第1章　并发编程的挑战

并发编程的目的是为了让程序运行得更快，但是，并不是启动更多的线程就能让程序最大限度地并发执行。在进行并发编程时，如果希望通过多线程执行任务让程序运行得更快，会面临非常多的挑战，比如上下文切换的问题、死锁的问题，以及受限于硬件和软件的资源限制问题，本章会介绍几种并发编程的挑战以及解决方案。

1.1 上下文切换

即使是单核处理器也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的，时间片一般是几十毫秒（ms）。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

这就像我们同时读两本书，当我们在读一本英文的技术书时，发现某个单词不认识，于是便打开中英文字典，但是在放下英文技术书之前，大脑必须先记住这本书读到了多少页的第多少行，等查完单词之后，能够继续读这本书。这样的切换是会影响读书效率的，同样上下文切换也会影响多线程的执行速度。

1.1.1 多线程一定快吗

下面的代码演示串行和并发执行并累加操作的时间，请分析：下面的代码并发执行一定比串行执行快吗？

```
public class ConcurrencyTest {
        private static final long count = 10000l;
        public static void main(String[] args) throws InterruptedException {
                concurrency();
                serial();
        }
        private static void concurrency() throws InterruptedException {
                long start = System.currentTimeMillis();
                Thread thread = new Thread(new Runnable() {
                        @Override
                        public void run() {
                                int a = 0;
                                for (long i = 0; i < count; i++) {
                                        a += 5;
                                }
                        }
                });
                thread.start();
                int b = 0;
                for (long i = 0; i < count; i++) {
                        b--;
                }
                long time = System.currentTimeMillis() - start;
                thread.join();
                System.out.println("concurrency :" + time+"ms,b="+b);
        }
        private static void serial() {
                long start = System.currentTimeMillis();
                int a = 0;
                for (long i = 0; i < count; i++) {
                        a += 5;
                }
                int b = 0;
                for (long i = 0; i < count; i++) {
                        b--;
                }
                long time = System.currentTimeMillis() - start;
                System.out.println("serial:" + time+"ms,b="+b+",a="+a);
        }
}
```

上述问题的答案是“不一定”，测试结果如表1-1所示。

表1-1 测试结果

循环次数	串行执行耗时 /ms	并发执行耗时	并发比串行快多少
1 亿	130	77	约 1 倍
1 千万	18	9	约 1 倍
1 百万	5	5	差不多
10 万	4	3	慢
1 万	0	1	慢

从表1-1可以发现，当并发执行累加操作不超过百万次时，速度会比串行执行累加操作要慢。那么，为什么并发执行的速度会比串行慢呢？这是因为线程有创建和上下文切换的开销。

1.1.2 测试上下文切换次数和时长

下面我们来看看有什么工具可以度量上下文切换带来的消耗。

·使用Lmbench3[1]可以测量上下文切换的时长。

·使用vmstat可以测量上下文切换的次数。

下面是利用vmstat测量上下文切换次数的示例。

```
$ vmstat 1
 procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
  r  b   swpd   free   buff   cache   si   so    bi    bo   in   cs us sy id wa st
  0  0      0 127876 398928 2297092    0    0     0     4    2    2  0  0 99  0  0
  0  0      0 127868 398928 2297092    0    0     0     0  595 1171  0  1 99  0  0
  0  0      0 127868 398928 2297092    0    0     0     0  590 1180  1  0 100 0  0
  0  0      0 127868 398928 2297092    0    0     0     0  567 1135  0  1 99  0  0
```

CS(Content Switch)表示上下文切换的次数，从上面的测试结果中我们可以看到，上下文每1秒切换1000多次。

[1] Lmbench3是一个性能分析工具。

1.1.3 如何减少上下文切换

减少上下文切换的方法有无锁并发编程、CAS算法、使用最少线程和使用协程。

·无锁并发编程。多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以用一些办法来避免使用锁，如将数据的ID按照Hash算法取模分段，不同的线程处理不同段的数据。

·CAS算法。Java的Atomic包使用CAS算法来更新数据，而不需要加锁。

·使用最少线程。避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都处于等待状态。

·协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

1.1.4 减少上下文切换实战

本节将通过减少线上大量WAITING的线程，来减少上下文切换次数。

第一步：用jstack命令dump线程信息，看看pid为3117的进程里的线程都在做什么。

```
sudo -u admin /opt/ifeve/java/bin/jstack 31177 > /home/tengfei.fangtf/dump17
```

第二步：统计所有线程分别处于什么状态，发现300多个线程处于WAITING（onobject-monitor）状态。

```
[tengfei.fangtf@ifeve ~]$ grep java.lang.Thread.State dump17 | awk '{print $2$3$4$5}'
        | sort | uniq -c
 39 RUNNABLE
 21 TIMED_WAITING(onobjectmonitor)
 6 TIMED_WAITING(parking)
 51 TIMED_WAITING(sleeping)
 305 WAITING(onobjectmonitor)
 3 WAITING(parking)
```

