1. **Python编程实现：单例模式**

单例模式  
 1.定义：单例模式（Singleton Pattern）是一种常用的软件设计模式，该模式的主要目的是确保某一个类只有一个实例存在。当你希望在整个系统中，某个类只能出现一个实例时，单例对象就能派上用场。  
 举例：比如，某个服务器程序的配置信息存放在一个文件中，客户端通过一个 AppConfig 的类来读取配置文件的信息。如果在程序运行期间，有很多地方都需要使用配置文件的内容，也就是说，很多地方都需要创建 AppConfig 对象的实例，这就导致系统中存在多个 AppConfig 的实例对象，而这样会严重浪费内存资源，尤其是在配置文件内容很多的情况下。事实上，类似 AppConfig 这样的类，我们希望在程序运行期间只存在一个实例对象。  
 我们先看看单例模式问题从何而来：  
class A(object):  
 def \_\_init\_\_(self):  
 pass  
 def foo(self):  
 pass  
  
a = A()  
print(id(a))  
b = A()  
print(id(b))  
 运行结果如下：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140534096496512  
 140534096497128  
 很明显，通过打印实例的id可以发现，A类默认被创建了两个实例a和b  
 那么，如何让类只去实例化一个对象，而后再创建的实例是返回上一次的对象的引用呢？  
 2.单例模式实现方式——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时，会生成 .pyc 文件，当第二次导入时，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。如果我们真的想要一个单例类，可以考虑这样做：   
创建一个单例模块mysingleton.py  
  
mysingleton.py  
class Singleton(object):  
 def \_\_init\_\_(self):  
 self.name = **'小安安'** def foo(self):  
 print(**'我是：%s'**%self.name)  
singleton = Singleton()  
  
  
from mysingleton import singleton  
singleton.foo()  
  
 3.单例模式实现方式——使用装饰器  
  
def singleton(cls,\*args,\*\*kwargs):  
 '''  
 使用装饰器的原理：  
 1.先创建外层函数，需要传入一个参数，此参数为类(对象)  
 2.创建一个空字典，用来保存单例  
 3.创建一个内层函数，用来获得单例  
 4.内层函数中进行判断：如果当前字典不存在单例，就创建单例，如果存在，直接返回该单例的引用  
 5.外层函数返回内层函数  
  
 '''  
 instances = {}  
 print(**'装饰器函数被调用！'**)  
 def get\_singleton(\*args,\*\*kwargs):  
 if cls not in instances:  
 instances[cls] = cls(\*args,\*\*kwargs)  
 print(**'装饰器内层函数也被调用！'**)  
 print(instances)  
 return instances[cls]  
 return get\_singleton  
  
@singleton  
#此处，相当于：Student = singleton(Student)  
class Student(object):  
 '''  
 创建单例的原理：  
 1.先由类实例化对象：xiaoanan = Student(30,'小安安')并传参，此时，因为前面已添加装饰器@singleton，Student此时相当于绑定到了get\_singleton函数，这句实例化对象的语句其实就是如下格式：xiaoanan = get\_singleton(30,'小安安'),而实例化对象时，这句话相当于调用了装饰器的内部函数get\_singleton，因为是第一次创建对象xiaoanan ,当前字典中并没有单例xiaoanan,所以内层函数get\_singleton中会执行if语句真值表达式为True时的语句，即：instances[cls] = cls(\*args,\*\*kwargs)，而这里的cls接收到的参数是Student，所以此句代码相当于：instances[Student] = Student(30,'小安安'),即给字典键名为Student的键添加一个值，这个值是Student类实例化的对象  
 2.当又一次实例化对象xiaochaochao时，因为仍然还是通过Student类实例化该对象，而之前的单例已经存在于字典中了，所以不会再创建第二个单例，只会直接返回已有的单例并绑定当前的引用，进而真正实现了单例模式！  
   
 '''  
 def \_\_init\_\_(self,age,name):  
 self.age = age  
 self.name = name  
 print(**'\_\_init\_\_()方法被调用了'**)  
xiaoan = Student(30,**'小安安'**)  
xiaochaochao = Student(18,**'小超超'**)  
print(id(xiaoan))  
print(id(xiaochaochao))  
  
