第八章 原子变量与非阻塞算法

第八章 原子变量与非阻塞算法.....1
8.1. 锁的劣势.....2
8.2. 原子变量类.....2
8.3. 非阻塞算法.....5
参考文献.....8

本章首先分析锁的劣势，然后分析原子变量类和非阻塞算法的优势。本章内容与第 3 章和第 4 章内容，紧密相关。相关内容情况参考前述章节。

8.1. 锁的劣势

从前面的章节可以看到，使用一致的加锁协议来协调对共享状态的访问，确保当线程持有守护变量的锁时，线程都能独占地访问这些变量，并且保证随后获得同一锁的线程都能看见该线程对变量所作的修改。

Java 虚拟机能够对非竞争锁的获取和释放进行优化，让它们非常高效，但是如果有多个线程同时请求锁，Java 虚拟机就需要向操作系统寻求帮助。倘若了出现这种情况，一些线程将可能被挂起，并稍后恢复运行。从线程开始恢复，到它真正被调度前，可能必须等待其他线程完成它们的调度限额规定的时问。挂起和恢复线程会带来很大的开销，并通常伴有冗长的中断。对于基于锁，并且其操作过度细分的类（比如同步容器类，大多数方法只包含很少的操作），当频繁地发生锁的竞争时，调度与真正用于工作的开销间的比值会很可观。

加锁还有其他的缺点。当一个线程正在等待锁时，它不能做任何其他事情。如果一个线程在持有锁的情况下发生了延迟（原因包括页错误、调度延迟，或者类似情况），那么其他所有需要该锁的线程都不能前进了。如果阻塞的线程是优先级很高的线程，持有锁的线程优先级较低，那么会造成性能风险，被称为优先级倒置（priority inversion）。即虽然更高的优先级占先，但它仍然需要等待锁被释放，这导致它的优先级会降至与优先级较低的线程相同的水平。如果持有锁的线程发生了永久性的阻塞（因为无限循环、死锁、活锁和其他活跃度失败），所有等待该锁的线程都不会前进了。

即使忽略上述的风险，加锁对于小的操作而言，仍然是重量级（heavy weight）的机制，比如自增操作。需要有更好的技术用来管理线程之问的竞争。在 Java 5.0 中，使用原子变量类（atomic variable classes）能够高效地构建非阻塞算法。

8.2. 原子变量类

在 JDK 5.0 之前，如果不使用本机代码，就不能用 Java 语言编写无等待、

无锁定的算法。在 java.util.concurrent 中添加原子变量类之后，这种情况发生了变化。本节了解这些新类开发高度可伸缩的无阻塞算法。

java.util.concurrent.atomic 包中添加原子变量类。所有原子变量类都公开“比较并设置”原语（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。

原子变量类共有 12 个，分成 4 组：计量器、域更新器（field updater）、数组以及复合变量。最常用的原子变量是计量器：AtomicInteger、AtomicLong、AtomicBoolean 以及 AtomicReference。他们都支持 CAS（比较并设置,详细参考第 3 章）；AtomicInteger 和 AtomicLong 还支持算术运算。

原子变量类可以认为是 volatile 变量的泛化，它扩展了 volatile 变量的概念，来支持原子条件的比较并设置更新。读取和写入原子变量与读取和写入对 volatile 变量的访问具有相同的存取语义。

虽然原子变量类表面看起来与 SynchronizedCounter 例子（参考 3.2.1 节）一样，但相似仅是表面的。在表面之下，原子变量的操作会变为平台提供的用于并发访问的硬件原语，比如比较并交换。

调整具有竞争的并发应用程序的可伸缩性的通用技术是降低使用的锁对象的粒度，希望更多的锁请求从竞争变为不竞争。从锁转换为原子变量可以获得相同的结果，通过切换为更细粒度的协调机制，竞争的操作就更少，从而提高了吞吐量。

下面的程序是使用原子变量后的计数器：

```
package jdkapidemo;
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounter {
   private AtomicInteger value = new AtomicInteger();
   public int getValue() {
      return value.get();
   }
   public int increment() {
      return value.incrementAndGet();
   }
   public int increment(int i) {
```

```
        return value.addAndGet(i);
    }
    public int decrement() {
        return value.decrementAndGet();
    }
    public int decrement(int i) {
        return value.addAndGet(-i);
    }
}
```

下面写一个测试类：

```
package jdkapidemo;
public class AtomicCounterTest extends Thread {
    AtomicCounter counter;
    public AtomicCounterTest(AtomicCounter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        int i = counter.increment();
        System.out.println("generated number:" + i);
    }
    public static void main(String[] args) {
        AtomicCounter counter = new AtomicCounter();
        for (int i = 0; i < 10; i++) {//10个线程
            new AtomicCounterTest(counter).start();
        }
    }
}
```

