2.6 实现类和对象

::: details INFO
译者：Jesper.Y

来源：2.6 Implementing Classes and Objects

对应：无
:::

在面向对象编程范式（object-oriented programming paradigm）中工作时，我们使用对象的隐喻来指导程序的组织。大部分关于数据表示和操作的逻辑都通过类声明来表达。在本节中，我们将看到类和对象本身可以使用函数和字典来表示。通过以这种方式实现对象系统的目的是为了说明使用对象的隐喻并不需要特定的编程语言。即使在没有内置对象系统的编程语言中，程序也可以是面向对象的。

为了实现对象，我们将放弃点表示法（它需要内置语言支持），而是创建行为方式与内置对象系统的元素非常相似的调度字典。我们已经看到了如何通过调度字典实现消息传递的行为。为了完全实现对象系统，我们在实例、类和基类之间发送消息，它们都是包含属性的字典。

我们不会实现完整的 Python 对象系统，其中包括我们在本文中未涵盖的功能（例如元类（meta-class）和静态方法）。相反，我们将重点放在没有多重继承和内省行为（例如返回实例的类）的用户定义类上。我们的实现不打算严格遵循 Python 类型系统的规范。相反，它旨在实现支持对象隐喻的核心功能。

2.6.1 实例

我们从实例开始。实例拥有可以被设置并检索的具名属性，例如一个银行账户的实例 account 拥有具名属性 balance。我们使用调度字典实现一个实例，这个调度字典可以响应设置和获取（"get" and "set"）属性值的消息。属性本身存储在一个名为 **attributes** 的本地字典中。

正如我们在前面的章节所看到的，字典本身也是抽象的数据类型。我们使用函数实现数据对，使用数据对实现列表，然后使用列表实现字典。当我们使用字典实现一个对象系统时，牢记我们也可以仅仅只使用函数来实现对象。

在开始我们的实现之前，假定我们有一个类的实现，它可以查出任何不属于实例的名称。我们将一个类作为参数传递给 `make_instance` 的形参 `cls`。

```
>>> def make_instance(cls):
        """Return a new object instance, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            else:
                value = cls['get'](name)
                return bind_method(value, instance)
        def set_value(name, value):
            attributes[name] = value
        attributes = {}
        instance = {'get': get_value, 'set': set_value}
        return instance
```

`instance` 是一个能够响应 `get` 和 `set` 消息的调度字典。`set` 消息和 Python 对象系统中的属性赋值操作相对应：所有已经赋值的属性直接存储在对象的本地属性字典中。在 `get` 消息中，如果 `name` 没有出现在本地 `attributes` 字典中，则会在类中查找。如果从类中查找返回的值是一个函数，则这个函数必须被绑定到实例。

对于绑定方法值。`make_instance` 中的 `get` 消息使用 `get_value` 函数在类中找到一个具名属性，然后调用 `bind_method` 函数。只有在这个具名属性是一个函数值时才会绑定一个方法，从函数值创建一个绑定方法值时，它会将 `instance` 作为第一个参数插入到函数值中，从而创建绑定方法值。

```
>>> def bind_method(value, instance):
        """Return a bound method if value is callable, or value otherwise."""
        if callable(value):
            def method(*args):
                return value(instance, *args)
            return method
        else:
            return value
```

在这个定义下，当方法被调用时，第一个参数 `self` 将会被绑定为 `instance` 的值。

2.6.2 类

不论是在 Python 的对象系统中还是在我们自己正在实现的对象系统中，都认为类也是一个对象。为了简单起见，我们可以说类这个对象并没有属于它的类类型（实际在 Python 中，类确实拥有它的类类型，几乎所有的类都共享同一个类类型，叫做 `type`）一个类可以响应 `get` 和 `set` 消息，以及 `new` 消息。

```
>>> def make_class(attributes, base_class=None):
        """Return a new class, which is a dispatch dictionary."""
        def get_value(name):
            if name in attributes:
                return attributes[name]
            elif base_class is not None:
                return base_class['get'](name)
        def set_value(name, value):
            attributes[name] = value
        def new(*args):
            return init_instance(cls, *args)
        cls = {'get': get_value, 'set': set_value, 'new': new}
        return cls
```

不像实例那样，类的 `get` 函数在没有找到属性时并不会查询它的类，而是查询它的基类（base_class）。类不需要进行方法绑定。

对于初始化，`make_class` 中的 `new` 函数会调用 `init_instance`，这个方法首先会创建一个新的实例，然后调用一个叫做 `__init__` 的方法。

```
>>> def init_instance(cls, *args):
    """Return a new object with type cls, initialized with args."""
    instance = make_instance(cls)
    init = cls['get']('__init__')
        if init:
            init(instance, *args)
        return instance
```

