Python 2.3 方法解析顺序

发行版本 *3.13.0rc2*

Guido van Rossum and the Python development team

九月 19, 2024

Python Software Foundation
Email: docs@python.org

Contents

备注

这是一份历史性的文档，作为官方文档的附录提供。这里所讨论的方法解析顺序在 Python 2.3 中 被引入，但在之后的版本中仍然被使用 -- 包括 Python 3。

由 Michele Simionato 撰写。

摘要

本文档的目标读者是那些希望理解 *Python 2.3* 中使用的 *C3* 方法解析顺序的 *Python* 程序员。虽然它不是为新手准备的，但它具有很强的教学性，包含许多实用的例子。据我所知还没有其他公开文档涵盖相同的领域，因此它应该是有用的。

致谢：

Python 邮件列表中所有对我表示支持的人。*Paul Foley*，他指出了各种不精确之处并让我添加了本地优先排序的部分。*David Goodger* 在 *reStructuredText* 格式化方面的帮助。*David Mertz* 在编辑方面提供的帮助。最后，*Guido van Rossum* 热心地将本文档添加到官方 *Python 2.3* 主页。

1 开始

Felix qui potuit rerum cognoscere causas -- Virgilius

事情开始于 Samuele Pedroni 在 Python 开发邮件列表上的一个帖子[1]。在他的帖子里，Samuele 表示 Python 2.2 方法解析顺序不是单调的并提议用 C3 方法解析顺序来替代它。Guido 认同他的意见因此现在 Python 2.3 使用了 C3。C3 方法本身与 Python 没有关系，因为它由使用 Dylan 的人发明并在一篇针对 lisp 程序员的论文中描述[2]。本文给出了面向希望理解这项改变的理由的 Python 使用者的（尽可能）易读的 C3 算法相关讨论。

首先，我要指出我即将介绍的情况仅作用于在 Python 2.2 中引入的 *新式类*: *经典类*将保持其原有的方法解析顺序，深度优先并且从左至右。因此，不存在对经典类原有代码的破坏；而且虽然在原理上存在对 Python 2.2 新式类代码的破坏，但在实践中 C3 解析顺序与 Python 2.2 方法解析顺序存在不同的情况是如此稀少以至于不会真正破坏原有代码。所以：

不必害怕！

此外，除非你高强度地使用多重继承并且有复杂的层级结构，否则你就不需要理解 C3 算法，可以轻松地跳过本文。另一方面，如果你真的想知道多重继承是如何工作的，那么本文就是为你准备的。好消息是事情并没有你想象的那么复杂。

让我们从一些基本的定义开始。

1) 在一个复杂的多重继承层级结构中给定一个类 C，要指明方法的覆盖顺序，即 C 的祖先的顺序是一项并不轻松的任务。
2) 类 C 的祖先列表（包括类本身）从最近的祖先到最远的祖先排序，称为类优先级列表或 C 的 *线性化*。
3) *方法解析顺序* (MRO) 是构造线性化的规则集合。在 Python 的语境中，术语"C 的 MRO" 也会被用作类 C 的线性化的同义词。
4) 举例来说，在单继承层级结构的情况下，如果 C 是 C1 的子类，而 C1 是 C2 的子类，那么 C 的线性化就是简单的列表 [C, C1 , C2]。但是，对于多继承层级结构，线性化的构造就比较麻烦了，因为要构造一个尊重 *局部优先级排序*和 *单调性*的线性化将更为困难。
5) 我稍后会讨论局部优先级顺序问题，但我可以先在这里给出单调性的定义。当以下情况为真时一个 MRO 就是单调的: *如果在 C 的线性化中 C1 先于 C2，那么在 C 的任何子类的线性化中 C1 都先于 C2*。在其他情况下，派生新类的无害操作就可能会改变方法的解析顺序，从而可能引入非常微妙的程序错误。稍后将举例说明这种情况。
6) 并非所有的类都允许线性化。在复杂的层级结构中，有些情况下不可能派生出一个类使其线性化遵循所有需要的属性。

