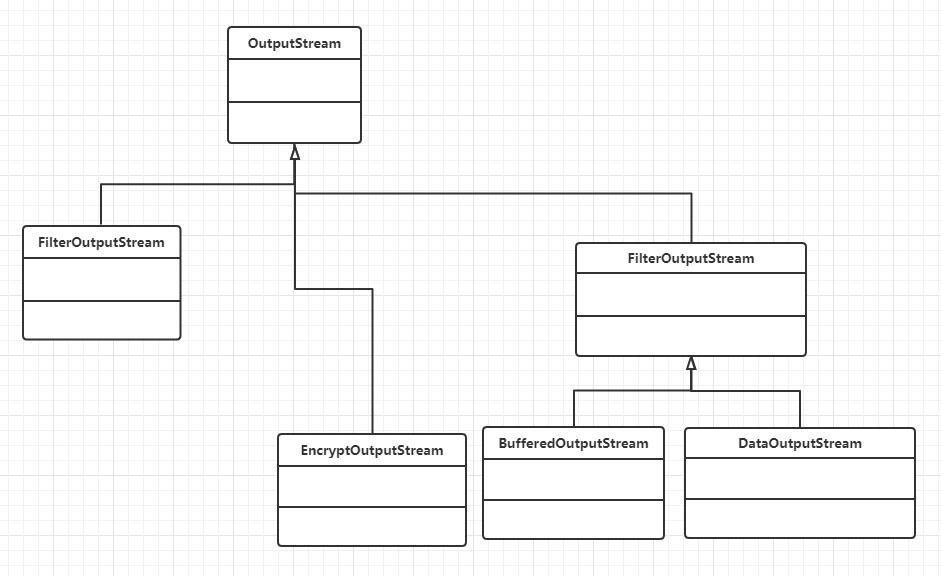
实现I/O流的装饰器

题目分析

Java中的I/O流部分是采用装饰模式实现的，如果我们想要添加新的功能，只需要实现新的装饰器，然后再使用时，组合进去就可以。以输出流为例实现装饰器

UML图

代码

**package** cokid1;  
  
**import** java.io.IOException;  
**import** java.io.OutputStream;  
  
**public class** EncryptOutputStream **extends** OutputStream{  
 **private** OutputStream **os** = **null**;*//装有被装饰的对象* **public** EncryptOutputStream(OutputStream os){  
 **this**.**os** = os;  
 }   
 *//实现把英文加密存放* **public void** write(**int** a) **throws** IOException {  
 a = a+2;*//先向后移动两位  
 //97是小写的a的ascii码值，如果大于97+26，表示已经是y或者z了，减去26就回到a或者b了* **if**(a >= (97+26)){  
 a = a-26;  
 }  
 **this**.**os**.write(a);  
 }  
}

**package** cokid1;  
  
**import** java.io.BufferedOutputStream;  
**import** java.io.DataOutputStream;  
**import** java.io.FileOutputStream;  
  
**public class** Client {  
 **public static void** main(String[] args) **throws** Exception {  
 *//流式输出文件* DataOutputStream dout = **new** DataOutputStream(  
 **new** BufferedOutputStream(  
 *//这是我们加的装饰器* **new** EncryptOutputStream(  
 **new** FileOutputStream(**"f://e.txt"**))));  
 *//然后就可以输出内容了* dout.write(**"abcdxyz"**.getBytes());  
 dout.close();  
 }  
}

总结

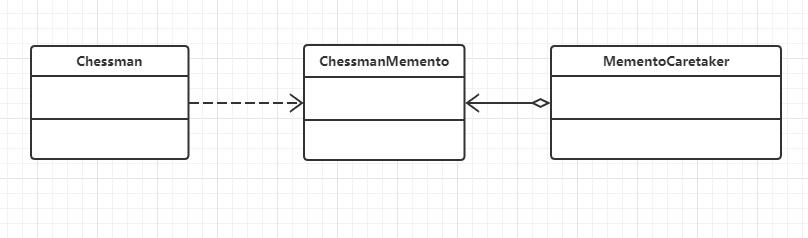
装饰模式比继承有更多的灵活性，可以让程序员定义多种不同行为的装饰器，进行组合，可以很方便地扩展装饰器，不过易产生大量的细粒度对象。

