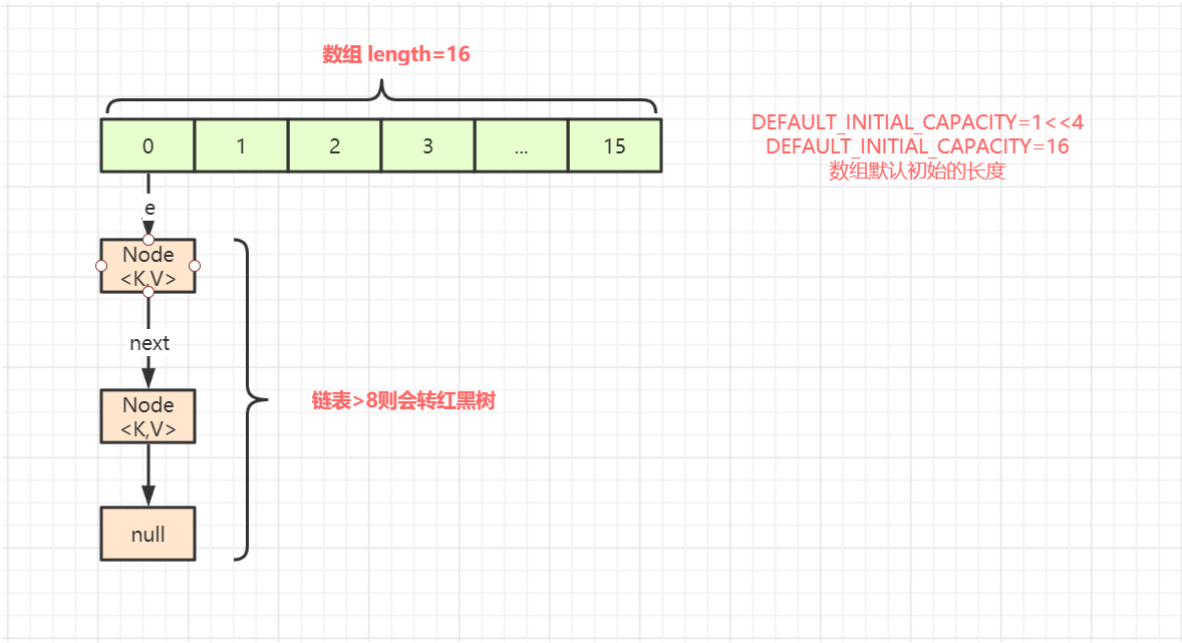
HashMap理解分析

1、底层数据结构是什么（JDK1.8）？



1、大体结构：数组+链表+红黑树。
2、数组的初始默认长度是16。数组最大的长度为1<<30=(2^30)。
3、数组每个槽位中链表的长度如果大于8则会转成红黑树结构。
4、我们put成功后的额数据会封装成Node(Entry)保存在数组曹中。
为什么1？2？3？
总结一句话：都是1.8之后HashMap性能优化的行为。从这点出发理解不会有错，细节分细会在后面给大家解释。

2、哪些常见成员变量，各是什么含义？

1、初始数组长度(The default initial capacity - MUST be a power of two.)。

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

2、数组扩容最大的极限长度(MUST be a power of two <= 1<<30)。

```
static final int MAXIMUM_CAPACITY = 1 << 30;
```

3、数组容量负载因子(The load factor used when none specified in constructor.)。

```
tatic final float DEFAULT_LOAD_FACTOR = 0.75f;
final float loadFactor;//负载因子，默认胃上面定义的常量DEFAULT_LOAD_FACTOR值0.75f。
```

4、数组容量

```
int threshold; //当前数组可以容纳的Node(Entry)数量，超过这个将会出发扩容。至于还不清楚Node对象接口在后面的内容会给大家解释。
```

5、链表实际保存的数据结构

```
transient Node<K,V>[] table;
```

6、Map的元素长度，简单好理解。

```
transient int size;
```

```
7、Map数据结构变更的次数。基本可以理解为没有put成功后都会modCount++，但是如果put是个覆盖操作则不会触发modCount++。
transient int modCount;

8、threshold、loadFactor、now_capacity(当前数组长度)之间的关系。
threshold = now_capacity*loadFactor;
拿数组初始化的情景做分析，这个时候的数组能存放多少Node元素呢,如何计算？
threshold = 16*0.75f = 12。
也就是说基于初始化的场景，当前hashMap要是存放的Node总数量超过12个则会触发扩容。

9、static  final  int TREEIFY_THRESHOLD = 8;
单个槽位链表最大的长度为8，超过8则会转红黑树。
```

3、如何对key进行hash算法？从源码看看

3.1、java.util.HashMap#put 611行源码

```
public V put(K key, V value) {
   return putVal(hash(key), key, value, false, true);
}
```

3.2、java.util.HashMap#hash 337行源码

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

重点分析：(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16)

当key==null return 0; //这也正好解释了为什么HashMap为什么允许key为空的原因。

当key!=null return (h = key.hashCode()) ^ (h >>> 16);

其实(h = key.hashCode()) ^ (h >>> 16)相当于key.hashCode() ^ (key.hashCode() >>> 16)方便理解

最终 hash = key.hashCode() ^ (key.hashCode() >>> 16);

举个具体的例子：

当key="abc"时计算当前key的hash值?

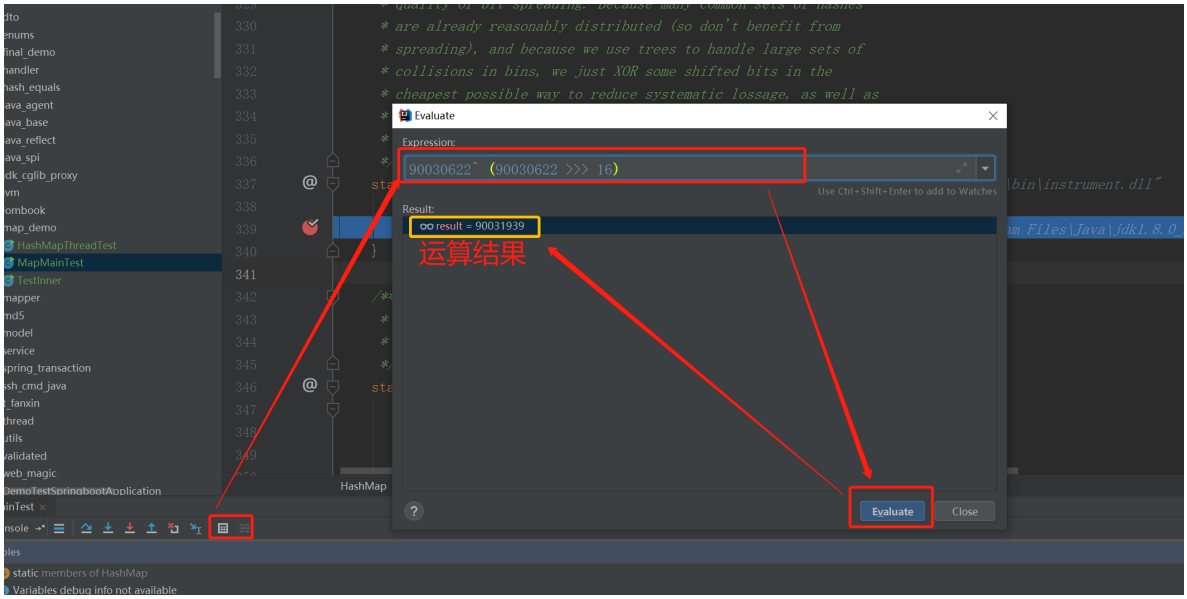
key.hashCode() = 96354；

```
"abc".hashCode()=96354
```

