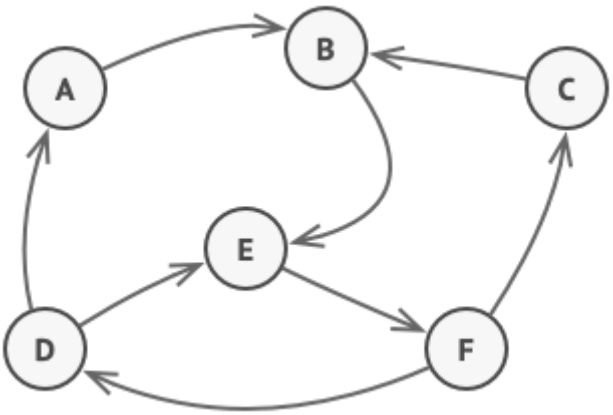
有限状态机FSM的简介与Demo

原文链接

Thursday, December 9, 2021

1. 概述

首先看一段维基百科中状态机的介绍：**有限状态机**（英语：finite-state machine，缩写：**FSM**）又称**有限状态自动机**（英语：finite-state automaton，缩写：**FSA**），简称**状态机**，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学计算模型，如下图所示：



本文所有的代码都托管在Github。

1.1 特性

有限状态机（Finite-state machine）是一个非常有用的模型，可以模拟世界上大部分事物，简单说，它有三个特征：

1. 状态总数（state）是有限的。
2. 任一时刻，只处在一种状态之中。
3. 某种条件下，会从一种状态转变（transition）到另一种状态。

1.2 核心概念

状态机有四个核心概念：

1. 状态 State：一个状态机至少要包含两个状态。
2. 事件 Event：事件就是执行某个操作的触发条件或者口令。
3. 动作 Action：事件发生以后要执行动作。
4. 变换 Transition：也就是从一个状态变化为另一个状态。

1.3 设计模式

状态模式(State Pattern)是一种行为设计模式，和状态机的概念紧密相关，其主要思想是程序在任意时刻仅可处于几种**有限**的状态中。 在任何一个特定状态中， 程序的行为都不相同， 且可瞬间从一个状态切换到另一个状态。 不过， 根据当前状态， 程序可能会切换到另

外一种状态， 也可能会保持当前状态不变。 这些数量有限且预先定义的状态切换规则被称为**转移**。

1.4 为什么要用状态机?

假设我们当前使用 `if` 或者 `switch` 维护着订单的状态流转，根据用户提交的操作执行不同的逻辑，假设当前只有 下单、支付、退款等三种操作，对应订单的状态只有 待支付、已支付、已退款 三个状态，我们可以使用 `switch` 进行管理，如下所示：

```
switch cmd {
  case "submit":
  fmt.Println("用户提交了订单，当前等待用户支付")
  fmt.Println("通知仓储系统，冻结部分库存")
  case "pay":
  fmt.Println("用户支付了订单，等待仓储系统发货")
  fmt.Println("通知仓储系统，扣减部分库存")
  fmt.Println("通知统计系统，进行订单统计")
  case "refund":
  fmt.Println("用户申请了退款")
  fmt.Println("通知仓储系统，回滚部分库存")
  fmt.Println("通知统计系统，回滚订单统计")
}
```

当订单状态和业务系统足够简单的时候，我们使用上述的 `switch` 代码也可以勉强维护，但实际中，一个订单系统不可能如此简单，并且仓储、统计、支付等等系统都是完全独立的，一些事件的通知很多都是复用的。

实际开发中，一个订单的状态流转往往很复杂，表现在：

1. 订单状态多
2. 一个订单往往包含多个子订单，并且子订单的状态流转也是独立的
3. 状态流转的操作很多都是公共的，比如通知类操作
4. 订单操作的执行都需要检测现有订单状态，特定状态下往往只允许特定操作执行
5. 随着业务的发展，会时不时的多几个状态进来

这个时候回过头看 `switch` 代码，就会发现用 `switch` 管理状态，就会出现代码耦合严重、逻辑繁琐、`switch` 分支巨多并且重复代码非常多，而且随着状态的增加，对原有代码逻辑的修改就会变的非常困难，这个时候就需要有限状态机来处理了。

