

《Rethink the Sync》论文笔记

External Synchrony: 外部同步 I/O 模型的设计、实现与评测

原文: Rethink the Sync (OSDI 2006)

作者: Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn

单位: University of Michigan, Department of Electrical Engineering and Computer Science

笔记性质: 零基础友好的深度论文精读 (中文, 术语保留英文原文)

涵盖: 背景与动机 → External Synchrony 模型 → 设计 → 实现细节 → 评测 → 参考文献关联分析

目录

引言与背景：文件系统的两难

- 1.1 论文基本信息与一句话总结
- 1.2 Durability 与 Performance 的对立
- 1.3 同步 I/O 与异步 I/O 两种模型
- 1.4 2006 年的系统生态背景

Motivation：为什么必须「重新思考 sync」

- 2.1 同步 I/O 慢得离谱：两个数量级
- 2.2 异步 I/O 快但不安全
- 2.3 fsync 也不可靠：硬盘缓存 (drive cache) 撒谎问题
- 2.4 应用程序员被迫手写 group commit
- 2.5 问题形式化：我们到底想要什么保证

External Synchrony 模型

- 3.1 核心思想：换保证对象，不换保证内容
- 3.2 从 application-centric 到 user-centric
- 3.3 正确性定义：值相同 + Lamport 因果序相同
- 3.4 磁盘提交也是外部输出
- 3.5 总体工作流程 (Figure 1 精讲)

设计 (Design)

- 4.1 两条设计原则
- 4.2 性能优化之一：group commit 与 value logging
- 4.3 性能优化之二：缓冲输出而非阻塞应用
- 4.4 commit dependency 与因果依赖传播
- 4.5 Deciding when to commit: output-triggered commits
- 4.6 Limitations: 作者自己承认的局限

实现细节 (Implementation)

- 5.1 Speculator 背景
- 5.2 从 speculation 到 synchronization: 数据结构的降维
- 5.3 xsyncfs: 对 ext3 的修改点
- 5.4 为什么必须用 journaled 模式而非 ordered 模式
- 5.5 write barrier 移植到 2.4 内核
- 5.6 Output-triggered commits 的实现: Speculator 回调

5.7 Rethinking sync: sync/fdatasync 的处理

5.8 共享内存、mmap 与多线程的依赖传播

5.9 因果依赖传播全流程演练

评测 (Evaluation)

6.1 评测问题与方法论

6.2 断电持久性实验: ext3 撒谎实锤

6.3 PostMark: I/O 密集型基准

6.4 Apache build: 吞吐与延迟的平衡

6.5 MySQL + 修改版 TPC-C: 应用自带同步策略时

6.6 SPECweb99: 网络密集型负载

6.7 Output-triggered commits 的收益

参考文献关联分析 ([1]–[33] 逐条)

总结与启示

附录

A. 术语表

B. 论文完整参考文献列表

C. 笔记使用说明

第 1 章 引言与背景：文件系统的两难

Introduction & Background

1.1 论文基本信息与一句话总结

《Rethink the Sync》发表于 OSDI 2006（操作系统设计与实现研讨会，系统领域顶级会议之一），作者全部来自密歇根大学：Edmund B. Nightingale、Kaushik Veeraraghavan、Peter M. Chen、Jason Flinn。论文收录于 OSDI '06 论文集第 1–14 页，并获得该届 OSDI 的 Best Paper 奖。

一句话总结：**同步 I/O (synchronous I/O) 之所以慢，是因为操作系统把「保证」给错了对象——它阻塞了调用进程，而真正需要保证的只是「外部观察者看到的东西」；把保证对象从应用进程换成外部观察者，就能在不牺牲可靠性的前提下获得接近异步 I/O 的性能。**作者把这个新模型称为 **external synchrony (外部同步)**，并在 Linux 上实现了一个外部同步文件系统 **xsyncfs**。

核心数字先放在这里，后文会逐一拆解：在 I/O 密集型的 PostMark 基准上，xsyncfs 与默认的异步 ext3 性能差距在 7% 以内；与同步挂载的 ext3 相比快一个数量级以上；与开启了 write barrier（写屏障）的同步 ext3 相比快**两个数量级**；而在自带同步策略的 MySQL 数据库上跑修改版 TPC-C 时，xsyncfs 反而能让吞吐提升最多 3 倍。

📖 基础知识补充

OSDI 是什么？ OSDI (Symposium on Operating Systems Design and Implementation) 与 SOSP 并称操作系统领域两大顶级会议，偶数年举办。能在 OSDI 发表的系统论文通常既有新想法又有真实实现和严格评测。

文件系统 (file system)：操作系统中负责把数据组织成「文件」和「目录」并持久存放到磁盘上的子系统。本文涉及的 ext3 是 2000 年代 Linux 的主流文件系统。

I/O (输入/输出)：程序与外部世界（磁盘、网络、屏幕、键盘）交换数据的动作。磁盘 I/O 的特点是比内存慢几个数量级——2006 年的机械硬盘一次随机写需要约 10 毫秒，而 CPU 执行一条指令只要不到 1 纳秒。

挂载 (mount) 选项 sync / async：Linux 挂载文件系统时可以指定 sync 选项，要求每次写操作都等数据真正落盘后才返回；默认是异步，写操作把数据放进内存缓存就返回。

💡 核心洞见

Insight#1: 这篇论文的本质是一次「视角转换」式的创新——它没有发明新的磁盘调度算法或新的日志格式，而是重新定义了「同步」二字的含义：同步不是一种实现方式（阻塞进程），而是一组对外部观察者可见的行为保证（durability + causal ordering）。一旦把定义从「怎么做」切换到「对外承诺什么」，实现空间就豁然开朗。

1.2 Durability 与 Performance 的对立

论文开篇第一句就点题：“File systems serve two opposing masters: durability and performance.”（文件系统伺候着两位互相敌对的主人：持久性与性能。）

Durability（持久性）的含义是：一旦一次写操作「完成」，即使紧接着操作系统崩溃（OS crash）或整机断电（power failure），数据也不能丢。**Performance（性能）**在这里主要指吞吐（throughput）和延迟（latency）。问题在于：机械硬盘极慢，要保证持久性就必须等磁盘；要等磁盘就必须阻塞应用；阻塞应用性能就崩。这个张力直接催生了两种截然不同的文件 I/O 模型。

📖 基础知识补充

为什么磁盘这么慢？ 2006 年的主流硬盘是 7200 RPM 机械盘：盘片旋转一周约 8.3 毫秒，平均旋转延迟约 4.2 毫秒，加上磁头寻道（seek）约 5-10 毫秒，一次随机 I/O 的总成本约 10 毫秒，即每秒最多约 100 次随机写。相比之下内存访问是纳秒级——磁盘比内存慢约 **一百万倍**。

崩溃（crash）与断电（power failure）的区别：OS 崩溃时硬盘本身还通电，已经到达硬盘缓存的数据硬盘自己还会写完；而断电时硬盘缓存里的数据会一起丢失（除非磁盘有电池）。本文反复强调的可靠性问题主要针对断电。

吞吐 vs 延迟：吞吐是单位时间完成的总工作量，延迟是单个操作从发起到完成的时间。批处理系统追求吞吐，交互式系统追求低延迟，两者经常冲突——本文第 4.5 节的 output-triggered commits 就是在调和这对矛盾。

💡 核心洞见

Insight#2: 「两个互相敌对的目标」往往是系统创新的富矿。当一个领域长期认为 A 和 B 不可兼得（持久性 vs 性能），通常是因为大家默认了某个隐含前提（「保证必须给调用进程」）。找到并推翻这个隐含前提，比在同一条路上优化 10% 更有价值。

1.3 同步 I/O 与异步 I/O 两种模型

同步 I/O (synchronous I/O)：以 Linux 以 `sync` 选项挂载的文件系统为代表。它通过「阻塞调用进程直到修改提交到磁盘」来保证持久性。给用户的是一个干净的抽象：**任何你观察到已经完成的文件系统操作都是持久的**，后续的 OS 崩溃或断电都不会让它丢失。同步 I/O 还保证**顺序性 (ordering)**：如果操作 A 因果上先于操作 B，那么 B 的效果可见时 A 的效果必然也可见。代价是性能：作者的实测表明，同步 I/O 的阻塞可以让磁盘密集型基准测试慢**两个数量级**（100 倍）。

异步 I/O (asynchronous I/O)：不阻塞调用进程，修改通常在调用返回很久之后才提交到磁盘。它快，但不安全：

- **用户会看到「无效输出」**：用户看到的屏幕/网络输出可能依赖于尚未提交的修改。如果系统在提交前崩溃，用户之前看到的输出就成了「从来没发生过的事」。
- **把负担转嫁给程序员**：需要持久性或顺序保证的应用必须自己插入显式同步操作（如 `fsync`），有时甚至要自己实现复杂的 group commit（组提交）策略才能拿到可接受的性能。

尽管保证如此糟糕，**绝大多数本地文件系统默认都提供异步 I/O 抽象**——因为同步 I/O 实在太慢了。这就是 2006 年的现实：可靠性正确的选项存在，但没人用得起。

📖 基础知识补充

fsync 是什么？ POSIX 提供的系统调用，语义是「把这个文件的所有脏数据强制写到持久存储，写完才返回」。数据库（MySQL、PostgreSQL）依赖 `fsync` 实现事务的 D（Durability）。类似的还有 `fdatasync`（只刷数据不刷元数据）和 `sync()`（刷整个系统的脏页）。

group commit (组提交)：把多个修改攒在一起，用一次磁盘写原子地提交。因为磁盘一次 I/O 的固定成本高，攒批能摊薄成本。数据库界的经典优化，后文 4.2 节详解。

页缓存 (page cache / file cache)：内核用空闲内存缓存文件数据。写文件时默认只写页缓存（数据变「脏页」），由内核回写线程稍后刷盘。这是异步 I/O 快的根本原因。

💡 核心洞见

Insight#3：同步 I/O 的问题不是「保证太多」，而是「给保证的方式太笨」——它用「阻塞进程」这种实现手段去交付保证，于是保证的粒度被迫等于进程的调度粒度。异步 I/O 的问题则是「保证太少」，把正确性负担推给了每个应用。External synchrony 的定位恰好是中间点：**保证一点不少，交付方式彻底换掉。**

1.4 2006 年的系统生态背景

理解这篇论文需要把它放回 2006 年的技术环境：

- **内核**：作者的实现基于 Red Hat Enterprise Linux 3，内核 2.4.21。当时 Linux 2.6 内核刚普及不久，2.6.11 才引入 write barrier 支持。

- **文件系统**：ext3 是绝对主流（journaled / ordered / writeback 三种日志模式），ReiserFS、JFS、XFS 各有拥趸；soft updates 是 BSD 阵营的方案。
- **硬件**：机械硬盘统治一切，磁盘自带 2 MB 左右的易失写缓存（volatile drive cache），且默认开启、会重排写请求；SSD 尚未进入消费市场；电池备份的 NVRAM 缓存是高端存储阵列才有的奢侈品。
- **应用**：MySQL 5.0 + InnoDB 是开源数据库主力；Apache 2.0 统治 Web 服务器；PostMark、SPECweb99、TPC-C 是标准评测负载。

这个背景下，「同步 I/O 慢 100 倍」不是夸张修辞，而是机械硬盘物理特性的直接后果；「fsync 不防断电」则是当时桌面系统的普遍默认配置（见第 2.3 节）。

📖 基础知识补充

ext3 的三种日志模式：journaled（数据+元数据都进日志，最稳但写两遍）、ordered（默认，只记元数据日志，但保证数据块先于元数据落盘）、writeback（只记元数据，不保证数据顺序，最弱）。第 5.4 节会深入为什么 xsyncfs 必须用 journaled 模式。

write barrier（写屏障）：一种磁盘协议机制，强制硬盘在 barrier 之前的所有写必须先于 barrier 之后的写落到盘片，实质是「在关键点冲刷 drive cache」。Linux 2.6.11 起支持。

NVRAM / 电池备份缓存：用带电池的内存做写缓存，断电也不丢，高端存储（如 NetApp）用它把「同步写」变成内存速度。本文相关工作部分会对比这类硬件方案（[2][7][31][33]）。

💡 核心洞见

Insight#4：评测一篇老论文时，先还原它的「技术坐标系」。xsyncfs 的 100 倍提速听起来惊世骇俗，但基数是「每次事务两次同步冲刷硬盘缓存」的极端糟糕实现；理解基线为什么那么差，才能正确理解改进的幅度与适用边界（比如换到今天的 NVMe + SSD 时代，绝对数字会完全不同，但「延迟保证而不阻塞」的思想依然成立）。

第 2 章 Motivation: 为什么必须「重新思考 sync」

Motivation

2.1 同步 I/O 慢得离谱：两个数量级

同步 I/O 的性能灾难来自一个简单的物理事实：**每次写都要等一次磁盘机械动作**。一块 7200 RPM 的磁盘每秒最多完成约 100 次随机写事务。如果应用做 10,000 次小文件写（PostMark 的量级），同步模式光等磁盘就要上百秒，而异步模式把数据攒在内存里顺序刷盘，几秒钟就完事。作者在 PostMark 上实测：同步 + write barrier 的 ext3 比 xsyncfs 慢超过两个数量级；即使不带 write barrier 的同步 ext3（断电会丢数据的那种）也慢一个数量级以上。

更要命的是，同步 I/O 无法对多线程应用做 group commit：应用必须等第一次修改提交返回后，才会发起第二次修改，文件系统永远没有「攒批」的机会。这是一个结构性缺陷——性能损失不是实现不好，而是抽象本身造成的。

📖 基础知识补充

数量级 (order of magnitude)：指 10 倍。「快两个数量级」≈ 快 100 倍。

为什么同步模式不能 group commit? group commit 的前提是「手头同时有多个待提交的修改」。同步语义下进程被阻塞在写调用里，第二个修改根本还没产生，文件系统自然无批可攒。这是一个因果死锁：要快就得攒批，要攒批就得等后续写，要等后续写就得先返回——但「先返回」恰恰是同步语义禁止的。

💡 核心洞见

Insight#5：同步 I/O 的性能问题是「语义⇒实现」的强耦合：语义要求「返回即持久」，实现只能「阻塞到持久」。External synchrony 的突破口就是切断这个耦合——语义改成「外部可见即持久」，实现就可以「立即返回 + 延迟持久 + 控制可见性」。换约束对象，而不降低约束强度。

2.2 异步 I/O 快但不安全

异步 I/O 的默认行为是：写调用把数据拷贝进页缓存就返回，内核的 pdflush 之类的回写线程每隔约 5 秒（ext3 默认提交间隔）或内存压力下才把脏页刷到磁盘。这意味着「调用完成」与「数据持久」之间有一个最长数秒的时间窗。窗口内崩溃/断电，数据丢失。

论文强调了一个常被忽视的角度：**丢失的不只是数据，还有「用户已经看到的输出」的合法性**。想象一个编辑器显示「文件已保存」，然后断电，重启后文件根本没有保存——用户看到的输出（「已保

存」) 依赖于一个从未真正发生的磁盘修改。这个输出是「invalid (无效)」的。异步系统无法阻止这种幻觉, 因为它根本不追踪「输出」与「未提交数据」之间的依赖关系。

此外, 异步 I/O 把可靠性负担完全推给了应用: 程序员必须自己决定在哪里插 fsync; 插少了丢数据, 插多了性能崩; 想让多个修改原子提交还得手写 group commit 协议。可靠性变成每个应用各自为战的「手工活」。

