首先介绍一下单例模式：   
    单例模式（Singleton），也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。   
  
实现单例模式的思路是：   
    一个类能返回对象一个引用(永远是同一个)和一个获得该实例的方法（必须是静态方法，通常使用getInstance这个名 称）；当我们调用这个方法时，如果类持有的引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用；同时我们 还将该类的构造函数定义为私有方法，这样其他处的代码就无法通过调用该类的构造函数来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。   
  
需要注意的地方：   
    单例模式在多线程的 应用场合下必须小心使用。如果当唯一实例尚未创建时，有两个线程同时调用创建方法，那么它们同时没有检测到唯一实例的存在，从而同时各自创建了一个实例， 这样就有两个实例被构造出来，从而违反了单例模式中实例唯一的原则。 解决这个问题的办法是为指示类是否已经实例化的变量提供一个互斥锁(虽然这样会降低效率)。   
  
优点：   
    1.在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就 防止其它对象对自己的实例化，确保所有的对象都访问一个实例   
    2.单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。   
    3.提供了对唯一实例的受控访问。   
    4.由于在系统内存中只存在一个对象，因此可以 节约系统资源，当 需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。   
    5.允许可变数目的实例。   
    6.避免对共享资源的多重占用。   
缺点：   
    1.不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。   
    2.由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。   
    3.单例类的职责过重，在一定程度上违背了“单一职责原则”。   
    4.滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。   
使用注意事项：   
    1.使用时不能用反射模式创建单例，否则会实例化一个新的对象   
    2.使用懒单例模式时注意线程安全问题   
    3.饿单例模式和懒单例模式构造方法都是私有的，因而是不能被继承的，有些单例模式可以被继承（如登记式模式）   
适用场景：   
    单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：   
    1.需要频繁实例化然后销毁的对象。   
    2.创建对象时耗时过多或者耗资源过多，但又经常用到的对象。   
    3.有状态的工具类对象。   
    4.频繁访问数据库或文件的对象。   
以下都是单例模式的经典使用场景：   
    1.资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如上述中的日志文件，应用配置。   
    2.控制资源的情况下，方便资源之间的互相通信。如线程池等。   
应用场景举例：   
    1.外部资源：每台计算机有若干个打印机，但只能有一个PrinterSpooler，以避免两个打印作业同时输出到打印机。内部资源：大多数软件都有一个（或多个）属性文件存放系统配置，这样的系统应该有一个对象管理这些属性文件   
    2. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~   
    3. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。   
    4. 网站的计数器，一般也是采用单例模式实现，否则难以同步。   
    5. 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。   
    6. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。   
    7. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。   
    8. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。   
    9. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。   
    10. HttpApplication 也是单位例的典型应用。熟悉ASP.Net(IIS)的整个请求生命周期的人应该知道HttpApplication也是单例模式，所有的HttpModule都共享一个HttpApplication实例.   
　　   
实现单利模式的原则和过程：   
    1.单例模式：确保一个类只有一个实例，自行实例化并向系统提供这个实例   
    2.单例模式分类：饿单例模式（类加载时实例化一个对象给自己的引用），懒单例模式（调用取得实例的方法如getInstance时才会实例化对象）（java中饿单例模式性能优于懒单例模式，c++中一般使用懒单例模式）   
    3.单例模式要素：   
        a.私有构造方法   
        b.私有静态引用指向自己实例   
        c.以自己实例为返回值的公有静态方法

**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。