InfoQ 软件开发丛书 Software Development Series

Java并发编程的艺术

◎作者：方腾飞　　◎审校　张龙

InfoQ

目录

推荐序

欣闻腾飞兄弟的《聊聊并发》系列文章将要集结成 InfoQ 迷你书进行发布，我感到非常的振奋。这一系列文章从最开始的发布到现在已经经历了两年多的时间，这两年间，Java 世界发生了翻天覆地的变化。Java 7 已经发布，而且 Java 8 也将在下个月姗姗来迟。围绕着 JVM 已经形成了一个庞大且繁荣的生态圈，Groovy、Scala、Clojure、Ceylon 等众多 JVM 语言在蓬勃发展着，如今的 Java 已经不是几年前的 Java 了，众多运行在 JVM 上的编程语言为我们带来了更多的选择，提供了更好的机会。

纵观这几年的技术发展趋势，唱衰 Java 的论调一直都萦绕在我们耳边。不可否认，Java 的发展确实有些缓慢，而且有些臃肿；但放眼望去，有如此之多的核心与关键系统依旧在使用 Java 进行开发并运行在 JVM 之上，这不仅得益于 Java 语言本身，强大的 JVM 及繁荣的 Java 生态圈在这其中更是发挥着重要的作用。在 Java 的世界中，我们想要完成一件事情有太多可用的选择了。

虽然如此，对于国内的一些开发人员来说，但凡提到 Java，想到的都是所谓的 SSH（Struts、Spring 及 Hibernate 等相关框架）。不可否认，这些框架对于我们又快又好地完成任务起到了至关重要的推进作用，然而 Java 并不是 SSH，SSH 也不是 Java 的代名词。

由于之前的系列文章都是本人审校的，因此我也非常幸运地成为了这些文章的第一个读者，在阅读之际不禁感叹腾飞的技术造诣及对技术执着的追求。腾飞兄弟的《聊聊并发》系列文章从发布以来一直高居 InfoQ 中文站浏览量的前列，每篇文章之后都有大量的读者评论，或是提问，或是补充相关知识，腾飞兄弟也都非常耐心地对读者的问题进行解答。并发是一个学科，Java 中也有自己的一套处理并发的框架与体系；不过遗憾的是，很多读者对这一领域知之甚少，这也直接造成了很多人并不了解有关并发的理论与实践知识。幸运的是，腾飞的《聊聊并发》系列文章非常完美地填补了这一空白，文章从 synchronized 关键字、volatile 实现原理到 ConcurrentHashMap、ConcurrentLinkedQueue 源码分析，再到阻塞队列和 Fork/Join 框架，为读者献上了一道丰盛的 Java 并发大餐。

相信腾飞以在淘宝的实际工作经验凝结而成的这部 InfoQ 迷你书会为广大读者打开通往 Java 并发之路的大门。这里我要小声做一个提示，也许文章中很多内容看一次未必就能完全消化吸收，这时请不要放弃，多看几次，多动手做实验，相信你会很快掌握 Java 并发的精髓的。

另外，值得一提的是，腾飞兄弟现在在维护着一个关于 Java 并发资源的站点——并发编程网（http://ifeve.com/），上面有大量高质量的原创与翻译文章，都是关于并发领域相关内容的，感兴趣的读者不妨移步一观。

最后，祝大家阅读愉快，能够轻松驾驭 Java 并发。

是为序。

InfoQ 中文站 Java 主编：张龙

第九章 Java 并发容器和框架

Java 程序员进行并发编程时，相比于其他语言的程序员而言要倍感幸福，因为并发编程大师 Doug Lea 不遗余力的为 Java 开发者提供了非常多的并发容器和框架。本章让我们一起来见识下大师操刀编写的这些容器和框架，并通过每节的原理分析一起来学习下，如何设计出精妙的并发程序。

1.1 ConcurrentHashMap 的分析和使用

ConcurrentHashMap 是线程安全并且高效的 HashMap。本节让我们一起研究下该容器是如何在保证线程安全的同时又能保证高效的操作。

1.1.1 为什么要使用 ConcurrentHashMap

线程不安全的 HashMap。因为多线程环境下，使用 HashMap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap，如执行以下代码：

```
final HashMap<String, String> map = new HashMap<String, String>(2);
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    new Thread(new Runnable() {
                        @Override
                        public void run() {
                            map.put(UUID.randomUUID().toString(), "");
                        }
                    }, "ftf" + i).start();
                }
            }
        }, "ftf");
        t.start();
        t.join();
```

效率低下的 HashTable。HashTable 容器使用 synchronized 来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率

越低。

基于以上两个原因，所以便有了 ConcurrentHashMap 的登场机会。

1.1.2 ConcurrentHashMap 的锁分段技术

HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问 HashTable 的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap 所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

1.1.3 ConcurrentHashMap 的结构

我们通过 ConcurrentHashMap 的类图来分析 ConcurrentHashMap 的结构，如图 9-1 所示。

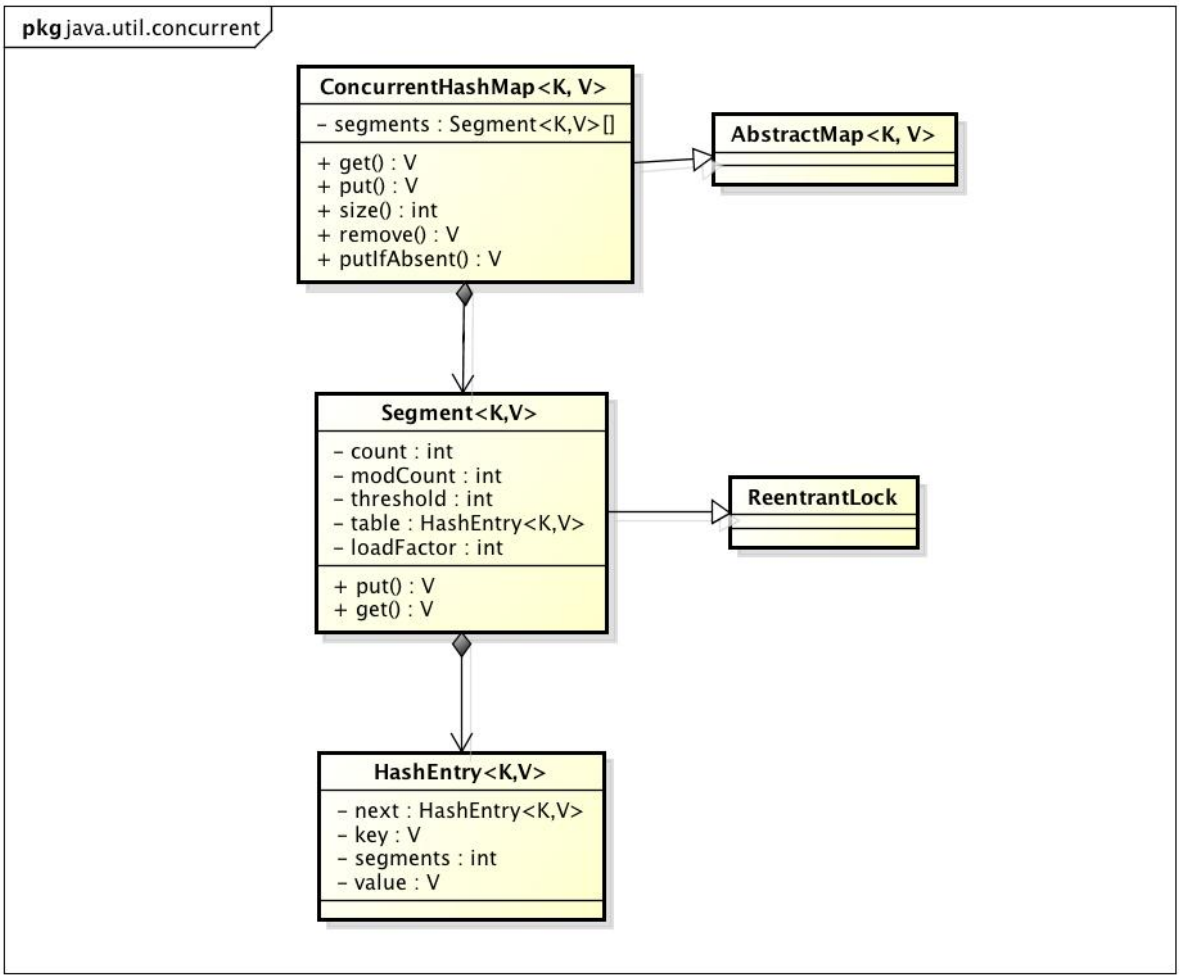


图1-1

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构， 一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素， 每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁，如图 7-2 所示。

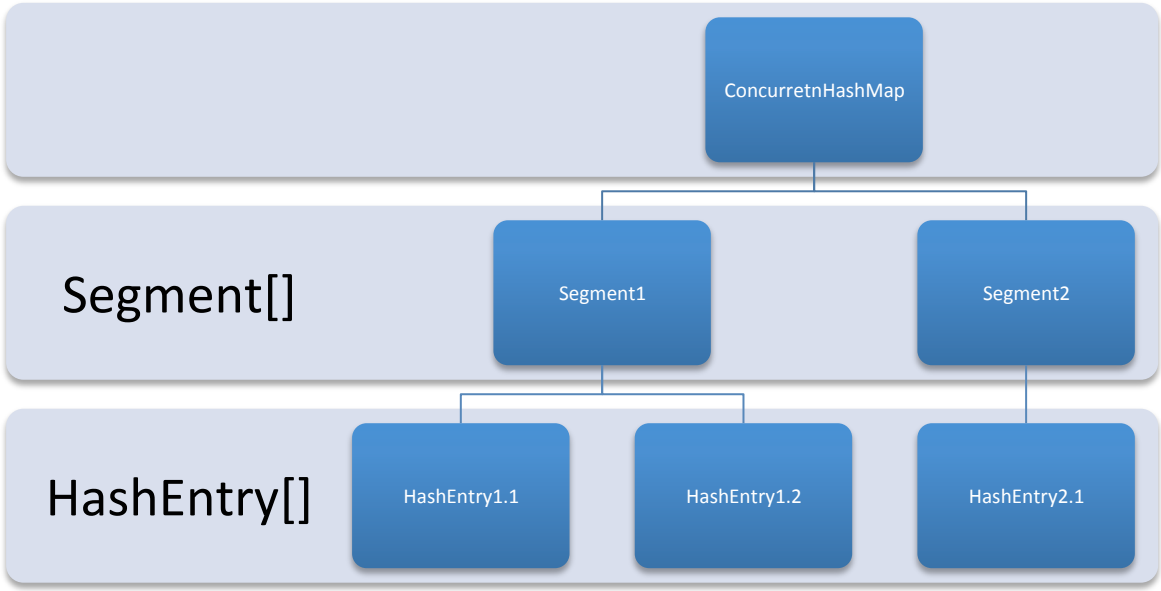


图1-2

1.1.4 ConcurrentHashMap 的初始化

ConcurrentHashMap 初始化方法是通过 initialCapacity，loadFactor, concurrencyLevel 几个参数来初始化 segments 数组，段偏移量 segmentShift，段掩码 segmentMask 和每个 segment 里的 HashEntry 数组。

初始化 segments 数组。让我们来看一下初始化 segmentShift，segmentMask 和 segments 数组的源代码。

```
if (concurrencyLevel > MAX_SEGMENTS)
            concurrencyLevel = MAX_SEGMENTS;

        // Find power-of-two sizes best matching arguments
        int sshift = 0;
        int ssize = 1;
        while (ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        segmentShift = 32 - sshift;
        segmentMask = ssize - 1;
        this.segments = Segment.newArray(ssize);
```

