2017年的秋招彻底结束了，感觉Java上面的最常见的集合相关的问题就是hash……系列和一些常用并发集合和队列，堆等结合算法一起考察，不完全统计，本人经历：先后百度、唯品会、58同城、新浪微博、趣分期、美团点评等都在1、2……面的时候被问过无数次，都问吐了&\_&，其他公司笔试的时候，但凡有Java的题，都有集合相关考点，尤其hash表……现在总结下。

* Java集合概述
* HashMap介绍
* HashMap源码学习
* 关于HashMap的几个经典问题
* HashTable介绍和源码学习
* HashMap 和 HashTable 比较

先上图

**Set和List接口是Collection接口的子接口，分别代表无序集合和有序集合，Queue是Java提供的队列实现。**

**Map用于保存具有key-value映射关系的数据**

Java 中有四种常见的Map实现——HashMap, TreeMap, Hashtable和LinkedHashMap：

* HashMap就是一张hash表，键和值都没有排序。
* TreeMap以红黑树结构为基础，键值可以设置按某种顺序排列。
* LinkedHashMap保存了插入时的顺序。
* Hashtable是同步的(而HashMap是不同步的)。所以如果在线程安全的环境下应该多使用HashMap，而不是Hashtable，因为Hashtable对同步有额外的开销，不过JDK 5之后的版本可以使用conncurrentHashMao代替HashTable。

本文重点总结HashMap，HashMap是基于哈希表实现的，每一个元素是一个key-value对，其内部通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

HashMap是非线程安全的，只用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

HashMap 实现了Serializable接口，因此它支持序列化。

HashMap还实现了Cloneable接口，故能被克隆。

**关于hashmap的用法，这里就不再赘述了，只说原理和一些注意点。**

* **HashMap的存储结构**

紫色部分即代表哈希表本身（其实是一个数组），数组的每个元素都是一个单链表的头节点，链表是用来解决hash地址冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中保存。

* **HashMap有四个构造方法，方法中有两个很重要的参数：初始容量和加载因子**

这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（默认为16），加载因子是哈希表当前key的数量和容量的比值，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表提前进行 resize 操作（即扩容）。如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），严重浪费。

JDK开发者规定的默认加载因子为0.75，因为这是一个比较理想的值。另外，无论指定初始容量为多少，构造方法都会将实际容量设为不小于指定容量的2的幂次方，且最大值不能超过2的30次方。

* **重点分析HashMap中用的最多的两个方法put和get的源码**
  + get方法源码

*// 获取key对应的value*

**public** V **get(**Object key**)** **{**

**if** **(**key **==** **null)**

**return** getForNullKey**();**

*// 获取key的hash值*

**int** hash **=** hash**(**key**.**hashCode**());**

*// 在“该hash值对应的链表”上查找“键值等于key”的元素*

**for** **(**Entry**<**K**,** V**>** e **=** table**[**indexFor**(**hash**,** table**.**length**)];** e **!=** **null;** e **=** e**.**next**)** **{**

Object k**;**

*// 判断key是否相同*

**if** **(**e**.**hash **==** hash **&&** **((**k **=** e**.**key**)** **==** key **||** key**.**equals**(**k**)))**

**return** e**.**value**;**

**}**

*// 没找到则返回null*

**return** **null;**

**}**

*// 获取“key为null”的元素的值，HashMap将“key为null”的元素存储在table[0]位置，但不一定是该链表的第一个位置！*

**private** V **getForNullKey()** **{**

**for** **(**Entry**<**K**,** V**>** e **=** table**[**0**];** e **!=** **null;** e **=** e**.**next**)** **{**

**if** **(**e**.**key **==** **null)**

**return** e**.**value**;**

**}**

**return** **null;**

**}**

get方法源码

首先，如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。**记住，key为null的键值对永远都放在以table[0]为头结点的链表中，当然不一定是存放在头结点table[0]中。**如果key不为null，则先求的key的hash值，根据hash值找到在table中的索引，在该索引对应的单链表中查找是否有键值对的key与目标key相等，有就返回对应的value，没有则返回null。