第三步：打开dump文件查看处于WAITING（onobjectmonitor）的线程在做什么。发现这些线程基本全是JBOSS的工作线程，在await。说明JBOSS线程池里线程接收到的任务太少，大量线程都闲着。

```
"http-0.0.0.0-7001-97" daemon prio=10 tid=0x000000004f6a8000 nid=0x555e in
    Object.wait() [0x0000000052423000]
 java.lang.Thread.State: WAITING (on object monitor)
 at java.lang.Object.wait(Native Method)
 - waiting on <0x00000007969b2280> (a org.apache.tomcat.util.net.AprEndpoint$Worker)
 at java.lang.Object.wait(Object.java:485)
 at org.apache.tomcat.util.net.AprEndpoint$Worker.await(AprEndpoint.java:1464)
 - locked <0x00000007969b2280> (a org.apache.tomcat.util.net.AprEndpoint$Worker)
 at org.apache.tomcat.util.net.AprEndpoint$Worker.run(AprEndpoint.java:1489)
 at java.lang.Thread.run(Thread.java:662)
```

第四步：减少JBOSS的工作线程数，找到JBOSS的线程池配置信息，将maxThreads降到100。

```
<maxThreads="250" maxHttpHeaderSize="8192"
 emptySessionPath="false" minSpareThreads="40" maxSpareThreads="75"
     maxPostSize="512000" protocol="HTTP/1.1"
 enableLookups="false" redirectPort="8443" acceptCount="200" bufferSize="16384"
 connectionTimeout="15000" disableUploadTimeout="false" useBodyEncodingForURI= "true">
```

第五步:**重启**JBOSS,**再**dump**线程信息**,**然后**统计WAITING(onobjectmonitor)**的线程**,发现**减少了**175**个。**WAITING**的线**程少了**,系**统上下文切**换**的次数就会少**,因为**每一次从**WAITTING**到**RUNNABLE**都会**进行一次上下文的切**换。读**者也可以使用**vmstat**命令**测试**一下。**

```
[tengfei.fangtf@ifeve ~]$ grep java.lang.Thread.State dump17 | awk '{print $2$3$4$5}'
    | sort | uniq -c
   44 RUNNABLE
   22 TIMED_WAITING(onobjectmonitor)
   9 TIMED_WAITING(parking)
   36 TIMED_WAITING(sleeping)
   130 WAITING(onobjectmonitor)
1  WAITING(parking)
```

1.2 死锁

锁是个非常有用的工具，运用场景非常多，因为它使用起来非常简单，而且易于理解。但同时它也会带来一些困扰，那就是可能会引起死锁，一旦产生死锁，就会造成系统功能不可用。让我们先来看一段代码，这段代码会引起死锁，使线程t1和线程t2互相等待对方释放锁。

```
public class DeadLockDemo {
    privat static String A = "A";
    private static String B = "B";
    public static void main(String[] args) {
            new DeadLockDemo().deadLock();
    }
    private void deadLock() {
            Thread t1 = new Thread(new Runnable() {
                    @Override
                    publicvoid run() {
                            synchronized (A) {
                                    try { Thread.currentThread().sleep(2000);
                                    } catch (InterruptedException e) {
                                            e.printStackTrace();
                                    }
                                    synchronized (B) {
                                            System.out.println("1");
                                    }
                            }
                    }
            });
            Thread t2 = new Thread(new Runnable() {
                    @Override
                    publicvoid run() {
                            synchronized (B) {
                                    synchronized (A) {
                                            System.out.println("2");
                                    }
                            }
                    }
            });
            t1.start();
            t2.start();
    }
}
```

这段代码只是演示死锁的场景，在现实中你可能不会写出这样的代码。但是，在一些更为复杂的场景中，你可能会遇到这样的问题，比如t1拿到锁之后，因为一些异常情况没有释放锁（死循环）。又或者是t1拿到一个数据库锁，释放锁的时候抛出了异常，没释放掉。

一旦出现死锁，业务是可感知的，因为不能继续提供服务了，那么只能通过dump线程查看到底是哪个线程出现了问题，以下线程信息告诉我们是DeadLockDemo类的第42行和第31行引起的死锁。

```
"Thread-2" prio=5 tid=7fc0458d1000 nid=0x116c1c000 waiting for monitor entry [116c1b00
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.ifeve.book.forkjoin.DeadLockDemo$2.run(DeadLockDemo.java:42)
        - waiting to lock <7fb2f3ec0> (a java.lang.String)
        - locked <7fb2f3ef8> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:695)
"Thread-1" prio=5 tid=7fc0430f6800 nid=0x116b19000 waiting for monitor entry [116b1800
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.ifeve.book.forkjoin.DeadLockDemo$1.run(DeadLockDemo.java:31)
        - waiting to lock <7fb2f3ef8> (a java.lang.String)
        - locked <7fb2f3ec0> (a java.lang.String)
        at java.lang.Thread.run(Thread.java:695)
```

