www.cnblogs.com/ITtangtang/p/3948406.html

一、HashMap概述

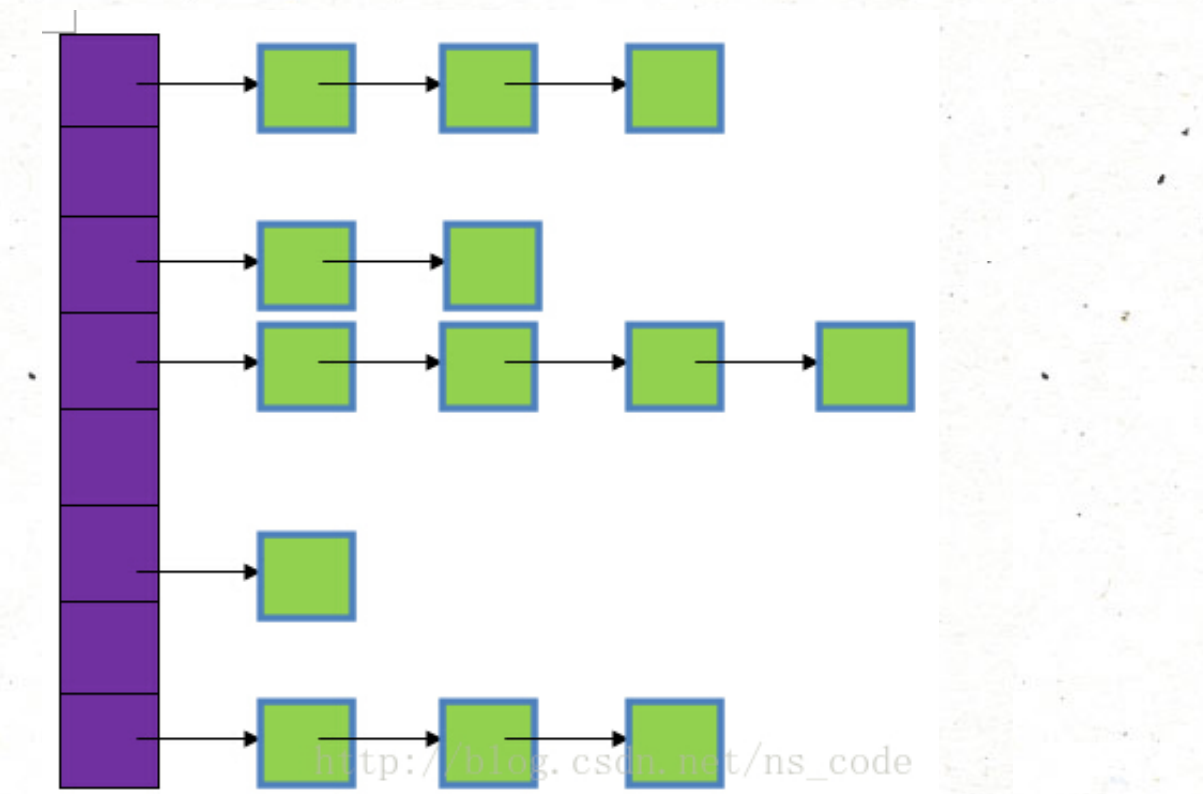
HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过Collections类的静态方法synchronizedMap获得线程安全的HashMap。

```
Map map = Collections.synchronizedMap(new HashMap());
```

二、HashMap的数据结构

HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。



图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

我们看看HashMap中Entry类的代码：

```
    /** Entry是单向链表。
     * 它是 “HashMap链式存储法”对应的链表。
     * 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value),
     * equals(Object o), hashCode()这些函数
    **/
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        // 指向下一个节点
        Entry<K,V> next;
        final int hash;

        // 构造函数。
        // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
        Entry(int h, K k, V v, Entry<K,V> n) {
```

```
        value = v;
        next = n;
        key = k;
        hash = h;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
        return value;
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    // 判断两个Entry是否相等
    // 若两个Entry的“key”和“value”都相等，则返回true。
    // 否则，返回false
    public final boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
        if (k1 == k2 || (k1 != null && k1.equals(k2))) {
            Object v1 = getValue();
            Object v2 = e.getValue();
            if (v1 == v2 || (v1 != null && v1.equals(v2)))
                return true;
        }
        return false;
    }

    // 实现hashCode()
    public final int hashCode() {
        return (key==null   ? 0 : key.hashCode()) ^
```

```
                (value==null ? 0 : value.hashCode());
        }

        public final String toString() {
            return getKey() + "=" + getValue();
        }

        // 当向HashMap中添加元素时，绘调用recordAccess()。
        // 这里不做任何处理
        void recordAccess(HashMap<K,V> m) {
        }

        // 当从HashMap中删除元素时，绘调用recordRemoval()。
        // 这里不做任何处理
        void recordRemoval(HashMap<K,V> m) {
        }
    }
```

