**前言：**

MRO（Method Resolution Order）方法解析顺序

Python语言包含了很多优秀的特性，其中多继承就是其中之一，但是如果如雷存在同名函数的时候还是会产生二义性，Python中处理这种问题的方法就是MRO

**历史中的MRO**

1. python2.2以前的版本：经典类时代。经典类时一种没有继承的类，实例类型都是type类型，如果经典类被作为父类，子类调用父类的构造函数的时候就会出错。这是的MRO方法是DFS(深度优先搜索（子节点顺序：从左到右）)

Class A:   # 是没有继承任何父类的

    def \_\_init\_\_(self):

        print "这是经典类"

所以我们在查看经典类的MRO顺序的时候是：

import inspect

class D:

pass

class C(D):

pass

class B(D):

pass

class A(B, C):

pass

if \_\_name\_\_ == '\_\_main\_\_':

print inspect.getmro(A)

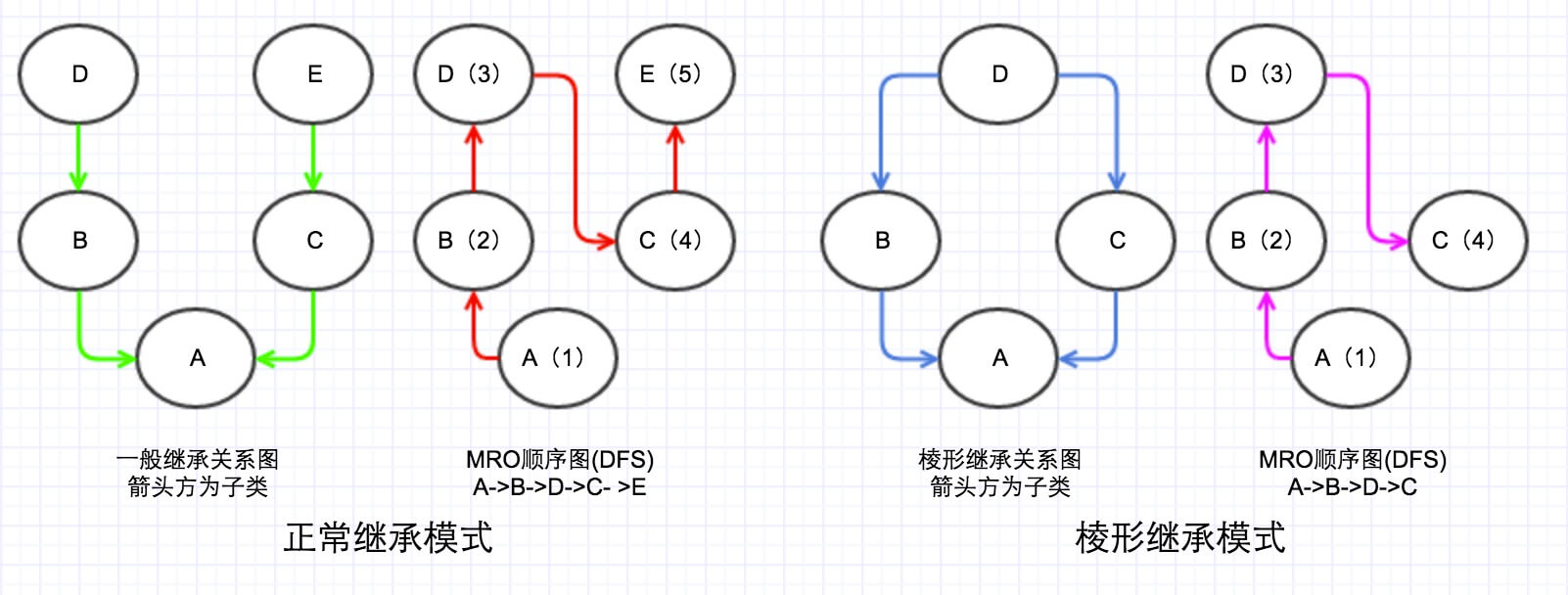
#输出：

>> (<class \_\_main\_\_.A at 0x10e0e5530>,

<class \_\_main\_\_.B at 0x10e0e54c8>,

<class \_\_main\_\_.D at 0x10e0e53f8>,

<class \_\_main\_\_.C at 0x10e0e5460>)