现在我们介绍避免死锁的几个常见方法。

·避免一个线程同时获取多个锁。

·避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。

·尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制。

·对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。

1.3 资源限制的挑战

(1)什么是资源限制

资源限制是指在进行并发编程时，程序的执行速度受限于计算机硬件资源或软件资源。例如，服务器的带宽只有2Mb/s，某个资源的下载速度是1Mb/s每秒，系统启动10个线程下载资源，下载速度不会变成10Mb/s，所以在进行并发编程时，要考虑这些资源的限制。硬件资源限制有带宽的上传/下载速度、硬盘读写速度和CPU的处理速度。软件资源限制有数据库的连接数和socket连接数等。

(2)资源限制引发的问题

在并发编程中，将代码执行速度加快的原则是将代码中串行执行的部分变成并发执行，但是如果将某段串行的代码并发执行，因为受限于资源，仍然在串行执行，这时候程序不仅不会加快执行，反而会更慢，因为增加了上下文切换和资源调度的时间。例如，之前看到一段程序使用多线程在办公网并发地下载和处理数据时，导致CPU利用率达到100%，几个小时都不能运行完成任务，后来修改成单线程，一个小时就执行完成了。

(3)如何解决资源限制的问题

对于硬件资源限制，可以考虑使用集群并行执行程序。既然单机的资源有限制，那么就让程序在多机上运行。比如使用ODPS、Hadoop或者自己搭建服务器集群，不同的机器处理不同的数据。可以通过“数据ID%机器数”，计算得到一个机器编号，然后由对应编号的机器处理这笔数据。

对于软件资源限制，可以考虑使用资源池将资源复用。比如使用连接池将数据库和Socket连接复用，或者在调用对方webservice接口获取数据时，只建立一个连接。

(4)在资源限制情况下进行并发编程

如何在资源限制的情况下，让程序执行得更快呢？方法就是，根据不同的资源限制调整程序的并发度，比如下载文件程序依赖于两个资源——带宽和硬盘读写速度。有数据库操作时，涉及数据库连接数，如果SQL语句执行非常快，而线程的数量比数据库连接数大很多，则某些线程会被阻塞，等待数据库连接。

1.4 本章小结

本章介绍了在进行并发编程时，大家可能会遇到的几个挑战，并给出了一些解决建议。有的并发程序写得不严谨，在并发下如果出现问题，定位起来会比较耗时和棘手。所以，对于Java开发工程师而言，笔者强烈建议多使用JDK并发包提供的并发容器和工具类来解决并发问题，因为这些类都已经通过了充分的测试和优化，均可解决了本章提到的几个挑战。

第2章 Java并发机制的底层实现原理

Java代码在编译后会变成Java字节码，字节码被类加载器加载到JVM里，JVM执行字节码，最终需要转化为汇编指令在CPU上执行，Java中所使用的并发机制依赖于JVM的实现和CPU的指令。本章我们将深入底层一起探索下Java并发机制的底层实现原理。

2.1 volatile的应用

在多线程并发编程中synchronized和volatile都扮演着重要的角色，volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。如果volatile变量修饰符使用恰当的话，它比synchronized的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。本文将深入分析在硬件层面上Intel处理器是如何实现volatile的，通过深入分析帮助我们正确地使用volatile变量。

我们先从了解volatile的定义开始。

1.volatile的定义与实现原理

Java语言规范第3版中对volatile的定义如下：Java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁要更加方便。如果一个字段被声明成volatile，Java线程内存模型确保所有线程看到这个变量的值是一致的。

在了解volatile实现原理之前，我们先来看下与其实现原理相关的CPU术语与说明。表2-1是CPU术语的定义。

表2-1 CPU的术语定义

术　　语	英文单词	术语描述
内存屏障	memory barriers	是一组处理器指令，用于实现对内存操作的顺序限制
缓冲行	cache line	缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期
原子操作	atomic operations	不可中断的一个或一系列操作
缓存行填充	cache line fill	当处理器识别到从内存中读取操作数是可缓存的，处理器读取整个缓存行到适当的缓存（L1，L2，L3 的或所有）
缓存命中	cache hit	如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存读取
写命中	write hit	当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中
写缺失	write misses the cache	一个有效的缓存行被写入到不存在的内存区域

volatile是如何来保证可见性的呢？让我们在X86处理器下通过工具获取JIT编译器生成的汇编指令来查看对volatile进行写操作时，CPU会做什么事情。

Java代码如下。

```
instance = new Singleton();                    // instance是volatile变量
```

转变成汇编代码，如下。

```
0x01a3de1d: movb $0×0,0×1104800(%esi);0x01a3de24: lock addl $0×0,(%esp);
```

