# 面向对象的六大原则

## 1.单一职责原则（SRP）类的定义

简单的说就是：一个类中应该是一组相关性很高的函数、数据的封装。两个不一样的功能不应该放在一个类中。

这个原则没有具体的划分界限，需要根据个人经验，具体业务逻辑而定。这也是优化代码的第一步。试想一下，如果所有的功能写在一个类里，那么这个类会越来越大，越来越复杂，越不易修改维护。那么根据功能，各自独立拆分出来，岂不是逻辑会清晰些。

## 开闭原则（OCP）继承多态

定义：软件中的对象（类、模块、函数等）应该对于扩展是开放的，但是对于修改是封闭的。

当软件需要变化时，我们应该尽量通过扩展的方式实现变化，而不是通过修改原有的代码来实现。因为直接的修改，可能会影响已有的正常代码。不利于出现错误时排除问题。当然实际开发中，修改原有代码与扩展代码是同时存在的。但应尽量以扩展为主。

## 3.里氏替换原则（LSP）继承多态

定义：所有引用父类的地方，必须能使用子类的对象。简单地说就是将父类替换为他的子类是不会出现问题，反之，未必可以。

那么里氏替换原则就是依赖于面向对象语言的继承与多态。核心原理是抽象。

这里列举一下继承的优缺点：

优点：

（1）代码重用，减少创建类的成本，每个子类都拥有父类的方法与属性。

（2）子类与父类基本相似，但与父类又有所区别。

（3）提高代码的可扩展性。

缺点：

（1）继承是侵入性的，只要继承就必须拥有父类所有的属性与方法。

（2）可能造成子类代码冗余、灵活性降低。

开闭原则和里氏替换原则是生死相依的、不离不弃的。他们都强调了抽象这一重要的特性。

## 4.依赖倒置原则（DIP）

定义：指代一种特定的解耦方式，使得高层次的模块不依赖于低层次的模块的实现细节的目的。他有一下几个关键点：

（1）高层模块不依赖于低层模块，应该都依赖其抽象。

（2）抽象不依赖细节。

（3）细节应依赖抽象。

解释：在Java中，抽象就是指接口或者抽象类，两者都是不能直接被实例化的；细节就是实现类，实现接口或者继承抽象类而产生的就是细节，也就是可以加上一个关键字new产生的对象。高层模块就是调用端，底层模块就是具体实现类。

依赖倒置原则在Java中的表现就是：模块间通过抽象发生，实现类之间不发生直接依赖关系，其依赖关系是通过接口或者抽象类产生的。如果类与类直接依赖细节，那么就会直接耦合，那么当修改时，就会同时修改依赖者代码，这样限制了可扩展性。

## 5.接口隔离原则（ISP）

定义：类间的依赖关系应该建立在最小的接口上，将庞大、臃肿的接口拆分成更小的、更具体的接口。目的是系统的解耦，从而更容易重构、更改和重新部署。

## 6.迪米特原则（LOD）

定义：一个类应该对自己需要耦合或者调用的类知道的最少，类的内部如何实现与调用者或者依赖者没有关系，调用者或依赖者只需知道他需要的方法，其他可以一概不管。这样使得系统具有更低的耦合与更好的可扩展性。

这六个原则，可以使我们在应用的后续升级、维护中更加方便、轻松应对。让我们的软件更加灵活。

# 单例模式

单例模式应该是日常使用最为广泛的一种模式了。他的作用是确保某个类只有一个实例，避免产生多个对象消耗过多的资源。比如对数据库的操作时，就可以使用单例模式。

1.各种单例

（1）饿汉模式

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){

}

public static Singleton getInstance() {

return instance;

}

}

这种写法是在类装载时就实例化instance，他避免了多线程的同步问题。但是不能保证有别的方式去装载，没有达到懒加载。

（2）懒汉模式（线程不安全）

public class Singleton {

private static Singleton instance;

private Singleton (){

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

达到了懒加载，但是在多线程不能正常工作。

（3）懒汉模式（线程安全）

public class Singleton {

private static Singleton instance;

private Singleton (){

}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法能够在多线程中很好的工作，但是每次调用getInstance方法都会进行同步，反应稍慢，还会造成不必要的开销，所以者这种不建议使用。

（4）DCL单例（双重检查锁定）

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){

}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

这种写法在getSingleton方法中对singleton进行了两次判空，第一次是为了不必要的同步，第二次是为了在null的情况下创建实例。我们会发现上面代码有一个volatile关键字，因为在这里会有DCL失效问题，原因是Java编译器允许处理器乱序执行。那么为了解决这个问题，在JDK1.5之后，具体化了volatile关键字，只要定义时加上他，可以保证执行的顺序，虽然会影响性能。这种方式第一次加载时会稍慢，在高并发环境会有缺陷，但是一般能够满足需求。

（5）静态内部类单例模式

public class Singleton {

private Singleton (){

}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

/\*\*

\*静态内部类

\*/

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

}

**这种是推荐使用的单例模式实现方式**。当第一次加载Singleton类时并不会初始化INSTANCE，只有在第一次调用getInstance方法时才会导致INSTANCE被初始化。这种方式不仅能够保证线程安全，也能保证单例对象的唯一性，同时也延长了单例的实例化。

（6）枚举单例

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。其实是饿汉模式的另一种写法。。

（7）使用容器实现单例模式

public class SingletonManager {

　　private static Map<String, Object> objMap = new HashMap<String,Object>();

　　private Singleton() { }

　　public static void registerService(String key, Objectinstance) {

　　　　if (!objMap.containsKey(key) ) {

　　　　　　objMap.put(key, instance) ;

　　　　}

　　}

　　public static ObjectgetService(String key) {

　　　　return objMap.get(key) ;

　　}

}

将多种单例类型注入到一个统一的管理类中，在使用时根据key获取对象对应类型的对象。这种方式使得我们可以管理多种类型的单例，并且在使用时可以通过统一的接口进行获取操作，降低了用户的使用成本，也对用户隐藏了具体实现，降低了耦合度。

## Android源码中的单例模式

在Android系统中，我们经常会通过Context获取系统级别的服务，如WindowsManagerService、ActivityManagerService等，更常用的是一个LayoutInflater的类，这些服务会在合适的时候以单例的形式注册在系统中，在我们需要的时候就通过Context的getSystemService(String name)获取。

## 总结

优点：

（1）由于单例模式在内存中只有一个实例，减少了内存开支，特别是一个对象需要频繁的创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显。

（2）单例模式可以避免对资源的多重占用，例如一个文件操作，由于只有一个实例存在内存中，避免对同一资源文件的同时操作。

（3）单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如，可以设计一个单例类，负责所有数据表的映射处理。

缺点：

（1）单例模式一般没有接口，扩展很困难，若要扩展，只能修改代码来实现。

（2）单例对象如果持有Context，那么很容易引发内存泄露。此时需要注意传递给单例对象的Context最好是Application Context。

# Builder模式

## 1.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。

## 2.使用场景

（1）相同的方法，不同的执行顺序，产生不同的事件结果时。

（2）多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时。

（3）产品类非常复杂，或者产品类中的调用顺序不同产生了不同的作用，这个使用建造者模式非常适合。

（4）当初始化一个对象特别复杂时，如参数多，且很多参数有默认值。

## 简单实现

|  |
| --- |
| public interface Builder {  　　//创建部件A　　比如创建汽车车轮  　　void buildPartA();  　　//创建部件B 比如创建汽车方向盘  　　void buildPartB();  　　//创建部件C 比如创建汽车发动机  　　void buildPartC();  　　//返回最后组装成品结果 (返回最后装配好的汽车)  　　Product getResult();  }  //Director 类，负责制造  public class Director {  　　private Builder builder;  　　public Director( Builder builder ) {  　　　　this.builder = builder;  　　}  　　// 将部件partA partB partC最后组成复杂对象  　　//这里是将车轮 方向盘和发动机组装成汽车的过程  　　public void construct() {  　　　　builder.buildPartA();  　　　　builder.buildPartB();  　　　　builder.buildPartC();  　　}  }  public class ConcreteBuilder implements Builder {  　　Part partA, partB, partC;  　　public void buildPartA() {  　　　　//这里是具体如何构建partA的代码  　　};  　　public void buildPartB() {  　　　　//这里是具体如何构建partB的代码  　　};  　　 public void buildPartC() {  　　　　//这里是具体如何构建partB的代码  　　};  　　 public Product getResult() {  　　　　//返回最后组装成品结果  　　};  }  public interface Product { }//产品  public interface Part { }//部件  //调用  ConcreteBuilder builder = new ConcreteBuilder();  Director director = new Director( builder );  director.construct();  Product product = builder.getResult(); |

从上面可以看到由于Director封装了构建复杂的产品对象过程，对外隐藏了细节。

现实开发中，Director一般被省略。而直接使用一个Builder来进行对象的组装，这个Builder通常为**链式调用**，他是将setter方法返回自身。代码大致如下：

new TestBuilder().setA("A").setB("B").create();

## Android源码中的Builder模式

### AlertDialog.Builder

源码太长截取部分：

|  |
| --- |
| public static class Builder {  private final AlertController.AlertParams P;  private int mTheme;  /\*\*  \* Constructor using a context for this builder and the {@link AlertDialog} it creates.  \*/  public Builder(Context context) {  this(context, resolveDialogTheme(context, 0));  }  /\*\*  \* Constructor using a context and theme for this builder and  \* the {@link AlertDialog} it creates. The actual theme  \* that an AlertDialog uses is a private implementation, however you can  \* here supply either the name of an attribute in the theme from which  \* to get the dialog's style (such as {@link android.R.attr#alertDialogTheme}  \* or one of the constants  \* {@link AlertDialog#THEME\_TRADITIONAL AlertDialog.THEME\_TRADITIONAL},  \* {@link AlertDialog#THEME\_HOLO\_DARK AlertDialog.THEME\_HOLO\_DARK}, or  \* {@link AlertDialog#THEME\_HOLO\_LIGHT AlertDialog.THEME\_HOLO\_LIGHT}.  \*/  public Builder(Context context, int theme) {  P = new AlertController.AlertParams(new ContextThemeWrapper(  context, resolveDialogTheme(context, theme)));  mTheme = theme;  }  /\*\*  \* Returns a {@link Context} with the appropriate theme for dialogs created by this Builder.  \* Applications should use this Context for obtaining LayoutInflaters for inflating views  \* that will be used in the resulting dialogs, as it will cause views to be inflated with  \* the correct theme.  \*  \* @return A Context for built Dialogs.  \*/  public Context getContext() {  return P.mContext;  }  /\*\*  \* Set the title displayed in the {@link Dialog}.  \*  \* @return This Builder object to allow for chaining of calls to set methods  \*/  public Builder setTitle(CharSequence title) {  P.mTitle = title;  return this;//\*\*这里返回自身，类似的来设置各种参数。  }  // ......省略  /\*\*  \* Creates a {@link AlertDialog} with the arguments supplied to this builder. It does not  \* {@link Dialog#show()} the dialog. This allows the user to do any extra processing  \* before displaying the dialog. Use {@link #show()} if you don't have any other processing  \* to do and want this to be created and displayed.  \*/  public AlertDialog create() {  final AlertDialog dialog = new AlertDialog(P.mContext, mTheme, false);  P.apply(dialog.mAlert);//将P中的参数应用到dialog中的mAlert对象中  dialog.setCancelable(P.mCancelable);  if (P.mCancelable) {  dialog.setCanceledOnTouchOutside(true);  }  dialog.setOnCancelListener(P.mOnCancelListener);  dialog.setOnDismissListener(P.mOnDismissListener);  if (P.mOnKeyListener != null) {  dialog.setOnKeyListener(P.mOnKeyListener);  }  return dialog;  }  /\*\*  \* Creates a {@link AlertDialog} with the arguments supplied to this builder and  \* {@link Dialog#show()}'s the dialog.  \*/  public AlertDialog show() {  AlertDialog dialog = create();  dialog.show();  return dialog;  }  } |

上面有个AlertController.AlertParams 的成员参数P，我们在Builder设置的title，icon等都储存在他里面。在调用create时在P.apply使用。

## 总结

优点：

（1）良好的封装性，使用建造者模式可以使客户端不必知道产品内部组成细节。

（2）建造者独立，容易扩展。

缺点：

（1）会产生多余的Builder对象及Director对象，消耗内存。

# 原型模式

## 定义

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。被复制的实例就是“原型”，这个原型是可定制的。

## 2、使用场景

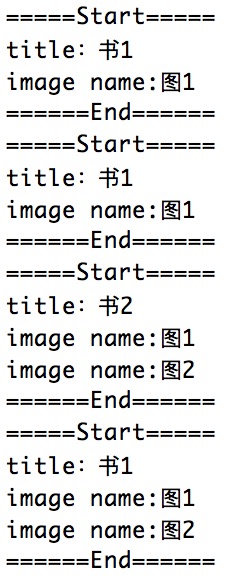
（1）类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗。   
（2）通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。   
（3）一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

## 3、简单实现

|  |
| --- |
| public class Book implements Cloneable{  private int price;  private String title;  private String content;  private ArrayList<String> image = new ArrayList<String>();  public Book() {  super();  }  public Book(int price, String title, String content) {  super();  this.price = price;  this.title = title;  this.content = content;  }  public ArrayList<String> getImage(){  return image;  }  public void addImage(String img){  this.image.add(img);  }  public int getPrice() {  return price;  }  public void setPrice(int price) {  this.price = price;  }  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public String getContent() {  return content;  }  public void setContent(String content) {  this.content = content;  }  @Override  protected Object clone() throws CloneNotSupportedException {  Book book = (Book)super.clone();  return book;  }  public void showBook(){  System.out.println("=====Start=====");  System.out.println("title："+title);  for(String img : image){  System.out.println("image name:"+img);  }  System.out.println("======End======");  }  } |

执行main方法：

|  |
| --- |
| public static void main(String[] args) {  try {  Book book1 = new Book(50,"书1","内容");  book1.addImage("图1");  book1.showBook();  Book book2 = (Book) book1.clone();  book2.showBook();  book2.setTitle("书2");  book2.addImage("图2");  book2.showBook();  book1.showBook();  } catch (CloneNotSupportedException e) {  e.printStackTrace();  }  } |



这里我们发现通过修改book2后，只是影响了book1的image，而没有改变title。这是为什么呢？这个我们暂时放下，先看看变化的image，很明显book1与book2是引用了同一地址，所以一方修改两边都改变。那么怎么解决？覆盖Object中的clone方法，实现深拷贝。同时我们称上面的原型模式为浅拷贝。

那么修改如下(其他不变)：

|  |
| --- |
| @Override  protected Object clone() throws CloneNotSupportedException {  Book book = (Book)super.clone();  book.image = (ArrayList<String>) this.image.clone();  return book;  } |

可以看出现在互不影响，这个叫做深拷贝。

接着上面的疑问，其实String类型在浅拷贝时和引用类型一样，没有单独复制，而是引用同一地址，因为String没有实现cloneable接口，也就是说只能复制引用。（这里我们可以查看源码可以看到，而ArrayList实现了cloneable接口）但是当修改其中的一个值的时候，会新分配一块内存用来保存新的值，这个引用指向新的内存空间，原来的String因为还存在指向他的引用，所以不会被回收，这样，虽然是复制的引用，但是修改值的时候，并没有改变被复制对象的值。

