Hashmap

## 重要参数

HashMap 的实例有两个参数影响其性能：初始容量 和加载因子。容量 是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部[数据结构](http://baike.baidu.com/view/9900.htm" \t "_blank)），从而哈希表将具有大约两倍的桶数。

　　通常，默认加载因子 (.75) 在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

如果很多映射关系要存储在 HashMap 实例中，则相对于按需执行自动的 rehash 操作以增大表的容量来说，使用足够大的初始容量创建它将使得映射关系能更有效地存储。

**HashMap的数据结构**

 HashMap主要是用数组来存储数据的，我们都知道它会对key进行哈希运算，哈系运算会有重复的哈希值，对于哈希值的冲突，HashMap采用链表来解决的。在HashMap里有这样的一句属性声明：  
transient Entry[] table;  
Entry就是HashMap存储数据所用的类，它拥有的属性如下  
final K key;  
V value;  
final int hash;  
Entry<K,V> next;

看到next了吗？next就是为了哈希冲突而存在的。比如通过哈希运算，一个新元素应该在数组的第10个位置，但是第10个位置已经有Entry，那么好吧，将新加的元素也放到第10个位置，将第10个位置的原有Entry赋值给当前新加的 Entry的next属性。数组存储的是链表，链表是为了解决哈希冲突的，这一点要注意。

**几个关键的属性**  
存储数据的数组  
transient Entry[] table; 这个上面已经讲到了  
默认容量  
static final int DEFAULT\_INITIAL\_CAPACITY = 16;  
最大容量  
static final int MAXIMUM\_CAPACITY = 1 << 30;  
默认加载因子，加载因子是一个比例，当HashMap的数据大小>=容量\*加载因子时，HashMap会将容量扩容  
static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  
当实际数据大小超过threshold时，HashMap会将容量扩容，threshold＝容量\*加载因子  
int threshold;  
加载因子  
final float loadFactor;

**正确的使用HashMap**

1：不要在并发场景中使用HashMap  
           HashMap是线程不安全的，如果被多个线程共享的操作，将会引发不可预知的问题，据sun的说法，在扩容时，会引起链表的闭环，在get元素时，就会无限循环，后果是cpu100%。  
看看get方法的红色部分  
public V get(Object key) {  
    if (key == null)  
        return getForNullKey();  
        int hash = hash(key.hashCode());  
        for (Entry<K,V> e = table[indexFor(hash, table.length)];  
             e != null;  
             e = e.next) {  
            Object k;  
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))  
                return e.value;  
        }  
        return null;  
    }  
  
2：如果数据大小是固定的，那么最好给HashMap设定一个合理的容量值  
        根据上面的分析，HashMap的初始默认容量是16，默认加载因子是0.75，也就是说，如果采用HashMap的默认构造函数，当增加数据时，数据实际容量超过10\*0.75=12时，HashMap就扩容，扩容带来一系列的运算，新建一个是原来容量2倍的数组，对原有元素全部重新哈希，如果你的数据有几千几万个，而用默认的HashMap构造函数，那结果是非常悲剧的，因为HashMap不断扩容，不断哈希，在使用HashMap的场景里，不会是多个线程共享一个HashMap,除非对HashMap包装并同步，由此产生的内存开销和cpu开销在某些情况下可能是致命的。

修改了一下题目=  =   
    发这个帖子是想深入了解在多线程环境下操作HashMap引发的问题，探讨在某些特定的多线程环境下是不是能直接使用HashMap?在哪些特定的并发环境下能否正常使用呢?   
  
    众所周知HashMap不是线程安全的，但到底HashMap在什么情况才不是线程安全的? 查看HashMap的源码，内部有一个modCount变量，在put、remove、等等进行结构性修改时改变这个值。在Hash Iterator中记录expectedModCount变量，在遍历或者删除时比较modCount与expectedModCount的值是否相等，不相等就抛出ConcurrentModificationException，类似一种乐观所的机制，代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **final** Entry<K,V> nextEntry() {
2. [color=red]**if** (modCount != expectedModCount)
3. **throw** **new** ConcurrentModificationException();[/color]
4. Entry<K,V> e = next;
5. **if** (e == **null**)
6. **throw** **new** NoSuchElementException();
8. **if** ((next = e.next) == **null**) {
9. Entry[] t = table;
10. **while** (index < t.length && (next = t[index++]) == **null**)
11. ;
12. }
13. current = e;
14. **return** e;
15. }
17. **public** **void** remove() {
18. **if** (current == **null**)
19. **throw** **new** IllegalStateException();
20. [color=red]**if** (modCount != expectedModCount)
21. **throw** **new** ConcurrentModificationException();[/color]
22. Object k = current.key;
23. current = **null**;
24. HashMap.**this**.removeEntryForKey(k);
25. expectedModCount = modCount;
26. }

那就是说，在多线程环境下是操作HashMap的内部类迭代器的时(Iterator的remove与遍历)，就有可能出现并发的问题。那假设我现在有一个HashMap，保证不对HashMap的迭代器操作，在多线程环境下单纯地调用HashMap的get、remove、put、containsKey等方法，是不是就不会出现并发的问题呢？ 

