状态模式

1.状态模式介绍

状态模式中行为是由状态来决定的，不同的状态下有不同的行为。状态模式和策略模式的结构几乎完全一样，但它们的目的、本质却完全不一样。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、可相互替换的。用一句话来表述，状态模式把对象的行为包装在不同的状态对象里，每一个状态对象都有一个共同的抽象状态基类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。

2.状态模式的定义

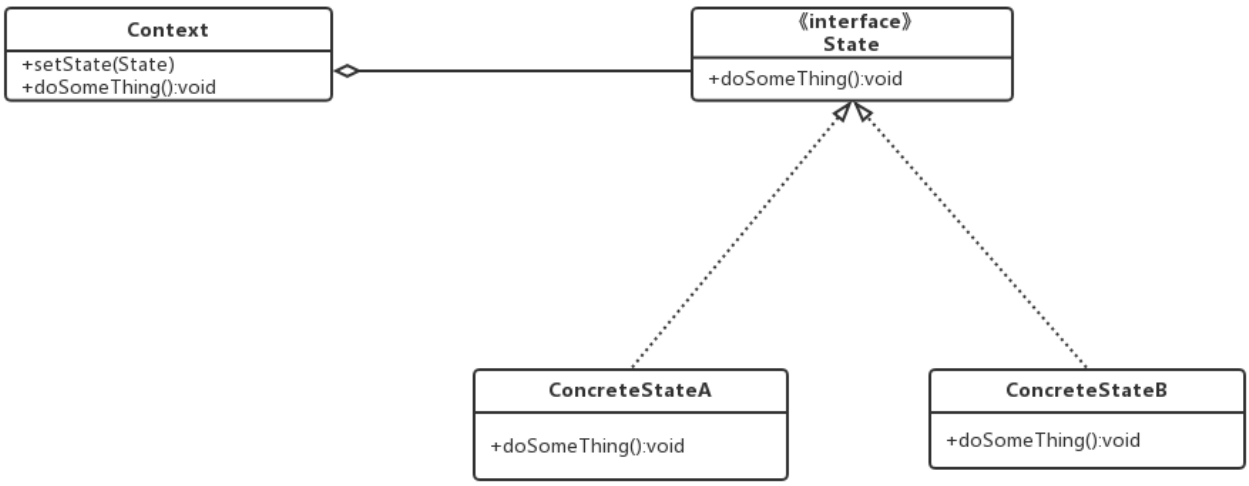
当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

3.状态模式的使用场景

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，一个操作中含有庞大的多分支语句（if-else或switch-case），且这些分支依赖于该对象的状态。
   状态模式将每一个条件分支放入一个独立的类中，这使得你可以根据对象自身情况将对象的状态作为一个对象，这一对象可以不依赖于其它对象而独立变化，这样通过多态来去除多的、重复的if-else等分支语句。

4.状态模式的UML图

UML类图如下：



角色介绍：

- Context：环境类，定义客户感兴趣的接口，维护一个State子类的实例，这个实例定义了对象的当前状态。
- State：抽象状态类或者状态接口，定义一个或者一组接口，表示该状态下的行为。
- ConcreteStateA、ConcreteStateB:具体状态类，每一个具体的状态类实现抽象State中定义的接口，从而大道不同状态下的不同行为。

5.状态模式的简单示例

下面我们就以电视遥控器为例来演示一下状态模式的实现。我们首先将电视的状态简单分为开机状态和关机状态，在开机状态下可以通过遥控器进行频道切换、调整音量等操作，但是，此时重复按开机键是无效的；而在关机状态下，频道切换、调整音量、关机都是无效的操作，只有按开机按钮时会生效。也就是说电视的内部状态决定了遥控器的行为，我们看看第一版实现：

```
/*
 * 电视遥控器，含有开机、关机、下一频道、上一频道、调高音量、调低音量这几个功能
 */
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;
        private int mState=POWER_OFF;
        public void powerOn(){
                mState=POWER_ON;
                if (mState==POWER_OFF) {
                        System.out.println("开机了");
                }
        }

        public void powerOff(){
                mState=POWER_OFF;
                if (mState==POWER_ON) {
```

```
                        System.out.println("关机啦");
                }
        }

        public void nextChannel(){
                if (mState==POWER_ON) {
                        System.out.println("下一个频道");
                }else{
                        System.out.println("两个红灯提示没有开机");
                }
        }

        public void prevChannel(){
                if (mState==POWER_ON) {
                        System.out.println("上一个频道");
                }else{
                        System.out.println("两个红灯提示没有开机");
                }
        }

        public void turnUp(){
                if (mState==POWER_ON) {
                        System.out.println("调高音量");
                }else{
                        System.out.println("两个红灯提示没有开机");
                }
        }

        public void turnDown(){
                if (mState==POWER_ON) {
                        if (mState==POWER_ON) {
                                System.out.println("调低音量");
                        }else{
                                System.out.println("两个红灯提示没有开机");
                        }
                }
        }

}

