缓存更新的策略

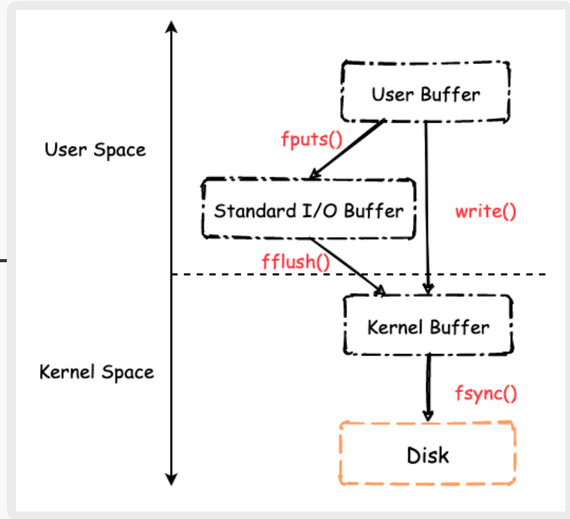
前言

缓存更新是一个非常有趣的话题，关于如何更新缓存我们有很多种选择方式，比如先更新缓存，再更新 DB，或者是先更新 DB 再更新缓存，又或者是先更新 DB，后删除缓存，等等。每一种方式都会带来不同的结果，我们需要做的事情就是根据自身的业务场景和对数据一致性的容忍程度，综合性地进行考量

缓存更新的常见方式

❶ 先更新缓存，再异步更新 DB

Linux Page Cache 就是这么干的，在写入数据时，先将数据写入 Page Cache 中，然后再使用异步任务集中地将 Cache 数据写入硬盘中

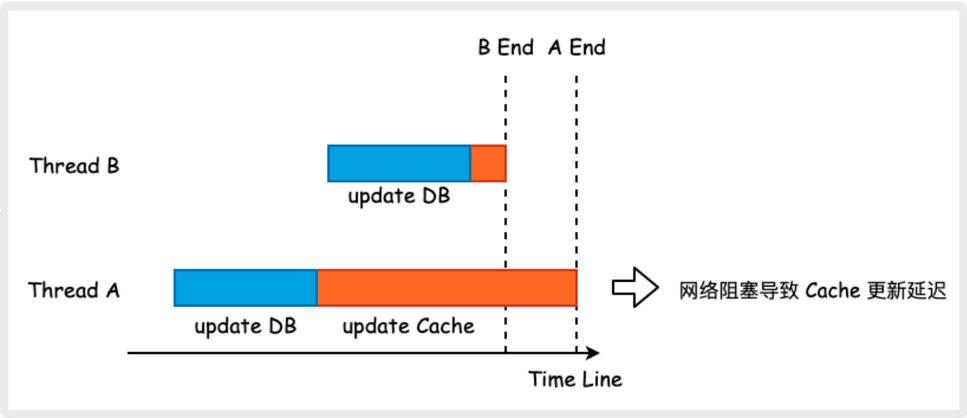


- Linux 这样的写入方式能够有效地减轻硬盘的写入压力，并提高写入效率
- 此时，缓存更多的是充当缓冲区的角色，'cache' 这一语义被弱化

这种方式又被称之为 Write Behind Caching Pattern，Linux 实现上比较复杂，因为需要跟踪哪些数据被更新，并且需要刷新至 DB 中。不过对于应用层来说，我们可以将更新的内容写入 MQ，再从 MQ 集中处理。只不过这样的方式又引入了额外的组件，增加了更多的不确定性

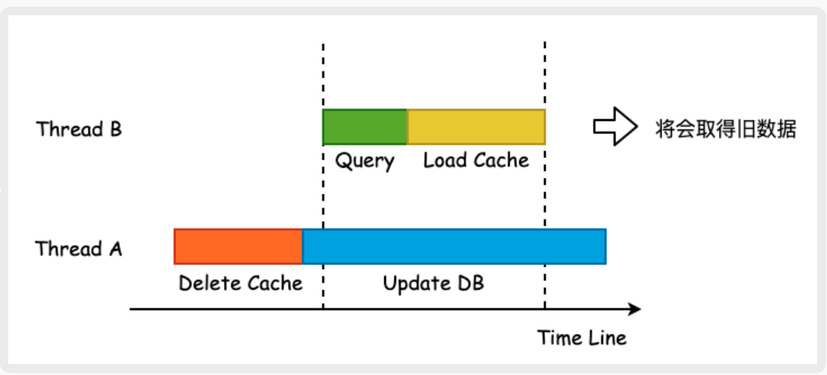
这里再讨论下问什么不同步更新 DB。一方面是因为更新的数据可能需要进行额外的检查，比如唯一性检查。另一方面就是当 DB 更新失败以后，我们需要删除已更新的缓存，避免出现脏数据，相当于做了一次 "无用功"

❷ 先更新 DB，再更新缓存



如上图所示，这种更新方式存在着严重的数据不一致问题。Thread A 明明先执行，但是却在 Thread B 执行完毕后才将缓存写入，即旧有的数据覆盖了最新修改的数据，导致缓存和 DB 两者之间出现不一致的情况

