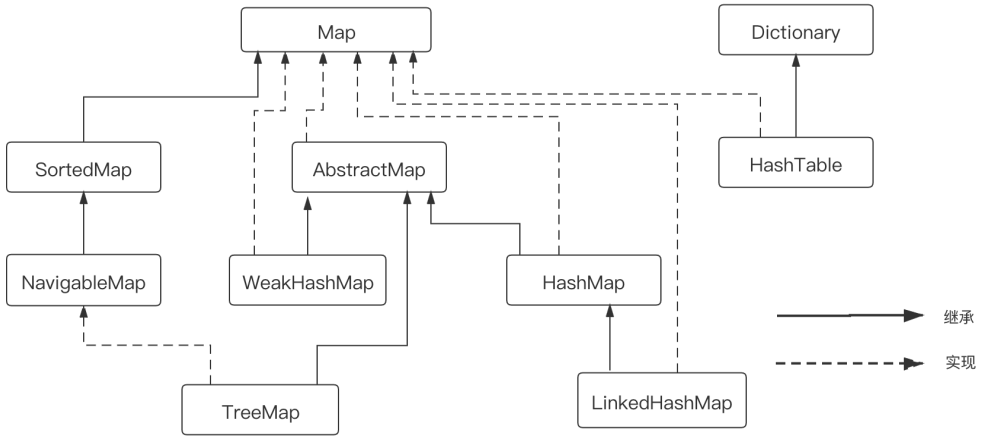
> 在日常开发中，集合作为存储数据的容器，被广泛使用在程序代码中，本文将从JDK集合类代表HashMap出发，着重理解HashMao底层实现。

一、Map家族关系图

在正式讨论HashMap之前，我们有必要把Map家族的继承实现关系展示出来，方便理解后续的内容。

Map家族关系图



上图很详细地展示了Map家族中各个成员之间的继承或者实现关系。Map接口是双列集合的顶层父接口，该集合中每个元素都是键值对key-value，成对出现。双列集合中，一个键一定只找到对应的一个值，键不能重复，但是值可以重复。Map集合常用的实现类有HashMap、LinkedHashMap、HashTable、TreeMap。

- **HashMap**

HashMap底层是数组+链表，与HashSet相似，只是数据的存储形式不同，HashMap可以使用null作为key或value，是线程不安全，但是效率相对较高。当给HashMap中存放自定义对象时，如果自定义对象作为key存在，这时要保证对象唯一，必须要重写对象类的hashCode和equals方法（重写equals必须重写hashCode）

- **HashTable**

HashTable与HashMap之间的关系完全类似于Vector与ArrayList的关系。HashTable是线程安全的，但是效率相对较低，Hashtable不允许使用null作为key和value。

- **LinkedHashMap**

LinkedHashMap是HashMap的子类，其底层是数组+链表（双向链表），其关系与HashSet与LinkedHashSet类似，使用链表来维护key-value的次序，可以记住键值对的插入顺序。

- **TreeMap**

TreeMap存储key-value键值对时，需要根据key对节点进行排序。TreeMap可以保证所有的key-value对处于有序状态。也有两种排序方式：

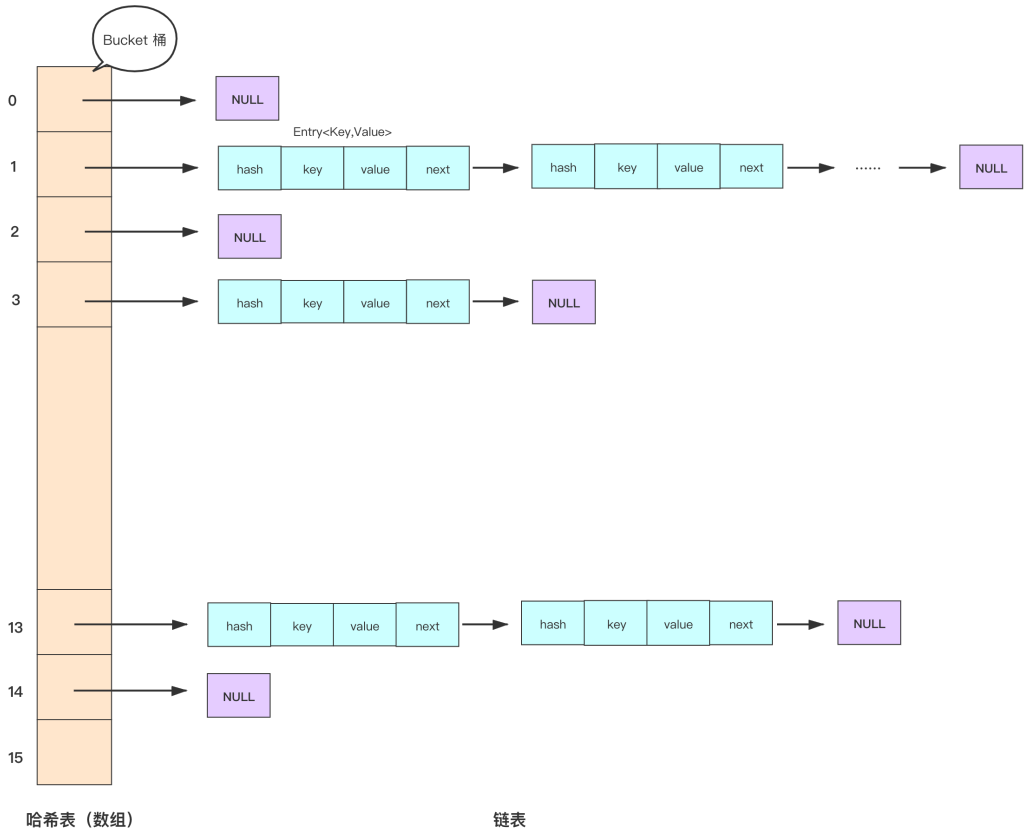
自然排序：TreeMap的所有key必须实现Comparable接口，而且所有的key应该时同一个类的对象，否则抛出ClassCastExecption异常。

