上善若水

In general the OO style is to use a lot of little objects with a lot of little methods that give us a lot of plug points for overriding and variation. To do is to be -Nietzsche, To bei is to do -Kant, Do be do be do -Sinatra

BlogJava 首页 新随笔 新文章 联系 聚合 管理

≤ 2013年10月 ≥

日	一	二	三	四	五	六
29	30	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	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

常用链接

- 我的随笔
- 我的文章
- 我的评论
- 我的参与
- 最新评论

留言簿(2)

- 给我留言
- 查看公开留言
- 查看私人留言

随笔分类(157)

- Architecture(7)
- Cassandra

Java Core系列之ConcurrentHashMap实现(JDK 1.7)

ConcurrentHashMap类似Hashtable，是HashMap更高效的线程安全版本的实现。不同于Hashtable简单的将所有方法标记为synchronized，它将内部数组分成多个Segment，每个Segment类似一个Hashtable，从而减少锁的粒度，并且它内部有一些比较tricky实现，让get操作很多时候甚至不需要锁（本文代码基于JDK 1.7，它在JDK 1.6的基础上做了进一步的优化，想要看JDK 1.6的实现，可以参考http://www.iteye.com/topic/344876以及http://www.ibm.com/developerworks/cn/java/j-jtp08223/。并且个人感觉这篇文章已经说的很好了，原本我没必要再写，继续决定写只是为了加深自己的印象。这是我的写之前的原话，当时当我写完以后，我发现我还是有必要写这一篇文章的，因为JDK 1.7版本和1.6版本的ConcurrentHashMap的实现已经有了一些优化，我在中间做了一些比较，通过这些比较，我们可以清楚的看到一些这些代码的演化过程，也为我们在自己的项目里做一些优化提供参考，虽然如我推荐的第一篇文章中的末尾写道：ConcurrentHashMap的实现值得学习，不值得模仿。我表示赞同，因为你如果有一点没考虑好的话，就容易出错，在性能和正确性上，我们当然选择正确性。。。。）。当时也要注意，虽然ConcurrentHashMap在性能上比Hashtable提高了很多，但是它也有它自己的限制：

1. 它没有一个基于整个Map的锁，因而如果需要基于整个Map做操作，则需要自己额外的在外层套锁。当然由于它的线程安全特性，你可以不额外加锁，因为你在遍历的时候可以继续添加、删除，然而有些时候这并不符合你的需求（具体理由参考第2点）。
2. 它实现了一种更加细粒度的happens-before的关系。由于上述提到的当一个线程在遍历时，可以有其他线程同时在做添加、删除等操作。而有些情况下，真实需求时当一个遍历发生在添加、删除中间时，该遍历线程应该等到添加、删除线程完成后再开始，从而遍历线程能看到添加、删除后的结果；同样对有线程正在遍历时，希望其他的添加、删除线程能够等待。虽然这种细微的差别好像对一般程序来手影响不大。

HashEntry定义

类似HashMap，Segment中内部数组的每一项都是一个单项链节点，它包含了key、hash、value等信息：

CodeTools(5) XML
Core Java(22) XML
Database(3) XML
EHCache(6) XML
GC(4) XML
GemFire(3) XML
Guava(2) XML
Hadoop(4) XML
HBase(9) XML
Jetty(19) XML
JUnit(6) XML
Linux(5) XML
Logging(11) XML
MultiThreading(8) XML
Netty(2) XML
Protobuf(3) XML
Spring(2) XML
学习积累(6) XML
收藏(12) XML
深入JVM(9) XML
深入源码(2) XML
经验积累(6) XML
金融知识(1) XML

随笔档案(125)

2015年11月 (1)
2015年9月 (5)
2015年8月 (7)
2015年7月 (1)
2015年4月 (6)
2015年2月 (1)
2015年1月 (1)
2014年12月 (2)
2014年7月 (3)
2014年6月 (2)
2014年5月 (15)
2014年4月 (2)
2014年3月 (5)
2013年12月 (2)
2013年11月 (3)
2013年10月 (8)
2012年12月 (2)
2012年11月 (13)
2012年9月 (1)
2012年7月 (6)
2012年6月 (2)
2012年5月 (6)
2012年4月 (1)
2012年2月 (1)
2011年12月 (2)
2011年11月 (3)

```
static final class HashEntry<K,V> {
    final int hash;
    final K key;
    volatile V value;
    volatile HashEntry<K,V> next;

    HashEntry(int hash, K key, V value, HashEntry<K,V> next) {
      this.hash = hash;
      this.key = key;
      this.value = value;
      this.next = next;
    }

    final void setNext(HashEntry<K,V> n) {
      UNSAFE.putOrderedObject(this, nextOffset, n);
    }
  }
```

