# 1. Java中的反射

## 1.1 说说你对Java中反射的理解

Java中的反射首先是能够获取到Java中要反射类的字节码，获取字节码有三种方法

* Class.forName(className)
* 类名.class
* this.getClass()

然后将字节码中的方法，变量，构造函数等映射成相应的Method、Filed、Constructor等类，这些类提供了丰富的方法可以被我们所使用。

# 2. Java中的动态代理

## 2.1 写一个ArrayList的动态代理类（笔试题）

final List<String> list = new ArrayList<String>();
List<String> proxyInstance = (List<String>) Proxy.newProxyInstance(list.getClass().getClassLoader(),
 list.getClass().getInterfaces(), new InvocationHandler() {
 @Override
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 return method.invoke(list, args);
 }
 }
);
proxyInstance.add("你好");
System.out.println(list);

## 3. Java中的设计模式

## 3.1 你所知道的设计模式有哪些

Java中一般认为有23种设计模式，我们不需要所有的都会，但是其中常用的几种设计模式应该去掌握。下面列出了所有的设计模式。需要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

* 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式
* 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式
* 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

## 3.2 单例设计模式

最好理解的一种设计模式，分为懒汉式和饿汉式。

### 3.2.1 饿汉式

public class Singleton {
 // 直接创建对象
 public static Singleton instance = new Singleton();

 // 私有化构造函数
 private Singleton() {
 }

 // 返回对象实例
 public static Singleton getInstance() {
 return instance;
 }
}

### 3.2.2 懒汉式

public class Singleton {
 // 声明变量
 private static volatile Singleton singleton2 = null;

 // 私有构造函数
 private Singleton2() {
 }

 // 提供对外方法
 public static Singleton2 getInstance() {
 if (singleton2 == null) {
 synchronized (Singleton2.class) {
 if (singleton == null) {
 singleton = new Singleton();
 }
 }
 }
 return singleton;
 }
}

## 3.3 工厂设计模式

工厂模式分为工厂方法模式和抽象工厂模式。工厂方法模式分为三种

* 普通工厂模式，就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建
* 多个工厂方法模式，是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象
* 静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可

### 普通工厂模式

public interface Sender {
 public void Send();
}
public class MailSender implements Sender {

 @Override
 public void Send() {
 System.out.println("this is mail sender!");
 }
}
public class SmsSender implements Sender {

 @Override
 public void Send() {
 System.out.println("this is sms sender!");
 }
}
public class SendFactory {
 public Sender produce(String type) {
 if ("mail".equals(type)) {
 return new MailSender();
 } else if ("sms".equals(type)) {
 return new SmsSender();
 } else {
 System.out.println("请输入正确的类型!");
 return null;
 }
 }
}

### 多个工厂方法模式

该模式是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

public class SendFactory {
 public Sender produceMail(){
 return new MailSender();
 }

 public Sender produceSms(){
 return new SmsSender();
 }
}

public class FactoryTest {
 public static void main(String[] args) {
 SendFactory factory = new SendFactory();
 Sender sender = factory.produceMail();
 sender.send();
 }
}

静态工厂方法模式，将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

public class SendFactory {
 public static Sender produceMail(){
 return new MailSender();
 }

 public static Sender produceSms(){
 return new SmsSender();
 }
}

public class FactoryTest {
 public static void main(String[] args) {
 Sender sender = SendFactory.produceMail();
 sender.send();
 }
}

### 抽象工厂模式

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

public interface Provider {
 public Sender produce();
}

public interface Sender {
 public void send();
}

public class MailSender implements Sender {

 @Override
 public void send() {
 System.out.println("this is mail sender!");
 }
}

public class SmsSender implements Sender {

 @Override
 public void send() {
 System.out.println("this is sms sender!");
 }
}

public class SendSmsFactory implements Provider {

 @Override
 public Sender produce() {
 return new SmsSender();
 }
}
public class SendMailFactory implements Provider {

 @Override
 public Sender produce() {
 return new MailSender();
 }
}

public class Test {
 public static void main(String[] args) {
 Provider provider = new SendMailFactory();
 Sender sender = provider.produce();
 sender.send();
 }
}

## 3.4 建造者模式（Builder）

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。

public class Builder {
 private List<Sender> list = new ArrayList<Sender>();

 public void produceMailSender(int count) {
 for (int i = 0; i < count; i++) {
 list.add(new MailSender());
 }
 }

 public void produceSmsSender(int count) {
 for (int i = 0; i < count; i++) {
 list.add(new SmsSender());
 }
 }
}

public class TestBuilder {
 public static void main(String[] args) {
 Builder builder = new Builder();
 builder.produceMailSender(10);
 }
}

