**1.ConcurrentHashMap的实现原理与使用**

ConcurrentHashMap是线程安全且高效的HashMap。

**1.1为什么要使用ConcurrentHashMap**

并发编程中使用HashMap可能导致程序死循环（**1.8解决了扩容和put成环**），可能使的对HashMap的数据操作出现未知的结果，而使用线程安全的Hashtable效率又非常低下，基于以上原因，才有了ConcurrentHashMap。

①线程不安全的HashMap，在并发执行put操作的时候，会引起数据出错，例如++size的过程中，其他线程也进行插入，++size，就有可能导致插入两个，只加一次

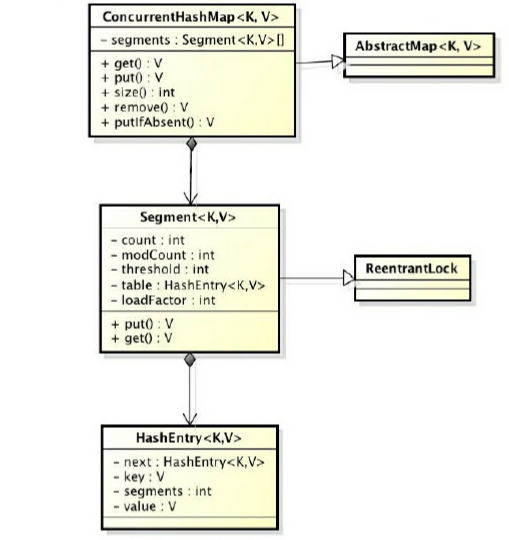
②效率低下Hashtable

Hashtable使用synchronized保证线程安全，但是在线程竞争激烈的情况下，一旦某个线程使用put进行添加元素，其他线程不能使用put方法，也不能使用get方法获取元素。

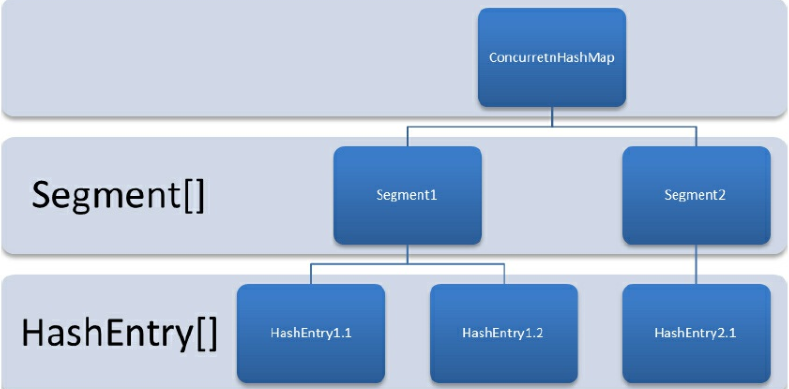
③ConcurrentHashMap锁分段技术可有效提高并发访问率

假如容器里有多把锁，每一把锁用于锁容器里的一部分数据，那么多线程访问容器里的不同数据段的数据时线程间就不会存在锁竞争，这就是锁分段。将数据分成一段一段存储，然后给每一段数据配一把锁，当一个线程占用锁，访问其中一个数据段的时候，其他段的数据能被其他线程访问。

**1.2ConcurrentHashMap的结构1.7**



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**



**1.3ConcurrenthashMap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**

|  |
| --- |
| 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); |

和concurrencyLevel相关，由于为了与hashmap一样计算index，因此每个segments数组的长度也要是2^n，所以ssize必定是等于或者大于concurrencyLevel的2的n次幂的数。容器里的锁的个数也为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。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。

|  |
| --- |
| 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

**1.4 定位segment**

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);  } |

减少hash散列冲突，使元素能够均匀的分布在不同的segment上，提高存储效率。

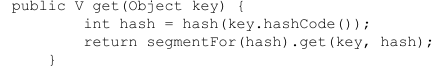
通过下面的散列算法定位segment

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

**1.5 ConcurrentHashMap的操作**

①get操作

先经过再散列，然后使用这个散列值通过散列运算定位到segment，通过散列算法定位到元素



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。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**Segement大小的count字段**和**用于存储值的HashEntry的value**。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。