定制排序：创建TreeMap时，传入一个Comparator对象，该对象负责对TreeMap中的所有key进行排序。不需要实现Map的key实现Comparable接口。

二、JDK7中HashMap底层原理

2.1 HashMap在JDK7中的结构

HashMap在JDK7或者JDK8中采用的基本存储结构都是**数组+链表**形式（JDK8:数组+链表+红黑树）。本节主要是研究HashMap在JDK7中的底层实现，其基本结构图如下所示：



上图中左边橙色区域是哈希表（数组），右边蓝色区域为链表，链表中的元素类型为Entry<K,V>,它包含四个属性分别是：

- K key
- V value
- int hash
- Entry<K,V> next

那么为什么会出现**数组+链表**形式的存储结构呢？这里简单阐述一下，后续将以源码的形式详细介绍。

我们在使用HashMap.put("Key","Value")方法存储数据的时候，底层实际是将key和value以Entry<Key,Value>的形式存储到哈希表中，哈希表是一个数组，那么它是如何将一个Entry对象存储到数组中呢？是如何确定当前key和value组成的Entry该存储到数组哪个位置上，换句话说是如何确定Entry对象在数组中的索引的呢？通常情况下，我们在确定数组的时候，都是在数组中挨个存储数据，直到数组全满，然后考虑数组的扩容，而HashMap并不是这么操作的。在Java及大多数面向对象的编程语言中，每个对象都有一个整形变量hashcode，这个hashcode是一个很重要的标识，它标识这不同的对象，有了这个hashcode，那

么就很容易确定Entry对象的下标索引了，在Java语言中，可以理解hashcode转化为数组下班时按照数组长度取模运算的，基本公式如下所示：

```
int index = HashCode(key) % Array.leng
```

实际上，在JDK中哈希函数并没有直接采取取模运算，而是利用了位运算的方式来提高性能，在这里我们理解为简单的取模运算。

我们知道了对Key进行哈希运算后然后对数组长度进行取模就可以得到当前Entry对象在数组中的下标，那么我们可以一直调用HashMap的put方法持续存储数据到数组中。但是存在一种现象，哪就是根据不同的key计算出来的结构有可能完全相同，这种现象叫做“**哈希冲突**”。既然出现了哈希冲突，那么发生冲突的这个数据该如何存储呢？哈希冲突其实是无法避免的一个事实，既然无法避免，那么就应该想办法来解决这个问题，目前常用的方法主要是两种，一种是开发寻址法，另外一种是链表法。