 最终执行结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 装饰器函数被调用！  
 \_\_init\_\_()方法被调用了  
 装饰器内层函数也被调用！  
 {<class **'\_\_main\_\_.Student'**>: <\_\_main\_\_.Student object at 0x7fe08913f748>}  
 装饰器内层函数也被调用！  
 {<class **'\_\_main\_\_.Student'**>: <\_\_main\_\_.Student object at 0x7fe08913f748>}  
 140602349188936  
 140602349188936  
  
 此方式即实现了单例模式。  
  
  
  
 4.单例模式实现方式——使用类  
import threading  
import time  
  
class Singleton(object):  
 def \_\_init\_\_(self):  
 # time.sleep(1)  
 pass  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
# b = Singleton.instance()  
  
  
for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
 程序暂时先写成这样，我们先运行一下，看看结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 140291643101480  
 看上去，没毛病，恩，实现了单例，但其实！有毛病！此时看上去没问题是因为执行速度过快，如果我们让\_\_init\_\_方法存在一些IO操作，就会发现问题。  
 下面，我们用睡眠代替可能的IO操作看看会出现什么效果  
  
import threading  
import time  
  
class Singleton(object):  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
# b = Singleton.instance()  
  
  
for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
 此时，我们再看看程序运行的结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140107826004320  
 140107826107280  
 140107825398672  
 140107825883008  
 140107825392944  
 140107825882896  
 140107825398952  
 140107826107504  
 140107825835816  
 140107825881496  
  
 问题出现了！按照以上方式创建的单例，不支持多线程！  
 ok，如果非得采用这种方式来创建单例，解决办法是：加锁！  
 未加锁部分并发执行,加锁部分串行执行,速度降低,但是保证了数据安全  
 再看如下代码：  
  
import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def instance(cls,\*args,\*\*kwargs):  
 with Singleton.\_instance\_lock:  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = Singleton(\*args,\*\*kwargs)  
 return Singleton.\_instance  
def test(args):  
 a = Singleton.instance()  
 print(id(a))  
# b = Singleton.instance()  
  
  
for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
time.sleep(5)  
a = Singleton.instance()  
print(id(a))  
  
  
 此时，我们再看看程序运行的结果：  
 anwc@anwc:~/文档/课程资料$ python3 23+1种设计模式——Python实现.py  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
 140142050939176  
  
 到此，其实大致解决了我们的需求了，但是还是有一些小问题，就是当程序执行时，执行了time.sleep(20)后，下面实例化对象时，此时已经是单例模式了，但我们还是加了锁，这样不太好，再进行一些优化，把intance方法，改成下面的这样就行：  
  
  
import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
  
 @classmethod  
 def 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 test(args):  
 a = Singleton.instance()  
 print(id(a))  
# b = Singleton.instance()  
  
  
for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
time.sleep(5)  
a = Singleton.instance()  
print(id(a))  
  
 至此，支持多线程的单例实现结束，但这种方式还有弊端，不过不痛不痒了，很好发现：这种方式实现的单例，使用时会有限制，以后实例化对象必须通过调用类方法实现，即：  
 a = Singleton.instance()  
 而原本单单通过累的实例化，得到的就不是单例了。  
  
 5.单例模式实现方式——基于\_\_new\_\_方法实现  
 接上面的例子，已知，对于多线程情况下的单例，为了保证线程安全我们在内部加入了锁。  
 同时，再引入一个知识点。当我们实例化对象的时候，是先执行了\_\_new\_\_方法(如果不写，会默认调用Object.\_\_new\_\_,因为现版本中都是新式类，新式类默认继承自object)实例化对象，再调用\_\_init\_\_方法初始化实例化对象。基于以上，我们可以使用\_\_new\_\_方法实现单例模式  
import threading  
import time  
  
class Singleton(object):  
 \_instance\_lock = threading.Lock()  
 def \_\_init\_\_(self):  
 time.sleep(1)  
 def \_\_new\_\_(cls,\*args,\*\*kwargs):  
 if not hasattr(Singleton,**'\_instance'**):  
 with Singleton.\_instance\_lock:  
 if not hasattr(Singleton,**'\_instance'**):  
 Singleton.\_instance = object.\_\_new\_\_(cls)  
 return Singleton.\_instance  
  
# o1 = Singleton()  
# o2 = Singleton()  
# print(o1)  
# print(o2)  
  
def test(args):  
 o = Singleton()  
 print(id(o))  
  
for i in range(10):  
 t = threading.Thread(target=test,args=[i,])  
 t.start()  
  
