## 元类模型

元类是用来创建一个类的类。常规的类默认是type类的实例。元类通常是type类的子类，它重新定义了类创建协议的方法，以便定制在一条class语句的末尾发布的类创建的调用；它通常会重定义\_\_new\_\_和\_\_init\_\_方法以便接入类创建协议。

### 类是类型的实例

* 在python3.0中，用户定义的类对象是名为type的对象的实例，type本身是一个类。
* 在python2.6中，新式类继承自object，它是type的一个子类；传统类是type的一个实例，并且并不创建自一个类。

函数type实际上是一个元类。type就是Python在背后用来创建所有类的元类。现在你想知道那为什么type会全部采用小写形式而不是Type呢？好吧，我猜这是为了和str保持一致性，str是用来创建字符串对象的类，而int是用来创建整数对象的类。type就是创建类对象的类。你可以通过检查\_\_class\_\_属性来看到这一点。Python中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来。

python3.0

>>> type([])

<class 'list'>

>>> type(type([]))

<class 'type'>

实例创建自类，类创建自type。在python3.0中**类是类型，类型是类**。

* 类型由派生自type的类定义。
* 用户定义的类是类型类的实例。
* 用户定义的类是产生它们的自己的实例的类型。

在python3.0中的类是type类的实例，并且实例对象是它们的类的实例。

>>> class C: pass

>>> type(C)

<class 'type'>

>>> C.\_\_class\_\_

<class 'type'>

>>> X = C()

>>> type(X)

<class '\_\_main\_\_.C'>

>>> X.\_\_class\_\_

<class '\_\_main\_\_.C'>

在python2.6中，对于派生自object的新式类，情况也是如此。

>>> class C(object): pass

>>> type(C)

<type 'type'>

>>> X = C()

>>> type(X)

<class '\_\_main\_\_.C'>

>>> X.\_\_class\_\_

<class '\_\_main\_\_.C'>

>>> C.\_\_class\_\_

<type 'type'>

在python2.6老式类中：

>>> class C: pass

>>> type(C)

<type 'classobj'>

>>> C.\_\_class\_\_

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

C.\_\_class\_\_

AttributeError: class C has no attribute '\_\_class\_\_'

>>> X = C()

>>> type(X)

<type 'instance'>

>>> X.\_\_class\_\_

<class \_\_main\_\_.C at 0x0000000002DDB2E8>

### 元类是type的子类

由于类实际上是type的实例，从type的定制的子类创建类允许我们实现各种定制的类。

在python3.0以及python2.6的新式类中：

* type是用户定义的类的一个类
* 元类是type类的一个子类
* 类对象是type类的一个实例，或是一个子类。
* 实力对象产生自一个类。

换句话说，为了控制创建类以及扩展其行为的方式，我们所需要做的只是指定一个用户

定义的类创建自一个用户定义的元类，而不是常规的type类。

注意，这个类型实例关系与继承并不完全相同：用户定义的类可能也拥有超类，它们及

其实例从那里继承属性（继承超类在c l a s s语句的圆括号中列出，并且出现在一个类的

\_\_bases\_\_元组中）。类创建自的类型，以及它是谁的实例，这是不同的关系。

### class语句协议

调用type对象来创建class对象：

class = type(classname, superclasses, attributedict)

type(类名, 父类的元组（针对继承的情况，可以为空），包含属性的字典（名称和值）)

type对象反过来定义了一个\_\_call\_\_运算符重载方法，当调用type对象时，该方法运行两个其他的方法：

type.\_\_new\_\_(typeclass, classname, superclasses, attributedict)

type.\_\_init\_\_(class, classname, superclasses, attributedict)

\_\_new\_\_方法创建并返回一个新的class对象，并且随后的\_\_init\_\_方法初始化了新创建的对象。

class Spam(Eggs): # Inherits from Eggs

data = 1 # Class data attribute

def meth(self, arg): # Class method attribute

pass

python将从内部运行嵌套的代码块来创建该类的两个属性（data和meth），然后在class语句的末尾调用type对象，产生class对象：

Spam = type(‘spam’, (Eggs,), {‘data’: 1, ‘meth’: meth, ‘\_\_moudle\_\_’: ‘\_\_main\_\_’})

类默认是type类创建的

## 声明元类

在python3.0中，在类标题中把想要的元类作为一个关键字参数列出来：

class Spam(metaclass=Meta): # 3.0 and later

继承超类也可以列在标题中，在元类之前。

class Spam(Eggs, metaclass=Meta):

在python2.6中得到同样的效果。

class Spam(object):

\_\_metaclass\_\_ = Meta

当以这些方式声明的时候，创建类对象的调用在class语句的底部运行，修改为调用元类而不是默认的的type：

class = Meta(classname, superclasses, attributedict)

