哈希算法（如MD5）：这类算法接受任意长度的二进制输入值，对输入值进行换算，最终给出固定长度的二进制输出值。

作用：

1 信息安全领域。用作加密算法。

2 数据结构领域。可以用作快速查找。

哈希表：使用哈希算法实现的数据结构。哈希表是根据设定的哈希函数H(key)和处理冲突方法将一组关键字映射到一个有限的地址区间上，并以关键字在地址区间中的映射作为记录在表中的存储位置，这种表称为哈希表或者散列。

哈希函数：是一个映像，即将关键字的集合映射到某个地址集合上。只要这个地址集合的大小不超出允许范围即可。

哈希函数是一个压缩映像，一般情况下很容易产生冲突。即key1!=key2，而f(key1)=f(key2)

只能尽量减少冲突，而不能完全避免冲突。这是因为关键字的集合比较大，其元素包含了所有可能的关键字，而地址集合的元素仅为哈希表中的地址值，在构造这种特殊的查找表时，除了需要一个比较好的哈希函数外，还需要找到一个处理冲突的方法。

## 哈希函数的构造方法

### 直接定址法

哈希函数为关键字的线性函数，即f(key)=key 或者f(key)=a\*key+b

特点：有多少个关键字就有多少个地址去对应，不会产生冲突。但是很少使用。

### 数字分析法

首先分析待存的一组*关键字* ，比如是一个班级学生的*出生年月日* ，我们发现他们的出生*年* 大体相同，那么我们肯定不能用他们的*年* 来作为*存储地址* ，这样出现*冲突* 的几率很大；但是，我们发现*月日* 的具体数字差别很大，如果我们用*月日* 来作为*Hash地址* ，则会明显降低冲突几率。因此，数字分析法就是找出*关键字* 的规律，尽可能用差异数据来构造*Hash地址* ；

特点：**需要提前知道所有可能的关键字**，才能分析运用此种方法，所以不太常用。

### 平方取中法

**先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址**。这是因为：平方后中间几位和关键字中每一位都相关，故不同关键字会以较高的概率产生不同的哈希地址。

特点：比较常用

### 折叠法

将关键字分割成位数相同的几部分（最后一部分位数可以不同），然后取这几部分的叠加和（去除进位）作为散列地址。数位叠加可以有移位叠加和间界叠加两种方法。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是从一端向另一端沿分割界来回折叠，然后对齐相加。

### 随机数法

选择一个随机函数，取关键字的随机函数值作为*Hash地址*，通常用于关键字长度不同的场合。即   
f(key)=random(key)

特点：通常，关键字长度不相等时，采用此法构建*Hash函数*较为合适。

### 除留取余法

f(k)=k mod p ， p<=m

取关键字被某个不大于*Hash表*长*m*的数*p*除后所得的余数为*Hash地址*。

特点：这是最简单也是**最常用的Hash函数构造方法**。可以直接取模，也可以在平法法、折叠法之后再取模。P的选取最好是一个质数或者不包含小于20的质因数的合数。

## 处理冲突

为产生冲突的关键字寻找下一个哈希地址。

假设哈希表的地址集为0…. (n−1)，冲突是指由关键字得到的哈希地址为j(0≤j≤n−1)的位置上已存有记录，则“处理冲突”就是为该关键字的记录找到另一个“空”的哈希地址。   
在处理冲突的过程中可能得到一个地址序列Hi,i=1,2,...,k(Hi∈[0,n−1])。即在处理哈希地址的冲突时，若得到的另一个哈希地址Hi仍然发生冲突，则再求下一个地址H2，若H2仍然冲突，再求H3，依次类推，直至Hk不发生冲突为止，则Hk为记录在表中的地址。   
（需要注意此定义不太适合链地址法）

### 开放定址法

Hi=(H(key)+di)modm，i=1,2,...,k(k≤m−1)

H(key) 为哈希函数；m 为哈希表表长；   
di 为增量序列，有3种取法：

1. di=1,2,3,...,m−1，称为线性探测再散列；
2. di=12,−12,22,−22,32,...,±k2，(k≤m2)，称为二次探测再散列；
3. di=伪随机数序列，称为伪随机探测再散列；

