《源码探秘 CPython》71. 类对象 MRO 的设置，与基类的继承

原创 古明地觉 古明地觉的编程教室 2022-04-19 08:30

收录于合集
#CPython
97个 >

当完成属性字典的设置，就开始确定 MRO 了，即 method resolve order。MRO 表示类继承之后，属性或方法的查找顺序。如果Python是单继承的话，那么这不是问题，直接一层一层向上找即可。但Python是支持多继承的，那么在多继承时，继承的顺序就成为了一个必须考虑的问题。

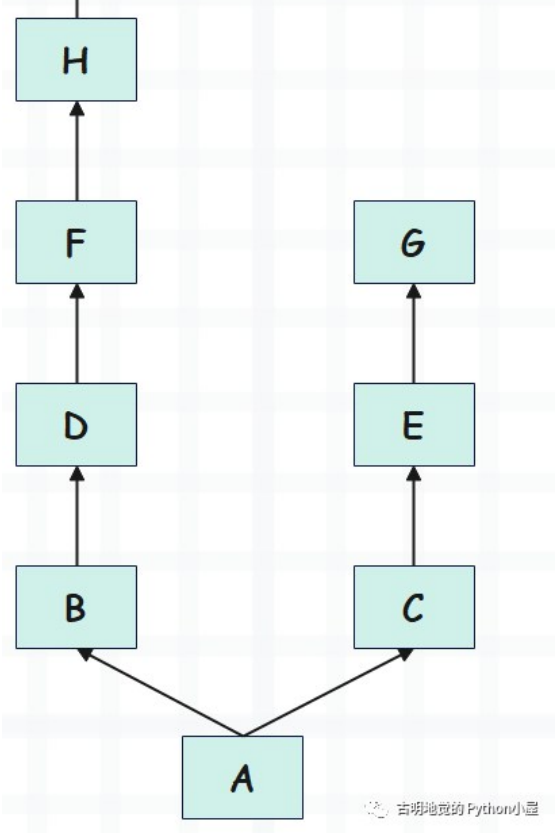
```
class A:
    def foo(self):
        print("A")

class B(A):
    def foo(self):
        print("B")

class C(A):
    def foo(self):
        print("C")
        self.bar()

    def bar(self):
        print("bar C")

class D(C, B):
    def bar(self):
        print("bar D")

d = D()
d.foo()
"""
C
bar D
"""
```

首先打印的是字符串 "C"，表示调用的是 C 的 foo，说明把 C 写在前面，会先从 C 里面查找。但是下面打印了 "bar D"，这是因为 C 里面的 self 实际上是 D 的实例对象。

因为 D 在找不到 foo 函数的时候，会到父类里面找，但是同时也会将 self 传递过去。调用 self.bar 的时候，这个 self 是 D 的实例对象，所以还是会先到 D 里面找，如果找不到再去父类里面找。

而对于虚拟机而言，则是会在PyType_Ready中通过mro_internal函数确定mro。虚拟机将创建一个PyTupleObject对象，里面存放一组类对象，这些类对象的顺序就是虚拟机确定的mro的顺序。而这个元组，最终会被交给 tp_mro 成员保存。

由于确定 MRO 的 mro_internal 函数非常复杂，这里我们就不看源码了，只要能从概念上理解它即可。另外 Python 早期有经典类和新式类两种，现在则只存在新式类，而经典类和新式类采用的搜索策略是不同的，举个例子：



