8.7 调用父类方法¶

问题¶

你想在子类中调用父类的某个已经被覆盖的方法。

解决方案¶

为了调用父类(超类)的一个方法，可以使用 `super()` 函数，比如：

```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()  # Call parent spam()
```

`super()` 函数的一个常见用法是在 `__init__()` 方法中确保父类被正确的初始化了：

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1
```

`super()` 的另外一个常见用法出现在覆盖Python特殊方法的代码中，比如：

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value) # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

在上面代码中， `__setattr__()` 的实现包含一个名字检查。 如果某个属性名以下划线(_)开头，就通过 `super()` 调用原始的 `__setattr__()` ， 否则的话就委派给内部的代理对象 `self._obj` 去处理。 这看上去有点意思，因为就算没有显式的指明某个类的父类， `super()` 仍然可以有效的工作。

讨论¶

实际上，大家对于在Python中如何正确使用 `super()` 函数普遍知之甚少。 你有时候会看到像下面这样直接调用父类的一个方法：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

尽管对于大部分代码而言这么做没什么问题，但是在更复杂的涉及到多继承的代码中就有可能导致很奇怪的问题发生。比如，考虑如下的情况：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

如果你运行这段代码就会发现 `Base.__init__()` 被调用两次，如下所示：

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
B.__init__
C.__init__
>>>
```

可能两次调用 `Base.__init__()` 没什么坏处，但有时候却不是。 另一方面，假设你在代码中换成使用 `super()` ，结果就很完美了：

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')
```

```
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()  # Only one call to super() here
        print('C.__init__')
```

运行这个新版本后，你会发现每个 `__init__()` 方法只会被调用一次了：

```
>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>
```

为了弄清它的原理，我们需要花点时间解释下Python是如何实现继承的。 对于你定义的每一个类，Python会计算出一个所谓的方法解析顺序(MRO)列表。 这个MRO列表就是一个简单的所有基类的线性顺序表。例如：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class '__main__.Base'>, <class 'object'>)
>>>
```

为了实现继承，Python会在MRO列表上从左到右开始查找基类，直到找到第一个匹配这个属性的类为止。

而这个MRO列表的构造是通过一个C3线性化算法来实现的。 我们不去深究这个算法的数学原理，它实际上就是合并所有父类的MRO列表并遵循如下三条准则：

子类会先于父类被检查
多个父类会根据它们在列表中的顺序被检查
如果对下一个类存在两个合法的选择，选择第一个父类
老实说，你所要知道的就是MRO列表中的类顺序会让你定义的任意类层级关系变得有意义。

当你使用 `super()` 函数时，Python会在MRO列表上继续搜索下一个类。 只要每个重定义的方法统一使用 `super()` 并只调用它一次， 那么控制流最终会遍历完整个MRO列表，每个方法也只会被调用一次。 这也是为什么在第二个例子中你不会调用两次 `Base.__init__()` 的原因。

`super()` 有个令人吃惊的地方是它并不一定去查找某个类在MRO中下一个直接父类， 你甚至可以在一个没有直接父类的类中使用它。例如，考虑如下这个类：

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

如果你试着直接使用这个类就会出错：

```
>>> a = A()
>>> a.spam()
```

```
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

但是，如果你使用多继承的话看看会发生什么：

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

你可以看到在类A中使用 `super().spam()` 实际上调用的是跟类A毫无关系的类B中的 `spam()` 方法。 这个用类C的MRO列表就可以完全解释清楚了：

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
>>>
```

在定义混入类的时候这样使用 `super()` 是很普遍的。可以参考8.13和8.18小节。

然而，由于 `super()` 可能会调用不是你想要的方法，你应该遵循一些通用原则。 首先，确保在继承体系中所有相同名字的方法拥有可兼容的参数签名(比如相同的参数个数和参数名称)。 这样可以确保 `super()` 调用一个非直接父类方法时不会出错。 其次，最好确保最顶层的类提供了这个方法的实现，这样的话在MRO上面的查找链肯定可以找到某个确定的方法。

在Python社区中对于 `super()` 的使用有时候会引来一些争议。 尽管如此，如果一切顺利的话，你应该在你最新代码中使用它。 Raymond Hettinger为此写了一篇非常好的文章 “Python’s super() Considered Super!”， 通过大量的例子向我们解释了为什么 `super()` 是极好的。