# Collection接口

Java.uitl

Collection是整个类集之中单值保存的最大父接口-即每次一可以向集合里保存一个对象

**定义:**

public interface Collection<E>

extends Iterable<E>

定义了几个常用的操作方法:

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 说明 |
| 1 | Public boolean add(E e) | 普通 | 向集合里保存数据 |
| 2 | Public boolean addAll(Collection<? extends E> c) | 普通 | 向集合里追加一个集合 |
| 3 | Public void clear() | 普通 | 清空集合，root=null |
| 4 | Public boolean contains(Object o) | 普通 | 判断是否包含某个元素(equals支持) |
| 5 | Public boolean isEmpty() | 普通 | 判断是否是空集合 |
| 6 | Public boolean remove(Object o) | 普通 | 删除对象(equals支持) |
| 7 | Public int size() | 普通 | 返回集合中元素的个数 |
| 8 | Public Object[] toArray() | 普通 | 将集合变为对象数组保存 |
| 9 | Public Iterator<E> iterator() | 普通 | 为Iterator实例化 |

# List接口

List是Collection的一个子接口

扩充的几个重要的方法:

|  |  |  |  |
| --- | --- | --- | --- |
| No | 方法名称 | 类型 | 说明 |
| 1 | Public E get(int index) | 普通 | 取得索引编号的元素 |
| 2 | Public E set(int index,E element) | 普通 | 修改指定索引编号的元素 |
| 3 | Public ListIterator<E> listIterator() | 普通 | 为ListIterator接口实例化 |

List接口有两个常见的子类ArrayList(90%),Vector(10%),旧时期的类

ArrayList和Vector的区别?

|  |  |  |  |
| --- | --- | --- | --- |
| No | 区别点 | ArrayList(90%) | Vector(10%) |
| 1 | 推出时间 | JDK1.2 | JDK1.0,旧的类 |
| 2 | 性能 | 异步处理 | 同步处理 |
| 3 | 数据安全 | 非线程安全 | 线程安全 |
| 4 | 输出 | Iterator,ListIterator,foreach | Iterator,ListIterator,foreach,Enumeration |

# Set接口

Set接口是Collection接口的子接口,Set接口不像List接口扩充很多方法,Set只是简单

的继承了Collection接口,没有List中的get方法

Set接口下有两个常见的子类:HashSet,TreeSet

Set接口特点:无重复元素

HashSet:无重复元素 + 无序

TreeSet:无重复元素 + 有序

注意:

1. TreeSet类主要是通过Comparable接口中的CompareTo()方法判断是否是重复数据,如果返回的是0,那么它就认为是重复数据,不会被保存,所以要将所有属性进行比较

2) HashSet判断两个元素的方法不能用Comparable,判断重复元素的方法有两步:

1. 取得哈希码:public int hashCode(),先判断对象的哈希码是否相同,依靠哈希码取得一个对象的内容

2 对象比较内容:public boolean equals(Object obj),将对象的属性进行一次的比较

只要是非排序的情况下,判断元素重复用equals和hashCode

**范例:TreeSet**

|  |
| --- |
| class test{  public static void main(String args[]) {  Set<Book> s = new TreeSet<Book>();  s.add(new Book("java开发",34.5));  s.add(new Book("java开发",34.5));  s.add(new Book("jsp开发",35.5));  s.add(new Book("Android",32.5));  System.out.print(s);  }  }  class Book implements Comparable<Book>{  private String title;  private double price;  public Book(String title, double price) {  this.title = title;  this.price = price;  }  @Override  public int compareTo(Book o) { //对所有属性进行比较  if(this.price > o.price){  return 1;  }else if(this.price <o.price){  return -1;  }else{  return this.title.compareTo(o.title);  }  }  @Override  public String toString() {  return "书名:" + this.title + ",价格:" + this.price;  }  } |

范例:HashSet

|  |
| --- |
| class test{  public static void main(String args[]) {  Set<Book> s = new HashSet<>();  s.add(new Book("java开发",34.5));  s.add(new Book("java开发",34.5));  s.add(new Book("jsp开发",35.5));  s.add(new Book("Android",32.5));  System.out.print(s);  }  }  class Book {  private String title;  private double price;  public Book(String title, double price) {  this.title = title;  this.price = price;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  Book book = (Book) o;  if (Double.compare(book.price, price) != 0) return false;  return title != null ? title.equals(book.title) : book.title == null;  }  @Override  public int hashCode() {  int result;  long temp;  result = title != null ? title.hashCode() : 0;  temp = Double.doubleToLongBits(price);  result = 31 \* result + (int) (temp ^ (temp >>> 32));  return result;  }  @Override  public String toString() {  return "书名:" + this.title + ",价格:" + this.price;  }  } |

hashCode()和equals()是IDE自动生成的

# 集合的四种输出

四种输出:Iterator(95%),ListIterator(0.05%),Enumenration(4.9%),foreach(0.05%)

## 4.1Iterator接口

Java.util

public interface Iterator<E>{

public boolean hasNext(); //这两个方法在Scanner里也有,Scanner是继承Iterator的

public E next();

}

Iterator实例化是通过Collection接口里的 public Iterator<E> iterator()方法

范例:使用Iterator输出集合

|  |
| --- |
| Set<String> s = new HashSet<String>();  s.add("a");  s.add("b");  s.add("c");  Iterator<String> it = s.iterator();  while(it.hasNext()){  String str = it.next();  System.out.println(str);  } |

## 4.2ListIterator:双向迭代

Iterator本身只能由前向后输出,但是ListIterator接口能由前向后和由后向前双向输出

,ListIterator是Iterator接口的子接口

**public interface ListIterator<E>**

**extends Iterator<E>**

**ListIterator是专门为List字节口定义的输出接口,List里的方法:**

**Public ListIterator<E> listIterator()**

ListIterator有两个方法:

