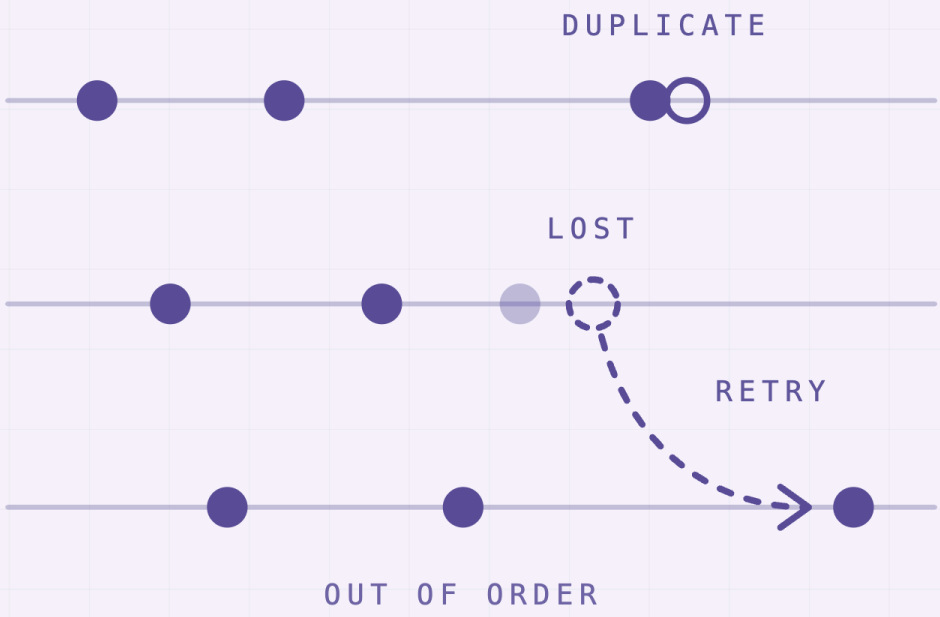
THE UNRELIABLE
SYSTEMS TRILOGY

II



设计可靠的事件驱动系统

Muhammet Şafak

设计可靠的事件驱动系统

消息传递、retry、幂等性、outbox、消息顺序与 schema 演进

Muhammet Şafak

© 2026 Muhammet Şafak
可免费下载和阅读。

目录

前言 5

第一部分 — 基础 **11**

第 1 章 — 快乐路径的谎言 12

第 2 章 — 消息传递语义 22

第 3 章 — 务实的故障模型 35

第二部分 — 可靠地发布 **50**

第 4 章 — 双写问题与 transactional outbox 51

第 5 章 — Relay、轮询与变更数据捕获 67

第三部分 — 可靠地消费 **84**

第 6 章 — 正确地重试 85

第 7 章 — 幂等性 105

第 8 章 — Poison message 与死信 124

第 9 章 — 消息顺序 143

第四部分 — 工作流与契约 **160**

第 10 章 — Saga 与长时间运行的工作流 161

第 11 章 — 事件设计 179

第 12 章 — Schema 演进 195

第五部分 — 运维与验证 **212**

第 13 章 — 可观测性与对账 213

第 14 章 — 恢复、灾难与 broker 迁移 231

第 15 章 — 测试事件驱动系统 252

第 16 章 — 可靠性检查清单与结语 266

附录：术语表与模式速查 276

前言

设想 Parcelly 的一个周二下午。Parcelly 是一个并不存在的市场平台。结账流程已经平稳运行了数月。买家下了一个订单，Orders 服务把它写入自己的数据库，随后一个 `OrderPlaced` 事件发往消息 broker。Payments 对银行卡进行预授权，Inventory 预留库存，Shipping 安排包裹发货。所有仪表盘都是绿色的。

随后，一次滚动部署经过 Orders。每个实例处理订单的方式都一样：先把订单提交到自己的数据库，再把 `OrderPlaced` 发布到消息 broker。在那几秒钟里，有的实例正在关闭，有的实例正在启动。某个实例提交了一个订单，却在调用发布之前被停掉了。订单安全地存了下来，事件却从未发出。没有任何东西发出巨响般的崩溃，也没有任何告警触发。到第二天早晨，买家看到一个标记为已下单的订单，但 Payments 从未对银行卡预授权，Inventory 也没有预留过任何一件商品。Orders 以为工作正在进行，而系统的其余部分从未听说过它。

没有人写出通常意义上的 bug。每一行代码都做了它所写的事。系统是在代码行与行之间的空隙里失败的：在数据库提交与网络调用之间，在消息被投递与被理解之间，在“我们发出去了”与“对方做完了”之间。

本书讨论的正是这个空隙。

本书缘起

事件驱动架构的名声有问题，而且来自两个方向的评价都不冤枉。支持者描述的是松耦合的服务：各自独立扩展，无需协调即可演进。批评者描述的是分布式的意大利面条：没有人说得清下订单时究竟会发生什么。两者描述的是同一个系统在不同日子里的样子。

区分这两种日子的，很少是 broker 的选择、框架，甚至架构图。区分它们的是一份并不光鲜的短问题清单。如果这条消息被投递两次，会发生什么？如果它始终没有到达，会发生什么？如果它在本应排在它前面的那条

消息之后才到达，会发生什么？如果消费者每一次都失败，会发生什么？如果生产者需要修改一个字段，而一百个消费者仍在运行旧代码，又会发生什么？

这些问题就是本书的论点。可靠的事件驱动系统围绕**故障语义**构建，而不是围绕消息 broker：对于某个接缝，什么可能失败，如何观察到失败，以及哪项保证能够在失败之后保留下来。接缝是指一个组件把工作交给另一个组件的任何位置，例如数据库提交之后紧跟一次发布，或者一次投递之后紧跟其效果。这里的模式之所以存在，是因为网络和数据库，而不是因为某个具体产品；broker 只是又一个组件，同样有它自己的接缝。

团队通常是一次遇到一个这样的问题，而且是在生产环境中，按系统碰巧出故障的顺序。每个答案都是单独学到的，这些教训很少汇集到同一个地方。broker 的文档会说明如何发布和消费，但对于围绕这两个动词的一个正确服务应当是什么样子，却沉默得多。

本书试图成为那个汇集之处。它是一份指南，讲述让事件驱动系统可靠的各种模式：transactional outbox、幂等消费者、带退避与抖动的 retry、反压、死信队列、消息顺序策略、saga、审慎的事件设计、schema 演进、可观测性、数据丢失与更换 broker 后的恢复以及测试。我尝试把每一种模式都当作有代价的决策来对待，而不是当作必须遵循的规则。对 Parcelly 的支付合适的模式，对它的通知可能就是浪费，在这一点重要的时候，本书会明说。

我也想把本书不是什么讲清楚。它不是针对任何特定 broker 的教程。它也不是曾经被命名过的所有企业集成模式的目录。它不承诺遵循本书就能让系统正确。可靠性不是可以额外添加的属性；它是一组已经有意决定如何处理的故障，加上一组已经有意决定接受的故障。我的目标是让这些决定变得可见。

本书面向的读者

本书写给构建或运维通过事件通信的服务的**后端和平台工程师**，也写给必须决定这些服务在出错时应如何表现的**技术负责人**。如果曾经盯着一夜之间膨胀的死信队列发呆，争论过某个消费者是否需要幂等，或者接手过一个没人能完整解释其事件的 topic，那么这正是我心目中的读者。

不需要是消息传递方面的专家。应当熟悉以下内容：

- 关系型数据库与事务：提交是什么，唯一约束起什么作用，隔离在大致意义上指什么。

- 通过网络通信的服务，以及超时和 HTTP 式请求与响应的基本词汇。
- 消息 broker 的一般概念：接收来自生产者的消息，并把它们交给消费者的东西。

不需要具备任何特定产品的经验。当某一章提到 Kafka、RabbitMQ 或 Amazon SQS 时，是把它们作为一类系统的例子，描述的是普遍成立的行为，而不是某个具体版本中的行为。即使一个也没有用过，本书依然读得通。如果曾经深入使用过其中某一个，会发现自己所知的一部分是它特有的，本书会尽量标出是哪些部分。

如果是不每天写代码的技术负责人，可以阅读每一章的开篇场景、权衡部分和要点，便能获得大部分决策价值。伪代码是留给需要看清事务边界究竟落在何处的人的。

本书的结构

本书分为五个部分，初次阅读时按顺序读，之后可以按任意顺序使用。

第一部分，基础（第 1 章至第 3 章）。首先把快乐路径拆开来看。第 1 章展示一个看似简单的流程如何隐藏着十几项决策。第 2 章明确消息传递保证究竟意味着什么，以及为什么 at-least-once 投递是现实的基线。第 3 章构建一个务实的故障模型，这是一套描述事物如何出错的小词汇，涵盖从超时和崩溃，到始终失败的消息，再到整个存储的丢失。本书其余部分把它用作共同的语言。

第二部分，可靠地发布（第 4 章和第 5 章）。第 4 章处理双写问题，即开篇场景中的那个缺口，并引入 transactional outbox。第 5 章介绍把 outbox 行搬运到 broker 的机制，比较轮询式 relay 与变更数据捕获，以及两者各自的代价。

第三部分，可靠地消费（第 6 章至第 9 章）。这是最长的一个部分，因为大多数重复效果、卡住的队列和丢失的更新都发生在消费者一侧。第 6 章讲 retry：何时 retry、何时不 retry，退避为什么需要抖动，以及如何避免消费者被自己的 retry 压垮。第 7 章讲幂等性与 inbox。第 8 章处理 poison message 与死信。第 9 章转向消息顺序，以及通常能够要求的顺序是多么有限。

第四部分，工作流与契约（第 10 章至第 12 章）。第 10 章为跨越多个服务、无法共享同一个事务的工作流引入 saga。第 11 章讲事件本身的设计：里面放什么，不放什么，如何命名，以及消息信封如何经受住以多

种语言写入和读取。第 12 章讲 schema 演进，这门安静的纪律让生产者和消费者能够以不同的速度变化。

第五部分，运维与验证（第 13 章至第 16 章）。第 13 章讲如何看清系统正在做什么，以及对账，即把一个系统的视图与另一个系统的视图相互核对的习惯。第 14 章讲从数据丢失、区域丢失和更换 broker 中恢复。第 15 章讲如何测试那些最严重的 bug 只在重复、延迟和乱序下才出现的代码。第 16 章把所有内容汇集成一份可靠性检查清单，并为本书作结。

附录包含一份术语表和一份模式速查，无需重读某一章，便可以查阅某个内容。

第 1 章至第 15 章的每一章都遵循同样的结构。每章以一个简短的场景开篇，解释问题为何困难，介绍模式，然后进入我认为最有价值的两个部分：**权衡与成本**，说明何时不应使用该模式；以及**常见错误**。每章以五到七条要点结束。如果时间紧张，这三个部分带来的收获会超出预期。

可以有选择地阅读各章。如果今天的痛点是不断增长的死信队列，可以直接跳到第 8 章，并按需沿着引用回到第 6 章和第 7 章。如果正在设计一个新服务，我会建议在写第一行发布代码之前先读第二部分，因为 outbox 在一开始就采用，要比事后补上容易得多。

认识 Parcelly

贯穿全书，由一个虚构的系统承载所有示例：**Parcelly**，一个在线市场平台，独立卖家在上面列出商品，买家下单购买。它有六个服务。

- **Orders** 负责订单生命周期，大多数工作流从这里开始。
- **Payments** 对资金进行预授权和扣款，并与外部支付提供方通信。
- **Inventory** 跟踪库存，并持有临时预留。
- **Shipping** 安排发货，并把包裹交给承运商。
- **Notifications** 发送电子邮件、推送消息和短信，并且绝不能阻塞任何其他环节。
- **Ledger** 以 append-only 的复式记账方式记录资金流动。

每个服务拥有自己的关系型数据库，并且从不读取其他服务的数据库。它们通过消息 broker 交换事件来通信，事件的名称例如 `OrderPlaced`、`PaymentAuthorized`、`StockReserved`、`PaymentFailed` 和

`ShipmentScheduled`。这些事件遵循一致的形态：一个携带标识符、类型、schema 版本、关联字段和 payload 的消息信封，这样各章就能用同一种口径来谈论它们。

我有意选择了市场平台。收款、预留商品和运送包裹，是一个丢失消息或重复效果的代价很快就会显现的领域。重复的邮件只是一种烦扰。重复的扣款则意味着一次退款、一张支持工单和一次道歉。丢失的预留则意味着一件超卖的商品。从无害到昂贵的这道梯度，恰恰是决定每个流程应配备多少可靠性机制所需要的。

Parcelly 是虚构的，它的事故也是。每章开篇的场景都是为说明某种故障形态而编造的，并且始终明确按虚构来叙述。我会用第一人称来表达判断，例如“我会在这里加一个唯一约束”或“在我看来这不值得付出这份复杂度”，但不会把推理包装成亲身经历的故事。当本书说某件事会发生时，可以相信那是因为这些系统的工作方式，而不是因为某段轶事。

本书还有意做到**与 broker 无关**。当某个论点取决于 broker 是基于日志的（消费者在有序分区中跟踪 offset），还是基于队列的（消息逐条被 ack 并重新投递）时，我会说明是哪一种。在无关紧要的地方，我不会假装它有关。

如何阅读代码

本书中的代码是**伪代码**。它不属于任何语言，无法编译，其中也没有任何调用真实库的地方。这是有意为之。每个片段的要点，是展示事务从哪里开始、到哪里结束，消息在哪里被 ack，以及某一步中途失败时会发生什么。这些事实在每种语言中都是一样的，语法只会碍事。

代码块很短，最多 30 行。在表的 schema 有帮助的地方，使用类 SQL 的记法书写。图表使用 Mermaid，每张图之前都有一句话说明应当关注什么。如果要把某个模式转译到自己的技术栈中，应当把伪代码视为对顺序与原子性的规约，而不是一种实现。

关于确定性的说明

本书会给出建议，其中一些我会坚持到底。例如，当事件的丢失对业务有影响时，我不会用单纯的双写来发布事件。另一些建议则取决于品味或具体情境，我会尽量说明哪些属于哪一类。

也会有一些地方，诚实的回答是“视情况而定，以下是取决于什么”。retry 预算、死信保留期限、分区数量以及合适的消息顺序级别，都取决于我看不到的、关于业务和团队的事实。在这些地方，我会列出变量，而不是编造一个数字。如果读完一章后，知道了在自己的系统中该提出哪些问题，我就认为这是成功。

可靠性工作有一个令人遗憾的特点：成功的时候，没有任何可见的事情发生。订单被履行，账目平衡，没有人会写一条消息来感谢。希望本书能帮助构建出那样一种安静。

让我们从快乐路径开始，也从不要信任它的理由开始。

第一部分

基础

第 1 章 — 快乐路径的谎言

设想 Parcelly 的第一年。Orders 通过 HTTP 调用 Payments，等待答复，然后调用 Inventory，再次等待，接着调用 Shipping。代码自上而下阅读，故障就是一个异常，看到错误页面的买家可以自行执行 retry。支付提供方慢的时候它就慢，Inventory 宕机的时候它就垮，但没有人搞不清发生了什么。

后来团队决定转向事件。Orders 将向消息 broker 发布 `OrderPlaced`，其他服务按各自的节奏作出反应。演示漂亮极了。买家点击“下单”，一秒之内库存被预留，银行卡被扣款，订单被确认，发货被安排。所有人都鼓掌。

现在设想生产环境的第一个月。一位买家被扣了款，订单却永远停留在“pending”。另一个订单被发了两次货。第三个订单有一条账本条目，却没有对应的付款，没有人说得清哪个服务才是对的。支持团队问：“订单 81724 出了什么事？”诚实的回答是：四个服务各自知道其中一部分，没有任何一个知道全部。

演示中没有任何错误。它只是不完整，因为演示是一个每一步都成功的故事。本章要回答的问题是：团队离开请求–响应时放弃了什么，以及在任何人动手把它重新建起来之前，“可靠”必须意味着什么。

问题及其难点

快乐路径，是每个组件都在运行、每次网络调用都到达、每条消息都恰好按顺序投递一次、每个消费者都完成其所开始的工作时，所发生的事件序列。它是画在白板上的路径，也是演示所演练的路径。同时，在一个持续运行数月的系统里，它也是最罕见的路径。

这些故障并不稀奇。部署会重启进程。连接会在写入途中断开。broker 会故障转移到副本。消费者花的时间超出预期，它的消息被交给了别人。这些情况在一个繁忙的系统里每天都会在某处发生。糟糕的生产月份，通常的形态是少数几个普通事件撞上了为快乐路径编写的代码。

之所以困难，是因为事件驱动系统移除了让故障可见的那样东西。在同步调用中，调用方在等待，结果沿同一条连接返回：一个结果或一个错误，紧挨着引发它的代码。在事件驱动的设计中，生产者把消息交给 broker 后就继续往前走。如果三个服务和十秒钟之后出了问题，生产者早已不在，证据散落在尚未有人关联起来的日志里。

显而易见的应对是让每个部件都可靠：一个好的 broker，经过充分测试的服务，每次调用都带 retry。每一项都有帮助，但没有一项足够，因为造成伤害的故障存在于组件之间的缝隙里。一条消息被写入数据库却从未发布，或者被处理了却从未 ack，这并不是任何一方组件的 bug。这是接缝的属性。本书的大部分内容都是关于接缝的。

对于每一个接缝，本书都问同样的三个问题：什么可能失败，如何观察到失败，以及哪项保证能够在失败之后保留下来。我把这些答案称为该接缝的**故障语义**。broker 的文档会给出 broker 自身接缝的语义。围绕在它周围的接缝，即位于数据库与发布调用之间、或位于一次投递与其效果之间的接缝，则需要自己去弄清楚，第 3 章会展示如何做到。

同步耦合与异步耦合

转向事件，通常被描述为“解耦”，这个词承诺的比它兑现的多。更准确的说法是，耦合改变了形态。

同步调用在**时间**、**可用性**和**知识**三个方面把两个服务耦合在一起。被调用方必须此刻正在运行，调用方的延迟包含了被调用方的延迟，而且调用方知道被调用方是谁、会如何答复。好处是故障模型简单：调用方知道调用失败了，可以立即决定 retry、回退或报告错误。代价是整条链路的可用性只取决于最薄弱的一环。承运商 API 一慢，结账就慢。

通过 broker 传递的消息，消除了时间和可用性上的耦合。Inventory 重启时，Orders 仍然可以接受订单，因为 broker 会保留消息，直到 Inventory 回来。新的消费者，例如分析服务，可以在 Orders 不做任何改动的情况下开始监听。这一收益是真实的，也是采用这种风格的理由。

但耦合只是转移了，并没有消失。服务现在通过**事件的契约**耦合在一起：它的名称、字段和含义。它们通过 **broker 的行为**耦合在一起：它对持久性、顺序和投递作出了什么承诺。它们还通过对彼此时序的、没有人写下来的假定耦合在一起。第 11 章和第 12 章处理契约，第 2 章处理 broker。

简而言之，同步系统是响亮地、成链地失败；异步系统是安静地、零散地失败。

时间解耦的代价

时间解耦意味着生产者和消费者不必同时处于活动状态。这是用事件换来的主要东西，因此值得精确地说明它的代价。

生产者不再得知结果

当 Orders 同步调用 Payments 时，它在回复买家之前就知道支付是否成功。使用事件时，Orders 发布 `OrderPlaced`，然后回复“已收到”。支付是否成功成了稍后的问题，答案作为另一个事件到来，如果它会到来的话。订单现在有了一个中间状态，“已下单但尚未确认”，系统必须决定在这一状态持续期间买家看到什么，以及如果它持续得比计划更久会怎样。

没有人负责整个流程

在同步版本中，一个函数包含了整个序列，业务流程在一个地方就能读到。在事件驱动版本中，流程是散布在各服务之中的反应之和。Payments 对 `OrderPlaced` 作出反应，Orders 对 `PaymentAuthorized` 和 `StockReserved` 作出反应，Shipping 对 `OrderConfirmed` 作出反应。流程只是作为涌现属性存在，没有人能指出它在哪里被定义。

一切正常时这无关紧要，出了问题时代价高昂，因为“这个订单卡在了哪里？”没有一个可以查看的单一位置。第 10 章会借助 saga 回到这个问题，第 13 章则介绍要回答这个问题所需观测的内容。

故障不再是异常

在调用链中，错误沿调用栈向上传播，直到有东西处理它。在消息流中没有调用栈。失败的消费者可以对消息执行 ack 然后丢失它，可以拒收它让它被重新投递，也可以把它暂放起来交给人处理。每一种都是一项策略决策，其后果落在从未见到该错误的服务上。在我看来，同步系统中的错误处理是控制流的问题，而在异步系统中则是设计的问题，因为处理分散在时间、进程，有时还有团队之间。

时间成为变量

在“已发布”与“已处理”之间有一段间隔，除非自己给它设定上限，否则它没有上界。在这段间隔内，商品可能售罄，买家可能取消，价格可能变化。消费者必须判断，一个在发布时为真的事件，在被处理时是否仍然有意义。

> **注意**：这就是事件以过去时命名的原因。`OrderPlaced` 是关于一个已经过去的时刻的事实。把它当作“立即行动”的指令的消费者，悄悄假定了没有任何时间流逝。命令，例如 `CapturePayment`，请求某个服务去做某件事，并且可能被拒绝；事件报告的是已经发生的事情。第 11 章会进一步展开这一区别。

“最终”的真实含义

对**最终一致性**的大多数解释都只有一句话：如果不再有新的更新，所有副本最终会收敛到同一个值。这句话准确，但对构建结账流程的人几乎毫无用处，因为它没有说“最终”有多长，也没有说在等待期间该做什么。

“最终”不是一段时长

一个在 200 毫秒内收敛的系统，和一个在夜间批处理之后才收敛的系统，都是最终一致的。对于盯着确认页面的买家来说，这是两种不同的产品。对每一个重要的跳，我都会要求给出一个数字，哪怕是粗略的。比如说，Orders 必须在扣款后的几秒之内显示已确认的订单，而 Notifications 可以用一分钟。这些数字只是示意，实际的数字会有所不同，但它们把“最终”变成了可以测试、可以告警的需求。没有上界的延迟不是一项设计选择，而是设计的缺席。

“最终”不是保证

只有在每条重要的消息最终都被投递、每个消费者都处理了它、并且没有任何一步悄悄丢弃它时，才会发生收敛。如果一条消息在数据库提交与 broker 发布之间丢失，各视图就不会收敛。它们会一直保持不同，直到有人或某个修复作业发现。“最终”是一个有条件的承诺，条件是系统具备让丢失可被检测、可被修复的机制。选择一个 broker 并不会提供这些机制。本书的第二部分和第三部分防止大部分丢失，第 13 章检测并修复其余部分。

等待期间该做什么

在不一致的窗口期内，其他读取方会看到一个中间状态。考虑下面这个订单的快照，时间是买家被扣款之后片刻：

```
// PaymentCaptured not yet consumed
orders.status      = "CAPTURE_REQUESTED"
payments.status    = "CAPTURED"
inventory.state    = "RESERVED"
shipping.shipment  = none
// LedgerEntryPosted not yet emitted
ledger.entries     = none
```

每个服务对自己的数据都是正确的。组合起来看似不一致，而在几秒之内这是正常的。只有当它一直保持这样，才成为问题。有用的运维问题不是“这些服务此刻一致吗？”，而是“这个订单处于这个状态已经多久了，是否超过了允许的时间？”

可靠性的工作定义

“可靠”这个词用得太随意，以至于两位工程师可以同意一个系统应当可靠，却指的是相反的东西。我想要一个可以检验的定义。对于事件驱动系统，我使用四个属性。它们说明了要保护的是什么；上面的三个问题则是据以检验某个接缝是否满足它们的方法。

无丢失。每个代表业务事实的事件，都到达每个需要它的消费者。丢失是破坏性最大的故障，因为它是无声的：系统继续运行，唯一的症状是一种缺席，直到买家投诉之前没有人注意到。无丢失是整条路径的属性，而不是 broker 的属性。如果生产者从未成功存下消息，或者消费者在完成工作之前就 ack 了，那么持久的 broker 也无济于事。第 2 章检视 broker 承诺了什么，第 4 章和第 5 章处理生产者一侧。

无有害重复。有些消息会被投递不止一次。这不是某个特定 broker 的缺陷，而是不丢失消息的代价：当发送方无法判断消息是否到达时，安全的做法是再发一次。可靠的系统不会试图在传输层阻止重复，而是让重复变得无害。“有害”这个词很重要。两封相同的邮件只是烦扰，而对同一笔付款的第二次扣款则是一起资金事故。第 7 章讲幂等性，这是主要的工具。

有界延迟。每个事件都在业务可容忍的时间内被处理，否则系统会明确大声地说明它没有做到。一条被正确投递却晚了三天的消息，在许多用途上就是一条丢失的消息。有界并不意味着快。它意味着存在一个上限，有人

选定了它，并且一旦突破就会产生告警。一次支付确认可能需要几秒；一份每周对账单可能容忍几小时。第 6 章处理 retry 与退避，第 13 章处理 lag 与告警。

可解释性。对于任何一个订单，在任何时间，都有人能够重建出发生了什么：产生并消费了哪些事件，每个服务作出了什么决定以及为什么。这是最不光鲜的属性，却决定了其他三个能否得以保持。一个丢失了消息但能确切指出丢在哪里的系统，是可以修复的。一个以没人能追查的方式出错的系统则不能。第 13 章介绍这需要的标识符、记录和比较。

这四者相互牵制，这就是为什么只包含其中之一的定义是危险的。追求无丢失而无限 retry，会威胁有界延迟。为了限定延迟而丢弃慢消息，会威胁无丢失。为避免重复而拒绝 retry，又会带回丢失。可靠性是一种有意的平衡，按流程选定，并写下来。

快乐路径为何在生产环境中失效

有了这个定义，回到 Parcelly 的第一个月，按属性对故障分类。少数几种机制造成了大部分损害。

Parcelly 的现象	受威胁的属性	根本原因	章
订单已被接受，但下游从未发生任何事	无丢失	事件已提交到数据库，却从未发布	4
订单被发了两次货	无有害重复	被重新投递的消息再次被处理	7
一条坏消息阻塞了一个分区	有界延迟	一条消息每次都失败，并被无休止地 retry	8
订单在“已下单”之前就“已确认”	可解释性	相关事件被乱序处理	9
账本与付款不一致	可解释性	没有共同的标识符，无法比较各视图	13

第四行是尴尬的一行。乱序不会造成重复。它会损坏状态，例如一次过期的更新覆盖了较新的更新，而这四个属性中没有一个直接指出这一点。我把它归在可解释性之下，因为其症状是一段没人能解释的历史，而第 9 章处理状态损坏本身。

第一行值得走一遍，因为它是最常终结蜜月期的故障。

```
function placeOrder(request):
    transaction:
        orderId = insertOrder(request)
    // commit happens here; the order now exists in the database

    // the process can crash right here, between the two steps
    publish("orders", OrderPlaced(orderId))
```

这个代码块并不安全，它是作为有缺陷的基线展示的。数据库提交和 broker 发布是两个没有共享事务的系统，本书称之为**双写**。如果进程在这两行之间停止，订单存在，却没有人听说它。如果把这两行反过来，可能会为一个从未提交的订单发布事件。两种顺序都不正确，而且除了恰好在最糟糕的时刻杀死进程的测试之外，二者都能通过所有测试。

这不是罕见的 bug。这是从未有过故障模型的设计的默认行为。第 3 章给出模型，第 4 章给出修复。

本书地图

本书其余部分按团队通常遇到的顺序，回答这个定义所引出的问题。

将会提出的问题	章
broker 究竟承诺了什么，又没有承诺什么？	2
什么会出故障，出故障时会看到什么？	3
如何修改数据库并发布事件而不丢失事件？	4
已发布的事件如何从数据库到达 broker？	5
如何 retry 而不让事情变得更糟？	6
如何让重复处理变得无害？	7
对一条永远无法成功的消息该怎么办？	8
顺序何时重要，如何保持？	9
如何跨服务协调工作并撤销它？	10
事件应当包含什么，叫什么名字？	11
如何修改事件而不破坏消费者？	12
如何查看、度量并修复正在发生的事？	13
如何从数据丢失、区域丢失或更换 broker 中恢复？	14
如何测试故障路径？	15

将会提出的问题	章
上线之前要检查什么？	16

第一部分由本章开启，构建词汇和故障模型。第二部分讲可靠地发布，第三部分讲可靠地消费，第四部分讲工作流与契约，第五部分讲运维与验证系统。初次阅读时按顺序读各部分。如果今天正在与某个具体问题搏斗，这张表就是索引。

权衡与成本

转向事件并没有错，本章也不是在反对它。本章主张的是睁大眼睛去转向，因为成本是真实的，并且付在具体的地方。

在状态的复杂度上付出。同步流程只有两种结果：完成或失败。事件驱动流程有许多中间状态，每一个都必须被命名、展示给用户、受到监控，并最终得到解决。总得有人负责这台状态机。

在运维技能上付出。调试一条调用链需要一份堆栈跟踪。调试一条消息流需要 correlation 标识符、lag 仪表盘以及推理时序的能力。缺少这些的团队会发现，即使事故变得更少，理解一起事故所需的时间也在增长。这一成本落在值班人员身上。

在测试工作量上付出。快乐路径几乎不费力气就能测试。故障路径则需要重复的消息、乱序的消息以及在尴尬时刻的重启。第 15 章展示如何做到这一点而不产生缓慢、不稳定的测试套件，但这份工作量并不为零。

在简单事情的延迟上付出。原本是一次函数调用的操作，变成了经过 broker 的若干跳。当调用方现在就需要答案时，例如在显示价格之前校验优惠码，同步调用才是更合适的工具。

当两个步骤必须一起成功或一起失败，并且位于同一个服务和同一个数据库中时，我不会使用事件。本地事务比本书中的任何模式都更简单、更强，也更便宜。当团队没有能力运维 broker 及其周边的监控时，或者当系统小到一个带单个数据库的单体部署单元就能胜任时，我也会犹豫。许多好的系统混合使用两种风格：对需要答案的问题用同步调用，对别人可能关心的事实用事件。我会按每一次交互来选择，而不是按系统来选择。

这个四属性的定义也有其局限。它是一面透镜，而不是规约。它不会说明适合业务的延迟上界是多少，也不会说明对于某个消费者哪些重复是有害的。我不知道具体的数字，而且这些答案很少仅凭工程就能得出。我能提

供的是去追问它们的习惯。

常见错误

- **只测试快乐路径**。这很诱人，因为功能需求描述的就是它，而且它很快就能通过。应当把流程图中的每一个箭头都当作一个关于故障的问题，并为每个答案至少写一个测试（第 15 章）。
- **把 broker 当作可靠性层**。broker 宣传持久性，这使得这种做法很诱人。broker 覆盖的是从接收到投递之间的那一段，而伤害最大的故障位于它之前和之后。应当列出 broker 承诺了什么，然后列出围绕这一承诺的缺口（第 2 章和第 3 章）。
- **把“最终”当作设计**。它听起来像一个决定，却回避了“究竟多久”这个更难的问题。应当为每个重要的异步跳附加一个上界和一条告警，并说明用户在等待期间看到什么。
- **假定重复太罕见而无需处理**。在安静的测试环境中，重复从不出现。应当基于每条消息都可能到达不止一次的假定来设计消费者（第 7 章）。
- **把提交到数据库和发布到 broker 当作两个普通步骤**。它读起来很自然，在每个演示里都能工作。修复方式是第 4 章的 transactional outbox。
- **把可靠性定义为正常运行时间**。正常运行时间的度量成本低廉，但一个服务可以整月都在运行，却仍然丢失消息或延迟处理。应当直接度量这四个属性：丢失了什么，重复了什么，晚了多久，以及能否解释它们。

要点

- 快乐路径是一个每一步都成功的故事，只针对它测试的系统，仅仅被证明在那个很难长期成立的情形下可以工作。
- 从同步调用转向事件并不会消除耦合；它把耦合从时间和可用性，转变为契约、broker 行为和没有写下来的时序假定。
- 事件驱动的故障是安静的，并且分散在各服务之间，因此没有故障模型的设计往往要从生产环境中去了解自己的接缝。
- “最终”是一个没有附带时长的有条件承诺，因此每个重要的异步跳都需要一个上界、一条告警以及一个为等待期间定义好的、面向用户的状态。

- 可靠的系统具备无丢失、无有害重复、有界延迟和可解释性；由于这些属性相互牵制，每个流程都需要一个有意的、写下来的平衡。
- 组件之间的接缝，例如数据库提交之后紧跟 broker 发布，造成的损害比组件本身更大，本书的大部分内容都是关于这些接缝的，并用三个问题来检视：什么可能失败，如何观察到失败，以及哪项保证能够在失败之后保留下来。
- 事件并不总是合适的工具：当各步骤可以共享本地事务，或者调用方需要即时答案时，同步调用更简单，也更强。

下一章考察 broker 本身。在围绕它的缺口进行设计之前，需要确切知道队列或日志在声称一条消息已被投递时承诺了什么，又悄悄地把什么留给了使用者。

第 2 章 — 消息传递语义

设想 Parcelly 的一个周五晚上，正值促销期间。Notifications 服务数月来一直在发送“订单已确认”邮件，从未收到过投诉。随着积压增长，自动扩缩容增加了消费者，几分钟后又减少了一些。到晚上结束时，客服收到两张工单。一位买家收到了三封相同的确认邮件。另一位一封也没有收到，尽管订单已经确认。

团队打开 broker 仪表盘，发现没有任何东西坏掉。每个 offset 都已提交，每个消费者都运行正常，曲线一片平直。然而同一个系统，对一位买家产生了重复消息，对另一位买家却一片沉默，而这两种结果都符合文档的描述。

有人说 broker 承诺的是“at-least-once 投递”，另一个人说他们原本以为 broker 会处理这件事。两种说法都没错，但都解释不了两位买家身上发生了什么。假定 Notifications 以一个 consumer group 的身份读取基于日志的 topic，其客户端自动提交 offset，并且每当该 group 的规模变化时就会发生一次 rebalance。这两个事实足以同时造成这两张工单，本章会说明其中的过程。

要解释这一点，团队需要一幅精确的图景：broker 承诺了什么，没有承诺什么，消息生命周期的哪些部分属于应用程序。从生产者到最终完成的副作用，一条消息究竟经历了什么，每一项保证又从哪里开始、到哪里结束？

问题及其难点

多数工程师接触 broker 时，听到的是最简单的描述：发布一条消息，消费者收到它。这个描述掩盖了三次相互独立的交接，而本书中的每个可靠性问题都出在其中某一次交接上。

第一次交接是生产者到 broker。第二次是 broker 到消费者。第三次是消费者到效果：数据库写入、邮件、对承运商的调用。broker 对前两次交接能说很多，对第三次几乎无话可说，因为第三次发生在 broker 看不见的代码和系统之中。

难点在于，每次交接都以一个 ack 结束，而 ack 是一方通过网络对另一方作出的、关于过去的陈述。它表示“我收到了这个”或“我完成了这个”，并不表示“其他任何地方都不会再出错”。当 ack 丢失时，发送方无法判断工作是否已经发生。这种不确定性正是三种投递语义的根源，掌握了相关词汇之后，我们会回到这一点。

第二个难点是，“broker”一词涵盖两个族，二者的行为差异在要紧之处各不相同。**队列**是 broker 中的一种结构，每条消息投递给一个消费者，并被单独 ack。**事件日志**是持久的、append-only 的事件序列，消费者按各自的节奏读取并记录一个位置。对其中一族成立的论断，往往对另一族不成立。每当这一区别要紧时，本书都会指明所说的是哪一族，而本章正是建立这一区别的地方。

第三个难点是，产品文档中的保证是有范围的。“至少投递一次”是关于 broker 自身边界的陈述。它不涵盖消费者收到消息之后做了什么，并且取决于生产者发布时所选的设置。读保证而不看它的范围，团队就会落入上面那个周五晚上的境地。

队列与日志

队列模型

在队列中，broker 跟踪每条消息的状态。一条消息要么在等待，要么已投递给某个消费者，要么已被 ack 而消失。多个消费者可以从同一个队列读取；这种安排称为**竞争消费者**，因为每条消息每次只投递给其中一个消费者，它们彼此竞争下一条消息。

消费者对消息发送 ack 时，broker 就删除它。因此保留期限在设计上就很短：消息一直存在，直到被处理，或直到过期。没有内置的办法可以向队列索取一条已被 ack 的消息，因为队列并不保存它。

其结果是，队列适合那些做一次就可以忘掉的工作。Parcelly 中的“渲染这张发票 PDF”这类作业就是天然的例子：完成工作，消息就再无用处。代价是无法获得历史。如果某个 bug 在一小时内发出了错误的文本，承载事实的消息已经不在了，修复必须从别处着手。

日志模型

在事件日志中，broker 不跟踪每条消息的状态。它按顺序把消息存储在分区中，并依据一项保留策略保存它们，该策略与谁读过无关。**分区**是 topic 的一个有序子序列，**topic** 是消息的一条具名 stream。每条消息

都有一个 **offset**，即它在分区中的位置。

消费者各自记录读到了哪里。这个位置同样称为 offset，由消费者自己记录，或由 broker 代为记录。**consumer group** 是一组分担同一个 topic 工作的消费者：每个分区每次只分配给 group 内的一个消费者。不同的 group 相互独立地读取同一个 topic，各有各的 offset。

日志的决定性特征是，读取并不消耗消息。十个 group 可以读取同一个 topic，新的 group 也可以从头开始。**重放**，即有意地重新读取历史，是本书专门用于这一行为的术语，它之所以可能，只是因为日志保留了数据。Parcelly 的 Ledger 就是受益的消费者之一：如果发现构建分录的方式有 bug，可以修复 Ledger 服务，再让它回头读取旧事件。

代价是，日志把记账的负担推给了消费者一侧。在经典的日志模型中，broker 无法在继续处理下一条消息的同时重新投递某一条失败的消息，因为它并不逐条跟踪消息。一个无法处理 offset 500 处消息的消费者，必须先决定怎么办，才能安全地前进到 501。

如何选择

关注点	队列族	日志族
跟踪单位	每条消息	每个分区一个位置
保留	直到被 ack 或过期	按时间或大小，与读取方无关
多个独立读取方	每个读取方需要一份消息副本	天然支持，每个 group 一个位置
历史重放	不可用	在保留期限内可用
一条失败的消息	单独重新投递，其余继续	除非处理，否则阻塞分区
消息顺序	最好情况下是存储中的 FIFO；跨竞争消费者和重新投递后即丢失	在分区内是强顺序

我不会按工具的流行程度来选择某一族。我会从三个问题出发：是否有人需要再次读取这段历史，是否有多个消费者需要同样的事实，同一个聚合的消息顺序是否要紧？任何一个问题的答案为“是”，都指向日志。纯粹的工作队列，不需要历史，而需要对单条消息独立 retry，则指向队列。

许多团队最终两者都运行。其他服务据以构建状态的事件放在日志上，只应发生一次的任务放在队列上。这是一种合理的设计，代价是要运维并理解两套语义。第 9 章会回到消息顺序在每一族中的含义。

Push、pull 与投递流程

broker 的区别还在于由谁发起投递。在 **push** 模型中，broker 在消息可用时就把它们发送给消费者。在 **pull** 模型中，消费者准备好时才去请求消息。

基于队列的 broker 通常采用 push（RabbitMQ 向消费者投递，而 Amazon SQS 的消费者则轮询，所以即便在同一族之内也各不相同），并限制一个消费者同一时刻最多持有多少条未 ack 的消息。这个限制通常称为 prefetch 数量或窗口。基于日志的 broker 通常采用 pull：消费者从当前 offset 起请求下一批。两种模型在两族中都能见到，所以我会把这当作一种倾向，而不是定律。

这一差异主要在消费者变慢时才要紧。在没有限制的 push 模型中，broker 持续发送，消费者的内存被尚未开始处理的消息填满，这些消息处于一种既不能被其他消费者获得、也没有完成的状态。在 pull 模型中，消费者控制自己的节奏，慢消费者表现为 consumer lag 不断增长，而不是进程过载。**consumer lag** 是消费者落后于分区或队列头部的程度，以消息数或时间计。

实用的规则是：消费者持有的工作量，永远不应超过它在对这些工作的占有到期之前所能完成的量。第 3 章把慢消费者和反压本身当作故障来处理，第 6 章讲如何限制负载。眼下需要记住的是，“已投递”与“正在处理”是不同的状态，两者之间的间隙正是时间流逝之处。

Ack、重新投递与 offset

队列中的 ack

acknowledgment，通常简称为 **ack**，是消费者所作的陈述：它已经处理完一条消息，broker 可以忘掉它。**nack**，即否定的 acknowledgment，表示消费者无法处理这条消息。取决于 broker 和设置，nack 会把消息放回队列、丢弃它，或把它送入死信队列（DLQ）。

关键的设计问题是何时发送 ack。合理的位置只有两个。

- **处理前 ack**。broker 立即忘掉这条消息。如果消费者在工作中途崩溃，消息就丢失了。这产生 at-most-once 行为。
- **处理后 ack**。broker 持有这条消息，直到工作完成。如果消费者在完成工作之后、发送 ack 之前崩溃，消息会回来，工作被做两次。这产生 at-least-once 行为。

没有哪个位置能同时给出两种属性，因为 ack 与工作是两个独立系统上的两个独立动作，崩溃可能恰好落在二者之间。

Visibility timeout 与重新投递

队列如何判断某个消费者已经崩溃？它无法直接观察到这一点。有些 broker 监视连接，把连接关闭视为故障。另一些使用 **visibility timeout**：消息投递出去后，在一段设定的时间内对其他消费者不可见。如果消费者在时间用完之前发送了 ack，消息就被删除。否则，消息重新变为可见，并被投递给别的消费者。这第二次投递就是一次**重新投递**。

这个超时是对处理耗时的一次猜测，而两种猜错的方式都有代价。如果设得太短，一个缓慢但仍然存活的消费者，的消息会在它还在工作时就被交给第二个消费者。此时两个消费者在处理同一条消息，第一个消费者最终发出的 ack 可能失败、被忽略，或者视 broker 而定，成功并删除消息，而第二个消费者还在处理它。在最后这种情况下，第二次尝试的失败会使消息丢失。如果设得太长，真正的崩溃会使消息在整个时段内保持隐藏，这个延迟会以延迟的形式出现在买家面前。

假设 Parcelly 的 Inventory 消费者通常在远不到一秒的时间内完成一次库存预留，而有人把超时设成了 30 秒。这个数字只是示意。一次数据库抖动使某次预留耗时 40 秒，就会导致同一个 `OrderPlaced` 被第二次投递，Inventory 将尝试两次预留库存。broker 里没有任何东西坏掉。超时编码了一个假定，而这个假定不再成立。

有些系统允许消费者在工作期间延长超时。这对确实耗时很长的作业有帮助，同时又增加了一个新的出错点：消费者对一条永远也处理不完的消息无休止地延长超时。

日志中的 offset

在日志中，没有逐条消息的 ack。消费者转而执行 **offset 提交**：它记录“此位置之前的一切我都已处理”。重启或重新分配之后，下一个消费者从已提交的 offset 开始读取。

这里同样存在那两个位置。先提交 offset 再处理，崩溃会丢失最后一条已完成消息之后、直到已提交位置的那些消息。先处理再提交，崩溃会重复最后一次提交与崩溃之间的那些消息。真实的消费者按批次提交，所以崩溃之后重复的消息数量是一个批次，而不是一条。

许多客户端库可以自动提交。这很方便，但也是设定投递语义时最不明确的方式，因为视客户端而定，提交可能发生在与处理完成时间无关的某个时刻。如果处理被交给了另一个线程，客户端就可能提交从未完成的工作。在我看来，重要的消费者应当显式提交，且提交点应当是团队能够在代码中指出并据以推理的位置。

Consumer group 与 rebalance

当消费者加入或离开一个 consumer group 时，broker 会在它们之间重新分配分区。这称为 **rebalance**。在 rebalance 期间，原本由一个消费者读取的分区被重新分配给另一个消费者，新的持有者从最后一次已提交的 offset 开始。

基于日志的 group 以另一种形式，作出了与队列中 visibility timeout 相同的猜测。在配置的时间间隔内没有回到 broker 的消费者被推定已死，其分区被重新分配，它未提交的工作由新的持有者再次读取。一个仍然存活的慢消费者，是 rebalance 重复消息的日常成因。

开头那个场景中的重复消息就来源于此。设想一个消费者已处理了消息 100 到 150，但只提交到 120。一次部署重启了它，分区转移给第二个消费者，后者从 120 开始读取。消息 120 到 150 被处理了两次，而系统的任何部分都没有发生故障。rebalance 是正常事件，每次部署、扩容和崩溃都可能发生，所以重新投递是日常现象，而不是罕见的意外。

那一片沉默的成因与此互为镜像。如果客户端为消费者仅仅交给后台发送器的消息提交了 offset，而进程在发送器完成之前就被停止，下一个持有者就会从这些消息之后开始。它们永远不会再被读取，也没有任何一行日志说明这一点。因此，同一次 rebalance 既可能重复一位买家的消息，又可能丢掉另一位买家的消息。

还有一个相关的隐患：在很短的时间内，旧的持有者可能尚不知道自己已经失去了分区，仍在处理它。这时两个消费者可能同时处理同一个分区。设计良好的客户端会设法防止这种情况，但我不会设计一个依赖于这种防止机制完美无缺的消费者。

投递语义

有了这些机制作为基础，三种投递语义就是 ack 放置位置的后果。

At-most-once

at-most-once 表示每条消息被投递零次或一次。做法是处理前 ack，或对失败的发布不做 retry。消息可能丢失，但绝不会重复。

当丢失一条消息的代价低于处理重复消息的代价时，这是合理的选择。Parcelly 承运商周期性发出的“当前位置”ping 就是一例：下一次 ping 会使丢失的那一次变得无关紧要。对于 `PaymentCaptured`，这就是错误的选择，因为消息丢失意味着钱已经转移，却没有人记录。

At-least-once

at-least-once 表示每条消息被投递一次或多次。做法是处理后 ack，并不断 retry 直到收到 ack。只要 broker 还持有消息，消息就不会丢失，但它可能被再次投递。

本书建立在这种语义之上。三者之中，只有它的故障是可见且可恢复的：重复消息是可以检测并忽略的东西，而丢失的消息可能在数天之内都不会被发现。代价是每个消费者都必须容忍**重复消息**，即被处理或被投递了不止一次的消息。第 7 章讲的就是如何做到。

“Exactly-once”

投递是穿越不可靠网络的一个物理行为，接收方的 ack 可能像其他任何消息一样丢失。没有收到 ack 的发送方有两个选择：再发一次，有重复的风险；或者不发，有丢失的风险。仅凭双方，没有任何协议能消除这一选择。

然而 exactly-once 确实是一种真实而有用的结果，它是一种效果，而不是投递保证。当 at-least-once 投递遇上一个重复处理与处理一次效果相同的消费者时，组合起来的系统表现得就像每条消息都恰好被处理了一次。其配方是 at-least-once 投递加上**幂等消费者**。

有些 broker 宣称提供 exactly-once 处理，这个说法在一个边界之内是成立的。当输入、输出和 offset 提交都位于同一个 broker 内部时，基于日志的 broker 可以使“读取-处理-写入”循环成为原子操作。在这个边界之内，崩溃会让所有操作一起回滚。这项保证还要求每个下游读取方只读取已提交的数据，并依赖稳定的生产者身份，使旧实例在重启后被隔离在外。边界终止于副作用开始之处。消费者一旦写入关系数据库、调用支付服务商或发送邮件，broker 的事务就不再覆盖该效果，重复消息的问题随之回归。

因此，本书的默认做法仍然是 at-least-once 加幂等消费者。它适用于任何 broker 和任何副作用，不依赖于某项功能被启用，并且能在迁移到另一款产品时保持有效，第 14 章会讲到这种迁移。如果能把整条流水线保持在一个 broker 的事务边界之内，这项功能或许能减少工作量，而我仍然会把消费者设计成不依赖于它。

> **注意**：生产者也可能在消费者介入之前就制造重复消息。一个等待发布 ack 而超时的生产者通常会再发布一次，此时 broker 可能持有同一事件的两份副本。只要生产者在事件创建时只分配了一次 `event_id`，而没有为 retry 生成新的，两份副本就携带相同的 `event_id`。有些 broker 会对生产者自身的 retry 去重，依据是生产者会话内的序列号，或某个时间窗口内的去重 ID。这覆盖了同一个连接内的 retry，但不覆盖重启后再次发布的进程，例如 outbox relay，这正是消费者仍然要检查 `event_id` 的原因。第 4 章讲如何在发布时不丢失事件，第 7 章讲如何识别重复消息。

Broker 实际保证什么

上面的一切描述的都是 broker 到消费者这一侧。生产者到 broker 这一侧有它自己的限制，而这些限制取决于配置，多于取决于产品名称。

生产者 ack

生产者发布消息时，broker 可以在不同的时刻发送 ack，区分这些设置的有效办法，是看生产者听到“完成”时有多少个节点持有这条消息。没有 ack 时，生产者对存储情况一无所知。由接收节点发送 ack 时，有一个节点接受了这条消息，无论它是否已刷盘。复制之后发送 ack 时，消息已存在于要求数量的节点上。刷盘是另一个单独的问题，见下文。

生产者的选择，决定了崩溃能造成什么后果。对前两种设置而言，刚发出 ack 就死掉的 broker 节点可能把消息一并带走。对复制后 ack 而言，答案来得更晚，丢失的窗口也更小。代价是每次发布的延迟更高，并且依赖于有足够多的节点可用来完成写入。

提供 publisher confirm 的 broker（属于队列族），与提供 ack 级别的 broker（属于日志族），描述的是同一个思路。需要查明，在具体配置中所等待的 ack 实际意味着什么。在我看来，这是团队值得读一遍并写下来的最有价值的文档页面之一。

复制与 fsync

复制能防止单个节点的丢失：消息存在于多个节点上。它不能防止一切同时失效，其保证取决于在生产者被告知“完成”之前，必须存在多少份副本。需要同时检查 ack 级别和它所要求的最小副本数，因为当其他副本已经落后时，“所有副本”可能只意味着一个。还要检查故障转移是否可能提升一个并不是最新的副本，因为这可能丢失生产者已被告知安全的数据。

关于磁盘还有一点相关且常被误解的内容。写文件通常只是把数据放入操作系统的缓存，数据之后才到达物理磁盘，除非程序用同步刷盘（通常称为 fsync）强制刷盘。对每条消息都强制刷盘很慢，所以许多 broker 不会对每次写入都这样做。它们依靠复制来弥补这段间隙，并按计划刷盘。某个 broker 的 ack 是否意味着数据已落盘，取决于它的配置，我不会擅自假定。

我不知道这些设置中哪一种适合某个具体的系统，这取决于丢失一个事件的代价。丢失 `PaymentCaptured` 与丢失一次“页面被浏览”是不同的问题，将二者配置得不同是合理的。

保证的界限

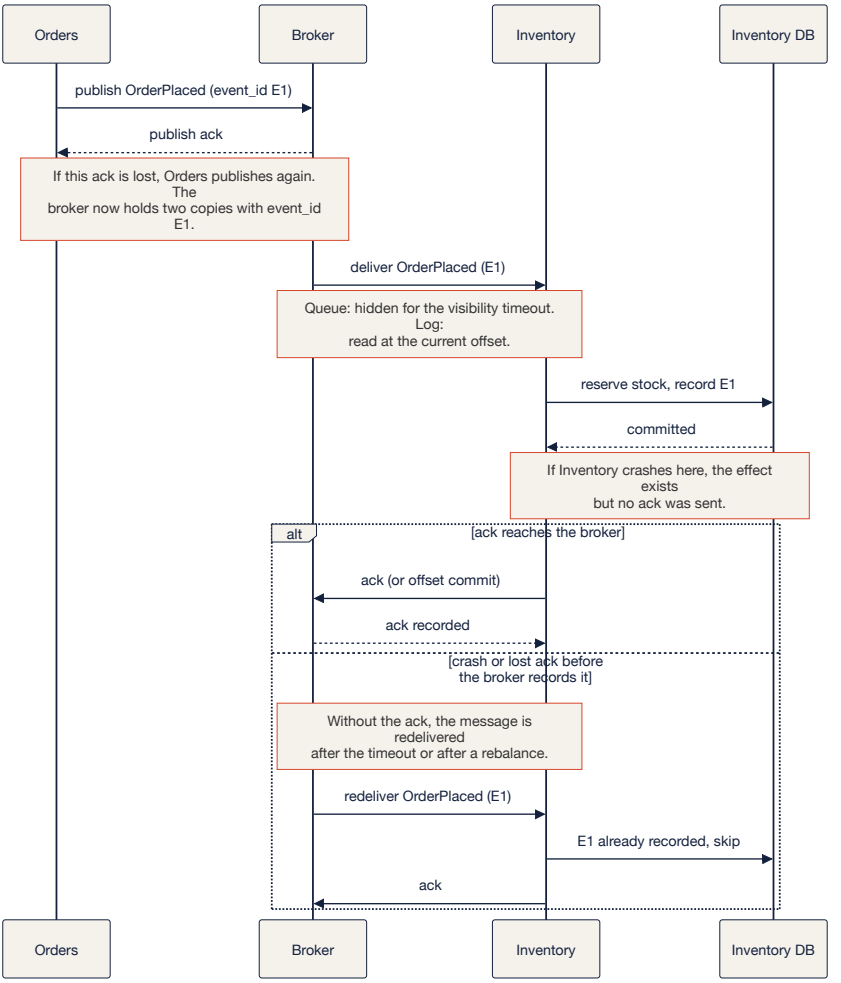
把这些合在一起，边界就清楚了。broker 能够承诺：它所接受的消息会被保存并提供给消费者，在其保留或过期限制之内，且其持久性仅达到上述 ack 与复制设置所允许的程度。它无法承诺消息一开始就能到达 broker，因为那取决于生产者在发布超时时做了什么。它也无法承诺消费者的效果恰好发生了一次。

有两处间隙留在应用程序手中。第一处在生产者一侧：保存在 Orders 数据库中的订单，与发布到 broker 的 `OrderPlaced` 事件，是两次写入，崩溃可能恰好落在二者之间。第 1 章把这称为双写，第 4 章会消除它。第二处在消费者一侧，位于效果与 ack 之间。第 7 章会弥合这一间隙。以本章开头的三次交接来看，broker 为前两次声明了故障语义，而剩下握在应用程序手中的就是这两处间隙：第一次交接之前的双写，以及第三次交接中效果与 ack 之间的间隙。

消息的一生

下面的图跟随一条 `OrderPlaced` 消息，从 Orders 服务一直到 Inventory 中完成的效果。注意其中两处故障使发送方得不到答复的位置，它们已在注释中标出：丢失的发布 ack 和丢失的消费者 ack。本章描述的一切，

都发生在这一个序列之内。



图中的 `else` 分支正是本章的要点。重新投递不是错误，也没有任何组件行为失常。唯一能避免库存被预留两次的，是 Inventory 把 `E1` 与预留记录在同一个事务中，并在第二次投递时再次检查它。

下面用伪代码展示具备这一属性的消费者是什么样子。它使用 **inbox**，即消费者自己数据库中一张记录已处理 `event_id` 值的表，并把 ack 放在提交之后。

```
function handleOrderPlaced(message):
    // a redelivery of work that already committed is acknowledged,
not redone
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique constraint on event_id makes a concurrent
duplicate
        // fail here, roll back, and come back as a redelivery that
the
        // check above then acknowledges
        inboxInsert(message.event_id)
    // transaction commits here, or rolls back on any error

    // ack comes last: a crash before this line causes a redelivery,
    // which the inbox check turns into an ack rather than a second
reservation
    ack(message)
```

这正是使 at-least-once 变得安全的设计，在此引入只是为了说明 ack 应当放在哪里。第 7 章会把它完整展开，包括仅靠 inbox 检查还不够的那些情形。

权衡与成本

选择语义，就是选择更愿意管理哪一种故障。没有任何选项能消除故障，每个选项都只是把它移到别处。

at-least-once 使每个消费者都要做幂等性的工作。每个消费者都需要一种识别重复工作的办法，通常是它自己数据库中的一张 inbox 表。这意味着存储、热路径上的唯一约束以及清理策略。对于有十几个消费者的团队，这是一个要应用十几次的模式。当消费者的效果天然幂等时，例如把状态设置为固定值，inbox 可能没有必要，而我仍然会把理由写下来。

at-most-once 使完整性丧失。它简单而快速，只有在有人事先决定丢失消息比重复消息代价更低时，才可以接受。这一决定应当是明确的，并且有记录。这类错误的常见形态是，团队通过自动 ack 设置默认选择了它，却没有意识到。

broker 内部的事务性处理使可移植性丧失。它可以减少其边界之内的重复处理，同时它把设计绑定在某一款产品的功能上，要求每个参与方都留在该边界之内，并带来运行它的运维工作量。一旦跨到关系数据库或外部

API，无论如何都需要幂等性。我不会拒绝这项功能，也不会仅仅在它之上构建正确性。

强生产者 ack 会消耗延迟和可用性。等待多个副本，会使每次发布变慢，并在可达节点太少时使写入失败。对于不容丢失的事件，这种权衡通常是对的。对于高吞吐、低价值的遥测数据，则未必。

日志要求承担消费者侧的记账，队列则没有历史。选择日志，就要自己负责 offset 提交策略，以及处理一条阻塞分区的消息。选择队列，就放弃了重放。当需要对单条消息独立 retry、又不需要历史时，日志对于这项工作来说，可能是过重的机制。

本章建议不适用的情形：只有一个生产者、一个消费者，且能容忍偶尔丢失的系统，例如内部指标数据流，既不需要 inbox，也不需要强复制。保证的成本应当与故障的成本相匹配。错误不在于选择了较弱的保证；错误在于不知道自己选择了较弱的保证。

常见错误

- **把“已投递”当作“已处理”**。这很诱人，因为仪表盘显示的是投递和 ack，而这些正是看起来健康的数字。已投递意味着 broker 把消息交了出去。已处理意味着效果已提交。两者都要跟踪，并让 ack 跟在效果之后。
- **在工作完成之前 ack**。这很诱人，因为它避免了重新投递，让队列看起来很干净。它把每一次消费者崩溃都变成一条丢失的消息。应当在效果提交之后 ack，并处理由此产生的重复消息。
- **默认开着 offset 自动提交**。这很诱人，因为它在演示中能跑，而且不需要代码。视客户端而定，它可能在与处理无关的时刻提交，使崩溃丢失或重复任意一段消息。应当在效果之后显式提交。
- **凭感觉设置 visibility timeout**。这很诱人，因为默认值通常看起来没问题。比一次缓慢但健康的运行还短的超时，会造成并发的重复处理。应当以能观察到的最慢合法处理时间加上余量为依据，并对长作业有意识地延长。
- **相信 broker 的 exactly-once 功能覆盖了自己的数据库**。这很诱人，因为文档用了这个说法。该保证的范围限于 broker 自己的边界。边界之外的任何东西，例如数据库中的一行或一封邮件，都需要幂等性。

- **以为已发布的消息就是已存储的消息**。这很诱人，因为发布调用返回了。视 ack 级别而定，消息可能只存在于某一个节点的内存中。要清楚生产者等待的是什么，并把它写下来。
- **以为 rebalance 很少发生**。这很诱人，因为它悄然发生。部署、扩缩容事件和崩溃都可能触发一次，视 group 协议和成员设置而定，每次 rebalance 都可能重复一批消息。应当按重新投递每天都会发生来设计，因为判断错误的代价是重复的效果。

要点

- 一条消息要经过三次交接：生产者到 broker、broker 到消费者、消费者到效果，而 broker 的保证只覆盖前两次。
- 队列跟踪每条消息，并在 ack 时删除它，而日志独立于读取方保留消息，并让每个 consumer group 保存各自的 offset，这正是重放得以实现的原因。
- ack 或 offset 提交放在哪里，决定了语义：处理前给出 at-most-once，处理后给出 at-least-once，没有任何放置方式能同时避免丢失和重复。
- 重新投递是常规现象，而不是例外，因为 visibility timeout 会到期，消费者会崩溃，rebalance（常由部署引起）会重复批次。
- exactly-once 是由 at-least-once 投递和幂等消费者构成的一种效果，而 broker 自己的 exactly-once 处理止于该 broker 的边界。
- 生产者 ack 的强度，只取决于其背后的复制和刷盘设置，因此团队应当了解并记录，对每个重要事件而言“已发布”意味着什么。
- 每个语义选项都是转移故障而不是消除故障，正确的选择是那个已确定自己有能力承担其剩余故障的选项。

消息传递语义说明，在正常的一天里 broker 会做什么、不会做什么。第 3 章转向不正常的日子，并构建一个实用的模型，说明究竟什么会出问题：网络分区、崩溃、慢消费者、时钟和人。

第 3 章 — 务实的故障模型

设想 Parcelly 的 Payments 服务正处在一次扣款的中途。Orders 已向 Payments 发送了 `CapturePayment` 命令，Payments 则调用外部支付服务商去扣款。与服务商的连接陷入沉默。三十秒后，Payments 的 HTTP 客户端放弃等待，抛出超时。

Payments 现在必须回答一个它无法回答的问题。服务商是在连接断开之前已经扣了款，还是从未收到请求？两种经过产生的观察结果完全相同：沉默，然后超时。如果 Payments 执行 retry，而第一次扣款其实已经成功，买家就被扣了两次钱。如果 Payments 报告失败，而第一次扣款其实已经成功，Orders 就会取消一个已经付过款的订单。

这个故事里没有任何离奇之处。没有 bug，没有事故报告，没有哪位工程师犯了错误。网络做了网络该做的事，而系统无从分辨自己身处两个世界中的哪一个。

事件驱动系统中的大多数问题都是这个形状：某一步发生了，或没有发生，而证据缺失。本章为证据缺失的各种方式命名，以便能有意识地针对每一种做设计。它要回答的问题是：在 Parcelly 这样的系统中，究竟什么会出问题，出问题时会看到什么，以及每种故障在本书的哪一处得到处理？

问题及其难点

故障模型是一份书面清单，列出假定可能出错的事情，以及假定不会出错的事情。每个系统都有一份，无论有没有人把它写下来。没有写下来的版本，通常只包含两条：“服务崩溃”和“网络很慢”。这两条是真实的，但它们是会自己宣告存在的故障。

简单的情形是可以检测到的故障，例如连接被拒绝或数据库报错。每一种都表明这一步没有完成。

困难的情形是让人无法确定的故障。超时并不是在说“操作失败了”，而是在说“我不再等了”。操作可能失败了，可能成功了，也可能仍在运行，并将在调用方放弃之后成功。只需要一个超时，加上另一端的一个副

作用，就能体会到这一点。

这正是显而易见的做法行不通的原因：为快乐路径编写代码，再在每次调用外面加上 `try` 和 retry。retry 是在不了解过去的情况下对未来作出的决定。如果先前的尝试成功了，retry 就是重复。如果拒绝 retry，消息就可能丢失。两种选择本身都没有错；每一种都在某个无法与当前世界区分开的世界里是错的。

还有第二个原因。故障会组合出现：一次部署重启了某个消费者，同时 broker 发生故障转移，生产者又在 retry。逐个列出它们仍然是正确的起点，但生产环境是把它们一起送来的。

所以本章的工作是分类。对每一种故障，我都会问第 1 章中的三个问题：这个接缝处什么会失败，如何观察到它，哪项保证在其中得以保全。在下面的条目中，“实际情况”回答第一个问题，“表现”回答第二个问题，而每个条目末尾点名的模式回答第三个问题。这三个答案合在一起，就是一种故障的语义，本书其余部分不断索取的正是它们。

十种要紧的故障

下面的清单，是我会为一个由服务、关系数据库和消息 broker 构成的系统所选的起点。它并不完整，也没有哪份清单是完整的。

网络分区与超时的不确定性

开头的场景是典型的例子。两方由一个网络连接，而网络丢弃、延迟或重排流量。**网络分区**是其极端形式，即两组机器完全无法互相到达。超时是日常形式，而且常见得多。

双方能观察到的是不对称的。调用方只看到没有答复到达。被调用方可能已做完了工作却丢了回复，也可能从未见过请求。Parcelly 中的每一跳都是如此：服务调用外部承运商，生产者向 broker 发送，消费者向 broker 发送 ack。

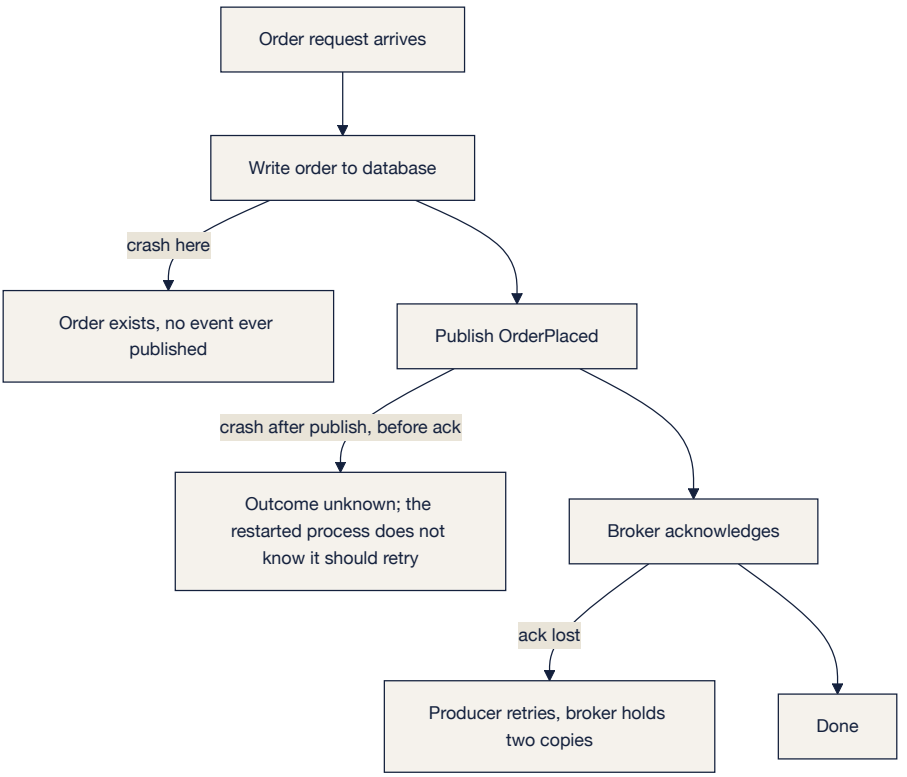
对此唯一诚实的应对，是让操作可以安全地重复。如果 Payments 在每次扣款时向服务商发送一个稳定的幂等键，而服务商把重复的键视为同一个请求，那么超时之后的 retry 就变得无害。设计不再依赖于“它发生了吗？”的答案，因为两个答案通向同一个最终状态。第 6 章讲何时以及如何 retry，第 7 章讲使 retry 安全的幂等性。

有些外部系统不提供这样的键。这时要在再次行动之前先询问该系统发生了什么，这是一个对账步骤，第 13 章会讲到。

生产者崩溃窗口

设想 Orders 处理一个新订单。它必须把订单保存到自己的数据库中，并把 `OrderPlaced` 发布到 broker。这是两个系统，没有任何事务横跨它们。这就是第 1 章的双写，有意思的地方在于每个崩溃窗口会留下什么。

进程可能在两次写入之间崩溃，而写入的先后顺序决定了会得到哪一种损害。



该图画的是先写数据库的顺序，并展示了三个窗口：发布之前崩溃，发布之后崩溃而结果未知，以及 ack 丢失。如果先提交数据库，进程在发布之前死掉，订单就存在，下游却没有人听说过它。Payments 从不预授权，Inventory 从不预留，买家等着一封永远不会到来的确认。没有任何报错。订单就那样躺在那里。

图中没有画出相反的顺序，它有自己的窗口。如果先发布，在提交之前崩溃会留下更糟的状态：一个事件宣告了一个从未成为事实的事实。下游服务会对一个不存在的订单采取行动。而且无论哪种顺序，只要 broker 的 ack 在返回途中丢失，生产者就会 retry，broker 就可能持有同一事件的两份副本。

从外部看到的是安静的：有订单而没有事件，有事件而没有订单，偶尔还有重复。没有任何日志写着“丢失”。第 4 章引入 transactional outbox，它通过让状态变更与发布意图构成同一个本地事务，来消除这个窗口。第 5 章讲 outbox 的行随后如何前往 broker。

Broker 故障转移

broker 是复制的，这样一台机器死掉时不会带走数据。当持有某个分区或队列 leader 副本的机器死掉时，另一台接管。这就是一次**故障转移**，它是一种被设计出来的行为，而不是故障。

要紧的是生产者在故障转移之前被告知了什么。在 leader 死掉前一刻刚被 ack 的消息能否保留下来，取决于 broker 被配置为如何发送 ack：完全不发送，接收节点接受之后发送（无论是否已刷盘），或要求数量的副本持有之后发送。最强的设置要付出延迟和吞吐量的代价。最弱的设置在故障转移中可能丢失已被 ack 的消息。这在不同的 broker 族之间、不同的配置之间各不相同，所以我不会在这里就某个具体产品声称某种具体行为；要点是保证有边界，而边界是某个人选定的一项设置。

