一、 复习一下基础知识

1. 截断低位与抹除高位 写道

如果一个数（二进制形式 n 位）对 2k 整除和取模：   
● （1）整除是截断低位（k），保留高位（n-k）；   
● （2）取模运算是抹除最高比特位（要求 k = n-1）；   
不妨以 10（1010） 和 8（1000） 为例：   
● （1）整除：10/8 == 1   
● （2）取模：10%8 == 010 == 2

2. 移位运算 写道

在构建哈希表散列值时，常用移位、异或等操作。   
1）左移运算符   
左移运算符用“<<”表示，是将运算符左边的对象，向左移动运算符右边指定的位数，并且在低位补零。其实，向左移n 位，就 相当于乘上2 的n 次方。   
例如：16 << 2 // 结果 64   
2）无符号右移运算符   
右移运算符用符号“>>>”表示，是将运算符左边的对象向右移动运算符右边指定的位数，并且在高位补0，其实右移n 位，就相当于除上2 的n 次方。   
例如：16 >>>2 // 结果 4   
3）右移运算符   
带符号的右移运算符用符号“>>”表示，是将运算符左边的运算对象，向右移动运算符右边指定的位数。如果是正数，在高位补零，如果是负数，则在高位补1。   
例如：16 << 2 // 结果 4   
-16 << 2 // 结果 -4

  二、 哈希表

1、Hash定义 写道

哈希表通过记录关键字为自变量的函数（称为哈希函数）得到访记录的存储地址查找，所以在哈希表中进行查找操作时，须用同一哈希函数计算得到待查询记录的存储地址，然后到相应的存储单元去获得相关信息再判定查找是否成功。   
根据设计哈希函数H(key)和处理冲突的方法，将一组关键字映射到一个有限连限的地址集（区间）上，并以关键字在地址集中的“像”为记录在表中的存储位置，这种表称为哈希表，这一映射过程为哈希造表或散列，所得的存储位置称为哈希地址或散列地址。   
冲突定义：对于某个哈希函数H（K）和两个关键字K1和K2，如果K1<>K2，而H(K1)=H(K2)，则称为冲突。具有相同哈希函数值的关键字对该哈希函数来说称为同义词。   
一般情况，冲突只能尽可能减少而不能完全避免，因为哈希函数是从关键字集合到地址集合映像。通常关键集合比较大，它的元素包含所有可能的关键字，而地址集合的元素仅为哈希表中的地址值。   
对于哈希表，主要考虑两个问题：一是如何构造哈希函数，二是如何解决冲突。

2、Hash构造方法 写道

Hash构造即为关键字与哈希地址映射关系建立。   
构造哈希函数的原则是：①函数本身便于计算；②计算出来的地址分布均匀，即对任一关键字k，f(k) 对应不同地址的概率相等，目的是尽可能减少冲突。   
哈希函数的构造方法很多，常用的有直接定址法、数字分析法、平方取中法、折叠法、除留余数法、随机数法等。

2.1 数字分析法 写道

如果事先知道关键字集合，并且每个关键字的位数比哈希表的地址码位数多时，可以从关键字中选出分布较均匀的若干位，构成哈希地址。例如，有80个记录，关键字为8位十进制整数d1d2d3…d7d8，如哈希表长取100，则哈希表的地址空间为：00~99。假设经过分析，各关键字中 d4和d7的取值分布较均匀，则哈希函数为：h(key)=h(d1d2d3…d7d8)=d4d7。例如，h(81346532)=43，h(81301367)=06。相反，假设经过分析，各关键字中 d1和d8的取值分布极不均匀， d1 都等于5，d8 都等于2，此时，如果哈希函数为：h(key)=h(d1d2d3…d7d8)=d1d8，则所有关键字的地址码都是52，显然不可取。

2.2 平方取中法 写道

当无法确定关键字中哪几位分布较均匀时，可以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。   
例：我们把英文字母在字母表中的位置序号作为该英文字母的内部编码。例如K的内部编码为11，E的内部编码为05，Y的内部编码为25，A的内部编码为01, B的内部编码为02。由此组成关键字“KEYA”的内部代码为11052501，同理我们可以得到关键字“KYAB”、“AKEY”、“BKEY”的内部编码。之后对关键字进行平方运算后，取出第7到第9位作为该关键字哈希地址，如图8.23所示。

|  |  |  |  |
| --- | --- | --- | --- |
| 关键字 | 内部编码 | 内部编码的平方值 | H(k)关键字的哈希地址 |
| KEYA | 11050201 | 122157778355001 | 778 |
| KYAB | 11250102 | 126564795010404 | 795 |
| AKEY | 01110525 | 001233265775625 | 265 |
| BKEY | 02110525 | 004454315775625 | 315 |

