非阻塞算法

原文地址 作者：Jakob Jenkov 译者：张坤

在并发上下文中，非阻塞算法是一种允许线程在阻塞其他线程的情况下访问共享状态的算法。在绝大多数项目中，在算法中如果一个线程的挂起没有导致其它的线程挂起，我们就说这个算法是非阻塞的。

为了更好的理解阻塞算法和非阻塞算法之间的区别，我会先讲解阻塞算法然后再讲解非阻塞算法。

阻塞并发算法

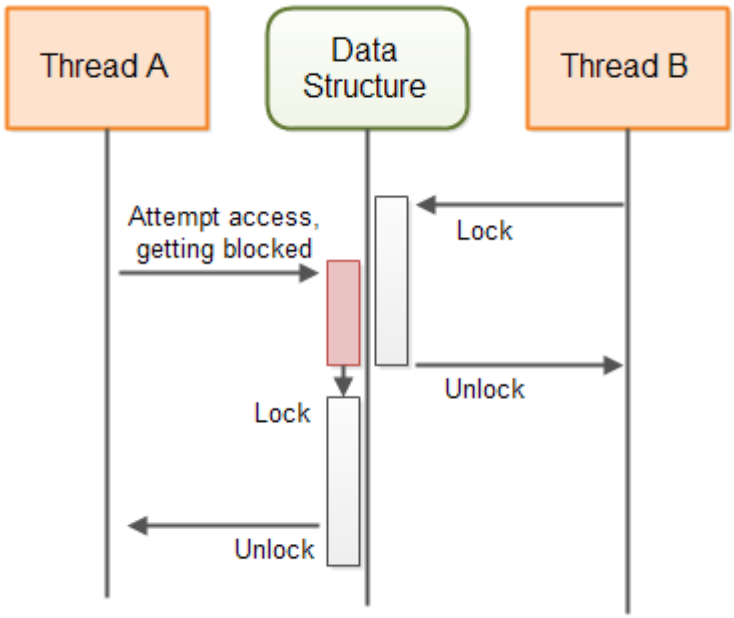
一个阻塞并发算法一般分下面两步：

执行线程请求的操作

阻塞线程直到可以安全地执行操作

很多算法和并发数据结构都是阻塞的。例如，`java.util.concurrent.BlockingQueue`的不同实现都是阻塞数据结构。如果一个线程要往一个阻塞队列中插入一个元素，队列中没有足够的空间，执行插入操作的线程就会阻塞直到队列中有了可以存放插入元素的空间。

下图演示了一个阻塞算法保证一个共享数据结构的行为：



非阻塞并发算法

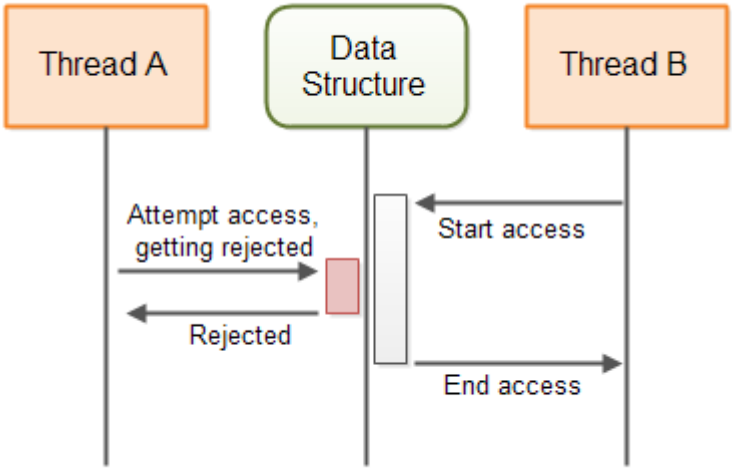
一个非阻塞并发算法一般包含下面两步：

执行线程请求的操作

通知请求线程操作不能被执行

Java也包含几个非阻塞数据结构。AtomicBoolean,AtomicInteger,AtomicLong,AtomicReference都是非阻塞数据结构的例子。