故障转移期间，生产者还会看到错误和超时，执行 retry，并可能重复发布旧 leader 已经接受的消息。第 2 章描述了 broker 保证止于何处。这里的规则是，把“已被 broker ack”视为有力的证据，而不是证明。

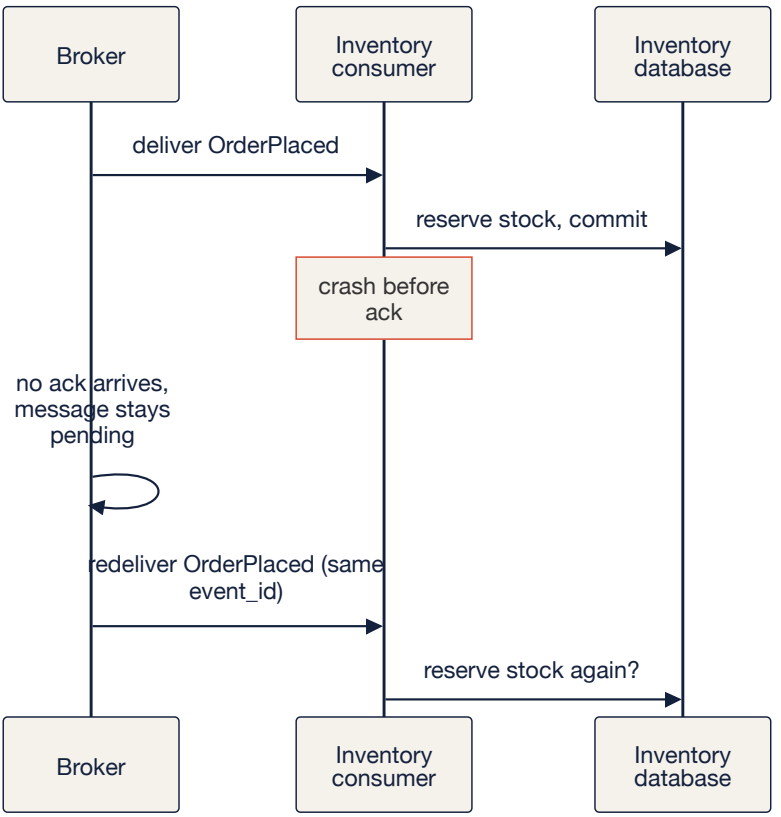
表现：一阵发布错误，消费者吞吐量下降，然后是大量重新投递。如果某条消息丢失了，什么也看不到，这正是第 13 章的对账之所以要紧的原因。

消费者崩溃：已处理但未 ack

现在看消费者一侧。Inventory 收到 `OrderPlaced`，在自己的数据库中预留库存并提交。然后进程在能够对消息发送 ack 之前死掉。

在 broker 看来，这条消息从未完成。在基于队列的 broker 上，broker 发现连接关闭，或 visibility timeout 到期，消息被重新投递。在基于日志的 broker 上，消费者的 offset 从未提交，所以下一个拥有该分区的

消费者从旧位置开始，再次读取这条消息。在两族中，第二次投递针对的都是已经完成的工作。



图的最后一行就是全部问题。每次投递都无条件预留库存的消费者，会预留两次。这是重复消息的普通来源，也是本书默认采用 at-least-once 投递加幂等消费者的原因。消费者可能在不同的点崩溃，每个点的后果各不相同：

```
function handleOrderPlaced(message):
    // crash point A: before any work. Redelivery starts clean.
    if inboxContains(message.event_id):
        ack(message)
        return

    transaction:
        reserveStock(message.payload.order_id, message.payload.lines)
        // the unique event_id makes a concurrent duplicate fail here
        inboxInsert(message.event_id)
    // transaction commits here
    // crash point B: committed, not acked. Redelivery hits the inbox
check
    // above and is acknowledged without repeating the work.

    ack(message)
    // crash point C: after ack. Nothing to repair.
```

这是第 2 章中处理器的精简版，删减后用来展示崩溃点。崩溃点 A 的后果是没有后果。崩溃点 C 的后果也是没有后果。有意思的是 B，只有把已处理的内容与工作记录在同一个事务中的消费者，才能挺过它。第 7 章会正式构建它。

表现：重复的副作用，例如两次预留、两封邮件、两条账本分录，且任何地方都没有报错，因为每次处理尝试都成功了。

慢消费者与反压

并非所有故障都是崩溃。仅仅是慢的消费者就能造成同样大的损害，而且由于没有任何东西坏掉，更难被注意到。

假设 Notifications 调用的邮件服务商，每次请求从 100 毫秒变成了 10 秒。Notifications 仍在工作，但每条消息现在占用一个 worker 的时间长了 100 倍。消息到达的速度快于完成的速度，所以第 2 章定义的 consumer lag 持续增长。

慢随后会引发次生故障。在基于队列的 broker 上，超过 visibility timeout 的消息会在第一个 worker 仍在处理它时重新出现，于是第二个 worker 开始处理同一条消息。重复消息给一个本来就很慢的系统增加了负载。在基于日志的 broker 上，没有及时回到 broker 的消费者可能被宣告死亡，其分区被重新分配，这就启动了本清单中的下一种故障。

反压是一种约束，即限制每个阶段的在途工作量，使负载保持有限。在以 broker 为中介的系统中，broker 吸收两端的差异，这正是时间解耦的好处，但缓冲区是有限的，它会以两种不同的方式耗尽。当 broker 的磁盘

写满时，它可能对生产者限流或阻塞，或完全停止服务，所用的 broker 属于哪一种，需要在这一天到来之前弄清楚。无论是哪一种，生产者都能看到，而那是真正的反压：响亮而可见。当消费者落后的程度超过日志的保留期限，或排队的消息超过其过期时间时，broker 会删除从未被读取的消息。那不是延迟。那是静默的丢失：没有报错，没有消费者失败，未读取的 `PaymentCaptured` 事件就这样不见了。在到达这两种限制之前很久，积压就可能已经违反了某个业务期限，例如一笔支付预授权在被扣款之前就已到期。

由于 broker 的缓冲区只是推迟问题，负载必须同时在多个位置受到限制。消费者可以限制在途消息的数量，或它预取的批次。它可以限制 worker 并发数，使慢速的邮件服务商只占用固定数量的 worker，而不是全部。它可以对发往慢速依赖的调用限速，或在依赖恢复期间完全暂停消费。队列可以有长度上限和明确的溢出策略，这样被丢弃或被拒绝的是什么由消费者一方选择，而不是由 broker 选择。

生产一侧和拓扑结构也设定了限制。在生产一侧，outbox 吸收了拒绝发布的 broker，因为这些行会在数据库中等待，直到 relay 能够投递它们。而在基于日志的 broker 上，分区数量限定了一个 group 中能并行工作的消费者数量，所以增加实例的作用在一个预先固定的数字处就不再有效。这些限制中的每一个都以吞吐量换取稳定性，每一个都需要一项指标，否则无从得知是它在拖住系统。

表现：lag 和消息年龄上升，积压增长，重新投递次数攀升。第 6 章讲退避、在途限制和暂停消费，第 8 章讲无法及时处理的消息该怎么办，第 13 章讲能及早预警的 lag 和年龄指标。

每次都失败的消息，与对每条消息都失败的依赖

到目前为止，每种故障都是偶然的：一次崩溃，一个丢失的回复，一个糟糕的时刻。有些故障是确定性的。**poison message** 是每次尝试都失败的消息，与时机无关。设想一条 `OrderPlaced` 到达 Inventory，其中的 SKU 在它的目录中不存在。Inventory 抛出错误，消息被重新投递，错误重复出现，下一次尝试与上一次没有任何不同。

原因也可能在消息之外。如果 Payments 的服务商宕机，每次扣款都会失败，只要故障持续，每条消息看起来都有毒。这两种情况要求相反的应对。一条永远不会成功的消息应当迅速离开主流程，进入**死信队列（DLQ）**，这是一个独立的地方，无法处理的消息在那里等待人工或修

复作业。因依赖宕机而失败的消息则根本不应转入死信队列，因为它在恢复之后会成功，而在故障期间把整个积压转入死信队列，会把一个暂时的问题变成人工问题。

这中间出错的方式有一定规律。对 poison message 执行 retry，会消耗尝试次数和容量，却没有任何成功的机会。在基于日志的 broker 或保持顺序的队列上，这样一条消息可能阻塞它后面的一切。许多消费者对同一个失败的依赖执行 retry，会给正在试图恢复的对象增加负载，这就是 retry 风暴。而代码 bug 可能毫无错误地失败：消费者发送 ack，返回，却写入了错误的值，于是失败路径上的任何东西都不会运行。

表现：同一个 `event_id` 在日志中反复出现，重新投递次数始终不下降，分区卡住，DLQ 在故障期间增长，而对于静默的 bug，则什么也看不到，直到对账检查发现不一致。第 6 章讲如何分辨哪些故障值得 retry，第 8 章讲 poison message 和死信路径，第 13 章和第 14 章讲如何发现并修复一个未抛出错误的 bug 所造成的损害。任何分类的代价是它可能出错，所以要让这个决定保持可见、可撤销。

无法信任的时钟与时间戳

每台机器都有一个时钟，没有两个时钟完全一致。它们会漂移，会被同步校正，而校正可能让时钟倒退。一台被暂停的虚拟机醒来时，其时钟偏差等于它睡了多久。

对事件而言，其后果是具体的。`occurred_at` 是关于某件事何时发生的事实，但依据的是某一台机器。比较来自 Orders 的事件和来自 Payments 的事件的 `occurred_at`，所能说明的，关于它们时钟的信息，不亚于关于事件顺序的信息。如果按时间戳排序来决定哪次更新胜出，那么时钟快了几秒的服务，会赢下它本该输掉的每一次冲突。

超时有一个相关的陷阱。以挂钟时间表达的截止时间，“这个预授权有效期至 14:00”，是由一台机器来比较的，而它对 14:00 的理解可能与签发方不同。在一台机器上测量时长，应当使用只向前走的单调时钟。对于跨机器的排序，我根本不会使用挂钟时间。

可靠地为事件排序的是系统的结构：在基于日志的 broker 上，是以聚合为键的分区，只要每个键只有一条生产者路径且分区数量不变，就能给出该键上事件的全序，或者是消费者可以比较的、聚合上的显式版本号。第 9 章深入讨论消息顺序。应当把 `occurred_at` 当作给人看和用于分析的信息，并对把它作为正确性的输入保持警惕。

表现：更新被乱序应用，日志中出现“不可能”的序列，即某个效果似乎先于它的原因，以及只在某一台主机上出现的 bug。

部署与 rebalance

部署是一种有计划的故障。实例被停止和启动，在这期间，工作在执行中被中断。如果部署流程发送了终止信号，而消费者在仍有消息在处理时退出，就等于有意地、在每次发布时反复地制造了“已处理但未 ack”的情形。

rebalance 是日志族的对应物，并且无需部署也可能发生。当成员加入或离开 consumer group，或某个成员被判定为无响应时，分区会被重新分配。在交接期间，两个消费者可能短暂地都认为自己拥有同一个分区，而一个慢的消费者在其分区已转移到别处之后，仍可能继续处理消息。队列族的对应物是竞争消费者在工作继续的同时其 visibility timeout 到期。

损害的模式是部署时的一阵重复消息，有时还有分区重新分配期间的短暂停滞。一次在所有实例上快速连续滚动的部署，可能连续触发多次 rebalance，最坏的情况下，消费者花在重新分配工作上的时间比做工作的时间还多。有两种缓解措施可以减轻这一点，代价是要理解更多配置：协作式重新分配，即只拿走必须移动的分区；以及稳定的成员标识，使重启的实例无需完全重新洗牌就能夺回它原来的分区。

应当让消费者优雅停止，即在退出之前完成或释放在途消息，并假定优雅路径有时会失败，所以幂等消费者仍然是真正的防线。我还会把滚动部署错峰，使它们不会在紧凑的循环中触发重新分配。表现：与发布时间吻合的重复消息和 lag 尖峰。

人为失误

最后一类是多数架构文档遗漏的一类。人在运维这些系统，而人在压力之下会犯错，通常是在修复先前的某个故障时。

设想 Parcelly 的值班工程师发现了 40,000 条死信，按下按钮把它们全部 redrive。如果原因已修复，它们会顺利流过。如果原因没有修复，它们会立刻再次失败，并淹没这位工程师本想修复的系统。如果其中一些已经通过人工变通办法处理过，它们就是对某人手工完成的工作的重复。

其他形式同样平常。有人为了让某个客户能继续，直接编辑一行数据，绕过了 outbox，于是没有事件被发布，下游服务出现分歧。有人在消费者能够读取之前，部署了一个修改过 payload 的生产者。有人在错误的环

境中清空了一个队列。这些都是修复工具范围太宽、防护措施缺失时会发生的事。

我会把运维人员的操作视为故障模型的一部分，并像设计其他任何组件一样设计它们：以小批次 redrive 并带暂停开关，修复操作走与正常变更相同的路径，使事件得以发出，并且每一次手工操作都写入审计轨迹。第 8 章讲死信和 redrive，第 12 章讲 payload 变更，第 13 章讲对账，即发现人工修复出了问题的办法。

表现：有人动过某样东西之后紧接着出现的突发尖峰，以及各服务之间数据不一致、却没有任何失败的消息可以解释。

丢失整个存储

到目前为止的每种故障，都假定存储是幸存的：数据库保留了它所提交的，broker 保留了它所 ack 的。最后一个条目抛弃了这个假定。一个集群或一个区域丢失了。一个 broker 丢失的数据超出了其复制所覆盖的范围。一个消费者的数据库从六小时前的备份中恢复。一个生产者的数据库恢复到了最近几百个订单之前的时间点。

这不是单一的故障，因为损害取决于哪个存储回到了过去。如果 Inventory 的数据库被恢复，它会忘记恢复点之后做出的预留，而 broker 也许已不再持有能重建它们的事件。如果 Orders 的数据库被恢复，它会忘记下游服务已经据以行动的订单，Payments 持有的扣款对应着 Orders 已不再知道的订单。双方都不会报错。每一方对自己是一致的，对对方则是错的。

有一个属性会有所帮助。由于 inbox 与效果位于同一个数据库中，恢复会把两者一起带回同一时刻。恢复点之前处理的消息仍在 inbox 中，所以重放会跳过它们，这是正确的，因为它们的效果也保留了下来。恢复点之后处理的消息才是有风险的，重放它们是安全的：inbox 已不记得它们，它们的效果也随之丢失了。

外部效果是例外。支付服务商处的一次扣款不会随本地数据库一起回滚，所以这些调用需要它们自己的幂等键，否则重放会向买家第二次扣款。规则是在恢复消费之前先对账：比较每个服务各自相信的内容，修复差异，然后才让流程运行。

表现：起初什么也看不到，而这正是问题所在。服务健康，仪表盘一片绿色，各服务之间的合计数却对不上。第 13 章讲发现不一致的对账，第 14 章“恢复、灾难与 broker 迁移”讲如何恢复、重放和继续。针对这一

点做设计的成本是：真正经过测试的备份，长于恢复时间的保留期限，以及在需要之前演练一次恢复的纪律。

故障地图

下面把同样的材料放在一处。请把它放在设计评审的手边。

故障	表现	实际情况	处理位置
网络分区或超时	调用方得不到答复	操作可能运行过，也可能没有	第 6 章、第 7 章、第 13 章
生产者在写入与发布之间崩溃	有订单而无事件，或有事件而无订单	状态与事件不一致	第 4 章、第 5 章
broker ack 丢失	broker 上出现重复事件	第一次发布成功了	第 7 章
Broker 故障转移	发布错误，随后是重新投递；很少见的情况下是静默丢失	保证取决于 ack 设置	第 2 章、第 13 章
消费者在工作后崩溃	重复的副作用，没有错误	工作已完成，消息仍待处理	第 7 章
慢消费者	lag 和消息年龄增长	消费者存活，但越来越落后	第 6 章、第 8 章、第 13 章
落后期间发生保留期限或过期	什么也没有；缺失的事件永远不会到达	未读取的消息已被 broker 删除	第 8 章、第 13 章
poison message 或失败的依赖	同一条消息被重新投递；分区卡住；故障期间 DLQ 增长	故障是确定性的，或原因在消息之外	第 6 章、第 8 章
无报错的消费者 bug	什么也没有；合计数事后对不上	处理器成功了，却写入了错误的结果	第 13 章、第 14 章
时钟偏差	乱序更新，不可能的序列	时间戳与因果关系不一致	第 9 章
部署或 rebalance	发布时出现重复消息和 lag	工作的归属在执行中发生了转移	第 7 章、第 13 章
运维人员失误	手工操作之后的尖峰；静默的分歧	修复改变了正常路径之外的状态	第 8 章、第 13 章
丢失整个存储	起初什么也没有；恢复之后服务之间不一致	某个存储回到了过去，或其数据已经丢失	第 13 章、第 14 章

有两点很突出。第一，“表现”一列中，重复消息出现在三种不同的成因下（ack 丢失、消费者崩溃、部署），重新投递则出现在第四种。这就是为什么一个模式，即幂等消费者，值得占用整整一章。第二，最后一列不断回到第 13 章和第 14 章。有些故障在发生的当下无法预防，甚至无法检测，所以系统还需要一种事后察觉的办法，能发现真相的两个视图彼此不一致。

把故障模型写下来

故障模型只有在被写下来、保持最新并被共享时才有用。在我看来，这一实践比其中任何一个条目都更有价值，因为写作这一行为迫使人提出“我们在这里假定了什么？”，而意外就藏在假定之中。

我会把它作为一份简短的文档放在代码仓库中，紧挨着架构说明，每种故障一个条目，每次使用同样的字段。这些字段对应那三个问题：“where”和“we assume”说明什么会失败，“observable as”说明如何看到它，“defense”说明哪项保证得以保全。一个可用的条目如下所示：

```
failure:        consumer crashes after commit, before ack
where:          Inventory, handler for OrderPlaced
we assume:      this happens during every deploy, and at random
otherwise
observable as:  redelivered message with the same event_id
effect if unhandled: second stock reservation for the same order
defense:        inbox row inserted in the same transaction as the
reservation
verified by:    test that kills the consumer between commit and ack
owner:          Inventory team
last reviewed:  2026-09-30
```

要紧的字段是最后三个。“Verified by”把条目从一个信念变成一个可以测试的断言，第 15 章展示如何测试它。“Owner”意味着有人会被呼叫。“Last reviewed”承认模型会腐朽：服务被添加，broker 被更换，去年成立的假定如今可能不再成立。

> **注意**：模型也应当写明假定不会发生什么。“我们假定数据库永远不会丢失已提交的事务”是一个正当的条目。写下这个假定，并知道它若不成立该怎么办，好过让它保持隐含。

在三个地方使用该模型。在设计评审中，把每个新流程对照清单走一遍，问它暴露于哪些故障之下。在事故评审中，找出事故属于哪个条目，如果哪个都不属于，就新增一个。在测试中，让每个条目成为一个场景。出现在模型中却不出现在任何测试中的故障，只是一种希望。

权衡与成本

编写故障模型需要时间，还要承受记录自己所不知道的东西的不适。只有一个服务和一个队列的小团队，不需要这份清单上的每一个条目。可以从最会造成伤害的三四种故障开始。

模型也有边界。本书中的每一项防御都假定某些故障不在范围之内：数据库是持久的，broker 的复制按配置工作。这些假定是合理的，而每一条都是模型可能出错的地方。把它们列出来，才能判断它们是否可以接受，而当它们不可见时，这件事就无法做到。自己代码中的 bug 则不同。它在范围之内，作为一种需要检测和恢复的故障，因为它一定会发生，而模型必须说明如何察觉它，并修复它所写入的内容。

有 40 个条目的模型没有人会读，只有三个条目的模型会漏掉情形。我会以新团队成员能一次坐下来读完的清单为目标。

以最大强度针对每一种故障做设计，同样不是目标。在 broker 故障转移中做到零丢失，要付出延迟和金钱的代价，而挺过双重故障，代价更高。偶尔发送两次的 Notifications 邮件，不值得与绝不能重复的 Ledger 分录同等的防御。模型的任务是让投入按风险分配。如果说不出一种故障的业务成本，我就会对付出技术成本去预防它犹豫不决。

模型不会使系统变得可靠。它使假定可以被检查。

常见错误

- **只为崩溃建模**。为崩溃建模很诱人，因为它干净而且可检测。代价高昂的故障是含糊的那些，例如超时和慢消费者。修复：对每一次调用，写下当没有答复到达时调用方会怎么做。
- **把超时当作失败**。把“没有响应”读作“没有发生”，感觉很自然。修复：把它视为未知，并设计使重复操作安全或可检查的操作。
- **把 broker 的 ack 当作证明**。“已 ack”这个词听起来是终局性的。修复：了解所用的 ack 设置实际保证什么，并为它不保证的情形做对账。

- **按时间戳排序**。`occurred_at` 就在信封里，按它排序只需一行。修复：在分区键内部按聚合版本排序，把时间戳留给人看。
- **忽视部署**。发布感觉像是例行公事，而不像故障。修复：把每一次部署都算作一次计划内的消费者崩溃，并在有消息在途时测试关闭流程。
- **把运维人员排除在外**。修复工具得到的评审比产品代码少。修复：把 redrive、重放和手工编辑纳入模型，配上批次限制、暂停开关和审计轨迹。
- **把已恢复的数据库当作干净的开始**。恢复感觉像是事故的结束。修复：任何一次恢复之后，在恢复消费之前先与其他服务对账，并检查没有回滚的外部效果。
- **对不会改变的失败执行 retry**。retry 感觉很勤勉。修复：把取决于时机的故障与取决于消息的故障分开，并尽早把第二类送入死信路径。
- **只写一次模型**。在设计阶段写出的文档只是一张快照。修复：指定负责人，在事故之后评审它，并为每个条目标注日期。

要点

- 超时意味着“我不再等了”，而不是“失败了”，所以要设计在未知以任何一种方式解决时都保持正确的操作。
- 每个接缝都有自己的崩溃窗口，而损害是静默的。双写会留下没有事件的订单，或关于不存在的订单的事件，而在提交之后、发送 ack 之前崩溃的消费者，每次重新投递都会得到一条重复消息，这使幂等消费者成为必需。
- broker 的 ack 是边界由复制和刷盘设置决定的证据，所以要了解边界，并计划对落在边界之外的部分做对账。
- 慢消费者和部署会在没有任何组件坏掉的情况下造成重复消息和停滞，所以要限制负载，测量 lag 和消息年龄，并让消费者优雅停止。
- 每次都失败的消息，或对每条消息都失败的依赖，需要与偶发故障不同的应对，所以要在 retry 之前先分类，并把转入死信留给不会恢复的情形。
- 挂钟时间戳在不同机器之间并不一致，所以要在分区键内按聚合版本为事件排序，并把 `occurred_at` 留给人看。
- 运维人员和恢复操作是系统的一部分：已恢复的存储回到了过去，所以要在恢复消费之前先对账，并保存一份写明负责人、测试和评审日期的书面故障模型，使修复不会成为下一次事故。

首先要消除的故障在生产者一侧，状态变更与它的事件可能在无人察觉的情况下分离。第 4 章以 transactional outbox 开启第二部分，它封闭了这个窗口。

第二部分

可靠地发布

第 4 章 — 双写问题与 transactional outbox

设想 Parcelly 推出卖家促销几周之后。一名支持工程师打开一张工单，内容全文如下：“买家说她已经完成订单 81724 的结账，但卖家一直没有收到发货标签。订单显示为已下单。”工程师查了 Orders 数据库，找到了这一行，状态为 `PLACED`，创建于 14:02:11。接着她查 Payments，那张订单没有预授权记录。Inventory 里也没有库存预留。就 Parcelly 的其余部分而言，这张订单并不存在。

她在 14:02 前后检索 Orders 的日志，找到了一条部署标记。服务在提交订单 200 毫秒之后、发布 `OrderPlaced` 之前重启了。代码做的恰是它被写成要做的事：先保存，再发布。没有抛出任何错误，因为本该抛出错误的进程已经不在了。假设再做一次快速查询，发现同一分钟内还有 11 张订单是同样的情形。

团队手工修复了这 11 张订单，并在 runbook 里补了一条说明。没有人觉得这样做令人满意。这段代码并没有哪里错得会被评审人指出，可是只要进程在不巧的时刻死掉，它就可能丢失工作。换一台更大的机器不会改变这一点，更小心的开发者也不会。本章要回答的问题是：在数据库与 broker 之间没有跨越两者的事务的前提下，服务能否修改自己的数据库，并以一种不会割裂的方式把这一变化告知系统的其余部分。

问题及其难点

第 1 章定义过双写：在没有共享事务的情况下更新两个系统。Orders 必须向它的关系型数据库写入一行，并向 broker 发布一条消息。数据库为前者保证原子性。broker 在某些配置下为后者保证持久性。没有任何东西保证两者同时发生或同时不发生。

第 3 章用一张图勾勒了 producer 的崩溃窗口。这里我想把它们列成一张表来逐项走一遍，因为两次写入的先后顺序是一项设计决策，大多数团队在做出它时并未察觉，而每一种选择背后都有一种故障在等待。

每种顺序都会出错

取两种显而易见的序列，对每一个可能的故障点，问一问事后世界是什么样子。这与写下数据库和 broker 之间接缝的故障语义是同一种练习：什么会失败，如何观察到，以及哪项保证仍然成立。下表就是针对生产者的这种练习。

序列	故障点	数据库状态	broker 状态	损害
先提交，后发布	提交之后、发布之前崩溃	订单已保存	没有事件	事件丢失。下游永远不会得知这张订单
先提交，后发布	发布失败或超时，进程存活	订单已保存	未知	调用方必须 retry 或放弃。放弃就会丢失事件
先提交，后发布	broker 已接受，ack 丢失	订单已保存	事件已存在	retry 会发布一条重复消息
先发布，后提交	发布之后、提交之前崩溃	没有订单	事件已存在	幽灵事件。下游依据一个从未成真的事实行事
先发布，后提交	提交失败（约束、死锁、超时）	没有订单	事件已存在	同样的幽灵事件，只是 producer 里多了一条可见的错误
在事务内发布，最后提交	发布之后提交失败	没有订单	事件已存在	事务块让代码产生虚假的安全感，而 broker 并不在其中

最后一行值得多说几句，因为它看起来最安全。许多框架允许在数据库事务块内部发布。读起来像是发布属于事务的一部分。其实不是。对 broker 的调用立即发生，无法回滚，而提交仍然在其后进行。这就是先发布后提交的顺序，只是缩进更好看。

与之对称的是许多框架提供的“提交后”回调，即事务提交之后运行的钩子。团队常常以为它是安全的，因为只有已提交的数据才会触发发布。它其实就是表中的第一行：先提交，后发布，而发布的意图只保存在进程内存里。提交与钩子之间发生崩溃就会丢失事件，也没有任何地方记录过这条事件曾经需要发出。

还有第四种形态是这张表没有涵盖的，那就是 broker 变慢。如果发布发生在事务之内，数据库在等待 broker 期间就要一直持有锁和连接。broker 慢一分钟，就会变成数据库慢一分钟。本该相互独立的两个系

统，可用性被耦合在了一起。

为什么 retry 没有用

很诱人的修补办法是在发布外面套一个 retry 循环。retry 对表中第二行有帮助，那里进程存活，可以再试一次。对第一行则毫无作用，那里进程已经不在了，而且它还制造出第三行的重复消息。第 6 章会专门讲 retry。眼下只需注意，retry 循环存在于进程的内存之中，而我们关心的故障恰恰是进程消失。

第二种诱人的修补办法是一个定期任务，比较数据库与 broker，再把缺失的部分重新发布。有些团队会这样做，我认为把它作为安全网是合理的，第 13 章会以对账的名义回到这一点。但把它当作主要机制则有两个问题。broker 很少能按业务键查询，因此“缺失了什么”就变成一次代价高昂的猜测。而且这个任务本身执行的也是同样的双写，所以会以同样的方式失败，只是更少发生，也更令人困惑。

为什么不用分布式事务

如果问题是两个系统没有共享事务，经典的答案就是给它们一个共享事务。**两阶段提交**（2PC）以 XA 之名在数据库与事务管理器之间实现了标准化，用于协调多个资源管理器。在第一阶段，每个参与者进行准备，也就是承诺自己可以提交，并持有自己的锁。在第二阶段，协调者通知每个参与者提交，如果有人未能完成准备，则通知回滚。

在纸面上，这就关闭了窗口。在实践中，对于 broker 与服务数据库，它很少是答案，原因大多出在运维上。

- **支持程度参差不齐**。许多流行的 broker 根本没有实现事务管理器接口。即使实现了，其保证与故障处理也与数据库不同，需要同时承接两套规则。
- **协调者做决定期间，参与者一直持有锁**。如果协调者在两个阶段之间崩溃，已准备的事务会停留在存疑状态。这些行一直被锁住，直到有人或有什么机制来解决它们。存疑窗口恰恰是那种罕见、持续很久、没有人演练过的故障。
- **协调者成为一个必须自己运行、保护和恢复的组件**。它自己需要持久化日志，还需要一套必须正确无误的恢复流程。
- **可用性会相乘**。一个事务需要数据库、broker 和协调者同时健康，那么只有三者都可用时它才可用。当初让 broker 有吸引力的独立性就没有了。

- **吞吐量和延迟都会受损**。两次往返、每个参与者处的强制写入以及跨网络调用持有的锁，叠加起来代价不小。
- **它无法延伸到外部世界**。邮件服务商或支付服务商不会加入本地事务。即使团队为 broker 采用了 2PC，每一次外部调用仍然需要另一套答案。

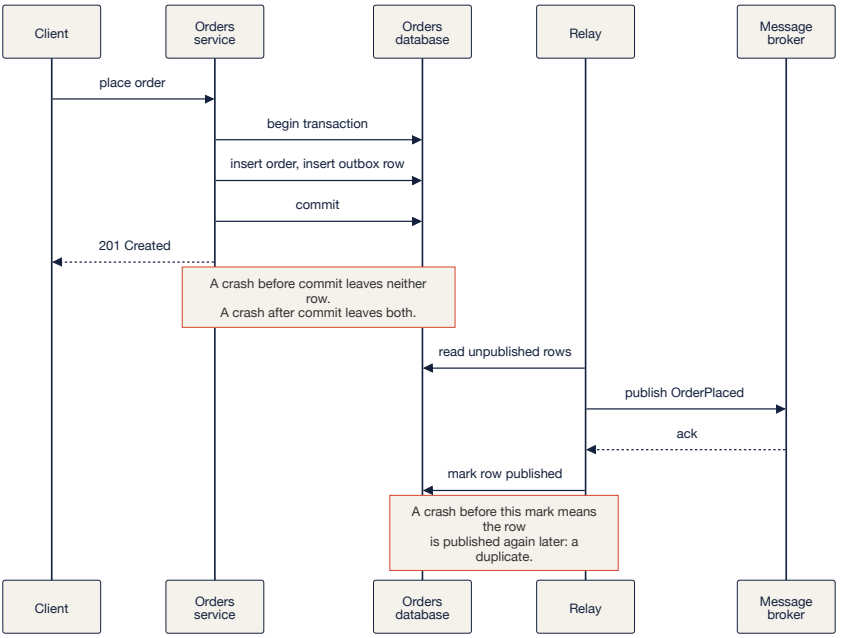
我不会在所有情况下都排除 2PC。一个同时掌控两种资源、运行着成熟的事务管理器、并且具备解决存疑事务的运维能力的团队，可以让它运转起来。但对大多数用服务搭建市场平台的团队来说，每个服务有各自的数据库，broker 又不提供 XA 接口，成本很高而收益有限。他们想要的比分布式事务小：他们想要状态变更与发布意图一起提交。这是一个本地问题，而本地事务已经被充分理解。

Transactional outbox

思路是把双写变成一次写入，外加其后一次可重复的搬运。服务不在请求之内向 broker 发布，而是在与状态变更相同的本地事务里，向自己数据库中的一张表写入一行，描述这个事件。这张表就是 **transactional outbox**。从这里开始，本书称之为 outbox。

另有一个独立的进程，即 **relay**，读取 outbox 的行并发布到 broker。第 5 章讲的就是 relay。在本章里，我们把它视为一个会把每一行至少投递一次的函数，并看看这能换来什么。

推理很简短。订单与它的 outbox 行要么一起提交，要么都不提交，因为它们位于同一个数据库、同一个事务之中。行一旦存在，发布的意图就是持久的。提交之后发生崩溃不再会丢失事件，因为服务或它的替身再次启动时，这一行仍然在那里。提交之前发生崩溃既没有订单也没有行，因此不会出现幽灵事件。



留意图中的两条注释。第一条表明我们关闭的窗口。第二条表明我们保留的窗口：relay 可能已经发布，却在记录这一事实之前崩溃。outbox 并不消除重复消息，而是把故障移到一个有界且可处理的地方。后文会回到这一点。

表

outbox 沿用本书其余部分使用的形态。它携带全部信封字段，因此 relay 不必向生产它的服务询问任何东西就能构造出事件。本章在信封之外增加了四列，每一列各有缘由。

```
table outbox:
    id              bigint primary key        // allocation order, not
commit order
    event_id        uuid unique not null      // stable across
redelivery
    aggregate_type text not null              // "order", "shipment"
    aggregate_version bigint not null         // aggregate's version,
bumped in this transaction
    event_type      text not null             // "OrderPlaced"
    schema_version int not null
    occurred_at     timestamp not null        // time of the fact
    created_at      timestamp not null default dbNow()  // insert time,
database clock
    producer        text not null             // "orders"
    correlation_id text not null
    causation_id    text not null
    partition_key   text not null             // aggregate id, e.g.
"order-81724"
    payload         json not null
    headers         json null                 // transport hints, trace
context
    published_at    timestamp null            // null means not yet
published
```

其中几项选择需要辩护。

- **`id` 与 `event_id` 是两回事**。`event_id` 是一个 UUID，消费者会把它用作幂等键，事件被发布两次时它不得改变。`id` 是数据库分配的序列号，relay 可以用它来查找行。它并不表示提交顺序，下一节会说明原因。
- **`aggregate_version` 是消息顺序的权威来源**。它是聚合的版本，在写入这一行的同一个事务里递增。对 Parcelly 的订单而言，它就是 `order_version`。需要按聚合保持顺序的消费者比较的是这个数字，而不是 `id`，也不是到达顺序。第 9 章会展示消费者如何使用它。标准信封中没有对应它的字段，所以 relay 构造事件时，会把这一列复制到 payload 中，命名为 `order_version`，消费者就从那里读取。
- **`aggregate_type` 与 `partition_key` 是分开的**。`partition_key` 是 broker 用来保证顺序的键，它就是聚合 ID，例如 `order-81724`。`aggregate_type` 告诉 relay，也告诉读这张表的人，这个键指向哪一种东西，同时它也便于把行路由到不同的 topic。
- **`created_at` 与 `occurred_at` 是分开的**。前者是这一行被写入的时间，后者是事实发生的时间。通常两者相差在毫秒之内。当服务记录一个它很晚才得知的事实时，两者就会分离，例如承运商的扫描记录在事件之后才到达。relay 延迟指标应当使用 `created_at`，因为延迟

是这张表的属性，而不是业务的属性。让数据库通过列默认值来填充它，而不是在应用代码里调用 `now()`，这样时钟就只有一个。它仍然是插入时间，而不是提交时间：长事务会让这一行看起来比 relay 所见的更旧，这部分延迟并不是 relay 的过错。

- **`headers` 存放传输层关注的内容**。trace 上下文和路由提示放在这里，使 payload 保持为业务事实。如果团队无法决定某个字段是数据还是传输提示，应当先读第 11 章，再决定是否把它加入 payload。
- **`published_at` 可以为空，而且空值有含义**。没有 `published_at` 的行，就是 relay 仍欠 broker 的一行。有些团队改用 `status` 列，取值如 `PENDING`、`PUBLISHED` 和 `FAILED`。两种设计承载的信息相同。我更倾向于时间戳，因为它还能说明发布发生在何时，在追问 relay 晚了多久时这很有用，而且出错的余地更小。如果需要一种办法来暂存永久失败的行，到那时再加一列 `failed_at`，不必提前。

那一刻来得比听上去更早。broker 永远不会接受的行，例如过大的 payload 或序列化器拒绝的行，会阻塞它的键，在最简单的 relay 里还会阻塞整张表，直到有人修复。第一次出现这种情况时，就需要 `failed_at` 列（如果选择了状态列，则需要 `FAILED` 状态），以及针对暂存行的告警。第 5 章会展示 relay 如何使用它。

对未发布行、按插入顺序建立的索引，是 relay 唯一需要的索引。没有它，relay 就要扫描一张永远增长的表。第 5 章会讲清理。

在同一个事务中写入

生产端的代码很短，短正是关键所在。下面这个函数就是 Orders 请求路径上的全部改动。

```
function placeOrder(request):
    order = buildOrder(request)
    // a new order starts at version 1
    order.version = 1

    transaction:
        orderInsert(order)
        outboxInsert({
            event_id:       newUuid(),
            aggregate_type: "order",
            event_type:     "OrderPlaced",
            schema_version: 1,
            aggregate_version: order.version,   // 1 for a new order
            occurred_at:    now(),
            producer:       "orders",
            correlation_id: order.id,
            causation_id:   request.call_id,
            partition_key:  order.id,
            payload:        orderPlacedPayload(order)
        })
    // commit point: the order and its event exist together or not at
all

    return order
```

有三个细节比看上去更重要。

第一，函数里没有任何东西与 broker 对话。broker 宕机不会导致下单失败，broker 响应一分钟也不会让一个数据库事务一直敞开。这是可用性上的实在收益：broker 宕机期间 Orders 仍然可以接受订单，outbox 持续增长，直到 relay 能够把它排空。

第二，`event_id` 在提交之前创建，并存入这一行。此后这个事件的每一次重新发布都带着同一个 ID。这是消费者所依赖的属性，也是为什么在 relay 里、在发布时生成这个 ID 会是一个错误。

第三，payload 在事务之内构建，依据的是事务已经写入的状态。如果 payload 是稍后重新读取订单来构建的，并发的更新可能改变它，事件就会描述一个当时并不存在的事实。事件是对已发生之事的快照，快照就在变更发生的同一处拍下。

伪代码假定订单的插入与 outbox 的插入使用同一个数据库连接和同一个事务。这一点被意外破坏的次数比听上去要多。数据访问层如果为 outbox 另开一个连接，或者自动提交 outbox 的插入，就会悄悄恢复双写，只不过现在发生在两张表之间。针对它的测试应当放进测试套件：在订单插入之后强制制造一次失败，并断言不存在任何 outbox 行。

提交顺序、`id` 顺序与聚合锁

新订单没有竞争者，所以 `placeOrder` 不需要锁。此后对既有订单的每一次变更都需要，而且锁必须先加。下面是 `cancelOrder`，它会发出 `OrderCancelled`。

```
function cancelOrder(order_id, reason):
    transaction:
        // lock the aggregate row before anything else is written
        order = orderSelectForUpdate(order_id)
        order.version = order.version + 1
        orderUpdate(order, status: "CANCELLED")
        outboxInsert({
            event_id:          newUuid(),
            aggregate_type:    "order",
            aggregate_version: order.version,
            event_type:        "OrderCancelled",
            partition_key:     order.id,
            payload:           orderCancelledPayload(order, reason)
            // remaining envelope fields as in placeOrder
        })
    // commit point: state, version bump and event together
```

锁使版本值可信，也是唯一让 `id` 顺序对单个聚合有意义的东西。没有锁的话，取同一张订单上的两个事务。事务 B 插入它的 outbox 行，得到 `id` 101，然后在事务 A 持有的订单行上阻塞。A 插入 `id` 102 并提交，B 随后提交。提交顺序是先 A 后 B，而 `id` 顺序是先 B 后 A。在一次轮询中同时看到这两行的 relay，会把它们反着发布。先加锁的话，A 提交之前 B 无法插入，所以同一个聚合的各个 ID 遵循它的提交顺序。

即便如此，我也不会让消费者依赖 `id`。不同聚合的行会交错，并行的 relay 会在批次之间打乱顺序，而且 Orders 数据库之外没有人能看到 `id`。`aggregate_version` 随事件一同传递，在 payload 里叫 `order_version`，它说明这张订单上的各个事实是以什么顺序发生的。第 5 章会回到 relay 能保持什么、不能保持什么，第 9 章会展示消费者一侧。

Relay 的义务

relay 的契约有意设计得很小。它选出未发布的行，把每一行发布到 broker，并在收到 broker 的 ack 之后将其标记为已发布。下面是最简单的版本，第 5 章会讲它忽略的生产环境问题，例如批处理、加锁、消息顺序与故障转移。

```
function relayOnce():
    // oldest first
    rows = outboxSelectUnpublished(limit: 100)

    for each row in rows:
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
eventFrom(row))
        if result is failure:
            // stop: publishing later rows would overtake this one for
its key
            return
        // if the process dies here, the row stays unpublished
        // and will be published again: this is the duplicate window
        outboxMarkPublished(row.id, now())
```

循环里的那条注释就是全部保证。只有在收到 broker 的 ack 之后才标记这一行，所以这次调用要等待 ack，所以一次失败就会停止整个批次。发布与标记之间发生崩溃，这一行仍是未发布。下一轮，relay 会再次发布它。另一种做法是先标记、后发布，那会让这种故障变成事件丢失，而不是重复消息。outbox 是有意选择了更安全的那种故障。

Outbox 能保证什么

outbox 带来三样东西，同时拿走一样。

- **状态与意图之间的原子性**。状态变更与发布的决定一起提交。不存在没有 outbox 行的已提交订单，也不存在没有已提交订单的 outbox 行。
- **意图的持久性**。事务一旦提交，事件就与数据库一样持久。relay 可以宕机，broker 可以宕机，producer 可以被替换，而这一行会一直等着。
- **至少一次发布**。每一个已提交的行最终都会被发布，前提是 relay 与 broker 最终都能恢复，relay 按状态而不是按 ID 游标来查找行，清理只清除已发布的行，并且 broker 永远不会接受的行会被暂存并触发告警，而不是任其永远阻塞（这一切都在第 5 章）。一行可能被发布不止一次。
- **不保证：exactly-once 发布**。relay 无法原子地完成向 broker 的发布与在数据库中的标记，原因与应用当初无法做到是一样的。窗口从请求路径移到了 relay，在那里它更小，可观测，也可修复。

我想谨慎地对待这个总结。outbox 并不声称事件永远不会丢失。它声称的是：双写在 producer 一侧的丢失窗口已经关闭，前提是应用对每一个必须产生事件的状态变更都经由 outbox 写入。开发者手工更新这张表，或者走了绕过 outbox 的路径，就会重新打开窗口。第 3 章把这列为人为错误，而这也是拥有 outbox 的团队仍然丢失事件的最常见方式。

重复消息成了消费者的问题

由于发布是至少一次的，通过 outbox 发布的事件，其每一个消费者都必须能容忍重复消息。这不是需要靠工程手段消除的缺陷，而是这项保证的代价，应当把由谁来付这笔代价讲清楚。

producer 团队承担较轻的部分。他们增加一张表，写入一行，运行一个 relay。消费者团队承担较重的部分，因为他们必须识别出自己已经见过的 `event_id`，并避免重复其效果。Payments 不得预授权两次。Notifications 不得把同一封邮件发送两次。Ledger 不得过账同一笔分录两次。

> **警告**：采用 outbox 却没有幂等的消费者，是用事件丢失问题换来了重复效果问题。对某些事件而言，例如重复扣款，第二个问题代价更高。应当把两者一起交付，消费者如何识别重复的工作，见第 7 章。

我也想把这句话直白地告诉 producer 团队：outbox 是与消费者之间的契约。同一个事实的每一次重新发布，`event_id` 都必须保持稳定，这份契约应当写进事件文档，而不是含而不露。

一次简短的故障走查

为了对照第 3 章的窗口检验这个设计，把 Orders 的各种崩溃再走一遍。

1. 进程在事务提交之前死亡。没有订单，没有 outbox 行，没有事件。客户端看到错误或超时，并可能对请求执行 retry，这是 API 层的另一个问题，由第 7 章的幂等性讨论来处理。
2. 进程在提交之后立即死亡。订单和行都存在。relay 在下一轮发布这一行。客户端可能没有收到响应，但系统是一致的。
3. relay 在发布之后、标记之前死亡。事件在 broker 里，而行仍是未发布。下一轮会再次发布它。消费者看到一条带有相同 `event_id` 的重复消息。

4. broker 不可用。relay 发布失败，让行保持未发布，稍后再试。Orders 继续接受请求。延迟增长，告警也应当随之触发。
5. 服务在提交期间失去连接，不知道结果。要么订单和它的行都存在，要么都不存在，服务无法分辨是哪一种。它不得猜测。稳妥的答案是让客户端的 retry 携带 API 层的幂等键，这样无论数据库处于哪种状态，retry 最终都会收敛为一张订单和一个事件（第 7 章）。

每种情形的结局要么是一致，要么是一条重复消息，没有一种以幽灵事件或静默丢失收场。这就是 outbox 的全部主张，足以证明它的价值。

有一种重复消息位于 outbox 的上游，下游的任何防御都看不到它。当客户端在响应丢失后执行 retry，而 API 自己没有键时，Orders 会创建第二张订单和第二个 `OrderPlaced`，带着全新的 `event_id`。每个消费者的 inbox 于是看到两个不同的事件，因为它们本来就不同。outbox 把工作做了两遍。修复应当落在 API 上，采用第 7 章的 `Idempotency-Key` 做法。

替代方案与相邻方案

outbox 并不是阻止双写的唯一办法。另有两种设计值得公平比较，因为团队完全可能更偏爱其中之一。

Listen to yourself

在 **listen-to-yourself** 模式中，服务不先写自己的数据库。它先把事件发布到 broker，然后消费自己的事件，在处理器里应用状态变更。broker 成为唯一的写入点，数据库则成为服务所听到内容的投影。

这消除了双写，因为只剩一次写入。它同时也改变了服务的形态。客户端的请求在状态存在之前就已返回，所以调用方无法读到自己的写入，除非系统提供某种等待的办法。校验必须在发布之前完成，因为事件一旦发布就是事实，无效的事件事后无法被拒绝，除非做补偿。而且应用事件的处理器本身必须幂等，这又带回了同样的消费者负担。

对于本来就以事件思考、并且能容忍自身读取存在最终一致性的服务，我会考虑这种做法。对于 Orders，买家的浏览器期望点击按钮后立即看到订单，我就不会采用。读到自己写入的缺口是一个产品问题，团队最终会靠各种变通办法来解决。

Event sourcing

在 **event sourcing** 中，事件就是权威来源。服务按聚合存储事件序列，当前状态由它们推导而来。写入事件存储就是状态变更，所以没有什么需要与一个独立的事件保持一致。随后向 broker 发布，是从事件存储中读取，这是换了个名字的 relay。

event sourcing 通过改变数据库是什么来解决双写。它还提供完整的历史，以及重建投影的能力。它的代价是一种不同的编程模型，已存储事件的 schema 演进（这些事件永远不能随意改写），以及整个团队的学习曲线。这是一项重大的架构承诺，我不会仅仅为了避开 outbox 就采用它。如果已经在用，仍然需要 relay，本章中关于至少一次发布的一切仍然适用。

比较各个选项

选项	消除了什么	代价是什么	最适合什么时候
先提交，后发布	什么也没有	崩溃时静默丢失	丢失一条事件可以接受，例如尽力而为的指标
2PC / XA	窗口，理论上	锁、协调者、支持程度参差不齐	同时掌控两种资源，并且自己负责恢复
Transactional outbox	producer 一侧的窗口	一张额外的表、一个 relay、重复消息	状态存放在关系型数据库中，且事件不得丢失
Listen to yourself	第二次写入	自身读取的最终一致性、需要提前校验	服务本来就以事件为先
Event sourcing	独立的状态写入	一种不同的状态模型	历史本身就是需求

对于已经把状态存放在关系型数据库中、并需要告知他人的服务，outbox 是最小的改动。这就是本书的大部分内容都以它为前提的原因。

权衡与成本

outbox 描述起来便宜，运行起来并不免费。以下是我会在团队做出承诺之前摆在他们面前的成本。

更多活动部件。有一张表，一个 relay 进程，还有要监控的延迟。出问题时，多了一个需要查看的地方。relay 是需要有人负责的组件，它需要部署、告警和 runbook。第 5 章会给出细节。

额外的数据库负载。每个事件都是与业务写入处于同一事务中的一次插入，每次发布都是一次读取加一次更新。事务会稍微变长，写入量会上升。对大多数服务而言，与其余工作相比，这点增长很小。写入事件量非常高的服务，应当在自己的硬件上测量，而不是相信我说的“小”。

增加的延迟。事件在事务提交之后、relay 发现这一行之后，才抵达 broker。采用轮询时，延迟取决于轮询间隔。采用基于日志的方式可以更短，第 5 章会比较两者。无论哪种，对于要求事件在请求之后几毫秒内抵达 broker 的流程，outbox 都不是合适的选择，选择之前应当清楚自己的延迟预算。

设计上就有重复消息。上文已经讲过。如果某个消费者无法做到幂等，那么 outbox 对它没有任何改善，团队必须判断这个消费者是否还应当以这种形式存在。

消息顺序只能部分保持。relay 按 `id` 读取行，而 `id` 顺序并不是提交顺序，除非每个事务在插入 outbox 行之前都先锁住聚合行。这把锁会让同一聚合的写入者串行化，在热点聚合上这本身就是一种成本。并行的 relay 会让顺序更糟，而且这把锁对跨聚合的情形无能为力。`aggregate_version` 列给了消费者可供检查的依据，代价是每次写入都要递增一次版本。outbox 提供的是一个可以强制保证顺序的地方，而不是白得的顺序。第 5 章和第 9 章会回到这一点。

增长与清理。已发布的行不断累积。不删除或归档的话，这张表会一直增长，直到 relay 的查询变慢。保留策略是一项决策，它与对账相互作用，因为旧的行同时也是证明发布过什么的证据。

只适用于状态位于支持事务的数据库中的情形。如果状态存放在没有多行事务的存储里，就需要另一种办法，例如该存储专属的变更流，或 event sourcing 式的设计。仅在单个文档内保证原子性的存储有它自己的一种变体：把待发布的事件保存在聚合文档内部，让 relay 从那里把它们排出。

何时不该使用

在以下情况下，我不会加入 outbox：

- 事件是尽力而为的，丢失它可以接受，例如一条指标或一个价值不高的分析提示
- 服务本身没有持久状态，因此没有什么可以与事件一起提交
- 系统小到由单个进程同时拥有状态及其消费者，直接调用更简单

- 团队尚且无法让消费者幂等，这种情况下我会先解决这一点，因为 outbox 会抬高这一缺口的风险

> **注意**：“我们用了 outbox”只是关于某个服务的 producer 路径的一句陈述。它不说明消费者是否安全，消息顺序是否成立，也不说明事件是否设计得当。应当把它视为一组模式中的一个，而不是可靠性的保证印章。

常见错误

- **在单独的事务中写入 outbox 行**。这很诱人，因为框架钩子或事件总线抽象隐藏了连接。两次写入于是变成同一个数据库里两张表之间的双写，窗口又回来了。应当使用同一个连接和同一个事务，并测试回滚。
- **在“提交后”钩子里发布**。这很诱人，因为它只对已提交的数据触发，也不需要表。它就是意图保存在内存里的先提交后发布，所以两者之间发生崩溃就会丢失事件。应当在事务里写入这一行，让 relay 来发布。
- **先插入 outbox 行，再锁定聚合**。这样的代码在所有只有一个写入者的测试里都能运行。在并发下，同一张订单的 `id` 顺序与提交顺序会分离，relay 可能把它的事件反着发布。应当先锁定聚合行，递增其版本，再插入。
- **从请求路径发布，同时把 outbox 当作备份**。这像是双保险。结果是出现两个发布者，重复消息的时机各不相同，顺序也无从推断。应当让 relay 成为唯一的发布者。
- **在 relay 里生成 `event_id`**。这看起来无害，因为 relay 是构建消息的那一方。重新发布时就会得到新的 ID，消费者无法把它识别为重复消息。应当在写入这一行时生成该 ID。
- **在收到 broker 的 ack 之前把行标记为已发布**。它减少了重复消息，也显得更整洁。但它把重复消息变成了事件丢失，后者是更糟的故障。应当在收到 ack 之后标记，并接受重复消息。
- **在提交之后构建 payload**。重新读取聚合来填充事件，似乎能让生产端的代码保持简单。其间另一个事务可能改动数据，事件就不再与事实相符。应当依据事务所写入的内容来构建 payload。
- **让某些代码路径绕过 outbox**。迁移脚本、管理工具或手工执行的 SQL 修复会改变状态却不发出任何事件。应当让每一个重要的状态

变更都经过同一个函数，并检查手工修复本应产生的事件。

- **忘记清理**。这张表运转了几个月，随后 relay 的查询在最糟糕的时刻变慢。创建这张表时就应当确定保留策略。
- **把 outbox 当作可靠性故事的终点**。它只关闭了一个窗口。消费者仍然需要幂等性、retry 以及一个存放永远无法成功的消息的地方。

要点

- 双写不能靠选一个更好的顺序来变得安全：先提交后发布会丢失事件，先发布后提交会凭空造出事件，而在事务块内发布仍然属于后一种。
- 两阶段提交原则上可以关闭这个窗口，但对于服务数据库和 broker，它在锁、协调者运维和可用性上的代价通常超过了问题本身的价值。
- transactional outbox 把双写变成一次本地事务加一次可重复的搬运，做法是把事件作为一行，与状态变更写在同一个事务里。
- outbox 行携带完整的事件信封，一个在写入时创建的稳定 `event_id`，使每一次重新发布都能被识别，以及一个在聚合锁之下递增的 `aggregate_version`，因为 `id` 顺序不是提交顺序。
- outbox 保证状态与意图的原子性、持久的意图和至少一次发布，它不保证一条消息恰好被发布一次。
- 重复消息是这项保证的代价，所以没有幂等消费者的 outbox，不过是用一种故障换另一种故障。
- Listen-to-yourself 和 event sourcing 同样能消除双写，但它们会改变服务读取和存储自身状态的方式，所以我会因为它们所能给予的东西而选择它们，而不只是为了避开 relay。

发布只是搬运的一半：必须有东西读取这些行，并把它们送到 broker，而不能丢失、打乱顺序或淹没 broker。第 5 章讲 relay、轮询与变更数据捕获的对比以及如何在生产环境中运行 outbox。

第 5 章 — Relay、轮询与变更数据捕获

设想 Parcelly 的一个周三上午。Orders 团队两周前上线了 transactional outbox。每个 `OrderPlaced` 事件现在都与订单本身写在同一个本地事务里，写入 `outbox` 表，第 4 章里的双写窗口已经关闭。团队认为问题已经解决，转去处理下一张工单。

relay 是一个小进程，每秒醒来一次，读取 `id` 大于它上次所见的行，发布它们，并记住新的最大 `id`。它以单实例运行，自从部署那天起就没有人再想过它。

然后，一次例行部署在错误的时刻替换了 relay。旧实例已经向 broker 发布了一批 200 行，却还没来得及保存新的位置。新实例从上次保存的位置启动，读到同样的 200 行，又发布了一遍。Inventory 对已经处理过的订单第二次预留了库存。一周后，新增了一个清理任务，悄悄删除了 relay 尚未读取的行。再过一个月，一个长时间运行的事务很晚才提交，它的行再也没有离开过这张表。

这些故障没有一个出在 outbox 模式本身。它们全都出在把行搬运到 broker 的那一部分。该如何运行这一部分，才能让它不易出故障、恢复迅速，并对它仍可能产生的重复消息保持坦诚？

问题及其难点

transactional outbox 只给出一项保证：如果订单已提交，发布其事件的意图也随之提交。对旅程的后半段，它只字未提。必须有人读取这些行，把它们交给 broker，并记住自己已经这样做过。这个人就是 **relay**，即把 outbox 行搬运到 broker 的进程。

relay 做三件事：找出需要发布的行，发布它们，记录自己已经完成。这是三个独立的步骤，而 broker 与数据库并不共享事务。我们在 producer 里消除的双写问题，在 relay 内部重新出现，只是窗口更窄。

崩溃窗口少到可以逐一列出。这张表就是 relay 的故障语义：什么会失败，如何观察到，以及哪项保证仍然成立。

崩溃发生在	行的状态	结果
发布之前	未发布	这一行会被再次取出，事件发布一次。安全。
发布之后、标记之前	未发布，但事件已在 broker 上	这一行会被再次取出，事件发布两次。一条重复消息。
标记之后	已发布	无事可做。安全。

三个步骤无论怎样排序，都无法关闭中间那一行。如果 relay 先标记、后发布，两者之间发生崩溃就会丢失事件，而这正是 outbox 存在的目的所要防止的故障。所以 relay 先发布、后标记，并接受两者之间的崩溃会产生一条重复消息。这与 outbox 一章所作的权衡相同：at-least-once 投递，把重复消息的处理推给消费者（第 7 章）。

因此，第一个设计问题不是如何避免重复消息。而是如何让窗口保持很小，如何确保行以合理的顺序出去，以及如何在客户察觉之前先发现 relay 正在落后。第二个设计问题是在两种读取行的机制之间做选择：轮询这张表，或者读取数据库自己的事务日志。每一种都把难点移到了别处。

轮询 relay

轮询 relay 按一定间隔查询 outbox 表，发布查到的内容，并把这些行标记为已发布。它是行得通的机制里最简单的一种，除非数字另有说法，我会从这里起步。

基本循环

```
function relayLoop():
    while running:
        // oldest first
        rows = selectUnpublished(limit: 100)
        if rows is empty:
            sleep(pollInterval)
            continue

        // defined below: only broker-confirmed ids
        ackedIds = publishAcked(rows)
        // a crash here leaves confirmed rows published but unmarked:
        // they will be published again, which consumers must tolerate

        // one update for the batch
        markPublished(ackedIds, now())
```

`selectUnpublished` 背后的查询，是大多数轮询 relay 出错的地方，所以值得细看。显而易见的形态是“给我 `id` 大于我上次处理的那一行的所有行”。它看起来高效，却并不安全。

为什么基于 id 的游标不安全

行 ID 通常是在插入执行时分配的，而不是在事务提交时。设想两个 Orders 事务。事务 A 插入一个 id 为 101 的 outbox 行，然后花了一些时间调用一个很慢的函数，之后才提交。事务 B 插入 id 为 102 的行并立即提交。relay 轮询，看到 102，发布它，并把游标移到 102。随后 A 提交。第 101 行现在落在游标之后，永远不会被读到。

这种故障悄无声息。没有错误，没有日志行出现，事件就是永远不会被发布。这正是 outbox 原本要消除的那种丢失，被一项优化重新引入。

我更倾向的修复办法，是不再记住位置，而是改为按状态查询：选出 `published_at` 为空的行，按 `id` 排序。很晚才提交的行，在下一次轮询到来时仍是未发布状态，因此能被找到。代价是需要一个针对未发布行的索引，以及一张必须保持足够小、让这个查询保持廉价的表。如果数据库支持，只覆盖未发布行的部分索引可以让查询保持快速，无论表里保存了多少历史。如果不支持，本章后面的保留工作就更重要。部分索引并不能消除全部代价：在保留旧行版本的数据库里，每一行都会被插入、更新、随后被删除，这种更替会留下死行，需要引擎去清理。

按状态读取解决了丢失问题。它没有解决顺序问题，那是另一个问题，保持顺序一节会回答它。

批处理与轮询间隔

有两项设置决定 relay 的行为：每次轮询取多少行，以及无事可做时睡眠多久。

较大的批次摊薄了查询和标记更新的成本，提高吞吐量。它同时也拓宽了崩溃窗口：relay 在一个 500 行的批次中途死亡，最多会留下 500 个事件需要重新发布。较小的批次在崩溃之后重新发布得更少，并且配合下文所述的行认领，持锁时间也更短。比方说，50 到 200 行是合理的初始猜测；调优之前先测量。

轮询间隔决定了延迟的下限。间隔为一秒时，事件在 relay 看到它之前可能要等上近一秒。缩短间隔会降低延迟，同时增加数据库要应付的空查询数量。对 Parcelly 而言，`OrderPlaced` 从提交到发布之间一两秒的延迟，买家是察觉不到的。对于有人在屏幕前等待的流程，则可能不行。

发布之后再标记

标记这一步值得专门留意。逐行发布，收集 broker 的 ack，只标记得到 broker ack 的行。不要标记尚未收到 ack 的行，即使调用返回时没有错误也不行，因为许多客户端库是异步发送的，失败要稍后才报告。

```
function publishAcked(rows):
    ackedIds = []
    for each row in rows:
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
buildEnvelope(row),
                                    key: row.partition_key)
        if result is failure:
            // stop here: later rows with the same partition_key must
not overtake this one
            break
        ackedIds.append(row.id)
    // only these rows may ever be marked
    return ackedIds
```

这个函数从不触碰表。它返回得到 broker ack 的那些 ID，本章里的每个 relay 都恰好标记这些 ID，不多也不少。失败的那一行以及批次中它之后的每一行都保持未发布，并在下一次轮询时重新读取。发布与标记有意分开，这样标记就可以在它发生的唯一位置由锁或租约来保护。每个事件都发往其聚合类型对应的 topic，以 `partition_key` 作为消息键，与第 4 章使用的路由相同。

`break` 是一个刻意的选择，它关乎消息顺序，下文还会再谈。跳过失败的行并发布它之后的行更快，但可能打乱同一聚合的事件顺序。停下来更慢，却是安全的。我会选择停下来，让监控告诉我 relay 卡住了。为简单起见，这个草图会让整个批次停止；只阻塞受影响的键，是下文保持顺序一节所述的改进做法。

一个活动 relay 还是多个

最简单的拓扑是由一个 relay 实例完成全部工作。它的顺序行为更容易推理，因为它按 `id` 顺序读取，并按同样的顺序发布，不过要注意下文保持顺序一节所讲的 `id` 与提交顺序的区别。它的弱点是可用性：如果这个实例死亡，在另一个启动之前什么都发布不出去。对许多系统来说，几秒内重启是可以接受的，一个配有监督进程负责重启的单 relay 就是正确的答案。

当一个 relay 跟不上，或者发布出现空档不可接受时，就要运行多个。这时出现了新的问题：两个 relay 不得同时发布同一行。标准工具是带跳过语义的行锁。许多关系型数据库提供一种办法，可以选出行用于更新，同时跳过已被其他事务锁定的行，通常称为 `SKIP LOCKED` 式加锁。relay 在一个事务内认领一个批次，其他 relay 就会越过这些行。

```
function claimAndPublish():
    transaction:
        rows = selectUnpublished(limit: 100, lockRows: true,
skipLocked: true)
        ackedIds = publishAcked(rows)
        // unconfirmed rows are not marked; releasing the locks lets
another poll retry them
        markPublished(ackedIds, now())
    // commit point: row locks are released here, and the marks become
visible
```

这样做可行，但它有一些常被忽略的成本。第一，relay 与 broker 通信期间事务一直保持打开，所以 broker 变慢意味着锁被长时间持有，打开的事务不断堆积。在为并发读取者保留旧行版本的数据库里，长时间打开的事务还会阻碍死行的清理，伤害的恰是更替最频繁的那张表。第二，如果事务在发布之后中止，例如连接断开或标记失败，它发布过的每一行都会被再次发布。一种更短的替代做法是：一个简短的认领事务，给这些行盖上认领时间戳；一次不在任何事务之内的发布；以及一个独立的标记事务；再加上一个认领超时，使已死亡的 relay 所认领的行可以重新被认领。它的代价是多出一个需要推理的状态。第三，并行 relay 会失去全局

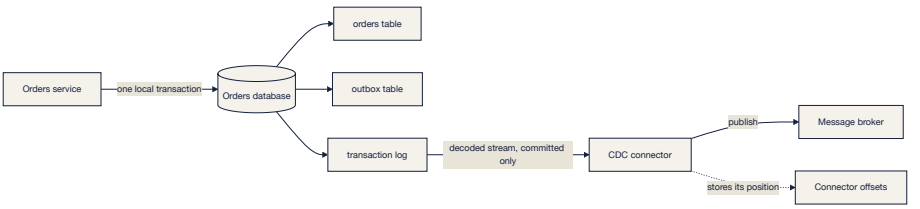
顺序。relay A 可能认领第 1 到 100 行，而 relay B 认领第 101 到 200 行，B 可能先完成。对于关于不同订单的事件，这无关紧要。对于关于同一张订单的两个事件，则很要紧。

补救办法是按 `partition_key` 而不是按行来划分工作。给每个 relay 分配一组键，例如把 `partition_key` 哈希到固定数量的桶中，每个桶在任一时刻恰好交给一个 relay。在桶内，行由一个 worker 按顺序处理，桶与桶之间并行运行。这与基于日志的 broker 上的分区是同一个思路，只是应用在更早的一步。“恰好”一词需要下文故障转移一节中的租约，而在 rebalance 期间的桶移交，正是按键顺序最容易出问题的时刻，因为旧的和新的所有者可能同时为同一个键发布。第 9 章会解释为什么按键的顺序是现实中能够保持的顺序。

变更数据捕获

第二种机制根本不查询 outbox 表。**变更数据捕获（CDC）** 读取数据库自己的事务日志，把已提交的行变更转换为一条变更记录流。CDC connector（Debezium 就是人们熟悉的例子）挂接到该日志上，发现 `outbox` 里插入了一行，并向 broker 发布一条消息。

这张表保留着它的角色，因为事务性保证仍然来自把它与状态变更写在同一个事务里。变化的是由谁来读取它。没有应用进程在轮询。数据库为了自身的恢复与复制，本来就在一份有序的日志里记录每一次已提交的变更，connector 像副本一样读取这份日志。



看一下从事务日志到 broker 的路径。数据库向 connector 暴露的变更流（PostgreSQL 称之为逻辑解码，MySQL 暴露的是它的二进制日志）只包含已提交的事务，而且按提交顺序出现。这就彻底消除了轮询一节中的晚提交问题：日志本身已经是提交的全序，所以晚提交的行会很晚出现，并且位置正确。

它的行为

这个设计带来几项属性。

顺序遵循提交顺序。事件按事务提交的顺序出现。connector 以行的 `partition_key` 作为消息键发布时，在基于日志的 broker 上保持按键顺序，前提是其内部的 producer 经过配置，使 retry 不会打乱同时在途的请求的顺序。应当检查客户端的设置；同样的提醒也适用于异步发布的轮询 relay。

延迟低且稳定。没有轮询间隔。connector 在提交之后很快就能看到变更，所以延迟通常很小，也不取决于有什么东西多久醒来一次。

connector 自己记录位置。它记录自己读到日志的哪里，重启之后从那里继续。connector 通常是定期保存位置，而不是每个事件之后都保存，所以崩溃可能导致上次保存的位置之后的每个事件都被重新发布。CDC 与轮询一样是 at-least-once 的。重复消息的窗口只是移动了，并没有消失。

已发布标志是可选的。由于 connector 不需要去找未发布的行，为了 relay 的缘故，这张表不需要 `published_at` 列。许多使用 CDC 的团队在插入之后不久就删除每个 outbox 行。插入已经记录在日志里，所以 connector 仍然能看到它。表保持得很小，清理问题基本消失。

信封需要整形。原始的变更记录描述的是一行，而不是一个事件。connector 或转换步骤必须把 outbox 的各列映射到标准信封：`event_id`、`event_type`、作为消息键的 `partition_key`、作为消息体的 `payload`。大多数 connector 为此提供了路由和字段映射的支持。应当让 outbox 的列与信封保持对齐，使映射成为机械性的工作。

它的成本

CDC 不是免费的，其成本体现在运维上。

现在需要运行一个 connector，通常运行在某个 connector 框架里，还要加上它所依赖的 broker 基础设施。必须有人去了解它是如何失败的。

数据库必须把它的日志暴露给 connector，这通常意味着启用某个特定的日志级别，并创建一个复制位置，由数据库代 connector 保留。如果 connector 停了而没有人注意到，数据库会保留着无法丢弃的日志段，磁盘就会写满。这是 CDC 这一类技术里众所周知的故障形态，具体细节因数据库而异。我会把“connector 宕机了”视为需要呼叫值班人员的事件，因为损害并不局限于 broker。

对 outbox 表以及 connector 所监视的任何表做 schema 变更，都需要事先规划。列重命名或类型变更可能破坏映射。

初始设置、快照以及长时间中断之后的 connector 重启，同样值得演练。其行为取决于产品和数据库，我不会凭一张图就想当然。

> **注意**：CDC 是一族技术，各产品在读取日志的方式、存储位置的方式以及长时间中断之后的处理上各不相同。应当把本节视为一般形态，并阅读所用 connector 和数据库的文档，了解所运行版本的确切行为。

日志消失之后

“磁盘写满”是一项权衡的一半，另一半更糟。数据库要么为 connector 的位置无限期保留日志，这会写满磁盘；要么限制保留的量，或者按计划让日志过期，这会保护磁盘。许多数据库允许自行选择。如果设了上限，而 connector 宕机的时间超过了上限，它保存的位置就指向已经不存在的日志。缺口里的事件要么丢失，要么 connector 必须从快照重新开始。

