20. 核能来袭-MRO和C3算法

本节主要内容:

1. python多继承
2. python经典类的MRO
3. python新式类的MRO, C3算法
4. super是什么鬼?

一. python多继承

在前面的学习过程中. 我们已经知道了Python中类与类之间可以有继承关系. 当出现了x是一种y的的时候. 就可以使用继承关系. 即"is-a" 关系. 在继承关系中. 子类自动拥有父类中除了私有属性外的其他所有内容. python支持多继承. 一个类可以拥有多个父类.

```
class ShenXian: # 神仙

    def fei(self):
        print("神仙都会飞")



class Monkey:   # 猴

    def chitao(self):
        print("猴子喜欢吃桃子")


class SunWukong(ShenXian, Monkey): # 孙悟空是神仙, 同时也是一只猴
    pass


sxz = SunWukong()   # 孙悟空
sxz.chitao()    # 会吃桃子
sxz.fei()       # 会飞
```

此时, 孙悟空是一只猴子, 同时也是一个神仙. 那孙悟空继承了这两个类. 孙悟空自然就可以执行这两个类中的方法.

多继承用起来简单. 也很好理解. 但是多继承中, 存在着这样一个问题. 当两个父类中出现了重名方法的时候. 这时该怎么办呢? 这时就涉及到如何查找父类方法的这么一个问题. 即MRO(method resolution order) 问题. 在python中这是一个很复杂的问题. 因为在不同的python版本中使用的是不同的算法来完成MRO的. 首先. 我们目前能见到的有两个版本:

- python2

在python2中存在两种类.

一个叫经典类. 在python2.2之前. 一直使用的是经典类. 经典类在基类的根如果什么都不写. 表示继承xxx.

一个叫新式类. 在python2.2之后出现了新式类. 新式类的特点是基类的根是object

- python3

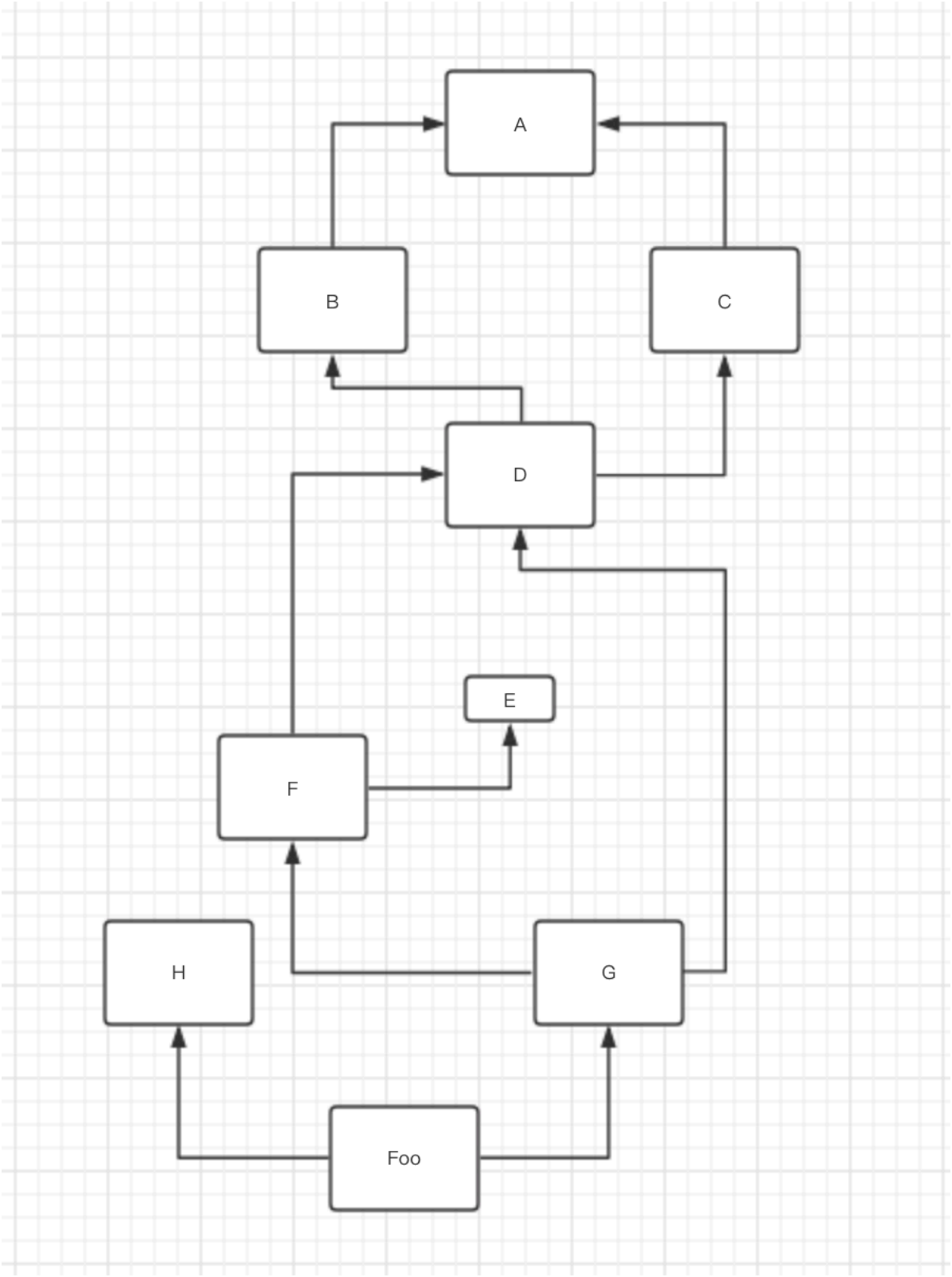
python3中使用的都是新式类. 如果基类谁都不继承. 那这个类会默认继承object

