8.13 实现数据模型的类型约束¶

问题¶

你想定义某些在属性赋值上面有限制的数据结构。

解决方案¶

在这个问题中，你需要在对某些实例属性赋值时进行检查。 所以你要自定义属性赋值函数，这种情况下最好使用描述器。

下面的代码使用描述器实现了一个系统类型和赋值验证框架：

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value


# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)


# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)


class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super().__set__(instance, value)
```

这些类就是你要创建的数据模型或类型系统的基础构建模块。 下面就是我们实际定义的各种不同的数据类型：

```
class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass
```

```
class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass
```

然后使用这些自定义数据类型，我们定义一个类：

```
class Stock:
    # Specify constraints
    name = SizedString('name', size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

然后测试这个类的属性赋值约束，可发现对某些属性的赋值违法了约束是不合法的：

```
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 23, in __set__
        raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 16, in __set__
        raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "example.py", line 17, in __set__
        super().__set__(instance, value)
    File "example.py", line 35, in __set__
        raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>
```

还有一些技术可以简化上面的代码，其中一种是使用类装饰器：

```
# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls

    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
```

```
        self.name = name
        self.shares = shares
        self.price = price
```

另外一种方式是使用元类：

```
# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

# Example
class Stock2(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()

    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

讨论¶

本节使用了很多高级技术，包括描述器、混入类、`super()` 的使用、类装饰器和元类。 不可能在这里一一详细展开来讲，但是可以在8.9、8.18、9.19小节找到更多例子。 但是，我在这里还是要提一下几个需要注意的点。

首先，在 `Descriptor` 基类中你会看到有个 `__set__()` 方法，却没有相应的 `__get__()` 方法。 如果一个描述仅仅是从底层实例字典中获取某个属性值的话，那么没必要去定义 `__get__()` 方法。

所有描述器类都是基于混入类来实现的。比如 `Unsigned` 和 `MaxSized` 要跟其他继承自 `Typed` 类混入。 这里利用多继承来实现相应的功能。

混入类的一个比较难理解的地方是，调用 `super()` 函数时，你并不知道究竟要调用哪个具体类。 你需要跟其他类结合后才能正确的使用，也就是必须合作才能产生效果。

使用类装饰器和元类通常可以简化代码。上面两个例子中你会发现你只需要输入一次属性名即可了。

```
# Normal
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()
```

所有方法中，类装饰器方案应该是最灵活和最高明的。 首先，它并不依赖任何其他新的技术，比如元类。其次，装饰器可以很容易的添加或删除。

最后，装饰器还能作为混入类的替代技术来实现同样的效果;

```
# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)
    super_set = cls.__set__

    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
```

```
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__

    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__

    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)

    cls.__init__ = __init__

    super_set = cls.__set__

    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super_set(self, instance, value)

    cls.__set__ = __set__
    return cls


# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass


@Unsigned
class UnsignedInteger(Integer):
    pass


@Typed(float)
class Float(Descriptor):
    pass


@Unsigned
class UnsignedFloat(Float):
    pass


@Typed(str)
class String(Descriptor):
    pass


@MaxSized
class SizedString(String):
    pass
```

这种方式定义的类跟之前的效果一样，而且执行速度会更快。 设置一个简单的类型属性的值，装饰器方式要比之前的混入类的方式几乎快100%。 现在你应该庆幸自己读完了本节全部内容了吧？^_^