

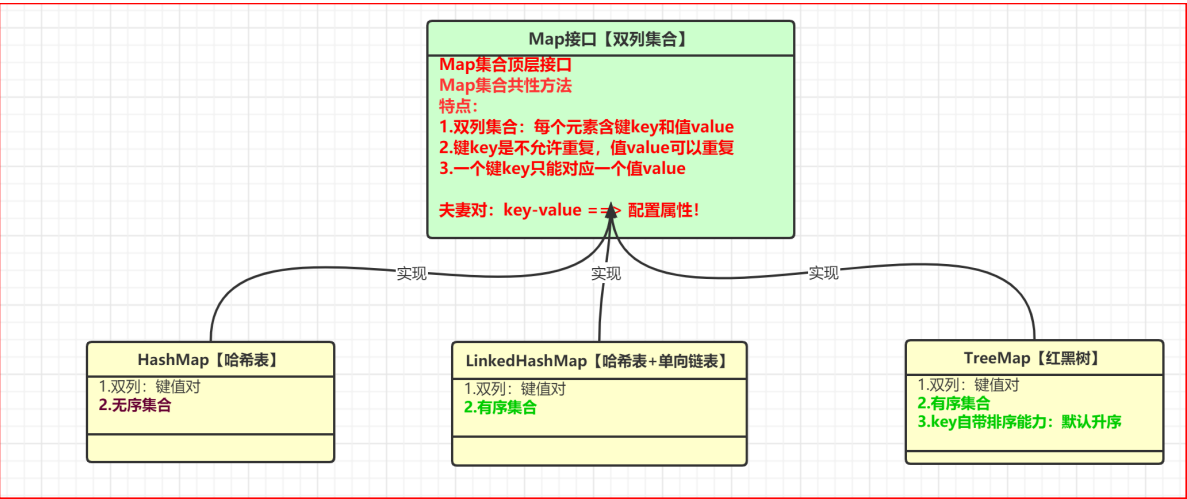
HashMap集合源码详解

学习目标:

- 掌握HashMap底层数据存储结构，及其变换【链表转红黑树】原理
- 掌握HashMap的初始化容量大小，及加载因子0.75原理
- 理解HashMap链表转红黑树边界值设计思想
- 掌握HashMap扩容机制，及初始化容量最佳实践
- 拿下常见大厂HashMap的面试题

一、HashMap集合简介

1.1 什么是HashMap?



HashMap是Map接口的实现类，基于哈希表结构实现的。其主要特点是以key-value存储形式存储数据，即用与存放键值对。HashMap 的操作不是同步的，这意味着它线程不安全。

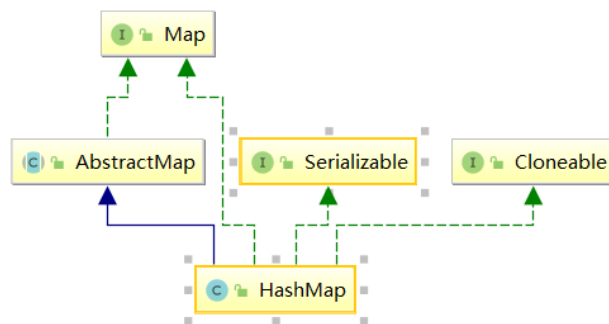
特点:

- 无序性: 存入和取出元素顺序不一致
- 唯一性: 键key是唯一的
- 可存null: 键和值位置都可以是null，但是键位置只能是一个null
- 数据结构: 数据结构控制的是键key而非值value!
 - jdk1.8之前数据结构是: 链表 + 数组
 - jdk1.8之前数据结构是: 链表 + 数组 + 红黑树
 - 单链表阈值(边界值) > 8 并且数组长度大于64，才将链表转换为红黑树。
 - 目的: 高效查询数据

扩展知识: 红黑树 (Red Black Tree) 是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。红黑树是在1972年由Rudolf Bayer发明的，当时被称为平衡二叉B树 (symmetric binary B-trees)

1.2 HashMap类的继承关系

HashMap继承关系如下图所示:



说明：

- Cloneable 空接口，表示可以克隆。创建并返回HashMap对象的一个副本。
- Serializable 序列化接口。属于标记性接口。HashMap对象可以被序列化和反序列化。
- AbstractMap 父类提供了Map实现接口。以最大限度地减少实现此接口所需的工作。

补充：通过上述继承关系我们发现一个很奇怪的现象，就是HashMap已经继承了AbstractMap而AbstractMap类实现了Map接口，那为什么HashMap还要在实现Map接口呢？同样在ArrayList中LinkedList中都是这种结构。

据 java 集合框架的创始人Josh Bloch描述，这样的写法是一个失误。在java集合框架中，类似这样的写法很多，最开始写java集合框架的时候，他认为这样写，在某些地方可能是有价值的，直到他意识到错了。显然的，JDK的维护者，后来不认为这个小小的失误值得去修改，所以就存在下来了。

二、HashMap原理分析

2.1 哈希表简介

什么是哈希表呢？

哈希表（Hash table，也叫散列表），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

哈希表本质上是一个数组，这个数组中存储的是哈希函数算出的值。

目的：为了加快数据查找的速度。

HashMap中哈希表的数组的大小？

我们说，HashMap中的底层数据结构是哈希表，又说是数组+链表+红黑树？那么到底是怎么回事呢？我接下来看HashMap的数据结构

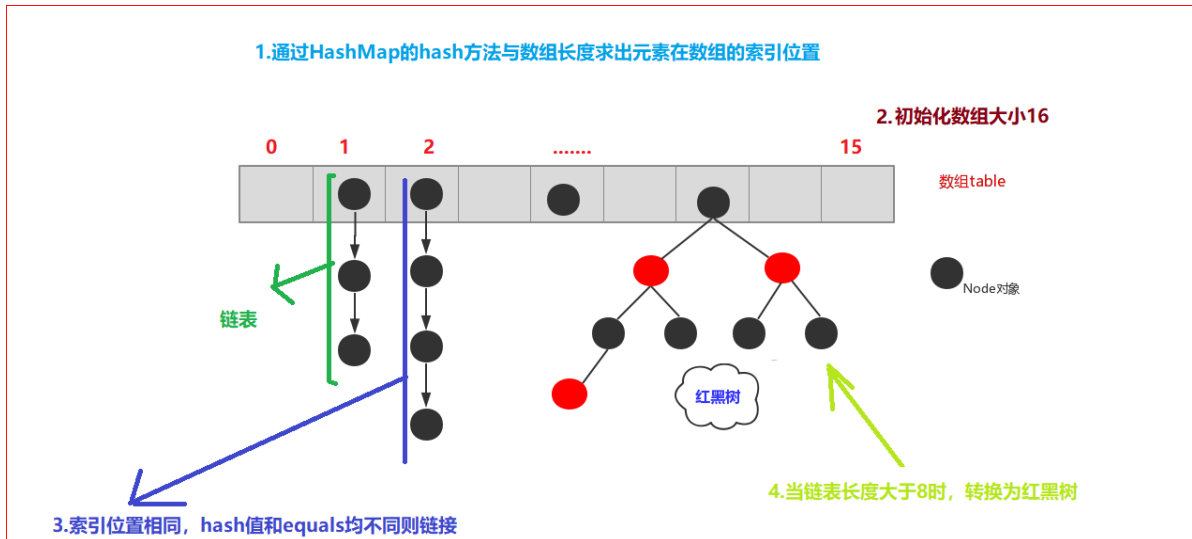
```
1 public class Demo01 {
2     public static void main(String[] args) {
3         HashMap<String, Integer> map = new HashMap<>();
4         map.put("hello", 53);
5         map.put("world", 35);
6         map.put("java", 55);
7     }
8 }
```

```

7      map.put("world", 52);
8      map.put("通话", 51);
9      map.put("重地", 55);
10     }
11 }
12 //1.HashMap<String, Integer> map = new HashMap<>();
13 当创建HashMap集合对象时，
14     JDK8前，构造方法创建一个长度是16的数组Entry[] table 来存储键值对的对象。
15     JDK8后，不是在构造方法中创建对象数组，而是在第一调用put方法时创建长度是16的Node[]
    table数组，存储Node对象。

