

dreamcatcher-cx

why is more important than what.

[博客园](#) [首页](#) [新随笔](#) [联系](#) [订阅](#) [管理](#)

HashMap实现原理及源码分析

哈希表 (hash table) 也叫散列表，是一种非常重要的数据结构，应用场景及其丰富，许多缓存技术（比如 memcached）的核心其实就是在内存中维护一张大的哈希表，而HashMap的实现原理也常常出现在各类的面试题中，重要性可见一斑。本文会对java集合框架中的对应实现HashMap的实现原理进行讲解，然后会对JDK7的HashMap源码进行分析。

目录

- 一、[什么是哈希表](#)
- 二、[HashMap实现原理](#)
- 三、[为何HashMap的数组长度一定是2的次幂？](#)
- 四、[重写equals方法需同时重写hashCode方法](#)
- 五、[总结](#)

公告

访问量：

 AmazingCounters.com

昵称：dreamcatcher-cx

园龄：2年2个月

粉丝：453

关注：29

[+加关注](#)

<	2018年11月						>
日	一	二	三	四	五	六	
28	29	30	31	1	2	3	
4	5	6	7	8	9	10	
11	12	13	14	15	16	17	
18	19	20	21	22	23	24	

一、什么是哈希表

在讨论哈希表之前，我们先大概了解下其他数据结构在新增，查找等基础操作执行性能

数组：采用一段连续的存储单元来存储数据。对于指定下标的查找，时间复杂度为 $O(1)$ ；通过给定值进行查找，需要遍历数组，逐一比对给定关键字和数组元素，时间复杂度为 $O(n)$ ，当然，对于有序数组，则可采用二分查找，插值查找，斐波那契查找等方式，可将查找复杂度提高为 $O(\log n)$ ；对于一般的插入删除操作，涉及到数组元素的移动，其平均复杂度也为 $O(n)$

线性链表：对于链表的新增，删除等操作（在找到指定操作位置后），仅需处理结点间的引用即可，时间复杂度为 $O(1)$ ，而查找操作需要遍历链表逐一进行比对，复杂度为 $O(n)$

二叉树：对一棵相对平衡的有序二叉树，对其进行插入，查找，删除等操作，平均复杂度均为 $O(\log n)$ 。

哈希表：相比上述几种数据结构，在哈希表中进行添加，删除，查找等操作，性能十分之高，不考虑哈希冲突的情况下，仅需一次定位即可完成，时间复杂度为 $O(1)$ ，接下来我们就来看看哈希表是如何实现达到惊艳的常数阶 $O(1)$ 的。

我们知道，数据结构的物理存储结构只有两种：**顺序存储结构**和**链式存储结构**（像栈，队列，树，图等是从逻辑结构去抽象的，映射到内存中，也这两种物理组织形式），而在上面我们提到过，在数组中根据下标查找某个元素，一次定位就可以达到，哈希表利用了这种特性，**哈希表的主干就是数组**。

比如我们要新增或查找某个元素，我们通过把当前元素的关键字 通过某个函数映射到数组中的某个位置，通过数组下标一次定位就可完成操作。

存储位置 = f(关键字)

其中，这个函数f一般称为**哈希函数**，这个函数的设计好坏会直接影响到哈希表的优劣。举个例子，比如我们要在哈希表中执行插入操作：

25	26	27	28	29	30	1
2	3	4	5	6	7	8

搜索

找找看

谷歌搜索

我的标签

Oracle(3)

hashmap(1)

随笔分类(20)

