

第6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不遗余力地为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一下大师操刀编写的并发容器和框架，并通过每节的原理分析一起来学习如何设计出精妙的并发程序。

6.1 ConcurrentHashMap的实现原理与使用

ConcurrentHashMap是线程安全且高效的HashMap。本节让我们一起研究一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。

6.1.1 为什么要使用ConcurrentHashMap

在并发编程中使用HashMap可能导致程序死循环。而使用线程安全的HashTable效率又非常低下，基于以上两个原因，便有了ConcurrentHashMap的登场机会。

(1)线程不安全的HashMap

在多线程环境下，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。例如，执行以下代码会引起死循环。

```
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();
```

HashMap在并发执行put操作时会引起死循环，是因为多线程会导致HashMap的Entry链表形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获取Entry。

(2)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，其他线程也访问HashTable的同步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方

法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

(3)ConcurrentHashMap的锁分段技术可有效提升并发访问率

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，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地存储，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。

6.1.2 ConcurrentHashMap的结构

通过ConcurrentHashMap的类图来分析ConcurrentHashMap的结构，如图6-1所示。

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守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁，如图6-2所示。

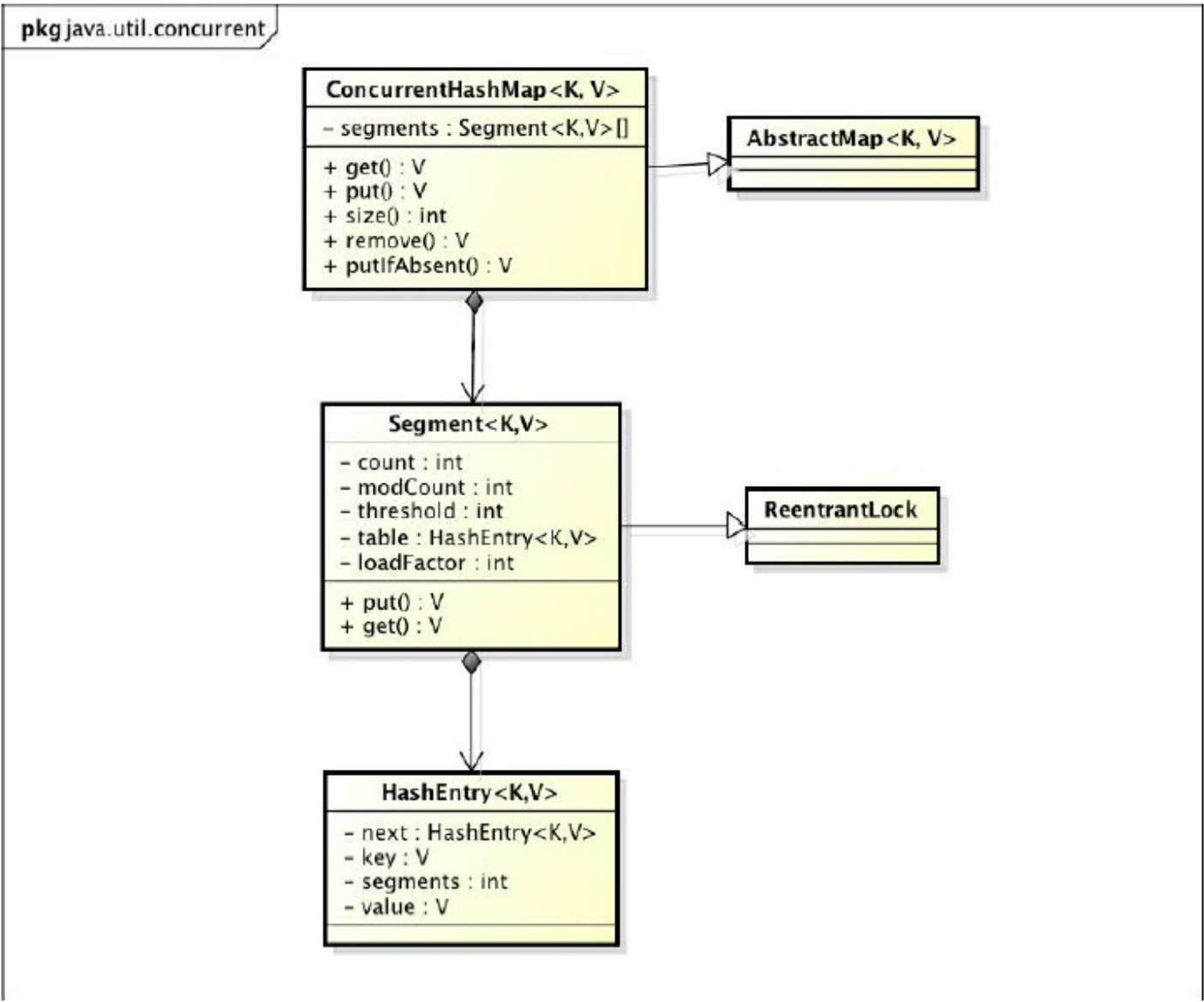


图6-1 ConcurrentHashMap的类图

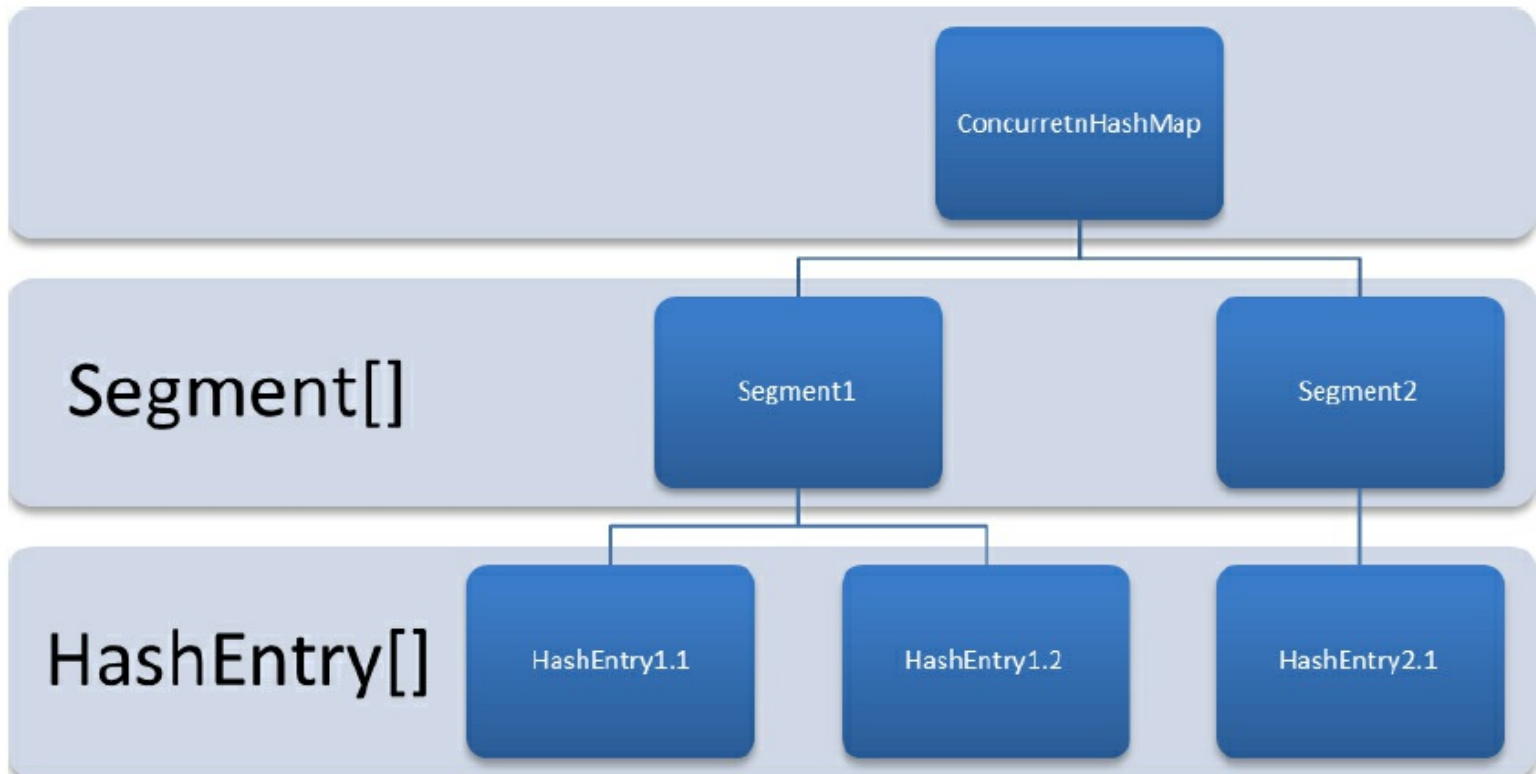


图6-2 ConcurrentHashMap的结构图

6.1.3 ConcurrentHashMap的初始化


