

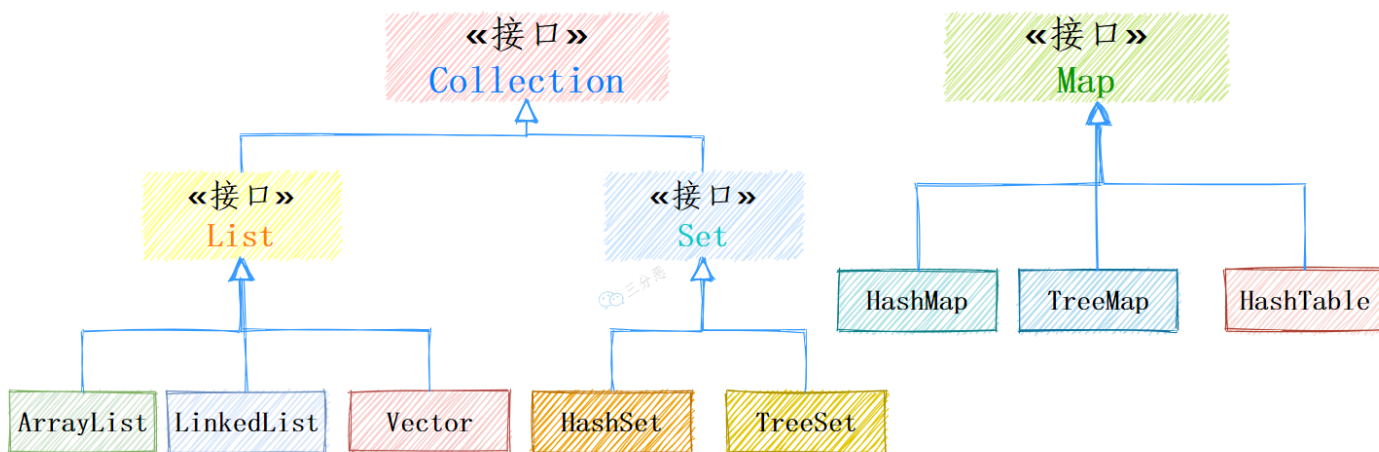
图文详解 30 道Java集合框架面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳[转载链接](#)，作者：三分恶，戳[原文链接](#)。

引言

1.说说有哪些常见集合？

集合相关类和接口都在java.util中，主要分为3种：List（列表）、Map（映射）、Set(集)。

Java主要集合关系



其中 **Collection** 是集合 **List**、**Set** 的父接口，它主要有两个子接口：

- **List**：存储的元素有序，可重复。
- **Set**：存储的元素不无序，不可重复。

Map 是另外的接口，是键值对映射结构的集合。

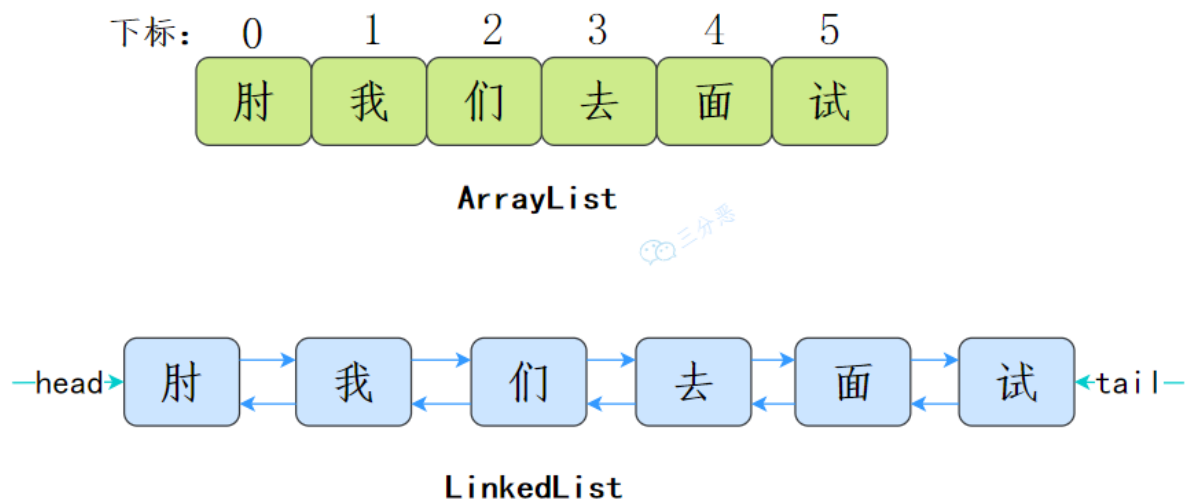
List

List，也没啥好问的，但不排除面试官剑走偏锋，比如面试官也看了我这篇文章。

2.ArrayList和LinkedList有什么区别？

(1) 数据结构不同

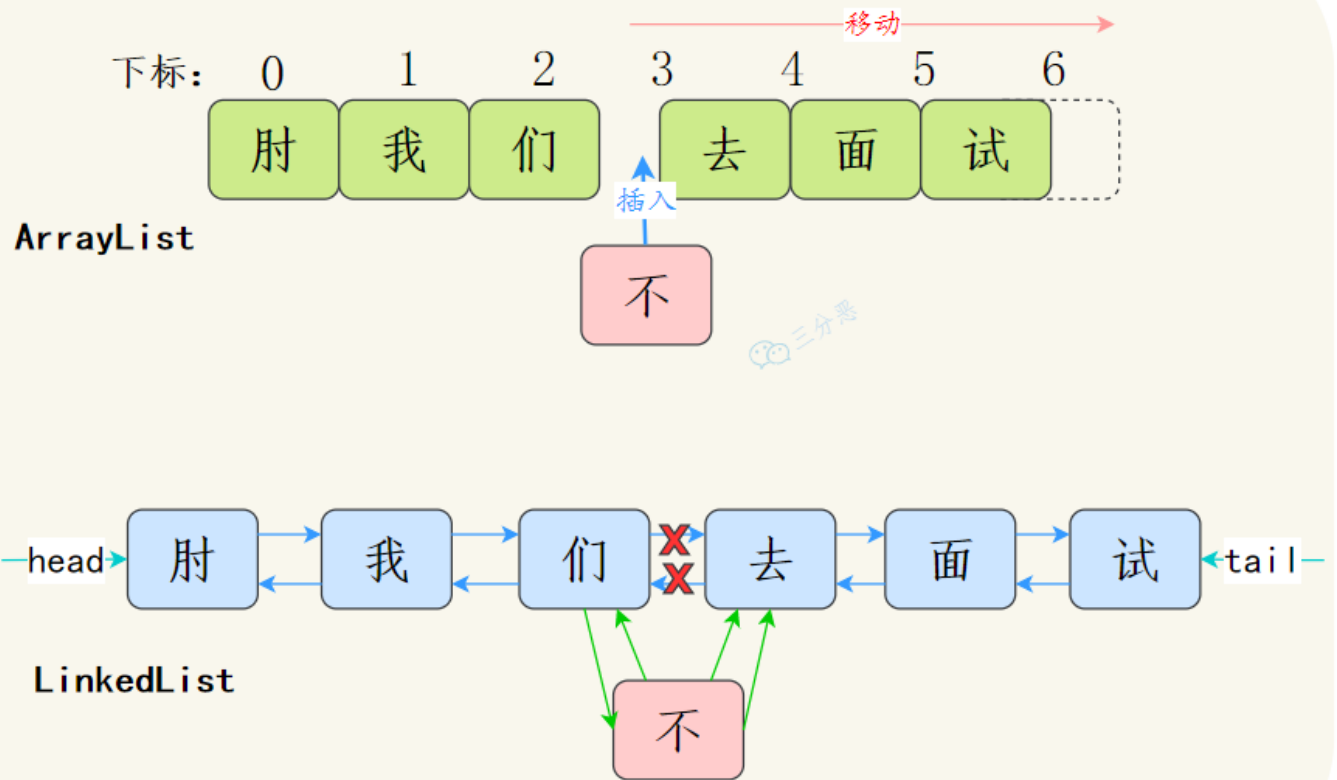
- ArrayList基于数组实现
- LinkedList基于双向链表实现

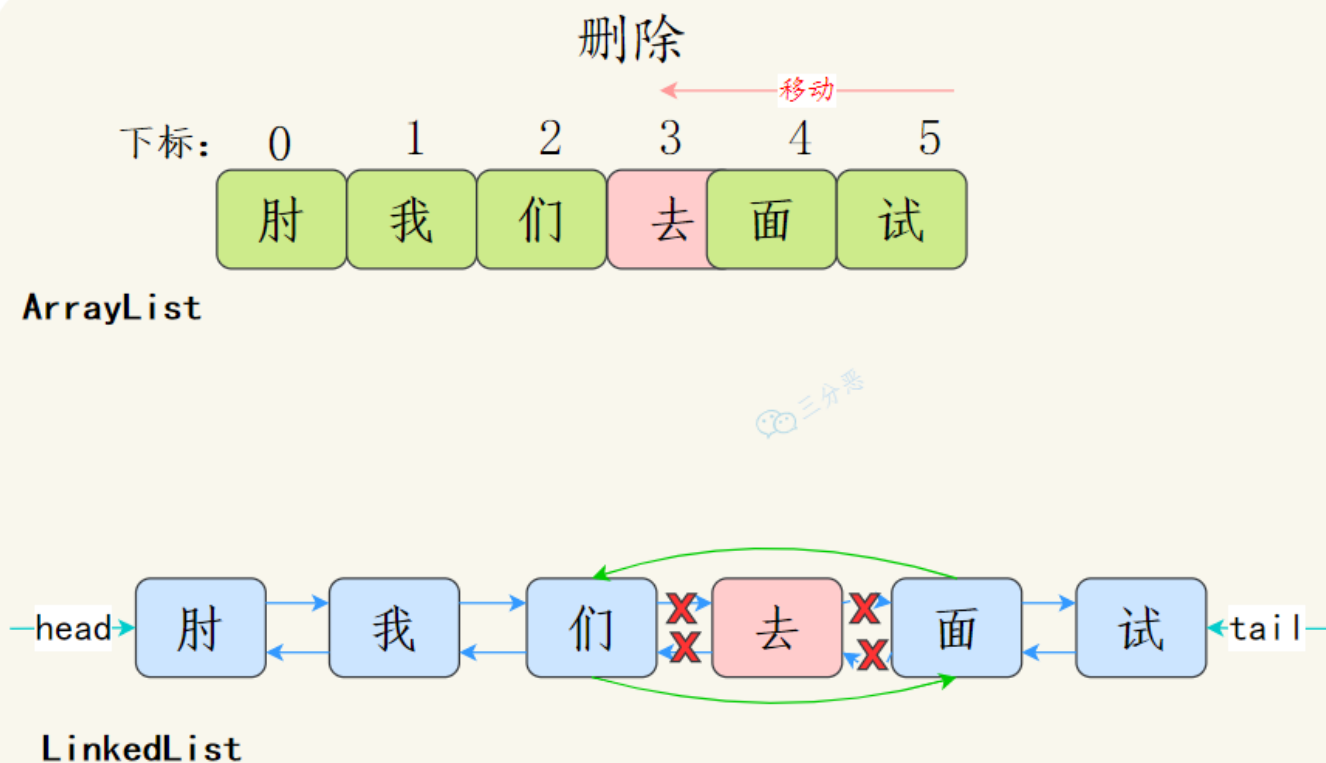


(2) 多数情况下，ArrayList更利于查找，LinkedList更利于增删

- ArrayList基于数组实现，`get(int index)`可以直接通过数组下标获取，时间复杂度是 $O(1)$ ；LinkedList基于链表实现，`get(int index)`需要遍历链表，时间复杂度是 $O(n)$ ；当然，`get(E element)`这种查找，两种集合都需要遍历，时间复杂度都是 $O(n)$ 。
- ArrayList增删如果是数组末尾的位置，直接插入或者删除就可以了，但是如果插入中间的位置，就需要把插入位置后的元素都向前或者向后移动，甚至还有可能触发扩容；双向链表的插入和删除只需要改变前驱节点、后继节点和插入节点的指向就行了，不需要移动元素。

