# ArrayList总结

（特点：扩容1.5倍、Object数组实现、序列化：readObject和writeObject、浅克隆）

## 一、继承：AbstractList类

## 二、实现：

1、List

2、RandomAccess（标记型的接口，只是说明实现类可以随机访问）

3、Cloneable（标记型的接口，说明对象可以被克隆，调用Object.clone()不会抛出CloneNotSupportedException异常）

4、java.io.Serializable(可以被序列化)

## 三、重要的成员变量

Ⅰ、private static final int DEFAULT\_CAPACITY=10//默认容量

Ⅱ、private static final Object[] EMPTY\_ELEMENTDATA= {}

Ⅲ、private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {}

Ⅳ、transient Object[] elementData;

①transient避免序列化，所以这个字段的生命周期仅存于调用者的内存中，不会写到磁盘里持久化。

②非私有化以简化嵌套类访问

V、private int size//实际大小

Ⅵ、private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE – 8//最大数组容量

Ⅶ、modCount：记录集合结构的修改次数

## 四、重要的方法

Ⅰ、public Object[] toArray()

Ⅱ、private void writeObject(java.io.ObjectOutputStream s)

Ⅲ、private void readObject(java.io.ObjectInputStream s)

## 五、遍历方式

Ⅰ、迭代器或forEach

Ⅱ、get(index)

Ⅲ、for:

# HashMap总结

## 一、特点

1、允许空键和空值

2、迭代时间基于桶数和映射数

3、两个重要的参数：初始桶数和负载因子(0.75)

4、条目数超过桶数\*负载因子，则哈希表重新哈希，且桶数\*2

5、迭代器快速失败

6、基于拉链式的散列算法实现，JDK1.8引入红黑树优化过长的链表

7、桶数超过指定参数变为红黑树并rehash，反之退为长链表

8、查找时需要判断数据结构：first instanceof TreeNode

9、序列化：readObject和writeObject

10、转为红黑树节点，链表结构仍然存在了：prev维持链表结构

## 二、重要变量

1、默认容量：static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;

2、最大容量：static final int MAXIMUM\_CAPACITY = 1 << 30;

3、负载因子：static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

4、树化负载因子：static final int TREEIFY\_THRESHOLD = 8;

5、存储数据：transient Node<K, V>[] table;

6、保存缓存的entrySet：transient Set<Map.Entry<K, V>> entrySet;

7、键值对个数：transient int size;

8、结构修改的次数：transient int modCount;

## 三、重要的方法

### 1、hash()

/\*\*

\* 高位和低位相异或

\* 不忽略高位，减少hash碰撞

\*/

static final int hash**(**Object key**)** **{**

int h**;**

**return** **(**key **==** **null)** **?** 0 **:** **(**h **=** key**.**hashCode**())** **^** **(**h **>>>** 16**);**

**}**

### 2、tableSizeFor()

/\*\*

\* 返回一个大于等于他的最小的2的指数值

\*/

static final int tableSizeFor**(**int cap**)** **{**

int n **=** cap **-** 1**;**

n **|=** n **>>>** 1**;**

n **|=** n **>>>** 2**;**

n **|=** n **>>>** 4**;**

n **|=** n **>>>** 8**;**

n **|=** n **>>>** 16**;**

**return** **(**n **<** 0**)** **?** 1 **:** **(**n **>=** MAXIMUM\_CAPACITY**)** **?** MAXIMUM\_CAPACITY **:** n **+** 1**;**

