Linux 内存压缩技术（ZRAM/Zswap）

杨岢瑞

Sep 23, 2026

物理内存始终是操作系统最宝贵的资源之一。随着容器化、微服务以及大数据分析的普及，应用程序对内存的需求往往呈现爆发式增长，而物理内存的增长速度却无法同步。这就导致了在生产环境中频繁出现 OOM 杀手介入、Swap 风暴以及系统卡顿等现象。内存压缩技术正是为了缓解这一矛盾而诞生的核心手段，它通过牺牲少量 CPU 周期来换取更大的有效内存容量，从而在有限的物理资源上支撑更高的负载。
目前 Linux 内核提供了两种主流的内存压缩实现：ZRAM 与 Zswap。二者虽然都利用压缩算法来减少内存占用，但所处的层次、生命周期以及适用场景却存在显著差异。ZRAM 直接在块设备层提供一个纯内存的压缩块设备，而 Zswap 则作为 frontswap 的后端，在匿名页即将换出时先进行压缩缓存，必要时再回写到真实 Swap 设备。理解这两种技术的内部机制，有助于运维与内核开发者根据实际负载做出最优选择。

1　背景知识：Linux 内存管理简述

Linux 内存管理以页帧为最小管理单元。内核通过 zone 结构将物理内存划分为不同的区域，例如 DMA、Normal 与 HighMem，以便针对不同硬件约束进行分配。当系统面临内存压力时，页帧回收算法 PFRA 会按照 LRU 策略扫描并释放不再活跃的页。匿名页与文件页在回收策略上有着本质区别：文件页可以直接丢弃或写回文件系统，而匿名页则必须通过 Swap 子系统写入磁盘。
然而，磁盘 I/O 的高延迟与低带宽往往成为瓶颈。内存压缩正是利用 CPU 强大的计算能力，在内存内部完成数据紧缩，从而避免昂贵的磁盘访问。代价是压缩与解压带来的 CPU 开销，以及元数据与碎片引入的额外内存占用。理解这一「CPU 换内存带宽」的权衡，是后续分析 ZRAM 与 Zswap 的前提。

2　ZRAM 原理与实现

ZRAM 本质上是一个块设备驱动，它把物理内存作为后端存储，并对写入的数据进行实时压缩。核心数据结构包括 `struct zram`，它描述一个 ZRAM 设备实例；`struct zcomp`，用于抽象压缩算法；以及 `zram_table_entry`，记录每个页的压缩后物理地址与长度。
当上层文件系统或 Swap 子系统发起写请求时，ZRAM 首先调用 `zram_write_page`。该函数从请求的 bio 中获取原始页数据，选择当前配置的压缩算法（例如 LZ4 或 ZSTD）进行压缩。若压缩成功，则分配物理内存存放压缩后的数据，并更新 `zram_table_entry` 中的元数据；若压缩率不佳，则可能直接存储未压缩数据以避免额外开销。读取路径则相反，`zram_read_page` 根据元数据定位压缩数据，解压后通过零拷贝方式返回给上层。
压缩算法的演进体现了延迟与压缩率的权衡。早期版本使用 LZO，具备极低的延迟；后续引入 LZ4 进一步降低 CPU 周期；ZSTD 则以更高的压缩比为代价换取更好的内存节省。在多 CPU 环境下，ZRAM 采用 percpu 互斥锁来保护关键结构，同时支持批处理以降低锁竞争。

3 Zswap 原理与实现

Zswap 的设计目标是在匿名页即将换出到磁盘时，先尝试在内存中进行压缩缓存，从而减少真实的磁盘 I/O。它通过 frontswap 子系统注册钩子，当内核决定换出匿名页时，Zswap 的 `frontswap_store` 会被调用。Zswap 使用 `zpool` 抽象来管理压缩后的内存池，目前主流实现包括 zbud 与 z3fold，二者在页组织与碎片管理上有所不同。

写入路径中，Zswap 先判断内存池是否已达 `zswap.max_pool_percent` 限制，若未达到则进行压缩并存入池中；若已满，则按照 LRU 策略淘汰旧条目，或直接丢弃。最坏情况下，Zswap 会将压缩页写回真实 Swap 设备，从而保证系统整体的换出语义。读取路径同样通过 frontswap 钩子实现，Zswap 负责解压并返回原始页。与 ZRAM 相比，Zswap 位于内存管理子系统更深层次，能够感知页的冷热属性并做出更智能的回写决策，但也因此引入了更复杂的生命周期管理。ZRAM 更适合作为独立块设备直接挂载，而 Zswap 更适合与现有 Swap 分区配合使用。

4 生产级配置实践

在内核启动参数中，可以通过 `zram.num_devices` 指定 ZRAM 设备数量，`zswap.enabled` 控制 Zswap 是否激活，以及 `zpool` 参数选择内存池实现。模块加载后，用户态脚本通常借助 systemd-zram-setup 服务或 udev 规则来自动化设备初始化与格式化。

运行时调优主要通过 sysfs 接口完成。对于 ZRAM，可通过 `/sys/block/zramX/comp_algorithm` 动态切换压缩算法，通过 `/sys/block/zramX/disksize` 调整逻辑容量；对于 Zswap，则主要调节 `zswap.max_pool_percent` 来限制压缩池占总内存的比例。监控指标方面，`/sys/block/zramX/mm_stat` 提供了原始数据大小、压缩后大小、节省内存等关键数值，结合 perf 或 eBPF 可以进一步追踪页错误与压缩延迟，为性能分析提供数据支撑。

5 风险、限制与误区

内存压缩并非银弹。首先，压缩与解压操作会消耗 CPU 周期，在 CPU 饱和的场景下可能导致整体性能下降。其次，元数据与内存对齐引入的额外开销可能造成最坏情况下的内存放大。此外，冷热数据混淆可能导致统计出的压缩率虚高，实际回收效果不及预期。历史上 ZRAM 与 Zswap 均出现过内核 BUG，例如并发访问时的死锁或内存泄漏，社区通过 upstream 修复不断完善稳定性。

6 未来演进

新压缩算法如 zstd 与 lz4hc 持续优化压缩比与延迟；硬件 offload 方案如 Intel QAT 可将压缩任务卸载到专用加速卡，进一步降低 CPU 负担。多层级内存架构的兴起，例如 CXL 与 PMEM，为 Zswap 提供了分层缓存的新可能。eBPF 的引入使得用户态能够编程控制压缩策略，实现更细粒度的冷热数据识别。社区仍在推进 ZRAM 热迁移与 NUMA-aware Zswap 等特性，以适应云原生与异构计算的需求。

选择 ZRAM 还是 Zswap，本质上取决于负载特征与运维偏好。ZRAM 适合对 Swap 延迟敏感且希望简化管理的场景；Zswap 则更适合已有 Swap 分区且希望在内存与磁盘之间建立缓冲层的环境。快速上线时，建议先

通过脚本一键启用 ZRAM，配置合理的磁盘大小与压缩算法；随后部署监控告警，关注压缩率、CPU 使用率与 OOM 事件；最后根据实际负载迭代调优参数。