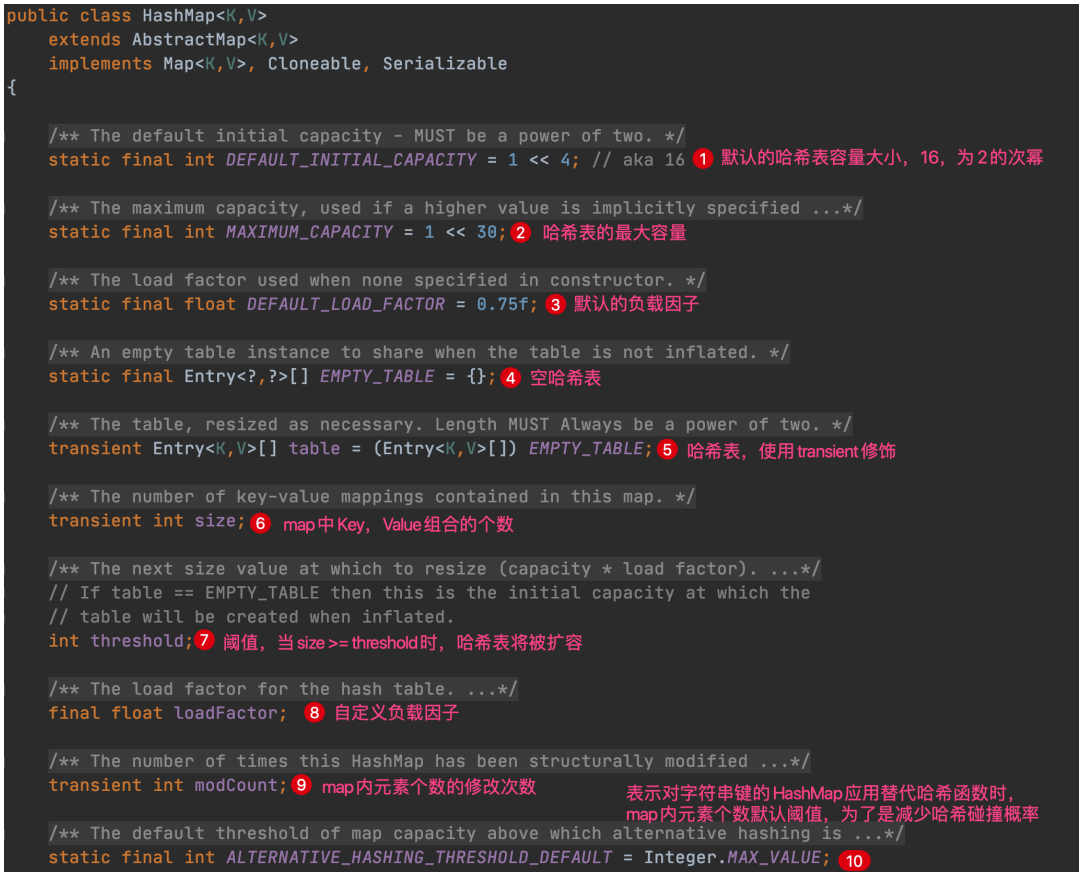
开发寻址法原理比较简单，就是在数组厘米“另谋高就”，尝试寻找下一个空档位置，JDK中TreadLocal发生哈希冲突以后就是采用的开放寻址法。而链表法则不是寻找下一个空档位置，而是继续在当前冲突的地方存储，与现有的数据组成链表，以链表的形式进行存储。

HashMap的存储形式是**数组+链表**就是采用的链表法来解决哈希冲突问题的。具体详细说明请继续往下看。

在日常开发中，开发中对于HashMap使用的最多的就是它的构造方法、put方法以及get方法，下面就开始详细地从三个方法出发，深入理解HashMap的实现原理。

2.2 深入理解HashMap的构造方法

查看HashMap的源码，首先得了解一下HashMap的一些成员属性，它的主要属性如下图所示：



1. 第一个属性是默认的初始化容量，表示哈希表的默认长度，当我们在创建HashMap对象的时候未指定容量时，默认的容量就是16。

这里多说一句，当我们知道我们的kye-value对的个数的时候（一般是个数大于16），我们在创建HashMap对象的时候最好就指定容量大小，很多人认为，我们要存入多说K-V对就制定多少初始容量，其实这样时不对的，你只的初始化容量不一定是最后真正的初始化容量，因为你设置的初始化容量要经过转换的，它会被转换成大于它本身且接近它的2的次幂。比如，我们需要在HashMap中存储27个K-V对，那么大雨27且靠近27的2的次幂就是32，那么如果你在创建HashMap对象的时候指定的容量为27，那么其实创建出来的HashMap容量为32，且当你连续存储到满了24个以后，当你存入第25个K-V对的时候，有可能就会触发扩容，这样不仅没有减少扩容次数，反而大概地发生扩容。

那么我们该如何确定传入的值呢？这里提供一个公式：`(int)((float) expectSize / 0.75f + 1.0f)`，这个公式来自于HashMap的一个构造方法，比如你的 expectSize = 27，那么及时的结果就是 37，那么就可以传入37作为初始化容量，经过内部计算以后，大于37且最接近37的2的次幂是64，最终初始化的结果就是数组长度为64，反向计算扩容阀值就是48，那么存入27个K-V对是不会发

生扩容的，如果仅仅是传入的27，那么有很大可能会发生扩容，影响性能。

2. 第二个属性是哈希表的最大容量。

3. 第三个属性默认的负载因子，也就是我们在创建HashMap对象的时候没有指定负载因子，那么就使用默认的负载因子（0.75），它是哈希表扩容的一个重要参数，当HashMap中存储的K-V对的个数大于等于哈希表容量乘以这个负载因子（size >= capacity * DEFAULT_LOAD_FACTOR）的时候，将触发扩容，且扩容为原来的两倍。举个例子：当我们创建HashMap对象未指定哈希表容量的时候，也就是使用默认的16，当调用HashMap对象的put方法存储数据的时候，当存储的数量为16*0.75f=12的时候，将触发扩容机制，此时哈希表容量将扩充为32，这也就是在上面第一个属性描述的时候为什么多说一句的原因。

4. 第四个属性是空的哈希表。

5. 第五个属性是后续操作的数组。

6. 第六个属性是HashMap中存储的K-V组合的个数。

7. 第七个属性是扩容阀值，当 size >= threshold 的时候（实际还需要满足有链表的条件），将触发哈希表的扩容机制，threshold = capacity * loadFactor。

