Python单例模式(Singleton)的N种实现

zhuanlan.zhihu.com/p/37534850

上海交通大学 计算机应用技术硕士

很多初学者喜欢用**全局变量**，因为这比函数的参数传来传去更容易让人理解。确实在很多场景下用全局变量很方便。不过如果代码规模增大，并且有多个文件的时候，全局变量就会变得比较混乱。你可能不知道在哪个文件中定义了相同类型甚至重名的全局变量，也不知道这个变量在程序的某个地方被做了怎样的操作。

因此对于这种情况，有种更好的实现方式：
单例（Singleton）

单例是一种**设计模式**，应用该模式的类只会生成一个实例。

单例模式保证了在程序的不同位置都**可以且仅可以取到同一个对象实例**：如果实例不存在，会创建一个实例；如果已存在就会返回这个实例。因为单例是一个类，所以你也可以为其提供相应的操作方法，以便于对这个实例进行管理。

举个例子来说，比如你开发一款游戏软件，游戏中需要有"场景管理器"这样一种东西，用来管理游戏场景的切换、资源载入、网络连接等等任务。这个管理器需要有多种方法和属性，在代码中很多地方会被调用，且被调用的必须是同一个管理器，否则既容易产生冲突，也会浪费资源。这种情况下，单例模式就是一个很好的实现方法。

单例模式广泛应用于各种开发场景，对于开发者而言是必须掌握的知识点，同时在很多面试中，也是常见问题。本篇文章总结了目前主流的实现单例模式的方法供读者参考。

希望看过此文的同学，在以后被面到此问题时，能直接皮一下面试官，"我会 4 种单例模式实现，你想听哪一种？"

以下是实现方法索引：

- 使用函数装饰器实现单例

- 使用类装饰器实现单例
- 使用 __new__ 关键字实现单例
- 使用 metaclass 实现单例

使用函数装饰器实现单例

以下是实现代码：

```
def singleton(cls):
    _instance = {}

    def inner():
        if cls not in _instance:
            _instance[cls] = cls()
        return _instance[cls]
    return inner

@singleton
class Cls(object):
    def __init__(self):
        pass

cls1 = Cls()
cls2 = Cls()
print(id(cls1) == id(cls2))
```

输出结果：

```
True
```

在 Python 中，id 关键字可用来查看对象在内存中的存放位置，这里 cls1 和 cls2 的 id 值相同，说明他们指向了同一个对象。

关于装饰器的知识，有不明白的同学可以查看之前的文章 【编程课堂】装饰器浅析 或者使用搜索引擎再学习一遍。代码中比较巧妙的一点是:

```
_instance = {}
```

使用不可变的**类地址**作为键，其实例作为值，每次创造实例时，首先查看该类是否存在实例，存在的话直接返回该实例即可，否则新建一个实例并存放在字典中。

使用类装饰器实现单例

代码：

```
class Singleton(object):
    def __init__(self, cls):
        self._cls = cls
        self._instance = {}
    def __call__(self):
        if self._cls not in self._instance:
            self._instance[self._cls] = self._cls()
        return self._instance[self._cls]

@Singleton
class Cls2(object):
    def __init__(self):
        pass

cls1 = Cls2()
cls2 = Cls2()
print(id(cls1) == id(cls2))
```

同时，由于是面对对象的，这里还可以这么用

```
class Cls3():
    pass

Cls3 = Singleton(Cls3)
cls3 = Cls3()
cls4 = Cls3()
print(id(cls3) == id(cls4))
```

使用 类装饰器实现单例的原理和 函数装饰器 实现的原理相似，理解了上文，再理解这里应该不难。

New、Metaclass 关键字

在接着说另外两种方法之前，需要了解在 Python 中一个类和一个实例是通过哪些方法以怎样的顺序被创造的。

简单来说，**元类(metaclass)** 可以通过方法 **__metaclass__** 创造了**类(class)**，而**类(class)**通过方法 **__new__** 创造了**实例(instance)**。

在单例模式应用中，在创造类的过程中或者创造实例的过程中稍加控制达到最后产生的实例都是一个对象的目的。

本文主讲单例模式，所以对这个 topic 只会点到为止，有感兴趣的同学可以在网上搜索相关内容，几篇参考文章：

- What are metaclasses in Python?
  stackoverflow.com/quest
- python-__new__-magic-method-explained
  howto.lintel.in/python-
- Why is __init__() always called after __new__()?
  stackoverflow.com/quest

使用 new 关键字实现单例模式

使用 __new__ 方法在创造实例时进行干预，达到实现单例模式的目的。

```
class Single(object):
    _instance = None
    def __new__(cls, *args, **kw):
        if cls._instance is None:
            cls._instance = object.__new__(cls, *args, **kw)
        return cls._instance
    def __init__(self):
        pass

single1 = Single()
single2 = Single()
print(id(single1) == id(single2))
```

在理解到 __new__ 的应用后，理解单例就不难了，这里使用了

```
_instance = None
```

来存放实例，如果 _instance 为 None，则新建实例，否则直接返回 _instance 存放的实例。

使用 metaclass 实现单例模式

同样，我们在类的创建时进行干预，从而达到实现单例的目的。

在实现单例之前，需要了解使用 type 创造类的方法，代码如下：

```
def func(self):
    print("do sth")

Klass = type("Klass", (), {"func": func})

c = Klass()
c.func()
```

以上，我们使用 type 创造了一个类出来。这里的知识是 mataclass 实现单例的基础。

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class Cls4(metaclass=Singleton):
    pass

cls1 = Cls4()
cls2 = Cls4()
print(id(cls1) == id(cls2))
```

这里，我们将 metaclass 指向 Singleton 类，让 Singleton 中的 type 来创造新的 Cls4 实例。

小结

本文虽然是讲单例模式，但在实现单例模式的过程中，涉及到了蛮多高级 Python 语法，包括装饰器、元类、new、type 甚至 super 等等。对于新手同学可能难以理解，其实在工程项目中并不需要你掌握的面面俱到，掌握其中一种，剩下的作为了解即可。

by 周鑫鑫

关于更多的设计模式，给初学者推荐《**Head First 设计模式**》（Head First Design Patterns），此书浅显易懂，在 Head First 系列书籍里面也算是很好的一本。

我们的资源网盘里有电子版，获取地址请在公众号（**Crossin的编程教室**）里回复关键字：**资源**

其他文章及回答：

如何自学Python | 新手引导 | 精选Python问答 | Python单词表 | 区块链 | 人工智能 | 双11 | 嘻哈 | 爬虫 | 排序算法

欢迎搜索及关注：**Crossin的编程教室**