1. 从招式与内功谈起——设计模式概述

模式是在特定环境下人们解决某类重复出现问题的一套成功或有效的解决方案。

软件模式是指在软件开发过程中某些可重现问题的有效解决方法，包含架构模式、分析模式、过程模式和设计模式等。

设计模式就是代码设计经验总结，使用设计模式的原因包括编写可重用的代码和提高代码的可靠性等。

设计模式可以分为三种：

1. 创建型
2. 结构型
3. 行为型

滥用模式还不如不用模式

1. 预备知识——UML类图与面向对象设计原则

UML类图用于描述每一个设计模式的结构

常用的面向对象设计原则包括7个，分别是单一职责原则、开闭原则、里氏替换原则、依赖倒转原则、借口隔离原则、合成复用原则和迪米特法则。

UML是一种由图形符号表达的建模语言，其结构主要包括以下几个部分：

1. 视图
   1. UML视图包括用户视图、结构视图、行为视图、实现视图和环境视图
2. 图
   1. 最新的UML2.0提供了13中图，分别是用例图、类图、对象图、包图、组合结构图、状态图、活动图、顺序图、通信图、定时图、交互概览图、组件图和部署图。
3. 模型元素
4. 通用机制

类之间的关系：

1. 关联关系
   1. 在UML类图中，用实线连接有关联关系对象所对应的类。

面向对象的设计原则之一在于支持可维护性复用：一方面需要实现设计方案或者源代码的重用；另一方面需要确保系统能够易于扩展和修改，具有较好的灵活性。

最常见的7种面向对象设计原则：

|  |  |
| --- | --- |
| 原则名称 | 描述 |
| 单一职责原则 | 一个类只负责和一个功能领域中的相应职责 |
| 开闭原则 | 软件实体应对扩展开放，而对修改关闭 |
| 里氏替换原则 | 所有引用基类对象的地方能够透明的使用其子类对象 |
| 依赖倒置原则 | 抽象不应该依赖于细节，细节应该以抽象 |
| 接口隔离原则 | 使用多个专门的接口而不使用单一的总接口 |
| 合成复用原则 | 尽量使用对象组合而不是继承来达到复用的目的 |
| 迪米特法则 | 一个软件实体应当尽可能少的与其他实体发生相互作用 |

1. 单一职责原则：

用于控制类的粒度的大小

实现高内聚低耦合的直到方针

1. 开闭原则

一个软件对象的实体应该对扩展开放对修改关闭，即软件实体应该在不修改原有代码的情况下进行扩展。

1. 里氏替换原则

通俗的说就是所有引用基类（父类）的地方必须能透明的使用其子类的对象。

里氏替换原则表明，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何的错误和异常（这里不讨论功能）。

这一原则要求尽量使用基类类型对对象进行定义（me：实际就是尽量使用多态，也即面向接口编程），将父类设计为抽象类或者接口。

1. 依赖倒置原则

该原则指的是抽象不应该依赖于细节，细节应该依赖于抽象，即要针对接口编程，而不是针对实现编程。

该原则要求尽量使用层次较高的抽象层类，进行变量声明、参数类型声明、方法返回类型声明和数据类型的转换等。为了能够使用父类或者接口调用实现类的方法，具体的实现类中不要定义其他的非实现自接口或者继承自基类的公开方法。

常用的恶依赖注入方式有三种：

构造注入、设置注入（setter方法）和接口注入（指的是通过实现在接口中声明的业务方法来传入具体的对象）。这些方法在使用时传入的都是抽象类型，实际调用时传递具体的类型。

5． 接口隔离原则

客户端不应该依赖那些它不需要的接口

使用多个专门的接口而不是一个总的大的接口。

接口仅仅通过给客户端（client）它需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

接口应该尽量细化，每个接口中的方法应该尽量少。

1. 合成复用原则

尽量使用对象服务而不是继承来达到复用的目的。

继承复用会破坏系统的封装性

1. 迪米特法则

又称为最小只是原则，一个软件实体应该可能少的与其他实体发生作用

第2 部分 创建的艺术——创建型模式

单例模式 用于创建那些在软件系统中独一无二的对象。

浪费资源，包括CPU资源和内存资源

实现步骤：

1. 私有化构造器

可见性只对类外有效，在类的内部任何都会可见的

1. 私有的类实例字段/域
2. 公有的静态方法返回该字段/域

单例的类别：饿汉式和懒汉式，实现代码如下：

/\*\*  
 \* 设计步骤：  
 \* 1. 私有化构造器  
 \* 2. 私有的该类的静态成员变量  
 \* 3. 公共的外部访问该成员变量的静态方法  
 \*  
 \* 两种构造方式：  
 \* 1. 懒汉式  
 \* 2. 饿汉式  
 \*/  
  
/\*\*  
 \* 懒汉式  
 \*/  
public class Singleton {  
 private volatile static Singleton instance;  
  
 private Singleton(){}  
  
 public static Singleton getInstance(){  
 if(null==instance){  
 synchronized (Singleton.class){  
 if(null==instance){  
 instance=new Singleton();  
 }  
 }  
 }  
  
 return instance;  
 }  
}  
  
/\*\*  
 \* 饿汉式  
 \*/  
class Singleton2{  
 private static final Singleton2 instance=new Singleton2();  
 private Singleton2(){}  
  
 public static Singleton2 getInstance(){  
 return instance;  
 }  
}

1. 懒汉式采用延迟加载技术，创建时需要使用双重检查锁定。
2. 懒汉式的静态对象需要加上volatile修饰，以防止编译器的指令排序，参考：<http://blog.csdn.net/jm_heiyeqishi/article/details/51052889>

饿汉式vs懒汉式：

1. 饿汉式在类加载的时候就会被创建，不用考虑多线程的问题，而懒汉式延迟加载，需要考虑线程安全的问题，虽然解决了线程安全的问题但是付出了性能的代价。
2. 饿汉式能够由于创建在类加载的时候，调用速度和反应时间较懒汉式更好。
3. 但是，无论系统是否真的使用了单例对象，饿汉式都创建，因此在资源利用效率的角度讲懒汉式较好。

一种更好的实现方法使用了IoDH技术，即使用了静态内部类：

public class BetterSingleton {  
  
 private BetterSingleton(){}  
  
 public static BetterSingleton getInstance(){  
 return HolderClass.instance;  
 }  
  
 private static class HolderClass{  
 private static final BetterSingleton instance=new BetterSingleton();  
 }  
}

这样实现，在外部类被加载时不会实例化内部类的对象，当第一次调用getInstance时将加载内部类HolderClass，由jvm来控制线程安全（即静态类是不会改变的）。

总结：

* 主要优点：

1. 全局唯一实例，受控访问
2. 节约系统资源
3. 在单例模式上扩展，可以实现多例模式

* 主要缺点：

1. 单例类很难扩展（指的是被继承）
2. 长时间不用可能被垃圾回收器回收，从而再次使用时重新创建对象导致单例的状态不能够延续（me：其他的对象不也是这样吗？严格的讲这个不是确点）

* 适用场景

系统只需要一个实例对象

* 其他类似实现和扩展：

1. 多例模式、数据库连接池等

通过在对象中维护多个实例来实现

集中式工厂的实现——简单工厂模式

该模式不在GoF23中设计模式，但在软件开发中频繁使用。

工厂模式一共有三种：

1. 简单工厂模式
2. 工厂方法模式
3. 抽象工厂模式

涉及类：

1. 抽象类
2. 具体类
3. 工厂类

定义：定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类，因为在简单工厂模式中用于创建实例的方法是静态的，所以简单工厂模式又被称为静态工厂方法模式。

它的要点在于当你需要什么，只需要传入一个正确的参数就可以获取你所需要的对象而无须直到其创建细节。

abstract class AbstractProduct {  
 abstract String price();  
}  
  
class ProductA extends AbstractProduct {  
 private String name;  
 public ProductA(String name){  
 this.name=name;  
 }  
  
 @Override  
 String price() {  
 return "A-price";  
 }  
}  
  
class ProductB extends AbstractProduct {  
 private String name;  
 public ProductB(String name){  
 this.name=name;  
 }  
  
 @Override  
 String price() {  
 return "B-price";  
 }  
}  
  
public class SimpleFactory {  
 public static AbstractProduct create(String arg) throws Exception {  
 AbstractProduct product = null;  
 if (arg.equals("A")) {  
 product = new ProductA("A");  
 } else if (arg.equals("B")) {  
 product = new ProductB("B");  
 } else {  
 throw new Exception("illegal arg");  
 }  
  
 return product;  
 }  
}

简单工厂的核心是工厂类，它提供了一个静态的方法供给client端使用，根据所传入的参数的不同可以创建不同的对象，这些对象继承自一个基类或抽象类或者实现同一接口。

client使用：

AbstractProduct product\_1 = SimpleFactory.create("A");  
AbstractProduct product\_2 = SimpleFactory.create("B");

以上实现存在问题：

1. 每次创建的对象类型固定写在了client端，修改时需要更改代码重新编译，不符而开闭原则，可以考虑将创建的字符串参数存储在其他非程序代码中，如配置文件、数据库等。
2. 如果新增或者删除产品类别时仍然需要修改工厂类中的静态方法，违背了开闭原则

讨论：

在java语言中，通常有以下几种创建对象的方式：

1. 使用new关键字
2. 通过反射机制创建对象
3. 通过clone方法创建对象
4. 通过工厂类创建对象

创建对象与使用对象：

与一个对象相关的职责通常有三类：

* 1. 对象本身具有的职责
  2. 创建对象的职责
  3. 使用对象的职责

创建对象和使用对象的职责耦合在一起，这样的设计会导致很严重的问题（但使用对象一方希望使用其他的对象时就需要修改源代码，违背了开闭原则）

解决办法是引入工厂类，工厂类负责创建对象而不使用，其他的类使用对象但不创建。这降低了耦合性。

所有的工厂模式都强调一点：两个类A和B之间的关系应该仅仅是A创建B或者是A使用B（或者反过来），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。

即在使用对象和对象之间添加一层工厂类来封装对象的创建逻辑和客户代码的实例化/配置选项进而实现对象的分离创建和使用。

工厂类还有另一个不明显的好处是可以为对象的特定创建指定不同的易区分的名字（而不是只有一个创建对象的名字或者根据参数来猜测特定的对象-因为类的构造方法的名字是相同的），如创建长方形和正方形，对应的是一个类则构造方法的名字相同，但是我们可以在工厂类中添加两个方法调用不同的构造函数。

使用工厂类要注意避免工厂泛滥，这会增加系统维护的难度。实际开发中要具体问题具体分析。

简单工厂模式的简化：有时为了简化简单工厂模式，可以将抽象产品类和工厂类合并，将静态工厂方法移至抽象产品类中。代码如下：

abstract class AbstractProduct {  
 abstract String price();  
  