在JDK 1.6中，HashEntry中的next指针也定义为final，并且每次插入将新添加节点作为链的头节点（同HashMap实现），而且每次删除一个节点时，会将删除节点之前的所有节点拷贝一份组成一个新的链，而将当前节点的上一个节点的next指向当前节点的下一个节点，从而在删除以后有两条链存在，因而可以保证即使在同一条链中，有一个线程在删除，而另一个线程在遍历，它们都能工作良好，因为遍历的线程能继续使用原有的链。因而这种实现是一种更加细粒度的happens-before关系，即如果遍历线程在删除线程结束后开始，则它能看到删除后的变化，如果它发生在删除线程正在执行中间，则它会使用原有的链，而不会等到删除线程结束后再执行，即看不到删除线程的影响。如果这不符合你的需求，还是乖乖的用Hashtable或HashMap的synchronized版本，Collections.synchronizedMap()做的包装。

另一个不同于1.6版本中的实现是它提供setNext()方法，而且这个方法调用了Unsafe类中的putOrderedObject()方法，该方法只对volatile字段有用，关于这个方法的解释如下：

> Sets the value of the object field at the specified offset in the supplied object to the given value. This is an ordered or lazy version of putObjectVolatile(Object,long,Object), which doesn't guarantee the immediate visibility of the change to other threads. It is only really useful where the object field is volatile, and is thus expected to change unexpectedly.

我对这个函数的理解：对volatile字段，按规范，在每次向它写入值后，它更新后的值立即对其他线程可见（可以简单的认为对volatile字段，每次读取它的值时都直接从内存中读取，而不会读缓存中的数据，如CPU的缓存；对写入操作也是直接写入内存），而这个函数可以提供一种选择，即使对volatile字段的写操作，我们也可以使用该方法将它作为一种普通字段来对待。这里setNext()方法的存在是为了在remove时不需要做拷贝额外链进行的优化，具体可以参看remove操作。

Segment中的put操作

在之前的JDK版本中，Segment的put操作开始时就会先加锁，直到put完成才解锁。在JDK 1.7中采用了自旋的机制，进一步减少了加锁的可能性。

```
  static final <K,V> void setEntryAt(HashEntry<K,V>[] tab, int i, HashEntry<K,V> e) {
      UNSAFE.putOrderedObject(tab, ((long)i << TSHIFT) + TBASE, e);
```

2011年9月 (10)
2011年8月 (1)
2011年7月 (6)
2011年6月 (7)

收藏夹(13)

DataBase(7) XML
Financial XML
Tools(6) XML

Java GC

GC - Jon Masamitsu @Oracle
GC - Poonam @Oracle
GC/Arch - Alexey Ragozin
Hotspot John Rose @Oracle
itcamel

Java General

Concurrent - xylz
ImportNew
landon

NoSQL

Carol McDonald @MapR
HBASE HADOOP INSTRUCTOR

Tech General

Learning Library @Oracle
Tech resource from Oracle
Quasar
王垠

Tech Master

Doug Lea @State University of New York
concurrent package author
Leslie Lamport
Distributed System, PAXOS author
MySQL-Performance
MySQL/MongoDB/MariaDB
Welsh, Matt @ Harvard
SEDA
董的博客-大数据相关
董的博客-大数据相关