插入





注意，这个地方可能会出陷阱，LinkedList更利于增删更多是体现在平均步长上，不是体现在时间复杂度上，二者增删的时间复杂度都是 $O(n)$

(3) 是否支持随机访问

- ArrayList基于数组，所以它可以根据下标查找，支持随机访问，当然，它也实现了RandmoAccess 接口，这个接口只是用来标识是否支持随机访问。
- LinkedList基于链表，所以它没法根据序号直接获取元素，它没有实现RandmoAccess 接口，标记不支持随机访问。

(4) 内存占用，ArrayList基于数组，是一块连续的内存空间，LinkedList基于链表，内存空间不连续，它们在空间占用上都有一些额外的消耗：

- ArrayList是预先定义好的数组，可能会有空的内存空间，存在一定空间浪费
- LinkedList每个节点，需要存储前驱和后继，所以每个节点会占用更多的空间

3.ArrayList的扩容机制了解吗？

ArrayList是基于数组的集合，数组的容量是在定义的时候确定的，如果数组满了，再插入，就会数组溢出。所以在插入时候，会先检查是否需要扩容，如果当前容量+1超过数组长度，就会进行扩容。

ArrayList的扩容是创建一个1.5倍的新数组，然后把原数组的值拷贝过去。



4.ArrayList怎么序列化的知道吗？为什么用transient修饰数组？

ArrayList的序列化不太一样，它使用 `transient` 修饰存储元素的 `elementData` 的数组，`transient` 关键字的作用是让被修饰的成员属性不被序列化。

为什么最ArrayList不直接序列化元素数组呢？

出于效率的考虑，数组可能长度100，但实际只用了50，剩下的50不用其实不用序列化，这样可以提高序列化和反序列化的效率，还可以节省内存空间。

那ArrayList怎么序列化呢？

ArrayList通过两个方法 `readObject`、`writeObject` 自定义序列化和反序列化策略，实际直接使用两个流 `ObjectOutputStream` 和 `ObjectInputStream` 来进行序列化和反序列化。

```
/**
 * 自定义序列化
 */
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException{
    // fail-fast, 后续判断是否有并发处理
```

```

int expectedModCount = modCount;
// 序列化没有标记为 static、transient 的字段，包括 size 等。
s.defaultWriteObject();

s.writeInt(size);

// 序列化数组的前size个元素
for (int i=0; i<size; i++) {
    s.writeObject(elementData[i]);
}

if (modCount != expectedModCount) {
    throw new ConcurrentModificationException();
}
}

/**
 * 自定义反序列化
 */
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {
    elementData = EMPTY_ELEMENTDATA;

    //反序列化没有标记为 static、transient 的字段，包括 size 等
    s.defaultReadObject();

    s.readInt(); // ignored

    if (size > 0) {
        // 数组扩容
        int capacity = calculateCapacity(elementData, size);
        SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, capacity);
        ensureCapacityInternal(size);

        Object[] a = elementData;
        // 反序列化元素并填充到数组中
        for (int i=0; i<size; i++) {
            a[i] = s.readObject();
        }
    }
}

```

5.快速失败(fail-fast)和安全失败(fail-safe)了解吗?

快速失败 (**fail-fast**)：快速失败是Java集合的一种错误检测机制

- 在用迭代器遍历一个集合对象时，如果线程A遍历过程中，线程B对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception。

- 原理：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 `modCount` 变量。集合在被遍历期间如果内容发生变化，就会改变 `modCount` 的值。每当迭代器使用 `hashNext()/next()` 遍历下一个元素之前，都会检测 `modCount` 变量是否为 `expectedmodCount` 值，是的话就返回遍历；否则抛出异常，终止遍历。
- 注意：这里异常的抛出条件是检测到 `modCount != expectedmodCount` 这个条件。如果集合发生变化时修改 `modCount` 值刚好又设置为了 `expectedmodCount` 值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的 bug。
- 场景：`java.util` 包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改），比如 `ArrayList` 类。

安全失败（fail-safe）

- 采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。
- 原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发 `Concurrent Modification Exception`。
- 缺点：基于拷贝内容的优点是避免了 `Concurrent Modification Exception`，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。
- 场景：`java.util.concurrent` 包下的容器都是安全失败，可以在多线程下并发使用，并发修改，比如 `CopyOnWriteArrayList` 类。

6. 有哪几种实现 `ArrayList` 线程安全的方法？

`fail-fast` 是一种可能触发的机制，实际上，`ArrayList` 的线程安全仍然没有保证，一般，保证 `ArrayList` 的线程安全可以通过这些方案：

- 使用 `Vector` 代替 `ArrayList`。（不推荐，`Vector` 是一个历史遗留类）
- 使用 `Collections.synchronizedList` 包装 `ArrayList`，然后操作包装后的 `list`。
- 使用 `CopyOnWriteArrayList` 代替 `ArrayList`。
- 在使用 `ArrayList` 时，应用程序通过同步机制去控制 `ArrayList` 的读写。

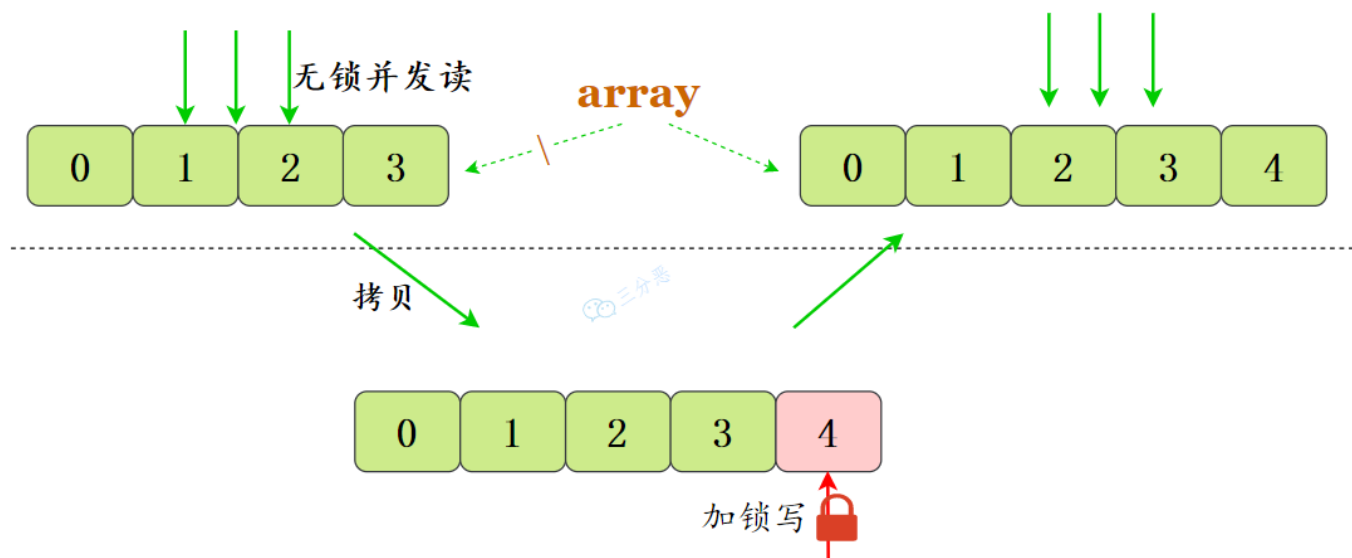
7. `CopyOnWriteArrayList` 了解多少？

`CopyOnWriteArrayList` 就是线程安全版本的 `ArrayList`。

它的名字叫 `CopyOnWrite` —— 写时复制，已经明示了它的原理。

`CopyOnWriteArrayList` 采用了一种读写分离的并发策略。`CopyOnWriteArrayList` 容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。

- ①将原数组拷贝一份
- ②写操作在副本上，加锁；此时读操作在原数组上
- ③写完将元素数组指向副本



关注沉默王二
学Java不迷路



Map

Map中，毫无疑问，最重要的就是HashMap，面试基本被盘出包浆了，各种问法，一定要好好准备。

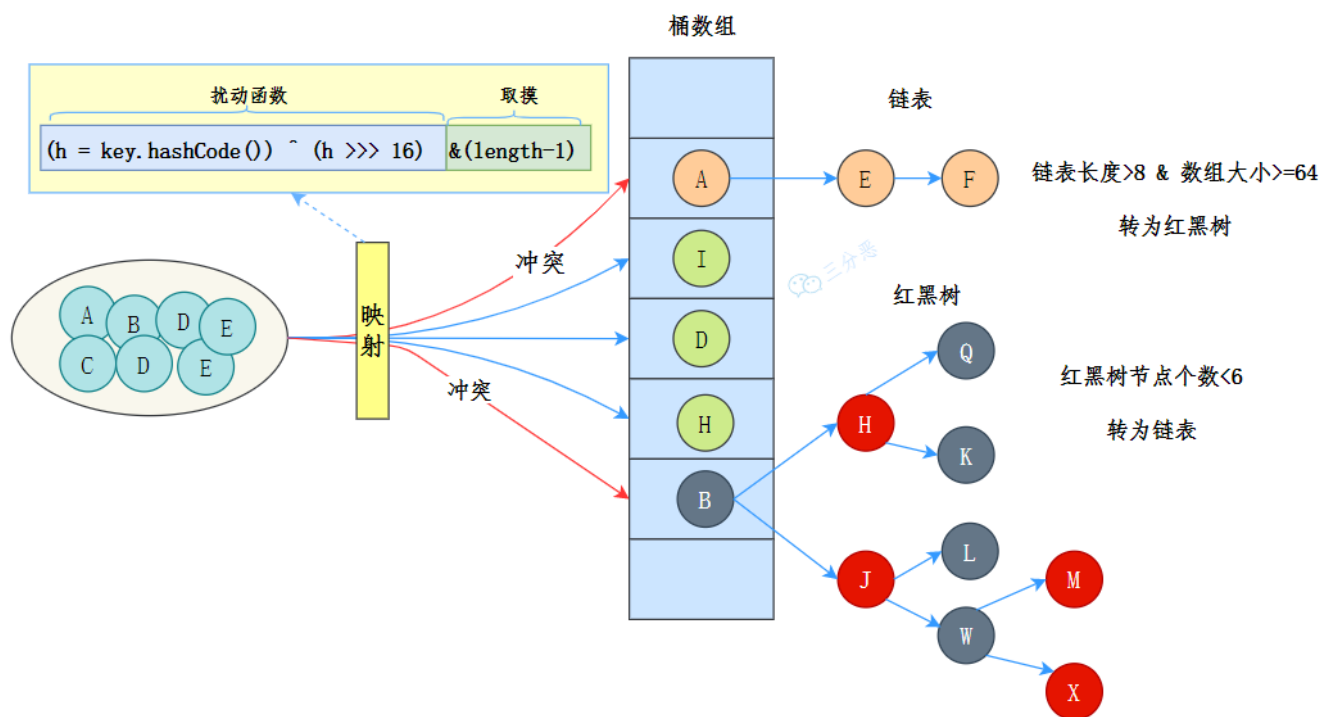
8.能说一下HashMap的数据结构吗？

JDK1.7的数据结构是 数组 + 链表 ，JDK1.7还有人在用？不会吧.....

