Collection 迭代器，增强for

|—— List 普通for

|—— ArrayList

|—— Vector

|—— LinkedList

|—— Set

|—— HashSet 无序，唯一

|—— LinkedHashSet 有序，唯一，底层是链表和哈希表

|—— TreeSet 排好序的，唯一

# Collection 接口必须实现15个方法：

1. **public** **boolean** add(Object e);

确保此 collection 包含指定的元素（可选操作）。

1. **public** **boolean** addAll(Collection c);

将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。

1. **public** **void** clear();

移除此 collection 中的所有元素（可选操作）。

1. **public** **boolean** contains(Object o);

如果此 collection 包含指定的元素，则返回 true。

1. **public** **boolean** containsAll(Collection c);

如果此 collection 包含指定 collection 中的所有元素，则返回 true。

1. **public** **boolean** equals(Object obj);

比较此 collection 与指定对象是否相等。

1. **public** **int** hashCode();

  返回此 collection 的哈希码值

1. **public** **boolean** isEmpty();

如果此 collection 不包含元素，则返回 true。

1. **public** Iterator iterator();

返回在此 collection 的元素上进行迭代的迭代器。

1. **public** **boolean** remove(Object o);

从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。

1. **public** **boolean** removeAll(Collection c);

移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。删除一个就返回true

1. **public** **boolean** retainAll(Collection c);

仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。两个A,B(A.retainAll(B))集合中有交集就返回true，无交集也返回true，结果集为[]。返回值表示的是A集合是否改变过。改变了为true，没改变为false。

1. **public** **int** size();

返回此 collection 中的元素数。

1. **public** Object[] toArray();

返回包含此 collection 中所有元素的数组。

1. **public** Object[] toArray(Object[] a);

返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

# 方法分类：

## 按功能分类：

1. 添加功能

**public** **boolean** add(Object e);

**public** **boolean** addAll(Collection c);

1. 删除功能

**public** **void** clear();

**public** **boolean** remove(Object o);

**public** **boolean** removeAll(Collection c);

1. 判断功能

**public** **boolean** contains(Object o);

**public** **boolean** containsAll(Collection c);

**public** **boolean** isEmpty();

1. 遍历功能

**public** Iterator iterator();

1. 长度功能

**public** **int** size();

1. 交集功能

**public** **boolean** retainAll(Collection c);

1. 集合转数组

**public** Object[] toArray();

**public** Object[] toArray(Object[] a);

## 按难易程度分类：

1. 简单方法：

**public** **boolean** add(Object e);

**public** **void** clear();

**public** **boolean** contains(Object o);

**public** **boolean** isEmpty();

**public** **boolean** remove(Object o);

**public** **int** size();

1. 带All的方法：

**public** **boolean** addAll(Collection c);

**public** **boolean** containsAll(Collection c);

**public** **boolean** removeAll(Collection c);

**public** **boolean** retainAll(Collection c); //较难理解

1. 遍历：

**public** Object[] toArray();

**public** Object[] toArray(Object[] a);

**public** Iterator iterator();

# Iterator

## 迭代器的基本原理

迭代器是依赖于集合而存在的，集合的总接口中需要实现一个返回迭代器的方法。

迭代器每一个具体实现子类实在每一个集合类里面的内部类定义的。

Iterator it = c.iterator();

//返回的是接口的子类对象 多态

遍历需要两个方法的帮助：

**public boolean [hasNext](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/Iterator.html" \l "hasNext())**(); 如果仍有元素可以迭代，则返回 true。

