* 学习目标

* 能够理解Map接口概述

* Key，Value

* 映射：一对一，Map存储映射集合

* 能够画出Map的继承框架图

* Map<K,V>

* HashMap(Key不能重复)

* LinkedHashMap

* TreeMap

* Comparable

* Comparator

* CocurrentHashMap(支持并发)

* HashTable（线程安全）

* Properties

* 能够掌握Map接口常见的方法

* put，
get,size,clear,isEmpty,reomve,containsKey,containsValue,keySet,values,entrySet

* id---Person.user--cart

* 能够掌握Map子类HashMap

* Hash，Hash算法 (作为标志)，MD5，SHA -1,...hashCode

* Hash冲突或者碰撞

* HashMap底层数据结构：散列表，闪列桶

* 数组+链表+红黑树

* HashMap 底层源码

* 常量：容量初始值（1<<4），容量最大值(1<<30)，负载因子0.75f，6（树转换链表）,8,64（链表转换树）

* Node（hash（key），key，value，next） Map.Entry(getKey,getValue)

* 变量：table，size，load factor,threshold=(capacity*load factor,),entrySet

* 构造器

* put,reisize,get

---

* 回顾

* LinkedList的源码：双向链表，队列，栈

* ArrayList的源码

* 扩容

* minCapacity>elementData.length：才要进行扩容

* 数组一半（size >> 1）0100 --> 0010 (size<< 1) 0100 1000

* newArr=Arrays.copyOf(oldArr,newCapacity)

* 数组复制

* Vector源码

* 扩容机制：扩展一倍

* 线程安全

* 数据结构--树（Tree）

* 根节点，父节点，子节点，兄弟节点，叶子节点

* 节点的度，树的度，层次，树的高度或者深度，...

* 二叉树

* 遍历方式

* 先序：根左右

* 中序：左根右

* 后序：左右根

* 广度优先遍历和深度优先遍历

* 二叉查找树（BST）

* 左小根

* 右大根

* 子树也是二叉查找树

* 红黑树(Red Black Tree)

* 二叉查找树

* 平衡树

* 根黑

* 红或黑

* 叶黑

* 红子黑

* 路径下黑相等

* 从根节点到叶子节点的最长路径不能超过最短路径的2倍

* 假如一个节点到红黑树，假如不符合规则

* 变色，左旋，右旋...

