Java集合学习手册（11）：Java HashMap源码全剖析

HashMap简介 HashMap是基于哈希表实现的，每一个元素都是一个key-value对，其内部通过单链表解决冲突问题，容量不足（超过了阈值）时，同样会自动增长。

HashMap是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

HashMap实现了Serializable接口，因此它支持序列化，实现了Cloneable接口，能被克隆。

一、HashMap源码剖析

HashMap的源码如下（加入了比较详细的注释）：

```
package java.util;
import java.io.*;

public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{

    // 默认的初始容量（容量为HashMap中槽的数目）是16，且实际容量必须是2的整数次幂。
    static final int DEFAULT_INITIAL_CAPACITY = 16;

    // 最大容量（必须是2的幂且小于2的30次方，传入容量过大将被这个值替换）
    static final int MAXIMUM_CAPACITY = 1 << 30;

    // 默认加载因子为0.75
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    // 存储数据的Entry数组，长度是2的幂。
    // HashMap采用链表法解决冲突，每一个Entry本质上是一个单向链表
    transient Entry[] table;

    // HashMap的底层数组中已用槽的数量
    transient int size;

    // HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量*加载因子）
    int threshold;
```

```
    // 加载因子实际大小
    final float loadFactor;

    // HashMap被改变的次数
    transient volatile int modCount;

    // 指定“容量大小”和“加载因子”的构造函数
    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        // HashMap的最大容量只能是MAXIMUM_CAPACITY
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //加载因此不能小于0
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);

        // 找出“大于initialCapacity”的最小的2的幂
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;

        // 设置“加载因子”
        this.loadFactor = loadFactor;
        // 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap
的容量加倍。
        threshold = (int)(capacity * loadFactor);
        // 创建Entry数组，用来保存数据
        table = new Entry[capacity];
        init();
    }


    // 指定“容量大小”的构造函数
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

    // 默认构造函数。
    public HashMap() {
        // 设置“加载因子”为默认加载因子0.75
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        // 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap
的容量加倍。
        threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
```

```
        // 创建Entry数组，用来保存数据
        table = new Entry[DEFAULT_INITIAL_CAPACITY];
        init();
    }

    // 包含“子Map”的构造函数
    public HashMap(Map<? extends K, ? extends V> m) {
        this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1,
                      DEFAULT_INITIAL_CAPACITY), DEFAULT_LOAD_FACTOR);
        // 将m中的全部元素逐个添加到HashMap中
        putAllForCreate(m);
    }

    //求hash值的方法，重新计算hash值
    static int hash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }

    // 返回h在数组中的索引值，这里用&代替取模，旨在提升效率
    // h & (length-1)保证返回值的小于length
    static int indexFor(int h, int length) {
        return h & (length-1);
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    // 获取key对应的value
    public V get(Object key) {
        if (key == null)
            return getForNullKey();
        // 获取key的hash值
        int hash = hash(key.hashCode());
        // 在“该hash值对应的链表”上查找“键值等于key”的元素
        for (Entry<K,V> e = table[indexFor(hash, table.length)];
             e != null;
             e = e.next) {
            Object k;
            //判断key是否相同
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
                return e.value;
        }
```

```
        //没找到则返回null
        return null;
    }

    // 获取“key为null”的元素的值
    // HashMap将“key为null”的元素存储在table[0]位置，但不一定是该链表的第一个位置!
    private V getForNullKey() {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }

    // HashMap是否包含key
    public boolean containsKey(Object key) {
        return getEntry(key) != null;
    }

    // 返回“键为key”的键值对
    final Entry<K,V> getEntry(Object key) {
        // 获取哈希值
        // HashMap将“key为null”的元素存储在table[0]位置，“key不为null”的则调用hash()计
算哈希值
        int hash = (key == null) ? 0 : hash(key.hashCode());
        // 在“该hash值对应的链表”上查找“键值等于key”的元素
        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 != null && key.equals(k))))
                return e;
        }
        return null;
    }

    // 将“key-value”添加到HashMap中
    public V put(K key, V value) {
        // 若“key为null”，则将该键值对添加到table[0]中。
        if (key == null)
            return putForNullKey(value);
        // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出!
```

```
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
        modCount++;
        //将key-value添加到table[i]处
        addEntry(hash, key, value, i);
        return null;
    }

    // putForNullKey()的作用是将“key为null”键值对添加到table[0]位置
    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;
            }
        }
        // 如果没有存在key为null的键值对，则直接题阿见到table[0]处!
        modCount++;
        addEntry(0, null, value, 0);