变成： 96354^ (96354>>>16) =96355

所以key="abc"最终的hash=96355。

3.3 计算技巧



一个小技巧：如果大家后续涉及到计算机底层相关的计算，按照上图方式个人觉得是全球最好的计算器。哈哈

纯属开玩笑，个人开发习惯。

注意：这个必须自己本地任意打个断点，就可以打开这个执行计算器了。

4、判断key重复的依据是什么?

4.1 key的相同判断机制。

同时根据hash和equals。从源码看看go ...

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        //下面这行代码就是put时候判断key时候相同。相同就覆盖。用到了hash和equals这也就是为什么人们常说hash和equals一定要重写的原因。否则这里判断会出问题。
        if (
            p.hash == hash //判断hash是否相同
            && //并且
            ((k = p.key) == key || (key != null && key.equals(k)))//key的equals相同
            )
            e = p; //覆盖原来的值
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
```

```
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;//数据结构变化计数器
    if (++size > threshold)
        resize();//扩容操作
    afterNodeInsertion(evict);
    return null;
}
```

4.1 通俗易懂的Hash和Equals

4.1.1 hash和equals到底什么关系?

```
答：从定义上来说他们没有任何关系，就相当于我们平时自己写个类里面定义了两个方法test1()和
test2().大家可以想一下，从单纯的这两个方法来说他们自己是自己，没有任何的关系。但是什么时候
test1()和test2()会有关系那就看咋们在逻辑上会不会都涉及他们关联应用场景。
比如说：
test1(){
   test2();
}
比如说：
test2(){
   test1();
}
比如说：
if(test1()&&test2()){ //假设返回的是boolean值
   //to do something else
}
所以hash和equals到底有没有关系，就看系统逻辑上需要怎么用他们。
比如说HashMap判断key是否存在相同的，这里就同时用到了hash和equals方法了。
if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
```

4.1.1-1 那么从细节上来说为什么作为HashMap的可以一定要重写hash和equals又或者不一定要重写呢?

1、如果HashMap保存的key instanceof String就不需要去关心hashCode和equals重写的问题了。

因为String默认帮我们做了重写工作。

java.lang.String#equals

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
```

```
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

java.lang.String#hashCode

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

从上面的源码可以看出：只要String的字面量值相同那么他们的equals和hashCode就一定会想等。

2、如果key是自定义的引用类型，对于HashMap的put适用环境就必须要重写equals和hashCode两个方法。

因为如果不重写的话那个equals和hashCode就是默认父类中的与之对应的方式，从java基础类的继承关系可以看出。

java.util.Objects#hashCode

```
//父类中是通过本地方法获取一个整型的hashCode值。这也是对象头里维护的那个值hashCode.
public native int hashCode();
public static int hashCode(Object o) {
    return o != null ? o.hashCode() : 0;//调用本地方法栈
}
```

从源码可以知道，如果你不从写hashCode方法，new出来的两个对象他们属性再怎么相同他们的hashCode就一定是不会相同的。因为这个时候的hashCode是本地方法回去的对象头中的内存地址。所以此时如果你要让hashCode 相同就必须自定义去重写hashCode()。这样他们才有可能会相同，记住是有可能相同哪怕你重写了hashCode，最终想不相同看你是怎么去自定义重写的。

情况一：只对对象的若干个成员属性参与equals和hashCode计算。

```
@Data
@AllArgsConstructor
public class HashEqualsTest {
    private int age;
    private String name;
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        HahEqualsTest that = (HahEqualsTest) o;
        return name.equals(that.name);//只有name参与比较
    }
    @Override//只对name重写计算hashCode
    public int hashCode() {
        return Objects.hash(name);//只有name参与hashCode计算
    }

    public static void main(String[] args) {
        //年龄相同 名字不同
        HashEqualsTest dsz1 = new HashEqualsTest(1, "dsz1");
        HashEqualsTest dsz2 = new HashEqualsTest(1, "dsz2");
        System.out.println("dsz1#hashCode=" + dsz1.hashCode());
        System.out.println("dsz2#hashCode=" + dsz2.hashCode());
    }
}
```

结果输出：

```
======名字不同======
dsz1#hashCode=3093477
dsz2#hashCode=3093478
dsz1.equals(dsz2)=false
======名字相同======
dsz3#hashCode=3093477
dsz4#hashCode=3093477
dsz3.equals(dsz4)=true
```

这个时候就是虽然你重写了hashCode但是你的结果还是不相等的。但是反过来如果两个对象的name字面量相同，那这两个对象hashCode就相同了。同理equals也是一样的。

情况二：对象的所有成员属性都参与equals和hash计算。

```
@Data
@AllArgsConstructor
public class HashEqualsTest {
    private int age;
    private String name;
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        HashEqualsTest that = (HashEqualsTest) o;
        return age == that.age && name.equals(that.name);//age和name都必须相同才为true。
    }
    @Override
    public int hashCode() {
        return Objects.hash(age, name);//age和name同时参与hashCode计算。
    }
```

