13.图解支付系统简洁而精妙的状态机设计与实现

1. 前言

2. 什么是状态机

3. 状态机对支付系统的重要性

4. 状态机设计基本原则

5. 状态机常见设计误区

6. 状态机设计的最佳实践

7. 常见代码实现误区

8. Spring Statemachine简要介绍

9. 用JAVA实现一个简洁的交易状态机

10. 并发更新问题

11. 结束语

大家好，我是隐墨星辰，深耕境内/跨境支付架构设计十余年。

我毕业很多年后才开始写状态机的代码。

进入支付行业第3年左右，因为状态判断不到位，出现过用户扣款【成功】，但订单状态都是【支付中】。原因是在并发场景下，外部渠道的异步通知【成功】先到，同步请求接口返回【支付中】后到，前者修改数据库状态为【成功】，后者又把数据库状态又修改为【支付中】。

多年后，看到别人使用状态机来推进交易订单状态，恍然大悟，原来还可以这样设计！

假如你做的是支付、电商等这类交易系统，但没有听过状态机，或者你听过但没有写过，又或者你是使用if else 或switch case来写交易订单的状态推进，建议花点时间看看，一定会有不一样的收获。如果你是产品经理，可以考虑推荐给你们的研发同学，说不定能提升系统的健壮性。

这篇文章主要讲清楚：*清楚什么是状态机，简洁的状态机对交易系统的重要性，状态机设计常见误区，以及如何设计出简洁而精妙的状态机，核心的状态机代码实现。*

1. 前言

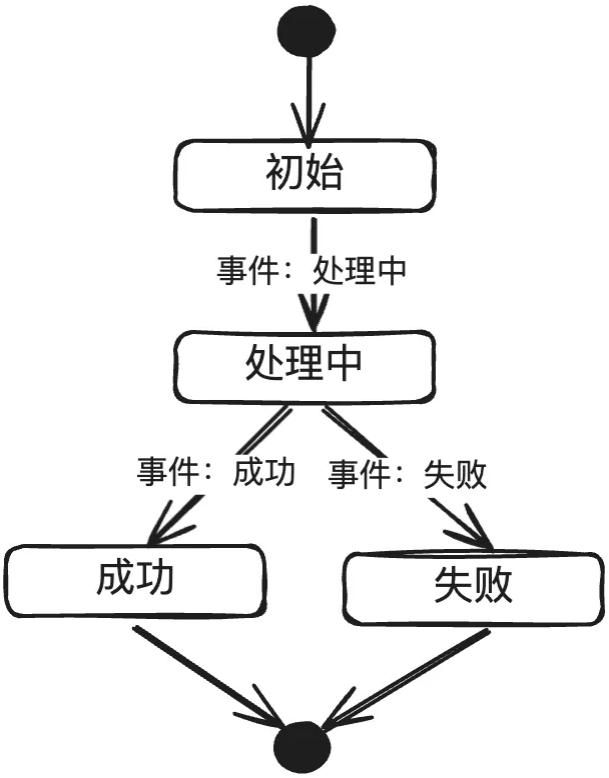
在交易类系统中，比如支付系统或电商系统，交易单据的状态管理至关重要。一个合理的状态机设计可以帮助我们清晰地掌握交易单据的状态变化，提高系统的健壮性和可维护性。