下图演示了一个非阻塞算法保证一个共享数据结构的行为：



非阻塞算法 vs 阻塞算法

阻塞算法和非阻塞算法的主要不同在于上面两部分描述的它们的行为的第二步。换句话说，它们之间的不同在于当请求操作不能够执行时阻塞算法和非阻塞算法会怎么做。

阻塞算法会阻塞线程知道请求操作可以被执行。非阻塞算法会通知请求线程操作不能够被执行，并返回。

一个使用了阻塞算法的线程可能会阻塞直到有可能去处理请求。通常，其它线程的动作使第一个线程执行请求的动作成为了可能。 如果，由于某些原因线程被阻塞在程序某处，因此不能让第一个线程的请求动作被执行，第一个线程会阻塞——可能一直阻塞或者直到其他线程执行完必要的动作。

例如，如果一个线程产生往一个已经满了的阻塞队列里插入一个元素，这个线程就会阻塞，直到其他线程从这个阻塞队列中取走了一些元素。如果由于某些原因，从阻塞队列中取元素的线程假定被阻塞在了程序的某处，那么，尝试往阻塞队列中添加新元素的线程就会阻塞，要么一直阻塞下去，要么知道从阻塞队列中取元素的线程最终从阻塞队列中取走了一个元素。

非阻塞并发数据结构

在一个多线程系统中，线程间通常通过一些数据结构”交流“。例如可以是任何的数据结构，从变量到更高级的俄数据结构（队列，栈等）。为了确保正确，并发线程在访问这些数据结构的时候，这些数据结构必须由一些并发算法来保证。这些并发算法让这些数据结构成为**并发数据结构**。

如果某个算法确保一个并发数据结构是阻塞的，它就被称为是一个**阻塞算法**。这个数据结构也被称为是一个**阻塞，并发数据结构**。

如果某个算法确保一个并发数据结构是非阻塞的，它就被称为是一个**非阻塞算法**。这个数据结构也被称为是一个**非阻塞，并发数据结构**。

每个并发数据结构被设计用来支持一个特定的通信方法。使用哪种并发数据结构取决于你的通信需要。在接下里的部分，我会引入一些非阻塞并发数据结构，并讲解它们各自的适用场景。通过这些并发数据结构工作原理的讲解应该能在非阻塞数据结构的设计和实现上一些启发。

Volatile 变量

Java中的volatile变量是直接从主存中读取值的变量。当一个新的值赋给一个volatile变量时，这个值总是会被立即写回到主存中去。这样就确保了，一个volatile变量最新的值总是对跑在其他CPU上的线程可见。其他线程每次会从主存中读取变量的值，而不是比如线程所运行CPU的CPU缓存中。

colatile变量是非阻塞的。修改一个volatile变量的值是一耳光原子操作。它不能够被中断。不过，在一个volatile变量上的一个 read-update-write 顺序的操作不是原子的。因此，下面的代码如果由多个线程执行可能导致**竞态条件**。

```
volatile myVar = 0;
...
int temp = myVar;
temp++;
myVar = temp;
```

首先，myVar这个volatile变量的值被从主存中读出来赋给了temp变量。然后，temp变量自增1。然后，temp变量的值又赋给了myVar这个volatile变量这意味着它会被写回到主存中。

如果两个线程执行这段代码，然后它们都读取myVar的值，加1后，把它的值写回到主存。这样就存在myVar仅被加1，而没有被加2的风险。

你可能认为你不会写像上面这样的代码，但是在实践中上面的代码等同于如下的代码：

```
myVar++;
```

执行上面的代码时，myVar的值读到一个CPU寄存器或者一个本地CPU缓存中，myVar加1，然后这个CPU寄存器或者CPU缓存中的值被写回到主存中。

单个写线程的情景

在一些场景下，你仅有一个线程在向一个共享变量写，多个线程在读这个变量。当仅有一个线程在更新一个变量，不管有多少个线程在读这个变量，都不会发生竞态条件。因此，无论时候当仅有一个线程在写一个共享变量时，你可以把这个变量声明为volatile。

当多个线程在一个共享变量上执行一个 read-update-write 的顺序操作时才会发生竞态条件。如果你只有一个线程在执行一个 raed-update-write 的顺序操作，其他线程都在执行读操作，将不会发生竞态条件。

