9.7 利用装饰器强制函数上的类型检查¶

问题¶

作为某种编程规约，你想在对函数参数进行强制类型检查。

解决方案¶

在演示实际代码前，先说明我们的目标：能对函数参数类型进行断言，类似下面这样：

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>
```

下面是使用装饰器技术来实现 `@typeassert`：

```
from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_types[name])
                            )
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

可以看出这个装饰器非常灵活，既可以指定所有参数类型，也可以只指定部分。 并且可以通过位置或关键字来指定参数类型。下面是使用示例：

```
>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>
```

讨论¶

这节是高级装饰器示例，引入了很多重要的概念。

首先，装饰器只会在函数定义时被调用一次。 有时候你去掉装饰器的功能，那么你只需要简单的返回被装饰函数即可。 下面的代码中，如果全局变量 `__debug__` 被设置成了False(当你使用-O或-OO参数的优化模式执行程序时)， 那么就直接返回未修改过的函数本身：

```
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
```

其次，这里还对被包装函数的参数签名进行了检查，我们使用了 `inspect.signature()` 函数。 简单来讲，它运行你提取一个可调用对象的参数签名信息。例如：

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

装饰器的开始部分，我们使用了 `bind_partial()` 方法来执行从指定类型到名称的部分绑定。 下面是例子演示：

```
>>> bound_types = sig.bind_partial(int,z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

在这个部分绑定中，你可以注意到缺失的参数被忽略了(比如并没有对y进行绑定)。 不过最重要的是创建了一个有序字典 `bound_types.arguments` 。 这个字典会将参数名以函数签名中相同顺序映射到指定的类型值上面去。 在我们的装饰器例子中，这个映射包含了我们要强制指定的类型断言。

在装饰器创建的实际包装函数中使用到了 `sig.bind()` 方法。 `bind()` 跟 `bind_partial()` 类似，但是它不允许忽略任何参数。因此有了下面的结果：

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

使用这个映射我们可以很轻松的实现我们的强制类型检查：

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
```

```
...
>>>
```

不过这个方案还有点小瑕疵，它对于有默认值的参数并不适用。 比如下面的代码可以正常工作，尽管items的类型是错误的：

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

最后一点是关于适用装饰器参数和函数注解之间的争论。 例如，为什么不像下面这样写一个装饰器来查找函数中的注解呢？

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

一个可能的原因是如果使用了函数参数注解，那么就被限制了。 如果注解被用来做类型检查就不能做其他事情了。而且 `@typeassert` 不能再用于使用注解做其他事情的函数了。 而使用上面的装饰器参数灵活性大多了，也更加通用。

可以在PEP 362以及 `inspect` 模块中找到更多关于函数参数对象的信息。在9.16小节还有另外一个例子。