* 能够理解Map接口概述

* 按<键,值>对的形式存储数据，是双列集合

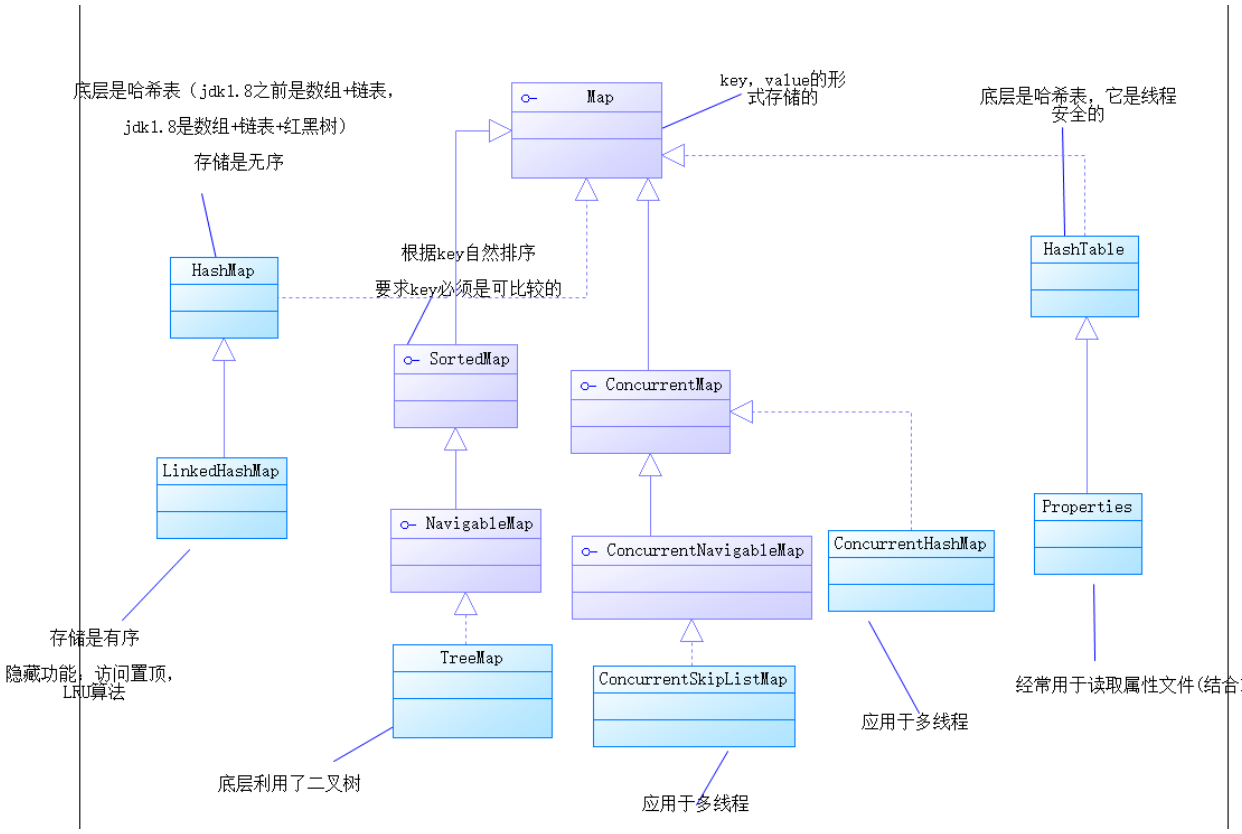
* 生活中：身份证号与个人，题号与答案，用户名与用户对象，等， 这种一一对应的关系，就叫做映射。Map就是用来存储这种映射对象的集合

* 01 --> Person(id,name)

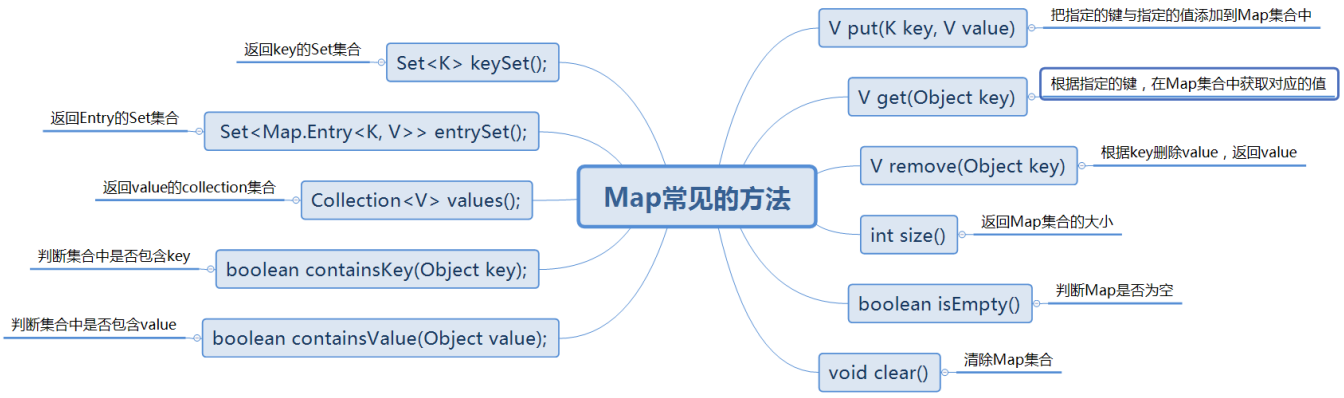
* 01 ---> Answer(id,answer)

* user ---> Cart(userid,List<Goods>)

