函数和类装饰器允许我们拦截并扩展函数调用和类实例创建调用。

元类允许我们拦截并扩展类创建。

实例是类创建的，而类本身也是实例，Python3.0中，类是type的实例，type本身也是一个类。

元类本身是type的子类。

元类的作用参考：程序proxy\_test中的crawler.py和getter.py，在Crawler中只负责写以crawl\_开头的函数，在元类ProxyMetaclass中维护一个函数对象列表。

class语句协议：

在一条class语句的末尾，调用type对象来创建class对象：

class = type(classname, superclasses, attributedict)

调用type对象时，运行下面两个方法：

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

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

\_\_new\_\_方法创建并返回新的class对象，\_\_init\_\_方法初始化了新创建的对象。

例：

class Spam(Eggs):

data = 1

def meth(self, arg):

pass

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

例：class\_detail/class\_detail17.py

def func(self, name='world'):  
 print('Hello, {0}'.format(name))  
  
  
Hello = type('Hello', (object,), dict(hello=func))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 h = Hello()  
 h.hello() # Hello, world  
 print(type(h)) # <class '\_\_main\_\_.Hello'>  
 print(type(Hello)) # <class 'type'>

由上可以看出，类默认是由type创建的，如果希望定制类的创建，就需要声明元类。

class Spam(Eggs, metaclass=Meta):

data = 1

def meth(self, arg):

pass

Spam = Meta(‘Spam’, (Eggs,), {‘data’: 1, ‘meth’: meth, ‘\_\_module\_\_’: ‘\_\_main\_\_’})

元类Meta如果定义了\_\_new\_\_或\_\_init\_\_的自己版本，将依次调用。

例：class\_detail/class\_detail18.py

class MetaClassOne(type):  
 def \_\_new\_\_(mcs, classname, superclasses, attrs):  
 print('In MetaClassOne.new: ', classname, superclasses, attrs, sep='\n...')  
 return type.\_\_new\_\_(mcs, classname, superclasses, attrs)  
  
 def \_\_init\_\_(cls, classname, superclasses, attrs):  
 print('In MetaClassOne.init: ', classname, superclasses, attrs, sep='\n...')  
  
  
print('making class')  
  
  
class Spam(metaclass=MetaClassOne):  
 data = 1  
  
 def meth(self, arg):  
 pass  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 print()  
 print('making instance')  
 x = Spam()

输出为：

making class

In MetaClassOne.new:

...Spam

...()

...{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Spam', 'data': 1, 'meth': <function Spam.meth at 0x000001F6D0BFB2F0>}

In MetaClassOne.init:

...Spam

...()

...{'\_\_module\_\_': '\_\_main\_\_', '\_\_qualname\_\_': 'Spam', 'data': 1, 'meth': <function Spam.meth at 0x000001F6D0BFB2F0>}

making instance

元类的使用：

例：程序class\_detail/class\_detail19.py

class SayMetaClass(type):  
 def \_\_new\_\_(mcs, classname, supers, attrs):  
 # 在元类中根据类的名字创建一个方法  
 attrs['say\_' + classname] = lambda self, value, saying=classname: print(saying + ',' + value)  
 return type.\_\_new\_\_(mcs, classname, supers, attrs)  
  
  
class Hello(metaclass=SayMetaClass):  
 pass  
  
  
class Sayolala(metaclass=SayMetaClass):  
 pass  
  
  
class Nihao(metaclass=SayMetaClass):  
 pass  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 hello = Hello()  
 # say\_Hello是通过元类创建的  
 hello.say\_Hello('world!') # Hello,world!  
  
 s = Sayolala()  
 s.say\_Sayolala('japan!') # Sayolala,japan!  
  
 n = Nihao()  
 n.say\_Nihao('中国!') # Nihao,中国!

通过元类实现ORM：

ORM：Object Relational Mapping，即对象-关系映射，把关系数据库表的一行映射为一个对象，也就是一个类对应一个数据库表。

例：程序class\_detail/class\_detail20.py

class Field:  
 def \_\_init\_\_(self, name, column\_type):  
 self.name = name  
 self.column\_type = column\_type  
  
 def \_\_str\_\_(self):  
 return '{0}:{1}'.format(self.name, self.column\_type)  
  
  
class StringField(Field):  
 def \_\_init\_\_(self, name):  
 super().\_\_init\_\_(name, 'varchar(100)')  
  
  
class IntegerField(Field):  
 def \_\_init\_\_(self, name):  
 super().\_\_init\_\_(name, 'bigint')  
  
  
# 元类  
# 排除对Model类的修改(其为所有表的基类)  
# 在Model子类中查找定义的类的所有属性，如果为Field，  
# 保存到dict中，并将字典存储到\_\_mappings\_\_属性，  
# 之后从类属性中删除该Field  
# 将表名存储到\_\_table\_\_属性中  
class ModelMetaClass(type):  
 def \_\_new\_\_(mcs, class\_name, supers, attrs):  
 if class\_name == 'Model':  
 return type.\_\_new\_\_(mcs, class\_name, supers, attrs)  
 print('Found model: {0}'.format(class\_name))  
 mappings = dict()  
 for key, value in attrs.items():  
 if isinstance(value, Field):  
 print('Found mapping: {0} ==> {1}'.format(key, value))  
 mappings[key] = value  
  
 for key in mappings.keys():  
 attrs.pop(key)  
  
 attrs['\_\_mappings\_\_'] = mappings  
 attrs['\_\_table\_\_'] = class\_name  
 return type.\_\_new\_\_(mcs, class\_name, supers, attrs)  
  
  
# Model父类  
class Model(dict, metaclass=ModelMetaClass):  
 def \_\_init\_\_(self, \*\*kw):  
 super().\_\_init\_\_(\*\*kw)  
  
 def \_\_getattr\_\_(self, key):  
 try:  
 return self[key]  
 except KeyError:  
 raise AttributeError(r"Model' object has no attribute {0}".format(key))  
  
 def \_\_setattr\_\_(self, key, value):  
 self[key] = value  
  
 def save(self):  
 fields = []  
 args = []  
 for key, value in self.\_\_mappings\_\_.items():  
 fields.append(value.name)  
 args.append(getattr(self, key, None))  
 sql = 'insert into {0} ({1}) values ({2})'\  
 .format(self.\_\_table\_\_, ','.join(fields), ','.join([str(i) for i in args]))  
 print('SQL:{0}'.format(sql))  
 print('ARGS: {0}'.format(str(args)))  
  
  
# 用户类  
# User会隐式继承Model的元类ModelMetaClass  
class User(Model):  
 id = IntegerField('id')  
 name = StringField('username')  
 password = StringField('password')  
  
  
class Data(Model):  
 id = IntegerField('id')  
 data = StringField('data')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 user = User(id=123, name='Batman', password='abcde')  
 print()  
 user.save()  
  
 data = Data(id=456, data='qwert')  
 print()  
 data.save()

输出为：

Found model: User

Found mapping: id ==> id:bigint

Found mapping: name ==> username:varchar(100)

Found mapping: password ==> password:varchar(100)

Found model: Data

Found mapping: id ==> id:bigint

Found mapping: data ==> data:varchar(100)

SQL:insert into User (id,username,password) values (123,Batman,abcde)

ARGS: [123, 'Batman', 'abcde']

SQL:insert into Data (id,data) values (456,qwert)

ARGS: [456, 'qwert']

从输出可以看出，元类同装饰器类似，在class语句末尾就会执行