所以在很多情况下，我们可以把String在clone的时候和基本类型做相同的处理，只是在equals时注意一些就行了。

## Android源码中的原型模式

|  |
| --- |
| Uri uri = Uri.parse("smsto:110");  Intent intent = new Intent(Intent.ACTION\_SEND,uri);  intent.putExtra("sms\_body", "The SMS text");  //克隆  Intent intent2 = (Intent)intent.clone();  startActivity(intent2); |

## 5、总结

优点

（1）原型模式是在内存中二进制流的拷贝，要比直接new一个对象性能好很多，特别是要在一个循环体内产生大量对象时，原型模式可能更好的体现其优点。

（2）还有一个重要的用途就是保护性拷贝，也就是对某个对象对外可能是只读的，为了防止外部对这个只读对象的修改，通常可以通过返回一个对象拷贝的形式实现只读的限制。

缺点：

（1）这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的，在实际开发中应该注意这个潜在问题。优点是减少了约束，缺点也是减少了约束，需要大家在实际应用时考虑。

（2）通过实现Cloneable接口的原型模式在调用clone函数构造实例时并不一定比通过new操作速度快，只有当通过new构造对象较为耗时或者说成本较高时，通过clone方法才能够获得效率上的提升。

## 参考

[链接：Java克隆中String的特殊性](http://blog.csdn.net/naivesoft/article/details/6774550)

# 工厂方法模式

## 定义

定义一个用于创建对象的接口，让子类决定实例化那个类。

## 使用场景

在任何需生成复杂对象的地方，都可以使用工厂方法模式。复杂对象适合使用工厂模式，用new就可以完成创建的对象无需使用工厂模式。

## 模式的简单实现

抽象产品类：

public abstract class Product {

/\*\*

\* 产品类的抽象方法

\* 由具体的产品类去实现

\* \*/

public abstract void method();

}

具体产品类A：

public class ConcreteProductA extends Product {

@Override

public void method() {

System.out.println("我是产品A");

}

}

具体产品类B：

public class ConcreteProductB extends Product {

@Override

public void method() {

System.out.println("我是产品B");

}

}

抽象工厂类：

public abstract class Factory {

/\*\*

\* 抽象工厂方法

\* 具体由子类实现

\*

\* @return 具体的产品对象

\* \*/

public abstract Product createProduct();

}

具体工厂类：

public class ConcreteFactory extends Factory {

/\*\*

\* 具体工厂类

\* \*/

@Override

public Product createProduct() {

return new ConcreteProductA();

}

}

调用：

public class Client {

public static void main(String[] args) {

Factory factory = new ConcreteFactory();

Product product = factory.createProduct();

product.method();

}

}

结果：

我是产品A

如果想得到ConcreteProductB的实例，更改ConcreteFactory逻辑：

public class ConcreteFactory extends Factory {

/\*\*

\* 具体工厂类

\* \*/

@Override

public Product createProduct() {

//return new ConcreteProductA();

return new ConcreteProductB();

}

}

上面这种方式可能不是很灵活，可以利用反射实现，如下：

public abstract class Factory {

/\*\*

\* 抽象工厂方法

\* 具体由子类实现

\*

\* @param clz 产品对象类类型

\*

\* @return 具体的产品对象

\* \*/

public abstract <T extends Product> T createProduct(Class<T> clz);

}

具体工厂类修改：

public class ConcreteFactory extends Factory {

/\*\*

\* 具体工厂类

\* \*/

@SuppressWarnings("unchecked")

@Override

public <T extends Product> T createProduct(Class<T> clz) {

Product product = null;

try {

product = (Product) Class.forName(clz.getName()).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return (T)product;

}

}

Client实现：

public class Client {

public static void main(String[] args) {

Factory factory = new ConcreteFactory();

Product product = factory.createProduct(ConcreteProductB.class);

product.method();

}

}

如果我们的工厂只有一个，我们可以进行简化，这种方式又称为简单工厂模式或者静态工厂模式。如下：

public class Factory {

/\*\*

\* 抽象工厂方法

\* 具体由子类实现

\*

\* @param clz 产品对象类类型

\*

\* @return 具体的产品对象

\* \*/

public static <T extends Product> T createProduct(Class<T> clz) {

Product product = null;

try {

product = (Product) Class.forName(clz.getName()).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return (T)product;

}

}

## Android源码中的工厂方法模式

1.Activity的各种生命周期

2.ArrayList和HashSet

## 总结

优点：

1.工厂方法模式完全符合设计原则，降低了对象之间的耦合。高层模块只需要知道产品的抽象类，其他的实现都不需要关心。

2.良好的封装性，代码结构清晰。扩展性好。

缺点：

每次我们为工厂方法模式添加新的产品时就要编写一个新的产品类。同时还要引入抽象层，这必然会导致类结构的复杂化，所以，在某些情况比较简单时，是否要使用工厂模式，需要设计者权衡利弊了。

# 6、抽象工厂模式

## 定义

为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定他们的具体实现类。

## 使用场景

一个对象族（或是一组没有任何关系的对象）都有相同的约束，则可以使用抽象工厂模式。例如一个文本编辑器和一个图片处理器，都是软件实体，但是Linix下的文本编辑器和WINDOWS下的文本编辑器虽然功能和界面都相同，但是代码实现是不同的，图片处理器也是类似情况，也就是具有了共同的约束条件：操作系统类型，于是我们可以使用抽象工厂模式，产生不同操作系统下的编辑器和图片处理器。

## 简单实现

以车厂生产汽车零部件为例,A、B两家车厂分别生产不同的轮胎、发动机、制动系统。虽然生产的零件不同，型号不同。但是根本上都有共同的约束，就是轮胎、发动机、制动系统。

轮胎相关类：

public interface ITire {

/\*\*

\* 轮胎

\*/

void tire();

}

1

2

3

4

5

6

public class NormalTire implements ITire{

@Override

public void tire() {

System.out.println("普通轮胎");

}

}

1

2

3

4

5

6

7

public class SUVTire implements ITire{

@Override

public void tire() {

System.out.println("越野轮胎");

}

}

1

2

3

4

5

6

发动机相关类：

public interface IEngine {

/\*\*

\*发动机

\*/

void engine();

}

1

2

3

4

5

6

public class DomesticEngine implements IEngine{

@Override

public void engine() {

System.out.println("国产发动机");

}

}

1

2

3

4

5

6

public class ImportEngine implements IEngine{

@Override

public void engine() {

System.out.println("进口发动机");

}

}

1

2

3

4

5

6

制动系统相关类：

public interface IBrake {

/\*\*

\*制动系统

\*/

void brake();

}

1

2

3

4

5

6

public class NormalBrake implements IBrake{

@Override

public void brake() {

System.out.println("普通制动");

}

}

1

2

3

4

5

6

public class SeniorBrake implements IBrake{

@Override

public void brake() {

System.out.println("高级制动");

}

}

1

2

3

4

5

6

7

抽象车厂类：

public abstract class CarFactory {

/\*\*

\* 生产轮胎

\*

\* @return 轮胎

\* \*/

public abstract ITire createTire();

/\*\*

\* 生产发动机

\*

\* @return 发动机

\* \*/

public abstract IEngine createEngine();

/\*\*

\* 生产制动系统

\*

\* @return 制动系统

\* \*/

public abstract IBrake createBrake();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

A车厂：

public class AFactory extends CarFactory{

@Override

public ITire createTire() {

return new NormalTire();

}

@Override

public IEngine createEngine() {

return new DomesticEngine();

}

@Override

public IBrake createBrake() {

return new NormalBrake();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

B车厂：

public class BFactory extends CarFactory{

@Override

public ITire createTire() {

return new SUVTire();

}

@Override

public IEngine createEngine() {

return new ImportEngine();

}

@Override

public IBrake createBrake() {

return new SeniorBrake();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

客户类：

public class Client {

public static void main(String[] args) {

//A车厂

CarFactory factoryA = new AFactory();

factoryA.createTire().tire();

factoryA.createEngine().engine();

factoryA.createBrake().brake();

System.out.println("---------------");

//B车厂

CarFactory factoryB = new BFactory();

factoryB.createTire().tire();

factoryB.createEngine().engine();

factoryB.createBrake().brake();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

结果：

普通轮胎

国产发动机

普通制动

------------------

越野轮胎

进口发动机

高级制动

1

2

3

4

5

6

7

可以看出上面模拟了两个车厂，如果有了C厂、D厂，各自厂家生产的零部件型号种类又不相同，那么我们创建的类文件就会翻倍。这也是抽象工厂模式的一个弊端，所以实际开发中要权衡使用。

4.区别

上一章有介绍了工厂模式，那么他们的区别是什么？抽象工厂模式是工厂方法模式的升级版本。对比如下：

工厂方法模式 抽象工厂模式

只有一个抽象产品类 有多个抽象产品类

具体工厂类只能创建一个具体产品类的实例 具体工厂类能创建多个具体产品类的实例

5.Android源码中的实现

抽象工厂模式在Android源码中使用较少，因为很少会出现多个产品种类的情况，大部分使用工厂方法模式即可解决。

1.MediaPlayer

MediaPlayer Factory分别会生成4个不同的MediaPlayer基类：StagefrightPlayer、NuPlayerDriver、MidiFile和TestPlayerStub，四者均继承于MediaPlayerBase。

6.总结

优点：

分离接口与实现，客户端使用抽象工厂来创建需要的对象，而客户端根本就不知道具体的实现是谁，客户端只是面向产品的接口编程而已，使其从具体的产品实现中解耦，同时基于接口与实现分离，使抽象该工厂方法模式在切换产品类时更加灵活、容易。

缺点：

一是对类文件的爆炸性增加，二是不太容易扩展新的产品类。

参考

# 策略模式

通常如果一个问题有多个解决方案时，最简单的就是利用if-else或者switch-case方式根据不同的情景选择不同的解决方案，但是这样耦合性太高 、代码臃肿、难以维护等。这时就可以使用策略模式来解决。

## 定义

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

## 使用场景

1.针对同一类型问题的多种处理方式，仅仅是具体行为有差别时。

2.需要安全地封装多种同一类型的操作时。

3.出现同一抽象类有多个子类，而又需要使用if-else或者switch-case来选择具体子类时。

## 简单实现

需求：计算图书价格，初级会员没有折扣，中级会员打9折，高级会员打8折。如果一般写法，应该是if-else判断他是什么级别的会员，在计算相应的折扣。下面使用策略模式来进行实现。

抽象折扣类

public interface MemberStrategy {

/\*\*

\* 计算图书的价格

\* @param booksPrice 图书的原价

\* @return 计算出打折后的价格

\*/

public double calcPrice(double booksPrice);

}

初级会员折扣类

public class PrimaryMemberStrategy implements MemberStrategy{

/\*\*

\* 初级会员折扣

\*/

@Override

public double calcPrice(double booksPrice) {

System.out.println("对于初级会员的没有折扣");

return booksPrice;

}

}

中级会员折扣类

public class IntermediateMemberStrategy implements MemberStrategy{

/\*\*

\* 中级会员折扣

\*/

@Override

public double calcPrice(double booksPrice) {

System.out.println("对于中级会员的折扣为10%");

return booksPrice \* 0.9;

}

}

高级会员折扣类

public class AdvancedMemberStrategy implements MemberStrategy{

/\*\*

\* 高级会员折扣

\*/

@Override

public double calcPrice(double booksPrice) {

System.out.println("对于高级会员的折扣为20%");

return booksPrice \* 0.8;

}

}

价格类

public class Price {

//持有一个具体的策略对象

private MemberStrategy strategy;

/\*\*

\* 构造函数，传入一个具体的策略对象

\* @param strategy 具体的策略对象

\*/

public Price(MemberStrategy strategy){

this.strategy = strategy;

}

/\*\*

\* 计算图书的价格

\* @param booksPrice 图书的原价

\* @return 计算出打折后的价格

\*/

public double quote(double booksPrice){

return this.strategy.calcPrice(booksPrice);

}

}

客户端

public class Client {

public static void main(String[] args) {

//选择并创建需要使用的策略对象

MemberStrategy strategy1 = new AdvancedMemberStrategy();

//创建环境

Price price = new Price(strategy1);

//计算价格

double quote = price.quote(300);

System.out.println("图书的最终价格为：" + quote);

}

}

结果

对于高级会员的折扣为20%

图书的最终价格为：240.0

## 策略模式和工厂模式的区别

工厂模式 策略模式

创建型的设计模式 行为型的设计模式

关注对象创建 关注行为的选择

黑盒子（无需知道具体的实现过程） 白盒子（知道具体的实现过程）

## Android源码中的策略模式实现

### 1.时间插值器（TimeInterpolator）

LinearInterpolator、AccelerateInterpolator、CycleInterpolator等实现Interpolator，通过getInterpolator（float input）获取当前的时间百分比，以此来计算动画的属性值。

## 6.总结

策略模式主要用来分离算法，在相同的行为抽象下有不同的具体实现策略。这个模式很好地演示了开闭原则，也就是定义抽象，注入不同的实现，从而达到很好的可扩展性。

优点：

1.结构清晰明了、使用简单直观。

2.耦合度相对而言较低，扩展方便。

3.操作封装也更为彻底，数据更为安全。

缺点：

1.随着策略的增加，子类也会变得繁多。

## 参考

链接：

1.《JAVA与模式》之策略模式

2. 策略模式和工厂模式的区别

# 状态模式

## 定义

状态模式中的行为是由状态来决定，不同的状态下有不同的行为。当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

## 使用场景

1.一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为。

2.代码中包含大量与对象状态有关的条件语句，例如，一个操作中含有大量的多分支语句，且这些分支依赖于该对象的状态。

## 简单实现

实现效果：首先将电视的状态分为开机与关机状态，开机时可以通过遥控器实现频道切换和调节音量，但是关机时，这些操作都会失效。

首先是普通的实现方法：

public class TVController {

//开机状态

private final static int POWER\_ON = 1;

//关机状态

private final static int POWER\_OFF = 2;

//默认状态

private int mState = POWER\_OFF;

public void powerOn(){

if(mState ==POWER\_OFF){

System.out.println("电视开机了");

}

mState = POWER\_ON;

}

public void powerOff(){

if(mState ==POWER\_ON){

System.out.println("电视关机了");

}

mState = POWER\_OFF;

}

public void nextChannel(){

if(mState ==POWER\_ON){

System.out.println("下一频道");

}else{

System.out.println("没有开机");

}

}

public void prevChannel(){

if(mState ==POWER\_ON){

System.out.println("上一频道");

}else{

System.out.println("没有开机");

}

}

public void turnUp(){

if(mState ==POWER\_ON){

System.out.println("调高音量");

}else{

System.out.println("没有开机");

}

}

public void turnDown(){

if(mState ==POWER\_ON){

System.out.println("调低音量");

}else{

System.out.println("没有开机");