* + put方法源码

*// 将“key-value”添加到HashMap中*

**public** V **put(**K key**,** V value**)** **{**

*// 若“key为null”，则将该键值对添加到table[0]中。*

**if** **(**key **==** **null)**

**return** putForNullKey**(**value**);**

*// 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。*

**int** hash **=** hash**(**key**.**hashCode**());**

**int** i **=** indexFor**(**hash**,** table**.**length**);**

**for** **(**Entry**<**K**,** V**>** e **=** table**[**i**];** e **!=** **null;** e **=** e**.**next**)** **{**

Object k**;**

*// 若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！*

**if** **(**e**.**hash **==** hash **&&** **((**k **=** e**.**key**)** **==** key **||** key**.**equals**(**k**)))** **{**

V oldValue **=** e**.**value**;**

e**.**value **=** value**;**

e**.**recordAccess**(this);**

**return** oldValue**;**

**}**

**}**

*// 若“该key”对应的键值对不存在，则将“key-value”添加到table中*

modCount**++;**

*// 将key-value添加到table[i]处*

addEntry**(**hash**,** key**,** value**,** i**);**

**return** **null;**

**}**

put方法源码

如果key为null，则将其添加到table[0]对应的链表中，如果key不为null，则同样先求出key的hash值，根据hash值得出在table中的索引，而后遍历对应的单链表，**如果单链表中存在与目标key相等的键值对，则将新的value覆盖旧的value，且将旧的value返回**，如果找不到与目标key相等的键值对，或者该单链表为空，则将该键值对插入到单链表的头结点位置（每次新插入的节点都是放在头结点的位置），该操作是有addEntry方法实现的，它的源码如下：

* + view code

*// 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。*

**void** **addEntry(int** hash**,** K key**,** V value**,** **int** bucketIndex**)** **{**

*// 保存“bucketIndex”位置的值到“e”中*

Entry**<**K**,** V**>** e **=** table**[**bucketIndex**];**

*// 设置“bucketIndex”位置的元素为“新Entry”，*

*// 设置“e”为“新Entry的下一个节点”*

table**[**bucketIndex**]** **=** **new** Entry**<**K**,** V**>(**hash**,** key**,** value**,** e**);**

*// 若HashMap的实际大小 不小于 “阈值”，则调整HashMap的大小*

**if** **(**size**++** **>=** threshold**)**

resize**(**2 **\*** table**.**length**);**

**}**

注意这里倒数第三行的构造方法，将key-value键值对赋给table[bucketIndex]，并将其next指向元素e，这便将key-value放到了头结点中，并将之前的头结点接在了它的后面。该方法也说明，每次put键值对的时候，总是将新的该键值对放在table[bucketIndex]处（即头结点处）。两外注意最后两行代码，每次加入键值对时，都要判断当前已用的槽的数目是否大于等于阀值（容量\*加载因子），如果大于等于，则进行扩容，将容量扩为原来容量的2倍。

**重点来分析下求hash值和索引值的方法，这两个方法便是HashMap设计的最为核心的部分，二者结合能保证哈希表中的元素尽可能均匀地散列。**

* **由hash值找到对应索引的方法如下**

**static** **int** **indexFor(int** h**,** **int** length**)** **{**

**return** h **&** **(**length**-**1**);**

**}**

因为容量初始还是设定都会转化为2的幂次。故可以使用高效的位与运算替代模运算。下面会解释原因。

* **计算hash值的方法如下**

**static** **int** **hash(int** h**)** **{**

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

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

**}**

JDK 的 HashMap 使用了一个 hash 方法对hash值使用位的操作，使hash值的计算效率很高。为什么这样做？主要是因为如果直接使用hashcode值，那么这是一个int值（8个16进制数，共32位），int值的范围正负21亿多，但是hash表没有那么长，一般比如初始16，自然散列地址需要对hash表长度取模运算，得到的余数才是地址下标。假设某个key的hashcode是0AAA0000，hash数组长默认16，如果不经过hash函数处理，该键值对会被存放在hash数组中下标为0处，因为0AAA0000 & (16-1) = 0。过了一会儿又存储另外一个键值对，其key的hashcode是0BBB0000，得到数组下标依然是0，这就说明这是个实现得很差的hash算法，因为hashcode的1位全集中在前16位了，导致算出来的数组下标一直是0。于是明明key相差很大的键值对，却存放在了同一个链表里，导致以后查询起来比较慢（蜕化为了顺序查找）。故JDK的设计者使用hash函数的若干次的移位、异或操作，把hashcode的“1位”变得“松散”，非常巧妙。

