**HashMap实现原理及源码分析**

# 1.1 不同数据结构在新增、查找等基础操作执行性能

数组：采用一段连续的存储单元来存储数据。对于指定下标的查找，时间复杂度为O(1)；通过给定值进行查找，需要遍历数组，逐一比对给定关键字和数组元素，时间复杂度为O(n)，当然，对于有序数组，则可采用二分查找，插值查找，斐波那契查找等方式，可将查找复杂度提高为O(logn)；对于一般的插入删除操作，涉及到数组元素的移动，其平均复杂度也为O(n)。

线性链表：对于链表的新增，删除等操作（在找到指定操作位置后），仅需处理结点间的引用即可，时间复杂度为O(1)，而查找操作需要遍历链表逐一进行比对，复杂度为O(n)。

二叉树：对一棵相对平衡的有序二叉树，对其进行插入，查找，删除等操作，平均复杂度均为O(logn)。

哈希表：相比上述几种数据结构，在哈希表中进行添加，删除，查找等操作，性能十分之高，不考虑哈希冲突的情况下，仅需一次定位即可完成，时间复杂度为O(1)，接下来我们就来看看哈希表是如何实现达到惊艳的常数阶O(1)的。

# 1.2 哈希表与哈悉函数、哈悉冲突

　　我们知道，数据结构的物理存储结构只有两种：**顺序存储结构和链式存储结构**（像栈，队列，树，图等是从逻辑结构去抽象的，映射到内存中，也这两种物理组织形式），而在上面我们提到过，在数组中根据下标查找某个元素，一次定位就可以达到，哈希表利用了这种特性，哈希表的主干就是数组。

比如我们要新增或查找某个元素，我们通过把当前元素的关键字 通过某个函数映射到数组中的某个位置，通过数组下标一次定位就可完成操作。

　　哈希函数： 存储位置 = f(关键字)

其中，这个函数f一般称为哈希函数，这个函数的设计好坏会直接影响到哈希表的优劣。

查找操作同理，先通过哈希函数计算出实际存储地址，然后从数组中对应地址取出即可。

哈希冲突

然而万事无完美，如果两个不同的元素，通过哈希函数得出的实际存储地址相同怎么办？也就是说，当我们对某个元素进行哈希运算，得到一个存储地址，然后要进行插入的时候，发现已经被其他元素占用了，其实这就是所谓的哈希冲突，也叫哈希碰撞。前面我们提到过，哈希函数的设计至关重要，好的哈希函数会尽可能地保证 计算简单和散列地址分布均匀,但是，我们需要清楚的是，数组是一块连续的固定长度的内存空间，再好的哈希函数也不能保证得到的存储地址绝对不发生冲突。

那么哈希冲突如何解决呢？哈希冲突的解决方案有多种:开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap即是采用了链地址法，也就是数组+链表的方式。

# 1.3 HashMap

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。

|  |
| --- |
| // HashMap的主干数组，可以看到就是一个Entry数组，初始值为空数组{}，主干数组的长度一定是2的次幂，至于为什么这么做，后面会有详细分析。  // transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY\_TABLE; |

Entry是HashMap中的一个静态内部类。代码如下:

|  |
| --- |
| static class Entry<K,V> implements Map.Entry<K,V> {  final K key;  V value;  Entry<K,V> next;//存储指向下一个Entry的引用，**单链表结构**  int hash;//对key的hashcode值进行hash运算后得到的值，存储在Entry，避免重复计算  /\*\*  \* Creates new entry.  \*/  Entry(int h, K k, V v, Entry<K,V> n) {  value = v;  next = n;  key = k;  hash = h;  } |

