无锁数据结构的设计原理与实现

杨其臻

Sep 28, 2025

1 副标题：告别锁的阻塞，探索高性能并发数据结构的核心奥秘

在现代并发编程中，锁作为传统的同步机制，虽然简单易用，却常常成为系统性能的瓶颈。锁会导致线程串行化，引发上下文切换的开销，并在高并发场景下产生严重的竞争问题。更糟糕的是，锁可能带来死锁、活锁和优先级反转等致命缺陷，随着多核处理器的普及，这些问题的负面影响被放大，限制了系统的可伸缩性。相比之下，无锁数据结构承诺更高的并发度和可伸缩性，能够避免锁相关的经典问题，对于实时系统和低延迟应用具有关键价值。本文旨在帮助读者理解无锁数据结构的核心设计原理，掌握实现无锁结构的关键技术与挑战，并通过经典案例如无锁队列和无锁栈来剖析实现细节，同时探讨无锁编程的适用场景与潜在陷阱。

2 基石：无锁编程的核心概念

无锁编程的核心概念源于对并发控制的重新思考。学术上，无锁被定义为在系统范围的任意延迟或故障下，保证至少有一个线程能够继续执行。通俗地说，无锁编程不使用互斥锁，而是通过原子操作如比较并交换（CAS）来保证并发正确性。关键特性包括无锁、无等待和无阻塞；无锁保证系统整体进度，无等待则更强，确保每个线程的进度都有保障，而无阻塞是上述两者的统称。这些特性使得无锁编程在高性能计算中备受青睐。

硬件基石是支撑无锁编程的关键，原子操作和内存序构成了其基础。比较并交换（CAS）操作是无锁编程的瑞士军刀，它允许原子地比较一个内存位置的值，并在匹配时更新为新值。其他原子指令如取并加（Fetch-And-Add）和加载链接/存储条件（LL/SC）也在不同架构中发挥作用。内存模型和内存屏障则确保指令执行的正确顺序；指令重排可能导致意想不到的并发问题，因此需要内存屏障来强制内存访问顺序。在 C++ 中，std::memory_order 提供了不同级别的内存序，例如宽松（relaxed）、获取（acquire）、释放（release）、获取释放（acq_rel）和顺序一致（seq_cst），这些枚举值帮助开发者精确控制内存可见性。

3 挑战：无锁编程的“三座大山”

无锁编程虽然强大，却面临三大挑战：ABA 问题、内存回收问题以及复杂度与正确性问题。ABA 问题描述了一个值从 A 变为 B 又变回 A，而 CAS 操作无法感知中间变化，这在链表操作中尤为常见，例如节点被释放后重用可能导致数据损坏。解决方案包括标签指针，它利用指针的高位作为版本号来检测变化；风险指针，线程声明自己正在访问的指针以防止其被回收；以及引用计数，通过原子计数管理对象生命周期，但在无锁环境下实现较为复杂。

内存回收问题涉及当一个线程准备释放内存时，另一个线程可能仍持有该内存的指针并准备访问，这会导致悬

空指针和未定义行为。解决方案有多种：风险指针通过线程本地存储来标记正在使用的指针；引用计数在无锁环境下需要原子操作来维护；基于时代的回收（Epoch Based Reclamation）适用于读多写少的场景，它将回收延迟到安全时期；基于静默状态的回收（Quiescent State Based Reclamation）常用于读-复制-更新（RCU）模式，它在线程进入静默状态时回收内存。复杂度与正确性问题体现在代码难以设计和验证，测试困难且竞态条件难以复现，同时对平台和编译器的依赖性强，这要求开发者在实现时格外谨慎。

4 实战：从零实现一个无锁队列

在设计无锁队列时，我们通常选择基于链表的实现，因为它能动态扩展并避免固定容量的限制。队列结构包含头指针和尾指针，指向单链表的起始和结束节点。核心操作包括入队（Enqueue）和出队（Dequeue），这些操作通过原子指令确保并发安全。

数据结构定义是基础，我们使用节点结构来存储数据和下一个指针，队列结构则维护头尾指针。例如，在 C++ 中，我们可以定义节点为包含整型数据和节点指针的结构，队列类则封装头尾指针及其操作。

入队操作涉及三个关键步骤：首先创建新节点，然后循环使用 CAS 更新尾节点的 next 指针，最后处理“滞后尾”问题，即通过另一个 CAS 循环更新队列的 tail 指针。出队操作则包括读取 head、tail 和 next 指针，判断队列是否为空，循环 CAS 更新 head 指针，并处理出队数据的内存回收，这里可以集成之前讨论的内存回收方案，例如使用风险指针来安全释放内存。