图中的箭头表示继承关系，比如：A 同时继承 B 和 C、B 继承 D、C 继承 E。

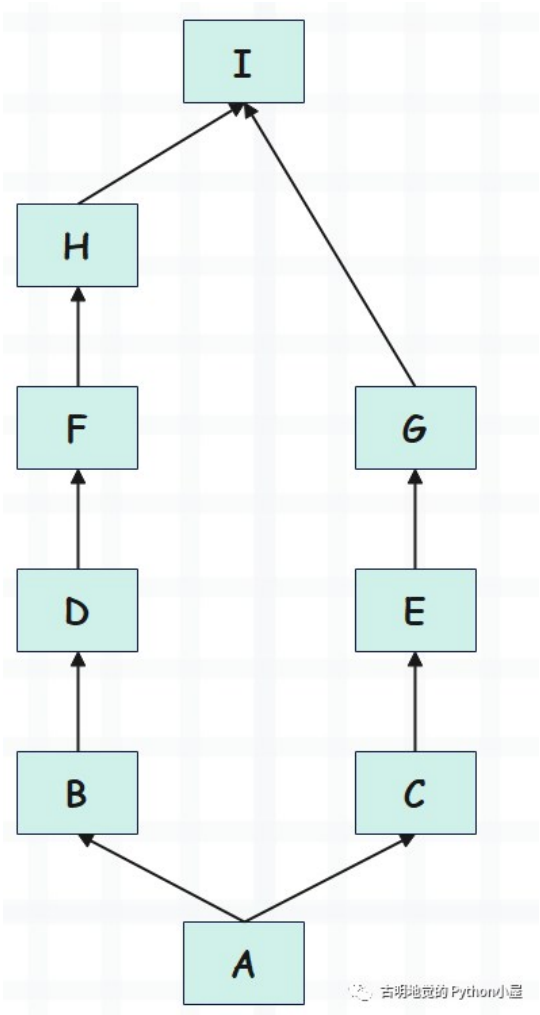
对于上图来说，经典类和新式类的查找方式是一样的，至于两边是否一样多则不重要。查找方式是先从 A 找到 I，再从 C 查找到 G。我们实际演示一下，由于经典类只在 Python2 中存在，所以下面我们演示只新式类。

```
# 这里是python3.8 新式类
I = type("I", (), {})
H = type("H", (I,), {})
F = type("F", (H,), {})
G = type("G", (), {})
D = type("D", (F,), {})
E = type("E", (G,), {})
B = type("B", (D,), {})
C = type("C", (E,), {})
A = type("A", (B, C), {})

for _ in A.__mro__:
    print(_)
"""
<class '__main__.A'>
<class '__main__.B'>
<class '__main__.D'>
<class '__main__.F'>
<class '__main__.H'>
<class '__main__.I'>
<class '__main__.C'>
<class '__main__.E'>
<class '__main__.G'>
<class 'object'>
"""
```

对于 A 继承两个类，然后这个两个类分别继续继承，如果最终没有继承公共的类(忽略 object)，那么经典类和新式类是一样的。像这种泾渭分明、各自继承各自的，都是先一条路找到黑，然后再去另外一条路找。

但如果是下面这种，最终分久必合、两者最终又继承了同一个类，那么经典类还是跟以前一样，按照每一条路都走到黑的方式。但是对于新式类，则是先从 A 找到 H，而 I 这个两边最终继承的类不

找了，然后从 C 找到 I，也就是在另一条路找到头。



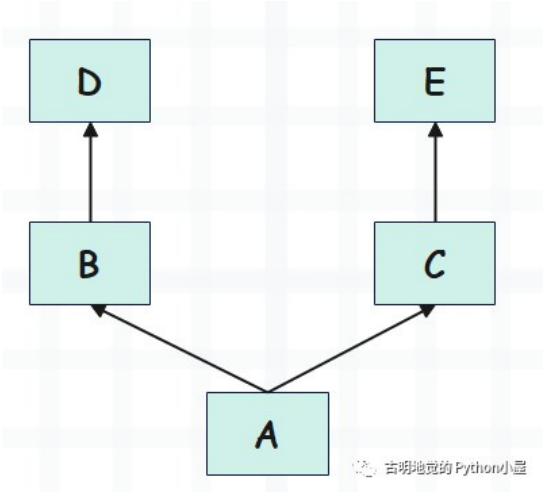

```
# 新式类
I = type("I", (), {})
H = type("H", (I,), {})
F = type("F", (H,), {})
G = type("G", (I,), {})    # 这里让G继承I
D = type("D", (F,), {})
E = type("E", (G,), {})
B = type("B", (D,), {})
C = type("C", (E,), {})
A = type("A", (B, C), {})

for _ in A.__mro__:
    print(_)
"""
<class '__main__.A'>
<class '__main__.B'>
<class '__main__.D'>
<class '__main__.F'>
<class '__main__.H'>
<class '__main__.C'>
<class '__main__.E'>
<class '__main__.G'>
<class '__main__.I'>
<class 'object'>
"""
```

但是Python的多继承比我们想象的要复杂，原因就在于可以任意继承，如果 B 和 C 再分别继承两个类呢？那么我们这里的线路就又要多出两条了。不过既然要追求刺激，就贯彻到底喽，我们就来看一下，如何从混乱不堪的继承关系中，找到正确的继承顺序。