简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

|  |
| --- |
| //实际存储的key-value键值对的个数  transient int size;  //阈值，当table == {}时，该值为初始容量（初始容量默认为16）；当table被填充了，也就是为table分配内存空间后，threshold一般为 capacity\*loadFactory。HashMap在进行扩容时需要参考threshold，后面会详细谈到  int threshold;  //负载因子，代表了table的填充度有多少，默认是0.75  final float loadFactor;  //用于快速失败，由于HashMap非线程安全，在对HashMap进行迭代时，如果期间其他线程的参与导致HashMap的结构发生变化了（比如put，remove等操作），需要抛出异常ConcurrentModificationException  transient int modCount; |

HashMap有4个构造器，其他构造器如果用户没有传入initialCapacity 和loadFactor这两个参数，会使用默认值。

initialCapacity默认为16，loadFactory默认为0.75。

在常规构造器中，没有为数组table分配内存空间（有一个入参为指定Map的构造器例外），而是在执行put操作的时候才真正构建table数组。

|  |
| --- |
| public V put(K key, V value) {  //如果table数组为空数组{}，进行数组填充（为table分配实际内存空间），入参为threshold，此时threshold为initialCapacity 默认是1<<4(24=16)。  //capacity一定是2的次幂  if (table == EMPTY\_TABLE) {  inflateTable(threshold);  }  //如果key为null，存储位置为table[0]或table[0]的冲突链上  if (key == null)  return putForNullKey(value);  int hash = hash(key);//对key的hashcode进一步计算，确保散列均匀  int i = indexFor(hash, table.length);//获取在table中的实际位置  for (Entry<K,V> e = table[i]; e != null; e = e.next) {  //如果该对应数据已存在，执行覆盖操作。用新value替换旧value，并返回旧value  Object k;  if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;//保证并发访问时，若HashMap内部结构发生变化，快速响应失败  addEntry(hash, key, value, i);//新增一个entry  return null;  } |

通过以上代码能够得知，当发生哈希冲突并且size大于阈值的时候，需要进行数组扩容，扩容时，需要新建一个长度为之前数组2倍的新的数组，然后将当前的Entry数组中的元素全部传输过去，扩容后的新数组长度为之前的2倍，所以扩容相对来说是个耗资源的操作。

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。

（key的hashCode相同，即数组的下标相同； key的equals()返回true，这表示同一个key； 如果key的equals()返回false, 则会在链表上新增节点。）

# 1.4 HashMap.resize()

当hashmap中的元素越来越多的时候，碰撞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，所以这是一个通用的操作，很多人对它的性能表示过怀疑，不过想想我们的“均摊”原理，就释然了，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，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。比如说，我们有1000个元素new HashMap(1000), 但是理论上来讲new HashMap(1024)更合适，不过上面annegu已经说过，即使是1000，hashmap也自动会将其设置为1024。 但是new HashMap(1024)还不是更合适的，因为0.75\*1000 < 1000, 也就是说为了让0.75 \* size > 1000, 我们必须这样new HashMap(2048)才最合适，既考虑了&的问题，也避免了resize的问题。

# 1.5 注意

在重写equals的方法的时候，必须注意重写hashCode方法，同时还要保证通过equals判断相等的两个对象，调用hashCode方法要返回同样的整数值。而如果equals判断不相等的两个对象，其hashCode可以相同（只不过会发生哈希冲突，应尽量避免）。

再归纳一下就是hashCode是用于查找使用的，而equals是用于比较两个对象的是否相等的。

# 1.6 小结

HashMap的实现原理：

1）利用key的hashCode重新hash计算出当前对象的元素在数组中的下标。

2）存储时，如果出现hash值相同的key，此时有两种情况。(1)如果key相同，则覆盖原始值；(2)如果key不同（出现冲突），则将当前的key-value放入链表中。

3）获取时，直接找到hash值对应的下标，在进一步判断key是否相同，从而找到对应值。

4）理解了以上过程就不难明白HashMap是如何解决hash冲突的问题，核心就是使用了数组的存储方式，然后将冲突的key的对象放入链表中，一旦发现冲突就在链表中做进一步的对比。