 6.单例模式实现方式——基于metaclass实现  
 1.知识点补充  
 1.Python中的类也是一个对象  
 2.类是由元类来创建的  
 3.type是众多类的元类  
 4.类由type创建。创建类时，type的\_\_init\_\_()方法自动执行；类()，即由类实例化对象时，执行type的\_\_call\_\_()方法(其中包含\_\_new\_\_()和\_\_init\_\_()方法)  
 5.对象由类创建。创建对象时，类的\_\_init\_\_()方法自动调用，对象()时执行类的\_\_call\_\_()方法  
import threading  
  
class SingletonType(type):  
 \_instance\_lock = threading.Lock()  
 def \_\_call\_\_(cls, \*args, \*\*kwargs):  
 if not hasattr(cls, **"\_instance"**):  
 with SingletonType.\_instance\_lock:  
 if not hasattr(cls, **"\_instance"**):  
 cls.\_instance = super(SingletonType,cls).\_\_call\_\_(\*args, \*\*kwargs)  
 return cls.\_instance  
  
class Foo(metaclass=SingletonType):  
 def \_\_init\_\_(self,name):  
 self.name = name  
  
  
obj1 = Foo(**'name'**)  
obj2 = Foo(**'name'**)  
print(obj1,obj2)

1. **Python编程实现：简单工厂模式**

简单工厂模式实现：  
import abc  
class Leifeng:  
 '''定义雷锋抽象类，后续会有学生类和志愿者类继承该抽象类'''  
 # \_\_metaclass\_\_ = abc.ABCMeta  
 @abc.abstractmethod  
 def \_\_init\_\_(self):  
 pass  
 @abc.abstractmethod  
 def but\_rice(self):  
 pass  
 @abc.abstractmethod  
 def sweep(self):  
 pass  
  
class Student(Leifeng):  
 '''学生类，继承雷锋抽象类，重写两个抽象方法'''  
 def buy\_rice(self):  
 print(**'大学生帮你买大米'**)  
 def sweep(self):  
 print(**'大学生帮你扫地'**)  
  
class Volunteer(Leifeng):  
 '''志愿者类，继承雷锋类，重写两个抽象方法'''  
 def buy\_rice(self):  
 print(**'志愿者帮你买大米'**)  
 def sweep(self):  
 print(**'志愿者帮你扫地'**)  
  
class LeifengFactory:  
 '''创建雷锋工厂类，根据传入的参数类型返回相应对象'''  
 def create\_leifeng(self,type):  
 map\_ = {  
 **'大学生'**:Student(),  
 **'志愿者'**:Volunteer()  
 }  
 return map\_[type]  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 student1 = LeifengFactory().create\_leifeng(**'大学生'**)  
 # student2 = LeifengFactory().create\_leifeng('大学生')  
 vol1 = LeifengFactory().create\_leifeng(**'志愿者'**)  
 # vol2 = LeifengFactory().create\_leifeng('志愿者')  
 student1.buy\_rice()  
 student1.sweep()  
 vol1.buy\_rice()  
 vol1.sweep()

1. **Python编程实现：工厂方法模式**

工厂方法模式实现：  
import abc  
class Leifeng(object):  
 '''定义一个雷锋基类，后续大学生类和志愿者类都要继承该类'''  
 \_\_metaclass\_\_ = abc.ABCMeta  
 @abc.abstractmethod  
 def but\_rice(self):  
 pass  
 @abc.abstractmethod  
 def sweep(self):  
 pass  
  
class Student(Leifeng):  
 '''学生类，继承雷锋类，重写两个成员方法'''  
 def buy\_rice(self):  
 print(**'大学生帮你买大米'**)  
 def sweep(self):  
 print(**'大学生帮你扫地'**)  
  
class Volunteer(Leifeng):  
 '''志愿者类，继承雷锋类，重写两个成员方法'''  
 def buy\_rice(self):  
 print(**'志愿者帮你买大米'**)  
 def sweep(self):  
 print(**'志愿者帮你扫地'**)  
#以上为工厂类  
#以下为客户端(工厂方法类)  
class LeifengFactory:  
 '''定义雷锋工厂类，定义一个雷锋工厂方法'''  
 @abc.abstractmethod  
 def create\_leifeng(self):  
 pass  
  
class StudentFactory(LeifengFactory):  
 '''定义学生工厂类，继承雷锋工厂并重写雷锋工厂方法'''  
 def create\_leifeng(self):  
 return Student()  
  
class VolunteerFactory(LeifengFactory):  
 '''定义志愿者工厂类，继承雷锋工厂并重写雷锋工厂方法'''  
 def create\_leifeng(self):  
 return Volunteer()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 studentFactory = StudentFactory()#创建学生工厂对象  
 student = studentFactory.create\_leifeng()#通过学生工厂创建学生对象  
 student.buy\_rice()  
 student.sweep()  
  
 volunteerFactory = VolunteerFactory()#创建志愿者工厂对象  
 vol = volunteerFactory.create\_leifeng()#通过志愿者工厂创建志愿者对象  
 vol.buy\_rice()  
 vol.sweep()

