持久化数据结构

黄梓淳

Jun 25, 2026

在多线程与函数式编程日益普及的今天，数据结构需要同时满足「不可变」与「可回溯」两个看似矛盾的需求。不可变数据让并发读写天然安全，而可回溯则支持撤销重做、版本控制等真实场景。持久化数据结构正是为这一目标而生，它通过结构共享实现多版本共存，历史版本保持只读，当前版本仍可高效写入。接下来我们将沿着定义、理论、实现、实践与前沿的方向，系统梳理这一领域。

1　理论基石

持久化数据结构可分为完全持久化与部分持久化。完全持久化允许在任意历史版本上创建新版本，所有版本均可继续演化；部分持久化则只允许在最新版本上写操作，旧版本仅供只读访问。二者的复杂度差异主要体现在指针维护与版本计数上。持久化本质上是不可变的一种特例：不可变只要求对象一旦创建就不再修改，而持久化还要求在修改时保留全部历史版本，从而形成显式的版本演化链条。

复杂度度量需同时考虑时间、空间与版本跨度三维。最坏情况摊还复杂度允许单次操作代价较高，但平均到一系列操作后仍保持期望界；严格最坏情况复杂度则要求每一次操作都必须在该界内完成。持久化结构常在这两种度量间权衡，以获得工程上可接受的常数因子。

2　核心实现技术

路径复制是最直观的持久化手段。以一棵二叉搜索树为例，当我们更新某个叶子节点时，仅需复制从根到该叶子的路径上所有节点。新版本的根指向新路径，旧版本的根仍指向旧路径，从而实现结构共享。路径长度为树高，因此单次修改的时间与空间开销均为对数级别。

胖节点法则在节点内部维护多版本字段，避免整条路径复制。每个字段带时间戳或版本号，读取时根据当前版本选择对应值。该方法用空间换时间，适合更新频繁但版本跨度不大的场景。

节点分裂与权重平衡树则利用旋转操作，将持久化代价分摊到平衡维护过程中。以红黑树为例，插入或删除后的平衡旋转同样产生新节点，旧版本指针保持不变，从而在严格最坏情况下仍能保证对数复杂度。

哈希数组映射树（HAMT）将哈希值按位分区，每层节点用位图记录有效子节点位置。写操作仅复制受影响的路径节点，并通过位运算快速定位子节点，实现工业级不可变哈希表。位分区策略让随机访问与更新复杂度稳定在常数级别。

惰性求值与分块共享则针对线性结构。以持久化向量为例，底层用树状索引组织数据块，尾部共享策略让追加操作仅需复制少量中继节点，而不触碰已有数据块。惰性求值进一步延迟实际复制，直到真正需要旧版本数据时才执行，从而降低平均开销。

3 典型数据结构实例

持久化栈可通过「反向链表加惰性求值」实现。每次压栈创建新节点，节点内部保存值与指向前驱的不可变指针；弹栈则返回前驱节点，旧栈仍可通过原根访问。双端队列在此基础上增加「惰性旋转」技巧，保证两端操作均为摊还常数时间。

持久化红黑树在标准实现中加入路径复制逻辑。插入时沿搜索路径复制节点，颜色调整与旋转同样作用于新节点，旧树根保持不变。Java 与 Scala 的标准库均以此为基础，提供线程安全的不可变集合。

Clojure 的 PersistentHashMap 采用 HAMT 实现。哈希值被切分为多段，每段对应树的一层；更新时仅新建受影响的路径节点，位图字段用「与运算」快速判断子节点存在性。Scala 的 HashMap 也复用类似结构，读写均摊还常数时间。

持久化向量在 Clojure 中用 RRB-Tree（Relaxed Radix Balanced Tree）实现。树高固定为五层，每层分支因子为三十二，随机访问复杂度为对数三十二。追加元素时仅复制末尾路径，尾部共享让内存占用接近线性。

持久化并查集通过扩展域记录版本信息。路径压缩与按秩合并操作均产生新节点，旧版本的父指针保持不变，从而支持历史版本的连通性查询。Gabow 等人的算法将复杂度维持在反阿克曼函数级别。