        return null;
    }

    // 创建HashMap对应的“添加方法”，
    // 它和put()不同。putForCreate()是内部方法，它被构造函数等调用，用来创建HashMap
    // 而put()是对外提供的往HashMap中添加元素的方法。
    private void putForCreate(K key, V value) {
        int hash = (key == null) ? 0 : hash(key.hashCode());
        int i = indexFor(hash, table.length);

        // 若该HashMap表中存在“键值等于key”的元素，则替换该元素的value值
        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;
            }
        }
```

```
        // 若该HashMap表中不存在“键值等于key”的元素，则将该key-value添加到HashMap中
        createEntry(hash, key, value, i);
    }

    // 将“m”中的全部元素都添加到HashMap中。
    // 该方法被内部的构造HashMap的方法所调用。
    private void putAllForCreate(Map<? extends K, ? extends V> m) {
        // 利用迭代器将元素逐个添加到HashMap中
        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());
        }
    }

    // 重新调整HashMap的大小，newCapacity是调整后的容量
    void resize(int newCapacity) {
        Entry[] oldTable = table;
        int oldCapacity = oldTable.length;
        //如果就容量已经达到了最大值，则不能再扩容，直接返回
        if (oldCapacity == MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return;
        }

        // 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，
        // 然后，将“新HashMap”赋值给“旧HashMap”。
        Entry[] newTable = new Entry[newCapacity];
        transfer(newTable);
        table = newTable;
        threshold = (int)(newCapacity * loadFactor);
    }

    // 将HashMap中的全部元素都添加到newTable中
    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;
                    int i = indexFor(e.hash, newCapacity);
                    e.next = newTable[i];
                    newTable[i] = e;
                    e = next;
                } while (e != null);
```

```
            }
        }
    }

    // 将"m"的全部元素都添加到HashMap中
    public void putAll(Map<? extends K, ? extends V> m) {
        // 有效性判断
        int numKeysToBeAdded = m.size();
        if (numKeysToBeAdded == 0)
            return;

        // 计算容量是否足够,
        // 若“当前阀值容量 < 需要的容量”，则将容量x2。
        if (numKeysToBeAdded > threshold) {
            int targetCapacity = (int)(numKeysToBeAdded / loadFactor + 1);
            if (targetCapacity > MAXIMUM_CAPACITY)
                targetCapacity = MAXIMUM_CAPACITY;
            int newCapacity = table.length;
            while (newCapacity < targetCapacity)
                newCapacity <<= 1;
            if (newCapacity > table.length)
                resize(newCapacity);
        }

        // 通过迭代器，将“m”中的元素逐个添加到HashMap中。
        for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet
().iterator(); i.hasNext(); ) {
            Map.Entry<? extends K, ? extends V> e = i.next();
            put(e.getKey(), e.getValue());
        }
    }

    // 删除“键为key”元素
    public V remove(Object key) {
        Entry<K,V> e = removeEntryForKey(key);
        return (e == null ? null : e.value);
    }

    // 删除“键为key”的元素
    final Entry<K,V> removeEntryForKey(Object key) {
        // 获取哈希值。若key为null，则哈希值为0；否则调用hash()进行计算
        int hash = (key == null) ? 0 : hash(key.hashCode());
        int i = indexFor(hash, table.length);
        Entry<K,V> prev = table[i];
        Entry<K,V> e = prev;

        // 删除链表中“键为key”的元素
        // 本质是“删除单向链表中的节点”
```

```
        while (e != null) {
            Entry<K,V> next = e.next;
            Object k;
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k)))) {
                modCount++;
                size--;
                if (prev == e)
                    table[i] = next;
                else
                    prev.next = next;
                e.recordRemoval(this);
                return e;
            }
            prev = e;
            e = next;
        }

        return e;
    }

    // 删除“键值对”
    final Entry<K,V> removeMapping(Object o) {
        if (!(o instanceof Map.Entry))
            return null;

        Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
        Object key = entry.getKey();
        int hash = (key == null) ? 0 : hash(key.hashCode());
        int i = indexFor(hash, table.length);
        Entry<K,V> prev = table[i];
        Entry<K,V> e = prev;

        // 删除链表中的“键值对e”
        // 本质是“删除单向链表中的节点”
        while (e != null) {
            Entry<K,V> next = e.next;
            if (e.hash == hash && e.equals(entry)) {
                modCount++;
                size--;
                if (prev == e)
                    table[i] = next;
                else
                    prev.next = next;
                e.recordRemoval(this);
                return e;
            }
            prev = e;
```

```
            e = next;