8. 第八个属性是自定义负载因子，当没有指定负载因子的时候，loadFactor = 0.75f

9. 第九个属性是记录了Map 新增/删除 K-V对，或者内部结构做了调整的次数。其主要作用，是对Map的遍历操作做一致性校验，如果在iterator操作的过程中，map的数值有修改，直接抛出ConcurrentModificationException异常。

10. 第十个属性表示对字符串键的HashMap应用代替哈希函数时，Map内元素个数的默认阀值，目的是为零减少哈希碰撞的概率。

在JDK7中，HashMap有4个构造方法，分别是：

```
1 // 该构造方法是第二第三构造方法的底层实现，该
2 // 然后将传入的负载因子复制给了loadFactor成
3 // 将初始化容量赋值给了扩容阈值（扩容临界数值
4 public HashMap(int initialCapacity, fl
5     if (initialCapacity < 0)
6         throw new IllegalArgumentExcep
7
```

```
8       if (initialCapacity > MAXIMUM_CAPA
9           initialCapacity = MAXIMUM_CAPA
10      if (loadFactor <= 0 || Float.isNa
11          throw new IllegalArgumentExce
12
13
14      this.loadFactor = loadFactor;
15      threshold = initialCapacity;
16      init();
17  }
18
19
20  // 指定容量来创建HashMap对象，该变量initi
21  public HashMap(int initialCapacity) {
22      this(initialCapacity, DEFAULT_LOA
23  }
24
25  // 使用默认的容量16和默认的负载因子0.75f来
26  public HashMap() {
27      this(DEFAULT_INITIAL_CAPACITY, DE
28  }
29
30  // 传入一个Map将其转化为HashMap
31  public HashMap(Map<? extends K, ? ext
32      // 创建一个HashMap
33      this(Math.max((int) (m.size() / D
34                      DEFAULT_INITIAL_CAP
35      // 初始化HashMap，重点方法，后面详细介
36      inflateTable(threshold);
37      // 将Map转化为HashMap的具体实现
38      putAllForCreate(m);
39  }
40
```

我们一起看看前三个构造方法，第二个第三个构造方法搜到考验第一个构造方法，这三个构造方法很简单，但是第一个构造方法有一点需要注意，那就是构造方法将初始化容量initialCapacity的值赋值给了扩容阀值threshold，这就给我们带来一个疑问，一开始我们都知道一个等式关系：`threshold=capacity*loadFactor`，难道默认情况下，我们使用HashMap的无参构造方法，扩容阀值和默认容量一样，都是16？难道第一次扩容是size=16猜开始吗？这个疑问我们先放一放，稍后在put方法中你就知道到底是怎么回事了。

我们先研究下第四个构造方法，第四个构造方法是将一个Map转换成HashMap，一共三行代码：

第一行代码创建了一个HashMap对象，设置了threshold和loadFactor值。

第二行代码是初始化HashMap对象。

第三行是将Map转换成HashMap的具体实现。

一起来看看第二行代码内部实现：

```
1  private void inflateTable(int toSize)
2      // 计算大于toSize的且最接近toSize的2的
3      // 比如toSize=27，那么大于27且最接近2
4      int capacity = roundUpToPowerOf2(t
5
6      // 这里重新计算了threshold，由代码可知
7      threshold = (int) Math.min(capacit
8      // 初始化了哈希表（桶数组）
9      table = new Entry[capacity];
10      initHashSeedAsNeeded(capacity);
11 }
12
13 private static int roundUpToPowerOf2(
14     // assert number >= 0 : "number m
15     return number >= MAXIMUM_CAPACITY
16             ? MAXIMUM_CAPACITY
17             : (number > 1) ? Integer.
18 }
```

从第四个构造方法可知，转化过程中，暂时将确定的一个数组容量值赋值给threshold暂存，在初始化数组之前，计算出最合适的capacity（大于threshold且最接近threshold的2的N次幂），然后再计算出真正的扩容阈值threshold（threshold = capacity * loadFactor），然后在创建Entry数组（桶数组）。这里其实也对上面疑问进行了一个解答，其实我们在使用HashMap的无参构造创建HashMap对象的时候，并没有初始化Entry数组，那么是何时初始化Entry数组的呢？那是在第一次调用put方法的时候，后续，我们会一起深入理解HashMap的put方法。接下来，我们继续看putAllForCreate方法。

```
// 这就是将指定Map转换为HashMap的方法，循环
private void putAllForCreate(Map<? ext
    for (Map.Entry<? extends K, ? exte
        putForCreate(e.getKey(), e.get
}


private void putForCreate(K key, V val
    // 计算hash值，如果key==null，那么ha
    int hash = null == key ? 0 : hash
    // 利用hash值和哈希表长度计算当前k-v应
    int i = indexFor(hash, table.leng

            // 由于table[i]处可能不止有一
    // 当key存在的时候，直接将key的值设置
    for (Entry<K,V> e = table[i]; e !
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (k
            e.value = value;
            return;
        }
    }

    // 当key在当前数组和所有链表中不存在的时
```

```
26      createEntry(hash, key, value, i);
27  }
28
29  void createEntry(int hash, K key, V v
30      // 获取当前数组的Entry链表头节点，并赋
31      Entry<K,V> e = table[bucketIndex]
32      // 创建一个新的Entry节点对象，并作为新
33      // 这一点很重要，可以看Entry的构造方法
34      table[bucketIndex] = new Entry<>(
35      // Entry对象数量+1
36      size++;
37  }
```