虽然两次都是用长度值减1，但是定位segment使用了高位，而hashentry直接使用



②put操作

Put方法首先定位到segment，然后在segment里进行插入操作。两个步骤：判断是否需要segment里的HashEntry数组进行扩容，第二部定位添加元素的位置，然后放到hashentry里面

1.是否需要扩容

首先判断hashentry数组长度是否超过threshold，如果超过阈值，就对数组进行扩容。

2.如何扩容

扩容的时候，首先会创建一个容量是原来两倍的数组，然后将原数组的元素再hash之后存到新的数组里，不会对整个容器进行扩容，而只是对某个segment进行扩容。

③size操作

统计所有segment里的数组元素个数求和。Segment里面的全局变量是一个volatile变量，那么在多线程场景下，并不能简单进行count的求和。ConcurrentHashMap先尝试两次通过不锁住segment的方式来统计各个segment的大小，如果统计过程中，count发生了变化，则采用加锁方式统计所有segment大小。使用modCount变量，在put、remove、clean方法操作元素前都会对modcount进行加1，那么只要统计size前后比较modcount是否发生变化，就可以知道容器大小是否发生变化。

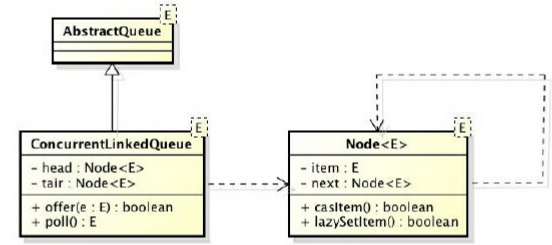
**2.ConcurrentLinkedQueue**

实现线程安全的队列，一种是使用阻塞算法，另一种是使用非阻塞算法。

阻塞算法可以用一个锁或者两个锁等方式实现。非阻塞实心方式，则可以使用循环CAS的方式实现。

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进行排序，当我们添加一个元素的时候，他会添加到队列的尾部；当我们获取一个元素的时候，他会返回队列头部的元素。**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。**

**2.1 ConcurrentLinkedQueue的结构**



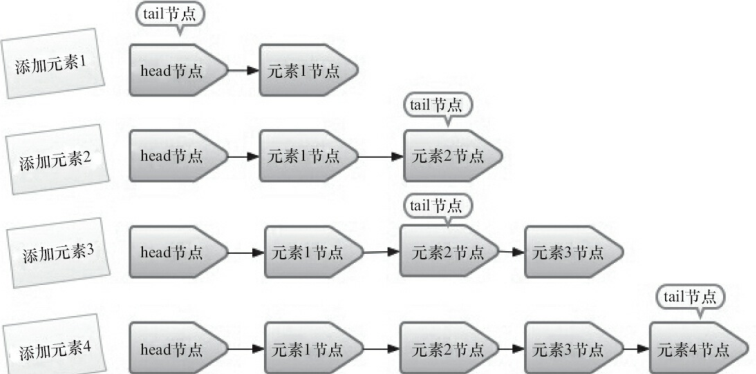
ConcurrentLinkedQueue由head节点和tail节点组成，每个节点（Node）由节点元素（item）和指向下一个节点（next）的引用组成，节点与节点之间就是通过这个next关联起来，从而组成一张链表结构的队列。默认情况下head节点存储的元素为空，tail节点等于head节点。