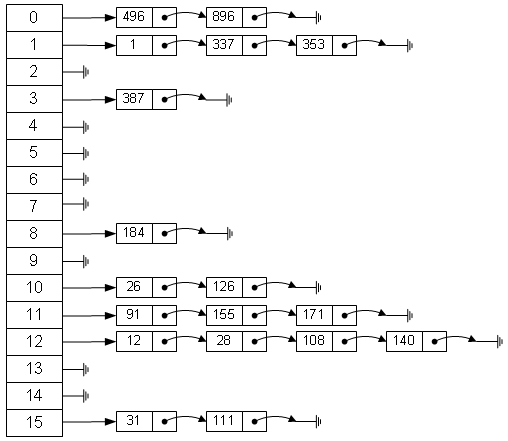
### 再哈希法

Hi=RHi(key),i=1,2,...,k

RHi 均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生，这种方法不易产生聚集，但增加了计算时间；

### 链地址法

将所有关键字为同义词的记录存储在同一线性表中。即在*Hash*出来的哈希地址中不直接存*Key*，而是存储一个*Key*的*链表*，当发生*冲突*时，将*同义的Key*加入链表。



### 公共溢出区

可以建立一个公共溢出区，用来存放有冲突的*Key*。比如设立另一个哈希表，专门用来存放出现冲突的同义词。

## 查找及分析

### 平均查找长度

虽然哈希表在关键字与记录的存储位置之间建立了直接映像，但**由于“冲突”的存在，使得哈希表的查找过程仍然是一个“给定值和关键字进行比较”的过程**。因此，仍需以**平均查找长度**作为衡量哈希表的查找效率的量度。

查找过程中需要和给定值进行比较的关键字的个数取决于下列三个因素：

* 哈希函数；
* 处理冲突的方法；
* 哈希表的装填因子；

### 装填因子

在一般情况下，我们设计的哈希函数肯定是尽量均匀的，所以可以不考虑它对平均查找长度的影响。那么，处理冲突方法相同的哈希表，其平均查找长度就依赖于哈希表的装填因子了。其定义如下：

α=表中填入的记录数/哈希表的长度 ； 装填因子α标志哈希表的装满程度

直观的看：

* α越小，发生冲突的可能性就越小；
* α越大，代表着表中已填入的元素越多，再填入元素时发生冲突的可能性就越大。那么在查找时，给定值需要比较的关键字的个数就越多；

哈希表的平均查找长度是α的函数，而不是n的函数。因此，不管n多大，我们总是可以选择一个合适的装填因子以便将平均查找长度限定在一个范围内。（Java中HashMap的默认装填因子是0.75）

也正是因为哈希冲突的存在，查找时间达不到O(1)。

尾部遍历。如果条件竞争发生了，那就会发生死循环。

# Hashcode

1 hashcode方法返回对象的哈希值（散列码）。对象的散列码是为了更好的支持基于哈希机制的Java集合类，例如 Hashtable, HashMap, HashSet 等。 主要用于查找的便捷性。

2在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改。

3从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。

4如果根据 equals(Object) 方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。

5以下情况不 是必需的：如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么在两个对象中的任一对象上调用 hashCode 方法必定会生成不同的整数结果。但是，程序员应该知道，为不相等的对象生成不同整数结果可以提高哈希表的性能。

6实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧。）

7当equals方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

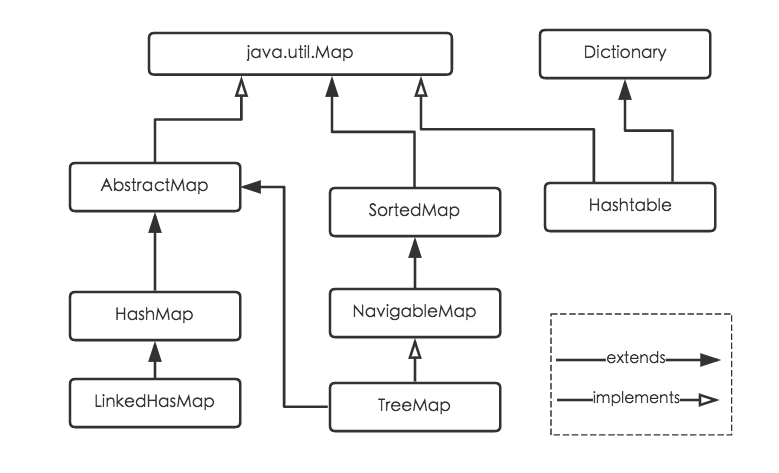
Hash code主要是用于查找使用的，equals方法主要是用于比较两个对象的。

