arXiv:2603.01384v1 [cs.DC] 2 Mar 2026

Unix 工具与 FITO 范畴错误：崩溃一致性与持久化的协议本质

Paul Borrill
DÆDÆLUS
paul@daedaelus.com

2026年3月

摘要

ls、cp、mv 和 rename 等 Unix 工具提供了一种文件系统抽象，看似呈现出单一且权威的状态，并通过原子转换不断演进。这种抽象并不成立。

我们提出一种系统化的仅向前时间（FITO）分析，证明认为状态转换瞬时且原子的假设，在计算栈的每一层都构成范畴错误——从 ext4 日志记录和延迟分配，到 fsync 失败语义、NVMe Flush/FUA 设备行为，再到 Linux 可重启序列，直至 x86-64 CPU 自身也无法保证在不可屏蔽中断期间原子地进入监管者态。

我们证明了一个形式化的不可能性结果：在发生故障时，不存在任何基于系统调用的持久化原语能够定义提交边界，因为在 Linux 文件系统中，同一个系统调用返回值可能对应多个实质不同的持久化状态。我们指出，跨层时序假设泄漏是范畴错误得以传播的结构性机制，并表明整个存储栈构成了一条由非原子依赖项递归组成的链，其表面上的原子性体现的是数学上的不可能性（Herlihy，1991年），而不只是工程上的不足。

附录记录了这一问题在现实世界中的后果：Google、AWS、Meta 和 Cloudflare 因重试放大而发生连锁云服务中断；PostgreSQL、etcd 和 MySQL 因 fsync 故障而出现数据库损坏；欧洲核子研究中心、NetApp 和 Meta 遭遇静默数据损坏；大规模人工智能训练浪费了 12–43% 的计算预算；金融系统故障每年造成数十亿美元损失。这些后果都源于同一个结构性原因：系统围绕 FITO 假设设计，并在该假设失效时采用重试与恢复协议进行补偿，结果反而放大了它们试图掩盖的故障。

1 引言

Unix 提供了一种看似简单的抽象：由命名对象组成的文件系统，这些对象由看起来能够原子执行的命令进行操作。

这种抽象意味着存在一个唯一的权威状态，该状态通过一系列离散的转换逐步演变。

然而，其底层实现从根本上说是基于协议的。

文件系统操作通过异步回写、日志记录和崩溃恢复机制实现。

应用程序观察到的表面上的瞬时转换，只是协议的可见终点。

抽象与实现之间的这种不匹配构成了一种范畴错误。

2 崩溃一致性与顺序错觉

应用程序开发者通常假设存在顺序崩溃一致性：

如果操作 A、B、C 按顺序执行，那么崩溃后，系统反映的状态应为 A、A+B 或 A+B+C。

这种假设在 ext4 和大多数现代文件系统中并不成立。
ext4 实现的是较弱的崩溃一致性语义，允许重新排列写入操作和元数据操作，除非显式使用 fsync() 等持久化屏障 [1]。
因此：

- 执行顺序并不意味着持久化顺序。
- 命名空间原子性并不意味着数据持久性。
- 系统调用完成并不意味着数据已持久化。

这是 FITO 在文件系统层面的体现。

3 ext4 架构与日志语义

ext4 使用 JBD2 日志层实现日志记录。
存在三种日志模式 [2]:

1. data=日志
2. data=有序
3. data=writeback

在 ordered 和 writeback 模式下，只有元数据保证会被写入日志 .
文件数据可能稍后才会写入。
因此，表示文件存在的元数据可能会先于文件内容本身持久化。

3.1 延迟分配

ext4 通过实现延迟分配来提升性能。
块不会在写入时分配，而是在之后的回写过程中分配。
这会在以下两者之间引入时间分离：

- 逻辑写入完成
- 物理块分配
- 持久化

这种分离否定了写完成定义时间提交点这一假设。

4 重命名操作：原子命名空间、非原子持久性

POSIX 将重命名定义为原子命名空间操作 [3]。然而：

- 重命名不保证持久性
- 重命名不保证与数据写入之间的顺序

正确的持久替换需要：

1. 写入临时文件 2.
fsync(临时文件) 3. 重命
名(临时文件, 目标文件)
4. fsync(父目录)

如果没有目录 fsync，重命名本身可能会丢失。

4.1 写入-同步-重命名模式

现实世界中的系统经常依赖这一四步协议。例如，Git 通过使用 open(O CREAT|O EXCL) 创建锁文件，并通过 write → fsync → rename 序列 [7]提交引用更新。 -

这种模式通常被视为能够提供事务语义：更新要么完整完成，要么代码仓库保持不变。但正如前文分析所示，这一事务保证取决于持久化栈的每一层——从 fsync 到块层，再到 NVMe 刷新——都能正确运行。当 iCloud 等云同步存储系统在这一协议下方插入一个最终一致性复制层时，锁文件本身也会进行异步传播，从而破坏其原本旨在提供的互斥机制。

因此，重命名在命名空间语义上是原子的，但在持久化语义上并非如此。

5 ext4 快速提交

引入 ext4 快速提交是为了提升那些高度依赖 fsync 来保证正确性的工作负载的性能 [4]。

快速提交记录足以重放近期操作的最少元数据。
此功能之所以存在，正是因为应用程序需要明确的协议屏障，才能获得正确的语义。

快速提交的存在证实，正确性取决于协议执行，而非瞬时状态转换。

6 FormalFITO 对 Unix 工具的分析

现在我们来分析经典的 Unix 工具。

6.1 ls

假设存在一致的目录状态。实际情况：它产生的是不断演变的命名空间的观察投影。

6.2 cp

假设采用原子复制。实际情况：这是一个多步协议，会产生部分外部可见状态。

6.3 mv

假设采用原子移动。

实际情况：

- 在文件系统内具有原子性
- 跨文件系统的复制-删除协议

6.4 grep

假设文件内容稳定。实际情况：文件可能在读取过程中发生变化。

6.5 rm

假设会立即删除。实际情况：移除的是命名空间绑定，而非立即移除底层对象。

7 VERITAS 文件系统

The VERITAS 文件系统（VxFS）通过意图日志记录实现崩溃恢复 [5] .