```

如果节点长度即链表长度大于阈值8，并且数组长度大于64则进行将链表变为红黑树。



2.2 HashMap存储数据过程

1、存储过程中相关属性

1、**加载因子**：默认值是0.75，决定了扩容的条件

```

1 // 加载因子
2 final float loadFactor;

```

2、**扩容的临界值**：计算方式为(容量 乘以 加载因子)

```

1 // 临界值 当实际大小超过临界值时，会进行扩容
2 int threshold;

```

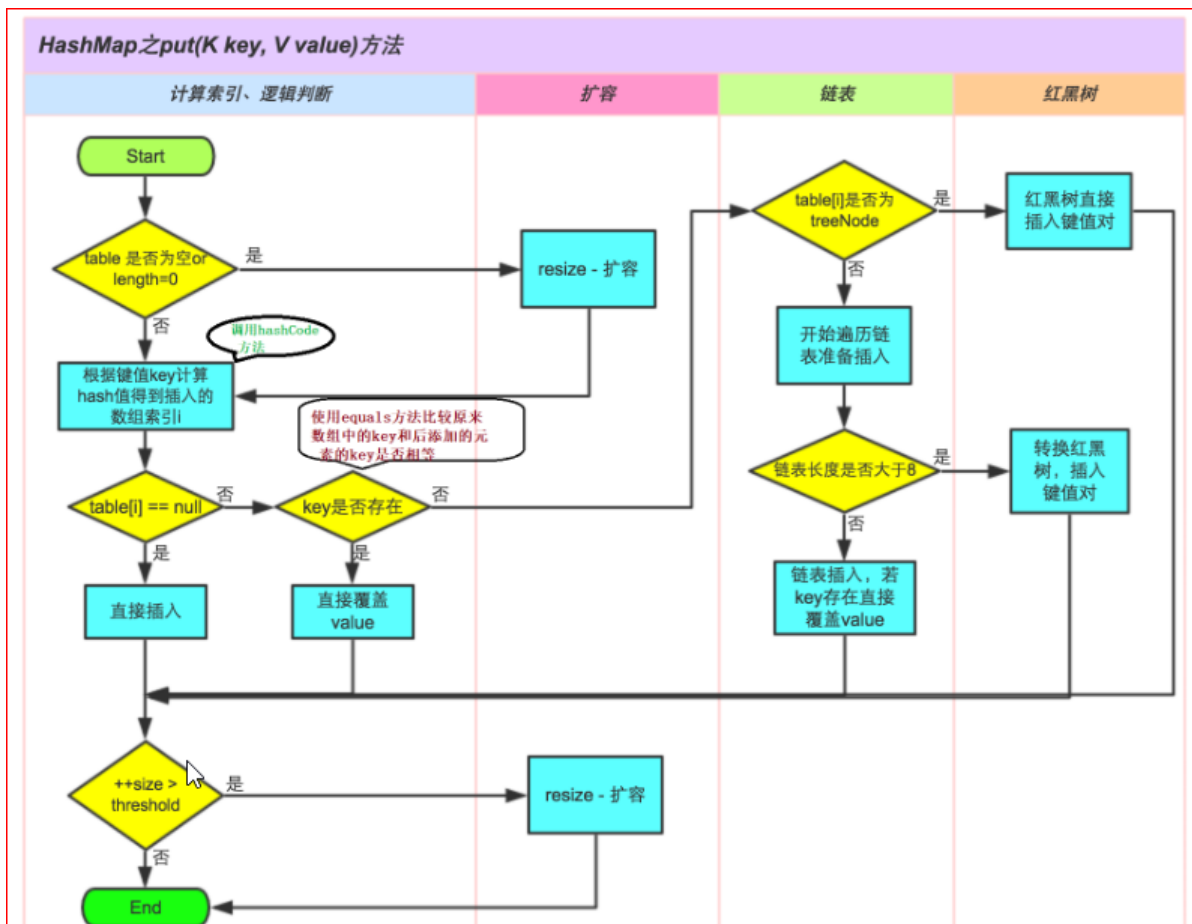
3、**容量capacity**：初始化为16

4、**扩容resize**：达到临界值就扩容。扩容后的 HashMap 容量是之前容量的两倍。

5、**集合元素个数size**：表示HashMap中键值对实时数量，不等于数组长度

2、存储过程图解

上述我们大概阐述了HashMap底层存储数据的方式。为了方便大家更好的理解，我们结合一个存储流程图来进一步说明一下：(jdk8存储过程)



3、存储过程源码分析

```

1  final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
2                boolean evict) {
3      Node<K,V>[] tab; Node<K,V> p; int n, i;
4      //1. 判断是否哈希表为空
5      if ((tab = table) == null || (n = tab.length) == 0)
6          //2. 如果为空初始化容量, 16
7          n = (tab = resize()).length;
8      //3. 如果不为空, 则判断当前key的hash值对应的索引位置是否有元素。
9      if ((p = tab[i = (n - 1) & hash]) == null)
10         //4. 如果没有, 往当前索引位置放入一个新的节点
11         tab[i] = newNode(hash, key, value, null);
12     else {
13         Node<K,V> e; K k;
14         //5. 如果有元素, 判断当前索引位的节点hash值和equals与新key是否相等
15         if (p.hash == hash && ((k = p.key) == key || (key != null &&
16 key.equals(k))))
17             //如果相等, 则覆盖value
18             e = p;
19         //6. 如果不相等, 则判断是否是红黑树
20         else if (p instanceof TreeNode)
21             //如果是红黑树节点, 则将元素存入红黑树节点
22             e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
23         else {
24             //7. 如果不相等, 也不是红黑树节点, 则遍历所有链表节点
25             for (int binCount = 0; ; ++binCount) {
26                 //如果到了最后一个节点还没找到相等的节点
27                 if ((e = p.next) == null) {
28                     //在尾部新增一个节点
29                     p.next = newNode(hash, key, value, null);

```

```

29         //8.判断链表的长度是否大于8
30         if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
31             //如果大于8直接将链表转换为红黑树
32             treeifyBin(tab, hash);
33
34         break;
35     }
36     //如果遍历的节点的hash值和equals值与新key相同，则跳出循环
37     if (e.hash == hash && ((k = e.key) == key || (key != null &&
key.equals(k))))
38         break;
39     p = e;
40 }
41 }
42 //如果key存在，则直接覆盖value值
43 if (e != null) { // existing mapping for key
44     v oldValue = e.value;
45     if (!onlyIfAbsent || oldValue == null)
46         e.value = value;
47     afterNodeAccess(e);
48     return oldValue;
49 }
50 }
51 ++modCount;
52 //判断HashMap中节点数是否大于临界值，如果大于则扩容，是之前的两倍
53 if (++size > threshold)
54     resize();
55 afterNodeInsertion(evict);
56 return null;
57 }

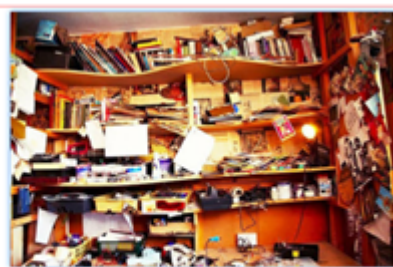
```