ConcurrentHashMap初始化方法是通过initialCapacity、loadFactor和concurrencyLevel等几个参数来初始化segment数组、段偏移量segmentShift、段掩码segmentMask和每个segment里的HashEntry数组来实现的。

1.初始化segments数组

让我们来看一下初始化segments数组的源代码。

```
if (concurrencyLevel > MAX_SEGMENTS)
    concurrencyLevel = MAX_SEGMENTS;
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是通过concurrencyLevel计算得出的。为了能通过按位与的散列算法来定位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的最大值是65535，这意味着segments数组的长度最大为65536，对应的二进制是16位。

2.初始化segmentShift和segmentMask

这两个全局变量需要在定位segment时的散列算法里使用，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用于定位参与散列运算的位数，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是散列运算的掩码，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。

3.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。

6.1.4 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，必须先通过散列算法定位到Segment。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再散列。

```
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);
}
```

之所以进行再散列，目的是减少散列冲突，使元素能够均匀地分布在不同的Segment上，从而提高容器的存取效率。假如散列的质量差到极点，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。笔者做了一个测试，不通过再散列而直接执行散列计算。

```
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);
```

计算后输出的散列值全是15，通过这个例子可以发现，如果不进行再散列，散列冲突会非常严重，因为只要低位一样，无论高位是什么数，其散列值总是一样。我们再把上面的二进制数据进行再散列后结果如下(为了方便阅读，不足32位的高位补了0，每隔4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, 通过这种再散列能让数字的每一位都参加到散列运算当中, 从而减少散列冲突。ConcurrentHashMap通过以下散列算法定位segment。