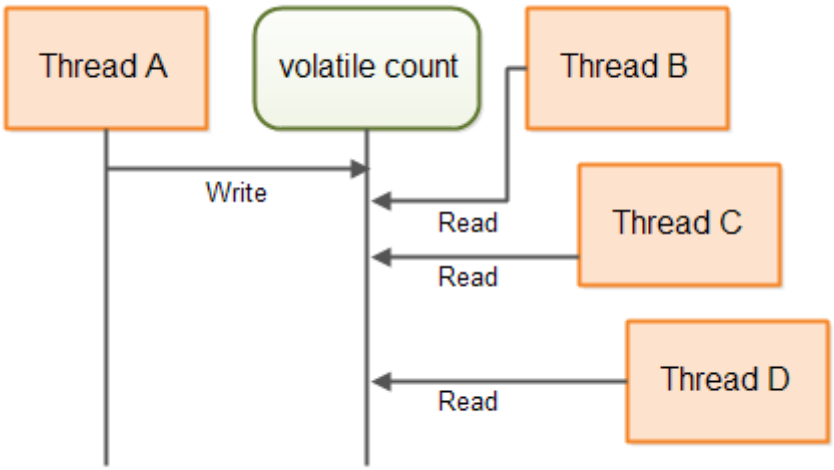
下面是一个单个写线程的例子，它没有采取同步手段但任然是并发的。

```
public class SingleWriterCounter{
    private volatile long count = 0;

    /**
     *Only one thread may ever call this method
     *or it will lead to race conditions
     */
     public void inc(){
         this.count++;
     }

     /**
      *Many reading threads may call this method
      *@return
      */
      public long count(){
```

```
        return this.count;
    }
}
```

多个线程访问同一个Counter实例，只要仅有一个线程调用inc()方法，这里，我不是说在某一时刻一个线程，我的意思是，仅有相同的，单个的线程被允许去调用inc()>方法。多个线程可以调用count()方法。这样的场景将不会发生任何竞态条件。

下图，说明了线程是如何访问count这个volatile变量的。



基于volatile变量更高级的数据结构

使用多个volatile变量去创建数据结构是可以的，构建出的数据结构中每一个volatile变量仅被一个单个的线程写，被多个线程读。每个volatile变量可能被一个不同的线程写（但仅有一个）。使用像这样的数据结构多个线程可以使用这些volatile变量以一个非阻塞的方法彼此发送信息。

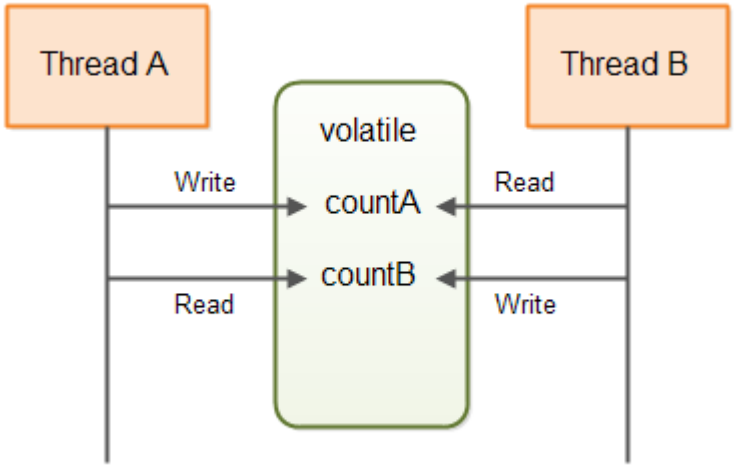
下面是一个简单的例子：

```
public class DoubleWriterCounter{
    private volatile long countA = 0;
    private volatile long countB = 0;

    /**
     *Only one (and the same from thereon) thread may ever call this method,
     *or it will lead to race conditions.
     */
```

```
    public void incA(){
        this.countA++;
    }

    /**
     *Only one (and the same from thereon) thread may ever call this method,
     *or it will  lead to race conditions.
     */
     public void incB(){
         this.countB++;
     }

     /**
      *Many reading threads may call this method
      */
     public long countA(){
         return this.countA;
     }

    /**
     *Many reading threads may call this method
     */
     public long countB(){
         return this.countB;
     }
}
```

如你所见，DoubleWriterCoounter现在包含两个volatile变量以及两对自增和读方法。在某一时刻，仅有一个单个的线程可以调用inc()，仅有一个单个的线程可以访问incB()。不过不同的线程可以同时调用incA()和incB()。countA()和countB()可以被多个线程调用。这将不会引发竞态条件。

DoubleWriterCoounter可以被用来比如线程间通信。countA和countB可以分别用来存储生产的任务数和消费的任务数。下图，展示了两个线程通过类似于上面的一个数据结构进行通信的。



聪明的读者应该已经意识到使用两个SingleWriterCounter可以达到使用DoubleWriterCoounter的效果。如果需要，你甚至可以使用多个线程和SingleWriterCounter实例。