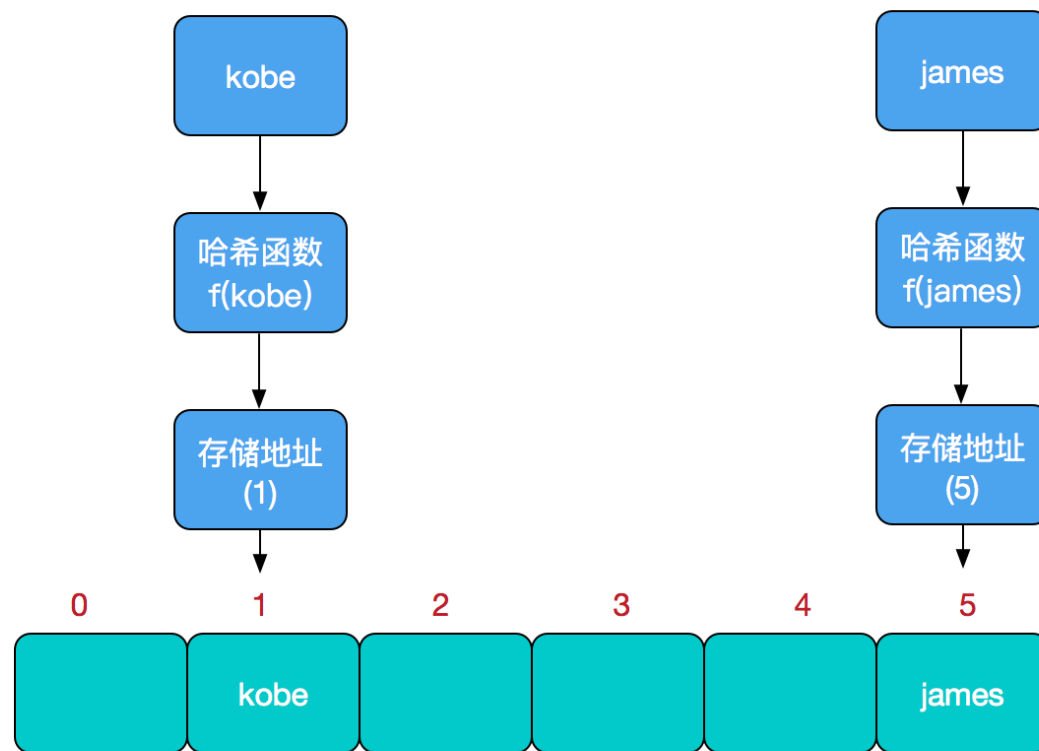
java 基础

java集合框架(1)

jvm

mysql

Oracle(4)



查找操作同理，先通过哈希函数计算出实际存储地址，然后从数组中对应地址取出即可。

哈希冲突

然而万事无完美，如果两个不同的元素，通过哈希函数得出的实际存储地址相同怎么办？也就是说，当我们对某个元素进行哈希运算，得到一个存储地址，然后要进行插入的时候，发现已经被其他元素占用了，其实这就是所谓的**哈希冲突**，也叫哈希碰撞。前面我们提到过，哈希函数的设计至关重要，好的哈希函数会尽可能地保证 **计算简单**和**散列地址分布均匀**，但是，我们需要清楚的是，数组是一块连续的固定长度的内存空间，再好的哈希函数也不能保证得到的存储地址绝对不发生冲突。那么哈希冲突如何解决呢？哈希冲突的解决方案有多种：开放定址法（发生冲突，继续寻找下一块未被占用的存储地址），再散列函数法，链地址法，而HashMap即是采用了链地址法，也就是**数组+链表**的方式，

二、HashMap实现原理

[并发编程\(8\)](#)[分布式系统](#)[数据结构\(2\)](#)[算法\(5\)](#)[随笔档案\(20\)](#)[2017年7月 \(2\)](#)[2017年6月 \(1\)](#)[2017年5月 \(2\)](#)[2017年4月 \(1\)](#)[2017年3月 \(1\)](#)[2017年2月 \(1\)](#)[2017年1月 \(1\)](#)[2016年12月 \(3\)](#)[2016年11月 \(4\)](#)

HashMap的主干是一个Entry数组。Entry是HashMap的基本组成单元，每一个Entry包含一个key-value键值对。

```
//HashMap的主干数组，可以看到就是一个Entry数组，初始值为空数组{}，主干数组的长度一定是2的次幂，至于为什么这么做，后面会有详细分析。  
transient Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE;
```

Entry是HashMap中的一个静态内部类。代码如下



```
static class Entry<K,V> implements Map.Entry<K,V> {  
    final K key;  
    V value;  
    Entry<K,V> next;//存储指向下一个Entry的引用，单链表结构  
    int hash;//对key的hashCode值进行hash运算后得到的值，存储在Entry，避免重复计算  
  
    /**  
     * Creates new entry.  
     */  
    Entry(int h, K k, V v, Entry<K,V> n) {  
        value = v;  
        next = n;  
        key = k;  
        hash = h;  
    }  
}
```



所以，HashMap的整体结构如下

2016年10月 (2)

2016年9月 (2)