```

可以看到，在TvController类中，通过mState字段存储了电视的状态，并且在各个操作中根据状态来判断是否应该执行。这就导致你了在每个功能中都需要使用if-else，代码重复、相对较为混乱，这是在只有两个状态和简单几个功能函数的情况下，那么当状态变成5个、功能函数变为10个呢？每个函数中都要用if-else进行判断，而这些代码都充斥在一个类中，这些重复的代码无法被提取出来，这使得这个类变得越来越难以维护。

状态模式就是为了解决这类问题而出现的，我们将这些状态用对象来代替，将这些行为封装到对象中，使得在不同的状态下有不同的实现，这样就将这些if-else从TvController类中去掉，整个结构也变得清晰起来。

```
//电视状态接口，定义了电视操作的函数
public interface TvState {
    public void nextChannel();
    public void prevChannel();
    public void turnUp();

```

```
    public void turnDown();

}


//关机状态，此时只有开机功能是有效的
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，此时再触发开机功能不做任何操作
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
public interface PowerController {
   public void powerOn();

   public void powerOff();
}

//电视遥控器，类似于经典状态模式中的Context
public class TvController implements PowerController {
   TvState mTvState;

   public void setmTvState(TvState mTvState) {
        this.mTvState = mTvState;
   }

   @Override
   public void powerOn() {
        setmTvState(new PowerOnState());
        System.out.println("开机啦");
   }

   @Override
   public void powerOff() {
        setmTvState(new PowerOffState());
        System.out.println("关机啦");
   }

   public void nextChannel() {
        mTvState.nextChannel();
   }

   public void prevChannel() {
        mTvState.prevChannel();
   }

   public void turnUp() {
        mTvState.turnUp();
   }

   public void turnDown() {
        mTvState.turnDown();
   }
}
```

下面是客户端调用的代码;

```
public class Client {
        public static void main(String[] args) {
           TvController tvController=new TvController();
           //设置开机状态
           tvController.powerOn();
           //下一个频道
           tvController.nextChannel();

```

```
            //调高音量
            tvController.turnUp();
            //设置关机
            tvController.powerOff();
            //调高音量，此时不会生效
            tvController.turnUp();
        }
}
```

输出结果如下：

```
开机啦
下一频道
调高音量
关机啦
```

上述实现中，我们抽象了一个TvState接口，该接口中有操作电视的所有函数，该接口有两个实现类，即开机状态（PowerOnState）和关机状态（PowerOffState）。开机状态下只有开机功能是无效的，也就是说在已经开机的时候用户在按下开机键不会产生任何反应；而在关机状态下，只有开机功能是可用的，其它功能都不会生效。同一个操作，如调高音量的turnUp函数，在关机状态下无效，在开机状态下就会将电视的音量调高，也就是说电视的内部状态影响了电视遥控器的行为。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，不同的状下有不同的实现，这样就通过多态的形式去除了重复、杂乱的if-else语句，这也是状态模式的精髓所在。

6.状态模式实战

在开发过程中，我们用到状态模式最常见的地方应该是用户登录系统。在用户已登录和未登录的情况下，对于同一事件的处理行为是不一样的，例如，在新浪微博中，用户在未登录的情况下点击转发按钮，此时会先让用户登录，然后再执行操作；如果是已登录的情况下，那么用户输入转发的内容就可以直接进行转发。

总结

状态模式的关键点在于不同的状态下对于同一行为有不同的响应，这其实就是一个将if-else用多态来实现的一个具体示例。在if-else或者switch-case形式下根据不同的状态进行判断，如果是状态A那么执行方法A、状态B执行方法B,但这种实现使得逻辑耦合在一起，易于出错，通过状态模式能够很好地消除这类“丑陋”的逻辑处理，当然并不是任何出现if-else的地方都应该通过状态模式重构，模式的运用一定要考虑所处的情景以及你要解决的问题，只有符合特定的场景才建议使用对应的模式。

状态模式的优点：

State模式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。

状态模式的缺点：

状态模式的使用必然会增加系统类和对象个数。