运行结果如下：

```
generated number:1
generated number:2
generated number:3
generated number:4
generated number:5
```

```
generated number:7
generated number:6
generated number:9
generated number:10
generated number:8
```

会发现 10 个线程运行中，没有重复的数字，原子量类使用本机 CAS 实现了值修改的原子性。

8.3. 非阻塞算法

基于锁的算法会带来一些活跃度失败的风险。譬如，如果线程在持有锁的时候因为阻塞 I/O、页面错误、或其他原因发生延迟，很可能所有线程都不能前进。一个线程的失败或挂起不应该影响其他线程的失败或挂起，这样的算法被称为非阻塞（non-blocking）算法。如果算法的每一步骤中都有一些线程能够继续执行，那么这样的算法称为无锁（lock-free）算法。非阻塞算法对死锁和优先级倒置有“免疫性”。好的非阻塞算法已经应用到多种常见的数据结构上，包括栈、队列、优先级队列、哈希表。

在实现相同功能的前提下，非阻塞算法往往比基于锁的算法更加复杂。创建非阻塞算法的前提是为了维护数据的一致性，解决如何把原子化范围缩小到一个唯一变量。在链式容器类（比如队列）中，有时你可以把它变为对单独链接的修改；进而，你可以使用一个 AtomicReference 来表达每一个必须被原子更新的链接。

非阻塞算法相对于基于锁的算法有几个性能优势。首先，它用硬件的原生形态代替 Java 虚拟机的锁定代码路径，从而在更细的粒度层次上（独立的内存位置）进行同步，失败的线程也可以立即重试，而不会被挂起后重新调度。更细的粒度降低了争用的机会，不用重新调度就能重试的能力也降低了争用的成本。即使有少量失败的 CAS 操作，这种方法仍然会比由于锁竞争造成的重新调度快得多。

下面给出一个非阻塞堆栈的示例代码：

```
public class ConcurrentStack<E> {
```

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

对于上面代码中的 ConcurrentStack 中的 push()和 pop()操作，其所做的工作

有些冒险，希望在“提交”工作的时候，底层假设没有失效。push()方法观察当前的栈顶节点，并构建一个新节点放在堆栈上，然后，观察最顶端的节点在初始之后有没有变化，如果没有变化，那么就安装新节点。如果 CAS 失败，意味着另一个线程已经修改了堆栈，那么过程就会重新开始。

所有非阻塞算法的一个基本特征是：有些算法步骤的执行是要冒险的，因为假如 CAS 不成功，可能不得不重做。非阻塞算法通常叫做乐观算法，因为它们继续操作的假设是不会有干扰；假如发现干扰，就会回退并重试。

在轻度到中度的争用情况下，非阻塞算法的性能会超越阻塞算法，因为 CAS 的多数时间都在第一次尝试时就成功，而发生争用时的开销也不涉及线程挂起和上下文切换，只多了几个循环迭代。没有争用的 CAS 要比没有争用的锁便宜得多（这句话肯定是真的，因为没有争用的锁涉及 CAS 加上额外的处理），而争用的 CAS 比争用的锁获取涉及更短的延迟。在高度争用的情况下（即有多个线程不断争用一个内存位置的时候），基于锁的算法开始提供比非阻塞算法更好的吞吐率，因为当线程阻塞时，它就会停止争用，耐心地等候轮到自己，从而避免了进一步争用。但是，这么高的争用程度并不常见，因为多数时候，线程会把线程本地的计算与争用共享数据的操作分开，从而给其他线程使用共享数据的机会。同时，这么高的争用程度也表明需要重新检查算法，朝着更少共享数据的方向努力。

如果深入 Java 虚拟机和操作系统，会发现非阻塞算法无处不在。垃圾收集器使用非阻塞算法加快并发和平行的垃圾搜集；调度器使用非阻塞算法有效地调度线程和进程，实现内在锁。在 Java 6 中，基于锁的 SynchronousQueue 算法被新的非阻塞版本代替。很少有开发人员会直接使用 SynchronousQueue，但是通过 Executors.newCachedThreadPool()工厂构建的线程池用它作为工作队列。比较缓存线程池性能的对比测试显示，新的非阻塞同步队列实现提供了几乎是当前实现 3 倍的速度。在 Java 6 的后续版本中，已经规划了进一步的改进。

非阻塞算法要比基于锁的算法复杂得多。开发非阻塞算法是相当专业的训练，而且要证明算法的正确也极为困难。但是在 Java 版本之间并发性能上的众多改进来自对非阻塞算法的采用，而且随着并发性能变得越来越重要，可以预见在 Java 平台的未来发行版中，会使用更多的非阻塞算法。

参考文献

1. Brian Goetz，Java 理论与实践:流行的原子，
   http://www-128.ibm.com/developerworks/cn/java/j-jtp11234/index.html
2. java 并发集合，
   http://www.ibm.com/developerworks/cn/java/j-tiger06164/index.html