**private** **transient** **volatile** Node<E> head;

**private** **transient** **volatile** Node<E> tail;

**2.2 入队列**

①入队列就是将入队节点添加到队列的尾部。

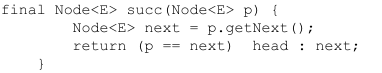


|  |
| --- |
| **public boolean offer(E e) {**  **if (e == null) throw new NullPointerException();**  **Node<E> n = new Node<E>(e);**  **retry:**  **for (;;) {**  **Node<E> t = tail;**  **Node<E> p = t;**  **for (int hops = 0; ; hops++) {**  **Node<E> next = succ(p); // 1.获取p的后继节点。(如果p的next指向自身，返回head节点)**  **if (next != null) { // 2.如果next不为null**  **if (hops > HOPS && t != tail)**  **continue retry; // 3.如果自旋次数大于HOPS，且t不是尾节点，跳出2层循环重试。**  **p = next; // 4.如果自旋字数小于HOPS或者t是尾节点，将p指向next。**  **} else if (p.casNext(null, n)) { // 5.如果next为null，尝试将p的next节点设置为n，然后自旋。**  **if (hops >= HOPS)**  **casTail(t, n); // 6.如果设置成功且自旋次数大于HOPS，尝试将n设置为尾节点，失败也没关系。**  **return true; // 7.添加成功。**  **} else {**  **p = succ(p); // 8。如果第5步尝试将p的next节点设置为n失败，那么将p指向p的后继节点，然后自旋。**  **}**  **}**  **}**  **}**  **final Node<E> succ(Node<E> p) {**  **Node<E> next = p.getNext();**  **//如果p节点的next节点指向自身，那么返回head节点；否则返回p的next节点。**  **return (p == next) ? head : next;**  **}**  **/\*\***  **\* 允许头尾节点的指针滞后，所以当头尾节点离"实际位置"的距离**  **\* (按节点)小于HOPS时，不会去更新头尾指针。这里是假设volatile写代价比volatile读高。**  **\*/**  **private static final int HOPS = 1;** |

1定位出尾节点2使用cas算法将入队节点设置成尾节点的next节点，不成功就重试

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。



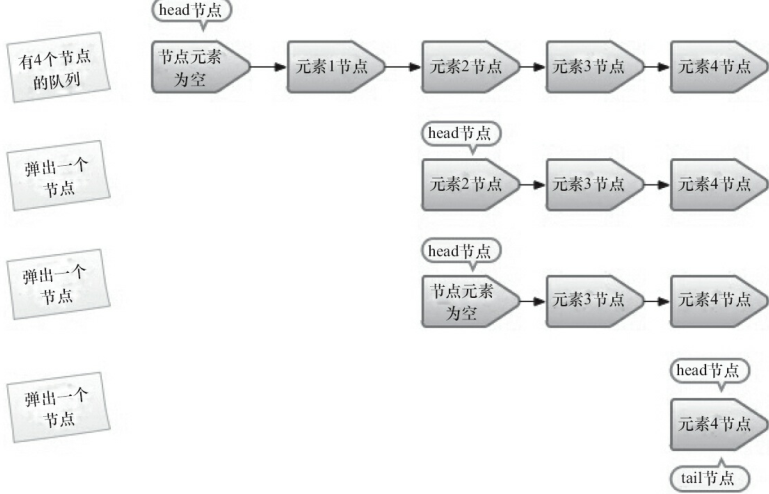
③设置入队节点为尾节点

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的next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。

④HOPS的设计意图

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。**

**2.3出队列**



|  |
| --- |
| public E poll() {  Node<E> h = head;  Node<E> p = h;  for (int hops = 0; ; hops++) {  E item = p.getItem(); // 1.获取p节点上的元素item。  if (item != null && p.casItem(item, null)) { // 2.如果item不为null，尝试将p的item设置为null。  if (hops >= HOPS) {  Node<E> q = p.getNext();  updateHead(h, (q != null) ? q : p); // 3.如果自旋次数大于HOPS，尝试更新头节点。  }  return item; // 4.获取元素成功。  }  **// 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); // 5.获取p的后继节点。(如果p的next指向自身，那么返回head节点)  if (next == null) {  updateHead(h, p); // 6.如果p的后继节点为null，尝试将p设置为头节点，然后跳出循环。  break;  }  p = next;  }  return null; // 7.从第6步过来，没有成功获取元素，返回null。  }    final void updateHead(Node<E> h, Node<E> p) {  if (h != p && casHead(h, p))  h.lazySetNext(h); //将h的next指向自身，帮助GC。  } |

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**。

**3.java中的阻塞队列**

**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。这两个附加的操作是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**。

阻塞队列不可用时，这两个附加操作提供了四种处理方式：



抛异常：add remove element；队列满时，再插入，与 队列空时再取数据，都会抛异常

返回特殊值：offer，poll，插入成功返回true，移除成功返回元素值。

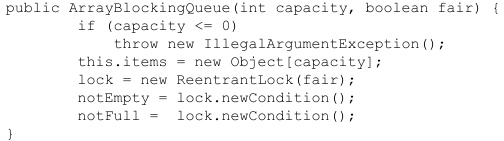
一直阻塞：put，take队列满，插入或者队列空删除，都会阻塞。知道不满或者非空