上面的代码分析很重要，对后续的put方法研究很有帮助，从上面的代码分析可知，我们从源码级别了解到JDK7中的链表新增的新成员是**插入头节点**的。

2.3 深入理解HashMap的put方法

对于HashMap，平常使用最多的就是put方法来，从上面的源代码分析可知，在创建HashMap对象后，并没有初始化哈希表，也就是HashMap的成员属性`Entry<K,V>[] table = (Entry<K,V>[]) EMPTY_TABLE = {}`，那这个哈希表是何时开辟空间的？答案很明显，那就是第一次使用put方法的时候。

```
1  public V put(K key, V value) {
2      // 首先判断哈希表table是否是空表，第一
3      // 如果我们使用的HashMap的无参构造方法
4      // 上面一句分析过inflabteTable方法来，
5      // 并创建了一个新的Entry<K,V>[16]赋值
6      // 第二次put开始，table就不再是空数组
7      if (table == EMPTY_TABLE) {
8          inflateTable(threshold);
9      }
10     // 判断key是否为null，如果为null，那
```

```
    if (key == null)
        return putForNullKey(value);
    // 计算key的hash值
    int hash = hash(key);
    // 根据hash值和哈希表长度来计算当前K-
    int i = indexFor(hash, table.leng
    // 找到下标后，获取当前下标对应的是否有
    // 判断链表中是否已经包含当前key，如果
    for (Entry<K,V> e = table[i]; e !
        Object k;
        if (e.hash == hash && ((k = e
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    // 能运行到这一步，说明上述链表不存在或
    // 那么就需要添加新的Entry对象到指定的
    // 记住这里也是头节点插入
    modCount++;
    // 新值Entry，不仅新值，而且还担任了扩
    addEntry(hash, key, value, i);
    // 新增Entry对象，返回null
    return null;
}
```

上面对put方法进行了解析，接下来我们深入理解一下putForNullKey方法和addEntry方法，首先我们一起理解一下putForNullKey方法，代码如下：

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

```
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(0, null, value, 0);
    return null;
}
```

该方法对key为null的特殊处理的方法，在HashMap设计的时候，旧规定了哈希表第一个位置，也就是下标为0的位置，只存放key=null的Entry对象，这也就合理解释了HashMap考验存储key为null的K-V对，且最多只存储一个key为null的Entry对象。

接下来一起看看addEntry方法，该方法不仅担任了添加Entry对象任务，还担任了扩容的任务

```
oid addEntry(int hash, K key, V value,
    // 扩容的条件：1. size >= threshold
    // 2. null != table[bucketIndex]
    // 两个条件同时满足才会进行扩容
    if ((size >= threshold) && (null !
            // 扩容为原来容量的两倍，具体作
        resize(2 * table.length);
        // 重新计算当前要新增的K-V对在新的
        hash = (null != key) ? hash(ke
         bucketIndex = indexFor(hash,
    }

    // 创建Entry对象，并插入到指定位置的头
    createEntry(hash, key, value, buc
}
```

上面的两处代码分析可以得出一个结论：

扩容的条件必须满足size >= threshold和null != table[bucketIndex]，看到很多资料说只要size >= threshold 就会触发扩容机制，这是不对的。看条件旧制度，size >= threshold 中的等于号一定会满足，只要一直执行put方法，那么一定就会有等于的时候，那么为啥还需要大于号呢/按照错误的想法，岂不是大于号多此一举？其实不然，扩容要满足两个条件，如果put方法在put第threshold个K-V对的时候，但是存放Entry对象的数组bucketIndex处并没有链表，那么也不会扩容，也就是说，put第threshold个K-V对且发生哈希冲突才会扩容。引申一点，当put键为null的K-V对的时候，永远不会发生扩容，因为它要么发生Value的替换，那么是一个Entry对象的插入，不会涉及到哈希冲突。

进一步总结如下：

- 扩容条件必须同时满足 size >= threshold和null != table[bucketIndex]
- put键为null的K-V对的时候永远不会发生扩容

2.4 深入理解HashMap的扩容机制

从上面小节的分析可知，扩容的条件为：

- 扩容条件必须同时满足 size >= threshold和null != table[bucketIndex]
- put键为null的K-V对的时候永远不会发生扩容