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

默认情况下segmentShift为28, segmentMask为15, 再散列后的数最大是32位二进制数据, 向右无符号移动28位, 意思是让高4位参与到散列运算中, (hash>>>segmentShift) &segmentMask的运算结果分别是4、15、7和8, 可以看到散列值没有发生冲突。

6.1.5 ConcurrentHashMap的操作

本节介绍ConcurrentHashMap的3种操作——get操作、put操作和size操作。

1.get操作

Segment的get操作实现非常简单和高效。先经过一次再散列，然后使用这个散列值通过散列运算定位到Segment，再通过散列算法定位到元素，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类型，如用于统计当前Segment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。之所以不会读到过期的值，是因为根据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的散列算法虽然一样，都与数组的长度减去1再相“与”，但是相“与”的值不一样，定位Segment使用的是元素的hashCode通过再散列后得到的值的高位，而定位HashEntry直接使用使用的是再散列后的值。其目的

是避免两次散列后的值一样，虽然元素在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
```

2.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Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置，然后将其放在HashEntry数组里。

(1) 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)，如果超过阈值，则对数组进行扩容。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。

(2) 如何扩容

在扩容的时候，首先会创建一个容量是原来容量两倍的数组，然后将原数组里的元素进行再散列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。

3.size操作

如果要统计整个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，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。

6.2 ConcurrentLinkedQueue

在并发编程中，有时候需要使用线程安全的队列。如果要实现一个线程安全的队列有两种方式：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。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一下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。

