**Non-blocking Algorithms**

* [Blocking Concurrency Algorithms](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#blocking-concurrency-algorithms)
* [Non-blocking Concurrency Algorithms](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#non-blocking-concurrency-algorithms)
* [Non-blocking vs Blocking Algorithms](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#nonblocking-vs-blocking-algorithms)
* [Non-blocking Concurrent Data Structures](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#non-blocking-concurrent-data-structures)
* [Volatile Variables](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#volatile-variables)
  + [The Single Writer Case](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#the-single-writer-case)
  + [More Advanced Data Structures Based on Volatile Variables](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#more-advanced-data-structures-based-on-volatile-variables)
* [Optimistic Locking With Compare and Swap](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#optimistic-locking-with-compare-and-swap)
  + [Why is it Called Optimistic Locking?](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#why-is-it-called-optimistic-locking)
  + [Optimistic Locking is Non-blocking](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#optimistic-locking-is-non-blocking)
* [Non-swappable Data Structures](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#non-swappable-data-structures)
  + [Completable Intended Modifications](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#completable-intended-modifications)
* [The A-B-A Problem](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#the-a-b-a-problem)
  + [A-B-A Solutions](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#a-b-a-solutions)
* [A Non-blocking Algorithm Template](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#a-non-blocking-algorithm-template)
* [Non-blocking Algorithms are Difficult to Implement](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#non-blocking-algorithms-are-difficult-to-implement)
* [The Benefit of Non-blocking Algorithms](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#the-benefit-of-nonblocking-algorithms)
  + [Choice](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#choice)
  + [No Deadlocks](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#no-deadlocks)
  + [No Thread Suspension](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#no-thread-suspension)
  + [Reduced Thread Latency](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html#reduced-thread-latency)

并发上下文中的非阻塞算法是允许线程访问共享状态（或以其他方式协作或通信）而不阻塞所涉及的线程的算法。更通用的说，一个算法被说是非阻塞的，当一个线程的挂起不能导致算法中涉及的其他线程的挂起。

为了更好理解阻塞和非阻塞并发算法之间的不同点，我将会通过解释并发算法开始，然后以非阻塞算法继续。

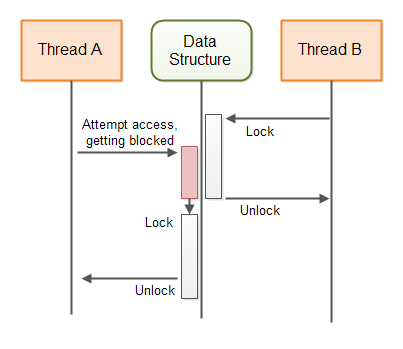
**Blocking Concurrency Algorithms**

阻塞并发算法是一种算法具有以下特点之一：

* A:运行被线程请求的行为 - 或
* B:阻塞线程直到行为能够被安全运行

许多种算法和并发数据结构是阻塞的。举例来说，[**java.util.concurrent.BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html)接口的不同实现都是阻塞的数据结构。如果一个线程尝试插入一个元素到一个BlockingQueue中，而队列没有空余，插入线程被阻塞(挂起)直到BlockingQueue有空间放新元素。

这个图表说明了一个阻塞算法守卫一个共享数据结构的行为：



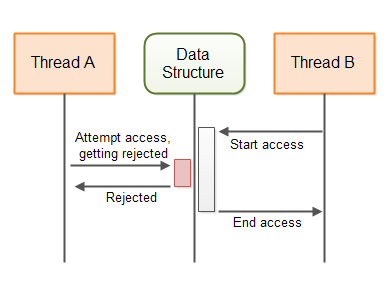
**Non-blocking Concurrency Algorithms**

非阻塞并发算法是一种算法有以下特点之一：

* A:运行线程请求的行为 - 或
* B:通知请求线程行为不能被运行

Java也包含若干个非阻塞数据结构。[**AtomicBoolean**](http://tutorials.jenkov.com/java-util-concurrent/atomicboolean.html)、[**AtomicInteger**](http://tutorials.jenkov.com/java-util-concurrent/atomicinteger.html)、[**AtomicLong**](http://tutorials.jenkov.com/java-util-concurrent/atomiclong.html)和[**AtomicReference**](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html)全部是非阻塞数据结构的例子。

