python-进阶-元类在ORM上的应用详解

时间之友 (/u/5f1d0ebcb6b1) + 关注
2017.11.20 16:35* 字数 1339 阅读 91 评论 0 喜欢 0
(/u/5f1d0ebcb6b1)

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是**先把调用接口写出来**。比如，使用者如果使用这个ORM框架，想定义一个User类来操作对应的数据库表User，我们期待他写出这样的代码：
期望代码

```
class User(Model):      #  User类，继承Model
    # 定义类的属性到列的映射：  右边的 StringField(‘username’),这里StringField是类，类(‘user
    id = IntegerField('id')     # ⚠id理解为一个变量不好，id理解成一个key最恰当，=右边的则是
    name = StringField('username')    # name是类属性，具体可参考类属性与实例属性定义，类调用
    email = StringField('email')       # id name email password 均是User的属性，User的
    password = StringField('password')  # User().name = StringField(‘username’),直接

# 创建一个实例：
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')     # ⚠
# 保存到数据库：
u.save()
```

其中，父类Model和属性类型StringField、IntegerField是由ORM框架提供的，剩下的魔术方法比如 `save()` 全部由 `metaclass` 自动完成。虽然 `metaclass` 的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义Field类，它**负责保存数据库表的字段名和字段类型**

```
class Field(object):    # 对应数据库中保存的字段名和字段类型

    def __init__(self, name, column_type):     # 添加__init__(self,name)方法后，类的实
        self.name = name    # 字段名
        self.column_type = column_type    # 字段类型

    def __str__(self):   # 输入print(xxx)会自动调用__str__,它为了使打印结果更好看而已，Pyth

        return '<%s:%s>' % (self.__class__.__name__, self.name)    # 不使用print(xxx)
```

在Field的基础上，进一步定义各种类型的Field，比如StringField，IntegerField等等：

```
class StringField(Field):     # ✌通过比对测试，StringField.name = name, StringField.c

    def __init__(self, name):     # 实例属性，`实例.name`调用
        super(StringField, self).__init__(name, 'varchar(100)')   # 为什么这么设计? su
         # ⚠super的用法，继承父类的方法而自己无需定义， super(StringFileld,self).__init__
------------------------测试------------------------------
In [67]: test = StringField('love')

In [68]: type(test)
Out[68]: __main__.StringField

In [70]: test.column_type
Out[70]: 'varchar(100)'

In [71]: test.name
Out[71]: 'love'
-------------------------------------------------------------
class IntegerField(Field):       # ⚠子类继承这些方法

    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')   # __init__()这样的写法,可
```

下一步，就是编写最复杂的ModelMetaclass了：(重点 难)

```
class ModelMetaclass(type):
    # 元类必须实现__new__方法，当一个类指定通过某元类来创建，那么就会调用该元类的__new__方法
    # 该方法接收4个参数
    # cls为当前准备创建的类的对象
    # name为类的名字，创建User类，则name便是User
    # bases类继承的父类集合,创建User类，则base便是Model
    # 🐷attrs廖神说是类的方法的集合，是类方法！类方法是一种函数，怎么集合？所以集合的key应是方法名

    def __new__(cls, name, bases, attrs):    # 新建new，所以后面4个参数都是类相关的
        if name=='Model':
        # 因为Model类是基类，所以排除掉，如果你print(name)的话，会依次打印出Model,User,Blog,
        # 所有的Model子类，因为这些子类通过Model间接继承元类
            return type.__new__(cls, name, bases, attrs)
        print('Found model: %s' % name)   # 打印提示

        mappings = dict()   #  用于存储所有的字段，以及字段值，注意是dict(),而不是[ ]，所以后
        for k, v in attrs.items():     # 注意这里attrs的key是字段名，value是字段实例，不是
            if isinstance(v, Field):       # 筛选 v,判断v是否是Field
            # attrs同时还会拿到一些其它系统提供的类属性，我们只处理自定义的类属性，所以判断一下
            # isinstance 方法用于判断v是否是一个Field,只处理自定义的类属性
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v   # 经测试，语法正确，attrs字典移植到mappings里
        for k in mappings.keys():    # mappings.keys会打印出所有的key
            attrs.pop(k)    # pop删除k,删除mappings含有的所有的字段名，v是字段实例
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系，猜测是mappings已经确定了
            # '__mappings__'是字符串，属于我们想要的属性，属性很多，想要的就这几个，其他全pop，
        attrs['__table__'] = name # 假设表名和类名一致
            # 以上都是要返回的东西了，刚刚记录下的东西，如果不返回给这个类，又谈得上什么动态创建呢
            # 到此，动态创建便比较清晰了，各个子类根据自己的字段名不同，动态创建了自己
            # 上面通过attrs返回的东西，在子类里都能通过实例拿到，如self
        return type.__new__(cls, name, bases, attrs)

#  一次复盘后的猜想：我们先看最上方的User，这个就是我们需要的效果，也正是我们 元类 __new__方法 可
 那么，我们想，__new__(cls, name, bases, attrs)里的attrs属性应该对应的是元类属性，也就是说元类
但是元类的属性继承虽然可以，因为在元类基础上定义的子类属性应该是动态创建的，所以属性的名称可能会不同，

#  廖-把类看成是metaclass创建出来的“实例”
#   类的方法（类中def的就是类方法，实例中def的是函数）就是类实例的属性。
所以，我们定义元类的方法就是给 类增加属性，这样类通过 __new__ 创建后自带属性

#  参第一块代码User，在创建User类时，中name == User, bases == Model, attrs 是即将创建的类 U
   相当于在attrs处传入了  字典{  'id': IntegerField('id'),  'name': 'StringField('usern
   那么for k, v in attrs.items()中，k 为'id'等（赋值号左边），'v'为IntegerField('id')实例
   在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个__mapping
```