说一下JDK1.8的数据结构吧：

JDK1.8的数据结构是 数组 + 链表 + 红黑树 。

数据结构示意图如下：



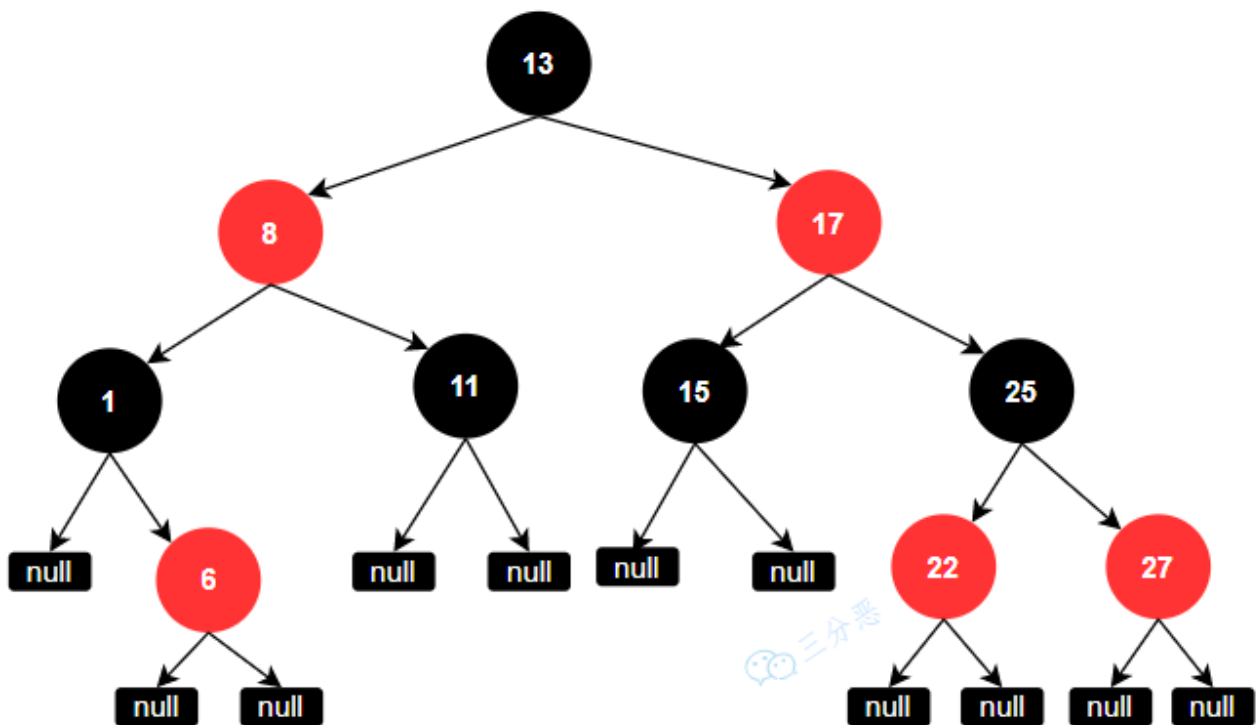
其中，桶数组是用来存储数据元素，链表是用来解决冲突，红黑树是为了提高查询的效率。

- 数据元素通过映射关系，也就是散列函数，映射到桶数组对应索引的位置
- 如果发生冲突，从冲突的位置拉一个链表，插入冲突的元素
- 如果链表长度>8&数组大小>=64，链表转为红黑树
- 如果红黑树节点个数<6，转为链表

9.你对红黑树了解多少？为什么不用二叉树/平衡树呢？

红黑树本质上是一种二叉查找树，为了保持平衡，它又在二叉查找树的基础上增加了一些规则：

1. 每个节点要么是红色，要么是黑色；
2. 根节点永远是黑色的；
3. 所有的叶子节点都是黑色的（注意这里说叶子节点其实是图中的 NULL 节点）；
4. 每个红色节点的两个子节点一定都是黑色；
5. 从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点；



之所以不用二叉树：

红黑树是一种平衡的二叉树，插入、删除、查找的最坏时间复杂度都为 $O(\log n)$ ，避免了二叉树最坏情况下的 $O(n)$ 时间复杂度。

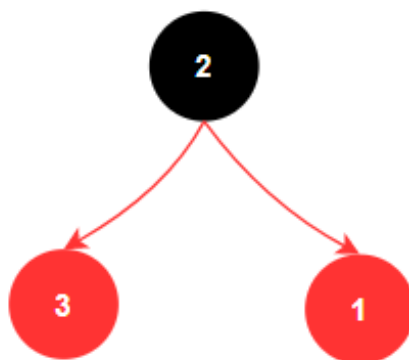
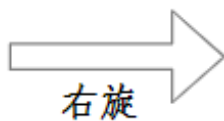
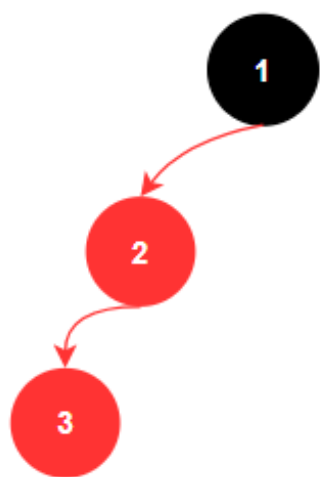
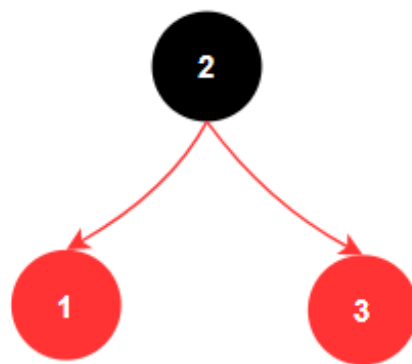
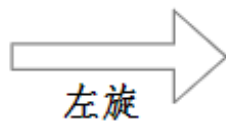
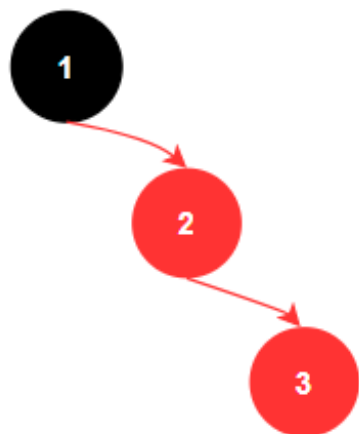
之所以不用平衡二叉树：

