链滴

JDK集合类学习

作者：guobing

原文链接：https://ld246.com/article/1470842521588

来源网站：链滴

1. 算法
在此输入正文
`反转链表` `冒泡排序` `二分查找（递归|非递归）` `另一道是在N个数中求前M大个数`

> * 一组排序数中，给定一个数，返回最接近且不大于这个数的位置
> * 把一个数组中奇数放前面，偶数放后面
> * 另一个是3亿条IP中，怎么找到次数出现最多的5000条IP
> * 一个大文件几个GB，怎么实现复制

3.LinkedHashMap
LinkedHashMap 维护着一个运行于所有条目的`双重链接列表`。此链接列表定义了迭代顺序，该迭顺序可以是插入顺序或者是访问顺序。
根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用 get 方法)的链表。默认是插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部不断访问可以形成按访问顺序排序的链表。
不过，我的建议是，大家首先首先需要记住的是：LinkedHashMap 能够做到按照`插入顺序或者访问序`进行迭代，这样在我们以后的开发中遇到相似的问题，才能想到用 LinkedHashMap 来解决，否就算对其内部结构非常了解，不去使用也是没有什么用的。

主要记忆点：
> * JDK描述
differs is maintains a doubly-linked list TODO
> * 成员变量
extends HashMap
LinkedHashMap 采用的 hash 算法和 HashMap 相同，但是它重新定义了数组中保存的元素 Entry 该 Entry 除了保存当前对象的引用外，还保存了其上一个元素 before 和下一个元素 after 的引用，而在哈希表的基础上又构成了双向链接列表。看源代码：
```
static class Entry<K,V> extends HashMap.Node<K,V> {
        Entry<K,V> before, after;
        Entry(int hash, K key, V value, Node<K,V> next) {
            super(hash, key, value, next);
        }
}

/**
 * The iteration ordering method for this linked hash map:          * <tt>true</tt>
 * for access-order, <tt>false</tt> for insertion-order.
 *
 * @serial
 */
    final boolean accessOrder; # true访问顺序，false插入顺序
/**
 * The head (eldest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> head; # 双向链表头结点

/**
 * The tail (youngest) of the doubly linked list.
 */
transient LinkedHashMap.Entry<K,V> tail; # 双向链表尾节点
```
> * 初始化
通过源代码可以看出，在 LinkedHashMap 的构造方法中，实际调用了父类 HashMap 的相关构造

法来构造一个底层存放的 table 数组，但额外可以增加 accessOrder 这个参数，如果不设置，默认为 alse，代表按照插入顺序进行迭代；当然可以显式设置为 true，代表以访问顺序进行迭代。如：
```
 /**
 * Constructs an empty insertion-ordered <tt>LinkedHashMap</tt> instance
 * with the specified initial capacity and load factor.
 *
 * @param  initialCapacity the initial capacity
 * @param  loadFactor      the load factor
 * @throws IllegalArgumentException if the initial capacity is negative
 *         or the load factor is nonpositive
 */
    public LinkedHashMap(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor);
        accessOrder = false;
    }
```

我们已经知道 LinkedHashMap 的 Entry 元素继承 HashMap 的 Entry，提供了双向链表的功能。在述 HashMap 的构造器中，最后会调用 init() 方法，进行相关的初始化，这个方法在 HashMap 的实中并无意义，只是提供给子类实现相关的初始化调用。

但在 LinkedHashMap 重写了 init() 方法，在调用父类的构造方法完成构造后，进一步实现了对其元 Entry 的初始化操作。
```
/**
* Called by superclass constructors and pseudoconstructors (clone,
* readObject) before any entries are inserted into the map.  Initializes
* the chain.
*/
@Override
void init() {
  header = new Entry<>(-1, null, null, null);
  header.before = header.after = header;
}
```
> * 存储
put:调用的还是父类hashMap的put方法。
get:先通过key的哈希值获取元素`e = getNode(hash(key) `如果accessOrder为true的话，执行`afte NodeAccess()`方法，然后返回e.value.
`afterNodeAccess()`的作用就是`move node to last`
afterNodeAccess代码如下：
```
void afterNodeAccess(Node<K,V> e) { // move node to last
        LinkedHashMap.Entry<K,V> last;
        if (accessOrder && (last = tail) != e) {
            LinkedHashMap.Entry<K,V> p =
                (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
            p.after = null;
            if (b == null)
                head = a;
            else
                b.after = a;
            if (a != null)
```

```
                a.before = b;
            else
                last = b;
            if (last == null)
                head = p;
            else {
                p.before = last;
                last.after = p;
            }
            tail = p;
            ++modCount;
        }
    }
```