以及基类Model：

```
class Model(dict, metaclass=ModelMetaclass):
# 让Model继承dict,主要是为了具备dict所有的功能，如get方法
# metaclass指定了Model类的元类为ModelMetaClass
    def __init__(self, **kw):
        super(Model, self).__init__(**kw)       # 继承父类的init

    def __getattr__(self, key):    # 定义实例属性 目标：获得属性
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)
            # 实现__getattr__与__setattr__方法，可以使引用属性像引用普通字段一样  如self['i
    def __setattr__(self, key, value):
        self[key] = value

    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields),
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

廖神解说：
当用户定义一个class User(Model)时，Python解释器首先在当前类User的定义中查找metaclass，如果没有找到，就继续在父类Model中查找metaclass，找到了，就使用Model中定义的metaclass的ModelMetaclass来创建User类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在ModelMetaclass中，一共做了几件事情：

1. 排除掉对Model类的修改；

2. 在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个 `__mappings__` 的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；

3. 把表名保存到 `__table__` 中，这里简化为表名默认为类名。

在Model类中，就可以定义各种操作数据库的方法，比如 `save()`， `delete()`， `find()`， `update` 等等。

我们实现了 `save()` 方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出INSERT语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org', password='my-pwd')
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>   # 注意⚠️，从结果导向，k 为 email,v 为<St
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,id) values (?,?,?,?)
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
# 从结果可以看出：self.__table是User; .join(fields)是(password,email,username,id); .joi
# 进而推导fields.append(v.name)，那么v.name就是(password,email,username,id)；推导args.ap
# getattr是python中自省功能，即返回展示对象的属性
```

可以看到， `save()` 方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能

12月6日补充

在ModelMetaclass 里为什么会有pop()这样的操作？请注意，我们初识定义的时候， `def __new__(cls, name, bases, attrs)` 中的 attrs 是 类的方法的集合(廖),类的方法，也就是实例的方法，那么方法有很多种，一部分方法是要归纳到 数据库 中，并建立映射 mappings，所以对应的 原来的类的方法的集合 attrs 就该删除这部分已经归纳的，防止出现意外，导致数据库 数据错误。
这两段代码里的v都是在类User定义时的实例化Field对象，注意v不是字符串，虽然在第二段代码里输出了字符串，那是在Field定义时有函数 `__str` ，输出的时这个函数返回的字符串，所有的Field实例都可以这样返回字符串

然后剩下的类方法，再 `return type.__new__(cls, name, bases, attrs)` ,此时的attrs里已经没有了那些属于 Field 类的 value 了。

廖大-----在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个**mappings**的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）

这里廖大又归结为类的属性，其实在使用上来说，类属性与类方法差别不大，除了类属性 类可以直接调用，面对实例时它们用法相同，如下，还真的可以判断是类的属性，所以现在初始的 attrs ，我可以认为它是类属性和类方法的合集

```
class User(Model):
    # 定义类的属性到列的映射：
    id = IntegerField('id')
    name = StringField('username')
```

网友总结：