最新随笔

1. 使用NamedParameterJdbcTemplate遇到无法使用的坑

```
  }
  static final <K,V> HashEntry<K,V> entryAt(HashEntry<K,V>[] tab, int i) {
    return (tab == null) ? null : (HashEntry<K,V>) UNSAFE.getObjectVolatile(tab, ((long)i << TSHIFT) + TBASE);
  }
    final V put(K key, int hash, V value, boolean onlyIfAbsent) {
      HashEntry<K,V> node = tryLock() ? null : scanAndLockForPut(key, hash, value);
      V oldValue;
      try {
        HashEntry<K,V>[] tab = table;
        int index = (tab.length - 1) & hash;
        HashEntry<K,V> first = entryAt(tab, index);
        for (HashEntry<K,V> e = first;;) {
          if (e != null) {
            K k;
            if ((k = e.key) == key || (e.hash == hash && key.equals(k))) {
              oldValue = e.value;
              if (!onlyIfAbsent) {
                e.value = value;
                ++modCount;
              }
              break;
            }
            e = e.next;
          }
          else {
            if (node != null)
              node.setNext(first);
            else
              node = new HashEntry<K,V>(hash, key, value, first);
            int c = count + 1;
            if (c > threshold && tab.length < MAXIMUM_CAPACITY)
              rehash(node);
            else
              setEntryAt(tab, index, node);
            ++modCount;
            count = c;
            oldValue = null;
            break;
          }
        }
      } finally {
        unlock();
      }
      return oldValue;
    }
```

先不考虑自旋等待的问题，假如put一开始就拿到锁，那么它会执行以下逻辑：

2. SSTable详解
3. [转]高性能IO模型浅析
4. Netty3架构解析
5. Intercepting Filter模式详解
6. Reactor模式详解
7. 深入HBase架构解析（二）
8. 深入HBase架构解析（一）
9. Log4J引起的程序“装死”
10. 实现自己的Lock对象

搜索

搜索

积分与排名

积分 - 525317
排名 - 75

最新评论 XML

1. re: 深入源码之SLF4J
评论内容较长,点击标题查看
--Rookie
2. re: Reactor模式详解
netty的Selector.select是使用系统的select实现的么？这个好像不大好，印象中好像是用epoll实现的？
--戈风
3. re: 【转】关于AccessController.doPrivileged
其实具体什么情况下需要用到AccessController.doPrivileged，平常业务代码基本不会用到，难道是在写框架的时候需要用到？什么类型的框架和场景？
--Kingson
4. re: 使用XStream序列化、反序列化XML数据时遇到的各种问题
继承上，子类对象 在toxml时, XStream只将子类的属性转xml了， 父类的属性没转换xml为何呢？
--MR熊
5. re: 深入Jetty源码之SecurityHandler
我想问一下，你这个有没有样例可以参考一下，我刚入jetty不久，对于这个LoginService的配置不是很熟悉，可以介绍一下，邮箱1204219804@qq.com
--小zhao

阅读排行榜

1. Reactor模式详解(29211)
2. 深入HBase架构解析（一）(19918)

1. 根据之前计算出来的hash值找到数组相应bucket中的第一个链节点。这里需要注意的是：
   a. 因为ConcurrentHashMap在计算Segment中数组长度时会保证该值是2的倍数，而且Segment在做rehash时也是每次增长一倍，因而数组索引只做"(tab.length - 1) & hash"计算即可。
   b. 因为table字段时一个volatile变量，因而在开始时将该引用赋值给tab变量，可以减少在直接引用table字段时，因为该字段是volatile而不能做优化带来的损失，因为将table引用赋值给局不变量后就可以把它左右普通变量以实现编译、运行时的优化。
   c. 因为之前已经将volatile的table字段引用赋值给tab局不变量了，为了保证每次读取的table中的数组项都是最新的值，因而调用entryAt()方法获取数组项的值而不是通过tab[index]方式直接获取（在put操作更新节点链时，它采用Unsafe.putOrderedObject()操作，此时它对链头的更新只局限与当前线程，为了保证接下来的put操作能够读取到上一次的更新结果，需要使用volatile的语法去读取节点链的链头）。
2. 遍历数组项中的节点链，如果在节点中能找到key相等的节点，并且当前是put()操作而不是putIfAbsent()操作，纪录原来的值，更新该节点的值，并退出循环，put()操作完成。
3. 如果在节点链中没有找到key相等的节点，创建一个新的节点，并将该节点作为当前链头插入当前链，并将count加1。和读取节点链连头想法，这里使用setEntryAt()操作以实现对链头的延时写，以提升性能，因为此时并不需要将该更新写入到内存，而在锁退出后该更新自然会写入内存[参考**Java**的内存模型，注**1**]。然后当节点数操作阀值(capacity*loadFactor)，而数组长度没有达到最大数组长度，会做rehash。另外，如果**scanAndLockForPut**()操作返回了一个非空HashEntry，则表示在**scanAndLockForPut**()遍历key对应节点链时没有找到相应的节点，此时很多时候需要创建新的节点，因而它预创建HashEntry节点（预创建时因为有些时候它确实不需要再创建），所以不需要再创建，只需要更新它的next指针即可，这里使用setNext()实现延时写也时为了提升性能，因为当前修改并不需要让其他线程知道，在锁退出时修改自然会更新到内存中，如果采用直接赋值给next字段，由于next时volatile字段，会引起更新直接写入内存而增加开销。

