

29 | 巧用上下文管理器和With语句精简代码

2019-07-15 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 08:30 大小 7.79M



你好，我是景霄。


我想你对 Python 中的 with 语句一定不陌生，在专栏里它也曾多次出现，尤其是在文件的输入输出操作中，不过我想，大部分人可能习惯了它的使用，却并不知道隐藏在其背后的“秘密”。

那么，究竟 with 语句要怎么用，与之相关的上下文管理器（context manager）是什么，它们之间又有着怎样的联系呢？这节课，我就带你一起揭开它们的神秘面纱。

什么是上下文管理器？


在任何一门编程语言中，文件的输入输出、数据库的连接断开等，都是很常见的资源管理操作。但资源都是有限的，在写程序时，我们必须保证这些资源在使用过后得到释放，不然就容易造成资源泄露，轻者使得系统处理缓慢，重则会使系统崩溃。

光说这些概念，你可能体会不到这一点，我们可以看看下面的例子：

 复制代码

```
1 for x in range(10000000):
2     f = open('test.txt', 'w')
3     f.write('hello')
```


这里我们一共打开了 10000000 个文件，但是用完以后都没有关闭它们，如果你运行该段代码，便会报错：

 复制代码

```
1 OSError: [Errno 23] Too many open files in system: 'test.txt'
```

这就是一个典型的资源泄露的例子。因为程序中同时打开了太多的文件，占据了太多的资源，造成系统崩溃。

为了解决这个问题，不同的编程语言都引入了不同的机制。而在 Python 中，对应的解决方案便是上下文管理器（context manager）。上下文管理器，能够帮助你自动分配并且释放资源，其中最典型的应用便是 with 语句。所以，上面代码的正确写法应该如下所示：

 复制代码

```
1 for x in range(10000000):
2     with open('test.txt', 'w') as f:
3         f.write('hello')
```

这样，我们每次打开文件“test.txt”，并写入‘hello’之后，这个文件便会自动关闭，相应的资源也可以得到释放，防止资源泄露。当然，with 语句的代码，也可以用下面的形式表示：

```
1 f = open('test.txt', 'w')
2 try:
3     f.write('hello')
4 finally:
5     f.close()
```

要注意的是，最后的 finally block 尤其重要，哪怕在写入文件时发生错误异常，它也可以保证该文件最终被关闭。不过与 with 语句相比，这样的代码就显得冗余了，并且还容易漏写，因此我们一般更倾向于使用 with 语句。

另外一个典型的例子，是 Python 中的 threading.lock 类。举个例子，比如我想要获取一个锁，执行相应的操作，完成后再释放，那么代码就可以写成下面这样：

```
1 some_lock = threading.Lock()
2 some_lock.acquire()
3 try:
4     ...
5 finally:
6     some_lock.release()
```

而对应的 with 语句，同样非常简洁：


```
1 some_lock = threading.Lock()
2 with somelock:
3     ...
```

我们可以从这两个例子中看到，with 语句的使用，可以简化了代码，有效避免资源泄露的发生。

上下文管理器的实现

基于类的上下文管理器


了解了上下文管理的概念和优点后，下面我们就通过具体的例子，一起来看看上下文管理器的原理，搞清楚它的内部实现。这里，我自定义了一个上下文管理类 `FileManager`，模拟 Python 的打开、关闭文件操作：

 复制代码

```
1 class FileManager:
2     def __init__(self, name, mode):
3         print('calling __init__ method')
4         self.name = name
5         self.mode = mode
6         self.file = None
7
8     def __enter__(self):
9         print('calling __enter__ method')
10        self.file = open(self.name, self.mode)
11        return self.file
12
13
14    def __exit__(self, exc_type, exc_val, exc_tb):
15        print('calling __exit__ method')
16        if self.file:
17            self.file.close()
18
19 with FileManager('test.txt', 'w') as f:
20     print('ready to write to file')
21     f.write('hello world')
22
23 ## 输出
24 calling __init__ method
25 calling __enter__ method
26 ready to write to file
27 calling __exit__ method
```

需要注意的是，当我们用类来创建上下文管理器时，必须保证这个类包括方法“`__enter__()`”和方法“`__exit__()`”。其中，方法“`__enter__()`”返回需要被管理的资源，方法“`__exit__()`”里通常会存在一些释放、清理资源的操作，比如这个例子中的关闭文件等等。