HashMap其实就是一个Entry数组，Entry对象中包含了键和值，其中next也是一个Entry对象，它就是用来处理hash冲突的，形成一个链表。

三、HashMap源码分析

1、关键属性

先看看HashMap类中的一些关键属性:

```
1 transient Entry[] table;   //存储元素的实体数组
2 transient int size;   //存放元素的个数
3 int threshold;   //阈值,当实际大小超过阈值时，会进行扩容threshold = 加载因子*容量
4  final float loadFactor;   //加载因子
5 transient int modCount;   //被修改的次数
```

其中loadFactor加载因子是表示Hsah表中元素的填满的程度.
若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.链表长度会越来越长,查找效率降低。
反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了.表中的数据将过于稀疏（很多空间还没用，就开始扩容了）

冲突的机会越大,则查找的成本越高.
因此,必须在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折衷. 这种平衡与折衷本质上是数据结构中有名的"时-空"矛盾的平衡与折衷.

如果机器内存足够，并且想要提高查询速度的话可以将加载因子设置小一点；相反如果机器内存紧张，并且对查询速度没有什么要求的话可以将加载因子设置大一点。不过一般我们都不用去设置它，让它取默认值0.75就好了。

2、构造方法

下面看看HashMap的几个构造方法：

```
 1  public HashMap(int initialCapacity, float loadFactor) {
 2          //确保数字合法
 3          if (initialCapacity < 0)
 4              throw new IllegalArgumentException
                ("Illegal initial capacity: " + initialCapacity);
 6          if (initialCapacity > MAXIMUM_CAPACITY)
 7              initialCapacity = MAXIMUM_CAPACITY;
 8          if (loadFactor <= 0 || Float.isNaN(loadFactor))
 9              throw new IllegalArgumentException
                ("Illegal load factor: " + loadFactor);
11
12          // Find a power of 2 >= initialCapacity
13          int capacity = 1;   //初始容量
           //确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次
幂
14          while (capacity < initialCapacity)
15              capacity <<= 1;
16
17          this.loadFactor = loadFactor;
18          threshold = (int)(capacity * loadFactor);
19          table = new Entry[capacity];
20          init();
21      }
22
23      public HashMap(int initialCapacity) {
24          this(initialCapacity, DEFAULT_LOAD_FACTOR);
25      }
26
27      public HashMap() {
28          this.loadFactor = DEFAULT_LOAD_FACTOR;
29          threshold = (int)(DEFAULT_INITIAL_CAPACITY *
DEFAULT_LOAD_FACTOR);
```

```
30          table = new Entry[DEFAULT_INITIAL_CAPACITY];
31          init();
32      }
```

我们可以看到在构造HashMap的时候如果我们指定了加载因子和初始容量的话就调用第一个构造方法，否则的话就是用默认的。默认初始容量为16，默认加载因子为0.75。我们可以看到上面代码中13-15行，这段代码的作用是确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂，至于为什么要把容量设置为2的n次幂，我们等下再看。

重点分析下HashMap中用的最多的两个方法put和get

3、存储数据

下面看看HashMap存储数据的过程是怎样的，首先看看HashMap的put方法：

```
public V put(K key, V value) {
    // 若"key为null"，则将该键值对添加到table[0]中。
    if (key == null)
        return putForNullKey(value);
    // 若"key不为null"，则计算该key.hashCode()的哈希值，然后将其添加到该哈希值对应的链表中。
    int hash = hash(key.hashCode());
    //搜索指定hash值在对应table中的索引
    int i = indexFor(hash, table.length);
    // 循环遍历Entry数组,若"该key"对应的键值对已经存在，则用新的value取代旧的value。然后退出!
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        //如果key相同则覆盖并返回旧值
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
        }
    }
    //修改次数+1
    modCount++;
    //将key-value添加到table[i]处
    addEntry(hash, key, value, i);
    return null;
}
```

