Java1.8-Collections源码解析

骑着乌龟去看海 关注
0.238 2018.02.05 16:18:31 字数 3,595 阅读 1,634

概述

在Java集合框架中，还有两个经常用到的工具类：Collections和Arrays。顾名思义，Collections是用来操作集合的工具类，而Arrays是用来操作数组的工具类。这两个工具类提供了许多用于各自操作的静态方法。
本篇文章，我们先来学习一下Collections工具类。

构造方法和属性

```
1 private Collections() {
2 }
```

构造方法私有，说明不对外提供，一般我们用到的时候都是用Collections提供的静态方法即可。

```
1 private static final int BINARYSEARCH_THRESHOLD   = 5000;
2 private static final int REVERSE_THRESHOLD       =   18;
3 private static final int SHUFFLE_THRESHOLD       =    5;
4 private static final int FILL_THRESHOLD          =   25;
5 private static final int ROTATE_THRESHOLD        =  100;
6 private static final int COPY_THRESHOLD          =   10;
7 private static final int REPLACEALL_THRESHOLD    =   11;
8 private static final int INDEXOFSUBLIST_THRESHOLD =   35;
```

上面这些属性是Collections的调优参数。通常Collections的许多算法都有两个实现，一个适用于随机访问，另一个适合顺序访问。通常随机访问在列表数据量小的适合可以获得很好的性能，这里的每个值代表了该操作使用随机访问的数据的阈值。而这些值的确定是根据以往的经验确定的，对LinkedList是很有效的。这里每个调优参数名的第一个词是它所应用的算法。

sort方法

```
1 public static <T extends Comparable<? super T>> void sort(List<T> list) {
2     list.sort(null);
3 }
4 public static <T> void sort(List<T> list, Comparator<? super T> c) {
5     list.sort(c);
6 }
```

Collections有两个sort方法，第一个要求List中的对象必须要实现了Comparable接口；而第二个方法则不要求实现Comparable接口，但可以自定义比较器。但两者底层实现都是通过List接口的默认方法sort。

```
1  /**
2   * 转成数组然后调用Arrays的sort方法进行排序
3   */
4  default void sort(Comparator<? super E> c) {
5      Object[] a = this.toArray();
6      Arrays.sort(a, (Comparator) c);
7      ListIterator<E> i = this.listIterator();
8      for (Object e : a) {
9          i.next();
10         i.set((E) e);
11     }
12 }
```

我们可以看到，List的sort方法是使用了JDK8接口的新特性-默认方法来实现的。

binarySearch方法

使用二分查找算法查找对象。调用这个方法必须有两个前提：

> 1. 这个集合必须已经排好序；
> 2. 这个集合必须可以比较；

如果没有排序，那查询出来的结果就没有什么意义。同样，如果对象类型不同，无法进行比较，将会抛出异常ClassCastException。并且如果列表中包含要查询对象的多个重复对象，那么不保证每次找到的元素的位置相同。

```
1 public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key) {
2     if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)
3         return Collections.indexedBinarySearch(list, key);
4     else
5         return Collections.iteratorBinarySearch(list, key);
6 }
```


牛仔裤系列

骑着乌龟去看海 关注
总资产19

Lombok首字母小写，第二个字母大写的问题
阅读 217
MySQL中一些非常规的函数使用总结
阅读 230

推荐阅读

Java集合
阅读 209
List 去除重复数据的五种方式
阅读 273
Python 字典和集合 - 认识字典
阅读 124
Map接口
阅读 134
HashMap实现原理（和Hashtable、HashSet的区别）
阅读 546

```
private static <T>
int indexedBinarySearch(List<? extends Comparable<? super T>> list, T key) {
    int low = 0;
    int high = list.size()-1;

    while (low <= high) {
        // 使用位运算，计算中间索引值
        int mid = (low + high) >>> 1;
        // 计算中间的元素值
        Comparable<? super T> midVal = list.get(mid);
        // 进行比较
        int cmp = midVal.compareTo(key);

        if (cmp < 0)
            // 比传入的key小，在list的高位部分查找
            low = mid + 1;
        else if (cmp > 0)
            // 比传入的key大，在list的低位部分查询
            high = mid - 1;
        else
            // 相等，直接返回
            return mid; // key found
    }
    // 没有找到，返回负数
    return -(low + 1);  // key not found
}
```