我们查看resize方法的具体实现来理解HashMap的扩容机制，代码如下：

```
1  void resize(int newCapacity) {
2      Entry[] oldTable = table;
3      int oldCapacity = oldTable.length;
4      // 如果老数组的容量达到了最大，那么就将t
5      if (oldCapacity == MAXIMUM_CAPACIT
6          threshold = Integer.MAX_VALUE;
7          return;
8      }
9
10     // 创建一个新的哈希表，容量是原来哈希表
11      Entry[] newTable = new Entry[newC
12      // 将重新计算所有Entry对象的下标，并重
13      transfer(newTable, initHashSeedAs
```

```
14      table = newTable;
15      // 重新计算新的扩容阀值threshold
16      threshold = (int)Math.min(newCapa
17  }
```

扩容很简单，就是将哈希表的长度变成原来的两倍，但是这仅仅是第一步，后面还需要对原有的数据进行迁移，使原有数据更加均匀地散列在哈希表中，数据迁移的具体操作方法由transfer来提供。

```
1  void transfer(Entry[] newTable, boolea
2      int newCapacity = newTable.length;
3      // 遍历老的tbale，遍历到每一个bucker的
4      for (Entry<K,V> e : table) {
5          while(null != e) {
6              Entry<K,V> next = e.next;
7              // 重新计算哈希
8              if (rehash) {
9                  e.hash = null == e.key
10             }
11             // 重新计算下标
12             int i = indexFor(e.hash,
13             // 头节点插入
14             e.next = newTable[i];
15             newTable[i] = e;
16             // 继续原链表的下一个节点
17             e = next;
18         }
19     }
20 }
```

这就基本完成了对HashMap的put方法的研究，其实研究起来并不是很难理解，耐下心来阅读一下源码，肯定会有很大收获。

2.5 深入理解HashMap的get方法

接下来我们继续来阅读get方法，相对于put方法，get方法实现起来就简单很

多，理解起来也不是很困难，基本原理就是计算key的hash值，然后计算当前key在哈希表中的索引位置，然后再遍历链表，追个比对，最后返回结果。

```
public V get(Object key) {
    // 首先判断key是否为null，如果为null，
    if (key == null)
        return getForNullKey();

    // 当key != null，命中到哈希表中的某个
    Entry<K,V> entry = getEntry(key);

    // 处理结果并返回，如果找到就返回对应的
     return null == entry ? null : ent
}

// 处理key = null的值问题
private V getForNullKey() {
    if (size == 0) {
        return null;
    }
    // 返回哈希表中下标为0的Entry对象并处
    for (Entry<K,V> e = table[0]; e !
        if (e.key == null)
            return e.value;
    }
    return null;
}

// 遍历链表匹配到相同key的Entry对象并返回
final Entry<K,V> getEntry(Object key)
    if (size == 0) {
        return null;
    }

```

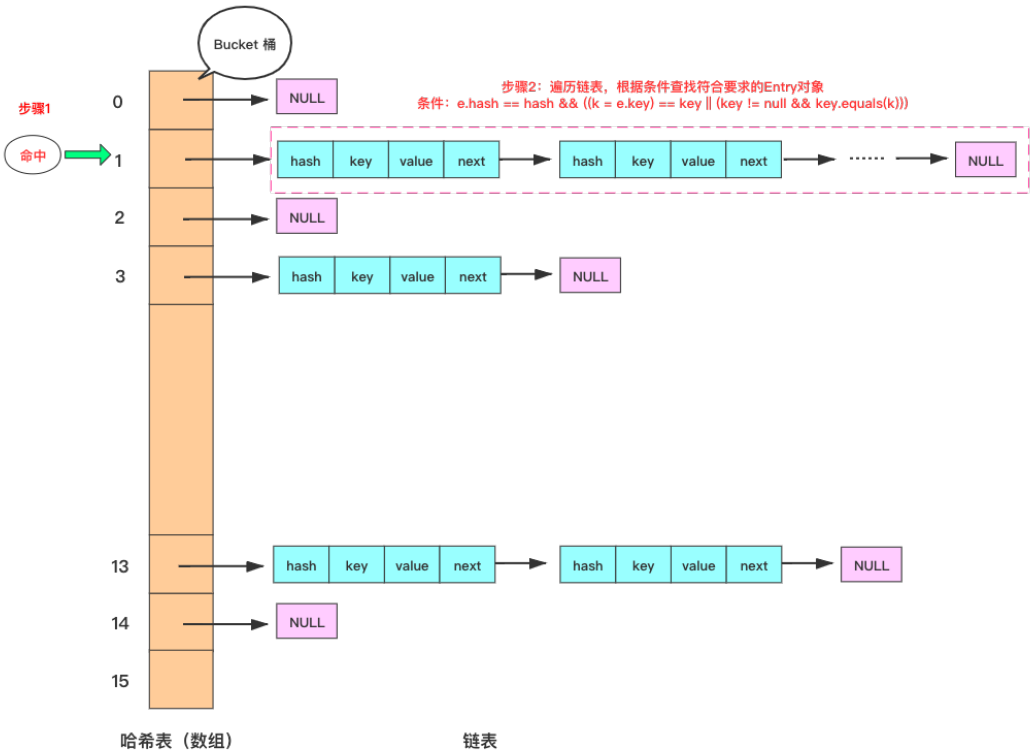
```
32      int hash = (key == null) ? 0 : ha
33      for (Entry<K,V> e = table[indexFo
34           e != null;
35           e = e.next) {
36          Object k;
37          if (e.hash == hash &&
38              ((k = e.key) == key || (k
39              return e;
40      }
41      return null;
42  }
```