积分与排名

积分 - 54299

排名 - 8561

最新评论

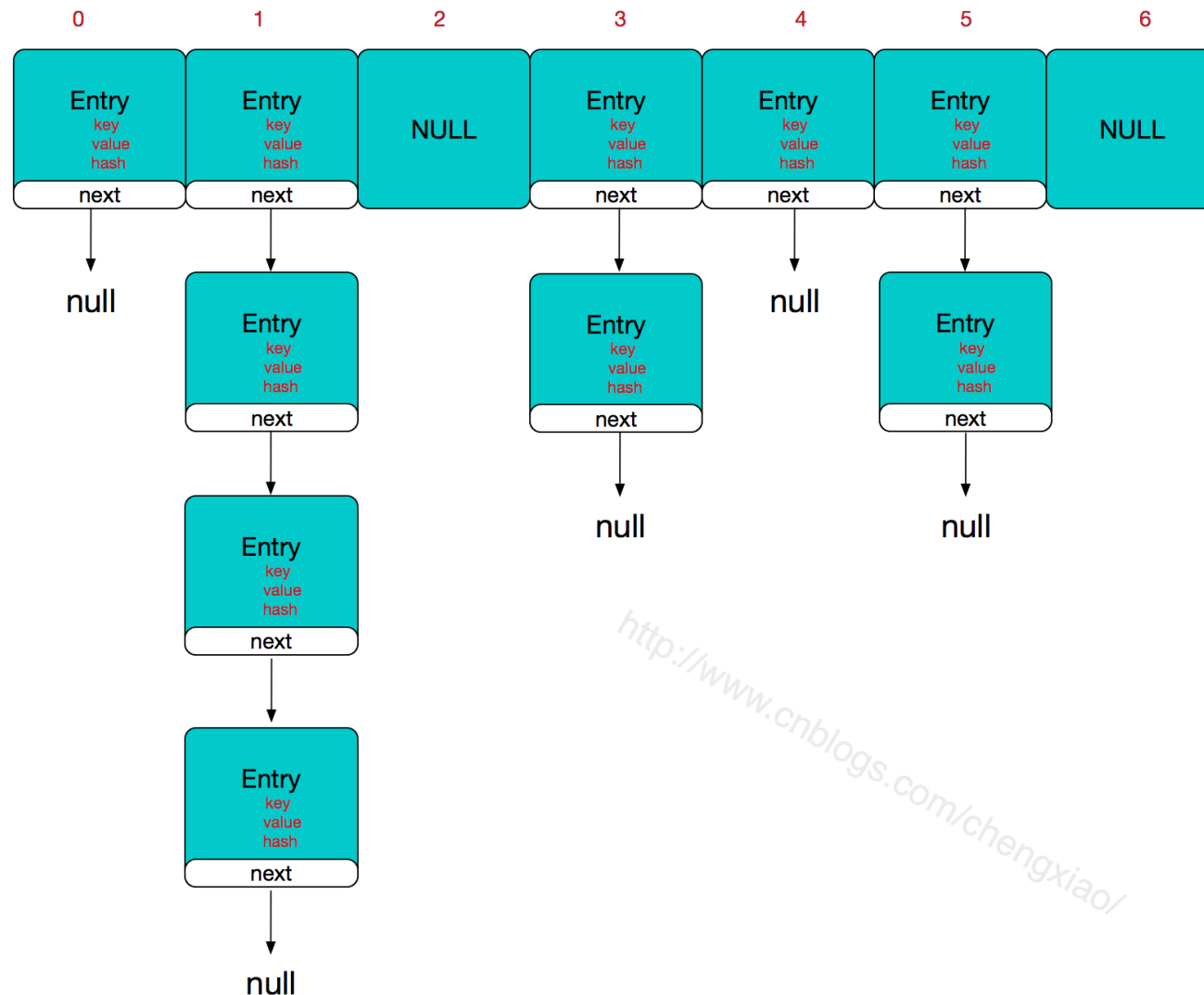
1. Re:HashMap实现原理及源码分析

这么好的文章不评论评论亏大了! www.x
ttblog.com

--业余草

2. Re:图解排序算法(三)之堆排序

@咖飞哥你这样构建的不是大顶堆，结果是能查询对，但是注意!!! 结构不是堆，你所谓的每次构建堆后只有完全二叉树的root节点是最大的，这点保证了，至于每个节点大于其子节点并没有保证，因为你构建堆的过程中.....



简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null），那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

其他几个重要字段

--BUG般存在

3. Re:图解排序算法(三)之堆排序

引用堆排序的基本思想是：将待排序序列构造一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余n-1个元素重新构造成一个堆，这样会得到n个元素的次.....

--birdou

4. Re:图解排序算法(三)之堆排序

引用第一个非叶子结点 $\text{arr.length}/2-1=5/2-1=1$ 楼主写的很好，但是我有个问题，为什么第一个叶子节点 $\text{arr.length}/2-1$? ...


--Umyng

5. Re:Java并发包基石-AQS详解

```
private void unparkSuccessor(Node node) { //获取wait状态 int ws = node.waitStatus; .....
```

--破石

阅读排行榜



```
//实际存储的key-value键值对的个数
transient int size;

//阈值, 当table == {}时, 该值为初始容量 (初始容量默认为16) ; 当table被填充了, 也就是为table分配内存空间后, threshold一般为
capacity*loadFactory。HashMap在进行扩容时需要参考threshold, 后面会详细谈到


int threshold;

//负载因子, 代表了table的填充度有多少, 默认是0.75

final float loadFactory;


//用于快速失败, 由于HashMap非线程安全, 在对HashMap进行迭代时, 如果期间其他线程的参与导致HashMap的结构发生了变化了 (比如put, remove等操作), 需要抛出异常ConcurrentModificationException

transient int modCount;
```



HashMap有4个构造器, 其他构造器如果用户没有传入initialCapacity 和loadFactor这两个参数, 会使用默认值
initialCapacity默认为16, loadFactory默认为0.75

我们看下其中一个



```
public HashMap(int initialCapacity, float loadFactory) {

    //此处对传入的初始容量进行校验, 最大不能超过MAXIMUM_CAPACITY = 1<<30 (230)
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
            initialCapacity);

    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;

    if (loadFactory <= 0 || Float.isNaN(loadFactory))
        throw new IllegalArgumentException("Illegal load factor: " +
            loadFactory);

    this.loadFactory = loadFactory;
    threshold = initialCapacity;
```

1. 图解排序算法(一)之3种简单排序(选择, 冒泡, 直接插入)(157486)

2. 图解排序算法(三)之堆排序(150164)

3. HashMap实现原理及源码分析(144736)

4. 图解排序算法(四)之归并排序(95205)

5. 图解排序算法(二)之希尔排序(78307)

评论排行榜

1. HashMap实现原理及源码分析(39)

2. 图解排序算法(三)之堆排序(30)

