9.17 在类上强制使用编程规约¶

问题¶

你的程序包含一个很大的类继承体系，你希望强制执行某些编程规约（或者代码诊断）来帮助程序员保持清醒。

解决方案¶

如果你想监控类的定义，通常可以通过定义一个元类。一个基本元类通常是继承自 `type` 并重定义它的 `__new__()` 方法或者是 `__init__()` 方法。比如：

```
class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)
```

另一种是，定义 `__init__()` 方法：

```
class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
```

为了使用这个元类，你通常要将它放到到一个顶级父类定义中，然后其他的类继承这个顶级父类。例如：

```
class Root(metaclass=MyMeta):
    pass

class A(Root):
    pass

class B(Root):
    pass
```

元类的一个关键特点是它允许你在定义的时候检查类的内容。在重新定义 `__init__()` 方法中， 你可以很轻松的检查类字典、父类等等。并且，一旦某个元类被指定给了某个类，那么就会被继承到所有子类中去。 因此，一个框架的构建者就能在大型的继承体系中通过给一个顶级父类指定一个元类去捕获所有下面子类的定义。

作为一个具体的应用例子，下面定义了一个元类，它会拒绝任何有混合大小写名字作为方法的类定义（可能是想气死Java程序员^_^）：

```
class NoMixedCaseMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        for name in clsdict:
            if name.lower() != name:
                raise TypeError('Bad attribute name: ' + name)
        return super().__new__(cls, clsname, bases, clsdict)

class Root(metaclass=NoMixedCaseMeta):
    pass

class A(Root):
    def foo_bar(self): # Ok
        pass

class B(Root):
    def fooBar(self): # TypeError
        pass
```

作为更高级和实用的例子，下面有一个元类，它用来检测重载方法，确保它的调用参数跟父类中原始方法有着相同的

参数签名。

```
from inspect import signature
import logging

class MatchSignaturesMeta(type):

    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        sup = super(self, self)
        for name, value in clsdict.items():
            if name.startswith('_') or not callable(value):
                continue
            # Get the previous definition (if any) and compare the signatures
            prev_dfn = getattr(sup,name,None)
            if prev_dfn:
                prev_sig = signature(prev_dfn)
                val_sig = signature(value)
                if prev_sig != val_sig:
                    logging.warning('Signature mismatch in %s. %s != %s',
                                    value.__qualname__, prev_sig, val_sig)

# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self,x,z):
        pass
```

如果你运行这段代码，就会得到下面这样的输出结果：

```
WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)
```

这种警告信息对于捕获一些微妙的程序bug是很有用的。例如，如果某个代码依赖于传递给方法的关键字参数， 那么当子类改变参数名字的时候就会调用出错。

讨论¶

在大型面向对象的程序中，通常将类的定义放在元类中控制是很有用的。 元类可以监控类的定义，警告编程人员某些没有注意到的可能出现的问题。

有人可能会说，像这样的错误可以通过程序分析工具或IDE去做会更好些。诚然，这些工具是很有用。 但是，如果你在构建一个框架或函数库供其他人使用，那么你没办法去控制使用者要使用什么工具。 因此，对于这种类型的程序，如果可以在元类中做检测或许可以带来更好的用户体验。

在元类中选择重新定义 `__new__()` 方法还是 `__init__()` 方法取决于你想怎样使用结果类。 `__new__()` 方法在类创建之前被调用，通常用于通过某种方式（比如通过改变类字典的内容）修改类的定义。 而 `__init__()` 方法是在类被创建之后被调用，当你需要完整构建类对象的时候会很有用。 在最后一个例子中，这是必要的，因为它使用了 `super()` 函数来搜索之前的定义。 它只能在类的实例被创建之后，并且相应的方法解析顺序也已经被设置好了。

最后一个例子还演示了Python的函数签名对象的使用。 实际上，元类将每个可调用定义放在一个类中，搜索前一个定义（如果有的话）， 然后通过使用 `inspect.signature()` 来简单的比较它们的调用签名。

最后一点，代码中有一行使用了 `super(self, self)` 并不是排版错误。 当使用元类的时候，我们要时刻记住一点就是 `self` 实际上是一个类对象。 因此，这条语句其实就是用来寻找位于继承体系中构建 `self` 父类的定义。