# 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（初始化桶）

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**);**

**}**

**}**

### 6、四个构造方法（只初始化参数）

public HashMap**(**int initialCapacity**,** float loadFactor**)** **{**

**if** **(**initialCapacity **<** 0**)**

**throw** **new** IllegalArgumentException**(**"Illegal initial capacity: " **+**

initialCapacity**);**

**if** **(**initialCapacity **>** MAXIMUM\_CAPACITY**)**

initialCapacity **=** MAXIMUM\_CAPACITY**;**

**if** **(**loadFactor **<=** 0 **||** Float**.**isNaN**(**loadFactor**))**

**throw** **new** IllegalArgumentException**(**"Illegal load factor: " **+**

loadFactor**);**

**this.**loadFactor **=** loadFactor**;**

**this.**threshold **=** tableSizeFor**(**initialCapacity**);**

**}**

public HashMap**(**int initialCapacity**)** **{**

**this(**initialCapacity**,** DEFAULT\_LOAD\_FACTOR**);**

**}**

public HashMap**()** **{**

**this.**loadFactor **=** DEFAULT\_LOAD\_FACTOR**;** // all other fields defaulted

**}**

public HashMap**(**Map**<?** **extends** K**,** **?** **extends** V**>** m**)** **{**

**this.**loadFactor **=** DEFAULT\_LOAD\_FACTOR**;**

putMapEntries**(**m**,** **false);**

**}**

### 7、resize()：初始化table数组

/\*\*

\* 初始化或增加表的大小（扩容）。

\* 如果null，则按照在字段阈值中持有的初始容量目标分配。

\* 否则，因为我们使用的是两种扩展的功能，每个箱子中的元素必须保持在相同的索引上，或者在新表中使用两个偏移量。

\*/

final Node<K, V>[] resize() {

Node<K, V>[] oldTab = table;

int oldCap = (oldTab == null) ? 0 : oldTab.length;

int oldThr = threshold;

int newCap, newThr = 0;

if (oldCap > 0) {

// 最大的情况，无法扩容

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

// 扩容为原来的2倍

} else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

} else if (oldThr > 0) // initial capacity was placed in threshold

newCap = oldThr;

// 链表为空，使用默认值

else { // zero initial threshold signifies using defaults

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int) (DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

// 计算容量

if (newThr == 0) {

float ft = (float) newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float) MAXIMUM\_CAPACITY ?

(int) ft : Integer.MAX\_VALUE);

}

threshold = newThr;

@SuppressWarnings({"rawtypes", "unchecked"})

Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];

table = newTab;

if (oldTab != null) {

// 把每个桶中的元素，分为原位置和原位置+oldCap

for (int j = 0; j < oldCap; ++j) {

Node<K, V> e;

if ((e = oldTab[j]) != null) {

oldTab[j] = null;

if (e.next == null)

newTab[e.hash & (newCap - 1)] = e;

else if (e instanceof TreeNode)

((TreeNode<K, V>) e).split(this, newTab, j, oldCap);

else { // preserve order

Node<K, V> loHead = null, loTail = null;

Node<K, V> hiHead = null, hiTail = null;

Node<K, V> next;

do {

// 链表重新hash

next = e.next;

// 还在原索引位置

if ((e.hash & oldCap) == 0) {

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

} else {// 位置变为原索引+oldCap

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

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

// 原索引放到bucket里

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

// 原索引+oldCap放到bucket里

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

}

}

}

return newTab;