```
    public static void main(String[] args) {
        System.out.println("======年龄相同 名字不同======");
        HashEqualsTest dsz1 = new HashEqualsTest(1, "dsz1");
        HashEqualsTest dsz2 = new HashEqualsTest(1, "dsz2");
        System.out.println("dsz1#hashCode=" + dsz1.hashCode());
        System.out.println("dsz2#hashCode=" + dsz2.hashCode());
        System.out.println("dsz1.equals(dsz2)=" + dsz1.equals(dsz2));
        System.out.println("======年龄相同 名字相同======");
        HashEqualsTest dsz3 = new HashEqualsTest(1, "dsz1");
        HashEqualsTest dsz4 = new HashEqualsTest(1, "dsz1");
        System.out.println("dsz3#hashCode=" + dsz3.hashCode());
        System.out.println("dsz4#hashCode=" + dsz4.hashCode());
        System.out.println("dsz3.equals(dsz4)=" + dsz3.equals(dsz4));
    }
}
结果输出：
======年龄相同 名字不同======
dsz1#hashCode=3094438
dsz2#hashCode=3094439
dsz1.equals(dsz2)=false
======年龄相同 名字相同======
dsz3#hashCode=3094438
dsz4#hashCode=3094438
dsz3.equals(dsz4)=true
这个时候就必须要age和name的字面量都要相同两者才会相同。
```

总结：如果自定义的应用类型作为key就得看具体hashCode和equals实现方式。从而才能判断出这两个对象在什么时候equals==true和hashCode是否相同。

所以从上面这么多的分析可知：equals和hashCode要不要同时重写就要看你的逻辑上的判定关系而定。

明白了上面的分析下面的两个问题就好回答了。

4.1.2 什么时候需要只重写hash?

答：取决于你逻辑要怎么用。完全自定义。

比如说我只希望上面的HashEqualsTest对象name相同，逻辑上我就判断他们的hash相同。那么这个时候跟equals半毛钱关系都没有，可重写也可以不用重写equals方法。

4.1.3 什么时候需要只重写equals?

答：取决于你逻辑要怎么用。完全自定义。

比如说我只希望上面的HashEqualsTest对象name相同，逻辑上我就判断他们的equals相同。那么这个时候跟hashCode半毛钱关系都没有，可重写也可以不用重写hashCode方法。

4.1.4 HashSet是如何判断两个元素是相同的?

首先看下HashSet的add()源码。

java.util.HashSet#add

```
private transient HashMap<E,Object> map;
private static final Object PRESENT = new Object();//所有元素公用一个全局value，节省内存。
public HashSet() {
        map = new HashMap<>();
}
//HashSet添加元素
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

从上上面的源码可以看出，原来HashSet底层是个HashMap。这会就清楚HashSet是如何判断集合中的元素的唯一性了。

所以HashSet如果存储String类型的数据会自动去重操作的。

如果hashSet存储的是自定义的引用类型数据，希望能达到去重的效果的话就必须要重写equals和hashCode两个方法。原因是HashMap上面已经解释过了。

5、HashMap是如何计算数据存放在数组中索引的位置的?

又回到put源码参观：

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        //下面这行代码就是计算当前我们保存的数据应该保存数组中的那个索引的位置。
        //tab[i = (n - 1) & hash]具体的计算我们放在代码块外去解释。往下翻滚。
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
```

