**Java 散列表 hash table**

@author ixenos

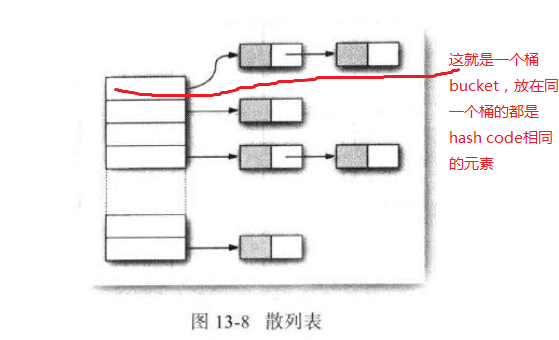
hash table, HashTable, HashMap, HashSet

* **hash table 是一种数据结构**

hash table 为每个对象计算一个整数，该整数被称为散列码 hash code

* + hash code 是由对象的实例域产生的一个整数，具有不同的数据域的对象将产生不同的hash code
  + 如果自定义类，就要负责实现这个类的hashCode方法，注意要与equals方法兼容，即如果a.equals(b)为true，则a与b的hash code必须相同

* **Java中hash table用链表数组实现（即数组元素是链表），为java.util.HashTable<K,V>，每个List被称为桶bucket。**
  + **桶：**bucket，用于收集具有相同hash code的元素。要想查找table中对象的位置，就要先计算它的hash code，然后与bucket的总数取余，得到的就是保存这个元素的bucket的index

**  //图片来自《Core Java》**

* + - 如果bucket中没有其他元素，此时将元素直接插入bucket中就可以了；如果bucket中有元素，需要用新对象与该bucket中所有的对象进行比较，查看这个对象是否已经存在，不存在则修改链表结点索引加入bucket；如果bucket被占满，此现象被称为散列冲突hash collision，此时需要用新对象与该bucket中所有的对象进行比较，查看这个对象是否已经存在
  + **散列冲突：**hash collision，如果插入到HashTable中的元素太多，就会增加hash collision的可能性，降低性能，所以要指定一个初始的桶数。通常，将桶数设置为预计元素个数的75%~150%//不同key是可以同hash的，这是hash冲突，由于链表访问效率低，所以尽量避免hash冲突
    - **【API文档】**
    - public class **Hashtable<K,V>**extends [Dictionary](http://www.cnblogs.com/)<K,V>implements [Map](http://www.cnblogs.com/)<K,V>, [Cloneable](http://www.cnblogs.com/), [Serializable](http://www.cnblogs.com/)

此类实现一个哈希表，该哈希表将键映射到相应的值。任何非 null 对象都可以用作键或值。为了成功地在哈希表中存储和获取对象，用作键的对象必须实现 hashCode 方法和 equals 方法。Hashtable 的实例有两个参数影响其性能：*初始容量* 和*加载因子*。*容量* 是哈希表中*桶* 的数量，*初始容量* 就是哈希表创建时的容量。注意，哈希表的状态为 *open*：在发生“哈希冲突”的情况下，单个桶会存储多个条目，这些条目必须按顺序搜索。*加载因子* 是对哈希表在其容量自动增加之前可以达到多满的一个尺度。初始容量和加载因子这两个参数只是对该实现的提示。关于何时以及是否调用 rehash 方法的具体细节则依赖于该实现。

* + **再散列：**rehashed，如果预估过低，HashTable太满，就需要再散列 rehashed 如果装载因子为0.75，而表中超过75%的位置已经填入元素，这个HashTable就会用双倍的桶数再散列rehashed

* **hash table可以用于实现几个重要的数据结构**
  + set类型：set是没有重复元素的元素集合(collection)，add方法首先在集set中查找要添加的对象，不存在就添加进去
  + 比如Java类库中的**HashSet**，它实现了基于hash table的set（基于HashMap）
    - **HashSet内部维护了一个HashMap<E,Object>对象用以存储set对象**，屏蔽了map中的键，大部分方法的实现都借助了HashMap的方法。
    - public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, java.io.Serializable{
    - private transient HashMap<E,Object> map;
    - ...
    - public HashSet() { map = new HashMap<>(); }
    - ...
    - }
    - 比如contains方法的实现
    - **//HashSet的contains方法源码（借助HashMap的方法）**
    - public boolean contains(Object o) {
    - return map.containsKey(o);
    - }
    - **//来自HashMap的源码**
    - final Node<K,V> getNode(int hash, Object key) {
    - Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    - if ((tab = table) != null && (n = tab.length) > 0 &&
    - (first = tab[(n - 1) & hash]) != null) {
    - if (first.hash == hash && // always check first node
    - ((k = first.key) == key || (key != null && key.equals(k))))
    - return first;
    - if ((e = first.next) != null) {
    - if (first instanceof TreeNode)
    - return ((TreeNode<K,V>)first).getTreeNode(hash, key);
    - do {
    - if (e.hash == hash &&
    - ((k = e.key) == key || (key != null && key.equals(k))))
    - return e;
    - } while ((e = e.next) != null);
    - }
    - }
    - return null;
    - }
    - public boolean containsKey(Object key) { **//被HashSet的contains方法调用**
    - return getNode(hash(key), key) != null;
    - }
      * 从源码中可以看出：contains方法能用来快速查看是否某个元素已经出现在集中，因为它只在某个桶中查找元素，而不必查看集合中的所有元素
      * HashSet迭代器将依次访问所有的bucket，由于hash将元素分散在表的各个位置上（只有不关心集合中元素的顺序时才应该使用HashSet），所以访问他们的顺序几乎是随机的（当然这随机是"固定"了的）
  + 由于元素的hash code决定所在bucket，因此修改集set中元素的内部特征（实例域）的时候，要小心hash code码改变导致元素在数据结构中的位置的变化