由上面的代码可知 segments 数组的长度 ssize 通过 concurrencyLevel 计算得出。为了能通过按位与的哈希算法来定位 segments 数组的索引，必须保证 segments 数组的长度是 2 的 N 次方（power-of-two size），所以必须计算出一个是大于或等于 concurrencyLevel 的最小的 2 的 N 次方值来作为 segments 数组的长度。假如 concurrencyLevel 等于 14，15 或 16，ssize 都会等于 16，即容器里锁的个数也是 16。

注意：concurrencyLevel 的最大大小是 65535，意味着 segments 数组的长度最大为 65536，对应的二进制是 16 位。

初始化 segmentShift 和 segmentMask。这两个全局变量在定位 segment 时的哈希算法里需要使用，sshift 等于 ssize 从 1 向左移位的次数，在默认情况下 concurrencyLevel 等于 16，1 需要向左移位移动 4 次，所以 sshift 等于 4。segmentShift 用于定位参与 hash 运算的位数，segmentShift 等于 32 减 sshift，所以等于 28，这里之所以用 32 是因为 ConcurrentHashMap 里的 hash()方法输出的最大数是 32 位的，后面的测试中我们可以看到这点。segmentMask 是哈希运算的掩码，等于 ssize 减 1，即 15，掩码的二进制各个位的值都是 1。因为 ssize 的最大长度是 65536，所以 segmentShift 最大值是 16，segmentMask 最大值是 65535，对应的二进制是 16 位，每个位都是 1。

初始化每个 Segment。输入参数 initialCapacity 是 ConcurrentHashMap 的初始化容量，loadfactor 是每个 segment 的负载因子，在构造方法里需要通过这两个参数来初始化数组中的每个 segment。

```
if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        int c = initialCapacity / ssize;
        if (c * ssize < initialCapacity)
            ++c;
        int cap = 1;
        while (cap < c)
            cap <<= 1;
        for (int i = 0; i < this.segments.length; ++i)
            this.segments[i] = new Segment<K,V>(cap, loadFactor);
```

上面代码中的变量 cap 就是 segment 里 HashEntry 数组的长度，它等于 initialCapacity 除以 ssize 的倍数 c，如果 c 大于 1，就会取大于等于 c 的 2 的 N 次方值，所以 cap 不是 1，就是 2 的 N 次方。segment 的容量 threshold＝(int)cap*loadFactor，默认情况下 initialCapacity 等于 16，loadfactor 等于 0.75，通过运算 cap 等于 1，threshold 等于零。

1.1.5 定位 Segment

既然 ConcurrentHashMap 使用分段锁 Segment 来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到 Segment。可以看到 ConcurrentHashMap 会首先使用 Wang/Jenkins hash 的变种算法对元素的 hashCode 进行一次再哈希。

```
private static int hash(int h) {
        h += (h << 15) ^ 0xffffcd7d;
        h ^= (h >>> 10);
        h += (h << 3);
        h ^= (h >>> 6);
        h += (h << 2) + (h << 14);
        return h ^ (h >>> 16);
    }
```

之所以进行再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的 Segment 上，从而提高容器的存取效率。假如哈希的质量差到极点，那么所有的元素都在一个 Segment 中，不仅存取元素缓慢，分段锁也会失去意义。我做了一个测试，不通过再哈希而直接执行哈希计算。

```
System.out.println(Integer.parseInt("0001111", 2) & 15);
System.out.println(Integer.parseInt("0011111", 2) & 15);
System.out.println(Integer.parseInt("0111111", 2) & 15);
System.out.println(Integer.parseInt("1111111", 2) & 15);
```

计算后输出的哈希值全是 15，通过这个例子可以发现如果不进行再哈希，哈希冲突会非常严重，因为只要低位一样，无论高位是什么数，其哈希值总是一样。我们再把上面的二进制数据进行再哈希后结果如下，为了方便阅读，不足 32 位的高位补了 0，每隔四位用竖线分割下。

```
0100|0111|0110|0111|1101|1010|0100|1110
1111|0111|0100|0011|0000|0001|1011|1000
0111|0111|0110|1001|0100|0110|0011|1110
1000|0011|0000|0000|1100|1000|0001|1010
```

可以发现每一位的数据都散列开了，通过这种再哈希能让数字的每一位都能参加到哈希运算当中，从而减少哈希冲突。ConcurrentHashMap 通过以下哈希算法定位 segment。

```
final Segment<K,V> segmentFor(int hash) {
        return segments[(hash >>> segmentShift) & segmentMask];
    }
```

默认情况下 segmentShift 为 28，segmentMask 为 15，再哈希后的数最大是 32 位二进制数据，向右无符号移动 28 位，意思是让高 4 位参与到 hash 运算中，(hash >>> segmentShift) & segmentMask

的运算结果分别是 4，15，7 和 8，可以看到 hash 值没有发生冲突。

1.1.6 ConcurrentHashMap 的 get 操作

Segment 的 get 操作实现非常简单和高效。先经过一次再哈希，然后使用这个哈希值通过哈希运算定位到 segment，再通过哈希算法定位到元素，代码如下：

```
public V get(Object key) {
    int hash = hash(key.hashCode());
    return segmentFor(hash).get(key, hash);
 }
```

get 操作的高效之处在于整个 get 过程不需要加锁，除非读到的值是空的才会加锁重读，我们知道 HashTable 容器的 get 方法是需要加锁的，那么 ConcurrentHashMap 的 get 操作是如何做到不加锁的呢？原因是它的 get 方法里将要使用的共享变量都定义成 volatile，如用于统计当前 Segement 大小的 count 字段和用于存储值的 HashEntry 的 value。定义成 volatile 的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在 get 操作里只需要读不需要写共享变量 count 和 value，所以可以不用加锁。只所以不会读到过期的值，是根据 java 内存模型的 happen before 原则，对 volatile 字段的写入操作先于读操作，即使两个线程同时修改和获取 volatile 变量，get 操作也能拿到最新的值，这是用 volatile 替换锁的经典应用场景。

```
transient volatile int count;
volatile V value;
```

在定位元素的代码里我们可以发现定位 HashEntry 和定位 Segment 的哈希算法虽然一样，都与数组的长度减去一相与，但是相与的值不一样，定位 Segment 使用的是元素的 hashcode 通过再哈希后得到的值的高位，而定位 HashEntry 直接使用的是再哈希后的值。其目的是避免两次哈希后的值一样，导致元素虽然在 Segment 里散列开了，但是却没有在 HashEntry 里散列开。

```
hash >>> segmentShift) & segmentMask//定位 Segment 所使用的 hash 算法
int index = hash & (tab.length - 1);// 定位 HashEntry 所使用的 hash 算法
```

1.1.7 ConcurrentHashMap 的 put 操作

由于 put 方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put 方法首先定位到 Segment，然后在 Segment 里进行插入操作。插入操作需要经历两个步

骤，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置然后放在 HashEntry 数组里。

是否需要扩容。在插入元素前会先判断 Segment 里的 HashEntry 数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment 的扩容判断比 HashMap 更恰当，因为 HashMap 是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时 HashMap 就进行了一次无效的扩容。

如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再 hash 后插入到新的数组里。为了高效 ConcurrentHashMap 不会对整个容器进行扩容，而只对某个 segment 进行扩容。

1.1.8 ConcurrentHashMap 的 size 操作

如果我们要统计整个 ConcurrentHashMap 里元素的大小，就必须统计所有 Segment 里元素的大小后求和。Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？不是的，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。所以最安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。

因为在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 2 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么 ConcurrentHashMap 是如何判断在统计的时候容器是否发生了变化呢？使用 modCount 变量，在 put , remove 和 clean 方法里操作元素前都会将变量 modCount 进行加 1，那么在统计 size 前后比较 modCount 是否发生变化，从而得知容器的大小是否发生变化。

1.2 ConcurrentLinkedQueue 的分析和使用

在并发编程中我们有时候需要使用线程安全的队列。如果我们要实现一个线程安全的队列有两种实现方式一种是使用阻塞算法，另一种是使用非阻塞算法。使用阻塞算法的队列可以用一个锁（入

队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现，而非阻塞的实现方式则可以使用循环 CAS 的方式来实现，本节让我们一起来研究下 Doug Lea 是如何使用非阻塞的方式来实现线程安全队列 ConcurrentLinkedQueue 的，相信从大师身上我们能学到不少并发编程的技巧。

1.2.1 ConcurrentLinkedQueue 的介绍

ConcurrentLinkedQueue 是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部，当我们获取一个元素时，它会返回队列头部的元素。它采用了“wait－free”算法（即 CAS 算法）来实现，该算法在 Michael & Scott 算法上进行了一些修改, Michael & Scott 算法的详细信息可以参见参考资料一。

1.2.2 ConcurrentLinkedQueue 的结构

我们通过 ConcurrentLinkedQueue 的类图来分析一下它的结构，如图 8-1 所示。

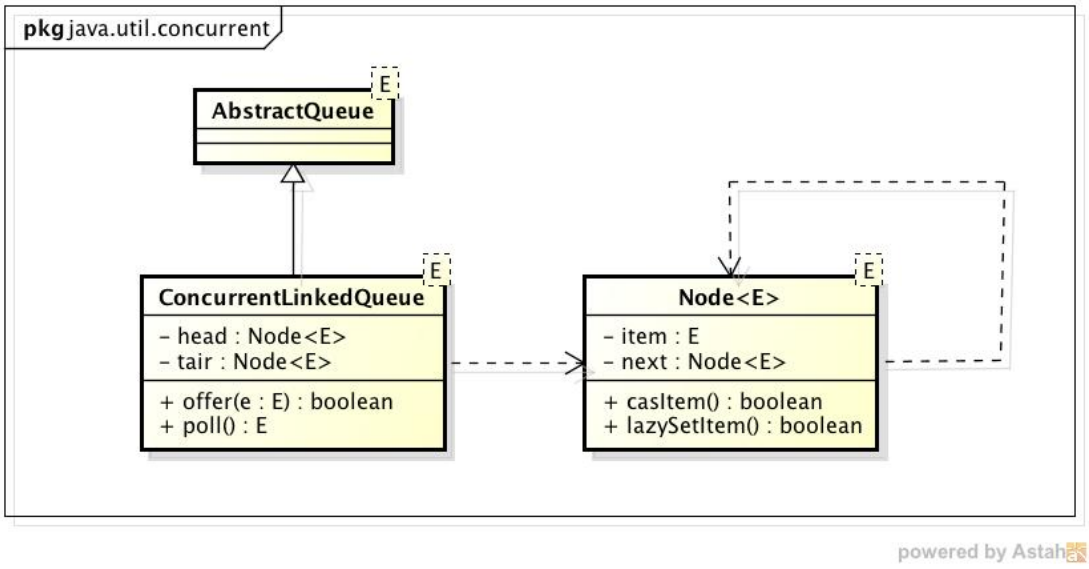


图1-3

ConcurrentLinkedQueue 由 head 节点和 tair 节点组成，每个节点（Node）由节点元素（item）和指向下一个节点的引用(next)组成，节点与节点之间就是通过这个 next 关联起来，从而组成一张链表结构的队列。默认情况下 head 节点存储的元素为空，tair 节点等于 head 节点。

```
private transient volatile Node<E> tail = head;
```

1.2.3 入队列

入队列就是将入队节点添加到队列的尾部。为了方便理解入队时队列的变化，以及 head 节点和 tair 节点的变化，每添加一个节点我就做了一个队列的快照图，如图 8-2 所示。