```
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }
```

这里的hash就是我们上面第3个问题：获取key的hash值传到了这里面。上面的key="abc"最终的

hash=96355；

按照：tab[i = (n - 1) & hash]这个算法，带入参数变成：

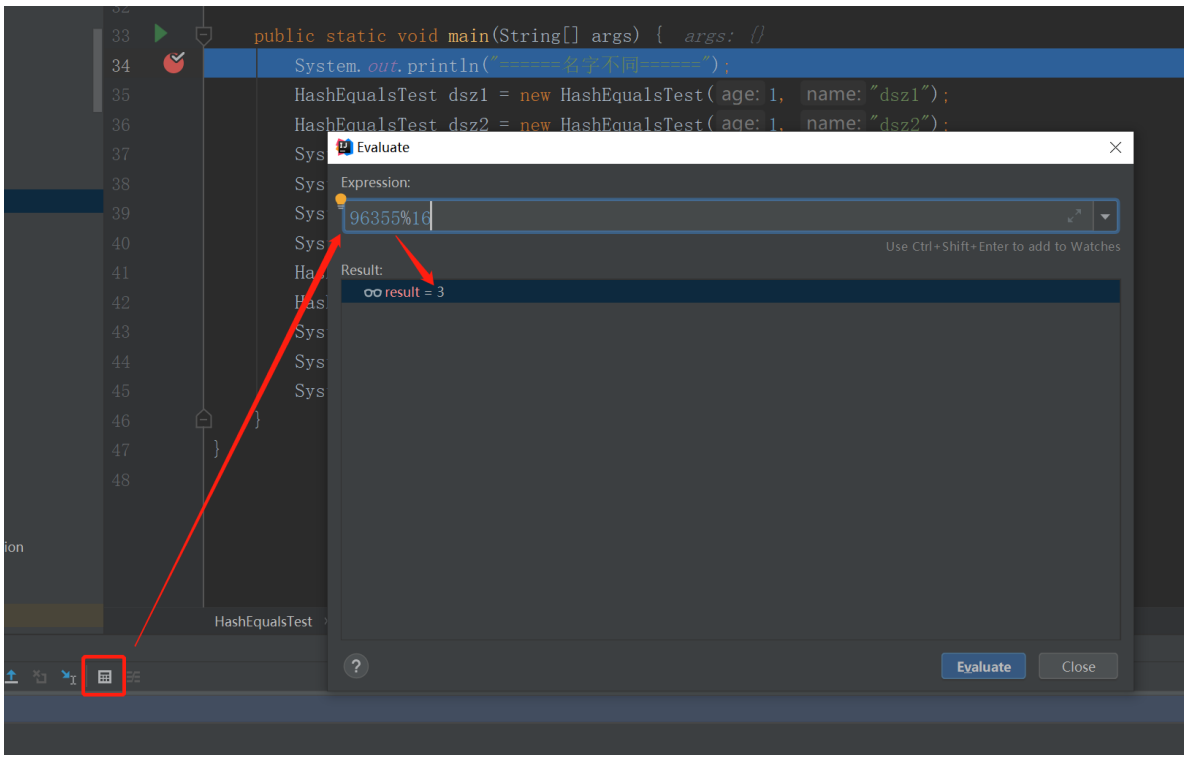
i = (n - 1) & 96355 那么括号中的n?是什么呢，很简单n其实就是数组+链表中的数组的length=16，以初始的时候为demo的话，不考虑后续的扩容。

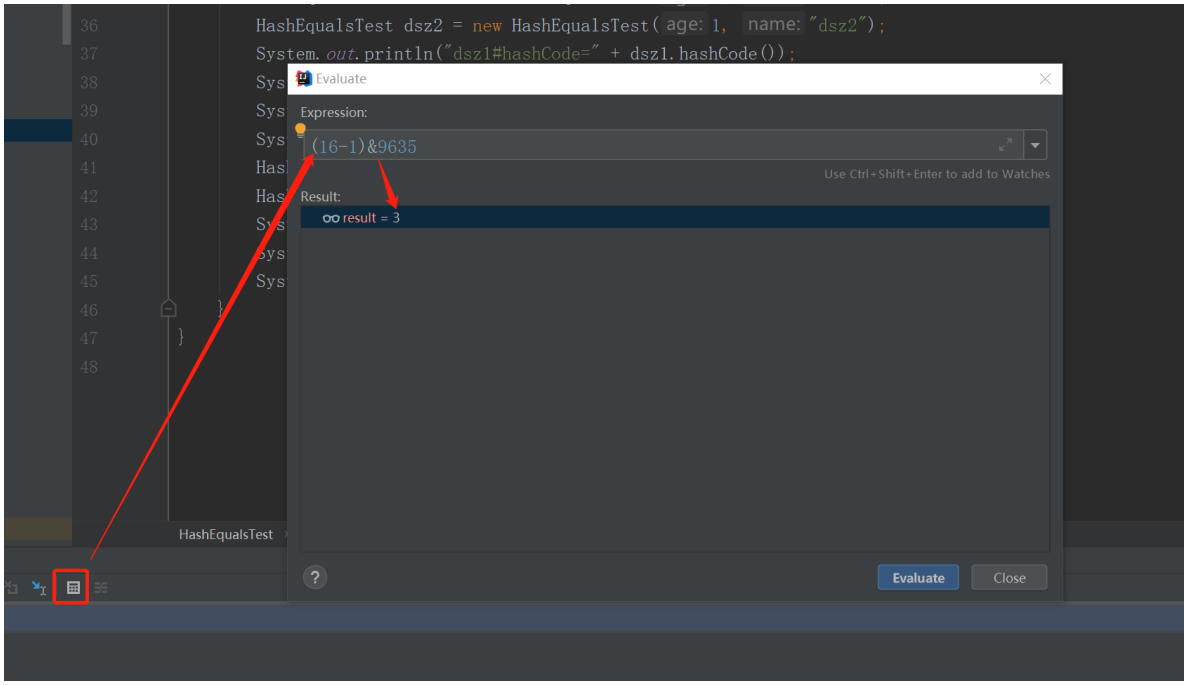
那这么一来：i = (16 - 1) & 96355 其实这里也相当于 i = 96355%16(对16的取模算法)。

所以这里也是为什么数组的长度必须是要2的幂次方的原因，因为当2的幂次方才可以造成

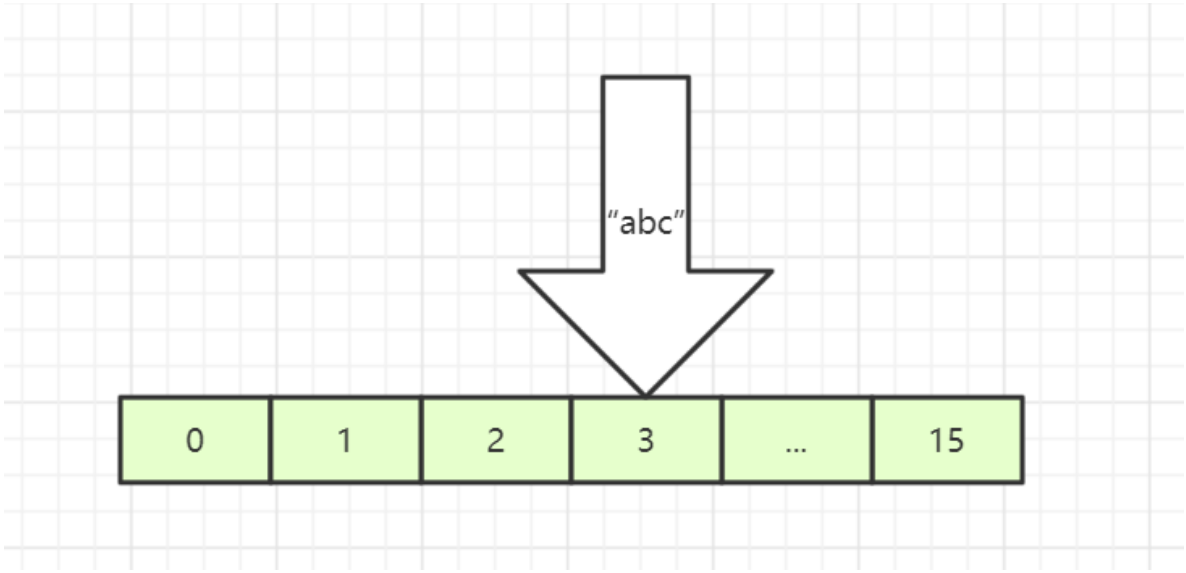
(16 - 1) & 96355=96355%16结论。但是有很多人在想为什么不直接96355%16计算，非要用 (16 - 1) & 9635计算，那是因为从计算机底层算法来分析：“与”（&）运算的性能要高于模运算（%）。所以采用了这么一套优化的方案。所以现在清楚了为什么规定数组的长度是2^X方了吗。

那么真的是96355%16会等于 (16 - 1) & 9635结果吗？很简单，看下图。不相信大家自己可以算一下。不对找我哈哈，这里不解释。。。。。。





从上图可以发现当前key="abc"是保存在数组的索引为3的位置上的。如下图：



6、如何扩容，扩容的机制是什么样的?

6.1 什么时候会触发扩容?

上面第2个问题：成员变量中已经说过了。当保存插入的数据超过当前扩容之前规定的数据容量就会触发扩容。

源码看看 java.util.HashMap#resize

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        //MAXIMUM_CAPACITY=1<<30=1*2^30最大的数组长度=1073741824
        if (oldCap >= MAXIMUM_CAPACITY) {
            //修改阈值为int的最大值(2^31-1)=2147483647，这样以后就不会扩容了。
            //21亿多，最大的容量。
```

```
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        //(newCap = oldCap << 1)表示数组的长度扩容为原来的两倍，也就是说如果原来的数组
length=2那么扩容后的数组length=4。
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; //同理数组的容量也将会的原理的两倍。同理数组的长度。
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    //下面的代码就是扩容后数据是如何迁移的逻辑，这里jdk1.8就跟1.7的有不同的区别。具体区别后面
会做出分析。
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                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;
}
```

6.1-1 通过上面的源码我们可以总结一下

a、数组扩容后的length是原来数组length的2倍。

b、数组最大的长度为：MAXIMUM_CAPACITY=1<<30=1073741824。

c、数组最大的容量为：(2^31-1)=2147483647。21亿约等于。

d、具体扩容分析

JDK1.7扩容源码分析

```
//扩容API
void resize(int newCapacity) { //传入新的容量
    Entry[] oldTable = table; //引用扩容前的Entry数组
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了
        threshold = Integer.MAX_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了
        return;
    }
    Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组
    transfer(newTable); //数据迁移到扩容后的数组中去
    table = newTable; //HashMap的table属性引用新的Entry数组
    threshold = (int)(newCapacity * loadFactor);//修改阈值
}
//数据迁移API
void transfer(Entry[] newTable) {
    Entry[] src = table; //src引用了旧的Entry数组
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
        Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素
        if (e != null) {
            src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity); //! ! 重新计算每个元素在数组中的位置
                e.next = newTable[i]; //标记[1]
                newTable[i] = e; //将元素放在数组上
                e = next; //访问下一个Entry链上的元素
            } while (e != null);
        }
    }
```

```
}
```

