**Concurrent 用户指南（Java 1.8）**

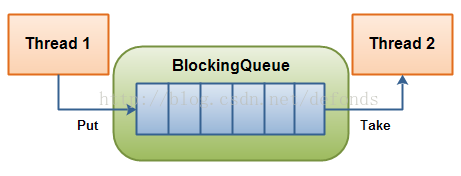
<http://www.importnew.com/26461.html>

**一、阻塞队列 BlockingQueue**

java.util.concurrent 包里的 BlockingQueue 接口表示一个线程安放入和提取实例的队列。本小节我将给你演示如何使用这个 BlockingQueue。本节不会讨论如何在 Java 中实现一个你自己的 BlockingQueue。

**1、BlockingQueue 概念**

BlockingQueue 通常用于一个线程生产对象，而另外一个线程消费这些对象的场景。下图是对这个原理的阐述：



**一个线程往里边放，另外一个线程从里边取的一个 BlockingQueue。**

一个线程将会持续生产新对象并将其插入到队列之中，直到队列达到它所能容纳的临界点。也就是说，它是有限的。如果*该阻塞队列到达了其临界点，负责生产的线程将会在往里边插入新对象时发生阻塞*。它会一直处于阻塞之中，直到负责消费的线程从队列中拿走一个对象。负责消费的线程将会一直从该阻塞队列中拿出对象。如果消费线程尝试去从一个空的队列中提取对象的话，这个消费线程将会处于阻塞之中，直到一个生产线程把一个对象丢进队列。

**2、BlockingQueue 用法**

BlockingQueue 具有 4 组不同的方法用于**插入、移除以及对队列中的元素进行检查**。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：



四组不同的行为方式解释：

* **抛异常：**如果试图的操作无法立即执行，抛一个异常。
* **特定值：**如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。
* **阻塞：**如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。
* **超时：**如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是 true / false)。

无法向一个 BlockingQueue 中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException。可以访问到 BlockingQueue 中的所有元素，而不仅仅是开始和结束的元素。比如说，你将一个对象放入队列之中以等待处理，但你的应用想要将其取消掉。那么你可以调用诸如 remove(o) 方法来将队列之中的特定对象进行移除。但是这么干效率并不高(译者注：基于队列的数据结构，获取除开始或结束位置的其他对象的效率不会太高)，因此你尽量不要用这一类的方法，除非你确实不得不那么做。

**3、BlockingQueue 的实现**

BlockingQueue 是个接口，你需要使用它的实现之一来使用 BlockingQueue。java.util.concurrent 具有以下 BlockingQueue 接口的实现(Java 8)：

* ArrayBlockingQueue：有界数组阻塞队列
* DelayQueue：延迟队列
* LinkedBlockingQueue：链阻塞队列
* PriorityBlockingQueue：具有优先级的阻塞队列
* SynchronousQueue：同步队列

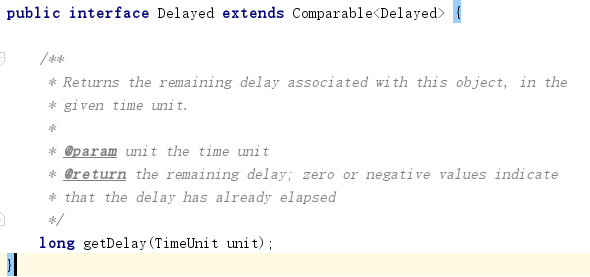
**1）ArrayBlockingQueue**

ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。

‘ArrayBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。**基于ReentrantLock实现锁操作的**。

**2）DelayQueue**

DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口，该接口的定义如下：



DelayQueue 将会在每个元素的 getDelay() 方法返回的值的时间段之后才释放掉该元素。如果返回的是 0 或者负值，延迟将被认为过期，该元素将会在 DelayQueue 的下一次 take 被调用的时候被释放掉。传递给 getDelay 方法的 getDelay 实例是一个枚举类型，它表明了将要延迟的时间段。

TimeUnit 枚举将会取以下值：

* DAYS
* HOURS
* MINUTES
* SECONDS
* MILLISECONDS
* MICROSECONDS
* NANOSECONDS

Delayed 接口也继承了 java.lang.Comparable 接口，这也就意味着 Delayed 对象之间可以进行对比。这个可能在对 DelayQueue 队列中的元素进行**排序**时有用，因此它们可以根据过期时间进行有序释放。

**3）LinkedBlockingQueue**

LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。LinkedBlockingQueue 内部以 FIFO(先进先出)的顺序对元素进行存储。队列中的头元素在所有元素之中是放入时间最久的那个，而尾元素则是最短的那个。

**4）PriorityBlockingQueue**

PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。注意 PriorityBlockingQueue 对于具有相等优先级(compare() == 0)的元素并不强制任何特定行为。

同时注意，如果你从一个 PriorityBlockingQueue 获得一个 Iterator 的话，*该 Iterator 并不能保证它对元素的遍历是以优先级为序的。*

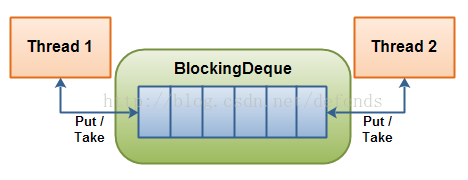
**5）SynchronousQueue**

SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。

**4、BlockingDeque的实现：阻塞双端队列**

java.util.concurrent 包里的 BlockingDeque 接口*表示一个线程安放入和提取实例的双端队列*。本小节我将给你演示如何使用 BlockingDeque。BlockingDeque 类是一个双端队列，在不能够插入元素时，它将阻塞住试图插入元素的线程；在不能够抽取元素时，它将阻塞住试图抽取的线程。deque(双端队列) 是 “Double Ended Queue” 的缩写。因此，双端队列是一个你可以从任意一端插入或者抽取元素的队列。

在线程既是一个队列的生产者又是这个队列的消费者的时候可以使用到 BlockingDeque。如果生产者线程需要在队列的两端都可以插入数据，消费者线程需要在队列的两端都可以移除数据，这个时候也可以使用 BlockingDeque。



**一个 BlockingDeque – 线程在双端队列的两端都可以插入和提取元素。**

一个线程生产元素，并把它们插入到队列的任意一端。如果双端队列已满，插入线程将被阻塞，直到一个移除线程从该队列中移出了一个元素。如果双端队列为空，移除线程将被阻塞，直到一个插入线程向该队列插入了一个新元素。

LinkedBlockingDeque

**二、ConcurrentHashMap**

<http://www.importnew.com/22007.html>

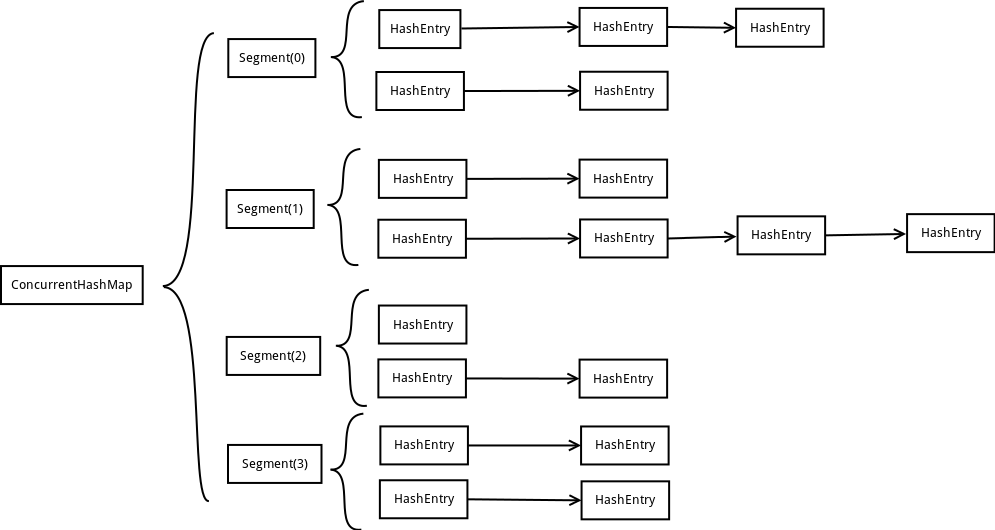
<https://my.oschina.net/hosee/blog/639352>