* 能够画出Map的继承框架图



* 能够掌握Map接口常见的方法




```
public class Test1 {
    public static void main(String[] args) {
         // * 01  --> Person(id,name)
        Map<String,Person> persons=new HashMap<String,Person>();
        // put,get,remove,size,isEmpty,clear,keySet,entrySet,values,
```

```
        // containsKey,containsValue
        // 01 --- Person id=01,name=刘备
        // 02 --- Person id=02,name=关羽
        // 03 --- Person id=03,name=张飞

        persons.put("01", new Person("01","刘备"));
        persons.put("02", new Person("02","关羽"));
//      persons.put("02", new Person("02","关羽"));// key 是不能重复
//      persons.put("02", new Person("02","关羽"));
        persons.put("03", new Person("03","张飞"));
        System.out.println("总共多少人:"+persons.size());
//      System.out.println("请输入编号: ");
//      Scanner input=new Scanner(System.in);
//      String id=input.next();
        Person person=persons.get("01");
        System.out.println(person);

        Person p1 = persons.remove("01");
        System.out.println(p1.getName()+"已经被干掉...");
        System.out.println("还剩多少人:"+persons.size());

        persons.clear();
        System.out.println("还剩多少人:"+persons.size());

        persons.put("01", new Person("01","刘备"));
        persons.put("02", new Person("02","关羽"));
        persons.put("03", new Person("03","张飞"));
        // key:equals , hashCode
        if(persons.containsKey("01")) {
            System.out.println(persons.get("01").getName()+"角色已经存在");
        }else {
            System.out.println(persons.get("01").getName()+"角色已经不存在");
        }

        // HashMap : 判断是不是同一个对象: equals
        if(persons.containsValue(new Person("01","刘备"))) {
            System.out.println("刘备存在");
        }else {
            System.out.println("刘备不存在");
        }
```

```
        // 迭代这个Map
        Set<String> keys = persons.keySet();
        Iterator<String> ite = keys.iterator();
        while(ite.hasNext()) {
            String key=ite.next();
            String name=persons.get(key).getName();
            System.out.printf("%s %s",key,name);
        }
        System.out.println();
        for(String key:keys) {
            System.out.printf("%s %s",key,persons.get(key).getName());
        }
        System.out.println();
        Collection<Person> values = persons.values();
        for(Person p:values) {
            System.out.println(p);
        }

        Set<Entry<String, Person>> entries = persons.entrySet();
        Iterator<Entry<String, Person>> ite1 = entries.iterator();
        while(ite1.hasNext()) {
            Entry<String, Person> entry = ite1.next();
            String key=entry.getKey();
            String name=entry.getValue().getName();
            System.out.printf("%s %s",key,name);
        }
        System.out.println();
        for(Entry<String,Person> entry:entries) {
            String key=entry.getKey();
            String name=entry.getValue().getName();
            System.out.printf("%s %s",key,name);
        }
         // * 01 ---> Answer(id,answer)
         // * user ---> Cart(userid,List<Goods>)
    }
}

测试结果:
总共多少人:3
Person [id=01, name=刘备]
```

```
刘备已经被干掉...
还剩多少人:2
还剩多少人:0
刘备角色已经存在
刘备存在
01 刘备02 关羽03 张飞
01 刘备02 关羽03 张飞
Person [id=01, name=刘备]
Person [id=02, name=关羽]
Person [id=03, name=张飞]
01 刘备02 关羽03 张飞
01 刘备02 关羽03 张飞
```