有volatile变量修饰的共享变量进行写操作的时候会多出第二行汇编代码，通过查IA-32架构软件开发者手册可知，Lock前缀的指令在多核处理器下会引发了两件事情[1]。

1）将当前处理器缓存行的数据写回到系统内存。

2）这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部

缓存(L1，L2或其他)后再进行操作，但操作完不知道何时会写到内存。如果对声明了volatile的变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。所以，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。

下面来具体讲解volatile的两条实现原则。

1)Lock前缀指令会引起处理器缓存回写到内存。Lock前缀指令导致在执行指令期间，声言处理器的LOCK#信号。在多处理器环境中，LOCK#信号确保在声言该信号期间，处理器可以独占任何共享内存[2]。但是，在最近的处理器里，LOCK#信号一般不锁总线，而是锁缓存，毕竟锁总线开销的比较大。在8.1.4节有详细说明锁定操作对处理器缓存的影响，对于Intel486和Pentium处理器，在锁操作时，总是在总线上声言LOCK#信号。但在P6和目前的处理器中，如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反，它会锁定这块内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性，此操作被称为“缓存锁定”，缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据。

2)一个处理器的缓存回写到内存会导致其他处理器的缓存无效。IA-32处理器和Intel 64处理器使用MESI(修改、独占、共享、无效)控制协议去维护内部缓存和其他处理器缓存的一致性。在多核处理器系统中进行操作的时候，IA-32和Intel 64处理器能嗅探其他处理器访问系统内存和它们的内部缓存。处理器使用嗅探技术保证它的内部缓存、系统内存和其他处理器的缓存的数据在总线上保持一致。例如，在Pentium和P6 family处理器中，如果通过嗅探一个处理器来检测其他处理器打算写内存地址，而这个地址当前处于共享状态，那么正在嗅探的处理器将使它的缓存行无效，在下次访问相同内存地址时，强制执行缓存行填充。

2.volatile的使用优化

著名的Java并发编程大师Doug lea在JDK 7的并发包里新增一个队列集合类LinkedTransferQueue，它在使用volatile变量时，用一种追加字节的方式来优化队列出队和入队的性能。LinkedTransferQueue的代码如下。

```
/** 队列中的头部节点 */
private transient f?inal PaddedAtomicReference<QNode> head;
/** 队列中的尾部节点 */
private transient f?inal PaddedAtomicReference<QNode> tail;
static f?inal class PaddedAtomicReference <T> extends AtomicReference T> {
      // 使用很多4个字节的引用追加到64个字节
      Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;
      PaddedAtomicReference(T r) {
         super(r);
      }
}
public class AtomicReference <V> implements java.io.Serializable {
      private volatile V value;
      // 省略其他代码
}
```

追加字节能优化性能？这种方式看起来很神奇，但如果深入理解处理器架构就能理解其中的奥秘。让我们先来看看LinkedTransferQueue这个类，它使用一个内部类类型来定义队列的头节点(head)和尾节点(tail)，而这个内部类PaddedAtomicReference相对于父类AtomicReference只做了一件事情，就是将共享变量追加到64字节。我们可以来计算下，一个对象的引用占4个字节，它追加了15个变量(共占60个字节)，再加上父类的value变量，一共64个字节。

为什么追加64字节能够提高并发编程的效率呢？因为对于英特尔酷睿i7、酷睿、Atom和NetBurst，以及Core Solo和Pentium M处理器的L1、L2或L3缓存的高速缓存行是64个字节宽，不支持部分填充缓存行，这意味着，如果队列的头节点和尾节点都不足64字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头、尾节点，当一个处理器试图修改头节点时，会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作则需要不停修改头

节点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。Doug lea使用追加到64字节的方式来填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行，使头、尾节点在修改时不会互相锁定。

那么是不是在使用volatile变量时都应该追加到64字节呢？不是的。在两种场景下不应该使用这种方式。

·缓存行非64字节宽的处理器。如P6系列和奔腾处理器，它们的L1和L2高速缓存行是32个字节宽。

·共享变量不会被频繁地写。因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定的性能消耗，如果共享变量不被频繁写的话，锁的几率也非常小，就没必要通过追加字节的方式来避免相互锁定。

不过这种追加字节的方式在Java 7下可能不生效，因为Java 7变得更加智慧，它会淘汰或重新排列无用字段，需要使用其他追加字节的方式。除了volatile，Java并发编程中应用较多的是synchronized，下面一起来看一下。

[1] 这两件事情在IA-32软件开发者架构手册的第三册的多处理器管理章节（第8章）中有详细阐述。

[2] 因为它会锁住总线，导致其他CPU不能访问总线，不能访问总线就意味着不能访问系统内存。

2.2 synchronized的实现原理与应用

在多线程并发编程中synchronized一直是元老级角色，很多人都会称呼它为重量级锁。但是，随着Java SE 1.6对synchronized进行了各种优化之后，有些情况下它就并不那么重了。本文详细介绍Java SE 1.6中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁，以及锁的存储结构和升级过程。