2.3 HashMap底层数据结构

1、什么是数据结构？



良好的数据结构



很差的数据结构

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关。

数据结构：就是存储数据的一种方式。ArrayList LinkedList

2、HashMap的数据结构

在JDK1.8之前 HashMap 由 **数组+链表** 数据结构组成的。

在JDK1.8之后 HashMap 由 **数组+链表 +红黑树【哈希表】** 据结构组成的。

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（“拉链法”解决冲突）。

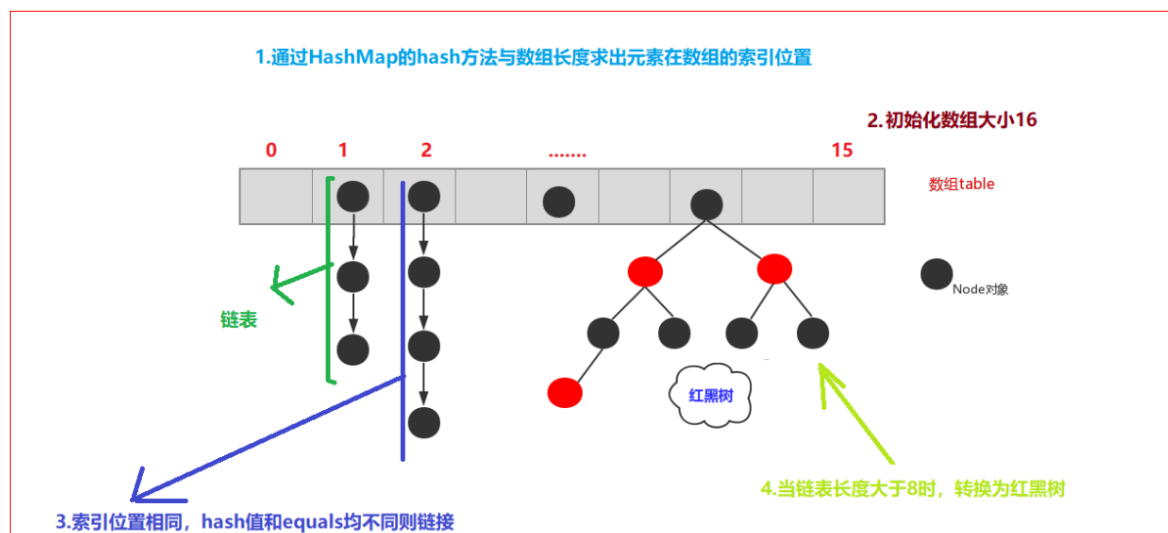
什么是哈希冲突？ 两个对象调用的hashCode方法计算的哈希码值一致导致计算的数组索引值相同。

JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（或者红黑树的边界值，默认为8）并且当前数组的长度大于64时，此时此索引位置上的所有数据改为使用红黑树存储。

JDK1.8引入红黑树大程度优化了HashMap的性能，那么对于我们来讲保证HashSet集合元素的唯一，其实就是根据对象的hashCode和equals方法来决定的。如果我们往集合中存放自定义的对象，那么保证其唯一，就必须复写hashCode和equals方法建立属于当前对象的比较方式。

当位于一个链表中的元素较多，即hash值相等但是内容不相等的元素较多时，通过key值依次查找的效率较低。而**JDK1.8**中，哈希表存储采用数组+链表+红黑树实现，当链表长度(阈值)超过 8 时且当前数组的长度 > 64时，将链表转换为红黑树，这样大大减少了查找时间。jdk8在哈希表中引入红黑树的原因只是为了查找效率更高。

简单的来说，哈希表是由数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的。如下图所示。



3、数据结构的源码

table用来初始化(必须是二的n次幂)(重点)

```
1 //存储元素的数组
2 transient Node<K,V>[] table;
```

用来存缓存

```
1 //存放具体元素的集合
2 transient Set<Map.Entry<K,V>> entrySet;
```

HashMap中存放元素的个数(重点)

```
1 //存放元素的个数, 注意这个不等于数组的长度。
2 transient int size;
```

三、HashMap源码分析

3.1 HashMap的默认初始化容量

初始化容量16

```
1 //默认的初始容量是16 -- 1<<4相当于1*2的4次幂---1*16
2 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
```

初始化容量必须是2的n次幂，为什么？

向HashMap中添加元素时，要根据key的hash值去确定其在数组中的具体位置。HashMap为了存取高效，要尽量较少碰撞，就是要尽量把数据分配均匀，每个链表长度大致相同。怎么让元素均匀分配呢？这里用到的算法是 $\text{hash} \& (\text{length}-1)$ 。hash值与数组长度减一的位运算。算法本质作用是类似于取模， $\text{hash} \% \text{length}$ 。但是计算机中直接求余效率远不如位运算。

$\text{hash} \% \text{length}$ 取模效果操作等于 $\text{hash} \& (\text{length}-1)$ 的前提，length是2的n次幂！

为什么这样能均匀分布减少碰撞呢？2的n次幂实际就是1后面n个0，2的n次幂-1 实际就是n个1；

举例：**位运算规则说明：按&位运算，相同的二进制数位上，都是1的时候，结果为1，否则为零。**

```
1 例如：数组长度8时候，均匀分布在数组中，哈希碰撞的几率比较小；
2
3 求位运算结果：
4 314924944 & (8-1) = 0
5
6 00010010110001010101111110010000
7 00000000000000000000000000000111
8 -----
9 00000000000000000000000000000000 --> 结果为0
10
11 程序员计算器求解：
12 314924944 & (8-1) = 0
13 314924945 & (8-1) = 1
14 314924946 & (8-1) = 2
15 314924947 & (8-1) = 3
16 314924948 & (8-1) = 4
17 314924949 & (8-1) = 6
18 314924950 & (8-1) = 7
19 314924951 & (8-1) = 8
20 314924952 & (8-1) = 0
21
22 结论是：数组索引存储的数据均匀分布了，减少哈希碰撞的几率
```

```
1 例如：数组长度10时候，没有均匀分布，碰撞几率比较大；
2
3 程序员计算器求解：
4 314924944 & (10-1) = 0
5 314924945 & (10-1) = 1
6 314924946 & (10-1) = 0
```



```

7 314924947 & (10-1) = 1
8 314924948 & (10-1) = 0
9 314924949 & (10-1) = 1
10 314924950 & (10-1) = 0
11 314924951 & (10-1) = 1
12 314924952 & (10-1) = 0
13
14
15 结论是：数据全部分布在第一个和第二个索引位置上，大大增加了哈希碰撞的几率。效率低下

```

手动设置初始化容量

HashMap构造方法还可以指定集合的初始化容量大小：

```

1 HashMap(int initialCapacity) 构造一个带指定初始容量和默认加载因子（0.75）的空
  HashMap。

```

注意：当然如果不考虑效率问题，求余即可。就不需要长度必须是2的n次幂了。如果采用位运算，必须是2的n次幂！

那么来了，如果有那个蠢蛋不知道，瞎搞。HashMap也自带纠错能力。具备防蠢货功能。如果创建HashMap对象时，输入的数组长度不是2的n次幂，HashMap通过一通位移运算和或运算得到的肯定是2的幂次数，并且是离那个数最近的数字。