意图日志记录会将元数据操作记录在日志中。

崩溃后，这些操作会被重放。

这种机制维护了文件系统结构一致性，但不会改变文件系统操作的协议本质。

这证实了文件系统演进是基于日志的，而非瞬时完成的。

7.1 实践者经验

在 VERITAS 从事企业级文件系统工程的直接经验揭示了文档化语义与实际部署行为之间的差距。作者参与了与 VERITAS 文件系统工程师的长期讨论，主题是将 fsync 误用作持久性原语，并主张存储硬件中观察到的原子性违规使得常见使用模式不可靠[21]。

这些讨论中逐渐浮现、后来在 Mulligan Stew 讨论组中正式确立的核心见解是：“原子”操作的正确规范并非瞬时状态转换，而是：

完备性要求

要么所有组成操作都按正确顺序发生并完成，要么所有组成操作都按正确的逆序回滚并完成。

这一表述以一种必须在正向和反向上都能收敛的协议，取代了单一提交时刻的 FITO 假设。它直接预示了第 15 节讨论的可逆子事务模型，并表明，本文指出的问题早在得到正式界定之前，就已在工业实践中得到认识。

8 VERITAS 卷管理器

VERITAS 卷管理器通过脏区域日志记录（DRL）跟踪镜像卷 [6] 中已修改的区域。

DRL 通过识别需要重新同步的区域来实现高效恢复。
这明确承认，存储演变会随着时间推移而发生，必须通过协议分析加以重建。

9 根本范畴错误

Unix 将文件系统操作呈现为状态转换。
实现现实：
文件系统操作是随时间展开的协议执行。
因此，文件系统语义从根本上是因果性的、基于协议的，而非瞬时的基于状态的。

这种不匹配构成了一种范畴错误。

10 跨层时序假设泄漏

FITO范畴错误并不局限于单个系统调用。只要某一抽象层假定的时序保证强于其下层实际提供的保证，它就会显现出来。
这种模式——跨层时序假设泄漏——在整个存储栈中反复出现：

1. 应用程序假定 fsync 定义了持久性边界，但文件系统可能在未实现持久化的情况下将页标记为干净（第 11 节）。2. 文件系统假定块层会保持顺序，但 NVMe 刷新可能是空操作。3. 构建系统假定依赖项具有原子可见性，但云同步存储仅提供最终一致性。4. 备份系统（例如时间机器）捕获的本地状态快照可能无法反映云同步全局状态。

关于确定性文档构建的既有研究 [22] 以一种最简且可复现的形式展示了这种泄漏：LaTeX 编译在构建过程中隐含地假定依赖集稳定且能够以原子方式可见——这一假设在本地文件系统上成立，却会被云同步存储违反。由此产生的故障并非编译器缺陷，而是

应用程序所假定的时间模型与存储基底所提供的较弱模型之间的结构性不匹配。

这一概括是：任何运行于最终一致或异步复制存储层之上的系统，都会继承FITO范畴错误，除非它明确考虑到下层的协议性质。

11 fsync故障作为时间性提交边界

fsync系统调用被广泛假定为定义了提交屏障，从而确立文件数据的持久化。

近期研究表明，这一假设并不成立。

11.1 fsync 失败后的未定义状态

文件系统可能在 fsync 失败后将页标记为干净，即使数据并未持久化 [8]。这破坏了该不变量：

$$clean(page) \Rightarrow durable(page)$$

应用程序无法确定真实的持久化状态。

11.2 结构不一致

ext4 在 fsync 失败后可能进入结构不一致的状态，例如索引节点引用了未标记为已分配的块 [8]。

文件系统可能错误地将自身报告为干净状态，从而阻止修复。

11.3 错误报告的时间错位

fsync 失败可能不会在失败的写入发生时报告，而是在后续无关操作时报告 [8]。

这会破坏因果时间推理。

11.4 非收敛持久性

后续的 fsync 调用不会重试失败的写入。因此，fsync 并不保证最终收敛到一致且持久的状态。

11.5 fsync 失败时系统调用可观测的持久性边界不可能存在

现在，我们将 fsync 无法定义持久化边界这一点，形式化为因果协议收敛问题。

定义 1（持久化协议）。设持久化协议是由因果层组成的分布式状态机：

$$N = \{Application,\ PageCache,\ FilesystemJournal,\ BlockLayer,\ ControllerCache,\ PersistentMedia\}$$

每一层 $i \in N$ 都有状态 $S_i(t)$。将持久性定义为收敛：

$$durable \iff \forall i \in N,\ S_i \in Committed$$

本定义　　将持久化刻画为分布式协议的一种收敛性质　　l.

