无锁并发队列的实现

黎永基

目录

[背景知识 1](#_Toc1724248175)

[应用场景 2](#_Toc2020765755)

[一般指导原则 2](#_Toc2002246468)

[例子：无锁并发队列 3](#_Toc301561541)

[初始化 3](#_Toc993525188)

[入队 3](#_Toc81953772)

[出队 5](#_Toc148075813)

[查看队头 6](#_Toc1168035092)

[与jdk1.8的ConcurrentLinkedQueue对比 7](#_Toc1771899550)

[参考文献 8](#_Toc220817004)

[思考题 9](#_Toc1666082019)

# 背景知识

“无锁算法”是一种夸张说法，实际上的意思是指没有使用lock/monitor语言机制实现的并发算法。并不是没有使用锁，而是使用了一些“低层次”的锁，例如CPU/JVM运行环境提供的简单高效微型的锁。

CPU有一些特殊的指令（X86,ARM64，RISC等都有，与操作系统无关），支持对内存修改的原子操作。这些指令统称CAS指令（Compare and swap instructions）

使用伪代码表示如下：

|  |
| --- |
| int compare\_and\_swap(int\* target, int expected, int newval){  ATOMIC();  int old\_target\_val = \*target;  **if** (old\_target\_val == expected)  \*target = newval;  END\_ATOMIC();  **return** old\_target\_val;  } |

上述CAS指令执行语义如下：

首先读取内存的一个整数，与期望的“旧值”比较，如果相等，则修改为“新值”。无论比较结果如何以及是否执行修改操作，均会返回内存当前的值（未修改前）。

这个操作是原子的，意思是不管这个内存有多少个CPU正在访问使用，这个CAS指令的修改都不会被打断，而且修改后所有CPU都可见。

强调所有CPU可见是因为每个CPU都存在多级缓存（例如L1/L2/L3），一般的赋值操作实际上是在缓存中发生的，内存并没有马上同步修改。而CAS指令一定会马上写入到内存中，不会只修改缓存而延时更新内存，这样就保证其他CPU通过相同的CAS指令能够读取其他CPU的修改。

|  |
| --- |
| bool compare\_and\_swap(int \*accum, int \*dest, int newval){  **if** (\*accum == \*dest) {  \*dest = newval;  **return** true;  } **else** {  \*accum = \*dest;  **return** false;  }} |

这个指令有以上变体，不直接返回内存当前值，而是返回布尔值表示是否相同，而“旧值”是一个内存地址，如果不能修改内存当前值，就在“旧值”所在的位置返回当前值。因此无论是否修改“目标”内存的值，“旧值”的内存一定是当前目标内存值，返回的布尔值表示是否修改了目标内存

Java volatile关键字（语言提供的微型锁）

* 修饰变量（类变量为主，也可以是局部变量）
* 一定会保持内存同步（不会使用缓存值）
* 单独一个读或者写是原子的（多个读写混合不是原子！例如++i，是先读后写）
* 64位数据如果需要读写原子就需要volatile（long/double在32位机器上不是原子读写）
* 编译器不能对volatile变量进行优化或者缓存，并且写入volatile变量以及之前的所有操作（包括写入非volatele变量）都会同步到内存，使得所有其他线程都可以看到修改后的值

# 应用场景

一般用于这种情况：使用lock/schronized的资源冲突概率较高，而且加锁中的代码执行时间较短，并发效率较低。

效率 = 锁内部执行时间/执行总时间 = 锁内部执行时间/（同步等待锁的时间+锁内部执行的时间）

因此提高效率就是需要降低“同步等待锁的时间”占比

# 一般指导原则

1. 对每个方法写出不考虑并发的步骤，并标记出每个需要并发访问保护的资源，要注意修改顺序（非并发情况下有些修改顺序可以调转，但是并发的时候因为多线程导致的逻辑依赖就不能调转了）
2. 重写每个方法，对需要并发保护的资源的修改需要保证前提条件（基于相等或者不等进行判断）
3. 需要考虑每个方法可能被异常终止或者因为CPU调度导致暂停执行，因此整个数据结构的每个方法在修改关键变量前都需要判断是否其他线程的某个方法执行到一半，并需要协助完成后续未完成的步骤（这些修改步骤CAS可以允许失败）

# 例子：无锁并发队列

为了方便，linked list使用带有空头的方式，减少条件判断

## 初始化

伪代码

|  |
| --- |
| private transient volatile Node<T> \_tail; private transient volatile Node<T> \_head;  public CASQueue(){  Node<T> emptyHead = new Node<>(null);  \_head = \_tail =emptyHead; } |

## 入队

推导过程

|  |
| --- |
| public void Enqueue1(T value){  Node<T> newNode = new Node<>(value);  \_tail.next = newNode;//write!  \_tail = newNode;//write! }  public void Enqueue2(T value){  Node<T> newNode = new Node<>(value);  Node<T> p;  do {//\_tail.next = newNode;  p = \_tail;  }while (CAS(p.next, null, newNode));  //\_tail = newNode;  CAS(\_tail,p,newNode);  //如果线程暂停执行或者异常终止而没有执行这一行，  // 产生的不一致（\_tail不是指向最后一个节点）  // 会导致其他Enqueue线程进入死循环 } |

