next()`方法极少使用。因为采用 `for` 循环不需要关心 `StopIteration` 异常
- 内置函数 `next()`和方法`__next__()`运行机制是相同的，因为内置函数 `next()`实际上就是调用了生成器自身方法`__next__()`
- 生成器只能遍历一次，生成最后一个元素后，再次调用 `next()`或`__next__()`会抛出 `StopIteration` 异常

2.6.4 `next()`与 `send()`

对于普通的生成器，第一个 `next` 调用，相当于启动生成器，会从生成器函数的第一行代码开始执行，直到第一次执行完 `yield` 语句（第 4 行）后，跳出生成器函数；然后第二个 `next` 调用，进入生成器函数后，从 `yield` 语句的下一句语句（第 5 行）开始执行，然后重新运行到 `yield` 语句，执行后，跳出生成器函数。对于这个参考示例如下：

```
In [73]: def consumer():
r = 'here'
for i in range(3):
    yield r
    r = '200 OK' + str(i)

In [74]: c=consumer()

In [75]: c1=next(c)

In [76]: c2=next(c)

In [77]: c3=next(c)

In [78]: print(c1)
print(c2)
print(c3)

here
200 OK0
200 OK1
```


图 30: 普通生成器的 next()调用示例

next()与 send()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, 我们可以看做 c.next() 和 c.send(None) 作用是一样的。需要提醒的是, 第一次调用时, 请使用 next()语句或是 send(None), 不能使用 send 发送一个非 None 的值, 否则会出错的, 因为没有 Python yield 语句来接收这个值

```
In [138]: def consumer():
          r='here'
          while True:
              n1 = yield r
              if not n1:
                  return
              print('[consumer] consuming %s...' % n1)
              r='200 OK'+str(n1)

          def produce(c):
              aa=c.send(None)
              n=0
              while n < 5:
                  n = n + 1
                  print('[produce] producing %s...' % n)
                  r1=c.send(n)
                  print('[produce] producing %s...' % r1)
              c.close()
```

```
In [139]: c=consumer()
          produce(c)

[produce] producing 1...
[consumer] consuming 1...
[produce] producing 200 OK1...
[produce] producing 2...
[consumer] consuming 2...
[produce] producing 200 OK2...
[produce] producing 3...
[consumer] consuming 3...
[produce] producing 200 OK3...
[produce] producing 4...
[consumer] consuming 4...
[produce] producing 200 OK4...
[produce] producing 5...
[consumer] consuming 5...
[produce] producing 200 OK5...
```

图 31: send()使用示例

当第一次 send (None) (对应 11 行) 时, 启动生成器, 从生成器函数的第一行代码开始执行, 直到第一次执行完 yield (对应第 4 行) 后, 跳出生成器函数。这个过程中, n1 一直没有定义。先进行 yield r 操作把 r 返回给 aa, yield 之后的操作 (n1 的赋值操作不进行) 下面运行到 send (1) 时, 进入生成器函数, 注意这里与调用 next 的不同。这里是从第 4 行开始执行, 把 1 赋值给 n1, 但是并不执行 yield 部分。下面继续从 yield 的下一语句继续执行, 然后重新运行到 yield 语句, 执行后, 跳出生成器函数。

即 send 和 next 相比, 只是开始多了一次赋值的动作, 其他运行流程是相同的。

2.6.5 throw 和 close

throw 有两方面的作用, 首先是抛给生成器一个异常, 然后如果生成器能处理掉异常的话, throw 方法接着迭代一次取得返回值。

```
In [79]: def gen_func():
        try:
            yield 1
        except Exception as e:
            print(e)
        yield 2
        yield 3
        yield 4
        yield 5
        return 'hhhhh'
```

```
In [84]: gen=gen_func()
        print(next(gen))

1
```

```
In [85]: a=gen.throw(Exception,'throw exception')

throw exception
```

```
In [86]: print(a)

2
```

```
In [87]: print(next(gen)) #是3不是2

3
```

图 32: throw()使用示例

close, 他只有一个作用, 就是向生成器抛出 `GeneratorExit` 异常。

```
In [88]: def Gen():
        try:
            yield 1
            yield 2
        except GeneratorExit:
            print('close:GeneratorExit!!!')
```

```
In [ ]: gen1=Gen()
```

```
In [90]: print(next(gen1))
        print(next(gen1))

close:GeneratorExit!!!
1
2
```

```
In [92]: gen1=Gen()
        gen1.close()
        # print(next(gen1))
```

```
StopIteration                                Traceback (most recent call last)
<ipython-input-92-c61134e74f3e> in <module>
      1 gen1=Gen()
      2 gen1.close()
--> 3 print(next(gen1))
      4 # print(next(gen1))

StopIteration:
```

图 33: close()使用示例