1. **Python编程实现：抽象工厂模式**

import sys  
  
class User(object):  
 '''定义抽象user类，并定义抽象获取user数据方法，抽象插入user数据方法'''  
 def get\_user(self):  
 pass  
  
 def insert\_user(self):  
 pass  
  
class Department(object):  
 '''定义抽象department类，并定义抽象获取department数据方法，抽象插入department数据方法'''  
 def get\_department(self):  
 pass  
  
 def insert\_department(self):  
 pass  
#以上为抽象类  
#以下为具体操作数据库类  
class MysqlUser(User):  
 '''定义操作具体user数据库的类，继承抽象user类，采用mysql数据库实现'''  
 def get\_user(self):  
 print(**'Mysqluser get user'**)  
 def insert\_user(self):  
 print(**'Mysqluser insert user'**)  
  
class MysqlDepartment(Department):  
 '''定义操作具体department数据库的类，继承抽象department类，采用mysql数据库实现'''  
 def get\_department(self):  
 print(**'Mysqldepartment get department'**)  
 def insert\_department(self):  
 print(**'Mysqldepartment insert department'**)  
  
class OracleUser(User):  
 '''定义操作具体user数据库的类，继承抽象user类，采用Oracle数据库实现'''  
 def get\_user(self):  
 print(**'Oracleuser get user'**)  
 def insert\_user(self):  
 print(**'Oracleuser insert user'**)  
  
class OracleDepartment(Department):  
 '''定义操作具体department数据库的类，继承抽象department类，采用Oracle数据库实现'''  
 def get\_department(self):  
 print(**'Oracledepartment get department'**)  
 def insert\_department(self):  
 print(**'Oracledepartment insert department'**)  
  
#以下为抽象工厂类  
class AbstractFactory:  
 '''定义抽象工厂类及抽象方法，后续mysql工厂类级oracle工厂类会继承该类'''  
 def create\_user(self):  
 pass  
  
 def create\_department(self):  
 pass  
  
class MysqlFactory(AbstractFactory):  
 '''定义mysql工厂类，继承抽象工厂类'''  
 def create\_user(self):  
 return MysqlUser()  
  
 def create\_department(self):  
 return MysqlDepartment()  
  
  
class OracleFactory(AbstractFactory):  
 '''定义Oracle工厂类，继承抽象工厂类'''  
 def create\_user(self):  
 return OracleUser()  
  
 def create\_department(self):  
 return OracleDepartment()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 db = input(**'请输入数据库类型：'**)  
 myfactory = **''** if db == **'mysql'**:  
 myfactory = MysqlFactory()  
 elif db == **'oracle'**:  
 myfactory = OracleFactory()  
 else:  
 print(**'不支持的数据库类型'**)  
 sys.exit()  
  
 user = myfactory.create\_user()  
 department = myfactory.create\_department()  
 user.insert\_user()  
 user.get\_user()  
 department.insert\_department()  
 department.get\_department()

1. **Python编程实现：建造者模式**

class Builder(object):  
 '''定义建造者抽象类，定义抽象方法用于子类继承'''  
 def create\_footer(self):  
 pass  
  
 def create\_body(self):  
 pass  
  
 def create\_header(self):  
 pass  
  
class Thin(Builder):  
 '''定义瘦子类，继承抽象方法并重写'''  
 def create\_footer(self):  
 print(**'瘦子的脚创建了'**)  
  
 def create\_body(self):  
 print(**'瘦子的身体创建了'**)  
  
 def create\_header(self):  
 print(**'瘦子的头创建了'**)  
  
class Fat(Builder):  
 '''定义胖子类，继承抽象方法并重写'''  
 def create\_footer(self):  
 print(**'胖子的脚创建了'**)  
  
 def create\_body(self):  
 print(**'胖子的身体创建了'**)  
  
 def create\_header(self):  
 print(**'胖子的头创建了'**)  
  
class Director(object):  
 '''定义指挥者类，根据传入参数决定调用哪个子类的方法'''  
 def \_\_init\_\_(self,person):  
 self.person = person  
 def create\_person(self):  
 self.person.create\_footer()  
 self.person.create\_body()  
 self.person.create\_header()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 thin = Thin()  
 fat = Fat()  
 direct\_thin = Director(thin)  
 direct\_fat = Director(fat)  
 direct\_thin.create\_person()  
 direct\_fat.create\_person()