1. 1.hashcode是用来查找的，如果你学过数据结构就应该知道，在查找和排序这一章有
2. 例如内存中有这样的位置
3. 0  1  2  3  4  5  6  7
4. 而我有个类，这个类有个字段叫ID,我要把这个类存放在以上8个位置之一，如果不用hashcode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。
5. 但如果用hashcode那就会使效率提高很多。
6. 我们这个类中有个字段叫ID,那么我们就定义我们的hashcode为ID％8，然后把我们的类存放在取得得余数那个位置。比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。这样，以后在查找该类时就可以通过ID除 8求余数直接找到存放的位置了。
8. 2.但是如果两个类有相同的hashcode怎么办那（我们假设上面的类的ID不是唯一的），例如9除以8和17除以8的余数都是1，那么这是不是合法的，回答是：可以这样。那么如何判断呢？在这个时候就需要定义 equals了。
9. 也就是说，我们先通过 hashcode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，那么我们就需要再通过 equals 来在这个桶里找到我们要的类。
10. 那么。重写了equals()，为什么还要重写hashCode()呢？
11. 想想，你要在一个桶里找东西，你必须先要找到这个桶啊，你不通过重写hashcode()来找到桶，光重写equals()有什么用啊

## 为什么要使用hashcode？

Java中的集合类有list和set。前者集合中的元素是有序的，可以重复，后者中元素是无序的，但是不能重复。怎么去判断元素是否重复？通过equal()方法，但是如果每添加一个元素就去检查一次，那么元素很多的时候，效率就会大大降低。Java中采用了哈希表的原理，哈希算法也就是散列算法，将数据依据特定的算法直接指定到一个地址上，hashcode方法返回的也就是对象存储位置的映像。当集合要添加新的元素时，会先调用该元素的hashcode方法，就能定位到它应该存储的位置。如果该位置上没有元素，就可以直接存储在该位置上，不需要进行比较，如果该位置上已经有了元素，就需要调用equal方法与新的元素进行比较。这样使用equal方法的频率就大大降低了。

# 介绍



(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。

# Jdk7 Hashmap

<http://www.importnew.com/20321.html>

1 hashmap是基于数组来实现哈希表的，数组就好比内存存储空间，数组的index就好比内存的地址。

2 hashmap中的每个记录就是一个Entry<K,V>对象，数组中存储的就是这些对象。有一个内部类Entry。

3 hashmap的哈希函数=计算出的hashcode+计算出数组的index

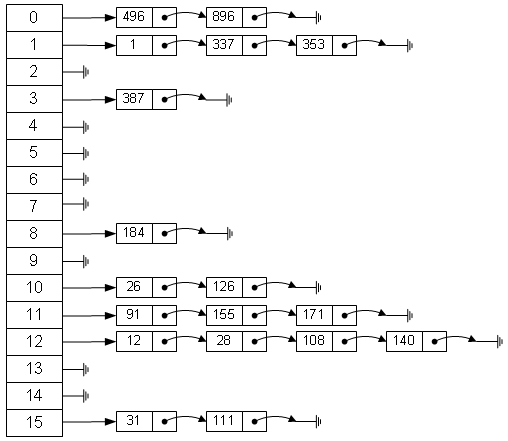
4 hashmap解决冲突的方法：使用的是链地址法，每个Entry对象都有一个引用next来指向链表中的下一个Entry对象。