getEntry方法是遍历链表来获取与当前key相同的ENtry对象，判断Key是否存在的标准是：

`e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))`

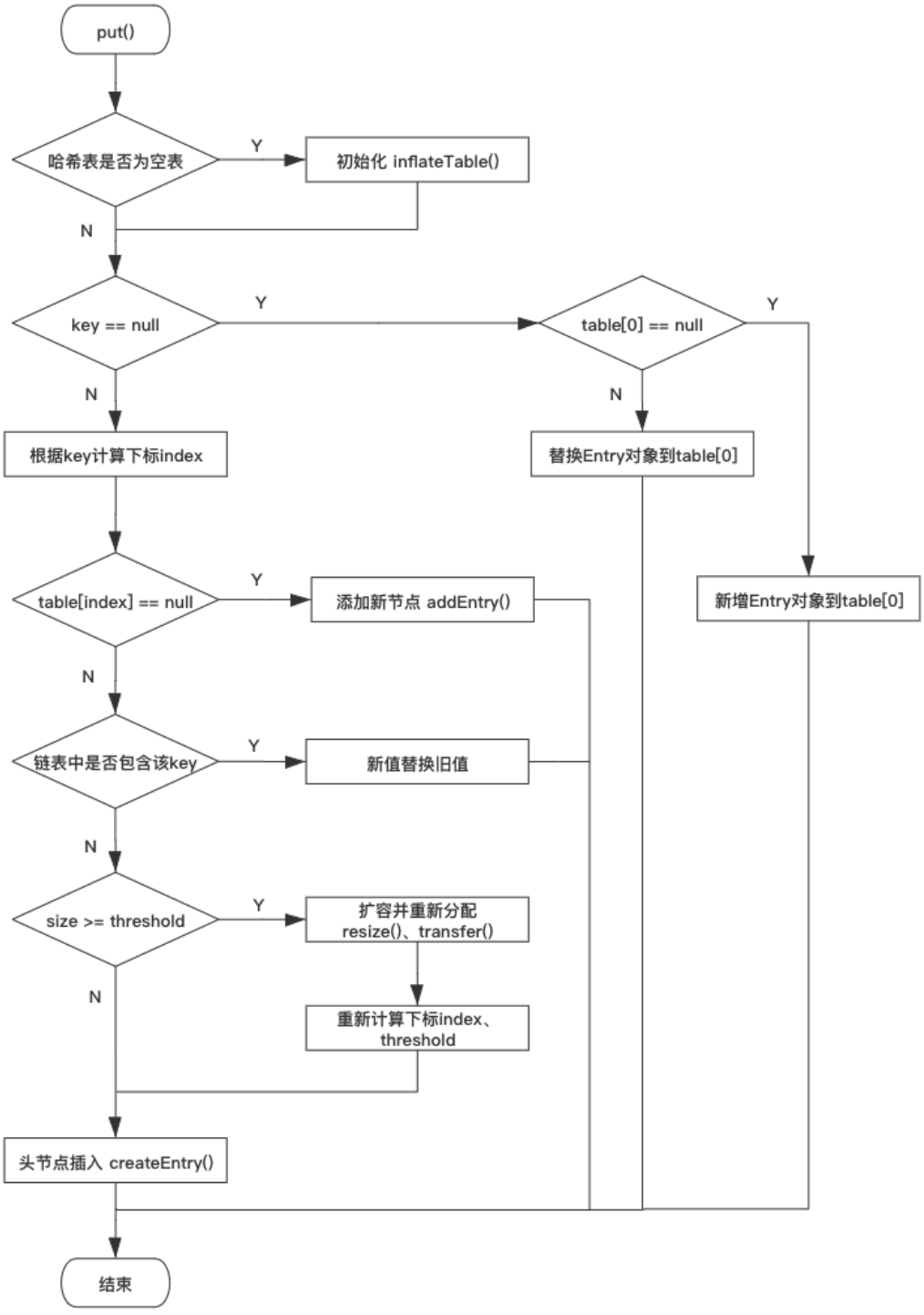
节点的key与目标的key的哈说值肯定是相等的，&& 右边的条件，即节点的key与目标key的相等，要么内存地址相等，要么逻辑上相等，两者有一个满足即可。

