原子操作和非阻塞算法

章子涵

2024 年 11 月

目录

1 多线程同步问题和同步原语

多线程的引入在提高性能的同时也带来了新的问题，主要是如何能够保证多个线程的同步执行。为了解决这些问题，操作系统提供了多种同步原语，包括互斥锁、信号量、条件变量等。这些同步原语可以帮助开发者编写出正确同步的代码，从而避免多线程竞争条件和死锁等问题。

1.1 多线程同步问题

多线程的同步问题，比如如果一个线程正在读取一个变量，而另一个线程正在修改这个变量，那么读取变量的值就可能是一个不稳定的值。为了解决这种数据竞争带来的问题，需要引入同步方法。

比较常用的同步方法主要有两种：互斥和条件同步。互斥是说，当一个线程进入了某个和其他线程共享数据的代码块，此时其他线程如果也想访问数据，就必须等待。条件同步是说当前线程设置一个条件，其他线程会等到条件满足后才尝试访问会竞争的数据。

1.2 同步原语

基于上面的同步方法，可以设计出多种同步原语用于同步的具体实现。比如信号量、锁和条件变量。

信号量是一个共享的计数器，可以用一个整数变量 sem 来表示。具体而言，如果 sem 为正表示可以同时执行的线程的数目，为负表示已经被阻塞的线程数，为 0 表示当前没有等待的线程，也没有多余的可以并行的线程数。对信号量有两个基本的原子操作：P(wait, 检测操作) 和 V(signal, 增量操作)。

- P 操作：sem 减 1，如果 sem 大于等于 0，线程继续执行；如果 sem 小于 0，则线程阻塞并等待，直到 sem 大于等于 0。
- V 操作：sem 加 1，如果此时有其他线程在等待，则唤醒一个线程。

锁类似于信号量，不同的地方在于同于时刻只能有一个线程持有锁。锁的两个原子操作分别是：

- 加锁：如果锁空闲，则获取锁，否则阻塞。
- 解锁：释放锁。

利用锁可以实现两个线程的互斥。比如线程 A 要修改数据，就在修改数据前上锁，

修改数据后释放锁，就可以保证没有其他线程在修改期间访问数据。

条件变量是用来通知共享数据状态信息的。当特定的条件满足时，条件变量会唤醒等待的线程。条件变量通常和互斥锁一起使用，比如条件变量 C 使用锁 L 来完成对共享数据的访问，则可以对条件变量 C 进行以下 3 种原子操作：

- Wait(L)：线程释放锁 L，并等待条件 C 满足。
- Signal(L)：线程唤醒一个等待线程。该操作执行完毕后，线程会重新获取锁 L。
- Broadcast(L)：线程唤醒所有的等待线程。该操作执行完毕后，线程也会重新获取锁 L。

条件变量在思路上是锁的互补，锁是防止主动的访问，条件变量是提供了被动的访问，等一个进程被唤醒时，需要程序本身承诺线程安全。

2 原子操作和实现锁

原子操作就是不可分割的操作，操作系统在多个线程间来回切换时，会保证原子操作的完整性。所以直接使用原子操作是安全的。但是原子操作本身实现的功能很有限，直接用它来实现程序非常复杂。为了提供更加方便的同步机制，我们可以通过原子操作的组合来实现锁。

2.1 原子操作

构建同步操作的一个典型操作就是原子交换，它将寄存器中的一个值和存储器中的一个值交换。如果我们有了这个操作，我们就可以构建一个简单的锁，0 表示没有锁住，1 表示锁住了。具体设置锁的方法就是，将寄存器中的 1 和跟这个锁对应的存储器地址交换，如果其他某个处理器已经获取了锁，那么就会返回 1，否则返回 0。

那么基于原子交换，我们就可以设计出测试并置位 (Test-and-Set, TAS) 操作。它包含几个步骤：首先，它读取存储器中的值，然后将存储器中的值设置为新的值，最后返回存储器中的旧值。实现这一功能我们只需要将新的值所在的寄存器和存储器进行原子交换，然后返回寄存器的值即可。