5 hashmap装填因子：默认0.75

6 table索引在逻辑上叫做 桶。它存储了链表中的第一个元素。

7 如果两个key有相同的hash值，就会把他们放在同一个桶里面。

8 value对象的equals和hashcode方法并没有使用到。



哈希表是由数组和链表组成。数组中每一个元素都是链表的头节点。

Hashmap底层维护一个数组，数组中存放的是Entry<k,V>对象

transient Entry<K,V>[] table;

我们向hashmap中存储的键值实际上就是存储在该数组中。

Map中的key value则以Entry的形式存放在数组中

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

int hash;

这个Entry应该放在数组的哪一个位置（这个位置通常称为位桶或者hash桶，即hash值相同的Entry会放在一个位置上，用链表相连），是通过key的hashcode来计算的。

final int hash(Object k) {

int h = 0;

h ^= k.hashCode();

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

通过hash计算出来的值，会使用indexFor方法找到对应的table下标

static int indexFor(int h, int length) {

return h & (length-1);

}

这个方法相当于对table.leng取模

当两个key通过hashcode计算相同时，则发生了hash冲突，hashmap解决冲突的方法是用链表。当发生hash冲突时，则将存放在数组中的Entry设置为新值的next（注意，比如A和B经过hash后都映射到下标i中，之前已经有了A，当map.put(B)时，将B放到下标i中，A为B的next，所以新值放在数组中，旧值存放在新值的链表上。

当hash冲突比较多时，会退化成链表。

 public V put(K key, V value) {

  if (key == null)

   return putForNullKey(value);

  int hash = hash(key.hashCode());

  int i = indexFor(hash, table.length);

  for (Entry<k , V> e = table[i]; e != null; e = e.next) {

   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++;

  addEntry(hash, key, value, i);

  return null;

 }

总结：当向hashmap中添加一对键值时，它会首先计算出key的hashcode值，然后通过indexFor方法计算出对应table数组中的下标，这里的indexFor方法实际上就是哈希值对数组长度的取模。该位置，就是此对象准备往数组中存放的位置。

如果该位置没有对象存在，就将此对象直接放入数组中。如果该位置已经有对象存在，则顺着此存在的对象的链开始寻找，如果此链上有对象，再使用equals方法进行比较，如果比较的结果都是false，则将该对象放到数组中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。如果返回的是true，说明是相同的key，则用当前Entry的value替换之前的value。

当key为null时，都存放到table[0]中。因为null的hashcode总是0.

private V putForNullKey(V value) {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(0, null, value, 0);

return null;

}

当size大于threshold时，就会发生扩容。Threshold等于capacity\*load factor

void addEntry(int hash, K key, V value, int bucketIndex) {

if ((size >= threshold) && (null != table[bucketIndex])) {

resize(2 \* table.length);

hash = (null != key) ? hash(key) : 0;

bucketIndex = indexFor(hash, table.length);

}

createEntry(hash, key, value, bucketIndex);

}

只有size>=threshold并且table中的每个槽都有entry时，才会resize。每次resize都会扩大一倍容量。

取出数据时

public V get(Object key) {

  if (key == null)

   return getForNullKey();

  int hash = hash(key.hashCode());

  for (Entry<k , V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

   Object k;

   if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

    return e.value;

  }

  return null;

 }

当从key获取value时，首先对key进行null检查，如果key为null，则返回table[0]位置上的元素，否则，会调用hashcode方法，计算hash值，然后根据indexFor方法计算要获取的Entry对象在table数组中精确的位置。在获得了table数组的索引之后，就会迭代链表，并调用equals方法检查key的相等性，如果返回true，则返回Entry对象的value，否则返回null。

添加 查找 删除都是先计算hash然后计算index,最后再遍历查找.