 /\*\*  
 \* 为了简化，也可以将工厂和抽象类合并  
 \* @param arg  
 \* @return  
 \* @throws Exception  
 \*/  
 public static AbstractProduct create(String arg) throws Exception {  
 AbstractProduct product = null;  
 if (arg.equals("A")) {  
 product = new ProductA("A");  
 } else if (arg.equals("B")) {  
 product = new ProductB("B");  
 } else {  
 throw new Exception("illegal arg");  
 }  
  
 return product;  
 }  
}

总结：

* 主要优点

1. 工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类实例，客户端可以免除直接创建产品对象的职责，而仅仅消费产品（即使用产品的功能）。它实现了对象创建和使用的分离
2. 客户端无需知道所创建的具体产品的类名，只需要知道具体产品类所对应的参数即可。甚至可以在工厂类中定义形象的名称
3. 引入配置文件，在一定程度上追求开闭原则。

* 主要缺点

1. 增加工厂类，可能增加了系统的复杂性和理解难度（但是系统的灵活性提高了）

* 使用场景

1. 工厂类创建对象比较少，多的话会增加系统的复杂性
2. 客户端对如何创建对象并不关心，只知道与对象联系的参数

实例：使用简单工厂模式设计一个可以创建不同集合形状（例如圆形、方形和三角形等）的绘图工具，每个几何图形都具有绘制draw和擦除erase方法，要求在绘制不支持的图形时，提示一个UnSupportedShapeException。

这里将工厂类和抽象类合并：

abstract class AbstractShape {  
 abstract void draw();  
 abstract void erase();  
  
 public static AbstractShape create(ShapeEnum shapeEnum)throws UnSupportedShapeException{  
 AbstractShape shape=null;  
 switch(shapeEnum){  
 case Tiangle:  
 shape=new Triangle();  
 break;  
 case Rectangle:  
 shape=new Rectangle();  
 break;  
 case Circular:  
 shape=new Circular();  
 break;  
 default:  
 throw new UnSupportedShapeException();  
 }  
  
 return shape;  
 }  
}  
  
class Triangle extends AbstractShape {  
 @Override  
 void draw() {  
 System.out.println("draw a triangle");  
 }  
  
 @Override  
 void erase() {  
 System.out.println("erase a triangle");  
 }  
}  
  
class Rectangle extends AbstractShape {  
 @Override  
 void draw() {  
 System.out.println("draw a rectangle");  
 }  
  
 @Override  
 void erase() {  
 System.out.println("erase a rectangle");  
 }  
}  
  
class Circular extends AbstractShape {  
 @Override  
 void draw() {  
 System.out.println("draw a circular");  
 }  
  
 @Override  
 void erase() {  
 System.out.println("erase a circular");  
 }  
}  
  
class UnSupportedShapeException extends Exception{  
 public UnSupportedShapeException(){}  
 public UnSupportedShapeException(String msg){super(msg);}  
}  
  
enum ShapeEnum{  
 Tiangle(1),  
 Rectangle(2),  
 Circular(3);  
  
 private int id;  
 ShapeEnum(int id){this.id=id;}  
  
 public int getId() {  
 return id;  
 }  
}  
  
class ShapeTest{  
 public static void main(String[] args) throws UnSupportedShapeException{  
 AbstractShape shape1= AbstractShape.create(ShapeEnum.Rectangle);  
 AbstractShape shape2= AbstractShape.create(ShapeEnum.Circular);  
  
 shape1.draw();  
 shape2.draw();  
 }  
}

第五章 多态工厂的实现——工厂方法模式

简单工厂操作面临的一个问题是：当系统增加新的产品类型时不得不修改源代码，这就违背了开闭原则。而工厂方法模式可以解决这一问题。

工厂方法模式：定义一个用于创建对象的接口，让子类决定将哪个类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为工厂模式，又可称作虚拟构造器模式或多态工厂模式。工厂方法模式是一种类创建型模式。

工厂方法包含四个角色：

1. 抽象产品：产品接口或者抽象类
2. 具体产品
3. 抽象工厂：工厂接口
4. 具体工厂

数据库日志和文件日志实例：

public class LoggerDemo {  
}  
  
interface ILogger{  
 void log(String msg);  
}  
  
class FileLogger implements ILogger{  
 public void log(String msg) {  
 //log to file  
 }  
}  
  
class DataBaseLogger implements ILogger{  
 public void log(String msg) {  
 //log to db  
 }  
}  
  
interface ILoggerFactory{  
 ILogger createLogger();  
}  
  
class FileLoggerFactory implements ILoggerFactory{  
 public ILogger createLogger() {  
 return new FileLogger();  
 }  
}  
  
class DataBaseLoggerFactory implements ILoggerFactory{  
 public ILogger createLogger() {  
 return new DataBaseLogger();  
 }  
}  
  
class LoggerTest{  
 public static void main(String[] args) {  
 ILogger logger=new FileLoggerFactory().createLogger();  
 logger.log("logging ...");  
 }  
}

这样实现，如果需要添加新的日志记录类，只需要在原有的基础上扩展而不需要修改原有的代码。

工厂方法的改进：使用配置文件的非代码的配置项来决定日志的种类而非使用new关键字，从而实现灵活扩展和更换。

class Client{  
 public static ILogger getLogger() {  
 ILogger iLogger=null;  
 //klassName from xml or db  
 String klassName=null;  
 try{  
 Class kclass=Class.forName(klassName);  
 iLogger=((ILoggerFactory) kclass.newInstance()).createLogger();  
 }catch(ClassNotFoundException ex){  
 ex.printStackTrace();  
 }catch(IllegalAccessException ex){  
 ex.fillInStackTrace();  
 }catch(InstantiationException ex){  
 ex.fillInStackTrace();  
 }  
  
 return iLogger;  
 }  
  
 //other code  
}

这时增加新的日志记录时与之前的工厂方法一样，创建实现ILogger的类和实现ILoggerFactory的工厂类；

当更换日志记录方式的时候只需要更改配置文件即可。

符合开闭原则

重载工厂方法：

与其他重载方法的功能一样，即通过传递不同的参数定制产品对象。如可以传递日志文件的路径，日志的记录级别等。当然，如果什么都不传入的话使用默认的配置。

工厂方法的隐藏：即将具体的产品的业务方法放入对对应的工厂中，client直接调用工厂中的业务方法，这时的工厂不止创建产品对象，也调用业务方法。

因为在抽象工厂中包含了具体的方法，所以需要使用抽象类来作为抽象工厂，代码如下：

abstract class AbstractLoggerFactory{  
 public void log(String msg){  
 //log  
 ILogger logger=this.createLogger();  
 logger.log(msg);  
 }  
  
 abstract ILogger createLogger();  
}  
  
class FileLoggerFactory2 extends AbstractLoggerFactory{  
 @Override  
 ILogger createLogger() {  
 return new FileLogger();  
 }  
}  
  
class Client2 {  
 public static void main(String[] args) {  
 //from config .etc  
 String klassName = null;  
 try {  
 Object loggerFactory = (Class.forName(klassName)).newInstance();  
 ((AbstractLoggerFactory)loggerFactory).log("logging ...");  
 } catch (ClassNotFoundException ex) {  
 ex.printStackTrace();  
 } catch (InstantiationException ex) {  
 ex.printStackTrace();  
 }catch(IllegalAccessException ex){  
 ex.fillInStackTrace();  
 }  
 }  
}

总结：

* 主要优点

基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。工厂如何创建产品对象完全封装在工厂的内部。

在系统中加入新的产品时，不需要修改原工厂和产品接口，只需要实现产品接口和添加实现工厂接口的具体工厂即可。

系统的扩展性也非常的好，完全符合开闭原则。

* 主要缺点

增加了抽象层，使得系统复杂程度增加，理解难度增加（但是系统的灵活性提高，具有更好的可维护性）

* 适用场景

对系统扩展性要求高的地方

第六章 产品族的创建——抽象工厂模式

抽象工厂模式的基本思想：将一些相关的产品组成一个产品族，有同一个工厂来统一生产。

在工厂方法中，一般情况下，一个具体工厂中只有一个或者一组重载的工厂方法。

产品族：在抽象工厂模式中，产品族是指由一个工厂生产的，位于不同产品等级结构（即不同类产品）中的一组产品。

当系统所提供的工厂生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构、属于不同类型的具体产品时，就可以使用抽象工厂模式。

抽象工厂模式为创建一组对象提供了一种解决方案。与工厂方法模式相比，抽象工厂中的具体工厂不只创建一种产品，它负责创建一族产品。

抽象工厂模式定义：提供一个创建一系列相关或者相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为kit模式，它是一种对象创建型模式。

在抽象工厂模式中，每一个具体工厂提供了多个工厂方法用于产生多种不同类型的产品，这些产品构成了一个产品族，结构定义如下：

1. AbstractFactory-抽象工厂
2. ConcreteFactory-具体工厂
3. AbstractProduct-抽象产品
4. ConcreteProduct-具体产品

抽象工厂可以是接口、抽象类或者具体类。具体工厂实现了抽闲工厂，每个具体的工厂方法可以返回一个特定的产品对象，而同一个具体工厂所创建的产品对象构成了一个产品族。

抽象可以使用接口，也可以使用抽象类。

interface Button{  
 void display();  
}  
  
class SpringButton implements Button{  
 public void display() {  
 System.out.println("spring button");  
 }  
}  
  
class SummerButton implements Button{  
 public void display() {  
 System.out.println("summer button");  
 }  
}  
  
interface TextField{  
 void display();  
}  
  
class SpringTextField implements TextField{  
 public void display() {  
 System.out.println("spring text field");  
 }  
}  
  
class SummerTextField implements TextField{  
 public void display() {  
 System.out.println("summer text field");  
 }  
}  
  
interface ComboBox{  
 void display();  
}  
  
class SpringComboBox implements ComboBox{  
 public void display() {  
 System.out.println("spring combo box");  
 }  
}  
  
class SummerComboBox implements ComboBox{  
 public void display() {  
 System.out.println("summer combo box");  
 }  
}  
  
interface SkinFactory{  
 Button createButton();  
 TextField createTextField();  
 ComboBox createComboBox();  
}  
  
class SpringSkinFactory implements SkinFactory{  
 public Button createButton() {  
 return new SpringButton();  
 }  
  
 public TextField createTextField() {  
 return new SpringTextField();  
 }  
  
 public ComboBox createComboBox() {  
 return new SpringComboBox();  
 }  
}  
  
class SummerSkinFactory implements SkinFactory{  
 public Button createButton() {  
 return new SummerButton();  
 }  
  
 public TextField createTextField() {  
 return new SummerTextField();  
 }  
  
 public ComboBox createComboBox() {  
 return new SummerComboBox();  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 SkinFactory factory=new SpringSkinFactory();  
 Button button=factory.createButton();  
 TextField textField=factory.createTextField();  
 ComboBox comboBox=factory.createComboBox();  
  
 button.display();  
 textField.display();  
 comboBox.display();  
 }  
}

这样开发可以灵活的扩展，符合开闭原则。

抽象工厂模式存在开闭原则的倾斜性，它以一种倾斜的方式来满足开闭原则，为增加新产品族提供方便，但不能为增加新产品结构提供方便，增加新的产品结构是需要修改源代码，这违背了开闭原则。这要求设计人员在设计之初就能够全面考虑，不要在开发节点在修改增加新的产品结构。