而通过使用迭代器遍历的方式，则需要借助Collections.get方法来实现，这种实现会涉及到循环遍历，所以效率会稍微低些。

```
private static <T>
int iteratorBinarySearch(List<? extends Comparable<? super T>> list, T key)
{
    int low = 0;
    int high = list.size()-1;
    // 通过ListIterator迭代器来进行查找
    ListIterator<? extends Comparable<? super T>> i = list.listIterator();

    while (low <= high) {
        int mid = (low + high) >>> 1;
        // 通过Collections.get方法获取中间索引处的元素值
        Comparable<? super T> midVal = get(i, mid);
        int cmp = midVal.compareTo(key);

        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found
}
```

get方法源码：

```
private static <T> T get(ListIterator<? extends T> i, int index) {
    T obj = null;
    // 获取下一个索引值
    int pos = i.nextIndex();
    // 循环判断获取的索引是否小于中间索引index
    // 如果小于，从前往后遍历，否则，从后往前遍历，最后返回元素值
    if (pos <= index) {
        do {
            obj = i.next();
        } while (pos++ < index);
    } else {
        do {
            obj = i.previous();
        } while (--pos > index);
    }
    return obj;
}
```

同样，binarySearch也是有两个重载的方法，实现是类似的，就不多说了。

reverse**方法**

列表反转方法，如果列表支持随机访问或者列表大小小于要反转的阈值18，则直接采用交换操作；否则采用双迭代操作，一个从头遍历，一个从尾遍历，然后交换。

```
public static void reverse(List<?> list) {
    int size = list.size();
    if (size < REVERSE_THRESHOLD || list instanceof RandomAccess) {
        for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)
            swap(list, i, j);
    } else {
        // 迭代器遍历从头开始（forward）
        ListIterator fwd = list.listIterator();
        // 迭代器遍历从尾部开始（reverse）
        ListIterator rev = list.listIterator(size);
        for (int i=0, mid=list.size()>>1; i<mid; i++) {
            // 从头开始
            Object tmp = fwd.next();
            // 设置fwd下一个元素为rev前一个元素，交换
            fwd.set(rev.previous());
            rev.set(tmp);
        }
    }
}
```

swap**方法**

```
public static void swap(List<?> list, int i, int j) {
    final List l = list;
    l.set(i, l.set(j, l.get(i)));
}
```

`l.set(j, l.get(i))` 这里在设置j处为新的值的同时，会返回索引j处原来的值，然后再次set，很巧妙的实现了交换操作。

shuffle**方法**

shuffle翻译过来是重新洗牌的意思，该方法是将list原有数据打乱生成一个新的乱序列表。通俗点来说，旧相当于重新洗牌，打乱原来的顺序。还有一点，shuffle方法再生成乱序列表的时候，所有元素发生交换的可能性是近似相等的。

```
public static void shuffle(List<?> list, Random rnd) {
    int size = list.size();
    if (size < SHUFFLE_THRESHOLD || list instanceof RandomAccess) {
        for (int i=size; i>1; i--)
            swap(list, i-1, rnd.nextInt(i));
    } else {
        // 转成数组进行处理
        Object arr[] = list.toArray();
        // 打乱顺序
        for (int i=size; i>1; i--)
            swap(arr, i-1, rnd.nextInt(i));
        // 将数组放回列表中
        ListIterator it = list.listIterator();
        for (int i=0; i<arr.length; i++) {
            it.next();
            it.set(arr[i]);
        }
    }
}
```