3. 图解排序算法(四)之归并排序(19)

4. 图解排序算法(一)之3种简单排序(选择, 冒泡, 直接插入)(15)

5. 图解排序算法(二)之希尔排序(14)

推荐排行榜

```
init();//init方法在HashMap中没有实际实现,不过在其子类如 linkedHashMap中就会有对应实现
```



从上面这段代码我们可以看出,在常规构造器中,没有为数组table分配内存空间(有一个入参为指定Map的构造器例外),而是在执行put操作的时候才真正构建table数组

OK,接下来我们来看看put操作的实现吧



```
public V put(K key, V value) {  
    //如果table数组为空数组{},进行数组填充(为table分配实际内存空间),入参为threshold,此时threshold为initialCapacity 默认是1<<4(2^4=16)  
    if (table == EMPTY_TABLE) {  
        inflateTable(threshold);  
    }  
    //如果key为null,存储位置为table[0]或table[0]的冲突链上  
    if (key == null)  
        return putForNullKey(value);  
    int hash = hash(key); //对key的hashCode进一步计算,确保散列均匀  
    int i = indexFor(hash, table.length); //获取在table中的实际位置  
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
        //如果该对应数据已存在,执行覆盖操作.用新value替换旧value,并返回旧value  
        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++; //保证并发访问时,若HashMap内部结构发生变化,快速响应失败  
    addEntry(hash, key, value, i); //新增一个entry
```

1. HashMap实现原理及源码分析(79)

2. 图解排序算法(三)之堆排序(53)

3. 图解排序算法(四)之归并排序(39)

4. ConcurrentHashMap实现原理及源码分析(18)

5. 图解排序算法(二)之希尔排序(17)

```
return null;
```

```
}
```



先来看看inflateTable这个方法



```
private void inflateTable(int toSize) {  
    int capacity = roundUpToPowerOf2(toSize); // capacity一定是2的次幂  
    threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1); // 此处为threshold赋值, 取  
    capacity * loadFactor和MAXIMUM_CAPACITY + 1的最小值, capacity一定不会超过MAXIMUM_CAPACITY, 除非loadFactor大于1  
    table = new Entry[capacity];  
    initHashSeedAsNeeded(capacity);  
}
```



inflateTable这个方法用于为主干数组table在内存中分配存储空间, 通过roundUpToPowerOf2(toSize)可以确保capacity为大于或等于toSize的最接近toSize的二次幂, 比如toSize=13, 则capacity=16; to_size=16, capacity=16; to_size=17, capacity=32.




```
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;  
}
```



roundUpToPowerOf2中的这段处理使得数组长度一定为2的次幂, Integer.highestOneBit是用来获取最左边的bit (其他bit位为0) 所代表的数值.

hash函数




//这是一个神奇的函数，用了很多的异或，移位等运算，对key的hashCode进一步进行计算以及二进制位的调整等来保证最终获取的存储位置尽量分布均匀


```
final int hash(Object k) {
    int h = hashSeed;
    if (0 != h && k instanceof String) {
        return sun.misc.Hashing.stringHash32((String) k);
    }

    h ^= k.hashCode();


    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```



以上hash函数计算出的值，通过indexFor进一步处理来获取实际的存储位置



```
/**
 * 返回数组下标
 */
static int indexFor(int h, int length) {
    return h & (length-1);
}
```



$h \& (length-1)$ 保证获取的index一定在数组范围内，举个例子，默认容量16， $length-1=15$ ， $h=18$ ，转换成二进制计算为

```

  1  0  0  1  0
&  0  1  1  1  1
-----
  0  0  0  1  0   = 2
```

最终计算出的index=2。有些版本的对于此处的计算会使用 取模运算，也能保证index一定在数组范围内，不过位运算对计算机来说，性能更高一些（HashMap中有大量位运算）

所以最终存储位置的确定流程是这样的：



再看看addEntry的实现：

```
void addEntry(int hash, K key, V value, int bucketIndex) {  
    if ((size >= threshold) && (null != table[bucketIndex])) {  
        resize(2 * table.length); //当size超过临界阈值threshold, 并且即将发生哈希冲突时进行扩容  
        hash = (null != key) ? hash(key) : 0;  
        bucketIndex = indexFor(hash, table.length);  
    }  
  
    createEntry(hash, key, value, bucketIndex);  
}
```

通过以上代码能够得知，当发生哈希冲突并且size大于阈值的时候，需要进行数组扩容，扩容时，需要新建一个长度为之前数组2倍的新的数组，然后将当前的Entry数组中的元素全部传输过去，扩容后的新数组长度为之前的2倍，所以扩容相对来说是个耗资源的操作。

三、为何HashMap的数组长度一定是2的次幂？

我们继续看上面提到的resize方法