**}**

### 3、getNode()

final Node**<**K**,** V**>** getNode**(**int hash**,** Object key**)** **{**

Node**<**K**,** V**>[]** tab**;**

Node**<**K**,** V**>** first**,** e**;**

int n**;**

K k**;**

**if** **((**tab **=** table**)** **!=** **null** **&&** **(**n **=** tab**.**length**)** **>** 0 **&&**

**(**first **=** tab**[(**n **-** 1**)** **&** hash**])** **!=** **null)** **{**

**if** **(**first**.**hash **==** hash **&&** // always check first node

**((**k **=** first**.**key**)** **==** key **||** **(**key **!=** **null** **&&** key**.**equals**(**k**))))**

**return** first**;**

**if** **((**e **=** first**.**next**)** **!=** **null)** **{**

**if** **(**first **instanceof** TreeNode**)**

**return** **((**TreeNode**<**K**,** V**>)** first**).**getTreeNode**(**hash**,** key**);**

**do** **{**

**if** **(**e**.**hash **==** hash **&&**

**((**k **=** e**.**key**)** **==** key **||** **(**key **!=** **null** **&&** key**.**equals**(**k**))))**

**return** e**;**

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

**}**

**}**

**return** **null;**

**}**

### 4、putVal

final V putVal**(**int hash**,** K key**,** V value**,** boolean onlyIfAbsent**,**

boolean evict**)** **{**

Node**<**K**,** V**>[]** tab**;**

Node**<**K**,** V**>** p**;**

int n**,** i**;**

**if** **((**tab **=** table**)** **==** **null** **||** **(**n **=** tab**.**length**)** **==** 0**)**

//如果tab是空，则使用resize初始化。resize方法兼顾初始化表格，和大小不足时，进行扩容

n **=** **(**tab **=** resize**()).**length**;**

//键值对在哈希表中的位置i = (n - 1) & hash决定

**if** **((**p **=** tab**[**i **=** **(**n **-** 1**)** **&** hash**])** **==** **null)**

//桶为空，则初始化根节点

tab**[**i**]** **=** newNode**(**hash**,** key**,** value**,** **null);**

**else** **{**

Node**<**K**,** V**>** e**;**

K k**;**

**if** **(**p**.**hash **==** hash **&&**

**((**k **=** p**.**key**)** **==** key **||** **(**key **!=** **null** **&&** key**.**equals**(**k**))))**

e **=** p**;**

**else** **if** **(**p **instanceof** TreeNode**)**

e **=** **((**TreeNode**<**K**,** V**>)** p**).**putTreeVal**(this,** tab**,** hash**,** key**,** value**);**