1. 从上面可以看出，如果列表支持随机访问或者列表大小小于重新打乱顺序的阈值5，那么就进行交换。交换的规则是从当前列表的最后一个元素开始，依次和前面随机一个元素进行交换，这样交换整个列表，就可以认为这个列表是无序的。
2. 如果列表不满足上述条件，就将列表先转为数组，然后按照相同的方式进行交换处理，最后再将数组放回列表中即可。
3. shuffle还有另一个重载方法，可以传入指定种子数的Random。也就是说一旦指定了种子数，那么每次将会产生相同的随机数，也就相当于这种随机生成的元素就是一种伪随机。我们可以根据需要调用相应的方法。

fill**方法**

将List的原有数据全部填充为一个固定的元素。同样也分两种情况，如果列表支持随机访问或者大小小于要填充的阈值，就直接遍历List进行set操作即可；否则，使用iterator迭代器模式进行设值。

```
public static <T> void fill(List<? super T> list, T obj) {
    int size = list.size();
    // 如果大小小于阈值或者支持随机访问
    if (size < FILL_THRESHOLD || list instanceof RandomAccess) {
        for (int i=0; i<size; i++)
            list.set(i, obj);
    } else {
        // 否则使用迭代器模式进行设值
        ListIterator<? super T> itr = list.listIterator();
        for (int i=0; i<size; i++) {
            itr.next();
            itr.set(obj);
        }
    }
}
```

copy**方法**

将原集合中元素拷贝到另一个集合中。

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    int srcSize = src.size();
    // 如果原列表大于目标列表大小，抛异常
    if (srcSize > dest.size())
        throw new IndexOutOfBoundsException("Source does not fit in dest");
    // 同样分两种情况处理
    if (srcSize < COPY_THRESHOLD ||
        (src instanceof RandomAccess && dest instanceof RandomAccess)) {
        for (int i=0; i<srcSize; i++)
            dest.set(i, src.get(i));
    } else {
        ListIterator<? super T> di=dest.listIterator();
        ListIterator<? extends T> si=src.listIterator();
        for (int i=0; i<srcSize; i++) {
            di.next();
            di.set(si.next());
        }
    }
}
```

> 方式也是有两种处理方式。

min**方法**，max**方法**

min方法返回指定集合的最小元素，根据自然顺序进行比较。需要注意的一点就是集合中的元素必须是可比较的（实现Comparable）。该方法通过使用迭代器迭代整个集合。


```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll) {
    // 使用迭代器来操作
    Iterator<? extends T> i = coll.iterator();
    // 通过一个变量来保存最小值
   T candidate = i.next();
    while (i.hasNext()) {
        T next = i.next();
        // 通过compareTo方法来进行比较
        if (next.compareTo(candidate) < 0)
            candidate = next;
    }
    return candidate;
}
```

该方法还有一个重载方法，可以指定比较器，实现相似，就不多说了：

```
public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
```

同理，max方法是获取集合的最大元素，和min方法类似，也有两个重载方法，不多说了。

rotate**方法**

> public static void rotate(List<?> list, int distance)

对集合进行旋转操作，实际上就是集合里的元素右移操作，参数distance就是右移的距离。我们先举个简单的例子看下就明白了：

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);
    list.add(6);
    list.add(7);
    list.add(8);
    list.add(9);
    System.out.println("src list : " + list.toString());
    Collections.rotate(list, 2);
    System.out.println("rotate list : " + list.toString());
}
```

output：

```
src list : [1, 2, 3, 4, 5, 6, 7, 8, 9]
rotate list : [8, 9, 1, 2, 3, 4, 5, 6, 7]
```

很明显，右移两位就是将列表整体右移两位，最后两位移动到最前面而已。我们来看下源码：

```
public static void rotate(List<?> list, int distance) {
    if (list instanceof RandomAccess || list.size() < ROTATE_THRESHOLD)
        rotate1(list, distance);
    else
        rotate2(list, distance);
}
```