重新快照会对 outbox 造成什么后果，取决于更早的一项决定。如果这张表仍然保存着行，快照会把它们全部重新发出，于是每个保留的行对每个消费者而言都成了重复消息。如果像 CDC 中常见的那样，行在插入后立刻被删除，快照什么也找不到，缺口里的事件根本无法从表中恢复。这是一次静默丢失：没有错误，没有重复消息，只有一张从未抵达 Inventory 的订单。让表保持很小的那次删除，同样也删掉了证据。

数据库主库的故障转移带来第三种情形。connector 的位置能否延续到新的主库，因数据库和版本而异，我会通过演练一次故障转移来弄清楚，而不是读一张图。

由此得出三个习惯。有意识地选择保留上限，把它当作磁盘与所能承受的中断时长之间的权衡，并把这个数字写下来。对余量设置告警，也就是在 connector 的延迟已经用掉了多少保留窗口、尚未耗尽之前就告警。并且在中断发生之前就决定，如何从缺口中恢复事件，例如通过对账任务（第 13 章），把订单状态与已发布的内容作比较。第 14 章讲述损害已经发生之后的恢复。

在轮询与 CDC 之间选择

抽象地说，两种机制没有哪个更好。下表展示它们的差异所在。

关注点	轮询 relay	CDC connector
活动部件	一个自己拥有的小进程	connector、它的框架、数据库日志设置
延迟	下限由轮询间隔决定	通常低且稳定
顺序来源	查询顺序加上自己的小心	来自日志的提交顺序
晚提交的事务	用基于状态的查询是安全的	由日志处理
数据库负载	重复查询、行锁	日志读取、保留的日志段
崩溃后的重复消息	可能，窗口是未标记的批次	可能，窗口是未保存的位置
表的增长	需要做保留工作	行可以立即删除
伤害最大的故障	静默的延迟、卡住的行	停滞的 connector、被保留的日志
所需技能	普通的应用技能	数据库日志内部机制、connector 运维

对刚起步的团队，我的默认选择是轮询 relay。它只有几十行代码，团队已经知道如何运行和调试这样的服务，它的故障模式在它所查询的那张表里清晰可见。当以下三件事之一成立时，我会转向 CDC：轮询的延迟过高，而不频繁冲击数据库就无法降低；outbox 的体量使轮询查询成为可测量的成本；或者已经运行着一个 CDC 平台，并且有人负责它。

我不会仅仅因为 CDC 听上去更严肃就采用它。一个从未运维过复制日志的团队，把订单流程放在上面，就是拿一个自己理解的问题换来了一个自己不理解的问题。反过来也成立：已经为了其他目的在运行 CDC 的团队，再另外写一个轮询 relay，收益很小。

运维 relay

选择机制是工作的前一半。后一半是让它保持健康的工作。

保留与清理

轮询 relay 会留下已发布的行。放任不管，这张表会无限增长，查询未发布行会变慢，数据库还要在没人读取的行上耗费精力。清理必须经过设计，而不是事后追加。

有两项选择很重要：已发布的行保留多久，以及如何删除它们。保留一段时间是有价值的。当消费者报告缺少某个事件时，outbox 是首先查看的地方，用来弄清它是否曾被发布过。比方说，几天是合理的起点。多长才

合适，取决于团队多快能察觉到事件缺失，这是一个关于人和告警的问题，而不是关于存储的问题。

```
function purgePublishedRows():
    while true:
        deleted = deleteRows(
            where: published_at is not null
               and published_at < now() - retentionPeriod,
            limit: 1000
        )
        // small batches keep locks short and replication lag low
        if deleted is 0:
            return
        sleep(shortPause)
```

应当分小批次删除，使这个任务不会长时间持锁，也不会冲垮副本。如果表很大，且数据库支持，按时间分区并整体丢弃分区，比逐行删除更便宜。无论怎么选，清除条件必须包含 `published_at is not null`。只按年龄删除的清理任务，会在 relay 落后的那一天删掉从未发布过的事件。那就是开篇场景里的故障，它是数据丢失，而不是重复消息。

> **警告**：切勿仅按年龄清除 outbox 行。relay 卡住的时间超过保留期限，它的未发布行就会被删除，这些事件永远消失。只清除已标记为已发布的行，并对变旧的未发布行设置告警。

使用 CDC 时，清理有所不同，但并非不存在。行可以在插入之后立刻删除，所以表保持很小。现在增长的是被保留的日志，需要盯住的是 connector 相对于它的延迟。立即删除还意味着重新快照什么也恢复不了，正如前面关于日志消失的一节所解释的，所以应当在立即删除与对已发布行做短期保留之间权衡。

监控 relay

停止工作的 relay 从外部是看不见的。Orders 照常成功，outbox 照常接受行，而其他每个服务都悄悄地得不到事件。没有人收到错误。这就是 relay 需要自己的信号的原因，我会在上线之前把其中三个就位。

1. **最旧未发布行的年龄**。这是最好的单一数字，因为它衡量的是用户所体验到的：最慢的事件已经等了多久。当它超过与业务而不是与技术相关联的阈值时，应当告警。如果买家的确认邮件应在一分钟内到达，那么最旧未发布行的年龄达到五分钟就是一次事故。

2. **未发布行的数量**。积压在增长，说明 relay 比 producer 慢，即使最旧的那一行还很新。
3. **发布错误率和批次耗时**。它们解释前两项。每次轮询都 retry 同一行的 relay，会表现为错误数稳定而年龄增长。

对于 CDC connector，同样的问题以不同的单位出现：connector 的位置落后日志头部多远，它是否在运行，以及数据库为它保留了多少日志。第 13 章会回到这些信号，也会回到对账，后者能抓住延迟指标漏掉的东西。

故障转移与重复发布

relay 必须能经受住自己的重启，这包括两个实例短暂同时运行的情形。部署、让旧实例看起来像已死亡的网络分区以及在旧进程退出之前就重启进程的编排器，都会造成这种重叠。

对于单活动 relay，应使用租约：数据库中的一行或一把锁，活动的 relay 以较短的间隔续期，持有者停止后它就会过期。备用实例在过期之后接管。绊倒人的细节在于，旧的持有者可能不知道自己的租约已经过期。它可能因为一次很长的垃圾回收或网络停滞而暂停，醒来后发布一批已不属于它的数据。

```
function relayWithLease():
    while running:
        // returns an increasing number
        token = acquireOrRenewLease(ttl: leaseTtl)
        if token is none:
            sleep(standbyInterval)
            continue
        // small: the batch must fit inside the lease
        rows = selectUnpublished(limit: 20)
        ackedIds = publishAckedWithin(rows, token)
        // marking rows the broker confirmed is true even if the lease
lapsed meanwhile
        markPublished(ackedIds, now())

function publishAckedWithin(rows, token):
    ackedIds = []
    for each row in rows:
        if leaseRemaining(token) < safetyMargin:
            // stop before publishing, not after: a lapsed lease
cannot be undone
            break
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
buildEnvelope(row),
                                    key: row.partition_key)
        if result is failure:
            break
        ackedIds.append(row.id)
    return ackedIds
```

递增的 token 是 fencing token，这个思路只有在受保护的资源会检查它时才有效。数据库可以检查：带着过期 token 的提交可以被设计为失败，第 7 章对消费者和 API 正是这样使用的。broker 通常做不到，所以 token 无法阻止过期的 relay 发布。标记则是另一回事。过期的持有者只标记得到 broker ack 的行，所以它的标记是真实的，回滚这些标记只会造成更多重复消息。租约必须限制的是并发发布，因为两个持有者同时发布会产生重复消息，并且可能把一个键较旧的行排到较新的行之后。

有四件事可以限制它。让租约 TTL 远高于一个批次的最坏耗时，并保持批次较小，因为租约每个循环只续期一次，仅一个变慢的 broker 就可能超过它。在每次发布之前检查剩余租约，如草图所示。在基于日志的 broker 支持按 epoch 的 producer fencing 之处加以使用，让 broker 自己拒绝旧的持有者。并且当旧的持有者醒来时，让它先通过询问数据库来发现这一点，在一个锁住租约行的事务里进行，然后才可以再发布任何东西。不持有锁的检查就是先检查后执行，会以同样的方式失败。

这就是诚实的极限。故障转移可能产生重复发布，没有任何租约方案能消除它。租约带来的是窗口既短又罕见。真正消除危害的，是把第二份副本视为同一个事件的消费者，依据的是两次发布中保持稳定的 `event_id`。因此 relay 必须从 outbox 行取得 `event_id`，发布时绝不能生成新的。每次发送都铸造一个新标识符的 relay，会把无害的重复消息变成两个看似不同的事件。有些 broker 接受 producer 提供的去重 ID，并在一个时间窗口内丢弃重复项；如果所用 broker 支持，应将其设为 `event_id`。它免费缩小了窗口，但并不取代消费者的 inbox。第 7 章讲消费一侧。

保持顺序

outbox 对事件顺序的保持，只到 relay 为止，而 relay 只能保持表里所保存的顺序。第 4 章说明过，`id` 顺序不是提交顺序：一个事务可以先取得一个 `id`，在锁上等待，然后在取得更晚 `id` 的事务之后才提交。`id` 是插入的顺序，而不是提交的顺序。

对轮询 relay，由此得出两点。如果同一个键较小的 `id` 在较大的 `id` 已被发布之后才提交，这两条会反着发出，而挽救了晚到行的那个基于状态的查询，也正是让它晚发布的原因。如果两者在同一次轮询中同时可见，而 ID 是颠倒的，它们会在一个批次里反着发出。因此，“按 `id` 顺序”发布的 relay，并不会按聚合变化的顺序投递每个聚合的事件，除非以下两个条件之一成立。

第一个是 producer 在插入 outbox 行之前先锁定聚合行，使同一个键的事务串行化，它们的 ID 遵循提交顺序（第 4 章）。第二个是顺序由数据携带：`aggregate_version` 列，消费者会比较它，第 9 章会用到它。我会两者都做。锁是对 relay 廉价的保险，而版本是在 relay 之后的一切环节中都能留存的东西。CDC connector 避免了第一个问题，因为日志按提交顺序排列，但它对跨分区的情形仍然没有帮助，所以版本依然有用。

即便如此，relay 自己的选择仍然很重要。下面每一项都可能破坏它所拥有的顺序：

- 跳过失败的行，发布同一个键之后的行。
- 运行并行的 relay 却不按键分区。
- 带外 retry 失败的发布，使较晚的行先到达。
- 对同一个聚合的事件使用不同的消息键。

安全的默认做法是按键的队首阻塞：如果 `order-81724` 的某一行无法发布，就停止发布 `order-81724` 之后的行，让其他键的行继续。这会给被阻塞的那一个聚合带来延迟，并保护它的序列。如果问题是一行永远无法发布，例如序列化器拒绝的 payload，就必须有人去修复它，而上文的监控正是让他们发现这一点的办法。

要弄清前面那个草图做了什么。对整个批次使用 `break`，再加上最旧的优先的查询，一个无法发布的行会在每次轮询中最先被返回，并永远卡住每一个键。按键阻塞需要两项补充，下面以对 `publishAcked` 的改进形式给出，它取代原来的 `break`：relay 记住本批次中哪些键失败了，反复失败的行则被暂存，使查询不再返回它，也不再返回它所属键之后的行。

```
function publishAcked(rows):
    ackedIds = []
    blockedKeys = emptySet()
    for each row in rows:
        if row.partition_key in blockedKeys:
            // an earlier row of this key failed: do not overtake it
            continue
        result = publishAndAwaitAck(topicFor(row.aggregate_type),
buildEnvelope(row),
                                    key: row.partition_key)
        if result is failure:
            blockedKeys.add(row.partition_key)
            // attempts + 1; failed_at = now() after N attempts
            recordFailure(row.id)
            continue
        ackedIds.append(row.id)
    return ackedIds
```

随后的查询会排除任何有暂存行的键，因此该键之后的行要等待，而其他键的行继续流动。暂存的时刻，正是第 4 章的 `failed_at` 列派上用场的时刻，它还要与一个 `attempts` 计数器配合，使瞬时故障在行被暂存之前先 retry 几次。暂存行就是设置了 `failed_at` 的行。应当对每个暂存行告警，因为暂存行就是一个被阻塞的聚合，必须有人来修复或丢弃它。代价是两个额外的列、一个更难建索引的查询以及一套必须在第一次事故之前就由人写好的修复流程。

顺序在 relay 之后，在消费者和 broker 中同样会丢失。尽到本分的 relay，提供的是一个值得保持的序列，而不是它能撑过路径其余部分的保证。第 9 章会沿着从 broker 到处理器的路径追踪顺序。

权衡与成本

两种机制都是在简单性与运维触达范围之间做权衡，并且都留下了一个无法关闭的重复消息窗口。

轮询 relay 付出数据库负载和延迟的代价，换来的是仍然停留在团队已经理解的代码之内。它的弱点是针对不断增长的表的未发布行查询、多个 relay 运行时的锁行为以及无人盯着最旧行时的静默延迟。如果系统以适中的速率发布，且一秒的延迟可以接受，那么在拥有成本上，轮询 relay 很难被超越。

CDC 付出一个新组件和新技能的代价，换来更低的延迟、按提交顺序读取以及一张保持很小的表。它的弱点是：停滞的 connector 使数据库保留日志，或者在 connector 停滞时被限制容量的日志过期并丢失事件；会破坏映射的 schema 变更；以及很少被演练的恢复流程。如果没有人愿意负责 connector，那么无论架构图看上去多么诱人，我都不会采用它。

关于何时完全不使用 outbox relay：如果事件丢失也没有后果，例如一条丢失了也没有成本的信息性通知，那么提交之后直接发布可能是可以接受的，而且更便宜。如果事件必须反映某个数据库中的状态，而该数据库不提供横跨状态与 outbox 的本地事务，relay 无法弥补这一点，需要另一种设计。

两种机制有两项共同的成本。两者都不提供 exactly-once 发布：发布与标记之间、或发布与保存位置之间崩溃所造成的重复消息，是设计的一部分。而且如果 relay 与 producer 对信封的看法不一致，两者也都帮不上忙，所以要让 outbox 的列与信封字段保持同步。

常见错误

- **使用基于 id 的游标而不是基于状态的查询**。这很诱人，因为它避免了扫描未发布的行。晚提交的事务于是落在游标之后，永远不会被发布。应当按 `published_at is null` 查询，或使用已经按提交顺序排列的、基于日志的机制。
- **在收到 broker 的 ack 之前标记行**。先标记看起来能消除重复消息。它把一次崩溃变成事件丢失。应当发布，等待 ack，然后标记。
- **在 relay 中生成新的事件标识符**。这感觉无害，因为发送的正是 relay。消费者的 inbox 于是无法再把重新发布的事件识别为同一个事件。应当从行中复制 `event_id`。

- **仅按年龄清除**。它让表保持很小，并且在 relay 健康的每个测试中都能通过。它在 relay 不健康时会删除未发布的事件。应当只清除已发布的行，并对变旧的未发布行设置告警。
- **为了维持吞吐量而跳过失败的行**。同一个聚合的后续行于是超车。应当改为按键阻塞，暂存反复失败的行，并修复原因。
- **认为 `id` 顺序就是提交顺序**。在每个单写入者的测试中它都成立。在并发写入者之下，较小的 `id` 可能更晚提交，relay 会把这一对反着发布。应当先锁定聚合行再插入 outbox，并携带 `aggregate_version`。
- **相信租约能对 broker 起到 fencing 作用**。租约失效之后醒来的过期 relay 仍然会发布，因为 broker 不检查 token。应当让批次保持在 TTL 之内，在每次发布之前检查租约，其余交给消费者吸收。
- **运行 CDC 却没有人选择过保留上限**。默认值感觉只是个细节，直到 connector 宕机了一个周末，位置也没了。应当选定上限，对余量设置告警，并清楚重新快照会对 outbox 造成什么后果。
- **运行多个 relay 却不按键分区**。行锁防止重复劳动，却对顺序只字不提。应当把键的桶分配给单个 worker。
- **不监控 relay**。outbox 让 producer 变得可靠，这使 relay 的故障成为仅剩的那一种，而已停止的 relay 看起来恰如一个安静的系统。应当对最旧未发布行的年龄设置告警。
- **把 connector 中断当作 broker 问题**。在 CDC 中，停滞的 connector 会让数据库保留它的日志。应当为此呼叫值班人员，并且要盯着磁盘，而不只是延迟。

要点

- relay 做三件彼此独立的事，即查找、发布和标记，而 broker 与数据库不共享事务，所以发布与标记之间的崩溃总会留下可能的重复消息。
- 先发布，后标记：反过来会把崩溃变成事件丢失，而这正是 outbox 存在的目的所要消除的故障。
- 轮询 relay 应当按状态选出未发布的行，而不是按递增的 id 游标，因为事务提交的顺序与它们分配 ID 的顺序不同，即便如此，按聚合的顺序也只有在 producer 先锁定聚合并携带 `aggregate_version` 时才成立。

- 要运行多个轮询 relay，应当用 SKIP LOCKED 式加锁认领行，并按 `partition_key` 分区，使同一个聚合的行始终由一个 worker 按顺序处理。
- 变更数据捕获按提交顺序从数据库日志读取已提交的变更，消除了轮询和晚提交问题，代价是一个 connector、一项在写满磁盘与丢失事件之间的日志保留权衡，以及团队必须自己掌握的技能。
- 除非延迟、负载或现有的 CDC 平台另有说法，应当从轮询 relay 起步，并把这一选择视为简单性与运维触达范围之间的权衡。
- 应当监控最旧未发布行的年龄，只清除已发布的行，暂存永远无法发布的行，从 outbox 行复制 `event_id`，并预期故障转移之后会出现重复消息，因为租约只是限制了并发发布，只有幂等的消费者才能让其余部分变得无害。

relay 现在至少把每个事件投递一次，顺序值得保持，有时会投递两次。下一个问题位于 broker 的另一侧：消费者的工作失败时应当怎么办，第 6 章用 retry 来回答。

第三部分

可靠地消费

第 6 章 — 正确地重试

设想 Parcelly 的一个周二上午，卖家促销刚刚开始。Orders 发出一条 `CapturePayment` 命令，Payments 把它转交给外部支付服务商。服务商不堪重负，响应缓慢。Payments 的调用开始超时，于是 Payments 让这条消息失败，broker 随即重新投递。每个 Payments 实例都在同一时刻做着同样的事，等待的也是同样固定的 1 秒。

一分钟之内，服务商收到的不再是促销期间的正常负载，而是正常负载加上所有失败的调用，一轮又一轮地同步涌回。服务商的缓慢原本可能只持续几秒，现在却要一直持续到 retry 不再到来为止。与此同时，Payments 的消费者忙于对注定不会成功的消息执行 retry，健康的消息只能排在它们后面。在这堆消息里，有几笔扣款其实已经在服务商处成功，却正在被再次发送。

这里没有人写过一行糟糕的代码。每个实例都做了合理的事：失败，等待，再次尝试。损害来自一千个无人协调的合理决定。

本章要做的，是让 retry 成为一项决策，而不是一种本能反应。本章回答的问题是：调用失败时，如何决定是否再次尝试、何时尝试、尝试多少次、在哪一层尝试，才能让 retry 帮助系统恢复，而不是妨碍恢复？

问题及其难点

retry 是再次尝试的承诺，而不是成功的承诺。本章中，一次**尝试**指任何一次试行，因此尝试次数等于 1 加上 retry 的次数。显而易见的做法是“把调用包进循环，retry 几次”，它因四个各不相同的原因而失效，而且这些原因会叠加。

第一，**并非每种故障都值得 retry**。因银行卡被拒而遭驳回的请求，第十次尝试同样会被拒。对它执行 retry 只会浪费容量，并推迟有人察觉真正问题的时刻。retry 循环必须知道自己面对的是哪一类故障，而大多数循环并不知道。

第二，**retry 会在最糟糕的时刻增加负载**。依赖在过载、降级或重启时失败。retry 是额外的请求，而且恰好指向那个依赖。如果每个调用方都 retry 3 次，也就是总共 4 次尝试，那么一个本已吃力的服务，在最无力承受的时候，会看到多达正常流量 4 倍的流量。

第三，**retry 会同步**。一起失败的调用方会一起 retry。固定延迟会把一次故障变成周期性的尖峰，因为所有调用方在同一瞬间醒来。这就是惊群效应，也是延迟不仅需要长度、还需要随机性的原因。

第四，**retry 会重复一个首次结果可能未知的操作**。第 3 章已经说明，超时并不能说明请求是否已被处理。因此，超时之后执行 retry，就有把工作做两遍的风险。没有幂等性（第 7 章），retry 就是一条迟早会出现的重复消息。

本章其余内容，都是在回答这四个问题之一。分类回答第一个问题。带抖动的退避与 retry 预算回答第二个和第三个问题。幂等性由第 7 章构建，回答第四个问题；后文谈缺少幂等性的一节，解释了为何不能跳过它。

故障分类

在决定如何 retry 之前，先决定是否 retry。第 1 章给出了针对任何接缝都应提出的三个问题：什么会失败，如何观察到它，哪项保证在失败之后依然成立。这些答案合起来就是一种故障的语义，而 retry 决策就是把这些语义付诸应用。我发现，依据“再次尝试是否可能有帮助”这一点，把每种故障归入三类之一很有用；然后再加上第四种信号，它对消息本身什么也没有说明。

瞬时故障

瞬时故障指其原因预期会自行消失的故障。连接被重置、超时、`503 Service Unavailable`、数据库中的死锁牺牲者、限流响应、broker 中的领导者选举。请求本身并没有错。只是外部世界暂时挡了路。

正确的应对是 retry，并且要带延迟，因为正是延迟让原因得以消除。

永久故障

永久故障指每次尝试都会重现的故障，因为原因存在于请求或数据之中。未通过校验的 payload；对某条记录的引用，在它早该到达之后很久仍然缺失（在事件驱动系统中，缺失的记录可能是瞬时的，因为创建它的事

件可能迟到；第 9 章讨论这种情况）；`400 Bad Request`；因余额不足而被拒的银行卡；消费者无法理解其 schema 版本的事件。

retry 是徒劳的，有时还有害。正确的应对是停止，记录原因，并把消息送到人工或修复作业能够处理的地方。永久故障的去向见第 8 章。尤其要注意银行卡被拒：对 Payments 而言，这根本不是错误，而是预期中的业务结果，会变成一条 `PaymentFailed` 事件，不涉及任何 retry。

未知故障

未知故障指无法判断其类别的故障。超时是典型的例子，带有无人预料到的异常信息的意外异常也是。超时可能是瞬时的。异常则可能是消费者中的 bug，每次尝试都会失败。

未知是一个真实的类别，而不是分类体系中的空缺，如何对待它是一项策略决定。我的默认做法是把未知故障当作瞬时故障，但限制比已知的瞬时故障更严格，因为错误地对未知故障执行 retry，代价是几次浪费的尝试，而错误地丢弃它，代价是丢失工作。各层的尝试次数都用尽之后，未知故障按永久故障处理：连同完整的历史进入死信。

当一次尝试的结果未知时，应先查询，再行动。在 retry 之前，尤其是在补偿之前，应当用幂等键向服务商查询该操作的状态。如果扣款已经发生，正确的后续动作是退款，而不是作废。如果没有发生，retry 或作废都是安全的。无论往哪个方向猜测，买家都可能就此被扣下一笔 Parcelly 以为已取消的订单的款项。

延后：已知依赖不可用

第四种信号不是对消息的裁决。circuit breaker（见下文）处于打开状态时，调用失败并不能说明请求本身如何：调用根本没有到达依赖。我把这一类称为**延后**。延后的消息不消耗尝试次数，不计入任何上限，也绝不会因此转入死信队列。消费者停止拉取，或暂停分区，消息留在原处等待，直到再次调用该依赖变得值得为止。

这一区分之所以重要，是因为否则结果只是一道算术题。一个保持打开 30 秒的 breaker，若位于一个允许处理器内 3 次尝试、每次间隔 1 秒的处理器之前，就会让一条健康的消息在大约最初 3 秒内耗尽全部预算，并因一个与它无关的问题落入 DLQ。

把分类明确写出来

分类必须存放在可以阅读和评审的地方。把故障信号映射到类别和处理方式的表，是消费者中最有价值的小制品之一。对于 Payments 调用服务商的场景，一个示意性的版本如下：

信号	类别	处理方式
连接被重置、DNS 故障	瞬时	带退避执行 retry
扣款调用超时	未知	使用相同的幂等键执行 retry，严格限制次数
`429`、限流响应	瞬时	执行 retry；若服务商给出等待提示，则遵从
服务商返回 `5xx`	瞬时	带退避执行 retry
`400`、校验失败	永久	不执行 retry，转入死信队列
银行卡被拒	永久，业务结果	发出 `PaymentFailed`，不执行 retry
无法识别的错误	未知	严格限制次数地执行 retry，然后转入死信队列
circuit breaker 处于打开状态	延后	不消耗尝试次数；暂停消费，待 breaker 允许时恢复

有两点需要提醒。第一，状态码只是提示，而非保证；对格式错误的请求返回 `500` 的服务商，等于替调用方把它错误分类了，这一点只能靠与该服务商打交道的经验来了解。第二，对于表中从未见过的信号，默认值应当是未知，绝不能是永久或瞬时。一个无人分类过的新错误，不能证明任何事情。

分类器本身很小：

```
// business outcomes such as a card decline are handled by the caller
// before this function is reached; they are not errors
function classifyFailure(error):
    if error is ValidationError:
        return "permanent"
    if error is CircuitOpen:
        // the call never reached the dependency: no verdict on the
message
        return "deferred"
    if error is ConnectionError or error is Throttled:
        return "transient"
    if error is Timeout:
        // the request may or may not have been processed
        return "unknown"
    return "unknown"
```

带抖动的指数退避

一旦决定 retry，下一个问题就是何时。立即 retry 不会给瞬时的原因留出消除的时间。固定延迟之后再 retry 能留出时间，却会让所有调用方保持步调一致。

指数退避让延迟随每次尝试增长：比如先 1 秒，再 2 秒，再 4 秒，直到某个上限。早期的 retry 很快，适合转瞬即逝的故障。后期的 retry 很慢，能减轻真正不可用的依赖所承受的压力。上限则避免延迟增长到超出业务所能容忍的程度。

抖动为每次延迟加入随机性。没有抖动，同一时刻失败的调用方会算出相同的延迟，步调一致地 retry，惊群在退避之后依然完好无损。有了抖动，retry 会分散到一段区间内，依赖看到的是一片零散的请求，而不是一堵墙。

抖动有多种方案，其中两种值得记住名称。在 **full jitter** 中，延迟是 0 与该次尝试的指数上限之间的随机值。在 **equal jitter** 中，延迟的一半固定，一半随机，因此最短等待时间永远不会接近 0。full jitter 分散负载的效果最好，而 equal jitter 保证等待时间有一个下限。我会从 full jitter 开始，只有当依赖受困于过快的 retry 时才改用 equal jitter。

```
function computeDelay(attempt, baseDelay, maxDelay):
    // the ceiling doubles with each attempt, capped at maxDelay
    ceiling = min(maxDelay, baseDelay * 2 ^ (attempt - 1))
    // full jitter: anywhere between zero and the ceiling
    return randomBetween(0, ceiling)

function callWithRetry(operation, policy):
    attempt = 0
    while true:
        try:
            return operation()
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent":
                raise error
            if failureClass == "deferred":
                // no attempt is counted; the consumer pauses, not
retries
                raise error
            attempt = attempt + 1
            if attempt >= attemptLimit(failureClass, policy):
                raise RetriesExhausted
            sleep(computeDelay(attempt, policy.baseDelay,
policy.maxDelay))

function attemptLimit(failureClass, policy):
    // unknown failures get the stricter limit promised above,
    // counted over all attempts so far, whatever their class
    if failureClass == "unknown":
        return policy.maxAttemptsWhenUnknown
    return policy.maxAttempts
```

这个循环对它所处理的故障是正确的，但在两个方面并不完整，下面几节会逐一解决。它不知道调用方还剩多少时间，也不知道整个系统已经在执行多少 retry。

如果服务商给出了何时再次尝试的提示，例如限流响应中的等待值，应当遵从。服务商比调用方自己的公式更清楚自己如何恢复。应把该提示当作计算所得延迟的下限，而不是替代品，并保留抖动，使收到相同提示的调用方不会同时回来。下限仍不是休眠到截止时间之后的许可：如果提示比剩余时间更长，就应停止 retry，交由下一层决定。

截止时间与 retry 预算

带抖动的退避决定 retry 的间隔。另有两项限制决定 retry 最多可以有多少次。

截止时间

截止时间是指过了这一时刻，结果就不再有用。由请求驱动的流程有一个天然的截止时间：买家正盯着结账页面，10 秒之后就会离开。后台消费者的截止时间则更为宽松：在承运商截单时间之后才到达的运单标签，与根本没有到达的一样迟了。

只计尝试次数而忽略时间的 retry 逻辑，可能远远越过有用的时间点。3 次尝试，每次超时 30 秒，中间还有不断增长的延迟，就是数分钟的等待。我倾向于给每个要执行 retry 的操作一个总时间预算，只要下一次延迟将越过它，无论尝试次数怎么说，都停止 retry。

```
function callWithDeadline(operation, policy, deadline):
    attempt = 0
    while true:
        remaining = deadline - now()
        if remaining <= 0:
            raise DeadlineExceeded
        try:
            // never let one attempt outlive the whole budget
            return operation(timeout: min(policy.attemptTimeout,
remaining))
        on error as error:
            failureClass = classifyFailure(error)
            if failureClass == "permanent" or failureClass ==
"deferred":
                raise error
            attempt = attempt + 1
            delay = computeDelay(attempt, policy.baseDelay,
policy.maxDelay)
            // the next attempt needs time to run, not only time to
start
            needed = delay + policy.minimumUsefulAttempt
            if attempt >= attemptLimit(failureClass, policy) or needed
>= deadline - now():
                raise RetriesExhausted
            sleep(delay)
```

截止时间还应当随调用传递。如果 Orders 给某个流程 10 秒并调用 Payments，Payments 应当把剩余的时间继续传下去，而不是自己重新开始一个 10 秒。否则每一层都在上一层的耐心之内执行 retry，总和会悄悄超过用户愿意等待的时间。

Retry 预算

单次调用的尝试上限，无法在许多调用同时失败时保护依赖。每个调用都停留在 3 次 retry 的上限之内，依赖收到的仍是正常流量的 4 倍。

retry 预算按占总流量的份额来限制 retry，而不是按单次调用。常见的形式是 token bucket：每次成功的调用往桶里加入一小份 token，每次 retry 消耗一个 token，桶空之后，调用不再 retry，而是快速失败。稳态下，retry 只占请求的一小部分。出现大范围宕机时，预算会在片刻之内耗尽，系统也就不再放大自身的故障。

具体数值由使用者自行选择，取决于依赖能吸收多少额外负载。我不会给出数字，因为任何数字都只是对某个具体依赖的猜测。重要的是预算要存在，要在一个进程内的各调用方之间共享，而且预算耗尽要能在仪表盘上看到。按进程维护的、基于比例的预算通常就够了，因为每个进程各自限制自己的比例，也就限制了总量。跨实例共享的预算是一种选择，而不是严格意义上的改进：它需要分布式计数器，而这个计数器是故障路径上的一个新依赖，有它自己的故障模式。

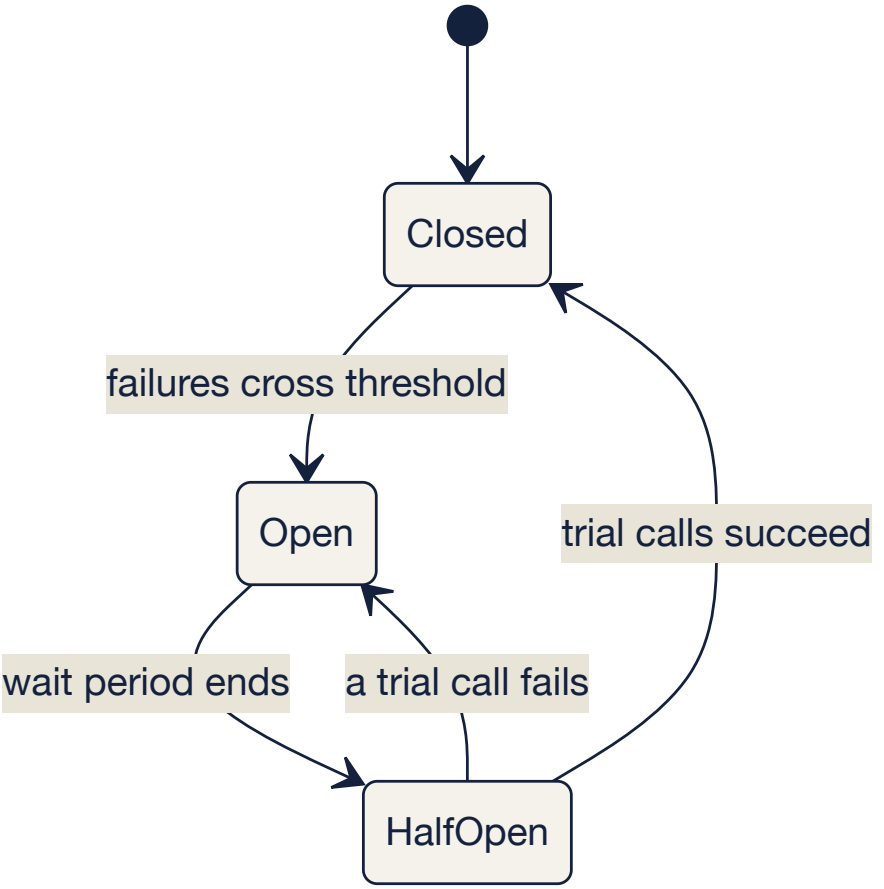
代价是，一些本可通过 retry 成功的调用会转而失败。这正是预算的意义所在：它用个别调用的成功换取系统的存续，而且应当是有意为之。

Retry 风暴与 circuit breaker

retry 风暴是指 retry 成为失败依赖上的主要负载时发生的情形。本章开头 Parcelly 的场景就是一例。它需要三个要素：一个缓慢或出故障的依赖、会执行 retry 的调用方以及没有任何机制通知调用方停下来。层层叠加会让情况更糟。如果 Orders 对 Payments 做 3 次尝试，Payments 又对服务商做 3 次尝试，一次失败的扣款就可能变成对服务商的 9 次调用。再加一层 3 次尝试，就是 27 次。

有两种防御直接针对风暴。上文的 retry 预算限制系统会执行多少 retry。circuit breaker 则限制系统是否还应当调用。

circuit breaker 监视对某个依赖的调用结果，有三种状态。正常运行时它处于关闭状态，调用直接通过。当失败越过阈值，它就打开，调用立即失败，不再触及依赖，让依赖有空间恢复。等待一段时间之后，它转入半开状态，放少量试探调用通过。如果这些调用成功，它就关闭；否则再次打开。



breaker 与 retry 策略互为补充，回答的是不同的问题。退避决定一次调用在下一次尝试之前等待多久。breaker 则根据各次调用呈现的模式，判断尝试是否还值得进行。处于打开状态的 breaker，会把 retry 循环的下一次尝试变成瞬间失败。这种失败既不是瞬时的，也不是未知的：它就是上文分类中的**延后**信号。retry 循环不得为它计数，不得对它退避，也不得在它上面空转。它把控制权交还给消费者，由消费者决定停止做什么。

消费者拥有请求驱动的调用方所没有的一项额外选择。breaker 打开时，消费者应当停止从 broker 拉取消息，或暂停分区，而不是把消息消费下来再逐条令其失败。留在 broker 中的消息，在其保留期限允许的范围内是安全的，它们会占用存储，并表现为 consumer lag。被消费、执行了 retry 又失败的消息，会耗掉自己的尝试次数，还可能因一个与自己无关的问题而转入死信队列。暂停是 breaker 在消费者一侧的形式，我会把它排在本章大多数其他事项之前去构建。

暂停会引出两个后续问题。第一，没有流量通过时，就没有任何东西来测试半开状态。应当以并发度 1 恢复，或发送合成的 probe 调用，使试探调用由知道自己在试探的主体来发出。第二，在基于日志的 broker 上，已暂停的消费者必须保持轮询循环存活——暂停分区，但不暂停轮询——否则 consumer group 会认定该消费者已死亡，并重新分配它的分区（见下文所述的存活窗口）。

breaker 的代价在于它可能误判。阈值过于敏感，会因一次短暂波动而打开，把瞬时故障变成宕机。阈值过于宽松，则打开得太晚。breaker 还增加了状态，凌晨三点某个流程停摆时，必须有人读懂这些状态：首先要问的是，依赖是真的不可用了，还是只有 breaker 处于打开状态。

过载下的 retry：反压

breaker 对已经失败的依赖作出反应。**反压**是针对更早阶段的一套做法：依赖缓慢、饱和或正在恢复，系统必须决定交给它多少工作。第 3 章把它放在故障模型之内，本节讲的是其机制。

回到开头的场景。服务商的缓慢并非源于某个 bug，而是源于超出其吸收能力的工作量，而每个 Payments 实例仍在不断向它输送。对在途工作没有限制的消费者，行为正是如此：不停地拉取，不停地发起调用，不断给本已过重的堆积再添负担。下面这些限制，是我会落实的，大致按这个顺序。

- **限定在途工作**。限制消费者同时持有的消息数量。在基于队列的 broker 上，这是 prefetch 数量或在途数量。在基于日志的 broker 上，则是每次轮询获取的记录数加上正在运行的处理器数。没有上限时，消费者的内存和 broker 的重新投递计时就成了限制，而这两者都比自行选定的限制更糟。
- **依赖变慢时降低并发度**。每次调用 50 毫秒时合适的并发度，在每次调用 5 秒时就不合适了，因为同样数量的槽位现在容纳的等待中工作要多出许多倍。延迟上升时应降低并发调用数，延迟恢复后再缓慢地提高。
- **暂停，而不是失败**。依赖饱和时停止拉取，做法与 breaker 打开时相同。暂停的消费者表现为 lag，这是可见且安全的。拉取了又失败的消费者则会耗掉尝试次数。
- **优先服务实时流量，而非 retry**。retry 与 redrive 同首次尝试争夺同一个依赖。容量紧张时，应当让首次尝试获胜，因为 retry 是系统已

经知道在吃力的工作。让 retry topic 和 redrive 的消费者以较低的并发度运行，或在主流程有积压时让它们等待。

- **在入口处削减负载**。上游某处必须存在对已接受工作的限制，否则在依赖恢复期间，队列会无限增长。应当拒绝或推迟新工作，并向调用方给出明确信号，而不是无限期地把它排在一个无法服务的依赖后面。

这些限制与 retry 预算是同一个思想的两个侧面。预算限制 retry 增加多少额外工作，反压则限制在途的工作总量，不论是否额外。单靠任何一个都不够：只有预算而没有在途上限，缓慢的依赖会让每个槽位都被首次尝试占满；只有在途上限而没有预算，槽位则会被 retry 占满。

代价是 lag 和拒绝。削减负载的系统会有意对一些调用方说不，暂停的系统则会让一些消息延迟投递。两者都是用吞吐量换取存续，而“哪些工作要等待”这个问题的答案，归流程的业务负责人所有，而不是调校数字的工程师。第 8 章把同样的速率限制思想用于 redrive，第 9 章把在途上限用于按键划分的通道。

Retry 位于何处

retry 可以发生在多个层次上，各层的行为并不相同。若在不自知的情况下混用，一条消息最终执行 retry 的次数就会远超任何人的预期，或者根本不执行。

应用层 retry

消费者在处理器内部执行 retry：即上文的代码，包裹在对服务商的调用之外。retry 运行期间，消息保持未 ack 状态。这一层成本低、速度快，适合在毫秒或秒级内消除的故障。它的局限在于：等待期间会占用消费者的容量；消费者崩溃会丢失 retry 状态（broker 会重新投递，尝试次数从头开始，这正是让进程崩溃的消息需要第 8 章所述检测手段的原因）；长时间等待有超出 broker 自身超时的风险。

在基于队列的 broker 上，未 ack 的消息会在 **visibility timeout** 期间保持隐藏，之后被重新投递。如果处理器内的 retry 持续时间超过该超时，broker 会在第一份副本仍在 retry 时重新投递，于是两份副本同时在运行。在基于日志的 broker 上，两次轮询之间耗时过长的消费者，可能被其 consumer group 视为已死亡，其分区也随之重新分配，结果相同。我遵循的规则是：处理器内 retry 的最长总时间必须稳稳地落在 broker 的超时之内，更长的等待则应放到其他层次。

Broker 重新投递

消费者让消息失败，broker 再次投递。两类 broker 在这里有所不同，而且这个差异很重要。

在基于队列的 broker 上，重新投递是内建的。消费者对消息发送 nack，或任由 visibility timeout 到期，消息就会回到队列，立即回去或在配置的延迟之后回去，任何竞争消费者都可能收到它。

基于日志的 broker 不会自行重新投递。消费者的读取位置随轮询向前移动，已提交的 offset 只记录重启后的消费者将从哪里恢复（第 2 章对此有说明）。因此，处理某条记录失败且不提交其 offset，本身不会产生任何效果：消费者会读取下一条记录，如果随后提交，这次提交也会覆盖失败的那条记录，它就悄无声息地被跳过了。失败的记录只有在重启、rebalance 或消费者采取明确步骤之后，才会再次被读取。

因此，在日志上，消费者必须在三种动作中选择其一。它可以回退 offset 至失败的位置，或暂停该分区，并原地 retry，这样能保持顺序，但会使其后的一切停滞。它可以把记录发布到 retry topic 然后继续前进。或者把它写入死信 topic 然后继续前进。每一种都是消费者代码中的一项有意决定，而不是 broker 替它完成的事。

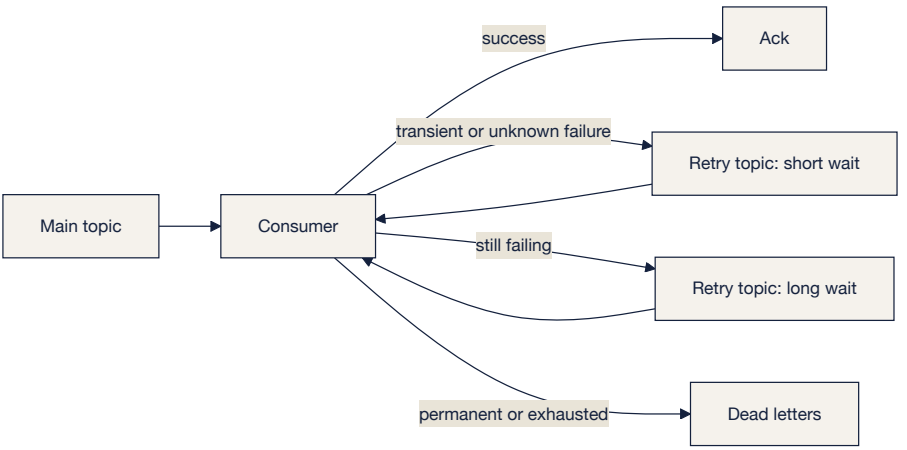
broker 重新投递是持久的：消费者崩溃后依然有效。它也很粗糙。许多 broker 会立即重新投递，或按固定时间表重新投递，没有退避，而尝试次数可能只能以投递计数器的形式获得，需要消费者自己读取并据此行动。应把它当作其他各层之下的安全网，而不是主要的 retry 策略。

Retry topic 与延迟队列

当故障所需的等待超出处理器内 retry 所能承受的时间，常见的结构是把消息移出主流程，稍后再让它回来。让消息失败的消费者，会把它连同递增后的尝试次数和一个“不早于”时间，发布到 **retry topic**（或 retry 队列），并对原消息发送 ack。另一个消费者，或同一个消费者在另一个 topic 上，在时间到来时将它取走。支持延迟投递或按消息设置延迟的 broker，可以直接提供这种能力。不支持的话，常见的安排是一组梯级式的 retry topic，每个 topic 有固定的等待时间，对应退避的每一步。

每个 topic 的等待时间固定，是有原因的。在日志上，消费者无法跳过尚未到期的记录：它必须在 head 处等待，直到该记录准备好。如果一个 topic 中的每条记录等待时间相同，这个 topic 本身就已按到期时间排好

序，在 head 处等待不会带来额外成本。混合了短延迟和长延迟的 topic，则会让短延迟的记录排在长延迟的记录后面。按消息设置“不早于”时间，只有原生支持延迟投递的 broker 才可行。



这一层让主流程得以解放，失败的消息不再拖住它后面的消息。它能挺过重启，因为状态存放在 broker 中。它还提供了一个可供观察的地方：retry topic 的深度，直接度量了有多少工作在等待一个正在恢复的依赖。

它需要更多的机制。要创建、命名、监控和保护更多的 topic。重新发布本身就是一次可能失败的发布操作，这又带回了第 1 章的双写问题，所以顺序很重要：先发布到 retry topic，再对原消息发送 ack。如果进程在两步之间死亡，消息就会存在两份。这就是一条重复消息，消费者本来就必须容忍它。

三条规则可避免梯级反过来损害它本应保护的东西：

- **每个消费者拥有自己的 topic**。retry topic 只属于一个 consumer group 或订阅。如果另一个 group 也能读取它，那个 group 就会重新处理一条它从未失败过的消息。
- **保持身份不变**。重新发布的消息保留原来的键、`event_id` 和业务 header。新的键会破坏顺序，新的 `event_id` 会破坏 inbox。
- **按等待时间确定保留期限**。retry topic 上的保留期限必须超过最长等待时间加上预期在其上出现的 lag，否则消息会在等待期间到期。

最重要的成本是**消息顺序**，下一节会专门讨论。

选择层次

实践中，消费者会三者并用。简短的应用层 retry 吸收一两秒内即可消除的短暂波动，例如连接重置或死锁。retry topic 或延迟队列覆盖需要数分钟的故障。broker 的重新投递覆盖崩溃。死信队列（DLQ）终结这条链，接收那些已耗尽尝试次数或永远无法成功的消息；它的设计见第 8 章。

各层应当一起设计，共用一个总的尝试预算，而不是三个相互独立的预算。如果每一层允许 3 次尝试，一条消息就可能累计尝试 27 次，而没有人计划过这一点。本章末尾的策略把这种相乘关系明确写出，并设置了上限。

Retry 与消息顺序

对一条消息执行 retry，会改变各消息的处理顺序。这是上述分层的必然结果，在造成损害之前往往不被察觉。

以 Parcelly 的 Inventory 消费者为例，它会依次收到同一订单的 `OrderPlaced` 和 `OrderCancelled`。假设处理 `OrderPlaced` 时发生瞬时失败，消费者把它移到 retry topic，然后继续处理 `OrderCancelled`。取消操作发现没有可释放的预留，因为就 Inventory 所知，预留还不存在。随后，重新处理的 `OrderPlaced` 到达，创建了一条再也没有任何东西会去释放的预留。最终状态是错误的，而每一个单独的步骤看起来都很合理。

有两种诚实的选择，各有代价。

- **原地阻塞并 retry**。在基于日志的 broker 上，消费者回退 offset 至失败的位置，或暂停该分区，并在处理其后任何内容之前，带退避地对其执行 retry。该分区的顺序得以保持。代价是整个分区都要等在一条消息后面，因此一条失败的消息就可能使该分区上的每个聚合都陷入停滞。当顺序比吞吐量更重要时，这是正确的选择，这也是迅速分类永久故障如此重要的原因：位于分区头部的 poison message 会阻塞它后面的一切。
- **挪到一边并接受乱序**。失败的消息进入 retry topic，消费者继续前进。吞吐量得以保持。代价是消费者必须容忍乱序到达，办法包括版本检查，或者在较早的消息成功之前，把同一聚合的所有后续消息也暂存起来。第 9 章讨论这些技术。

我会按 stream 来选择。对于同一聚合的消息必须按顺序应用的事件 stream，我会原地 retry，并投入精力做快速分类和死信出口。对于由相互独立的事实构成的 stream，消息之间的顺序无关紧要，挪到一边成本低，还能让流程保持流动。

在没有分组功能的基于队列的 broker 上，竞争消费者之间的顺序本来就很弱，所以 retry 只是让一项既有的特性稍微变差，而不是破坏某项保证。即便如此，假定有顺序的消费者也不应指望队列来提供顺序。若采用分组功能（FIFO 队列、消息组、会话），权衡与日志上相同：失败的消息会阻塞它所在的组，直到成功或转入死信队列，就像原地 retry 会阻塞一个分区一样。

没有幂等性的 retry

每次 retry 都是对某项工作的重复尝试，而此前的结果可能未知。如果先前的尝试在做任何事之前就失败了，retry 是无害的。如果它在做完工作之后才失败，retry 就会把工作做两遍。

这并非罕见的边缘情况。回到开头的场景。Payments 调用服务商，扣款 4,990 个 EUR 最小货币单位。服务商完成扣款并发出响应，响应却丢了。Payments 看到的是超时。该超时被归为未知，策略规定要 retry。第二个扣款请求发了出去。买家被扣一次还是两次，此刻完全取决于有没有什么东西能让第二个请求被识别为重复请求。

这个东西就是**幂等键**：随请求一同发送的稳定标识，使接收方能够认出自己见过这项工作。对于服务商调用，同一次扣款的每次尝试都使用同一个键，认可这类键的服务商会把它们视为一次扣款。对于消费事件，它通常是 `event_id`，在与该工作相同的事务中对照 inbox 进行检查。第 7 章会构建这两者。

> **警告**：retry 策略只有建立在幂等操作之上才是安全的。在对任何会产生效果的调用（例如银行卡扣款、发送邮件或预留库存）启用 retry 之前，应当先确定是什么让第二次尝试无害。如果没有任何东西能做到这一点，自动 retry 的正确次数就是零，故障应当交给人处理。由于 at-least-once 的 broker 在崩溃或超时之后，无论处理器是否愿意都会重新投递，“零”必须由处理器来强制保证：它必须检测到自己见过这条消息，或者在产生效果之前先 ack 并接受 at-most-once，然后把故障转交给人。

这种依赖是双向的。幂等性让 retry 变得安全，而 retry 让幂等性成为必需。一个激进地执行 retry、消费者却不幂等的系统，会以与其故障率成正比的速率产生重复的效果，也就是恰好发生在系统已经承压、无人有空清理的时刻。

还有一种更隐蔽的情形。一个操作对同一个键可以是幂等的，却仍然不能安全地 retry，因为键变了。如果处理器在每次尝试时都构造一个新的标识，例如每次调用都用一个新的 UUID，服务商每次看到的都是不同的请求。键必须只派生一次，来源是 `event_id` 或订单这类稳定的输入，并在每次尝试中重复使用。

Parcelly 的 retry 策略

为了把各个部分汇总起来，下面是我会为 Payments 中处理 `CapturePayment` 的消费者写下的内容。其中的数字只是示意，而非建议，具体的依赖需要有自己的数字。

```
retryPolicy for CapturePayment:
    // illustrative values
    attemptTimeout:   5 seconds
    inHandler:
        maxAttempts:            3
        maxAttemptsWhenUnknown: 2
        baseDelay:              200 milliseconds
        maxDelay:               2 seconds
        jitter:                 full
    retryTopics:
        steps:        [1 minute, 5 minutes, 30 minutes]
        appliesTo:    transient and unknown failures; the deadline
caps both
    // after this, a capture is stale
    totalDeadline:    2 hours
    // all layers and redeliveries together
    maxTotalAttempts: 8
    retryBudget:      per process, visible on a dashboard
    circuitBreaker:   pause consumption when open; deferred calls cost
no attempt
    onExhausted:      dead-letter with the full attempt history
    onStale:          ask the provider for the capture's status, then
                      void if not captured, refund if captured
    idempotencyKey:   derived once from command_id, reused on each
attempt
```

应把它读作一组决策，每项都有理由。处理器内这一层吸收短暂波动。retry topic 覆盖宕机数分钟的服务商。截止时间说明这项工作在何时不再值得做，而这一决定的负责方是业务，而不是工程师。一笔过期的扣款

可能需要变成一条 `OrderCancelled`，但补偿取决于服务商处实际发生了什么。如果最后一次尝试超时，扣款可能已经成功，此时正确的后续动作是 `PaymentRefunded`，而不是 `PaymentVoided`。这就是策略要先查询状态的原因。只派生一次的键，才是让整个结构不会向买家重复扣款的关键。

策略中有两行值得再看一眼。`maxTotalAttempts` 之所以存在，是因为各层会相乘：处理器内 3 次尝试，乘以一次首轮加 3 个 retry topic 步骤，对服务商最多是 12 次调用，这还没有算上崩溃后的任何重新投递。一个总数 8（示意值）才是有人真正负责的数字。此外，整个 retry 的时间跨度（这里是两小时）必须短于服务商记住幂等键的时间。服务商保留键的窗口有限，窗口之后到达的 retry 是一个携带旧键的新请求。

当第 8 章的 redrive 把一笔已成为死信的扣款放回时，尝试计数器会重新开始，但另有一个独立的 redrive 计数器保持不变，因此历史不会丢失，反复回来的消息也能一眼认出来。

值得注意的是缺少了什么：单一的“retry 次数”设置。在分层的系统里，这一个数字至少隐藏了四项决策。

权衡与成本

retry 是提升系统表面可靠性成本最低的办法之一，也是最容易过度使用的办法之一。它们的成本是具体的。

延迟。每次 retry 都会增加等待时间。退避和抖动能分散负载，也会拉长成功所需的时间。一个 retry 数分钟的流程可能最终正确完成，但对 10 秒后就已离开的买家而言仍然毫无用处。有人在等待的场景中，我更倾向于快速失败并给出明确的提示，而不是长时间无声地 retry。

放大。retry 会成倍放大负载，分层的 retry 会再次成倍放大。在每个服务中合理的策略，放在整条调用链中可能就是危险的。必须有人负责端到端的尝试预算，因为没有任何单个服务能看到它。

容量。原地 retry 的消费者在等待期间占用着自己的槽位。长时间故障下，所有消费者都可能在等待，系统甚至对本可成功的工作也不再有进展。通过 breaker 暂停消费、用反压限制在途工作以及把长时间的等待移到 retry topic，是常用的防御手段。

隐藏的故障。最终成功的 retry 从外部看就是成功。如果不统计 retry 和尝试次数，依赖可能在悄悄吸收它的策略背后降级数周。第 13 章讲述应度量什么：尝试次数、retry 率和仍在 retry 的最老消息的年龄应当出现在仪表盘上。

复杂性。retry topic、延迟队列和预算都是必须有人去构建、命名、监控和解释的基础设施。对于只有一个消费者、一个依赖的小系统，带退避的处理器内 retry 加上死信出口可能就足够了，一组梯级式的 retry topic 可能超出问题所需的机制。

何时不要 retry。不要对永久故障执行 retry。不要对不幂等且有副作用的操作执行 retry。不要越过截止时间继续 retry。当 circuit breaker 表明依赖已宕机时，不要 retry，也不要为这条消息消耗一次尝试。不要在已处于另一层 retry 之内的层次里再执行 retry，除非已经算清了相乘的结果。也不要把 retry 当作回避决策的手段：不断重现的故障本身就是信息，正确的应对往往是停下来查看。

常见错误

对所有故障采用同一种 retry

这样做很有诱惑力，因为一个循环是最短的代码。它把银行卡被拒与连接被重置同等对待，把容量浪费在不可能成功的故障上。应先分类，再为每个类别设定各自的处理方式。

固定延迟而没有抖动

恒定的等待感觉可预测且简单。它会把一起失败的所有调用方同步成反复出现的尖峰。应使用带抖动的指数退避，而且在我看来，full jitter 是合适的起点。

只统计尝试次数，不统计时间

尝试上限看起来像是一道安全边界。在超时很长、延迟不断增长的情况下，它仍可能在已无用处之后继续运行数分钟。应为整个操作设置截止时间，并在下一次延迟将越过截止时间时停止。

每一层都执行 retry

每一层的负责人都添加了一个合理的 retry，而相乘的结果对他们所有人都是不可见的。应为整条调用链确定一个总的尝试预算，并把 retry 放在成本最低的地方。

Breaker 打开时仍然执行 retry

继续运行循环很诱人，因为这个循环原本是有效的。消费者对着一个已宕机的依赖拉取消息并令其失败，会消耗这些消息的尝试次数，还可能错误地将它们转入死信队列。应把打开的 breaker 视为不消耗尝试次数的延后信号，暂停消费，让消息留在 broker 中等待。

让消费者无限制地拉取

无界的消费者是默认状态，而且感觉上吞吐量最大。面对缓慢的依赖，它会让每个槽位都被等待中的工作占满，并助长 retry 风暴。应限定在途消息，延迟上升时降低并发度，并让首次尝试优先于 retry。

忽视 broker 的超时

处理器内的 retry 感觉上只是处理器的局部事务。如果它们持续的时间超过 visibility timeout 或 group 的存活窗口，broker 就会重新投递或重新分配，于是同一项工作的两份副本同时运行。应让处理器内的总时间远远落在 broker 的限制之内。

越过必须保持顺序的消息去 retry

把失败的消息移到 retry topic，是处理停滞分区的快速办法。它会让该聚合的事件乱序。应先确定顺序要求，再在阻塞和挪到一边之间选择。

对非幂等的效果执行 retry

添加 retry 循环只需一个下午，而幂等性被推后了。成功调用之后的第一次超时就会产生重复的效果。应在启用第一次 retry 之前，先确定是什么让第二次尝试无害。

每次尝试都生成新的键

这看起来很勤勉，因为每个请求都有唯一标识。它却违背了目的，因为接收方无法认出重复请求。应从稳定的输入中只派生一次键，并重复使用。

不统计 retry

如果只记录最终的失败，那么在 retry 不再够用之前，依赖的降级是看不见的。应记录每一次尝试，连同其类别和延迟。

要点

- retry 是再次尝试的承诺，而不是成功的承诺，因此面对任何故障，第一个问题都是再次尝试是否有可能有帮助。
- 把故障分为瞬时、永久或未知，将分类写成一张表，并把无法识别的故障当作未知处理，施以更严格的限制。
- 使用带抖动的指数退避，用截止时间限制整个操作，并共享 retry 预算，使 retry 不会成为失败依赖上的主要负载。
- circuit breaker 决定是否还要调用；打开的 breaker 是不消耗尝试次数的延后信号，对消费者而言，最强的形式是暂停消费。反压在更早的阶段完成同样的工作，办法是限定在途工作，并让实时流量优先于 retry 和 redrive。
- retry 可以位于处理器内、broker 的重新投递中和 retry topic 里，应在一个尝试预算之下一起设计，并让处理器内的时间保持在 broker 的超时之内。
- 把失败的消息挪到一边会让其聚合的事件乱序；应根据 stream 的顺序要求，在原地阻塞与容忍乱序之间选择。
- retry 策略只有建立在幂等操作之上才是安全的，幂等键只派生一次，并在每次尝试中重复使用。

retry 决定了再次尝试的频率，却无法决定尝试两次是否无害。第 7 章构建另一半：幂等的消费者，它把重复消息从一场灾难变成一件平淡无奇的小事。

第 7 章 — 幂等性

设想 Parcelly 的 Ledger 服务正处于滚动部署之中，消费者被逐个重启。某个消费者读到一条 `PaymentCaptured` 事件，写下一条 4,990 个 EUR 最小货币单位的账本记录，正要对消息发送 ack，进程却收到了关闭信号。这条 ack 没能离开本机。broker 恰恰按 at-least-once 投递所承诺的那样，把同一条消息交给下一个来要消息的消费者。

第二个消费者读取事件，没发现任何异常，又写下第二条账本记录。账本是仅追加的，按其构造本应永远正确，于是它如实且永久地记录下：买家付了两次款。什么都没有崩溃，没有任何告警响起。每个组件都按设计运行，账目却错了。

Notifications 也有同一个下午的另一个版本。买家收到了三封“您的包裹已排期”邮件，因为一个缓慢的处理器超出了 visibility timeout，消息被重新投递。这件事令人尴尬，而账本的那件事代价高昂。

前面的章节已经确立，重复是常态：broker 会重新投递，relay 在故障转移后会重复发布，第 6 章的 retry 本身就是按设计产生的重复。本章回答这些章节留下的问题。同一条消息到达两次时，消费者究竟应当怎么做，才能让第二次到达不改变任何东西？

问题及其难点

目标听起来很朴素：每条消息只处理一次，至少在外界看来如此。难点在于，消费者不靠记忆就无法知道自己是否见过某条消息，而记忆本身就是一次写入，可能恰好在最糟糕的时刻失败。

先看重复从哪里来，因为这份清单比 broker 要长。

来源	重复如何出现
Broker 重新投递	ack 丢失，visibility timeout 到期，或者在 offset 提交之前 rebalance 就把分区挪走了
Relay	relay 发布了一行，在将其标记为已发布之前崩溃，重启后又发布了一次（第 5 章）

来源	重复如何出现
应用层 retry	消费者对一次调用执行了 retry，或者生产者对一次发布执行了 retry，而首次尝试其实已经成功（第 6 章）
Redrive 与重放	有人把死信移回，或有意从更早的 offset 重新读取事件日志
人与客户端	买家双击了“支付”，移动应用在不稳定的网络上执行了 retry，webhook 发送方投递了两次

其中多数情形产生的是*同*一条消息的两份副本，带有相同的 `event_id`。少数情形产生的是表达同一意图的两条不同消息，这一区别在选择键时会很重要。

诱人的办法是先查后做。先问这条消息是否已经处理过，没有的话再处理。我想把它命名为**先检查后执行**反模式，因为它感觉正确，却悄无声息地失效。

```
function handlePaymentCaptured(message):
    // UNSAFE: the check and the action are separate steps
    if inboxContains("ledger-poster", message.event_id):
        ack(message)
        return

    postLedgerEntry(message.payload)
    inboxInsert("ledger-poster", message.event_id)
    ack(message)
```

在故障下阅读它。如果进程在 `postLedgerEntry` 之后、`inboxInsert` 之前死亡，账本记录已经存在，而 inbox 并不知道，于是重新投递会再记一次。把两次写入的顺序对调，故障也随之反转：在 `inboxInsert` 之后、账本写入之前崩溃，会把尚未处理的消息标记为已处理，其效果就永远丢失了。两种顺序都不安全，因为这两次写入不是一个原子动作。

还有第二种故障，与崩溃毫无关系。两个消费者可能几乎在同一时刻收到同一条消息，例如缓慢的首次尝试超出了它的 visibility timeout。两者都执行检查，都没发现什么，于是都继续往下做。“检查”与“执行”之间的窗口很小，但在足够大的流量下，会被例行地撞上。

所以真正的问题有两半。关于“处理过什么”的记忆，必须与效果原子地一起写入，并且这份记忆必须能在并发的尝试之间做出裁决。本章其余部分将构建这两者，先从两者都不需要的情形开始。

天然幂等与强制幂等

一个操作，若应用两次后系统所处的状态与应用一次相同，就是**幂等**的。有些操作因其形态而天然幂等，在构建任何机制之前，我会先找出这些操作。

设置一个值是幂等的：`set order.status = "PAID"` 可以运行 10 次，订单仍然是 `PAID`。按 id 删除一行是幂等的。插入一行，其主键由事件派生，并忽略冲突，也是幂等的。写入聚合的完整新状态（“运单现在是 `DISPATCHED`，物流单号为此”）是幂等的。我把这称为**天然幂等**：重复没有额外的效果，是由操作本身的性质决定的，而不是由额外添加的任何东西决定的。

其他操作则不然。`balance = balance + 4990` 应用两次就是两笔存款。`send email` 会发出两封邮件。`append a ledger entry` 会追加两条记录。`increment the counter of units sold` 会把这件商品数两次。不带键去调用支付服务商的扣款接口，可能会扣两次款。对于这些操作，幂等性必须另行加上，我称之为**强制幂等**：记录这项工作已经完成，并在再次执行之前查阅该记录。

天然幂等有两个诚实的局限。第一，它悄悄依赖于顺序。设置 `status = "PAID"` 是幂等的，但一个较早事件的延迟重复可能覆盖较新的状态：重新投递的 `OrderPlaced` 在 `OrderConfirmed` 之后到达，把状态设回了 `PLACED`。操作是幂等的，系统仍然是错的。第 9 章讨论顺序和版本检查；眼下只需注意，“设置为某个值”只有在该值不会过期时，才能抵御重复。

第二个局限是，人很容易高估它。一个“只更新一行”的处理器，往往还会写一条审计记录、发出一个后续事件，或者给某个计数器加一，而这些每一项都是伪装起来的增量。评审消费者时，我会列出它的每一个效果，并逐一追问：它能否经受第二次运行。只要有一个效果不幂等，就需要 inbox。

我的经验法则是：效果允许时，优先选择天然幂等，并在处理器中写明它为何如此。当效果是计数器、追加、邮件或外部调用时，使用强制的那种。有疑问时，仍然使用 inbox：它的成本小而可预测，而出错的代价则不然。

选择幂等键

幂等键是消费者用来识别重复工作的依据。在建表之前，必须先决定表里放什么，这个选择决定了能捕获哪些重复。

event_id

对于事件消费，默认选择是信封中的 `event_id`。它在每个事件上唯一，而且关键在于，**在重新投递时保持稳定**：broker 交付的消息的每一份副本都带有相同的值。生产者在事件写入 outbox 时只分配它一次（第 4 章），因此发布了两次的 relay 会把同一个 `event_id` 发布两次。这恰恰是去重检查所需要的性质。

`event_id` 标识的是一条*消息*。它回答的是“我以前处理过这个具体的事实吗？”，仅此而已。

业务键

有时，两条不同的消息描述的是同一个意图。假设 Orders 在超时后对 `CapturePayment` 命令执行 retry，而由于某个 bug 或某项设计选择，每次 retry 都被写成带有全新 `command_id` 的新命令。Payments 现在收到两条 id 不同的命令，它们都表示“为订单 81724 扣款”。按消息 id 去重，两条都捕获不到。

补救办法是**业务键**：由工作的含义派生、而不是由信封派生的标识符。对扣款而言，它可以是 `payment_id` 加上操作，比如 `capture:payment-5521`。对账本记录而言，它可以是支付 id 加上记录种类。业务键表达的是“这一单位的工作只能发生一次”，不论有多少条消息要求它发生。

方面	`event_id` / `command_id`	业务键
能否捕获同一条消息的重新投递	能	能
能否捕获表达同一意图的两条消息	不能	能
由谁分配	生产者，自动完成	每个操作由人来决定
风险	漏掉逻辑上的重复	键过粗会拒绝合法的工作
存放位置	通用的 inbox 表	通常是领域表上的唯一约束

倒数第二行中的风险，值得用一个具体例子说明。如果退款的业务键只是 `payment_id`，那么对同一笔支付的第二次、合法的部分退款，就会被当作重复而拒绝。键必须包含使两个请求*确实不同*的一切：在这里，也许是由调用方提供的退款幂等键。选择键，意味着要决定“同一项工作”在所属领域里指什么，这是一项设计决策，而不是框架的设置。

在我看来，这几层配合得很好。把通用 inbox 中的 `event_id` 作为每个消费者的基线，并在任何两条不同消息有可能表示同一件事的地方，于领域表上加一个业务键唯一约束。inbox 抵御 broker 和 relay。约束则抵御系统的其余部分，包括它自身的 bug。

Inbox 模式

inbox 是消费者一侧已处理消息 id 的表。第 4 章为生产者提供了事务性 outbox，使状态变更和它的事件一起提交。inbox 是它的镜像：消费者在自己数据库的同一个本地事务中，把状态变更和“已处理该消息”的记录一起提交。

```
table inbox:
    consumer      text not null
    event_id      uuid not null
    processed_at  timestamp not null
    primary key (consumer, event_id)
```

`consumer` 列让一个服务可以针对同一张表运行多个相互独立的消费者，使 Ledger 的记账消费者和报表消费者各自记住自己的历史。这就是主键为 `(consumer, event_id)` 这一对的原因。若只用 `event_id`，第一个处理该消息的消费者会让第二个消费者的插入失败，第二个消费者的效果就会被当作重复而跳过。一张表只对应一个消费者的服务，可以去掉这一列，直接以 `event_id` 为键，这是本书其他地方使用的更简单的形式。

处理器在同一个事务中写入 inbox 行和业务变更。

```
function handlePaymentCaptured(message):
    if inboxContains("ledger-poster", message.event_id):
        // fast path only; this check is not the safety net
        ack(message)
        return

    transaction:
        postLedgerEntry(message.payload)
        // primary key (consumer, event_id): a concurrent duplicate
        // fails here, the transaction rolls back, and the
        // message is redelivered and finds the committed row
        inboxInsert("ledger-poster", message.event_id)
    // commit point: effect and inbox row exist together or not at all

    ack(message)
```

逐一检视各种故障。如果进程在提交之前死亡，账本记录和 inbox 行都不存在，因此重新投递会从头处理这条消息。如果它在提交之后、ack 之前死亡，两者都存在，重新投递走快速路径：检查发现了这一行，消息被 ack，什么也不写入。效果与它的记忆永远不会不一致，因为它们作为一个整体提交。

