## 常用集合类源码解析

源码解析JDK1.8：ArrayList、LinkedList、HashSet、TreeSet、HashMap、TreeMap、ConcurrentHashMap

#### 一、ArrayList源码解析

List有索引、有序、可重复。

查询快，新增删除慢。

**1. 实现接口：**

List、RandomAccess、Cloneable、Serializable

实现RandomAccess便于集合元素的快速访问即查询。

实现Cloneable便于集合的拷贝，本质是数组的复制，属于浅拷贝。

实现Serializable便于集合的序列化

**深拷贝、浅拷贝的概念：**

Java中对象的复制通常通过赋值来实现，但是这种方式通过任何一方的引用操作都会改变对象。

Clone方法，可以实现对象的复制，得到一个新的对象。对于一个对象中的属性既包含基本类型又包含引用类型，clone方法完成的拷贝是浅拷贝，因为引用类型的对象仍然是共用的。

深拷贝就是让对象中的任何一个属性，包括基本数据类型和应用数据类型都是不相同的，这个拷贝得到的对象就是一个深拷贝的对象。这种方法需要各个引用类型都实现Cloneable接口，重写clone方法。

以List<Student>为例，直接调用ArrayList的clone方法属于浅拷贝，即ArrayList对象是新对象，但是里面的Student元素是相同的，新老集合指向相同的Student引用。

**2. 成员变量：**

初始容量10、最大容量Integer.MAX\_VALUE – 8(和jvm相关)，底层数据结构Object[]、集合大小size。

**3. 构造方法：**

可指定初始容量、或者直接传递一个集合实现类来构造ArrayList

**4. 常用方法：**

**5. 扩容：**

size+1 > elementData.length()进行扩容;

扩容1.5倍，即1.5(elementData.length());

进行数组的复制拷贝System.arraycopy()。

**6.线程安全问题：**

add方法中elementData[size++] = e;不是原子操作且没有加锁，多线程情况下回出现同一索引处的元素值被覆盖。

#### 二、LinkedList源码解析

查询慢，新增删除快。

**1.实现接口：**

List、Deque、Cloneable、Serializable

实现Deque表示这个一个支持元素在两端进行插入和删除操作的线性集合。

实现Cloneable，集合的浅拷贝。

**2.成员变量**

集合容量size,第一个节点 first、最后一个节点last，底层数据结构是双向链表。

first的特性(first == null && last == null) ||(first.prev == null && first.item != null)。

last的特性(first == null && last == null) ||(last.next == null && last.item != null)。

当集合初始化时,fist和last均为null;当集合中只有一个元素时，first和last是同一对象。first和last节点会携带集合元素的。

**3.构造方法**

支持传递集合Collection的构造方式

**4.常见问题**

无扩容问题，但是元素的查找效率不高，都是从头结点或者尾结点开始依次查找。

#### 三、HashSet源码解析

Set无索引、无序、不可重复。

HashSet<E>的底层是HashMap<E,Object>。

#### 四、TreeSet源码解析

TreeSet<E>的底层是NavigableMap<E,Object>。本质是TreeMap<E,Object>。

#### 五、HashMap源码解析

JDK1.7

**1.接口实现**

Map、Cloneable、Serializable

**2.成员变量**

默认初始化值16、最大容量2的30次方。容量必须为2的次方。加载因子loadFactor默认为0.75，集合容量size,底层是一个单链表数组Entry<K,V>[] table。扩容阈值threshold。Entry对象中的属性有key、value、hash、next。

**3.构造方法**

构造方法中可以传递容量大小capacity，加载因子loadFacotor，Map实现类。

**4.添加元素的过程、扩容**

1.将Map集合的容量设置为一个合理的值（这个值是2的次幂且和当前值最接近）。设置扩容阈值，0.75\*容量。

2.计算当前key的hash值。

3.获取key在数组中的位置：h & (length-1)，因为length-1都为1，故key在桶中的位置只由key的hash值决定。

4.如果添加元素的key值已存在（hashCode和equals方法相等），将对应的value替换成新值。

5.如果对应的桶中没有节点，或者遍历完没有相同的key值节点，那么就新增一个entry节点。

6.当前Map集合的size大于等于扩容阈值并且发生了hash冲突，则进行扩容。扩容至当前数组的两倍长度，重新分配元素在桶中的位置，更新扩容阈值threshold。(size >= threshold) && (null != table[bucketIndex])

因为上面这两个条件，所以存在下面这些情况

就是hashmap在存值的时候（默认大小为16，负载因子0.75，阈值12），可能达到最后存满16个值的时候，再存入第17个值才会发生扩容现象，因为前16个值，每个值在底层数组中分别占据一个位置，并没有发生hash碰撞。