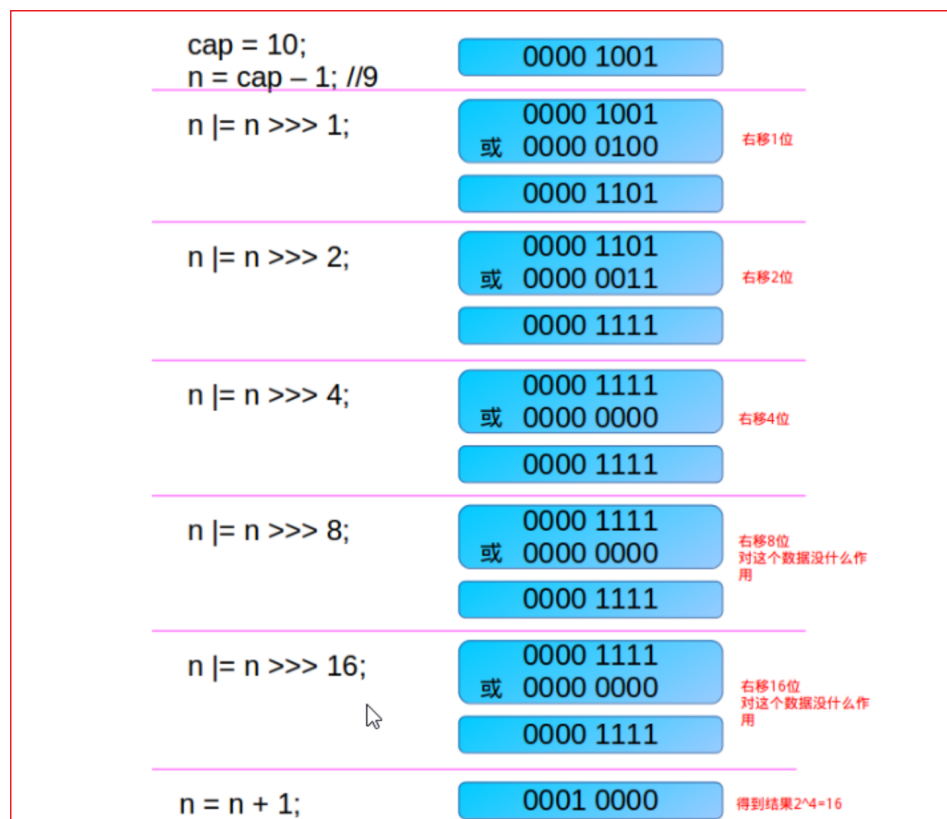
```

1 //创建HashMap集合的对象，指定数组长度是10，不是2的幂
2 HashMap hashMap = new HashMap(10);
3 public HashMap(int initialCapacity) { //initialCapacity=10
4     this(initialCapacity, DEFAULT_LOAD_FACTOR);
5 }
6 public HashMap(int initialCapacity, float loadFactor) { //initialCapacity=10
7     if (initialCapacity < 0)
8         throw new IllegalArgumentException("Illegal initial capacity: "
9 +
10                                     initialCapacity);
11     if (initialCapacity > MAXIMUM_CAPACITY)
12         initialCapacity = MAXIMUM_CAPACITY;
13     if (loadFactor <= 0 || Float.isNaN(loadFactor))
14         throw new IllegalArgumentException("Illegal load factor: " +
15                                     loadFactor);
16     this.loadFactor = loadFactor;
17     this.threshold = tableSizeFor(initialCapacity); //initialCapacity=10
18 }
19 /**
20  * Returns a power of two size for the given target capacity.
21  */
22 static final int tableSizeFor(int cap) { //int cap = 10
23     int n = cap - 1;
24     n |= n >> 1;
25     n |= n >> 2;
26     n |= n >> 4;
27     n |= n >> 8;
28     return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
29 }

```




加入给的初始化容量为10，最终容量会变为最近的16！



小结：

1. 根据key的hash确定存储位置时，数组长度是2的n次幂，可以保证数据的均匀插入。如果不是，会浪费数组的空间，降低集合性能！
2. 一般情况下，我们通过求余%来均匀分散数据。只不过其性能不如位运算【&】。
3. length的值为2的n次幂，`hash & (length - 1)` 作用完全等同于 `hash % length`。
4. HashMap中初始化容量为2次幂原因是为了数组数据均匀分布。尽可能减少哈希冲突，提升集合性能。
5. 即便可以手动设置HashMap的初始化容量，但是最终还是会被重设为2的n次幂。

3.2 HashMap的加载因子0.75和最大容量

1、加载因子相关属性

哈希表的加载因子(重点)

```
1 // 加载因子
2 final float loadFactor;
```

默认的加载因子，默认值是0.75，决定了扩容的条件

```
1 static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

集合最大容量：10 7374 1824 【10亿】

```
1 //集合最大容量的上限是：2的30次幂
2 static final int MAXIMUM_CAPACITY = 1 << 30;
```

扩容的临界值：计算方式为(容量 乘以 加载因子)

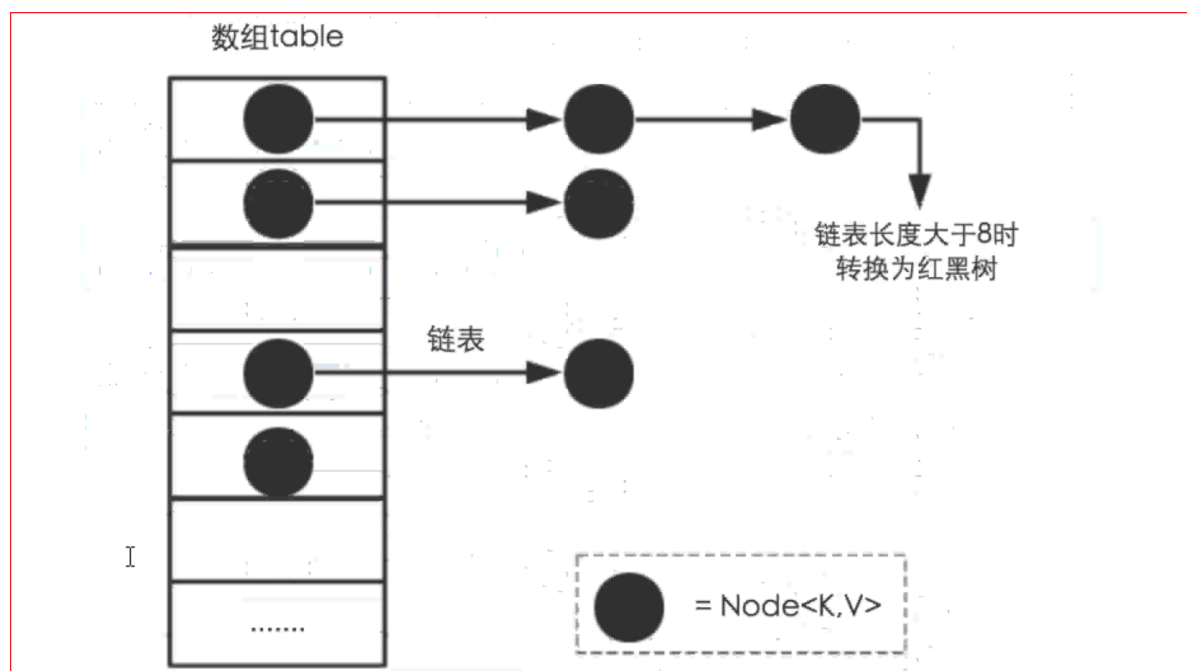
```
1 // 临界值 当实际大小超过临界值时，会进行扩容
2 int threshold;
```

。

2、为什么加载因子设置为0.75，初始化临界值是12？

loadFactor太大导致查找元素效率低，小导致数组的利用率低，存放的数据会很分散。loadFactor的默认值为0.75f是官方给出的一个比较好的临界值。

- 加载因子越大趋近于1，数组中的存放的数据(Node)也就越多，也就越稠密，也就是会让链表的长度增加。
- 加载因子越小趋近于0，数组中的存放的数据(Node)也就越少，也就越稀疏。也就是会让链表的长度不会太长。



如果希望链表尽可能少些，性能更好。就要提前扩容，但导致的问题是数组空间浪费，有些桶没有存储数据！典型的鱼与熊掌不可兼得！



