集合总结

一、基本概念

1. Collection:一组"对立"的元素，通常这些元素都服从某种规则,每个位置只能保存一个元素(对象)
    1. List必须保持元素特定的顺序
    2. Set不能有重复元素
    3. Queue保持一个队列(先进先出)的顺序
2. Map:一组成对的"键值对"对

二、Collection

> Iterable迭代器接口，这是Collection类的父接口。实现这个Iterable接口的对象允许使用foreach进行遍历，也就是说，所有的Collection集合对象都具有"foreach可遍历性"。这个Iterable接口只有一个方法: iterator()。它返回一个代表当前集合对象的泛型<T>迭代器，用于之后的遍历操作。

2.1 set(接口)

> Set判断两个对象相同不是使用"=="运算符，而是根据equals方法。也就是说，我们在加入一个新元素的时候，如果这个新元素对象和Set中已有对象进行注意equals比较都返回false，否则Set就会接受这个新元素对象，否则拒绝。

因为Set的这个制约，在使用Set集合的时候，应该注意两点：

- 1）为Set集合里的元素的实现类实现一个有效的equals(Object)方法。
- 2）对Set的构造函数，传入的Collection参数不能包含重复的元素。

2.1.1 HashSet

HashSet:通过hashmap实现，将元素作为key，value作为统一的Object(static final)，且只用这一个。

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable...
{
    private transient HashMap<E,Object> map;  //用hashmap实现
    private static final Object PRESENT = new Object();//map中的value
    public HashSet(int initialCapacity) {
        map = new HashMap<>(initialCapacity);
    }
    //该构造方法仅供下面的LinkedHashSet使用(通过链表维护顺序)
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
        map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
    public boolean contains(Object o) {
        return map.containsKey(o);
    }
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
    public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
    ...
}
```

2.1.2 LinkedHashSet

继承自HashSet，通过调用父类构造器，使用LinkedHashMap对象，而不是HashMap对象。在HashSet的基础上，同时使用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的。

当遍历LinkedHashSet集合里的元素时，LinkedHashSet将会按元素的添加顺序来访问集合里的元素。

LinkedHashSet需要维护元素的插入顺序，因此性能略低于HashSet的性能，但在迭代访问Set里的全部元素时(遍历)将有很好的性能(链表很适合进行遍历)

```
public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable... {
    //调用父类的构造器，使用LinkedHashSet，(通过链表维护顺序)
    public LinkedHashSet(int initialCapacity, float loadFactor) {
        super(initialCapacity, loadFactor, true);
    }
```

2.1.3 SortedSet

通过treeMap实现

2.1.4 EnumSet

2.2 List

List集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List集合允许加入重复元素，因为它可以通过索引来访问指定位置的集合元素。List集合默认按元素的添加顺序设置元素的索引。

List接口：代表了有序元素的一系列操作。

Queue接口：队列操作(add,offer,poll,remove,peek等操作)

Deque接口，继承自Queue(队列操作)：双端队列(addFirst,offerLast,peekLast,**push**,**pop**等操作)

2.2.1 LinkedList

实现了List和Deque接口，可以当作list、队列、双端队列、栈来使用。双向链表实现。元素可以为null。

```
public class LinkedList<E> extends AbstractSequentialList<E>
    implements List<E>, Deque<E>...{  }
```

2.2.2 ArrayList

数组实现，可以存null元素(==判断)，默认初始容量10，扩容1.5倍。拷贝通过Arrays.copyOf(Data, Cap)实现。

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final int DEFAULT_CAPACITY = 10; //默认容量
    private static final Object[] EMPTY_ELEMENTDATA = {};
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
    transient Object[] elementData; // non-private to simplify nested class access
    ..
}
```

2.2.3 Vector

Vector和ArrayList在用法上几乎相同,但是方法名不一样，而且线程安全，通过synchronized实现。默认初始容量10，扩容时可以指定增量，不指定则翻倍。

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    protected Object[] elementData;
    protected int elementCount;
    protected int capacityIncrement;
    public synchronized void addElement(E obj) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);
        elementData[elementCount++] = obj;
    }
    ...
```

```
}
```

2.2.4 Stack

模拟栈，继承Vector，通过synchronized实现线程安全。

```
public
class Stack<E> extends Vector<E> {
    public Stack() {
    }
    public synchronized E peek() {
        int len = size();
        if (len == 0)
            throw new EmptyStackException();
        return elementAt(len - 1);
    }
    ...
}
```

2.3 Queue

2.3.1 PriorityQueue

AbstractQueue类包装了Queue接口中的方法。

默认初始容量10，扩容1.5倍，如果原来容量<64，则扩容为2*cap+2。通过最小堆实现。元素可实现Comparable接口来自定义比较大小。