总结：

* 主要优点

隔了了对象的使用和创建

增加新的产品族很方便，符合开闭原则

* 主要缺点

增加新的产品结构需要修改原有的代码，违背了开闭原则。

* 适用场景

系统中需要使用产品族，同一产品族的产品总是一起被使用；产品等级结构稳定

第七章 对象克隆——原型模式

原型模式定义：使用原型实例指定创建对象的类。并且通过克隆这些原型创建新的对象。原型模式是一种另类的创建型模式，创建克隆的工厂就是原型类自身，工厂方法由克隆方法来提供。

结构包含：

1. Prototype——抽象原型类
2. ConcretePrototype——具体原型类
3. Client——客户类

原型模式的核心在于如何实现克隆方法。

实现方式：

1. 抽象的接口或者抽象类，该类或接口定义了clone的抽象方法
2. 具体的原型类实现此接口继承此抽象类并重写clone方法
3. 另外，java提供了cloneable接口，所以可以作为1部分

class ConcretePrototype implements Cloneable{  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 ConcretePrototype concretePrototype=null;  
 //do stuff  
  
 return concretePrototype;  
 }  
}

两种克隆方式：

1. 浅克隆
2. 深克隆

浅克隆和深克隆的主要区别在于是否支持引用类型的成员变量的复制。

浅克隆：

1. 如果成员变量为值类型则复制一份给克隆对象
2. 如果成员变量未引用类型则将原对象中的成员变量的地址复制一份给克隆对象

由此可见，在浅克隆下，原对象和克隆对象的引用成员变量指向同一个实例。

Object的克隆方法实现就是浅克隆，所以可以直接使用Object的clone方法：

class ConcretePrototype implements Cloneable{  
 //do other stuff  
  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 Object obj=super.clone();  
  
 return (ConcretePrototype)obj;  
 }  
}

深克隆：

无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象。

实现深度克隆可以通过序列化方式来实现（即将原对象序列化存入到流，然后在从此流中读取出对象，则得到的就是一个新的对象），但是需要原型对象实现序列化接口；否则就需要手动实现克隆操作。

1. 序列化实现深度克隆，注意成员变量也必须实现序列化接口：

class MemberClass implements Serializable{  
  
}  
  
class Prototype implements Serializable {  
 private MemberClass memberClass;  
  
 public Prototype deepClone() {  
 Prototype prototype = null;  
 try {  
 //serialize  
 ByteArrayOutputStream out = new ByteArrayOutputStream();  
 ObjectOutputStream oos = new ObjectOutputStream(out);  
 oos.writeObject(this);  
 //deep clone  
 ByteArrayInputStream bis = new ByteArrayInputStream(out.toByteArray());  
 ObjectInputStream ois = new ObjectInputStream(bis);  
 prototype = (Prototype) ois.readObject();  
  
 return prototype;  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 } catch (ClassNotFoundException ex) {  
 ex.printStackTrace();  
 }  
  
 return prototype;  
 }  
}

原型管理器的引入和实现：原型管理器是将多个原型对象存储在一个集合中供客户端使用，他是一个专门负责克隆对象的工厂。在原型管理器中针对抽象原型类进行编程，以便扩展。实现代码：

interface OfficialDoc{  
 OfficialDoc clone();  
 void display();  
}  
  
class ADoc implements OfficialDoc{  
 public OfficialDoc clone() {  
 return new ADoc();  
 }  
  
 public void display() {  
 System.out.println("ADoc display");  
 }  
}  
  
class BDoc implements OfficialDoc{  
 public OfficialDoc clone() {  
 return new BDoc();  
 }  
  
 public void display() {  
 System.out.println("BDoc display");  
 }  
}  
  
public class PrototypeManager {  
 private Map<String,OfficialDoc> prototypes;  
 public PrototypeManager(){  
 this.prototypes=new HashMap<String, OfficialDoc>();  
 }  
  
 public void addPrototype(String key,OfficialDoc doc){  
 this.prototypes.put(key,doc);  
 }  
  
 public OfficialDoc getPrototype(String key){  
 return this.prototypes.get(key);  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 PrototypeManager manager=new PrototypeManager();  
 manager.addPrototype("a",new ADoc());  
 manager.addPrototype("b",new BDoc());  
  
 OfficialDoc doc=manager.getPrototype("a");  
 }  
}

总结

* 主要优点

提高创建新实例的效率，如果实例创建过于复杂的话

扩展性较好

* 主要缺点

深克隆实现较复杂

* 适用场景

创建新的对象的成本较大

可用原型模式保存一份对象的状态

第8章 复杂对象的组装与创建——建造者模式

建造者模式也称为生成器模式，它是一种较为复杂、使用频率也相对较低的创建型模式。

它将客户端与包含多个组成部分（或部件）的复杂对象的创建过程分离，它关注如何一步一步的创建一个复杂对。

建造者模式定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。结构包含如下：

1. Builder——抽象建造者，它为创建一个产品Product对象的各个部件指定抽象接口。在抽象/接口中，一般包含两类方法：
   1. buildPartX()方法用于创建复杂对象的各个组成部分
   2. 另一类方法是getResult()，用于返回复杂对象。

Builder既可以是抽象类，也可以是接口。

1. ConcreteBuilder——具体建造者
2. Product——产品角色

它是被构建的复杂对象，包含多个组件，具体建造者创建该产品的内部表示并定义其装配过程。

1. Director——指挥者

复杂对象是指那些包含多个成员变量的对象。

实现一：带有指挥者Director

class Actor{  
 private String Type;  
 private int sex;  
 private String face;  
 private String costume;  
 private String hairStyle;  
  
 public String getType() {  
 return Type;  
 }  
  
 public void setType(String type) {  
 this.Type = type;  
 }  
  
 public int getSex() {  
 return sex;  
 }  
  
 public void setSex(int sex) {  
 this.sex = sex;  
 }  
  
 public String getFace() {  
 return face;  
 }  
  
 public void setFace(String face) {  
 this.face = face;  
 }  
  
 public String getCostume() {  
 return costume;  
 }  
  
 public void setCostume(String costume) {  
 this.costume = costume;  
 }  
  
 public String getHairStyle() {  
 return hairStyle;  
 }  
  
 public void setHairStyle(String hairStyle) {  
 this.hairStyle = hairStyle;  
 }  
}  
  
//抽象类作为构建者抽象  
abstract class ActorBuilder{  
 protected Actor actor=new Actor();  
  
 public abstract void buildType();  
 public abstract void buildSex();  
 public abstract void buildFace();  
 public abstract void buildCostume();  
 public abstract void buildHairStyle();  
  
 public Actor build(){  
 return this.actor;  
 }  
}  
  
class HeroBuilder extends ActorBuilder{  
 @Override  
 public void buildType() {  
 this.actor.setType("hero");  
 }  
  
 @Override  
 public void buildSex() {  
 this.actor.setSex(1);  
 }  
  
 @Override  
 public void buildFace() {  
 this.actor.setFace("英俊");  
 }  
  
 @Override  
 public void buildCostume() {  
 this.actor.setCostume("盔甲");  
 }  
  
 @Override  
 public void buildHairStyle() {  
 this.actor.setHairStyle("飘逸");  
 }  
  
 @Override  
 public Actor build() {  
 return super.build();  
 }  
}  
  
class AngleBuilder extends ActorBuilder{  
 @Override  
 public void buildType() {  
 this.actor.setType("天使");  
 }  
  
 @Override  
 public void buildSex() {  
 this.actor.setSex(0);  
 }  
  
 @Override  
 public void buildFace() {  
 this.actor.setFace("漂亮");  
 }  
  
 @Override  
 public void buildCostume() {  
 this.actor.setCostume("白裙");  
 }  
  
 @Override  
 public void buildHairStyle() {  
 this.actor.setHairStyle("披肩长发");  
 }  
  
 @Override  
 public Actor build() {  
 return super.build();  
 }  
}  
  
class ActorController{  
 public Actor construct(ActorBuilder builder){  
 builder.buildType();  
 builder.buildSex();  
 builder.buildCostume();  
 builder.buildFace();  
 builder.buildHairStyle();  
  
 return builder.build();  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 ActorController actorController=new ActorController();  
 ActorBuilder builder=new HeroBuilder();  
 Actor actor=actorController.construct(builder);  
  
 System.out.println(actor.getType());  
 System.out.println(actor.getCostume());  
 }  
}

扩展时只需提供具体的构建者即可。

* 省略Director类：需要将其construct方法放到抽象构建者类/接口中

实例1：

abstract class ActorBuilder2 {  
 private Actor actor;  
  
 public abstract void buildType();  
 public abstract void buildSex();  
 public abstract void buildFace();  
 public abstract void buildCostume();  
 public abstract void buildHairStyle();  
  
 public Actor construct(){  
 this.buildType();  
 this.buildSex();  
 this.buildFace();  
 this.buildCostume();  
 this.buildHairStyle();  
  
 return this.actor;  
 }  
}

除了以上的建造顺序控制外，建造者还能够有很多的方式来控制产品的创建流程。

总结：

* 主要优点

1. Client不需要直到内部的创建细节，将产品的本身和产品的创建过程解耦，将产品的创建和产品的使用解耦。
2. 系统扩展方便，创建新的产品对象只需要增加新的建造者，抽象建造者面向抽象编程，符合开闭原则。
3. 能够更加精细的控制产品的创建过程

