Week04｜课时 2｜Iceberg 的状态模型：snapshot、manifest、metadata log 与 time travel

Table of contents

先把 Iceberg 的状态对象讲清，再谈 time travel 和演进

这一讲不是在讲目录技巧，而是在讲：

为什么 Iceberg 能把一组文件组织成可提交、可回看、可演进的表状态。

进入课时 3 回看课时 1 返回 Week04 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 已经建立了 Week04 的总判断：

没有“有记忆的表”，AI 系统就没有稳定的数据状态锚点。

这一讲往前再推进一步：

Iceberg 到底靠什么把这套状态记忆组织起来。

你要真正分清的是：

- snapshot 记录的是什么；
- manifest list / manifest 怎样指向 data files；
- metadata file 和 metadata log 为什么是状态证据链；
- time travel 为什么不是复制整张表；
- schema evolution 为什么必须在表状态模型里解释。

参考学习时间

建议按一节标准课安排：读完正文后，把状态结构图和 metadata inspection 解读写进 snapshot_state_model_v1.md。

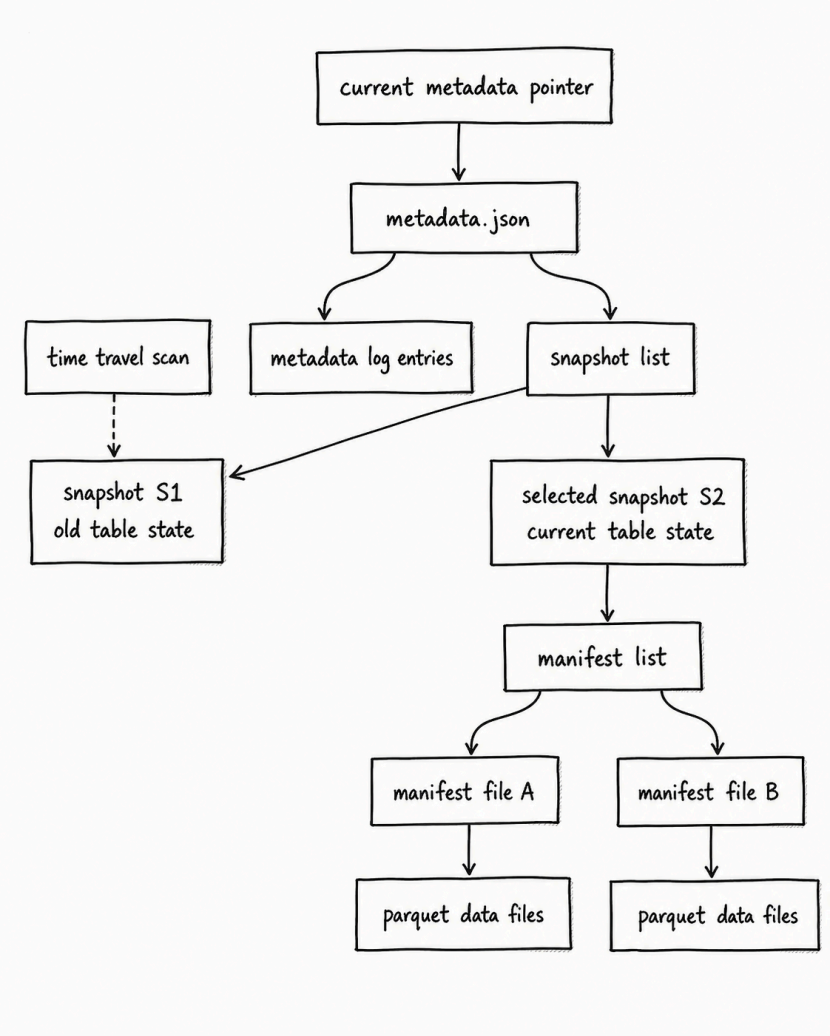
学完这一讲，你应该能做到什么

1. 解释 Iceberg 为什么不是“按日期分目录”的升级版。
2. 能说清 snapshot、manifest list、manifest、data files、metadata log 的关系。
3. 能解释 time travel 为什么能成立，又为什么不是每次复制全表。
4. 能读懂最小 inspection 输出里 snapshots、history、files、metadata log 各自说明什么。
5. 能说明 expire snapshots / cleanup 是维护边界，不是本周主线。