        }

        return e;
    }

    // 清空HashMap，将所有的元素设为null
    public void clear() {
        modCount++;
        Entry[] tab = table;
        for (int i = 0; i < tab.length; i++)
            tab[i] = null;
        size = 0;
    }

    // 是否包含“值为value”的元素
    public boolean containsValue(Object value) {
    // 若“value为null”，则调用containsNullValue()查找
    if (value == null)
            return containsNullValue();

    // 若“value不为null”，则查找HashMap中是否有值为value的节点。
    Entry[] tab = table;
        for (int i = 0; i < tab.length ; i++)
            for (Entry e = tab[i] ; e != null ; e = e.next)
                if (value.equals(e.value))
                    return true;
    return false;
    }

    // 是否包含null值
    private boolean containsNullValue() {
    Entry[] tab = table;
        for (int i = 0; i < tab.length ; i++)
            for (Entry e = tab[i] ; e != null ; e = e.next)
                if (e.value == null)
                    return true;
    return false;
    }

    // 克隆一个HashMap，并返回Object对象
    public Object clone() {
        HashMap<K,V> result = null;
        try {
            result = (HashMap<K,V>)super.clone();
        } catch (CloneNotSupportedException e) {
            // assert false;
        }
```

```
        result.table = new Entry[table.length];
        result.entrySet = null;
        result.modCount = 0;
        result.size = 0;
        result.init();
        // 调用putAllForCreate()将全部元素添加到HashMap中
        result.putAllForCreate(this);

        return result;
    }

    // Entry是单向链表。
    // 它是 “HashMap链式存储法”对应的链表。
    // 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(
Object o), hashCode()这些函数
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        // 指向下一个节点
        Entry<K,V> next;
        final int hash;

        // 构造函数。
        // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
        Entry(int h, K k, V v, Entry<K,V> n) {
            value = v;
            next = n;
            key = k;
            hash = h;
        }

        public final K getKey() {
            return key;
        }

        public final V getValue() {
            return value;
        }

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

        // 判断两个Entry是否相等
        // 若两个Entry的“key”和“value”都相等，则返回true。
        // 否则，返回false
```

```
        public final boolean equals(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry e = (Map.Entry)o;
            Object k1 = getKey();
            Object k2 = e.getKey();
            if (k1 == k2 || (k1 != null && k1.equals(k2))) {
                Object v1 = getValue();
                Object v2 = e.getValue();
                if (v1 == v2 || (v1 != null && v1.equals(v2)))
                    return true;
            }
            return false;
        }

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

        public final String toString() {
            return getKey() + "=" + getValue();
        }

        // 当向HashMap中添加元素时，绘调用recordAccess()。
        // 这里不做任何处理
        void recordAccess(HashMap<K,V> m) {
        }

        // 当从HashMap中删除元素时，绘调用recordRemoval()。
        // 这里不做任何处理
        void recordRemoval(HashMap<K,V> m) {
        }
    }

    // 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
    void addEntry(int hash, K key, V value, int bucketIndex) {
        // 保存“bucketIndex”位置的值到“e”中
        Entry<K,V> e = table[bucketIndex];
        // 设置“bucketIndex”位置的元素为“新Entry”，
        // 设置“e”为“新Entry的下一个节点”
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
        if (size++ >= threshold)
            resize(2 * table.length);
    }
```

```
    // 创建Entry。将“key-value”插入指定位置。
    void createEntry(int hash, K key, V value, int bucketIndex) {
        // 保存“bucketIndex”位置的值到“e”中
        Entry<K,V> e = table[bucketIndex];
        // 设置“bucketIndex”位置的元素为“新Entry”,
        // 设置“e”为“新Entry的下一个节点”
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        size++;
    }

    // HashIterator是HashMap迭代器的抽象出来的父类，实现了公共了函数。
    // 它包含“key迭代器(KeyIterator)”、“Value迭代器(ValueIterator)”和“Entry迭代器(Entr
yIterator)”3个子类。
    private abstract class HashIterator<E> implements Iterator<E> {
        // 下一个元素
        Entry<K,V> next;
        // expectedModCount用于实现fast-fail机制。
        int expectedModCount;
        // 当前索引
        int index;
        // 当前元素
        Entry<K,V> current;

        HashIterator() {
            expectedModCount = modCount;
            if (size > 0) { // advance to first entry
                Entry[] t = table;
                // 将next指向table中第一个不为null的元素。