而当我们用 `with` 语句，执行这个上下文管理器时：

 复制代码


```
1 with FileManager('test.txt', 'w') as f:
```

```
2 f.write('hello world')
```

下面这四步操作会依次发生：

1. 方法“`__init__()`”被调用，程序初始化对象 `FileManager`，使得文件名（`name`）是“`test.txt`”，文件模式（`mode`）是“`w`”；
2. 方法“`__enter__()`”被调用，文件“`test.txt`”以写入的模式被打开，并且返回 `FileManager` 对象赋予变量 `f`；
3. 字符串“`hello world`”被写入文件“`test.txt`”；
4. 方法“`__exit__()`”被调用，负责关闭之前打开的文件流。

因此，这个程序的输出是：

 复制代码

```
1 calling __init__ method
2 calling __enter__ method
3 ready to write to file
4 calling __exit__ meth
```

另外，值得一提的是，方法“`__exit__()`”中的参数“`exc_type, exc_val, exc_tb`”，分别表示 `exception_type`、`exception_value` 和 `traceback`。当我们执行含有上下文管理器的 `with` 语句时，如果有异常抛出，异常的信息就会包含在这三个变量中，传入方法“`__exit__()`”。

因此，如果你需要处理可能发生的异常，可以在“`__exit__()`”添加相应的代码，比如下面这样来写：

 复制代码

```
1 class Foo:
2     def __init__(self):
3         print('__init__ called')
4
5     def __enter__(self):
6         print('__enter__ called')
7         return self
```


```

8
9     def __exit__(self, exc_type, exc_value, exc_tb):
10         print('__exit__ called')
11         if exc_type:
12             print(f'exc_type: {exc_type}')
13             print(f'exc_value: {exc_value}')
14             print(f'exc_traceback: {exc_tb}')
15             print('exception handled')
16         return True
17
18 with Foo() as obj:
19     raise Exception('exception raised').with_traceback(None)
20
21 # 输出
22 __init__ called
23 __enter__ called
24 __exit__ called
25 exc_type: <class 'Exception'>
26 exc_value: exception raised
27 exc_traceback: <traceback object at 0x1046036c8>
28 exception handled

```

这里，我们在 with 语句中手动抛出了异常 “exception raised”，你可以看到，“__exit__ ()”方法中异常，被顺利捕捉并进行了处理。不过需要注意的是，如果方法“__exit__ ()”没有返回 True，异常仍然会被抛出。因此，如果你确定异常已经被处理了，请在“__exit__ ()”的最后，加上“return True”这条语句。

同样的，数据库的连接操作，也常常用上下文管理器来表示，这里我给出了比较简化的代码：

 复制代码

```

1 class DBConnectionManager:
2     def __init__(self, hostname, port):
3         self.hostname = hostname
4         self.port = port
5         self.connection = None
6
7     def __enter__(self):
8         self.connection = DBClient(self.hostname, self.port)
9         return self
10
11     def __exit__(self, exc_type, exc_val, exc_tb):
12         self.connection.close()
13

```

```
14 with DBConnectionManager('localhost', '8080') as db_client:
```

与前面 FileManager 的例子类似：

方法“__init__()”负责对数据库进行初始化，也就是将主机名、接口（这里是 localhost 和 8080）分别赋予变量 hostname 和 port；

方法“__enter__()”连接数据库，并且返回对象 DBConnectionManager；


方法“__exit__()”则负责关闭数据库的连接。

这样一来，只要你写完了 DBconnectionManager 这个类，那么在程序每次连接数据库时，我们都只需要简单地调用 with 语句即可，并不需要关心数据库的关闭、异常等等，显然大大提高了开发的效率。

基于生成器的上下文管理器

诚然，基于类的上下文管理器，在 Python 中应用广泛，也是我们经常看到的形式，不过 Python 中的上下文管理器并不局限于此。除了基于类，它还可以基于生成器实现。接下来我们来看一个例子。

比如，你可以使用装饰器 contextlib.contextmanager，来定义自己所需的基于生成器的上下文管理器，用以支持 with 语句。还是拿前面的类上下文管理器 FileManager 来说，我们也可以用下面形式来表示：