顶部的检查是一种优化。它在常见的重复情形中省下一个事务，去掉它也不会损失正确性。真正的安全网是事务内的唯一约束，真正的裁决就发生在那里。

在事务内部，写入的顺序并不改变原子性，所以前面关于对调两次写入的警告，只适用于上面那个不安全的版本。顺序确实会改变成本。在这个版本中，并发的重复会先做完全部工作，然后才在插入时失败，而在某些数据库中，一次违反约束会中止整个事务，处理器因而无法在事务内恢复。我会先认领：以“on conflict do nothing”的语义插入 inbox 行，查看是否真的插入了一行，没有的话就跳过效果。这样重复者就在键上等待，不做任何工作，也不会抛出异常。下一节的示例使用这种形式。

关于 `ack` 要说明一句，因为这个术语掩盖了两个家族之间的差别。在基于队列的 broker 上，它对一条消息发送 ack。在基于日志的 broker 上，它通常是提交一个 offset，意思是“到这里为止的一切”，而并行处理某个分区的消费者可能会越过一条尚未完成的消息去提交，并因此丢失它。inbox 抵御的是重复，而不是这种情况，第 9 章在顺序与并行交汇之处讨论它。

消费者发出的后续事件

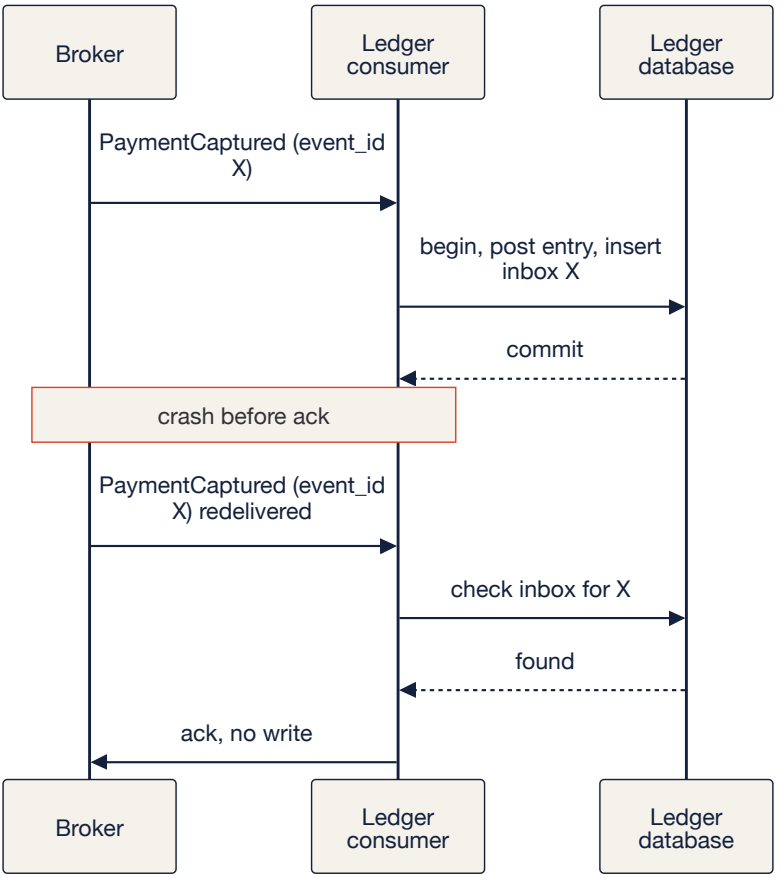
许多消费者做的不止是改变自己的状态：它们还会宣布自己做了什么。Ledger 记下一条账本记录，并发出 `LedgerEntryPosted`。这个后续事件与其他任何效果一样，必须经受住崩溃，且恰好发生一次，不能是零次，也不能是两次。

在提交之后直接发布后续事件的处理器，在崩溃时会永久丢失它。重新投递走 inbox 的快速路径，发现了这一行，发送 ack，并跳过发布。状态是对的，事件却从未存在过。解决办法是把第 4 章的 outbox 应用到消费一侧：inbox 行、状态变更和 outbox 插入在一个本地事务中一起提交。

```
function handlePaymentCaptured(message):
    transaction:
        claimed = inboxInsertIfAbsent("ledger-poster",
message.event_id)
        // on conflict do nothing: a concurrent duplicate waits on the
key,
        // then sees claimed = false and does no work
        if claimed:
            postLedgerEntry(message.payload)
            outboxInsert({
                event_type:     "LedgerEntryPosted",
                causation_id:   message.event_id,
                partition_key:  message.partition_key
                // remaining envelope fields as in Chapter 4
            })
    // commit point: inbox row, ledger entry and follow-up event, or
none of them
    ack(message)
```

如果进程在提交之前死亡，什么都不存在，重新投递会把这一切都做一遍。如果它在提交之后死亡，后续事件就留在 outbox 中，由 relay 将其发布，而重新投递发现了 inbox 行，什么也不做。消费者的 outbox 与生产者的 outbox 具有同样的 at-least-once 发布，因此消费 `LedgerEntryPosted` 的一方需要它自己的 inbox。第 10 章在某个 saga 步骤对一个事件作出反应并发出下一个事件时，建立在这条规则之上。

这也是 inbox 位于与其所保护的状态*同一个数据库*中的原因。如果已处理消息的记录放在缓存或另一个数据库中，我们就又回到了没有共享事务的两次写入，也就是第 4 章的双写问题在消费者一侧的翻版。inbox 不是一种可以从服务数据存储之外附加上去的技术。



看中间的那条注释：崩溃窗口位于提交与 ack 之间，而 inbox 行正是把重新投递变成空操作的东西。

并发重复

上面的时序假定重复在第一次尝试结束之后才到达。并发更难，而起到拯救作用的是唯一约束。

设想两个消费者同时处理同一条消息。两者都开始一个事务，都做了工作，都尝试插入 inbox 行。数据库只允许其中恰好一个提交；另一个收到唯一约束违反，其事务回滚，它的账本记录也随之消失。落败者随后要么不 ack 该消息，要么重新检查后再 ack，无论哪种，账本里都只有一条记录。

整个机制就是这样，值得说明它不需要什么：不需要分布式锁，不需要两个消费者之间的协调，也不需要对时序做任何假设。数据库的约束是单一的裁决点，两次尝试都必须从它通过。

```
function handleSafely(message):
    try:
        transaction:
            applyChange(message)
            inboxInsert("ledger-poster", message.event_id)
        // commit point; a losing concurrent attempt rolls back here
        ack(message)
    on UniqueViolation as e:
        if e.constraint == "inbox_pkey":
            // someone else committed this message first; the work is
done
            ack(message)
        else:
            // a business constraint failed: a real error, not a
duplicate
            raise e
```

唯一约束违反应当 ack 还是 retry，是一个值得慎重作出的小决定。如果违反发生在 inbox 的键上，说明该消息已经由一个已提交的事务处理过，ack 它是正确的。如果违反发生在 `applyChange` 内部的其他约束上，这个错误的含义就完全不同，ack 它会掩盖一个真正的 bug。在我看来，处理器应当区分这两者，办法是检查被违反的约束名称，上面的代码正是这样做的。

较长的处理器与在途工作

上面的模式把整个处理器放在一个事务里。当工作只是几次写入时，这没有问题。当处理器要调用某个缓慢的东西时，就令人不安了，因为事务一直开着，锁一直被持有，而 visibility timeout 可能在等待期间到期。到期的消息被重新投递给第二个消费者，后者并行地开始同样缓慢的工作。

有两种合理的应对。第一种是让事务保持简短：把缓慢的外部工作放在事务*之外*，采用下一节的外部效果做法，只把本地状态变更和 inbox 行放进事务。第二种适用于工作确实必须被认领的情形，即通过一个状态使认领显式化。

```
table inbox:
    consumer      text not null
    event_id      uuid not null
    status        text not null    // "processing" or "done"
    lease_until   timestamp null
    lease_token   bigint not null  // increases on every takeover
    primary key (consumer, event_id)
```

消费者把该行以 `processing` 状态连同租约一起插入，这次认领在缓慢的工作开始之前，于它自己的短事务中提交。发现租约仍然有效的重复者不得 ack 该消息，因为第一次尝试仍可能失败，而这个 ack 会抹掉唯一的 retry。它应当把消息延迟重新入队，延迟要长于剩余的租约，并在下一次投递时再看一遍。发现租约已过期的一方，可以用更新的租约 token 接管，原持有者在提交其本地变更之前必须检查这个 token；发现 `done` 的一方则 ack。

token 只保护能够检查它的对象。本地事务可以：它锁住 inbox 行，比对 token，若存在更新的 token 就回滚。缓慢的外部调用无法回滚，所以这个效果要使用下一节所述的确定性键，而醒来后重复该调用的过期持有者，会被下游认出。这就是第 5 章 relay 租约中的 fencing 思想，也有同样的局限：只有受保护的资源检查 token，fence 才起作用。

这套方案行得通，但它增加了一个必须有人去推敲的状态机，包括工作者持有租约而死亡时会发生什么。只有当第一种办法不可用时，我才会选用它，并让租约保持简短且可续期。

> **警告**：不要依赖事先的 `SELECT` 或行锁来挡住重复。第二个消费者能否看到第一个消费者尚未提交的工作，取决于数据库的隔离级别，在一个级别下正确的检查，在另一个级别下会失效。把级别提高到 serializable 也无法免除对约束的需要：它只是把悄无声息的重复换成了序列化失败，处理器必须把它视为可 retry 的，并从头再运行一遍。锁可以减少争用；真正的保证，是事务内的唯一约束。

去重窗口与保留期限

永远保留每个 id 的 inbox 会无限增长。总有一天会需要删除旧行，而删除改变了保证，其方式值得一个名字：**去重窗口**。一条 inbox 行已被删除的消息，如果再次出现，就可能被再次处理。

那么窗口必须多长？它必须涵盖重复在原始消息之后合理地可能到达的最长时间。我不知道某个具体系统里这个数字是多少，它取决于下面这份简短的清单。

- **Broker 的保留期限和重新投递的时间范围**。重新投递长达四天的队列，或保留一周的日志，意味着重复可能晚到那么久。
- **死信与 redrive 的做法**。如果团队可能在死信失败一周之后才 redrive 它们，窗口就必须长于这个时间（第 8 章）。
- **重放**。从旧的 offset 重新读取事件日志，会产生与重放范围一样久远的事件的重复。要么窗口足够长，要么针对一个被有意重置的消费者运行重放，这是需要事先设计的流程。
- **Relay 故障转移**。故障转移后的重复发布通常在时间上与原始发布很接近，因此远在任何会选用的窗口之内。

常见的形态是：保留期限明显长于 broker 自身的保留期限，比如作为示意取几周，并用一个定时作业分批删除。按时间对 inbox 表分区可以让删除变得便宜，因为删除整个分区的成本远低于删除数百万行。

请记住，inbox 并不是唯一的防线。领域表上的业务键约束与数据同寿，因此能抵御在 inbox 行消失之后才到达的重复。这是同时使用两层的又一个理由：inbox 以有界的内存为常见情形提供廉价的保护，约束则覆盖长尾。

如果系统能够接受，时间窗口之外还有一种替代办法：从某个单调的量派生键。这条规则比乍看起来要窄。它适用于携带状态的事件，也就是第 9 章与增量区分开的那一种，其中每个事件都携带聚合的完整新状态。对这类事件，消费者保存已应用的最高聚合版本，并拒绝任何不比它新的事件。offset 是单个分区内的位置，版本则是单个聚合的属性，二者不可互换：offset 对位于其他分区上的聚合什么也说明不了。

对于增量事件，“不比它新就拒绝”是错的，因为跳过一个版本意味着丢失一次变更。这样的消费者应用版本 `last + 1`，把更晚的版本缓冲起来直到缺口被填上，并把任何小于或等于 `last` 的版本视为重复；第 9 章描述这种缓冲及其局限。无论哪种方式，内存都是有界的，无需 TTL，并且依赖于第 9 章要检验的顺序保证。它适合带版本计数器的聚合，不适合从多个分区取数的消费者。

外部副作用

inbox 保护的是存在于自己数据库中的效果。许多效果并非如此。邮件在发出的那一刻就离开了系统，对支付服务商的调用无论本地事务之后是否提交，都会给银行卡扣款。没有任何本地事务能回滚这些，所以原子性的技巧在此不适用。

以 Payments 处理 `CapturePayment` 命令为例。它必须调用外部服务商，然后记录结果。

```
function handleCapturePayment(command):
    // unsafe for the crash we care about: the provider call has no
key
    result = provider.capture(command.payload.payment_id)
    transaction:
        recordCapture(command.payload.payment_id, result)
        inboxInsert("payments-capture", command.command_id)
    ack(command)
```

如果在服务商调用与事务之间崩溃，钱已经转走，而 Payments 没有任何记录。重新投递时，处理器会再次调用服务商。买家是否会被扣两次款，此刻完全取决于服务商。

把键传给下游

解决办法是把幂等键传给执行该效果的系统，使它能够认出重复。支付服务商通常支持这一点：随请求发送一个键，带有相同键的第二个请求会返回第一次的结果，而不是再次执行。header 或字段的细节因服务商而异，我会去读服务商的文档，了解它的确切行为，以及它记住键的时长。

最重要的细节是*如何选择键*。它必须**由工作确定性地派生，而不是每次尝试都生成**。在处理器内部创建的随机键，在每次 retry 时看起来都与新请求无异，什么也保护不了。

```
function handleSafely(command):
    try:
        if inboxContains("payments-capture", command.command_id):
            // local memory, kept as long as the inbox row: the long-
term guard
            ack(command)
            return

        key = "capture-" + command.command_id
        // same command, same key, on every attempt and every
redelivery
        result = provider.capture(command.payload.payment_id, key)
        // an "in progress" or conflict answer for a concurrent
request with
        // this key is a retryable failure, not a result to record

        transaction:
            recordCapture(command.payload.payment_id, result)
            inboxInsert("payments-capture", command.command_id)
        // commit point
        ack(command)
    on UniqueViolation as e:
        if e.constraint == "inbox_pkey":
            // a concurrent attempt recorded it first
            ack(command)
        else:
            raise e
```

现在再追踪各种故障。在服务商调用之后、提交之前崩溃，钱已转走而 Payments 不知情。重新投递的命令用同样的键运行同样的代码，服务商返回原来的结果，事务将其记录下来。在服务商调用期间崩溃、响应丢失，情形相同：retry 带着同一个键，服务商要么只执行一次扣款，要么返回先前的结果。两个消费者针对同一条命令竞争，会同时发出相同的键，而将第二个请求以“处理中”拒绝的服务商并没有让这项工作失败：处理器把这一回答视为可 retry 的，并在下一次投递时再看。

服务商的记忆与 inbox 不是同一种保护。服务商记住键的时间有限，我会去读它的文档了解多久。在该期限之后到达的重新投递，会造成第二次扣款。本地 inbox 行至少保留到前面所述的去重窗口那么久，它使处理器根本不会去调用服务商，因此服务商的键保留期限绝不是所依赖的那个窗口。

如果工作是按业务而不是按消息来标识的，这里也要使用由业务派生的键，比如支付 id 加上操作。这样，即便 id 不同的两条命令，到达服务商时也是同一个请求。

下游没有键时

并非每个下游都接受键。普通的 SMTP 没有幂等键，许多邮件和短信服务商也没有可靠地提供。这时就面临两种不完美结果之间的选择，我会针对每个效果明确作出选择。

先发送再记录，崩溃可能导致一封重复的邮件。先记录再发送，崩溃可能导致一封缺失的邮件。这就是该效果的 at-least-once 与 at-most-once 行为。对于“您的包裹已排期”这类通知，偶尔重复的代价很低，而丢失一条消息会变成一张客服工单，所以我会先发送，并接受偶尔的重复。对于带来价值的一次性优惠券码，我会倾向另一边，并增加一种检测和修复缺失发送的办法。

有些效果可以变得更安全，办法是在本地事务中记录*意图*，并由单独的工作者执行该效果，这就是把 outbox 的思想应用到命令上。它并不能消除服务商边界上的问题，因为工作者仍可能在调用与标记之间崩溃。但它确实把问题移到了一处，在那里可以看到它，有意地对它执行 retry，并对账。第 13 章讨论对账：把自己的记录与服务商的记录进行比较并修复差异，这是对那些无法做到幂等的效果的最后一道防线。

Idempotency-Key 与 HTTP API

同样的问题也出现在自己的前门。买家的手机提交“下单”，网络丢掉了响应，应用于是执行 retry。除非客户端说明，否则 Orders 无法分辨 retry 与第二张订单。

广泛使用的惯例是 `Idempotency-Key` 请求头：**客户端**为每个逻辑操作生成一个唯一的值，并在该操作的每次 retry 中发送相同的值。这是前面所给定义的另一半：对于事件消费，键通常是由生产者分配的 `event_id`，而对于面向客户端的请求，它由调用方提供。服务器的任务，是让重复的键表现得如同第一次请求。

服务器需要的不止是键。它要把键连同请求的指纹，以及所产生的响应一起存储。

```
function handleCreateOrder(request):
    key = request.header("Idempotency-Key")
    if key is missing:
        return error(400, "Idempotency-Key required")
    try:
        transaction:
            record = idempotencyFindForUpdate(request.caller, key)
            if record is none:
                token = idempotencyInsert(request.caller, key,
fingerprint(request.body),
                                          "in_progress", lease_until:
dbNow() + LEASE)
            else if record.fingerprint != fingerprint(request.body):
                return error(422, "key reused with a different
request")
            else if record.status == "completed":
                return record.stored_response
            else if record.lease_until > dbNow():
                return error(409, "request in progress")
            else:
                // the holder crashed or stalled: take over with a
newer token
                token = idempotencyTakeOver(request.caller, key,
lease_until: dbNow() + LEASE)
        // commit point: the key is claimed, fenced by an increasing
token
    on UniqueViolation as e:
        if e.constraint == "idempotency_pkey":
            // a concurrent first claim won
            return error(409, "request in progress")
        raise e
    transaction:
        // locks the row; raises LeaseLost
        requireLeaseToken(request.caller, key, token)
        response = createOrderWithOutbox(request.body)
        idempotencyComplete(request.caller, key, response)
    // commit point: on LeaseLost the order and its outbox rows roll
back
    return response
```

指纹检查很重要。客户端对不同的请求体复用同一个键，是它自己的 bug，而悄悄返回第一次的响应会掩盖这个 bug。处理中状态对于并发情形很重要：带有同一个键的两个同时到达的请求，不得都执行。行锁只是给对已有行的接管排序；首次认领由唯一约束裁决，所以落败的插入会被捕获，并以 409 作答。

租约对崩溃情形很重要，而它的 token 对缓慢情形很重要。暂停超过租约期限的持有者可能醒来并试图完成，但 `requireLeaseToken` 会看到更新的 token，于是它的整个事务，包括订单和 outbox 行在内，都会回滚。这里 token 之所以起到 fence 的作用，是因为受保护的资源，也就

是本地数据库，在事务内检查了它，与第 5 章 relay 租约是同一个思想。租约时长只是缩短重叠的时间；真正让过期持有者的提交失败的是 token。过期的调用方会收到一个错误，它的 retry 则会得到已存储的响应。两项检查使用的都是数据库的时钟 `dbNow()`，而不是应用服务器的时钟，理由见第 3 章关于时钟的论述。把键限定在调用方的范围内，可以避免一个客户端与另一个客户端相互冲突。

伪代码展示的是成功路径，失败路径同样需要一条规则。如果请求因确定性的原因失败，例如返回 4xx 的校验错误，服务器就把该响应存为已完成的响应。这样，带同一个键的 retry 会得到同样的 4xx，而不会针对可能已经变化的数据再次运行校验。如果它因瞬时原因失败，例如超时或依赖不可用，则释放这次认领，或任其到期，使客户端的 retry 能够运行这项工作。把瞬时故障缓存为最终答复，会把一次小波动变成该键上的永久错误。

`(caller, key)` 上的唯一约束所起的裁决作用，与 inbox 的主键相同。这是同一个模式：原子的认领，已存储的结果，对重复请求的确定性答复。当 API 产生事件时，订单行、outbox 行和幂等记录的完成状态可以在一个本地事务中全部写入，因此被 retry 的请求既不会创建第二张订单，也不会发出第二个 `OrderPlaced`。已存储的键至少要保留到客户端可能执行 retry 的那么久，这是关于客户端行为的决定，而不是关于 broker 的决定。

还有一层联系值得一提：客户端提供的键可以流入事件。Orders 可以把它放进事件 header，或者由它派生 `OrderPlaced` 的 `event_id`，使下游消费者看到的是从买家点击到账本记录的同一个身份。如果 `event_id` 是派生的，它必须确定性地同时来自调用方和键，例如作为覆盖两者的基于名称的 UUID，这样两个碰巧选了同一个键的调用方就永远不会共享一个身份。第 11 章讨论事件中应当包含什么；键是否属于事件，取决于希望幂等性沿着请求走多远。

权衡与成本

inbox 给每条消息增加一次写入。每条被消费的消息都要多付出一次插入，位于同一个事务内，处在快速路径上。对于只做少量写入的消费者，开销很小。对于吞吐量极高、工作极为廉价的消费者，例如更新读模型缓存的消费者，它可能是主要的成本，那里往往以天然幂等为更好的选择。

inbox 的存储和清理是一项持续的工作。表随流量增长，保留策略是需要自己负责并不时重新审视的决定。窗口过短，会重新放进旧的重复；窗口过长，会耗费存储和索引空间。必须有人盯着清理作业，因为一旦它停了，表会悄悄增长，直到成为性能问题。

快速路径上的唯一约束可能成为争用点。键分布良好时，它很廉价。键聚集，或者许多消费者撞上同一个热点聚合时，它可能排起队来。按消费者和事件 id 为表建键，并让处理器事务保持简短，可使这一点保持在可控范围内。

业务键是一项建模决策，而错误的键很难被发现。太细，逻辑上的重复会通过；太粗，合法的工作会被拒绝。两个方向上的失败都是悄无声息的。在依赖一个键之前，我会用刻意刁钻的案例来测试它，比如同一聚合上的两个合法操作。

幂等性并不覆盖一切。它让重复变得无害；它不能修复第一次就写错的处理器，不能给消息排序，也不能撤销服务商没有键的那种效果。inbox 同样不能抵御*发生了变化*的重复：如果生产者以不同的 `event_id` 重新发布“同一个”事实，且不存在业务键，就没有任何东西能认出它。

何时不用 inbox。如果消费者的每个效果都天然幂等且不会过期，例如用最新的完整状态覆盖一个反规范化字段，那么 inbox 就是有成本而无收益，我会跳过它，并把理由记录在代码中。同样不适合的情形，是消费者无法写入自己的事务性数据库，例如无状态的 forwarder。这样的消费者需要它的下游去重，并把键一路传下去，或者需要重新设计，使它拥有一个存储。

常见错误

- **把检查和执行分成两个步骤**。这很诱人，因为代码读起来很自然：“见过这个吗？没有，那就做。”它在两步之间崩溃时会失效，在并发下也会失效。应把 inbox 写入与效果放在同一个事务里，让唯一约束来裁决。
- **在 ack 之后，或在效果提交之前写入 inbox 行**。先 ack 的话，工作随后失败就会丢失消息。在效果提交之前记录，则会把从未发生的工作标记为已完成。效果、inbox 行和提交是一个整体；ack 跟在提交之后。
- **把 inbox 放在与效果不同的存储里**。缓存或另一个数据库感觉上更便宜，却重新制造了双写。应把它放在与其所保护的状态相同的数据

库中。

- **每次尝试都生成新的键**。在处理器或客户端的 retry 循环内创建的随机值，会让每次 retry 看起来都是新的。应从命令、事件或业务操作派生键，使它在每次尝试中都完全一致。
- **只使用 `event_id`，并以为这涵盖了所有重复**。它捕获不到表达同一意图的两条不同消息。领域中存在这类风险的地方，应加上业务键约束。
- **凭猜测选择去重窗口**。窗口短于 broker 的保留期限，或短于 redrive 的做法，会悄悄放回旧的重复。应由这些数字推出窗口，并把推导过程写下来。
- **把唯一约束违反当作一般性错误**。对每个违反都 ack，会掩盖真正的 bug；对每个违反都 retry，会在一条已处理的消息上无限循环。应单独捕获 inbox 的违反。
- **忘了外部效果**。inbox 无法回滚一封邮件或一次银行卡扣款。应向服务商传递确定性的键，或者在可能的重复与可能的丢失之间有意识地作出选择。
- **直接从处理器发布后续事件**。在提交与发布之间崩溃会丢失该事件，而重新投递走 inbox 的快速路径，永远不会重做它。应在与 inbox 行和状态变更相同的事务中，把后续事件写入 outbox。
- **对租约属于他人的消息发送 ack**。看到有效租约却发送 ack 的重复者，会在第一次尝试随后失败时，移除唯一的 retry。应改为延迟重新入队。

要点

- 重复来自 broker、relay、retry、redrive 与重放以及客户端，因此每个消费者都应当按“任何消息都可能到达两次”来编写。
- 先检查后执行并不安全，因为检查和效果是分开的步骤，崩溃或并发的尝试可以从两者之间溜进来。
- inbox 行和业务变更必须在消费者自己数据库的一个本地事务中提交，而键上的唯一约束才是真正裁决并发重复的东西。宣布自己做了什么的消费者，要把一条 outbox 行加入同一个事务。
- 对事件消费，使用 `event_id` 作为基线幂等键，并在两条不同消息可能表达同一意图的地方，加上业务键。
- 去重窗口必须长于 broker 的保留期限、redrive 的做法以及计划中的任何重放，而领域表上的业务键约束，在 inbox 行被删除之后仍然

起保护作用。

- 对于外部副作用，应发送由工作而非由尝试派生的确定性幂等键；由于服务商会忘记键，要把本地 inbox 行作为长期的防线；下游没有键时，则在可能的重复与可能的丢失之间有意识地作出选择。
- 在 API 边界上，幂等键由调用方提供，服务器存储键、请求指纹和响应，使重复请求返回第一次的结果。

幂等性让重复的消息变得无害，但它假定处理最终会成功。有些消息永远不会成功：它们每次尝试都失败，retry 和重新投递堆积在它们身后。第 8 章讨论这些 poison message，以及它们留下的死信。

第 8 章 — Poison message 与死信

设想新字段 `gift_message` 随 `OrderPlaced` 上线后的第二天早晨，Parcelly 出了这样一件事。Notifications 读取该字段来生成确认邮件，而某位卖家的一笔测试订单带有一条礼品留言，其中有一个字符是邮件模板无法编码的。消费者在这一条消息上抛出异常。消息被送回来。消费者再次抛出异常。

到午饭时分，这条消息已被投递数千次。每次尝试都在同一个位置、以同样的方式、在几毫秒内失败。消费者一直在忙，按所有基础设施指标看也很健康，却已经好几个小时没有对任何有用的消息发送过 ack。在基于队列的 broker 上，其余 worker 仍会处理其他消息，所以损害看起来很小。在基于日志的 topic 上，情况就不同了：消费者不断回退 offset 去重读失败的记录，而不是跨过它，于是一个分区停止了前进，该分区上的每个订单都在它后面等待。

终于有人注意到了，有人提议把这条消息挪到别处，好让工作继续。这只是答案中容易的一半。困难的一半在后面：谁来查看这条消息，如何判断它意味着什么，又如何让它重新回到系统中而不引发第二次事故。本章同时处理这两半。系统应当如何对待一条自己无法处理的消息？团队又如何避免这个答案沦为一座垃圾填埋场？

问题及其难点

poison message 是每次处理都会失败的消息。定义的关键在“每次”二字。因数据库短暂不可达而失败的消息不是 poison；数据库恢复后，它就会成功。poison message 的失败则源于消息本身或读取它的代码：在有人工或一次部署改变某些东西之前，成功是不可能的。用第 3 章的话说，这类故障的语义表明再次尝试无济于事，这正是对它执行 retry 纯属浪费的原因。

常见原因分为几类：

- **格式错误或出乎意料的内容**。必填字段缺失、取值越界，或 payload 无法解析。与旧版消费者不兼容的 schema 变更会产生这种情况（第 12 章讨论如何避免）。
- **消费者中的 bug**。消息本身有效，但代码错误地处理了一个无人测试过的情形，例如在从未预期会出现 null 的地方出现了 null。
- **违反了某项业务假定**。`PaymentCaptured` 事件到达了一个没有对应预授权记录的消费者，原因可能是数据被删除，也可能是事件在途中交错了顺序，而 retry 已经超出了任何合理的乱序窗口。在窗口之内，同样的症状是瞬时的，如第 6 章所述、第 9 章详细讨论，属于 retry 的范畴，而不属于 DLQ。过了窗口，消费者既无法继续，也无法凭空造出缺失的记录。
- **下游永久的拒绝**。外部承运商认定某个地址无效而拒绝。再试一次，得到的还是同样的拒绝。

第 6 章引入了瞬时故障与永久故障的区别，本章从那里接着往下讲。一套运作正确的 retry 策略终究会放弃某条消息。retry 受尝试次数、时间或预算的限制。到了那一刻会发生什么，是一项设计决策；如果没人做这个决策，broker 会代为做出。

无限重新投递循环

如果消费者什么特别的事都不做，broker 的默认行为就会接管。基于队列的 broker 会重新投递未 ack 的消息，要么在 nack 之后立即重新投递，要么在 visibility timeout 到期后重新投递，并且通常会维护一个随每次尝试递增的重新投递计数。

基于日志的 broker 工作方式不同，这个差别很重要。它没有按消息计的重新投递：消费者持有一个 offset，并向前读取。不提交 offset 并不会让 broker 把记录交还；消费者在内存中的位置继续前移，失败的记录被悄悄跳过，直到下一次重启或 rebalance 把消费者倒回到上一次提交的 offset。日志上的循环之所以存在，只是因为消费者自己造出了它：回退到失败的 offset 重新读取，或者在原地 retry 期间停止推进分区。如果消费者两者都不做，而是照样越过失败提交 offset，这条记录就会在没有任何信号的情况下丢失。

在两类 broker 中，结果可能是相同的：消息回来，失败，再回来；或者在 offset 处理不当的日志上，消息直接消失。日志类 broker 的正确工具是显式的回退 offset 或暂停、让失败记录进入以便主分区继续前进的 retry topic，以及存放被放弃记录的死信 topic。

这个循环有三重代价。第一是浪费容量：worker 把时间花在注定无法成功的工作上。第二是噪声：日志里充满相同的堆栈跟踪，真正的问题藏身其中。第三是最要紧的代价，即**队首阻塞**。在日志上，消费者按顺序处理一个分区，因此它不肯跳过的一条记录会阻塞其后的每一条记录。在普通队列上，单条 poison message 本身不会阻塞其他消息，但若干条就能占满所有 worker。如果队列为了保证顺序而对消息分组（FIFO 队列、消息组、会话），poison message 会阻塞它所在的组，影响范围与日志分区相同，只是更窄。

两种糟糕的出路

处于压力之下的团队通常会伸手去拿两种出路，而二者都是错的，原因相同：它们掩盖了故障，而不是处理故障。

第一种是**丢弃消息**：捕获异常，记一条日志，发送 ack，然后继续。流水线重新流动了，但业务事件没了。对于 `OrderPlaced`，这意味着一个订单存在于 Orders 中，却不存在于任何其他服务中。对于 `PaymentCaptured`，这意味着账本与现实对不上。只有当消息没有任何价值时，丢弃才是正确的选择，而消费者在失败的那一刻无从知晓这一点。

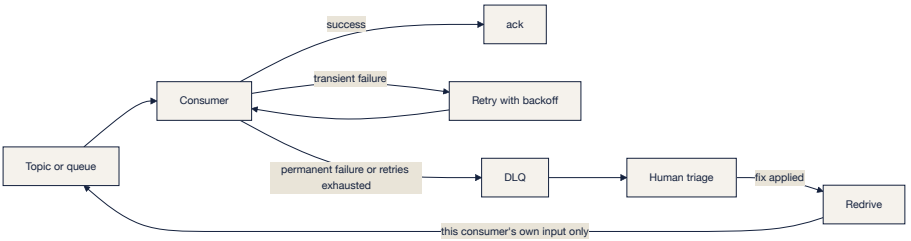
第二种是**永远 retry**，并相信总会有人修复原因。这就是上面的无限循环，它把一条坏消息变成了排在它后面的一切内容的故障。

第三种选择介于二者之间：把消息放到一边，附上足够的上下文，让人以后能看懂它，同时让 stream 的其余部分继续运行。这种选择就是死信队列。

死信队列模式

死信队列（DLQ） 是一个独立的目的地，消费者或 broker 把自己已放弃的消息放在这里。在基于日志的 broker 上，同样的思路通常表现为**死信 topic**：一个接收失败记录的普通 topic。当某个陈述对两类 broker 都成立时，本章统称为 DLQ。

这个模式的核心很简单。消息用尽 retry 之后，或者以消费者归类为永久的方式失败之后，消费者把它转入 DLQ 并对原消息发送 ack。stream 继续运行。失败的消息没有丢失；它被停放在可以检查的地方。



请看图中最后两个箭头。DLQ 不是出口。它是一间等候室，门通向出故障的那个消费者自己的输入，而本章的全部难点就在这两扇门之间的距离上。这扇门并不通向共享的 topic：其他消费者已经处理过该事件，不应当再看到它一次。

转移发生在哪里

实现这一转移有两种方式，选择会影响能保存什么。

由 broker 管理的死信处理。许多基于队列的 broker 可以配置为：消息被投递超过一定次数后，自动转移到指定的队列。设置成本低，即使消费者在来得及做任何事之前就崩溃，它也有效。它能提供投递次数，但通常提供不了消费者失败的原因，因为 broker 并不知道。

由消费者管理的死信处理。消费者捕获故障，构造一条包含原因的记录，把它发布到 DLQ，然后才对原消息发送 ack。这样可以对故障分类，并在第一次尝试时就把永久错误转入死信，而不是等到第五次。代价是每个消费者里都要多写代码，而且必须写得谨慎：先发布到 DLQ，再发送 ack，这样两步之间发生崩溃时，产生的是 DLQ 中的重复消息，而绝不会是丢失。

在 broker 提供前一种机制的地方，我会两种都用。由 broker 管理的规则只存在于部分基于队列的 broker 上，并非每种队列类型都限制投递次数。有它的地方，它是应对消费者崩溃或以代码捕获不到的方式挂起的兜底机制；它与消费者管理的路径合在一起，意味着即使自身的处理有 bug，poison message 也不可能永远循环下去。

基于日志的 broker 通常既没有按消息计的投递计数器，也没有 broker 侧的 DLQ，这就留下了缺口。一条会让进程崩溃的消息，例如在处理器运行之前发生的反序列化器故障、内存溢出错误或原生崩溃，永远到不了 `try` 块。消费者从已提交的 offset 重启，读到同一条记录，再次崩溃。第 6 章指出过，崩溃之后尝试次数会从头计数。对 poison record 来说，这就是问题的全部。

补救办法是在处理之前计数，让计数在崩溃后依然存在：

```
function pollLoopStep(record):
    key = record.partition + ":" + record.offset
    // the counter is written before any work, so a crash still counts
    attempts = incrementDurableCounter(key)
    if attempts > MAX_PROCESS_STARTS:
        // already tried too many times: do not run the code that
crashes
        deadLetterRaw(record, "exceeded process starts")
        clearDurableCounter(key)
        return
    // deserialization is inside the guarded path, not before it
    handleMessage(deserializeGuarded(record))
    clearDurableCounter(key)
```

计数器以分区和 offset 为键（如果能在不反序列化的情况下读到 `event_id`，也可以用它）。代价是热路径上每条记录都要做一次持久化写入，而且上限必须设得足够高，使一次普通的部署重启不会把健康的记录推过上限。另一条路是接受这样的后果：反复崩溃的 poison record 会让所在分区停滞，直到有人发现，这也正是下文所述的 lag 告警很重要的原因。

```
function handleMessage(message):
    try:
        processBusinessLogic(message)
        ack(message)
    on PermanentFailure as failure:
        // classified permanent: retrying cannot help, so skip the
retry budget
        deadLetter(message, failure)
    on error as failure:
        // Chapter 6's classifier: anything unclassified is "unknown"
        // and gets the stricter attempt limit, never an escaped
exception
        limit = attemptLimitFor(classifyFailure(failure))
        if message.attempt_count >= limit:
            deadLetter(message, failure)
        else:
            // nack with backoff on a queue; retry topic on a log
(Chapter 6)
            retryLater(message)
```

`deadLetter` 这一步是捕获分诊所需上下文的地方，其最后两行的顺序是关键。

```
function deadLetter(message, failure):
    record = {
        original_message: message.body,
        original_headers: message.headers,
        original_key:     message.key,
        original_time:    message.timestamp,
        source:           message.source_name,
        source_position:  message.position,
        consumer_input:   CONSUMER_INPUT,
        redrive_count:    message.redrive_count,
        error_class:      failure.class,
        error_message:    truncate(failure.message, 1000),
        attempt_count:    message.attempt_count,
        consumer:         CONSUMER_NAME,
        consumer_version: BUILD_VERSION,
        first_failed_at:  message.first_failed_at,
        failed_at:        now(),
        correlation_id:   message.correlation_id,
        stack_trace_ref:  failure.trace_pointer
    }
    // publish first, ack second: a crash between them yields a
duplicate
    // dead letter, which is recoverable; the reverse order loses the
message
    publish(DLQ_NAME, record)
    ack(message)
```

如果 `publish` 本身失败，`deadLetter` 会抛出异常，消息不会得到 ack。在队列类 broker 上这就够了：消息会通过正常的重新投递回来。在日志类 broker 上则不够，因为未提交的 offset 不会让记录回来；消费者必须回退到失败记录的位置，或暂停该分区，并在继续读取之前对死信的发布执行 retry。两种情况的目标是同一种故障模式：消息一直留在系统中，直到它有了一个安全的新归宿。

死信应保存哪些内容

没有上下文的死信是一道谜题。六周之后在 DLQ 里发现一条消息的团队，需要回答四个问题：它是什么，它来自哪里，它为何失败，现在处理它是否仍然安全？这条记录应当无需打开源码仓库就能回答这些问题。

字段	存在的原因
原始 body 和 headers，逐字节保存	redrive 必须原样重现消息，包括 `event_id`
来源名称与位置（topic、分区与 offset，或队列名）	在历史中定位该消息，支持顺序方面的判断
原始键（记录键或路由键）与原始时间戳	redrive 必须使用相同的键发布，否则消息会落到另一个分区，丢失其按键的顺序

字段	存在的原因
消费者输入（出故障的消费者自己的队列或 redrive topic）	redrive 唯一可以发布到的地方
`redrive_count`	与 `attempt_count` 分开：这条消息已经回来过多少次
`error_class` 和 `error_message`	按原因对故障分组；过长的消息要截断
堆栈跟踪，或指向日志行的指针	让工程师无需复现就能看到失败发生在哪里
`attempt_count`	区分“失败一次即永久”与“多次 retry 后失败”
`first_failed_at` 和 `failed_at`	显示消息已经等待了多久，以及问题从何时开始
消费者名称和版本	把故障与某次部署联系起来；往往是找到原因的最快途径
`correlation_id`	把消息与其所在业务流程的其余部分连接起来

有两点需要注意。第一，保持原始 payload 不被修改。如果用自己的信封把它包起来，redrive 工具必须原样解开，否则 `event_id` 和 `causation_id` 会漂移，第 7 章所述的幂等性保证也就失效了。第二，要记住 DLQ 是生产数据的一份副本，而且往往保留很长时间。它继承了原数据的访问控制和保留义务，含个人数据的 payload 在 DLQ 中需要与其他任何地方同样的小心。

Parking lot 与 DLQ

DLQ 这个词被用来指两种不同的东西，把它们分开会让运维更容易理清。

parking lot 是存放需要人工决策的死信的地方。DLQ 是系统自动放入消息的地方；parking lot 则是由人在查看之后、判定这条消息无法靠 redrive 修复时，有意放入消息的地方。它是一个独立的地方，有独立的用途。

来看看两者各自装着什么。DLQ 装的是已经失败、尚未被检查的消息。它应当被消化：预期状态是空的或接近空，其中任何陈旧的内容都是一种症状。parking lot 装的是有人检查过并搁置的消息：一个在事件到达之前就已取消的订单，一个需要人工修正的 payload，一条必须与另一个团队手工对账的记录。这些消息不是在等待某个修复被部署，而是在等待一个决定。

把二者混在一起，会产生可预见的故障。如果已经决定并停放的消息留在 DLQ 里，DLQ 的深度就永远降不到零，为了止住噪声而调高了告警阈值，下一条真正的 poison message 到达时便无人察觉。把它们分开，才能让 DLQ 的告警保持灵敏。

parking lot 可以朴素到只是一张带有状态、负责人、原因和复查日期的数据库表。它完全不必是 broker 的目的地，因为不会有任何东西去自动消费它。复查日期最为重要：一条没有下次查看日期的停放消息，已经被遗忘了。

责任归属与告警

没有负责人的 DLQ 是一个悄悄被填满的排水口。在动用任何工具之前，第一个决定就是谁来负责每一个 DLQ。

我的默认做法是：拥有该消费者的团队拥有该 DLQ。他们写了出错的代码，知道这条消息本应意味着什么，也只有他们能修复并部署。每个消费者各设一个 DLQ，或者每个 consumer group 与来源的组合各设一个，这样责任归属一目了然。让整个公司共用一个 DLQ，则会使归属含糊，而含糊意味着没人去看。如果由平台团队提供工具，平台团队负责机制，使用方团队负责内容。

对什么告警

要对**到达**告警，而不只是对深度告警。在一个通常为空的 DLQ 中出现单独一条消息，就值得发出通知，因为第一条死信是发现 bug 成本最低的时刻。单靠深度阈值，要等到队列已经很深了才能说明什么。

三种信号合在一起，覆盖了大多数情形：

- **任何新增到达**，发生在本应为空的 DLQ 中。把它发送到工作时间内有人阅读的渠道。
- **到达速率**。突然的爆发通常意味着一次有问题的部署或不兼容的 schema 变更，值得呼叫某个人。
- **最老消息的年龄**。这能捕捉缓慢的故障：一个 DLQ 每天只收到一条消息，无人处理，直到最老的一条已有一个月，业务上下文也已不复存在。

我不知道适合某个具体系统的呼叫阈值是多少，不了解相关消息的业务价值，没有人能给出答案。挂在 `PaymentCaptured` 消费者上的 DLQ，应当比挂在 Notifications 上的那个设置更低的阈值，因为丢一封邮件只是令人烦恼，而丢一笔扣款则是账本差异。阈值应当反映这种差异，并且这种差异应当写在告警旁边。

> **注意**：对 consumer lag 也要告警。卡在日志分区中某条 poison record 上的消费者，不一定会产生死信，因为它从不放弃。lag 上升而吞吐为零，就是本章开头所述的循环正在运行的信号。

Redrive：把消息送回去

redrive 指把死信交还给当初处理它们失败的那个消费者，以便重新处理。目标是该消费者自己的输入：它的 retry topic 或 redrive topic，或它自己的队列。绝不是消息最初来自的那个共享 topic，因为该 topic 的其他每个消费者都已经处理过这个事件，在那里重新发布会把它第二次投递给所有这些消费者。原始键随消息一起传递，所以它仍留在同一个分区，保持其按键的位置。尝试计数器重新从零开始，而单独的 `redrive_count` 加一，于是一条不断回来的消息会被看到，而不是每次都像新的一样。redrive 是让 DLQ 有价值的操作，也是最容易引发第二次事故的操作。第 3 章描述过那位按下按钮、一次 redrive 40,000 条消息的工程师。补救办法是把 redrive 当作一套经过设计的流程，而不是一个方便按钮。

先修复原因，再 redrive

因消费者 bug 而失败的消息，在 bug 被修复并部署之前还会再次失败。在修复之前 redrive，只是让消息兜圈子，并用同样的条目重新填满 DLQ，只不过尝试计数更高、历史更令人困惑。正确的工作顺序是：弄清原因，部署修复，redrive 一小批样本，观察它成功，然后 redrive 其余部分。

Redrive 必须可安全重复

被 redrive 的消息按定义就是重复消息：系统以前见过这个 `event_id`，消费者也可能见过。如果原来的处理在失败之前部分成功，某些效果可能已经存在。这件事之所以承受得住，靠的正是第 7 章。带有 inbox 的幂等消费者会像对待其他任何重新投递一样对待被 redrive 的消息，并且只应用一次其效果。

有两点值得说明。redrive 必须保留原始的 `event_id`，因为它就是幂等键。为每条消息生成新标识符的 redrive 工具，会使 inbox 失效，效果可能被应用两次。其次，不幂等的消费者根本不应当成为批量 redrive 的目标；先修好这一点，或者接受每条被 redrive 的消息都是一次人工风险评估。

分批受控地 redrive

一个安全的 redrive 工具有几个特性，没有一个是稀奇的：

1. **筛选条件**。按 error class、时间窗口或 `correlation_id` 来 redrive，而不只是“全部”。
2. **速率限制**。消息以消费者及其依赖项能够承受的节奏回到消费者的输入。大量涌入的被 redrive 消息就是自己给自己造成的 retry 风暴，“过载下的 retry：反压”（第 6 章）中的在途上限和负载削减，同样适用于被 redrive 的流量，与实时流量无异。
3. **暂停开关**。如果故障再次出现，运维人员可以中途停止 redrive。
4. **dry run**。工具在移动任何东西之前，先报告它将要移动什么。
5. **审计记录**。谁在何时因何原因 redrive 了什么，保存在某个持久的地方。

```
function redriveBatch(dlqName, filter, batchSize, maxPerSecond):
    // an operator procedure, not a consumer: run it with a human
watching
    batch = readFromDlq(dlqName, filter, batchSize)
    for each record in batch:
        if redriveIsPaused():
            return "paused"
        // the failing consumer's own input, never the shared source
topic;
        // the original key keeps the message on its partition, and
the
        // original headers and event_id keep the consumer's inbox
effective
        publish(record.consumer_input, record.original_message,
                key: record.original_key,
                headers: record.original_headers
                         + { attempt: 0, redrive_count:
record.redrive_count + 1 })
        // mark this record as redriven, one by one; a crash leaves a
few
        // already-published records unmarked, which is a duplicate,
not a loss
        markRedriven(record, operator(), now())
        sleep(1 second / maxPerSecond)
    return "batch complete"
```

注意循环内部的顺序：记录是在发布成功之后才标记为已 redrive，而不是之前。如果工具在中途崩溃，就会有少数消息已被 redrive，却仍在 DLQ 中未被标记，这会导致重复，而不会导致丢失。带有 inbox 的消费者能吸收这些重复；没有 inbox 的消费者则会把一次工具崩溃变成数据问题。记录是被标记，而不是被删除，所以审计轨迹和 `redrive_count` 得以保留；清理则遵循下文“避免沦为垃圾填埋场”中的保留规则。

Redrive 与消息顺序

除非专门设计，redrive 会破坏顺序。一条被转入死信的消息被跳过了；同一聚合的后续消息在此期间已被处理。当被 redrive 的消息到达时，它排在这些消息之后。这是第 6 章的 Inventory 例子，只是间隔更长：如果 `OrderPlaced` 被转入死信而不是被 retry，`OrderCancelled` 仍然找不到可释放的预留，而数小时后被 redrive 的下单消息仍会创建一个无人释放的预留。保留键并不能修复这一点：键把消息放到了正确的分区，但分区已经向前走了。

这是否要紧，取决于消费者。**满足交换律**的消费者，即无论事件以什么顺序到达，最终状态都相同，不受影响。依赖状态转换的消费者则不然。第 9 章讲述如何用版本号或序列检查来检测和处理乱序到达，基于状态的消

费者在有人向它 redrive 之前，应当先用上这些手段。如果没有，redrive 流程就必须包含一项检查，确认自消息被转入死信以来，外部世界没有已经变化。

死信对有序 stream 的影响

有序 stream 迫使人做出本章最难的决定，因为保护顺序的那个选择，恰恰是会让业务停摆的选择。

设想一个以 `order_id` 为键的基于日志的 topic，Ledger 从中记账。消费者按顺序处理每个分区。针对 `order-81724` 的第 41 条消息永久失败。消费者有两个选项：

- **停下来等待**。把分区停在第 41 条消息，直到有人修复原因。顺序被完整保留。哈希到同一分区的其他每个订单也随之停止，尽管它们的消息本身没有问题。影响范围是整个分区。
- **转入死信并继续**。停放第 41 条消息，处理第 42 条及其后的消息。分区继续流动。但第 42 条可能依赖第 41 条，而消费者随后是在一个不包含第 41 条的状态上处理了它。

两种选择都不是免费的。停下来是用可用性换正确性；继续则是用正确性换可用性。没有哪种设置能两者兼得，与其继承框架恰好给出的默认值，不如针对每个消费者有意识地做出选择。

我的经验法则分三部分：

1. **如果消费者对顺序不敏感，或满足交换律**，就转入死信并继续。Notifications 是典型情形：一个订单上发送失败的邮件，与下一个订单无关。
2. **如果同一键的后续消息依赖失败的那一条**，就不要对该键继续。把失败的消息转入死信，然后也把同一 `partition_key` 的后续消息转移到暂存区，直到第一条得到解决。其他键继续。这样把影响范围限制在一个聚合，而不是一个分区，代价是多出一套机制：按键跟踪“被阻塞”的键。
3. **如果消费者是财务账本，就停下**。乱序处理的代价，是一条错得难以察觉、修复成本高昂的记录，而被暂停的分区则声势浩大、一目了然。正确优先于速度，是为 Ledger 立下的规则，在这里它以最朴素的形式适用。

第二条规则中的按键变体值得画个草图，因为它最有用，也最少见。草图假定有一张小的 `key_block` 表，每个被阻塞的键一行（`partition_key`、`blocked_event_id`、`state`），并假定同一时刻只有一个消费者处理某个键的实时流量，分区或消息组正好提供这一点。

```
function handleOrderedMessage(message):
    key = message.partition_key
    parked = false
    transaction:
        // select ... for update; none if the key is open
        block = lockBlockRow(key)
        if block exists and message.event_id !=
block.blocked_event_id:
            // park under the same lock that resolveBlockedKey takes,
so a
            // message cannot be parked after the block was cleared

            // held messages are unique on event_id; a repeat is
ignored
            appendToHeldMessages(key, message)
            parked = true
    if parked:
        ack(message)
        return
    try:
        // the inbox absorbs a repeat
        processBusinessLogic(message)
        // this was the redriven message
        if block exists:
            // durable: a crash from here on is recoverable
            markResolving(key)
        ack(message)
        if block exists:
            resolveBlockedKey(key)
    on PermanentFailure as failure:
        transaction:
            // state = "blocked"
            markBlocked(key, message.event_id)
            deadLetterRecord(message, failure)
        // blocked key and dead letter commit together
        ack(message)
```

解除阻塞是一道单独的流程，周期性的清扫使它可以重新启动。

```
function resolveBlockedKey(key):
    loop:
        transaction:
            block = lockBlockRow(key)
            if block is none or block.state != "resolving":
                // another resolver cleared or re-blocked it
                return
            next = oldestHeldMessage(key)
            if next is none:
                // emptiness check and clear under the lock
                clearBlocked(key)
                return
        try:
            // the inbox absorbs a repeat after a crash
            processBusinessLogic(next)
            transaction:
                removeFromHeld(key, next)
        on PermanentFailure as failure:
            transaction:
                block = lockBlockRow(key)
                if block is none or block.state != "resolving":
                    // the key moved on while next was in flight
                    return
                // state = "blocked", anchored on next
                markBlocked(key, next.event_id)
                deadLetterRecord(next, failure)
                removeFromHeld(key, next)
            return

function sweepResolvingKeys():
    // runs at worker start and on a timer
    for each key in keysInState("resolving"):
        resolveBlockedKey(key)
```

这份草图中有三个细节决定它能否奏效。第一个是 redrive 本身。被 redrive 的消息正是该键所阻塞在的那一条，所以它必须绕过阻塞：检查时比较它的 `event_id` 与被阻塞的那个。没有这一例外，redrive 的消息就会被停放在本应排在它之后的那些消息后面，键永远解除不了阻塞。

第二个是与实时流量的竞争。`resolveBlockedKey` 在消化暂扣的消息期间，同一键的新消息仍在不断到达。在暂扣集合清空之前，它们仍必须被暂扣，而不是被处理。只有把“停放”的决定和“清除”的决定串行化，这一点才成立：两者都获取该键的行锁，而且停放操作与检查写在同一个事务中。如果停放是单独的一步，一条实时消息可能先看到键处于被阻塞状态，随后发现暂扣集合已被清空、阻塞已被清除，于是被停放在一个已不存在的阻塞之后。没有人会去消化它，而后续的消息会排在它前面被处理。

第三个是解除过程中的崩溃。一旦被 redrive 的消息处理完毕，会在发送 ack 之前写入 `resolving` 状态，这样要做的工作就记录在数据库中，而不是只存在于一条再也不会回来的消息里。清扫会接手任何停留在该状态的键并继续；崩溃前不久处理过的消息由 inbox 吸收。如果某条暂扣的消息在解除过程中永久失败，该键会再次被阻塞，以这条消息为锚点，并带有它自己的死信，暂扣集合中的其余部分则等待它被 redrive。按键阻塞是一个循环，而不是一次性事件。

这份草图并不完整。清扫可能与一次正在进行的解除重叠，于是两个解除者可能同时处理同一个键。在锁内做的状态检查，使其中任何一个都无法越过一次重新阻塞：一旦某个解除者把阻塞锚定在一条失败的消息上，另一个就会发现该键已不再是 `resolving`，从而停下。剩下的是对同一条暂扣消息的重复工作，由 inbox 吸收；给 `resolving` 状态加一个租约，可以省下这些工作，但不关乎正确性。草图也没有给暂扣集合设定上限，还略去了运维人员如何丢弃一条已不再有意义的暂扣消息。应将它视为这套机制的大致形状，而不是一个完成的机制。

这是真正的机制，带有它自己真实的故障模式，例如暂扣消息存储无限增长，所以我只会在按键顺序重要、且死信率低到暂扣集合能保持较小的 stream 上才构建它。它还是一种不知道失败的消息解决之后，暂扣的消息是否仍有意义的机制。当状态转换本身需要被验证，而不仅仅是被排序时，第 9 章中的版本缓冲区是更强的工具：它按版本号暂扣一个事件，直到其前驱真正被应用，并且无需死信记录作为锚点即可工作。

避免沦为垃圾填埋场

当消息进入的速度超过了任何人判断其含义的速度，DLQ 就变成了垃圾填埋场。症状众所周知：成千上万条记录，各种各样的错误，没人确定哪些要紧，还有一条不成文的规矩：别去碰它。到了这一步，DLQ 就不再是一种安全机制；它成了一个用礼貌的名字来存放丢失数据的地方。

垃圾填埋场的形成有原因，每一种原因都有对策。

- **没有负责人**。对策是上文的责任归属规则，并且写下来。
- **没有分诊节奏**。死信应当按计划复查，而不只是在告警触发时才看。对于低流量的 DLQ，每周复查是一个合理的起点。
- **保留期限比有用期更长，或者在分诊之前就过期**。针对上个月已过期的库存预留的死信，无法被有意义地 redrive。应根据消息对业务保持有效的时长设定保留期限，并事先决定到期时怎么办：归档、带审

计条目地丢弃，或升级。相反的错误同样常见：保留期限短于分诊节奏，会在任何人看到之前就删掉消息。如果 DLQ 是一个 broker 队列，其保留期限短于复查间隔，那么 broker 会先把证据清除掉。应把 DLQ 的保留期限设得高于两次复查之间最长的间隔，并查阅所用平台的文档，弄清保留时钟从何时开始计时。如果它从原始入队时间算起，而不是从转入死信的时刻算起，那么一条在失败前等了很久的消息，到达时就已临近过期。这是记录要保存原始时间戳的又一个理由。

- **一切都被转入死信**。如果例行的业务结果，例如属于流程正常组成部分的校验拒绝，也被送进 DLQ，真正的故障就会被淹没。一个被拒绝的订单是有其自身处理方式的业务事件，而不是死信。
- **瞬时故障过早被转入死信**。如果 retry 策略在短暂的中断期间几秒钟后就放弃，那么中断期间在途的每条消息都会落入 DLQ。对策见第 6 章：retry 策略的尝试次数与截止时间，应按预期要扛过的中断来确定。

我还会给每条死信一个处置结果。分诊以一份简短清单中的某一项结果结束：修复后 redrive，修正后 redrive，停放等待人工决定，或带着记录在案的原因被丢弃。“还放在那儿”不是处置结果。

权衡与成本

DLQ 把一个故障，从一个声势浩大的地方，即被阻塞的消费者，转移到了一个悄无声息的地方。这既是它的好处，也是它的隐患。stream 在流动，而失败的消息可能被遗忘。本章中除 DLQ 本身之外的一切，包括告警、负责人、复查节奏和保留期限，都是为了抵消这种悄无声息而存在的。

需要权衡的具体成本：

- **运维负担**。总得有人去分诊、决策和 redrive。一个没人有时间处理的 DLQ，比没有更糟，因为它以一套机制的外表掩盖了问题。
- **数据治理**。死信保存着生产 payload，有时含个人数据，位于通常的生命周期之外。它们需要访问控制、保留规则和删除路径。
- **顺序风险**。把有序 stream 中的消息转入死信，就是跳过它。除非构建按键阻塞或停止分区，否则后续事件会在缺少它的情况下被应用。按键阻塞本身又增加了一个暂扣消息存储、一条 redrive 旁路、一个

必须与实时流量安全竞争的解除循环以及一个让被中断的解除重新启动的清扫。

- **存储的身份信息**。安全的 redrive 需要每条记录上都有原始键、分区和消费者输入以及尝试计数旁边的 `redrive_count`。这意味着更多要存储、更多要保持一致，而没有它们，redrive 就只能靠猜。
- **工具投入**。一个带有筛选、速率限制、逐条处置结果和审计轨迹的安全 redrive 工具，是一个小项目。购买或构建它，是拥有 DLQ 的成本的一部分。

何时不用：如果一条消息可以重新生成，或者丢失它无害，DLQ 带来的流程多于价值。缓存预热事件或纯粹外观上的更新，可以丢弃，并用指标统计丢弃的数量。而在正确的故障响应是停下来的地方，如账本的例子，让 stream 继续的 DLQ 反而适得其反。这种情况下，应停止消费者，大声告警，并修复原因。

这个模式的能力也有限。DLQ 保存的是明显失败的消息。对于被错误处理却没有抛出错误的消息，它无能为力，那是第 13 章及其对账工作的领地。

> **警告**：绝不要在死信被持久存储之前对原消息发送 ack。如果发送 ack 之后 DLQ 发布失败，这条消息就哪里都不存在了，也不会有任何告警提示。

常见错误

- **转入死信时不带上下文**。这很诱人，因为直接移动原始 body 的代码最短。解决办法是上文所述的元数据记录，包括精确的原始 headers，使 redrive 和诊断都成为可能。
- **对永久故障执行 retry**。解析错误不会随时间改善，而一刀切的 retry 策略却把它当作超时对待。应对故障分类，把永久故障直接送入 DLQ，而不是耗费 retry 预算。
- **在修复原因之前 redrive**。“让消息动起来”的压力是真实的，结果却是同样的消息又回到了 DLQ。先部署修复，redrive 一批样本，然后再处理其余的。
- **一次性 redrive 全部**。大量旧消息与实时流量争抢资源，可能引发新的事故。应限速、分批，并保留暂停开关。

- **只对深度告警**。它感觉像是显而易见的指标，但触发得太晚。应对首次到达、到达速率和最老消息的年龄告警。
- **所有东西共用一个 DLQ**。创建成本低，却无法有人负责。给每个消费者各设一个，并指名道姓地指定负责人。
- **让停放的消息和未检查的消息混在一起**。DLQ 永远消化不完，告警被调低，真正的 poison message 无人察觉。应另设一个 parking lot，配有负责人和复查日期。
- **redrive 时忽略顺序**。被 redrive 的消息会在同一聚合的后续事件之后到达。使用第 9 章的版本检查，或确认状态没有已经变化。
- **redrive 到共享 topic**。这看起来像是“把消息放回它来的地方”，其实是把事件第二次投递给该 topic 的所有其他消费者。应使用原始键，redrive 到出故障的消费者自己的输入。
- **在反复崩溃的记录上永远等待**。在日志类 broker 上，永不放弃的消费者既不产生死信，也不产生告警，只有 lag 会变化。应按分区和 offset 持久地统计尝试次数，使重启不会重置计数。

要点

- poison message 每次处理都会失败，所以永远 retry 它，会把一条坏消息变成排在它后面的一切的故障。
- 丢弃失败的消息和永远 retry 它，都是掩盖故障的做法；死信队列则带着上下文把消息放到一边，使 stream 的其余部分得以继续。
- 死信应当携带未经改动的原始 body 和 headers、来源位置、错误、尝试次数以及消费者版本，因为数周之后读它的人，手里只有这一条记录。
- 先发布到 DLQ，再对原消息发送 ack，这样崩溃产生的是可恢复的重复消息，而绝不会丢失消息。
- 每个 DLQ 都需要一个指名的负责团队、针对首次到达、速率和年龄的告警以及定期的分诊节奏，否则它会沦为垃圾填埋场。
- 只有在原因已修复、原始 `event_id` 和键得以保留、消息送到出故障的消费者自己的输入而绝非共享 topic、消费者幂等，并且消息以限速的批次、带着暂停开关返回时，redrive 才是安全的。
- 在有序 stream 上，把一条消息转入死信，会迫使人在停止分区与越过缺口继续之间选择；应针对每个消费者有意识地选择，对账本选择停下，仅在机制值得其成本的地方才按键阻塞。

死信和 redrive 都依赖一个本章一再推迟的问题：消息以正确的顺序到达意味着什么？当它们没有这样到达时，又能构建什么？第 9 章将直接处理这个问题。

第 9 章 — 消息顺序

设想 Parcelly 的一个周四傍晚。一位买家下了订单，几秒钟之内又改变主意，把它取消了。Orders 依次发出 `OrderPlaced`、`OrderConfirmed`、`OrderCancelled`，全都针对 `order-81724`，顺序完全正确。relay 也按这个顺序把它们发布出去。

为 Shipping 供数的 topic 是用默认设置创建的：消息不考虑内容，被分散到六个分区。`OrderConfirmed` 落在分区 2，`OrderCancelled` 落在分区 5。Shipping 为每个分区运行一个消费者。分区 5 上的消费者空闲而且很快，于是先读到了取消消息，发现对应的发货单还不存在。它耸耸肩，对这条消息发送了 ack。片刻之后，分区 2 上那个正忙于一个慢批次的消费者读到了确认消息，为一个已经不存在的订单安排了一次发货。

没有任何东西崩溃，没有消息丢失，也没有消息重复。每个组件都做了配置要它做的事。周一，一位承运商将会到来，取走一个没人下单的包裹。这样的系统需要多少顺序？顺序在哪里丢失？在无法拥有顺序时，又能构建什么？

问题及其难点

顺序并不是系统要么具备、要么缺失的单一属性。它是一条链，每个环节都可以独立地破坏它。relay 发布各行的顺序，可能与数据库提交的顺序不同（第 4 章和第 5 章讨论这一环节）。broker 可能把相关的消息分散到不同的分区或队列。消费者可能并行处理。一次 retry 可能让消息排到其后继者的后面（第 6 章），而一条死信则可能把它完全跳过（第 8 章）。

环节	顺序如何丢失	去哪里看
生产者与 relay	并行的 relay、并发提交、与提交顺序不一致的 id	第 4 章和第 5 章
Broker	相关消息位于不同的分区或队列	本章
消费者	多个 worker 处理来自同一分区或队列的消息	本章
故障处理	retry topic、nack、死信、redrive	第 6 章和第 8 章

环节	顺序如何丢失	去哪里看
拓扑变更	分区数量变化、consumer rebalance	本章

显而易见的反应是要求全局顺序：系统中的每个事件都排在同一个序列中，被每个消费者看到。这听起来像是安全的默认选择。它代价高昂，而且对大多数系统而言并无必要。

全局顺序为什么代价高昂

单一的全序意味着单一的序列点。在基于日志的 broker 上，那就是一个分区，同一个 consumer group 中同一时刻至多有一个消费者读取它。吞吐量的上限就成了一个写入方加一个读取方的速度。增加容量毫无作用，因为容量来自并行运行，而并行恰恰是全序所禁止的。

跨越多个服务的系统会使情况更糟。Orders、Payments 和 Inventory 各有自己的数据库和自己的 outbox。要在它们之间建立全局顺序，就需要一个为每个事件盖戳的协调者，而这个协调者会位于每个服务的写入路径上。这等于买回了这套架构原本要消除的耦合。

全局顺序为什么多半没有必要

不妨问问业务实际需要什么。Parcelly 并不在乎 `order-81724` 的 `PaymentCaptured` 是否先于 `order-90311` 的 `OrderPlaced`。这两个订单互不影响。重要的是，对同一个订单，`OrderPlaced` 要先于 `OrderConfirmed` 被看到，`OrderConfirmed` 要先于 `OrderCancelled` 被看到。

这就是**按聚合保证顺序**。聚合是一致性边界，其标识符充当分区键。如果关于同一个聚合的每个事件都按其产生的顺序被看到，而关于不同聚合的事件可以自由交错，大多数状态机就能正确工作。这是负担得起的顺序，依我看，也是应当围绕其进行设计的顺序。

即使是按聚合的顺序，也有一道它无法跨越的边界。同一个订单的事件，如果由不同的服务产生、在不同的 topic 上传送，它们彼此之间没有顺序。`PaymentAuthorized` 和 `StockReserved` 可能以任意次序到达 Orders，这正是本书通篇使用的订单流程要等二者都到达后才请求扣款的原因。不要把共享的分区键当作跨 topic 的承诺。

按聚合保证顺序

这套机制说起来简单，维持起来却很苛刻：选定一个键，把带有同一个键的一切送上同一条路径，并在该路径上一次处理一条消息。

分区键

在基于日志的 broker 上，生产者对 `partition_key` 做哈希来选择分区。带有相同键的每条消息都落在同一个分区上，而分区是一个有序序列。本书中的信封正因此携带 `partition_key`，outbox 也存储它，使 relay 无需询问生产服务就能设置它（第 4 章）。

这个保证有一个条件：**相同的键、相同的 partitioner、相同的分区数量**。partitioner 是哈希函数，加上键在哈希之前被序列化为字节的方式。如果两个生产者使用的不同，同一个 `order_id` 就会去往两个分区，按键的顺序随之消失，而且哪里都不会报错。这是顺序问题中最具体的多语言故障。不同语言的客户端库并非都采用同一个默认哈希函数，所以用一种语言写的服务和用另一种语言写的 relay，可能把同一个 `order_id` 发往不同的分区。有两种防御办法。其一，在每个生产者中显式固定 partitioner，使用相同的算法和相同的键编码。其二，把这个决定从生产者手中拿走：让 relay 根据存储的 `partition_key` 自己计算分区，由一段代码掌管这一映射。当每个事件本来就要经过第 5 章的 outbox relay 时，第二种做法是顺理成章的。

基于队列的 broker 则不同。一旦一个队列上有多个竞争消费者，就得不到任何顺序保证，因为 broker 会把下一条消息交给任何空闲的消费者。一些基于队列的 broker 提供分组功能，可以把共享同一组标识符的消息串行化，例如 Amazon SQS FIFO 队列上的消息组；另一些则提供一种模式，让一个队列上同一时刻只有一个消费者处于活动状态。这些功能都有效，每一种都以不同的方式牺牲并行度。要查清在该功能下，所用 broker 对重新投递和 visibility timeout 承诺了什么，因为这类保证往往附有小字条款。要说明所依赖的是哪一类 broker，以及这一类真正承诺了什么。

选择键

键是聚合的标识符，选择它就是在声明一致性存在于何处。对 Parcelly 的订单生命周期而言，键是 `order_id`。推动订单经历各个状态的每个事件都带着它，这使 Shipping、Orders 和 Ledger 得到一致的按订单序列。

当答案并不显而易见时，几个问题会有所帮助：

- 消费者改变的是哪个实体的状态？就按那个实体取键。
- 两个事件可能发生冲突的最小单位是什么？那就是聚合。
- 同一个 stream 是否被多个服务消费，而它们的聚合各不相同？那么要么由生产者选取最细的公共聚合，要么每个消费者需要自己的视图。

Inventory 体现了这种张力。库存水平属于某个 SKU，所以 Inventory 自己的命令自然按 `sku` 取键。从 Orders 流向 Inventory 的事件则按 `order_id` 取键。一条预留事件同时携带这两个标识符，如果 Inventory 需要按 SKU 的顺序，就在内部重新取键。为系统中的每个 topic 选同一个键很诱人，但这白白浪费了设立聚合的意义。

热点分区

分区键只有在各个键同样繁忙时，才能均匀地分散负载。而它们并不是。设想 Parcelly 的一场限时抢购，某位卖家的一件商品在一小时内收到了一天中的大部分订单。如果 Inventory 对自己的 topic 按 `sku` 取键，这一个键就产生了大部分流量，全部落在一个分区上，一个消费者被要求承担整个 group 的工作，其余的却在空转。