                // 这里利用了index的初始值为0，从0开始依次向后遍历，直到找到不为null的元
素就退出循环。
                while (index < t.length && (next = t[index++]) == null)
                    ;
            }
        }

        public final boolean hasNext() {
            return next != null;
        }

        // 获取下一个元素
        final Entry<K,V> nextEntry() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            Entry<K,V> e = next;
            if (e == null)
                throw new NoSuchElementException();

            // 注意！！！
```

```
            // 一个Entry就是一个单向链表
            // 若该Entry的下一个节点不为空，就将next指向下一个节点;
            // 否则，将next指向下一个链表(也是下一个Entry)的不为null的节点。
            if ((next = e.next) == null) {
                Entry[] t = table;
                while (index < t.length && (next = t[index++]) == null)
                    ;
            }
            current = e;
            return e;
        }

        // 删除当前元素
        public void remove() {
            if (current == null)
                throw new IllegalStateException();
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            Object k = current.key;
            current = null;
            HashMap.this.removeEntryForKey(k);
            expectedModCount = modCount;
        }

    }

    // value的迭代器
    private final class ValueIterator extends HashIterator<V> {
        public V next() {
            return nextEntry().value;
        }
    }

    // key的迭代器
    private final class KeyIterator extends HashIterator<K> {
        public K next() {
            return nextEntry().getKey();
        }
    }

    // Entry的迭代器
    private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {
        public Map.Entry<K,V> next() {
            return nextEntry();
        }
    }

    // 返回一个“key迭代器”
```

```
    Iterator<K> newKeyIterator()   {
        return new KeyIterator();
    }
    // 返回一个“value迭代器”
    Iterator<V> newValueIterator()   {
        return new ValueIterator();
    }
    // 返回一个“entry迭代器”
    Iterator<Map.Entry<K,V>> newEntryIterator()   {
        return new EntryIterator();
    }

    // HashMap的Entry对应的集合
    private transient Set<Map.Entry<K,V>> entrySet = null;

    // 返回“key的集合”，实际上返回一个“KeySet对象”
    public Set<K> keySet() {
        Set<K> ks = keySet;
        return (ks != null ? ks : (keySet = new KeySet()));
    }

    // Key对应的集合
    // KeySet继承于AbstractSet，说明该集合中没有重复的Key。
    private final class KeySet extends AbstractSet<K> {
        public Iterator<K> iterator() {
            return newKeyIterator();
        }
        public int size() {
            return size;
        }
        public boolean contains(Object o) {
            return containsKey(o);
        }
        public boolean remove(Object o) {
            return HashMap.this.removeEntryForKey(o) != null;
        }
        public void clear() {
            HashMap.this.clear();
        }
    }

    // 返回“value集合”，实际上返回的是一个Values对象
    public Collection<V> values() {
        Collection<V> vs = values;
        return (vs != null ? vs : (values = new Values()));
    }

    // “value集合”
```

```
    // Values继承于AbstractCollection，不同于“KeySet继承于AbstractSet”，
    // Values中的元素能够重复。因为不同的key可以指向相同的value。
    private final class Values extends AbstractCollection<V> {
        public Iterator<V> iterator() {
            return newValueIterator();
        }
        public int size() {
            return size;
        }
        public boolean contains(Object o) {
            return containsValue(o);
        }
        public void clear() {
            HashMap.this.clear();
        }
    }

    // 返回“HashMap的Entry集合”
    public Set<Map.Entry<K,V>> entrySet() {
        return entrySet0();
    }

    // 返回“HashMap的Entry集合”，它实际是返回一个EntrySet对象
    private Set<Map.Entry<K,V>> entrySet0() {
        Set<Map.Entry<K,V>> es = entrySet;
        return es != null ? es : (entrySet = new EntrySet());
    }

    // EntrySet对应的集合
    // EntrySet继承于AbstractSet，说明该集合中没有重复的EntrySet。
    private final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
        public Iterator<Map.Entry<K,V>> iterator() {
            return newEntryIterator();
        }
        public boolean contains(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry<K,V> e = (Map.Entry<K,V>) o;
            Entry<K,V> candidate = getEntry(e.getKey());
            return candidate != null && candidate.equals(e);
        }
        public boolean remove(Object o) {
            return removeMapping(o) != null;
        }
        public int size() {
            return size;
        }
        public void clear() {
```

