7.3 对于设计无锁数据结构的指导建议

本章中的例子中，看到了一些复杂的代码可让无锁结构工作正常。如果要设计自己的数据结构，一些指导建议可以帮助你找到设计重点。第6章中关于并发通用指导建议还适用，不过这里需要更多的建议。我从例子中抽取了几个实用的指导建议，在你设计无锁结构数据的时候就可以直接引用。

7.3.1 指导建议：使用 std::memory_order_seq_cst 的原型

`std::memory_order_seq_cst` 比起其他内存序要简单的多，因为所有操作都将其作为总序。本章的所有例子，都是从 `std::memory_order_seq_cst` 开始，只有当基本操作正常工作的时候，才放宽内存序的选择。在这种情况下，使用其他内存序就是进行优化(早起可以不用这样做)。通常，当你看整套代码对数据结构的操作后，才能决定是否要放宽该操作的内存序选择。所以，尝试放宽选择，可能会让你轻松一些。在测试后的时候，工作的代码可能会很复杂(不过，不能完全保证内存序正确)。除非你有一个算法检查器，可以系统的测试，线程能看到的所有可能性组合，这样就能保证指定内存序的正确性(这样的测试的确存在)，仅是执行实现代码是远远不够的。

7.3.2 指导建议：对无锁内存的回收策略

这里与无锁代码最大的区别就是内存管理。当有其他线程对节点进行访问的时候，节点无法被任一线程删除；为避免过多的内存使用，还是希望这个节点在能删除的时候尽快删除。本章中介绍了三种技术来保证内存可以被安全的回收：

- 等待无线程对数据结构进行访问时，删除所有等待删除的对象。
- 使用风险指针来标识正在被线程访问的对象。
- 对对象进行引用计数，当没有线程对对象进行引用时，将其删除。

在所有例子中，主要的想法都是使用一种方式去跟踪指定对象上的线程访问数量，当没有现成对对象进行引用的时候，将对象删除。当然，在无锁数据结构中，还有很多方式可以用来回收内存。例如，理想情况下使用一个垃圾收集器。比起算法来说，其实现更容易一些。只需要让回收器知道，当节点没被引用的时候，回收节点，就可以了。

其他替代方案就是循环使用节点，只在数据结构被销毁的时候才将节点完全删除。因为节点能被复用，那么就不会有非法的内存，所以这就能避免未定义行为的发生。这种方式的缺点：产生“ABA问题”。

7.3.3 指导建议：小心ABA问题

在“基于比较/交换”的算法中要格外小心“ABA问题”。其流程是:

1. 线程1读取原子变量x，并且发现其值是A。
2. 线程1对这个值进行一些操作，比如，解引用(当其是一个指针的时候)，或做查询，或其他操作。
3. 操作系统将线程1挂起。
4. 其他线程对x执行一些操作，并且将其值改为B。
5. 另一个线程对A相关的数据进行修改(线程1持有)，让其不再合法。可能会在释放指针指向的内存时，代码产生剧烈的反应(大问题)；或者只是修改了相关值而已(小问题)。
6. 再来一个线程将x的值改回为A。如果A是一个指针，那么其可能指向一个新的对象，只是与旧对象共享同一个地址而已。
7. 线程1继续运行，并且对x执行“比较/交换”操作，将A进行对比。这里，“比较/交换”成功(因为其值还是A)，不过这是一个*错误的A*(the wrong A value)。从第2步中读取的数据不再合法，但是线程1无法言明这个问题，并且之后的操作将会损坏数据结构。

本章提到的算法不存在这个问题，不过在无锁的算法中，这个问题很常见。解决这个问题的一般方法是，让变量x中包含一个ABA计数器。“比较/交换”会对加入计数器的x进行操作。每次的值都不一样，计数随之增长，所以在x还是原值的前提下，即使有线程对x进行修改，“比较/交换”还是会失败。

“ABA问题”在使用释放链表和循环使用节点的算法中很是普遍，而将节点返回给分配器，则不会引起这个问题。

7.3.4 指导建议：识别忙等待循环和帮助其他线程

在最终队列的例子中，已经见识到线程在执行push操作时，必须等待另一个push操作流程的完成。等待线程就会被孤立，将会陷入到忙等待循环中，当线程尝试失败的时候，会继续循环，这样就会浪费CPU的计算周期。当忙等待循环结束时，就像一个阻塞操作解除，和使用互斥锁的行为一样。通过对算法的修改，当之前的线程还没有完成操作前，让等待线程执行未完成的步骤，就能让忙等待的线程不再被阻塞。在队列例中，需要将一个数据成员转换为一个原子变量，而不

是使用非原子变量和使用“比较/交换”操作来做这件事；要是在更加复杂的数据结构中，这将需要更加多的变化来满足需求。