* 购物的车案例

```
public static void main(String[] args) {
        User u1=new User(1001,"项羽","123");
        User u2=new User(1002,"刘邦","123");
        Cart cart1=new Cart();
        cart1.setUserId(u1.getId());
        Goods goods1=new Goods(10001,"万宝路",23);
        Goods goods2=new Goods(10002,"利群",23);
        Goods goods3=new Goods(10003,"经典1906",23);
        Goods goods4=new Goods(10004,"软中",23);
        Goods goods5=new Goods(10005,"九五至尊",23);
        List<Goods> list1=new ArrayList<Goods>();
        list1.add(goods1);
        list1.add(goods2);
        list1.add(goods3);
        list1.add(goods4);
        list1.add(goods5);
        cart1.setCartList(list1);
        Map<User,Cart> carts=new HashMap<User,Cart>();
        carts.put(u1, cart1);

        Cart cart2=new Cart();
        cart2.setUserId(u2.getId());
```

```
        List<Goods> list2=new ArrayList<Goods>();
        list2.add(goods1);
        list2.add(goods2);
        list2.add(goods3);
        list2.add(goods4);
        list2.add(goods5);
        cart2.setCartList(list2);
        carts.put(u2, cart2);

        // 登录之后查看自己购物
        Scanner input=new Scanner(System.in);
        System.out.println("请输入用户名:");
        String userName=input.next();
        System.out.println("请输入密码:");
        String password=input.next();
        // 数据库的查询
        Set<User> users = carts.keySet();
        User user=null;
        for(User u:users) {
            if(u.getName().equals(userName) && u.getPassword().equals(password)
                user=u;
                break;
            }
        }
        if(user!=null) {
            // 登陆成功
            Cart cart = carts.get(user);
            System.out.println(cart);
        }
    }

结果:
请输入用户名:
项羽
请输入密码:
123
Cart [userId=1001, cartList=[Goods [id=10001, name=万宝路, price=23.0], Goods [i