在此我举一个例子来说明这种情况。考虑以下层级结构

```
>>> O = object
>>> class X(O): pass
>>> class Y(O): pass
>>> class A(X,Y): pass
>>> class B(Y,X): pass
```

它可以用以下继承图来表示，其中我用 O 来标记 `object` 类，它是任何新式类层级结构的起点：

```
 ------------
|            |
|    O       |
|  /   \     |
 - X     Y   /
   |   / | /
   | /   |/
   A     B
```

(续下页)

[1] 由 Samuele Pedroni 在 python-dev 发起的讨论: https://mail.python.org/pipermail/python-dev/2002-October/029035.html
[2] 论文 *A Monotonic Superclass Linearization for Dylan*: https://doi.org/10.1145/236337.236343

(接上页)

```
  \   /
    ?
```

在此情况下，从 A 和 B 派生新类是不可能的，因为在 A 中 X 先于 Y，但在 B 中 Y 先于 X，因此在 C 中方法解析顺序将出现歧义。

Python 2.3 在此情况下会引发异常 (TypeError: MRO conflict among bases Y, X) 以防止程序员在无意中创建有歧义的层级结构。Python 2.2 不会引发异常，而是会选择一个 *临时顺序* (在本例中为 CABXYO)。

2 C3 方法解析顺序

让我们引入一些适用于接下来的讨论的简单标记法。我会使用这样的快捷标记:

```
C1 C2 ... CN
```

来表示类列表 [C1, C2, ... , CN]。

列表的 *head* 是其第一个元素:

```
head = C1
```

而 *tail* 则是列表的其余元素:

```
tail = C2 ... CN.
```

我还将使用这样的标记:

```
C + (C1 C2 ... CN) = C C1 C2 ... CN
```

来表示列表 [C] + [C1，C2，...，CN] 的总和。

现在我就可以解释 MRO 在 Python 2.3 中的工作原理了。

考虑多重继承层级结构中的类 C，C 继承自基类 B1, B2, ... , BN。我们想要计算类 C 的线性化 L[C]。规则如下:

> *C 的线性化就是 C 加上父类的线性化和父类列表的执行合并的总和。*

使用符号标记法:

```
L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)
```

特别地，如果 C 为 `object` 类，它是没有父类的，其线性化很简单:

```
L[object] = object.
```

不过，在通常情况下我们需要根据以下预设规则来计算合并结果:

> *取第一个列表的 head，即 L[B1][0]；如果这个 head 不在任何其他列表的 tail 内，则将其添加到 C 的线性化中，并在合并结果中将其从列表中移除，否则如果下一个列表的 head 是好的 head 则使用它。然后重复上述操作直到所有类都被移除或是无法找到好的 head。在这种情况下将无法构造合并结果，Python 2.3 将拒绝创建类 C 并将引发异常。*

这一预设规则可以确保合并操作 *保留顺序*，如果顺序能被保留的话。在另一方面，如果顺序无法被保留（如上文讨论的顺序严重不一致的例子）则无法计算合并结果。

如果 C 只有一个父类（单一继承）则合并结果的计算将很简单；在这种情况下:

```
L[C(B)] = C + merge(L[B],B) = C + L[B]
```

