策略模式

1.策略模式介绍

在软件开发中经常会遇到这样的情况：实现某一个功能可以有多种算法或策略，我们根据实际情况选择不同的算法或策略来完成该功能。例如，排序算法，可以使用插入排序、归并排序、冒泡排序等。

针对这种情况啊，一种常规的方法是将多种算法写在一个类中。例如,需要提供多种排序算法，可以将这些算法写到一个类中，每一种方法对应一个具体的排序算法；当然，也可以将这些排序封装在一个统一的方法中，通过if…else…或者case等条件判断语句来选择具体的算法。这两种实现方法我们都可以称之为硬编码。然而，当很多算法集中在一个类中时，这个类就会变得臃肿，这个类的维护成本会变高，在维护时更容易引发错误。如果我们需要增加一种新的排序算法，需要修改封装算法类的源代码。这就明显违反了常说的OCP原则和单一职责原则。

如果将这些算法或者策略抽象出来，提供一个统一的接口，不同的算法或策略有不同的实现类，这样在程序客户端就可以通过注入不同的实现对象来实现算法或策略的动态替换，这种模式的可扩展性、可维护性也就更高。

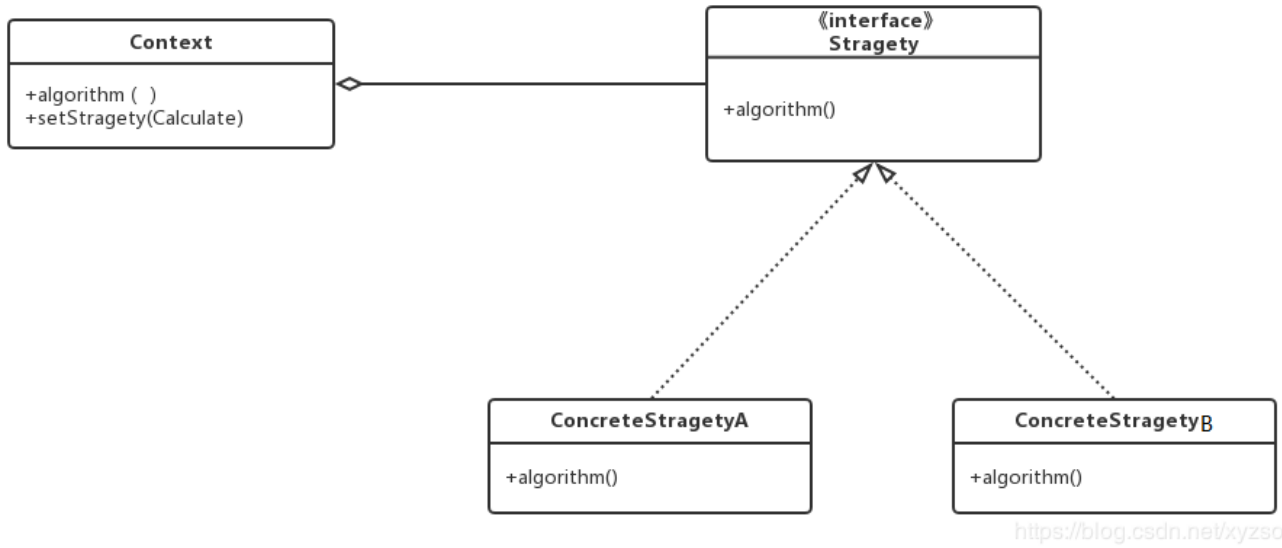
2.策略模式的定义

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

3.策略模式的使用场景

- 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；
- 需要安全的封装多种同一类型的操作时；
- 出现同一抽象类有多个子类，而有需要使用if-else或者switch-case来选择具体子类时；

4.策略模式的UML类图



角色介绍:

- Context：用来操作策略的上下文环境；
- Stragety：策略的抽象类；
- ConcreteStragetyA、ConcreteStragetyB：具体的策略实现；

5.策略模式的简单实现

通常如果一个问题有多个解决方案时，最简单的方式就是利用if-else或者switch-case 方法根据不同的情景选择不同的解决方案，但这种简单的方案问题太多，例如耦合性太高、代码臃肿、难以维护等。但是，如果解决方案中包括大量的处理逻辑需要封装，或者处理方式变动较大的时候则就显得混乱、复杂，当需要增加一种方案时就需要修改类中的代码。这种情况情况策略模式就能很好地解决这类问题，它将各种方案分离出来，让程序客户端根据具体的需求来动态的选择不同的策略方案。