<http://ifeve.com/concurrenthashmap/>

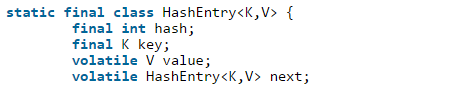
**1、JDK6与JDK7中的实现**

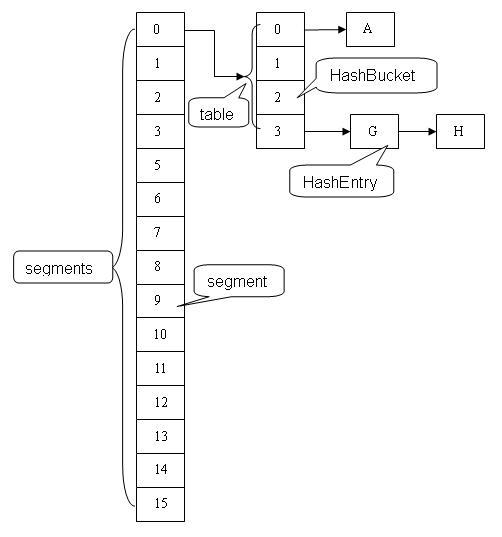
**1）设计思路**

ConcurrentHashMap采用了**分段锁的设计**，只有*在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争*。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size()、containsValue()）需要使用特殊的实现，另外一些方法（如clear()）甚至放弃了对一致性的要求（ConcurrentHashMap是**弱一致性**的。具体请查看[ConcurrentHashMap能完全替代HashTable吗？](https://my.oschina.net/hosee/blog/675423)）

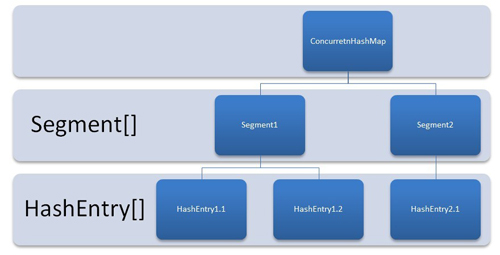


ConcurrentHashMap中的分段锁称为**Segment**，它即类似于HashMap（[JDK7与JDK8中HashMap的实现](https://my.oschina.net/hosee/blog/618953)）的结构，即内部拥有一个**Entry数组**，数组中的每个元素又是一个链表；同时又是一个**ReentrantLock**（**Segment**继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的**可见性**，代码如下：





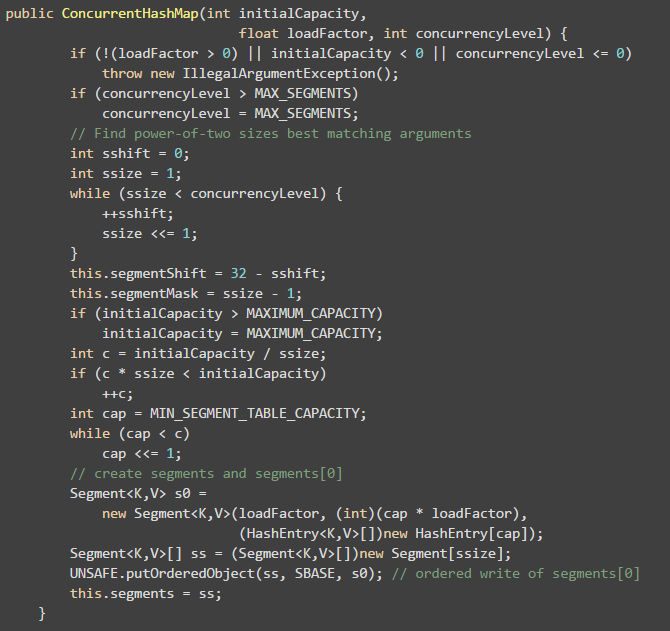
ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment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



**2）并发度（Concurrency Level）：分段锁长度**

并发度可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap默认的并发度为16，但用户也可以在**构造函数中设置并发度**。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的**最小2幂指数**作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。运行时通过将key的**高n位**（n = 32 – segmentShift）和并发度减1（segmentMask）做位与运算**定位**到所在的Segment。segmentShift与segmentMask都是在构造过程中根据concurrency level被相应的计算出来。

如果并发度设置的过小，会带来严重的**锁竞争问题**；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起**程序性能下降**。（文档的说法是根据你并发的线程数量决定，太多会导性能降低）。



传入的参数有initialCapacity，loadFactor，concurrencyLevel这三个。

* initialCapacity表示新创建的这个ConcurrentHashMap的初始容量，也就是上面的结构图中的Entry数量。默认值为static final int DEFAULT\_INITIAL\_CAPACITY = 16;
* loadFactor表示**负载因子**，就是当ConcurrentHashMap中的元素个数大于loadFactor \* 最大容量时就需要rehash，**扩容**。默认值为static final float DEFAULT\_LOAD\_FACTOR = 0.75f;
* concurrencyLevel表示**并发级别**，这个值用来确定Segment的个数，Segment的个数是大于等于concurrencyLevel的第一个2的n次方的数。比如，如果concurrencyLevel为12，13，14，15，16这些数，则Segment的数目为16(2的4次方)。默认值为static final int DEFAULT\_CONCURRENCY\_LEVEL = 16;。理想情况下ConcurrentHashMap的真正的并发访问量能够达到concurrencyLevel，因为有concurrencyLevel个Segment，假如有concurrencyLevel个线程需要访问Map，并且需要访问的数据都恰好分别落在不同的Segment中，则这些线程能够无竞争地自由访问（因为他们不需要竞争同一把锁），达到同时访问的效果。这也是为什么这个参数起名为“并发级别”的原因。

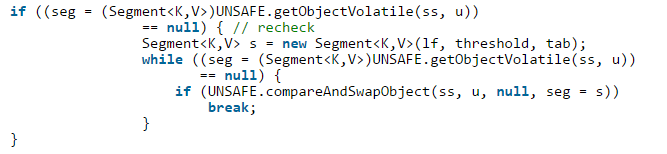
初始化的一些动作：

* 验证参数的合法性，如果不合法，直接抛出异常。
* concurrencyLevel也就是Segment的个数**不能超过规定的最大Segment的个数**，默认值为static final int MAX\_SEGMENTS = 1 << 16;，如果超过这个值，设置为这个值。
* 然后使用循环找到大于等于concurrencyLevel的第一个2的n次方的数ssize，这个数就是Segment**数组的大小**，并记录一共**向左按位移动的次数sshift，并令segmentShift = 32 - sshift**，并且segmentMask的值等于ssize - 1，segmentMask的各个二进制位都为1，目的是之后可以通过key的hash值与这个值做&运算确定Segment的索引。
* 检查给的容量值是否大于允许的最大容量值，如果大于该值，设置为该值。最大容量值为static final int MAXIMUM\_CAPACITY = 1 << 30;。
* 然后计算每个Segment平均应该放置多少个元素，这个值c是向上取整的值。比如初始容量为15，Segment个数为4，则每个Segment平均需要放置4个元素。
* 最后创建一个Segment实例，将其当做Segment数组的第一个元素。

**3）创建分段锁**

和JDK6不同，JDK7中除了第一个Segment之外，剩余的Segments采用的是**延迟初始化的机制**：每次put之前都需要**检查key对应的Segment是否为null**，如果是则调用ensureSegment()以确保对应的Segment被创建。

ensureSegment可能在并发环境下被调用，但与想象中不同，**ensureSegment并未使用锁来控制竞争，而是使用了Unsafe对象的getObjectVolatile()提供的原子读语义结合CAS来确保Segment创建的原子性**。代码段如下：



**4）put/putIfAbsent/putAll**

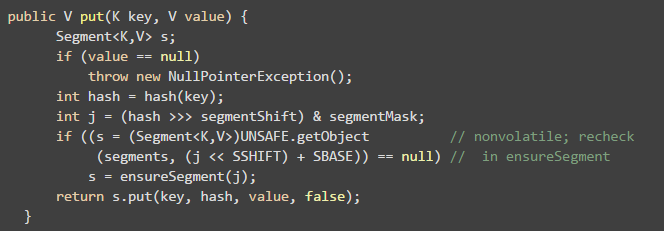
和JDK6一样，ConcurrentHashMap的put方法被代理到了对应的Segment（定位Segment的原理之前已经描述过）中。与JDK6不同的是，JDK7版本的ConcurrentHashMap在获得Segment锁的过程中，做了一定的优化 - 在真正申请锁之前，put方法会通过tryLock()方法尝试获得锁，在尝试获得锁的过程中会对对应hashcode的链表进行**遍历**，如果遍历完毕仍然找不到与key相同的HashEntry节点，则为后续的put操作提前创建一个HashEntry。当tryLock一定次数后仍无法获得锁，则通过lock申请锁。