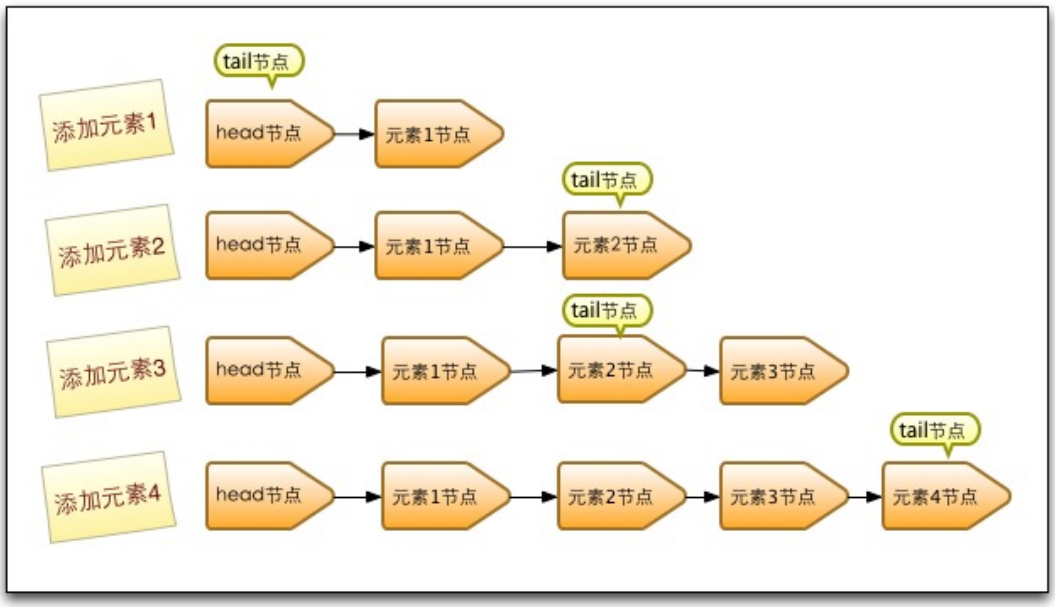


图1-4

步骤1 添加元素 1。队列更新 head 节点的 next 节点为元素 1 节点。又因为 tail 节点 默认情况下等于 head 节点，所以它们的 next 节点都指向元素 1 节点。

步骤2 添加元素 2。队列首先设置元素 1 节点的 next 节点为元素 2 节点，然后更新 tail 节点指向元素 2 节点。

步骤3 添加元素 3，设置 tail 节点的 next 节点为元素 3 节点。

步骤4 添加元素 4，设置元素 3 的 next 节点为元素 4 节点，然后将 tail 节点指向元素 4 节点。

通过 debug 入队过程并观察 head 节点和 tail 节点的变化，发现入队主要做两件事情，第一是将入队节点设置成当前队列尾节点的下一个节点。第二是更新 tail 节点，如果 tail 节点的 next 节点不为空，则将入队节点设置成 tail 节点，如果 tail 节点的 next 节点为空，则将入队节点设置成 tail 的 next 节点，所以 tail 节点不总是尾节点，理解这一点对于我们研究源码会非常有帮助。

上面的分析让我们从单线程入队的角度来理解入队过程，但是多个线程同时进行入队情况就变得更加复杂，因为可能会出现其他线程插队的情况。如果有一个线程正在入队，那么它必须先获取

尾节点，然后设置尾节点的下一个节点为入队节点，但这时可能有另外一个线程插队了，那么队列的尾节点就会发生变化，这时当前线程要暂停入队操作，然后重新获取尾节点。让我们再通过源码来详细分析下它是如何使用 CAS 算法来入队的。

```
   public boolean offer(E e) {
         if (e == null) throw new NullPointerException();
         //入队前，创建一个入队节点
         Node<E> n = new Node<E>(e);
         retry:
         //死循环，入队不成功反复入队。
         for (;;) {
             //创建一个指向 tail 节点的引用
             Node<E> t = tail;
             //p 用来表示队列的尾节点，默认情况下等于 tail 节点。
             Node<E> p = t;
             for (int hops = 0; ; hops++) {
             //获得 p 节点的下一个节点。
                 Node<E> next = succ(p);
       //next 节点不为空，说明 p 不是尾节点，需要更新 p 后在将它指向 next 节点
                 if (next != null) {
                    //循环了两次及其以上，并且当前节点还是不等于尾节点
                     if (hops > HOPS && t != tail)
                         continue retry;
                     p = next;
                 }
                 //如果 p 是尾节点，则设置 p 节点的 next 节点为入队节点。
                 else if (p.casNext(null, n)) {
                   //如果 tail 节点有大于等于 1 个 next 节点，则将入队节点设置成 tair 节点，更新失败了也没关系，
因为失败了表示有其他线程成功更新了 tair 节点。
   if (hops >= HOPS)
                         casTail(t, n); // 更新 tail 节点，允许失败
                     return true;
                 }
                // p 有 next 节点,表示 p 的 next 节点是尾节点，则重新设置 p 节点
                 else {
                     p = succ(p);
                 }
             }
         }
     }
```

从源代码角度来看整个入队过程主要做二件事情。第一是定位出尾节点，第二是使用 CAS 算法能将入队节点设置成尾节点的 next 节点，如不成功则重试。

定位尾节点。tail 节点并不总是尾节点，所以每次入队都必须先通过 tail 节点来找到尾节点，尾节点可能就是 tail 节点，也可能是 tail 节点的 next 节点。代码中循环体中的第一个 if 就是判断 tail 是否有 next 节点，有则表示 next 节点可能是尾节点。获取 tail 节点的 next 节点需要注意的是 p 节点等于 p 的 next 节点的情况，只有一种可能就是 p 节点和 p 的 next 节点都等于空，表示这个队列刚初始化，正准备添加第一次节点，所以需要返回 head 节点。获取 p 节点的 next 节点代码如下

```
final Node<E> succ(Node<E> p) {
      Node<E> next = p.getNext();
      return (p == next) ? head : next;
  }
```

设置入队节点为尾节点。p.casNext(**null**, n)方法用于将入队节点设置为当前队列尾节点的 next 节点，p 如果是 null 表示 p 是当前队列的尾节点，如果不为 null 表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

hops 的设计意图。上面分析过对于先进先出的队列入队所要做的事情就是将入队节点设置成尾节点，doug lea 写的代码和逻辑还是稍微有点复杂。那么我用以下方式来实现行不行？

```
public boolean offer(E e) {
      if (e == null)
                  throw new NullPointerException();
            Node<E> n = new Node<E>(e);
            for (;;) {
                  Node<E> t = tail;
                  if (t.casNext(null, n) && casTail(t, n)) {
                        return true;
                  }
            }
  }
```

让 tail 节点永远作为队列的尾节点，这样实现代码量非常少，而且逻辑非常清楚和易懂。但是这么做有个缺点就是每次都需要使用循环 CAS 更新 tail 节点。如果能减少 CAS 更新 tail 节点的次数，就能提高入队的效率，所以 doug lea 使用 hops 变量来控制并减少 tail 节点的更新频率，并不是每次节点入队后都将 tail 节点更新成尾节点，而是当 tail 节点和尾节点的距离大于等于常量 HOPS 的值（默认等于 1）时才更新 tail 节点，tail 和尾节点的距离越长使用 CAS 更新 tail 节点的次数就会越少，但是距离越长带来的负面效果就是每次入队时定位尾节点的时间就越长，因为循环体需要多循环一次来定位出尾节点，但是这样仍然能提高入队的效率，因为从本质上来看它通过增加对 volatile 变量的读操作来减少了对 volatile 变量的写操作，而对 volatile 变量的写操作开销要远

远大于读操作，所以入队效率会有所提升。

```
private static final int HOPS = 1;
```

注意：入队方法永远返回 true，所以不要通过返回值判断入队是否成功。

1.2.4 出队列

出队列的就是**从队列里返回一个节点元素**，并清空该节点对元素的引用。让我们通过每个节点出队的快照来观察下 head 节点的变化，如图 8-3 所示。

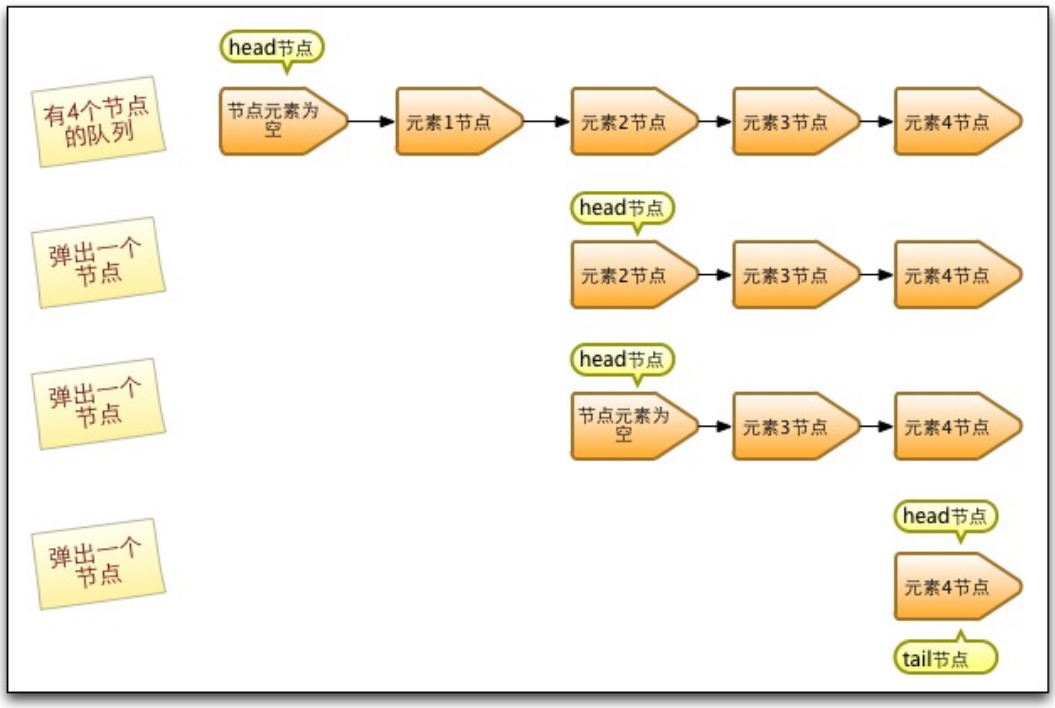


图1-5

从上图可知，并不是每次出队时都更新 head 节点，当 head 节点里有元素时，直接弹出 head 节点里的元素，而不会更新 head 节点。只有当 head 节点里没有元素时，出队操作才会更新 head 节点。这种做法也是通过 hops 变量来减少使用 CAS 更新 head 节点的消耗，从而提高出队效率。让我们再通过源码来深入分析下出队过程。

```
public E poll() {
                Node<E> h = head;
    // p 表示头节点，需要出队的节点
                Node<E> p = h;
                for (int hops = 0;; hops++) {
```

```
                                // 获取 p 节点的元素
                                E item = p.getItem();
                                // 如果 p 节点的元素不为空，使用 CAS 设置 p 节点引用的元素为 null,如果成功则
返回 p 节点的元素。
                                if (item != null && p.casItem(item, null)) {
                                        if (hops >= HOPS) {
                                                //将 p 节点下一个节点设置成 head 节点
                                                Node<E> q = p.getNext();
                                                updateHead(h, (q != null) ? q : p);
                                        }
                                        return item;
                                }
                                // 如果头节点的元素为空或头节点发生了变化，这说明头节点已经被另外一个线程修
改了。那么获取 p 节点的下一个节点
                                Node<E> next = succ(p);
                                // 如果 p 的下一个节点也为空，说明这个队列已经空了
                                if (next == null) {
                        // 更新头节点。
                                        updateHead(h, p);
                                        break;
                                }
                                // 如果下一个元素不为空，则将头节点的下一个节点设置成头节点
                                p = next;
                        }
                        return null;
                }
```