Clear方法就是在遍历数组时,把每个位置置为null,同时修改元素的个数为0，并不会重置capacity

containsKey()是先计算hash，然后使用hash和数组长度取模，遍历数组查找是否包含相同key的值。

Containsvalue方法比较粗暴，直接遍历所有元素直到找到value，并不高效。

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。

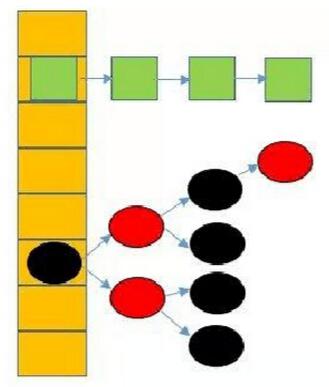
接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会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。

# Jdk8 Hashmap

<http://www.importnew.com/20386.html>

如果成百上千个节点在hash时发生了冲突，都存储在一个链表中，那么如果要查找其中一个节点，就需要O(N)的查找时间。在jdk8中，在最坏情况下，链表查找的时间复杂度为O(N),而红黑树是O(logN)，提高了hashmap效率。

Jsk7中hashmap采用的是位桶+链表的方式，非线程安全。当某个位桶的链表的长度达到某个阈值时，就会转变为红黑树。



在jdk8中，当一个hash值的节点数不小于8时，将不再以单链表形式存储，会被调整为一棵红黑树。

Jdk8中的Entry对象变为Node，是因为和红黑树的实现TreeNode相关联。

transient Node<K,V>[] table;

当冲突节点数不小于8时，会转换成红黑树。

static final int TREEIFY\_THRESHOLD = 8;

put方法

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab;

Node<K,V> p;

int n, i;

//如果当前map中无数据，执行resize方法。并且返回n

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

//如果要插入的键值对要存放的这个位置刚好没有元素，那么把他封装成Node对象，放在这个位置上就完事了

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

//否则的话，说明这上面有元素

else {

Node<K,V> e; K k;

//如果这个元素的key与要插入的一样，那么就替换一下，也完事。

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

//1.如果当前节点是TreeNode类型的数据，执行putTreeVal方法

else if (p instanceof TreeNode)

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else {

//还是遍历这条链子上的数据，跟jdk7没什么区别

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

//2.完成了操作后多做了一件事情，判断，并且可能执行treeifyBin方法

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e;

}

}

if (e != null) { // existing mapping for key

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null) //true || --

e.value = value;

//3.

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

//判断阈值，决定是否扩容

if (++size > threshold)

resize();

//4.

afterNodeInsertion(evict);

return null;

}

treeifyBin()就是将链表转换成红黑树。

之前的indefFor()方法消失 了，直接用(tab.length-1)&hash，所以看到这个，代表的就是数组的下角标。

final Node<K,V> removeNode(int hash, Object key, Object value,boolean matchValue, boolean movable) {

Node<K,V>[] tab; Node<K,V> p; int n, index;

if ((tab = table) != null && (n = tab.length) > 0 && (p = tab[index = (n - 1) & hash]) != null) {

Node<K,V> node = null, e; K k; V v;

//根据key的hash值找到对应位置的练表

//如果练表的第一个元素的key和要删除元素的key不一致，说明没有找到，继续往后找

if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))

node = p;

//顺着练表往后找

else if ((e = p.next) != null) {

//如果该练表已经转换成了红黑树，那么用红黑树的方式查找

if (p instanceof TreeNode)

node = ((TreeNode<K,V>)p).getTreeNode(hash, key);

//如果还是练表，那么继续用遍历的方式往后找

else {

do {

if (e.hash == hash &&

((k = e.key) == key ||

(key != null && key.equals(k)))) {

node = e;

break;

}

p = e;

} while ((e = e.next) != null);

}

}

//找到该节点位置后

if (node != null && (!matchValue || (v = node.value) == value ||

(value != null && value.equals(v)))) {

//找到该节点位置后，如果该节点已经是一个红黑树节点，那么通过红黑树的方式删除该节点

if (node instanceof TreeNode)

((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);

//如果找到的节点是练表的第一个节点，那么让这个节点的下一个节点作为这个链表的第一个节点

else if (node == p)

tab[index] = node.next;

//如果找到的节点不是链表的第一个节点，那么此时p是node的前一个节点（观察前面在链表中遍历查找位置的代码就可以发现）

//那么直接通过p.next = node.next跨过node节点，达到删除效果

else

p.next = node.next;

++modCount;

--size;

afterNodeRemoval(node);

return node;

}

}

return null;