经典类有两种继承模式，这两种集成模式在DFS下有如下的优缺点：

第一种，正常继承模式。连个互不相关的类的多继承，DFS顺序正常，不会引起任何问题。

第二种，菱形继承模式。存在公共父类(D)的多继承，这种情况下DFS必然经过公共父类D，这时候想想，如果这个公共父类D有一些初始化的属性或者方法，但是子类C又重写了这些属性或者方法，那么按照DFS的顺序必定会是先找到D的属性或者方法，那么C的属性或者方法永远访问不到，导致C只能继承无法重写(override)这就是为什么新式类不使用DFS的原因，因为他们都有一个公共的祖先object

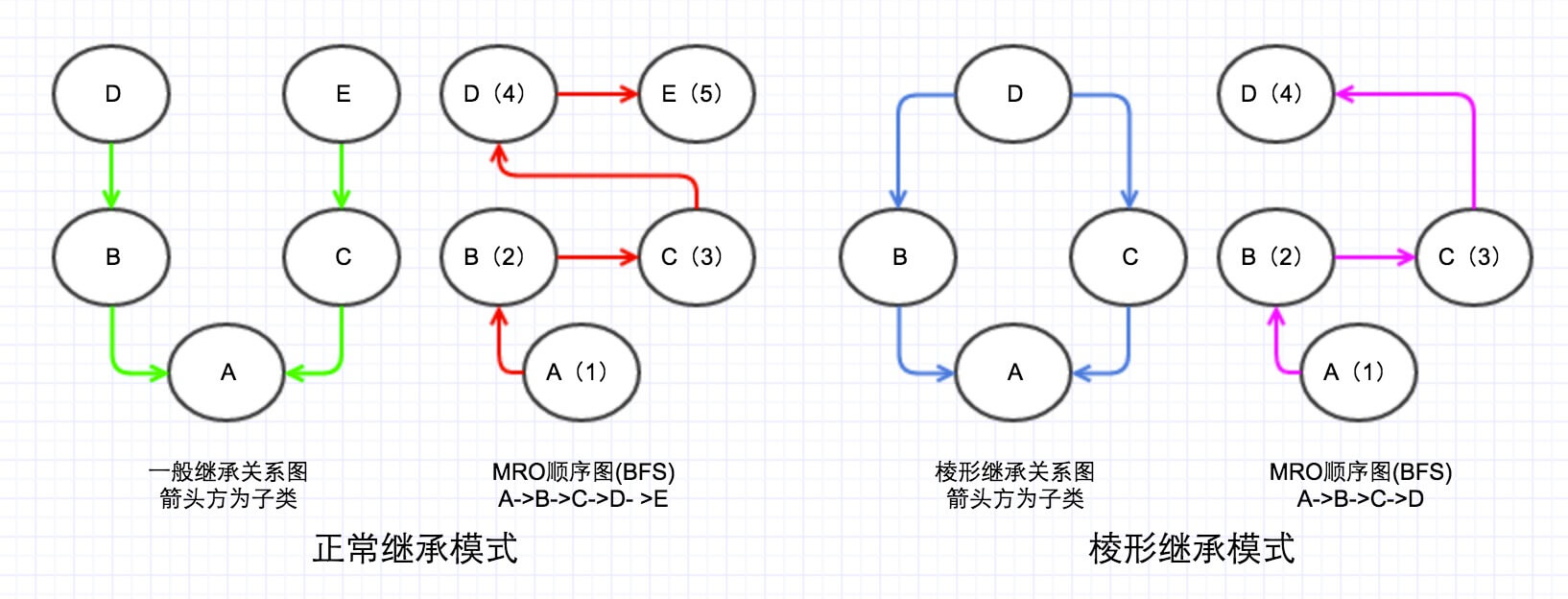
**Python2.2版本：新式类（new-style class）诞生**

为了使类和内置类型更加统一，引入了新式类。新式类的每个类都继承于同一个基类，可以是自定义类或者其它类，默认继承自object，子类可以调用父类的构造函数

