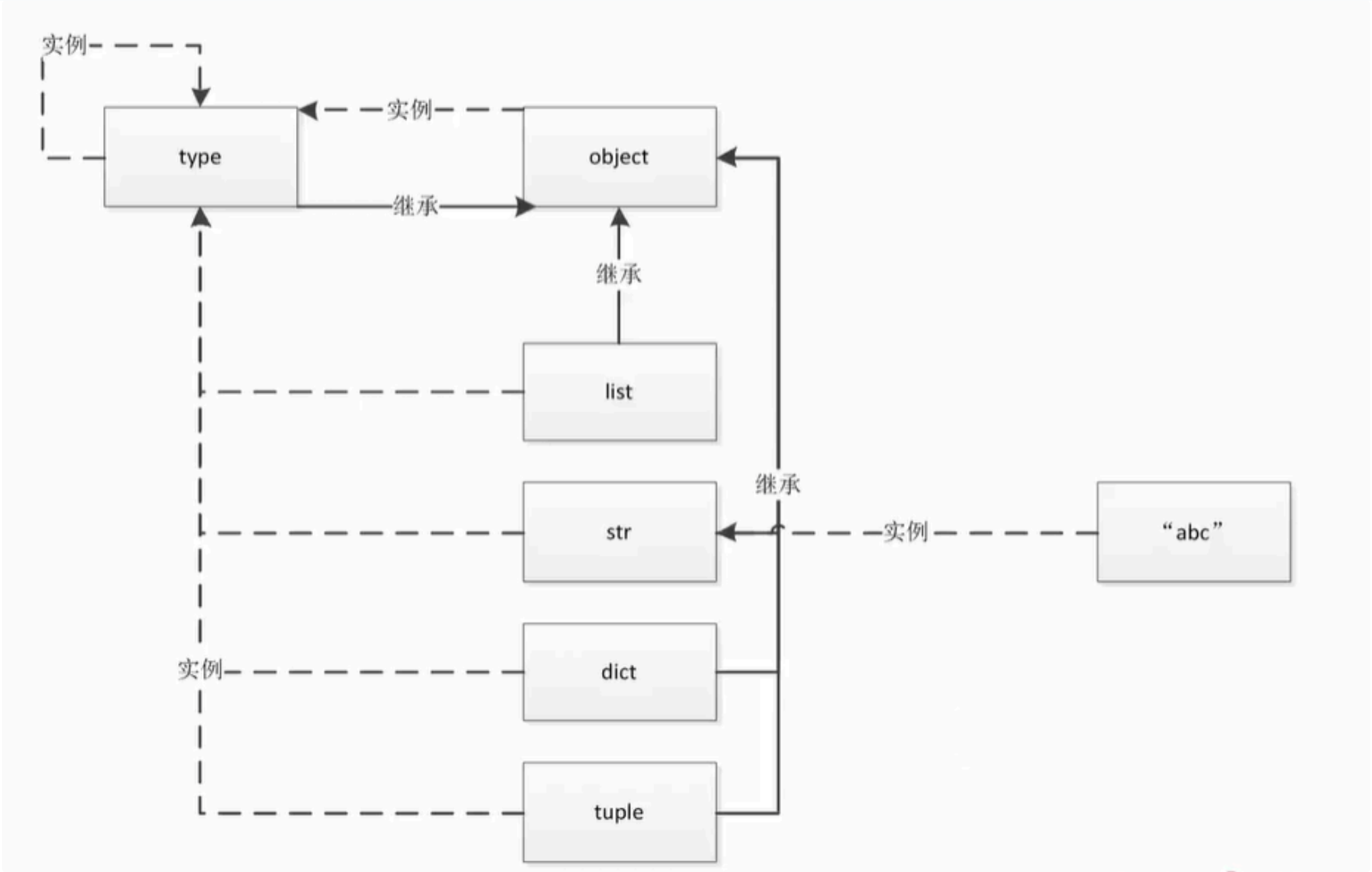
# 进阶必看：python元类编程

发布于 2022-09-20 14:59:31




魔

法

元类编程

在介绍元类编程前，我们先回顾下前面的内容，在之前的文章我们有了解过 python 的面对对象编程的基本实现机制就是魔法函数，而在python3中，新型类统一了类和类型的概念。所有的类都是type的实例包括type自身也是自己的实例，除object之外其他类都继承object。

如上图

动态创建类

由所有类的实例都为type可以得到如下结论

class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义，如下图的例子

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

直接用type创建一个类而不是写Class

**多**

**态**

type是python多态的体现

通过type()函数创建的类和直接写class是完全一样的，因为Python 解释器 遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用 编译器 ，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

**元**

**类**

metaclass

metaclass 直译过来就是元类

简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类就是元类编程。

定义 `ListMetaclass` ，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

```
# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value: self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数 `metaclass` ：

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

**深**

**入**

Perseverance Prevails

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。但是如果你是进阶，还是了解一下，和抽象是息息相关的。

上图是网上找来的一个栗子

当我们传入关键字参数metaclass时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.__new__()来创建。这个方法中的MyList是有add方法的，但是普通的python自带的list是没有这个方法的。

思考：动态修改有什么意义？直接在MyList定义中写上add()方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass不太符合常理。

**应**

**用**

一个单例模式

```
class Logger(metaclass=SingletonMetaclass):
    def __init__(self):
        self._formatter = None
        self._logger = logging.getLogger(self.name)
        self._logger.setLevel(logging.INFO)
        self.__init_syslog_handler()
        self.__init_console_handler()
        self.__init_file_handler()
        self.logger.info(f"######## {self.name}日志类初始化#######")
```

```python
class SingletonMetaThreadSafe(type):
    """
    线程安全的单例类的metaclass
    >>> class BusinessClass(metaclass=SingletonMetaThreadSafe):
    >>>     pass
    """
    _instances = {}
    _lock: Lock = Lock()

    def __call__(cls, *args, **kwargs):
        with cls._lock:
            if cls not in cls._instances:
                instance = super().__call__(*args, **kwargs)
                cls._instances[cls] = instance
        return cls._instances[cls]
```

元类使类的创建行为发生了改变，当用户定义一个class Logger()时，Python解释器首先在当前类Logger的定义中查找metaclass,找到了，就使用Logger中定义的metaclass的SingletonMetaThreadSafe来创建Logger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，换一句话，元类里的方法优先级最高，所以说，它改变了类创建的行为，是很危险的，一般不推荐使用。

**优**

**点**

我们讲了这么多，肯定不会只得到不推荐使用的结论，毕竟，存在即合理。

就元类本身而言，它的作用是：

1.拦截类的创建

2.修改类

3.返回修改之后的类

使用元类还是有一些好处的：

1.意图更加明确。当然你的metaclass名字要起好

2.面向对象。可以隐式继承到子类

3.可以更好地组织代码

可以用__new__，__init__,__call__等方法更好地控制。

回到例子本身思考

这个单例模式的栗子已经说明了它的优点。如果我想做个线程安全的日志，我可以把获取锁的操作单独加在logger中，那如果我们又要写一个线程安全的缓存呢，是不是也要加锁在缓存类。由此，我们可以抽象出一个很重要的基类用来控制线程安全，也就是加一个锁，并且这个基类中的一些行为方法，比如基类中的__call的优先级是最高的,它就拦截了logger实例的创建，那么日志中再写call就会被覆盖掉，更能保证线程的安全。



html python 日志服务 java