HashMap？面试？我是谁？我在哪

Java杂记 昨天

现在是晚上11点了，学校屠猪馆的自习室因为太晚要关闭了。勤奋且疲惫的小鲁班也从屠猪馆出来了，正准备回宿舍洗洗睡，由于自习室位置比较偏僻所以是接收不到手机网络信号的，因此小鲁班从兜里掏出手机的时候，信息可真是炸了呀。小鲁班心想，微信群平时都没什么人聊天，今晚肯定是发生了什么大事。仔细一看，才发现原来是小鲁班的室友达摩（光头）拿到了阿里巴巴 Java 开发实习生的 Offer，此时小鲁班真替他室友感到高兴的同时，心里也难免会产生一丝丝的失落感，那是因为自己投了很多份简历，别说拿不拿得到 Offer，就连给面试邀的公司也都寥寥无几。小鲁班这会可真是受到了一万点 **真实暴击。** 不过小鲁班还是很乐观的，很快调整了心态，带上耳机，慢慢的走回了宿舍，正打算准备向他那神室友达摩取取经。

片刻后~

小鲁班：666，听说你拿到了阿里的 Offer，能透露一下面试内容和技巧吗？

达摩：嘿嘿嘿，没问题鸭，叫声爸爸我就告诉你。

小鲁班：耙耙（表面笑嘻嘻，心里MMP）

达摩：其实我也不是很记得了（请继续装），但我还是记得那么一些。如果你是面的 Java，首先当然是JAVA的基础知识：数据结构（Map / List / Set等）、 设计模式 、算法、线程相关、IO/NIO、序列化等等。其次是高级特征：反射机制，并发与锁，JVM（GC策略，类加载机制，内存模型）等等。

小鲁班：问这么多内容，那岂不是一个人都面试很久吗？

达摩：不是的，面试官一般都会用连环炮的方式提问的。

小鲁班：你说的连环炮是什么意思鸭？

达摩：那我举个例子：

- 就比如问你 **HashMap 是不是有序的？** 你回答不是有序的。
- 那面试官就会可能继续问你， **有没有有序的Map实现类呢？** 你如果这个时候说不知道的话，那这块问题就到此结束了。如果你说有 TreeMap 和 LinkedHashMap。
- 那么面试官接下来就可能会问你， **TreeMap 和 LinkedHashMap 是如何保证它的顺序的？** 如果你回答不上来，那么到此为止。如果你说 TreeMap 是通过实现

SortMap 接口，能够把它保存的键值对根据 key 排序，基于红黑树，从而保证 TreeMap 中所有键值对处于有序状态。LinkedHashMap 则是通过插入排序（就是你 put 的时候的顺序是什么，取出来的时候就是什么样子）和访问排序（改变排序把访问过的放到底部）让键值有序。

- 那么面试官还会继续问你， **你觉得它们两个哪个的有序实现比较好？** 如果你依然可以回答的话，那么面试官会继续问你， **你觉得还有没有比它更好或者更高效的实现方式？**

无穷无尽深入，直到你回答不出来或者面试官认为问题到底了。

小鲁班捏了一把汗，我去……这是魔鬼吧，那我们来试试呗（因为小鲁班刚刚在自习室才看了这章的知识，想趁机装一波逼，毕竟刚刚叫了声爸爸~~）

于是达摩 and 小鲁班就开始了对决：

1、为什么用HashMap?