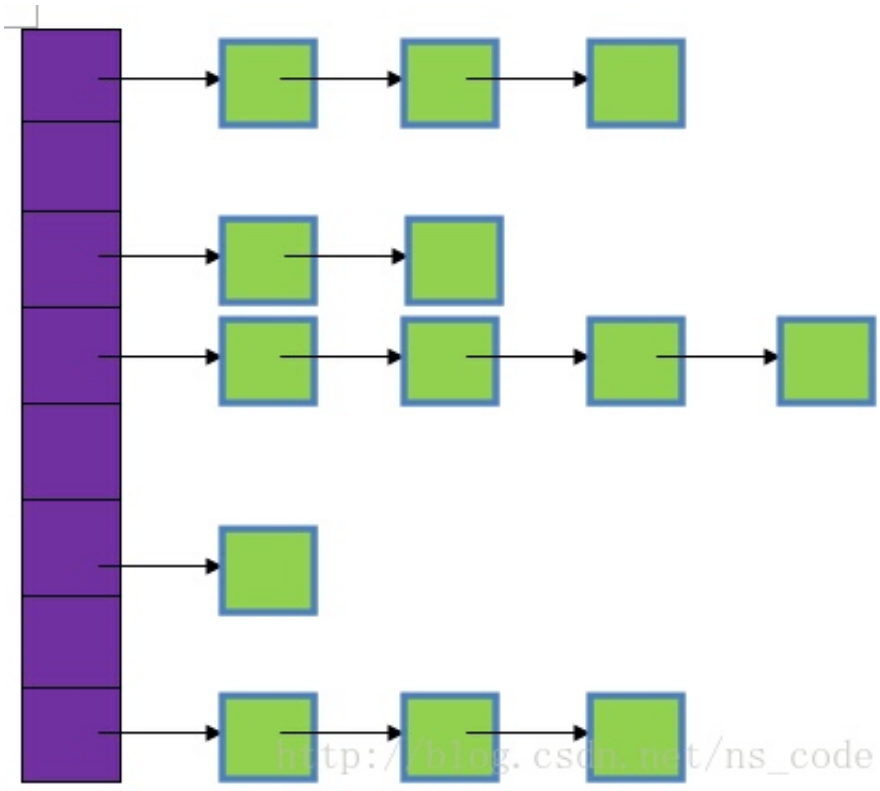
```
            HashMap.this.clear();
        }
    }

    // java.io.Serializable的写入函数
    // 将HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中
    private void writeObject(java.io.ObjectOutputStream s)
        throws IOException
    {
        Iterator<Map.Entry<K,V>> i =
            (size > 0) ? entrySet0().iterator() : null;

        // Write out the threshold, loadfactor, and any hidden stuff
        s.defaultWriteObject();

        // Write out number of buckets
        s.writeInt(table.length);

        // Write out size (number of Mappings)
        s.writeInt(size);

        // Write out keys and values (alternating)
        if (i != null) {
            while (i.hasNext()) {
            Map.Entry<K,V> e = i.next();
            s.writeObject(e.getKey());
            s.writeObject(e.getValue());
            }
        }
    }


    private static final long serialVersionUID = 362498820763181265L;

    // java.io.Serializable的读取函数：根据写入方式读出
    // 将HashMap的“总的容量，实际容量，所有的Entry”依次读出
    private void readObject(java.io.ObjectInputStream s)
         throws IOException, ClassNotFoundException
    {
        // Read in the threshold, loadfactor, and any hidden stuff
        s.defaultReadObject();

        // Read in number of buckets and allocate the bucket array;
        int numBuckets = s.readInt();
        table = new Entry[numBuckets];

        init();  // Give subclass a chance to do its thing.
```

```
        // Read in size (number of Mappings)
        int size = s.readInt();

        // Read the keys and values, and put the mappings in the HashMap
        for (int i=0; i<size; i++) {
            K key = (K) s.readObject();
            V value = (V) s.readObject();
            putForCreate(key, value);
        }
    }

    // 返回“HashMap总的容量”
    int   capacity()     { return table.length; }
    // 返回“HashMap的加载因子”
    float loadFactor()   { return loadFactor;   }
}
```

