[**JAVA**](javascript:;)**HashMap**[**底层**](javascript:;)**实现原理**

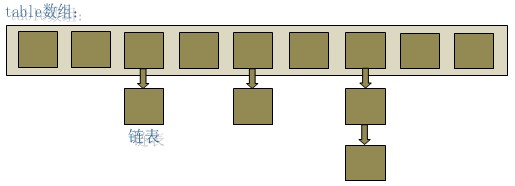
互联网 2016年01月14日 10:54:00

      1.    HashMap概述：

      HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

      2.    HashMap的[数据结构](javascript:;)：

      在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“[链表](javascript:;)散列”的数据结构，即数组和链表的结合体。



      从上图中可以看出，HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会[初始化](javascript:;)一个数组。

      源码如下：

      Java代码

1. /\*\*
2. \* The table, resized as necessary. [Length](javascript:;) MUST Always be a power of two.
3. \*/
4. transient Entry[] table;
6. static class Entry<K,V> implements Map.Entry<K,V> {
7. final K key;
8. V value;
9. Entry<K,V> next;
10. final int hash;
11. ……
12. }

      可以看出，Entry就是数组中的元素，每个 Map.Entry 其实就是一个key-value对，它持有一个指向下一个元素的引用，这就构成了链表。

      3.    HashMap的存取实现：

      1) 存储：

      Java代码

1. public V put(K key, V value) {
2. // HashMap允许存放null键和null值。
3. // 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。
4. if (key == null)
5. return putForNullKey(value);
6. // 根据key的keyCode[重新计算](javascript:;)hash值。
7. int hash = hash(key.hashCode());
8. // 搜索指定hash值在对应table中的索引。
9. int i = indexFor(hash, table.length);
10. // 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。
11. for (Entry<K,V> e = table[i]; e != null; e = e.next) {
12. Object k;
13. if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
14. V oldValue = e.value;
15. e.value = value;
16. e.recordAccess(this);
17. return oldValue;
18. }
19. }
20. // 如果i索引处的Entry为null，表明此处还没有Entry。
21. modCount++;
22. // 将key、value添加到i索引处。
23. addEntry(hash, key, value, i);
24. return null;
25. }

      从上面的源代码中可以看出：当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

      addEntry(hash, key, value, i)方法根据计算出的hash值，将key-value对放在数组table的i索引处。addEntry 是HashMap 提供的一个包访问权限的方法，代码如下：

      Java代码

1. [void](javascript:;) addEntry(int hash, K key, V value, int bucketIndex) {
2. // 获取指定 bucketIndex 索引处的 Entry
3. Entry<K,V> e = table[bucketIndex];
4. // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
5. table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
6. // 如果 Map 中的 key-value 对的数量超过了极限
7. if (size++ >= threshold)
8. // 把 table 对象的长度扩充到原来的2倍。
9. resize(2 \* table.length);
10. }

      当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

      hash(int h)方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

      Java代码

1. static int hash(int h) {
2. h ^= (h >>> 20) ^ (h >>> 12);
3. return h ^ (h >>> 7) ^ (h >>> 4);
4. }

      我们可以看到在HashMap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过HashMap的[数据结构](javascript:;)是数组和链表的结合，所以我们当然希望这个HashMap里面的元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表，这样就大大优化了查询的效率。

      对于任意给定的对象，只要它的 hashCode() [返回值](javascript:;)相同，那么程序调用 hash(int h) 方法所计算得到的 hash 码值总是相同的。我们首先想到的就是把hash值对数组长度取模[运算](javascript:;)，这样一来，元素的分布相对来说是比较均匀的。但是，“模”运算的消耗还是比较大的，在HashMap中是这样做的：调用 indexFor(int h, int length) 方法来计算该对象应该保存在 table 数组的哪个索引处。indexFor(int h, int length) 方法的代码如下：