```

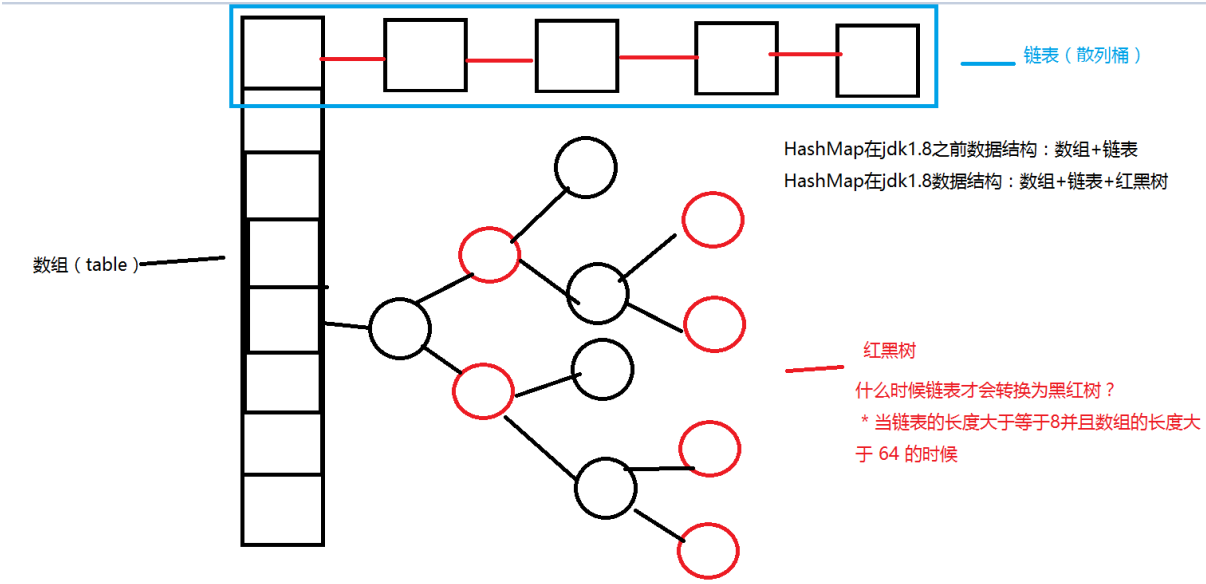
* 能够掌握Map子类HashMap

* Hash 碰撞（也叫做Hash冲突）

    * Hash算法可以出计算几乎出独一无二的hashCode，如果出现了重复的hashCode，就称作碰撞或者冲突。

* HashMap的数据结构

    * HashMap底层用散列表（哈希表）



* HashMap 的源码分析

```
* 常量
 * static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
    * map默认初始化容量：16
 * static final int MAXIMUM_CAPACITY = 1 << 30;
    * map最大容量
 * static final float DEFAULT_LOAD_FACTOR = 0.75f;
    * 作为map扩展因子：假如map的size超过：CAPACITY*DEFAULT_LOAD_FACTOR，就进行扩容
    * 例如：16*0.75f=12，默认map超过12，就进行扩容
 * static final int TREEIFY_THRESHOLD = 8;
    * 链表中数据的临界值，如果达到8并且数组大于64则转换为树
 * static final int UNTREEIFY_THRESHOLD = 6;
    * 如果链表的数据小于6，则从树转换为链表
```

```
 * static final int MIN_TREEIFY_CAPACITY = 64;
    * 如果数组的size大于64，则把链表进行转化为树

* 数组存放的节点
    * 在做数组的节点上，额外实现获取K，V的值
    interface Map.Entry<K,V>{
        K getKey();
        V getValue();
        V setValue(V value);
        hashCode()
        equals()
    }
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;// key的hash值
        final K key;
        V value;
        Node<K,V> next;// 单向链表：指向下个节点

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

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

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

        public final boolean equals(Object o) {
```

```
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }

* 成员变量
  * Node<K,V>[] table
    * 数组：建议长度是2的平方
  * int size
    * map的大小
  * Set<Map.Entry<K,V>> entrySet
    * Map.Entry<K,V> 的Set集合
  * int threshold;
    * size 超过这个值，就会进行扩容
    * next value:threshold=capacity * load factor

* 构造器
   * HashMap()
     * 构造一个空的HashMap，初始容量为16，负载因子为0.75
     public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
     }
   * HashMap(int initialCapacity)
     * 构造一个初始容量为initialCapacity，负载因子为0.75的空的HashMap
     public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
      }
   * HashMap(int initialCapacity, float loadFactor)
     * 构造一个空的初始容量为initialCapacity，负载因子为loadFactor的HashMap
      public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0) // 假如initialCapacity为0，抛非法参数异常
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
```

```
93          if (initialCapacity > MAXIMUM_CAPACITY) // 假如initialCapacity大于定义最
94              initialCapacity = MAXIMUM_CAPACITY; // 赋值为MAXIMUM_CAPACITY
95          if (loadFactor <= 0 || Float.isNaN(loadFactor))
96              // 假如loadFactor 小于等于0，或者不是一个数字，抛非法参数异常
97              throw new IllegalArgumentException("Illegal load factor: " +
98                                                 loadFactor);
99          this.loadFactor = loadFactor;// 负载因子赋值
100         this.threshold = tableSizeFor(initialCapacity);//阀值赋值，把初始化容量转
101     }
102     * tableSizeFor：容量转换为2的平方
103      static final int tableSizeFor(int cap) {
104         int n = cap - 1;
105         n |= n >>> 1;
106         n |= n >>> 2;
107         n |= n >>> 4;
108         n |= n >>> 8;
109         n |= n >>> 16;
110         return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1
111     }
112   * 测试了解
113   public static final int MAXIMUM_CAPACITY = 1 << 30;
114     public static void main(String[] args) {
115         int cap=6;
116         cap=tableSizeFor(cap);
117         System.out.println(cap);
118     }
119
120     public  static final int tableSizeFor(int cap) {
121         int n = cap - 1;// 0101
122         n |= n >>> 1;// 0010 | 0101 = 0111
123         System.out.println(Integer.toBinaryString(n));
124         n |= n >>> 2;// 0001 | 0111 = 0111
125         System.out.println(Integer.toBinaryString(n));
126         n |= n >>> 4;// 0000 | 0111 = 0111
127         System.out.println(Integer.toBinaryString(n));
128         n |= n >>> 8;
