

18 | [名师分享] metaclass, 是潘多拉魔盒还是阿拉丁神灯?

2019-06-19 蔡元楠

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 10:12 大小 9.35M



你好，我是蔡元楠，是极客时间《大规模数据处理实战》的作者。很高兴受邀来我们专栏分享，今天我分享的主题是：metaclass，是潘多拉魔盒还是阿拉丁神灯？

Python 中有很多黑魔法，比如今天我将分享的 metaclass。我认识许多人，对于这些语言特性有两种极端的观点。

一种人觉得这些语言特性太牛逼了，简直是无所不能的阿拉丁神灯，必须找机会用上才能显示自己的 Python 实力。

另一种观点则是认为这些语言特性太危险了，会蛊惑人心去滥用，一旦打开就会释放“恶魔”，让整个代码库变得难以维护。

其实这两种看法都有道理，却又都浅尝辄止。今天，我就带你来看看，metaclass 到底是潘多拉魔盒还是阿拉丁神灯？

市面上的很多中文书，都把 metaclass 译为“元类”。我一直认为这个翻译很糟糕，所以也不想在这里称 metaclass 为元类。因为如果仅从字面理解，“元”是“本源”“基本”的意思，“元类”会让人以为是“基本类”。难道 Python 的 metaclass，指的是 Python 2 的 Object 吗？这就让人一头雾水了。


事实上，meta-class 的 meta 这个词根，起源于希腊语词汇 meta，包含下面两种意思：

1. “Beyond” ， 例如技术词汇 metadata， 意思是描述数据的超越数据；
2. “Change” ， 例如技术词汇 metamorphosis， 意思是改变的形态。

metaclass， 一如其名， 实际上同时包含了 “超越类” 和 “变形类” 的含义， 完全不是 “基本类” 的意思。 所以， 要深入理解 metaclass， 我们就要围绕它的**超越变形**特性。 接下来， 我将为你展开 metaclass 的超越变形能力， 讲清楚 metaclass 究竟有什么用？ 怎么应用？ Python 语言设计层面是如何实现 metaclass 的？ 以及使用 metaclass 的风险。

metaclass 的超越变形特性有什么用？

YAML是一个家喻户晓的 Python 工具， 可以方便地序列化 / 逆序列化结构数据。 YAMLObject 的一个**超越变形能力**， 就是它的任意子类支持序列化和反序列化（serialization & deserialization）。 比如说下面这段代码：

 复制代码

```
1 class Monster(yaml.YAMLObject):
2     yaml_tag = u'!Monster'
3     def __init__(self, name, hp, ac, attacks):
4         self.name = name
5         self.hp = hp
```

```

6     self.ac = ac
7     self.attacks = attacks
8     def __repr__(self):
9         return "%s(name=%r, hp=%r, ac=%r, attacks=%r)" % (
10             self.__class__.__name__, self.name, self.hp, sel
11             self.attacks)
12
13 yaml.load("""
14 --- !Monster
15 name: Cave spider
16 hp: [2,6]    # 2d6
17 ac: 16
18 attacks: [BITE, HURT]
19 """)
20
21 Monster(name='Cave spider', hp=[2, 6], ac=16, attacks=[
22
23 print yaml.dump(Monster(
24     name='Cave lizard', hp=[3,6], ac=16, attacks=['BITE
25
26 # 输出
27 !Monster
28 ac: 16
29 attacks: [BITE, HURT]
30 hp: [3, 6]
31 name: Cave lizard

```



这里 `YAMLObject` 的特异功能体现在哪里呢？

你看，调用统一的 `yaml.load()`，就能把任意一个 `yaml` 序列载入成一个 `Python Object`；而调用统一的

`yaml.dump()`，就能把一个 `YAMLObject` 子类序列化。对于 `load()` 和 `dump()` 的使用者来说，他们完全不需要提前知道任何类型信息，这让超动态配置编程成了可能。在我的实战经验中，许多大型项目都需要应用这种超动态配置的理念。

比方说，在一个智能语音助手的大型项目中，我们有 1 万个语音对话场景，每一个场景都是不同团队开发的。作为智能语音助手的核心团队成员，我不可能去了解每个子场景的实现细节。

在动态配置实验不同场景时，经常是今天我要实验场景 A 和 B 的配置，明天实验 B 和 C 的配置，光配置文件就有几万行量级，工作量不可谓不小。而应用这样的动态配置理念，我就可以让引擎根据我的文本配置文件，动态加载所需要的 Python 类。