}

}

}

可以看到，每次执行通过判断当前状态来进行操作，部分的代码重复，假设状态和功能增加，就会越来越难以维护。这时可以使用状态模式，如下：

电视的操作

/\*\*

\* 电视状态接口，定义了电视的操作函数

\*

\* \*/

public interface TVState {

public void nextChannel();

public void prevChannel();

public void turnUp();

public void turnDown();

}

关机状态

/\*\*

\*

\* 关机状态，操作无结果

\*

\* \*/

public class PowerOffState implements TVState{

@Override

public void nextChannel() {

}

@Override

public void prevChannel() {

}

@Override

public void turnUp() {

}

@Override

public void turnDown() {

}

}

开机状态

/\*\*

\*

\* 开机状态，操作有效

\*

\* \*/

public class PowerOnState implements TVState{

@Override

public void nextChannel() {

System.out.println("下一频道");

}

@Override

public void prevChannel() {

System.out.println("上一频道");

}

@Override

public void turnUp() {

System.out.println("调高音量");

}

@Override

public void turnDown() {

System.out.println("调低音量");

}

}

电源操作接口

/\*\*

\* 电源操作接口

\*

\* \*/

public interface PowerController {

public void powerOn();

public void powerOff();

}

电视遥控器

/\*\*

\* 电视遥控器

\*

\* \*/

public class TVController implements PowerController{

TVState mTVState;

public void setTVState(TVState mTVState){

this.mTVState = mTVState;

}

@Override

public void powerOn() {

setTVState(new PowerOnState());

System.out.println("开机了");

}

@Override

public void powerOff() {

setTVState(new PowerOffState());

System.out.println("关机了");

}

public void nextChannel(){

mTVState.nextChannel();

}

public void prevChannel(){

mTVState.prevChannel();

}

public void turnUp(){

mTVState.turnUp();

}

public void turnDown(){

mTVState.turnDown();

}

}

调用：

public class Client {

public static void main(String[] args) {

TVController tvController = new TVController();

//设置开机状态

tvController.powerOn();

//下一频道

tvController.nextChannel();

//调高音量

tvController.turnUp();

//关机

tvController.powerOff();

//调低音量，此时不会生效

tvController.turnDown();

}

}

结果

开机了

下一频道

调高音量

关机了

可以看出，状态模式将这些行为封装到状态类中，在进行操作时将这些功能转发给状态对象，不同的状态有不同的实现，去除了重复了if-else语句，这正是状态模式的精髓所在。

## 与策略模式的区别

状态模式与策略模式的结构几乎是一样的，就像是孪生兄弟。但是他们的目地、本质不一样。状态模式的行为是平行的、不可替换的，策略模式的行为是彼此独立的、可相互替换的。状态模式，通常是自我控制状态的改变。而策略模式，是由外部指定使用什么样的策略。

## 5.Android实战中的使用

登录系统，根据用户是否登录，判断事件的处理方式。

Wi-Fi管理，在不同的状态下，WiFi的扫描请求处理不一。

## 总结

### 1.优点

将所有与一个特定的状态相关的行为都放入一个状态对象中，它提供了一个更好的方法来组织与特定状态相关的代码，将繁琐的状态判断转换成结构清晰的状态类族，在避免代码膨胀的同时也保证了可扩展性与可维护性。

### 2.缺点

状态模式的使用必然会增加系统类和对象的个数。

# 责任链模式

## 定义

责任链模式是行为型设计模式之一，它使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

## 使用场景

1.多个对象可以处理同一请求，但具体由哪个对象处理则在运行时动态决定。

2.在请求处理者不明确的情况下向多个对象中的一个提交请求。

3.需要动态指定一组对象处理请求。

## 简单实现

这个例子我觉得很贴切。我们在公司有各种原因需要报销费用，首先我们要找我们的上级领导去审批，报销额度如果在领导的权限范围内，那就审批通过，否则领导在找自己的上级去审批，以此类推。

抽象领导类

public abstract class Leader {

/\*\*

\* 上级领导处理者

\*/

protected Leader nextHandler;

/\*\*

\* 处理报账请求

\*

\* @param money 能批复的报账额度

\*

\*/

public final void handleRequest(int money){

System.out.println(getLeader());

if(money <=limit()){

handle(money);

}else{

System.out.println("报账额度不足，提交领导");

if(null != nextHandler){

nextHandler.handleRequest(money);

}

}

}

/\*\*

\* 自身能批复的额度权限

\*

\* @return 额度

\*/

public abstract int limit();

/\*\*

\* 处理报账行为

\*

\* @param money 具体金额

\*/

public abstract void handle(int money);

/\*\*

\* 获取处理者

\*

\* @return 处理者

\*/

public abstract String getLeader();

}

组长（额度1000）：

public class GroupLeader extends Leader{

@Override

public int limit() {

return 1000;

}

@Override

public void handle(int money) {

System.out.println("组长批复报销"+ money +"元");

}

@Override

public String getLeader() {

return "当前是组长";

}

}

主管（额度5000）：

public class Director extends Leader{

@Override

public int limit() {

return 5000;

}

@Override

public void handle(int money) {

System.out.println("主管批复报销"+ money +"元");

}

@Override

public String getLeader() {

return "当前是主管";

}

}

经理（额度10000）：

public class Manager extends Leader{

@Override

public int limit() {

return 10000;

}

@Override

public void handle(int money) {

System.out.println("经理批复报销"+ money +"元");

}

@Override

public String getLeader() {

return "当前是经理";

}

}

老板（额度…）:

public class Boss extends Leader{

@Override

public int limit() {

return Integer.MAX\_VALUE;

}

@Override

public void handle(int money) {

System.out.println("老板批复报销"+ money +"元");

}

@Override

public String getLeader() {

return "当前是老板";

}

}

发起申请：

public class Client {

public static void main(String[] args) {

//构造各个领导对象

GroupLeader groupLeader = new GroupLeader();

Director director = new Director();

Manager manager = new Manager();

Boss boss = new Boss();

//设置上级领导处理者对象

groupLeader.nextHandler = director;

director.nextHandler = manager;

manager.nextHandler = boss;

//发起报账申请

groupLeader.handleRequest(8000);

}

}

结果：

当前是组长

报账额度不足，提交领导

当前是主管

报账额度不足，提交领导

当前是经理

经理批复报销8000元

责任链模式非常灵活，请求的发起可以从责任链的任何一个节点开始，也可以改变内部的传递规则。比如主管不在，我们完全可以跨过主管直接从组长那里转到经理。

对于责任链中的一个处理者对象，有两个行为。一是处理请求，二是将请求传递到下一节点，不允许某个处理者对象在处理了请求后又将请求传送给上一个节点的情况。

对于一条责任链来说，一个请求最终只有两种情况。一是被某个处理对象所处理，另一个是所有对象均未对其处理，对于前一种情况我们称为纯的责任链模式，后一种为不纯的责任链。实际中大多为不纯的责任链。

## Android源码中的责任链模式

1.View事件的分发处理

ViewGroup事件投递的递归调用就类似于一条责任链，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体体现在View的onTouchEvent方法中返回值的设置，如果返回false，那么意味着当前的View不会是该次的责任人，将不会对其持有；如果返回true，此时View会持有该事件并不再向外传递。

## 总结

### 1.优点

可以对请求者和处理者的关系解耦，提高代码的灵活性。

### 2.缺点

每次都需要对链中请求处理者遍历，如果处理者太多那么遍历必定会影响性能，特别是在一些递归调用者中，要慎用。

# 解释器模式

解释器模式是一种用的比较少的行为型模式，其提供了一种解释语言的语法或表达式的方式。但是它的使用场景确实很广泛，只是因为我们自己很少回去构造一个语言的文法，所以使用较少。

## 定义

给定一个语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。（其中语言就是我们需要解释的对象，文法就是这个语言的规律，解释器就是翻译机，通过文法来翻译语言。）

## 使用场景

1.如果某个简单的语言需要解释执行而且可以将该语言中的语句表示为一个抽象的语法树时可以考虑使用解释器模式。

2.在某些特定的领域出现不断重复的问题时，可以将该领域的问题转化为一种语法规则下的语句，然后构建解释器来解释该语句。

## 简单实现

我们使用解释器模式对“m+n+p”这个表达式进行解释，那么代表数字的m、n和p就可以看成终结符号，而“+”这个运算符号可以当做非终结符号。

TerminalExpression：终结符表达式，实现文法中与终结符有关的解释操作。文法中每个终结符都有一个具体的终结表达式与之对应。

NonterminalExpression ：非终结符表达式，实现文法中与非终结符有关的解释操作。非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式。

抽象的算数运算解释器

public abstract class ArithemticExpression {

/\*\*

\* 抽象的解析方法

\* 具体的解析逻辑由具体的子类实现

\*

\* @return 解析得到具体的值

\*/

public abstract int interpreter();

}

数字解释器

public class NumExpression extends ArithemticExpression{

private int num;

public NumExpression(int num){

this.num = num;

}

@Override

public int interpreter() {

return num;

}

}

运算符号解释器

public abstract class OperatorExpression extends ArithemticExpression{

protected ArithemticExpression exp1, exp2;

public OperatorExpression(ArithemticExpression exp1, ArithemticExpression exp2){

this.exp1 = exp1;

this.exp2 = exp2;

}

}

具体的加法运算符解释器

public class AdditionExpression extends OperatorExpression{

public AdditionExpression(ArithemticExpression exp1,

ArithemticExpression exp2) {

super(exp1, exp2);

}

@Override

public int interpreter() {

return exp1.interpreter() + exp2.interpreter();

}

}

处理解释器

public class Calculator {

//声明一个Stack栈储存并操作所有相关的解释器

private Stack<ArithemticExpression> mExpStack = new Stack<ArithemticExpression>();

public Calculator(String expression){

//声明两个ArithemticExpression类型的临时变量，储存运算符左右两边的数字解释器

ArithemticExpression exp1,exp2;

//根据空格分割表达式字符串(比如1 + 2 + 3 + 4)

String[] elements = expression.split(" ");

/\*

\* 遍历表达式元素数组

\*/

for(int i = 0; i < elements.length; i++){

/\*

\* 判断运算符号

\*/

switch (elements[i].charAt(0)) {

case '+':

//如果是加号，则将栈中的解释器弹出作为运算符号左边的解释器

exp1 = mExpStack.pop();

//同时将运算符号数组下标的下一个元素构造为一个数字解释器

exp2 = new NumExpression(Integer.parseInt(elements[++i]));

//通过上面的两个数字解释器构造加法运算解释器

mExpStack.push(new AdditionExpression(exp1, exp2));

break;

default:

/\*

\* 如果为数字，直接构造数字解释器并压入栈

\*/

mExpStack.push(new NumExpression(Integer.valueOf(elements[i])));

break;

}

}

}

/\*\*

\* 计算结果

\*

\* @return 最终的计算结果

\*/

public int calculate(){

return mExpStack.pop().interpreter();

}

}

调用

public class Client {

public static void main(String[] args) {

Calculator c = new Calculator("22 + 553 + 83 + 5");

System.out.println("计算结果："+c.calculate());

}

}

结果：

计算结果：663

1

如果相加如减法的操作，在Calculator中加入相应判断即可：

public class SubtractionExpression extends OperatorExpression{

public SubtractionExpression(ArithemticExpression exp1,

ArithemticExpression exp2) {

super(exp1, exp2);

}

@Override

public int interpreter() {

return exp1.interpreter() - exp2.interpreter();

}

}

Calculator中加入：

case '-':

exp1 = mExpStack.pop();

exp2 = new NumExpression(Integer.parseInt(elements[++i]));

mExpStack.push(new SubtractionExpression(exp1, exp2));

break;

从上面可以看出解释器模式很灵活，他将复杂问题可以简单化、模块化、分离实现、解释执行。

## Android源码中的模式实现

PackageParser是对AndroidManifest.xml配置文件进行读取的，具体原理参考

[解析AndroidManifest原理](http://blog.csdn.net/zhbinary/article/details/7353739)

## 总结

### 1.优点

最大的优点使其灵活的扩展性，当我们想对文法规则进行扩展延伸时，只需要增加相应的非终结符解释器，并在构建抽象语法树时，使用到新增的解释器对象进行具体的解释即可，非常方便。

### 2.缺点

1.每个语法都要产生一个非终结符表达式，语法规则比较复杂时，就可能产生大量的类文件，为维护带来了非常多的麻烦。   
2.解释器模式由于使用了大量的循环和递归，效率是个问题，特别是用于解析复杂、冗长的语法时，效率是难以忍受的。

# 命令模式

命令模式是行为型模式之一。总体来说并不难理解，只是比较繁琐，他会将简单的调用关系解耦成多个部分，增加类的复杂度，但是即便如此，命令模式的结构依然清晰。

## 定义

将一个请求封装成一个对象，从而让用户使用不同的请求把客户端参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

## 使用场景

（1）需要抽出待执行的动作，然后以参数的形式提供出来。

（2）在不同的时刻指定、排列和执行请求。一个命令对象可以有与初始请求无关的生存期。

（3）需要支持操作取消。

（4）支持修改日志功能，这样当系统崩溃时，这些修改可以被重做一遍。

（5）需要支持事务操作。

## 简单实现

以推箱子游戏为例，一般游戏中会有五个按钮，分别是左移、右移、下移、上移和撤销。那么玩游戏的人就是客户端，五个按钮就是调用者，执行具体按钮命令的方法是命令角色。

接收者角色：推箱子游戏

public class PushBox {

/\*\*

\* 执行向左命令

\*/

public void toLeft(){

System.out.println("向左");

}

/\*\*

\* 执行向右命令

\*/

public void toRight(){

System.out.println("向右");

}

/\*\*

\* 执行向下命令

\*/

public void toDown(){

System.out.println("向下");

}

/\*\*

\* 执行向上命令

\*/

public void toUp(){

System.out.println("向上");

}

/\*\*

\* 执行撤销命令

\*/

public void revoke(){

System.out.println("撤销");

}

}

命令者抽象

public interface Command {

/\*\*

\* 命令执行方法

\*/

void execute();

/\*\*

\* 获取命令类型

\*/

void getCommand();

}

具体命令者，左移命令类

public class LeftCommand implements Command{

//持有一个接受推箱子游戏对象的引用

private PushBox pushBox;

public LeftCommand(PushBox pushBox){

this.pushBox = pushBox;

}

@Override

public void execute() {

//调用具体命令

pushBox.toLeft();

}

@Override

public void getCommand() {

System.out.print("向左-->");

}

}

具体命令者，右移命令类

public class RightCommand implements Command{

//持有一个接受推箱子游戏对象的引用

private PushBox pushBox;

public RightCommand(PushBox pushBox){

this.pushBox = pushBox;

}

@Override

public void execute() {

//调用具体命令

pushBox.toRight();

}

@Override

public void getCommand() {

System.out.print("向右-->");

}

}

具体命令者，上移命令类

public class UpCommand implements Command{

//持有一个接受推箱子游戏对象的引用

private PushBox pushBox;

public UpCommand(PushBox pushBox){

this.pushBox = pushBox;

}

@Override

public void execute() {

//调用具体命令

pushBox.toUp();