📖 基础知识补充

脏页回写 (writeback): Linux 内核定期把页缓存里的脏页写回磁盘。2.4 内核由 bdfush/kupdated 负责, 默认脏页最多存活 30 秒; ext3 的日志提交间隔默认 5 秒。这就是「崩溃最多丢最近几秒数据」说法的来源。

什么是「无效输出」? 用数据库术语类比: 这相当于「脏读」的对外版本——外部观察者读到了一个从未被提交的状态。容错领域称之为 output commit problem (输出提交问题), 见第 7 章对文献 [3][11][30] 的分析。

💡 核心洞见

Insight#6: 异步 I/O 的真正代价不是「丢几秒数据」这种可量化的风险, 而是它摧毁了「眼见为实」的因果链条: 用户看到的任何输出都可能是幻觉。可靠性问题的本质是「外部世界观察到的状态」与「系统稳定存储的状态」之间失去了同步——这正是「external synchrony」这个名字的由来: 让外部观察与稳定存储重新同步。

2.3 fsync 也不可靠: 硬盘缓存撒谎问题

这是论文中最「惊悚」的动机部分, 也是很多读者第一次意识到的系统级陷阱: **在 2006 年的大多数桌面操作系统上, 即使应用显式调用 fsync, 也无法在断电时保护数据。**论文引用文献 [13] (McKusick 在 ;login: 上的文章) 与 [27] (Slashdot 那篇著名的《Your Hard Drive Lies to You》) 指出: 这不是 bug, 而是**有意识的、为了性能牺牲持久性的设计决定。**

具体机制: Linux 2.4 内核执行 fsync 时, 只是把数据写到了**硬盘自带的易失缓存 (drive cache)**, 而不是盘片。硬盘收到数据就报告「写完成」, 实际还在自己的 RAM 里排队, 并且会为了优化磁头路径而**重排写顺序**。此时断电, 缓存里的数据全部蒸发。更糟的是 (第 6.2 节的实验证明), 由于日志的 commit block 可能先于数据块落盘, ext3 的日志恢复机制会**把垃圾数据当成有效数据重放**, 导致已落盘的数据和元数据 (甚至超级块) 被覆盖损坏——fsync 不仅没保住新数据, 还毁了旧数据。

因此 MySQL 官方文档 (文献 [15]) 建议用户直接禁用硬盘缓存; MacOS X 与 Linux 2.6 提供了显式冲刷 drive cache 的机制 (即 write barrier), 但默认不开启, 因为性能损失太严重 (见 6.3: 每个事务两次同步冲刷缓存 → 慢 100 倍)。

📖 基础知识补充

drive cache (磁盘缓存)：机械硬盘电路板上的 RAM（本文的盘是 2 MB）。开启写缓存后，硬盘收到写请求立即应答，再在后台写入盘片，还会做电梯调度重排。它让硬盘「看起来」快很多，所以厂商默认开启——这就是「硬盘对你撒谎」的含义。

write barrier 的工作方式：文件系统在写日志 commit block 前后各放一个 barrier，等价于强制硬盘「把缓存里此前的所有写先落盘」。每次事务两次同步冲刷 = 两次磁盘旋转延迟，这就是同步+barrier 慢 100 倍的直接原因。

UDP：无连接的网络协议，发出去不保证到达，但开销极小。作者用 UDP 做断电实验的「外部观察者日志」正合适——它本身就是一次「外部输出」。

超级块 (superblock)：文件系统的「元数据中的元数据」，记录文件系统整体布局。超级块被垃圾覆盖 ≈ 整个文件系统报废，需手工 fsck 修复。

💡 核心洞见

Insight#7：「fsync 成功返回」≠「数据安全」——**存储栈的每一层都可能偷偷缓存**（页缓存 → 驱动 → drive cache），可靠性必须端到端验证（end-to-end）。作者用真实断电实验而不是代码走读来验证这一点，方法论上堪称典范：不要信文档，拉电闸试试。

Insight#8：注意责任链条：硬盘厂商为了 benchmark 数字默认开缓存，内核为了不丢性能默认不冲刷，于是「持久性」这个用户最朴素的需求在层层优化中被悄悄牺牲。系统论文常常要对抗这种「局部最优累积成全局错误」。

2.4 应用程序员被迫手写 group commit

因为默认异步抽象不可靠、同步抽象用不起，认真的应用只能自己在用户态实现可靠性协议：数据库自己写 redo log、自己调 fsync、自己做 group commit；邮件服务器、版本控制系统莫不如此。这带来三重问题：

- **重复造轮子**：每个应用都重新实现一遍事务持久化，质量参差不齐。
- **抽象泄漏**：应用必须理解 fsync、drive cache、write barrier 这些底层细节才能写对代码——而这些细节还会随内核版本和硬件变化。
- **保守提交损失性能**：应用的 group commit 只能看到进程内部，无法知道「这次提交之后外部世界是否真的在看」，只能保守地在每次对外输出前提交。第 5.7 节会看到，xsyncfs 因为掌握全局因果信息，能做出应用层做不到的更优决策。

📖 基础知识补充

redo log / WAL (write-ahead log, 预写日志)：数据库先把「我要做什么修改」顺序追加到日志并 fsync，再慢慢把修改应用到数据页。崩溃后重放日志恢复。WAL 是数据库 ACID 中 A（原子性）和 D（持久性）的基石。ext3 的日志、jbd 层本质也是 WAL。

ACID：事务四属性——原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。TPC-C 要求数据库提供完整 ACID。

💡 核心洞见

Insight#9：当大量应用都在用户态重复实现同一种机制（可靠性协议）时，说明**抽象被放错了层**。External synchrony 的本质是把「什么时候必须提交」的决策权从应用收回内核——因为只有内核能看到全局因果图（哪个输出依赖哪个事务）。这呼应了 Saltzer 的端到端原则的一个反向应用：有些决策恰恰只有下层才能做对。

2.5 问题形式化：我们到底想要什么

把动机收拢成一个清晰的问题陈述。我们想要一个本地文件 I/O 模型，同时满足：

1. **持久性**：任何外部观察者可见的效果，其依赖的修改必须已持久（断电也不丢）；
2. **因果顺序**：外部可见效果之间的因果关系，与同步执行时完全一致；
3. **性能**：接近异步 I/O，进程几乎不阻塞；
4. **零应用改造**：不要求应用插 fsync 或声明事务边界；
5. **不依赖特殊硬件**：不需要电池缓存或 NVRAM。

2006 年之前，没有任何系统同时满足这五条：同步 ext3 满足 1、2、4 但违反 3；异步 ext3 满足 3、4 但违反 1、2；Rio/Conquest 等硬件方案满足 1、2、3 但违反 5；事务文件系统满足 1、2 但违反 4（要标注事务边界）。External synchrony 宣称五者兼得——这就是「Rethink」的含义。

📖 基础知识补充

「外部观察者」 (external observer)：指用户、另一台机器上的程序等只能通过各种外设输出（屏幕、网络包、磁盘稳定状态）来观察计算机的实体。它是本论文的核心概念，第 3 章会形式化。

为什么「零应用改造」这么重要？ 一个需要改应用才能用的 OS 特性，部署成本是 $O(\text{应用数})$ ；一个内核透明的特性，部署成本是 $O(1)$ 。历史上事务文件系统（QuickSilver [24]、TABS [28]、Locus [32]）都没能成为主流，需要改造应用是重要原因之一。

 核心洞见

Insight#10: 好的系统论文往往在开头就能把「设计空间地图」画出来：每个已有方案占了哪些格子、空着的格子是什么。本章的五条要求就是这张地图——空格子（全满足）正是 xsyncfs 的位置。阅读论文时学会先问「这张地图上还有别的空格吗？这个空格真的是空的吗？」是评判创新性的基本功。

第 3 章 External Synchrony 模型

The External Synchrony Model

3.1 核心思想：换保证对象，不换保证内容

论文对 external synchrony 的定义值得逐字咀嚼：“External synchrony provides the reliability and simplicity of synchronous I/O, while closely approaching the performance of asynchronous I/O.” 关键创新在于对「同步 I/O 抽象」的重新解读：

- **传统解读**：同步 I/O 是一种**实现方式**——阻塞调用进程直到修改提交（文献 [26] 《Operating System Concepts》给出的教科书定义正是如此）。
- **本文解读**：同步 I/O 是一组**提供给文件系统客户端的保证**——持久性 + 因果顺序。既然它是「保证」，那么问题就变成：**这个保证到底应该给谁？**

异步 I/O 的做法是大幅削弱保证来换性能；external synchrony 的做法是**保证原封不动，但更换保证的接收方 (clients)**：不再保证给「发起调用的应用进程」，而是保证给「观察应用输出的外部实体」。这个替换之所以可行，是因为外部实体本来就无法直接观察应用内部状态——它只能通过外部输出观察。只要管住输出，应用进程跑得多快、状态变得多早，外部世界根本无从知晓。

📖 基础知识补充

抽象 (abstraction) 与实现 (implementation) 的分离：计算机科学的基本方法论。同一个抽象（如「文件已保存」）可以有多种实现（同步阻塞 / 延迟提交+输出缓冲）。只要可观察行为一致，用户无法区分。本文是把这条方法论应用到「同步性」这个过去被认为只能按实现定义的概念上。

类比：快递「签收」的定义。旧定义：快递员必须等收件人当面拆开验货才算签收（阻塞、慢）。新定义：只要包裹放进带监控的快递柜且柜门没关上前不通知收件人，那么从收件人视角看，「能拿到包裹」和旧定义完全等价——但快递员早就可以去送下一单了（不阻塞、快）。监控柜门 = 输出缓冲，柜门打开 = 事务提交。

💡 核心洞见

Insight#11：这是全文最锋利的一刀：「同步」过去被定义为**实现（阻塞）**，现在被定义为**可观察行为（外部不可区分）**。一旦接受行为定义，「阻塞进程」就从「语义的必要组成部分」降级为「众多实现手段之一」，而且是最笨的一种。这种「重新定义」式的创新在系统领域屡试不爽（虚拟机、容器、微内核都可如此理解）。

3.2 从 application-centric 到 user-centric

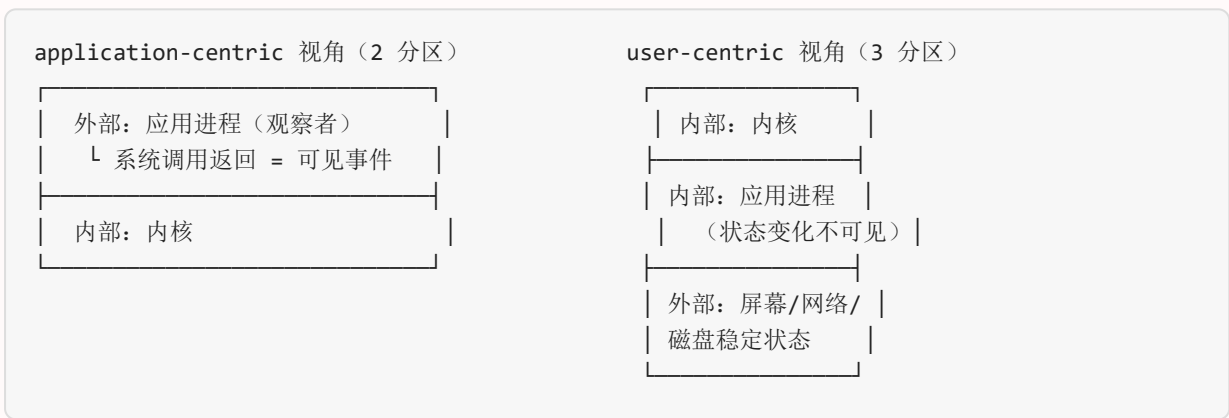
论文用「分区」的语言精确描述了这次视角转换：

传统的 application-centric（应用中心）视角把计算机系统分为两个区：内核是「内部状态」，用户态是「外部状态」。在这个视角下，**系统调用返回本身就是一次「外部可见事件」**——因为应用被视为操作系统的观察者。于是同步 I/O 必须在返回前完成提交，否则「观察者」（应用）就看到了未提交的状态。

本文的 user-centric（用户中心）视角指出：应用不是计算机系统真正的观察者，**用户（以及网络上的对端）才是**。应用状态只有通过外设输出（屏幕、网络等）才能被外部观察。于是分区变成三个：**内核 + 应用都是「内部状态」，外部接口才是「外部可见」**。在三分视角下，系统调用返回、寄存器变化、内存更新——这些都不算外部可见事件，操作系统无需为它们提供持久性保证。

这个视角为什么可以落地？因为**操作系统控制着所有对外设的访问**：应用想产生外部事件，必须直接或间接（如 mmap 一块映射到屏幕的设备内存）请求操作系统合作。操作系统只要把守这个唯一的出口，就能实施 user-centric 保证。

图 3-A：两种视角的分区对比（笔记自绘）



📖 基础知识补充

系统调用返回为什么是「事件」？ 在传统视角下，write() 返回成功就是一个事实广播给了应用。如果数据其实没持久，这个「事实」就是谎言。同步 I/O 用阻塞来保证「返回时不撒谎」。

mmap（内存映射）：把文件或设备直接映射进进程地址空间，读写内存即读写文件/设备，无需系统调用。它是操作系统「把守出口」的例外通道，所以论文专门讨论（共享内存、5.8 节）。

外部设备包括什么？ 屏幕（终端帧缓冲）、网络（socket 发包）、打印机等。注意：**磁盘提交也被算作外部输出**（3.4 节），这是论文一个精妙而关键的决定。

💡 核心洞见

Insight#12: 「观察者是谁」决定了「需要保证什么」。把应用从观察者降格为内部状态，系统就获得了在应用层面「随意折腾」的自由——只要出口把关得住。这与分布式系统中「客户端只关心外部一致性」、数据库中「用户只关心已提交读」一脉相承：**系统的契约只在边界上成立。**

Insight#13: 这个视角转换能成立，依赖一个结构性事实：OS 是应用与外部世界之间的**唯一中介**。凡是存在绕过中介的旁路（mmap 设备内存、RDMA、后来的 io_uring 注册缓冲区等），保证就会出现缺口——论文对共享内存的专门处理正是堵旁路的例子。

3.3 正确性定义：值相同 + Lamport 因果序相同

模型要严谨，必须回答「什么叫外部同步系统的行为与同步系统不可区分」。论文给出两个条件：外部输出（external output）与同步系统等价，当且仅当：

1. **(a) 值相同**：外部输出的值（values）相同；
2. **(b) 因果序相同**：输出按 Lamport 「happens-before」关系定义的因果顺序（causal order）相同。

条件 (a) 如何保证？ 外部同步 I/O 返回给应用的结果，与同步 I/O 会返回的结果完全相同：文件系统执行同步 I/O 会做的所有处理——参数校验、权限检查、修改易失（内存）态文件系统状态——一样不少，只是**不真的把修改提交到磁盘**就返回。由于应用看到的每个结果都与同步情形一致，应用之后产生的输出值自然一致。

条件 (b) 如何保证？ 操作系统必须保证：如果一次外部输出因果上跟随某次外部同步 I/O，那么该输出**绝不能在该 I/O 提交到磁盘之前被观察到**。论文 Figure 1 的例子：应用先改文件两次再向屏幕写消息，则第二次修改不得先于第一次提交，屏幕消息不得先于两次修改提交而显示。

📖 基础知识补充

Lamport happens-before (文献 [9], 1978)：分布式系统奠基性概念。事件 a happens-before b（记 $a \rightarrow b$ ），如果：(1) a、b 在同一进程内且 a 先发生；(2) a 是发送消息、b 是接收同一消息；(3) 传递性。它定义的不是「物理时间先后」而是「因果上是否可能互相影响」。两个无因果关联的事件是并发的（concurrent），谁先谁后无所谓。本论文把它搬到单机：文件修改、IPC、输出之间的因果链。

