设计模式总结

单例模式

全局只有一个实例

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，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。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

工厂模式

将创建对象和使用对象的功能分离

- 解决的场景
  - 两个类A和B之间的关系应该仅仅是A创建B或是A使用B，而不能两种关系都有
  - 将对象的创建和使用分离，也使得系统更加符合“单一职责原则”，有利于对功能的复用和系统的维护
  - 防止用来实例化一个类的数据和代码在多个类中到处都是
  - 一些类实例的构建过程可能十分复杂，需要进行封装
  - 为类实例的构建提供更好的多态支持（不确定具体对象）

- 工厂模式的好处
  - 解耦
  - 降低代码重复
  - 减少了使用者因为创建逻辑导致的错误
  - Demo：水果工厂
    - 简单工厂，工厂方法，抽象工厂
    - Fruit, Apple, Orange, FruitFactory
    - f.py

```
'''
除下实现方式外，还对以对工厂进一步进行抽象，得到抽象工厂，使得进一步解耦，
不再通过参数来得到要生产的类的对象。
'''
class Fruit:
  pass

class Apple(Fruit):
  pass
  def pie(self):
    print("making apple pie")

class Orange(Fruit):
  pass

  def juice(self):
    print("making orange juice")

class FruitFactory:

  def generate_fruit(self,type):
    if type=='a':
      return Apple() #可以根据apple的定义进行初始化
    elif type=='o':
      return Orange()#可以根据orange的定义进行初始化
    else:
      return None
```

```
ff=FruitFactory()
apple=ff.generate_fruit('a')
orange=ff.generate_fruit('o')
apple.pie()
orange.juice()

#“抽象”工厂的简要实现

class Factory:

  def generate(self):
    pass

class AppleFactory(Factory):

  def generate(self):
    return Apple()#

class OrangeFactory(Factory):

  def generate(self):
    return Orange()
```

代理模式

- Proxy Pattern
  - 在访问某个对象之前执行一个或多个重要的额外操作
  - 访问敏感信息或关键功能前需要具备足够的权限
  - 将计算成本较高的对象的创建过程延迟到用户首次真正使用时才进行
    - 惰性求值

- 常见类型
  - 远程代理：实际存在于不同地址空间（如网络服务器）的对象在本地的代理者
    - 对使用者透明
  - 虚拟代理：用于惰性求值，将一个大计算量对象的创建延迟到真正需要的时候进行
  - 保护/防护代理：控制对敏感对象的访问
  - 智能（引用）代理：在对象被访问时执行额外的动作，如引用计数或线程安全检查等
  - Demo: pdp.py

```
import getpass

class Record:
  def read(self):
    pass
  def add(self,user):
    pass

class RecordError(Exception):
  def __init__(self):
    self.message='Access Record Failed'

class AddUserNotAllowedRecordError(RecordError):
  def __init__(self,user):
    self.message="Add user of {} failed due to no permission!".format(user)

class ReadUsersNotAllowedRecordError(RecordError):
  def __init__(self):
    self.message="read users not allowed due to no permission!"

class KeyRecords(Record):
  def __init__(self):
    self.users=['admin']

  def read(self):
    return ' '.join(self.users)
```

```
  def add(self,user):
    self.users.append(user)

class ProxyRecords(Record):
  def __init__(self):
    self.key_records=KeyRecords()
    self.secrect='test' #硬编码密码，不推荐，仅示例

  def read(self,pwd):
    if self.secrect==pwd:
      return self.key_records.read()
    else:
      raise ReadUsersNotAllowedRecordError()

  def add(self,pwd,user):
    if self.secrect == pwd:
      self.key_records.add(user)
    else:
      raise AddUserNotAllowedRecordError(user)
    return 1

def main():
  pr=ProxyRecords()
  pwd=getpass.getpass("plz input the pwd:")
  try:
    print(pr.read(pwd))
    if pr.add(pwd,'zjc'):
      print("ADD Succeeded...")
  except ReadUsersNotAllowedRecordError as runare:
    print(runare.message)
  except AddUserNotAllowedRecordError as aunare:
    print(aunare.message)
  except RecordError as re:
    print(re.message)

if __name__=='__main__':
  main()
```

观察者模式

- 亦称
  - 发布（publish ）-订阅（Subscribe）模式
  - 模型-视图（View）模式
  - 源-收听者(Listener)模式
  - 从属者模式
- 要义
  - 一个目标对象管理所有依赖于它的观察者对象，并且在它本身的状态改变时主动发出通知
  - 观察者模式完美地将观察者和被观察的对象分离
- 优点
  - 观察者与被观察者之间抽象耦合
  - 可以触发多个符合单一职责的模块
  - 可以很方便地实现广播
- 场景
  - 消息交换，如消息队列；
  - 多级触发，如一个中断即会引发一连串反应
- 缺点
  - 效率不一定高