需要注意的是，由于在并发环境下，其他线程的put，rehash或者remove操作可能会**导致链表头结点的变化**，因此在过程中需要进行**检查**，如果头结点发生变化则重新对表进行遍历。而如果其他线程引起了链表中的某个节点被删除，即使该变化因为是非原子写操作（删除节点后链接后续节点调用的是Unsafe.putOrderedObject()，该方法不提供原子写语义）可能导致当前线程无法观察到，但因为不影响遍历的正确性所以忽略不计。

之所以在获取锁的过程中对整个链表进行遍历，主要目的是**希望遍历的链表被CPU cache所缓存，为后续实际put过程中的链表遍历操作提升性能**。

在获得锁之后，Segment对链表进行遍历，如果某个HashEntry节点具有相同的key，则更新该HashEntry的value值，否则新建一个HashEntry节点，将它设置为链表的新head节点并将原头节点设为新head的下一个节点。新建过程中如果节点总数（含新建的HashEntry）超过threshold，则调用rehash()方法对Segment进行**扩容**，最后将新建HashEntry写入到数组中。

put方法中，链接新节点的下一个节点（HashEntry.setNext()）以及将链表写入到数组中（setEntryAt()）都是通过Unsafe的**putOrderedObject**()方法来实现，这里并未使用具有原子写语义的putObjectVolatile()的原因是：JMM会保证获得锁到释放锁之间所有对象的状态更新都会在锁被释放之后更新到主存，从而保证这些变更对其他线程是可见的。

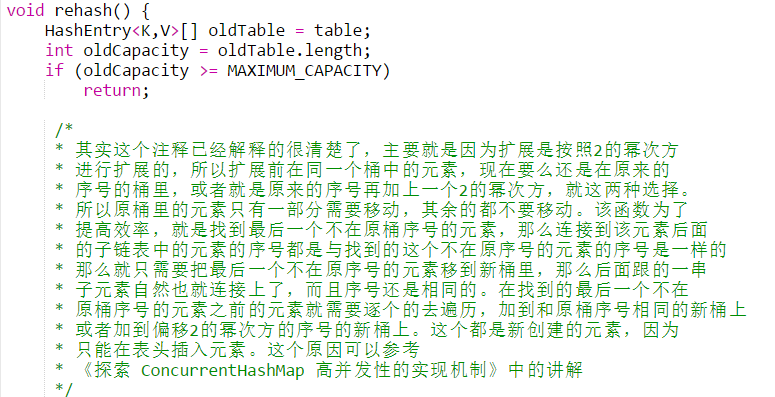


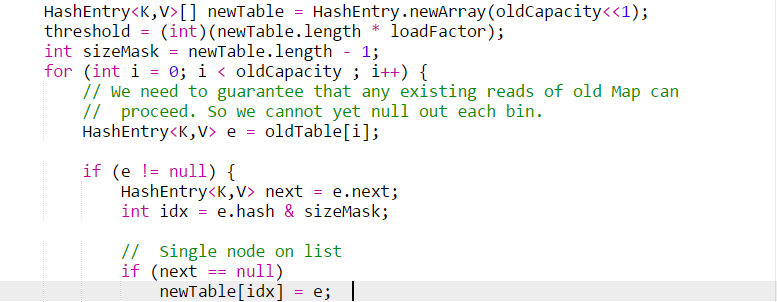
操作步骤如下：

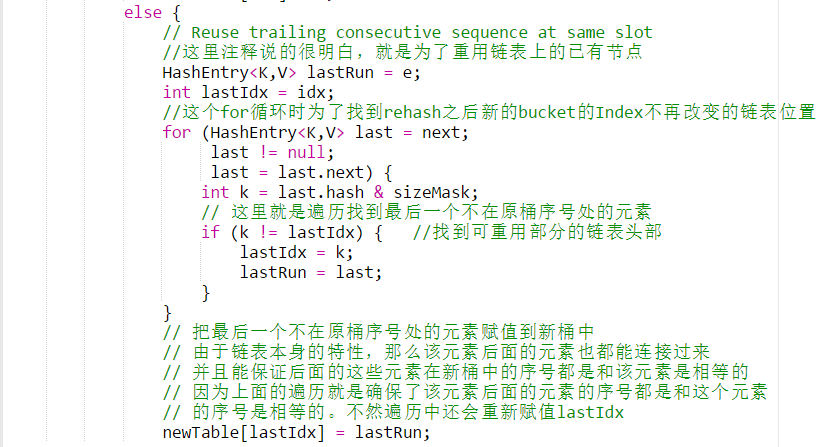
* 判断value是否为null，如果为null，直接抛出异常。
* key通过一次hash运算得到一个hash值。(这个hash运算下文详说)
* 将得到hash值**向右按位移动segmentShift位**，然后再与segmentMask做&运算得到segment的索引j。
* 在初始化的时候我们说过segmentShift的值等于32-sshift，例如concurrencyLevel等于16，则sshift等于4，则segmentShift为28。hash值是一个32位的整数，将其向右移动28位就变成这个样子：0000 0000 0000 0000 0000 0000 0000 xxxx，然后再用这个值与segmentMask做&运算，也就是取最后四位的值。这个值确定Segment的索引。
* 使用Unsafe的方式从Segment数组中获取该**索引对应的Segment对象**。
* 向这个Segment对象中put值，这个put操作也基本是一样的步骤（通过&运算获取HashEntry的索引，然后set）。

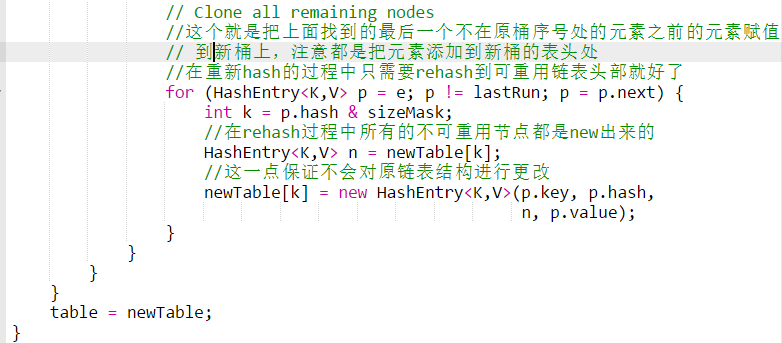
**5）rehash 扩容（节点复制）**

相对于HashMap的resize，ConcurrentHashMap的rehash**原理类似**，但是Doug Lea为rehash做了一定的优化，**避免让所有的节点都进行复制操作：**由于扩容是基于2的幂指来操作，假设扩容前某HashEntry对应到Segment中数组的index为i，数组的容量为capacity，那么扩容后该HashEntry对应到新数组中的index只可能为**i或者i+capacity**，因此大多数HashEntry节点在扩容前后index可以保持不变。基于此，rehash方法中会定位第一个到 后续所有节点在扩容后index都保持不变的节点，然后将这个节点之前的所有节点重排即可。这部分代码如下：

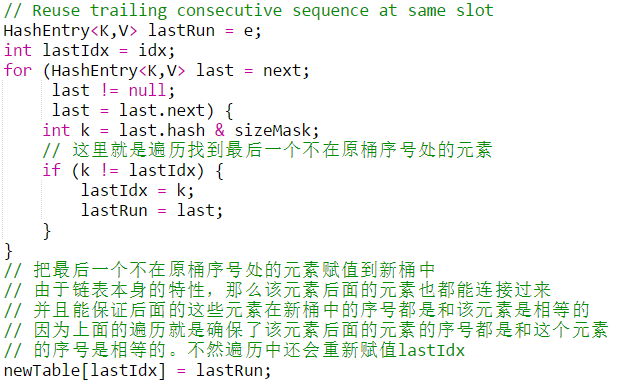








这个函数里之前最让我迷惑的就是那段遍历找最后一个不在原桶序号处元素的代码：



当时我主要想不明白的就是newTable[lastIdx] = lastRun;这里。我刚开始会觉着那么新桶里如果这个lastIdx位置已经有元素了，怎么办，岂不是就给覆盖掉了。后来发现这种情况是不可能出现的。这还是因为table的大小是按照2的幂次方的方式去扩展的。假设原来table的大小是2^k大小，那么现在新table的大小是2^(k+1)大小。而获取序号的方式是

