

# 集合-HashMap



## HashMap

### 存储

- 数组+链表；采用Entry数组来存储Key-Value对，每一个键值对组成了一个Entry实体，Entry类实际上是一个单向的链表结构，它具有Next指针，可以连接下一个Entry实体。
- 在JDK8采用了node节点的形式来存储
- 数组是为了确定桶的位置，链表是解决冲突之后的问题

### HashMap结构（源码）

hash是一种查找速率很高的算法，在HashMap的设计中最主要的问题就是降低由于Hash冲突而造成的性能降低

### HashMap的设计思想

方案一：Hash寻址方法降低冲突。

- hash值:  $(h = \text{key.hashCode()}) \wedge (h \ggg 16)$ , 让hash值的高16位异或低16位; 这么做的原因是找桶的位置是通过和  $(n-1)$  与运算, 而长度一般不会太大, 导致高位的影响不大, 为了让高低位参与进来从而减少冲突, 让高低16位异或运算, 即高低位都参与寻址运算
- 取模的优化:  $\text{hash} \& (n-1)$ ; 因为长度为2的次幂,  $n-1$ 之后就全部二进制位都是1, 能有效降低与运算之后的重复概率; 不直接使用取模是因为两者功能一样, 位移元素效率更高

方案二：数组扩容

- 数组初始化长度为16, 如果bucket满了(超过load factor\*current capacity), 就要resize扩容。load factor为0.75, 为了最大程度避免哈希冲突
- 扩容是2的次幂, 为了 $n-1$ 之后与运算的时候减少冲突。resize创建一个新的空数组, 长度是原数组的2倍。
- rehash遍历原数组, 把所有的节点重新Hash到新数组。
- 扩容后, 元素要么是在原位置, 要么是在原位置再移动2次幂的位置, 且链表顺序不变。

方案三：冲突之后的解决方法-链表法

- 在链表长度大于8时转化为红黑树, 小于6时重新变回链表
- 不直接使用红黑树的原因: 红黑树需要左旋右旋变色等操作来保持平衡, 小于8个的时候链表已经可以保证性能了

## 并发扩容死循环问题

- 两个线程同时扩容, 第一个做了标记之后中断, 然后第二线程完成了扩容
- 第一个再执行就会出现死循环, 因为线程二已经把链表顺序调转了, 而线程1标记的仍然是旧的顺序。
- jdk1.7使用的是头插法, 扩容的时候从头节点开始, 顺序翻转, 所以出现死循环扩容问题
- jdk1.8之后改用了尾插法
- 使用头插会改变链表的上的顺序, 但是如果使用尾插, 在扩容时会保持链表元素原本的顺序, 就不会出现链表成环的问题了。

## hash冲突常用解决办法

- 比较出名的有四种(1)开放定址法(2)链地址法(3)再哈希法(4)公共溢出区域法
- 链表法: 红黑树
- 再hash法: 布隆过滤器
- 开放寻址法
  - 线性探测: 被占用之后往后查找空闲位置

- 二次探测：步长为2次方
- 双重探测：多个hash函数

## ArrayList，底层是数组，查找也快，不用ArrayList的原因

- 因为采用基本数组结构，扩容机制可以自己定义，HashMap中数组扩容刚好是2的次幂，在做取模运算的效率高。而ArrayList的扩容机制是1.5倍扩容
- 而我们不需要ArrayList的其他操作，简单原则直接用数组

## Hash Map的的get/put过程

### 1. PUT过程

- 对key的hashCode()做hash运算，计算index;
- 如果没碰撞直接放到bucket里;
- 如果碰撞了，以链表的形式存在buckets后;
- 如果碰撞导致链表过长(大于等于TREEIFY\_THRESHOLD)，就把链表转换成红黑树(JDK1.8中的改动);
- 如果节点已经存在就替换old value(保证key的唯一性)
- 如果bucket满了(超过load factor\*current capacity)，就要resize。

### 2. get元素的过程

- 对key的hashCode()做hash运算，计算index;
- 如果在bucket里的第一个节点里直接命中，则直接返回;
- 如果有冲突，则通过key.equals(k)去查找对应的Entry;
- 若为树，则在树中通过key.equals(k)查找， $O(\log n)$ ;
- 若为链表，则在链表中通过key.equals(k)查找， $O(n)$ 。

