Python自省（反射）指南

作者: AstralWind 来源: 博客园 发布时间: 2011-01-07 10:46 阅读: 3170 次 推荐: 0 原文链接 [收藏]

首先通过一个例子来看一下本文中可能用到的对象和相关概念。

```
#coding: UTF-8
import sys #  模块，sys指向这个模块对象
import inspect
def foo(): pass # 函数，foo指向这个函数对象
class Cat(object): # 类，Cat指向这个类对象
    def __init__(self, name='kitty'):
        self.name = name
    def sayHi(self): #  实例方法，sayHi指向这个方法对象，使用类或实例.sayHi访问
        print self.name, 'says Hi!' # 访问名为name的字段，使用实例.name访问

cat = Cat() # cat是Cat类的实例对象

print Cat.sayHi # 使用类名访问实例方法时，方法是未绑定的(unbound)
print cat.sayHi # 使用实例访问实例方法时，方法是绑定的(bound)
```

有时候我们会碰到这样的需求，需要执行对象的某个方法，或是需要对对象的某个字段赋值，而方法名或是字段名在编码代码时并不能确定，需要通过参数传递字符串的形式输入。举个具体的例子：当我们需要实现一个通用的DBM框架时，可能需要对数据对象的字段赋值，但我们无法预知用到这个框架的数据对象都有些什么字段，换言之，我们在写框架的时候需要通过某种机制访问未知的属性。

这个机制被称为反射（反过来让对象告诉我们他是什么），或是自省（让对象自己告诉我们他是什么，好吧我承认括号里是我瞎掰的- -#），用于实现在运行时获取未知对象的信息。反射是个很吓唬人的名词，听起来高深莫测，在一般的编程语言里反射相对其他概念来说稍显复杂，一般来说都是作为高级主题来讲；但在Python中反射非常简单，用起来几乎感觉不到与其他的代码有区别，使用反射获取到的函数和方法可以像平常一样加上括号直接调用，获取到类后可以直接构造实例；不过获取到的字段不能直接赋值，因为拿到的其实是另一个指向同一个地方的引用，赋值只能改变当前的这个引用而已。

1. 访问对象的属性

以下列出了几个内建方法，可以用来检查或是访问对象的属性。这些方法可以用于任意对象而不仅仅是例子中的Cat实例对象；Python中一切都是对象。

```
cat = Cat('kitty')

print cat.name # 访问实例属性
cat.sayHi() # 调用实例方法

print dir(cat) # 获取实例的属性名，以列表形式返回
if hasattr(cat, 'name'): # 检查实例是否有这个属性
    setattr(cat, 'name', 'tiger') # same as: a.name = 'tiger'
print getattr(cat, 'name') # same as: print a.name

getattr(cat, 'sayHi')() # same as: cat.sayHi()
```

- **dir([obj]):**
  调用这个方法将返回包含obj大多数属性名的列表（会有一些特殊的属性不包含在内）。obj的默认值是当前的模块对象。
- **hasattr(obj, attr):**

搜索文章

推荐链接
程序员问答平台，解决您的技术难题
找优秀程序员，就在博客园
国家职业资格证书.NET/Java免费培训(上海)
遇到技术问题怎么办，在园子里"找找看"

编程语言热门文章
高级编程语言的发展历程
函数式编程是一个倒退
苹果编程语言和 API 的未来
金旭亮：第一门编程语言选谁？
Asp.net MVC2 使用经验，性能优化建议

编程语言最新文章
十年前的Java企业应用开发世界
使用面向对象的技术创建高级 Web 应用程序
丑陋的Java API
我再也不想在任何头文件中看到"using namespace
深入PHP使用技巧之变量

最新新闻
CyanogenMod开发者成立公司
Linus Torvalds开玩笑的承认被要求在内核中植入后门
前任天堂社长山内溥逝世 享年85岁
Gartner：今年全球应用程序下载量将突破1020亿
基层码农忆搜搜：坐守腾讯数据金矿，缺"人和"

热门新闻
从事色情业的富士康女工：后悔没能早些做这个兼职
一个在清华附近蹲了17年的男人
长见识啦！世界上最神奇的数字是：142857
诺基亚重回俄罗斯手机销量首位，一个不错的开始
微软WP要来新队友了