Segment中的**scanAndLockForPut**操作

如put源码所示，当put操作尝试加锁没成功时，它不是直接进入等待状态，而是调用了**scanAndLockForPut**()操作，该操作持续查找key对应的节点链中是已存在该机节点，如果没有找到已存在的节点，则预创建一个新节点，并且尝试n次，直到尝试次数操作限制，才真正进入等待状态，计所谓的自旋等待。对最大尝试次数，目前的实现单核次数为1，多核为64：

```
    private HashEntry<K,V> scanAndLockForPut(K key, int hash, V value) {
        HashEntry<K,V> first = entryForHash(this, hash);
        HashEntry<K,V> e = first;
        HashEntry<K,V> node = null;
        int retries = -1; // negative while locating node
        while (!tryLock()) {
            HashEntry<K,V> f; // to recheck first below
            if (retries < 0) {
                if (e == null) {
                    if (node == null) // speculatively create node
```

3. DOM树节点解析(17831)
4. Java Cache系列之Guava Cache实现详解(15862)
5. 使用XStream序列化、反序列化XML数据时遇到的各种问题(14685)

评论排行榜

1. equals方法实现小记(10)
2. 深入JUnit源码之Runner(8)
3. [多问几个为什么]为什么匿名内部类中引用的局部变量和参数需要final而成员字段不用？(8)
4. 深入Log4J源码之Log4J Core(5)
5. finally知多少(5)

```
                node = new HashEntry<K,V>(hash, key, value, null);
              retries = 0;
            }
            else if (key.equals(e.key))
              retries = 0;
            else
              e = e.next;
          }
          else if (++retries > MAX_SCAN_RETRIES) {
            lock();
            break;
          }
          else if ((retries & 1) == 0 && (f = entryForHash(this, hash)) != fi
rst) {
            e = first = f; // re-traverse if entry changed
            retries = -1;
          }
        }
        return node;
      }
```

在这段逻辑中，它先获取key对应的节点链的头，然后持续遍历该链，如果节点链中不存在要插入的节点，则预创建一个节点，否则retries值资增，直到操作最大尝试次数而进入等待状态。这里需要注意最后一个else if中的逻辑：当在自旋过程中发现节点链的链头发生了变化，则更新节点链的链头，并重置retries值为－1，重新为尝试获取锁而自旋遍历。

Segment中的**rehash**操作

rehash的逻辑比较简单，它创建一个大原来两倍容量的数组，然后遍历原来数组以及数组项中的每条链，对每个节点重新计算它的数组索引，然后创建一个新的节点插入到新数组中，这里需要重新创建一个新节点而不是修改原有节点的next指针时为了在做rehash时可以保证其他线程的get遍历操作可以正常在原有的链上正常工作，有点copy-on-write思想。然而Doug Lea继续优化了这段逻辑，为了减少重新创建新节点的开销，这里做了两点优化：1，对只有一个节点的链，直接将该节点赋值给新数组对应项即可（之所以能这么做是因为Segment中数组的长度也永远是2的倍数，而将数组长度扩大成原来的2倍，那么新节点在新数组中的位置只能是相同的索引号或者原来索引号加原来数组的长度，因而可以保证每条链在rehash是不会相互干扰）；2，对有多个节点的链，先遍历该链找到第一个后面所有节点的索引值不变的节点p，然后只重新创建节点p以前的节点即可，此时新节点链和旧节点链同时存在，在p节点相遇，这样即使有其他线程在当前链做遍历也能正常工作：

```
    private void rehash(HashEntry<K,V> node) {
      HashEntry<K,V>[] oldTable = table;
      int oldCapacity = oldTable.length;
      int newCapacity = oldCapacity << 1;
      threshold = (int)(newCapacity * loadFactor);
      HashEntry<K,V>[] newTable = (HashEntry<K,V>[]) new HashE
ntry[newCapacity];
      int sizeMask = newCapacity - 1;
      for (int i = 0; i < oldCapacity ; i++) {
        HashEntry<K,V> e = oldTable[i];
```

```
            if (e != null) {
              HashEntry<K,V> next = e.next;
              int idx = e.hash & sizeMask;
              if (next == null)   //  Single node on list
                newTable[idx] = e;
              else { // Reuse consecutive sequence at same slot
                HashEntry<K,V> lastRun = e;
                int lastIdx = idx;
                for (HashEntry<K,V> last = next; last != null; last = last.ne
xt) {
                  int k = last.hash & sizeMask;
                  if (k != lastIdx) {
                    lastIdx = k;
                    lastRun = last;
                  }
                }
                newTable[lastIdx] = lastRun;
                // Clone remaining nodes
                for (HashEntry<K,V> p = e; p != lastRun; p = p.next) {
                  V v = p.value;
                  int h = p.hash;
                  int k = h & sizeMask;
                  HashEntry<K,V> n = newTable[k];
                  newTable[k] = new HashEntry<K,V>(h, p.key, v, n);
                }
              }
            }
          }
          int nodeIndex = node.hash & sizeMask; // add the new node
          node.setNext(newTable[nodeIndex]);
          newTable[nodeIndex] = node;
          table = newTable;
        }
```

