链滴

Java 集合框架

作者：someone33881

原文链接：https://ld246.com/article/1544927645163

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20180821.jpg?imageView2/1/w/960/h/520/interlace/1/q/100">
    本文主要是记录在学习 Java 集合框架过程中的一些知识点备忘!
/p>
20181218
1、List VS Set VS Map

List:核心区别在于有序性，该接口存储一组不唯一（可以有多个元素引用相同的对象）、有序的
象
Set:核心区别在于唯一性，该接口是不允许元素重复的集合，即不会有多个元素引用相同的对象</
i>
Map:核心趋避在于键值对存储，搜索效率较高；Map 会维护与 Key 有关联的值，两个 Key 可
引用相同的对象，但 Key 不能重复，典型的 Key 是 String 类型，也可以是任何对象

2、Arraylist VS LinkedList
Arraylist 底层使用的是数组（=> 存储、读取效率高，插入删除特定位置效率低【时间复杂
浸塑为 O（n）】）；LinkedList 底层使用的是双向循环链表数据结构（插入删除特定位置侠侣特别
【时间复杂度近似为 O（1）】）
3、Arraylist VS Vector
20181217
1、HashMap 为什么是线程不安全的？
在多线程下，进行 put 操作可能会导致 HashMap 死循环问题，原因在于 HashMap 的扩容 resi
e()方法；
这是由于扩容是新建一个数组，复制原数据到数组；又由于数组下标挂有链表，因此也需要复制
表，但是多线程操作可能导致出现环形链表，例如：
若 2 个线程同时扩容，比如线程 1 先将 A 复制到新的 hash 表中，然后接着复制 B 到链头,本来 B
next = null 到此也就结束了（跟线程 2 过程一样）；但是，由于线程 2 扩容的原因，将 B.next = A,
续复制 A，让 A.next=B,由此出现 B.next=A;A.next=B
（线程 2：将**0-A->**B->NULL =》0-B->A->NULL，则线程 1：0->B-><
trong>A->B）
=》JDK1.8 中已经解决了死循环问题(在 resize 方法中，声明两个引用地址，维护两个链表，依
在末端添加新元素，在多线程操作情况下，无非是第二个线程重复第一个线程一模一样的操作而已)，
然多线程 put 操作不会导致死循环问题，但依然有其他的弊端如数据丢失等问题，因此多线程情况下
是应该使用 ConcurrentHashMap
2、HashMap VS HashSet
HashSet 底层是基于 HashMap 实现的，HashSet 中的方法除了 clone\writeObject\readObjec
方法外，其他方法都是直接调用 HashMap 中的方法的

HashMap 实现了 Map 接口，HashSet 实现了 Set 接口
HashMap 存储键值对，HashSet 仅存储对象
HashMap 调用 put 方法向 map 中添加元素，HashSet 调用 add 方法向 Set 中添加元素
HashMap 使用键 Key 计算 HashCode；HashSet 使用成员对象来计算 hashcode 值
HashMap 相对于 HashSet 较快，因为它是使用唯一的键获取对象

3、ConcurrentHashMap VS Hashtable
二者的区别主要体现于线程安全的实现方式上不同

底层数据结构：JDK1.7 中的 ConcurrentHashMap 底层采用分段的数组 + 链表</str
ng>实现，JDK1.8 则采用的是跟 HashMap1.8 的结构一样，即数组 + 链表/红黑二叉树</
trong>；Hashtable 和 JDK1.8 之前的 HashMap 底层数据结构类似也是采用的数组 + 链表形式，
组是 HashMap 的主体，链表则主要是为了解决哈希冲突而存在的
实现线程安全的方式(重要)：（1）JDK1.7 中，ConcurrentHashMap(分段锁)对整个桶数组进行
割分段（Segment），每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就
会存在锁竞争，提高并发访问率（默认分配 16 个 Segment，笔 Hashtable 效率提高 16 倍）；而在

DK1.8 中则摒弃了 Segment 的概念，直接使用 Node 数组 + 链表 + 红黑树的数据结构来实现，并控制使用 synchronized 和 CAS 来操作（JDK1.6 以后对 synchronized 锁做了很多优化），整个看来就像是优化过且线程安全的 HashMap，尽管在 DJK1.8 中还能够看得到 Segment 的数据结构，已经简化了属性只是为了兼容旧版本；（2）Hashtable(同一把锁)：使用 synchronized 来保证线程全，效率非常低下；当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询态，如使用 put 添加元素，则另一个线程不能使用 put 添加元素也不能使用 get，竞争会越来越激烈率越低