平衡二叉树是比红黑树更严格的平衡树，为了保持平衡，需要旋转的次数更多，也就是说平衡二叉树保持平衡的效率更低，所以平衡二叉树插入和删除的效率比红黑树要低。

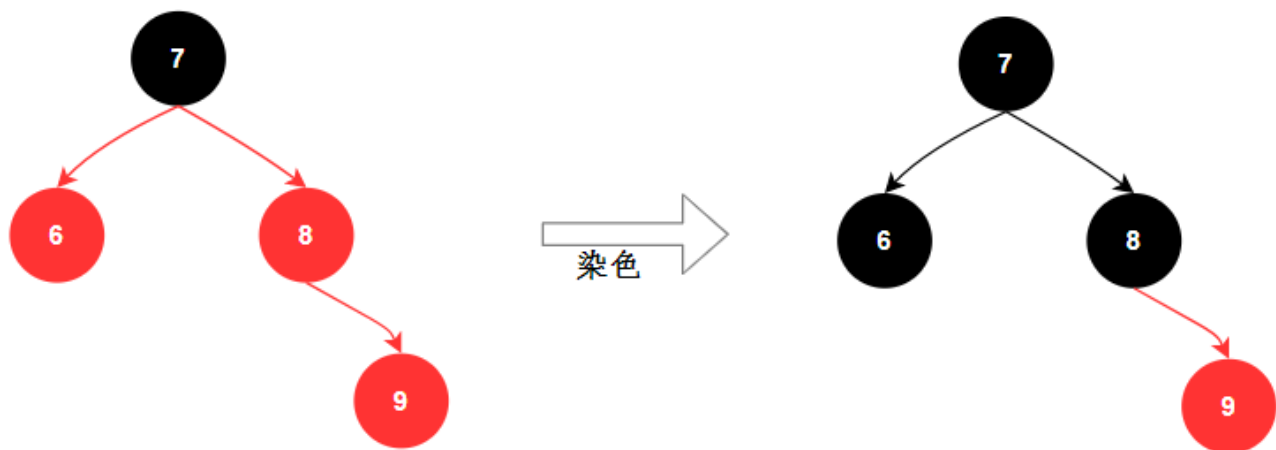
10. 红黑树怎么保持平衡的知道吗？

红黑树有两种方式保持平衡：旋转 和 染色。

- 旋转：旋转分为两种，左旋和右旋



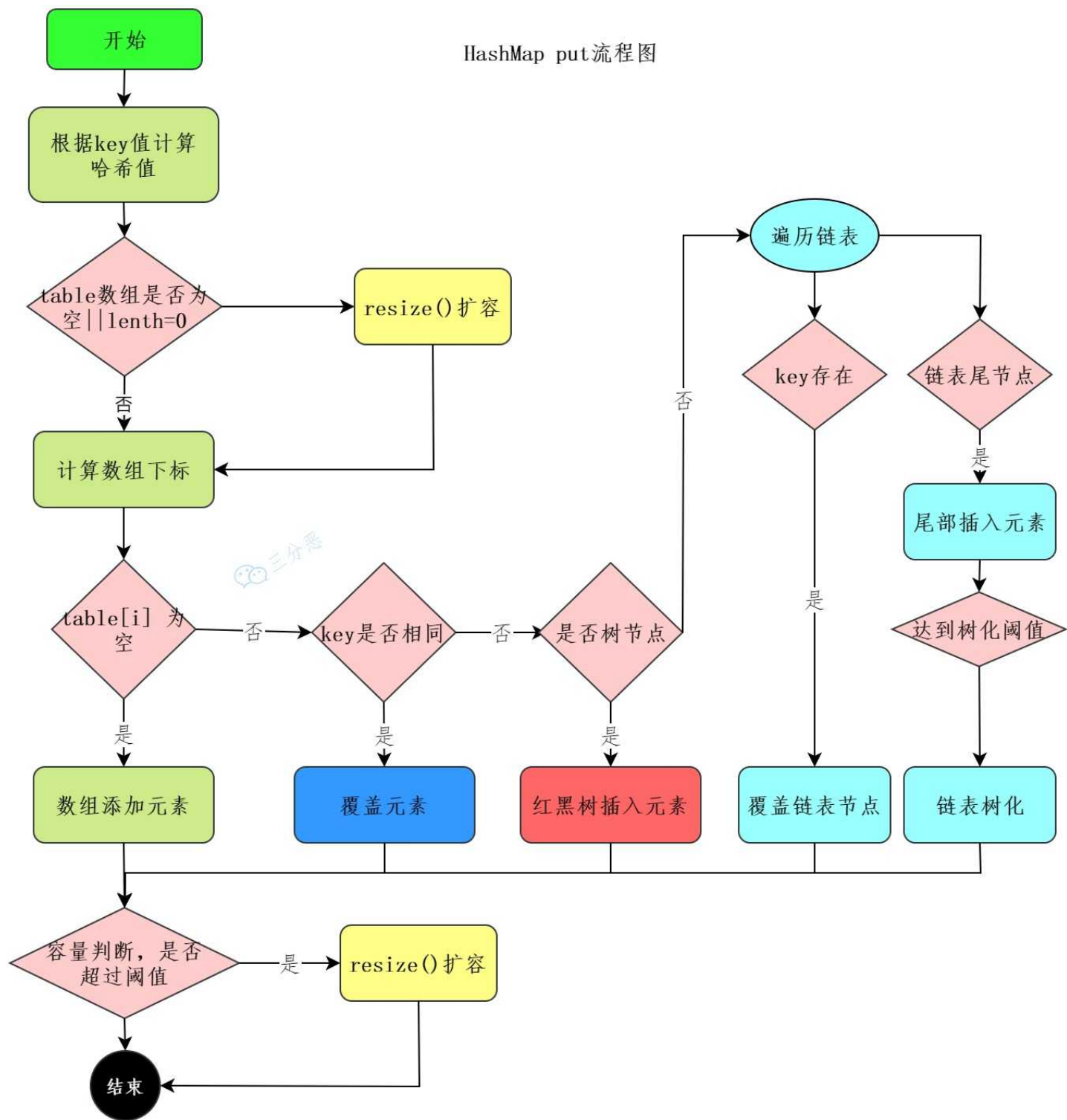
- 染色：



11.HashMap的put流程知道吗？

先上个流程图吧：

HashMap put流程图



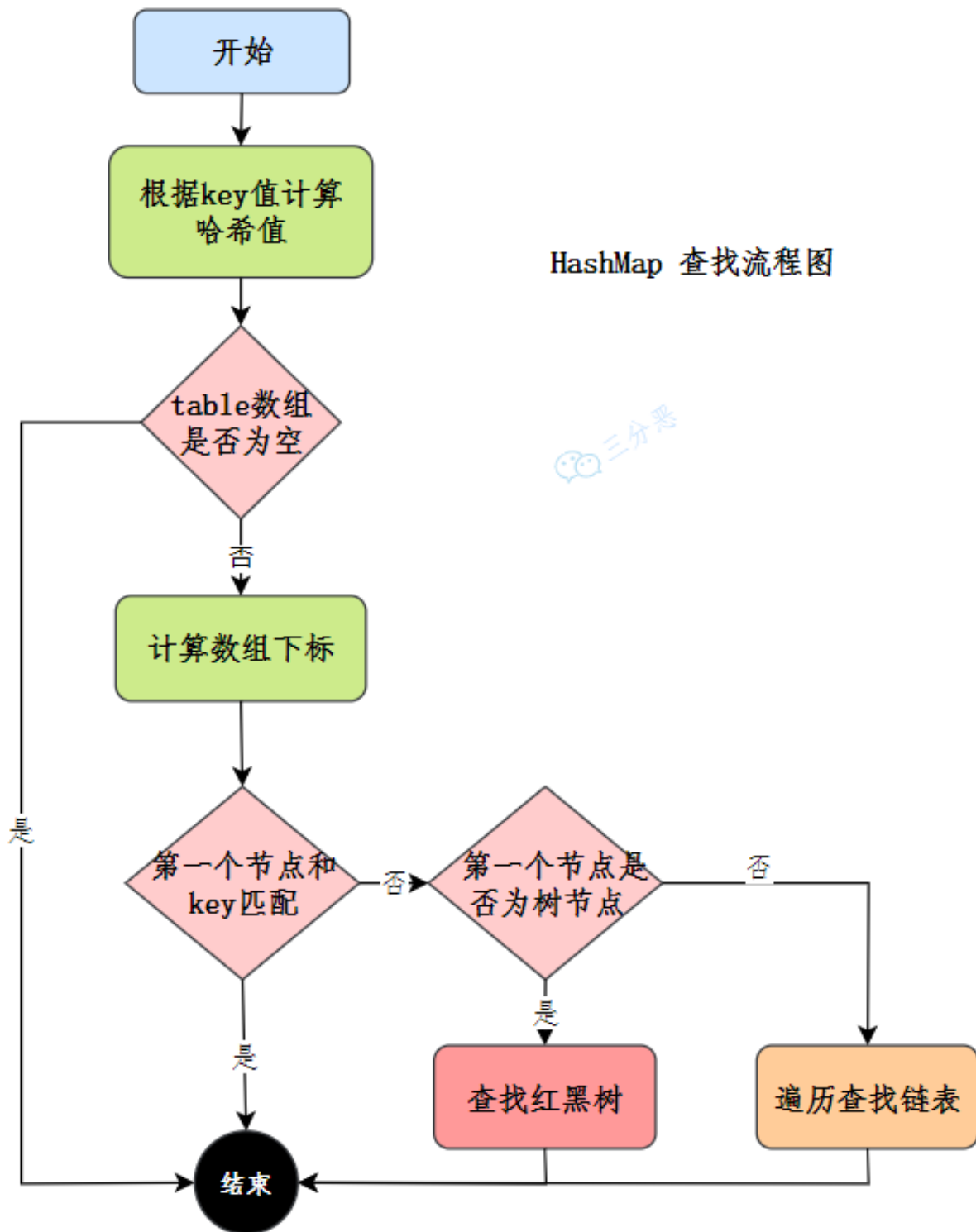
1. 首先进行哈希值的扰动，获取一个新的哈希值。 `(key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);`
2. 判断tab是否位空或者长度为0，如果是则进行扩容操作。

```
if ((tab = table) == null || (n = tab.length) == 0)
    n = (tab = resize()).length;
```

3. 根据哈希值计算下标，如果对应小标正好没有存放数据，则直接插入即可否则需要覆盖。 `tab[i = (n - 1) & hash]`
4. 判断`tab[i]`是否为树节点，否则向链表中插入数据，是则向树中插入节点。
5. 如果链表中插入节点的时候，链表长度大于等于8，则需要把链表转换为红黑树。 `treeifyBin(tab, hash);`
6. 最后所有元素处理完成后，判断是否超过阈值； `threshold`，超过则扩容。

| 12.HashMap怎么查找元素的呢？

先看流程图：



HashMap的查找就简单很多：

1. 使用扰动函数，获取新的哈希值
2. 计算数组下标，获取节点

3. 当前节点和key匹配，直接返回
4. 否则，当前节点是否为树节点，查找红黑树
5. 否则，遍历链表查找

13.HashMap的哈希/扰动函数是怎么设计的？

HashMap的哈希函数是先拿到 key 的 hashCode，是一个32位的int类型的数值，然后让hashCode的高16位和低16位进行异或操作。

```
static final int hash(Object key) {
    int h;
    // key的hashCode和key的hashCode右移16位做异或运算
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

这么设计是为了降低哈希碰撞的概率。

14.为什么哈希/扰动函数能降hash碰撞？

因为 key.hashCode() 函数调用的是 key 键值类型自带的哈希函数，返回 int 型散列值。int 值范围为 -2147483648~2147483647，加起来大概 40 亿的映射空间。

只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个 40 亿长度的数组，内存是放不下的。

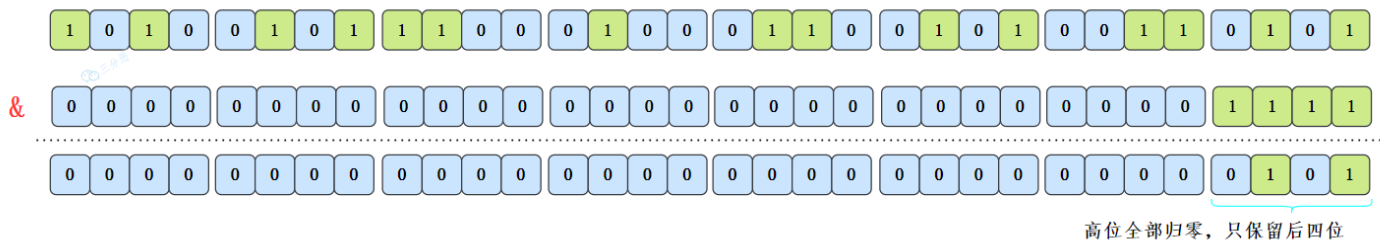
假如 HashMap 数组的初始大小才 16，就需要用之前需要对数组的长度取模运算，得到的余数才能用来访问数组下标。

源码中模运算就是把散列值和数组长度 - 1 做一个 "与&" 操作，位运算比取余 % 运算要快。

```
bucketIndex = indexFor(hash, table.length);

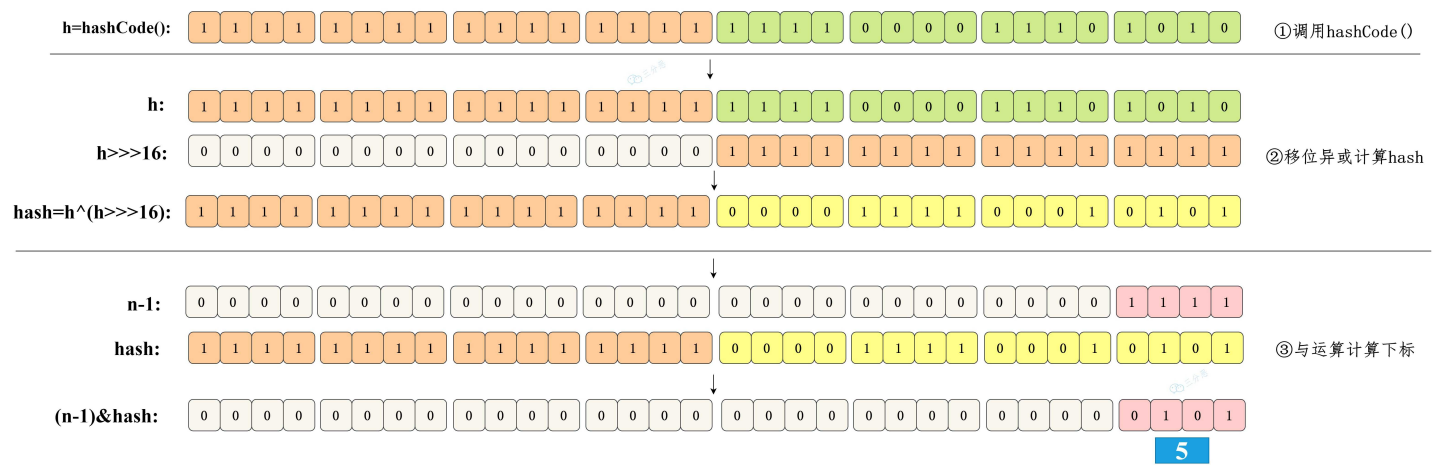
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

顺便说一下，这也正好解释了为什么 HashMap 的数组长度要取 2 的整数幂。因为这样（数组长度 - 1）正好相当于一个“低位掩码”。与 操作的结果就是散列值的高位全部归零，只保留低位值，用来做数组下标访问。以初始长度 16 为例，16-1=15。2 进制表示是 0000 0000 0000 0000 0000 0000 0000 1111。和某个散列值做 与 操作如下，结果就是截取了最低的四位值。



这样是要快捷一些，但是新的问题来了，就算散列值分布再松散，要是只取最后几位的话，碰撞也会很严重。如果散列本身做得不好，分布上成等差数列的漏洞，如果正好让最后几个低位呈现规律性重复，那就更难搞了。

这时候 扰动函数 的价值就体现出来了，看一下扰动函数的示意图：



右移 16 位，正好是 32bit 的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

15.为什么HashMap的容量是2的倍数呢？

- 第一个原因是为了方便哈希取余：

将元素放在table数组上面，是用hash值%数组大小定位位置，而HashMap是用hash值&(数组大小-1)，却能和前面达到一样的效果，这就得益于HashMap的大小是2的倍数，2的倍数意味着该数的二进制位只有一位为1，而该数-1就可以得到二进制位上1变成0，后面的0变成1，再通过&运算，就可以得到和%一样的效果，并且位运算比%的效率 high 得多

HashMap的容量是2的n次幂时，(n-1)的2进制也就是1111111***111这样形式的，这样与添加元素的hash值进行位运算时，能够充分的散列，使得添加的元素均匀分布在HashMap的每个位置上，减少hash碰撞。
- 第二个方面是在扩容时，利用扩容后的大小也是2的倍数，将已经产生hash碰撞的元素完美的转移到新的table中去