本文将一步步介绍状态机的概念、其在支付系统中的重要性、设计原则、常见误区、最佳实践，以及一个实际的Java代码实现。

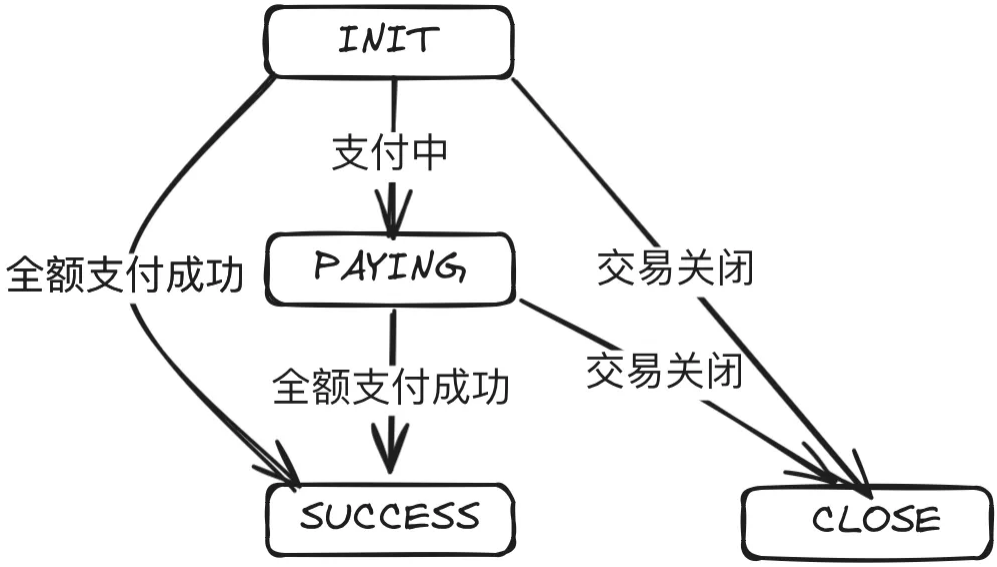
2. 什么是状态机

状态机，也称为有限状态机（FSM, Finite State Machine），是一种行为模型，由一组定义良好的状态、状态之间的转换规则和一个初始状态组成。它根据当前的状态和输入的事件，从一个状态转移到另一个状态。

下图是典型的状态机设计：



下图就是在《支付交易的三重奏：收单、结算与拒付在支付系统中的协奏曲》中提到的交易单的状态机。



从图中可以看到，一共4个状态，每个状态之间的转换由指定的事件触发。假如当前是“INIT”，当有“支付中”的事件发生，就会推进到“PAYING”状态。

3. 状态机对支付系统的重要性

想像一下，如果没有状态机，支付系统如何知道订单已经支付成功了呢？如果你的订单已经被一个线程更新为“成功”，另一个线程又更新成“失败”，你会不会跳起来？

在支付系统中，状态机管理着每笔交易的生命周期，从初始化到成功或失败。它确保交易在正确的时间点，以正确的顺序流转到正确的状态。这不仅提高了交易处理的效率和一致性，还增强了系统的健壮性，使其能够有效处理异常和错误，确保支付流程的顺畅。

4. 状态机设计基本原则

无论是设计支付类的系统，还是电商类的系统，在设计状态机时，都建议遵循以下原则：

1. **明确性：**状态和转换必须清晰定义，避免含糊不清的状态。
2. **完备性：**为所有可能的事件–状态组合定义转换逻辑。
3. **可预测性：**系统应根据当前状态和给定事件可预测地响应。

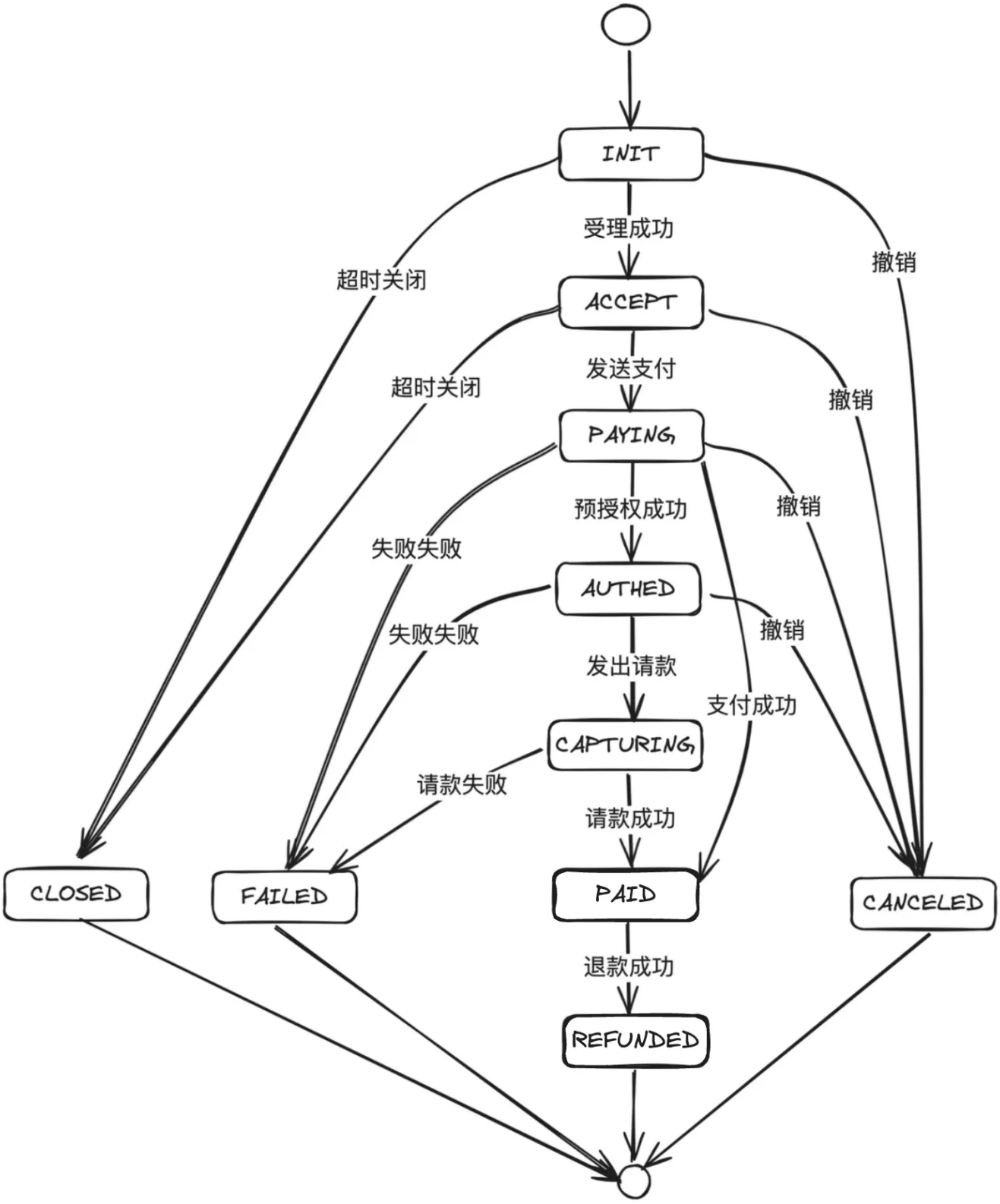
4. **最小化：**状态数应保持最小，避免不必要的复杂性。
5. **可扩展性**：状态机应该具有良好的可扩展性，以便在业务需求变化时能够方便地添加新的状态和转换规则