下面我们以坐公共交通工具的费用计算来演示一个简单的示例。

计费规则：

1. 公交车 2元；
2. 地铁 ：

距离（km）	价格(元)
<=6	3
>6 &&<=12	4
>12 &&<=22	5
>22 &&<=32	6
>32	7

第一个版本

```
public class PriceCalculator {
        // 公交车类型
        private static final int BUS = 1;
        // 地铁类型
        private static final int SUBWAY = 2;

        public static void main(String[] args) {
           PriceCalculator calculator=new PriceCalculator();
           System.out.println("坐16公里的公交车票价为："+calculator.calculatePrice(16, BUS));
           System.out.println("坐16公里的地铁票价为："+calculator.calculatePrice(16, SUBWAY));
        }

        /*
         * 公交车票价按次收费 每次2元
         */
        private int busPrice(int km) {
                return 2;
        }
        /*
         * 地铁6公里内3元 6~12公里4元 12~22公里5元 22~32公里6元
         */
        private int subwayPrice(int km) {
                if (km <= 6) {
                        return 3;
                } else if (km <= 12) {
                        return 4;
                } else if (km <= 22) {
                        return 5;
                } else if (km <= 32) {
                        return 6;
                } else {
                        return 7;
                }
        }

        private int calculatePrice(int km, int type) {
                if (type == BUS) {
                        return busPrice(km);
                } else if (type == SUBWAY) {
                        return subwayPrice(km);
                }
                return 0;
        }
}

```

PriceCalculator类很明显的问题就是并不是单一职责（一个类中应该是一组相关性很高的函数、数据的封装），首先它承担了计算公交车和地铁坐价格的职责；另一个问题就是通过if-else的形式来判断使用哪种计算形式。当我们增加一种出行方式时，如出租车，那么我们就需要在PriceCalculator中增加一个方法来计算出租车出行的价格，并且在calculatePircate(int km, int type)方法中增加一个判断，代码：

```
public class PriceCalculator {
        // 公交车类型
        private static final int BUS = 1;
        // 地铁类型

```

```
        private static final int SUBWAY = 2;
        // 出租车类型
        private static final int TAXI = 3;
        /*
         * 2公里内11元 超过部分每公里2.4元
         */
        private double taxiPrice(double km) {
                if (km <= 2) {
                        return 11;
                } else {
                        return 2.4 * (km - 2) + 11;
                }
        }
        private double calculatePrice(double km, int type) {
                if (type == BUS) {
                        return busPrice(km);
                } else if (type == SUBWAY) {
                        return subwayPrice(km);
                } else if (type == TAXI) {
                        return taxiPrice(km);
                }
                return 0;
        }
```

此时的代码已经比较混乱了，各种if-else 语句缠绕其中。当价格的计算方法变化时，需要直接修改这个类中的代码，那么很可能有一段代码是其它几个计算方法所共用的，这就容易引入错误。另外，在增加出行方式时，我们有需要在calculatePrice中添加if-else，此时很可能就是复制上一个if-else，然后手动进行修改，手动复制代码也是很容易引入错误的做法之一。这类代码必然是很难以应对变化的，它会使得代码变得越来越臃肿，难以维护，下面使用策略模式修改上面代码。

```
public interface Animal {
   public void breath();
}

public class Cat implements Animal{

        @Override
        public void breath() {
                System.out.println("用肺呼吸");
        }

}

public class Fish implements Animal{

        @Override
        public void breath() {
                System.out.println("用腮呼吸");
        }

}

public class Client {
        public static void main(String[] args) {
        Animal animal=new Cat();

```

```
        animal.breath();
        }
}

//输出：
用肺呼吸
```

升级版本

首先我们需要定义一个抽象的价格计算接口，这里命名为CalculateStrategy:

```
/*
 * 计算接口
 */
public interface CalculateStragegy {
        /*
         * 按距离计算价格
         */
        int calculatePrice(int km);
}
```