这个方法用于检查obj是否有一个名为attr的值的属性，返回一个布尔值。

- **getattr(obj, attr):**
  调用这个方法将返回obj中名为attr值的属性的值，例如如果attr为'bar'，则返回obj.bar。
- **setattr(obj, attr, val):**
  调用这个方法将给obj的名为attr的值的属性赋值为val。例如如果attr为'bar'，则相当于obj.bar = val。

2. 访问对象的元数据

当你对一个你构造的对象使用dir()时，可能会发现列表中的很多属性并不是你定义的。这些属性一般保存了对象的元数据，比如类的__name__属性保存了类名。大部分这些属性都可以修改，不过改动它们意义并不是很大；修改其中某些属性如function.func_code还可能导致很难发现的问题，所以改改name什么的就好了，其他的属性不要在不了解后果的情况下修改。

接下来列出特定对象的一些特殊属性。另外，Python的文档中有提到部分属性不一定会一直提供，下文中将以红色的星号*标记，使用前你可以先打开解释器确认一下。

2.0. 准备工作：确定对象的类型

在types模块中定义了全部的Python内置类型，结合内置方法isinstance()就可以确定对象的具体类型了。

- **isinstance(object, classinfo):**
  检查object是不是classinfo中列举出的类型，返回布尔值。classinfo可以是一个具体的类型，也可以是多个类型的元组或列表。

types模块中仅仅定义了类型，而inspect模块中封装了很多检查类型的方法，比直接使用types模块更为轻松，所以这里不给出关于types的更多介绍，如有需要可以直接查看types模块的文档说明。本文第3节中介绍了inspect模块。

2.1. 模块(module)

- __doc__: 文档字符串。如果模块没有文档，这个值是None。
- *__name__: 始终是定义时的模块名；即使你使用import .. as 为它取了别名，或是赋值给了另一个变量名。
- *__dict__: 包含了模块里可用的属性名-属性的字典；也就是可以使用模块名.属性名访问的对象。
- __file__: 包含了该模块的文件路径。需要注意的是内建的模块没有这个属性，访问它会抛出异常！

```
import fnmatch as m
print m.__doc__.splitlines()[0] # Filename matching with shell patterns.
print m.__name__                # fnmatch
print m.__file__                # /usr/lib/python2.6/fnmatch.pyc
print m.__dict__.items()[0]     # ('fnmatchcase', )
```

2.2. 类(class)

- __doc__: 文档字符串。如果类没有文档，这个值是None。
- *__name__: 始终是定义时的类名。
- *__dict__: 包含了类里可用的属性名-属性的字典；也就是可以使用类名.属性名访问的对象。
- __module__: 包含该类的定义的模块名；需要注意，是字符串形式的模块名而不是模块对象。
- *__bases__: 直接父类对象的元组；但不包含继承树更上层的其他类，比如父类的父类。

```
print Cat.__doc__           # None
print Cat.__name__          # Cat
print Cat.__module__        # __main__
print Cat.__bases__         # (,)
print Cat.__dict__          # {'__module__': '__main__', ...}
```

2.3. 实例(instance)

实例是指类实例化以后的对象。

- *__dict__: 包含了可用的属性名-属性字典。
- *__class__: 该实例的类对象。对于类Cat，cat.__class__ == Cat 为 True。

```
print cat.__dict__
```

```
print cat.__class__
print cat.__class__ == Cat # True
```

2.4. 内建函数和方法(built-in functions and methods)

根据定义，内建的(built-in)模块是指使用C写的模块，可以通过sys模块的builtin_module_names字段查看都有哪些模块是内建的。这些模块中的函数和方法可以使用的属性比较少，不过一般也不需要在代码中查看它们的信息。

- __doc__: 函数或方法的文档。
- __name__: 函数或方法定义时的名字。
- __self__: 仅方法可用，如果是绑定的(bound)，则指向调用该方法的类（如果是类方法）或实例（如果是实例方法），否则为None。
- *__module__: 函数或方法所在的模块名。

2.5. 函数(function)