**int idx = e.hash & sizeMask;**

而sizeMask = newTable.length – 1，即sizeMask = 11...1，即全是1，共k个1。获取序号的算法是用**元素的hash值与sizeMask做与的操作**。这样得到的*idx实际上就是元素的hashcode值的低k位的值*。而原table的sizeMask也全是1的二进制，不过总共是k-1位。那么原table的idx就是元素的hashcode的低k-1位的值。所以说**如果元素的hashcode的第k为如果是0，那么元素在新桶的序号就是和原桶的序号是相等的。如果第k位的值是1，那么元素在新桶的序号就是原桶的序号+(2^k-1)**。所以说只可能是这两个值。那么上面的那个newTable[lastIdx] = lastRun；就没问题了，newTable中新序号处此时肯定是空的。

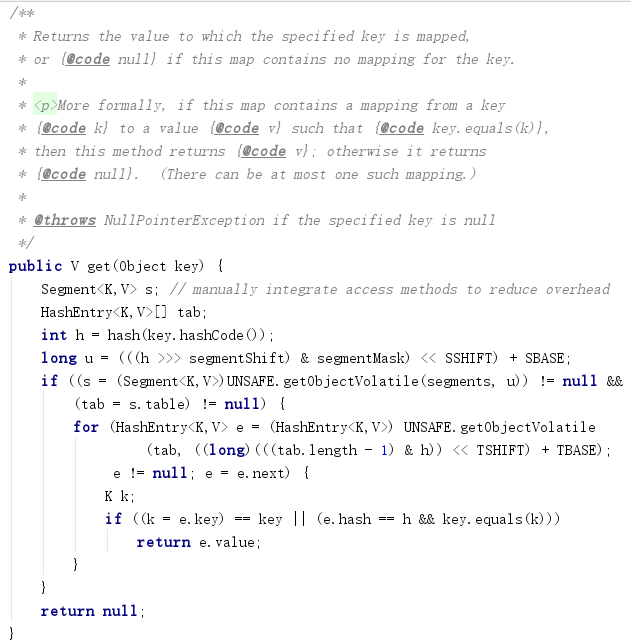
* **是否需要扩容：**在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。
* **如何扩容：**扩容的时候首先会创建一个两倍于原容量的**数组**，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

**6）remove**

和put类似，remove在真正获得锁之前，也会对链表进行遍历以提高缓存命中率。

**7）get与containsKey**

get与containsKey两个方法几乎完全一致：他们都没有使用锁，而是**通过Unsafe对象的getObjectVolatile()方法提供的原子读语义，来获得Segment以及对应的链表，然后对链表遍历判断是否存在key相同的节点以及获得该节点的value**。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在**弱一致性**上的体现。如果要求强一致性，那么必须使用Collections.synchronizedMap()方法。



操作步骤为：

* 和put操作一样，先通过key进行两次hash确定应该去哪个Segment中取数据。
* 使用Unsafe获取对应的Segment，然后再进行一次**&运算得到HashEntry链表的位置**，然后从链表头开始遍历整个链表（因为Hash可能会有碰撞，所以用一个链表保存），如果找到对应的key，则返回对应的value值，如果链表遍历完都没有找到对应的key，则说明Map中不包含该key，返回null。

值得注意的是，get操作是不需要加锁的（如果value为null，会调用readValueUnderLock，只有这个步骤会加锁），通过前面提到的volatile和final来确保数据安全。

**8）size、containsValue**

这些方法都是基于整个ConcurrentHashMap来进行操作的，他们的原理也基本类似：**首先不加锁循环执行以下操作**：循环所有的Segment（通过Unsafe的getObjectVolatile()以保证原子读语义），获得对应的值以及所有Segment的modcount之和。如果连续两次所有Segment的modcount和相等，则过程中没有发生其他线程修改ConcurrentHashMap的情况，返回获得的值。

当循环次数超过预定义的值时，这时需要对所有的Segment依次进行加锁，获取返回值后再依次解锁。值得注意的是，加锁过程中要强制创建所有的Segment，否则容易出现其他线程创建Segment并进行put，remove等操作。

一般来说，应该**避免在多线程环境下使用size和containsValue方法**。

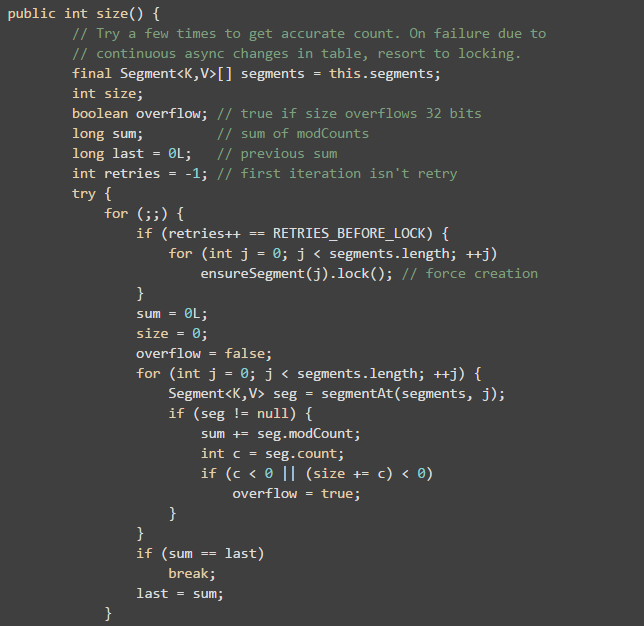
**注1：**modcount在put, replace, remove以及clear等方法中都会被修改。

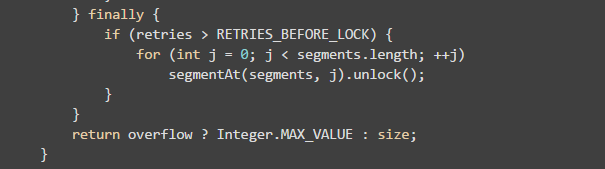
**注2：**对于containsValue方法来说，如果在**循环过程中发现匹配value的HashEntry**，则直接返回true。

最后，与HashMap不同的是，ConcurrentHashMap并不允许key或者value为null，按照Doug Lea的说法，这么设计的原因是在ConcurrentHashMap中，一旦value出现null，则代表HashEntry的key/value没有映射完成就被其他线程所见，需要特殊处理。在JDK6中，get方法的实现中就有一段对HashEntry.value == null的防御性判断。

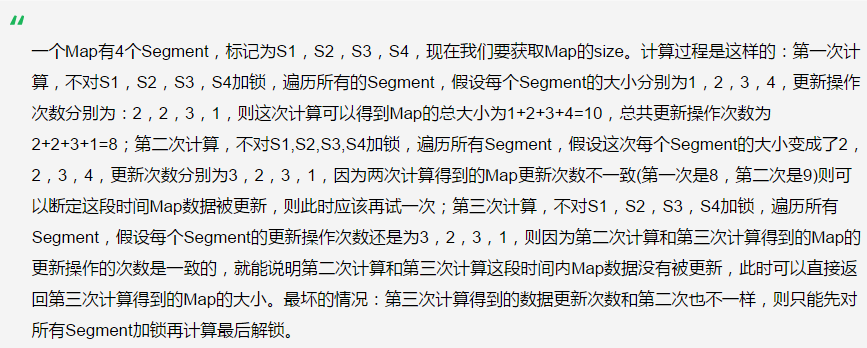
* Size操作

size操作与put和get操作最大的区别在于，**size操作需要遍历所有的Segment才能算出整个Map的大小，而put和get都只关心一个Segment**。假设我们当前遍历的Segment为SA，那么在遍历SA过程中其他的Segment比如SB可能会被修改，于是这一次运算出来的size值可能并不是Map当前的真正大小。所以一个比较简单的办法就是计算Map大小的时候所有的Segment都Lock住，不能更新(包含put，remove等等)数据，计算完之后再Unlock。这是普通人能够想到的方案，但是牛逼的作者还有一个**更好的Idea：**先给3次机会，不lock所有的Segment，遍历所有Segment，累加各个Segment的大小得到整个Map的大小，如果某相邻的两次计算获取的所有Segment的更新的次数（每个Segment都有一个**modCount变量**，这个变量在Segment中的*Entry被修改时会加一，通过这个值可以得到每个Segment的更新操作的次数*）是一样的，说明计算过程中没有更新操作，则直接返回这个值。**如果这三次不加锁的计算过程中Map的更新次数有变化，则之后的计算先对所有的Segment加锁，再遍历所有Segment计算Map大小，最后再解锁所有Segment**。源代码如下：

