Android HashMap 源码详解（上）

2016-09-18 安卓开发精选

(点击上方公众号，可快速关注)

> 来源：山大王
> 链接：blog.csdn.net/abcdef314159/article/details/51165630

这一篇来分析一下HashMap的源码，为了在后面讲解Android缓存机制做准备，因为我们知道在Android的缓存机制中无论是用第三方的还是我们自己写的，一般都会用到LruCache或者LinkedHashMap类，而LruCache里面封装的又是LinkedHashMap，LinkedHashMap又是HashMap的子类，所以这一篇我们有必要把HashMap的源码分析一下，然后最终再来讲解一下Android的缓存机制。HashMap的构造方法比较多，我们就随便挑两个最常用的来分析，其实其他的也都差不多，我们看一下

```
public HashMap() {
    table = (HashMapEntry<K, V>[]) EMPTY_TABLE;
    threshold = -1; // Forces first put invocation to replace EMPTY_TABLE
}
```

在HashMap中有一个table，保存的是一个HashMapEntry类型的数组，也是后面我们要讲的专门存储数据用的，而EMPTY_TABLE其实就是一个程度为2的HashMapEntry类型的数组，来看一下

```
private static final int MINIMUM_CAPACITY = 4;
.....................
private static final Entry[] EMPTY_TABLE
        = new HashMapEntry[MINIMUM_CAPACITY >>> 1];
```

MINIMUM_CAPACITY往右移动一位，大小变为2了，我们再来看一下HashMapEntry这个类

```
static class HashMapEntry<K, V> implements Entry<K, V> {
    final K key;
    V value;
    final int hash;
    HashMapEntry<K, V> next;

    HashMapEntry(K key, V value, int hash, HashMapEntry<K, V> next) {
        this.key = key;
        this.value = value;
        this.hash = hash;
```

```
        this.next = next;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
        return value;
    }

    public final V setValue(V value) {
        V oldValue = this.value;
        this.value = value;
        return oldValue;
    }

    @Override public final boolean equals(Object o) {
        if (!(o instanceof Entry)) {
            return false;
        }
        Entry<?, ?> e = (Entry<?, ?>) o;
        return Objects.equal(e.getKey(), key)
                && Objects.equal(e.getValue(), value);
    }

    @Override public final int hashCode() {
        return (key == null ? 0 : key.hashCode()) ^
                (value == null ? 0 : value.hashCode());
    }

    @Override public final String toString() {
        return key + "=" + value;
    }
}
```

我们看到他有4个变量，其中的key和value就是我们常见的两个，而另外的两个是存储HashMapEntry的时候用的，其中他还有几个的方法，因为我们知道HashMap是一个数组加链表的形式存储的，hashCode是用来判断存储在哪个数组里面的，equals判断是否是同一个对象，HashMap存储的其实就是HashMapEntry。

我们再看HashMap的另外一个构造方法

```
private static final int MINIMUM_CAPACITY = 4;

/**
 * Max capacity for a HashMap. Must be a power of two >= MINIMUM_CAPACITY.
 */
private static final int MAXIMUM_CAPACITY = 1 << 30;
.....
public HashMap(int capacity) {
    if (capacity < 0) {
        throw new IllegalArgumentException("Capacity: " + capacity);
    }

    if (capacity == 0) {
        @SuppressWarnings("unchecked")
        HashMapEntry<K, V>[] tab = (HashMapEntry<K, V>[]) EMPTY_TABLE;
        table = tab;
        threshold = -1; // Forces first put() to replace EMPTY_TABLE
        return;
    }

    if (capacity < MINIMUM_CAPACITY) {
        capacity = MINIMUM_CAPACITY;
    } else if (capacity > MAXIMUM_CAPACITY) {
        capacity = MAXIMUM_CAPACITY;
    } else {
        capacity = Collections.roundUpToPowerOfTwo(capacity);
    }
    makeTable(capacity);
}
```

上面的比较简单，我们先来看第26行，调用的是Collections的一个方法，其实他表示的就是找到一个比capacity大的2的n次方的最小值，可能不是太明白，我举个例子，如果capacity是3就返回4，因为2的1次方比3小，不合适，所以是2的2次方，同样如果capacity是9则返回16，因为2的3次方是8比9小，所以返回2的4次方16,这样说大家可能比较明白。我们顺便来看一下他的源码

```
/**
 * Returns the smallest power of two >= its argument, with several caveats:
 * If the argument is negative but not Integer.MIN_VALUE, the method returns
 * zero. If the argument is > 2^30 or equal to Integer.MIN_VALUE, the method
```

```
 * returns Integer.MIN_VALUE. If the argument is zero, the method returns
 * zero.
 * @hide
 */
public static int roundUpToPowerOfTwo(int i) {
    i--; // If input is a power of two, shift its high-order bit right.

    // "Smear" the high-order bit all the way to the right.
    i |= i >>>  1;
    i |= i >>>  2;
    i |= i >>>  4;
    i |= i >>>  8;
    i |= i >>> 16;

    return i + 1;
}
```