本例中BufferedOutputStream，DataOutputStream都继承于FilterOutputStream，而FilterOutputStream又继承于OutputStream，它们的构造函数都一个相同的点就是以OutputStream类型为参数，遇到的最大问题就是谁是被装饰的最终对象而谁是装饰器。本例中FilterOutputStream为被装饰的对象，因此定义自己的装饰器，只需要构造函数以OutputStream为参数即可，之后调用继承的方法write就会与包装顺序相反调用write方法进行写入操作。

备忘录模式实现-象棋中“悔棋”操作

题目分析

本题是最基本的备忘录模式的应用，“发起人”就是象棋本身，备忘录类记录每个象棋的信息包括象棋的棋子名称和棋子所在的位置，具体的象棋管理者对备忘录类进行操作。

UML图

代码

**package** cokid2;  
  
**public class** Chessman {  
 *//棋子名称* **private** String **label**;  
 **public** String getLabel() {  
 **return label**;  
 }  
 **public void** setLabel(String label) {  
 **this**.**label** = label;  
 }  
 *//x位置（第x行）* **private int x**;  
 **public int** getX() {  
 **return x**;  
 }  
 **public void** setX(**int** x) {  
 **this**.**x** = x;  
 }  
 *//y位置（第y行）* **private int y**;  
 **public int** getY() {  
 **return y**;  
 }  
 **public void** setY(**int** y) {  
 **this**.**y** = y;  
 }  
 **public void** show() {  
 System.***out***.println(**this**.**label** + **"当前位置： 第"** + **this**.**x** + **"行，第"** + **this**.**y** + **"列"**);  
 }  
 *//保存棋子状态* **public** ChessmanMemento save() {  
 **return** (**new** ChessmanMemento(**label**, **x**, **y**));  
 }  
 *//恢复棋子状态* **public void** restore(ChessmanMemento memento) {  
 **this**.**label** = memento.getLabel();  
 **this**.**x** = memento.getX();  
 **this**.**y** = memento.getY();  
 }  
 **public** Chessman(String label, **int** x, **int** y) {  
 **this**.**label** = label;  
 **this**.**x** = x;  
 **this**.**y** = y;  
 }  
}

**package** cokid2;  
  
**public class** ChessmanMemento {  
 **private** String **label**;*//棋子名称* **private int x**;*//x位置（第x行）* **private int y**;*//y位置（第y行）* **public** ChessmanMemento(String label, **int** x, **int** y) {  
 **this**.**label** = label;  
 **this**.**x** = x;  
 **this**.**y** = y;  
 }  
 **public** String getLabel() {  
 **return label**;  
 }  
 **public void** setLabel(String label) {  
 **this**.**label** = label;  
 }  
 **public int** getX() {  
 **return x**;  
 }  
 **public void** setX(**int** x) {  
 **this**.**x** = x;  
 }  
 **public int** getY() {  
 **return y**;  
 }  
 **public void** setY(**int** y) {  
 **this**.**y** = y;  
 }  
}

**package** cokid2;  
  
**public class** MementoCaretaker {  
 **private** ChessmanMemento **memento**;*//备忘录* **public** ChessmanMemento getMemento() {  
 **return memento**;  
 }  
 **public void** setMemento(ChessmanMemento memento) {  
 **this**.**memento** = memento;  
 }  
}

**package** cokid2;  
  
**import** java.util.Scanner;  
  
**public class** Client {  
 **public static void** main(String[] args) {  
 Scanner sc=**new** Scanner(System.***in***);  
 Chessman chess=**new** Chessman(**"兵"**,1,1);*//兵当前的位置* chess.show();  
 MementoCaretaker mc=**new** MementoCaretaker();  
 mc.setMemento(chess.save());*//保存兵当前的位置* System.***out***.println(**"兵走一步"**);  
 chess.setY(2);*//兵向前走一步的位置* chess.show();  
 System.***out***.println(**"是否要悔棋？ 1.是 2.否"**);  
 **int** panduan;  
 panduan=sc.nextInt();  
 **if**(panduan==1){  
 chess.restore(mc.getMemento());*//恢复兵之前的位置* }  
 chess.show();   
 }  
}