这个图表说明了一个非阻塞算法守卫一个共享数据结构的行为：



**Non-blocking vs Blocking Algorithms**

阻塞和非阻塞算法之间的主要区别在于它们行为的第二步，正如在上面两节描述的。换言之，不同点在于当被请求的行为不能被运行时，阻塞和非阻塞算法做什么：

阻塞算法阻塞线程直到请求行为可以被运行。非阻塞算法提醒请求该行为的线程行为不能被运行。

通过阻塞算法，线程可能被阻塞，直到有可能执行所请求的动作为止。通常，将会是另一个线程的动作使得第一个线程能够执行所请求的动作。如果为了某个原因其他的线程被挂起(阻塞)在应用的其他某处，因此不能执行使得第一个线程请求的行为成为可能的行为，第一个线程保持阻塞 - 或者无限期，或者直到其他的线程最终执行必要的行为。

举例来说，如果一个线程试着插入一个元素到一个满BlockingQueue中，线程将阻塞直到另一个线程从BlockingQueue中取走了一个元素。如果出于某个原因预计从BlockingQueue中取走元素的线程被阻塞(挂起)在应用中其他某处，尝试插入新的元素的线程保持阻塞 - 或者无限期，或者直到取走元素的线程最终从BlockingQueue中取走一个元素。

**Non-blocking Concurrent Data Structures**

在一个多线程系统中，线程通常借由某种数据结构进行交流。这样的数据结构可能是任何事物，从简单变量到更高级的数据结构如队列、映射、栈等。为了便于多个线程对数据结构的正确并发访问，必须使用某些*并发算法(concurrent algorithm)*来保护数据结构。保护算法是使数据结构成为*并发数据结构(concurrent data structure)*的原因。

如果保护并发数据结构的算法是阻塞的(使用线程挂起)，它被称作*阻塞算法*。数据结构因此被称为一个*阻塞、并发数据结构*。

如果保护一个并发数据结构的算法是非阻塞的，它被称作*非阻塞算法*。数据结构因此被称作一个*非阻塞、并发数据结构*。

每个并发数据结构被设计支持一种特定的交流方法。你可以使用哪种并发数据结构因此依赖于你的交流需要。我将会讲到一些非阻塞并发数据结构在接下来的章节，并解释什么情况下它们可以被使用。这些非并发数据结构如何工作的解释应当告诉您如何设计和实现非阻塞数据结构。

**Volatile Variables**

[**Java volatile variables**](http://tutorials.jenkov.com/java-concurrency/volatile.html)是总是直接从主内存中读取的变量。当一个新值被分配给一个volatile变量，值总是被立刻写到主内存中。这保证了一个volatile变量的最新的值总是对其他运行在其他CPU上线程可见。其他线程将每次都从主内存中读取volatile的值，而不是从如线程运行的CPU的CPU缓存中。

volatile变量是非阻塞的。一个值向一个volatile变量的写入是一个原子操作。它不能被打断。然而，一个读-更新-写序列运行在一个volatile变量上不是原子的。因此，这段代码仍旧可能导向竞争条件([**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html))如果被超过一个线程运行：

volatile myVar = 0;

...

int temp = myVar;

temp++;

myVar = temp;

首先volatile变量的值myVar被从主内存中读取到临时变量中。然后临时变量增加1。然后临时变量的值被分配给volatile的myVar变量，意思是它将被写回到主内存中。

如果两个线程执行这段代码，他们两个都读myVar的值，加1给它，并把值写回到主内存，那么你可能有风险不是2被加到myVar变量，只有1将会被加(例如两个线程读值19，加到20，然后把20写回)。

你可能认为你不会像上面那样写代码，但是实际上上面的代码等价于这个：

myVar++;

当执行时，myVar的值被读取到一个CPU寄存器或者本地CPU缓存，1被加上，然后来自CPU寄存器或者CPU缓存的值被写回到主内存中。

**The Single Writer Case**

在一些情况下，你只有一个单独的线程写入一个共享变量，多个线程读取那个变量的值。没有竞争条件可以发生当知有一个单独的线程更新一个变量，无论多少个线程正在读取它。因此，无论何时一个共享变量你只有一个单独的写，你可以使用一个volatile变量。

竞争条件在多个线程在一个共享变量上运行一个读-更新-写操作序列时发生。如果你只有一个线程运行一个读-更新-写操作序列，全部其他的线程只运行读操作，你没有竞争条件。