为什么「值相同」是免费得到的？ 因为应用逻辑没变、输入没变、每个系统调用的返回值没变，确定性应用的行为自然不变。风险只在于「时机」——所以真正需要机制保证的只有条件 (b)。

「不可区分」(indistinguishable)：分布式系统理论的标准概念。两个系统对某观察者不可区分，意味着该观察者的任何实验都无法分辨它们。这是「行为等价」的严格表述。

💡 核心洞见

Insight#14: 正确性定义的两分法（值 + 因果序）非常讲究：值靠「对应用完全诚实」免费获得，因果序靠「输出缓冲 + 依赖追踪」强制维持。这告诉我们设计协议时应先分清「哪些性质可以白拿、哪些性质必须付费」，然后只为后者设计机制。

Insight#15: 用 Lamport 因果序而非物理时钟来定义正确性，是刻意为之：物理序太强（不必要的约束会损失性能），「值相同」太弱（允许乱序提交，崩溃后状态不可解释）。因果序恰好是「外部观察者能感知的最强顺序」——保证到此为止，不多不少。

3.4 磁盘提交也是外部输出

论文中一个容易滑过去但极其关键的设计决定：**把磁盘提交（disk commit）本身视为外部输出**。理由：提交改变了文件系统的稳定镜像（stable image）；如果系统崩溃重启，稳定镜像的变化就对外可见了。而操作系统无法控制崩溃何时发生，所以必须保守地把每次提交都当作「可能被观察的外部事件」。

这个决定直接导出一条硬约束：**提交顺序必须遵守因果序**。若操作 A 因果先于操作 B，则 A 的提交必须先于 B 的提交（或与 B 原子地一起提交）。否则崩溃恰好落在两个提交之间时，重启后 B 可见而 A 不可见——违反因果序。这也是为什么 xsyncfs 必须用 ext3 的 journaled 模式（5.4 节）：ordered 模式不保证数据块的落盘顺序。

📖 基础知识补充

stable image（稳定镜像）：文件系统在持久介质上的状态，即崩溃重启后你看到的那个文件系统。与之相对的是内存中的易失状态（页缓存里的脏数据）。

为什么磁盘提交「外部可见」？ 想象崩溃是一台「随机的照相机」：它随时可能拍下稳定镜像的照片并展示给重启后的用户。既然拍照时机不可控，每一次稳定镜像的变化都必须已经是「合法可展示」的——这就是把提交当外部输出的含义。容错领域称之为 recovery line 的一致性。

💡 核心洞见

Insight#16: 把磁盘提交纳入「外部输出」是整个模型的点睛之笔——它把「崩溃恢复后的一致性」与「运行时的外部可见性」统一进了同一个因果序框架：**崩溃不过是一次时机未知的外部观察**。于是「防崩溃丢数据」不再需要一套单独的推理，它自动从输出排序规则中导出。好的抽象的标志就是：边界情形自动落入一般规则。

3.5 总体工作流程（Figure 1 精讲）

论文 Figure 1 用一个单线程应用的例子对比两种模型：应用做两次文件修改，然后向屏幕输出一条消息。

左图（同步 I/O）：时间线上有三个外部输出——两次磁盘提交和一次屏幕显示。应用每改一次文件就被阻塞，等磁盘提交完成才返回；第二次修改只能在第一次提交后开始；最后 printf 时数据早已安全。总耗时 $\approx$ 两次磁盘事务延迟 + 计算时间。

右图（external synchrony）：

1. 应用发起修改 1：xsyncfs 校验、把修改加入当前活动事务（active transaction），**立即返回**；同时给进程打上 commit dependency（提交依赖）标记；
2. 应用紧接着发起修改 2：同样立即返回，修改 2 加入同一事务（group commit 自动发生）；
3. 应用执行 printf 想往屏幕写：内核发现该进程带有未解决的 commit dependency，于是**缓冲这次输出**（不显示），**但不阻塞进程**——应用继续往下跑；
4. 输出被缓冲这件事触发 output-triggered commit：事务立即开始提交；
5. 事务落盘完成 $\rightarrow$ commit dependency 解除 $\rightarrow$ 内核把缓冲的屏幕输出真正送往显示设备。

外部观察者看到的顺序：修改 1、2 原子可见，然后屏幕消息——与同步情形**完全一致**；而应用总耗时 $\approx$ 计算时间 + 一次（而非两次）磁盘事务延迟，且等待期间应用还可以继续做别的事。

图 3-B: Figure 1 的时间线重绘（笔记自绘）

同步 I/O: [修改1]—阻塞等待提交—[修改2]—阻塞等待提交—[输出]→
总耗时 $\approx 2 \times$ 磁盘延迟

外部同步: [修改1][修改2][输出请求→缓冲]—后台提交事务—[输出真正显示]→
总耗时 $\approx 1 \times$ 磁盘延迟（且计算可与提交重叠）
外部可见顺序: {修改1∧修改2} → 屏幕输出 ← 与同步完全一致

📖 基础知识补充

taint（沾染）：论文用语，指给进程打上「携带未提交依赖」的标记。类似动态污点分析（taint tracking）的思想：依赖像墨水一样随数据流传播。

活动事务（active transaction）：ext3/jbd 概念，任意时刻最多一个正在收集修改的事务。提交开始后再来的修改进入新事务。详见 5.3 节。

人眼感知阈值：约 50–100 ms。论文指出输出被缓冲的时间通常不超过一次事务提交（几到几十毫秒），低于人类感知阈值（文献 [5] 引用 50 ms），所以用户不会察觉延迟——这是整个方案对交互体验无害的关键经验事实。

💡 核心洞见

Insight#17: 整个方案的「戏法」可以总结为一句话：**把等待磁盘的时间，从「阻塞应用」挪到「阻塞输出」**。应用等不起（它在干活），输出等得起（反正用户感知阈值是 50 ms，磁盘提交只要几毫秒）。这是一次经典的「把延迟转移到不敏感路径上」的优化，与乱序执行、写缓冲、延迟绑定等同属一族。

Insight#18: 注意 group commit 在这里是**免费副产品**：因为应用不再被阻塞，后续修改自然源源不断地涌入活动事务，文件系统第一次拥有了「攒批」的机会。同步语义的性能死结（Insight#5）就此解开——不是通过更聪明的调度，而是通过解除阻塞。

第 4 章 设计 (Design)

Design Overview: 原则、性能优化、提交时机与局限

4.1 两条设计原则

为什么需要：先立原则再谈机制。 设计一个「替代同步 I/O」的模型，最大的风险是偷偷削弱保证却不自知。作者因此在动笔设计机制之前，先立下两条元原则，后续所有机制都可以回溯到这两条原则上接受检验。

原则一：用外部可观察行为定义，而非用实现定义。 同步 I/O 的传统定义是「阻塞到 I/O 完成」（教科书定义，文献 [26]）；本文的定义是「系统产生的外部输出，无法与同步 I/O 可能产生的输出相区分」。行为定义把「什么是对的」与「怎么做到的」解耦，为后面所有非常规机制（缓冲、延迟提交、依赖传播）留出合法性空间。

原则二：应用状态是系统内部属性。 应用状态不被外部实体直接观察，因此操作系统无需把应用状态的变化（如系统调用返回）当作外部输出。这一条是 3.2 节 user-centric 视角的直接复述，它是「允许应用在数据未提交时继续运行」的许可证。

📖 基础知识补充

设计原则 (design principles) 在系统论文中的作用：原则是「机制的灵魂」。机制复杂而原则简单，读者先认同原则，再核对每个机制是否真的服从原则。这也是评审论文的方法：先问原则是否成立，再问机制是否忠实。

「用行为定义」先例：分布式事务的可串行化 (serializability) 也是行为定义——不关心内部怎么调度，只要求外部效果等价于某个串行执行。本文相当于把同样的思路引入文件系统同步语义。

💡 核心洞见

Insight#19：两条原则一攻一守：原则一负责「打开设计空间」（凡行为等价者皆可为），原则二负责「划出边界」（只有外部输出需要被管束）。所有后续机制——group commit、输出缓冲、依赖传播、output-triggered commit——都是这两条原则的逻辑推论。

4.2 性能优化之一：group commit 与 value logging

为什么需要这个设计？ 问题：磁盘随机写的固定成本极高（约 10 ms），每次修改单独提交，吞吐被物理上限锁死在约 100 次/秒。约束：因果序要求「B 的效果可见 $\Rightarrow$ A 的效果可见」。权衡：要么为每次

修改付一次磁盘钱（同步 I/O 的老路），要么想办法让多个修改共享一次磁盘写。方案：**把多个修改合并为一个原子事务提交（group commit）**。

group commit 的三重收益：

- **摊薄提交成本**：所有修改用一次顺序磁盘写（日志 append）完成提交；
- **写合并（write coalescing）**：同一磁盘块被多次修改时，日志里只保留最新版本——ext3 的这个特性叫 **value logging**（值日志，相对 record logging 记录每次操作）；
- **删除抵消写入**：如果一个临时文件在同一事务窗口内被创建又删除，它的块**根本不用写盘**。编译类负载（大量临时中间文件）因此收益巨大。

更妙的是正确性自动成立：事务是原子提交的，所以「第二次修改的效果可见 $\Rightarrow$ 第一次的效果可见」天然满足。而同步文件系统对单线程应用**结构性无法做 group commit**（Insight#5 的因果死锁）——external synchrony 因为不阻塞应用，修改自然汇聚到活动事务中。

📖 基础知识补充

value logging vs record logging：record logging 记录「做了什么操作」的完整历史（A 块写值 1，A 块写值 2.....）；value logging 只记录每个块的最终值（A 块 = 2）。后者日志更小、合并更狠，代价是日志反映的是「状态」而非「操作序列」。ext3 用 value logging。

日志（journal）为什么用顺序写？机械盘顺序写几乎不需寻道，吞吐可达 50–100 MB/s，而随机写只有约 100 次/秒 $\times$ 4 KB $\approx$ 0.4 MB/s。把随机修改汇聚成日志顺序写，是 WAL 类系统的根本性能来源。

事务（transaction）与原子性：jbd（ext3 的日志层）把一组修改打包成事务，先写日志数据块、最后写 commit block。恢复时只重放有完整 commit block 的事务——要么全有、要么全无。

💡 核心洞见

Insight#20：group commit 在数据库界是三十年前的老技术（Cedar 文件系统 1987 年就用过，文献 [6]），本文的贡献不在发明它，而在**让它对单线程、无改造的应用也自动生效**——这只有解除阻塞才能做到。老机制 + 新前提 = 新能力。

Insight#21：注意「删除抵消写入」揭示的深层事实：**大量磁盘写本来就是浪费**（临时文件、被覆盖的块）。异步系统靠延迟天然获得这个优化，同步系统则永远享受不到。External synchrony 把「延迟」这个性能利器从异步阵营夺了回来，同时没接下「不可靠」的包袱。

4.3 性能优化之二：缓冲输出而非阻塞应用

为什么需要？ 问题：因果序约束要求「依赖于未提交修改的输出不得先于提交被观察」。最直白的实现是：应用发起外部输出时，把它阻塞到提交完成。约束：阻塞意味着应用的吞吐被拖回磁盘速度——等于把同步 I/O 的病从写路径转移到输出路径，换汤不换药。权衡：输出动作本身往往是异步的（屏幕显示、网络发包都不需要应用等结果），应用根本不需要「等输出完成」的语义。方案：**缓冲输出 (buffered output)**，**放行进程**。

具体做法：进程产生外部输出时，内核不把它送到设备，而是暂存在内核缓冲区里，输出继承生成它的进程的所有 commit dependency；进程立即继续执行。当某个缓冲输出依赖的所有事务都提交完毕（所有 dependency 被解除），内核才把输出真正释放到目标设备。

这个设计与 3.4 节的原则完美咬合：外部观察者看到的输出流，无论值还是顺序，都与同步执行不可区分；而应用的执行几乎不受影响——它被「欠债」的不是 CPU 时间，而是「输出的可见性」。

基础知识补充

输出为什么天然可以缓冲？ 因为大多数输出通道本来就是异步的：写终端是往帧缓冲/pty 写数据，写 socket 是往发送缓冲区写数据，应用从系统调用返回时数据通常也还没真正到达对端。内核在缓冲区多捂几十毫秒，语义上无可察觉。

输出缓冲的历史渊源：容错领域的 output commit problem（文献 [3][30]）早就提出「消息发出前必须保证其依赖的状态已持久」；Manetho [4]、Rio 相关工作中的延迟发送 [12] 都是同类技术。本文把它从「分布式容错协议」移植为「单机文件系统性能机制」——详见第 7 章。

核心洞见

Insight#22：「缓冲输出而非阻塞进程」生效的前提是**输出通道的异步性**——这是一个常被忽视的环境红利。设计系统时值得盘点：你的瓶颈路径上，有哪些环节的使用者其实不需要同步语义？凡是「调用者不等结果」的环节，都是延迟可以被隐藏的地方。

4.4 commit dependency 与因果依赖传播

为什么需要？ 问题：输出缓冲要求内核知道「这次输出依赖哪些未提交事务」。但依赖不是静态的——进程会通过 IPC（管道、文件缓存、UNIX socket、共享内存）互相影响，依赖会**流动**。约束：内核必须能拦截所有因果传播通道；对无法追踪的通道（如某些 ioctl）必须有安全兜底。权衡：精确追踪所有数据流（如全系统污点分析）成本太高；不追踪则保证失效。方案：**在内核对象粒度上追踪 commit dependency 的传播**。

传播规则（论文原文归纳）：

- 进程写文件 → 进程继承一个指向「包含该修改的未提交事务」的 commit dependency；

- 带依赖的进程通过系统调用修改另一内核对象（进程、管道、文件、UNIX socket 等）→ 该对象被标记同样的依赖；
- 进程观察到带依赖的内核对象的状态 → 进程继承该对象的依赖；
- 带依赖的进程执行内核无法追踪因果的系统调用（如 `ioctl`）→ **阻塞**该进程直到其修改提交（兜底，保证不泄漏）；
- 任何外部输出继承进程的依赖 → 缓冲到最后一个依赖解除。

这套规则保证了依赖关系沿因果链闭合传播：任何「可能」携带未提交信息的实体都被打上标记，宁可保守（多缓冲一点输出），不可遗漏（泄漏未提交状态）。

📖 基础知识补充

IPC（进程间通信）：管道（pipe）、UNIX socket、信号、共享内存、消息队列等。对本文而言，IPC 是「因果污染的传播渠道」：A 进程把未提交数据写进管道，B 进程读走，B 的输出从此也依赖那个事务。

内核对象：文件、管道、socket、进程本身在内核中的数据结构。在对象粒度打标记的好处是实现集中——系统调用天然经过内核，标记随调用传播。

ioctl：设备特定的杂项系统调用，语义由驱动自定义，内核无法统一理解其数据流向，所以只能保守阻塞。这是「不可追踪通道一律兜底阻塞」原则的例子。

为什么保守是正确的？ 依赖追踪是典型的「宁可错杀不可放过」场景：多传播一个依赖，代价只是输出多等一次提交；漏传播一个依赖，则崩溃后外部世界看到了不可能的状态——正确性破产。

💡 核心洞见

Insight#23：依赖传播的本质是把 Lamport 的因果链**物化为内核里的标记**。理论的因果序是抽象的偏序关系；工程上它变成「对象上挂 dependency、syscall 时搬运、输出时检查、提交后清除」四个机械动作。读系统论文时，留意这种「理论概念 → 四个机械动作」的降维过程，它往往是实现章节的主线。

