# 设计模式

学习设计模式的一个有效的方式是自己尝试写个示例代码来演示它

## 创建型

### 工厂模式

* 解决对象创建问题
* 解耦对象的创建和使用
* 包括工厂方法和抽象工厂

class DogToy:
 def speak(self):
 print("wang wang")

class CatToy:
 def speak(self):
 print("miao miao")

def toy\_factory(toy\_type):
 if toy\_type == 'dog':
 return DogToy()
 elif toy\_type == 'cat':
 return CatToy()

### 构造模式

* 用于控制复杂对象的构造
* 创建和表示分离

### 原型模式

### 单例模式

单例模式应用的场景一般发现在以下条件下： 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，如日志文件，应用配置。 控制资源的情况下，方便资源之间的互相通信。如线程池等，1,网站的计数器 2,应用配置 3.多线程池 4数据库配置 数据库连接池 5.应用程序的日志应用...

* 单例模式的实现有多种方式
* 单例模式：一个类创建出来的对象都是同一个
* python的模块其实就是单例的，只会导入一次
* 使用共享同一个实例的方法来创建单例模式

class Singleton:
 def \_\_new\_\_(cls, \*args, \*\*kwargs):
 if not hasattr(cls, '\_instance'):
 \_instance = super().\_\_new\_\_(cls, \*args, \*\*kwargs)
 cls.\_instance = \_instance
 return cls.\_instance

class MyClass(Singleton):
 pass

c1 = MyClass()
c2 = MyClass()
c1 is c2 # true

**习题：python如何实现单例模式**

第一种方法:使用装饰器

def singleton(cls):
 instances = {}
 def wrapper(\*args, \*\*kwargs):
 if cls not in instances:
 instances[cls] = cls(\*args, \*\*kwargs)
 return instances[cls]
 return wrapper

@singleton
class Foo(object):
 pass
foo1 = Foo()
foo2 = Foo()
print(foo1 is foo2) # True

第二种方法：使用基类

New 是真正创建实例对象的方法，所以重写基类的new 方法，以此保证创建对象的时候只生成一个实例

class Singleton(object):
 def \_\_new\_\_(cls, \*args, \*\*kwargs):
 if not hasattr(cls, '\_instance'):
 cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls, \*args, \*\*kwargs)
 return cls.\_instance

class Foo(Singleton):
 pass

foo1 = Foo()
foo2 = Foo()

print(foo1 is foo2) # True

第三种方法：元类，元类是用于创建类对象的类，类对象创建实例对象时一定要调用call方法，因此在调用call时候保证始终只创建一个实例即可，type是python的元类

class Singleton(type):
 def \_\_call\_\_(cls, \*args, \*\*kwargs):
 if not hasattr(cls, '\_instance'):
 cls.\_instance = super(Singleton, cls).\_\_call\_\_(\*args, \*\*kwargs)
 return cls.\_instance

# Python2
class Foo(object):
 \_\_metaclass\_\_ = Singleton

# Python3
class Foo(metaclass=Singleton):
 pass

foo1 = Foo()
foo2 = Foo()
print(foo1 is foo2) # True

## 结构型

### 代理模式

* 把一个对象的操作代理到另一个对象
* 通常使用has a 组合关系

### 适配器模式

* 把不同对象的接口适配到同一个接口
* 当我们需要给不同的对象统一接口的时候可以使用适配器模式

## 行为型

### 迭代器模式

* python内置对迭代器模式的支持
* 比如我们可以用for遍历各种Interable的数据类型
* python里可以实现\_\_next\_\_和\_\_iter\_\_实现迭代器

### 观察者模式

* 发布订阅是一种罪常用的实现方式
* 发布订阅用于解耦逻辑
* 可以通过回调等方式实现，当发生事件时，调用相应的回调函数

### 策略模式

* 根据不同的输入采取不同的策略
* 对外暴露统一的接口，内部采用不同的策略计算

## 函数式编程

### map/reduce/filter

reduce(lambda x, y: x+y, range(1,6))