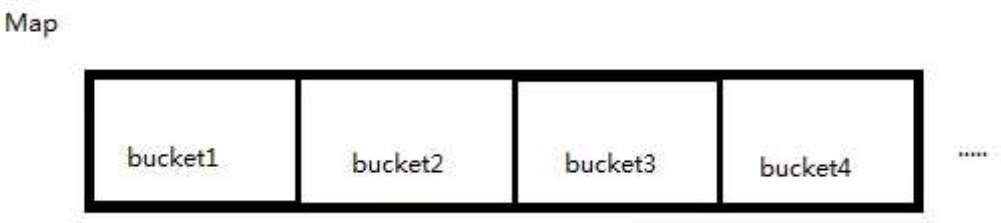
- HashMap 是一个散列桶（数组和链表），它存储的内容是键值对 key-value 映射
- HashMap 采用了数组和链表的数据结构，能在查询和修改方便继承了数组的线性查找和链表的寻址修改
- HashMap 是非 synchronized，所以 HashMap 很快
- HashMap 可以接受 null 键和值，而 Hashtable 则不能（原因就是 equlas() 方法需要对象，因为 HashMap 是后出的 API经过处理才可以）

2、HashMap 的工作原理是什么?

HashMap 是基于 hashing 的原理

我们使用 put(key, value) 存储对象到 HashMap 中，使用 get(key) 从 HashMap 中获取对象。当我们给 put() 方法传递键和值时，我们先对键调用 hashCode() 方法，计算并返回的 hashCode 是用于找到 Map 数组的 bucket 位置来储存 Node 对象。

这里关键点在于指出，HashMap 是在 bucket 中储存键对象和值对象，作为Map.Node 。



以下是 HashMap 初始化

简化的模拟数据结构：

```
Node[] table = new Node[16]; // 散列桶初始化，table
class Node {
```

```
    hash; //hash值
    key; //键
    value; //值     node next; //用于指向链表的下一层（产生冲突，用拉链法）
}
```

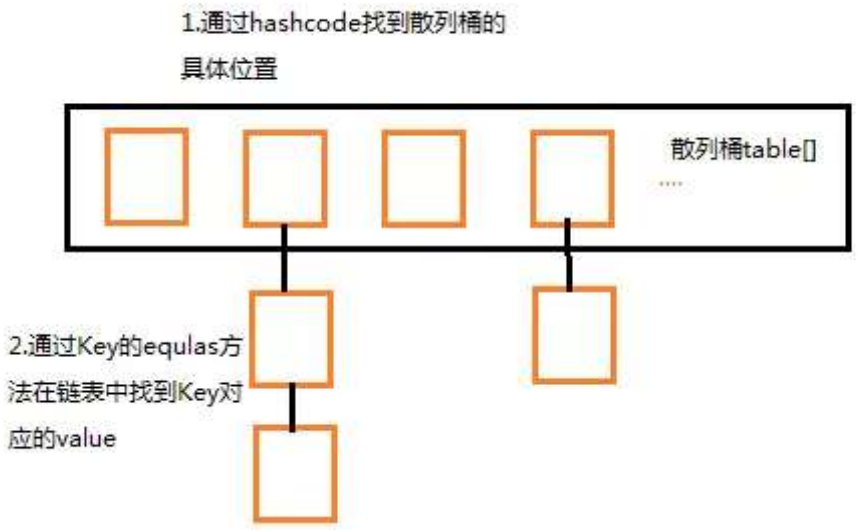
以下是具体的 put 过程（JDK1.8）

1. 对 Key 求 Hash 值，然后再计算下标
2. 如果没有碰撞，直接放入桶中（碰撞的意思是计算得到的 Hash 值相同，需要放到同一个 bucket 中）
3. 如果碰撞了，以链表的方式链接到后面
4. 如果链表长度超过阀值（TREEIFY THRESHOLD==8），就把链表转成红黑树，链表长度低于6，就把红黑树转回链表
5. 如果节点已经存在就替换旧值
6. 如果桶满了（容量16*加载因子0.75），就需要 resize（扩容2倍后重排）

以下是具体 get 过程

考虑特殊情况：如果两个键的 hashcode 相同，你如何获取值对象？

当我们调用 get() 方法，HashMap 会使用键对象的 hashcode 找到 bucket 位置，找到 bucket 位置之后，会调用 keys.equals() 方法去找到链表中正确的节点，最终找到要找的值对象。