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。

6.2.1 ConcurrentLinkedQueue的结构

通过ConcurrentLinkedQueue的类图来分析一下它的结构，如图6-3所示。

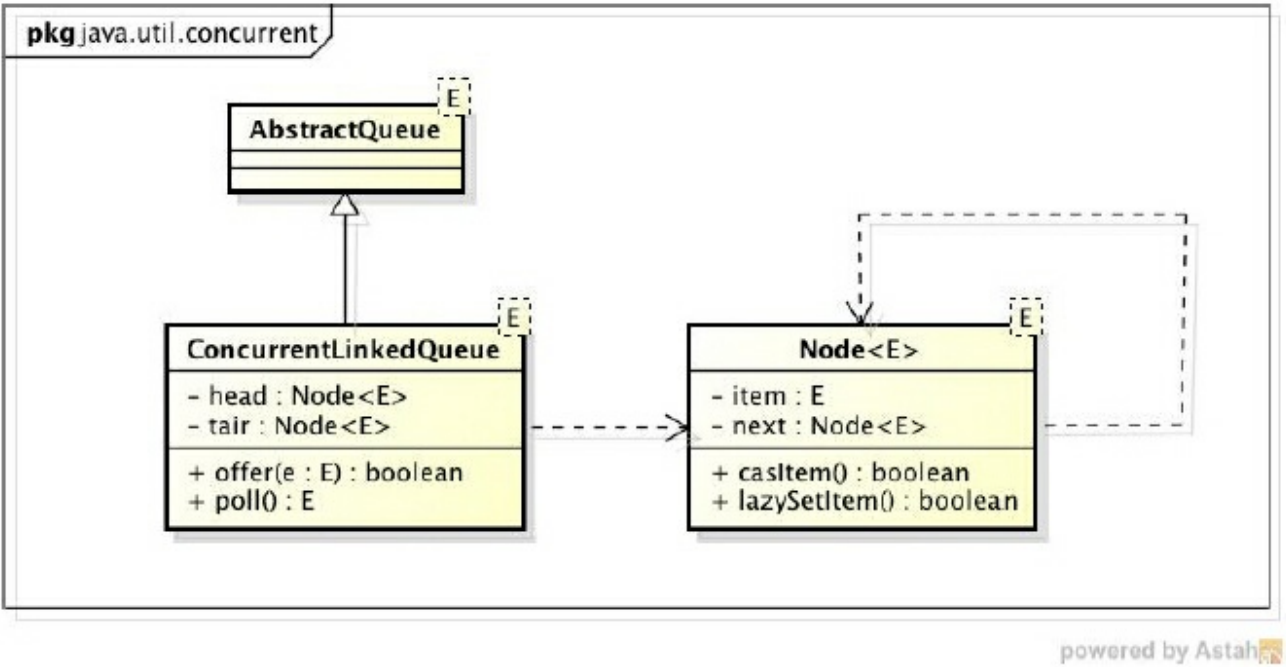


图6-3 ConcurrentLinkedQueue的类图

ConcurrentLinkedQueue由head节点和tail节点组成，每个节点(Node)由节点元素(item)和指向下一个节点(next)的引用组成，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，tail节点等于head节点。

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

6.2.2 入队列

本节将介绍入队列的相关知识。

1.入队列的过程

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tail节点的变化，这里以一个示例来展开介绍。假设我们想在一个队列中依次插入4个节点，为了帮助大家理解，每添加一个节点就做了一个队列的快照图，如图6-4所示。