2. 订单系统分析

实际的订单很多都是一个主订单包含多个子订单，在支付之前这些订单的状态都会统一的，支付之后这些订单的状态就开始独立流转，比如订单的发货、售后、退款等。

这里为了简化说明，只允许子订单整单售后，不允许部分售后，且只能在待发货和已签收时进行售后，对应的售后类型如下：

1. 待发货状态：退款
2. 已签收状态：退货退款

2.1 主订单

由于发货、售后等是子订单的操作，因此主订单的状态只到已支付状态即完成。主订单包含的状态和允许的操作如下：

状态	编码	允许操作及目标状态
待支付	0	支付：待确认；取消：已取消；支付确认：已支付
已取消	1	无
待确认	2	支付确认：已支付
已支付	3	无

2.2 子订单

子订单包含的状态和操作如下：

状态	编码	允许操作
待支付	0	支付：待确认；取消：已取消；支付确认：已支付
已取消	1	无
待确认	2	支付确认：待发货
待发货	3	发货：待收货；申请退款：售后中-退款
售后中-退款	4	取消售后：待发货；退款完成：已完成
待收货	5	签收：已签收
已签收	6	申请退货退款：售后中-退货退款；订单完成：已完成
售后中-退货退款	7	取消售后：已签收；退款完成：已完成
已完成	8	无

2.3 售后

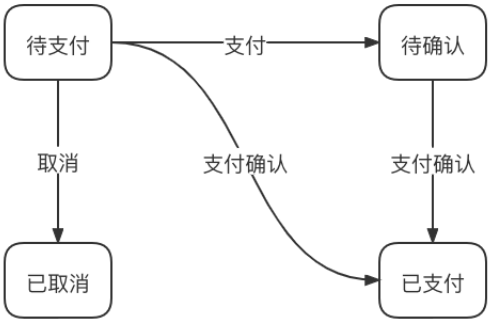
售后的状态和操作如下：

状态	编码	允许操作
待审批	0	通过：已通过；驳回：已驳回；取消：已取消
已取消	1	无
已驳回	2	无
已通过	3	提交退款申请(未发货订单)：退款中；等待用户寄回(已发货订单)：退货中；取消：已取消
退货中	4	发货：待收货；取消：已取消
待收货	5	签收：退款中
退款中	6	退款完成：已完成
已完成	7	无