3、有什么方法可以减少碰撞？

扰动函数可以减少碰撞

原理是如果两个不相等的对象返回不同的 hashcode 的话，那么碰撞的几率就会小些。这就意味着存链表结构减小，这样取值的话就不会频繁调用 equal 方法，从而提高 HashMap 的性能（扰动即 Hash 方法内部的算法实现，目的是让不同对象返回不同hashcode）。

使用不可变的、声明作 final 对象，并且采用合适的 equals() 和 hashCode() 方法，将会减少碰撞的发生

不可变性使得能够缓存不同键的 hashcode，这将提高整个获取对象的速度，使用 String、Integer 这样的 wrapper 类作为键是非常好的选择。

为什么 String、Integer 这样的 wrapper 类适合作为键?

因为 String 是 final，而且已经重写了 equals() 和 hashCode() 方法了。不可变性是必要的，因为为了要计算 hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的 hashcode 的话，那么就不能从 HashMap 中找到你想要的对象。

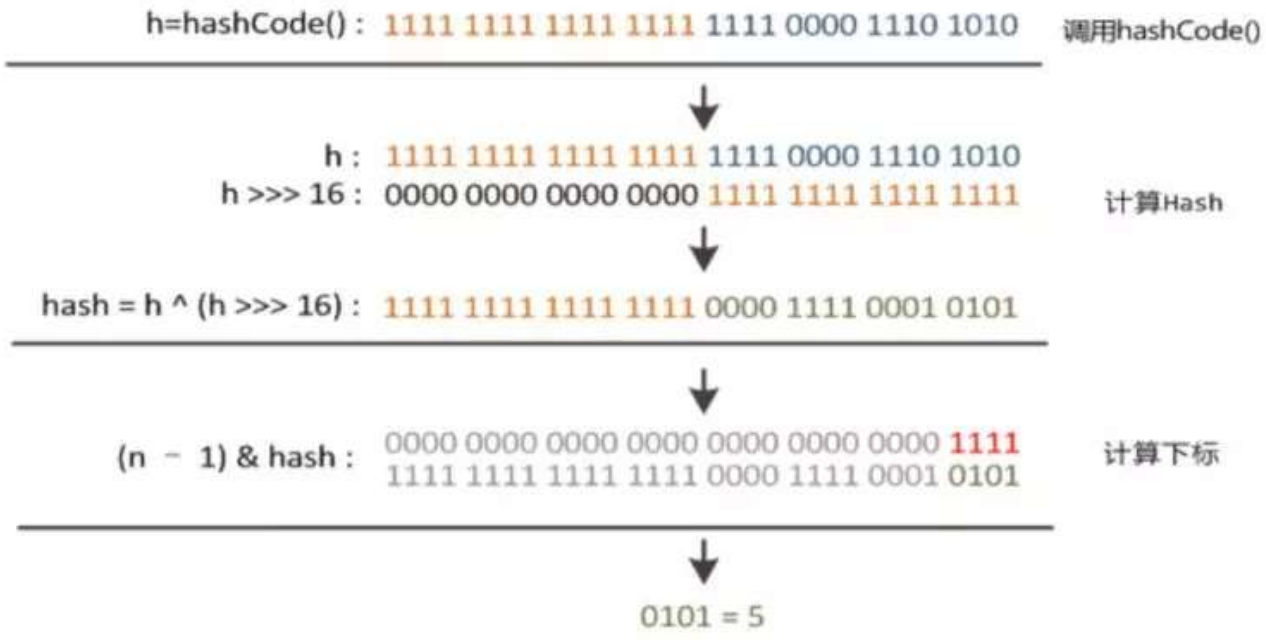
4、HashMap 中 hash 函数怎么是实现的?