我们可以简单看看HashMap的扩容机制，HashMap中的元素在超过 `负载因子*HashMap` 大小时就会产生扩容。

```
++modCount;
if (++size > threshold)
    resize();
afterNodeInsertion(evict);
return null;
```

put时，当大小超过threshold，就会扩容

16.如果初始化HashMap，传一个17的值 `new HashMap<>`，它会怎么处理？

简单来说，就是初始化时，传的不是2的倍数时，HashMap会向上寻找 离得最近的2的倍数，所以传入17，但HashMap的实际容量是32。

我们来看看详情，在HashMap的初始化中，有这样一段方法：

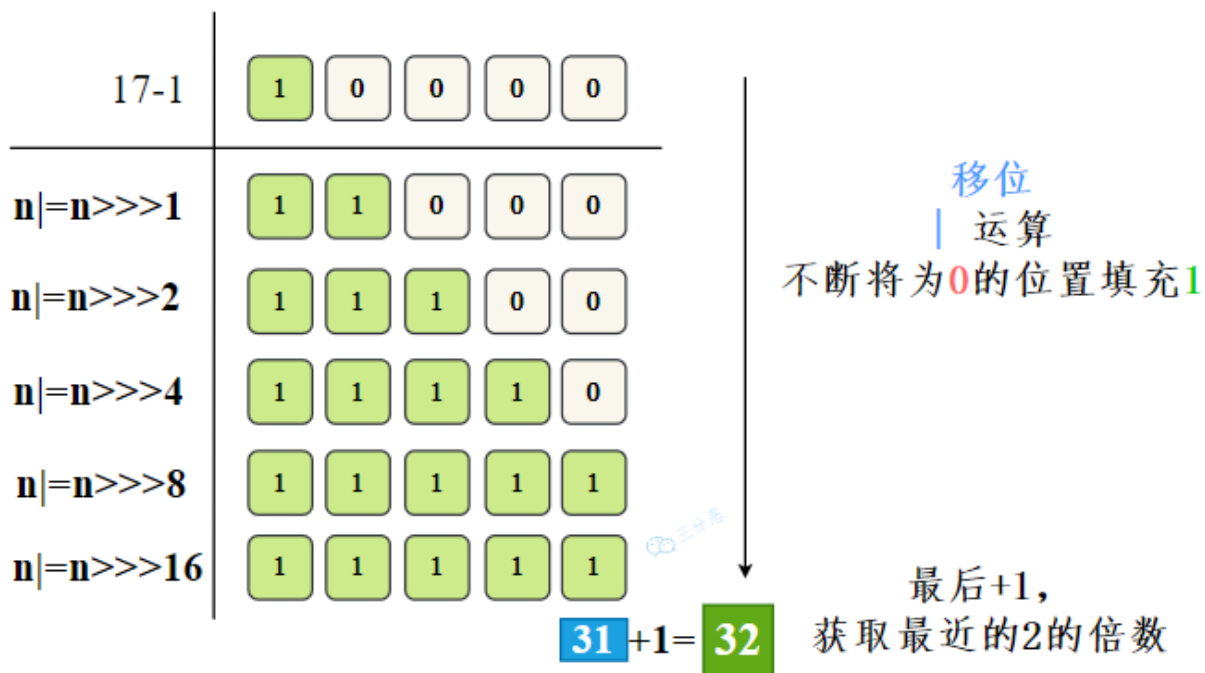
```
public HashMap(int initialCapacity, float loadFactor) {
    ...
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

- 阈值 `threshold`，通过方法 `tableSizeFor` 进行计算，是根据初始化传的参数来计算的。
- 同时，这个方法也要寻找比初始值大的，最小的那个2进制数值。比如传了17，我应该找到的是32。

```
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

- `MAXIMUM_CAPACITY = 1 << 30`，这个是临界范围，也就是最大的Map集合。
- 计算过程是向右移位1、2、4、8、16，和原来的数做 `|` 运算，这主要是为了把二进制的各个位置都填上1，当二进制的各个位置都是1以后，就是一个标准的2的倍数减1了，最后把结果加1再返回即可。

以17为例，看一下初始化计算table容量的过程：



17.你还知道哪些哈希函数的构造方法呢？

HashMap里哈希构造函数的方法叫：

- 除留取余法： $H(key) = key \% p$ ($p \leq N$)，关键字除以一个不大于哈希表长度的正整数 p ，所得余数为地址，当然HashMap里进行了优化改造，效率更高，散列也更均衡。

除此之外，还有这几种常见的哈希函数构造方法：

- 直接定址法

直接根据 `key` 来映射到对应的数组位置，例如1232放到下标1232的位置。

- 数字分析法

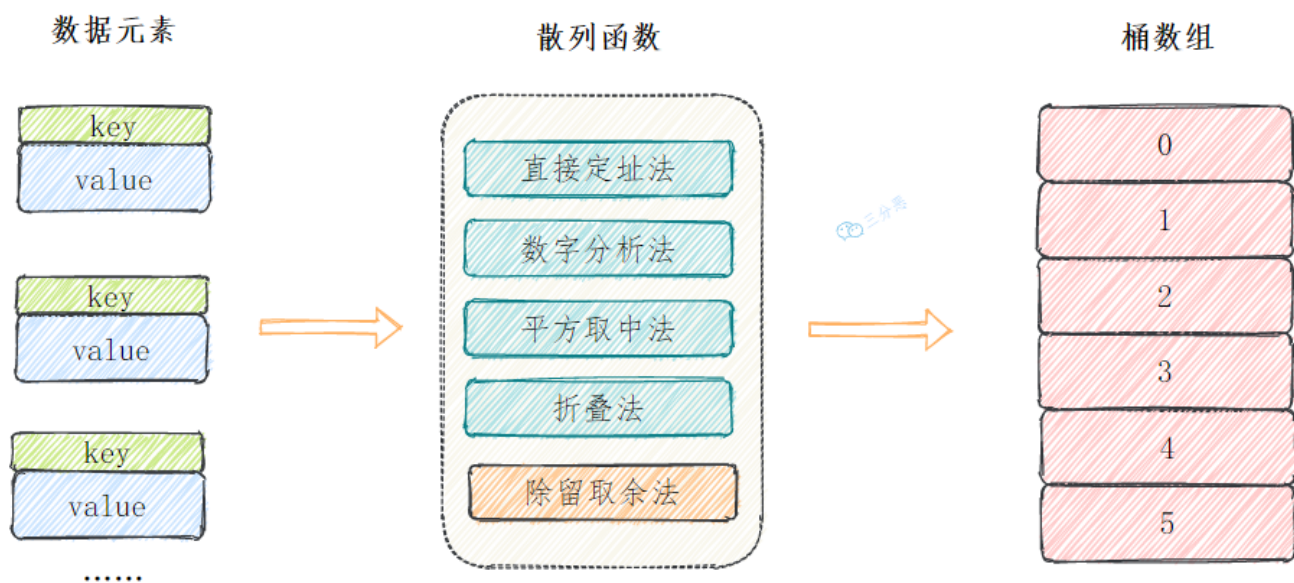
取 `key` 的某些数字（例如十位和百位）作为映射的位置

- 平方取中法

取 `key` 平方的中间几位作为映射的位置

- 折叠法

将 `key` 分割成位数相同的几段，然后把它们的叠加和作为映射的位置



18. 解决哈希冲突有哪些方法呢？

我们到现在已经知道，HashMap使用链表的原因为了处理哈希冲突，这种方法就是所谓的：

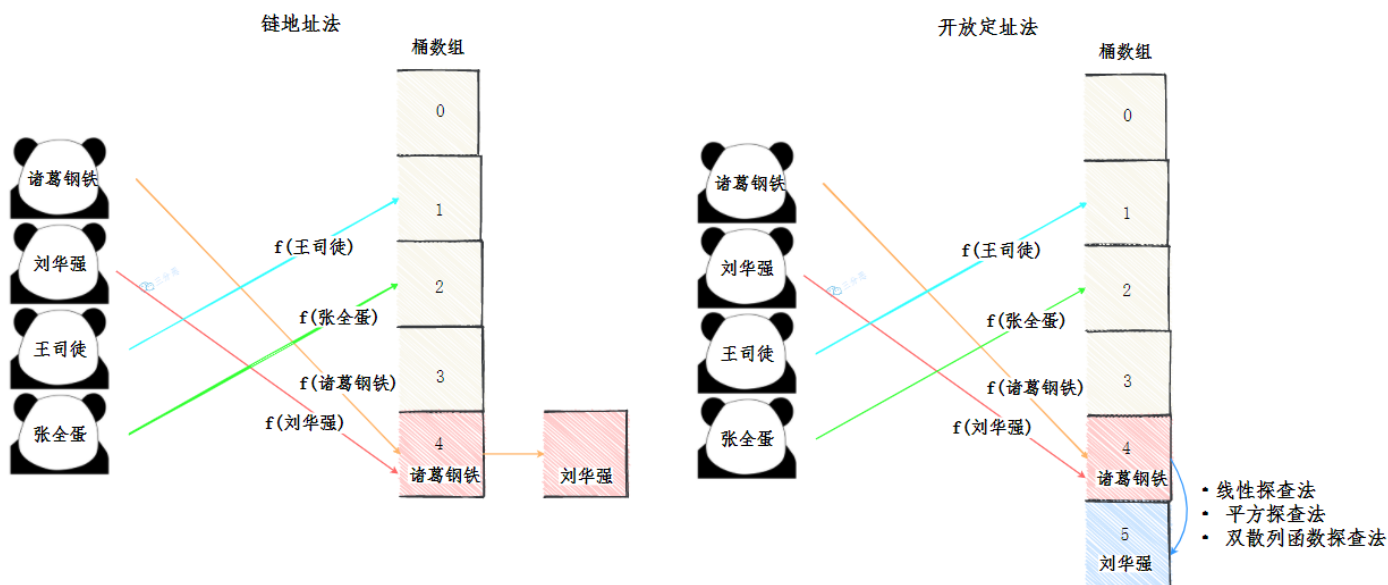
- 链地址法：在冲突的位置拉一个链表，把冲突的元素放进去。

除此之外，还有一些常见的解决冲突的办法：

- 开放定址法：开放定址法就是从冲突的位置再接着往下找，给冲突元素找个空位。

找到空闲位置的方法也有很多种：

- 线行探查法：从冲突的位置开始，依次判断下一个位置是否空闲，直至找到空闲位置
- 平方探查法：从冲突的位置 x 开始，第一次增加 1^2 个位置，第二次增加 2^2 ...，直至找到空闲的位置
-



- 再哈希法：换种哈希函数，重新计算冲突元素的地址。
- 建立公共溢出区：再建一个数组，把冲突的元素放进去。

19.为什么HashMap链表转红黑树的阈值为8呢？

树化发生在table数组的长度大于64，且链表的长度大于8的时候。

为什么是8呢？源码的注释也给出了答案。

```
*
* Because TreeNodes are about twice the size of regular nodes, we
* use them only when bins contain enough nodes to warrant use
* (see TREEIFY_THRESHOLD). And when they become too small (due to
* removal or resizing) they are converted back to plain bins. In
* usages with well-distributed user hashCodes, tree bins are
* rarely used. Ideally, under random hashCodes, the frequency of
* nodes in bins follows a Poisson distribution
* (http://en.wikipedia.org/wiki/Poisson\_distribution) with a
* parameter of about 0.5 on average for the default resizing
* threshold of 0.75, although with a large variance because of
* resizing granularity. Ignoring variance, the expected
* occurrences of list size k are (exp(-0.5) * pow(0.5, k) /
* factorial(k)). The first values are:
*
* 0:      0.60653066
* 1:      0.30326533
* 2:      0.07581633
* 3:      0.01263606
* 4:      0.00157952
* 5:      0.00015795
* 6:      0.00001316
* 7:      0.00000094
* 8:      0.00000006
* more: less than 1 in ten million
```

红黑树节点的大小大概是普通节点大小的两倍，所以转红黑树，牺牲了空间换时间，更多的是一种兜底的策略，保证极端情况下的查找效率。

阈值为什么要选8呢？和统计学有关。理想情况下，使用随机哈希码，链表里的节点符合泊松分布，出现节点个数的概率是递减的，节点个数为8的情况，发生概率仅为 **0.00000006**。