1. **Python编程实现：原型模式**

from copy import copy,deepcopy  
class Prototype(object):  
 '''创建原型抽象类，用于子类继承'''  
 def clone(self):  
 pass  
  
 def deep\_clone(self):  
 pass  
  
class WorkExp(object):  
 '''创建工作经验类，定义添加工作经验方法'''  
 def \_\_init\_\_(self):  
 self.timearea = **''** self.company = **''** def set\_workexp(self,timearea,company):  
 self.timearea = timearea  
 self.company = company  
  
class Resume(Prototype):  
 '''创建简历类，继承原型抽象类，重写抽象方法的同时添加部分属性'''  
 def \_\_init\_\_(self,name):  
 self.name = name  
 self.workexp = WorkExp()  
  
 def set\_personinfo(self,sex,age):  
 self.sex = sex  
 self.age = age  
  
 def set\_workexp(self,timearea,company):  
 self.workexp.set\_workexp(timearea,company)  
  
 def display(self):  
 print(self.name)  
 print(self.sex,self.age)  
 print(**'工作经历：%s,%s'**%(self.workexp.timearea,self.workexp.company))  
  
 def clone(self):  
 return copy(self)  
  
 def deep\_clone(self):  
 return deepcopy(self)  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 obj1 = Resume(**'安伟超'**)  
 obj2 = obj1.clone()  
 obj3 = obj1.deep\_clone()  
  
 obj1.set\_personinfo(**'男'**,30)  
 obj1.set\_workexp(**'2016-2018'**,**'达内时代科技集团天津天大中心'**)  
  
 obj2.set\_personinfo(**'男'**,32)  
 obj2.set\_workexp(**'2018-2020'**,**'达内时代科技集团天津学府中心'**)  
  
 obj3.set\_personinfo(**'男'**,34)  
 obj3.set\_workexp(**'2020-2022'**,**'达内时代科技集团天津长虹中心'**)  
  
 obj1.display()  
 obj2.display()  
 obj3.display()

1. **Python编程实现：责任链模式**

class Manager():  
 """经理类"""  
 successor = None  
 name = **''** def \_\_init\_\_(self, name):  
 self.name = name  
 def setSuccessor(self, successor):  
 # 设置上级  
 self.successor = successor  
 def handleRequest(self, request):  
 # 处理请求  
 pass  
  
class LineManager(Manager):  
 '''直属经理'''  
 def handleRequest(self, request):  
 if request.requestType == **'DaysOff'** and request.number <= 3:  
 return **'%s:%s Num:%d Accepted OVER'** % (self.name, request.requestContent, request.number)  
 else:  
 return **'%s:%s Num:%d Accepted CONTINUE'** % (self.name, request.requestContent, request.number)  
 if self.successor != None:  
 self.successor.handleRequest(request)  
  
class DepartmentManager(Manager):  
 '''部门经理'''  
 def handleRequest(self, request):  
 if request.requestType == **'DaysOff'** and request.number <= 7:  
 return **'%s:%s Num:%d Accepted OVER'** % (self.name, request.requestContent, request.number)  
 else:  
 return **'%s:%s Num:%d Accepted CONTINUE'** % (self.name, request.requestContent, request.number)  
 if self.successor != None:  
 self.successor.handleRequest(request)  
  
class GeneralManager(Manager):  
 '''总经理'''  
 def handleRequest(self, request):  
 if request.requestType == **'DaysOff'**:  
 return **'%s:%s Num:%d Accepted OVER'** % (self.name, request.requestContent, request.number)  
  
class Request():  
 def \_\_init\_\_(self, requestType ,requestContent, number = 0):  
 self.requestType = requestType  
 self.requestContent = requestContent  
 self.number = number  
  
