私塾在线《研磨设计模式》
——跟着CC学设计系列精品教程

本节课程概览

- 学习状态模式

一：初识状态模式

包括：定义、结构、参考实现

二：体会状态模式

包括：场景问题、不用模式的解决方案、使用模式的解决方案

三：理解状态模式

包括：认识状态模式、状态的维护和转换控制 、使用数据库来维护状态 、模拟工作流 、状态模式的优缺点

四：思考状态模式

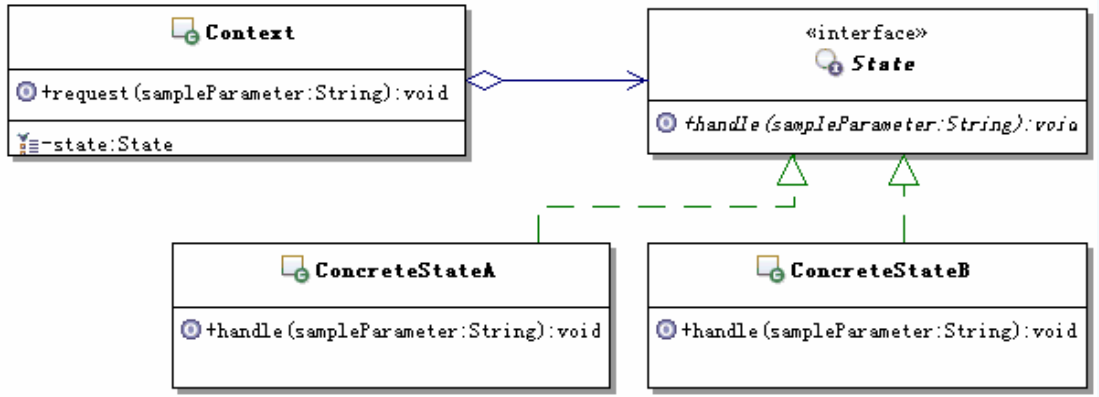
包括：状态模式的本质、何时选用

初识状态模式

- 定义

  允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

- 结构和说明



Context：环境，也称上下文，通常用来定义客户感兴趣的接口，同时维护一个来具体处理当前状态的实例对象。

State：状态接口，用来封装与上下文的一个特定状态所对应的行为。

ConcreteState：具体实现状态处理的类，每个类实现一个状态的具体处理。

体会状态模式

- 实现在线投票

考虑一个在线投票的应用，要实现控制同一个用户只能投一票，如果一个用户反复投票，而且投票次数超过5次，则判定为恶意刷票，要取消该用户投票的资格，当然同时也要取消他所投的票。如果一个用户的投票次数超过8次，将进入黑名单，禁止再登录和使用系统。

该怎么实现这样的功能呢？

体会状态模式

- 不用模式的解决方案

- 有何问题

看起来很简单，是不是？但是你想想，在vote()方法中那么多判断，还有每个判断对应的功能处理都放在一起，是不是有点太杂乱了，那简直就是个大杂烩，如果把每个功能都完整的实现出来，那vote()方法会很长的。

一个问题是：如果现在要修改某种投票情况所对应的具体功能处理，那就需要在那个大杂烩里面，找到相应的代码块，然后进行改动。

另外一个问题是：如果要添加新的功能，比如投票超过8次但不足10次的，给个机会，只是禁止登录和使用系统3天，如果再犯，才永久封掉账号，该怎么办呢？那就需要改动投票管理的源代码，在上面的if-else结构中再添加一个else if块进行处理。

该如何实现才能做到：既能够很容易的给vote()方法添加新的功能，又能够很方便的修改已有的功能处理呢？

体会状态模式

- 使用模式来解决的思路

仔细分析上面的问题，会发现，那几种用户投票的类型，就相当于是描述了人员的几种投票状态，而各个状态和对应的功能处理具有很强的对应性，有点类似于“一个萝卜一个坑”，各个状态下的处理基本上都是不一样的，也不存在可以相互替换的可能。