图6-4所示的过程如下。

- 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。

- 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。

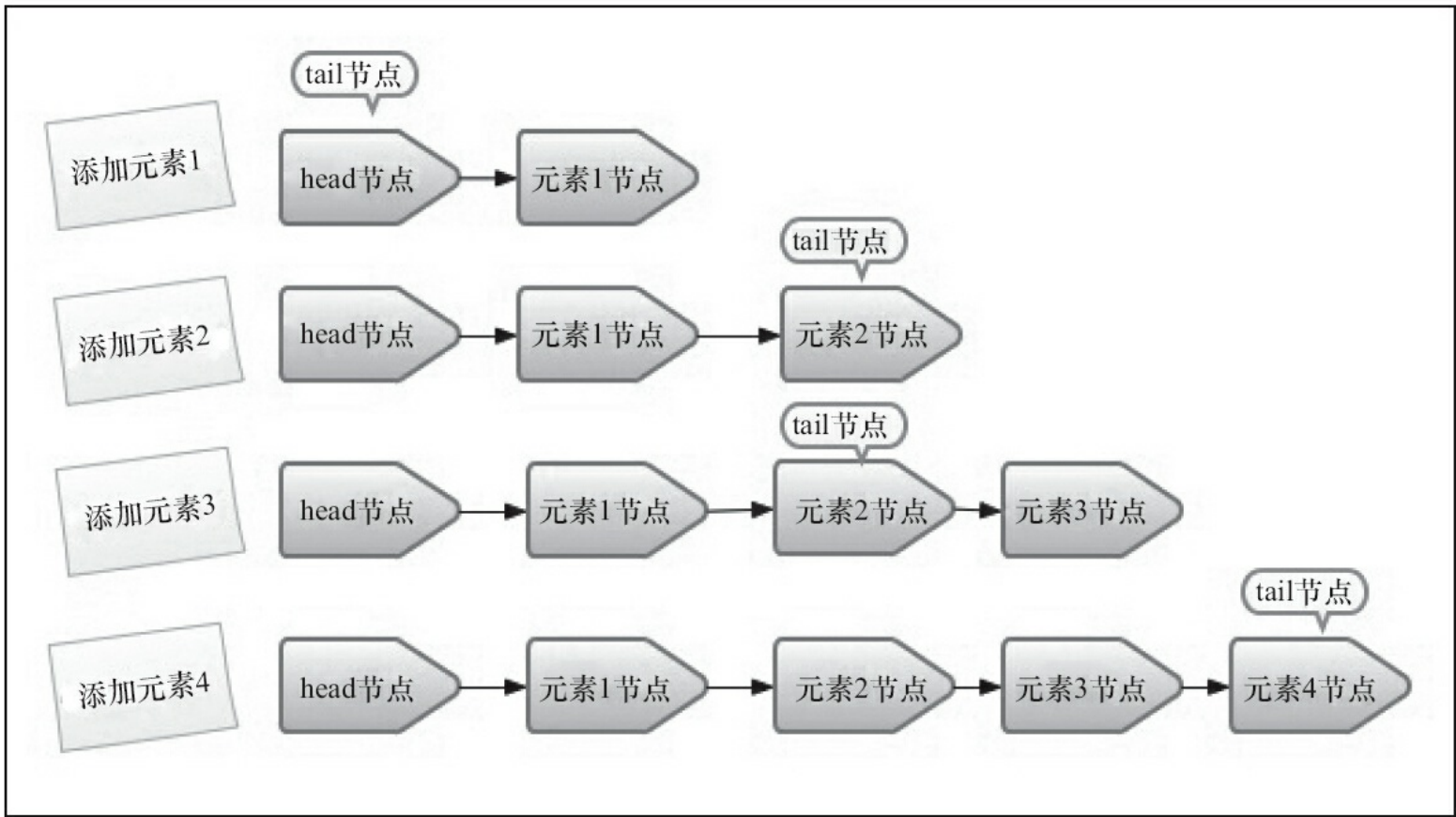


图6-4 队列添加元素的快照图

·添加元素3, 设置tail节点的next节点为元素3节点。

·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。

通过调试入队过程并观察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)。

通过对上面的分析, 我们从单线程入队的角度理解了入队过程, 但是多个线程同时进行入队的情况就变得更加复杂了, 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一下它是如何使用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tail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tail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两件事情: 第一是定位出尾节点; 第二是使用CAS算法将入队节点设置成尾节点的next节点, 如不成功则重试。

2. 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;
}
```

3.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，如果p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。

4.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是否可行？

```
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，而且逻辑清晰和易懂。但是，这么做有个缺点，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。