**下面是几个常见的面试题**

* **说下hashmap的 扩容机制？**

前面说了，hashmap的构造器里指明了两个对于理解HashMap比较重要的两个参数 int initialCapacity, float loadFactor,这两个参数会影响HashMap效率，HashMap底层采用的散列数组实现，利用initialCapacity这个参数我们可以设置这个数组的大小，也就是散列桶的数量，但是如果需要Map的数据过多，在不断的add之后，这些桶可能都会被占满，这是有两种策略，一种是不改变Capacity，因为即使桶占满了，我们还是可以利用每个桶附带的链表增加元素。但是这有个缺点，此时HaspMap就退化成为了LinkedList，使get和put方法的时间开销上升，这是就要采用另一种方法：增加Hash桶的数量，这样get和put的时间开销又回退到近于常数复杂度上。Hashmap就是采用的该方法。

* **关于扩容。看hashmap的扩容方法，resize方法，它的源码如下**

扩容的resize方法

// 重新调整HashMap的大小，newCapacity是调整后的单位

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

// 新建一个HashMap，将“旧HashMap”的全部元素添加到“新HashMap”中，

// 然后，将“新HashMap”赋值给“旧HashMap”。

Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int) (newCapacity \* loadFactor);

}

扩容的resize方法

很明显，是从新建了一个HashMap的底层数组，长度为原来的两倍，而后调用transfer方法，将旧HashMap的全部元素添加到新的HashMap中（要重新计算元素在新的数组中的索引位置）。transfer方法的源码如下：

*// 将HashMap中的全部元素都添加到newTable中*

**void** **transfer(**Entry**[]** newTable**)** **{**

Entry**[]** src **=** table**;**

**int** newCapacity **=** newTable**.**length**;**

**for** **(int** j **=** 0**;** j **<** src**.**length**;** j**++)** **{**

Entry**<**K**,** V**>** e **=** src**[**j**];**

**if** **(**e **!=** **null)** **{**

src**[**j**]** **=** **null;**

**do** **{**

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

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

e**.**next **=** newTable**[**i**];**

newTable**[**i**]** **=** e**;**

e **=** next**;**

**}** **while** **(**e **!=** **null);**

**}**

**}**

**}**

transfer方法源码

很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用HashMap时，最好能提前预估下HashMap中元素的个数，这样有助于提高HashMap的性能。

* **hashmap什么时候需要增加容量呢？**

因为效率问题，JDK采用预处理法，这时前面说的loadFactor就派上了用场，当size > initialCapacity \* loadFactor，hashmap内部resize方法就被调用，使得重新扩充hash桶的数量，在目前的实现中，是增加一倍，这样就保证当你真正想put新的元素时效率不会明显下降。所以一般情况下HashMap并不存在键值放满的情况。当然并不排除极端情况，比如设置的JVM内存用完了，或者这个HashMap的Capacity已经达到了MAXIMUM\_CAPACITY（目前的实现是2^30）。

* **initialCapacity和loadFactor参数设什么样的值好呢？**

initialCapacity的默认值是16，有些人可能会想如果内存足够，是不是可以将initialCapacity设大一些，即使用不了这么大，就可避免扩容导致的效率的下降，反正无论initialCapacity大小，我们使用的get和put方法都是常数复杂度的。这么说没什么不对，但是可能会忽略一点，实际的程序可能不仅仅使用get和put方法，也有可能使用迭代器，如initialCapacity容量较大，那么会使迭代器效率降低。所以理想的情况还是在使用HashMap前估计一下数据量。

加载因子默认值是0.75，是JDK权衡时间和空间效率之后得到的一个相对优良的数值。如果这个值过大，虽然空间利用率是高了，但是对于HashMap中的一些方法的效率就下降了，包括get和put方法，会导致每个hash桶所附加的链表增长，影响存取效率。如果比较小，除了导致空间利用率较低外没有什么坏处，只要有的是内存，毕竟现在大多数人把时间看的比空间重要。但是实际中还是很少有人会将这个值设置的低于0.5。

* **HashMap的key和value都能为null么？如果k能为null，那么它是怎么样查找值的？**

如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中。

* **HashMap中put值的时候如果发生了冲突，是怎么处理的？**

JDK使用了链地址法，hash表的每个元素又分别链接着一个单链表，元素为头结点，如果不同的key映射到了相同的下标，那么就使用头插法，插入到该元素对应的链表。

* **HashMap的key是如何散列到hash表的？相比较HashTable有什么改进？**

我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除留余数法），HashTable就是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，且hashtable直接使用了hashcode值，没有重新计算。

