HashMap源码解析（JAVA 1.6）

本文主要从构造函数、put函数、get函数、底层hash表等4个方面来对HashMap的源码进行解析。前三个都是我们在编程中经常要用到的，最后一个则是对前三个的支撑。

构造函数

HashMap提供的三个构造函数

```
/**
    *自定义HashMap的初始容量与负载因子
    * @param  initialCapacity   初始容量
    * @param  loadFactor    负载因子
    * @throws IllegalArgumentException if the initial capacity is negative
    *         or the load factor is nonpositive
    */
   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);

      // Find a power of 2 >= initialCapacity
      int capacity = 1;
      while (capacity < initialCapacity)
         capacity <<= 1;

      this.loadFactor = loadFactor;
      threshold = (int)(capacity * loadFactor);  //这个参数是HashMap进行扩容的标志，当容量达到threshold时，HashMap便会进行扩容操作。它的值由容量和负载一起决定
      table = new Entry[capacity];
      init();
   }
/**
    * 自定义HashMap的初始容量
    *
    * @param  initialCapacity 初始容量
    * @throws IllegalArgumentException if the initial capacity is negative.
    */
   public HashMap(int initialCapacity) {
      this(initialCapacity, DEFAULT_LOAD_FACTOR); //默认负责因子为0.75
   }

   /**
    *无参构造函数 默认初始容量为16  默认负责因子为0.75
    */
   public HashMap() {
      this.loadFactor = DEFAULT_LOAD_FACTOR;
      threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
      table = new Entry[DEFAULT_INITIAL_CAPACITY];
      init();
   }
/**
    * 根据已有的散列表进行初始化。
    * @throws  NullPointerException if the specified map is null
    */
   public HashMap(Map<? extends K, ? extends V> m) {
      this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1,
                    DEFAULT_INITIAL_CAPACITY), DEFAULT_LOAD_FACTOR);
      putAllForCreate(m);
   }
```

了解了这些构造函数后，如果在已知将要构造的HashMap大小的情况下，可以用前两种方法，设定初始容量大小，这样可以减少HashMap的扩容次数，毕竟每次扩容都涉及到数组的拷贝，这样可以提升效率。这属于细节问题。

HashMap的put函数

```
/**
 *插入K-V键值对
 */
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key.hashCode()); //获取主键的HASH值
    int i = indexFor(hash, table.length); //根据主键的HASH值得到其在散列表中的索引
    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))) { //判断是否已经存在
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    modCount++;
    addEntry(hash, key, value, i); //添加到散列表
    return null;
}

/**
 * 添加KEY为NULL的K-V键值对
 */
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); //KEY为NULL的键值对，索引永远为0
    return null;
}

/**
 * 插入K-V键值对，但不会对扩容进行影响。即不进行扩容判断
 */
private void putForCreate(K key, V value) {
    int hash = (key == null) ? 0 : hash(key.hashCode());
    int i = indexFor(hash, table.length);

    /**
     * Look for preexisting entry for key.  This will never happen for
     * clone or deserialize.  It will only happen for construction if the
     * input Map is a sorted map whose ordering is inconsistent w/ equals.
     */
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) {
            e.value = value;
            return;
        }
    }

    createEntry(hash, key, value, i);
}
/**
 * 插入一个散列表，但不会对扩容进行影响。
 */
private void putAllForCreate(Map<? extends K, ? extends V> m) {
    for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(); i.hasNext(); ) {
```

```
            Map.Entry<? extends K, ? extends V> e = i.next();
            putForCreate(e.getKey(), e.getValue());
        }
    }
```

HashMap的get函数

```
/**
     *根据K值获取V值
     */
    public V get(Object key) {
        if (key == null) //判断K是否为NULL
            return getForNullKey();
        int hash = hash(key.hashCode()); //获取其HASH值
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) { //遍布K所在的链表
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
                return e.value;
        }
        return null;
    }

    /**
     *获取K为NULL的V值
     */
    private V getForNullKey() {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }

    /**
     *获取K值 对应的实体类对象(散列对象)
     */
    final Entry<K,V> getEntry(Object key) {
        int hash = (key == null) ? 0 : hash(key.hashCode()); //判断K是否为NULL
        for (Entry<K,V> e = table[indexFor(hash, table.length)]; //遍历K所在的链表
             e != null;
             e = e.next) {
            Object k;
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                return e;
        }
        return null;
    }
```