超时退出：offer（time，unit），poll（time，unit）队列满插入或者队列空删除，阻塞指定时间，然后退出。

**3.2 Java里的阻塞队列**

**①ArrayBlockingQueue**

用数组实现的有界阻塞队列。此队列按照先进先出的原则对元素进行排序。默认不保证公平性；当然也可以用可重入锁实现公平性



**②LinkedBlockingQueue**

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.按照先进先出的规则排序

**③PriorityBlockingQueue**

支持优先级的无界阻塞队列，默认情况下采用自然排序升序，使用堆实现，不能保证同优先级的顺序。

**④DelayQueue**

支持延时获取元素的无界阻塞队列，队列使用PriorityQueue来实现，队列中元素实现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。只有在延时到的时候才能取出队列数据

运用场景：

**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

第一步：创建对象的时候，初始化基本数据，使用time记录当前对象延迟到什么时候可使用，使用sequenceNumber来标识元素在队列中的先后顺序。

第二步，实现getDelay方法，该方法返回当前元素还需要延时多少时间，单位纳秒，

第三步，实现compareTo方法来指定元素的顺序，让延时时间最长的放在对列末尾

|  |
| --- |
| package cn.huangwei.sixth;  import java.util.Date;  import java.util.concurrent.DelayQueue;  public class DelayTask implements Runnable {  private int id;  private DelayQueue<DelayEvent> queue;    public DelayTask(int id, DelayQueue<DelayEvent> queue) {  super();  this.id = id;  this.queue = queue;  }  @Override  public void run() {  // TODO Auto-generated method stub  Date now = new Date();  Date delay = new Date();  delay.setTime(now.getTime() + id \* 5000);  System.out.println("Thread " + id + " " + delay);  DelayEvent delayEvent = new DelayEvent(id + "", delay);  queue.add(delayEvent);  }  }  **package** cn.huangwei.sixth;  **import** java.util.Date;  **import** java.util.concurrent.Delayed;  **import** java.util.concurrent.TimeUnit;  **public** **class** DelayEvent **implements** Delayed{  **private** String name;  **private** Date removeTime;    **public** DelayEvent(String name, Date removeTime){  **super**();  **this**.name = name;  **this**.removeTime = removeTime;  }    @Override  /\*  \* DelayQueue内部使用PriorityQueue，即用了堆结构，此比较方法的result表明，this与o对象剩余的时间大小  \* 剩余时间小的放到堆顶，在DelayQueue中poll方法只有在getDelay小于等于0的时候才会调用poll方法，  \* 从而实现缓存的效果。对象只能在缓存中存活一定时间，时间到了就要被清除出去  \*/  **public** **int** compareTo(Delayed o) {  **long** result = **this**.getDelay(TimeUnit.***MILLISECONDS***)  - o.getDelay(TimeUnit.***MILLISECONDS***);  **if**(result < 0){  **return** -1;  }**else** **if**(result > 0){  **return** 1;  }**else**{  **return** 0;  }  }  @Override  **public** **long** getDelay(TimeUnit unit) {  Date now = **new** Date();  **long** diff = removeTime.getTime() - now.getTime();  **return** unit.convert(diff, TimeUnit.***MILLISECONDS***);  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  }  **package** cn.huangwei.sixth;  **import** java.util.Date;  **import** java.util.concurrent.DelayQueue;  **import** java.util.concurrent.TimeUnit;  **public** **class** DelayedQueueCache {  **public** **static** **void** main(String[] args) **throws** Exception {  DelayQueue<DelayEvent> queue = **new** DelayQueue<DelayEvent>();  Thread[] ts = **new** Thread[5];  **for**(**int** i = 0; i < ts.length; i++){  DelayTask task = **new** DelayTask(i + 1, queue);  ts[i] = **new** Thread(task);  }  **for**(**int** i = 0; i < ts.length; i++){  ts[i].start();  }    **for** (**int** i = 0; i < ts.length; i++) {  **try** {  ts[i].join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **do** {  DelayEvent delayEvent;  **do** {  delayEvent = queue.poll();  **if** (delayEvent != **null**) {  System.***out***.println("At " + **new** Date() + " you have read " + delayEvent.getName()+ " event");  }**else**{  System.***out***.println("no data");  }  } **while** (delayEvent != **null**);  TimeUnit.***MILLISECONDS***.sleep(500);  } **while** (queue.size() > 0);  }  } |

**⑤SynchronizedQueue**

SynchronizedQueue是一个不存储元素的阻塞队列，每个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();

}

队列本身并不存储任何元素，非常适合传递性场景.

