Python_Metaclass

一 你可以从这里获取什么?

1. 也许你在阅读别人的代码的时候碰到过 metaclass，那你可以参考这里的介绍。

2. 或许你需要设计一些底层的库，也许 metaclass 能帮你简化你的设计（也有可能复杂化：）

3. 也许你在了解 metaclass 的相关知识之后，你对 python 的类的一些机制会更了解。

4. more.....

二 metaclass 的作用是什么？（感性认识）

metaclass 能有什么用处，先来个感性的认识:

1. 你可以自由的、动态的修改/增加/删除 类的或者实例中的方法或者属性

2. 批量的对某些方法使用 decorator，而不需要每次都在方法的上面加入@decorator_func

3. 当引入第三方库的时候，如果该库某些类需要 patch 的时候可以用 metaclass

4. 可以用于序列化(参见 yaml 这个库的实现，我没怎么仔细看）

5. 提供接口注册，接口格式检查等

6. 自动委托(auto delegate)

7. more...

三 metaclass 的相关知识

1. what is metaclass?

1.1 在 wiki 上面，metaclass 是这样定义的：In object-oriented programming,

a metaclass is a class whose instances are classes.

Just as an ordinary class defines the behavior of certain objects,

a metaclass defines the behavior of certain classes and their instances.

也就是说 metaclass 的实例化结果是类，而 class 实例化的结果是 instance。我是这么理解的:

metaclass 是类似创建类的模板，所有的类都是通过他来 create 的(调用__new__)，这使得你可以自由的控制

创建类的那个过程，实现你所需要的功能。

1.2 metaclass 基础

* 一般情况下, 如果你要用类来实现 metaclass 的话，该类需要继承于 type，而且通常会重写 type 的 __new__方法来控制创建过程。

当然你也可以用函数的方式（下文会讲）

* 在 metaclass 里面定义的方法会成为类的方法，可以直接通过类名来调用

2. 如何使用 metaclass

2.1 用类的形式

2.1.1 类继承于 type, 例如： class Meta(type):pass

2.1.2 将需要使用 metaclass 来构建 class 的类的__metaclass__属性（不需要显示声明，直接有的了）赋值为 Meta（继承于 type 的类）

2.2 用函数的形式

2.2.1 构建一个函数，例如叫 metaclass_new, 需要 3 个参数：name, bases, attrs，

name: 类的名字

bases: 基类，通常是 tuple 类型

attrs: dict 类型，就是类的属性或者函数

2.2.2 将需要使用 metaclass 来构建 class 的类的__metaclass__属性（不需要显示声明，直接有的了）赋值为函数 metaclas_new

3 metaclass 原理

3.1 basic

metaclass 的原理其实是这样的：当定义好类之后，创建类的时候其实是调用了 type 的__new__方法为这个类分配内存空间，创建

好了之后再调用 type 的__init__方法初始化（做一些赋值等）。所以 metaclass 的所有 magic 其实就在于这个__new__方法里面了。

说说这个方法：__new__(cls, name, bases, attrs)

cls: 将要创建的类，类似与 self，但是 self 指向的是 instance，而这里 cls 指向的是 class

name: 类的名字，也就是我们通常用类名.__name__获取的。

bases: 基类

attrs: 属性的 dict。dict 的内容可以是变量(类属性），也可以是函数（类方法）。

所以在创建类的过程，我们可以在这个函数里面修改 name，bases，attrs 的值来自由的达到我们的功能。这里常用的配合方法是

getattr 和 setattr（just an advice)

3.2 查找顺序

再说说关于__metaclass__这个属性。这个属性的说明是这样的：

This variable can be any callable accepting arguments for name, bases, and dict. Upon class creation, the callable is used instead of the built-in type(). New in version 2.2.(所以有了上面介绍的分别用类或者函数的方法）

The appropriate metaclass is determined by the following precedence rules:

If dict['__metaclass__'] exists, it is used.

Otherwise, if there is at least one base class, its metaclass is used (this looks for a __class__ attribute first and if not found, uses its type).

Otherwise, if a global variable named __metaclass__ exists, it is used.

Otherwise, the old-style, classic metaclass (types.ClassType) is used.