5. 状态机常见设计误区

工作多年，见过很多设计得不好的状态机，导致运维特别麻烦，还容易出故障，总结出来一共有这么几条：

1. **过度设计：**引入不必要的状态和复杂性，使系统难以理解和维护。
2. **不完备的处理：**未能处理所有可能的状态转换，导致系统行为不确定。
3. **硬编码逻辑：**过多的硬编码转换逻辑，使系统不具备灵活性和可扩展性。

举一个例子感受一下。下面是亲眼见过的一个交易单的状态机设计，而且一眼看过去，好像除了复杂一点，整体还是合理的，比如初始化，受理成功就到ACCEPT，然后到PAYING，如果直接成功就到PAID，退款成功就到REFUNDED。



这个状态机有几个明显不合理的地方：

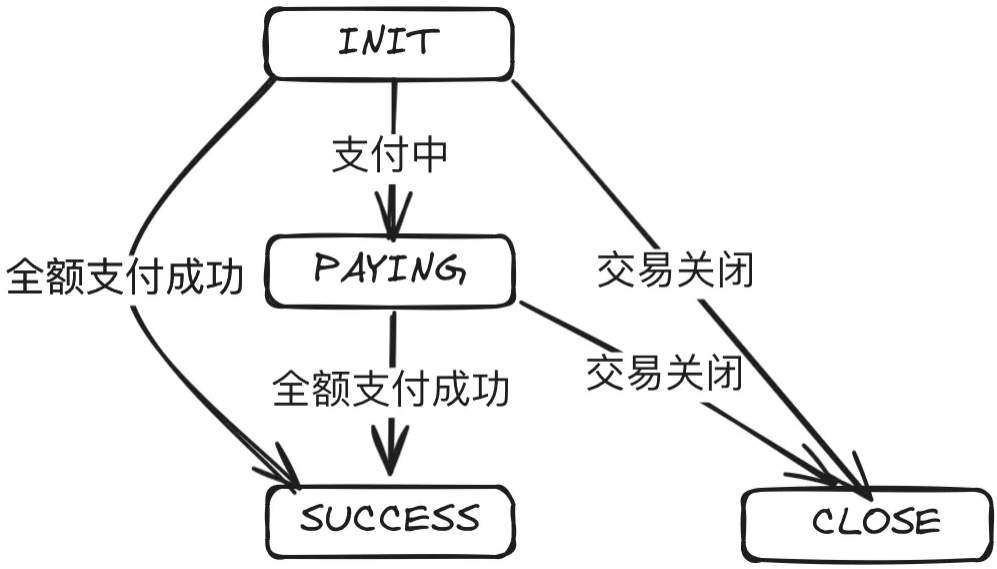
1. **过于复杂**。一些不必要的状态可以去掉，比如ACCEPT没有存在的必要，因为INIT说明已经受理入库。