增加分区无济于事。单个键始终映射到单个分区，增加分区只会让其他键在它周围移动。有四种坦率的应对办法：

- **把聚合缩小**。如果业务规则允许，就按更细的东西取键，例如用预留 id 而不是 SKU。放弃 SKU 内部的顺序以换取分散，只有在没有任何规则要求按 SKU 保证顺序时，这才可以接受。
- **在消费者上投入，而不是在分区上**。让每条消息的处理成本很低，使一个消费者能够跟上热点键。
- **隔离**。把已知的热点键路由到专用的 topic 或分区，使它们不会饿死共享该分区的普通流量。
- **对该 stream 接受放宽的顺序**。给键加一个随机后缀以分散负载，并依靠版本检查（本章后文）在消费者处修复顺序。这是最灵活的，也是最费工夫的。请留意加盐能买到什么、买不到什么：它分散的是 broker 上的负载，但版本检查会在消费者数据库中该聚合的行锁上再次把这个键串行化。只有当昂贵的工作发生在串行化步骤之前，或者各操作满足交换律时，吞吐量才会提高。

选择之前，我会先测量键的倾斜程度。在平静的一周里看起来均匀的键分布，在繁忙的一天可能严重失衡，而第一个迹象通常是恰好某一个分区上的 consumer lag（第 13 章）。

并行与顺序相互拉扯

consumer group 通过把每个分区交给一个消费者来扩展，所以分区数量就是并行度的上限。如果 Shipping 有六个分区，六个消费者可以并行工作。第七个只会闲着。

团队常常尝试在单个消费者内部通过把消息交给线程池来提高吞吐量。这看似无害，却破坏了顺序。针对同一个订单、先后读到的两条消息被交给了两个 worker，较慢的那个 worker 排在第二个完成。分区按顺序交付了它们，而处理过程并没有保住这个顺序。

两者兼得的办法，是跨键并行、在键内串行。在消费者内部，把键哈希到固定的通道，并给每个通道恰好一个 worker。

```
function pollLoop():
    while running:
        // bounded lanes: a slow lane pauses intake instead of filling
memory
        if anyLaneAbove(HIGH_WATER):
            pause(assignedPartitions)
        else if allLanesBelow(LOW_WATER):
            resume(assignedPartitions)
        for message in poll(maxRecords: POLL_LIMIT):
            // the same key always maps to the same lane, so per-key
order survives
            lanes[hash(message.partition_key) mod
LANE_COUNT].enqueue(message)
        // commits are per partition and happen here, on the polling
thread
        for partition in assignedPartitions:
            commit(partition, contiguousDoneOffset(partition) + 1)

function runLane(lane):
    while running:
        message = lane.dequeue()
        try:
            handleMessage(message)
        on error as failure:
            // in place or aside, as decided in Chapter 6
            decideInPlaceOrAside(message)
        markDone(message.partition, message.offset)
```

有三个细节支撑着这一设计。第一个是 offset。提交按分区进行，而 `contiguousDoneOffset` 是该分区中这样一个最高的 offset：低于它的每条消息都已完成。当各通道以不同的速度完成时，若提交最快通道的 offset，崩溃之后就会跳过较慢通道尚未完成的消息。提交发生在轮询循环中，是因为许多日志客户端无法安全地从多个线程调用。消费者内部并行的代价，就是这套记账工作，以及崩溃之后的重复消息，后者由第 7 章的 inbox 处理。

第二个是上限。没有上限，慢通道会让提交点原地不动，而轮询循环则不断填满内存，已提交的 offset 与在途工作之间的差距，恰好就是崩溃后重新投递的消息数。当某个通道越过它的高水位线时暂停分区，就把过载变成了 lag，这是可见且安全的，而不是内存增长，后者两样都不是。这套机制，以及在暂停与削减之间的选择，正是第 6 章“过载下的 retry：反压”中所述的那套。

第三个是故障。如果 `handleMessage` 失败而通道在原地 retry，共享该通道的每个键都要在它后面等待，已提交的 offset 也停止前进。把消息送到一旁则让通道得以继续，却会让该键的顺序被打乱。这与第 6 章对整个分区所做的决定相同，只不过现在是按通道来做，而答案同样取决于消费者。

在基于队列的 broker 上，同样的思路出现在更上一层。如果需要跨竞争消费者的按键顺序，那么要么使用 broker 的分组功能，要么接受队列会乱序投递，并在消费者中处理。不要因为消息是有序进入的，就假定有多个消费者的队列会保持顺序。它不会。

用版本检测乱序到达

有时 broker 无法给出所需的顺序，原因可能是 retry topic、redrive、并行的消费者，或者分区数量的变化。在这些情况下，消费者必须自己察觉。为此，事件必须携带消费者能够比较的东西。

时间戳是不该拿来比较的东西。第 3 章解释过，`occurred_at` 反映的是某一台机器的时钟，用它来决定哪次更新胜出，会让时钟走得快的服务赢下它本该输掉的每一次冲突。`occurred_at` 应当用于人和分析。

真正管用的是**版本号**（也称序列号）：聚合在每次变更时递增的计数器，与状态变更写入同一个事务，并随事件一起携带。因为只有所属服务才会改变聚合，而且是在事务内进行，所以版本在每个聚合内形成无缺口的序

列：1、2、3。无缺口还需要反方向的一条规则：**每次版本递增恰好产生一个已发布的事件**。一次内部更新如果递增了版本却什么也不发布，就会留下永久的缺口，而每个消费者都会等待一个并不存在的事件。

标准信封里没有给它留字段，我也不会为了一章的内容去改动信封。我会把它放在 payload 中，对 Orders 事件命名为 `order_version`。它就是聚合版本（第 4 章），与 outbox 行所存储的是同一个数字。关于信封设计的更深入讨论，我留到第 11 章。计数器属于一个生产者和一个聚合。消费者只在针对同一订单的 Orders 事件之间比较 `order_version` 的值，绝不拿它去和来自 Payments 或 Inventory 的事件比较，那些事件需要它们自己的计数器。服务之间的依赖，不能靠共享序列来解决，而要靠状态检查或前置条件检查来解决，就像扣款步骤等待 `PaymentAuthorized` 和 `StockReserved` 二者都到达那样。outbox 行可以顺带得到它，因为服务把新版本、状态变更和 outbox 行写在同一个事务里。

消费者的判断

保留着某个聚合视图的消费者，也会保留它最后应用的版本。对于每个到达的事件，它比较版本，并走三条路径之一。

到达的版本，与最后应用的版本相比	含义	动作
相等或更低	重复或过期	忽略，计数，ack
恰好高一个	下一个事件	应用，记录新版本
高出不止一个	缺口：有东西缺失或迟到	缓冲该事件，不应用

```
function handleOrderEvent(message):
    key = message.partition_key
    transaction:
        view = loadOrderView(key)
        // loadOrderView locks the row; a concurrent worker waits
here.
        // For the first event there is no row yet: last_version
defaults
        // to 0 and the key's unique constraint settles a concurrent
insert.
        version = message.payload.order_version
        if version <= view.last_version:
            countIgnored(message)
            // a lower version was already applied in sequence: a
duplicate
        else if version == view.last_version + 1:
            applyToView(view, message)
            view.last_version = version
            saveOrderView(view)
            // applying may unblock held events (drain is defined
below)
            drainHeldEvents(key)
        else:
            // insert-if-absent: a redelivered held version must not
fail
            holdEvent(key, version, message)
    // the commit covers the view and the held event together
    ack(message)
```

有两条性质使它是安全的。版本检查与状态更新原子地提交，所以处理同一个键的两个 worker 不可能都应用版本 4。而且这个检查是幂等的：重复消息就是一个与其他过期版本无异的过期版本，会被忽略，所以它是对第 7 章 inbox 的补充，而不是取代。`holdEvent` 也必须是幂等的。对已被暂扣的版本再次重复投递，否则会撞上缓冲区的唯一约束，使事务失败，让消息保持未 ack，最终变成 poison 循环。

有两个前提条件常被忽视。第一个是序列的起点。一个从流的中途开始的消费者，例如新服务或从较晚 offset 开始的重放，会先看到版本 5，并将永远等待版本 1 到 4。应当用快照或所属服务的 API，连同该快照所反映的版本，为它的视图播下种子，并从那里开始比较。

第二个是，只有当消费者看到每一个版本时，缺口检测才有效。只订阅某个聚合部分事件的消费者，比如只订阅 `OrderCancelled`，会看到版本 3 和 7，并错误地得出 4 到 6 缺失的结论。对这样的消费者，规则弱化为“绝不应用任何不比我已有的更新的东西”，它仍然能防止过期覆盖，却无法检测缺口。

后写入者胜出的陷阱

最简单的冲突策略是最后一次写入胜出。它很简单，却隐藏着三种不同的陷阱，取决于“最后”指什么。

最后到达的。视图被消费者最近处理的那条消息覆盖。一条重新投递的 `OrderPlaced` 如果在 `OrderConfirmed` 之后到达，会把状态改回 `PLACED`，这正是第 7 章所指出的故障。到达顺序，恰恰是 broker 拓扑可能不作承诺的那一件事。

时间戳最晚的。看上去更好，实际上更糟。服务之间的时钟偏差意味着，胜出的事件可能只是时钟更快的那一个。在不同机器上相隔几毫秒写入的两个事件，其时间戳顺序毫无意义。

版本最高的。这是可行的，但有它自己的局限。只有当较新的事件已经包含了过期事件所说的一切时，丢弃过期事件才是正确的。对携带状态的事件（“订单现在是 `CANCELLED`”），这成立。对携带变化的事件，比如“余额加 4,990”，则不成立。忽略一个过期的变化事件，就丢掉了这个变化。

有一点澄清，可以避免它与上表相矛盾。版本无缺口且严格按序应用时，较低的版本必然是重复消息，因为它已经被应用过；即使对增量事件，忽略它也是正确的。增量陷阱出现在消费者跳到最新版本、略过缺口之时，因为被跳过的变化就此丢失了。

这一区别决定了遇到过期或乱序版本时的做法：

- **携带状态的事件**（完整的状态，或会覆盖前一个状态的状态）：忽略过期事件是正确的，因为较晚的那个取代了它。
- **增量或转换事件**（增量、账本分录以及任何不会取代前者的东西）：绝不要丢弃。应缓冲并重排，或者在操作满足交换律时以任意顺序处理。尤其是 Ledger，绝不能因为某条分录较旧就丢弃它。

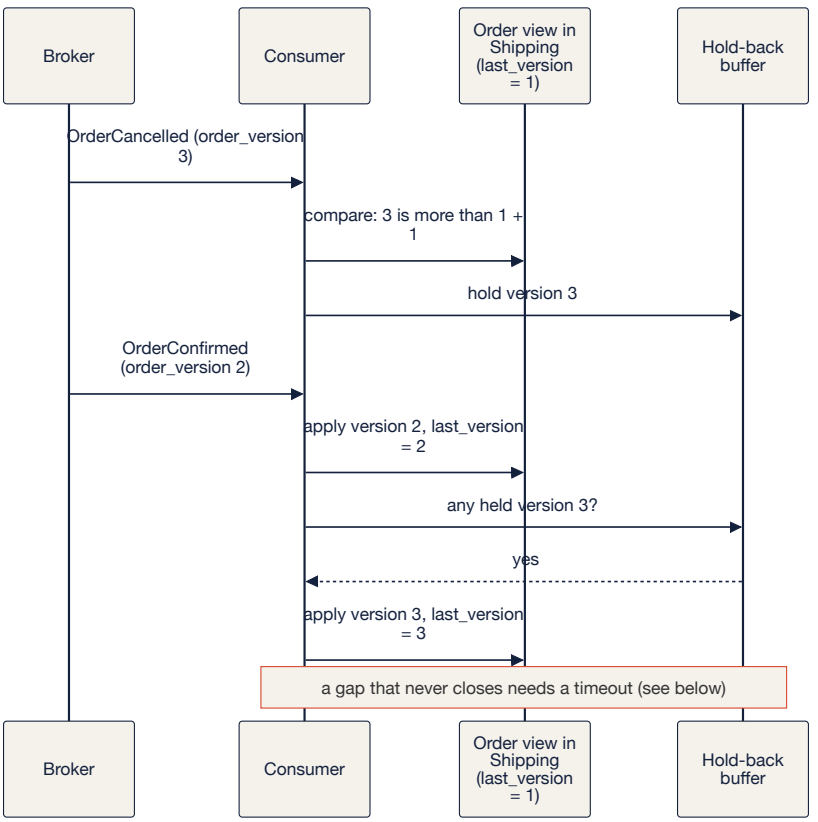
一个相关的陷阱是被删除的聚合。如果消费者在看到 `OrderCancelled` 时删除了自己的行，迟到的 `OrderPlaced` 就会发现没有行，因为没有东西可比较而通过版本检查，进而创建一个新订单。复活，是在删除前后忽视顺序时的典型 bug。应当保留一个 tombstone：键、最后的版本和一个已删除标记，至少保留到过期事件不可能再到达为止。第 8 章展示过同样的复活：被 redrive 的 `OrderPlaced` 在取消之后到达，而带版本检查的 tombstone 正是阻止它的东西。

最后，忽略是一个值得留下痕迹的决定。被悄悄丢弃的过期事件，是一个没人看得见的 bug。应按消费者统计被忽略事件的数量，并在日志行里记录键和版本，使计数的激增能在客户发现之前显现出来（第 13 章）。

缓冲与重排

当消费者看到缺口时，它有两条路可走。它可以照样应用该事件，冒着状态出错的风险；也可以把事件暂扣起来，等缺失的那个到达。第二种做法就是**暂扣缓冲区**。

下面的时序图展示了 Shipping 为 `order-81724` 收到来自 Orders 的两个事件，版本 3 先于版本 2，如开头的场景那样。两者来自同一个生产者和同一个聚合，这正是比较它们版本有意义的原因。Shipping 只对 `OrderConfirmed` 采取行动，但为使这一做法奏效，它会读取该订单的整个 Orders stream，并记录每个事件的版本，包括它本来会忽略的类型。这使视图的 `last_version = 1` 成为真实的，也使缺口检测成为可能。请留意版本 3 在哪里等待，又是什么释放了它。



释放这一步是一个循环。应用任何事件之后，消费者检查缓冲区里是否有下一个版本，应用它，并重复，直到缓冲区里没有后继者。

```
function drainHeldEvents(key):
    // runs inside the same transaction that applied the triggering
event
    while true:
        view = loadOrderView(key)
        next = findHeldEvent(key, view.last_version + 1)
        if next is null:
            return
        applyToView(view, next.message)
        view.last_version = next.version
        saveOrderView(view)
        deleteHeldEvent(key, next.version)
```

缓冲区本身是一张以聚合和版本为键的表，所以它能在重启后保留。内存中的缓冲区会随进程一起消失，而 broker 早已把这些消息视为已 ack。

清理还需要一条规则。每当 `last_version` 发生跳变，例如用获取到的状态替换了视图，`version <= last_version` 的暂扣行就已作废，消费者要在同一个事务中删除它们。否则它们会永远留在表里，而排空循环永远不会到达它们。

永不闭合的缺口

缓冲是乐观的：它假定缺失的事件只是迟到了。有时它丢失了，被转入死信了，或者从未发给这个消费者。没有上限的话，聚合会永远被阻塞，暂扣集合也会不断增长。

每个缓冲区都需要一个超时，以及对超时到期后怎么办的决定。有三种合理的答案，我会根据出错的代价来挑选：

- **获取缺失的状态**。通过 API 向所属服务询问当前状态，并替换视图。当生产者提供这样的 API 时，这是可行的，并使缓冲区成为带有安全网的快速路径。
- **告警并等待**。由人来看。这适合 Ledger，那里一条错误的分录比一条缓慢的分录代价更高。
- **把视图标记为可疑，交由对账来修复**。这适合能够容忍过期的读模型（第 13 章）。

第 8 章的按键阻塞，是这套机制不带版本的形式。在那里，死信标记阻塞了键，后续事件在暂扣存储中等待，一条特殊的旁路让被 redrive 的事件通过以释放它们。在这里，被转入死信的事件造成一个缺口，后续事件在缓冲区中等待，而被 redrive 的事件自己就能释放它们，因为它以 `last_version + 1` 的身份到达，排空循环会做完其余的事。版本取代了标记和旁路，这就是为什么在生产者能够提供版本的地方，这是更强的设计。

成本是实实在在的：一张需要确定大小并清理的缓冲表，一套需要调节的超时策略，以及一类需要有人为之被呼叫的新型卡死聚合。如果事件携带完整状态，而消费者可以安全地应用“最新者胜出”，缓冲区就是不必要的，我会把它省掉。

Retry、死信与顺序

第 6 章和第 8 章各自以一个有关顺序的问题收尾，并把它留到了这里。简而言之，每一种故障处理机制都会在时间上移动一条消息，而这就是一次重排。

- 日志分区上的**原地 retry** 会阻塞后继者，所以顺序得以保留，而整个分区在延迟上付出代价。
- **retry topic 或延迟队列**让后继者先行，所以失败的消息会比它本应在其前面的那些事件更晚返回。
- 基于队列的 broker 上的 **visibility timeout**，会在超时之后把消息送回队列，在那里它与在它之后发送的消息竞争。
- **死信**跳过这条消息，而 **redrive** 会把它带回到其后的一切之后。

版本检查，正是使第二到第四种选项可以承受的东西。拿第 6 章给出的例子来说：Inventory 从 Orders 为同一个订单先后收到 `OrderPlaced` 和 `OrderCancelled`，而 `OrderPlaced` 在一次瞬时故障后被移入 retry topic。有了来自 Orders 的聚合版本，`OrderPlaced` 是版本 1，`OrderCancelled` 是版本 2。取消先到达，此时 Inventory 还没有该订单的视图，所以 `last_version` 默认为 0，而 2 大于 0 + 1。取消被暂扣，而不是被应用到一个并不存在的预留上。当 retry 的 `OrderPlaced` 以版本 1 到达时，它被应用，排空循环释放了取消。第 6 章所描述的那个游离的预留就不会出现。

这只对保留版本和缓冲区的消费者有效。不这样做的消费者必须像第 6 章所论证的那样，在阻塞分区与容忍重排之间做出选择。而第 8 章对账本的建议依然成立：对于迟到或缺失的一条分录就意味着一份错误财务记录的消费者，应停下来等待。被暂停的消费者声势浩大，而悄悄地乱序应用分录的 Ledger 则不会。

> **警告**：retry topic 加上没有版本检查的消费者，会悄悄地重排事件。没有东西失败，没有告警触发，状态却是错的。如果把失败的消息路由到一旁，就要让版本检查成为同一次变更的一部分。

更改分区数量

某个键所在的分区，是键和分区数量的函数，通常是哈希对数量取模。改变数量，函数也就变了。

举例来说，假设某个键的哈希值为 17。有四个分区时，它映射到分区 1；有六个分区时，它映射到分区 5。哈希值 20 从分区 0 移到分区 2，而 14 和 15 留在原处。大多数键都会移动。在取模哈希下，从四个分区增加到六个时，只有哈希值对 12 取模为 0、1、2 或 3 的键，才会保留它的分区，所以大约三分之二的键会落到新的位置。这种故障并不罕见：变更期间，每个有在途消息的键都会受到影响。

危险在于重叠。在变更之前写入的、属于正在移动的键的消息，位于旧分区上，仍未被消费。变更之后写入的消息则去往新分区。现在有两个消费者在同时处理同一个键，二者之间没有顺序。如果消费者还按键保留本地状态，比如缓存或聚合窗口，那么这份状态就在错误的实例上。

应对办法，按大致的优先顺序：

1. **一开始就预配足够的分区**。针对预期的并行度峰值预留充足，因为这个数字保留起来很便宜，改起来很昂贵。许多基于日志的 broker 允许增加分区，却不允许删除，所以向某一个方向的失误是永久的。
2. **在变更之前先排空**。停止生产者或暂停 stream，让消费者到达旧分区的末尾，然后再更改数量。顺序得以保留，代价是一段不可用的窗口期。
3. **迁移到新 topic**。创建一个分区数量正确的 topic，执行切换，让消费者先读完旧 topic 再开始读新 topic，然后下线旧的。这是最安全的，也是最费工夫的。

4. **依靠版本检查**。如果每个消费者都能检测并缓冲乱序事件，这一过渡无需暂停就能承受，而且这次过渡本身也是检验该检查是否有效的一次好测试。

Rebalance 与撤销

consumer group 的 rebalance 与此形状相似，只是规模更小，而且每次部署都会发生。当一个分区转移给另一个消费者时，旧消费者上的在途工作可能与新消费者上的最初几条消息重叠。消费者在撤销时的做法，决定了这个重叠有多大。

撤销时，消费者停止分发来自被撤销分区的新消息，让通道完成手头的工作或放弃它，并在交出分区之前提交连续的 offset。一次 rebalance 是撤销每一个分区，还是只撤销要移动的那些，取决于客户端的分配策略，而一次性全部撤销的策略会让整个 group 在这段时间内停滞；要查清实际运行的是哪一种。对失去了某个分区反应迟钝的消费者，会像僵尸一样继续处理，而新的所有者在做同样的工作，所以无论怎么做，都要预期会有一些重叠。

inbox 和版本检查，只对发生在消费者自己的数据库事务内部的效果，才能让这种重叠无害。对于两个消费者都发出的邮件或承运商 API 调用，它们毫无作用。这些外部副作用需要它们自己的幂等键，如第 7 章所述，因为 inbox 只保护本地状态，别无其他。

Parcelly 的各个消费者需要什么

并非每个消费者都需要相同的顺序保证，一视同仁既是 bug 的来源，也是浪费的来源。下面这张针对 Parcelly 绘制的表，展示了我会如何决定。

消费者	顺序需求	机制	如果有消息缺失
Notifications	无；每条消息都是独立的事实	任意顺序，无需键	转入死信并继续
Orders 状态视图	按订单，携带状态	按 `order_id` 取键，忽略过期版本	获取当前状态
Inventory 预留	按订单，转换	按 `order_id` 取键，版本加暂扣缓冲区	暂扣该键，告警

消费者	顺序需求	机制	如果有消息缺失
Shipping	按订单，`OrderConfirmed` 先于 `OrderCancelled`	按 `order_id` 取键，版本加暂扣缓冲区	暂扣该键，告警
Ledger	按订单，无缺口，无丢弃	按 `order_id` 取键，在 Payments 拥有的按订单序列出现缺口时停止	停下来等待人工

Ledger 一行有意点明了它的计数器。Ledger 消费 Payments 事件，所以它比较的是 Payments 按订单递增的序列，而不是属于 Orders 的 `order_version`。计数器只能在拥有它的那一个服务的事件之间进行比较。

第一行成本最低，也理应得到比现在更多的认可。处理顺序最好的办法，是把消费者设计成顺序无关紧要。满足交换律的操作，比如向集合中添加一项，或取两个值中的最大值，无论以什么顺序都得到相同的结果。由这类操作构成的消费者，不需要键、缓冲区和版本。当能把处理器改造成这样时，我会先这样做，再去动用那套机制。

权衡与成本

这是在用并行度换取顺序，而汇率就是聚合。更粗的键给出更强的顺序和更少的分散。更细的键给出分散和更弱的顺序。没有哪个键能两者兼得，这个决定属于了解业务规则的人。

热点键是设计成本，而不是运维意外。从业务标识符中选取的分区键，会继承业务本身的倾斜。一场限时抢购就会把它暴露出来。可以缓解它，但除非放弃该键的顺序，否则无法消除它。

版本对生产者提出纪律要求。聚合必须在与状态变更相同的事务中递增版本，对每一次变更、每一条代码路径都是如此。只要有一条路径忘了，就会留下消费者永远等待的缺口。反过来也一样：每次递增都必须恰好发布一个事件，否则序列就有一个任何消费者都无法闭合的缺口。backfill、管理工具和数据迁移，是这件事最容易出错的地方，而从流中途开始的消费者，在比较任何东西之前，需要一个播好种子的视图。

缓冲区增加状态，也增加一种新的故障。暂扣表需要确定大小、清理、超时、负责人和告警。它也是卡死的聚合可以藏身的又一个地方。对于可以应用“最新者胜出”的消费者，就跳过它。

版本检查不能在聚合或 topic 之间排序。它们保护的是一个聚合的序列。如果某条规则需要知道 topic A 上的一个事件先于 topic B 上的一个事件，版本号提供不了这一点。那是一个工作流问题，第 10 章把它当作工作流问题来处理。

何时不必费心。如果消费者满足交换律，如果事件携带完整状态而“最新者胜出”是正确的，或者迟到的事件无害、周期性对账会修复它，就跳过这套机制。少做是一个有成本的决定，而且应当是有记录的决定。

常见错误

- **要求全局顺序**。它感觉像是安全的默认选择，却把吞吐量限制在一个读取方。应说明真正的要求，即按聚合的顺序，并围绕它设计。
- **用不是聚合的东西取键**。按事件类型、按服务或按随机值取键，是因为默认设置就是这么做的。解决办法是按其状态被消费者改变的那个实体的标识符取键。
- **用时间戳给事件排序**。它们看起来像是天然的版本号。时钟偏差使这个顺序毫无意义，所以应当改为携带来自聚合的版本。
- **在消费者内部并行却没有按键划分的通道**。线程池提高了吞吐量，却破坏了分区的顺序。应把键哈希到通道，并给每个通道一个 worker。
- **丢弃过期的增量事件**。版本检查适合携带状态的事件。把它用于增量和账本分录，会丢弃真实的变更。选择策略之前，先把每个事件分类为状态事件或增量事件。
- **忘记 tombstone**。取消时删除一行，会让迟到的事件重新创建订单。删除之后，应在过期事件可能到达的整个期间内，保留键和最后的版本。
- **把事件放在内存里暂扣**。随进程一起消亡的缓冲区，会丢掉 broker 已视为送达的事件。应把它持久化，并与视图在同一个事务中。
- **没有计划就在生产环境中更改分区数量**。从四个分区变成六个时，大约三分之二的键的映射会改变，顺序也随之悄悄改变。应排空、迁移，或用版本来防护。
- **在生产者之间混用不同的 partitioner**。不同语言的服务可能对同一个键做出不同的哈希。应在各处固定同一个 partitioner，或在 relay 中计算分区。

- **没有上限的通道**。慢通道随后会填满内存，而提交点原地不动。应给每个通道设上限，暂停分区，并在轮询循环中按分区提交。
- **在 rebalance 时指望 inbox 管住外部效果**。inbox 只覆盖消费者自己的数据库。邮件或承运商调用需要它自己的幂等键。
- **以为共享的键能在 topic 之间给事件排序**。两个 topic 上的同一个 `order_id`，在它们之间没有任何先后。应让消费者等待二者都到达，或在工作流中为这一依赖建模。

要点

- 全局顺序会让全部并行度丧失，而且很少需要；按聚合的顺序才是值得为之设计的顺序。
- 标识聚合的分区键，能在基于日志的 broker 上为每个实体提供有序的事件，但前提是各处使用相同的 partitioner 和相同的分区数量；基于队列的 broker 需要分组功能，或需要能容忍乱序的消费者。
- 消费者内部的并行必须按键划分，每个通道一个 worker，有上限，并按分区提交，否则它会悄悄抹掉分区所交付的顺序。
- 由聚合在其事务内递增的版本号，使消费者能检测过期、重复和缺失的事件，而时间戳做不到。
- 忽略过期事件，对携带状态的事件是正确的，对增量和账本分录则是错误的，后者必须被缓冲，或以任意顺序应用。
- retry topic、redrive 和分区变更都会重排消息，所以任何依赖顺序的消费者，在使用它们之前都需要版本检查。
- 每个暂扣缓冲区都需要超时、负责人和告警，因为永不闭合的缺口就是一个卡死的聚合；第 8 章的按键阻塞，是同一思路中不带版本的形式。

顺序为每个事件在一个聚合之内安排了位置，但一个业务流程会跨越多个聚合和服务，而且当后面的步骤失败时，它的某些步骤必须被撤销。第 10 章转向 saga，其中步骤的顺序就是设计，补偿则是工作流恢复的方式。

第四部分

工作流与契约

第 10 章 — Saga 与长时间运行的工作流

设想 Parcelly 的一个周二下午。一位买家下单购买一盏陶瓷灯。Orders 发出 `OrderPlaced`，Payments 对银行卡预授权，Inventory 为卖家预留库存中的最后一盏灯。随后银行卡被拒，Payments 发出 `PaymentFailed`。与此同时，另一位买家正盯着商品页面上的“缺货”提示，因为第一笔订单的预留仍占着那盏灯。

没有人做错什么。每个服务都完成了自己的工作，每个本地事务都已提交，每条消息都已投递。问题在于，“把这盏灯卖给这位买家”这项业务操作分散在四个服务和几分钟的时间里，没有任何事务覆盖它。一半已经发生，另一半不会发生，而总得有人把灯还回去。

时机很关键。此时尚未扣款，所以撤销的成本很低：释放预留，让订单结束即可。倘若这盏灯已经付款，失败发生得更晚，那么归还就是另一种、代价更高的操作，本章稍后会回到这条界线。

现在假设同样的故障发生在卖家正在取消商品上架、买家正在修改送货地址，或扣款响应在返回途中丢失的时候。本章讨论的，就是让这样的流程干净地完成或干净地撤销自身的那套机制。工作流跨越多个服务，无法在其上持有锁，而且任何一步都可能失败，如何才能让它结束在业务可以接受的状态？

问题及其难点

在一个数据库之内，事务提供原子性：要么所有变更都发生，要么都不发生。跨服务则没有这种保证。每个服务拥有自己的数据库，二者之间隔着消息 broker。第 4 章说明了如何让单个服务的状态变更与其发布的事件成为原子操作。但它没有让下一个服务的反应与第一个服务的变更成为原子操作，也做不到这一点。

旧的答案是分布式事务，通常是两阶段提交。它在每个参与者上持有锁，直到协调者做出决定；如果协调者在参与者投票之后发生故障，参与者就带着这些锁一直等到它恢复。它要求参与者支持该协议，并使最慢或可用

性最低的参与者成为所有参与者的上限。外部支付服务商或承运商 API 根本不会参与自己的事务。对于包含第三方调用的流程，这一选项在开始权衡之前就已经被排除。

Saga 是替代方案。它把一项业务操作拆分成一系列本地事务，每个服务一个。某一步失败时，saga 不会回滚，因为之前的事务已经提交，别人可能已经读取了它们的结果。它执行的是**补偿**：以新的本地事务，在语义上撤销之前的步骤。补偿是一个向前的动作，把业务带回到可以接受的状态，而不是时光机。

实践中，有三件事使它变得困难。

第一，中间状态是可见的。在“库存已预留”与“款项已扣”之间，订单既未确认也未取消，所有读取它的方都会看到这种中间状态。原子性没有了，隔离性也没有了。

第二，补偿是设计决策，而不是机制。有些步骤有干净的逆操作，例如释放预留。另一些只有近似的逆操作，例如为已扣款的付款退款，这会产生手续费，并需要几天时间。还有些没有逆操作：发出去的邮件就是发出去了。

第三，saga 本身也会失败。驱动工作流的进程可能在两步之间崩溃，消息可能到达两次，回复可能永远不会到达，补偿也可能和动作一样失败。第 4 章至第 9 章为单条消息建立的一切，现在都必须对整个流程成立。

> **注意**：采用 saga 之前，有一个有用的检验：列出 pivot 之前的每一步（过了这一步，saga 就只向前推进；本章后面会给出定义），写出它的补偿，并说明代价。pivot 之后的步骤本来就不应有补偿。如果 pivot 之前有若干步写的是“无”或“一张支持工单”，那么这些服务之间的边界可能划错了位置，把这些步骤并入某一个服务的本地事务，可能比协调它们更便宜。

Parcelly 的订单作为 saga

取快乐路径中的流程，把它的步骤按 saga 命名。每一步都是某个服务中的一个本地事务，随后是一个事件，该事件在同一事务中写入该服务的 transactional outbox。

步骤	服务	本地事务	补偿
1	Orders	以待处理状态创建订单，发出 `OrderPlaced`	标记为已取消，发出 `OrderCancelled`
2	Payments	对银行卡预授权，发出 `PaymentAuthorized`	作废预授权，发出 `PaymentVoided`
3	Inventory	预留库存，发出 `StockReserved`	释放预留，发出 `StockReleased`
4	Payments	扣款，发出 `PaymentCaptured`	saga 内无补偿：这是 pivot（退款属于单独的取消流程）
5	Orders	确认订单，发出 `OrderConfirmed`	无：一直 retry 直到成功

严格说来，两个 Payments 步骤并不是数据库事务。每一步都是带着稳定幂等键对外部服务商的一次调用，随后在本地记录结果，这正是第 7 章的模式。saga 的保证建立在这一配对之上，而服务商调用上的超时带有第 3 章所描述的不确定性。

步骤 2 和步骤 3 并行执行，只有在 `PaymentAuthorized` 和 `StockReserved` 都已到达之后，才会请求扣款。Shipping 对 `OrderConfirmed` 作出反应，Notifications 对 Shipping 发出的事件作出反应。我们暂且把后两者视为 saga 之外，稍后再回到原因。

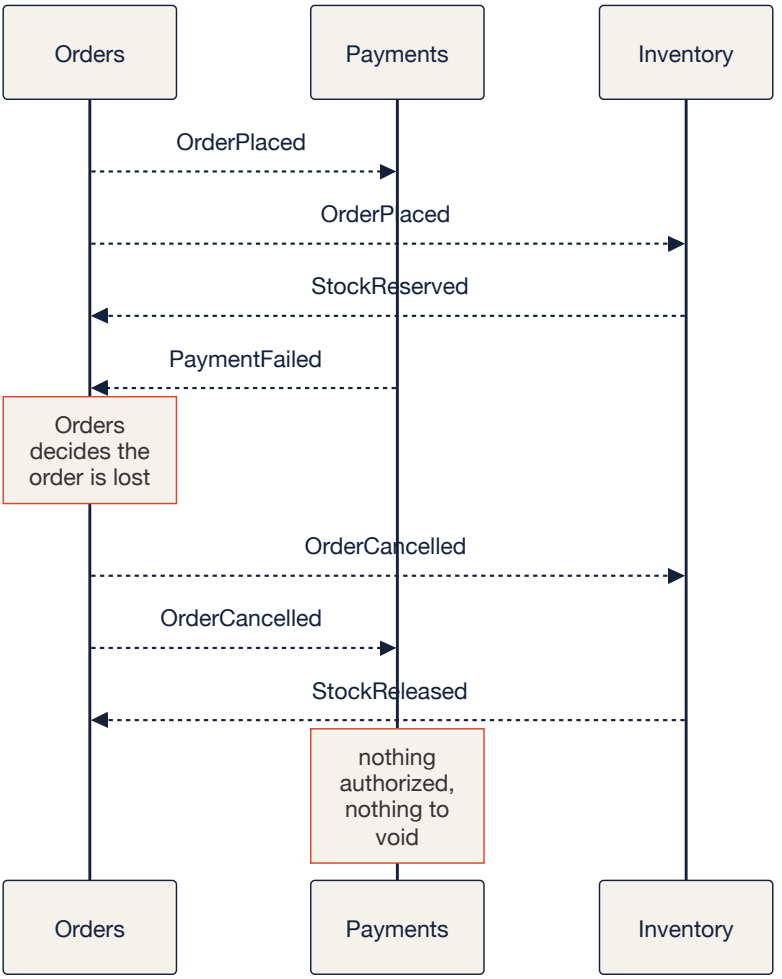
失败分支沿用事件流表中的步骤。如果 `PaymentFailed` 到达，说明不存在预授权，因此唯一的补偿是由 Inventory 释放一个可能已经、也可能尚未建立的预留，并发出 `StockReleased`。如果 `StockReservationFailed` 到达，说明预授权仍被持有而尚未扣款，于是 Payments 将其作废，并发出 `PaymentVoided`。退款只出现在扣款之后。扣款之前，撤销是释放或作废，把它称为退款，在账本上是错的，在买家的银行对账单上也是错的。

在两个失败分支中，Orders 都会作出反应并发出 `OrderCancelled`，其他服务则据此进行补偿。这句话就是整个编舞式协调设计。它是不是所需要的全部设计，是下一个问题。

编舞式协调

在**编舞式协调**的 saga 中，没有中心的负责方。每个服务监听事件，完成自己的本地事务，再发布下一个事件。工作流只作为这些订阅的总和而存在。

下图把支付失败的分支画成编舞式协调。请留意谁在决定什么：没有人下达指令，每个反应都由该服务本已订阅的事件触发。



它的优点是实实在在的。服务之间保持松耦合：Inventory 知道 `OrderCancelled` 的含义，对 Payments 却一无所知。没有哪个单一组件的故障会让所有工作流停下。增加一个消费者，例如对 `OrderConfirmed` 作出反应的分析服务，无需改动其他服务。

随着流程增长，缺点便显现出来。工作流的定义分散在多个代码库的订阅之中，“`StockReserved` 之后会发生什么？”无法通过阅读一个文件来回答。循环依赖悄然出现：Orders 监听 Payments，Payments 又监听 Orders。每增加一个分支，就要增加事件，每个服务都必须按谨慎的顺序更新。最重要的是，没有人负责回答“订单 81724 现在到了哪一步？”。答案是对多个服务日志的一次连接查询，这正是第 1 章指出的问题，第 13 章会借助 tracing 和对账再次回到它。

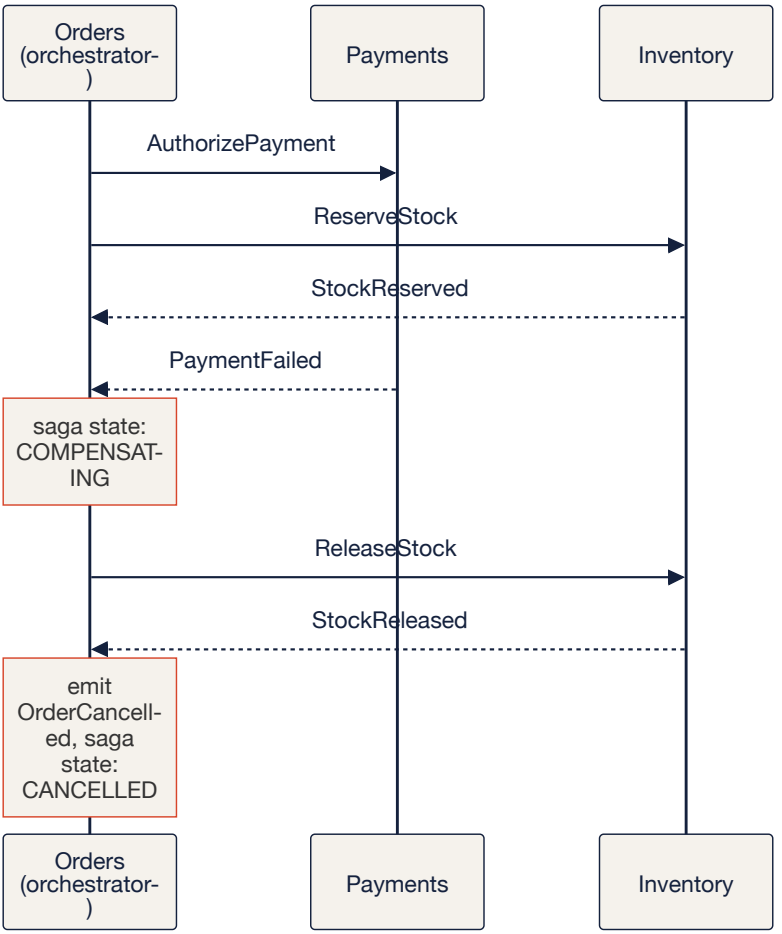
我发现，当流程很短、分支很少，且没有哪一步依赖前面若干步的合取条件时，编舞式协调效果良好。两三个服务对同一个事件作出反应，是它天然适合的场景。Parcelly 的订单流程已经越过了这条线，因为扣款需要两个先前的结果都已到达。

编排

在**编排**的 saga 中，由一个组件负责整个工作流。它持有状态，决定下一步，向参与者发送命令，并对它们的回复作出反应。参与者知道如何完成自己的本地工作并报告结果；它们不知道自己在更大的流程中处于什么位置。

本节引入两条规范之外的命令：`AuthorizePayment`（Orders 发往 Payments）和 `ReserveStock`（Orders 发往 Inventory）。确认不需要命令，因为它是编排器自己的本地事务。与本书中所有命令一样，它们携带 `command_id` 以取代 `event_id`，而回复则是普通事件。在编舞式协调的版本中，Payments 和 Inventory 由 `OrderPlaced` 启动；在这里，则由编排器通知它们。

下图展示与之前相同的失败，但这次从 Orders 发出的每一条箭头都是命令。请留意 saga 状态如何集中在一处，以及补偿命令如何准确指明需要撤销的工作。



编排器是回答“下一步是什么？”的地方。它是一个状态机，而状态机可以阅读、测试和绘制。新增一个分支，只需改动一处。编排器知道哪些步骤已经完成，所以补偿针对的是实际发生过的事，而不是一切可能发生的事。

代价则是它的镜像。编排器认识每一个参与者，成为耦合点。若处理不慎，领域逻辑会从各个服务泄漏到编排器中，参与者就变成了多此一举的远程过程调用。它还是一个必须正确、始终在线、自身也能恢复的组件，因为状态丢失的编排器一旦崩溃，就是一个再也没有人能完成的 saga。

在我看来，位于拥有该业务流程的服务内部的编排器，是最站得住脚的形式。在 Parcelly 中，这就是 Orders，它本来就拥有订单的生命周期。一个为所有人拥有工作流的独立编排服务，往往会变成每个团队发布任何东西都必须修改的那个东西。

如何在二者之间选择

两种风格都不是笼统意义上的更好。选择取决于流程的形态，以及谁必须回答关于它的问题。

问题	编舞式协调	编排
流程定义在哪里？	分散在各个订阅中	集中在一个状态机中
服务之间的耦合	仅通过事件契约	编排器认识每个参与者
增加一个步骤或分支	涉及多个服务	涉及编排器和相应参与者
并行步骤与汇合	别扭；每个服务各自记录自己见过什么	自然；由编排器持有汇合
“订单 81724 在哪里？”	跨日志连接查询	读取一行
单点故障	没有，但也没有负责方	编排器，它必须可恢复
增长时的风险	循环的、隐藏的依赖	塞满领域逻辑的 God Service
适用场景	短流程、少分支	长流程、有条件、超时与汇合

把二者混合使用是正常且合理的。编排的订单 saga 仍然可以把 `OrderConfirmed` 作为普通事件发出，让 Shipping 和 Notifications 以编舞式协调的方式对它作出反应。saga 应当覆盖流程中需要协调撤销的部分；通知和分析不需要，把它们拉进来只会增加状态。

补偿与无法撤销的步骤

设计步骤的同时就要设计它的补偿，而不是等到第一次事故提出要求时才做。对每一步，都要写下 saga 在它之后失败时，业务想要的是什么。

有几条规则可以让补偿保持诚实。

补偿是一个新的业务动作，而不是对数据库的撤销。`StockReleased` 不会删除预留行的历史，它记录的是预留已经结束。`PaymentRefunded` 不会抹去扣款；账本中两条记录都在。这就是为什么 Ledger 是 append-only 的，也是补偿在其中表现为真实事件的原因。

按撤销的难度排列步骤。把容易撤销的步骤放在前面，把难以撤销或无法撤销的步骤放在最后。预授权和预留的撤销成本很低，这就是它们排在扣款之前的原因。扣款是退款开始耗费金钱和时间的地方。由此引出 **pivot**

步骤的概念：过了这一点，saga 不再补偿，只向前推进。在 Parcelly 中，扣款就是 pivot。在它之前，失败意味着释放和作废。在它之后，确认环节的失败不是撤退而是 retry，因为业务已经拿到了买家的钱，就欠下一笔订单。

有些动作没有逆操作。已经发出的邮件、已经收到货件的承运商、卖家系统已据以行动的 webhook：可以再发一条后续消息，但无法撤回。这些步骤属于 pivot 之后，它们的失败应当通过 retry 和更正来处理，而不是补偿。这也是 Notifications 纯粹被动响应、从不属于 saga 的原因：它的步骤无法补偿，也不应当能够阻塞任何事情。

补偿必须 retry 直到成功。动作失败会引发补偿。补偿失败则什么也不会引发，除了一个卡住的 saga 和一个不一致的系统。应当把补偿视为最终必须完成的事情，采用第 6 章的 retry 纪律和第 8 章的 poison message 处理。永久失败的补偿，应当连同记录下的 saga 状态，经由 parking lot 交到人的手上，而不是被丢弃。

补偿必须幂等，并且必须容忍提前到达。前者属于第 7 章的范畴：重复的 `ReleaseStock` 不得释放两次。后者则更为微妙。假设 Orders 发送了 `ReserveStock`，超时后开始补偿，发送 `ReleaseStock`。预留只是慢，并没有丢失，释放可能先到达 Inventory。如果 Inventory 把针对未知预留的释放当作空操作，那么迟到的 `ReserveStock` 随后会成功，这盏灯就永远被占着。安全的做法是，Inventory 针对订单记录下这次释放，使后到的预留看到它并拒绝。有些文献称之为 tombstone。无论叫什么名字，补偿都必须留下动作能够读到的标记。

Saga 状态与超时

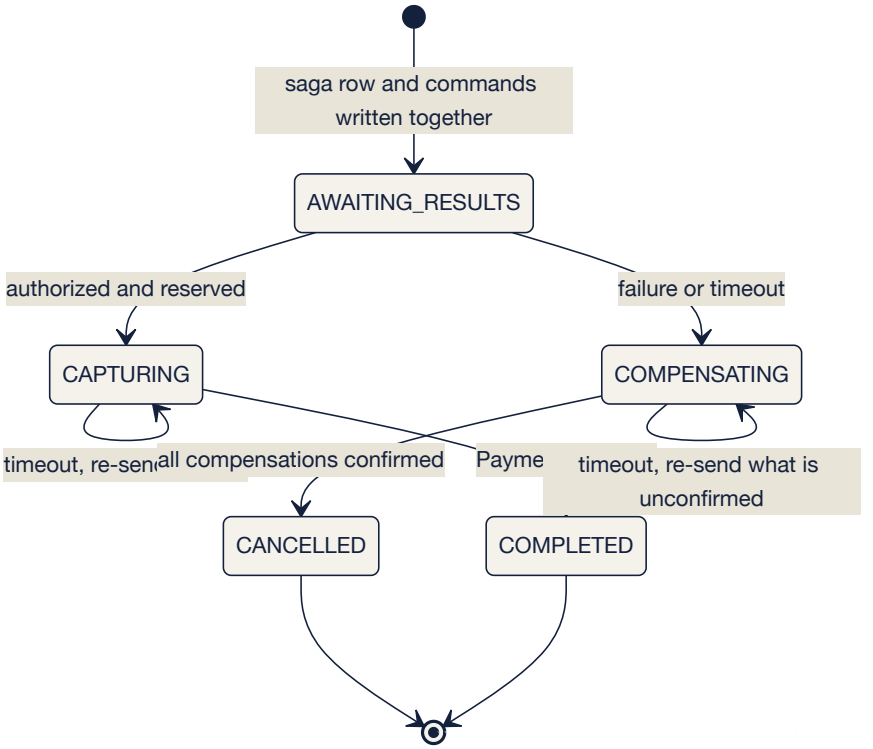
编排的 saga 需要持久化的状态，且它属于运行该 saga 的服务所在的同一个数据库，这样状态变更才能与 outbox 和 inbox 的写入加入同一个本地事务。

```
table saga_instance:
    saga_id        text primary key
    order_id       text unique not null
    state          text not null      // see below
    authorized     boolean not null
    reserved       boolean not null
    stock_released boolean not null   // compensation confirmed
    auth_voided    boolean not null   // compensation confirmed
    rounds         int not null       // deadlines spent in this state
    deadline_at    timestamp not null
    version        int not null
    updated_at     timestamp not null
```

`state` 列是一小组有名字的位置：`AWAITING_RESULTS`、`CAPTURING`、`COMPLETED`、`COMPENSATING`、`CANCELLED`。saga 行与最初的命令写在同一个事务中，因此不存在任何人能够观察到的单独“已启动”位置。两个布尔值 `authorized` 和 `reserved` 记录哪些结果已经到达，汇合就是这样实现的：两者都为 true 且状态为 `AWAITING_RESULTS` 时，才请求扣款。两个标志 `stock_released` 和 `auth_voided` 记录哪些补偿已被确认，因此“所有补偿均已确认”是这一行能够回答的问题。不需要的补偿一开始就视为已确认：`PaymentFailed` 之后没有预授权可供作废，所以进入 `COMPENSATING` 时 `auth_voided` 为 true。

`order_id` 上的 `unique` 约束防止同一订单运行两个 saga。`version` 列防止两个处理器同时更新同一行，下面的处理器都依赖它。

画一次的状态机，胜过任何文字描述。下一张图展示合法的转换；请留意那些不通向 `COMPLETED` 的出口，以及两个因自身超时而循环的状态。



请注意，`CAPTURING` 没有通向 `COMPENSATING` 的出口。这就是用状态机表达出来的 pivot。回复始终没有到来的扣款会被重新发送；如果服务商报告永久拒绝，则作为一条单独、明确、有人参与的业务路径来处理。

循环与出口同样重要。处于 `CAPTURING` 或 `COMPENSATING` 的 saga 在等待回复，而回复和任何其他消息一样可能丢失。如果只有 `AWAITING_RESULTS` 设有截止时间，那么卡在另外两个状态中的 saga 就会停在那里，没有任何东西去唤醒它。所以每个非终态都有截止时间，并且每次转换时都会重置它。

超时

saga 中的每一次等待都需要一个截止时间，因为永远不会到来的回复同样是一种故障，而且是最容易被遗忘的一种。第 3 章描述过超时如何掩盖三种可能：动作失败了、动作成功但回复丢失了、动作尚未完成。saga 无法分辨是哪一种，因此需要为每一次等待写下一项策略。

超时也是一条消息。本章为此引入一个内部事件 `SagaTimedOut`，由编排器自己的定时器发出。常见的实现是一个定时器进程，它选出 `deadline_at` 已过的 saga，并通过与其他所有输入相同的处理器，把一

个 `SagaTimedOut` 事件送给每一个。这样，超时与其他一切走同一条路径，使用同一个 inbox、同样的状态检查和同一个 outbox。

```
function handleSagaTimedOut(message):
    if inboxContains(message.event_id):
        // fast path only; the unique constraint is the safety net
        ack(message)
        return
    try:
        transaction:
            inboxInsert(message.event_id)
            saga = loadSaga(message.payload.saga_id)
            if saga.state == "AWAITING_RESULTS":
                saga.state = "COMPENSATING"
                saga.rounds = 0
                sendOutstandingCompensations(saga, message.event_id)
            else if saga.state == "COMPENSATING":
                saga.rounds = saga.rounds + 1
                sendOutstandingCompensations(saga, message.event_id)
            else if saga.state == "CAPTURING":
                // same command_id as the first attempt: a repeat, not
new work
                saga.rounds = saga.rounds + 1
                outboxInsert(CapturePayment, saga.order_id,
                             command_id: commandIdFor(saga.saga_id,
"capture"))
            // any other state (COMPLETED, CANCELLED): a stale
timeout, recorded and dropped

            // sets a clock for a non-terminal state; escalates past
MAX_ROUNDS
            armNextDeadline(saga)
            saveSaga(saga, expected_version: saga.version)
        ack(message)
    on UniqueViolation:
        // a concurrent duplicate committed first; the work is done
        ack(message)
    on VersionConflict:
        // another handler changed the row; nothing above committed
        nack(message)
```

`AWAITING_RESULTS` 分支发送补偿命令时，并不询问哪一步已经完成。这是有意为之。超时的时候，编排器并不知道迟到的 `StockReserved` 是否还在路上，所以它请求仍未完成的撤销，并依赖参与者把“无需撤销”视为成功。这样做之所以安全，靠的是第 7 章。

`armNextDeadline` 在同一个事务内做两件事。它把 `deadline_at` 设为 `now() + timeoutFor(saga.state)`，使得没有任何非终态不带时钟。而一旦 `saga.rounds` 超过 `MAX_ROUNDS`，比如五次，这个 saga 就不再是时间问题，而成为人的问题：它触发告警，把 saga 状态记录到第 8 章

的 parking lot 中，而轮次仍在继续。终态的 saga 不会再有新的截止时间。由于截止时间、状态变更和命令一起提交，崩溃不会留下一个已经向前推进却不再被监视的 saga。

```
function sendOutstandingCompensations(saga, causing_event_id):
    // asked for again each round until its confirmation has arrived
    if not saga.stock_released:
        outboxInsert(ReleaseStock, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "release",
causing_event_id))
    if not saga.auth_voided:
        outboxInsert(VoidAuthorization, saga.order_id,
                     command_id: commandIdFor(saga.saga_id, "void",
causing_event_id))
```

处于 `CAPTURING` 的 saga 绝不会掉头。pivot 已经过去，唯一的做法是用相同的键重新发送扣款，并询问一个人。

超时多长，不是我能替人决定的。它取决于最慢的健康路径要花多久、买家愿意等多久以及占着库存会让卖家付出什么代价。把截止时间设得很短，会把慢变成失败，并让系统充斥补偿。设得很长，则会在买家看着缺货页面的同时一直占着预留。

迟到的消息同样需要规则。如果 `PaymentAuthorized` 在 saga 已经是 `CANCELLED` 之后才到达，编排器必须再次发出 `VoidAuthorization`，因为有人正在一张银行卡上占着一笔资金，对应的却是一个并不存在的订单。迟到的事件证明预授权确实存在。超时时发出的作废可能仍在途中，可能在预授权刚完成之后才被处理，也可能服务商调用发生在 Payments 检查作废所留下的标记之前。再次发送作废，三种情形都能解决。

只有当这次重发不被误认为重复时，它才有效。这里，两条关于标识的规则交汇了。重复不得引发新工作的命令，例如 `CapturePayment`，在 saga 和步骤上使用同一个 `command_id`，这样 retry 会被识别，并以第一次的结果作答。可以合理地再次发出的补偿，则从触发它的消息取得 `command_id`：saga、步骤以及引发它的 `event_id`。这样，重新发出的作废就是一条新命令。它还必须在目标状态上幂等，而不仅仅在命令上幂等，所以 Payments 把 `VoidAuthorization` 理解为“订单 81724 不得残留任何有效的预授权”并检查这一点，而不是重放第一次尝试时缓存的答案“无需作废”。忽略在错误状态下到达的事件的状态机，会丢失这些情形。应当明确处理它们：针对每个状态和每个事件，决定是应用、忽略还是补偿。

语义锁

由于 saga 放弃了隔离性，其他流程可以在 saga 进行中读取并修改订单。买家可能在扣款待处理时修改送货地址，卖家可能取消，支持人员可能退款。第 9 章的版本号有助于处理过期事件，但它们无法阻止两个流程做出互不相容的业务变更。

语义锁是数据中的一个标志，表示“有一个 saga 正在处理它”。它不是数据库锁，而是一个状态值，其他代码被编写成遵守它。在 Parcelly 中，订单状态 `PENDING` 就是这样一个值：只要它保持，订单就不能被编辑、不能被发货，也不能被退款，订单页面显示“处理中”。同样，Inventory 中的预留就是这盏灯上的一把锁，这正是第二位买家看到“缺货”的原因。

语义锁必须在每条路径上释放：完成时、补偿时和超时时。停滞而没有释放锁的 saga 会让订单被冻结，这是一种悄无声息的故障，最终以支持工单的形式浮出水面。上面的超时是防护措施，第 13 章的对账是第二道防线。

还有两种对策值得了解。一种是尽可能让更新可交换，使顺序无关紧要。另一种是设计一个保守的视图：例如向卖家展示已经扣除待处理预留的库存数量，这样没有人会承诺即将被占用的东西。两者都无法免除一项工作：针对每个状态，决定 saga 进行中其他流程可以做什么。我把这张表，即状态对应允许的操作，视为任何 saga 设计的一部分。

Saga 内部的 outbox 与幂等性

saga 处理器是前面各章交汇之处。它消费一条消息，改变状态，并发出下一条命令或事件。处理不当，它就是两次叠加的双写。

行之有效的形态结合了第 4 章和第 7 章。每个处理器运行一个本地事务，做三件事：把消息记录到 inbox，更新 saga 行，把外发的命令或事件写入 transactional outbox。relay 从那里发布。

```
function handleStockReserved(message):
    if inboxContains(message.event_id):
        ack(message)
        return
    try:
        transaction:
            // unique event_id: a concurrent duplicate fails here
            inboxInsert(message.event_id)
            saga = loadSagaByOrder(message.payload.order_id)
            if saga.state in ["CANCELLED", "COMPENSATING"]:
                // late success after we gave up: undo it
                outboxInsert(ReleaseStock, saga.order_id,
                             command_id: commandIdFor(saga.saga_id,
"release", message.event_id))
            else if saga.state == "AWAITING_RESULTS":
                saga.reserved = true
                if saga.authorized:
                    saga.state = "CAPTURING"
                    outboxInsert(CapturePayment, saga.order_id,
                                 command_id:
commandIdFor(saga.saga_id, "capture"))
                // a conflict rolls back the whole transaction, inbox
row included
                saveSaga(saga, expected_version: saga.version)
            // any other state: irrelevant, recorded and dropped
        // commit point: inbox, saga state and outbox rows are one
unit
        ack(message)
    on UniqueViolation:
        ack(message)
    on VersionConflict:
        // the redelivery will see the other handler's flag
        nack(message)
```

这就是第 7 章为发布消息的消费者引入的规则：inbox 行、状态变更和 outbox 插入在一个事务中提交。如果进程在提交前崩溃，什么都没有发生，broker 会重新投递。如果在提交之后、ack 之前崩溃，重新投递的消息会发现 inbox 行，随即收到 ack，工作被跳过；并发的重复消息在唯一约束处落败，以同样的方式收到 ack。过期或无关的消息同样会被记录并收到 ack，所以它不会再回来。无论消息被投递多少次，saga 对每条消息只推进一次。

`expected_version` 检查使汇合变得正确。`PaymentAuthorized` 和 `StockReserved` 可能同时被两个 worker 处理，各自加载到两个标志都为 false 的一行。没有这个检查，后一次写入会覆盖前一次，saga 就停在 `AWAITING_RESULTS`，直到超时去补偿一个健康的订单。有了它，一个事务落败，连同它的 inbox 行一起回滚，其重新投递会加载到已经带有另一个标志的行。用行锁加载该行，也能达到同样的效果。

saga 发送的命令本身也是至少投递一次的消息，所以参与者需要同样的保护，而 `command_id` 就是它的来源。对于不得重复的动作，从 saga 和步骤名派生它。这使 Payments 能够识别被 retry 的 `CapturePayment`，并以第一次的结果作答。如果编排器在每次尝试时都重新生成一个随机的 `command_id`，每次 retry 看起来都像新工作，买家就可能被扣款两次。可能被重新发出的补偿则遵循延迟到达一节中的另一条规则：标识来自触发它的消息，参与者在目标状态上幂等。信封中携带的 `causation_id` 和 `correlation_id` 也使每个 saga 可被追踪：`correlation_id` 是流程，每个 `causation_id` 是其中的一步。

> **警告**：不带幂等键发送的补偿，与不带幂等键的付款一样危险。没有稳定键而被 retry 的 `RefundPayment` 可能给买家退款两次，而第二次退出去的钱是追不回来的。

工作流引擎适用于何处

上面的一切都可以用一张表、一个定时器和几个处理器来构建，对于 Parcelly 订单这样规模的流程，我会从这里起步。到了某个时刻，这套机制会成为代码的主要部分：持久化定时器、按步骤的 retry、对运行中工作流的版本管理、每个实例的可视化历史。这正是工作流引擎要解决的问题。

这类工具范围很广。有些是库或服务，它们持久化工作流的进度并在崩溃后恢复，代码看起来像顺序执行的程序。另一些则解释一份声明式的状态与转换定义。云服务商提供托管版本，也有若干开源项目。我不会比较具体产品，因为这个领域变化得比书快，而选择取决于所用语言、运维能力以及对多运行一个关键组件的容忍度。

我会对任何引擎提出一份简短的清单。工作流状态存放在哪里，能否检查？它如何让一个步骤的副作用与自身进度成为原子操作，还是把双写留给使用者？当部署了变更后的定义时，已在运行的工作流会怎样？补偿和超时如何表达？把进度存起来、却在单独的事务中运行步骤的引擎，仍然需要幂等的步骤；引擎只是转移了问题，并没有消除它。

诚实的总结是：引擎省去了构建和运维状态机的工作，代价是一项依赖、一套需要学习的模型以及一个可能宕机的新东西。对于一两个小型 saga，一张表更便宜。对于大量带有人工步骤和长时间等待的长时间运行工作流，引擎物有所值。

权衡与成本

saga 用原子性换取可用性和独立性，而代价以多种货币支付。

复杂性。每个 saga 都会让状态成倍增加。一个有三步的流程，每步有一个动作和一个补偿，再加上超时和迟到的消息，可达到的状态远多于快乐路径所暗示的数量，每一个都需要一次决策和一项测试。第 15 章会回到如何测试它们。

可见的中间状态。没有隔离。买家、卖家、支持工具和其他服务都能看到一个做到一半的订单。可以用语义锁和清晰的状态为此做设计，同时接受有些用户无论如何都会看到它。

有意为之的最终一致性。在一次失败与最后一次补偿之间，系统是不一致的，持续几秒，或者在超时设置不当时持续数小时。只有业务方同意，这才是可以接受的。

运维责任归属。必须有人盯着：停留在某个状态超过截止时间的 saga、无法完成的补偿以及最老的运行中 saga 的年龄。没有负责人和告警，saga 会悄无声息地失败。

补偿不是免费的。退款有手续费和延迟，取消的货运可能要收费，而一条告诉买家“订单已取消”的消息本身就是一个可见的事件。应当这样设计流程：让常见的失败撤销起来成本低。

何时不要使用 saga：步骤可以共享一个本地事务时；失败很少见、人工修复比构建补偿更便宜时，对于低流量的内部流程，这是一个合理的选择；所谓“工作流”其实是互相独立、不需要协调撤销的反应时，例如发送邮件或更新搜索索引。还有无法定义补偿的时候。如果一个步骤既无法撤销也无法推迟，诚实的设计问题就是：这些服务之间的边界是否划错了位置。

常见错误

- **在一个事务就够用的地方构建 saga**。这很诱人，因为服务已经存在，图看起来也很现代。如果两个步骤共享同一个所有者和同一个数据库，就把它们合并成一个本地事务。
- **只设计快乐路径，把补偿留到以后**。以后会以事故的形式到来。把补偿写在步骤旁边，并为它发出的事件命名。
- **把每次撤销都称为回滚或退款**。扣款之前，补偿是释放或作废；退款只在扣款之后才存在。用错了，会在账本和买家的对账单中掩盖真实

状态。

- **让 saga 永远等下去**。缺少截止时间，在客户打来电话之前都是不可见的。为每个非终态中的每一次等待设置超时和有名字的结果，包括对扣款回复和补偿确认的等待。
- **忽略在错误状态下到达的事件**。已取消订单迟到的 `PaymentAuthorized` 意味着一张被占着的银行卡。针对每个状态和事件，决定是应用、忽略还是补偿。
- **非幂等的补偿**。被 retry 的释放或退款若执行两次，会在修复第一次事故的同时造成第二次事故。给不得重复的动作一个稳定的 `command_id`，从原因派生可重新发出的补偿的标识，并让参与者在目标状态上幂等。为重发的作废使用一个固定标识，会重放“无需作废”，而让预授权继续被占着。
- **把整个业务放进编排器**。编排器应当决定顺序和结果；规则应当由参与者拥有。如果编排器去校验库存水平，它就在做 Inventory 的工作。
- **把无法补偿的步骤放得太早**。在扣款之前就把货件交给承运商，意味着一旦失败，手里留下的是一票货运而没有钱。按这样的顺序排列步骤：难以撤销的放在最后。
- **漏掉锁的释放**。设在 `PENDING` 上、在超时或补偿时又从未清除的语义锁，会冻结订单。应当在每条退出路径上释放它。

要点

- saga 以一系列本地事务和补偿取代分布式事务，并为此放弃了隔离性。
- 补偿是一个新的业务动作，例如 `StockReleased` 或 `PaymentVoided`，而不是回滚，并且在扣款之前它绝不是退款。
- 编舞式协调适合分支少的短流程，编排适合有汇合、超时和许多分支的流程，二者常常混合使用，编排器位于拥有该流程的服务之内。
- 按撤销的难度排列步骤，并把第一个无法补偿的步骤，例如扣款，视为 pivot，过了这一点，saga 就只向前推进。
- 每个非终态中的每一次等待都需要截止时间，每个状态都需要一条处理迟到事件和重复事件的规则，每个补偿都必须幂等，并容忍先于它所要撤销的动作到达。
- 在处理器内部，一个本地事务应当把 inbox、saga 状态和 outbox 结合起来，由版本检查保护，使并发的处理器无法互相覆盖；对于不

得重复的动作，例如扣款，`command_id` 是固定的，而对于可能被重新发出的补偿，则从触发它的消息派生。

- 工作流引擎可以接管状态机、定时器和历史，但它不能消除对幂等步骤的需要，也不能消除对卡住的工作流明确责任归属的需要。

saga 假定事件以每个参与者都能读懂并信任的形式描述已经发生的事。它们是否做到这一点，取决于每个事件携带什么、叫什么名字、由谁拥有，这就是第 11 章的主题。

第 11 章 — 事件设计

设想 Parcelly 的 Shipping 团队想要更快地调度货运。目前它接收的 `OrderConfirmed` 只带有一个 `order_id` 和寥寥几个字段，然后通过 HTTP 调用 Orders 去获取送货地址和订单行。Orders 在周一上午很慢，Shipping 也就跟着慢下来。一位技术负责人提出了显而易见的修复办法：把整个订单放进事件，让 Shipping 再也不用去问。

不到一周，这个事件就膨胀了。Orders 加入了买家的邮箱地址、完整地址、卖家的付款账户以及一个名为 `internal_status` 的字段，供它自己的管理界面使用。Notifications 开始从事件中读取邮箱，而不是读自己的表。Ledger 团队发现，付款账户现在经由 broker 传输，并存放在保留期限很长的 topic 里。随后，Orders 在重构时重命名了 `internal_status`，三个消费者在同一个下午出了故障。

这个故事里没有人犯下前面各章意义上的可靠性错误。outbox 运转正常，消费者是幂等的，retry 也合理。损害来自消息本身的设计：它说了什么，携带了什么，以及谁有权修改它。本章要问的是，如何设计一个事件，使它在第一个消费者读取之后的许多年里，仍然有用、安全且可以变更。

问题及其难点

第 1 章指出，事件驱动系统中的服务通过事件的契约耦合在一起：它的名称、字段和含义。在编写事件的时候，这种耦合是不可见的，因为那时只有一个生产者和一个消费者，二者由坐得很近的人编写。契约感觉像是内部细节。

第二个消费者订阅的那一刻，它就不再是内部的了，而生产者通常很晚才知道，甚至永远不知道。在基于日志的 broker 上，新的 consumer group 可以不通知任何人就开始读取 topic。在基于队列的 broker 上，有人把新队列绑定到同一个 exchange 或 topic 订阅上。从那一刻起，payload 中的每个字段都是对可能永远不会见到的人所作的承诺。

实践中，有三件事使它变得困难。

- **向事件里添加容易，从中拿走很难**。添加一个字段只需一行代码。删除它则需要找出每一个消费者，包括那些在没人读的代码库里的，以及那些会重放旧历史的。
- **设计压力指向了错误的方向**。生产团队想要发布，所以手边有什么就发出什么。消费团队想要避免额外的调用，所以索要更多字段。没有人的工作是说“不”。
- **错误是持久的**。今天写下的事件可能在事件日志中存放数月，并被重放到一个尚不存在的消费者中。命名糟糕的事件，或泄漏的个人字段，在产生它的代码消失之后很久，仍留在存储里。

所以事件设计不是装饰。系统中有这么一个部分，花几个小时的思考，就决定了之后每一次变更的成本有多高。

事件、命令与文档

第一个设计决策是：所发送的是哪一种消息。常见的有三种，它们的规则各不相同。

事件陈述一个已经发生的事实：`PaymentCaptured`。生产者不关心谁在监听，也不期待任何回应。事件以过去时命名，因为它们报告的是已经发生的事；之后的事件可以抵消它，却无法使它不曾发生。消费者可以忽略事件，这是一种合法的回应。

命令请求某一个特定的服务去做某件事：`CapturePayment`。它有单一的预期接收方，并期待接收方采取行动。命令可以被拒绝，发送方通常关心结果。命令是祈使式的，携带 `command_id` 以取代 `event_id`，并且绝不称为事件。由编排驱动的 saga（第 10 章讲述）就是用这类命令来驱动其参与者的。

文档消息携带一块数据，却不说明发生了什么，也不说明谁应当采取行动，例如卖家每晚的商品目录快照。接收方自行决定它意味着什么。文档消息是合法的，但它是最常被伪装成事件的一种，这种伪装会造成混乱。

类型	表达	接收方	命名	接收方能否拒绝？
事件	某个事实发生了	任何订阅者	过去时：`OrderPlaced`	能，通过忽略它
命令	做这件事	一个指定的服务	祈使式：`CapturePayment`	能，通过回复失败

