

## 一、\_\_slots\_\_

### 问题的引出

都是字典惹的祸。

字典为了提升查询效率，必须用空间换时间。

一般来说一个对象，属性多一点，都存储在字典中便于查询，问题不大。

但是如果数百万个对象，那么字典占得就有点大了。

这个时候，能不能把属性字典 `__dict__` 省了？

Python提供了 `__slots__`

```
class A:
    X = 1
    def __init__(self):
        self.y = 5
        self.z = 6

    def show(self):
        print(self.X, self.y, self.z)

a = A()
print(A.__dict__) # ?
print(a.__dict__) # ?
```

#### 思考

上面2个字典，谁的字典是个问题？

实例多达百万个的时候，这么多存放实例属性的字典是个问题

```
class A:
    X = 1

    __slots__ = ('y', 'z') # 元组
    # __slots__ = ['y', 'z'] # 可以吗
    # __slots__ = 'y', 'z' # 可以吗
    # __slots__ = 'y'

    def __init__(self):
        self.y = 5
```

```

        #self.z = 6

    def show(self):
        print(self.X, self.y)

a = A()
a.show()
#
print('A', A.__dict__)
#print('obj', a.__dict__)
print(a.__slots__)

```

`__slots__` 告诉解释器，实例的属性都叫什么，一般来说，既然要节约内存，最好还是使用元组比较好。

一旦类提供了 `__slots__`，就阻止实例产生 `__dict__` 来保存实例的属性。

尝试为实例a动态增加属性

```
a.newx = 5
```

返回AttributeError: 'A' object has no attribute 'newx'

说明实例不可以动态增加属性了

A.NEWX = 20，这是可以的，因为这个类属性。

## 继承

```

class A:
    X = 1

    __slots__ = ('y', 'z') # 元组

    def __init__(self):
        self.y = 5
        #self.z = 6

    def show(self):
        print(self.X, self.y)

a = A()
a.show()
#

```

```
print('A', A.__dict__)
#print('obj', a.__dict__)
print(a.__slots__)
```

```
class B(A):
    pass
```

```
print('B', B().__dict__)
```

`__slots__` 不影响子类实例，不会继承下去，除非子类里面自己也定义了 `__slots__`。

## 应用场景

使用需要构建在数百万以上对象，且内存容量较为紧张，实例的属性简单、固定且不用动态增加的场景。

## 二、未实现和未实现异常

```
print(type(NotImplemented))
print(type(NotImplementedError))

# <class 'NotImplementedType'>
# <class 'type'>

#raise NotImplemented
raise NotImplementedError
```

`NotImplemented` 是个值，单值，是 `NotImplementedType` 类的实例

`NotImplementedError` 是类型，是异常，返回 `type`

## 三、运算符重载中的反向方法

前面学习过运算符重载的方法，例如add和iadd

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

    def __add__(self, other):
        print(self, 'add')
        return self.x + other.x

    def __iadd__(self, other):
        print(self, 'iadd')
        return A(self.x + other.x)

    def __radd__(self, other):
        print(self, 'radd')
        return self.x + other.x
```

```
a = A(4)
b = A(5)
print(a, b)
print(a + b)
print(b + a)
b += a
a += b
```

# 运行结果

```
<__main__.A object at 0x0000000000B8C550> <__main__.A object at 0x0000000000B8C240>
<__main__.A object at 0x0000000000B8C550> add
9
<__main__.A object at 0x0000000000B8C240> add
9
<__main__.A object at 0x0000000000B8C240> iadd
<__main__.A object at 0x0000000000B8C550> iadd
```

`__radd__` 方法根本没有执行过，为什么？

因为都是A的实例，都是调用的 `__add__`，无非就是实例a还是b调用而已。

测试一下 `a + 1`

```

class A:
    def __init__(self, x):
        self.x = x

    def __add__(self, other):
        print(self, 'add')
        return self.x + other.x

    def __iadd__(self, other):
        print(self, 'iadd')
        return A(self.x + other.x)

    def __radd__(self, other):
        print(self, 'radd')
        return self.x + other.x

```

```
a = A(4)
```

```
a + 1
```

# 运行结果

```
<__main__.A object at 0x0000000000B7DA58> add
```

```
Traceback (most recent call last):
```

```
File "test.py", line 22, in <module>
```

```
    a + 1
```

```
File "test.py", line 10, in __add__
```

```
    return self.x + other.x
```

```
AttributeError: 'int' object has no attribute 'x'
```

出现了AttributeError，因为1是int类型，没有x这个属性，还是 `__add__` 被执行了。

测试1 + a，运行结果如下

```
<__main__.A object at 0x0000000000DADA58> radd
```

```
Traceback (most recent call last):
```

```
File "test.py", line 22, in <module>
```

```
    1 + a
```

```
File "test.py", line 18, in __radd__
```

```
    return self.x + other.x
```

```
AttributeError: 'int' object has no attribute 'x'
```

这次执行的是实例a的 `__radd__` 方法。

`1 + a` 等价于 `1.__add__(a)`，而int类型实现了 `__add__` 方法的，为什么却不抛出异常，而是执行了实例a的 `__radd__` 方法？

再看一个例子

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

    def __add__(self, other):
        print(self, 'add')
        return self.x + other.x

    def __iadd__(self, other):
        print(self, 'iadd')
        return A(self.x + other.x)

    def __radd__(self, other):
        print(self, 'radd')
        return self.x + other.x

class B: # 未实现__add__
    def __init__(self, x):
        self.x = x

a = A(4)
b = B(10)
print(a + b)
print(b + a)

# 运行结果
<__main__.A object at 0x0000000000B6C240> add
14
<__main__.A object at 0x0000000000B6C240> radd
14
```

`b + a` 等价于 `b.__add__(a)`，但是类B没有实现 `__add__` 方法，就去找a的 `__radd__` 方法  
`1 + a` 等价于 `1.__add__(a)`，而int类型实现了 `__add__` 方法的，不过这个方法对于这种加法的返回值是 `NotImplemented`，解释器发现是这个值，就会发起对第二操作对象的 `__radd__` 方法的

调用。

1 + a能解决吗？

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

    def __add__(self, other):
        print(self, 'add')
        try:
            x = other.x
            return self.x + other.x
        except AttributeError:
            try:
                x = int(other)
            except:
                x = 0
            return self.x + x

    def __iadd__(self, other):
        print(self, 'iadd')
        return A(self.x + other.x)

    def __radd__(self, other):
        print(self, 'radd')
        return self + other

class B:
    def __init__(self, x):
        self.x = x

a = A(4)
b = B(10)
print(a + b)
print(b + a)
print(a + 2)
print(2 + a)
print(a + 'abc')
print('abc' + a)
```

---

'abc' + a，字符串也实现了 `__add__` 方法，不过默认是处理不了和其他类型的加法，就返回 `NotImplemented`。