* 主要缺点
* 适用场景

1. 需要的产品具有复杂的内部结构
2. 隔离复杂对象的创建和使用

Apache对复杂对象构建的实例，返回this，链式构建：

public class HttpClientBuilder {  
 private HttpRequestExecutor requestExec;  
 private HostnameVerifier hostnameVerifier;  
 private LayeredConnectionSocketFactory sslSocketFactory;  
 private SSLContext sslContext;  
 private HttpClientConnectionManager connManager;  
 private boolean connManagerShared;  
 private SchemePortResolver schemePortResolver;  
 private ConnectionReuseStrategy reuseStrategy;  
 private ConnectionKeepAliveStrategy keepAliveStrategy;  
 private AuthenticationStrategy targetAuthStrategy;  
 private AuthenticationStrategy proxyAuthStrategy;  
 private UserTokenHandler userTokenHandler;  
 private HttpProcessor httpprocessor;  
 private DnsResolver dnsResolver;  
 private LinkedList<HttpRequestInterceptor> requestFirst;  
 private LinkedList<HttpRequestInterceptor> requestLast;  
 private LinkedList<HttpResponseInterceptor> responseFirst;  
 private LinkedList<HttpResponseInterceptor> responseLast;  
 private HttpRequestRetryHandler retryHandler;  
 private HttpRoutePlanner routePlanner;  
 private RedirectStrategy redirectStrategy;  
 private ConnectionBackoffStrategy connectionBackoffStrategy;  
 private BackoffManager backoffManager;  
 private ServiceUnavailableRetryStrategy serviceUnavailStrategy;  
 private Lookup<AuthSchemeProvider> authSchemeRegistry;  
 private Lookup<CookieSpecProvider> cookieSpecRegistry;  
 private Map<String, InputStreamFactory> contentDecoderMap;  
 private CookieStore cookieStore;  
 private CredentialsProvider credentialsProvider;  
 private String userAgent;  
 private HttpHost proxy;  
 private Collection<? extends Header> defaultHeaders;  
 private SocketConfig defaultSocketConfig;  
 private ConnectionConfig defaultConnectionConfig;  
 private RequestConfig defaultRequestConfig;  
 private boolean evictExpiredConnections;  
 private boolean evictIdleConnections;  
 private long maxIdleTime;  
 private TimeUnit maxIdleTimeUnit;  
 private boolean systemProperties;  
 private boolean redirectHandlingDisabled;  
 private boolean automaticRetriesDisabled;  
 private boolean contentCompressionDisabled;  
 private boolean cookieManagementDisabled;  
 private boolean authCachingDisabled;  
 private boolean connectionStateDisabled;  
 private int maxConnTotal = 0;  
 private int maxConnPerRoute = 0;  
 private long connTimeToLive = -1L;  
 private TimeUnit connTimeToLiveTimeUnit;  
 private List<Closeable> closeables;  
 private PublicSuffixMatcher publicSuffixMatcher;  
  
 public static HttpClientBuilder create() {  
 return new HttpClientBuilder();  
 }  
  
 protected HttpClientBuilder() {  
 this.connTimeToLiveTimeUnit = TimeUnit.MILLISECONDS;  
 }  
  
 public final HttpClientBuilder setRequestExecutor(HttpRequestExecutor requestExec) {  
 this.requestExec = requestExec;  
 return this;  
 }

……

public CloseableHttpClient build() {  
 PublicSuffixMatcher publicSuffixMatcherCopy = this.publicSuffixMatcher;  
 if(publicSuffixMatcherCopy == null) {  
 publicSuffixMatcherCopy = PublicSuffixMatcherLoader.getDefault();  
 }  
  
 HttpRequestExecutor requestExecCopy = this.requestExec;  
 if(requestExecCopy == null) { //判断已查看是否被设置  
 requestExecCopy = new HttpRequestExecutor();  
 }  
 final HttpClientConnectionManager connManagerCopy = this.connManager;  
 Object reuseStrategyCopy;  
 Object proxyAuthStrategyCopy;  
 if(connManagerCopy == null) {  
 reuseStrategyCopy = this.sslSocketFactory;  
 if(reuseStrategyCopy == null) {  
 String[] supportedProtocols = this.systemProperties?split(System.getProperty("https.protocols")):null;  
 String[] supportedCipherSuites = this.systemProperties?split(System.getProperty("https.cipherSuites")):null;  
 proxyAuthStrategyCopy = this.hostnameVerifier;  
 if(proxyAuthStrategyCopy == null) {  
 proxyAuthStrategyCopy = new DefaultHostnameVerifier(publicSuffixMatcherCopy);  
 }  
 ……  
 if(this.maxConnTotal > 0) {  
 poolingmgr.setMaxTotal(this.maxConnTotal);  
 }  
  
 if(this.maxConnPerRoute > 0) {  
 poolingmgr.setDefaultMaxPerRoute(this.maxConnPerRoute);  
 }  
  
 connManagerCopy = poolingmgr;  
 }  
 reuseStrategyCopy = this.reuseStrategy;  
 if(reuseStrategyCopy == null) {  
 if(this.systemProperties) {  
 String s = System.getProperty("http.keepAlive", "true");  
 if("true".equalsIgnoreCase(s)) {  
 reuseStrategyCopy = DefaultClientConnectionReuseStrategy.INSTANCE;  
 } else {  
 reuseStrategyCopy = NoConnectionReuseStrategy.INSTANCE;  
 }  
 } else {  
 reuseStrategyCopy = DefaultClientConnectionReuseStrategy.INSTANCE;  
 }  
 }  
 ……

return new InternalHttpClient((ClientExecChain)execChain, ……)

第三部分 组合的艺术——结构性模式

结构性模式关注如何将现有的类或对象组织在一起形成更加强大的结构。

结构性模式包含7种，分别为：

1. 适配器模式
   1. 接口转换，使得不兼容的那些类可以一起工作
2. 桥接模式
3. 组合模式
4. 装饰模式
   1. 动态的给一个对象增加一些额外的职责
5. 外观模式
6. 享元模式
7. 代理模式
   1. 给某一个对象提供一个代理，并由代理对象控制对原对象的引用

第九章 不兼容结构的协调——适配器模式

适配器模式，专门用于解决不兼容结构的问题。

在适配器模式中，存在一个被称为适配器的包装类，与之相对应，它所包装的对象称为适配者，即被适配的类。

适配器的实现就是把客户类的请求转换为对适配者的相应接口的调用。即当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，这个过程对客户类是透明的。因此适配器是那些因为接口不兼容的类能够一起工作。

适配器模式的定义：将一个接口转换成客户希望的另一个接口，是接口不兼容的那些类可以一起工作。器别名为包装器（Wrapper）。

适配器模式分为两类：

1. 对象适配器模式：适配器与适配者之间是关联关系。
2. 类适配器模式：适配器与适配者之间时继承或者实现关系。

实际开发中对象适配器模式使用频率较高。

对象适配器模式包含以下部分：

1. Target(目标抽象类)
2. Adapter(适配器类)：作为转换器，对Adaptee和Target进行适配，它通过继承Target并关联一个Adaptee对象使二者产生联系
3. Adaptee(适配者类)

对象适配器简单实例：

interface ScoreOperation{  
 int[] sort(int[] arr);//排序  
 int seatch(int[] arr,int v);//查找  
}  
  
class QuickSort{  
 public int[] quickSort(int[] arr){  
 System.out.println("快速排序算法排序");  
 return null;  
 }  
}  
  
class BinnarySearch{  
 public int binSearch(int[] arr,int v){  
 System.out.println("二分查找");  
 return 0;  
 }  
}  
  
class OperationAdapter implements ScoreOperation{  
 private QuickSort sort;  
 private BinnarySearch search;  
 public OperationAdapter(){  
 this.sort=new QuickSort();  
 this.search=new BinnarySearch();  
 }  
 public int[] sort(int[] arr) {  
 return this.sort.quickSort(arr);  
 }  
  
 public int seatch(int[] arr,int v) {  
 return this.search.binSearch(arr,v);  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 int[] arr={1,2,3};  
 OperationAdapter adapter=new OperationAdapter();  
 adapter.sort(arr);  
 adapter.seatch(arr,2);  
  
 }  
}

适配器模式对新增加的适配者可以增加新的适配器，符合开闭原则。

类适配器模式与对象适配器模式之间的最大区别是器适配器和适配者之家是继承关系而非关联关系。由于java等语言不支持多重类继承，因此类适配器模式的使用受到很大的影响。所以在java等面向对象的语言中，大部分情况下使用的是对象适配器模式，类适配器模式很少使用。

双向适配器模式：适配器中同时包含Target对象和适配者对象，Target可以通过适配器调用适配者对象的方法，适配者对象也可以通过适配器调用Target对象的方法。

代码实现：

interface QuickSort2{  
 int[] quickSort(int[] arr);  
}  
  
class Adapter implements ScoreOperation,QuickSort2{  
 private ScoreOperation scoreOperation;  
 private QuickSort quickSort;  
  
 private Adapter(ScoreOperation operation,QuickSort sort){  
 this.scoreOperation= operation;  
 this.quickSort=sort;  
 }  
  
 public int[] sort(int[] arr) {  
 //do some stuff  
 return this.quickSort.quickSort(arr);  
 }  
  
 public int search(int[] arr, int v) {  
 return 0;  
 }  
   
 public int[] quickSort(int[] arr) {  
 //do some stuff  
 return this.scoreOperation.sort(arr);  
 }  
}

缺省适配器模式定义：当不需要实现一个接口所提供的所有方法时，可先设计一个抽象类实现该接口，并为接口中每个方法提供一个默认的实现（空方法），那么抽象类的子类可以选择性的覆盖父类的某些方法来实现需求，它适用于不想使用一个接口中的所有方法的情况。又被称为单接口适配器模式。它包含以下结构：

1. ServiceInterface适配器接口
2. AbstractServiceClass缺省适配器类:通常定义为抽象类，因为对它实例化没有任何的意义；使用空方法实现在适配器接口中定义的所有方法
3. ConcreteServiceClass具体业务类

代码实例：

interface IOperation{  
 void doA();  
 void doB();  
 void doC();  
 void doD();  
 void doE();  
 void doF();  
 void doG();  
}  
  
abstract class AbsOperation implements IOperation{  
 public void doA() {  
  
 }  
  
 public void doB() {  
  
 }  
  
 public void doC() {  
  
 }  
  
 public void doD() {  
  
 }  
  
 public void doE() {  
  
 }  
  
 public void doF() {  
  
 }  
  
 public void doG() {  
  
 }  
}  
  
class OperationABC extends AbsOperation{  
 @Override  
 public void doA(){  
 //do some stuff  
 }  
  
