**面试官**: 你先自我介绍一下吧！

**安琪拉**: 我是安琪拉，草丛三婊之一，最强中单（钟馗不服）！哦，不对，串场了，我是\*\*，目前在--公司做--系统开发。

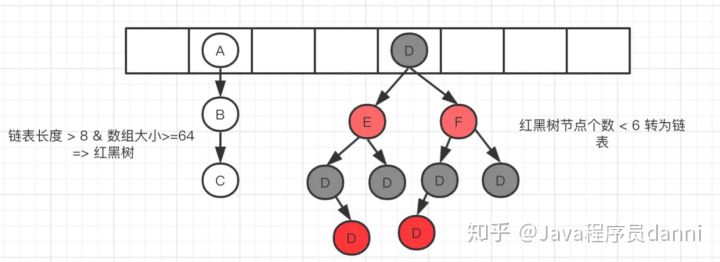
**面试官**: 看你简历上写熟悉Java集合，HashMap用过的吧？

**安琪拉**: 用过的。(还是熟悉的味道)

**面试官**: 那你跟我讲讲HashMap的内部数据结构？

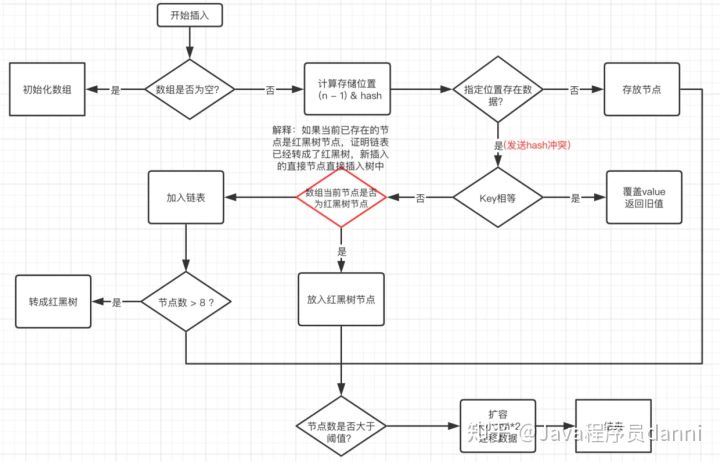
**安琪拉**: 目前我用的是JDK1.8版本的，内部使用数组 + 链表红黑树；

**安琪拉**: 方便我给您画个数据结构图吧：



**面试官**: 那你清楚HashMap的数据插入原理吗？

**安琪拉**: 呃[做沉思状]。我觉得还是应该画个图比较清楚，如下：



1. 判断数组是否为空，为空进行初始化;
2. 不为空，计算 k 的 hash 值，通过(n - 1) & hash计算应当存放在数组中的下标 index;
3. 查看 table[index] 是否存在数据，没有数据就构造一个Node节点存放在 table[index] 中；
4. 存在数据，说明发生了hash冲突(存在二个节点key的hash值一样), 继续判断key是否相等，相等，用新的value替换原数据(onlyIfAbsent为false)；
5. 如果不相等，判断当前节点类型是不是树型节点，如果是树型节点，创造树型节点插入红黑树中；
6. 如果不是树型节点，创建普通Node加入链表中；判断链表长度是否大于 8， 大于的话链表转换为红黑树；
7. 插入完成之后判断当前节点数是否大于阈值，如果大于开始扩容为原数组的二倍。