这时有两种MRO方法：

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)）



两种继承模式在BFS下的优缺点：

第一种，正常继承模式，看起来正常，不过实际上感觉很别扭，比如B明明继承了D的某个属性（假设为foo），C中也实现了这个属性foo，那么BFS明明先访问B然后再去访问C，但是为什么foo这个属性会是C？这种应该先从B和B的父类开始找的顺序，我们称之为单调性。

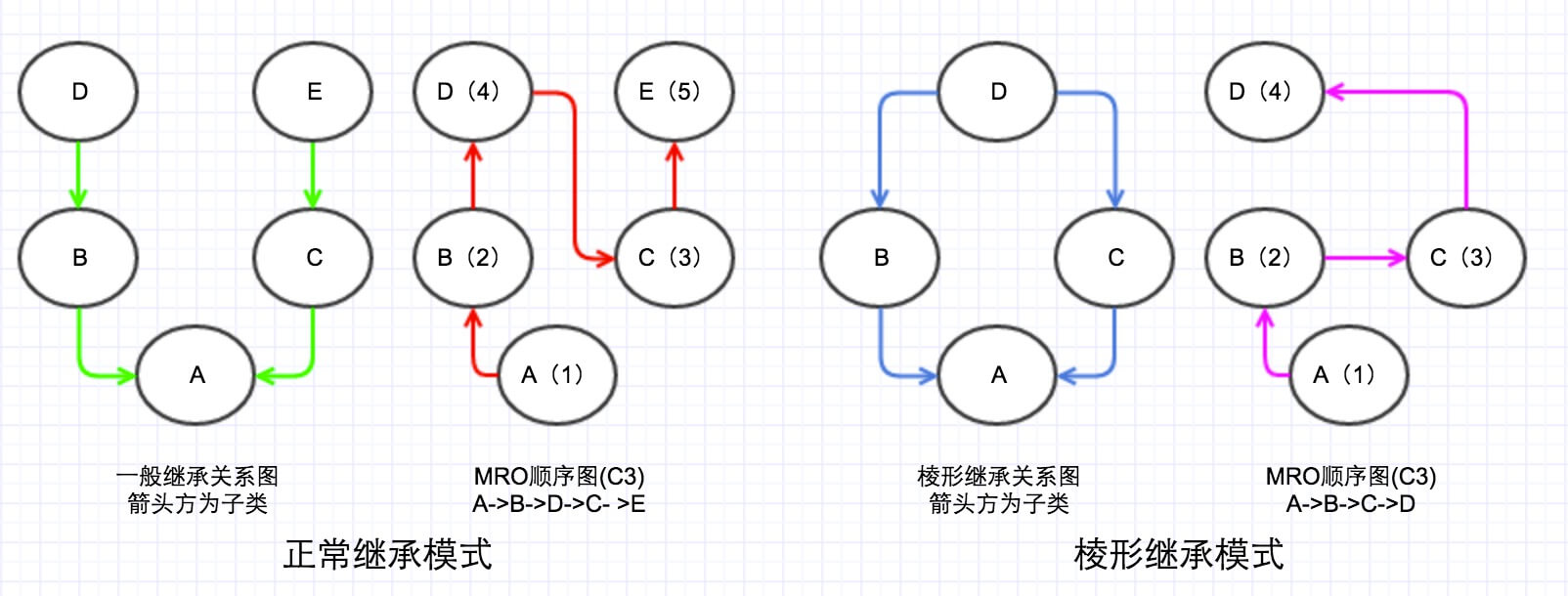
第二种，棱形继承模式，这种模式下面，BFS的查找顺序虽然解了DFS顺序下面的棱形问题，但是它也是违背了查找的单调性。