}

## 四、与Hashtable的区别

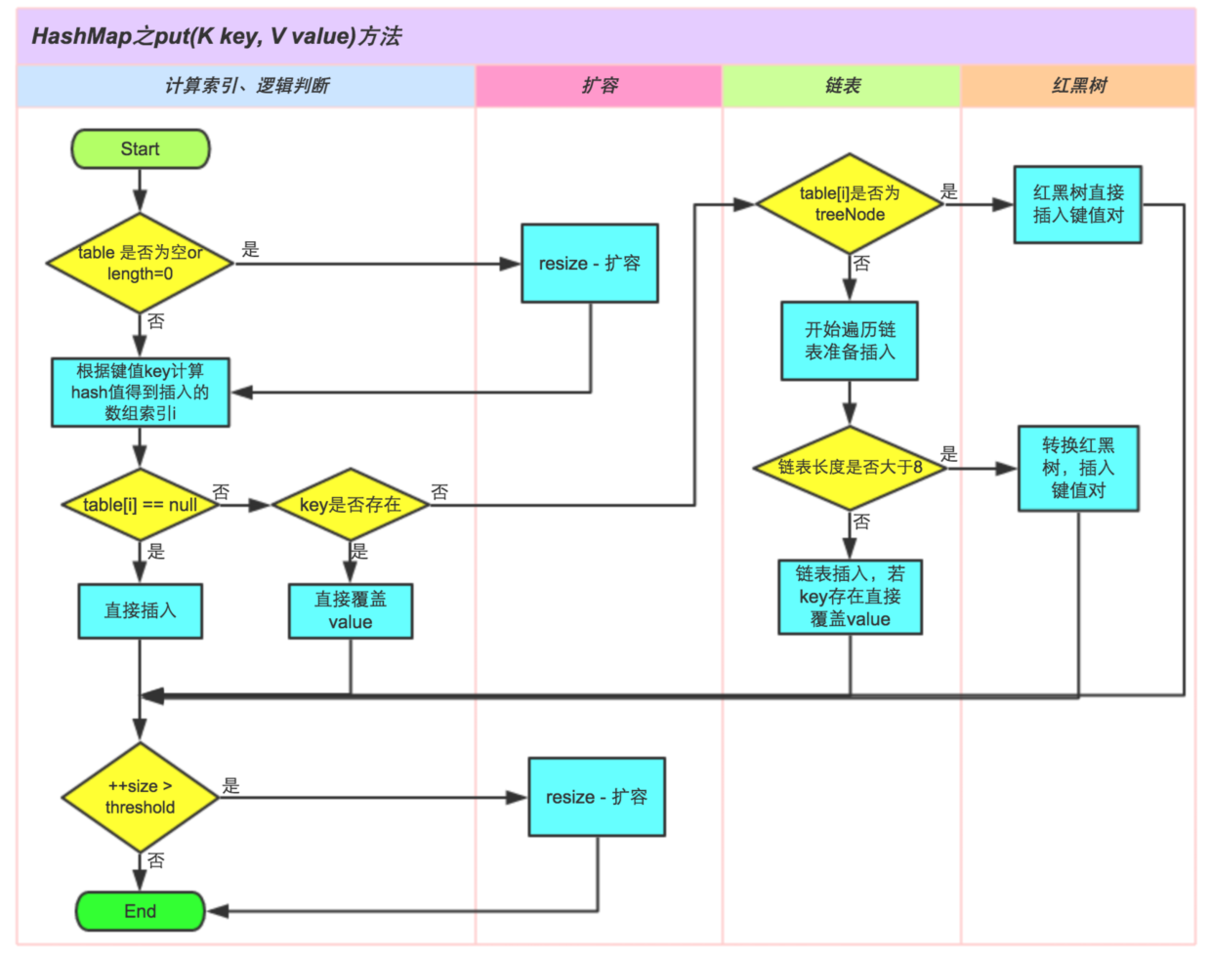
1、Hashtable是线程安全的，且不允许key、value为null。

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**;**

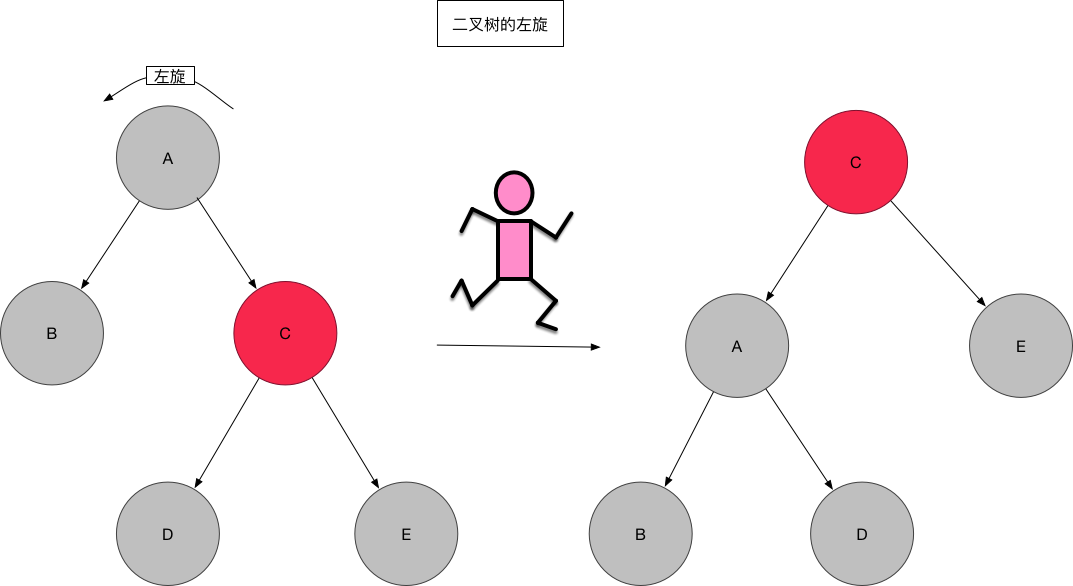
r**.**left **=** p**;**

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**;**

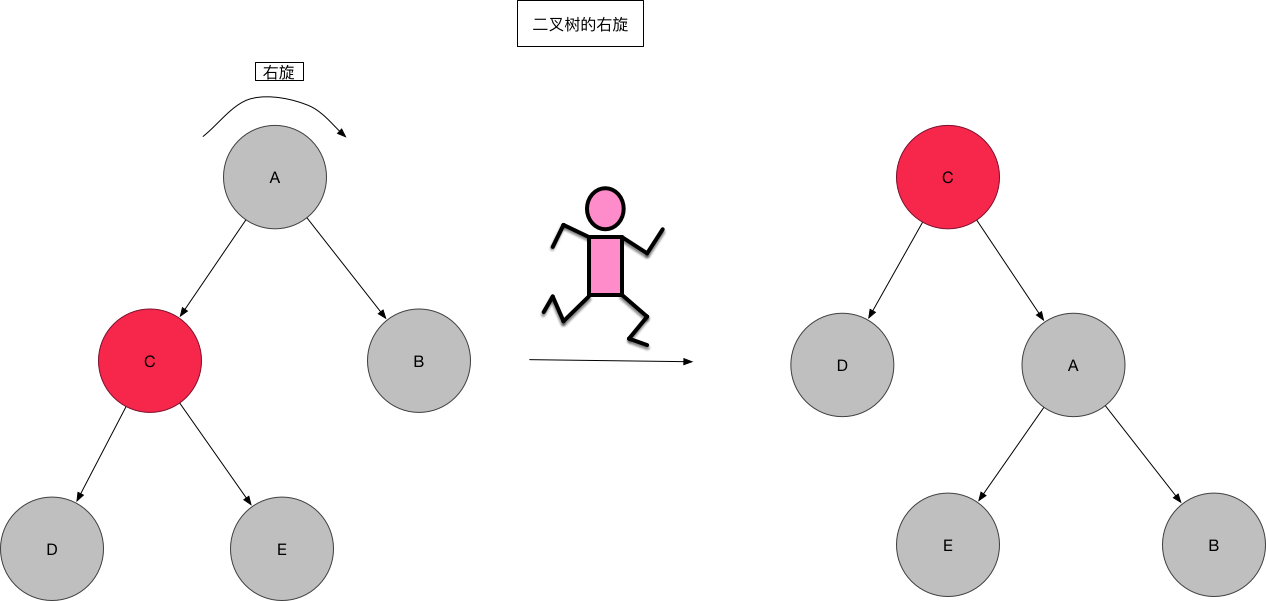
l**.**right **=** p**;**

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

**}**

**return** root**;**

**}**



# ConcurrentHashMap

1、在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设为16），及任意数量线程的读操作。

2、CAS和synchronized保证并发更新安全

3、键值均不能为null

## 一、重要变量

1、transient volatile Node<K,V>[] table;// 默认大小为16

2、private transient volatile Node<K,V>[] nextTable;// 扩容时新生成的数组

3、private transient volatile int sizeCtl;

默认为0

-1：table正在初始化

-N：有N-1个线程正在进行扩容操作

其余情况：

(1)table未初始化，表示table需要初始化的大小

(2)table初始化完成，表示table的容量。0.75即（n-（n>>>2））

4、Node：保存key，value及hash值的数据结构

5、ForwardingNode：hash为-1，存储nextTable的引用。只有当table发生扩容时，才会发挥作用，作为一个占位符放在table中表示当前节点为null或已经被移动。