这个查找顺序也比较好懂，而且利用这个顺序的话，如果是 old-style 类的话，可以在某个需要的模块里面指定全局变量

__metaclass__ = type 就能把所有的 old-style 变成 new-style 的类了。(这是其中一种 trick)

四 例子

针对第二点说的 metaclass 的作用，顺序来给些例子：

1. 你可以自由的、动态的修改/增加/删除 类的或者实例中的方法或者属性

```
#!/usr/bin/python
def ma(cls):
    print 'method a'

def mb(cls):
    print 'method b'

method_dict = {
    'ma': ma,
    'mb': mb,
}

class DynamicMethod(type):
    def __new__(cls, name, bases, dct):
        if name[:3] == 'Abc':
            dct.update(method_dict)
        return type.__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
```

```
        super(DynamicMethod, cls).__init__(name, bases, dct)



class AbcTest(object):
    __metaclass__ = DynamicMethod
    def mc(self, x):
        print x * 3

class NotAbc(object):
    __metaclass__ = DynamicMethod
    def md(self, x):
        print x * 3

def main():
    a = AbcTest()
    a.mc(3)
    a.ma()
    print dir(a)

    b = NotAbc()
    print dir(b)

if __name__ == '__main__':
    main()
```

通过 DynamicMethod 这个 metaclass 的原型，我们可以在那些指定了__metaclass__属性位 DynamicMethod 的类里面，

根据类名字，如果是以'Abc'开头的就给它加上 ma 和 mb 的方法(这里的条件只是一种简单的例子假设了，实际应用上

可能更复杂）,如果不是'Abc'开头的类就不加. 这样就可以打到动态添加方法的效果了。其实，你也可以将需要动态

添加或者修改的方法改到__new__里面，因为 python 是支持在方法里面再定义方法的. 通过这个例子，其实可以看到

只要我们能操作__new__方法里面的其中一个参数 attrs，就可以动态添加东西了。

2. 批量的对某些方法使用 decorator，而不需要每次都在方法的上面加入@decorator_func

这个其实有应用场景的，就是比如我们 cgi 程序里面，很多需要验证登录或者是否有权限的，只有验证通过之后才

可以操作。那么如果你有很多个操作都需要这样做，我们一般情况下可以手动在每个方法的前头用 @login_required

类似这样的方式。那如果学习了 metaclass 的使用的话，这次你也可以这样做:

```
#!/usr/bin/python
from types import FunctionType

def login_required(func):
    print 'login check logic here'
    return func



class LoginDecorator(type):
    def __new__(cls, name, bases, dct):
        for name, value in dct.iteritems():
            if name not in ('__metaclass__', '__init__', '__module__') and\
                type(value) == FunctionType:
                value = login_required(value)

            dct[name] = value
        return type.__new__(cls, name, bases, dct)



class Operation(object):
    __metaclass__ = LoginDecorator

    def delete(self, x):
        print 'deleted %s' % str(x)



def main():
    op = Operation()
```

```
    op.delete('test')


if __name__ == '__main__':
    main()
```

这样子你就可以不用在 delete 函数上面写@login_required, 也能达到 decorator 的效果了。不过可读性就差点了。

3. 当引入第三方库的时候，如果该库某些类需要 patch 的时候可以用 metaclass

```
def monkey_patch(name, bases, dct):
    assert len(bases) == 1
    base = bases[0]
    for name, value in dct.iteritems():
        if name not in ('__module__', '__metaclass__'):
            setattr(base, name, value)
    return base

class A(object):
    def a(self):
        print 'i am A object'



class PatchA(A):
    __metaclass__ = monkey_patch

    def patcha_method(self):
        print 'this is a method patched for class A'

def main():
    pa = PatchA()
    pa.patcha_method()
    pa.a()
    print dir(pa)
```

```
    print dir(PatchA)


if __name__ == '__main__':
    main()
```

5. 提供接口注册，接口格式检查等, 这个功能可以参考这篇文章：

http://mikeconley.ca/blog/2010/05/04/python-metaclasses-in-review-board/

6. 自动委托(auto delegate)