这里特指非内建的函数。注意，在类中使用def定义的是方法，方法与函数虽然有相似的行为，但它们是不同的概念。

- __doc__: 函数的文档；另外也可以用属性名func_doc。
- __name__: 函数定义时的函数名；另外也可以用属性名func_name。
- *__module__: 包含该函数定义的模块名；同样注意，是模块名而不是模块对象。
- *__dict__: 函数的可用属性；另外也可以用属性名func_dict。
  不要忘了函数也是对象，可以使用函数.属性名访问属性（赋值时如果属性不存在将新增一个），或使用内置函数has/get/setattr()访问。不过，在函数中保存属性的意义并不大。
- func_defaults: 这个属性保存了函数的参数默认值元组；因为默认值总是靠后的参数才有，所以不使用字典的形式也是可以与参数对应上的。
- func_code: 这个属性指向一个该函数对应的code对象，code对象中定义了其他的一些特殊属性，将在下文中另外介绍。
- func_globals: 这个属性指向当前的全局命名空间而不是定义函数时的全局命名空间，用处不大，并且是只读的。
- *func_closure: 这个属性仅当函数是一个闭包时有效，指向一个保存了所引用到的外部函数的变量cell的元组，如果该函数不是一个内部函数，则始终为None。这个属性也是只读的。

下面的代码演示了func_closure：

```
<div class="cnblogs_code"><pre><div><!--<br /><br />Code highlighting produced by
Actipro CodeHighlighter (freeware)<br />http://www.CodeHighlighter.com/<br /><br />-->
<span style="color: #008000;">#</span><span style="color: #008000;">coding: UTF-
8</span><span style="color: #008000;"><br></span><span style="color:
#0000ff;">def</span><span style="color: #000000;"> foo():<br>    n </span><span
style="color: #000000;">=</span><span style="color: #000000;"> </span><span
style="color: #000000;">1</span><span style="color: #000000;"><br>    </span><span
style="color: #0000ff;">def</span><span style="color: #000000;"> bar():<br>
</span><span style="color: #0000ff;">print</span><span style="color: #000000;"> n
</span><span style="color: #008000;">#</span><span style="color: #008000;"> 引用非全局的
外部变量n，构造一个闭包</span><span style="color: #008000;"><br></span><span style="color:
#000000;">    n </span><span style="color: #000000;">=</span><span style="color:
#000000;"> </span><span style="color: #000000;">2</span><span style="color: #000000;">
<br>    </span><span style="color: #0000ff;">return</span><span style="color:
#000000;"> bar<br><br>closure </span><span style="color: #000000;">=</span><span
style="color: #000000;"> foo()<br></span><span style="color: #0000ff;">print</span>
<span style="color: #000000;"> closure.func_closure<br></span></div></pre>
</div>
# 使用dir()得知cell对象有一个cell_contents属性可以获得值
print closure.func_closure[0].cell_contents # 2
```

由这个例子可以看到，遇到未知的对象使用dir()是一个很好的主意 ：）

2.6. 方法(method)

方法虽然不是函数，但可以理解为在函数外面加了一层外壳；拿到方法里实际的函数以后，就可以使用2.5节的属性了。

- __doc__: 与函数相同。
- __name__: 与函数相同。
- *__module__: 与函数相同。
- im_func: 使用这个属性可以拿到方法里实际的函数对象的引用。另外如果是2.6以上的版本，还可以使用属性名__func__。
- im_self: 如果是绑定的(bound)，则指向调用该方法的类（如果是类方法）或实例（如果是实例方法），否则为None。如果是2.6以上的版本，还可以使用属性名__self__。
- im_class: 实际调用该方法的类，或实际调用该方法的实例的类。注意不是方法的定义所在的类，如果有继承关系的话。

```
im = cat.sayHi
print im.im_func
print im.im_self # cat
print im.im_class # Cat
```

这里讨论的是一般的实例方法，另外还有两种特殊的方法分别是类方法(classmethod)和静态方法(staticmethod)。类方法还是方法，不过因为需要使用类名调用，所以他始终是绑定的；而静态方法可以看成是在类的命名空间里的函数（需要使用类名调用的函数），它只能使用函数的属性，不能使用方法的属性。

2.7. 生成器(generator)

