Week03｜课时 3｜增量与 CDC：cursor、WAL、乱序、去重与“不要轻易承诺 exactly-once”

Table of contents

先把增量 ingest 的边界讲清，再谈更复杂的 CDC 语义

这一讲开始进入 Week03 最容易被误判的部分：

很多团队不是不会做全量，而是把“看起来像增量”误当成“已经有了稳定增量链路”。

如果 cursor 推进方式、checkpoint 记录、迟到数据边界和去重策略没有讲清，系统迟早会在重跑、补数和回放上失控。

进入课时 4 回看课时 2 返回 Week03 总览

下载讲义

提供适合离线阅读的 PDF 版和适合批注整理的 Word 版。

PDF 版 · 打印 / 离线阅读 Word 版 · 批注 / 二次整理

这节课解决什么问题

课时 1 你已经建立了一个很重要的判断：

Week03 的重点不是“能把数据搬上来”，而是让 ingestion 可重复、可追踪、可恢复。

课时 2 又把这个判断进一步压成了 batch 主链路：manifest、批次边界、幂等写入、完整性校验。

但只要你的系统开始从“整批导入”走向“持续更新”，问题就会立刻升级：

- 哪个字段才配当增量游标?
- 为什么 updated_at 看起来简单，实际上最容易把你带沟里?
- WAL / logical decoding / CDC 到底解决了什么，没解决什么?
- 为什么很多人一提流式，就会过早承诺 exactly-once?
- 迟到数据、重复数据、乱序事件和崩溃恢复，应该怎么设计边界?

所以这节课真正要解决的是：

如何把 Week03 从“批处理基线”推进到“增量 / CDC 语义基线”，并且不在术语上过度承诺。

参考学习时间（55–65 分钟）

如果你只阅读正文，大约需要 35–40 分钟；如果你跟着本课一起把 cursor / watermark / checkpoint / dedupe / idempotency 的边界写进 incremental_ingest_strategy_v1.md 和 checkpoint_state_v1.md，建议预留 55–65 分钟。

学完这一讲，你应该能做到什么

完成这一讲后，你应该能：

1. 解释为什么增量 ingest 比 batch ingest 更容易出错。
2. 分清 cursor、watermark、checkpoint 三个概念，不再混用。
3. 理解 PostgreSQL logical replication / logical decoding / replication slot 在 CDC 中扮演什么角色。
4. 理解为什么当前课程主线不应该轻易承诺 exactly-once，而应优先落地 at-least-once + idempotent write + dedupe + replayable recovery。
5. 基于 OmniSupport Copilot 当前 repo，写出一份可直接进入后续实现的《增量采集策略说明》与《checkpoint 设计草案》。

本课产出

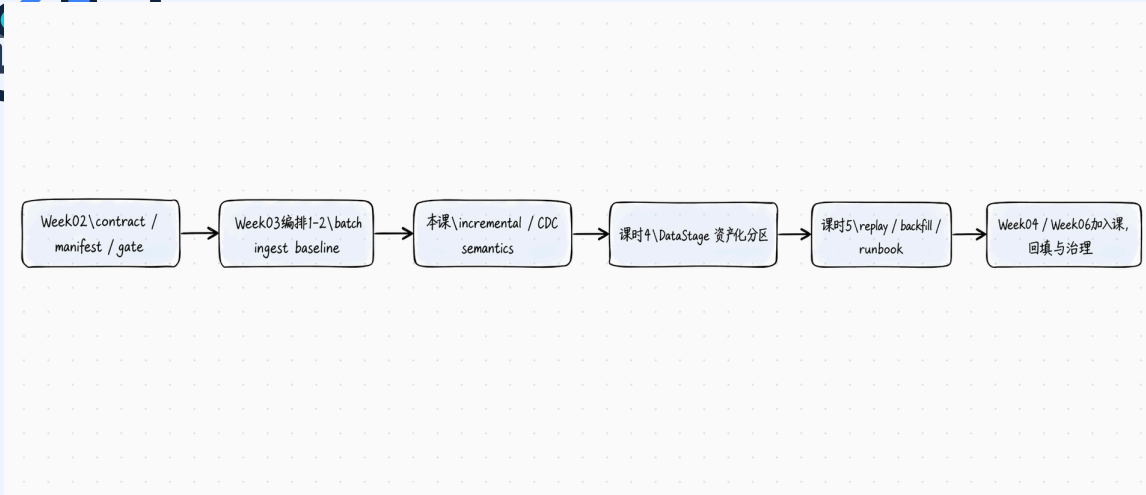
学完这一讲后，你至少应该在仓库里新增或补齐这 3 个文件：

- docs/blueprints/week03/incremental_ingest_strategy_v1.md
- docs/blueprints/week03/checkpoint_state_v1.md
- reports/week03/late_arrival_decision_table.csv