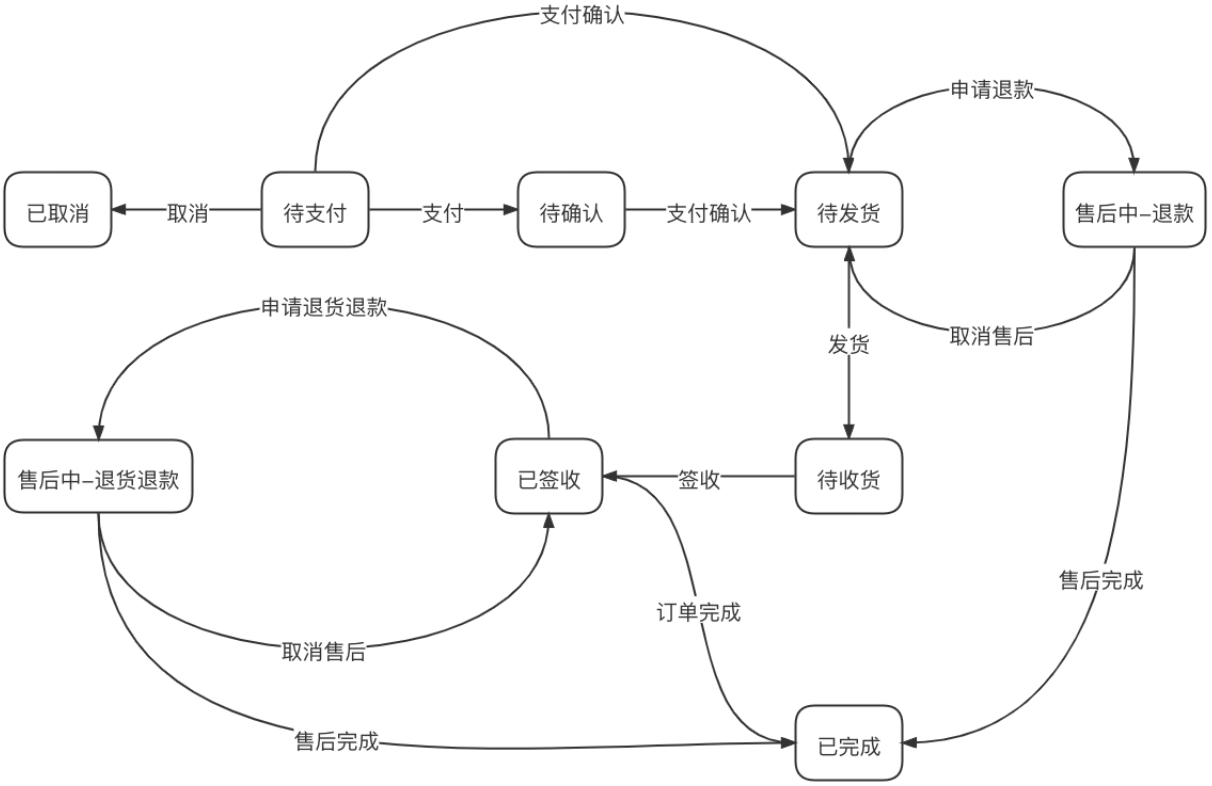
2.4 流转图

三个状态机的状态流转图如下所示，这是很理想化很简化的流转图，这篇文章主要说明流转的过程，省略了很多的细节，实际情况要复杂的多。

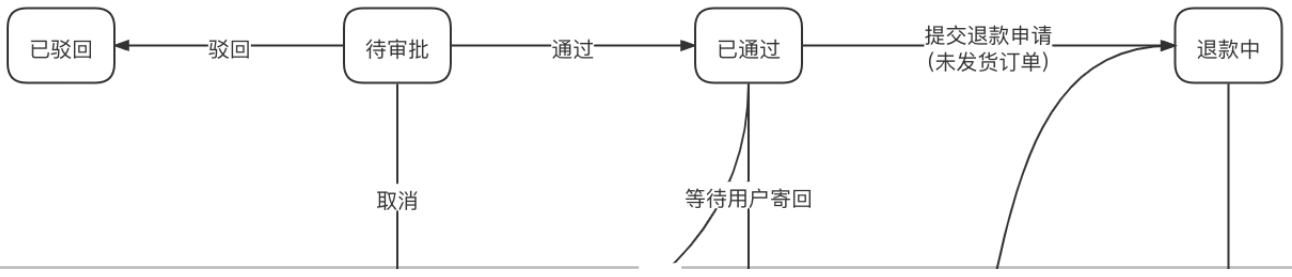
主订单状态流转图

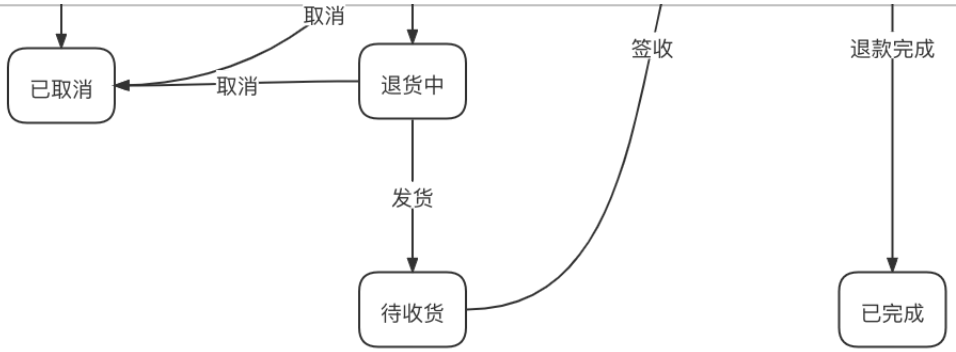


子订单状态流转图



售后状态流转图





4. Demo 实现

这篇文章的主要目的是说明状态机的状态流转，所以所有的操作都只通过打印日志来完成，并且在打印前等待100豪秒，这样所有日志的打印时间就可以区别开来。

4.1 状态机

在状态机的实现中，先定义上文所说的四个概念：

1. 状态 State
2. 事件 Event
3. 动作 Action
4. 变换 Transition

接着定义了一个监听处理器接口**EventProcessor**，这个接口包含两个方法：

1. ExitOldState 离开旧状态：状态离开旧状态之前执行
2. EnterNewState 进入新状态；状态进入新状态之后执行

每个状态机都需要定义一个默认的处理器 Processor，并且每个转变器 Transition 也可以自定义自己的处理器，注意，状态机和转变器的 处理器不是覆盖关系，而是先后执行的关系。