我们可以看到，在 hashmap 中要找到某个元素，需要根据 key 的 hash 值来求得对应数组中的位置。如何计算这个位置就是 hash 算法。

前面说过，hashmap 的数据结构是数组和链表的结合，所以我们当然希望这个 hashmap 里面的元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个。那么当我们用 hash 算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表。所以，我们首先想到的就是把 hashcode 对数组长度取模运算。这样一来，元素的分布相对来说是比较均匀的。

但是“模”运算的消耗还是比较大的，能不能找一种更快速、消耗更小的方式？我们来看看 JDK1.8 源码是怎么做的（被楼主修饰了一下）

```
static final int hash(Object key) {
    if (key == null){
        return 0;
    }
    int h;
    h = key.hashCode(); 返回散列值也就是hashcode
    // ^ ：按位异或
    // >>>:无符号右移，忽略符号位，空位都以0补齐
    //其中n是数组的长度，即Map的数组部分初始化长度
    return (n-1)&(h ^ (h >>> 16));
}
```



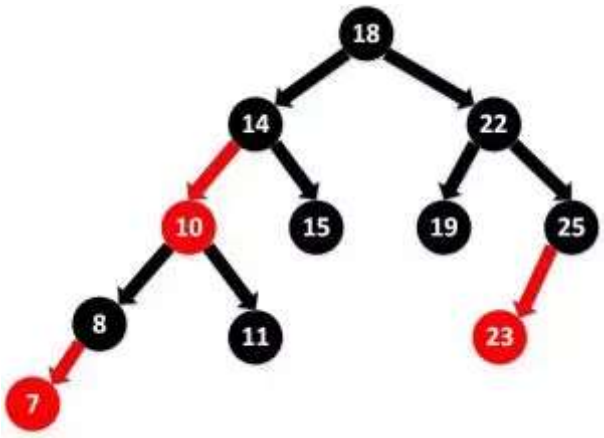
简单来说就是：

- 高16 bit 不变，低16 bit 和高16 bit 做了一个异或（得到的 hashcode 转化为32位二进制，前16位和后16位低16 bit和高16 bit做了一个异或）
- (n·1) & hash = -> 得到下标

5、拉链法导致的链表过深，为什么不用二叉查找树代替而选择红黑树？为什么不一直使用红黑树？

之所以选择红黑树是为了解决二叉查找树的缺陷：二叉查找树在特殊情况下会变成一条线性结构（这就跟原来使用链表结构一样了，造成层次很深的问题），遍历查找会非常慢。而红黑树在插入新数据后可能需要通过左旋、右旋、变色这些操作来保持平衡。引入红黑树就是为了查找数据快，解决链表查询深度的问题。我们知道红黑树属于平衡二叉树，为了保持“平衡”是需要付出代价的，但是该代价所损耗的资源要比遍历线性链表要少。所以当长度大于8的时候，会使用红黑树；如果链表长度很短的话，根本不需要引入红黑树，引入反而会慢。

6、说说你对红黑树的见解？



1. 每个节点非红即黑
2. 根节点总是黑色的

3. 如果节点是红色的，则它的子节点必须是黑色的（反之不一定）
4. 每个叶子节点都是黑色的空节点（NIL节点）
5. 从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）

7、解决 hash 碰撞还有那些办法?

开放定址法

当冲突发生时，使用某种探查技术在散列表中形成一个探查（测）序列。沿此序列逐个单元地查找，直到找到给定的地址。按照形成探查序列的方法不同，可将开放定址法区分为线性探查法、二次探查法、双重散列法等。

下面给一个线性探查法的例子：

问题：已知一组关键字为 (26，36，41，38，44，15，68，12，06，51)，用除余法构造散列函数，用线性探查法解决冲突构造这组关键字的散列表。

解答：为了减少冲突，通常令装填因子 α 由除余法因子是13的散列函数计算出的上述关键字序列的散列地址为 (0，10，2，12，5，2，3，12，6，12)。

前5个关键字插入时，其相应的地址均为开放地址，故将它们直接插入 T[0]、T[10)、T[2]、T[12] 和 T[5] 中。

当插入第6个关键字15时，其散列地址2（即 h(15)=15%13=2）已被关键字 41（15和41互为同义词）占用。故探查 h1=(2+1)%13=3，此地址开放，所以将 15 放入 T[3] 中。