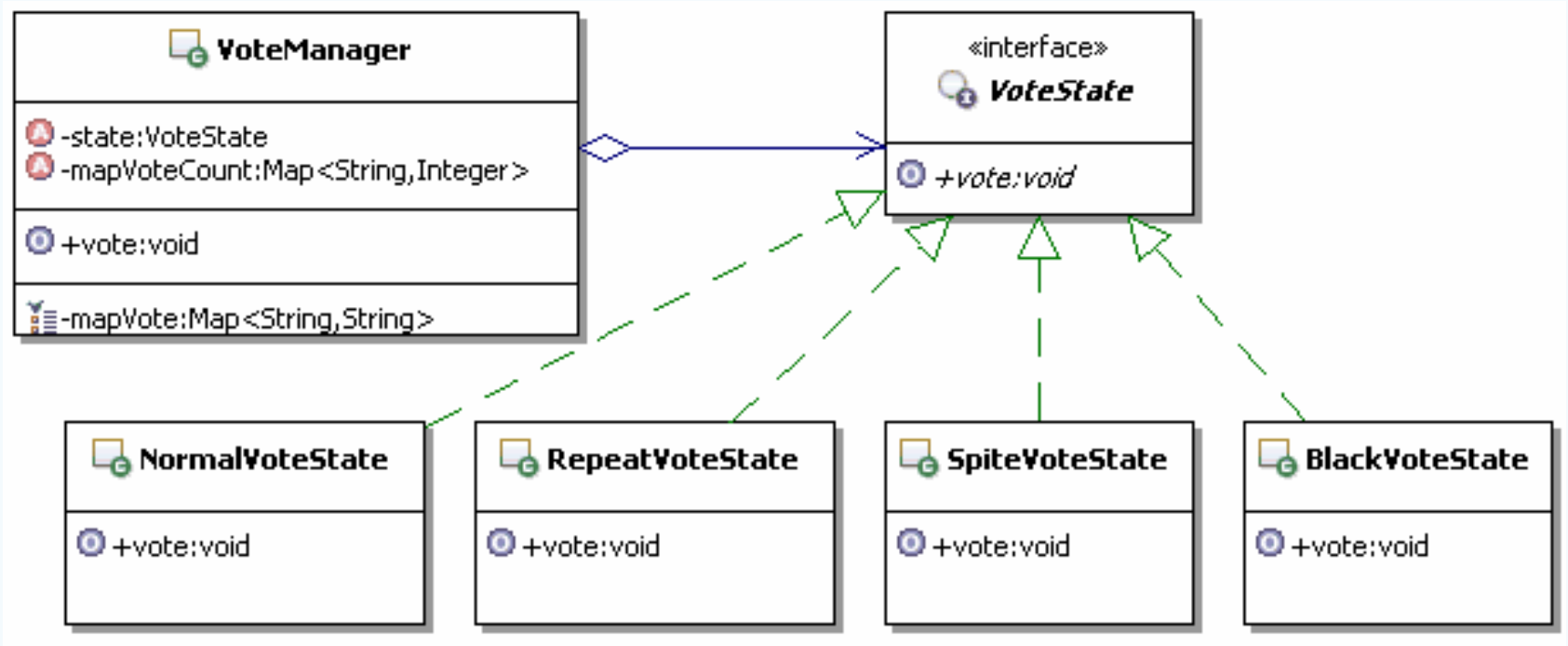
为解决上面的问题，很自然的一个设计就是首先把状态和状态对应的行为从原来的大杂烩代码中分离出来，把每个状态所对应的功能处理封装在一个独立的类里面，这样选择不同处理的时候，其实就是在选择不同的状态处理类。

然后为了统一操作这些不同的状态类，定义一个状态接口来约束它们，这样外部就可以面向这个统一的状态接口编程，而无需关心具体的状态类实现了。

这样一来，要修改某种投票情况所对应的具体功能处理，那就是直接修改或者扩展某个状态处理类的功能就可以了。而要添加新的功能就更简单，直接添加新的状态处理类就可以了，当然在使用Context的时候，需要设置使用这个新的状态类的实例。

