ConcurrentHashMap源码剖析

2016/09/29 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [3 条评论](http://www.importnew.com/21781.html#comments) | 标签： [CONCURRENTHASHMAP](http://www.importnew.com/tag/concurrenthashmap)

分享到：**10**

原文出处： [我爱物联网](http://www.cnblogs.com/yydcdut/p/3959815.html)

**ConcurrentHashMap**是Java5中新增加的一个线程安全的Map集合，可以用来替代**HashTable**。对于ConcurrentHashMap是如何提高其效率的，可能大多人只是知道它使用了多个锁代替HashTable中的单个锁，也就是**锁分离技术（Lock Stripping）**。实际上，ConcurrentHashMap对提高并发方面的优化，还有一些其它的技巧在里面（比如你是否知道在get操作的时候，它是否也使用了锁来保护？）。

ConcurrentMap

*提供其他原子 putIfAbsent、remove、replace 方法的 Map。 内存一致性效果：当存在其他并发 collection 时，将对象放入 ConcurrentMap 之前的线程中的操作 happen-before 随后通过另一线程从 ConcurrentMap 中访问或移除该元素的操作。*

*我们不关心ConcurrentMap中新增的接口，重点理解一下内存一致性效果中的****“happens-before”****是怎么回事。因为要想从根本上讲明白，这个是无法避开的。这又不得不从Java存储模型来谈起了。*

理解JAVA存储模型(JMM)的Happens-Before规则

在解释该规则之前，我们先看一段多线程访问数据的代码例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Test1 {      private int a=1, b=2;        public void foo(){  // 线程1          a=3;          b=4;      }        public int getA(){ // 线程2          return a;      }      public int getB(){ // 线程2          return b;      }  } |

上面的代码，当线程1执行foo方法的时候，线程2访问getA和getB会得到什么样的结果？  
答案：

|  |  |
| --- | --- |
| 1  2  3  4 | A：a=1, b=2  // 都未改变  B：a=3, b=4  // 都改变了  C：a=3, b=2  //  a改变了，b未改变  D：a=1, b=4  //  b改变了，a未改变 |

上面的A,B,C都好理解，但是D可能会出乎一些人的预料。  
一些不了解JMM的同学可能会问怎么可能 b=4语句会先于 a=3 执行？

这是一个多线程之间内存可见性（Visibility）顺序不一致的问题。有两种可能会造成上面的D选项。

**1) Java编译器的重排序(Reording)操作有可能导致执行顺序和代码顺序不一致。指令重排**  
关于Reording：

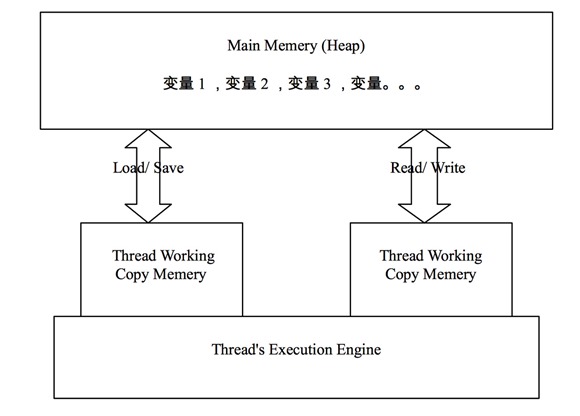
*Java语言规范规定了JVM要维护内部线程类似顺序化语义(within-thread as-is-serial semantics)：只要程序的最终结果等同于它在严格的顺序化环境中执行的结果，那么上述所有的行为都是允许的。*

上面的话是《Java并发编程实践》一书中引自Java语言规范的，感觉翻译的不太好。简单的说：假设代码有两条语句，代码顺序是语句1先于语句2执行；那么只要语句2不依赖于语句1的结果，打乱它们的顺序对最终的结果没有影响的话，那么真正交给CPU去执行时，他们的顺序可以是没有限制的。可以允许语句2先于语句1被CPU执行，和代码中的顺序不一致。

重排序（Reordering）是JVM针对现代CPU的一种优化，Reordering后的指令会在性能上有很大提升。(不知道这种优化对于多核CPU是否更加明显，也或许和单核多核没有关系。)

因为我们例子中的两条赋值语句，并没有依赖关系，无论谁先谁后结果都是一样的，所以就可能有Reordering的情况，这种情况下，对于其他线程来说就可能造成了可见性顺序不一致的问题。

**2) 从线程工作内存写回主存时顺序无法保证。**  
下图描述了JVM中主存和线程工作内存之间的交互：

[](http://www.importnew.com/21781.html/1-39)

JLS中对线程和主存互操作定义了6个行为，分别为load，save，read，write，assign和use，这些操作行为具有原子性（me:除了操作long和double类型变量在某些平台上需要多次操作，从而不具备原子性），且相互依赖，有明确的调用先后顺序。这个细节也比较繁琐，我们暂不深入追究。先简单认为线程在修改一个变量时，先拷贝入线程工作内存中，在线程工作内存修改后再写回主存(Main Memery)中。

假设例子中Reording后顺序仍与代码中的顺序一致，那么接下来呢？有意思的事情就发生在线程把Working Copy Memery中的变量写回Main Memery的时刻。线程1把变量写回Main Memery的过程对线程2的可见性顺序也是无法保证的。  
上面的列子，a=3; b=4; 这两个语句在 Working Copy Memery中执行后，写回主存的过程对于线程2来说同样可能出现先b=4；后a=3；这样的相反顺序。

正因为上面的那些问题，JMM中一个重要问题就是：如何让多线程之间，对象的状态对于各线程的“可视性”是顺序一致的。它的解决方式就是 Happens-before 规则：  
JMM为所有程序内部动作定义了一个偏序关系，叫做happens-before。要想保证执行动作B的线程看到动作A的结果（无论A和B是否发生在同一个线程中），A和B之间就必须满足happens-before关系。

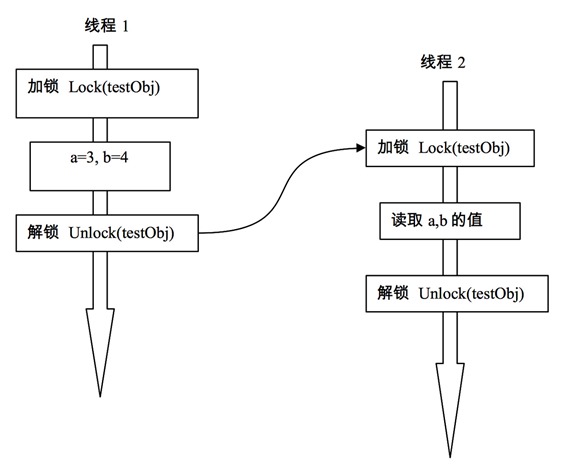
我们现在来看一下“Happens-before”规则都有哪些（摘自《Java并发编程实践》）：

*① 程序次序法则：线程中的每个动作A都happens-before于该线程中的每一个动作B，其中，在程序中，所有的动作B都能出现在A之后。  
② 监视器锁法则：对一个监视器锁的解锁 happens-before于每一个后续对同一监视器锁的加锁。*

*③ volatile变量法则：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。  
④ 线程启动法则：在一个线程里，对Thread.start的调用会happens-before于每个启动线程的动作。  
⑤ 线程终结法则：线程中的任何动作都happens-before于其他线程检测到这个线程已经终结、或者从Thread.join调用中成功返回，或Thread.isAlive返回false。  
⑥ 中断法则：一个线程调用另一个线程的interrupt happens-before于被中断的线程发现中断。  
⑦ 终结法则：一个对象的构造函数的结束happens-before于这个对象finalizer的开始。  
⑧ 传递性：如果A happens-before于B，且B happens-before于C，则A happens-before于C*