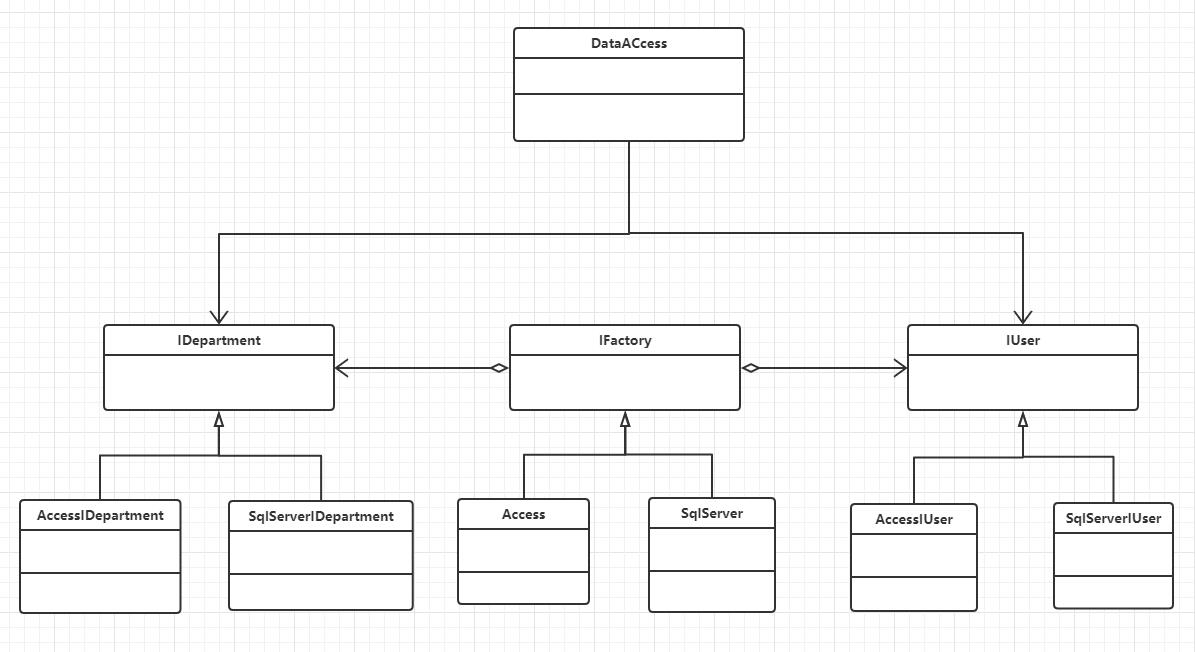
总结

备忘录模式可以把复杂的发起人的内部信息对其他人的对象隐蔽起来，从而可以恰当地保持封装的边界。 简化了发起人类。发起人不需要管理和保存内部状态的一个个版本，客户端自行管理。使用暂时存储的备忘录进行状态复原。但是资源消耗上面备忘录对象可能会很大，无法提醒用户一个操作是否很大。编程的时候还要考虑周全要记录哪些内容，保证不多不少，少了可能无法恢复到先前状态，过多会消耗更多系统资源。

用反射+配置文件的方法实现数据库连接

题目分析

用反射主要是解决程序中大量出现的分支语句，运用反射可以根据用户输入来动态生成具体要用的数据库源类。用配置文件主要是解决一旦更换数据库源就与要更改源代码的问题，将要用的不同数据库抽取成配置文件，以后修改只需要修改配置文件就可以。

UML图

代码

**package** cokid3;  
  
**public interface** IDepartment {  
 **public void** insert();  
 **public void** update();  
}

**package** cokid3;  
  
**public interface** IFactory {  
 **public** IUser CreateIUSer();  
 **public** IDepartment CreateIDepartment();  
}

**package** cokid3;  
  
**public interface** IUser {  
 **public void** insert();  
 **public void** delete();  
}

**package** cokid3;  
  
**public class** AccessIDepartment **implements** IDepartment {  
  
 @Override  
 **public void** insert() {  
 System.***out***.println(**"调用了Access插入语句"**);  
 }  
 @Override  
 **public void** update() {  
 System.***out***.println(**"调用了Access删除语句"**);  
 }  
}

**package** cokid3;  
  
**public class** SqlServerIDepartment **implements** IDepartment {  
  
 @Override  
 **public void** insert() {  
 System.***out***.println(**"调用了sqlserver插入语句"**);  
 }  
  