以上源码可以得出一个简单的说法，当数组扩容后原来的老数据会由transfer方法迁移到新的数组中去。当然迁移的细节过程是比较复杂的，下面就用几个简单的数据给大家演示一下jdk1.7数据库容到迁移的一个过程。

环境：

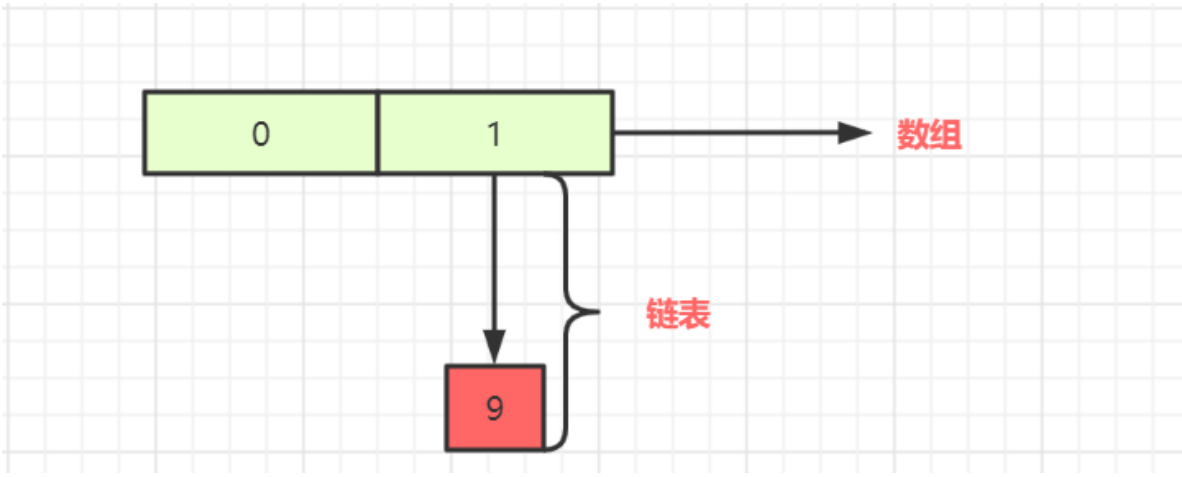
capaCity=2；　　//数组的长度

loadFactor=1f;　　//数组的负载因子

threshold=2*1=2;　　//数组最大的容量，也就是最大能存放的数据个数

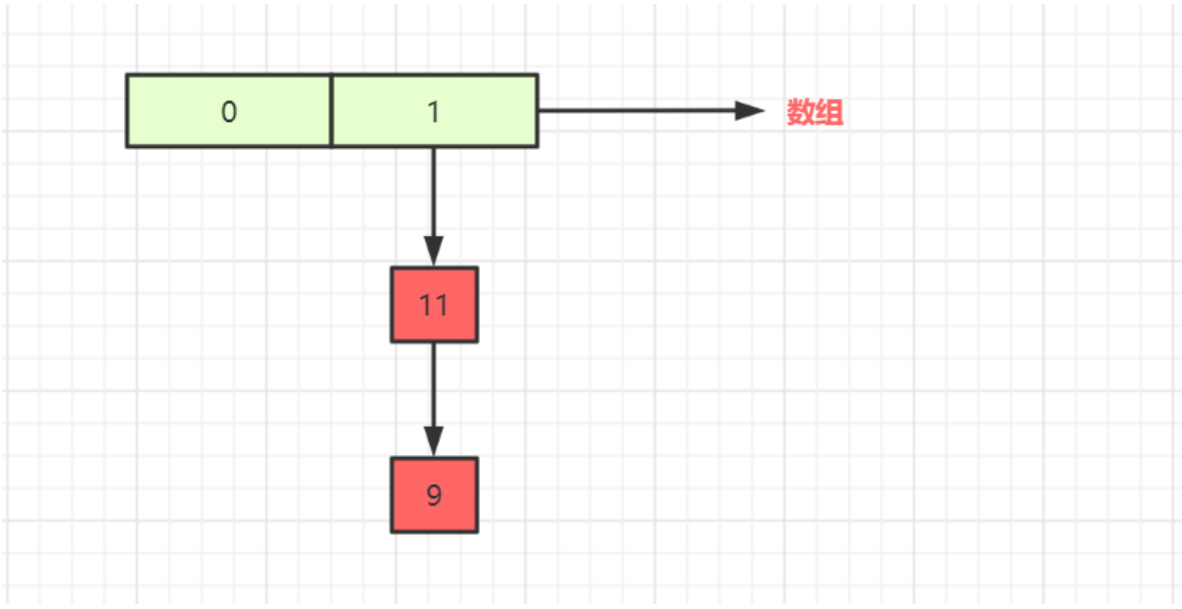
key=9、11、13。同时放这3个数据。这里为了也是简单方便好描述，就将key存入数组中索引的算法规定成简单的模运算。比说9%2=1以此类推。put顺序为9->11->13

第一步：put(9)的结果应该是如下图：9%2=1



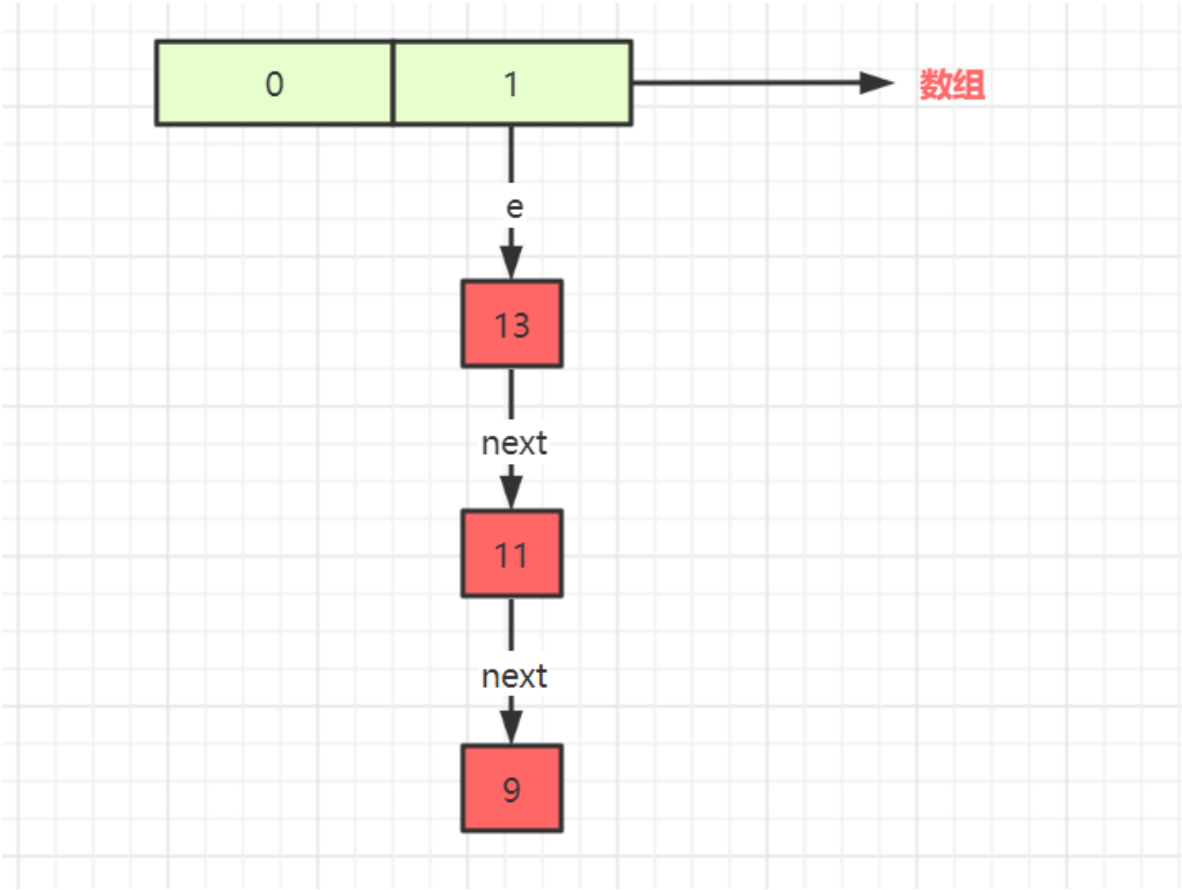
第二步：put(11)的结果应该是如下图 11%2=1 这个时候还不会出发扩容，因为我们定了数组的容量为2。