## 二、重要方法

### 1、四个构造方法（只初始化参数，不分配空间，防止首次操作就需要初始化）

public ConcurrentHashMap**()** **{**

**}**

public ConcurrentHashMap**(**int initialCapacity**)** **{**

**if** **(**initialCapacity **<** 0**)**

**throw** **new** IllegalArgumentException**();**

int cap **=** **((**initialCapacity **>=** **(**MAXIMUM\_CAPACITY **>>>** 1**))** **?**

MAXIMUM\_CAPACITY **:**

tableSizeFor**(**initialCapacity **+** **(**initialCapacity **>>>** 1**)** **+** 1**));**

**this.**sizeCtl **=** cap**;**

**}**

public ConcurrentHashMap**(**Map**<?** **extends** K**,** **?** **extends** V**>** m**)** **{**

**this.**sizeCtl **=** DEFAULT\_CAPACITY**;**

putAll**(**m**);**

**}**

public ConcurrentHashMap**(**int initialCapacity**,** float loadFactor**)** **{**

**this(**initialCapacity**,** loadFactor**,** 1**);**

**}**

### 2、putVal()

final V putVal**(**K key**,** V value**,** boolean onlyIfAbsent**)** **{**

**if** **(**key **==** **null** **||** value **==** **null)** **throw** **new** NullPointerException**();**

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

int binCount **=** 0**;**

**for** **(**Node**<**K**,**V**>[]** tab **=** table**;;)** **{**

Node**<**K**,**V**>** f**;** int n**,** i**,** fh**;**

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

tab **=** initTable**();**// 初始化table数组

**else** **if** **((**f **=** tabAt**(**tab**,** i **=** **(**n **-** 1**)** **&** hash**))** **==** **null)** **{**// 当前桶为空

**if** **(**casTabAt**(**tab**,** i**,** **null,**

**new** Node**<**K**,**V**>(**hash**,** key**,** value**,** **null)))**// cas操作初始化桶

**break;** // no lock when adding to empty bin

**}**

**else** **if** **((**fh **=** f**.**hash**)** **==** MOVED**)**// 当前Map在扩容，先协助扩容，再更新值

tab **=** helpTransfer**(**tab**,** f**);**

**else** **{**// hash冲突

V oldVal **=** **null;**

synchronized **(**f**)** **{**

**if** **(**tabAt**(**tab**,** i**)** **==** f**)** **{**// 链表头节点

**if** **(**fh **>=** 0**)** **{**

binCount **=** 1**;**

**for** **(**Node**<**K**,**V**>** e **=** f**;;** **++**binCount**)** **{**

K ek**;**

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

**((**ek **=** e**.**key**)** **==** key **||**

**(**ek **!=** **null** **&&** key**.**equals**(**ek**))))** **{**// 节点已存在，则覆盖值

oldVal **=** e**.**val**;**

**if** **(!**onlyIfAbsent**)**

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

**break;**

**}**

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

**if** **((**e **=** e**.**next**)** **==** **null)** **{**// 节点不存在，添加到链表尾部

pred**.**next **=** **new** Node**<**K**,**V**>(**hash**,** key**,**

value**,** **null);**

**break;**

**}**

**}**

**}**

**else** **if** **(**f **instanceof** TreeBin**)** **{**// 红黑树结构

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

binCount **=** 2**;**

**if** **((**p **=** **((**TreeBin**<**K**,**V**>)**f**).**putTreeVal**(**hash**,** key**,**

value**))** **!=** **null)** **{**

oldVal **=** p**.**val**;**

**if** **(!**onlyIfAbsent**)**

p**.**val **=** value**;**

**}**

**}**

**}**

**}**

**if** **(**binCount **!=** 0**)** **{**

**if** **(**binCount **>=** TREEIFY\_THRESHOLD**)**// 链表节点超过了8，链表转换为红黑树

treeifyBin**(**tab**,** i**);**

**if** **(**oldVal **!=** **null)**

**return** oldVal**;**

**break;**

**}**

**}**

**}**

addCount**(**1L**,** binCount**);**// 统计节点个数，检查是否需要resize

**return** **null;**

**}**

### 3、tabAt()

调用Unsafe的getObjectVolatile直接获取指定内存的数据。虽然table是volatile修饰的，但不能保证线程每次都拿到table中的最新元素。

static final **<**K**,**V**>** Node**<**K**,**V**>** tabAt**(**Node**<**K**,**V**>[]** tab**,** int i**)** **{**

**return** **(**Node**<**K**,**V**>)**U**.**getObjectVolatile**(**tab**,** **((**long**)**i **<<** ASHIFT**)** **+** ABASE**);**

**}**