```
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);
}
```



如果数组进行扩容，数组长度发生变化，而存储位置 $\text{index} = h \& (\text{length} - 1)$, index也可能会发生变化，需要重新计算index，我们先来看看transfer这个方法



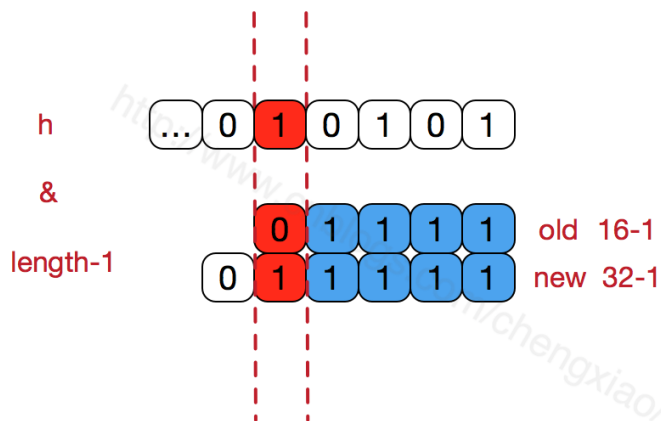
```
void transfer(Entry[] newTable, boolean rehash) {
    int newCapacity = newTable.length;
    //for循环中的代码，逐个遍历链表，重新计算索引位置，将老数组数据复制到新数组中去（数组不存储实际数据，所以仅仅是拷贝引用而已）
    for (Entry<K,V> e : table) {
        while(null != e) {
            Entry<K,V> next = e.next;
            if (rehash) {
                e.hash = null == e.key ? 0 : hash(e.key);
            }
            int i = indexFor(e.hash, newCapacity);
            //将当前entry的next链指向新的索引位置，newTable[i]有可能为空，有可能也是个entry链，如果是entry链，直接在链表头部插入。

            e.next = newTable[i];
            newTable[i] = e;
            e = next;
        }
    }
}
```

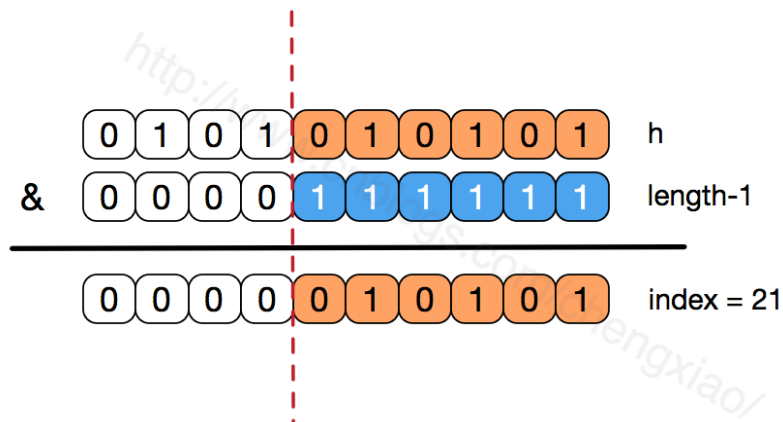


这个方法将老数组中的数据逐个链表地遍历，扔到新的扩容后的数组中，我们的数组索引位置的计算是通过 对key值的hashcode进行hash扰乱运算后，再通过和 $\text{length}-1$ 进行位运算得到最终数组索引位置。

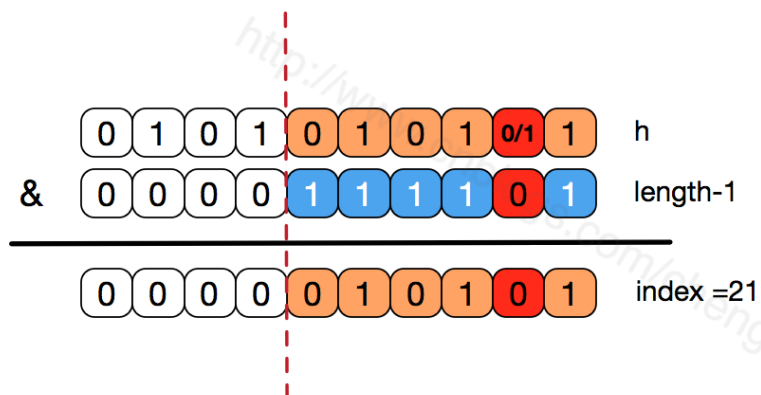
hashMap的数组长度一定保持2的次幂，比如16的二进制表示为 10000，那么 $\text{length}-1$ 就是15，二进制为01111，同理扩容后的数组长度为32，二进制表示为100000， $\text{length}-1$ 为31，二进制表示为011111。从下图可以我们也能看到这样会保证低位全为1，而扩容后只有一位差异，也就是多出了最左位的1，这样在通过 $h \& (\text{length}-1)$ 的时候，只要h对应的最左边的那一个差异位为0，就能保证得到的新的数组索引和老数组索引一致(大大减少了之前已经散列良好的老数组的数据位置重新调换)，个人理解。



还有，数组长度保持2的次幂， $\text{length}-1$ 的低位都为1，会使得获得的数组索引index更加均匀，比如：



我们看到，上面的&运算，高位是不会对结果产生影响的（hash函数采用各种位运算可能也是为了使得低位更加散列），我们只关注低位bit，如果低位全部为1，那么对于h低位部分来说，任何一位的变化都会对结果产生影响，也就是说，要得到index=21这个存储位置，h的低位只有这一种组合。这也是数组长度设计为必须为2的次幂的原因。



如果不是2的次幂，也就是低位不是全为1此时，要使得index=21，h的低位部分不再具有唯一性了，哈希冲突的几率会变的更大，同时，index对应的这个bit位无论如何不会等于1了，而对应的那些数组位置也就被白白浪费了。

get方法