2.3 除留余数法（Jdk HashMap实现） 写道

假设哈希表长为m，p为小于等于m的最大素数，则哈希函数为   
h（k）=k % p ，其中%为模p取余运算。   
例如，已知待散列元素为（18，75，60，43，54，90，46），表长m=10，p=7，则有   
h(18)=18 % 7=4 h(75)=75 % 7=5 h(60)=60 % 7=4   
h(43)=43 % 7=1 h(54)=54 % 7=5 h(90)=90 % 7=6   
h(46)=46 % 7=4   
此时冲突较多。为减少冲突，可取较大的m值和p值，如m=p=13，结果如下：   
h(18)=18 % 13=5 h(75)=75 % 13=10 h(60)=60 % 13=8   
h(43)=43 % 13=4 h(54)=54 % 13=2 h(90)=90 % 13=12   
h(46)=46 % 13=7   
此时没有冲突，如图8.25所示。

0 1 2 3 4 5 6 7 8 9 10 11 12

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 54 |  | 43 | 18 |  | 46 | 60 |  | 75 |  | 90 |

/\*\*

\* Retrieve object hash code and applies a supplemental hash function to the

\* result hash, which defends against poor quality hash functions. This is

\* critical because HashMap uses power-of-two length hash tables, that

\* otherwise encounter collisions for hashCodes that do not differ

\* in lower bits. Note: Null keys always map to hash 0, thus index 0.

\* 获取散列值

\*/

final int hash(Object k) {

int h = hashSeed;

if (0 != h && k instanceof String) {

return sun.misc.Hashing.stringHash32((String) k);

}

h ^= k.hashCode();

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

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

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

}

/\*\*

\* Returns index for hash code h.

\* 采用除留余数法，构建哈希表

\*/

static int indexFor(int h, int length) {

//◇◇ert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";

return h & (length-1);

}

2.4 与运算法（Jdk ThreadLocalMap实现） 写道

首先定义2的整次幂哈希表长度M，取出小于等于M的最大素数N（一般为N=M-1）   
其次通过算法构建一个散列值V（Jdk定义为：0x61c88647）   
最后执行与操作：   
index = V & N 或   
index = V & (M-1)   
为什么使用“与”操作，目的是为了将散列值高位全部归零，只保留低位，用于做哈希表地址（即数组下标）。如以哈希表长度16为例，16-1=15，二进制为表示为：00000000 00000000 00001111，与某散列值做“与”操作如下，结果就是截取了最低的四位值。

00000000 00000000 00001111

& 10011100 00001101 00100101 (散列值)

00000000 00000000 00000101 //高全部归零，只留最低4位

/\*\*

\* Construct a new map initially containing (firstKey, firstValue).

\* ThreadLocalMaps are constructed lazily, so we only create

\* one when we have at least one entry to put in it.

\* 采用“与”运算构建哈希表

\*/

ThreadLocalMap(ThreadLocal firstKey, Object firstValue) {

table = new Entry[INITIAL\_CAPACITY];

// firstKey.threadLocalHashCode哈希散列值

// INITIAL\_CAPACITY哈希表长度

int i = firstKey.threadLocalHashCode & (INITIAL\_CAPACITY - 1);

table[i] = new Entry(firstKey, firstValue);

size = 1;

setThreshold(INITIAL\_CAPACITY);

}

2.5 分段叠加法 写道

这种方法是按哈希表地址位数将关键字分成位数相等的几部分（最后一部分可以较短），然后将这几部分相加，舍弃最高进位后的结果就是该关键字的哈希地址。具体方法有折叠法与移位法。移位法是将分割后的每部分低位对齐相加，折叠法是从一端向另一端沿分割界来回折叠（奇数段为正序，偶数段为倒序），然后将各段相加。例如：key=12360324711202065,哈希表长度为1000，则应把关键字分成3位一段，在此舍去最低的两位65，分别进行移位叠加和折叠叠加，求得哈希地址为105和907。

2.6 伪随机数法 写道

采用一个伪随机函数做哈希函数，即h(key)=random(key)。   
在实际应用中，应根据具体情况，灵活采用不同的方法，并用实际数据测试它的性能，以便做出正确判定。通常应考虑以下五个因素 ：   
1） 计算哈希函数所需时间 （简单）。   
2） 关键字的长度。   
3） 哈希表大小。   
4） 关键字分布情况。   
5） 记录查找频率