类型	表达	接收方	命名	接收方能否拒绝?
文档	这里有一些数据	读取它的任何人	名词：`CatalogSnapshot`	不适用

常见的失败是披着事件外衣的命令。假设 Orders 发布 `SendConfirmationEmail`，一个祈使式的名称，却因为它经由 topic 传输而称之为事件。Orders 现在依赖 Notifications 以某种方式行事，却没有办法听到拒绝。如果 Notifications 宕机，Orders 并不知道。如果第二个服务订阅了它，面对的就是一条并非发给自己的指令。

我会采用一个快速检验。问一问：即使没有人消费这条消息，生产者是否仍然正确。对于 `OrderPlaced`，答案是肯定的：订单无论如何都已经下了。对于 `SendConfirmationEmail`，答案是否定的，因为这条消息只有在有人采取行动时才有意义。后一种应当放在接收方有名有姓、失败会回到发送方的通道上。

这在一定程度上也是方向的问题。事件从事实的所有者指向任何感兴趣的人。命令从协调者指向执行者。当事件名称开始听起来像是对某个特定服务的指令时，方向就反了。

命令是什么样子

这两种消息的失败方式也不同，信封必须反映这一点。事件的丢失按设计就是无声的：生产者从未期待某个特定的人采取行动，所以没有人注意到它的缺席。丢失或迟到的命令则不同。发送方在等待结果，唯一能暴露丢失的，是它自己的截止时间，这就是第 10 章中的 saga 为每个非终态都设置截止时间的原因。命令还可能在发送方已经放弃并补偿之后才到达，所以接收方需要一种拒绝它的办法。

因此，命令携带的信封比事件更小，而且不同：

```
{
    command_id:     "cmd-capture-saga-7731",       // stable across
retries of the same intent
    command_type:   "CapturePayment",
    target:         "payments",                    // one named
recipient
    reply_to:       "orders.saga-replies",         // where the
outcome goes
    expects:        ["PaymentCaptured", "PaymentCaptureFailed"],
    correlation_id: "order-81724",
    causation_id:   "e3",
    expires_at:     "2026-10-06T14:05:00Z",        // optional; judged
on the receiver's clock
    payload:        { order_id: "81724", amount_minor: 4990, currency:
"EUR" }
}
```

`command_id` 取代 `event_id`，第 10 章说明了它如何派生：对于不得重复的动作，由 saga 和步骤派生；对于可能再次发出的补偿，再加上引发它的 `event_id`。`reply_to` 和 `expects` 为结果提供了返回的路径，因此拒绝是发送方能够听到的。`expires_at` 让接收方丢弃在发送方已经不再等待之后才到达的命令。它是可选的，因为它有代价：它与接收方的时钟比较，而第 3 章已经给出了这个时钟无法精确到秒的原因。设置时应当留出余量，并把它当作对抗数小时前的命令的保护，而不是对抗数秒前的命令。

领域事件与集成事件

在服务内部，代码经常针对自己的聚合引发事件：`OrderLineAdded`、`DiscountApplied`、`AddressCorrected`。这些**领域事件**描述内部模型，模型一变它们就变。它们在服务内部很有用，例如用来更新读模型，或在同一事务中触发一条规则。

集成事件是跨越服务边界的版本。它是经过刻意挑选的，以其他服务需要知道的内容命名，并保持稳定。`OrderPlaced` 是一个集成事件。它不是订单聚合在那一刻的样子的转储。

诱惑在于把领域事件直接发布到 broker，因为现成的机制已经存在。在我看来，这是泄漏内部结构最常见的单一途径。设想一下，Orders 团队把一个聚合拆成两个，或合并两种订单行类型时会发生什么。如果领域事件是公共契约，每个消费者都会出问题，尽管业务上没有任何变化。

修复办法是在生产服务中加入一个转换步骤。领域事件仍然可以自由变化。一个小小的映射函数决定其中哪些成为集成事件，叫什么名字，带哪些字段。这个函数就是决定公共契约是什么的地方，而第 4 章的 transactional outbox 是调用它的天然位置：写入 outbox 行的，就是集成事件。

```
function toIntegrationEvent(domainEvent, context):
    // the mapping is the public contract; internals stay behind it
    if domainEvent.kind == "OrderSubmitted":
        return {
            event_id:       newEventId(),
            event_type:     "OrderPlaced",
            schema_version: 1,
            occurred_at:    domainEvent.occurred_at,
            producer:       "orders",
            correlation_id: context.correlation_id,
            causation_id:   context.causation_id,
            partition_key:  domainEvent.order_id,
            payload: {
                order_id:   domainEvent.order_id,
                buyer_id:   domainEvent.buyer_id,
                line_count: domainEvent.lines.length,
                total_minor: domainEvent.total_minor,
                currency:   domainEvent.currency
            }
        }
    // most domain events never leave the service
    return nothing
```

请注意，内部名称 `OrderSubmitted` 与公共名称 `OrderPlaced` 不同。这是允许的，也是健康的。公共名称是其他团队要学习的。内部名称可以跟着代码走。

瘦事件与胖事件

payload 应当携带多少？两个极端值得命名。

瘦事件，有时称为通知，只携带一个标识符，外加很少的内容：“订单 81724 已下单”。需要详情的消费者要回头调用生产者。**胖事件**携带足够的状态，让消费者无需回调即可行动。这种风格常被称为事件携带状态传输，因为事件把一份状态从一个服务搬到另一个服务。

两边各有一种失败模式，前面几章有助于解释。

瘦事件带回了事件本应消除的同步耦合。只有在那一刻 Orders 可以访问时，Shipping 才能继续，这正是开头那个周一上午的问题。它们还会造成竞争：等消费者回调时，数据可能已经变了，消费者读到的是比触发事件的那个状态更新的状态。如果回调失败，消费者就落入第 6 章的世界，对一个可能长时间缓慢的依赖执行 retry。

胖事件的代价在别的地方。payload 增大，承诺要保持稳定的字段数量也随之增加。消费者开始把事件当作生产者数据的副本，而副本会老化。周一从 `OrderConfirmed` 缓存了地址的消费者，可能在周五、买家更正地址之后仍在使用它，除非它同时读取后来的事件。而且，payload 中的所有内容都会被复制到每一个消费者和每一个保留的日志段中，这抬高了敏感字段的风险。

	瘦	胖
payload	标识符和事件类型	标识符加上消费者所需的状态
消费者需要生产者在线	需要，为了获取详情	不需要
过期风险	读取时低，但会与后来的变更竞争	消费者持有会老化的快照
契约大小	小	大
隐私暴露	低	较高
重放	需要生产者当前的数据，它可能与当时不同	自包含，重现当时已知的内容

我倾向于走中间路线：**携带消费者做决定和行动所需的内容，而不是它可能想要的内容**。对于 `OrderConfirmed`，Shipping 需要订单 ID、必须打包的订单行和送货地址，所以这些属于事件。它不需要买家的付款方式、卖家的付款详情，或得出总额的折扣规则。找到边界的一个有用办法，是问每个消费团队会用每个字段做什么。没有指名的消费者、也没有说明用途的字段，是可以删除的候选。

第二个细节也很重要。如果事件携带状态，就应当在 payload 中说明这份状态属于哪个时刻。使用聚合版本（第 4 章），第 9 章把 Parcelly 订单的该版本称为 `order_version`，消费者就可以判断它持有的快照比第二个事件更新还是更旧；`occurred_at` 只是一个粗略的提示，原因见下文。第 9 章介绍了如何用这类数字检测乱序到达，所以本章只指出：没有版本的胖事件会招来那里所述的后写入者胜出陷阱。不要把这个数字

与信封中的 `schema_version` 混淆：聚合版本统计对一个订单的变更次数，每次变更都会推进，而 `schema_version` 指的是 payload 形态的破坏性版本，很少推进（第 12 章）。

这个选择不必是全局的。团队可以对很少使用的事件从瘦开始，在真有消费者提出要求时再增加字段。从瘦开始再增长，比从胖开始再缩减更便宜，因为缩减意味着破坏性变更，第 12 章会详细讨论。

事件命名

本书中自始至终使用的 Parcelly 事件名称，遵循一小组规则，本节连同理由一并说明。

- **过去时**，因为事件是事实：`PaymentCaptured`，而不是 `CapturePayment`，后者是命令。
- **主语在前，PascalCase**：先是聚合，然后是发生了什么。`ShipmentDispatched`，而不是 `DispatchedShipment`。按字母顺序排序时，关于同一个聚合的事件会在目录和仪表盘中聚在一起。
- **使用业务语言，而不是技术语言**。`OrderConfirmed` 说的是产品负责人能够理解的事。`OrderRowUpdated` 说的是一张表的事。
- **名称中不含技术，也不含消费者**。`OrderPlacedKafkaEvent` 泄漏了 broker。`OrderPlacedForShipping` 泄漏了一个消费者，下一个消费者将不得不怀疑这个事件是不是给自己的。
- **除非两个服务发出关于同一个名词的事实，否则不加服务前缀**。假设 Orders 和 Shipping 都可以报告 `DeliveryDelayed`：一个表示卖家尚未交出包裹，另一个表示承运商延误。消费者无法通过名称区分二者，所以有必要加以限定（`OrderDeliveryDelayed`、`ShipmentDeliveryDelayed`）。这个例子仅作说明；本书中的 Parcelly 服务并不会同时发出两者。除此之外，生产者已经出现在 `producer` 字段中。

最重要的命名规则是关于含义的那一条。`OrderUpdated` 是一个通不过事实检验的名称：它只说有东西变了，别的什么也没说。消费者必须把新状态与旧状态比较，才能知道发生了什么，而且分不清是地址变了还是总额变了。应当选用陈述业务事实的名称：`OrderCancelled`、`ShipmentAddressCorrected`。当找不到业务名称时，这往往是一个信号：该事件描述的是数据库变更，而不是业务变更。

另一条规则是稳定性。一旦消费者依赖了某个事件名称，重命名它就是一次破坏性变更，成本与删除字段相同。在首次发布之前，多花 10 分钟在名称上。

信封

本书中的每个事件都携带相同的信封：`event_id`、`event_type`、`schema_version`、`occurred_at`、`producer`、`correlation_id`、`causation_id`、`partition_key` 和 `payload`。信封是消息中工具可以依赖、而无需理解业务的部分。relay、死信处理器、tracing 系统和重放工具都可以在不解析 payload 的情况下使用它。

每个字段都回答某个组件会提出的一个问题：

字段	它回答的问题	谁在问
`event_id`	我以前见过这个事件吗?	inbox、去重（第 7 章）
`event_type`	如何解释 payload?	路由、反序列化
`schema_version`	payload 的形态是哪个破坏性版本?	消费者、schema registry（第 12 章）
`occurred_at`	就生产者所见，事实是何时发生的?	对账、分析、调试
`producer`	哪个服务拥有这个事实?	责任归属、支持、问题的路由
`correlation_id`	这属于哪个业务流程?	支持、对账、日志搜索
`causation_id`	直接是什么引起了它?	重建因果链
`partition_key`	这个事件涉及哪个聚合?	broker、消息顺序（第 9 章）
`payload`	详情是什么?	业务逻辑

有两点需要留意。`occurred_at` 是生产者的时钟，第 3 章解释过为什么不应当按它给事件排序。它大致说明事实何时发生，而不能充当序列。`partition_key` 是聚合标识符，如何选择它是第 9 章的主题。

`correlation_id` 不是 trace ID。trace 上下文（W3C Trace Context）在信封旁边的 header 中传递，第 13 章展示了如何把它捕获到 outbox 中。trace 会被采样，也会过期；`correlation_id` 是两者都能留存下来的业务键，也是支持工程师用来搜索的那一个。

跨语言的信封

Parcelly 的服务并非都用同一种语言编写。Orders 和 Payments 运行在 JVM 上，Notifications 是 Node，Ledger 团队的对账作业是 Python。上面的字段列表说明了信封包含什么。它没有说明每个字段在传输中如何书写，而这正是第二种语言破坏一份在第一种语言里看似没问题的契约的地方。

字段	传输类型	规则
`event_id`、`correlation_id`、`causation_id`、`partition_key`	string	标识符是字符串，即使它们看起来像数字。作为 JSON 数字的 64 位 ID，会被 JavaScript 消费者悄悄舍入
`event_type`	string	稳定的、与语言无关的名称，例如 `PaymentCaptured`，绝不是类名或包名，也绝不是框架默认填入的类型 header
`schema_version`	integer	数值小，在每种语言里都是整数
`occurred_at`	string	UTC 的 RFC 3339，带 `Z` 后缀，固定为毫秒精度，无偏移量。其他一切，解析器各执一词
`payload` 中的金额	integer 或 string	在文档化的范围内使用整数最小货币单位（`total_minor`），或使用十进制字符串。绝不使用浮点数。JavaScript 数字只在 2^53 − 1 以内是精确的，所以可能超出该范围的字段以字符串传输
类枚举字段	string	把取值集合视为开放的：消费者必须处理一个它没见过的值（第 12 章）

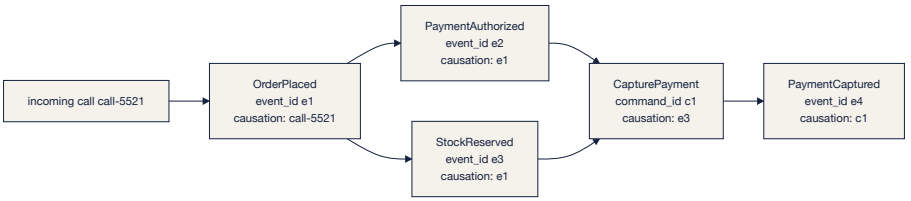
另有两条放置规则完善这份契约。信封存在于消息体中，broker 的 header 只镜像消息体已经说明的内容，因为消息被复制、转入死信队列或桥接到另一个 broker 时，header 是被丢掉的部分。而内容类型在 schema 中声明一次，不是逐条消息去猜。消费者不认识的字段会怎样处理，以及不同格式在解码这些字段时有何差异，是兼容性问题，属于第 12 章。

Correlation 与 causation

最常让读者困惑的两个标识符是 `correlation_id` 和 `causation_id`。它们回答的是不同的问题。

`correlation_id` 由一个业务流程中的每个事件共享。当买家下了订单 81724，随后的一切都携带 `correlation_id: "order-81724"`：预授权、预留、扣款、货运。买家来信询问发生了什么时，就用它来搜索。

`causation_id` 指向直接引起当前这条消息的那一条消息。它是事件的 `event_id`，或命令的 `command_id`。它不会在整个流程中重复。每一跳都把它设为其直接触发者的标识符，沿着链条向后追溯，就能重建路径。



请看最后两步。`CapturePayment` 由两个事件触发，Orders 必须为 `causation_id` 选择其一，这里选的是使条件成立的那个。这种歧义是真实存在的，我会把规则写下来，让所有生产者以同样的方式选择。`PaymentCaptured` 事件指向的是命令，而不是它之前的那个事件，这就是信封允许该字段使用两种标识符的原因。

流程中的每个事件都共享 `correlation_id: "order-81724"`。流程中的第一个事件设定关联标识符，方式是从聚合派生，或从传入的请求中带入。后面的事件原样复制它。

本书自始至终保留以聚合为范围的 `order-81724`，这是一个有后果的选择。以聚合为范围的 ID 很简单，但该订单上的每个流程都合并在它之下：三周后的一次退款与最初的结账共用同一个 ID，二者无法区分。另一种做法是以**流程为范围**的 ID，为每个流程新生成一个（`checkout-81724-1`、`refund-81724-1`），并在 payload 中保留可搜索的 `order_id`。它多出一样需要生成和传播的东西，当一个聚合可能发起第二个必须与第一个区分开的流程时，这个代价是值得的。本章其余部分在两种情况下都成立。复制就是全部工作，忘记复制它的消费者，会让下游的一切线索中断。

```
function buildFollowUpEvent(trigger, eventType, payload):
    // trigger is the message being handled: an event or a command
    return {
        event_id:       newEventId(),
        event_type:     eventType,
        schema_version: currentVersionOf(eventType),
        occurred_at:    now(),
        producer:       thisServiceName,
        correlation_id: trigger.correlation_id,  // copied, never
regenerated
        causation_id:   identifierOf(trigger),   // event_id or
command_id
        partition_key:  aggregateIdOf(payload),  // this event's
aggregate, not the trigger's
        payload:        payload
    }
```

只有当两条消息涉及同一个聚合时，才可以复制触发者的 `partition_key`。如果关于付款或货运的后续消息继承了订单的键，它就会按错误的 stream 排序（第 9 章）。Parcelly 几乎所有东西都以订单为键，这正是这个错误难以发现的原因。

出错的代价在最糟糕的时候支付。事故期间，问题是“这个订单在五个服务中发生了什么？”。没有一致的 `correlation_id`，工程师就得手工搜索五份日志。第 13 章展示 tracing 和对账如何使用这些字段，所以本章的工作，是确保它们从第一个事件起就存在且正确。

事件中不应包含什么

事件离开生产者的数据库，进入一个访问规则不同、保留期限不同、读者众多的系统。所以设计问题也是这样一个问题：愿意把什么复制进那个系统。

个人数据。带有邮箱地址、电话号码或完整送货地址的事件，会存放在 broker 存放数据的任何地方，可能在整个保留窗口期内都在，并被每个消费者读取。在事件日志中，事后删除某个人的数据很麻烦，因为日志是 append-only 的，而且消费者已经复制了这些字段。有些团队以加密方案或单独的 topic 来应对。我会更早着手：在事件中放入标识符，例如 `buyer_id`，让有正当需要的消费者在那一刻向拥有数据的服务获取个人字段，并遵守该服务的访问规则。这倾向于对敏感字段使用瘦事件，我认为这个取舍是值得的。如果胖事件确实需要地址，比如 Shipping 那样，那么事件就应当有意地、面向那一类受众携带它，并且在知道它存在的前提下选择保留期限。

哪些字段算个人数据、随之产生哪些义务，取决于所在的司法辖区和法律顾问。我不了解各位的约束条件。无论在哪里，设计习惯都是相同的：事件中的每个敏感字段都应当有一个明确的理由。

机密与凭据。token、密钥和付款账号不属于 payload。消息会被记录到日志、复制到死信目的地、显示在管理工具中，并附在工单上。应当假定每个 payload 都会被人读到。

内部细节。表名、内部状态码、功能开关、只在某一个服务内部才有意义的数据库 ID 以及含义取决于生产者代码的值。开头场景中的 `internal_status` 就是例子。这样的字段，一旦有人读取，就是一个公开的承诺。

对特定消费者的指令。像 `skip_email_if_gift` 这样的标志，是在告诉 Notifications 该怎么做。事件应当陈述事实，由消费者决定如何处理。

> **警告**：凡是进入事件日志的内容，都应当视为永久的、可被广泛读取的。写入一次的机密或个人字段，会被复制到死信目的地、备份和重放工具中，要在所有地方把它删除，远比一开始就不放进去困难得多。

审查新事件时，一份简短的清单会有帮助：每个字段是否有需要它的消费者，每个字段是否是生产者愿意保持稳定的内容，以及这个字段若被不该看到它的人读到，是否会伤害到谁。

粒度

一个事件有多大？有两种相反的错误。

太细。为每次列更新都发布一个 `FieldChanged` 事件，会产生一股噪声。消费者必须从许多消息中重新拼出含义，这些消息的顺序也开始以第 9 章警告过的方式变得重要。原本是一个业务事实的变更，即买家更正地址，现在以四个事件的形式到来，可能被看到处于半应用状态。

太粗。一个在每次修改时触发、并携带整个订单的 `OrderChanged` 事件，迫使每个消费者把新状态与旧状态做 diff，并去猜业务含义。它还意味着每个消费者在每次变更时都被唤醒，包括它并不关心的那些。

我会以业务事实为单位：每个业务中有人会说出名字的决定或事件，对应一个事件。买家下了订单是一个事实。付款被扣是另一个事实。当一个事务产生多个事实时，应当通过同一次 transactional outbox 写入发布多

个事件，使它们一起提交或回滚。

一个实用的检验是问一问：消费者能否仅凭这一个事件做出有意义的反应。如果答案是“要等接下来的三个事件到达之后”，粒度就太细了。如果大多数消费者忽略 payload 的大部分内容，或者事件名称需要用“和”来描述，它就可能太粗，并且可能藏着两个事实。

责任归属与公共契约

每个事件都有且仅有一个所有者：拥有该事件所描述的聚合的服务。Payments 拥有 `PaymentCaptured`。Orders 拥有 `OrderPlaced`。消费者可以读取事件，但只有所有者定义它。想要不同形态的消费者去向所有者提出，而不是在同一个名称下自行发布一个变体。

把事件视为**公共契约**会带来实际的后果，我会把它们写进团队的工作约定。

- **所有者记录事件文档**：它的名称、含义、何时发出、何时不发出、每个字段的类型和含义以及关于顺序和分区键的保证。
- **事件有版本和变更流程**。第 12 章描述了如何在不破坏消费者的情况下演进 schema。基本规则属于这里：所有者在不了解对读取方的后果之前，不会改动已发布的事件。
- **消费者自我声明**。一份简短的登记册，记录谁消费什么，把一次无声的订阅变成已知的依赖。它不需要花哨。仓库中的一份清单，要求 pull request 对其进行更新，就足以开始。
- **消费者只读取自己需要的内容**。第 12 章的 tolerant reader 规则从这里开始：忽略不使用的字段，这样新增式的变更就不会出问题。
- **发出事件是行为的一部分**。某个服务在某条代码路径上不再发出 `OrderCancelled`，就像删除一个 HTTP endpoint 一样，改变了它的公共接口。

“契约”和“所有者”这两个词都有分量。契约可以被测试。所有者可以被呼叫。如果没有人能说出谁拥有某个事件，就没有人能说出谁有权修改它，开头的场景就会重演。

权衡与成本

规范会消耗速度。把领域事件映射为集成事件、仔细命名、编写文档、维护消费者清单，都会拖慢首次发布。回报在以后到来，表现为更便宜的变更。对于只有一个生产者和一个消费者、且由同一个人编写的原型，这套仪式可能不划算，我会让它保持轻量，直到出现第二个团队。

胖事件用对时间的耦合换取对形态的耦合。消费者不再需要生产者可访问，却依赖它复制出来的字段。更大的 payload 意味着更大的契约，需要保持稳定和保护的东西也更多。

瘦事件发布起来便宜，消费起来昂贵。每个消费者都要付出一次回调的代价，而生产者必须承担这些读取负载，它们通常与事件同时达到峰值。如果生产者的 API 很慢，瘦事件就把问题从生产者转移到了每一个消费者身上。

细致的信封消耗字节和注意力。每条消息 9 个字段，在高流量下会累积起来，而且每个生产者都必须正确填写。另一种做法是在事故期间重建缺失的信息，那样成本更高。这有多重要，取决于消息量。通常的做法是让信封保持小巧，payload 保持精简。

传输层面的规则消耗灵活性。ID 和金额用字符串，固定的时间戳格式，以及信封位于消息体中，使多语言系统在好的意义上变得乏味。它们也意味着要在边缘处转换，而想在自己的语言中使用原生类型的团队，要为每条消息支付这笔转换成本。

命令的过期带来对时钟的依赖。`expires_at` 防范陈旧的命令，但以接收方的时钟来判断。所加的余量，就是一条真正迟到的命令仍会被接受的时间。

匿名化的或仅含标识符的事件限制了消费者单独能做的事。必须联系买家的消费者，需要调用所有者，或持有一份本地数据副本，并且有理由持有它。这是摩擦，是有意选择的摩擦。

何时不必给事件加上这么多结构。纯粹在一个服务内部、由同一个代码库消费且从不保留的内部事件，不需要信封、版本或所有者清单。完整的处理应当留给跨越边界的事件，尤其是活得比一次部署更久的事件。

常见错误

- **直接发布领域事件**。这很诱人，因为机制现成，结果是内部重构变成了破坏性变更。加入一个到集成事件的显式映射，并把该映射视为契

约。

- **把命令称为事件**。过去时被去掉或只是装点门面地加上，生产者最终依赖于某个特定的接收方，却没有办法听到拒绝。把命令发给有名有姓的接收方，并期待结果回来。
- **`Updated` 事件**。`OrderUpdated` 没有说明什么变了，所以每个消费者都去 diff 状态并猜测。为业务事实命名，如果不存在这样的名称，就质疑是否需要这个事件。
- **把整个聚合倒进 payload**。这省去了一次沟通，却造出一份与数据库表一样大的公共契约。问每个消费者它用每个字段做什么，并砍掉没人使用的。
- **因为手边有就放进个人数据和内部字段**。应当在评审时用清单来解决，因为发布之后，数据已经在无法控制的存储中了。
- **按语言默认值编码**。类名作为 `event_type`，ID 用数字，金额用浮点数，时间戳带着运行时选的任何偏移量。它一直有效，直到第二种语言来读取。把传输规则一次性写下来，就像上面的表格那样。
- **复制触发者的 `partition_key`**。当两条消息涉及同一个聚合时它是对的，对涉及另一个聚合的后续消息则是错的。应当从事件自己的聚合派生键。
- **在每一跳重新生成 `correlation_id`**。有人写了一个默认创建新标识符的辅助函数。应当从触发者那里复制它，并测试每个消费者都这样做。
- **把 `occurred_at` 当作顺序**。时间戳是一次时钟读数，而时钟之间并不一致。应当像第 9 章所述那样，使用分区键和版本号。
- **没有所有者**。由“平台”发出的事件，或两个服务在同一名称下发出的事件，没有人能批准变更。为每个事件类型指定一个所有服务。

要点

- 事件陈述事实，命令向指定的接收方请求一个动作，文档携带数据；把这三者混用，会导致隐藏的耦合和丢失的拒绝；命令需要回复路径和可选的过期时间，因为丢失的命令只会被发送方的截止时间察觉。
- 领域事件描述服务的内部模型，集成事件则是经过刻意挑选的公共契约，二者之间的显式映射，使重构不至于变成破坏性变更。
- 瘦事件使契约保持小巧，却重新引入运行时耦合和回调竞争；胖事件消除了回调，却扩大了契约；所以应当携带消费者行动所需的内容，而不是它们可能想要的内容。

- 以过去时命名事件，主语在前，使用业务语言，并避免会掩盖事实的 `Updated` 式名称。
- 信封的存在，是为了让工具无需读取 payload 就能去重、路由、追踪和排序；它的传输编码（字符串 ID、RFC 3339 UTC 时间戳、安全的金额、与语言无关的类型名称）使它在每种语言中都保持可读，而 `correlation_id` 贯穿整个流程被复制，`causation_id` 则指向那一个直接触发者。
- 个人数据、机密、内部状态和针对特定消费者的指令不应出现在 payload 中，因为进入事件日志的一切都会被广泛复制，并且难以删除。
- 每个事件有一个所有服务、一份有文档的含义和一份已知消费者的清单，这使一条消息变成可以有意变更的契约。

设计良好的事件仍然必须变化，因为业务会变，消费者会以不同的速度升级，而旧事件还留在日志里。第 12 章转向 schema 演进：如何变更其他团队已经依赖的契约而不破坏他们。

第 12 章 — Schema 演进

设想 Parcelly 的 Payments 团队想要支持部分扣款。卖家发出了一个含两件商品的订单中的一半，买家应当现在为这一半付款，其余部分稍后再付。`PaymentCaptured` 事件只有一个字段 `amount_minor`，迄今为止它始终表示“整个订单”。团队把它改名为 `captured_minor`，并新增 `remaining_minor`，这样更清晰，也更能表达字段的含义。Payments 仓库中的测试全部通过，部署在周四很晚的时候上线。

Ledger 不在那个仓库里。它读取 `amount_minor` 来记一笔复式记账记录，什么也没读到，于是把每条新事件都当作格式错误拒绝。Notifications 读取同一个字段来生成收据，打印出一个空的金额。一个重放上月事件的报表作业，如今在同一个 topic 里遇到两批数据，一批用旧字段名，一批用新字段名，而它的解析器只能处理其中一种。

这里没有任何一处是 Payments 的 bug。代码做的正是团队要求它做的事。故障的根源在于：事件会被作者从未见过的程序读取，其中一些程序读取的还是几个月前写下的事件。读者不受自己控制，旧的写入又仍然躺在日志里，该如何修改这份契约？

问题及其难点

事件 schema 是生产者与所有消费者（包括现在的和将来的）之间的契约。自己代码库里的函数签名也是一种契约，但编译器会检查两侧，而且两侧可以在一次提交中一起修改。事件 schema 的两侧之间没有编译器。生产者和消费者在不同的日子部署，归不同的团队负责，而 broker 对到达的任何字节都照单全收。

有三个特性使它比普通的 API 版本管理更难。

第一，**事件的寿命比写出它们的代码更长**。在基于日志的 broker 上，事件会保留到配置的保留期限，有些系统甚至无限期保留。在基于队列的 broker 上，这个窗口较短，但积压或死信队列（DLQ）可能把消息留存数天。今天部署的消费者，可能需要读取一年前按旧 schema 写下的内容。

第二，**重放使过去成为实时输入**。重放（第 2 章已定义）是重建投影、为新服务 backfill 或在 bug 之后恢复的手段。它会把所有历史 schema 一次性交给今天的代码。只处理最新形态的消费者，在有人执行重放之前一直运转正常，到了那一天，它会先在最旧的事件上失败。形态只是问题的一半：重放还会把今天的业务规则和副作用施加到旧的事实上，例如再次发送收据，或再次调用支付服务商。通过重放重建的消费者必须知道自己正在重放，方式是显式的模式或 offset 范围，必须抑制其外部效果，并且持有对历史事实同样成立的规则。去重方面的内容见第 7 章。

第三，**无法让所有人同时升级**。滚动部署意味着同一服务的新旧版本会并行运行数分钟甚至数小时。一项在全部升级完成后才正确的变更，在升级期间仍然是一次事故。

显而易见的做法是把 schema 当作内部细节，事后再去修复消费者。当只有一个消费者、一个团队时，这种做法行得通。当生产者不知道有多少消费者存在时，它就行不通了，而这恰恰是事件驱动设计要创造的局面。第 11 章论证过，已发布的事件是一份公开契约。Schema 演进就是随着时间推移守住这一承诺的方法。

兼容性的定义

这些词在许多场合被宽松地使用，因此这里给出本书采用的定义。

- **向后兼容**：使用新 schema 的消费者可以读取用旧 schema 写入的事件。新读者，旧数据。
- **向前兼容**：使用旧 schema 的消费者可以读取用新 schema 写入的事件。旧读者，新数据。
- **完全兼容**：两者同时成立。
- 每一种的**传递性**变体，会把新 schema 与此前的每个版本对照检查，而不只是与最新版本对照。

方向很容易在脑子里弄反，所以我改问一个具体的问题：“如果部署这项变更，谁会出问题？”

兼容性	谁读什么	安全的部署顺序	防范的对象
向后	新读者，旧数据	先消费者，后生产者	历史重放、滞后的积压
向前	旧读者，新数据	先生产者，后消费者	无法按需升级的消费者

兼容性	谁读什么	安全的部署顺序	防范的对象
完全	双向	任意顺序	新旧版本混合运行的实例、未知的消费者
无	两者都不保证	协调进行，附迁移计划	无；见迁移一节

关于名称，还有一点提醒。registry 和序列化库使用这些词时略有差异。依赖某项设置之前，应先阅读所用工具中的定义，并确认 registry 默认采用哪一种变体：非传递性规则对可重放的日志并不够。需要用每个版本的真实样例事件做测试。

对于带重放的日志，需要的特性是针对全部历史的向后兼容，因为重放中的消费者可能遇到曾经发布过的任何版本。对于保留期限很短的队列，与上一个版本对照检查或许就够了。这既是 schema 决策，也是保留期限决策。

安全变更与破坏性变更

当旧消费者可以忽略某项变更、新消费者又能应对它的缺失时，这项变更是**增量式**的。当某个读者（无论新旧）再也无法解释某些事件时，这项变更是**破坏性**的。下面的判定假定读者按 tolerant reader 一节所述的方式行事。

对 `PaymentCaptured` 的变更	通常的判定	原因
添加可选字段（`payment_method`）	安全	旧读者忽略它；新读者能处理它的缺失
添加没有默认值的必填字段	破坏性	新读者无法解释旧事件
删除可选字段	仅当无人读取时安全	依赖它的读者什么也看不到
删除必填字段	破坏性	旧读者期望它存在
重命名字段	破坏性	相当于一次删除加一次添加；少数格式可以用别名解决重命名
更改类型（`int` 改为 `string`）	破坏性	解析器失败，或悄悄地做强制转换
拓宽数值类型	取决于格式	有些格式会提升类型，有些不会
更改字段的含义	破坏性，而且是最坏的一种	没有任何东西失败；结果却是错的

对 `PaymentCaptured` 的变更	通常的判定	原因
向枚举添加一个值	有风险	旧读者可能拒绝它或错误处理它

其中两行值得细看。

更改含义是危险的变更

重命名字段，故障会很响亮。Ledger 在第一条事件上就失败，告警触发，有人去读堆栈跟踪。更改字段的含义而保持名称和类型不变，故障则悄无声息。设想 Payments 决定让 `amount_minor` 从此表示“本事件中扣款的金额”，而不再是“订单总金额”。每个解析器仍然成功。Ledger 把部分金额当作总额记账，错误数周之后才在对账（第 13 章）中浮现，表现为一个无人能解释的缺口。

没有任何 schema 工具能发现这一点，因为 schema 没有变。只有评审这项变更的人能发现。我的规则是：如果含义变了，就应当使用新字段或新的事件类型，绝不要编辑旧的那个。

枚举是一份隐藏的契约

添加枚举值，在 schema 中是增量式的，而在消费者中往往不是。如果某个消费者把 `payment_method` 映射到处理分支，并带有一个假定为“card”的 `else` 分支，那么它会在没有任何错误的情况下错误处理新的 `"bank_transfer"`。生产者可以声明枚举是开放的。消费者可以把未知值当作未知值，把它们路由到可见的错误或安全的默认处理，而不是把它们并入某个已知的情形。

Schema registry 与格式

schema registry 是一种服务，它存储系统使用的各个 schema，为其分配标识符，并能把拟议的 schema 与此前的 schema 做对照检查。它承担着两项值得区分的职责。

1. **查找**。消息携带写入时所用 schema 的标识符，或指向该标识符。消费者取回那个 schema 并解码字节。这使紧凑的二进制格式得以实用：字段名不必随每条消息传输。
2. **把关**。生产者注册新 schema 之前，registry 会按选定的兼容性规则把它与此前的版本比较，违反规则就予以拒绝。

第二项职责是我最愿意坚持的。存在于评审检查清单里的规则，会在团队忙碌的那天被跳过。部署流水线里的检查则不会忙碌。

Subject、topic 与事件类型

registry 按 **subject** 检查兼容性，而 subject 对应的是 topic 还是事件类型，决定了多个事件类型能否共用一个 topic。跨事件类型保持每个聚合内的消息顺序（第 9 章），需要共用：同一笔支付的 `PaymentAuthorized` 和 `PaymentCaptured`，只有走同一个 topic 的同一个分区，才能保持顺序。Parcelly 的示例假定每种聚合类型对应一个 topic，例如 `payments.payment-events`，它携带按聚合 id 分区的多种事件类型，兼容性按事件类型而不是按 topic 配置。代价是消费者要按 `event_type` 过滤，而且版本化 topic（策略 3）会一次性迁移该聚合的所有事件类型。每个事件类型一个 topic 配置起来更简单，但放弃了跨类型的顺序。

三类格式

本书不绑定具体格式，所以这里只是帮助建立整体印象，而不是产品对比。Avro、Protobuf 和 JSON Schema 的区别在于数据在传输中的样子，这决定了哪些变更是安全的。

类别	传输形态	字段的标识方式	实际后果	陷阱
Avro 风格	紧凑二进制，无字段名	名称，在写入方与读取方 schema 之间解析	读者需要写入方的 schema；默认值承载兼容性	别名使读者可以解析被重命名的字段，所以重命名并不总是破坏性的。缺失的字段会被填入读者的默认值，所以缺失是不可见的
Protobuf 风格	紧凑二进制	字段编号	重命名成本低；复用编号很危险	只有在二进制传输上才成本低：ProtoJSON 和生成的代码使用名称，所以重命名会破坏源代码和 JSON。没有显式存在性的 proto3 标量会把缺失字段解码为 0
JSON Schema	文本 JSON	名称；运行时 schema 可选	易于调试；除非强制执行，兼容性只是一种约定	在封闭模型（`additionalProperties: false`）下，添加字段会破坏旧读者，而且各 registry 对开放模型和封闭模型的判断不同

有三条思路适用于所有这些格式。

- **默认值使添加变得安全，前提是默认值是诚实的**。带有明确默认值的新字段，能让新读者处理旧事件。没有默认值，添加就变成了破坏。对于读者必须能检测出缺失的字段，例如金额，默认值必须是“无值”：用 Avro 中 `null` 与该类型组成的联合类型并以 `null` 为默认值，或使用 proto3 的 `optional` 或包装类型。默认值为 0 会把旧事件解码成一次金额为零的扣款，读者无法区分两者。此时读者应当检查存在性，而不是检查值。
- **标识不应复用**。无论字段按名称还是按编号标识，给旧标识赋予新含义都是破坏。应当停用标识符，而不是回收它们。
- **应当容忍未知字段**。解码器对不认识的字段是保留、忽略还是拒绝，取决于格式及其设置。围绕它做设计之前，应先弄清楚。

如果运行的是没有 schema 语言的 JSON，仍然可以获得大部分收益：把 schema 文件放在版本控制中，在生产者的测试里做校验，并为每个版本保留真实的样例事件。registry 增加的是集中的查找与强制执行。它不会替代思考。

版本放在哪里

标准信封把 `schema_version` 作为整数放在 `event_type` 旁边，消费者读取这一对值，就知道该期望哪种形态。把它放在信封里，意味着路由器、指标计数器或 DLQ 工具无需解码 payload 就能看到它。“还有多少 `schema_version: 1` 的事件在持续到达？”于是成为一次廉价的查询，而不是一个项目。

这里有两个数字容易混淆，所以本章给它们起了不同的名字。**破坏性版本**是信封中的 `schema_version`。只有旧读者无法处理的变更才会使它变动：重命名、删除、类型变更或含义变更。**registry 修订版本**是 registry 为每个已注册 schema 分配的标识符，每次注册变更（包括增量式变更）它都会变动。消费者会拒绝其破坏性版本自己不认识的事件。对于 registry 修订版本较新、但破坏性版本已知的事件，消费者把它当作自己所知的最新形态来读取，并忽略不认识的部分。两者都不是第 4 章的聚合版本，后者统计的是对同一个订单的变更次数，与 payload 的形态无关。

如果每次增量式变更都递增 `schema_version`，每个尚未升级的消费者都会拒绝每一条新事件，向前兼容也就荡然无存。拆分的代价是需要分清两个数字。同一思路的一个例子是 BabelQueue（babelqueue.com），一个开源的、与语言无关的信封标准：它把信封冻结在 `schema_version`

1，只允许新字段以可选且可忽略的方式出现，重命名、删除或类型变更必须递增版本，要求消费者拒绝不认识的版本，并使未知字段不会妨碍处理，而且在消息再次发布时予以保留。

下面是 Payments 在下文迁移的双发布步骤中发布的事件：

```
{
  event_id:        "9c04b6e2-…",
  event_type:      "PaymentCaptured",
  schema_version:  1,
  occurred_at:     "2026-10-05T11:20:44Z",
  producer:        "payments",
  correlation_id:  "order-90412",
  causation_id:    "5ad1e07c-…",
  partition_key:   "order-90412",
  payload: {
    order_id:        "order-90412",
    amount_minor:    4990,
    captured_minor:  4990,
    currency:        "EUR"
  }
}
```

跨语言兼容性

一个 schema 按 registry 的规则可以完全兼容，却仍然在两种语言之间出问题，因为规则描述的是结构，而问题出在运行时如何读取它。第 11 章已经确定了信封的编码。下面是 payload 中的陷阱，以及消费者如何处理不认识的字段的陷阱。

未知字段在重新发出时丢失。假设 Notifications 中的一个 Node 消费者把每条 `PaymentCaptured` 反序列化为带有 `order_id`、`captured_minor` 和 `currency` 的类，并把事件复制到一个面向合作伙伴的 topic。之后 Payments 添加了 `payment_method`，这份副本就会悄悄丢掉它。没有任何东西失败。那个字段从未出现过。Protobuf 二进制解码器通常会保留未知字段；许多带类型的 JSON 映射器则不会。规则：转发消息的一跳必须保留未知字段，做法是携带原始 payload 或通用映射，或者按自己的契约发布一个属于自己的新事件。

数字。`captured_minor` 在任何语言的整数中都绰绰有余。而一个 int64 标识符或计数器，一旦超过 2^53 − 1，就放不进 JavaScript 的 JSON 数字，消费者会在没有任何错误的情况下对它做舍入。规则：记录每个数值字段的取值范围，凡是可能超出该范围的值，以及任何小数，都以字符串发送。

时间戳。Avro 的 `timestamp-millis` 和 `timestamp-micros` 保存的是同一类值，只是刻度不同。以微秒写入、却按毫秒读取，得到的日期会远在数万年之后，却仍然能解析成功。规则：在 schema 中固定单位，并且永不更改；单位变更属于破坏性版本。

枚举。生成的代码对未知值的处理各不相同：有的保留它，有的映射到默认值，有的抛出异常。规则：把每个枚举都当作开放的，并在消费者中决定未知值意味着什么。

Null 与缺失。许多解码器不区分二者，有些还会用默认值填补空缺。规则：对于缺失与否很重要的字段，使用上文默认值一条所述的、能感知存在性的形式。

这些都不需要某个特定库的特定版本。它们都需要有人用 Parcelly 所运行的每种语言的消费者去测试，这是第 15 章的契约测试可以承担的成本。

版本管理策略

引入变更有三种方式，成本由低到高。我会按顺序尝试，只有下一级承载不了这项变更时，才往上爬一级。

策略 1：添加可选字段

如果变更是旧读者可以忽略的新内容，就添加一个可选字段。给 `PaymentCaptured` 添加 `payment_method` 是典型的例子。它会注册一个新的 registry 修订版本，registry 是否接受则由兼容性规则决定。它不会改变 `schema_version`，因为没有任何旧读者失去什么。为增量式变更递增破坏性版本，会使每个尚未升级的消费者拒绝新事件，这就违背了以兼容方式添加字段的初衷。

这种策略无法表达重命名、类型变更、删除必填内容或含义变更。对于这些，需要往上一级。

策略 2：引入新的事件类型

当含义改变时，应当用一个新名称把这件事说出来。假设 Parcelly 把部分扣款视为与全额扣款不同的业务事实。诚实的建模方式是新增一个事件，按第 11 章的规则命名。本章把 `PaymentPartiallyCaptured` 作为额外的事件引入。

旧事件保持原有含义，其消费者继续正常工作。代价是两种事件现在并存，想要“所有扣款”的消费者必须同时订阅两者。当新的事实在概念上确实不同时，这是正确的取舍；当它只是同一事实换了更好的形态时，则是错误的取舍。后一种情形是表示方式的问题，更适合用 upcaster 解决。

策略 3：版本化 topic

当破坏性变更涉及整个 stream 时，把新形态放到新 topic 上，例如 `payments.payment-events.v2`。消费者按各自的节奏迁移，当最后一个读者离开时，旧 topic 下线。按上文所述的布局，新 topic 承载该聚合的所有事件类型。

这是最清晰的选项，也是最昂贵的。两个 topic 的分区键、消息顺序和 offset 互不相关，所以从一个迁到另一个的消费者必须决定从哪里开始读取，以及如何避免重复处理同一个事实。第 9 章的消息顺序规则分别适用于每个 topic，而跨 topic 的事件之间没有任何顺序可言。在基于队列的 broker 上，对应的做法是另建一个队列，带有它自己的绑定、死信处理和告警。我只会在遇到 upcaster 无法弥合的真正破坏性变更时才考虑版本化 topic，而且只在能说出由谁来删除旧 topic 时才这样做。

Upcasting 与 tolerant reader

有两种技术，能让消费者应对多个版本，而无需迫使生产者冻结。

Upcasting

upcaster 会在业务逻辑看到事件之前，把按旧 schema 写入的事件转换为当前 schema。业务代码针对单一形态编写，对每个历史版本的转换则集中在一处。

```
function upcastPaymentCaptured(event):
    p = event.payload
    if event.schema_version == 1:
        // during the migration a v1 event carries either field, or
both

        // check presence, not value
        if hasField(p, "captured_minor"):
            captured = p.captured_minor
        else:
            // v1 only ever described full captures, so the
            // order amount is the captured amount; may be absent
            captured = p.amount_minor
        return { order_id: p.order_id, captured_minor: captured,
currency: p.currency }
    else if event.schema_version == 2:
        // v2 carries captured_minor only
        return { order_id: p.order_id, captured_minor:
p.captured_minor, currency: p.currency }
    else:
        // an unknown newer breaking version must not be guessed at
        raise UnsupportedSchemaVersion
```

版本 1 分支上的注释承载着真正的工作。upcaster 编码的是一条关于过去的陈述，而这条陈述必须是真的。如果有些旧事件在实际中是部分扣款，upcaster 就会悄悄地把部分金额报告为整笔扣款。应当用真实归档的事件来测试 upcaster，而不是编造的事件。

`UnsupportedSchemaVersion` 需要独立的分类。对当前代码而言，它是永久的，因为 retry 不会改变任何结果，但部署消费者可以治好它。所以应当把它留待消费者升级之后 redrive（第 8 章），而不是送进 parking lot，在那里它会等待一个无事可决的人。

消费者一侧，也就是 handler 内、业务逻辑之前，是最简单的起点。当许多消费者否则都需要同一个 upcaster 时，把历史改写进新 topic 就有用了。就地改写已存储的事件则是另一种决策：event sourcing 的系统往往拒绝这样做，因为日志就是证据。除非有法律或安全方面的理由，否则我会尊重这一点，即便有理由，我也会把它作为一次带计划的数据迁移来执行。

Tolerant reader

tolerant reader 只从事件中取走自己需要的部分，忽略其余。它读取 `order_id` 和 `captured_minor`，并不在意还有一个 `payment_method`。它对接受的内容宽松，对依赖的内容严格。

```
function readPaymentForLedger(event):
    payload = upcastPaymentCaptured(event)

    if not hasField(payload, "order_id") or
       not hasField(payload, "captured_minor") or
       not hasField(payload, "currency"):
        // a missing dependency will fail on every attempt,
        // so it is a poison message, not a retry case
        raise InvalidEvent

    return {
        order_id: payload.order_id,
        amount:   payload.captured_minor,
        currency: payload.currency
    }
```

容忍有两面。生产者添加内容时，忽略未知字段是安全的。悄悄地用零替代缺失的金额则是危险的。上面的读者在它所依赖的字段缺失时拒绝继续，并且询问的是存在性，而不是值。只有当解码器能区分缺失的字段与零时，这才是诚实的，这也是上文的默认值如此重要的原因。这种拒绝本身就是一次分类：该事件永远不可能成功，所以它走第 6 章所说的永久失败路径，最终成为死信，如第 8 章所述。

> **注意**：容忍是双方之间的约定。生产者承诺只添加，不改变含义。消费者承诺忽略自己不理解的内容，并拒绝自己无法安全处理的内容。任何一方违背了自己的那一半，另一方都救不了这个系统。

registry 在生产者发布之前对其做检查。tolerant reader 则在有东西漏网之后保护消费者。我不会用其中一个替代另一个。

迁移破坏性变更

有时破坏无法避免：旧的形态是错的，法规要求去掉某个字段，或者两个服务必须一起重塑。办法是把一次破坏性部署变成若干次非破坏性部署。这就是应用于事件的**扩展与收缩**模式：先扩展契约以同时支持两种形态，迁移每个读者，然后收缩，也就是移除旧形态。

`PaymentCaptured` 的迁移计划

假设 Parcelly 确实必须把 `amount_minor` 重命名为 `captured_minor`，事件的其余部分都不变。该值保持其含义：整个订单的扣款金额。这是一次纯粹的重命名，下面的计划只因此才是安全的。部分扣款属于含义变

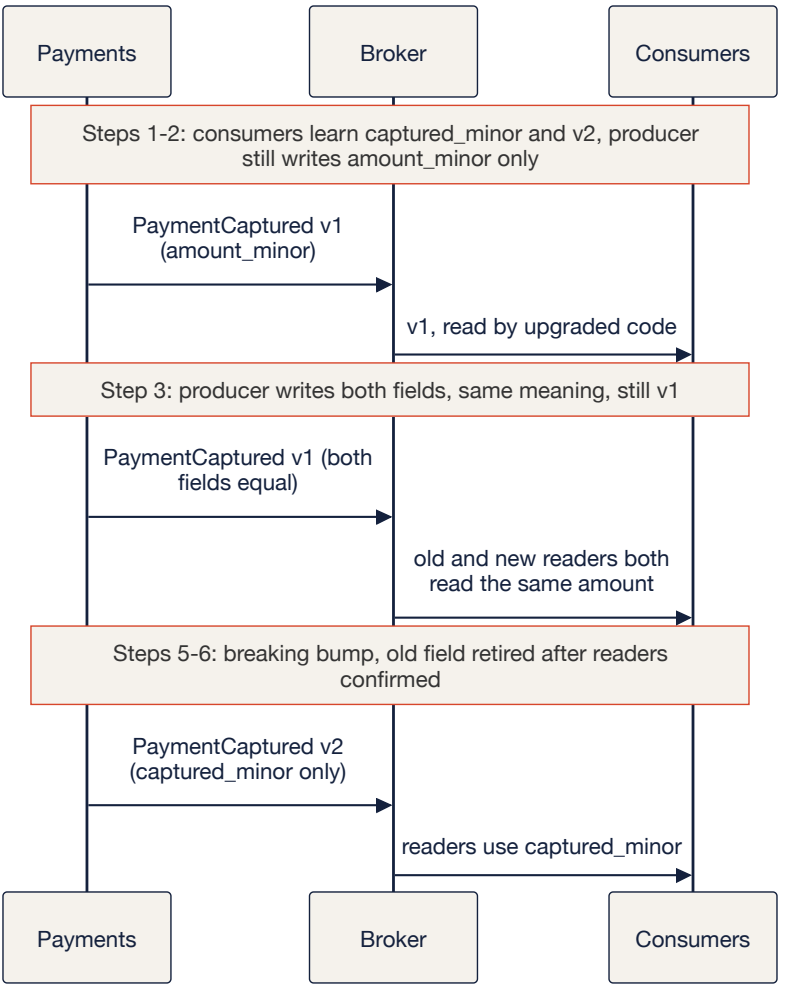
更，将在这些步骤之后单独处理。

1. **注册新增内容**。新的 registry 修订版本保留 `amount_minor`，并把 `captured_minor` 作为可选字段添加进来，其默认值是“无值”，而不是 0。在向后兼容规则下，registry 会接受它。`schema_version` 保持为 1，因为没有任何旧读者失去什么。
2. **先升级消费者**。每个消费者学会读取 `captured_minor`，并在它缺失时回退到 `amount_minor`。它还要学会读取尚不存在的破坏性版本 2，这样第 5 步的递增就不会碰到拒绝它的消费者。新读者先于新数据存在。
3. **双发布**。Payments 在每个事件里同时写入两个字段，仍然使用 `schema_version` 1；或者，如果变更需要新的事件类型或 topic，则发布同一个事实的旧形式和新形式。这不是第 4 章的双写：两条消息来自同一个 transactional outbox 事务，因此通常的保证依然成立。
4. **观察**。度量谁还在读取旧字段或旧 topic。生产者针对 `schema_version` 的指标显示仍在写入什么；旧 topic 上的 consumer group 活动、第 11 章的消费者清单以及第 15 章的契约测试显示谁还在读取。一直不迁移的消费者，是需要去谈的一场对话，而不是可以忽略的一个数字。
5. **停止写入旧形态**。Payments 只发布 `captured_minor`，并把 `schema_version` 递增到 2，因为这是旧读者无法承受的唯一一步。错过第 2 步的消费者会拒绝该事件，事件进入 DLQ，等该消费者升级后再 redrive（第 8 章）。第 4 步的存在，就是为了让这份名单保持为空。`amount_minor` 作为已弃用字段保留在 schema 中，直到携带它的最旧事件离开保留期限。
6. **收缩**。移除旧字段，如果有旧 topic，则让它下线。只有当最旧的、没有 `captured_minor` 的事件离开保留期限之后，才从消费者中移除 fallback 逻辑，或者把它移入一个永久的 upcaster。

部分扣款不在这份计划之内。一旦扣款可以是部分的，`amount_minor` 就没有诚实的值了。如果它携带已扣款金额，它的含义就被就地改变了，而跳过第 2 步的读者（无论是不知情还是被回滚）会在没有任何错误的情况下，把部分金额当作订单总额记录。如果它携带订单总额，同一个读者记录的就会偏多。registry 会放行这两个版本，因为结构没有变。这正是教训所在：结构性的兼容性检查看不到含义的变更。应当把新含义放进新

字段、新的事件类型（策略 2，`PaymentPartiallyCaptured`）或新 topic 上的新主版本。然后，在每个读者都迁移到新形式之前，完全不要发布部分扣款，此后也只以新形式发布。

顺序是这份计划的核心。对于向后兼容的变更，消费者先行；对于向前兼容的变更，生产者可以先行。当无法判断适用哪一种时，就按所写的计划执行，并把两个字段保留一段时间。



跨 topic 的双发布

当计划使用版本化 topic 时，第 3 步会把事实发布两次，每个 topic 各一次。同时读取两者的消费者，不得对同一个业务事实处理两次。我会给这两份副本不同的 `event_id`，因为它们是不同的消息，并把原始事件的

id 放进一个有文档说明的字段。无论选择哪种方式，消费者针对该事实的幂等键都必须是有意定义的。幂等键见第 7 章；这里要强调的是，迁移恰恰是幂等性假设受到检验的时刻。

> **警告**：双发布期间，绝不能让消费者同时读取旧流和新流并两者都应用。对同一事实有两个来源的消费者，需要一条写明的去重规则，否则会把一条账本记录记两次。

回滚与历史

回滚是计划的一部分。如果版本 2 的事件已经发布，把生产者回滚并不会把它们从日志中移除。回滚到无法读取版本 2 的代码的消费者，现在会在已经存在的事件上失败。因此，在任何生产者写入版本 2 之前，消费者必须能读取版本 2；消费者的回滚目标必须能读取保留期限内的每个版本，包括版本 2。只能读取版本 1 的代码是应当避免的回滚目标，即使它是变更之前的最后一个发布版本。

向后兼容在不触碰旧事件的前提下处理它们。如果想删除 fallback 代码，有三种选择：保留 upcaster 直到旧事件过期，把已存储的事件改写进新 topic，或者在别处对状态做快照并任由历史失效。每一种都有代价：永久的代码、必须信任的迁移，或者失去重放的能力。如果不知道系统重放得有多频繁，也不知道监管方期望保留记录多长时间，我就无法说出该选哪一种。

弃用事件

添加事件比移除事件容易，因为一个未知的消费者可能依赖其中任何一个。一个合理的生命周期有四个阶段。

1. **公告**。在 registry 或事件目录中把该事件标记为已弃用，指明替代者，给出下线日期和联系人。
2. **度量**。查明谁还在读取它：日志上的 consumer group 成员、队列上的绑定、registry 查询。应把这份名单当作证据，而不是证明，因为某个每月运行一次的批处理作业今天可能不在，下周又出现了。
3. **停止生产**。日期一到，就停止发布。
4. **下线**。一旦保留期限已让最后一个事件过期，读者也已离开，就移除 schema 以及 topic 或队列。

有两点提醒。没有期限的弃用永远不会结束，所以应当把日期写下来并守住它。而且，已弃用的事件在下线之后被重放，将会失败，所以在下线之前，应当先决定对受影响的时段，重放应该怎么做。

责任归属很重要。如第 11 章所述，生产者所在的团队拥有该事件及其弃用。消费者拥有各自的迁移。让生产者独自决定、或让消费者无限期拒绝的策略，都会失败。有效的策略是：生产者发出通知，消费者按日期完成。

权衡与成本

- **registry 是又一个要运行的服务**。它位于每个生产者部署的路径上，视配置而定，也位于消费者解码的路径上。它需要可用性、访问控制和一个负责人，客户端还需要一份它不可达时的预案。
- **兼容性规则有意拖慢生产者**。原本可以在一个下午重命名字段的团队，现在要执行六步迁移。这就是独立部署的代价。
- **upcaster 会不断累积**。每个历史版本都会增加一个分支和一项测试，在旧事件离开保留期限之前，代码不会变得更简单。无限期保留意味着无限期的 upcaster。
- **tolerant reader 会掩盖错误**。忽略未知字段的消费者，也就无从察觉生产者开始发送了某些它本应关心的内容。没有监控的容忍，会把响亮的故障变成无声的漂移。
- **双发布会成倍增加流量和去重工作**，持续时间与迁移一样长。
- **两个版本号而不是一个**。破坏性版本和 registry 修订版本使增量式变更不致令消费者失败，代价是团队要花力气在仪表盘和对话中把它们区分开。
- **版本号不是文档**。`schema_version: 4` 只说明某处变了，不说明变了什么，也不说明谁受影响。随 schema 一起保存的变更日志，比这个数字更有价值。
- **跨语言规则以便利为代价**。对大数字使用字符串、保留原始 payload、固定时间戳单位，意味着每种语言都要多写转换代码，并为每种语言多写一项测试。

何时不使用这些模式

对于只有一个生产者、一个消费者、且由同一个团队一起部署两者的事件，不要搭建完整的机制。协调部署成本更低，而所要防范的风险并不存在。当第二个消费者出现时，尤其是归另一个团队所有的消费者，再重新

评估。

不要把 registry 的严格模式当作设计的替代品。一连串安全的新增，可能留下一个有 40 个可选字段、却没有明确目的的事件。如果一个事件不断增长，它多半是两个事件，此时适用第 11 章的粒度规则。

不要为每次变更都创建版本化 topic。这样做的系统最终会留下许多只下线了一半的 stream，没有人知道哪一个是当前的。

常见错误

- **把重命名当作小变更**。这很诱人，因为新名称更干净，测试也通过了。但对每个读者来说，它都是一次删除加一次添加。应在旧名称旁边添加新名称，迁移读者，然后再移除旧名称。
- **就地更改字段的含义**。名称和类型都不变，所以没有任何工具提出异议。应当为新含义使用新字段或新的事件类型。
- **添加没有默认值的必填字段**。这感觉像是一次新增，但旧事件里没有它。应当把它设为带默认值的可选字段，或者通过新版本加一个 upcaster 来引入。
- **只针对最新版本测试兼容性**。重放会读取保留期限内的所有内容。应当针对仍然存在的每个版本检查，并在消费者测试中保留每个版本的样例事件（第 15 章）。
- **先升级生产者，后升级消费者**。生产者所在的团队拥有这项变更，也最先准备就绪。对于向后兼容的变更，消费者先行。部署之前应确认顺序。
- **遇到未知字段就大声失败**。严格的解析器会把每一次增量式变更都变成一次事故。应当忽略自己不使用的字段；只拒绝自己所依赖的字段。
- **悄悄地给缺失字段填默认值**。用零替代缺失的金额会掩盖缺陷，而在代码运行之前就填入 0 的解码器，会更早地掩盖它。对这类字段应使用“无值”默认值，检查存在性，并抛出错误，使失败路径把该事件留给人处理。
- **每次变更都递增 `schema_version`**。这样一来，每次增量式变更都会成为每个尚未升级的消费者的硬性故障。只对旧读者无法处理的变更才递增破坏性版本，其余的交给 registry 修订版本去计数。
- **经由带类型的类转发**。副本会丢掉类中没有声明的每个字段，而且没有任何东西提出抱怨。重新发出时应当保留未知字段，或者按自己的

契约发布一个新事件。

- **复用字段编号或 topic 名称**。被回收的编号会把旧数据读成新字段，被复用的 topic 会混入两种形态。应当停用标识符。
- **弃用时没有日期或负责人**。事件会一直拖着，下一个团队不敢移除它。应当给每次弃用一个日期、一个替代者以及一个回答问题的人。
- **回滚消费者时不检查 schema**。应当验证回滚目标能读取保留期限内的每个版本。

要点

- 事件 schema 是与不受自己控制的读者、以及很久以前写下的数据之间的契约；向后兼容意味着新读者可以读取旧数据，向前兼容意味着旧读者可以读取新数据，而带重放的日志需要针对保留期限内每个版本的向后兼容。
- 添加可选字段是安全的，而重命名、类型变更、新增必填字段和含义变更都是破坏性的，其中含义变更最糟，因为没有任何东西会大声失败；只有破坏性变更才会使 `schema_version` 变动，其余的由 registry 修订版本去计数。
- schema registry 的价值在于在部署流水线中强制执行兼容性，但它检查的是结构而不是含义，所以仍然需要人来评审一个字段到底应当表达什么。
- 优先选用能承载该变更的最廉价的版本管理策略：先用可选字段，当含义是另一个事实时用新的事件类型，只有整个 stream 必须破坏时才用版本化 topic。
- upcaster 和 tolerant reader 让消费者能够应对多个版本，前提是 upcaster 编码的是一条关于过去的真实陈述，读者检查的是存在性而不是值，并且拒绝自己无法安全解释的内容。
- 跨语言时，无声的破坏来自丢失的未知字段、大整数、时间戳单位、封闭枚举以及 null 与缺失的区别，所以应当保留自己不了解的内容，并把表示方式写下来。
- 把一次破坏性变更迁移为一连串非破坏性变更，让版本递增发生在旧读者唯一无法承受的那一步，并把回滚计划写进同一份文档。

只有能够看到系统中正在流动的东西，一份可以安全演进的契约才真正让人安心，第 13 章就从这里开始。

第五部分

运维与验证

第 13 章 — 可观测性与对账

设想 Parcelly 的一个周四上午。客服转来一张工单：一位买家两天前为订单付了款，银行卡账单上有这笔扣款，订单页面却仍然显示“等待确认”。客服打开 Orders 管理界面，看到的是 `PAYMENT_PENDING`。他们去问 Payments 团队，对方在自己的数据库中找到一行 `PaymentCaptured`，outbox 也是干净的。他们又去问 Orders 团队，对方的消费者日志里没有这个订单的任何错误。每个人的仪表盘都是绿色的。

三位工程师现在按订单号在三个服务里搜索日志。一条日志行用的是 `order-81724`，另一条用的是 `81724`，而 Payments 的 relay 只记录它内部的行 id。一小时后，他们在 Payments 的 outbox 中找到了这条事件，标记为已发布。又过了一小时，他们才得知，为 Orders 供数的 broker topic 上有一个 consumer group，在一次部署期间落后了。broker 的保留机制删除了该 group 尚未读取的一段消息，而该 group 的 offset 重置策略被设置为跳到最新位置，所以它越过这段缺口后继续运行，没有报出任何错误。事件存在，已发布，却从未被读取。

这里没有任何东西是评审者会指出的那种代码 bug。系统做了它被构建来做的事，只是没有人能看到它在做。更糟的是，这张工单之所以存在，是因为一位买家在任何告警之前就发现了问题，而且没有人知道还有多少其他订单也是这个样子。本章提出两个问题。团队需要看到什么，才能在几分钟内回答“这个订单在哪里？”；团队又如何找到那些尚无人投诉的订单？

问题及其难点

在同步系统里，故障通常会随请求带着堆栈跟踪一起到来。在事件驱动系统里，一个业务流程的工作分散在多个服务、多个数据库、一个 broker 以及若干个不同的时刻之中。没有任何一个进程掌握完整的故事，也没有任何一份日志包含它。

有三个特性使这件事变得困难。第一，**各跳之间在时间上是解耦的**。生产者完成后就忘了这件事，而消费者可能在几秒或几小时之后，在另一个进程中运行，手里没有生产者的任何调用栈。沿着嵌套调用追踪请求的工具，在第一条消息处就丢失了线索。

第二，**最糟的故障是静默的**。第 3 章描述过：消费者落后超过保留期限时，broker 会删除尚未读取的消息。第 8 章描述过：消息被错误地处理，却没有报出任何错误。这两种情况下什么都没有失败，所以什么都不会告警。系统只有事先知道某条事件应当到达，才能察觉它从未到达。

第三，**看上去健康的组件加起来，可能是一个损坏的流程**。relay 在运行，broker 在线，消费者报告没有错误，订单却依然卡住。组件健康回答的是“这个进程还活着吗？”，而业务问的是“这个订单完成了吗？”。这是两个不同的问题，本章的工作就发生在二者之间的缺口里。

第 3 章把故障的语义描述为三个问题：什么会失败，如何观察到它，哪一种保证在故障中依然存续。本章讨论的是第二个问题，以及当答案是“观察不到”时该怎么办。

做法分为两半。可观测性为我们提供追踪流程、察觉压力正在积聚的手段。**对账**，即比较两个系统各自眼中的事实并修复差异，则提供了一种找出任何信号都没有报告的问题的方法。只有前一半的团队，看到的是响亮的故障。只有后一半的团队，发现问题要晚上好几天。我会两者都建。

跨异步跳传递上下文

每个事件已经携带了第 11 章的两个标识符：`correlation_id`，由同一业务流程中的每个事件共享；`causation_id`，指向直接的触发者。它们是故事的主干。分布式 tracing 增加了第二条线索，记录耗时以及工作单元之间的父子关系。

trace 是由带计时的 span 组成的一棵树，每个 span 是一个进程中的一个工作单元。在进程内部，tracing 库把当前 span 保存在环境状态中。跨越一条消息时，这份状态会消失，除非把它放进消息里。标准做法是把 trace 上下文，即 trace id、span id 和采样标志，序列化到消息头中。W3C Trace Context 为此定义了通用格式，包含 `traceparent` 和 `tracestate` 两个消息头，大多数 tracing 库都遵循它。消息头的容量与传播方式因 broker 和桥接而异：有的限制消息头的数量或大小，有的桥接会丢弃它们，所以要检查消息实际经过的路径。

上下文在何处捕获，比乍看起来重要得多。第 4 章正是为此在 outbox 中放了一个 `headers` 列。relay 稍后在另一个进程中运行，没有环境 span，所以在发布时才注入上下文的 relay，会附上它自己的 trace，而不是下订单的那一个。上下文必须在写入 outbox 行时捕获，也就是在生产者的事务之内。

```
function writeOutboxEvent(event):
    // runs inside the caller's transaction
    headers = {}
    injectTraceContext(currentSpan(), headers)
    headers.correlation_id = event.correlation_id
    outboxInsert(event, headers)

function relayPublish(row):
    // the relay forwards stored headers verbatim and never
    // recomputes them: a republication must carry the original
context
    publish(topicFor(row.event_type), row.event, row.headers)
```

在消费者一侧，把新 span 与旧 span 关联起来有两种方式。消费者 span 可以作为生产者 span 的子节点，这样画出的是一棵树。或者，它可以开启一条新的 trace，并持有一个指向生产者上下文的**链接**（trace）。选哪种取决于具体情形。

情形	关系	原因	代价
单条消息，及时消费	作为生产者 span 的子节点	从请求到效果是一幅完整的瀑布图	如果投递延迟，trace 会在整个延迟期间保持打开，耗时数据会产生误导
批处理或扇入	新 trace，每个生产者一个链接	一个 span 只有一个父节点，所以批处理消费者不能成为十二个生产者的子节点	后端必须支持链接，而且得有人去浏览它们
扇出	一个生产者 span，每个订阅者一个子消费者 span	树仍然是树	每增加一个订阅者，trace 就变大一些
重放、redrive 或延迟投递	带链接的新 trace	一个已有数小时的父节点会拉长原来的 trace	采样决策会重新做一次，所以被采样的生产者 trace 可能链接到未被采样的消费者 trace

我仍然默认使用链接，因为延迟和重新投递正是本书所规划的常态。代价是失去大多数 tracing 界面为父子关系绘制的那幅端到端的瀑布图。对于总是在生产者之后片刻内运行的消费者，父子关系是更好的选择，我会

采用它。

```
function startConsumerSpan(message):
    producerContext = extractTraceContext(message.headers)
    span = newSpan("consume " + message.event_type)
    if producerContext exists:
        span.addLink(producerContext)
    span.setAttribute("event_id", message.event_id)
    span.setAttribute("correlation_id", message.correlation_id)
    // from a broker attribute where one exists; otherwise from a
header
    // that the retry path increments on every republish
    span.setAttribute("delivery_attempt", deliveryAttempt(message))
    return span
```

注意这个 span 记录了 `event_id` 和投递尝试次数。基于日志的 broker 不保存投递计数，只有部分基于队列的 broker 会暴露它，所以在依赖这个数字之前，应先确定它从哪里来。当消息被重新投递时，会得到第二个带有相同 `event_id` 的 span。这一对 span 说明消息被投递了两次，而下文所述的 `outcome` 字段说明它是否被应用了两次。

tracing 有其局限，我会小心不把保证押在它上面。后端会采样，所以所关心的那个订单的 trace 可能并不存在。上下文消息头可能被桥接、死信副本或手写的生产者丢弃。避免依赖消息头的一种办法，是把 trace 标识符放进消息体：BabelQueue 信封（babelqueue.com，一个开源的、与语言无关的标准）在消息体中携带一个 `trace_id`，由第一个生产者生成，后续的生产者原样复制，并且在转入死信后依然保留，所以即使某一跳丢掉了消息头，线索也不会断。止步于 outbox 行的 trace，永远到不了尚未读取该行的消费者。因此，信封中的标识符比 tracing 库更重要：`correlation_id` 必须出现在处理该事件的每一行日志和每一条数据库记录中，这样即使 tracing 关闭，它也能发挥作用。