我们重点关注的是②，③，这两条也是我们通常编程中常用的。  
后续分析ConcurrenHashMap时也会看到使用到锁(ReentrantLock)，Volatile，final等手段来保证happens-before规则的。

使用锁方式实现“Happens-before”是最简单，容易理解的。

[](http://www.importnew.com/21781.html/2-29)

早期Java中的锁只有最基本的synchronized，它是一种互斥的实现方式。在Java5之后，增加了一些其它锁，比如ReentrantLock，它基本作用和synchronized相似，但提供了更多的操作方式，比如在获取锁时不必像synchronized那样只是傻等，可以设置定时，轮询，或者中断，这些方法使得它在获取多个锁的情况可以避免死锁操作。

而我们需要了解的是ReentrantLock的性能相对synchronized来说有很大的提高。（不过据说Java6后对synchronized进行了优化，两者已经接近了。）在ConcurrentHashMap中，每个hash区间使用的锁正是ReentrantLock。

Volatile可以看做一种轻量级的锁，但又和锁有些不同。  
a) 它对于多线程，不是一种互斥（mutex）关系。  
b) 用volatile修饰的变量，不能保证该变量状态的改变对于其他线程来说是一种“原子化操作”。

在Java5之前，JMM对Volatile的定义是：保证读写volatile都直接发生在main memory中，线程的working memory不进行缓存。它只承诺了读和写过程的可见性，并没有对Reording做限制，所以旧的Volatile并不太可靠。在Java5之后，JMM对volatile的语义进行了增强。就是我们看到的③ volatile变量法则。

那对于“原子化操作”怎么理解呢？看下面例子：

|  |  |
| --- | --- |
| 1  2  3  4  5 | private static volatile int nextSerialNum = 0;    public static int generateSerialNumber(){      return nextSerialNum++;  } |

上面代码中对nextSerialNum使用了volatile来修饰，根据前面“Happens-Before”法则的第三条Volatile变量法则，看似不同线程都会得到一个新的serialNumber

问题出在了 **nextSerialNum++** 这条语句上，它不是一个原子化的，实际上是**read-modify-write**三项操作，这就有可能使得在线程1在write之前，线程2也访问到了nextSerialNum，造成了线程1和线程2得到一样的serialNumber。  
所以，在使用Volatile时，需要注意  
a)  需不需要互斥；  
b) 对象状态的改变是不是原子化的。

最后也说一下final 关键字。

**不变模式（immutable）**是多线程安全里最简单的一种保障方式。因为你拿他没有办法，想改变它也没有机会。  
不变模式主要通过final关键字来限定的。在JMM中final关键字还有特殊的语义。Final域使得确保初始化安全性（initialization safety）成为可能，初始化安全性让不可变对象不需要同步就能自由地被访问和共享。

**3）经过前面的了解，下面我们用Happens-Before规则理解一个经典问题：双重检测锁(DCL)为什么在java中不适用？**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class LazySingleton {      private int someField;      private static LazySingleton instance;        private LazySingleton(){          this.someField = new Random().nextInt(200) + 1; // (1)      }        public static LazySingleton getInstance() {          if (instance == null) {// (2)              synchronized (LazySingleton.class) { // (3)                if (instance == null) { // (4)                  instance = new LazySingleton(); // (5)                }              }          }          return instance; // (6)      }        public int getSomeField() {          return this.someField;  // (7)      }  } |

我想简单的用对象创建期间的实际场景来分析一下：（注意，这种场景是我个人的理解，所看的资料也是非官方的，不完全保证正确。如果发现不对请指出。

假设线程1执行完(5)时，线程2正好执行到了(2)；  
看看 new LazySingleton(); 这个语句的执行过程： 它不是一个原子操作，实际是由多个步骤，我们从我们关注的角度简化一下，简单的认为它主要有2步操作好了：  
a） 在内存中分配空间，并将引用指向该内存空间。  
b） 执行对象的初始化的逻辑(和操作)，完成对象的构建。

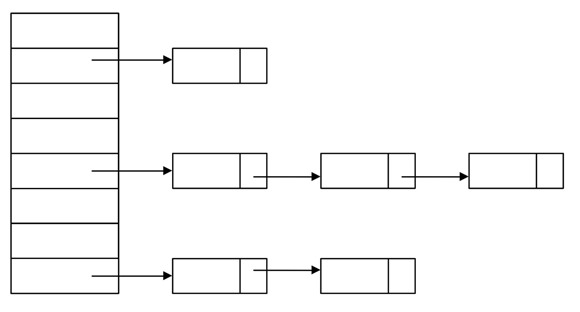
此时因为线程1和线程2没有用同步，他们之间不存在“Happens-Before”规则的约束，所以在线程1创建LazySingleton对象的 a),b)这两个步骤对于线程2来说会有可能出现a)可见，b)不可见  
造成了线程2获取到了一个未创建完整的lazySingleton对象引用，为后边埋下隐患。

之所以这里举到 DCL这个例子，是因为我们后边分析ConcurrentHashMap时，也会遇到相似的情况。  
对于对象的创建，出于乐观考虑，两个线程之间没有用“Happens-Before规则来约束”另一个线程可能会得到一个未创建完整的对象，这种情况必须要检测，后续分析ConcurrentHashMap时再讨论。

ConcurrentHashMap

我们关注的操作有：get，put，remove 这3个操作。

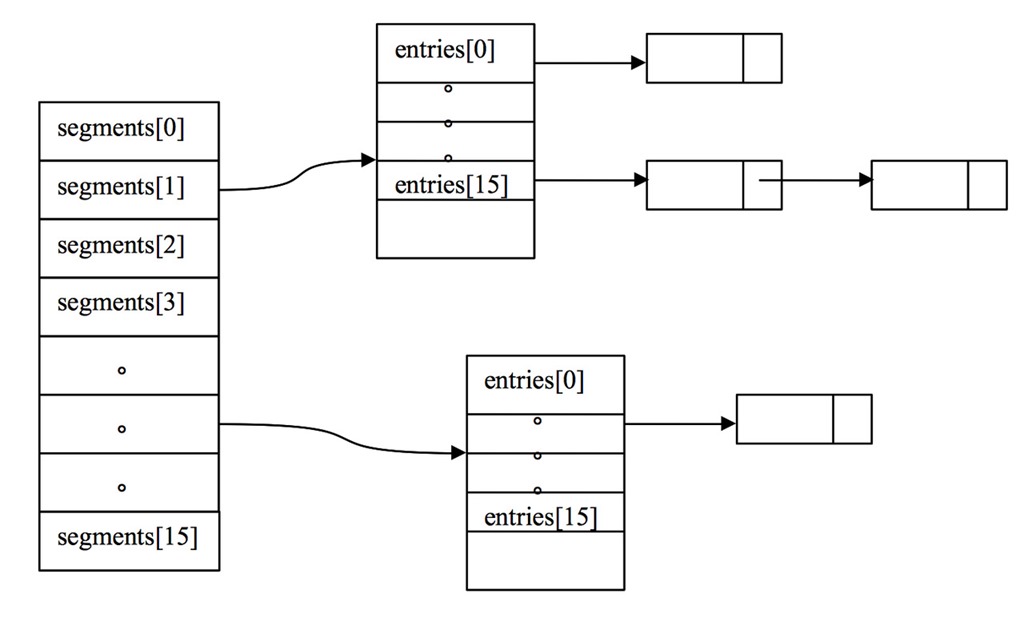
对于哈希表，Java中采用链表的方式来解决hash冲突的。一个HashMap的数据结构(me：hashmap是由数组+链表来实现的)看起来类似下图：

[](http://www.importnew.com/21781.html/3-21)

 实现了同步的HashTable也是这样的结构（me：hashtable底层就是hashmap，然后在相关的方法中做了同步），它的同步是用锁来保证的，并且所有同步操作使用的是同一个锁对象。这样若有n个线程同时在get时，这n个线程要串行的等待来获取锁。

