元编程

元编程概念来自LISP和smalltalk。
我们写程序是直接写代码，是否能够用代码来生成未来我们需要的代码吗？这就是元编程

用来生成代码的程序称为元程序metaprogram，编写这种程序就称为元编程metaprogramming
Python语言能够通过反射实现元编程。

type类

```
class type(object):
    def __init__(cls, what, bases=None, dict=None): # known special case of type.__init__
        """
        type(object_or_name, bases, dict)
        type(object) -> the object's type
        type(name, bases, dict) -> a new type
        # (copied from class doc)
        """
        pass
```

`type(object) -> the object's type`，返回对象的类型，例如type(10)

`type(name, bases, dict) -> a new type` 返回一个新的类型

```
XClass = type('myclass', (object,), {'a':100, 'b':'string'})
print(XClass)
print(XClass.__dict__)
print(XClass.__name__)
print(XClass.__bases__)
print(XClass.mro())
```

上例并不稀奇，我们用它创建更加复杂的类

```
def __init__(self):
    self.x = 1000

def show(self):
    print(self.__dict__)

XClass = type('myclass', (object,), {'a':100, 'b': 'string', 'show':show, '__init__':__init__})
print(XClass)
print(XClass.__name__)
print(XClass.__dict__)
print(XClass.mro())

XClass().show()
```

可以借助type构造任何类，用代码来生成代码，这就是元编程。

换一种写法，可以使用继承type的方法

```
class ModelMeta(type):
    def __new__(cls, *args, **kwargs):
        print(cls)
        print(args)
        print(kwargs)
        return super().__new__(cls, *args,**kwargs)
```

继承自type，相当于`ModelMeta = type('ModelMeta', bases, dict)`
可以认为ModelMeta就是元类，它可以创建出类。

```
class ModelMeta(type): # 继承自type
    def __new__(cls, *args, **kwargs):
        print(cls)
        print(args)
        print(kwargs, '-------------')
        return super().__new__(cls, *args, **kwargs)

# 第一种 使用metaclass关键字参数指定元类
class A(metaclass=ModelMeta):
    id = 100

    def __init__(self):
        print('~~A.init~~')

# 第二种 B继承自A后，依然是从ModelMeta的类型
class B(A):
    def __init__(self):
        print('~~B.init~~')

# 第三种 元类就可以使用下面的方式创建新的类
D = ModelMeta('D', (), {})

# C、E是type的实例
class C:pass # C = type('C', (), {})
E = type('E', (), {})
class F(ModelMeta):pass

print('~~~~~~~~~~~~')
print(type(A))
print(type(B))
print(type(D))
print(type(C))
print(type(E))
print(type(F))
```

```
<class '__main__.ModelMeta'>
```

```
('A', (), {'__init__': <function A.__init__ at 0x0000000001148C80>, '__qualname__': 'A',
'__module__': '__main__', 'id': 100})
{} -------------
<class '__main__.ModelMeta'>
('B', (<class '__main__.A'>,), {'__qualname__': 'B', '__module__': '__main__', '__init__':
<function B.__init__ at 0x0000000001148D08>})
{} -------------
<class '__main__.ModelMeta'>
('D', (), {})
{} -------------
~~~~~~~~~~~~
<class '__main__.ModelMeta'>
<class '__main__.ModelMeta'>
<class '__main__.ModelMeta'>
<class 'type'>
<class 'type'>
<class 'type'>
```

从运行结果还可以分析出`__new__(cls, *args, **kwargs)`的参数结构
中间是一个元组`('A', (), {'__init__': <function A.__init__ at 0x0000000000B6E598>, '__module__': '__main__', '__qualname__': 'A', 'id': 100})`
对应`(name, bases, dict)`
修改代码如下

```
class ModelMeta(type): # 继承自type
    def __new__(cls, name, bases, dict):
        print(cls)
        print(name)
        print(bases)
        print(dict, '-------------')
        return super().__new__(cls, name, bases, dict)
```

元类的应用

```
class Field:
    def __init__(self, fieldname=None, pk=False, nullable=True):
        self.fieldname = fieldname
        self.pk = pk
        self.nullable = nullable

    def __repr__(self):
        return "<Field {}>".format(self.fieldname)


class ModelMeta(type): # 继承自type
    def __new__(cls, name, bases, attrs:dict):
        print(cls)
        print(name)
        print(bases)
        print(attrs, '-------------')
```

```
        if '__tablename__' not in attrs.keys():
            attrs['__tablename__'] = name

        primarykeys = []
        for k,v in attrs.items():
            if isinstance(v, Field):
                if v.fieldname is None:
                    v.fieldname = k # 没有名字则使用属性名
                if v.pk:
                    primarykeys.append(v)

        attrs['__primarykeys__'] = primarykeys

        return super().__new__(cls, name, bases, attrs)


class ModelBase(metaclass=ModelMeta):
    '''从ModelBase继承的类的类型都是ModelMeta'''
    pass


class Student(ModelBase):
    id = Field(pk=True, nullable=False)
    name = Field('username', nullable=False)
    age = Field()

print(Student.__dict__)
```

元编程总结

元类是制造类的工厂，是生成类的类。
定义一个元类，需要使用type(name, bases, dict)，也可以继承type。
构造好元类，就可以在类定义是使用关键字参数metaclass指定元类，可以使用最原始的metatype(name, bases, dict)的方式构造一个类。
元类的`__new__()`方法中，可以获取元类信息、当前类、基类、类属性字典。

元编程一般用于框架开发中。

> 开发中除非你明确的知道自己在干什么，否则不要随便使用元编程
> 99%的情况下用不到元类，可能有些程序员一辈子都不会使用元类

Django、SQLAlchemy使用了元类，让我们使用起来很方便。