本课产出

- docs/blueprints/week04/snapshot_state_model_v1.md
- reports/week04/time_travel_demo_notes.md

先看一张总图

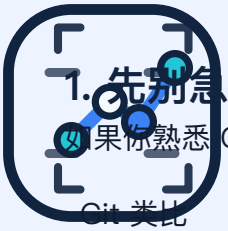


这张图要表达的重点是：

Iceberg 不是直接“看目录”，而是通过 metadata 把一次表状态提交组织成可追踪对象。

当前读表时，catalog 会先找到当前 metadata pointer，再进入 metadata.json，由它定位当前 snapshot、manifest list、manifest files，最后扫描真正的 Parquet data files。time travel 不是去猜某个日期目录，而是显式指定旧 snapshot，让读取路径从旧状态出发。

这对 AI 系统复盘很关键：当 silver.ticket_fact 的一次写入导致评测分数变化，团队能沿着 metadata -> snapshot -> manifest -> files 的链路说明“这次状态到底由哪些文件构成”。它不是把所有历史表复制一遍，而是通过 metadata 保留状态关系。



1. 先别急着背名词：把 Iceberg 想成“数据表的提交账本”

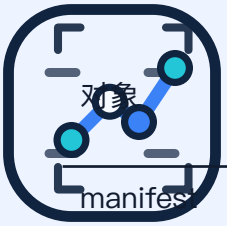
如果你熟悉 Git，可以先借用一个有限类比：

Git 类比	Iceberg 对象	类比能帮助理解什么	类比边界
commit	snapshot	一次成功提交后的状态可以被引用	snapshot 记录的是表状态，不是代码 diff
提交记录	metadata log	状态如何一路变化到现在	metadata log 不是人写的 commit message
文件清单	manifest / manifest list	某个状态引用了哪些数据文件	manifest 有统计和分区信息，不只是路径列表
工作目录文件	data files	真正存数据的 Parquet 文件	Iceberg 不直接靠目录表达语义

这个类比只能帮你入门。Iceberg 是 table format，不是代码仓库；它关心的是表状态、schema、partition、snapshot 与 data files 的关系。

2. 每个状态对象到底解决什么问题

对象	小白理解	工程上解决什么问题	在 Pyiceberg 里怎么看	常见误解
table metadata file	账本当前页	保存 schema、partition、snapshot 列表等表状态入口	`table.metadata` 或 metadata log	以为它就是数据文件
snapshot	一次表状态提交	让 `silver.ticket_fact` 某次物化结果可引用、可回看	`table.inspect.snapshots()`	以为 snapshot 是一个目录
manifest list	snapshot 的索引目录	说明该 snapshot 下面有哪些 manifest	inspection 输出里的 `manifest_list`	以为它等于所有 data files



对象	小白理解	工程上解决什么问题	在 Pylceberg 里怎么看	常见误解
manifest	文件级清单与统计	帮助定位 data files、分区与文件统计	table.inspect.files() 间接观察	以为它只是路径列表
data file	真正的数据文件	承载 ticket/doc 表数据	table.scan().to_arrow() /files inspection	以为查询直接扫整个 bucket
metadata log	账本页历史	复盘 metadata 如何演进	table.inspect.metadata_log_entries()	以为历史永久都在
history	哪个 snapshot 何时成为 current	看表状态切换时间线	table.inspect.history()	以为 history 等于业务审计日志

把这张表看懂，比背 Iceberg 定义重要得多。因为后续 Lab 里你真正要做的是：

把 inspection 输出翻译成工程判断。

3. Snapshot 不是一个文件，而是一版表状态

snapshot 可以先粗暴理解成：

> **某一时刻这张表的一个可命名状态。**

它不是“单个文件”，也不是“一个分区目录”，而是一组表状态信息的入口点。

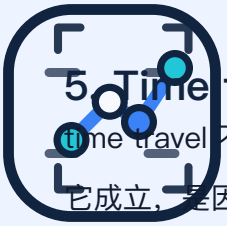
一次 append、overwrite 或 schema change，只要形成提交，就会改变表状态。后续你能不能解释“这次提交影响了哪些数据”，就取决于 snapshot 和 metadata 是否保留了足够证据。