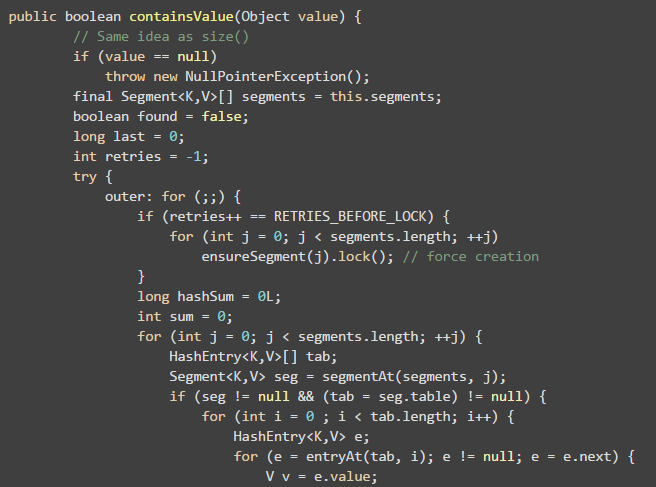


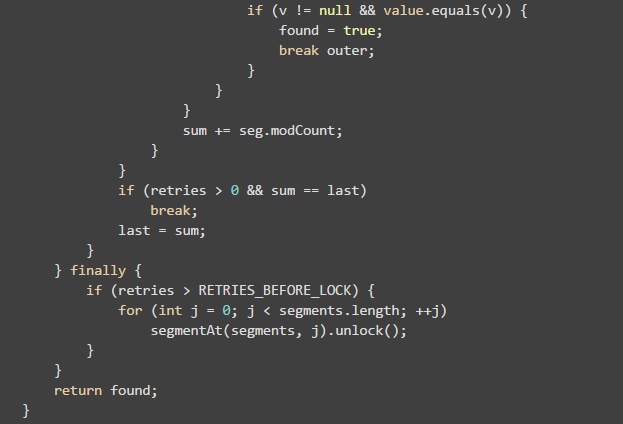


举个例子：



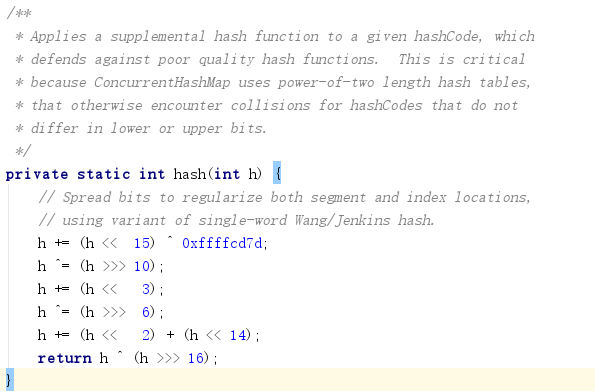
* containsValue操作：containsValue操作采用了和size操作一样的想法



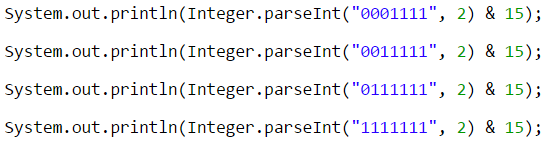


**9）hash**

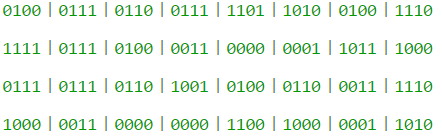
**定位Segment：**既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素的时候，必须**先通过哈希算法定位到Segment**。可以看到ConcurrentHashMap会首先使用Wang/Jenkins hash的变种算法对元素的hashCode进行一次再哈希。



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。



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。



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。

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。

**2、JDK8实现**

<http://blog.csdn.net/elricboa/article/details/70199409>

ConcurrentHashMap在JDK8中进行了巨大改动，它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现，利用CAS算法。它沿用了与它同时期的HashMap版本的思想，底层依然由“数组”+链表+红黑树的方式思想，但是为了做到并发，又增加了很多辅助的类，例如TreeBin，Traverser等对象内部类。

**1）重要的属性**

sizeCtl这个属性。可以说它是ConcurrentHashMap中出镜率很高的一个属性，因为它是一个控制标识符，在不同的地方有不同用途，而且它的取值不同，也代表不同的含义。

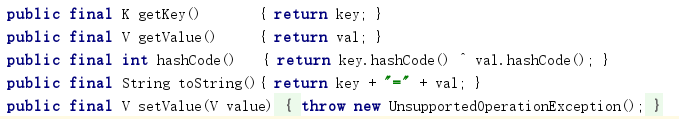
* 负数代表正在进行初始化或扩容操作
* -1代表正在初始化
* -N 表示有N-1个线程正在进行扩容操作
* 正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小，这一点类似于扩容阈值的概念。还后面可以看到，它的值始终是当前ConcurrentHashMap容量的0.75倍，这与loadfactor是对应的。

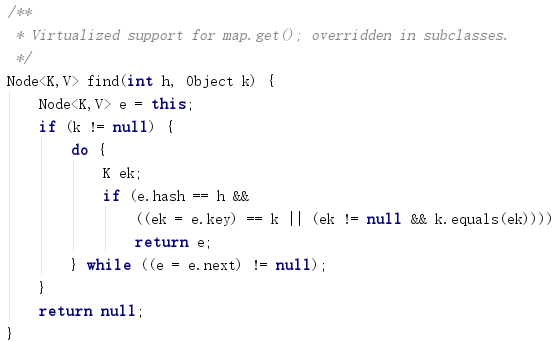


**2）重要的类**

**A. Node**

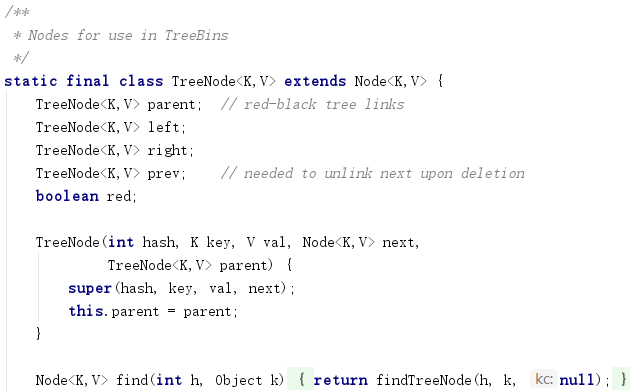
Node是最核心的内部类，它包装了key-value键值对，所有插入ConcurrentHashMap的数据都包装在这里面。它与HashMap中的定义很相似，但是但是有一些差别它对value和next属性设置了**volatile同步锁**(与JDK7的Segment相同)，它不允许调用setValue方法直接改变Node的value域，它增加了find方法辅助map.get()方法。





**B. TreeNode**

树节点类，另外一个核心的数据结构。当**链表长度过长的时候，会转换为TreeNode**。但是与HashMap不相同的是，它并不是直接转换为红黑树，而是把这些结点包装成TreeNode放在**TreeBin对象**中，由TreeBin完成对红黑树的包装。而且TreeNode在ConcurrentHashMap集成自Node类，而并非HashMap中的集成自LinkedHashMap.Entry<K,V>类，也就是说**TreeNode带有next指针**，这样做的目的是方便基于TreeBin的访问。