这里测试并置位操作也是原子的，所以我们可以利用这一操作来实现一个锁。

2.2 基于 TAS 和 TTAS 自旋锁

基于 TAS，我们可以设计一个简单的 TAS 自旋锁。

```
1  public class TASLock {
2      private AtomicBoolean state = new AtomicBoolean(false);
```

```
    public void lock() {
        while (state.getAndSet(true));                    // 自旋等待
    }
    public void unlock() {
        state.set(false);
    }
}
```

其中state.getAndSet()实现的结果和上面讲的 TAS 操作相同。它会将目标的值设置为参数里的值，并返回目标原来的值。如果当前锁空闲 (值为 0)，有线程要请求锁，那么state.getAndSet(true)会把锁置为 1，并且返回 false。如果当前锁被占用 (值为 1)，那么state.getAndSet(true)会返回 1，导致线程进入自旋等待，直到锁被释放。

这种做法好处是能够在第一时间获得锁，但是坏处很明显，等待锁的过程中会不停交换寄存器和存储器的值，占用的资源过多。下面给出的 TTAS 自旋锁给了一个更好的解决方案。

```
public class TTASLock {
    private AtomicBoolean state = new AtomicBoolean(false);
    public void lock() {
      while (true) {
          while (state.get());                    // 自旋等待
          if (state.getAndSet(true)) {
              return;
          }
      }
    }
    public void unlock() {
        state.set(false);
    }
}
```

测试-测试-设置 (Test-Test-and-Set, TTAS) 锁通过在获取锁之前先进行一次测试来减少不必要的原子操作。它首先检查锁的状态，如果锁已经被占用，则进入自旋等待。如果锁是空闲的，再进行原子交换操作来获取锁。

虽然它能减少不必要的原子操作，但是它在自旋等待期间占用的资源也不少。

2.3 后退锁和队列锁

后退锁是为了针对激烈竞争而设计的锁。它是指当线程尝试请求锁时，发现锁在某一瞬间空闲，但是在下一瞬间又被占用了，这时候它可以推断出这把锁可能竞争很激烈，

所以它决定等一会再试。具体实现可以把逻辑封装到一个简单的 Backoff 类中，其中 minDelay 是最小的时延 (线程后退时间太短是没有意义的)，maxDelay 是最大的时延 (防止后退的时间太多)。每次请求失败就会增加最大时延，下面的代码中采用的是指数后退。

```
public class Backoff {
    final int minDelay, maxDelay;
    int limit;
    final Random random;
    public Backoff(int min, int max) {
        minDelay = min;
        maxDelay = max;
        limit = minDelay;
        random = new Random();
    }
    public void backoff() throws InterruptedException {
        int delay = random.nextInt(limit);
        limit = Math.min(maxDelay, limit * 2);
        Thread.sleep(delay);
    }
}
```

基于这样的 Backoff 类，可以设计出后退锁。

```
public class BackoffLock {
    private AtomicBoolean state = new AtomicBoolean(false);
    private static final int MIN_DELAY = ...;
    private static final int MAX_DELAY = ...;
    public void lock() {
        Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
        while (true) {
            while (state.get());                // 自旋等待
            if (!state.getAndSet(true)) {
                return;
            }
            backoff.backoff();
        }
    }
    public void unlock() {
        state.set(false);
    }
}
```

要注意，只有当线程发现一个锁为空闲且不能立即获得该锁时才会后退。观测到锁被另一个线程所持有并不能说明竞争的激烈程度。

不过这看上去就不是一个很聪明的做法。多人同时抢夺一把锁，最好能直接分配给每个人，而不是通过抢夺。抢夺要么会造成额外的开销，比如上面讲的 TAS 和 TTAS，要么会造成不必要的时延，比如后退锁可能会造成一个锁已经被释放了，但是要请求它的线程还在时延。一个很直接的想法应该是给这些请求排队，按照队列的顺序依次放锁。