这是一个单一写计数器，不使用同步但仍旧是并发的：

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() {

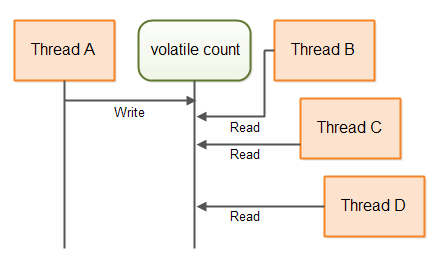
return this.count;

}

}

多个线程可以访问这个计数器的同一个实例，只要只有一个线程调用inc()。我的意思不是一次一个线程。我意思是，只有同一个、单独线程被允许调用inc()。多个线程能够调用count()。这将不会导致任何竞争条件。

这个表格说明了线程将如何访问volatile count变量：



**More Advanced Data Structures Based on Volatile Variables**

创建使用volatile变量的组合的数据结构是很可能的，其中每个volatile变量只被一个单独线程写，被多个线程读。每个volatile变量可能被一个不同的线程写(但只有一个线程)。使用这样的一个数据结构，多个线程可能能够互相发送信息以一种非阻塞的方式，使用volatile变量。

这是一个简单的双写计数器类，展示那看起来将如何：

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

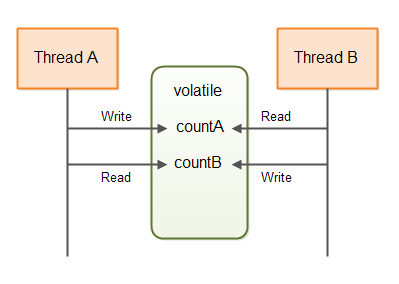
\*/

public long countB() { return this.countB; }

}

正如你所见，DoubleWriterCounter现在包含两个volatile变量，和两对增加以及读方法。只有一个单独线程可能调用incA()，只有一个线程可以调用incB()。不过调用incA()和incB()的线程可以不同。许多线程被允许调用countA()和countB()。这将不会导致竞争条件。

DoubleWriterCounter能被用于如两个线程交流的场景。这两个计数器可以是生产任务和消费任务。这个图表展示两个线程交流通过一个数据结构类似于上述：



机智的读者将会认识到你可能已经获取了DoubleWriterCounter的效果，通过使用两个SingleWriterCounter实例。你甚至可能已经用过更多线程和SingleWriterCounter实例，如果你需要这样。

**Optimistic Locking With Compare and Swap**

如果你真的需要超过一个线程写到同样的共享变量，一个volatile变量将不满足。你将会需要某种额外的对变量的访问。这是这样的使用一个同步块([**synchronized block in Java**](http://tutorials.jenkov.com/java-concurrency/synchronized.html))额外访问应当看起来的样子:

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。这是我们想要避免的 - 同步块和wait() - notify()调用等。

取代两个同步块，我们可以使用Java原子变量之一。这种情况下使用AtomicLong。这是相同的计数器类使用一个AtomicLong来代替看起来的样子：

import java.util.concurrent.atomic.AtomicLong;

public class AtomicCounter {

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();

}

}

这个版本就像之前的版本一样线程安全。这一版令人感兴趣的是inc()方法的实现。inc()方法不再包含一个同步块。取而代之它包含这些行：

boolean updated = false;

while(!updated){

long prevCount = this.count.get();

updated = this.count.compareAndSet(prevCount, prevCount + 1);

}

这些行不是原子操作。那意味着，两个不同的线程调用inc()方法执行long prevCount = this.count.get()语句是可能的，因此两个都获取计数器先前的计数。然而，上面的代码不包含竞争条件。

秘密在于while循环内部的两行的第二行。compareAndSet()方法是一个原子操作。它比较这个AtomicLong的内部值与一个期望值，如果两个值相同，给这个AtomicLong设置一个新的内部值。compareAndSet()方法通常通过在CPU中直接的比较和交换指令来支持。因此，不需要同步，并且不需要线程挂起。这节省了线程挂起开销。

想像这个AtomicLong的内部值是20。然后两个线程读取那个值，都尝试调用compareAndSet(20, 20 + 1)。因为compareAndSet()是一个原子操作，线程将会序列地(一次一个地)执行这个方法。