当插入第7个关键字68时，其散列地址3已被非同义词15先占用，故将其插入到T[4]中。

当插入第8个关键字12时，散列地址12已被同义词38占用，故探查 hl=(12+1)%13=0，而 T[0] 亦被26占用，再探查 h2=(12+2)%13=1，此地址开放，可将12插入其中。

类似地，第9个关键字06直接插入 T[6] 中；而最后一个关键字51插人时，因探查的地址 12，0，1，…，6 均非空，故51插入 T[7] 中。

8、如果 HashMap 的大小超过了负载因子（load factor）定义的容量怎么办?

HashMap 默认的负载因子大小为0.75。也就是说，当一个 Map 填满了75%的 bucket 时候，和其它集合类一样（如 ArrayList 等），将会创建原来 HashMap 大小的两倍的 bucket 数组来重新调整 Map 大小，并将原来的对象放入新的 bucket 数组中。这个过程叫作 **rehashing** 。

因为它调用 hash 方法找到新的 bucket 位置。这个值只可能在两个地方，一个是原下标的位置，另一种是在下标为 **<原下标+原容量>** 的位置。

9、重新调整 HashMap 大小存在什么问题吗?

重新调整 HashMap 大小的时候，确实存在条件竞争。

因为如果两个线程都发现 HashMap 需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来。因为移动到新的 bucket 位置的时候，HashMap 并不会将元素放在链表的尾部，而是放在头部。这是为了避免尾部遍历（tail traversing）。如果条件竞争发生了，那么就死循环了。多线程的环境下不使用 HashMap。

为什么多线程会导致死循环，它是怎么发生的?

HashMap 的容量是有限的。当经过多次元素插入，使得 HashMap 达到一定饱和度时，Key 映射位置发生冲突的几率会逐渐提高。这时候， HashMap 需要扩展它的长度，也就是进行 Resize。

1. 扩容：创建一个新的 Entry 空数组，长度是原数组的2倍
2. rehash：遍历原 Entry 数组，把所有的 Entry 重新 Hash 到新数组

（这个过程比较烧脑，暂不作流程图演示，有兴趣去看看我的另一篇博文“ HashMap扩容全过程 ”）

达摩：哎呦，小老弟不错嘛~~意料之外呀

小鲁班：嘿嘿，优秀吧，中场休息一波，我先喝口水

达摩：不仅仅是这些哦，面试官还会问你相关的集合类对比，比如：

10、HashTable

- 数组 + 链表方式存储
- 默认容量：11（质数为宜）
- put 操作：首先进行索引计算 （key.hashCode() & 0x7FFFFFFF） % table.length；若在链表中找到了，则替换旧值，若未找到则继续；当总元素个数超过 容量 * 加载因子 时，扩容为原来 2 倍并重新散列；将新元素加到链表头部
- 对修改 Hashtable 内部共享数据的方法添加了 synchronized，保证线程安全

11、HashMap 与 HashTable 区别

- 默认容量不同，扩容不同
- 线程安全性：HashTable 安全
- 效率不同：HashTable 要慢，因为加锁

12、可以使用 CocurrentHashMap 来代替 Hashtable 吗?