**else** **{**

**for** **(**int binCount **=** 0**;** **;** **++**binCount**)** **{**

**if** **((**e **=** p**.**next**)** **==** **null)** **{**

p**.**next **=** newNode**(**hash**,** key**,** value**,** **null);**

// 超过了树化负载因子

**if** **(**binCount **>=** TREEIFY\_THRESHOLD **-** 1**)**

treeifyBin**(**tab**,** hash**);**

**break;**

**}**

**if** **(**e**.**hash **==** hash **&&**

**((**k **=** e**.**key**)** **==** key **||** **(**key **!=** **null** **&&** key**.**equals**(**k**))))**

**break;**

p **=** e**;**

**}**

**}**

**if** **(**e **!=** **null)** **{** // existing mapping for key

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

**if** **(!**onlyIfAbsent **||** oldValue **==** **null)**

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

afterNodeAccess**(**e**);**

**return** oldValue**;**

**}**

**}**

**++**modCount**;**

//如果长度大于容量，则扩容

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

resize**();**

afterNodeInsertion**(**evict**);**

**return** **null;**

**}**

5、treeifyBin()：树化函数

/\*\*

\* 树化改造

\* bin 的数量大于 TREEIFY\_THRESHOLD 时：

\* 如果容量小于 MIN\_TREEIFY\_CAPACITY，只会进行简单的扩容。

\* 如果容量大于 MIN\_TREEIFY\_CAPACITY ，则会进行树化改造。

\* 在给定的散列中，替换bin中的所有链接节点，除非表太小，在这种情况下，可以进行调整。

\*/

final void treeifyBin**(**Node**<**K**,** V**>[]** tab**,** int hash**)** **{**

int n**,** index**;**

Node**<**K**,** V**>** e**;**

**if** **(**tab **==** **null** **||** **(**n **=** tab**.**length**)** **<** MIN\_TREEIFY\_CAPACITY**)**

resize**();**

**else** **if** **((**e **=** tab**[**index **=** **(**n **-** 1**)** **&** hash**])** **!=** **null)** **{**

TreeNode**<**K**,** V**>** hd **=** **null,** tl **=** **null;**

// 转化为树节点的双向链表

**do** **{**

//将节点替换为TreeNode

TreeNode**<**K**,** V**>** p **=** replacementTreeNode**(**e**,** **null);**

// 记录头节点

**if** **(**tl **==** **null)**

hd **=** p**;**

// 这里其实是将单链表转化成了双向链表

**else** **{**

p**.**prev **=** tl**;**

tl**.**next **=** p**;**

**}**

tl **=** p**;**

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

// 将链表进行树化

**if** **((**tab**[**index**]** **=** hd**)** **!=** **null)**

hd**.**treeify**(**tab**);**

**}**

**}**

## 四、与Hashtable的区别

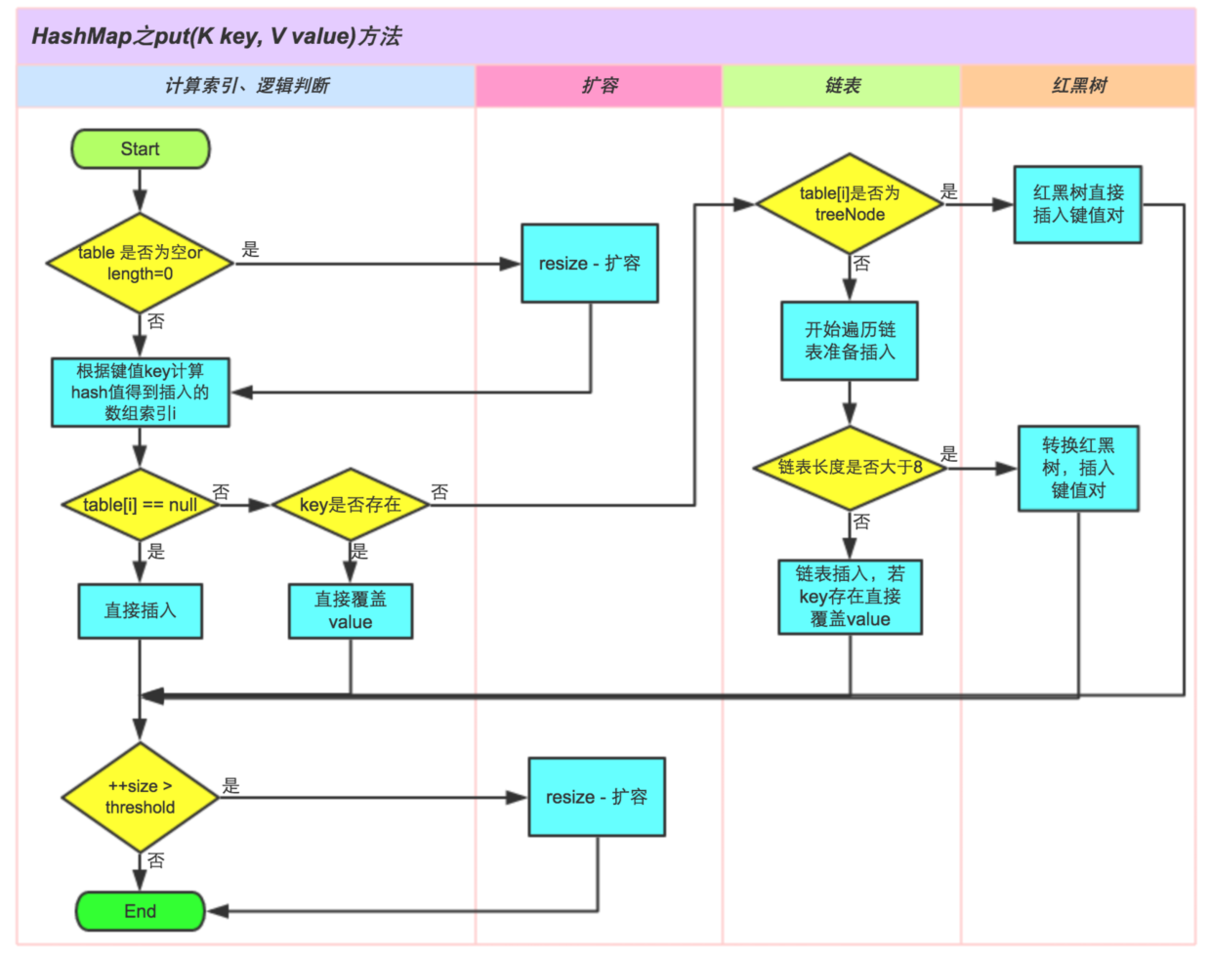
1、Hashtable是线程安全的，且不允许key、value为null。

2、Hashtable默认容量为11

3、Hashtable是直接使用key的hashCode作为hash值。

4、Hashtable取哈希桶的下标是直接用模运算%。（因为其容量不是2的n次方）

5、扩容时，新容量是原来的2倍+1



# 红黑树

## 一、特点

1、二叉搜索树