当然也有可能存储更多值（超多16个值，最多可以存26个值）都还没有扩容。原理：前11个值全部hash碰撞，存到数组的同一个位置（这时元素个数小于阈值12，不会扩容），后面所有存入的15个值全部分散到数组剩下的15个位置（这时元素个数大于等于阈值，但是每次存入的元素并没有发生hash碰撞，所以不会扩容），前面11+15=26，所以在存入第27个值的时候才同时满足上面两个条件，这时候才会发生扩容现象。

7.添加新的entry节点，不管是添加新节点还是扩容时元素重新分配，采用的都是单链表的头插法，即新元素在最前面。

**5.常见问题**

1.扩容过程中，多线程模式在transfer方法中可能会有死循环问题出现。出现aba即a指向b,b指向a，查询的时候就会进入死循环。

2. HashMap的遍历输出的key值顺序：通过对key值的hash算法分配在桶中位置，按在桶中的位置输出（从索引小到大）。

3. LinkedHashMap遍历顺序：双向链表，每次插入都是放在最后的节点，遍历从head到tail，如果不指定顺序的话，一般是和插入的顺序一致。

4.TreeMap遍历顺序：按自然顺序（ascii码）排序输出。

JDK1.8

**1.接口实现**

Map、Cloneable、Serializable

**2.成员变量**

默认初始化值16、桶的最大容量2的30次方。容量必须为2的次方。加载因子loadFactor默认为0.75，集合容量size,底层是一个单链表+数组+红黑树Node<K,V>[] table。扩容阈值threshold。Entry对象中的属性有key、value、hash、next。树化阈值（链表转红黑树）TREEIFY\_THRESHOLD、非树化阈值（红黑树转链表）UNTREEIFY\_THRESHOLD、最小树化容量（桶可能被转化为树形结构的最小容量的临界值）MIN\_TREEIFY\_CAPACITY。

数据结构为：数组+链表+红黑树

**3.构造方法**

构造方法中可以传递容量大小capacity，加载因子loadFacotor，Map实现类。

**4.元素添加**

**5.元素查找**

1.检查查找的key与其所在桶中位置的第一个元素是否相等。相等则返回。

2.若节点是红黑树，则进入红黑树的查找算法。

3.若节点是链表则进行链表的查找。

**6.元素移除**

1.检查查找的key与其所在桶中位置的第一个元素是否相等。相等则返回。

**2.按照红黑树或者链表结构找到需要删除的节点。**

**7.扩容**

1.新增节点后，size值大于扩容阈值threshold,扩容。

2.数组大小和扩容阈值变为原来的两倍。

3. (e.hash & oldCap) == 0则节点在桶中的索引位置不变，若不为0则索引位置为（index+oldCap）。等价于e.hash & (newCap -1)。newCap = 2\*oldCap

4.对于树形结构扩容，扩容后节点数小于等于6则红黑树转链表，否则继续树化（链表转红黑树）。

**8.链表转红黑树过程**

**Node->TreeNode**

1.增加节点后若链表长度达到阈值8则开始树化。

2.将桶中指定索引位置的单链表转换为双链表，链表中元素位置不变，增加prev引用。

3.节点插入顺序是节点再链表中的顺序，根据节点的hash值大小转化为红黑树。将红黑树根节点置为桶中的第一个节点。

4.校验红黑树是否符合规范。

**9.红黑树转链表过程**

**TreeNode->Node**

1.树节点长度小于等于6时进行非树化。

2.树节点转为链表节点，单链表，采用尾插法。

**10.常见问题**

关于HashMap节点的属性说明：

key,value,hash,next,prev,parent,left,right,red

TreeNode继承了Node，prev,next用于链表；parent,left,right,red用于红黑树；方便与链表和红黑树的相互转换。

#### 六、TreeMap源码解析

**1.接口实现**

NavigableMap<K,V>, Cloneable, Serializable

**2.成员变量**

排序比较器comparator(默认按key的ascii码升序)，Entry<K,V> root根节点，size键值对数量，底层是红黑树。

Entry字段包含key,value,left,right,parent,color

**3.构造方法**

无参构造、传比较器方式、传Map实现类方式：

public TreeMap(Comparator<? super K> comparator) {}

public TreeMap(Map<? extends K, ? extends V> m) {}

**4.添加元素过程**

添加键值对，如果key值存在则替换旧的value值。

1.判断根节点是否存在，如果不存在将新增节点设置为根节点。结束

2.若根节点存在，判断是否传入比较器，如果有比较器根据比较器规则比较key值大小，如果未传入，则按照自然顺序进行排序（ascii码升序）。

3.找到插入节点的父节点。从根节点开始，比较插入节点的key值和当前节点的key值，如果大于将当前节点的右子树节点设置为当前节点，如果小于将当前节点的左子树节点设置为当前节点继续查找，如果等于则将该key值对应的value更新。循环判断直到当前节点为空，返回此时的父节点。