Segment中的**remove**操作

在JDK 1.6版本中，remove操作比较直观，它先找到key对应的节点链的链头（数组中的某个项），然后遍历该节点链，如果在节点链中找到key相等的节点，则为该节点之前的所有节点重新创建节点并组成一条新链，将该新链的链尾指向找到节点的下一个节点。这样如前面rehash提到的，同时有两条链存在，即使有另一个线程正在该链上遍历也不会出问题。然而Doug Lea又挖掘到了新的优化点，为了减少新链的创建同时利用CPU缓存的特性，在1.7中，他不再重新创建一条新的链，而是只在当起缓存中将链中找到的节点移除，而另一个遍历线程的缓存中继续存在原来的链。当移除的是链头是更新数组项的值，否则更新找到节点的前一个节点的next指针。这也是HashEntry中next指针没有设置成final的原因。当然remove操作如果第一次尝试获得锁失败也会如put操作一样先进入自旋状态，这里的scanAndLock和scanAndLockForPut类似，只是它不做预创建节点的步骤，不再细说：

```
    final V remove(Object key, int hash, Object value) {
      if (!tryLock())
        scanAndLock(key, hash);
```

```
        V oldValue = null;
        try {
            HashEntry<K,V>[] tab = table;
            int index = (tab.length - 1) & hash;
            HashEntry<K,V> e = entryAt(tab, index);
            HashEntry<K,V> pred = null;
            while (e != null) {
                K k;
                HashEntry<K,V> next = e.next;
                if ((k = e.key) == key ||  (e.hash == hash && key.equals(k))) {
                    V v = e.value;
                    if (value == null || value == v || value.equals(v)) {
                        if (pred == null)
                            setEntryAt(tab, index, next);
                        else
                            pred.setNext(next);
                        ++modCount;
                        --count;
                        oldValue = v;
                    }
                    break;
                }
                pred = e;
                e = next;
            }
        } finally {
            unlock();
        }
        return oldValue;
    }
```

Segment中的其他操作

ConcurrentHashMap添加了replace接口，它和put的区别是put操作如果原Map中不存在key会将传入的键值对添加到Map中，而replace不会这么做，它只是简单的返回false。Segment中的replace操作先加锁或自旋等待，然后遍历相应的节点链，如果找到节点，则替换原有的值，返回true，否则返回false，比较简单，不细究。

Segment中的clear操作不同于其他操作，它直接请求加锁而没有自旋等待的步骤，这可能是因为它需要对整个table做操作，因而需要等到所有在table上的操作的线程退出才能执行，而不象其他操作只是对table中的一条链操作，对一条链操作的线程执行的比较快，因而自旋可以后获得锁的可能性比较大，对table操作的等待相对要比较久，因而自旋等待意义不大。clear操作只是将数组的每个项设置为null，它使用setEntryAt的延迟设置，从而保证其他读线程的正常工作。

Segment类的实现是ConcurrentHashMap实现的核心，因而理解了它的实现，要看ConcurrentHashMap的其他代码就感觉很简单和直观了。

ConcurrentHashMap中的**hash**算法

这个貌似没什么好说的，我也不知道为什么它要这么做，它的实现和HashMap类似，从1.6到1.7的变化也不大。类似HashMap实现，在每次操作都先用该方法计算出hash值，然后根据该值计算出Segment数组中的索