不过如何维护这个队列是一个问题，这个队列也是需要线程安全的，我们不可能为了维护这样的队列而重新使用上面的锁。

```
1  public class CLHLock {
2      AtomicReference<QNode> tail;
3      ThreadLocal<QNode> myPred;
4      ThreadLocal<QNode> myNode;
5      public CLHLock() {
6          tail = new AtomicReference<QNode>(null);
7          myNode = new ThreadLocal<QNode>() {
8              QNode initialValue() {
9                  return new QNode();
10             }
11         };
12         myPred = new ThreadLocal<QNode>() {
13             QNode initialValue() {
14                 return null;
15             }
16         };
17     }
18     public void lock() {
19         QNode qnode = myNode.get();
20         qnode.locked = true;
21         QNode pred = tail.getAndSet(qnode);
22         myPred.set(pred);
23         while (pred.locked);                // 自旋等待
24     }
25     public void unlock() {
26         QNode qnode = myNode.get();
27         qnode.locked = false;
28         myNode.set(myPred.get());
29     }
30 }
```

其中，`tail.getAndSet(qnode)`是一个原子操作，可以让`tail`的值设置为`qnode`，返回`tail`之前的结果。`ThreadLocal`是 Java 提供的一种数据结构，它可以在每个线程中存储一个值，这个值对其他线程是不可见的。在这里，我们用它来存储它在队列中的前一项。每个线程如果要请求锁，会把队尾的节点设置为当前节点，并将之前的队尾设置成当前节点的

前置。每个线程都会等待前置节点释放锁后再获得锁，这样就形成了一个有序的队列。

3 非阻塞算法

前面讲的锁的方法来保证线程安全，它们的使用方法比较简单，但缺点是当线程需要等待锁时，会进入阻塞状态，这会导致 CPU 资源的浪费。要解决这种问题，一种方法就是不使用锁，按照这种思路来设计的算法就成为非阻塞算法。非阻塞算法的本质特征就是停止一个线程的执行不会阻碍到系统中其他执行实体的运行。它具有以下的特征:

- 无锁：不阻塞线程的执行。
- 无阻塞：当线程无法获取资源时，不会阻塞，而是继续执行。
- 无等待：每个线程都可以持续执行，即使遇到竞争也是如此。不过只有极少数的非阻塞算法实现了这一点。

3.1 使用 CAS 实现非阻塞算法

现在的处理器使用的最通用的方法是实现名为“比较并转换”(Compare-and-Swap, CAS) 的原语。

CAS 操作包含三个操作数：内存位置 (V)、预期原值 (A) 和新值 (B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会返回该位置的值。它其实就是在描述这样的要求“我认为位置 V 应该包含值 A，如果正确就把新值 B 写入，否则不更新，只需要告诉我现在这个位置的值就可以。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行计算后尝试写入新值 B。如果在这段时间内没有其他线程修改该位置的值，那么 CAS 操作会成功，否则会失败。

下面我们用 CAS 操作来实现一个非阻塞的计数器。

```
1   public class NonblockingCounter {
2       private AtomicInteger value;
3       public int getValue() {
4           return value.get();
5       }
6       public int increment() {
7           int v;
8           do {
9               v = value.get();
10          } while (v != value.compareAndSet(v, v + 1));
11          return v + 1;
```

```
        }
    }
```

这里的compareAndSet()方法就是 CAS 操作。它会比较value的值是否为v，如果是，就更改v为v+1并且返回value的值，如果不是，就只返回value当前的值。如果当前进程执行了v=value.get()后存在其他进程修改了value的值，那么compareAndSet(v, v+1)就会失败，然后当前进程会重新读取value的值，并重新尝试写回，直到成功为止。通过这样的方法，可以保证最后value的值的增量恰好等于所有线程调用increment()方法的次数。