> 由于 Python3 只有新式类，因为下面我们会以介绍新式类为主，经典类了解一下即可。

很多文章可能告诉你经典类采用深度优先算法，新式类采用广度优先算法，真的是这样吗？我们举

一个例子：



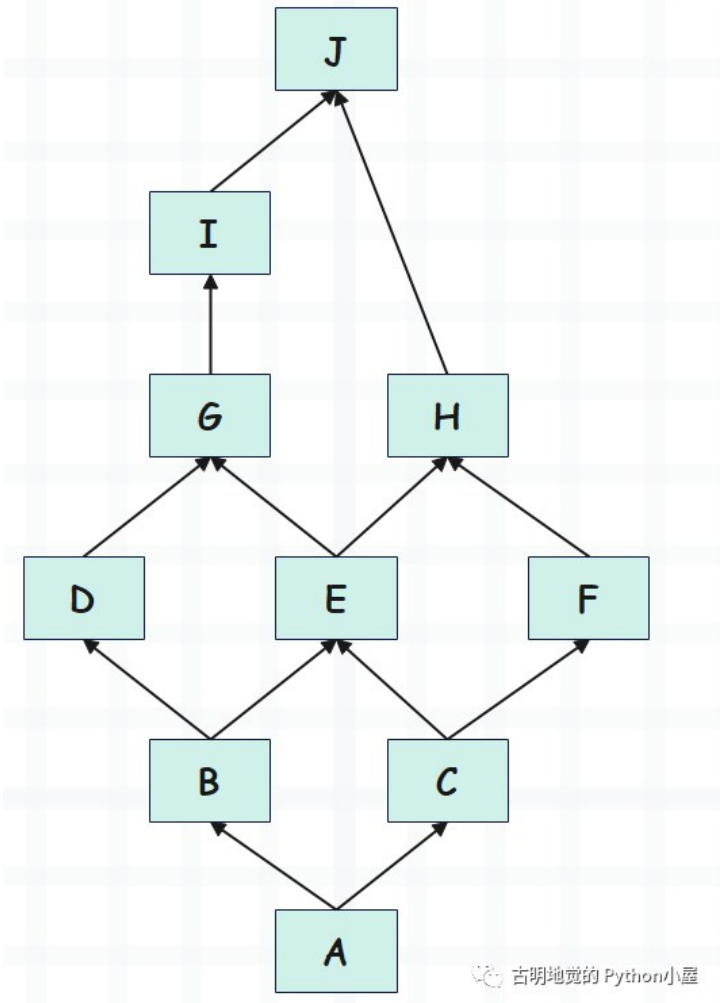
假设我们调用 A() 的 foo 方法，但是 A 里面没有，那么理所应当会去 B 里面找。但是 B 里面也没有，而 C 和 D 里面有，那么这个时候是去 C 里面找还是去 D 里面找呢？

根据我们之前的结论，显然是去 D 里面找，可如果按照广度优先的逻辑来说，那么应该是去 C 里面找啊。所以广度优先理论在这里就不适用了，因为 B 继承了 D，而 B 和 C 并没有直接关系，我们应该把 B 和 D 看成一个整体。

而Python的 MRO 实际上是采用了一种叫做 C3 的算法，这个 C3 算法比较复杂（其实也不算复杂），但是我个人总结出一个更加好记的结论，如下：

> 当沿着一条继承链寻找类时，默认会沿着该继承链一直找下去。但如果发现某个类出现在了另一条继承链当中，那么当前的继承链的搜索就会结束，然后在"最开始"出现分歧的地方转向下一条继承链的搜索。

这是我个人总结的，或许光看字面意思的话会比较难理解，但是通过例子就能明白了。



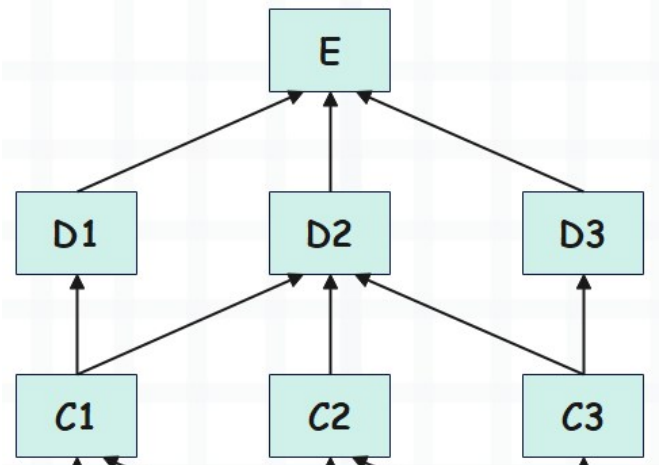
箭头表示继承关系，继承顺序是从左到右，比如这里的 A 就相当于 **class A(B, C)**，下面我们来从头到尾分析一下 A 的 MRO。