2. **职责不明确**。支付单就只管支付，到PAID就支付成功，就是终态不再改变。REFUNDED应该由退款单来负责处理，否则部分退款怎么办。

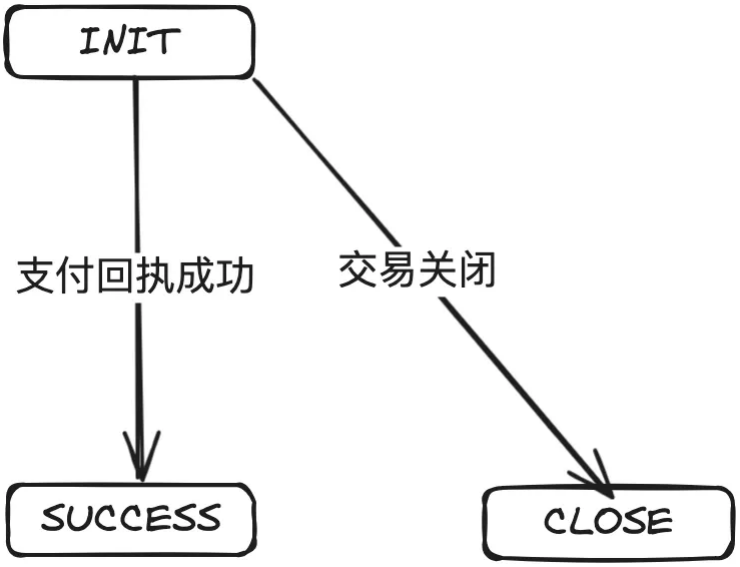
我们需要的改造方案：

1. **精简掉不必要的状态**，比如ACCEPT。
2. **把一些退款、请款等单据单独抽出去**，通过独立的退款单、请款单来独立管理，这样单据类型虽然多了，但是架构更加清晰合理。