**⑥LinkedTransferQueue**

由链表组成的无界阻塞TransferQueue队列。多了tryTransfer和transfer方法

1.transfer方法

如果有消费者等待接收元素（消费者使用take方法）时，transfer可以把生产者传入的元素立刻transfer给消费者。如果没有消费者等待接受元素，transfer方法会将元素存放在队列的tail节点，等到消费者接收的时候在返回。



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

是用来试探生产者传入的元素能否直接传给消费者，如果消费者等待接收元素，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**。

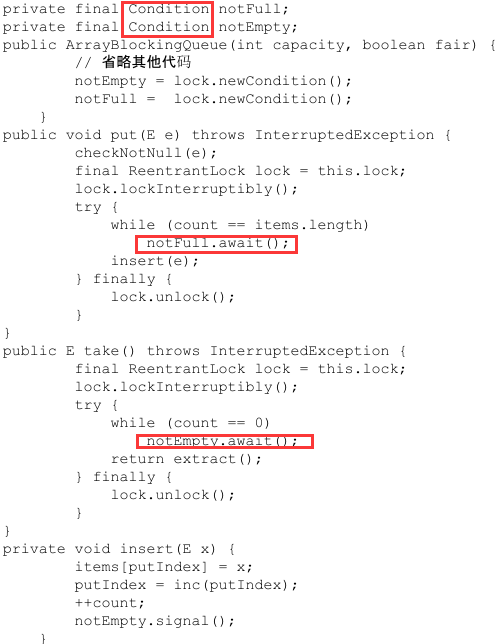
**⑦LinkedBlockingDeque**

由链表结构组成的双向阻塞队列。可以从队列的两端插入和移除元素，含有first与last相关的方法。

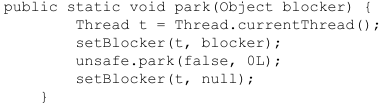
**3.3 阻塞队列的实现原理**

①使用通知模式实现

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。



Await方法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。park（this）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。



