**with术语**

要使用 with 语句，首先要明白上下文管理器这一概念。有了上下文管理器，with 语句才能工作。

下面是一组与上下文管理器和with 语句有关的概念。

**上下文管理协议（Context Management Protocol）**：包含方法 \_\_enter\_\_() 和 \_\_exit\_\_()，支持

该协议的对象要实现这两个方法。

**上下文管理器（Context Manager）**：支持上下文管理协议的对象，这种对象实现了

\_\_enter\_\_() 和 \_\_exit\_\_() 方法。上下文管理器定义执行 with 语句时要建立的运行时上下文，

负责执行 with 语句块上下文中的进入与退出操作。通常使用 with 语句调用上下文管理器，

也可以通过直接调用其方法来使用。

**运行时上下文（runtime context）**：由上下文管理器创建，通过上下文管理器的 \_\_enter\_\_() 和

\_\_exit\_\_() 方法实现，\_\_enter\_\_() 方法在语句体执行之前进入运行时上下文，\_\_exit\_\_() 在

语句体执行完后从运行时上下文退出。with 语句支持运行时上下文这一概念。

**上下文表达式（Context Expression）**：with 语句中跟在关键字 with 之后的表达式，该表达式

要返回一个上下文管理器对象。

**语句体（with-body）**：with 语句包裹起来的代码块，在执行语句体之前会调用上下文管

理器的 \_\_enter\_\_() 方法，执行完语句体之后会执行 \_\_exit\_\_() 方法。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-pythonwith/#ibm-pcon)

**基本语法和工作原理**

with 语句的语法格式如下：

**清单 1. with 语句的语法格式**

with context\_expression [as target(s)]:

with-body

这里 context\_expression 要返回一个上下文管理器对象，该对象并不赋值给 as 子句中的 target(s) ，如果指定了 as 子句的话，会将上下文管理器的 \_\_enter\_\_() 方法的返回值赋值给 target(s)。target(s) 可以是单个变量，或者由“()”括起来的元组（不能是仅仅由“,”分隔的变量列表，必须加“()”）。

Python 对一些内建对象进行改进，加入了对上下文管理器的支持，可以用于 with 语句中，比如可以自动关闭文件、线程锁的自动获取和释放等。假设要对一个文件进行操作，使用 with 语句可以有如下代码：

**清单 2. 使用 with 语句操作文件对象**

with open(r'somefileName') as somefile:

for line in somefile:

print line

# ...more code

这里使用了 with 语句，不管在处理文件过程中是否发生异常，都能保证 with 语句执行完毕后已经关闭了打开的文件句柄。如果使用传统的 try/finally 范式，则要使用类似如下代码：

**清单 3. try/finally 方式操作文件对象**

somefile = open(r'somefileName')

try:

for line in somefile:

print line

# ...more code

finally:

somefile.close()

比较起来，使用 with 语句可以减少编码量。已经加入对上下文管理协议支持的还有模块 threading、decimal 等。

[PEP 0343](http://www.python.org/dev/peps/pep-0343/)对 with 语句的实现进行了描述。with 语句的执行过程类似如下代码块：

**清单 4. with 语句执行过程**

context\_manager = context\_expression

exit = type(context\_manager).\_\_exit\_\_

value = type(context\_manager).\_\_enter\_\_(context\_manager)

exc = True # True 表示正常执行，即便有异常也忽略；False 表示重新抛出异常，需要对异常进行处理

try:

try:

target = value # 如果使用了 as 子句

with-body # 执行 with-body

except:

# 执行过程中有异常发生

exc = False

# 如果 \_\_exit\_\_ 返回 True，则异常被忽略；如果返回 False，则重新抛出异常

# 由外层代码对异常进行处理

if not exit(context\_manager, \*sys.exc\_info()):

raise

finally:

# 正常退出，或者通过 statement-body 中的 break/continue/return 语句退出

# 或者忽略异常退出

if exc:

exit(context\_manager, None, None, None)

# 缺省返回 None，None 在布尔上下文中看做是 False

1. 执行 context\_expression，生成上下文管理器 context\_manager
2. 调用上下文管理器的 \_\_enter\_\_() 方法；如果使用了 as 子句，则将 \_\_enter\_\_() 方法的返回值赋值给 as 子句中的 target(s)
3. 执行语句体 with-body
4. 不管是否执行过程中是否发生了异常，执行上下文管理器的 \_\_exit\_\_() 方法，\_\_exit\_\_() 方法负责执行“清理”工作，如释放资源等。如果执行过程中没有出现异常，或者语句体中执行了语句 break/continue/return，则以 None 作为参数调用 \_\_exit\_\_(None, None, None) ；如果执行过程中出现异常，则使用 sys.exc\_info 得到的异常信息为参数调用 \_\_exit\_\_(exc\_type, exc\_value, exc\_traceback)
5. 出现异常时，如果 \_\_exit\_\_(type, value, traceback) 返回 False，则会重新抛出异常，让with 之外的语句逻辑来处理异常，这也是通用做法；如果返回 True，则忽略异常，不再对异常进行处理