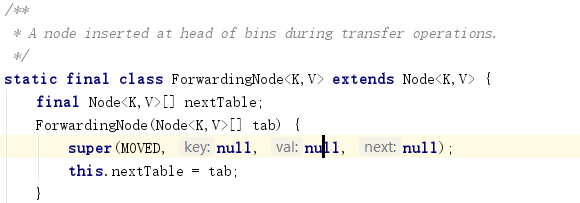


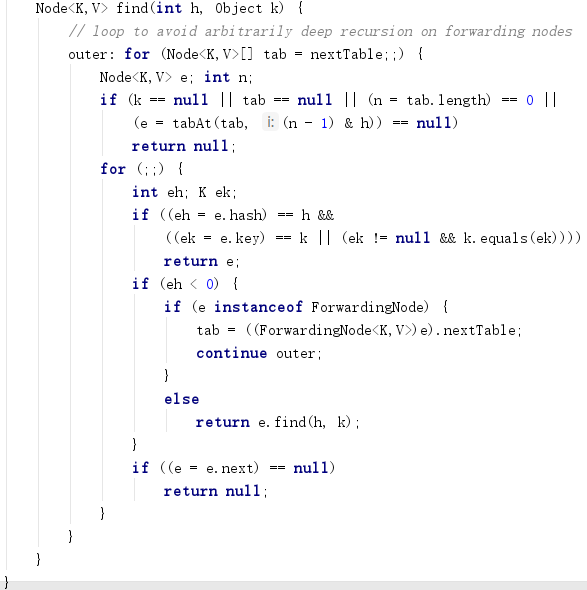
**C. TreeBin**

这个类并不负责包装用户的key、value信息，而是**包装很多TreeNode节点**。它代替了TreeNode的根节点，也就是说在实际的ConcurrentHashMap“数组”中，存放的是TreeBin对象，而不是TreeNode对象，这是与HashMap的区别。另外这个类还带有了**读写锁**。

**D. ForwardingNode**

一个用于**连接两个table的节点类**。它包含一个**nextTable指针，用于指向下一张表**。而且这个节点的key value next指针全部为null，它的hash值为-1. 这里面定义的find的方法是从nextTable里进行查询节点，而不是以自身为头节点进行查找。



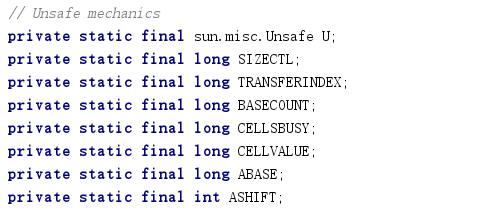


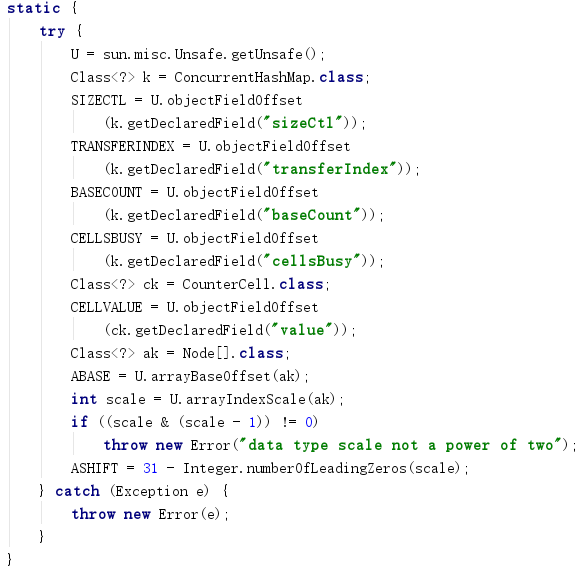
**3）Unsafe与CAS**

在ConcurrentHashMap中，随处可以看到U, 大量使用了U.compareAndSwapXXX的方法，这个方法是利用一个CAS算法实现无锁化的修改值的操作，他可以大大降低锁代理的性能消耗。这个算法的**基本思想**就是不断地去比较当前内存中的变量值与你指定的一个变量值是否相等，如果相等，则接受你指定的修改的值，否则拒绝你的操作。因为当前线程中的值已经不是最新的值，你的修改很可能会覆盖掉其他线程修改的结果。这一点与乐观锁，SVN的思想是比较类似的。

**A. unsafe静态块**

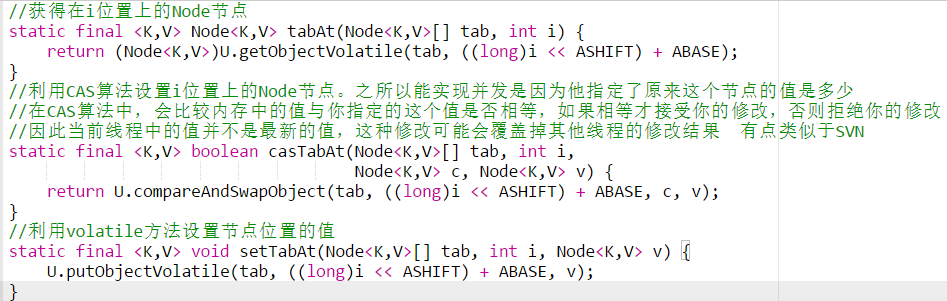
unsafe代码块控制了一些属性的修改工作，比如最常用的SIZECTL 。在这一版本的concurrentHashMap中，大量应用来的**CAS方法进行变量、属性的修改工作**。利用CAS进行无锁操作，可以大大提高性能。





**B. 三个核心方法**

ConcurrentHashMap定义了***三个原子操作，用于对指定位置的节点进行操作***。正是这些原子操作保证了ConcurrentHashMap的线程安全。



**C. 初始化方法initTable**

对于ConcurrentHashMap来说，调用它的构造方法仅仅是设置了一些参数而已。而整个table的初始化是在向ConcurrentHashMap中插入元素的时候发生的。如调用put、computeIfAbsent、compute、merge等方法的时候，**调用时机是检查table==null**。

初始化方法主要应用了关键属性sizeCtl 如果这个值〈0，表示其他线程正在进行初始化，就放弃这个操作。在这也可以看出**ConcurrentHashMap的初始化只能由一个线程完成。如果获得了初始化权限，就用CAS方法将sizeCtl置为-1，防止其他线程进入**。初始化数组后，将sizeCtl的值改为0.75\*n。



**D. 扩容方法 transfer**

当ConcurrentHashMap容量不足的时候，需要对table进行扩容。这个方法的**基本思想跟HashMap是很像的**，但是由于它是**支持并发扩容**的，所以要复杂的多。原因是它支持多线程进行扩容操作，而并没有加锁。我想这样做的目的不仅仅是为了满足concurrent的要求，而是希望利用并发处理去减少扩容带来的时间影响。因为在扩容的时候，总是会涉及到从一个“数组”到另一个“数组”拷贝的操作，如果这个操作能够并发进行，那真真是极好的了。

