metaclass

类也是对象

"Python中一切皆对象"，这个描述虽然不准确，但是可以用于理解Python中的类和函数

函数是一等公民

Python函数是一等公民，这一点毋庸置疑，我们可以复制一个函数对象，将函数对象传递给函数，或者是赋值给某一个变量

赋值

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

func = factorial
print(func(3))
```

作为函数参数传递给函数

```
import threading

t = threading.Thread(target=factorial, args=(10, ))
t.start()
```

类也是一等公民

和函数一样，类也是一等公民，可以将类对象赋值给一个变量，传递给某一个函数

赋值

```
class Bar(object):
    pass

dynamic_class = Bar
ins = dynamic_class()
```

也就是说，在Python中，函数和类都是对象

对于一个具体的类实例对象而言，其创建者就是类(class)

```
In [25]: class Bar(object):
    ...:     pass
    ...:

In [26]: ins = Bar()

In [27]: ins.__class__
Out[27]: __main__.Bar
```

那么，函数或者类这一对象又是谁创建的？ 答案是type

使用type来创建类

type()方法调用本身用于获取某个对象的具体类型

```
In [2]: type(10)
Out[2]: int

In [3]: type("Zero")
Out[3]: str
```

但是，同样可以使用type来创建一个类对象

type(name, bases, dict)

- name 所创建类对象的名称，为一字符串
- bases 所创建类对象所继承的类，为一tuple
- dict 所创建类对象的类属性，为一字典，key-value形式

```
In [39]: s = type('Bar', (object, ), {'name': 'zero'})

In [40]: s.name
Out[40]: 'zero'

In [41]: type(s)
Out[41]: type
```

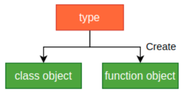
实际上，对于Python的基本数据类型对象，均是由type所创建的

```
In [6]: str.__class__
Out[6]: type

In [7]: int.__class__
Out[7]: type
```

type其实就是Python内建的一个元类(metaclass)，用于创建其它的类对象

换而言之，使用type来创建类对象其实是Python的一个默认行为，用户也可以不选择使用该默认行为，使用自定义的方式去创建类对象，也就是说，在创建类对象时使用自己的元类



使用自定义的metaclass

type的真实面目

首先，type本身是一个类，也就是Python中的class，在调用type()方式时实际调用的是__call__方法

```
class type(object):
    """
    type(object_or_name, bases, dict)
    type(object) -> the object's type
    type(name, bases, dict) -> a new type
    """
    ...
```

如果想要实现自定义的方式来创建对象，那么定义的metaclass必须继承type，来获得与type相同的能力: 创建类对象

__init__方法和__new__方法

在Python中，实际创建对象的过程是由__new__方法控制的，该方法接收class对象(cls)。而__init__方法则是在__new__方法所创建的对象实例上，进行属性的赋值或者其它操作，所以接收实例对象(self)

```
class Bar(object):
    def __new__(cls, *args, **kwargs):
        cls = super(Bar, cls).__new__(cls, *args, **kwargs)
        print("__new__ method")
        return cls

    def __init__(self):
        self.name = "Bar"
        print("__init__ method")
```

这也是为什么__new__方法必须要有返回值，而__init__方法不需要返回值的原因

所以，要想实现自定义创建类对象的行为，首先需要取得创建类对象的能力(继承type类)，其次就是定义实际创建类对象的行为(编写__new__方法)

```
class BarMeta(type):
    def __new__(mcs, *args, **kwargs):
        # 第一个参数写作mcs，而不是cls，是个约定
        cls = super(BarMeta, mcs).__new__(mcs, *args, **kwargs)
        print('Do something else')
        return cls


class Bar(object, metaclass=BarMeta):
    foo = "foo"

    def __init__(self):
        self.name = "Zero"
```

对于类Bar而言，其元类为BarMeta，即BarMeta将会用于创建类对象Bar

而对于BarMeta而言，做的事情也非常简单: 使用Python默认的行为创建出类对象，而后打印一行东西，仅此而已

首先来看一个很有意思的现象

```
class BarMeta(type):
    def __new__(mcs, *args, **kwargs):
        cls = super(BarMeta, mcs).__new__(mcs, *args, **kwargs)
        print(getattr(cls, "foo"))
        return cls


class Bar(object, metaclass=BarMeta):
    foo = "keyerror"

    def __init__(self):
        self.name = "Zero"
```

在BarMeta这一自定义元类的__new__方法中，尝试去获取Bar的类属性: foo，会抛出AttributeError吗？

答案是不会，完全可以获取到"子类"的相关类属性，甚至是类方法

这就是元类的杀手级特性: 预先获取原有类的所有属性和行为

现象

```
class BarMeta(type):
    def __new__(mcs, *args, **kwargs):
        cls = super(BarMeta, mcs).__new__(mcs, *args, **kwargs)

        # 获取类型为MyField全部类属性
        attrs = args[2]
        for k, v in attrs.items():
            if isinstance(v, MyField):
                print("Get field: {}".format(k))

        return cls

class MyField(object):
    def __init__(self, value):
        self.value = value

class Bar(object, metaclass=BarMeta):
    foo = MyField("foo value")
    baz = MyField("baz value")
```

通过这种方式，能够获取到Bar类所有类型为MyField的类属性，获取完毕之后可以做一些加工类的动作

预先获取类所有的行为和属性

元类作为创建类对象的基本对象，在创建类对象时需要获取到关于该类对象的全部信息，包括类属性和类方法。那么到底有什么用？

应用

Serializer

对于一个序列化器而言，用户通常会继承某一个第三方库所提供的类，而后编写将要序列化的字段

```
from marshmallow import Schema, fields


class User(Schema):
    username = fields.Str()
    password = fields.Str()
```

当我们调用load或者dump方法时，总是能够得到想要的数据格式。但是，load()和dump()是父类的方法，我们又没有传递其它参数，序列化器是如何得知有哪些字段的？

```
user = {
    "username": "smart",
    "password": "zero"
}

user_schema, err = User().dump(user)
```

在上面的例子中，username和password都是类属性，而非实例属性，那么通过指定经过精心设计的metaclass，就能够获得User类的全部信息: 有哪些字段，分别是什么类型等等

如此一来，在进行dump()或者是load()时，直接使用在创建User类对象之前就已经获取到的类属性信息，而无需用户传入

ORM

ORM应该是Python metaclass使用最为频繁的地方了，其应用元类的初衷和Serializer是一样的: 在创建真正的类对象之前，将用户定义的字段保存在自身类结构中，无需用户维护