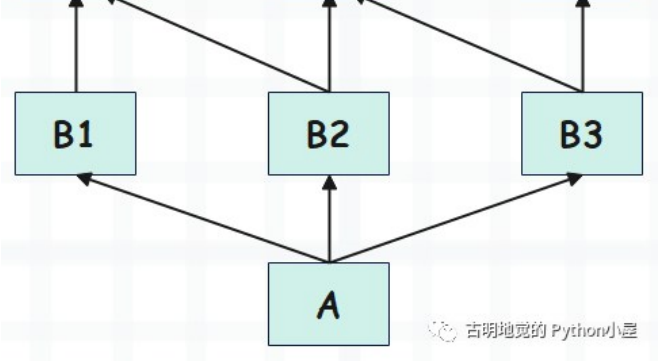
- 1）因为是 A 的 MRO，所以查找时，第一个类就是 A；
- 2）然后 A 继承 B 和 C，由于是两条路，因此我们说 A 这里就是一个分歧点。但由于 B 在前，所以接下来是 B，而现在 MRO 的顺序就是 A B；
- 3）但是 B 这里也出现了分歧点，不过不用管，因为我们说会沿着继承链不断往下搜索，现在 MRO 的顺序是A B D；
- 4）然后从 D 开始继续寻找，这里注意了，按理说会找到 G 的。但是 G 不止被一个类继承，也就是说沿着当前的继承链查找 G 时，发现 G 还出现在了其它的继承链当中。怎么办？显然要回到最初的分歧点，转向下一条继承链的搜索；
- 5）最初的分歧点是 A，那么该去找 C 了，现在 MRO 的顺序就是 A B D C；
- 6）注意 C 这里也出现了分歧点，而 A 的两条分支已经结束了，所以现在 C 就是最初的分歧点了。而 C 继承自 E 和 F，显然要搜索 E，那么此时 MRO 的顺序就是 A B D C E；
- 7）然后从 E 开始搜索，显然要搜索 G，此时 MRO 顺序变成 A B D C E G；
- 8）从 G 要搜索 I，此时 MRO 的顺序是 A B D C E G I；
- 9）从 I 开始搜索谁呢？由于 J 出现在了其它的继承链中，那么要回到最初的分歧点，也就是 C。那么下面显然要找 F，此时 MRO 的顺序是 A B D C E G I F；
- 10）F 只继承了 H, 那么肯定要找 H, 此时 MRO 的顺序是 A B D C E G I F H；
- 11）H 显然只能找 J 了，因此最终 A 的 MRO 的顺序就是 A B D C E G I F H J object；

我们实际测试一下：

```
J = type("J", (object, ), {})
I = type("I", (J, ), {})
H = type("H", (J, ), {})
G = type("G", (I, ), {})
F = type("F", (H, ), {})
E = type("E", (G, H), {})
D = type("D", (G, ), {})
C = type("C", (E, F), {})
B = type("B", (D, E), {})
A = type("A", (B, C), {})

# A B D C E G I F H J
for _ in A.__mro__:
    print(_)
"""
<class '__main__.A'>
<class '__main__.B'>
<class '__main__.D'>
<class '__main__.C'>
<class '__main__.E'>
<class '__main__.G'>
<class '__main__.I'>
<class '__main__.F'>
<class '__main__.H'>
<class '__main__.J'>
<class 'object'>
"""
```