第一个线程将会比较期望值20(计数器先前的值)与这个AtomicLong的内部值。因为两个值相同，这个AtomicLong将会更新内部值为21(20+1)。updated变量将设置为true，while循环将停止。

现在第二个线程调用compareAndSet(20, 20 + 1)。因为AtomicLong的内部值不再是20，调用将会失败。AtomicLong的内部值将不会被设置为21。updated变量将被设置为false，线程将在while循环中再自旋一次。这次它将读值21，尝试更新为22。如果没有其他线程已经同时调用inc()，第二次循环将成功更新AtomicLong为22。

**Why is it Called Optimistic Locking?**

展示在先前章节的代码被称作*乐观锁*(*optimistic locking*)。乐观锁定与传统锁，有时也被叫做悲观锁，不同。传统锁定阻塞对带有共享块或者某种锁的共享内存的访问。同步块或者锁可能导致线程挂起。

乐观锁定允许全部线程创建共享内存的副本而不阻塞。然后线程可能对它们的副本做出修改，尝试写它们的修改版本回到共享内存中。如果没有其他的线程已经对共享内存做出任何修改，比较和替换操作允许线程写改变到共享内存中。如果另一个线程已经改变了共享内存，线程将不得不获取一个新的副本，使它改变并尝试再次写他们到共享内存中。

这被称作乐观锁的原因是线程获取他们想要改变的数据备份并应用它们的修改，基于没有其他线程将已经在同时对共享内存做出改变的乐观假定。当这个乐观假定保持为真，线程仅仅成功更新共享内存而不锁定。当这个假定为假，工作被废弃，但仍旧没有锁定被应用。

乐观锁倾向于往往在共享内存的低到中等竞争中发挥作用。如果在共享内存上的竞争非常高，线程将浪费大量CPU循环复制和修改共享内存只得到失败地将改变写回共享内存。但是，如果你在共享内存上有大量竞争，你无论如何应当考虑重新设计代码以降低竞争。

**Optimistic Locking is Non-blocking**

乐观锁机制我已经展示在这里的是不阻塞的。如果一个线程获取共享内存的一份副本，当尝试着修改它时被阻塞(处于无论什么原因)，没有其他的线程被阻塞于访问共享内存。

在传统的加锁/解锁范例中，当一个线程锁定一个锁时，该锁对于所有其他线程保持锁定，直到拥有该锁的线程再次解锁它。如果锁定锁的线程在别处被阻塞，那么该锁将保持锁定很长时间 - 甚至可能无限期。

**Non-swappable Data Structures**

简单的比较和交换乐观锁定适用于共享数据结构，其中整个数据结构可以在单个比较和交换操作中与新的数据结构交换。然而，用修改过的副本交换整个数据结构并不总是可行的。

想象如果共享数据结构是一个队列。每个线程尝试或者插入或者拿取元素从队列中，将不得不复制整个队列并对副本做出期望的修改。这可能被获取借由一个AtomicReference。复制这个引用，复制和修改这个队列，尝试替换在AtomicReference指向的引用为新创建的队列。

然而，一个大数据结构可能需要大量内存和CPU循环来复制。这将使你的应用耗费多得多的内存，浪费大量时间在复制上。这将影响你的应用的性能，尤其如果在数据结构上的竞争很高。除此之外，线程复制和修改这个数据结构花费时间越长，某个其他线程将在其中间修改数据结构的可能性越大。正如你所知，如果另一个线程已经修改共享数据结构因为它被复制，全部其他的线程不得不重启它们的复制-修改操作。这将增加性能和内存消费上的影响更多。

下一节将解释一个方法来实现非阻塞数据结构，可以被并发更新，不仅仅复制和修改。

**Sharing Intended Modifications**

取代复制和修改整个共享数据结构，一个线程可以共享它对共享数据结构的*预期修改*。线程想要对共享数据结构作出修改的过程则变为：

1. 检查是否另一个线程已经提交一个预期修改到这个数据结构
2. 如果没有其他的线程已经提交一个预期修改，创建一个预期修改(由对象表示)并提交那个预期修改到数据结构(使用比较和替换操作)
3. 执行共享数据结构的修改
4. 移除预期修改的引用来发信号给其他线程预期修改已经被移除