引（在Segment中计算出HashEntry的索引），计算Segment数组的索引和计算Segment中HashEntry的索引不太一样，在计算Segment数组索引时取的时hash值高位的值和(segments.length - 1)的值做'&'操作，而Segmnt中计算HashEntry的索引则使用低位值。

```
private int hash(Object k) {
    int h = hashSeed;
    if ((0 != h) && (k instanceof String)) {
        return sun.misc.Hashing.stringHash32((String) k);
    }
    h ^= k.hashCode();
    h += (h <<  15) ^ 0xffffcd7d;
    h ^= (h >>> 10);
    h += (h <<   3);
    h ^= (h >>>  6);
    h += (h <<   2) + (h << 14);
    return h ^ (h >>> 16);
}
```

初始化**segments**数组

在ConcurrentHashMap构造函数中，它根据传入的concurrencyLevel决定segments数组的长度，默认值为16，而将传入的initialCapacity（保证2的倍数）除以segments数组的长度（最小值2）作为第一个segments数组中第一个Segment的HashEntry数组的长度，而在每次找到一个要插入的segments数组项的值为null时，参考第一个Segment实例的参数创建一个新的Segment实例赋值该对应的segments数组项：

```
private Segment<K,V> ensureSegment(int k) {
    final Segment<K,V>[] ss = this.segments;
    long u = (k << SSHIFT) + SBASE; // raw offset
    Segment<K,V> seg;
    if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == null) {
        Segment<K,V> proto = ss[0]; // use segment 0 as prototype
        int cap = proto.table.length;
        float lf = proto.loadFactor;
        int threshold = (int)(cap * lf);
        HashEntry<K,V>[] tab = (HashEntry<K,V>[])new HashEntry[ca
p];
        if ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(ss, u)) == n
ull) { // recheck
            Segment<K,V> s = new Segment<K,V>(lf, threshold, tab);
            while ((seg = (Segment<K,V>)UNSAFE.getObjectVolatile(s
s, u)) == null) {
                if (UNSAFE.compareAndSwapObject(ss, u, null, seg = s))
                    break;
            }
        }
    }
    return seg;
}
```

ConcurrentHashMap中的**get**、**containsKey**、**put**、**putIfAbsent**、**replace**、**Remove**、**clear**操作

由于前面提到Segment中对HashEntry数组以及数组项中的节点链遍历操作是线程安全的，因而get、containsKey操作只需要找到相应的Segment实例，通过Segment实例找到节点链，然后遍历节点链即可，不细说。
对put、putIfAbsent、replace、remove、clear操作，它们在Segment中都实现，只需要通过hash值找到Segment实例，然后调用相应方法即可。

ConcurrentHashMap中的size、containsValue、contains、isEmpty操作

因为这些操作需要全局扫瞄整个Map，正常情况下需要先获得所有Segment实例的锁，然后做相应的查找、计算得到结果，再解锁，返回值。然而为了竟可能的减少锁对性能的影响，Doug Lea在这里并没有直接加锁，而是先尝试的遍历查找、计算2遍，如果两遍遍历过程中整个Map没有发生修改（即两次所有Segment实例中modCount值的和一致），则可以认为整个查找、计算过程中Map没有发生改变，我们计算的结果是正确的，否则，在顺序的在所有Segment实例加锁，计算，解锁，然后返回。
以containsValue为例：

```
public boolean containsValue(Object value) {
    // Same idea as size()
    if (value == null)
        throw new NullPointerException();
    final Segment<K,V>[] segments = this.segments;
    boolean found = false;
    long last = 0;
    int retries = -1;
    try {
        outer: for (;;) {
            if (retries++ == RETRIES_BEFORE_LOCK) {
                for (int j = 0; j < segments.length; ++j)
                    ensureSegment(j).lock(); // force creation
            }
            long hashSum = 0L;
            int sum = 0;
            for (int j = 0; j < segments.length; ++j) {
                HashEntry<K,V>[] tab;
                Segment<K,V> seg = segmentAt(segments, j);
                if (seg != null && (tab = seg.table) != null) {
                    for (int i = 0 ; i < tab.length; i++) {
                        HashEntry<K,V> e;
                        for (e = entryAt(tab, i); e != null; e = e.next) {
                            V v = e.value;
                            if (v != null && value.equals(v)) {
                                found = true;
                                break outer;
                            }
                        }
                    }
                    sum += seg.modCount;
                }
            }
            if (retries > 0 && sum == last)
                break;
            last = sum;
```

```
        }
      } finally {
        if (retries > RETRIES_BEFORE_LOCK) {
          for (int j = 0; j < segments.length; ++j)
            segmentAt(segments, j).unlock();
        }
      }
      return found;
    }
```