至于红黑树转回链表的阈值为什么是6，而不是8？是因为如果这个阈值也设置成8，假如发生碰撞，节点增减刚好在8附近，会发生链表和红黑树的不断转换，导致资源浪费。

20.扩容在什么时候呢？为什么扩容因子是0.75？

为了减少哈希冲突发生的概率，当当前HashMap的元素个数达到一个临界值的时候，就会触发扩容，把所有元素rehash之后再放在扩容后的容器中，这是一个相当耗时的操作。

```
if (++size > threshold)    size达到临界值，就扩容
    resize();
```

而这个 临界值threshold 就是由加载因子和当前容器的容量大小来确定的，假如采用默认的构造方法：

临界值 (threshold) = 默认容量 (DEFAULT_INITIAL_CAPACITY) * 默认扩容因子 (DEFAULT_LOAD_FACTOR)

```
else { // zero initial threshold signifies using defaults
    newCap = DEFAULT_INITIAL_CAPACITY;
    newThr = (int) (DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
}
if (newThr == 0) { // 临界值计算
    float ft = (float) newCap * loadFactor;
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float) MAXIMUM_CAPACITY ?
        (int) ft : Integer.MAX_VALUE);
}
threshold = newThr;
```

那就是大于 $16 \times 0.75 = 12$ 时，就会触发扩容操作。

那么为什么选择了0.75作为HashMap的默认加载因子呢？

简单来说，这是对 空间 成本和 时间 成本平衡的考虑。

在HashMap中有这样一段注释：

```
*
* <p>As a general rule, the default load factor (.75) offers a good
* tradeoff between time and space costs. Higher values decrease the
* space overhead but increase the lookup cost (reflected in most of
* the operations of the <tt>HashMap</tt> class, including
* <tt>get</tt> and <tt>put</tt>). The expected number of entries in
* the map and its load factor should be taken into account when
* setting its initial capacity, so as to minimize the number of
* rehash operations. If the initial capacity is greater than the
* maximum number of entries divided by the load factor, no rehash
* operations will ever occur.
*
```

我们都知道，HashMap的散列构造方式是Hash取余，负载因子决定元素个数达到多少时候扩容。

假如我们设的比较大，元素比较多，空位比较少的时候才扩容，那么发生哈希冲突的概率就增加了，查找的时间成本就增加了。

我们设的比较大的话，元素比较少，空位比较多的时候就扩容了，发生哈希碰撞的概率就降低了，查找时间成本降低，但是就需要更多的空间去存储元素，空间成本就增加了。

21.那扩容机制了解吗？

HashMap是基于数组+链表和红黑树实现的，但用于存放key值的桶数组的长度是固定的，由初始化参数确定。

那么，随着数据的插入数量增加以及负载因子的作用下，就需要扩容来存放更多的数据。而扩容中有一个非常重要的点，就是jdk1.8中的优化操作，可以不需要再重新计算每一个元素的哈希值。

因为HashMap的初始容量是2的次幂，扩容之后的长度是原来的二倍，新的容量也是2的次幂，所以，元素，要么在原位置，要么在原位置再移动2的次幂。

看下这张图，n为table的长度，图 a 表示扩容前的key1和key2两种key确定索引的位置，图 b 表示扩容后key1和key2两种key确定索引位置。

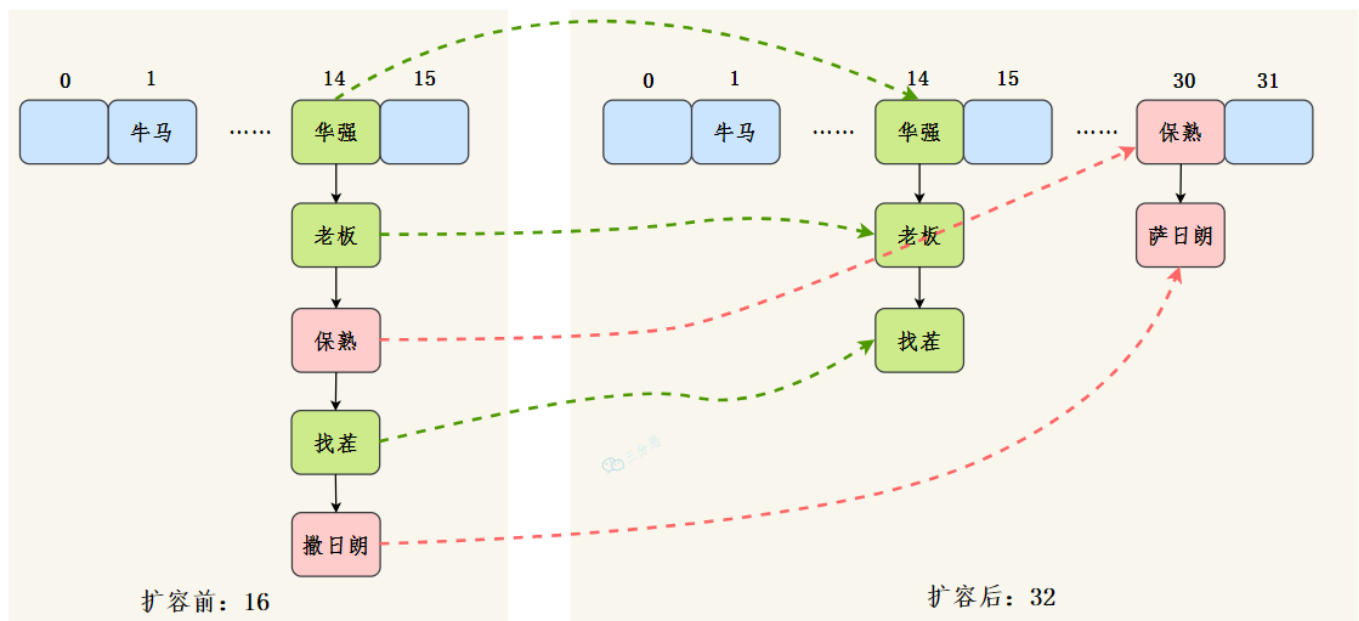


元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：

$$\begin{array}{ccc}
 0101=5 & \xrightarrow[16 \Rightarrow 2*16]{\text{resize}} & \begin{array}{l} 0 \ 0101 = 5 \\ 1 \ 0101 = 21 = 5 + 16 \end{array}
 \end{array}$$

原位置
原位置+oldCap

所以在扩容时，只需要看原来的hash值新增的那一位是0还是1就行了，是0的话索引没变，是1的化变成 原索引+oldCap，看看如16扩容为32的示意图：



扩容节点迁移主要逻辑：

```
if (e.next == null)
    //未形成链表，直接分配到新table
    newTab[e.hash & (newCap - 1)] = e;
else if (e instanceof TreeNode)
    //树节点分割
    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
else { // preserve order
    //链表拆成两个链表lo、hi，两个链表的头、尾节点
    Node<K,V> loHead = null, loTail = null;
    Node<K,V> hiHead = null, hiTail = null;
    Node<K,V> next;
    do {
        //遍历链表，将节点放到相应的新链表
        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);
    //lo链表放到新table原位置
    if (loTail != null) {
        loTail.next = null;
        newTab[j] = loHead;
    }
    //hi链表放到j+oldCap位置
```

```
if (hiTail  $\neq$  null) {  
    hiTail.next = null;  
    newTab[j + oldCap] = hiHead;  
}
```

22.jdk1.8对HashMap主要做了哪些优化呢？为什么？

jdk1.8 的HashMap主要有五点优化：

1. 数据结构：数组 + 链表改成了数组 + 链表或红黑树

原因：发生 hash 冲突，元素会存入链表，链表过长转为红黑树，将时间复杂度由 $O(n)$ 降为 $O(\log n)$

2. 链表插入方式：链表的插入方式从头插法改成了尾插法

简单说就是插入时，如果数组位置上已经有元素，1.7 将新元素放到数组中，原始节点作为新节点的后继节点，1.8 遍历链表，将元素放置到链表的最后。

原因：因为 1.7 头插法扩容时，头插法会使链表发生反转，多线程环境下会产生环。

3. 扩容rehash：扩容的时候 1.7 需要对原数组中的元素进行重新 hash 定位在新数组的位置，1.8 采用更简单的判断逻辑，不需要重新通过哈希函数计算位置，新的位置不变或索引 + 新增容量大小。

原因：提高扩容的效率，更快地扩容。

4. 扩容时机：在插入时，1.7 先判断是否需要扩容，再插入，1.8 先进行插入，插入完成再判断是否需要扩容；

5. 散列函数：1.7 做了四次移位和四次异或，jdk1.8只做一次。

原因：做 4 次的话，边际效用也不大，改为一次，提升效率。

23.你能自己设计实现一个HashMap吗？

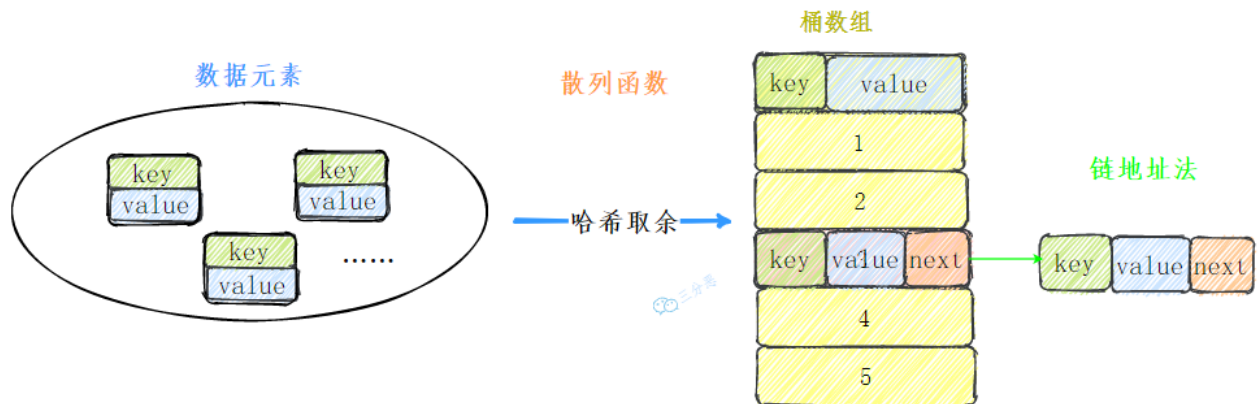
这道题快手常考。

不要慌，红黑树版咱们多半是写不出来，但是数组+链表版还是问题不大的，详细可见：[手写HashMap，快手面试官直呼内行！](#)。

整体的设计：

- 散列函数：hashCode()+除留余数法

- 冲突解决：链地址法
- 扩容：节点重新hash获取位置



完整代码：

```
/**
 * @Author 三分恶
 * @Date 2021/11/21
 * @Description 自己实现HashMap
 */
public class ThirdHashMap<K, V> {

    /**
     * 节点类
     */
    @param <K>
    @param <V>
    */
    class Node<K, V> {
        //键值对
        private K key;
        private V value;

        //链表，后继
        private Node<K, V> next;

        public Node(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

```

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

    //默认容量
    final int DEFAULT_CAPACITY = 16;
    //负载因子
    final float LOAD_FACTOR = 0.75f;
    //HashMap的大小
    private int size;
    //桶数组
    Node<K, V>[] buckets;

    /**
     * 无参构造器，设置桶数组默认容量
     */
    public ThirdHashMap() {
        buckets = new Node[DEFAULT_CAPACITY];
        size = 0;
    }

    /**
     * 有参构造器，指定桶数组容量
     *
     * @param capacity
     */
    public ThirdHashMap(int capacity) {
        buckets = new Node[capacity];
        size = 0;
    }

    /**
     * 哈希函数，获取地址
     *
     * @param key
     * @return
     */
    private int getIndex(K key, int length) {
        //获取hash code
        int hashCode = key.hashCode();
        //和桶数组长度取余
        int index = hashCode % length;
        return Math.abs(index);
    }