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-pythonwith/#ibm-pcon)

**自定义上下文管理器**

开发人员可以自定义支持上下文管理协议的类。自定义的上下文管理器要实现上下文管理协议所需要的 \_\_enter\_\_() 和 \_\_exit\_\_() 两个方法：

* **context\_manager.\_\_enter\_\_()** ：进入上下文管理器的运行时上下文，在语句体执行前调用。with 语句将该方法的返回值赋值给 as 子句中的 target，如果指定了 as 子句的话
* **context\_manager.\_\_exit\_\_(exc\_type, exc\_value, exc\_traceback)** ：退出与上下文管理器相关的运行时上下文，返回一个布尔值表示是否对发生的异常进行处理。参数表示引起退出操作的异常，如果退出时没有发生异常，则3个参数都为None。如果发生异常，返回

True 表示不处理异常，否则会在退出该方法后重新抛出异常以由 with 语句之外的代码逻辑进行处理。如果该方法内部产生异常，则会取代由 statement-body 中语句产生的异常。要处理异常时，不要显示重新抛出异常，即不能重新抛出通过参数传递进来的异常，只需要将返回值设置为 False 就可以了。之后，上下文管理代码会检测是否 \_\_exit\_\_() 失败来处理异常

下面通过一个简单的示例来演示如何构建自定义的上下文管理器。注意，上下文管理器必须同时提供 \_\_enter\_\_() 和 \_\_exit\_\_() 方法的定义，缺少任何一个都会导致 AttributeError；with 语句会先检查是否提供了 \_\_exit\_\_() 方法，然后检查是否定义了 \_\_enter\_\_() 方法。

假设有一个资源 DummyResource，这种资源需要在访问前先分配，使用完后再释放掉；分配操作可以放到 \_\_enter\_\_() 方法中，释放操作可以放到 \_\_exit\_\_() 方法中。简单起见，这里只通过打印语句来表明当前的操作，并没有实际的资源分配与释放。

**清单 5. 自定义支持 with 语句的对象**

class DummyResource:

def \_\_init\_\_(self, tag):

self.tag = tag

print 'Resource [%s]' % tag

def \_\_enter\_\_(self):

print '[Enter %s]: Allocate resource.' % self.tag

return self # 可以返回不同的对象

def \_\_exit\_\_(self, exc\_type, exc\_value, exc\_tb):

print '[Exit %s]: Free resource.' % self.tag

if exc\_tb is None:

print '[Exit %s]: Exited without exception.' % self.tag

else:

print '[Exit %s]: Exited with exception raised.' % self.tag

return False # 可以省略，缺省的None也是被看做是False

DummyResource 中的 \_\_enter\_\_() 返回的是自身的引用，这个引用可以赋值给 as 子句中的 target 变量；返回值的类型可以根据实际需要设置为不同的类型，不必是上下文管理器对象本身。

\_\_exit\_\_() 方法中对变量 exc\_tb 进行检测，如果不为 None，表示发生了异常，返回 False 表示需要由外部代码逻辑对异常进行处理；注意到如果没有发生异常，缺省的返回值为 None，在布尔环境中也是被看做 False，但是由于没有异常发生，\_\_exit\_\_() 的三个参数都为 None，上下文管理代码可以检测这种情况，做正常处理。

下面在 with 语句中访问 DummyResource ：

**清单 6. 使用自定义的支持 with 语句的对象**

with DummyResource('Normal'):

print '[with-body] Run without exceptions.'

with DummyResource('With-Exception'):

print '[with-body] Run with exception.'

raise Exception

print '[with-body] Run with exception. Failed to finish statement-body!'

第1个 with 语句的执行结果如下：

**清单 7. with 语句1执行结果**

Resource [Normal]

[Enter Normal]: Allocate resource.

[with-body] Run without exceptions.

[Exit Normal]: Free resource.

[Exit Normal]: Exited without exception.

可以看到，正常执行时会先执行完语句体 with-body，然后执行 \_\_exit\_\_() 方法释放资源。

第2个 with 语句的执行结果如下：

**清单 8. with 语句2执行结果**

Resource [With-Exception]

[Enter With-Exception]: Allocate resource.

[with-body] Run with exception.

[Exit With-Exception]: Free resource.

[Exit With-Exception]: Exited with exception raised.