- Demo 1
  - ps1.py
- Demo 2
  - ps2.py

```
import time
class Investor:
  def __init__(self,name,stock):
    self._name=name
    self._stock=stock

  @property
  def stock(self):
    return self._stock
  @stock.setter
  def stock(self,value):
    self._stock=value

  def update(self):
    print("{} invest on {} with price {}: sell it
now!!!".format(self._name,self._stock.name,self._stock.price))

class Stock:
  def __init__(self,name,price):
    self._name=name
    self._price=price
    self._investors=[]

  @property
  def name(self):
    return self._name

  @property
  def price(self):
    return self._price

  @price.setter
  def price(self,value):
    if self._price>value:
      self.notify()
    self._price=value

  def attach(self,investor):
```

```
    self._investors.append(investor)

  #deattach

  def notify(self):
    for investor in self._investors:
      investor.update()

def main():
  s=Stock('区块链',11.11)
  i1=Investor('zjc',s)
  i2=Investor('lys',s)
  s.attach(i1)
  s.attach(i2)
  s.price=13
  time.sleep(1)
  s.price=10

if __name__=='__main__':
  main()
```

```
import time

class Observer:
  def update(self,info):
    pass

class Subject:
  def __init__(self):
    self.observers=[]
  def addObserver(self,observer):
    self.observers.append(observer)
  #deleteObserver(self,observer):
  def notifyAll(self,info):
    for observer in self.observers:
      observer.update(info)

class Investor(Observer):
  def __init__(self,name):
    self._name=name

  @property
  def name(self):
    return self._name
  #override
  def update(self,info):
    print("Alarm:{}".format(info))
```

```
class Student(Observer):
  def update(self,info):
    pass

class Stock(Subject):
  def __init__(self,name,price):
    super().__init__()
    self._name=name
    self._price=price

  @property
  def price(self):
    return self._price

  @price.setter
  def price(self,value):
    if self._price>value:
      info="Stock {}'s price decrease from {} to
{}".format(self._name,self._price,value)
      self.notifyAll(info)
    self._price=value

class Future(Subject):
  def __init__(self,name,volume):
    super().__init__()
    self._name=name
    self._volume=volume

  @property
  def volume(self):
    return self._volume

  @volume.setter
  def volume(self,value):
    if self._volume < value:
      info="Future {}'s volume grows from {} to
{}".format(self._name,self._volume,value)
      self.notifyAll(info)
    self._volume=value

def main():
  s=Stock("区块链",11.0)
  f=Future("猪肉",2000)
  for i in range(10):
    inv=Investor(str(i+1))
    s.addObserver(inv)
    f.addObserver(inv)
  s.price=8.0
  time.sleep(1)
```

```
    f.volume=3000

if __name__=='__main__':main()
```

适配器模式

- 适配器模式(Adapter)
  - 将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题
  - 目标类(Target)
    - 定义客户所需的接口，可以是一个抽象类或接口，也可以是具体类
  - 适配器类(Adapter)
    - 转换器，通过调用另一个接口对Adaptee和Target进行适配
  - 适配者类(Adaptee)
    - 被适配类，包括了客户希望的业务方法

- Demo
  - ad.py
- 适用场景
  - 没有现成的代码
  - 利用既有组件可能成本更低
  - 版本升级与兼容性
    - 新版本：Adaptee，旧版本：Target，Adapter类：实现旧版本类与新版本类的兼容

```
import abc
class Computer:
  def __init__(self,name):
    self._name=name

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self,value):
    self._name=value

  def __str__(self):
    return 'the {} computer'.format(self._name)

  def execute(self):
    print('execute a computer program')

class Human:
  def __init__(self,name):
    self._name=name
  def __str__(self):
    return '{} is an human'.format(self._name)
  def speak(self):
    print('hello word from human')

class Synthesizer:
  def __init__(self,name):
    self._name=name
```

```
  def __str__(self):
    return 'the {} synthesizer'.format(self._name)
  def play(self):
    print('play a synthesizer')

class Target(metaclass=abc.ABCMeta):
  @abc.abstractmethod
  def execute(self):
    pass

class HumanAdapter(Target):
  def __init__(self,human):
    self.human=human

  def execute(self):
    self.human.speak()

class SynthesizerAdapter(Target):
  def __init__(self,syn):
    self.syn=syn
  def execute(self):
    self.syn.play()

#另一种实现策略
class Adapter:
  def __init__(self,adp_obj,adp_methods):
    self.obj=adp_obj
    self.__dict__.update(adp_methods)

  def __str__(self):
    return str(self.obj)

def main():

  objects=[Computer('mac')]
  syn=Synthesizer('yamaha')
  h=Human('zjc')

  objects.append(HumanAdapter(h))
  objects.append(SynthesizerAdapter(syn))
  for obj in objects:
    obj.execute()


  objects2=[Computer('mac')]
  objects2.append(Adapter(syn,dict(execute=syn.play)))
  objects2.append(Adapter(h,dict(execute=h.speak)))
  for obj in objects2:
    obj.execute()
```

```
if __name__=='__main__':main()
```