 复制代码

```
1 from contextlib import contextmanager
2
3 @contextmanager
4 def file_manager(name, mode):
5     try:
6         f = open(name, mode)
7         yield f
8     finally:
9         f.close()
10
11 with file_manager('test.txt', 'w') as f:
12     f.write('hello world')
```

这段代码中，函数 `file_manager()` 是一个生成器，当我们执行 `with` 语句时，便会打开文件，并返回文件对象 `f`；当 `with` 语句执行完后，`finally block` 中的关闭文件操作便会执行。

你可以看到，使用基于生成器的上下文管理器时，我们不再用定义“`__enter__()`”和“`__exit__()`”方法，但请务必加上装饰器 `@contextmanager`，这一点新手很容易疏忽。

讲完这两种不同原理的上下文管理器后，还需要强调的是，基于类的上下文管理器和基于生成器的上下文管理器，这两者在功能上是一致的。只不过，

基于类的上下文管理器更加 flexible，适用于大型的系统开发；

而基于生成器的上下文管理器更加方便、简洁，适用于中小型程序。

无论你使用哪一种，请不用忘记在方法“`__exit__()`”或者是 `finally block` 中释放资源，这一点尤其重要。

总结

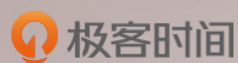
这节课，我们先通过一个简单的例子，了解了资源泄露的易发生性，和其带来的严重后果，从而引入了应对方案——即上下文管理器的概念。上下文管理器，通常应用在文件的打开关闭和数据库的连接关闭等场景中，可以确保用过的资源得到迅速释放，有效提高了程序的安全性，

接着，我们通过自定义上下文管理的实例，了解了上下文管理工作的原理，并一起学习了基于类的上下文管理器和基于生成器的上下文管理器，这两者的功能相同，具体用哪个，取决于你的具体使用场景。

另外，上下文管理器通常和 `with` 语句一起使用，大大提高了程序的简洁度。需要注意的是，当我们用 `with` 语句执行上下文管理器的操作时，一旦有异常抛出，异常的类型、值等具体信息，都会通过参数传入“`__exit__()`”函数中。你可以自行定义相关的操作对异常进行处理，而处理完异常后，也别忘了加上“`return True`”这条语句，否则仍然会抛出异常。

思考题

那么，在你日常的学习工作中，哪些场景使用过上下文管理器？使用过程中又遇到了哪些问题，或是有什么新的发现呢？欢迎在下方留言与我讨论，也欢迎你把这篇文章分享出去，我们一起交流，一起进步。



Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | 如何合理利用assert？

精选留言 (10)

写留言



new

2019-07-15

打开文件时用更方便

展开 ∨



2



enjoylearning

2019-07-15

主要用于数据库连接

展开 ▾



👍 2



AllenGFLiu

2019-07-15

第一次有教程提到这个上下文管理器，学习。
对知识的学习就是需要从多角度重复去看，在这个过程中查漏补缺，才能保持不断进步。

展开 ▾

作者回复: 嗯嗯，是这样的



👍 1



大牛凯

2019-07-15

老师好，请问基于类的上下文，“__enter__”方法什么时候返回self呢？
DBConnectionManager的例子中可以说明一下为什么是返回self不是返回self.connection么？



Geek_d848f7

2019-07-15

还是不怎么清楚基于生成器的上下文管理器的运行过程

展开 ▾

作者回复: 哪里不明白？文章中应该讲的很清楚了



Geek_d848f7

2019-07-15

利用邮箱发邮件中使用到了，但文中提到的基于生成器的上下文管理第一次听过，这个的使用是不是主要在于需要关注资源是否被释放的生成器中呢？

展开 ▾





程序员人生

2019-07-15

老师，最后一段代码执行后报这个错：

TypeError: file_manager() missing 1 required positional argument: 'mode'

应该写成这样把，

with file_manager('test.txt','w') as f:...

展开 ▾

作者回复: 谢谢指正，已更新



倾

2019-07-15

只是在文件操作时使用，今天第一次学习到还能这么用。



ZJY

2019-07-15

http的session是上下文管理器吗？

展开 ▾



安排

2019-07-15

是不是只有程序出了with代码块，管理的对象才会析构，也就是释放资源？

作者回复: 准确的说释放资源的行为发生在with语句的最后（上下文管理器的__exit__函数内）

