9.20 利用函数注解实现方法重载¶

问题¶

你已经学过怎样使用函数参数注解，那么你可能会想利用它来实现基于类型的方法重载。 但是你不确定应该怎样去实现（或者到底行得通不）。

解决方案¶

本小节的技术是基于一个简单的技术，那就是Python允许参数注解，代码可以像下面这样写：

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3) # Prints Bar 1: 2 3
s.bar('hello') # Prints Bar 2: hello 0
```

下面是我们第一步的尝试，使用到了一个元类和描述器：

```
# multiple.py
import inspect
import types

class MultiMethod:
    '''
    Represents a single multimethod.
    '''
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        '''
        Register a newmethod as a multimethod
        '''
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
                continue
            if parm.annotation is inspect.Parameter.empty:
                raise TypeError(
                    'Argument {} must be annotated with a type'.format(name)
                )
            if not isinstance(parm.annotation, type):
                raise TypeError(
                    'Argument {} annotation must be a type'.format(name)
                )
```

```
            if parm.default is not inspect.Parameter.empty:
                self._methods[tuple(types)] = meth
            types.append(parm.annotation)

        self._methods[tuple(types)] = meth

    def __call__(self, *args):
        '''
        Call a method based on type signature of the arguments
        '''
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            raise TypeError('No matching method for types {}'.format(types))

    def __get__(self, instance, cls):
        '''
        Descriptor method needed to make calls work in a class
        '''
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

class MultiDict(dict):
    '''
    Special dictionary to build multimethods in a metaclass
    '''
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

class MultipleMeta(type):
    '''
    Metaclass that allows multiple dispatch of methods
    '''
    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()
```

为了使用这个类，你可以像下面这样写：

```
class Spam(metaclass=MultipleMeta):
```

```
class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)

    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time

class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)
```

下面是一个交互示例来验证它能正确的工作：

```
>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 42, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)
>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
2012
>>> e.month
12
>>> e.day
3
>>>
```

讨论¶

坦白来讲，相对于通常的代码而已本节使用到了很多的魔法代码。 但是，它却能让我们深入理解元类和描述器的底层工作原理， 并能加深对这些概念的印象。因此，就算你并不会立即去应用本节的技术， 它的一些底层思想却会影响到其它涉及到元类、描述器和函数注解的编程技术。

本节的实现中的主要思路其实是很简单的。 `MutipleMeta` 元类使用它的 `__prepare__()` 方法 来提供一个作为 `MultiDict` 实例的自定义字典。这个跟普通字典不一样的是， `MultiDict` 会在元素被设置的时候检查是否已经存在，如果存在的话，重复的元素会在 `MultiMethod` 实例中合并。

`MultiMethod` 实例通过构建从类型签名到函数的映射来收集方法。 在这个构建过程中，函数注解被用来收集这些签名然后构建这个映射。 这个过程在 `MultiMethod.register()` 方法中实现。 这种映射的一个关键特点是对于多个方法，所有参数类型都必须要指定，否则就会报错。

为了让 `MultiMethod` 实例模拟一个调用，它的 `__call__()` 方法被实现了。 这个方法从所有排除 `slef` 的参数中构建一个类型元组，在内部map中查找这个方法， 然后调用相应的方法。为了能让 `MultiMethod` 实例在类定义时正确操作， `__get__()` 是必须得实现的。 它被用来构建正确的绑定方法。比如：

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

不过本节的实现还有一些限制，其中一个是它不能使用关键字参数。例如：

```
>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>
```

也许有其他的方法能添加这种支持，但是它需要一个完全不同的方法映射方式。 问题在于关键字参数的出现是没有顺序的。当它跟位置参数混合使用时， 那你的参数就会变得比较混乱了，这时候你不得不在 `__call__()` 方法中先去做个排序。

同样对于继承也是有限制的，例如，类似下面这种代码就不能正常工作：

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)
```

```
    def foo(self, x:C):
        print('Foo 2:', x)
```

原因是因为 `xA` 注解不能成功匹配子类实例（比如B的实例），如下：

```
>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 44, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class '__main__.B'>,)
>>>
```

作为使用元类和注解的一种替代方案，可以通过描述器来实现类似的效果。例如：

```
import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self
```

为了使用描述器版本，你需要像下面这样写：

```
class Spam:
    @multimethod
```

```
def bar(self, *args):
    # Default method called if no match
    raise TypeError('No matching method for bar')

@bar.match(int, int)
def bar(self, x, y):
    print('Bar 1:', x, y)

@bar.match(str, int)
def bar(self, s, n = 0):
    print('Bar 2:', s, n)
```

描述器方案同样也有前面提到的限制（不支持关键字参数和继承）。

所有事物都是平等的，有好有坏，也许最好的办法就是在普通代码中避免使用方法重载。 不过有些特殊情况下还是有意义的，比如基于模式匹配的方法重载程序中。 举个例子，8.21小节中的访问者模式可以修改为一个使用方法重载的类。 但是，除了这个以外，通常不应该使用方法重载（就简单的使用不同名称的方法就行了）。

在Python社区对于实现方法重载的讨论已经由来已久。 对于引发这个争论的原因，可以参考下Guido van Rossum的这篇博客： Five-Minute Multimethods in Python