129         System.out.println(Integer.toBinaryString(n));
130         n |= n >>> 16;
131         System.out.println(Integer.toBinaryString(n));
132         return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1
```

```
133     }
134   * 结果：
135     111
136     111
137     111
138     111
139     111
140     8
141 * hash
142     * int hash(Object key)
143     static final int hash(Object key) {
144         int h;
145         // key的hash值的计算是通过hashCode()的高16位异或低16位实现的
146         return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
147     }
148     * 直接返回key.hashCode()不行吗？为什么还要异或
149       * 当数组table的length比较小的时候，保证考虑到高低Bit都参与到Hash的计算中，
150         同时不会有太大的开销
151     * 例子
152      原 来 的 hashCode : 1111 1111 1111 1111 0100 1100 0000 1010
153      移位后的hashCode:   0000 0000 0000 0000 1111 1111 1111 1111
154      进行异或运算 结果:   1111 1111 1111 1111 1011 0011 1111 0101
155
156 * V put(K key, V value)
157   * return putVal(hash(key), key, value, false, true);
158   * 参数：onlyIfAbsent：假如为true，不改变原来的值
159         evict：假如为fasle，代表table为表处于创建模式
160   * V putVal(int hash, K key, V value, boolean onlyIfAbsent,
161                     boolean evict)
162     final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
163                     boolean evict) {
164         Node<K,V>[] tab; // map的数组
165         Node<K,V> p; // 数组的节点
166         int n;// 数组的长度
167         int i;// Node 添加到数组的下标
168         // 如果table为空或者长度为0，则resize()
169         // resize():后面讲解
170         if ((tab = table) == null || (n = tab.length) == 0)
171             n = (tab = resize()).length;
172         // 确定插入table的位置，算法是(n - 1) & hash，在n为2的幂时，相当于hash % n
```

```
        // 找到key值对应的槽并且是第一个，直接加入
      if ((p = tab[i = (n - 1) & hash]) == null)
          // 假如为null意味着数组下标所指的元素为没有存放值，直接构建新节点存进去
            tab[i] = newNode(hash, key, value, null);
        else {
            // 假如不null意味着下标所指的元素有存放值
            Node<K,V> e;
            K k;
            //数组已经存在元素， p就是i下已经存在元素
            //判断数组已经存在元素与需要添加的元素是否是同一对象
                //判断p的hash值与需要添加到hashmap的key的hash值对象对比
                //假如p的hash值与需要添加到hashmap的key的hash值相等
                //再判断p的key与需要添加到hashmap的key是否相等或者equals
                //假如都相等存放到e中，代表就是同一对象，不需要添加的（
                //(后面需要判断e是否null,假如e不为null，看看需不需要覆盖它值）
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            // 假如p是树形节点，存储到红黑树中
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
            // 到了这里，既不是同一对象，也不是树形节点，存放到链表中
                for (int binCount = 0; ; ++binCount) {
                    // 一步步走，看看p的下一个节点是否为null
                    // 假如为p.next=null
                    if ((e = p.next) == null) {
                      // 假如为p.next=null，意味着，需要添加到hashmap的Node可以添
                        p.next = newNode(hash, key, value, null);
                      // 判断链表的长度是否超过8，假如超过8，调用treeifyBin
                      // treeifyBin方法会去判断数组长度是否大于64，假如大于64，把链
                      // 否则重新resize(),重新设置表格的大小
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    // 假如为e=p.next不为null,p.next 是否同一对象（hash，hashcode
                    // 假如是同一对象，直接跳出，同时e也被赋值
                    // 假如不是同一对象，p更新为下一个值：p=e;
                    if (e.hash == hash &&
```

```
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            // 假如e不等于null，看看要不要覆盖它的值，默认是覆盖
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        // threshold:这个变量在resize的时候，进行赋值:capacity*factor
        // 假如这个size添加后大于threshold
        // 进行数组进行重新resize
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

备注：hash 碰撞(冲突)发生的几种情况：

 * 两节点key 值相同（hash值一定相同），导致碰撞；
 * 两节点key 值不同，由于 hash 函数的局限性导致hash 值相同，碰撞；
 * 两节点key 值不同，hash 值不同，但 hash 值对数组长度取模后相同，碰撞；