读这张图时，要看那个不变的东西：`correlation_id` 在每一跳都保持不变，而每个服务都在自己的 outbox 中存入一份新的 trace 上下文。

还原一个订单的完整经过

有了标识符，“订单 81724 在哪里？”就成了一次查询。最廉价的实现是一种约定，而不是一个产品：每个服务为每条处理的事件、每条发布的事件各写一行结构化日志，字段包括 `correlation_id`、`event_id`、

`event_type`、`causation_id`、结果和时间戳。在这些日志中搜索 `correlation_id = "order-81724"`，就能得到每个服务中的每一跳。

```
log {
    at: "2026-03-14T09:41:09Z", service: "orders",
    action: "consumed", outcome: "applied",
    event_type: "PaymentCaptured", event_id: "3f2a9c1e-…",
    occurred_at: "2026-03-14T09:41:07Z",
    correlation_id: "order-81724", causation_id: "b81d77f0-…",
    delivery_attempt: 1, lag_ms: 1840
}
```

`outcome` 字段值得留意。应使用一个小而封闭的集合：`applied`、`duplicate_skipped`、`stale_ignored`、`failed_retry` 和 `dead_lettered`。第 9 章论证过，被忽略的过期事件值得留下痕迹，第 7 章则讲述产生 `duplicate_skipped` 的 inbox 检查。一旦这些结果成为字段，“这个消费者上一小时把多少事件当作过期丢弃了？”就成了一次计数，而不是一场调查。

按 `occurred_at` 排序得到时间线，沿着 `causation_id` 追踪得到一棵树：`OrderPlaced` 引发了 `PaymentAuthorized`，后者与 `StockReserved` 一起引发了 `CapturePayment`，依此类推。在开头的场景中，这棵树会显示 `PaymentCaptured` 已由 Payments 发布，而 Orders 中没有 `consumed` 行。第一个缺失的那一跳，就是应当查看的地方。这比任何单独的一行日志都更有用：链条中的缺口是某物丢失的证据，而一条很长的链条只能说明流程有所进展。

如果运行着第 10 章的 saga，它的 `saga_instance` 行就是第二个信息来源。它回答“编排器认为状态是什么？”，而日志回答“实际发生了什么？”。当两者不一致时，就发现了值得开一张工单的东西。

对于最重要的流程，我还会保留一份**事件日志簿**：一份 append-only 的事件信封记录，每条记录带有其 payload 的哈希而不是 payload 本身，由每个服务写在它的 inbox 旁边。它要消耗存储和一些写入负载，作为回报，在 broker 已经忘记这些事件之后，仍然可以回答问题，包括重建出来的事件是否仍与原始事件一致（见下文修复一节）。保留时间要足以让最慢的对账跑完。

重要的指标

指标让人在任何单条消息出错之前就感知到压力。事件驱动系统中有用的指标不多，其中大多数在前面的章节里已有所暗示。我会从这些开始。

信号	度量的内容	重要的原因
Consumer lag（消息数）	消费者落后 head 多远	积压规模；有助于容量规划
消息年龄，消费者侧	自最旧的未处理消息产生以来的时间	业务所体验到的延迟
Outbox 年龄	最旧的未发布行的年龄	relay 的健康状况（第 5 章）
端到端延迟	从流程第一个 `occurred_at` 到最后一个消费者中产生效果的时间	流程真正的截止时间
retry 率与尝试次数	需要多于一次尝试的消息所占比例	被 retry 掩盖的退化（第 6 章）
DLQ 新增消息及其年龄	新增的死信，以及最旧的未经评审的那一条	需要人处理的故障（第 8 章）
被忽略与重复消息的计数	被丢弃的过期事件，被跳过的重复消息	消息顺序与重新投递带来的压力
消化速率与追平时间	处理速率与到达速率之比；lag 除以净消化速率	积压是否会在其截止时间或保留上限之前清空（第 6 章）

Lag，以及为什么消息年龄更好

以消息数计的 **consumer lag** 是大多数仪表盘最先显示的数字，也最容易被误读。孤立地看，10,000 条消息的 lag 没有多大意义：如果消费者每秒消化 5,000 条，它落后两秒；如果每秒只消化 10 条，它就落后一刻钟。以消息数计的 lag 回答的是容量问题。以时间计的 lag 回答的是业务问题。

消息年龄是自仍在等待的最旧消息产生以来的时间。有些基于队列的 broker 直接暴露最旧消息的年龄；对其他 broker，则从最旧的等待消息上的时间戳推导。在基于日志的 broker 上，则从消费者已提交的 offset 处那条记录的时间戳推导。两种情况下，这个数字都以秒为单位，因此可以与业务选定的截止时间比较。

用来相减的时间戳很重要，因为候选的几种度量的是不同的东西。broker 的追加时间度量一条消息在 broker 处等待了多久，反映的是消费者的健康状况。`occurred_at` 度量的是事实发生以来的时间，其中包含一行在 outbox 中等待 relay 的时间，这是业务眼中的延迟。生产者的创建时间取决于配置：根据配置不同，日志 broker 存储的要么是创建时间，要么是追加时间，而重新发布旧行的 relay，除非显式设置，否则会得到一个新的创建时间。我会用追加时间作为消费者年龄的 gauge，用 `occurred_at` 度量端到端延迟。用 `occurred_at` 做这个 gauge，会使一个健康的消费者在 relay 重新发布旧行的那一刻看起来严重迟到。一个 group 的年龄取其各分区的最大值，告警也应使用这个最大值。

有一个陷阱值得点明。没有新流量的分区，lag 为零，而按朴素的算法，其年龄要么为零，要么永远增长。应当决定空闲分区如何上报，并对此做测试。同一个函数还必须经得起下文所述的保留故障：已提交的 offset 落到了日志起点之后，就指向已被删除的数据，所以这项检查要放在最前面，并单独上报。如第 8 章所述，卡在 poison record 上的消费者，吞吐量为零而年龄不断上升，这个组合应当成为呼叫值班人员的条件。

```
function reportConsumerAge(partition):
    committed = committedOffset(partition)
    head = headOffset(partition)
    if committed >= head:
        // nothing waiting: report zero, and report liveness
separately
        emitGauge("message_age_seconds", 0, partition)
        return
    logStart = logStartOffset(partition)
    if committed < logStart:
        // retention already deleted unread data: report how much, as
a
        // gauge that follows the condition and not how often we
checked
        emitGauge("unread_offsets_lost", logStart - committed,
partition)
        // the gauge clears at the next commit, so also leave a record
        logLostRange(partition, committed, logStart)
        committed = logStart
    record = readRecordAt(partition, committed)
    // append time: how long this record has waited at the broker
    emitGauge("message_age_seconds", now() - appendTime(record),
partition)
```

保留期限是看不见的截止时间

消息年龄还有第二项不那么明显的作用。在基于日志的 broker 上，落后超过保留期限的消费者会悄悄丢失消息：broker 删除了从未被读取的数据，消费者则从其 offset 重置策略所指向的位置继续。在基于队列的 broker 上，消息过期时也会发生同样的事。第 3 章指出过这种故障，而它不伴随任何错误。

重置策略是一项决策，每个选项都有代价。重置到最早可用的 offset，会重放剩下的所有内容，所以丢失是局部的，消费者必须消化涌入的大批旧消息。重置到最新的 offset，会向前跳过，使丢失变得无声，开头的场景中发生的正是这一点。选择失败，让消费者停下来等待人工处理，则会把无声的丢失变成一次事故。对于任何缺口会造成资金损失的流程，我会选择失败，并接受在有人做出决定之前消费者处于停止状态。第 14 章讲述在发现缺口之后如何恢复：重置到某个时间戳并重放，同时抑制副作用。

防御靠的是算术。把消息年龄与保留期限或过期期限比较，当年龄达到其某个比例（比如一半）时告警。此时截止时间仍在前方，人还有时间增加消费者、暂停生产者或延长保留期限。截止时间过去之后，唯一的证据是事件的缺席，而这正是对账所要检测的。我会把这个比例与年龄本身放在同一块仪表盘上，因为对业务可以接受、对保留期限却是致命的 lag，是真实存在的情形。

对于 CDC connector，对应的信号是 connector 尚未消费的数据库日志有多少，与数据库愿意为它保留的日志有多少相比。宕机时间超过数据库保留日志的时间的 connector，可能需要重新做一次快照才能恢复，而这本身就是一次停机。

延迟、retry 及其他

端到端延迟需要先有定义，然后才需要图表。我会从流程中第一个事件的 `occurred_at` 开始，度量到最终效果被提交的时刻，例如从 `OrderPlaced` 到 `OrderConfirmed`。应报告百分位数，而不是平均值：缓慢的长尾就是那些给客服写信的买家。最后一个消费者只能看到它所消费的信封，而这个信封携带的是它自己那个事实发生的时间，不是流程的起点。要么在消息头中传递流程的起始时间，要么凭 `correlation_id` 在事件日志簿或所属服务中查到它。然后用最终的 `event_type` 给这项度量打标签。两个时间戳来自不同服务的时钟，所以结果中会有一些偏差；与业务截止时间相比，它很小，这也是不要对亚秒级差异告警的原因。

消化速率是反压的可观测形式。用 lag 除以消费者处理消息的速率与消息到达速率之差，就得到追平时间。如果到达超过处理，lag 就永远清不掉，这正是应当告警的数字，应当远在年龄达到保留上限之前。第 3 章定义了反压，第 6 章讨论如何限定消费者承担的工作量。

retry 率与尝试次数暴露出第 6 章所称的隐藏故障：某个依赖已经退化了数周，而 retry 在悄悄地将其吸收。尝试次数的直方图比错误数提供更多信息，因为一条总是在第三次尝试时成功的消息，根本不会产生任何错误。

DLQ 新增消息以及最旧的未经评审死信的年龄来自第 8 章。再加上来自第 7 章和第 9 章的**重复消息与过期消息的计数**。部署之后重复消息计数上升是正常的，因为工作的归属发生了转移；而在平稳时段计数上升，则指向缓慢的处理器。在基于队列的 broker 上，常见原因是 visibility timeout 比处理器的运行时间短。在基于日志的 broker 上，原因是 rebalance，或者是消费者超出了轮询超时而失去了它的分区。

有意义的告警

大多数告警体系的失败，不在于数据太少。而在于数据是为原因而不是为影响去呼叫人，以及在数周的噪声之后，没有人再信任 pager。我对告警只坚持一条标准：如果它在凌晨三点触发，一个具名的人能说出该怎么办。

应当针对**与业务截止时间挂钩的症状**告警。“最旧的未发布 outbox 行已超过两分钟”是一个症状，表明状态变更与其宣告之间有东西卡住了。“消费者 CPU 超过 80%”是一个原因，它可能重要，也可能不重要。原因类指标应放在供人调查的仪表盘上，而 pager 留给症状。

给每个告警配一个**负责人和一个有理由的阈值**。第 8 章对 DLQ 说过同样的话：丢一封邮件和丢一次扣款，应当有不同的阈值，这种差别应当写在告警旁边。理由之所以重要，是因为阈值会漂移，而一个没有说明理由的数字，会在第一次令人厌烦时就被调高。

告警既要针对**存在，也要针对缺席**。崩溃的 relay 不会发出错误；它什么也不发出。一个丢失了全部成员的 consumer group 会显示 lag，但一个彻底停止接收事件的 topic，既没有 lag，也没有错误。对于有稳定节奏的流程，例如营业时间内的订单，诸如“10 分钟内未观察到

`OrderPlaced` 事件”这样的告警，能抓住所有其他信号都漏掉的生产者侧的沉默。阈值必须跟随流量模式，而对没有节奏的流程，它是错误的工具。

最后，对大多数团队来说，两个层级就够了：任何威胁截止时间或丢失数据的情况发出呼叫，而缓慢的压力，例如堆积着低价值消息的 DLQ，开一张工单。第三个层级往往会被忽视。

> **警告**：为了消除噪声而调高告警阈值，是团队用一份烦恼换来一个盲区的办法。如果某个告警触发得太频繁，应当修复它的原因或它的定义。不要把数字调高到它不再触发为止。

对账

指标和 trace 观察的是系统做了什么。它们无法告诉我们一个从未进入系统的事件、一个没有报错却提交了错误状态的处理器，或者某人手工做的一次修复。对于这些，需要把系统与一份独立的、关于什么应当为真的记录做比较。这就是对账。

对账作业取同一批事实的两种视图，并报告差异。一旦去找，这些配对通常是显而易见的：

- **Orders 对 Payments**：每个处于 `PAID` 状态的订单都有一笔已扣款的支付，每笔已扣款的支付都属于一个知道它的订单。
- **Payments 对 Ledger**：每个 `PaymentCaptured` 都有一个金额相同、币种相同的 `LedgerEntryPosted` 与之匹配。
- **自己的记录对服务商的记录**：支付服务商的结算报告对自己的扣款记录。第 7 章在讲到无法做到幂等的效果时指向这里，对账是其最后一道防线。
- **读模型对其来源**：搜索索引，或第 9 章所说可以标记为存疑、稍后再修复的视图。

三种差异类型涵盖了大多数情形：**缺失**（来源有，派生一侧没有）、**过期**（两边都有，但派生一侧不一致，通常因为它落后了）和**多余**（派生一侧有来源不知道的东西）。每一种指向不同的原因。缺失暗示事件丢失或从未被消费，过期暗示消息顺序或 lag，或者偶尔是处理器写错了值，多余则暗示 bug 或人工修改。

需要格外小心的一点是，在途的事件看起来与差异一模一样。四秒钟前扣款、尚未进入 Ledger 的支付是正常的。把它报告出来的作业只会产生噪声，并教会人们忽略它。所以作业只比较早于**宽限期**的记录，宽限期根据正常的端到端延迟选定，并且在报告之前，对不匹配的项再确认一次。

```
function reconcileCaptures(window_start, window_end):
    upper = min(window_end, now() - gracePeriod)
    suspects = []
    for each batch in readCapturesInWindow(window_start, upper,
batchSize):
        // list per payment_id
        entries = fetchLedgerEntries(keysOf(batch))
        for each capture in batch:
            found = entries[capture.payment_id]
            if count(found) is 0:
                suspects.add({ key: capture.payment_id, kind:
"missing" })
            else if count(found) > 1:
                suspects.add({ key: capture.payment_id, kind: "extra"
})
            else if differs(found[0], capture):
                suspects.add({ key: capture.payment_id, kind: "stale"
})
    for each batch in readLedgerEntriesInWindow(window_start, upper,
batchSize):
        captures = fetchCaptures(keysOf(batch))
        for each entry in batch:
            if captures[entry.payment_id] is null:
                suspects.add({ key: entry.payment_id, kind: "extra" })
    // a second look after a delay removes events that were merely in
flight
    sleep(recheckDelay)
    confirmed = filter(suspects, stillMismatched)
    for each item in confirmed:
        // upsert on (key, kind): first_seen_at survives, last_seen_at
moves
        upsertDiscrepancy(item.key, item.kind, now())
    // anything open in this window that no longer mismatches is
closed here
    resolveOpenDiscrepanciesInWindow(window_start, upper, except:
confirmed)
    return count(confirmed)
```

这段代码中有几处细节承担着正确性。窗口是有界的，截止时间已经应用，所以作业读取的是已经有时间沉淀下来的数据。每个批次都是按键的查找，而不是跨数据库的 join，因为 Parcelly 的服务不会互相读取对方的表，所以作业通过对方服务自己的接口或一份已发布的提取数据来获取另一侧。作业只记录差异，不改变任何东西。而且由于它是以条目和类型为键向差异表做 upsert，运行两次，或在窗口中途崩溃之后再运行，产

生的是同一张表，不会堆积重复的行。同一个函数还会在两侧一致后关闭差异，下文的修复代码依赖的正是这一步。向 Ledger 记了两次账的支付，会被统计匹配数的正向遍历发现。反向遍历则负责找出没有扣款支撑的“多余”条目，例如为一笔从未扣款的支付所做的记账，而单向的检查永远发现不了这种情况。

应根据差异造成的损害来选择频率。Payments 对 Ledger 涉及资金，值得频繁运行，并以每日一次的全量比较作为兜底。搜索索引或许每晚运行一次就够了。作业自身的健康同样重要：如果它没有在预期的窗口内完成，就应告警，因为悄悄停止的对账比没有对账更糟，人们会相信它仍在盯着。

> **注意**：对账作业是它所检查的数据的一个消费者，它也可能出错。应先把差异当作调查的线索。新作业的头几周，大部分时间都花在查找由时序、舍入或与业务不符的比较规则造成的误报上。

修复与 backfill

发现差异只是工作的一半。另一半是修改状态使其一致，而不产生第二个差异。

我所坚持的规则是：**修复走与正常变更相同的路径**。如果 Ledger 缺少一笔已扣款支付的条目，不要用数据库控制台插入那一行。第 3 章把运维人员的操作视为故障模型的一部分，而手工编辑的行不会发出任何事件，所以该事实的其他每个消费者仍然是错的，下一次对账会报告一个由自己造成的不一致。正确的做法是，通过生产者的 outbox 重新发布该事实，或发送引发它的命令，这样消费者就会像处理其他一切那样处理它。

修复可能被执行 retry、被打断，也可能被两位工程师各运行一次，在每种情况下它都必须保持安全。只有当修复事件携带**原始**的 `event_id` 时，第 7 章的幂等性才能回答这个问题。带有新 id 的修复事件，对每个消费者而言都是另一个事件。如果第一个事件只是被延迟了，它会在修复之后到达，Ledger 就会把这笔支付记两次。所以，修复要重新发布已经存在的那个事实：相同的信封，相同的 `event_id` 和 `causation_id`，而 inbox 把后到的那一份当作重复消息处理。

这要求生产者保留原始 id。最简单的办法是，在写入 outbox 行的同时，把 `event_id` 连同 payload 的哈希一起存在业务记录上（这里是扣款记录）。如果 outbox 行仍然存在，就把它标记为未发布，由 relay 再

次发送。如果它已被清除，就用存储的 id 从记录重建信封。

重建有其限度。记录是当前状态，而事件是过去的事实。如果记录此后发生了变化，比如一次部分退款调整了金额，那么据此重建的 payload 会与原始 payload 不同，却带着相同的 `event_id`。这就是要存储哈希的原因：修复时重建 payload，把它的哈希与存储的哈希比较，不一致就停下来交给人处理。没有存储 id 或哈希的记录，通常是在养成这一习惯之前写入的，必须回退到由其业务键派生出的 id，以及不会改变的字段，此时消费者还必须靠该键上的唯一约束来保护，如第 7 章所述。无论如何，我都会给 Ledger 加上这个约束，作为第二道防线。

重新发布还会**扇出**。topic 会把信封投递给每个订阅者，而不只是 Ledger。inbox 中仍保有该 `event_id` 的订阅者会跳过它。inbox 行已被清除的订阅者，因为该事件已经超出其去重窗口（第 7 章），会再应用一次，Notifications 就可能再发一次同样的邮件。所以有两条安全的路径。如果事件比所有订阅者中最短的去重窗口更新，就按上文所述通过生产者的 outbox 重新发布。如果更旧，就只把修复投递给漏掉它的那个消费者：通过该消费者自己的修复队列或 topic，或者重置那一个 group 的 offset，并依靠业务键约束提供保护。第 8 章对 redrive 应用了同样的原则。

```
function repairMissingLedgerEntry(discrepancy):
    capture = loadCapture(discrepancy.item_key)
    if ageOf(capture.event_id) > shortestDedupWindow():
        // a broadcast could be applied twice where the inbox row is
gone
        return sendRepairToConsumer("ledger", capture, discrepancy)
    try:
        transaction:
            original = outboxFindByEventId(capture.event_id)
            if original exists:
                // same envelope and event_id; the relay publishes it
again
                outboxMarkUnpublished(original.id)
            else:
                // purged: rebuild the identical envelope, never a new
id
                envelope = rebuildOriginalEnvelope(capture)
                if hashOf(envelope.payload) != capture.payload_hash:
                    // the record changed since the event was
published
                    raise RepairNeedsReview(discrepancy.id)
                outboxInsert(envelope)
            recordRepairAttempt(discrepancy.id, actor, now())
        // commit point: the repair is durable and the relay delivers
it
    on UniqueViolation as e:
        if e.constraint != "outbox_event_id_key":
            // a different constraint: a real error
            raise e
        // a concurrent repair inserted the same event_id first
        return
    // resolution is recorded by the next reconciliation, not here
```

有两处细节承载着安全性。把一行未发布的记录再次标记为未发布，重复执行是无害的；而第二次并发插入引发的唯一约束冲突，被当作“已处理”，而不是错误。重新发布的信封还保留其原始的 `causation_id`，因此因果链保持真实，而与差异的关联存放在修复记录里。最后一条注释最重要。修复函数不会把差异标记为已解决。这件事由下一次对账运行完成，在两侧一致时才会完成。这样可以避免一次已发布却丢失的修复被报告为已修好。

批量修复，通常称为 **backfill**，增加了规模问题。把 30,000 条事件 backfill 进一个在线运行的消费者，会与正常流量竞争，并可能把流程的延迟推过其截止时间。应当分小批次运行，带速率限制和暂停开关，如第 8 章对 redrive 所建议的那样，并在运行期间盯着消息年龄和 consumer lag。在基于日志的 broker 上，可能更倾向于有意地把事件重放到一个

独立的 consumer group 中，由它重建派生视图，然后切换过去。当消费者是投影时，这样更干净，而当消费者有发送邮件之类的副作用时，这样就是错的。

经由生产者的修复适合少量记录。当损害范围更广时，例如某个 consumer group 跳过了整整一段保留缺口、数据库被恢复或 broker 集群丢失，逐条记录地重新发布就不再是合适的工具。第 14 章讨论通过重放恢复、灾难以及 broker 迁移。

每次修复都需要一份记录：谁执行的，涵盖了哪些差异，发布了什么，何时发布。审计轨迹让下一位工程师得知 14:10 的重复消息激增是一次 backfill，也让监管方或财务团队日后得知某条 Ledger 条目是一次更正。

业务不变量

对账比较的是两个系统。**不变量检查**比较的是一个系统与业务认定为真的一条规则。不变量往往评估起来更廉价，并能抓住任何第二个系统都发现不了的问题，例如一个使两个系统在错误答案上达成一致的 bug。

以 Parcelly 为例，下面几个例子展示了它的形态：

- 没有哪个订单在没有已扣款支付的情况下处于 `CONFIRMED`。
- 没有哪个预留的持有时间超过其截止时间。
- 没有哪个 saga 实例在超过其 `deadline_at` 后仍处于 `AWAITING_RESULTS`。
- 每笔已扣款的支付恰好有一条 Ledger 条目，不是零条，也不是两条。
- 一个订单的 Ledger 条目之和平衡为零。

每一条都是一个应当返回零行的查询。按计划运行它，并把返回的每一行当作一个有负责人的差异。saga 的例子正是第 10 章遗留的问题：超时处理器是防止订单冻结的第一道防线，而针对卡住实例的不变量查询是第二道，因为本应触发的处理器自己也可能失败了。

```
table invariant_violation:
    invariant_name  text not null
    item_key        text not null
    first_seen_at   timestamp not null
    last_seen_at    timestamp not null
    resolved_at     timestamp null
    primary key (invariant_name, item_key)
```

以不变量和条目为键，意味着持续存在的违规会被更新而不是重复插入，`first_seen_at` 则说明一个问题已经开放了多久。这个年龄本身就是值得告警的指标。

我会与产品和财务人员一起编写不变量，而不是替他们编写。工程师知道系统能做什么，却不一定知道业务把什么视为缺陷。像“退款绝不超过扣款”这样的规则，对财务而言显而易见，在代码里却可能哪里也没有写。

权衡与成本

可观测性与对账不是免费的，也不会自己运转。

遥测有代价。为每条处理的事件写结构化日志，会成倍增加日志量，而指标上 `order_id` 这样的高基数标签可能压垮指标后端。应把标识符放在日志和 trace 中，把指标标签限制在服务、topic 和事件类型这样的小集合内。对 trace 采样，并接受自己需要的那条 trace 可能并不存在。

事件日志簿会复制数据。对于“发生了什么？”具有法律或财务分量的流程，这样做是值得的。对于装饰性通知这类低价值流程，日志就够了，事件日志簿只是没有读者的成本。

对账会读取生产数据。比较作业可能加重源数据库的负载，批处理做得草率的作业可能拖慢它所保护的服务。如有副本，应在副本上运行，并限定其批次大小和速率。它还增加了一个依赖：如果另一侧的接口宕机，作业就无法运行。

告警消耗注意力。每一次呼叫都在消耗下一次呼叫的可信度。一小组归属明确的告警，比一大堆无人阅读的告警更有用。

修复自身有风险。修复机制是一条在正常请求流程之外改变生产状态的通路。没有审计、对危险类型没有审批、设计上也不幂等，它就会用一类更糟的问题取代另一类问题。

何时不必搭建这一切：只有一个服务、一个消费者、风险低、流程短的系统，不需要事件日志簿、不变量表或每晚的对账。带有 `correlation_id` 的结构化日志，加上针对消息年龄的告警，是我在任何规模下都会坚持的最低配置。其余部分随静默故障的代价而增长。如果一个丢失的事件会造成账本差异，就先建对账。如果它只会造成一个过期的搜索结果，就最后再建。

常见错误

- **针对健康状况告警，而不是针对流程。**绿色的 CPU 和内存曲线说明进程还活着，却说不出订单是否完成。这很诱人，因为基础设施指标已经存在。应当添加症状类告警：消息年龄、outbox 年龄和不变量违规。
- **只盯着以消息数计的 lag。**消息数没有附带截止时间。应当换算成年龄，并把年龄与业务截止时间和保留期限做比较。
- **在 relay 中注入 trace 上下文。**relay 没有环境 span，所以会附上错误的那个，或者什么也没有。应当在写入 outbox 行时捕获上下文，并原样转发。
- **标识符不一致。**一个服务记录 `order-81724`，另一个记录 `81724`。这样按 correlation 搜索，恰恰在最需要的时候失败。应在事件设计中一次性定义格式，并通过共享的日志辅助工具强制执行。合成流量也要保持这唯一的格式：第 15 章用单独的消息头标记 probe，绝不使用特殊的 `correlation_id`。
- **所有年龄都用同一个时间戳。**生产者的创建时间、broker 的追加时间和 `occurred_at` 度量的是不同的延迟。应用追加时间做消费者年龄的 gauge，用 `occurred_at` 度量端到端延迟。
- **把修复重新发布给每个订阅者。**inbox 行已被清除的消费者会再次应用该事件。应当用最短的去重窗口检查该事件，或者把修复只投递给漏掉它的那个消费者。
- **没有宽限期就做对账。**作业会把每一个在途事件都报告为差异，团队学会了忽略它，唯一真正的发现就淹没在噪声里。应当只比较已沉淀的记录，并在报告之前再核对一遍。
- **通过手工编辑行来修复。**它只修好了一张表，不发出事件，使每个下游消费者仍然是错的。应当通过生产者和 outbox 来修复，让这次变更像其他任何变更一样传播。

- **不幂等的修复**。第二次运行会让更正重复。应当以原始的 `event_id` 重新发布，让 inbox 既能识别第二次运行，也能识别迟到的原始事件。
- **任由对账腐烂**。作业在一次 schema 变更之后坏了，没有人注意到，团队还以为自己受到了覆盖。运行延迟或失败时应当告警，并按计划评审差异积压。

要点

- 分散在多个服务的流程，需要能经受住每一跳的标识符：每行日志、每条记录中的 `correlation_id`，以及存放在 outbox 行中的 trace 上下文，这样 relay 转发的才是正确的那一份。
- tracing 有帮助，但会被采样，也可能被丢弃，而链接与父子关系是视情形而定的权衡，所以信封中的标识符必须足以独立还原一个订单的经过。
- 对照业务截止时间和 broker 保留期限度量的消息年龄，比以消息数计的 lag 更有用，也是在落后的消费者悄悄丢失事件之前向人预警的信号；offset 重置策略决定这样的丢失是无声的还是响亮的。
- 针对与截止时间挂钩的症状告警，给每个告警一个负责人和一条说明阈值的理由，并包含针对沉默的告警，因为丢失数据的故障往往不产生任何信号。
- 对账能发现任何指标都没有报告的问题：它比较同一事实的两种视图中已沉淀的记录，把差异分类为缺失、过期或多余，并只记录它们而不改变任何东西。
- 修复必须走正常路径，以原始的 `event_id` 重新发布，使迟到的原始事件与修复事件不会同时生效，并留有审计轨迹；差异由下一次成功的比较关闭，而不是由修复本身关闭。重新发布会到达每个订阅者，所以旧事件只发给漏掉它们的那个消费者。
- 业务不变量检查，例如没有已扣款支付就没有已确认订单，能发现两个彼此一致的系统所掩盖的缺陷。

观察和对账告诉我们系统行为异常，而修复能修好少量记录。第 14 章讨论损害超出修复所能触及的范围时该怎么办：通过重放恢复、从灾难中幸存以及迁移到另一个 broker。

第 14 章 — 恢复、灾难与 broker 迁移

设想 Parcelly 的某个周二。承载整个市场平台所有事件的托管 broker 集群无法访问。服务商的状态页面说，托管该集群的可用区已经故障，一小时内会提供替换集群。替换集群上线后，topic 都在，但要么是空的，要么保留着截至最近几分钟内某个时间点的数据。没有人能说清这个时间点究竟在哪里。

服务本身没有问题。Orders、Payments、Inventory、Shipping 和 Ledger 各有自己的数据库，每个数据库都完好无损。每个服务都有一个 relay 在不断对发布执行 retry，outbox 越积越多。消费者处于空闲状态，等着一个不存在的 broker。与此同时，团队必须决定 broker 恢复后怎么办，而各种方案听上去都说得通：把消费者指向新集群，让它们继续运行；从 outbox 重新发布全部内容；恢复某个东西的最近一次备份。每种方案对一个存储是正确的，对另一个存储则是错误的。

本章讲的是某个存储出错、丢失或被有意替换之后的那一天。它只问一个问题：当 broker、数据库或区域不再是原来的样子时，已构建的保证中还有哪些站得住，又该如何借助它们回到一个可以信任的状态？

问题及其难点

前面的章节防御的是不会损坏存储的故障：两个步骤之间的崩溃、缓慢的消费者、重复消息、poison message。恢复则不同，因为出问题的是记忆本身。某个存储回到了过去，或者丢失了一部分历史，或者正在被另一个存储替换。本书中的每一项防御都依赖某个存储记住了某些东西，所以问题就变成：哪些记忆幸存了，哪些没有。

这很难，原因有二。第一，各个存储独立失效，而且方向各不相同。一个恢复到六小时前的数据库，和一个丢失了最后两分钟数据的 broker，彼此不一致，而双方都无法察觉。两者各自都是自洽的。第 3 章把这称为丢失整个存储，并指出起初不会有任何错误被抛出。

第二，恢复工作很少发生，而少见的工作在压力下往往做不好。把 consumer group 往回拨的命令、阻止重放再发一轮邮件的设置、各服务重启的顺序：这些都是提前做出决定成本很低、临场即兴发挥代价很高的事情。

本章的论点是本书论点最朴素的形态。幸存下来的不是 broker。丢失一个集群、恢复一个数据库或更换一款产品之后，留存下来的是围绕它设计的故障语义：记住了从未被确认之事的 transactional outbox（第 4 章）、认得出已经完成之事的 inbox、稳定的标识符、契约以及比对两种真相视图的习惯。这些都就位时，broker 是可以替换的。若没有，那就是 broker 一直在承担系统的正确性，而没有人知道。

通过重放恢复

重放在第 2 章中定义，是从事件日志中有意重新读取历史。它只适用于日志族，因为队列在投递之后就删除了消息。第 15 章把重放用作一种测试技术。在这里它是一种恢复工具，并且有一套步骤。

它适用的情形正是第 13 章开篇的那一种：某个 consumer group 落后了，保留期限删除了它尚未读取的消息，该 group 越过空缺继续前进。如果数据仍在保留期限内，首先要考虑的恢复手段不是从生产者那里逐条重新生成事件，而是把该 group 的位置移回空缺之前，重新读取。

将 consumer group 重置到某个时间点

consumer group 的位置按每个分区一个 offset 来存储。将其向后移动是 broker 上的一项管理操作，大多数基于日志的 broker 既可以指定明确的 offset，也可以指定时间戳，broker 会把时间戳解析为该时间点或之后的第一个 offset。我倾向于使用时间戳。offset 这个数字只对某个集群上某个 topic 的某个分区有意义，而时间戳是已知的信息：部署在 09:40 开始，所以回到 09:30。需要查阅所用 broker 的文档，了解它如何解析时间戳，因为它使用的并不总是生产者的时间或 broker 的追加时间。

步骤很短，但顺序很重要。

1. 停止该 group。重置一个正在运行的 group 的 offset，要么失败，要么被下一次提交覆盖，具体取决于 broker。
2. 在做任何更改之前，先记录当前的 offset。如果重置有误，这是唯一的回头路。

3. 选择目标时留出余量。回退的幅度要比认为需要的更早，因为重复消息代价低，而空缺代价高。
4. 重置到目标位置，重启该 group，并在消化过程中观察最老未处理消息的年龄（第 13 章）。
5. 持续观察，直到该 group 追回 head，然后再对账，因为重置只能修复日志仍然保留的部分。

留出余量之所以安全，只是因为有 inbox（第 7 章），它把对 group 已处理消息的重新读取变成空操作。如果一个 group 的消费者不是幂等的，那么无论步骤怎么写，都不应当重置它。

将重放限制在一个 consumer group 内

一个日志允许 10 个 group 读取同一个 topic，每个 group 有各自的 offset。正是这一特性使重置可以精确地定向。重置漏掉事件的那个 group，其他 group 毫无察觉。只重置漏掉了某些内容的最小 group 集合，并在运行之前把它们写进计划。Payments 对 `OrderPlaced` 的消费者保持原有位置，而 Inventory 的 group 重新读取最近一天的内容。

同一纪律的第二种形式，是为重放单独使用一个 group。如果必须重建某个投影，例如 Orders 的搜索视图，就启动一个名称不同的新 group，让它从头开始读取并写入一张全新的表，追平之后再把读取方切换过去。旧 group 从不移动，所以重建出错时没有什么需要撤销。这需要一段时间的存储空间和第二个消费者，而换来的是免费的回滚，代价很小。

抑制副作用

重放是把今天的代码应用于昨天的事实。如果某个处理器会发送邮件、调用支付服务商或通知承运商，重放就会让它再做一遍。第 12 章提出过这一点，其中给出的规则是出发点：通过重放重建的消费者必须知道自己正在重放。

人们的第一个念头是在消息里放一个标志，即一个写着“这是重放”的 header。这行不通，因为重放的消息就是原始消息。broker 保存的是生产者写入的字节，没有任何东西标明它们正在被第二次读取。模式属于消费者，而不属于消息。执行重放的 group 作为单独的部署运行，带有类似 `mode: replay` 的配置值，或者由副作用层检查 group 名称。

知道了模式之后，对副作用本身有四种选择，每一种都有代价。

- **在边缘用 gate 管控副作用**。每个外部调用都经过同一个函数，该函数检查模式，在重放时什么也不做。这是最清晰的设计，也最便于审计，但前提是没有任何处理器绕过这道 gate。新的处理器如果直接调用服务商，就会使它失效，所以这条规则需要测试（第 15 章）。
- **为外部调用加上幂等键**。如果扣款调用携带由 `event_id` 派生的键，服务商就能识别第二次尝试。第 7 章讲过这一点。它只在服务商还记得该键的期间内提供保护，所以它是第二道防线，不应当依靠它来应对上个月的重放。
- **暂停派生事件的 outbox**。被重放的处理器会在同一个事务中向自己的 outbox 写入新行，这些行会发往下一个服务。如果新事件带有全新的标识符，下游消费者会把它们当作新的事实。要么由输入的 `event_id` 和处理器名称确定性地派生出每个外发事件的标识符，使第二次运行产生相同的标识符，让下游的 inbox 将其吸收；要么扣住重放服务的 relay，直到检查过它产生了什么。
- **重放到另一个环境中**。对于没有副作用的投影，新表是最干净的答案，如上所述。对于有副作用的处理器，使用另一个部署，并把它的副作用层接到 stub 上，就可以在真正运行之前测试这次重放。

> **警告**：对一个会发送邮件、扣款或调用承运商的 group 执行重置，既没有 gate 也没有键，就会把每一个历史事件变成一次新的操作。如果连 inbox 一起重置，它也帮不上忙；而对外部副作用，它本来就无能为力，因为服务商不会随着自己的数据库一起回滚。要在移动 offset 之前确定抑制方法，而不是在收到第一个投诉之后。

去重窗口与 inbox

inbox 改变了重放的效果，而这种改变有利还是不利，取决于想要什么。

如果消费者的数据库完好，并且正在重新读取的范围处于其去重窗口（第 7 章）之内，inbox 会识别出每一个已处理过的 `event_id` 并跳过它。这正是出现空缺之后想要的：该 group 只处理缺失的部分，其余的开销很小。这也是重放可以修复从未见过某个事件的消费者、而不扰动见过的消费者的原因。

如果范围比窗口更旧，inbox 的行已经不在了。消费者会把旧事件第二次应用。对于受业务键约束保护的消费者，例如每次扣款只对应一条账本记录，第二次应用会无害地失败。对于没有这种约束的消费者，副作用会发生两次。所以去重窗口是恢复计划的一部分：在重放之前，要写明范围内

最老的事件、将会看到它的各个 group 中最短的窗口，以及两者没有重叠的地方由什么来保护消费者。第 7 章给出的规则是，窗口必须比计划中的任何重放都长。这就是计划。

inbox 也可能成为障碍。某个消费者带着 bug 写入了错误的状态。修复了 bug 再重放，每条消息都被跳过，因为每个 `event_id` 都在 inbox 里。这种情况稍后会讲到。

重放、修复与 redrive

有三种工具可以让消息再次移动，它们作用的对象不同。

工具	作用对象	章	适用场景
修复	少量记录，通过生产者的 outbox 以原始 `event_id` 重新发布	13	缺失或错误的条目很少，且生产者能说明是哪些
Redrive	死信，被移回拒绝过它们的消费者	8	消息明显失败且原因已经修复
重放	日志的某个范围，由一个 group 重新读取	14	大量键受到影响，数据仍在保留期限内，且消费者是幂等的

区分的问题是：谁知道哪里出了问题。修复时，由生产者或对账作业指出具体条目。redrive 时，是系统自己把消息搁置了起来。重放时，没有人手里有清单：损坏是一个时间范围，覆盖一个范围最便宜的办法就是把它整个重新读一遍，让 inbox 去分辨哪些已经做过。重放以精确性换取覆盖面。

保留期限是恢复的边界

重放只能回溯到日志保存数据的位置。保留期限这个很久以前出于磁盘成本考虑而选定的设置，因此也成了手头最便宜的恢复手段的极限。停机时间超过保留期限，或者在保留期限之后才发现 bug，都会让恢复从便宜的路径转向从生产者重新发布并对账。

我会依据团队发现并修复问题所需的时间来设定保留期限，而不只是依据磁盘。如果对账作业每天运行一次，而修复需要一天才能上线，那么一天的保留期限就什么也没留下。保留期限还会与成本方面相互作用：保留期限越长，重放可能遇到的旧 schema 越多（第 12 章），个人数据留在日志中的时间也越长。我不知道适合该系统的数字是多少。它取决于发现问题有多快、修复需要多久以及日志被允许保存什么。

没有失败却出错的消费者

第 3 章中有一行是关于没有错误的消费者 bug：处理器成功了，但写入的结果是错的。这是最需要重放的恢复场景，也是 inbox 碍事的场景。

设想 Ledger 对 `PaymentCaptured` 的消费者，一周内对某一种货币使用了错误的指数来换算金额。每条消息都已被 ack。什么都没有失败，inbox 里有每一个 `event_id`。对账（第 13 章）最终报告，Ledger 在该货币上的合计与 Payments 不一致。修复上线了，现在错误的行已经存在，必须更正。

把 topic 重放到同一个消费者中毫无作用，因为 inbox 说每个事件都已经完成。有两条诚实的路线，选择取决于 Ledger 是什么。

如果消费者的状态是派生的、可以丢弃的，例如投影，就重建它。启动一个新 group，写入一个 inbox 为空的新存储，从第一个受影响的事件开始重放，用对账与数据源进行核对，然后切换过去。错误的状态被丢弃，而不是被更正。

如果消费者的状态是一份不得改写的记录，例如 append-only 账本，重放仍然有用，但不是用来覆盖。以更正模式用修复后的代码读取受影响的范围。对每个事件，计算这条记录本应是什么，与扣除此前更正后已入账的金额进行比较，并为差额入账一条更正记录，其标识符由原始 `event_id` 和更正自身的名称派生，这样把更正运行两次就是空操作。账本记录自身标识符上的唯一约束承担了这项工作，所以更正不需要 inbox 行，循环把重复键视为“已更正”，继续处理下一个事件。

错误的记录作为历史保留下来。更正是新的事实，说明了原因。这比重建更慢、更明确，也是审计人员唯一接受的形式。

```
function correctLedgerRange(from_time, to_time, correction_name):
    for each event in readLog("payments.captured", from: from_time,
to: to_time):
        original = ledgerEntryFor(event.event_id)
        posted = original.amount_minor + sumCorrections(original.id)
        // the fixed logic
        should_be = buildEntry(event)
        if posted == should_be.amount_minor:
            continue
        correction_id = deriveId(event.event_id, correction_name)
        try:
            transaction:
                postLedgerEntry({
                    id: correction_id,         // unique: re-running
this correction fails
                    amount_minor: should_be.amount_minor - posted,
                    corrects: original.id
                })
            // commit point: the correction is durable
        on DuplicateKey(constraint: "ledger_entry_id"):
            // an earlier run already made this correction; skip it
            continue
```

共同的线索是：重放能够低成本地找出受影响的范围，而对每个事件如何处理，则取决于消费者自身的语义。

灾难场景

灾难是这样一种故障：丢失的不是一条消息，而是一个存储。梳理各种可能性的有用办法，是按哪个存储回到了过去来组织。每个场景都按第 3 章使用的模式写下来：什么坏了、如何表现、哪项保证依然成立。三者之中最后一项最有用，因为它指明了依靠的是什么，而在这一项下无从可写的流程，在该场景中就是暴露的。在讲场景之前，先说说目标是什么。

每个存储的目标

有两个数字描述一次恢复。**恢复点目标**（RPO）是能承受丢失多少数据，以时间计量：每六小时一次备份意味着 RPO 最多六小时。**恢复时间目标**（RTO）是能承受多长时间没有这项服务。两者都是决策，而常见的错误是为整个系统只定一个。

它们不同，是因为存储不同。针对本章涉及的存储，大致如下。

存储	丢失的代价	通常的恢复方式	决定因素
服务数据库	别处不存在的状态，除非某个事件携带了它	备份加日志传送	每个数据库各有 RPO，由数据的业务价值决定
outbox	尚未发布的事件	同一个数据库，所以 RPO 相同	丢失的任何一行，都是一个从未存在过的事件
inbox	对已处理之事的记忆	与副作用在同一个数据库	与它们一起回退（第 3 章）
broker	在途事件和保留的历史	复制，或从生产者重新发布	RPO 由复制决定，RTO 由消费者消化的速度决定
派生的投影	如果可以重建，则没有损失	重放	RTO 是重放时间，如果日志保存了全部内容则 RPO 为零
外部服务商	不受控制的副作用	它们的记录，以及对账	在两个目标之外

这张表的要点在于，被重放投影的 RTO 是一道消化计算题。如果日志保存了一个月的数据，而 group 每小时能处理一天的量，重建大约需要 30 小时，那就是它的 RTO（假设这些数字；实际的数字来自测量，如第 13 章所述）。一个从未测量过消化速率的团队，并不知道自己的 RTO。

我还会把目标与第 3 章的故障模型逐个流程地挂钩。扣款流程与通知流程的容忍度不同，而 Ledger 对空缺的容忍度接近于零。统一的目标最终对通知过于严格，对账本又过于宽松。

从备份恢复数据库

假设 Orders 的数据库恢复到六小时之前。这对本书的各套机制意味着什么？

outbox 随状态一起回退。这六小时内写入的行都没了，连同它们所宣告的订单，尽管 Payments 和 Inventory 可能已经根据这些事件采取了行动。在恢复点时尚未发布、但在旧时间线中后来才发布的行，重新变成未发布，所以 relay 会把它们第二次发布。这是无害的重复消息，因为 `event_id` 是相同的。

还有一个更尖锐的隐患。如果订单标识符来自序列，恢复后的序列被倒回，下一个订单会收到一个已经分配过的标识符，而在丢失的时间线里它属于另一个订单。Payments 现在持有一笔针对 `order-81724` 的扣款，它属于 Orders 已经遗忘的某位买家，而新的 `order-81724` 属于另一个

人。出于这个原因，我更倾向于为聚合使用不透明的随机标识符。如果必须保留序列，就在接受流量之前把它推进到曾经签发过的最大值之后，这个数字可以从下游服务读到。

数据库被恢复的消费者的 **inbox**，与副作用一起回退。第 3 章指出这是正确的：恢复点之前处理的消息仍在 inbox 中，它们的副作用也幸存下来，而恢复点之后的消息在两处都被遗忘。只有 inbox 与副作用位于同一个数据库时，这一特性才成立。

不会回退的是消费者的 **offset**，它存放在 broker 中。该 group 停在 head 的位置，越过了恢复后的数据库已经遗忘的六小时事件。除非有人把 group 往回移，否则消费者再也看不到它们，所以恢复计划要包括对该 group 的重置。

目标需要仔细选取，因为恢复点是数据库时钟上的一个时间，而时间戳重置针对的是日志中的事件时间。恢复的那一刻，该 group 并不在 head。它处于某个位置，而该位置上的事件是在一段时间之前产生的，相隔的时间最多是该 group 的消息年龄（第 13 章）。所以目标是该 group 当时所在位置的事件时间，再减去一个余量。如果只根据恢复点来设定目标，一个落后一小时的 group，就会跳过数据库回退时它尚未处理的那一小时。

要从消息年龄历史或记录下来的 offset 中读取位置，并把这份历史保存在即将恢复的数据库之外。有些团队把 offset 与副作用存放在同一个数据库的同一个事务中，使两者一起回退。这消除了手动步骤，同时把 offset 的记账从 broker 手中接了过来，这是需要权衡的成本。

外部副作用留在原处。Payments 在丢失的时间线里发给服务商的扣款，存在于服务商的记录中，而不在 Payments 的数据库中。这就是必须进行对账（第 13 章）的地方。把自己的记录与服务商的记录比较，修复差异。

系统重启的顺序很重要，值得写成一套步骤。

```
function resumeAfterRestore(service, restore_point):
    // 1. nothing moves while the stores disagree
    pauseConsumers(service)
    pauseRelay(service)

    // 2. compare before anything is published or applied
    findings = reconcileAgainstNeighbors(service, since:
restore_point)
    repairThroughNormalPath(findings)

    // 3. start from where each group was, not from when the incident
was
    for each group in consumerGroupsOf(service):
        position_time = eventTimeAtPosition(group, at: restore_point)
        resetToTimestamp(group, position_time - safetyMargin)

    // 4. re-reading derives events and commands again; their ids must
be
    //    deterministic, so the relay stays paused until they are
checked
    resumeConsumers(service)
    verifyDerivedOutboxRows(service)
    resumeRelay(service)

    // 5. a restore is not finished until reconciliation is quiet
    scheduleReconciliation(service, within: "one hour")
```

重点就在这些步骤。先暂停两侧，可以防止恢复后的服务让不一致的状态变得更糟；先对账再恢复，遵循的是第 3 章。重置把恢复变成了一个重放问题，只多了一点。

重新读取那丢失的几个小时的 Orders，会产生新的 outbox 行，其中包括一条 `CapturePayment` 命令，而全新的 `command_id` 在 Payments 的 inbox 看来是新的，可能导致重复扣款。所以派生事件的 id 来自输入的 `event_id`，如副作用一节所述。由两个事件触发的命令，如 `CapturePayment`，其 id 则取自业务键，例如 `order_id` 加命令类型（第 7 章），因为重放时两个事件中哪一个最后被读到可能会变。relay 保持扣住，直到有人检查过产生了什么。

丢失 broker 数据

broker 可能在不丢失集群的情况下丢失数据。在日志族中，落后的副本可能被提升为 leader，它所没有的消息就此消失。ack 级别和最小同步副本数，决定了生产者是否被告知这样一条消息是安全的（第 2 章）。

有些 broker 通过配置允许或禁止提升过期副本，我会查明所用 broker 的默认值并把它记下来。保留期限也可能悄悄造成同样的后果，如第 13 章所示。

可以看到的现象：消费者位置发生跳变或被报告为超出范围，topic 的末尾 offset 比之前更低，以及事后出现“缺失”类的对账发现项。broker 的 ack 是有边界的证据，这个边界就是当时存在的副本数。

幸存下来的是：outbox，前提是保留了这些行。发布后立即清除行的 relay，对自己发送过什么没有任何记忆。如果把已发布的行保留一段时间，比如几天，仅作示意，就可以把受影响时间窗口内的行标记为未发布，再次发送。重新发布保留原始的 `event_id`，所以收到过该事件的消费者会把第二份当作重复消息。这与第 13 章的修复是同一个步骤，只是应用于一个范围，同样带有扇出方面的告诫：比订阅者去重窗口更旧的事件，需要投递给漏掉它们的那一个消费者，而不是投递给所有消费者。

如果 outbox 的行也没了，下一个来源是生产者自己的记录：用业务记录中保存的 `event_id` 和 payload 哈希重建消息信封，哈希不一致的地方交给人工处理。如果两者都不存在，事件就无法恢复，对账会一直报告它。到了这一步，正确的问题是业务问题：缺失的事实是从另一个系统重建，还是接受其丢失并加以说明？设计时让这个问题很少被问到，正是把 outbox 的行保留一段时间所换来的。

丢失一个区域

区域丢失结合了前面两种情况，并增加了自己的问题。假定 Parcelly 在第二个区域运行一个备用集群，由持续复制 topic 的复制工具供数，而数据库是异步复制的。

首先要说明的特性是：**异步复制有 RPO**，它就是故障时刻的复制延迟。凡是已经写入主区域但尚未复制的内容，从备用集群的角度看都丢失了。这是对损失的度量，而且是一个每分钟都在变化的数字。同样值得针对既定上限对它设置告警，方式与第 13 章处理消息年龄相同：监视的就是自己的 RPO。

第二个特性是：**offset 不会随之迁移**。备用集群持有消息的副本，但副本有自己的 offset，是副本写入时分配的。消费者在主集群中已提交的位置，对备用集群没有任何意义。有些复制工具提供 offset 转换；我会查明所用工具保证了什么，而且不会设计依赖其精确性的恢复方案。

可移植的方案是，让备用集群上的每个 group，从主集群故障时它所到达位置的事件时间减去一个余量开始，由 inbox 吸收重叠部分。这与数据库恢复中的规则相同。落后 40 分钟的 group 从 40 分钟前开始。从未被复制的事件根本不在备用集群上，下面的 outbox 窗口会覆盖它们。

由于主集群自己的提交信息随区域一同丢失，消息年龄历史必须保存在区域之外。如果它也没了，就退回到消费者去重窗口内最老的那个时间点，并接受额外的重复消息。仅有告警阈值是不够的，因为告警在当时可能一直在触发。

第三个特性是：**故障转移时出现重复消息是必然的，而不是可能的**。比真实位置更早重启的消费者，会把事件处理两次。区域故障时正处于发布中途的 relay，可能从幸存的数据库再发布一次。

空缺则是另一面：主集群已 ack 而备用集群从未收到的事件。如果生产者的数据库幸存下来，并且比备用集群的 broker 更新，它就持有备用集群从未见过、却被标记为已发布的行。对这些行，要像 broker 丢失的情形那样扩大 outbox 的重新发布窗口：把最近一个复制延迟区间内的行加上一个余量标记为未发布，再次发送。复制延迟决定这个窗口的大小，而且只决定这个窗口。重复消息在消费者处没有成本。空缺的代价，则是试图保护的东西本身。

如果生产者的数据库本身随区域一起丢失，并从较旧的副本恢复，情况就归结为从备份恢复，并带着随之而来的被倒回的序列和被遗忘的订单。



从左往右阅读这张图。处理未知情况的两个步骤，即重新发布窗口和时间戳余量，都偏向于产生重复。随后的每一个步骤，都是为了让重复变得无害。

Broker 迁移

迁移是一场由自己安排的灾难。团队从一个 broker 迁移到另一个，可能出于成本、停止支持、托管服务或访问模式的变化，而这一迁移与上述灾难有相同的要素：一边是数据，另一边是新的空白之地，位置无法带过去，还有一段两个系统同时存活的时期。区别在于有时间谨慎行事，没有紧急情况迫使临场发挥。

本书的各种工具使迁移成为一项可控的变更。原因在于，其中好几项被有意放在了应用与 broker 之间。

把 relay 作为唯一的切换点

有了 transactional outbox（第 4 章），生产服务从不与 broker 通信。它写入自己的数据库，由 relay（第 5 章）把行搬运到 broker。所以选择哪个 broker，被编码在一个进程中。改变事件的去向，意味着改变 relay 的发布目标，生产者的代码不用变。这比遍及每个服务的变更要小，是第 4 章出于另一个理由推荐的模式所带来的附带好处。

relay 还掌管消息信封。它从 outbox 行取出 `event_id`，放进它发布的消息中。在新 broker 上，同一个事件有相同的 `event_id`，这种相等性正是迁移其余部分得以安全的原因。

双发布

最安全的第一步，是让 relay 把每一行同时发布到两个 broker。在一段时间内，每个事件既存在于旧 broker，现有的消费者在那里读取它，也存在于新 broker，那里暂时还没有人读取。

双发布本身就是一次双写，而第 4 章解释了为什么双写是危险的。在这里风险较小，因为 outbox 行是真相的来源，崩溃只会让这一行被重新发送。

设计上的问题是 relay 要等待什么。如果它在两次发布都成功之后才把行标记为已发布，那么一个无法访问的影子 broker 就会让行一直处于未发布状态，并使消费者所依赖的实时流程停摆，而这只是为了一个无人读取的 broker。所以要分别跟踪两个目标。当旧 broker 已有该行时，这一行就算已发布，这是第 5 章中的含义，另用第二个标记记录新 broker。按 `id` 顺序做一遍，把缺少第二个标记的行送入新 broker，这样可以保持每个键的顺序，而清除要等两者都完成。代价是多一列、多一遍处理，换来的是新 broker 的故障表现为比对中不断扩大的差距，而不是 Parcelly 的停机。

在这一阶段，旧 broker 仍然是消费者所信任的那个。新 broker 是影子，它唯一的任务是与原来的 broker 做比对。

镜像与 bridge

在 relay 中做双发布之外的另一种办法，是用工具在 broker 之间复制消息：日志到日志用镜像工具，日志到队列或队列到日志用 bridge 或 connector。这不需要改动 relay，还会带上迁移开始之前已有的历史，而双发布做不到这一点。

有两点需要提醒。镜像或 bridge 既是消费者又是生产者，所以它是 at-least-once 的，会产生重复消息，每一对消费者-生产者都是如此。而且它必须原样透传 `event_id`，无论是在消息体中还是在 header 中。自行生成标识符的 bridge，会把每个被复制的事件变成新事件，inbox 一个也认不出来。在依赖它之前，要用样本测试这一点。镜像是否保留分区、键和 header 以及跨分区时按什么顺序，各个工具不同，答案在它的文档中，而不在本书里。

Offset 无法转换

consumer group 在旧 broker 上的 offset，在新 broker 上毫无意义。即使在两个同类 broker 之间，副本也有自己的编号。在队列与日志之间，一侧根本没有 offset。

答案与区域丢失一节得出的相同：选一个安全的起点，并依靠幂等性。对每个 group，选择一个时间戳，使该 group 在旧 broker 上尚未处理完的每个事件，都是在它之后发布的。取切换时刻该 group 的 lag，换算成时间，再加上一个余量。让新 group 在新 broker 上从这个时间戳开始。它会重新读取旧 group 已经处理过的事件，而 inbox 会跳过它们。

去重窗口在这里的适用方式与重放相同：新 group 将重新读取的最老事件，必须处于每个消费者的窗口之内。

顺序与分区

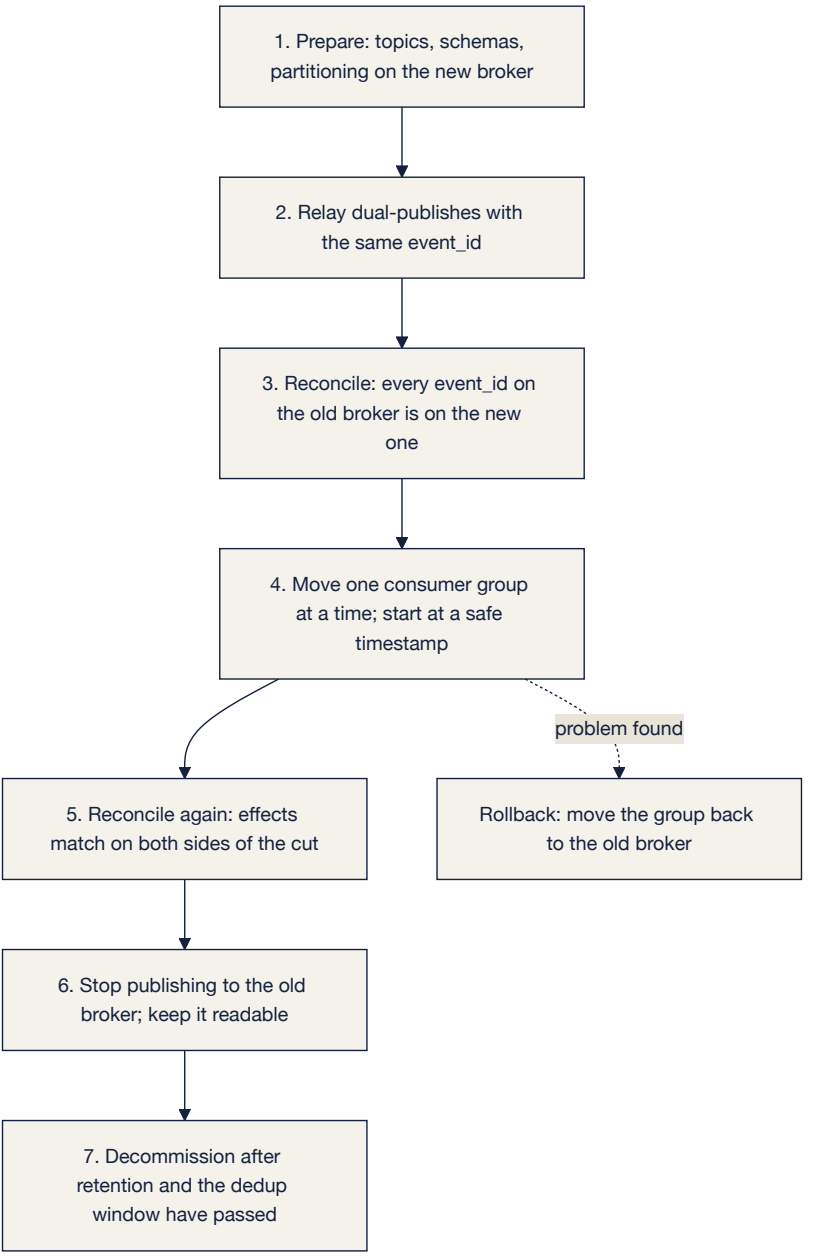
第 9 章给出了每个键保持顺序的条件：相同的键、相同的 partitioner、相同的分区数。迁移至少会改变这三者之一。新 broker 可能使用不同的 partitioner，于是同一个 `order_id` 落到一个它从未到过的分区上，而新 topic 创建时分区数也可能不同，因为这正是人们重新考虑这个数字的时机。

由此引出两件事。按第 9 章的建议，在 relay 中根据存储的 `partition_key` 计算分区，让一段代码掌管这一映射，并测试同一个键的事件在新 broker 中落在同一个分区上。再就是规划切换，使任何键都不会在两处同时有在途事件。如果消费者开始读取新 topic，而旧 topic

中仍保留着同一个 `order_id` 的未处理事件，那么新事件可能先于旧事件被处理：`OrderCancelled` 先于 `OrderConfirmed` 再次出现，这是第 9 章中的那种故障。避免的办法有两种。在启动新 group 之前把旧积压消化到零，这会让处理暂停与消化一样长的时间。或者让消费者在设计上具备容忍性，使用携带状态的事件和一个会拒绝任何更旧事件的聚合版本，如第 9 章所述。对于任何顺序重要且可以承受暂停的流程，我会先消化。

切换

操作的顺序是人们常问的部分：先动消费者，还是先动生产者？有了双发布，这个问题基本上消解了，因为在任何一个消费者移动之前，生产者就已经在向两边写入。顺序变成这样。



对账像括号一样框住这次迁移。在迁移之前，比较两个 broker 在一个窗口内的 `event_id` 集合：凡在旧 broker 上而不在新 broker 上的，都是双发布的缺陷，趁没有人依赖新 broker 时发现。迁移之后，比较副作用：14:00 迁移的 group，应当产生与它在旧 broker 上本应产生的相同状态，而第 13 章中跨服务的对账正是用来验证这一点的。仅仅因为消费者健康就宣布迁移完成，验证的只是没有任何东西崩溃。

一次迁移一个 consumer group，从最不关键的开始，比如 Notifications，以 Ledger 结束。每个 group 都是一个小的、可单独回退的步骤。

回滚

没有回滚计划的迁移是一扇单向门。有了这样的设计，计划几乎是免费的，只要有一个条件成立：旧 broker 持续收到每个事件。relay 在做双发布期间，旧 broker 是完整的，所以回滚意味着把 group 指回旧 broker，并带余量地指向某个时间戳。inbox 吸收该 group 在新位置处理过的事件。

这个条件就是第六步靠后的原因。一旦 relay 停止向旧 broker 写入，回滚就需要把流程反转，让 relay 再次写入旧 broker，并让旧 broker 的消费者从它停止的位置开始。这依然可行，因为 outbox 保存着这些行，只是更慢。回滚还受旧 broker 上保留期限和去重窗口的限制，所以我只会在自它收到最后一个事件以来这两者都已过去之后才下线它。在最后一个消费者迁移的当天就退役旧 broker，会在新 broker 的问题最可能出现的时刻去掉回滚的可能。

在队列与日志之间

当迁移跨越两个家族时，语义会改变，把它当作更换地址的计划是错误的。

方面	队列到日志	日志到队列
历史	获得：可以重放（保留期限也成为一项决策）	失去：没有重放，所以恢复依赖 outbox 和生产者
ack	逐条消息的 ack 变成 offset 提交；一条缓慢的消息现在可能阻塞它所在的分区（第 9 章）	offset 提交变成逐条消息的 ack，带有 visibility timeout 和重新投递
顺序	队列提供的组内顺序，变成每个分区的顺序；需要选择键	每个分区的顺序必须重新建立，例如在队列支持的情况下使用 message group
死信	Redrive 需要自己构建或配置；日志没有内置的死信队列（DLQ），只有自行创建的死信 topic（第 8 章）	出现队列的 DLQ 功能，其限制和保留期限有所不同
延迟与过期	延迟投递和逐条消息过期可能不存在，需要替代方案	延迟和过期可能是原生的；基于保留期限的恢复随之消失

每一行都是消费者所依赖的某种行为可能悄悄改变的地方。我会在选择目标之前，写下每个消费者所依赖的行为清单，并在演练中测试其中重要的几项，比如一条不得阻塞所在分区的 poison message（第 15 章）。

演练恢复

如果第一次运行是在压力之下，这一切都没有多大价值。从未做过的恢复只是一种信念。从未在真实规模下运行过的重放步骤，没有耗时可言。

有两种演练值得花成本。**恢复测试**取一份备份，把它恢复到临时环境中，检查恢复对 outbox、inbox 和 offset 做了什么，然后运行恢复后的继续步骤并为其计时。**game day** 在一个安全的地方有意移除某样东西，比如一个 broker、一个可用区、一个消费者数据库，让团队只凭 runbook 进行恢复。第 15 章讲到如何设计这些演练。理由属于这里：只有实际运行恢复，才能发现忘记了哪些步骤。

针对本章场景的 runbook 必须回答一个简短的问题清单，要写下来，而不是存在某个人的脑子里。第 16 章把它们放进了它的 runbook 指引中。

- 哪个存储受到影响，哪个回到了过去？
- 恢复点是什么，是如何确定的？
- 必须重置哪些 consumer group，重置到哪个时间戳，留多大余量？
- 重放期间必须抑制哪些副作用，通过什么机制？
- 每个要重新读取的范围，是否都处于每个受影响消费者的去重窗口之内？
- 暂停了什么，按什么顺序，由谁恢复？
- 哪些对账能确认恢复成功，由谁来看？
- 如果整个过程出了问题，如何撤销？

权衡与成本

保留期限是恢复预算。更长的保留期限让便宜的恢复手段保持更久，代价见前文所列。更短的保留期限会迫使从生产者重新发布。

保留 outbox 的行需要存储空间和清理作业。使重新发布窗口成为可能的那些行，正是更精简的设计会立即删除的行。使用变更数据捕获（CDC）并立即删除时，这个窗口就不存在（第 5 章）。如果想要这份

安全性，就把已发布的行保留一段既定的时间，并监控这张表。

重放是一件钝器。它重新读取范围内的一切，包括没有问题的部分，并且依赖每个消费者都是幂等的、每个副作用都受 gate 管控。对于少数已知的记录，修复更便宜也更安全。当损坏范围很广而且没有人手里有清单时，重放才有用武之地。

抑制副作用的是必须经过测试的代码。一道存在却被新处理器绕过的 gate，比没有更糟，因为它制造了信心。代价是一个长期存在的测试和代码评审中的一条规则。

带双发布的迁移成本翻倍。两个 broker 同时运行，relay 要做更多工作并跟踪两个目标，旧一侧的重复消息也会增多。作为回报，在有人依赖新 broker 之前就能得到验证，并且有回滚。如果系统能容忍一个维护窗口，直接切换更便宜：停止 relay，排空旧消费者，把 relay 指向新 broker，并在那里启动消费者。有了 outbox，这需要停机但不会丢数据，因为这些行会等着。它放弃了验证和回滚，我不会为账本选择这种方式。

何时不必构建这一切。对于空缺代价很低的流程，比如内部指标馈送，不需要重新发布窗口、runbook 或 game day。让恢复与故障的成本相匹配，并写下决定不予覆盖的流程。它们不在计划中，应当是一个决定。

常见错误

- **重置 group 之前不检查它的副作用**。之所以诱人，是因为重置只是一条命令，而数据就在那里。结果是第二轮邮件和第二次扣款尝试。在移动任何 offset 之前，先确定如何抑制副作用。
- **重置过多的 group**。让所有 group 重新开始，感觉很周全。每重置一个 group，就多一次对历史的重新读取，并带来其自身副作用的风险。只重置漏掉了某些内容的 group。
- **依赖重放 header**。之所以诱人，是因为消息上的一个标志看起来是最便宜的信号。重放的消息就是原始消息，不带任何被再次读取的迹象。模式属于消费者的部署或副作用层。
- **向带有 bug 的 inbox 的消费者重放**。消费者出现 bug 之后，inbox 会把每条消息都当作已完成而跳过。对于派生的状态，把它重建到全新的存储中；对于 append-only 的状态，则入账明确的更正。
- **整个系统只有一个恢复目标**。之所以诱人，是因为只需就一个数字达成一致。它对通知过于严格，对 Ledger 又过于宽松。按流程和存储

分别设定目标。

- **恢复数据库后立即恢复运行**。服务看起来健康，仪表盘一片绿色，于是恢复了流量。随后，恢复后的服务发布被倒回的状态，复用标识符，并漏掉了其 offset 跳过的事件。要先暂停、对账、重置 offset，然后再恢复。
- **恢复后复用基于序列的标识符**。序列是默认做法，在两个订单共用同一个标识符之前，看不出任何问题。被倒回的序列会分发一个已经属于另一个订单的标识符。使用不透明的标识符，或者把序列推进到曾经签发过的最大值之后。
- **信任 offset 转换**。之所以诱人，是因为工具宣称支持，而且能省去时间戳的计算。复制消息的复制工具，可能精确映射 offset，也可能不会。规划不依赖它的恢复方案，使用时间戳和 inbox。
- **生成新标识符的 bridge**。bridge 常被配置为用自己的信封包裹每条消息，这样配置在只有一个消费者时能通过测试。被复制的事件看起来都是新的，所以 inbox 什么也认不出来，每个重复消息都变成了副作用。原样透传 `event_id`，并测试它。
- **在最后一个消费者迁移时就下线旧 broker**。感觉迁移已经完成。这恰恰在最可能需要回滚的时候去掉了回滚。要等保留期限和去重窗口都过去。
- **从不恢复备份**。备份作业一片绿色。于是第一次恢复就是真正的恢复。要定期恢复到临时环境并为其计时。