6.2.3 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一下head节点的变化，如图6-5所示。

从图中可知，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。

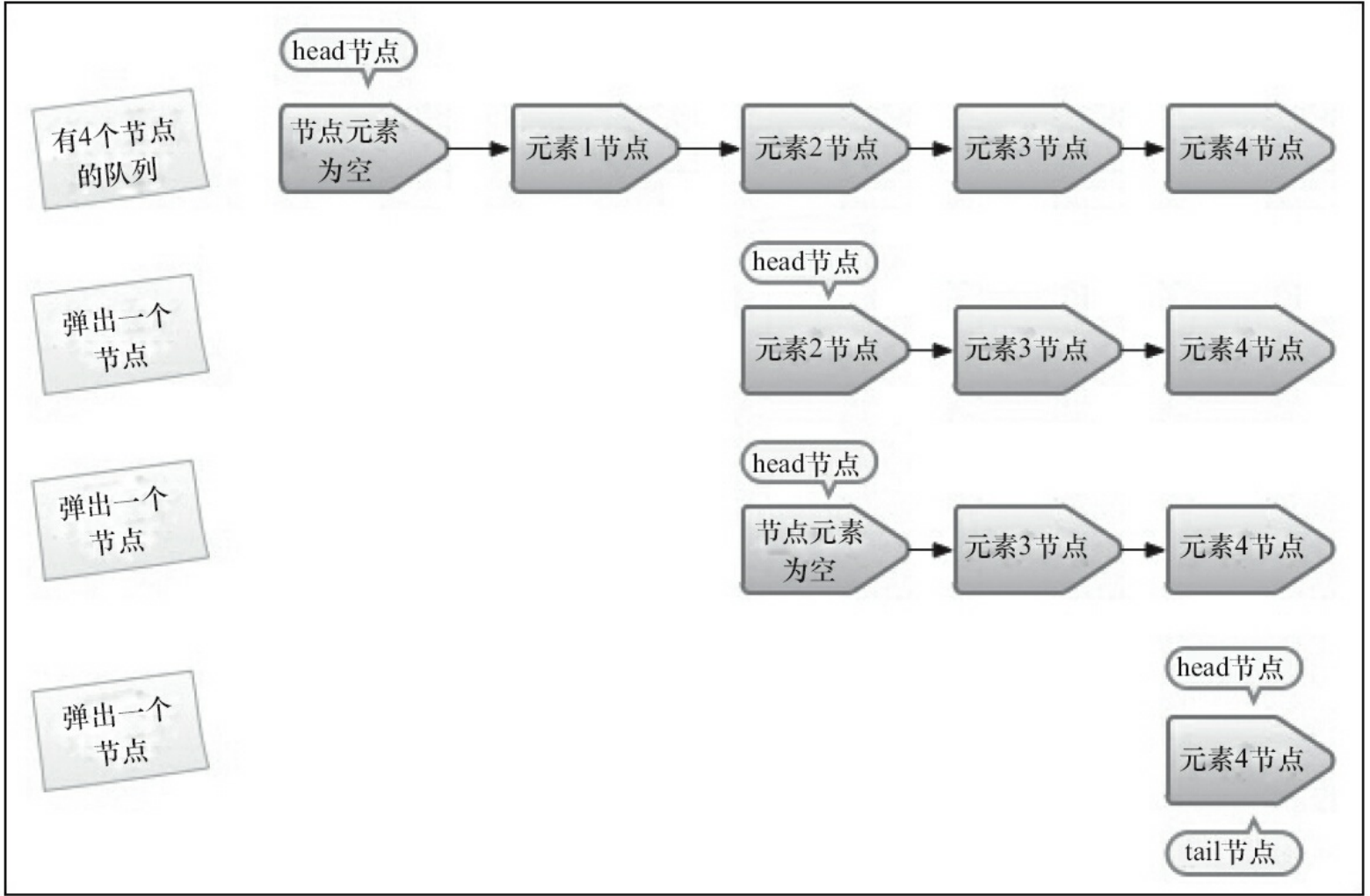


图6-5 队列出节点快照图

```
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。

6.3 Java中的阻塞队列

本节将介绍什么是阻塞队列，以及Java中阻塞队列的4种处理方式，并介绍Java 7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。

6.3.1 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,生产者是向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4种处理方式,如表6-1所示。

表6-1 插入和移除操作的4中处理方式

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,则是从队列里取出一个元素,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。

这两个附加操作的4种处理方式不方便记忆,所以我找了一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,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。

6.3.2 Java里的阻塞队列

JDK 7提供了7个阻塞队列，如下。

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

1.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();
}
```

2. 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。

3. 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。

4. 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，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

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,一共有三步。

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来标识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) {
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,单位是纳秒,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,因为实现getDelay()方法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。例如,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。

5.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。

它支持公平访问队列。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，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

6.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。

(1)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。

(2) tryTransfer方法

tryTransfer方法是用来试探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, 试图把生产者传入的元素直接传给消费者, 但是如果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。

7.LinkedBlockingDeque

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队列的两端插入和移出元素。双向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和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度膨胀。另外, 双向阻塞队列可以运用在“工作窃取”模式中。

