# 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

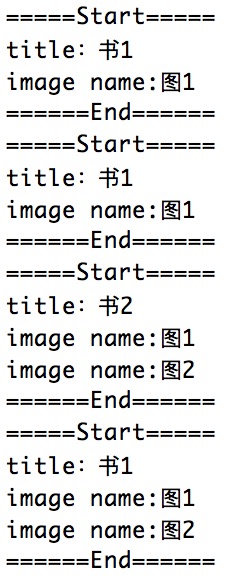
（1）类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗。   
（2）通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。   
（3）一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

## 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

# 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)