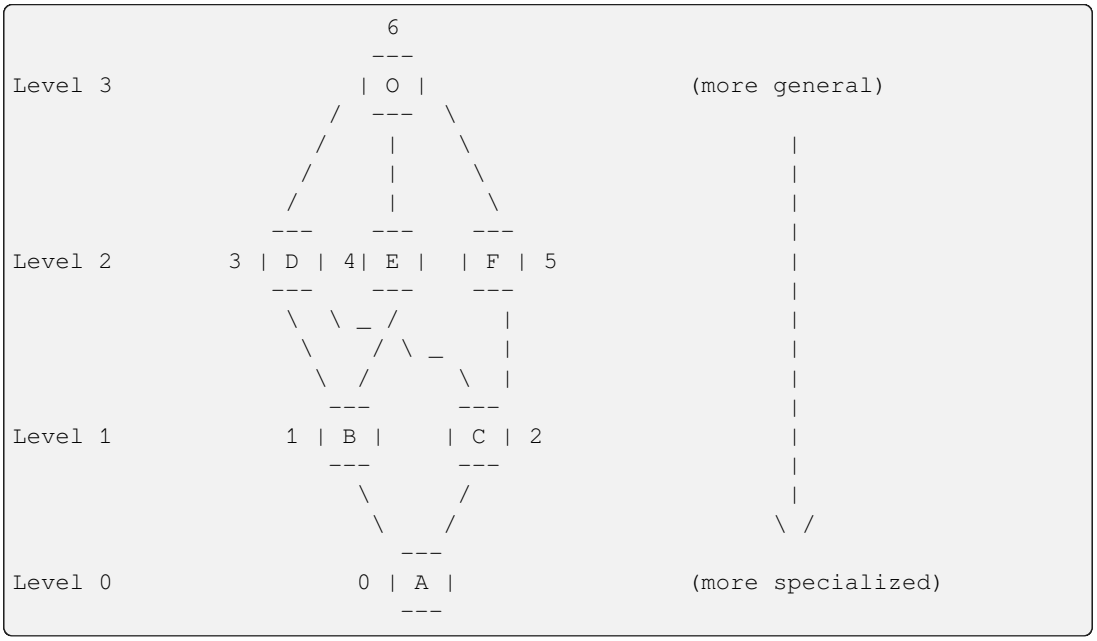
不过，对于多重继承的情况事情就会比较麻烦，如果不举几个例子我估计你是无法理解具体规则的;-)

3 例子

第一个例子。考虑以下层级结构：

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

在这种情况下继承图可以绘制为：



O、D、E 和 F 的线性化很简单：

```
L[O] = O
L[D] = D O
L[E] = E O
L[F] = F O
```

B 的线性化可以被计算为：

```
L[B] = B + merge(DO, EO, DE)
```

我们看到 D 是一个好的 head，因此我们使用它这样就可以简化为计算 `merge(O,EO,E)`。现在 O 不是一个好的 head，因为它在序列 EO 的 tail 内。在这种情况下规则要求我们必须跳到下一个序列。然后我们可以看到 E 是一个好的 head；我们使用它这样就可以简化为计算 `merge(O,O)` 从而得到 O。因此：

```
L[B] =  B D E O
```

使用同样的步骤我们将发现：

```
L[C] = C + merge(DO,FO,DF)
     = C + D + merge(O,FO,F)
     = C + D + F + merge(O,O)
     = C D F O
```

现在我们可以计算：

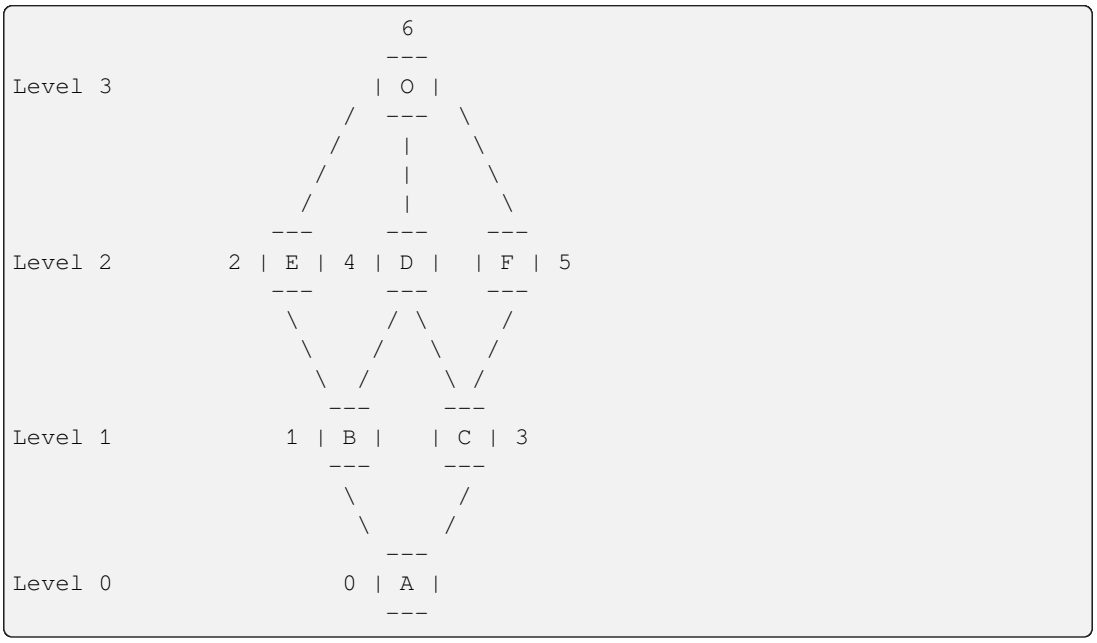
```
L[A] = A + merge(BDEO,CDFO,BC)
     = A + B + merge(DEO,CDFO,C)
     = A + B + C + merge(DEO,DFO)
     = A + B + C + D + merge(EO,FO)
     = A + B + C + D + E + merge(O,FO)
     = A + B + C + D + E + F + merge(O,O)
     = A B C D E F O
```