Insight#24：对不可追踪通道（`ioctl`）选择「阻塞」而非「禁止」或「放任」，是一个漂亮的工程决策：**正确性不妥协（绝不泄漏），可用性最大化（已知通道全追踪），未知通道付出可预期的代价**。这让系统对新设备/新 syscall 天然安全——默认安全（fail-safe）而非默认放行。

4.5 Deciding when to commit: output-triggered commits

为什么需要？ 问题：group commit 存在经典的吞吐-延迟权衡——攒得越久，单次提交摊得越薄（吞吐高），但等待中的输出被压得越久（延迟高）。约束：在本模型中延迟格外敏感，因为输出不释放用户就看不见结果；但一味低延迟（每次修改立即提交）又退化成本地 I/O。权衡：固定的提交间隔（ext3

的 5 秒) 或固定的提交触发 (每次修改即提交) 都是**瞎猜**——系统真正需要的是「按需提交」: 没人看的时候尽管攒, 有人等的时候立刻交。方案: **output-triggered commits (输出触发提交)**。

机制: 用外部输出与文件修改之间的**因果关系**决定提交时机。只要还没有任何输出因果依赖于某个事务, 延迟提交它不改变系统的可观察行为——此时放心攒批, 优化吞吐。一旦有输出被缓冲 (或某进程因 `ioctl` 被阻塞) 且它依赖于活动事务, 立即触发该事务的提交——最小化外部观察者的等待。

一个关键论断: *“Latency is unimportant if no external entity is observing the result.”* (如果没有外部实体在观察结果, 延迟就不重要。) 提交一个没人等的事务, 只是白白浪费 IOPS。这使得文件系统的行为**自适应于系统的可观察行为**:

- 用户把输出重定向到文件 (无人实时观看) → 按 ext3 默认 5 秒攒批, 吞吐最优;
- 用户盯着屏幕看输出 → 输出一被缓冲就触发提交, 延迟最优;
- 用户用 `tail` 另开一个窗口读输出文件 → Speculator 通过内核数据结构追踪到 `tail` 对事务的因果依赖, `tail` 试图上屏时触发提交——依然延迟最优。

📖 基础知识补充

吞吐-延迟权衡的直观例子: 公交车发车间隔。等够 50 人再发车, 人均成本最低 (吞吐高) 但先到的人等得久 (延迟高); 来人就发车, 体验好但成本爆炸。固定间隔调度 vs 按需调度之争在操作系统的磁盘调度、网络 Nagle 算法、数据库提交策略中反复出现。

为什么「没有观察者时延迟不重要」是安全的? 因为外部同步的正确性只定义在**外部可观察行为**上 (原则一)。不可观察的内部延迟不改变任何外部行为, 自然不违反任何保证。这是原则一的又一次「分红」。

eager commit (急切提交): 作为对照的策略——文件系统一被修改就触发提交。6.7 节会量化 output-triggered 相对 eager 的收益 (PostMark 快 14%、MySQL 单客户端快 35%)。

💡 核心洞见

Insight#25: output-triggered commits 的洞见可以推广为一个普适原则: **提交的正确时机不是一个时间常量, 而是一个因果事件**。「多久提交一次」这个问题本身问错了; 正确的问题是「谁正在等这次提交」。所有「定时 flush」类机制 (页缓存回写、日志组提交、metrics 聚合上报) 都可以用这个视角重新审视。

Insight#26: 这个策略让系统行为随「可观察性」自适应, 是一种**以观察者为中心的资源调度**: 没人看时把资源让给吞吐, 有人看时把资源让给延迟。它有点像「如果森林里没有观察者, 树倒下就不必发出声音」——而且这不是偷工减料, 因为正确性定义本来就只覆盖可观察行为。

4.6 Limitations: 作者自己承认的局限

诚实的论文会自己列出局限。论文列了三个，每一个都值得细读：

局限一：灾难性介质故障的应用级恢复变复杂了。内核在把修改写入文件缓存前会做校验，但随后物理写盘仍可能失败。坏块等小错误由磁盘/驱动层处理，但灾难性介质故障（catastrophic media failure）很少能在这些层面被掩盖。理论上同步挂载的文件系统可以把这种错误传播给应用（write 返回失败，应用决定怎么办）；但文献 [20]（Prabhakaran 等的 IRON file systems, SOSP'05）的调查表明：现实中 ext3/jbd 和 NTFS **根本不检测或不上报写错误**，ReiserFS 则直接 kernel panic——也就是说同步系统在这件事上的「理论优势」在现实中并不存在。作者还讨论了另一种可能：用 Speculator 在修改文件系统前给进程做 checkpoint，灾难发生时回滚进程并通知——但**明确拒绝了这个方案**，因为它会大幅增加复杂度、严重损害性能，而且灾难性故障本来就更适合由操作系统统一处理（panic 或把数据写往别处），而不是推给每个应用。

局限二：用户对「何时落盘」可能有时间预期。按纯模型定义，一个从不产生输出的应用的数据可以被无限期延迟提交；一旦崩溃，丢失的工作可能很多。xsyncfs 因此兜底：**即使没有输出，也每 5 秒强制提交一次**（与异步 ext3 的默认提交间隔相同）。这是用一条固定的最坏情况上界，补偿纯因果触发在「零输出」极端下的失速。

局限三：跨文件系统的原子提交困难。两个不同文件系统上的修改无法用单个磁盘事务原子提交。可能的方案是共享日志或两阶段提交（two-phase commit），但都复杂。作者选择更简单的方案：当一个进程带着文件系统 A 的依赖想去改文件系统 B 时，**阻塞它直到 A 的修改提交**。Speculator 把每个依赖映射到具体文件系统，写之前校验进程的依赖是否只涉及目标文件系统。

基础知识补充

kernel panic：内核检测到无法恢复的内部错误时主动宕机，防止以损坏状态继续运行造成更大破坏。「宁可panic也不静默写坏数据」是一种设计哲学（与文献 [20] 提倡的 IRON——internal robustness——一致）。

两阶段提交（2PC）：分布式事务协议，先让所有参与方「准备」（投票可否提交），再统一「提交」。能保证跨节点原子性，但协调者单点、阻塞、慢。

checkpoint + rollback（检查点+回滚）：定期保存进程完整状态，出错时恢复到检查点重新执行。Speculator 原生支持它（5.1 节），但 xsyncfs 特意只用依赖追踪不用回滚——这是 5.2 节「降维」的核心。

💡 核心洞见

Insight#27: 局限一的处理展示了「用证据拆解假想敌」的论证艺术：批评者说「同步系统能把介质错误告诉应用」，作者引用文献 [20] 的实证调查指出**现有同步系统根本做不到这一点**——被比较的「对手优势」是纸面上的。做系统设计时，先核实对照组的真实能力，再为差异付费。

Insight#28: 拒绝 checkpoint+rollback 的理由值得背诵：为一个**罕见且归属存疑**的故障（灾难性介质损坏）让所有正常路径背上 checkpoint 的复杂度与性能税，是糟糕的工程经济学。「优雅地处理罕见故障」与「让常见路径保持干净」冲突时，作者选择后者并用 panic/异地写兜底——分层处理故障，而非就地处理。

Insight#29: 5 秒兜底揭示了一个务实真相：**纯行为主义模型管不到「从未被观察」的状态**，而用户的心理预期（「我干了这么久活，总该存下来点什么吧」）是模型外的需求。理论模型给出机制，工程系统补上安全带——两者缺一都成不了产品。

第 5 章 实现细节 (Implementation)

Speculator、xsyncfs 与内核机制

5.1 Speculator 背景

为什么需要了解 Speculator? xsyncfs 不是从零写的——它的依赖追踪与输出缓冲机制复用自同一个课题组的前作 **Speculator** (文献 [17], SOSP 2005)。不理解 Speculator 就无法理解 xsyncfs 的实现选择 (以及 5.2 节的「降维改造」)。

Speculator 原本解决的是**分布式文件系统**的性能问题：远程操作慢，与其阻塞等待，不如**预测**操作结果 (比如预测文件没被改过)，然后：

1. 给调用进程做 checkpoint (用 copy-on-write fork 保存状态)；
2. 基于预测结果**投机执行** (speculative execution) 继续跑；
3. 预测对了 → 丢弃 checkpoint；预测错了 → 回滚到 checkpoint，重试操作。

为支撑投机执行，Speculator 向内核引入两个数据结构：

- **speculation object (投机对象)**：追踪所有依赖于「某次投机操作成败」的进程与内核状态；
- **undo log (撤销日志)**：每个被投机污染的内核对象各有一份，包含撤销投机修改所需的**状态**。例如投机进程写管道时，Speculator 在管道的 undo log 里记一条，指向写者依赖的 speculations；另一进程读该管道时，再给读者建一条 undo log 项，指向管道依赖的所有 speculations——依赖就这样沿 IPC 传播。

并且 Speculator 保证**投机状态绝不对外部观察者可见**：投机进程执行会产生外部输出的系统调用时，输出被缓冲直到投机结果确定；遇到既无法传播依赖也无法缓冲输出的系统调用，则阻塞该进程直到它变成非投机状态。

📖 基础知识补充

投机执行 (speculative execution)：与 CPU 分支预测同源的思想——与其等待确定结果，先按最可能的猜测往下跑，错了再回滚。CPU 用它隐藏流水线延迟，Speculator 用它隐藏网络延迟，xsyncfs 借用它的基础设施隐藏磁盘延迟。

copy-on-write (COW) fork：fork 时不真复制内存，父子进程共享物理页，任一方写入时才复制该页。用它做 checkpoint 很便宜：只 fork 不写入就几乎零成本。

undo log (撤销日志) 与 redo log 的区别：redo 记录「新值」用于崩溃后重放；undo 记录「旧值/依赖」用于回滚或 (在 xsyncfs 中) 记录依赖关系。注意：在 xsyncfs 里 undo log 退化成了纯依赖簿记，见 5.2。

💡 核心洞见

Insight#30: Speculator 提供了三件宝物：**因果依赖追踪、输出缓冲、checkpoint/rollback**。作者的关键洞察是：支持 external synchrony 只需要前两件，第三件（最贵的那件）可以扔掉。识别出「前作机制中哪个子集恰好支撑新问题」，是比从头造轮子高得多的研究品位。

5.2 从 speculation 到 synchronization：数据结构的降维

为什么需要改造？ 问题：Speculator 把依赖追踪、输出缓冲与 checkpoint/rollback 等功能捆绑在一起，而后者对 external synchrony 不仅无用，还有可观性能成本。约束：希望同一台机器既能跑投机分布式文件系统、又能跑外部同步本地文件系统。方案：**把 Speculator 中两类应用共用的功能分解 (factor out) 出来，修改接口让每个文件系统声明自己需要哪些特性。**

两类应用的对比揭示了「降维」的本质：

	投机执行 (Speculator 原场景)	外部同步 (xsyncfs)
输出何时可见	基于 正确性 ：依赖的投机都被证实正确后	基于 持久性 ：依赖的文件操作都提交到磁盘后
依赖标记	speculation object	commit dependency
继承依赖时要做什么	先 COW fork 做 checkpoint (可能回滚)	什么都不用做 (永不回滚)
undo log 内容	撤销修改所需的旧数据	只记依赖关系，不含回滚数据
多线程进程	checkpoint 是「棘手问题」 ([17, 21])	依赖传播即可，简单得多 (5.8 节)

具体数据结构关系：commit dependency 表示内核状态与某个未提交文件系统修改之间的因果关系；凡带一个或多个 commit dependency 的内核对象称为 **uncommitted** (未提交态)。进程写 xsyncfs → Speculator 标记进程为 uncommitted，并在进程与包含该修改的活动事务之间建一条 commit dependency；事务提交到磁盘 → dependency 移除；某条缓冲输出的所有 dependency 都移除 → 输出释放到设备；进程的最后一条 dependency 移除 → 进程重新标记为 committed。Speculator 在 commit dependency 与 undo log 之间维持与投机情形相同的 **many-to-many (多对多)** 关系——一个对象可被多个依赖污染，一个依赖也可污染多个对象——但 undo log 里**没有回滚数据**，只剩「谁依赖谁」的簿记，以及据此判断「哪些缓冲输出可以安全释放」。这份简化还带来了额外红利：无回滚需求意味着可以支持更多种进程间交互（例如多线程与共享内存，5.8 节）。

📖 基础知识补充

many-to-many 关系的含义：依赖图是稠密的——进程 P 可能依赖事务 T1、T2（写了两个文件）；事务 T1 又可能污染进程 P、Q、管道 R（多人写同一文件、管道被污染）。用 undo log 项作为「边」来表达这张二分图，是 Speculator 的核心数据结构设计。

为什么「永不回滚」能省掉 checkpoint？ 投机的风险是「猜错要复原」，所以必须先存快照；而 external synchrony 中文件修改**一定会提交**（只是早晚），进程看到的结果永远是「对的」，不存在「错了要重来」的路径。风险消失，保险随之取消。

💡 核心洞见

Insight#31：这是一次教科书级的「机制瘦身」：**从投机到同步，语义负担从「可能出错」降为「必然正确」**，于是 **checkpoint、回滚数据、多线程快照这一整套重型机制整体蒸发**，只剩依赖簿记与输出缓冲。复杂性分析能力——说清「哪个前提砍掉了哪堆机制」——是系统研究者最重要的内功之一。

Insight#32：「把输出可见性的判据从正确性换成持久性」这个对照（表格第一行）是理解全文的钥匙：Speculator 问「这个结果对吗？」，xsyncfs 问「这个状态稳吗？」。两个问题的追踪机制同构，答案来源不同——这就是「factor out 共用功能」的精确含义。

5.3 xsyncfs：对 ext3 的修改点

xsyncfs 是在 Linux 2.4 的 ext3 上改造的。先回顾 ext3/jbd 的事务模型（这对理解修改点必不可少）：

- 许多文件系统操作的修改可被合并进一个**复合日志事务（compound journal transaction）**，原子提交；
- 任意时刻至多一个**活动事务（active transaction）**接收新修改；
- 活动事务的提交由三类事件触发：日志空间耗尽、应用显式同步（如 fsync）、事务中最老的修改超过 5 秒；
- 事务开始提交后，下一个修改触发创建新活动事务；任意时刻至多一个事务在提交中，新事务必须等前一个提交完成。

xsyncfs 对 ext3 的修改点（论文明确列举）：

1. 新建日志事务时**通知 Speculator**——Speculator 由此能追踪依赖于这个未提交事务的所有状态；
2. 有新修改加入事务时通知 Speculator；
3. 事务提交时通知 Speculator（用于解除 dependency、释放缓冲输出）；
4. 默认挂载模式改为 **journald**（理由见 5.4）；

5. 使用 **write barrier** 保证断电持久性（理由见 5.5）；

6. 接受 Speculator 的回调（callback）以支持 output-triggered commits（5.6）。

也就是说，ext3 的事务引擎（jbd）几乎原样保留，xsyncfs 主要做的是在 jbd 的生命周期事件上挂钩子（hooks），把事务的存在、成长与消亡同步给 Speculator 的依赖图。

图 5-A：Figure 2 数据结构变化示意（笔记自绘）

(a) 进程 1234 完成 3 次文件操作，每次操作后都向屏幕输出：



(b) T1 提交完成后：输出1 释放到屏幕；其 commit dependency 被丢弃；输出2、3 仍缓冲，等待 T2 提交。

📖 基础知识补充

jbd (Journaling Block Device)：ext3 的日志层，独立于 ext3 主体，负责事务的创建、日志写入、提交与恢复。理解 jbd 就理解了 ext3 的持久性机制。

