享元模式

# 1定义

运用共享技术有效地支持大量细粒度的对象。

# 2 内涵

一个类具有多个属性，将其中不变的属性（内部状态，独立于场景的信息，可以共享）提取出来，抽取为一个新的类，这个新类的属性组合就很有限，可以对新类的Flyweight对象进行共享。而其它受环境影响变化的属性（外部状态，根据场景而变化，不可共享），则在必要的时候由用户对象传递给Flyweight对象。

# 3 例子

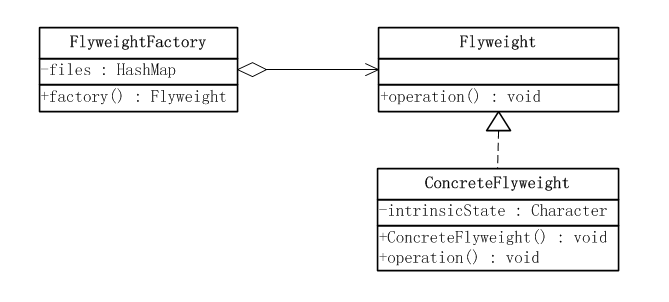
qq游戏中的五子棋、象棋等，不是每放一个棋子就new一个对象，而是存在更细粒度的对象。如五子棋的属性就包括哪个房间哪个棋盘的哪个位置的什么颜色，颜色值是固定黑色或者白色不变的，可以提取出黑色棋和白色棋两种棋作为享元实例。

咖啡的口味有限，将咖啡的口味作为享元对象提取出来。

考试系统的考试科目和场地是固定的那几种，抽取出来。

Java中对String的类型的存储

# 4 uml图



单纯享元模式所涉及到的角色如下：

　　●　　抽象享元(Flyweight)角色 ：给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

　　●　　具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

●　　享元工厂(FlyweightFactory)角色 ：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

# 5 代码

ICoffeeFlyweight

**public** **interface** **ICoffeeFlyweight** {

**public** void sell(Customer customer);

}

CoffeeFlyweightFactory

**public** **class** **CoffeeConcreteFlyweight** **implements** ICoffeeFlyweight {

**public** String flavor;

*// 获取咖啡口味*

**public** CoffeeConcreteFlyweight(String flavor) {

**this**.flavor = flavor;

}

**@Override**

**public** void sell(Customer customer) {

System.out.println("卖出给顾客"+customer.getName()+"一杯"+flavor);

}

}

CoffeeFlyweightFactory

**public** **class** **CoffeeFlyweightFactory** {

**private** **static** HashMap<String, CoffeeConcreteFlyweight> coffeePool = **new** HashMap<>();

**public** **static** CoffeeConcreteFlyweight getCoffee(String key){

CoffeeConcreteFlyweight result;

**if**(coffeePool.containsKey(key)){

result = coffeePool.get(key);

}**else** {

result = **new** CoffeeConcreteFlyweight(key);

coffeePool.put(key, result);

}

**return** result;

}

**public** **static** void getTotal(){

System.out.println("总共产生的coffee对象数量为"+coffeePool.size());

}

}

Customer

**public** **class** **Customer** {

**private** String name;

**public** Customer(String name) {

**this**.name = name;

}

**public** String getName(){

**return** name;

}

}

Client

**public** **class** **Client** {

**public** **static** void main(String[] args) {

initData();

CoffeeFlyweightFactory.getTotal();

}

**private** **static** void initData() {

**for**(int i=0;i<10;i++){

Customer customer = **new** Customer("小红"+i);

CoffeeConcreteFlyweight coffeeConcreteFlyweight = **null**;

**if**(i%3 ==0){

coffeeConcreteFlyweight = CoffeeFlyweightFactory.getCoffee("拿铁");

}**else** **if** (i%3 ==1) {

coffeeConcreteFlyweight = CoffeeFlyweightFactory.getCoffee("卡布奇诺");

}**else** {

coffeeConcreteFlyweight = CoffeeFlyweightFactory.getCoffee("摩卡");

}

coffeeConcreteFlyweight.sell(customer);

}

}

}

运行结果：

卖出给顾客小红0一杯拿铁

卖出给顾客小红1一杯卡布奇诺

卖出给顾客小红2一杯摩卡

卖出给顾客小红3一杯拿铁

卖出给顾客小红4一杯卡布奇诺

卖出给顾客小红5一杯摩卡

卖出给顾客小红6一杯拿铁

卖出给顾客小红7一杯卡布奇诺

卖出给顾客小红8一杯摩卡

卖出给顾客小红9一杯拿铁

总共产生的coffee对象数量为3

# 6 优缺点

享元模式是一个非常简单的模式，它可以大大减少应用程序创建的对象，降低程序内存的占用，增强程序的性能。但是享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。读取外部状态使得运行时间稍微变长。

# 7 使用场景

系统中存在大量相似的对象，有一些不变的属性可以提取出细粒度的对象。应用程序不依赖于对象标识。

# 8 注意事项

## 8.1 key值

尽量使用基本数据类型，性能高一些

## 8.2享元模式和对象池技术的区别

对象池中的所有对象是等价的，可以相互代替。而享元模式维护对象是不同的，不可以相互代替。

## 8.3 享元模式和单例模式的比较

目的都是一样，通过共享节省内存开销。享元设计模式是一个类有很多对象，而单例是一个类仅一个对象。享元模式可以说是单例模式的扩展。