因为违背了单调性，所以BFS方法只在Python2.2中出现了，在其后版本中用C3算法取代了BFS

**Python2.3到Python2.7：经典类、新式类和平发展**

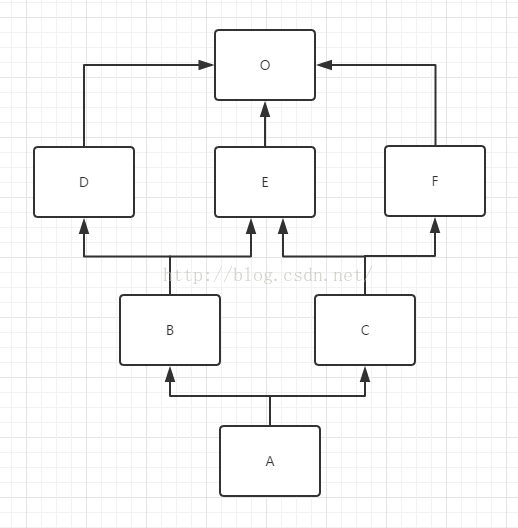
因为之前的BFS存在较大的问题，所以从Python2.3开始新式类的MRO取而代之的是C3算法，我们可以知道C3算法肯定解决了单调性问题和只能继承无法重写的问题

在MRO的C3算法顺序如下图，看起来简直是DFS和BFS的合体有木有，但是仅仅是看起来像而已。



**神奇的C3算法：**

1.多继承UML图：



备注：O==object

**#C3 定义引用开始**

C3 算法：MRO是一个有序列表L，在类被创建时就计算出来。

L（Child（Base1，Base2）） = [ Child + merge（ L（Base1） ,  L（Base2） ,  Base1Base2 ）]

L（object） = [ object ]

L的性质：结果为列表，列表中至少有一个元素即类自己。

+        ： 添加到列表的末尾，即 [ A + B ] = [ A，B ]

merge ： ① 如果列表空则结束，非空 读merge中第一个列表的表头，

                ② 查看该表头是否在 merge中所有列表的表尾中。

                ②-->③ 不在，则 放入 最终的L中，并从merge中的所有列表中删除，然后 回到①中

               ②-->④ 在，查看 当前列表是否是merge中的最后一个列表

               ④-->⑤ 不是 ，跳过当前列表，读merge中下一个列表的表头，然后 回到 ②中

               ④-->⑥ 是，异常。类定义失败。

表头： 列表的第一个元素 （列表：ABC，那么表头就是A,B和C就是表尾）

表尾： 列表中表头以外的元素集合（可以为空）

merge 简单的说即寻找合法表头（也就是不在表尾中的表头），如果所有表中都未找到合法表头则异常。

**#C3定义引用结束**

例如：

L(D) = L(D(O))

     = D + merge(L(O))

     = D + O

     = [D,O]

L(B) = L(B(D,E))

     = B + merge(L(D) , L(E))

     = B + merge(DO , EO) # 第一个列表DO的表头D，其他列表比如EO的表尾都不含有D，所以可以将D提出来，即D是合法表头

     = B + D + merge(O , EO) #从第一个开始表头是O,但是后面的列表EO的表尾中含有O所以O是不合法的，所以跳到下一个列表EO

     = B + D + E + merge(O , O)

     = [B,D,E,O]

同理:

L(C) = [C,E,F,O]

L(A(B,C)) = A + merge(L(B),L(C),BC)

          = A + merge(BDEO,CEFO,BC)#B是合法表头

          = A + B + merge(DEO,CEFO,C)#D是合法表头

          = A + B + D + merge(EO,CEFO,C)#E不是合法表头，跳到下一个列表CEFO，此时C是合法表头

          = A + B + D + C + merge(EO,EFO)#由于第三个列表中的C被删除，为空，所以不存在第三个表，只剩下两个表；此时E是合法表头

          = A + B + D + C + E + merge(O,FO)#O不是合法表头，跳到下一个列表FO，F是合法表头，

          = A + B + D + C + E + F + merge(O,O)#O是合法表头

          = A + B + D + C + E + F + O

          = [A,B,D,C,E,F,O]