先来看下利用synchronized实现同步的基础：Java中的每一个对象都可以作为锁。具体表现为以下3种形式。

·对于普通同步方法，锁是当前实例对象。

·对于静态同步方法，锁是当前类的Class对象。

·对于同步方法块，锁是Synchonized括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。那么锁到底存在哪里呢？锁里面会存储什么信息呢？

从JVM规范中可以看到Synchonized在JVM里的实现原理，JVM基于进入和退出Monitor对象来实现方法同步和代码块同步，但两者的实现细节不一样。代码块同步是使用monitorenter和monitorexit指令实现的，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明。但是，方法的同步同样可以使用这两个指令来实现。

monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。任何对象都有一个monitor与之关联，当且一个monitor被持有后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁。

2.2.1 Java对象头

synchronized用的锁是存在Java对象头里的。如果对象是数组类型，则虚拟机用3个字宽（Word）存储对象头，如果对象是非数组类型，则用2字宽存储对象头。在32位虚拟机中，1字宽等于4字节，即32bit，如表2-2所示。

表2-2 Java对象头的长度

长　　度	内　　容	说　　明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/32bit	Array length	数组的长度（如果当前对象是数组）

Java对象头里的Mark Word里默认存储对象的HashCode、分代年龄和锁标记位。32位JVM的Mark Word的默认存储结构如表2-3所示。

表2-3 Java对象头的存储结构

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

在运行期间，Mark Word里存储的数据会随着锁标志位的变化而变化。Mark Word可能变化为存储以下4种数据，如表2-4所示。

表2-4 Mark Word的状态变化

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

在64位虚拟机下，Mark Word是64bit大小的，其存储结构如表2-5所示。

表2-5 Mark Word的存储结构

锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit) Epoch(2bit)				1	01

2.2.2 锁的升级与对比

Java SE 1.6为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”，在Java SE 1.6中，锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率，下文会详细分析。

1.偏向锁

HotSpot[1]的作者经过研究发现，大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁。当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需简单地测试一下对象头的Mark Word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁。如果测试失败，则需要再测试一下Mark Word中偏向锁的标识是否设置成1（表示当前是偏向锁）：如果没有设置，则使用CAS竞争锁；如果设置了，则尝试使用CAS将对象头的偏向锁指向当前线程。

（1）偏向锁的撤销

偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有正在执行的字节码）。它会首先暂停拥有偏向锁的线程，然后检查持有偏向锁的线程是否活着，如果线程不处于活动状态，则将对象头设置成无锁状态；如果线程仍然活着，拥有偏向锁的栈会被执行，遍历偏向对象的锁记录，栈中的锁记录和对象头的Mark Word要么重新偏向于其他线程，要么恢复到无锁或者标记对象不适合作为偏向锁，最后唤醒暂停的线程。图2-1中的线程1演示了偏向锁初始化的流程，线程2演示了偏向锁撤销的流程。