生成器是调用一个生成器函数(generator function)返回的对象，多用于集合对象的迭代。

- __iter__: 仅仅是一个可迭代的标记。
- gi_code: 生成器对应的code对象。
- gi_frame: 生成器对应的frame对象。
- gi_running: 生成器函数是否在执行。生成器函数在yield以后、执行yield的下一行代码前处于frozen状态，此时这个属性的值为0。
- next|close|send|throw: 这是几个可调用的方法，并不包含元数据信息，如何使用可以查看生成器的相关文档。

```
def gen():
    for n in xrange(5):
        yield n
g = gen()
print g             # <generator object gen at 0x...>
print g.gi_code     # <code object gen at 0x...>
print g.gi_frame    # <frame object at 0x...>
print g.gi_running  # 0
print g.next()      # 0
print g.next()      # 1
for n in g:
    print n,        # 2 3 4
```

接下来讨论的是几个不常用到的内置对象类型。这些类型在正常的编码过程中应该很少接触，除非你正在自己实现一个解释器或开发环境之类。所以这里只列出一部分属性，如果需要一份完整的属性表或想进一步了解，可以查看文末列出的参考文档。

2.8. 代码块(code)

代码块可以由类源代码、函数源代码或是一个简单的语句代码编译得到。这里我们只考虑它指代一个函数时的情况；2.5节中我们曾提到可以使用函数的func_code属性获取到它。code的属性全部是只读的。

- co_argcount: 普通参数的总数，不包括*参数和**参数。
- co_names: 所有的参数名（包括*参数和**参数）和局部变量名的元组。
- co_varnames: 所有的局部变量名的元组。
- co_filename: 源代码所在的文件名。
- co_flags: 这是一个数值，每一个二进制位都包含了特定信息。较关注的是0b100(0x4)和

0b1000(0x8)，如果co_flags & 0b100 != 0，说明使用了*args参数；如果co_flags & 0b1000 != 0，说明使用了**kwargs参数。另外，如果co_flags & 0b100000(0x20) != 0，则说明这是一个生成器函数(generator function)。

```
co = cat.sayHi.func_code
print co.co_argcount        # 1
print co.co_names           # ('name',)
print co.co_varnames        # ('self',)
print co.co_flags & 0b100   # 0
```

2.9. 栈帧(frame)

栈帧表示程序运行时函数调用栈中的某一帧。函数没有属性可以获取它，因为它在函数调用时才会产生，而生成器则是由函数调用返回的，所以有属性指向栈帧。想要获得某个函数相关的栈帧，则必须在调用这个函数且这个函数尚未返回时获取。你可以使用sys模块的_getframe()函数、或inspect模块的currentframe()函数获取当前栈帧。这里列出来的属性全部是只读的。

- f_back: 调用栈的前一帧。
- f_code: 栈帧对应的code对象。
- f_locals: 用在当前栈帧时与内建函数locals()相同，但你可以先获取其他帧然后使用这个属性获取那个帧的locals()。
- f_globals: 用在当前栈帧时与内建函数globals()相同，但你可以先获取其他帧……。

```
def add(x, y=1):
    f = inspect.currentframe()
    print f.f_locals    # same as locals()
    print f.f_back      # <frame object at 0x...>
    return x+y
add(2)
```

2.10. 追踪(traceback)

追踪是在出现异常时用于回溯的对象，与栈帧相反。由于异常时才会构建，而异常未捕获时会一直向外层栈帧抛出，所以需要使用try才能见到这个对象。你可以使用sys模块的exc_info()函数获得它，这个函数返回一个元组，元素分别是异常类型、异常对象、追踪。traceback的属性全部是只读的。

- tb_next: 追踪的下一个追踪对象。
- tb_frame: 当前追踪对应的栈帧。
- tb_lineno: 当前追踪的行号。

```
def div(x, y):
    try:
        return x/y
    except:
        tb = sys.exc_info()[2]  # return (exc_type, exc_value, traceback)
        print tb
        print tb.tb_lineno      # "return x/y" 的行号
div(1, 0)
```