使用CAS的乐观锁

如果你确实需要多个线程区写同一个共享变量，volatile变量是不合适的。你将会需要一些类型的排它锁

```
public class SynchronizedCounter{
    long count = 0;

    public void inc(){
        synchronized(this){
            count++;
        }
    }

    public long count(){
        synchronized(this){
            return this.count;
        }
    }
}
```

注意，，inc()和count()方法都包含一个同步块。这也是我们像避免的东西——同步块和 wait()-notify 调用等。

我们可以使用一种Java的原子变量来代替这两个同步块。在这个例子是AtomicLong。下面是SynchronizedCounter类的AtomicLong实现版本。

```
import java.util.concurrent.atomic.AtomicLong;

public class AtomicLong{
    private AtomicLong count = new AtomicLong(0);

    public void inc(){
        boolean updated = false;
        while(!updated){
            long prevCount = this.count.get();
            updated = this.count.compareAndSet(prevCount, prevCount + 1);
        }
    }

    public long count(){
        return this.count.get();
    }
}
```

```
这个版本仅仅是上一个版本的线程安全版本。这一版我们感兴趣的是inc()方法的实现。inc()方法中不再
```

```
boolean updated = false;
while(!updated){
    long prevCount = this.count.get();
    updated = this.count.compareAndSet(prevCount, prevCount + 1);
}
```

上面这些代码并不是一个原子操作。也就是说，对于两个不同的线程去调用inc()方法，然后执行long prevCount = this.count.get()语句，因此获得了这个计数器的上一个count。但是，上面的代码并没有包含任何的竞态条件。

秘密就在于while循环里的第二行代码。compareAndSet()方法调用是一个原子操作。它用一个期望值和AtomicLong 内部的值去比较，如果这两个值相等，就把AtomicLong内部值替换为一个新值。

compareAndSet()通常被CPU中的compare-and-swap指令直接支持。因此，不需要去同步，也不需要去挂起线程。

假设，这个AtomicLong的内部值是20,。然后，两个线程去读这个值，都尝试调用compareAndSet(20, 20 + 1)。尽管compareAndSet()是一个原子操作，这个方法也会被这两个线程相继执行（某一个时刻只有一个）。

第一个线程会使用期望值20（这个计数器的上一个值）与AtomicLong的内部值进行比较。由于两个值是相等的，AtomicLong会更新它的内部值至21（20 + 1 ）。变量updated被修改为true，while循环结束。

现在，第二个线程调用compareAndSet(20, 20 + 1)。由于AtomicLong的内部值不再是20，这个调用将不会成功。AtomicLong的值不会再被修改为21。变量，updated被修改为false，线程将会再次在while循环外自旋。这段时间，它会读到值21并企图把值更新为22。如果在此期间没有其它线程调用inc()。第二次迭代将会成功更新AtomicLong的内部值到22。

为什么称它为乐观锁

上一部分展现的代码被称为**乐观锁**（optimistic locking）。乐观锁区别于传统的锁，有时也被称为**悲观锁**。传统的锁会使用同步块或其他类型的锁阻塞对临界区域的访问。一个同步块或锁可能会导致线程挂起。

乐观锁允许所有的线程在不发生阻塞的情况下创建一份共享内存的拷贝。这些线程接下来可能会对它们的拷贝进行修改，并企图把它们修改后的版本写回到共享内存中。如果没有其它线程对共享内存做任何修改，CAS操作就允许线程将它的变化写回到共享内存中去。如果，另一个线程已经修改了共享内存，这个线程将不得不再次获得一个新的拷贝，在新的拷贝上做出修改，并尝试再次把它们写回到共享内存中去。

称之为“乐观锁”的原因就是，线程获得它们想修改的数据的拷贝并做出修改，在乐观的假在此期间没有线程对共享内存做出修改的情况下。当这个乐观假设成立时，这个线程仅仅在无锁的情况下完成共享内存的更新。当这个假设不成立时，线程所做的工作就会被丢弃，但任然不使用锁。

乐观锁使用于共享内存竞用不是非常高的情况。如果共享内存上的内容非常多，仅仅因为更新共享内存失败，就用浪费大量的CPU周期用在拷贝和修改上。但是，如果砸共享内存上有大量的内容，无论如何，你都要把你的代码设计的产生的争用更低。