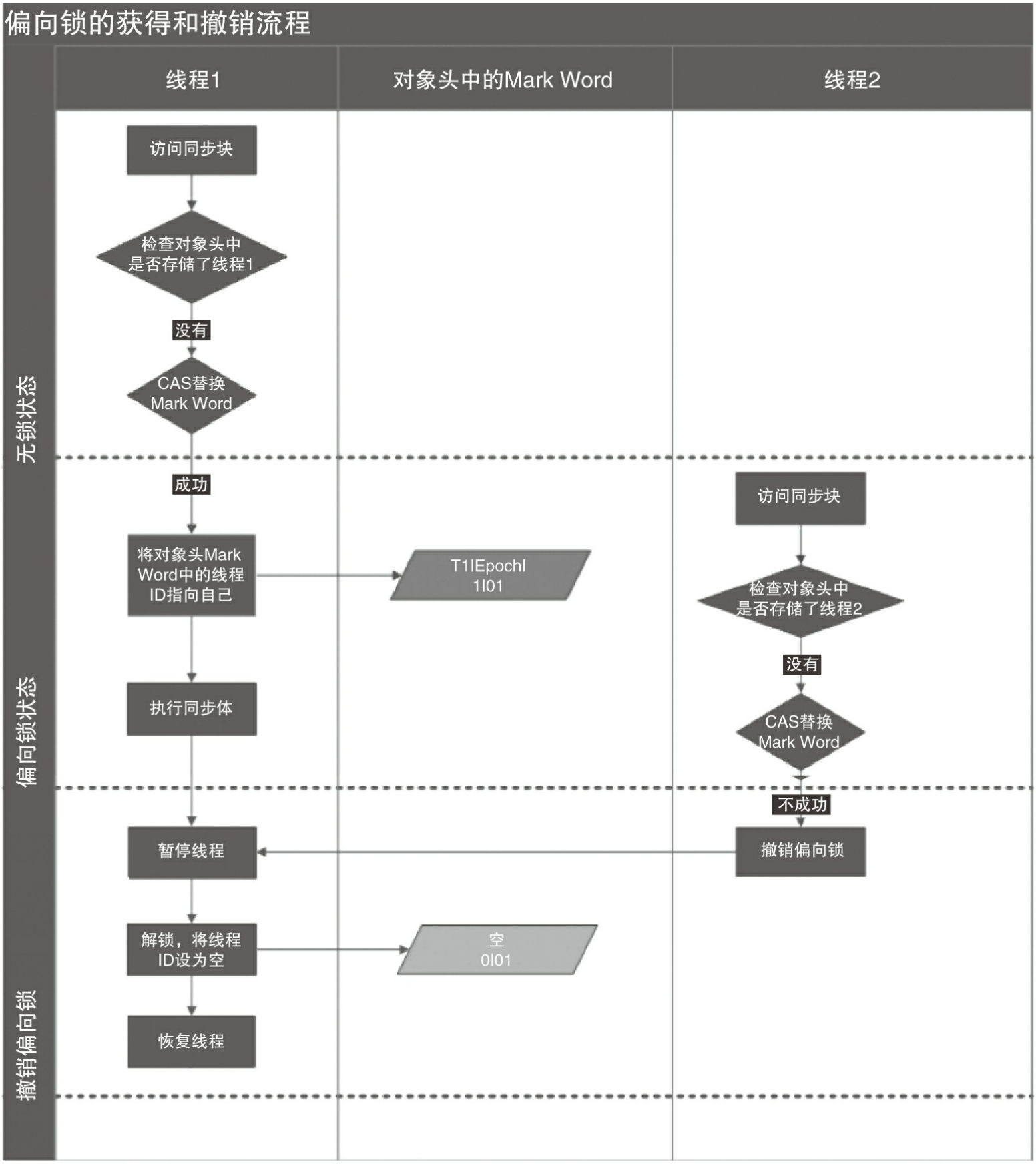

图2-1 **偏向**锁**初始化的流程**

(2)**关**闭**偏向**锁

偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如有必要可以使用JVM参数来关闭延迟：-XX:BiasedLockingStartupDelay=0。如果你确定应用程序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁：-XX:-UseBiasedLocking=false，那么程序默认会进入轻量级锁状态。

2.轻量级锁

（1）轻量级锁加锁

线程在执行同步块之前，JVM会先在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中，官方称为Displaced Mark Word。然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，当前线程获得锁，如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。

（2）轻量级锁解锁

轻量级解锁时，会使用原子的CAS操作将Displaced Mark Word替换回到对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀成重量级锁。图2-2是两个线程同时争夺锁，导致锁膨胀的流程图。

