



1. Collection接口(扩展了Iterable接口，所以都可以使用for each循环)。

定义了：

Add

Iterator<E> iterator

int size

boolean isEmpty

boolean contains

boolean containsAll(Collection<?> other)

boolean add(Object)

boolean addAlll(Collection<?> other) 如果改变了这个集合，就返回true

boolean remove(Obejct)

boolean removeAll(Collection<?> other)如果改变了这个集合，就返回true

boolean removeIf(Predicate<? Super E> filter) 8从这个集合删除filter返回true的所有元素，如果改变了这个集合，就返回true

clear() 从这个集合中删除所有的元素

retainAll(Collection<?> other)从这个集合中删除所有于other集合中的元素不同的元素，如果改变了这个集合，就返回true

Object[] toArray() 返回这个集合的对象数组

<T> T[] toArray(T[] arrayToFill) 返回这个集合的对象数组,如股票arrayToFill足够大，就将集合中的元素填入这个数组中，剩余空间填补null；否则，分配一个新数组，其成员类型与arrayToFIll的成员类型相同，其长度等于集合的大小，并填充集合元素。

1. Iterator接口包含4个方法

E next、boolean hasNext、remove删除上次调用next返回的元素。

Java 8中，定义了forEachRemaining方法只要提供lambda表达式，将对迭代器的每一个元素都调用这个lambda

Iterator.forEachRemaining(element -> do something with element);

1. 编译器将for each循环翻译为带有迭代器的循环。

for each 循环可以于任何实现了Iterable接口的对象一起工作，这个接口只包含一个抽象方法：

Iterator iterator();

1. Map

V put(K key, V value)

V get(K key)

default V getOrDefault(K key, V defaultValue) 如果获取不到，则返回默认值

putAll(Map<? Extends K, ? extends V> entries) 将给定映射中的所有条目添加到这个映射中

boolean containsKey(K key)

boolean containsValue(V value)

default forEach(BiConsumer<? Super K, ? super V> action) 8 对这个映射中的所有键/值对应用这个动作

views视图，实现了Collection接口或者某个子接口的对象

映射到的视图：

Set<K> keySet()键集，因为keySet实现了Collection接口，所以可以像使用集合一样使用keySet，如for each循环。

在keySet上调用remove方法，实际上会从映射中删除这个键和与它关联的值。

Collection<V> values()值集

Set<Map.Entry<K,V>> entrySet()键值集

Map.Entry

K getKey

V getValue

V setValue(V newValue) //将映射中的值改为新值，并返回原来的值

putIfAbsent(K, V) 如果K不存在，将把K与A关联推进去

merge(K, V, BiFunction<? Super V, ? super V, ? extends V> remappingFunction) 8 如果K不存在，将K与A关联推进去，否则使用function组合原值与V，如果function计算的结果为null，则删除这个键。

Default V compute(K, BiFunction )8 将函数应用到key和get(key) 。将key与结果关联，如果结果为null，则删除key。返回get(key)。

Default V compteIfPresent(K, BiFunction) 如果key与一个非null值v关联，将函数应用到key和v，将key与结果关联，或者如果结果为null，则删除这个键。返回get(key)。

Default V computeIfAbsent(K, Function) 8 将函数应用到key，除非key与一个非null值关联。将key与结果关联，或者如果结果为null，则删除这个键。返回get(key).

Default void replaceAll(BiFunction)8 在所有映射项上应用函数。将键值与非null结果关联，对于null结果，则将相应的键删除。

HashMap

HashMap(int initialCapacity) 构造一个空的具有指定容量（桶数）的散列表

HashMap(int initialCapacity, float loadFactor) 构造一个具有指定容量和装填因子(0.5 – 1.0)的空散列表

TreeMap

TreeMap(Map<? Extends K, ? extends V> entries) 构造一个树映射，并添加一个集合或有序集合中的所有元素

TreeMap(SortedMap<? Extends K, ? extends V> entries) 构造一个树映射，并添加一个集合或有序集合中的所有元素（对于后一种情况，要使用同样的顺序），并使用与给定的有序映射相同的比较器

TreeMap(Comparator<? Super K > comparator)

SortedMap

Comparator<? Super K> comparator() 返回用于对元素进行排序的比较器。如果元素用Comparable接口的compareTo方法进行比较则返回null

E first()

E last() 返回映射中的最小元素或最大元素

//从指定key到指定key,也是和父类同个引用，只是参数范围不同而已  
SortedMap sortedMap1 = sortedMap.subMap("key1", "key3");  
//从头到指定key  
SortedMap sortedMap2 = sortedMap.headMap("key3");  
//从尾到指定key  
SortedMap sortedMap3 = sortedMap.tailMap("key1");