二. 经典类的MRO

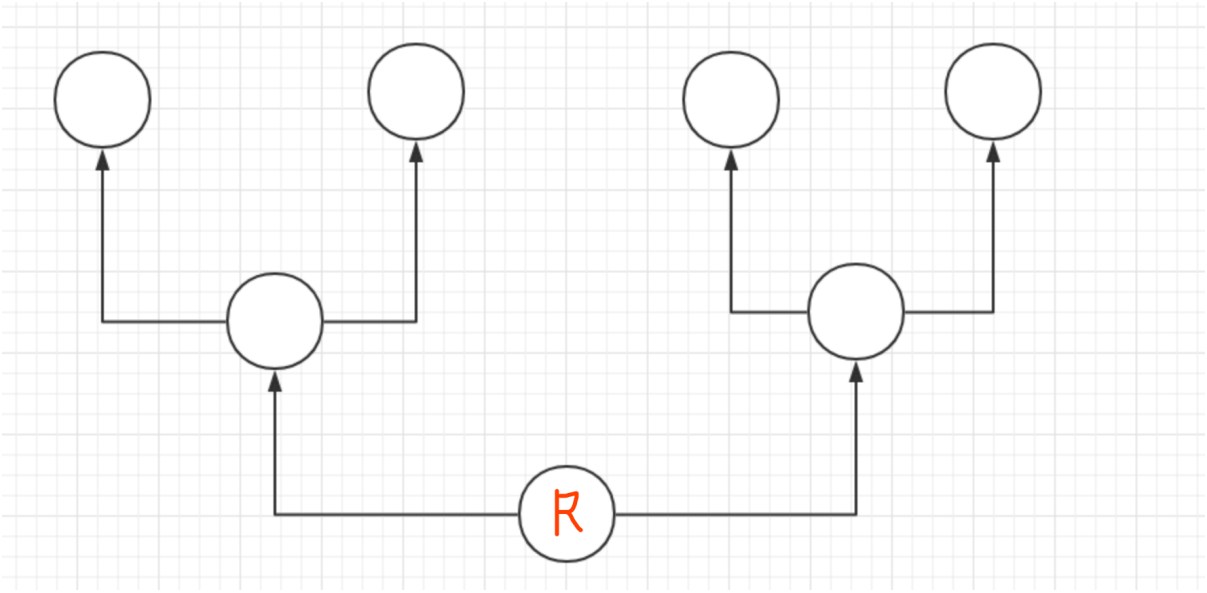
虽然在python3中已经不存在经典类了. 但是经典类的MRO最好还是学一学. 这是一种树形结构遍历的一个最直接的案例. 在python的继承体系中. 我们可以把类与类继承关系化成一个树形结构的图. 来, 上代码:

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

class E:
    pass

class F(D, E):
    pass

class G(F, D):
    pass

class H:
    pass

class Foo(H, G):
    pass
```

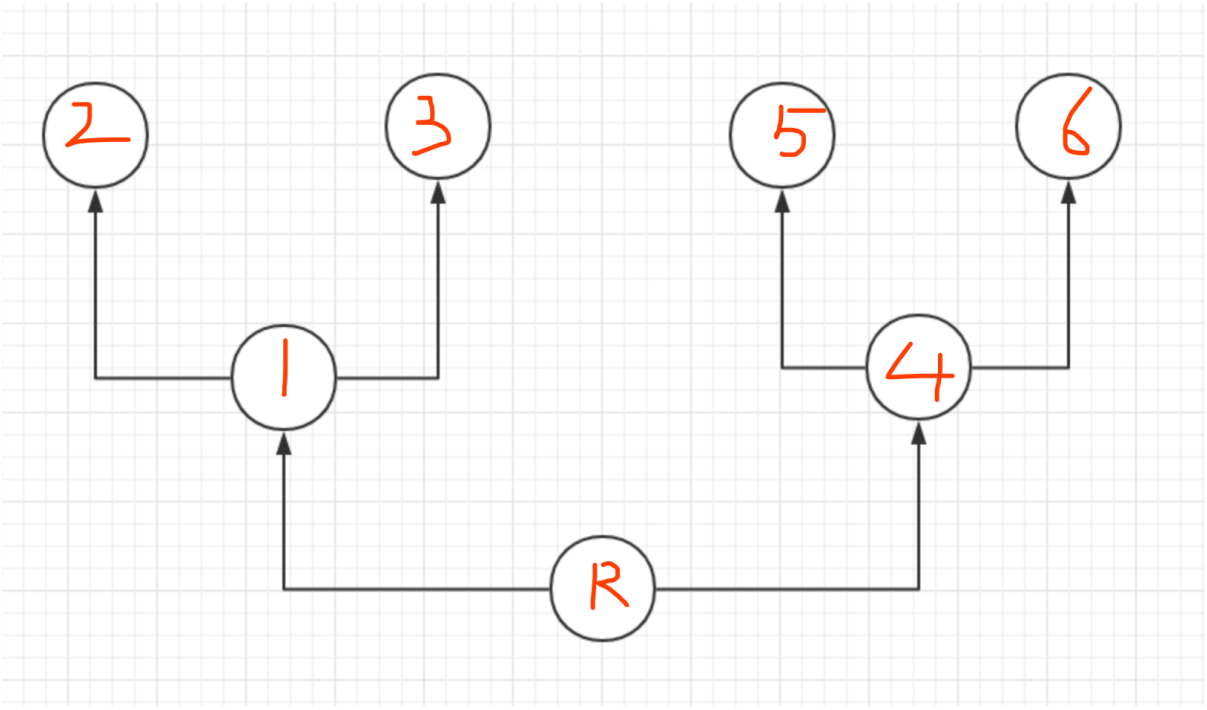
对付这样的MRO. 很简单. 画图即可:



继承关系图已经有了. 那如何进行查找呢? 记住一个原则. 在经典类中采用的是深度优先遍历方案. 什么是深度优先. 就是一条路走到头. 然后再回来. 继续找下一个. 比如. 有一个快递员. 去给每家每户送鸡蛋.



图中每个圈都是准备要送鸡蛋的住址. 箭头和黑线表示线路. 那送鸡蛋的顺序告诉你入口在最下面R. 并且必须从左往右送. 那怎么送呢?



如图. 肯定是按照123456这样的顺序来送. 那这样的顺序就叫深度优先遍历. 而如果是142356呢? 这种被称为广度优先遍历. 好了. 深度优先就说这么多. 那么上面那个图怎么找的呢? MRO是什么呢? 很简单. 记住. 从头开始. 从左往右. 一条路跑到头, 然后回头. 继续一条路跑到头. 就是经典类的MRO算法.

类的MRO: Foo-> H -> G -> F -> D -> B -> A -> C -> E. 你猜对了么?

三. 新式类的MRO

python中的新式类的MRO是采用的C3算法来完成的.

c3算法很简单. 就看你的代码就够了. 不需要去画图. 而且画图也看不出来什么. 不过如果写得多了是可以从图上总结出一些规律来的. 先看代码:

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

class E(C, A):
    pass

class F(D, E):
    pass

class G(E):
    pass

class H(G, F):
    pass
```

首先. 我们要确定从H开始找. 也就是说. 创建的是H的对象.

如果从H找. 那找到H+H的父类的C3, 我们设C3算法是L(x) , 即给出x类. 找到x的MRO

L(H) = H + L(G) + L(F) + GF

继续从代码中找G和F的父类往里面带

L(G) = G + L(E) + E

L(F) = F + L(D)+ L(E) + DE

继续找E 和 D

L(E) = E + L(C) + L(A) + CA

L(D) = D + L(B) + L(C) + BC

继续找B和C

L(B) = B + L(A) + A

L(C) = C + L(A) + A

最后就剩下一个A了. 也就不用再找了. 接下来. 把L(A) 往里带. 再推回去. 但要记住. 这里的 + 表示的是merge. merge的原则是用每个元组的头一项和后面元组的除头一项外的其他元素进行比较, 看是否存在. 如果存在. 就从下一个元组的头一项继续找. 如果找不到. 就拿出来. 作为merge的结果的一项. 以此类推. 直到元组之间的元素都相同. 也就不用再找了.