    /**
     * put方法

```

```

*
* @param key
* @param value
* @return
*/
public void put(K key, V value) {
    //判断是否需要进行扩容
    if (size >= buckets.length * LOAD_FACTOR) resize();
    putVal(key, value, buckets);
}

/**
 * 将元素存入指定的node数组
 *
 * @param key
 * @param value
 * @param table
 */
private void putVal(K key, V value, Node<K, V>[] table) {
    //获取位置
    int index = getIndex(key, table.length);
    Node node = table[index];
    //插入的位置为空
    if (node == null) {
        table[index] = new Node<>(key, value);
        size++;
        return;
    }
    //插入位置不为空，说明发生冲突，使用链地址法，遍历链表
    while (node != null) {
        //如果key相同，就覆盖掉
        if ((node.key.hashCode() == key.hashCode())
            && (node.key == key || node.key.equals(key))) {
            node.value = value;
            return;
        }
        node = node.next;
    }
    //当前key不在链表中，插入链表头部
    Node newNode = new Node(key, value, table[index]);
    table[index] = newNode;
    size++;
}

/**
 * 扩容
 */
private void resize() {
    //创建一个两倍容量的桶数组
    Node<K, V>[] newBuckets = new Node[buckets.length * 2];
}

```

```

        //将当前元素重新散列到新的桶数组
        rehash(newBuckets);
        buckets = newBuckets;
    }

    /**
     * 重新散列当前元素
     *
     * @param newBuckets
     */
    private void rehash(Node<K, V>[] newBuckets) {
        //map大小重新计算
        size = 0;
        //将旧的桶数组的元素全部刷到新的桶数组里
        for (int i = 0; i < buckets.length; i++) {
            //为空, 跳过
            if (buckets[i] == null) {
                continue;
            }
            Node<K, V> node = buckets[i];
            while (node != null) {
                //将元素放入新数组
                putVal(node.key, node.value, newBuckets);
                node = node.next;
            }
        }
    }

    /**
     * 获取元素
     *
     * @param key
     * @return
     */
    public V get(K key) {
        //获取key对应的地址
        int index = getIndex(key, buckets.length);
        if (buckets[index] == null) return null;
        Node<K, V> node = buckets[index];
        //查找链表
        while (node != null) {
            if ((node.key.hashCode() == key.hashCode())
                && (node.key == key || node.key.equals(key))) {
                return node.value;
            }
            node = node.next;
        }
        return null;
    }
}

```

```
/**
 * 返回HashMap大小
 *
 * @return
 */
public int size() {
    return size;
}

}
```

24.HashMap 是线程安全的吗？多线程下会有什么问题？

HashMap不是线程安全的，可能会发生这些问题：

- 多线程下扩容死循环。JDK1.7 中的 HashMap 使用头插法插入元素，在多线程的环境下，扩容的时候有可能导致环形链表的出现，形成死循环。因此，JDK1.8 使用尾插法插入元素，在扩容时会保持链表元素原本的顺序，不会出现环形链表的问题。
- 多线程的 put 可能导致元素的丢失。多线程同时执行 put 操作，如果计算出来的索引位置是相同的，那会造成前一个 key 被后一个 key 覆盖，从而导致元素的丢失。此问题在 JDK 1.7 和 JDK 1.8 中都存在。
- put 和 get 并发时，可能导致 get 为 null。线程 1 执行 put 时，因为元素个数超出 threshold 而导致 rehash，线程 2 此时执行 get，有可能导致这个问题。这个问题在 JDK 1.7 和 JDK 1.8 中都存在。

25.有什么办法能解决HashMap线程不安全的问题呢？

Java 中有 HashTable、Collections.synchronizedMap、以及 ConcurrentHashMap 可以实现线程安全的 Map。

- HashTable 是直接操作方法上加 synchronized 关键字，锁住整个table数组，粒度比较大；
- Collections.synchronizedMap 是使用 Collections 集合工具的内部类，通过传入 Map 封装出一个 SynchronizedMap 对象，内部定义了一个对象锁，方法内通过对象锁实现；
- ConcurrentHashMap 在jdk1.7中使用分段锁，在jdk1.8中使用CAS+synchronized。

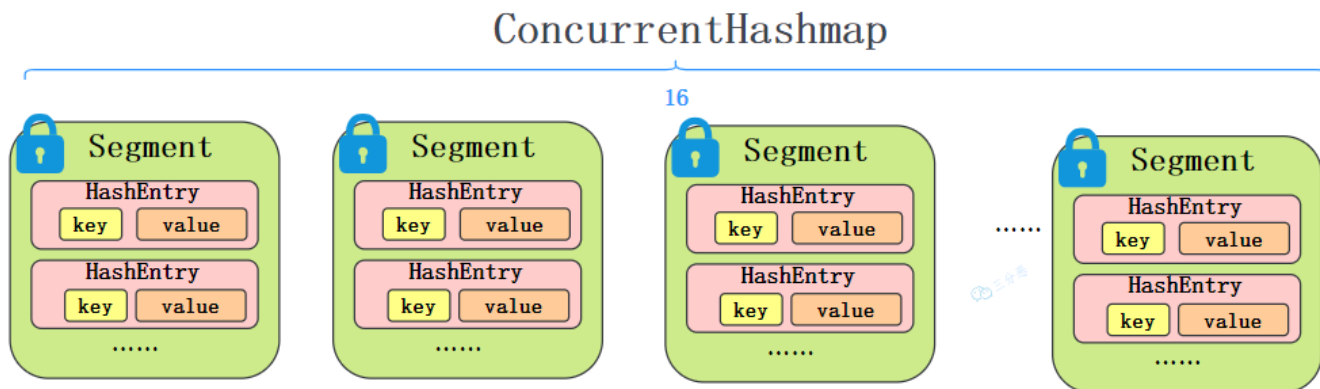
26.能具体说一下ConcurrentHashmap的实现吗？

ConcurrentHashmap线程安全在jdk1.7版本是基于 分段锁 实现，在jdk1.8是基于 CAS+synchronized 实现。

1.7分段锁

从结构上说，1.7版本的ConcurrentHashMap采用分段锁机制，里面包含一个Segment数组，Segment继承于ReentrantLock，Segment则包含HashEntry的数组，HashEntry本身就是一个链表的结构，具有保存key、value的能力能指向下一个节点的指针。

实际上就是相当于每个Segment都是一个HashMap，默认的Segment长度是16，也就是支持16个线程的并发写，Segment之间相互不会受到影响。



put流程

整个流程和HashMap非常类似，只不过是先定位到具体的Segment，然后通过ReentrantLock去操作而已，后面的流程，就和HashMap基本上是一样的。

1. 计算hash，定位到segment，segment如果是空就先初始化
2. 使用ReentrantLock加锁，如果获取锁失败则尝试自旋，自旋超过次数就阻塞获取，保证一定获取锁成功
3. 遍历HashEntry，就是和HashMap一样，数组中key和hash一样就直接替换，不存在就再插入链表，链表同样操作

get流程

get也很简单，key通过hash定位到segment，再遍历链表定位到具体的元素上，需要注意的是value是volatile的，所以get是不需要加锁的。

1.8 CAS+synchronized

jdk1.8实现线程安全不是在数据结构上下功夫，它的数据结构和HashMap是一样的，数组+链表+红黑树。它实现线程安全的关键点在于put流程。

put流程

1. 首先计算hash，遍历node数组，如果node是空的话，就通过CAS+自旋的方式初始化

```
tab = initTable();
```

node数组初始化：

```
private final Node<K,V>[] initTable() {
    Node<K,V>[] tab; int sc;
    while ((tab = table) == null || tab.length == 0) {
        //如果正在初始化或者扩容
        if ((sc = sizeCtl) < 0)
            //等待
            Thread.yield(); // lost initialization race; just spin
        else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) { //CAS操作
            try {
                if ((tab = table) == null || tab.length == 0) {
                    int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                    @SuppressWarnings("unchecked")
                    Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                    table = tab = nt;
                    sc = n - (n >>> 2);
                }
            } finally {
                sizeCtl = sc;
            }
            break;
        }
    }
    return tab;
}
```

2.如果当前数组位置是空则直接通过CAS自旋写入数据

```
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i,
                                     Node<K,V> c, Node<K,V> v) {
    return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);
}
```

3. 如果hash==MOVED，说明需要扩容，执行扩容

```

else if ((fh = f.hash) == MOVED)
    tab = helpTransfer(tab, f);

```

```

final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
    Node<K,V>[] nextTab; int sc;
    if (tab != null && (f instanceof ForwardingNode) &&
        (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
        int rs = resizeStamp(tab.length);
        while (nextTab == nextTable && table == tab &&
            (sc = sizeCtl) < 0) {
            if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                sc == rs + MAX_RESIZERS || transferIndex <= 0)
                break;
            if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                transfer(tab, nextTab);
                break;
            }
        }
        return nextTab;
    }
    return table;
}

```

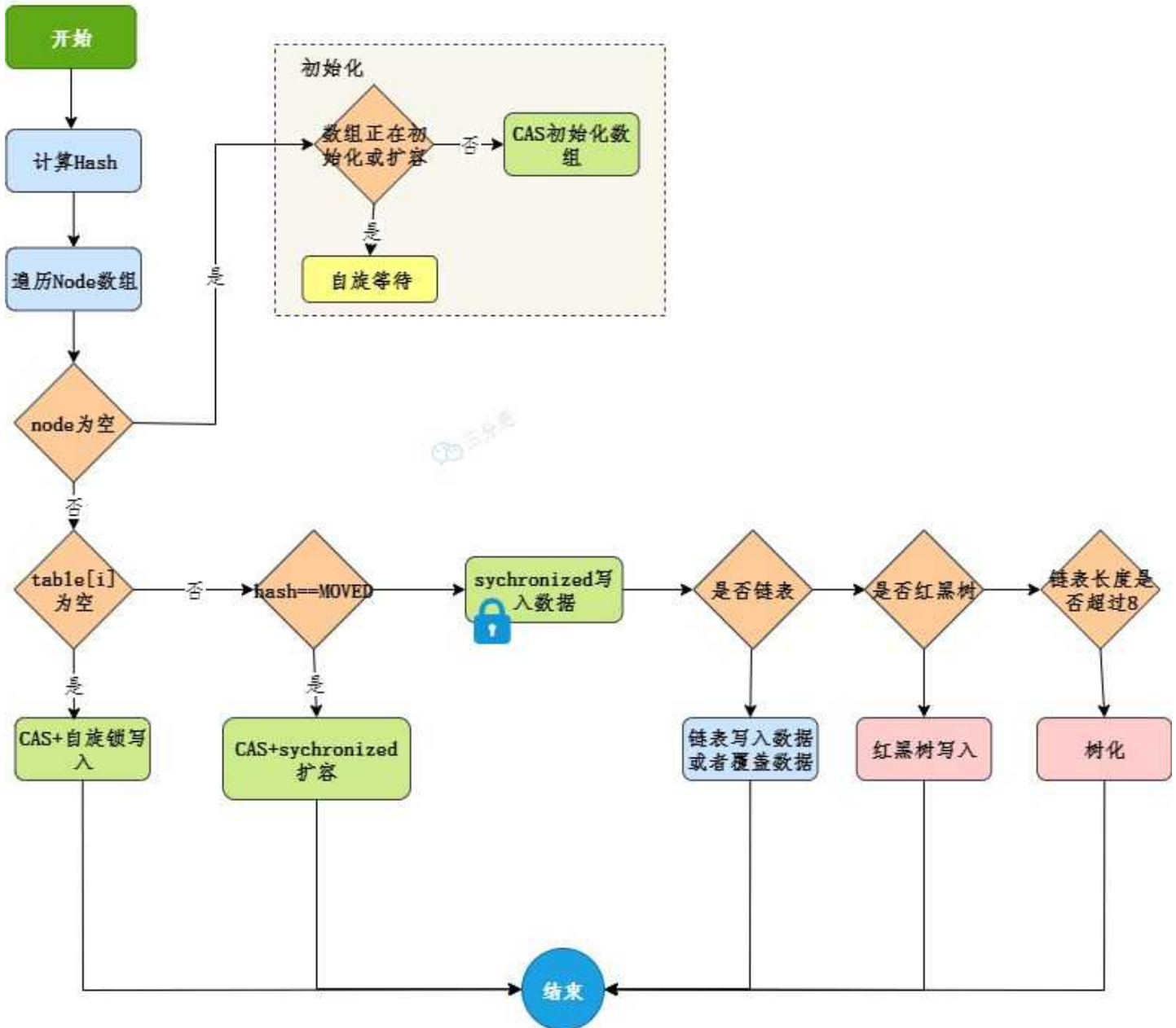
4. 如果都不满足，就使用synchronized写入数据，写入数据同样判断链表、红黑树，链表写入和HashMap的方式一样，key hash一样就覆盖，反之就尾插法，链表长度超过8就转换成红黑树