在这个例子中，线性化按照继承级别进行了良好的排序，即级别越低（即更特化的类）优先级越高（见继承图）。然而，这并不是一般的情况。

我把计算第二个例子的线性化作为一个练习留给读者完成：

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```

与前一例子的唯一区别在于 B(D,E) --> B(E,D)；然而即使是这样一个小小的改动也完全改变了层级结构的顺序：



请注意处在层级结构第二层级的类 E，它先于处在层级结构第一层级的类 C，也就是说，E 比 C 更特化，即便它处在更高的层级。

懒惰的程序员可以直接获取 Python 2.2 的 MRO，因为在这种情况下它与 Python 2.3 的线性化相吻合。只需发起调用类 A 的.mro() 方法即可：

```
>>> A.mro()
[<class 'A'>, <class 'B'>, <class 'E'>,
<class 'C'>, <class 'D'>, <class 'F'>,
<class 'object'>]
```

最后，让我来讲解第一小节所讨论的例子，其中涉及严重的顺序不一致问题。在这种情况下，可以直接计算 O、X、Y、A 和 B 的线性化：

```
L[O] = 0
L[X] = X O
```

(续下页)

(接上页)

```
L[Y] = Y O
L[A] = A X Y O
L[B] = B Y X O
```

然而，要计算继承自 A 和 B 的类 C 的线性化则是不可能的:

```
L[C] = C + merge(AXYO, BYXO, AB)
     = C + A + merge(XYO, BYXO, B)
     = C + A + B + merge(XYO, YXO)
```

此时我们无法合并列表 XYO 和 YXO，因为 X 在 YXO 的 tail 内，而 Y 在 XYO 的 tail 内：因此没有好的 head 从而 C3 算法将停止。Python 2.3 将引发一个错误并拒绝创建类 C。

4 坏的方法解析顺序

当一个 MOR 破坏了诸如局部优先顺序和单调性等基本属性时它就是 坏的。在本节中，我将证明 Python 2.2 中经典类的 MRO 和新式类的 MRO 都是坏的。

从局部优先顺序开始会更简单。请看下面的例子:

```
>>> F=type('Food',(),{'remember2buy':'spam'})
>>> E=type('Eggs',(F,),{'remember2buy':'eggs'})
>>> G=type('GoodFood',(F,E),{}) # under Python 2.3 this is an error!
```

继承图如下

```
             O
             |
(buy spam)   F
             | \
             | E   (buy eggs)
             | /
             G

      (buy eggs or spam ?)
```

我们看到类 G 继承自 F 和 E，其中 F 先于 E：因此我们预期属性 *G.remember2buy* 会被 *F.rembermer2buy* 而不是被 *E.remember2buy* 继承：然而 Python 2.2 给出的结果是

```
>>> G.remember2buy
'eggs'
```