4.新增一个节点，该节点为红色。他的父节点是上述返回的父节点。

5.红黑树自平衡。场景具有对称性。3层判断（插入节点的父节点是否为红色、父节点是否是祖父节点的左子树、叔叔节点是否是红色、插入节点是父节点的左右子树）。

**5.查找元素过程**

根据key值查找指定的value

1.传入比较器则按照比较器规则比较，否则按照自然顺序排序。

2.查找规则：将根节点设置为当前节点，若当前节点为空则直接返回null,若插入节点key值等于当前节点key值则返回当前节点，若小于则将当前节点的左子树设置为当前节点，若大于则将当前节点的右子树设置为当前节点。重复步骤2。

**6.移除元素过程**

根据key值移除键值对

1.根据key值查找到相应的节点，参考查找元素过程。

2.查找替代节点，若删除节点无子节点，则替代节点为本身；若删除节点有一个子节点，则替代节点为该子节点；若删除节点有两个子节点，则替代节点为右子树节点的最左节点。删除节点。

3.红黑树完成自平衡。具有对称性。四层判断（替代节点是否为黑色，替代节点是否为其父节点的左子树，替代节点的兄弟节点是否为黑色，替代节点的兄弟节点的子树颜色）。确定好具体的场景进行变色和旋转，源码中的步骤同红黑树算法步骤。

#### 七、ConcurrentHashMap 源码解析

<https://www.cnblogs.com/zhaojj/p/8942647.html>

#### 八、红黑树

二叉树：每个节点最多有两个子树的树结构。

二叉查找树：将所有节点映射到水平轴，键值不重复，键值从左到右依次递增的二叉树。

红黑树：能够完成自平衡的二叉查找树。

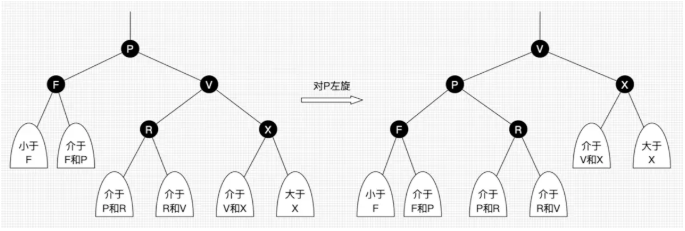
<https://www.jianshu.com/p/e136ec79235c>