接着我们定义了状态机图表**StateGraph**，图表是一个每一个状态机数据的定义集合，包含状态机的名称、状态列表、事件列表、状态及事件之间转换器的关联列表。

最后定义状态机**StateMachine**，并且为状态机内部的图表定义了一些设置属性的方法，最后为状态机定义了执行的方法**Run**，这也是状态机执行状态流转的核心方法，它的流程如下：

1. 状态及事件检测
2. 执行状态机的处理器的 ExitOldState 方法
3. 检查转变器是否定义了处理器，如果定义了，执行该处理器的 ExitOldState 方法
4. 执行转变器定义的 Action
5. 执行状态机的处理器的 EnterNewState 方法
6. 检查转变器是否定义了处理器，如果定义了，执行该处理器的 EnterNewState 方法
7. 执行完毕

定义好状态机之后，主订单、子订单、售后 只需要按需配置以下内容即可：

1. 定义状态列表
2. 定义事件列表
3. 定义转换器列表
4. 定义动作方法列表
5. 定义处理器列表

6. 对外保留一个生成状态机的 `NewStateMachine` 方法

完整的代码示例如下：

```
package fsm_order_demo

import (
    "fmt"
    "sync"

    "github.com/jormin/fsm-order-demo/helper"
)

// State 状态
type State uint8

// Event 事件
type Event string

// Action 动作
type Action func(from State, event Event, to State) error

// Transition 转换器
type Transition struct {
    From      State          `desc:"旧状态"`
    Event     Event          `desc:"事件"`
    To        State          `desc:"新状态"`
    Action    Action         `desc:"动作"`
    Processor EventProcessor `desc:"处理器"`
}

// StateGraph 状态机图表
type StateGraph struct {
    name        string                          `desc:"图表名称"`
    start       State                           `desc:"起始状态"`
    end         State                           `desc:"结束状态"`
    states      map[State]string                `desc:"状态集合"`
    transitions map[State]map[Event]Transition `desc:"转变器集合"`
}

// EventProcessor 事件处理器
type EventProcessor interface {
    // ExitOldState 离开旧状态
    ExitOldState(state State, event Event) error
    // EnterNewState 进入新状态
    EnterNewState(state State, event Event) error
}

// StateMachine 状态机
type StateMachine struct {
    locker    sync.Mutex     `desc:"排它锁"`
    Processor EventProcessor `desc:"事件处理器"`
    Graph     *StateGraph    `desc:"状态图表"`
}
```

```go
// SetName 设置状态图表名称
func (s *StateMachine) SetName(name string) {
    s.Graph.name = name
}

// SetStart 设置状态图表起始状态
func (s *StateMachine) SetStart(start State) {
    s.Graph.start = start
}

// SetEnd 设置状态图表最终状态
func (s *StateMachine) SetEnd(end State) {
    s.Graph.end = end
}

// SetStates 设置状态图表状态列表
func (s *StateMachine) SetStates(states map[State]string) {
    s.Graph.states = states
}

// SetTransitions 设置状态图表转变器列表
func (s *StateMachine) SetTransitions(transitions map[State]map[Event]Transi
tion) {
    s.Graph.transitions = transitions
}

// GetStateDesc 获取状态描述
func (s *StateMachine) GetStateDesc(state State) string {
    return fmt.Sprintf("%s(%d)", s.Graph.states[state], state)
}

// Run 执行
func (s *StateMachine) Run(from State, event Event) (State, error) {
    helper.Log("开始执行，旧状态为 %d，事件为 %s", from, event)
    // 检测状态是否存在
    if _, ok := s.Graph.states[from]; !ok {
        return 0, helper.ErrOldStateNotExists
    }
    // 检测到状态是否已到最终状态
    if from == s.Graph.end {
        return 0, helper.ErrOldStateIsEndState
    }
    // 检测状态和事件是否匹配
    transition, ok := s.Graph.transitions[from][event]
    if !ok {
        return 0, helper.ErrOldStateDontHaveTheEventTransition
    }
    // 新状态
    to := transition.To
    // 加锁
    s.locker.Lock()
    // 执行完毕时解锁
    defer s.locker.Unlock()
    // 执行状态机处理器的退出旧状态方法
    _ = s.Processor.ExitOldState(from, event)
```

```
    // 如果当前转变期设置了处理器，则执行该处理器的退出旧状态方法
    if transition.Processor != nil {
        _ = transition.Processor.ExitOldState(from, event)
    }
    // 执行转变期的动作
    _ = transition.Action(from, event, transition.To)
    // 执行状态机处理器的进入新状态方法
    _ = s.Processor.EnterNewState(to, event)
    // 如果当前转变期设置了处理器，则执行该处理器的进入新状态方法
    if transition.Processor != nil {
        _ = transition.Processor.EnterNewState(to, event)
    }
    return to, nil
}
```