WeakHashMap

周期性地检查队列，以便找出新添加的弱引用，并删除它。

NavigableMap

navigableMap.subMap("key1", true ,"key2", true);  
navigableMap.headMap("key1", true);  
navigableMap.tailMap("key1", true);

1. List

add(int index, E element)

addAll(int I, Collection<? Extends E> elements)

remove(int index)

E get(int index)

E set(int index, E element)

int indexOf(Object element) 返回与指定元素相等的元素在列表中第一次出现的位置，如果没有这样的元素将返回-1

int lastIndexOf(Object element) 返回与指定元素相等的元素在列表中最后一次出现的位置，如果没有这样的元素将返回-1

ListIterator<E> listIterator() 返回一个列表迭代器，以便访问列表中的元素

ListIterator<E> listIterator(int index) 返回一个列表迭代器，以便访问列表中的元素,index是第一次调用next返回的给定索引的元素。

1. ListIterator是Iterator的子接口，定义了：

add(E element) 用于在迭代器位置前面添加一个元素

set(E new Element) 用新元素取代next或previous上次访问的元素。

boolean hasPrevious() 当反向迭代列表时，还有可提供访问的元素，返回true

E previous()

int nextIndex()

int previousIndex()

1. Set
2. LinkedList 链表

LinkedList(Collection<? Extends E> elements) 创建一个链表，并将集合中的所有元素添加到这个链表中

addFirst(E)

addLast(E)

E getFirst()

E getLast()

E removeFirst()

E removeLast()

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。

ArrayList 是线性表（数组）  
get() 直接读取第几个下标，复杂度 O(1)  
add(E) 添加元素，直接在后面添加，复杂度O（1）  
add(index, E) 添加元素，在第几个元素后面插入，后面的元素需要向后移动，复杂度O（n）  
remove（）删除元素，后面的元素需要逐个移动，复杂度O（n）  
  
LinkedList 是链表的操作  
get() 获取第几个元素，依次遍历，复杂度O(n)  
add(E) 添加到末尾，复杂度O(1)  
add(index, E) 添加第几个元素后，需要先查找到第几个元素，直接指针指向操作，复杂度O(n)  
remove（）删除元素，直接指针指向操作，复杂度O(1)

1. HashSet

HashSet(Collection<? Extends E> elements)

HashSet(int initialCapacity) 构造一个空的具有指定容量（桶数）的散列表

HashSet(int initialCapacity, float loadFactor) 构造一个具有指定容量和装填因子(0.5 – 1.0)的空散列表

1. TreeSet

TreeSet(Collection<? Extends E> elements)

TreeSet(SortedSet<E > s) 构造一个树集，并添加一个集合或有序集合中的所有元素（对于后一种情况，要使用同样的顺序）

TreeSet(Comparator<? Super E> comparator)

1. SortedSet

Comparator<? Super E> comparator() 返回用于对元素进行排序的比较器。如果元素用Comparable接口的compareTo方法进行比较则返回null

E first()

E last() 返回有序集合中的最小元素或最大元素

//返回大于等于from且小于to的所有元素子集,也是和父类同个引用，只是参数范围不同而已  
SortedSet sortedSet1 = sortedSet.subSet("adsw","dassss");  
//从头到指定位置  
SortedSet sortedSet2 = sortedSet.headSet("dassss");  
//从尾到指定位置  
SortedSet sortedSet3 = sortedSet.tailSet("dassss");

1. NavigableSet

E higher(E value)

E lower(E value) 返回大于value的最小元素或小于value的最大元素，没有返回null

E ceiling(E value)

E floor(E value) 返回大于等于value的最小元素或小于等于value的最大元素，没有返回null

E pollFirst()

E pollLast 删除并返回这个集合中的最大元素或最小元素，这个集合为空时返回null

//是否包含两端的元素  
navigableSet.subSet("aaa", true ,"bbb", true);  
navigableSet.headSet("aaa", true);  
navigableSet.tailSet("aaa", true);

Iterator<E> descendingIterator() 返回一个按照递减顺序遍历集合中元素的迭代器

1. LinkedHashSet和LinkedHashMap 记住插入元素的顺序。

LinkedHashSet方法:

Protected boolean removeEldestEntry(Map.Entry<K,V> eldest) 如果想删除eldest元素，并同时返回true，就应该覆盖这个方法。如，映射超过了一点大小，或者其他条件，就返回true。

