20180326_hashmap 的实现和reHash

1.HashMap的内部实现机制

HashMap是对数据结构中哈希表(Hash Table)的实现，Hash表又叫散列表。Hash表是根据关键码Key来访问其对应的值Value的数据结构，它通过一个映射函数把关键码映射到表中一个位置来访问该位置的值，从而加快查找的速度。这个映射函数叫做Hash函数，存放记录的数组叫做Hash表。

在Java中，HashMap的内部实现结合了链表和数组的优势，链接节点的数据结构是Entry，每个Entry对象的内部又含有指向下一个Entry类型对象的引用，如以下代码所示：

```
static class Entry<K,V> implements Map.Entry<K,V> {
      final K key;
      V value;
      Entry<K,V> next; //Entry类型内部有一个自己 类型的引用,指向下一个 Entry
      final int hash;
      ...
}
```

在HashMap的构造函数中可以看到，Entry表被申明为了数组，如以下代码所示：

```
 public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        threshold = (int)(DEFAULT_INITIAL_CAPACITY  * DEFAULT_LOAD_FACTOR);
        table = new Entry[DEFAULT_INITIAL_CAPACITY];
        init();
    }
```

在以上构造函数中，默认的DEFAULT_INITIAL_CAPACITY值为16，DEFAULT_LOAD_FACTOR的值为0.75。当put一个元素到HashMap中去时，其内部实现如下：

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        ...
}
```

可以看到put函数中用一个hash函数来得到哈希值，需要指出的是，HashTable在实现时直接用了hashCode作为哈希值，因此采用HashMap代替HashTable有一定的优化。

put函数中用到的两个函数hash和indexFor其实现分别如下：

```
static int hash(int h) {
        // This function ensures that hashCodes that differ only by
        // constant multiples at each bit position have a bounded
        // number of collisions (approximately 8 at default load factor).
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }
```

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

至于hash函数为什么这样设计，这涉及到具体哈希函数的设计问题了，需要考虑的是哈希算法的时间复杂度，同时尽量使得数组上每个位置都有值，求得时间和空间的最优。

indexFor函数则用了一个很巧妙的与运算将index值限制在了length-1之内。

当然，hash函数存在冲突的情况，同一个key对应的hash值可能相同，这时候hash值相同的元素就会用链接进行存储，HashMap的get方法在获取value的时候会对链表进行遍历，把key值相匹配的value取出来。

2.Hash的实现

主要是哈希算法和冲突的解决。

3.什么时候ReHash

在介绍HashMap的内部实现机制时提到了两个参数，DEFAULT_INITIAL_CAPACITY和DEFAULT_LOAD_FACTOR，DEFAULT_INITIAL_CAPACITY是table数组的容量，DEFAULT_LOAD_FACTOR则是为了最大程度避免哈希冲突，提高HashMap效率而设置的一个影响因子，将其乘以DEFAULT_INITIAL_CAPACITY就得到了一个阈值threshold，当**HashMap的容量达到threshold**时就需要进行扩容，这个时候就要进行ReHash操作了，可以看到下面addEntry函数的实现，当size达到threshold时会调用resize函数进行扩容。

```
void addEntry(int hash, K key, V value, int bucketIndex) {
ntry<K,V> e = table[bucketIndex];
      table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
      if (size++ >= threshold)
          resize(2 * table.length);
  }
```

在扩容的过程中需要进行ReHash操作，而这是非常耗时的，在实际中应该尽量避免。