以下是网上的例子：

http://marlonyao.iteye.com/blog/762156

五 总结

1. metaclass 的使用原则：

If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why). --Tim Peters

也就是说如果你不知道能用 metaclass 来干什么的话，你尽量不要用，因为通常 metaclass 的代码会增加代码的复杂度，

降低代码的可读性。所以你必需权衡 metaclass 的利弊。

2. metaclass 的优势在于它的动态性和可描述性（比如上面例子中的 self.delegate.__getitem__(i)这样的代码，它

可以用另外的函数代码生成,而无需每次手动编写）, 它能把类的动态性扩展到极致。

六 补充

以下是同事们的很好的补充：

张同学：

1.metaclass 属于元编程(metaprogramming)的范畴，所谓元编程就是让程序来写(generate/modify)程序，这通常依赖于语言及其运行时系统的动态特性(其实像 C 这样的语言也可以进行元编程)。正如楼主所说，元编程的一个用途就是“可以用另外的函数代码生成,而无需每次手动编写“，在 python 中我们可以做得更多。

2.对于 python 而言，metaclass 使程序员可以干涉 class 的创建过程，并可以在任何时候修改这样的 class(包括修改 metaclass)，由于 class 的意义是为 instance 集合持有“方法”，所以修改了一个 class 就等于修改了所有这些 instance 的行为，这是

很好的 service。

3.注意 metaclass 的__new__和__init__的区别。

```
class DynamicMethod(type):
    def __new__(cls, name, bases, dct):  # cls=DynamicMethod
    def __init__(cls, name, bases, dct): # cls=你创建的 class 对象
```

这意味着在__new__中我们通常只是修改 dct，但是在__init__中，我们可以直接修改创建好的类，所以我认为这两个接口的主要区别有 2 点：1)调用时机不同(用处请发散思维)；2)__init__比__new__更有用，我在实际项目中一般都是用__init__的。

4.在 python 中我们为什么要修改 class？那当然是为了改变它的行为，或者为了创建出独一无二的类。实际中常常需要为 class 动态添加方法。比如一个数据库表 A 有字段 name, address 等，表 B 有 name, phone 等，你希望 A 的模型类有 find_by_address、find_by_name_and_address 等方法，希望 B 的模型类有 find_by_name、find_by_phone 等方法，但是又不想手写这些方法(其实不可能手写，因为这种组合太多了)，这时你可以在 A、B 共同的 metaclass 中定义一个自动添加方法的子程序，当然你也可以重写__getattr__之类的接口来 hook 所有 find_by_XXX 函数调用，这就是时间换空间了，想象你要执行 find_by_XXX 一百万次。也可以比较一下在 c++/java 中如何应对这种需求。

5.python 的成功之处在于它无处不在的 namespace(就是那个__dict__，其意义可以参考 SICP 第一章的 environment 模型，对计算理论感兴趣的也可以用 lambda 演算来解释)，而且函数又是 first class 对象，又强化了 interface 的使用。我们知道了 metaclass->class->instance 的关系，又知道了对象的方法是放在类里的(请详细考察 python 查找一个方法的流程)，那么用 python 实现各种设计模式就比较简单了。

6.metaclass 不会使程序变晦涩，前提是了解了 metaclass 的固有存在，许多教程的问题就在于它没有告诉读者 metaclass 的存在，而是借助某些其他语言(比如 c++)的类模型来讲解 python。在 ruby 的类型系统中 metaclass 是无限的，metaclass 也有自己的 metaclass（你可以称之为 metametaclass、metametametaclass 等等），ruby 善于实现 DSL 和语法分析器也部分得益于此。

岳同学：

不能说__init__比__new__更有用吧。我觉得要看场合。毕竟__new__能做到比

__init__更多的事情。比如有时候想改生成的类型名字，或者改类型的父类。:)

不过的确大多数场合用__init__就够用了。＋1

在__init__中控制类生成的过程有一点要注意：在__init__()的最后一个参数(attrs)中，对于类中定义的函数类型的属性，比如：

```
def abc(self):
    pass
```

仍然具有以下的 key->value 形式：

"abc":<function object>

但是在生成的类中，"abc"对应的属性已经从一个 function 变成了一个 unbind method：

self.abc --> unbind method

不过实际使用中影响不大。