Traceback (most recent call last):

File "G:/demo", line 20, in <module>

raise Exception

Exception

可以看到，with-body 中发生异常时with-body 并没有执行完，但资源会保证被释放掉，同时产生的异常由 with 语句之外的代码逻辑来捕获处理。

可以自定义上下文管理器来对软件系统中的资源进行管理，比如数据库连接、共享资源的访问控制等。Python 在线文档 [Writing Context Managers](http://docs.python.org/release/2.6/whatsnew/2.6.html#module-contextlib) 提供了一个针对数据库连接进行管理的上下文管理器的简单范例。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-pythonwith/#ibm-pcon)

**contextlib 模块**

contextlib 模块提供了3个对象：装饰器 contextmanager、函数 nested 和上下文管理器 closing。使用这些对象，可以对已有的生成器函数或者对象进行包装，加入对上下文管理协议的支持，避免了专门编写上下文管理器来支持 with 语句。

**装饰器 contextmanager**

contextmanager 用于对生成器函数进行装饰，生成器函数被装饰以后，返回的是一个上下文管理器，其 \_\_enter\_\_() 和 \_\_exit\_\_() 方法由 contextmanager 负责提供，而不再是之前的迭代子。被装饰的生成器函数只能产生一个值，否则会导致异常 RuntimeError；产生的值会赋值给 as 子句中的 target，如果使用了 as 子句的话。下面看一个简单的例子。

**清单 9. 装饰器 contextmanager 使用示例**

from contextlib import contextmanager

@contextmanager

def demo():

print '[Allocate resources]'

print 'Code before yield-statement executes in \_\_enter\_\_'

yield '\*\*\* contextmanager demo \*\*\*'

print 'Code after yield-statement executes in \_\_exit\_\_'

print '[Free resources]'

with demo() as value:

print 'Assigned Value: %s' % value

结果输出如下：

**清单 10. contextmanager 使用示例执行结果**

[Allocate resources]

Code before yield-statement executes in \_\_enter\_\_

Assigned Value: \*\*\* contextmanager demo \*\*\*

Code after yield-statement executes in \_\_exit\_\_

[Free resources]

可以看到，生成器函数中 yield 之前的语句在 \_\_enter\_\_() 方法中执行，yield 之后的语句在 \_\_exit\_\_() 中执行，而 yield 产生的值赋给了 as 子句中的 value 变量。

需要注意的是，contextmanager 只是省略了 \_\_enter\_\_() / \_\_exit\_\_() 的编写，但并不负责实现资源的“获取”和“清理”工作；“获取”操作需要定义在 yield 语句之前，“清理”操作需要定义 yield 语句之后，这样 with 语句在执行 \_\_enter\_\_() / \_\_exit\_\_() 方法时会执行这些语句以获取/释放资源，即生成器函数中需要实现必要的逻辑控制，包括资源访问出现错误时抛出适当的异常。

**函数 nested**

nested 可以将多个上下文管理器组织在一起，避免使用嵌套 with 语句。

**清单 11. nested 语法**

with nested(A(), B(), C()) as (X, Y, Z):

# with-body code here

类似于：

**清单 12. nested 执行过程**

with A() as X:

with B() as Y:

with C() as Z:

# with-body code here

需要注意的是，发生异常后，如果某个上下文管理器的 \_\_exit\_\_() 方法对异常处理返回 False，则更外层的上下文管理器不会监测到异常。

**上下文管理器 closing**

closing 的实现如下：

**清单 13. 上下文管理 closing 实现**

class closing(object):

# help doc here

def \_\_init\_\_(self, thing):

self.thing = thing

def \_\_enter\_\_(self):

return self.thing

def \_\_exit\_\_(self, \*exc\_info):

self.thing.close()

上下文管理器会将包装的对象赋值给 as 子句的 target 变量，同时保证打开的对象在 with-body 执行完后会关闭掉。closing 上下文管理器包装起来的对象必须提供 close() 方法的定义，否则执行时会报 AttributeError 错误。

**清单 14. 自定义支持 closing 的对象**

class ClosingDemo(object):

def \_\_init\_\_(self):

self.acquire()

def acquire(self):

print 'Acquire resources.'

def free(self):

print 'Clean up any resources acquired.'

def close(self):

self.free()

with closing(ClosingDemo()):

print 'Using resources'

结果输出如下：

**清单 15. 自定义 closing 对象的输出结果**

Acquire resources.

Using resources

Clean up any resources acquired.

closing 适用于提供了 close() 实现的对象，比如网络连接、数据库连接等，也可以在自定义类时通过接口 close() 来执行所需要的资源“清理”工作。

[**回页首**](https://www.ibm.com/developerworks/cn/opensource/os-cn-pythonwith/#ibm-pcon)

**小结**

本文对 with 语句的语法和工作机理进行了介绍，并通过示例介绍了如何实现自定义的上下文管理器，最后介绍了如何使用 contextlib 模块来简化上下文管理器的编写。