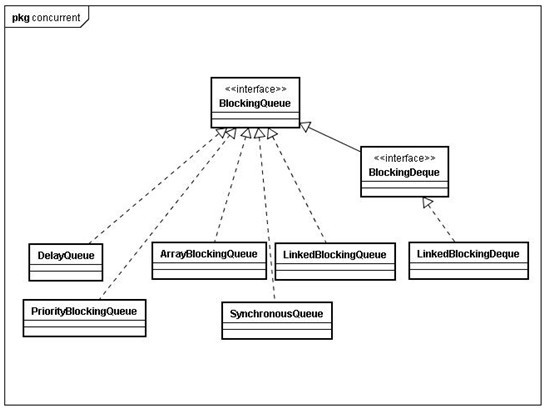
1. java.util.concurrent所提供的并发容器  
java.util.concurrent提供了多种并发容器，总体上来说有4类，队列类型的BlockingQueue和 ConcurrentLinkedQueue,Map类型的ConcurrentMap,Set类型的ConcurrentSkipListSet和CopyOnWriteArraySet,List类型的CopyOnWriteArrayList.

这些并发容器都采用了多种手段控制并发的存取操作，并且尽可能减小控制并发所带来的性能损耗。接下来我们会对每一种类型的实现类进行代码分析，进而得到java.util.con current包所提供的并发容器在传统容器上所做的工作。

2. BlockingQueue  
BlockingQueue接口定义的所有方法实现都是线程安全的，它的实现类里面都会用锁和其他控制并发的手段保证这种线程安全，但是这些类同时也实现了Collection接口(主要是AbstractQueue实现)，所以会出现BlockingQueue的实现类也能同时使用Conllection接口方法，而这时会出现的问题就是像addAll,containsAll,retainAll和removeAll这类批量方法的实现不保证线程安全，举个例子就是addAll 10个items到一个ArrayBlockingQueue,可能中途失败但是却有几个item已经被放进这个队列里面了。

下面我们根据这幅类图来逐个解析不同实现类的特性和特性实现代码  
  
DelayQueue提供了一个只返回超时元素的阻塞队列，也就是说，即使队列中已经有数据了，但是poll或者take的时候还要判定这个element有没达到规定的超时时间,poll方法在element还没达到规定的超时时间返回null,take则会通过condition.waitNanos()进入等待状态。一般存储的element类型为Delayed，这个接口JDK中实现的类有ScheduledFutureTask,而DelayQueue为DelayedWorkQueue的Task容器，后者是ScheduledThreadPoolExecutor的工作队列，所以DelayQueue所具有的超时提供元素和线程安全特性对于并发的定时任务有很大的意义。

|  |
| --- |
| public E take() throws InterruptedException { |
| final ReentrantLock lock = this.lock; |

|  |
| --- |
| //控制并发 |
| lock.lockInterruptibly(); | |

|  |
| --- |
| try { |
| for (;;) { | |

|  |
| --- |
| E first = q.peek(); |
| if (first == null) { | |

|  |  |
| --- | --- |
| //condition协调队列里面元素 | |
| available.await(); |

|  |
| --- |
| } else { |
| long delay =  first.getDelay(TimeUnit.NANOSECONDS); | |

|  |
| --- |
| if (delay > 0) { |
| //因为first在队列里面的delay最短的(优先队列保证),所以wait这个时间那么队列中最短delay的元素就超时了.即 | |

|  |
| --- |
| //队列有元素供应了. |
| long tl = available.awaitNanos(delay); | |

|  |
| --- |
| } else { |
| E x = q.poll(); | |

|  |
| --- |
| assert x != null; |
| if (q.size() != 0) | |

|  |  |
| --- | --- |
| available.signalAll(); // wake up other takers | |
| return x; |

|  |
| --- |
|  |
| } | |

|  |  |
| --- | --- |
| } | |
| } |

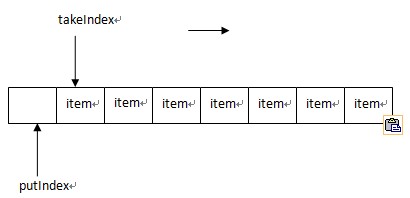
|  |
| --- |
| } finally { |
| lock.unlock(); | |

|  |  |
| --- | --- |
| } | |
| } |

DelayQueue的内部数据结构是PriorityQueue,因为Delayed接口同时继承了Comparable接口，并且Delayed的实现类对于这个compareTo方法的实现是基于超时时间进行大小比较，所以DelayQueue无需关心数据的排序问题，只需要做好存取的并发控制（ReetranLock）和超时判定即可。另外，DelayQueue有一个实现细节就是通过一个Condition来协调队列中是否有数据可以提供，这对于take和带有提取超时时间的poll是有意义的（生产者，消费者的实现）。