对于 YAML 的使用者，这一点也很方便，你只要简单地继承 `yaml.YAMLObject`，就能让你的 Python Object 具有序列化和逆序列化能力。是不是相比普通 Python 类，有一点“变态”，有一点“超越”？

事实上，我在 Google 见过很多 Python 开发者，发现能深入解释 YAML 这种设计模式优点的人，大概只有 10%。而


能知道类似 YAML 的这种动态序列化 / 逆序列化功能正是用 metaclass 实现的人，更是凤毛麟角，可能只有 1% 了。

metaclass 的超越变形特性怎么用？

刚刚提到，估计只有 1% 的 Python 开发者，知道 YAML 的动态序列化 / 逆序列化是由 metaclass 实现的。如果你追问，YAML 怎样用 metaclass 实现动态序列化 / 逆序列化功能，可能只有 0.1% 的人能说得出一二了。


因为篇幅原因，我们这里只看 YAMLObject 的 load() 功能。简单来说，我们需要一个全局的注册器，让 YAML 知道，序列化文本中的 !Monster 需要载入成 Monster 这个 Python 类型。

一个很自然的想法就是，那我们建立一个全局变量叫 registry，把所有需要逆序列化的 YAMLObject，都注册进去。比如下面这样：

 复制代码

```
1 registry = {}
2
3 def add_constructor(target_class):
4     registry[target_class.yaml_tag] = target_class
```


然后，在 `Monster` 类定义后面加上下面这行代码：

 复制代码

```
1 add_constructor(Monster)
```

但这样的缺点也很明显，对于 YAML 的使用者来说，每一个 YAML 的可逆序列化的类 `Foo` 定义后，都需要加上一句话，`add_constructor(Foo)`。这无疑给开发者增加了麻烦，也更容易出错，毕竟开发者很容易忘了这一点。

那么，更优的实现方式是什么样呢？如果你看过 YAML 的源码，就会发现，正是 `metaclass` 解决了这个问题。


 复制代码

```
1 # Python 2/3 相同部分
2 class YAMLObjectMetaclass(type):
3     def __init__(cls, name, bases, kwds):
4         super(YAMLObjectMetaclass, cls).__init__(name, base
5             if 'yaml_tag' in kwds and kwds['yaml_tag'] is not N
6                 cls.yaml_loader.add_constructor(cls.yaml_tag, cls
7     # 省略其余定义
8
9 # Python 3
10 class YAMLObject(metaclass=YAMLObjectMetaclass):
```

```
11     yaml_loader = Loader
12     # 省略其余定义
13
14 # Python 2
15 class YAMLObject(object):
16     __metaclass__ = YAMLObjectMetaclass
17     yaml_loader = Loader
18     # 省略其余定义
```




你可以发现，YAMLObject 把 metaclass 都声明成了 YAMLObjectMetaclass，尽管声明方式在 Python 2 和 3 中略有不同。在 YAMLObjectMetaclass 中，下面这行代码就是魔法发生的地方：

 复制代码


```
1 cls.yaml_loader.add_constructor(cls.yaml_tag, cls.from_
```



YAML 应用 metaclass，拦截了所有 YAMLObject 子类的定义。也就说说，在你定义任何 YAMLObject 子类时，Python 会强行插入运行下面这段代码，把我们之前想要的 `add_constructor(Foo)` 给自动加上。

 复制代码

```
1 cls.yaml_loader.add_constructor(cls.yaml_tag, cls.from_
```

所以 YAML 的使用者，无需自己去手写

`add_constructor(Foo)`。怎么样，是不是其实并不复杂？

看到这里，我们已经掌握了 metaclass 的使用方法，超越了世界上 99.9% 的 Python 开发者。更进一步，如果你能够深入理解，Python 的语言设计层面是怎样实现 metaclass 的，你就是世间罕见的“Python 大师”了。


Python 底层语言设计层面是如何实现 metaclass 的？

刚才我们提到，metaclass 能够拦截 Python 类的定义。它是怎么做到的？

要理解 metaclass 的底层原理，你需要深入理解 Python 类型模型。下面，我将分三点来说明。

第一，所有的 Python 的用户定义类，都是 `type` 这个类的实例。

可能会让你惊讶，事实上，类本身不过是一个名为 `type` 类的实例。在 Python 的类型世界里，`type` 这个类就是造物的上帝。这可以在代码中验证：

 复制代码


```
1 # Python 3 和 Python 2 类似
2 class MyClass:
3     pass
4
5 instance = MyClass()
6
7 type(instance)
8 # 输出
9 <class '__main__.C'>
10
11 type(MyClass)
12 # 输出
13 <class 'type'>
```