定义 2（提交边界）。系统调用 X 定义提交边界，当且仅当其返回值唯一确定持久性条件是否成立：

$$Return(X) \Rightarrow durable \ \text{ or } \ \neg\, durable$$

并且应用程序可以仅依据这一观察结果安全地继续执行。

引理（故障情况下 fsync 提交边界的不可能性）。在已观察到的 Linux 文件系统语义下，fsync 失败时并不定义提交边界。

证明。Rebello 等人证明，在 fsync 失败之后，多个 Linux 文件系统会将受影响页面标记为干净，即使这些页面尚未写入持久化介质 [8]。具体而言，ext4、XFS 和 Btrfs 在 I/O 提交之前（通过 clear page dirty for io）清除页面脏位，并在发生故障时不将其恢复，从而使页在持久化尚未完成的情况下仍被标记为干净。

因此，系统调用的返回值

$$Return(fsync) = failure$$

与多种实质上不同的协议状态相一致：

$$\exists \Sigma_1, \Sigma_2 :\ Return(fsync) = failure$$
$$\wedge\ durable(\Sigma_1) \neq durable(\Sigma_2)$$

其中 Σ_1 与 Σ_2 在持久化收敛方面有所不同。（例如，ext4 和 XFS 会在内存中保留最新的写入，而 Btrfs 则会将内存中的页面内容恢复为先前的磁盘状态。）

因此，对系统调用的观察并不能唯一确定持久性。故而 fsync 失败并不界定提交边界。

□

推论 1（重试不健全性）。仅基于 fsync 失败的应用程序重试并非正确的恢复策略，因为后续的 fsync 调用并不必然重试失败的写入，且协议可能已经转换到将数据视为干净的状态 [8]。（XFS 默认会重试元数据写入，但这并不延伸到 ext4 或 Btrfs 上的一般数据页面恢复。）

推论 2（认知模糊性）。在 fsync 失败之后，应用可见状态并不能确定协议收敛。持久性在系统调用边界处在认知上变得不可知。

推论 3（FITO范畴错误）。假设 fsync 定义了一个时间上的持久化边界，这就犯了仅向前时间类别错误。fsync 是一种试图推动分布式持久化协议趋于收敛的操作，而不是瞬时状态转换。

定理（无法在系统调用边界观察持久性）。在 Linux 持久化栈中，不存在仅凭返回值就能在故障条件下确定持久性的基于系统调用的持久性原语。

证明。 由引理可直接得出：如果 fsync 失败无法唯一确定收敛状态，那么任何系统调用返回值都不能充当持久化边界。 □

关键洞见

持久性不是某个时间事件的属性，而是分布式协议达到不动点的属性。

12 NVMe 刷新、强制单元访问与掉电保护

在最底层，文件系统耐久性取决于存储设备的持久化语义。

对于 NVMe 设备，规范提供了两种相关机制：(i) 显式的 刷新命令，以及写命令中的强制单元访问（ FUA ）位。

Linux 在块层中明确建模了用于控制回写缓存行为的机制 [10]。因此，通过 fsync() 获得的任何持久性保证，都隐含地涉及存储设备上 Flush 和强制单元访问的行为。

12.1 Flush：一种有条件的持久性机制

NVMe 刷新的定义是将数据从易失性写缓存强制写入非易失性介质。

不过，规范还指出，如果不存在易失性写缓存或该缓存未启用，Flush 命令仍会成功完成，但不会产生任何效果。因此，“Flush 成功”并不能唯一确定持久性边界；根据设备配置和缓存设计，它可能只是一个空操作 [11]。

12.2 强制单元访问与写完成语义

NVMe 还通过定义一个明确的例外，区分写完成与持久化：如果控制器支持易失性写缓存、该缓存已启用、未设置 FUA 位，并且尚未针对该命名空间成功完成任何 Flush，那么后续读取可能返回较旧的数据，而不是最近一次已完成写入的数据 [9]。

这是在规范层面直接否定了“写入完成即意味着持久化”这一天真的 FITO 假设。

12.3 Linux 映射：作为块层原语的刷新与强制单元访问

Linux 块层在设备边界提供了两种控制缓存行为的机制：强制缓存刷新，以及针对请求 [13] 的 FUA 标志。因此，依赖 fsync() 的文件系统和应用程序，实际上也隐式依赖于将文件 API 语义正确转换为 NVMe 刷新和 Write(FUA) 语义。

12.4 断电保护重新分类缓存易失性

当设备包含写缓存，但能够保证在断电时将缓存数据写入非易失性介质时，会出现一种微妙但决定性的情况。在这种情况下，该缓存被视为非易失性 缓存，易失性写缓存功能并不适用；因此，用于配置“易失性写缓存”的控制项可能不会产生任何效果 [12, 14]。

因此，同一命令接口可能对应实质上不同的因果现实：需要 Flush/FUA 才能确立持久性边界的易失性缓存，或者持久化行为根本不同的非易失性（由断电保护支持的）缓存。

12.5 验证：系统能够证明与不能证明的内容

要正确推理持久性，系统必须至少区分四个问题：

（Q1）是否存在易失性写缓存（识别控制器 VWC 字段）？

（Q2）是否已启用易失性写缓存（特性标识符 06h）？

（Q3）是否 操作系统/设备栈支持对写入操作使用强制单元访问（例如 Linux 队列支持强制单元访问）)?

（Q4）发生注入故障时，报告的 fsync 失败是否会使文件系统处于明确定义的认知状态?