这个最终的函数完成了我们的对象系统。我们现在有了实例，它们会在本地 `set` 设置属性，但是在 `get` 获取属性时它们会转而回退到类中。实例从类中查询名称后，会将它自己绑定到函数值以此来创建方法。最后，类可以创建新的实例并且在创建后马上调用它们的 `__init__` 构造器函数。

在这个对象系统中，唯一应该被用户调用的函数是 `make_class`。所有其他的功能都是通过消息传递实现的。类似地，Python 的对象系统通过 `class` 语句调用，并且其他所有功能通过点表达式和对类的调用来实现。

2.6.3 使用已经实现的对象

我们现在重新使用前面的章节中的银行账户的例子。我们将使用我们自己实现的对象系统来创建一个 `Account` 类，一个 `CheckingAccount` 子类，以及为他们各自创建一个实例。

`Account` 类通过 `make_account_class` 函数创建，这个函数和 Python 中的 `class` 语句有着相似结构的，但是最后调用了 `make_class`。

```
>>> def make_account_class():
        """Return the Account class, which has deposit and withdraw methods."""
        interest = 0.02
        def __init__(self, account_holder):
            self['set']('holder', account_holder)
            self['set']('balance', 0)
        def deposit(self, amount):
            """Increase the account balance by amount and return the new
balance."""
            new_balance = self['get']('balance') + amount
            self['set']('balance', new_balance)
            return self['get']('balance')
        def withdraw(self, amount):
            """Decrease the account balance by amount and return the new
balance."""
            balance = self['get']('balance')
            if amount > balance:
                return 'Insufficient funds'
            self['set']('balance', balance - amount)
            return self['get']('balance')
        return make_class(locals())
```

最后对 `locals` 的调用返回一个以字符串为 key 的字典，其中包含了当前局部帧的名称 - 值的绑定。

`Account` 类最终通过赋值完成了实例化。

```
>>> Account = make_account_class()
```

然后一个账户实例通过 `new` 消息被创建，这要求一个与新创建的账户相关联的名称。

```
>>> kirk_account = Account['new']('krik')
```

然后通过对 `krik_account` 发送 `get` 消息就可以检索属性和方法。通过调用方法来更新账户的余额。

```
>>> kirk_account['get']('holder')
'krik'
>>> krik_account['get']('interest')
0.02
>>> kirk_account['get']('deposit')(20)
20
>>> kirk_account['get']('withdraw')(5)
15
```

正如 Python 对象系统那样，设置一个实例的属性不会改变它的类中所对应的属性。

```
>>> kirk_account['set']('interest', 0.04)
>>> Account['get']('interest')
0.02
```

对于继承，我们可以通过重载一部分类的属性来创建一个子类 `CheckingAccount`。在这种情况下，我们改变 `withdraw` 方法来收取费用，同时降低利率。

```
>>> def make_checking_account_class():
        """Return the CheckingAccount class, which imposes a $1 withdrawal
fee."""
        interest = 0.01
        withdraw_fee = 1
        def withdraw(self, amount):
            fee = self['get']('withdraw_fee')
            return Account['get']('withdraw')(self, amount + fee)
        return make_class(locals(), Account)
```

在这个实现中，我们通过子类的 `withdraw` 函数调用了基类 `Account` 的 `withdraw` 函数。正如我们在 Python 的内置对象系统中会做的那样。我们可以像之前一样创建子类本身和一个实例。

```
>>> CheckingAccount = make_checking_account_class()
>>> jack_acct = CheckingAccount['new']('Spock')
```

存款的行为与之前相同，构造函数也是如此。取款通过专门的 `withdraw` 方法收取了 1$ 的费用，并且 `interest` 通过 `CheckingAccount` 获得了新的较低的值。

```
>>> jack_acct['get']('interest')
0.01
>>> jack_acct['get']('deposit')(20)
20
>>> jack_acct['get']('withdraw')(5)
14
```

我们基于字典构建的对象系统在实现上与 Python 中的内置对象系统非常相似。在 Python 中，任何用户定义的类的实例都有一个特殊属性 `__dict__`，它将该对象的本地实例属性存储在一个字典中，就像我们的 `attribute` 字典一样。但 Python 与我们的系统不同之处在于，它区分了一些特殊方法，这些方法与内置函数交互，以确保这些函数对许多不同类型的参数都能正确运行。接下来的一节将详细讨论操作不同类型的函数的问题。