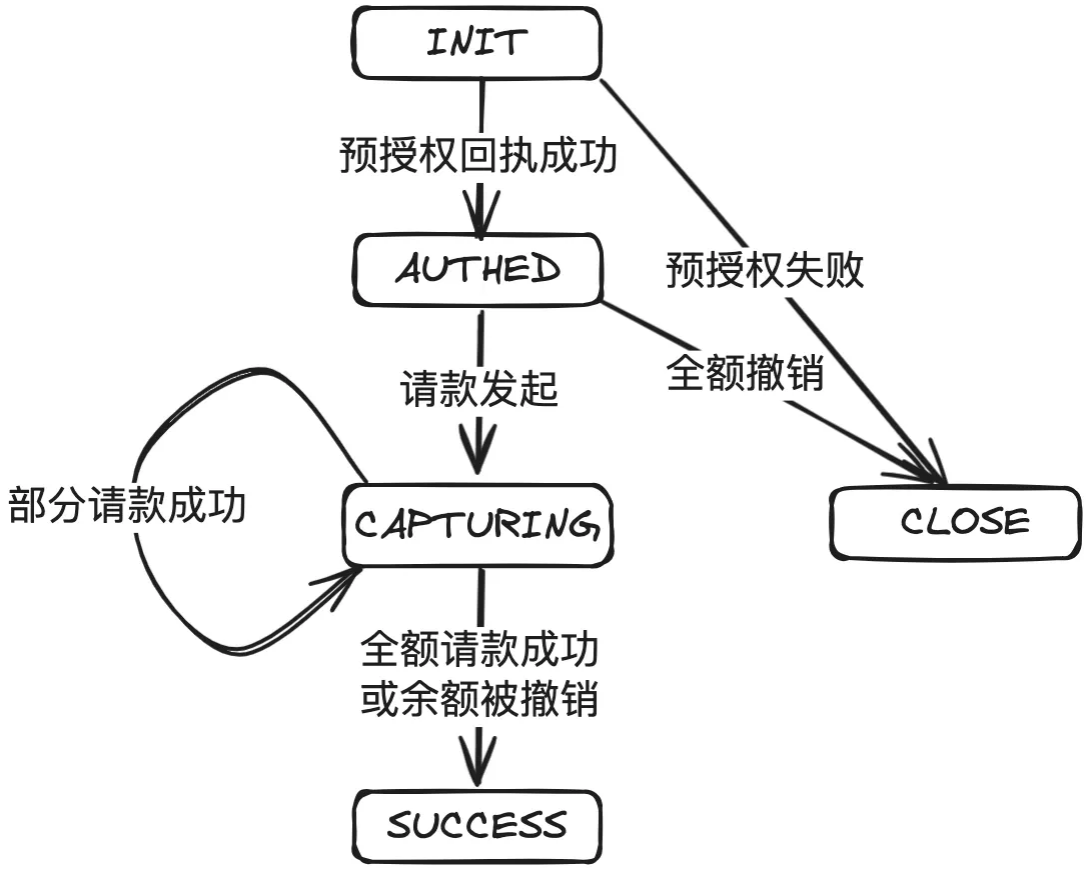
主单：



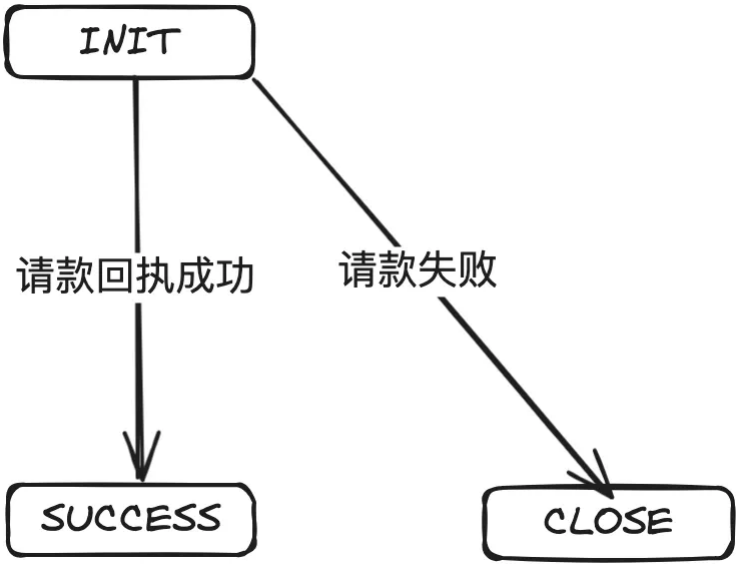
普通支付单：


INIT
支付回执成功
交易关闭
SUCCESS
CLOSE

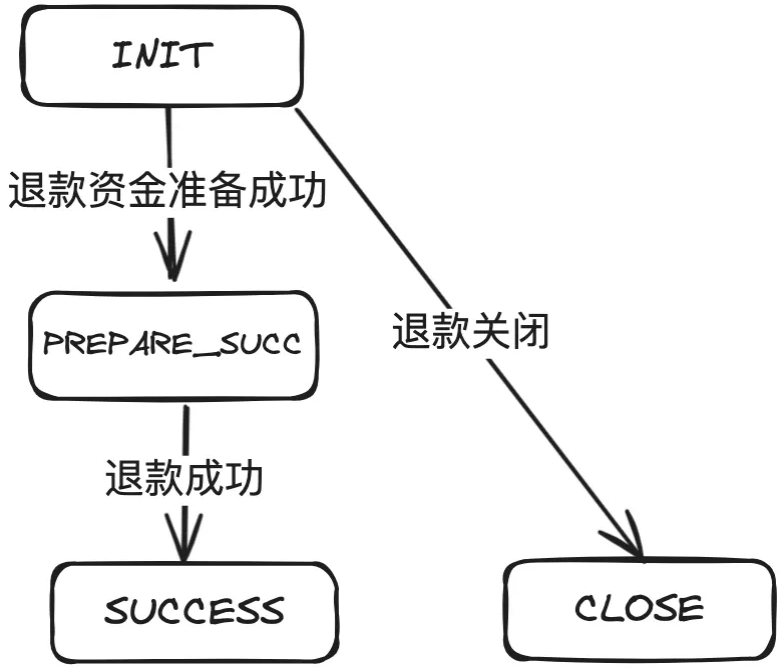
预授权单：


INIT
预授权回执成功
AUTHED
预授权失败
全额撤销
请款发起
部分请款成功
CAPTURING
CLOSE
全额请款成功
或余额被撤销
SUCCESS

请款单：



退款单：



6. 状态机设计的最佳实践

在代码实现层面，需要做到以下几点：

1. **分离状态和处理逻辑**：使用状态模式，将每个状态的行为封装在各自的类中。
2. **使用事件驱动模型**：通过事件来触发状态转换，而不是直接调用状态方法。

3. **确保可追踪性**：状态转换应该能被记录和追踪，以便于故障排查和审计。

具体的实现参考后面的“JAVA版本状态机核心代码实现”。

7. 常见代码实现误区

经常看到工作几年的同学实现状态机时，仍然使用if else或switch case来写。这不是最优的方案，会让实现变得复杂，且容易出现问题。

甚至还见过直接在订单的领域模型里面使用String来定义状态，而不是把状态模式封装单独的类。

还有就是直接调用领域模型更新状态，而不是通过事件来驱动。

错误的代码示例：

通过if else来实现：

```java
if (status.equals("PAYING") {
    status = "SUCCESS";
} else if (...) {
    ...
}
```

直接设置状态，出现不可预测性：

```java
class OrderDomainService {
    public void notify(PaymentNotifyMessage message) {
        PaymentModel paymentModel = loadPaymentModel(message.getPaymentId
());
        // 直接设置状态
        paymentModel.setStatus(PaymentStatus.valueOf(message.status);
        // 其它业务处理
        ... ...
    }
}
```

