call()魔术方法

```
作用：实现了__call__()魔术方法的类的实例对象会变成一个可调用对象，对实例对象进行调用，实际上是调用了实例对象的__call__()魔术方法

class Maker():
    name="hello"

    def __call__(self):
        print("当对象象函数一样调用时,我被调用")
    def myfunc(self):
        print("myfunc")

m=Maker()
m()#当对象象函数一样调用时,我被调用
```

使用类作为装饰器

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
import time

class Maker():
    def __init__(self,func):
        self.func=func

    def __call__(self,*arg):
        ret=self.func(*arg)
        return ret+1000

#这里的Maker(myfunc)是生成一个对象
@Maker#myfunc=Maker(myfunc)
def myfunc(a,b):
    time.sleep(3)
    return a+b

#这里使用对象调用,那么类中就必须有__call__函数
print(myfunc(10,20))
```

使用装饰器来实现单例模式

```
单例模式:一个类只能实例化一个对象
```

通过装饰器实现

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

# class Maker():
#     def myfunc(self):
#         print("Maker myfunc")
#
# m=Maker()
# m2=Maker()
# print(id(m))
# print(id(m2))


#1.定义一个全局的字典变量,用于保存单例模式下实例化出的对象,键是类名,值是对象
instances={}


def myfunc(cls):
    def mytest(*args,**kwargs):
        if cls.__name__ not in instances.keys():
            #如果这个类名不在字典的键里,那么生成对象
            instances[cls.__name__]=cls(*args,**kwargs)
        return instances[cls.__name__]
    return mytest

@myfunc    #Maker=myfunc(Maker)
class Maker():
    pass


t=Maker()
t2=Maker()
print(id(t))
print(id(t2))
```

通过new()魔术方法实现单例模式

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
```

```
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

class Maker():
    __instance=None

    def __new__(cls, *args, **kwargs):
        if cls.__instance is None:
            #如果__instance没有值,那么就使用object的来实例化对象
            cls.__instance=super().__new__(cls)
        return cls.__instance

t=Maker()
t2=Maker()
print(id(t))
print(id(t2))

```

类属性

```
实例属性： 在__init__魔术方法中初始化的属性，或者通过实例对象添加的属性
类属性：  在类名称下面定义的属性

class Maker():
    name="mytest"#类属性
    def myfunc(self):
        self.age=18#实例属性

#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

class Maker():
    name="张安"

    def mytest(self):
        print(self.name)#在成员函数中使用self来访问类属性
        print(id(self.name))

m=Maker()
m.mytest()
m.name="李四"#这里不是修改类属性,而是增加一个实例属性
m.mytest()
```

```
28
29 Maker.name="王五"#修改类属性的值
30 print(m.name)
31 print(Maker.name)
```

方法

实例方法

第一个参数是self, 代表当前实例对象本身

类方法

第一个参数是cls,代表当前的类对象
需要在方法的上面加上@clssmethod
类方法可以通过实例对象和类对象来访问

```
1  #!/usr/bin/env python
2  # -*- coding:utf-8 -*-
3  #====#====#====#====
4  #Author:
5  #CreatDate:
6  #Version:
7  #====#====#====#====
8
9  class Maker():
10     def myfunc(self):
11         print("我是普通的成员函数")
12
13     @classmethod
14     def mytest(cls):
15         print("我是类方法")
16
17
18 m=Maker()
19 m.myfunc()
20
21 # Maker.myfunc()#报错,不能使用类名调用普通的成员函数
22
23 Maker.mytest()
24
25
26 作用:方便别人直接使用类名来访问类方法,而不需要再去实例化对象
```

静态方法

不需要额外的参数
通过@staticmethod来进行装饰
静态方法可以通过类对象和实例对象来进行访问

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====

class Maker():

    name="hello"
    @staticmethod
    def mytest():
        #静态方法中不能使用类的属性和方法
        a=10
        a+=1
        print("我是静态方法",a)

m=Maker()
m.mytest()
Maker.mytest()


静态方法的作用:
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#静态方法在类中是独立的,单纯的函数,只是托管在类的空间,增强代码的逻辑性,简化代码的维护
import time
class Maker():
    def __init__(self,h,m,s):
        self.h=h
        self.m=m
        self.s=s

    @staticmethod
    def showtime():
        return time.strftime("%H:%M:%S",time.localtime())

print(Maker.showtime())
m=Maker(21,37,56)
kk=m.showtime()
print(kk)
```

进程的概念

```
进程是什么：进程是操作系统进行资源分配的基本单位
比如，我们在操作系统上运行一个应用程序，其实对操作系统来说你就开启了一个进程

单核的CPU,同一时刻只能运维单个进程，虽然可以同时运行多个程序，但进程之间是通过轮流占用CPU来执行的

现在的CPU核数越来越多，有4核，8核，28核等等，为了充分发挥多核优势，提高程序的并发度，我们要使用多进程

```

创建进程的类Process

```
Multiprocessing模块提供了一个创建进程的类 Process,所以你使用Process类之前要引入Multiprocessing模块
创建进程有以下两种方法:
1.创建一个Process类的实例，并指定目标任务函数
2.自定义一个类，并继承Process类，重写__init__()方法和run()方法

一.使用Process类的实例创建进程
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#引入模块
from multiprocessing import Process
#为了获取进程pid
import os
import time
def mytest(d):
    num=0
    for i in range(d*1000000):
        num+=i
    print(f"进程的pid为{os.getpid()}")

#在该代码块内写的代码只能在本文件有,不能被别的文件引用
if __name__=="__main__":
    print("父进程PID为%s "%os.getpid())
    #创建子进程,target的值是子进程要执行的函数,args的值是函数的参数
    p1=Process(target=mytest,args=(3,))
    t0=time.time()#记录当前时间
    #激活子进程
    p1.start()
    p1.join()#阻塞主进程,让子进程完成任务或子进程被终止
    t1 = time.time()  # 记录当前时间
    print(t1-t0)


```

```
#注意,创建进程必须在__main__里面,因为windows创造进程相当于导入该文件，由于创建进程在main函数之前
#  ，会无限递归导入模块创建进程。所以在windows创造进程应放在main函数之后，因为main函数不会执行被导入模块。

二.使用类成绩子进程
#!/usr/bin/env python
# -*- coding:utf-8 -*-
#====#====#====#====
#Author:
#CreatDate:
#Version:
#====#====#====#====
#引入模块
from multiprocessing import Process
#为了获取进程pid
import os
import time

class Maker(Process):
    def __init__(self,d):
        self.d=d
        super().__init__()

    #子进程要执行任务函数
    def run(self):
        num = 0
        for i in range(self.d * 1000000):
            num += i
        print(f"进程的pid为{os.getpid()}")



#在该代码块内写的代码只能在本文件有,不能被别的文件引用
if __name__=="__main__":
    print("父进程PID为%s "%os.getpid())
    #创建子进程,使用类
    p1=Maker(3)
    t0=time.time()#记录当前时间
    #激活子进程
    p1.start()
    p1.join()#阻塞主进程,让子进程完成任务或子进程被终止
    t1 = time.time()  # 记录当前时间
    print(t1-t0)
```

##