ConcurrentHashMap中对这个数据结构，针对并发稍微做了一点调整。它把区间按照并发级别(concurrentLevel)，分成了若干个segment。默认情况下内部按并发级别为16来创建。对于每个segment的容量，默认情况也是16。当然并发级别(concurrentLevel)和每个段(segment)的初始容量都是可以通过构造函数设定的。

创建好默认的ConcurrentHashMap之后，它的结构大致如下图：

[](http://www.importnew.com/21781.html/4-20)

看起来只是把以前HashTable的一个hash bucket创建了16份而已。有什么特别的吗？没啥特别的。

继续看每个segment是怎么定义的：

|  |  |
| --- | --- |
| 1 | static final class Segment<K,V> extends ReentrantLock implements Serializable |

Segment继承了ReentrantLock，表明每个segment都可以当做一个锁。（ReentrantLock前文已经提到，不了解的话就把当做synchronized的替代者吧）这样对每个segment中的数据需要同步操作的话都是使用每个segment容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的segment。

面的这种做法，就称之为**“分离锁（lock striping）”**。有必要对**“分拆锁”**和**“分离锁”**的概念描述一下：

*分拆锁(lock spliting)就是若原先的程序中多处逻辑都采用同一个锁，但各个逻辑之间又相互独立，就可以拆(Spliting)为使用多个锁，每个锁守护不同的逻辑。 分拆锁有时候可以被扩展，分成可大可小加锁块的集合，并且它们归属于相互独立的对象，这样的情况就是分离锁(lock striping)。（摘自《Java并发编程实践》）*

看上去，单是这样就已经能大大提高多线程并发的性能了。还没完，继续看我们关注的get,put,remove这三个函数怎么保证数据同步的。

先看get方法

|  |  |
| --- | --- |
| 1  2  3  4 | public V get(Object key) {      int hash = hash(key); // throws NullPointerException if key null      return segmentFor(hash).get(key, hash);  } |

它没有使用同步控制，交给segment去找，再看Segment中的get方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | V get(Object key, int hash) {          if (count != 0) { // read-volatile // ①              HashEntry<K,V> e = getFirst(hash);              while (e != null) {                  if (e.hash == hash && key.equals(e.key)) {                      V v = e.value;                      if (v != null)  // ② 注意这里                          return v;                      return readValueUnderLock(e); // recheck                  }                  e = e.next;              }          }          return null;  } |

它也没有使用锁来同步，只是判断获取的entry的value是否为null，为null时才使用加锁的方式再次去获取。

这个实现很微妙，没有锁同步的话，靠什么保证同步呢？我们一步步分析。

第一步，先判断一下 count != 0；count变量表示segment中存在entry的个数。如果为0就不用找了。  
假设这个时候恰好另一个线程put或者remove了这个segment中的一个entry，会不会导致两个线程看到的count值不一致呢？  
看一下count变量的定义： transient volatile int count;  
它使用了volatile来修改。我们前文说过，Java5之后，JMM实现了对volatile的保证：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。  
所以，每次判断count变量的时候，即使恰好其他线程改变了segment也会体现出来。

第二步，获取到要该key所在segment中的索引地址，如果该地址有相同的hash对象，顺着链表一直比较下去找到该entry。当找到entry的时候，先做了一次比较： if(v != null) 我们用红色注释的地方。  
这是为何呢？

考虑一下，如果这个时候，另一个线程恰好新增/删除了entry，或者改变了entry的value，会如何？

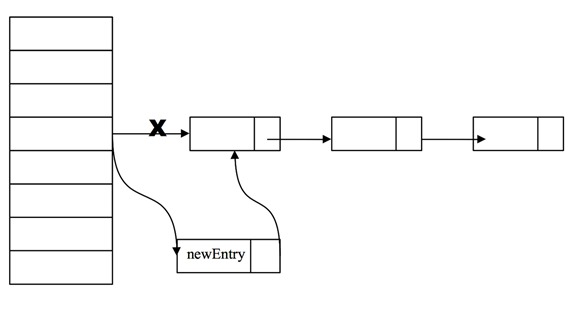
先看一下HashEntry类结构。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static final class HashEntry<K,V> {      final K key;      final int hash;      volatile V value;      final HashEntry<K,V> next;      。。。  } |

除了 value，其它成员都是final修饰的，也就是说value可以被改变，其它都不可以改变，包括指向下一个HashEntry的next也不能被改变。（那删除一个entry时怎么办？后续会讲到。）

**1) 在get代码的①和②之间，另一个线程新增了一个entry**  
如果另一个线程新增的这个entry又恰好是我们要get的，这事儿就比较微妙了。

下图大致描述了put 一个新的entry的过程。

[](http://www.importnew.com/21781.html/5-14)

因为每个HashEntry中的next也是final的，没法对链表最后一个元素增加一个后续entry（me：实际上无法为任何链表上的节点增加后继节点）所以新增一个entry的实现方式只能通过头结点来插入了。

newEntry对象是通过 new HashEntry(K k , V v, HashEntry next) 来创建的。如果另一个线程刚好new 这个对象时，当前线程来get它。因为没有同步，就可能会出现当前线程得到的newEntry对象是一个没有完全构造好的对象引用。

回想一下我们之前讨论的DCL的问题，这里也一样，没有锁同步的话，new 一个对象对于多线程看到这个对象的状态是没有保障的，这里同样有可能一个线程new这个对象的时候还没有执行完构造函数就被另一个线程得到这个对象引用。  
所以才需要判断一下：if (v != null) 如果确实是一个不完整的对象，则使用锁的方式再次get一次。

有没有可能会put进一个value为null的entry？ 不会的，已经做了检查，这种情况会抛出异常，所以 ②处的判断完全是出于对多线程下访问一个new出来的对象的状态检测。

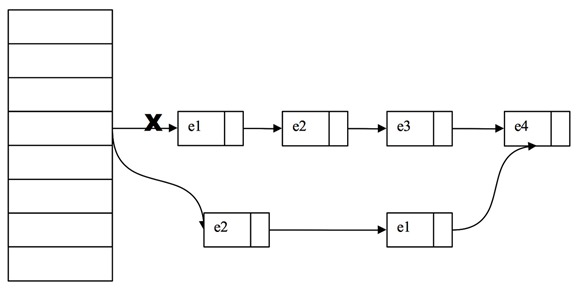
**2) 在get代码的①和②之间，另一个线程修改了一个entry的value**

value是用volitale修饰的，可以保证读取时获取到的是修改后的值。

**3) 在get代码的①之后，另一个线程删除了一个entry**

假设我们的链表元素是：e1-> e2 -> e3 -> e4 我们要删除 e3这个entry，因为HashEntry中next的不可变，所以我们无法直接把e2的next指向e4，而是将要删除的节点之前的节点复制一份，形成新的链表。

它的实现大致如下图所示：

[](http://www.importnew.com/21781.html/6-13)

如果我们get的也恰巧是e3，可能我们顺着链表刚找到e1，这时另一个线程就执行了删除e3的操作，而我们线程还会继续沿着旧的链表找到e3返回。这里没有办法实时保证了。

我们第①处就判断了count变量，它保障了在 ①处能看到其他线程修改后的。①之后到②之间，如果再次发生了其他线程再删除了entry节点，就没法保证看到最新的了。

不过这也没什么关系，即使我们返回e3的时候，它被其他线程删除了，暴露出去的e3也不会对我们新的链表造成影响。

这其实是一种乐观设计，设计者假设 ①之后到②之间 发生被其它线程增、删、改的操作可能性很小，所以不采用同步设计，而是采用了事后（其它线程这期间也来操作，并且可能发生非安全事件）弥补的方式。  
而因为其他线程的“改”和“删”对我们的数据都不会造成影响，所以只有对“新增”操作进行了安全检查，就是②处的非null检查，如果确认不安全事件发生，则采用加锁的方式再次get。