使用复杂的switch，不好理解：

```java
public void transition(Event event) {
    switch (currentState) {
        case INIT:
            if (event == Event.PAYING) {
                currentState = State.PAYING;
            } else if (event == Event.SUCESS) {
                currentState = State.SUCESS;
            } else if (event == Event.FAIL) {
                currentState = State.FAIL;
            }
            break;
            // Add other case statements for different states and events
        case PROCESSING:
            if (event == OrderEvent.PAY_SUCCESS) {
                ... ...
            }
            ... ...
    }
}
```

8. Spring Statemachine简要介绍

Spring Statemachine项目是一个非常成熟的状态机实现项目，核心思路如下：

1. State ， 状态。一个状态机至少要包含两个或以上的状态。状态与状态之间可以转换。
2. Event ， 事件。事件就是执行状态转换的触发条件。
3. Action ， 动作。事件发生以后要执行动作。
4. Transition ， 变换。也就是从一个状态变化为另一个状态。

网上的介绍和应用示例有很多，可直接参考网上优秀文章，这里不重复。

9. 用JAVA实现一个简洁的交易状态机

Spring Statemachine使用起来稍显复杂，这里重点介绍如何使用Java代码来实现一个简洁的状态机，部分思路参考Spring Statemachine。

我们将采用枚举来定义状态和事件，以及一个状态机类来管理状态转换。下面是详细步骤：

1. 定义状态基类。

```java
/**
 * 状态基类
 */
public interface BaseStatus {
}
```

2. 定义事件基类。

```java
/**
 * 事件基类
 */
public interface BaseEvent {
}
```

3. 定义“状态-事件对”，**指定的状态只能接受指定的事件。**

Java

```java
/**
 * 状态事件对，指定的状态只能接受指定的事件
 */
public class StatusEventPair<S extends BaseStatus, E extends BaseEvent> {
    /**
     * 指定的状态
     */
    private final S status;
    /**
     * 可接受的事件
     */
    private final E event;

    public StatusEventPair(S status, E event) {
        this.status = status;
        this.event = event;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof StatusEventPair) {
            StatusEventPair<S, E> other = (StatusEventPair<S, E>)obj;
            return this.status.equals(other.status) && this.event.equals(other.event);
        }
        return false;
    }

    @Override
    public int hashCode() {
        // 这里使用的是google的guava包。com.google.common.base.Objects
        return Objects.hash(status, event);
    }
}
```

4. 定义状态机。

```java
/**
 * 状态机
 */
public class StateMachine<S extends BaseStatus, E extends BaseEvent> {
    private final Map<StatusEventPair<S, E>, S> statusEventMap = new HashMap<>();

    /**
     * 只接受指定的当前状态下，指定的事件触发，可以到达的指定目标状态
     */
    public void accept(S sourceStatus, E event, S targetStatus) {
        statusEventMap.put(new StatusEventPair<>(sourceStatus, event), targetStatus);
    }

    /**
     * 通过源状态和事件，获取目标状态
     */
    public S getTargetStatus(S sourceStatus, E event) {
        return statusEventMap.get(new StatusEventPair<>(sourceStatus, event));
    }
}
```

5. 实现交易订单（比如支付）的状态机。注：**支付、退款等不同的业务状态机是独立的**。

Java

```java
/**
 * 支付状态机
 */
public enum PaymentStatus implements BaseStatus {

    INIT("INIT", "初始化"),
    PAYING("PAYING", "支付中"),
    PAID("PAID", "支付成功"),
    FAILED("FAILED", "支付失败"),
    ;

    // 支付状态机内容
    private static final StateMachine<PaymentStatus, PaymentEvent> STATE_MACHINE = new StateMachine<>();
    static {
        // 初始状态
        STATE_MACHINE.accept(null, PaymentEvent.PAY_CREATE, INIT);
        // 支付中
        STATE_MACHINE.accept(INIT, PaymentEvent.PAY_PROCESS, PAYING);
        // 支付成功
        STATE_MACHINE.accept(PAYING, PaymentEvent.PAY_SUCCESS, PAID);
        // 支付失败
        STATE_MACHINE.accept(PAYING, PaymentEvent.PAY_FAIL, FAILED);
    }

    // 状态
    private final String status;
    // 描述
    private final String description;

    PaymentStatus(String status, String description) {
        this.status = status;
        this.description = description;
    }

    /**
     * 通过源状态和事件类型获取目标状态
     */
    public static PaymentStatus getTargetStatus(PaymentStatus sourceStatus, PaymentEvent event) {
        return STATE_MACHINE.getTargetStatus(sourceStatus, event);
    }
}
```