3.2 非阻塞栈和队列

上面的非阻塞计数器是一个比较简单的例子，稍微复杂一点的例子是非阻塞栈和队列。

在设计非阻塞算法的时候，首先需要考虑怎么做可能让数据发生变化。比如栈，如果要改变栈的数据，必须从栈的顶部操作。根据这一特性，我们可以设计一个非阻塞栈，如果栈顶的元素没有被修改，那么认为栈没有被其他线程修改。如果这里你发现如果先弹出两个元素，再插入原来的栈顶，这样栈顶元素仍然不变，但是栈的元素却发生了变化。这个问题就是我们后面会讲到的 ABA 问题，这里你可以先认为栈顶元素不变可以推出栈的元素不变。

```
public class ConcurrentStack<E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();
    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }
    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }
```

```
22          static class Node<E> {
23              final E item;
24              Node<E> next;
25              public Node(E item) {
26                  this.item = item;
27              }
28          }
29      }
```

本质上就是栈顶的元素没有变，那么就保持当前的操作，否则重新尝试。比如维护一个单调栈，这样的操作可以保证栈的单调性。

上面的例子都是非常简单的非阻塞算法，因为栈和计数器在修改的时候只涉及到一个指针。下面的队列就可能涉及到对多个指针的更新。由于 CAS 支持对单一指针的原子性的条件更新，但是不支持两个以上的指针。比如对于队列的插入过程，需要将tail指针的next指向新节点，并且将tail指针指向新节点。

对于稍微复杂一点的数据结构，构建非阻塞算法的思路是要让线程能够判断出是否有其他线程仍在更新数据的途中，如果是，那么它可以帮正在执行更新的线程完成更新。这种“帮助邻居”的要求，可以让数据结构免受单个线程失败的影响。具体而言，对于队列的插入过程，如果某个线程发现当前的tail指针指向的元素的next指针指向非空，那么说明有其他线程更新了第一步，但是第二步也就是更改tail指针还未完成。因此，它可以帮助这个线程完成第二步，然后再进行它自己的操作。

```
1   public class ConcurrentLinkedQueue<E> {
2       private static class Node<E> {
3           final E item;
4           final AtomicReference<Node<E>> next;
5           public Node(E item, Node<E> next) {
6               this.item = item;
7               this.next = new AtomicReference<Node<E>>(next);
8           }
9       }
10      private AtomicReference<Node<E>> head = new AtomicReference<Node<E>>(
            new Node<E>(null, null));
11      private AtomicReference<Node<E>> tail = head;
12      public boolean enq(E item) {
13          Node<E> newNode = new Node<E>(item, null);
14          while (true) {
15              Node<E> curTail = tail.get();
16              Node<E> residue = curTail.next.get();
17              if (curTail == tail.get()) {
18                  if (residue == null) {
```

```
19                          if (curTail.next.compareAndSet(null, newNode)) {
20                              tail.compareAndSet(curTail, newNode);
21                              return true;
22                          }
23                      } else {
24                          tail.compareAndSet(curTail, residue);
25                      }
26                  }
27              }
28          }
29      }
```

其中，第 24 行就是当其他线程更新了第一步，但是第二步还未完成时，当前进程会帮助这个线程完成第二步。

对于删除操作，如果不涉及内存问题，就只需要操作一个指针，这就和前面的差不多了。只需要对head指针的修改的时候使用 CAS 操作，如果head未被修改则可以继续操作。

```
1   public E deq() {
2       Node<E> oldHead, newHead;
3       do {
4           oldHead = head.get();
5           if (oldHead == null)
6               return null;
7           newHead = oldHead.next.get();
8       } while (!head.compareAndSet(oldHead, newHead));
9       return oldHead.item;
10  }
```

3.3 内存问题和 ABA 问题

当然，如果依赖 Java 自带的垃圾回收器会比较简单。但是，由类本身来提供自己的内存管理往往有更高的效率，特别是在类创建和释放许多小的对象时。以及，如果进程本身是无锁的，肯定希望自己的垃圾回收器也是无锁的。