这样做减少了使用互斥锁对并发性能的影响。可能有人怀疑remove操作中复制链表的方式是否代价太大，这里我没有深入比较，不过既然Java5中这么实现，我想new一个对象的代价应该已经没有早期认为的那么严重。

我们基本分析完了get操作。对于put和remove操作，是使用锁同步来进行的，不过是用的ReentrantLock而不是synchronized，性能上要更高一些。它们的实现前文都已经提到过，就没什么可分析的了。

我们还需要知道一点，ConcurrentHashMap的迭代器不是Fast-Fail的方式，所以在迭代的过程中别其他线程添加/删除了元素，不会抛出异常，也不能体现出元素的改动。但也没有关系，因为每个entry的成员除了value都是final修饰的，暴露出去也不会对其他元素造成影响。

加深

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ConcurrentHashMap<String, Boolean> map = new ...;  Thread a = new Thread {      void run() {          map.put("first", true);          map.put("second", true);      }  };    Thread b = new Thread {      void run() {          map.clear();      }  };    a.start();  b.start();  a.join();  b.join(); |

结果：

|  |  |
| --- | --- |
| 1  2  3  4 | Map("first" -> true, "second" -> true)  Map("second" -> true)  Map()  Map("first" -> true) |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | ConcurrentHashMap<String, Boolean> map = new ...;  List<String> myKeys = new ...;    Thread a = new Thread {      void run() {          map.put("first", true);          // more stuff          map.remove("first");          map.put("second", true);      }  };    Thread b = new Thread {      void run() {          Set<String> keys = map.keySet();          for (String key : keys) {              myKeys.add(key);          }      }  };    a.start();  b.start();  a.join();  b.join(); |

结果：

|  |  |
| --- | --- |
| 1  2  3  4 | List()  List("first")  List("second")  List("first", "second") |

解释：  
对于这两个现象的解释：ConcurrentHashMap中的clear方法：

|  |  |
| --- | --- |
| 1  2  3  4 | public void clear() {      for (int i = 0; i < segments.length; ++i)          segments[i].clear();  } |

如果线程b先执行了clear，清空了一部分segment的时候，线程a执行了put且正好把“first”放入了“清空过”的segment中，而把“second”放到了还没有清空过的segment中，就会出现上面的情况。

第二段代码，如果线程b执行了迭代遍历到first，而此时线程a还没有remove掉first，那么即使后续删除了first，迭代器里不会反应出来，也不抛出异常，这种迭代器被称为“弱一致性”(weakly consistent)迭代器。

参考：<http://blog.csdn.net/u010723709/article/details/48007881>

# [ConcurrentHashMap源码分析（JDK8版本）](http://blog.csdn.net/u010723709/article/details/48007881)

2015-08-26 19:10 15049人阅读 [评论](http://blog.csdn.net/u010723709/article/details/48007881#comments)(16) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/u010723709/article/details/48007881#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Java源码（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/u010723709/article/details/48007881)

注：本文源码是JDK8的版本，与之前的版本有较大差异

ConcurrentHashMap是conccurrent家族中的一个类，由于它可以高效地支持并发操作，以及被广泛使用，经典的开源框架Spring的底层数据结构就是使用ConcurrentHashMap实现的。与同是线程安全的老大哥HashTable相比，它已经更胜一筹，因此它的锁更加细化，而不是像HashTable一样为几乎每个方法都添加了synchronized锁，这样的锁无疑会影响到性能。

本文的分析的源码是JDK8的版本，与JDK6的版本有很大的差异。实现线程安全的思想也已经完全变了，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

首先来看几个重要的属性，与HashMap相同的就不再介绍了，这里重点解释一下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。

//=====20171026

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* 盛装Node元素的数组 它的大小是2的整数次幂
3. \* Size is always a power of two. Accessed directly by iterators.
4. \*/
5. **transient** **volatile** Node<K,V>[] table;
7. /\*\*
8. \* Table initialization and resizing control.  When negative, the
9. \* table is being initialized or resized: -1 for initialization,
10. \* else -(1 + the number of active resizing threads).  Otherwise,
11. \* when table is null, holds the initial table size to use upon
12. \* creation, or 0 for default. After initialization, holds the
13. \* next element count value upon which to resize the table.
14. hash表初始化或扩容时的一个控制位标识量。
15. 负数代表正在进行初始化或扩容操作
16. -1代表正在初始化
17. -N 表示有N-1个线程正在进行扩容操作
18. 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
20. \*/
21. **private** **transient** **volatile** **int** sizeCtl;
22. // 以下两个是用来控制扩容的时候 单线程进入的变量
23. /\*\*
24. \* The number of bits used for generation stamp in sizeCtl.
25. \* Must be at least 6 for 32bit arrays.
26. \*/
27. **private** **static** **int** RESIZE\_STAMP\_BITS = 16;
28. /\*\*
29. \* The bit shift for recording size stamp in sizeCtl.
30. \*/
31. **private** **static** **final** **int** RESIZE\_STAMP\_SHIFT = 32 - RESIZE\_STAMP\_BITS;

34. /\*
35. \* Encodings for Node hash fields. See above for explanation.
36. \*/
37. **static** **final** **int** MOVED     = -1; // hash值是-1，表示这是一个forwardNode节点
38. **static** **final** **int** TREEBIN   = -2; // hash值是-2  表示这时一个TreeBin节点

# 2 重要的内部类

## 2.1 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，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。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **static** **class** Node<K,V> **implements** Map.Entry<K,V> {
2. **final** **int** hash;
3. **final** K key;
4. **volatile** V val;//带有同步锁的value
5. **volatile** Node<K,V> next;//带有同步锁的next指针
7. Node(**int** hash, K key, V val, Node<K,V> next) {
8. **this**.hash = hash;
9. **this**.key = key;
10. **this**.val = val;
11. **this**.next = next;
12. }
14. **public** **final** K getKey()       { **return** key; }
15. **public** **final** V getValue()     { **return** val; }
16. **public** **final** **int** hashCode()   { **return** key.hashCode() ^ val.hashCode(); }
17. **public** **final** String toString(){ **return** key + "=" + val; }
18. //不允许直接改变value的值
19. **public** **final** V setValue(V value) {
20. **throw** **new** UnsupportedOperationException();
21. }
23. **public** **final** **boolean** equals(Object o) {
24. Object k, v, u; Map.Entry<?,?> e;
25. **return** ((o **instanceof** Map.Entry) &&
26. (k = (e = (Map.Entry<?,?>)o).getKey()) != **null** &&
27. (v = e.getValue()) != **null** &&
28. (k == key || k.equals(key)) &&
29. (v == (u = val) || v.equals(u)));
30. }
32. /\*\*
33. \* Virtualized support for map.get(); overridden in subclasses.
34. \*/
35. Node<K,V> find(**int** h, Object k) {
36. Node<K,V> e = **this**;
37. **if** (k != **null**) {
38. **do** {
39. K ek;
40. **if** (e.hash == h &&
41. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
42. **return** e;
43. } **while** ((e = e.next) != **null**);
44. }
45. **return** **null**;
46. }
47. }
49. 这个Node内部类与HashMap中定义的Node类很相似，但是有一些差别
50. 它对value和next属性设置了**volatile**同步锁
51. 它不允许调用setValue方法直接改变Node的value域
52. 它增加了find方法辅助map.get()方法

## 2.2 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。

## 2.3 TreeBin

这个类并不负责包装用户的key、value信息，而是包装的很多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。