上面程序中用到了一个重要的内部接口：Map.Entry，每个 Map.Entry

其实就是一个 key-value 对。从上面程序中可以看出：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。这也说明了前面的结论：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。
我们慢慢的来分析这个函数，第2和3行的作用就是处理key值为null的情况，我们看看putForNullKey(value)方法：

```
1    private V putForNullKey(V value) {
2        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
3            if (e.key == null) {    //如果有key为null的对象存在，则覆盖掉
4                V oldValue = e.value;
5                e.value = value;
6                e.recordAccess(this);
7                return oldValue;
8            }
9        }
10         modCount++;
11         addEntry(0, null, value, 0); //如果键为null的话，则hash值为0
12         return null;
13     }
```

注意：如果key为null的话，hash值为0，对象存储在数组中索引为0的位置。即table[0]
我们再回去看看put方法中第4行，它是通过key的hashCode值计算hash码，下面是计算hash码的函数：

```
1  //计算hash值的方法 通过键的hashCode来计算
2      static int hash(int h) {
3          // This function ensures that hashCodes that differ only by
4          // constant multiples at each bit position have a bounded
5          // number of collisions (approximately 8 at default load
factor).
6          h ^= (h >>> 20) ^ (h >>> 12);
7          return h ^ (h >>> 7) ^ (h >>> 4);
8      }
```

得到hash码之后就会通过hash码去计算出应该存储在数组中的索引，计算索引的函数如下：

```
1      static int indexFor(int h, int length) { //根据hash值和数组长度算出索
```

```
引值
2          return h & (length-1);  //这里不能随便算取，用hash&(length-1)是有
原因的，这样可以确保算出来的索引是在数组大小范围内，不会超出
3      }
```

这个我们要重点说下，我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除法散列法），Hashtable中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

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。

```
1 void addEntry(int hash, K key, V value, int bucketIndex) {
          //链表的头插法,将原来头位置的Entry设置成新Entry的下一个节点
2         Entry<K,V> e = table[bucketIndex];
3         table[bucketIndex] = new Entry<>(hash, key, value, e);
4         if (size++ >= threshold) //如果大于临界值就扩容
5             resize(2 * table.length); //以2的倍数扩容
6     }
```

参数bucketIndex就是indexFor函数计算出来的索引值，第2行代码是取得数组中索引为bucketIndex的Entry对象，第3行就是用hash、key、value构建一个新的Entry对象放到索引为bucketIndex的位置，并且将该位置原先的对象设置为新对象的next构成链表。

第4行和第5行就是判断put后size是否达到了临界值threshold，如果达到了临界值就要进行扩容，HashMap扩容是扩为原来的两倍。

4、调整大小

resize()方法如下：

重新调整HashMap的大小，newCapacity是调整后的单位

```
1     void resize(int newCapacity) {
2         Entry[] oldTable = table;
3         int oldCapacity = oldTable.length;
4         if (oldCapacity == MAXIMUM_CAPACITY) {
5             threshold = Integer.MAX_VALUE;
6             return;
7         }
8
9         Entry[] newTable = new Entry[newCapacity];
10        transfer(newTable);//用来将原先table的元素全部移到newTable里面
11        table = newTable;  //再将newTable赋值给table
12        threshold = (int)(newCapacity * loadFactor);//重新计算临界值
13    }
```

```
void transfer(Entry[] newTable){
    Entry[] src=table;
    int newCapacity=newTable.length;
    for(int j=0;j<src.length;j++){
        Entry<K, V> e=src[j];
        if(e!=null){
            src[j]=null;
            do{
                Entry<K, V> next = e.next; //保存下一次循环的Entry
                int i = indexFor(e.hash, newCapacity); //计算新table需要插入的位置
                //头插法,原位置上的Entry设置成新节点的next
                e.next = newTable[i];
                //新节点占据原位置
```

```
                newTable[i] = e;
                e = next; //轮替，下一次循环
            }while(e!=null);
        }
    }
}
```

新建了一个HashMap的底层数组，上面代码中第10行为调用transfer方法，将HashMap的全部元素添加到新的HashMap中,并重新计算元素在新的数组中的索引位置

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，然后重新计算每个元素在数组中的位置，扩容是需要进行数组复制的，复制数组是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、数据读取

```
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。

6、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：

```
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：