4、ConcurrentHashMap 线程安全的具体实现方式/底层实现原理

JDK1.7：将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中个段数据时，其他段的数据也能被其他线程访问；

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成；Segment 实现了eentrantLock,是一种可重入锁，扮演锁的角色；HashEntry 用于存储键值对数据
一个 ConcurrentHashMap 中包含一个 Segment 数组，Segment 的结构与 HashMap 类似，一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改，必须首先获得对应的 Segment 的锁

JDK1.8:ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并安全，数据结构跟 HashMap1.8 类似，数组 + 链表/红黑二叉树

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并，效率又提升 N 倍
5、集合框架底层数据结构总结

Collection

List:
<pre><code class="highlight-chroma">(1)Arraylist:Object数组
(2)Vector:Object 组
(3)LinkedList:双向表（JDK1.6之前为循环链表，JDK1.7取消了循环）
</code></pre>
Set:
<pre><code class="highlight-chroma">(1)HashSet（无序，唯一）：基于HashMap实现，底层采用HashMap来保存元素
(2)LinkedHashSet 继承于HashSet，且其内部是通过LinkedHashMap实现的
(3)TreeSet(有序，一)：红黑树（自平衡的排序二叉树）
</code></pre>

Map
(1)HashMap:JDK1.8 之前是由数组 + 链表组成，数组是其主体，链表主要是为了解决哈希冲突存在的（拉链法解决冲突）；JDK1.8 之后在解决哈希冲突时，增加了红黑树数据结构即当链表长度于阈值（默认 8）时，则会将链表转化为红黑树以减少搜索时间
(2)LinkedHashMap:继承于 HashMap，底层仍然是基于拉链式散列结构即数组 + 链表/红黑树，在基础之上增加了一条双向链表，使得该结构可以保持键值对的插入顺序，同时通过链表进行相应的操，实现了访问顺序相关逻辑

(3)Hashtable:数据 + 链表组成，数组是其主体，链表则主要是为了解决哈希冲突而存在的
(4)TreeMap:红黑树（自平衡的排序二叉树）

20181216
1、ArrayList VS LinkedList

是否线程安全：二者均是不同步的，即不保证线程安全；
底层数据结构：ArrayList - Object 数组，LinkedList - 双向链表数据结构（JDK1.6 之前为循环表，JDK1.7 取消了循环，注意双向链表和双向循环链表的区别）
插入和删除是否受元素位置的影响：（1）数组，因此插入和删除元素的时间复杂度受元素位置影响，近似为 O(n);(2)链表，因此插入和删除元素的时间复杂度不受元素位置的影响，近似为 O(1)</l
>
是否支持快速随机访问：LinkedList 不支持高效的随机元素访问，而 ArrayList 支持，直接通过素的序号快速获取元素对象
内存空间占用：ArrayList 的空间浪费主要是 list 列表的结尾会预留一定的容量空间，而 LinkedLi t 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和接前驱以及数据）

=>

RandomAccess 接口：该接口中无任何定义，因此只是一个标识，即标识实现这个接口的类具随机访问功能！
binarySearch()方法：该方法会判断传参 List 是否是 RandomAccess 的实例，若是则调用 index dBinarySearch 方法，否则调用 iteratorBinarySearch 方法

=>

ArrayList 实现了 RandomAccess 接口，LinkedList 没有实现；
数组天然支持随机访问，时间复杂度 O(1),因此称为快速随机访问；链表需要遍历到特定位置才访问特定位置的元素，时间复杂度为 O(n)，所以不支持快速随机访问
ArrayList 是实现了 RandomAccess 接口，是表明了其具有快速随机访问功能，该接口仅是标识并不是说 ArrayList 实现了该接口才具有快速随机访问功能的

=>

实现了 RandomAccess 接口的 List，优先使用普通 for 循环，其次是 foreach

未实现 RandomAccess 接口的 List，优先选择 iterator 遍历(foreach 遍历底层也是通过 iterator 实现的)，大 size 的 List 数据不要使用普通 for 循环