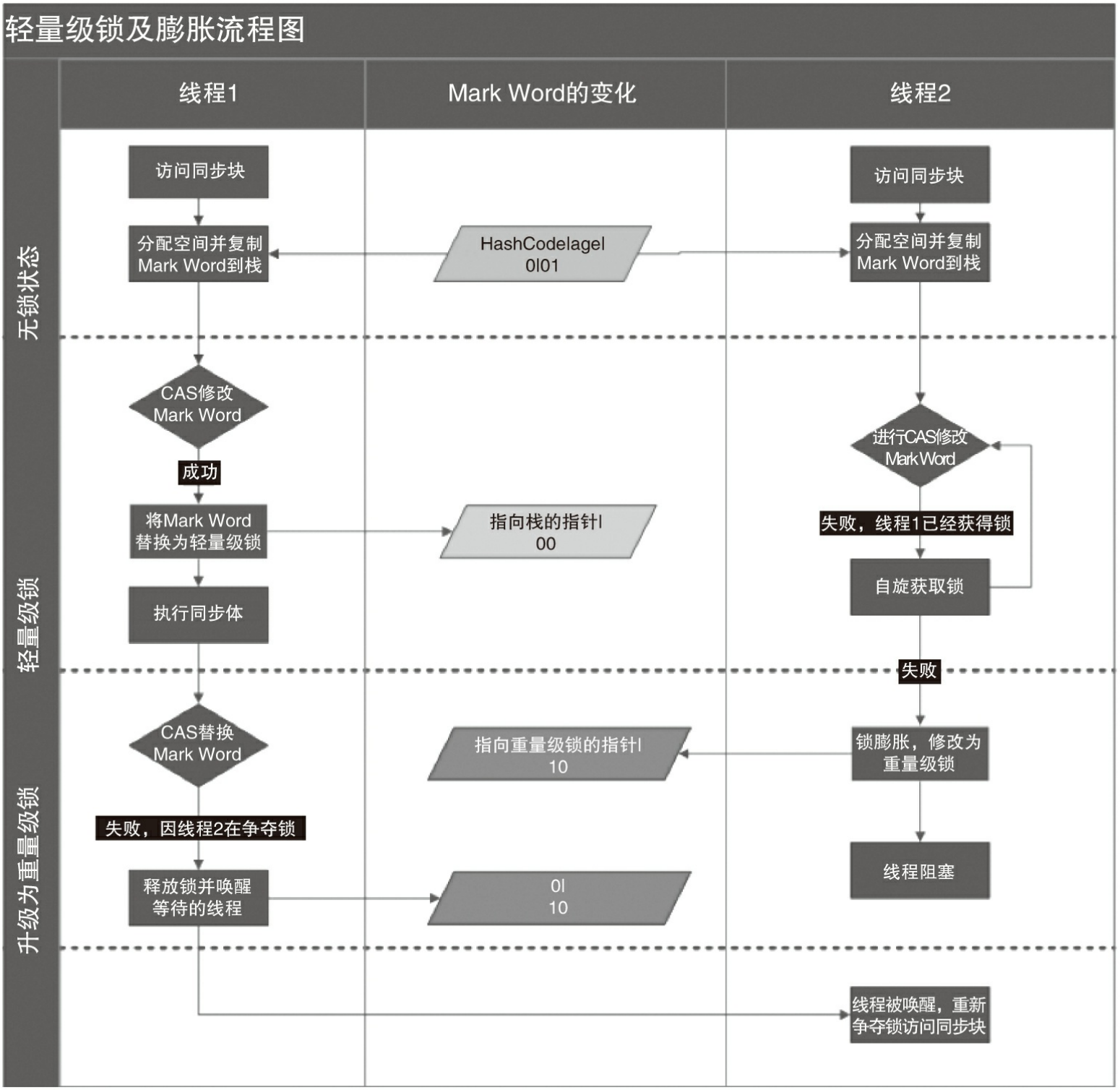

图2-2 争夺锁导致的锁膨胀流程图

因为自旋会消耗CPU，为了避免无用的自旋（比如获得锁的线程被阻塞住了），一旦锁升级成重量级锁，就不会再恢复到轻量级锁状态。当锁处于这个状态下，其他线程试图获取锁时，都会被阻塞住，当持有锁的线程释放锁之后会唤醒这些线程，被唤醒的线程就会进行新一轮的夺锁之争。

3.锁的优缺点对比

表2-6是锁的优缺点的对比。

表2-6 锁的优缺点的对比

锁	优　　点	缺　　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

[1] 本节一些内容参考了HotSpot源码、对象头源码markOop.hpp、偏向锁源码biasedLocking.cpp，以及其他源码ObjectMonitor.cpp和BasicLock.cpp。

2.3 原子操作的实现原理

原子(atomic)本意是“不能被进一步分割的最小粒子”，而原子操作(atomic operation)意为“不可被中断的一个或一系列操作”。在多处理器上实现原子操作就变得有点复杂。让我们一起来聊一聊在Intel处理器和Java里是如何实现原子操作的。

1.术语定义

在了解原子操作的实现原理前，先要了解一下相关的术语，如表2-7所示。

表2-7 CPU术语定义

术语名称	英　　文	解　　释
缓存行	Cache line	缓存的最小操作单位
比较并交换	Compare and Swap	CAS 操作需要输入两个数值，一个旧值（期望操作前的值）和一个新值，在操作期间先比较旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换
CPU 流水线	CPU pipeline	CPU 流水线的工作方式就像工业生产上的装配流水线，在 CPU 中由 5 ~ 6 个不同功能的电路单元组成一条指令处理流水线，然后将一条 X86 指令分成 5 ~ 6 步后再由这些电路单元分别执行，这样就能实现在一个 CPU 时钟周期完成一条指令，因此提高 CPU 的运算速度
内存顺序冲突	Memory order violation	内存顺序冲突一般是由假共享引起的，假共享是指多个 CPU 同时修改同一个缓存行的不同部分而引起其中一个 CPU 的操作无效，当出现这个内存顺序冲突时，CPU 必须清空流水线

2.处理器如何实现原子操作

32位IA-32处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、

跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

(1)使用总线锁保证原子性

第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写操作(i++就是经典的读改写操作)，那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致。举个例子，如果i=1，我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2，如图2-3所示。

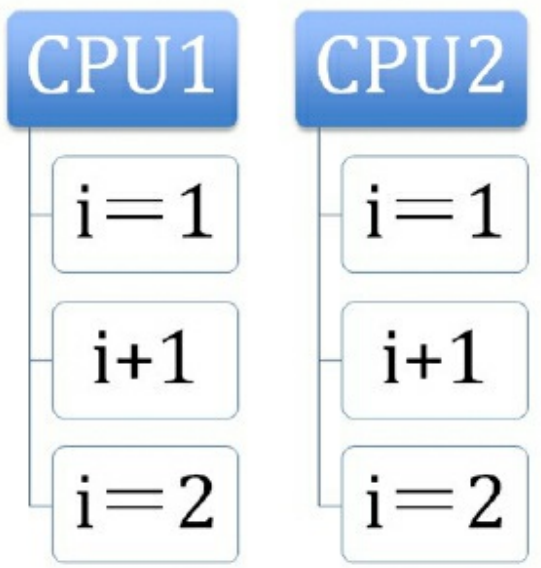

图2-3 结果对比

原因可能是多个处理器同时从各自的缓存中读取变量i，分别进行加1操作，然后分别写入系统内存中。那么，想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。所谓总线锁就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。