二、HashMap细节剖析

2.1 存储结构

首先要清楚HashMap的存储结构，如下图所示：



图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链

表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

2.2 链表节点的数据结构

```
// Entry是单向链表。
// 它是 “HashMap链式存储法”对应的链表。
// 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Obje
ct o), hashCode()这些函数
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    // 指向下一个节点
    Entry<K,V> next;
    final int hash;

    // 构造函数。
    // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"
    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
        return value;
    }

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

    // 判断两个Entry是否相等
    // 若两个Entry的“key”和“value”都相等，则返回true。
    // 否则，返回false
    public final boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
```

```
            if (k1 == k2 || (k1 != null && k1.equals(k2))) {
                Object v1 = getValue();
                Object v2 = e.getValue();
                if (v1 == v2 || (v1 != null && v1.equals(v2)))
                    return true;
            }
            return false;
        }

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

        public final String toString() {
            return getKey() + "=" + getValue();
        }

        // 当向HashMap中添加元素时，绘调用recordAccess()。
        // 这里不做任何处理
        void recordAccess(HashMap<K,V> m) {
        }

        // 当从HashMap中删除元素时，会调用recordRemoval()。
        // 这里不做任何处理
        void recordRemoval(HashMap<K,V> m) {
        }
    }
```

它的结构元素除了key、value、hash外，还有next，next指向下一个节点。另外，这里覆写了equals和hashCode方法来保证键值对的独一无二。

2.3 构造方法

HashMap共有四个构造方法。构造方法中提到了两个很重要的参数：初始容量和加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（从构造函数中可以看出，如果不指明，则默认为16），加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即扩容）。

下面说下加载因子，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为0.75，这是

一个比较理想的值，一般情况下我们是无需修改的。

另外，无论我们指定的容量为多少，构造方法都会将实际容量设为不小于指定容量的2的次方的一个数，且最大值不能超过2的30次方

2.4 HashMap中key和value都允许为null。

2.5 重点分析put和get

要重点分析下HashMap中用的最多的两个方法put和get。先从比较简单的get方法着手，源码如下：

```
// 获取key对应的value
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    // 获取key的hash值
    int hash = hash(key.hashCode());
    // 在“该hash值对应的链表”上查找“键值等于key”的元素
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k;
/判断key是否相同
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
没找到则返回null
    return null;
}

// 获取“key为null”的元素的值
// HashMap将“key为null”的元素存储在table[0]位置，但不一定是该链表的第一个位置！
private V getForNullKey() {
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null)
            return e.value;
    }
    return null;
}
```