 @Override  
 **public void** update() {  
 System.***out***.println(**"调用了sqlserver删除语句"**);  
 }  
  
}

**package** cokid3;  
  
  
**public class** Access **implements** IFactory {  
  
 @Override  
 **public** IUser CreateIUSer() {  
 **return new** AccesslUser();  
 }  
  
 @Override  
 **public** IDepartment CreateIDepartment() {  
 **return new** AccessIDepartment();  
 }  
  
}

**package** cokid3;  
  
**public class** SqlServer **implements** IFactory {  
  
 @Override  
 **public** IUser CreateIUSer() {  
 **return new** SqlServerIUser();  
 }  
  
 @Override  
 **public** IDepartment CreateIDepartment() {  
 **return new** SqlServerIDepartment();  
 }  
  
}

**package** cokid3;  
  
**public class** AccesslUser **implements** IUser {  
  
 @Override  
 **public void** insert() {  
 System.***out***.println(**"调用了Access插入语句"**);  
 }  
  
 @Override  
 **public void** delete() {  
 System.***out***.println(**"调用了Access删除语句"**);  
 }  
  
}

**package** cokid3;  
  
  
**public class** SqlServerIUser **implements** IUser {  
  
 @Override  
 **public void** insert() {  
 System.***out***.println(**"调用了sqlserver插入语句"**);  
 }  
  
 @Override  
 **public void** delete() {  
 System.***out***.println(**"调用了sqlserver删除语句"**);  
 }  
  
}

**package** cokid3;  
  
**import** java.io.File;  
**import** java.util.HashMap;  
**import** java.util.List;  
**import** java.util.Map;  
  
**import** org.jdom.Document;  
**import** org.jdom.Element;  
**import** org.jdom.input.SAXBuilder;  
  
**public class** DataAccess {  
 **private static** Map<String,String> *databases*=**new** HashMap<String,String>();  
   
 **static**{  
 String path=System.*getProperty*(**"user.dir"**)+**"/bin/Factory.xml"**;  
 File file= **new** File(path);  
 SAXBuilder builder = **new** SAXBuilder();  
 **try** {  
 Document document = builder.build(file);  
 Element root = document.getRootElement();  
 List<Element> factories = root.getChildren(**"factory"**);  
 **for**(**int** i=0;i<factories.size();i++){  
 Element factory = factories.get(i);  
 String value=factory.getAttributeValue(**"value"**);  
 String clazz=factory.getChildText(**"class"**);  
 *databases*.put(value, clazz);  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 **public static** Map<String, String> getDatabases() {  
 **return** *databases*;  
 }  
  
 **public static void** setDatabases(Map<String, String> databases) {  
 DataAccess.*databases* = databases;  
 }  
   
 **public void** ShowAllDataBase(){  
 System.***out***.println(**"当前支持数据库有："**);  
 **int** i =1;  
 **for**(Map.Entry<String , String >me:*databases*.entrySet()){  
 System.***out***.println(i+**"、"**+me.getKey());  
 i++;  
 }  
 System.***out***.println(**"请输入："**);  
 }  
 **public** IFactory getDatabase(**int** i){  
 **int** j=1;  
 String clazz=**null**;  
 **for**(Map.Entry<String , String >me:*databases*.entrySet()){  
 **if**(j==i){  
 clazz=me.getValue();  
 **break**;  
 }  
 j++;  
 }  
 IFactory factory=**null**;  
 **try** {  
 factory = (IFactory) Class.*forName*(clazz).newInstance();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** factory;  
 }  
}

**package** cokid3;  
  
**import** java.util.Scanner;  
  
  
**public class** Client {  
 **public static void** main(String[] args) {  
 Scanner sc=**new** Scanner(System.***in***);  
 **int** n;  
 DataAccess dataFactory = **new** DataAccess();  
 dataFactory.ShowAllDataBase();  
 n=sc.nextInt();  
 IFactory factory=dataFactory.getDatabase(n);  
 System.***out***.println(factory.toString());  
 }  
}

总结

抽象工厂模式虽然代码量相对复杂一些，但是结构性却很明确。它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户端代码中。再结合反射于配置文件，可以让代码更容易扩展。只要把握好接口和类之间的逻辑关系，运用抽象工厂模式不是很困难。