这 3 个文件的作用分别是：

文件	作用
incremental_ingest_strategy_v1.md	说明哪些 source 该走 incremental，游标如何选，哪些场景仍保留 batch / snapshot
checkpoint_state_v1.md	定义 checkpoint / watermark / replay 边界，避免后续实现时各写各的
late_arrival_decision_table.csv	明确迟到、重复、乱序、修复、补数各类事件的处理动作

先看一张总图



如果把 Week03 看成一条连续的工程链路，本课的作用不是“多讲一点流式概念”，而是：

把 batch 基线推进到增量与 CDC 的边界判断。

1. 为什么增量 ingest 比全量更难

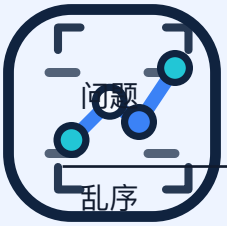
全量 ingest 的边界通常比较简单：

- 输入范围是全部
- 失败后重跑逻辑直观
- 很多问题都可以粗暴覆盖

增量 ingest 则必须同时处理：

- 新批次进入
- 重复事件进入
- 迟到事件进入
- checkpoint 继续前进
- 某一段历史窗口需要 replay / backfill

问题	Batch 里通常怎么处理	Incremental / CDC 里为什么更难
重复	失败就整批重跑或整批覆盖	同一条记录可能跨多次同步反复出现
迟到	重新补一批即可	迟到数据可能落到已关闭窗口之后



问题	Batch 里通常怎么处理	Incremental / CDC 里为什么更难
乱序	批次内部还能整体排序	流里不同 source / topic / slot 的顺序不一定等于业务顺序
恢复	重跑这一批	恢复点、LSN、cursor、checkpoint 需要被持续记录

Airbyte 在官方文档里把增量同步定义得非常清楚：增量同步通常依赖一个 cursor 字段，很常见的是 updated_at，只有大于这个 cursor 的记录才会在下一次同步中被导出。[1] 同时，Airbyte 也明确提醒：Incremental | Append 的第一次同步本质上等价于 full refresh，而且这种模式是 **at least once**，因此目的端会出现重复记录。[2]

这两点对 Week03 特别重要，因为它们说明：

1. 增量首先是一个**边界声明问题**；
2. 增量不是天然无重复的，重复是正常工程现实。

> ! 关键判断
>
> **增量 ingest 的重点不是“更快”，而是“边界更清楚、恢复更可控”。**

2. 先把 5 个最容易混淆的概念拆开：cursor、watermark、checkpoint、dedupe key、idempotency key

很多同学一到增量 / CDC 就把这三个词混着用。 这会直接导致后面的实现边界混乱。

2.1 Cursor：告诉系统“下一次从哪里开始看”

cursor 通常是某个可比较字段，例如：

- updated_at
- event_time
- sequence_id
- lsn

---

[1] Airbyte 文档将增量同步定义为基于 cursor 的变更抓取，常见 cursor 就是 updated_at；只有大于该 cursor 的记录才会在下一次同步中被导出。Airbyte | Incremental Sync

[2] Airbyte 文档说明，Incremental | Append 第一次运行等价于 full refresh，而且这种模式提供的是 **at least once** 复制保证，因此目的端可能出现同一记录的多个副本。Airbyte | Incremental Sync – Append