其实上面代码很好理解，就是高位（这里我们只研究正整数，这个高位不是二进制的最高位，这里是指把它转化为二进制之后从左往右数第一次出现1的那个位置）的1往后移位然后通过或运算把后面的所有位都置1，然后最后在加上1就相当于高位之后（包括高位）全部为0，而高位的前一位进位为1，就是2的n次方了，并且这个2的n次方正好是比我们要计算的这个数大的最小的2的n次方，可能大家会有疑问，第10行为什么还要执行i–，因为如果不执行i–，当我们传入的正好是2的n次方的时候，结果返回的是2的n+1次方，这不是我们想要的结果，我来给大家画个图可能大家就明白了

然后我们再看上面的makeTable方法，我们来看一下源码

```
/**
 * Allocate a table of the given capacity and set the threshold accordingly.
 * @param newCapacity must be a power of two
 */
private HashMapEntry<K, V>[] makeTable(int newCapacity) {
    @SuppressWarnings("unchecked") HashMapEntry<K, V>[] newTable
            = (HashMapEntry<K, V>[]) new HashMapEntry[newCapacity];
    table = newTable;
    threshold = (newCapacity >> 1) + (newCapacity >> 2); // 3/4 capacity
    return newTable;
}
```

我们看到其实就是初始化table，这个table是一个HashMapEntry类型的数组，就是存放HashMapEntry的，然后我们还看到threshold这样的一个值，其实他就是数组存放的阈值，他不像ArrayList等数组满了之后再扩容，HashMap是判断当前的HashMapEntry对象如果超过threshold就会扩容，他扩容的最终大小必须是2的n次方，这一点要牢记，待会会讲到，我们看到threshold值大概是newCapacity的3/4也就是数组长度的75%。

到现在为止HashMap的初始化我们已经基本上讲完了，我们看到HashMap源码中的方法也比较多，我们就捡我们常用的的几个来分析，我们先看一下put方法

```
@Override public V put(K key, V value) {
    if (key == null) {
        return putValueForNullKey(value);
    }

    int hash = Collections.secondaryHash(key);
    HashMapEntry<K, V>[] tab = table;
    int index = hash & (tab.length - 1);
    for (HashMapEntry<K, V> e = tab[index]; e != null; e = e.next) {
        if (e.hash == hash && key.equals(e.key)) {
            preModify(e);
            V oldValue = e.value;
            e.value = value;
            return oldValue;
        }
    }
```

```
    // No entry for (non-null) key is present; create one
    modCount++;
    if (size++ > threshold) {
        tab = doubleCapacity();
        index = hash & (tab.length - 1);
    }
    addNewEntry(key, value, hash, index);
    return null;
}
```

HashMap中是允许key为null的，我们看上面的2-4行，如果key为null就会调用putValueForNullKey这个方法，我们看一下他的源码

```
private V putValueForNullKey(V value) {
    HashMapEntry<K, V> entry = entryForNullKey;
    if (entry == null) {
        addNewEntryForNullKey(value);
        size++;
        modCount++;
        return null;
    } else {
        preModify(entry);
        V oldValue = entry.value;
        entry.value = value;
        return oldValue;
    }
}
```

我们看到entryForNullKey其实就是个HashMapEntry

```
transient HashMapEntry<K, V> entryForNullKey;
```

我们再来看一下如果entry == null，说明HashMap中没有key为null的HashMapEntry，那么就造一个,然后size和modCount都要加一，size是 HashMap的大小，modCount是指修改的次数，主要在循环输出的时候用来判断HashMap是否有改动，如果有改动就会报ConcurrentModificationException异常，我们来看一下addNewEntryForNullKey的源码

```
void addNewEntryForNullKey(V value) {
    entryForNullKey = new HashMapEntry<K, V>(null, value, 0, null);
}
```

