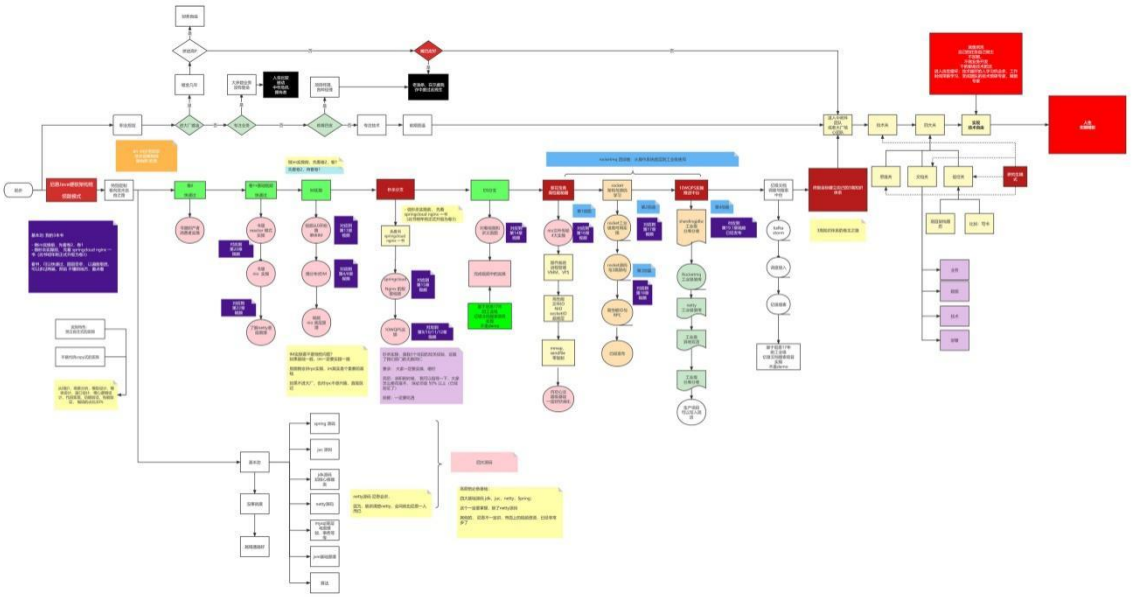
牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

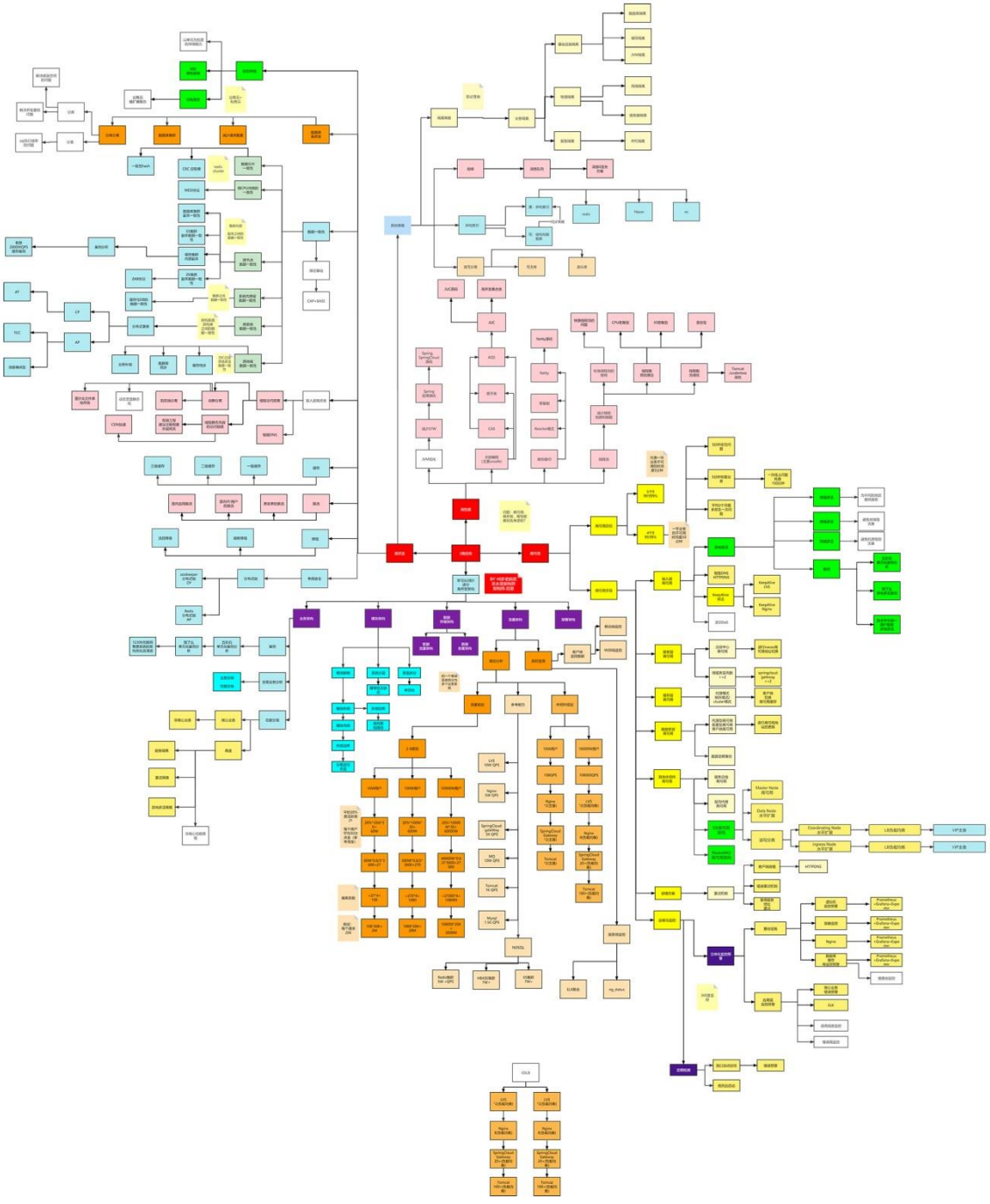
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

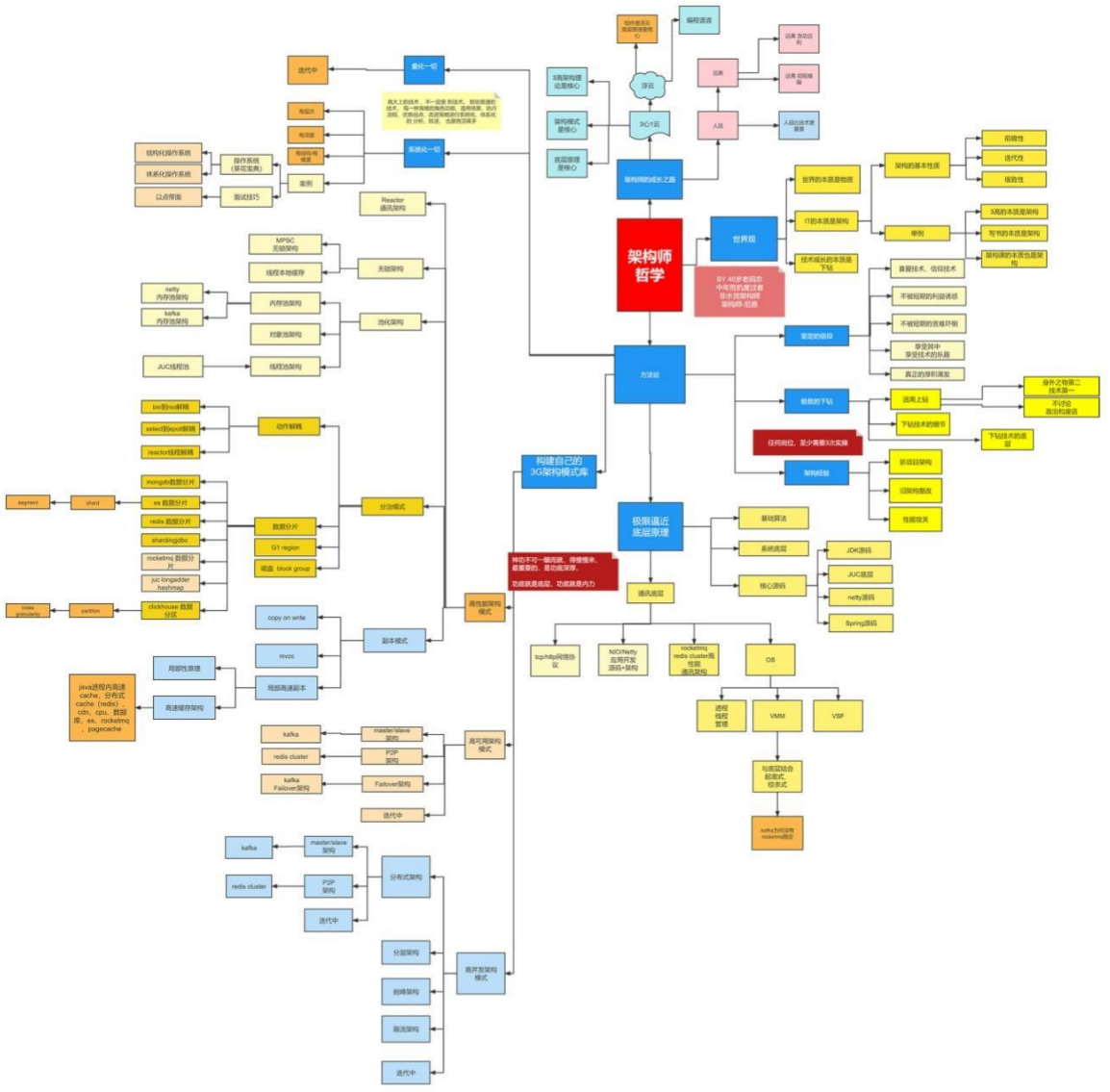
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



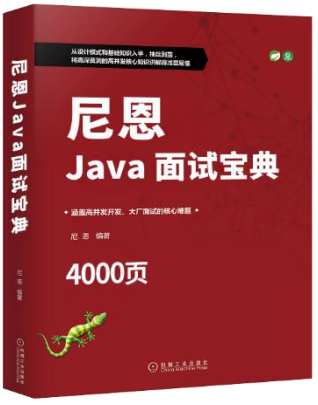
牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

尼恩Java面试宝典

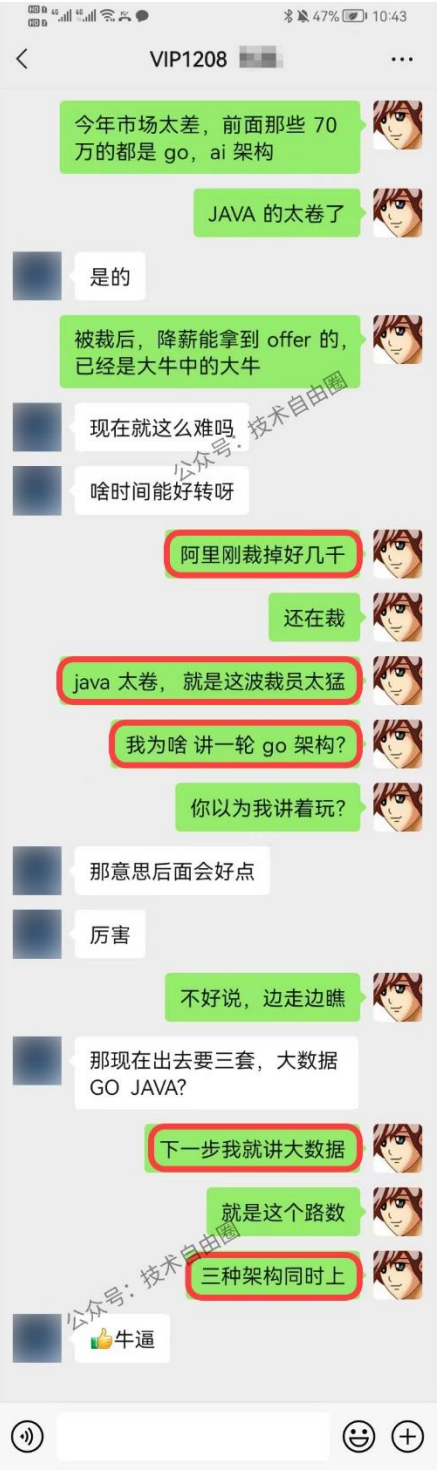
40 个专题（卷王专供+ 史上最全 + 2023 面试必备）
详情：https://www.cnblogs.com/crazymakercircle/p/13917138.html



名称

专题01：JVM面试题（卷王专供 + 史上最全 + 2022面试必备）-V81-from-尼恩Java面试宝典.pdf
专题02：Java算法面试题（卷王专供 + 史上最全 + 2022面试必备）-V80-from-Java面试红宝书.pdf
专题03：Java基础面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书 -release.pdf
专题04：架构设计面试题（卷王专供+ 史上最全 + 2023面试必备）-V86-from-尼恩Java面试宝典.pdf
专题05：Spring面试题_专题06：SpringMVC_专题07：Tomcat面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩面试宝典-release.pdf
专题08：SpringBoot面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题09：网络协议面试题（卷王专供+ 史上最全 + 2023面试必备）-V46-from-尼恩Java面试宝典-release.pdf
专题10：TCPIP协议（卷王专供+ 史上最全 + 2022面试必备）-V57-from-Java面试红宝书.pdf
专题11：JUC并发包与容器类（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题12：设计模式面试题（卷王专供+ 史上最全 + 2022面试必备）-V84-from-Java面试红宝书.pdf
专题13：死锁面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题14：Redis 面试题 （卷王专供+ 史上最全 + 2022面试必备）-V65-from-Java面试红宝书.pdf
专题15：分布式锁 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题16：Zookeeper 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题17：分布式事务面试题（卷王专供 + 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题18：一致性协议 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题19：Zab协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题20：Paxos 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题21：raft 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题22：Linux面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题23：Mysql 面试题（卷王专供+ 史上最全 + 2023面试必备）-V82-from-尼恩Java面试宝典.pdf
专题24：SpringCloud 面试题（卷王专供+ 史上最全 + 2023面试必备）-V12-from-Java面试红宝书-release.pdf
专题25：Netty 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（卷王专供+ 史上最全 + 2023面试必备）-V10-from-Java面试红宝书-release.pdf
专题27：内存泄漏 内存溢出（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题28：JVM 内存溢出 实战（卷王专供+ 史上最全 + 2023面试必备）-V17-from-Java面试红宝书-release.pdf
专题29：多线程面试题（卷王专供+ 史上最全 + 2023面试必备）-V66-from-Java面试红宝书.pdf
专题30：HR面试题：过五关斩六将后，小心阴沟翻船！（史上最全、避坑宝典）-V2-from-Java面试红宝书-release.pdf
专题31：Hash连环炮面试题（卷王专供+ 史上最全 + 2022面试必备）-V68-from-Java面试红宝书.pdf
专题32：大厂面试的基本流程和面试准备（阿里、腾讯、网易、京东、头条.....）-V2-from-Java面试红宝书-release.pdf
专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题34：Elasticsearch面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-Java面试红宝书-release.pdf
专题35：Mybatis面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩Java面试宝典-release.pdf
专题36：Dubbo面试题（卷王专供+ 史上最全 + 2023面试必备）-V21-from-尼恩Java面试宝典-release.pdf
专题37：Docker面试题（卷王专供+ 史上最全 + 2023面试必备）-V47-from-尼恩Java面试宝典.pdf
专题38：K8S面试题（卷王专供+ 史上最全 + 2023面试必备）-V59-from-尼恩Java面试宝典.pdf
专题39：Nginx面试题（卷王专供+ 史上最全 + 2023面试必备）-V27-from-尼恩Java面试宝典-release.pdf
专题40：操作系统面试题（卷王专供+ 史上最全 + 2023面试必备）-V28-from-尼恩Java面试宝典-release.pdf
专题41：大厂面试真题（卷王专供+ 史上最全 + 2023面试必备）-V84-from-尼恩Java面试宝典.pdf

未来职业，如何突围：三栖架构师



专题31：Hashmap 面试题 （史上最全、定期更新）

本文版本说明：V68

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《尼恩Java面试宝典》， 后面会不断升级，迭代。
>
> 本专题，作为 《尼恩Java面试宝典》专题之一， 《尼恩Java面试宝典》一共**41个面试专题，后续还会增加**

V68版本 （2023-5-17）

真题：京东太猛，手写hashmap又一次重现江湖

《Java面试红宝书》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？ 请参见文末

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可，

为什么 HashMap 那么重要?

HashMap的工作原理是目前java面试问的较为常见的问题之一，

这里面主要会包含是否用过Hashmap，hashMap的hash碰撞的机制是什么，hashMap是如何扩容的，hashMap的底层数据结构是什么，

jdk1.8中对hash算法和寻址算法是如何优化的等问题，那么我们现在就针对这些问题做个简要的分析和解答。

为什么在java面试中一定会深入考察HashMap?

hashMap作为一个键值对(key-value)的常见集合，在整个java的使用过程中都起着举足轻重的作用，

比如从DB中取值、数据的加工、数据回传给前端、数据转换为json等都可能使用到hashMap，

且hashMap作为一个可以允许空键值对的集合，也能实现自动的扩容，扩容的参数值为0.75，达到后自动扩容一倍，这样给一些处理未知数据量大小的数据来说，是很方便的。

虽然hashMap是线程不安全的，主要体现在1.7和1.8上，1.7的hashMap在扩容的时候容易形成循环链，导致死循环而报错，或者数据的丢失情况，

在1.8上，虽然对这方面做了改进，但是仍然是线程不安全的，

主要是体现在，若多线程操作数据，如线程A B同时进行数据的put操作，在put操作前，会进行key的hash碰撞，但是线程A B有可能同时碰撞且碰撞的值相同，那么就会发生线程A先插入到了碰撞的地方值，然后B也随后插入到同样的地方，

导致线程B会覆盖线程A所插入的值，导致数据丢失。

所以，在存在线程安全问题的场景下，需要使用JDK1.8版本的CurrentHashMap，而 实现原理上，JDK8中ConcurrentHashMap参考了JDK8 HashMap的实现，采用了数组+链表+红黑树的实现方式来设计，内部大量采用CAS操作。

所以，在面试的时候，都很喜欢问hashMap。

学习说明，这部分内容，建议大家配合卷2一起学习

卷2获得好评非常多，甚至被小伙伴评为**直逼jolt大奖**

卷2被小伙伴评为直逼jolt大奖

HashMap作为我们日常使用最频繁的容器之一，相信你一定不陌生了。

这里，从HashMap的底层实现讲起，深度了解下它的设计与优化，以及面试题。

常用的数据结构

首先 一起来温习下常用的数据结构，这样也有助于你更好地理解后面地内容。

数组：采用一段连续的存储单元来存储数据。对于指定下标的查找，时间复杂度为O(1)，但在数 组中间以及头部插入数据时，需要复制移动后面的元素。

链表：一种在物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。

> 链表由一系列结点（链表中每一个元素）组成，结点可以在运行时动态生成。每个结点都包含"存储数据单元的数据域"和"存储下一个结点地址的指针域"这两个部分。

由于链表不用必须按顺序存储，所以链表在插入的时候可以达到O(1)的复杂度，但查找一个结点或者访问特定编号的结点需要O(n)的时间。

哈希表：根据关键码值（Key value）直接进行访问的数据结构。通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做哈希函数，存放记录的数组就叫做哈希表。

树：由n（n≥1）个有限结点组成的一个具有层次关系的集合，就像是一棵倒挂的树。

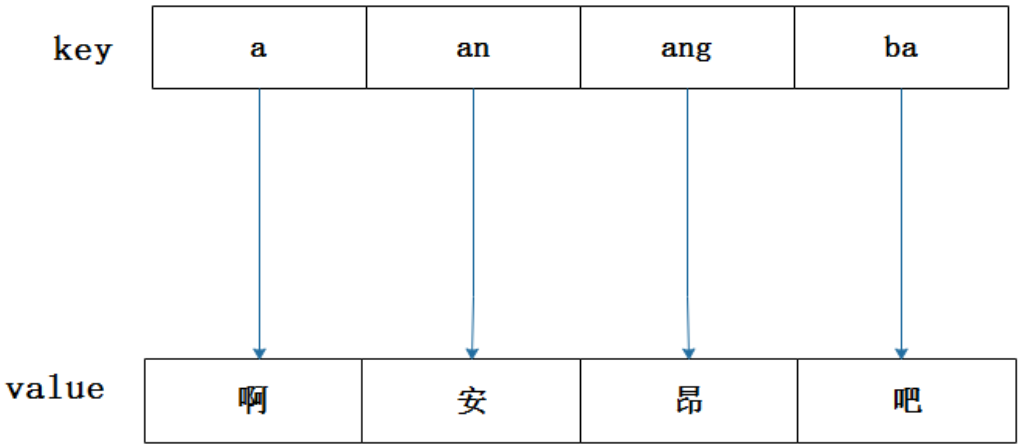
什么是哈希表

从根本上来说，一个哈希表包含一个数组，通过特殊的关键码(也就是key)来访问数组中的元素。

哈希表的主要思想是:

- 存放Value的时候，通过一个**哈希函数**，通过 关键码（key）进行哈希运算得到哈希值，然后得到映射的位置， 去寻找存放值的地方 ，
- 读取Value的时候，也是通过同一个**哈希函数**，通过 关键码（key）进行哈希运算得到哈希值，然后得到 映射的位置，从那个位置去读取。

最直接的例子就是字典，例如下面的字典图，如果我们要找 “啊” 这个字，只要根据拼音 “a” 去查找拼音索引，查找 “a” 在字典中的位置 “啊”，这个过程就是哈希函数的作用，用公式来表达就是：f(key)，而这样的函数所建立的表就是哈希表。



哈希表的优势：加快了查找的速度。

比起数组和链表查找元素时需要遍历整个集合的情况来说，哈希表明显方便和效率的多。

常见的哈希算法

哈希表的组成取决于哈希算法，也就是哈希函数的构成，下面列举几种常见的哈希算法。

1） 直接定址法

- 取关键字或关键字的某个线性函数值为散列地址。
- 即 f(key) = key 或 f(key) = a*key + b，其中a和b为常数。

2） 除留余数法

- 取关键字被某个不大于散列表长度 m 的数 p 求余，得到的作为散列地址。
- 即 f(key) = key % p, p < m。这是最为常见的一种哈希算法。

3） 数字分析法

- 当关键字的位数大于地址的位数，对关键字的各位分布进行分析，选出分布均匀的任意几位作为散列地址。
- 仅适用于所有关键字都已知的情况下，根据实际应用确定要选取的部分，尽量避免发生冲突。

4） 平方取中法

- 先计算出关键字值的平方，然后取平方值中间几位作为散列地址。
- 随机分布的关键字，得到的散列地址也是随机分布的。

5） 随机数法

- 选择一个随机函数，把关键字的随机函数值作为它的哈希值。
- 通常当关键字的长度不等时用这种方法。

什么是哈希冲突（hash碰撞）

哈希表因为其本身的结构使得查找对应的值变得方便快捷，但也带来了一些问题，

以上面的字典图为例，key中的一个拼音对应一个字，那如果字典中有两个字的拼音相同呢？

例如，我们要查找 “按” 这个字，根据字母拼音就会跳到 “安” 的位置，这就是典型的哈希冲突问题。

哈希冲突问题，用公式表达就是：

```
key1 ≠ key2 ， f(key1) = f(key2)
```

一般来说，哈希冲突是无法避免的，

如果要完全避免的话，那么就只能一个字典对应一个值的地址，也就是一个字就有一个索引 (**安** 和 **按**就是两个索引)，

这样一来，空间就会增大，甚至内存溢出。

需要想尽办法，减少 哈希冲突（hash碰撞）为啥呢？**Hash碰撞的概率就越小，map的存取效率就会越高**

哈希冲突的解决办法

常见的哈希冲突解决办法有两种:

- 开放地址法
- 链地址法。

一、开放地址法

开发地址法的做法是，当冲突发生时，使用某种探测算法在散列表中寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到。

按照探测序列的方法，一般将开放地址法区分为线性探查法、二次探查法、双重散列法等。

这里为了更好的展示三种方法的效果，我们用以一个模为8的哈希表为例，采用**除留余数法**，

往表中插入三个关键字分别为26，35，36的记录，分别除8取模后，在表中的位置如下：

0	1	2	3	4	5	6	7
		26	35	36			

这个时候插入42，那么正常应该在地址为2的位置里，但因为关键字30已经占据了位置，

所以就需要解决这个地址冲突的情况，接下来就介绍三种探测方法的原理，并展示效果图。

1） 线性探查法：

fi=(f(key)+i) % m ， 0 ≤ i ≤ m-1

探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+1]，…，直到 T[m-1]，此后又循环到 T[0]，T[1]，…，直到探查到有空余的地址或者到 T[d-1]为止。

插入42时，探查到地址2的位置已经被占据，接着下一个地址3，地址4，直到空位置的地址5，所以39应放入地址为5的位置。

缺点：需要不断处理冲突，无论是存入还是查找效率都会大大降低。

0	1	2	3	4	5	6	7
		26	35	36	42		

2） 二次探查法

fi=(f(key)+di) % m，0 ≤ i ≤ m-1

探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+di]，di 为增量序列12，-12，22，-22，……，q2，-q2 且q≤1/2 (m-1) ,直到探查到 有空余地址或者到 T[d-1]为止。

缺点：无法探查到整个散列空间。

所以插入42时，探查到地址2被占据，就会探查T[2+1^2]也就是地址3的位置，被占据后接着探查到地址7，然后插入。

0	1	2	3	4	5	6	7
		26	35	36			42

3） 双哈希函数探测法

fi=(f(key)+i*g(key)) % m (i=1，2，……，m-1)

其中，f(key) 和 g(key) 是两个不同的哈希函数，m为哈希表的长度

步骤：

双哈希函数探测法，先用第一个函数 **f(key)** 对关键码计算哈希地址，一旦产生地址冲突，再用第二个函数 **g(key)** 确定移动的步长因子，最后通过步长因子序列由探测函数寻找空的哈希地址。

比如，f(key)=a 时产生地址冲突，就计算g(key)=b，则探测的地址序列为 f1=(a+b) mod m，f2=(a+2b) mod m，……，fm-1=(a+(m-1)b) % m，假设 b 为 3，那么关键字42应放在 “5” 的位置。

0	1	2	3	4	5	6	7
		26	35	36	42		

开发地址法的问题：

开发地址法，通过持续的探测，最终找到空的位置。

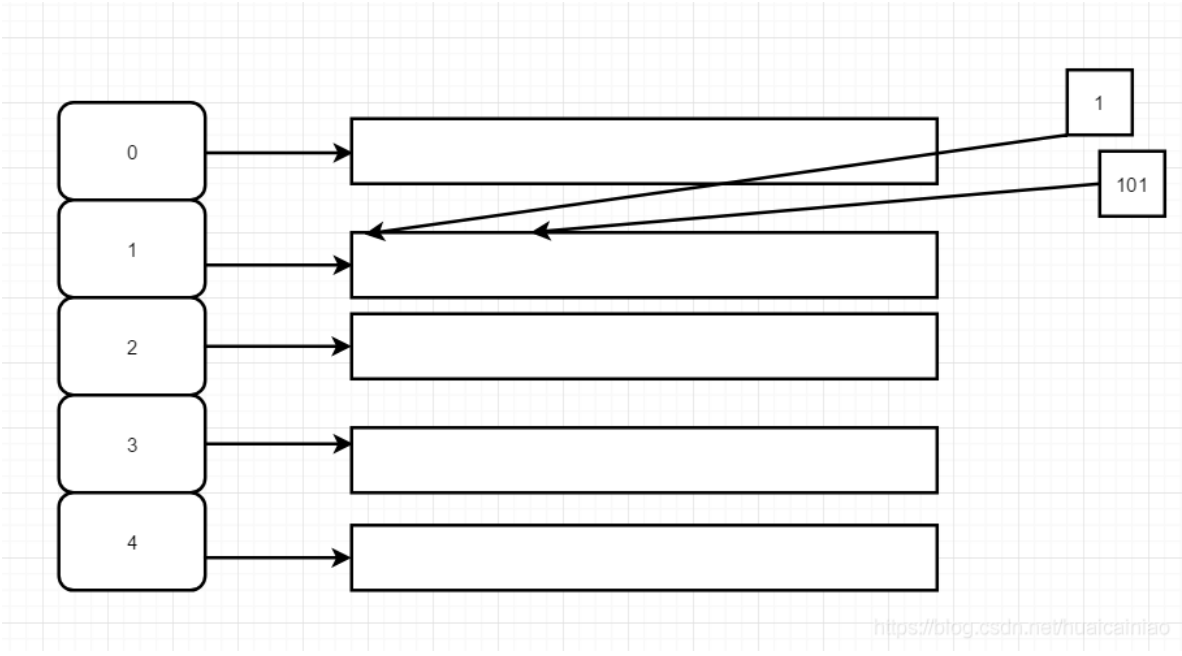
上面的例子中，开发地址方虽然解决了问题，但是26和42,占据了一个数组同一个元素，42只能向下，此时再来一个取余为2 的值呢，只能向下继续寻找，同理，每一个来的值都只能向下寻找。

为了解决这个问题，引入了链地址法。

二、链地址法：

在哈希表每一个单元中设置链表，某个数据项对的关键字还是像通常一样映射到哈希表的单元中，而数据项本身插入到单元的链表中。

链地址法简单理解如下：

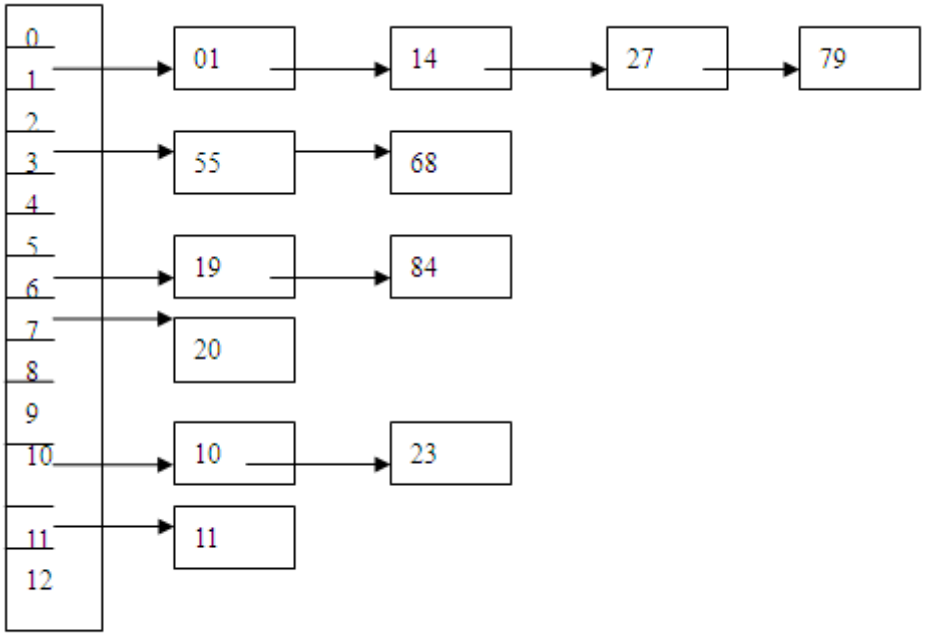


来一个相同的数据，就将它插入到单元对应的链表中，在来一个相同的，继续给链表中插入。

链地址法解决哈希冲突的例子如下：

（1）采用**除留余数法**构造哈希函数，而 冲突解决的方法为 **链地址法。**

（2）具体的关键字列表为（19,14,23,01,68,20,84,27,55,11,10,79），则哈希函数为H（key）=key MOD 13。则采用除留余数法和链地址法后得到的预想结果应该为：



链地址法处理冲突得到的哈希表

（3）哈希造表完成后，进行查找时，首先是根据哈希函数找到关键字的位置链，然后在该链中进行搜索，如果存在和关键字值相同的值，则查找成功，否则若到链表尾部仍未找到，则该关键字不存在。

哈希表性能

哈希表的特性决定了其高效的性能，大多数情况下**查找元素**的时间复杂度可以达到O(1)， 时间主要花在计算hash值上，

然而也有一些极端的情况，最坏的就是hash值全都映射在同一个地址上，这样**哈希表就会退化成链表**，例如下面的图片：

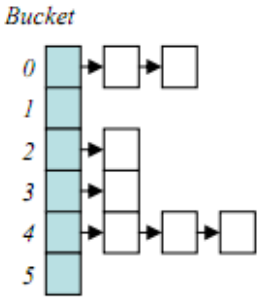


Figure 1: Normal operation of a hash table.

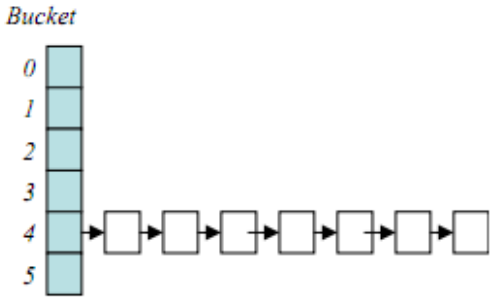


Figure 2: Worst-case hash table collisions.

当hash表变成图2的情况时，**查找元素**的时间复杂度会变为O(n)，效率瞬间低下，

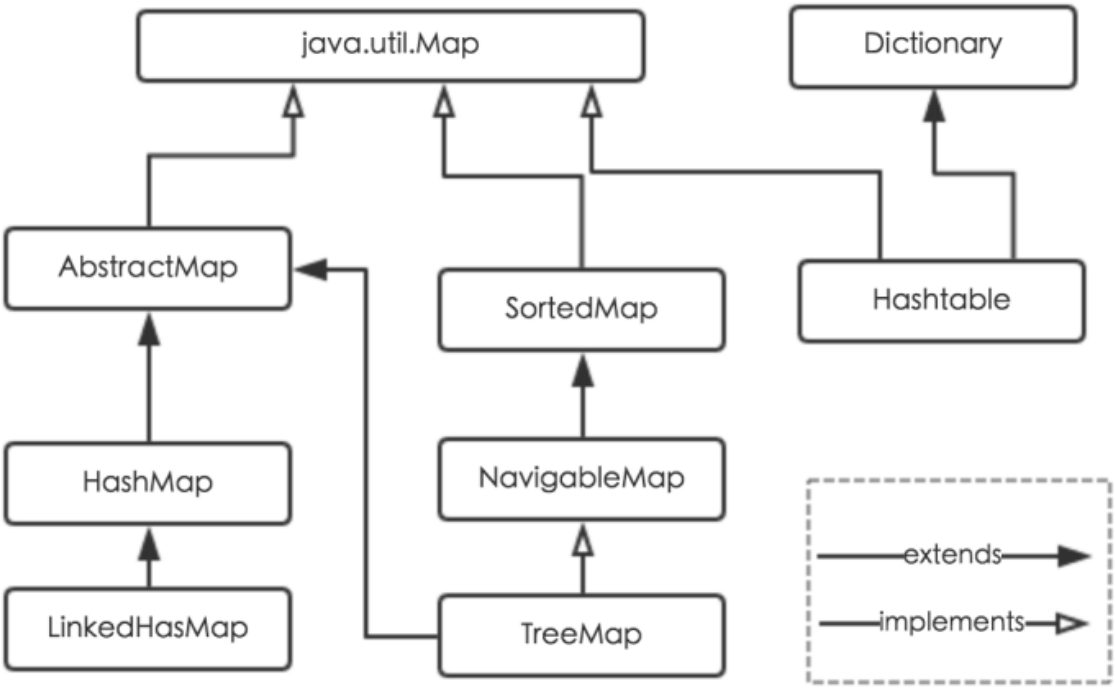
所以，设计一个好的哈希表尤其重要，如HashMap在jdk1.8后引入的红黑树结构就很好的解决了这种情况。

HashMap的类结构

类继承关系

Java为数据结构中的映射定义了一个接口java.util.Map，此接口主要有四个常用的实现类，分别是HashMap、Hashtable、LinkedHashMap和TreeMap，

类继承关系如下图所示：



下面针对各个实现类的特点做一些说明：

(1) HashMap：

它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。

HashMap 最多只允许一条记录的键为null，允许多条记录的值为null。

HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。

如果需要满足线程安全，可以用：

- Collections的synchronizedMap方法使HashMap具有线程安全的能力，
- 或者使用ConcurrentHashMap。

(2) Hashtable：

Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是**线程安全**的。

这个是老古董，Hashtable**不建议在代码中使用**，

不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

为何不建议用呢?

任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap。后者使用了 分段保护机制，也就是 分而治之的思想。

(3) LinkedHashMap：

LinkedHashMap是HashMap的一个子类，其优点在于： **保存了记录的插入顺序**，

在用Iterator遍历LinkedHashMap时，**先得到的记录肯定是先插入的**，也可以在构造时带参数，按照访问次序排序。

(4) TreeMap：

TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，**默认是按键值的升序排序**，也可以**指定排序的比较器**，

当用Iterator遍历TreeMap时，得到的记录是排过序的。

如果使用**排序的映射**，建议使用TreeMap。

在使用TreeMap时，**key必须实现Comparable接口**，或者在构造TreeMap传入自定义的Comparator，

否则会在运行时抛出java.lang.ClassCastException类型的异常。

注意：

对于上述四种Map类型的类，要求映射中的key是不可变的。

在创建内部的Entry后， key的哈希值不会被改变。

为啥呢?