首先获取头节点的元素，然后判断头节点元素是否为空，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果不为空，则使用 CAS 的方式将头节点的引用设置成 null，如果 CAS 成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了 head 节点，导致元素发生了变化，需要重新获取头节点。

1.3 Java 线程池的分析和使用

1.3.1 为什么需要使用线程池

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系

统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

1.3.2 线程池的创建

我们可以通过 ThreadPoolExecutor 来创建一个线程池。

```
   new  ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime,
milliseconds,runnableTaskQueue, handler);
```

创建一个线程池需要输入几个参数：

- corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的 prestartAllCoreThreads 方法，线程池会提前创建并启动所有基本线程。
- runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。
  1) ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
  2) LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按 FIFO （先进先出） 排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法 Executors.newFixedThreadPool()使用了这个队列。
  3) SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。
  4) PriorityBlockingQueue：一个具有优先级得无限阻塞队列。
  5) maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
  6) ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。

7) RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 AbortPolicy，表示无法处理新任务时抛出异常。

1.3.3 线程池的饱和策略

在 JDK1.5 中 Java 线程池框架提供了以下四种策略：

- AbortPolicy：直接抛出异常。
- CallerRunsPolicy：只用调用者所在线程来运行任务。
- DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
- DiscardPolicy：不处理，丢弃掉。

当然也可以根据应用场景需要来实现 RejectedExecutionHandler 接口自定义策略。如记录日志或持久化不能处理的任务。

- keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
- TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

1.3.4 向线程池提交任务

我们可以使用 execute 提交的任务，但是 execute 方法没有返回值，所以无法判断任务知否被线程池执行成功。通过以下代码可知 execute 方法输入的任务是一个 Runnable 类的实例。

```
threadsPool.execute(new Runnable() {
                            @Override
                            public void run() {
                                    // TODO Auto-generated method stub
                            }
                    });
```

我们也可以使用 submit 方法来提交任务，它会返回一个 future, 通过这个 future 来判断任务是否执行成功，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用

get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

```
Future<Object> future = executor.submit(harReturnValuetask);
                try {
                        Object s = future.get();
                } catch (InterruptedException e) {
                        // 处理中断异常
                } catch (ExecutionException e) {
                        // 处理无法执行任务异常
                } finally {
                        // 关闭线程池
                        executor.shutdown();
                }
```

1.3.5 线程池的关闭

我们可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的的区别，shutdownNow 首先将线程池的状态设置成 STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而 shutdown 只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown 方法就会返回 true。当所有的任务都已关闭后, 才表示线程池关闭成功，这时调用 isTerminaed 方法会返回 true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用 shutdown 来关闭线程池，如果任务不一定要执行完，则可以调用 shutdownNow。

1.3.6 线程池的分析

线程池的主要工作流程如图 1-6：

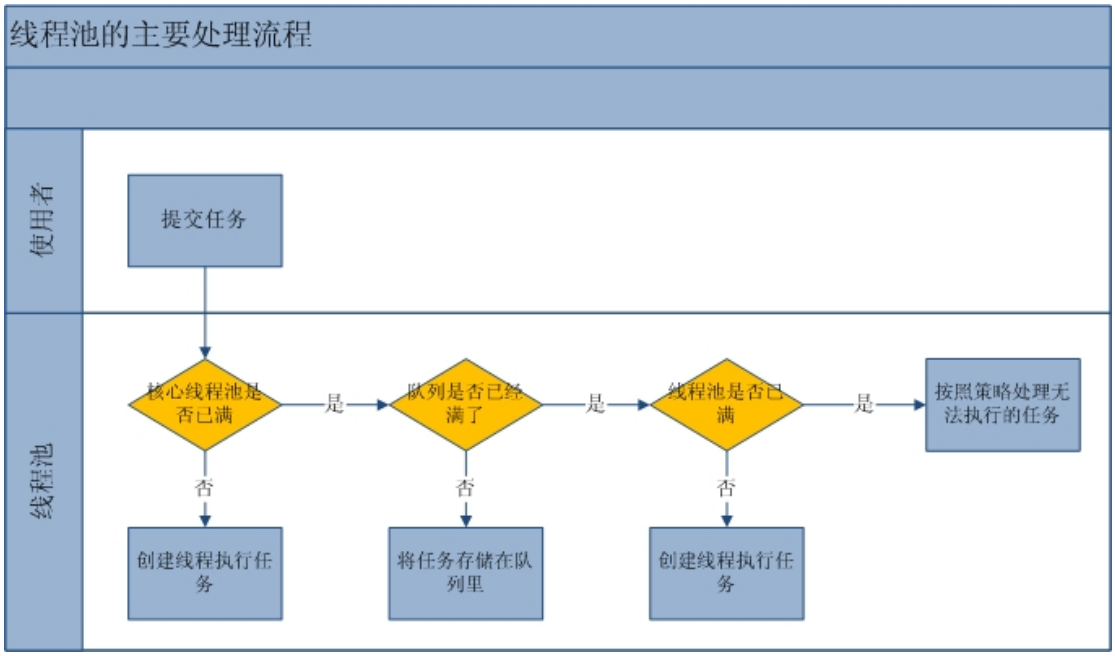


图1-6

从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1) 首先线程池判断**基本线程池**是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。

2) 其次线程池判断**工作队列**是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。

3) 最后线程池判断**整个线程池**是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

源码分析：上面的流程分析让我们很直观的了解的线程池的工作原理，让我们再通过源代码来看看是如何实现的，线程池执行任务的方法如下：

```
public void execute(Runnable command) {
        if (command == null)
           throw new NullPointerException();
//如果线程数小于基本线程数，则创建线程并执行当前任务
if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
//如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。
if (runState == RUNNING && workQueue.offer(command)) {
        if (runState != RUNNING || poolSize == 0)
                        ensureQueuedTaskHandled(command);
}
//如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，则创建一个线程执行任务。
```

```
    else if (!addIfUnderMaximumPoolSize(command))
    //抛出 RejectedExecutionException 异常
    reject(command); // is shutdown or saturated
                }
            }
```

工作线程:线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从 Worker 类的 run 方法里看到这点：

```
public void run() {
    try {
         Runnable task = firstTask;
         firstTask = null;
          while (task != null || (task = getTask()) != null) {
                runTask(task);
                task = null;
          }
     } finally {
          workerDone(this);
     }
}
```

1.3.7 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

- 任务的性质：CPU 密集型任务，IO 密集型任务和混合型任务。
- 任务的优先级：高，中和低。
- 任务的执行时间：长，中和短。
- 任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU 密集型任务配置尽可能小的线程，如配置 N_{cpu}+1 个线程的线程池。IO 密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如 2*N_{cpu}。混合型的任务，如果可以拆分，则将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过 Runtime.getRuntime().availableProcessors()方法获得当前设备的 CPU 个数。

优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 来处理。它可以让优先级高的任务先得到执行。

注意：如果一直有优先级高的任务提交到队列里，那么优先级高的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，如果等待的时间越长 CPU 空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用 CPU。

建议使用有界队列，有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行 SQL 变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

1.3.8 线程池的监控

通过线程池提供的参数进行监控。线程池里有以下属性在监控线程池的时候可以使用：

- taskCount：线程池需要执行的任务数量。
- completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于 taskCount。
- largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
- getPoolSize:线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不减。
- getActiveCount：获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的 beforeExecute，afterExecute 和 terminated 方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。如：

```
protected void beforeExecute(Thread t, Runnable r) { }
```

1.4 Java 中的阻塞队列

本节介绍了什么是阻塞队列，以及 Java 中阻塞队列的四种处理方式，并介绍和分析了 Java7 中提供的 7 种阻塞队列，最后分析阻塞队列的一种实现方式。

1.4.1 什么是阻塞队列？

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是支持阻塞的插入和移除方法。

1) 支持阻塞的插入方法：意思是当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2) 支持阻塞的移除方法：意思是在队列为空时，获取元素的线程会等待队列变为非空。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素，消费者用来获取元素的容器。

在阻塞队列不可用时，这两个附加操作提供了四种处理方式，如表 1-2 所示:

表1-1

方法\处理方式	抛出异常	返回特殊值	一直阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove()	poll()	take()	poll(time,unit)
检查方法	element()	peek()	不可用	不可用

- 抛出异常：当队列满时，如果再往队列里插入元素，会抛出 IllegalStateException("Queue full")异常。当队列空时，从队列里获取元素会抛出 NoSuchElementException 异常。
- 返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回 true。如果是移除方法，则是从队列里拿出一个元素，如果没有则返回 null。
- 一直阻塞：当阻塞队列满时，如果生产者继续线程往队列里 put 元素，队列会一直阻塞生产者线程，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里 take 元素，队列会阻塞住消费者线程，直到队列不为空。
- 超时退出：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

这 2 个附加操作的四种处理方式不方便记忆，所以我找了下这几个方法的规律。put 和 take 分别尾首含有字母 t，offer 和 poll 都含有字母 o。

注意：如果是无界阻塞队列，因为队列不可能会出现满的情况，所以使用 put 或 offer 方法永远不会被阻塞，而且使用 offer 方法时，该方法永远返回 true。

1.4.2 Java 里的阻塞队列

JDK7 提供了 7 个阻塞队列，分别是：

1) ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
2) LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
3) PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
4) DelayQueue：一个使用优先级队列实现的无界阻塞队列。
5) SynchronousQueue：一个不存储元素的阻塞队列。
6) LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
7) LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

1.4.3 ArrayBlockingQueue

ArrayBlockingQueue 是一个用数组实现的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。

默认情况下不保证线程公平的访问队列，所谓公平访问队列是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列。非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可以争夺访问队列的资格，有可能先阻塞的线程最后才访问队列。为了保证公平性通常情况下会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列：

```
ArrayBlockingQueue fairQueue = new  ArrayBlockingQueue(1000,true);
```

访问者的公平性是使用可重入锁实现的，代码如下：

```
public ArrayBlockingQueue(int capacity, boolean fair) {
      if (capacity <= 0)
         throw new IllegalArgumentException();
      this.items = new Object[capacity];
      lock = new ReentrantLock(fair);
      notEmpty = lock.newCondition();
      notFull =  lock.newCondition();
```

```
}
```

1.4.4 LinkedBlockingQueue

LinkedBlockingQueue 是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为 Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序。

1.4.5 PriorityBlockingQueue

PriorityBlockingQueue 是一个支持优先级的无界阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现 compareTo 方法来指定元素排序规则，或者初始化 PriorityBlockingQueue 时，指定构造参数 Comparator 来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

1.4.6 DelayQueue

DelayQueue 是一个支持延时获取元素的无界阻塞队列。队列使用 PriorityQueue 来实现。队列中的元素必须实现 Delayed 接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。

DelayQueue 非常有用，我们可以将 DelayQueue 运用在以下应用场景：

- 缓存系统的设计：可以用 DelayQueue 保存缓存元素的有效期，使用一个线程循环查询 DelayQueue，一旦能从 DelayQueue 中获取元素时，表示缓存有效期到了。
- 定时任务调度： 使用 DelayQueue 保存当天将会执行的任务和执行时间，一旦从 DelayQueue 中获取到任务就开始执行，比如 TimerQueue 就是使用 DelayQueue 实现的。