}

# HashMap线程安全的问题

<http://www.cnblogs.com/andy-zhou/p/5402984.html>

1 两个线程在做put操作时，当添加元素的hashcode相同时，也就是说，对同一个数组位置调用了addEntry方法，两个线程会同时得到现在的头节点，然后A线程写入新的头节点后，B线程也写入新的头节点，那么B的写入操作就会覆盖A的写入操作，造成A的写入丢失。（解决hash冲突是用链地址法，每次新增的元素都会放在链表的头部）

2 当多个线程同时操作同一个数组位置的时候，也都会先取得现在状态下该位置存储的头结点，然后各自去进行计算操作，之后再把结果写会到该数组位置去，其实写回的时候可能其他的线程已经就把这个位置给修改过了，就会覆盖其他线程的修改。

3 在resize时，多个线程同时调用resize方法，各自生成新的数组并rehash后赋给该map底层的数组table，结果最终只有最后一个线程生成的新数组被赋给table变量，其他线程均会丢失。而且会产生环形链表，从而发生死循环问题。

# ConcurrentHashMap

参考<http://www.importnew.com/22007.html>

并发编程实践中，ConcurrentHashMap是一个经常被使用的数据结构，相比于Hashtable以及Collections.synchronizedMap()，ConcurrentHashMap在线程安全的基础上提供了更好的写并发能力，但同时降低了对读一致性的要求. ConcurrentHashMap的设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响.

ConcurrentHashMap采用了分段锁的设计，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size(), containsValue()）需要使用特殊的实现，另外一些方法（如clear()）甚至放弃了对一致性的要求（ConcurrentHashMap是弱一致性的.

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

static final class HashEntry<K,V> {

        final int hash;

        final K key;

        volatile V value;

        volatile HashEntry<K,V> next;

# 问题

http://blog.csdn.net/aichuanwendang/article/details/53317351

Hasnmap中的一些参数值

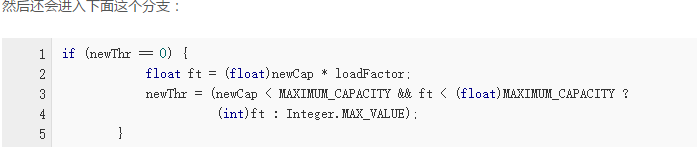
static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;//默认的桶数组大小

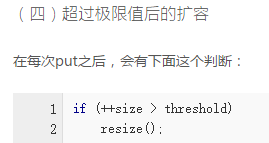
static final int MAXIMUM\_CAPACITY = 1 << 30;//极限值（超过这个值就将threshold修改为Integer.MAX\_VALUE（此时桶大小已经是2的31次方了），表明不进行扩容了）

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;//负载因子

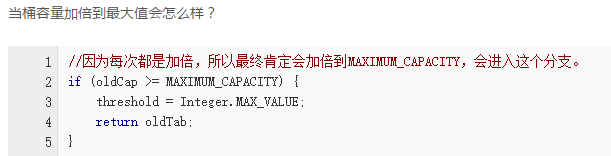


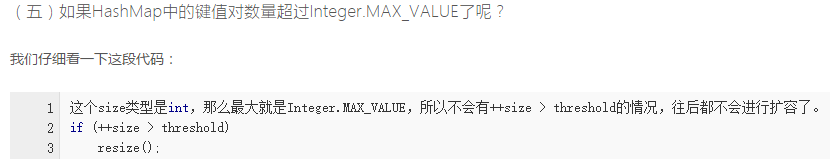












http://blog.csdn.net/fyxxq/article/details/42066843

为什么要有hashmap？ 使用了hash表的数据结构，可以快速存取。

为什么hash算法可以快速存取？

哈希函数建立了从“键—值对”到哈希表地址集合的一个映射，有了哈希函数，我们可以根据键来确定“键值对”在哈希表中的位置的地址。使用这种方法由于不必进行多次键的比较，所以搜索速度非常快。

Hash表和hashmap关系？ Hashmap是java中的一种数据类型，通过代码实现了hash表这种数据结构，并在此结构上定义了一系列操作。

**HashMap的工作原理**，**HashMap的get()方法的工作原理？**

HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用 hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。

