# 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、并不是转为红黑树节点，链表结构就不存在了

## 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;**

**}**