 @Override  
 public void doB(){  
 //do some stuff  
 }  
  
 @Override  
 public void doC(){  
 //do some stuff  
 }  
}

总结：

* 主要优点

1. 将目标类和适配者类解耦
2. 增加了类的透明新和复用性
3. 灵活性和扩展性都很好。只需新增适配器来匹配新的适配者，符合开闭原则。

* 主要缺点

1. 对于单继承语言，类适配器的实现有限制

* 适用场景

兼容两个类的结构

第十章 处理多维度变化——桥接模式

单一职责：仅有一个引起类变化的原因，否则就不符合单一职责原则。

如果软件系统中存在某个类存在两个独立变化的维度，通过该模式可以将这两个维度分离出来，是两者可以独立扩展，让系统更加符合单一职责原则。

它将两个独立变化的维度设计为两个独立的继承等级结构，并且在抽象层建立一个抽象关联，该关联关系类似一条连接两个独立继承结构的桥，故名桥接模式。

桥接模式定义：将抽象部分与其实现部分分离，是它们都可以独立的变化，它是一种对象结构型模式，又称为柄体模式或接口模式。其结构包含：

1. Abstraction类
2. RefinedAbstraction扩充抽象类
3. Implementor实现类接口
4. ConcreteImplementor具体实现类

桥接模式的实现体现了很多面向对象设计原则，包括单一职责原则、开闭原则、合成复用原则、里氏替换原则和依赖倒置原则等。

实例代码：

class Matrix{}  
  
abstract class Image{  
 protected ImageImp imp;  
 public void setImp(ImageImp imp){  
 this.imp=imp;  
 }  
  
 public abstract void parseFile(String fileName);  
}  
  
interface ImageImp{  
 public void doPaint(Matrix m);  
}  
  
class WindowsImp implements ImageImp{  
 public void doPaint(Matrix m) {  
 System.out.println("windows do paint");  
 }  
}  
  
class UnixImp implements ImageImp{  
 public void doPaint(Matrix m) {  
 System.out.println("Unix do paint");  
 }  
}  
class JPGImage extends Image{  
 public void parseFile(String fileName){  
 Matrix m=new Matrix();  
 imp.doPaint(m);  
 System.out.println(fileName+" ,格式为JPG");  
 }  
}  
  
class PNGImage extends Image{  
 @Override  
 public void parseFile(String fileName) {  
 Matrix m=new Matrix();  
 imp.doPaint(m);  
 System.out.println(fileName+" ,格式为PNG");  
 }  
}  
  
class BMPImage extends Image{  
 @Override  
 public void parseFile(String fileName) {  
 Matrix m=new Matrix();  
 imp.doPaint(m);  
 System.out.println(fileName+" ,格式为BMP");  
 }  
}  
  
class GIFImage extends Image{  
 @Override  
 public void parseFile(String fileName) {  
 Matrix m=new Matrix();  
 imp.doPaint(m);  
 System.out.println(fileName+" ,格式为GIF");  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 Image image;  
 ImageImp imp=new WindowsImp();  
 image=new JPGImage();  
 image.setImp(imp);  
 image.parseFile("");  
 }  
}

当增加新的图片格式或者新的操作系统时，主需要增加一个对应的扩充类或者具体的实现类即可。符合开闭原则。

采用增加适配器的方式将第三方类集成到系统中。

桥接模式用于系统的初步设计。

总结：

桥接模式是jvm和实现jdbc等驱动程序的核心模式之一。桥接模式为多维度变化的系统提供了一套完整的解决方案，并且降低了系统的复杂度。

* 主要优点

1. 分离抽象接口及其实现部分
2. 提供了系统的可扩展性，符合开闭原则

* 主要缺点

1. 桥接模式的增加会增加系统的理解和设计难度

* 适用场景

1. 一个系统存在多个维度变化
2. 一个类存在两个或者多个独立变化的维度，且这两个或者多个维度都需要独立进行扩展

第十一章 树形结构的处理——组合模式

组合模式通过一种巧妙的设计方案使得用户可以一致性的处理整个树形机构或者树形结构的一部分，也可以一致性的处理树形结构中的叶子节点和容器节点。

组合模式定义：组合多个对象形成树形结构以表示具有整体-部分关系的层次结构。组合模式对单个对象（即叶子对象）和组合对象（即容器对象）的使用具有一致性，组合模式又可称为部分-整体模式，它是一种对象结构型模式。

在组合模式中引入了抽象构件类Component,它是所有容器类和叶子类的公共父类。客户端对Component进行编程。

组合模式包含：

1. Component 抽象构件
   1. 可以是结构或者抽象类
   2. 为叶子和容器对象的公共父类或实现接口
   3. 在抽象构件中定义了访问和管理它的子构件的方法
2. Leaf 叶子构件
   1. 表示叶子对象
   2. 没有子节点，可以在访问子节点方法中抛出异常
   3. 也可以使用抽象类对抽象构件的方法做空实现
3. Composite 容器构件
   1. 表示容器节点
   2. 提供集合用于存储子节点
   3. 在其业务方法中可以递归调用其子节点的业务方法

组合模式的关键是定义了一个抽象构件类，它既可以代表叶子节点又可以代表容器节点，而客户端针对该抽象类构件进行编程且无需知道它到底是叶子还是容器，从而达到统一处理。

简单实例：

abstract class Component{  
 public abstract void add(Component c);//添加成员  
 public abstract void remove(Component c);//删除成员  
 public abstract Component getChild(int i);//获取成员  
 public abstract void operation();//业务方法  
}  
  
class Leaf extends Component{  
 @Override  
 public void add(Component c) {  
 //叶子节点不能添加成员  
 //异常处理或者错误提示  
 }  
  
 @Override  
 public void remove(Component c) {  
 //叶子节点不能删除成员  
 //异常处理或者错误提示  
 }  
  
 @Override  
 public Component getChild(int i){  
 //叶子节点不能获取成员  
 //异常处理或者错误提示  
 return null;  
 }  
  
 @Override  
 public void operation() {  
 //叶子构件具体业务方法实现  
 }  
}  
  
class Composite extends Component{  
 private List<Component> list=new ArrayList<Component>();  
 @Override  
 public void add(Component c) {  
 list.add(c);  
 }  
  
 @Override  
 public void remove(Component c) {  
 list.remove(c);  
 }  
  
 @Override  
 public Component getChild(int i) {  
 return list.get(i);  
 }  
  
 @Override  
 public void operation() {  
 //容器构件具体业务方法实现  
 //递归调用成员构件的业务方法  
 for(Component c:list){  
 c.operation();  
 }  
 }  
}

完整解决方案：

abstract class AbstractFile{  
 public abstract void add(AbstractFile file);  
 public abstract void remove(AbstractFile file);  
 public abstract AbstractFile getChild(int i);  
 public abstract void killVirus();  
}  
  
class ImageFile extends AbstractFile{  
 private String name;  
 public ImageFile(String name){  
 this.name=name;  
 }  
  
 @Override  
 public void add(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public void remove(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public AbstractFile getChild(int i) {  
 System.out.println("对不起，不支持该方法");  
 return null;  
 }  
  
 @Override  
 public void killVirus() {  
 System.out.println("---- 对图像文件 '"+this.name+"' 进行杀毒");  
 }  
}  
  
class TextFile extends AbstractFile{  
 private String name;  
 public TextFile(String name){  
 this.name=name;  
 }  
  
 @Override  
 public void add(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public void remove(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public AbstractFile getChild(int i) {  
 System.out.println("对不起，不支持该方法");  
 return null;  
 }  
  
 @Override  
 public void killVirus() {  
 System.out.println("---- 对文本文件 '"+this.name+"' 进行杀毒");  
 }  
}  
  
class VideoFile extends AbstractFile{  
 private String name;  
 public VideoFile(String name){  
 this.name=name;  
 }  
  
 @Override  
 public void add(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public void remove(AbstractFile file) {  
 System.out.println("对不起，不支持该方法");  
 }  
  
 @Override  
 public AbstractFile getChild(int i) {  
 System.out.println("对不起，不支持该方法");  
 return null;  
 }  
  
 @Override  
 public void killVirus() {  
 System.out.println("---- 对视频文件 '"+this.name+"' 进行杀毒");  
 }  
}  
  
class Folder extends AbstractFile{  
 private List<AbstractFile> list=new ArrayList<AbstractFile>();  
 private String name;  
 public Folder(String name){  
 this.name=name;  
 }  
  
 @Override  
 public void add(AbstractFile file) {  
 list.add(file);  
 }  
  
 @Override  
 public void remove(AbstractFile file) {  
 list.remove(file);  
 }  
  
 @Override  
 public AbstractFile getChild(int i) {  
 return list.get(i);  
 }  
  
 @Override  
 public void killVirus() {  
 System.out.println("\*\*\*\* 对文件夹 '"+this.name+"' 进行杀毒");  
 for(AbstractFile file:list){  
 file.killVirus();  
 }  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 AbstractFile file1,file2,file3,file4,file5,folder1,folder2,folder3,folder4;  
  
 folder1=new Folder("sunny 的资料");  
 folder2=new Folder("图像文件");  
 folder3=new Folder("文本文件");  
 folder4=new Folder("视频文件");  
  
 file1=new ImageFile("a.jpg");  
 file2=new ImageFile("b.gif");  
 file3=new TextFile("c.txt");  
 file4=new TextFile("d.doc");  
 file5=new VideoFile("e.rmvb");  
  
 //图像目录  
 folder2.add(file1);  
 folder2.add(file2);  
  
 //文本目录  
 folder3.add(file3);  
 folder3.add(file4);  
  
 //视频目录  
 folder4.add(file5);  
  
 //sunny 目录  
 folder1.add(folder2);  
 folder2.add(folder3);  
 folder1.add(folder4);  
  
 //对sunny 目录进行杀毒  
 folder1.killVirus();  
 }  
}

递归调用

可根据自己需要修改所需要操作的根节点，符合开闭原则。

透明模式与安全组合模式：

由于抽象构件作为叶子节点和容器节点的父类，需要定义很多的访问和管理节点的方法，这就导致它定义了很多适用于容器节点但是不适用于叶子节点的方法，而叶子节点不得不给出异常或者错误提示。

解决方案一：可以将这些异常或者错误提示移动到抽象构件中提供统一的默认实现。

abstract class AbsrtractFile{  
 public void add(AbstractFile file){  
 System.out.println("对不起，不支持该方法");  
 }  
  
 public void remove(AbstractFile file){  
 System.out.println("对不起，不支持该方法");  
 }  
  
 public AbstractFile getChild(int i){  
 System.out.println("对不起，不支持该方法");  
 return null;  
 }  
  
 abstract public void killVirus();  
}

解决方案二：也可以在抽象构件中不声明任何用于访问和管理成员构件的方法：这时client不得不使用容器构件本身来定义容器构件，不能达到叶子节点和容器节点处理的一致性，容器节点对client来说不是透明的，即client必须明确定义。

abstract class AbstractFile2{  
 abstract public void killVirus();  
}

在使用组合模式时，根据抽象构件类的定义形式，可将组合模式分为透明组合模式和安全组合模式。

1. 透明模式：抽象类构件Component中声明了所有用于管理成员对象的方法，包括add、remove以及getChild等方法。这样做的好处是确保所有的构建类都有相同的接口。在客户端看来，叶子对象和容器对象锁提供的方法是一致的。客户端可以相同的对待所有的对象。缺点是不够安全。因为叶子节点和容器节点本质上是有区别的，叶子节点不能包含成员对象，为其提供add、remove和getChild等方法是没有意义的。

透明模式是组合模式的标准模式。

1. 安全模式：在抽象构件Component中没有声明任何用于管理成员对象的方法，而是在Composite中声明并实现这些方法。这样做是安全的，因为根本不向叶子对象提供这些管理成员对象的方法。缺点是不够透明，client不能完全针对抽象进行编程，必须有区别的对待叶子构件和容器构件。

总结

* 主要优点

1. 组合模式可以清楚定义分层次的复杂对象，表示对象的全部或部分层次，它让client忽略了层次的差异。
2. Client可以一致的是以使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了client的编码。
3. 新增容器构件和叶子构件都很方便，不需对现有类库做任何的修改，符合开闭原则。
4. 通过叶子对象和容器对象的递归组合，可以形成复杂的树形结构，而对树形结构的操作则非常简单。