会发现1.7中插入后的列表元素顺序和插入的顺序是倒置的，这也说明链表在插入的时候是表头插入，才会倒置这个结果。但是1.8不是这样，1.8是按照插入的顺序放在链表的尾部的。



第三步：put(13)的结果应该是如下图 13%2=1 此时会扩容因为3个元素超过我们规定的2了。

1、扩容前的数据状态为下图：

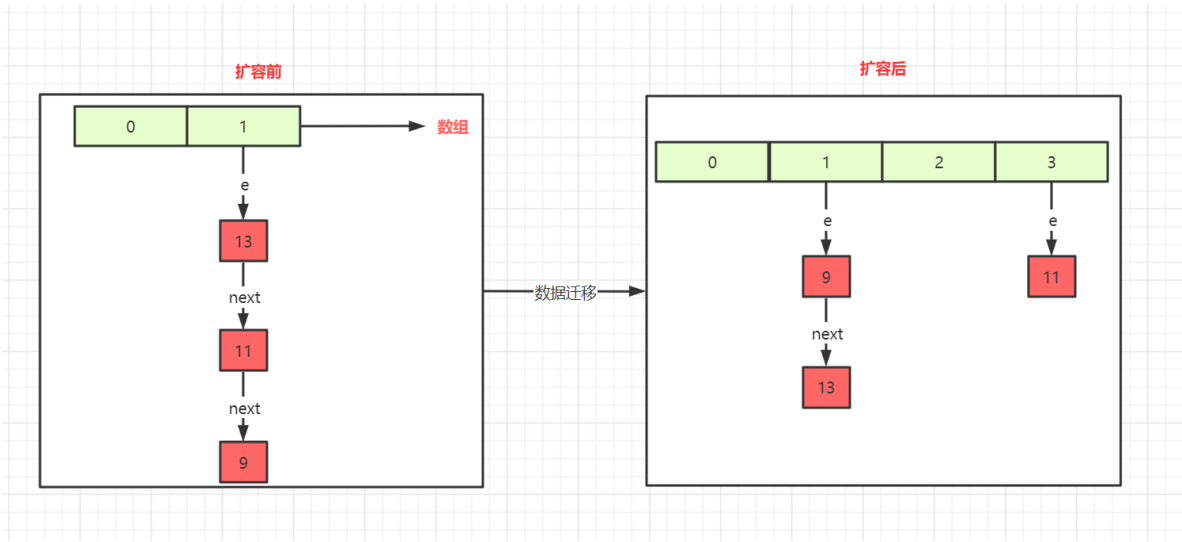


2、扩容过程，现在数组的长度将会是4。这个时候会将老的数组里面的数据重新hash计算数组中落地的索引。

9%4 = 1

11%4 = 3

13%4 = 1



扩容后会返现：

a、新的链表的元素数据顺序又被倒置了（如果是发生了hash碰撞的话）。

b、数据在迁移的过程，有些数据还是在原来数组的索引位置例如9、13。有些数据会挪到别的数组索引位置例如11。这是因为再重新hash的时候导致的。

c、现在我们具体的来分析一下11为甚跑到3上去了？

- 首先11%4=3这个是没有错的。当然实际的源码算法是
- hashCode(11) = Integer.valueOf(11).hashCode;
- hash(11) = hashCode(11)^(hashCode(11)>>>16)
- index(11) = hash(11)&(4-1)因为当前数组的长度等于4,这里的index表示数组中落的索引值为3。
- 这里为了易于理解： index(11) = 11%4 = 3 = table[3]
- 所以这里key=11迁移后，经过重新计算hash就落到了table[3]这个位置上了。

d、但是jdk1.8实现数据迁移的逻辑不是这样的有所不一样。(一下讲解一下1.8相对1.7优化扩容迁移算法)

- 首先上面我们知道jdk1.7数据迁移的话，老数据每一个都会经过重新hash算法确定槽位（数组的索引位置）。有所浪费性能。
- 但是jdk1.8是如何优化扩容数据迁移的算法呢?
- 这里就要回头讲一下为什么数组扩容规定为是原来老数组时长读的2倍。

因为HashMap在数组落地索引计算的过程是：

- hashCode(11) = 90031939;
- index = 90031939&(4-1) =3; //好巧这里既然也真的是3，哈哈。
- index = 90031939&(4-1)转换成二进制演算过程

> 11在老的数组index算法：

90031939 = 0101 0101 1101 1100 0111 0100 0011

1 = 0000 0000 0000 0000 0000 0000 0001

&运算结果= 0000 0000 0000 0000 0000 0000 0001

> 11在老的数组index算法：

90031939 = 0101 0101 1101 1100 0111 0100 0011

3 = 0000 0000 0000 0000 0000 0000 0011

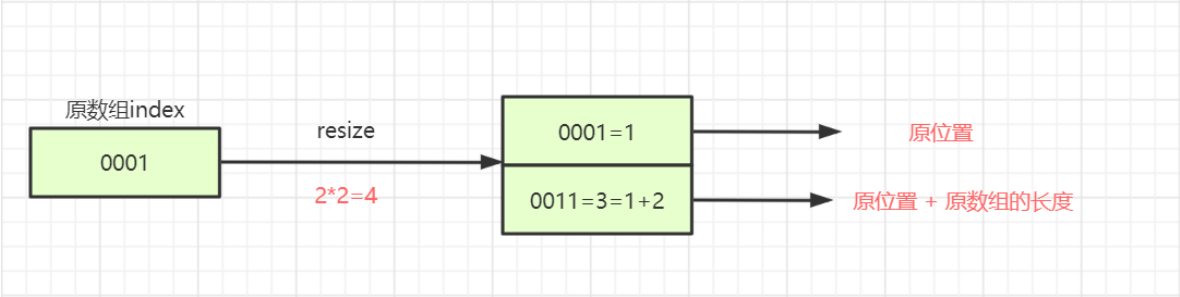
&运算结果= 0000 0000 0000 0000 0000 0000 0011