6. 定义支付事件。注：**支付、退款等不同业务的事件是不一样的。**

```java
/**
 * 支付事件
 */
public enum PaymentEvent implements BaseEvent {
    // 支付创建
    PAY_CREATE("PAY_CREATE", "支付创建"),
    // 支付中
    PAY_PROCESS("PAY_PROCESS", "支付中"),
    // 支付成功
    PAY_SUCCESS("PAY_SUCCESS", "支付成功"),
    // 支付失败
    PAY_FAIL("PAY_FAIL", "支付失败");

    /**
     * 事件
     */
    private String event;
    /**
     * 事件描述
     */
    private String description;

    PaymentEvent(String event, String description) {
        this.event = event;
        this.description = description;
    }
}
```

以上状态机定义完成。下面是如何使用的代码：

7. 在支付单模型中声明状态和根据事件推进状态的方法：

```java
/**
 * 支付单模型
 */
public class PaymentModel {
    /**
     * 其它所有字段省略
     */

    // 上次状态
    private PaymentStatus lastStatus;
    // 当前状态
    private PaymentStatus currentStatus;


    /**
     * 根据事件推进状态
     */
    public void transferStatusByEvent(PaymentEvent event) {
        // 根据当前状态和事件，去获取目标状态
        PaymentStatus targetStatus = PaymentStatus.getTargetStatus(currentStatus, event);
        // 如果目标状态不为空，说明是可以推进的
        if (targetStatus != null) {
            lastStatus = currentStatus;
            currentStatus = targetStatus;
        } else {
            // 目标状态为空，说明是非法推进，进入异常处理，这里只是抛出去，由调用者去具体处理
            throw new StateMachineException(currentStatus, event, "状态转换失败");
        }
    }
}
```

代码注释已经写得很清楚，其中StateMachineException是自定义，不想定义的话，直接使用RuntimeException也是可以的。

8. 在支付业务代码中的使用：只需要 paymentModel.transferStatusByEvent(message.getEvent())。

```java
/**
 * 支付领域域服务
 */
public class PaymentDomainServiceImpl implements PaymentDomainService {

    /**
     * 支付结果通知
     */
    public void notify(PaymentNotifyMessage message) {
        PaymentModel paymentModel = loadPaymentModel(message.getPaymentId());
        try {

            // 状态推进
            paymentModel.transferStatusByEvent(message.getEvent());
            savePaymentModel(paymentModel);
            // 其它业务处理
            ... ...
        } catch (StateMachineException e) {
            // 异常处理
            ... ...
        } catch (Exception e) {
            // 异常处理
            ... ...
        }
    }
}
```

上面的代码只需要完善异常处理，优化一下注释，就可以直接用起来。

好处：

1. 定义了明确的状态、事件、以及通过什么事件可以推动哪些状态转换。
2. 状态机的推进，只能通过“当前状态、事件、目标状态”来推进，不能通过if else 或case switch来直接写。比如：STATE_MACHINE.accept(INIT, PaymentEvent.PAY_PROCESS, PAYING);
3. 避免终态变更。比如线上碰到if else写状态机，渠道异步通知比同步返回还快，异步通知回来把订单更新为“PAID”，然后同步返回的代码把单据重新推进到PAYING。