**面试官**: 刚才你提到HashMap的初始化，那HashMap怎么设定初始容量大小的吗？

**安琪拉**: [这也算问题？?] 一般如果new HashMap() 不传值，默认大小是16，负载因子是0.75， 如果自己传入初始大小k，初始化大小为 大于k的 2的整数次方，例如如果传10，大小为16。（补充说明:实现代码如下）

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

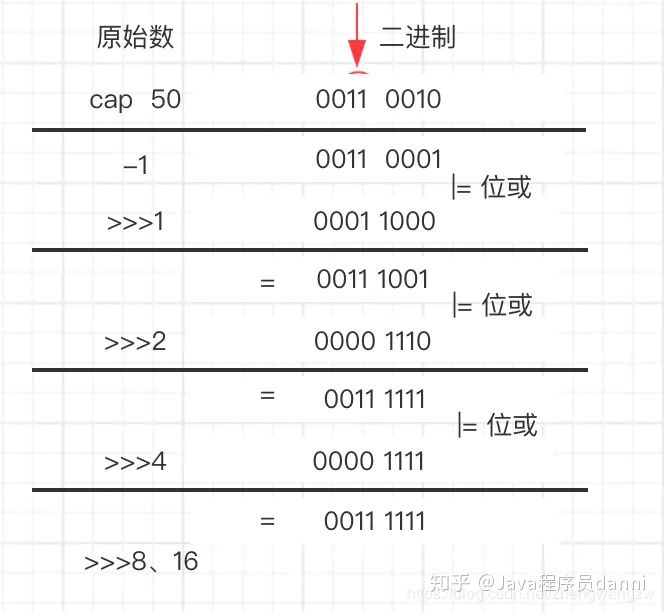
n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

补充说明：下图是详细过程，算法就是让初始二进制右移1，2，4，8，16位，分别与自己异或，把高位第一个为1的数通过不断右移，把高位为1的后几位全变为1，111111 + 1 = 1000000 = [图片上传失败...(image-27b652-1584612480673)]  
（符合大于50并且是2的整数次幂 ）



**面试官**: 你提到hash函数，你知道HashMap的哈希函数怎么设计的吗？

**安琪拉**: [问的还挺细] hash函数是先拿到通过key 的hashcode，是32位的int值，然后让hashcode的高16位和低16位进行异或操作。



**面试官**: 那你知道为什么这么设计吗？

**安琪拉**: [这也要问]，这个也叫扰动函数，这么设计有二点原因：

1. 一定要尽可能降低hash碰撞，越分散越好；
2. 算法一定要尽可能高效，因为这是高频操作, 因此采用位运算；

**面试官**: 为什么采用hashcode的高16位和低16位异或能降低hash碰撞？hash函数能不能直接用key的hashcode？

[这问题有点刁钻], 安琪拉差点原地💥了，恨不得出biubiubiu 二一三连招。

**安琪拉**: 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**，前后加起来大概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。你想，如果HashMap数组的初始大小才16，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。(来自知乎-[胖君](https://www.zhihu.com/question/20733617/answer/111577937))

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%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00000000 00000000 00001111。和某散列值做“与”操作如下，结果就是截取了最低的四位值。

10100101 11000100 00100101

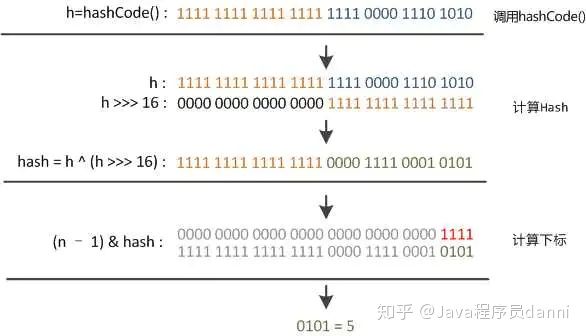
& 00000000 00000000 00001111

----------------------------------

00000000 00000000 00000101 //高位全部归零，只保留末四位

但这时候问题就来了，这样就算我的散列值分布再松散，要是只取最后几位的话，碰撞也会很严重。更要命的是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，就无比蛋疼。

时候“扰动函数”的价值就体现出来了，说到这里大家应该猜出来了。看下面这个图，



右位移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。

最后我们来看一下Peter Lawley的一篇专栏文章《An introduction to optimising a hashing strategy》里的的一个实验：他随机选取了352个字符串，在他们散列值完全没有冲突的前提下，对它们做低位掩码，取数组下标。



结果显示，当HashMap数组长度为512的时候

https://pic2.zhimg.com/80/v2-e990e5676fea622087fa359c719e76fd_720w.png

，也就是用掩码取低9位的时候，在没有扰动函数的情况下，发生了103次碰撞，接近30%。而在使用了扰动函数之后只有92次碰撞。碰撞减少了将近10%。看来扰动函数确实还是有功效的。

另外Java1.8相比1.7做了调整，1.7做了四次移位和四次异或，但明显Java 8觉得扰动做一次就够了，做4次的话，多了可能边际效用也不大，所谓为了效率考虑就改成一次了。