最终这个问题并非理论问题：近期研究表明，fsync 失败后，多个 Linux 文件系统会将页标记为干净，使应用级重试失效，并令应用程序开发者无法确定文件系统的状态 [8]。

12.6 FITO 解释

持久性边界并非一个时间瞬间。

这是一个涌现协议状态，横跨应用程序、文件系统、Linux 块层、NVMe 控制器固件、缓存设计和存储介质。

持久化边界取决于：(a) 设备是否有易失性缓存，(b) 该缓存是否已启用，(c) 相关写入操作是否设置了强制单元访问（FUA），以及 (d) 刷新是否已完成。

因此，即使在设备层，持久性也无法建模为单一线性时间线上的瞬时事件；它是横跨操作系统、块层、控制器固件和存储介质的多层协议所涌现的属性。

将持久性视为单一线性“提交时间”，构成了一种仅向前时间类别错误。

13 CPU 级原子性：可重启序列与不可屏蔽中断

前面的章节表明，从应用程序到 NVMe 存储介质，存储栈的每一层都无法保证原子性。现在我们将说明，故障甚至会深入到更底层：CPU 本身也无法保证指令序列在中断发生时的原子性。

13.1 不可屏蔽中断与监管者态入口

在 x86-64 上，通过 SYSCALL 从用户模式切换到管理程序模式时，内核必须执行 SWAPGS（以加载内核的 GS 基址），并加载内核栈指针，然后才能安全地处理任何异常。这些操作相对于不可屏蔽中断（NMI）而言并非原子操作。

Torvalds 指出，这是一个根本性的架构缺陷：在 SWAPGS/栈切换窗口期间，不可屏蔽中断可能触发，导致处理器既不完全处于用户模式，也不完全处于内核模式 [18]。

这不是软件错误，而是硬件设计限制：x86-64 SYSCALL 指令在设计时没有考虑监管者态入口与异步中断之间的交互。英特尔（通过 FRED——灵活返回与事件传递机制）和超威半导体（通过管理程序入口扩展）都提出了硬件修改方案来弥补这一缺陷，这证实了它是一个公认的架构缺陷，而非理论上的担忧 [19]。

13.2 Linux 可重启序列

Linux 内核对廉价用户空间原子性无法实现这一问题的应对方案是可重启序列 （rseq）；该机制经过五年的开发，最终合并到 Linux 4.18 中 [20]。

rseq机制的工作方式如下：

1. 线程通过 rseq(2) 系统调用向内核注册一个线程专属的 rseq 结构体。2. 进入临界区之前，线程设置一个指针，指向描述该区域指令范围和中止处理程序的元数据。3. 线程先执行准备序列，再执行一条原子提交指令。4. 如果内核在准备阶段抢占线程、向其发送信号，或将其迁移到另一 CPU，就会将执行重定向到中止处理程序，该程序通常会触发重试。

关键洞见在于，rseq 并不会阻止中断，而是通过检测中断并从中恢复来重新启动该序列。

CPU 层级洞见

原子性并非执行过程本身的属性，而是重试协议所产生的结果——这正是从数据库事务到 NVMe 固件，再到 CPU 指令序列，在堆栈的每一层都能看到的模式。

13.3 FITO 解释

可重启序列证实，即使在 CPU 指令层级，原子性也不是瞬时属性，而是协议产生的结果。

这种“原子”每 CPU 操作实际上是：

1. 尝试执行临界区。
2. 如果被中断，则中止并重试。
3. 重复执行，直到该部分在不中断的情况下完成。

这是一个随时间进行的协议执行，而非瞬时状态转换。rseq 机制通过与文件系统日志、fsync屏障和 NVMe Flush命令完全相同的模式，实现了原子性的表象： 一个多步协议，只有在没有故障介入时才会收敛到一致状态。

x86-64 NMI问题与 rseq机制共同表明，FITO范畴错误并非仅存在于软件抽象中。它存在于系统的每一层，从应用程序语义一直到 CPU 自身的中断架构。若将任何一层视为能够提供瞬时原子转换，便构成范畴错误。