```
1 例如：加载因子是0.4。 那么16*0.4--->6 如果数组中满6个空间就扩容会造成数组利用率太低了。
2 加载因子是0.9。 那么16*0.9---->14 那么这样就会导致链表有点多了。导致查找元素效率低。
```

所以既兼顾数组利用率又考虑链表不要太多，经过大量测试0.75是最佳方案。

threshold计算公式: $\text{capacity}(\text{数组长度默认}16) * \text{loadFactor}(\text{加载因子默认}0.75)$ 。

这个值是当前已占用数组长度的最大值。当 $\text{size} \geq \text{threshold}$ 的时候, 那么就要考虑对数组的resize(扩容)。扩容后的新HashMap容量是之前容量的两倍。

3、修改加载因子

同时在HashMap的构造器中可以定制loadFactor。但最好别改~

```
1 构造方法:  
2  HashMap(int initialCapacity, float loadFactor) 构造一个带指定初始容量和加载因子的空  
   HashMap。
```

3.3 HashMap的红黑树转换边界值详解

1、边界转换相关属性

1.转换边界值1: 当链表超过转换边界值8, 就会转红黑树(**1.8新增**), 前提条件: 数组长度大于64。

```
1  //当桶(bucket)上的结点数大于这个值时会转成红黑树  
2  static final int TREEIFY_THRESHOLD = 8;
```

2.转换边界值2: 当Map里面的数量超过这个值时, 表中的桶才能进行树形化, 这个值不能小于 $4 * \text{TREEIFY_THRESHOLD}$ (8)

```
1  //桶中结构转化为红黑树对应的数组长度最小的值  
2  static final int MIN_TREEIFY_CAPACITY = 64;
```

3.桶(bucket): 所谓的桶, 指的是一个数组索引位中的所有元素。

4.降级转换边界值: 当链表的值小于6则会从红黑树转回链表

```
1  //当桶(bucket)上的结点数小于这个值时树转链表  
2  static final int UNTREEIFY_THRESHOLD = 6;
```

2、为什么Map桶中节点个数超过8才转为红黑树?

HashMap的红黑树数据结构几乎不会被用到, 本质上还是一个链表+数组!!!

8阈值定义在HashMap中, 针对这个成员变量, 在源码的注释中只说明了8是bin (bin就是bucket(桶))从链表转成树的阈值, 但是并没有说明为什么是8:

在HashMap官方注释说明:

```
jdk1.8.0_171\...\HashMap.java x Integer.java x jdk1.6.0_45\...\HashMap.java x Cloneable.java x
...
*
* Because TreeNodes are about twice the size of regular nodes, we
* use them only when bins contain enough nodes to warrant use
* (see TREEIFY_THRESHOLD). And when they become too small (due to
* removal or resizing) they are converted back to plain bins. In
* usages with well-distributed user hashCodes, tree bins are
* rarely used. Ideally, under random hashCodes, the frequency of
* nodes in bins follows a Poisson distribution
* (http://en.wikipedia.org/wiki/Poisson\_distribution) with a
* parameter of about 0.5 on average for the default resizing
* threshold of 0.75, although with a large variance because of
* resizing granularity. Ignoring variance, the expected
* occurrences of list size k are (exp(-0.5) * pow(0.5, k) /
* factorial(k)). The first values are:
*
* 0: 0.60653066
* 1: 0.30326533
* 2: 0.07581633
* 3: 0.01263606
* 4: 0.00157952
* 5: 0.00015795
* 6: 0.00001316
* 7: 0.00000094
* 8: 0.00000006
* more: less than 1 in ten million
*
```

```
1 //翻译后内容
2 因为树节点的大小大约是普通节点的两倍，所以我们只在箱子包含足够的节点时才使用树节点(参见
  TREEIFY_THRESHOLD)。当它们变得太小(由于删除或调整大小)时，就会被转换回普通的桶。在使用
  分布良好的用户hashCode时，很少使用树。
3 想情况下，在随机哈希码下，桶中节点的频率服从泊松分布，默认调整阈值为0.75，平均参数约为
  0.5，尽管由于调整粒度的差异很大。忽略方差，列表大小k的预期出现次数是(exp(-0.5)*pow(0.5,
  k)/factorial(k))。
4 第一个值是：
5 0: 0.60653066
6 1: 0.30326533
7 2: 0.07581633
8 3: 0.01263606
9 4: 0.00157952
10 5: 0.00015795
11 6: 0.00001316
12 7: 0.00000094
13 8: 0.00000006
14 more: less than 1 in ten million
```

红黑树节点对象占用空间是普通链表节点的两倍，所以只有当桶中包含足够多的节点时才会转成红黑树。当桶中节点数变少时，又会转成普通链表。并且我们查看源码的时候发现，链表长度达到8就转成红黑树，当长度降到6就转成链表。

这样就解释了为什么不是开始就将其转换为红黑树节点，而是数量数才转。

说白了就是空间和时间的权衡！

官方还说了：当hashCode哈希函数值离散性很好的情况下。红黑树被用到的概率非常小！概率为0.00000006。

理想的情况下，优秀的hash算法，会让所有桶的节点的分布频率会遵循泊松分布。我们可以看到，一个桶中链表长度达到8个元素的概率为0.00000006，几乎是不可能事件。因为数据被均匀分布在每个桶中，所以几乎不会有桶中的链表长度会达到阈值！

所以，之所以选择8，不是随便决定的，而是根据概率统计决定的。由此可见，发展将近30年的Java每一项改动和优化都是非常严谨和科学的。也就是说：选择8因为符合泊松分布，超过8的时候，概率已经非常小了，所以我们选择8这个数字。

但是哈希函数【hashCode】是有用户控制，用户选择的hash函数，离散性可能会很差。JDK又不能阻止用户实现这种不好的hash算法。因此，就可能导致不均匀的数据分布。所以超过8了，就采用红黑树，来提升效率。

扩展：Poisson分布(泊松分布)，是一种统计与概率学里常见到的离散[概率分布]。泊松分布的概率函数为：

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}, k = 0, 1, \dots$$

泊松分布的参数 λ 是单位时间(或单位面积)内随机事件的平均发生次数。泊松分布适合于描述单位时间内随机事件发生的次数。

3.5 HashMap的treeifyBin()方法详解-链表转红黑树

1、转换相关属性

1.转换边界值1：当链表超过转换边界值8，就会转红黑树(1.8新增)，前提条件：数组长度大于64。

```
1 //当桶(bucket)上的结点数大于这个值时会转成红黑树
2 static final int TREEIFY_THRESHOLD = 8;
```

2.转换边界值2：当Map里面的数量超过这个值时，表中的桶才能进行树形化，这个值不能小于 $4 * TREEIFY_THRESHOLD$ (8)