首先，如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中，当然不一定是存放在头结点table[0]中。

如果key不为null，则先求的key的hash值，根据hash值找到在table中的索引，在该索引对应的单链表中查找是否有键值对的key与目标key相等，有就返回对应的value，没有则返回null。

put方法稍微复杂些，代码如下：

```
    // 将“key-value”添加到HashMap中
    public V put(K key, V value) {
        // 若“key为null”，则将该键值对添加到table[0]中。
        if (key == null)
            return putForNullKey(value);
        // 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            // 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
            }
        }

        // 若“该key”对应的键值对不存在，则将“key-value”添加到table中
        modCount++;
    //将key-value添加到table[i]处
        addEntry(hash, key, value, i);
        return null;
    }

    ```
```

如果key为null，则将其添加到table[0]对应的链表中，putForNullKey的源码如下：

// putForNullKey()的作用是将“key为null”键值对添加到table[0]位置

private V putForNullKey(V value) {

for (Entry e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;
```

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

// 如果没有存在key为null的键值对，则直接题阿见到table[0]处!

modCount++;

addEntry(0, null, value, 0);

return null;

}

```

如果key不为null，则同样先求出key的hash值，根据hash值得出在table中的索引，而后遍历对应的单链表，如果单链表中存在与目标key相等的键值对，则将新的value覆盖旧的value，比将旧的value返回，如果找不到与目标key相等的键值对，或者该单链表为空，则将该键值对插入到改单链表的头结点位置（每次新插入的节点都是放在头结点的位置），该操作是有addEntry方法实现的，它的源码如下：

```
// 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。
void addEntry(int hash, K key, V value, int bucketIndex) {
    // 保存“bucketIndex”位置的值到“e”中
    Entry<K,V> e = table[bucketIndex];
    // 设置“bucketIndex”位置的元素为“新Entry”，
    // 设置“e”为“新Entry的下一个节点”
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    // 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

注意这里倒数第三行的构造方法，将key-value键值对赋给table[bucketIndex]，并将其next指向
```

元素e，这便将key-value放到了头结点中，并将之前的头结点接在了它的后面。该方法也说明，每次put键值对的时候，总是将新的该键值对放在table[bucketIndex]处（即头结点处）。

两外注意最后两行代码，每次加入键值对时，都要判断当前已用的槽的数目是否大于等于阀值（容量*加载因子），如果大于等于，则进行扩容，将容量扩为原来容量的2倍。

2.6 关于扩容

上面我们看到了扩容的方法，resize方法，它的源码如下：

```
// 重新调整HashMap的大小，newCapacity是调整后的单位
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    // 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，
    // 然后，将“新HashMap”赋值给“旧HashMap”。
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

很明显，是新建了一个HashMap的底层数组，而后调用transfer方法，将就HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。transfer方法的源码如下：

```
// 将HashMap中的全部元素都添加到newTable中
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;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
```

```
                    e = next;
                } while (e != null);
            }
        }
    }
```

很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap的时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

2..7 containsKey方法和containsValue方法。

注意containsKey方法和containsValue方法。前者直接可以通过key的哈希值将搜索范围定位到指定索引对应的链表，而后者要对哈希数组的每个链表进行搜索。

2.8 求hash值和索引值的方法

我们重点来分析下求hash值和索引值的方法，这两个方法便是HashMap设计的最为核心的部分，二者结合能保证哈希表中的元素尽可能均匀地散列。

计算哈希值的方法如下：

```
static int hash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }
```

它只是一个数学公式，IDK这样设计对hash值的计算，自然有它的好处，至于为什么这样设计，我们这里不去追究，只要明白一点，用的位的操作使hash值的计算效率很高。

由hash值找到对应索引的方法如下：

```
static int indexFor(int h, int length) {
return h & (length-1);
}
```

这个我们要重点说下，我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除法散列法），Hashtable中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。