L(B) =(B,) + (A,) + (A) -> (B, A)

L(C) =(C,) + (A,) + (A) -> (C, A)
继续带.
L(E) = (E,) + (C, A) + (A) + (C,A) -> E, C, A
L(D) = (D,) + (B, A) + (C, A) + (B, C) -> D, B, C, A
继续带.
L(G) = (G,) + (E, C, A) + (E) -> G, E, C, A
L(F) = (F,) + (D, B, C, A) + (E, C, A) + (D, E)-> F, D, B, E, C, A
加油, 最后了
L(H) = (H,) + (G, E, C, A) + (F, D, B, E, C, A) + (G, F) -> H, G, F, D, B, E, C, A

算完了. 最终结果 HGFDBECA. 那这个算完了. 如何验证呢? 其实python早就给你准备好了. 我们可以使用类名.__mro__获取到类的MRO信息.

```
print(H.__mro__)
结果:
(<class '__main__.H'>, <class '__main__.G'>, <class '__main__.F'>, <class
'__main__.D'>, <class '__main__.B'>, <class '__main__.E'>, <class
'__main__.C'>,<class '__main__.A'>, <class 'object'>)
```

结果OK. 那既然python提供了. 为什么我们还要如此麻烦的计算MRO呢? 因为笔试.....你在笔试的时候, 是没有电脑的. 所以这个算法要知道. 并且简单的计算要会. 真是项目开发的时候很少有人这么去写代码.

这个说完了. 那C3到底怎么看更容易呢? 其实很简单. C3是把我们多个类产生的共同继承留到最后去找. 所以. 我们也可以从图上来看到相关的规律. 这个要大家自己多写多画图就能感觉到了. 但是如果没有所谓的共同继承关系. 那几乎就当成是深度遍历就可以了.

四. super是什么鬼?

super()可以帮我们执行MRO中下一个父类的方法. 通常super()有两个使用的地方:

1. 可以访问父类的构造方法
2. 当子类方法想调用父类(MRO)中的方法

我们先看第一种:

```
class Foo:

    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

class Bar(Foo):
    def __init__(self, a, b, c, d):
        super().__init__(a, b, c)   # 访问父类的构造方法
        self.d = d
```

```
b = Bar(1, 2, 3, 4)
print(b.__dict__)

结果:
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

这样就方便了子类. 不需要写那么多了. 直接用父类的构造帮我们完成一部分代码

第二种:

```
class Foo:
    def func1(self):
        super().func1() # 此时找的是MRO顺序中下一个类的func1()方法
        print("我的老家. 就住在这个屯")

class Bar:
    def func1(self):
        print("你的老家. 不在这个屯")

class Ku(Foo, Bar):
    def func1(self):
        super().func1() # 此时super找的是Foo
        print("他的老家. 不知道在哪个屯")

k = Ku() # 先看MRO . KU, FOO, BAR object
k.func1()


k2 = Foo() # 此时的MRO. Foo object
k2.func1() # 报错
```

最后. 给大家扔一个面试题

```


# MRO + super 面试题
class Init(object):
    def __init__(self, v):
        print("init")
        self.val = v


class Add2(Init):
    def __init__(self, val):
        print("Add2")
        super(Add2, self).__init__(val)
        print(self.val)
        self.val += 2

class Mult(Init):
    def __init__(self, val):
```

```
        print("Mult")
        super(Mult, self).__init__(val)
        self.val *= 5

class HaHa(Init):
    def __init__(self, val):
        print("哈哈")
        super(HaHa, self).__init__(val)
        self.val /= 5

class Pro(Add2,Mult,HaHa): #
    pass

class Incr(Pro):

    def __init__(self, val):
        super(Incr, self).__init__(val)
        self.val+= 1

# Incr Pro Add2 Mult HaHa Init

p = Incr(5)
print(p.val)

c = Add2(2)
print(c.val)
```

提示. 先算MRO . 然后看清楚self是谁.

结论: **不管super()写在哪儿. 在哪儿执行. 一定先找到MRO列表. 根据MRO列表的顺序往下找. 否则一切都是错的**