2、节点非黑即红

3、根节点是黑色

4、每个叶子节点(NIL)是黑色

5、红色节点的子节点必须是黑色的

6、 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点（这是平衡的关键）

## 二、重要的函数

### 1、treeify()：树化（hash碰撞：compareTo和引用比较）

/\*\*

\* 树化这个双向链表

\*/

final void treeify**(**Node**<**K**,** V**>[]** tab**)** **{**

TreeNode**<**K**,** V**>** root **=** **null;**

**for** **(**TreeNode**<**K**,** V**>** x **=** **this,** next**;** x **!=** **null;** x **=** next**)** **{**

next **=** **(**TreeNode**<**K**,** V**>)** x**.**next**;**

x**.**left **=** x**.**right **=** **null;**

// 初始化根节点

**if** **(**root **==** **null)** **{**

x**.**parent **=** **null;**

// 根节点为黑色

x**.**red **=** **false;**

root **=** x**;**

**}** **else** **{**

K k **=** x**.**key**;**

int h **=** x**.**hash**;**

Class**<?>** kc **=** **null;**

// 循环遍历，进行二叉搜索树的插入

**for** **(**TreeNode**<**K**,** V**>** p **=** root**;** **;** **)** **{**

// dir用来指示x节点与p的比较，-1表示比p小，1表示比p大，

// 不存在相等情况，因为HashMap中是不存在两个key完全一致的情况。

int dir**,** ph**;**

K pk **=** p**.**key**;**

**if** **((**ph **=** p**.**hash**)** **>** h**)**

dir **=** **-**1**;**

**else** **if** **(**ph **<** h**)**

dir **=** 1**;**

// 如果hash值相等，那么判断k是否实现了comparable接口，

// 如果实现了comparable接口就使用compareTo进行进行比较，

// 如果仍旧相等或者没有实现comparable接口，则在tieBreakOrder中比较

**else** **if** **((**kc **==** **null** **&&**

**(**kc **=** comparableClassFor**(**k**))** **==** **null)** **||**

**(**dir **=** compareComparables**(**kc**,** k**,** pk**))** **==** 0**)**

dir **=** tieBreakOrder**(**k**,** pk**);**

TreeNode**<**K**,** V**>** xp **=** p**;**

**if** **((**p **=** **(**dir **<=** 0**)** **?** p**.**left **:** p**.**right**)** **==** **null)** **{**

x**.**parent **=** xp**;**

**if** **(**dir **<=** 0**)**

xp**.**left **=** x**;**

**else**

xp**.**right **=** x**;**

// 进行插入平衡处理

root **=** balanceInsertion**(**root**,** x**);**

**break;**

**}**

**}**

**}**

**}**

// 将根节点设为链表的首节点

moveRootToFront**(**tab**,** root**);**

**}**

// 在插入中保持一致的顺序

static int tieBreakOrder**(**Object a**,** Object b**)** **{**

int d**;**

// 用两者的类名进行比较，如果相同则使用对象默认的hashcode（引用）进行比较

**if** **(**a **==** **null** **||** b **==** **null** **||**

**(**d **=** a**.**getClass**().**getName**().**

compareTo**(**b**.**getClass**().**getName**()))** **==** 0**)**

d **=** **(**System**.**identityHashCode**(**a**)** **<=** System**.**identityHashCode**(**b**)** **?**