为什么「至多一个活动事务」？ 单活动事务简化了并发控制：所有修改追加到同一事务，无需处理事务间冲突；代价是新事务要等旧事务提交完（提交是串行的）。xsyncfs 的 group commit 收益正来自这个串行管道。

回调 (callback)：反向的函数调用——下层（文件系统）注册函数，上层（Speculator）在事件发生时调用它。这里用于「输出被缓冲/进程被阻塞」时通知文件系统「该提交了」。

💡 核心洞见

Insight#33：xsyncfs 对 ext3 的侵入极小——**没有重写日志格式、没有改磁盘布局，只是在事务生命周期上挂了通知钩子**。这大幅降低了对既有稳定代码的破坏风险，也说明作者对「关注点分离」的纪律性：持久化交给 jbd（久经考验），因果管理交给 Speculator，两者之间只通消息不共享逻辑。

5.4 为什么必须用 journaled 模式而非 ordered 模式

为什么需要这个选择？ 问题：ext3 默认 ordered 模式（只把元数据进日志，数据块直接写原位）性能更好，为什么 xsyncfs 偏要用更贵的 journaled 模式（数据+元数据都进日志）？约束：3.4 节的硬约束——提交顺序必须遵守因果序（B 不得先于 A 提交）。权衡与论证：

journaled 模式保证顺序性的论证是构造性的：操作 A 完成时，A 的所有修改**已经在日志里**；而 B 因果跟随 A（B 开始时 A 已完成），所以 B 的修改进入日志时 A 必然已在日志。于是只有两种情形：(1) B 与 A 在同一事务——原子提交，顺序性自动成立；(2) A 所在事务先于 B 所在事务开始提交——提交串行化（5.3 节）保证 A 先落盘。两种情形都安全。

ordered 模式则不保证：数据修改不进日志，内核可以以任意顺序把 A、B 的脏数据块写回原位，只要「数据先于对应元数据日志提交」即可。于是 B 的数据完全可能在崩溃后可见而 A 的不可见——因果序破产。

📖 基础知识补充

journaled 模式的代价：所有数据写两遍（日志一遍 + 原位一遍），理论写吞吐减半。但作者的实验发现 journaled 在每个基准上都比 ordered **快**——因为日志顺序写+写合并的收益超过了双写的成本（6.1 节脚注了这一反直觉结果）。

ordered 模式的真正承诺：只有「元数据一致性」——崩溃后目录结构不损坏，但不承诺你文件里的数据是哪一版。它的设计目标是「不丢文件系统」，不是「不丢数据顺序」。

💡 核心洞见

Insight#34：这个选择说明：**正确性需求会反过来挑选底层机制**。ordered 模式对「元数据不坏」够用，但对「因果序可见性」不够；journaled 模式的「修改完成时必已入日志」这一性质恰好是因果论证的支点。选型不是比参数，而是看哪个性质恰好支撑你的不变式（invariant）。

5.5 write barrier 移植到 2.4 内核

为什么需要？ 2.3 节的断电分析表明：不冲刷 drive cache，一切持久性承诺都是空话。约束：Speculator 运行在 Linux 2.4 内核，而 write barrier 是 2.6.11 才有的特性；直接禁用 drive cache 也能保证持久性，但会丧失缓存带来的读加速与写聚合（6.1 节实测 write barrier 在所有场景都快于禁用缓存）。方案：**把 write barrier 反向移植（backport）到 2.4 内核**，并让 xsyncfs 在写每个事务的 commit record 前后用 write barrier 冲刷 drive cache。

这样，xsyncfs 成为 Figure 3 中四种配置里唯一同时回答两个「Yes」的非阻塞方案：write 时数据持久（Yes）、fsync 时数据持久（Yes）——详见 6.2 节的持久性矩阵。

📖 基础知识补充

backport (反向移植)：把新版本内核的特性移植回旧版本。工业界常见（企业版内核生命周期长），研究原型中少见——作者为了把实验做实在真实 2.4 系统上而做了这份脏活。

commit record / commit block：日志事务的「封口」标记。恢复时只认有 commit block 的事务，因此 commit block 必须最后落盘——write barrier 就是保证这个「最后」对物理盘片也成立，而不只是对 drive cache 成立。

💡 核心洞见

Insight#35：write barrier 的移植提醒我们：**存储栈的持久性是一个端到端属性，任何一层偷偷缓存都会击穿全局承诺**。内核→驱动→硬盘缓存→盘片，链条上每一环都要么有序要么可冲刷。今天 NVMe 的 FLUSH 命令、数据库的「disk persistence」检查清单，都是同一原则的延续。

5.6 Output-triggered commits 的实现：Speculator 回调

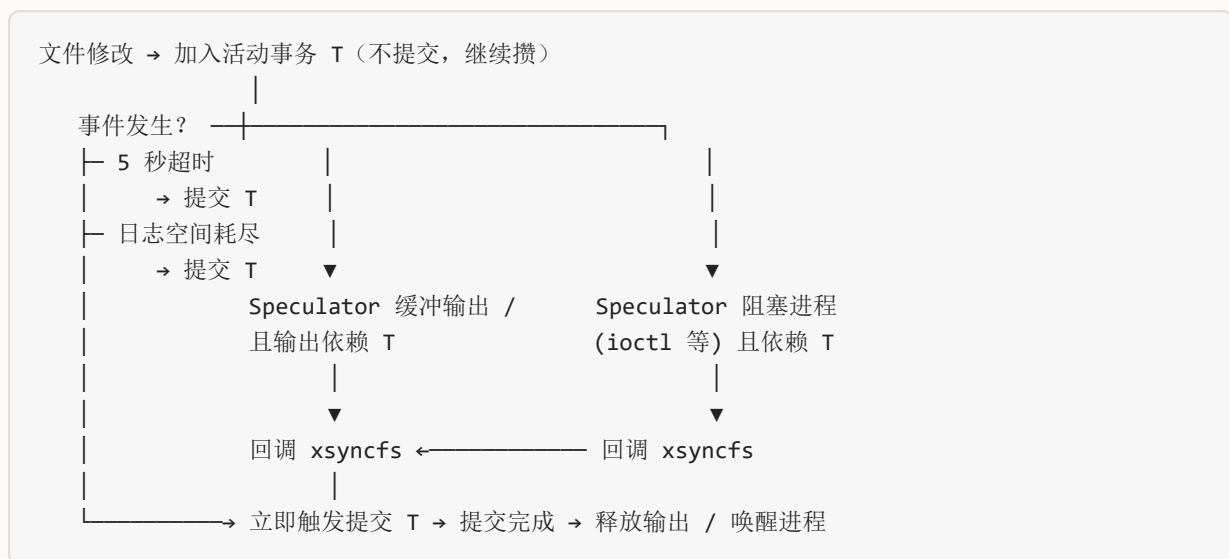
为什么需要？ 4.5 节给出了策略，这里回答「机制上怎么落地」。如果沿用 ext3 默认策略（每 5 秒提交），吞吐很好，但用户可能被迫等最多 5 秒才看到输出——对交互应用不可接受。

实现：作者修改 Speculator，让它在两个时刻回调 xsyncfs：

1. Speculator **缓冲**了某个依赖于活动事务的进程的输出时；
2. Speculator **阻塞**了某个进程（因其执行了无法追踪依赖的系统调用，如 ioctl）且该进程依赖活动事务时。

xsyncfs 平时沿用 ext3 的 5 秒提交策略；一旦收到上述回调，立即触发活动事务的提交。由此得到 4.5 节描述的自适应行为：输出重定向到文件时 5 秒攒批（吞吐优先）；输出上屏时立即提交（延迟优先）；另一个程序（如 tail）通过文件缓存读到未提交数据时，依赖沿内核数据结构传到 tail，tail 上屏同样触发提交。

图 5-B: output-triggered commit 的决策流程 (笔记自绘)



基础知识补充

tail: Unix 工具, 显示文件末尾内容, `tail -f` 可持续跟踪文件增长。论文用它举例说明依赖可以穿过「另一个进程读文件」这条间接路径传播——即使写者自己不输出, 读者的输出也会触发提交。

为什么缓冲输出才触发提交, 而不是写文件就触发? 因为没人观察时提交纯属浪费 IOPS (Insight#25)。回调机制让文件系统第一次能「感知」自己正被观察——这是把「可观察性」变成运行时信号的巧思。

核心洞见

Insight#36: 回调方向的选择很讲究: 是 Speculator (因果图的持有者) 通知 xsyncfs (磁盘的持有者), 而不是 xsyncfs 定期去轮询。**事件驱动而非轮询**, 让提交延迟精确等于「第一个等待者出现的时刻」, 不多不少。凡是「时机敏感 + 触发源分散」的策略, 都值得考虑这种「依赖图驱动」的反向通知结构。

5.7 Rethinking sync: sync/fdatasync 的处理

为什么需要重新思考? 异步文件系统提供 `sync/fdatasync` 等显式同步操作给有持久性需求的应用; 同步文件系统中这些操作冗余 (所有操作本已同步); 而外部同步文件系统中, 保证已经默认覆盖所有操作——但这些调用还需要一点「额外支持来最小化延迟」: 在终端敲 `sync` 的用户希望命令尽快完成。

处理方案（本节标题 Rethinking sync 的点题之处）：xsyncfs 从 VFS 层收到 sync 调用时，**不做任何磁盘写**，只在调用进程与活动事务之间建一条 commit dependency，然后**几乎瞬时返回**。之后如果发生可见事件（比如 sync 进程退出、shell 提示符打印——这本身就是屏幕输出），Speculator 的回调会触发 xsyncfs 提交活动事务。于是：sync 命令的「完成」对用户可见的时刻，恰好是数据真正提交之后——语义分毫未损，进程却一秒没阻塞。

由此引出更宏观的简化：**程序员可以按「同步挂载」的简单语义编程，却不用付同步的代价**——不需要插 fsync，不需要手写 group commit（xsyncfs 透明提供）。论文还指出 xsyncfs 有应用层方案**做不到的**优化空间：应用的提交策略无法知道自己进程之外的可观察性，只能保守地在每次对外输出前提交。例子：客户端向同机数据库连发两个事务再输出——xsyncfs 可以把两个事务 group commit；而数据库服务器必须逐条提交，因为它不知道客户端在得知第一个事务提交后会不会立即产生外部输出。

基础知识补充

VFS (Virtual File System)：Linux 内核的文件系统抽象层，sync/fsync 等调用先经 VFS 再发到具体文件系统。xsyncfs 在 VFS 之下的 ext3 改造层拦截这些调用。

sync 命令的传统语义：「把所有脏数据刷到磁盘再退出」。在 xsyncfs 下变成「把我对磁盘的依赖登记下来，等提交完再让我的退出可见」——又一次「阻塞转缓冲」的戏法 (Insight#17)。

UNIX socket：同机进程间通信的 socket，走内核不走网卡。6.5 节 MySQL 评测用它让 client/server 依赖可追踪。

核心洞见

Insight#37：对 sync 的处理是全文「点题」的一幕：**同步操作被重新解释为「登记依赖 + 延迟可见」，而不是「立即落盘」**。返回值没变、可见语义没变，物理动作却从「等待磁盘」变成「更新依赖图」。这是「Rethink the Sync」标题的最直接注脚。

Insight#38：「内核能做应用做不到的优化」的论据值得牢记：应用的视野止于自身进程边界，内核的因果图覆盖全系统。**可观察性是一种全局信息，只有全局观察者才能据此优化**——这为「把可靠性下沉到内核」提供了性能（而不仅是便利）层面的理由。MySQL 评测（6.5 节）用真实数据验证了这一点。

5.8 共享内存、mmap 与多线程的依赖传播

为什么这是难点？Speculator 追踪依赖的前提是「进程只能通过系统调用交互」——syscall 是天然的拦截点。但共享内存（shared memory）与内存映射文件（mmap）打破了前提：进程直接读写内存页，**只有共享/解除共享的动作对内核可见**，每次读写内核无从拦截。原版 Speculator 因此干脆不传播共享内存的投机依赖（任意执行状态下做 checkpoint 太复杂）。

方案一（被否决）：页保护陷阱。当对共享页有写权限的进程继承了读方没有的 commit dependency 时，把该页设为不可访问；之后每次读写都触发 page fault，内核在 fault 处理里精确传播依赖；双方依赖一致后恢复页权限。否决理由：反复保护/解除保护的开销与额外的 page fault 太多，性能会很差。

方案二（采纳）：保守假设传播。假设「有写权限的进程在持续写、有读权限的进程在持续读」——当写方继承新的 commit dependency 时，所有读方**原子地**继承同一依赖。代价是可能传播了并不严格必要的依赖（输出多缓冲一会）；收益是零 page fault、零页表操作。同一机制也用于 mmap 文件；对多线程应用同样保守——任一线程继承的依赖全体线程继承（线程天然共享地址空间，等价于共享内存的极端情形）。

📖 基础知识补充

page fault（缺页异常）：CPU 访问无权限/未映射的页时陷入内核的异常。用 mprotect 改页权限制造 fault 是「用硬件当探针」的经典技巧（分布式共享内存 DSM、垃圾回收屏障都用过），但每次 fault 的上下文切换成本在微秒级，高频路径上吃不消。

保守传播为什么安全？与 Insight#24 同理：多传播只损失少量延迟，漏传播破坏正确性。非对称代价下，保守是唯一理性选择。

「原子地继承」：指依赖传播对所有读方同时生效，避免「读到一半依赖才到」的窗口期。

💡 核心洞见

Insight#39：共享内存方案的选择是一个「精确 vs 保守」的范式案例：**当精确追踪的单位成本（页保护+fault）高于保守误判的期望成本（多余缓冲）时，保守方案胜出。**这个不等式在很多场景可复用（缓存失效、GC 写屏障、权限检查），做决策时先把两边成本量化，而不是本能地追求精确。

Insight#40：注意 5.2 节伏笔的回收：多线程对投机执行是「棘手问题」（checkpoint 难），对外部同步却轻而易举（依赖传播不需要 checkpoint）。**换语义不仅省钱，还解锁了新场景——好的简化往往有复利。**

5.9 因果依赖传播全流程演练

把 5.2–5.8 的所有机制串成一次端到端演练。场景：进程 A 写文件，进程 B 通过管道读 A 的输出并打印到屏幕。

时刻	事件	依赖图状态
t1	A: write(file, data) xsyncfs: 校验、修改页缓存、加入T、立即返回 Speculator: 标记 A 为 uncommitted	A —dep→ 事务T (活动)
t2	A: write(pipe, "结果=...")	A 已带 dep(T) → 管道 P 被标记 dep(T) A 的输出入 P 的内核缓冲 (未提交)
t3	B: read(pipe)	B 观察到带 dep 的对象 → B 继承 dep(T) B 被标记 uncommitted
t4	B: printf(screen, ...)	输出继承 B 的 dep(T) → 缓冲在内核, 不显示 → Speculator 回调 xsyncfs!
t5	xsyncfs: 立即提交事务 T	jbd 写日志 + write barrier 冲刷
t6	提交完成	dep(T) 全部解除: A、B 恢复 committed; P 的缓冲数据标记可提交; B 的屏幕输出释放 → 用户看到结果

外部观察者视角: 文件修改持久化 → 然后屏幕出现结果。与同步执行不可区分。

若 B 在 t4 之前先执行了一个 ioctl (不可追踪), 则流程变为: B 被阻塞 → 回调 → T 提交 → B 被唤醒继续。可见所有路径殊途同归: **任何试图让未提交因果链「漏到外部」的动作, 都会触发提交或阻塞。**

📖 基础知识补充

管道 (pipe) 的内核实现: 一个内核环形缓冲区。写端 write 进缓冲, 读端 read 取走。它在内核里, 所以是依赖传播的合法「检查站」——这也是 xsyncfs 能追踪它的原因。

为什么 A 自己向管道写的数据也要等 T 提交才算安全? 严格说, 管道数据在内核缓冲里被 B 读到并不直接违反持久性——违反的是「B 据此产生的**外部输出**」不能先于 T 提交。所以真正被拦在 t4, 而不是 t2/t3。依赖追踪只打标记, 输出缓冲才是闸门。

💡 核心洞见

Insight#41: 全流程演练揭示系统的「防御纵深」结构: **打标记 (追踪) → 缓冲 (隔离) → 回调 (触发) → 提交 (解除) → 释放 (可见)** 五环相扣, 且每一环的兜底都是「阻塞」——宁可慢, 不可错。读者可对照此流程检验论文任何一处机制描述: 它在这五环里的哪一环?