14 持久化的形式化因果模型

现在，我们将持久化形式化为一种因果收敛性质，而非时间事件。

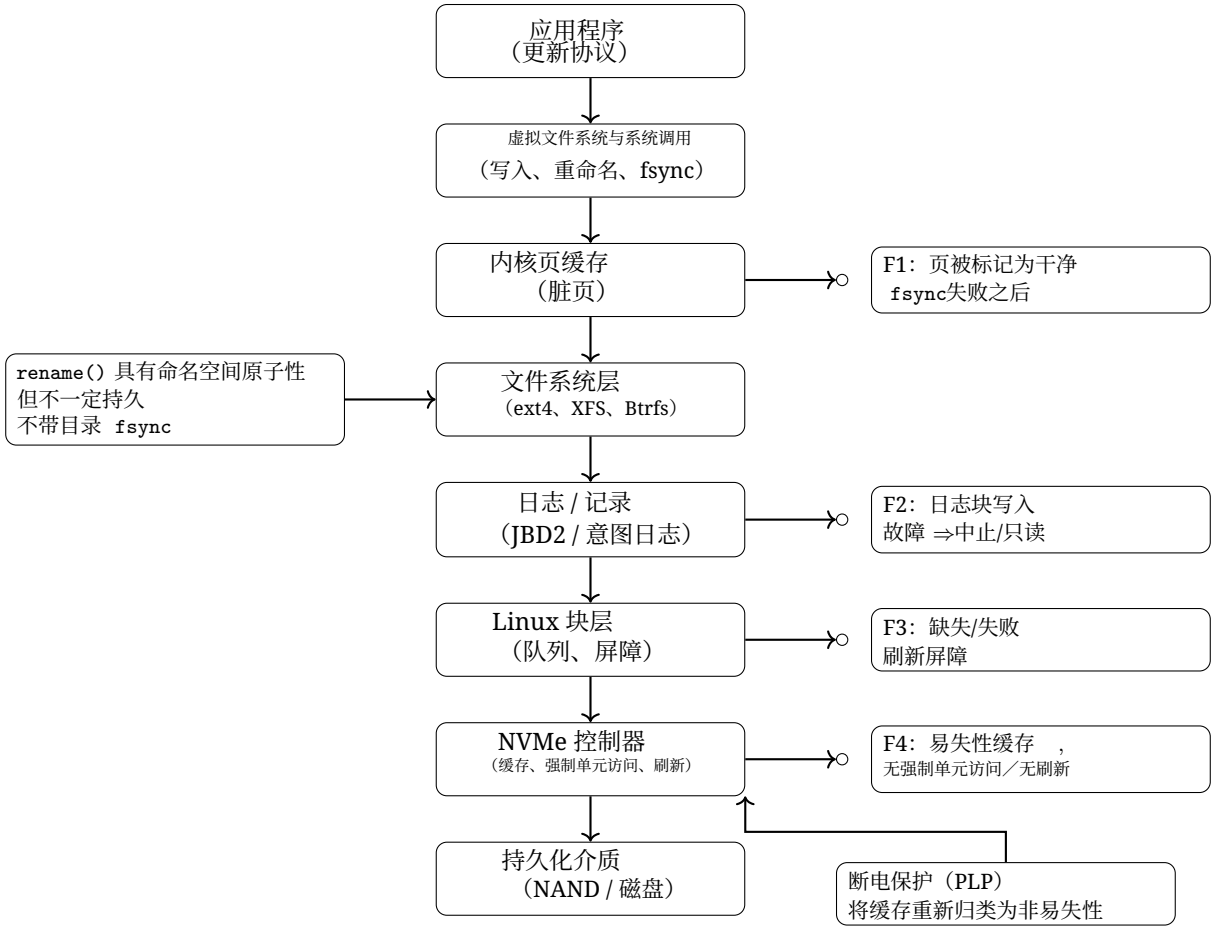

图 1：因果持久性图：持久性不是一个时间瞬间，而是一种涌现协议状态，贯穿应用程序逻辑、内核、文件系统/日志、Linux 块层、NVMe 控制器（Flush/FUA/缓存）和持久化介质。故障点（F1–F4）标示出线性的“提交时间”心智模型失效之处。

设持久化协议由以下节点组成：

$$N = \{Application, KernelPageCache, FilesystemJournal, BlockLayer, ControllerCache, PersistentMedia\}$$

每个节点都有状态：

$$S_i(t)$$

持久化被定义为收敛：

$$durable \iff \forall i \in N,\ S_i = committed$$

没有任何单一的时间瞬间能保证这一性质。
相反，只有当分布式协议收敛时，持久化才会实现。

14.1 时间提交边界的不存在

假设在时间 t_c 存在一个提交边界。

那么：

$$\exists t_c \quad \forall t \geq t_c,\ durable(t)$$

然而，fsync 失败结果表明，fsync 返回后，文件系统层与设备层可能会出现分歧。

因此：

$$\nexists t_c$$

形式化结论

持久化并非一个时间事件，而是一种协议收敛属性。

14.2 验证不可能性

应用程序只能观察到系统调用返回值：

$$observe = syscall_return$$

但持久性取决于隐藏的设备状态：

$$durable = f(hidden_protocol_state)$$

因此：

$$observe \not\Rightarrow durable$$

仅凭系统调用无法验证持久性。

14.3 FITO 解释

Unix 抽象将协议收敛投射到线性时间轴上。
这种投射并不正确。持久性仅作为因果协议收敛的一种属性而存在。

将持久化视为一种时间事件，构成了仅向前时间类别错误。

15 原子性栈：一连串非原子依赖

前面的分析已经追溯了 FITO 范畴错误在各个层中的体现：文件系统语义、日志记录、fsync 和 NVMe 设备行为。现在我们看到，这些层构成了一条依赖链，其中每一层的原子性保证都取决于其下方的层：

1. 数据库依赖事务日志。
2. 事务日志依赖文件系统。
3. 文件系统依赖 fsync。
4. fsync 依赖存储硬件（NVMe Flush/FUA）。
5. 存储硬件依赖 CPU 原子地执行指令。
6. CPU 本身无法在不可屏蔽中断下保证原子性 [18]。

这条链中的任何一层都无法提供无条件的原子性。每一层都假定其下层会正常运行，但正如前文所示，每一层在发生故障时都会违背这一假设。

这种结构构成了一种递归式的范畴错误：每个层级表面上的原子性并非该层级本身的属性，而是对下层的假设。任何一层发生故障时，恢复模式都会退化为实践者所说的“摧毁并重启”——丢弃状态，并从最近一个已知良好的检查点重建。

15.1 与原子性不可能性的关系

这种分层故障模式不仅仅是工程上的缺陷。Herlihy 共识层次表明，读写寄存器的共识数为 1：它们无法在两个进程之间解决共识问题 [15]。

由于文件系统操作最终都归结为针对存储介质的读写操作，因此，无论叠加多少层协议，都无法凭空从数学上不具备提供原子性能力的原语中变出原子性 [16]。

因此，Unix 工具抽象同时犯了两个范畴错误：

1. 将协议执行视为瞬时状态转换（时间性 FITO 错误）。
2. 将非原子原语的组合视为原子操作（代数不可能性错误）。

15.2 对恢复的启示

标准恢复模型——带有重做/撤销的预写日志——本身就是一种 FITO 构造：它假定存在一条可从已知良好状态向前重放的线性日志。但正如 fsync 失败结果所表明的， [8], “已知良好状态”本身可能已经损坏，且日志重放可能无法收敛。

先前关于可逆子事务 [17], 的研究提出了一种替代方法，它完全摒弃了 FITO 恢复模型。可逆协议并不假设存在一条向前重放的线性日志，而是构造逆操作，以便在无需全局检查点的情况下撤销部分状态变更。