为了加深理解，我们再举个更复杂的例子：





- 1）首先是 A，A 继承 B1、B2、B3，会先走 B1，此时 MRO 是 A B1。并且现在 A 是分歧点；
- 2）从 B1 处本来应该去找 C1，但是 C1 还被其他类继承，也就是出现在了其它的继承链当中。因此要回到最初的分歧点 A，从下一条继承链开始找，显然要找 B2，此时 MRO 就是 A B1 B2；
- 3）从 B2 开始，显然要找 C1，此时 MRO 的顺序就是 A B1 B2 C1；
- 4）从C1开始，显然要找 D1，因为 D1 只被 C1 继承，也就是说，它没有出现在另一条继承链当中，因此此时 MRO 的顺序是 A B1 B2 C1 D1；
- 5）对于 D1 而言，显然接下来是不会去找 E 的，因为 E 还出现在另外的继承链当中。咋办？回到最初的分歧点，注意这里的分歧点显然还是 A，因为 A 的分支还没有走完。显然此时要走 B3，那么 MRO 的顺序就是 A B1 B2 C1 D1 B3；
- 6）从 B3 开始找，显然要找C2，注意：A 的分支已经走完，此时 B3 就成了新的最初分歧点。现在 MRO 的顺序是 A B1 B2 C1 D1 B3 C2；
- 7）C2 会找 D2吗？显然不会，因为 D2 还被 C3 继承，所以它出现在了其他的继承链中。于是要回到最初的分歧点，这里是 B3，显然下面要找 C3。另外由于 B3 的分支也已经走完，所以现在 C3 就成了新的最初分歧点。此时 MRO 的顺序是 A B1 B2 C1 D1 B3 C2 C3；
- 8）从 C3 开始，显然要找 D2，此时 MRO 的顺序是 A B1 B2 C1 D1 B3 C2 C3 D2；
- 9）但是 D2 不会找 E，因此回到最初分歧点 C3，下面要找 D3。而 D3 找完之后显然只能再找 E 了，因此最终 MRO 的顺序是 A B1 B2 C1 D1 B3 C2 C3 D2 D3 E object；

下面测试一下：

```
E = type("E", (), {})
D1 = type("D1", (E,), {})
D2 = type("D2", (E,), {})
D3 = type("D3", (E,), {})
C1 = type("C1", (D1, D2), {})
C2 = type("C2", (D2,), {})
C3 = type("C3", (D2, D3), {})
B1 = type("B1", (C1,), {})
B2 = type("B2", (C1, C2), {})
B3 = type("B3", (C2, C3), {})
A = type("A", (B1, B2, B3), {})

for _ in A.__mro__:
    print(_)
"""
<class '__main__.A'>
<class '__main__.B1'>
<class '__main__.B2'>
<class '__main__.C1'>
<class '__main__.D1'>
<class '__main__.B3'>
<class '__main__.C2'>
<class '__main__.C3'>
<class '__main__.D2'>
<class '__main__.D3'>
<class '__main__.E'>
<class 'object'>
"""
```

以上就是计算 MRO 所采用的策略，关于源码部分我们就不看了，复杂是一方面，重点是没什么太大必要。个人觉得，关于多继承从目前这个层面上来理解已经足够了。

另外通过以上可以看出，Python 支持非常复杂的继承关系。但是实际使用多继承的时候，我们最好还是要斟酌一下，因为多继承如果设计的不好，容易使得类的继承关系变得非常混乱。

> 为此，Python 提供了一种模式叫做 Mixin，可以网上搜索一下。比较简单，这里就不展开了。

在执行父类函数时传入的 self 参数，是很多初学者容易犯的错误。

```
class A:

    def foo(self):
        print("A: foo")
        self.bar()

    def bar(self):
        print("A: bar")

class B:

    def bar(self):
        print("B: bar")

class C(A, B):

    def bar(self):
        print("C: bar")

C().foo()
"""
A: foo
C: bar
"""
```

C 的实例对象在调用 foo 的时候，首先会去 C 里面查找，但是C没有，所以按照 MRO 的顺序会去 A 里面找。而 A 里面存在，所以调用，但是调用时传递的 self 是 C 的实例对象，因为是 C 的实例对象调用的。

所以里面的 self.bar，这个 self 还是 C 的实例对象，那么调用 bar 的时候，会去哪里找呢？显然还是从 C 里面找，所以 self.bar() 的时候打印的是字符串 **C: bar**，而不是 **A: bar**。

同理再来看一个关于 super 的栗子：

```
class A:

    def foo(self):
        super(A, self).foo()

class B:

    def foo(self):
        print("B: foo")

class C(A, B):
    pass

try:
    A().foo()
except Exception as e:
    print(e)  # 'super' object has no attribute 'foo'

#首先A的父类是object
#所以super(A, self).foo()的时候会去执行object的foo
#但是object没有foo，所以报错
```

```
22
23 #但如果是 C 的实例调用呢？
24 C().foo()  # B: foo
```