}

@Override

public void getCommand() {

System.out.print("向上-->");

}

}

具体命令者，下移命令类

public class DownCommand implements Command{

//持有一个接受推箱子游戏对象的引用

private PushBox pushBox;

public DownCommand(PushBox pushBox){

this.pushBox = pushBox;

}

@Override

public void execute() {

//调用具体命令

pushBox.toDown();

}

@Override

public void getCommand() {

System.out.print("向下-->");

}

}

1

具体命令者，撤销命令类

public class RevokeCommand implements Command{

//持有一个接受推箱子游戏对象的引用

private PushBox pushBox;

public RevokeCommand(PushBox pushBox){

this.pushBox = pushBox;

}

@Override

public void execute() {

//调用具体命令

pushBox.revoke();;

}

@Override

public void getCommand() {

}

}

请求者类，命令由按钮发起

public class Buttons {

private LeftCommand leftCommand; //向左移动的命令对象引用

private RightCommand rightCommand; //向右移动的命令对象引用

private UpCommand upCommand; //向上移动的命令对象引用

private DownCommand downCommand; //向下移动的命令对象引用

private RevokeCommand revokeCommand; //撤销命令对象引用

private ArrayList<Command> commandList = new ArrayList<Command>();//用于记录命令动作

/\*\*

\* 获取执行命令

\*/

public void getCommandList(){

for(Command c : commandList){

c.getCommand();

}

System.out.println("");

}

/\*\*

\* 设置向左移动的命令对象

\*

\* @param leftCommand 向左移动的命令对象

\*/

public void setLeftCommand(LeftCommand leftCommand){

this.leftCommand = leftCommand;

}

/\*\*

\* 设置向右移动的命令对象

\*

\* @param rightCommand 向右移动的命令对象

\*/

public void setRightCommand(RightCommand rightCommand){

this.rightCommand = rightCommand;

}

/\*\*

\* 设置向上移动的命令对象

\*

\* @param upCommand 向上移动的命令对象

\*/

public void setUpCommand(UpCommand upCommand){

this.upCommand = upCommand;

}

/\*\*

\* 设置向下移动的命令对象

\*

\* @param downCommand 向下移动的命令对象

\*/

public void setDownCommand(DownCommand downCommand){

this.downCommand = downCommand;

}

/\*\*

\* 设置撤销命令对象

\*

\* @param revokeCommand 撤销命令对象

\*/

public void setRevokeCommand(RevokeCommand revokeCommand){

this.revokeCommand = revokeCommand;

}

/\*\*

\* 按下向左按钮

\*/

public void toLeft(){

leftCommand.execute();

commandList.add(leftCommand);

}

/\*\*

\* 按下向右按钮

\*/

public void toRight(){

rightCommand.execute();

commandList.add(rightCommand);

}

/\*\*

\* 按下向上按钮

\*/

public void toUp(){

upCommand.execute();

commandList.add(upCommand);

}

/\*\*

\* 按下向下按钮

\*/

public void toDown(){

downCommand.execute();

commandList.add(downCommand);

}

/\*\*

\* 按下撤销按钮

\*/

public void toRevoke(){

revokeCommand.execute();

commandList.remove(commandList.size()-1);

}

}

客户端调用

public class Client {

public static void main(String[] args) {

//首先创建游戏

PushBox pushBox = new PushBox();

//根据游戏构造5种命令

LeftCommand leftCommand = new LeftCommand(pushBox);

RightCommand rightCommand = new RightCommand(pushBox);

UpCommand upCommand = new UpCommand(pushBox);

DownCommand downCommand = new DownCommand(pushBox);

RevokeCommand revokeCommand = new RevokeCommand(pushBox);

//按钮可以执行不同命令

Buttons buttons = new Buttons();

buttons.setLeftCommand(leftCommand);

buttons.setRightCommand(rightCommand);

buttons.setUpCommand(upCommand);

buttons.setDownCommand(downCommand);

buttons.setRevokeCommand(revokeCommand);

//执行操作

buttons.toLeft();

buttons.toDown();

buttons.toDown();

buttons.toRight();

buttons.getCommandList();

buttons.toRevoke();

buttons.toUp();

buttons.toLeft();

buttons.toDown();

buttons.toUp();

buttons.getCommandList();

}

}

执行结果

向左

向下

向下

向右

向左-->向下-->向下-->向右-->

撤销

向上

向左

向下

向上

向左-->向下-->向下-->向上-->向左-->向下-->向上-->

在这么长的代码之后是不是觉得很烦琐，明明可以很简单的实现，如下：

public class Client {

public static void main(String[] args) {

//首先创建游戏

PushBox pushBox = new PushBox();

pushBox.toDown();

pushBox.toRight();

pushBox.toUp();

}

}

其实设计模式有一个重要的原则：对修改关闭对扩展开放。如果使用如上的简单方式，那么以后的修改只能去修改PushBox类，然后修改Client类，这显然违反了这一原则。如果使用命令模式，那么Client类无需修改，只需要修改PushBox类的内部操作，Client类无需知道具体的内部实现。

设计模式的使用之前也有提到，主要是要看当前场景的复杂度和以后的需求进行扩展、维护等方面，完全使用设计模式也是不提倡的，这就需要设计者权衡利弊了。

## Android源码中的命令模式实现

### PackageHandler

PackageManagerService中，其对包的相关消息处理右其内部类PackageHandler承担，其将需要处理的请求作为对象通过消息传递给相关的方法，而对于包的安装、移动以及包大小的测量则分别封装为HandlerParams的具体子类InstallParams、MoveParams和MeasureParams。源码如下：

private abstract class HandlerParams {

private static final int MAX\_RETRIES = 4;

/\*\*

\* Number of times startCopy() has been attempted and had a non-fatal

\* error.

\*/

private int mRetries = 0;

final boolean startCopy() {

boolean res;

try {

if (DEBUG\_INSTALL) Slog.i(TAG, "startCopy");

if (++mRetries > MAX\_RETRIES) {

Slog.w(TAG, "Failed to invoke remote methods on default container service. Giving up");

mHandler.sendEmptyMessage(MCS\_GIVE\_UP);

handleServiceError();

return false;

} else {

handleStartCopy();

res = true;

}

} catch (RemoteException e) {

if (DEBUG\_INSTALL) Slog.i(TAG, "Posting install MCS\_RECONNECT");

mHandler.sendEmptyMessage(MCS\_RECONNECT);

res = false;

}

handleReturnCode();

return res;

}

final void serviceError() {

if (DEBUG\_INSTALL) Slog.i(TAG, "serviceError");

handleServiceError();

handleReturnCode();

}

abstract void handleStartCopy() throws RemoteException;

abstract void handleServiceError();

abstract void handleReturnCode();

}

可以看出HandlerParams也是一个抽象命令者

## 5.总结

1.优点

命令模式的封装性很好，更弱的耦合性，更灵活的控制性以及更好的扩展性。

2.缺点

类的膨胀，大量衍生类的创建。

# 观察者模式

观察者模式是一个使用率非常高的模式，它最常用在GUI系统、订阅–发布系统。因为这个模式的一个重要作用就是解耦，将被观察者和观察者解耦，使得它们之间的依赖性更小，甚至做到毫无依赖。比如安卓的开源项目EventBus、Otto、AndroidEventBus等事件总线类的和RxJava响应式编程其核心都是使用观察者模式。

## 定义

观察者模式是一种行为类模式，它定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

## 使用场景

1）关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。

（2）事件多级触发场景。

（3）跨系统的消息交换场景，如消息队列、事件总线的处理机制。

## 简单实现

这里举一个追剧的例子，平常为了不错过最新的电视剧我们会订阅或关注这个电视剧，当电视剧更新后会第一时间推送给我们，下来就简单实现一下。

抽象观察者类

/\*\*

\* 抽象观察者类，为所有具体观察者定义一个接口，在得到通知时更新自己

\*/

public interface Observer {

/\*\*

\* 有更新

\*

\* @param message 消息

\*/

public void update(String message);

}

1

2

3

4

5

6

7

8

9

10

11

12

抽象被观察者类

/\*\*

\* 抽象被观察者类

\*/

public interface Observable {

/\*\*

\* 推送消息

\*

\* @param message 内容

\*/

void push(String message);

/\*\*

\* 订阅

\*

\* @param observer 订阅者

\*/

void register(Observer observer);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

具体的观察者类

/\*\*

\* 具体的观察者类，也就是订阅者

\*/

public class User implements Observer {

@Override

public void update(String message) {

System.out.println(name + "," + message + "更新了！");

}

// 订阅者的名字

private String name;

public User(String name) {

this.name = name;

}

}

具体的被观察者类

/\*\*

\* 具体的被观察者类，也就是订阅的节目

\*/

public class Teleplay implements Observable{

private List<Observer> list = new ArrayList<Observer>();//储存订阅者

@Override

public void push(String message) {

for(Observer observer:list){

observer.update(message);

}

}

@Override

public void register(Observer observer) {

list.add(observer);

}

}

实现

public class Client {

public static void main(String[] args) {

//被观察者，这里就是用户订阅的电视剧

Teleplay teleplay = new Teleplay();

//观察者，这里就是订阅用户

User user1 = new User("小明");

User user2 = new User("小光");

User user3 = new User("小兰");

//订阅

teleplay.register(user1);

teleplay.register(user2);

teleplay.register(user3);

//推送新消息

teleplay.push("xxx电视剧");

}

}

结果

小明,xxx电视剧更新了！

小光,xxx电视剧更新了！

小兰,xxx电视剧更新了！

由上面的代码可以看出实现了一对多的消息推送，推送消息都是依赖Observer和Observable这些抽象类，而User和Teleplay完全没有耦合，保证了订阅系统的灵活性和可扩展性。

## Android源码中的观察者模式

### BaseAdapter

BaseAdapter我相信大家都不陌生，在ListView的适配器中我们都是继承它。下面来简单分析分析。

BaseAdapter 部分代码：

|  |
| --- |
| public abstract class BaseAdapter implements ListAdapter, SpinnerAdapter {  //数据集观察者  private final DataSetObservable mDataSetObservable = new DataSetObservable();  public boolean hasStableIds() {  return false;  }  public void registerDataSetObserver(DataSetObserver observer) {  mDataSetObservable.registerObserver(observer);  }  public void unregisterDataSetObserver(DataSetObserver observer) {  mDataSetObservable.unregisterObserver(observer);  }  /\*\*  \* 当数据集变化时，通知所有观察者  \*/  public void notifyDataSetChanged() {  mDataSetObservable.notifyChanged();  } |

看看mDataSetObservable.notifyChanged()方法：

|  |
| --- |
| public class DataSetObservable extends Observable<DataSetObserver> {  /\*\*  \* Invokes {@link DataSetObserver#onChanged} on each observer.  \* Called when the contents of the data set have changed. The recipient  \* will obtain the new contents the next time it queries the data set.  \*/  public void notifyChanged() {  synchronized(mObservers) {  // since onChanged() is implemented by the app, it could do anything, including  // removing itself from {@link mObservers} - and that could cause problems if  // an iterator is used on the ArrayList {@link mObservers}.  // to avoid such problems, just march thru the list in the reverse order.  for (int i = mObservers.size() - 1; i >= 0; i--) {  mObservers.get(i).onChanged();  }  }  } |

可以看出在mDataSetObservable.notifyChanged()中遍历所有观察者，并调用他们的onChanged()，从而告知观察者发生了什么。

那么观察者怎么来的，那就是setAdapter方法，代码如下：

|  |
| --- |
| @Override  public void setAdapter(ListAdapter adapter) {  if (mAdapter != null && mDataSetObserver != null) {  mAdapter.unregisterDataSetObserver(mDataSetObserver);  }  resetList();  mRecycler.clear();  if (mHeaderViewInfos.size() > 0|| mFooterViewInfos.size() > 0) {  mAdapter = new HeaderViewListAdapter(mHeaderViewInfos, mFooterViewInfos, adapter);  } else {  mAdapter = adapter;  }  mOldSelectedPosition = INVALID\_POSITION;  mOldSelectedRowId = INVALID\_ROW\_ID;  // AbsListView#setAdapter will update choice mode states.  super.setAdapter(adapter);  if (mAdapter != null) {  mAreAllItemsSelectable = mAdapter.areAllItemsEnabled();  mOldItemCount = mItemCount;  mItemCount = mAdapter.getCount();  checkFocus();  mDataSetObserver = new AdapterDataSetObserver();  mAdapter.registerDataSetObserver(mDataSetObserver);//注册观察者  ......省略  } |

AdapterDataSetObserver定义在ListView的父类AbsListView中，是一个数据集观察者，代码：

|  |
| --- |
| class AdapterDataSetObserver extends AdapterView<ListAdapter>.AdapterDataSetObserver {  @Override  public void onChanged() {  super.onChanged();  if (mFastScroller != null) {  mFastScroller.onSectionsChanged();  }  }  @Override  public void onInvalidated() {  super.onInvalidated();  if (mFastScroller != null) {  mFastScroller.onSectionsChanged();  }  }  } |

它由继承自AbsListView的父类AdapterView的AdapterDataSetObserver, 代码如下 :

|  |
| --- |
| class AdapterDataSetObserver extends DataSetObserver {  private Parcelable mInstanceState = null;  // 上文有说道，调用Adapter的notifyDataSetChanged的时候会调用所有观察者的onChanged方法,核心实现就在这里  @Override  public void onChanged() {  mDataChanged = true;  mOldItemCount = mItemCount;  // 获取Adapter中数据的数量  mItemCount = getAdapter().getCount();  // Detect the case where a cursor that was previously invalidated has  // been repopulated with new data.  if (AdapterView.this.getAdapter().hasStableIds() && mInstanceState != null  && mOldItemCount == 0 && mItemCount > 0) {  AdapterView.this.onRestoreInstanceState(mInstanceState);  mInstanceState = null;  } else {  rememberSyncState();  }  checkFocus();  // 重新布局ListView、GridView等AdapterView组件  requestLayout();  }  // 代码省略  public void clearSavedState() {  mInstanceState = null;  }  } |

当ListView的数据发生变化时，调用Adapter的notifyDataSetChanged函数，这个函数又会调用DataSetObservable的notifyChanged函数，这个函数会调用所有观察者 (AdapterDataSetObserver) 的onChanged方法。这就是一个观察者模式！

## 总结

### 1.优点

（1）观察者和被观察者之间是抽象耦合，应对业务变化。

（2）增强系统的灵活性和可扩展性。

### 2.缺点

在应用观察者模式时需要考虑一下开发效率和运行效率的问题，程序中包括一个被观察者、多个观察者，开发、调试等内容会比较复杂，而且在Java中消息的通知一般是顺序执行，那么一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般会采用异步实现。

# 备忘录模式

备忘录模式是一种行为模式，该模式用于保存对象当前的状态，并且在之后可以再次恢复到此状态，有点像是我们平常说的”后悔药”。

## 定义

在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样，以后就可将该对象恢复到原先保存的状态

## 使用场景

（1）需要保存一个对象在某一个时刻的状态或部分状态。

（2）如果用一个接口来让其他对象得到这些状态，将会暴露对象的实现细节并破坏对象的封装性，一个对象不希望外界直接访问其内部状态，通过中间对象可以间接访问其内部状态。