4.2 主订单

状态

```
const (
    // StateWaitPay 待支付
    StateWaitPay = iota
    // StateCancel 已取消
    StateCancel
    // StateWaitConfirm 待确认
    StateWaitConfirm
    // StatePayied 已支付
    StatePayied
)
```

事件

```
const (
    // EventPay 支付
    EventPay = "pay"
    // EventPayConfirm 支付确认
    EventPayConfirm = "pay_confirm"
    // EventCancel 取消
    EventCancel = "cancel"
)
```

转变器

```
// transitions 转变器
var transitions = map[fsmorderdemo.State]map[fsmorderdemo.Event]fsmorderdem
o.Transition{
    StateWaitPay: {
        // 取消：待支付 ---> 已取消
        EventCancel: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventCancel, To: StateCancel, Action:
```

```
 action.Cancel, Processor: nil,
        },
        // 支付: 待支付 ---> 待确认
        EventPay: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventPay, To: StateWaitConfirm, Actio
n: action.Pay,
            Processor: &processor.PayEventProcessor{},
        },
        // 支付确认: 待支付 ---> 已支付
        EventPayConfirm: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventPayConfirm, To: StatePayied, Acti
on: action.PayConfirm, Processor: nil,
        },
    },
    StateWaitConfirm: {
        // 支付确认: 待确认 ---> 已支付
        EventPayConfirm: fsmorderdemo.Transition{
            From: StateWaitConfirm, Event: EventPayConfirm, To: StatePayied,
 Action: action.PayConfirm, Processor: nil,
        },
    },
}
```

事件对应的状态流转

```
// 取消: 待支付 ---> 已取消
// 支付: 待支付 ---> 待确认
// 支付确认: 待支付 ---> 已支付
// 支付确认: 待确认 ---> 已支付
```

4.3 子订单

状态

```
const (
    // StateWaitPay 待支付
    StateWaitPay = iota
    // StateCancel 已取消
    StateCancel
    // StateWaitConfirm 待确认
    StateWaitConfirm
    // StateWaitDelive 待发货
    StateWaitDelive
    // StateRefund 售后中-退款
    StateRefund
    // StateWaitReceive 待收货
    StateWaitReceive
    // StateSigned 已签收
    StateSigned
    // StateGoodsRefund 售后中-退货退款
    StateGoodsRefund
    // StateCompleted 已完成
```

```
    StateCompleted
)
```

事件

```
const (
    // EventPay 支付
    EventPay = "pay"
    // EventCancel 取消
    EventCancel = "cancel"
    // EventPayConfirm 支付确认
    EventPayConfirm = "pay_confirm"
    // EventDelive 发货
    EventDelive = "delive"
    // EventApplyRefund 申请退款
    EventApplyRefund = "apply_refund"
    // EventCancelRefund 取消售后
    EventCancelRefund = "cancel_refund"
    // EventRefundCompleted 退款完成
    EventRefundCompleted = "refund_completed"
    // EventSigned 签收
    EventSigned = "signed"
    // EventApplyGoodsRefund 申请退货退款
    EventApplyGoodsRefund = "apply_goods_refund"
    // EventCompleted 订单完成
    EventCompleted = "completed"
)
```

