单例模式

参考

-
-

单例模式是设计模式之一。这种类型的设计模式数据创建模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

要点：一个类只有一个实例存在。

> 由于要保证对象的唯一性，往往需要利用全局变量的来实现。而不同的语言根据其不同的特性思想一致但是写法上会有所不同。

使用场景

1. 要求生成唯一的序列号
2. 缓存之类的数据，可以用单例缓存起来
3. 创建一个对象需要消耗过多的资源，比如文本I/O与数据的连接

Python中单例模式的实现

1. 使用模块

Python的模块其实就是天然的单例模式，因为模块在第一次导入时，会生成`.pyc`文件，当我们第二次导入时，就会直接加载`.pyc`文件，而不会再次执行模块代码。因此，我们只需把相关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。按照上述思路我们可以这样做：

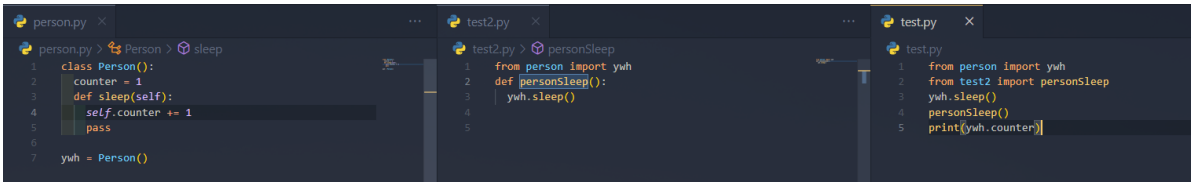
创建一个person.py文件，内容：

```
class Person():
  def sleep(self):
    pass
ywh = Person()
```

要使用时，我们可直接在其他的文件中导入这里创建的对象，这个对象即是单例模式的对象。

```
from person import ywh
```

测试



我们按照上述截图创建三个文件，并且执行`test.py`，最终结果输出`3`,说明单例创建成功。

2. 使用装饰器

```
def Singleton(cls):
  instances = {}
  def getinstance(*args, **kargs):
      if cls not in instances:
          instances[cls] = cls()
      return instances[cls]
  return getinstance

@Singleton
class Person():
  counter = 1
  def sleep(self):
    self.counter += 1

ywh = Person()
ly = Person()
ly.sleep()
print(ywh.counter)
```

最后结果输出:

```
2
```

3. 使用类

```
import threading

class Person():
  _instance_lock = threading.Lock()
  counter = 1
  def sleep(self):
    self.counter += 1

  @classmethod
  def getInstance(cls, *args, **kwargs):
    if not hasattr(cls, '_instance'):
      with cls._instance_lock:    ## 阻塞线程，避免在多线程的情况下创建出多个对象
        if not hasattr(cls, '_instance'):
          cls._instance = Person(*args, **kwargs)
    return cls._instance

ywh = Person.getInstance()
ly = Person.getInstance()
ly.sleep()
print(ywh.counter)
```

> 要点分析:
>
> 1. 在上述代码中要获得单例的对象需要通过 `Person.getInstance()` 获得，直接通过 `Person()` 这种方式创建得到的不是单例的。
> 2. 通过加上线程锁 `threading.Lock()`，保证在多线程程序上的单例。

最后结果输出:

```
2
```

4. 基于__new__方法实现

```
import threading

class Person(object):
  _instance_lock = threading.Lock()
  counter = 1
  def sleep(self):
    self.counter += 1
  def __new__(cls, *args, **kwargs):
    if not hasattr(cls, '_instance'):
      with cls._instance_lock:   ## 阻塞线程，避免在多线程的情况下创建出多个对象
        if not hasattr(cls, '_instance'):
          cls._instance = object.__new__(cls)
    return cls._instance

ywh = Person()
ly = Person()
ly.sleep()
print(ywh.counter)
```

> 利用我们在创建实例化对象时，先执行 __new__ 方法，然后再执行 __init__ 方法，基于这个想法我们对上一种方法进行改进。

最后结果输出:

```
2
```

5. 基于元类(metaclass)实现

> metaclass 可参考:
>
>)

相关知识

```
1.类由type创建，创建类时，type的__init__方法自动执行，类() 执行type的 __call__方法(类的__new__方法,类的__init__方法)
2.对象由类创建，创建对象时，类的__init__方法自动执行，对象()执行类的 __call__ 方法
```

```
import threading
class SingletonType(type):
  _instance_lock = threading.Lock()
  def __call__(cls, *args, **kwargs):
    if not hasattr(cls, "_instance"):
      with cls._instance_lock:
        if not hasattr(cls, "_instance"):
          cls._instance = super(SingletonType, cls).__call__(*args, **kwargs)
    return cls._instance
```

```
class Person(metaclass=SingletonType):
  counter = 1
  def sleep(self):
    self.counter += 1
ywh = Person()
ly = Person()
ly.sleep()
print(ywh.counter)
```

Go语言中的单例模式

> 利用 `sync.Once` 类型去同步 `getInstance` 的访问，并确保我们的对应得到结构体仅被床创建一次。

```
package main

import (
    "fmt"
    "sync"
)

type singleton struct {
    Counter int
}

var once sync.Once
var instance *singleton

// 获得单例对象
func getInstance() *singleton {
    once.Do(func() {
        instance = &singleton{}
    })
    return instance
}
func testSingleton() {
    ywh := getInstance()
    yhm := getInstance()
    ywh.Counter++
    fmt.Println(ywh)
    fmt.Println(yhm)
}

func main() {
    testSingleton()
}
```

输出结果

```
&{1}
&{1}
```