第 6 章 评测 (Evaluation)

Durability、PostMark、Apache Build、MySQL/TPC-C、SPECweb99

6.1 评测问题与方法论

作者用四个问题组织评测：

1. xsyncfs 的持久性与现有文件系统相比如何？
2. xsyncfs 的性能与现有文件系统相比如何？
3. xsyncfs 对「自己做显式同步」的应用（MySQL）有什么影响？
4. output-triggered commits 本身贡献了多少性能？

实验平台：3.02 GHz Pentium 4、1 GB RAM、单块 Western Digital WD-XL40（7200 RPM、120 GB、ATA-100、2 MB 盘载缓存）、Red Hat Enterprise Linux 3（内核 2.4.21）、ext3 与 xsyncfs 均用 400 MB 日志。每个基准都同时测了 ext3 的 journaled 与 ordered 模式——**journaled 在每个基准上都更快**，故只报告 journaled；write barrier 也一律快于禁用 drive cache，故只报告 write barrier 组。多数结果为 5 次试验均值，误差棒为 90% 置信区间。

📖 基础知识补充

90% 置信区间：重复实验均值落在该区间的概率为 90%（在正态假设下）。系统评测的标准做法，用来区分「真差异」与「噪声」。

为什么 journaled 会比 ordered 快？反直觉但真实：journaled 把所有写汇聚成日志的顺序写 + value logging 写合并，机械盘上顺序写远快于 ordered 模式的随机原位写。再次验证「机械盘上，顺序性就是生命」。

对照组的选择逻辑：四种配置——异步 ext3（快但不安全，性能上限参照系）、同步 ext3（安全语义但慢且断电仍丢数据）、同步+write barrier（真正安全，慢 100 倍）、xsyncfs（声称又快又安全）。Figure 3 的持久性矩阵把「安全」维度钉死，后面的性能图把「快」维度钉死。

💡 核心洞见

Insight#42：评测的四个问题分别对应论文的四层主张：可靠性（问题1）、性能（问题2）、对存量应用的价值（问题3）、机制归因（问题4）。**好评测的结构是主张的镜像**——每一层主张都有独立证据，且第 4 问做了消融实验（ablation），把收益归因到具体机制。写论文时先列主张再设计实验，而不是有什么机器跑什么。

6.2 断电持久性实验：ext3 撒谎实锤

这是最硬核的实验——**真的拉电闸**。方法：测试机持续向本地文件系统写数据，每次写完成后向远端机器发一条 UDP 消息（被记录）。实验中途直接切断测试机电源，重启后比对文件系统状态与远端日志。判定标准：

- **不持久 (fails durability)**：远端日志记下了某次写，但重启后测试机缺这次写的数据——即外部观察者（远端）看到了依赖于后来丢失数据的输出；
- **乱序 (fails ordering)**：重启后文件状态违反写的时间序——某块存在但它之前写的某块不存在。

每种配置跑 4 次 (journal_{ed}×2、ordered×2)。结果：

配置	write 后数据持久？	fsync 后数据持久？	备注
异步 ext3	No	断电时不保证	性能上限参照
同步 ext3	断电时不保证	断电时不保证	数据只到 drive cache
同步 + write barrier	Yes	Yes	真正持久，但慢 100×
external synchrony (xsyncfs)	Yes	Yes	且不阻塞应用

(该表对应论文 Figure 3 「When is data safe?」：Yes 表示 OS 崩溃或断电都安全。)

实验还暴露了 ext3 一个**危险行为**：没有 write barrier 时，断电不只丢新数据，还可能**损坏盘上已有数据与元数据**——一次实验中，被修改文件的某个块被静默覆盖为垃圾；另一次，包括**超级块**在内的大量元数据被垃圾覆盖，机器无法重启，只能手工修复文件系统。机理：事务的 commit block 先于其数据块落盘——OS 按正确顺序把块写进了 drive cache，但硬盘重排了写盘片的顺序；恢复时该事务被当作已提交重放，若干数据块实为无效数据，等于用未初始化数据覆盖了磁盘块。此外，journal_{ed} 模式单独不足以保证顺序性：事务落盘片的顺序可能与进 drive cache 的顺序不同。结论：ext3 只有把 write barrier 与某种同步操作（同步挂载或每次修改后 fsync）组合，才同时提供持久性与顺序性；若无 write barrier，等效手段是禁用硬盘缓存。

📖 基础知识补充

为什么用 UDP 而不是 TCP？ UDP 无确认、无重传、开销最小——消息一旦发出就是一次纯粹的「外部输出」，不会被协议栈的可靠性机制缓冲或重试污染实验语义。它在这里扮演「外部观察者」的忠实记录仪。

为什么硬盘会重排写？ 硬盘为最小化磁头移动做电梯调度 (NCQ/TCQ)，缓存中的写请求按物理位置而非到达顺序落盘。这对吞吐是好事，对顺序性承诺是灾难——除非用 write barrier/FLUSH 显式约束。

fsck：文件系统一致性检查与修复工具。元数据严重损坏时需要它手工介入，可能耗时数小时。

💡 核心洞见

Insight#43: 这个实验的方法论价值超过结果本身：**持久性承诺只能用「真实断电 + 外部记录仪」来验证**，代码审查和正常关机测试都不够（缓存里的数据在正常流程下会被慢慢冲刷，伪造出安全的假象）。今天测试持久性的工程实践（如 Jepsen 的断电/崩溃注入、AWS 的故障注入）都是这一思想的延续。

Insight#44: 「commit block 先于数据块落盘 → 垃圾被当有效数据重放」说明：**日志的正确性依赖于一个跨层假设（写序被尊重），而该假设被硬件默认配置悄悄违反**。系统设计文档里的每个「假设」都值得问一句：这个假设在 hostile 的现实硬件上还成立吗？

6.3 PostMark: I/O 密集型基准

负载: PostMark (文献 [8], Network Appliance 设计) 模拟电子邮件、网络新闻、Web 商务的小文件负载：创建 10,000 个文件，执行 10,000 次读/写/创建/删除事务，然后删除所有文件。单线程、以最快速度执行文件系统操作，**不产生输出、不做实质计算**——是纯粹压测文件系统吞吐的负载 (Figure 4, y 轴为对数刻度, 5 次均值)。

配置	相对表现 (定性, 原文 Figure 4)	解读
ext3 异步	最快 (基线 1×)	数据在内存缓冲最多 5 秒才写盘
xsyncfs	与异步差距 < 7%	差距主要来自内核中追踪因果依赖的开销
ext3 同步	比 xsyncfs 慢 10× 以上	阻塞到数据进 drive cache; 吞吐被缓存大小卡死
ext3 同步 + write barrier	比 xsyncfs 慢 100× 以上	每事务两次同步冲刷 drive cache

要点: xsyncfs 提供了**更强**的持久性保证 (write 即持久)，却比只保证「进缓存」的同步 ext3 快一个数量级。作者还讨论了硬件路线：高端存储用 NVRAM 缓存 (文献 [7]) 消除 write barrier 需求，但即使如此，同步挂载的性能也只能达到「第三根柱」（写入易失缓存的速度）——xsyncfs 对这类硬件方案依然能提速。

📖 基础知识补充

对数坐标轴：y 轴按 10 的幂刻度，用来在同一张图上显示相差百倍的数据。看到对数轴要警觉：视觉上的「小差距」可能是 10 倍。

为什么同步模式的吞吐被 drive cache 大小卡死？ 同步写只保证到缓存；缓存一旦写满，后续写必须等缓存里的数据腾挪到盘片——吞吐从「写内存速度」跌回「写盘片速度」。

NVRAM 缓存为什么快？ 电池保护的内存断电不丢，「提交到缓存」=「已持久」，同步写变内存速度。NetApp 的 WAFL（文献 [7]）用此路线。

💡 核心洞见

Insight#45：PostMark 是「纯压力测试」——没有输出意味着 output-triggered commit 几乎不触发，xsyncfs 实质上退化到「5 秒攒批」模式，性能自然贴近异步。这提示我们读评测时要问：**这个基准锻炼的是系统的哪块肌肉？**PostMark 测的是「无观察者的吞吐」，Apache build（下一节）才测「有观察者的延迟」。

Insight#46：7% 的开销归因于依赖追踪——这是整个模型的「门票价格」。值得记住的结论：**因果簿记在内核态的常数成本很低**，换来的是语义层面的巨大简化。与后来 Linux 的 io_uring、eBPF 等机制的「低常数开销换能力」哲学一致。

6.4 Apache build：吞吐与延迟的平衡

负载：把 Apache 2.0.48 源码树解包到文件系统，在其中建 object 目录跑 configure，然后 make，最后删除全部文件。这个负载的特点是**大量屏幕输出与磁盘 I/O、计算交织**——正好同时压吞吐与延迟（Figure 5，柱内分段显示各阶段耗时，5 次均值）。

配置	结果（相对 xsyncfs）	解读
ext3 异步	xsyncfs 差距 ≤ 3%	输出随依赖提交即释放，显示及时
ext3 同步	慢 46%	计算占主导，I/O 差异被摊薄
ext3 同步 + write barrier	慢 11 倍以上	两次冲刷缓存的税照付不误
RAMFS（内存文件系统）	仅比 xsyncfs 快 8%	本地文件系统性能的下界参照

若剥离计算成本（用 RAMFS 结果近似），只看 I/O 部分：同步 ext3 比 xsyncfs 慢 7.5 倍，同步 + barrier 慢两个数量级以上——与 PostMark 的结论一致。这说明 xsyncfs 在混合负载下既没有吞掉吞吐（靠 group commit），也没有牺牲交互延迟（靠 output-triggered commit 让输出随提交即时释放）。

📖 基础知识补充

RAMFS：把文件系统放在内存里，不落盘。用它做「去掉磁盘后的纯计算成本」参照，是分离 I/O 开销与计算开销的标准技巧（下界分析）。

configure/make 负载的 I/O 特征：大量小文件创建（.o、临时文件）、频繁进程 fork/exec、屏幕日志滚动。临时文件在事务窗口内生灭，正好触发 4.2 节「删除抵消写入」的红利。

💡 核心洞见

Insight#47：「xsyncfs 距 RAMFS 只有 8%」是一个被低估的结果——它意味着在该负载下**持久性几乎免费**。当计算占主导时，「可靠性太贵」从来只是错觉；可靠性贵的场景（PostMark 型纯 I/O）里 external synchrony 又把成本压到 7%。两个基准合起来封死了「同步语义必然昂贵」的退路。

6.5 MySQL + 修改版 TPC-C：应用自带同步策略时

动机：前面都是通用负载；如果一个应用（MySQL）自己实现了 **group commit**，xsyncfs 还有意义吗？**负载**：OSDL TPC-C 的修改版（文献 [18]），MySQL 5.0.16 + InnoDB；客户端与服务器同机，改用 UNIX socket（这样依赖可以在 client↔server 间传播）；移除事务间等待时间、数据集完全放进内存，以压满服务器。TPC-C 指标为 NOTPM（New Order Transactions Per Minute，每分钟新订单事务数，总事务数约为其两倍），要求 ACID 语义；MySQL 要满足持久性须禁用 drive cache 或用 write barrier，故对照组为「异步 ext3 + write barrier」（Figure 6，客户端数从 1 增至 20，5 次均值）。

客户端数	xsyncfs vs 异步 ext3 + write barrier	机理
1	快 3 倍 (NOTPM)	依赖传播到 server 与 client，xsyncfs 能对单客户端事务做 group commit；MySQL 只能保守地逐条提交
20	快 22%	客户端多了以后 MySQL 自己也能攒批，差距收窄
32	持平	应用层 group commit 饱和，内核不再占优

另一个有趣数据：对比「异步 ext3 无 write barrier」（不满足 TPC-C 持久性要求，纯好奇）——1-2 客户端时 xsyncfs 仍快 11%，4 客户端以上持平。结论：**即使应用自带定制 group commit，只要并发事务数低到中等，external synchrony 依然带来收益**——因为内核掌握跨进程的可观察性信息，而应用的策略看不到进程边界之外（Insight#38 的实证）。

📖 基础知识补充

TPC-C：数据库联机事务处理（OLTP）标准基准，模拟批发商的订单系统，含五种事务类型，以 New Order 事务占比最大。NOTPM 是它的官方吞吐指标。

InnoDB 的 group commit：把多个事务的 redo log 合并一次 fsync。但 InnoDB 只能在「同时有多个并发事务待提交」时攒批——单客户端串行场景下无批可攒，每条事务一次 fsync，磁盘延迟直接暴露在关键路径上。

为什么同机 UNIX socket 是关键改造？xsyncfs 的因果图只在单机内核里。若客户端走网络在另一台机器，服务器无法知道「客户端收到回复后会不会输出」，只能保守提交。这个改造也揭示了模型的边界（跨机依赖不追踪，见 6.6 与第 8 章）。

💡 核心洞见

Insight#48：MySQL 结果是全论文最反直觉也最有商业价值的数据：**一个打磨了多年的数据库级 group commit，在低并发下被内核的透明机制反超 3 倍**。原因不在算法优劣，而在信息不对称——InnoDB 不知道客户端会不会立刻输出，内核知道。这印证了「把优化放到信息最全的层」。

Insight#49：同时注意差距随并发增加而收敛——这诚实地划出了机制的适用边界：**external synchrony 的红利 = f(并发空隙)**，高并发时应用层攒批已饱和，内核机制退化为持平。好论文不给「万能药」叙事，而是给出有效区间的地图。

6.6 SPECweb99：网络密集型负载

动机：前面的负载都限于单机；网络密集型应用如何？SPECweb99（文献 [29]）：多客户端发 HTTP GET/POST，GET 含最大 1 MB 的静态与动态内容；单客户端模拟 50 个并发连接，经 100 Mb/s 交换机连 Apache 2.0.48 服务器（默认配置下 50 连接足以压满）。

为什么作者预计它「尤其有挑战」？发网络消息就是 externalize 状态；xsyncfs 只追踪单机因果，必须缓冲每条消息直到其依赖的文件数据提交。且 SPECweb99 除 Apache 常规日志外，每个 POST 都会写一条日志——45 分钟运行约 150,000 次文件系统事务，小写频繁。

结果 (Figure 7、8)：

- 吞吐：xsyncfs 与异步 ext3 差距在 **8%** 以内；且 xsyncfs 保证 POST 关联数据在客户端收到响应前已持久（ext3 不保证）。同步 ext3 反而比 xsyncfs 高 6%（该负载写盘少，多数写被 drive cache 吸收）；xsyncfs 比同步+barrier 高 7%。
- 延迟：xsyncfs 给每个 HTTP 请求平均增加的延迟不超过 **33 ms**，低于文献 [5] 引用的 50 ms 人类感知阈值——用户察觉不到差别。