它的最核心的作用，是定义：

> **“哪些记录已经被看过，下一次应该从哪里继续。”**

2.2 Watermark：告诉系统“当前已确认到哪里”

watermark 更偏向系统级确认点，它不只是“看过”，而是“这之前的数据我已经按当前规则处理完了”。

在实践里，watermark 经常被用来表示：

- 当前 source 的完成边界
- 当前窗口的截止点
- 当前晚到数据还能被接受到什么范围

2.3 Checkpoint：把状态真的写下来

checkpoint 不是概念，它是**持久化状态**。

如果你没有把状态落盘，那么：

- 崩溃恢复就没有边界
- replay 就无从谈起
- backfill 会靠记忆驱动
- exactly-once 讨论也没有根基

2.4 Dedupe Key 与 Idempotency Key：不要用一个键去解决两个问题

这两个词在增量和 CDC 里也特别容易混。

dedupe key 解决的是：

> **两条输入在业务上是不是同一件事。**

例如：

- event_id
- ticket_id + updated_at
- source_id + lsn

idempotency key 解决的是：

> **同一次写入动作如果重复执行，会不会产生额外副作用。**

例如：

- batch_id + primary_key
- manifest_id + source_fingerprint + record_key
- run_id + sink_key

很多链路正是因为把这两个概念混成一个，才会出现：

- 输入层已经 dedupe 了，但写入层仍然重复 side effect
- 写入层看似幂等，输入层却已经把不同语义的事件错误合并

下面这张表是这节课你必须记住的“最小区分表”。

概念	它回答什么问题	最常见的形式
cursor	下一次从哪里继续读?	updated_at、event_ts、LSN、offset
watermark	当前处理确认到了哪里?	一个已确认时间点 / 序列位置
checkpoint	这个边界被持久化到了哪里?	JSON / DB table / metadata store
dedupe key	这两条输入在业务上是不是同一件事?	event_id、组合业务键、source_id + lsn
idempotency key	同一写入动作重复执行会不会产生额外副作用?	batch_id + sink_key、 run_id + primary_key

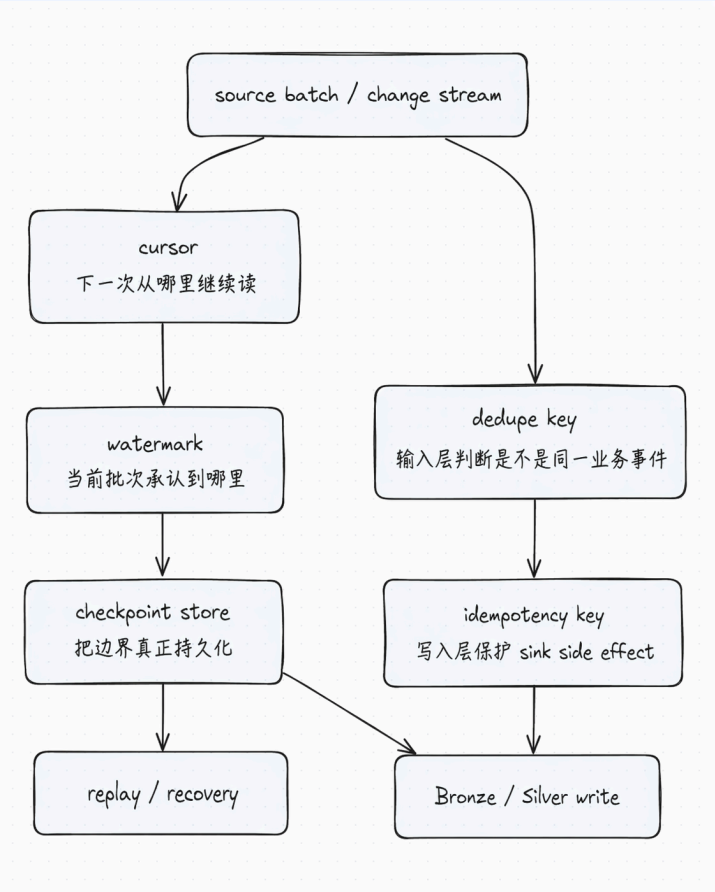
2.5 在 Week03 里，你应该先怎么用这几个概念

对当前课程主线，更稳的理解是：

- cursor：先从最简单可比较字段开始，比如 updated_at
- watermark：明确记录“本轮成功处理到哪里”
- checkpoint：先落一个最小状态对象，让 Week05～Week06 再逐步资产化与编排化
- dedupe key：优先选能稳定标识“同一业务事件”的键
- idempotency key：优先选能稳定保护 sink side effect 的键

这样比一上来就承诺完整 CDC 状态机更稳。

一张状态结构图把它们串起来



你可以把这张图直接读成 3 层：

- **读取层**：cursor 决定下一次从哪里继续看
- **状态层**：watermark + checkpoint 决定本轮到底承认到哪里
- **写入层**：dedupe key + idempotency key 决定重复输入和重复写入如何被隔离