这种方法不将原子性视为一种假设，而是将其视为必须通过协议设计加以实现的约束——具体而言，要借助具有无限共识数的原语，例如内存到内存交换 [15, 16]。

16 结论

ext4 以及 VxFS 和 VxVM 等企业级文件系统表明，文件系统操作是涉及日志记录、延迟分配和崩溃恢复的协议执行。

应用程序观察到的表面上瞬时的状态转换，是底层协议执行的投影。

将这 些操作视为原子的时间事件，构成了 FITO范畴错误。

此外，整个存储栈——从数据库事务到文件系统日志、fsync 和 NVMe设备语义——构成了一条非原子依赖链。每一层都假设下一层具备原子性，但没有任何一层能够无条件地提供原子性。这种对非原子原语的递归依赖不仅仅是实现上的缺陷，也反映了仅通过读写操作实现共识在数学上的不可能性 [15]。

正确的推理要求我们认识到，文件系统操作是基于因果结构执行的协议，而非瞬时状态转换。需要原子性时，必须通过在代数上具备提供原子性能力的原语来构造，例如内存到内存交换，而不能想当然地认为无法提供原子性的读写操作组合也能实现原子性。

A ReFITO范畴错误的现实后果 e

前文分析表明，FITO范畴错误贯穿整个存储栈，从应用语义一直延伸到 CPU架构。本附录将记录其现实后果：将协议执行视为瞬时状态转换，直接导致级联故障、数据损坏和经济损失。

A.1 云基础设施：大规模重试放大

FITO 系统遇到不确定性时，会将其解读为故障并发起重试。在负载压力下，重试会增加负载，进而加剧不确定性，又导致更多重试——形成正反馈循环，将局部故障演变为全球范围的服务中断。

Google Cloud，2025年6月12日。Google Cloud 的 Service Control 系统（负责验证所有 GCP API 调用的认证关卡）发生空指针异常，故障经由 Spanner 的全球复制扩散，导致服务中断 7 小时，影响了 70+ 项服务，波及 40+ 个区域，并产生 140 万条用户报告。Cloudflare 、OpenAI、Spotify、Discord 和 Snapchat 均受到波及。根本原因并非空指针，而是“羊群效应”：Service Control 任务集体重启，在没有退避机制的情况下向区域 Spanner 数据库涌入大量请求，引发了次生级联故障，使服务中断持续数小时 [23]。

AWS DynamoDB，2025年10月19日至20日。DynamoDB 的内部 DNS 管理微服务发生竞态条件，导致 60+ 个国家和地区全面中断 15 小时。DNS 恢复、DynamoDB 重新连接后，数百万个 EC2 实例和 Lambda 函数同时重试失败的连接，形成惊群效应，立即再次压垮数据库控制平面，导致 DNS 再次故障。AWS 最终禁用了 DynamoDB DNS 自动化

在全球范围内，并增加了人工审核——这证实了重试放大是架构层面的问题，而非实现层面的问题。Parametrix 估计，仅美国公司的直接财务损失就达 5 亿至 6.5 亿美元 [24]。

AWS Kinesis，2020年11月25日。为 Kinesis 服务器增加容量时，超出了操作系统的最大线程限制，导致后端分片路由映射损坏。由于许多 AWS 服务依赖 Kinesis 进行内部通信，故障蔓延至 CloudWatch、Cognito、Lambda、IoT Core 和 EventBridge。缓存中持续保留着失败的 DNS 查询结果，客户端重试使中断延长至 17 小时 [25]。

Meta，2021年10月4日。一条原本用于评估骨干网容量的常规维护命令，意外地切断了所有 Facebook 数据中心与互联网的连接。DNS 服务器失去连接后，宣告自身运行不健康，并撤回了 BGP 通告。随后，来自 35 亿用户设备的重试风暴导致全球 DNS 流量增加 40%，使全球无关的 Web 应用程序延迟上升。由于内部运维工具依赖已瘫痪的基础设施，恢复工作必须通过物理方式进入数据中心才能进行。持续时间：6–7 小时；预估广告收入损失：$60+百万 [26]。

Cloudflare，2025年9月12日。Cloudflare 控制面板中的一个 React useEffect 缺陷在依赖数组中创建了一个对象，而该对象在每次状态变化时都会重新创建，从而产生过多的 API 调用。这些调用在一次并发更新期间压垮了 Tenant Service，进而波及所有控制面板 API。Cloudflare 的事后分析报告明确指出，需要采用随机重试延迟来防止惊群效应 [27]。

A.2 数据库损坏：实践中的 fsync 失败

第 11 节分析的 fsync 故障模式，在生产系统中表现为实际的数据丢失。

PostgreSQL“fsyncgate”（2018–2019）。Craig Ringer 向 PostgreSQL 邮件列表报告称，PostgreSQL 在 Linux 上的 fsync 错误处理存在根本性缺陷。缓冲写入失败时，内核会将页标记为带有错误标志（AS EIO），但在第一次 fsync 失败后就清除了该标志。重试时，由于错误标志已被清除，fsync 会返回成功——尽管数据从未写入磁盘。PostgreSQL 12 中合并的修复方案是在 fsync 失败时触发 *panic* ，而不是重试，因为在观察到的内核语义下，不存在正确的重试策略 [28]。

这直接从经验上证实了我们的引理：应用可见的观察无法决定持久化状态。

etcd Raft 日志损坏（2021年）。etcd 3.5.0–3.5.2 版本中，在高负载下杀死集群领导者后，Raft 日志会遭到损坏；重启后，已提交事务未能反映在所有成员上。根本原因是 fsync 性能瓶颈：Raft 日志上的 fsync 速度缓慢，导致领导者不可用，因为领导者在等待磁盘时无法提交新的提案 [29]。

MySQL InnoDB 双写缓冲区。InnoDB 双写缓冲区之所以存在，完全是因为 16 KB 的 InnoDB 页面在写入过程中发生崩溃时可能会出现撕裂，导致页面中混杂着新旧数据。自 MySQL 5.1 起，这一机制便一直存在，并给全球每个 InnoDB 部署带来了可衡量的