3. 使用inspect模块

inspect模块提供了一系列函数用于帮助使用自省。下面仅列出较常用的一些函数，想获得全部的函数资料可以查看inspect模块的文档。

3.1. 检查对象类型

- **is{module|class|function|method|builtin}(obj):**
  检查对象是否为模块、类、函数、方法、内建函数或方法。
- **isroutine(obj):**
  用于检查对象是否为函数、方法、内建函数或方法等等可调用类型。用这个方法会比多个is*()更方便，不过它的实现仍然是用了多个is*()。

```
im = cat.sayHi
if inspect.isroutine(im):
    im()
```

对于实现了__call__的类实例，这个方法会返回False。如果目的是只要可以直接调用就需要是True的话，不妨使用**isinstance(obj, collections.Callable)**这种形式。我也不知道为什么Callable会在collections模块中，抱歉！我大概是因为collections模块中包含了很多其他的ABC(Abstract Base Class)的缘故吧：）

3.2. 获取对象信息

- **getmembers(object[, predicate]):**

  这个方法是dir()的扩展版，它会将dir()找到的名字对应的属性一并返回，形如[(name, value), ...]。另外，predicate是一个方法的引用，如果指定，则应当接受value作为参数并返回一个布尔值，如果为False，相应的属性将不会返回。使用is*作为第二个参数可以过滤出指定类型的属性。

- **getmodule(object):**

  还在为第2节中的__module__属性只返回字符串而遗憾吗？这个方法一定可以满足你，它返回object的定义所在的模块对象。

- **get{file|sourcefile}(object):**

  获取object的定义所在的模块的文件名|源代码文件名（如果没有则返回None）。用于内建的对象（内建模块、类、函数、方法）上时会抛出TypeError异常。

- **get{source|sourcelines}(object):**

  获取object的定义的源代码，以字符串|字符串列表返回。代码无法访问时会抛出IOError异常。只能用于module/class/function/method/code/frame/traceack对象。

- **getargspec(func):**

  仅用于方法，获取方法声明的参数，返回元组，分别是(普通参数名的列表, *参数名, **参数名, 默认值元组)。如果没有值，将是空列表和3个None。如果是2.6以上版本，将返回一个命名元组(Named Tuple)，即除了索引外还可以使用属性名访问元组中的元素。

```
def add(x, y=1, *z):
    return x + y + sum(z)
print inspect.getargspec(add)
#ArgSpec(args=['x', 'y'], varargs='z', keywords=None, defaults=(1,))
```

- **getargvalues(frame):**

  仅用于栈帧，获取栈帧中保存的该次函数调用的参数值，返回元组，分别是(普通参数名的列表, *参数名, **参数名, 帧的locals())。如果是2.6以上版本，将返回一个命名元组(Named Tuple)，即除了索引外还可以使用属性名访问元组中的元素。

```
def add(x, y=1, *z):
    print inspect.getargvalues(inspect.currentframe())
    return x + y + sum(z)
add(2)
#ArgInfo(args=['x', 'y'], varargs='z', keywords=None, locals={'y': 1, 'x': 2, 'z':
 ()})
```

- **getcallargs(func[, *args][, **kwds]):**

  返回使用args和kwds调用该方法时各参数对应的值的字典。这个方法仅在2.7版本中才有。

- **getmro(cls):**

  返回一个类型元组，查找类属性时按照这个元组中的顺序。如果是新式类，与cls.__mro__结果一样。但旧式类没有__mro__这个属性，直接使用这个属性会报异常，所以这个方法还是有它的价值的。

```
print inspect.getmro(Cat)
#(<class '__main__.Cat'>, <type 'object'>)
print Cat.__mro__
#(<class '__main__.Cat'>, <type 'object'>)
class Dog: pass
print inspect.getmro(Dog)
#(<class __main__.Dog at 0x...>,)
print Dog.__mro__ # AttributeError
```

- **currentframe():**

  返回当前的栈帧对象。

其他的操作frame和traceback的函数请查阅inspect模块的文档，用的比较少，这里就不多介绍了。

参考资料：

1. The standard type hierarchy[官方文档][英文]
2. inspect — Inspect live objects[官方文档][英文]

0 0

标签： Python 自省 反射

« 上一篇：坐标高速插入，移动和查询算法

» 下一篇：异常处理之ThreadException、unhandledException及多线程异常处理

找优秀程序员，就在博客园

关于我们 联系我们 广告服务 沪ICP备09004260号 © 2013 博客园