伪代码

|  |
| --- |
| public void Enqueue(T value){  Node<T> newNode = new Node<>(value);  Node<T> t;  // 入队分两步走，首先修改\_tail.next指向新增节点，然后修改\_tail指向新增节点  // 必须保证修改\_tail.next，再修改\_tail(可以失败，这种情况表示其他线程可以继续修改)  while (true){  t = \_tail;  Node<T> next = t.next;  if (t != \_tail){  // 尾指针被修改了，其他线程已经完整的执行了两步入队操作  // 重启入队流程  continue;  }  if (next != null){  // 尾指针的next非空，表示有线程正在入队并进行了一半的修改  // (修改\_tail.next，但未来得及再修改\_tail)  // 使用CAS跟进其他线程对\_tail.next的修改  // 即主动帮助修改\_tail.next的线程完成后半部分对\_tail的修改  // 这个修改允许失败，防止还有第三个以上的线程同时在操作  // 因此这个修改的条件是\_tail跟t相等  CAS(\_tail,t, next);  continue;  }  if (CAS(t.next,next,newNode)){  // 在这个地方next是空  // t.next也是空，修改就会成功  // 修改失败需要重新入队流程  break;  }  }  // 这个修改允许失败，防止还有第三个以上的线程同时在操作  // 因此这个修改的条件是\_tail跟t相等:  CAS(\_tail,t, newNode); } |

## 出队

伪代码

|  |
| --- |
| public T Dequeue(){  T val = null;  // 出队分两步走，首先修改\_head指向\_head.next（队列非空的情况下），再读取\_head前一个节点包含的值  // 虽然修改操作只有一个，但是需要考虑入队操作导致\_tail可能落后于\_head违反了队列的性质  while (true){  Node<T> h = \_head;  Node<T> t = \_tail;  Node<T> hNext = h.next;  if (h != \_head){  // \_head被修改了, 重启出队流程  continue;  }  if (h == t){  if (hNext == null) {  // 空队列，返回空值  break;  }else  { // (h == t && hNext != null)  // “半空”队列：入队线程执行了一半另\_tail.next非空，  // 但是\_head和\_tail还指向同一个节点  // 也就是入队线程执行前队列是空的，正在加入的节点将另队列包含1个节点  // 协助入队线程完成第二个修改\_tail的操作（允许失败）  // 然后重启出队流程  CAS(\_tail,t,hNext);  continue;  }  }  if (CAS(\_head, h, hNext)){   val = hNext.value;  h.next = null;// 没有这一步会导致内存泄漏  break;  }  }  return val; } |

## 查看队头

伪代码

|  |
| --- |
| public T peek(){  T val = null;  while (true){  Node<T> h = \_head;  Node<T> t = \_tail;  Node<T> hNext = h.next;  if (h != \_head){  // \_head被修改了, 重启出队流程  continue;  }  if (h == t && hNext == null){  // 空队列，返回空值  break;  }  if (h == t){  // “半空”队列：入队线程执行了一半另\_tail.next非空，  // 但是\_head和\_tail还指向同一个节点  // 也就是入队线程执行前队列是空的，正在加入的节点将另队列包含1个节点  // 协助入队线程完成第二个修改\_tail的操作（允许失败）  // 然后重启出队流程   CAS(\_tail,t,hNext);  continue;  }  val = hNext.value;  break;  }  return val; } |

Java实现

|  |
| --- |
| https://github.com/DragonLi/exercise.git/Java |

## 与jdk1.8的ConcurrentLinkedQueue对比

入队操作是offer，出队操作是poll，查看队头操作是peek

自行实现更简单：循环中条件基本没有嵌套的，空队头减少了大量判空操作。

自行实现没有支持size和迭代器两个重要的功能，对于生产消费者模式，一般不需要迭代器

|  |
| --- |
| public boolean offer(E e) {  *checkNotNull*(e);  final Node<E> newNode = new Node<E>(e);   for (Node<E> t = tail, p = t;;) {  Node<E> q = p.next;  if (q == null) {  // p is last node  if (p.casNext(null, newNode)) {  // Successful CAS is the linearization point  // for e to become an element of this queue,  // and for newNode to become "live".  if (p != t) // hop two nodes at a time  casTail(t, newNode); // Failure is OK.  return true;  }  // Lost CAS race to another thread; re-read next  }  else if (p == q)  // We have fallen off list. If tail is unchanged, it  // will also be off-list, in which case we need to  // jump to head, from which all live nodes are always  // reachable. Else the new tail is a better bet.  p = (t != (t = tail)) ? t : head;  else  // Check for tail updates after two hops.  p = (p != t && t != (t = tail)) ? t : q;  } }  public E poll() {  restartFromHead:  for (;;) {  for (Node<E> h = head, p = h, q;;) {  E item = p.item;   if (item != null && p.casItem(item, null)) {  // Successful CAS is the linearization point  // for item to be removed from this queue.  if (p != h) // hop two nodes at a time  updateHead(h, ((q = p.next) != null) ? q : p);  return item;  }  else if ((q = p.next) == null) {  updateHead(h, p);  return null;  }  else if (p == q)  continue restartFromHead;  else  p = q;  }  } }  public E peek() {  restartFromHead:  for (;;) {  for (Node<E> h = head, p = h, q;;) {  E item = p.item;  if (item != null || (q = p.next) == null) {  updateHead(h, p);  return item;  }  else if (p == q)  continue restartFromHead;  else  p = q;  }  } } |

## 参考文献

John D. Valois 1994年10月在拉斯维加斯的并行和分布系统系统国际大会上的一篇论文——《Implementing Lock-Free Queues》

美国纽约罗切斯特大学 Maged M. Michael 和 Michael L. Scott 在1996年3月发表的一篇论文 《Simple, Fast, and Practical Non-Blocking and Blocking ConcurrentQueue Algorithms》

## 思考题

如果生产者和消费者每一个时刻都最多只有一个，那可以怎样实现这个队列？（游戏客户端就存在这种需求：生产者是网络socket接收的数据，消费者是UI渲染线程）