**使用场景：**

**状态模式**：当对象的行为随对象的状态的改变而改变时，我们为了解耦多重判断条件，封装行为的变化，可以定义一个抽象的状态类，提供对象行为接口。具体与状态相关的行为，由它的子类去实现。

**策略模式**：“策略”二字等同于算法，当现实系统中需要对算法动态指定，并且可以互相替换，可以抽象出算法的调用接口，具体的算法实现由具体的策略角色去实现，根据里氏替换原则，任何出现父类的地方都可以使用它的子类去替换，这样符合我们的业务需求。

      我们是如何区分不同的状态？就是通过每个状态不同的行为来区分，所以不同的状态都有相同的动作，但是这个动作的执行结果是不同的，而且执行结果还有可能会修改当前的状态。为了体现多态，摆脱对具体状态类的依赖，**使用Context可以动态的替换状态类**，就像策略模式一样。

**状态的转换不一定是放在Context中，有时候状态类本身也会自动切换状态。**当状态的转换是固定的，像是这里的容量为0，就变成空桶状态，我们可以在Context中完成转换，但如果转换是动态的，也就是没有固定的判断条件，像是一完成就自动切换，我们可以放在状态类中。

      但将状态的切换放在状态类中，会让状态类间产生依赖，而且状态类还需要拥有Context类的引用才能切换状态，也就是采用观察者模式的方法来通知Context类更新状态。所以，是否要这样做，就看具体的编程环境以及我们的经验了。

      程序员的经验是非常重要的，它决定我们是否可以达到更高的境界，新手是在积累经验，等累积到一定的程度，编程的时候就基本靠经验的驱动了，什么是安全的代码，什么样的代码扩展性好，都会在不知不觉间体现出来，不需要特意从头脑中挖出来。

**状态对象也是可以共享的，但前提就是状态对象不能持有它们自己的内部状态，否则无法保证另一个线程得到的对象是正确的状态。所以，我们如果想要共享状态，需要把每个状态都指定到静态的实例变量中。**

      状态模式的意图其实非常简单：将与状态有关的处理逻辑分散到代表对象状态的各个类中，但我们的Context类必须拥有这些状态对象集合的引用，这也就引出一个问题：实例化Context类对象会导致状态类对象集合初始化方面的问题，也就是对象的依赖问题。

**为了尽可能减少这方面的依赖，我们的Context类通常拥有的只是一个状态类的抽象引用，然后设置一个setter以便动态的更改状态类对象。**

      如果不是使用对象，而是采用常量的方法：

private static final int FULL\_STATE = 0;

private static final int NULL\_STATE = 1;

      也可以使用enum将它们封装起来：

enum STATE{ NULL\_STATE, FULL\_STATE};

      采用这种做法是因为我们需要根据当前的状态自动跳转到下一个状态，比如说，饮水机可以在空桶的状态自动加水：

if(state == NULL\_STATE){

capacity++;

if(capacity == 20){

state = FULL\_STATE;

}

}

 这种做法非茶馆常见，因为我们只是想要有一个状态的判断和切换的简单动作，并不需要特意创建一个对象，因为这些状态只是单纯的标识，没有任何的职责。**但程序中出现过多的静态变量总是让人觉得这个程序的设计不具有良好的弹性，如果可以从这些状态中提取出抽象，可以考虑使用状态模式来优化我们的代码结构。**

**之所以说状态模式是策略模式的孪生兄弟，是因为它们的UML图是一样的，但意图却完全不一样，策略模式是让用户指定更换的策略算法，而状态模式是状态在满足一定条件下的自动更换，用户无法指定状态，最多只能设置初始状态。**