乐观锁是非阻塞的

我们这里提到的乐观锁机制是非阻塞的。如果一个线程获得了一份共享内存的拷贝，当尝试修改时，发生了阻塞，其它线程去访问这块内存区域不会发生阻塞。

对于一个传统的加锁/解锁模式，当一个线程持有一个锁时，其它所有的线程都会一直阻塞直到持有锁的线程再次释放掉这个锁。如果持有锁的这个线程被阻塞在某处，这个锁将很长一段时间不能被释放，甚至可能一直不能被释放。

不可替换的数据结构

简单的CAS乐观锁可以用于共享数据结果，这样一来，整个数据结构都可以通过一个单个的CAS操作被替换成为一个新的数据结构。尽管，使用一个修改后的拷贝来替换真个数据结构并不总是可行的。

假设，这个共享数据结构是队列。每当线程尝试从向队列中插入或从队列中取出元素时，都必须拷贝这个队列然后在拷贝上做出期望的修改。我们可以通过使用一个AtomicReference来达到同样的目的。拷贝引用，拷贝和修改队列，尝试替换在AtomicReference中的引用让它指向新创建的队列。

然而，一个大的数据结构可能会需要大量的内存和CPU周期来复制。这会使你的程序占用大量的内存和浪费大量的时间再拷贝操作上。这将会降低你的程序的性能，特别是这个数据结构的竞用非常高情况下。更进一步说，一个线程花费在拷贝和修改这个数据结构上的时间越长，其它线程在此期间修改这个数据结构的可能性就越大。如你所知，如果另一个线程修改了这个数据结构在它被拷贝后，其它所有的线程都不等不再次执行 拷贝-修改 操作。这将会增大性能影响和内存浪费，甚至更多。

接下来的部分将会讲解一种实现非阻塞数据结构的方法，这种数据结构可以被并发修改，而不仅仅是拷贝和修改。

共享预期的修改

用来替换拷贝和修改整个数据结构，一个线程可以共享它们对共享数据结构预期的修改。一个线程向对修改某个数据结构的过程变成了下面这样：

检查是否另一个线程已经提交了对这个数据结构提交了修改
如果没有其他线程提交了一个预期的修改，创建一个预期的修改，然后向这个数据结构提交预期的修改
执行对共享数据结构的修改
移除对这个预期的修改的引用，向其它线程发送信号，告诉它们这个预期的修改已经被执行

如你所见，第二步可以阻塞其他线程提交一个预期的修改。因此，第二步实际的工作是作为这个数据结构的一个锁。如果一个线程已经成功提交了一个预期的修改，其他线程就不可以再提交一个预期的修改直到第一

