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

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;  
 }