三、 哈希表 **冲突处理方法**

3.1 开放定址法 写道

这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。这种方法有一个通用的再散列函数形式：   
Hi=（H（key）+di）% m i=1，2，…，n   
其中H（key）为哈希函数，m 为表长，di称为增量序列。增量序列的取值方式不同，相应的再散列方式也不同。主要有以下三种：   
线性探测再散列   
dii=1，2，3，…，m-1   
这种方法的特点是：冲突发生时，顺序查看表中下一单元，直到找出一个空单元或查遍全表。   
二次探测再散列   
di=12，-12，22，-22，…，k2，-k2 ( k<=m/2 )   
这种方法的特点是：冲突发生时，在表的左右进行跳跃式探测，比较灵活。   
伪随机探测再散列   
di=伪随机数序列。   
具体实现时，应建立一个伪随机数发生器，（如i=(i+p) % m），并给定一个随机数做起点。   
例如，已知哈希表长度m=11，哈希函数为：H（key）= key % 11，则H（47）=3，H（26）=4，H（60）=5，假设下一个关键字为69，则H（69）=3，与47冲突。如果用线性探测再散列处理冲突，下一个哈希地址为H1=（3 + 1）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 + 2）% 11 = 5，还是冲突，继续找下一个哈希地址为H3=（3 + 3）% 11 = 6，此时不再冲突，将69填入5号单元，参图8.26 (a)。如果用二次探测再散列处理冲突，下一个哈希地址为H1=（3 + 12）% 11 = 4，仍然冲突，再找下一个哈希地址为H2=（3 - 12）% 11 = 2，此时不再冲突，将69填入2号单元，参图8.26 (b)。如果用伪随机探测再散列处理冲突，且伪随机数序列为：2，5，9，……..，则下一个哈希地址为H1=（3 + 2）% 11 = 5，仍然冲突，再找下一个哈希地址为H2=（3 + 5）% 11 = 8，此时不再冲突，将69填入8号单元

/\*\*

\* Set the value◇◇ociated with key.

\*

\* @param key the thread local object

\* @param value the value to be set

\*/

private void set(ThreadLocal key, Object value) {

// We don't use a fast path as with get() because it is at

// least as common to use set() to create new entries as

// it is to replace existing ones, in which case, a fast

// path would fail more often than not.

Entry[] tab = table;

int len = tab.length;

int i = key.threadLocalHashCode & (len-1);

for (Entry e = tab[i];

e != null;

e = tab[i = nextIndex(i, len)]) {

ThreadLocal k = e.get();

if (k == key) {

e.value = value;

return;

}

if (k == null) {

replaceStaleEntry(key, value, i);

return;

}

}

tab[i] = new Entry(key, value);

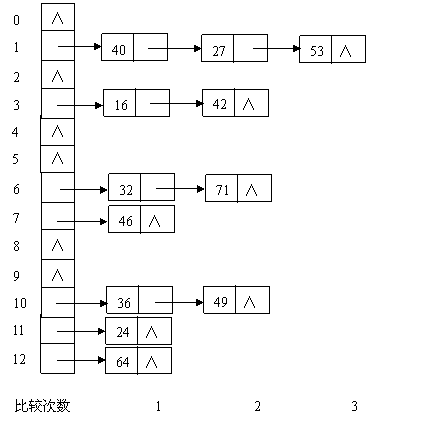
int sz = ++size;

if (!cleanSomeSlots(i, sz) && sz >= threshold)

rehash();

}

3.2 链地址法（Jdk HashMap实现） 写道

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。   
例如，已知一组关键字（32，40，36，53，16，46，71，27，42，24，49，64），哈希表长度为13，哈希函数为：H（key）= key % 13，则用链地址法处理冲突的结果如图8.27所示：   
  
 

IMG_257

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

final K key;

V value;

Entry<K,V> next;

int hash;

/\*\*

\* Creates new entry.

\*/

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

/\*\*

\* Adds a new entry with the specified key, value and hash code to

\* the specified bucket. It is the responsibility of this

\* method to resize the table if appropriate.

\*

\* Subclass overrides this to alter the behavior of put method.

\*/

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);

}

}

3.3 再哈希法 写道

这种方法是同时构造多个不同的哈希函数：   
Hi=RH1（key） i=1，2，…，k   
当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间

3.4 公共溢出区 写道

这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表