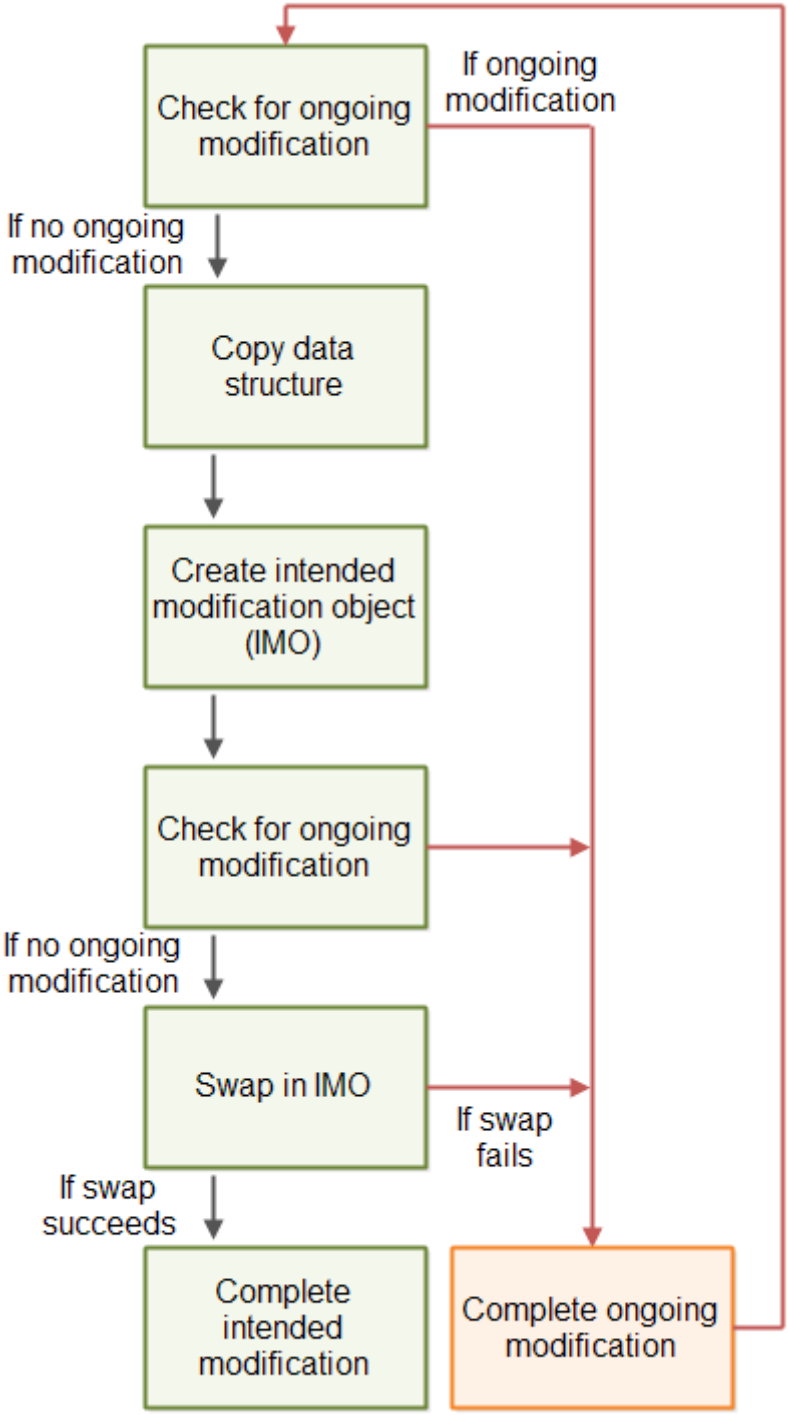
个预期的修改执行完毕。

如果一个线程提交了一个预期的修改，然后做一些其它的工作时发生阻塞，这时候，这个共享数据结构实际上是被锁住的。其它线程可以检测到它们不能够提交一个预期的修改，然后回去做一些其它的事情。很明显，我们需要解决这个问题。

可完成的预期修改

为了避免一个已经提交的预期修改可以锁住共享数据结构，一个已经提交的预期修改必须包含足够的信息让其他线程来完成这次修改。因此，如果一个提交了预期修改的线程从未完成这次修改，其他线程可以在它的支持下完成这次修改，保证这个共享数据结构对其他线程可用。

下图说明了上面描述的非阻塞算法的蓝图：



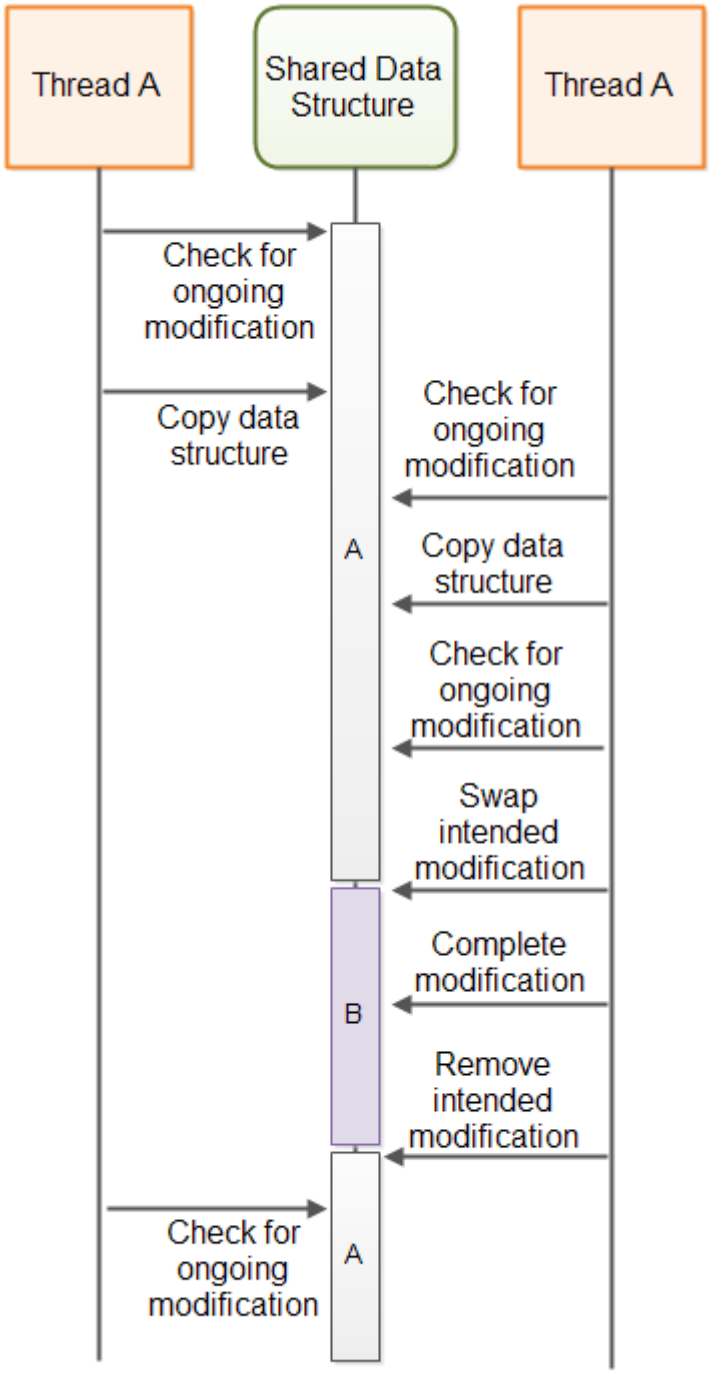
修改必须被当做一个或多个CAS操作来执行。因此，如果两个线程尝试去完成同一个预期修改，仅有一个线程可以所有的CAS操作。一旦一条CAS操作完成后，再次企图完成这个CAS操作都不会“得逞”。

