# with语句的用法

Python对with的处理还很聪明。基本思想是with所求值的对象必须有一个\_\_enter\_\_()方法，一个\_\_exit\_\_()方法。

紧跟with后面的语句被求值后，返回对象的\_\_enter\_\_()方法被调用，这个方法的返回值将被赋值给as后面的变量。当with后面的代码块全部被执行完之后，将调用前面返回对象的\_\_exit\_\_()方法.

with真正强大之处是它可以处理异常.

如果一个对象没有实现上下文，我们就不能把它用于with语句。这个时候，可以用closing()来把该对象变为上下文对象。

不是只有open()函数返回的fp对象才能使用with语句。实际上，任何对象，只要正确实现了上下文管理，就可以用于with语句。

实现上下文管理是通过\_\_enter\_\_和\_\_exit\_\_这两个方法实现的，也可以通过contextlib中的@contextmanager这个装饰器来实现上下文管理

super(curClass, self).parMethod()//调用curClass父类的方法

# Python中的MRO方法解析顺序问题

详细地址：<http://python.jobbole.com/85685/>

两种类的差别：

新式类：class A(object):　默认继承于object，A.\_\_mro\_\_ 可以查看新式类的方法解析顺序

经典类：class　Ａ：

对于python2.2以前的版本，也就是经典类获取一个类的MRO顺序:

print inspect.getmro(A)