```

synchronized (f){
    .....
}

```

1.8 ConcurrentHashMap Put流程



get 查询

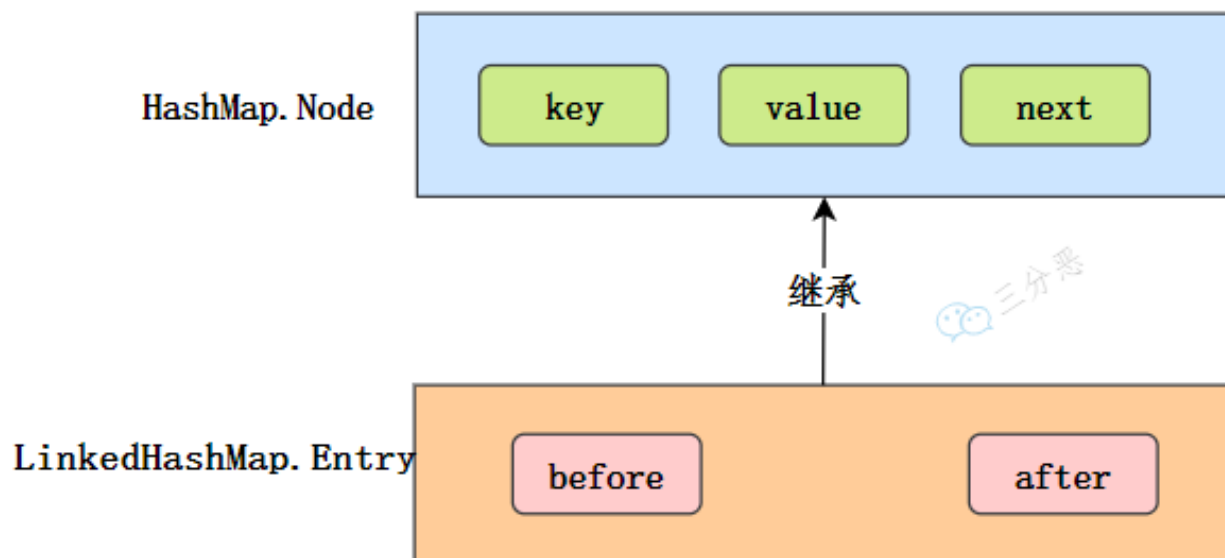
get很简单，和HashMap基本相同，通过key计算位置，table该位置key相同就返回，如果是红黑树按照红黑树获取，否则就遍历链表获取。

27.HashMap 内部节点是有序的吗？

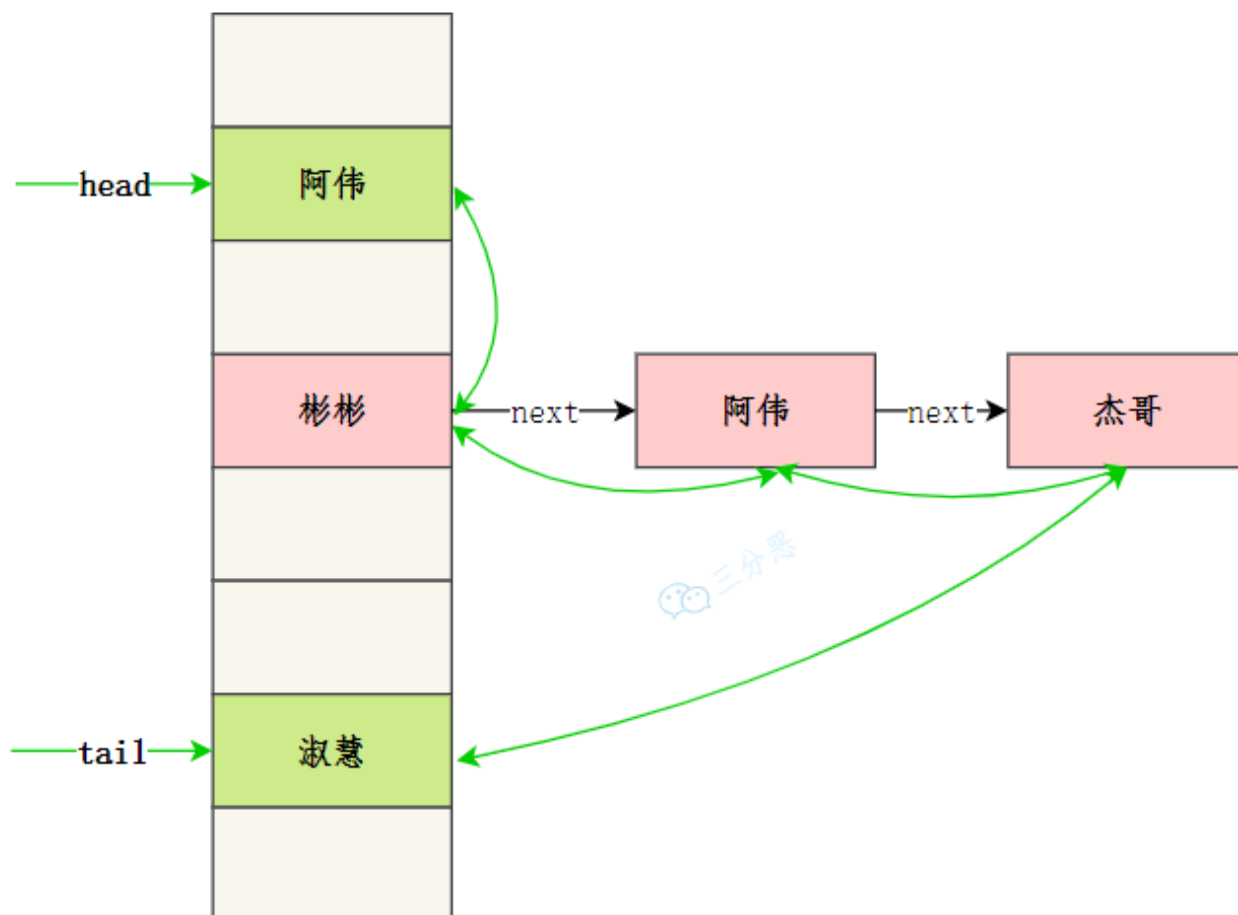
HashMap是无序的，根据 hash 值随机插入。如果想使用有序的Map，可以使用LinkedHashMap 或者 TreeMap。

28.讲讲 LinkedHashMap 怎么实现有序的？

LinkedHashMap维护了一个双向链表，有头尾节点，同时 LinkedHashMap 节点 Entry 内部除了继承 HashMap 的 Node 属性，还有 before 和 after 用于标识前置节点和后置节点。

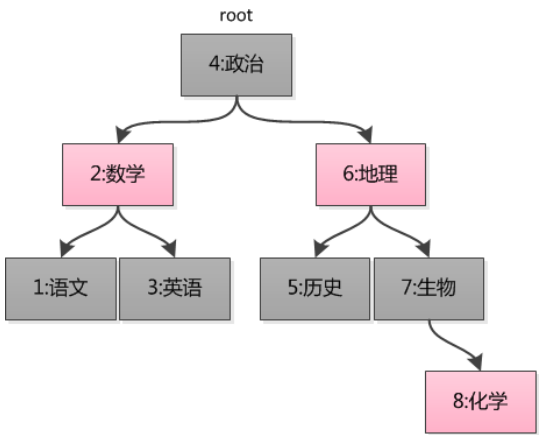


可以实现按插入的顺序或访问顺序排序。



29.讲讲 TreeMap 怎么实现有序的？

TreeMap 是按照 Key 的自然顺序或者 Comprator 的顺序进行排序，内部是通过红黑树来实现。所以要么 key 所属的类实现 Comparable 接口，或者自定义一个实现了 Comparator 接口的比较器，传给 TreeMap 用于 key 的比较。



tmap	TreeMap<K,V> (id=27)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	8
navigableKeySet	null
root	TreeMap\$Entry<K,V> (id=31)
color	true
key	Integer (id=34)
left	TreeMap\$Entry<K,V> (id=36)
parent	null
right	TreeMap\$Entry<K,V> (id=39)
value	"政治" (id=40)
size	8
values	null

{1=语文, 2=数学, 3=英语, 4=政治, 5=历史, 6=地理, 7=生物, 8=化学}

Set

Set 面试没啥好问的，拿 HashSet 来凑个数。

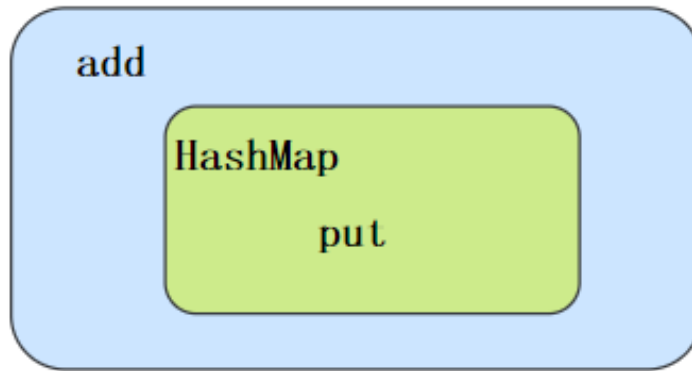
30.讲讲 HashSet 的底层实现？

HashSet 底层就是基于 HashMap 实现的。（ HashSet 的源码非常非常少，因为除了 clone()、writeObject()、readObject() 是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。

HashSet 的 add 方法，直接调用 HashMap 的 put 方法，将添加的元素作为 key，new 一个 Object 作为 value，直接调用 HashMap 的 put 方法，它会根据返回值是否为空来判断是否插入元素成功。

```
public boolean add(E e) {
    return map.put(e, PRESENT) != null;
}
```

HashSet



而在HashMap的putVal方法中，进行了一系列判断，最后的结果是，只有在key在table数组中不存在的时候，才会返回插入的值。

```
if (e != null) { // existing mapping for key
    V oldValue = e.value;
    if (!onlyIfAbsent || oldValue == null)
        e.value = value;
    afterNodeAccess(e);
    return oldValue;
}
```

没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟。

系列内容：

- [面渣逆袭 Java SE 篇](#) 🍷
 - [面渣逆袭 Java 集合框架篇](#) 🍷
 - [面渣逆袭 Java 并发编程篇](#) 🍷
 - [面渣逆袭 JVM 篇](#) 🍷
 - [面渣逆袭 Spring 篇](#) 🍷
 - [面渣逆袭 Redis 篇](#) 🍷
 - [面渣逆袭 MyBatis 篇](#) 🍷
 - [面渣逆袭 MySQL 篇](#) 🍷
 - [面渣逆袭操作系统篇](#) 🍷
 - [面渣逆袭计算机网络篇](#) 🍷
-

图文详解 30 道Java集合框架面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳[转载链接](#)，作者：三分恶，戳[原文链接](#)。



**关注沉默王二
学Java不迷路**