4. Manifest 不是目录清单，而是状态索引

manifest 真正解决的是：

- 这次状态包含哪些 data files；
- 哪些 files 属于哪些分区；
- 文件级统计信息是否可用于读优化；
- 哪些状态变化应该被纳入下一次读取。

如果没有 manifest 这一层，time travel 和 evolution 很快就会退化成文件猜谜。



5. Time travel 为什么不是复制整张表

time travel 不是魔法，也不是“每次提交都复制一整张表”。

它成立，是因为：

1. 你有稳定的 snapshot 标识；
2. snapshot 能指向对应的 manifest list；
3. manifest list 能继续指向 manifest 和 data files；
4. 表状态历史没有被随意丢掉。

所以 time travel 的本质，不是“查一张旧表”，而是：

回到一个旧的 snapshot 所代表的状态集合。

6. Snapshot 时间线示例

以 silver.ticket_fact 为例，Week04 最小闭环至少应该能形成这样一条状态时间线：

Snapshot	触发动作	业务含义	能用于什么复盘
S1	首次物化 silver.ticket_fact	Week03 ticket ingest baseline 第一次进入 Silver	证明最小事实表已经站住
S2	追加一批 ticket 更新	新一批工单状态进入当前表	解释答案或统计变化是否来自数据更新
S3	add-column 后再写入	schema 演进后旧数据仍可读	证明字段扩展没有破坏历史状态

这是一条表状态时间线。后续 Week08 的索引、Week11 的评测和 Week14 的发布治理，都可以引用其中某个 snapshot。Week04 不要求完成后续链路，但必须先让这条时间线可见。

7. Metadata inspection 应该怎么看

真实命令以 Week4 项目代码落地后同步为准。课程讲义里先用结构化示意说明你应该观察什么：

```
# 示意：待 Week4 项目代码落地后，同步 `pipelines.lakehouse.*` 的真实 inspection 命令。
inspect table bronze.raw_ticket_event

snapshots:
  - snapshot_id: 1001
    operation: append
```

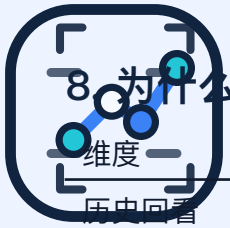
```
    committed_at: 2026-04-xxT10:12:00Z
  - snapshot_id: 1002
    operation: append
    committed_at: 2026-04-xxT10:18:00Z

history:
  - made_current_at: 2026-04-xxT10:12:00Z
    snapshot_id: 1001
  - made_current_at: 2026-04-xxT10:18:00Z
    snapshot_id: 1002

files:
  - file_count: 12
  - avg_file_size_mb: 18
  - partition_summary: ingest_date=2026-04-xx
```

解读顺序不要反过来：

1. 先看 snapshots：有没有形成明确状态提交；
2. 再看 history：当前状态怎么一路演进；
3. 再看 files：当前文件组织是不是过碎、过散；
4. 最后看 metadata log：状态链是否有足够历史。

更细一点，读 inspection 输出时不要只看“有没有结果”，要看这些字段：

字段	你要判断什么
snapshot_id	后续 time travel / report 是否能引用明确状态
parent_id	这次状态是否接在预期父 snapshot 之后
committed_at / made_current_at	写入时间是否和本次实验对齐
operation	append、overwrite、schema-only change 是否符合预期
manifest_list	snapshot 是否有可追溯的 manifest 入口
file_path	数据文件是否落在预期 warehouse / table location 下
record_count	row count 是否和 source coverage 对得上
file_size_in_bytes	是否出现明显小文件问题
partition	分区分布是否过散或过偏
latest_schema_id	schema evolution 是否留下清晰记录



8. 为什么不是按日期分目录

维度	按日期分目录	Iceberg 表状态
历史回看	依赖目录命名和人工约定	依赖 snapshot / metadata
原子提交	很难避免读到半成品文件	读者不会看到 partial / uncommitted changes
Schema 演进	需要自己维护新旧文件兼容	schema evolution 进入 metadata
分区演进	改策略常常牵涉重写目录	partition evolution 是 metadata operation，不急切重写历史文件
查询规划	查询者常暴露在目录布局里	hidden partitioning 让查询按业务字段写
复盘证据	很难证明“当前表状态由哪些文件构成”	snapshot / manifest / files 构成证据链

目录组织只是文件布局；Iceberg 管的是版本化表状态。这个区别决定了它能否承担 AI 系统的数据状态锚点。

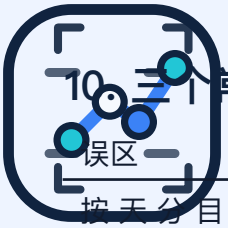
9. Schema evolution 的边界

Iceberg 支持 add、drop、rename、update、reorder 等 schema evolution 能力，但 Week04 只要求你演示 **add-column**。