请求大小	ext3-async (秒)	xsyncfs (秒)
0–1 KB	0.064 (±0.025)	0.097 (±0.002)
1–10 KB	0.150 (±0.034)	0.180 (±0.001)
10–100 KB	1.084 (±0.052)	1.094 (±0.003)
100–1000 KB	10.253 (±0.098)	10.072 (±0.066)

📖 基础知识补充

SPECweb99: SPEC 组织的 Web 服务器基准，混合静态/动态 GET 与 POST，考验吞吐与连接管理。

为什么同步 ext3 在这里反而略快？ 网络瓶颈（100 Mb/s）远低于磁盘能力，写量小且被缓存吸收，持久性路径的差异被网络延迟掩盖；xsyncfs 还多付依赖追踪和输出缓冲的税。6% 的反转因此不矛盾——它提醒我们：瓶颈不在磁盘时，文件系统策略几乎无关大局。

人类感知阈值 50 ms: 交互设计中常用的「瞬时感」上界。输出缓冲的延迟只要低于它，用户就无感——整个模型的交互合法性都建立在这个经验数字上。

💡 核心洞见

Insight#50: SPECweb99 的意义在于**边界勘探**：xsyncfs 的因果图止步于单机边界，网络发包 = 不可追踪的外部化，只能缓冲等提交。模型没有破，但「额外延迟 $\leq 33\text{ ms}$ 」给出了跨界成本的具体刻度。这预示了后续工作的方向：把因果追踪扩展到分布式（事实上作者团队的后续工作确实沿此展开）。

6.7 Output-triggered commits 的收益（消融实验）

为量化 output-triggered commits 本身的贡献，作者实现了对照策略 **eager commit**：文件系统一被修改就触发提交（仍可 group commit——前一事务提交期间，多个修改汇入下一事务）。eager 最小化单操作延迟，但牺牲了攒批提吞吐的机会。

基准	Eager Commits	Output-Triggered Commits	提速
PostMark (秒, 越低越好)	9.879 (± 0.056)	8.668 (± 0.478)	14%
Apache (秒, 越低越好)	111.41 (± 0.32)	109.42 (± 0.71)	2%
MySQL 1 客户端 (NOTPM, 越高越好)	3323 (± 60)	4498 (± 73)	35%
MySQL 20 客户端 (NOTPM)	3646 (± 217)	4052 (± 200)	11%
SPECweb99 (Kb/s)	312 (± 1)	311 (± 2)	0%

(数据对应论文 Figure 9, 均值 5 次试验, SPECweb99 为 3 次, 括号内为 90% 置信区间。)

解读：除 SPECweb99（输出太多，两种策略都频繁触发提交，行为趋同）外，output-triggered 全面占优；MySQL 单客户端差距高达 35%——攒批窗口在低并发时价值最大。eager 策略为「单操作延迟」付了全部吞吐代价，却几乎没有换回用户可感知的收益（反正提交本身已经够快）。

📖 基础知识补充

消融实验 (ablation study)：把系统的一个机制摘掉/换掉，看性能掉多少，以此证明该机制的独立贡献。这是系统论文证明「我的每个设计决定都有用」的标准手段。

为什么 eager 也有一点 group commit? 因为事务提交是串行管道：T1 提交期间到达的修改都进 T2，T1 完成后 T2 才开始提交——攒批窗口等于「一次提交的时长」。eager 不是「每修改一次提交」，而是「提交管道永不空闲」。

💡 核心洞见

Insight#51：消融数据回答了 4.5 节留下的定量问题：因果触发的提交比事件触发的提交最多好 35%，且从不更差。「等第一个等待者出现再提交」严格优于「一来活就干」——延迟没有变坏（等待者出现时两者同样立即提交），吞吐却在无人等待时白赚。这是一个 dominated strategy（严格占优策略），应成为同类系统的默认选择。

Insight#52：纵观整个评测章：断电实验钉死正确性下界，PostMark/Apache 钉死通用负载性能，MySQL 钉死存量应用价值，SPECweb99 钉死边界成本，消融钉死机制归因——**五组实验分别对应五层主张，无一冗余**。这是 OSDI 级评测的范式结构，值得作为模板收藏。

第 7 章 参考文献关联分析

References in Context: [1]–[33] 逐条解析

本章先梳理论文 Related Work 一节的论证结构，再对全部 33 篇参考文献逐条说明其在论文中的角色：是背景、是对比基线、是问题来源的证据，还是方法借鉴。理解引用网络，就是理解作者的「学术坐标系」。

7.1 Related Work 的论证结构

作者的 Related Work 不是简单罗列，而是围绕两个「first」主张组织的：

1. **xsyncfs 是第一个不需要专用硬件、不需要修改应用就能提供高性能同步 I/O 的本地文件系统；**
2. **xsyncfs 是第一个利用「文件修改与外部输出之间的因果关系」来决定何时提交数据的文件系统。**

围绕这两个主张，相关工作被分为五路「对手与盟友」：

- **硬件路线：**Rio file cache [2]、Conquest [31] 用电池保护的主存让写持久；Hitz et al. [7] (NetApp WAFL 设备) 把日志放电池 RAM 缓存；eNVy [33] 用闪存 NVRAM + 电池 RAM 写缓存。作者的评价策略是「即使你有硬件，同步操作也最多达到写易失缓存的速度 (PostMark 第三根柱)，xsyncfs 仍能提速」——不承认对方错误，但论证自己**正交且可叠加**。
- **放松持久性的商用主流：**FFS [14] 与原始 UNIX 文件系统 [22] 引入内存缓冲缓存、异步写回；Cedar [6] 与 LFS [23] 加 WAL 换快速恢复；Soft Updates [25] 用精细写序代替日志。作者指出：评测证明「只把数据写进 WAL 而不管 drive cache 重排，照样挡不住文件系统损坏」——即这些方案的持久性故事在真实硬件上有洞。xsyncfs 站在这些工作的肩膀上（它自己也是 WAL），并可以推广到 JFS [1]、ReiserFS [16] 等其他日志文件系统。
- **容错领域的「外部可观察性」传统：**容错研究早就用「外部世界看到的输出」定义一致恢复 [3, 11, 30]——output commit problem 要求消息发往外界前，其发送状态必须已持久。本文相当于把这套原则从分布式容错搬到通用 I/O。作者还饶有兴致地讨论了「为什么这个原则在容错界人尽皆知、在通用系统界却是新的」：两个领域的**恢复范围与恢复标准**不同——容错领域恢复范围是整个进程（不用外部可观察性就得每次状态变化都同步写盘，不可行），通用应用的恢复范围只是应用发出的 I/O（逐条同步虽慢但可行）；容错领域恢复标准是「不丢任何可见输出」，通用系统的标准却松得多（异步 I/O 普遍，连同步 I/O 也常只同步到易失缓存）。
- **容错技术的具体借鉴：**缓冲输出直到提交 $\approx$ 延迟消息发送直到提交 [12]；追踪因果依赖决定提交什么、何时提交 $\approx$ 消息日志协议中的因果追踪 [4]。作者单独用它们改善性能、维持同步 I/O 的外观，又通过 output-triggered commits 把两者组合起来自动平衡吞吐与延迟。
- **事务系统：**QuickSilver [24]、TABS [28]、Locus [32] 等操作系统级事务，以及事务文件系统 [10, 19]，都提供与 xsyncfs 同样强的持久性与顺序保证，外加原子性。但事务系统通常要求应用改造以声明事务边界；xsyncfs 不需要。

📖 基础知识补充

output commit problem (输出提交问题)：容错领域的经典问题——进程崩溃恢复后，它崩溃前发给外界的消息无法「收回」，所以任何消息发出前必须保证「即使立刻崩溃，也能恢复到发出该消息的状态」。本文的文件系统版本：任何输出可见前，其依赖的磁盘状态必须已持久。

message logging (消息日志)：分布式容错技术，记录进程收到的消息，崩溃后重放消息以确定性重建状态。因果追踪用于判断「哪些状态必须先落盘，这条消息才能发」。

WAL 与 soft updates 之争：WAL（日志）用「先写日志再写原位」保证一致性；soft updates 用「精心安排每次写的顺序，使任意时刻磁盘状态都可恢复」达到同样目的。文献 [25] 是两者正面对比的名篇。

💡 核心洞见

Insight#53：作者的 Related Work 是「定位」而非「致敬」：每个相关工作都被放到「硬件/软件 × 改应用/免改造 × 强保证/弱保证」的坐标格里，空格自然显现（Insight#10 的地图在相关工作章完成收口）。**好的 Related Work 让读者自己推出「为什么必须是你」。**

Insight#54：「为什么外部可观察性原则在容错界老掉牙、在文件系统界新鲜」这段元分析极具启发：**同一原则在不同子领域的渗透率，取决于该领域是否「付得起绕开它的代价」。**通用系统界长期用异步 I/O 的松散标准「绕开」了它，直到性能与可靠性的张力大到绕不动——原则的迟到不是疏忽，而是经济平衡的结果。

7.2 逐条关联分析 ([1]–[33])

背景与硬件对比类

编号	文献	在论文中的角色
[1]	Best, «JFS overview», IBM 技术报告, 2000	推广性论据。 JFS 是 IBM 的日志文件系统。作者在 Related Work 末尾指出 external synchrony 同样能提升 JFS 等其他日志文件系统的同步 I/O 性能——[1] 作为「其他日志文件系统」的代表被引用。
[2]	Chen et al., «The Rio file cache: Surviving operating system crashes», ASPLOS 1996	硬件路线对照 （且是本课题组的嫡系前作，Peter Chen 参与）。Rio 用电池保护的主存做文件缓存，让写操作在掉电后存活。作者借它划定「专用硬件方案」一格，论证 xsyncfs 的纯软件路线与之正交且可叠加。
[7]	Hitz, Lau, Malcolm, «File system design for an NFS file server appliance», USENIX Winter 1994	双重引用： 评测章（6.3）用它说明高端存储用 NVRAM 缓存提速同步写；Related Work 用它代表「日志进电池 RAM」的混合方案，并论证即使这种混合方案也不如 xsyncfs 快（其同步操作上限 $\approx$ 写易失缓存）。NetApp WAFL 的经典文献。
[31]	Wang et al., «Conquest: Better performance through a disk/persistent-RAM hybrid file system», USENIX ATC 2002	硬件路线对照。 Conquest 把小文件和元数据放 persistent RAM、大文件留磁盘。与 [2] 同格：电池路线，持久性以「电池有电」为前提。
[33]	Wu, Zwaenepoel, «eNVy: A non-volatile, main memory storage system», ASPLOS 1994	硬件路线对照 + 外推依据。 eNVy 用闪存 NVRAM 存储，发现读快写慢，被迫加电池 RAM 写缓存。作者借此推断：xsyncfs 的技术也可能改善闪存文件系统的写性能——这是论文对 SSD 时代的含蓄预言。

动机与问题证据类

编号	文献	在论文中的角色
[13]	McKusick, «Disks from the perspective of a file system», ;login: 2006	问题来源证据 。引言中「即使 fsync 也防不住断电」这一论断的权威出处。BSD 宗师 McKusick 亲口陈述现代磁盘对文件系统的「谎言」，为全文动机背书。
[27]	Slashdot, «Your Hard Drive Lies to You», 2005	问题来源证据（舆论侧） 。佐证「牺牲持久性换性能是有意识的设计决定」而非 bug。引用 Slashdot 这种社区媒体在顶会论文中很少见，说明该问题当时在业界已是公开痛点。
[15]	MySQL AB, «MySQL Reference Manual»	问题来源证据（应用侧） 。MySQL 官方建议禁用 drive cache——证明真实世界的严肃应用被迫绕过操作系统的默认行为来自保，坐实「抽象失败」。
[20]	Prabhakaran et al., «IRON file systems», SOSP 2005	拆假想敌的关键证据 。Limitations 一节（4.6）中，作者用它回应「同步系统能把介质错误传播给应用」的潜在批评：调查显示 ext3/jbd 与 NTFS 不检测/不上报写错误，ReiserFS 直接 panic——同步系统的「理论优势」在现实中不存在。同期 SOSP 论文，问题意识同源（文件系统对故障的真实行为）。
[5]	Flautner, Mudge, «Vertigo: Automatic performance-setting for Linux», OSDI 2002	经验常量出处 。「人类感知阈值 50 ms」的引用源。这个数字是 xsyncfs「缓冲输出用户无感」论断的支点（SPECweb99 的 33 ms 与之对照）。

方法借鉴类（思想血缘）

编号	文献	在论文中的角色
[9]	Lamport, «Time, clocks, and the ordering of events in a distributed system», CACM 1978	理论基石 。正确性定义中的因果序 (happens-before) 直接来自此文。没有 Lamport 时钟, 「输出按相同因果序发生」这句话无法严格表述。全论文引用链中理论地位最高的一篇。
[17]	Nightingale, Chen, Flinn, «Speculative execution in a distributed file system», SOSP 2005	实现基石 (直接前作) 。依赖追踪、输出缓冲、undo log、checkpoint 机制全部来自 Speculator; 本文是「Speculator 机制的瘦身再利用」。同一作者群, 形成清晰的 research program。
[3]	Elnozahy et al., «A survey of rollback-recovery protocols in message-passing systems», ACM Computing Surveys 2002	思想渊源 。回滚恢复协议综述, 代表容错界「用外部输出定义一致恢复」的传统, 本文的外部同步定义是它的单机文件系统版。
[4]	Elnozahy, Zwaenepoel, «Manetho: Transparent Rollback-Recovery...», IEEE TC 1992	方法借鉴 。Manetho 的因果追踪 (消息日志协议中判断哪些状态须先持久) 对应本文的 commit dependency 追踪。Related Work 明示这一对应。
[12]	Lowell, Chen, «Persistent messages in local transactions», PODC 1998	方法借鉴 (本课题组) 。「缓冲输出直到提交」对应此文「延迟消息发送直到提交」。Related Work 明示对应关系。
[11]	Lowell, Chandra, Chen, «Exploring failure transparency and the limits of generic recovery», OSDI 2000	思想渊源 (本课题组) 。与 [3][30] 并列, 作为「以输出定义恢复一致性」传统的代表。可见 Peter Chen 组长期在「恢复语义」一条线上耕耘。
[30]	Strom, Yemini, «Optimistic Recovery in Distributed Systems», ACM TOCS 1985	思想渊源 。乐观恢复的奠基文献之一, output commit 问题的早期形式化。同上, 为外部可观察性原则背书。
[21]	Qin et al., «Rx: Treating bugs as allergies», SOSP 2005	旁证 。在「多线程进程 checkpoint 是棘手问题」处与 [17] 并列引用, 说明该困难是领域共识, 反衬外部同步「无需 checkpoint」的简化红利 (Insight#40)。

文件系统史与技术脉络类

编号	文献	在论文中的角色
[14]	McKusick et al., «A fast file system for UNIX», ACM TOCS 1984	历史背景。 FFS 引入内存缓冲缓存与异步写回，是「放松持久性换性能」传统的源头之一；也是崩溃后需漫长 fsck 的反面教材。
[22]	Ritchie, Thompson, «The UNIX time-sharing system», CACM 1974	历史背景。 原始 UNIX 文件系统，与 FFS 并列作为缓冲缓存传统的起点。镇纸级引用，标示讨论的历史纵深。
[6]	Hagmann, «Reimplementing the Cedar file system using logging and group commit», SOSP 1987	方法鼻祖。 group commit 与 WAL 进入文件系统的开山之作。本文 4.2 节的 group commit 在思想上直承 Cedar——作者用引用承认了这笔遗产。
[23]	Rosenblum, Ousterhout, «The design and implementation of a log-structured file system», ACM TOCS 1992	历史脉络。 LFS 与 Cedar 并列，作为「加 WAL 换快速一致恢复」路线的代表。作者同时指出该路线在 drive cache 重排面前仍会损坏——为 write barrier 的必要性提供语境。
[25]	Seltzer et al., «Journaling versus soft updates», USENIX ATC 2000	技术对照。 soft updates (Ganger/McKusick 一系) 是与 WAL 平行的元数据一致性路线。引用它完成「异步元数据保护」话题的全覆盖，并把 xsyncfs 定位为 WAL 路线的延续。
[16]	Namesys, «ReiserFS»	推广性论据 + 问题旁证。 推广处：external synchrony 可用于 ReiserFS；旁证处：[20] 的调查中 ReiserFS 遇写错误直接 panic。