**Park这个方法返回的情况**：

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park**的情况。

②线程被**中断**时。

③等待**完time参数指定的毫秒数**时。

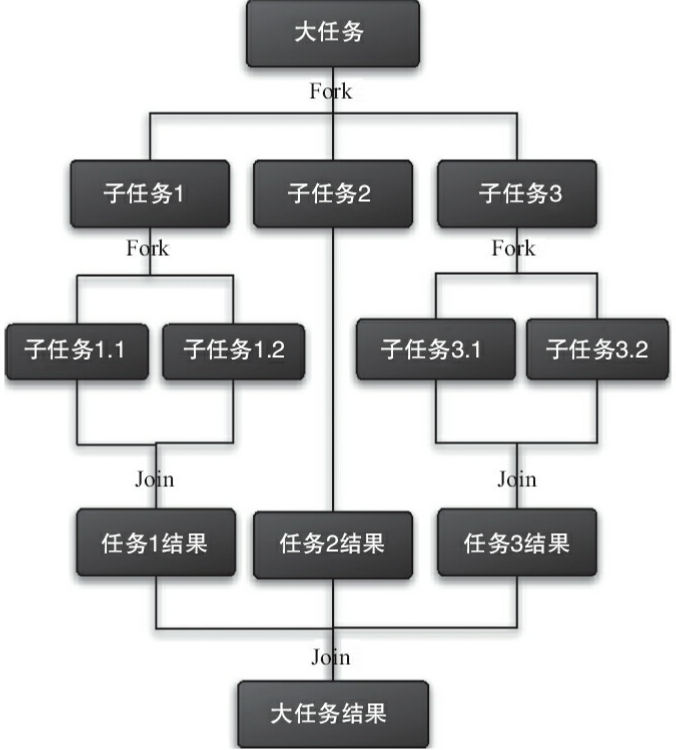
④**异常现象**发生时，这个异常现象没有任何原因。

**4.fork/join框架**

**4.1什么是fork/join框架**

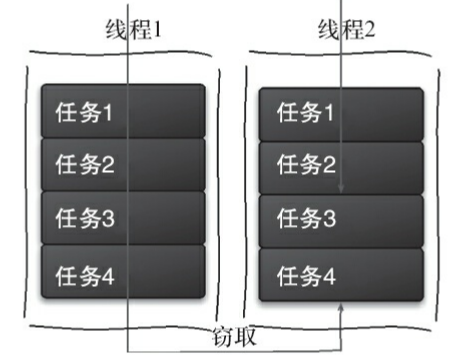
一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

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。



**4.2工作窃取算法**

某个线程从其他队列里窃取任务来执行；一个任务被分为若干个互不依赖的子任务，把这些子任务放到不同队列中，为每个队列创建一个单独的线程执行任务，线程和队列一一对应。如果某个线程的任务已做完，就会去执行其他线程的任务，采用双端队列，一个从头取，一个从尾取。



**4.3 fork/join框架的设计**

步骤1 分割任务，fork类来把大人物分割成子任务，有可能子任务还是很大，还需要不停分割，直到分割出的子任务足够小

步骤2 执行任务并合并结果，分割的任务都放在双端队列里，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中，启动一个线程，从队列里拿数据，并合并

Fork/join使用两个类完成以上事情。

①ForkJoinTask：创建一个ForkJoin任务，它提供fork和join操作机制，通常情况下，不需要直接继承ForkJoinTask，只需要继承其子类，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。

**4.4使用Fork/Join框架**

使用Fork/Join框架，需求是：计算1+2+3+4的结果

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** cn.huangwei.sixth;  **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**(Exception e){  e.printStackTrace();  }  }  } |

主要方法是compute，首先的判断任务是否足够小，如果足够小，就执行任务，如果不够小，就分割成两个子任务，每个子任务在**调用fork方法**时，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；如果不需要分割，执行当前子任务并返回，使用join方法会等待子任务执行完毕得到其结果。

**4.5 fork/Join框架的异常处理**

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；

**if(task.isCompletedAbnormally())**

**{**

**System.out.println(task.getException());**

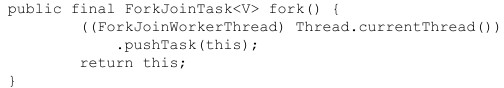
**}**

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返回null

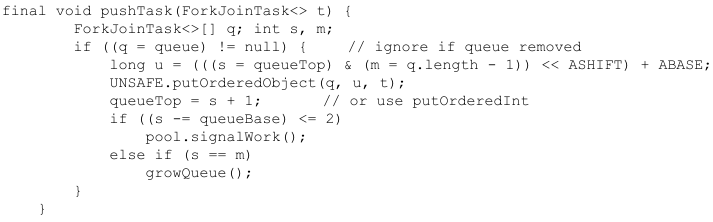
**4.6 Fork/join框架实现原理**

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。**

①ForkJoinTask中的fork方法，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。

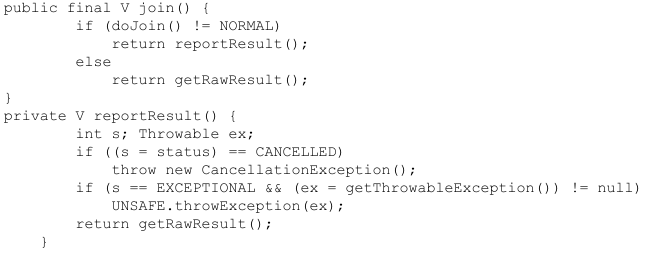


pushTask方法把当前任务存放在ForkJoinTask数组队列里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。



②ForkJoinTask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**。



调用doJoin方法，得到当前任务的状态，任务状态有4种：已完成NORMAL,被取消(CANCELLED),信号（等待被唤醒）SIGNAL, ,出现异常（EXCEPTIONAL）

1. 如果状态已完成，直接返回任务结果
2. 状态 被取消，直接抛出cancellationException
3. 状态抛出异常，直接抛出对应的异常。

|  |
| --- |
| 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。