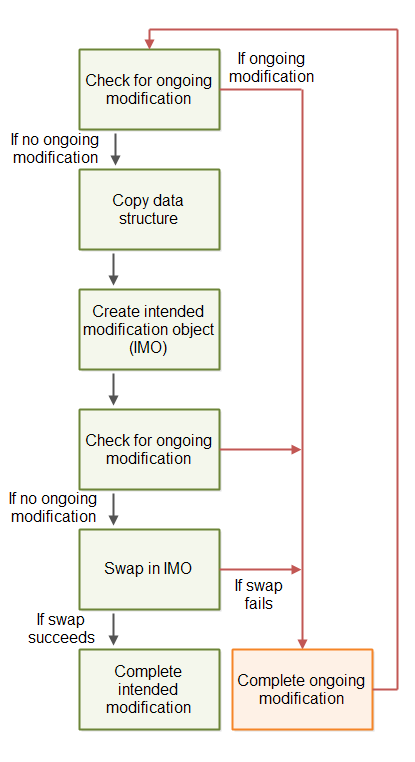
正如你所见，第二步可以阻塞其他线程提交一个预期修改。因此，第二步有效地用作共享数据结构的锁。如果一个线程成功地提交一个预期修改，没有其他的线程可以提交一个预期修改直到第一个预期修改被执行。

如果一个线程提交一个预期修改然后被阻塞在做某项其他工作，共享数据结构被有效地锁定了。共享数据结构不直接阻塞其他的线程使用数据结构。其他的线程可以检测到它们不能提交一个预期修改并决定做其他的事。显然，我们需要修正那个。

**Completable Intended Modifications**

为了避免提交的预期修改能够锁定共享数据结构，提交的预期修改对象必须包含足够的信息对于另一个线程来完成修改。因此，如果提交预期修改的线程不再完成修改，另一个线程能够完成它自己的修改，保持共享数据结构可供其他线程使用。

这是一个图表说明上述非阻塞算法的蓝图：



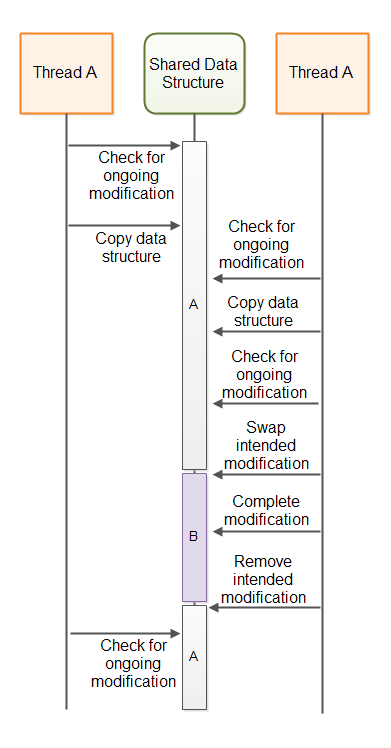
修改必须被执行作为一个或更多比较和替换操作。因此，如果两个线程试着完成预期修改，只有一个线程将能够执行任何比较和替换操作。只要一个比较和替换操作已经被完成，更多的完成比较和替换操作的尝试将会失败。

**The A-B-A Problem**

上述的说明算法可能遭受A-B-A问题。A-B-A问题指的是一个变量从A被改变为B然后再次变回A的情况。因此对另一个线程，检测变量确实被改变是不大可能的。

如果线程A检查进行中的更新，复制数据并被线程调度器挂起，线程B运行数据结构的一个完整更新，并移除它的预期修改，它将看向线程A好像没有修改发生过，自从它复制数据结构开始。然而，修改确实发生了。当线程A继续基于它现在过时的数据结构副本执行更新时，数据结构将取消线程B的修改。

接下来的图表说明了来自上述情况的A-B-A问题：



**A-B-A Solutions**

A-B-A问题的一种通用的解决方法是不仅仅替换一个指针为一个预期修改对象，并且组合指针和一个计数器，并且使用一个单独的比较和替换操作替换指针+计数器。在支持指针的语言像C和C++中这是可能的。因此，即使当前的修改指针被设置回指向“没有进行中的修改”，指针+计数器的计数器部分将被增加，使得更新对其他线程可见。

在Java中你不能合并一个引用和一个计数器一起到一个单独的变量。取而代之，Java提供[**AtomicStampedReference**](http://tutorials.jenkov.com/java-util-concurrent/atomicstampedreference.html)类，该类能够使用一个比较和替换操作原子替换一个引用和一个标志戳。