**public E [next](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/Iterator.html" \l "next())**(); 返回迭代的下一个元素。

## 为什么迭代器是一个接口，而不是一个具体的类呢？迭代器怎么实现的？

假设迭代器是一个具体的类。那么，这个具体的类的方法就必须是具体的。

但是，我们的集合数据结构不同，所以有很多种。

并且，由于数据结构不同，各种的集合的存储方式和获取方式是不一样的。

所以，用一个具体实现类来表示所有的集合的遍历方式是不可以的。

所以迭代器不能是一个具体的类。

但是，无论是哪种集合，都肯定有获取的功能。在创建哪种集合的对象的时候，该集合对象就应该提供对应的遍历方式。具体的类中提供了具体的获取方式。那么，这种方式怎么体现的呢？

内部类，实现三个方法

## 迭代器的实现源码：

|  |
| --- |
| interface Iterator{  boolean hasNext();  E next();  void remove();  }  interface Conllection{  Iterator<E> iterator();  }  interface List extends Collection {    }  class ArrayList implement List {  public Iterator iterator() {  return new Itr();  }    private class Itr implements Iterator<E> {  int cursor; // index of next element to return  int lastRet = -1; // index of last element returned; -1 if no such  int expectedModCount = modCount;  public boolean hasNext() {  return cursor != size;  }  @SuppressWarnings("unchecked")  public E next() {  checkForComodification();  int i = cursor;  if (i >= size)  throw new NoSuchElementException();  Object[] elementData = ArrayList.this.elementData;  if (i >= elementData.length)  throw new ConcurrentModificationException();  cursor = i + 1;  return (E) elementData[lastRet = i];  }  public void remove() {  if (lastRet < 0)  throw new IllegalStateException();  checkForComodification();  try {  ArrayList.this.remove(lastRet);  cursor = lastRet;  lastRet = -1;  expectedModCount = modCount;  } catch (IndexOutOfBoundsException ex) {  throw new ConcurrentModificationException();  }  }  @Override  @SuppressWarnings("unchecked")  public void forEachRemaining(Consumer<? super E> consumer) {  Objects.requireNonNull(consumer);  final int size = ArrayList.this.size;  int i = cursor;  if (i >= size) {  return;  }  final Object[] elementData = ArrayList.this.elementData;  if (i >= elementData.length) {  throw new ConcurrentModificationException();  }  while (i != size && modCount == expectedModCount) {  consumer.accept((E) elementData[i++]);  }  // update once at end of iteration to reduce heap write traffic  cursor = i;  lastRet = i - 1;  checkForComodification();  }  final void checkForComodification() {  if (modCount != expectedModCount)  throw new ConcurrentModificationException();  }  }  } |

# List

## list基础

特点：元素有序（元素的存储顺序和取出顺序一致）

可以重复

## ListIterator 与 Iterator的区别：

ListIterator是List类特有的，Iterator是所有集合共有的。ListIterator是Iterator的子类，Iterator有的方法ListIterator也有。Iterator只能从前往后遍历，而ListIterator可以从任意方向遍历（不过ListIterator只能先逆向遍历完才能正向，因为指针默认指向头结点）

## Iterator的并发修改异常问题：

需求：请判断集合中是否存在“list”这个元素，如果有就添加一个元素叫“zhaoliu”

实现代码:

|  |
| --- |
| Iterator it = new Iterator();  while(it.hasNext()){  String s = (String) it.next();  if(“lisi”.equals(s)){  list.add(“zhaoliu”);  }  } |

会出现ConcurrentModificationException异常。

为什么会出现？

迭代器是依赖于集合而存在的，在通过迭代器迭代的过程中，用集合往集合中添加了元素。而并没有重新获取迭代器，所以会报错。

原因：在迭代器迭代的过程中，是不能通过集合修改集合的。

解决方案：

1. 不用迭代器，用集合遍历集合，即for循环，可添加带末尾。
2. 用迭代器迭代，用迭代器修改元素。遍历到哪里，添加到哪里，代码为将list.add()，改为it.add()