这里仅贴出它的构造方法。可以看到在构造TreeBin节点时，仅仅指定了它的hash值为TREEBIN常量，这也就是个标识为。同时也看到我们熟悉的红黑树构造方法

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* Creates bin with initial set of nodes headed by b.
3. \*/
4. TreeBin(TreeNode<K,V> b) {
5. **super**(TREEBIN, **null**, **null**, **null**);
6. **this**.first = b;
7. TreeNode<K,V> r = **null**;
8. **for** (TreeNode<K,V> x = b, next; x != **null**; x = next) {
9. next = (TreeNode<K,V>)x.next;
10. x.left = x.right = **null**;
11. **if** (r == **null**) {
12. x.parent = **null**;
13. x.red = **false**;
14. r = x;
15. }
16. **else** {
17. K k = x.key;
18. **int** h = x.hash;
19. Class<?> kc = **null**;
20. **for** (TreeNode<K,V> p = r;;) {
21. **int** dir, ph;
22. K pk = p.key;
23. **if** ((ph = p.hash) > h)
24. dir = -1;
25. **else** **if** (ph < h)
26. dir = 1;
27. **else** **if** ((kc == **null** &&
28. (kc = comparableClassFor(k)) == **null**) ||
29. (dir = compareComparables(kc, k, pk)) == 0)
30. dir = tieBreakOrder(k, pk);
31. TreeNode<K,V> xp = p;
32. **if** ((p = (dir <= 0) ? p.left : p.right) == **null**) {
33. x.parent = xp;
34. **if** (dir <= 0)
35. xp.left = x;
36. **else**
37. xp.right = x;
38. r = balanceInsertion(r, x);
39. **break**;
40. }
41. }
42. }
43. }
44. **this**.root = r;
45. **assert** checkInvariants(root);
46. }

## 2.5 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

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* A node inserted at head of bins during transfer operations.
3. \*/
4. **static** **final** **class** ForwardingNode<K,V> **extends** Node<K,V> {
5. **final** Node<K,V>[] nextTable;
6. ForwardingNode(Node<K,V>[] tab) {
7. **super**(MOVED, **null**, **null**, **null**);
8. **this**.nextTable = tab;
9. }
11. Node<K,V> find(**int** h, Object k) {
12. // loop to avoid arbitrarily deep recursion on forwarding nodes
13. outer: **for** (Node<K,V>[] tab = nextTable;;) {
14. Node<K,V> e; **int** n;
15. **if** (k == **null** || tab == **null** || (n = tab.length) == 0 ||
16. (e = tabAt(tab, (n - 1) & h)) == **null**)
17. **return** **null**;
18. **for** (;;) {
19. **int** eh; K ek;
20. **if** ((eh = e.hash) == h &&
21. ((ek = e.key) == k || (ek != **null** && k.equals(ek))))
22. **return** e;
23. **if** (eh < 0) {
24. **if** (e **instanceof** ForwardingNode) {
25. tab = ((ForwardingNode<K,V>)e).nextTable;
26. **continue** outer;
27. }
28. **else**
29. **return** e.find(h, k);
30. }
31. **if** ((e = e.next) == **null**)
32. **return** **null**;
33. }
34. }
35. }
36. }

# 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。

## 3.1 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。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **private** **static** **final** sun.misc.Unsafe U;
2. **private** **static** **final** **long** SIZECTL;
3. **private** **static** **final** **long** TRANSFERINDEX;
4. **private** **static** **final** **long** BASECOUNT;
5. **private** **static** **final** **long** CELLSBUSY;
6. **private** **static** **final** **long** CELLVALUE;
7. **private** **static** **final** **long** ABASE;
8. **private** **static** **final** **int** ASHIFT;
10. **static** {
11. **try** {
12. U = sun.misc.Unsafe.getUnsafe();
13. Class<?> k = ConcurrentHashMap.**class**;
14. SIZECTL = U.objectFieldOffset
15. (k.getDeclaredField("sizeCtl"));
16. TRANSFERINDEX = U.objectFieldOffset
17. (k.getDeclaredField("transferIndex"));
18. BASECOUNT = U.objectFieldOffset
19. (k.getDeclaredField("baseCount"));
20. CELLSBUSY = U.objectFieldOffset
21. (k.getDeclaredField("cellsBusy"));
22. Class<?> ck = CounterCell.**class**;
23. CELLVALUE = U.objectFieldOffset
24. (ck.getDeclaredField("value"));
25. Class<?> ak = Node[].**class**;
26. ABASE = U.arrayBaseOffset(ak);
27. **int** scale = U.arrayIndexScale(ak);
28. **if** ((scale & (scale - 1)) != 0)
29. **throw** **new** Error("data type scale not a power of two");
30. ASHIFT = 31 - Integer.numberOfLeadingZeros(scale);
31. } **catch** (Exception e) {
32. **throw** **new** Error(e);
33. }
34. }

## 3.2 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。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. @SuppressWarnings("unchecked")
2. //获得在i位置上的Node节点
3. **static** **final** <K,V> Node<K,V> tabAt(Node<K,V>[] tab, **int** i) {
4. **return** (Node<K,V>)U.getObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE);
5. }
6. //利用CAS算法设置i位置上的Node节点。之所以能实现并发是因为他指定了原来这个节点的值是多少
7. //在CAS算法中，会比较内存中的值与你指定的这个值是否相等，如果相等才接受你的修改，否则拒绝你的修改
8. //因此当前线程中的值并不是最新的值，这种修改可能会覆盖掉其他线程的修改结果  有点类似于SVN
9. **static** **final** <K,V> **boolean** casTabAt(Node<K,V>[] tab, **int** i,
10. Node<K,V> c, Node<K,V> v) {
11. **return** U.compareAndSwapObject(tab, ((**long**)i << ASHIFT) + ABASE, c, v);
12. }
13. //利用volatile方法设置节点位置的值
14. **static** **final** <K,V> **void** setTabAt(Node<K,V>[] tab, **int** i, Node<K,V> v) {
15. U.putObjectVolatile(tab, ((**long**)i << ASHIFT) + ABASE, v);
16. }

# 4 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

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* Initializes table, using the size recorded in sizeCtl.
3. \*/
4. **private** **final** Node<K,V>[] initTable() {
5. Node<K,V>[] tab; **int** sc;
6. **while** ((tab = table) == **null** || tab.length == 0) {
7. //sizeCtl表示有其他线程正在进行初始化操作，把线程挂起。对于table的初始化工作，只能有一个线程在进行。
8. **if** ((sc = sizeCtl) < 0)
9. Thread.yield(); // lost initialization race; just spin
10. **else** **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, -1)) {//利用CAS方法把sizectl的值置为-1 表示本线程正在进行初始化
11. **try** {
12. **if** ((tab = table) == **null** || tab.length == 0) {
13. **int** n = (sc > 0) ? sc : DEFAULT\_CAPACITY;
14. @SuppressWarnings("unchecked")
15. Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n];
16. table = tab = nt;
17. sc = n - (n >>> 2);//相当于0.75\*n 设置一个扩容的阈值
18. }
19. } **finally** {
20. sizeCtl = sc;
21. }
22. **break**;
23. }
24. }
25. **return** tab;
26. }

# 5 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

* 第一部分是构建一个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；
* 第二个部分就是将原来table中的元素复制到nextTable中，这里允许多线程进行操作。

