关于作者

廖雪峰 北京 朝阳区
+加关注

使用元类

阅读: 406832

type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个`Hello`的class，就写一个`hello.py`模块：

```
class Hello(object):
    def hello(self, name='world'):
        print('Hello, %s.' % name)
```

当Python解释器载入`hello`模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个`Hello`的class对象，测试如下：

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

`type()`函数可以查看一个类型或变量的类型，`Hello`是一个class，它的类型就是`type`，而`h`是一个实例，它的类型就是class `Hello`。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用`type()`函数。

`type()`函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过`type()`函数创建出`Hello`类，而无需通过`class Hello(object)...`的定义：

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

要创建一个class对象，`type()`函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数`fn`绑定到方法名`hello`上。

通过`type()`函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class。

正常情况下，我们都用`class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

metaclass

除了使用`type()`动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个`add`方法：

定义`ListMetaclass`，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

```
# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数`metaclass`：

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

当我们传入关键字参数`metaclass`时，魔术就生效了，它指示Python解释器在创建`MyList`时，要通过`ListMetaclass.__new__()`来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

`__new__()`方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合。

测试一下`MyList`是否可以调用`add()`方法：

```
>>> L = MyList()
>>> L.add(1)
>> L
[1]
```

而普通的`list`没有`add()`方法：

```
>>> L2 = list()
>>> L2.add(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'
```

动态修改有什么意义？直接在`MyList`定义中写上`add()`方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个`User`类来操作对应的数据库表`User`，我们期待他写出这样的代码：

```
class User(Model):
    # 定义类的属性到列的映射：
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例：
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
# 保存到数据库：
u.save()
```

其中，父类`Model`和属性类型`StringField`、`IntegerField`是由ORM框架提供的，剩下的魔术方法比如`save()`全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义`Field`类，它负责保存数据库表的字段名和字段类型：

```
class Field(object):

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

    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__, self.name)
```

在`Field`的基础上，进一步定义各种类型的`Field`，比如`StringField`，`IntegerField`等等：

```
class StringField(Field):

    def __init__(self, name):
        super(StringField, self).__init__(name, 'varchar(100)')

class IntegerField(Field):

    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')
```

下一步，就是编写最复杂的`ModelMetaclass`了：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        print('Found model: %s' % name)
        mappings = dict()
        for k, v in attrs.items():
            if isinstance(v, Field):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
        for k in mappings.keys():
            attrs.pop(k)
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        attrs['__table__'] = name # 假设表名和类名一致
        return type.__new__(cls, name, bases, attrs)
```

以及基类`Model`：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(params))
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

当用户定义一个`class User(Model)`时，Python解释器首先在当前类`User`的定义中查找`metaclass`，如果没有找到，就继续在父类`Model`中查找`metaclass`，找到了，就使用`Model`中定义的`metaclass`的`ModelMetaclass`来创建`User`类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在`ModelMetaclass`中，一共做了几件事情：

1. 排除掉对`Model`类的修改；
2. 在当前类（比如`User`）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个`__mappings__`的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；
3. 把表名保存到`__table__`中，这里简化为表名默认为类名。

在`Model`类中，就可以定义各种操作数据库的方法，比如`save()`，`delete()`，`find()`，`update`等等。

我们实现了`save()`方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出`INSERT`语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,id) values (?,?,?,?)
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
```

可以看到，`save()`方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架，是不是非常简单？

小结

metaclass是Python中非常具有魔术性的对象，它可以改变类创建时的行为。这种强大的功能使用起来务必小心。

参考源码

create_class_on_the_fly.py

use_metaclass.py

orm.py

读后有收获可以请作者喝咖啡，读后有疑问请加群讨论：

还可以分享给朋友：

分享到微博

上一页 下一页

评论

第二个理解起来有点困难 来加个注释

枫-落J created at July 2, 2016 8:55 PM, Last updated at 2 hours ago