整个扩容操作分为两个部分：

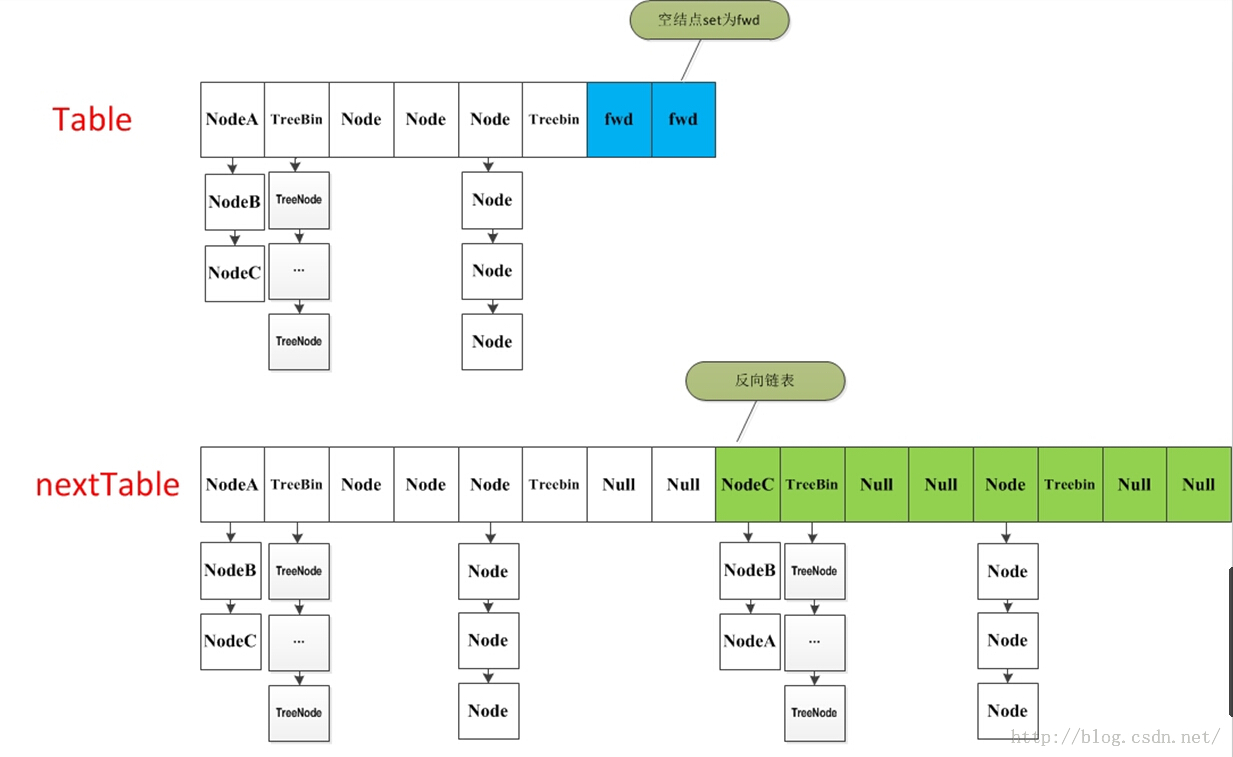
* 第一部分：构建一个nextTable，它的容量是原来的两倍，这个操作是**单线程完成**的。这个单线程的保证是通过RESIZE\_STAMP\_SHIFT这个常量经过一次运算来保证的，这个地方在后面会有提到；
* 第二个部分：将原来table中的元素复制到nextTable中，这里允许**多线程进行操作**。

先来看一下单线程是如何完成的：大体思想就是**遍历、复制的过程**。首先根据运算得到需要遍历的次数i，然后利用tabAt方法获得i位置的元素：

* 如果这个位置为空，就在原table中的i位置**放入forwardNode节点**，这个也是**触发并发扩容的关键点**；
* 如果这个位置是Node节点（fh>=0），如果它是一个链表的头节点，就构造一个反序链表，把他们分别放在nextTable的i和i+n的位置上
* 如果这个位置是TreeBin节点（fh<0），也做一个反序处理，并且判断是否**需要untreefi**，把处理的结果分别放在nextTable的i和i+n的位置上
* 遍历过所有的节点以后就完成了复制工作，这时让nextTable作为新的table，并且更新sizeCtl为新容量的0.75倍 ，完成扩容。

**多线程是如何完成的：**

在代码的69行有一个判断，如果遍历到的节点是forward节点，就向后继续遍历，再加上给节点上**锁的机制**，就完成了多线程的控制。多线程遍历节点，处理了一个节点，就把对应点的值set为forward，另一个线程看到forward，就**向后遍历**。这样交叉就完成了复制工作。而且还很好的解决了线程安全的问题。 这个方法的设计实在是让我膜拜。



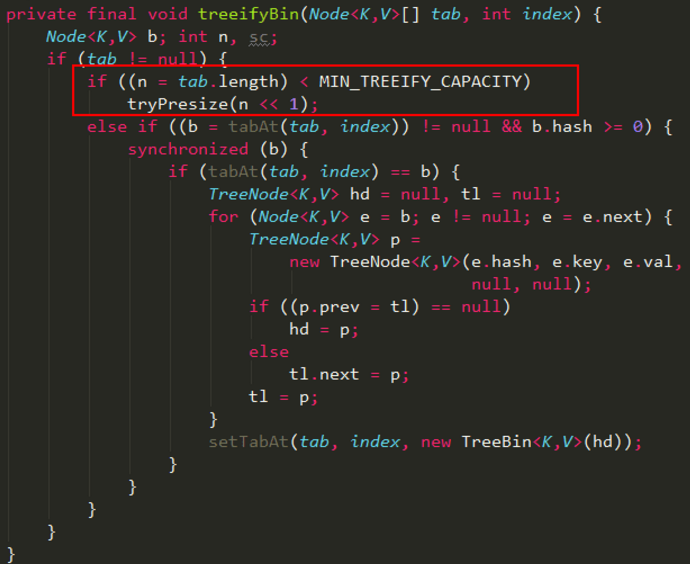
深入分析ConcurrentHashMap1.8的扩容实现

<http://www.jianshu.com/p/f6730d5784ad>

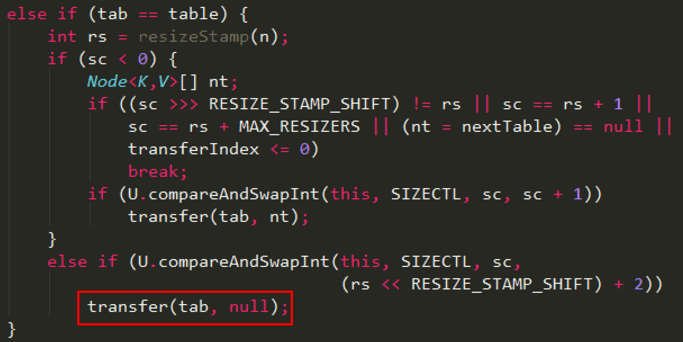
**什么情况会触发扩容？**

当往hashMap中成功插入一个key/value节点时，有可能触发扩容动作：

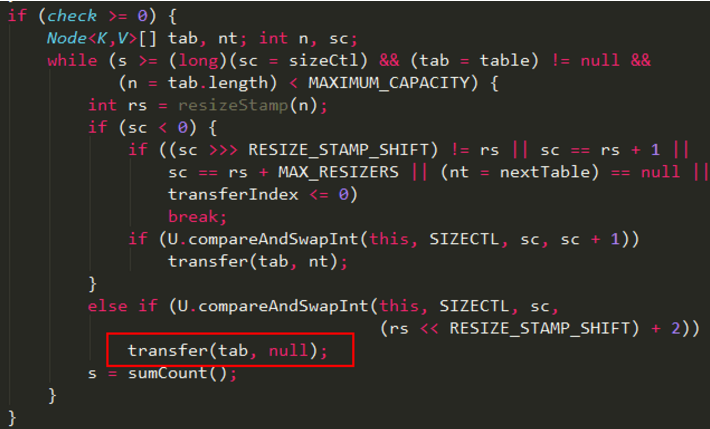
1、如果新增节点之后，所在**链表的元素个数**达到了阈值 8，则会调用**treeifyBin方法把链表转换成红黑树**，不过在结构转换之前，会**对数组长度进行判断**，实现如下：



如果数组长度**n小于阈值**MIN\_TREEIFY\_CAPACITY，默认是64，则会调用tryPresize方法把数组长度扩大到原来的两倍，并触发transfer方法，重新调整节点的位置。

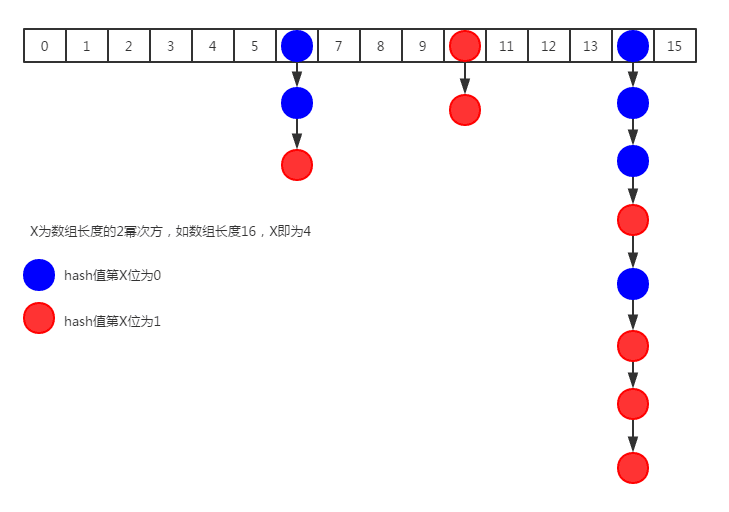


2、新增节点之后，会调用addCount方法记录元素个数，并检查是否需要进行扩容，当数组元素个数达到阈值时，会触发transfer方法，重新调整节点的位置。



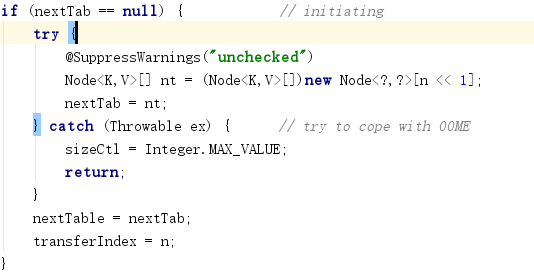
**transfer实现**

transfer方法实现了在并发的情况下，**高效的从原始组数往新数组中移动元素**，假设扩容之前节点的分布如下，这里区分蓝色节点和红色节点，是为了后续更好的分析：