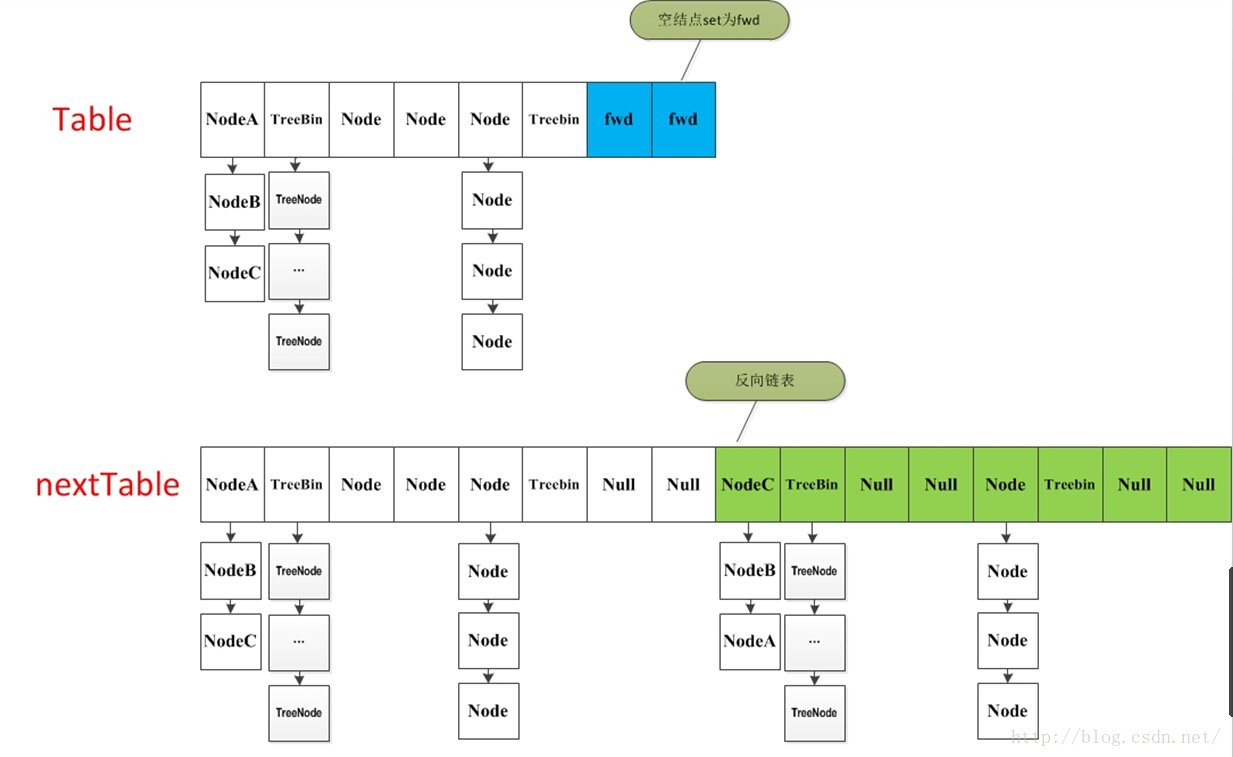
先来看一下单线程是如何完成的：

它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。

再看一下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。



**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* 一个过渡的table表  只有在扩容的时候才会使用
3. \*/
4. **private** **transient** **volatile** Node<K,V>[] nextTable;
6. /\*\*
7. \* Moves and/or copies the nodes in each bin to new table. See
8. \* above for explanation.
9. \*/
10. **private** **final** **void** transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
11. **int** n = tab.length, stride;
12. **if** ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN\_TRANSFER\_STRIDE)
13. stride = MIN\_TRANSFER\_STRIDE; // subdivide range
14. **if** (nextTab == **null**) {            // initiating
15. **try** {
16. @SuppressWarnings("unchecked")
17. Node<K,V>[] nt = (Node<K,V>[])**new** Node<?,?>[n << 1];//构造一个nextTable对象 它的容量是原来的两倍
18. nextTab = nt;
19. } **catch** (Throwable ex) {      // try to cope with OOME
20. sizeCtl = Integer.MAX\_VALUE;
21. **return**;
22. }
23. nextTable = nextTab;
24. transferIndex = n;
25. }
26. **int** nextn = nextTab.length;
27. ForwardingNode<K,V> fwd = **new** ForwardingNode<K,V>(nextTab);//构造一个连节点指针 用于标志位
28. **boolean** advance = **true**;//并发扩容的关键属性 如果等于true 说明这个节点已经处理过
29. **boolean** finishing = **false**; // to ensure sweep before committing nextTab
30. **for** (**int** i = 0, bound = 0;;) {
31. Node<K,V> f; **int** fh;
32. //这个while循环体的作用就是在控制i--  通过i--可以依次遍历原hash表中的节点
33. **while** (advance) {
34. **int** nextIndex, nextBound;
35. **if** (--i >= bound || finishing)
36. advance = **false**;
37. **else** **if** ((nextIndex = transferIndex) <= 0) {
38. i = -1;
39. advance = **false**;
40. }
41. **else** **if** (U.compareAndSwapInt
42. (**this**, TRANSFERINDEX, nextIndex,
43. nextBound = (nextIndex > stride ?
44. nextIndex - stride : 0))) {
45. bound = nextBound;
46. i = nextIndex - 1;
47. advance = **false**;
48. }
49. }
50. **if** (i < 0 || i >= n || i + n >= nextn) {
51. **int** sc;
52. **if** (finishing) {
53. //如果所有的节点都已经完成复制工作  就把nextTable赋值给table 清空临时对象nextTable
54. nextTable = **null**;
55. table = nextTab;
56. sizeCtl = (n << 1) - (n >>> 1);//扩容阈值设置为原来容量的1.5倍  依然相当于现在容量的0.75倍
57. **return**;
58. }
59. //利用CAS方法更新这个扩容阈值，在这里面sizectl值减一，说明新加入一个线程参与到扩容操作
60. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc = sizeCtl, sc - 1)) {
61. **if** ((sc - 2) != resizeStamp(n) << RESIZE\_STAMP\_SHIFT)
62. **return**;
63. finishing = advance = **true**;
64. i = n; // recheck before commit
65. }
66. }
67. //如果遍历到的节点为空 则放入ForwardingNode指针
68. **else** **if** ((f = tabAt(tab, i)) == **null**)
69. advance = casTabAt(tab, i, **null**, fwd);
70. //如果遍历到ForwardingNode节点  说明这个点已经被处理过了 直接跳过  这里是控制并发扩容的核心
71. **else** **if** ((fh = f.hash) == MOVED)
72. advance = **true**; // already processed
73. **else** {
74. //节点上锁
75. **synchronized** (f) {
76. **if** (tabAt(tab, i) == f) {
77. Node<K,V> ln, hn;
78. //如果fh>=0 证明这是一个Node节点
79. **if** (fh >= 0) {
80. **int** runBit = fh & n;
81. //以下的部分在完成的工作是构造两个链表  一个是原链表  另一个是原链表的反序排列
82. Node<K,V> lastRun = f;
83. **for** (Node<K,V> p = f.next; p != **null**; p = p.next) {
84. **int** b = p.hash & n;
85. **if** (b != runBit) {
86. runBit = b;
87. lastRun = p;
88. }
89. }
90. **if** (runBit == 0) {
91. ln = lastRun;
92. hn = **null**;
93. }
94. **else** {
95. hn = lastRun;
96. ln = **null**;
97. }
98. **for** (Node<K,V> p = f; p != lastRun; p = p.next) {
99. **int** ph = p.hash; K pk = p.key; V pv = p.val;
100. **if** ((ph & n) == 0)
101. ln = **new** Node<K,V>(ph, pk, pv, ln);
102. **else**
103. hn = **new** Node<K,V>(ph, pk, pv, hn);
104. }
105. //在nextTable的i位置上插入一个链表
106. setTabAt(nextTab, i, ln);
107. //在nextTable的i+n的位置上插入另一个链表
108. setTabAt(nextTab, i + n, hn);
109. //在table的i位置上插入forwardNode节点  表示已经处理过该节点
110. setTabAt(tab, i, fwd);
111. //设置advance为true 返回到上面的while循环中 就可以执行i--操作
112. advance = **true**;
113. }
114. //对TreeBin对象进行处理  与上面的过程类似
115. **else** **if** (f **instanceof** TreeBin) {
116. TreeBin<K,V> t = (TreeBin<K,V>)f;
117. TreeNode<K,V> lo = **null**, loTail = **null**;
118. TreeNode<K,V> hi = **null**, hiTail = **null**;
119. **int** lc = 0, hc = 0;
120. //构造正序和反序两个链表
121. **for** (Node<K,V> e = t.first; e != **null**; e = e.next) {
122. **int** h = e.hash;
123. TreeNode<K,V> p = **new** TreeNode<K,V>
124. (h, e.key, e.val, **null**, **null**);
125. **if** ((h & n) == 0) {
126. **if** ((p.prev = loTail) == **null**)
127. lo = p;
128. **else**
129. loTail.next = p;
130. loTail = p;
131. ++lc;
132. }
133. **else** {
134. **if** ((p.prev = hiTail) == **null**)
135. hi = p;
136. **else**
137. hiTail.next = p;
138. hiTail = p;
139. ++hc;
140. }
141. }
142. //如果扩容后已经不再需要tree的结构 反向转换为链表结构
143. ln = (lc <= UNTREEIFY\_THRESHOLD) ? untreeify(lo) :
144. (hc != 0) ? **new** TreeBin<K,V>(lo) : t;
145. hn = (hc <= UNTREEIFY\_THRESHOLD) ? untreeify(hi) :
146. (lc != 0) ? **new** TreeBin<K,V>(hi) : t;
147. //在nextTable的i位置上插入一个链表
148. setTabAt(nextTab, i, ln);
149. //在nextTable的i+n的位置上插入另一个链表
150. setTabAt(nextTab, i + n, hn);
151. //在table的i位置上插入forwardNode节点  表示已经处理过该节点
152. setTabAt(tab, i, fwd);
153. //设置advance为true 返回到上面的while循环中 就可以执行i--操作
154. advance = **true**;
155. }
156. }
157. }
158. }
159. }
160. }