相对于Spring Statemachine而言，有两个优势：

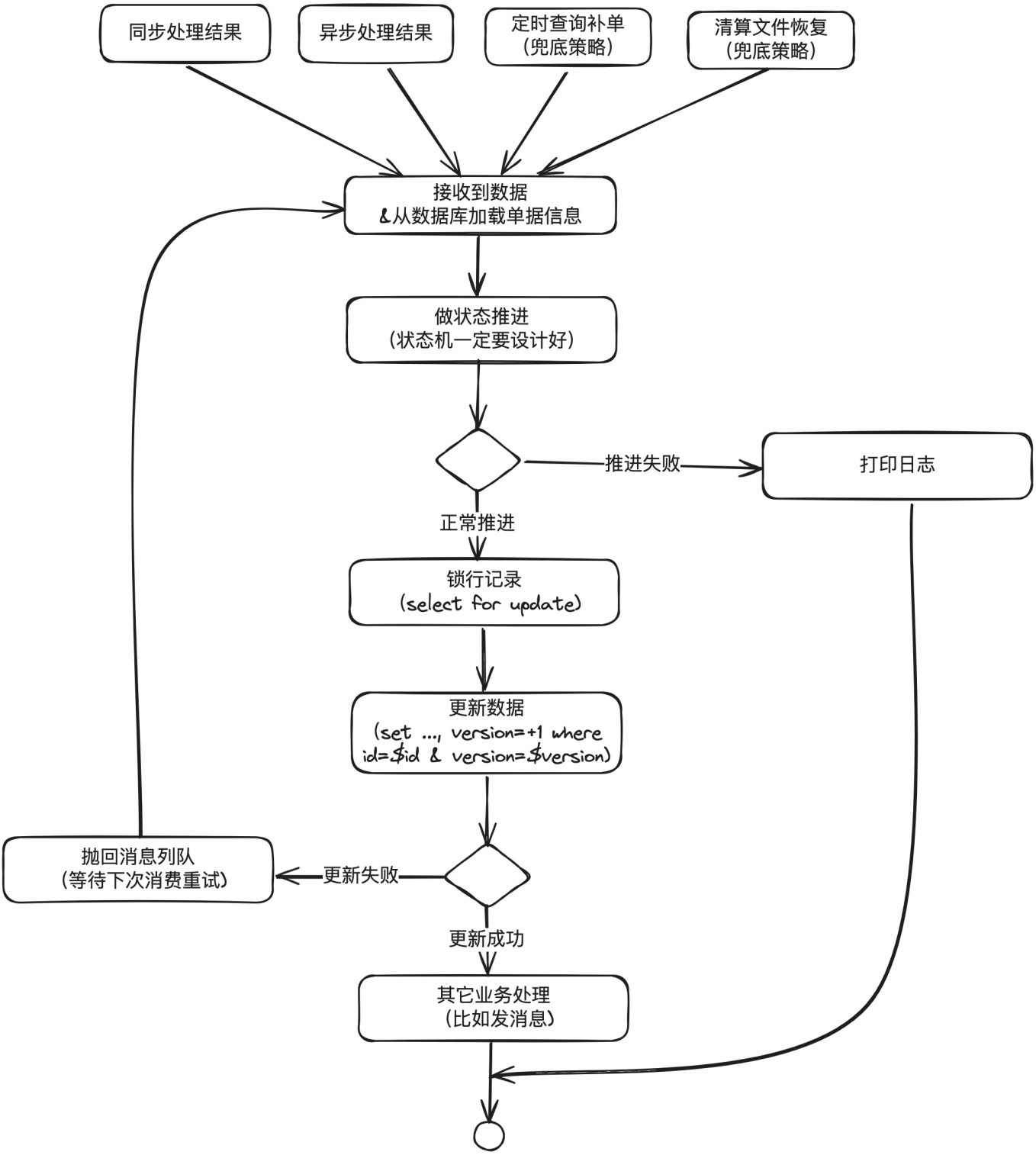
1. 更为简洁，所有状态机相关的代码也就100行左右。
2. 使用起来更清晰明确。

10. 并发更新问题

有位读者提到：“状态机领域模型同时被两个线程操作怎么避免状态幂等问题？”

这是一个好问题。在分布式场景下，这种情况太过于常见。同一机器有可能多个线程处理同一笔业务，不同机器也可能处理同一笔业务。

业内通常的做法是设计良好的状态机 + 数据库锁 + 数据版本号解决。



简要说明：

1. 状态机一定要设计好，只有特定的原始状态 + 特定的事件才可以推进到指定的状态。比如INIT + 支付成功才能推进到sucess。
2. 更新数据库之前，先使用select for update进行锁行记录，同时在更新时判断版本号是否是之前取出来的版本号，更新成功就结束，更新失败就组成消息发到消息队列，后面再消费。（这里为什么要有select for update，又有版本号判断？因为先使用普通的select数据进行业务处

理，在update时需要判断版本号，避免被其它线程已经更新。同时因为更新多张表，所以需要select for update做为一个完整的事务）

3. 通过补偿机制兜底，比如查询补单。
4. 通过上述三个步骤，正常情况下，最终的数据状态一定是正确的。除非是某个系统有异常，比如外部渠道开始返回支付成功，然后又返回支付失败，说明依赖的外部系统已经异常，这样只能进人工差错处理流程。

11. 结束语

状态机在交易系统中扮演着不可或缺的角色。通过合理的状态机设计，我们可以清晰地管理交易单据的状态变化，提高系统的健壮性和可维护性。本文介绍了状态机设计原则、常见误区和最佳实践，并展示了一个简洁的JAVA版本状态机的核心代码实现。希望本文能为大家在实际项目中设计和实现交易单据状态机提供一些有益的参考。

这是《图解支付系统设计与实现》专栏系列文章中的第（13）篇。

深耕境内/跨境支付架构设计十余年，欢迎关注并星标公众号“隐墨星辰”，和我一起深入解码支付系统的方方面面。

专栏系列文章PDF合集不定时更新，欢迎关注我的公众号“隐墨星辰”，留言“PDF”获取。

有个支付系统设计与实现讨论群，添加个人微信（yinmon_sc）备注666进入。

隐墨星辰 个人微信

10年顶尖境内/跨境支付公司架构经验

著有《图解支付系统设计与实现》

备注666进支付讨论群