## 简单实现

书中例子：以”使命召唤”游戏为例，用游戏中的存档功能来举例。

首先是备忘录类

/\*\*

\* 备忘录类

\*/

public class Memento {

public int mCheckpoint;//武器

public int mLiftValue;//生命

public String mWeapon;//关卡

@Override

public String toString() {

return "Memento [mCheckpoint=" + mCheckpoint + ",mLiftValue="

+ mLiftValue + ",mWeapon=" + mWeapon + "]";

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

游戏类，在该类可以通过createMemento函数来创建该用户的备忘录对象，外部可以通过restore函数将CallOfDuty 对象的状态从备忘录对象中恢复。

/\*\*

\*

\* 简单模拟“使命召唤”游戏

\*

\*/

public class CallOfDuty {

private int mCheckpoint = 1;

private int mLiftValue = 100;

private String mWeapon = "沙漠之鹰";

//玩游戏

public void play(){

System.out.println("打游戏："+String.format("第%d关", mCheckpoint) + "奋战杀敌中");

mLiftValue -= 10;

System.out.println("进度升级了");

mCheckpoint++;

System.out.println("到达" + String.format("第%d关", mCheckpoint));

}

//退出游戏

public void quit(){

System.out.println("--------------");

System.out.println("退出前的游戏属性：" + this.toString());

System.out.println("退出游戏");

System.out.println("--------------");

}

/\*\*

\*创建备忘录

\*/

public Memento createMemento(){

Memento memento = new Memento();

memento.mCheckpoint = mCheckpoint;

memento.mLiftValue = mLiftValue;

memento.mWeapon = mWeapon;

return memento;

}

//恢复游戏

public void restore(Memento memento){

this.mCheckpoint = memento.mCheckpoint;

this.mLiftValue = memento.mLiftValue;

this.mWeapon = memento.mWeapon;

System.out.println("恢复后的游戏属性：" + this.toString());

}

//省略getter和setter方法

@Override

public String toString() {

return "CallOfDuty [mCheckpoint=" + mCheckpoint + ",mLiftValue="

+ mLiftValue + ",mWeapon=" + mWeapon + "]";

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

备忘录操作类：

/\*\*

\* Caretaker，负责管理Memento

\*/

public class Caretaker {

Memento mMemento; //备忘录

/\*\*

\* 存档

\*/

public void archive(Memento memento){

this.mMemento = memento;

}

/\*\*

\* 获取存档

\*/

public Memento getMemento(){

return mMemento;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

客户端使用代码：

public class Client {

public static void main(String[] args) {

//构建游戏对象

CallOfDuty game = new CallOfDuty();

//1.打游戏

game.play();

Caretaker caretaker = new Caretaker();

//2.游戏存档

caretaker.archive(game.createMemento());

//3.退出游戏

game.quit();

//4.恢复游戏

CallOfDuty newGame = new CallOfDuty();

newGame.restore(caretaker.getMemento());

//5.再次打游戏(不存档)

game.play();

//6.恢复之前存档

newGame.restore(caretaker.getMemento());

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

结果：

打游戏：第1关奋战杀敌中

进度升级了

到达第2关

--------------

退出前的游戏属性：CallOfDuty [mCheckpoint=2,mLiftValue=90,mWeapon=沙漠之鹰]

退出游戏

--------------

恢复后的游戏属性：CallOfDuty [mCheckpoint=2,mLiftValue=90,mWeapon=沙漠之鹰]

打游戏：第2关奋战杀敌中

进度升级了

到达第3关

恢复后的游戏属性：CallOfDuty [mCheckpoint=2,mLiftValue=90,mWeapon=沙漠之鹰]

1

2

3

4

5

6

7

8

9

10

11

12

13

上面的代码中，各个角色职责清晰、单一，代码也比较简单，即对外屏蔽了对CallOfDuty角色的直接访问，在满足了对象状态存取功能的同时也使得该模块的结构保持清晰、整洁。

## Android源码中的备忘录模式

### 1.onSaveInstanceState和onRestoreInstanceState

当Activity不是正常方式退出，且Activity在随后的时间内被系统杀死之前会调用这两个方法让开发人员可以有机会存储Activity相关信息，且在下次返回Activity时恢复这些数据。通过这两个函数。开发人员能够在某些特殊场景下储存与界面相关的信息，提升用户体验。

基本使用参考：[链接](http://blog.csdn.net/shulianghan/article/details/38297083)

## 5.总结

1.优点

（1）给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史状态。

（2）实现了信息的封装，使用户不需要关心状态的保存细节。

2.缺点

消耗资源，如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。\

# 认真的雪：迭代器模式

迭代器模式，又叫做游标模式，是行为型设计模式之一。我们知道对容器对象的访问必然会涉及遍历算法，我们可以将遍历的方法封装在容器中，或者不提供遍历方法，让使用容器的人自己去实现去吧。这两种情况好像都能够解决问题。

然而在前一种情况，容器承受了过多的功能，它不仅要负责自己“容器”内的元素维护（添加、删除等等），而且还要提供遍历自身的接口；而且由于遍历状态保存的问题，不能对同一个容器对象同时进行多个遍历。第二种方式倒是省事，却又将容器的内部细节暴露无遗。

正因于此，迭代器模式应运而生，在客户访问类与容器体之间插入一个第三者–迭代器，很好的解决了上述弊端。

## 定义

提供一种方法顺序访问一个容器对象中的各个元素，而又不需要暴露该对象的内部表示。

## 使用场景

遍历一个容器对象时。

## 简单实现

用书中的例子：小民和小辉分别在公司两个事业部，某天老板安排任务让他们俩统计一下各自部门的员工数据。

员工实体类：

public class Employee {

private String name;// 姓名

private int age;// 年龄

private String sex;// 性别

private String position;// 职位

public Employee(String name, int age, String sex, String position) {

super();

this.name = name;

this.age = age;

this.sex = sex;

this.position = position;

}

// 简化代码，省略setter和getter方法

@Override

public String toString() {

return "Employee{" + "name='" + name + '\'' + ", age=" + age + ", sex="

+ sex + ", position='" + position + '\'' + "}";

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

小辉部门：

public class CompanyHui {

private Employee[] array = new Employee[3];

public CompanyHui(){

array[0] = new Employee("辉哥", 28, "男", "程序猿");

array[1] = new Employee("小红", 23, "男", "程序猿");

array[2] = new Employee("小辉", 25, "男", "程序猿");

}

public Employee[] getEmployees(){

return array;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

小民部门：

public class CompanyMin {

private List<Employee> list = new ArrayList<>();

public CompanyMin(){

list.add(new Employee("小民", 26, "男", "程序猿"));

list.add(new Employee("小芸", 22, "女", "测试"));

list.add(new Employee("小方", 18, "女", "测试"));

list.add(new Employee("可儿", 21, "女", "设计"));

list.add(new Employee("朗情", 19, "女", "设计")); //吐槽一下，为什么就小民一个男的，小辉部门全男的。

}

public List<Employee> getEmployees(){

return list;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

Boss查看：

public class Boss {

public static void main(String[] args) {

CompanyHui hui = new CompanyHui();

Employee[] huiList = hui.getEmployees();

for(int i = 0; i < huiList.length; i++){

System.out.println(huiList[i]);

}

CompanyMin min = new CompanyMin();

List minList = min.getEmployees();

for(int i = 0; i < minList.size(); i++){

System.out.println(minList.get(i).toString());

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

结果：

Employee{name='辉哥', age=28, sex=男, position='程序猿'}

Employee{name='小红', age=23, sex=男, position='程序猿'}

Employee{name='小辉', age=25, sex=男, position='程序猿'}

Employee{name='小民', age=26, sex=男, position='程序猿'}

Employee{name='小芸', age=22, sex=女, position='测试'}

Employee{name='小方', age=18, sex=女, position='测试'}

Employee{name='可儿', age=21, sex=女, position='设计'}

Employee{name='朗情', age=19, sex=女, position='设计'}

1

2

3

4

5

6

7

8

这样看似也没有问题，但是如果有多个部门，每个部门有各自的实现，那么我们就要在Boss类中增加一遍遍历逻辑，这样Boss类的功能会越来越多，同时暴露了内部细节。那么我们需要定义一个迭代器接口：

public interface Iterator {

/\*\*

\* 是否还有下一个元素

\*

\* @return true表示有，false表示没有

\*/

boolean hasNext();

/\*\*

\* 返回当前元素，并将位置移至下一位

\*/

Object next();

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

小民的迭代器：

public class MinIterator implements Iterator{

private List<Employee> list;

private int position;

public MinIterator(List<Employee> list){

this.list = list;

}

@Override

public boolean hasNext() {

return !(position > list.size() - 1 || list.get(position) == null);

}

@Override

public Object next() {

Employee e = list.get(position);

position++;

return e;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

小辉的迭代器：

public class HuiIterator implements Iterator{

private Employee[] array;

private int position;

public HuiIterator(Employee[] array){

this.array = array;

}

@Override

public boolean hasNext() {

return !(position > array.length - 1 || array[position] == null);

}

@Override

public Object next() {

Employee e = array[position];

position++;

return e;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

定义容器类的接口

public interface Company {

/\*\*

\* 返回一个迭代器对象

\*

\* @return 迭代器对象

\*/

Iterator iterator();

}

1

2

3

4

5

6

7

8

9

10

修改一下之前的两个容器类：

public class CompanyHui implements Company{

private Employee[] array = new Employee[3];

public CompanyHui(){

array[0] = new Employee("辉哥", 28, "男", "程序猿");

array[1] = new Employee("小红", 23, "男", "程序猿");

array[2] = new Employee("小辉", 25, "男", "程序猿");

}

public Employee[] getEmployees(){

return array;

}

@Override

public Iterator iterator() {

return new HuiIterator(array);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

public class CompanyMin implements Company{

private List<Employee> list = new ArrayList<>();

public CompanyMin(){

list.add(new Employee("小民", 26, "男", "程序猿"));

list.add(new Employee("小芸", 22, "女", "测试"));

list.add(new Employee("小方", 18, "女", "测试"));

list.add(new Employee("可儿", 21, "女", "设计"));

list.add(new Employee("朗情", 19, "女", "设计"));

}

public List<Employee> getEmployees(){

return list;

}

@Override

public Iterator iterator() {

return new MinIterator(list);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

Boss查看：

public class Boss {

public static void main(String[] args) {

CompanyHui hui = new CompanyHui();

check(hui.iterator());

CompanyMin min = new CompanyMin();

check(min.iterator());

}

private static void check(Iterator iterator){

while (iterator.hasNext()) {

System.out.println(iterator.next().toString());

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

结果不变，就不重复写了。

## Android源码中的模式实现

### 1.Cursor

当我们使用SQLiteDatabase的query方法查询数据库时，会返回一个Cursor游标对象，该游标的实质就是一个具体的迭代器，我们可以使用它来遍历数据库查询所得的结果集。

## 5.总结

迭代器模式发展至今，几乎所有的高级语言都有相应的内置实现，对于开发者而言，已经极少会自己去实现迭代器了，所以本章内容更多的是了解而非应用。

1.优点

（1）符合面向对象设计原则中的单一职责原则。

（2）支持对容器对象的多种遍历。弱化了容器类与遍历算法之间的关系。

2.缺点

（1）类文件的增加。

（3）会出现ConcurrentModificationException异常。

（2）遍历过程是一个单向且不可逆的遍历。

# 模板方法模式

模板方法模式是结构最简单的行为型设计模式，也是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。在其结构中只存在父类与子类之间的继承关系。

## 定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

## 模板方法模式中的方法

### 1.模板方法

一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。这个模板方法定义在抽象类中，并由子类不加以修改地完全继承下来。所以模板方法大多会定义为**final类型**，指明主要的逻辑功能在子类中不能被重写。模板方法是一个具体方法，它给出了一个顶层逻辑框架，而逻辑的组成步骤在抽象类中可以是具体方法，也可以是抽象方法。由于模板方法是具体方法，因此模板方法模式中的抽象层只能是抽象类，而不是接口。

### 2.基本方法

（1）抽象方法：一个抽象方法由**抽象类声明，由具体子类实现**。在Java语言里抽象方法以abstract关键字标示。

（2）钩子方法：一个钩子方法由抽象类**声明并实现**，而子类会加以扩展。子类可以通过扩展钩子方法来影响模板方法的逻辑。

## 使用场景

（1）多个子类有公有的方法，并且逻辑基本相同。

（2）重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能由各个子类实现。

（3）重构时，模板方法是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子方法约束其行为。

## 简单实现

以电脑开机为例，假设现在有两台电脑，一台Windows系统电脑，一台Mac系统电脑。但是开机流程基本一致：步骤为开启电源、系统检查、加载系统、检查是否需要登录。

抽象的 Computer

/\*\*

\* 抽象的 Computer

\*/

public abstract class AbstractComputer {

//下面是抽象方法，子类可以覆盖,不允许外部直接调用这些方法，所以用protected

/\*\*

\* 开启电源

\*/

protected abstract void powerOn();

/\*\*

\* 检查硬件

\*/

protected abstract void checkHardware();

/\*\*

\* 载入操作系统

\*/

protected abstract void loadOS();

/\*\*

\* 登录

\*/

protected abstract void login();

//下面是钩子方法，声明并实现

/\*\*

\* 是否需要登录

\*

\* @return true为需要登录

\*/

protected boolean isLogin(){

return true;

}

//下面是模板方法，定义为final，子类不能覆盖此方法

/\*\*

\* 启动计算机方法，步骤为开启电源、系统检查、加载系统、检查是否登录。

\*/

public final void startUp(){

System.out.println("--------开机 START--------");

powerOn();

checkHardware();

loadOS();

if(isLogin()){

login();

}

System.out.println("-------- 开机 END --------");

}

}

Windows系统电脑（不需登录）：

/\*\*

\* Windows系统电脑

\*/

public class WindowsComputer extends AbstractComputer{

@Override

protected void powerOn() {

System.out.println("Windows电脑开启电源");

}

@Override

protected void checkHardware() {

System.out.println("Windows电脑检查硬件");

}

@Override

protected void loadOS() {

System.out.println("Windows电脑载入操作系统");

}

@Override

protected void login() {

}

@Override

protected boolean isLogin() {

return false;//返回false，不需登录

}

}

Mac系统电脑（需登录）：

/\*\*

\* Mac系统电脑

\*/

public class MacComputer extends AbstractComputer{

@Override

protected void powerOn() {

System.out.println("Mac电脑开启电源");

}

@Override

protected void checkHardware() {

System.out.println("Mac电脑检查硬件");

}

@Override

protected void loadOS() {

System.out.println("Mac电脑载入操作系统");

}

@Override

protected void login() {

System.out.println("Mac电脑登录");

}

}

调用：

public class Client {

public static void main(String[] args) {

AbstractComputer comp = new WindowsComputer();

comp.startUp();

comp = new MacComputer();

comp.startUp();