**A Non-blocking Algorithm Template**

下面是一个代码模板，旨在让您了解如何实现非阻塞算法。模板基于本教程前面给出的描述。

注意：我不是阻塞算法的专家，因此下面的模板可能有一些错误。不要基于你自己的非阻塞算法在我的模板上实现。模板只是试图给你一个想法关于一个非阻塞算法的代码长什么样子。如果你想要实现你自己的非阻塞算法，首先研究一些真正的有效的非阻塞算法实现，来了解更多关于它们实际中怎么实现。

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>

ongoingMod =

new AtomicStampedReference<IntendedModification>(null, 0);

//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 state - for use

//in intended modification

//prepare intended modification

IntendedModification newMod =

new IntendedModification();

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;

}

}

**Non-blocking Algorithms are Difficult to Implement**

非阻塞算法很难正确设计和实现。在试图实现你自己的非阻塞算法之前，看看是否某人已经开发了符合你需求的非阻塞算法。

Java已经带来了一些非阻塞实现(例如ConcurrentLinkedQueue)，并将很可能得到更多非阻塞算法实现在未来的Java版本中。

除了Java内置的非阻塞数据结构之外，也有一些开源非阻塞数据结构你可以使用。举个例子，LMAX干扰器(一个拟队列数据结构)，和非阻塞HashMap来自Cliff Click。参见我的[**Java concurrency references page**](http://tutorials.jenkov.com/java-concurrency/references.html)链接获得更多资源。

**The Benefit of Non-blocking Algorithms**

非阻塞算法和阻塞算法对比，有几个好处。这一节将会描述这些好处。

**Choice**

非阻塞算法的第一个好处是，线程被给出一个选择关于当他们的请求行为不能被运行时做什么。替代仅仅被阻塞，请求线程有个选择关于做什么。有时候没有线程可以做的事。那种情况下它能选择它自己阻塞或者等待，因此释放CPU对于其他任务。但至少请求线程被给出一个选择。

在一个单独的CPU系统中也许挂起一个不能运行想要的行为的线程，并让其他的可以运行它们的工作的线程在CPU上跑有意义。但是甚至在一个单独的CPU系统上阻塞算法可能导致问题，像[**deadlock**](http://tutorials.jenkov.com/java-concurrency/deadlock.html)、[**starvation**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)和其他并发问题。

**No Deadlocks**

非阻塞算法的第二个好处是，一个线程的挂起不可能导致其他线程的挂起。这意味着死锁不能发生。两个线程不能阻塞在互相等待来释放它们想要的锁。因为当它们不能够运行它们的请求行为时，线程不是阻塞的，他们不能够被阻塞在互相等待中。非阻塞算法可能仍旧导致活锁，即两个线程保持尝试某行为，但持续被告诉行为不可能(因为其他线程的行为)。

**No Thread Suspension**

挂起和重新激活一个线程是很大花销的。是的，挂起和重激活的花销已经降低了，随着操作系统和线程库变得更高效。然而，线程挂起和重激活仍需付出高昂的代价。

无论什么时候一个线程被阻塞它被挂起，从而导致线程挂起和重新激活的开销。因为线程不会被阻塞算法挂起，这个开销不会发生。这意味着CPU可以花更多的时间执行实际的业务逻辑而不是上下文切换。

在一个多CPU系统中，阻塞算法可以有更有意义的影响在整体运行上。一个运行在CPU A上的线程能被阻塞在等待执行在CPU B上的线程。这将降低应用有能力获取的平行化的水平。当然，CPU A仅能够调度另一个线程运行，但挂起和激活线程(上下文切换)很昂贵。需要被挂起的线程越少越好。

**Reduced Thread Latency**

本文中的等待时间意思是在一个请求行为变得可能和线程实际运行它之间的时间。因为在非阻塞算法中线程不会被挂起，它们不需要支付高昂的、缓慢的重激活花销。那意味着当一个请求行为变得可能，线程可以响应得更快，因此减少它们的响应等待时间。

非阻塞算法通常获取更低的等待时间通过忙等待，直到请求行为变得可能。当然，在一个在非阻塞数据结构上高线程竞争的系统里，CPU可能最终烧毁大量循环在这些忙等待中。这需要记在心中。非阻塞算法可能不是最好的，如果你的数据结构有高线程竞争。然而，总是有办法重新设计你的应用以获得更少的线程竞争。