```
public V get(Object key) {
    //如果key为null,则直接去table[0]处去检索即可。
    if (key == null)
        return getForNullKey();
    Entry<K,V> entry = getEntry(key);
    return null == entry ? null : entry.getValue();
}
```



get方法通过key值返回对应value，如果key为null，直接去table[0]处检索。我们再看一下getEntry这个方法



```
final Entry<K,V> getEntry(Object key) {
```

```
if (size == 0) {
    return null;
}
//通过key的hashCode值计算hash值
int hash = (key == null) ? 0 : hash(key);
//indexOf (hash&length-1) 获取最终数组索引, 然后遍历链表, 通过equals方法比对找出对应记录
for (Entry<K,V> e = table[indexOf(hash, table.length)];
    e != null;
    e = e.next) {
    Object k;
    if (e.hash == hash &&
        ((k = e.key) == key || (key != null && key.equals(k))))
        return e;
}
return null;
}
```



可以看出, get方法的实现相对简单, key(hashcode)-->hash-->indexOf-->最终索引位置, 找到对应位置 table[i], 再查看是否有链表, 遍历链表, 通过key的equals方法比对查找对应的记录。要注意的是, 有人觉得上面在定位到数组位置之后然后遍历链表的时候, e.hash == hash这个判断没必要, 仅通过equals判断就可以。其实不然, 试想一下, 如果传入的key对象重写了equals方法却没有重写hashCode, 而恰巧此对象定位到这个数组位置, 如果仅仅用equals判断可能是相等的, 但其hashCode和当前对象不一致, 这种情况, 根据Object的hashCode的约定, 不能返回当前对象, 而应该返回null, 后面的例子会做出进一步解释。

四、重写equals方法需同时重写hashCode方法


关于HashMap的源码分析就介绍到这儿了, 最后我们再聊聊老生常谈的一个问题, 各种资料上都会提到, “重写equals时也要同时覆盖hashCode”, 我们举个小例子来看看, 如果重写了equals而不重写hashcode会发生什么样的问题



```
/**
 * Created by chengxiao on 2016/11/15.
 */
public class MyTest {
    private static class Person{
        int idCard;
        String name;

        public Person(int idCard, String name) {
            this.idCard = idCard;
            this.name = name;
        }
        @Override
        public boolean equals(Object o) {
            if (this == o) {
                return true;
            }
            if (o == null || getClass() != o.getClass()){
                return false;
            }
            Person person = (Person) o;
            //两个对象是否等值, 通过idCard来确定
            return this.idCard == person.idCard;
        }
    }

    public static void main(String []args){
        HashMap<Person,String> map = new HashMap<Person, String>();
        Person person = new Person(1234,"乔峰");
        //put到hashmap中去
        map.put(person,"天龙八部");
        //get取出, 从逻辑上讲应该能输出"天龙八部"
        System.out.println("结果:"+map.get(new Person(1234,"萧峰")));
    }
}
```