假设get方法传入的key值计算的下标为1，HashMap的get方法的基本示意图如下所示：

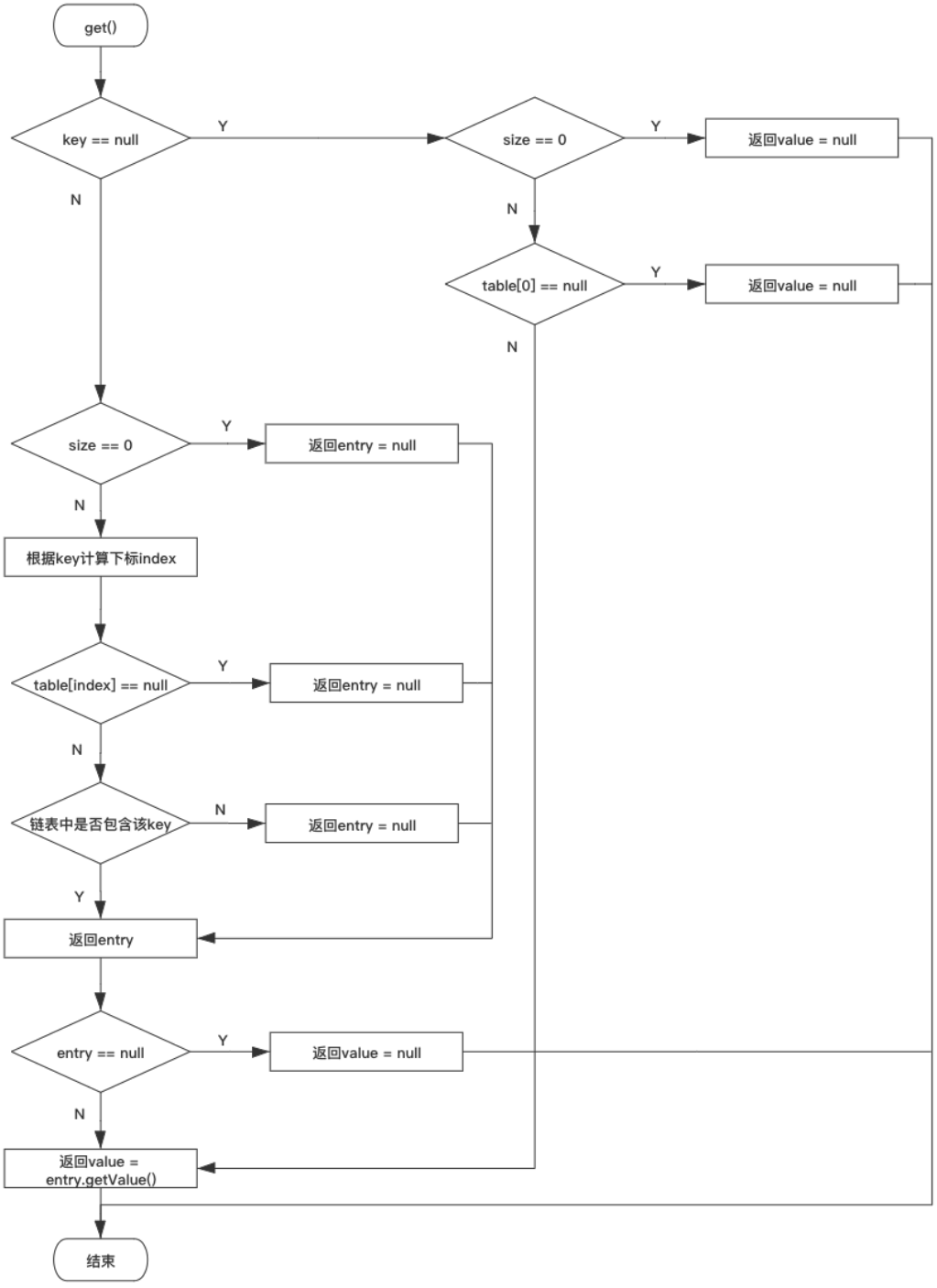


2.6 HashMap put、get方法流程图

这里提供一个HashMap的put方法存储数据的流程图供参考：



这里提供了一个HashMap的get方法获取数据的流程图供参考：



2.7 常见的HashMap的迭代方式

在实际开发过程中，我们对于HashMap的迭代遍历也是常见的操作，HashMap的迭代遍历常用方式有如下几种：

- 方式一：迭代器模式

```
Map<String,String> map = new HashMap<>
Iterator<Map.Entry<String,String>> ite
while(iterator.hasNext()){
    Map.Entry<String,String> next = it
    System.out.println(next.getKey() +
}
```

- 方式二：遍历Set<Map.Entry<K,V>>方式

```
Map<String,String< map = new HashMap<>
for(Map.Entry<String,String> entry : m
    System.out.println(entry.getKey()
}
```

- 方式三：forEach方式（JDK8特性，lambda）

```
Map<String,String> map = new HashMap<>
map.forEach((key,value) -> System.out.
```

- 方式四：KeySet方式

```
Map<String,String> map = new HashMap(1
Iterator<String> keyIterator = map.key
while(keyIterator.hasNext())){
    String key = keyIterator.next();
    System.out.println(key + ":" + map
}
```

把这四种方式进行比较，前三种其实属于同一种，都是迭代器遍历方式，如果要同时使用到key和value，推荐使用前三种方式，如果仅仅使用到key，那么推荐使用第四种方式。