体会状态模式

- 使用模式的解决方案的类图



理解状态模式

- 认识状态模式

1：状态和行为

所谓对象的状态，通常指的就是对象实例的属性的值；而行为指的就是对象的功能，再具体点说，行为多半可以对应到方法上。

状态模式的功能就是分离状态的行为，通过维护状态的变化，来调用不同的状态对应的不同的功能。

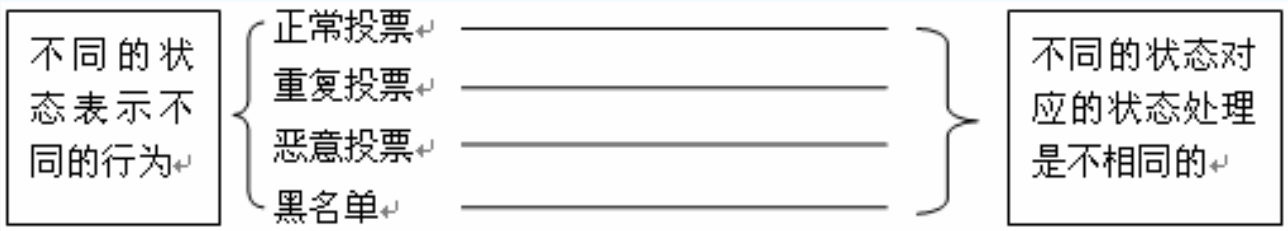
也就是说，状态和行为是相关联的，它们的关系可描述为：**状态决定行为**

由于状态是在运行期被改变的，因此行为也会在运行期，根据状态的改变而改变，看起来，同一个对象，在不同的运行时刻，行为是不一样的，就像是类被修改了一样

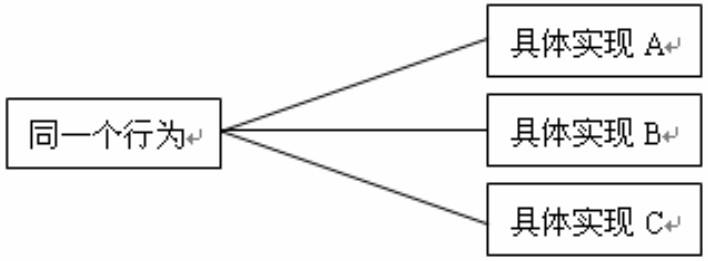
2：行为的平行性

注意是平行性而不是平等性。所谓平行性指的是各个状态的行为所处的层次是一样的，相互是独立的、没有关联的，是根据不同的状态来决定到底走平行线的那一条，行为是不同的，当然对应的实现也是不同的，相互之间不可替换。

理解状态模式



而平等性强调的是可替换性，大家是同一行为的不同描述或实现，因此在同一个行为发生的时候，可以根据条件来挑选任意一个实现来进行相应的处理。



大家可能会发现状态模式的结构和策略模式的结构完全一样，但是，它们的目的、实现、本质都是完全不一样的。这个行为之间的特性也是状态模式和策略模式一个很重要的区别，状态模式的行为是平行性的，不可相互替换的；而策略模式的行为是平等性的，是可以相互替换的

理解状态模式

3：上下文和状态处理对象

在状态模式中，上下文是持有状态的对象，但是上下文自身并不处理跟状态相关的行为，而是把处理状态的功能委托给了状态对应的状态处理类来处理。

在具体的状态处理类里面经常需要获取上下文自身的数据，甚至在必要的时候会回调上下文的方法，因此，通常将上下文自身当作一个参数传递给具体的状态处理类。

客户端一般只和上下文交互，客户端可以用状态对象来配置一个上下文，一旦配置完毕，就不再需要和状态对象打交道了，客户端通常不负责运行期间状态的维护，也不负责决定到底后续使用哪一个具体的状态处理对象。

4：不完美的OCP体验

使用状态模式来修改和扩展功能，是**没有完全遵循OCP原则** 的。由于状态的维护和转换在状态模式结构里面，不管你是扩展了状态实现类，还是新添加了状态实现类，都需要修改状态维护和转换的地方，以使用新的实现。

理解状态模式

5：创建和销毁状态对象

在应用状态模式的时候，有一个常见的考虑，那就是：究竟何时创建和销毁状态对象。常见的有几个选择：

1：一个是当需要使用状态对象的时候创建，使用完后就销毁它们

2：另一个是提前创建它们并且始终不销毁

3：还有一种是采用延迟加载和缓存合用的方式，就是当第一次需要使用状态对象的时候创建，使用完后并不销毁对象，而是把这个对象缓存起来，等待下一次使用，而且在合适的时候，会由缓存框架销毁状态对象