对于每一种出行方式我们都有一个独立的计算策略类，这些策略类都实现了CalculateStrategy接口，例如下面是公交车和地铁的计算策略类：

```
//公交计算策略
public class BusStrategy implements CalculateStragegy{

        /*
         * 公交车每次2元
         */
        @Override
        public int calculatePrice(int km) {
                return 2;
        }
}

//地铁计算策略
public class SubwayStratety implements CalculateStragegy {
        /*
         * 6公里内3元 6~12公里4元 12~22公里5元 22~32公里6元
         */
        @Override
        public int calculatePrice(int km) {
                if (km <= 6) {
                        return 3;
                } else if (km <= 12) {
                        return 4;
                } else if (km <= 22) {
                        return 5;
                } else if (km <= 32) {
                        return 6;
                } else {
                        return 7;
                }
        }
}
```

我们再创建一个扮演Context角色的类，这里命名为TranficCalculator，具体代码如下：

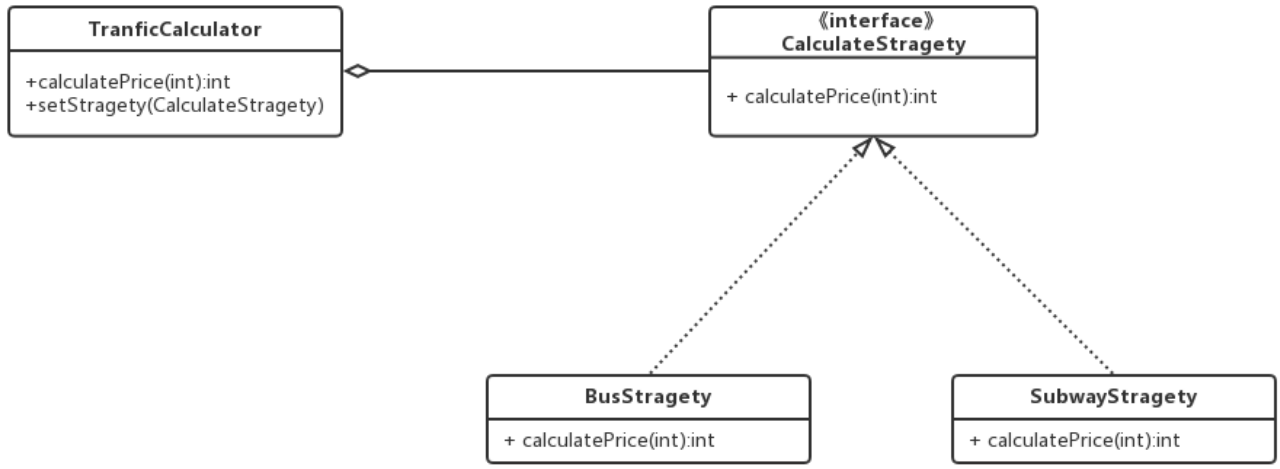
```
public class TranficCalculator {
        public static void main(String[] args) {
                TranficCalculator calculator = new TranficCalculator();
        //设置计算策略
                calculator.SetStragegy(new BusStrategy());
                //计算价格
                System.out.println("公交车乘16公里的价格: "+calculator.calculatePrice(16));
        }

        CalculateStragegy mStragegy;

        public void SetStragegy(CalculateStragegy mStragegy) {
                this.mStragegy = mStragegy;
        }

        public double calculatePrice(int km) {
                return mStragegy.calculatePrice(km);
        }
}
```

经过上述的重构之后，去掉了各种各样的if-else语句，结构变得也清晰，其结构如下：



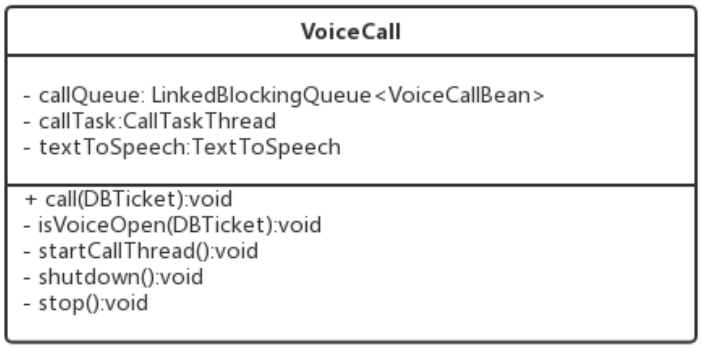
这种方案在隐藏实现的同时，可扩展性变得更强，例如，当我们需要添加出租车的计算策略时，只需要添加一个出租车计算策略类，然后将该策略设置给TranficCalculator，最好直接通过TranficCalculator对象的计算方法即可。