```
}  
}  

```

实际输出结果：

结果: null

如果我们已经对HashMap的原理有了一定了解，这个结果就不难理解了。尽管我们在进行get和put操作的时候，使用的key从逻辑上讲是等值的（通过equals比较是相等的），但由于没有重写hashCode方法，所以put操作时，key(hashcode1)-->hash-->indexFor-->最终索引位置，而通过key取出value的时候 key(hashcode1)-->hash-->indexFor-->最终索引位置，由于hashCode1不等于hashCode2，导致没有定位到一个数组位置而返回逻辑上错误的值null（也有可能碰巧定位到一个数组位置，但是也会判断其entry的hash值是否相等，上面get方法中有提到。）

所以，在重写equals的方法的时候，必须注意重写hashCode方法，同时还要保证通过equals判断相等的两个对象，调用hashCode方法要返回同样的整数值。而如果equals判断不相等的两个对象，其hashCode可以相同（只不过会发生哈希冲突，应尽量避免）。

五、总结

本文描述了HashMap的实现原理，并结合源码做了进一步的分析，也涉及到一些源码细节设计缘由，最后简单介绍了为什么重写equals的时候需要重写hashCode方法。希望本篇文章能帮助到大家，同时也欢迎讨论指正，谢谢支持！

作者：dreamcatcher-cx

出处：<<http://www.cnblogs.com/chengxiao/>>

本文版权归作者和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在页面明显位置给出原文链接。

分类： java集合框架

标签： hashmap

好文要顶

关注我

收藏该文



dreamcatcher-cx

关注 - 29

粉丝 - 453

79

0

[+加关注](#)[« 上一篇: Oracle约束\(Constraint\)详解](#)[» 下一篇: 图解排序算法\(一\)之3种简单排序\(选择, 冒泡, 直接插入\)](#)

posted @ 2016-11-16 00:27 dreamcatcher-cx 阅读(144740) 评论(39) 编辑 收藏

评论列表

#1楼 2017-04-13 21:03 BugBean

你的jdk版本是什么, 我看1.7的源码和你这有点不一样, 构建table数组是在构造方法里进行的, 不是在put方法里进行的

支持(0) 反对(0)

#2楼[楼主] 2017-04-23 15:24 dreamcatcher-cx

@ BugBean

是1.7, put方法调用了inflateTable()方法

支持(0) 反对(0)

#3楼 2017-07-17 18:34 熊二哥

很帅!

支持(0) 反对(0)

#4楼[楼主] 2017-07-17 20:31 dreamcatcher-cx

@ 熊二哥

多谢支持

支持(0) 反对(0)

#5楼[楼主] 2017-08-02 17:08 dreamcatcher-cx

@ Paul_bai

谢谢

支持(0) 反对(0)

#6楼 2017-08-02 17:11 Paul_bai

@ dreamcatcher-cx

引用

@Paul_bai

谢谢

不客气，有人才可以推荐下，谢谢！

支持(0) 反对(0)

#7楼[楼主] 2017-08-02 17:12 dreamcatcher-cx

@ Paul_bai

引用

@dreamcatcher-cx

引用

引用@Paul_bai

谢谢

不客气，有人才可以推荐下，谢谢！

好的☺

支持(0) 反对(0)

#8楼 2017-10-16 14:42 Dougest

求画图软件

支持(0) 反对(0)

#9楼[楼主] 2017-10-16 14:48 dreamcatcher-cx

@ Dougest
OmniGraffle

支持(0) 反对(0)

#10楼 2017-11-08 15:13 涛声依旧~

求分享画图软件和博客主题css文件

支持(0) 反对(0)

#11楼[楼主] 2017-12-15 14:35 dreamcatcher-cx

@ 涛声依旧~
OmniGraffle

支持(0) 反对(0)

#12楼 2018-03-13 11:48 奥特曼之父

膜拜大神

支持(0) 反对(0)

#13楼 2018-03-16 17:20 doveshelly

终于明白为何HashMap的数组长度一定是2的次幂,楼主解答了我心中一直以来的疑惑,非常感谢!

支持(0) 反对(0)

#14楼[楼主] 2018-03-23 10:56 dreamcatcher-cx

@ doveshelly
🤝 共勉

支持(0) 反对(0)

#15楼[楼主] 2018-03-23 10:58 dreamcatcher-cx

@ 奥特曼之父
感谢支持

支持(0) 反对(0)

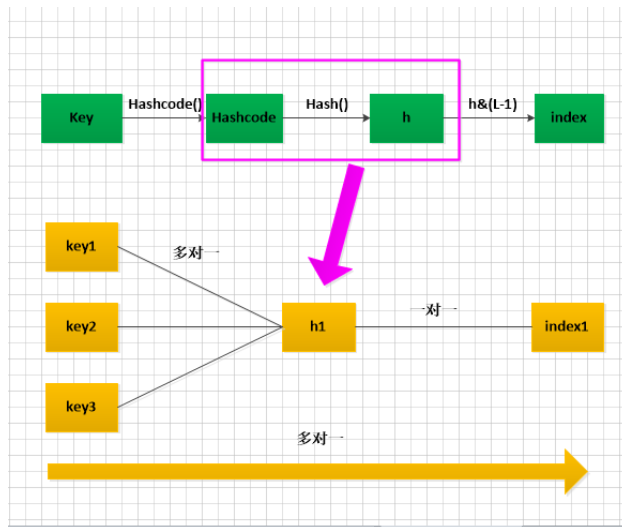
#16楼 2018-03-25 12:56 大脸喵

终于看到有个图的了，网上一对hashmap的文章，全是将使用方法，就没一个图，给作者打call

支持(0) 反对(0)

#17楼 2018-03-28 21:21 大师兄咚咚咚

博主，你好，在分析table的容量为什么是2的N次方的时，个人的想法是：**key---->hashcode----->hash----->index**，整体上这是一个多对一的映射，而**key----->hash**这个过程也是一个多对一的映射，因此**hash----->index**必须为一对一的映射，而只有在bit位都为1时才符合这种情况，无论是否扩容都需要保证bit位全部置1。



支持(0) 反对(0)

#18楼[楼主] 2018-03-28 21:25 dreamcatcher-cx

@ 大脸猫
感谢支持哈

支持(0) 反对(0)

#19楼 2018-03-29 21:41 云淡wy

如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为 $O(1)$ ，因为最新的Entry会插入链表头部，**急需要简单改变引用链即可**

这个是不是写错了,急是不是应该改成仅

支持(0) 反对(0)

#20楼[楼主] 2018-03-29 21:45 dreamcatcher-cx

@ 云淡wy
谢谢指正，改过了。

支持(0) 反对(0)

#21楼 2018-04-12 22:39 敲代码的小哥

如果不重写HashCode () 好像会默认使用Object的hashCode()方法

支持(0) 反对(0)

#22楼 2018-04-21 13:28 数据放大镜

学到了，谢谢！

支持(0) 反对(0)

#23楼 2018-05-24 18:43 wangyanxiang-

“如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为 $O(1)$ ，因为最新的Entry会插入链表头部，仅需简单改变引用链即可。”

楼主这句话我不理解，对于添加操作，应该会遍历链表查找是否存在key相同的Node节点吧？

支持(0) 反对(0)

#24楼[楼主] 2018-05-25 11:01 dreamcatcher-cx

@ wangyanxiang-

sorry，是我表述有问题。是会先遍历链表，存在即覆盖。不存在则新增。已处理，谢指正。

支持(2) 反对(0)

#25楼 2018-05-25 11:05 wangyanxiang-

@ dreamcatcher-cx

赞

支持(0) 反对(0)

#26楼 2018-07-05 16:44 albert0707

非常感谢

支持(0) 反对(0)

#27楼 2018-07-10 10:46 甜酒0917

@Override