```
1 //桶中结构转化为红黑树对应的数组长度最小的值
2 static final int MIN_TREEIFY_CAPACITY = 64;
```

3.桶(bucket)：所谓的桶，指的是一个数组索引位中的所有元素。

4.降级转换边界值：当链表的值小于6则会从红黑树转回链表

```
1 //当桶(bucket)上的结点数小于这个值时树转链表
2 static final int UNTREEIFY_THRESHOLD = 6;
```

2、转换treeifyBin()方法源码分析

节点添加完成之后判断此时节点个数是否大于TREEIFY_THRESHOLD临界值8，如果大于则将链表转换为红黑树，转换红黑树的方法 treeifyBin，整体代码如下：

```
1 if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
2     //转换为红黑树 tab表示数组名 hash表示哈希值
3     treeifyBin(tab, hash);
```

treeifyBin方法如下所示：

```

1 //对HashMap集合中的桶的链表转为红黑树
2 //如果集合中数组太短，会对数组进行扩容
3 final void treeifyBin(Node<K,V>[] tab, int hash) {
4     int n, index; Node<K,V> e;
5     /*
6         如果当前数组为空或者数组的长度小于进行树形化的阈值(MIN_TREEIFY_CAPACITY
7         = 64),
8         就去扩容。而不是将节点变为红黑树。
9         目的：如果数组很小，那么转换红黑树，然后遍历效率要低一些。这时进行扩容，那么
10        重新计算哈希值
11        ，链表长度有可能就变短了，数据会放到数组中，这样相对来说效率高一些。
12        */
13 //判断数组是否满足转红黑树最小长度。原因：数组太短，转为红黑树不仅没有提高效率，反而
14 降低了。
15 if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
16     //如果不满足最小长度64，则对数组进行扩容
17     resize();
18 //判断数组中桶内非空，并且获取桶中第一个节点
19 else if ((e = tab[index = (n - 1) & hash]) != null) {
20     //开始链表转红黑树
21     //hd: 红黑树的头结点
22     //tl :红黑树的尾结点
23     TreeNode<K,V> hd = null, tl = null;
24     do {
25         //新建一个树的节点，内容和当前链表节点e一致
26         TreeNode<K,V> p = replacementTreeNode(e, null);
27         if (tl == null)
28             //将新创键的p节点赋值给红黑树的头结点
29             hd = p;
30         else {
31             p.prev = tl; //将上一个节点p赋值给现在的p的前一个节点
32             tl.next = p; //将现在节点p作为树的尾结点的下一个节点
33         }
34         tl = p;
35         //e = e.next 将当前节点的下一个节点赋值给e,如果下一个节点不等于null
36         //则回到上面继续取出链表中节点转换为红黑树
37     } while ((e = e.next) != null);
38     //将桶中的第一个元素，替换为红黑树节点。
39     if ((tab[index] = hd) != null)
40         hd.treeify(tab);
41 }
42 }

```

小结:

1. HashMap集合中，链表节点红黑树节点的临界值是8，前提是集合中数组的最大容量是64以上。否则会对数组进行扩容！

3.6 HashMap的扩容机制

在不断的添加数据的过程中，会涉及到扩容问题，当超出临界值(且要存放的位置非空)时，扩容。默认的扩容方式：扩容为原来容量的2倍，并将原有的数据复制过来。

1、扩容相关属性

1.扩容计数器：用来记录HashMap的修改次数

```
1 // 每次扩容和更改map结构的计数器
2 transient int modCount;
```

2.转换边界值1：当链表超过转换边界值8，就会转红黑树(1.8新增)，前提条件：数组长度大于64。

```
1 //当桶(bucket)上的结点数大于这个值时会转成红黑树
2 static final int TREEIFY_THRESHOLD = 8;
```

3.转换边界值2：当Map里面的数量超过这个值时，表中的桶才能进行树形化，这个值不能小于 $4 * TREEIFY_THRESHOLD$ (8)

```
1 //桶中结构转化为红黑树对应的数组长度最小的值
2 static final int MIN_TREEIFY_CAPACITY = 64;
```

4.桶(bucket)：所谓的桶，指的是一个数组索引位中的所有元素。

5.哈希表的加载因子(重点) 默认值是0.75，决定了扩容的条件

```
1 // 加载因子
2 final float loadFactor;
3 static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

6.集合最大容量：10, 7374, 824【10亿】

```
1 //集合最大容量的上限是：2的30次幂
2 static final int MAXIMUM_CAPACITY = 1 << 30;
```

7.扩容的临界值：计算方式为(容量 乘以 加载因子)

```
1 // 临界值 当实际大小超过临界值时，会进行扩容
2 int threshold;
```

2、扩容机制

了解HashMap的扩容机制，你需要搞懂这两个问题：

1. 什么时候需要扩容？
2. HashMap的扩容做了那些事？

问题1：什么时候需要扩容？

主要在两种种情况下进行扩容：

1. 当HashMap集合中，实际存储元素个数超过临界值(threshold)时，会进行扩容。默认初始化临界值是12。
2. 当HashMap中，单个桶的链表长度达到了8，并且数组长度还没到达64。会进行扩容

问题2：HashMap的扩容做了那些事？

将原数组中桶内的节点，均匀分散在了新的数组的桶中。

HashMap扩容时分散使用的rehash方式非常巧妙。并没有进行hash函数调用。

由于每次扩容都是翻倍，与原来计算的 $(n-1) \& \text{hash}$ 的结果相比，只是多了一个bit位。所以节点要么就在原来的位置，要么就被分配到“原位置+旧容量”这个位置。

