python内置方法和属性应用：反射和单例

个人简介： wedo实验君, 数据分析师；热爱生活，热爱写作

1. 前言

python除了丰富的第三方库外，本身也提供了一些内在的方法和底层的一些属性，大家比较常用的如dict、list、set、min、max、range、sorted等。笔者最近在做项目框架时涉及到一些不是很常用的方法和属性，在本文中和大家做下分享。

2. 内置方法和函数介绍

- enumerate

  如果你需要遍历可迭代的对象，有需要获取它的序号，可以用`enumerate`, 每一个next返回的是一个tuple

```
list1 = [1, 2, 3, 4]
list2 = [4, 3, 2, 1]
for idx, value in enumerate(list1):
    print(idx, value, list2[idx])
# 0 1 4
# 1 2 3
# 2 3 2
# 3 4 1
```

- zip zip从参数中的多个迭代器取元素组合成一个新的迭代器;

```
# 给list加上序号
b = [4, 3, 2, 1]
for i in zip(range(len(b)), b):
    print(i)
# (0, 4)
# (1, 3)
# (2, 2)
# (3, 1)
```

- globals()： 一个描述当前执行过程中全局符号表的字典，可以看出你执行的所有过程
- id(object)： python对象的唯一标识
- staticmethod 类静态函数注解

```
@staticmethod
def test():
    print('this is static method')
```

```
Foo.test = test
Foo.test()
```

- 类的属性 我们来看下一个类的申明，如下：

```
class Foo():
    """this is test class"""
    def __init__(self, name):
        self.name = name

    def run(self):
        print('running')
```

```
# 列出类的所有成员和属性
dir(Foo)
['__class__',
'__delattr__',
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'run']


# 类的注释
Foo.__doc__
# 'this is test class'

# 类自定义属性
```

```
Foo.__dict__
mappingproxy({'__module__': '__main__',
              '__doc__': 'this is test class',
              '__init__': <function __main__.Foo.__init__(self, name)>,
              'run': <function __main__.Foo.run(self)>,
              '__dict__': <attribute '__dict__' of 'Foo' objects>,
              '__weakref__': <attribute '__weakref__' of 'Foo' objects>})

# 类的父类
Foo.__base__

# 类的名字
Foo.__name__
```

类的实例化和初始化

```
# python类先通过__new__实例化，再调用__init__进行初始化类成员
foo = Foo('milk')
```

类的属性添加和访问

```
# 类的访问
foo.name
foo.run()

# 可以通过setattr 动态的添加属性
def method():
    print("cow")

setattr(foo, "type", "cow")
setattr(foo, "getcow", method)
# cow
foo.type
foo.getcow()

# 动态删除属性 delattr
delattr(foo, "type")

# getattr 获取成员属性
if hasattr(foo, "run"): # 判断是否有属性
    func = getattr(foo, "run")
    func()
```

3. 单例模式应用

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。单例模式要求在类的使用过程中只实例化一次，所有对象都共享一个实例。创建的方法是在实例的时候判断下是否已经实例过了，有则返回实例化过的全局实例。python是如何实现的呢？关键是找到实例化的地方，对就是前面说的__new__

```
class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            cls._instance = object.__new__(cls)
        return cls._instance

    def __init__(self, name):
        self.name = name


a = Singleton('name1')
b = Singleton('name2')
print(id(a), id(b))
print(a.name, b.name)
# 1689352213112 1689352213112
# name2 name2
```

4. 反射应用

反射在许多框架中都有使用到，简单就是通过类的名称（字符串）来实例化类。一个典型的场景就是通过配置的方式来动态控制类的执行，比如定时任务的执行，通过维护每个定时任务类的执行时间，在执行时间到的时候，通过反射方式实例化类，执行任务，在java中也非常的常见。

python的实现可以通过上面说的getattr获取模块中的类， 通过methodcaller来调用方法。我们来看一个简单的例子

```
import importlib
from operator import methodcaller

class Foo():
    """this is test class"""
    def __init__(self, name):
        self.name = name

    def run(self, info):
        print('running %s' % info)

# 类所在的模块，默认情况__main__，  可以通过Foo.__dict__ 中'__module__'获取
api_module = importlib.import_module('__main__')
# getattr获取模块中的类， 这里Foo是字符串哦
clazz = getattr(api_module, 'Foo')

# 实例化
params = ["milk"]
instance = clazz(*params)
```

```
# 方法调用， 方法也是字符串methodcaller(方法名， 方法参数)
task_result = methodcaller("run", "reflection")(instance)

# running reflection
```

5. 总结

本文通过分享了python内置方法和属性， 并在单例模式和反射中进行应用。希望对你有帮助，欢迎交流 @mintel 要点总结如下：

- dir下类
- 查看类自定义属性__dict__
- __new__实例化类，__init__初始化类
- getattr 获取属性
- setattr 设置属性
- 记住importlib和methodcaller