```
public class PriorityQueue<E> extends AbstractQueue<E>
    implements java.io.Serializable {

    private static final int DEFAULT_INITIAL_CAPACITY = 11;//默认初始容量
    transient Object[] queue; // non-private to simplify nested class access
```

2.3.2 ArrayQueue

通过循环数组来实现队列。

2.3.3 LinkedList

三、Map

Map接口：定义了一些map的基本操作.内部接口map.Entry<K,V>定义了节点。

AbstractMap 实现了Map接口：实现了一些基本操作

3.1 HashMap

支持null，默认16，加载因子0.75，转树8（超过64），转链表6.扩容翻倍。

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
    static final int MAXIMUM_CAPACITY = 1 << 30;
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    static final int TREEIFY_THRESHOLD = 8;
    static final int UNTREEIFY_THRESHOLD = 6;
    static final int MIN_TREEIFY_CAPACITY = 64;
```

3.2 LinkedHashMap

继承HashMap，节点也继承Node，并添加before、after属性成为双向链表。

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
{
    // 节点多了before,after属性
    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);
        }
    }
    transient LinkedHashMap.Entry<K,V> head;//双向链表的头节点
    transient LinkedHashMap.Entry<K,V> tail;//双向链表的尾节点
    final boolean accessOrder;//默认false， 插入顺序
```

3.3 HashTable

方法通过synchronized来实现线程安全。继承字典类Dictionary。

1. HashMap允许key和value为null，Hashtable不允许。
2. HashMap的默认初始容量为16，Hashtable为11。
3. HashMap的扩容为原来的2倍，Hashtable的扩容为原来的2倍加1。
4. HashMap是非线程安全的，Hashtable是线程安全的。
5. HashMap的hash值重新计算过，Hashtable直接使用hashCode。
6. HashMap去掉了Hashtable中的contains方法。
7. HashMap继承自AbstractMap类，Hashtable继承自Dictionary类。

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {
```

3.4 Properties

继承HashTable,可以用来存放 属性-属性值

```
public class Properties extends Hashtable<Object,Object> {
```

3.5 SortedMap

sortedMap接口定义了一些排序后的基本方法：

```
public interface SortedMap<K,V> extends Map<K,V> {
    Comparator<? super K> comparator();
    SortedMap<K,V> subMap(K fromKey, K toKey);
    SortedMap<K,V> headMap(K toKey);
    SortedMap<K,V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
}
```

3.5 TreeMap

通过红黑书实现,继承自AbstractMap

```
public class TreeMap<K,V> extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

3.6 WeakHashMap

WeakHashMap与HashMap的用法基本相似。区别在于，HashMap的key保留了对实际对象的"强引用"，这意味着只要该HashMap对象不被销毁，该HashMap所引用的对象就不会被垃圾回收。

但WeakHashMap的key只保留了对实际对象的弱引用，这意味着如果WeakHashMap对象的key所引用的对象没有被其他强引用变量所引用，则这些key所引用的对象可能被垃圾回收，当垃圾回收了该key所对应的实际对象之后，WeakHashMap也可能自动删除这些key所对应的key-value对.

如果key为null，内部用一个名为NULL_KEY的Object对象代替。

```
public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>
```

这个“弱键”的原理呢？大致上就是，**通过WeakReference和ReferenceQueue实现的**。 WeakHashMap的key是“弱键”，即是WeakReference类型的；ReferenceQueue是一个队列，它会保存被GC回收的“弱键”。实现步骤是：

1. 新建WeakHashMap，将**“键值对”**添加到WeakHashMap中。实际上，WeakHashMap是通过数组table保存Entry(键值对)；每一个Entry实际上是一个单向链表，即Entry是键值对链表。
2. 当**某“弱键”不再被其它对象引用**，并**被GC回收**时。在GC回收该“弱键”时，**这个“弱键”也同时会被添加到ReferenceQueue(queue)队列**中。
3. 当下一次我们需要操作WeakHashMap时，会先同步table和queue。table中保存了全部的键值对，而queue中保存被GC回收的键值对；同步它们，就是**删除table中被GC回收的键值对。**

这就是“弱键”如何被自动从WeakHashMap中删除的步骤了。

3.7 IdentityHashMap

IdentityHashMap的实现机制与HashMap(hash值相等 && (key1 == kwy2 || key1.equal(key2)))基本相似，在IdentityHashMap中，当且仅当两个key严格相等(key1 == key2)时，IdentityHashMap才认为两个key相等.

3.8 EnumMap

EnumMap是一个与枚举类一起使用的Map实现，EnumMap中的所有key都必须是单个枚举类的枚举值。创建EnumMap时必须显式或隐式指定它对应的枚举类。EnumMap根据key的自然顺序(即枚举值在枚举类中的定义顺序)