由Python的super()函数想到的

python-super

由Python的super()函数想到的

首先看一下`super()`函数的定义:

```
super([type [,object-or-type]])

Return a **proxy object** that deleg
```

返回一个**代理对象**, 这个对象负责将方法调用**分配**给第一个参数的一个**父类或者同辈的类**去完成.

parent or sibling class 如何确定?

第一个参数的`__mro__`属性决定了搜索的顺序, super指的的是 **MRO**(Method Resolution Order) 中的下一个类, **而不一定是父类**!
super()和getattr() 都使用`__mro__`属性来解析搜索顺序, `__mro__`实际上是一个只读的元组.

MRO中类的顺序是怎么排的呢?

实际上MRO列表本身是根据一种C3的线性化处理技术确定的, 理论说明可以参考这里, 这里只简单说明一下原则:

> 在MRO中, 基类永远出现在派生类的后面, 如果有多个基类, 基类的相对顺序不变.

MRO实际上是对*继承树*做层序遍历的结果, 把一棵带有结构的**树**变成了一个**线性的表**, 所以沿着这个列表一直往上, 就可以无重复

的遍历完整棵树, 也就解决了多继承中的Diamond问题.

比如说:

```
class Root:
    pass

class A(Root):
    pass

class B(Root):
    pass

class C(A, B):
    pass

print(C.__mro__)

# 输出结果为:
# (<class '__main__.C'>, <class '__m
```

super()实际返回的是一个代理的super对象!

调用super()这个构造方法时, 只是返回一个super()对象, 并不做其他的操作.
然后对这个super对象进行方法调用时, 发生的事情如下:

1. 找到第一个参数的`__mro__`列表中的下一个**直接定义了该方法**的类, 并实例化出一个对象
2. 然后将这个对象的`self`变量绑定到第二个参数上, 返回这个对象

举个例子:

```
class Root:
    def __init__(self):
        print('Root')

class A(Root):
    def __init__(self):
        super().__init__() # 等同于s
```

在`A`的构造方法中, 先调用super()得到一个`super对象`, 然后向这个对象调用**init**方法,

随笔档案

2016年7月 (1)

阅读排行榜

1. 由Python的super()函数想到的(1948)

这是super对象会搜索A的__mro__列表, 找到第一个定义了__init__方法的类, 于是就找到了Root, 然后调用Root.__init__(self), 这里的self是super()的第二个参数, 是编译器自动填充的, 也就是A的__init__的第一个参数, 这样就完成对__init__方法调用的分配.

注意: 在许多语言的继承中, 子类必须调用父类的构造方法, 就是为了保证子类的对象能够**填充**上父类的属性! 而不是初始化一个父类对象...(我之前就一直是这么理解的..). Python中就好多了, 所谓的调用父类构造方法, 就是明明白白地把self传给父类的构造方法, ~~我的小身子骨就这么交给你了, 随便你怎么折腾吧~~😂

参数说明

```
super() -> same as super(__class__,
super(type) -> unbound super object
super(type, obj) -> bound super obje
super(type, type2) -> bound super ob

  Typical use to call a cooperative
    class C(B):
        def meth(self, arg):
            super().meth(arg)
    This works for class methods too
    class C(B):
        @classmethod
        def cmeth(cls, arg):
            super().cmeth(arg)
```

- 如果提供了第二个参数, 则找到的父类对象的self就绑定到这个参数上, 后面调用这个对象的方法时, 可以自动地隐式传递self.

  > 如果第二个参数是一个对象, 则isinstance(obj, type)必须为True. 如果第二个参数为一个类型, 则issubclass(type2, type)必须为True

- 如果没有传递第二个参数, 那么返回的对象就是Unbound, 调用这个

unbound对象的方法时需要手动传递第一个参数, 类似于`Base.__int__(self, a, b)`.

- 不带参数的super()只能用在类定义中(因为依赖于caller的第二个参数), 编译器会自动根据当前定义的类填充参数.

也就是说, 后面所有调用super返回对象的方法时, 第一个参数`self`都是`super()`的第二个参数. 因为Python中所谓的方法, 就是一个第一个参数为self的函数, 一般在调用方法的时候`a.b()`会隐式的将`a`赋给`b()`的第一个参数.

super()的两种常见用法:

1. 单继承中, *super*用来指代隐式指代父类, 避免直接使用父类的名字
2. 多继承中, 解决Diamond问题 (TODO)

对面向对象的理解

其实我觉得Python里面这样的语法更容易理解面向对象的本质, 比Java中隐式地传`this`更容易理解.
所谓函数, 就是一段代码, 接受输入, 返回输出. 所谓方法, 就是一个函数有了一个隐式传递的参数. 所以方法就是一段代码, 是类的所有实例共享的, 唯一不同的是各个实例调用的时候传给方法的`this` 或者`self`不一样而已.

构造方法是什么呢? 其实也是一个实例方法啊, 它只有在对象生成了之后才能调用, 所以Python中`__init__`方法的参数是`self`啊. 调用构造方法时其实已经为对象分配了内存, 构造方法只是起到初始化的作用, 也就是为这段内存里面赋点初值而已.

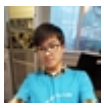
Java中所谓的*静态变量*其实也就是*类的变量*, 其实也就是为类也分配了内存, 里面存了这些变量, 所以Python中的**类对象**我觉得是很合理的, 也比Java要直观. 至于静态方法, 那就与对象一点关系都没有了, 本质就是个独立的函数, 只不过写在了类里面而已. 而Python中的**classmethod**其实也是一种

静态方法, 不过它会依赖于`cls`对象, 这个`cls`就是类对象, 但是只要想用这个方法, 类对象必然是存在的, 不像实例对象一样需要手动的实例化, 所以classmethod也可以看做是一种静态变量. 而**staticmethod**就是真正的静态方法了, 是独立的函数, 不依赖任何对象.

Java中的实例方法是必须依赖于对象存在的, 因为要隐式的传输`this`, 如果对象不存在这个`this`也没法隐式了. 所以在静态方法中是没有`this`指针的, 也就没法调用实例方法. 而Python中的实例方法是可以通过类名来调用的, 只不过因为这时候`self`没办法隐式传递, 所以必须得显式地传递.

分类: python

标签: python, super, 面向对象



相关博文：

· Python利用new创建一个类的对象
· Python进阶-继承中的MRO与super
· ZetCode PyQt4 tutorial work with menus, toolbars, a statusbar, and a main application window
· Python的程序结构[2] -> 类/Class[4] -> 内建类 super
· Python内置函数(63)——super

最新新闻：

· 华为的西欧新故事：进入市场18年，终于反超苹果紧追三星
· 从常胜到常败，腾讯是不是BAT中最焦虑的那一家？
· 王坚：别再让住在回龙观的年轻人堵在路上
· LG申请新专利：手机或将搭载16颗摄像头
· 腾讯QQ发布公告：因安全功能升级 不再提供单独帐号申诉入口
» 更多新闻...