6.3.3 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，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，如何让生产者和消费者进行高效率的通信呢？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();  
}
```

当往队列里插入一个元素时，如果队列不可用，那么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先保存一下将要阻塞的线程，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4种情况中的一种发生时，该方法才会返回。

·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park

的情况。

- 线程被中断时。
- 等待完time参数指定的毫秒数时。
- 异常现象发生时, 这个异常现象没有任何原因。

继续看一下JVM是如何实现park方法:park在不同的操作系统中使用不同的方式实现, 在Linux下使用的是系统方法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方法在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, 如下。

```
"main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting on condition [0x000000010
  java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
      - parking to wait for  <0x0000000140559fe8> (a java.util.concurrent.lock
AbstractQueuedSynchronizer$ConditionObject)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject
await(AbstractQueuedSynchronizer.java:2043)
    at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:3
    at blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQueueTest.java
```

6.4 Fork/Join框架

本节将会介绍Fork/Join框架的基本原理、算法、设计方式、应用与实现等。

6.4.1 什么是Fork/Join框架

Fork/Join框架是Java 7提供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这两个单词来理解一下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 $1+2+\dots+10000$ ，可以分割成10个子任务，每个子任务分别对1000个数进行求和，最终汇总这10个子任务的结果。Fork/Join的运行流程如图6-6所示。

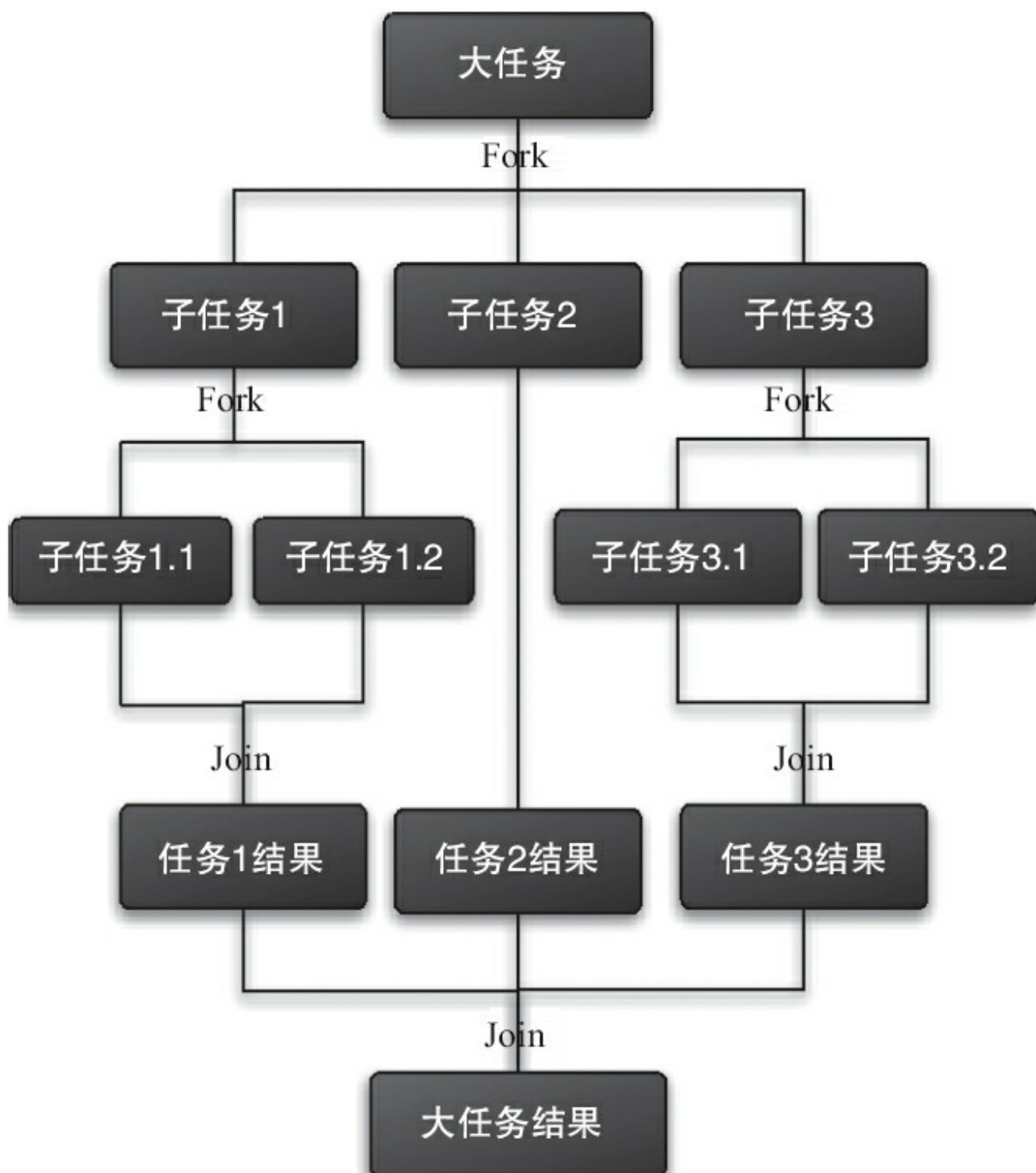


图6-6 Fork Join的运行流程图

6.4.2 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。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，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，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。