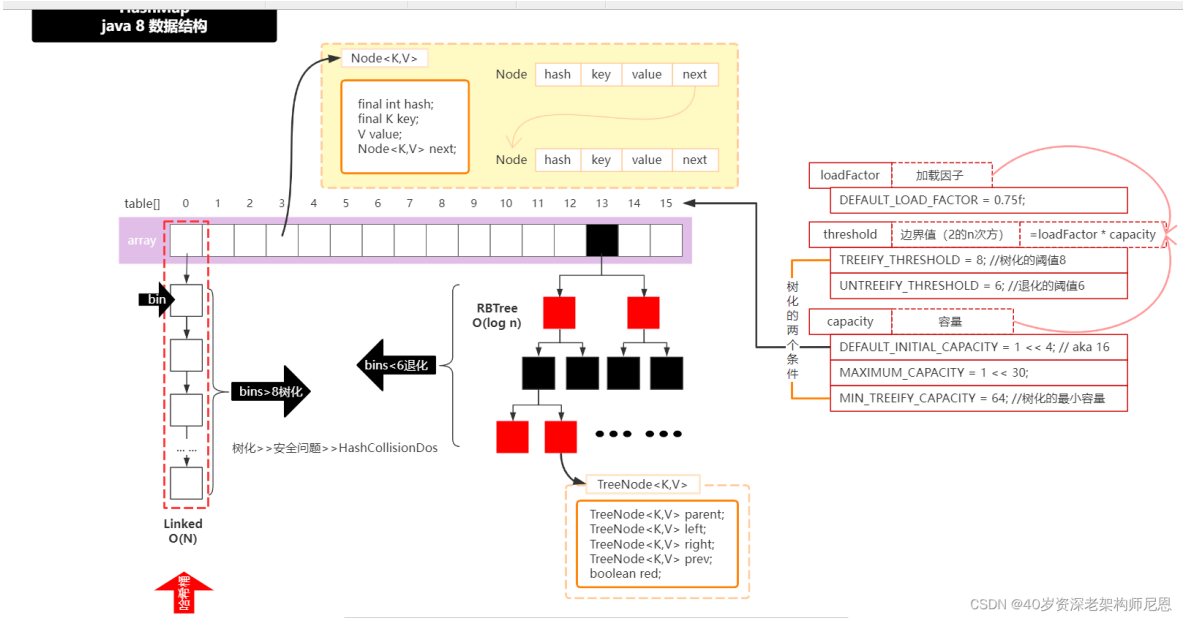
如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;  //key的哈希值不会被改变
        final K key; // 映射中的key是不可变的
        V value;
        Node<K,V> next;
```

HashMap存储结构

通过上面的比较，我们知道了HashMap是Java的Map家族中一个普通成员，鉴于它可以满足大多数场景的使用条件，所以是使用频度最高的一个。

下文我们主要结合源码，从存储结构、常用方法分析、扩容以及安全性等方面深入讲解HashMap的工作原理。



HashMap的重要属性：table 桶数组

从HashMap的源码中，我们可以发现，HashMap有一个非常重要的属性 —— table，

这是由一个Node类型的元素构成的数组：

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

table 也叫 **哈希数组**， **哈希槽位 数组** ， **table 桶数组** ， **散列表**， 数组中的一个 元素，常常被称之为一个 槽位 slot

Node类作为HashMap中的一个内部类，每个 Node 包含了一个 key-value 键值对。

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

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

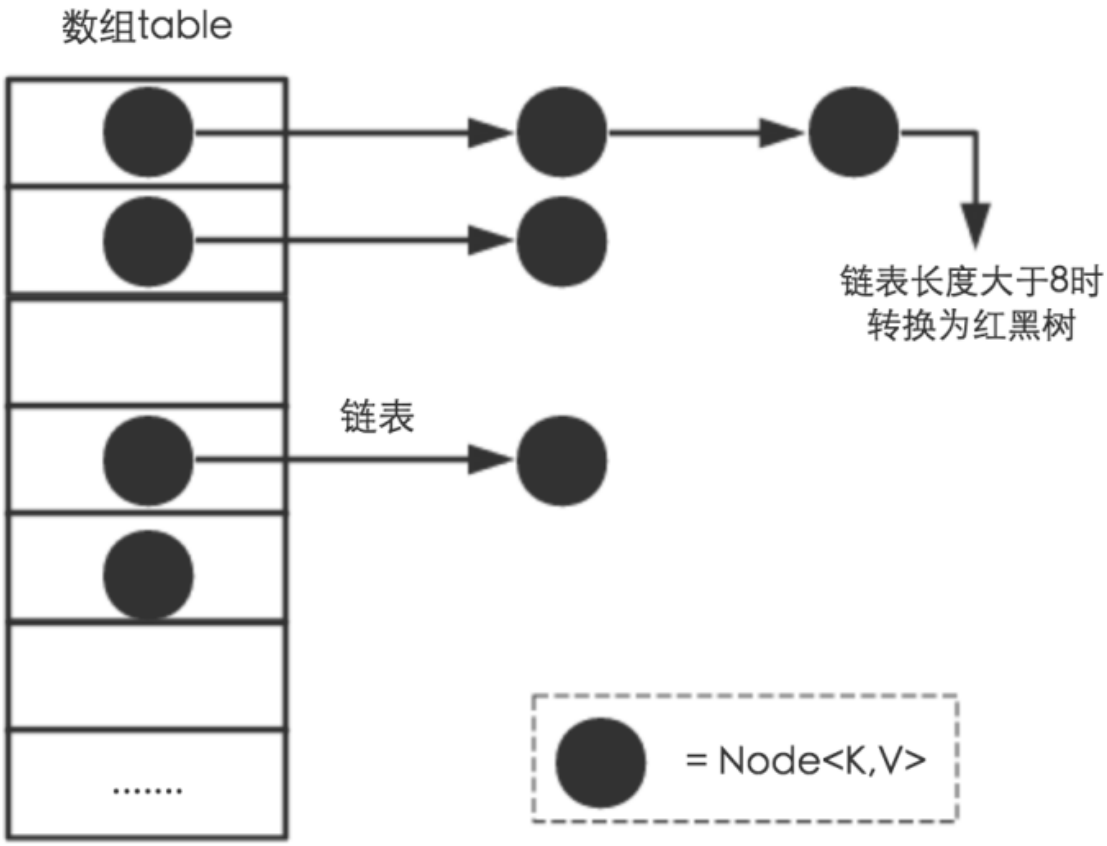
Node 类作为 HashMap 中的一个内部类，除了 key、value 两个属性外，还定义了一个next 指针。

next 指针的作用：链地址法解决哈希冲突。

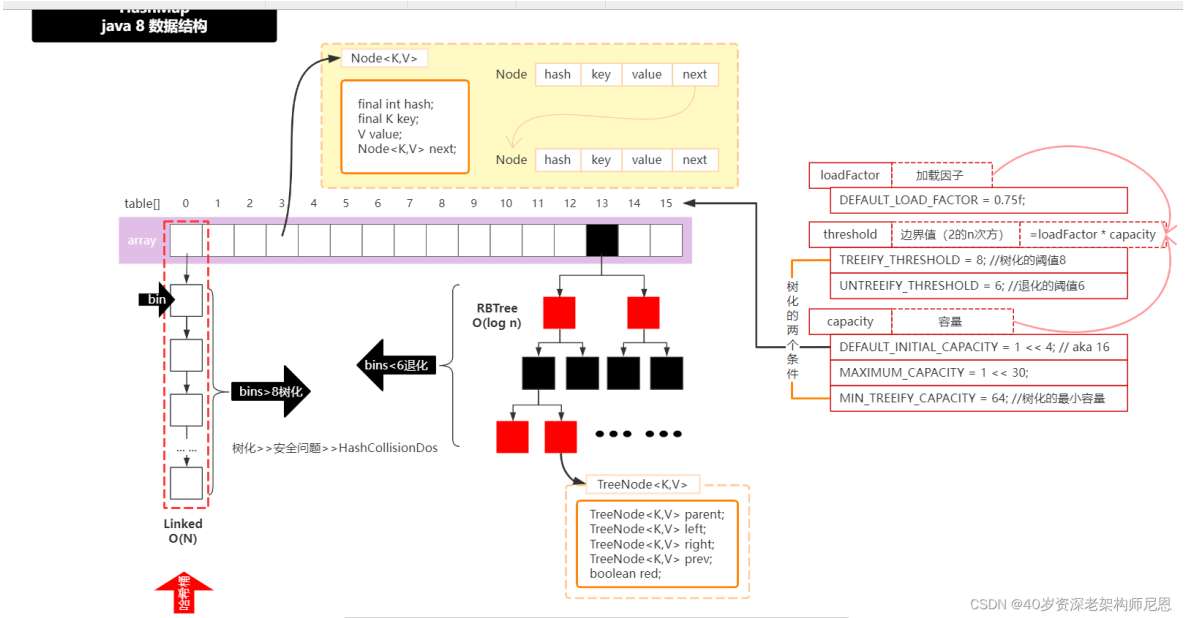
> 当有哈希冲突时，HashMap 会用之前数组当中相同哈希值对应存储的 Node 对象，通过指针指向新增的相同哈希值的 Node 对象的引用。

JDK1.8的table结构图

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，如下如所示。



来个大点的图



问题：

> HashMap的有什么特点呢?

HashMap的有什么特点

（1）HashMap采用了链地址法解决冲突

HashMap就是使用哈希表来存储的。

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。

上图中的每个黑色圆点就是一个Node对象。

Java中HashMap采用了链地址法。链地址法，简单来说，就是 **数组加链表** 的结合。

在每个数组元素上都一个链表结构， 当数据被Hash后，首先得到**数组下标**，然后， 把数据放在**对应下标元素的链表**上。

例如程序执行下面代码：

```
map.put("keyA","value1");
map.put("keyB","value2");
```

对于 第一句， 系统将调用"keyA"的hashCode()方法得到其hashCode ，然后再通过Hash算法来定位该键值对的存储位置，然后将 构造 entry 后加入到 存储位置 指向 的 链表中

对于 第一句， 系统将调用"keyB"的hashCode()方法得到其hashCode ，然后再通过Hash算法来定位该键值对的存储位置，然后将 构造 entry 后加入到 存储位置 指向 的链表中

有时两个key会定位到相同的位置，表示发生了Hash碰撞。

Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

（2）HashMap有较好的Hash算法和扩容机制

哈希桶数组的大小， 在空间成本和时间成本之间权衡，**时间和空间** 之间进行 权衡：

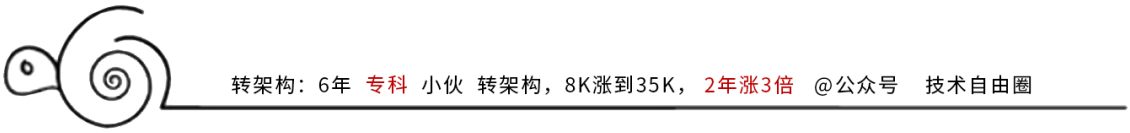
- 如果哈希桶数组很大，**即使较差的Hash算法**也会比较分散， **空间换时间**
- 如果哈希桶数组很小，即使**好的Hash算法**也会出现较多碰撞， **时间换空间**

所以，就需要在空间成本和时间成本之间权衡，

其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。

那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢?

> 答案就是好的Hash算法和扩容机制。



HashMap的重要属性：加载因子（loadFactor）和边界值（threshold)

HashMap还有两个重要的属性：

- 加载因子（loadFactor）
- 边界值（threshold）。

> 在初始化 HashMap时，就会涉及到这两个关键初始化参数。

loadFactor和threshold的源码如下：

```
    int threshold;             // 所能容纳的key-value对极限
    final float loadFactor;    // 负载因子

```

Node[] table的初始化长度length(默认值是16)，

loadFactor 为负载因子(默认值是0.75)，

threshold是HashMap所能容纳的最大数据量的Node 个数。

threshold 、length 、loadFactor 三者之间的关系：

> threshold = length * Load factor。
>
> 默认情况下 threshold = 16 * 0.75 =12。

threshold就是允许的哈希数组 最大元素数目，超过这个数目就重新resize(扩容)，扩容后的哈希数组容量length 是之前容量length 的两倍。

threshold是通过初始容量和LoadFactor计算所得，在初始HashMap不设置参数的情况下，默认边界值为12。

如果HashMap中Node的数量超过边界值，HashMap就会调用resize()方法重新分配table数组。

这将会导致HashMap的数组复制，迁移到另一块内存中去，从而影响HashMap的效率。

HashMap的重要属性：loadFactor 属性

为什么loadFactor 默认是0.75**这个值呢?**

loadFactor 也是可以调整的，默认是0.75，但是，如果loadFactor 负载因子越大，在数组定义好 length 长度之后，所能容纳的键值对个数越多。

> LoadFactor属性是用来间接设置Entry数组（哈希表）的内存空间大小，在初始HashMap不设置参数的情况下，默认LoadFactor值为0.75。

为什么loadFactor 默认是0.75这个值呢?

这是由于 加载因子的两面性导致的

加载因子越大，对空间的利用就越充分，碰撞的机会越高，这就意味着链表的长度越长，查找效率也就越低。

> 因为对于使用链表法的哈希表来说，查找一个元素的平均时间是O(1+n)，这里的n指的是遍 **历链表的长度**，

如果设置的加载因子太小，那么哈希表的数据将过于稀疏，对空间造成严重浪费。

当然，加载因子小，碰撞的机会越低， 查找的效率就搞，性能就越好。

默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下。

分为两种情况：

- 如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；
- 相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

HashMap的重要属性：size属性

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。

注意: size和table的长度length的区别，length是 哈希桶数组table的长度

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方，这是一定是一个合数，这是一种反常规的设计.

> 常规的设计是把桶数组的大小设计为素数。相对来说素数导致冲突的概率要小于合数，

比如，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。

HashMap采用这种非常规设计，主要是为了方便扩容。

而 HashMap为了减少冲突，采用另外的方法规避：计算哈希桶索引位置时，哈希值的高位参与运算。

HashMap的重要属性：modCount属性

我们能够发现，在集合类的源码里，像HashMap、TreeMap、ArrayList、LinkedList等都有modCount属性，字面意思就是修改次数，

首先看一下源码里对此属性的注释

HashMap部分源码：

```
    /**
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of mappings
in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     */
    transient int modCount;
```

汉译：

> 此哈希表已被**结构性修改**的次数，**结构性修改**是指哈希表的内部结构被修改，比如桶数组被修改或者拉链被修改。
>
> 那些更改桶数组或者拉链的操作如，重新哈希。 此字段用于HashMap集合迭代器的快速失败。

所以，modCount主要是为了防止在迭代过程中某些原因改变了原集合，导致出现不可预料的情况，从而抛出并发修改异常，

这可能也与Fail-Fast机制有关： 在可能出现错误的情况下提前抛出异常终止操作。

HashMap的remove方法源码(部分截取)：

```
if (node != null && (!matchValue || (v = node.value) == value ||
                                (value != null && value.equals(v)))) {
                if (node instanceof TreeNode)
                    ((TreeNode<K,V>)node).removeTreeNode(this, tab,
movable);
                else if (node == p)
                    tab[index] = node.next;
                else
                    p.next = node.next;
                ++modCount;  //进行了modCount自增操作
                --size;
                afterNodeRemoval(node);
                return node;
```

remove方法则进行了modCount自增操作，

然后来看一下HashMap的put方法源码(部分截取)：

```
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
         ++modCount;  //对于之前不存在的key进行put的时候，对modCount有修改
         if (++size > threshold)
             resize();
         afterNodeInsertion(evict);
         return null;
```

对于已经存在的key进行put修改value的时候，**对modCount没有修改**，

对于之前不存在的key进行put的时候，**对modCount有修改**，

通过比较put方法和remove方法可以看出，所以只有当对HashMap元素个数产生影响的时候才会修改modCount。

也是是说：modCount表示 HashMap集合的元素个数，导致集合的结构发生变化。

那么修改modCount有什么用呢?

这里用HashMap举例，大家知道当用迭代器遍历HashMap的时候，调用HashMap.remove方法时，

会产**并发修改的异常**ConcurrentModificationException

这是因为remove改变了HashMap集合的元素个数，导致集合的结构发生变化。

```
public static void main(String args[]) {
        Map<String, String> map = new HashMap<>();
        map.put("1", "zhangsan");
        map.put("2", "lisi");
        map.put("3", "wangwu");

        Iterator<String> iterator = map.keySet().iterator();
        while(iterator.hasNext()) {
            String name = iterator.next();
            map.remove("1");
        }
    }
```

执行结果： 抛出ConcurrentModificationException异常

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.HashMap$HashIterator.nextNode(HashMap.java:1442)
    at java.util.HashMap$KeyIterator.next(HashMap.java:1466)
    at com.cesec.springboot.system.service.Test.main(Test.java:14)
```

我们看一下抛出异常的KeyIterator.next()方法源码：

```
final class KeyIterator extends HashIterator
        implements Iterator<K> {
        public final K next() { return nextNode().key; }
    }
final Node<K,V> nextNode() {
            Node<K,V>[] t;
            Node<K,V> e = next;
            if (modCount != expectedModCount) //判断modCount和
expectedModCount是否一致
                throw new ConcurrentModificationException();
            if (e == null)
                throw new NoSuchElementException();
            if ((next = (current = e).next) == null && (t = table) != null)
{
                do {} while (index < t.length && (next = t[index++]) ==
null);
            }
            return e;
        }
```

在迭代器初始化时，会赋值expectedModCount，

在迭代过程中判断modCount和expectedModCount是否一致，如果不一致则抛出异常，

可以看到KeyIterator.next()调用了nextNode()方法，nextNode()方法中进行了modCount与expectedModCount判断。

这里更详细的说明一下，在迭代器初始化时，赋值expectedModCount，

假设与modCount相等，都为0，在迭代器遍历HashMap每次调用next方法时都会判断modCount和expectedModCount是否相等，

当进行remove操作时，modCount自增变为1，而expectedModCount仍然为0，再调用next方法时就会抛出异常。

需要通过迭代器的删除方法进行删除

所以迭代器遍历时，如果想删除元素，需要通过迭代器的删除方法进行删除， 这样下一次迭代操作，才不会抛出 **并发修改的异常**ConcurrentModificationException

那么为什么通过迭代器删除就可以呢?

HashIterator的remove方法源码：

```
public final void remove() {
    Node<K,V> p = current;
    if (p == null)
        throw new IllegalStateException();
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    current = null;
    K key = p.key;
    removeNode(hash(key), key, null, false, false);
    expectedModCount = modCount;
}
```

通过迭代器进行remove操作时，会重新赋值expectedModCount。

这样下一次迭代操作，才不会抛出 **并发修改的异常**ConcurrentModificationException

hashmap属性总结

HashMap通过哈希表数据结构的形式来存储键值对，这种设计的好处就是查询键值对的效率 高。

我们在使用HashMap时，可以结合自己的场景来设置初始容量和加载因子两个参数。当查询操 作较为频繁时，我们可以适当地减少加载因子；如果对内存利用率要求比较高，我可以适当的增加加载因子。

我们还可以在预知存储数据量的情况下，提前设置初始容量（初始容量=预知数据量/加载因 子）。这样做的好处是可以减少resize()操作，提高HashMap的效率。

HashMap还使用了数组+链表这两种数据结构相结合的方式实现了链地址法，当有哈希值冲突 时，就可以将冲突的键值对链成一个链表。

但这种方式又存在一个性能问题，如果链表过长，查询数据的时间复杂度就会增加。HashMap 就在Java8中使用了红黑树来解决链表过长导致的查询性能下降问题。以下是HashMap的数据结 构图：

HashMap源码分析

HashMap构造方法：

HashMap有两个重要的属性：加载因子（loadFactor）和边界值（threshold）。

loadFactor 属性是用来间接设置 Entry 数组（哈希表）的内存空间大小，在初始 HashMap 不设置参数的情况下，默认 loadFactor 为0.75。

为什么是0.75这个值呢?

这是因为对于使用链表法的哈希表来说，查找一个元素的平均时间是 O(1+n)，这里的 n 指的是遍历链表的长度，

因此加载因子越大，对空间的利用就越充分，这就意味着链表的长度越长，查找效率也就越低。

如果设置的加载因子太小，那么哈希表的数据就过于稀疏，对空间造成严重浪费。

有什么办法可以来解决因链表过长而导致的查询时间复杂度高的问题呢?

在JDK1.8后就使用了将链表转换为红黑树来解决这个问题。

Entry 数组（哈希槽位数组）的 threshold 阈值 是通过初始容量和 loadFactor计算所得，

在初始 HashMap 不设置参数的情况下，默认边界值为12（16*0.75）。

如果我们在初始化时，设置的初始化容量较小，HashMap 中 Node 的数量超过边界值，HashMap 就会调用 resize() 方法重新分配 table 数组。

这将导致 HashMap 的数组复制，迁移到另一块内存中去，从而影响 HashMap 的效率。

```
public HashMap() {//默认初始容量为16，加载因子为0.75
        this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }
    public HashMap(int initialCapacity) {//指定初始容量为initialCapacity
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }
    static final int MAXIMUM_CAPACITY = 1 << 30;//最大容量

    //当size到达threshold这个阈值时会扩容，下一次扩容的值，根据capacity * load factor进行计算，
    int threshold;
    /**由于HashMap的capacity都是2的幂，因此这个方法用于找到大于等于initialCapacity的最小的2的幂（initialCapacity如果就是2的幂，则返回的还是这个数）
     * 通过5次无符号移位运算以及或运算得到：
     *     n第一次右移一位时，相当于将最高位的1右移一位，再和原来的n取或，就将最高位和次高位都变成1，也就是两个1；
     *     第二次右移两位时，将最高的两个1向右移了两位，取或后得到四个1；
     *     依次类推，右移16位再取或就能得到32个1；
     *     最后通过加一进位得到2^n。
     * 比如initialCapacity = 10 ，那就返回16， initialCapacity = 17，那么就返回32
     *     10的二进制是1010，减1就是1001
     *     第一次右移取或： 1001 | 0100 = 1101 ；
     *     第二次右移取或： 1101 | 0011 = 1111 ；
     *     第三次右移取或： 1111 | 0000 = 1111 ；
     *     第四次第五次同理
     *     最后得到 n = 1111，返回值是 n+1 = 2 ^ 4 = 16 ；
```

```
24      * 让cap-1再赋值给n的目的是另找到的目标值大于或等于原值。这是为了防止，cap已经是2的幂。如果cap已经是2的幂，又没有执行这个减1操作，则执行完后面的几条无符号右移操作之后，返回的capacity将是这个cap的2倍。
25      * 例如十进制数值8，二进制为1000，如果不对它减1而直接操作，将得到答案10000，即16。显然不是结果。减1后二进制为111，再进行操作则会得到原来的数值1000，即8。
26      * 问题：tableSizeFor()最后赋值给threshold，但threshold是根据capacity * load factor进行计算的，这是不是有问题？
27      * 注意：在构造方法中，并没有对table这个成员变量进行初始化，table的初始化被推迟到了put方法中，在put方法中会对threshold重新计算。
28      * 问题：既然put会重新计算threshold，那么在构造初始化threshold的作用是什么？
29      * 答：在put时，会对table进行初始化，如果threshold大于0，会把threshold当作数组的长度进行table的初始化，否则创建的table的长度为16。
30      */
31     static final int tableSizeFor(int cap) {
32         int n = cap - 1;
33         n |= n >>> 1;
34         n |= n >>> 2;
35         n |= n >>> 4;
36         n |= n >>> 8;
37         n |= n >>> 16;
38         return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
39     }
40     public HashMap(int initialCapacity, float loadFactor) {//指定初始容量和加载因子
41         if (initialCapacity < 0)
42             throw new IllegalArgumentException("Illegal initial capacity: " +
43                                                initialCapacity);
44         if (initialCapacity > MAXIMUM_CAPACITY)//大于最大容量，设置为最大容量
45             initialCapacity = MAXIMUM_CAPACITY;
46         if (loadFactor <= 0 || Float.isNaN(loadFactor))//加载因子小于等于0或为NaN抛出异常
47             throw new IllegalArgumentException("Illegal load factor: " +
48                                                loadFactor);
49         this.loadFactor = loadFactor;
50         this.threshold = tableSizeFor(initialCapacity);//边界值
51     }
```

put方法源码：

当将一个 key-value 对添加到 HashMap 中，

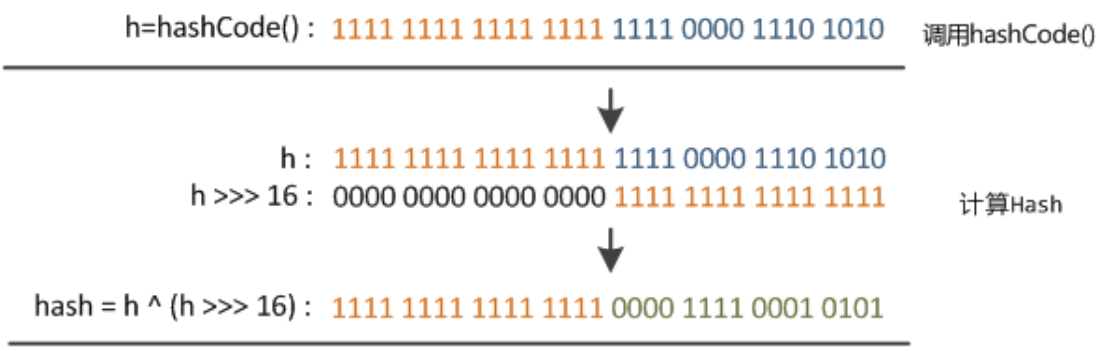
- 首先会根据该 key 的 hashCode() 返回值，再通过 hash() 方法计算出 hash 值，
- 再**除留余数法**，取得余数，这里通过位运算来完成。 putVal 方法中的 (n-1) & hash 就是 hash值除以n留余数， n 代表哈希表的长度。余数 (n-1) & hash 决定该 Node 的存储位置，哈希表习惯将长度设置为2的 n 次方，这样可以恰好保证 (n-1)&hash 计算得出的索引值总是位于 table 数组的索引之内。

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

hash计算：

key的hash值**高16位不变**，**低16位与高16位异或**，作为key的最终hash值。

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

// 要点1： h >>>  16，表示无符号右移16位，高位补0，任何数跟0异或都是其本身，因此key的hash值高16位不变。

//  要点2： 异或的运算法则为：0⊕0=0，1⊕0=1，0⊕1=1，1⊕1=0（同为0，异为1）
```



即取 int 类型的一半，刚好可以将该二进制数对半切开，

利用异或运算（如果两个数对应的位置相反，则结果为1，反之为0），这样可以避免哈希冲突。

底16位与高16位异或，其目标：

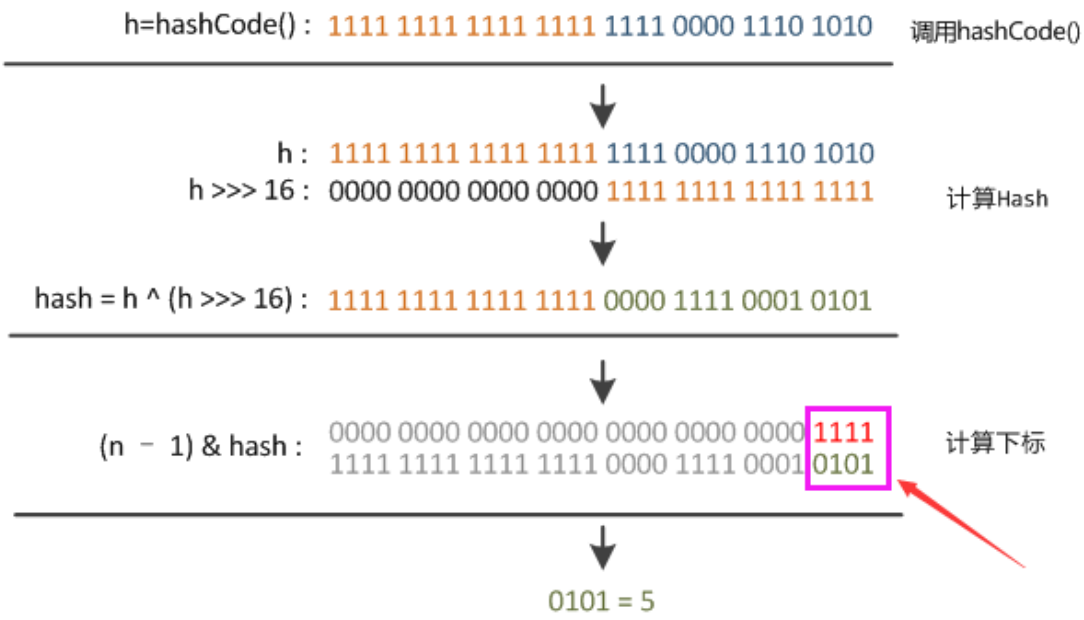
尽量打乱 hashCode 真正参与运算的低16位，减少hash 碰撞。

之所以要无符号右移16位，是跟table的下标有关，位置计算方式是：

（n-1)&hash 计算 Node 的存储位置

假如n=16，从下图可以看出：

table的下标仅与hash值的低n位有关，hash值的高位都被与操作置为0了，只有hash值的低4位参与了运算。



putVal方法源码

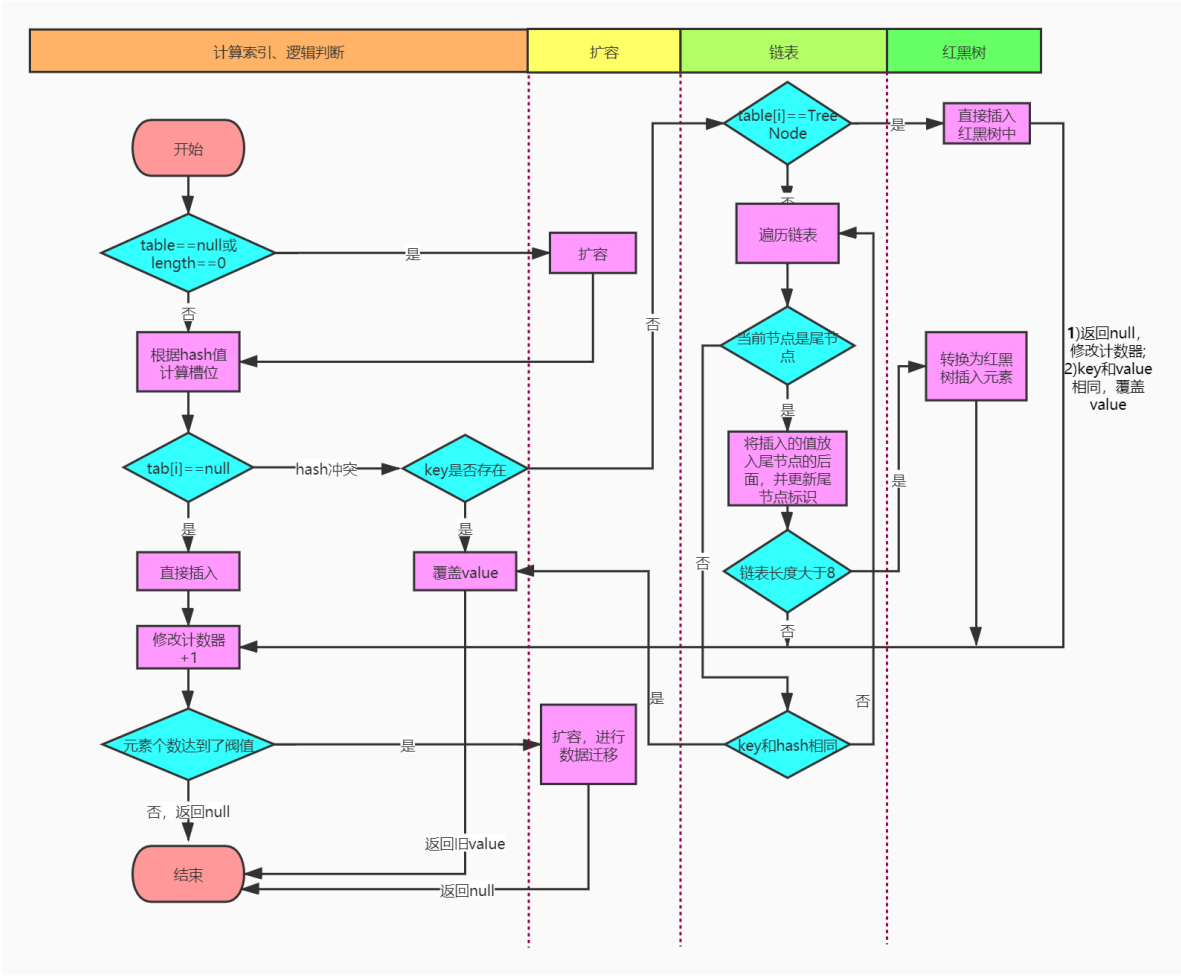
putVal：

而当链表长度太长（默认超过 8）时，链表就进行转换红黑树的操作。

这里利用**红黑树快速增删改查**的特点，提高 HashMap 的性能。

当红黑树结点个数少于 6 个的时候，又会将红黑树转化为链表。

因为在数据量较小的情况下，红黑树要维护平衡，比起链表来，性能上的优势并不明显。




```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //此时 table 尚未初始化，通过 resize 方法得到初始化的table
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // （n-1)&hash 计算 Node 的存储位置，如果判断 Node 不在哈希表中(链表的第一个节点位置），新增一个 Node，并加入到哈希表中
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {//hash冲突了
        Node<K,V> e; K k;
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;//判断key的条件是key的hash相同和eqauls方法符合，p.key等于插入的key，将p的引用赋给e
        else if (p instanceof TreeNode)// p是红黑树节点，插入后仍然是红黑树节点，所以直接强制转型p后调用putTreeVal，返回的引用赋给e
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {//链表
            // 循环，直到链表中的某个节点为null，或者某个节点hash值和给定的hash值一致且key也相同，则停止循环。
            for (int binCount = 0; ; ++binCount) {//binCount是一个计数器，来计算当前链表的元素个数
                if ((e = p.next) == null) {//next为空，将添加的元素置为next
                    p.next = newNode(hash, key, value, null);
                    //插入成功后，要判断是否需要转换为红黑树，因为插入后链表长度+1，而binCount并不包含新节点，所以判断时要将临界阀值-1.【链表长度达到了阀值TREEIFY_THRESHOLD=8，即链表长度达到了7】
```

```
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        // 如果链表长度达到了8，且数组长度小于64，那么就重新散列resize()，如果大于64，则创建红黑树，将链表转换为红黑树
                        treeifyBin(tab, hash);
                    //结束循环
                    break;
                }
                //节点hash值和给定的hash值一致且key也相同，停止循环
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                //如果给定的hash值不同或者key不同。将next值赋给p，为下次循环做铺垫。即结束当前节点，对下一节点进行判断
                p = e;
            }
        }
        //如果e不是null，该元素存在了(也就是key相等)
        if (e != null) { // existing mapping for key
            // 取出该元素的值
            V oldValue = e.value;
            // 如果 onlyIfAbsent 是 true，就不用改变已有的值；如果是false(默认)，或者value是null，将新的值替换老的值
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            //什么都不做
            afterNodeAccess(e);
            //返回旧值
            return oldValue;
        }
    }
    //修改计数器+1，为迭代服务
    ++modCount;
    //达到了边界值，需要扩容
    if (++size > threshold)
        resize();
    //什么都不做
    afterNodeInsertion(evict);
    //返回null
    return null;
}
```

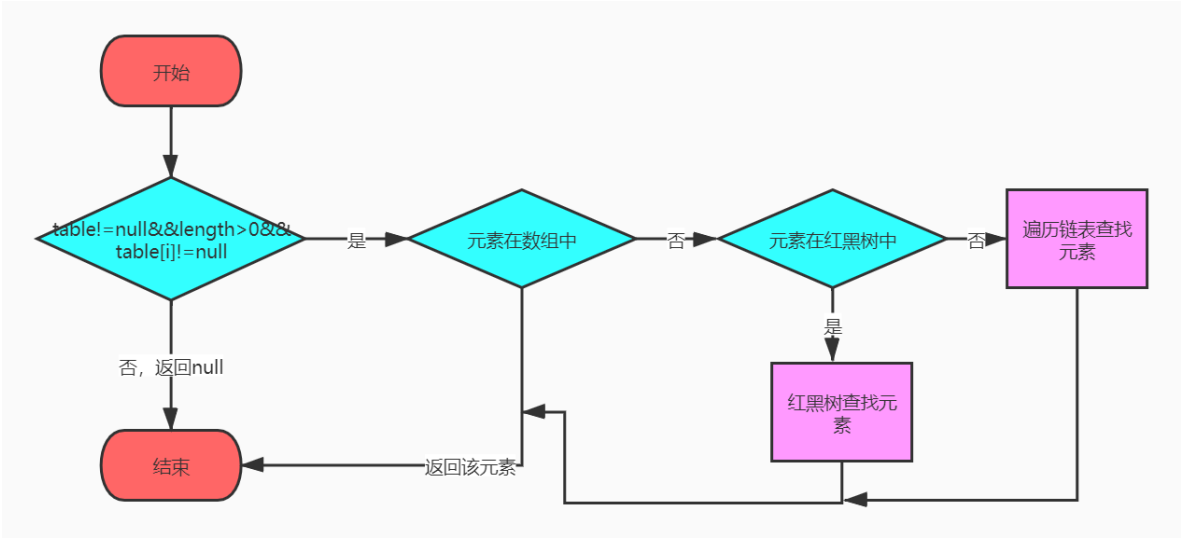
get方法源码：

当 HashMap 只存在数组，而数组中没有 Node 链表时，是 HashMap 查询数据性能最好的时候。

一旦发生大量的哈希冲突，就会产生 Node 链表，这个时候每次查询元素都可能遍历 Node 链表，从而降低查询数据的性能。

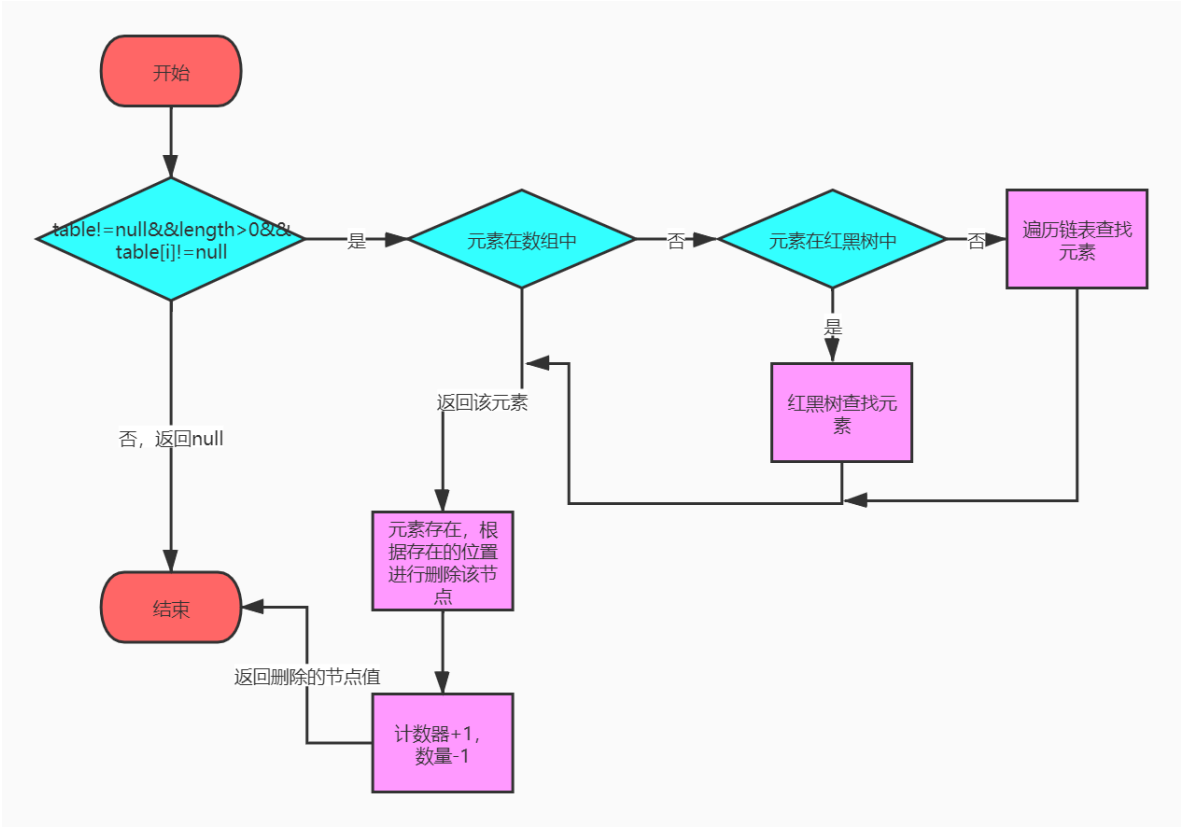
特别是在链表长度过长的情况下，性能明显下降，**使用红黑树**就很好地解决了这个问题，

红黑树使得查询的平均复杂度降低到了 **O(log(n))**，链表越长，使用红黑树替换后的查询效率提升就越明显。




```java
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    //数组不为null，数组长度大于0，根据hash计算出来的槽位的元素不为null
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        //查找的元素在数组中，返回该元素
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {//查找的元素在链表或红黑树中
            if (first instanceof TreeNode)//元素在红黑树中，返回该元素
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {//遍历链表，元素在链表中，返回该元素
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    //找不到返回null
    return null;
}
```

remove方法源码：




```java
public V remove(Object key) {
    Node<K,V> e;
    return (e = removeNode(hash(key), key, null, false, true)) == null ?
        null : e.value;
}
final Node<K,V> removeNode(int hash, Object key, Object value,
                           boolean matchValue, boolean movable) {
    Node<K,V>[] tab; Node<K,V> p; int n, index;
    //数组不为null，数组长度大于0，要删除的元素计算的槽位有元素
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (p = tab[index = (n - 1) & hash]) != null) {
        Node<K,V> node = null, e; K k; V v;
        //当前元素在数组中
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;
        //元素在红黑树或链表中
        else if ((e = p.next) != null) {
            if (p instanceof TreeNode)//是树节点，从树种查找节点
                node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
            else {
                do {
                    //hash相同，并且key相同，找到节点并结束
                    if (e.hash == hash &&
                        ((k = e.key) == key ||
                         (key != null && key.equals(k)))) {
                        node = e;
                        break;
                    }
                    p = e;
                } while ((e = e.next) != null);//遍历链表
            }
        }
```

```
        //找到节点了，并且值也相同
        if (node != null && (!matchValue || (v = node.value) == value ||
                             (value != null && value.equals(v)))) {
            if (node instanceof TreeNode)//是树节点，从树中移除
                ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
            else if (node == p)//节点在数组中，
                tab[index] = node.next;//当前槽位置为null，node.next为null
            else//节点在链表中
                p.next = node.next;//将节点删除
            ++modCount;//修改计数器+1，为迭代服务
            --size;//数量-1
            afterNodeRemoval(node);//什么都不做
            return node;//返回删除的节点
        }
    }
    return null;
}
```

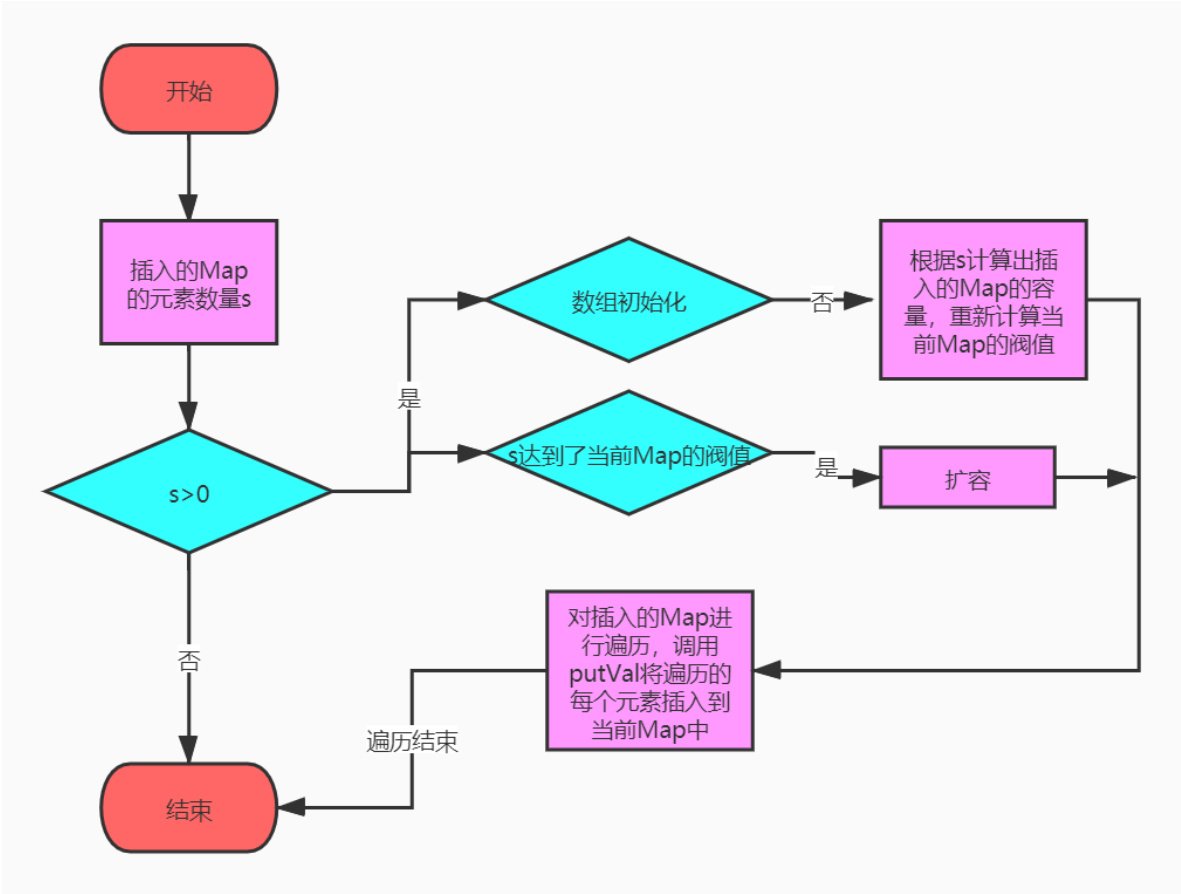
containsKey方法：

```
public boolean containsKey(Object key) {
    return getNode(hash(key), key) != null;//查看上面的get的getNode
}
```

containsValue方法：

```
public boolean containsValue(Object value) {
    Node<K,V>[] tab; V v;
    //数组不为null并且长度大于0
    if ((tab = table) != null && size > 0) {
        for (int i = 0; i < tab.length; ++i) {//对数组进行遍历
            for (Node<K,V> e = tab[i]; e != null; e = e.next) {
                //当前节点的值等价查找的值，返回true
                if ((v = e.value) == value ||
                    (value != null && value.equals(v)))
                    return true;
            }
        }
    }
    return false;//找不到返回false
}
```

putAll方法：




```java
public void putAll(Map<? extends K, ? extends V> m) {
    putMapEntries(m, true);
}
final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
    int s = m.size();//获得插入整个m的元素数量
    if (s > 0) {
        if (table == null) { // pre-size，当前map还没有初始化数组
            float ft = ((float)s / loadFactor) + 1.0F;//m的容量
            //判断容量是否大于最大值MAXIMUM_CAPACITY
            int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                     (int)ft : MAXIMUM_CAPACITY);
            //容量达到了边界值，比如插入的m的定义容量是16，但当前map的边界值是12，需要对当前map进行重新计算边界值
            if (t > threshold)
                threshold = tableSizeFor(t);//重新计算边界值
        }
        else if (s > threshold)//存放的数量达到了边界值，扩容
            resize();
        //对m进行遍历，放到当前map中
        for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
            K key = e.getKey();
            V value = e.getValue();
            putVal(hash(key), key, value, false, evict);
        }
    }
}
```

clear方法：

```
public void clear() {
    Node<K,V>[] tab;
    modCount++;//修改计数器+1，为迭代服务
    if ((tab = table) != null && size > 0) {
        size = 0;//将数组的元素格式置为0，然后遍历数组，将每个槽位的元素置为null
        for (int i = 0; i < tab.length; ++i)
            tab[i] = null;
    }
}
```

replace方法：

