**单例模式**

单例模式就是确保一个类只有一个实例.当你希望整个系统中,某个类只有一个实例时,单例模式就派上了用场.  
比如,某个服务器的配置信息存在在一个文件中,客户端通过AppConfig类来读取配置文件的信息.如果程序的运行的过程中,很多地方都会用到配置文件信息,则就需要创建很多的AppConfig实例,这样就导致内存中有很多AppConfig对象的实例,造成资源的浪费.其实这个时候AppConfig我们希望它只有一份,就可以使用单例模式.

**实现单例模式的几种方法**

**1. 使用模块**  
其实,python的模块就是天然的单例模式,因为模块在第一次导入的时候,会生成.pyc文件,当第二次导入的时候,就会直接加载.pyc文件,而不是再次执行模块代码.如果我们把相关的函数和数据定义在一个模块中,就可以获得一个单例对象了.  
新建一个python模块叫singleton,然后常见以下python文件  
mysingleton.py

class Singleton(object):

def foo(self):

pass

singleton = Singleton()

使用:

from singleton.mysingleton import singleton

**2. 使用装饰器**  
装饰器里面的外层变量定义一个字典,里面存放这个类的实例.当第一次创建的收,就将这个实例保存到这个字典中.  
然后以后每次创建对象的时候,都去这个字典中判断一下,如果已经被实例化,就直接取这个实例对象.如果不存在就保存到字典中.

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 10:22'

def singleton(cls):

# 单下划线的作用是这个变量只能在当前模块里访问,仅仅是一种提示作用

# 创建一个字典用来保存类的实例对象

\_instance = {}

def \_singleton(\*args, \*\*kwargs):

# 先判断这个类有没有对象

if cls not in \_instance:

\_instance[cls] = cls(\*args, \*\*kwargs) # 创建一个对象,并保存到字典当中

# 将实例对象返回

return \_instance[cls]

return \_singleton

@singleton

class A(object):

a = 1

def \_\_init\_\_(self, x=0):

self.x = x

print('这是A的类的初始化方法')

a1 = A(2)

a2 = A(3)

print(id(a1), id(a2))

**3.使用类**  
思路就是,调用类的instance方法,这样有一个弊端就是在使用类创建的时候,并不是单例了.也就是说在创建类的时候一定要用类里面规定的方法创建

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 11:06'

class Singleton(object):

def \_\_init\_\_(self,\*args,\*\*kwargs):

pass

@classmethod

def get\_instance(cls, \*args, \*\*kwargs):

# 利用反射,看看这个类有没有\_instance属性

if not hasattr(Singleton, '\_instance'):

Singleton.\_instance = Singleton(\*args, \*\*kwargs)

return Singleton.\_instance

s1 = Singleton() # 使用这种方式创建实例的时候,并不能保证单例

s2 = Singleton.get\_instance() # 只有使用这种方式创建的时候才可以实现单例

s3 = Singleton()

s4 = Singleton.get\_instance()

print(id(s1), id(s2), id(s3), id(s4))

**注意,这样的单例模式在单线程下是安全的,但是如果遇到多线程,就会出现问题.如果遇到多个线程同时创建这个类的实例的时候就会出现问题.**

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 11:26'

import threading

class Singleton(object):

def \_\_init\_\_(self, \*args, \*\*kwargs):

pass

@classmethod

def get\_instance(cls, \*args, \*\*kwargs):

if not hasattr(Singleton, '\_instance'):

Singleton.\_instance = Singleton(\*args, \*\*kwargs)

return Singleton.\_instance

def task(arg):

obj = Singleton.get\_instance(arg)

print(obj)

for i in range(10):

t = threading.Thread(target=task, args=[i, ])

t.start()

*执行结果好像也没有问题,那是因为执行的速度足够的快,如果在****init****()方法中有阻塞,就看到非常的明显.*

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 11:26'

import threading

import time

class Singleton(object):

def \_\_init\_\_(self, \*args, \*\*kwargs):

time.sleep(1)

pass

@classmethod

def get\_instance(cls, \*args, \*\*kwargs):

if not hasattr(Singleton, '\_instance'):

Singleton.\_instance = Singleton(\*args, \*\*kwargs)

return Singleton.\_instance

def task(arg):

obj = Singleton.get\_instance(arg)

print(obj)

for i in range(10):

t = threading.Thread(target=task, args=[i, ])

t.start()

*可以看到是创建了10个不同的实例对象,这是什么原因呢.因为在一个对象创建的过程中,另外一个对象也创建了.当它判断的时候,会先去获取\_instance属性,因为这个时候还没有,它就会调用****init****()方法.结果就是调用了10次,然后就创建了10个对象.*

**如何解决呢?**  
加锁:  
在哪里加锁呢?在获取对象属性\_instance的时候加锁,如果已经有人在获取对象了,其他的人如果要获取这个对象,就要等一哈.因为前面的那个人,可能在第一次创建对象.

**创建对象的时候加锁即可**

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 11:38'

import time

import threading

class Singleton(object):

\_instance\_lock = threading.Lock()

def \_\_init\_\_(self,\*args,\*\*kwargs):

time.sleep(1)

@classmethod

def get\_instance(cls,\*args,\*\*kwargs):

if not hasattr(Singleton,'\_instance'):

with Singleton.\_instance\_lock:

if not hasattr(Singleton,'\_instance'):

Singleton.\_instance = Singleton(\*args,\*\*kwargs)

return Singleton.\_instance

def task(arg):

obj = Singleton.get\_instance(arg)

print(obj)

for i in range(10):

t = threading.Thread(target=task,args=[i,])

t.start()

obj = Singleton.get\_instance()

print(obj)

这种方式创建的单例,必须使用Singleton\_get\_instance()方法,如果使用Singleton()的话,得到的并不是单例.所以我们推荐使用\_\_new\_\_()方法来创建单例,这样创建的单例可以使用类名()的方法进行实例化对象

**4.基于\_\_new\_\_方法实现的单例模式(推荐使用,方便)**  
知识点:  
1> 一个对象的实例化过程是先执行类的\_\_new\_\_方法,如果我们没有写,默认会调用object的\_\_new\_\_方法,返回一个实例化对象,然后再调用\_\_init\_\_方法,对这个对象进行初始化,我们可以根据这个实现单例.  
2> 在一个类的\_\_new\_\_方法中先判断是不是存在实例,如果存在实例,就直接返回,如果不存在实例就创建.

# encoding:utf-8

\_\_author\_\_ = 'Fioman'

\_\_time\_\_ = '2019/3/6 13:36'

import threading

class Singleton(object):

\_instance\_lock = threading.Lock()

def \_\_init\_\_(self, \*args, \*\*kwargs):

pass

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not hasattr(cls, '\_instance'):

with Singleton.\_instance\_lock:

if not hasattr(cls, '\_instance'):

Singleton.\_instance = super().\_\_new\_\_(cls)

return Singleton.\_instance

obj1 = Singleton()

obj2 = Singleton()

print(obj1, obj2)

def task(arg):

obj = Singleton()

print(obj)

for i in range(10):

t = threading.Thread(target=task, args=[i, ])

t.start()

作者：莫辜负自己的一世韶光  
链接：https://www.jianshu.com/p/6a1690f0dd00  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。