(2)使用缓存锁保证原子性

第二个机制是通过缓存锁定来保证原子性。在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但总线锁定把CPU和内存之间的通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1、L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在Pentium 6和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言LOCK#信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如图2-3所示的例子中，当CPU1修改缓存行中的i时使用了缓存锁定，那么CPU2就不能同时缓存i的缓存行。

但是有两种情况下处理器不会使用缓存锁定。

第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。

第二种情况是：有些处理器不支持缓存锁定。对于Intel 486和Pentium处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

针对以上两个机制，我们通过Intel处理器提供了很多Lock前缀的指令来实现。例如，位测试和修改指令：BTS、BTR、BTC；交换指令XADD、CMPXCHG，以及其他一些操作数和逻辑指令（如ADD、OR）等，被这些指令操作的内存区域就会加锁，导致其他处理器不能同时访问它。

3.Java如何实现原子操作

在Java中可以通过锁和循环CAS的方式来实现原子操作。

(1)使用循环CAS实现原子操作

JVM中的CAS操作正是利用了处理器提供的CMPXCHG指令实现的。自旋CAS实现的基本思路就是循环进行CAS操作直到成功为止，以下代码实现了一个基于CAS线程安全的计数器方法safeCount和一个非线程安全的计数器count。

```
private AtomicInteger atomicI = new AtomicInteger(0);
    private int i = 0;
    public static void main(String[] args) {
        final Counter cas = new Counter();
        List<Thread> ts = new ArrayList<Thread>(600);
        long start = System.currentTimeMillis();
        for (int j = 0; j < 100; j++) {
            Thread t = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        cas.count();
                        cas.safeCount();
                    }
                }
            });
            ts.add(t);
        }
        for (Thread t : ts) {
            t.start();
        }
    // 等待所有线程执行完成
        for (Thread t : ts) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(cas.i);
        System.out.println(cas.atomicI.get());
        System.out.println(System.currentTimeMillis() - start);
    }
    /**         * 使用CAS实现线程安全计数器         */
    private void safeCount() {
        for (;;) {
            int i = atomicI.get();
            boolean suc = atomicI.compareAndSet(i, ++i);
            if (suc) {
                break;
            }
        }
    }
    /**
     * 非线程安全计数器
```

```
     */
    private void count() {
        i++;
    }
}
```

从Java 1.5开始，JDK的并发包里提供了一些类来支持原子操作，如AtomicBoolean（用原子方式更新的boolean值）、AtomicInteger（用原子方式更新的int值）和AtomicLong（用原子方式更新的long值）。这些原子包装类还提供了有用的工具方法，比如以原子的方式将当前值自增1和自减1。

（2）CAS实现原子操作的三大问题

在Java并发包中有一些并发框架也使用了自旋CAS的方式来实现原子操作，比如LinkedTransferQueue类的Xfer方法。CAS虽然很高效地解决了原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大，以及只能保证一个共享变量的原子操作。

1）ABA问题。因为CAS需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A→B→A就会变成1A→2B→3A。从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

```
public boolean compareAndSet(
            V           expectedReference,          // 预期引用
            V           newReference,               // 更新后的引用
            int          expectedStamp,             // 预期标志
            int          newStamp                   // 更新后的标志
)
```

2）循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如

果JVM能支持处理器提供的pause指令，那么效率会有一定的提升。pause指令有两个作用：第一，它可以延迟流水线执行指令（de-pipeline），使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零；第二，它可以避免在退出循环的时候因内存顺序冲突（Memory Order Violation）而引起CPU流水线被清空（CPU Pipeline Flush），从而提高CPU的执行效率。

3）只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

（3）使用锁机制实现原子操作

锁机制保证了只有获得锁的线程才能够操作锁定的内存区域。JVM内部实现了很多种锁机制，有偏向锁、轻量级锁和互斥锁。有意思的是除了偏向锁，JVM实现锁的方式都用了循环CAS，即当一个线程想进入同步块的时候使用循环CAS的方式来获取锁，当它退出同步块的时候使用循环CAS释放锁。

2.4 本章小结

本章我们一起研究了volatile、synchronized和原子操作的实现原理。Java中的大部分容器和框架都依赖于本章介绍的volatile和原子操作的实现原理，了解这些原理对我们进行并发编程会更有帮助。