同样，rotate方法也是分为两种情况，如果集合支持随机访问或者集合大小小于旋转的阈值，则执行rotate1操作；否则，执行rotate2操作。

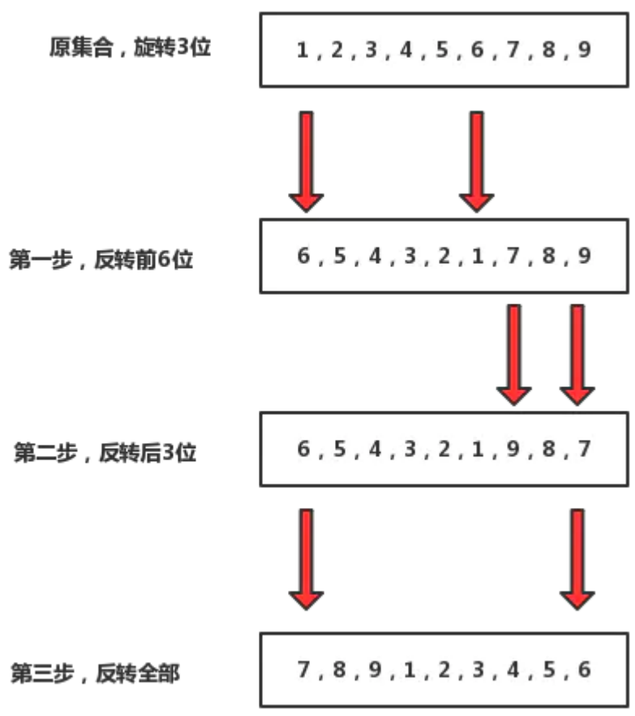
```
private static <T> void rotate1(List<T> list, int distance) {
    int size = list.size();
    if (size == 0)
        return;
    // 距离取余，计算实际要移动距离
    distance = distance % size;
    // 考虑有可能是负数
    if (distance < 0)
        distance += size;
    if (distance == 0)
        return;
    // 循环移动
    for (int cycleStart = 0, nMoved = 0; nMoved != size; cycleStart++) {
        T displaced = list.get(cycleStart);
        int i = cycleStart;
        do {
            // 通过distance来确定下标
            i += distance;
            if (i >= size)
                i -= size;
            displaced = list.set(i, displaced);
            // nMoved是最终移动的次数
            nMoved ++;
```

而对于rotate2方法，则是借助于反转方法reverse方法来进行操作的。

```
private static void rotate2(List<?> list, int distance) {
    int size = list.size();
    if (size == 0)
        return;
    int mid =  -distance % size;
    if (mid < 0)
        mid += size;
    if (mid == 0)
        return;

    reverse(list.subList(0, mid));
    reverse(list.subList(mid, size));
    reverse(list);
}
```


这个方法比较精妙。比如我们要对[1,2,3,4,5,6,7,8,9]进行3位旋转，则我们旋转的方式可以是：先对前size-3位进行反转，然后再对后3位进行反转，最后整体再进行反转就可以实现旋转的操作了。其中，mid的值就是确定要前后反转的中间值。
我们用一张图来看一下就明白了：



旋转操作.png

replaceAll方法

替换集合中的某一个元素为新的元素，可以替换null元素。该方法同样分为两种操作，如果集合支持随机访问或者集合大小小于要替换的阈值大小，使用对象的equals方法加list的set方法进行操作；否则，使用迭代器进行迭代操作。

```
public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal) {
    boolean result = false;
    int size = list.size();
    // 如果支持序列化或者集合大小小于替换的阈值11
    if (size < REPLACEALL_THRESHOLD || list instanceof RandomAccess) {
        // 如果旧值为null
        if (oldVal==null) {
            // 遍历数组，将为null的都替换为新的值
            for (int i=0; i<size; i++) {
                if (list.get(i)==null) {
                    list.set(i, newVal);
                    result = true;
                }
            }
        } else {
            // 不为null，遍历集合，通过equals方法进行判断
            for (int i=0; i<size; i++) {
                if (oldVal.equals(list.get(i))) {
                    list.set(i, newVal);
                    result = true;
                }
            }
        }
    } else {
        // 获取迭代器，使用迭代器进行操作
        ListIterator<T> itr=list.listIterator();
        if (oldVal==null) {
            for (int i=0; i<size; i++) {
                if (itr.next()==null) {
                    // 通过迭代器的set方法设置新值
                    itr.set(newVal);
                    result = true;
                }
            }
        } else {
            for (int i=0; i<size; i++) {
                if (oldVal.equals(itr.next())) {
                    itr.set(newVal);
                    result = true;
                }
            }
        }
    }
    return result;
}
```