事务系统类

编号	文献	在论文中的角色
[24]	Schmuck, Wylie, «Experience with transactions in QuickSilver», SOSP 1991	强保证对手 。OS 级事务提供持久性+顺序+原子性，但要求应用声明事务边界。作者承认其保证更强，以「零改造」胜出。
[28]	Spector et al., «Distributed transactions for reliable systems», SOSP 1985	强保证对手 。TABS，同上。
[32]	Weinstein et al., «Transactions and synchronization in a distributed operating system», SOSP 1985	强保证对手 。Locus，同上。注意标题里就有"synchronization"——本文「Rethink the Sync」与这条1985年的线索形成跨20年的呼应。
[10]	Liskov, Rodrigues, «Transactional file systems can be fast», SIGOPS EW 2004	强保证对手（文件系统侧） 。事务文件系统，同样需应用改造。
[19]	Paxton, «A client-based transaction system to maintain data integrity», SOSP 1979	强保证对手（最老的一篇，1979） 。引用它把「事务保证 vs 改造负担」的张力追溯到 SOSP 第一届年代，显示问题之古老。

评测工具与教科书类

编号	文献	在论文中的角色
[8]	Katcher, «PostMark: A new file system benchmark», NetApp TR3022, 1997	评测工具 。PostMark 负载的出处与设计意图（模拟邮件/新闻/电商小文件）。
[18]	OSDL, «OSDL Database Test 2»	评测工具 。修改版 TPC-C 的来源（OSDL 开源实现）。
[29]	SPEC, «SPECweb99»	评测工具 。网络负载基准出处。
[26]	Silberschatz, Galvin, «Operating System Concepts» (5th ed.), 1998	定义出处 。设计章开头「同步 I/O 按实现定义（阻塞到完成）」的教科书依据——引用它是为树起一个要被「Rethink」的靶子。

💡 核心洞见

Insight#55: 33 篇引用的结构本身就是一张「研究地形图」：理论基石 1 篇 ([9])、实现基石 1 篇 ([17])、问题证据 5 篇 ([13][27][15][20][5])、思想血缘 7 篇 (容错线)、历史脉络 6 篇、对手 5 篇 (事务系统)、工具 4 篇、推广对象 2 篇、硬件对照 4 篇。**没有任何一篇是装饰性引用**——每篇都在论证链上有明确岗位。写作时如果删掉某篇引用不影响论证，那篇引用就不该存在。

Insight#56: 注意作者对「自己课题组前作」的引用策略 ([2][11][12][17])：不掩饰血缘，反而把 lineage 摆明，再用「瘦身」(5.2 节)说清新的增量在哪。**承认继承 + 精确划定增量**，比装作全凭空想更有说服力，也更诚实。

Insight#57: 被引文献的年代分布 (1974→2005) 也值得玩味：作者把 30 年的文件系统史 (UNIX→FFS→Cedar→LFS→soft updates→ext3) 和 20 年的容错史 (乐观恢复→Manetho→综述) 两条线在 2006 年打了一个结。重大创新常常是**两条成熟思想河流的汇流**，而不是凭空涌现。

第 8 章 总结与启示

Conclusion & Takeaways

8.1 论文结论回顾

作者的结论言简意赅：如果系统的地基不可靠，在其上建造简单而可靠的软件就困难重重；异步 I/O 正是这样一个不可靠的地基——OS 崩溃与断电会导致丢数据、文件系统损坏、修改乱序；而文件系统之所以默认提供异步接口，只是因为人们「以为」同步 I/O 的性能代价太大。External synchrony 证明这个「以为」是错的：同步 I/O 接口的简单与可靠可以被完整保留，性能却逼近异步接口。因此像 xsyncfs 这样的外部同步文件系统，可以成为构建可靠软件系统的**更好地基**。

8.2 技术贡献清单

1. **模型**：提出 external synchrony——以「外部不可区分性」定义的同步语义，用 Lamport 因果序严格化，并把磁盘提交纳入外部输出（统一运行时可见性与崩溃恢复）。
2. **机制**：commit dependency + undo log 簿记的因果追踪；输出缓冲替代进程阻塞；output-triggered commits 用因果信息调和吞吐与延迟；共享内存/mmap/多线程的保守传播。
3. **系统**：xsyncfs——在 Linux 2.4 ext3 上以极小侵入（jbd 生命周期钩子 + journaled 模式 + backport write barrier）实现完整模型。
4. **证据**：真实断电实验证明无 write barrier 的 ext3 连 fsync 也不安全且会损坏元数据；四个互补基准 + 消融实验量化性能（异步差距 <7%、同步快 10–100 倍、MySQL 低并发快 3 倍）。

8.3 对今天的启示

启示一：用「可观察行为」重审老抽象。凡是按实现定义的语义（同步=阻塞、事务=两阶段、一致=全副本应答），都值得问一次「它对外承诺的到底是什么？」——承诺往往比实现便宜得多。

启示二：可靠性必须端到端验证。drive cache 事件说明栈中任一层的「善意优化」都会击穿全局承诺。今天 SSD 的掉电保护、NVMe 的 FLUSH、数据库的 fsync 错误处理（2018 年 PostgreSQL fsync 门事件）都在重演同一课。

启示三：把决策放到信息最全的层。应用不知道输出是否被观察，内核知道；内核不知道事务语义，数据库知道。选择机制归属层的标准是「谁拥有决策所需的信息」。

启示四：保守传播 + 兜底阻塞是处理「不可追踪通道」的通用范式。ioctl 阻塞、共享内存保守继承、跨文件系统阻塞——凡图灵完备不可分析的交互，一律 fail-safe。这个范式在后来的信息流控制、机密计算中反复出现。

启示五：模型的边界同样值得铭记。因果追踪止步于单机边界（SPECweb99 的 33 ms）；5 秒兜底是纯行为主义管不到的盲区；灾难性介质故障留给 OS 而非应用。知道一个模型在哪里失效，与知道它在哪里生效同等重要。

💡 核心洞见

Insight#58：如果只能记住这篇论文的一句话，那就是：「同步」是一种对外部世界的承诺，不是一种对内部进程的刑罚。把承诺留给自己（内核缓冲、延迟提交），把自由还给应用（不阻塞），把确定性交给用户（输出永不撒谎）——三者各得其所，就是 external synchrony 的全部智慧。

附录

Appendix

附录 A：术语表

术语 (英文)	中文注释
external synchrony (外部同步)	本文提出的 I/O 模型: 外部观察者无法区分其输出与同步 I/O 系统的输出, 但应用不被阻塞
synchronous / asynchronous I/O	同步/异步 I/O: 前者阻塞到提交, 后者立即返回延迟提交
durability (持久性)	已完成的修改在崩溃/断电后不丢失
causal ordering (因果顺序)	按 Lamport happens-before 定义的事件偏序
commit dependency (提交依赖)	内核对象与未提交事务之间的因果标记
uncommitted (未提交态)	携带一个或多个 commit dependency 的内核对象状态
buffered output (缓冲输出)	依赖未提交事务、暂存内核不送外设的输出
output-triggered commit (输出触发提交)	有输出依赖于活动事务时立即触发该事务提交的策略
eager commit (急切提交)	文件系统一被修改即触发提交的对照策略
group commit (组提交)	多个修改合并为一个原子事务提交
value logging (值日志)	日志中每个块只保留最新值 (ext3 特性), 对立面是 record logging
journal (日志) / jbd	预写式日志 / ext3 的日志块设备层
journaled / ordered / writeback 模式	ext3 三档日志模式: 全日志 / 元数据日志+数据先写 / 仅元数据日志
active transaction (活动事务)	正在接收新修改的那个日志事务, 至多一个
write barrier (写屏障)	强制 drive cache 中此前的写先于此后的写落盘的机制
drive cache (磁盘缓存)	硬盘电路板上的易失 RAM, 默认开启并重排写
fsync / fdatasync / sync	POSIX 显式持久化系统调用家族
Speculator	作者前作 (SOSP'05): 分布式文件系统的投机执行基础设施
speculation object / undo log	Speculator 的投机对象 / 撤销日志 (xsyncfs 中退化为依赖簿记)
checkpoint / rollback	检查点 / 回滚: 保存与恢复进程状态
copy-on-write fork	写时复制的进程分叉, 廉价 checkpoint 手段
page cache (页缓存)	内核中的文件数据内存缓存
IPC	进程间通信: 管道、socket、共享内存等
shared memory / mmap	共享内存 / 内存映射文件: 绕过系统调用的交互通道
ioctl	设备特定系统调用, 因果不可追踪, 触发兜底阻塞

happens-before	Lamport 1978 定义的事件因果偏序关系 [9]
TPC-C / NOTPM	OLTP 标准基准 / 每分钟 New Order 事务数指标
PostMark / SPECweb99	小文件负载基准 / Web 服务器基准 [8][29]
WAL (write-ahead log)	预写日志：先写日志后写数据原位
soft updates	靠精细写序保证元数据一致性的 BSD 方案 [25]
ACID	事务四性：原子、一致、隔离、持久
UDP	无连接网络协议，断电实验中用作外部观察记录
superblock (超级块)	文件系统全局元数据，损坏则全系统不可挂载
kernel panic	内核主动宕机以防止损坏扩大
two-phase commit (2PC)	跨节点原子提交协议
output commit problem	容错领域问题：消息发出前其依赖状态须已持久
NVRAM / 电池缓存	断电不丢的内存，高端存储的硬件持久化方案

附录 B：论文完整参考文献列表

以下列表按原文顺序完整转录 ([1]–[33])，每条括号内为本笔记第 7 章归纳的角色标签。

1. Best, S. JFS overview. Tech. rep., IBM, 2000. (推广对象：JFS)
2. Chen, P. M., Ng, W. T., Chandra, S., Ayccock, C., Rajamani, G., and Lowell, D. The Rio file cache: Surviving operating system crashes. ASPLOS 1996, pp. 74–83. (硬件对照)
3. Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34, 3 (2002), 375–408. (思想渊源)
4. Elnozahy, E. N., and Zwaenepoel, W. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. IEEE TC C-41, 5 (1992), 526–531. (方法借鉴：因果追踪)
5. Flautner, K., and Mudge, T. Vertigo: Automatic performance-setting for Linux. OSDI 2002, pp. 105–116. (50 ms 感知阈值出处)
6. Hagmann, R. Reimplementing the Cedar file system using logging and group commit. SOSP 1987, pp. 155–162. (group commit 鼻祖)
7. Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. Winter 1994 USENIX. (NetApp WAFL, 硬件对照)
8. Katcher, J. PostMark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997. (评测工具)
9. Lamport, L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (1978), 558–565. (理论基石：happens-before)
10. Liskov, B., and Rodrigues, R. Transactional file systems can be fast. SIGOPS European Workshop 2004. (事务文件系统对照)

11. Lowell, D. E., Chandra, S., and Chen, P. M. Exploring failure transparency and the limits of generic recovery. OSDI 2000. (思想渊源)
12. Lowell, D. E., and Chen, P. M. Persistent messages in local transactions. PODC 1998, pp. 219–226. (方法借鉴: 延迟消息发送)
13. McKusick, M. K. Disks from the perspective of a file system. ;login: 31, 3 (2006), 18–19. (fsync 不可靠的证据)
14. McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for UNIX. ACM TOCS 2, 3 (1984), 181–197. (FFS, 历史背景)
15. MySQL AB. MySQL Reference Manual. <https://dev.mysql.com/>. (应用侧证据)
16. Namesys. ReiserFS. <https://www.namesys.com/>. (推广对象)
17. Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. SOSP 2005, pp. 191–205. (实现基石: Speculator)
18. OSDL. OSDL Database Test 2. <https://www.osdl.org/>. (修改版 TPC-C 来源)
19. Paxton, W. H. A client-based transaction system to maintain data integrity. SOSP 1979, pp. 18–23. (事务系统对照)
20. Prabhakaran, V., Bairavasundaram, L. N., Agrawal, N., Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. IRON file systems. SOSP 2005, pp. 206–220. (写错误不上报的证据)
21. Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. SOSP 2005, pp. 235–248. (多线程 checkpoint 困难旁证)
22. Ritchie, D. M., and Thompson, K. The UNIX time-sharing system. CACM 17, 7 (1974), 365–375. (历史背景)
23. Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. ACM TOCS 10, 1 (1992), 26–52. (LFS, 历史脉络)
24. Schmuck, F., and Wylie, J. Experience with transactions in QuickSilver. SOSP 1991, pp. 239–53. (事务系统对照)
25. Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A. Journaling versus soft updates. USENIX ATC 2000, pp. 18–23. (soft updates 对照)
26. Silberschatz, A., and Galvin, P. B. Operating System Concepts (5th Edition). Addison Wesley, 1998, p. 27. (同步 I/O 教科书定义)
27. Slashdot. Your Hard Drive Lies to You. <https://hardware.slashdot.org/article.pl?sid=05/05/13/0529252>. (舆论侧证据)
28. Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L., and Pausch, R. Distributed transactions for reliable systems. SOSP 1985, pp. 127–146. (TABS, 事务系统对照)
29. Standard Performance Evaluation Corporation. SPECweb99. <https://www.spec.org/web99>. (评测工具)
30. Strom, R. E., and Yemini, S. Optimistic Recovery in Distributed Systems. ACM TOCS 3, 3 (1985), 204–226. (思想渊源: 乐观恢复)
31. Wang, A.-I. A., Reiher, P., Popek, G. J., and Kuenning, G. H. Conquest: Better performance through a disk/persistent-RAM hybrid file system. USENIX ATC 2002. (硬件对照)
32. Weinstein, M. J., Thomas W. Page, J., Livezey, B. K., and Popek, G. J. Transactions and synchronization in a distributed operating system. SOSP 1985, pp. 115–126. (Locus, 事务系统对照)
33. Wu, M., and Zwaenepoel, W. eNVy: A non-volatile, main memory storage system. ASPLOS 1994, pp. 86–97. (硬件对照: 闪存 NVRAM)

附录 C：笔记使用说明

本笔记按「背景 → 动机 → 模型 → 设计 → 实现 → 评测 → 参考文献」的顺序组织，每个小节后附「📖 基础知识补充」（灰色块，面向零基础读者）与「💡 核心洞见」（酒红色 Insight#N，全文连续编号，共 58 条）。阅读建议：

- **快速通道**：只读各章正文第一句 + 全部 Insight 块，约 30 分钟掌握论文骨架；
- **精读通道**：顺序阅读，基础知识块遇到不懂就查，约 3 小时；
- **研讨通道**：以第 7 章为纲，把 [9][17][20][25] 四篇关键引用找来对照阅读，理解论文的思想坐标系。

本笔记基于 USENIX 网站公开的论文 HTML 全文整理，所有实验数据均转录自原文（Figure 3-9），未做独立复现。转录数据以原文为准；笔记中的自绘图（图 3-A、3-B、5-A、5-B）为帮助理解的再创作，不对应原论文插图。