}

}

结果：

--------开机 START--------

Windows电脑开启电源

Windows电脑检查硬件

Windows电脑载入操作系统

-------- 开机 END --------

--------开机 START--------

Mac电脑开启电源

Mac电脑检查硬件

Mac电脑载入操作系统

Mac电脑登录

-------- 开机 END --------

## Android源码中的模板方法模式

### 1.AsyncTask

在使用AsyncTask时，我们都知道把耗时操作放到doInBackground(Params… params)中，在doInBackground之前，如果想做一些初始化操作，可以把实现写在onPreExecute中，当doInBackground执行完后会执行onPostExecute方法，而我们只需要构建AsyncTask对象，然后执行execute方法。

### 2.Activity的生命周期

ActivityThread的main函数被调用后，依次执行Activity的onCreate、onStart、onResume函数，用户通常在Activity的子类中覆写onCreate方法，并且在该方法中调用setContentView来设置布局。

## 6.区别

（1）工厂方法是模板方法的一种特殊版本。

（2）策略模式和模板方法模式都是封装算法，一个用组合，一个用继承。

（3）策略模式和模板模式通常可以互相替换。它们都像试卷，策略模式是选择题，模板模式是填空题

## 7.总结

模板方法模式用4个字概括就是：流程封装。也就是把某个固定的流程封装到一个final方法中，并且让子类能够定制这个流程中的某些或者所有步骤，这就要求父类提取公用的代码，提升代码的复用率，同时带来了更好的可扩展性。

1.优点

（1）封装不变部分，扩展可变部分。

（2）提取公共部分代码，便于维护。

2.缺点

需要为每一个基本方法的不同实现提供一个子类，如果父类中可变的基本方法太多，将会导致类的个数增加，系统更加庞大，设计也更加抽象，此时，可结合桥接模式来进行设计。

# 中介者模式

中介者模式也称为调解者模式或调停者模式，是一种行为型模式。

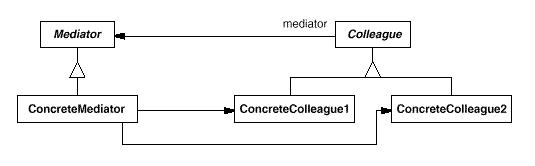
## 定义

中介者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使它们可以松散耦合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。

## 使用场景

当对象之间的交互操作很多且每个对象的行为操作都依赖彼此时，为防止在修改一个对象的行为时，同时涉及很多其他对象的行为，可使用中介者模式。

## UML类图



（1）Mediator：抽象中介者角色，定义了同事对象到中介者对象的接口，一般以抽象类的方式实现。

（2）ConcreteMediator：具体中介者角色，继承于抽象中介者，实现了父类定义的方法，它从具体的同事对象接受消息，向具体同事对象发出命令。

（3）Colleague：抽象同事类角色，定义了中介者对象的接口，它只知道中介者而不知道其他的同事对象。

（4）ConcreteColleague1、ConcreteColleague2：具体同事类角色，继承于抽象同事类，每个具体同事类都知道本身在小范围的行为，而不知道在大范围内的目的。

## 简单实现

在电脑中，主机部分主要分为：CPU、内存、显卡、IO设备，而将它们整合起来的就是主板，这里主板就是一个中介者。以此为例。

在电脑中，主机部分主要分为：CPU、内存、显卡、IO设备，而将它们整合起来的就是主板，这里主板就是一个中介者。以此为例。

抽象中介者：

public abstract class Mediator {

/\*\*

\* 同事对象改变时通知中介者的方法

\* 在同事对象改变时由中介者去通知其他的同事对象

\*

\* @param c 同事对象

\*/

public abstract void changed(Colleague c);

}

1

2

3

4

5

6

7

8

9

10

11

抽象同事：

public abstract class Colleague {

protected Mediator mediator;//每一个同事都该知道其中介者

public Colleague(Mediator mediator) {

this.mediator = mediator;

}

}

1

2

3

4

5

6

7

8

CPU同事：

public class CPU extends Colleague{

private String dataVideo, dataSound; //视频和音频数据

public CPU(Mediator mediator) {

super(mediator);

}

/\*\*

\* 获取视频数据

\*

\* @return 视频数据

\*/

public String getDataVideo(){

return dataVideo;

}

/\*\*

\* 获取音频数据

\*

\* @return 音频数据

\*/

public String getDataSound(){

return dataSound;

}

/\*\*

\* 解码数据

\*

\* @param data音、视频数据

\*/

public void decodeData(String data){

//分割音、视频数据

String[] tmp = data.split(",");

//解析音、视频数据

dataVideo = tmp[0];

dataSound = tmp[1];

//告诉中介者自身状态改变

mediator.changed(this);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

光驱同事：

public class CDDevice extends Colleague{

private String data; //视频数据

public CDDevice(Mediator mediator) {

super(mediator);

}

/\*\*

\* 读取视频数据

\*

\* @return 视频数据

\*/

public String read(){

return data;

}

/\*\*

\* 加载视频数据

\*

\* @return 音频数据

\*/

public void load(){

data = "视频数据,音频数据";

//告诉中介者自身状态改变

mediator.changed(this);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

显卡同事：

public class GraphicsCard extends Colleague{

public GraphicsCard(Mediator mediator) {

super(mediator);

}

/\*\*

\* 播放视频数据

\*

\* @param 视频数据

\*/

public void videoPlay(String data){

System.out.println("视频：" + data);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

声卡同事：

public class SoundCard extends Colleague{

public SoundCard(Mediator mediator) {

super(mediator);

}

/\*\*

\* 播放音频数据

\*

\* @param 音频数据

\*/

public void soundPlay(String data){

System.out.println("音频：" + data);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

主板中介者：

public class MainBoard extends Mediator{

private CDDevice cdDevice; //光驱设备

private CPU cpu; //CPU

private SoundCard soundCard; //声卡设备

private GraphicsCard graphicsCard; //显卡设备

@Override

public void changed(Colleague c) {

//如果光驱读取了数据

if(c == cdDevice){

handleCD((CDDevice) c);

}

//如果CPU处理完数据

else if(c == cpu){

handleCD((CPU) c);

}

}

/\*\*

\* 处理光驱读取数据后与其他设备的交互

\*

\* @param cdDevice 光驱设备

\*/

public void handleCD(CDDevice cdDevice){

cpu.decodeData(cdDevice.read());

}

/\*\*

\* 处理CPU读取数据后与其他设备的交互

\*

\* @param cpu CPU

\*/

public void handleCD(CPU cpu){

soundCard.soundPlay(cpu.getDataSound());

graphicsCard.videoPlay(cpu.getDataVideo());

}

/\*\*

\* 设置CD设备

\*

\* @param CDDevice CD设备

\*/

public void setCDDevice(CDDevice cdDevice){

this.cdDevice = cdDevice;

}

/\*\*

\* 设置CPU

\*

\* @param cpu CPU

\*/

public void setCPU(CPU cpu){

this.cpu = cpu;

}

/\*\*

\* 设置声卡设备

\*

\* @param soundCard 声卡设备

\*/

public void setSoundCard(SoundCard soundCard){

this.soundCard = soundCard;

}

/\*\*

\* 设置显卡设备

\*

\* @param graphicsCard 显卡设备

\*/

public void setGraphicsCard(GraphicsCard graphicsCard){

this.graphicsCard = graphicsCard;

}

}

播放电影：

public class Client {

public static void main(String[] args) {

//构造主板对象

MainBoard mediator = new MainBoard();

//分别构造各个零件

CDDevice cd = new CDDevice(mediator);

CPU cpu = new CPU(mediator);

GraphicsCard gc = new GraphicsCard(mediator);

SoundCard sc = new SoundCard(mediator);

//将各个零件安装到主板

mediator.setCDDevice(cd);

mediator.setCPU(cpu);

mediator.setGraphicsCard(gc);

mediator.setSoundCard(sc);

//播放电影

cd.load();

}

}

结果：

音频：音频数据

视频：视频数据

可以看出中介者模式将多对多的相互作用转化为一对多的相互作用，将系统从网状结构变为以中介者为中心的星形结构（这里就是主板），达到降低系统的复杂性，提高可扩展性。

## Android源码中的中介者模式

### Keyguard解锁屏

详细机制参考：Android4.0 Keyguard解锁屏机制

## 6.总结

其实在Android开发中我们可能无意间就使用了中介者模式，比如登录注册界面，我们使用EditText的addTextChangedListener监听输入密码的位数、用户名是否为空，密码与确认密码是否一致等等判断时，此时多个控件交互，就是由Activity充当中介者来协调。

1.优点

（1）适当地使用中介者模式可以避免同事类之间的过度耦合，使得各同事类之间可以相对独立地使用。

（2）使用中介者模式可以将对象的行为和协作进行抽象，能够比较灵活的处理对象间的相互作用。

（3）使用中介者模式可以将对象间多对多的关联转变为一对多的关联，使对象间的关系易于理解和维护。

2.缺点

中介者模式是一种比较常用的模式，也是一种比较容易被滥用的模式。对于大多数的情况，同事类之间的关系不会复杂到混乱不堪的网状结构，因此，大多数情况下，将对象间的依赖关系封装的同事类内部就可以的，没有必要非引入中介者模式。滥用中介者模式，只会让事情变的更复杂。所以，我们决定使用中介者模式之前要多方考虑、权衡利弊。

# 代理模式

代理模式也称委托模式，是结构型设计模式之一。是应用广泛的模式之一。

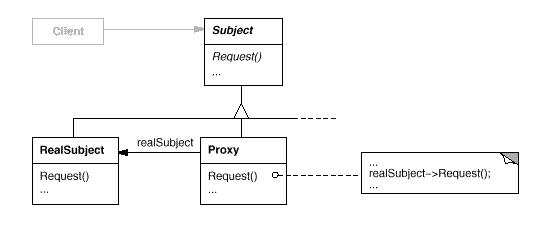
## 定义

为其他对象提供一种代理以控制对这个对象的访问。

## 使用场景

当无法或不想直接访问某个对象或访问某个对象存在困难时可以通过一个代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口。

## .UML类图



（1）Subject：抽象主题类，声明真实主题与共同接口方法，该类可以是抽象类或接口。

（2）RealSubject：真实主题类（被委托类），尤其执行具体的业务逻辑方法。

（3）Proxy：代理类（委托类），该类持有一个对真实主题类的引用，在其所实现的接口方法中调用真实主题类中相应的接口方法执行，以此起到代理作用。

## 简单实现

书中例子：以小民诉讼的流程举例。那么需要代理律师代理，诉讼简单流程：提交申请–>进行举证–>开始辩护–>诉讼完成。

诉讼接口类：

public interface ILawsuit {

/\*\*

\* 提交申请

\*/

void submit();

/\*\*

\* 进行举证

\*/

void burden();

/\*\*

\* 开始辩护

\*/

void defend();

/\*\*

\* 诉讼完成

\*/

void finish();

}

具体诉讼人小民：

public class XiaoMin implements ILawsuit{

@Override

public void submit() {

//小民申请仲裁

System.out.println("老板年底拖欠工资，特此申请仲裁！");

}

@Override

public void burden() {

//小民提交证据

System.out.println("这是合同书和过去一年的银行工资流水！");

}

@Override

public void defend() {

//铁证如山

System.out.println("证据确凿，不需要再说什么！");

}

@Override

public void finish() {

//结果

System.out.println("诉讼成功，判决老板即日起七天内结算工资！");

}

}

代理律师：

public class Lawyer implements ILawsuit{

private ILawsuit mLawsuit; //持有一个具体被代理者的引用

public Lawyer(ILawsuit lawsuit) {

this.mLawsuit = lawsuit;

}

@Override

public void submit() {

mLawsuit.submit();

}

@Override

public void burden() {

mLawsuit.burden();

}

@Override

public void defend() {

mLawsuit.defend();

}

@Override

public void finish() {

mLawsuit.finish();

}

}

开始仲裁：

public class Client {

public static void main(String[] args) {

//构造出诉讼人小民

ILawsuit xiaomin = new XiaoMin();

//构造一个代理律师，并将小民传递进去

ILawsuit lawyer = new Lawyer(xiaomin);

//律师提交申请

lawyer.submit();

//律师进行举证

lawyer.burden();

//律师代小民辩护

lawyer.defend();

//完成诉讼

lawyer.finish();

}

}

结果：

老板年底拖欠工资，特此申请仲裁！

这是合同书和过去一年的银行工资流水！

证据确凿，不需要再说什么！

诉讼成功，判决老板即日起七天内结算工资！

同样我们也可以代理其他人，只需要实现ILawsuit即可。上面的代理模式也叫静态代理，也就是在代码运行前代理类的class文件就已经存在。那么相反，当然也会有动态代理，下面用动态代理实现上述例子：

Java提供了一个便捷的动态代理接口InvocationHandler，我们来实现它：

public class DynamicPorxy implements InvocationHandler{

private Object obj; //被代理类的引用

public DynamicPorxy(Object obj) {

this.obj = obj;

}

@Override

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

// 调用被代理类对象的方法

Object result = method.invoke(obj, args);

return result;

}

}

修改后的Client类：

public class Client {

public static void main(String[] args) {

//构造出诉讼人小民

ILawsuit xiaomin = new XiaoMin();

//1.静态代理

//构造一个代理律师，并将小民传递进去

//ILawsuit lawyer = new Lawyer(xiaomin);

//--------------------------------------

//2.动态代理

//构造一个动态代理

DynamicPorxy proxy = new DynamicPorxy(xiaomin);

//获取被代理类小民的ClassLoader

ClassLoader loader = xiaomin.getClass().getClassLoader();

//动态构造一个代理者律师

ILawsuit lawyer = (ILawsuit) Proxy.newProxyInstance(loader, new Class[]{ ILawsuit.class }, proxy);

//律师提交申请

lawyer.submit();

//律师进行举证

lawyer.burden();

//律师代小民辩护

lawyer.defend();

//完成诉讼

lawyer.finish();

}

}

这里我们通过invoke方法来调用具体的被代理方法

结果不变，由此可以看出动态代理通过一个代理类来处理N多个被代理类，其实质是对代理者与被代理者解耦。相对而言静态代理则只能为给定接口下的实现类做代理，如果接口不同那么就需要重新定义不同的代理类，较为复杂，但是静态代理更符合面向对象原则。具体使用哪种方式，根据个人喜好。

## .Android源码中的代理模式实现

### 1.ActivityManagerProxy代理类

ActivityManager是Android中管理和维护Activity的相关信息的类，为了隔离它与ActivityManagerService，有效降低二者的耦合，在这中间使用了ActivityManagerProxy代理类，所有对ActivityManagerService的访问都转换成对代理类的访问，这样ActivityManager就与ActivityManagerService解耦了。

## 6.总结

### 1.优点

（1）对代理者与被代理者进行解耦。

（2）代理对象在客户端和目标对象之间起到一个中介的作用，这样可以起到对目标对象的保护。

### 2.缺点

基本没有缺点，真要说缺点就是设计模式的通病：对类的增加。

# 组合模式

组合模式也称为部分-整体模式，结构型设计模式之一。

## 定义

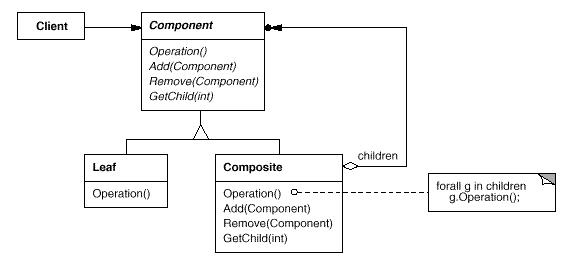
将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