查找集合包含子集合的下标索引，如果查找不到则返回-1。indexOfSubList是查找第一次出现的索引，而lastIndexOfSubList则是查找最后一次出现的索引。这两个方法的性能都不是太好，都是一种属于暴力搜索的算法，并且这里用到了Java中循环标签的概念。


```
public static int indexOfSubList(List<?> source, List<?> target) {
    // 原集合大小
    int sourceSize = source.size();
    // 目标集合大小
    int targetSize = target.size();
    int maxCandidate = sourceSize - targetSize;
    // 如果原集合和目标集合都支持随机访问，或者原集合小于阈值
    if (sourceSize < INDEXOFSUBLIST_THRESHOLD ||
        (source instanceof RandomAccess&&target instanceof RandomAccess)) {
    nextCand:
        // 双层遍历
        for (int candidate = 0; candidate <= maxCandidate; candidate++) {
            for (int i=0, j=candidate; i<targetSize; i++, j++)
                if (!eq(target.get(i), source.get(j)))
                    // 使用循环标签跳转至最外层
                    continue nextCand;  // Element mismatch, try next cand
            // 全部匹配，返回索引
            return candidate;
        }
    } else {  // Iterator version of above algorithm
        ListIterator<?> si = source.listIterator();
    nextCand:
        // 使用迭代器来进行循环
        for (int candidate = 0; candidate <= maxCandidate; candidate++) {
            ListIterator<?> ti = target.listIterator();
            for (int i=0; i<targetSize; i++) {
                if (!eq(ti.next(), si.next())) {
                    // 游标前移
                    for (int j=0; j<i; j++)
                        si.previous();
                    continue nextCand;
                }
            }
            return candidate;
        }
    }
    // 查询不到，返回-1
    return -1;
}
```

unmodifiable**方法**

Collections提供了一系列以unmodifiable开头的方法，用来在原集合基础上生成一个不可变的集合。比如unmodifiableSet，unmodifiableSortedMap等等。

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<> ();
    list.add(1);
    list.add(2);
    list.add(3);
    List list1 = Collections.unmodifiableList(list);
    list1.add(4);
    System.out.println(list1);
}
```

output：

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
    at com.jdk8.ListTest.main(ListTest.java:19)
```

我们大致看一下unmodifiableList的几个方法，来看一下它是不和保证不可变的。

```
public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
public E remove(int index) {
    throw new UnsupportedOperationException();
}
```

方法很简单，就是在调用一些增删改方法的时候，直接抛异常来保证不可变。

synchronized**方法**

Collections也提供了一系列以synchronized开头的方法，用来将原集合转成一个线程安全的集合。比如synchronizedList，synchronizedMap等。我们来大致看下synchronizedList的实现。

```
public static <T> List<T> synchronizedList(List<T> list) {
    return (list instanceof RandomAccess ?
            new SynchronizedRandomAccessList<>(list) :
            new SynchronizedList<>(list));
}
```

可以看到，synchronizedList底层调用了Collections的静态内部类SynchronizedList。再看下SynchronizedList：

```
static class SynchronizedList<E> extends SynchronizedCollection<E>
    implements List<E> {
```

```
    }
    public E set(int index, E element) {
        synchronized (mutex) {return list.set(index, element);}
    }
    public void add(int index, E element) {
        synchronized (mutex) {list.add(index, element);}
    }
}
```