原因很简单：

- add-column 最适合教学本地闭环；
- 它能说明 schema evolution 的存在；
- 不会让课程提前进入复杂兼容性治理；
- 后续 rename / drop / type update 必须结合数据消费方、contract 和发布策略一起判断。

> ℹ 不要把 schema evolution 理解成“字段随便改”
>
> 表格式支持演进，不代表团队可以无成本修改语义。字段变化仍然要进入 contract、manifest、release note 和下游兼容性检查。



10. 三个常见误区

误区	为什么错	正确判断
按天分目录就是 time travel	目录只能提供物理组织，不能保证表状态历史	time travel 依赖 snapshot / metadata
每次 snapshot 都复制全表	snapshot 指向状态集合，不等于复制全部 data files	关注 snapshot 与 manifest 的关系
metadata 只是额外开销	没有 metadata，状态记忆就无法成立	metadata 是复盘证据链

11. Snapshot 保留与清理边界

snapshots 会积累，metadata file 也会积累。真实生产系统要考虑：

- expire snapshots；
- metadata cleanup；
- orphan file cleanup；
- compaction。

但 Week04 不把这些做成主线，因为本周最重要的是先证明：

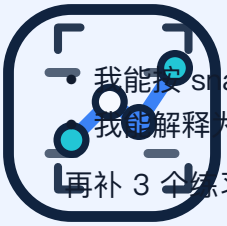
表状态存在、历史可看、旧状态可回到、baseline 可记录。

> ⚠ snapshot 保留不是越多越好，也不是越少越好
>
> 保留更多 snapshot 有利于 time travel 和复盘，但 metadata 与存储成本会增长。expire snapshots 可以清理旧状态，也会缩短历史回看范围。企业里要基于合规、审计、成本和恢复目标设计保留策略；Student Core Pack 只要求你理解并记录这个取舍。

> 💡 读 inspection 输出的顺序
>
> 先问“有没有形成 snapshot”，再问“它何时成为 current”，最后再看 files 和 partition。不要一上来盯着文件大小做调优。

12. 本课自检清单

- 我能从表状态图解释 snapshot、manifest list、manifest、data files、metadata log 的关系。
- 我能说明 time travel 为什么不是复制全表。



- 我能按 snapshots -> history -> files -> metadata log 的顺序读 inspection 输出。
- 我能解释为什么本周只演示 add-column，而不展开复杂 schema governance。

再补 3 个练习：

1. 画出从 metadata pointer 到 data files 的读取路径。
2. 用 100 字解释 time travel 为什么能成立。
3. 看一段示意 inspection 输出，指出哪一个字段最适合进入 baseline report。

13. 课后最小行动

新建：

```
docs/blueprints/week04/snapshot_state_model_v1.md
```

至少写清：

1. 当前 Week04 最小 4 表中，哪个表最适合作为 inspection 示例；
2. 你预期会看到哪些 snapshots / history / files 信息；
3. 哪些 metadata 指标会进入课时 5 的 baseline report。

延伸阅读

- Apache Iceberg / Reliability[1]
- Apache Iceberg / Evolution[2]
- Apache Iceberg / Maintenance[3]
- Pyiceberg / API[4]

[1] Iceberg 官方 Reliability 文档解释了 snapshot-based reads、atomic metadata swap、version history 与 rollback 等机制，适合用来加深本课的 snapshot / metadata log / time travel 理解。Apache Iceberg | Reliability

[2] Iceberg Evolution 文档说明 schema evolution 支持 add、drop、rename、update、reorder，且不需要重写数据文件；partition evolution 也是 metadata operation，不急切重写历史数据。本课只要求把这些能力和状态模型对应起来。Apache Iceberg | Evolution

[3] Iceberg Maintenance 文档说明 snapshot 过期会影响 time travel / rollback 可用范围；这能帮助你理解“历史状态不是无限免费保留”。Apache Iceberg | Maintenance

[4] Pyiceberg API 文档包含 InspectTable、snapshots、history、metadata_log_entries、files 等 inspection 入口，后续项目 runbook 落地后会和本课的 inspection 解读对应起来。Pyiceberg | API