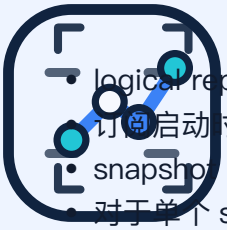
如果这 3 层没有被同时设计，增量链路通常会在 replay、迟到数据或 crash recovery 时一起失真。

3. PostgreSQL 的 logical replication / logical decoding 到底解决了什么

Week03 大纲里写了 Batch / CDC / Stream，因此你必须知道 CDC 的语义来源。

3.1 logical replication 的最基本工作方式

PostgreSQL 官方文档里写得很清楚：



- logical replication 使用发布 / 订阅模型
- 订阅启动时，通常会先做一次 snapshot copy
- snapshot 完成后，再持续发送后续变更
- 对于单个 subscription 内的 publication，订阅侧按发布顺序应用数据，从而保证事务一致性。[3]

这恰好说明了一个很重要的工程现实：

CDC 并不是“完全取代 snapshot”，而是“snapshot + 持续变化流”。

这和我们 Week03 的教学顺序是一致的：

1. 先建立 batch / snapshot 基线
2. 再引入 incremental / CDC 语义
3. 最后再谈 replay / backfill / runbook

3.2 logical decoding 在干什么

PostgreSQL 官方文档对 logical decoding 的定义是：

- 从 WAL（write-ahead log）里抽取持久化变更
- 把底层存储变化解码成应用可理解的流
- replication slot 代表一条可以被客户端重放的 change stream。[4]

也就是说，到了 CDC 这一层，你不再只是处理“文件里有哪几条记录”，而是在处理：

- 变更序列
- LSN
- slot 位置
- 客户端恢复点

3.3 为什么“有 slot”不等于“不重不漏”

这里一定要看官方文档里的一个关键提醒：

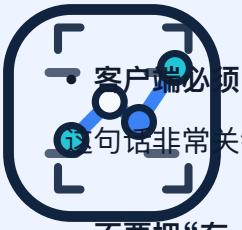
PostgreSQL 说得非常明确：

- logical slot 在正常运行下会把每个 change 发出一次
- slot 的当前位置只会在 checkpoint 时持久化
- 如果发生 crash，slot 可能回退到更早的 LSN
- 这会导致最近的一些 change 在重启后再次发送

---

[3] PostgreSQL 官方文档说明，logical replication 启动时通常会先复制快照数据，之后持续发送增量变化；同一 subscription 内按发布端顺序应用变更，以维持事务一致性。PostgreSQL | Logical Replication

[4] PostgreSQL 官方文档说明，logical decoding 从 WAL 中抽取变化；replication slot 在正常情况下每个变更发出一次，但由于位置只在 checkpoint 时持久化，crash 后可能回退到更早 LSN，从而导致最近变化再次发送；客户端需要自行避免重复处理的副作用。PostgreSQL | Logical Decoding Concepts



- **客户端必须自己负责避免重复处理这些消息带来的副作用。**[5]

这句话非常关键，因为它直接告诉你：

不要把“有 replication slot”误解成“天然 exactly-once”。

4. 为什么这节课要反复强调：不要轻易承诺 exactly-once

这是 Week03 最容易被讲虚的一部分。

4.1 先从最保守、最可靠的现实出发

Debezium 在官方 exactly-once 文档里明确写了：

- Debezium 默认提供的是 **at-least-once** 保证
- 这意味着不会漏 change，但在某些情况下会重复交付
- Debezium 本身**没有内部 deduplication layer**
- 如果要使用 exactly-once，需要依赖 Kafka Connect 对 source connector 的 EOS 支持。[6]

它的博客文章也把这个事实说得更直接：

- 过去 Debezium 的标准答案一直是：如果你需要 exactly-once，用户自己要实现 deduplication
- 之后才逐步转向 Kafka Connect 3.3+ distributed mode 上的 EOS 支持。[7]

4.2 即便进入 Kafka Connect EOS，也不要轻易说“问题解决了”

Debezium 的官方 EOS 文档在介绍 exactly-once 时，还专门加了一段非常重要的“Known issues and considerations”：

- 目前对于 Kafka / Kafka Connect exactly-once 的正确性，仍存在公开的已知问题与边界条件
- 由于 Kafka Connect EOS 依赖 Kafka transactions，这些问题也可能影响 Kafka Connect 的 EOS 保证。[8]