## jdk1.8中修改了什么

- 由数组+链表的结构改为数组+链表+红黑树。
- 优化了高位运算的hash算法： $h \oplus (h \gg 16)$
- 扩容后，元素要么是在原位置，要么是在原位置再移动2次幂的位置，且链表顺序不变。

## 高并发时HashMap存在的问题

- (1) 多线程扩容，引起的死循环问题
- (2) 多线程put的时候可能导致元素丢失
- (3) put非null元素后get出来的却是null

在jdk1.8中，死循环问题已经解决。其他两个问题还是存在。

怎么解决这些问题的

ConcurrentHashMap，Hashtable等线程安全等集合类。

## 关于HashMap的key的一系列问题

### 1. 键可以为Null值？

- 可以，key为null的时候，hash算法最后的值以0来计算，也就是放在数组的第一个位置

### 2. 一般用什么作为HashMap的key？

- 一般用Integer、String这种不可变类当HashMap当key，而且String最为常用。
- (1) 因为字符串是不可变的，所以在它创建的时候hashCode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。
- (2) 因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的,这些类已经很规范的覆写了hashCode()以及equals()方法。

### 3. 用可变类当HashMap的key有什么问题？

- hashCode可能发生改变，导致put进去的值，无法get出

### 4. 如果让你实现一个自定义的class作为HashMap的key该如何实现？

两个方向

- 1) 重写hashCode和equals方法注意什么？
  - 两个对象相等，hashCode一定相等；对象不等，hashCode不一定不等
  - hashCode相等，两个对象不一定相等；hashCode不等，两个对象一定不等
- 2) 如何设计一个不变类

## ConcurrentHashMap

### hashmap并发问题

- 扩容死循环（jdk1.8解决）
- put元素被覆盖导致丢失
- put非空元素get到的是null

HashTable和Collections.synchronizedMap比较

性能差，无论是读操作还是写操作都会给整个集合加锁

### 1. Collections.synchronizedMap

- 在SynchronizedMap内部维护了一个普通对象Map，还有排斥锁mutex

- 我们在调用构造方法的时候就需要传入一个Map，可以看到有两个构造器
- 如果你传入了mutex参数，则将对象排斥锁赋值为传入的对象。
- 如果没有，则将对象排斥锁赋值为this，即调用synchronizedMap的对象，就是上面的Map。
- 创建出synchronizedMap之后，再操作map的时候，就会对方法上锁，如图全是🔒

## 2. Hashtable

- 对数据操作的时候都是synchronized加锁

# HashTable和HashMap的差别

- 线程安全和线程不安全
- Hashtable 是不允许键或值为 null 的，HashMap 的键值则都可以为 null。
  - Hashtable在我们put 空值的时候会直接抛空指针异常，但是HashMap却做了特殊处理。
  - 这是因为Hashtable使用的是安全失败机制（fail-safe），这种机制会使你此次读到的数据不一定是最新的数据。
- 迭代器不同
  - HashMap 中的 Iterator 迭代器是 fail-fast 的，而 Hashtable 的 Enumerator 不是 fail-fast 的。
- 所以，当其他线程改变了HashMap 的结构，如：增加、删除元素，将会抛出 ConcurrentModificationException 异常，而 Hashtable 则不会。
- 初始化容量不同
  - HashMap 的初始容量为：16，Hashtable 初始容量为：11，两者的负载因子默认都是：0.75。
- 扩容机制不同
  - 当现有容量大于总容量 \* 负载因子时，HashMap 扩容规则为当前容量翻倍，Hashtable 扩容规则为当前容量翻倍 + 1。
- 如果你使用null值，就会使得其无法判断对应的key是不存在还是为空，因为你无法再调用一次 contain(key) 来对key是否存在进行判断，ConcurrentHashMap同理

# ArrayList

ArrayList是一种以数组实现的List，与数组相比，它具有动态扩展的能力，因此也可称之为动态数组。

## 初始化

- 不传入参数，容量是DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA
- 第一次使用时变成defaultCapacity，也就是10

## 操作

- ArrayList添加/删除中间元素比较慢，因为要搬移元素，平均时间复杂度为 $O(n)$ ；
- 从尾部添加/删除元素极快，时间复杂度为 $O(1)$ ；

扩容（ArrayList不会进行缩容；）

- 新容量是老容量的1.5倍，如果加了这么多容量发现比需要的容量还小，则以需要的容量为准；
- 扩容时机：元素数量大于总容量时
- 创建新容量的数组并把老数组拷贝到新数组；