可以看到，SynchronizedList内部的每个方法基本都使用了synchronized关键字，mutex是要同步的对象，位于SynchronizedCollection中。

```
final Object mutex;     // Object on which to synchronize
```

checked**方法**

Collections提供了一系列以checked开头的方法，用于获取动态类型安全的集合，常用于泛型相关操作。比如说当我们想往集合中插入一组数据的时候，除了可以明确指定数据的类型（List<Integer>），也可以使用Collections的checked方法来检查类型安全。比如：

```
public static void main(String[] args) {
    List list = new ArrayList<> ();
    list.add(1);
    list.add(2);
    list.add(3);
    List list1 = Collections.checkedList(list, String.class);
    list1.add(4);
    System.out.println(list1);
}
```

我们通过 `Collections.checkedList(list, String.class);` 方法将list1的对象设置为了字符串类型，如果再传入其他类型的值，将会抛出异常：

```
Exception in thread "main" java.lang.ClassCastException: Attempt to insert class java.lang.Int
    at java.util.Collections$CheckedCollection.typeCheck(Collections.java:3037)
    at java.util.Collections$CheckedCollection.add(Collections.java:3080)
    at com.jdk8.ListTest.main(ListTest.java:19)
```

需要注意的是，checked方法只会检查新插入的元素，并不会校验列表中已经存在的元素。如果我们有需要，可以新创建一个新的checked列表，并调用addAll方法重新插入所有元素进行校验。

所以它的用处也大概有两点：

> 1. 检查类型安全，比如我们上面使用的方式；
> 2. 在某种程度上也可以用作调试工具，来查找代码在哪里插入了错误类型的类，以防出现这种类型转换问题但却无法找出其中的原因的这种情况。可参考：
>   What is the Collections.checkedList() call for in java?

empty**方法**

Collections也提供了一系列以empty开头的方法，用户获取空的集合。比如emptySet，emptyList，emptyMap等方法。

```
public static final <T> List<T> emptyList() {
    return (List<T>) EMPTY_LIST;
}
```

当然，获取到的集合是无法修改的。一般用于接口返回空的数据。

frequency**方法**

该方法用于获取某一个元素在集合中出现的次数，并且可以统计null，底层通过遍历比较来实现。

```
public static int frequency(Collection<?> c, Object o) {
    int result = 0;
    if (o == null) {
        for (Object e : c)
            if (e == null)
                result++;
    } else {
        for (Object e : c)
            if (o.equals(e))
                result++;
    }
    return result;
}
```

disjoint**方法** 查看两个集合中有没有相同的元素

> public static boolean disjoint(Collection<?> c1, Collection<?> c2)

> 1. 两个参数都不能为null，否则抛出空指针异常；
> 2. 如果两个参数传递相同的集合，这种情况下，如果集合是空的，在返回true；否则返回false；

先简单看一下例子：

```
public static void main(String[] args) {
    List<Integer> srcList = new ArrayList<>(5);
    srcList.add(1);
    srcList.add(2);
    srcList.add(3);

    List<Integer> destList = new ArrayList<>(10);
    destList.add(1);
    destList.add(4);
    destList.add(5);

    // check elements in both collections
    boolean isCommon = Collections.disjoint(srcList, destList);
    System.out.println("No commom elements: "+isCommon);
}
```

output：

```
No commom elements: false
```

我们来看一下源码：

```
public static boolean disjoint(Collection<?> c1, Collection<?> c2) {
    Collection<?> contains = c2;
    Collection<?> iterate = c1;

    // Performance optimization cases. The heuristics:
    //   1. Generally iterate over c1.
    //   2. If c1 is a Set then iterate over c2.
    //   3. If either collection is empty then result is always true.
    //   4. Iterate over the smaller Collection.
    if (c1 instanceof Set) {
        // Use c1 for contains as a Set's contains() is expected to perform
        // better than O(N/2)
        iterate = c2;
        contains = c1;
    } else if (!(c2 instanceof Set)) {
        // Both are mere Collections. Iterate over smaller collection.
        // Example: If c1 contains 3 elements and c2 contains 50 elements and
        // assuming contains() requires ceiling(N/2) comparisons then
        // checking for all c1 elements in c2 would require 75 comparisons
        // (3 * ceiling(50/2)) vs. checking all c2 elements in c1 requiring
        // 100 comparisons (50 * ceiling(3/2)).
        int c1size = c1.size();
        int c2size = c2.size();
        if (c1size == 0 || c2size == 0) {
            // At least one collection is empty. Nothing will match.
            return true;
        }

        if (c1size > c2size) {
            iterate = c2;
            contains = c1;
        }
    }
    // 遍历iterate集合，然后通过contains方法比较
    for (Object e : iterate) {
        if (contains.contains(e)) {
           // 发现了相同的元素，直接返回false
            return false;
        }
    }

    // 没有发现相同的元素
    return true;
}
```