如果是 C() 调用 foo 的话，最终却执行了 B 的 foo，这是什么原因呢？首先 C 里面里面没有 foo，那么会去执行 A 的 foo，但是执行时候的 self 是 C 的实例对象，super 里面的 self 也是 C 里面的 self。

然后我们知道对于 C 而言，其 MRO 是 C、A、B、object。所以此时的 super(A, self).foo() 就表示：沿着继承链 C 的 MRO 去找 foo 函数，但是 super 里面有一个 A，表示不要从头开始找，而是从 A 的后面开始找，所以下一个就找到 B 了。

所以说 super 不一定就是父类，而是要看里面的 self 是谁。总之：super(xxx, self) 一定是 type(self) 对应的 MRO 中，xxx 的下一个类。再举个栗子：

```
1  class A:
2
3      def foo(self):
4          # 从 B 里面找 foo
5          super(A, self).foo()
6          # 从 C 里面找 foo
7          super(B, self).foo()
8
9  class B:
10
11     def foo(self):
12         print("B: foo")
13
14 class C:
15
16     def foo(self):
17         print("C: foo")
18
19
20 class D(A, B, C):
21     pass
22
23 D().foo()
24 """
25 B: foo
26 C: foo
27 """
```

当 D() 调用 foo 的时候，D 里面没有，那么按照继承关系一定是去 A 里面找。而 self 是 D 的 self，那么 **super(A, self)** 就是 D 对应的 MRO 中，A 的下一个类，也就是 B；**super(B, self)** 就是 D 对应的 MRO 中，B 的下一个类，也就是 C。

虽然可以继承多个类，但是这些类之间有可能发生冲突，举个栗子：

```
1 class A(int, str):
2     pass
3 # TypeError: multiple bases have instance lay-out conflict
```

我们发现执行的时候报错了，这个类虚拟机压根就不会让你创建，原因很简单，它们的实例对象在内存布局上面发生冲突了。

比如 A("123") 这个实例，虚拟机是把它看成整数呢？还是字符串呢？因为 A 同时继承 int 和 str，就意味着其实例既可以像整数一样使用除法，也可以像字符串一样调用 join 方法。但是整数没有 join，字符串也无法使用除法，因此这个类是不合理的。继承的多个基类，在实例对象上面发生了内存布局上的冲突。

虽然上面给出了解释，但如果仔细推敲，会发现这个解释有些站不住脚，因为我们自定义的类没有这个问题。如果是自定义的类，别说继承两个，就算继承十个毫无关系的类，也不会出现冲突，这又是什么原因呢？其实两者的差别，就在于内置类对象的实例对象是没有属性字典的，但是自定义类对象的实例对象有。

```
class Base1:
    __slots__ = ("a",)

class Base2:
    __slots__ = ("a",)

class A(Base1, Base2):
    pass
# TypeError: multiple bases have instance lay-out conflict
```

此时 Base1 和 Base2 的实例，和内置类对象的实例是类似的，都没有属性字典，或者说实例对象可以绑定哪些属性已经定好了。同时继承 Base1 和 Base2 也会发生冲突。

结论：对于实例对象没有属性字典的类对象，最多同时继承一个。但是也有特例，如果类对象不接收任何参数，即使它的实例对象没有属性字典，也没有影响。

```
class Base1:
    __slots__ = ("a",)

class Base2:
    __slots__ = ()

# 合法
class A(Base2, int):
    pass

# 不合法
class B(Base1, int):
    pass
```

Base1 的实例对象显然可以绑定一个属性叫 a，意味着 Base1 可以接收一个参数；Base2 的实例对象无法绑定任何属性，意味着 Base2 不接收任何参数。

因此 **class A(Base1, int)** 是不合法的，因为 Base1 和 int 的实例没有属性字典，但它们都接收参数。同理 **class A (Base1, str)**、**class A(Base1, dict)** 也一样。

但是 **class A(Base2, str)** 没有影响，因为 Base2 的实例无法绑定任何属性，因此也就不存在内存布局上的冲突。

看个栗子：

```
class Base:
    pass

class A(Base):
    pass

class B(Base, A):
    pass
"""
TypeError: Cannot create a consistent method resolution
order (MRO) for bases Base, A
"""
```