由于元类是type的一个子类，所以type类的\_\_call\_\_把创建和初始化新的类对象的调用

委托给元类，如果它定义了这些方法的定制版本：

Meta.\_\_new\_\_(Meta, classname, superclasses, attributedict)

Meta.\_\_init\_\_(class, classname, superclasses, attributedict)

## 编写元类

### 基本元类

class MetaOne(type):

def \_\_new\_\_(meta, classname, supers, classdict):

print('In MetaOne.new:', classname, supers, classdict, sep='\n...')

return type.\_\_new\_\_(meta, classname, supers, classdict)

class Eggs:

pass

print('make class')

class Spam(Eggs, metaclass=MetaOne):

data = 1

def meth(self):

pass

print('making instance')

X = Spam()

print('data:', X.data)

执行结果：

make class

In MetaOne.new:

...Spam

...(<class '\_\_main\_\_.Eggs'>,)

...{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Spam', 'data': 1, 'meth': <functi

on Spam.meth at 0x000000000290E1E0>}

making instance

data: 1

### 定制构建和初始化

class MetaOne(type):

def \_\_new\_\_(meta, classname, supers, classdict):

print('In MetaOne.new:', classname, supers, classdict, sep='\n...')

return type.\_\_new\_\_(meta, classname, supers, classdict)

def \_\_init\_\_(Class, classname, supers, classdict):

print('In MetaOne init:', classname, supers, classdict, sep='\n...')

print('...init class object:', list(Class.\_\_dict\_\_.keys()))

class Eggs:

pass

print('make class')

class Spam(Eggs, metaclass=MetaOne):

data = 1

def meth(self):

pass

print('making instance')

X = Spam()

print('data:', X.data)

执行结果：

make class

In MetaOne.new:

...Spam

...(<class '\_\_main\_\_.Eggs'>,)

...{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Spam', 'data': 1, 'meth': <functi

on Spam.meth at 0x00000000021FE268>}

In MetaOne init:

...Spam

...(<class '\_\_main\_\_.Eggs'>,)

...{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Spam', 'data': 1, 'meth': <functi

on Spam.meth at 0x00000000021FE268>}

...init class object: ['\_\_module\_\_', 'data', 'meth', '\_\_doc\_\_']

making instance

data: 1

### 其他元类编程技巧

#### 使用简单的工厂函数

def MetaFunc(classname, supers, classdict):

print('In MetaFunc:', classname, supers, classdict, sep='\n...')

return type(classname, supers, classdict)

class Eggs:

pass

print('make class')

class Spam(Eggs, metaclass=MetaFunc):

data = 1

def meth(self):

pass

print('making instance')

X = Spam()

print('data:', X.data)

### 实例与继承的关系

* 元类继承自type类。
* 元类声明由子类继承。
* 元类属性没有由类实例继承。元类声明指定了一个实例关系，**它和继承不同**。由于类是元类的实例，所以**元类中定义的行为应用于与类，而不是类随后的实例**。

class MetaOne(type):

def \_\_new\_\_(meta, classname, supers, classdict):

print('In MetaOne.new:', classname, supers, classdict, sep='\n...')

return type.\_\_new\_\_(meta, classname, supers, classdict)

def toast(self):

print('toast')

class Super(metaclass=MetaOne):

def spam(self):

print('spam')

class C(Super):

def eggs(self):

print('eggs')

X = C()

X.eggs() # Inherited from C

X.spam() # Inherited from Super

X.toast() # Not inherited from metaclass

\_\_new\_\_方法负责创建一个实例对象，在对象被创建的时候调用该方法它是一个类方法。\_\_new\_\_方法在返回一个实例之后，会自动的调用\_\_init\_\_方法，对实例进行初始化。如果\_\_new\_\_方法不返回值，或者返回的不是实例，那么它就不会自动的去调用\_\_init\_\_方法。

\_\_init\_\_ **方法负责将该实例对象进行初始化**，在对象被创建之后调用该方法，在\_\_new\_\_方法创建出一个实例后对实例属性进行初始化。\_\_init\_\_方法可以没有返回值。

\_\_call\_\_方法其实和类的创建过程和实例化没有多大关系了，定义了\_\_call\_\_方法才能被使用函数的方式执行。

* \_\_new\_\_就是一个房屋的框架，每个具体的房屋都需要先搭好框架后才能进行专修，当然现有了房屋设计才能有具体的房屋框架出来。这个就是从类到类实例的创建。
* \_\_init\_\_就是装修房子的过程，对房屋的墙面和地板等颜色材质的丰富就是它该做的事情，当然先有具体的房子框架出来才能进行装饰了。这个就是**实例属性的初始化，**它是在\_\_new\_\_出一个实例后才能初始化。
* \_\_call\_\_就是房子的电话，有了固定电话，才能被打电话嘛（就是通过括号的方式像函数一样执行）。