## 3.5 适配器设计模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

### 类的适配器模式

public class Source {
 public void method1() {
 System.out.println("this is original method!");
 }
}

public interface Targetable {
 /\* 与原类中的方法相同 \*/
 public void method1();
 /\* 新类的方法 \*/
 public void method2();
}

public class Adapter extends Source implements Targetable {
 @Override
 public void method2() {
 System.out.println("this is the targetable method!");
 }
}

public class AdapterTest {
 public static void main(String[] args) {
 Targetable target = new Adapter();
 target.method1();
 target.method2();
 }
}

### 对象的适配器模式

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。

public class Wrapper implements Targetable {
 private Source source;

 public Wrapper(Source source) {
 super();
 this.source = source;
 }

 @Override
 public void method2() {
 System.out.println("this is the targetable method!");
 }

 @Override
 public void method1() {
 source.method1();
 }
}

public class AdapterTest {
 public static void main(String[] args) {
 Source source = new Source();
 Targetable target = new Wrapper(source);
 target.method1();
 target.method2();
 }
}

### 接口的适配器模式

接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。

## 3.6 装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

public interface Sourceable {
 public void method();
}

public class Source implements Sourceable {
 @Override
 public void method() {
 System.out.println("the original method!");
 }
}

public class Decorator implements Sourceable {
 private Sourceable source;
 public Decorator(Sourceable source) {
 super();
 this.source = source;
 }

 @Override
 public void method() {
 System.out.println("before decorator!");
 source.method();
 System.out.println("after decorator!");
 }
}

public class DecoratorTest {
 public static void main(String[] args) {
 Sourceable source = new Source();
 Sourceable obj = new Decorator(source);
 obj.method();
 }
}

## 3.7 策略模式（strategy）

策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数。策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

public interface ICalculator {
 public int calculate(String exp);
}

public class Minus extends AbstractCalculator implements ICalculator {

 @Override
 public int calculate(String exp) {
 int arrayInt[] = split(exp, "-");
 return arrayInt[0] - arrayInt[1];
 }
}

public class Plus extends AbstractCalculator implements ICalculator {

 @Override
 public int calculate(String exp) {
 int arrayInt[] = split(exp, "\\+");
 return arrayInt[0] + arrayInt[1];
 }
}

public class AbstractCalculator {
 public int[] split(String exp, String opt) {
 String array[] = exp.split(opt);
 int arrayInt[] = new int[2];
 arrayInt[0] = Integer.parseInt(array[0]);
 arrayInt[1] = Integer.parseInt(array[1]);
 return arrayInt;
 }
}

public class StrategyTest {
 public static void main(String[] args) {
 String exp = "2+8";
 ICalculator cal = new Plus();
 int result = cal.calculate(exp);
 System.out.println(result);
 }
}

## 3.8 观察者模式（Observer）

观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

public interface Observer {
 public void update();
}

public class Observer1 implements Observer {
 @Override
 public void update() {
 System.out.println("observer1 has received!");
 }
}

public class Observer2 implements Observer {
 @Override
 public void update() {
 System.out.println("observer2 has received!");
 }
}

public interface Subject {
 /\*增加观察者\*/
 public void add(Observer observer);

 /\*删除观察者\*/
 public void del(Observer observer);
 /\*通知所有的观察者\*/
 public void notifyObservers();

 /\*自身的操作\*/
 public void operation();
}

public abstract class AbstractSubject implements Subject {

 private Vector<Observer> vector = new Vector<Observer>();

 @Override
 public void add(Observer observer) {
 vector.add(observer);
 }

 @Override
 public void del(Observer observer) {
 vector.remove(observer);
 }

 @Override
 public void notifyObservers() {
 Enumeration<Observer> enumo = vector.elements();
 while (enumo.hasMoreElements()) {
 enumo.nextElement().update();
 }
 }
}

public class MySubject extends AbstractSubject {

 @Override
 public void operation() {
 System.out.println("update self!");
 notifyObservers();
 }
}

public class ObserverTest {
 public static void main(String[] args) {
 Subject sub = new MySubject();
 sub.add(new Observer1());
 sub.add(new Observer2());
 sub.operation();
 }
}