这个错误的原因应该不难理解，B 继承了 Base 和 A，但是 Base 和 A 之间又是父子关系。比如 B 在找不到某个方法时，会到 Base 里面找，Base 找不到再去 A 里面找。但如果 A 在找不到，就又回到了 Base，因为 A 继承 Base，那么此时就出现了闭环，这是不允许的。

但如果把 B 的创建改成 **class B(A, Base)** 则没问题，所以当继承的多个类之间也存在继承关系时，子类要在父类的前面。当然啦，最正确的做法应该是只继承子类，比如这里的 class B(A)。因为 A 已经继承了 Base，所以 B 只需继承 A 即可，无需再继承 Base。像 **class B(A, Base)** 这种做法虽然不会报错，但明显有些画蛇添足。

子类在找不到某个属性或方法时，会去基类里面找，这背后的原理是什么呢？答案简单到超乎你想象，就是单纯的拷贝。虚拟机在确定 MRO 之后，就会遍历 MRO 顺序列表，里面存储了该类的所有直接基类、间接基类（也就是基类的基类）。而虚拟机会将自身没有、但是基类中存在的操作拷贝到该类当中，从而完成对基类操作的继承动作。

而这个继承的动作是发生在 inherit_slots 中。

```c
//typeobject.c
int
PyType_Ready(PyTypeObject *type)
{
    //...
    //获取 MRO 顺序列表
    bases = type->tp_mro;
    assert(bases != NULL);
    assert(PyTuple_Check(bases));
    n = PyTuple_GET_SIZE(bases);
    //遍历所有的基类，包括直接基类、间接基类
    //由于 MRO 的第一个类是其本身，所以遍历是从第二项开始的
    for (i = 1; i < n; i++) {
        PyObject *b = PyTuple_GET_ITEM(bases, i);
        if (PyType_Check(b))
            //继承基类操作
            inherit_slots(type, (PyTypeObject *)b);
    }
    //...
}
```

在inherit_slots中会拷贝相当多的操作，这里就以整型为例：

```c
static void
inherit_slots(PyTypeObject *type, PyTypeObject *base)
{
    PyTypeObject *basebase;

#undef SLOTDEFINED
#undef COPYSLOT
#undef COPYNUM
#undef COPYSEQ
#undef COPYMAP
#undef COPYBUF

#define SLOTDEFINED(SLOT) \
    (base->SLOT != 0 && \
     (basebase == NULL || base->SLOT != basebase->SLOT))

```

```
#define COPYSLOT(SLOT) \
    if (!type->SLOT && SLOTDEFINED(SLOT)) type->SLOT = base->SLOT

#define COPYASYNC(SLOT) COPYSLOT(tp_as_async->SLOT)
#define COPYNUM(SLOT) COPYSLOT(tp_as_number->SLOT)
#define COPYSEQ(SLOT) COPYSLOT(tp_as_sequence->SLOT)
#define COPYMAP(SLOT) COPYSLOT(tp_as_mapping->SLOT)
#define COPYBUF(SLOT) COPYSLOT(tp_as_buffer->SLOT)

    /* This won't inherit indirect slots (from tp_as_number etc.)
       if type doesn't provide the space. */

    if (type->tp_as_number != NULL && base->tp_as_number != NULL) {
        basebase = base->tp_base;
        if (basebase->tp_as_number == NULL)
            basebase = NULL;
        COPYNUM(nb_add);
        COPYNUM(nb_subtract);
        COPYNUM(nb_multiply);
        COPYNUM(nb_remainder);
        COPYNUM(nb_divmod);
        COPYNUM(nb_power);
        COPYNUM(nb_negative);
        COPYNUM(nb_positive);
        COPYNUM(nb_absolute);
        COPYNUM(nb_bool);
        COPYNUM(nb_invert);
        COPYNUM(nb_lshift);
        COPYNUM(nb_rshift);
        COPYNUM(nb_and);
        COPYNUM(nb_xor);
        COPYNUM(nb_or);
        COPYNUM(nb_int);
        COPYNUM(nb_float);
        COPYNUM(nb_inplace_add);
        COPYNUM(nb_inplace_subtract);
        COPYNUM(nb_inplace_multiply);
        COPYNUM(nb_inplace_remainder);
        COPYNUM(nb_inplace_power);
        COPYNUM(nb_inplace_lshift);
        COPYNUM(nb_inplace_rshift);
        COPYNUM(nb_inplace_and);
        COPYNUM(nb_inplace_xor);
        COPYNUM(nb_inplace_or);
        COPYNUM(nb_true_divide);
        COPYNUM(nb_floor_divide);
        COPYNUM(nb_inplace_true_divide);
        COPYNUM(nb_inplace_floor_divide);
        COPYNUM(nb_index);
        COPYNUM(nb_matrix_multiply);
        COPYNUM(nb_inplace_matrix_multiply);
    }
    //......
    //......
```