写入放大——这是为弥补页写入不具备原子性而付出的永久性能代价 [30]。

NVMe 断电导致数据丢失。独立测试（2022年）表明，SK海力士和 Sabrent 的消费级 NVMe 固态硬盘在 fsync 调用完成后，仍会在断电后丢失数据；而三星和西部数据的硬盘则能保留数据。差异在于硬件断电保护（PLP）电容器。配备断电保护的企业级固态硬盘价格高出许多，这正是非原子持久化栈 [31]带来的直接经济后果。

A.3 静默数据损坏

欧洲核子研究中心（2007年）。一项为期六个月的研究在 3,000 个节点上写入了约 97 PB 数据，发现有 128 MB 数据永久损坏，发生率为 1.2×10^{-9}；研究人员在 100 个节点上检测到 500 次错误 [32]。

NetApp（2008年）。一项为期 41 个月、涉及 153 万块磁盘的研究记录了超过 40 万次校验和不匹配，其中 3 万次未被 RAID 控制器检测到，仅在数据清洗期间才被发现。近线磁盘发生数据损坏的速率 $10\times$ 与企业级硬盘相同 [33]。

Meta静默数据损坏（2021年–2026年）。Meta 的集群分析发现，约每 1,000 台机器中就有 1 台受到静默数据损坏（SDC）的影响。对于大规模人工智能训练任务，预计每隔一到两周就会发生 SDC 事件。随着制程缩小，这一问题愈发严重：软错误率从 65 nm 时每年发生一次故障，升至 16 nm 时每 1.5 小时发生一次故障 [34,35]。

A.4 人工智能训练基础设施

FITO范畴错误会给人工智能训练带来巨大的成本，因为检查点与重试是其主流容错策略。

大规模场景下的故障率。Meta 使用 16,384 个 NVIDIA H100 GPU 训练 Llama 3，在 54 天内遭遇了 419 次意外中断，平均每 3 小时发生一次故障。GPU 和 NVLink 故障占 30%，HBM3内存故障占 17%，软件／网络问题占 41% [36]。对 Meta 生产集群的学术分析预测，一项使用 131,072 个 GPU 的任务平均故障间隔约为 14 分钟 [37]。

检查点开销。 在大规模训练中，检查点开销占总训练时间的 12–43%。一个包含 16,000 个加速器的集群每天需要大约 155 个检查点（每 9.3 分钟一次）；加速器数量达到 100,000 个时，每天则需要 967 个检查点（每 1.5 分钟一次）。对于计算成本接近 10 亿美元的前沿模型训练任务，这些开销意味着 1.2 亿至 4.3 亿美元的纯粹浪费 [38,39]。

恢复时间。 传统的基于检查点的恢复在每次故障发生后需要 15–30+ 分钟。若每 3 小时发生一次故障，一个包含 16,384 个 GPU 的集群仅恢复一项就会损失 8–17% 的训练时间，此外还要承担检查点开销。

A.5 金融系统与关键基础设施

Knight Capital，2012年8月1日。 SSH 连接中断时，一个部署脚本未能发出任何提示，导致八台交易服务器中的一台仍在运行已弃用代码。订单发起与完成记录之间的失步，导致该服务器在 45 分钟内发送了 4+ 百万笔非预期订单，造成 $460+ 百万美元损失。随后，美国证券交易委员会指控 Knight Capital 违反市场准入规则 [40]。

CME Globex，2025年11月27日至28日。 数据中心冷却系统故障导致价值1万亿美元的衍生品交易停摆 11+ 小时，波及E-mini标准普尔500指数、美国国债期货、原油、黄金和外汇市场。物理基础设施中的一个单点故障，最终波及全球金融市场 [41]。

美国联邦航空局航行通告系统，2023年1月11日。 合同人员在同步主航行通告数据库与备份航行通告数据库时，意外删除了文件。由于两个系统共用同一网络基础设施，数据损坏蔓延至备份系统——这是典型的原子性故障，备份机制未能提供任何故障隔离。结果：1,300+ 架次航班取消，近10,000架次航班延误，全国停飞令持续2小时 [42]。

A.6 FITO的经济代价

累积证据量化了FITO范畴错误造成的经济负担：

- Gartner估计，截至2024年，IT停机的平均成本为每分钟9,000美元（每小时540,000美元），较此前十年增长61%。
- 44% 的组织表示，每小时停机造成的损失超过 100 万美元（Uptime Institute，2025年）。
- 单次云服务中断造成的损失高达数亿美元：5亿至6.5亿美元（AWS，2025年10月）、1.5亿美元（AWS S3，2017年）、6000万美元+ （Meta，2021年）。
- 仅人工智能训练检查点开销一项，就会使运行成本接近 10 亿美元的前沿模型训练浪费 1.2亿至 4.3亿美元的计算资源。
- 金融市场每年因延迟套利损失约 50 亿美元；这种套利利用了全国最佳买卖报价同步性惯例与物理因果性之间的差距 [43]。

这份清单中的每一项都可追溯到同一个结构性原因：系统设计时假定操作都是瞬时状态转换，而当这一假设失效时，又以重试与恢复协议作为补偿，结果反而加剧了它们试图掩盖的故障。

网络层事务完整性的必要性

FITO 范畴错误并非理论上的奇谈。它是云服务接连中断、数据库损坏、人工智能训练浪费以及每年造成数十亿美元损失的金融系统故障的结构性根本原因。要消除重试放大，就必须以双边事务取代单向消息——即 FITO 原语：一种双方端点要么共同完成、要么共同失败的原子交互。没有“传输中的”消息，也不会产生重试放大。

