1. HashMap 的数据结构

数据结构中有数组和链表来实现对数据的存储，但这两者基本上是两个极端。

数组

数组存储区间是连续的，占用内存严重，故空间复杂的很大。但数组的二分查找时间复杂度小，为 O(1)；数组的特点是：寻址容易，插入和删除困难；

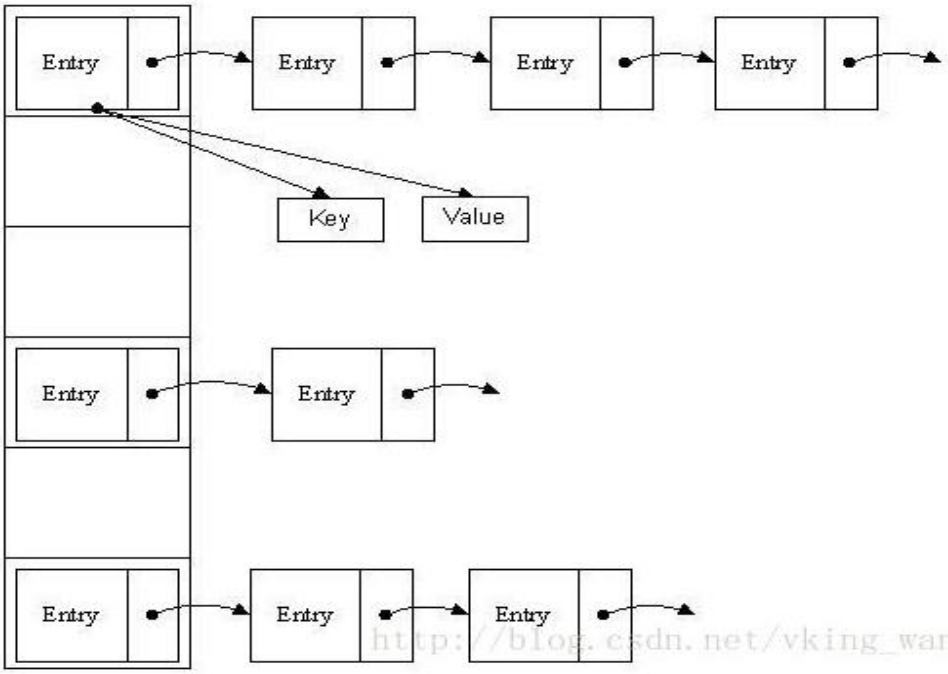
链表

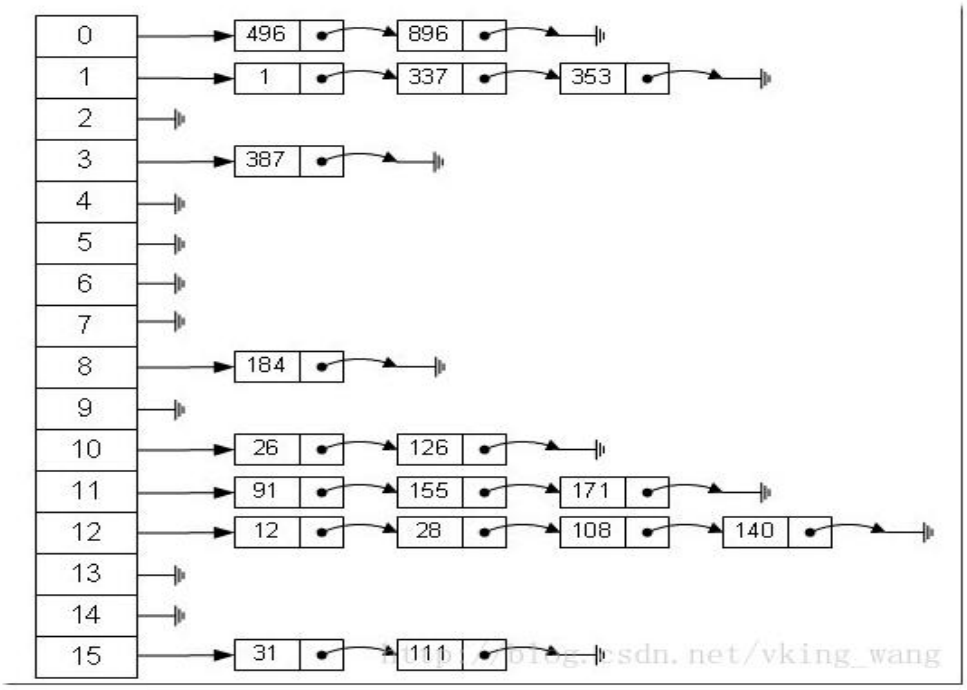
链表存储区间离散，占用内存比较宽松，故空间复杂度很小，但时间复杂度很大，达 O（N）。**链表**的特点是：寻址困难，插入和删除容易。

哈希表

那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表。哈希表（(Hash table）既满足了数据的查找方便，同时不占用太多的内容空间，使用也十分方便。

哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，我们可以理解为“**链表的数组**”，如图：





从上图我们可以发现哈希表是由**数组+链表**组成的，一个长度为 16 的数组中，每个元素存储的是一个链表的头结点。那么这些元素是按照什么样的规则存储到数组中呢。一般情况是通过 hash(key)%len 获得，也就是元素的 key 的哈希值对数组长度取模得到。比如上述哈希表中，12%16=12,28%16=12,108%16=12,140%16=12。所以 12、28、108 以及 140 都存储在数组下标为 12 的位置。