1. 集合包装器

//将一个数组封装成一个list  
//但是实际上也还是一个数组，只是外表看起来是个list而已，只能调用get和set方法  
List<String> stringList = Arrays.*asList*(strings);  
//创建一个K外部看起长度为100，100个都为DEFAULT的list  
// 但实际上只有一个DEFAULT的一个对象  
List<String> settings = Collections.*nCopies*(100, "DEFAULT");  
//生成一个只有一个元素的集，不需要为建立数据结构开销。  
Set<String[]> stringSet = Collections.*singleton*(strings);  
//建立一个空集  
Set<String> deepThoughts = Collections.*emptySet*();  
//取出第1-2个元素,为原list的子范围,此子范围的操作对父范围生效  
List<String> stringList2 = stringList1.subList(1,3);  
//清除子范围的元素，同时也会清除父范围的元素  
//因为两个list都是引用同一个集合，只是起始值和size不同而已  
stringList2.clear();

1. 可选集合类

SortedSet<String> sortedSet = new TreeSet<>();  
 sortedSet.add("dada");  
 sortedSet.add("dassss");  
 sortedSet.add("adsw");  
  
 SortedMap<String, String> sortedMap = new TreeMap<>();  
 sortedMap.put("key1", "v1");  
 sortedMap.put("key2", "v2");  
 sortedMap.put("key3", "v3");  
  
 NavigableSet<String> navigableSet = new TreeSet<>();  
 navigableSet.add("aaa");  
 navigableSet.add("bbb");  
 navigableSet.add("ccc");  
  
 NavigableMap<String, String> navigableMap = new TreeMap<>();  
 navigableMap.put("key1", "v1");  
 navigableMap.put("key2", "v2");  
 navigableMap.put("key3", "v3");  
  
  
 //将一个数组封装成一个list  
 //但是实际上也还是一个数组，只是外表看起来是个list而已，只能调用get和set方法  
 List<String> stringList = Arrays.*asList*(strings);  
 //创建一个K外部看起长度为100，100个都为DEFAULT的list  
 // 但实际上只有一个DEFAULT的一个对象  
 List<String> settings = Collections.*nCopies*(100, "DEFAULT");  
 //生成一个只有一个元素的集，不需要为建立数据结构开销。  
 Set<String[]> stringSet = Collections.*singleton*(strings);  
 //建立一个空集  
 Set<String> deepThoughts = Collections.*emptySet*();  
 //取出第1-2个元素,为原list的子范围,此子范围的操作对父范围生效  
 List<String> stringList2 = stringList1.subList(1,3);  
 //清除子范围的元素，同时也会清除父范围的元素  
 //因为两个list都是引用同一个集合，只是起始值和size不同而已  
// stringList2.clear();  
  
  
  
  
 //返回大于等于from且小于to的所有元素子集,也是和父类同个引用，只是参数范围不同而已  
 SortedSet sortedSet1 = sortedSet.subSet("adsw","dassss");  
 //从头到指定位置  
 SortedSet sortedSet2 = sortedSet.headSet("dassss");  
 //从尾到指定位置  
 SortedSet sortedSet3 = sortedSet.tailSet("dassss");  
  
 //从指定key到指定key,也是和父类同个引用，只是参数范围不同而已  
 SortedMap sortedMap1 = sortedMap.subMap("key1", "key3");  
 //从头到指定key  
 SortedMap sortedMap2 = sortedMap.headMap("key3");  
 //从尾到指定key  
 SortedMap sortedMap3 = sortedMap.tailMap("key1");  
  
 //是否包含两端的元素  
 navigableSet.subSet("aaa", true ,"bbb", true);  
 navigableSet.headSet("aaa", true);  
 navigableSet.tailSet("aaa", true);  
  
 navigableMap.subMap("key1", true ,"key2", true);  
 navigableMap.headMap("key1", true);  
 navigableMap.tailMap("key1", true);  
  
  
 sortedMap1.clear();  
  
  
 //产生不可修改视图,调用get，set等方法时会报错  
 Collections.*unmodifiableList*(stringList1);  
 Collections.*unmodifiableCollection*(stringList1);  
 Collections.*unmodifiableMap*(sortedMap);  
 Collections.*unmodifiableSet*(navigableSet);  
 Collections.*unmodifiableSortedMap*(sortedMap);  
 Collections.*unmodifiableSortedSet*(sortedSet);  
 Collections.*unmodifiableNavigableMap*(navigableMap);  
 Collections.*unmodifiableNavigableSet*(navigableSet);  
  