**Java代码  [收藏代码](javascript:void())**

1. **static** Random random = **new** Random();
2. **static** HashMap<Integer, String> map = **new** HashMap<Integer, String>();
4. /\*\*
5. \* @param args the command line arguments
6. \*/
7. **public** **static** **void** main(String[] args) {

10. Thread t1 = **new** Thread() {
11. @Override
12. **public** **void** run() {
13. **for** (**int** i = 0; i < 1000; i++) {
14. **try** {
15. TimeUnit.MILLISECONDS.sleep(1);
16. } **catch** (InterruptedException ex) {
17. System.err.println("t1 catch the InterruptedException...");
18. }
19. **int** key = random.nextInt(1000);
20. map.put(key, String.valueOf(key));
21. }
22. }
23. };
24. t1.start();
26. Thread t2 = **new** Thread() {
27. @Override
28. **public** **void** run() {
29. **for** (**int** i = 0; i < 1000; i++) {
30. **try** {
31. TimeUnit.MILLISECONDS.sleep(1);
32. } **catch** (InterruptedException ex) {
33. System.err.println("t2 catch the InterruptedException...");
34. }
35. **int** key = random.nextInt(1000);
36. map.get(key);
37. }
38. }
39. };
40. t2.start();
42. Thread t3 = **new** Thread() {
43. @Override
44. **public** **void** run() {
45. **for** (**int** i = 0; i < 1000; i++) {
46. **try** {
47. TimeUnit.MILLISECONDS.sleep(1);
48. } **catch** (InterruptedException ex) {
49. System.err.println("t3 catch the InterruptedException...");
50. }
51. Integer key = random.nextInt(1000);
52. map.containsKey(key);
53. }
54. }
55. };
56. t3.start();
58. Thread t4 = **new** Thread() {
59. @Override
60. **public** **void** run() {
61. **for** (**int** i = 0; i < 1000; i++) {
62. **try** {
63. TimeUnit.MILLISECONDS.sleep(1);
64. } **catch** (InterruptedException ex) {
65. System.err.println("t4 catch the InterruptedException...");
66. }
67. **int** key = random.nextInt(1000);
68. map.remove(key);
69. }
70. }
71. };
72. t4.start();
73. }

经过测试之后，代码能正确运行，暂且不考虑数据的是否100%正确，最少没有抛出ConcurrentModificationException。原因是，上面代码中的4种操作都没有直接操作到HashMap内部的迭代器。   
当然如果加上操作迭代器的代码，异常马上就出来了

**Java代码  [收藏代码](javascript:void())**

1. Thread t5 = **new** Thread() {
2. @Override
3. **public** **void** run() {
4. **for** (**int** i = 0; i < 1000; i++) {
5. **try** {
6. TimeUnit.MILLISECONDS.sleep(1);
7. } **catch** (InterruptedException ex) {
8. System.err.println("t5 catch the InterruptedException...");
9. }
10. **for** (Integer key : map.keySet()) {
11. }
12. }
13. }
14. };
15. t5.start();

但在多线程环境下，HashMap的get、put、remove、containsKey这样的操作是否就正确并且保证一定没有问题！？(甚至保证对同一个key的get、put、remove、containsKey的操作是单线程的前提下，多线程是对于不同key)(否则就算多线程对相同的key操作最多也是会出现数据的不一致性，如对相同的key进行多次的put操作，后一个覆盖了前一个....)   
  
带着查看一下JavaDoc关于HashMap同步的描述：   
**注意，此实现不是同步的。如果多个线程同时访问一个哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须 保持外部同步。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 Collections.synchronizedMap 方法来“包装”该映射。最好在创建时完成这一操作，以防止对映射进行意外的非同步访问，如下所示：**   
  
JavaDoc的说法也是有点模糊，如果对HashMap多线程访问，从结构上修改了该映射就必须保持同步?但测试代码中的put、remove操作不也是结构性修改的动作吗？到了这里问题变得有点难以解释和理清思路了。。。   
  
**跳出以上描述重新考虑这个问题，其中一种想法是：上面是建立在关心HashMap内部实现的基础上去讨论和测试的，但API文档已经说明HashMap并非线程安全的，不要在多线程环境下对HashMap进行结构性的动作，去冒将来不确定的时间发生任意不确定行为的风险。比如在某个版本会为了某种原因而去修改HashMap的内部实现，单纯调用HashMap的put、remove操作也会抛出ConcurrentModificationException(当然这个可能性比较低....)，那么以前写的旧代码就有可能出问题了。。。。**   
  
说到这里，希望各位高手、有经验的热心朋友可以指点一下，或一起探讨一下这个问题。。。谢谢http://www.iteye.com/images/smiles/icon_smile.gif   
  
  
考虑了一下，多线程环境下put还是可能会有问题

**Java代码  [收藏代码](javascript:void())**

1. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
2. Entry<K,V> e = table[bucketIndex];
3. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
4. **if** (size++ >= threshold)
5. resize(2 \* table.length);
6. }

