HashMap

(1) HashMap：它根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。

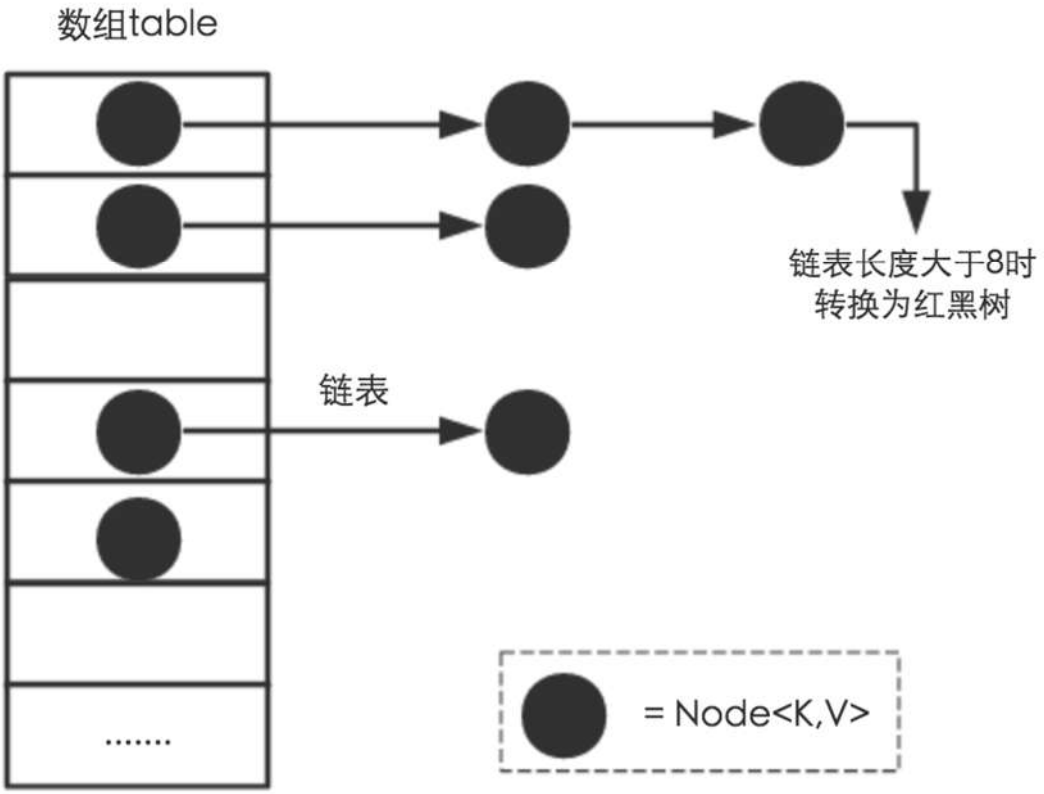
(2) Hashtable：Hashtable 是遗留类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类，并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁。Hashtable 不建议在新代码中使用，不需要线程安全的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换。

(3) LinkedHashMap：LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

(4) TreeMap：TreeMap 实现 SortedMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。如果使用排序的映射，建议使用 TreeMap。在使用 TreeMap 时，key 必须实现 Comparable 接口或者在构造 TreeMap 传入自定义的 Comparator，否则会在运行时抛出 java.lang.ClassCastException 类型的异常。

存储结构-字段

从结构实现来讲，HashMap 是数组+链表+红黑树



HashMap 就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java 中 HashMap 采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被 Hash 后，得到数组下标，把数据放在对应下标元素的链表上。