PriorityBlockingQueue实现对于外部而言是按照元素的某种顺序返回元素，同时对存取提供并发保护(ReetranLock),使用Condition协调队列是否有新元素提供。PriorityBlocking Queue内部的数据结构为PriorityQueue,优先级排序工作交给PriorityQueue，至于怎么排序，需要根据插入元素的Comparable的接口实现，和DelayQueue比起来，它没有限定死插入数据的Comparable实现，而DelayQueue的元素实现Comparable必须按照超时时间的长短进行比较，否则DelayQueue返回的元素就很可能是错误的。

ArrayBlockingQueue是一个先入先出的队列，内部数据结构为一个数组，并且一旦创建这个队列的长度是不可改变的，当然put数据时，这个队列也不会自动增长。ArrayBlockingQueue也是使用ReetranLock来保证存取的原子性，不过使用了notEmpty和notFull两个Condition来协调队列为空和队列为满的状态转换，插入数据的时候，判定当前内部数据结构数组E[] items的长度是否等于元素计数，如果相等，说明队列满，notFull.await()，直到items数组重新不为满（removeAt,poll等），插入数据后notEmpty.sinal()通知所有取数据或者移除数据并且因为items为空而等待的线程可以继续进行操作了。提取数据或者移除数据的过程刚好相反。

ArrayBlockingQueue使用三个数字来维护队列里面的数据变更,包括takeIndex,putIndex,count,这里需要讲一下 takeIndex和putIndex,其中takeIndex指向下一个能够被提取的元素,而putIndex指向下一个能够插入数据的位置,实现类似下图的结构,当takeIndex移到内部数组items最大长度时,重新赋值为0,也就是回到数组头部,putIndex也是相同的策略.  


|  |
| --- |
| /\*\* |
| \* 循环增加putIndex和takeIndex,如果到数组尾部,那么 | |

|  |  |
| --- | --- |
| \* 置为0 | |
| \*/ |

|  |
| --- |
| final int inc(int i) { |
| return (++i == items.length)? 0 : i; | |

|  |
| --- |
| } |
|  |

|  |
| --- |
| //\*\* |
| \* 插入一个item,需要执行线程获得了锁 | |

|  |
| --- |
| \*/ |
| private void insert(E x) { | |

|  |
| --- |
| items[putIndex] = x; |
| //累加putIndex,可能到数组尾部,那么重新指向0位置 | |

|  |  |
| --- | --- |
| putIndex = inc(putIndex); | |
| ++count; |

|  |  |
| --- | --- |
| //put后,使用Condition通知正在等待take的线程可以做提取操作 | |
| notEmpty.signal(); |

|  |
| --- |
| } |
|  |

|  |
| --- |
| /\*\* |
| \* 获取一个元素,执行这个操作的前提是线程已经获得锁, | |

|  |  |
| --- | --- |
| \* 内部调用 | |
| \*/ |

|  |
| --- |
| private E extract() { |
| final E[] items = this.items; | |

|  |
| --- |
| E x = items[takeIndex]; |
| items[takeIndex] = null; | |

|  |  |
| --- | --- |
| //累加takeIndex,有可能到数组尾部,重新调到数组头部 | |
| takeIndex = inc(takeIndex); |

|  |
| --- |
| --count; |
| //take后,使用Condition通知正在等待插入的线程可以插入 | |

|  |  |
| --- | --- |
| notFull.signal(); | |
| return x; |

|  |
| --- |
| } |

这里需要解释下Condition的实现，Condition现在的JDK实现只有AQS的ConditionObject，并且通过ReetranLock的newConditon()方法暴露出来，这是因为Condition的await()或者sinal()一般在lock.lock()与lock.unlock()之间执行，当执行condition.await()方法时，它会首先释放掉本线程持有的锁，然后自己进入等待队列，直到sinal(),唤醒后又会重新去试图拿到锁,拿到后执行await下方的代码,其中释放当前锁和得到当前锁都需要ReetranLock的tryAcquire（int args）方法来判定，并且享受ReetranLock的重进入特性。