# 6 Put方法

前面的所有的介绍其实都为这个方法做铺垫。ConcurrentHashMap最常用的就是put和get两个方法。现在来介绍put方法，这个put方法依然沿用HashMap的put方法的思想，根据hash值计算这个新插入的点在table中的位置i，如果i位置是空的，直接放进去，否则进行判断，如果i位置是树节点，按照树的方式插入新的节点，否则把i插入到链表的末尾。ConcurrentHashMap中依然沿用这个思想，有一个最重要的不同点就是ConcurrentHashMap不允许key或value为null值。另外由于涉及到多线程，put方法就要复杂一点。在多线程中可能有以下两个情况

1. 如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就帮助进行扩容；
2. 如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。尽管这个有一些影响效率，但是还是会比hashTable的synchronized要好得多。

整体流程就是首先定义不允许key或value为null的情况放入  对于每一个放入的值，首先利用spread方法对key的hashcode进行一次hash计算，由此来确定这个值在table中的位置。

如果这个位置是空的，那么直接放入，而且不需要加锁操作。

    如果这个位置存在结点，说明发生了hash碰撞，首先判断这个节点的类型。如果是链表节点（fh>0）,则得到的结点就是hash值相同的节点组成的链表的头节点。需要依次向后遍历确定这个新加入的值所在位置。如果遇到hash值与key值都与新加入节点是一致的情况，则只需要更新value值即可。否则依次向后遍历，直到链表尾插入这个结点。  如果加入这个节点以后链表长度大于8，就把这个链表转换成红黑树。如果这个节点的类型已经是树节点的话，直接调用树节点的插入方法进行插入新的值。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **public** V put(K key, V value) {
2. **return** putVal(key, value, **false**);
3. }
5. /\*\* Implementation for put and putIfAbsent \*/
6. **final** V putVal(K key, V value, **boolean** onlyIfAbsent) {
7. //不允许 key或value为null
8. **if** (key == **null** || value == **null**) **throw** **new** NullPointerException();
9. //计算hash值
10. **int** hash = spread(key.hashCode());
11. **int** binCount = 0;
12. //死循环 何时插入成功 何时跳出
13. **for** (Node<K,V>[] tab = table;;) {
14. Node<K,V> f; **int** n, i, fh;
15. //如果table为空的话，初始化table
16. **if** (tab == **null** || (n = tab.length) == 0)
17. tab = initTable();
18. //根据hash值计算出在table里面的位置
19. **else** **if** ((f = tabAt(tab, i = (n - 1) & hash)) == **null**) {
20. //如果这个位置没有值 ，直接放进去，不需要加锁
21. **if** (casTabAt(tab, i, **null**,
22. **new** Node<K,V>(hash, key, value, **null**)))
23. **break**;                   // no lock when adding to empty bin
24. }
25. //当遇到表连接点时，需要进行整合表的操作
26. **else** **if** ((fh = f.hash) == MOVED)
27. tab = helpTransfer(tab, f);
28. **else** {
29. V oldVal = **null**;
30. //结点上锁  这里的结点可以理解为hash值相同组成的链表的头结点
31. **synchronized** (f) {
32. **if** (tabAt(tab, i) == f) {
33. //fh〉0 说明这个节点是一个链表的节点 不是树的节点
34. **if** (fh >= 0) {
35. binCount = 1;
36. //在这里遍历链表所有的结点
37. **for** (Node<K,V> e = f;; ++binCount) {
38. K ek;
39. //如果hash值和key值相同  则修改对应结点的value值
40. **if** (e.hash == hash &&
41. ((ek = e.key) == key ||
42. (ek != **null** && key.equals(ek)))) {
43. oldVal = e.val;
44. **if** (!onlyIfAbsent)
45. e.val = value;
46. **break**;
47. }
48. Node<K,V> pred = e;
49. //如果遍历到了最后一个结点，那么就证明新的节点需要插入 就把它插入在链表尾部
50. **if** ((e = e.next) == **null**) {
51. pred.next = **new** Node<K,V>(hash, key,
52. value, **null**);
53. **break**;
54. }
55. }
56. }
57. //如果这个节点是树节点，就按照树的方式插入值
58. **else** **if** (f **instanceof** TreeBin) {
59. Node<K,V> p;
60. binCount = 2;
61. **if** ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
62. value)) != **null**) {
63. oldVal = p.val;
64. **if** (!onlyIfAbsent)
65. p.val = value;
66. }
67. }
68. }
69. }
70. **if** (binCount != 0) {
71. //如果链表长度已经达到临界值8 就需要把链表转换为树结构
72. **if** (binCount >= TREEIFY\_THRESHOLD)
73. treeifyBin(tab, i);
74. **if** (oldVal != **null**)
75. **return** oldVal;
76. **break**;
77. }
78. }
79. }
80. //将当前ConcurrentHashMap的元素数量+1
81. addCount(1L, binCount);
82. **return** **null**;
83. }

## 6.1 helpTransfer方法

这是一个协助扩容的方法。这个方法被调用的时候，当前ConcurrentHashMap一定已经有了nextTable对象，首先拿到这个nextTable对象，调用transfer方法。回看上面的transfer方法可以看到，当本线程进入扩容方法的时候会直接进入复制阶段。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* Helps transfer if a resize is in progress.
3. \*/
4. **final** Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
5. Node<K,V>[] nextTab; **int** sc;
6. **if** (tab != **null** && (f **instanceof** ForwardingNode) &&
7. (nextTab = ((ForwardingNode<K,V>)f).nextTable) != **null**) {
8. **int** rs = resizeStamp(tab.length);//计算一个操作校验码
9. **while** (nextTab == nextTable && table == tab &&
10. (sc = sizeCtl) < 0) {
11. **if** ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||
12. sc == rs + MAX\_RESIZERS || transferIndex <= 0)
13. **break**;
14. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, sc + 1)) {
15. transfer(tab, nextTab);
16. **break**;
17. }
18. }
19. **return** nextTab;
20. }
21. **return** table;
22. }

## 6.2 treeifyBin方法

这个方法用于将过长的链表转换为TreeBin对象。但是他并不是直接转换，而是进行一次容量判断，如果容量没有达到转换的要求，直接进行扩容操作并返回；如果满足条件才链表的结构抓换为TreeBin ，这与HashMap不同的是，它并没有把TreeNode直接放入红黑树，而是利用了TreeBin这个小容器来封装所有的TreeNode.