---

[5] PostgreSQL 官方文档说明，logical decoding 从 WAL 中抽取变化；replication slot 在正常情况下每个变更发出一次，但由于位置只在 checkpoint 时持久化，crash 后可能回退到更早 LSN，从而导致最近变化再次发送；客户端需要自行避免重复处理的副作用。PostgreSQL | Logical Decoding Concepts

[6] Debezium 官方文档说明，默认提供的是 **at least once**，并不内置 deduplication layer；若要利用 exactly-once，需要依赖 Kafka Connect 对 source connector 的 EOS 支持，而且文档也明确列出 known issues and considerations。Debezium | Exactly Once Delivery

[7] Debezium 官方博客也强调，过去如果需要 exactly-once，用户需要自行 dedupe；后续才借助 Kafka Connect 3.3+ distributed mode 上的 EOS 支持推进这条路线。Debezium Blog | Towards Debezium exactly-once delivery

对教学来说，这意味着：

Week03 不应该把学生实践写成“我们已经实现了 exactly-once”。 更稳妥也更工程化的表述应该是：

> 当前 Week03 的课程主线目标是 **at-least-once ingest + idempotent write + dedupe + replayable recovery**

4.3 这不是保守，而是工程诚实

对于课程项目来说，真正应该优先交付的是：

- 能复现
- 能定位
- 能补数
- 能回放
- 能解释为什么会重复、为什么不会漏

而不是为了追求“听起来更高级”的词，提前承诺一个连生产系统都要谨慎表述的保证。

5. Week03 在 OmniSupport Copilot 里该怎样落：先把可靠增量基线立住

这一节非常重要，因为它决定了学生页会不会再次脱离真实 repo。

5.1 当前阶段必须先交付什么

当前阶段应该优先做到：

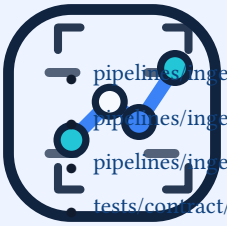
- batch + incremental cursor
- manifest-driven ingest
- checkpoint state
- late-arrival handling
- dedupe
- replay / backfill

并且继续沿着 repo 当前已经存在的对象走：

- contracts/data/*.json
- data/seed_manifests/*.json

---

[8] Debezium 官方文档说明，默认提供的是 **at least once**，并不内置 deduplication layer；若要利用 exactly-once，需要依赖 Kafka Connect 对 source connector 的 EOS 支持，而且文档也明确列出 known issues and considerations。Debezium | Exactly Once Delivery



- pipelines/ingestion/seed_loader.py
- pipelines/ingestion/ticket_ingest.py
- pipelines/ingestion/doc_ingest.py
- tests/contract/test_json_schemas.py

这样你既不会掉到“纯概念”，也不会被逼着在本地搭完整 Kafka / Debezium。

5.2 这些进阶对象你现在要知道，但不必本地全量搭起

下面这些对象你现在应该知道它们的工程意义：

- PostgreSQL logical decoding
- replication slot / LSN
- Debezium snapshot + stream
- Kafka Connect exactly-once support
- 更细致的流式边界与观测

它们在本周的角色更像：

- 认知边界
- 架构比较
- 未来接到真实 CDC / Stream 时的预习对象

而不是当前仓库里必须立刻全量实现的本地主线。

5.3 为什么先把边界收在这里

最稳的工程解释其实很简单：

- 先把 **可以在当前项目和当前仓库中稳定成立的可靠增量链路** 做出来
- 先把 contract / manifest / checkpoint / dedupe / replay 这些边界写清
- 再去接 Debezium / Kafka / streaming，才不容易把系统讲虚

6. 动手实践：把 batch 基线推进到 incremental 设计

这一节你不需要先写一整套流式框架， 而是要先做 3 个最重要的工程动作。

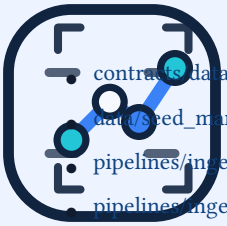
6.1 先读一遍当前 repo 里最相关的对象

先跑环境和最小 baseline：

```
docker compose --env-file infra/env/.env.local -f infra/docker-compose.yml up -d --build

docker compose --profile tools --env-file infra/env/.env.local -f infra/docker-compose.yml run --rm devbox \
  pytest tests/contract/ -v
```