**当两个对象的hashcode相同会发生什么？**

因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。

**如果两个键的hashcode相同，你如何获取值对象？**

当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，如果有两个值对象储存在同一个bucket，因为HashMap在链表中存储的是键值对，将会遍历链表，调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。

许多情况下，面试者会在这个环节中出错，因为他们混淆了hashCode()和equals()方法。因为在此之前hashCode()屡屡出现， 而equals()方法仅仅在获取值对象的时候才出现。一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和 hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用 String，Interger这样的wrapper类作为键是非常好的选择。

如果hashmap大小超过了负载因子定义的容量，怎么办？

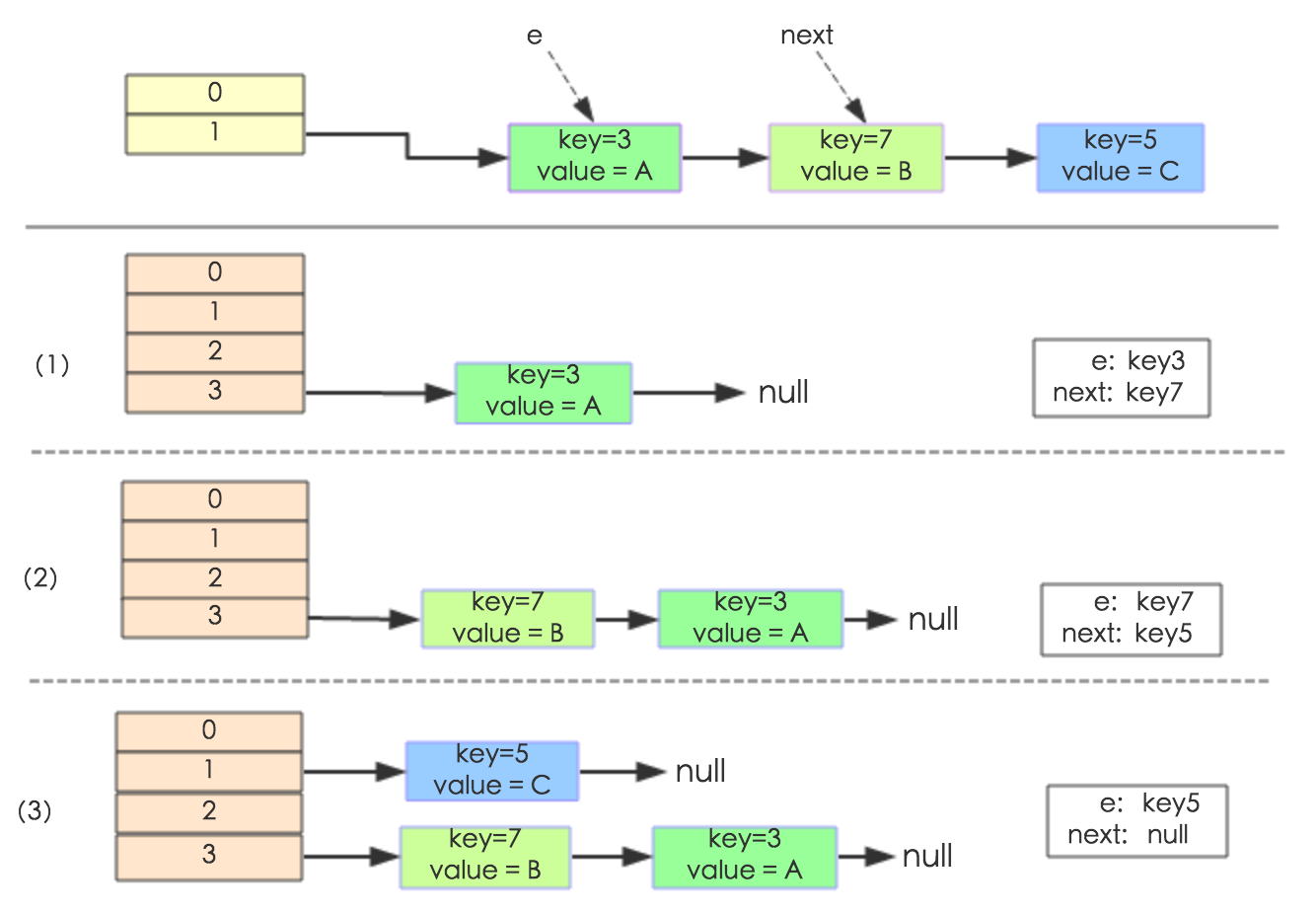
默认的负载因子大小是0.75，也就是说当一个map中填满了75%的元素时，将会进行扩容，创建原来hashmap大小的两倍的bucket数组，重新调整map大小，并将原来的对象放入新的bucket数组中。叫做rehashing。因为他调用了hash方法找到新的bucket位置。