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **private** **final** **void** treeifyBin(Node<K,V>[] tab, **int** index) {
2. Node<K,V> b; **int** n, sc;
3. **if** (tab != **null**) {
4. **if** ((n = tab.length) < MIN\_TREEIFY\_CAPACITY)//如果table.length<64 就扩大一倍 返回
5. tryPresize(n << 1);
6. **else** **if** ((b = tabAt(tab, index)) != **null** && b.hash >= 0) {
7. **synchronized** (b) {
8. **if** (tabAt(tab, index) == b) {
9. TreeNode<K,V> hd = **null**, tl = **null**;
10. //构造了一个TreeBin对象 把所有Node节点包装成TreeNode放进去
11. **for** (Node<K,V> e = b; e != **null**; e = e.next) {
12. TreeNode<K,V> p =
13. **new** TreeNode<K,V>(e.hash, e.key, e.val,
14. **null**, **null**);//这里只是利用了TreeNode封装 而没有利用TreeNode的next域和parent域
15. **if** ((p.prev = tl) == **null**)
16. hd = p;
17. **else**
18. tl.next = p;
19. tl = p;
20. }
21. //在原来index的位置 用TreeBin替换掉原来的Node对象
22. setTabAt(tab, index, **new** TreeBin<K,V>(hd));
23. }
24. }
25. }
26. }
27. }

# 7 get方法

get方法比较简单，给定一个key来确定value的时候，必须满足两个条件  key相同  hash值相同，对于节点可能在链表或树上的情况，需要分别去查找.

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **public** V get(Object key) {
2. Node<K,V>[] tab; Node<K,V> e, p; **int** n, eh; K ek;
3. //计算hash值
4. **int** h = spread(key.hashCode());
5. //根据hash值确定节点位置
6. **if** ((tab = table) != **null** && (n = tab.length) > 0 &&
7. (e = tabAt(tab, (n - 1) & h)) != **null**) {
8. //如果搜索到的节点key与传入的key相同且不为null,直接返回这个节点
9. **if** ((eh = e.hash) == h) {
10. **if** ((ek = e.key) == key || (ek != **null** && key.equals(ek)))
11. **return** e.val;
12. }
13. //如果eh<0 说明这个节点在树上 直接寻找
14. **else** **if** (eh < 0)
15. **return** (p = e.find(h, key)) != **null** ? p.val : **null**;
16. //否则遍历链表 找到对应的值并返回
17. **while** ((e = e.next) != **null**) {
18. **if** (e.hash == h &&
19. ((ek = e.key) == key || (ek != **null** && key.equals(ek))))
20. **return** e.val;
21. }
22. }
23. **return** **null**;
24. }

# 8 Size相关的方法

对于ConcurrentHashMap来说，这个table里到底装了多少东西其实是个不确定的数量，因为不可能在调用size()方法的时候像GC的“stop the world”一样让其他线程都停下来让你去统计，因此只能说这个数量是个估计值。对于这个估计值，ConcurrentHashMap也是大费周章才计算出来的。

## 8.1 辅助定义

为了统计元素个数，ConcurrentHashMap定义了一些变量和一个内部类

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. /\*\*
2. \* A padded cell for distributing counts.  Adapted from LongAdder
3. \* and Striped64.  See their internal docs for explanation.
4. \*/
5. @sun.misc.Contended **static** **final** **class** CounterCell {
6. **volatile** **long** value;
7. CounterCell(**long** x) { value = x; }
8. }
10. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
12. /\*\*
13. \* 实际上保存的是hashmap中的元素个数  利用CAS锁进行更新
14. 但它并不用返回当前hashmap的元素个数
16. \*/
17. **private** **transient** **volatile** **long** baseCount;
18. /\*\*
19. \* Spinlock (locked via CAS) used when resizing and/or creating CounterCells.
20. \*/
21. **private** **transient** **volatile** **int** cellsBusy;
23. /\*\*
24. \* Table of counter cells. When non-null, size is a power of 2.
25. \*/
26. **private** **transient** **volatile** CounterCell[] counterCells;

## 8.2 mappingCount与Size方法

mappingCount与size方法的类似  从Java工程师给出的注释来看，应该使用mappingCount代替size方法 两个方法都没有直接返回basecount 而是统计一次这个值，而这个值其实也是一个大概的数值，因此可能在统计的时候有其他线程正在执行插入或删除操作。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **public** **int** size() {
2. **long** n = sumCount();
3. **return** ((n < 0L) ? 0 :
4. (n > (**long**)Integer.MAX\_VALUE) ? Integer.MAX\_VALUE :
5. (**int**)n);
6. }
7. /\*\*
8. \* Returns the number of mappings. This method should be used
9. \* instead of {@link #size} because a ConcurrentHashMap may
10. \* contain more mappings than can be represented as an int. The
11. \* value returned is an estimate; the actual count may differ if
12. \* there are concurrent insertions or removals.
13. \*
14. \* @return the number of mappings
15. \* @since 1.8
16. \*/
17. **public** **long** mappingCount() {
18. **long** n = sumCount();
19. **return** (n < 0L) ? 0L : n; // ignore transient negative values
20. }
22. **final** **long** sumCount() {
23. CounterCell[] as = counterCells; CounterCell a;
24. **long** sum = baseCount;
25. **if** (as != **null**) {
26. **for** (**int** i = 0; i < as.length; ++i) {
27. **if** ((a = as[i]) != **null**)
28. sum += a.value;//所有counter的值求和
29. }
30. }
31. **return** sum;
32. }

## 8.3 addCount方法

在put方法结尾处调用了addCount方法，把当前ConcurrentHashMap的元素个数+1这个方法一共做了两件事,更新baseCount的值，检测是否进行扩容。

**[java]** [view plain](http://blog.csdn.net/u010723709/article/details/48007881) [copy](http://blog.csdn.net/u010723709/article/details/48007881)

1. **private** **final** **void** addCount(**long** x, **int** check) {
2. CounterCell[] as; **long** b, s;
3. //利用CAS方法更新baseCount的值
4. **if** ((as = counterCells) != **null** ||
5. !U.compareAndSwapLong(**this**, BASECOUNT, b = baseCount, s = b + x)) {
6. CounterCell a; **long** v; **int** m;
7. **boolean** uncontended = **true**;
8. **if** (as == **null** || (m = as.length - 1) < 0 ||
9. (a = as[ThreadLocalRandom.getProbe() & m]) == **null** ||
10. !(uncontended =
11. U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
12. fullAddCount(x, uncontended);
13. **return**;
14. }
15. **if** (check <= 1)
16. **return**;
17. s = sumCount();
18. }
19. //如果check值大于等于0 则需要检验是否需要进行扩容操作
20. **if** (check >= 0) {
21. Node<K,V>[] tab, nt; **int** n, sc;
22. **while** (s >= (**long**)(sc = sizeCtl) && (tab = table) != **null** &&
23. (n = tab.length) < MAXIMUM\_CAPACITY) {
24. **int** rs = resizeStamp(n);
25. //
26. **if** (sc < 0) {
27. **if** ((sc >>> RESIZE\_STAMP\_SHIFT) != rs || sc == rs + 1 ||
28. sc == rs + MAX\_RESIZERS || (nt = nextTable) == **null** ||
29. transferIndex <= 0)
30. **break**;
31. //如果已经有其他线程在执行扩容操作
32. **if** (U.compareAndSwapInt(**this**, SIZECTL, sc, sc + 1))
33. transfer(tab, nt);
34. }
35. //当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
36. **else** **if** (U.compareAndSwapInt(**this**, SIZECTL, sc,
37. (rs << RESIZE\_STAMP\_SHIFT) + 2))
38. transfer(tab, **null**);
39. s = sumCount();
40. }
41. }
42. }

（未完待续。。。）