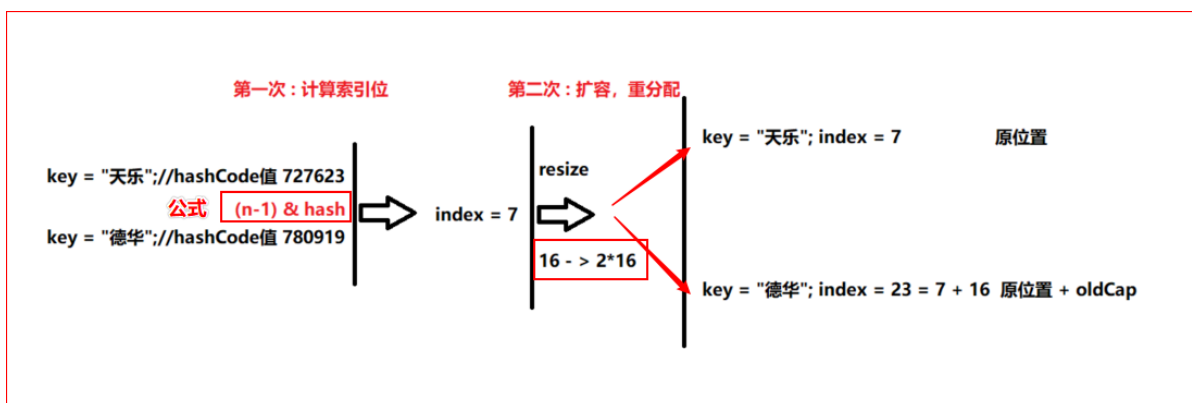


我举个栗子

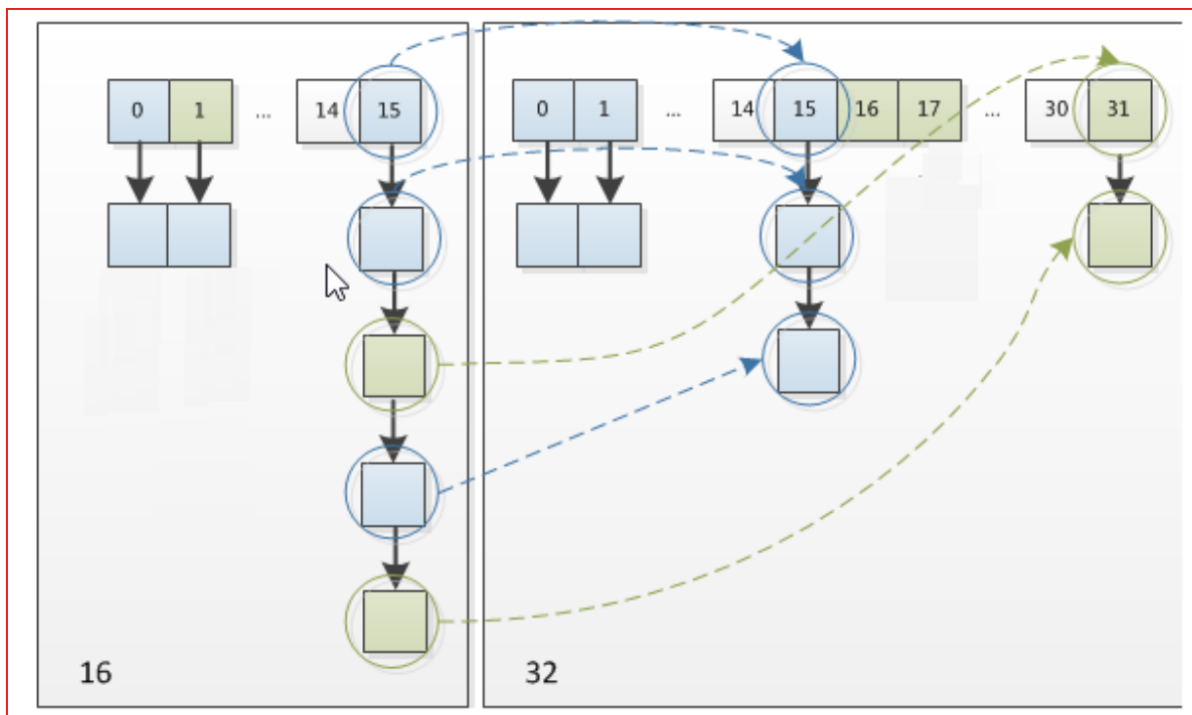
怎么理解呢？例如我们从16扩展为32时，具体的变化如下所示：

```
1  "娜扎"的哈希值740274
2  公式：  $(n-1) \& \text{hash}$ 
3   $740274 \& (16-1) = 2$ 
4
5  String key = "天乐";//hashCode值 727623
6  String key = "德华";//hashCode值 780919
7  计算hash
8   $727623 \& (16-1) = 7$ 
9   $780919 \& (16-1) = 7$ 
10
11 扩容之后，巧妙的计算rehash【更高效】
12   $727623 \& (32-1) = 7$ 
13   $780919 \& (32-1) = 7 + 16 = 23$ 
14  //结论："原位置+旧容量"
```

画图说明：



下图为16扩充为32的resize示意图：将原数组中桶内的节点，均匀分散在了新的数组的桶中。



结论：

1. 7是是两个字符串计算的原始索引，存入同一个桶中。扩容之后，"天乐"在原来位置，"德华"被分配"原位置+旧容量"位置。
2. 因此我们在扩充HashMap时，不需要重新计算hash。只需要看原来的hash值新增bit是1还是0就可以了！
 1. 是0索引没变
 2. 是1索引变成"原索引+oldCap(原位置+旧容量)"。

注意：扩容必定伴随rehash操作，遍历hash表中所有元素。这种操作比较耗时！在编程中，超大HashMap要尽量避免resize，避免的方法之一就是初始化固定HashMap大小！

3、扩容方法resize()源码解读

下面是代码的具体实现：

```

1  final Node<K,V>[] resize() {
2      //得到当前数组
3      Node<K,V>[] oldTab = table;
4      //如果当前数组等于null长度返回0，否则返回当前数组的长度
5      int oldCap = (oldTab == null) ? 0 : oldTab.length; //原始数组容量
6      //当前阈值点 默认是12(16*0.75)
7      int oldThr = threshold; //原始阈值
8      //新的阈值点newThr
9      //新数组容量
10     int newCap, newThr = 0;
11     //如果老的数组长度大于0
12     if (oldCap > 0) {
13         // 超过最大值就不再扩充了，就只好随你碰撞去吧
14         if (oldCap >= MAXIMUM_CAPACITY) {
15             //修改阈值为int的最大值
16             threshold = Integer.MAX_VALUE;
17             return oldTab;
18         }

```

```

19 //没超过最大值，就扩充为原来的2倍
20 //1)(newCap = oldCap << 1) < MAXIMUM_CAPACITY 扩大到2倍之后容量要小于最
    大容量
21 //2) oldCap >= DEFAULT_INITIAL_CAPACITY 原数组长度大于等于数组初始化长度
    16
22     else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
23             oldCap >= DEFAULT_INITIAL_CAPACITY)
24         //阈值扩大一倍
25         newThr = oldThr << 1; // double threshold
26     }
27     //如果原始数组为空，且原始的扩容阈值大于0，原始阈值赋值给新的数组容量
28     else if (oldThr > 0)
29         newCap = oldThr; //原始阈值赋值给新的数组容量
30     else {
31         //如果原始数组为空，扩容阈值为0，则设置为默认值
32         newCap = DEFAULT_INITIAL_CAPACITY; //16
33         newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
34     }
35     // 如果新的扩容阈值为0，则重新计算新的resize最大上限
36     if (newThr == 0) {
37         //计算公式：新数组容量 * 加载因子
38         float ft = (float)newCap * loadFactor;
39         newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
    ?
40             (int)ft : Integer.MAX_VALUE);
41     }
42     //赋值新的扩容阈值
43     threshold = newThr;
44     //创建新的哈希表
45     @SuppressWarnings({"rawtypes", "unchecked"})
46     //创建新的数组，容量是之前的2倍。newCap是新的数组长度--》32
47     Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
48     table = newTab;
49     //过滤，排除旧数组为空的情况
50     if (oldTab != null) {
51         //遍历旧的哈希表的每个桶，重新计算桶里元素的新位置
52         for (int j = 0; j < oldCap; ++j) { //把每个bucket都移动到新的buckets中
53             Node<K,V> e;
54             if ((e = oldTab[j]) != null) {
55                 //原来的数据赋值为null 便于GC回收
56                 oldTab[j] = null;
57                 //判断数组是否有下一个引用
58                 if (e.next == null)
59                     //没有下一个引用，说明不是链表，当前桶上只有一个键值对，直接插入
60                     newTab[e.hash & (newCap - 1)] = e;
61                 //判断是否是红黑树
62                 else if (e instanceof TreeNode)
63                     //说明是红黑树来处理冲突的，则调用相关方法把树分开
64                     ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
65                 else { // 采用链表处理冲突
66                     Node<K,V> loHead = null, loTail = null;
67                     Node<K,V> hiHead = null, hiTail = null;
68                     Node<K,V> next;
69                     //通过上述讲解的原理来计算节点的新位置
70                     do {
71                         // 原索引
72                         next = e.next;