持久化线段树常用于算法竞赛。动态开点策略让未访问区间不分配节点，更新时沿路径新建节点，旧版本根仍指向旧结构。区间求和与前缀和操作均可在对数时间内完成，且支持多版本共存。

4 工程实践与性能权衡

持久化对象因额外指针与对象头，内存占用显著高于可变版本。现代垃圾收集器如 ZGC 与 Shenandoah 通过并发标记与整理，降低持久化对象带来的停顿压力，但仍需关注「被遗忘旧版本」的内存泄漏风险。

路径复制导致节点在内存中不连续，缓存局部性变差。NUMA 架构下跨版本遍历可能触发远端内存访问，进一步放大延迟。工程师常通过「预取」与「节点池」缓解这一问题。

并发场景下，读写分离与 Copy-On-Write 结合可实现无锁读取。RCU 与 Hazard Pointer 则用于保护正在遍历的旧版本指针，避免悬垂引用。实际系统中常将这些机制组合使用，以平衡吞吐与延迟。

持久化数据结构与持久化存储（Persistence on Disk）概念不同。前者指内存中的多版本共存，后者指数据落盘后仍能恢复。写时复制文件系统如 Btrfs、ZFS、APFS 借鉴了类似思路，在块级别实现快照与回滚。

性能剖析工具如 JMH、Criterion 与 perf 可量化持久化开销。火焰图能直观显示「被遗忘旧版本」占用的 CPU 与内存，帮助定位泄漏点。

5 实际应用场景

函数式编程语言运行时天然依赖持久化集合。Clojure、Scala、Haskell 的默认集合均为持久化实现，开发者无需额外处理并发问题即可安全共享状态。

数据库与分布式系统广泛采用多版本并发控制（MVCC）。LSM-Tree 的 MemTable 形成版本链，RocksDB 通过版本链实现快照隔离与回滚。持久化结构在此提供底层数据支撑。

撤销重做与时间旅行调试需要保留操作历史。Git 对象模型本质上是 Merkle DAG，Emacs Undo-Tree 与游戏存档回溯也依赖持久化结构实现高效版本切换。

响应式 UI 框架如 Redux 与 Immutable.js 使用持久化状态树。React 的 Fiber 架构同样借鉴了结构共享，让

状态更新仅影响受影响子树，降低重渲染代价。
区块链依赖 Merkle Patricia Trie 与 Verkle Tree 实现状态证明。持久化特性让轻节点仅存储路径哈希即可验证交易，兼顾安全与存储效率。

6 前沿与研究课题

最坏情况常数时间持久化栈与队列仍是开放问题。Brodal、Kaplan、Tarjan 等人提出多种候选方案，但常数因子与实现复杂度仍需进一步优化。
外存持久化结构如 FD-Tree 与 B ε -Tree 将持久化思想扩展到磁盘。它们在缓冲区与刷盘策略上做特殊设计，以降低随机 I/O 对持久化开销的影响。
持久内存（PMem/NVDIMM）带来新的语义挑战。字节寻址持久内存要求显式 flush 与 fence 保证持久性，NOVA、SoupFS 等文件系统探索了在持久内存上构建持久化数据结构的方法。
代数效应与代数数据类型为编译器辅助的结构共享提供了可能。通过静态分析，编译器可自动推导哪些字段需要复制，从而降低手动优化负担。
硬件辅助方面，Intel TSX 与 ARMv8.5 MemTag 可加速写时复制的冲突检测与内存标记检查，为持久化结构提供新的性能拐点。
持久化数据结构以结构共享为核心，在提供线程安全与版本回溯的同时，保持了对数或常数级别的读写性能。其代价是额外内存占用与缓存不友好，但现代垃圾收集器与硬件特性正在逐步缓解这些问题。展望未来，持久内存、异构计算与形式化验证将与持久化结构深度融合，推动其在数据库、区块链与函数式运行时中的进一步演进。

7 参考文献与延伸阅读

Okasaki 的《Purely Functional Data Structures》系统阐述了函数式数据结构的持久化实现。Driscoll 等人的论文《Making Data Structures Persistent》奠定了理论基础。Clojure、Scala 与 Haskell 的官方源码提供了大量工程实践案例。近期关于 RRB-Tree、Verkle Tree 与 NOVA 文件系统的论文则代表了该领域的前沿进展。