重新调整大小的时，会出现什么问题？

当重新调整大小时，在多线程情况下，会存在条件竞争，因为如果两个线程都发现了hashmap需要重新调整大小，他们会同时试着调整大小，在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，hashmap并不会将元素放在链表的尾部，而是放在头部，这是为了避免这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。

<http://blog.csdn.net/xuefeng0707/article/details/40797085>

<http://blog.csdn.net/qq_27093465/article/details/52270519>



**为什么String, Interger这样的wrapper类适合作为键？**

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final 的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算 hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象 的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的 hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。

**我们可以使用自定义的对象作为键吗？**

这是前一个问题的延伸。当然你可能使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之 后将不会再改变了。如果这个自定义对象时不可变的，那么它已经满足了作为键的条件，因为当它创建之后就已经不能改变了。

也就是说，需要重写hashcode和equals方法。如果没有重写，会默认调用object类的这两个方法，hashcode方法是根据对象的地址进行计算的，那么每次得到的hashcode不一样，即存入时候的key的hashcode和取出时key的hashcode是不一样，这样自然就无法取到对应的值了。（最好以不可变对象作为键）（这里要注意，比如在hashcode方法中使用了id，那么要确保id这个属性是不能被改变的，可以设置为final类型的）

**我们可以使用CocurrentHashMap来代替Hashtable吗？**

这是另外一个很热门的面试题，因 为ConcurrentHashMap越来越多人用了。我们知道Hashtable是synchronized的，但是 ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替 HashTable，但是HashTable提供更强的线程安全性。看看[这篇博客](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html)查看Hashtable和ConcurrentHashMap的区别。

默认初始容量是16，加载因子是0.75。容量是哈希表中数组的长度,如文章最开始给的图其容量就为16，初始容量只是哈希表在创建时的容量。加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度。

为什么容量一定为2的幂

HashMap的初始化函数

public HashMap(int initialCapacity, float loadFactor) {

......

// Find a power of 2 >= initialCapacity

// 设置最重的容量为小于initialCapacity的2的整数次幂的最小数

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

......

}

所以可以看出capacity被设计成了一个一定为2的整数幂的数   
如果initialCapacity设置为11   
那么最终的capacity就为16 也就是说capacity会被设置为大于initialCapacity的最小的2的次幂值

HashMap中的**数据结构**是数组+单链表的组合，我们希望的是元素存放的更均匀，最理想的效果是，Entry数组中每个位置都只有一个元素，这样，查询的时候效率最高，不需要遍历单链表，也不需要通过equals去比较K，而且空间利用率最大。那么怎么才能将分布最大的均匀化呢？那就是取余运算%，哈希值 % 容量 = bucketIndex   
SUN的大师们是否也是如此做的呢？我们阅读一下这段源码：

static int indexFor(int h, int length) {

return h & (length-1);

}

上面这段代码就实现了取余操作   
**当容量一定是2^n时，h & (length - 1) == h % length。**

**另，2的次幂保证了数组的长度是偶数， length-1是奇数，对应的二进制的最低位是1，这样在与h做&操作时，就可以得到0或者1，这样就可以在数组中均匀分配，但是length-1的长度是偶数，那么二进制的最低位是0，做&操作时，只会得到0，那么以1结尾的就永远不能存放元素了，浪费空间。**