我们在里面看到了很多熟悉的东西，这些都是在继承时需要拷贝的操作。比如布尔类型，PyBool_Type 中并没有设置 nb_add，但是 PyLong_Type 中设置了 nb_add 操作，而 bool 继承 int。所以布尔值是可以直接进行运算的，当然和整数、浮点数运算也是可以的。

因此在numpy中，判断一个数组里面有多少个满足条件的元素，可以使用 numpy 提供的机制进行比较，会得到一个具有同样长度的数组，里面的每一个元素为是否满足条件所对应的布尔值。然后直接通过 sum 运算即可，因为运算的时候，True 会被解释成 1，False 会被解释成 0。

```
import numpy as np
```

```
2
3  arr = np.array([2, 4, 7, 3, 5])
4  print(arr > 4)  # [False False  True False  True]
5  print(np.sum(arr > 4))  # 2
6  # 也可以和整数直接运算
7  print(2.2 + True)  # 3.2
```

所以在 Python 中，整数可以和布尔值进行运算，看似不可思议，但又在情理之中。

填充基类的子类列表

到这里，PyType_Ready 还剩下最后一个重要的动作：设置基类的子类列表。

在PyTypeObject中，有一个tp_subclasses，这个东西在 PyType_Ready 完成之后，将会是一个 list 对象，其中存放着所有直接继承该类的类对象。

```
1  class Base:
2      pass
3
4  class A(Base):
5      pass
6
7  class B(A):
8      pass
9
10 # A 继承 Base
11 print(
12     Base.__subclasses__()
13 )  # [<class '__main__.A'>]
```

tp_subclasses对应Python的__subclasses__，当调用的时候会打印直接继承自己的类。注意是直接继承，间接继承不算，所以上面打印的列表里面只有 A 没有 B。

那么问题来了，这一步是何时发生的呢？Base 怎么知道 A 继承了它呢？很简单，A 在继承 Base 的操作之后，也会将自身设置到 Base 的 tp_subclasses 中。

而在 PyType_Ready 中，这一步是通过调用 add_subclass 实现的。

```
1  int
2  PyType_Ready(PyTypeObject *type)
3  {
4      PyObject *dict, *bases;
5      PyTypeObject *base;
6      Py_ssize_t i, n;
7
8      //拿到所有的基类
9      bases = type->tp_bases;
10     n = PyTuple_GET_SIZE(bases);
11     //挨个遍历，将自身加入到基类的 tp_subclasses 中
12     //因为一个类可以直接继承很多基类，而每个基类都要添加
13     for (i = 0; i < n; i++) {
14         PyObject *b = PyTuple_GET_ITEM(bases, i);
15         if (PyType_Check(b) &&
16             add_subclass((PyTypeObject *)b, type) < 0)
17             goto error;
18     }
19     //...
```

```
20 }
```

当然啦，一个类可以继承多个基类，一个基类也可以被多个子类继承，比如 object。

```
1 print(object.__subclasses__())
```

打印会输出非常多的内容，因为 Python 里面所有的类都继承 object，而在创建这些类的时候，都会将自身加入到 object 的 tp_subclasses 里面。

到了这里，我们算是完整地剖析了 PyType_Ready 的动作，我们用了三篇文章将类型对象的初始化介绍完毕。

可以看到，虚拟机对类对象进行了多种繁杂的改造工作，可以包括以下几部分：

- 设置类型信息，基类及基类列表；
- 填充属性字典；
- 确定 MRO 顺序列表；
- 基于 MRO 顺序列表从基类继承操作；
- 设置基类的子类列表，或者说，将自身加入到基类的 tp_subclasses 中；

不同的类型，有些操作也会有一些不同的行为，但整体是一致的。因此具体到某个特定类型，可以自己跟踪 PyType_Ready 的具体过程。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》72. 自定义类对象的底层实现与 metaclass（上）

下一篇 ›
《源码探秘 CPython》70. 类对象属性字典的填充