```
public boolean replace(K key, V oldValue, V newValue) {
    Node<K,V> e; V v;
    //根据hash计算得到槽位的节点不为null，并且节点的值等于旧值
    if ((e = getNode(hash(key), key)) != null &&
        ((v = e.value) == oldValue || (v != null && v.equals(oldValue)))) {
        e.value = newValue;//覆盖旧值
        afterNodeAccess(e);
        return true;
    }
    return false;
}

public V replace(K key, V value) {
    Node<K,V> e;
    //根据hash计算得到槽位的节点不为null
    if ((e = getNode(hash(key), key)) != null) {
        V oldValue = e.value;//节点的旧值
        e.value = value;//覆盖旧值
        afterNodeAccess(e);
        return oldValue;//返回旧值
    }
    return null;//找不到key对应的节点
}
```

HashMap要点分析

HashMap允许键值对为null;

HashMap允许键值对为null;

HashTable则不允许，会报空指针异常;

```
HashMap<String, String> map= new HashMap<>(2);
map.put(null,null);
map.put("1",null);
```

HashMap是由一个 Node 数组组成的，每个 Node 包含了一个 key-value 键值对：

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

Node 类作为 HashMap 中的一个内部类，除了 key、value 两个属性外，还定义了一个next 指针，当有哈希冲突时，

HashMap 会用之前数组当中相同哈希值对应存储的 Node 对象，通过指针指向新增的相同哈希值的 Node 对象的引用。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

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

HashMap 初始容量是16，扩容方式为2N：

在 JDK1.7 中，HashMap 整个扩容过程就是：

> 分别取出数组元素，一般该元素是最后一个放入链表中的元素，然后遍历以该元素为头的单向链表元素，依据每个被遍历元素的 hash 值计算其在新数组中的下标，然后进行交换。

这样的扩容方式，会将原来哈希冲突的单向链表尾部，变成扩容后单向链表的头部。

而在 JDK1.8 后，HashMap 对扩容操作做了优化。

由于扩容数组的长度是2倍关系，

所以对于假设初始 tableSize=4 要扩容到8来说就是 0100 到 1000 的变化（左移一位就是2倍），

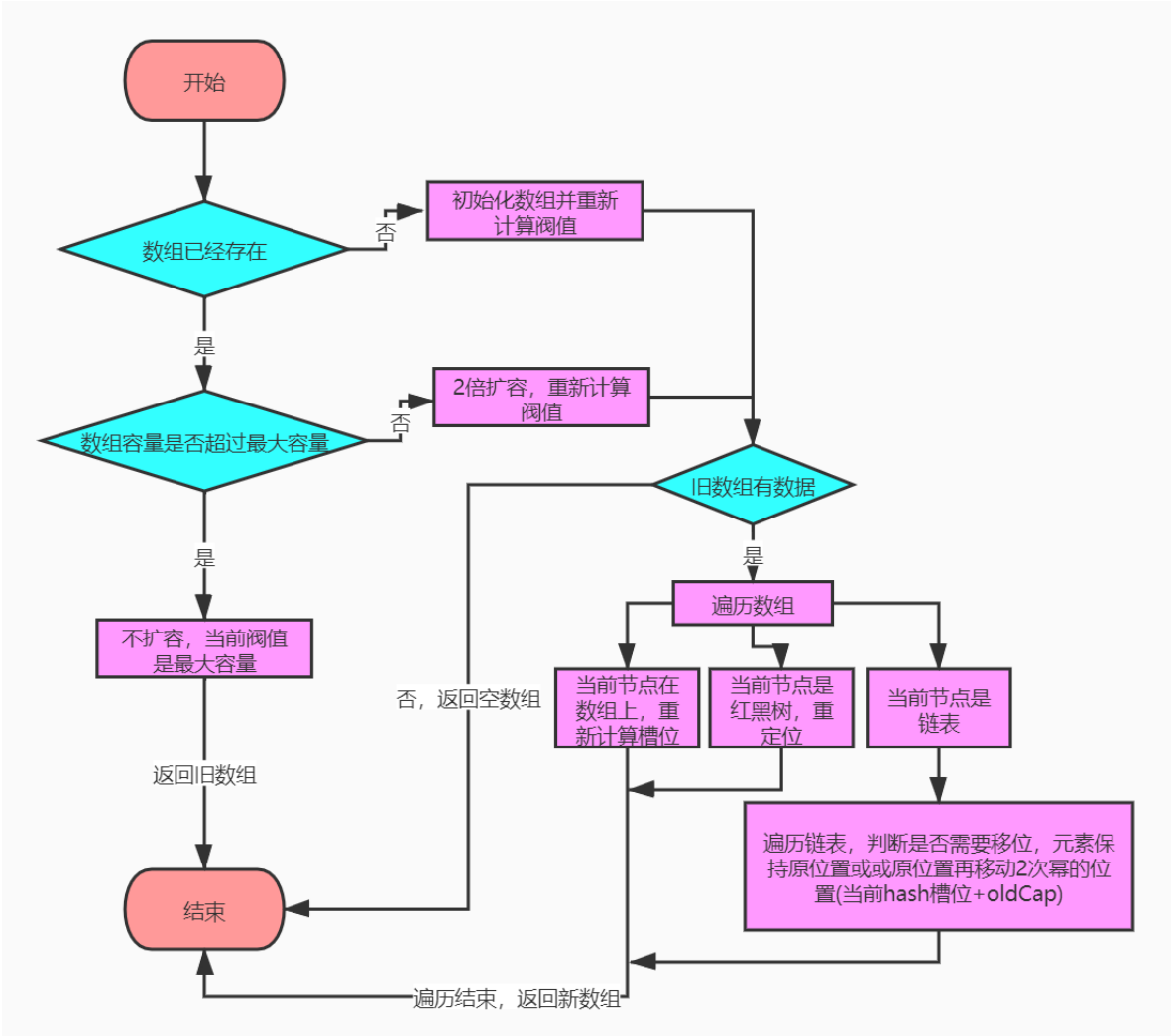
在扩容中只用判断原来的 hash 值和 oldCap（旧数组容量）按位与操作是 0 或 1 就行:

- 0的话索引不变，
- 1的话索引变成原索引加扩容前数组。

之所以能通过这种“与”运算来重新分配索引，

是因为 hash 值本来是随机的，而 hash 按位与上 oldCap 得到的 0 和 1 也是随机的，

所以扩容的过程就能把之前哈希冲突的元素再随机分布到不同的索引中去。




```java
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16，默认大小
//元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;//原先的数组，旧数组
    int oldCap = (oldTab == null) ? 0 : oldTab.length;//旧数组长度
    int oldThr = threshold;//阀值
    int newCap, newThr = 0;
    if (oldCap > 0) {//数组已经存在不需要进行初始化
        if (oldCap >= MAXIMUM_CAPACITY) {//旧数组容量超过最大容量限制，不扩容直接返回旧数组
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        //进行2倍扩容后的新数组容量小于最大容量和旧数组长度大于等于16
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold，重新计算阀值为原来的2倍
    }
    //初始化数组
    else if (oldThr > 0) // initial capacity was placed in threshold，有阀值，初始容量的值为阀值
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults，没有阀值
        newCap = DEFAULT_INITIAL_CAPACITY;//初始化的默认容量
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);//重新计算阀值
```

```
    }
    //有阀值，定义了新数组的容量，重新计算阀值
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;//赋予新阀值
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];//创建新数组
    table = newTab;
    if (oldTab != null) {//如果旧数组有数据，进行数据移动，如果没有数据，返回一个空数
组
        for (int j = 0; j < oldCap; ++j) {//对旧数组进行遍历
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;//将旧数组的所属位置的旧元素清空
                if (e.next == null)//当前节点是在数组上，后面没有链表，重新计算槽位
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)//当前节点是红黑树，红黑树重定位
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order，当前节点是链表
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    //遍历链表
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {//不需要移位
                            if (loTail == null)//头节点是空的
                                loHead = e;//头节点放置当前遍历到的元素
                            else
                                loTail.next = e;//当前元素放到尾节点的后面
                            loTail = e;//尾节点重置为当前元素
                        }
                        else {//需要移位
                            if (hiTail == null)//头节点是空的
                                hiHead = e;//头节点放置当前遍历到的元素
                            else
                                hiTail.next = e;//当前元素放到尾节点的后面
                            hiTail = e;//尾节点重置为当前元素
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {//不需要移位
                        loTail.next = null;
                        newTab[j] = loHead;//原位置
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;//移动到当前hash槽位 +
oldCap的位置，即在原位置再移动2次幂的位置
                    }
                }
            }
        }
    }
    return newTab;
```

```
79  }
```

当前节点是数组，后面没有链表，重新计算槽位:位与操作的效率比效率高

```
    定位槽位：e.hash & (newCap - 1)
    我们用长度16，待插入节点的hash值为21举例：
    (1)取余：21 % 16 = 5
    (2)位与：
    21：0001 0101
           &
    15：0000 1111
    5：  0000 0101
```

遍历链表，对链表节点进行移位判断：(e.hash & oldCap) == 0

```
比如oldCap=8,hash是3，11，19，27时，
（1）JDK1.8中(e.hash & oldCap)的结果是0，8，0，8，这样3，19组成新的链表，index为3；而11，27组成新的链表，新分配的index为3+8；
（2）JDK1.7中是(e.hash & newCap-1)，newCap是oldCap的两倍，也就是3，11，19，27对(16-1)与计算，也是0，8，0，8，但由于是使用了单链表的头插入方式，即同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这样index为3的链表是19，3，index为3+8的链表是 27，11。
也就是说1.7中经过resize后数据的顺序变成了倒叙，而1.8没有改变顺序。
```

HashMap总结

HashMap 通过哈希表数据结构的形式存储键值对，这种设计的好处就是查询键值对的效率高；

我们在编码中可以优化 HashMap的性能，例如重写 key 的 hashCode 方法，降低哈希冲突，从而减少链表的产生，高效利用哈希表，达到提高性能的效果。

我们在使用 HashMap 时，可以结合自己的场景来设置初始容量和加载因子两个参数。当查询操作较为频繁时，可以适当地减少加载因子；如果对内存利用率要求比较高，可以适当的增加加载因子；

我们可以在预知存储数据量的情况下，提前设置初始容量（初始容量=预知数据量/加载因子），这样做的好处是可以减少 resize() 操作，提高 HashMap 的效率；

HashMap 使用了数组+链表这两种数据结构相结合的方式实现了链地址法，当有哈希值冲突时，就可以将冲突的键值对链成一个链表。但这种方式存在一个性能问题，如果链表过长，查询数据的时间复杂度就会增加。所以 HashMap 在JDK1.8中使用了红黑树来解决链表过长导致的查询性能下降问题。

HashMap的面试题

hash的基本概念就是把任意长度的输入通过一个hash算法之后，映射成固定长度的输出

问：hash冲突可以避免么?

理论上是没有办法避免的，就类比“抽屉原理”，

比如说一共有10个苹果，但是咱一共有9个抽屉，最终一定会有一个抽屉里的数量是大于1的，

所以hash冲突没有办法避免，只能尽量避免。

问：好的hash算法考虑的点，应该是哪些呢?

首先这个hash算法，它一定效率得高，要做到长文本也能高效计算出hash值，

这二点就是hash值不能让它逆推出原文吧；

两次输入，只要有一点不同，它也得保证这个hash值是不同的。

其次，就是尽可能的要分散吧，因为，在table中slot中slot大部分都处于空闲状的，要尽可能降低hash冲突。

问：HashMap中存储数据的结构，长什么样啊?

JDK1.7 是 数组 + 链表；
JDK1.8是 数组 + 链表 + 红黑树，每个数据单元都是一个Node结构，Node结构中有key字段、有value字段、还有next字段、还有hash字段。

Node结构next字段就是发生hash冲突的时候，当前桶位中node与冲突的node连成一个链表要用的字段。

问：HashMap 的底层数据结构是怎样的？

JDK1.8 之前

- JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列。**
- HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。
- 所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK1.8 之后

当链表长度大于阈值（默认为 8）时，会首先调用 treeifyBin()方法。这个方法会根据 HashMap 数组来决定是否转换为红黑树。只有当数组长度大于或者等于 64 的情况下，才会执行转换红黑树操作，以减少搜索时间。否则，就是只是执行 resize() 方法对数组扩容。

问：hashmap中的这个散列表数组长度，那初始长度是多少啊?

初始长度默认是16

问：那这个散列表，new HashMap() 的时候就创建了，还是说在什么时候创建的?

散列表是懒加载机制，

只有第一次put数据的时候，它才创建的

问：默认的负载因子是多少？ 并且这个负载因子有啥用?

默认负载因子0.75，就是75%，

负载因子它的作用就是计算扩容阈值用的，

比如使用无参构造方法创建的hashmap对象，它默认情况下扩容阈值就 16*0.75 = 12

问：链表它转化为这个红黑树需在达到什么条件?

链表转红黑树，主要是有两个指标，其中一个就是链表长度达到8，还有一个指标就是当前散列表数组长度它已经达到64。

如果前散列表数组长度它已经达到64，就算slot内部链表长度到了8，它也不会链转树，

它仅仅会发生一次resize，散列表扩容。

问：HashMap 的扩容机制是怎样的?

一般情况下，当元素数量超过阈值时便会触发扩容。每次扩容的容量都是之前容量的 2 倍。

HashMap 的**容量**是有上限的，必须小于 1<<30，即 1073741824。

如果容量超出了这个数，则不再增长，且**阈值**会被设置为 Integer.MAX_VALUE。

JDK7 中的扩容机制

- 空参数的构造函数：以默认容量、默认负载因子、默认阈值初始化数组。内部数组是空数组。
- 有参构造函数：根据参数确定容量、负载因子、阈值等。
- 第一次 put 时会初始化数组，其容量变为不小于指定容量的 2 的幂数，然后根据负载因子确定阈值。
- 如果不是第一次扩容，则 **新容量=旧容量 x 2 ， 新阈值=新容量 x 负载因子** 。

JDK8 的扩容机制

- 空参数的构造函数：实例化的 HashMap 默认内部数组是 null，即没有实例化。第一次调用 put 方法时，则会开始第一次初始化扩容，长度为 16。
- 有参构造函数：用于指定容量。会根据指定的正整数找到不小于指定容量的 2 的幂数，

将这个数设置赋值给阈值（threshold）。第一次调用 put 方法时，会将阈值赋值给容量，然后让 **阈值 = 容量 x 负载因子。**

- 如果不是第一次扩容，则容量变为原来的 2 倍，阈值也变为原来的 2 倍。（容量和阈值都变为原来的 2 倍时，负载因子还是不变）。

此外还有几个细节需要注意：

- 首次 put 时，先会触发扩容（算是初始化），然后存入数据，然后判断是否需要扩容；

- 不是首次 put，则不再初始化，直接存入数据，然后判断是否需要扩容；

问：Node对象hash值与key对象的hashcode() 有什么关系?

Node对象hash值是key.hashcode二次加工得到的。

加工原则是：

key的hashcode 高16位 ^ 低16位，得到的一个新值。

问：hashCode值为什么需要高16位 ^ 低16位

主要为了优化hash算法，近可能的分散得比较均匀，尽可能的减少 碰撞

因为hashmap内部散列表，它大多数场景下，它不会特别大。

hashmap内部散列表的长度，也就是说 length - 1 对应的 二进制数，实际有效位很有限，一般都在（低）16位以内，

注意：2的16次方为 64K

这样的话，key的hash值高16位就等于完全浪费了，没起到作用。

所以，node的hash字段才采用了 高16位 异或 低16位 这种方式来增加随机的概率，近可能的分散得比较均匀，尽可能的减少 碰撞

问：hashmap Put写数据的具体流程，尽可能的详细点去说

主要为4种情况：
前面这个，寻址算法是一样的，都是根据key的hashcode 经过 高低位 异或 之后的值，然后再 按位与 & （table.length -1），得到一个槽位下标，然后根据这个槽内状况，状况不同，情况也不同，大概就是4种状态，

第一种是slot == null，直接占用slot就可以了，然后把当前put方法传进来的key和value包状成一个Node 对象，放到这个slot中就可以了

第二种是slot != null 并且 它引用的node 还没有链化；需要对比一下，node的key 与当前put 对象的key 是否完全相等；

如果完全相等的话，这个操作就是replace操作，就是替换操作，把那个新的value替换当前slot -> node.value 就可以了；

否则的话，这次put操作就是一个正儿八经的hash冲突了，slot->node 后面追加一个node就可以了，采用尾插法。

第三种就是slot 内的node已经链化了；

这种情况和第二种情况处理很相似，首先也是迭代查找node，看看链表上的元素的key，与当前传来的key是不是完全一致。如果一致的话，还是repleace操作，替换当前node.value，否则的话就是我们迭代到链表尾节点也没有匹配到完全一致的node，把put数据包装成node追加到链表尾部；

这块还没完，还需要再检查一下当前链表长度，有没有达到树化阈值，如果达到阈值的话，就调用一个树化方法，树化操作都在这个方法里完成

第四种就是冲突很严重的情况下，就是那个链已经转化成红黑树了

问：jdk8 HashMap为什么要引入红黑树呢?

其实主要就是解决hash冲突导致链化严重的问题，如果链表过长，查找时间复杂度为O(n)，效率变慢。

本身散列表最理想的查询效率为O(1)，但是链化特别严重，就会导致查询退化为O(n)。

严重影响查询性能了，为了解决这个问题，JDK1.8它才引入的红黑树。红黑树其实就是一颗特殊的二叉排序树，这个时间复杂度是log(N)

问：那为什么链化之后性能就变低了呀?

因为链表它毕竟不是数组，它从内存角度来看，它没有连续着。

如果我们要往后查询的话，要查询的数据它在链表末尾，那只能从链表一个节点一个节点Next跳跃过去，非常耗费性能。

问：再聊聊hashmap的扩容机制吧？你说一下，什么情况下会触发这个扩容呢?

在写数据之后会触发扩容，可能会触发扩容。hashmap结构内，我记得有个记录当前数据量的字段，这个数据量字段达到扩容阈值的话，下一个写入的对象是在列表才会触发扩容

问：扩容后会扩容多大呢？这块算法是咋样的呢?

因为table 数组长度必须是2的次方数嘛，扩容其实，每次都是按照上一次的tableSize位移运算得到的。就是做一次左移1位运算，假设当前tableSize是16的话，16 << 1 == 32

问：这里为什么要采用位移运算呢？咋不直接tableSize乘以2呢?

主要是因为性能，因为cpu毕竟它不支持乘法运算，所有乘法运算它最终都是在指令层面转化为加法实现的。

效率很低，如果用位运算的话对cpu来说就非常简洁高效

问：创建新的扩容数组，老数组中的这个数据怎么迁移呢?

迁移其实就是，每个桶位推进迁移，就是一个桶位一个桶位的处理；

主要还是看当前处理桶位的数据状态吧

聊聊：HashMap为什么从链表换成了树？ 为啥不用AVL树?

上一节我们在阅读源码的时候，发现这样一句话：

```
if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
      treeifyBin(tab, hash);
```

当链表节点的计数超过TREEIFY_THRESHOLD - 1则将该链表树化，为什么要这样呢?

其实比较一下链表和树的优缺点就能大致明白该优化的目的。

我们假设一条链表上有10个节点，在查询时，最坏情况需要查询10次，N(10)。

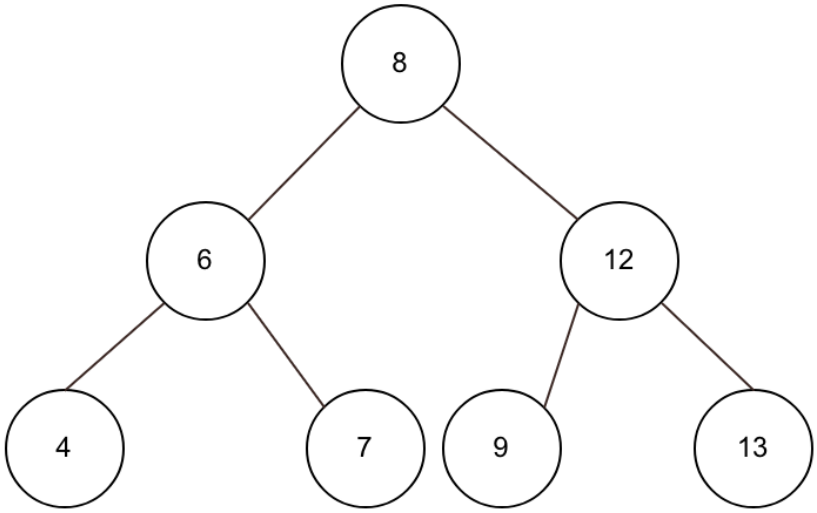
对于树而言不同的树复杂度不同，但是对于最基本的二叉树：

> 左子树一定比root小，右子树一定比root大，

相当于是通过二分法在进行查找，查询速度绝大部分时候比链表要快。

完美的情况下二叉搜索树 BST

一般人们理解的二叉树（**又叫二叉搜索树 BST**）会出现一个问题，完美的情况下，它是这样的：

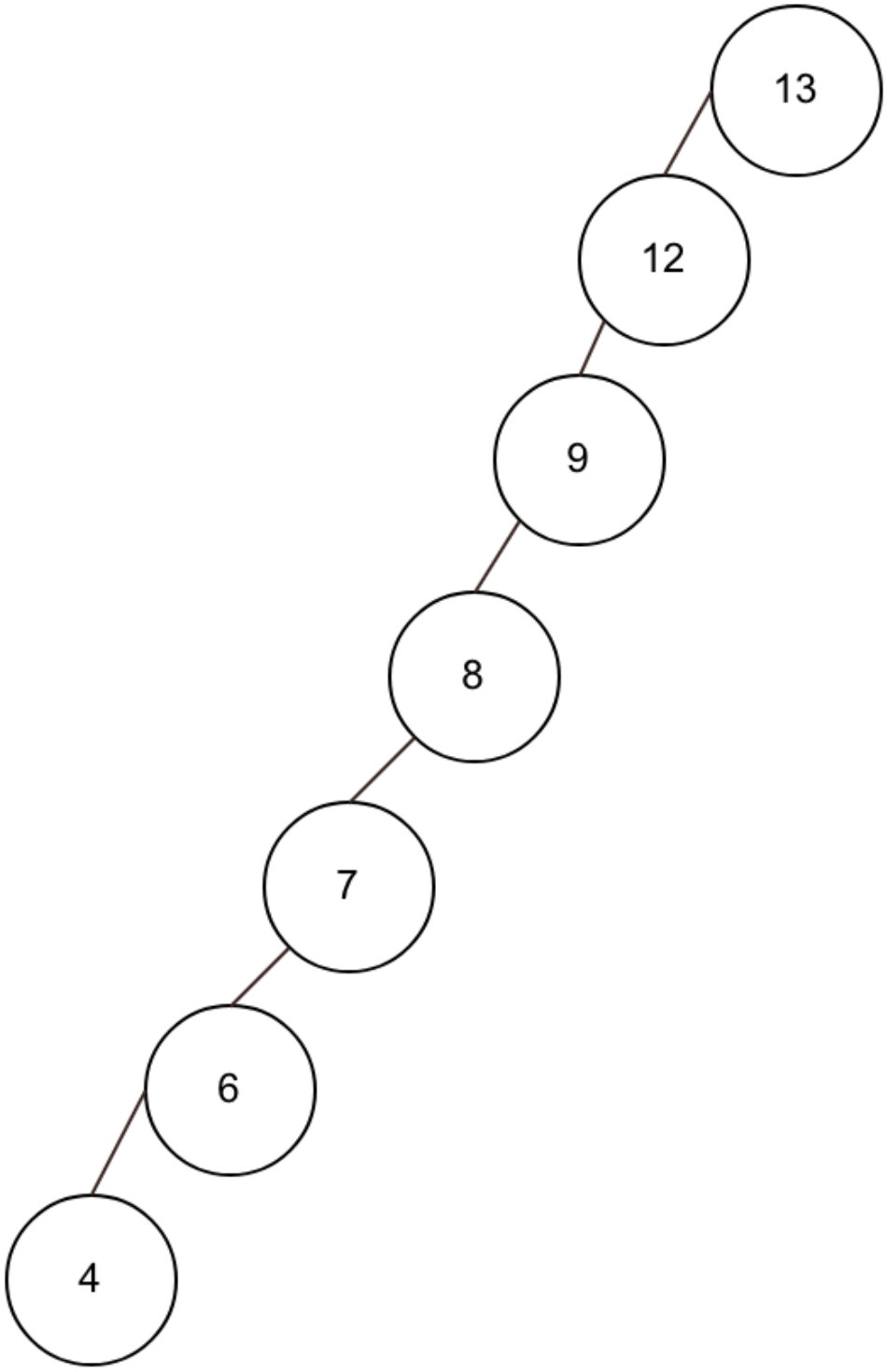


但是也有可能出现这样一种情况：

树的节点正好从大到小的插入，此时树的结构也类似于链表结构，这时候的查询或写入耗时与链表相同。

退化成为了 链表的特殊BST

一颗特殊BST，退化成为了 链表，如下图：



为了避免这种特殊的情况发生，引入了平衡二叉树（AVL）和红黑树（red-black tree）。

它们都是通过本身的建树原则来控制树的层数和节点位置，因为rbtree是由AVL演变而来，所以我们从了解AVL开始。

从平衡二叉树到红黑树

平衡二叉树

平衡二叉树也叫AVL（发明者名字简写），也属于二叉搜索树的一种，与其不同的是AVL通过机制保证其自身的平衡。

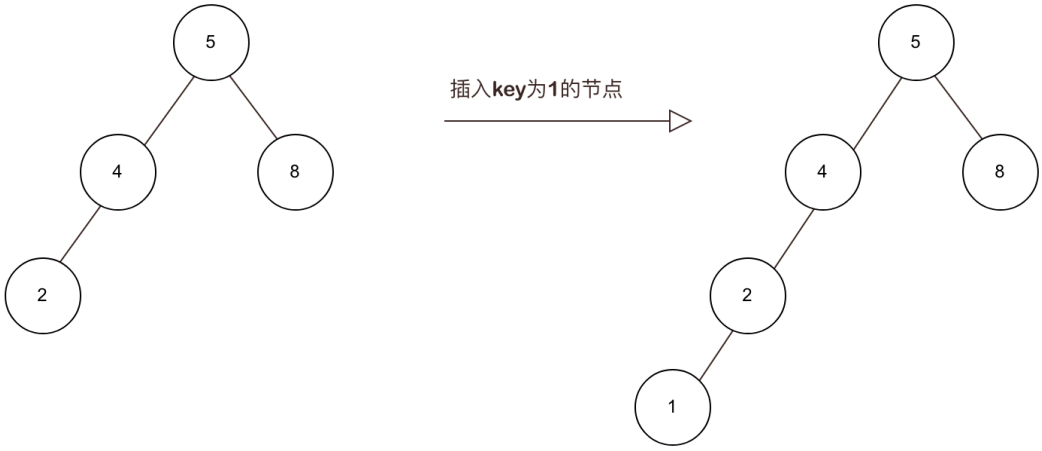
平衡二叉树的原则有以下几点：

- 对于根结点而言，它的左子树任何节点的key一定比其小而右子树任何节点的key一定比其大；
- 对于AVL树而言，其中任何子树仍然是AVL树；
- 每个节点的左右子节点的高度之差的绝对值最多为1；

在插入、删除树节点的时候，如果破坏了以上的原则，**AVL树会自动进行调整**使得以上三条原则仍然成立。

举个例子，下左图为AVL树最长的2节点与最短的8节点高度差为1；

当插入一个新的节点后，根据上面第一条原则，它会出现在2节点的左子树，但这样一来就违反了原则3。



此时AVL树会通过节点的旋转进行调整，AVL调整的过程称之为左旋和右旋，

旋转之前，首先确定旋转点，

这个旋转点就是失去平衡这部分树，在自平衡之后的根节点——pivot，

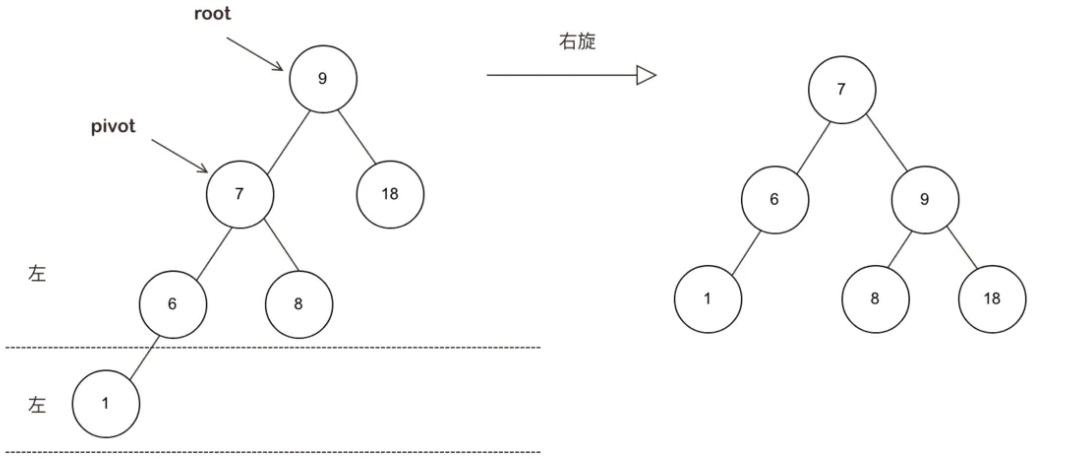
因为我们要根据它来进行旋转。

我们在学习AVL树的旋转时，不要将失衡问题扩大到整个树来看，这样会扰乱你的思路，

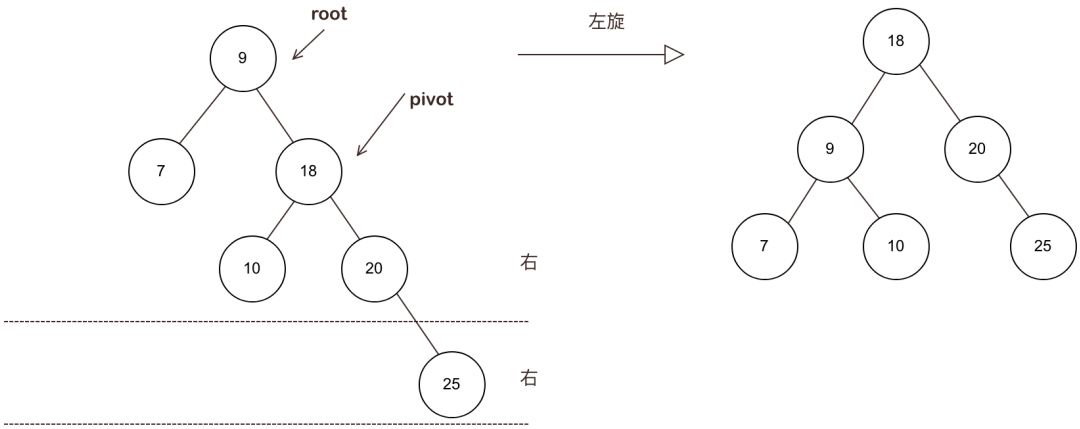
我们只关注失衡子树的根结点及它的子节点和孙子节点即可。

事实上，AVL树的旋转，我们权且叫"AVL旋转"是有规律可循的，因为只要聚焦到失衡子树，那么场景就是有限的4个：

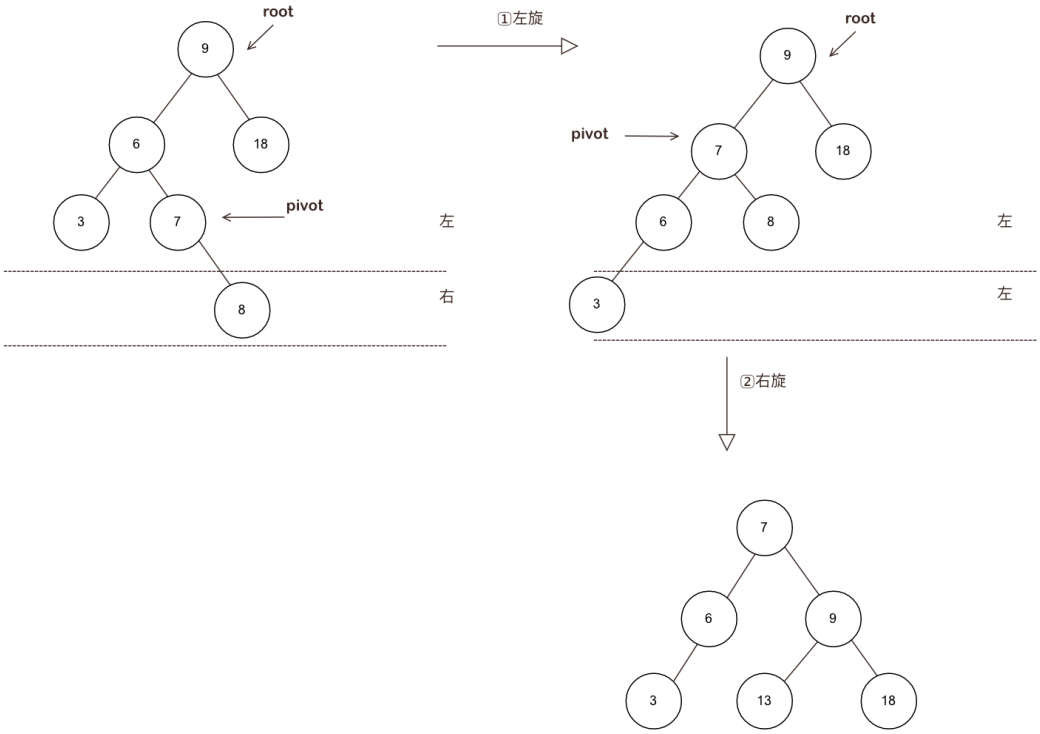
场景1 左左结构（右旋）：



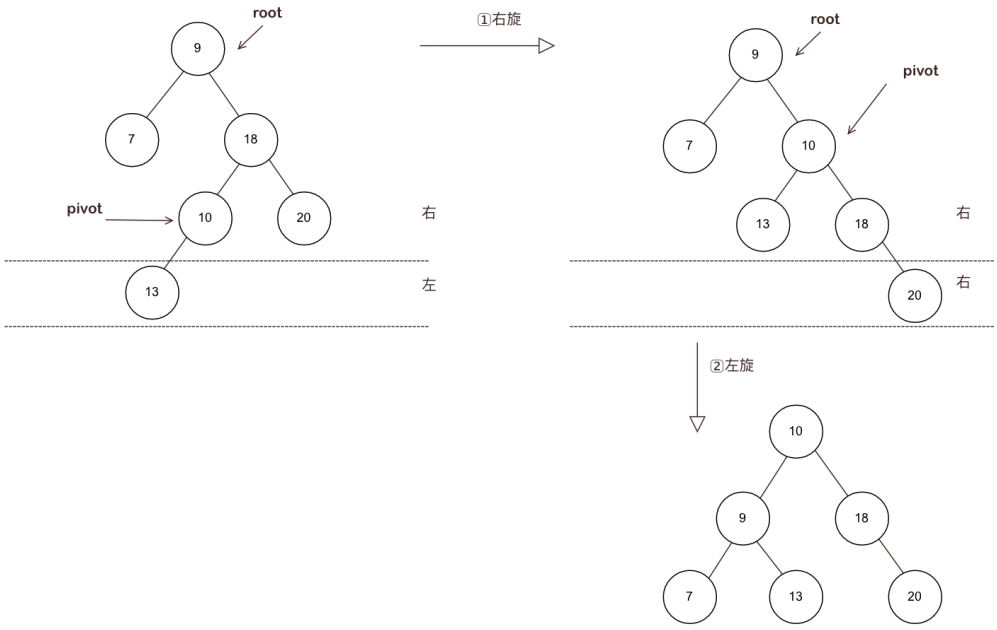
场景2 右右结构（左旋）



场景3 左右结构（左旋+右旋）：



场景4 右左结构（右旋+左旋）：



可见无论哪种情况的失衡，都可以通过旋转来调整。

不难看出，旋转在图上像是将pivot节点向上提（将它提升为root节点），而后两边的节点会物理的分布在新root节点的两边，

接下来按照二叉树的要求：

> 左子树小于root，右子树大于root进行调整。

从图左左结构可以看出，当右旋时原来pivot（7）的右子树会转变到用root点（9）的左子树处；

从图右右结构可见，当左旋时，原来pivot（18）的左子树会分布到原root点（9）的右子树。

对于左右结构和右左结构无非是经过多次旋转达到稳定，旋转的方式并没有区别，

AVL树平衡总结

既然AVL树可以保证二叉树的平衡，这就意味着它最坏情况的时间复杂度O(logn) 要低于普通二叉树和链表的最坏情况O(n)。

那么HashMap就直接使用AVL树来替换链表就好了，为什么选择用红黑树呢?

我们会发现，由于AVL树必须保证Max(最大树高-最小树高) <= 1所以在插入的时候很容易出现不平衡的情况，一旦这样，就需要进行旋转以求达到平衡。

正是由于这种严格的平衡条件，导致需要花大量时间在调整上，故AVL树一般使用场景在于查询而弱于增加删除。

红黑树继承了AVL可自平衡的优点，同时在查询速率和调整耗时中寻找平衡，放宽了树的平衡条件，在实际应用中，红黑树的使用要多得多。

红黑树（RBTree）

红黑树也是一种自平衡二叉查找树，它与AVL树类似，都在添加和删除的时候通过旋转操作保持二叉树的平衡，以求更高效的查询性能。

与AVL树相比，红黑树牺牲了部分平衡性以换取插入/删除操作时少量的旋转操作，整体来说性能要优于AVL树。

红黑树的原则有以下几点：

- 节点非黑即红
- 整个树的根节点一定是黑色
- 叶子节点（包括空叶子节点）一定是黑色
- 每个红色节点的两个子节点都为黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

基于上面的原则，我们一般在插入红黑树节点的时候，会将这个节点设置为红色，原因参照最后一条原则，红色破坏原则的可能性最小，如果是黑色很可能导致这条支路的黑色节点比其它支路的要多1。

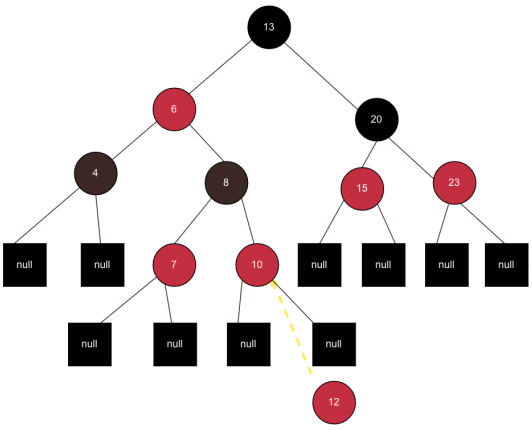
一旦红黑树上述原则有不满足的情况，我们视为平衡被打破，红黑树会通过变色、左旋、右旋的方式恢复平衡。

前文已经详细解释过什么是左旋和右旋，这里就不赘述；变色这个概念很好理解，就是红变黑或黑变红。

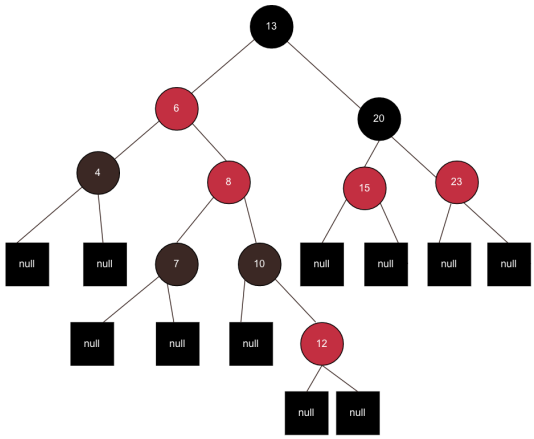
红黑树的平衡过程

但是我们会好奇，红黑树的平衡会不会和上文的AVL树一样，也有可以归纳的平衡场景呢？

答案是肯定的：



①插入新节点



②父节点和叔叔节点红转黑，祖父节点黑转红

场景1 第一次插入：

RBTree第一次插入节点时，新节点会是红色，违背了原则二，直接将颜色变黑即可。

场景2 父节点为黑色：

当插入时节点为红色且父节点为黑色，满足RBTree所有原则，已经平衡。

场景3 父节点为红色且叔叔节点为红色：

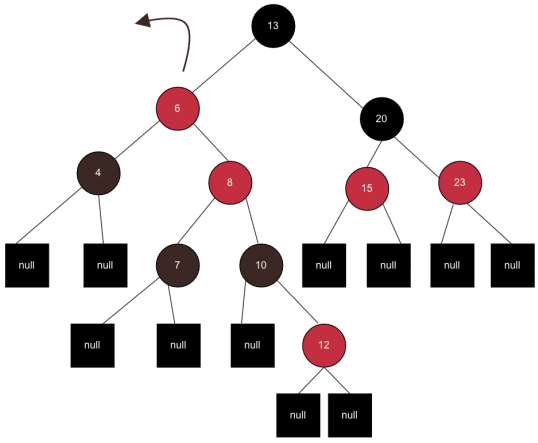
父节点叔叔节点都为红色

在平衡的过程中，要注意红黑树的规定原则。

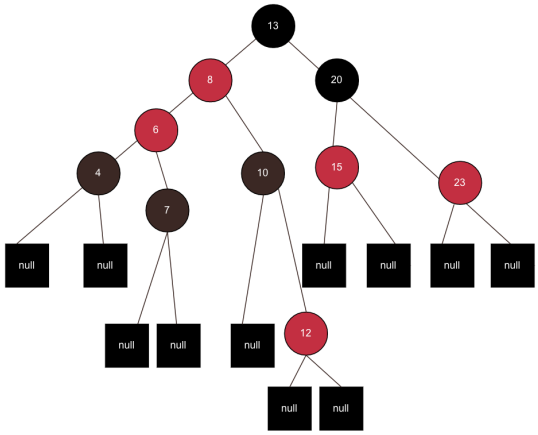
插入红节点，不能仅仅将父节点由红变黑，因为这样会增加这条支路的黑节点数，从而违反**“从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点”**。

将父节点和叔叔节点都变黑，再将祖父节点由黑变红，这样一来，以13为root的红黑树对外黑色节点数没变，对内各条支路节点数一致。

场景4 父节点为红色，叔叔节点为黑色且新节点为右子树：



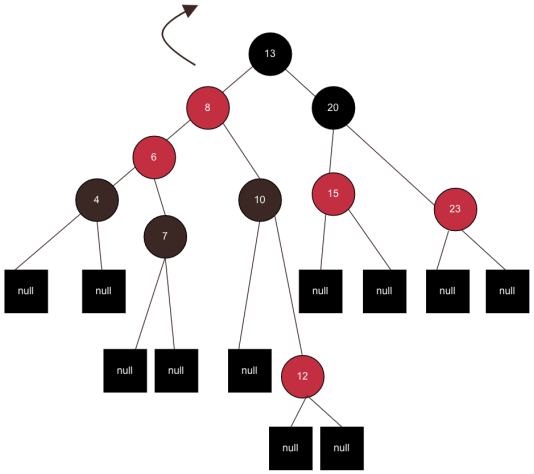
③以节点6为pivot进行左旋



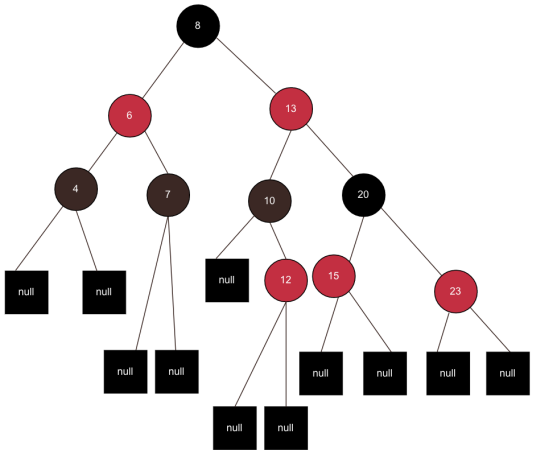
④对于节点6来说父节点为红，叔叔节点为黑，且节点6为节点8的左子树

节点8的父节点为红，叔叔节点为黑，且通过左旋的方式，让整个情况变成下一个场景：父节点红色，叔叔节点为黑色且新节点为左子树。

场景5 父节点为红色，叔叔节点为黑色且新节点为左子树：



⑤以节点8为pivot进行右旋



⑥右旋后将新的root节点8由红变黑，将原root节点13由黑变红

问：红黑树写入操作，是如何找到它的父节点的?

说清楚红黑树，的节点 TreeNode它就是继承Node结构，

TreeNode在Node基础上加了几个字段，分别指向父节点parent，然后指向左子节点left，还有指向右子节点的right，然后还有表示颜色red/black，这个就是TreeNode的基本结构

红黑树的插入操作：

首先是找到一个合适的插入点，就是找到插入节点的父节点，然后这个红黑树 它又满足二叉树的所有排序特性...(满足二叉排序树的所有特性)，这个找父节点的操作和二叉树是完全一致的。

二叉查找树，左子节点小于当前节点，右子节点大于当前节点，然后每一次向下查找一层就可以排除掉一半的数据，插入效率在log(N)

查找的过程也是分情况的，

第一种情况就是一直向下探测，直到查询到左子树或者右子树为null，

说明整个树中，它没有发现node.key与当前put key 一致的这个TreeNode。此时探测节点就是插入父节点所在了，这就找到了父节点；将当前插入节点插入到父节点的左子树或者右子树，，

当然，插入后会破坏平衡，还需要一个红黑树的平衡算法。

第二种情况就是根节点向下探测过程中，发现这个TreeNode.key 与当前 put.key 完全一致。这就不需要插入，替换value就可以了，父节点就是当前节点的父节点

红黑树那几个原则，你还记得么?

- 节点非黑即红
- 整个树的根节点一定是黑色
- 叶子节点（包括空叶子节点）一定是黑色
- 每个红色节点的两个子节点都为黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

问：红黑树的有那些内部操作

变色

把一个红色的节点变成黑色，或者把一个黑色的节点变成红色，就是对这个节点的`变色`。

左旋

与平衡二叉树的旋转操作类似。

问：什么是AVL左旋和右旋?

> 加入节点后，左旋和右旋，维护AVL平衡性

右旋转

场景： 插入的元素在不平衡元素的左侧的左侧

> x.right = y
>
> y.left = xxx(原x.right)

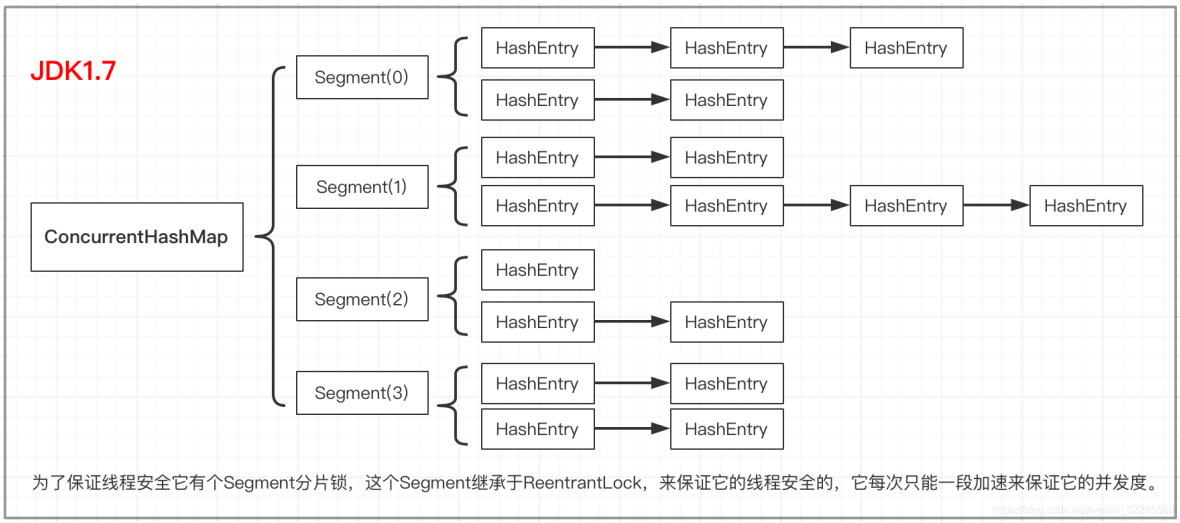
```
对节点y进行向右旋转操作，返回旋转后新的根节点x
        y                               x
       / \                             /   \
      x   T4      向右旋转 (y)        z     y
     / \       - - - - - - - ->      / \   / \
    z   T3                          T1  T2 T3 T4
   / \
  T1  T2
```

> 场景：插入的元素在不平衡元素的右侧的右侧
>
> // 向左旋转过程
>
> x.left = y;
>
> y.right =(原x.left)