**-**1 **:** 1**);**

**return** d**;**

**}**

### 2、rotateLeft()

// 左旋

static **<**K**,** V**>** TreeNode**<**K**,** V**>** rotateLeft**(**TreeNode**<**K**,** V**>** root**,**

TreeNode**<**K**,** V**>** p**)** **{**

TreeNode**<**K**,** V**>** r**,** pp**,** rl**;**

**if** **(**p **!=** **null** **&&** **(**r **=** p**.**right**)** **!=** **null)** **{**

**if** **((**rl **=** p**.**right **=** r**.**left**)** **!=** **null)**

rl**.**parent **=** p**;**

**if** **((**pp **=** r**.**parent **=** p**.**parent**)** **==** **null)**

**(**root **=** r**).**red **=** **false;**

**else** **if** **(**pp**.**left **==** p**)**

pp**.**left **=** r**;**

**else**

pp**.**right **=** r**;**

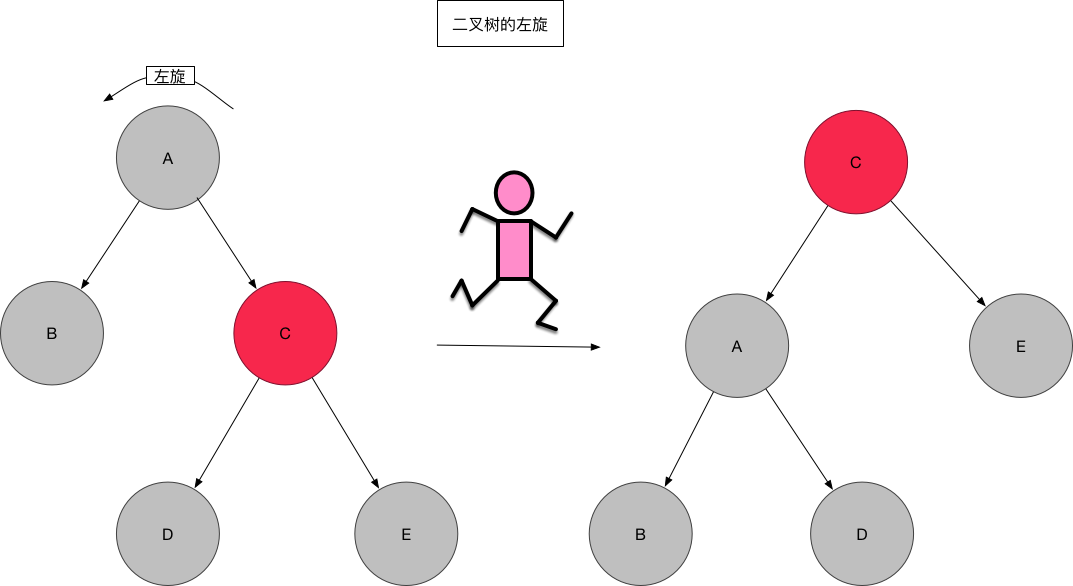
r**.**left **=** p**;**

p**.**parent **=** r**;**

**}**

**return** root**;**

**}**



### 3、rotateRight()

// 右旋

static **<**K**,** V**>** TreeNode**<**K**,** V**>** rotateRight**(**TreeNode**<**K**,** V**>** root**,**

TreeNode**<**K**,** V**>** p**)** **{**

TreeNode**<**K**,** V**>** l**,** pp**,** lr**;**

**if** **(**p **!=** **null** **&&** **(**l **=** p**.**left**)** **!=** **null)** **{**

**if** **((**lr **=** p**.**left **=** l**.**right**)** **!=** **null)**

lr**.**parent **=** p**;**

**if** **((**pp **=** l**.**parent **=** p**.**parent**)** **==** **null)**

**(**root **=** l**).**red **=** **false;**

**else** **if** **(**pp**.**right **==** p**)**

pp**.**right **=** l**;**

**else**

pp**.**left **=** l**;**

l**.**right **=** p**;**

p**.**parent **=** l**;**

**}**

**return** root**;**

**}**