* 主要缺点
* 适用场景

1. 在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，client可以一致性的对待他们。
2. 处理一个树形结构
3. 能够从系统中分离出容器对象和叶子对象

第十二章 扩展系统功能——装饰者模式

根据合成复用原则，在实现功能复用时，要多用关联少用继承。

装饰模式可以在不改变一个对象本身功能的基础上该对象增加额外的新行为。它是一种替代继承的技术，它无需定义子类就能实现给对象动态的增加在指责。在装饰模式中引入了装饰类，在装饰类中既可以调用待装饰的原有类的方法，还可以增加新的方法，以扩充原有类的方法。

装饰器模式定义：动态的给一个对象增加一些额外的职责，就增加对象的功能来说，装饰模式比生成子类实现更为灵活。装饰模式是一种对象结构型模式。

在装饰模式中，为了让系统具有更好的灵活性和可扩展性，通常会定义一个抽象装饰类，而将具体的装饰类作为它的子类。

装饰模式的结构：

1. Component 抽象构件
   1. 它是具体构件和抽象装饰类的共同父类，声明了在具体构件中实现的业务方法
   2. 它的引入可以使client以一致的方式来处理未被装饰的对象和装饰之后的对象，实现client的透明操作（这点有点类似于组合模式）
2. ConcreteComponent 具体构件
   1. 它是抽象构件的子类，用于定义具体的构件对象
   2. 它实现了在抽象构件中声明的方法
   3. 装饰器可以给它增加额外的职责/方法
3. Decorator 抽象装饰类
   1. 它也是抽象构件的子类，用于给具体的构件增加职责
   2. 具体的职责在其子类中实现
   3. 它维护一个指向抽象构件对象的引用，通过该引用可以调用装饰之前构件对象的方法，并通过具体的装饰类扩展该方法
4. ConcreteDecorator 具体装饰类
   1. 它是抽象装饰类的子类，负责向构件中添加新的职责。
   2. 每个具体装饰类都定义类一些新的行为，可以调用在抽象装饰类中定义的方法，并可以增加新的方法以扩充对象的行为。

装饰模式的核心在于抽象装饰类的设计。

抽象装饰类和具体装饰类的简单实例：

interface Component{  
 void operation();  
}  
  
class Decorator implements Component{  
 private Component component;//维持一个对抽象构件对象的引用  
 public Decorator(Component component){  
 this.component=component;  
 }  
  
 public void operation() {//调用原有的业务方法，具体的装饰过程需要其子类来完成  
 component.operation();  
 }  
}

class ConcreteDecoratro extends Decorator{  
 public ConcreteDecoratro(Component component){  
 super(component);  
 }  
  
 @Override  
 public void operation(){  
 super.operation();//调用原有业务方法  
 this.addBehavior();  
 }  
  
 public void addBehavior(){  
 //增强业务逻辑  
 }  
}

因为Decorator继承或者实现抽象构件类，因此Decorator及其子类都是Component，还可以将一个已经装饰过的Decorator子类的对象再注入其中进行多次装饰，从而对原有功能进行多次扩展。

完整代码如下：

//抽象构件  
abstract class Component{  
 public abstract void display();  
}  
  
//具体构件  
class Window extends Component{  
 public void display(){  
 System.out.println("Window display");  
 }  
}  
  
class TextBox extends Component{  
 @Override  
 public void display() {  
 System.out.println("TextBox display");  
 }  
}  
  
//具体构件  
class ListBox extends Component{  
 @Override  
 public void display() {  
 System.out.println("ListBox display");  
 }  
}  
  
//抽象装饰器类  
class ComponentDecorator extends Component{  
 private Component component;  
 public ComponentDecorator(Component component){  
 this.component=component;  
 }  
  
 @Override  
 public void display() {  
 this.component.display();  
 }  
}  
  
//具体装饰器类  
class ScrollBarDecorator extends ComponentDecorator{  
 public ScrollBarDecorator(Component component){  
 super(component);  
 }  
  
 public void setScrollBar(){  
 System.out.println("为构件增加滚动条");  
 }  
  
 public void display(){  
 this.setScrollBar();  
 super.display();  
 }  
}  
  
//具体装饰器类  
class BlackBorderDecorator extends ComponentDecorator{  
 public BlackBorderDecorator(Component component){  
 super(component);  
 }  
  
 public void setBlackBorder(){  
 System.out.println("为构件增加黑色边框");  
 }  
  
 public void display(){  
 this.setBlackBorder();  
 super.display();  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 Component component,componentSB,componentBB;  
 component=new Window();  
 componentSB=new ScrollBarDecorator(component);  
 componentSB.display();  
  
 //扩展多重功能  
 componentBB=new BlackBorderDecorator(componentSB);  
 componentBB.display();

}  
}

如果需要在原有系统中增加一个新的具体构件类或者新的具体装饰类，无需修改现有类库的代码，只需要将它们分别作为抽象构件类或者抽象装饰类的子类即可，符合开闭原则。

* 透明装饰模式和半透明装饰模式（类似透明组合模式）

在实际使用过程中，由于新增行为可能需要单独调用（即直接使用具体装饰器对象调用新增的方法，因此在声明时就需要指定类型而不是统一的Component，所以不是透明的），因此这种形式的装饰器模式也经常出现，称为半透明装饰器模式。而标准的装饰器模式是透明模式。

1. 半透明装饰器模式实例：

class Client{  
 public static void main(String[] args) {  
 Component component;  
 component=new Window();  
 ScrollBarDecorator componentSB;   
 componentSB=new ScrollBarDecorator(component);  
 componentSB.setScrollBar();  
 }  
}

1. 透明模式实例

在透明模式中，要求client完全针对抽象编程，即上面详细实例。对于client来说具体构件类型和具体装饰对象没有任何的区别。

使用抽象构件类型Component定义全部具体构件对象和具体装饰对象，client可以一致性的使用这些对象。

在实现透明装饰模式时，要求具体的装饰类的operation方法覆盖抽象装饰类的operation方法，除了调用原有对象的operation外还需要调用新增的addBehavior方法来增加新行为。

注意事项：

1. 尽量装饰类的接口与被装饰类的接口相同，这样对于client而言无论是装饰之前的对象还是装饰之后的对象都可以一致对待。即尽量使用透明模式
2. 尽量保持具体装饰类的轻便，即不要在一个具体装饰类中定义过多的扩展，可以使用多次扩展来代替

总结

* 主要优点

1. 扩展对象的功能，不需要继承，不会导致类的数量急剧增加
2. 多次装饰从而扩展多项功能
3. 具体构建类和具体的装饰类可以独立变化，系统耦合度低，符合开闭原则

* 主要缺点
* 适用场景

1. 动态透明的方式给单个对象添加职责
2. 代替继承扩展系统的功能

第十三章 提供统一入口——外观模式

外观类充当了软件系统中的服务员，它为多个业务类的调用提供了一个统一的入口，简化了类与类之间的交互。

在client类和业务类之间增加一个外观类，由外观类来封装与业务类之间的交互，而client只需要与外观类交互即可。

外观模式定义：外部与一个子系统的通信通过一个统一的外观角色进行，为子系统中的一组接口提供一个一致的入口，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式有成为门面模式，它是一种对象结构性模式。

外观模式的结构包含：

1. Façade 外观角色
   1. Client可以调用这个角色的方法，在外观角色中可以知道相关（一个或者多个）子系统的功能和责任
   2. 在正常情况下，它将所有从client发来的请求委派到相应的子系统中去，传递给相应的子系统对象处理
2. SubSystem 子系统角色
   1. 每个子系统都可以由client直接调用或者由外观角色调用，它处理有外观类传递过来的请求
   2. 子系统并不知道外观的存在，对于字系统而言，外观角色仅仅是另外一个client而已

外观模式的主要目的在于减低系统的复杂程度和耦合程度。在外观角色中维护了对子系统对象的引用，client可以通过外观角色来间接调用子系统对象的业务方法，而无须与子系统对象直接交互。

完整解决方案：

//文件读取子系统  
class FileReader{  
 public String read(String fileNameSrc) {  
 System.out.println("读取文件，获取明文: ");  
 StringBuffer sb = new StringBuffer();  
 FileInputStream fis = null;  
 try {  
 fis = new FileInputStream(fileNameSrc);  
 int data;  
 while ((data = fis.read()) != -1) {  
 sb.append((char) data);  
 }  
 } catch (FileNotFoundException ex) {  
 ex.printStackTrace();  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 } finally {  
 if (null != fis) {  
 try {  
 fis.close();  
 } catch (Exception ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
  
 return sb.toString();  
 }  
}  
  
//数据加密类：子系统  
class CipherMachine{  
 public String encrypt(String plainText){  
 System.out.println("数据加密，将明文转换为密文: ");  
 String es="";  
 for(int i=0;i<plainText.length();i++){  
 String c=String.valueOf(plainText.charAt(i)%7);  
 es+=c;  
 }  
  
 System.out.println(es);  
 return es;  
 }  
}  
  
class FileWrier{  
 public void write(String encryptStr,String fileNameDes){  
 System.out.println("保存密文，写入文件: ");  
 FileOutputStream fos=null;  
 try{  
 fos=new FileOutputStream(fileNameDes);  
 fos.write(encryptStr.getBytes());  
 }catch(FileNotFoundException ex){  
 ex.printStackTrace();  
 }catch(IOException ex){  
 ex.printStackTrace();  
 }finally{  
 if(null!=fos){  
 try{  
 fos.close();  
 }catch(Exception ex){  
 ex.printStackTrace();  
 }  
 }  
 }  
 }  
}  
  
//加密外观类：外观类  
class EncryptFacade{  
 private FileReader reader;  
 private CipherMachine cipher;  
 private FileWrier writer;  
  
 public EncryptFacade(){  
 this.reader=new FileReader();  
 this.cipher=new CipherMachine();  
 this.writer=new FileWrier();  
 }  
  
 //调用其对象的业务方法  
 public void fileEncrypt(String fileNameSrc,String fileNameDes){  
 String plainText=this.reader.read(fileNameSrc);  
 String encryptStr=this.cipher.encrypt(plainText);  
 this.writer.write(encryptStr,fileNameDes);  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 EncryptFacade ef=new EncryptFacade();  
 ef.fileEncrypt("origin.txt","encrypt.txt");  
 }  
}

其实就是对多个固定的操作进行了进一步的封装。

抽象外观类的引入：

在标准的外观模式结构图中(即上面的代码实现)，如果需要增加或者删除字系统就需要对原有的代码进行修改，不符合开闭原则。可以引入抽象外观类来部分解决此问题。在引入抽象外观类后，client可以针对抽象外观类进行编程，对于新的业务需求，不需要修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的字系统对象，来达到不修改任何源代码而更换外观的目的。

class NewCipherMachine{  
 public String encrypt(String plainText){  
 //具体加密算法  
 return null;  
 }  
}  
  
abstract class AbstractEncryptFacade{  
 abstract public void fileEncrypt(String fileNameSrc,String fileNameDes);  
}  
  