你可以看到，`instance` 是 `MyClass` 的实例，而 `MyClass` 不过是“上帝”`type` 的实例。

第二，用户自定义类，只不过是 `type` 类的 `__call__` 运算符重载。


当我们定义一个类的语句结束时，真正发生的情况，是 Python 调用 `type` 的 `__call__` 运算符。简单来说，当你定

义一个类时，写成下面这样时：

 复制代码


```
1 class MyClass:  
2     data = 1
```

Python 真正执行的是下面这段代码：

 复制代码


```
1 class = type(classname, superclasses, attributedict)
```

这里等号右边的`type(classname, superclasses, attributedict)`，就是 `type` 的 `__call__` 运算符重载，它会进一步调用：

 复制代码

```
1 type.__new__(typeclass, classname, superclasses, attrib  
2 type.__init__(class, classname, superclasses, attribute
```

当然，这一切都可以通过代码验证，比如下面这段代码示例：

 复制代码


```
1 class MyClass:
2     data = 1
3
4 instance = MyClass()
5 MyClass, instance
6 # 输出
7 (__main__.MyClass, <__main__.MyClass instance at 0x7fe4
8 instance.data
9 # 输出
10 1
11
12 MyClass = type('MyClass', (), {'data': 1})
13 instance = MyClass()
14 MyClass, instance
15 # 输出
16 (__main__.MyClass, <__main__.MyClass at 0x7fe4f0aea5d0>
17
18 instance.data
19 # 输出
20 1
```

由此可见，正常的 MyClass 定义，和你手工去调用 type 运算符的结果是完全一样的。

第三，metaclass 是 type 的子类，通过替换 type 的__call__运算符重载机制，“超越变形”正常的类。

其实，理解了以上几点，我们就会明白，正是 Python 的类创建机制，给了 metaclass 大展身手的机会。

一旦你把一个类型 MyClass 的 metaclass 设置成 MyMeta，MyClass 就不再由原生的 type 创建，而是会调用 MyMeta 的__call__运算符重载。

 复制代码

```
1 class = type(classname, superclasses, attributedict)
2 # 变为了
3 class = MyMeta(classname, superclasses, attributedict)
```

所以，我们才能在上面 YAML 的例子中，利用 YAMLObjectMetaclass 的__init__方法，为所有 YAMLObject 子类偷偷执行add_constructor()。

使用 metaclass 的风险

前面的篇幅，我都是在讲 metaclass 的原理和优点。的确，只有深入理解 metaclass 的本质，你才能用好

metaclass。而不幸的是，正如我开头所说，深入理解 metaclass 的 Python 开发者，只占了 0.1% 不到。

不过，凡事有利必有弊，尤其是 metaclass 这样“逆天”的存在。正如你所看到的那样，metaclass 会“扭曲变形”正常的 Python 类型模型。所以，如果使用不慎，对于整个代码库造成的风险是不可估量的。

换句话说，metaclass 仅仅是给小部分 Python 开发者，在开发框架层面的 Python 库时使用的。而在应用层，metaclass 往往不是很好的选择。

也正因为这样，据我所知，在很多硅谷一线大厂，使用 Python metaclass 需要特例特批。

总结

这节课，我们通过解读 YAML 的源码，围绕 metaclass 的设计本意“超越变形”，解析了 metaclass 的使用场景和使用方法。接着，我们又进一步深入到 Python 语言设计层面，搞明白了 metaclass 的实现机制。

正如我取的标题那样，metaclass 是 Python 黑魔法级别的语言特性。天堂和地狱只有一步之遥，你使用好

metaclass，可以实现像 YAML 那样神奇的特性；而使用不好，可能就会打开潘多拉魔盒了。

所以，今天的内容，一方面是帮助有需要的同学，深入理解 metaclass，更好地掌握和应用；另一方面，也是对初学者的科普和警告：不要轻易尝试 metaclass。

思考题

学完了上节课的 Python 装饰器和这节课的 metaclass，你知道了，它们都能干预正常的 Python 类型机制。那么，你觉得装饰器和 metaclass 有什么区别呢？欢迎留言和我讨论。

 极客时间

Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 强大的装饰器

下一篇 19 | 深入理解迭代器和生成器

精选留言 (23)

写留言



奔跑的蜗牛

2019-06-19

看不懂了 😊



8



Hoo-Ah

2019-06-19

之前讲装饰器的时候讲到函数装饰器和类装饰器，而类装饰器就是在类里面定义了`__call__`方法，之后在函数执行的时候会调用类的`__call__`方法。

在metaclass中重载了`__call__`方法，在使用metaclass实例化生成类的时候也是调用了`__call__`方法，从这方面来...