|  |  |
| --- | --- |
| public final void await() throws InterruptedException { | |
| if (Thread.interrupted()) |

|  |  |
| --- | --- |
| throw new InterruptedException(); | |
| //加一个新的condition等待节点 |

|  |  |
| --- | --- |
| Node node = addConditionWaiter(); | |
| //释放自己占用的锁 |

|  |  |
| --- | --- |
| int savedState = fullyRelease(node); | |
| int interruptMode = 0; |

|  |
| --- |
| while (!isOnSyncQueue(node)) { |
| //如果当前线程等待状态是CONDITION,park住当前线程,等待condition的signal来解除 | |

|  |
| --- |
| LockSupport.park(this); |
| if ((interruptMode =checkInterruptWhileWaiting(node)) != 0) | |

|  |  |
| --- | --- |
| break; | |
| } |

|  |  |
| --- | --- |
| if (acquireQueued(node, savedState) && interruptMode != THROW\_IE) | |
| interruptMode = REINTERRUPT; |

|  |
| --- |
| if (node.nextWaiter != null) |
| unlinkCancelledWaiters(); | |

|  |
| --- |
| if (interruptMode != 0) |
| reportInterruptAfterWait(interruptMode); | |

|  |
| --- |
| } |

LinkedBlockingQueue是一个链表结构构成的队列，并且节点是单向的，也就是只有next,没有prev,可以设置容量，如果不设置，最大容量为Integer.MAX\_VALUE，队列只持有头结点和尾节点以及元素数量，通过putLock和takeLock两个ReetranLock分别控制存和取的并发,但是remove,toArray,toString,clear, drainTo以及迭代器等操作会同时取得putLock和takeLock，并且同时lock,此时存或者取操作都会不可进行，这里有个细节需要注意的就是所有需要同时lock的地方顺序都是先putLock.lock再takeLock.lock，这样就避免了可能出现的死锁问题。takeLock实例化出一个notEmpty的Condition,putLock实例化一个notFull的Condition,两个Condition协调即时通知线程队列满与不满的状态信息，这在前面几种BlockingQueue实现中也非常常见，在需要用到线程间通知的场景时，各位不妨参考下。另外dequeue的时候需要改变头节点的引用地址,否则肯定会造成不能GC而内存泄露

|  |
| --- |
| private E dequeue() { |
| Node<E> h = head; |

|  |
| --- |
| Node<E> first = h.next; |
| //将原始节点的next指针指向自己,这样就能GC到自己否则虚拟机会认为这个节点仍然在用而不销毁(不知道是否理解有误) | |

|  |  |
| --- | --- |
| h.next = h; // help GC | |
| head = first; |

|  |
| --- |
| E x = first.item; |
| first.item = null; | |

|  |  |
| --- | --- |
| return x; | |
| } |

BlockingDequeue为阻塞的双端队列接口，继承了BlockingQueue，双端队列的最大的特性就是能够将元素添加到队列末尾,也能够添加到队列首部，取元素也是如此。LinkedBlockingDequeue实现了BlockingDequeue接口，就像LinkedBlockingQueue类似，也是由链表结构构成，但是和LinkedBlockingQueue不一样的是，节点元素变成了可双向检索，也就是一个Node持有next节点引用，同时持有prev节点引用，这对队列的头尾数据存取是有决定性意义的。LinkedBlockingDequeue只采用了一个ReetranLock来控制存取并发，并且由这个lock实例化了2个Condition notEmpty和notFull,count变量维护队列长度,这里只使用一个lock来维护队列的读写并发,个人理解是头尾的读写如果使用头尾分开的2个锁,在维护队列长度和队列Empty/Full状态会带来问题,如果使用队列长度做为判定依据将不得不对这个变量进行锁定.