A-B-A问题

上面演示的算法可以称之为**A-B-A问题**。A-B-A问题指的是一个变量被从A修改到了B，然后又被修改回A的一种情景。其他线程对于这种情况却一无所知。

如果线程A检查正在进行的数据更新，拷贝，被线程调度器挂起，一个线程B在此期可能可以访问这个共享数据结构。如果线程对这个数据结构执行了全部的更新，移除了它的预期修改，这样看起来，好像线程A自从拷贝了这个数据结构以来没有对它做任何的修改。然而，一个修改确实已经发生了。当线程A继续基于现在已经过期的数据拷贝执行它的更新时，这个数据修改已经被线程B的修改破坏。

下图说明了上面提到的A-B-A问题：



A-B-A问题的解决方案

A-B-A通常的解决方法就是不再仅仅替换指向一个预期修改对象的指针，而是指针结合一个计数器，然后使用一个单个的CAS操作来替换指针 + 计数器。这在支持指针的语言像C和C++中是可行的。因此，尽管当前修改指针被设置回指向 “不是正在进行的修改”（no ongoing modification），指针 + 计数器的计数器部分将会被自增，使修改对其它线程是可见的。

在Java中，你不能将一个引用和一个计数器归并在一起形成一个单个的变量。不过Java提供了AtomicStampedReference类，利用这个类可以使用一个CAS操作自动的替换一个引用和一个标记（stamp）。

一个非阻塞算法模板

下面的代码意在在如何实现非阻塞算法上一些启发。这个模板基于这篇教程所讲的东西。

注意：在非阻塞算法方面，我并不是一位专家，所以，下面的模板可能错误。不要基于我提供的模板实现自己的非阻塞算法。这个模板意在给你一个关于非阻塞算法大致是什么样子的一个idea。如果，你想实现自己的非阻塞算法，首先学习一些实际的工业水平的非阻塞算法的时间，在实践中学习更多关于非阻塞算法实现的知识。

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicStampedReference;

public class NonblockingTemplate{
    public static class IntendedModification{
        public AtomicBoolean completed = new AtomicBoolean(false);
    }

    private AtomicStampedReference<IntendedModification> ongoinMod = new AtomicStampedRefer
    //declare the state of the data structure here.

    public void modify(){
        while(!attemptModifyASR());
    }


    public boolean attemptModifyASR(){
        boolean modified = false;
```

```
        IntendedMOdification currentlyOngoingMod = ongoingMod.getReference();
        int stamp = ongoingMod.getStamp();

        if(currentlyOngoingMod == null){
            //copy data structure - for use
            //in intended modification