```
对节点y进行向左旋转操作，返回旋转后新的根节点x
    y                               x
   /  \                            /   \
  T1   x      向左旋转 (y)        y     z
      / \   - - - - - - - ->     / \   / \
     T2  z                      T1 T2 T3 T4
        / \
       T3 T4
```

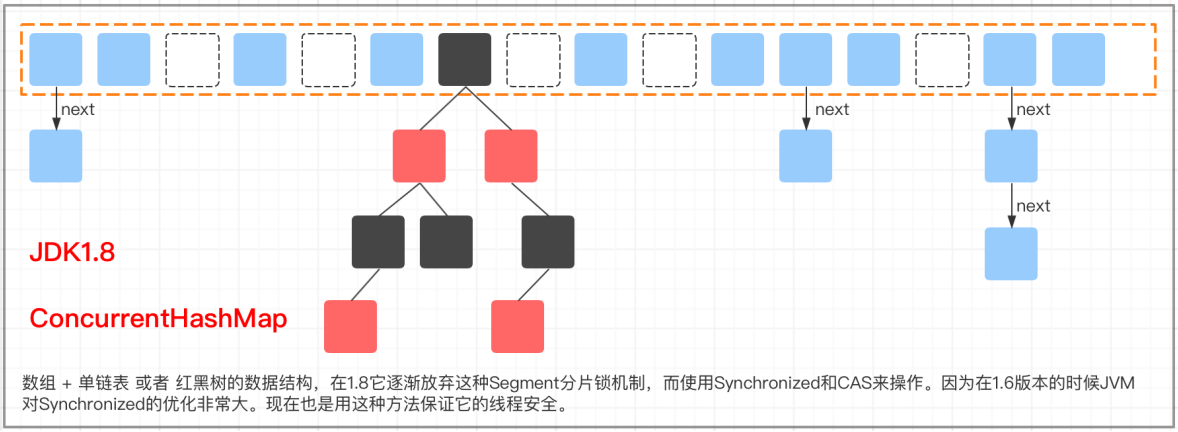
问：聊下ConcurrentHashMap

首先它的数据结构在JDK1.7 版本底层是个分片数组



为了保证线程安全它有个Segment分片锁，这个Segment继承于ReentrantLock，来保证它的线程安全的，它每次只能一段加速来保证它的并发度。

在JDK1.8版本，它改成了与HashMap一样的数据结构，



数组 + 单链表 或者 红黑树的数据结构，

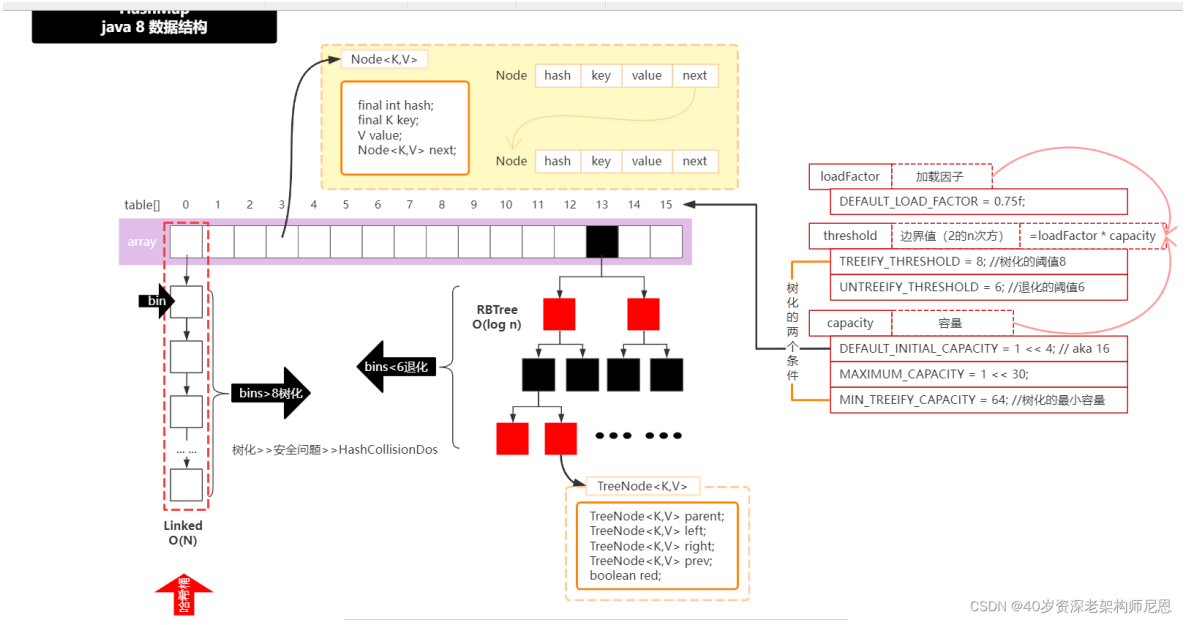
在1.8它逐渐放弃这种Segment分片锁机制，而使用Synchronized和CAS来操作。

因为在1.6版本的时候JVM对Synchronized的优化非常大。

现在也是用这种方法保证它的线程安全。

问：ConcurrentHashMap 的存储结构是怎样的?

- Java7 中 ConcurrnetHashMap 使用的分段锁，也就是每一个 Segment 上同时只有一个线程可以操作，每一个 Segment 都是一个类似 HashMap 数组的结构，它可以扩容，它的冲突会转化为链表。但是 Segment 的个数一但初始化就不能改变，默认 Segment的个数是 16 个。
- Java8 中的 ConcurrnetHashMap 使用的 Synchronized 锁加 CAS 的机制。结构也由 Java7 中的 **Segment 数组 + HashEntry 数组 + 链表** 进化成了 **Node 数组 + 链表 / 红黑树**，Node 是类似于一个 HashEntry 的结构。它的冲突再达到一定大小时会转化成红黑树，在冲突小于一定数量时又退回链表。



https://www.processon.com/view/link/638bfe69e0b34d527931186b

问：说说HashMap底层原理，ConcurrentHashMap与HashMap的区别

HashMap结构及原理

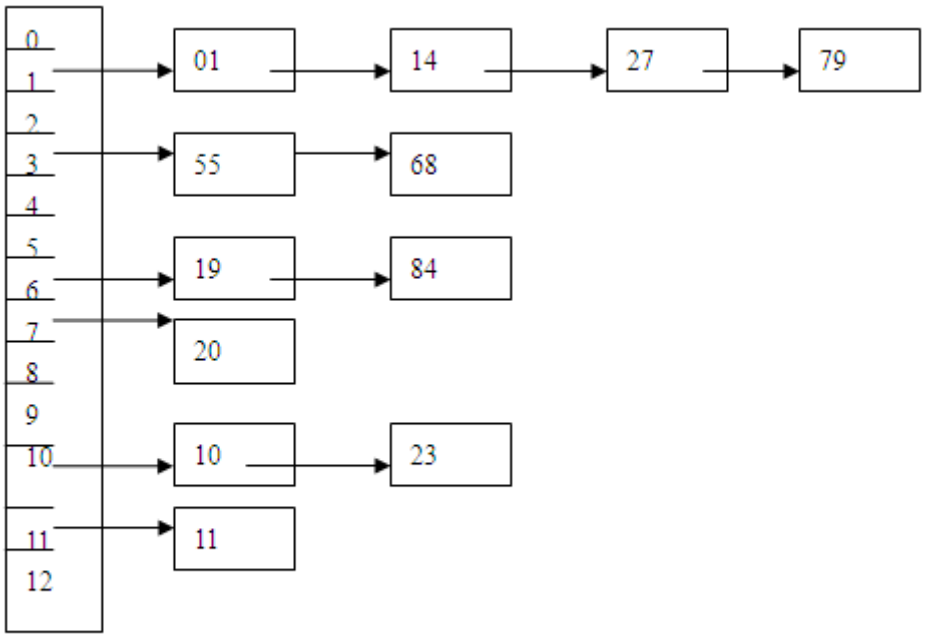
HashMap是基于哈希表的Map接口的非同步实现。实现HashMap对数据的操作，允许有一个null键，多个null值。

HashMap底层就是一个数组结构，数组中的每一项又是一个链表。数组+链表结构，新建一个HashMap的时候，就会初始化一个数组。

Entry就是数组中的元素，每个Entry其实就是一个key-value的键值对，它持有一个指向下一个元素的引用，这就构成了链表，HashMap底层将key-value当成一个整体来处理，这个整体就是一个Entry对象。

HashMap底层采用一个Entry数组来保存所有的key-value键值对，当需要存储一个Entry对象时，会根据hash算法来决定在其数组中的位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；

当需要取出一个Entry对象时，也会根据hash算法找到其在数组中的存储位置， 在根据equals方法从该位置上的链表中取出Entry;



链地址法处理冲突得到的哈希表

ConcurrentHashMap与HashMap的区别

1.HashMap

我们知道HashMap是线程不安全的，在多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

2.HashTable

HashTable和HashMap的实现原理几乎一样，差别无非是
HashTable不允许key和value为null
HashTable是线程安全的
但是HashTable线程安全的策略实现代价却太大了，简单粗暴，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁。
多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞，相当于将所有的操作串行化，在竞争激烈的并发场景中性能就会非常差。

3.ConcurrentHashMap

主要就是为了应对hashmap在并发环境下不安全而诞生的，ConcurrentHashMap的设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响。

我们都知道Map一般都是数组+链表结构（JDK1.8该为数组+红黑树）。

ConcurrentHashMap避免了对全局加锁改成了局部加锁操作，这样就极大地提高了并发环境下的操作速度，由于ConcurrentHashMap在JDK1.7和1.8中的实现非常不同，接下来我们谈谈JDK在1.7和1.8中的区别。

JDK1.7版本的CurrentHashMap的实现原理

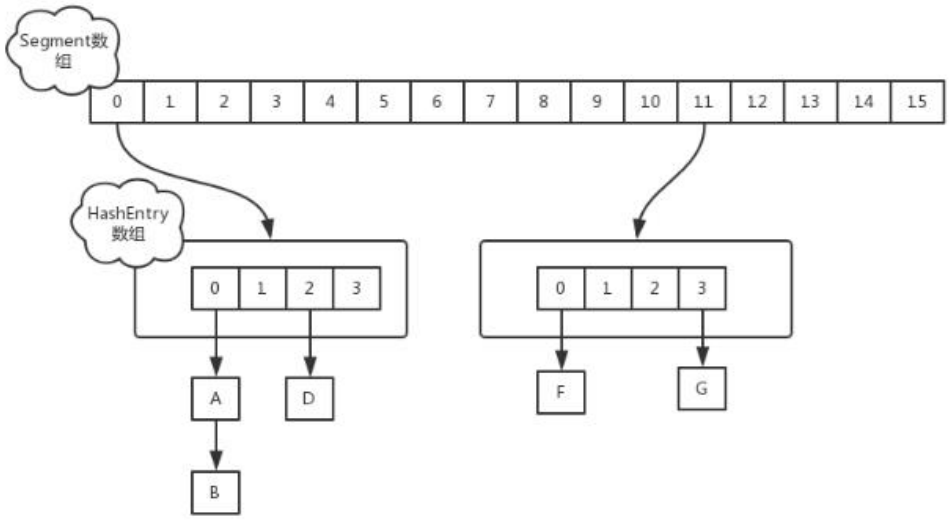
1）在JDK1.7中ConcurrentHashMap采用了数组+Segment+分段锁的方式实现。

1.Segment(分段锁)

ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表,同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

2.内部结构

ConcurrentHashMap使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。如下图是ConcurrentHashMap的内部结构图：



从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作。
第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部。

3.该结构的优劣势

坏处
这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长
好处
写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上）。
所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

JDK1.8版本的CurrentHashMap的实现原理

JDK8中ConcurrentHashMap参考了JDK8 HashMap的实现，采用了数组+链表+红黑树的实现方式来设计，内部大量采用CAS操作，这里我简要介绍下CAS。
CAS是compare and swap的缩写，即我们所说的比较交换。cas是一种基于锁的操作，而且是乐观锁。

在java中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。

而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加version来获取数据，性能较悲观锁有很大的提高。
CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和A的值是一样的，那么就将内存里面的值更新成B。

CAS是通过无限循环来获取数据的，若果在第一轮循环中，a线程获取地址里面的值被b线程修改了，那么a线程需要自旋，到下次循环才有可能机会执行。

JDK8中彻底放弃了Segment转而采用的是Node，其设计思想也不再是JDK1.7中的分段锁思想。
Node：保存key，value及key的hash值的数据结构。其中value和next都用volatile修饰，保证并发的可见性。

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;
        volatile Node<K,V> next;
```

Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。

在JDK8中ConcurrentHashMap的结构，由于引入了红黑树，使得ConcurrentHashMap的实现非常复杂，

我们都知道，红黑树是一种性能非常好的二叉查找树，其查找性能为O（logN），但是其实现过程也非常复杂，而且可读性也非常差，

DougLea的思维能力确实不是一般人能比的，早期完全采用链表结构时Map的查找时间复杂度为O（N），

JDK8中ConcurrentHashMap在链表的长度大于某个阈值的时候会将链表转换成红黑树进一步提高其查找性能。

总结

其实可以看出JDK1.8版本的ConcurrentHashMap的数据结构已经接近HashMap，相对而言，ConcurrentHashMap只是增加了同步的操作来控制并发，从JDK1.7版本的ReentrantLock+Segment+HashEntry，到JDK1.8版本中synchronized+CAS+HashEntry+红黑树。

1.数据结构：取消了Segment分段锁的数据结构，取而代之的是数组+链表+红黑树的结构。

2.保证线程安全机制：JDK1.7采用segment的分段锁机制实现线程安全，其中segment继承自ReentrantLock。JDK1.8采用CAS+Synchronized保证线程安全。3.锁的粒度：原来是对需要进行数据操作的Segment加锁，现调整为对每个数组元素加锁（Node）。4.链表转化为红黑树:定位结点的hash算法简化会带来弊端,Hash冲突加剧,因此在链表节点数量大于8时，会将链表转化为红黑树进行存储。5.查询时间复杂度：从原来的遍历链表O(n)，变成遍历红黑树O(logN)。

聊聊：HashMap的时间复杂度

HashMap容器O(1)的查找时间复杂度只是其理想的状态，而这种理想状态需要由java设计者去保证。

在由设计者保证了链表长度尽可能短的前提下，由于利用了数组结构，使得key的查找在O(1)时间内完成。

可以将 HashMap分成两部分来看待，hash和map。map只是实现了键值对的存储。而其整个O(1)的查找复杂度很大程度上是由hash来保证的。

HashMap对hash的使用体现出一些设计哲学，如：通过key.hashCode()将普通的object对象转换为int值，从而可以将其视为数组下标，利用数组O(1)的查找性能。

OK，下面我们来看看HashMap中新增元素的时间复杂度。

一：put操作的流程：

第一步：key.hashcode()，时间复杂度O(1)。

第二步：找到桶以后，判断桶里是否有元素，如果没有，直接new一个entey节点插入到数组中。时间复杂度O(1)。

第三步：如果桶里有元素，并且元素个数不超过8时，则调用equals方法，比较是否存在相同名字的key，不存在则new一个entry插入都链表尾部。时间复杂度O(1)+O(n)=O(n)。

第四步：如果桶里有元素，并且元素个数超过8时，则调用equals方法，比较是否存在相同名字的key，不存在则new一个entry插入红黑树。时间复杂度O(1)+O(logn)=O(logn)。红黑树查询的时间复杂度是logn。

通过上面的分析，我们可以得出结论，HashMap新增元素的时间复杂度是不固定的，可能的值有O(1)、O(logn)、O(n)。

二，hash碰撞问题

HashMap在put元素时，首先会计算key的hashcode，这时候不会去调用equals方法。为什么呢？因为equals方法的时间复杂度是O(n)。

但是HashMap存在hash碰撞问题，最坏的情况下，所有的key都被分配到了同一个桶，这时map的put和get时间复杂度都是O(n)。

所以HashMap的设计者必须要考虑的一个问题就是减少hash碰撞。

HashMap解决哈希冲突采用的是哪种方式呢?

答：HashMap解决哈希冲突采用的是**链地址法**。说白了就是把冲突的key连接起来，放到桶里。

当桶中的元素个数不超过8时，以单链表的形式串起来，put和get时间复杂度都是O(1)+O(n)。

当桶中的元素个数超过8个时，以红黑树的形式串起来，put和get时间复杂度都是O(1)+O(logn)。

真题：京东太猛，手写hashmap又一次重现江湖

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有小伙伴拿到了一线互联网企业如京东、极兔、有赞、希音、百度、网易的面试资格，遇到一个很重要的京东面试题：

> 手写一个hashmap?

尼恩读者反馈说，之前总是听人说，大厂喜欢手写hashmap、手写线程池，这次终于碰到了。

和线程池的知识一样，hashmap既是面试的核心知识，又是开发的核心知识。

手写线程池，之前已经通过博客、公众号的形式已经发布：

网易一面：如何设计线程池？请手写一个简单线程池？

在这里，老架构尼恩再接再厉，和架构师唐欢一块，给大家做一下手写hashmap系统化、体系化的线程池梳理，使得大家可以充分展示一下大家雄厚的“技术肌肉”，**让面试官爱到“不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的《尼恩Java面试宝典》V68版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 注：本文以 PDF 持续更新，最新尼恩 架构笔记、面试题 的PDF文件，请关注本公众号【技术自由圈】获取，暗号：领电子书

手写极简版本的HashMap

如果对HashMap理解不深，可以手写一个极简版本的HashMap，不至于颗粒无收

尼恩给大家展示，两个极简版本的首先HashMap

- 一个GoLang手写HashMap极简版本
- 一个Java手写HashMap极简版本

一个GoLang手写HashMap极简版本

设计不能少，首先，尼恩给大家做点简单的设计：

如果确实不知道怎么写， 可以使用 Wrapper 装饰器模式，把Java或者Golang内置的 HashMap包装一下，然后可以交差了。

如果是使用Go语言实现的话，具体实现方式是通过Go语言内置的map来实现，其中key和value都是int类型。

以下是一个Go语言版本 简单的手写HashMap示例：

```go
package main

import "fmt"

type HashMap struct {
    data map[int]int
}

func NewHashMap() *HashMap {
    return &HashMap{
        data: make(map[int]int),
    }
}

func (h *HashMap) Put(key, value int) {
    h.data[key] = value
}

func (h *HashMap) Get(key int) int {
    if val, ok := h.data[key]; ok {
        return val
    } else {
        return -1
    }
}

func main() {
    m := NewHashMap()
    m.Put(1, 10)
    m.Put(2, 20)
    fmt.Println(m.Get(1)) // Output: 10
    fmt.Println(m.Get(3)) // Output: -1
}
```

这个HashMap实现了Put方法将key-value对存储在map中，Get方法从map中获取指定key的value值。

为啥要先说go语言版本， Go性能高、上手快，未来几年的Java开发，理论上应该是 Java、Go 并存模式， 所以，首先来一个go语言的版本。

当然，以上的版本，太low了。

这样偷工减料，一定会被嫌弃。 只是在面试的时候，可以和面试官提一嘴， 咱们对设计模式还是很娴熟滴。

既然是手写 手写HashMap ，那么就是要从0开始，自造轮子。接下来，来一个简单版本的Java手写HashMap示例。

一个Java手写HashMap极简版本

设计不能少，首先，尼恩给大家做点简单的设计：

- 数据模型设计：

设计一个Entry数组来存储每个key-value对，其中每个Entry又是一个链表结构，用于解决hash冲突问题。

- 访问方法设计：

设计Put方法将key-value对存储在map中，Get方法从map中获取指定key的value值。

以下是一个简单的Java手写HashMap示例：

```
public class MyHashMap<K, V> {
    private Entry<K, V>[] buckets;
    private static final int INITIAL_CAPACITY = 16;

    public MyHashMap() {
        this(INITIAL_CAPACITY);
    }

    @SuppressWarnings("unchecked")
    public MyHashMap(int capacity) {
        buckets = new Entry[capacity];
    }

    public void put(K key, V value) {
        Entry<K, V> entry = new Entry<>(key, value);
        int bucketIndex = getBucketIndex(key);
        Entry<K, V> existingEntry = buckets[bucketIndex];

        if (existingEntry == null) {
            buckets[bucketIndex] = entry;
        } else {
            while (existingEntry.next != null) {
                if (existingEntry.key.equals(key)) {
                    existingEntry.value = value;
                    return;
                }
                existingEntry = existingEntry.next;
            }
            if (existingEntry.key.equals(key)) {
                existingEntry.value = value;
            } else {
                existingEntry.next = entry;
            }
        }
    }

    public V get(K key) {
        int bucketIndex = getBucketIndex(key);
        Entry<K, V> existingEntry = buckets[bucketIndex];

        while (existingEntry != null) {
            if (existingEntry.key.equals(key)) {
                return existingEntry.value;
            }
            existingEntry = existingEntry.next;
        }

        return null;
    }
```

```
50
51      private int getBucketIndex(K key) {
52          int hashCode = key.hashCode();
53          return Math.abs(hashCode) % buckets.length;
54      }
55
56      static class Entry<K, V> {
57          K key;
58          V value;
59          Entry<K, V> next;
60
61          public Entry(K key, V value) {
62              this.key = key;
63              this.value = value;
64              this.next = null;
65          }
66      }
67  }
```

咱们这个即为简单的版本，有两个特色：

- 解决hash碰撞，使用了 链地址法
- 将键转化为数组的索引的时候，使用 了 优化版本的 除留余数法

如果对这些基础知识不熟悉，可以看一下 尼恩给大家展示的基本原理。

哈希映射（哈希表）基本原理

为了一次存储便能得到所查记录，在记录的存储位置和它的关键字之间建立一个确定的对应关系H，已H（key)作为关键字为key的记录在表中的位置，这个对应关系H为哈希（Hash）函数， 按这个思路建立的表为哈希表。

哈希表也叫散列表。

从根本上来说，一个哈希表包含一个数组，通过特殊的关键码(也就是key)来访问数组中的元素。

哈希表的主要思想：
（1）存放Value的时候，通过一个哈希函数，通过关键码（key）进行哈希运算得到哈希值，然后得到映射的位置， 去寻找存放值的地方 ，

（2）读取Value的时候，也是通过同一个哈希函数，通过关键码（key）进行哈希运算得到哈希值，然后得到 映射的位置，从那个位置去读取。

哈希函数

哈希表的组成取决于哈希算法，也就是哈希函数的构成。

哈希函数计算过程会将键转化为数组的索引。

一个好的哈希函数至少具有两个特征：
（1）计算要足够快；
（2）最小化碰撞，即输出的哈希值尽可能不会重复。

那接下来我们就来看下几个常见的哈希函数：

直接定址法

- 取关键字或关键字的某个线性函数值为散列地址。
- 即 f(key) = key 或 f(key) = a*key + b，其中a和b为常数。

除留余数法

将整数散列最常用方法是除留余数法。除留余数法的算法实用得最多。

我们选择大小为m的数组，对于任意正整数k，计算k除以m的余数，即f(key)=k%m,f(key)<m。这个函数的计算非常容易（在Java中为k% M）并能够有效地将键散布在0到M-1的范围内。

数字分析法

- 当关键字的位数大于地址的位数，对关键字的各位分布进行分析，选出分布均匀的任意几位作为散列地址。
- 仅适用于所有关键字都已知的情况下，根据实际应用确定要选取的部分，尽量避免发生冲突。

平方取中法

- 先计算出关键字值的平方，然后取平方值中间几位作为散列地址。
- 随机分布的关键字，得到的散列地址也是随机分布的。

随机数法

- 选择一个随机函数，把关键字的随机函数值作为它的哈希值。
- 通常当关键字的长度不等时用这种方法。

每种数据类型都需要相应的散列函数.

例如，Interge的哈希函数就是直接获取它的值：

```
public static int hashCode(int value) {
    return value;
}
```

对于字符串类型则是使用了s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]的算法：

```
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        hash = h = isLatin1() ? StringLatin1.hashCode(value)
                              : StringUTF16.hashCode(value);
    }
    return h;
}

public static int hashCode(byte[] value) {
    int h = 0;
    for (byte v : value) {
        h = 31 * h + (v & 0xff);
    }
    return h;
}
public static int hashCode(byte[] value) {
    int h = 0;
    int length = value.length >> 1;
    for (int i = 0; i < length; i++) {
        h = 31 * h + getChar(value, i);
    }
```

```
    return h;
}
```

double类型则是使用位运算的方式进行哈希计算:

```
public int hashCode() {
    long bits = doubleToLongBits(value);
    return (int)(bits ^ (bits >>> 32));
}
public static long doubleToLongBits(double value) {
    long result = doubleToRawLongBits(value);
    if ( ((result & DoubleConsts.EXP_BIT_MASK) ==
DoubleConsts.EXP_BIT_MASK)
    &&
   (result & DoubleConsts.SIGNIF_BIT_MASK) != 0L)
        result = 0x7ff8000000000000L;
    return result;
}
```

于是Java让所有数据类型都继承了超类Object类，并实现hashCode()方法。接下来我们看下Object.hashcode方法。Object类中的hashcode方法是一个native方法。

```
public native int hashCode();
```

hashCode 方法的实现依赖于jvm，不同的jvm有不同的实现，我们看下主流的hotspot虚拟机的实现。

hotspot 定hashCode方法在src/share/vm/prims/jvm.cpp中，源码如下:

```
JVM_ENTRY(jint, JVM_IHashCode(JNIEnv* env, jobject handle))
  JVMWrapper("JVM_IHashCode");
  return handle == NULL ? 0 : ObjectSynchronizer::FastHashCode (THREAD,
JNIHandles::resolve_non_null(handle)) ;
JVM_END
```

接下来我们看下ObjectSynchronizer::FastHashCode 方法是如何返回hashcode的，ObjectSynchronizer::FastHashCode 在synchronized.hpp文件中，

```
intptr_t ObjectSynchronizer::identity_hash_value_for(Handle obj) {
  return FastHashCode (Thread::current(), obj()) ;
}

intptr_t ObjectSynchronizer::FastHashCode (Thread * Self, oop obj) {
  if (UseBiasedLocking) {

    if (obj->mark()->has_bias_pattern()) {
      // Box and unbox the raw reference just in case we cause a STW
safepoint.
      Handle hobj (Self, obj) ;
      // Relaxing assertion for bug 6320749.
      assert (Universe::verify_in_progress() ||
              !SafepointSynchronize::is_at_safepoint(),
             "biases should not be seen by VM thread here");
      BiasedLocking::revoke_and_rebias(hobj, false, JavaThread::current());
      obj = hobj() ;
```

```
      assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
    }
  }


  ObjectMonitor* monitor = NULL;
  markOop temp, test;
  intptr_t hash;
  // 获取调用hashCode() 方法的对象的对象头中的mark word
  markOop mark = ReadStableMark (obj);

  // object should remain ineligible for biased locking
  assert (!mark->has_bias_pattern(), "invariant") ;

  if (mark->is_neutral()) {  //普通对象
    hash = mark->hash();              // this is a normal header
    //如果mark word 中已经保存哈希值，那么就直接返回该哈希值
    if (hash) {                       // if it has hash, just return it
      return hash;
    }
    // 如果mark word 中还不存在哈希值，那就调用get_next_hash(Self, obj)方法计算该对象的哈希值
    hash = get_next_hash(Self, obj);  // allocate a new hash code
   // 将计算的哈希值CAS保存到对象头的mark word中对应的bit位，成功则返回，失败的话可能有几下几种情形：
   //(1)、其他线程也在install the hash并且先于当前线程成功，进入下一轮while获取哈希即可
   //（2）、有可能当前对象作为监视器升级成了轻量级锁或重量级锁，进入下一轮while走其他case；
    temp = mark->copy_set_hash(hash); // merge the hash code into header
    // use (machine word version) atomic operation to install the hash
    test = (markOop) Atomic::cmpxchg_ptr(temp, obj->mark_addr(), mark);
    if (test == mark) {
      return hash;
    }
    // If atomic operation failed, we must inflate the header
    // into heavy weight monitor. We could add more code here
    // for fast path, but it does not worth the complexity.
  } else if (mark->has_monitor()) { //重量级锁
  // 果对象是一个重量级锁monitor，那对象头中的mark word保存的是指向ObjectMonitor的指针，
  //此时对象非加锁状态下的mark word保存在ObjectMonitor中，到ObjectMonitor中去拿对象的默认哈希值：
    monitor = mark->monitor();
    temp = monitor->header();
    assert (temp->is_neutral(), "invariant") ;
    hash = temp->hash();
  //(1)如果已经有默认哈希值，则直接返回；
    if (hash) {
      return hash;
    }
    // Skip to the following code to reduce code size
  } else if (Self->is_lock_owned((address)mark->locker())) {  //轻量级锁锁
  //如果对象是轻量级锁状态并且当前线程持有锁，那就从当前线程栈中取出mark word：

    temp = mark->displaced_mark_helper(); // this is a lightweight monitor owned
```

```
    assert (temp->is_neutral(), "invariant") ;
    hash = temp->hash();                       // by current thread, check if the displaced
     //（1）如果已经有默认哈希值，则直接返回；
    if (hash) {                                // header contains hash code
      return hash;
    }

  }

  // Inflate the monitor to set hash code
  monitor = ObjectSynchronizer::inflate(Self, obj);
  // Load displaced header and check it has hash code
  mark = monitor->header();
  assert (mark->is_neutral(), "invariant") ;
  hash = mark->hash();
  //计算默认哈希值并保存到mark word中后再返回
  if (hash == 0) {
    hash = get_next_hash(Self, obj);
    temp = mark->copy_set_hash(hash); // merge hash code into header
    assert (temp->is_neutral(), "invariant") ;
    test = (markOop) Atomic::cmpxchg_ptr(temp, monitor, mark);
    if (test != mark) {

      hash = test->hash();
      assert (test->is_neutral(), "invariant") ;
      assert (hash != 0, "Trivial unexpected object/monitor header usage.");
    }
  }
  // We finally get the hash
  return hash;
}
```

关于对象头、java内置锁的内容请阅读《Java 高并发核心编程 卷2 加强版》。

ObjectSynchronizer :: FastHashCode()也是通过调用identity_hash_value_for方法返回值的，调用了get_next_hash()方法生成hash值，源码如下：

```
static inline intptr_t get_next_hash(Thread * Self, oop obj) {
  intptr_t value = 0 ;
  if (hashCode == 0) { //随机数 openjdk6、openjdk7 采用的是这种方式
     // This form uses an unguarded global Park-Miller RNG,
     // so it's possible for two threads to race and generate the same RNG.
     // On MP system we'll have lots of RW access to a global, so the
     // mechanism induces lots of coherency traffic.
     value = os::random() ;
  } else
  if (hashCode == 1) { //基于对象内存地址的函数
     // This variation has the property of being stable (idempotent)
     // between STW operations.  This can be useful in some of the 1-0
     // synchronization schemes.
     intptr_t addrBits = cast_from_oop<intptr_t>(obj) >> 3 ;
     value = addrBits ^ (addrBits >> 5) ^ GVars.stwRandom ;
  } else
  if (hashCode == 2) { //恒等于1（用于敏感性测试）
     value = 1 ;            // for sensitivity testing
```

```
  } else
  if (hashCode == 3) { //自增序列
     value = ++GVars.hcSequence ;
  } else
  if (hashCode == 4) {  //将对象的内存地址强转为int
     value = cast_from_oop<intptr_t>(obj) ;
  } else {
    //生成hash值的方式六： Marsaglia's xor-shift scheme with thread-specific
state
  //（基于线程具体状态的Marsaglias的异或移位方案） openjdk8之后采用的就是这种方式
     // Marsaglia's xor-shift scheme with thread-specific state
     // This is probably the best overall implementation -- we'll
     // likely make this the default in future releases.
     unsigned t = Self->_hashStateX ;
     t ^= (t << 11) ;
     Self->_hashStateX = Self->_hashStateY ;
     Self->_hashStateY = Self->_hashStateZ ;
     Self->_hashStateZ = Self->_hashStateW ;
     unsigned v = Self->_hashStateW ;
     v = (v ^ (v >> 19)) ^ (t ^ (t >> 8)) ;
     Self->_hashStateW = v ;
     value = v ;
  }

  value &= markOopDesc::hash_mask;
  if (value == 0) value = 0xBAD ;
  assert (value != markOopDesc::no_hash, "invariant") ;
  TEVENT (hashCode: GENERATE) ;
  return value;
}

```

到底用的哪一种计算方式，和参数hashCode有关系，在src/share/vm/runtime/globals.hpp中配置了默认：
openjdk6：

```
 product(intx, hashCode, 0,                                          \
 "(Unstable) select hashCode generation algorithm")                   \

```

openkjdk8:

```
 product(intx, hashCode, 5,                                          \
 "(Unstable) select hashCode generation algorithm")                   \

```

也可以通过虚拟机启动参数-XX\:hashCode=n来做修改。

到这里你知道hash值是如何生成的了吧。

哈希表因为其本身结构使得查找对应的值变得方便快捷，但是也带来了一些问题，问题就是无论使用哪种方式生成hash值，总有产生相同值的时候。接下来我们就来看下如何解决hash值相同的问题。

hash 碰撞（哈希冲突）

对于两个不同的数据元素通过相同哈希函数计算出来相同的哈希地址（即两不同元素通过哈希函数取模得到了同样的模值），这种现象称为哈希冲突或哈希碰撞。

一般来说，哈希冲突是无法避免的。如果要完全避免的话，那么就只能一个字典对应一个值的地址，这样一来， 空间就会增大，甚至内存溢出。减少哈希冲突的原因是Hash碰撞的概率就越小，map的存取效率就会越高。 常见的哈希冲突的解决方法有开放地址法和链地址法：

开放地址法

开放地址法又叫开放寻址法、开放定址法，当冲突发生时，使用某种探测算法在散列表中寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到。开放地址法需要的表长度要大于等于所需要存放的元素。

按照探测序列的方法，可以细分为线性探查法、平法探查法、双哈希函数探查法等。

这里为了更好的展示三种方法的效果，我们用例子来看看：设关键词序列为{47,7,29,11,9,84,54,20,30}，哈希表长度为13，装载因子=9/13=0.69，哈希函数为f(key)=key%p=key%11

关键词（key）	**47**	**7**	**29**	**11**	**9**	**84**	**54**	**20**	**30**
散列地址k（key）	3	7	7	0	9	7	10	9	8

（1）线性探测法

当我们的所需要存放值的位置被占了，我们就往后面一直加1并对m取模直到存在一个空余的地址供我们存放值，取模是为了保证找到的位置在0~m-1的有效空间之中。

公式：fi=(f(key)+i) % m ， 0 ≤ i ≤ m-1i会逐渐递增加1）

具体做法： 探查时从地址d开始，首先探查T[d]，然后依次探查T[d+1]....直到T[m-1]，然后又循环到T[0]、T[1],...直到探查到有空余的地址或者直到T[d-1]为止。

用线性探测法处理冲突得到的哈希表如下

地址 / 操作	0	1	2	3	4	5	6	7	8	9	10	11	12	说明
插入47				47										无冲突
插入7				47				7						无冲突
插入29				47				7	29					d1=1
插入11	11			47				7	29					无冲突
插入9	11			47				7	29	9				无冲突
插入84	11			47				7	29	9	84			d3=3
插入54	11			47				7	29	9	84	54		d1=2
插入20	11			47				7	29	9	84	54	20	d3=3
插入30	11	30		47				7	29	9	84	54	20	d6=6

缺点：需要不断处理冲突，无论是存入还是查找效率都会大大降低。

（2）平方探查法

当我们的所需要存放值的位置被占了，会前后寻找而不是单独方向的寻找。

公式：fi=(f(key)+di) % m，0 ≤ i ≤ m-1

具体操作：探查时从地址 d 开始，首先探查 T[d]，然后依次探查 T[d+di]，di 为增量序列1^2、-1^2，2^2、-2^2， ……，q^2、$-q^2$ 且q≤1/2 (m-1) ,直到探查到 有空余地址或者到 T[d-1]为止。

用平方探查法处理冲突得到的哈希表如下

操作＼地址	0	1	2	3	4	5	6	7	8	9	10	说明
插入47				47								无冲突
插入7				47				7				无冲突
插入29				47				7	29			d1=1
插入11	11			47				7	29			无冲突
插入9	11			47				7	29	9		无冲突
插入84	11			47			84	7	29	9		d3=3
插入54	11			47			84	7	29	9	54	d1=2
插入20	11		20	47			84	7	29	9	54	d3=3
插入30	11	30	20	47			84	7	29	9	54	d6=6

（3）双哈希函数探查法

公式：fi=(f(key)+i*g(key)) % m (i=1，2，……，m-1)

其中f（key） 和g（key） 是两个不同的哈希函数，m为哈希表的长度。

具体步骤：

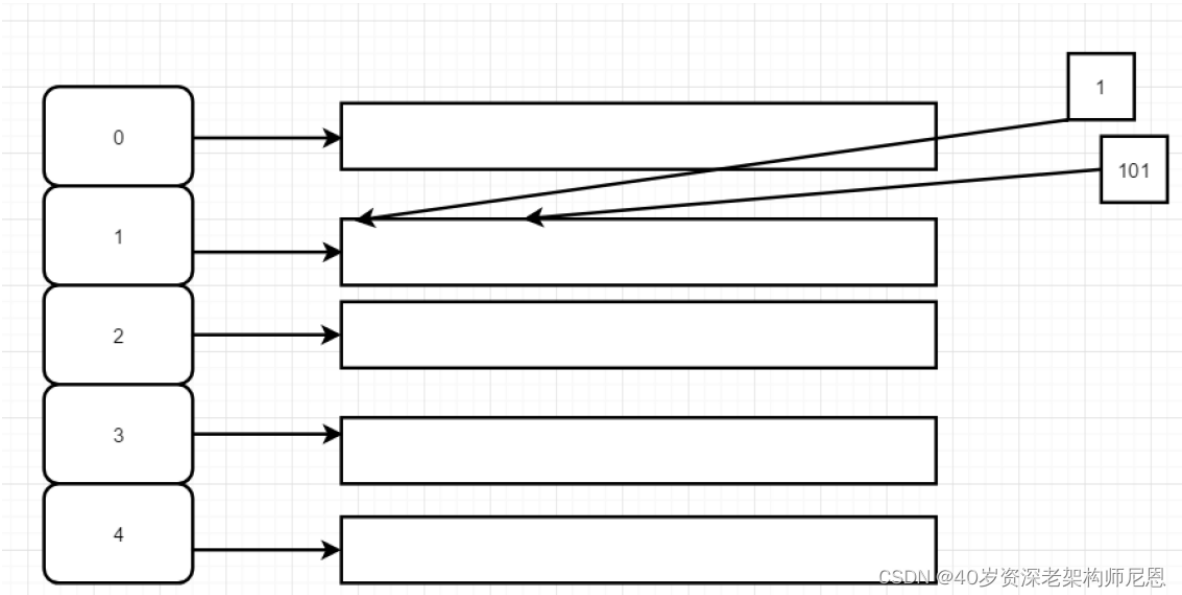
双哈希函数探测法，先用第一个函数f(key)对关键码计算哈希地址，一旦产生地址冲突，再用第二个函数 g(key)确定移动的步长因子，最后通过步长因子序列由探测函数寻找空的哈希地址。

开发地址法，通过持续的探测，最终找到空的位置。为了解决这个问题，引入了链地址法。

链地址法

在哈希表每一个单元中设置链表，某个数据项对的关键字还是像通常一样映射到哈希表的单元中，而数据项本身插入到单元的链表中.