理解状态模式

怎么选择呢？下面给出选择建议：

1：如果要进入的状态在运行时是不可知的，而且上下文是比较稳定的，不会经常改变状态，而且使用也不频繁，这个时候建议选第一种方案。

2：如果状态改变很频繁，也就是需要频繁的创建状态对象，而且状态对象还存储着大量的信息数据，这种情况建议选第二种方案。

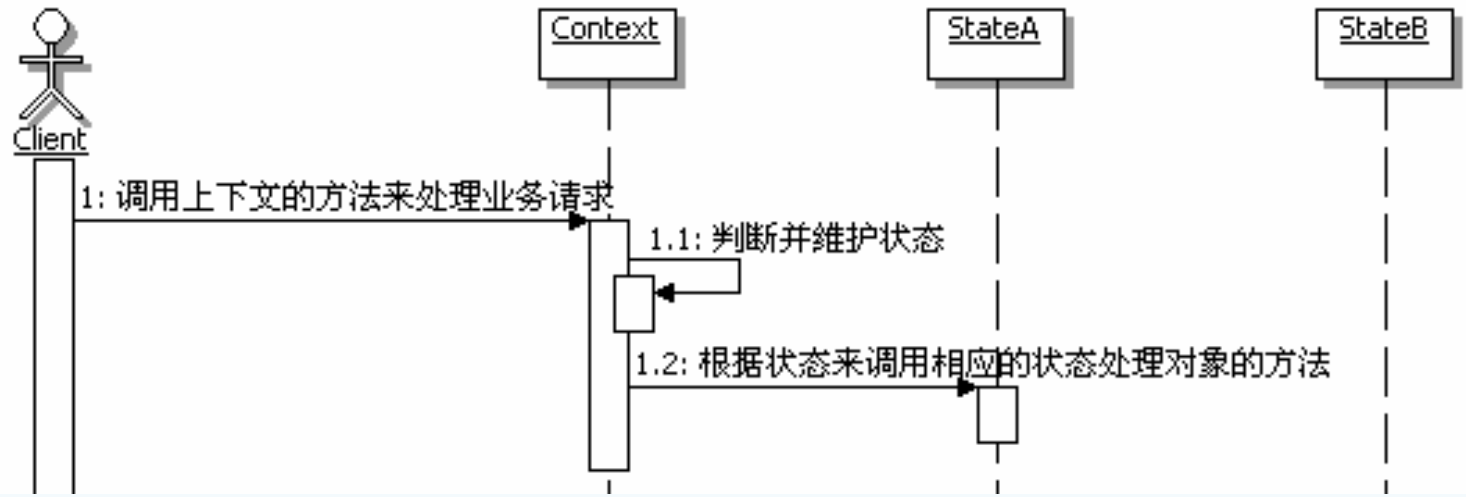
3：如果无法确定状态改变是否频繁，而且有些状态对象的状态数据量大，有些比较小，一切都是未知的，建议选第三种方案。

事实上，在实际工程开发中，第三种方案是首选，因为它兼顾了前面两种方案的优点，而又避免了它们的缺点，几乎能适应各种情况的需要。只是这个方案在实现的时候，要实现一个合理的缓存框架，而且要考虑多线程并发的问题，因为需要由缓存框架来在合适的时候销毁状态对象，因此实现上难度稍高点。另外在实现中还可以考虑结合享元模式，通过享元模式来共享状态对象。

理解状态模式

- 状态模式调用顺序示意图

前面的示例中，采用的是在Context中进行状态的维护和转换，这里就先画出这种方式的调用顺序示意图



理解状态模式

- 状态的维护和转换控制

所谓状态的维护，指的就是维护状态的数据，就是给状态设置不同的状态值；而状态的转换，指的就是根据状态的变化来选择不同的状态处理对象。在状态模式中，通常有两个地方可以进行状态的维护和转换控制。

一个就是在上下文当中，因为状态本身通常被实现为上下文对象的状态，因此可以在上下文里面进行状态维护，当然也就可以控制状态的转换了。前面投票的示例就是采用的这种方式。