HashMap中则通过 h&(length-1) 的方法来代替取模，其中h是key的hash值，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。

首先，length为2的整数次幂的话，h&(length-1) 在数学上就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；

其次，length为2的整数次幂的话，则一定为偶数，那么 length-1 一定为奇数，奇数的二进制的最后一位是1，这样便保证了 h&(length-1) 的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀，而如果length为奇数的话，很明显 length-1 为偶数，它的最后一位是0，这样 h&(length-1) 的最后一位肯定为0，即只能为偶数，这样导致了任何hash值都只会被散列到数组的偶数下标位置上，浪费了一半的空间，因此length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

**作为对比，在讨论一下Hashtable**

HashTable同样是基于哈希表实现的，其实类似HashMap，只不过有些区别，HashTable同样每个元素是一个key-value对，其内部也是通过单链表解决冲突问题，容量不足（超过了阀值）时，同样会自动增长。

HashTable比较古老， 是JDK1.0就引入的类，而HashMap 是 1.2 引进的 Map 的一个实现。

HashTable 是线程安全的，能用于多线程环境中。Hashtable同样也实现了Serializable接口，支持序列化，也实现了Cloneable接口，能被克隆。

Hashtable继承于Dictionary类，实现了Map接口。Dictionary是声明了操作"键值对"函数接口的抽象类。 有一点注意，HashTable除了线程安全之外（其实是直接在方法上增加了synchronized关键字，比较古老，落后，低效的同步方式），还有就是它的key、value都不为null。另外Hashtable 也有 **初始容量** 和 **加载因子**。

**public** **Hashtable()** **{**

**this(**11**,** 0.75f**);**

**}**

**默认加载因子也是 0.75，HashTable在不指定容量的情况下的默认容量为11，而HashMap为16，Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。因为HashTable是直接使用除留余数法定位地址。且Hashtable计算hash值，直接用key的hashCode()。**

还要注意：前面说了Hashtable中key和value都不允许为null，而HashMap中key和value都允许为null（key只能有一个为null，而value则可以有多个为null）。但如在Hashtable中有类似put(null,null)的操作，编译同样可以通过，因为key和value都是Object类型，但运行时会抛出NullPointerException异常，这是JDK的规范规定的。

最后针对扩容：**Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。**

**下面是几个常见的笔试，面试题**

* **HashTable和HashMap的区别有哪些？**

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

理解HashMap是Hashtable的轻量级实现（非线程安全的实现，hashtable是非轻量级，线程安全的），都实现Map接口，主要区别在于：

1、由于HashMap非线程安全，在只有一个线程访问的情况下，效率要高于HashTable

2、HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。

3、HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey。因为contains方法容易让人引起误解。

4、Hashtable继承自陈旧的Dictionary类，而HashMap是Java1.2引进的Map 的一个实现。

5、Hashtable和HashMap扩容的方法不一样，HashTable中hash数组默认大小11，扩容方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数，增加为原来的2倍，没有加1。

6、两者通过hash值散列到hash表的算法不一样，HashTbale是古老的除留余数法，直接使用hashcode，而后者是强制容量为2的幂，重新根据hashcode计算hash值，在使用hash 位与 （hash表长度 – 1），也等价取膜，但更加高效，取得的位置更加分散，偶数，奇数保证了都会分散到。前者就不能保证。

7、另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。

* + fail-fast和iterator迭代器相关。如果某个集合对象创建了Iterator或者ListIterator，然后其它的线程试图“结构上”更改集合对象，将会抛出ConcurrentModificationException异常。但其它线程可以通过set()方法更改集合对象是允许的，因为这并没有从“结构上”更改集合。但是假如已经从结构上进行了更改，再调用set()方法，将会抛出IllegalArgumentException异常。
  + 结构上的更改指的是删除或者插入一个元素，这样会影响到map的结构。
  + 该条说白了就是在使用迭代器的过程中有其他线程在结构上修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。
* **为什么HashMap是线程不安全的，实际会如何体现？**