链地址法简单理解如下：

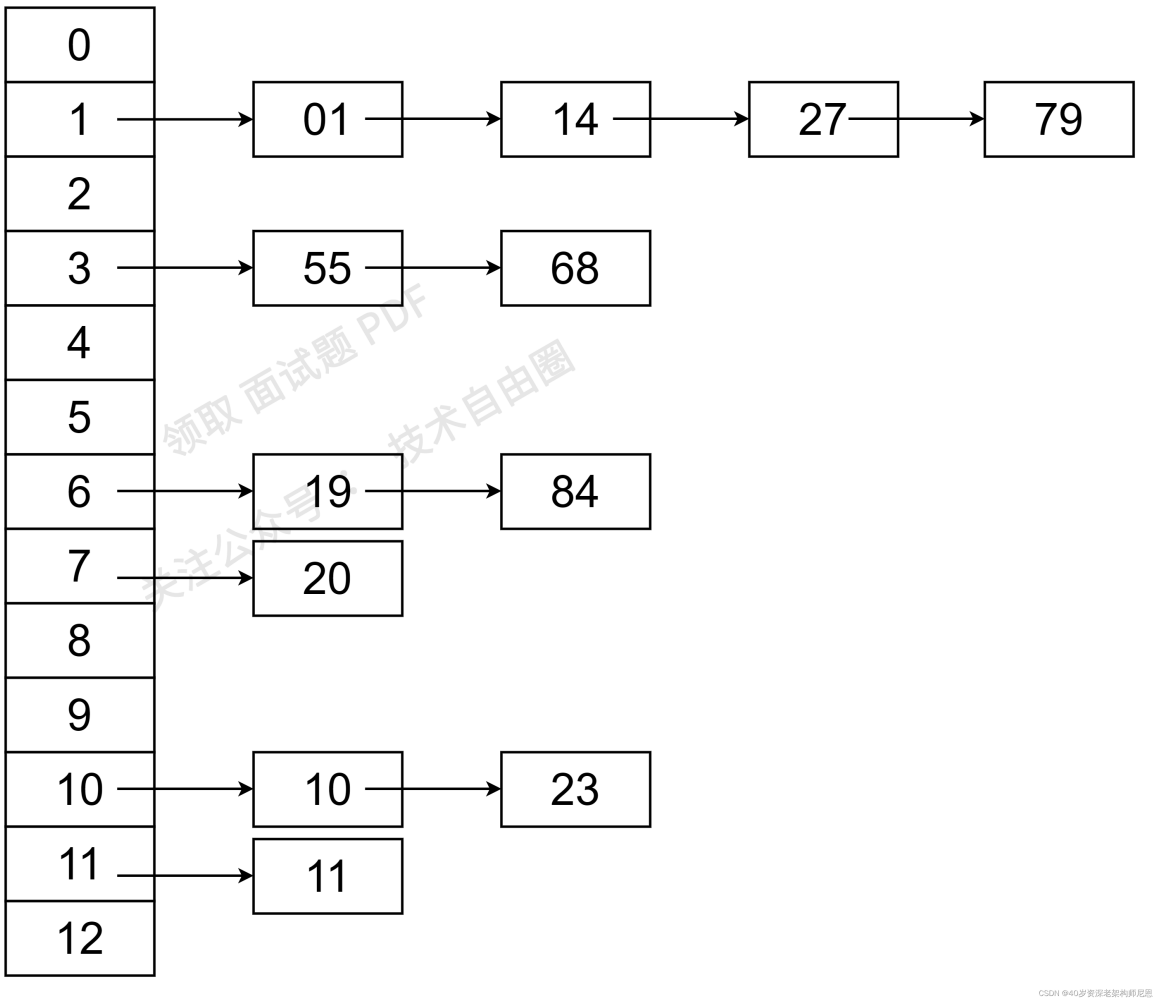


来一个相同的数据，就将它插入到单元对应的链表中，在来一个相同的，继续给链表中插入。

链地址法解决哈希冲突的例子如下：

（1）采用除留余数法构造哈希函数，而 冲突解决的方法为 链地址法。

（2）具体的关键字列表为（19,14,23,01,68,20,84,27,55,11,10,79），则哈希函数为f(key)=key MOD 13。则采用除留余数法和链地址法后得到的预想结果应该为：



哈希造表完成后，进行查找时，首先是根据哈希函数找到关键字的位置链，然后在该链中进行搜索，如果存在和关键字值相同的值，则查找成功，否则若到链表尾部仍未找到，则该关键字不存在。

哈希表作为一个非常常用的查找数据结构，它能够在O(1)的时间复杂度下进行数据查找,时间主要花在计算hash值上。在Java中，典型的Hash数据结构的类是HashMap。

然而也有一些极端的情况，最坏的就是hash值全都映射在同一个地址上，这样哈希表就会退化成链表，例如下面的图片：

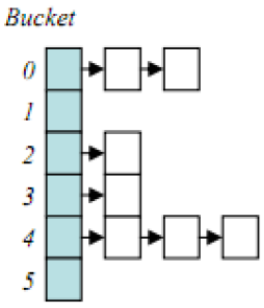


Figure 1: Normal operation of a hash table.

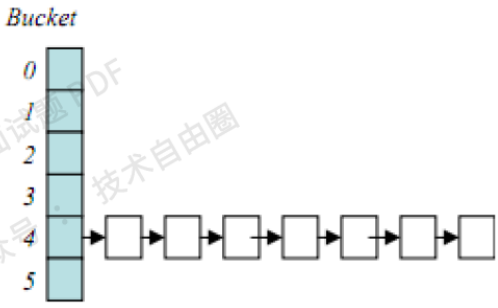


Figure 2: Worst-case hash table collisions.

当hash表变成图2的情况时，查找元素的时间复杂度会变为O(n)，效率瞬间低下，

所以，设计一个好的哈希表尤其重要，如HashMap在jdk1.8后引入的红黑树结构就很好的解决了这种情况。

手写一个相对复杂的HashMap

要拿高分，写个极简的版本，是不够的。

接下来模拟JDK的HashMap，我们就自己来手写hashMap。

复杂的HashMap的数据模型设计+接口设计

设计不能少，首先，尼恩给大家做点简单的设计：

- 数据模型设计：

宏观上来说：数组+ 链表

设计一个Table 数组来存储每个key-value对，一个key-value对封装为一个Node，其中每个Node可以增加指针，指向后继节点，可以形成一个链表结构，用于解决hash冲突问题。

- 访问方法设计：

设计一组方法，进行原始的 put、get。

既然接下来模拟JDK的HashMap，知己方能知彼，首先来看看，一个JDK 1.8版本ConcurrentHashMap实例的内部结构示例如图7-16所示。

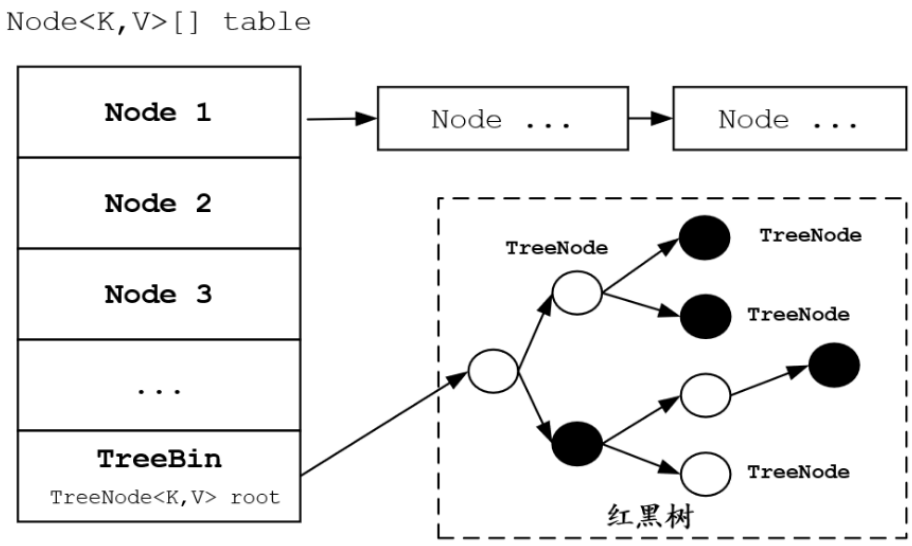


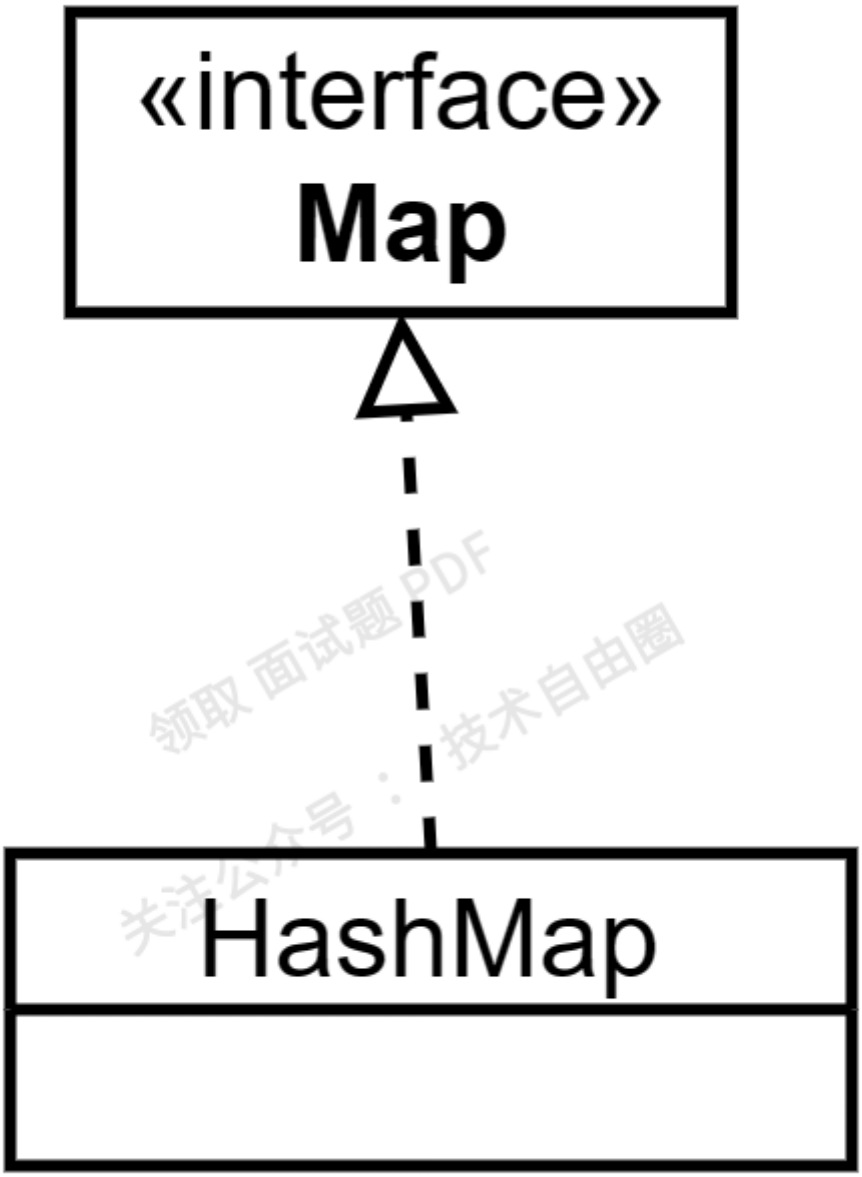
图7-16 一个JDK 1.8 版本ConcurrentHashMap实例的内部结构

以上的内容，来自 尼恩的 《Java 高并发核心编程 卷2 加强版》，尼恩的高并发三部曲，很多小伙伴反馈说：相见恨晚，爱不释手。

接下来，开始定义顶层的访问接口

定义顶层的访问接口

首先，我们需要的是确定HashMap结构，那么咱们就定义一个Map接口和一个Map实现类HashMap，其结构如下：



Map接口的实现

在Map接口中定义了以下几个方法

```
public interface Map<K,V> {
    int size();

    boolean isEmpty();

    void clear();

    V put(K key,V value);

```

```
    V remove(K key);

    V get(K key);

    boolean containsKey(K key);

    boolean containsValue(V value);

    boolean equals(Object o);

    int hashCode();

    interface Entry<K, V> {
        K getKey();
        V getValue();
        V setValue(V value);
        int hashCode();
        boolean equals(Object o);
    }
}

```

手写实现类HashMap

定义好Map接口后，那么接下来我们就需要实现Map接口，定义实现类为HashMap。

HashMap类如下：

```
public class HashMap<K, V> implements Map<K, V> {
    //数组默认初始容4
    private int DEFAULT_CAPACITY = 1 << 2;
    // 加载因子
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
     //数组扩容的阈值= loadFactorx 容量(capacity)
     int threshold;

    public HashMap() {
        threshold = (int) (DEFAULT_CAPACITY * DEFAULT_LOAD_FACTOR);
    }
    ......
}

```

HashMap类的构造函数中，仅对数组扩容阈值做了默认设置， 默认的数组扩容阈值等于数组默认容量*负载因子（0.75）

Node<K,V>节点的实现

在hashMap中定义数组集合节点Node<K,V>，Node<K,V>节点实现了Map中的Entry接口类，

在Node节点中定义了hash值，Key值，Value值和指针，

其核心代码如下：

```
/**
* 链表结点
*
```

```
 * @param <K>
 * @param <V>
 */
static class Node<K, V> implements Map.Entry<K, V> {
    int hash;
    K key;
    V value;
    Node<K, V> next;


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

......
}

```

定义好Node<K,V>节点后，我们需要定义一个**数组**，来存储Key-Value键值对；

数组定义如下:

```
// 数组
Node<K, V>[] table;

```

hash()函数实现

使用table中存储key-value键值对的前提是获取到table的下标值，在这我们采用最常用的hash函数-除留余数法f(key) =key%m 获取散列地址作为数组table的下标值,hash()函数实现如下：

```
private int hash(Object key) {
    if (key == null) return 0;
    int hash = (Integer) key % 4;
    return hash;
}

```

开放地址法解决hash碰撞

通过hash()函数计算出散列地址作为数组下标后，那么我们就可以实现Key-Value键值对的存储。

HashMap的构造函数中仅设置了数组扩容的阈值，但是并没对数组进行初始化，那么就需要在第一次保存Key-Value值时进行数组table的初始化。

hash表最常见的问题就是hash碰撞，hash碰撞的解决方法有两种，开放地址法和链地址法，

我们先用最简单的开放地址法来解决hash冲突，那么保存Key-Value键值对的具体实现如下：

```
/**
 * 插入节点
 *
 * @param key   key值

```

```
 * @param value value值
 * @return
 */
@Override
public V put(K key, V value) {
    //通过key计算hash值
    int hash = hash(key);

    //数组
    Node<K, V>[] tab;
    // 数组长度
    int n;

    // 数组的位置,即hash槽位
    int i;

    //根据数组长度和哈子自来寻址
    Node<K, V> parent;

    if ((tab = table) == null || (n = tab.length) == 0) {
        //第一次put的时候,调用ensureCapacity初始化数组table
        tab = ensureCapacity();
        n = tab.length;
    }

    // 开始时插入元素
    if ((parent = tab[i = hash]) == null) { //无hash碰撞，在当前下标位置直接插入
        System.out.println("下标:" + i + ",数组插入的key:" + key + ",value:" + value);
        //如果没有hash碰撞,就直接插入数组中
        tab[i] = new Node<>(hash, key, value, null);
        ++size;
    } else {  // 有hash碰撞的时候，就采用线性探查法解决hash碰撞：fi=（f(key)+i)%4

        if (i == (n - 1)) {
            //若已是下标最大值，就从头开始查找空位置插入
            for (int j = 0; j < i; j++) {
                if (tab[j] == null) {
                    System.out.println("已最后一个下标，从0下标开始找,下标为：" + j + ",数组插入的key:" + key + ",value:" + value);
                    tab[j] = new Node<>(hash, key, value, null);
                    ++size;
                    break;
                }
            }
        } else { // 若不是下标最大值，那就从当前下标往后查找空位置插入

            for (int index = 1; index < n - i - 1; index++) {
                //先往后查找，若往后查找有空位，就直接插入，
                if (tab[i + index] == null) {
                    System.out.println("从当前下标往后找,下标为：" + (i + index) + ",数组插入的key:" + key + ",value:" + value);
                    tab[i + index] = new Node<>(hash, key, value, null);
                    ++size;
                    break;
                }
            }
        }
```

```
    }

    // 判断当前数组是否需要扩容
    if (size > threshold ){
        //扩容操作
        ensureCapacity();
    }
    return value;
}

```

在第一次调用put()方法保存Key-Value键值对的时候，调用ensureCapacity()方法初始化数组。

在保存Key-Value键值对后需要判断是否需要扩容，扩容的条件是当前数组中元素个数超过阈值就需要扩容。

调用ensureCapacity()方法进行扩容操作，每次新容量=1.5 * 数组原容量；

具体代码实现如下：

```
/**
 * 数组扩容
 */
private Node<K, V>[] ensureCapacity() {
    int oldCapacity = 0;
    //数组未初始化，对数组进行初始化
    if (table == null || table.length == 0) {
        table = new Node[DEFAULT_CAPACITY];
        return table;
    }

    // 数组已初始化，旧容量
    oldCapacity = table.length;
    // 扩容后新的数组容量
    int newCapacity = 0;
    // 如果数组的长度 == 容量
    if (size > threshold) {
        // 新容量为旧容量的1.5倍
        newCapacity = oldCapacity + (oldCapacity >> 1);

        //数组扩容阈值= 新容量*负载因子（0.75）
        threshold = (int) (newCapacity * DEFAULT_LOAD_FACTOR);
        //创建一个新数组
        Node<K, V>[] newTable = new Node[newCapacity];
        // 把原来数组中的元素放到新数组中
        for (int i = 0; i < size; i++) {
            newTable[i] = table[i];
        }
        table = newTable;
        System.out.println(oldCapacity + "扩容为" + newCapacity);
    }
    return table;
}

```

从上述代码可看到，在原数组容量超过阈值的时候，就会进行扩容操作，扩容成功后还需要做以下几件事：

- 重新设置数组扩容的阈值， 这个时候扩容阈值= 数组新容量* 负载因子；
- 把旧数组的元素赋值到新数组中， 新数组的元素存放位置按照旧数组的位置进行存储，这一个步骤是最影响性能的。
- 返回新创建的数组。

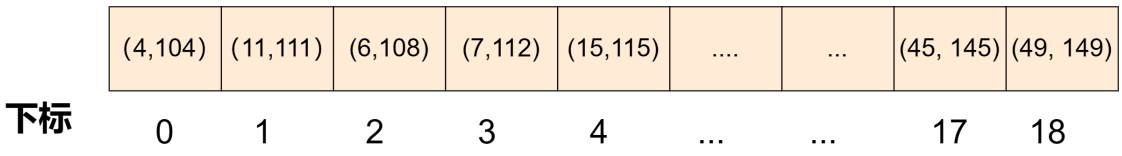
HashMap存储Key-Value键值对到此就完成了，我们来写一个测试单元来看下执行效果，测试单元代码如下：

```
@Test
public void hashMapTest() {
    HashMap<Integer, Integer> hashMap = new HashMap<>();
    hashMap.put(4, 104);
    hashMap.put(6, 108);
    hashMap.put(7, 112);
    hashMap.put(11, 111);
    hashMap.put(15, 115);
    hashMap.put(19, 119);
    hashMap.put(1, 100);
    hashMap.put(5, 105);
    hashMap.put(9, 109);
    hashMap.put(29, 129);
    hashMap.put(13, 113);
    hashMap.put(17, 117);
    hashMap.put(21, 121);
    hashMap.put(25, 125);

    hashMap.put(33, 133);
    hashMap.put(37, 137);
    hashMap.put(41, 141);
    hashMap.put(45, 145);
    hashMap.put(49, 149);
}

```

执行结果如下：

```
Tests passed: 1 of 1 test – 2 ms
D:\Java\jdk-11.0.11\bin\java.exe ...
下标:0,数组插入的key:4,value:104
下标:2,数组插入的key:6,value:108
下标:3,数组插入的key:7,value:112
已最后一个下标，从0下标开始找,下标为：1,数组插入的key:11,value:111
4扩容为6
从当前下标往后找,下标为：4,数组插入的key:15,value:115
6扩容为9
从当前下标往后找,下标为：5,数组插入的key:19,value:119
从当前下标往后找,下标为：6,数组插入的key:1,value:100
9扩容为13
从当前下标往后找,下标为：7,数组插入的key:5,value:105
从当前下标往后找,下标为：8,数组插入的key:9,value:109
从当前下标往后找,下标为：9,数组插入的key:29,value:129
13扩容为19
从当前下标往后找,下标为：10,数组插入的key:13,value:113
从当前下标往后找,下标为：11,数组插入的key:17,value:117
从当前下标往后找,下标为：12,数组插入的key:21,value:121
从当前下标往后找,下标为：13,数组插入的key:25,value:125
从当前下标往后找,下标为：14,数组插入的key:33,value:133
19扩容为28
从当前下标往后找,下标为：15,数组插入的key:37,value:137
从当前下标往后找,下标为：16,数组插入的key:41,value:141
从当前下标往后找,下标为：17,数组插入的key:45,value:145
从当前下标往后找,下标为：18,数组插入的key:49,value:149
```


存储结构如下图所示：



Key-Value 键值对已经保存到数组中了，那接下来我们就来探索下在HashMap中如何通过Key值某个Value值。

主要是通过for循环遍历查找，如果hash值相同或者Key值相同就说明找到Key-Value键值对，然后返回对应的value值，具体实现如下：

```
@Override
public V get(Object key) {
    Node<K, V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
/**
 * 通过key值在数组中查找value值
 *
 * @param hash
 * @param key
 * @return
 */
private Node<K, V> getNode(int hash, Object key) {
    K k;

    //如果不是就for循环查找
    for (int i = 0; i < table.length; i++) {
        if (table[i].hash == hash && ((k = table[i].key) == key || (key != null && key.equals(k)))) {
            return table[i];
        }
    }
    return null;
}

```

用测试单元来查看Key = 19 看返回的值是否正确，测试单元如下：

```
@Test
public void hashMapTest01() {
    HashMap<Integer, Integer> hashMap = new HashMap<>();
    hashMap.put(4, 104);
    hashMap.put(6, 108);
    hashMap.put(7, 112);
    hashMap.put(11, 111);
    hashMap.put(15, 115);
    hashMap.put(19, 119);

    System.out.println("hashMap get() value:" + hashMap.get(19));
}

```

执行结果如下：

```
hashMap get() value:119
```

从上述的HashMap 的put()方法采用的开发地址法持续探测最终找到空的位置保存Key-Value键值对，在get()方法中也是通过循环不断的探测hash值或Key值。这种方式在记录总数可以预知的情况下，可以创建完美的hash表，这种情况下存储效率是很高的。

但是在实际应用中，往往记录的数据量是不确定的，那么存储的数组元素超过阈值的时候就需要进行扩容操作，扩容操作的时间成本是很高的，频繁的扩容操作同样也会程序的性能。 采用开放地址法是通过不断的探测寻找空地址，探测的过程的时间成本也是很高的，而且在查找key-value键值对时，就不能单纯的使用数组下标的方式获取，而是通过循环的方式进行查找，这个过程也是十分消耗时间的。

针对hash表的开放地址法存在的问题，我们引入链地址法来解决， jdk1.7以及之前的HashMap就是采用的数组+链表的方式进行解决的。

链地址法解决hash碰撞

首先，我们对存储Key-Value键值对的put方法进行优化，优化的内容就是把有hash碰撞的Key-Value键值对用链表的形式进行存储，采用尾插入的方式往链表中插入有hash碰撞的Key-Value键值对，具体实现如下：

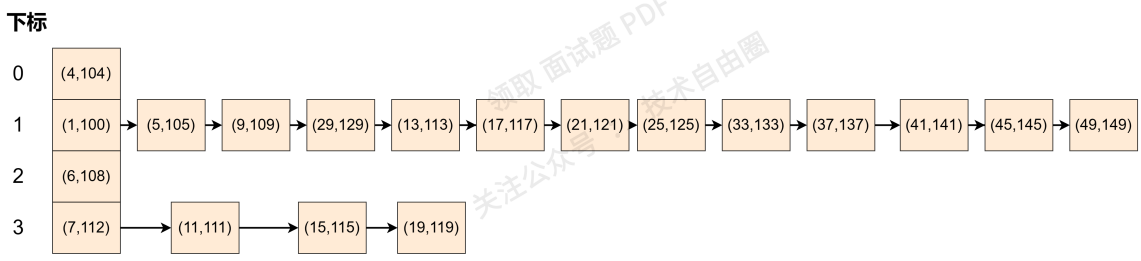
```
/**
 * 插入节点
 *
 * @param key   key值
 * @param value value值
 * @return
 */
@Override
public V put(K key, V value) {
   ...
    // 开始时插入元素
    if ((parent = tab[i = hash]) == null) {
        System.out.println("下标为: "+i+"数组插入的key:" + key + ",value:" + value);
        //如果没有hash碰撞,就直接插入数组中
        tab[i] = new Node<>(hash, key, value, null);
        ++size;
    } else { //有哈希碰撞时,采用链表存储
        // 下一个子结点
        Node<K, V> next;
        K k;
        System.out.println("下标为: "+i+"有哈希碰撞的key:" + key + ",value:" + value);
        if (parent.hash == hash
                && ((k = parent.key) == key || (key != null && key.equals(k)))) {
            // 哈希碰撞,且节点已存在,直接替换数组元素
            next = parent;
        } else {
            System.out.println("下标为: "+i+"链表插入的key:" + key + ",value:" + value);
            // 哈希碰撞, 链表插入
            for (int linkSize = 0; ; ++linkSize) {
                //
System.out.println("linkSize="+linkSize+",node:"+parent);
                //如果当前结点的下一个结点为null,就直接插入
                if ((next = parent.next) == null) {
                    System.out.println("new链表长度为:" + linkSize);
                    parent.next = new Node<>(hash, key, value, null);
                    break;
                }
                if (next.hash == hash
```

```
38                          && ((k = next.key) == key || (key != null &&
key.equals(k)))) {
39                      //如果节点已经存在,直接跳出for循环
40                      break;
41                  }
42                  parent = next;
43              }
44              printLinked(hash);
45          }
46      }
47      ...
48  }
```

执行测试单元结果如下：

```
D:\Java\jdk-11.0.11\bin\java.exe ...
下标为：0数组插入的key:4,value:104
下标为：2数组插入的key:6,value:108
下标为：3数组插入的key:7,value:112
下标为：3有哈希碰撞的key:11,value:111
下标为：3链表插入的key:11,value:111
new链表长度为:0
 [->(7,112)->(11,111)],链表长度：2
下标为：3有哈希碰撞的key:15,value:115
下标为：3链表插入的key:15,value:115
new链表长度为:1
 [->(7,112)->(11,111)->(15,115)],链表长度：3
下标为：3有哈希碰撞的key:19,value:119
下标为：3链表插入的key:19,value:119
new链表长度为:2
 [->(7,112)->(11,111)->(15,115)->(19,119)],链表长度：4
下标为：1数组插入的key:1,value:100
下标为：1有哈希碰撞的key:5,value:105
下标为：1链表插入的key:5,value:105
new链表长度为:0
 [->(1,100)->(5,105)],链表长度：2
下标为：1有哈希碰撞的key:9,value:109
下标为：1链表插入的key:9,value:109
new链表长度为:1
 [->(1,100)->(5,105)->(9,109)],链表长度：3
下标为：1有哈希碰撞的key:29,value:129
下标为：1链表插入的key:29,value:129
new链表长度为:2
 [->(1,100)->(5,105)->(9,109)->(29,129)],链表长度：4
下标为：1有哈希碰撞的key:13,value:113
下标为：1链表插入的key:13,value:113
new链表长度为:3
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)],链表长度：5
下标为：1有哈希碰撞的key:17,value:117
下标为：1链表插入的key:17,value:117
new链表长度为:4
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)],链表长度：6
下标为：1有哈希碰撞的key:21,value:121
下标为：1链表插入的key:21,value:121
new链表长度为:5
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)],链表长度：7
下标为：1有哈希碰撞的key:25,value:125
下标为：1链表插入的key:25,value:125
new链表长度为:6
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)],链表长度：8
下标为：1有哈希碰撞的key:33,value:133
下标为：1链表插入的key:33,value:133
new链表长度为:7
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)->(33,133)],链表长度：9
下标为：1有哈希碰撞的key:37,value:137
下标为：1链表插入的key:37,value:137
new链表长度为:8
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)->(33,133)->(37,137)],链表长度：10
下标为：1有哈希碰撞的key:41,value:141
下标为：1链表插入的key:41,value:141
new链表长度为:9
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)->(33,133)->(37,137)->(41,141)],链表长度：11
下标为：1有哈希碰撞的key:45,value:145
下标为：1链表插入的key:45,value:145
new链表长度为:10
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)->(33,133)->(37,137)->(41,141)->(45,145)],链表长度：12
下标为：1有哈希碰撞的key:49,value:149
下标为：1链表插入的key:49,value:149
new链表长度为:11
 [->(1,100)->(5,105)->(9,109)->(29,129)->(13,113)->(17,117)->(21,121)->(25,125)->(33,133)->(37,137)->(41,141)->(45,145)->(49,149)],链表长度：13

Process finished with exit code 0
```


存储的结构如下图所示：



保存有hash碰撞的Key-Value键值对时采用了链表形式，那么在调用get()方法查找的时候，

首先通过hash()函数计算出数组的下标索引值，然后通过下标索引值查找数组对应的Node<K,V>节点，通过key值和hash值判断第一个结点是否是查找的Key-Value键值对，

若不是第一个结点不是要查找的Key-Value键值对，就从头开始变量链表进行Key-Value键值查找，查找到了就返回Key-Value键值对，没有查找到就返回null，

使用链地址法优化后的get()方法实现代码如下：

```
@Override
public V get(Object key) {
    Node<K, V> e;

    return (e = getNode(hash(key), key)) == null ? null : e.value;
}
/**
     * 通过key值在数组/链表/红黑树中查找value值
     *
     * @param hash
     * @param key
     * @return
     */
private Node<K, V> getNode(int hash, Object key) {
    //数组
    Node<K, V>[] tab;

    //数组长度
    int n;
    // (n-1)$hash 获取该key对应的数据节点的hash槽位,即链表的根结点
    Node<K, V> parent;

    //root的子节点
    Node<K, V> next;

    K k;

    //如果数组为空,并且长度为空, hash槽位对应的节点为空,就返回null
    if ((tab = table) != null && (n = table.length) > 0
        && (parent = tab[ hash]) != null) {
        // 如果计算出来的hash槽位所对应的结点hash值等于hash值,结点的key=查找key值,
        // 返回hash槽位对应的结点,即数组
        if (parent.hash == hash && ((k = parent.key) == key || (key != null
&& key.equals(k)))) {
            return parent;
        }
        //如果不在根结点,在子结点
        if ((next = parent.next) != null) {

            //在链表中查找,需要通过循环一个个往下查找
            while (next != null) {
                if (next.hash == hash && ((k = next.key) == key || (key !=
null && key.equals(k)))) {
                    return next;
                }
                next = next.next;
            }

        }

```

```
    }
    return null;
}
```

采用链地址法解决hash碰撞问题相比开放地址法来说，处理冲突简单且无堆积现象， 发生hash碰撞后不用探测空位置保存元素，数组table也不需要频繁的进行扩容操作。

而且链表地址法中链表采用的时候尾插入方式增加节点，不会出现环问题，而且链表的节点插入效率比较高；链表上的节点空间是动态申请的，它更适合需要保存的Key-Value键值对个数不确定的情况，节省了空间也提高了插入效率。

但是链表不支持随机访问，查找元素效率比较低，需要遍历结点，所以当链表长度过长的时候，查找元素效率就会比较低，那么在链表长度超过一定阈值的时候，我们可以把链表转换成红黑树来提升查询的效率。

红黑树提升查询效率

采用红黑树来提升查询效率，首先需要定义红黑树的节点，该节点继承了Node节点，同时新增了左右结点和父节点。代码如下：

```
/**
 * 红黑树结点
 *
 * @param <K>
 * @param <V>
 */
static final class RBTreeNode<K, V> extends Node<K, V> {
    boolean color = RED;
    // 左节点
    RBTreeNode<K, V> left;
    // 右节点
    RBTreeNode<K, V> right;
    // 父节点
    RBTreeNode<K, V> parent;


public RBTreeNode(int hash, K key, V value, Node<K, V> next) {
    super(hash, key, value, next);
}


public boolean hasTwoChildren() {
    return left != null && right != null;
}

/**
 * 是否为左结点
 *
 * @return
 */
public boolean isLeftChild() {
    return parent != null && this == parent.left;
}


/**
 * 判断是否为右子树
```

```
     *
     * @return
     */
    public boolean isRightChild() {
        return parent != null && this == parent.right;
    }

    /**
     * 获取兄弟结点
     *
     * @return
     */
    public RBTreeNode<K, V> sibling() {
        if (isLeftChild()) {
            return parent.right;
        }

        if (isRightChild()) {
            return parent.left;
        }

        return null;
    }

    ....
    }
```

接下来我们来优化一下Key-Value键值存储的put()方法，优化的点主要是Hash碰撞后的处理，具体如下：

- 通过hash()函数获得table下标索引值后，若该结点已是红黑树节点，就把需保存的Key-Value()节点插入到红黑树中，并判断是否平衡，若不平衡则进行自平衡操作；
- 通过hash()函数获得table下标索引值的节点是链表节点，则采用尾插入的方式插入链表结点，插入完成后判断链表长度是否超过链表长度阈值，若超过阈值就把链表转换成红黑树。

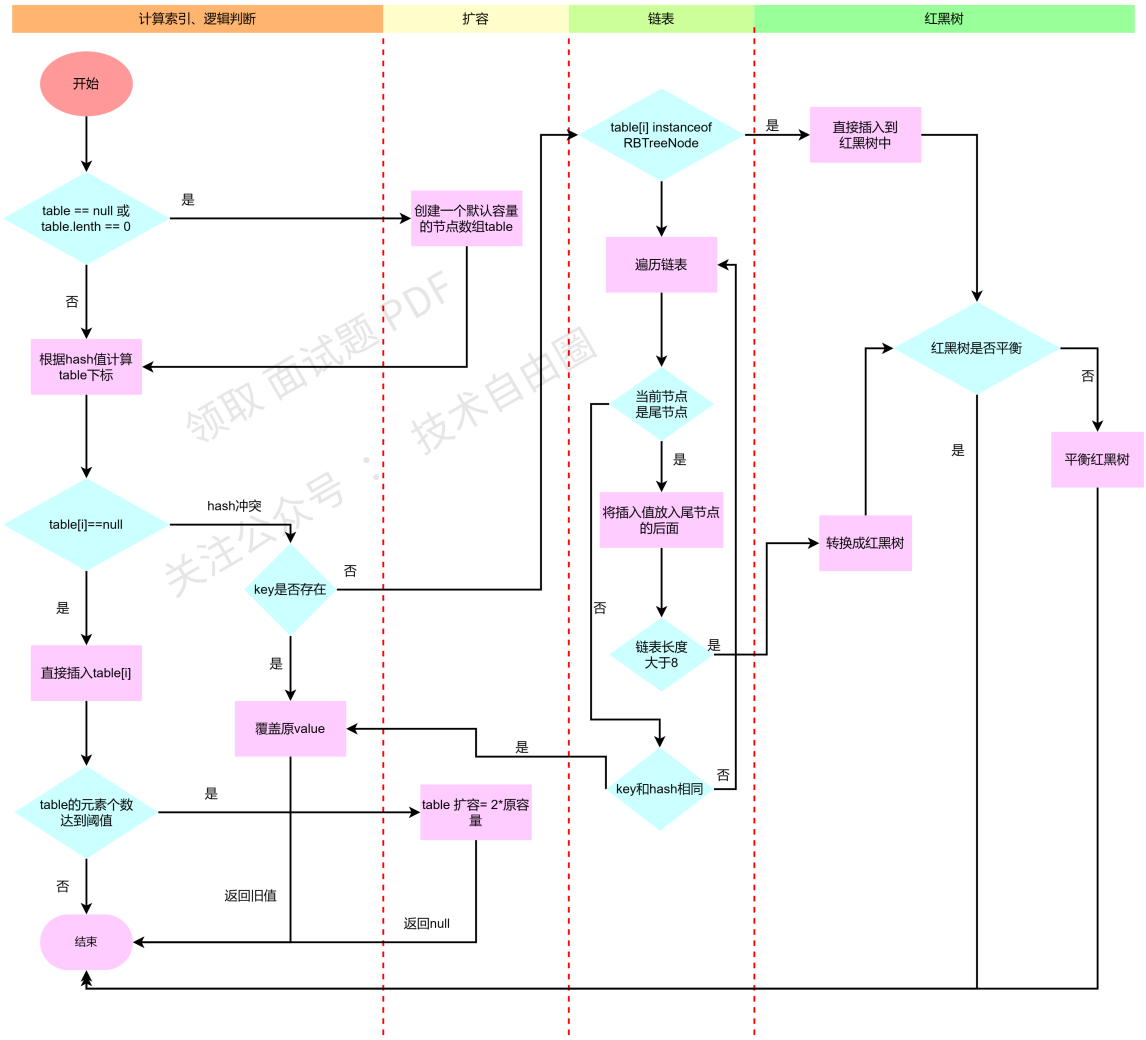
首先，我们先定义一个链表转红黑树的阈值，

```
//链表长度到达8时转成红黑树
private static final int TREEIFY_THRESHOLD = 8;
```

接下来我们看下put()方法的执行流程：

1. 首先判断table是否有足够的容量，若没有足够容量，就进行扩容操作；
2. 判断是否有hash冲突， 若无hash冲突，就把新增的key-value插入数组中对应的位置；
3. 若有hash冲突的时候，判断是否该数组下标的结点是树节点还是链表节点，若是树节点就添加到树上； 若是链表节点就采用尾节点插入。
4. 链表插入成功后需要判断一下链表的长度，若链表长度超过8时，就需要把链表转换成红黑树。

执行流程如下图所示：



put()方法优化后的代码如下：

```
/**
 * 插入节点
 *
 * @param key   key值
 * @param value value值
 * @return
 */
@Override
public V put(K key, V value) {
    //通过key计算hash值
    int hash = hash(key);

    //数组
    Node<K, V>[] tab;
    // 数组长度
    int n;

    // 数组的位置,即hash槽位
    int i;

    //根据数组长度和哈子自来寻址
    Node<K, V> parent;

    if ((tab = table) == null || (n = tab.length) == 0) {
        //第一次put的时候,调用ensureCapacity创建数组
```

```
26          tab = ensureCapacity();
27          n = tab.length;
28      }
29
30      // 开始时插入元素
31      if ((parent = tab[i = (n - 1) & hash]) == null) {
32          System.out.println("数组插入的key:" + key + ",value:" + value);
33          //如果没有hash碰撞,就直接插入数组中
34          tab[i] = new Node<>(hash, key, value, null);
35
36      } else { //有哈希碰撞时,需要判断是红黑树还是链表
37          // 下一个子结点
38          Node<K, V> next;
39          K k;
40          System.out.println("有哈希碰撞的key:" + key + ",value:" + value);
41          if (parent.hash == hash
42                  && ((k = parent.key) == key || (key != null &&
key.equals(k)))) {
43              // 哈希碰撞,且节点已存在,直接替换数组元素
44              next = parent;
45          } else if (parent instanceof RBTreeNode) {
46              // 如果是红黑树节点,就插入红黑树节点
47              System.out.println("往红黑树中插入的key:" + key + ",value:" +
value);
48              //先找到root根节点
49              int index = (tab.length - 1) & hash;
50              //取出红黑树的根结点
51              RBTreeNode<K, V> root = (RBTreeNode<K, V>) tab[index];
52
53
54              putRBTreeVal(root, hash, key, value);
55          } else {
56              System.out.println("链表插入的key:" + key + ",value:" + value);
57              printLinked(hash);
58              // 哈希碰撞, 链表插入
59              for (int linkSize = 0; ; ++linkSize) {
60                  //
System.out.println("linkSize="+linkSize+",node:"+parent);
61                  //如果当前结点的下一个结点为null,就直接插入
62                  if ((next = parent.next) == null) {
63                      System.out.println("new链表长度为:" + linkSize);
64                      parent.next = new Node<>(hash, key, value, null);
65                      // 链表长度 >8时,链表的第九个元素开始转换为红黑树
66                      if (linkSize >= TREEIFY_THRESHOLD - 1) {
67                          Node<K, V> testNode = tab[i];
68                          System.out.println("转换成红黑树插入的key:" + key +
",value:" + value);
69                         /* for (int linkSize1 = 0; linkSize1 <linkSize+1;
++linkSize1){
70
 System.out.println("linkSize1="+linkSize+",testNode:"+testNode);
71                              testNode = testNode.next;
72                          }*/
73                          //System.out.println("node:"+parent.next);
74
75                          System.out.println("链表长度为:" + linkSize);
76
77                          linkToRBTree(tab, hash, ++linkSize);
```

```
                    }
                    break;
                }
                if (next.hash == hash
                        && ((k = next.key) == key || (key != null &&
key.equals(k)))) {
                    //如果节点已经存在,直接跳出for循环
                    break;
                }
                parent = next;
            }
        }
    }
    if (++size > DEFAULT_CAPACITY * DEFAULT_LOAD_FACTOR) {
        ensureCapacity();
    }
    return value;
}
```