|  |  |
| --- | --- |
| //无论是offerLast,offerFirst,pollFirst,pollLast等等方法都会使用同一把锁. | |
| public E pollFirst() { |

|  |  |
| --- | --- |
| final ReentrantLock lock = this.lock; | |
| lock.lock(); |

|  |
| --- |
| try { |
| return unlinkFirst(); | |

|  |
| --- |
| } finally { |
| lock.unlock(); | |

|  |  |
| --- | --- |
| } | |
| } |

|  |
| --- |
|  |
| public E pollLast() { | |

|  |  |
| --- | --- |
| final ReentrantLock lock = this.lock; | |
| lock.lock(); |

|  |
| --- |
| try { |
| return unlinkLast(); | |

|  |
| --- |
| } finally { |
| lock.unlock(); | |

|  |  |
| --- | --- |
| } | |
| } |

3. ConcurrentMap  
ConcurrentMap定义了V putIfAbsent(K key,V value),Boolean remove(Object Key,Object value),Boolean replace(K key,V oldValue,V newValue)以及V replace(K key,V value)四个方法，几个方法的特性并不难理解，4个方法都是线程安全的。

ConcurrentHashMap是ConcurrentMap的一个实现类，这个类的实现相当经典，基本思想就是分拆锁，默认ConcurrentHashMap会实例化一个持有16个Segment对象的数组，Segment数组大小是可以设定的,构造函数里的concurrencyLevel指定这个值,但是需要注意的是,这个值并不是直接赋值.Segment数组最大长度为MAX\_SEGMENTS = 1 << 16

|  |  |
| --- | --- |
| int sshift = 0; | |
| int ssize = 1; |

|  |  |
| --- | --- |
| //ssize是左移位的,也就是2,4,8,16,32增长(\*2),所以你设定concurrencyLevel为10的时候,这个时候并发数最大为8. | |
| while (ssize < concurrencyLevel) { |

|  |
| --- |
| ++sshift; |
| ssize <<= 1; | |

|  |
| --- |
| } |
| 每个Segment维持一个自动增长的HashEntry数组(根据一个阈值确定是否要增长长度，并不是满了才做). | |

|  |
| --- |
| int c = count; |
| //threshold一般(int)(capacity \* loadFactor), | |

|  |  |
| --- | --- |
| if (c++ > threshold) | |
| rehash(); |

下面3段代码是ConcurrentHashMap的初始化Segment,计算hash值,以及如何选择Segment的代码以及示例注解.

|  |
| --- |
| public ConcurrentHashMap(int initialCapacity, |
| float loadFactor, int concurrencyLevel) { | |

|  |  |
| --- | --- |
| if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0) | |
| throw new IllegalArgumentException(); |

|  |
| --- |
|  |
| //首先确定segment的个数,左移位,并且记录移了几次,比如conurrencyLevel为30,那么2->4->8->16,ssize为16,sshift为4 | |

|  |
| --- |
| if (concurrencyLevel > MAX\_SEGMENTS) |
| concurrencyLevel = MAX\_SEGMENTS; |

|  |
| --- |
|  |
| int sshift = 0; | |

|  |
| --- |
| int ssize = 1; |
| while (ssize < concurrencyLevel) { | |

|  |
| --- |
| ++sshift; |
| ssize <<= 1; | |

|  |
| --- |
| } |
| //segmentShift为28 | |

|  |  |
| --- | --- |
| segmentShift = 32 - sshift; | |
| //segmentMask为15 |

|  |
| --- |
| segmentMask = ssize - 1; |
| //this.segments=new Segment[16] | |

|  |  |
| --- | --- |
| this.segments = Segment.newArray(ssize); | |
|  |

|  |
| --- |
| if (initialCapacity > MAXIMUM\_CAPACITY) |
| initialCapacity = MAXIMUM\_CAPACITY; |

|  |  |
| --- | --- |
| //假设initialCapacity使用32,那么c=2 | |
| int c = initialCapacity / ssize; |

|  |  |
| --- | --- |
| if (c \* ssize < initialCapacity) | |
| ++c; |

|  |  |
| --- | --- |
| int cap = 1; | |
| //cap为2 |

|  |  |
| --- | --- |
| while (cap < c) | |
| cap <<= 1; |

|  |
| --- |
| //每个Segment的容量为2 |
| for (int i = 0; i < this.segments.length; ++i) | |

|  |  |
| --- | --- |
| this.segments[i] = new Segment<K,V>(cap, loadFactor); | |
| } |

|  |
| --- |
|  |
| /\*\* \*//\*\* | |

|  |  |
| --- | --- |
| \*segmentShift为28,segmentMask为15(1111) | |
| \*因为hash值为int,所以32位的 |

