**super() 的入门使用**

在类的继承中，如果重定义某个方法，该方法会覆盖父类的同名方法，但有时，我们希望能同时实现父类的功能，这时，我们就需要调用父类的方法了，可通过使用 super 来实现，比如：

**class** **Animal**(object):

**def** **\_\_init\_\_**(self, name):

self.name = name

**def** **greet**(self):

**print** 'Hello, I am %s.' % self.name

**class** **Dog**(Animal):

**def** **greet**(self):

super(Dog, self).greet() # Python3 可使用 super().greet()

**print** 'WangWang...'

在上面，Animal 是父类，Dog 是子类，我们在 Dog 类重定义了 greet 方法，为了能同时实现父类的功能，我们又调用了父类的方法，看下面的使用：

**>>>** dog = Dog('dog')

**>>>** dog.greet()

Hello, I am dog.

WangWang..

super 的一个最常见用法可以说是在子类中调用父类的初始化方法了，比如：

**class** **Base**(object):

**def** **\_\_init\_\_**(self, a, b):

self.a = a

self.b = b

**class** **A**(Base):

**def** **\_\_init\_\_**(self, a, b, c):

super(A, self).\_\_init\_\_(a, b) # Python3 可使用 super().\_\_init\_\_(a, b)

self.c = c

**深入 super()**

看了上面的使用，你可能会觉得 super 的使用很简单，无非就是获取了父类，并调用父类的方法。其实，在上面的情况下，super 获得的类刚好是父类，但在其他情况就不一定了，super 其实和父类没有实质性的关联。

让我们看一个稍微复杂的例子，涉及到多重继承，代码如下：

**class** **Base**(object):

**def** **\_\_init\_\_**(self):

**print** "enter Base"

**print** "leave Base"

**class** **A**(Base):

**def** **\_\_init\_\_**(self):

**print** "enter A"

super(A, self).\_\_init\_\_()

**print** "leave A"

**class** **B**(Base):

**def** **\_\_init\_\_**(self):

**print** "enter B"

super(B, self).\_\_init\_\_()

**print** "leave B"

**class** **C**(A, B):

**def** **\_\_init\_\_**(self):

**print** "enter C"

super(C, self).\_\_init\_\_()

**print** "leave C"

其中，Base 是父类，A, B 继承自 Base, C 继承自 A, B，它们的继承关系如下：

Base

/ \

/ \

A B

\ /

\ /

C

现在，让我们看一下使用：

**>>>** c = C()

enter C

enter A

enter B

enter Base

leave Base

leave B

leave A

leave C

如果你认为 super 代表『调用父类的方法』，那你很可能会疑惑为什么 enter A 的下一句不是 enter Base 而是 enter B。原因是，**super 和父类没有实质性的关联**，现在让我们搞清 super 是怎么运作的。

**MRO 列表**

事实上，对于你定义的每一个类，Python 会计算出一个**方法解析顺序（Method Resolution Order, MRO）列表**，**它代表了类继承的顺序**，我们可以使用下面的方式获得某个类的 MRO 列表：

**>>>** C.mro() # or C.\_\_mro\_\_ or C().\_\_class\_\_.mro()

[\_\_main\_\_.C, \_\_main\_\_.A, \_\_main\_\_.B, \_\_main\_\_.Base, object]

那这个 MRO 列表的顺序是怎么定的呢，它是通过一个 [C3 线性化算法](https://www.python.org/download/releases/2.3/mro/)来实现的，这里我们就不去深究这个算法了，感兴趣的读者可以自己去了解一下，总的来说，一个类的 MRO 列表就是合并所有父类的 MRO 列表，并遵循以下三条原则：

* 子类永远在父类前面
* 如果有多个父类，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

**super 原理**

super 的工作原理如下：

**def** **super**(cls, inst):

mro = inst.\_\_class\_\_.mro()

**return** mro[mro.index(cls) + 1]

其中，cls 代表类，inst 代表实例，上面的代码做了两件事：

* 获取 inst 的 MRO 列表
* 查找 cls 在当前 MRO 列表中的 index, 并返回它的下一个类，即 mro[index + 1]

当你使用 super(cls, inst) 时，Python 会在 inst 的 MRO 列表上搜索 cls 的下一个类。

现在，让我们回到前面的例子。

首先看类 C 的 \_\_init\_\_ 方法：

**super**(C, self).\_\_init\_\_()

这里的 self 是当前 C 的实例，self.\_\_class\_\_.mro() 结果是：

[\_\_main\_\_.C, \_\_main\_\_.A, \_\_main\_\_.B, \_\_main\_\_.Base, object]

可以看到，C 的下一个类是 A，于是，跳到了 A 的 \_\_init\_\_，这时会打印出 enter A，并执行下面一行代码：

super(A, self).\_\_init\_\_()

注意，这里的 self 也是当前 C 的实例，MRO 列表跟上面是一样的，搜索 A 在 MRO 中的下一个类，发现是 B，于是，跳到了 B 的 \_\_init\_\_，这时会打印出 enter B，而不是 enter Base。

整个过程还是比较清晰的，关键是要理解 super 的工作方式，而不是想当然地认为 super 调用了父类的方法。