1. 如何实现 Delayed 接口呢？

DelayQueue 队列的元素必须实现 Delayed 接口。我们可以参考 ScheduledThreadPoolExecutor 里 ScheduledFutureTask 类的实现。一共三步，第一步：在对象创建的时候，初始化基本数据。使用 time 记录当前对象延迟到什么时候可以使用，使用 sequenceNumber 来标示元素在队列中的先后顺序。代码如下：

```
private static final AtomicLong sequencer = new AtomicLong(0);

ScheduledFutureTask(Runnable r, V result, long ns, long period) {
          super(r, result);
          this.time = ns;
          this.period = period;
          this.sequenceNumber = sequencer.getAndIncrement();
}
```

第二步：实现 getDelay 方法，该方法返回当前元素还需要延时多长时间，单位是毫秒，代码如下：

```
public long getDelay(TimeUnit unit) {
          return unit.convert(time - now(), TimeUnit.NANOSECONDS);
      }
```

通过构造函数可以看出延迟时间参数 ns 的单位是纳秒，自己设计的时候最好使用纳秒，因为 getDelay 时可以指定任意单位，一旦以秒或分作为单位，而延时时间又精确不到纳秒就麻烦了。使用时请注意当 time 小于当前时间时，getDelay 会返回负数。

第三步：实现 compareTo 方法来指定元素的顺序。比如让延时时间最长的放在队列的末尾。实现代码如下：

```
public int compareTo(Delayed other) {
          if (other == this) // compare zero ONLY if same object
              return 0;
          if (other instanceof ScheduledFutureTask) {
              ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;
              long diff = time - x.time;
              if (diff < 0)
                  return -1;
              else if (diff > 0)
                  return 1;
              else if (sequenceNumber < x.sequenceNumber)
                  return -1;
              else
                  return 1;
          }
          long d = (getDelay(TimeUnit.NANOSECONDS) -
                    other.getDelay(TimeUnit.NANOSECONDS));
          return (d == 0) ? 0 : ((d < 0) ? -1 : 1);
      }
```

2. 如何实现延时阻塞队列?

延时阻塞队列的实现很简单，当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

```
long delay = first.getDelay(TimeUnit.NANOSECONDS);
 if (delay <= 0)
     return q.poll();
else if (leader != null)
      available.await();
else {
     Thread thisThread = Thread.currentThread();
leader = thisThread;
       try {
                available.awaitNanos(delay);
            } finally {
                if (leader == thisThread)
                leader = null;
            }
       }
```

代码中的变量 leader 是一个等待获取队列头部元素的线程。如果 leader 不等于空，表示已经有线程在等待获取队列的头元素。所以使用 await 方法让当前线程等待信号。如果 leader 等于空，则把当前线程设置成 leader，并使用 awaitNanos 方法让当前线程等待接收信号或等待 `delay` 时间。

1.4.7 SynchronousQueue

SynchronousQueue 是一个不存储元素的阻塞队列。每一个 put 操作必须等待一个 take 操作，否则不能继续添加元素。

支持公平访问队列。默认情况下线程采用非公平性策略访问队列。使用以下构造方法可以创建公平性访问的 SynchronousQueue，如果设置为 true，则等待的线程会采用先进先出的顺序访问队列。

```
public SynchronousQueue(boolean fair) {
       transferer = fair ? new TransferQueue() : new TransferStack();
    }
```

SynchronousQueue 可以看成是一个传球手，负责把生产者线程处理的数据直接传递给消费者线程。队列本身并不存储任何元素，非常适合于传递性场景。SynchronousQueue 的吞吐量高于 LinkedBlockingQueue 和 ArrayBlockingQueue。

1.4.8 LinkedTransferQueue

LinkedTransferQueue 是一个由链表结构组成的无界阻塞 TransferQueue 队列。相对于其他阻塞队列，LinkedTransferQueue 多了 tryTransfer 和 transfer 方法。

transfer 方法。如果当前有消费者正在等待接收元素（消费者使用 take()方法或带时间限制的 poll()方法时），transfer 方法可以把生产者传入的元素立刻 transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer 方法会将元素存放在队列的 tail 节点，并等到该元素被消费者消费了才返回。transfer 方法的关键代码如下：

```
Node pred = tryAppend(s, haveData);
return awaitMatch(s, pred, e, (how == TIMED), nanos);
```

第一行代码是试图把存放当前元素的 s 节点作为 tail 节点。第二行代码是让 CPU 自旋等待消费者消费元素。因为自旋会消耗 CPU，所以自旋一定的次数后使用 Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

tryTransfer 方法。则是用来试探下生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回 false。和 transfer 方法的区别是 tryTransfer 方法无论消费者是否接收，方法立即返回。而 transfer 方法是必须等到消费者消费了才返回。

对于带有时间限制的 tryTransfer(E e, long timeout, TimeUnit unit)方法，则是试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回 false，如果在超时时间内消费了元素，则返回 true。

1.4.9 LinkedBlockingDeque

LinkedBlockingDeque 是一个由链表结构组成的双向阻塞队列。所谓双向队列指的你可以从队列的两端插入和移出元素。双端队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque 多了 addFirst，addLast，offerFirst，offerLast，peekFirst，peekLast 等方法，以 First 单词结尾的方法，表示插入，获取（peek）或移除双端队列的第一个元素。以 Last 单词结尾的方法，表示插入，获取或移除双端队列的最后一个元

素。另外插入方法 add 等同于 addLast，移除方法 remove 等效于 removeFirst。但是 take 方法却等同于 takeFirst，不知道是不是 Jdk 的 bug，使用时还是用带有 First 和 Last 后缀的方法更清楚。

在初始化 LinkedBlockingDeque 时可以设置容量防止其过渡膨胀。另外双向阻塞队列可以运用在“工作窃取”模式中。

1.4.10 阻塞队列的实现原理

如果队列是空的，消费者会一直等待，当生产者添加元素时候，消费者是如何知道当前队列有元素的呢？如果让你来设计阻塞队列你会如何设计，让生产者和消费者能够高效率的进行通讯呢？让我们先来看看 JDK 是如何实现的。

使用通知模式实现。所谓通知模式，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。通过查看 JDK 源码发现 ArrayBlockingQueue 使用了 Condition 来实现，代码如下：

```
private final Condition notFull;
private final Condition notEmpty;

public ArrayBlockingQueue(int capacity, boolean fair) {
        //省略其他代码
        notEmpty = lock.newCondition();
        notFull =  lock.newCondition();
    }

public void put(E e) throws InterruptedException {
        checkNotNull(e);
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            insert(e);
        } finally {
            lock.unlock();
        }
}

public E take() throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
```

```
        try {
            while (count == 0)
                notEmpty.await();
            return extract();
        } finally {
            lock.unlock();
        }
    }

    private void insert(E x) {
        items[putIndex] = x;
        putIndex = inc(putIndex);
        ++count;
        notEmpty.signal();
    }
```

当我们往队列里插入一个元素时，如果队列不可用，阻塞生产者主要通过 LockSupport.park(**this**)来实现。

```
    public final void await() throws InterruptedException {
            if (Thread.interrupted())
                throw new InterruptedException();
            Node node = addConditionWaiter();
            int savedState = fullyRelease(node);
            int interruptMode = 0;
            while (!isOnSyncQueue(node)) {
                LockSupport.park(this);
                if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
                    break;
            }
            if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
                interruptMode = REINTERRUPT;
            if (node.nextWaiter != null) // clean up if cancelled
                unlinkCancelledWaiters();
            if (interruptMode != 0)
                reportInterruptAfterWait(interruptMode);
        }
```

继续进入源码，发现调用 *setBlocker* 先保存下将要阻塞的线程，然后调用 unsafe.park 阻塞当前线程。

```
    public static void park(Object blocker) {
        Thread t = Thread.currentThread();
        setBlocker(t, blocker);
```

```
        unsafe.park(false, 0L);
        setBlocker(t, null);
    }
```

unsafe.park 是个 native 方法，代码如下：

```
public native void park(boolean isAbsolute, long time);
```

park 这个方法会阻塞当前线程，只有以下四种情况中的一种发生时，该方法才会返回。

1) 与 park 对应的 unpark 执行或已经执行时。已经执行是指 unpark 先执行，然后再执行的 park。
2) 线程被中断时。
3) 等待完 time 参数指定的毫秒数时。
4) 异常现象时，这个异常现象没有任何原因。

我们继续看一下 JVM 是如何实现 park 方法的，park 在不同的操作系统使用不同的方式实现，在 Linux 下是使用的是系统方法 pthread_cond_wait 实现。实现代码在 JVM 源码路径 src/os/linux/vm/os_linux.cpp 里的 os::PlatformEvent::park 方法，代码如下：

```
void os::PlatformEvent::park() {
    int v ;
            for (;;) {
                  v = _Event ;
            if (Atomic::cmpxchg (v-1, & _Event, v) == v) break ;
            }
            guarantee (v >= 0, "invariant") ;
            if (v == 0) {
            // Do this the hard way by blocking ...
            int status = pthread_mutex_lock(_mutex);
            assert_status(status == 0, status, "mutex_lock");
            guarantee (_nParked == 0, "invariant") ;
            ++ _nParked ;
            while (_Event < 0) {
            status = pthread_cond_wait(_cond, _mutex);
            // for some reason, under 2.7 lwp_cond_wait() may return ETIME ...
            // Treat this the same as if the wait was interrupted
            if (status == ETIME) { status = EINTR; }
            assert_status(status == 0 || status == EINTR, status, "cond_wait");
            }
            -- _nParked ;

            // In theory we could move the ST of 0 into _Event past the unlock(),
```

```
            // but then we'd need a MEMBAR after the ST.
            _Event = 0 ;
            status = pthread_mutex_unlock(_mutex);
            assert_status(status == 0, status, "mutex_unlock");
            }
            guarantee (_Event >= 0, "invariant") ;
            }
        }
```

pthread_cond_wait 是一个多线程的条件变量函数，cond 是 condition 的缩写，字面意思可以理解为线程在等待一个条件发生，这个条件是一个全局变量。这个方法接收两个参数，一个共享变量_cond，一个互斥量_mutex。而 unpark 方法在 linux 下是使用 pthread_cond_signal 实现的。park 在 windows 下则是使用 WaitForSingleObject 实现的。想知道 pthread_cond_wait 是如何实现的，可以参考 glibc-2.5 的 nptl/sysdeps/pthread/pthread_cond_wait.c。

当线程被阻塞队列阻塞时，线程会进入 WAITING (parking)状态。我们可以使用 jstack dump 阻塞的生产者线程看到这点：

```
   "main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting on condition [0x000000010164d000]
      java.lang.Thread.State: WAITING (parking)
           at sun.misc.Unsafe.park(Native Method)
           - parking to wait for  <0x0000000140559fe8> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
           at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
           at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSy
nchronizer.java:2043)
           at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:324)
           at blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQueueTest.java:11)
```

1.5 Fork/Join 框架

1.5.1 什么是 Fork/Join 框架

Fork/Join 框架是 Java7 提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

我们再通过 Fork 和 Join 这两个单词来理解下 Fork/Join 框架，Fork 就是把一个大任务切分为若干子任务并行的执行，Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如

计算 1+2+。。＋10000，可以分割成 10 个子任务，每个子任务分别对 1 千个数进行求和，最终汇总这 10 个子任务的结果。Fork/Join 的运行流程图如图 1-7 所示：

