## Java集合的学习

Java中常用的集合有List（链表）、Set（集合）、Map（键值映射）、Array（数组），各种集合类在内存中的组织形式不同，导致适用场景各不相同，这些类都在java.util包下。

List接口继承Collection接口

|  |
| --- |
| **public interface** List<E> **extends** Collection<E> |

Collection接口继承自Iterable接口

|  |
| --- |
| **public interface** Collection<E> **extends** Iterable<E> |

Iterable接口为最上层接口，实现该接口的类可以支持“for-each 循环遍历”

|  |
| --- |
| **public interface** Iterable<T> {  // 该方法返回元素的迭代器Iterator<T> iterator();  */\*\*  \** ***@since*** *1.8 传入action，可以为lamda表达式，*  <? **super** T>*表示包含T在内的任何T的父类，也就是说不能从中读取任何元素，因为无法保证类型。*<? extendsT> *表示包括T在内的任何T的子类，也就是说不能向列表添加任何元素，因为不确定子类，只能读取*  *泛型通配符的使用应该符合PECS原则：生产者（Producer）使用extends，消费者（Consumer）使用super。*  *生产者使用extends*  *如果你需要一个列表提供T类型的元素（即你想从列表中读取T类型的元素），你需要把这个列表声明成<? extends T>，比如List<? extends Integer>，因此你不能往该列表中添加任何元素。*  *消费者使用super*  *如果需要一个列表使用T类型的元素（即你想把T类型的元素加入到列表中），你需要把这个列表声明成<? super T>，比如List<? super Integer>，因此你不能保证从中读取到的元素的类型。*  *即是生产者，也是消费者,如果一个列表即要生产，又要消费，你不能使用泛型通配符声明列表，比如List<Integer>。  \*/* **default void** forEach(Consumer<? **super** T> action) {  Objects.*requireNonNull*(action);  **for** (T t : **this**) {  action.accept(t);  }  } **default** Spliterator<T> spliterator() {  **return** Spliterators.*spliteratorUnknownSize*(iterator(), 0);  } } |
|  |

Collection接口提供了集合常用的***Query Operations*、*Modification Operations、Bulk Operations。***

与Collection做比较，List接口没有覆盖Collection接口中提供的默认方法

|  |
| --- |
| **default boolean** removeIf(Predicate<? **super** E> filter) {  Objects.*requireNonNull*(filter);  **boolean** removed = **false**;  **final** Iterator<E> each = iterator();  **while** (each.hasNext()) {  **if** (filter.test(each.next())) {  each.remove();  removed = **true**;  }  }  **return** removed; } |

增加了替换方法

|  |
| --- |
| **default void** replaceAll(UnaryOperator<E> operator) {  Objects.*requireNonNull*(operator);  **final** ListIterator<E> li = **this**.listIterator();  **while** (li.hasNext()) {  li.set(operator.apply(li.next()));  } } |

增加了默认排序方法

|  |
| --- |
| **default void** sort(Comparator<? **super** E> c) {  Object[] a = **this**.toArray();  Arrays.*sort*(a, (Comparator) c);  ListIterator<E> i = **this**.listIterator();  **for** (Object e : a) {  i.next();  i.set((E) e);  } } |

增加抽象方法

|  |
| --- |
| **int** indexOf(Object o);  **int** lastIndexOf(Object o);  ListIterator<E> listIterator();  ListIterator<E> listIterator(**int** index); |