首先我们来看下当链表的长度大于8时，是如何把链表转换成红黑树的， 这里采用的是遍历链表，然后把链表中的节点一个个转换成功红黑树节点后，插入到红黑树中，最后做自平衡操作。

我们来看下把链表转换成红黑树的实现代码如下

```
/**
 * 把链表转换成红黑树
 *
 * @param tab
 * @param hash
 */
    private void linkToRBTree(Node<K, V>[] tab, int hash, int linkSize) {
        // 通过hash计算出当前table数组的位置
        int index = (tab.length - 1) & hash;
        Node<K, V> node = tab[index];
        int n = 0;

        //遍历链表中的每个节点,将链表转换为红黑树
        do {
            //把链表结点转换成红黑树结点
            RBTreeNode<K, V> next = replacementTreeNode(node, null);
            putRBTreeVal(next, hash, next.key, next.value);
            System.out.println("转换成红黑树数组的循环次数:" + n);
            ++n;
            node = node.next;
        } while (node != null);
        System.out.println("n:" + n);
        print(hash);
    }


/**
 * 把链表结点转换成红黑树结点
 *
 * @param p
 * @param next
 * @return
 */
```

```
RBTreeNode<K, V> replacementTreeNode(Node<K, V> p, Node<K, V> next) {
    return new RBTreeNode<K, V>(p.hash, p.key, p.value, next);
}
```

链表转红黑树的时候，调用了节点插入的 putRBTreeVal()方法， 由于红黑树是二叉树的其中一种，根据二叉树的特性，左子树的值都比根结点值小，右子树的值都比根结点值大。

由于同一颗红黑树的hash值都是相同的，在插入新节点之前，那我们就需要比较Key值的大小，大的往右子树放，小的就往左子树放，那么putRBTreeVal()方法的实现如下：

```
RBTreeNode<K, V> putRBTreeVal(RBTreeNode<K, V> tabnode, int hash, K key, V value) {
    if ((table[hash]) instanceof RBTreeNode) {

        RBTreeNode<K, V> root = (RBTreeNode<K, V>) table[hash];
        RBTreeNode<K, V> parent = root;
        RBTreeNode<K, V> node = root;

        int cmp = 0;

        // 先找到父节点
        do {
            parent = node;
            K k1 = node.key;
            //比较key值
            cmp = compare(key, k1);
            if (cmp > 0) {
                node = node.right;
            } else if (cmp < 0) {
                node = node.left;
            } else {
                V oldValue = node.value;
                node.key = key;
                node.value = value;
                node.hash = hash;
                return node;
            }

        } while (node != null);


        //插入新节点
        RBTreeNode<K, V> newNode = new RBTreeNode<>(hash, key, value, parent);
        if (cmp > 0) {
            parent.right = newNode;
        } else if (cmp < 0) {
            parent.left = newNode;
        }
        newNode.parent = parent;
        //插入成功后自平衡操作
        fixAfterPut(newNode, hash);
    } else {
        table[hash] = tabnode;
        fixAfterPut(tabnode, hash);
    }
    return null;
```

```
}
```

虽然说红黑树不是严格的平衡二叉查找树，但是红黑树插入/移除节点后仍然需要根据红黑树的五个特性进行自平衡操作。

由于红色破坏原则的可能性最小，插入的新节点颜色默认是红色。

若红黑树还没有根结点，新插入的红黑树节点就会被设置为根结点，然后根据特性2（根节点一定是黑色）把根节点设置为黑色后返回。

若父节点是黑色的，插入节点是红色的，不会影响红黑树的平衡，所以直接插入无需做自平衡。

若插入节点的父节点为红色的，那么该父节点不可能成为根结点，就需要找到祖父节点和叔父节点，那这个时候就会出现两种状态:（1）父亲和叔叔为红色；（2）父亲为红色，叔叔为黑色。

出现这两种状态的时候就需要做自平衡操作，

如果父节点和叔父节点都是红色的话，根据红黑树的特性4（红色节点不能相连）可以推断出祖父节点肯定为黑色。那这个时候只需进行变色操作即可，把祖父节点变成红色，父节点和叔父节点变成黑色操作

若叔父节点为黑色， 父节点为红色，若新插入的红色节点在父节点的左侧，此处就出现了LL型失衡，自平衡操作就需要先进行变色，然后父节点进行右旋操作；若新插入的红色节点在父节点的右侧，此处就出现了LR型失衡，自平衡操作就需要先父节点进行左旋，将父节点设置为当前节点，然后再按LL型失衡操作进行自平衡操作即可。

若叔叔为黑节点，父亲为红色，并且父亲节点是祖父节点的右子节点，如果新插入的节点为其父节点的右子节点，此时就出现了RR型失衡操作， 自平衡处理操作是先进行变色处理，把父节点设置成黑色，把祖父节点设置为红色，然后祖父节点进行左旋操作；若新插入节点，为其父节点的左子节点，此时就出现了RL型失衡，自平衡操作是对父节点进行右旋，并将父节点设置为当前节点，接着按RR型失衡进行自平衡操作。

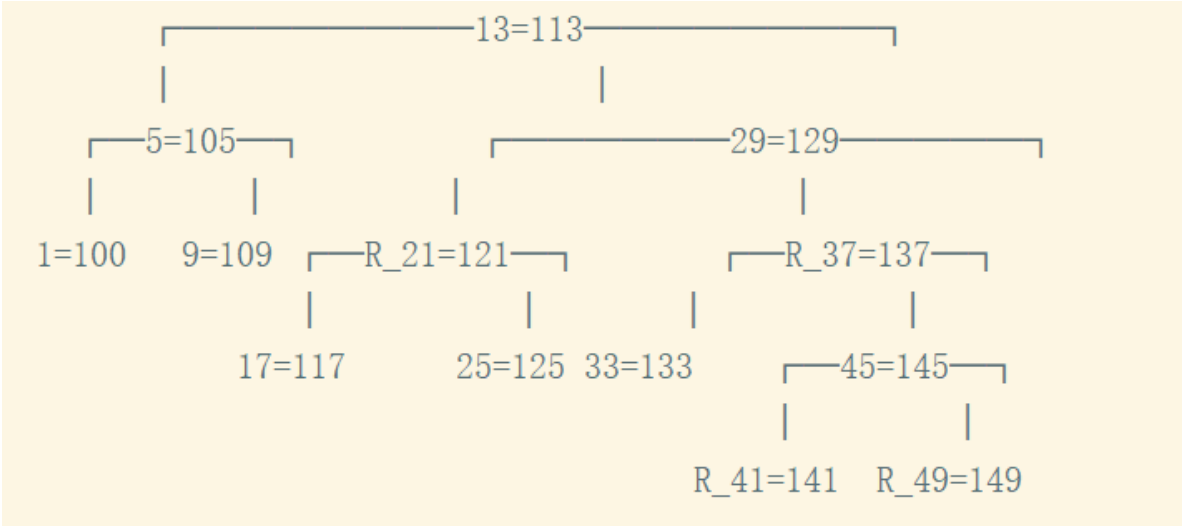
自平衡操作的实现代码如下：

```java
/**
 * 添加后平衡二叉树并设置结点颜色
 *
 * @param node 新添结点
 * @param hash hash值
 */
private void fixAfterPut(RBTreeNode<K, V> node, int hash) {
    RBTreeNode<K, V> parent = node.parent;

    // 添加的是根节点 或者 上溢到达了根节点
    if (parent == null) {
        black(node);
        return;
    }

    // 如果父节点是黑色，直接返回
    if (isBlack(parent)) {
        return;
    }

    // 叔父节点
    RBTreeNode<K, V> uncle = parent.sibling();
    // 祖父节点
    RBTreeNode<K, V> grand = red(parent.parent);
```

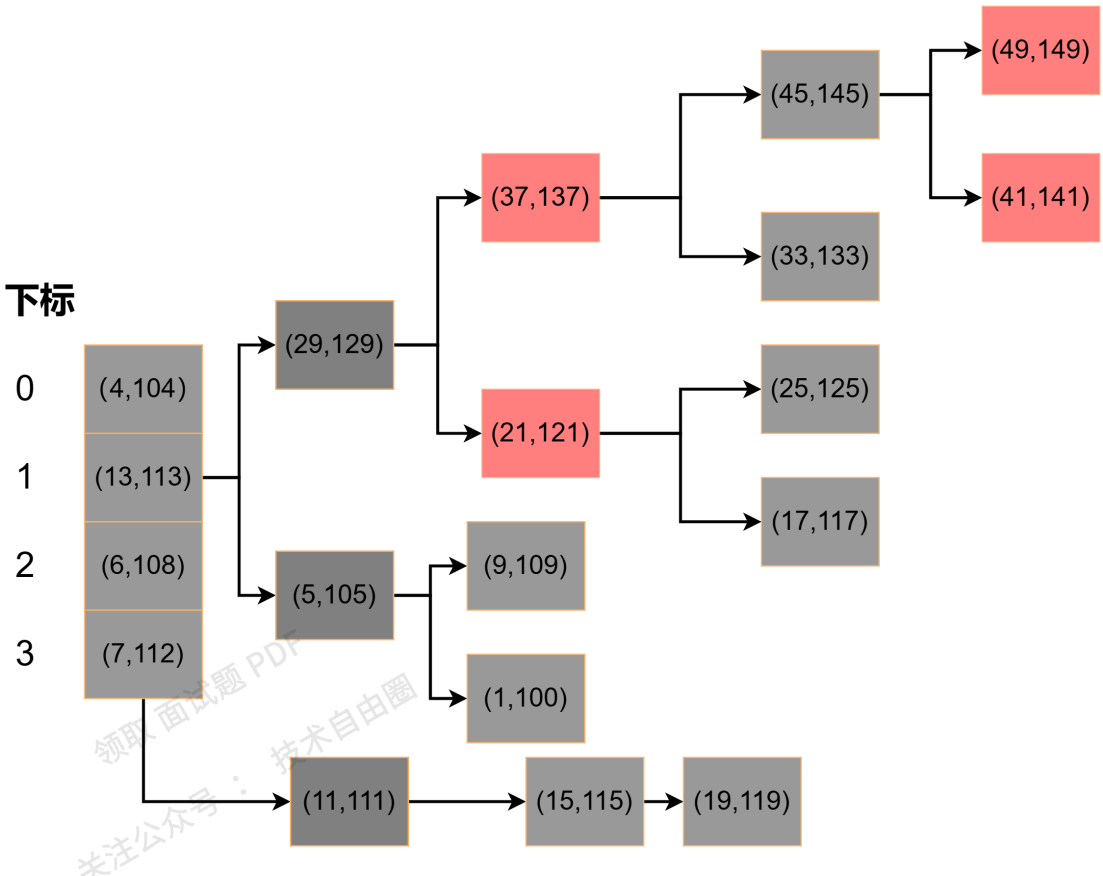
```
    if (isRed(uncle)) { // 叔父节点是红色【B树节点上溢】
        black(parent);
        black(uncle);
        // 把祖父节点当做是新添加的节点
        fixAfterPut(grand, hash);
        return;
    }

    // 叔父节点不是红色
    if (parent.isLeftChild()) { // L
        if (node.isLeftChild()) { // LL
            black(parent);
        } else { // LR
            black(node);
            rotateLeft(parent, hash);
        }
        rotateRight(grand, hash);
    } else { // R
        if (node.isLeftChild()) { // RL
            black(node);
            rotateRight(parent, hash);
        } else { // RR
            black(parent);
        }
        rotateLeft(grand, hash);
    }
}
/**
 * 左旋
 *
 * @param grand
 */
private void rotateLeft(RBTreeNode<K, V> grand, int hash) {
    RBTreeNode<K, V> parent = grand.right;
    RBTreeNode<K, V> child = parent.left;
    grand.right = child;
    parent.left = grand;
    afterRotate(grand, parent, child, hash);
}

/**
 * 右旋
 *
 * @param grand
 */
void rotateRight(RBTreeNode<K, V> grand, int hash) {
    RBTreeNode<K, V> parent = grand.left;
    RBTreeNode<K, V> child = parent.right;
    grand.left = child;
    parent.right = grand;
    afterRotate(grand, parent, child, hash);
}

void afterRotate(RBTreeNode<K, V> grand, RBTreeNode<K, V> parent,
RBTreeNode<K, V> child, int hash) {
    // 让parent称为子树的根节点
    parent.parent = grand.parent;
    if (grand.isLeftChild()) {
```

```
            grand.parent.left = parent;
        } else if (grand.isRightChild()) {
            grand.parent.right = parent;
        } else { // grand是root节点
            int index = table.length - 1 & hash;
            table[index] = parent;
        }

        // 更新child的parent
        if (child != null) {
            child.parent = grand;
        }

        // 更新grand的parent
        grand.parent = parent;
        print(hash);
    }
```

使用单元测试看下红黑树的结果：



存储结构如下：



同样，查找Key-Value键值对的get()方法也同样需要做优化， 主要优化的内容就是在红黑树中查找Key-Value键值对；

实现步骤如下：

（1）通过hash值找到数组table的下标，

（2）通过数组table下标判断是否是红黑树节点，若是红黑树节点就在红黑树中查找；

（3）通过数组table下标判断是否是链表节点，若是链表节点就在链表中查找；

（4）若结点都不在红黑树和链表中，就在数组table中查找；

实现代码如下：

```
 @Override
public V get(Object key) {
    Node<K, V> e;

    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

/**
 * 通过key值在数组/链表/红黑树中查找value值
 *
 * @param hash
 * @param key
 * @return
 */
private Node<K, V> getNode(int hash, Object key) {
    //数组
    Node<K, V>[] tab;

```

```
    //数组长度
    int n;
    // (n-1)$hash 获取该key对应的数据节点的hash槽位,即链表的根结点
    Node<K, V> parent;

    //root的子节点
    Node<K, V> next;

    K k;

    //如果数组为空,并且长度为空, hash槽位对应的节点为空,就返回null
    if ((tab = table) != null && (n = table.length) > 0
            && (parent = tab[(n - 1) & hash]) != null) {
        // 如果计算出来的hash槽位所对应的结点hash值等于hash值,结点的key=查找key值,
        // 返回hash槽位对应的结点,即数组
        if (parent.hash == hash && ((k = parent.key) == key || (key !=
null && key.equals(k)))) {
            return parent;
        }
        //如果不在根结点,在子结点
        if ((next = parent.next) != null) {
            //有子结点的时候,需要判断是链表还是红黑树

            //在链表中查找,需要通过循环一个个往下查找
            while (next != null) {
                if (next.hash == hash && ((k = next.key) == key || (key !=
null && key.equals(k)))) {
                    return next;
                }
                next = next.next;
            }

        }

        if (parent instanceof RBTreeNode) {
            //在红黑树中查找
            return getRBTreeNode((RBTreeNode<K, V>) parent, hash, key);
        }
    }
    return null;
}


 /**
 * 在红黑树中查找结点
 *
 * @param node 根结点
 * @param hash hash(key) 计算出的哈希值
 * @param key  需要寻找的key值
 * @return
 */
public Node<K, V> getRBTreeNode(RBTreeNode<K, V> node, int hash, Object
key) {

    // 存储查找结果
    Node<K, V> result = null;
    K k;
    int cmp = 0;
```

```
    while (node != null) {
        //左节点
        RBTreeNode<K, V> nl = node.left;
        // 右节点
        RBTreeNode<K, V> nr = node.right;
        K k2 = node.key;
        int hash1 = node.hash;
        //比较hash值,判断是在左子树还是右子树
        if (hash > hash1) {
            //查找结点在右子树
            node = nr;
        } else if (hash < hash1) {
            //查找结点在左子树
            node = nl;
        } else if ((k = node.key) == key || (key != null &&
key.equals(k))) {
            //如果key 相等,就返回node
            return node;
        } else if (nl == null) {
            node = nr;
        } else if (nr == null) {
            node = nl;
        } else if (key != null & k2 != null
                && key.getClass() == k2.getClass()
                && key instanceof Comparable
                && (cmp = compare(key, k2)) != 0
        ) {
            node = cmp > 0 ? node.right : node.left;
        } else if (node.right != null && (result =
getRBTreeNode(node.right, hash, key)) != null) {
            return result;
        } else {
            node = node.left;
        }
    }
    return null;
}

```

如果HashMap需要通过key值移除Key-Value键值对，首先通过key值查找到节点，然后进行移除；

若需移除的节点在红黑树中，首先需要判断移除节点的度是多少，若度为2的话，就需要先找到后继节点后才可以移除，若度为1或0的话，可以直接进行移除操作，红黑树移除节点同样也需要判断红黑树是否平衡，若不平衡就需要红黑树自平衡操作，自平衡操作和插入节点的平衡操作一样，就不在赘述了。具体代码实现如下：

```
 /**
 * 结点删除
 *
 * @param key
 * @return
 */
@Override
public V remove(K key) {
    int hash = hash(key);
    //数组
    Node<K, V>[] tab;
```

```
    Node<K, V> parent;
    K k;
    int index;
    V oldValue = null;
    //节点是存在的
    if ((parent = table[index = (table.length - 1) & hash]) != null) {
        if (parent instanceof RBTreeNode) { // 红黑树删除


            RBTreeNode<K, V> willNode = (RBTreeNode<K, V>) parent;
            //找到要删除的结点
            RBTreeNode<K, V> removeNode = (RBTreeNode<K, V>)
getRBTreeNode(willNode, hash, key);
            oldValue = removeNode.value;

            // 度为2 的结点
            if (removeNode.hasTwoChildren()) {
                //找到后继接地那
                RBTreeNode<K, V> s = successor(removeNode);
                removeNode.key = s.key;
                removeNode.value = s.value;
                removeNode.hash = s.hash;
                // 删除后继节点
                removeNode = s;
            }

            // 删除node节点（node的度必然是1或者0）
            RBTreeNode<K, V> replacement = removeNode.left != null ?
removeNode.left : removeNode.right;

            if (replacement != null) { // node是度为1的节点
                // 更改parent
                replacement.parent = removeNode.parent;
                // 更改parent的left、right的指向
                if (removeNode.parent == null) { // node是度为1的节点并且是根
节点
                    table[index] = replacement;
                } else if (removeNode == removeNode.parent.left) {
                    removeNode.parent.left = replacement;
                } else { // node == node.parent.right
                    removeNode.parent.right = replacement;
                }

                // 删除节点之后的处理
                fixAfterRemove(replacement, hash);
            } else if (removeNode.parent == null) { // node是叶子节点并且是根
节点
                table[index] = null;
            } else { // node是叶子节点，但不是根节点
                if (removeNode == removeNode.parent.left) {
                    removeNode.parent.left = null;
                } else { // node == node.parent.right
                    removeNode.parent.right = null;
                }

                // 删除节点之后的处理
                fixAfterRemove(removeNode, hash);

```

```
            }
            System.out.println("删除结点后的红黑树:"+key);
            print(hash);
            size--;
            return oldValue;
        } else if (parent.next != null) { //链表删除
            Node<K, V> node = parent;
            Node<K,V> preNode = null;
            for (int linkSize = 0; ; ++linkSize) {

                if (node.hash == hash
                        && ((k = node.key) == key || (key != null &&
key.equals(k)))) {
                    if (linkSize == 0) {
                        //如果是第一个结点,就把第二个结点挂载到table中
                        oldValue = node.value;
                        table[index] = node.next;
                    } else {
                        if (preNode.next.next == null) {
                            //删除的如是尾节点, 就把尾节点置为null
                            oldValue = preNode.next.value;
                            preNode.next = null;


                        } else {

                            oldValue = preNode.next.value;
                            preNode.next = preNode.next.next;
                        }
                    }
                    size--;
                    break;
                }
                //删除结点的前结点
                preNode = node;
                if ((node = node.next) == null) {
                    break;
                }

            }
            System.out.println("链表删除元素:"+key);
            printLinked(hash);
        } else { //数组删除
            if (parent.hash == hash
                    && ((k = parent.key) == key || (key != null &&
key.equals(k)))) {
                oldValue = parent.value;
                for (int i = index + 1; i < table.length; i++) {
                    table[i - 1] = table[i];
                }
                --size;
                table[(table.length-1)] =null;
                return  oldValue;
            }


        }
    }
```

```
    return oldValue;
}

private RBTreeNode<K, V> successor(RBTreeNode<K, V> node) {

    // 前驱节点在左子树当中（right.left.left.left....）
    RBTreeNode<K, V> p = node.right;
    if (p != null) {
        while (p.left != null) {
            p = p.left;
        }
        return p;
    }

    // 从父节点、祖父节点中寻找前驱节点
    while (node.parent != null && node == node.parent.right) {
        node = node.parent;
    }

    return node.parent;
}

private void fixAfterRemove(RBTreeNode<K, V> node, int hash) {
    // 如果删除的节点是红色
    // 或者 用以取代删除节点的子节点是红色
    if (isRed(node)) {
        black(node);
        return;
    }

    RBTreeNode<K, V> parent = node.parent;
    if (parent == null) return;

    // 删除的是黑色叶子节点【下溢】
    // 判断被删除的node是左还是右
    boolean left = parent.left == null || node.isLeftChild();
    RBTreeNode<K, V> sibling = left ? parent.right : parent.left;
    if (left) { // 被删除的节点在左边，兄弟节点在右边
        if (isRed(sibling)) { // 兄弟节点是红色
            black(sibling);
            red(parent);
            rotateLeft(parent, hash);
            // 更换兄弟
            sibling = parent.right;
        }

        // 兄弟节点必然是黑色
        if (isBlack(sibling.left) && isBlack(sibling.right)) {
            // 兄弟节点没有1个红色子节点，父节点要向下跟兄弟节点合并
            boolean parentBlack = isBlack(parent);
            black(parent);
            red(sibling);
            if (parentBlack) {
                fixAfterRemove(parent, hash);
            }
        } else { // 兄弟节点至少有1个红色子节点，向兄弟节点借元素
            // 兄弟节点的左边是黑色，兄弟要先旋转
            if (isBlack(sibling.right)) {
```

```
                rotateRight(sibling, hash);
                sibling = parent.right;
            }

            color(sibling, colorOf(parent));
            black(sibling.right);
            black(parent);
            rotateLeft(parent, hash);
        }
    } else { // 被删除的节点在右边，兄弟节点在左边
        if (isRed(sibling)) { // 兄弟节点是红色
            black(sibling);
            red(parent);
            rotateRight(parent, hash);
            // 更换兄弟
            sibling = parent.left;
        }

        // 兄弟节点必然是黑色
        if (isBlack(sibling.left) && isBlack(sibling.right)) {
            // 兄弟节点没有1个红色子节点，父节点要向下跟兄弟节点合并
            boolean parentBlack = isBlack(parent);
            black(parent);
            red(sibling);
            if (parentBlack) {
                fixAfterRemove(parent, hash);
            }
        } else { // 兄弟节点至少有1个红色子节点，向兄弟节点借元素
            // 兄弟节点的左边是黑色，兄弟要先旋转
            if (isBlack(sibling.left)) {
                rotateLeft(sibling, hash);
                sibling = parent.left;
            }

            color(sibling, colorOf(parent));
            black(sibling.left);
            black(parent);
            rotateRight(parent, hash);
        }
    }
}
```

熟背JDK的hashMap源码剖析

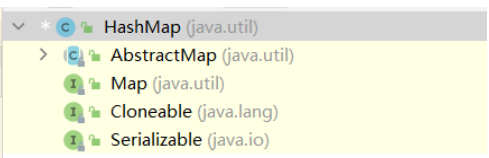
最终，要想让面试官五体投地，咱们还是的熟背hashMap源码。

(hashmap 的源码剖析是jdk1.8的)

HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。

HashMap 继承于AbstractMap，实现了Map、Cloneable、java.io.Serializable接口。

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
```

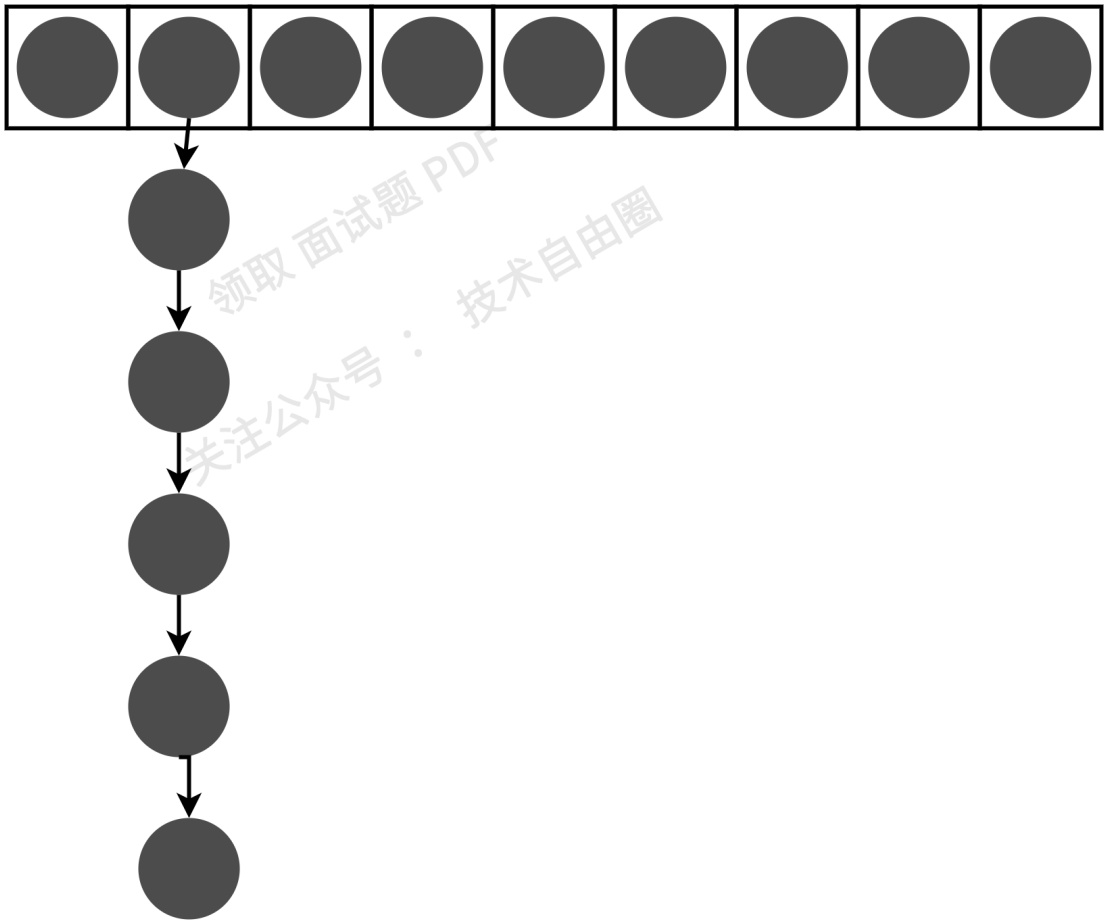


HashMap 的实现不是同步的，这意味着它是线程不安全的。

它的key、value都可以为null。此外，HashMap中的映射是无序的。

Jdk1.7中HashMap的实现的基础数据结构是数组+链表，每一对key->value的键值对组成Entity类以双向链表的形式存放到这个数组中；

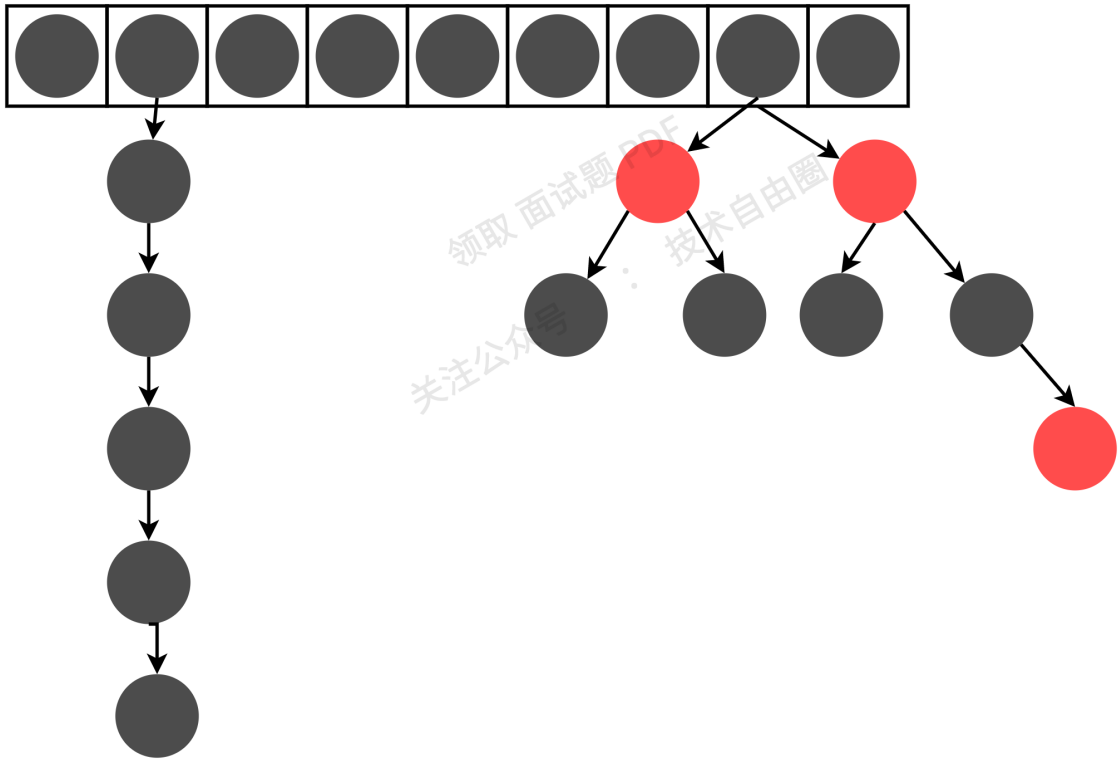
元素在数组中的位置由key.hashCode()的值决定，如果两个key的哈希值相等，即发生了哈希碰撞，则这两个key对应的Entity将以链表的形式存放在数组中。如下图所示



在jdk1.8及以后的版本，HashMap的实现的基础数据结构是数组+链表+红黑树；

为了提高hashmap的效率，新增了红黑树，如果链表的长度超过8，且table的容量必须大于64时，会将链表转换成红黑树。

如下图所示：



既然红黑树的效率高，为什么不直接用红黑树？为什么链表超过8转换为红黑树？

官方给出的解释如下：

```
* 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
```


这段话的意思提现了时间和空间平衡的思想。

最开始使用链表的时候，空间占用是比较少的，而且由于链表短，所以查询时间也没有太大的问题。

可是当链表越来越长，需要用红黑树的形式来保证查询的效率。

对于何时应该从链表转化为红黑树，需要确定一个阈值，这个阈值默认为 8，链表长度达到 8 就转成红黑树，而当长度降到 6 就转换回链表。

如果 hashCode 分布良好，也就是hash计算的结果离散好的话，那么红黑树这种形式是很少会被用到的，因为各个值都均匀分布，很少出现链表很长的情况。

在理想情况下，链表长度符合泊松分布，各个长度的命中概率依次递减，当长度为8的时候，概率仅为0.00000006。这是一个小于千万分之一的概率，通常我们的 Map 里面是不会存储这么多的数据的，所以通常情况下，并不会发生从链表向红黑树的转换。

事实上，链表长度超过 8 就转为红黑树的设计，更多的是为了防止用户自己实现了不好的哈希算法时导致链表过长，从而导致查询效率低，而此时转为红黑树更多的是一种保底策略，用来保证极端情况下查询的效率。

除了jdk1.8中新增了红黑树外，从jdk1.8开始，链表节点的插入使用尾插入替换了jdk1.7的头插入，替换的原因是在并发情况下，头插法会出现链表成环的问题，

数组下标获取

为了利用数组索引进行快速查找，hashMap采取hash算法的是先将 key值映射成数组下标。

hash()算法的源码如下：

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

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

从源码中可以看到，没有直接使用hashCode返回hash值，是因为hashCode返回的是int值，它的范围是在-2147483648-2147483647。如果存的元素并不多的情况，创建int范围的的数组空间太过于浪费。

hash()分为两个步骤：

①先得到扰动后的key的hashCode：(h = key.hashCode())^ (h >>> 16)

首先 h = key.hashCode()是key对象的一个hashCode，每个不同的对象其哈希值都不相同，其实底层是对象的内存地址的散列值，所以最开始的h是key对应的一个整数类型的哈希值；右移16位（h>>>16），然后高位补0是为了让高16位参与进来。

②再将hashCode映射成有限的数组下标index：(n - 1) & hash；

采用异或（^)运算是为了让h的低16位更有散列性。 为什么异或运算的散列性更好呢？我们来看组运算例子；

上面的计算过程如下：
与运算：其中1&1=1，其他三种情况1&0=0, 0&0=0, 0&1=0 都等于0，可以看到与运算的结果更多趋向于0，这种散列效果就不好了，运算结果会比较集中在小的值

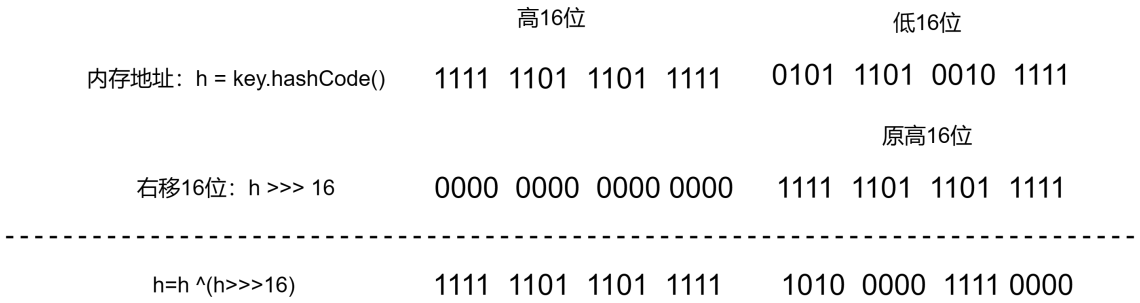
或运算：其中0&0=0，其他三种情况 1&0=1, 1&1=1, 0&1=1 都等于1，可以看到或运算的结果更多趋向于1，散列效果也不好，运算结果会比较集中在大的值

异或运算：其中0&0=0, 1&1=0，而另外0&1=1, 1&0=1 ，可以看到异或运算结果等于1和0的概率是一样的，这种运算结果出来当然就比较分散均匀了

总的来说，与运算的结果趋向于得到小的值，或运算的结果趋向于得到大的值，异或运算的结果大小值比较均匀分散，这就是我们想要的结果。

右移16位，然后再与原hashcode做异或运算，是为了高低位二进制特征混合起来，使该hashCode映射成数组下标时可以更均匀。更好地均匀散列，从而减少碰撞，进一步降低hash冲突的几率。

所以计算的过程如下：

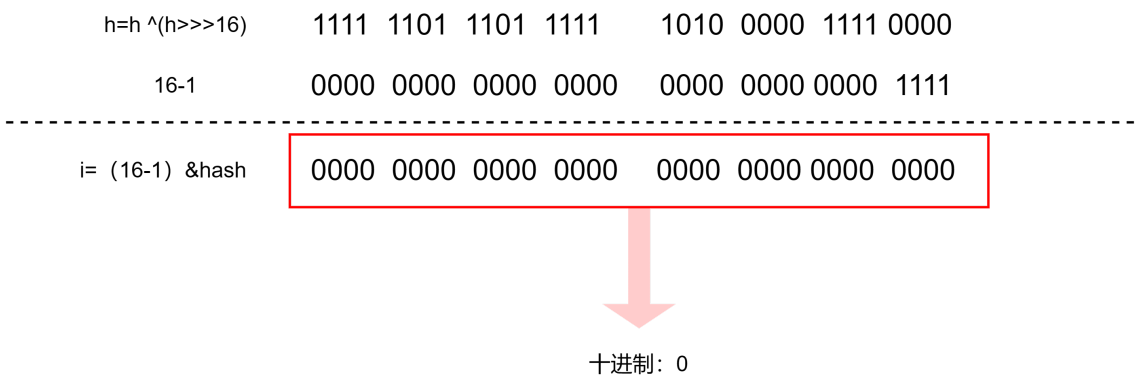


这部分产生的hash值是h，这个数有可能很大，不能直接拿来当数组下标，那么接下来就需要进行第二部分的内容(n - 1) & hash，这部分内容就是hashMap中获取数组下标的代码，n=table.length 新数组长度。 hash是参数h（上一步计算返回的hash结果）。

HashMap为了存取高效，要尽量较少碰撞，就是要尽量把数据分配均匀，每个链表长度大致相同。想到的办法就是取模运算：hash%length，但是在计算机中取模运算效率与远不如位移运算(&)高。主要原因是位运算直接对内存数据进行操作，不需要转成十进制，因此处理速度非常快。所以官方决定采用使用位运算(&)来实现取模运算(%)，也就是源码中优化为:hash&(length-1)。

(n - 1) & hash的计算过程如下： 假设数组长度为16，经过(h = key.hashCode())^ (h >>> 16)得到的hash值的低16位是1101001010100110（一般数hashmap数组长度都在2^16范围内，所以就用低16位演示了）；

首先，n-1 = 15，转换成二进制是1111.然后与hash值进行与运算(当两个数字对应的二进位均为1时，结果位为1，否则为0。参与运算的数以补码出现).计算过程如下：



结果是0000，换算成十进制就是0，对应的数组下标就是0；

这样两步就完成了key对象映射到指定数组索引上了。

table桶的扩容机制

哈希桶数组的大小， 在空间成本和时间成本之间权衡，时间和空间之间进行权衡：

- 如果哈希桶数组很大，即使较差的Hash算法也会比较分散， 空间换时间
- 如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞， 时间换空间

其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。

在剖析table扩容之前，我们先来了解hashMap中几个比较重要的属性。

HashMap中有两个比较重要的属性：加载因子（loadFactor）和边界值（threshold），在HashMap时，就会涉及到这两个关键初始化参数，loadFactor和threshold的源码如下：

```
final float loadFactor;
int threshold;
```

Node[] table的初始化长度length(默认值是16)，length大小必须为2的n次方，主要是为了方便扩容。

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

loadFactor 为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node 个数。threshold 、length 、loadFactor 三者之间的关系：

threshold = length * Load factor

默认情况下 threshold = 16 * 0.75 =12。

threshold就是允许的哈希数组最大元素数目，超过这个数目就重新resize(扩容)，扩容后的哈希数组 容量length 是之前容量length 的两倍。

threshold是通过初始容量和LoadFactor计算所得，在初始HashMap不设置参数的情况下，默认边界值为12。

如果HashMap中Node的数量超过边界值，HashMap就会调用resize()方法重新分配table数组。这将会导致HashMap的数组复制，迁移到另一块内存中去，从而影响HashMap的效率。

loadFactor默认值是0.75，官方给的解释如下：

```
* <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 {@code HashMap} class, including
* {@code get} and {@code put}).  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.
```


大概意思是：作为一般规则，默认负载因子 (.75) 在时间和空间成本之间提供了良好的折衷。较高的值会减少空间开销，但会增加查找成本（反映在HashMap类的大多数操作中，包括get和put)。在设置其初始容量时，应考虑映射中的预期条目数及其负载因子，以尽量减少重新哈希操作的次数。 如果初始容量大于最大条目数除以负载因子，则不会发生重新哈希操作。

loadFactor 也是可以调整的，建议大家尽量不要修改，除非在时间和空间比较特殊的情况：

- 如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；
- 如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1

接下来我们再来看一个size属性。

size属性是HashMap中实际存在的键值对数量；而length是哈希桶数组table的长度。

当HashMap中的元素越来越多的时候，碰撞的几率也就越来越高，所以为了提高查询的效率，就要对HashMap的数组进行扩容，其实数组扩容这个操作在ArrayList中也出现了，所以这是一个通用的操作，

table是一个Node<K,V>类型的数组，其定义如下：

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

```

Node 类作为 HashMap 中的一个内部类，除了 key、value 两个属性外，还定义了一个next 指针，当有哈希冲突时，HashMap 会用之前数组当中相同哈希值对应存储的 Node 对象，通过指针指向新增的相同哈希值的Node 对象的引用。

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

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

table在首次使用put的时候初始化，并根据需求调整大小。

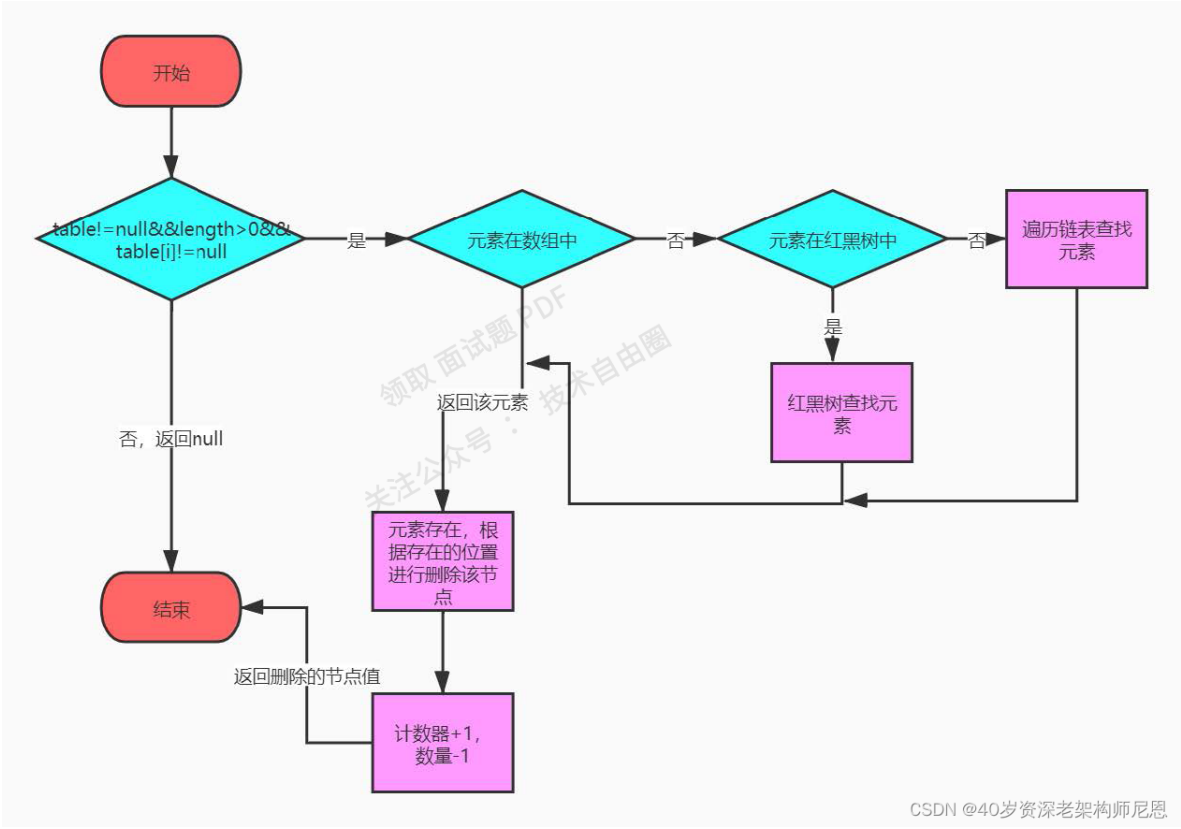
当table中的Node<K,V>个数超过数组大小*loadFactor时，就会触发扩容机制。每次扩容的容量都是之前容量的 2 倍。HashMap 的容量是有上限的，必须小于 1<<30，即 1073741824。如果容量超出了这个数，则不再增长，且阈值会被设置为 Integer.MAX_VALUE。