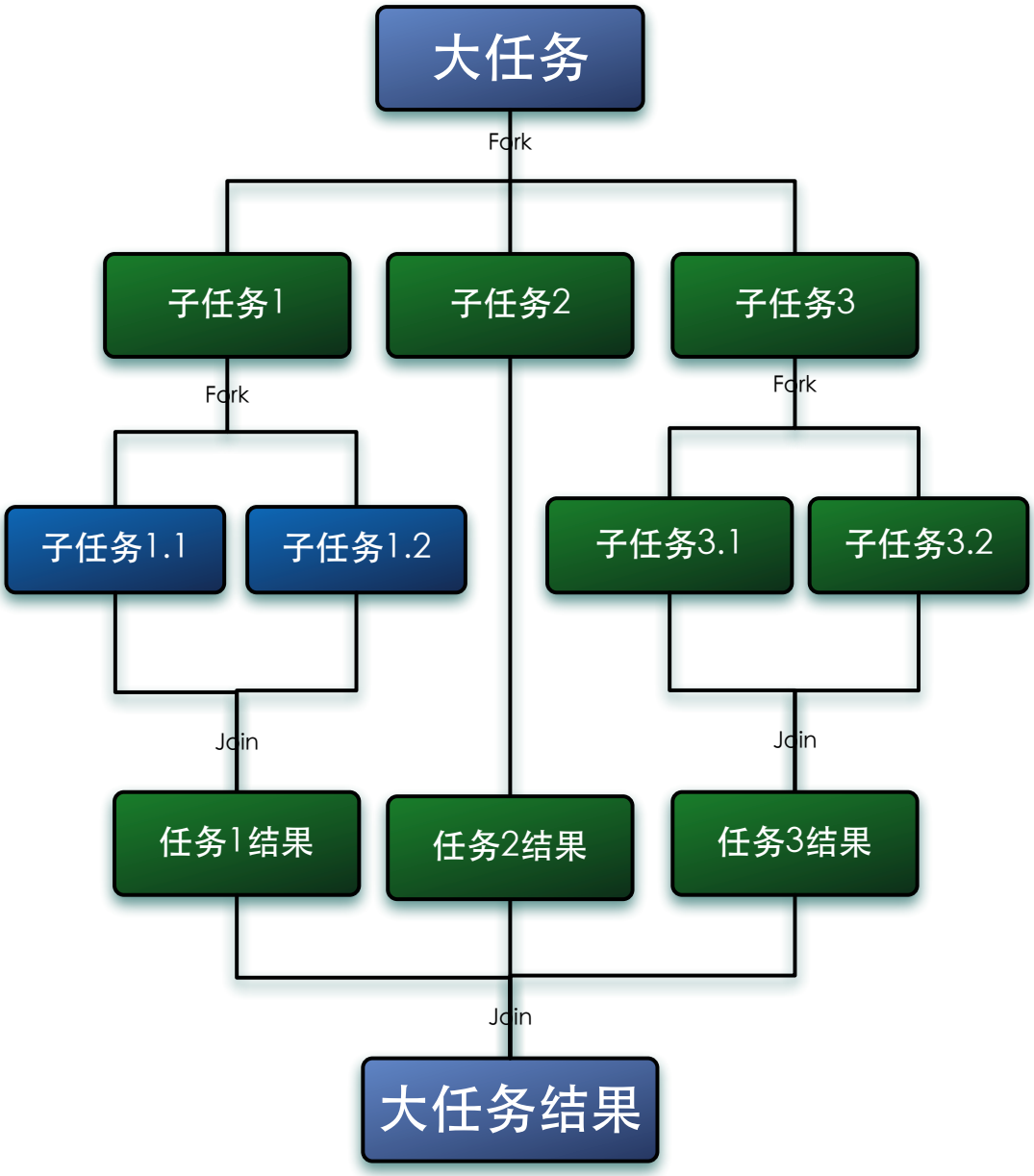


图1-7

1.5.2 工作窃取算法

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如图 1-8 下：

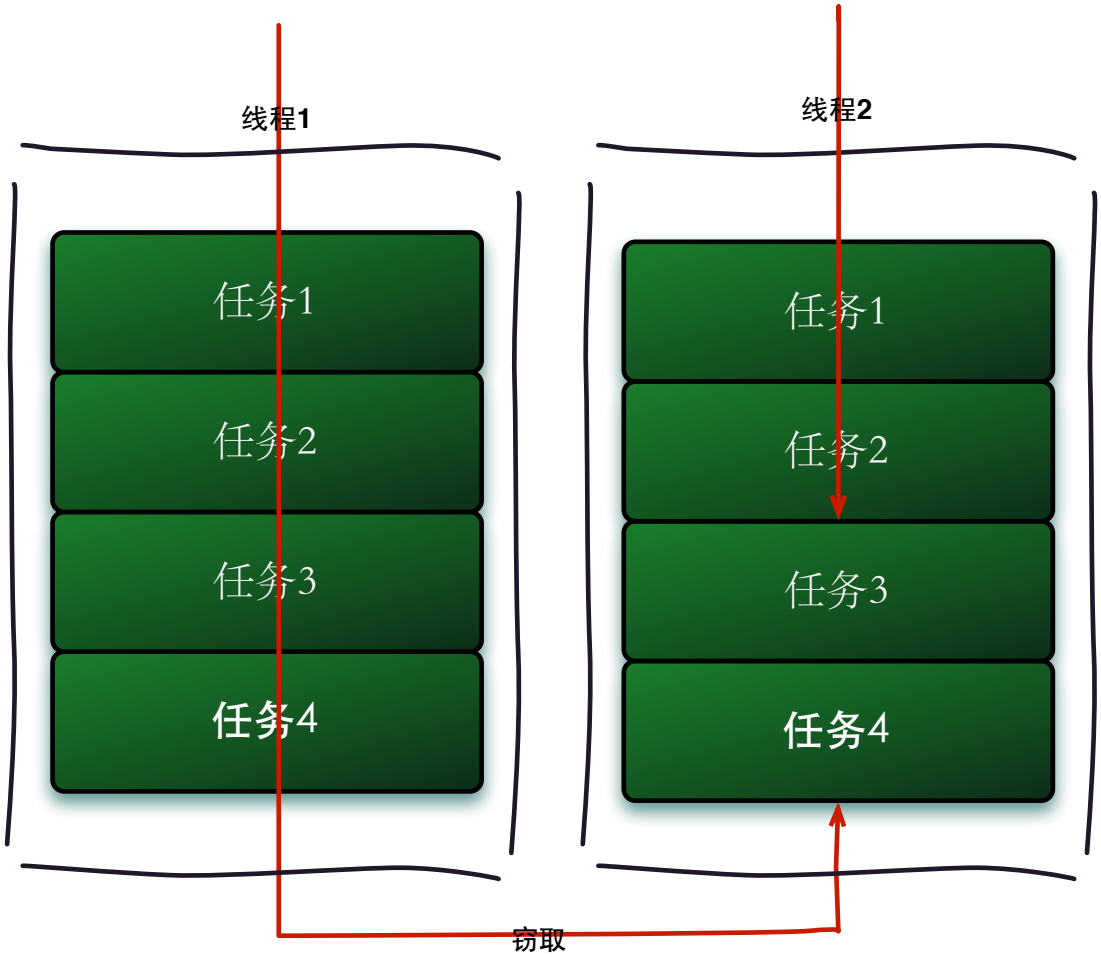


图1-8

那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如 A 线程负责处理 A 队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

1.5.3 Fork/Join 框架的介绍

我们已经很清楚 Fork/Join 框架的需求了，那么我们可以思考一下，如果让我们来设计一个 Fork/Join 框架，该如何设计？这个思考有助于你理解 Fork/Join 框架的设计。

步骤1 分割任务。首先我们需要有一个 fork 类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停的分割，直到分割出的子任务足够小。

步骤2 执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join 使用两个类来完成以上两件事情：

1) ForkJoinTask：我们要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork()和 join()操作的机制，通常情况下我们不需要直接继承 ForkJoinTask 类，而只需要继承它的子类，Fork/Join 框架提供了以下两个子类：
   - RecursiveAction：用于没有返回结果的任务。
   - RecursiveTask ：用于有返回结果的任务。
2) ForkJoinPool ：ForkJoinTask 需要通过 ForkJoinPool 来执行。

任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

1.5.4 使用 Fork／Join 框架

让我们通过一个简单的需求来使用下 Fork／Join 框架，需求是：计算 1+2+3+4 的结果。

使用 Fork／Join 框架首先要考虑到的是如何分割任务，如果我们希望每个子任务最多执行两个数的相加，那么我们设置分割的阈值是 2，由于是 4 个数字相加，所以 Fork／Join 框架会把这个任务 fork 成两个子任务，子任务一负责计算 1+2，子任务二负责计算 3+4，然后再 join 两个子任务的结果。因为是有结果的任务，所以必须继承 RecursiveTask，实现代码如下：

```
package fj;


import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
```

```
import java.util.concurrent.RecursiveTask;

public class CountTask extends RecursiveTask<Integer> {

        private static final int THRESHOLD = 2;// 阈值
        private int start;
        private int end;

        public CountTask(int start, int end) {
                this.start = start;
                this.end = end;
        }

        @Override
        protected Integer compute() {
                int sum = 0;

                // 如果任务足够小就计算任务
                boolean canCompute = (end - start) <= THRESHOLD;
                if (canCompute) {
                        for (int i = start; i <= end; i++) {
                                sum += i;
                        }
                } else {
                        // 如果任务大于阈值，就分裂成两个子任务计算
                        int middle = (start + end) / 2;
                        CountTask leftTask = new CountTask(start, middle);
                        CountTask rightTask = new CountTask(middle + 1, end);
                        //执行子任务
                        leftTask.fork();
                        rightTask.fork();
                        //等待子任务执行完，并得到其结果
                        int leftResult=leftTask.join();
                        int rightResult=rightTask.join();
                        //合并子任务
                        sum = leftResult  + rightResult;
                }
                return sum;
        }

        public static void main(String[] args) {
                ForkJoinPool forkJoinPool = new ForkJoinPool();
                // 生成一个计算任务，负责计算 1+2+3+4
```

```
                    CountTask task = new CountTask(1, 4);
                    // 执行一个任务
                    Future<Integer> result = forkJoinPool.submit(task);
                    try {
                              System.out.println(result.get());
                    } catch (InterruptedException e) {
                    } catch (ExecutionException e) {
                    }
          }

}
```

通过这个例子让我们再来进一步了解 ForkJoinTask，ForkJoinTask 与一般的任务的主要区别在于它需要实现 compute 方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用 fork 方法时，又会进入 compute 方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用 join 方法会等待子任务执行完并得到其结果。

1.5.5 Fork/Join 框架的异常处理

ForkJoinTask 在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以 ForkJoinTask 提供了 isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过 ForkJoinTask 的 getException 方法获取异常。使用如下代码：

```
if(task.isCompletedAbnormally())
                    {
                              System.out.println(task.getException());
                    }
```

getException 方法返回 Throwable 对象，如果任务被取消了则返回 CancellationException。如果任务没有完成或者没有抛出异常则返回 null。

1.5.6 Fork/Join 框架的实现原理

ForkJoinPool 由 ForkJoinTask 数组 和 ForkJoinWorkerThread 数组组成，ForkJoinTask 数组负

责存放程序提交给 ForkJoinPool 的任务，而 ForkJoinWorkerThread 数组负责执行这些任务。

ForkJoinTask 的 fork 方法实现原理。当我们调用 ForkJoinTask 的 fork 方法时，程序会调用 ForkJoinWorkerThread 的 pushTask 方法异步的执行这个任务，然后立即返回结果。代码如下：

```
public final ForkJoinTask<V> fork() {
        ((ForkJoinWorkerThread) Thread.currentThread())
            .pushTask(this);
        return this;
}
```

pushTask 方法把当前任务存放在 ForkJoinTask 数组 queue 里。然后再调用 ForkJoinPool 的 signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下：

```
final void pushTask(ForkJoinTask<?> t) {
        ForkJoinTask<?>[] q; int s, m;
        if ((q = queue) != null) {    // ignore if queue removed
            long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;
            UNSAFE.putOrderedObject(q, u, t);
            queueTop = s + 1;         // or use putOrderedInt
            if ((s -= queueBase) <= 2)
                pool.signalWork();
            else if (s == m)
                growQueue();
        }
    }
```