> > 通过老的和新的index二进制算法结果对照

老结果 0001

新结果 0011

> > 元素在重新计算hash之后，因为n变为2倍，那么n-1的二进制在高位多1bit，因此新的index就会发生这样的变化：



所以这里可以看出为什么扩容的时候为原来的2倍，原来是为了在扩容的时候快速定位数据迁移后的数组槽位。当扩容后是原来的2倍的时候，如果元素的hash没有与之前的槽位反生hash碰撞的话，那么他的index将会很快确定为：原index+原数组长度。这样便大大的提高了数据迁移的性能和效率。所以别问HashMap为什么这样为什么那样，总之记住一句话：为了提升性能。哈哈。。。。。。

7、为什么是线程不安全的？以jdk1.7为例讲解。

相信大家聊完以上这么多关于HashMap的性能优化的细节，但是HashMap他是一个线程不安全的数据结构。但是为了解决这个问题线程安全问题JDK官方也考虑到了这个问题：在性能提升的基础上怎么又能保证HashMap的一个安全性呢？那就是CHM的诞生，全称：ConcurrentHashMap。他是HashMap线程安全的应用产品，至于CMH的详解将会放在下一个文章单独去做讲解。

- 那么什么是线程安全问题呢

  个人理解就是在多个线程共享操作一个全局可变参数的过程，会出现数据的不一致性。也就是说操作后的最终结果不是我们想要的结果。

- 为什么说HashMap调试现成不安全的呢?
  - 在多线程操作的时候，HashMap扩容的时候会出现链表闭环以及数据迁移丢失的情况。所以他是线程不安全的，当然这只是他不安全的其中一个漏洞，应该还会存在别的漏洞大家可以在研究。这里就以这个漏洞给大家分析一下他的不安全的原因。
  - HashMap不安全的漏洞具体分析,上源码。

```
//数据迁移API
void transfer(Entry[] newTable) {
    Entry[] src = table; //src引用了旧的Entry数组
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
        Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素
        if (e != null) {
            src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity); //! ! 重新计算每个元素在数组中的位置
                e.next = newTable[i]; //标记[1]
                newTable[i] = e; //将元素放在数组上
                e = next; //访问下一个Entry链上的元素
            } while (e != null);
        }
    }
}
```

> 这段源码的意思大概就是在数据迁移的时候首先会遍历原来的数组，将每个数组原来位置的链表取出来遍历转移到新的数组中去。其实在上面的问题中又给大家展示1.7数据迁移的过程。数据迁移原来的链表数据会发生反转，就是这段代码的原因。
>
> e.next = newTable[i]; //指出当前e元素的下个元素的引用到新的数组索引位置。
> newTable[i] = e; //将当前元素复制给新的newTable[i]位置，这样就说明了为什么迁移后的链表数据被倒置的原因了。
> e = next; //访问下一个Entry链上的元素，如果为空就停止当前链表的循环迁移操作，进入下一个index的链表循环迁移。

- 图解HashMap扩容不安全原因
  - 假设两个线程Thread-A和Thread-B同时对HashMap进行扩容数据转移。

> ```
> @Data
> public class ThreadMapTest {
>  private static HashMap<String,String> map = new HashMap();
>  public static void main(String[] args) throws InterruptedException
> {
>        new Thread(()->{//线程1
> ```

```
                for (int i=0;i<1000;i++){
                    map.put("key"+i,"value");
                }
            },"Thread-A").start();
            new Thread(()->{//线程2
                for (int i=1000;i<2000;i++){
                    map.put("key"+i,"value");
                }
            },"Thread-B").start();
            Thread.sleep(5000);
            System.out.println("map.size=" + map.size());
     }
    }
```

输出结果：

map.size=1917很明显错误，如果线程安全的话结果应该是map.size=2000.

这里就是扩容数据丢失的情况了。

从源码transfer()分析可知，每一次数据转移的时候Entry[] src = table;每次老的数组会赋值个中间变量操作。然后去遍历这个中间变量槽中的链。

第1步：获取当前的链上的当前的头数据e：Entry<K,V> e = src[j];

第2步：获取当前的链上的当前的头数据的下一个指向的数据next：Entry<K,V> next = e.next;

第3步：算出当前处理的头数据e在新newTable数组中落脚的索引index：

int i = indexFor(e.hash, newCapacity);

第4步：将e.next引用指向新数组newTable的当前索引的头数据。

第5步：然后将当前的e数据节点赋值给当前的newTable[index]位置上，这样原来的next就反过来指向他的上节点了。newTable[i] = e;

(这就是为什么1.7数据迁移，链表hash冲突的时候，数据会被反转的原因)；

第6步：将e.next赋值到下一个待转移的e节点，这样当前的下个节点，又变成当前的e节点操作数据了。知道e.next=null；while (e != null);//这里看出。

现在按照上面的1、2、3、4、5、6步，用图来描述看看中间会有什么并发的问题。

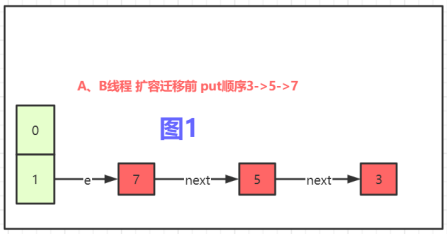
- 假如A、B两个线程同时进入这段代码，如下：

newTable[i] = e;

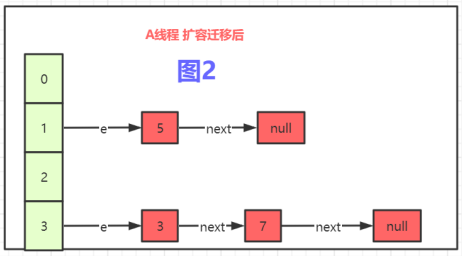
```
7 -> 5 -> 3
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j]; // 第一次：e->7  第二次：e->5
        if (e != null) {
            src[j] = null;//释放旧Entry数组的对象引用,便于垃圾回收。
            do {
                Entry<K,V> next = e.next;  -> 7.next=5   5.next=null
                int i = indexFor(e.hash, newCapacity); //重新计算钱以后数组的落地索引
                // -> i=3(因为7%4=3)   i=1(因为5%4=1)
                e.next = newTable[i];
                //newTable[3]==null
                //newTable[1]==null 5.next=null
                // 5->null   null->7
```

```
                newTable[i] = e; //假设A、B两线程同时执行到这里。
                // newTable[1]=7;  newTable[1]=5;
                e = next;
                //e->5  e->null结束
            } while (e != null);
        }
    }
}
```