那么访问元素的时候如何做到按插入顺序还是访问顺序输出的呢？答案是遍历的时候用了map.entry et().iterator()，而entrySet()又调用了LinkedEntryIterator，
LinkedEntryIterator代码如下：

```
final class LinkedEntryIterator extends LinkedHashIterator
        implements Iterator<Map.Entry<K,V>> {
        public final Map.Entry<K,V> next() { return nextNode(); }
}
```

是返回了一个nextNode(),个nextNode()代码如下：

```
final LinkedHashMap.Entry<K,V> nextNode() {
            LinkedHashMap.Entry<K,V> e = next;
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            if (e == null)
                throw new NoSuchElementException();
            current = e;
            next = e.after;
            return e;
}
```

还有两个重要的方法
`afterNodeRemoval`、`afterNodeInsertion`

```
void afterNodeRemoval(Node<K,V> e) { // unlink
        LinkedHashMap.Entry<K,V> p =
            (LinkedHashMap.Entry<K,V>)e, b = p.before, a = p.after;
        p.before = p.after = null;
        if (b == null)
            head = a;
        else
            b.after = a;
        if (a == null)
            tail = b;
        else
            a.before = b;
}
```

```
 void afterNodeInsertion(boolean evict) { // possibly remove eldest
        LinkedHashMap.Entry<K,V> first;
        if (evict && (first = head) != null && removeEldestEntry(first)) {
            K key = first.key;
            removeNode(hash(key), key, null, false, true);
        }
}
```

4. LinkedHashSet
`public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, java.io.erializable `
只有六个方法
```
public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
}

public LinkedHashSet(int initialCapacity) {
    super(initialCapacity, .75f, true);
}

public LinkedHashSet() {
        super(16, .75f, true);
}

public LinkedHashSet(Collection<? extends E> c) {
        super(Math.max(2*c.size(), 11), .75f, true);
        addAll(c);
}

public Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, Spliterator.DISTINCT | Spliterator.ORDERED);
}
```
> * LinkedHashSet 是 Set 的一个具体实现，其维护着一个运行于所有条目的双重链接列表。此链接表定义了迭代顺序，该迭代顺序可为插入顺序或是访问顺序。
> * LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的（具体的别大家可以自己去思考一下）。
> * 如果我们需要迭代的顺序为插入顺序或者访问顺序，那么 LinkedHashSet 是需要你首先考虑的。

4. hashSet
hashSet没有get方法，因为get方法没有意义。
hashSet的处理过程：调用hashMap的put方法，要插入的值做key，value是object对象。在hashMa中，当key不重复时，插入。重复时（hashCode()返回值相等，通过 equals 比较也返回 true）valu覆盖旧值，key值不会有任何变化。返回null。hashSet利用这个实现去重功能。

5. CopyOnWriteArrayList
CopyOnWriteArrayList相当于线程安全的ArrayList，通过增加写时复制语义来实现线程安全性。底也是通过一个可变数组来实现的。但是和ArrayList不同的时，它具有以下特性：
> * 它最适合于具有以下特征的应用程序：List 大小通常保持很小，只读操作远多于可变操作，需要遍历期间防止线程间的冲突
> * 支持高效率并发且是线程安全的

因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大
> * 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等操作
> * 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖不变的数组快照

6. CopyOnWriteArraySet
它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。有意思的是，CopyOnWriteArraSet和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过“散列表(HashMap)”现的，而CopyOnWriteArraySet则是通过“动态数组(CopyOnWriteArrayList)”实现的，并不是散表。
和CopyOnWriteArrayList类似，CopyOnWriteArraySet具有以下特性：
1. 它最适合于具有以下特征的应用程序：Set 大小通常保持很小，只读操作远多于可变操作，需要在历期间防止线程间的冲突。
2. 它是线程安全的。
3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。
4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等 操作。
5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于变的数组快照。