6.项目中的应用

项目中 由讯飞语音播报 改为由用户设置使用讯飞语音或百度语音

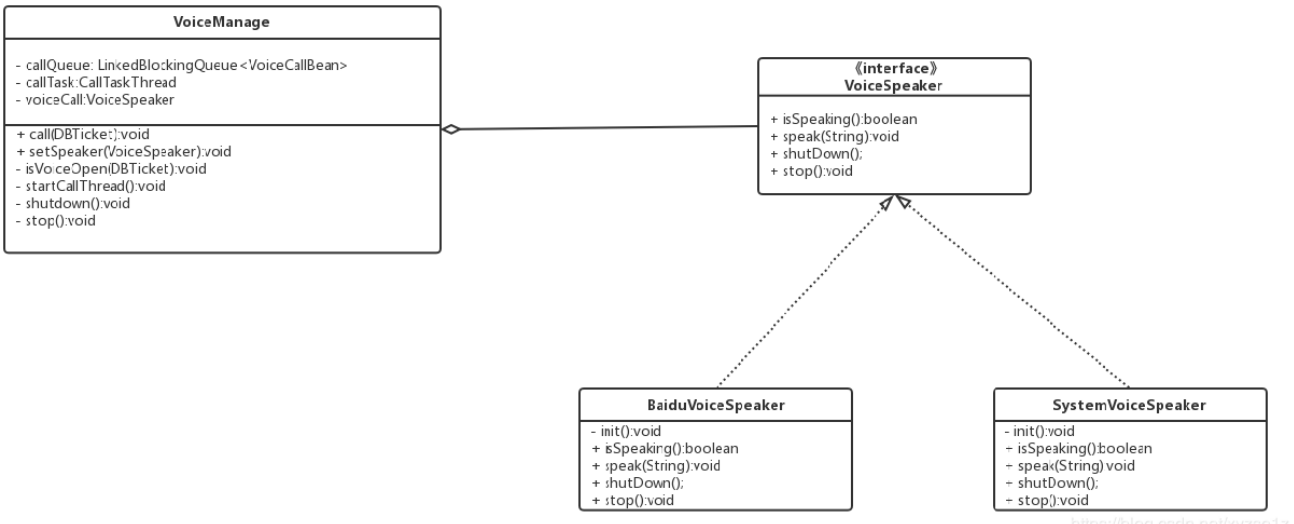
修改过程

现有代码结构



- callQueue:保存语音对象队列
- callTask:执行语音合成的任务
- textToSpeech:语音合成的具体执行者

因为讯飞语音合成和百度语音合成是平行的处理方式，仅仅是具体行为有差别，因此可以把两种处理共有方法提取出来抽象成类VoiceSpeaker，让其(BaiduSpeaker、SystemVoiceSpeaker) 继承该类，在VoiceManager类中声明voiceSpeaker对象，通过 + setSpeaker(VoiceSpeaker speaker):void 给voiceSpeaker赋值。
类图:



核心代码：
VoidSpeaker: 该类包含讯飞语音和百度语音对外提供的共有方法

```
abstract class VoiceSpeaker {
    /**
     * @return 是否正在合成
     */
    abstract boolean isSpeaking();

    /**
     * @param text 需要合成的内容
```

```
     */
    abstract void speak(String text);

    /**
     * 关闭
     */
    abstract void shutDown();

    /**
     * 停止
     */
    abstract void stop();
}
```

BaiduSpeaker:百度语音合成逻辑，

```
/**
 * 百度语音合成
 */
public class BaiduVoiceSpeaker extends VoiceSpeaker implements SpeechSynthesizerListener {
    SpeechSynthesizer mSpeechSynthesizer;
    private volatile int speakingTime = 0;
    private final byte[] lock = new byte[0];

    public BaiduVoiceSpeaker(Context context) {
        this.context = context;
        init();
    }

    @Override
    public boolean isSpeaking() {
        return speakingTime > 0;
    }

    @Override
    public void speak(String text) {
      //具体合成逻辑
    }

    @Override
    public void shutDown() {
    }

    @Override
    public void stop() {
        speakingTime = 0;
        if (mSpeechSynthesizer != null) {
            mSpeechSynthesizer.stop();
        }
    }

    /**
     * 不再使用后，请释放资源
     */
    public void release() {
        if (mSpeechSynthesizer != null) {