底层使用了两个临时变量contains和iterate，iterate适合于数据量小的集合，因为要遍历iterate，而contains适用于数据量大的集合，因为可以使用集合的contains方法。这样做的原因是因为contains方法的复杂度要比遍历iterate的复杂度低，这是一种简单的优化方式。

addAll**方法**

> public static <T> boolean addAll(Collection<? super T> c, T... elements)

用于向集合中添加多个元素，其中elements是一个可变参数，可以传递多个值。

```
public static <T> boolean addAll(Collection<? super T> c, T... elements) {
    boolean result = false;
    for (T element : elements)
        result |= c.add(element);
    return result;
}
```


方法很简单，底层通过循环调用集合的add方法来实现，然后通过位与运算来返回是否添加成

这个方法是基于指定的Map对象创建一个新的Set对象，它持有这个Map对象的引用，并且可以保持Map的顺序，并发和性能特征。在调用该方法时，指定的map对象必须是空的，并且对象的value属性是boolean类型。一般对这个类的最佳实践如下：

```
Set<String> concurrentSet = Collections.newSetFromMap(new IdentityHashMap<String, Boolean>());
```


由于在 `java.util.concurrent` 包下没有线程安全的ConcurrentHashSet的实现，我们可以借助于ConcurrentHashMap来实现，而另一种实现方式就是通过这个方法；

```
Set<String> concurrentSet = Collections.newSetFromMap(new ConcurrentHashMap<String, Boolean>()
```

reverseOrder**方法**

> public static <T> Comparator<T> reverseOrder()

返回对象集合排序的自然排序的逆序，通常我们可以如下使用：

```
Collections.sort(list, Collections.reverseOrder());
```

我们使用一个简单的例子来实现，比如：

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(5);
    list.add(3);
    System.out.println(list);
    Collections.sort(list, Collections.reverseOrder());
    System.out.println(list);
}
```

output：

```
[1, 5, 3]
[5, 3, 1]
```

同样，该方法也有一个重载方法：

```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

该方法就是返回我们所传入的比较器的逆序排序比较器。

其他方法

Collections中还有一些方法，比如 `singletonList(T o)` ，返回包含指定对象的不可变列表；`asLifoQueue` 将Deque转成后进先出（LIFO）队列，也就是栈结构； `nCopies` ，返回包含指定对象的n个不可变的列表；还有一些使用不太多的方法就不多说了。
比如：

```
public static void main(String[] args) {
    List<String> list = Collections.nCopies(5, "Java");
    for (String object : list) {
        System.out.println("object : " + object);
    }
}
```

打印：

```
object : Java
object : Java
object : Java
object : Java
object : Java
```

总结

本篇文章学习了Collections中几乎所有的public方法，通过学习这些方法，让我们以后在对集合进行操作的时候，可以想到使用Collections的这些方法。并且有些方法的源码实现很精妙，我们在写代码的时候也可以参考以一下。

如果有些方法大家不知道怎么使用，可以去stackoverflow上面搜索一下就可以了。

本文参考自：

https://docs.oracle.com/javase/8/docs/api/

Java 8, Implementing a ConcurrentHashSet

Collections.newSetFromMap(»ConcurrentHashMap«) vs. Collections.synchronizedSet(»HashSet«)