//新加密外观类，具体外观类  
class NewEncyptFacade extends AbstractEncryptFacade{  
 private FileReader reader;  
 private NewCipherMachine cipher;  
 private FileWrier wrier;  
  
 public NewEncyptFacade(){  
 this.reader=new FileReader();  
 this.cipher=new NewCipherMachine();  
 this.wrier=new FileWrier();  
 }  
  
 @Override  
 public void fileEncrypt(String fileNameSrc, String fileNameDes) {  
 String plainText= this.reader.read(fileNameSrc);  
 String encryptStr=this.cipher.encrypt(plainText);  
 this.wrier.write(encryptStr,fileNameDes);  
 }  
}  
  
class Client2{  
 public static void main(String[] args) {  
 AbstractEncryptFacade ef=new NewEncyptFacade(); //面向抽象外观编程  
 ef.fileEncrypt("origin.txt","encrypt.txt");  
 }  
}

注意：

1. 在很多情况下为了节约系统资源，系统中只需要一个外观类的实例，换言之外观类可以是一个单例。当然单例是对具体的外观类实现为单例。
2. 在一个系统中可以设计多个外观类，每个外观类都负责和一些特定的子系统交互，向client提供相应的业务功能。

总结

外观模式并不给系统增加新的系统功能，它仅仅是简化系统迪接口。绝大多数B/S系统都有一个首页或者导航页面，大多数的C/S系统都提供菜单或者工具栏。它们充当的就是外观角色。此外，所有涉及与多个业务对象交互的场景都可以考虑使用外观模式进行重构。

* 主要优点

1. 降低耦合性
2. 只是提供了一个访问子系统的统一入口，并不影响client直接使用子系统类。（可以将client的入口和子系统放在不同的包内，然后将子系统做成包级私有的来达到只允许client通过外观类访问子系统的目的）

* 主要缺点
* 使用场景

1. 当要为访问一系列复杂的子系统提供一个简单入口时
2. Client与多个子系统之间存在很大的依赖性
3. 在层次化的结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

第十四章 实现对象的复用——享元模式

享元模式通过共享技术实现相同或相似对象的重用，在逻辑上每一个出现的字符都有一个对象与之对应，然而在物理上它们却共享同一个享元对象。

在享元模式中，存储这些共享实例对象的地方成为享元池。可以针对每一个不同的字符创建一个享元对象，将其放在享元池中，需要时再从享元池中取出。

享元模式以共享的方式高效的支持大量细粒度对象的重用。享元对象能做到共享的关键是区分了内部状态和外部状态。

1. 内部状态：是存储在享元对象内部并且不会随环境改变而改变的状态，内部状态可以共享
2. 外部状态时随环境改变而改变的、不可以共享的状态。

通过向取出的对象注入不同的外部状态，可以得打一系列相似的对象，而这些对象在内存中实际只存储一份。

享元模式定义：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小。可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度的对象，因此它又称为轻量级模式，是一种对象结构型模式。

享元模式较为复杂，一般结合工厂模式一起使用，在其结构中包含了一个享元工厂类。

1. Flyweight 抽象享元类
   1. 通常是一个接口或者抽象类，在抽象享元类中声明了具体享元类公共的方法，这些方法可以向外界提供享元对象的内部数据（内部状态），同时也可以通过这些方法来设置外部数据（外部状态）。
2. ConcreteFlyweight 具体享元类
   1. 实现了享元抽象类，其实例成为享元对象
   2. 通常可以结合单例模式来设计具体享元类，为每一个具体享元类提供唯一的享元对象
3. UnsharedConcreteFlyweight 非共享具体享元类
   1. 不能被共享的子类可以设计为非共享具体享元类
   2. 当需要一个非共享具体享元类的对象时可以直接通过实例化构建
4. FlyweightFactory 享元工厂类
   1. 享元工厂类用于创建并管理享元对象
   2. 针对抽象享元类编程
   3. 将各种类型的具体享元对象存储在一个享元池中
   4. 当client请求一个具体享元对象时，享元工厂提供一个存储在享元池中已创建的实例或者创建一个新的实例（如果不存在的话），返回新创建的实例并将其存储在享元池中。

在享元模式中引入享元工厂类，它的作用在于提供一个用于存储享元对象的享元池。典型的享元工厂类的代码如下：

class Flyweight{  
 //内部状态intrinsicState作为成员变量，同一个享元对象其内部状态是一致的  
 private String intrinsicState;  
  
 public Flyweight(String intrinsicState){  
 this.intrinsicState=intrinsicState;  
 }  
  
 //外部状态extrinsicState在使用时由外部设置，不保存在享元对象中，即使是  
 //同一个对象，在每次调用时可以传入不同外部状态  
 public void operation(String extrinsicState){  
 //...  
 }  
}  
  
class FlyweightFactory{  
 private Map<String,Flyweight> flyweights=new HashMap<String, Flyweight>();  
  
 public Flyweight getFlyweight(String key){  
 if(!this.flyweights.containsKey(key)){  
 this.flyweights.put(key,new Flyweight(""));  
 }  
  
 return this.flyweights.get(key);  
 }  
}

享元类的设计：通常将内部状态作为享元类的成员变量，而外部状态通过注入的方式添加到享元类中，见上面代码。

完整解决方案：

//围棋棋子类，抽象享元类  
abstract class IgoChessman{  
 abstract public String getColor();  
 public void display(){  
 System.out.println("棋子颜色："+this.getColor());  
 }  
}  
  
//黑色棋子类：具体享元类  
class BlackIgoChessman extends IgoChessman{  
 public String getColor(){  
 return "黑色";  
 }  
}  
  
//白色棋子类：具体享元类  
class WhiteIgoChessman extends IgoChessman{  
 public String getColor(){  
 return "白色";  
 }  
}  
  
//围棋棋子工厂类，享元工厂类，使用单例模式进行设计  
class IgoChessmanFactory{  
 private static IgoChessmanFactory instance=new IgoChessmanFactory();  
 private static Map<String,IgoChessman> igoChessmanMap;  
  
 private IgoChessmanFactory(){  
 igoChessmanMap=new Hashtable<String, IgoChessman>();  
  
 IgoChessman black,white;  
 black=new BlackIgoChessman();  
 white=new WhiteIgoChessman();  
  
 igoChessmanMap.put("b",black);  
 igoChessmanMap.put("w",white);  
 }  
  
 public static IgoChessmanFactory getInstance(){  
 return instance;  
 }  
  
 //通过key获取存储在HashTable中的享元对象  
 public IgoChessman getIgoChessman(String key){  
 return igoChessmanMap.get(key);  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 IgoChessman black1,black2,black3,white1,white2;  
 IgoChessmanFactory factory;  
  
 factory=IgoChessmanFactory.getInstance();  
  
 black1=factory.getIgoChessman("b");  
 black2=factory.getIgoChessman("b");  
 black3=factory.getIgoChessman("b");  
  
 white1=factory.getIgoChessman("w");  
 white2=factory.getIgoChessman("w");  
  
 black1.display();  
 black2.display();  
 black3.display();  
 white1.display();  
 white2.display();  
 }  
}

带外部状态的解决方案:

为了描述棋子显示在棋盘的不同位置，解决方法是将棋子的位置定义为棋子的一个外部状态，在需要时再进行设置。这里增加一个新类Coordinates坐标类。改进后的代码如下：

class Coordinates{ //外部状态类  
 private int x;  
 private int y;  
  
 public Coordinates(int x,int y){  
 this.x=x;  
 this.y=y;  
 }  
  
 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;  
 }  
}  
  
//围棋棋子类，抽象享元类  
abstract class IgoChessman{  
 abstract public String getColor();  
 public void display(Coordinates coord){//动态设置外部状态  
 System.out.println("棋子颜色："+this.getColor()+", 棋子位置: "+coord.getX()+", "+coord.getY());  
 }  
}  
  
//黑色棋子类：具体享元类  
class BlackIgoChessman extends IgoChessman{  
 public String getColor(){  
 return "黑色";  
 }  
}  
  
//白色棋子类：具体享元类  
class WhiteIgoChessman extends IgoChessman{  
 public String getColor(){  
 return "白色";  
 }  
}  
  
//围棋棋子工厂类，享元工厂类，使用单例模式进行设计  
class IgoChessmanFactory{  
 private static IgoChessmanFactory instance=new IgoChessmanFactory();  
 private static Map<String,IgoChessman> igoChessmanMap;  
  
 private IgoChessmanFactory(){  
 igoChessmanMap=new Hashtable<String, IgoChessman>();  
  
 IgoChessman black,white;  
 black=new BlackIgoChessman();  
 white=new WhiteIgoChessman();  
  
 igoChessmanMap.put("b",black);  
 igoChessmanMap.put("w",white);  
 }  
  
 public static IgoChessmanFactory getInstance(){  
 return instance;  
 }  
  
 //通过key获取存储在HashTable中的享元对象  
 public IgoChessman getIgoChessman(String key){  
 return igoChessmanMap.get(key);  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 IgoChessman black1,black2,black3,white1,white2;  
 IgoChessmanFactory factory;  
  
 factory=IgoChessmanFactory.getInstance();  
  
 black1=factory.getIgoChessman("b");  
 black2=factory.getIgoChessman("b");  
 black3=factory.getIgoChessman("b");  
 System.out.println("black1==black2? "+(black1==black2));//true  
  
 white1=factory.getIgoChessman("w");  
 white2=factory.getIgoChessman("w");  
 System.out.println("white1==white2? "+(white1==white2));//true  
  
 black1.display(new Coordinates(1,2));  
 black2.display(new Coordinates(2,3));  
 black3.display(new Coordinates(3,4));  
 white1.display(new Coordinates(5,6));  
 white2.display(new Coordinates(6,7));  
 }  
}

单纯享元模式和复合享元模式：

标准的享元模式结构中既包含可以共享的具体享元类，也包含不可以共享的非共享具体享元类。在实际使用过程中具体分为两种：

1. 单纯享元类
   1. 所有的享元类都是可以共享的，不存在不可共享的具体享元类
2. 复合享元类
   1. 将一些单纯享元对象使用组合模式加以组合，还可以形成复合享元对象（即使用日容器包含多个单纯享元对象）
   2. 这样的复合享元对象本身不能共享，但是他们可以分解成单纯享元对象，而后者则可以共享

注意：

享元模式通常和其他的几种模式一起联用，常见的联用方式如下：

1. 享元模式+工厂模式：工厂模式提供方法用于返回享元对象
2. 享元模式+工厂模式+单例模式：通常只有唯一一个工厂，可以设置为单例
3. 享元模式+组合模式：统一对多个享元对象设置外部状态