发生在index相同的情况下，大家拿到的链头可能不是最新的，后一个会直接覆盖了前一个   
而且当多条线程检测到容量超过负载因子时，会能发生多次resize。   
结构性修改总是采用持有，替换的方式去操作，因此操作内部数组与单向链表的时候不会抛出NullPointerException   
  
//======================add at 5.1=================================   
小结一下：remove与put都是一样的，由于大家拿到的不是最新链头，只要大家在Entry数组的index相同时(经过hash后的index)，就有可能出现后一个覆盖前一个的操作，即前一个的操作无效。   
可能产生的现象会是：   
1)put进行的data有可能丢失了   
2)一些通过remove(Object key)删除掉的元素(返回删除成功)又出来了。   
3)多线程检测到HashMap容量超过负载因子时会进行多次的resize，由于要rehash，所以消耗的性能也是巨大的。   
  
**但如果put、与remove，是在同一条线程中处理，而get发生在多线程环境，是不会出现上述3种情况或抛出ConcurrentModificationException的。**唯一可能出现的是返回不是最新的数据(可能拿到刚好被remove的，拿不到刚好put进去的)，但这种情况即使ConcurrentHashMap也会出现。但ConcurrentHashMap的get方法多了一步处理，看起来好像更精确. 

**Java代码  [收藏代码](javascript:void())**

1. V get(Object key, **int** hash) {
2. **if** (count != 0) { // read-volatile
3. HashEntry<K,V> e = getFirst(hash);
4. **while** (e != **null**) {
5. **if** (e.hash == hash && key.equals(e.key)) {
6. V v = e.value;
7. **if** (v != **null**)
8. **return** v;
9. **return** readValueUnderLock(e); // recheck
10. }
11. e = e.next;
12. }
13. }
14. **return** **null**;
15. }
17. V readValueUnderLock(HashEntry<K,V> e) {
18. lock();
19. **try** {
20. **return** e.value;
21. } **finally** {
22. unlock();
23. }
24. }

当调用ConcurrentHashMap的get方法时，找到key所对应的Entry而getValue为null时会调用readValueUnderLock(e)方法上分离锁取一次。**这具体会发生什么情况下会发生呢且readValueUnderLock(e)能生效呢**？网上有两种比较常见的说法：   
1)另外一条线程在修改节点   
2)HashEntry构造函数中对value的赋值以及对tab[index]的赋值可能被重新排序，或许源出自<http://www.iteye.com/topic/344876>   
查看ConcurrentHashMap源码能看到put方法其实是创建一个新的节点并以之前的第一个节点为这个节点的next节点(即这个新创建的节点为单向链表的第一个节点)，然后

**Java代码  [收藏代码](javascript:void())**

1. tab[index] = **new** HashEntry<K,V>(key, hash, first, value);

执行赋值   
remove方法则是重新把整条单向链表(除被删除Entry)全部拷贝一次执行赋值

**Java代码  [收藏代码](javascript:void())**

1. **for** (HashEntry<K,V> p = first; p != e; p = p.next)
2. newFirst = **new** HashEntry<K,V>(p.key, p.hash,
3. newFirst, p.value);
4. tab[index] = newFirst;

所有，既不是另外一条线程正在修改某节点，也不是重新排序。最后追踪到源码的注释：

**引用**

This is possible only if a   
compiler happens to reorder a HashEntry initialization with   
its table assignment, which is legal under memory model   
but is not known to ever occur.

原来还是Java内存模型的问题(Jmm)，执行

**Java代码  [收藏代码](javascript:void())**

1. tab[index] = newFirst

或

**Java代码  [收藏代码](javascript:void())**

1. tab[index] = **new** HashEntry<K,V>(key, hash, first, value);

时，Entry已经被认为初始化ok并返回给table(在堆中分派一块空间)，这是正好有其他线程调用到get方法，获取到这个刚“实例化好”的单向链表，但由于JMM初始化的无序性，当你get这个链表Entry中的value 时，他有可能还没完全初始化好！或按注释所说不知道发生过这样的初始化但这是合法的。。。。这和为什么Java的单例模式不能使用双重检查加锁的原因类似，都是由于Java内存模型(JMM)的无序性。这里有部分介绍<http://rjx2008.iteye.com/admin/blogs/342474>   
  
数据一致性的问题扯远了，回到上述：但如果put、与remove，是在同一条线程中处理，而get发生在多线程环境，是不会出现上述3种情况或抛出ConcurrentModificationException的。   
最少有两种的适用场合   
1)类似委托的机制   
2)类似命令模式的执行方式   
这两种处理get,put时方式是很接近的，就是多线程接收到要删除、添加数据的指令   
但最终会由一条线程去过滤这些指令，判断、分析、处理，最后单线程地对HashMap执行put、remove的操作。只要类似这种的处理方式，都应该可以使用。   
这只是个人观点，请大家多提点意见、补充或纠正:-)