链滴

大厂是如何考察 HashMap 的

作者：yuanzhicun
原文链接：https://ld246.com/article/1638861761867
来源网站：链滴

一、HashMap的底层数据结构

HashMap是我们非常常用的数据结构，由数组和链表组合构成的数据结构。

在不发生hash冲撞的情况下数据结构是数组，一但出现hash冲突，则Entry.next 来实现链表结构

大概如下，数组里面每个地方都存了Key-Value这样的实例，在Java7叫Entry在Java8中叫Node。

二、链表节点是怎么插入的

java7 头插法，就是说新来的值会取代原有的值，原有的值就顺推到链表中去，就像上面的例子一样因为写这个代码的作者认为后来的值被查找的可能性更大一点，提升查找的效率。

但是，在java8之后，都是所用尾部插入了。（图画的有点low，体谅）

<font color=#999AAA >

tip:为啥尾部插呢？就是因为多线程情况，在扩容的时候容易形成闭环（死循环），最后面我会画图明这个问题的，大家耐心往下看。毕竟这是本文的重点

三、什么时候扩容

数组容量是有限的，数据多次插入的，到达一定的数量就会进行扩容，也就是resize。什么时候resiz呢？有两个因素：Capacity：HashMap当前长度。

LoadFactor：负载因子，默认值0.75f。

怎么理解呢，就比如当前的容量大小为16，当你存进第14个的时候，判断发现需要进行resize了，那进行扩容，但是HashMap的扩容也不是简单的扩大点容量这么简单的。

扩容分两步

1.创建一个新的Entry空数组，长度是原数组的2倍。

2.遍历原Entry数组，把所有的Entry重新Hash到新数组。

（为什么不直接复制过去呢？是因为hash在计算的时候其实涉及了数组长度，你扩容长度都改变了那么就好导致扩容前后计算的hash值是不一样的）

四、为什么默认初始化长度为16

因为计算位置的时候用到了位运算(位与运算比算数计算的效率高了很多)

index的计算公式：index = HashCode（Key） & （Length- 1）其实这个计算index的方法和hash ode%length小说是一样的

例如： hashCode : 2 2%16 index= 2

hashCode: 2 （0010） & 1111(16-1的二进制） = 0010（2）

所以用位与运算效果与取模一样，性能也提高了不少！

那为啥用16不用别的呢

因为在使用不是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。

这是为了实现均匀分布。

五、为什么要求是2的指数幂

因为 底层hash运算 是 h &(length -1）这样情况能保证空间不浪费，如果传的是偶数，-1后就是奇，他的2进制位末尾肯定是0 进行位运算 末尾也是0 一旦高位算出是 1 则好多空间浪费

HashMap初始化时，如果指定容量大小为10，那么实际大小是多少?

16，因为HashMap的初始化函数中规定容量大小要是2的指数倍，即2，4，8，16，所以当指定容量10时，实际容量为16。

六、 为啥不直接使用hashCode

HashMap的 hash(Obeject k)方法中为什么在调用 k.hashCode()方法获得hash值后，为什么不直接这个hash进行取余，而是还要将hash值进行右移和异或运算?

A：如果HashMap容量比较小而hash值比较大的时候，哈希冲突就容易变多。基于HashMap的index or底层设计，假设容量为16，那么就要对二进制0000 1111（即15）进行按位与操作，那么hash值二进制的高28位无论是多少，都没意义，因为都会被0&，变成0。所以哈希冲突容易变多。那么hash Obeject k)方法中在调用 k.hashCode()方法获得hash值后，进行的一步运算：h^=(h>>>20)^(h>>12);有什么用呢？首先，h>>>20和h>>>12是将h的二进制中高位右移变成低位。其次异或运算是用了特性：同0异1原则，尽可能的使得h>>>20和h>>>12在将来做取余（按位与操作方式）时都参到运算中去。综上，简单来说，通过h^=(h>>>20)^(h>>>12);运算，可以使k.hashCode()方法获的hash值的二进制中高位尽可能多地参与按位与操作，从而减少哈希冲突。

七、 HashMap扩容的原因

提升HashMap的get、put等方法的效率，因为如果不扩容，链表就会越来越长，导致插入和查询效都会变低。

八、 jdk 7 与 jdk 8的比较

1.1.7基于头插，1.8是尾插

2. 1.7 采用数组加链表，1.8采用 数组+红黑树（jdk1.8引入红黑树后，如果单链表节点个数超过8个是否一定会树化?

不一定，它会先去判断是否需要扩容（即判断当前节点个数是否大于扩容的阈值），如果满足扩容条，直接扩容，不会树化，因为扩容不仅能增加容量，还能缩短单链表的节点数，一举两得。

，当链表节点个数超过8个（m默认值）以后，开始使用红黑树，使用红黑树一个综合取优的选择，对于其他数据结构，红黑树的查询和插入效率都比较高。而当红黑树的节点个数小于6个（默认值）后，又开始使用链表。这两个阈值为什么不相同呢？主要是为了防止出现节点个数频繁在一个相同的值来回切换，举个极端例子，现在单链表的节点个数是9，开始变成红黑树，然后红黑树节点个数又成8，就又得变成单链表，然后节点个数又变成9，就又得变成红黑树，这样的情况消耗严重浪费，因干脆错开两个阈值的大小，使得变成红黑树后“不那么容易”就需要变回单链表，同样，使得变成单表后，“不那么容易”就需要变回红黑树。

）

九、为什么jdk.17的hashMap扩容存在死循环

重点就是这：

Entry<K,V> next = e.next;（两个线程同时走到这，其中一个等待CPU的执行权，另一个直接顺利执
扩容，然后循环遍历旧集合，将数据迁移到新创建的集合中，因为链表这种节点的特殊性：当前节点
有下一个节点的引用，所以在迁移的过程中next的值会有问题下面看源代码大家简单了解下过程，后
画图讲解 ）