 //产生同步视图  
 Collections.*synchronizedList*(stringList1);  
 Collections.*synchronizedCollection*(stringList1);  
 Collections.*synchronizedMap*(sortedMap);  
 Collections.*synchronizedSet*(navigableSet);  
 Collections.*synchronizedSortedMap*(sortedMap);  
 Collections.*synchronizedSortedSet*(sortedSet);  
 Collections.*synchronizedNavigableMap*(navigableMap);  
 Collections.*synchronizedNavigableSet*(navigableSet);  
  
 //受查视图，如果类型不正确，将马上抛出错误  
 //比如在使用add时，会检测是否属于给定的类  
 //可以使用List<String> 等规避错误  
 Collections.*checkedList*(stringList1, String.class);  
 Collections.*checkedCollection*(stringList1, String.class);  
 Collections.*checkedMap*(sortedMap, String.class, String.class);  
 Collections.*checkedSet*(navigableSet, String.class);  
 Collections.*checkedSortedMap*(sortedMap, String.class, String.class);  
 Collections.*checkedSortedSet*(sortedSet, String.class);  
 Collections.*checkedNavigableMap*(navigableMap, String.class, String.class);  
 Collections.*checkedNavigableSet*(navigableSet, String.class);  
  
  
  
  
 //对集合进行排序，按自然排列  
 Collections.*sort*(stringList1);  
  
 //对集合进行排序，使用对于的Comparator  
 userList.sort(Comparator.*comparingInt*(User::getAge));  
 userList.sort(Comparator.*comparingDouble*(User::getAge));  
 userList.sort(Comparator.*comparingLong*(User::getAge));  
 //使其进行逆序排列  
 userList.sort(Comparator.*comparing*(User::getAge).reversed());  
  
  
 //返回降序比较器  
 Collections.*reverseOrder*();  
 //对数组进行随机排列，打乱  
 Collections.*shuffle*(userList);  
 Collections.*shuffle*(userList, new Random());  
 //进二分查找,需要数组已经有序  
 Collections.*binarySearch*(userList, new User(2,"321","23","12",21));  
  
 //Collection.removeIf 移除元素  
 //如果满足lambada表达式的话，如果修改了集合的话返回true  
 stringList1.removeIf(w -> w.length() == 3);  
 //Collection.replaceAll 对集合里的元素都调用这个lambda  
 stringList1.replaceAll(String::toLowerCase);  
 //Collection.removeAll  
 //删除出现在参数集合里面的元素  
// stringList1.removeAll(stringList3);  
 //求两个集合的交集  
 Set<String> result = new HashSet<>(stringList1);  
 //求同时在两个集合都有的元素  
 result.retainAll(stringList3);  
  
  
  
 //Collections.min  
 //返回集合中最小的元素  
 Collections.*min*(stringList1);  
 //Collections.max  
 //返回集合中最大的元素  
 Collections.*min*(stringList1);  
 //Collections.min  
 //返回集合中最小的元素  
 Collections.*min*(stringList1, Comparator.*naturalOrder*());  
 //Collections.max  
 //返回集合中最大的元素  
 Collections.*min*(stringList1, Comparator.*naturalOrder*());  
  
 //用数组填充集合,如果集合被修改，返回true  
 Collections.*addAll*(destList, new User[userList.size()]);  
 //拷贝集合，是浅拷贝，在使用前需要先使用上面的语句  
 Collections.*copy*(destList, userList);  
 //填充列表的所有位置为相同的值  
 Collections.*fill*(destList, new User(1,"312","312","312",3));  
 destList.get(0).setName("wdwad");  
 //用netValue取代所有值为oldValue的元素  
 Collections.*replaceAll*(destList, new User(100,"312","11","312",3), new User(3,"ww00","aa","ee",5));  
  
  
 //获取被包含的子集合一段元素  
 Collections.*lastIndexOfSubList*(lists, stringList1);  
 //交换两个下标的元素  
 Collections.*swap*(stringList1, 1, 3);  
 //将集合逆序  
 Collections.*reverse*(stringList1);  
 //将集合旋转，也就是右移，如果多出的元素添加在左边  
 Collections.*rotate*(stringList1, 1);  
 //返回与元素相同的元素的个数,使用equals方法  
 int count = Collections.*frequency*(stringList1, "daada");  
 //如果两个集合没有相同的元素，返回true  
 boolean contains = Collections.*disjoint*(stringList1, stringList3);  
 //把一个数组转换成一个集合  
 List<String> asList = Arrays.*asList*(strings);  
 //把一个集合转换成数组  
 //带数组类型的会直接把参数加到数组new的数组里  
 //不带数组类型的会新创建一个数组  
 String[] strings1 = asList.toArray(new String[asList.size()]);  
 String[] strings2 = (String[]) asList.toArray();