HashMap 其实也是一个线性的数组实现的,所以可以理解为其存储数据的容器就是一个线性数组。这可能让我们很不解，一个线性的数组怎么实现按键值对来存取数据呢？这里 HashMap 有做一些处理。

首先 HashMap 里面实现一个静态内部类 Entry，其重要的属性有 *key , value, next*，从属性 key,value 我们就能很明显的看出来 Entry 就是 HashMap 键值对实现的一个基础 bean，我们上面说到 HashMap 的基础就是一个线性数组，这个数组就是 Entry[]，Map 里面的内容都保存在 Entry[]里面。

```
    /**
     * The table, resized as necessary. Length MUST Always be a power of tw
o.
     */

    transient Entry[] table;
```

2. HashMap 的存取实现

既然是线性数组，为什么能随机存取？这里 HashMap 用了一个小算法，大致是这样实现：

```
// 存储时：
int hash = key.hashCode(); // 这个 hashCode 方法这里不详述,只要理解每个 key 的 hash 是一个固定的 int 值
int index = hash % Entry[].length;
Entry[index] = value;

// 取值时：
int hash = key.hashCode();
int index = hash % Entry[].length;
return Entry[index];
```

1）put

疑问：如果两个 key 通过 hash%Entry[].length 得到的 index 相同，会不会有覆盖的危险？

这里 HashMap 里面用到链式数据结构的一个概念。上面我们提到过 Entry 类里面有一个 next 属性，作用是指向下一个 Entry。打个比方， 第一个键值对 A 进来，通过计算其 key 的 hash 得到的 index=0，记做:Entry[0] = A。一会后又进来一个键值对 B，通过计算其 index 也等于 0，现在怎么办？ HashMap 会这样做:*B.next = A*,Entry[0] = B,如果又进来 C,index 也等于 0,那么 *C.next = B*,Entry[0] = C；这样我们发现 index=0 的地方其实存取了 A,B,C 三个键值对,他们通过 next 这个属性链接在一起。所以疑问不用担心。**也就是说数组中存储的是最后插入的元素。**到这里为止，HashMap 的大致实现，我们应该已经清楚了。

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value); //null 总是放在数组的第一个链表中
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        //遍历链表
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            //如果 key 在链表中已存在，则替换为新 value
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
```

```
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }
        modCount++;
        addEntry(hash, key, value, i);
        return null;

    }
```

```
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); //参数 e，是
Entry.next
    //如果 size 超过 threshold，则扩充 table 大小。再散列
    if (size++ >= threshold)
            resize(2 * table.length);
}
```

当然 HashMap 里面也包含一些优化方面的实现，这里也说一下。比如：Entry[] 的长度一定后，随着 map 里面数据的越来越长，这样同一个 index 的链就会很长，会不会影响性能？HashMap 里面设置一个因子，随着 map 的 size 越来越大，Entry[] 会以一定的规则加长长度。

2）get

```
 public V get(Object key) {
        if (key == null)
            return getForNullKey();
        int hash = hash(key.hashCode());
        //先定位到数组元素，再遍历该元素处的链表
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
                return e.value;
        }
        return null;
}
```

3）null key 的存取

null key 总是存放在 Entry[]数组的第一个元素。

```
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);
    return null;
}

private V getForNullKey() {
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null)
            return e.value;
    }
    return null;
}
```

4）确定数组 index：hashcode % table.length 取模

HashMap 存取时，都需要计算当前 key 应该对应 Entry[]数组哪个元素，即计算数组下标；算法如下：

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

按位取并，作用上相当于取模 mod 或者取余%。
这意味着数组下标相同，并不表示 hashCode 相同。

5）table 初始大小

```
public HashMap(int initialCapacity, float loadFactor) {
        .....          // Find a power of 2 >= initialCapacity
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;
        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
        table = new Entry[capacity];
        init();
    }
```

注意 table 初始大小并不是构造函数中的 initialCapacity！！

而是 >= initialCapacity 的 2 的 n 次幂！！！！

——为什么这么设计呢？—

3. 解决 hash 冲突的办法

1. 开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）
2. 再哈希法
3. **链地址法**
4. 建立一个公共溢出区

Java 中 hashmap 的解决办法就是采用的链地址法。

4. 再散列 rehash 过程

当哈希表的容量超过默认容量时，必须调整 table 的大小。当容量已经达到最大可能值时，那么该方法就将容量调整到 Integer.MAX_VALUE 返回，这时，需要创建一张新表，将原表的映射到新表中。

```
/**
 * Rehashes the contents of this map into a new array with a
 * larger capacity.  This method is called automatically when the
 * number of keys in this map reaches its threshold.
 *
 * If current capacity is MAXIMUM_CAPACITY, this method does not
 * resize the map, but sets threshold to Integer.MAX_VALUE.
 * This has the effect of preventing future calls.
 *
 * @param newCapacity the new capacity, MUST be a power of two;
 *        must be greater than current capacity unless current
 *        capacity is MAXIMUM_CAPACITY (in which case value
 *        is irrelevant).
 */
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);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);

}



/**
 * Transfers all entries from current table to newTable.
 */
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                //重新计算 index
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
```

```
                    newTable[i] = e;
                    e = next;
                } while (e != null);
            }
        }

    }
```

---------- 作者：AlphaWang 来源：CSDN