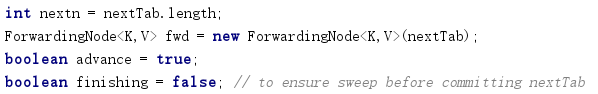


在上图中，第14个槽位插入新节点之后，链表元素个数已经达到了8，且数组长度为16，优先通过扩容来缓解链表过长的问题，实现如下：

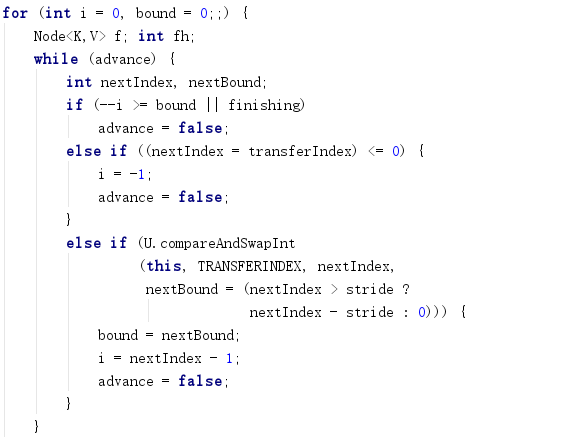
* 根据当前数组长度n，新建一个**两倍长度的数组nextTable；**



* 初始化ForwardingNode节点，其中**保存了新数组nextTable的引用**，在处理完每个槽位的节点之后当做占位节点，表示该槽位已经处理过了；



* 通过for自循环处理每个槽位中的链表元素，默认advace为真，通过CAS设置transferIndex属性值，并初始化i和bound值，i指当前处理的槽位序号，bound指需要处理的槽位边界，先处理槽位15的节点；



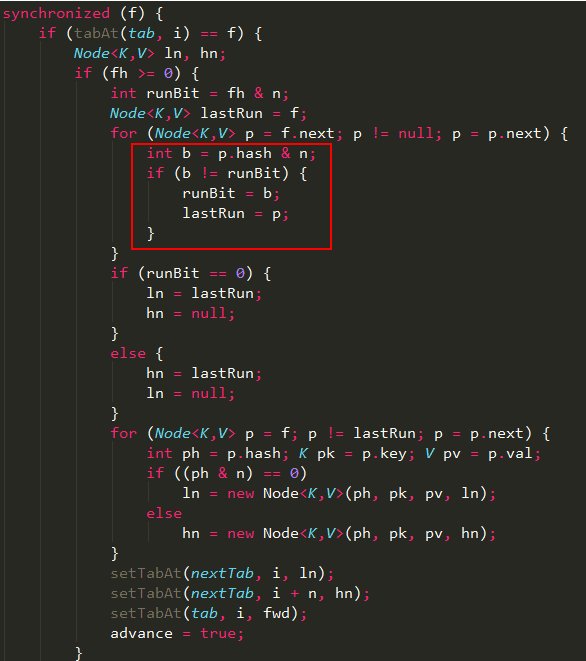
* 在当前假设条件下，槽位15中没有节点，则通过CAS插入在第二步中初始化的ForwardingNode节点，用于告诉其它线程该槽位已经处理过了；



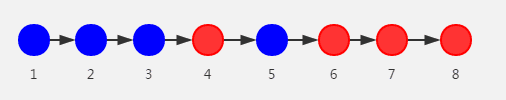
* 如果槽位15已经被线程A处理了，那么线程B处理到这个节点时，取到该节点的hash值应该为MOVED，值为-1，则**直接跳过，继续处理下一个槽位14的节点**；



* 处理槽位14的节点，是一个链表结构，先定义两个变量节点ln和hn，按我的理解应该是lowNode和highNode，分别保存hash值的第X位为0和1的节点，具体实现如下：

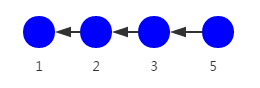


使用fn&n可以快速把链表中的元素区分成两类，A类是hash值的第X位为0，B类是hash值的第X位为1，并通过lastRun记录最后需要处理的节点，A类和B类节点可以分散到新数组的槽位14和30中，在原数组的槽位14中，蓝色节点第X为0，红色节点第X为1，把链表拉平显示如下：

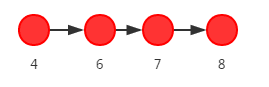


* 通过**遍历链表**，记录runBit和lastRun，分别为1和节点6，所以设置hn为节点6，ln为null；
* **重新遍历链表**，以lastRun节点为终止条件，根据第X位的值分别构造ln链表和hn链表：

ln链：和原来链表相比，顺序已经不一样了

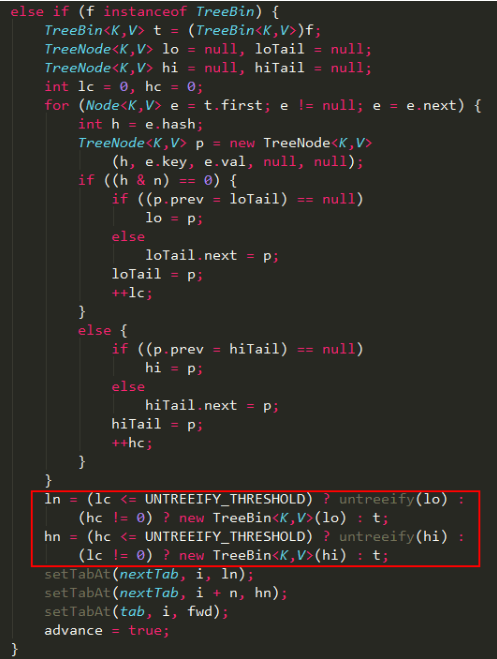


hn链：



通过CAS把ln链表设置到新数组的i位置，hn链表设置到i+n的位置；

* 如果该槽位是红黑树结构，则构造树节点lo和hi，遍历红黑树中的节点，同样根据hash&n算法，把节点分为两类，**分别插入到lo和hi为头的链表中**，根据lo和hi链表中的元素个数分别生成ln和hn节点，其中ln节点的生成逻辑如下：
* 如果lo链表的元素个数小于等于UNTREEIFY\_THRESHOLD，默认为6，则通过untreeify方法把树节点链表转化成普通节点链表；
* 否则判断hi链表中的元素个数是否等于0：如果等于0，表示lo链表中包含了所有原始节点，则设置原始红黑树给ln，否则根据lo链表重新构造红黑树。



最后，同样的通过CAS把ln设置到新数组的i位置，hn设置到i+n位置。

**3、总结**

JDK6,7中的ConcurrentHashmap主要使用**Segment来实现减小锁粒度**，把HashMap分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用**volatile来保证可见性**，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

jdk7中ConcurrentHashmap中，当**长度过长碰撞会很频繁，链表的增改删查操作都会消耗很长的时间，影响性能**，所以jdk8 中完全重写了concurrentHashmap，代码量从原来的1000多行变成了 6000多 行，实现上也和原来的分段式存储有很大的区别。

主要设计上的变化有以下几点:

* 不采用segment而采用node，**锁住node来实现减小锁粒度**。
* 设计了MOVED状态 当resize的中过程中线程2还在put数据，线程2会帮助resize。
* 使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。
* sizeCtl的不同值来代表不同含义，起到了**控制的作用**。

**三、阻塞队列 BlockingQueue**

**四、阻塞队列 BlockingQueue**