ForkJoinTask 的 join 方法实现原理。Join 方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看 ForkJoinTask 的 join 方法的实现，代码如下：

```
public final V join() {
        if (doJoin() != NORMAL)
            return reportResult();
        else
            return getRawResult();
}
private V reportResult() {
        int s; Throwable ex;
        if ((s = status) == CANCELLED)
            throw new CancellationException();
        if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)
            UNSAFE.throwException(ex);
        return getRawResult();
    }
```

首先，它调用了 doJoin()方法，通过 doJoin()方法得到当前任务的状态来判断返回什么结果，任务状态有四种：已完成（NORMAL），被取消（CANCELLED），信号（SIGNAL）和出现异常（EXCEPTIONAL）。

- 如果任务状态是已完成，则直接返回任务结果。
- 如果任务状态是被取消，则直接抛出 CancellationException。
- 如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析下 doJoin()方法的实现代码：

```
private int doJoin() {
      Thread t; ForkJoinWorkerThread w; int s; boolean completed;
      if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
         if ((s = status) < 0)
            return s;
         if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {
            try {
               completed = exec();
            } catch (Throwable rex) {
               return setExceptionalCompletion(rex);
            }
            if (completed)
               return setCompletion(NORMAL);
         }
         return w.joinTask(this);
      }
      else
         return externalAwaitDone();
   }
```

在 doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为 NORMAL，如果出现异常，则纪录异常，并将任务状态设置为 EXCEPTIONAL。

1.6 本章小结

通过本章介绍了 Java 中提供的各种并发容器和框架，每个小结都会有该容器和框架的原理分析，从中我们能够领略到大师级的设计思路，希望读者能够充分理解这种设计思想，并在以后自己开发的并发程序时，运用上这些并发编程的技巧。

第十三章 Java 并发机制的底层实现原理

Java 代码在编译后会变成字节码，然后在 JVM 里运行，而字节码最终需要转化为汇编代码在 CPU 上执行，所以 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。

1.7 Volatile 的实现原理

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。它在某些情况下比 synchronized 的开销更小，本文将深入分析在硬件层面上 Inter 处理器是如何实现 Volatile 的，通过深入分析能帮助我们正确的使用 Volatile 变量。

1.7.1 术语定义

表1-2

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。
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。

1.7.2 Volatile 的官方定义

Java 语言规范第三版中对 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的更新，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。

1.7.3为什么要使用 Volatile

Volatile 变量修饰符如果使用**恰当**的话，它比 synchronized 的**使用和执行成本会更低**，因为它不会引起线程上下文的切换和调度。

1.7.4Volatile 的实现原理

那么 Volatile 是如何来保证可见性的呢？在 x86 处理器下通过工具获取 JIT 编译器生成的汇编指令来看看对 Volatile 进行写操作 CPU 会做什么事情。

Java 代码：

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

汇编代码：

```
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) 将当前处理器缓存行的数据会写回到系统内存。

2) 这个写回内存的操作会引起在其他 CPU 里缓存了该内存地址的数据无效。

处理器为了提高处理速度，不直接和内存进行通讯，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，但操作完之后不知道何时会写到内存，如果对声明了 Volatile 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，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

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（第八章）中有详细阐述。

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，处理器可以独占使用任何共享内存。（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），但是在最近的处理器里，LOCK＃信号一般不锁总线，而是锁缓存，毕竟锁总线开销比较大。在 8.1.4 章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 和最近的处理器中，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。相反地，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被两个以上处理器缓存的内存区域数据**。

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。它们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，而这个地址当前处理共享状态，那么正在嗅探的处理器将无效它的缓存行，在下次访问相同内存地址时，强制执行缓存行填充。

1.7.5Volatile 的使用优化

著名的 Java 并发编程大师 Doug lea 在 JDK7 的并发包里新增一个队列集合类 LinkedTransferQueue，他在使用 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。

```
/** head of the queue */
private transient final PaddedAtomicReference<QNode> head;

/** tail of the queue */
private transient final PaddedAtomicReference<QNode> tail;

static final class PaddedAtomicReference <T> extends AtomicReference <T> {

  // enough padding for 64bytes with 4byte refs
  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队列（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， 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接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是需要不停修改头接点和尾节点，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，避免头接点和尾节点加载到同一个缓存行，使得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。第一：**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。第二：**共享变量不会被频繁的写**。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，共享变量如果不被频繁写的话，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。

1.8 Synchronized 的实现原理

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 SE1.6 对 Synchronized 进行了各种优化之后，有些情况下它并不那么重了，本文详细介绍了 Java SE1.6 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。

1.8.1 同步的基础

Java 中的每一个对象都可以作为锁。

- 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。那么锁存在哪里呢？锁里面会存储什么信息呢？

1.8.2同步的原理

JVM 规范规定 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，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。

1.8.3 Java 对象头

锁存在 Java 对象头里。如果对象是数组类型，则虚拟机用 3 个 Word（字宽）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，一字宽等于四字节，即 32bit。如表 1-2 所示：

表1-3

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 1-3 所示：

表1-4

锁状态	25 bit	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 1-4 所示：

表1-5

锁状态	25 bit		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 1-5 所示：

表1-6

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

1.8.4 锁的升级

Java SE1.6 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在 Java SE1.6 里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它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。

1.8.5 偏向锁

Hotspot 的作者经过以往的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花费 CAS 操作来加锁和解锁，而只需简单的测试一下对象头的 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。

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字节码正在执行），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。下图中的线程 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。

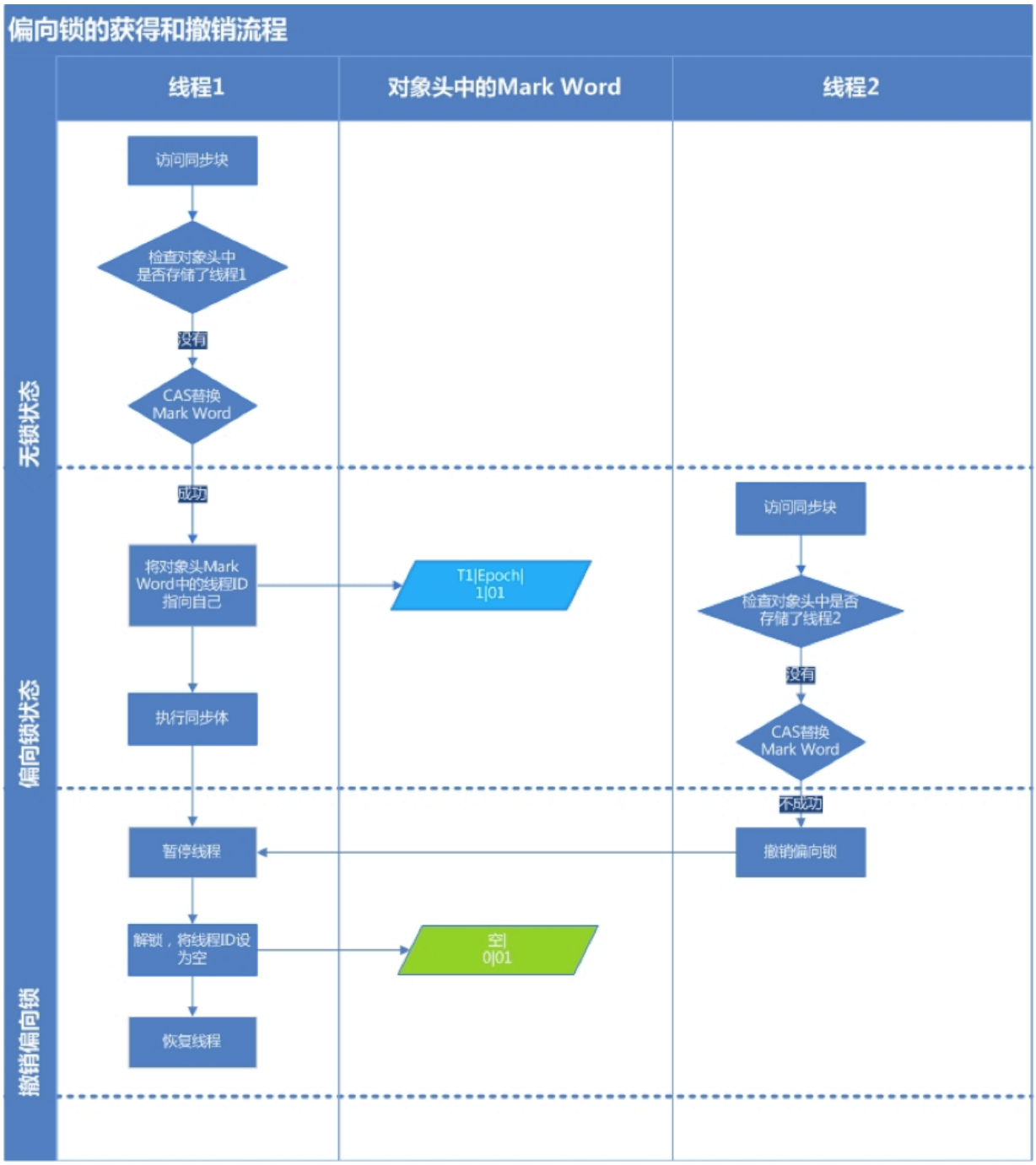


图1-9

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。如果你确定自己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，那么默认会进入轻量级锁状态。

1.8.6 轻量级锁

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。

轻量级锁解锁：轻量级解锁时，会使用原子的 CAS 操作来将 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。下图是两个线程同时争夺锁，导致锁膨胀的流程图。