```

```
            mSpeechSynthesizer.release();
        }
    }

    private void init() {
    //初始化
    }
}
```

SystemVoiceSpeaker :讯飞语音合成具体逻辑

```
/**
 * Created by Charliu on 2017/3/30.
 * 系统自带TTS 播报
 */
public class SystemVoiceSpeaker extends VoiceSpeaker {

    private TextToSpeech textToSpeech;

    public SystemVoiceSpeaker(Context context) {
        textToSpeech = new TextToSpeech(context, new TextToSpeech.OnInitListener() {
            @Override
            public void onInit(int status) {

            }
        });
        textToSpeech.setSpeechRate(0.8f);
    }
    @Override
    boolean isSpeaking() {
        return textToSpeech.isSpeaking();
    }

    @Override
    void speak(String text) {
        textToSpeech.speak(text, TextToSpeech.QUEUE_ADD, null);
    }

    @Override
    void shutDown() {
        textToSpeech.shutdown();
    }

    @Override
    void stop() {
        textToSpeech.stop();
    }
}
```

VoiceCallManager :

```
public class VoiceCallManager {
    private LinkedBlockingQueue<VoiceCallBean> callQueue = new LinkedBlockingQueue<>();
    private CallTaskThread callTask;
    private VoiceSpeaker voiceCall;

```

```
    public VoiceCallManager(VoiceSpeaker voiceCall) {
        this.voiceCall = voiceCall;
        startCallThread();
    }

    public void call(DBTicket dbTicket) {
        if (dbTicket != null && isVoiceOpen(dbTicket)) {
            String callText = ConvertUtil.generateCallText(dbTicket);
            int count = 2;
            try {
                count = App.getApp().getHospital().getConfig().getVoiceCallCount();
            } catch (Exception e) {
                e.printStackTrace();
                count = 2;
            }
            if (callText != null) {
                call(new VoiceCallBean(callText, count));
            }
        }
    }

    private boolean isVoiceOpen(DBTicket dbTicket) {
         //doSomething
    }

    private void call(VoiceCallBean call) {
        callQueue.add(call);
    }

    public void cancel(VoiceCallBean call) {
        callQueue.remove(call);
    }

    public void clear() {
        callQueue.clear();
    }

    private void startCallThread() {
        if (callTask == null || !callTask.isAlive()) {
            callTask = new CallTaskThread(callQueue);
            callTask.start();
        }
    }

    public void shutdown() {
        stop();
        if (voiceCall != null) {
            voiceCall.shutDown();
        }
    }

    public void stop() {
        voiceCall.stop();
        if (callTask != null) {
            callTask.stopTask();
            callTask = null;
```

```
        }
    }

    private class CallTaskThread extends Thread {
        private final LinkedBlockingQueue<VoiceCallBean> queue;
        private boolean running = true;

        CallTaskThread(LinkedBlockingQueue<VoiceCallBean> callQueue) {
            queue = callQueue;
        }

        @Override
        public void run() {
            while (running) {
                try {
                    if (voiceCall.isSpeaking()) {
                        TimeUnit.SECONDS.sleep(3);
                        continue;
                    }
                    VoiceCallBean call = queue.take();
                    for (int i = 0; i < call.callTime; i++) {
                        voiceCall.speak(call.callText);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    break;
                }
            }
        }

        public void stopTask() {
            this.running = false;
            interrupt();
        }
    }



    /**
     * Created by Charliu on 2017/3/30.
     */

    public static class VoiceCallBean {
        public String callText;
        public int callTime = 2;//默认叫两次

        public VoiceCallBean(String text, int callTime) {
            this.callText = text;
            if (callTime < 0) {
                throw new IllegalArgumentException("Call time must be big than zero!");
            }
            this.callTime = callTime;
        }
    }
}
```

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：

- 结构清晰明了、使用简单直观
- 耦合度相对而言较低，扩展方便
- 操作封装也更加彻底，数据更为安全

缺点：

- 随着策略的增加，子类也会变得繁多