List接口的常用实现是ArrayList和Linkedlist。

其中ArrayList类继承关系为：



ArrayList的实现就是数组，当容量不足时，需要扩容，1.5倍，如果扩容1.5倍后的容量依然小于minCapacity，则直接设置为最小容量，继续判断，若此时新容量大于最大容量，则会调用*hugeCapacity方法计算新容量。*

|  |
| --- |
| **private void** grow(**int** minCapacity) {  *// overflow-conscious code* **int** oldCapacity = **elementData**.**length**;  **int** newCapacity = oldCapacity + (oldCapacity >> 1);  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity);  *// minCapacity is usually close to size, so this is a win:* **elementData** = Arrays.*copyOf*(**elementData**, newCapacity); } |

|  |
| --- |
| **private static int** hugeCapacity(**int** minCapacity) {  **if** (minCapacity < 0) *// overflow* **throw new** OutOfMemoryError();  **return** (minCapacity > ***MAX\_ARRAY\_SIZE***) ?  Integer.***MAX\_VALUE*** :  ***MAX\_ARRAY\_SIZE***; } |

在指定位置插入新元素的方法，System.*arraycopy是本地方法，具有更好的性能。*

|  |
| --- |
| **public void** add(**int** index, E element) {  rangeCheckForAdd(index);   ensureCapacityInternal(**size** + 1); *// Increments modCount!!* System.*arraycopy*(**elementData**, index, **elementData**, index + 1,  **size** - index);  **elementData**[index] = element;  **size**++; } |

其中有一个变量

|  |  |
| --- | --- |
| */\*\*  \* 该数字表示ArrayLisyt结构改变，比如改变size或其他在迭代器中的扰乱，主要哦用在迭代器中，检查是否有错误，多线程并发时有可能产生错误*   |  | | --- | | **private void** checkForComodification() {  **if** (**this**.**modCount** != **l**.**modCount**)  **throw new** ConcurrentModificationException(); } |   *\*/* **protected transient int modCount** = 0; |

ArrayList长度有限制，但是其本质是数组，所以适合查询、替换、排序操作较多的场景。

LinkedList的继承关系为：



比较ArrayList，RandomAccess接口被替换成Deque，RandomAccess即随机访问支持，同样也表示了访问特性，Deque继承自Queue，Queue的描述为

|  |
| --- |
| *Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning \* {****@code*** *true} upon success and throwing an {****@code*** *IllegalStateException}if no space is currently available.* |

其中的抽象方法为：

|  |
| --- |
| **boolean** add(E e);**boolean** offer(E e);E remove();E poll();E element();E peek(); |

提供了队列的各种操作。

而Deque继承自Queue，其描述为：

|  |
| --- |
| *A linear collection that supports element insertion and removal at \* both ends.* |

包含以下方法（增加了很多方法，堆栈、队列，都能使用）：

|  |
| --- |
| **void** addFirst(E e);**void** addLast(E e);**boolean** offerFirst(E e);**boolean** offerLast(E e);E removeFirst();E removeLast();E pollFirst();E pollLast();E getFirst();E getLast();E peekFirst();E peekLast();**boolean** removeFirstOccurrence(Object o);**boolean** removeLastOccurrence(Object o);  **boolean** add(E e); **boolean** offer(E e);  E remove();E poll();  E element();E peek();  **void** push(E e);E pop();  **boolean** remove(Object o);**boolean** contains(Object o); **public int** size();Iterator<E> iterator(); Iterator<E> descendingIterator(); |

一些使用上的区别：offer，add区别：

一些队列有大小限制，因此如果想在一个满的队列中加入一个新项，多出的项就会被拒绝。这时新的 offer 方法就可以起作用了。它不是对调用 add() 方法抛出一个 unchecked 异常，而只是得到由 offer() 返回的 false。

poll，remove区别：

remove() 和 poll() 方法都是从队列中删除第一个元素。remove() 的行为与 Collection 接口的版本相似，但是新的 poll() 方法在用空集合调用时不是抛出异常，只是返回 null。因此新的方法更适合容易出现异常条件的情况。

peek，element区别：

element() 和 peek() 用于在队列的头部查询元素。与 remove() 方法类似，在队列为空时， element() 抛出一个异常，而 peek() 返回 null

Map类主要提供键值对映射，常用的是HashMap和TreeMap，主要用于检索，查询效率高。