```

```

73 //这里来判断如果等于true e这个节点在resize之后不需要移动位
置
74 if ((e.hash & oldCap) == 0) {
75     if (loTail == null)
76         loHead = e;
77     else
78         loTail.next = e;
79     loTail = e;
80 }
81 // 原索引+oldCap
82 else {
83     if (hiTail == null)
84         hiHead = e;
85     else
86         hiTail.next = e;
87     hiTail = e;
88 }
89 } while ((e = next) != null);
90 // 原索引放到bucket里
91 if (loTail != null) {
92     loTail.next = null;
93     newTab[j] = loHead;
94 }
95 // 原索引+oldCap放到bucket里
96 if (hiTail != null) {
97     hiTail.next = null;
98     newTab[j + oldCap] = hiHead;
99 }
100 }
101 }
102 }
103 }
104 return newTab;
105 }

```

3.7 HashMap容量初始化最佳策略

1、HashMap的初始化问题描述

《阿里巴巴Java开发手册》中建议初始化HashMap的容量。

10. 【推荐】集合初始化时，指定集合初始值大小。

说明：HashMap 使用 `HashMap(int initialCapacity)` 初始化。

为什么要建议初始化HashMap容量？

- 防止自动扩容，影响效率！

当然阿里的建议是有理论支撑的。我们上面介绍过HashMap的扩容机制，就是当达到扩容条件时会进行扩容。HashMap的扩容条件就是当HashMap中的元素个数（size）超过临界值（threshold）时就会自动扩容。在HashMap中， $\text{threshold} = \text{loadFactor} * \text{capacity}$ 。

所以，如果我们没有设置初始容量大小，随着元素的不断增加，HashMap会有可能发生多次扩容，而HashMap中的扩容机制决定了每次扩容都需要重建hash表，是非常影响性能的。

设置初始化容量，数值不同性能也不一样！

当已知HashMap中，将存放的键值对个数时，容量设置成多少合适呢？

是直接设置键值个数吗？并不是，我接下来看

2、HashMap中容量初始化多少合适？

假如现在HashMap集合要存入，16个元素。如果你初始化16个。集合总容量是16，扩容阈值是12。最终HashMap集合在存入到12个元素时，会进行一次扩容操作。这样会导致性能损耗。

有没有更好的办法？

《阿里巴巴Java开发手册》有以下建议：

正例：`initialCapacity = (需要存储的元素个数 / 负载因子) + 1`。注意负载因子(即 `loader factor`) 默认为 0.75，

如果我们通过`initialCapacity / 0.75F + 1.0F`计算： $16 / 0.75 + 1 = 22$ 。

22经过jdk处理之后【2的n次幂】，集合的初始化容量会被设置成32。

集合总容量是32，扩容阈值是24。最终HashMap集合在存入到16个元素时，完全不会进行扩容。榨取最后一滴性能！

有利必有弊，这样的做法会增加数组的无效容量，牺牲一小部分内存。出于对性能的极值追求，这部分牺牲是值得的！

四、HashMap面试题精讲

1、HashMap中hash函数是怎么实现的？还有哪些hash函数的实现方式？

底层采用的是key的hashCode方法 加 异或(^) 加 无符号右移(>>>)操作计算出hash值。

```
1 static final int hash(Object key) {
2     int h;
3     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
4 }
```

而哈希表中，计算数组索引的方法是位运算，如下代码。保证所有的hash最终落在数组最大索引范围之内。

```
1 i = (n - 1) & hash
```

还可以采用：取余法、平方取中法，伪随机数法。

取余数方式较为常见 `10%8`，但是与位运算相比，效率较低。

2、当两个对象的hashCode相等时会怎么样？

- 1 会产生哈希碰撞
- 2 如果key值相同，则替换旧的value。不然连接到链表后面，链表长度超过阈值8，数组长度大于64自动转换为红黑树存储。

3、何时发生哈希碰撞和什么是哈希碰撞,如何解决哈希碰撞?

- 1 只要两个元素的key计算的哈希码值相同就会发生哈希碰撞。
- 2 jdk8前使用链表解决哈希碰撞。
- 3 jdk8及以后使用链表+红黑树解决哈希碰撞。

4、 hashCode 和 equals 方法有何重要性?

非常重要!

1.HashMap 使用 key 对象的 `#hashCode()` 和 `#equals(Object obj)` 方法去决定键值对的存储位置。

当从 HashMap 中获取值时, 也会被用到

- 如果这两个方法没有被正确地实现, 两个不同 Key 也许会产生相同的 `#hashCode()` 和 `#equals(Object obj)` 输出。这就会导致元素存储位置的混乱, 降低HashMap性能。

2. `#hashCode()` 和 `#equals(Object obj)` 保证了集合元素的唯一性

所有不允许存储重复数据的集合类, 都使用使用这两个方法, 所以正确实现它们非常重要。

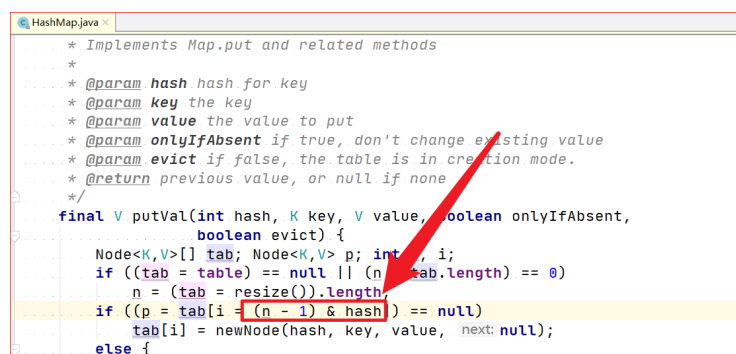
- 如果 `o1.equals(o2)`, 那么 `o1.hashCode() == o2.hashCode()` 总是为 `true` 的。
- 如果 `o1.hashCode() == o2.hashCode()`, 并不意味 `o1.equals(o2)` 会为 `true`。

5、 HashMap 默认容量是多少?

- 默认容量都是 16, 加载因子是 0.75。
- 就是当 HashMap 填充了 75% 就会扩容, 最小扩容阈值是 ($16 * 0.75 = 12$)
- 扩容一般会扩为原内存 2 倍。

6、 HashMap 的长度为什么是2的n次幂?

为了能让 HashMap 存取高效, 尽量较少碰撞, 也就是要尽量把数据分配均匀分散在数组中。这个问题的关键在与存储数组索引位的算法【`hash & (length - 1)`】。



```
... * Implements Map.put and related methods
... *
... * @param hash hash for key
... * @param key the key
... * @param value the value to put
... * @param onlyIfAbsent if true, don't change existing value
... * @param evict if false, the table is in creation mode.
... * @return previous value, or null if none
... */
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = new Node(hash, key, value, next: null);
    else {
```

这个算法可以如何设计呢?

1. 程序员们一般都会想到 `%` 取余, 但是效率太低。
2. 在计算机运算中, 位运算高于取余操作。而位运算能够做到与取余相同效果的前提是, 数组长度是 2 的 n 次幂。

这就解释了 HashMap 的长度为什么是 2 的幂次幂。

7、加载因子值的大小对HashMap有什么影响？

加载因子的大小决定了HashMap的数据密度

- 加载因子越大，数据密度越大，发生碰撞概率越高，数组桶中链表越长，查询或者插入时的比较次数增多。从而导致性能下降。
- 加载因子越小，数据密度越小，发生碰撞概率越小，数组桶中链表越短，查询和插入时比较的次数也就越小。性能会更高。
 - 加载因子越小，越容易触发扩容，会影响性能
 - 加载因子越小，存储数据量少，会浪费内存空间

鱼与熊掌不可兼得！

按照其他语言的参考及研究经验，会考虑将加载因子设置为0.7到0.75