下面是1.7的hash代码：

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

**面试官**: 看来做过功课，有点料啊！你刚刚说到1.8对hash函数做了优化，1.8还有别的优化吗？

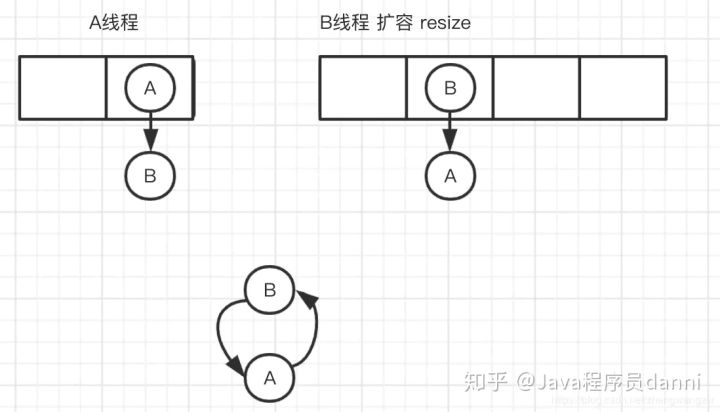
**安琪拉**: 1.8还有三点主要的优化：

1. 数组+链表改成了数组+链表或红黑树；
2. 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；
3. 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，位置不变或索引+旧容量大小；
4. 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；

**面试官**: 你分别跟我讲讲为什么要做这几点优化；

**安琪拉**: 【咳咳，果然是连环炮】

1. 1.8使用红黑树：防止发生hash冲突，链表长度过长，将时间复杂度由O(n)降为O(logn);
2. 1.8使用尾插法：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；  
   A线程在插入节点B，B线程也在插入，遇到容量不够开始扩容，重新hash，放置元素，采用头插法，后遍历到的B节点放入了头部，这样形成了环，如下图所示：



1.7的扩容调用transfer代码，如下所示：

```

void transfer(Entry[] newTable, boolean rehash) {

int newCapacity = newTable.length;

for (Entry<K,V> e : table) {

while(null != e) {

Entry<K,V> next = e.next;

if (rehash) {

e.hash = null == e.key ? 0 : hash(e.key);

}

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

e.next = newTable[i]; //A线程如果执行到这一行挂起，B线程开始进行扩容

newTable[i] = e;

e = next;

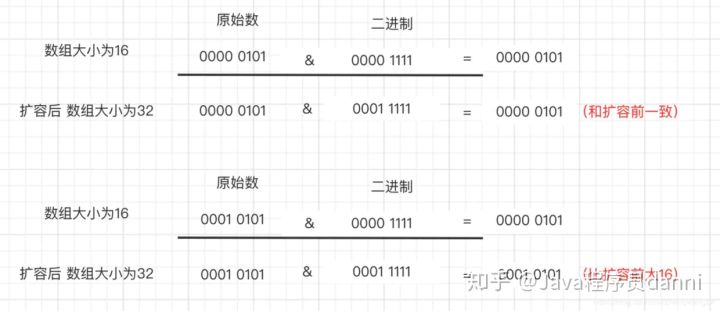
}

}

}

```

1. 扩容的时候为什么1.8 不用重新hash就可以直接定位原节点在新数据的位置呢?  
   这是由于扩容是扩大为原数组大小的2倍，用于计算数组位置的掩码仅仅只是高位多了一个1，怎么理解呢？  
   扩容前长度为16，用于计算(n-1) & hash 的二进制n-1为0000 1111，扩容为32后的二进制就高位多了1，为0001 1111。  
   因为是& 运算，1和任何数 & 都是它本身，那就分二种情况，如下图：原数据hashcode高位第4位为0和高位为1的情况；  
   第四位高位为0，重新hash数值不变，第四位为1，重新hash数值比原来大16（旧数组的容量）



**面试官**: 那HashMap是线程安全的吗？

**安琪拉**: 不是，在多线程环境下，1.7 会产生死循环、数据丢失、数据覆盖的问题，1.8 中会有数据覆盖的问题，以1.8为例，看👇的代码，当A线程判断index位置为空后正好挂起，B线程开始往index位置的写入节点数据，这时A线程恢复现场，执行赋值操作，就把A线程的数据给覆盖了；还有++size这个地方也会造成多线程同时扩容等问题。

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) //多线程执行到这里

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) // 多个线程走到这，可能重复resize()

resize();

afterNodeInsertion(evict);

return null;