其他的关于Collection和Iterator的实现和HashMap的实现类似，不再详述。需要注意的一点是由于ConcurrentHashMap的线程安全性，因而它没有如HashMap一样实现fail-fast原则，即在遍历时，依然可以对其做修改（put、remove），而HashMap不可以，否则会抛出ConcurrentModificationException。另一点区别时ConcurrentHashMap同样不支持key或value为null的情况。

注1(http://www.ibm.com/developerworks/cn/java/j-jtp08223/):
JMM 掌管着一个线程对内存的动作 （读和写）影响其他线程对内存的动作的方式。由于使用处理器寄存器和预处理 cache 来提高内存访问速度带来的性能提升，Java 语言规范（JLS）允许一些内存操作并不对于所有其他线程立即可见。有两种语言机制可用于保证跨线程内存操作的一致性——synchronized和volatile。

按照 JLS 的说法，“在没有显式同步的情况下，一个实现可以自由地更新主存，更新时所采取的顺序可能是出人意料的。”其意思是说，如果没有同步的话，在一个给定线程中某种顺序的写操作对于另外一个不同的线程来说可能呈现出不同的顺序， 并且对内存变量的更新从一个线程传播到另外一个线程的时间是不可预测的。

虽然使用同步最常见的原因是保证对代码关键部分的原子访问，但实际上同步提供三个独立的功能——原子性、可见性和顺序性。原子性非常简单——同步实施一个可重入的（reentrant）互斥，防止多于一个的线程同时执行由一个给定的监视器保护的代码块。不幸的是，多数文章都只关注原子性方面，而忽略了其他方面。但是同步在 JMM 中也扮演着很重要的角色，会引起 JVM 在获得和释放监视器的时候执行内存壁垒（memory barrier）。

一个线程在获得一个监视器之后，它执行一个读屏障（read barrier）——使得缓存在线程局部内存（比如说处理器缓存或者处理器寄存器）中的所有变量都失效，这样就会导致处理器重新从主存中读取同步代码块使用的变量。与此类似，在释放监视器时，线程会执行一个写屏障（write barrier）——将所有修改过的变量写回主存。互斥独占和内存壁垒结合使用意味着只要您在程序设计的时候遵循正确的同步法则（也就是说，每当写一个后面可能被其他线程访问的变量，或者读取一个可能最后被另一个线程修改的变量时，都要使用同步），每个线程都会得到它所使用的共享变量的正确的值。

如果在访问共享变量的时候没有同步的话，就会发生一些奇怪的事情。一些变化可能会通过线程立即反映出来，而其他的则需要一些时间（这由关联缓存的本质所致）。结果，如果没有同步您就不能保证内存内容必定一致（相关的变量相互间可能会不一致），或者不能得到当前的内存内容

（一些值可能是过时的）。避免这种危险情况的常用方法（也是推荐使用的方法）当然是正确地使用同步。然而在有些情况下，比如说在像 ConcurrentHashMap 之类的一些使用非常广泛的库类中，在开发过程当中还需要一些额外的专业技能和努力（可能比一般的开发要多出很多倍）来获得较高的性能。

posted on 2013-10-18 22:24 DLevin 阅读(8142) 评论(0) 编辑 收藏 所属分类: Core Java

新用户注册 刷新评论列表

只有注册用户登录后才能发表评论。

网站导航:
博客园 IT新闻 知识库 C++博客 博问 管理

相关文章:
分布式Map中实现引用计数
ReferenceCountSet无锁实现
使用Exchanger实现两个线程之间的数据交互
Java Core系列之TreeMap实现详解
Java Core系列之HashMap实现
SAX解析XML文件
【转】关于AccessController.doPrivileged
Spring中集合定义
[多问几个为什么]为什么匿名内部类中引用的局部变量和参数需要final而成员字段不用?