PR #16946 完整报告

sgl-project/sglang

[HiCache] Prevent KV cache data loss when radix tree node is split b···

合并时间：2026-06-02 06:58

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/16946

执行摘要

- 一句话：修复 HiCache radix 树节点分片导致的 KV 缓存数据丢失
- 推荐动作：值得精读，尤其是 _concat_split_chain 的设计思路——如何将 Python 引用语义与不可变快照之间的冲突转化为廉价的链式恢复。这种延迟恢复模式对于其他先入队再可能变动的场景有借鉴意义。建议补充正式的单元测试，覆盖节点分割后写穿、多次分割、bigram 模式等场景。

功能与动机

PR 描述中报告了一个关键 bug：当两个相同请求先后到达时，第一个请求创建节点并追加到 ongoing_write_through，第二个请求因匹配前缀触发了 _split_node()，导致原节点 key 缩小，而 ongoing_write_through 中仍持有对该节点的引用。后续 writing_check() 从队列取出节点时只获取了缩小后的数据，最终写入存储的只有 1 个 token 而非原始的 375 个。根源在于 Python 引用语义使得修改节点对象会影响到队列中的引用。

实现拆解

1. 入队记录原始长度：在 write_backup() 中，将 self.ongoing_write_through[node.id] = node 改为 self.ongoing_write_through[node.id] = (node, len(node.key))，存储节点引用及其入队时的 key 长度。
2. 新增链式恢复方法：实现 _concat_split_chain()，从当前节点沿 parent 链向上遍历直到累积长度覆盖 backup_len，然后反转顺序、拼接沿途各节点的 key.token_ids、hash_value、host_value。特别处理了 is_bigram 模式下的边界 token 重叠（只保留第一个节点的首 token，后续节点去掉首 token）。
3. 存储写入支持延迟恢复：修改 write_backup_storage() 增加可选 backup_len 参数。如果 backup_len 为 None 或等于当前节点长度，则走快路径直接使用当前数据；否则调用 _concat_split_chain() 获取分割前的完整数据再执行存储写入。写入时 prefix_keys 锚定在链顶节点（最上层父节点），避免重复计算。
4. 确认阶段传递备份长度：在 writing_check() 的 ack 处理中，从 ongoing_write_through 弹出 (node, backup_len)，调用 write_backup_storage(node, backup_len) 确保存储写入使用原始完整数据。
5. 测试与验证：作者在 PR 中以 log_requests=true 模式运行了针对性的端到端测试，并在 Mooncake (RDMA) 部署上进行了 13.5h 的连续浸泡测试。不过本次提交未包含新的单元

测试文件。

关键文件：

- python/sglang/srt/mem_cache/hiradix_cache.py（模块 缓存层；类别 source；类型 core-logic；符号 write_backup_storage, _concat_split_chain）：唯一修改的文件，包含所有核心变更：write_backup 入队方式调整、新增 _concat_split_chain 方法、write_backup_storage 支持备份长度恢复、writing_check 传递备份长度。

关键符号：write_backup_storage, _concat_split_chain

关键源码片段

python/sglang/srt/mem_cache/hiradix_cache.py

唯一修改的文件，包含所有核心变更：write_backup 入队方式调整、新增 _concat_split_chain 方法、write_backup_storage 支持备份长度恢复、writing_check 传递备份长度。

```
def write_backup_storage(self, node: TreeNode, backup_len: Optional[int] = None):
    # 当 backup_len 不为 None 且节点当前长度与备份长度不一致时，
    # 说明节点已被 _split_node() 分割过，需要通过父链恢复原始数据。
    if backup_len is None or len(node.key) == backup_len:
        top, key, hash_value, host_value = node, node.key, node.hash_value, node.host_value
    else:
        # 沿父链向上遍历并拼接，直到累积长度等于 backup_len
        top, key, hash_value, host_value = self._concat_split_chain(node, backup_len)

    # prefix_keys 锚定在链顶节点，避免重复计算前置 hash
    prefix_keys = (
        top.get_prefix_hash_values(top.parent)
        if self.hicache_storage_pass_prefix_keys
        else None
    )

    operation_id = self.cache_controller.write_storage(
        host_value, key, hash_value, prefix_keys, **self._get_extra_pools()
    )
    self.ongoing_backup[operation_id] = node
    node.protect_host()


def _concat_split_chain(self, node: TreeNode, backup_len: int):
    """Recover enqueue-time key/hash/host by walking the split chain."""
    chain, accumulated = [], 0
    current = node
    while current is not self.root_node and accumulated < backup_len:
        chain.append(current)
        accumulated += len(current.key)
        current = current.parent
    # 确保累积长度精确匹配 backup_len，否则触发 assert
```

```
    assert accumulated == backup_len, (
        f"backup chain length mismatch for node {node.id}: "
        f"expected {backup_len}, got {accumulated}"
    )
    chain.reverse() # 从父节点到当前节点
    top = chain[0]
    if top.key.is_bigram:
        # Bigram 模式下相邻节点首尾 token 重叠，只保留第一个节点的全部 token，
        # 后续节点跳过第一个 token（最后一个重叠 token 已在前面包含）
        token_ids = list(chain[0].key.token_ids)
        for n in chain[1:]:
            token_ids.extend(n.key.token_ids[1:])
    else:
        token_ids = []
        for n in chain:
            token_ids.extend(n.key.token_ids)
    key = RadixKey(token_ids, top.key.extra_key, top.key.is_bigram)

    # 只有所有节点的 hash_value 均不为 None 时才拼接 hash
    if all(n.hash_value is not None for n in chain):
        hash_value = []
        for n in chain:
            hash_value.extend(n.hash_value)
    else:
        hash_value = None
    # 拼接 host_value 张量
    host_value = torch.cat([n.host_value for n in chain])

    return top, key, hash_value, host_value
```