以下是一个简化的 C++ 代码示例，展示无锁队列的核心部分。我们使用 std::atomic 来实现原子操作，并添加详细注释以解释关键步骤。

```
#include <atomic>

struct Node {
    int data;
    std::atomic<Node*> next;
    Node(int value) : data(value), next(nullptr) {}
};

class LockFreeQueue {
private:
    std::atomic<Node*> head;
    std::atomic<Node*> tail;
public:
    LockFreeQueue() {
        Node* dummy = new Node(0);
        head.store(dummy);
        tail.store(dummy);
    }

    void enqueue(int value) {
```

```
    Node* newNode = new Node(value);
    while (true) {
        Node* currentTail = tail.load(std::memory_order_acquire);
        Node* nextTail = currentTail->next.load(std::memory_order_acquire);
        if (currentTail == tail.load(std::memory_order_acquire)) {
            if (nextTail == nullptr) {
                if (currentTail->next.compare_exchange_weak(nextTail, newNode, std::
                    ↪ memory_order_release)) {
                    tail.compare_exchange_weak(currentTail, newNode, std::
                        ↪ memory_order_release);
                    return;
                }
            } else {
                tail.compare_exchange_weak(currentTail, nextTail, std::
                    ↪ memory_order_release);
            }
        }
    }
}

int dequeue() {
    while (true) {
        Node* currentHead = head.load(std::memory_order_acquire);
        Node* currentTail = tail.load(std::memory_order_acquire);
        Node* nextHead = currentHead->next.load(std::memory_order_acquire);
        if (currentHead == head.load(std::memory_order_acquire)) {
            if (currentHead == currentTail) {
                if (nextHead == nullptr) {
                    return -1; // 队列为空
                }
                tail.compare_exchange_weak(currentTail, nextHead, std::
                    ↪ memory_order_release);
            } else {
                int value = nextHead->data;
                if (head.compare_exchange_weak(currentHead, nextHead, std::
                    ↪ memory_order_release)) {
                    // 此处需处理内存回收，例如使用风险指针
                    delete currentHead;
                    return value;
                }
```

```
            }
        }
    }
}
};
```

在这段代码中，enqueue 函数通过循环 CAS 确保新节点被正确链接到队列尾部。首先，它加载当前尾指针和其 next 指针，然后检查尾指针是否未被其他线程修改。如果 next 指针为空，则尝试原子地将其设置为新节点，成功后更新尾指针。dequeue 函数类似，它加载头指针和尾指针，检查队列状态，如果队列非空，则原子地更新头指针并返回数据。内存回收部分在出队时删除旧头节点，但实际应用中需集成风险指针等机制以避免 use-after-free 错误。内存序参数如 std::memory_order_acquire 和 std::memory_order_release 确保操作的有序性，防止指令重排带来的问题。

另一种思路是基于数组的无锁环形缓冲区，它通过模运算操作下标实现循环访问。优点是内存连续、缓存友好且实现简单，但缺点在于容量固定，适用于生产者-消费者场景。相比之下，链表实现更灵活，但复杂度更高。

5 进阶：更多无锁数据结构掠影

除了无锁队列，无锁栈是理解无锁编程的理想起点，它通过 CAS 原子地更新栈顶指针来实现推送和弹出操作。无锁哈希表通常采用锁分段思想，将哈希桶划分为独立的无锁结构，如无锁链表，以减少竞争。全无锁哈希表实现更为复杂，但能提供更高的并发度。无锁链表是所有无锁结构的基础，支持完整的增删改查操作，但实现难度大，需要处理 ABA 问题和内存回收等挑战。

6 现实考量：何时使用以及如何正确使用

无锁编程并非银弹，它适用于高并发场景，当锁竞争成为主要瓶颈时，无锁结构能显著提升性能。在需要极低延迟和确定性响应的系统，如实时计算或金融交易中，无锁方案尤为关键。然而，在并发度低、业务逻辑复杂或团队经验不足的情况下，无锁实现可能得不偿失，反而引入难以调试的错误。

最佳实践包括优先使用成熟库如 Intel TBB、Boost.Lockfree 或 Java 的 JUC，以减少自行实现的風險。充分测试是必须的，使用线程检查工具如 TSAN 进行压力测试，以捕捉竞态条件。保持代码简单明了，避免过度优化，同时通过代码审查让多人参与，以发现潜在的并发问题。

无锁编程通过原子操作和 CAS 循环替代锁，以换取更好的可伸缩性，但带来了 ABA 问题、内存回收等新挑战。随着硬件发展，如硬件事务内存（HTM）的兴起，并发编程的未来将更加多元化。最终建议是，在性能瓶颈确实存在且由锁引起时，再谨慎考虑无锁方案，理解原理比盲目使用更为重要。