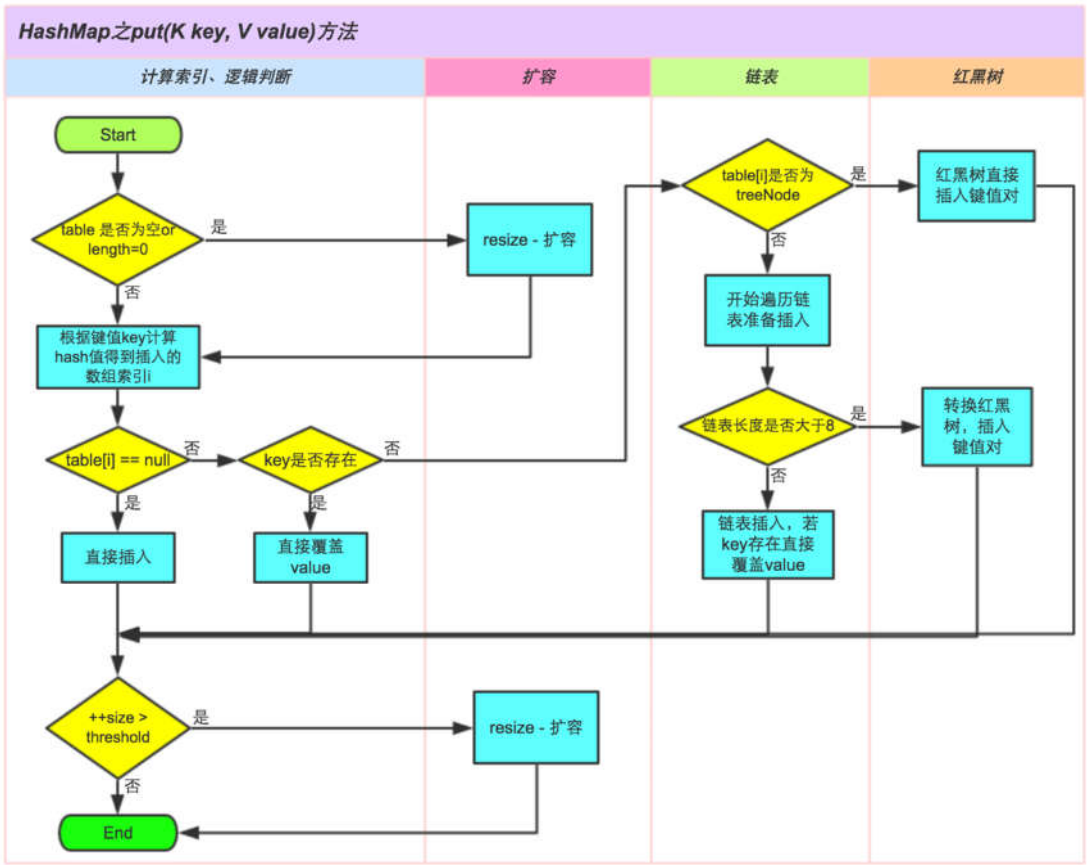
通过什么方式来控制 map 使得 Hash 碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的 Hash 算法和扩容机制。

在 HashMap 中，哈希桶数组 table 的长度 length 大小必须为 2 的 n 次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考 http://blog.csdn.net/liuqiyao_01/article/details/14475159，Hashtable 初始化桶大小为

11，就是桶大小设计为素数的应用（Hashtable 扩容后不能保证还是素数）。HashMap 采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap 定位哈希桶索引位置时，也加入了高位参与运算的过程。

Hash 算法本质上就是三步：**取 key 的 hashCode 值、高位运算、取模运算。**

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。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。



这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有 Entry

数组的元素拷贝到新的 Entry 数组里。

下面我们讲解下 JDK1.8 做了哪些优化。经过观测可以发现，我们使用的是 2 次幂的扩展(指长度扩为原来 2 倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动 2 次幂的位置。看下图可以明白这句话的意思，n 为 table 的长度，图（a）表示扩容前的 key1 和 key2 两种 key 确定索引位置的示例，图（b）表示扩容后 key1 和 key2 两种 key 确定索引位置的示例，其中 hash1 是 key1 对应的哈希与高位运算结果。

元素在重新计算 hash 之后，因为 n 变为 2 倍，那么 n-1 的 mask 范围在高位多 1bit(红色)，因此新的 index 就会发生这样的变化：

因此，我们在扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 hash，只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了，是 0 的话索引没变，是 1 的话索引变成“原索引+oldCap”，可以看看下图为 16 扩充为 32 的 resize 示意图：
这个设计确实非常的巧妙，既省去了重新计算 hash 值的时间，而且同时，由于新增的 1bit 是 0 还是 1 可以认为是随机的，因此 resize 的过程，均匀的把之前的冲突的节点分散到新的 bucket 了。这一块就是 JDK1.8 新增的优化点。