HashMap的底层hash表

```
/**
     * 默认容量 -必须为2的幂次方.
     */
    static final int DEFAULT_INITIAL_CAPACITY = 16;
    /**
     * 最大容量为2的30次方
     */
    static final int MAXIMUM_CAPACITY = 1 << 30;
    /**
     * 默认负载值
     */
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    /**
     *散列表
     */
    transient Entry[] table;
```

HashMap的底层存的不是一个简单的数组，而是一个链表数组。如图1所示：

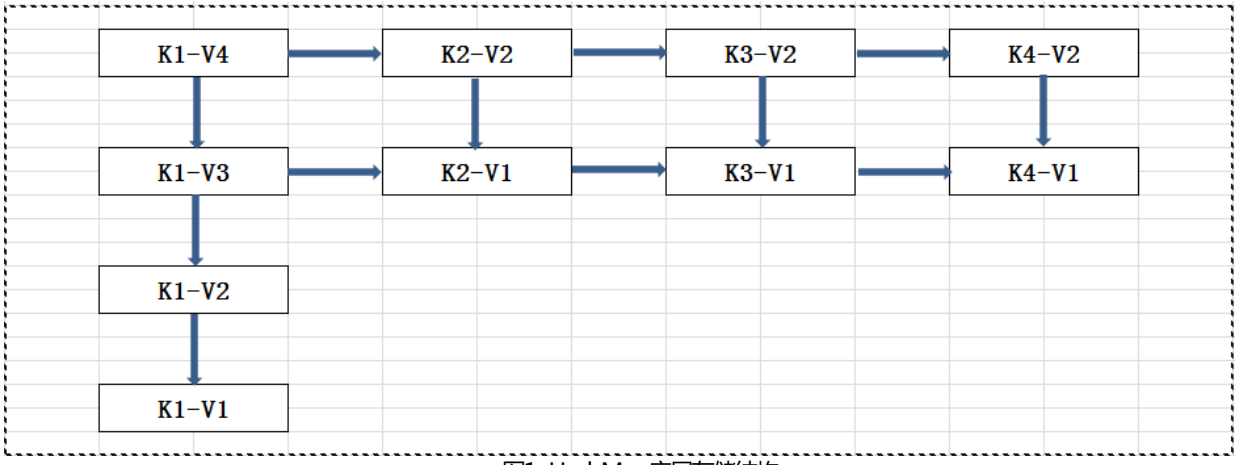

图1 HashMap底层存储结构

从横向来看，它是一个数组，从纵向来看它是一个链表。在HashMap里面是用一个 Entry[] table 来存储这个链表数组的。那么它又是怎样对这个链表数组进行操作的呢？

这里以向HashMap中插入数组为例：

```
 /**
    *插入K-V键值对
    */
   public V put(K key, V value) {
      if (key == null)
         return putForNullKey(value);
      int hash = hash(key.hashCode()); //获取主键的HASH值
      int i = indexFor(hash, table.length); //根据主键的HASH值得到其在散列表中的索引
      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))) { //判断是否已经存在
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
         }
      }

      modCount++;
      addEntry(hash, key, value, i); //添加到散列表
      return null;
   }
```

主要分为4步：

第一步 对Key进行判断，看是否为NULL 如果为NULL 则直接可得到HASH值为0

第二步 如果非NULL 则对Key进行HASH，得到对应的hash(Key)

第三步 根据IndexFor()方法，用hash(Key)与length-1进行位与运算，得到当前hash(Key)在散列表中对应的索引，即横向索引值。

IndexFor()方法源码如下：

```
 /**
    * Returns index for hash code h.
    */
   static int indexFor(int h, int length) {
      return h & (length-1);
   }
```

第四步 在对应的链表中查询，看是否已经存在对应的KEY，如果存在，则直接替换，如果不存在，则在创建一个Entry对象，同时将其添加到hash(Key)对应的链表的头部。

运用数组进行快速索引，同时利用链表去解决hash冲突。最新插入的在最前。

```
/**
    * 在链表中添加Entry对象
    */
   void addEntry(int hash, K key, V value, int bucketIndex) {
 Entry<K,V> e = table[bucketIndex];
      table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
      if (size++ >= threshold)
         resize(2 * table.length);
   }
/**
       *Entry 对象构造 方法
       */
```

```
Entry(int h, K k, V v, Entry<K,V> n) {
    value = v;
    next = n;
    key = k;
    hash = h;
}
```