## 使用场景

（1）表示对象的部分-整体层次结构时。

（2）从一个整体中能够独立出部分模块或功能的场景。

## UML类图



（1）Component：抽象根节点，为组合中的对象声明接口。在适当的情况下，实现所有类共有接口的缺省行为。声明一个接口用于访问和管理Component的子节点。可在递归结构中定义一个接口，用于访问一个父节点，并在合适的情况下实现它。

（2）Composite：定义有子节点的那些枝干节点行为，存储子节点，在Component接口中实现与子节点有关的操作。

（3）Leaf：在组合中表示叶子节点对象，叶子节点没有子节点，在组合中定义节点对象的行为。

（4）Client：通过Component接口操纵组合节点的对象。

如图这种将组合所使用的方法全部定义在抽象类的方式称为透明的组合模式，如果将Component中的Add、Remove、GetChild去除，只在Composite中单独添加，这种方式称为安全的组合模式。然而后者违背了依赖倒置原则。

## 简单实现

以文件和文件夹这样的文件系统为例

文件和文件夹的抽象类：（Component）

以文件和文件夹这样的文件系统为例

文件和文件夹的抽象类：（Component）

public abstract class Dir {

/\*\*

\* 声明一个List成员变量来储存文件夹下的所有元素

\*/

protected List<Dir> dirs = new ArrayList<Dir>();

private String name; //当前文件或文件夹名

public Dir(String name) {

this.name = name;

}

/\*\*

\* 添加一个文件或文件夹

\*

\* @param dir 文件或文件夹

\*/

public abstract void addDir(Dir dir);

/\*\*

\* 移除一个文件或文件夹

\*

\* @param dir 文件或文件夹

\*/

public abstract void rmDir(Dir dir);

/\*\*

\* 清空文件夹下所有元素

\*/

public abstract void clear();

/\*\*

\* 输出文件夹目录结构

\*/

public abstract void print();

/\*\*

\* 获取文件夹下所有的文件或文件夹

\*

\* @return 文件夹下所有的文件或文件夹

\*/

public abstract List<Dir> getFiles();

/\*\*

\* 获取文件或文件夹名

\*

\* @return 文件或文件夹名

\*/

public String getName(){

return name;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

表示文件夹的类：（Composite）

public class Folder extends Dir{

public Folder(String name) {

super(name);

}

@Override

public void addDir(Dir dir) {

dirs.add(dir);

}

@Override

public void rmDir(Dir dir) {

dirs.remove(dir);

}

@Override

public void clear() {

dirs.clear();

}

@Override

public void print() {

System.out.print(getName() + "(");

Iterator<Dir> iter = dirs.iterator();

while (iter.hasNext()) {

Dir dir = iter.next();

dir.print();

if(iter.hasNext()){

System.out.print(", ");

}

}

System.out.print(")");

}

@Override

public List<Dir> getFiles() {

return dirs;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

表示文件夹的类：（Leaf）

public class File extends Dir{

public File(String name) {

super(name);

}

@Override

public void addDir(Dir dir) {

throw new UnsupportedOperationException("文件对象不支持该操作！");

}

@Override

public void rmDir(Dir dir) {

throw new UnsupportedOperationException("文件对象不支持该操作！");

}

@Override

public void clear() {

throw new UnsupportedOperationException("文件对象不支持该操作！");

}

@Override

public void print() {

System.out.print(getName());

}

@Override

public List<Dir> getFiles() {

throw new UnsupportedOperationException("文件对象不支持该操作！");

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

客户类：

public class Client {

public static void main(String[] args) {

//构造一个目录对象表示C盘根目录

Dir diskC = new Folder("C");

//C盘根目录下有一个文件Log.txt

diskC.addDir(new File("Log.txt"));

//C盘根目录下有三个目录Windows、PerfLogs、Program File

Dir dirWin = new Folder("Windows");

//Windows目录下有文件explorer.exe

dirWin.addDir(new File("explorer.exe"));

diskC.addDir(dirWin);

//PerfLogs目录

Dir dirPer = new Folder("PerfLogs");

//PerfLogs目录下有文件null.txt

dirPer.addDir(new File("null.txt"));

diskC.addDir(dirPer);

//Program File目录

Dir dirPro = new Folder("Program File");

//Program File目录下有文件ftp.txt

dirPro.addDir(new File("ftp.txt"));

diskC.addDir(dirPro);

//打印出文件结构

diskC.print();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

结果：

C(Log.txt, Windows(explorer.exe), PerfLogs(null.txt), Program File([ftp.txt)](ftp://ftp.txt)))

## .Android源码中的模式实现

### 1.View和ViewGroup的嵌套组合

View和ViewGroup的结构很像上面的UML类图，不过View的视图层级使用的是安全的组合模式。ViewGroup有对View的addView、removeView、getChildAt等方法，想必大家也很熟悉。

6.总结

1.优点

（1）组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，他让高层模块忽略了层次的差异，方便对整个层次结构进行控制。

（2）简化了高层模块的代码。

（3）在组合模式中增加新的枝干构件和叶子构件都很方便，无须对现有类库进行修改，符合“开闭原则”。

（4）对树形结构的控制变得简单。

2.缺点

组合模式不容易限制组合中的构件。因为大多数情况下，它们都来自于相同的抽象层，此时，必须进行类型检查来实现，这个实现过程较为复杂。

# 适配器模式

适配器模式是结构型设计模式之一，它在我们的开发中使用率极高，比如ListView、GridView以及RecyclerView都需要使用Adapter。

## 定义

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配无法在一起工作的两个类可以在一起工作。

## 使用场景

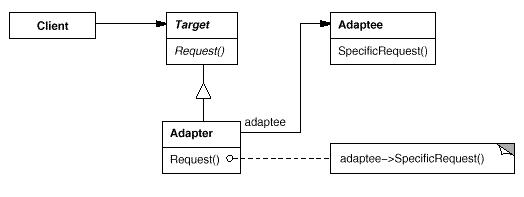
（1）系统需要使用现有的类，但此类的接口不符合系统的需要，即接口不兼容。

（2）想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

（3）需要一个统一的输出接口，而输入端的类型不可预知。

## UML类图

适配器模式分为两种，即类适配器模式与对象适配器模式。以下是类适配器的UML图。



（1）Target：目标角色，也就是所期待得到的接口。注意：由于这里讨论的是类适配器模式，因此目标不可以是类。

（2）Adaptee：现在需要适配的接口。

（3）Adapter：适配器角色，也就是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

## 简单实现

以笔记本电源适配器为例，电源适配器将220V的电压转换到5V。那么5V电压就是Target接口，220V电压就是Adaptee类，转换就是Adapter。

（1）以类适配器模式实现

//Target角色

public interface FiveVolt {

public int getVolt5();

}

1

2

3

4

5

//Adapter角色

public class Volt220 {

public int getVolt220(){

return 220;

}

}

1

2

3

4

5

6

7

public class VoltAdapter extends Volt220 implements FiveVolt{

@Override

public int getVolt5() {

return 5;

}

}

1

2

3

4

5

6

7

8

9

public class Test {

public static void main(String[] args) {

VoltAdapter adapter = new VoltAdapter();

System.out.println("输出电压：" + adapter.getVolt5());

}

}

1

2

3

4

5

6

7

8

9

结果：

输出电压：5

1

（2）对象适配器模式实现

对象适配器模式与类适配器模式不同的是，对象适配器模式不是使用继承关系连接到Adapter类，而是使用代理关系连接到Adapter类。

所以FiveVolt 、Volt220 不变，VoltAdapter 修改如下：

public class VoltAdapter1 implements FiveVolt{

Volt220 mVolt220;

public VoltAdapter1(Volt220 adaptee) {

this.mVolt220 = adaptee;

}

public int getVolt220(){

return mVolt220.getVolt220();

}

@Override

public int getVolt5() {

return 5;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

Test修改为：

public class Test {

public static void main(String[] args) {

VoltAdapter adapter = new VoltAdapter(new Volt220());

System.out.println("输出电压：" + adapter.getVolt5());

}

}

这种实现方式直接将要适配的对象传递到Adapter中，使用组合的形式实现接口兼容的效果。这比类适配器方式更为灵活，同时被适配对象的方法不会暴露出来。因此对象适配器模式更加灵活、实用。

## Android源码中的适配器模式

### 1.ListView的Adapter

这里ListView的Adapter就是使用的对象适配器模式，Target就是View，Adapter角色就是将Item View输出为View抽象的角色，Adaptee就是需要被处理的Item View。

## 总结

### 1.优点

（1）更好的复用性：系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

（2）更好的扩展性：在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

### 2.缺点

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

# 装饰模式

装饰模式也称为包装模式，是结构型设计模式之一。装饰模式是一种用于替代继承技术的一种方案。

## 定义

动态的给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

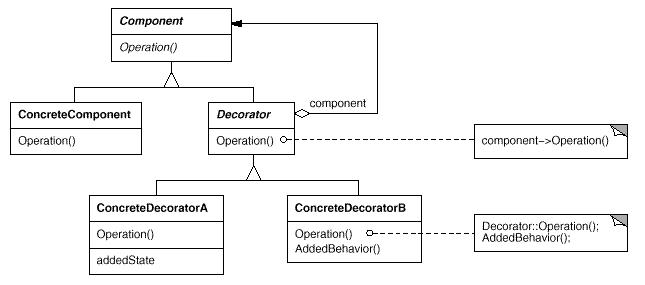
## 使用场景

（1）需要透明且动态地扩展类的功能时。且在不影响其他对象的情况下。

（2）当不能采用继承对系统进行扩展时可以使用装饰模式。比如final类。

## UML类图

## UML类图



（1）Component：抽象组件。可以是一个接口或抽象类，其充当的就是被装饰的原始对象。

（2）ConcreteComponent：组件具体实现类，该类是Component类的基本实现，也是我们装饰的具体对象。

（3）Decorator：抽象装饰者，其职责就是装饰我们的组件对象，通过其子类扩展该方法以达到装饰的目的。其内部一定要有一个指向组件对象的引用。在大多数情况下，该类为抽象类，需要根据不同的装饰逻辑实现不同的具体子类。

（4）ConcreteDecoratorA、ConcreteDecoratorB：装饰着具体实现类。负责向构件添加新的职责。

## 简单实现

以一个男孩穿衣装扮为例。实现给男孩在家与出门的穿衣装扮。

抽象组件类（Component）：

public abstract class Person {

/\*\*

\* Person下有一个穿着的抽象方法

\*/

public abstract void dressed();

}

1

2

3

4

5

6

具体实现类（ConcreteComponent）：表示要装扮的Boy

public class Boy extends Person{

@Override

public void dressed() {

System.out.println("Boy穿了内衣内裤");

}

}

1

2

3

4

5

6

7

8

抽象装饰类（Decorator）：PersonCloth 表示人所穿着的衣服

public class PersonCloth extends Person{

protected Person mPerson; //保持一个Person类的引用

public PersonCloth(Person mPerson) {

super();

this.mPerson = mPerson;

}

@Override

public void dressed() {

mPerson.dressed();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

出门穿的衣服：

public class OutsideCloth extends PersonCloth{

public OutsideCloth(Person mPerson) {

super(mPerson);

}

/\*\*

\* 穿短袖

\*/

private void dressShirt(){

System.out.println("穿件短袖");

}

/\*\*

\* 穿牛仔裤

\*/

private void dressJean(){

System.out.println("穿牛仔裤");

}

/\*\*

\* 穿鞋子

\*/

private void dressShoes(){

System.out.println("穿鞋子 ");

}

@Override

public void dressed() {

super.dressed();

dressShirt();

dressJean();

dressShoes();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

在家穿的衣服：

public class HomeCloth extends PersonCloth{

public HomeCloth(Person mPerson) {

super(mPerson);

}

/\*\*

\* 穿短裤

\*/

private void dressShorts(){

System.out.println("穿短裤");//在家里随便点

}

@Override

public void dressed() {

super.dressed();

dressShorts();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

装扮：

public class Client {

public static void main(String[] args) {

//首先有一个男孩

Person person = new Boy();

//在家

PersonCloth personCloth = new HomeCloth(person);

personCloth.dressed();

System.out.println("--------------");

//出门

PersonCloth personCloth1 = new OutsideCloth(person);

personCloth1.dressed();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

结果

Boy穿了内衣内裤

穿短裤

--------------

Boy穿了内衣内裤

穿件短袖

穿牛仔裤

穿鞋子

## Android源码中的实现

Context类在Android中被称为“上帝对象”，它的本质就是一个抽象类，在装饰模式中相当于抽象组件，而在内部定义了大量的抽象方法，比如我们经常用到的startActivity方法。而真正实现是在ContextImpl中完成，那么ContextImpl 就是具体实现类。因为ContextWrapper 继承于Context ，所以ContextWrapper 就是装饰者。具体大家可以自行查看源码。

## 6.区别

### 1.与[代理模式](http://blog.csdn.net/qq_17766199/article/details/50492805)的区别

（1）装饰模式是以对客户端透明的方式扩展对象的功能，是继承方案的一个替代；而代理模式则是给一个对象提供一个代理对象，并有代理对象来控制对原有对象的引用。

（2）装饰模式应该为所装饰的对象增强功能；代理模式是对代理对象施加控制，不对对象本身功能进行增强。

### 2.与[适配器模式](http://blog.csdn.net/qq_17766199/article/details/50514877)的区别

适配器模式是用新接口来调用原接口，原接口对新系统是不可见的；装饰模式增强了其他对象的功能而同时又不改变它的接口。

## 总结

在实际开发中我们应该写过如下代码：其实这些新增方法的调用就类似装饰模式中的装饰者的职责，只不过这里我们没有保持对组件类的引用。

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

//初始化数据

initData();

//初始化控件

initViews();

//初始化事件

initEvent();

}

1.优点

（1）对于扩展一个对象的功能，装饰模式比继承更加灵活性，不会导致类的个数急剧增加。

（2）可以通过一种动态的方式在运行时选择不同的具体装饰类，从而实现不同的行为。

（3）可以对一个对象进行多次装饰，通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，得到功能更为强大的对象。

（4）具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，原有类库代码无须改变，符合“开闭原则”。

2.缺点

（1）使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，大量小对象的产生势必会占用更多的系统资源，在一定程序上影响程序的性能。

（2）对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为繁琐。

# 享元模式

享元模式是结构型设计模式之一，是对对象池的一种实现。就像它的名字一样，共享对象，避免重复的创建。我们常用的String 就是使用了共享模式，所以String类型的对象创建后就不可改变，如果当两个String对象所包含的内容相同时，JVM只创建一个String对象对应这两个不同的对象引用。