      Java代码

1. static int indexFor(int h, int length) {
2. return h & (length-1);
3. }

      这个方法非常巧妙，它通过 h & (table.length -1) 来得到该对象的保存位，而HashMap底层数组的长度总是 2 的n 次方，这是HashMap在速度上的优化。在 HashMap 构造器中有如下代码：

      Java代码

1. int capacity = 1;
2. while (capacity < initialCapacity)
3. capacity <<= 1;

      这段代码保证初始化时HashMap的容量总是2的n次方，即底层数组的长度总是为2的n次方。

      当length总是 2 的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。

      这看上去很简单，其实比较有玄机的，我们举个例子来说明：

      假设数组长度分别为15和16，优化后的hash码分别为8和9，那么&[运算](javascript:;)后的结果如下：

      h & (table.length-1)                     hash                             table.length-1

      8 & (15-1)：                                 0100                   &              1110                   =                0100

      9 & (15-1)：                                 0101                   &              1110                   =                0100

      -----------------------------------------------------------------------------------------------------------------------

      8 & (16-1)：                                 0100                   &              1111                   =                0100

      9 & (16-1)：                                 0101                   &              1111                   =                0101

      从上面的例子中可以看出：当它们和15-1（1110）“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8和9会被放到数组中的同一个位置上形成链表，那么查询的时候就需要遍历这个链 表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为15的时候，hash值会与15-1（1110）进行“与”，那么 最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！而当数组长度为16时，即为2的n次方时，2n-1得到的二进制数的每个位上的值都为1，这使得在低位上&时，得到的和原hash的低位相同，加之hash(int h)方法对key的hashCode的进一步优化，加入了高位计算，就使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。

      所以说，当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

      根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() [返回值](javascript:;)相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

      2) 读取：

      Java代码

1. public V get(Object key) {
2. if (key == null)
3. return getForNullKey();
4. int hash = hash(key.hashCode());
5. for (Entry<K,V> e = table[indexFor(hash, table.length)];
6. e != null;
7. e = e.next) {
8. Object k;
9. if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
10. return e.value;
11. }
12. return null;
13. }

      有了上面存储时的hash算法作为基础，理解起来这段代码就很容易了。从上面的源代码中可以看出：从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

      3) 归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

      4.    HashMap的resize（rehash）：

      当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

      那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后[重新计算](javascript:;)每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

      5.    HashMap的性能参数：

      HashMap 包含如下几个构造器：

      HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

      HashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。

      HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

      HashMap的基础构造器HashMap(int initialCapacity, float loadFactor)带有两个参数，它们是初始容量initialCapacity和加载因子loadFactor。

      initialCapacity：HashMap的最大容量，即为[底层](javascript:;)数组的长度。

      loadFactor：负载因子loadFactor定义为：散列表的实际元素数目(n)/ 散列表的容量(m)。

      负载因子衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用[链表](javascript:;)法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

      HashMap的实现中，通过threshold字段来判断HashMap的最大容量：

      Java代码

1. threshold = (int)(capacity \* loadFactor);

      结合负载因子的定义公式可知，threshold就是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize后的HashMap容量是容量的两倍：

      Java代码

1. if (size++ >= threshold)
2. resize(2 \* table.length);

      6.    Fail-Fast机制：

      我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

      这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器[初始化](javascript:;)过程中会将这个值赋给迭代器的expectedModCount。

      Java代码

1. HashIterator() {
2. expectedModCount = modCount;
3. if (size > 0) { // advance to first entry
4. Entry[] t = table;
5. while (index < t.[length](javascript:;) && (next = t[index++]) == null)
6. ;
7. }
8. }

      在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

      注意到modCount声明为volatile，保证线程之间修改的可见性。

      Java代码

1. final Entry<K,V> nextEntry() {
2. if (modCount != expectedModCount)
3. throw new ConcurrentModificationException();

      在HashMap的API中指出：

      由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：在迭代器创建之后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定的时间发生任意不确定行为的风险。

      注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。