            //prepare intended modification
            IntendedModification newMod = new IntendModification();

            boolean modSubmitted = ongoingMod.compareAndSet(null, newMod, stamp, stamp + 1)

            if(modSubmitted){
                 //complete modification via a series of compare-and-swap operations.
                //note: other threads may assist in completing the compare-and-swap
                // operations, so some CAS may fail
                modified = true;
            }
        }else{
             //attempt to complete ongoing modification, so the data structure is freed up
            //to allow access from this thread.
            modified = false;
        }

        return modified;
    }
}
```

非阻塞算法是不容易实现的

正确的设计和实现非阻塞算法是不容易的。在尝试设计你的非阻塞算法之前，看一看是否已经有人设计了一种非阻塞算法正满足你的需求。

Java已经提供了一些非阻塞实现（比如 ConcurrentLinkedQueue），相信在Java未来的版本中会带来更多的非阻塞算法的实现。

除了Java内置非阻塞数据结构还有很多开源的非阻塞数据结构可以使用。例如，LAMX Disrupter和Cliff Click实现的非阻塞 HashMap。查看我的Java concurrency references page查看更多的资源。

使用非阻塞算法的好处

非阻塞算法和阻塞算法相比有几个好处。下面让我们分别看一下：

选择

非阻塞算法的第一个好处是，给了线程一个选择当它们请求的动作不能够被执行时做些什么。不再是被阻塞在那，请求线程关于做什么有了一个选择。有时候，一个线程什么也不能做。在这种情况下，它可以选择阻塞或自我等待，像这样把CPU的使用权让给其它的任务。不过至少给了请求线程一个选择的机会。

在一个单个的CPU系统可能会挂起一个不能执行请求动作的线程，这样可以让其它线程获得CPU的使用权。不过即使在一个单个的CPU系统阻塞可能导致死锁，线程饥饿等并发问题。

没有死锁

非阻塞算法的第二个好处是，一个线程的挂起不能导致其它线程挂起。这也意味着不会发生死锁。两个线程不能互相彼此等待来获得被对方持有的锁。因为线程不会阻塞当它们不能执行它们的请求动作时，它们不能阻塞互相等待。非阻塞算法任然可能产生活锁（live lock），两个线程一直请求一些动作，但一直被告知不能够被执行（因为其他线程的动作）。

没有线程挂起

挂起和恢复一个线程的代价是昂贵的。没错，随着时间的推移，操作系统和线程库已经越来越高效，线程挂起和恢复的成本也不断降低。不过，线程的挂起和户对任然需要付出很高的代价。

无论什么时候，一个线程阻塞，就会被挂起。因此，引起了线程挂起和恢复过载。由于使用非阻塞算法线程不会被挂起，这种过载就不会发生。这就意味着CPU有可能花更多时间在执行实际的业务逻辑上而不是上下文切换。

在一个多个CPU的系统上，阻塞算法会对阻塞算法产生重要的影响。运行在CPUA上的一个线程阻塞等待运行在CPU B上的一个线程。这就降低了程序天生就具备的并行水平。当然，CPU A可以调度其他线程去运行，但是挂起和激活线程（上下文切换）的代价是昂贵的。需要挂起的线程越少越好。

降低线程延迟

在这里我们提到的延迟指的是一个请求产生到线程实际的执行它之间的时间。因为在非阻塞算法中线程不会被挂起，它们就不需要付昂贵的，缓慢的线程激活成本。这就意味着当一个请求执行时可以得到更快的响应，减少它们的响应延迟。

非阻塞算法通常忙等待直到请求动作可以被执行来降低延迟。当然，在一个非阻塞数据数据结构有着很高的线程争用的系统中，CPU可能在它们忙等待期间停止消耗大量的CPU周期。这一点需要牢牢记住。非阻塞算法可能不是最好的选择如果你的数据结构哦有着很高的线程争用。不过，也常常存在通过重构你的程序来达到更低的线程争用。

Yuu
2016/05/03 2:53下午

所谓非阻塞就是把线程阻塞在while循环里么

耗子
2016/10/20 4:11下午

错字很多哦

zhijian
2017/03/27 6:12下午

文章开头的：“在并发上下文中，非阻塞算法是一种允许线程在阻塞其他线程的情况下访问共享状态的算法”。 这句翻译有误，应该是”允许线程在不阻塞其他线程…”,完全改变作者意思了。原文“Non-blocking algorithms in the context of concurrency are algorithms that allows threads to access shared state (or otherwise collaborate or communicate) without blocking the threads involved.”

yxrswx
2018/03/07 6:00下午

阻塞算法会阻塞线程知道请求操作可以被执行。 —> 阻塞算法会阻塞线程直到请求操作可以被执行。