对于python2.2版本：

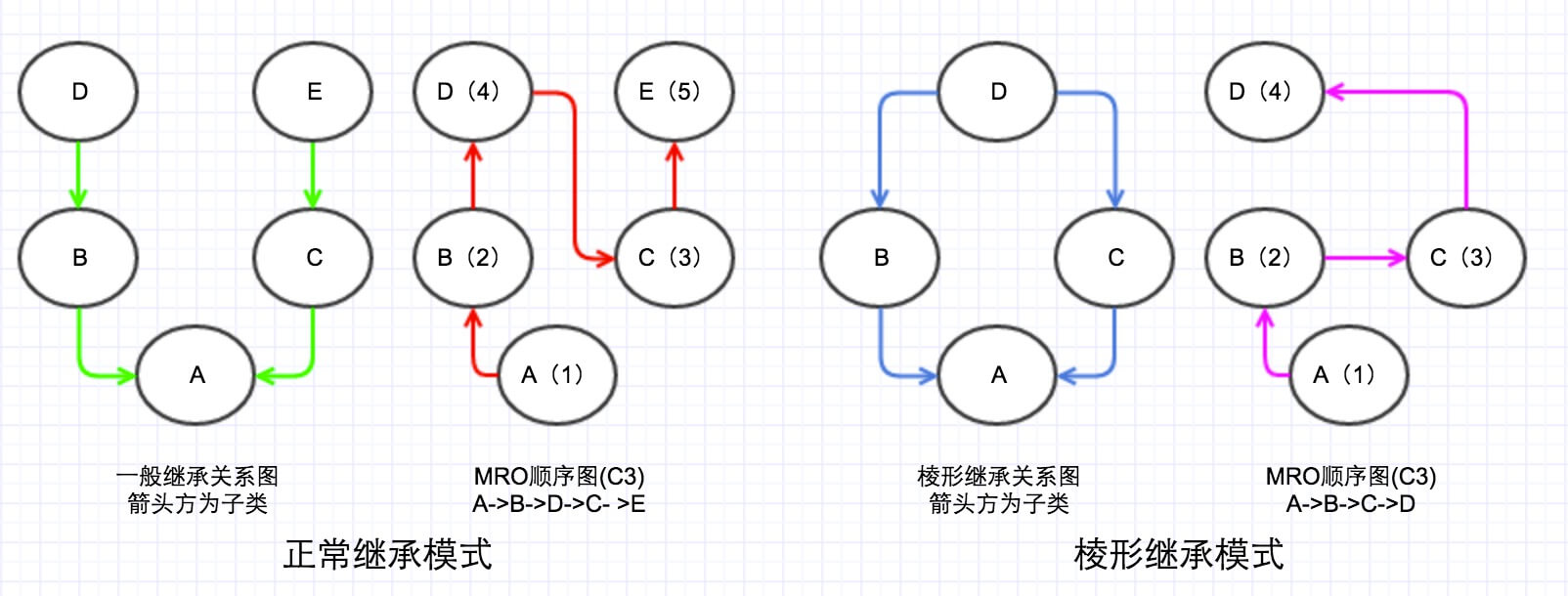
1. 如果是经典类MRO为DFS（深度优先搜索（子节点顺序：从左到右））。

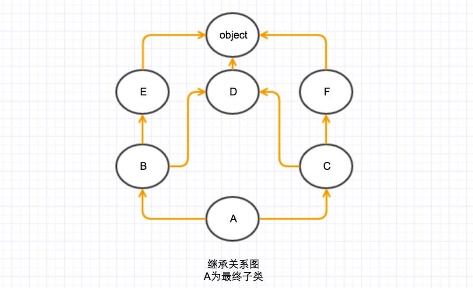
2. 如果是新式类MRO为BFS（广度优先搜索（子节点顺序：从左到右））。

Python2.3开始新式类的MRO取而代之的是C3算法

Python3只剩下新式类，采用的MRO也依旧是C3算法。

C3算法解决了单调性问题和只能继承无法重写问题，结合了DFS和BFS，搜索的过程如图所示



模拟一下例子的拓扑排序：首先找入度为0的点，只有一个A，把A拿出来，把A相关的边剪掉，再找下一个入度为0的点，有两个点（B,C）,取最左原则，拿B，这是排序是AB，然后剪B相关的边，这时候入度为0的点有E和C，取最左。这时候排序为ABE，接着剪E相关的边，这时只有一个点入度为0，那就是C，取C，顺序为ABEC。剪C的边得到两个入度为0的点（DF），取最左D，顺序为ABECD，然后剪D相关的边，那么下一个入度为0的就是F，然后是object。那么最后的排序就为ABECDFobject。

# 描述符

描述符基于三个必须实现的特殊方法：\_\_set\_\_ \_\_get\_\_ \_\_delete\_\_这些方法将在\_\_dict\_\_特性之前被调用

实现了\_\_set\_\_ \_\_get\_\_的称为数据描述符，只实现了后者的称为非数据描述符

描述符，用一句话来说，就是将某种特殊类型的类的实例指派给另一个类的属性(注意：这里是类属性，而不是对象属性)。

class UpperString(object):

"""docstring for UpperString"""

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

self.\_value = ''

def \_\_get\_\_(self, instance, klass):

return self.\_value

def \_\_set\_\_(self, instance, value):

self.\_value = value.upper()

class MyDesc(object):

"""docstring for UpperString"""

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

self.\_value = ''

def \_\_get\_\_(self, instance, klass):

return self.\_value

def \_\_set\_\_(self, instance, value):

self.\_value = value.upper()

class MyClass(object):

attribute = UpperString()

ins = MyClass()

print(ins.attribute)#空行

ins.attribute = "my value"

print(ins.attribute)#转换成大写

ins.\_\_dict\_\_ = {}

ins.new\_attr = 1

print(ins.\_\_dict\_\_)#{'new\_attr':1}

MyClass.new\_attr = MyDesc()

print(ins.\_\_dict\_\_)#{'new\_attr':1}

print(ins.new\_attr)#空行

ins.new\_attr = "other Value"

print(ins.new\_attr)#大写

print(ins.\_\_dict\_\_)#空行

除了隐藏类的内容这个角色，描述还有其他使用：

1. 内省描述符(Introspection descriptor) ：检查宿主类签名
2. 元描述符(Meta descriptor)：使用类方法本身完成值计算

# @property装饰器

其实现原理就是上面所说的描述符，可以将一个与属性同名的方法设置成该属性的setter具体方法如下：

# -\*- coding:utf-8 -\*-

class Student(object):

@property

def score(self):

return self.\_score

@score.setter

def score(self, value):

if not isinstance(value, int):

raise ValueError("score must be an integer")

if value < 0 or value > 100:

raise ValueError("score must between 0-100")

self.\_score = value

s = Student()

s.score = 60

print(s.score)

s.score = 600

print(s.score)