工作窃取的运行流程如图6-7所示。

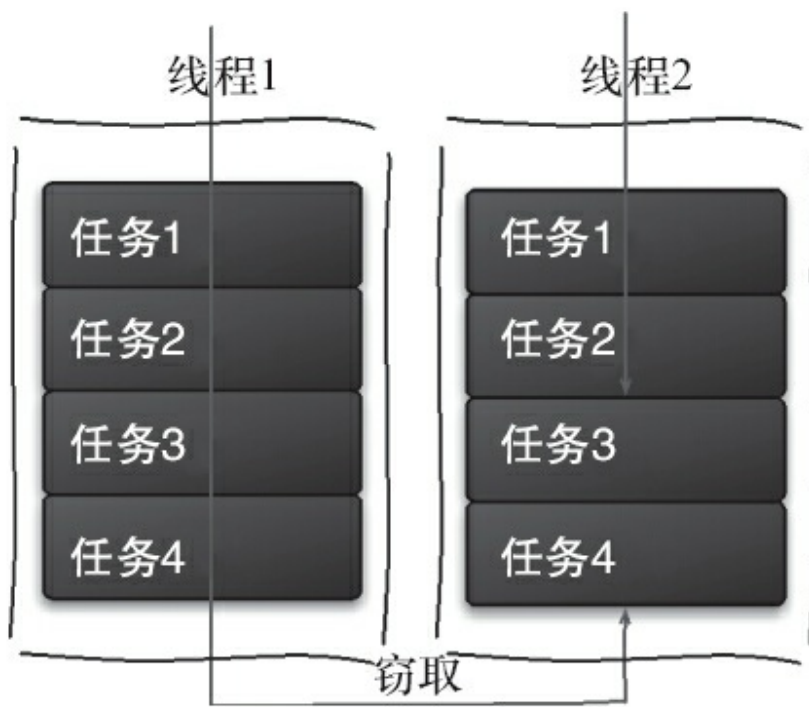


图6-7 工作窃取运行流程图

工作窃取算法的优点:充分利用线程进行并行计算,减少了线程间的竞争。

工作窃取算法的缺点: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。并且该算法会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。

6.4.3 Fork/Join框架的设计

我们已经很清楚Fork/Join框架的需求了，那么可以思考一下，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，所以还需要不停地分割，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。

①ForkJoinTask: 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，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。

②ForkJoinPool: 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。

6.4.4 使用Fork/Join框架

让我们通过一个简单的需求来使用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, 我们进一步了解ForkJoinTask, ForkJoinTask与一般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子任务, 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。

6.4.5 Fork/Join框架的异常处理

ForkJoinTask在运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。

6.4.6 Fork/Join框架的实现原理

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成, ForkJoinTask数组负责将存放程序提交给ForkJoinPool的任务, 而ForkJoinWorkerThread数组负责执行这些任务。

(1)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方法异步地执行这个任务, 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数组队列里。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();
    }
}
```

(2)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。让我们一起来看看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4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。

让我们再来分析一下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成, 如果执行完成, 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, 则设置任务状态为NORMAL, 如果出现异常, 则记录异常, 并将任务状态设置为EXCEPTIONAL。

6.5 本章小结

本章介绍了Java中提供的各种并发容器和框架，并分析了该容器和框架的实现原理，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，并在以后开发的

并发程序时，运用上这些并发编程的技巧。