参考文献

[1] Bornholt 等人，《文件系统崩溃一致性模型的规范与检查》，ASPLOS 2016。[2]Linux 内核文档，《ext4 文件系统文档》，https://docs.kernel. org/filesystems/ext4.html[3] POSIX，《rename() 规范》，https://pubs.opengroup.org[4] Linux 内核文档，《ext4 快速提交》，https://lwn.net/Articles/842385[5] VERITAS，《VERITAS 文件系统管理员指南》。[6] VERITAS，《VERITAS 卷管理器管理员指南》。[7] Borrill，《iCloud 为何失效》，arXiv:2602.19433[8] Rebello 等人，《应用程序能否从 fsync 失败中恢复？》，ACM 存储汇刊，2021年。[9]NVM Express 公司，《NVM Express NVM 命令集规范，修订版 1.1》（修订版于 2024年8月 获批准），https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-Revision-1.1-2024.08.05-Ratified. pdf。[10] Linux 内核文档，《显式易失性回写缓存控制》，https://docs. kernel.org/block/writeback_cache_control.html。[11] 引用 NVMe 规范中关于易失性写缓存不存在或已禁用时 Flush 为空操作的行为的讨论，https://news.ycombinator.com/item?id=46535896.[12] Microsoft 文档（NVMe VWC 功能说明），《NVME CDW11 FEATURE VOLATIL

_ _ _ E_写入 CA_

https://learn.microsoft.com/en-us/windows/win32/api/nvme/ns-nvme-nvme_cdw11_feature_volatile_write_cache.[13] Linux 内核文档（队列 sysfs），“队列 sysfs：强制单元访问”，https://www.infradead. org/~mchehab/kernel_docs/block/queue-sysfs.html.[14]金士顿科技，“深入了解 SSD 断电保护”，https://www.kingston.com/en/blog/servers-and-data-centers/ssd-power-loss-protection.[15] M. Herlihy，“无等待同步”，ACM 编程语言与系统汇刊，1991年。[16] P. Borrill，“通过读写操作实现原子性的不可能性：RDMA 语义、范畴错误与基于交换架构的理由”，DÆDÆLUS，2026年。[17] P. Borrill，“重新思考原子性：反事实事务”，DÆDÆLUS，2026年。[18] L. Torvalds，“回复：x86/entry：FRED CPU 异常/NMI 入口注意事项”，Linux 内核邮件列表，2023年。https://lore.kernel.org/lkml/[19] 英特尔公司，“灵活返回与事件传递（FRED）规范”，2023年。

[20]M. Desnoyers，“将可重启序列引入 Linux 的五年历程”，EfficiOS 博客，2019年2月。https://www.efficios.com/blog/2019/02/08/ linux-restartable-sequences/[21] P. Borrill 等人，“Mulligan Stew：原子性、完备性与可逆协议”，关于因果一致性和基于协议的原子性的私人讨论组，2024年。https: //mulligan-stew-portal.lovable.app[22]P. Borrill，“最终一致性存储下的确定性文档构建”，DÆDÆLUS，2026年。[23] Google Cloud，“事件报告：2025年6月12日服务控制中断”，https://status. cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW.[24] ThousandEyes，“AWS 故障分析：2025年10月20日”，https://www.thousandeyes.com/blog/aws-outage-analysis-october-20-2025.[25] AWS，“北弗吉尼亚（US-EAST-1）区域 Amazon Kinesis 事件摘要”，2020年11月25日。[26] Meta Engineering，“有关10月4日故障的更多详情”，2021年10月5日。https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/.[27]Cloudflare，“深入剖析 Cloudflare 于2025年9月12日发生的仪表板和 API 故障”，https://blog.cloudflare.com/deep-dive-into-cloudflares-sept-12-dashboard-and-api-outage/.[28] D. Luu，“Fsyncgate”，https://danluu.com/fsyncgate/.[29] etcd-io，“etcd 3.5.0 panic：损坏的 Raft 日志”，GitHub Issue #13509，2021年。[30] Percona，“什么是 InnoDB 双写？它如何工作？”，https://www.percona. com/blog/innodb-double-write/.[31] R. Bishop，“我测试了四家厂商的四款 NVMe 固态硬盘——断电时有一半会丢失已执行 FLUSH 的数据”，2022年2月。[32] 欧洲核子研究中心，“数据损坏研究”，2007年。StorageMojo 于2007年9月报道。[33] L. Bairavasundaram 等人，“存储栈中的数据损坏分析”，FAST 2008。[34] Meta Engineering，“大规模静默数据损坏”，2021年2月。https://engineering.fb.com/2021/02/23/data-infrastructure/silent-data-corruption/.[35] 开放计算项目，“人工智能中的静默数据损坏”，OCP 白皮书，2024年。[36] Meta，“Llama 3 模型群”，2024年。训练基础设施数据：在 16,384 个 H100 GPU 上，54 天内发生了 419 次中断。[37] S. Patel 等人，“重新审视大规模机器学习研究集群的可靠性”，arXiv:2410.21680，2024年。[38] Google Cloud 博客，“为大型人工智能训练任务使用多层检查点”，2024年。

[39] AWS 博客，《为 AWS 上的大规模机器学习训练设计可扩展的检查点存储》，2024年。[40] 美国证券交易委员会，《美国证券交易委员会指控 Knight Capital 违反市场准入规则》，新闻稿 2013-222，2013 年 10 月 16 日。[41] CNBC，《数据中心发生问题后，芝加哥商品交易所暂停外汇、大宗商品和期货交易》，2025 年 11 月 28 日。[42] 美国交通部，《美国联邦航空管理局航行通告系统故障及其影响》，2023年 1 月。[43] P. Borrill，《人为设计的同时性》，DÆDÆLUS，2026年。