一个很自然的方法就是让每个线程维护自己的回收队列，当当前线程需要插入一个元素时，优先从回收队列中取结点。因为这个队列是线程本地的，所以不需要很大的同步开销。当然这样也有缺点，就是空间的回收率不好说，如果一个线程一直插入，另一个一直删除，那么空间利用率就很低了。

那如果要让所有线程一起维护一个公共的回收队列，这会带来额外的问题。考虑只

有 2 个元素 a,b 的队列，线程 A 要删去队头的元素 a，那么它首先读取了队头。此时有其他的线程清空了队列，并重新插入了一个元素，恰好这个插入的元素是从回收队列取的空间 a，那么线程 A 这时候判断时就会发现队首元素和原来读取的一样，所以把head指针指向了原来 a 的next，也就是空间 b。这就造成了错误，空间 b 已经被回收了。

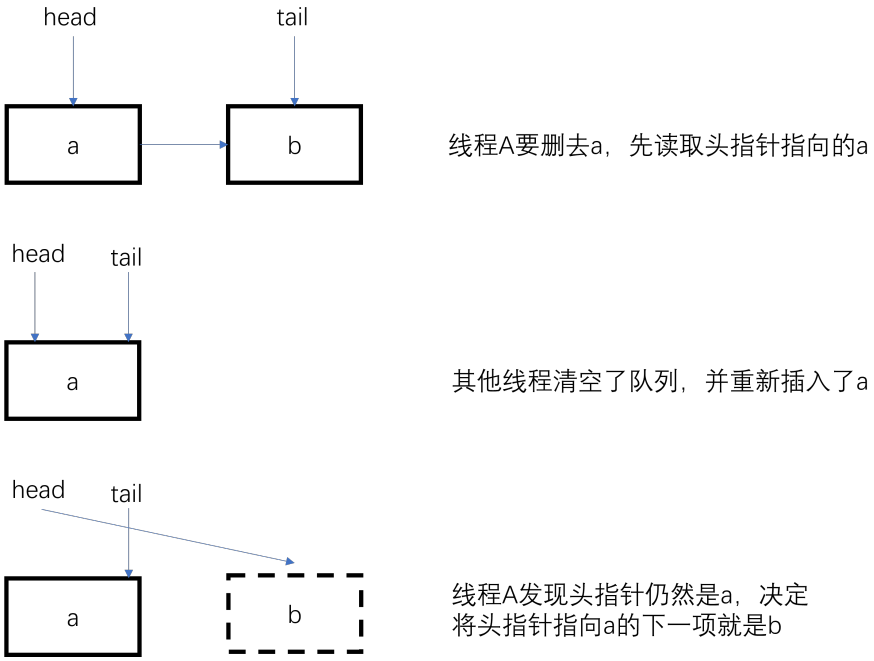

图 1: 公共回收队列的 ABA 问题

这个问题发生的本质原因就是，线程 A 根据队首元素和过去读取的元素相同，这一点判断出队首没有发生改变。这种现象就是 ABA 问题。上面我们提到的栈的栈顶元素不变但是栈的元素发生了改变，也是这个问题。

解决这个问题的一种直接的方法就是给每一个原子引用附上一个唯一的时间戳。每次更新的时候，时间戳加 1。这样，即使两个元素相同，只要时间戳不同，那么它们就是不同的。

此外，还有一种解决方法是使用保留载入和条件存储这一对指令。保留载入是将 rs1 指示的存储器的内容加载到 rd 中，并在该存储器地址上创建一个保留。条件存储将 rs2 的值存储到 rs1 指向的存储器中。如果存在对同一内存地址的写操作破坏了对该载入的保留，那么条件存储失败并将非零值写入 rd，如果成功，则将 0 写入 rd。

利用这样的一组指令，可以检测一个值在两个时间点之间是否改变过，而不是像前面那样检测这个值在两个时间点是否刚好相等的方式来判断是否改变过。