JDK7 中的扩容机制
(1)空参数的构造函数：以默认容量、默认负载因子、默认阈值初始化数组。内部数组是空数组。
(2)有参构造函数：根据参数确定容量、负载因子、阈值等。
(3)第一次 put 时会初始化数组，其容量变为不小于指定容量的 2 的幂数，然后根据负载因子确定阈值。
(4)如果不是第一次扩容，则 新容量=旧容量 x 2 ，新阈值=新容量 x 负载因子 。

JDK8 的扩容机制
(1)空参数的构造函数：实例化的 HashMap 默认内部数组是 null，即没有实例化。第一次调用 put 方法时，则会开始第一次初始化扩容，长度为 16。
(2)有参构造函数：用于指定容量。会根据指定的正整数找到不小于指定容量的 2 的幂数，

哈希桶数组table的扩容核心是resize()方法。在resize的时候会将原来的数组rehash重新计算hash值转移到新数组上。在HashMap数组扩容之后，最消耗性能的点是原数组中的数据必须重新计算其在新数组中的位置，并放进去。

resize()方法扩容流程如下：



那接下来我们就来看下resize()方法中是如何初始化table数组和table扩容的。源码如下：

```
final Node<K,V>[] resize() {
    //保存原数组
    Node<K,V>[] oldTab = table;
    //保存原数组长度
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    //保存原阈值(没有初始化的时候是0)
    int oldThr = threshold;
    //定义成员变量 新数组长度,新阈值
    int newCap, newThr = 0;
    //如果原数组长度>0
    if (oldCap > 0) {
        //如果原数组长度大于最大容量
        if (oldCap >= MAXIMUM_CAPACITY) {
            //增加阈值
            threshold = Integer.MAX_VALUE;
            //返回
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            // 把数组长度变为原的两倍看是否小于最大容量,且原数组长度大于默认初始容量16
            //阈值也扩大到原来的2倍
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        //如果原阈值>0,将原阈值赋给新数组长度
        newCap = oldThr;
    else {               // zero initial threshold signifies using
defaults
        // 零初始阈值表示使用默认值,新容量为16.
        newCap = DEFAULT_INITIAL_CAPACITY;
        //新阈值为0.75*16=12
```

```
32          newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
33      }
34      //新阈值为0
35      if (newThr == 0) {
36          float ft = (float)newCap * loadFactor;
37          newThr = (newCap < MAXIMUM_CAPACITY && ft <
(float)MAXIMUM_CAPACITY ?
38                    (int)ft : Integer.MAX_VALUE);
39      }
40      //新阈值赋值给成员变量threshold
41      threshold = newThr;
42      @SuppressWarnings({"rawtypes","unchecked"})
43      //创建一个长度确定的新节点数组
44      Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
45      //新数组赋值给成员变量table
46      table = newTab;
47      //原数组不为空
48      if (oldTab != null) {
49          //对数组进行遍历
50          for (int j = 0; j < oldCap; ++j) {
51              Node<K,V> e;
52              //如果元素组上元素不为空
53              if ((e = oldTab[j]) != null) {
54                  oldTab[j] = null;
55                  //e下一个元素如果为空,说明只有单节点
56                  if (e.next == null)
57                      //把e放到新数组中,e要么在原来的位置,要么在 原来的位置+旧容量
58                      newTab[e.hash & (newCap - 1)] = e;
59                  else if (e instanceof TreeNode) //如果e是树节点
60                      //用拆分树的方式进行转移
61                      ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
62                  else { // preserve order    低位表示:原位置   高位表示:原位置+旧容
量
63                      // 非单节点和树节点情况,也就是有链表结构
64                      //低位的头节点和尾节点
65                      Node<K,V> loHead = null, loTail = null;
66                      // 高位的头节点和尾节点
67                      Node<K,V> hiHead = null, hiTail = null;
68                      Node<K,V> next;
69                      do {
70                          next = e.next;
71                          //如果放到新数组原位置上
72                          if ((e.hash & oldCap) == 0) {
73                              //如果低位尾节点为null,说明位置上没有节点
74                              if (loTail == null)
75                                  //e作为头节点
76                                  loHead = e;
77                              else  //低位尾节点不为空,说明位置上右节点
78                                  // 让低位尾节点下一位指向e
79                                  loTail.next = e;
80                              //e成为高位尾节点
81                              loTail = e;
82                          }
83                          else {  //放的位置为 原位置+原容量
84                              //若高位尾节点没有元素
85                              if (hiTail == null)
86                                  //e作为高位头结点
87                                  hiHead = e;
```

```
                                else //高位已有元素时
                                    //让高位尾节点next指向e
                                    hiTail.next = e;
                                //所以e成为了高位位节点
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        //如果低位尾节点不为空
                        if (loTail != null) {
                            //让低位下一位为空
                            loTail.next = null;
                            //将原来下标指向低位的链表
                            newTab[j] = loHead;
                        }
                        //如果高位尾节点不为空
                        if (hiTail != null) {
                            //让高位下一位为空
                            hiTail.next = null;
                            ////将原来下标指向高位的链表
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        //返回新数组
        return newTab;
    }
```

从源码中我们知道，默认的数组长度length 是16.这个主要是为了实现均匀分布。因为在使用2的幂的数字的时候，Length-1的值是所有二进制位全为1，这种情况下，index的结果等同于HashCode后几位的值。只要输入的HashCode本身分布均匀，Hash算法的结果就是均匀的。

table的threshold 阈值是通过初始容量和 loadFactor计算所得，在初始HashMap 不设置参数的情况下，默认边界值为12（16*0.75*）。*当HashMap中元素个数超过16*0.75=12的时候，就把数组的大小扩展为2*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置。

table的扩容分为两步：
第一步：扩容——创建一个新的Entry空数组，长度是原数组的2倍。
第二步：ReHash——遍历原Entry数组，把所有的Entry重新Hash到新数组。

扩容的后重新计算hash的原因是因为长度扩大以后，Hash的规则也随之改变。

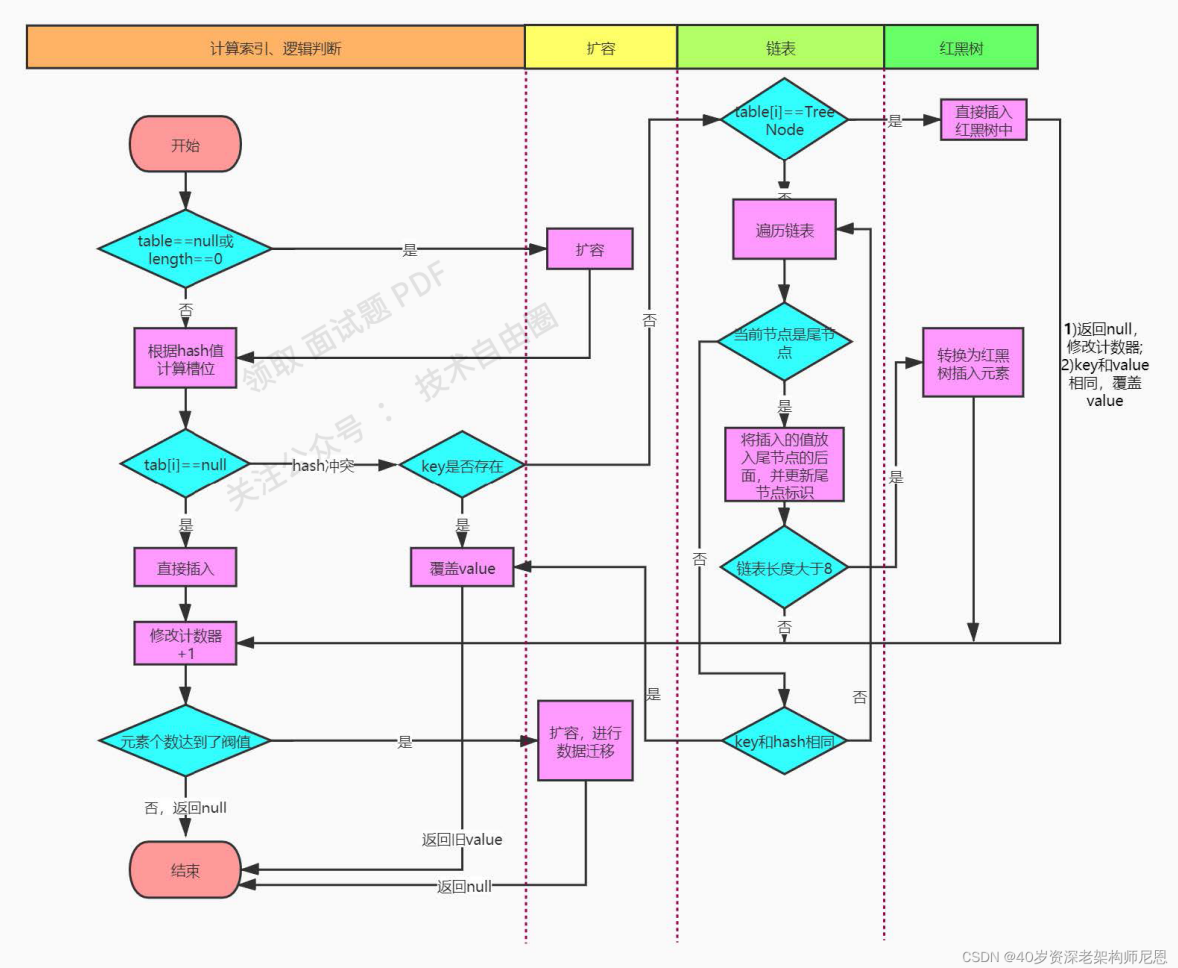
put(K key, V value) 添加key-value

首先我们来看下put(K key, V value)的源码下：

```
public V put(K key, V value) {
    //返回putVal方法， 给key进行了一次rehash
    return putVal(hash(key), key, value, false, true);
}
```

从源码可以看到，put()方法首先调用hash()算法计算hash值，然后调用putVal()对添加的key-value键值对进行存储。

在putVal()中主要完成了一下几件事：
(1)如果发现当前的桶数组为null，则调用resize()方法进行初始化
(2)如果没有发生哈希碰撞，则直接放到对应的桶中
(3)如果发生哈希碰撞，且节点已经存在，就替换掉相应的value
(4)如果发生哈希碰撞，且桶中存放的是树状结构，则挂载到树上
(5)如果碰撞后为链表，添加到链表尾，如果链表长度超过TREEIFY_THRESHOLD默认是8，则将链表转换为树结构
(6)数据put完成后，如果HashMap的总数超过threshold就要resize
putVal的执行流程如下：



我们来看下添加树节点的方法putTreeVal()的源码；

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    //tab:引用hashMap的散列表
    //p:表示当前散列表的元素
    // n :表示散列表数组的长度
    //i: 表示路由寻址的结果
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    //延迟初始化逻辑，当第一次调用putVal的时候，才去初始化HashMap对象的散列表大小

    if ((tab = table) == null || (n = tab.length) == 0)
        //进入此处表示第一次调用put方法
        //第一次put时,调用resize()进行桶数组初始化
        n = (tab = resize()).length;
    //（n-1)&hash 计算 Node 的存储位置，如果判断 Node 不在哈希表中(链表的第一个节
    //点位置），新增一个 Node，并加入到哈希表中
    if ((p = tab[i = (n - 1) & hash]) == null)
        //如果没有哈希碰撞,直接放入数组中
```

```
        tab[i] = newNode(hash, key, value, null);
    else {
        //hash 冲突了
        //e：不为null时，找到一个与当前要插入的key-val一致的key对象
        //k：临时的一个key
        Node<K,V> e; K k;
        //表示数组中的该元素，与你当前插入的元素key一致，后续会有替换操作
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            //判断key的条件是key的hash相同和eqauls方法符合，p.key等于插入 的key，将p的引用赋给e
            e = p;
        else if (p instanceof TreeNode)
            //p是红黑树节点，插入后仍然是红黑树节点，所以直接强制转型p后调用putTreeVal，返回的引用赋给e
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            //哈希碰撞,链表结构
            //循环，直到链表中的某个节点为null，或者某个节点hash值和给定的hash值一致且key也相同，则停止循环。
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    //next为空，将添加的元素置为next
                    p.next = newNode(hash, key, value, null);

                    ////插入成功后，要判断是否需要转换为红黑树，因为插入后链表长度+1>8,就转成红黑树，
                    //而binCount并不包含新节点，所以判断时要将临界阀值-1.【链表长度达到了阀值
                    //TREEIFY_THRESHOLD=8，即链表长度达到了7】
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        // 如果链表长度达到了8，且数组长度小于64，那么就重新散列resize()，如果大于64，则创建红黑树，将链表转换为红黑树
                        treeifyBin(tab, hash);
                    break;
                }
                //节点hash值和给定的hash值一致且key也相同，停止循环
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    //如果节点已存在,则跳出循环
                    break;
                //如果给定的hash值不同或者key不同。将next值赋给p，为下次循环做铺垫。即结束当前节点，对下一节点进行判断
                p = e;
            }
        }
        //如果e不是null，该元素存在了(也就是key相等)
        if (e != null) { // existing mapping for key
            // 取出该元素的值
            V oldValue = e.value;
            // 如果 onlyIfAbsent 是 true，就不用改变已有的值；如果是false(默认)，或者value是null，将新的值替换老的值
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            //什么都不做
            afterNodeAccess(e);
            //返回旧值
            return oldValue;
```

```
        }
    }
    //修改计数器+1，为迭代服务
    ++modCount;
    //达到了边界值，需要扩容
    if (++size > threshold)
        //超过阈值,进行扩容
        resize();
    //什么都不做
    afterNodeInsertion(evict);
    return null;
}
```

那接下来我们来看下是如何把节点添加到红黑树上的，调用的是putTreeVal()方法，源码如下：

```
final TreeNode<K,V> putTreeVal(HashMap<K,V> map, Node<K,V>[] tab,
                               int h, K k, V v) {
    Class<?> kc = null;
    boolean searched = false;
    //找到根节点
    TreeNode<K,V> root = (parent != null) ? root() : this;
    //遍历树节点元素
    for (TreeNode<K,V> p = root;;) {
        //节点位置,当前遍历到的节点hash值;key值
        int dir, ph; K pk;
        //如果树上元素的hash值大于添加进来元素的hash值
        if ((ph = p.hash) > h)
            //表示添加元素应在数的左节点
            dir = -1;
        else if (ph < h)
            // 表示添加元素应在树的右节点
            dir = 1;
        else if ((pk = p.key) == k || (k != null && k.equals(pk))) // 看key是否相同,是则代表找到要覆盖的节点位置
            //返回当前树节点,返回后会赋值给e,最终对value进行覆盖
            return p;

        else if ((kc == null &&
                  (kc = comparableClassFor(k)) == null) ||
                 (dir = compareComparables(kc, k, pk)) == 0) {  // 到此说明当前节点的hash值和指定key的hash值是相等的,但equals不等
            if (!searched) {  //如果还没有比对完成当前节点的所有子节点
                //继续遍历数进行寻找,如果还是没有找到key相同的,说明需要创建一个新节点
                TreeNode<K,V> q, ch;
                searched = true;
                if (((ch = p.left) != null &&
                     (q = ch.find(h, k, kc)) != null) ||
                    ((ch = p.right) != null &&
                     (q = ch.find(h, k, kc)) != null))
                    //找到就返回
                    return q;
            }
            // 最后的比较方法，调用System.identityHashCode()对k和要比较结点的key进行比较
            dir = tieBreakOrder(k, pk);
        }

```

```
        TreeNode<K,V> xp = p;
        // 根据方向dir决定
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            //是去左节点还是右节点,如果是null表示整棵树找完了,但还没有找到符合的节点,就要添加新节点了.
            // xpn作为新节点的next
            Node<K,V> xpn = xp.next;
            //创建新树节点
            TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
            //根据方向判断,新节点是在树左边还是右边
            if (dir <= 0)
                xp.left = x;
            else
                xp.right = x;
            //当前链表中的next节点指向到这个新的树节点
            xp.next = x;
            //新的树节点的父节点,前节点均设置为当前的树节点
            x.parent = x.prev = xp;
            //如果原来xp的next节点不为空
            if (xpn != null)
                //那么原来的next节点的前节点指向到新的树节点;
                ((TreeNode<K,V>)xpn).prev = x;
            // 平衡树,确保不会太深,确保树的根节点在数组上
            moveRootToFront(tab, balanceInsertion(root, x));
            return null;
        }
    }
}
```

在HashMap的红黑树中不是直接以key作为排序关键字来判断key的大小，而是以key的hash值作为排序的关键字来判断key的大小；当key的hash值相同时（hash 冲突），有2大类情况：
（1）key实现了Comparable接口，比较key大小，决定搜索分支；
（2）key没有实现Comparable接口，没法直接比较key大小，因此会搜索当前节点的左右分支；
putTreeVal()方法调用了find()方法从左右子树搜寻Key，find()源码实现如下：

```
/**
 * 从左右子树搜寻K
 * K 搜索目标
 * h 目标key的hash值
 * kc key的class对象
 */
final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
    //获取当前节点
    TreeNode<K,V> p = this;
    do {
        int ph, dir; K pk;
        //pl:当前节点的左子节点
        //pr :当前节点的右子节点
        TreeNode<K,V> pl = p.left, pr = p.right, q;
        //ph:当前节点的hash
        if ((ph = p.hash) > h)
            //case1 : 小于当前hash，继续在左子节点搜索
            p = pl;
        else if (ph < h)
            //case 2 : 大于当前hash，继续在右子节点中搜索
            p = pr;
        else if ((pk = p.key) == k || (k != null && k.equals(pk)))
```

```
            // case 3:等于当前hash值，并且(当前节点key值)pk == k(目标key)；直接返回当前节点
            return p;
        else if (pl == null) //该节点没有左子节点
            p = pr;
        else if (pr == null) //该节点没有右子节点
            p = pl;
        else if ((kc != null ||
                  (kc = comparableClassFor(k)) != null) &&
                 (dir = compareComparables(kc, k, pk)) != 0) //利用key的class类实现的比大小的方法，比较key的大小，然后决定查找的分支
            p = (dir < 0) ? pl : pr;
        else if ((q = pr.find(h, k, kc)) != null)
            //没有实现Comparable接口，或者实现了接口但是比较结果dir=0都会检测左右分支，
            // q = pr.find(h, k, kc)检查右分支；q,是右分支查询结果；q!=null在右分支中找到了目标key，
            return q;
        else
            //q==null,查询左分支；
            p = pl;
    } while (p != null);
    return null;
}
```

红黑树插入新节点后，会出现不平衡的情况，在putTreeVal中调用了balanceInsertion()方法平衡红黑树，关于红黑树的如何平衡的可参考前文。balanceInsertion()源码如下：

```
/**
 * 红黑树添加平衡
 * @param root
 * @param x
 * @param <K>
 * @param <V>
 * @return
 */
static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,
                                            TreeNode<K,V> x) {
    //新插入树节点默认红色
    x.red = true;
    //x是新插入节点、xp是新插入节点的父节点、xpp是新插入节点的祖父节点、
    //xppl是新插入节点的左叔叔节点、xppr是新插入节点的右叔叔节点
    for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {
        //1.空树
        if ((xp = x.parent) == null) {
            //新插入节点颜色变为黑色
            x.red = false;
            return x;
        }
        else if (!xp.red || (xpp = xp.parent) == null)
            //2.父节点黑色或祖父节点为空
            return root;
        if (xp == (xppl = xpp.left)) { //3.父节点红色  3.1父节点是祖父节点的左儿子
            if ((xppr = xpp.right) != null && xppr.red) { //3.1.1叔叔节点红色
                xppr.red = false;
```

```
                xp.red = false;
                xpp.red = true;
                x = xpp;
            }
            else { //3.1.2叔叔节点不存在
                if (x == xp.right) { //3.1.2.1新插入节点是父节点右儿子
                    //左旋
                    root = rotateLeft(root, x = xp);
                    xpp = (xp = x.parent) == null ? null : xp.parent;
                }
                //3.1.2.2新插入节点是父节点左儿子
                if (xp != null) {
                    xp.red = false;
                    if (xpp != null) {
                        xpp.red = true;
                        root = rotateRight(root, xpp);
                    }
                }
            }
        }
        else { //3.2父节点是祖父节点右儿子
            //3.2.1叔叔节点红色
            if (xppl != null && xppl.red) {
                xppl.red = false;
                xp.red = false;
                xpp.red = true;
                x = xpp;
            }
            else { //3.2.2叔叔节点不存在
                //3.2.2.1新插入节点是父节点左儿子
                if (x == xp.left) {
                    //右旋
                    root = rotateRight(root, x = xp);
                    xpp = (xp = x.parent) == null ? null : xp.parent;
                }
                if (xp != null) { //3.2.2.2新插入节点是父节点右儿子
                    xp.red = false;
                    if (xpp != null) {
                        xpp.red = true;
                        //左旋
                        root = rotateLeft(root, xpp);
                    }
                }
            }
        }
    }
}

/**
 * 左旋
 *
 * @param root 整个红黑树的根节点
 * @param p    旋转的根节点
 */
static <K, V> HashMap.TreeNode<K, V> rotateLeft(HashMap.TreeNode<K, V>
root, HashMap.TreeNode<K, V> p) {
    /**
     * 以节点P为根节点进行左旋
```

```
85          * 1、p的右节点指向r的左孩子（即rl），如果rl不为空，其父节点指向p;
86          * 2、r的父节点指向p的父节点（即PP），
87          *    2.1、如果pp为null,说明p节点为根节点，直接root指向r,同时颜色置为黑色（根节点颜色都为黑色）;
88          *    2.2、如果pp的右孩子为p,则将pp的右孩子指向r;
89          *    2.3、如果pp的左孩子为p,则将pp的左孩子指向r;
90          * 3、将r的左孩子指向p;
91          * 4、将p的父节点指向r;
92          */
93         // r-支点的右孩子节点，pp-支点的父节点，rl-支点右孩子的左节点
94         HashMap.TreeNode<K,V> r, pp, rl;
95         // 如果支点为NULL或者支点的右孩子节点为NULL，无法进行旋转，直接返回
96         if (p != null && (r = p.right) != null) {
97             if ((rl = p.right = r.left) != null)
98                 rl.parent = p;
99             if ((pp = r.parent = p.parent) == null)
100                (root = r).red = false;
101            else if (pp.left == p)
102                pp.left = r;
103            else
104                pp.right = r;
105            r.left = p;
106            p.parent = r;
107        }
108        // 返回树的根节点
109        return root;
110    }
111
112
113    /**
114     * 右旋
115     *
116     * @param root 整个红黑树的根节点
117     * @param p    旋转的根节点
118     */
119    static <K, V> HashMap.TreeNode<K, V> rotateRight(HashMap.TreeNode<K, V> root,
120                                                     HashMap.TreeNode<K, V> p) {
121        /**
122         * 以节点P为根节点进行左旋
123         * 1、p的左节点指向l的右孩子（即lr），如果lr不为空，其父节点指向p;
124         * 2、l的父节点指向p的父节点（即PP），
125         *    2.1、如果pp为null,说明p节点为根节点，直接root指向l,同时颜色置为黑色（根节点颜色都为黑色）;
126         *    2.2、如果pp的右孩子为p,则将pp的右孩子指向l;
127         *    2.3、如果pp的左孩子为p,则将pp的左孩子指向l;
128         * 3、将l的右孩子指向p;
129         * 4、将p的父节点指向l;
130         */
131        // l-支点的右孩子节点，pp-支点的父节点，lr-支点左孩子的右节点
132        HashMap.TreeNode<K, V> l, pp, lr;
133        // 如果支点为NULL或者支点的左孩子节点为NULL，无法进行旋转，直接返回
134        if (p != null && (l = p.left) != null) {
135            if ((lr = p.left = l.right) != null)
136                lr.parent = p;
137            if ((pp = l.parent = p.parent) == null)
138                (root = l).red = false;
```

```
            else if (pp.right == p)
                pp.right = l;
            else
                pp.left = l;
            l.right = p;
            p.parent = l;
        }
        // 返回树的根节点
        return root;
    }
```

看完红黑树节点的插入，接下来我们来看下hashMap是如何把链表转换成红黑树的， 核心方法是treeify()方法，调用treeify()方法的是treeifyBin()方法,当链表的长度超过8的时候，就会调用treeifyBin()方法链表转化为以树节点存在的双向链表。treeifyBin()源码如下：

```
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    //如果当前数组长度小于树化阈值
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        //将数组扩容为原来2倍
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) { //如果链表的头节点不为空
        //定义头节点、尾节点
        TreeNode<K,V> hd = null, tl = null;
        /**
         * 先将树节点全部用双向链表连接起来
         */
        do {
            //将链表节点转换为树节点
            TreeNode<K,V> p = replacementTreeNode(e, null);
            //将链表节点转换为树节点
            if (tl == null)
                //把p节点赋值给列表头
                hd = p;
            else {
                //新节点前一个结点设置为尾部
                p.prev = tl;
                //尾部下一个节点设置为新节点
                tl.next = p;
            }
            //p节点赋值给尾部tl
            tl = p;
        } while ((e = e.next) != null);
        //链表头节点放到到数组索引上
        if ((tab[index] = hd) != null)
            //树化
            hd.treeify(tab);
    }
}
```

treeify()将该双向链表转换为红黑树结构，源码如下：

```
final void treeify(Node<K,V>[] tab) {
    //树的根节点
    TreeNode<K,V> root = null;
```

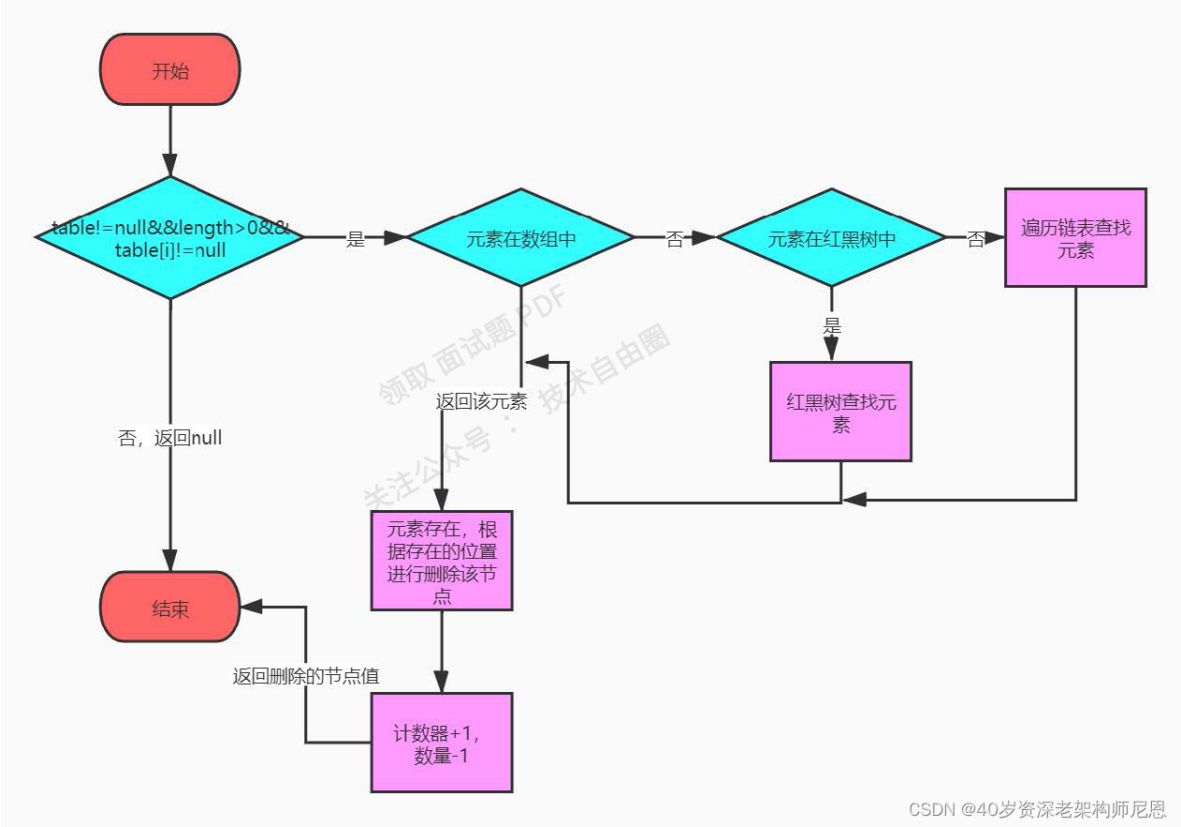
```
    //声明树节点x和next，先把当前节点赋值给x，开始循环
    for (TreeNode<K,V> x = this, next; x != null; x = next) {
        //next节点作为x的下一个节点
        next = (TreeNode<K,V>)x.next;
        //x左右孩子为空
        x.left = x.right = null;
        //如果根节点为空，x作为根节点
        if (root == null) {
            // 根节点无父节点
            x.parent = null;
            //节点颜色设置为黑色
            x.red = false;
            root = x;
        }
        /*
                *以下部分和putTreeVal()添加树节点元素代码是类似的，找到方向，然后进行插入
                * */
        else { // 除首次循环外其余均走这个分支
            // 除首次循环外其余均走这个分支
            K k = x.key;
            int h = x.hash;
            // 定义key的Class对象kc
            Class<?> kc = null;
            // 循环,每次循环从根节点开始,寻找位置
            for (TreeNode<K,V> p = root;;) {
                // 定义节点相对位置、节点p的hash值
                int dir, ph;
                // 获取节点p的key
                K pk = p.key;
                //如果root节点的hash值大于
                if ((ph = p.hash) > h)
                    // 当前节点在节点p的左子树
                    dir = -1;
                else if (ph < h)
                    // 当前节点在节点p的右子树
                    dir = 1;
                else if ((kc == null &&
                          (kc = comparableClassFor(k)) == null) ||
                         (dir = compareComparables(kc, k, pk)) == 0)
                    // 当前节点与节点p的hash值相等,当前节点key并没有实现Comparable接口
                    // 或者实现Comparable接口并且与节点pcompareTo相等,该方法是为了保证在特殊情况下节点添加的一致性用于维持红黑树的平衡
                    dir = tieBreakOrder(k, pk);

                TreeNode<K,V> xp = p;
                // 根据dir判断添加位置也是节点p的左右节点,是否为空,若不为null在p的子树上进行下次循环
                if ((p = (dir <= 0) ? p.left : p.right) == null) {
                    // 若添加位置为null,建立当前节点x与父节点xp之间的联系
                    x.parent = xp;
                    // 确定当前节点时xp的左节点还是右节点
                    if (dir <= 0)
                        xp.left = x;
                    else
                        xp.right = x;
                    // 对红黑是进行平衡操作并结束循环
```

```
                    root = balanceInsertion(root, x);
                    break;
                }
            }
        }
    }
    // 将红黑树根节点复位至数组头结点
    moveRootToFront(tab, root);
}

```

以上代码就是hashmap 基于数组+链表+红黑树实现的Key-Value键值对的存储，最后用一种流程图总结一下put()方法：

remove() 删除Key-Value

接下来我们来看下删除key-value键值对。hashMap的删除方法是remove(Object key)方法，执行流程如下：



源码如下

```
public V remove(Object key) {
    Node<K,V> e;
    return (e = removeNode(hash(key), key, null, false, true)) == null ?
        null : e.value;
}

/**
 * @param hash       key对应的hash值
 * @param key        key
 * @param value      key对应的值
 * @param matchValue 是否需要对值进行匹配操作
 * @param movable    是否将根节点移动到链表顶端
```

```
 */
final Node<K,V> removeNode(int hash, Object key, Object value,
                           boolean matchValue, boolean movable) {
    Node<K,V>[] tab; Node<K,V> p; int n, index;
    //数组不为null，数组长度大于0，要删除的元素计算的槽位有元素
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (p = tab[index = (n - 1) & hash]) != null) {
        Node<K,V> node = null, e; K k; V v;
        //当前元素在数组中
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;
        else if ((e = p.next) != null) { //在链表中
            if (p instanceof TreeNode) //元素在红黑树或链表中
                node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
            else {
                do {
                    //hash相同，并且key相同，找到节点并结束
                    if (e.hash == hash &&
                        ((k = e.key) == key ||
                         (key != null && key.equals(k)))) {
                        node = e;
                        break;
                    }
                    p = e;
                } while ((e = e.next) != null);
            }
        }
        if (node != null && (!matchValue || (v = node.value) == value ||
                             (value != null && value.equals(v)))) { //找到节点了，并且值也相同

            if (node instanceof TreeNode) //是树节点，从树中移除
                ((TreeNode<K,V>)node).removeTreeNode(this, tab, movable);
            else if (node == p) //节点在数组中，
                tab[index] = node.next;
            else //节点在链表中
                p.next = node.next; //将节点删除
            //修改计数器+1，为迭代服务
            ++modCount;
            --size;
            //什么都不做
            afterNodeRemoval(node);
            //返回删除的节点
            return node;
        }
    }
    return null;
}
```

如果节点在红黑树中，就需要到树中进行删除，调用removeTreeNode()方法删除，源码如下：

```
/**
 * 这个方法是HashMap.TreeNode的内部方法，调用该方法的节点为待删除节点
 *
 * @param map     删除操作的map
 * @param tab     map存放数据的链表
```

```
 * @param movable 是否移动跟节点到头节点
 */
final void removeTreeNode(HashMap<K, V> map, HashMap.Node<K, V>[] tab,
boolean movable) {
    int n;
    if (tab == null || (n = tab.length) == 0)
        return;
    // 获取索引值
    int index = (n - 1) & hash;
    /**
     * first-头节点，数组存放数据索引位置存在存放的节点值
     * root-根节点,红黑树的根节点，正常情况下二者是相等的
     * rl-root节点的左孩子节点,succ-后节点,pred-前节点
     */
    HashMap.TreeNode<K, V> first = (HashMap.TreeNode<K, V>) tab[index],
root = first, rl;
    // succ-调用这个方法的节点（待删除节点）的后驱节点，prev-调用这个方法的节点（待删
除节点）的前驱节点
    HashMap.TreeNode<K, V> succ = (HashMap.TreeNode<K, V>) next, pred =
prev;
    /**
     * 维护双向链表（map在红黑树数据存储的过程中，除了维护红黑树之外还对双向链表进行了
维护）
     * 从链表中将该节点删除
     * 如果前驱节点为空，说明删除节点是头节点，删除之后，头节点直接指向了删除节点的后继
节点
     */
    if (pred == null)
        tab[index] = first = succ;
    else
        pred.next = succ;
    if (succ != null)
        succ.prev = pred;
    // 如果头节点（即根节点）为空，说明该节点删除后，红黑树为空，直接返回
    if (first == null)
        return;
    // 如果父节点不为空，说明删除后，调用root方法重新获取当前树的根节点
    if (root.parent != null)
        root = root.root();
    /**
     * 当以下三个条件任一满足时，当满足红黑树条件时，说明该位置元素的长度少于
6（UNTREEIFY_THRESHOLD），需要对该位置元素链表化
     * 1、root == null：根节点为空，树节点数量为0
     * 2、root.right == null：右孩子为空，树节点数量最多为2
     * 3、(rl = root.left) == null || rl.left == null):
     *      (rl = root.left) == null：左孩子为空，树节点数最多为2
     *      rl.left == null：左孩子的左孩子为NULL，树节点数最多为6
     */
    if (root == null || root.right == null ||
            (rl = root.left) == null || rl.left == null) {
        // 链表化，因为前面对链表节点完成了删除操作，故在这里完成之后直接返回，即可完成
节点的删除
        tab[index] = first.untreeify(map);
        return;
    }
    /**
     * p-调用此方法的节点（待删除节点），pl-待删除节点的左子节点，pr-待删除节点的右子
节点，replacement-替换节点
```

```
     * 以下是对红黑树进行维护
     */
    HashMap.TreeNode<K, V> p = this, pl = left, pr = right, replacement;
    // 1、删除节点有两个子节点
    if (pl != null && pr != null) {
        // 第一步：找到当前节点的后继节点（注意与后驱节点的区别，值大于当前节点值的最小节点，以右子树为根节点，查找它对应的最左节点）
        HashMap.TreeNode<K, V> s = pr, sl;
        // 循环右子树中查找后继节点(大于当前节点的最小值)
        while ((sl = s.left) != null) // find successor
            s = sl;
        // 第二步：交换后继节点和删除节点的颜色，最终的删除是删除后继节点，故平衡是否是以后继节点的颜色来判断的
        boolean c = s.red;
        s.red = p.red;
        p.red = c; // swap colors
        // sr-后继节点的右孩子（后继节点是肯定不存在左孩子的，如果存在的话，那么它肯定不是后继节点）
        HashMap.TreeNode<K, V> sr = s.right;
        // pp-待删除节点的父节点
        HashMap.TreeNode<K, V> pp = p.parent;
        // 第三步：修改当前节点和后继节点的父节点
        // 如果后继节点与当前节点的右孩子相等，类似于当前节点只有一个右孩子
        if (s == pr) { // p was s's direct parent
            // 交换两个节点的位置，父节点变子节点，子节点变父节点
            p.parent = s;
            s.right = p;
        } else {
            // 如果当前节点的右子树不止一个节点，记录sp-后继节点的父节点
            HashMap.TreeNode<K, V> sp = s.parent;
            // 交换待删除节点和后继节点的的父节点，如果后继节点父节点不为null，指定后继节点父节点的孩子节点
            if ((p.parent = sp) != null) {
                // 如果前后节点是其父节点的左孩子，修改父节点左孩子值
                if (s == sp.left)
                    sp.left = p;
                    // 如果后继节点是其父节点的右孩子，修改父节点右孩子值
                else
                    sp.right = p;
            }
            // 修改后继节点的右孩子值，如果不为null，同时指定其父节点的值
            if ((s.right = pr) != null)
                pr.parent = s;
        }
        // 第四步：修改当前节点和后继节点的孩子节点，当前节点现在变成后继节点了，故其左孩子为null.
        p.left = null;
        // 修改当前节点的右孩子值，如果其不为空，同时修改其父节点指向当前节点
        if ((p.right = sr) != null)
            sr.parent = p;
        // 修改后继节点的左孩子值，如果其不为空，同时修改其父节点指向后继节点
        if ((s.left = pl) != null)
            pl.parent = s;
        // 修改后继节点的父节点值，如果其为null，说明后继节点现在变成了root节点
        if ((s.parent = pp) == null)
            root = s;
            // 当前节点是其父节点的左孩子
        else if (p == pp.left)
```

```
108             pp.left = s;
109             // 当前节点是其父节点的右孩子
110         else
111             pp.right = s;
112         /**
113          * sr-后继节点的右孩子节点（有一个孩子节点），
114          * 如果右孩子节点不为空，删除节点后，替代节点就是其右孩子节点
115          * 如果为空，那么替代节点就是其本身
116          */
117         if (sr != null)
118             replacement = sr;
119         else
120             replacement = p;
121         // 2、删除节点有一个左子节点，左子节点作为替代节点
122     } else if (pl != null)
123         replacement = pl;
124         // 3、删除节点有一个右子节点，右子节点作为替代节点
125     else if (pr != null)
126         replacement = pr;
127         // 4、删除节点没有子节点，直接删除当前节点
128     else
129         replacement = p;
130     /**
131      * 如果删除节点存在两个孩子节点，最终与后继节点交换后，删除的节点的位置位于后继节点的位置，那么此时删除节点所处的位置演变成：
132      * a、只有一个孩子节点：(replacement = p.right) != p
133      * b、没有孩子节点：replacement == p
134      * 只有当删除节点与替换节点不相等的时候，才对删除节点进行删除操作
135      */
136     if (replacement != p) {
137         // 从红黑树中将待删除节点（即当前节点移除）
138         HashMap.TreeNode<K, V> pp = replacement.parent = p.parent;
139         // 是否为根节点
140         if (pp == null)
141             root = replacement;
142             // 其父节点的左子节点
143         else if (p == pp.left)
144             pp.left = replacement;
145             // 其父节点的右子节点
146         else
147             pp.right = replacement;
148         // 节点的指向全部置NULL
149         p.left = p.right = p.parent = null;
150     }
151     /**
152      * 如果删除节点的颜色是红色，不会影响整棵树的黑色高度，毋需自平衡，根节点不会变化，如果是黑色，则需要进行自平衡，重新获取根节点
153      * 注意：
154      * 自平衡的时候 替代节点可能与删除节点相等：replacement == p
155      * 自平衡的时候 替代节点可能与删除节点不相等：replacement ! = p
156      */
157     HashMap.TreeNode<K, V> r = p.red ? root : balanceDeletion(root, replacement);
158     /**
159      * 当 replacement == p 时，是先进行了红黑树的进行了平衡操作，再将这个节点从红黑树中移除
160      * 这个地方我也没明白原理是什么，但是我按照这个步骤去走了一遍，确实这样操作来完成平衡，如果有哪位大神明白的，麻烦指导一下，谢谢！
```