 def commit(self,generalManager):  
 ret = generalManager.handleRequest(self)  
 print(ret)  
 return ret  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 line\_manager = LineManager(**'LINE MANAGER'**)  
 department\_manager = DepartmentManager(**'DEPARTMENT MANAGER'**)  
 general\_manager = GeneralManager(**'GENERAL MANAGER'**)  
  
 line\_manager.setSuccessor(department\_manager)  
 department\_manager.setSuccessor(general\_manager)  
  
 request = Request(requestContent=**"'Ask 1 day off'"**, requestType=**'DaysOff'**, number=1)  
 request.commit(line\_manager)  
  
 request = Request(requestContent=**"'Ask 5 day off'"**, requestType=**'DaysOff'**, number=5)  
 request.commit(line\_manager)  
  
 request = Request(requestContent=**"'Ask 10 day off'"**, requestType=**'DaysOff'**, number=10)  
 request.commit(line\_manager)

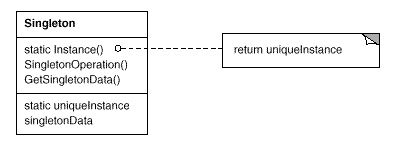
1. **列举23+1种设计模式**

设计模式分类：  
 设计模式分为基本的三种类型：  
 1.创建模式，提供实例化的方法，为适合的状况提供相应的对象创建方法。  
 2.结构化模式，通常用来处理实体之间的关系，使得这些实体能够更好地协同工作。  
 3.行为模式，用于在不同的实体建进行通信，为实体之间的通信提供更容易，更灵活的通信方法。  
 具体包含以下分类：  
 创建型  
 1. Factory Method（工厂方法）  
 2. Abstract Factory（抽象工厂）  
 3. Builder（建造者）  
 4. Prototype（原型）  
 5. Singleton（单例）  
 结构型  
 6. Adapter Class/Object（适配器）  
 7. Bridge（桥接）  
 8. Composite（组合）  
 9. Decorator（装饰）  
 10. Facade（外观）  
 11. Flyweight（享元）  
 12. Proxy（代理）  
 行为型  
 13. Interpreter（解释器）  
 14. Template Method（模板方法）  
 15. Chain of Responsibility（责任链）  
 16. Command（命令）  
 17. Iterator（迭代器）  
 18. Mediator（中介者）  
 19. Memento（备忘录）  
 20. Observer（观察者）  
 21. State（状态）  
 22. Strategy（策略）  
 23. Visitor（访问者）

1. **为什么编程时需要使用到设计模式？**

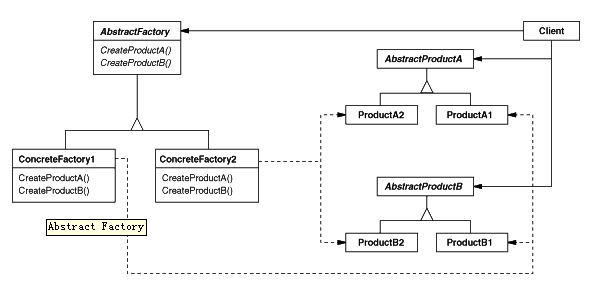
设计模式是经过总结、优化的，对我们经常会碰到的一些编程问题的可重用解决方案。  
一个设计模式并不像一个类或一个库那样能够直接作用于我们的代码。反之，设计模式更为高级，它是一种必须在特定情形下实现的一种方法模板。  
 设计模式不会绑定具体的编程语言。一个好的设计模式应该能够用大部分编程语言实现(如果做不到全部的话，具体取决于语言特性)。最为重要的是，  
设计模式也是一把双刃剑，如果设计模式被用在不恰当的情形下将会造成灾难，进而带来无穷的麻烦。  
 然而如果设计模式在正确的时间被用在正确地地方，它将是你的救星。起初，你会认为“模式”就是为了解决一类特定问题而特别想出来的明智之举。  
 说的没错，看起来的确是通过很多人一起工作，从不同的角度看待问题进而形成的一个最通用、最灵活的解决方案。  
 也许这些问题你曾经见过或是曾经解决过，但是你的解决方案很可能没有模式这么完备。虽然被称为“设计模式”，但是它们同“设计“领域并非紧密联系。设计模式同传统意义上的分析、设计与实现不同，事实上设计模式将一个完整的理念根植于程序中，所以它可能出现在分析阶段或是更高层的设计阶段。很有趣的是因为设计模式的具体体现是程序代码，因此可能会让你认为它不会在具体实现阶段之前出现(事实上在进入具体实现阶段之前你都没有意识到正在使用具体的设计模式)。  
 可以通过程序设计的基本概念来理解模式：增加一个抽象层。抽象一个事物就是隔离任何具体细节，这么做的目的是为了将那些不变的核心部分从其他细节中分离出来。当你发现你程序中的某些部分经常因为某些原因改动，而你不想让这些改动的部分引发其他部分的改动，这时候你就需要思考那些不会变动的设计方法了。这么做不仅会使代码可维护性更高，  
而且会让代码更易于理解，从而降低开发成本。