转变器

```
// transitions 转变器
var transitions = map[fsmorderdemo.State]map[fsmorderdemo.Event]fsmorderdem
o.Transition{
    StateWaitPay: {
        // 取消：待支付 ---> 已取消
        EventCancel: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventCancel, To: StateCancel, Action:
 action.Cancel, Processor: nil,
        },
        // 支付：待支付 ---> 待确认
        EventPay: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventPay, To: StateWaitConfirm, Actio
n: action.Pay,
            Processor: &processor.PayEventProcessor{},
        },
        // 支付确认：待支付 ---> 待发货
        EventPayConfirm: fsmorderdemo.Transition{
            From: StateWaitPay, Event: EventPayConfirm, To: StateWaitDelive,
 Action: action.PayConfirm, Processor: nil,
        },
    },
    StateWaitConfirm: {
```

```
        // 支付确认：待确认 ---> 待发货
        EventPayConfirm: fsmorderdemo.Transition{
            From: StateWaitConfirm, Event: EventPayConfirm, To: StateWaitDeli
ve, Action: action.PayConfirm,
            Processor: nil,
        },
    },
    StateWaitDelive: {
        // 发货：待发货 ---> 待收货
        EventDelive: fsmorderdemo.Transition{
            From: StateWaitDelive, Event: EventDelive, To: StateWaitReceive,
 Action: action.Delive, Processor: nil,
        },
        // 申请退款：待发货 ---> 售后中-退款
        EventApplyRefund: fsmorderdemo.Transition{
            From: StateWaitDelive, Event: EventApplyRefund, To: StateRefund,
 Action: action.ApplyRefund, Processor: nil,
        },
    },
    StateRefund: {
        // 取消售后：售后中-退款 ---> 待发货
        EventCancelRefund: fsmorderdemo.Transition{
            From: StateRefund, Event: EventCancelRefund, To: StateWaitDelive,
Action: action.CancelRefund,
            Processor: nil,
        },
        // 退款完成：售后中-退款 ---> 已完成
        EventRefundCompleted: fsmorderdemo.Transition{
            From: StateRefund, Event: EventRefundCompleted, To: StateComplete
d, Action: action.RefundCompleted,
            Processor: nil,
        },
    },
    StateWaitReceive: {
        // 签收：待收货 ---> 已签收
        EventSigned: fsmorderdemo.Transition{
            From: StateWaitReceive, Event: EventSigned, To: StateSigned, Acti
on: action.Signed, Processor: nil,
        },
    },
    StateSigned: {
        // 申请退货退款：已签收 ---> 售后中-退货退款
        EventApplyGoodsRefund: fsmorderdemo.Transition{
            From: StateSigned, Event: EventApplyGoodsRefund, To: StateGoodsRe
fund, Action: action.ApplyGoodsRefund,
            Processor: nil,
        },
        // 订单完成：已签收 ---> 已完成
        EventCompleted: fsmorderdemo.Transition{
            From: StateSigned, Event: EventCompleted, To: StateCompleted, Act
ion: action.Completed, Processor: nil,
        },
    },
    StateGoodsRefund: {
        // 取消售后：售后中-退货退款 ---> 已签收
        EventCancelRefund: fsmorderdemo.Transition{
            From: StateGoodsRefund, Event: EventCancelRefund, To: StateSigne
```

```
d, Action: action.CancelRefund,
            Processor: nil,
        },
        // 退款完成：售后中-退货退款 ---> 已完成
        EventRefundCompleted: fsmorderdemo.Transition{
            From: StateGoodsRefund, Event: EventRefundCompleted, To: StateCom
pleted, Action: action.RefundCompleted,
            Processor: nil,
        },
    },
}
```

事件对应的状态流转

```
// 取消：待支付 ---> 已取消
// 支付：待支付 ---> 待确认
// 支付确认：待支付 ---> 待发货
// 支付确认：待确认 ---> 待发货
// 发货：待发货 ---> 待收货
// 申请退款：待发货 ---> 售后中-退款
// 取消售后：售后中-退款 ---> 待发货
// 退款完成：售后中-退款 ---> 已完成
// 签收：待收货 ---> 已签收
// 申请退货退款：已签收 ---> 售后中-退货退款
// 订单完成：已签收 ---> 已完成
// 取消售后：售后中-退货退款 ---> 已签收
// 退款完成：售后中-退货退款 ---> 已完成
```

4.4 售后

状态

```
const (
    // StateWaitApprove 待审批
    StateWaitApprove = iota
    // StateCancel 已取消
    StateCancel
    // StateRefused 已驳回
    StateRefused
    // StateAgreed 已通过
    StateAgreed
    // StateWaitDelive 退货中
    StateWaitDelive
    // StateWaitReceive 待收货
    StateWaitReceive
    // StateWaitRefund 退款中
    StateWaitRefund
    // StateCompleted 已完成
    StateCompleted
)
```