❸ 先删除缓存，再更新 DB

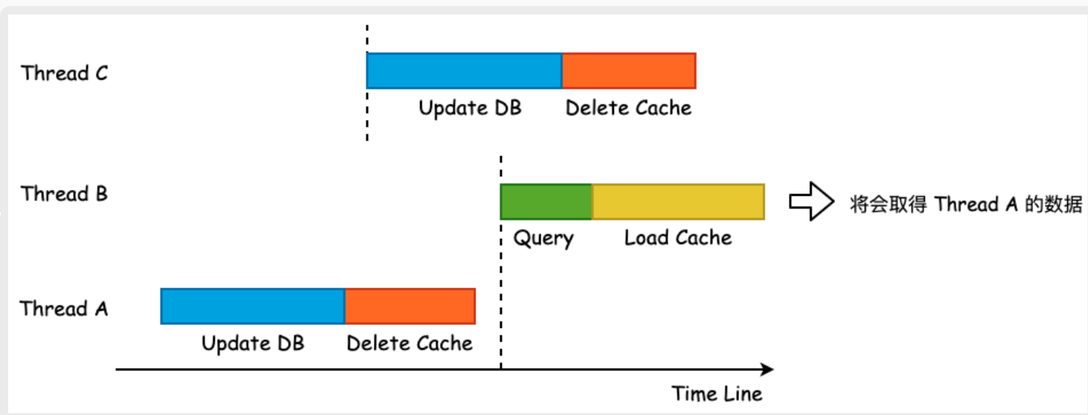


先删除缓存，再更新 DB，在 DB 未更新成功之前，读取数据将会 miss cache，故而从 DB 中取数据，此时取出的数据为旧值。DB 更新完毕后，那么此时缓存中的数据就是错误数据，并且会一直错到缓存过期或者是下一次的数据更新

❹ 先更新 DB，再删除缓存

这种方式被称之为 Cache Aside Pattern，同时也是 spring-cache 所默认的实现方式

Cache Aside Pattern 能够解决多数的不一致情况，或者说，在绝大多数场景下 Cache Aside Pattern 都能够保证良好的数据一致性。但是，在某些极端高并发的场景下，Cache Aside Pattern 仍然会导致数据不一致的情况

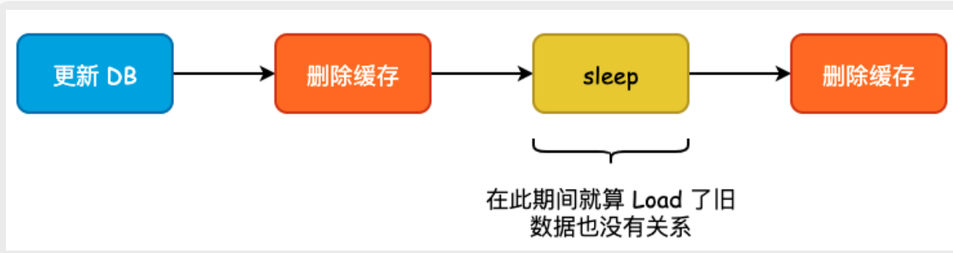


如上图所示，当 Thread B 读取数据时，由于 Thread A 删除 cache 过程结束，所以会发生 cache miss，Thread B 将从 DB 中取出数据，并 Load 进缓存中。若此时又恰好有 Thread C 进行数据更新的话，那么就有可能导致 Thread C 在 Thread B 之前完成，Thread C 在 delete cache 时其实是删除了空数据，那么 Thread B Load 进缓存的数据其实是 Thread A 写入的旧数据，导致数据再次出现不一致

实际上，上述情况出现的条件非常苛刻，必须要在某一时间区间内同时存在两个或多个写入和多个读取。而一般业务场景下更新 DB + 删除缓存的操作通常能够在 200ms 以内返回，出现上述情况的概率极低

延时双删

对于 Cache Aside Pattern 来说，在并发量较高的情况下可能会出现数据不一致的问题，如果我们能够保证缓存一定会被删除的话，就能够尽可能地缩短数据不一致的窗口。最常用的方式就是延时双删，当删除完缓存以后，休眠一段时间，再进行删除。定时任务可由 MQ 或者是定时任务系统来完成



此时我们能够将数据不一致的窗口缩短到一个非常短的时间，最长也就是 sleep 的时间

但是，延时双删虽然能够将数据不一致的窗口缩短，但是由于多删除了一次缓存，也就意味着在极端情况下 DB 的查询压力是原来的两倍，会有大量的查询直接落入 DB 中，此时系统性能可能还不如双更新策略

订阅 binlog

采用订阅 binlog 的方式是一种看起来更美好的方式，无需在业务端对 Redis 进行任何的更新/删除动作，所有的更新都交由 binlog 订阅线程处理

之所以说看起来很美好的原因在于解析 binlog 以及如何更新缓存会比较复杂，并且对业务架构的前期设计要求很高，并且订阅 binlog 通常会引入额外的 MQ 中间件，编码更为复杂