## List三个儿子的特点：

用哪个看需求：

安全：Vector

不安全：ArrayList,LinkedList

查询多：ArrayList

增删多：LinkedList

### ArrayList

底层数据结构是数组，查询快，增删慢。

线程不安全，效率高。

### Vector

底层数据结构是数组，查询快，增删慢。

线程安全，效率不高。

Vector特有功能：

添加方法：

public void addElement([E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/Vector.html" \o "Vector 中的类型参数) obj)

遍历元素：

public [E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/Vector.html" \o "Vector 中的类型参数) elementAt(int index)

### LinkedList

可以当链表使，也可以当队列，当实例化Queue时，Queue接口窄化了对LinkedList的方法的访问权限

底层数据结构是双向链表，查询慢，增删快。

线程不安全，效率高。

LinkedList特有功能：

添加元素：

public void addFirst([E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) e)

public void addLast([E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) e)

删除元素：

public [E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) removeFirst()

public void addFirst([E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) e)

获取元素：

public [E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) getFirst()

public [E](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/LinkedList.html" \o "LinkedList 中的类型参数) getLast()

# Set

特点：元素无序，唯一。

## Hashset

hashset底层是hashmap

hashset存储自定义对象来保证唯一性时，

自定义对象需要重新写hashCode()函数和equals()函数。

因为在源码中可以看到(k = e.key) == key || (key != null && key.equals(k)))，使得它执行顺序为：

首先判断对象的哈希值是否相同，

如果相同，就继续执行equals()方法

如果equals()方法返回true，说明元素重复了。就不添加。

如果equals()方法返回false。说明没有重复，就添加到集合中。

如果不同，就直接添加到集合

让hashCode方法return一个固定的值。优化性的就是让hash返回的值不一样，提高效率。

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((name == null) ? 0 : name.hashCode());

return result;

}

见到hash的集合，类就需要重写两个方法：hashCode()和equals()方法

## TreeSet

使用元素的[自然顺序](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/lang/Comparable.html" \o "java.lang 中的接口)对元素进行排序，或者根据创建 set 时提供的 [Comparator](mk:@MSITStore:D:\\study\\JDK_API_1_6完整中文.CHM::/java/util/../../java/util/Comparator.html" \o "java.util 中的接口) 进行排序，具体取决于使用的构造方法。 无参数是自然排序。

具有的特点：

唯一：根据Comparable接口的compareTo(Student o)方法返回值是否为0来决定。如果返回值是0，说明元素重复，就不在添加元素了。

排序：根据Comparable接口的compareTo(Student o)方法返回值决定

正：说明数据大，往后放。

0：说明重复，不添加

负：说明数据小，往前放

TreeSet底层是红黑树结构

TreeSet集合，采用无参构造方法的时候，要让集合实现元素所属类的Comparable接口。然后再重写该接口中的方法的时候，去按照需求来实现代码比较即可。

实现TreeSet排序的两种方式：

1.在每一个子类当中实现Comparable接口

2.在创建TreeSet对象的时候传入一个Comparable类进行比较

# map

## HashMap（Map接口的1.2版本实现）：

当类作为Key的值时，类中应该重写equals方法和hashCode方法。

HashMap允许使用null值和null键。（除了非同步和允许使用null之外，HashMap与Hashtable大致相同）

HashMap的出现是替代Hashtable出现的

## TreeMap：

当类作为Key的值时，实现TreeMap排序的两种方式：

1.在每一个子类当中实现Comparable接口

2.在创建TreeMap对象的时候传入一个Comparable类进行比较

Map<Student, String> treeMap = new TreeMap<Student, String>(

new Comparator<Student>(){

@Override

public int compare(Student o1, Student o2) {

return o1.getAge() - o2.getAge();

}

});

## Hashtable（Map接口1.2版本的实现）

Hashtable是将HashMap中的所有方法前面加上一个synchronized锁

hashmap和hashtable区别：

A：HashMap线程不安全，效率高。Hashtable线程安全，效率低

B：HashMap允许null键和null值。Hashtable不允许null键和null值

# concurrentHashMap