事件

```
const (
    // EventCancel 取消
    EventCancel = "cancel"
    // EventRefuse 驳回
    EventRefuse = "refuse"
    // EventAgree 通过
    EventAgree = "agree"
    // EventRefund 提交退款申请(未发货订单)
    EventRefund = "refund"
    // EventGoodsRefund 等待用户寄回(已发货订单)
    EventGoodsRefund = "goods_refund"
    // EventDelive 发货
    EventDelive = "delive"
    // EventSigned 签收
    EventSigned = "signed"
    // EventRefundCompleted 退款完成
    EventRefundCompleted = "refund_completed"
)
```

转变器

```
var transitions = map[fsmorderdemo.State]map[fsmorderdemo.Event]fsmorderdem
o.Transition{
    StateWaitApprove: {
        // 取消: 待审批 ---> 已取消
        EventCancel: fsmorderdemo.Transition{
            From: StateWaitApprove, Event: EventCancel, To: StateCancel, Acti
on: action.Cancel, Processor: nil,
        },
        // 驳回: 待审批 ---> 已驳回
        EventRefuse: fsmorderdemo.Transition{
            From: StateWaitApprove, Event: EventRefuse, To: StateRefused, Act
ion: action.Refuse, Processor: nil,
        },
        // 通过: 待审批 ---> 已通过
        EventAgree: fsmorderdemo.Transition{
            From: StateWaitApprove, Event: EventAgree, To: StateAgreed, Actio
n: action.Agree, Processor: nil,
        },
    },
    StateAgreed: {
        // 提交退款申请(未发货订单): 已通过 ---> 退款中
        EventRefund: fsmorderdemo.Transition{
            From: StateAgreed, Event: EventRefund, To: StateWaitRefund, Actio
n: action.Refund, Processor: nil,
        },
        // 等待用户寄回(已发货订单): 已通过 ---> 退货中
        EventGoodsRefund: fsmorderdemo.Transition{
            From: StateAgreed, Event: EventGoodsRefund, To: StateWaitDelive,
 Action: action.GoodsRefund, Processor: nil,
        },
```

```
        // 取消：已通过 ---> 已取消
        EventCancel: fsmorderdemo.Transition{
            From: StateAgreed, Event: EventCancel, To: StateCancel, Action: a
ction.Cancel, Processor: nil,
        },
    },
    StateWaitDelive: {
        // 发货：退货中 ---> 待收货
        EventDelive: fsmorderdemo.Transition{
            From: StateWaitDelive, Event: EventDelive, To: StateWaitReceive,
 Action: action.Delive,
            Processor: &processor.DeliveEventProcessor{},
        },
        // 取消：退货中 ---> 已取消
        EventCancel: fsmorderdemo.Transition{
            From: StateWaitDelive, Event: EventCancel, To: StateCancel, Actio
n: action.Cancel, Processor: nil,
        },
    },
    StateWaitReceive: {
        // 签收：待收货 ---> 退款中
        EventSigned: fsmorderdemo.Transition{
            From: StateWaitReceive, Event: EventSigned, To: StateWaitRefund,
 Action: action.Signed, Processor: nil,
        },
    },
    StateWaitRefund: {
        // 退款完成：退款中 ---> 已完成
        EventRefundCompleted: fsmorderdemo.Transition{
            From: StateWaitRefund, Event: EventRefundCompleted, To: StateComp
leted, Action: action.RefundCompleted,
            Processor: nil,
        },
    },
}
```

事件对应的状态流转

```
// 取消：待审批 ---> 已取消
// 驳回：待审批 ---> 已驳回
// 通过：待审批 ---> 已通过
// 提交退款申请(未发货订单)：已通过 ---> 退款中
// 等待用户寄回(已发货订单)：已通过 ---> 退货中
// 取消：已通过 ---> 已取消
// 发货：退货中 ---> 待收货
// 取消：退货中 ---> 已取消
// 签收：待收货 ---> 退款中
// 退款完成：退款中 ---> 已完成
```

5. 测试

状态机的测试涵盖了上一部分中列出的每个状态机所有的时间状态流转场景，以及三个错误场景，具体的测试用例可在代码中查看，这里不再列出，错误场景如下：

1. ErrOldStateNotExists：旧状态不存在，无法进行流转
2. ErrOldStateIsEndState：旧状态已是最终状态，无法进行流转
3. ErrOldStateDontHaveTheEventTransition：旧状态没有指定事件的的转变器，无法进行流转

6. 总结

这篇文章简单介绍了状态机的概念，并通过一个简化的订单流程来实际分析并实践，最后通过分析的结果写了个 Demo 来验证。