另外一个地方就是在状态的处理类里面，当每个状态处理对象处理完自身状态所对应的功能后，可以根据需要指定后继的状态，以便让应用能正确处理后续的请求。

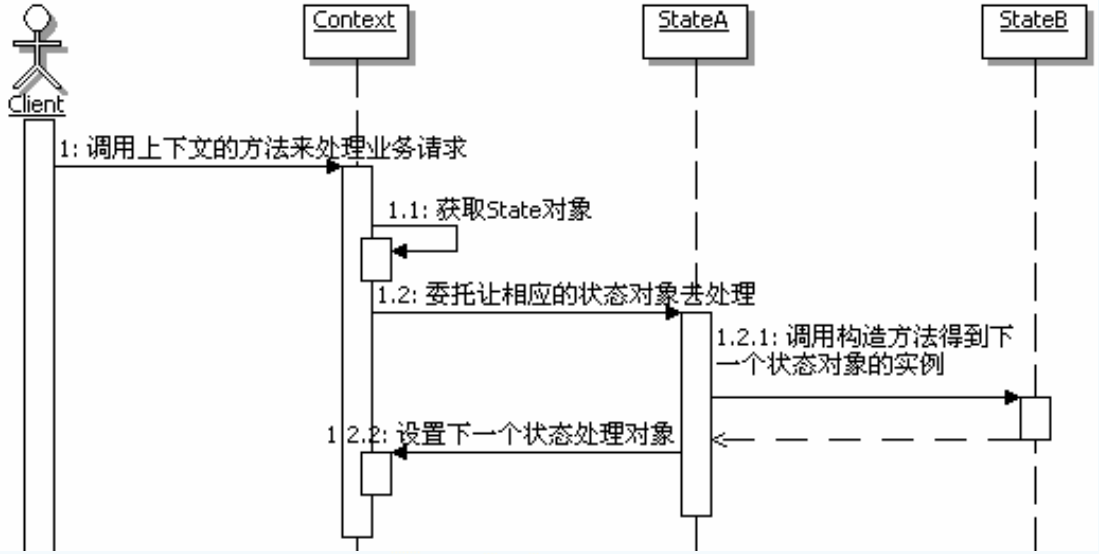
理解状态模式

- 那么到底如何选择这两种方式呢？

1：一般情况下，如果状态转换的规则是一定的，一般不需要进行什么扩展规则，那么就适合在上下文中统一进行状态的维护。

2：如果状态的转换取决于前一个状态动态处理的结果，或者是依赖于外部数据，为了增强灵活性，这种情况下，一般是在状态处理类里面进行状态的维护。

采用让状态对象来维护和转换状态的调用顺序示意图如图



理解状态模式

- 使用数据库来维护状态

在实际开发中，还有一个方式来维护状态，那就是使用数据库，在数据库中存储下一个状态的识别数据，也就是说，维护下一个状态，演化成了维护下一个状态的识别数据，比如状态编码。

这样在程序中，通过查询数据库中的数据来得到状态编码，然后再根据状态编码来创建出相应的状态对象，然后再委托相应的状态对象进行功能处理。

- 直接把“转移”记录到数据库中

还有一种情况是直接把“转移”记录到数据库中，这样会更灵活。**所谓转移，指的就是描述从A状态到B状态的这么一个转换变化。**

比如：在正常投票状态处理对象里面指定使用“转移A”，而“转移A”描述的就是从正常投票状态转换成重复投票状态。这样一来，假如今后想要让正常投票处理过后变换成恶意投票状态，那么就不需要修改程序，直接修改数据库中的数据，把数据库中“转移A”的描述数据修改一下，使其描述从正常投票状态转换成恶意投票状态就可以了。