5



你曾是少年

2019-06-19

基础不够，之前没接触过metaclass，这一讲读起来太费劲了



你看起来很好吃

2019-06-19

‘用户自定义类，只不过是 type 类的__call__运算符’
景老师，这里这段是不是有点问题，我做了以下实验：

```
class MyMeta(type):  
    def __init__(cls, name, bases, dict):...
```



尘墨

2019-06-20

我尝试着自己写了一个例子，发现好像清晰多了，没有看懂的大家可以看一下

```
class Mymeta(type):  
    def __init__(self, name, bases, dic):  
        super().__init__(name, bases, dic)...
```

作者回复: 棒



SCAR

2019-06-19

哎，对metaclass的机制真是一知半解啊，是90%那坨的！

metaclass和类装饰器都可以动态定制或修改类，类装饰器比metaclass以更简单的方式做到创建类时定制类，但它只能定制被装饰的这个类，而对继承这个类的类失效。metaclass的功能则是要强大的多，它可以定制类的继承关系。



KaitoShy

2019-06-19

```
yaml.load("""  
--- !Monster  
name: Cave spider  
hp: [2,6] # 2d6  
ac: 16...
```

作者回复: 很好





程序员人生

2019-06-19

装饰器像AOP，metaclass像反射机制

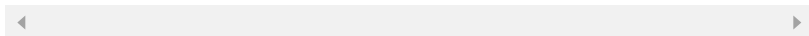


Destroy、

2019-06-19

一开始还以为我打开错专栏了。 目前看了好多解释metaclass的文章，感觉这一篇看起来最明了。

作者回复: 谢谢



lllong33

2019-06-23

metaclass 开发框架层面的 Python 库时使用，应用层不适用。唯一看到过就是看别人写的简易爬虫框架源码见过。yaml模块也没有用，json用的最多，路很长啊，



Geek_Wison

2019-06-23

老师你好，为什么我执行示例代码会一个constructor错

误，查了好久资料都解决不了。

```
import yaml  
class Monster(yaml.YAMLObject):  
    yaml_tag = u'!Monster'...
```



小侠龙旋风

2019-06-22

我默认安装最新的pyyaml5.1版本上面的案例没运行通过，需要pip install PyYAML==3.10安装这个版本才行。

请教老师，继承于yaml.YAMLObject的类能否重写__call__()使之变成类装饰器，这样使用序列化和反序列化会不会更简便呢？



Wing·三金

2019-06-22

个人粗浅的理解是：metaclass 与 类装饰器相似，大多数情况下本质上都是重载了 __call__ 函数，但有一个明显的区别是前者对【继承了 metaclass 的子类本身】的属性造成了影响，而类装饰器是对【作为装饰器本身的类】造成影响而已，对【被装饰的类】的属性没有直接影响...





图·美克尔

2019-06-21

装饰器和metaclass都是给对象增加一些额外的公共配件，但装饰器不影响对象本身，而metaclass是将对象本身进行改造。是设计模式层面的东西。



Geek_974cd5

2019-06-20

装饰器在不影响代码整体业务逻辑的基础上，方便代码的调试，跟踪，日志记录等；
metaclass更多的强调动态性，需要有安全性，完整性校验的代码做保障，容易造成反序列化漏洞；



Jon徐

2019-06-20

之前没有接触过 metaclass，感觉用metaclass的作用就是超动态生成类。这节课感觉确实比较魔术，跟上一节装饰器还要再细想一下。

pyyaml 5.1以上，这段代码会报错，要把 `yaml.load()` ...





yshan-20

另一方面，也是对初学者的科普和警告：不要轻易尝试 metaclass。



zengyunda

2019-06-19

注定成为不了1%的人，这一讲似懂非懂的



John Si

2019-06-19

装饰器跟metaclass这两节课内容都很复杂，不知老师能否再详细说明一下，谢谢老师



hlz-123

2019-06-19

关于类装饰器和metaclass，我的理解如下：

1、类装饰器实现功能

```
class A:
    def __init__:
        .....
```