## 定义

采用一个共享来避免大量拥有相同内容对象的开销。使用享元模式可有效支持大量的细粒度对象。

## 使用场景

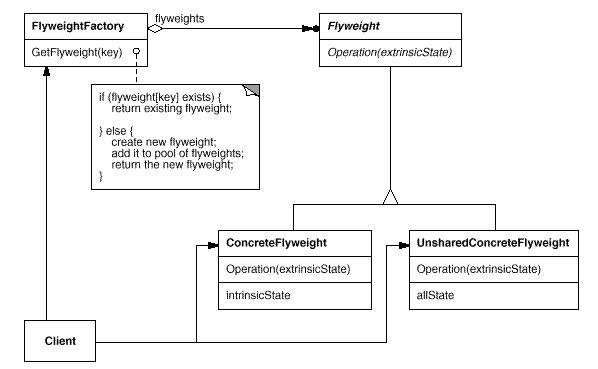
（1）系统中存在大量的相似对象。

（2）细粒度的对象都具备较接近的外部状态，而且内部状态与环境不关，也就是说对象没有特定身份。

（3）需要缓冲池的场景。

PS：内部状态与外部状态：在享元对象内部并且不会随着环境改变而改变的共享部分，可以称之为享元对象的内部状态，反之随着环境改变而改变的，不可共享的状态称之为外部状态。

## UML类图



享元模式分为单纯享元模式和复合享元模式，上图是复合享元模式。

（1）Flyweight：享元对象抽象基类或者接口。

（2）ConcreateFlyweight：具体的享元对象，如果有内部状态的话，必须负责为内部状态提供存储空间。

（3）UnsharadConcreateFlyweight：复合享元角色所代表的对象是不可以共享的，并且可以分解成为多个单纯享元对象的组合。单纯享元模式没有此项，这也是两者在结构上的区别。

（4）FlyweightFactoiy：享元工厂，负责管理享元对象池和创建享元对象。

（5）Client：维护对所有享元对象的引用，而且还需要存储对应的外蕴状态。

## 简单实现

情景：过年买火车票的时候，我们需要查询车票的情况，那么如果每次查询车票时都创建一个结果，那么必然会大量的创建出许多重复的对象，频繁的去销毁他们，使得GC任务繁重。那么这时我们可以使用享元模式，将这些对象缓存起来，查询时优先使用缓存，没有缓存在重新创建。

首先是Ticket接口（Flyweight）：

public interface Ticket {

public void showTicketInfo(String bunk);

}

1

2

3

4

5

6

TrainTicket具体实现类（ConcreateFlyweight）：

//火车票

public class TrainTicket implements Ticket{

public String from; // 始发地

public String to; // 目的地

public String bunk; //铺位

public int price; //价格

public TrainTicket(String from, String to) {

this.from = from;

this.to = to;

}

@Override

public void showTicketInfo(String bunk) {

price = new Random().nextInt(300);

System.out.println("购买 从 " + from + " 到 " + to + "的" + bunk + "火车票" + ", 价格：" + price);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

TicketFactory 管理查询火车票（FlyweightFactoiy）:

public class TicketFactory {

static Map<String, Ticket> sTicketMap = new ConcurrentHashMap<String, Ticket>();

public static Ticket getTicket(String from ,String to){

String key = from + "-" + to;

if(sTicketMap.containsKey(key)){

System.out.println("使用缓存 ==> " + key);

return sTicketMap.get(key);

}else{

System.out.println("创建对象 ==> " + key);

Ticket ticket = new TrainTicket(from, to);

sTicketMap.put(key, ticket);

return ticket;

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

查询：

final class Client {

public static void main(String[] args) {

Ticket ticket01 = TicketFactory.getTicket("北京", "青岛");

ticket01.showTicketInfo("上铺");

Ticket ticket02 = TicketFactory.getTicket("北京", "青岛");

ticket02.showTicketInfo("下铺");

Ticket ticket03 = TicketFactory.getTicket("北京", "西安");

ticket03.showTicketInfo("坐票");

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

结果

创建对象 ==> 北京-青岛

购买 从 北京 到 青岛的上铺火车票, 价格：71

使用缓存 ==> 北京-青岛

购买 从 北京 到 青岛的下铺火车票, 价格：32

创建对象 ==> 北京-西安

购买 从 北京 到 西安的坐票火车票, 价格：246

## .Android源码中的实现

### 1.Message

因为Android是事件驱动的，因此如果通过new创建 Message 就会创建大量的 Message 对象，导致内存占用率高，频繁GC等问题。那么 Message 就采用了享元模式。

Message通过next成员变量保有对下一个Message的引用，最后一个可用Message的next则为空。从而构成了一个**Message链表**。Message Pool就通过该链表的表头管理着所有闲置的Message，一个Message在使用完后可以通过recycle()方法进入Message Pool，并在需要时通过obtain静态方法从Message Pool获取。Message 承担了享元模式中3个元素的职责，即是Flyweight抽象，又是ConcreateFlyweight角色，同时又承担了FlyweightFactoiy管理对象池的职责。

所以使用Message推荐obtain()，不要去new了。

//1。使用new Message()

//Message mess = new Message();

//2。使用Message.obtain()

Message mess = Message.obtain();

mess.what = 1;

//Message mess = mHandler.obtainMessage(1); 与上两行的代码一样，可以参考源码查看

mHandler.sendMessage(mess);

## 总结

### 1.优点

（1）大大减少应用程序创建的对象，降低程序内存的占用，增强程序的性能。

（2）使用享元模式，可以让享元对象可以在不同的环境中被共享。

### 2.缺点

（1）使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。

（2）享元模式将需、享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

# 外观模式

外观模式是结构型设计模式之一，它在开发中的运用频率非常高，是我们封装API的常用手段。我们经常使用的三方SDK基本都使用的外观模式，这样可以对用户屏蔽很多实现细节，降低用户使用成本。

## 定义

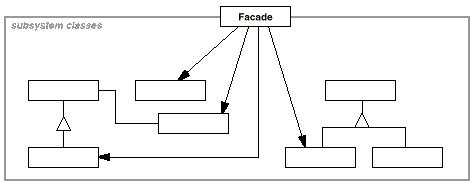
要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。外观模式提供一个高层次的接口，使得子系统更易于使用。

## 使用场景

1）为复杂子系统提供一个简单接口，对外隐藏子系统的具体实现、隔离变化。

（2）当你需要构建一个层次结构的子系统时，使用外观模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过外观接口进行通信，从而简化了它们之间的依赖关系。

## UML类图



（1）Facade：系统对外的统一接口，系统内部系统地工作。

（2）其他分支：子系统接口。

可以看出外观模式结构很简单，但是如果没有封装，那么用户就要操作几个子系统的交互逻辑，容易出现错误。

## 简单实例

手机集合了电话功能、短信功能、拍照和GPS等功能。那么以手机为例，简单的用外观模式实现一下。

Phone接口与PhoneImpl：

public interface Phone {

//拨电话

public void dail();

//挂断

public void hangup();

}

1

2

3

4

5

6

7

8

public class PhoneImpl implements Phone{

@Override

public void dail() {

System.out.println("打电话");

}

@Override

public void hangup() {

System.out.println("挂断");

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

Camera接口与Camera的实现类：

public interface Camera {

//打开相机

public void open();

//拍照

public void takePicture();

//关闭相机

public void close();

}

1

2

3

4

5

6

7

8

9

10

11

12

public class SamsungCamera implements Camera{

@Override

public void open() {

System.out.println("打开相机");

}

@Override

public void takePicture() {

System.out.println("拍照");

}

@Override

public void close() {

System.out.println("关闭相机");

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

外观类MobilePhone：

public class MobilePhone {

private Phone mPhone = new PhoneImpl();

private Camera mCamera = new SamsungCamera();

public void dail(){

mPhone.dail();

}

public void hangup() {

mPhone.hangup();

}

public void takePicture() {

mCamera.open();

mCamera.takePicture();

}

public void closeCamera() {

mCamera.close();

}

public void videoChat(){

System.out.println("--> 视频聊天接通中");

mCamera.open();

mPhone.dail();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

调用：

public class Client {

public static void main(String[] args) {

MobilePhone mobilePhone = new MobilePhone();

//拍照

mobilePhone.takePicture();

//视频聊天

mobilePhone.videoChat();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

结果：

打开相机

拍照

--> 视频聊天接通中

打开相机

打电话

## Android源码中的外观模式

### 1.Context

Context 是一个抽象类，它的真正实现是ContextImpl 类，通过查看ContextImpl 源码我们可以看到ContextImpl内部封装了很多不同子系统的操作。例如：Activity的跳转、发送广播、启动服务和设置壁纸等，这些工作不是在ContextImpl 中实现，而是交给了具体的子系统进行处理。通过Context 这个抽象类定义了一组接口，ContextImpl实现。这样用户通常情况下就不需要对每个子系统进行了解。这样对用户屏蔽了具体的实现细节，降低了使用成本。

## 6.总结

1.优点

（1）对客户程序隐藏子系统的细节，因而减少了客户对于子系统的耦合，能够拥抱变化。

（2）外观类对子系统的接口封装，使得系统更易于使用。

2.缺点

（1）外观类接口膨胀。由于子系统的接口都有外观类统一对外暴露，使得外观类的API接口较多，在一定程度上增加了用户使用成本。

（2）外观类没有遵循开闭原则，当业务出现变更时，可能需要直接修改外观类。

# 桥接模式

桥接模式也称为桥梁模式，是结构型设计模式之一。桥接模式中体现了“单一职责原则”、“开闭原则”、“里氏替换原则”、“依赖倒置原则”等。同时它也是很实用的一种模式。

## 定义

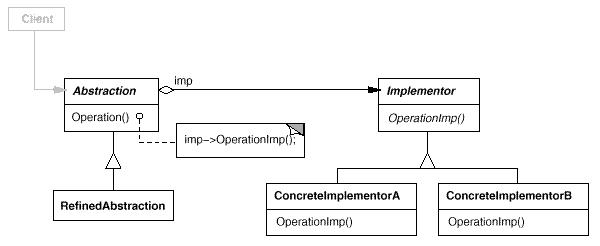
将抽象部分与现实部分分离，使它们都可以独立地进行变化。

## 使用场景

（1）如果一个系统需要在构建的抽象化角色和具体角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系。

1. （2）对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，也可以考虑使用桥接模式。
2. （3）一个类存在两个独立UML类图变化的维度，且这两个维度都需要扩展。

## UML类图



（1）Abstraction：抽象部分，该类保持一个对实现部分对象的引用，抽象部分中的方法需要调用实现部分的对象来实现，该类一般为抽象类。

（2）RefinedAbstraction：优化抽象部分，抽象部分的具体实现，该类一般是对抽象部分的方法进行完善和扩展。

（3）Implementor：实现部分。可以为接口或抽象类，其方法不一定要与抽象部分中的一致，一般情况下是由现实部分提供基本操作，而抽象部分定义的则是基于实现部分这些基本操作的业务方法。

（4）ConcreteImplementorA、ConcreteImplementorB：实现部分的具体实现。完善实现部分中的方法定义的具体逻辑。

## 简单实现

以去咖啡店喝咖啡为例，我们假定咖啡有大杯加糖、大杯不加糖、小杯加糖和小杯不加糖四种。

给咖啡添加东西的抽象类：（Implementor）

public abstract class CoffeeAdditives {

/\*\*

\* 具体要往咖啡里添加什么东西

\*

\* @param 具体添加的东西

\*/

public abstract String addSomething();

}

加糖类实现：（ConcreteImplementorA）

public class Sugar extends CoffeeAdditives{

@Override

public String addSomething() {

return "加糖";

}

}

原味类实现：（ConcreteImplementorB）

public class Ordinary extends CoffeeAdditives{

@Override

public String addSomething() {

return "原味";

}

}

咖啡类：（Abstraction）

public abstract class Coffee{

protected CoffeeAdditives impl;

public Coffee(CoffeeAdditives impl) {

this.impl = impl;

}

/\*\*

\* 咖啡具体什么样由子类决定

\*/

public abstract void makeCoffee();

}

大杯咖啡：（RefinedAbstraction）

public class LargeCoffee extends Coffee{

public LargeCoffee(CoffeeAdditives impl) {

super(impl);

}

@Override

public void makeCoffee() {

System.out.println("大杯的" + impl.addSomething() + "咖啡");

}

}

小杯咖啡：

public class SmallCoffee extends Coffee{

public SmallCoffee(CoffeeAdditives impl) {

super(impl);

}

@Override

public void makeCoffee() {

System.out.println("小杯的" + impl.addSomething() + "咖啡");

}

}

10

11

调用：

public class Client {

public static void main(String[] args) {

//原味

Ordinary implOrdinary = new Ordinary();

//加糖

Sugar implSugar = new Sugar();

//大杯咖啡 原味

LargeCoffee largeCoffeeOrdinary = new LargeCoffee(implOrdinary);

largeCoffeeOrdinary.makeCoffee();

//小杯咖啡 原味

SmallCoffee smallCoffeeOrdinary = new SmallCoffee(implOrdinary);

smallCoffeeOrdinary.makeCoffee();

//大杯咖啡 加糖

LargeCoffee largeCoffeeSugar = new LargeCoffee(implSugar);

largeCoffeeSugar.makeCoffee();

//小杯咖啡 加糖

SmallCoffee smallCoffeeSugar = new SmallCoffee(implSugar);

smallCoffeeSugar.makeCoffee();

}

}

结果：

大杯的原味咖啡

小杯的原味咖啡

大杯的加糖咖啡

## Android源码中的桥接模式

### Window与WindowManager

主要代码如下：

public abstract class Window {

//部分省略

/\*\*

\* Set the window manager for use by this Window to, for example,

\* display panels. This is <em>not</em> used for displaying the

\* Window itself -- that must be done by the client.

\*

\* @param wm The window manager for adding new windows.

\*/

public void setWindowManager(WindowManager wm, IBinder appToken, String appName) {

setWindowManager(wm, appToken, appName, false);

}

/\*\*

\* Set the window manager for use by this Window to, for example,

\* display panels. This is <em>not</em> used for displaying the

\* Window itself -- that must be done by the client.

\*

\* @param wm The window manager for adding new windows.

\*/

public void setWindowManager(WindowManager wm, IBinder appToken, String appName,

boolean hardwareAccelerated) {

mAppToken = appToken;

mAppName = appName;

mHardwareAccelerated = hardwareAccelerated

|| SystemProperties.getBoolean(PROPERTY\_HARDWARE\_UI, false);

if (wm == null) {

wm = (WindowManager)mContext.getSystemService(Context.WINDOW\_SERVICE);

}

//\*将Window与WindowManager绑定

mWindowManager = ((WindowManagerImpl)wm).createLocalWindowManager(this);

}

//部分省略

}

## 6.总结

1.优点

（1）分离抽象与现实、灵活的扩展以及对客户来说透明的实现。

（2）桥接模式可以取代多层继承，大大减少了子类的个数。

2.缺点

不容易设计，对开发者来说要有一定的经验要求。理解很容易，设计却不容易。

# REF

1. [《Android源码设计模式解析与实战》读书笔记](http://blog.csdn.net/column/details/mode.html?&page=2)