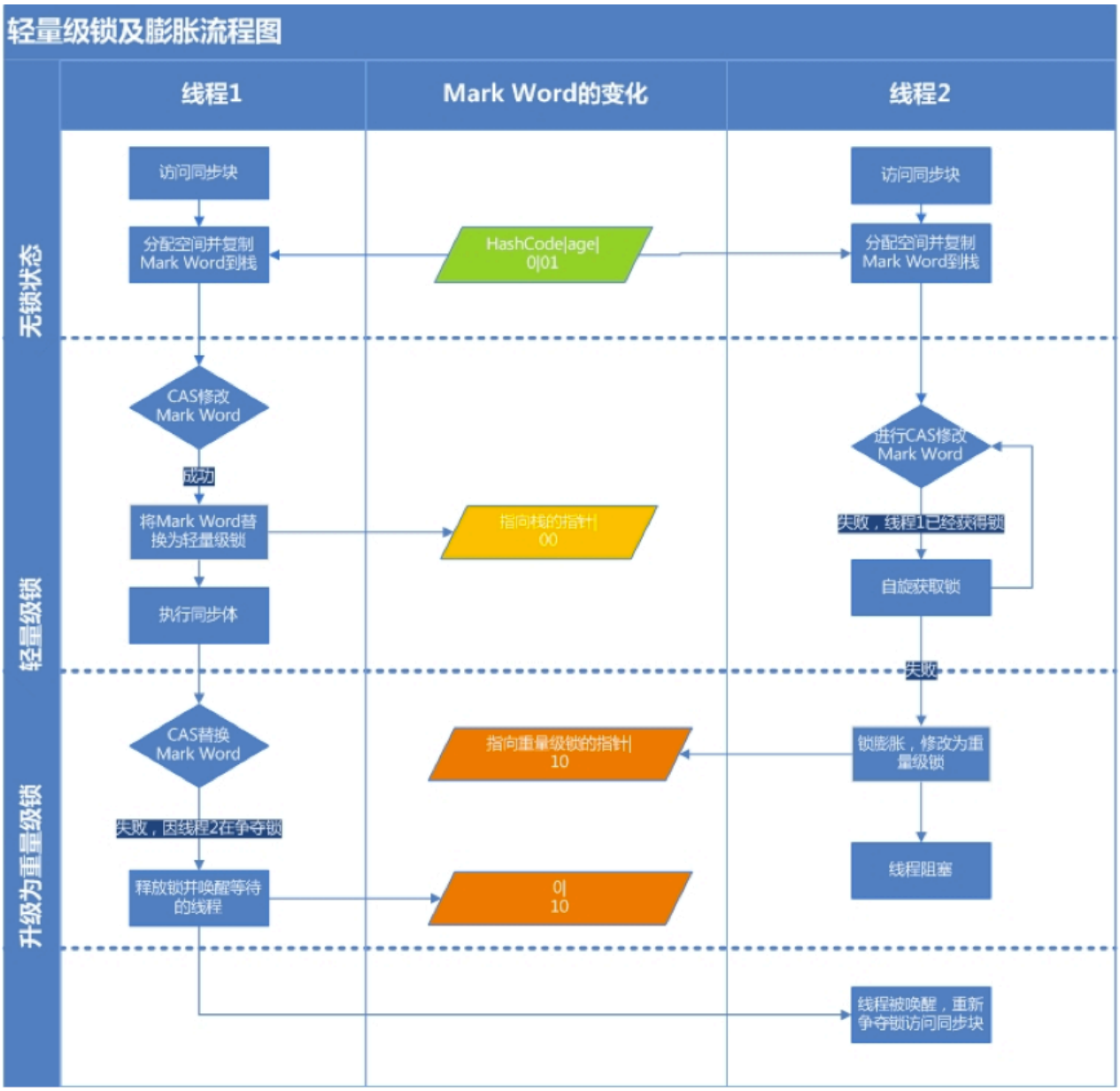


图1-10

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。

1.8.7 锁的优缺点对比

表1-7

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

1.8.8 6 参考源码

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。

1.9 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。本文让我们一起来聊一聊在 Inter 处理器和 Java 里是如何实现原子操作的。

1.9.1 术语定义

表1-8

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 期间先比较下在旧值有没有发生变化，如果没有发生变化，才交换成新值， 了变化则不交换
CPU 流水线	CPU pipeline	CPU 流水线的工作方式就象工业生产上的装配流水线，在 CPU 中由 5~6 个 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 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 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，假共享是指多个 CPU 同时修改同一个缓 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， 必须清空流水线

1.9.2 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。

1.9.3处理器自动保证基本内存操作的原子性

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当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。奔腾 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不能自动保证其原子性，比如跨总线宽度，跨多个缓存行，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。

1.9.4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(i++就是经典的读改写操作)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。如下图

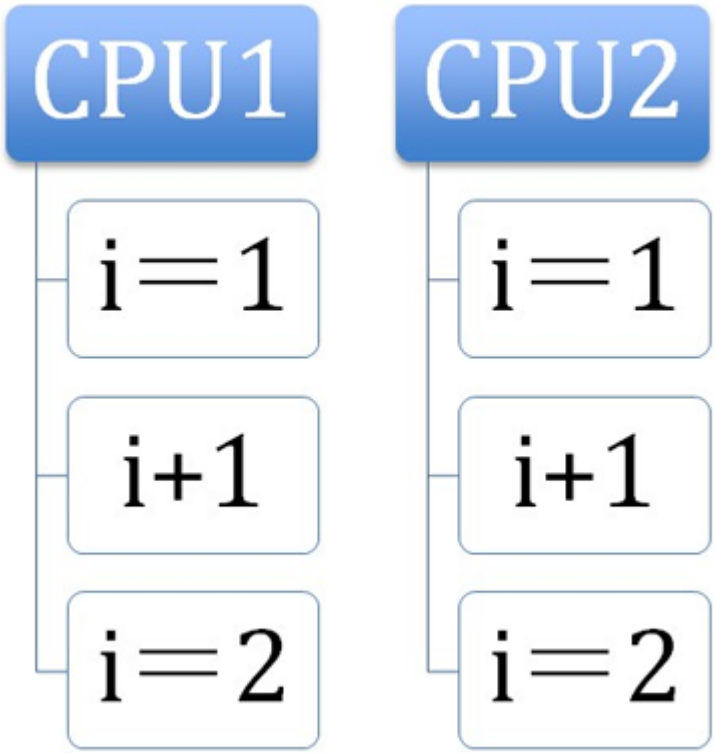


图1-11

原因是有可能多个处理器同时从各自的缓存中读取变量 i，分别进行加一操作，然后分别写入系统内存当中。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使用共享内存。

1.9.5使用缓存锁保证原子性

第二个机制是通过缓存锁定保证原子性。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，最近的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，在奔腾 6 和最近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”就是如果缓存在处理器缓存行中内存区域在 LOCK 操作期间被锁定，当它执行锁操作回写内存时，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被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起缓存行无效，在例 1 中，当 CPU1 修改缓存行中的 i 时使用缓存锁定，那么 CPU2 就不能同时缓存了 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），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 Inter486 和奔腾处理器,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。

以上两个机制我们可以通过 Inter 处理器提供了很多 LOCK 前缀的指令来实现。比如位测试和修改指令 BTS，BTR，BTC，交换指令 XADD，CMPXCHG 和其他一些操作数和逻辑指令，比如 ADD（加），OR（或）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。

1.9.6Java 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。

1.9.7使用循环 CAS 实现原子操作

JVM 中的 CAS 操作正是利用了上一节中提到的处理器提供的 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 Java1.5 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），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。

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 虽然很高效的解决原子操作，但是 CAS 仍然存在三大问题。ABA 问题，循环时间长开销大和只能保证一个共享变量的原子操作。

ABA 问题。因为 CAS 需要在操作值的时候检查下值有没有发生变化，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一，那么 A－B－A 就会变成 1A-2B－3A。从 Java1.5 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 方法作用是首先检查当前引用是否等于预期引用，并且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
)
```

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。

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，或者有一个取巧的办法，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 Java1.5 开始 JDK 提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。

1.9.8使用锁机制实现原子操作

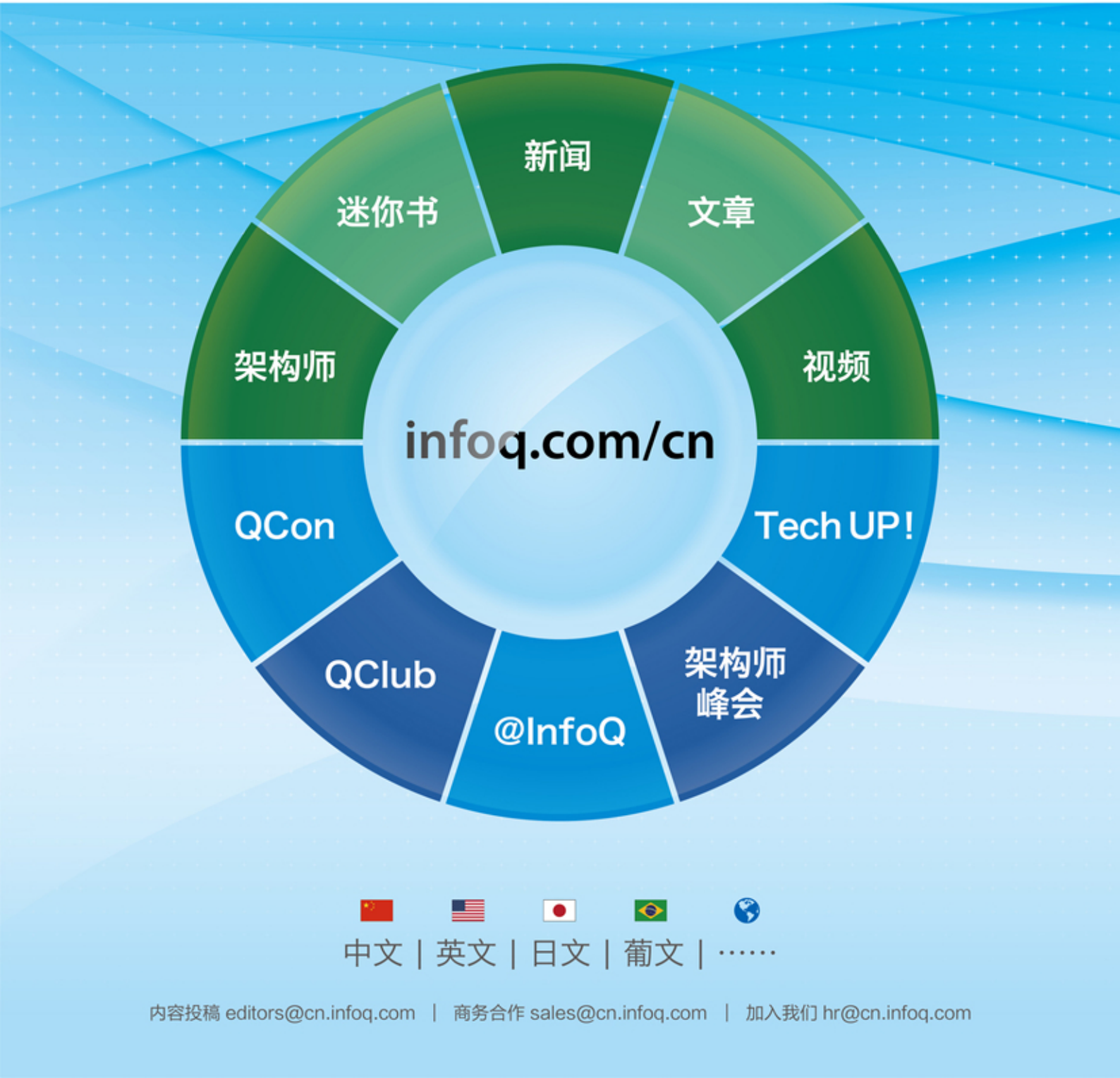
锁机制保证了只有获得锁的线程能够操作锁定的内存区域。JVM 内部实现了很多种锁机制，有偏向锁，轻量级锁和互斥锁，有意思的是除了偏向锁，JVM 实现锁的方式都用到的循环 CAS，当一个线程想进入同步块的时候使用循环 CAS 的方式来获取锁，当它退出同步块的时候使用循环 CAS 释放锁。

1.10 本章小节

在本章节中我们一起研究下 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节介绍的 Volatile 和原子操作的实现原理，了解这些原理对进行并发编程会更有帮助。

InfoQueue

促进软件开发领域知识与创新的传播

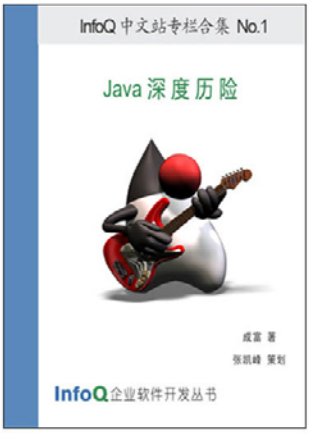


中文 | 英文 | 日文 | 葡文 | ……

内容投稿 editors@cn.infoq.com | 商务合作 sales@cn.infoq.com | 加入我们 hr@cn.infoq.com

InfoQ软件开发丛书

欢迎免费下载



商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com


1kg.org
多背一公斤
爱自然 | 更爱孩子