- 我们知道 Hashtable 是 synchronized 的，但是 ConcurrentHashMap 同步性能更好，因为它仅仅根据同步级别对 map 的一部分进行上锁
- ConcurrentHashMap 当然可以代替 HashTable，但是 HashTable 提供更强的线程安全性
- 它们都可以用于多线程的环境，但是当 Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。由于 ConcurrentHashMap 引入了分割（segmentation），不论它变得多么大，仅仅需要锁定 Map 的某个

部分，其它的线程不需要等到迭代完成才能访问 Map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定 Map 的某个部分，而 Hashtable 则会锁定整个 Map

13、CocurrentHashMap（JDK 1.7）

- CocurrentHashMap 是由 Segment 数组和 HashEntry 数组和链表组成
- Segment 是基于重入锁（ReentrantLock）：一个数据段竞争锁。每个 HashEntry 一个链表结构的元素，利用 Hash 算法得到索引确定归属的数据段，也就是对应到在修改时需要竞争获取的锁。ConcurrentHashMap 支持 CurrencyLevel（Segment 数组数量）的线程并发。每当一个线程占用锁访问一个 Segment 时，不会影响到其他的 Segment
- 核心数据如 value，以及链表都是 volatile 修饰的，保证了获取时的可见性
- 首先是通过 key 定位到 Segment，之后在对应的 Segment 中进行具体的 put 操作如下：
  - 将当前 Segment 中的 table 通过 key 的 hashcode 定位到 HashEntry。
  - 遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value
  - 不为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容
  - 最后会解除在 1 中所获取当前 Segment 的锁。
- 虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理

首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁。

- 尝试自旋获取锁
- 如果重试的次数达到了 MAX_SCAN_RETRIES 则改为阻塞锁获取，保证能获取成功。最后解除当前 Segment 的锁

14、CocurrentHashMap（JDK 1.8）

CocurrentHashMap 抛弃了原有的 Segment 分段锁，采用了 `CAS + synchronized` 来保证并发安全性。 其中的 `val next` 都用了 volatile 修饰，保证了可见性。

最大特点是引入了 CAS

借助 Unsafe 来实现 native code。CAS有3个操作数，内存值 V、旧的预期值 A、要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值V修改为 B，否则什么都不做。Unsafe 借助 CPU 指令 cmpxchg 来实现。

CAS 使用实例

对 sizeCtl 的控制都是用 CAS 来实现的：

- -1 代表 table 正在初始化
- N 表示有 -N-1 个线程正在进行扩容操作
- 如果 table 未初始化，表示table需要初始化的大小
- 如果 table 初始化完成，表示table的容量，默认是table大小的0.75倍，用这个公式算 0.75（n – (n >>> 2)）

CAS 会出现的问题：ABA

解决：对变量增加一个版本号，每次修改，版本号加 1，比较的时候比较版本号。

put 过程

- 根据 key 计算出 hashcode
- 判断是否需要进行初始化
- 通过 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功
- 如果当前位置的 hashcode == MOVED == -1,则需要进行扩容
- 如果都不满足，则利用 synchronized 锁写入数据
- 如果数量大于 TREEIFY_THRESHOLD 则要转换为红黑树

get 过程

- 根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值
- 如果是红黑树那就按照树的方式获取值
- 就不满足那就按照链表的方式遍历获取值

此时躺着床上的张飞哄了一声：睡觉了睡觉了~

见此不太妙：小鲁班立马回到床上把被子盖过头，心里有一丝丝愉悦感。不对，好像还没洗澡……

by the way

ConcurrentHashMap 在 Java 8 中存在一个 bug 会进入死循环，原因是递归创建 ConcurrentHashMap 对象，但是在 JDK 1.9 已经修复了。场景重现如下：

```
public class ConcurrentHashMapDemo{
    private Map<Integer,Integer> cache =new ConcurrentHashMap<>(15);

    public static void main(String[]args){
        ConcurrentHashMapDemo ch =    new ConcurrentHashMapDemo();
        System.out.println(ch.fibonaacci(80));
    }

    public int fibonaacci(Integer i){
        if(i==0||i ==1) {
            return i;
        }

        return cache.computeIfAbsent(i,(key) -> {
```

```
            System.out.println("fibonaacci : "+key);
            return fibonaacci(key -1)+fibonaacci(key - 2);
        });
    }
}
```

目前100000+人已关注加入我们