我们发现代码量很少，就一行，初始化了一个HashMapEntry对象，然后把value存进去，其他的都为null或0，这就是一个key为null的HashMapEntry，在上面我们看到如果存在key为null的HashMapEntry就会把entryForNullKey的value修改，然后返回原来的value，这之前又调用了preModify方法，其实他是个空方法，什么都没做,但他会在他的子类LinkedHashMap中调用，key为null的我们分析完了。

下面我们再来分析key不为null的情况，我们还看上面的put方法，在第6行计算出hash值，第8行根据计算的hash值算出存储的位置，其实第8行很有讲究，设计的非常巧妙，我们在前面说过HashMap初始化大小的时候都是2的n次方，原因就在这，因为2的n次方换成二进制就是前面有个1后面全是0，如果减去1就变成之前为1的位和他前面的都是0，而他后面的全是1。

举个简单例子，2的3次方是8，换成二进制就是前面有个1后面有3个0，一共4位，如果减去1就变成后面3位都是1，前面的都为0，如果在与hash值与运算，那么算得结果永远都是0到2的n次方减1之间，永远都不会出现数组越界。

我们看到上面put方法中的第8行，通过与运算获得存放数组的下标index，然后在10到15行，通过查找他所在的那个数组有没有存放过key值相同的，如果有就把他替换，然后返回原来的value，比较的时候首先是通过hash值，如果hash值相同则调用equals方法，这里要说一下，如果hash值不同，则entry肯定不同，如果hash相同，则entry可能相同也可能不同，需要调用equals进行比较，相反如果equals相同则hash值肯定相同，如果equals不相同则hash值有可能相同也有可能不相同。我们再来看一下put的20到23行，如果放进去之后大小大于阈值则会扩容，这个阈值threshold我们刚才在上面讲过，大概是数组长度的75%，扩容调用了doubleCapacity方法，因为扩容之后老数据还要重新排放，这个源码我们最后在分析，扩容之后然后调用addNewEntry方法，把它存进去，我们来看一下他的源码

```
void addNewEntry(K key, V value, int hash, int index) {
    table[index] = new HashMapEntry<K, V>(key, value, hash, table[index]);
}
```

代码很简单，就一行，我们看到前面有个table[index]，后面也有一个，大家不要晕，代码是先要执行后面的，执行完之后然后在赋值给前面的，我们上面讲过HashMapEntry构造函数的最后一个参数就是他的next，也是HashMapEntry类型的，如果原来没有就为空，就把当前new的HashMapEntry放到数组中，如果有就把他挂到当前new的HashMapEntry后面，然后再把new的这个HashMapEntry放到数组中，因为HashMap是数组加链表的形式存放的，我给大家画一个图便于理解。

首先是根据Hash值找到所在数组的下标，因为不同的Hash值通过数组长度的与运算可能会有相同的结果，如果原来数组中没有就把它存进去，如果原来数组有了就判断他的key值是否相同，如果相同就把他替换，如果不相同就把原来数组这个位置的挂载到当前的下面，然后把当前的放到数组中的这个位置。

下面我们在看它的另一个方法get

```
public V get(Object key) {
    if (key == null) {
        HashMapEntry<K, V> e = entryForNullKey;
        return e == null ? null : e.value;
    }

    int hash = Collections.secondaryHash(key);
    HashMapEntry<K, V>[] tab = table;
    for (HashMapEntry<K, V> e = tab[hash & (tab.length - 1)];
            e != null; e = e.next) {
        K eKey = e.key;
        if (eKey == key || (e.hash == hash && key.equals(eKey))) {
            return e.value;
        }
    }
```

```
    return null;
}
```

上面代码也很简单，首先如果key为null则第2-5行是取出key为null的value，我们主要来看9-15行，第9行通过计算的hash值找到所在的数组的下标，然后取出HashMapEntry，如果不为空就比较key值是否相同，如果相同就返回，不相同就找它的next下一个，就像上面HashMap结构图中的那样，首先要找到所在的位置，然后从上往下比较，找到则返回value，否则返回null。

我们再来看一下HashMap的还一个方法containsKey和上面的get差不多，只不过他是个判断，但并不会返回value的值，我们简单看一下

```
@Override public boolean containsKey(Object key) {
    if (key == null) {
        return entryForNullKey != null;
    }

    int hash = Collections.secondaryHash(key);
    HashMapEntry<K, V>[] tab = table;
    for (HashMapEntry<K, V> e = tab[hash & (tab.length - 1)];
            e != null; e = e.next) {
        K eKey = e.key;
        if (eKey == key || (e.hash == hash && key.equals(eKey))) {
            return true;
        }
    }
    return false;
}
```

接下文