Jdk的String类即使用了享元模式设计

总结：

当系统中存在大量相同或者相似的对象时，享元模式是一种较好的解决方案。享元模式以节约内存提高性能为出发点。

* 主要优点

1. 可以极大地减少内存中对象的数量。
2. 内部状态和外部状态

* 主要缺点

1. 系统设计变得复杂，需要区分内部状态和外部状态

* 适用场景

1. 一个系统存有大量相同或相似的对象，造成内存的大量耗费
2. 对象的大部分状态都可以外部化，可以将外部化状态传入到对象中
3. 享元池的设计满足多次重复使用享元对象

第15章 对象的间接访问——代理模式

代理模式为对象的访问控制提供了一种设计方案，而且它具有多种不同的类型，应用相当广泛。

在代理模式中引入了一个新的代理对象，代理对象可以在client和目标对象之间起到中介的作用。

代理模式的定义：给某个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式是一种对象结构型模式。

代理模式的结构很简单，其核心是代理类，为了让client能够一致性的对待真实对象和代理对象，在代理模式中引入了抽象层，结构如下：

1. Subject 抽象主题角色
   1. 声明了真实主题和代理主题的共同接口，使得在任何使用真实主题的地方都可以使用代理主题
   2. Client通常需要针对抽象主题角色进行编程
2. Proxy 代理主题角色
   1. 包含了对真实主题的引用
   2. 提供了与真实主题角色相同的接口，以便在任何时候都可以替代真实主题
   3. 代理主题角色还可以控制对真实主题角色的引用，负责在需要的时候创建和删除真实主题角色，并约束真实主题角色的使用
3. RealSubject 真实主题角色
   1. 定义了代理角色所代表的真实对象，在真实主题角色中实现了真实的业务操
   2. client可以通过代理主题角色间接调用真实主题角色中定义的操作。

代理类的典型实现：

//抽象查询类：抽象主题类  
interface Subject{  
 void do();  
}

Class Proxy implements Subject{

Private Subject realSubject=new RealSubject();

private void preDo(){   
//before do  
}   
  
 Private void postDo(){

//after do  
 }

public void do(){  
 preDo();  
 realSubject.do();  
 postDo();  
}  
}  
  
Class RealSubject implements Subject{  
 public void do(){  
 //真实角色实现  
}  
}

代理模式的进一步细分：

1. 远程代理：为一个位于不同地址空间对象提供一个本地代理，这个不同地址空间可以在同一台主机上，也可以在另一台主机上。
2. 虚拟代理
3. 保护代理：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限
4. 缓冲代理：为某个目标操作的结果提供临时的存储空间，以便多个client可以共享这些结果。
5. 智能引用代理：当一个对象被应用时，提供一些额外的操作，例如将对象被调用的次数记录下来

代理模式和装饰器模式在实现时有些类似，但是代理模式主要给真实主题类增加一些全新的职责，例如：权限控制、缓冲处理、智能引用和远程访问等。这些职责与原有的职责不属于同一个问题域。而装饰器模式是对原有职责的扩展，这些职责属于同一问题域。

代理模式和装饰器模式的目的也不相同，前者是控制对对象的访问，而后者是为对象动态的增加功能。

完整的解决方案：

//抽象查询类：抽象主题类  
interface Searcher{  
 String doSearch(String userId,String keyword);  
}  
  
//身份验证类：业务类  
class AccessValidator{  
 //模拟实现登陆验证  
 public boolean validate(String userId){  
 System.out.println("在数据库中验证用户 "+userId+" 是否合法");  
 if(userId.equalsIgnoreCase("userName")){  
 System.out.println("userId: "+userId+" 登陆成功");  
 return true;  
 }else{  
 System.out.println("userId: "+userId+" 登陆失败");  
 return false;  
 }  
 }  
}  
  
//日志记录类：业务类  
class Logger{  
 //模拟日志记录  
 public void log(String userId){  
 System.out.println("更新数据库，用户："+userId+" 查询此数+1");  
 }  
}  
  
//具体查询类：真实主题  
class RealSearcher implements Searcher{  
 public String doSearch(String userId, String keyword) {  
 System.out.println("用户 "+userId+" 使用关键词 "+keyword+" 查询商务信息");  
 return "返回具体内容";  
 }  
}  
  
class ProxySearcher implements Searcher{  
 private Searcher searcher;  
 private AccessValidator validator;  
 private Logger logger;  
  
 public ProxySearcher(){  
 this.searcher=new RealSearcher();  
 this.validator=new AccessValidator();  
 this.logger=new Logger();  
 }  
  
 public boolean validate(String userId){  
 return this.validator.validate(userId);  
 }  
  
 public void log(String userId){  
 this.logger.log(userId);  
 }  
  
 public String doSearch(String userId, String keyword) {  
 if(validate(userId)){  
 String result=this.searcher.doSearch(userId,keyword);  
 this.log(userId);  
 return result;  
 }else{  
 return null;  
 }  
 }  
}  
  
class Client{  
 public static void main(String[] args) {  
 Searcher searcher=new ProxySearcher();  
 System.out.println(searcher.doSearch("user","keyword"));  
 System.out.println(searcher.doSearch("userName","keyword"));  
 }  
}

以上实例是保护代理和智能引用代理的实例。如果需要增加新的直接和功能只需要增加新的代理类和业务类，符合开闭原则。

* 远程代理（Remote Proxy）

典型的实例为远程调用RPC

远程业务对象在本地主机中有一个代理对象，该代理对象负责对远程业务对象的访问和业务通信，它对于client对象而言是透明的。Client只需要按照服务接口定义的方式直接与本地主机的代理对象交互即可。

在java中可以通过RMI机制来实现远程代理，其实现步骤如下：

1. Client发起一个请求，将请求转发至RMI client的Stub类
2. Stub类将请求接口、方法和参数等信息进行序列化
3. 将序列化后的流使用socket传输至server
4. Server接收到流后将其转发到相应的Skeleton类
5. Skeleton类将请求信息反序列化后调用实际的业务处理类
6. 业务处理类处理完毕后将结果返回给Skeleton类
7. Skeleton类将结果序列化，再次通过socket将流传送给client的stub
8. Stub在接收到流后进行反序列化，将反序列化后得到的java object对象返回给client调用者

* 虚拟代理

虚拟代理是一种常用的代理模式，对于一些占用系统资源较多或加载时间较长的对象，可以给这些对象提供一个虚拟代理。在真实对象创建成功之前虚拟代理扮演真实对象的替身。而当真实对象创建之后，虚拟代理将用户的请求转发给真实的对象。

使用场景：

1. 由于对象本身的复杂性或者网络等原因导致一个对象需要较长的加载时间。
2. 当一个对象的加载十分耗费系统资源的时候

静态代理和动态代理：

通常情况下，每一个代理类编译之后都会生成一个class文件，代理类所实现的接口和所代理的方法都被固定，这种代理称为静态代理。

动态代理：指的是让系统在运行时动态创建代理类，它是一种较为高级的代理模式，它在事务管理、AOP等领域都发挥了重要的作用。

Java动态代理实现相关类：

Proxy类：提供了用于创建动态代理类和实例对象的方法，它是所创建的动态代理类的父类。常用方法如下：

1. Public satic Class<?> getProxyClass(ClassLoader loader,Class<?>… interfaces);

该方法用于返回一个Class类型的代理类，在参数中需要提供类加载器并需要指定代理的接口数组（与真实主题类的接口列表一致）

1. Public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h);

该方法返回一个动态创建的代理类的实例，方法中第一个参数loader表示代理类的类加载器，第二个参数interfaces表示代理类所实现的接口列表（与真实主题类的接口列表一致），第三个参数h表示所指派的调用处理程序类。

InvocationHandler接口：

是代理处理程序类的实现接口，该接口作为代理实例的调用处理者的公共父类。在该接口中声明了如下方法：

Public Object invoke(Object proxy,Method method, Object[] args);

该方法用于处理对代理类实例的方法调用并返回相应的结果，当一个代理实例中的业务方法被调用时将自动调用该方法。Invoke方法包含3个参数，其中第一个参数proxy表示代理类的实例，第2个参数method表示需要代理的方法，第三个参数args表示大力方法的参数数组。

完整实例：

//抽象UserDAO:抽象主题角色  
interface IUserDAO{  
 boolean findUserById(String userId);  
}  
  
//抽象DocumentDAO:抽象主题角色  
interface IDocumentDAO{  
 boolean deleteDocumentById(String documentId);  
}  
  
//具体UserDAO类：真实角色  
class UserDAO implements IUserDAO{  
 public boolean findUserById(String userId) {  
 if(userId.equalsIgnoreCase("userName")){  
 System.out.println("查询ID为 "+userId+" 的用户信息成功");  
 return true;  
 }else{  
 System.out.println("查询ID为 "+userId+" 的用户信息失败");  
 return false;  
 }  
 }  
}  
  
class DocumentDAO implements IDocumentDAO{  
 public boolean deleteDocumentById(String documentId) {  
 if(documentId.equalsIgnoreCase("docId")){  
 System.out.println("删除ID为 "+documentId+" 的文档信息成功");  
 return true;  
 }else{  
 System.out.println("删除ID为 "+documentId+" 的文档信息失败");  
 return false;  
 }  
 }  
}  
  
//自定义请求处理程序类  
class DAOLogHandler implements InvocationHandler{  
 private Calendar calendar;  
 private Object object;  
 public DAOLogHandler(){}  
  
 public DAOLogHandler(Object o){  
 this.object=o;  
 }  
  
 //实现invoke方法，调用在真实主题类中定义的方法  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 beforeInvoke();  
 Object result=method.invoke(object,args);//转发调用  
 afterInvoke();  
 return result;  
 }  
  
 private void beforeInvoke(){  
 calendar=new GregorianCalendar();  
 int hour=calendar.get(Calendar.HOUR\_OF\_DAY);  
 int minute=calendar.get(Calendar.MINUTE);  
 int second=calendar.get(Calendar.SECOND);  
  
 String time=hour+":"+minute+":"+second;  
 System.out.println("调用时间 "+time);  
 }  
  
 private void afterInvoke(){  
 System.out.println("方法调用结束");  
 }  
}  
  
class Client2{  
 public static void main(String[] args) {  
 InvocationHandler handler=null;  
 IUserDAO userDAO=new UserDAO();  
 handler=new DAOLogHandler(userDAO);  
  
 IUserDAO proxy;  
 //动态创建代理对象，用于代理一个IUserDAO类型的真实主题对象  
 //当该接口中的任何方法被调用时都会触发handler的invoke方法  
 proxy=(IUserDAO) Proxy.newProxyInstance(IUserDAO.class.getClassLoader(),new Class[]{IUserDAO.class},handler);  
 proxy.findUserById("userName");//调用代理对象代理的业务方法  
 System.out.println("--------------------------------");  
 IDocumentDAO docDao=new DocumentDAO();  
 handler=new DAOLogHandler(docDao);  
 IDocumentDAO proxy2;  
 proxy2=(IDocumentDAO)Proxy.newProxyInstance(IDocumentDAO.class.getClassLoader(),new Class[]{IDocumentDAO.class},handler);  
 proxy2.deleteDocumentById("docId");  
 }  
}