}

**面试官**: 那你平常怎么解决这个线程不安全的问题？

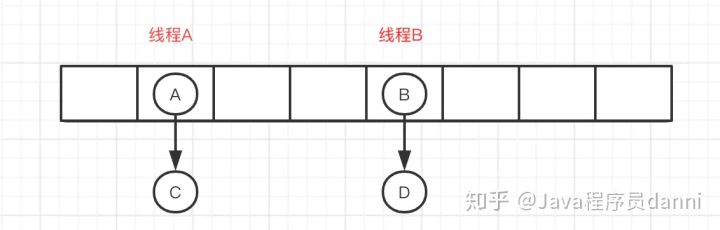
**安琪拉**: Java中有HashTable、Collections.synchronizedMap、以及ConcurrentHashMap可以实现线程安全的Map。

HashTable是直接在操作方法上加synchronized关键字，锁住整个数组，锁粒度比较大，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，方法通过对象锁实现线程安全；ConcurrentHashMap使用分段锁，降低了锁粒度，让并发度大大提高。

**面试官**: 那你知道ConcurrentHashMap的分段锁的实现原理吗？

**安琪拉**: 【天啦撸! 俄罗斯套娃，一个套一个】ConcurrentHashMap成员变量使用volatile 修饰，免除了指令重排序，同时保证内存可见性，另外使用CAS操作和synchronized结合实现赋值操作，多线程操作只会锁住当前操作索引的节点。

如下图，线程A锁住A节点所在链表，线程B锁住B节点所在链表，操作互不干涉。



**面试官**: 你前面提到链表转红黑树是链表长度达到阈值，这个阈值是多少？

**安琪拉**: 阈值是8，红黑树转链表阈值为6

**面试官**: 为什么是8，不是16，32甚至是7 ？又为什么红黑树转链表的阈值是6，不是8了呢？

**安琪拉**: 【你去问作者啊！天啦撸，biubiubiu 真想213连招】因为作者就这么设计的，哦，不对，因为经过计算，在hash函数设计合理的情况下，发生hash碰撞8次的几率为百万分之6，概率说话。。因为8够用了，至于为什么转回来是6，因为如果hash碰撞次数在8附近徘徊，会一直发生链表和红黑树的互相转化，为了预防这种情况的发生，设置为6。

**面试官**: HashMap内部节点是有序的吗？

**安琪拉**: 是无序的，根据hash值随机插入

**面试官**: 那有没有有序的Map？

**安琪拉**: LinkedHashMap 和 TreeMap

**面试官**: 跟我讲讲LinkedHashMap怎么实现有序的？

**安琪拉**: LinkedHashMap内部维护了一个单链表，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。

/\*\*

\* The head (eldest) of the doubly linked list.

\*/

transient LinkedHashMap.Entry<K,V> head;

/\*\*

\* The tail (youngest) of the doubly linked list.

\*/

transient LinkedHashMap.Entry<K,V> tail;

//链接新加入的p节点到链表后端

private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {

LinkedHashMap.Entry<K,V> last = tail;

tail = p;

if (last == null)

head = p;

else {

p.before = last;

last.after = p;

}

}

//LinkedHashMap的节点类

static class Entry<K,V> extends HashMap.Node<K,V> {

Entry<K,V> before, after;

Entry(int hash, K key, V value, Node<K,V> next) {

super(hash, key, value, next);

}

}

示例代码：

public static void main(String[] args) {

Map<String, String> map = new LinkedHashMap<String, String>();

map.put("1", "安琪拉");

map.put("2", "啦");

map.put("3", "啦");

for(Map.Entry<String,String> item: map.entrySet()){

System.out.println(item.getKey() + ":" + item.getValue());

}

}

//console输出

1:安琪拉

2:啦

3:啦

**面试官**: 跟我讲讲TreeMap怎么实现有序的？

**安琪拉**：TreeMap是按照Key的自然顺序或者实现的Comprator接口的比较函数的顺序进行排序，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。

**面试官**: 前面提到通过CAS 和 synchronized结合实现锁粒度的降低，你能给我讲讲CAS 的实现以及synchronized的实现原理吗？

**安琪拉**: 下一期咋们再约时间，OK？

**面试官**: 好吧，回去等通知吧！