|  |  |
| --- | --- |
| \*hash >>> segentShift会留下最高的4位, | |
| \*再与mask 1111做&操作 |

|  |  |
| --- | --- |
| \*所以这个最终会产生 0-15的序列. | |
| \*/ |

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

|  |
| --- |
| } |
|  |

|  |
| --- |
| /\*\* |
| \*将计算的hash值补充到原始hashCode中,这是为了防止 | |

|  |  |
| --- | --- |
| \*外部用户传进来劣质的hash值(比如重复度很高)所带来 | |
| \*的危害. |

|  |
| --- |
| \*/ |
| private static int hash(int h) { | |

|  |  |
| --- | --- |
| // Spread bits to regularize both segment and index locations, | |
| // using variant of single-word Wang/Jenkins hash. |

|  |  |
| --- | --- |
| h += (h <<  15) ^ 0xffffcd7d; | |
| h ^= (h >>> 10); |

|  |
| --- |
| h += (h <<   3); |
| h ^= (h >>>  6); |

|  |  |
| --- | --- |
| h += (h <<   2) + (h << 14); | |
| return h ^ (h >>> 16); |

|  |
| --- |
| } |

当put进来一个key、value对，ConcurrentHashMap会计算Key的hash值，然后从Segment数组根据key的Hash值选出一个Segment，调用其put方法，Segment级别的put方法通过ReetranLock来控制读取的并发，其实Segment本身继承了ReetranLock类。

Segment的put方法在lock()后，首先对数组长度加了新的元素之后是否会超过阈值threshold进行了判定，如果超过，那么进行rehash()，rehash()的过程相对繁琐，首先数组会自动增长一倍，然后需要对HashEntry数组中的所有元素都需要重新计算hash值，并且置到新数组的新的位置，同时为了减小操作损耗，将原来不需要移动的数据不做移动操作(power-of-two expansion,在默认threshold,在数组扩大一倍时只需要移动1/6元素,其他都可以不动)。所有动作完成之后，通过一个while循环寻找Segment中是否有相同Key存在，如果已经存在，那么根据onlyIfAbsent参数确定是否替换(如果为true,不替换，如果为false,替换掉value),然后返回替换的value,如果不存在，那么新生成一个HashEntry,并且根据一开始计算出来的index放到数组指定位置，并且累积元素计数，返回put的值。最后unlock()释放掉锁.

4. CopyOnWriteArrayList和CopyOnWriteArraySet

CopyOnWriteList是线程安全的List实现，其底层数据存储结构为数组(Object[] array),它在读操作远远多于写操作的场景下表现良好，这其中的原因在于其读操作(get(),indexOf(),isEmpty(),contains())不加任何锁，而写操作(set(),add(),remove())通过Arrays.copyOf()操作拷贝当前底层数据结构(array)，在其上面做完增删改等操作，再将新的数组置为底层数据结构，同时为了避免并发增删改， CopyOnWriteList在这些写操作上通过一个ReetranLock进行并发控制。另外需要注意的是，CopyOnWriteList所实现的迭代器其数据也是底层数组镜像，所以在CopyOnWriteList进行interator，同时并发增删改CopyOnWriteList里的数据实不会抛“ConcurrentModificationException”，当然在迭代器上做remove,add,set也是无效的（抛UnsupportedOperationExcetion），因为迭代器上的数据只是当前List的数据数组的一个拷贝而已。

CopyOnWriteSet是一个线程安全的Set实现，然后持有一个CopyOnWriteList实例,其所有的操作都是这个CopyOnWriteList实例来实现的。CopyOnWriteSet与CopyOnWriteList的区别实际上就是Set与List的区别，前者不允许有重复的元素，后者是可以的，所以CopyOnWriteSet的add和addAll两个操作使用的是其内部CopyOnWriteList实例的addAbsent()和addAllAbsent()两个防止重复元素的方法，addAbsent()实现是拷贝底层数据数组，然后逐一比较是否相同，如果有一个相同，那么直接返回false,说明插入失败，如果和其他元素不同，那么将元素加入到新的数组中，最后置回新的数组, addAllAbsent()方法实现则是能有多少数据插入就插入，也就是说addAllAbsent一个集合的数据，可能只有一部分插入成功，另外一部分因为元素相同而遭丢弃，完成后返回插入的元素。