双向链表：也即双链表，是链表的一种，它的每个数据节点均有两个指针，分别指向直接后继和接前驱；因此，从双向链表中的任意一个节点开始，均可以很方便地访问它的前驱节点和后继节点，般都是构造双向循环链表，JDK1.6 之前的 LinkedList 底层使用的就是双向循环链表

2、ArrayList VS Vector

Vector 类的所有方法均是同步的，两个线程可以安全地访问同一个 Vector 对象，但是一个线程问 Vector 需要在同步操作上耗费大量的时间
ArrayList 不是同步的，不需要保证线程安全时建议使用 ArrayList

3、HashMap 的底层实现

JDK1.8 之前：

底层是“数组 + 链表”数据结构，即链表散列；
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（n 即数组的长度），如果当前位置存在元素的话则判断该元素与要存入 h 元素的 ash 值以及 key 是否相同，如果相同的话则直接覆盖，否则不相同则通过拉链法解决冲突
扰动函数：也就是 HashMap 的 hash 方法，该方法即扰动函数主要是为了防止一些实现比较差 hashCode 方法，以减少碰撞
hash 方法源码：
<pre><code class="highlight-chroma">//jdk1.8方法相较于jdk1.7更加简化，但是原理不变
static final int hash(Object key) {
    int h;
    // key.hashCode()：返回散列值也就是hashcode
    // ^ ：按位异

    // >>>:无符号右移，忽略符号位，空位都以0补齐
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

//jdk1.7，该方法能会稍差一点点，因为毕竟扰动了4次
static int hash(int) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
</code></pre>
拉链法：将链表和数组相结合，即创建一个链表数组，数组中每一格都是一个链表，若遇到 hash 冲突则将冲突的值加到链表中即可

JDK1.8 之后

在 JDK1.8 中，对于解决哈希冲突有了较大的变化，当链表长度大于阈值（默认 8），则会将链转化为红黑树，以减少搜索时间
TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层均用到红黑树，红黑树就是为了解决二查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构

《Java 8 系列之重新认识 HashMap》：https:/zhuanlan.zhihu.com/p/21673805

4、HashMap VS HashTable

线程安全：HashMap 非线程安全，HashTable 线程安全；HashTable 内部的方法基本都是经过 ynchronized 修饰的（若需要保证线程安全的话，可以使用 ConcurrentHashMap）
效率：由于线程安全的问题，HashTable 的效率比 HashMap 低一点，且 HashTable 已经基本淘汰，不要在代码中使用它
对 Null key 和 Null value 的支持：HashMap 中，null 可以作为主键，这样的键只有一个，可有一个或多个键所对应的值为 null；但是在 HashTable 中 put 进的键值只要有一个 null，则直接抛出 NullPointerException
初始容量大小&每次扩充容量大小的不同：（1）创建时若未指定初始容量值，HashTable 认初始大小为 11，每次扩充容量变为原来的 2n+1;HashMap 默认初始大小为 16，每次扩充容量变原来的 2 倍；（2）创建时若指定初始容量值，则 HashTable 会直接使用给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小（HashMap 中的 tableSizeFor 方法保证）
底层数据结构：JDK1.8 以后的 HashMap 在解决哈希冲突时，当链表长度大于阈值（默认 8）则会将链表转化为红黑树以减少搜索时间，而 HashTable 则没有这样的机制

5、HashMap 的长度为什么是 2 的幂次方
为了能够让 HashMap 存取高效，尽量减少碰撞，也即要尽量把数据分配均匀！Hash 值范围是-147483648~2147483648，共约 40 亿映射空间，只要 hash 函数映射的比较均匀松散，一般很难现碰撞，但是 40 亿长度的数组在内存中存放不下的，因此这个散列值是不能直接使用的
=> 考虑先对数组的长度进行取模运算，计算的余数用来作为存放的位置也即数组下标，即数下标的计算方法是“(n-1) & hash”，其中 n 为数组长度,这也就是为什么 HashMap 的长度是 2 的幂次方
=> 为什么是 2 的幂次方？：：取模运算，首先就是采用 % 操作进行实现，=>"取余 % 作中，在除数是 2 的幂次方时，等价于与其除数减一的与&操作，也即 hash%length == hash amp;(length-1),,这个等价的前提就是 length 是 2 的 n 次方"
=> 并且，在采用二进制位操作&，相对于 % 能够提高运算效率，这也是为什么 HashMap 的长度要是 2 的幂次方！