理解状态模式

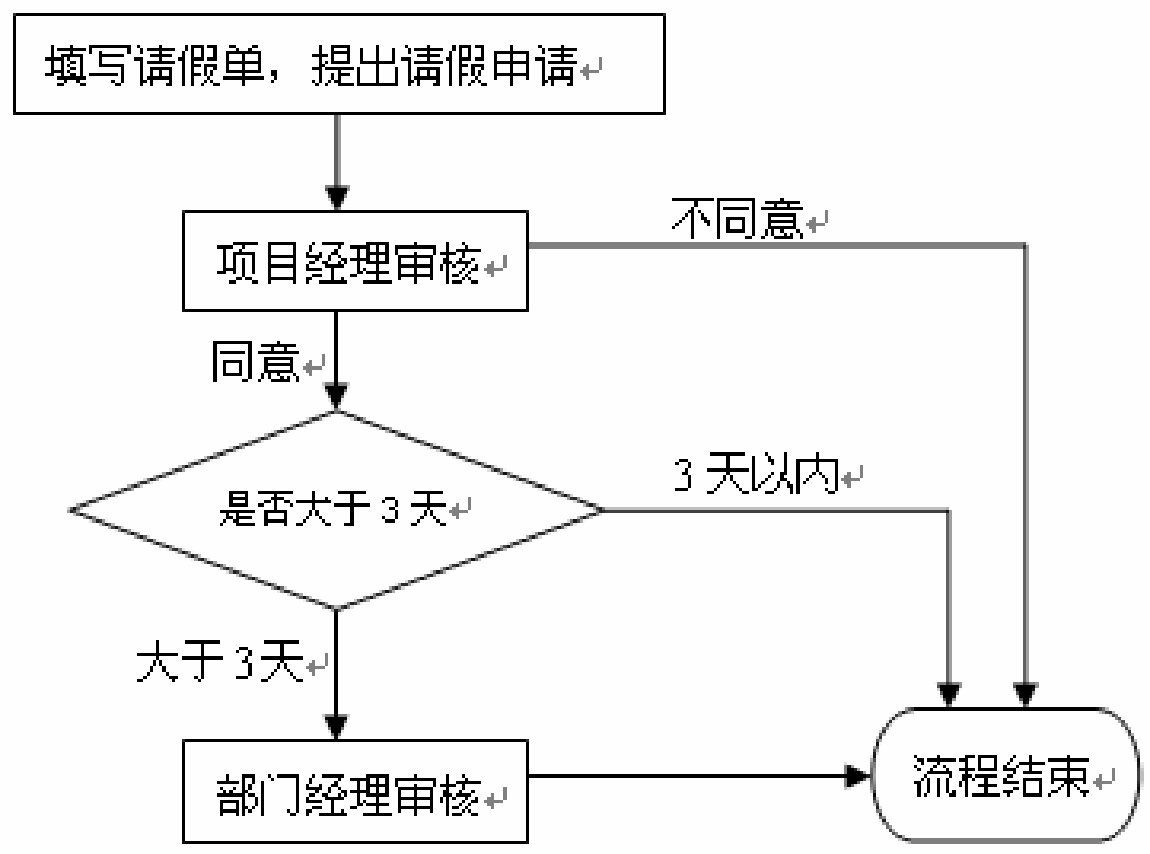
- 模拟工作流

做企业应用的朋友，大多数都接触过工作流，至少处理过业务流程。当然对于工作流，复杂的应用可能会使用工作流中间件，用工作流引擎来负责流程处理，这个会比较复杂，其实工作流引擎的实现也可以应用上状态模式，这里不去讨论。

简单点的，把流程数据存放在数据库里面，然后在程序里面自己来进行流程控制。对于简单点的业务流程控制，可以使用状态模式来辅助进行流程控制，因为大部分这种流程都是状态驱动的。

举个例子来说明吧，举个最常见的“请假流程”，流程是这样的：当某人提出请假申请过后，先由项目经理来审批，如果项目经理不同意，审批就直接结束；如果项目经理同意了，再看请假的天数是否超过3天，项目经理的审批权限只有3天以内，如果请假天数在3天以内，那么审批也直接结束，否则就提交给部门经理；部门经理审核过后，无论是否同意，审批都直接结束。流程图如图

理解状态模式



理解状态模式

- 实现思路

仔细分析上面的流程图和运行过程，把请假单在流程中的各个阶段的状态分析出来，会发现，整个流程完全可以看成是状态驱动的。

在上面的流程中，请假单大致有如下状态：等待项目经理审核、等待部门经理审核、审核结束。

既然可以把流程看成是状态驱动的，那么就可以自然的使用上状态模式，每次当相应的工作人员完成工作，请求流程响应的时候，流程处理的对象会根据当前所处的状态，把流程处理委托给相应的状态对象去处理。

又考虑到在一个系统中会有很多流程，虽然不像通用工作流那么复杂的设计，但还是稍稍提炼一下，至少把各个不同的业务流程，在应用状态模式时的公共功能，或者是架子给搭出来，以便复用这些功能。

理解状态模式

- 状态模式的优缺点
  1：简化应用逻辑控制
  2：更好的分离状态和行为
  3：更好的扩展性
  4：显式化进行状态转换
  5：引入太多的状态类

思考状态模式

- 状态模式的本质

  状态模式的本质是：**根据状态来分离和选择行为**

- 何时选用状态模式

1：如果一个对象的行为取决于它的状态，而且它必须在运行时刻根据状态来改变它的行为。可以使用状态模式，来把状态和行为分离开，虽然分离开了，但状态和行为是有对应关系的，可以在运行期间，通过改变状态，就能够调用到该状态对应的状态处理对象上去，从而改变对象的行为

2：如果一个操作中含有庞大的多分支语句，而且这些分支依赖于该对象的状态。可以使用状态模式，把各个分支的处理分散包装到单独的对象处理类里面，这样，这些分支对应的对象就可以不依赖于其它对象而独立变化了