评论区精华

- xiezhq-hermann建议使用更轻量的方法：“只记录 (node_id, backup_len)，在 ack 时沿 node.parent 遍历拼接，恢复原始数据”。理由是无昂贵的节点克隆，不需要 mutate-then-restore，快路径仍零开销。
- 结论：作者采纳并实现了 walk-and-concat 方案，替代了最初的 snapshot 方法。这成为最终合并的实现。
- Copilot指出最初 snapshot 方案中 RadixKey(token_ids=node.key.token_ids.clone()) 会运行时错误（token_ids 是 list 或 sliceable，无 .clone() 方法），且缺少 extra_key/is_bigram 保留。另外 hash_value 的 None 判断应使用 is not None。
- 结论：最终方案不再需要 snapshot，这些点不再适用。
- xiezhq-hermann建议使用向量化操作提高效率，并精简注释。
- 结论：作者在最终提交中将注释压缩，并使用 extend 方法提升拼接效率。
- 修复方案的设计取舍：snapshot 与 walk-and-concat (design): 作者采纳了 walk-and-concat 方案，最终实现从 snapshot 改为记录长度 + 链式恢复。

- 快照实现中的 API 正确性问题 (correctness): 由于最终方案放弃了 snapshot，这些具体缺陷不再存在。
- 代码简洁性与效率改进建议 (style): 作者在提交 5fb61e6 中压缩了注释并改用 extend 提升效率。

风险与影响

- 风险：
  1. 性能风险：在节点分割路径上，_concat_split_chain 需要向上遍历父链并拼接数据，引入了额外开销。不过分割场景发生率低、链长度有限（通常仅 2-3 层），且不影响未分割的快路径，因此整体性能影响可控。
  2. 自动化测试缺失：本次提交未包含新的单元测试或集成测试覆盖新路径。虽然作者提供了手动浸泡验证，但缺乏可重复的 CI 测试来防止回归。
  3. 并发安全：所有 radix cache 操作在调度器单线程中执行，ongoing_write_through 无竞争条件，线程安全已有保证。
  4. 兼容性：仅影响 HiCache 写穿与存储后端（如 Mooncake）交互的路径，不改变公共 API 或数据结构格式，向后兼容。
  5. 异常处理：Walk-and-concat 中有 assert 检查累积长度是否等于 backup_len，若因意料之外的树结构导致 assert 失败会直接崩溃，需要生产环境谨慎对待。 - 影响：用户：任何使用 HiCache 写穿模式（尤其是结合 Mooncake 等远程存储）的用户都将受益，避免了因节点分割导致的数据丢失。数据完整性得到保证。系统：影响范围限于 hiradix_cache.py 中写穿确认流程的存储写入步骤，不影响缓存命中最热路径。不会增加额外网络开销或存储负载。团队：修复了一个难以发现的临界竞争 bug，为 HiCache + 存储后端的可靠性夯实了基础。后续在类似 split 场景下可参考该模式。
- 风险标记：缺少自动测试覆盖，核心路径变更，断言崩溃风险，性能影响小

关联脉络

- PR #25173 Refactor NIXL hicache. Add O_DIRECT support: 同样是 HiCache 模块的重构，修改了同一文件 hiradix_cache.py 及存储后端交互逻辑，与本 PR 有直接的模块关联。
- PR #26919 Split SWA leaf to one window on insert: 涉及 radix cache 节点分割逻辑（SWA 叶子），与本 PR 的分割恢复问题具有相似性，可对照参考。