**特征：**

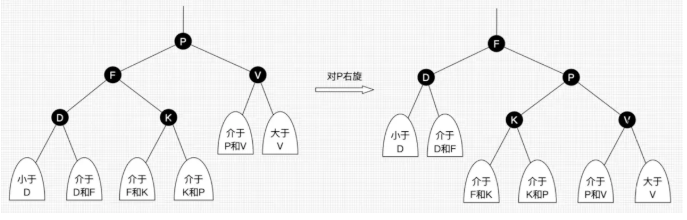
能实现自平衡的二叉查找树。（key值从左到右递增）

**自平衡（左旋、右旋、变色）**

左旋：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。



右旋：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。



红黑树总是通过旋转和变色达到自平衡。**通过旋转，使两边的节点数差不多。**

**性质：**

性质1：每个节点要么是黑色，要么是红色。

性质2：根节点是黑色。

性质3：每个叶子节点（NIL）是黑色。

性质4：每个红色结点的两个子结点一定都是黑色。（黑节点的两个子节点可以是两黑、两红、一黑一红）

性质5：任意一结点到每个叶子结点的路径都包含数量相同的黑结点。

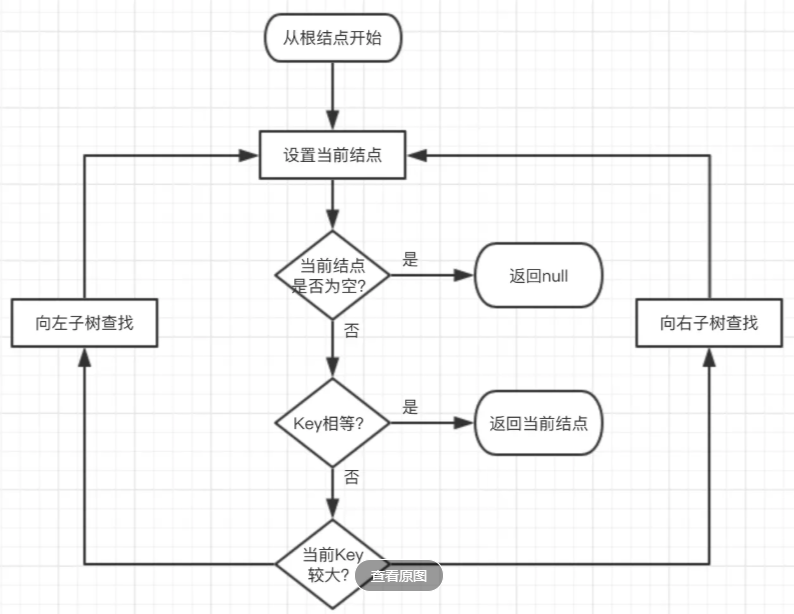
**存储：**

数组、链表

**遍历：**

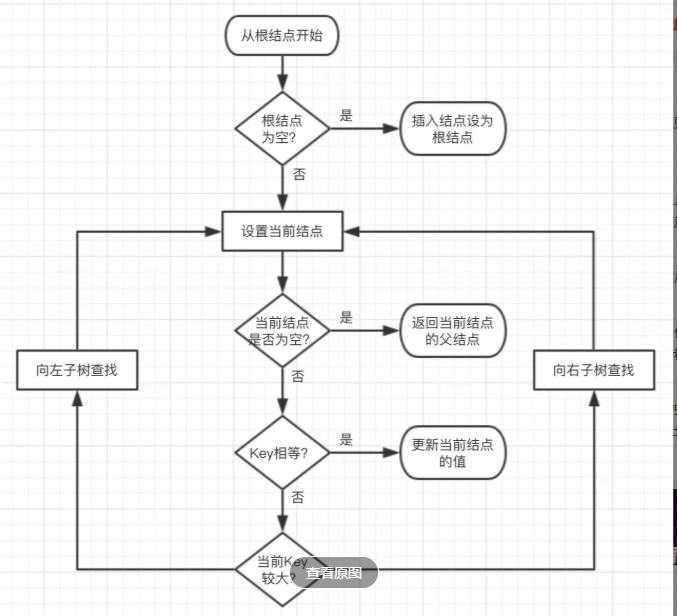
查找最坏时间复杂度为O(2lgN)

Key值从左到右递增，采用的递归算法。二分法思想



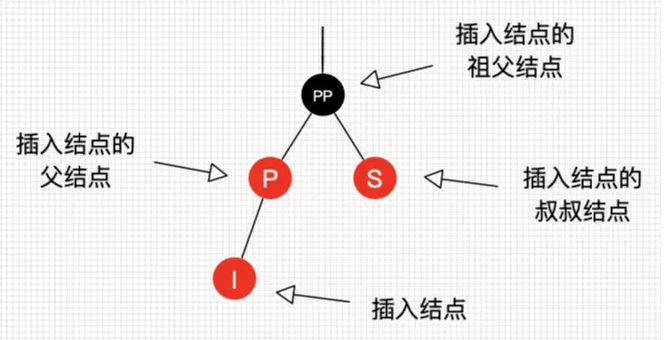
**插入：**

查找插入的位置,返回的是子节点为空的父节点。插入的节点为红色！（大部分情况）。



8中基本场景

更改：情景4.3.2：插入节点是其父节点的左子节点

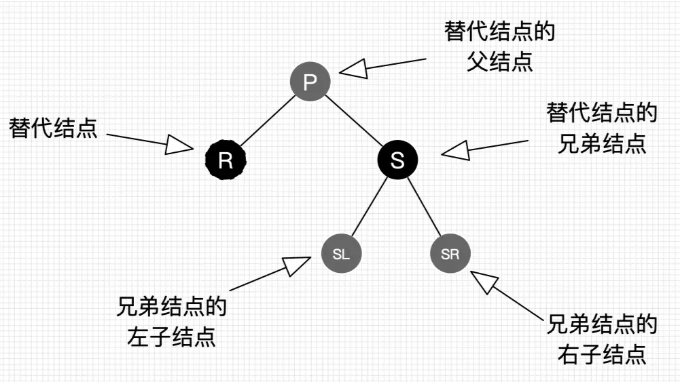




**删除：**

**删除节点-》自平衡（替代节点参与自平衡过程）-》替代节点替代**

**重新进入场景处理：进入2.1.2；进入2.2.2**



9中基本场景



删除结点被替代后，在不考虑结点的键值的情况下，对于树来说，可以认为删除的是替代结点！删除操作删除的结点可以看作删除替代结点，而替代结点最后总是在树末。

场景：替代节点的查找

情景1：若删除结点无子结点，直接删除

情景2：若删除结点只有一个子结点，用子结点替换删除结点

情景3：若删除结点有两个子结点，用后继结点（大于删除结点的最小结点即子树的最左节点）替换删除结点

总结：自己能够搞定的自己消化；不能搞定的找兄弟帮忙；兄弟帮不了的，通过父母，找远方亲戚。