这是对局部优先顺序的破坏因为在 Python 2.2 对 G 进行线性化时，局部优先列表即 G 的父类列表中的顺序并不会被保留:

```
L[G,P22]= G E F object    # F *follows* E
```

有人可能会说在 Python 2.2 的线性化中 F 在 E 之后的原因是 F 的特化程度低于 E，因为 F 是 E 的超类；然而打破局部优先排序是相当反直觉且容易导致错误的。这一点因为它与旧式类不同而尤其明显:

```
>>> class F: remember2buy='spam'
>>> class E(F): remember2buy='eggs'
>>> class G(F,E): pass
>>> G.remember2buy
'spam'
```

在这种情况下 MRO 为 GFEF 并保留了局部优先顺序。

作为一般规则，应当避免像上面这样的层级结构，因为不清楚 F 是否应该重写 E，反之亦然。Python 2.3 通过在创建类 G 时引发异常解决了这个歧义性问题，有效地阻止了程序员生成有歧义的层级结构。其原因是 C3 算法在执行以下合并时将会失败:

```
merge(FO,EFO,FE)
```

这是无法计算的，因为 F 在 EFO 的 tail 内而 E 在 FE 的 tail 内。

真正的解决办法是设计一个无歧义的层级结构，即从 E 和 F（更具体的说是第一个）而不是从 F 和 E 派生出 G；在这种情况下 MRO 毫无疑问就是 GEF。

```
          O
          |
          F (spam)
        / |
(eggs)  E |
        \ |
          G
            (eggs, no doubt)
```

Python 2.3 会强迫程序员编写好的（或者，至少不那么容易出错的）层级结构。

与此相关的一点，我要指出 Python 2.3 的算法足够聪明，它能识别明显的错误，比如父类列表中重复的类:

```
>>> class A(object): pass
>>> class C(A,A): pass # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: duplicate base class A
```

在这种情况下，Python 2.2（包括经典类和新式类）则不会引发任何异常。

最后，我想指出我们从这个例子中汲取的两点教训:

1. 尽管名称如此，MRO 是决定属性的解析顺序，而不仅仅是方法的解析顺序;
2. Python 爱好者的默认食物是 spam ! (不过你已经知道这一点了;-)

在讨论了局部优先顺序问题之后，现在再让我来讲解单调性。我的目标是证明经典类和 Python 2.2 新式类的 MRO 都不是单调的。

要证明经典类的 MRO 是非单调的相当简单，只要看一下这个钻石形图就够了:

```
   C
  / \
 /   \
A     B
 \   /
  \ /
   D
```

人们很容易发现其中的不一致性:

```
L[B,P21] = B C        # B precedes C : B's methods win
L[D,P21] = D A C B C  # B follows C  : C's methods win!
```

另一方面，Python 2.2 和 Python 2.3 的 MRO 则没有问题，它们都将给出以下结果:

```
L[D] = D A B C
```

Guido 在他的文章[3] 中指出经典的 MRO 在实践中并没有那么坏，因为人们通常可以避免经典类形成钻石

[3] Guido van Rossum 的文章，*Unifying types and classes in Python 2.2*: https://web.archive.org/web/20140210194412/http://www.python.org/download/releases/2.2.2/descrintro

形继承图。但是所有新式类都继承自 object，因此钻石形继承图是不可避免的并且在每个多重继承图中都会出现不一致性。

Python 2.2 的 MRO 使打破单调性变得困难，但并非不可能。下面是最初由 Samuele Pedroni 提供的例子，显示 Python 2.2 的 MRO 是非单调的：

```
>>> class A(object): pass
>>> class B(object): pass
>>> class C(object): pass
>>> class D(object): pass
>>> class E(object): pass
>>> class K1(A,B,C): pass
>>> class K2(D,B,E): pass
>>> class K3(D,A):   pass
>>> class Z(K1,K2,K3): pass
```

以下是根据 C3 MRO 进行的线性化 (读者应当将验证这些线性化作为练习并绘制继承图;-)

```
L[A] = A O
L[B] = B O
L[C] = C O
L[D] = D O
L[E] = E O
L[K1]= K1 A B C O
L[K2]= K2 D B E O
L[K3]= K3 D A O
L[Z] = Z K1 K2 K3 D A B C E O
```