1. **举一个实例，此实例采用单例模式，画出其类图。**



class Singleton(object):  
 ''''' A python style singleton '''  
  
 def \_\_new\_\_(cls, \*args, \*\*kw):  
 if not hasattr(cls, **'\_instance'**):  
 org = super(Singleton, cls)  
 cls.\_instance = org.\_\_new\_\_(cls, \*args, \*\*kw)  
 return cls.\_instance  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 class SingleSpam(Singleton):  
 def \_\_init\_\_(self, s):  
 self.s = s  
  
 def \_\_str\_\_(self):  
 return self.s  
  
  
 s1 = SingleSpam(**'spam'**)  
 print  
 id(s1), s1  
 s2 = SingleSpam(**'spa'**)  
 print  
 id(s2), s2  
 print  
 id(s1), s1

1. **举一个实例，此实例采用抽象工厂模式，画出其类图。**

****

import sys  
  
class User(object):  
 '''定义抽象user类，并定义抽象获取user数据方法，抽象插入user数据方法'''  
 def get\_user(self):  
 pass  
  
 def insert\_user(self):  
 pass  
  
class Department(object):  
 '''定义抽象department类，并定义抽象获取department数据方法，抽象插入department数据方法'''  
 def get\_department(self):  
 pass  
  
 def insert\_department(self):  
 pass  
#以上为抽象类  
#以下为具体操作数据库类  
class MysqlUser(User):  
 '''定义操作具体user数据库的类，继承抽象user类，采用mysql数据库实现'''  
 def get\_user(self):  
 print(**'Mysqluser get user'**)  
 def insert\_user(self):  
 print(**'Mysqluser insert user'**)  
  
class MysqlDepartment(Department):  
 '''定义操作具体department数据库的类，继承抽象department类，采用mysql数据库实现'''  
 def get\_department(self):  
 print(**'Mysqldepartment get department'**)  
 def insert\_department(self):  
 print(**'Mysqldepartment insert department'**)  
  
class OracleUser(User):  
 '''定义操作具体user数据库的类，继承抽象user类，采用Oracle数据库实现'''  
 def get\_user(self):  
 print(**'Oracleuser get user'**)  
 def insert\_user(self):  
 print(**'Oracleuser insert user'**)  
  
class OracleDepartment(Department):  
 '''定义操作具体department数据库的类，继承抽象department类，采用Oracle数据库实现'''  
 def get\_department(self):  
 print(**'Oracledepartment get department'**)  
 def insert\_department(self):  
 print(**'Oracledepartment insert department'**)  
  
#以下为抽象工厂类  
class AbstractFactory:  
 '''定义抽象工厂类及抽象方法，后续mysql工厂类级oracle工厂类会继承该类'''  
 def create\_user(self):  
 pass  
  
 def create\_department(self):  
 pass  
  
class MysqlFactory(AbstractFactory):  
 '''定义mysql工厂类，继承抽象工厂类'''  
 def create\_user(self):  
 return MysqlUser()  
  
 def create\_department(self):  
 return MysqlDepartment()  
  
  
class OracleFactory(AbstractFactory):  
 '''定义Oracle工厂类，继承抽象工厂类'''  
 def create\_user(self):  
 return OracleUser()  
  
 def create\_department(self):  
 return OracleDepartment()  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 db = input(**'请输入数据库类型：'**)  
 myfactory = **''** if db == **'mysql'**:  
 myfactory = MysqlFactory()  
 elif db == **'oracle'**:  
 myfactory = OracleFactory()  
 else:  
 print(**'不支持的数据库类型'**)  
 sys.exit()  
  
 user = myfactory.create\_user()  
 department = myfactory.create\_department()  
 user.insert\_user()  
 user.get\_user()  
 department.insert\_department()  
 department.get\_department()