第一，如果多个线程同时使用put方法添加元素

假设正好存在两个put的key发生了碰撞(hash值一样)，那么根据HashMap的实现，这两个key会添加到数组的同一个位置，这样最终就会发生其中一个线程的put的数据被覆盖。

第二，如果多个线程同时检测到元素个数超过数组大小\*loadFactor

这样会发生多个线程同时对hash数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给table，也就是说其他线程的都会丢失，并且各自线程put的数据也丢失。**且会引起死循环的错误。**

具体细节上的原因，可以参考：[不正当使用HashMap导致cpu 100%的问题追究](https://link.zhihu.com/?target=http%3A//ifeve.com/hashmap-infinite-loop/" \t "_blank)

* **能否让HashMap实现线程安全，如何做？**

1、直接使用Hashtable，但是当一个线程访问HashTable的同步方法时，其他线程如果也要访问同步方法，会被阻塞住。举个例子，当一个线程使用put方法时，另一个线程不但不可以使用put方法，连get方法都不可以，效率很低，现在基本不会选择它了。

2、HashMap可以通过下面的语句进行同步：

Collections**.**synchronizeMap**(**hashMap**);**

3、直接使用JDK 5 之后的 ConcurrentHashMap，如果使用Java 5或以上的话，请使用ConcurrentHashMap。

* **Collections.synchronizeMap(hashMap);又是如何保证了HashMap线程安全？**

直接分析源码吧

*// synchronizedMap方法*

**public** **static** **<**K**,**V**>** Map**<**K**,**V**>** **synchronizedMap(**Map**<**K**,**V**>** m**)** **{**

**return** **new** SynchronizedMap**<>(**m**);**

**}**

*// SynchronizedMap类*

**private** **static** **class** **SynchronizedMap<**K**,**V**>**

**implements** Map**<**K**,**V**>,** Serializable **{**

**private** **static** **final** **long** serialVersionUID **=** 1978198479659022715L**;**

**private** **final** Map**<**K**,**V**>** m**;** *// Backing Map*

**final** Object mutex**;** *// Object on which to synchronize*

SynchronizedMap**(**Map**<**K**,**V**>** m**)** **{**

**this.**m **=** Objects**.**requireNonNull**(**m**);**

mutex **=** **this;**

**}**

SynchronizedMap**(**Map**<**K**,**V**>** m**,** Object mutex**)** **{**

**this.**m **=** m**;**

**this.**mutex **=** mutex**;**

**}**

**public** **int** **size()** **{**

**synchronized** **(**mutex**)** **{return** m**.**size**();}**

**}**

**public** **boolean** **isEmpty()** **{**

**synchronized** **(**mutex**)** **{return** m**.**isEmpty**();}**

**}**

**public** **boolean** **containsKey(**Object key**)** **{**

**synchronized** **(**mutex**)** **{return** m**.**containsKey**(**key**);}**

**}**

**public** **boolean** **containsValue(**Object value**)** **{**

**synchronized** **(**mutex**)** **{return** m**.**containsValue**(**value**);}**

**}**

**public** V **get(**Object key**)** **{**

**synchronized** **(**mutex**)** **{return** m**.**get**(**key**);}**

**}**

**public** V **put(**K key**,** V value**)** **{**

**synchronized** **(**mutex**)** **{return** m**.**put**(**key**,** value**);}**

**}**

**public** V **remove(**Object key**)** **{**

**synchronized** **(**mutex**)** **{return** m**.**remove**(**key**);}**

**}**

*// 省略其他方法*

**}**

从源码中看出 synchronizedMap()方法返回一个SynchronizedMap类的对象，而在SynchronizedMap类中使用了synchronized来保证对Map的操作是线程安全的，故效率其实也不高。

* **为什么HashTable的默认大小和HashMap不一样？**

前面分析了，Hashtable 的扩容方法是乘2再+1，不是简单的乘2，故hashtable保证了容量永远是奇数，结合之前分析hashmap的重算hash值的逻辑，就明白了，因为在数据分布在等差数据集合(如偶数)上时，如果公差与桶容量有公约数 n，则至少有(n-1)/n 数量的桶是利用不到的，故之前的hashmap 会在取模（使用位与运算代替）哈希前先做一次哈希运算，调整hash值。这里hashtable比较古老，直接使用了除留余数法，那么就需要设置容量起码不是偶数（除（近似）质数求余的分散效果好）。而JDK开发者选了11