再重点读 4 类对象：



contracts/data/ticket_contract.json
data/seed_manifests/*.json
pipelines/ingestion/seed_loader.py
pipelines/ingestion/ticket_ingest.py

6.2 写出一份《增量采集策略说明》

在仓库里新增：

```
docs/blueprints/week03/incremental_ingest_strategy_v1.md
```

建议至少包含这些小节：

```
# Incremental Ingest Strategy v1

## 1. 哪些 source 适合继续走 batch / snapshot
## 2. 哪些 source 可以升级为 incremental cursor
## 3. 每类 source 的 cursor 候选字段
## 4. 为什么这些字段可用 / 不可用
## 5. 迟到、重复、乱序如何处理
## 6. 哪些场景仍需要 replay / backfill
```

6.3 再写一份《checkpoint 设计草案》

新增：

```
docs/blueprints/week03/checkpoint_state_v1.md
```

建议至少回答下面 5 个问题：

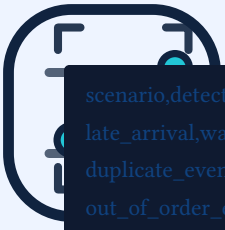
1. checkpoint 要按 source 记录什么？
2. cursor_value 和 watermark 是否分开记录？
3. 如果本轮 ingest 失败，checkpoint 什么时候更新，什么时候不更新？
4. 如果 crash 恢复后可能拿到重复 change，消费端怎么处理？
5. replay / backfill 会不会覆盖当前 checkpoint，还是走独立边界？

6.4 最后补一个“迟到 / 重复 / 乱序决策表”

新增：

```
reports/week03/late_arrival_decision_table.csv
```

建议格式：



```
scenario,detected_by,default_action,needs_manual_review,notes
late_arrival,watermark_gap,replay_window,false,重新纳入窗口
duplicate_event,idempotency_key,dedupe,false,依赖主键或事件键去重
out_of_order_event,sequence_or_ts,quarantine,true,需确认业务顺序
crash_after_emit,recovery_lsn,replay_and_dedupe,false,允许重复但不允许副作用重复
```

这 3 份文件就是本课真正的工程产出。

7. 这节课最重要的 7 个判断

1. 增量 ingest 的复杂度远高于 batch ingest，因为它引入了 cursor、checkpoint、重复、迟到和恢复点。
2. cursor、watermark、checkpoint 是三个不同层级的对象，不能混用。
3. PostgreSQL logical replication 的现实是：先 snapshot，再持续送变更，而不是从一开始就只有流。
4. replication slot 在 crash 后可能回退到更早位置，因此消费端必须负责处理重复消息的副作用。
5. Debezium 默认是 at-least-once，不自带内部 dedupe。
6. 当前课程主线最该交付的是 at-least-once + idempotent write + dedupe + replayable recovery。
7. Week03 的真正价值，不是“讲到流式”，而是把 ingest 变成可解释、可恢复、可继续工程化的链路。

8. 本课自检清单

学完这一讲后，请确认你已经能勾掉这些项：

- ☐ 我能解释为什么增量 ingest 比 batch ingest 更容易出错
- ☐ 我能区分 cursor / watermark / checkpoint
- ☐ 我知道 logical replication 和 logical decoding 各自解决什么问题
- ☐ 我知道 replication slot crash 恢复后为什么可能重复发消息
- ☐ 我不会再轻易把当前课程主线写成“exactly-once 已实现”
- ☐ 我已经新增 incremental_ingest_strategy_v1.md
- ☐ 我已经新增 checkpoint_state_v1.md
- ☐ 我已经新增 late_arrival_decision_table.csv

9. 课后最小行动

在进入课时 4 之前，请至少完成这 4 件事：

1. 跑一次 pytest tests/contract/ -v
2. 读完 seed_loader.py 和 ticket_ingest.py

3. 写完 incremental_ingest_strategy_v1.md
4. 写完 checkpoint_state_v1.md

如果你已经能把这两份文档写清楚，说明你不是只会说“增量更复杂”，而是真的开始具备 Week03 后半程所需要的工程判断。

延伸阅读

- Airbyte / Incremental Sync
- Airbyte / Incremental Append
- PostgreSQL / Logical Replication
- PostgreSQL / Logical Decoding Concepts
- Debezium / Exactly Once Delivery
- Debezium / Towards Exactly-once Delivery