1. 判断是否有前一个元素:public boolean hasPrevious();
2. 取得前一个元素:public previous();

范例:双向迭代

|  |
| --- |
| List<String> s = new ArrayList<String>();  s.add("a");  s.add("b");  s.add("c");  System.out.print("有前向后输出:");  ListIterator<String> it = s.listIterator();  while (it.hasNext()){  String str = it.next();  System.out.print(str+",");  }  System.out.print("\n由后向前输出:");  while(it.hasPrevious()){  System.out.print(it.previous() + ",");  } |

## 4.3foreach

|  |
| --- |
| List<String> s = new ArrayList<String>();  s.add("a");  s.add("b");  s.add("c");  for(String i : s){  System.out.print(i+ ",");  } |

操作很简单

## 4.4Enumeration接口

Enumeration是与Vector类一起在JDK1.0推出的输出接口:即最早的

Vector输出是通过Enumeration接口完成的

public interface Enumeration<E>{

public boolean hasMoreElements();//判断是否有下一个元素等同于hasNext()

public E nextElement();//取出当前元素等同于next()

}

要取得Enumeration接口的实例化对象,只能依靠Vector类,在Vector类里有个方法:

取得Enumeration接口对象:public Enumeration<E> elements();

范例:

|  |
| --- |
| Vector<String> s = new Vector<String>();  s.add("a");  s.add("b");  s.add("c");  Enumeration<String> enu = s.elements();  while (enu.hasMoreElements()){  System.out.print(enu.nextElement());;  } |

在一些古老的操作方法上,此接口依然要使用

# Map接口

用来保存关联(key=value)数据

public interface Map<K,V>

常用方法:

|  |  |  |  |
| --- | --- | --- | --- |
| NO | 方法名称 | 类型 | 说明 |
| 1 | Public V put(K key,V value) | 普通 | 向集合中保存数据 |
| 2 | Public V get(Object key) | 普通 | 根据key查找对象的value |
| 3 | Public Set<Map.Entry<K,V>> entrySet() | 普通 | 将Map集合转化为Set集合 |
| 4 | Public Set<K> keySet() | 普通 | 取出全全部的Key |

Map接口下有两个常用的子类:HashMap,Hashtable

## 5.1HashMap类

特点:

1. 无序
2. 出现了重复的key,会进行覆盖

**注意:如果key是自定义的类时,要实现hashCode和equals方法和HashSet一样,不过key的首选类型是String,自定义的类基本不用**

范例:HashMap基本操作

|  |
| --- |
| Map<String,Integer> map = new HashMap<String,Integer>();  map.put("一",1);  map.put("二",2);  map.put("三",3);  map.put("三",33);  map.put(null,10);  System.out.println(map.get("一")); //1  System.out.println(map.get(null)); //10  System.out.println(map.get("hah")); //如果key不存在,返回null  Set<String> keys = map.keySet();//全的全部的key  Iterator<String> iter = keys.iterator();  while(iter.hasNext()){  System.out.print(iter.next());  } |

Map存放数据的目的是为了信息的查找,但是Collection是为了输出

## 5.2Hashtable类

JDK1.0,属于最早的Map集合的是实现操作,在JDK1.2时多实现了一个Map接口

Hashtable里面对于key和value的数据都不允许为null

## 5.3Map的Iterator的输出

整个Map接口里并没有可以返回Iterator接口对象的方法

每当使用put方法向Map集合里保存一对数据的时候,实际上所有的数据都会被封装

为Map.Entry对象,即Map里保存的元素为Map.Entry,Map.Entry里保存的是key和value

Map.Entry接口定义:

public static interface Map.Entry<K,V>

在这个接口里定义了两个操作:

1. 取得key:public K getKey();
2. 取得value:public V getValue();

Map集合里有一个将Map集合转化为Set集合的方法:

**Public Set<Map.Entry<K,V>> entrySet()**

Map集合利用Iterator接口输出的步骤:

1. 利用Map接口的entrySet()将Map集合变为Set集合
2. 利用Set集合中的iterator方法将Set集合进行Iterator输出
3. 每次读取的都是Map.Entry接口对象,利用此对象进行key和value的取出

范例:利用Iterator实现Map接口的输出

|  |
| --- |
| Map<String,Integer> map = new Hashtable<>();  map.put("一",1);  map.put("二",2);  map.put("三",3);  map.put("三",33);  Set<Map.Entry<String,Integer>> s = map.entrySet();  Iterator<Map.Entry<String,Integer>>it = s.iterator();  while(it.hasNext()){  Map.Entry<String,Integer> entry = it.next();  System.out.println(entry.getKey() + "=" + entry.getValue());  } |

# 6.Stack类

Stack表示的是栈,栈是一种先进后出的数据结构,Stack是Vector的子类,但是不会使用Vector的方法,它使用自己的方法：

1）入栈:public E push(E item)

2）出栈:pop E pop()

**定义:**

public class Stack<E>

extends Vector<E>

范例:

|  |
| --- |
| Stack<String> all = new Stack<String>();  all.push("A");  all.push("B");  all.push("C");  System.out.println(all); //[A,B,C]  all.pop(); //C  System.out.println(all); //[A,B]  all.pop();//B  System.out.println(all); //[A]  all.pop();//A  System.out.println(all); //[]  //all.pop();//A //异常 |

|  |  |  |  |
| --- | --- | --- | --- |
| NO |  | 类型 |  |
| 1 |  | 普通 |  |
| 2 |  | 普通 |  |
| 3 |  | 普通 |  |
| 4 |  | 普通 |  |
| 5 |  | 普通 |  |