Python 2.2 对 A、B、C、D、E、K1、K2 和 K3 给出了完全相同的线性化，但对 Z 则给出了不同的线性化:

```
L[Z,P22] = Z K1 K3 A K2 D B C E O
```

很明显这种线性化是 错误的，因为 A 在 D 之前，而在 K3 的线性化中 A 在 D 之后。换句话说，在 K3 中由 D 派生的方法会重写由 A 派生的方法，但在仍为 K3 子类的 Z 中，由 A 派生的方法会重写由 D 派生的方法！这破坏了单调性。此外，Z 的 Python 2.2 线性化也与局部优先顺序不一致，因为类 Z 的局部优先列表是 [K1, K2, K3] (K2 先于 K3)，而在 Z 的线性化中则是 K2 跟随 K3。这些问题解释了为什么 2.2 规则被否定而改用 C3 规则。

5 结束

本节是为没有耐心的读者准备的，他们会跳过前面的所有章节，直接跳到结尾。这部分也是为懒惰的程序员准备的，因为他们不想动脑筋。最后，这部分也是为有些自负的程序员准备的，否则他/她就不会去阅读一篇关于多重继承层次结构中的 C3 方法解析顺序的论文了;-) 这三个优点合在一起（而 不是分开）应该得到一个奖励：这个奖励就是一个简短的 Python 2.2 脚本，它可以在不影响你的大脑的情况下计算 2.3 MRO。只需修改最后一行就可以尝试我在本文中讨论的各种示例:

```
#<mro.py>

"""C3 algorithm by Samuele Pedroni (with readability enhanced by me)."""

class __metaclass__(type):
    "All classes are metamagically modified to be nicely printed"
    __repr__ = lambda cls: cls.__name__

class ex_2:
    "Serious order disagreement" #From Guido
    class O: pass
    class X(O): pass
    class Y(O): pass
```

(续下页)

(接上页)

```
    class A(X,Y): pass
    class B(Y,X): pass
    try:
        class Z(A,B): pass #creates Z(A,B) in Python 2.2
    except TypeError:
        pass # Z(A,B) cannot be created in Python 2.3

class ex_5:
    "My first example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(D,E): pass
    class A(B,C): pass

class ex_6:
    "My second example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(E,D): pass
    class A(B,C): pass

class ex_9:
    "Difference between Python 2.2 MRO and C3" #From Samuele
    class O: pass
    class A(O): pass
    class B(O): pass
    class C(O): pass
    class D(O): pass
    class E(O): pass
    class K1(A,B,C): pass
    class K2(D,B,E): pass
    class K3(D,A): pass
    class Z(K1,K2,K3): pass

def merge(seqs):
    print '\n\nCPL[%s]=%s' % (seqs[0][0],seqs),
    res = []; i=0
    while 1:
      nonemptyseqs=[seq for seq in seqs if seq]
      if not nonemptyseqs: return res
      i+=1; print '\n',i,'round: candidates...',
      for seq in nonemptyseqs: # find merge candidates among seq heads
          cand = seq[0]; print ' ',cand,
          nothead=[s for s in nonemptyseqs if cand in s[1:]]
          if nothead: cand=None #reject candidate
          else: break
      if not cand: raise "Inconsistent hierarchy"
      res.append(cand)
      for seq in nonemptyseqs: # remove cand
          if seq[0] == cand: del seq[0]

def mro(C):
    "Compute the class precedence list (mro) according to C3"
    return merge([[C]]+map(mro,C.__bases__)+[list(C.__bases__)])
```

(续下页)

(接上页)

```
def print_mro(C):
    print '\nMRO[%s]=%s' % (C,mro(C))
    print '\nP22 MRO[%s]=%s' % (C,C.mro())

print_mro(ex_9.Z)

#</mro.py>
```

就是这样了朋友们，

好好享受吧！

6 参考资源