```
public int hashCode(){  
    return this.idCard;  
}
```

支持(0) 反对(0)

#28楼 2018-07-20 17:01 清歌丶

讲解透彻，支持

支持(0) 反对(0)

#29楼 2018-07-26 19:59 YaoShuangQisBlogs

写的很不错，学到了，只是有些还是不大明白，博主，我还是有个问题需要请教你下，我看完你的整篇文章后，总感觉懵懵懂懂的样子，也不知道学习这个后，在实际项目中该如何使用其中的思想，也就是不知道该在哪里能用到，还请博主指点迷津。

支持(0) 反对(0)

#30楼 2018-07-30 15:44 longforus

谢谢博主分享,写得很好

支持(0) 反对(0)

#31楼 2018-08-03 05:18 安然菇凉

博主, 我觉得你写的那个demo并不能说明问题: 你放进去的是person, 所以你取的时候也应该是person对象啊, 但是你new了一个新的对象, 不就应该为null吗

支持(1) 反对(0)

#32楼 2018-08-11 09:57 Go-Alone

@ BugBean

可能你的jdk7小版本比较早, 最新的jdk里面是在put方法中初始化的

支持(0) 反对(0)

#33楼 2018-08-11 10:10 Go-Alone

@ 安然菇凉

hashmap中存储时, 是按照hashcode的值计算出在hashmap数组的位置, 进行存储的。

你想表达的是, 因为两个对象的内存地址不一样, 两个对象就是不一样, 所以取出来的是null。这个理解是有错误的, 两个不一样的对象是可以产生同一个hashcode值的, 产生了同一个hash值, 并且equals又返回true, 按楼主demo就不应该返回null。

楼主主要是演示【为什么重写equals方法需同时重写hashCode方法】, 具体原因可以看一下<<Effective Java>> 第9条, 讲解的非常详细, 大致是 如果不一致重写, 则会违反Object的【相互equals的对象, 必须具有相同的hash值】约定

支持(1) 反对(0)

#34楼 2018-08-25 15:43 Aliang-seu

@ wangyanxiang-

链表是无序的所以插入节点的操作复杂度为O(1), 你说的那个是查找, 当然put操作首先要进行查找

支持(0) 反对(0)

#35楼 2018-09-03 11:55 mistermoney

不错，收藏了

支持(0) 反对(0)

#36楼 2018-10-15 10:35 yexinyan

博主写得非常好，但是有一点质疑的是，博主说将capacity长度定为2的次幂的好处之一是“大大减少了之前已经散列良好的老数组的数据位置重新调换”，但是从transfer方法的代码中并没有看到这一好处。在数组扩容后，老数组中的各链表是从头往后遍历，然后插入到新数组相应位置的链表头部。不管这一链表中的hash值计算出的数组位置有没有变化，都会经过取出，重新插入的过程。并没有减少数据位置的变换。

支持(1) 反对(0)

#37楼 2018-10-16 21:02 Jinke2017

@ yexinyan

同学你好，请问你看懂扩容过程中 if(rehash){....}的判断吗？

如果为true就要重新哈希，这个boolean rehash是怎么求的？

支持(0) 反对(0)

#38楼 2018-10-16 22:47 Jinke2017

楼主，虽然时隔已久，但我还是想请教：

threshold = (int) Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1); //此处为threshold赋值，取capacity*loadFactor和MAXIMUM_CAPACITY+1的最小值，capacity一定不会超过MAXIMUM_CAPACITY，除非loadFactor大于1

从代码上看，即使loadFactor大于1, capacity也不会超过MAXIMUM_CAPACITY的吧？

支持(0) 反对(0)

#39楼 2018-11-16 13:09 业余草

这么好的文章不评论评论亏大了！www.xttblog.com

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

【推荐】超50万VC++源码：大型组态工控、电力仿真CAD与GIS源码库！

【活动】华为云普惠季 1折秒杀 狂欢继续

【工具】SpreadJS纯前端表格控件，可嵌入应用开发的在线Excel

【腾讯云】拼团福利，AMD云服务器8元/月





最新新闻：

- 揭秘恒大与贾跃亭纠纷的事实与隐秘情节
 - 贾跃亭的诱饵：冒险精神 有钱人或有点成绩的群体更易被说服
 - Instagram下载数据副本工具出漏洞：用户密码或泄露
 - 苹果CEO库克：自由市场在科技行业失效 监管不可避免
 - 小米激光电视上架美国沃尔玛：生态链军团统一跟进
- » 更多新闻...

Copyright ©2018 dreamcatcher-cx