链滴

HashMap 详解 (二)

作者：someone10652
原文链接：https://ld246.com/article/1513685725644
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

初始容量和加载因子

首先我们来看看HashMap的构造方法(jdk1.7版本)：

```
/**
 * 默认的初始化的容量，必须是2的幂次数
 */
 static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
 /**
 * 默认的加载因子
 */
 static final float DEFAULT_LOAD_FACTOR = 0.75f;
 /**
 * 下一个需要重新分配的尺寸值。等于容量乘以加载因子。
 * 也就是说，一旦容量到了这个数值，将调用resize (capacity * load factor)方法重新分配容器的尺
。
 */
 int threshold;
 /**
 * 无参构造方法，使用默认的初始容量和加载因子
 */
public HashMap() {
    this(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR);
}
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException('Illegal initial capacity: ' + initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException('Illegal load factor: ' + loadFactor);

    this.loadFactor = loadFactor;
    threshold = initialCapacity;
    init();
}
```

从上面的代码我们可以发现在创建HashMap时用到了两个非常重要的参数：initialCapacity(**初始容量**和loadFactor(**加载因子**)，这两个参数直接影响HashMap的性能。

初始容量 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度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，通过调用 resize (int newCapacity)法扩容。

```
/**
 * 在put方法中会调用addEntry方法。
 * 该方法在创建Entry对象前首先判断需不需要扩容
 */
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);
}
/**
 * HashMap扩容
 */
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable, initHashSeedAsNeeded(newCapacity));
    table = newTable;
    threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
}
```

默认的初始容量是16，默认加载因子是 0.75，因此默认的容量阈值为：16 * 0.75 = 12。

加载因子

系统默认加载因子为0.75，这是在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折中，一般情况我们是无需修改的。

加载因子过高虽然减少了空间开销，但同时也增加了hash冲突的可能性，这会导致链表中的元素增加于是一个 O(1)的查找算法，就变成了链表遍历，性能变成了 O(n)，这是 Hash 表的缺陷。**相对准确估算数据量，将极大的影响HashMap的性能**，因为resize是一个重新分配的过程，耗时应该是里面大的。

初始容量

HashMap一种最常见的优化方案就是在构造时指定初始容量。通过上面的分析我们能很容易理解其的原因：可以避免发生多次resize操作。

有一点值得注意，我们说初始容量必须是2的幂次，这如何保证呢？当我调用HashMap的构造方法并入了一个非2的幂次数时发生了什么？

```
private static int roundUpToPowerOf2(int number) {
    // assert number >= 0 : "number must be non-negative";
    return number >= MAXIMUM_CAPACITY
            ? MAXIMUM_CAPACITY
            : (number > 1) ? Integer.highestOneBit((number - 1) << 1) : 1;
}
```

没错，就是执行了这个方法，看方法名我们也大概能明白这个方法的作用：返回大于或等于指定数值最小的2的幂次数，最大值为MAXIMUM_CAPACITY，即1073741824，最小值为1。举个栗子，如输入的参数是22，则返回的实际初始容量为32。

那为什么初始容量必须是2的幂次数呢?

事实上，选择2的幂次数为初始容量是为了服务于key映射到index的Hash算法。

之前说过HashMap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。计算这个置的算法就是Hash算法。所以我们当然希望这个hashmap里面的元素位置尽量的分布均匀些，尽量得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对位置的元素就是我们要的，而不用再去遍历链表。

既然如此，那这个算法应该怎么去实现呢？首先我们想到的就是把hashcode对数组长度取模运算，样一来，元素的分布相对来说是比较均匀的。但是，取模运算的性能消耗还是比较大的，为了实现高的Hash算法，HashMap的发明者采用了位运算的方式。

```
/**
 * Returns index for hash code h.
 */
static int indexFor(int h, int length) {
    // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";
    return h & (length-1);
}
```

下面我们以值为"apple"的key来演示整个过程：

1. 计算"apple"的hashcode，结果为十进制的93029210，二进制为101100010111000001101011100。

2. 假定HashMap长度是默认的16，计算Length-1的结果为十进制的15，二进制的1111。

3. 把以上两个结果做与运算101100010111000001101011010 & 1111 = 1010。十进制是10，所以ndex=10。

可以发现，Hash算法最终得到的index结果，完全取决于Key的Hashcode值的最后几位。

这种位运算不但效果上等同于取模，而且还大大提高了性能。不愧是大神写出来的代码啊！至于为什初始容量必须是2的幂次数，我们不妨假设容量为11时会发生什么事，重复刚才的运算步骤：

HashCode : 101 1000 1011 1000 0011 0101 1010

Length-1 :
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;   1010
br/>

Index :
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;

nbsp;
nbsp;
nbsp;
nbsp;   1010

让我们再换一个HashCode 10110001011100000110101 1111 试试：

HashCode : 101 1000 1011 1000 0011 0101 1111

Length-1 :
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;   1010
br/>

Index :
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;
nbsp;   1010

是的，虽然HashCode的倒数第一第三位从0变成了1，但是运算的结果都是1010。也就是说，当HasMap长度为11的时候，有些index结果的出现几率会更大，而有些index结果永远不会出现（比如111）！这样显然不符合Hash算法均匀分布的原则。

反观长度16或者其他2的幂，Length-1的值是所有二进制位全为1，这种情况下，index的结果等同HashCode后几位的值。只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的。这就是什么初始容量必须是2的幂次数的原因。