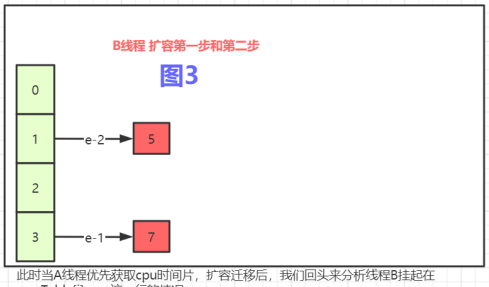
- 数据丢失分析



假设演示的数据如下，并且数组的初始长度=2，负载因子默认0.75f。那么
查过2*0.75=1就要扩容数据迁移。
put(key1,v1)
put(key2,v2)
put(key3,v3)
key1=3 -> 3%2=1
key2=5 -> 5%2=1
key3=7 -> 7%2=1



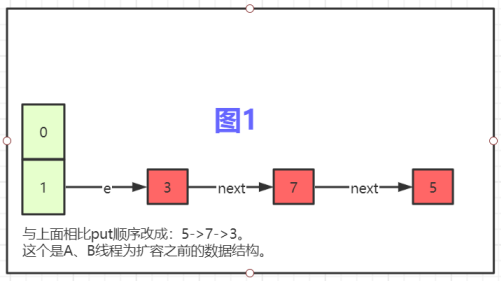
假设A线程和B线程同时执行到：newTable[i] = e;
这个时候将A线程放掉完成它当下的扩容数据迁移工作，此时B线程继续挂起在这行代码。我们将会看到A线程扩容后数据迁移的新数组，如下：
key1=3 -> 3%4=3
key2=5 -> 5%4=1
key3=7 -> 7%4=3



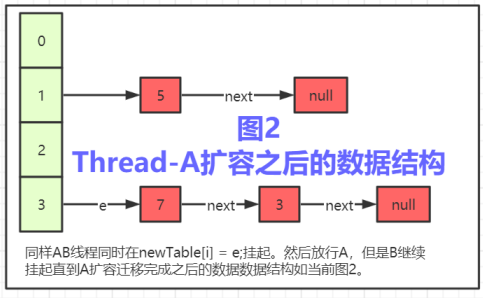
此时当A线程优先获取cpu时间片，扩容迁移后，我们回头来分析线程B挂起在
newTable[i] = e;这一行的情况。
1、Entry<K,V> e = src[j]= src[0]=7; -> e=7
2、Entry<K,V> next = e.next; -> next=5
这时e=7,next=5数据都是在A线程newTable[i] = e;这行挂起之前已经执行了，
所以以上逻辑是没错的。
再往后看B线程，此时B在由newTable[i] = e;这里往后走的场景。
3、这个时候newTable[i] = e;
表示这个时候7已经赋值到新数组newTable[3]上了，而且此时5反过来指向7
了，原来是7指向的next指针。再往下走
4、e = next;
这个时候e=5了；但是这个时候因为线程A已经扩容迁移完数据，有jvm内存我们
清楚，这个时候数据的引用指针关系已经被A改变了，5现在
由图2可知，指向了null,也就是原来由图1可知5指向next->3的，变成了null。
5、这时候当e=5走完transfer之后变成如下图3：

这个时候由图3可知，如果e=5，e.next=null再往下就走不了了。原本5.next是
指向3的，所以这个时候就会出现3数据会被丢失迁移。所以会导致多线程操作
HashMap的时候数据不一致的问题。

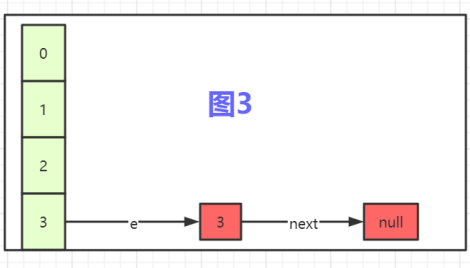
- 链表闭环分析（这个时候将put顺序改一下put：5->7->3）



与上面相比put顺序改成：5->7->3。
这个是A、B线程为扩容之前的数据结构。



同样AB线程同时在newTable[i] = e;挂起。然后放行A，但是B继续
挂起直到A扩容迁移完成之后的数据数据结构如当前图2。



Thread-B 第1步 以图1做为参考，此时：
e=3;
e.next=7;
newTable[3]=null;
===当前这步走完后，进入下面第2步===

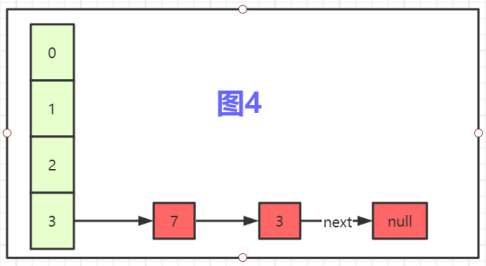


图4

Thread-B 第2步 以图1做为参考，此时：
e=7;
因为源码可知，这一步的等于上一步的next节点数据，e = next;
e.next=3;
按理说这里应该是7.next=5,但是因为A线程并发扩容后变成了7.next=3了。
===当前这步走完后，进入下面第3步===

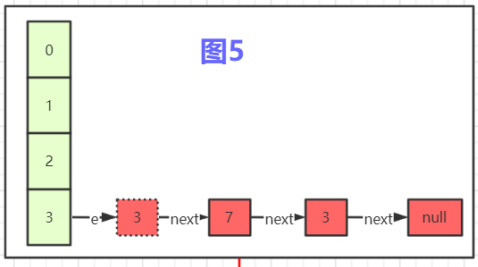


图5

Thread-B 第3步 以图1做为参考，此时：
e=3;
因为源码可知，这一步的等于上一步的next节点数据，e = next;
e.next=null;
===当前这步走完后，进入下面第4步===
这时候因为下一步的e=e.next=null
由源码while (e != null)可知不会再继续虚幻下去了，结束当前的槽位上的链表迁移。并且形成闭环。

等价于

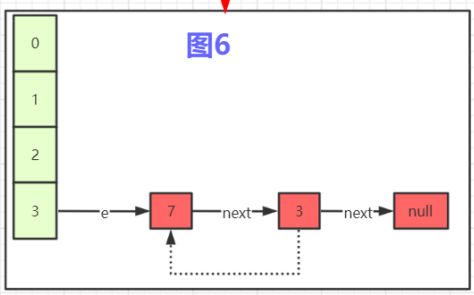


图6

最后形成闭环，会给后续遍历链表带来回环异常操作。