```
* resize
  final Node<K,V>[] resize() {
        // 保存当前table
        Node<K,V>[] oldTab = table;
        // 保存当前table的容量：假如当前table为null返回0，假如不null，返回table的长
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        // 保存阀值
        int oldThr = threshold;
        int newCap = 0;// 新容量
        int newThr = 0;// 新的阀值
        // oldCap大于 0 代表原来的 table 表非空
        if (oldCap > 0) {
```

```
253             // 若旧table容量已超过最大容量，更新阈值为Integer.MAX_VALUE（最大整形值）
254             // 这样以后就不会自动扩容了。
255             if (oldCap >= MAXIMUM_CAPACITY) {
256                 threshold = Integer.MAX_VALUE;
257                 return oldTab;
258             }
259             // 容量翻倍，使用左移，效率更高
260             else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
261                      oldCap >= DEFAULT_INITIAL_CAPACITY)
262                 // 假如新容量小于最大值并且老的容量大于默认初始化容量
263                 // 阀值翻倍
264                 newThr = oldThr << 1; // double threshold
265         }
266         // 当table没初始化时，threshold持有初始容量。(初始化：threshold = tableSiz
267         else if (oldThr > 0) // initial capacity was placed in threshold
268             newCap = oldThr;
269         // 假如table为空并且老的阀值等于0
270         // 新的容量的值等于默认容量（16）
271         // 新的阀值等于负载因子*默认容量（0.75f*16=12）
272         else {               // zero initial threshold signifies using defaults
273             newCap = DEFAULT_INITIAL_CAPACITY;
274             newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
275         }
276         // 新阈值为0
277         if (newThr == 0) {
278             float ft = (float)newCap * loadFactor;
279             newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
280                       (int)ft : Integer.MAX_VALUE);
281         }
282         // 重新给threshold赋值
283         threshold赋值 = newThr;
284         // 初始化table
285         Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
286         table = newTab;
287         // 假如老table有值，放到到新table中
288         if (oldTab != null) {
289             for (int j = 0; j < oldCap; ++j) {
290                 Node<K,V> e;
291                 if ((e = oldTab[j]) != null) {
292                     // 逐步把老table值置null，并保存到e中
```

```
                    oldTab[j] = null;
                    // 判断数组j位置是不是只有一个元素
                    // 假如数组j位置只有一个元素
                    if (e.next == null)
                        // 放到新的数组中
                        newTab[e.hash & (newCap - 1)] = e;
                    // 假如数组j位置不只有一个元素，并且是树形节点，添加到红黑树中
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    // 假如数组j位置不只有一个元素，并且不是树节点，添加到链表中
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
```

```
                }
            }
        }
        return newTab;
    }
* get
  public V get(Object key) {
        Node<K,V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }
  * getNode(hash(key), key)

   final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab;
        Node<K,V> first;
        Node<K,V>e;
        int n;
        K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }


* 总结
  * HashMap在1.8之前使用时链表+数组，在1.8的时候变链表+数组+红黑树
     * 当链表的长度大于8并且数组的长度大于64，链表转换为红黑树
```

```
373         * 当链表的长度大于8并且数组的长度小于=64，数组resize
374         * 当链表的长度小于等于6的时候，红黑树转换成链表
375     * HashMap默认初始化容量: 16
376     * HashMap默认的加载因子: 0.75
377     * 当哈希表数组元素的数量大于（threshold=数组的长度 * 加载因子）时，数组扩容
378     * hashMap默认扩容: 2倍 （ capacity=capacity << 1 ）
379     * 在定义HashMap时，指定初始化容量，系统会自动调整为2的幂次方(tableSizeFor)
380     * 调整为2的幂次方后,可以快速的计算出数组的下标
381
```