要点

- 当存储丢失、被恢复或被替换时，留存下来的是围绕它的故障语义，即 outbox、inbox、稳定的 `event_id`、契约和对账，而 broker 本身是可以替换的。
- 重放是日志族的一种恢复工具：停止一个 consumer group，记录它的 offset，根据该 group 实际所在位置量出时间戳，留出余量，把它重置过去，由 inbox 吸收重复消息，前提是数据仍在保留期限之内。
- 重放的消息看起来与原始消息一样，所以执行重放的消费者必须知道自己的模式，用 gate 管控它的副作用，确定性地派生下游标识符或扣住它的 outbox，并且绝不依赖 header。
- 去重窗口是每个恢复计划的一部分，因为比窗口更旧的范围会重新放进重复消息；消费者 bug 则会让 inbox 从保护变成障碍，需要重建

或明确的更正。

- 恢复点目标和恢复时间目标属于每个存储和每个流程，而恢复后的数据库会倒回它的 outbox、inbox 和序列，offset 却仍停留在 broker 留下的位置，所以恢复后的继续步骤是先暂停、对账、重置 offset，然后才恢复。
- 跨区域的异步复制，其 RPO 等于它的延迟，且携带数据而不携带消费者 offset，所以故障转移时，每个 group 从其自身在故障时所在位置的事件时间开始，按复制延迟确定 outbox 重新发布窗口的大小，并把重复消息视为必然，把空缺视为真正的风险。
- broker 迁移把 relay 作为唯一的切换点，用相同的 `event_id` 做双发布，每次从一个安全的时间戳移动一个 consumer group，在前后都做对账，并让旧 broker 保持可读，直到不再需要回滚。

一个从未运行过的恢复步骤只是一个假设，而本应让系统挺过故障的代码也是如此。第 15 章讲述如何测试事件驱动系统，包括如何演练本章所描述的恢复、故障转移和重放。

第 15 章 — 测试事件驱动系统

设想 Parcelly 的 Inventory 团队，在一个长周末前的周五。他们的测试套件一片绿色：400 个测试，运行只要几分钟，覆盖率也没有人争论。`OrderPlaced` 处理器有一个测试，构造一个事件，调用处理器，检查库存减少了两件。它每次都能通过。这些测试使用的是 broker 的内存替身，它把每条消息只投递一次，按顺序，立即送达。

周六，一次部署重启了消费者。处理器在预热期间比平时慢，其中一个超出了 visibility timeout，于是 broker 在第一个消费者仍在处理时，把同一条 `OrderPlaced` 消息交给了第二个消费者。两者都读了 inbox，都没找到记录，于是都去预留库存。本该让其中一个失败的唯一约束，在真实的 schema 中缺失了，而没有任何测试发现，因为测试跑在一个手写的 fake 数据库上，它从不强制执行约束，也从未同时运行两个处理器。这套测试对它检查的任何事情都没有出过错。它从未检查那件至关重要的事。

与此同时，Orders 套件里有一个测试，每 20 次运行失败一次。团队已经不再理会它。它等待两秒钟让事件到达，而有时事件需要三秒。

这两个故事有同一个根源：测试描述的是一个并不存在的系统。本章回答一个实际的问题。对于最严重的 bug 只在重复、延迟和乱序下才出现的代码，如何测试，又不至于搭出一套缓慢、不稳定或被无视的套件？

问题及其难点

第 1 章论证过，快乐路径是不费力就能测试的路径，也是人们倾向于相信的路径。第 2 章至第 14 章随后构建了针对快乐路径从不显现的故障的防御。防御是代码，而从未在其所处理的故障下被检验过的代码，只是一种希望。第 3 章引入了故障模型，其中每个条目都有一个“验证方式”字段。本章就是填写这个字段的地方。

有三个特性使这种测试变得困难。

故障是交错执行，而不是输入。函数测试提供一个输入，检查一个输出。重复预留的 bug 需要特定的时序：两次投递在时间上重叠，或者一次提交之后跟着一次崩溃和一次重新投递。没有任何输入值能触发它。测试必须控制事情发生的时间，这意味着测试工具必须掌控 broker 的行为、时钟和崩溃点。

异步掩盖了关键时刻。处理器运行之后，副作用可能在几秒钟后才落在另一个服务里。测试无法立即断言，也无从知道该等多久。固定时长的 sleep 两头都不对：太短则测试不稳定，太长则套件缓慢。

Fake 恰恰在最关键的地方撒谎。幂等性依赖唯一约束。原子性依赖事务。顺序依赖分区分配。跳过这些的 fake 给出一套绿色的测试，却不提供任何信息。真实性的代价是速度，所以这门手艺在于决定在哪里付出这个代价。

调整测试金字塔

经典的测试金字塔依然成立：底部是大量快速的测试，顶部是少量缓慢的测试。变化的是每一层的用途，以及每一层负责捕获哪种故障。

层	运行什么	真实还是 fake	捕获什么
处理器测试	一个处理器对接真实数据库	真实数据库，没有 broker	错误的副作用、缺失的 inbox、被破坏的原子性、对重复消息和顺序的错误处理
契约测试	生产者与消费者之间的事件形态	完全没有服务	破坏消费者的 schema 变更
组件测试	一个服务对接真实的 broker、outbox 和 relay	真实 broker，其他服务以 stub 替代	连线：从 outbox 到 relay 再到 topic、ack、retry、转入死信队列
流程测试	几个服务一起运行，并注入一个故障	一切都是真实的，规模很小	跨服务的行为，如补偿和超时
生产环境 probe	在线上系统中的合成事件	真实	任何预生产环境都无法显现的部署和配置问题

表中有两个选择值得各用一句话说明。处理器这一层使用真实的数据库引擎，因为 inbox、outbox 写入和版本检查都依赖约束与事务，fake 必须重新实现它们，而重新实现它们正是 Inventory 的套件被蒙骗的原因。相反，在这一层把 broker 排除在外。处理器是一个函数，输入是一条消息和一个状态，输出是新的状态和一些外发消息。如果可以直接调用它，就可以快速测试它。

金字塔的形状也源自故障所在的位置。本书中的大多数防御都是消费者侧或生产者侧的逻辑，处理器这一层可以低成本地触达它们。只有连线需要 broker，而在那里测试数量很少，因为每个测试证明的是配置的一项属性，而不是业务逻辑的属性。

测试处理器

处理器测试应当用本书的语言陈述契约：给定这条消息和这个状态，副作用发生一次，且外发事件被写入 outbox。然后同一个测试再加上第二幕，即消息再次到达。

```
function testReservationIsIdempotent():
    db = freshDatabase()
    seedStock(db, "sku-17", available: 5)
    message = buildEvent("OrderPlaced", order_id: "order-1",
                         lines: [{ sku: "sku-17", qty: 2 }])

    handleOrderPlaced(db, message)
    // same event_id: the broker redelivered it
    handleOrderPlaced(db, message)

    assert stockAvailable(db, "sku-17") == 3
    assert reservationCount(db, "order-1") == 1
    assert outboxCount(db, "StockReserved", "order-1") == 1
```

最后一个断言是团队会忘记的。一条重复消息即使没有预留两次，却发出了两次 `StockReserved`，仍然会在下游造成损害，而检查 outbox 可以抓住这一点。

这个测试覆盖的是顺序出现的重复消息，由 inbox 中的快速路径检查来处理。危险的情形是并发的重复消息，即两个 worker 都通过了检查。要测试它，测试工具同时运行两个处理器，并用屏障在事务之前恰好把两者拦住，从而强制出现这种交错，而不是寄希望于它出现。

```
function testConcurrentDuplicatesCommitOnce():
    db = freshDatabase()
    seedStock(db, "sku-17", available: 5)
    message = buildEvent("OrderPlaced", order_id: "order-1",
                         lines: [{ sku: "sku-17", qty: 2 }])
    barrier = newBarrier(parties: 2)
    setHook("before-transaction", barrier.wait)

    results = runConcurrently(
        handleOrderPlaced(db, message),
        handleOrderPlaced(db, message))

    // one commits; the other fails on the unique constraint and
    // would be redelivered, which is allowed
    assert stockAvailable(db, "sku-17") == 3
    assert reservationCount(db, "order-1") == 1
    assert results.committed == 1
```

如果有人移除了唯一约束，转而依赖先读后插入的检查，这个测试就会失败，而这正是评审中最难发现的那种回归。第 7 章解释过为什么约束是安全网。这样的测试让这一解释变得可执行。

崩溃点的做法与此相同。如果处理器在本该 ack 的位置调用 `crashPoint("before-ack")`，测试就可以启用这个点，并在那里抛出一次模拟的崩溃。

```
function testCrashBetweenCommitAndAck():
    db = freshDatabase()
    seedStock(db, "sku-17", available: 5)
    message = buildEvent("OrderPlaced", order_id: "order-1",
                         lines: [{ sku: "sku-17", qty: 2 }])
    armCrash("before-ack")

    crashed = false
    try:
        handleOrderPlaced(db, message)
    on SimulatedCrash:
        // the process died after commit; the broker saw no ack
        crashed = true
        assert reservationCount(db, "order-1") == 1
    // if the crash point was renamed, the test must fail, not pass
    assert crashed

    // a crash point fires once per arming; disarm it for the
redelivery
    disarmCrash("before-ack")
    handleOrderPlaced(db, message)
    assert reservationCount(db, "order-1") == 1
    assert stockAvailable(db, "sku-17") == 3
```

请阅读 `on` 块内部的断言。它检查提交在崩溃之后仍然存在，这只是故事的一半。另一半是随后发生的重新投递。只检查后一半的崩溃测试，对一个从未提交过任何东西的处理器也会通过。

注入重复消息、乱序与延迟

单个测试编码的是已经想到的那些情形。故障模型中还包含没有想到的情形，所以我会增加一层测试，对整个事件流施加扰动，并检查结果不变。

这个思路是一种收敛性。取一个已记录的、顺序正确的某个聚合的事件流。直接对事件做折叠，不涉及 broker，计算出预期的最终状态。然后让同样的事件按受扰动的调度通过处理器投递，其中一部分在窗口内被打乱，一部分被重复，并断言最终状态等于预期状态。

```
function testOrderViewConvergesUnderAnyDelivery(seed):
    // versions 1 to 6, true order
    events = recordedOrderStream("order-1")
    // reference, no broker, no handler
    expected = foldEvents(events)
    random = randomSource(seed)
    // a failure must be reproducible
    log("seed", seed)

    schedule = shuffleWithin(events, random, window: 3)
    schedule = duplicateSome(schedule, random, rate: 0.3)

    db = freshDatabase()
    for each message in schedule:
        handleOrderEvent(db, message)

    // assumes held events are released when a gap closes (Chapter 9)
    assert heldEventCount(db, "order-1") == 0
    assert loadOrderView(db, "order-1") == expected
```

在 CI 中用几百个种子运行它。有两个细节让它保持诚实。种子会被记录，所以种子 4172 上的失败是可以在笔记本电脑上重新运行的失败，而不是一个传说。而且参照结果来自与处理器不同的代码路径，所以处理器中的 bug 不会同样出现在预期之中。

延迟是第三种扰动。延迟的消息是指在其后继消息之后才到达的消息，或在超时触发之后才到达的消息。这里要控制的是时间：直接读取系统时钟的处理器和 saga，如果不等待，就无法测试超时。我会把时钟作为依赖

传入，让测试去推进它。第 10 章列出了 saga 可能处于的各个状态。这样 saga 测试就变成一张表：对每个状态，投递每一个可能到达的事件，包括迟到的事件，并检查得到的状态和发出的补偿。

Saga 状态	到达的事件	预期结果
`AWAITING_RESULTS`	`PaymentAuthorized`	仍处于 `AWAITING_RESULTS`，等待库存
`AWAITING_RESULTS`，支付已持有	超时触发	进入 `COMPENSATING`；命令 `ReleaseStock` 和 `VoidAuthorization` 被写入 outbox
`COMPENSATING` 或 `CANCELLED`	迟到的 `StockReserved`	saga 不发生转换。迟到的 `StockReserved` 意味着 Inventory 在 `ReleaseStock` 到达之前就已预留，所以该命令会释放库存。如果释放先到达，它的 tombstone 就会拒绝这次预留（第 10 章）
`COMPLETED`	重复的 `PaymentCaptured`	忽略

第三行是关键的一行。超时之后迟到，是 saga 最容易出错的情形，因为快乐路径和超时路径各自单独测试过，而两者的碰撞从未被测试。这张表必须由追问“这个状态下还可能到达什么？”的人来写，而不是由沿着流程图推演的人来写。

> **注意**：如果测试依赖随机化的调度，一旦发现失败的种子，就把其中几个保留为固定的回归用例。曾经复现过一次的 bug，应当保持可复现。

契约测试与兼容性检查

处理器测试自己构建事件，这意味着它们测试的是消费者对生产者的想象。当生产者更改某个字段时，消费者的套件里没有任何东西会察觉。第 12 章介绍了兼容性规则和强制执行这些规则的 schema registry。这覆盖的是结构。契约测试覆盖的是 registry 看不到的部分：每个消费者实际依赖什么。

在**消费者驱动契约**中，每个消费者写下它从事件中读取的字段和类型，生产者的流水线检查其 schema 是否仍然提供这些字段。方向很重要。生产者无法知道哪些字段是承重的，而消费者知道。因此，契约是一个由消费者拥有的小文件，存放在生产者的 CI 能读取的地方。

```
contract inventory-reads-order-placed:
    consumer:        inventory
    event_type:      OrderPlaced
    schema_version:  1
    requires:
        payload.order_id        text, required
        payload.lines[].sku     text, required
        payload.lines[].qty     integer, greater than zero
    tolerates:       unknown fields
    sample:          minimalEventFrom(requires)

function verifyContracts(contracts, producerSchemas):
    for each contract in contracts:
        schema = producerSchemas.versionOf(contract.event_type,
                                           contract.schema_version)
        for each requirement in contract.requires:
            // optional is not enough: a consumer that reads the field
            // must be able to rely on it being there
            if not schema.guarantees(requirement):
                raise ContractBroken(contract.consumer, requirement)
```

需要两个部分。生产者的流水线针对备案的每一份契约运行 `verifyContracts`。消费者的流水线用自己真实的反序列化器解码其契约中的每个样本事件，并对它运行处理器。两者合在一起，既能捕获不再提供某个字段的生产者，也能捕获开始读取一个从未声明过的字段的消费者。后一种捕获只有在样本由契约构建、且只包含已声明的字段时才成立；从生产者的完整事件复制来的样本，会让消费者读取未声明的字段而依然通过。

样本应当覆盖保留期限内仍然存在的每个版本，如第 12 章所建议。重放读取的是现存的一切，只处理最新形态的消费者，会在第一个旧事件上失败。registry 的兼容性检查也属于 CI 中的这个位置：生产者的 schema 变更应当在注册之前就让构建失败，而不是在它抵达 topic 之后。

要诚实地面对局限。契约证明 `amount_minor` 存在且是整数。它不能证明这个字段的含义仍然是“以最小货币单位计的金额”。改变含义而不改变结构，会通过每一项契约测试，只有人工评审者或消费者中的行为测试才会察觉。我不会把绿色的契约当作关于含义的陈述。

最终一致的断言

端到端测试发布某样东西，然后对其他地方的副作用进行断言。副作用最终才会落地，所以断言需要一种能容忍这一点的形式。标准形式是带截止时间的轮询。

```
function eventually(check, timeout, interval):
    deadline = now() + timeout
    while true:
        observation = check()
        if observation.satisfied:
            return
        if now() >= deadline:
            // report what was last seen, not only that time ran out
            raise AssertionTimeout(observation.description)
        sleep(interval)
```

这个结构比看上去更重要。检查在到达截止时间之后还会再运行一次，所以在最后一次 sleep 期间满足的条件仍然能通过，间隔相对于超时来说很短，而失败消息带有最后一次观察的结果。一条写着“超时”的消息什么也说明不了。一条写着“order-1 停在 AWAITING_RESULTS，有 1 个暂扣事件”的消息，则指明了该去哪里查。

不稳定的异步测试成因不多，大多数都可以从成因上修复。

- **固定时长的 sleep**。用 `eventually` 替换。超时变成上限，条件一满足测试就结束。
- **按快速笔记本电脑调出来的超时**。把上限设得宽裕一些，因为条件很快满足时它没有任何成本，并且要把它与性能预期分开。延迟预算是另一个测试，环境也不同。
- **测试之间共享的状态**。共享 topic 中遗留的消息，会让下一个测试看到一个并非它发布的事件。为每个测试提供各自的 correlation ID 并据此过滤，或者提供各自的 topic 或队列。
- **在没有承诺顺序的地方假设了顺序**。如果测试对来自不同分区的两个事件的顺序做断言，它断言的就是 broker 从未承诺过的东西（第 9 章）。
- **断言“什么都没发生”**。“没有发送第二封邮件”无法通过轮询变成成功，因为等得更久可能仍会看到一封。宁可把系统驱动到静默期，再做断言。静默期是整条路径的属性，而不是某一个消费者的属性：outbox 为空，relay 空闲，所有相关消费者的 lag 为零，没有已调度的 retry 或定时器，retry 队列和延迟队列都是空的。到那时才不再有任何东西可能产生该副作用，包括消息仍在途中的下游一跳。

对于最后一种情况，测试工具需要一种方式来表达“一直运行，直到没有任何待处理的东西”。在处理器测试或组件测试中，测试工具可以不断驱动消费者，直到队列排空、且注入的时钟上没有剩余的定时器可触发。这

比等待挂钟时间既更快也更确定，也是我把大多数行为测试放在处理器这一层、只在连线是被测对象时才使用真实时间的主要原因。

故障注入与混沌实验

到目前为止的一切，都是在隔离状态下测试代码。下一个问题是，组装起来的系统是否像故障模型所声称的那样运转。故障注入是有意把一个故障放进运行中的环境，并检查系统的响应。

从第 3 章的故障模型出发，挑选的是条目，而不是工具。对每一条，用该条目自己的措辞写下假设：“如果消费者在提交与 ack 之间被杀掉，重新投递不会改变任何东西”。然后注入它：在那一刻杀掉进程、暂停 relay、延迟 broker 的响应、切断服务与其数据库之间的连接、在发布中途重启一个 broker 节点。事后检查的是不变量，而不是日志：库存与预留相符，账本中每笔扣款对应一条记录，没有 outbox 行被卡住。

把恢复演练加入这个清单，因为从未被演练过的防御只是一种希望。从未被恢复过的备份是经典的例子。把一份恢复到临时环境中，检查恢复对 outbox、inbox 和消费者 offset 做了什么，这些在第 14 章中讲过：outbox 的行复活，inbox 的行倒回导致重复消息不再被识别，offset 不再匹配。故障转移到备用集群，或对此进行演练，并把恢复时间和丢失的数据与第 14 章为该流程选定的恢复目标相比较。还要测试追平：暂停一个消费者足够长的时间以积累积压，在流量持续到达的同时恢复它，并检查积压是否在该流程的截止时间之前、且在保留期限过期之前清空。这检查的是第 13 章的消化速率和第 6 章对在途工作的限制。

我会分层运行，影响范围逐层增大。在专用环境中的流程测试，按计划在 CI 中注入故障，规模小且可重复。定期的 game day 在更大的环境中注入故障，作为团队演习，一部分是为了发现 bug，一部分是为了看看是否有人注意到。第 13 章的告警与代码一样处于被测之列。如果杀掉了一个消费者而没有告警触发，那就是在监控中发现了一个 bug。

生产环境中的混沌实验是可能的下一步，而不是起点。只有当系统在预生产环境中已被证明能挺过同样的故障、影响范围可以被限定、并且有人能立即停止实验时，它才有意义。不具备这些条件的团队不应从这里开始，在 game day 阶段停留很长时间，我也不会觉得难为情。故障注入实践的价值在于假设、注入、检查这一循环，而不在于注入有多戏剧化。

> **警告**：如果一次故障注入运行与真实流量共用同一个 broker、数据库或支付服务商账户，那它就是一场有计划的停机。在关闭任何东西之前，先隔离环境，或将注入限制在合成流量上。

作为回归测试的重放

事件日志是对实际发生之事的记录，而重放，即有意重新读取历史，把它变成了一套回归测试。取一段真实事件的切片，在临时环境中让新版本的消费者处理它，并将派生状态与旧版本产生的状态进行比较。差异要么是预期的，比如 bug 修复，要么就是回归。重放还有运维上的用途，即在出现空缺或恢复之后通过重新读取历史来恢复，第 14 章讲述了这一点。本节只讨论用它来测试。

这能捕获手写 fixture 永远抓不到的东西：三个 schema 版本之前的事件形态、没人想象过的事件序列、在测试中总是存在而在一小部分历史中为 null 的字段。它也是对第 12 章所称旧事件仍可解码这一说法最接近的测试。

一套可行的步骤有五步。

1. **选择切片**。一个时间窗口，或者一批样本聚合的完整历史。涵盖每个 schema 版本并包含已知事故的样本，胜过随机样本。
2. **脱敏**。重放的事件包含买家数据。在它们离开原有边界之前进行匿名化，或者让临时环境受到与生产环境相同的管控。
3. **禁用副作用和发出的事件**。执行重放的消费者不得扣款、发送邮件或调用承运商。副作用交给记录自己被要求做什么的 stub，这些 stub 的记录是输出的一部分。它的 outbox 和 relay 必须发布到临时 broker，或者被截获，绝不能指向生产环境的 topic，否则重放会把历史重新发布给线上的消费者。
4. **在同一个切片上运行新旧版本**。逐个聚合地比较派生状态和发出的事件。
5. **对每一处差异分类**。预期的、无害的，或是回归。没有人能解释的差异，在被证明并非如此之前就是回归。

重放适用于日志族，那里历史仍然存在。在经典队列上，已消费的消息就没有了。对应的做法是捕获样本：用一个 tap 把一部分消息经过匿名化复制到自己掌控的存储中。这是一种较弱的工具，因为捕获的质量取决于采样，但它服务于同一个目的。

重放也有成本上限。庞大的历史需要真实的时间和存储，而比较工具需要团队自己构建并保持可用。我会把完整历史的重放留给触及事件解释方式的变更，常规发布则使用较小的切片。

在生产环境中验证

没有任何预生产环境能复现生产环境的流量形态、数据、配置和怪癖。有些故障，比如部署之后 topic 上缺失的权限、过期的机密或错误路由的队列，只出现在真实系统中。答案不是放弃预生产测试，而是在故障所在之处增加最后一层。

合成事件是有意注入、明确标注的一份流量，它走过真实的流水线，并在末端接受检查。对 Parcelly 而言，这是一个 probe 每隔几分钟下的一笔订单，从 `OrderPlaced` 一路流转到 `ShipmentScheduled`。

```
function runOrderProbe():
    probeId = newId()
    // recorded first, so a crash of the probe itself leaves a trail
    recordProbe(probeId, startedAt: now())
    try:
        // goes through the real API, outbox, relay and broker; the
flag
        // travels in a header and is copied like correlation_id
        orderId = placeSyntheticOrder(probeId, headers: { synthetic:
"true" })
        eventually(
            check: flowReached(orderId, "ShipmentScheduled"),
            timeout: probeBudget,
            interval: probeInterval)
    finally:
        // runs on success, on timeout and on error alike
        cancelSyntheticOrder(probeId)
```

标注是安全机制，它是一个专用的 `synthetic` header，而不是 `correlation_id` 上的前缀。同时还控制行为的标识符是一个陷阱，而第 13 章要求使用统一的 id 格式 `order-81724`，使按 correlation 搜索在各处都有效。代价是，某一跳如果忘记复制该 header，就会让 probe 变成真实流量，而且不产生任何错误。每个服务都用复制 `correlation_id` 的同一个辅助函数来复制它，probe 还使用一个专用的合成买家账户，Payments 和 Shipping 各自独立地识别它，并且有一个测试会故意丢掉该 header，检查不会产生真实的扣款或发货。

每个服务识别该标志，并改变的是它的边缘，而不是核心逻辑：Payments 与服务商的沙箱对话，Shipping 跳过承运商交接，Notifications 抑制投递，Ledger 把账记到财务在报表中排除的测试账户。这些分支是只为 probe 运行的代码，所以其中的错误是只有生产环境才会暴露的 bug。Inventory 也需要同样的处理，否则每个 probe 都会预留真实库存：针对专用的合成 SKU 进行预留，或者对已标注的订单跳过预留。我会在处理器这一层测试这些分支本身，并通过故意破坏某些东西、检查 probe 是否失败来测试 probe。

清理还有一个陷阱。超时的 probe 不一定是没能完成；它的事件可能仍在途中，迟到的 `StockReserved` 或 `PaymentAuthorized` 可能在取消之后才到达。这就是为什么要预先记录 probe，并让每个服务都以 `synthetic` 标志为依据：定期清扫可以找到超过预算的 probe，并取消它们留下的一切，而取消本身是幂等的，所以运行两次也无害。

probe 还需要预算，预算需要负责人。失败的 probe 会呼叫某个人，所以预算必须足够宽松以避免误报，又足够严格以保持意义。第 13 章讲述了端到端延迟以及如何对其告警。probe 是这一度量最便宜的来源，因为它走遍整条路径，包括过去一小时内没有任何真实订单触及的部分。

生产环境验证包括第 13 章的对账作业。一次对账运行是对真实数据的不变量测试，持续执行，能捕获没有人想到去模拟的故障。我把它算作金字塔的最后一层。

权衡与成本

真实性要花时间。真实的数据库、真实的 broker 和注入的故障使测试变慢。那套很快的测试之所以快，是因为跳过了重要的东西。我会明确为各个缓慢的层次编列预算：处理器测试以秒计，组件测试以几分钟计，流程测试和故障注入按计划运行，而不是每次提交都运行。

契约测试增加协调工作。每个消费者拥有的文件，存放在生产者的流水线中，需要有人让它们保持最新。对于只有两个服务、一个团队的系统，对 schema 的一个共享测试可能就足够了。当生产者和消费者团队各自独立部署时，契约测试才开始值得，在此之前收益很小。

随机化调度增加调查工作。失败的种子既是一个谜题，也是一个真实的缺陷。不喜欢这一点的团队，最终会减少种子的数量，直到测试再也发现不了任何东西。解药是记录种子并保留固定的回归用例，而不是放弃这项技术。

混沌实验和 probe 有影响范围。两者都会触及线上或共享的系统。何时不要使用：系统尚不能在隔离状态下挺过这个故障时，没有人能立即停止运行时，或者已标注的流量无法与真实资金和真实买家隔开时。

重放的搭建成本很高。脱敏、副作用 stub 和比较所需的基础设施本身就是一个项目。对于历史很短的小系统，处理器测试和崩溃点能带来更多收益。我不知道盈亏平衡点在哪里，因为它取决于消费者变更的频率，以及对历史的错误解读代价有多高，但已经因为一个意外的旧事件而出过一次事故的团队，通常能猜到。

常见错误

- **针对从不出错的 broker fake 做测试**。之所以诱人，是因为它很快，而且最初的测试都能通过。让 fake 产生重复、延迟和乱序，或者像本章这样通过显式调用把故障放进处理器测试。
- **对数据库的约束使用 fake**。手写的 fake 不会强制执行唯一性、隔离或原子性，所以无论防御是否存在，测试都会通过。对任何依赖约束的测试，使用真实的引擎，放在容器或临时实例中。
- **sleep 固定的时长**。它看起来是最简单的等待方式。应当对条件进行轮询并设定上限，失败时报告最后一次观察的结果。
- **从不测试提交与 ack 之间的崩溃**。感觉它很罕见，然而每次部署都会发生。在每个处理器中加一个崩溃点，并写一个启用它的测试。
- **只测试最终状态，不测试 outbox**。没有改变状态的重复消息仍然可能发出第二个事件。对写入 outbox 的内容做断言。
- **契约测试只检查是否存在，不检查是否必需**。在生产者的 schema 中是可选的字段，可能缺失。让契约写明“必需”，并检查它。
- **没有种子的随机化测试**。无法复现的失败会被删掉。记录种子，并把失败的种子保留为固定用例。
- **信任从未恢复过的备份**。备份作业报告成功，于是感觉防御已经测试过。定期把它恢复到临时环境，事后检查 outbox、inbox 和 offset。
- **用真实资金运行 probe**。触及线上支付服务商或真实承运商的合成流量，就不是 probe。在边缘把已标注的流量路由到沙箱，并测试这一路由。

要点

- 假设 broker 把每条消息只按顺序投递一次的测试，测试的是一个并不存在的系统，所以本书中的每项防御都需要一个施加其所防御故障的测试。
- 处理器测试应当放在真实的数据库引擎上，因为 inbox、outbox 和版本检查依赖 fake 无法重现的约束与事务。
- 重复消息、乱序和延迟，最好作为受扰动事件流上的收敛性来测试，并记录种子，使任何失败都能被复现。
- 消费者驱动契约能在部署之前捕获结构性破坏，但看不到含义的变化，所以对字段本应表达什么的人工评审，仍然是流程的一部分。
- 用针对条件和上限的轮询替换固定时长的 sleep，对静默期而不是对沉默做断言，并注入时钟，使超时无需等待即可测试。
- 故障注入应当从故障模型的条目出发，事后检查不变量，只在系统证明它挺过每一步之后才扩大影响范围，并且应当包括针对恢复、故障转移和追平的恢复演练。
- 对真实历史的重放和生产环境中的合成 probe，能捕获手写 fixture 和预生产环境捕获不到的东西，它们需要脱敏、副作用 stub 和已标注的流量才能保证安全。

以这种方式测试过的系统可以多信任一些，尽管永远不能完全信任，这就留下了一个问题：团队在信任一个设计之前，应当检查什么。第 16 章把全书汇集为一份可靠性检查清单，并以检查清单无法做到的事情作结。

第 16 章 — 可靠性检查清单与结语

设想 Parcelly 在一个平静的周四举行设计评审。团队即将增加一个新流程：卖家可以提供部分退款，Payments 将为此发出一个新事件。图画得很干净。有人画好了箭头，给事件命名，并列出了消费者。会议预定 30 分钟，看起来 20 分钟就能结束。

接着，一位技术负责人问了一个朴素的问题：“如果退款已经发出，而事件始终没有发布，会怎样？”会议室安静下来，不是因为没人知道这个模式，而是因为没人问过。outbox 在代码库的别处已经存在。幂等的消费者在别处也已存在。这个新流程是否用上了它们则是另一回事，图上并没有说明。

可靠性通常就是这样被侵蚀的。团队了解这些模式。逐渐淡去的，是检查的习惯：对每个新流程，确认每个模式是否真的就位。因此，最后这一章要回答的是一个实际问题：为了弄清系统在糟糕的一天里能否信守承诺，应当逐个流程问些什么，又由谁来负责这些答案？

问题及其难点

可靠性是接缝的属性，而接缝在任何单个文件里都看不到。生产者可以是正确的，消费者也可以是正确的，二者之间的流程仍可能丢失事件，因为交接处从未被审视过。第 1 章在全书开头就提出了这一论点，此后各章逐一处理了这些接缝。

现在的难处在于，这些知识分散在 15 章之中。评审时没有人能把这么多内容记在脑子里。在截止时间的压力下，被跳过的问题，恰恰是那些答案为“在出事之前一直没问题”的问题：去重窗口、死信负责人、兼容性检查以及依据年龄而不是数量触发的告警。

清单是通行的应对办法，它也有人所共知的局限。它不会思考。不理解某一项的人照样可以勾选，填完的清单反而会带来虚假的信心。我会把下面的清单当作引发讨论的提示，而不是一张填表即可通过的关卡。每一项都指向解释其来由的章节，因此不同意的读者可以找到推理过程，并对前提提出质疑。

应当把每一项当作针对某个具体流程的问题，例如 Parcelly 的订单流程，而不是针对系统的泛泛之问。“我们使用 outbox”不是答案。“Orders 在写入订单行的同一个事务中，把 `OrderPlaced` 写入其 outbox”才是。

清单背后的方法

这份清单并不是新想法。它是本书论点在评审中的应用：可靠的事件驱动系统围绕**故障语义**构建，而不是围绕消息 broker，即针对某一接缝，弄清什么会失效、如何观察到、哪项保证能在失效后保留下来。

这就归结为三个问题，下面的每一项，都是针对某一接缝提出的其中一个问题。点出某种故障的条目，问的是什么会失效。点出某个指标、告警或测试的条目，问的是如何观察到。点出某项设置、约束或模式的条目，问的是哪项保证能在失效后保留下来。一个接缝若只有其中一个问题的条目有答案，就只做了一半；而第二个问题没有答案的接缝，会在无人知晓的情况下失效。

检查清单

清单按故障所在之处分组。右栏列出答案为“否”或“我不确定”时应当重读的章节。

生产者

检查项	章节
状态变更与事件在同一个本地事务中写入，业务路径中没有直接发布。	4
outbox 行携带完整的消息信封，`event_id` 在写入时创建，且从不重新生成。	4
团队已经明确写下：发布是 at-least-once 的，重复消息是为此付出的代价。	4
relay 先发布后标记，并按状态选取，而不是按递增的 id 游标选取。	5
若有多个 relay，它们能安全地认领行，并让同一个聚合的行由同一个 worker 处理。	5
对最旧的未发布行的年龄进行度量，且只清除已发布的行。	5

如果团队在一次评审中只能回答一个问题，我希望是第一个，因为后面的每一项保证都以事件确实存在为前提。如果某个流程不需要 outbox，因为它的各步骤可以共用一个本地事务，团队应当这样说明，并说明原因

（第 1 章）。

Broker

检查项	章节
对每个重要事件，团队已记录“已发布”的含义：生产者 ack 背后的复制与刷盘设置。	2, 3
ack 或 offset 提交发生在产生效果之后，因此语义是 at-least-once，而不是意外形成的 at-most-once。	2
visibility timeout、会话超时与处理器内的 retry 时间一并设定，使慢速工作在仍在运行时不会被重新投递。	2, 6
分区键标识聚合，团队清楚其所用的 broker 系列提供何种顺序保证。	9
每个生产者无论使用何种语言，都采用相同的 partitioner 和键编码，或者由 relay 自行计算分区。	9
保留期限超过最长的现实停机时间加上发现停机所需的时间。	3, 13
消费者在部署时优雅停止，因此 rebalance 不会成为重复消息的常规来源。	3

这些条目大多是设置而不是代码。它们一直保持默认值，直到一次事故让人看清默认值是什么，而到那时已很难追溯。

消费者

检查项	章节
inbox 行与业务变更在同一个本地事务中提交，并在键上设有唯一约束。	7
去重窗口长于保留期限、redrive 和计划内的重放，超出窗口的部分由领域表上的业务键约束覆盖。	7
外部副作用携带确定性的幂等键，该键由工作本身而不是某次尝试派生。	7
故障被分为瞬时、永久或未知三类，retry 采用带抖动的指数退避、截止时间和共享预算。	6
在途工作设有上限，依赖变慢时并发度随之下降，消费者选择暂停而不是失败，因此过载表现为 lag；入口处的限制会拒绝或推迟新工作，并向调用方给出明确的信号。	3, 6
poison message 最终进入 DLQ，DLQ 中的记录带有消息体、消息头、源位置、错误、尝试次数和消费者版本，并在对原消息发送 ack 之前发布。	8

检查项	章节
卡在某一条记录上的消费者能被检测到，依据是有 lag 而无吞吐量，以及按位置计数的崩溃次数，且不会一直循环到有人察觉为止。	8, 13
redrive 保留原始 `event_id`，以限速的批次运行，并设有暂停开关。	8
单个聚合内的顺序由按键通道、版本检查和带超时的暂扣缓冲区来保护。	9
pivot 之前的每个工作流步骤都有幂等的补偿，每次等待都有截止时间。	10

这是最长的一组，因为本书的大多数模式都落在这里。如果觉得沉重，那是准确的：at-least-once 投递把负担交给了消费者。

契约

检查项	章节
每个事件陈述一个事实，以过去时命名，并有一个所属服务。	11
集成事件是经过深思熟虑的公开契约，由领域事件映射而来，payload 中没有个人数据或机密。	11
命令以祈使式命名，有一个明确的接收方，携带 `command_id`，设有回复路径，使拒绝结果能够送达发送方，也从不称其为事件；发送方以自己的截止时间等待。	10, 11
`correlation_id` 沿流程原样传递，`causation_id` 指向直接触发者。	11
消息信封的 wire 类型，如时间戳、标识符和金额，对读写它的每种语言都已固定。	11
每项变更都归类为兼容变更或破坏性变更，并在部署流水线中检查兼容性。	12
含义变更由人工评审，因为没有工具能检测到它。	12
消费者忽略自己不使用的字段，容忍没见过的取值，转发消息的 hop 保留未知字段，对于 `schema_version` 不认识的事件则拒绝或暂存。	12
破坏性变更以一系列非破坏性步骤完成迁移，回滚方案写在同一份文档中，仍在读取旧版本的消费者是已知的。	11, 12

运维

检查项	章节
trace 上下文和 `correlation_id` 在每一个异步 hop 中都得以保留，因此可以凭标识符重建一笔订单的完整经过。	13
度量 consumer lag、消息年龄、outbox 年龄、retry 速率和 DLQ 新增消息，告警依据年龄和速率触发。	13

检查项	章节
对账作业按计划把权威来源与派生状态进行比对，修复在运行前经过评审。	13
直接检查业务不变量，例如“每笔已扣款的支付都有一条账本条目”。	13
处理器针对重复消息、乱序和延迟进行了测试，契约测试在 CI 中运行。	15
对最重要的流程，存在故障注入和基于重放的回归测试。	15
每个存储都有明确的恢复点目标（RPO）和恢复时间目标（RTO），团队清楚恢复时哪个存储会回到过去。	14
重放流程写明 consumer group、时间戳及其余量、所抑制的副作用以及每个被重放的范围都必须落入的去重窗口。	7, 14
broker 迁移方案以 relay 作为唯一的切换点，采用双发布并由消费者去重，并有切换和回滚步骤。	14
恢复和 game day 都已演练过，下文的 runbook 是根据演练所显示的情况写成的。	14, 15
书面的故障模型写明负责人、测试和评审日期。	3

设计评审问题

检查清单用来核对一个已经存在的设计。下面这些问题则帮助塑造一个尚不存在的设计。我会按这个顺序提问，因为每个答案都会约束下一个。

1. **事实是什么，谁拥有它**？如果无法用过去时给事件命名，也说不出唯一的所属服务，这个设计就还没有准备好交给 broker（第 11 章）。
2. **这里是否需要事件**？如果各步骤可以共用一个本地事务，或者调用方需要立即得到答复，同步调用可能是更合适的工具（第 1 章）。
3. **提交与发布之间会发生什么**？应当追问崩溃窗口，而不是追问模式。“我们在提交后立即发布”，就是一次等待部署到来的丢失（第 4 章）。
4. **这条消息到达两次会怎样，迟到又会怎样**？它们是不同的故障，修复方法也不同（第 7 章和第 9 章）。
5. **“最终”的界限是什么**？要为这段等待给出时长、告警和面向用户的状态（第 1 章）。
6. **这条消息永远无法成功时会怎样**？要问它最终去了哪里、谁能看到、又如何回来（第 8 章）。
7. **消费者或其背后的依赖比生产者慢时会怎样**？要问什么被设了上限、什么会暂停、哪些工作需要等待以及这一选择由谁负责（第 3 章和第

6 章）。

8. **后面的步骤失败时，什么来撤销前面的步骤**？要指明补偿，以及此后便无法撤销的 pivot（第 10 章）。
9. **一年后如何变更这份契约**？如果没人想过，答案就会在事故中临时拼凑出来（第 12 章）。
10. **如何知道它坏了，又如何知道它修好了**？检测与验证是两个独立的问题（第 13 章和第 15 章）。
11. **如果丢失了某个存储、broker 或某个区域，会损失多少，多久才能恢复**？要求每个存储给出 RPO 和 RTO，以及随后的重放或恢复（第 14 章）。
12. **谁会被呼叫，首先做什么**？如果答案是一个名字，就有了负责人。如果答案是“团队”，那可能还没有。

其中一些问题在某一天可能没有明确的答案，这没有关系。“我们还不知道，下面是我们将如何查清”，比一个没人核实过的自信答案更好。

组织层面

本书中的每个模式都有人的那一半。outbox 需要有人盯着最旧一行的年龄。DLQ 需要有人去读。schema registry 需要有人决定“兼容”意味着什么。人的那一半缺失时，模式仍然画在图上，却不再发挥作用。

责任归属

我会写下三种责任归属。

- **事件责任归属**。每个事件都有一个所属服务和团队，由其记录该事件的含义以及谁在消费它（第 11 章）。
- **运维责任归属**。每个 DLQ、parking lot、对账报告和告警都有指定的负责人。无人负责的告警只是一条通知，而通知会被忽略。
- **跨服务流程**。Parcelly 的订单流程跨越六个服务。即使没有任何单个服务负责，也必须有某个团队对整个流程负责，包括其超时和卡住的工作流（第 10 章）。

Runbook

runbook 是在假定读者疲惫、且已有六个月没见过这个系统的前提下写成的文档。对于每条告警，要写明它意味着什么、首先看什么、哪些操作是安全的、哪些不是以及前几步没有帮助时该找谁。

有些 runbook 与某条告警相关联，有些则对应极少发生的事件，少到团队里没有人最近做过。第 14 章补充了第二类。其中关于演练恢复的一节列出了恢复 runbook 必须回答的八个问题，要写下来，而不是存在某个人的脑子里：

1. 哪个存储受到了影响，哪个存储回到了过去？
2. 恢复点是什么，是如何确定的？
3. 哪些 consumer group 必须重置，重置到什么时间戳，余量是多少？
4. 重放期间必须抑制哪些副作用，靠什么机制？
5. 要重新读取的每个范围，是否都落在每个受影响消费者的去重窗口之内？
6. 暂停了什么，顺序如何，由谁恢复？
7. 哪些对账能确认恢复已经奏效，由谁来读？
8. 如果一切进展不顺，整个过程如何撤销？

下表列出我会为之编写 runbook 的事件。最后一列标出我预期每一项应当回答八个问题中的哪几个。它只是起点，团队自己的系统会增加自己的问题。

事件	危险操作	问题
消费者卡在 poison message 上（第 8 章）	不留记录就跳过该位置，或不断提高 retry 上限直到消息通过	6, 7, 8
lag 逼近保留期限的消费者（第 13 章和第 14 章）	任由消息未读就过期，或无人决定就向前重置	2, 3, 5, 7
offset 重置与重放（第 14 章）	在副作用仍然生效时重放，或重放超出去重窗口	3, 4, 5, 6, 8
redrive（第 8 章）	把整个 DLQ 一次性送回仍在艰难支撑的依赖	5, 6, 8
修复与 backfill（第 13 章）	对生产状态运行未经评审的脚本	4, 7, 8
从备份恢复数据库（第 14 章）	不重置 offset 就恢复消费者，或重新使用回退的序列再次分配出的标识符	1, 2, 3, 4, 5, 6, 7, 8
broker 数据丢失或故障转移（第 14 章）	在含义已与从前不同的位置上恢复 group	1, 2, 3, 5, 6, 7
区域撤离（第 14 章）	在不知道备用节点落后多远的情况下提升它	1, 2, 3, 6, 7, 8

事件	危险操作	问题
broker 迁移切换（第 14 章）	在双发布的副本完成去重之前就迁移消费者，或没有回退路径	3, 5, 6, 8

其中有两项需要格外小心。首先是 redrive，因为草率的 redrive 会把可控的故障变成洪水（第 8 章）。其次是修复与 backfill，因为修正派生状态的脚本就是一次生产变更，应当经过评审、dry run，并留下它所做操作的记录（第 13 章）。

值班

对事件驱动系统的值班与对请求–响应服务的值班，有一点重要的不同：故障往往是安静的。同步服务会返回错误，图表会变红。事件驱动系统可能什么也不返回，因为消息正躺在队列里等待。基于年龄、lag 和未处理工作的告警是首要信号，值班工程师需要足够的权限，无需等待许可就能查看队列、DLQ 和 outbox 的内部。

> **注意**：修复要在需要之前先练习。在白天、有同事在旁观看的情况下对测试 DLQ 做过一次 redrive 的团队，处理真正的 DLQ 时，会和只读过 runbook 的团队很不一样。第 14 章对恢复和 game day 提出了同样的观点，第 15 章讨论如何设计这些演练。

权衡与成本

检查清单有成本，首先是注意力。把一份含 43 项的清单用于每个流程太长了，把全部条目都用在低风险通知流程上的团队，会得出清单本身有问题的结论。我会按故障的代价给清单加权。涉及资金或改变库存的流程应当使用整份清单。提醒邮件只需幂等性和一个 DLQ 负责人，或许再无其他。

第二项成本是虚假的保证感。填完的清单说明有人考虑过每一项，而不是说明系统是正确的。第 13 章、第 14 章和第 15 章之所以存在，是因为经过考虑的设计仍会以其作者想象不到的方式失败。应当把检查清单视为最廉价的防御层，而不是最后一层。

第三项成本是漂移。只写一次的清单，会变成团队在某一时刻所相信之事的记录。每次事故之后都应当重新审视，并删去不再值得保留的条目。一份人们真正使用的短清单，比一份人们只是签字的长清单更有价值。

不适用的情形：不要用检查清单取代判断，也不要对用完即弃的原型提出这一要求。如果某个流程下个月就会被删除，完整清单的成本就超过了故障的成本。应当明确说出这一点，并写下该原型要么终止、要么接受正式评审的日期。

常见错误

- **把检查清单当作关卡**。数勾选数量就能审计关卡，这正是诱惑所在。结果是一张填好的表和一个没有讨论过的设计。应当把各项当作问题，并记录让某人感到意外的答案。
- **检查模式而不是检查流程**。“我们有 outbox”让人感觉有进展。应当追问这个流程是否用了它，并指明表和事务。
- **处处套用全部条目**。这让人觉得周全，随后却导致疲劳和弃用。应当按故障的代价给清单加权，并说明跳过哪些条目及原因。
- **把人的条目留到以后**。负责人、runbook 和评审日期看起来不如代码紧迫，而夜里告警响起时，缺的恰恰首先是它们。应当在设计评审当场就指派。
- **从不回头检查答案**。一年前通过评审的流程，此后已增加了消费者、版本和人员。应当为答案标上评审日期，就像第 3 章对故障模型的建议那样。
- **相信清单是完整的**。系统自身会有本书没有点名的故障模式。遇到时，应当把它们补进自己的清单。

要点

- 可靠性被侵蚀，更多是由于问题被跳过，而不是由于不了解模式；因此对每个新流程都提出一组固定问题，比现场有专家更能保护设计。
- 检查清单上的每一项，都是本书三个问题之一——什么会失效、如何观察到、哪项保证能在失效后保留下来——针对某一接缝的提问，因此每一项都应当针对具体流程回答，而不是针对泛泛的系统。
- 按故障的代价给清单加权：涉及资金或库存的流程应当满足每一项，低风险流程则满足那几项能防止重复副作用和静默丢失的条目。
- 本书中的每个模式都有人的那一半，因此负责人、runbook、告警接收人和评审日期应当与代码一起，在同一次设计评审中确定。
- 事件驱动的故障是安静的，因此值班依赖于对年龄、lag 和未处理工作的告警；而恢复、重放、区域丢失和 broker 迁移这些罕见事件，

需要真正跑过一遍的 runbook。

- 填完的检查清单说明有人想过每一项，并不说明系统是正确的，这正是对账和测试作为更多防御层而存在的原因。
- 可靠性不是一次交付的功能，而是需要持续践行的纪律，体现在小问题、书面答案和在需要之前演练过的修复之中。

Parcelly 永远不会像一个功能那样“完成”并变得可靠。每当有人问起消息到达两次会怎样，写下答案，并指定负责人，它就更可靠一分。没有哪个 broker 提供这一点，也没有哪个库附带它。这是一种注意力的习惯，由这样一些人保持：他们知道自己构建的每个系统，总有一天会被网络、一次部署或一位疲惫的同事误用，并且决定不大惊小怪地为那一天做好设计。这就是我希望读者从本书带走的东西：不是一套模式，而是继续追问下一个问题的习惯。

附录：术语表与模式速查

本附录分为两部分。A 部分是本书所用术语的术语表，按字母和拼音顺序排列，并注明各术语被定义或最充分使用的章节。B 部分是模式速查，以表格形式给出，便于在设计评审时扫读。

两部分都不能取代各章。这里的定义只是提醒，表中的一行只是指引。推理、成本以及某个模式不应使用的情形，都在最后一列所标明的章节中。

A. 术语表

ack（acknowledgment）。消费者向 broker 发出的信号，表示某条消息已处理完毕，可以从队列中移除，或消费者的位置可以前移。其否定形式为 **nack**。第 2 章。

at-least-once。一种投递语义：消息被投递一次或多次，因此可能出现重复消息。只要消息仍在保留期限内，且消费者在产生效果之后才 ack，该语义就成立。消费者落后超出保留期限时，仍会悄无声息地丢失消息。它是本书的默认语义。第 2 章给出定义；第 3 章讨论丢失问题。

at-most-once。一种投递语义：消息被投递零次或一次，因此消息可能丢失，但绝不会被投递两次。第 2 章。

backfill。批量修复或重建，把大批事件送入运行中的消费者，以小批量、限速的方式执行。第 13 章。

causation_id。信封字段，指向直接引起当前事件的那个事件或命令，而 `correlation_id` 标识整个流程。第 11 章。

circuit breaker。围绕依赖项的保护措施：失败超过阈值后停止调用，等待一段时间后放行试探性调用。第 6 章。

consumer group。共享一个 topic 的各个分区的一组消费者，使每个分区在同一时刻只由一个成员读取。第 2 章。

consumer lag。消费者落后于分区或队列 head 的程度，以消息数或时间衡量。第 2 章给出定义；第 3 章和第 13 章使用。

consumer outbox。由发布后续消息的消费者写入的 outbox，与它的 inbox 行和状态变更在同一个事务中写入。提交前崩溃不会留下任何东西，提交后崩溃则在重新投递时由 inbox 吸收。第 7 章引入这条规则；第 10 章把它应用于 saga 步骤。

correlation_id。同一业务流程中所有事件共享的信封字段。与之配套的 `causation_id` 指向直接引起当前事件的事件或命令。第 11 章。

deferred。针对一类失败的 retry 类别：该失败与消息本身无关，因为已知依赖项处于故障状态，例如 circuit breaker 处于打开状态。deferred 消息不消耗尝试次数，也绝不会因此转入死信队列。消费者暂停，消息原地等待。第 6 章。

event_id。标识单个事件的信封字段。它在写入 outbox 行时创建，永不重新生成，也是消费侧通常使用的幂等键。第 4 章和第 7 章。

exactly-once。本书中仅作为带限定条件的效果使用：at-least-once 投递加幂等消费者，本书也称之为 effectively-once。它不是投递保证。broker 的 exactly-once 处理在该 broker 的边界之内是真实的，到边界为止。第 2 章。

inbox。消费者一侧已处理消息 id 的表，与处理器的效果在同一个事务中写入。第 7 章。

offset。消费者在事件日志某个分区中的位置。第 2 章。

offset 重置。移动 consumer group 已提交的位置，通常带一段余量回退到某个时间戳，以便重新读取历史。offset 属于单个集群，无法转换到另一个集群。第 14 章。

offset 提交。在基于日志的 broker 上，消费者的一条记录，表明它已处理某个位置之前的全部内容。它取代逐条消息的 ack。第 2 章。

outbox。与状态变更在同一个本地事务中写入的表，relay 从中发布事件。完整术语是 **transactional outbox**。第 4 章。

parking lot。存放需要人工决策的死信的地方，与 DLQ 本身不同。第 8 章。

pivot 步骤。saga 中的一个点，在此之后工作流不再补偿，只向前推进。第 10 章。

poison message。每次处理都失败的消息。在基于日志的 broker 上，让进程崩溃的这类消息会使消费者卡在某一个位置，因此可通过“有 lag 而无吞吐量”以及每个位置一个计数器来发现。第 8 章。

rebalance。当成员加入或离开 consumer group，或某个成员被判定无响应时，对分区的重新分配。两个消费者可能短暂地处理同一个分区。第 3 章。

redrive。把死信交还给在其上失败的那个消费者，送回它自己的输入，绝不送回它们原先所在的共享 topic。保留 key 和 `event_id`，尝试次数重新计数，由 `redrive_count` 记录这一轮。第 8 章。

registry 修订版本。schema registry 为每个已注册 schema 分配的标识符。每次注册变更都会推进它，包括只增字段的变更，它与 **schema 版本**和**聚合版本**是不同的数字。第 12 章。

relay。把 outbox 行搬运到 broker 的进程。第 5 章。

retry 风暴。大量 retry 骤增，给本已吃力的依赖项增加负载并使其持续不可用。第 6 章。

retry 预算。把 retry 限制为整体流量的一定比例，而不是按单次调用限制。第 6 章。

RPO（recovery point objective）。能承受丢失多少数据，以时间衡量。每六小时一次备份，意味着 RPO 最长可达六小时，且每个存储各有各的 RPO。第 14 章。

RTO（recovery time objective）。服务能容忍中断多久。对于重放的投影，它是一次排空计算。第 14 章。

saga。由带补偿的本地事务构成的长时间运行的工作流，由协调者编排，或通过事件以编舞式协调完成。第 10 章。

schema registry。存储 schema、分配标识符，并将拟注册的 schema 与此前各版本作兼容性检查的服务。第 12 章。

schema 版本。信封中的整数 `schema_version`，第 12 章也称之为破坏性版本。只有旧读取方无法处理的变更才会推进它，它与 **registry 修订版本**和**聚合版本**是不同的数字。第 12 章。

schema 演进。随时间改变事件的形态，同时不破坏现有的读取方和写入方。第 12 章。

tolerant reader。只从事件中取自己需要的字段、忽略其余字段的消费者。第 12 章。

tombstone。消费者删除某个聚合的行之后保留的 key、最后版本和已删除标记，使迟到的事件无法把它重新创建出来。第 9 章。

topic。两类 broker 中任意一类里具名的消息流。第 2 章。

trace 上下文。把一个工作单元置于分布式 trace 中的 trace id 和 span id，写入 outbox 行并随消息头发送，使消费者的 span 能链接到生产者的 span。第 13 章。

upcaster。在业务逻辑看到事件之前，把按旧 schema 写入的事件转换为当前 schema 的代码。第 12 章。

visibility timeout。在基于队列的 broker 上，已投递消息在未被 ack 的情况下，被隐藏多久之后才重新投递。第 2 章。

版本检查。消费者把事件中的版本与它为该聚合最后应用的版本相比较，以忽略过期事件并发现缺口。第 9 章。

保留期限。broker 保存消息的时长，无论消息是否已被读取：日志上是时间或大小上限，队列上是过期时间。它限定了重放、redrive 以及落后的消费者最多能停机多久。第 2 章给出定义；第 3 章、第 13 章和第 14 章使用。

变更数据捕获（CDC）。读取数据库的事务日志，把已提交的变更转换为消息。在 outbox 场景中，基于 CDC 的 relay 从日志中读取 outbox 行，而不是轮询该表。第 5 章。

不变量检查。针对业务认定始终成立的规则而定时执行的查询，例如每笔已扣款的支付恰好对应一条账本条目。返回的每一行都是一处差异。第 13 章。

补偿。在语义上撤销 saga 先前某一步的步骤，例如释放库存或作废预授权。本书不称之为回滚。第 10 章。

重放。有意地重新读取事件日志中的历史。第 2 章给出定义；第 7 章、第 12 章、第 13 章、第 14 章和第 15 章使用。

重复消息。被处理或被投递多于一次的消息。第 2 章。

重新投递。broker 再次投递某条消息，例如在 ack 丢失或 visibility timeout 到期之后。重放是另一回事。第 2 章。

抖动。加在 retry 延迟上的随机性，使大量客户端不会同步 retry。“带抖动的指数退避”是标准说法。第 6 章。

队列。每条消息被单独消费并 ack 的 broker 结构。第 2 章。

队首阻塞。消费者不肯跳过的一条记录，会阻塞同一分区或同一流中排在它后面的所有记录。第 8 章。

对账。比较两个系统各自认定的真相，并修复差异。第 13 章。

反压。限制每个阶段的在途工作量，使负载保持有限。在经由 broker 中转的系统里，它表现为对消费者摄入量的限制，例如 prefetch 或每次轮询取回的记录数，以及由 broker 吸收的 lag。只有通过 broker 的流量控制或显式限流，压力才会传到生产者。第 3 章把它放在故障模型中；第 6 章给出机制。

分区。基于日志的 topic 中有序的单元，同一时刻由一个 group 中的一个消费者读取。第 2 章和第 9 章。

分区键。用于把事件路由到某个分区、从而保持每个聚合内顺序的聚合标识符。第 9 章。

故障模型。书面列出系统的各个接缝，并针对每个接缝写明什么会失效、如何观察到、哪项保证能保留下来，附有负责人、测试和评审日期。第 3 章。

故障语义。针对某一接缝：什么会失效、如何观察到、哪项保证能在失效后保留下来。这是本书的论点。前言提出它，第 3 章把它作为故障模型加以应用，第 16 章把它转化为评审检查清单。

故障转移。当前活动的 broker 分区副本、数据库或集群失效时，经设计切换到另一份副本。它可能丢失尚未复制的内容。第 3 章和第 14 章。

合成事件。由探针注入运行中流水线的带标记流量，在流程末端进行检查。第 15 章。

集成事件。作为有意设计的公开契约发布的事件，由领域事件映射而来，不含个人数据和机密。第 11 章。

截止时间。超过该时刻后结果不再有用。retry 必须在此之前停止。第 6 章。

镜像。用工具在 broker 之间复制消息：两个日志之间用镜像工具，日志与队列之间用桥接组件或 connector。它还会带上迁移之前的历史，并可让备用集群持续有数据，而副本上的 offset 是自成一套的。第 14 章。

竞争消费者。多个消费者共用一个队列，使每次投递只交给其中一个。重新投递的消息可能到达另一个消费者。它是 consumer group 在队列类 broker 中的对应物。第 2 章。

聚合。拥有分区键的一致性边界。同一聚合的事件彼此之间有序；不同聚合的事件则没有顺序。第 9 章。

聚合版本。对单个聚合的变更计数，存放在 outbox 行中并随 payload 携带，例如 Orders 的 `order_version`。它为同一聚合的事件排序，对 payload 的形态不作任何说明。它与 **schema 版本**和 **registry 修订版本**是不同的数字。第 4 章给出定义；第 9 章使用。

宽限期。记录必须达到的年龄，对账作业才会拿它比较，使仍在途中的事件不被报告为差异。第 13 章。

扩展与收缩。把一次破坏性变更拆成若干兼容步骤的迁移模式：同时支持两种形态，迁移所有读取方，然后移除旧形态。第 12 章。

链接（trace）。从消费者 span 指向生产者 trace 上下文的引用，在投递延迟、重复或批量时用来代替父子关系。第 13 章。

幂等键。消费者或 API 用来识别重复工作的键。对事件消费而言，通常是 `event_id`。第 7 章。

幂等消费者。对同一条消息重复处理与只处理一次效果相同的消费者。第 7 章。

命令。要求做某事的请求，用祈使式命名，例如 `CapturePayment`。命令发给某一个服务，可以被拒绝。它不是事件。第 11 章。

胖事件。携带足够状态、使消费者无需回调生产者即可行动的事件。与之相对的是**瘦事件**。第 11 章。

切换。broker 迁移中的一步：某个 consumer group 开始从新 broker 读取。在两个 broker 完成对账之后、并预先规划好回滚的前提下，逐个 consumer group 进行。第 14 章。

去重窗口。消费者记住已处理 id 的时间段。inbox 行已被删除的消息可能被再次处理。第 7 章。

生产者。发布消息的进程或服务，也是其发出的事件的所有者。第 11 章。

事件。内容为已经发生的事实的消息，以过去时命名。第 11 章。

事件日志。持久的、append-only 的事件序列。第 2 章。

事件日志簿。由服务保存在 inbox 旁边的 append-only 记录，记录事件信封而不含 payload，使 broker 忘记这些事件之后仍能回答相关问题。第 13 章。

双发布。迁移期间以两种形式或向两个目的地发布同一事实：schema 变更期间同时写两个字段或两个 topic（第 12 章），broker 迁移期间同时写两个 broker（第 14 章）。在 broker 迁移中，relay 为新目标另设一个标记，使旧 broker 保持完整，新 broker 可以与之对账。

双写。在没有共享事务的情况下更新两个系统，例如数据库和 broker。第 4 章。

死信。因处理失败而被移到一旁的消息。作为修饰语时该词带连字符：dead-letter queue。第 8 章。

死信队列（DLQ）。消费者或 broker 把已放弃处理的消息放入的独立目的地。在基于日志的 broker 上，同一概念称为**死信 topic**。第 8 章。

向后兼容。使用新 schema 的消费者可以读取以旧 schema 写入的事件：新读取方，旧数据。第 12 章。

向前兼容。使用旧 schema 的消费者可以读取以新 schema 写入的事件：旧读取方，新数据。第 12 章。

消费者。从 topic 或队列读取消息并据此执行操作的进程。第 2 章。

消费者驱动契约。由消费者持有的文件，列出它从事件中读取的字段和类型，并在生产者的流水线中检查。第 15 章。

消息。通过 broker 发送的数据单元：事件、命令或文档。第 11 章。

消息 broker。在服务之间传递消息的基础设施。本书涉及两类：基于日志的和基于队列的。第 2 章。

消息年龄。从仍在等待的最旧消息被生产出来起算的时间，用来与业务截止时间和保留期限比较。第 13 章。

消息信封。随每个事件携带的标准字段集合，例如 `event_id`、`event_type`、`schema_version`、`occurred_at`、`correlation_id` 和 `causation_id`，与 payload 分开。第 11 章。

业务键。由工作的含义派生的标识符，例如支付 id 加一个操作，而不是由信封派生。它防止同一项工作以两条不同的消息到达。第 7 章。

语义锁。数据中的一个状态标志，表示某个 saga 正在处理一条记录，其他代码按约定遵守它。它不是数据库锁。第 10 章。

暂扣缓冲区。一张持久化的表，消费者用它暂扣前序事件尚未到达的事件，并在缺口补上或超时到期时释放。第 9 章。

最终一致性。若更新停止，各副本或各服务会收敛到相同状态，但收敛耗时没有上界。第 1 章。

B. 模式速查

每一行都用本书的术语写出该模式所应对的故障，以及它要付出的代价。模式只是把故障移到可以处理的地方，并不会消除故障。第 16 章把这些行转化为评审检查清单。

模式	解决的故障	成本	章节
transactional outbox	数据库与 broker 之间的双写	多一张表、要运行一个 relay、把重复消息推给消费者	4
轮询 relay	以很少的基础设施把 outbox 行搬到 broker	轮询延迟、数据库负载、需要审慎设计游标	5
变更数据捕获	以低延迟从数据库日志发布 outbox 行	运维负担、connector 故障转移、需要管理日志保留	5
带抖动的指数退避	retry 使正在恢复的依赖项过载	增加延迟，需要调整延迟和尝试次数上限	6
retry 预算	retry 放大故障	部分本可能成功的调用会快速失败	6
反压与流量控制	依赖项或消费者收到的工作超出其承受能力	lag、被暂停的消费者和被拒绝的工作以及“什么该等待”这一业务决策	6
circuit breaker	对失效依赖项的反复调用	需要调整的状态；breaker 打开期间调用被拒绝	6
retry topic 或延迟队列	retry 延迟过长，无法在处理器内等待	更多 topic、更多活动部件、顺序更难保证	6

模式	解决的故障	成本	章节
带 inbox 的幂等消费者	at-least-once 投递带来的重复消息	每条消息都要写表，需要管理保留	7
业务键	同一项工作以不同消息到达	每个操作的键设计，以及唯一约束	7
向下游传递的幂等键	外部副作用重复	需要下游服务的支持	7
死信队列	poison message 卡在重新投递循环中	需要有人负责的一堆消息、告警和修复路径	8
parking lot	需要人工决策的死信	人工工作和明确的归属规则	8
redrive	原因修复后恢复死信	重复消息、重新排序以及主流程上的负载	8
consumer outbox	消费者变更状态并发布后续消息，却丢失了两者之一	消费者一侧的 outbox，以及要运行的 relay	7、10
每个聚合一个分区键	同一聚合内事件乱序	热点分区、并发度受限	9
消费者侧的版本检查	过期或乱序事件覆盖较新的状态	一个版本字段，以及对每个缺口的处理决策	9
暂扣缓冲区	必须按顺序应用的序列中出现缺口	内存、超时以及永远补不上的缺口	9
saga（编排式或编舞式）	跨服务、没有单一事务的业务工作流	需要设计和运维的状态、超时与补偿	10
补偿	后续失败后撤销已完成的 saga 步骤	有些步骤无法撤销；pivot 步骤标出了界限	10
语义锁	对 saga 正在处理的记录的并发变更	saga 运行期间操作被阻塞	10
跨语言按字段固定 wire 类型	同一信封字段被两个运行时读成不同的值	每种语言都要自律，并需要跨语言测试	11
标准事件信封	工具和消费者无法追踪或路由的事件	每个生产者都要自律	11
瘦事件或胖事件	在回调与携带状态之间选择	瘦事件：读取时产生耦合。胖事件：契约更大，数据被复制	11
带兼容性检查的 schema registry	生产者悄悄破坏消费者	要运行一项服务，要遵循一套流程	12
tolerant reader 与 upcaster	读取方遇到新字段或旧版本而出错	每个历史版本都要保留转换代码	12

模式	解决的故障	成本	章节
扩展与收缩	无法避免的破坏性变更	多次部署、双发布、更长的迁移	12
trace 上下文与 correlation id	在异步跳转间丢失流程的来龙去脉	每个生产者和消费者都要传播	13
对账与修复	源系统与派生状态之间悄无声息的偏离	比较作业、修复工具和结果评审	13
事件日志簿	关于 broker 已经遗忘的事件的疑问	每条消息都要写一张表，且需要管理保留	13
backfill	必须根据现有数据重建或补全的派生状态	限速的批次和一个暂停开关，以及运行中消费者承受的负载	13
不变量检查	没有任何技术指标能显示的业务层面破坏	定义不变量，并负责对应的告警	13
offset 重置式重放	消费者出错，或存储回到了过去，需要再次读取历史	覆盖该范围的去重窗口、被抑制的副作用以及作为上限的保留期限	14
每个存储的 RPO 和 RTO	灾难靠临场应对，没有约定的损失或停机时间	每个存储都要有一个需要度量和演练的数字	14
用于恢复的跨集群复制	失去 broker 集群或某个区域	要运行的备用节点、就是 RPO 的复制 lag 以及无法转换的 offset	14
双发布与切换	迁移到另一个 broker 时不丢事件、不重复效果	outbox 中的第二个标记、对账以及更慢的迁移	14
消费者驱动契约测试	团队之间的契约漂移	测试维护，以及对由谁运行测试达成一致	15
处理器测试与扰动测试	处理器从未在重复、延迟或乱序下被检验	真实数据库的测试基础设施，以及更慢的测试套件	15
故障注入	故障模型中的条目从未在运行的系统里被检验	专用环境，以及需要限定的影响范围	15
基于重放的回归测试	在真实历史上破坏行为的变更	需要维护经过筛选、匿名化的样本数据	15
生产环境中的合成事件	运行路径中的静默故障	测试数据必须与真实业务数据隔离	15

如何阅读此表

许多行成对出现：一个模式修复一种故障，另一个模式修复前一个模式引入的故障。outbox 会产生重复消息，所以消费者需要 inbox。retry 会产生负载，所以需要预算。死信队列会形成堆积，所以需要负责人和 redrive 流程。

这条链条就是本书的形状。如果采用了某一行却跳过与它配对的那一行，故障只是被转移，并没有消失。在我看来，这张表最适合从右向左读：从系统今天面对的故障出发，找到点名该故障的那一行，先读成本列，再读相应章节。

关于本书

可靠的事件驱动系统围绕故障语义构建，而不是围绕消息 broker。

对于每一个接缝，即一个组件把工作交给另一个组件的地方，本书都问同样的三个问题：什么可能失败，如何观察到失败，以及哪项保证能够在失败之后保留下来。At-least-once 投递被视为事实，而不是缺陷，因此处理器要做到幂等，重复消息、丢失的消息、retry 和乱序到达都在设计中加以应对，而不是寄希望于它们不发生。本书以 Parcelly 为线索，逐一讲解 transactional outbox、relay 与变更数据捕获、retry、幂等性、死信、消息顺序、saga、schema 演进、可观测性与恢复，并说明每一项所要付出的成本。

关于作者

Muhammet Şafak 自 2008 年起从事软件开发，多年来一直在医疗、金融和电信领域一线任职。Muhammet 主张开发应当聚焦于问题，而不是技术。自 2024 年起，Muhammet 设计并构建可端到端扩展的高流量系统。

Muhammet 为中小型软件开发团队提供指导，主导开源项目，并把每一项决策连同其理由与代价一并写下来。

可靠的系统并非由可靠的部件构成，而是围绕会失效的部件构建而成。

I 访问

II 事件

III AGENT