```
161         */
162         if (replacement == p) {  // detach
163             // pp-存储当前节点的父节点值
164             HashMap.TreeNode<K, V> pp = p.parent;
165             // 当前节点的父节点指向NULL
166             p.parent = null;
167             // 如果父节点不为空，根据当前节点位于父节点的不同子节点，修改父节点的孩子节点值
168             if (pp != null) {
169                 if (p == pp.left)
170                     pp.left = null;
171                 else if (p == pp.right)
172                     pp.right = null;
173             }
174         }
175         // movable为true，需要将根节点移动到头节点，即数组所以位置指向的节点
176         if (movable)
177             moveRootToFront(tab, r);
178     }
179     /**
180      * 红黑树删除节点后，平衡红黑树的方法
181      *
182      * @param root 根节点
183      * @param x     节点删除后，替代其位置的节点，这个节点可能是一个节点，也可能是一棵平衡的红黑树，在此处就当作一个节点，在该节点以上部分需要自平衡
184      * @return 返回新的根节点
185      */
186     static <K, V> HashMap.TreeNode<K, V> balanceDeletion(HashMap.TreeNode<K, V> root, HashMap.TreeNode<K, V> x) {
187         /**
188          * 进入这个方法，说明被替代的节点之前是黑色的，如果是红色的不会影响黑色高度，黑色的会影响以其作为根节点子树的黑色高度
189          * xp-父节点,xpl-父节点的左孩子,xpr-父节点的右孩子节点
190          * 注意：
191          *     进入该方法的时候  替代节点可能与删除节点相等：x == replacement == p
192          *                      替代节点可能与删除节点不相等：x == replacement ! = p
193          */
194         for (HashMap.TreeNode<K, V> xp, xpl, xpr; ; ) {
195             /**
196              * 1、x == null，当 replacement == p 时，删除节点不存在，返回；
197              *       因为当 replacement ! = p 时，replacement 肯定不会为null.在移除节点的方法中有三个地方对 replacement 进行赋值。
198              *           1、if (sr != null) replacement = sr;
199              *           2、if (pl != null) replacement = pl;
200              *           3、if (pr != null) replacement = pr;
201              * 2、x == root，如果替代完成后，该节点就是整棵红黑树的根节点，本身就是平衡的，直接返回
202              */
203             if (x == null || x == root)
204                 return root;
205             else if ((xp = x.parent) == null) {
206                 // 如果父节点为空，说明当前节点就是根节点，设置根节点的颜色为黑色，返回
207                 x.red = false;
208                 return x;
209             } else if (x.red) {
210                 /**
211                  * 被替换节点（删除节点）的颜色是黑色的，删除之后黑色高度减1，如果替换节点是红色，将其设置为黑色，可以保证
212                  * 1、与替换之前的黑色高度相等
```

```
                 * 2、满足红黑树的所有特性
                 * 达到平衡返回
                 */
                x.red = false;
                return root;
                /**
                 * 如果替换节点是黑色的，替换之前的节点也是黑色的，替换之后，以替换节点作为根节点子树黑色高度减少1，需要进行相关的自平衡操作
                 * 1、替换节点是父节点的左孩子
                 */

                // 前提是X为黑色，左侧分支
            } else if ((xpl = xp.left) == x) {
                /**
                 * 情况1、父节点的右孩子（兄弟节点）存在且为红色
                 * 处理方式：兄弟节点变黑，父节点变红，以父节点为支点进行左旋，重新获取兄弟节点，继续参与自平衡
                 */
                if ((xpr = xp.right) != null && xpr.red) {
                    xpr.red = false;
                    xp.red = true;
                    root = rotateLeft(root, xp);
                    // 重新获取XPR
                    xpr = (xp = x.parent) == null ? null : xp.right;
                }

                // 不存在兄弟节点，x指向父节点，向上调整
                if (xpr == null)
                    x = xp;
                else {
                    // sl-兄弟节点的左孩子，sr-兄弟节点的右孩子
                    HashMap.TreeNode<K, V> sl = xpr.left, sr = xpr.right;
                    /**
                     * 情况2-1：兄弟节点存在，且两个孩子的颜色均为黑色
                     * 1、sr == null || !sr.red：兄弟的右孩子为黑色（空节点的颜色其实也是黑色）
                     * 2、sl == null || !sl.red：兄弟的左孩子为黑色（空节点的颜色其实也是黑色）
                     * 处理方式：兄弟节点为红色，替换节点指向父节点，继续参与自平衡
                     */
                    if ((sr == null || !sr.red) && (sl == null || !sl.red)) {
                        xpr.red = true;
                        x = xp;
                    } else {
                        /**
                         * 该条件综合评价为：兄弟节点的右孩子为黑色
                         * 1、sr == null：兄弟的右孩子为黑色（空节点的颜色其实也是黑色）
                         * 2、!sr.red：兄弟节点的右孩子颜色为黑色
                         */
                        if (sr == null || !sr.red) {
                            /**
                             * sl != null：兄弟的左孩子是存在且颜色是红色的
                             * 情况2-2、兄弟节点右孩子为黑色、左孩子为红色
                             * 处理方式：兄弟节点的左孩子设为黑色，兄弟节点设为红色，以兄弟节点为支点进行右旋，重新设置x的兄弟节点，继续参与自平衡
                             */
                            if (sl != null)
```

```
                            sl.red = false;
                        xpr.red = true;
                        root = rotateRight(root, xpr);
                        xpr = (xp = x.parent) == null ? null : xp.right;
                    }
                    /**
                     * 情况2-3、兄弟节点的右孩子是红色
                     * 处理方式：
                     * 1、如果兄弟节点存在，兄弟节点的颜色设置为父节点的颜色
                     * 2、兄弟节点的右孩子存在，颜色设为黑色
                     * 3、如果父节点存在，将父节点的颜色设为黑色
                     * 4、以父节点为支点进行左旋
                     */
                    if (xpr != null) {
                        xpr.red = (xp == null) ? false : xp.red;
                        if ((sr = xpr.right) != null)
                            sr.red = false;
                    }
                    // 父节点不为空
                    if (xp != null) {
                        xp.red = false;
                        root = rotateLeft(root, xp);
                    }
                    // 替换节点指向根节点，平衡完成
                    x = root;
                }
            }
        } else {
            // X为黑色 右侧分支
            /**
             * 替换节点是父节点的右孩子节点
             * 情况1、兄弟节点存在且为红色
             * 处理方式：兄弟节点变黑，父节点变红，以父节点为支点进行左旋，重新获取兄弟节点，继续参与自平衡
             */
            if (xpl != null && xpl.red) {
                xpl.red = false;
                xp.red = true;
                root = rotateRight(root, xp);
                xpl = (xp = x.parent) == null ? null : xp.left;
            }
            // 不存在兄弟节点，x指向父节点，向上调整
            if (xpl == null)
                x = xp;
            else {
                // sl-兄弟节点的左孩子，sr-兄弟节点的右孩子
                HashMap.TreeNode<K, V> sl = xpl.left, sr = xpl.right;
                /**
                 * 情况2-1：兄弟节点存在，且两个孩子的颜色均为黑色
                 * 1、sr == null || !sr.red：兄弟的右孩子为黑色（空节点的颜色其实也是黑色）
                 * 2、sl == null || !sl.red：兄弟的左孩子为黑色（空节点的颜色其实也是黑色）
                 * 处理方式：兄弟节点为红色，替换节点指向父节点，继续参与自平衡
                 */
                if ((sl == null || !sl.red) && (sr == null || !sr.red)) {
                    xpl.red = true;
                    x = xp;
```

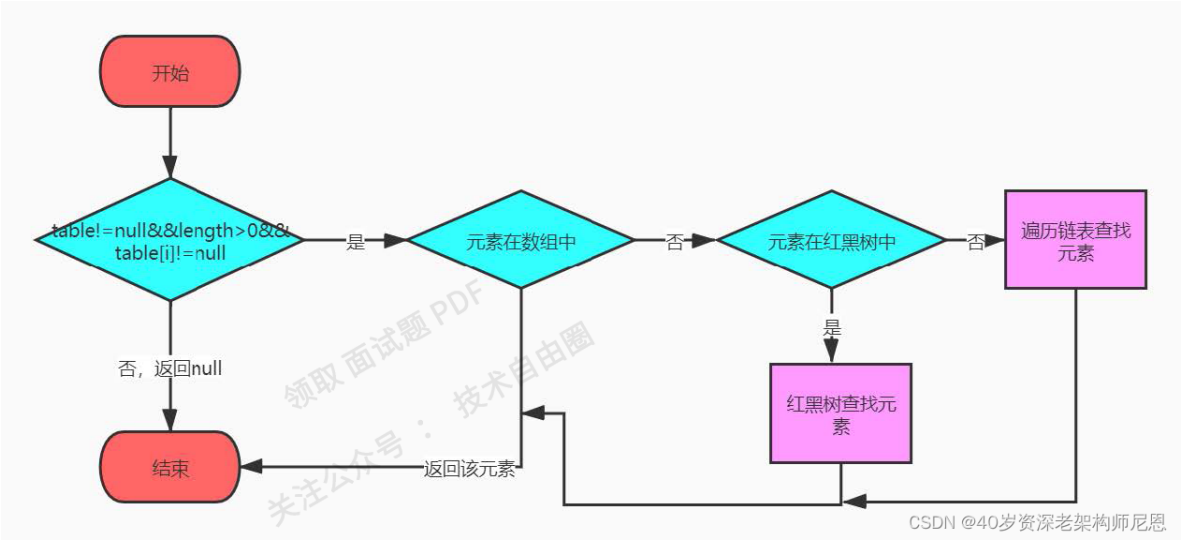
```
                } else {
                    /**
                     * 该条件综合评价为：兄弟节点的左孩子为黑色
                     * 1、sr == null：兄弟的左孩子为黑色（空节点的颜色其实也是黑色）
                     * 2、!sr.red：兄弟节点的左孩子颜色为黑色
                     */
                    if (sl == null || !sl.red) {
                        /**
                         * sl != null：兄弟的右孩子是存在且颜色是红色的
                         * 情况2-2、兄弟节点左孩子为黑色、右孩子为红色
                         * 处理方式：兄弟节点的右孩子设为黑色，兄弟节点设为红色，以兄弟节点为支点进行左，重新设置x的兄弟节点，继续参与自平衡
                         */
                        if (sr != null)
                            sr.red = false;
                        xpl.red = true;
                        root = rotateLeft(root, xpl);
                        xpl = (xp = x.parent) == null ? null : xp.left;
                    }
                    /**
                     * 情况2-3、兄弟节点的左孩子是红色
                     * 处理方式：
                     * 1、如果兄弟节点存在，兄弟节点的颜色设置为父节点的颜色
                     * 2、兄弟节点的左孩子存在，颜色设为黑色
                     * 3、如果父节点存在，将父节点的颜色设为黑色
                     * 4、以父节点为支点进行右旋
                     */
                    if (xpl != null) {
                        xpl.red = (xp == null) ? false : xp.red;
                        if ((sl = xpl.left) != null)
                            sl.red = false;
                    }
                    if (xp != null) {
                        xp.red = false;
                        root = rotateRight(root, xp);
                    }
                    // 替换节点指向根节点，平衡完成
                    x = root;
                }
            }
        }
    }
}
```

get(Object key)方法

当 HashMap 只存在数组，而数组中没有Node链表时，是HashMap查询数据性能最好的时候。
一旦发生大量的哈希冲突，就会产生 Node 链表，这个时候每次查询元素都可能遍历 Node 链表，从而降低查询数据的性能。

特别是在链表长度过长的情况下，性能明显下降，使用红黑树就很好地解决了这个问题，红黑树使得查询的平均复杂度降低到了O(log(n))，链表越长，使用红黑树替换后的查询效率提升就越明显。

get(Object key)方法执行流程如下：



get(Object key)源码如下：

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


final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    //数组不为null，数组长度大于0，根据hash计算出来的槽位的元素不为null
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        //查找的元素在数组中，返回该元素
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) { //查找的元素在链表或红黑树中
            if (first instanceof TreeNode) //在红黑树中查找
               //遍历链表，元素在链表中，返回该元素
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                //遍历链表，元素在链表中，返回该元素
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

关于红黑树

HashMap使用了数组+链表+红黑树三种数据结构相互结合的形式存储键值对，提升了查询键值对的效率。

关于红黑树的知识，非常重要，尼恩专门写了一篇长长的文章，也是结合面试题写的，作为尼恩Java面试宝典的 专题33.

该PDF的名字为：

《尼恩Java面试宝典专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备》

很多小伙伴评价，通过此文，终于搞懂了红黑树。 建议大家去看看。

最后总结一下本文：

本文通过首先hashmap的学习，重点是要学会hashMap的思想，在实际开发中如何去引用hashMap的思路去解决问题，如何更好的使用HashMap，优化HashMap的性能。

尼恩提示：要想拿高薪，首先hashmap、手写 线程池，都是必须课哈。

作者介绍：

本文1作： 唐欢，资深架构师， 《Java 高并发核心编程 加强版》作者之1 。

本文2作： 尼恩，40岁资深老架构师， 《Java 高并发核心编程 加强版 卷1、卷2、卷3》创世作者，著名博主 。《K8S学习圣经》《Docker学习圣经》等11个PDF 圣经的作者。

参考文献：

清华大学出版社《Java高并发核心编程 卷2 加强版》

《尼恩Java面试宝典专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备》

https://blog.csdn.net/longsq602/article/details/114165028

https://www.jianshu.com/p/d7024b52858c

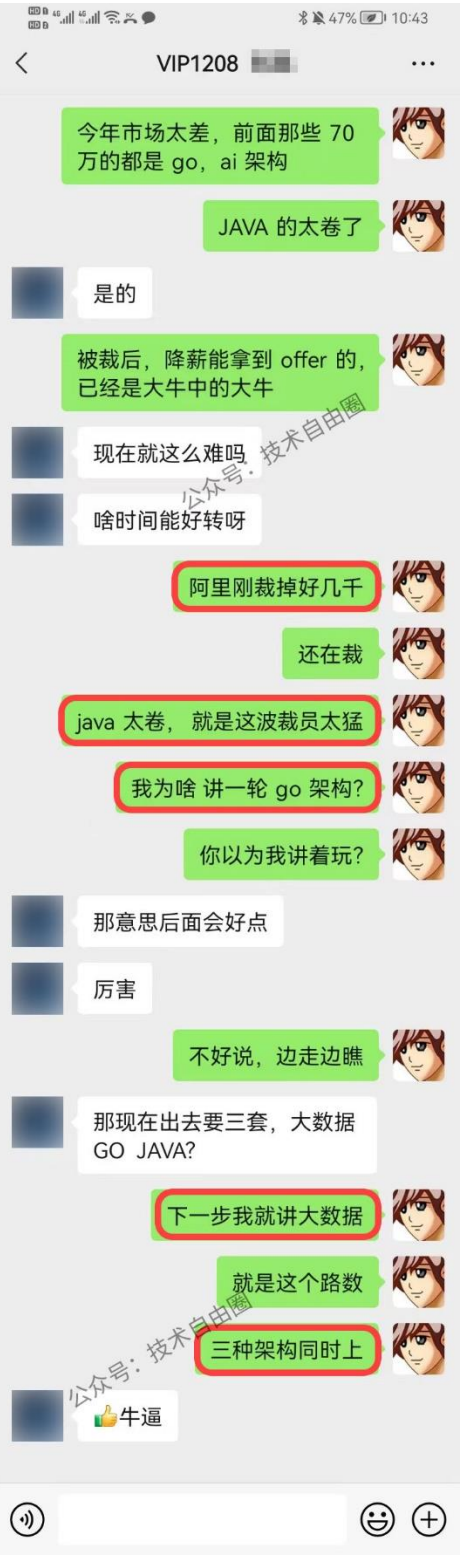
https://juejin.cn/post/6844903877188272142

https://blog.csdn.net/qq_50227688/article/details/114301326

https://blog.csdn.net/qq116165600/article/details/103361385

https://blog.csdn.net/qq_30757161/article/details/103580299

未来职业，如何突围：三栖架构师



成功案例：2年翻3倍，35岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新　最后发表　热门　精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200%
  卷王1号　超级版主　前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W
  卷王1号　超级版主　2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W
  卷王1号　超级版主　2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W
  卷王1号　超级版主　2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30%
  卷王1号　超级版主　前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25%
  卷王1号　超级版主　前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研
  卷王1号　超级版主　前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W
  卷王1号　超级版主　2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100%
  卷王1号　超级版主　2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K
  卷王1号　超级版主　2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30%
  卷王1号　超级版主　2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺
  卷王1号　超级版主　2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+
  卷王1号　超级版主　2022-7-5

成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120%

卷王1号 超级版主 2022-8-13

成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭

卷王1号 超级版主 2022-7-6

成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺

卷王1号 超级版主 2022-5-30

成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-12

成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-13

成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60%

卷王1号 超级版主 2022-5-11

成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家

卷王1号 超级版主 2022-4-21

成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W

卷王1号 超级版主 2022-4-16

成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万

卷王1号 超级版主 2022-4-14

成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer

卷王1号 超级版主 2022-4-12

成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer

卷王1号 超级版主 2022-4-7

成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer

卷王1号 超级版主 2022-3-25

成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer

卷王1号 超级版主 2022-3-24

成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer

卷王1号 超级版主 2022-3-21

成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+

卷王1号 超级版主 2022-3-20

成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer

卷王1号 超级版主 2022-3-16

简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请

卷王1号 超级版主 2022-3-10

成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面

卷王1号 超级版主 2022-3-10

成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer

卷王1号 超级版主 2022-3-10

成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer

尼恩 超级版主 2022-3-5

成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer

卷王1号 超级版主 2022-2-27

成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到字节，年薪45W

卷王1号 超级版主 2022-2-27

成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer

卷王1号 超级版主 2022-2-27

成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer

卷王1号 超级版主 2022-2-27

成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+

卷王1号 超级版主 2022-2-27

成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer

卷王1号 超级版主 2022-2-27

成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端

卷王1号 超级版主 2022-2-27

成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W

卷王1号 超级版主 2022-2-27

成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer, 年薪33W

卷王1号 超级版主 2022-2-27

成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习

卷王1号 超级版主 2022-2-10

成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪

卷王1号 超级版主 2022-2-10

成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer

卷王1号 超级版主 2022-2-10

成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer

卷王1号 超级版主 2022-2-10

[612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈

卷王1号 超级版主 2022-2-10

成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer, 年薪35W

卷王1号 超级版主 2022-2-27

成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer

卷王1号 超级版主 2022-2-27

成功案例：[569号卷王] 双非应届生卷王, 喜收字节跳动实习offer

卷王1号 超级版主 2022-2-25

成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元!

卷王1号 超级版主 2022-2-25

成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K

卷王1号 超级版主 2022-2-10

硬核推荐：尼恩Java硬核架构班

详情：https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

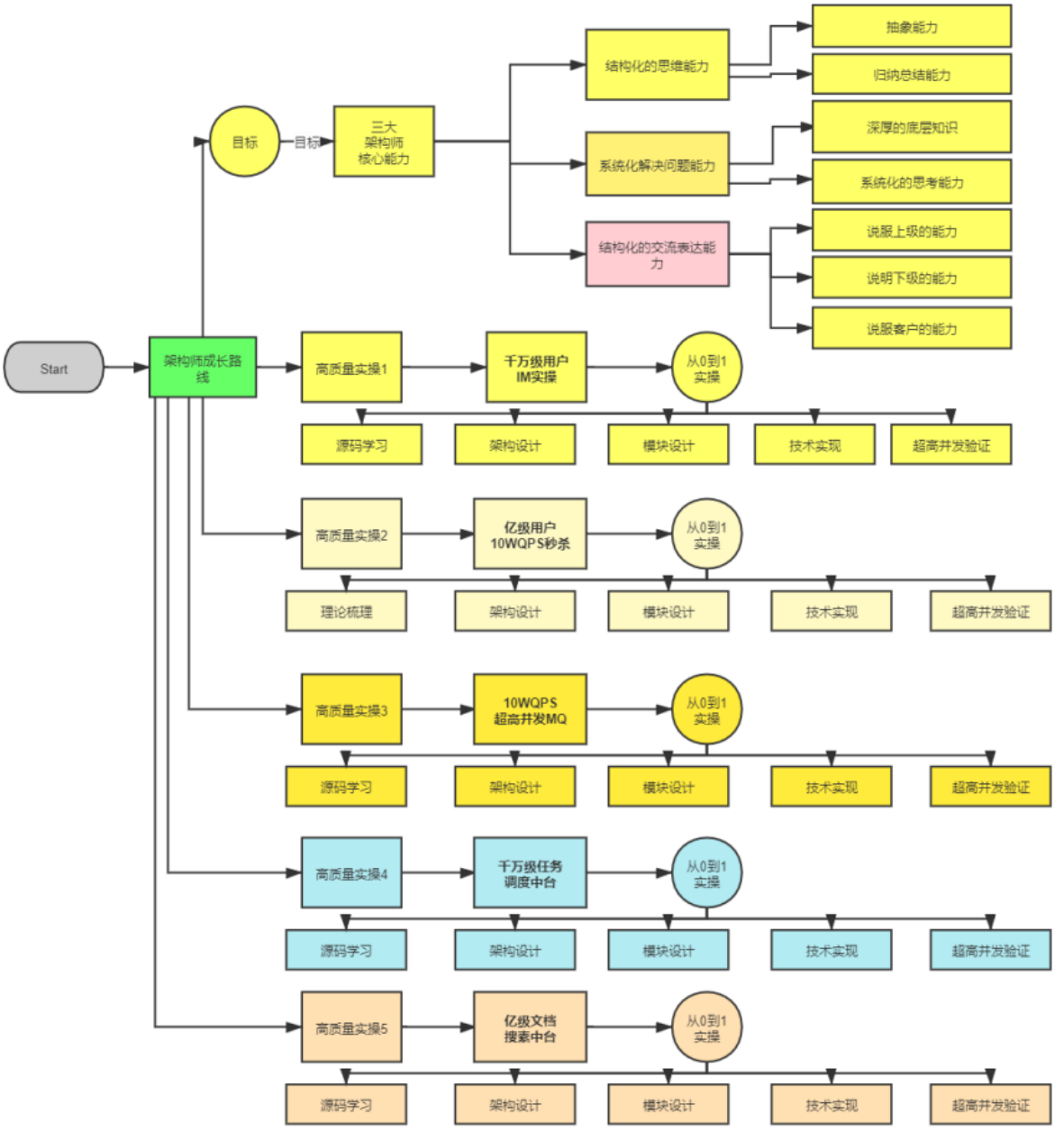
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章：横扫全网 Rocketmq 视频第 2 部曲：工业级 rocketmq 高可用（HA）底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

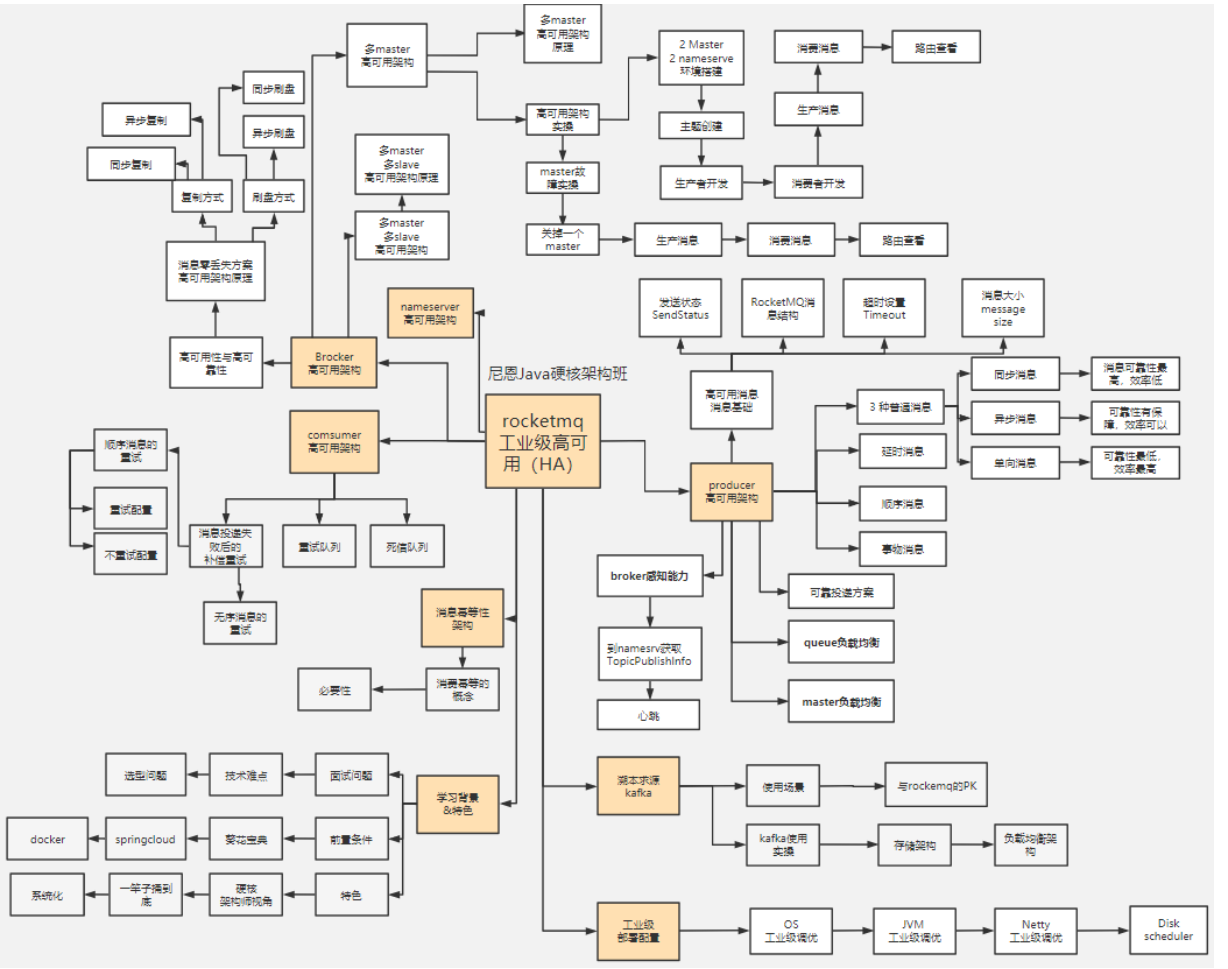
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



简历优化后的成功涨薪案例（VIP含免费简历优化）


6年小伙收60W年薪
一月速提3大offer
4.24号改简历
5.27号报喜
VIP1239 6年
4月24日 晚上19:10
预期的岗位：60W
预期的岗位：go 和 java 的后端开
4月24日 晚上19:56
秘诀：
简历指导+ 狠狠卷
4月24日 晚上20:50
6年经验 1239
年薪 60W.
21.7 KB
微信电脑版
前几天改 ai，今天改 go
谢谢，我根据这个模式，再整理一下我简历，到时候老师再帮我把把关
ok
4月27日 下午14:37
老师，再帮我看看
准备开始投简历了
简历-6 年后端开发.pdf
210.1 KB
PDF
VIP1239 6年
嗯嗯，我先对着简历准备些东西，然后再开始投吧，反正现在刚刚好五一放假
上午8:01
来还愿啥，3周斩获3个offer，准备入职了
上午8:05
您太牛啦
简历优化后，面试机会太多了，拿完3个offer后，还有许多公司在流程中的都拒绝了
这几天大厂动作不断，联想，阿里都在裁员，您太牛啦
还是老师你强，offer中也达到了预期60w
非常不错
现在都上岸有offer就行，薪酬还能达到预期，已经超级牛掰啦
很多人，连一个面试电话都没有，崩溃的一塌糊涂
抖音上到处是这种
主要是简历优化后，感觉如有神助，每天基本3个面试，除了字节一面没过，其它都通过了
恭喜您
谢谢老师
后续有啥问题，可以找我支撑哈
好嘞，学习圈一直在，要持续提高自己
好的，撸起袖子加油卷，搞技术前途无限好
被裁后转架构，逆涨50%
8年小伙喜提年薪75W
4.16号改简历
5.6号报喜
VIP1236
最近面试了几个一轮游
捞了太多人上岸了
都是你这号
捞我
助力我 一个月时间
绞杀 下钻 打破瓶颈
咱们开始不？
好
4月16日 下午15:04
预期岗位：高级开发、架构
4月16日 下午15:12
预期薪酬：60W
-8 年高级服务端-0404 - 副本(2).
30.2 KB
微信电脑版
4月16日 下午16:19
秘诀：
简历指导+ 狠狠卷
4月16日 下午19:21
通话时长 03:02:25
辛苦老师
VIP1236
上午11:08
尼架，我决定要去上海了。
拿了几个 offer 了？
两个
上海这个是架构师对吧？还有一个呢？
还有个广州高级 Java，待遇 40w 左右，老板比较喜欢我，开了很多绿灯，薪资可以再加，但我还是想闯闯，昨天拒绝了。
年包多少呢？
就之前说的
那都快 80 个 W 了
我 argue 了下，他们控制内部薪酬平衡有点难办到 80，但已经是标出来的上限了。
都快是 高级 java 的 两倍
你撤回了一条消息
75 w 也非常多了，在现在的环境下
除开税，差不了多少，主要是这个方向的潜在价值
关键对你来说，这个是一个 成长机会
是的，寒意还是有的
您是我的贵人
是呢 有个外接大脑就是爽
好好卷，先祝贺您，拿到年薪 75W +
再预祝您，2 年之后，年薪 200W+
谢谢


9年 小伙伴拿到
年薪90W offer
9月11日改简历
11月29日晒offer
秘诀:
简历指导+ 狠狠卷
这个是给你微调的
省略号的地方，需要你再补充一点
上面你留着这个就行
Java 开发 9年-修改.docx
34.1 KB
薪资高 稳
主要的工作是啥?
提升了自己的实力，就不用怕
易所 关于数字货 的 易
po 也有 打盹的时候，该裁员，照样一个不少
年包比 po 多 19w
这么多
po 估计有 70 万
那你不有 90 个 W?
po 给我 68
就是吗


小伙8年经验
年薪60w
7月12日改简历
8月10日晒offer
秘诀:
改简历 + 狠狠卷
明天晚上哈
好哒
恩哥，今晚还改简历吗?
今晚还在外边应酬，估计回去比较晚
要不，咱们延迟到明天，如何
明天白天也行
好的，白天吧，答应别人明天给他们简历了。
OK
那就上午 11 点左右哈
好的
涨幅多少呀?
或者，额度多少呀
之前 36*15，现在这个 39*15
今年行情不太好，还有一些 offer 基本都是平薪，或降薪的。
OK
这个马上来
刚在指导简历
哈哈
稍等哈
辛苦啊恩哥
这次找工作，您的 起到效果了。
我这次复习基本看的都是咱们课程的 。


6年小伙伴
年薪40w
9月6日改简历
9月21日晒offer
秘诀:
简历指导+ 狠狠卷
Java - 6年.docx
24.7 KB
恩哥，简历我改好了，您再帮我看一下
Java - 6年(1).docx
24.5 KB
恩哥，看第二个吧
给你前面调整了一下
涨这多
今年这行情，也算可以了
总包多少呀，让我也陶醉一下
大概 40w 吧
谢谢恩哥的指导和鼓励
是在深圳
深圳的行情尤其难
能有面试电话就不错啦
是的
太牛啦
哈哈哈哈，恩哥的鼓励指导也很重要


5年小伙喜提3个offer
年薪 35个W
5月22日改简历
11月29日晒offer
我简历还没写出来 不然可以给老师您看看
算法方面，可以找点 刷刷，当作复习
项目的话，再好好改造一下
好的 谢谢老师
java 开发-本科统招-2022-5 年经验.docx
36.0 KB
OK
我排好号，回头找你哈
秘诀:
简历指导+ 狠狠卷
尼恩老师晚上好，汇报下最近的 offer 情况。最近面试收到了三个 offer，两个是平薪，一个是跨境电商公司的 offer，涨幅暂时不到 20%。通过这次面试也让我知道自己距离高级开发还有一点点距离，还要再多看才能突破。
最后结合自己的情况，先选择去跨境电商的公司再提升下
之前是 20k*13.5，跨境电商这个是说 23k*（14-15）
恭喜恭喜
就业寒冬，能提到 3 个 offer，已经很 ，已经是高手了
很多小伙伴，面试电话一个都接不到，简历海投 7000 份，只收到 3 次面试机会，没有一个机会拿到最终 offer
您这个年薪，算下来也有 35W 了吧
年终部分还要确认下，大部分人听说只有两个月
这个时间点，拿到这个水平，挺不错的啦
持续加油卷哈
嗯嗯！看看明年自己有没有能力冲击高开
把你我视频都吃透
辛苦老师再帮忙指导下哈


1.5年小伙搞定15K offer
就业寒冬涨100%
5月7日改简历
11月21日晒offer
秘诀：
简历指导+狠狠卷
多谢啦
你好，好多公司都觉得我的简历太简单了，我感觉是不是没法优化了
简历.pdf
397.4 KB
好的
晚上指导你改
推送中台需要加不
我还没做啊，准备八股文暂时没时间搞这些，入职了会开始搞
OK
还有其他的项目吗
一个项目，太少啦
那种练习项目也行
他那个不止两倍吧，我才是两倍，原先7k，现在15k
那也很牛
都翻倍了，都是牛人
正常就30%
慢慢来
下一步可以瞄准25k
其实我工作一年
那你就更牛逼拉
一年经验，搞定15k offer，舍你其谁


卷王逆袭成功案例
6年小伙从很少面试机会到
搞定35K*14薪
3月5日改简历
4月11日拿offer
面试邀请少
小伙很苦恼
理想很丰满
目标35K
一个月拿到了
理想的offer
面试法宝
rocketmq四部曲

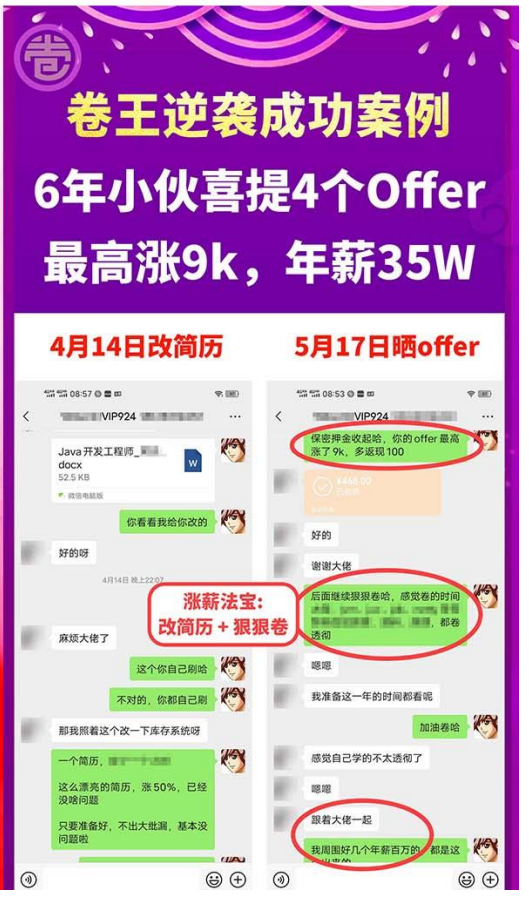

6年 经验小伙伴
喜收25K offer
3月12日改简历
12月1日晒offer
秘诀：
简历指导+狠狠卷
老师
咱们以这个项目为模板改哈
嗯好
项目有多少人，你带领多少人？
你工作几年了？
6年了
项目有多少人，你带领多少人？
语音交流一下
好
你方便吗
OK
不知道往哪个方向发展
改简历挪窝，20 涨到 25 k，高级开发，P7 带的后端团队
任务重，道路远，并不是没有方向


7年经验卷王
薪酬涨30%
7月11日改简历
9月1日晒offer
秘诀：
改简历 + 狠狠卷
咱们开始修改哈
要语音还是打字也行，还没到家
解决过最复杂的难题是啥？说 3 个哈
看看，有没有问题？
没问题
看看，有没有问题？
只能挪窝
下次你喊足一点哦
只要本事好，拿 offer 比较容易的
喊了，但是我流水太差了。。。我这家是补贴，银行不认这玩意，只能在上家的基础上加到 30% 封顶（特别优秀除外，我只是不错。。。）
入职了，最终并不能大幅涨薪，还是旧的太低，只涨了 30%
恭喜一下
现在不比往年
能涨 30% 是大牛
拿到 offer 都算大牛
现在很多人投几百发，连个电话都没有
你已经很


4年经验卷王逆袭
被毕业后，反涨24W
7月改简历
8月30日晒offer
秘诀:
改简历 + 狠狠卷
这就是你的简历
差得太多啦
ok
总共写了四个项目，最近的一年的还没补充上
你是在职，还是离职呢?
离职
原因大概是啥?
这个地方要这样写
项目被终止
方便语音沟通不
ok
是的 经你这么指导，非常重要
老哥 我八月十号开始找工作，今天已经入职了
现金基本持平，股票 +24W
总计涨了多少呢
能涨24W
股票这个吧只能到手了才算
也不错啦
很多小伙伴，面试机会都没有
感谢老哥的指导，继续跟你卷技术
继续狠狠卷哈，马上就技术自由拉

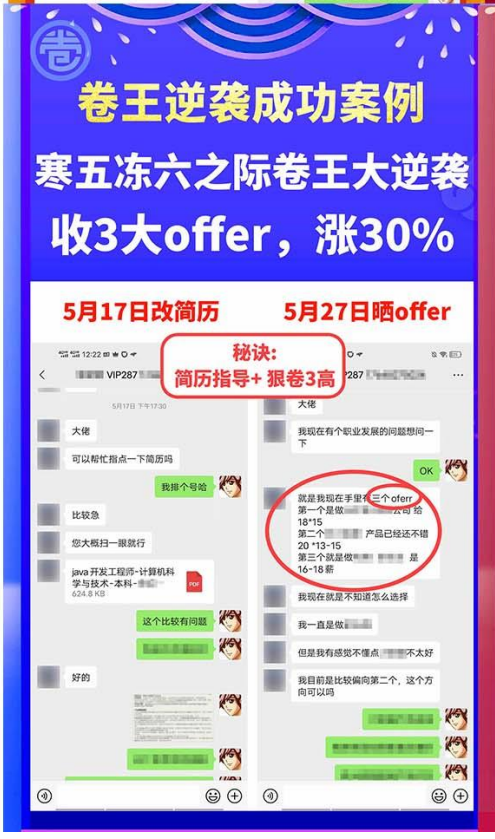

小伙5月份"被毕业"，改简历后
斩获顶级央企Offer
涨薪7000+
5月29日改简历
7月5日晒offer
秘诀:
简历指导+ 狠卷3高
快速看书，就要不求甚解，把目录和 场景大概一下，然后重点的地方，用到的地方，再去回炉
尼恩 我被“ 毕业 ”了
这周末或下周我你改一下简历
毕业了没有关系
ok，发我吧
it行业，跳来跳去，太频繁啦
嗯，其实有点心理准备
简历
不太会写简历
尼恩 我拿到的offer了
涨20%，不还价的
看起来不差钱呀
超过了8000没
平均算下来
7000多
好的
有啥面试的心得吗
可以分享给其他小伙伴的


卷王逆袭成功案例
武汉6年喜收4个优质offer
最高的年薪35W
2月9日改简历
4月15日晒offer
尼恩老师，新年好!
能帮忙修改下简历吗?
金三银四准备跳了
可以的
java 开发-6年-简历-.pdf
拜托了，恩师。希望能拿 25k 回来给你报喜
好，我加一下
还有吗?
面试法宝:
改简历 + 实操
恩哥，决赛圈 offer 了
截图是以我目前认知能写出来的评分了，麻烦帮我参考下
选择大于努力，恩师助我上岸
这么多 offer，我看看哈
都是恩哥指点有方，本来还有个新能源汽车的，35w给拒了，主要太远了
跟着恩哥卷的时间太短了，目前实力也只能到这儿了
这里边有个大数据的，感觉也不错

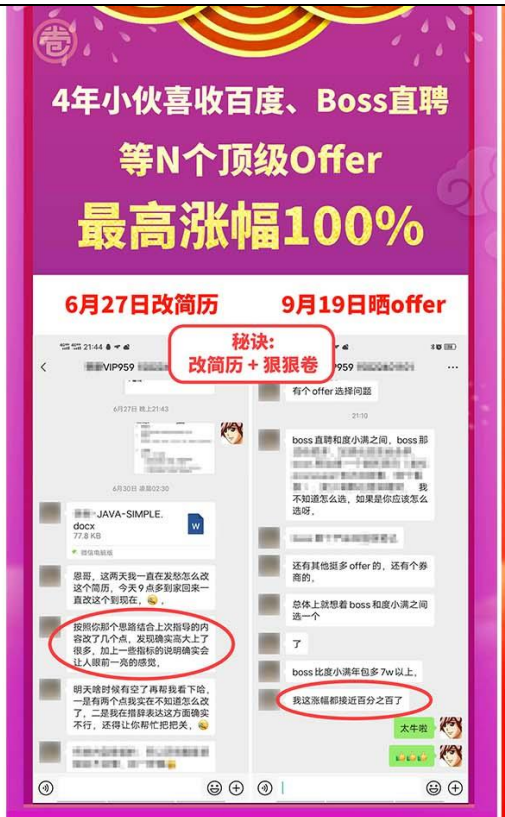

卷王逆袭成功案例
6年小伙喜提4个Offer
最高涨9k，年薪35W
4月14日改简历
5月17日晒offer
Java 开发工程师_.docx
你看看我给你改的
好的呀
麻烦大佬了
这个你自己刷哈
不对的，你都自己刷
那我照着这个改一下库存系统呀
一个简历，
这么漂亮的简历，涨 50%，已经没啥问题
只要准备好，不出大纰漏，基本没问题啦
涨薪法宝:
改简历 + 狠狠卷
保密押金收起哈，你的 offer 最高涨了 9k，多返现 100
好的
谢谢大佬
后面继续狠狠卷哈，感觉卷的时间
透彻
嗯嗯
我准备这一年的时间都看呢
加油卷哈
感觉自己学的不太透彻了
嗯嗯
跟着大佬一起
我周围好几个年薪百万的，都是这


卷王逆袭成功案例
5年经验小伙收2个offer
最高涨薪8k，年薪42W
5月9日改简历
5月30日晒offer
秘诀:
简历指导+ 狠卷3高
以此为样
大家狠狠卷
打造最卷IT社群


卷王逆袭成功案例
非全日制 6年经验卷王
喜提3个Offer，年包30W
5月9日改简历
5月18日晒offer
面试法宝:
改简历 + 狠狠卷


卷王逆袭成功案例
寒五冻六之际卷王大逆袭
收3大offer，涨30%
5月17日改简历
5月27日晒offer
秘诀:
简历指导+ 狠卷3高


卷王逆袭成功案例
4年卷王入职微软，涨50%
3月7日改简历
5月12日晒offer
涨薪法宝:
改简历 + 狠狠卷


4年小伙喜收百度、Boss直聘
等N个顶级Offer
最高涨幅100%
6月27日改简历
9月19日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
4年卷王入收2个offer，涨50%
3月23日改简历
5月12日晒offer
涨薪法宝:
改简历＋狠狠卷


小伙大三暑期很焦虑
跟着尼恩卷一年
校招斩获顶级央企Offer
去年5月19日加入VIP群
今年7月5日晒offer
秘诀:
狠狠卷 书+视频


小伙高中学历
薪酬涨120%
5月6日改简历
7月22日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
非全日制卷王 面试3家
收2个offer 涨薪30%
4月13日改简历
4月21日晒offer
面试法宝：
改简历 + 面试题


5年卷王喜收2大Offer
最高涨5K
5月19日改简历
9月13日晒offer
秘诀：
改简历 + 狠狠卷


卷王逆袭成功案例
3年经验卷王，涨60%
4月16日改简历
5月11日晒offer
涨薪法宝：
改简历 + 狠狠卷


卷王逆袭成功案例
双非二本小伙春招大翻身
喜提9大offer
2月22日改简历
4月13日晒offer
面试法宝：
改简历 + IM实操
9大offer
最高年薪30万

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md