List 一组由顺序的元素，可以通过下标访问，是Collection子接口

ArrayList LinkedList

Set 一组取值唯一的元素，是Collection子接口

HashSet(自己写类必须提供hashCode, equals) LinkedHashSet（会记录放入顺序）

.add .remove .contains .size

.get

for(Object o : 集合) {

}

Iterator iter = 集合.iterator()

while(iter.hasNext()){

iter.next()

}

# SortedSet

（会把集合中的元素根据大小进行排序）

父接口是 Set, 比较典型的实现类：TreeSet

Set set = new TreeSet();

set.add(5);

set.add(4);

set.add(7);

set.add(1);

// 会按照升序（由小到大）对集合中的元素进行排序

System.out.println(set);

System.out.println("======= 遍历 ======");

for(Object o : set) {

System.out.println(o);

}

Set set2 = new TreeSet();

set2.add("5");

set2.add("4");

set2.add("7");

set2.add("11");

System.out.println(set2);

// 放入TreeSet集合的对象必须实现一个Comparable接口， 它可以让我们实现比较规则

Set set3 = new TreeSet();

set3.add(new Student("张三",20)); // 5f20

set3.add(new Student("钱七",17)); // 94b1

set3.add(new Student("王五",55)); // 738b

set3.add(new Student("李四",18)); // 674e

System.out.println(set3);

// java.lang.ClassCastException:

// day11.set.Student cannot be cast to java.lang.Comparable

public class Student implements Comparable{

String name;

int age;

...

public Student(String name, int age) {

super();

this.name = name;

this.age = age;

}

@Override

public int compareTo(Object o) {

// 比较student 谁大谁小

Student s2 = (Student) o;

// int r = this.name.compareTo(s2.name); // 按照姓名的unicode排序

// this.age > s2.age 返回 正

// this.age < s2.age 返回负数

// this.age == s2.age 返回0

// return this.age - s2.age; // 按年龄升序（由小到大）

return (s2.age - this.age); // 按年龄降序 (由大到小)

}

Comparable 能够配合TreeSet进行排序，缺点是：一个类里只能写一种比较规则，如果比较规则变化了，类也需要跟着变化。

# 排序

java.util.Collections （工具类） 可以对集合进行通用的排序操作，比TreeSet更为灵活

java.util.Collection (接口) List, Set 的父接口

java.util.Arrays （工具类） 针对数组进行排序

java中的排序算法

对基本类型数据进行排序，使用了快速排序的改进版, 如果数据量特别少，会使用插入排序

对集合进行 （List）进行排序，归并排序的改进版 （jdk6.0）

对集合进行 （List）进行排序，TimSort的改进版 （jdk7.0以后） 包括Python语言也在使用

java 中提供了优秀的排序算法，我们需要提供排序规则

Comparable 接口只能提供一种排序规则，不够灵活

Comparator (比较器) 接口，每个实现都是一个排序规则

## Collections

Collections.sort(List)

对list中数据进行排序， list中元素也需实现Comparable

Collections.sort(list, Comparator) 更为灵活

由 Comparator 提供一个比较规则，判断list中任意两个元素谁大谁小

姓名比较器：

import java.util.Comparator;

public class NameComparator implements Comparator {

@Override

// 实现比较方法：需要我们来实现这两个对象谁大谁小

// 返回0 表示o1=o2, 负数表示o1<o2, 正数表示o1>o2

public int compare(Object o1, Object o2) {

// 将Object 转换为待比较的Teacher类型

Teacher t1 = (Teacher) o1;

Teacher t2 = (Teacher) o2;

// 将两个teacher 的名字进行比较

return t1.name.compareTo(t2.name);

}

}

年龄比较器：

import java.util.Comparator;

public class AgeComparator implements Comparator {

@Override

public int compare(Object o1, Object o2) {

Teacher t1 = (Teacher) o1;

Teacher t2 = (Teacher) o2;

return t1.age - t2.age;

}

}

使用比较器：

List list = new ArrayList();

list.add(new Teacher("zhang", 30, 10000.0));

list.add(new Teacher("li", 28, 9000.0));

list.add(new Teacher("wang", 50, 19000.0));

list.add(new Teacher("zhao", 18, 7000.0));

// 按名字进行排序

Collections.sort(list, new NameComparator());

System.out.println(list);

// 按年龄排序

Collections.sort(list, new AgeComparator());

System.out.println(list);

// 倒序

Collections.reverse(list);

System.out.println(list);

// 用匿名内部类来实现比较器

// new 接口名|父类名() {

// 实现接口的方法或重写父类方法

// }

Comparator c = new Comparator() {

public int compare(Object o1, Object o2) {

Teacher t1 = (Teacher) o1;

Teacher t2 = (Teacher) o2;

return (int) (t1.salary - t2.salary);

}

};

Collections.sort(list, c);

System.out.println(list);

// 等价于：

Collections.sort(list, new Comparator() {

public int compare(Object o1, Object o2) {

Teacher t1 = (Teacher) o1;

Teacher t2 = (Teacher) o2;

return (int) (t1.salary - t2.salary);

}

});

// 数组排序

// Arrays.sort(数组); // String 基本类型，Date

// Arrays.sort(数组, 比较器); // 处理自定义类型

# Queue 队列 （FIFO 先进先出）

java.util.Queue 接口

java.util.LinkedList 也是Queue接口的实现

常见方法：

offer 入队 （把新元素加入队列尾部）

poll 出队 （找到队列头部元素，并移除）

peek （找到队列头部元素）

Queue queue = new LinkedList();

queue.offer(1);

queue.offer(2);

queue.offer(3);

queue.offer(4);

System.out.println(queue);

queue.offer(5); // 将新元素加入队列尾部

// 头 [1, 2, 3, 4, 5] 尾

System.out.println(queue);

// 找到头部元素

System.out.println("头部元素:" + queue.peek());

// 移除头部元素

queue.poll(); // 返回值就是头部元素

queue.poll();

queue.poll();

queue.poll();

queue.poll(); // 5

queue.poll(); // null

System.out.println(queue);

# Deque 双向队列

java.util.Deque

头部，尾部均可以添加和移除元素

常见方法:

offerFirst 向头部添加

offerLast 向尾部添加

peekFirst 取得头部元素

peekLast 取得尾部元素

pollFirst 移除头部元素

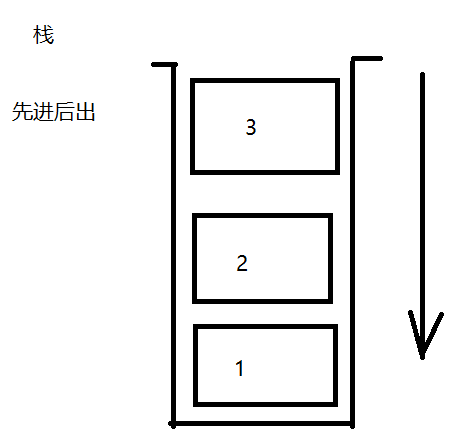
pollLast 移除尾部元素

实现类： LinkedLink

# 栈 Stack

Deque 也可以被当作栈 Stack FILO 先进后出

队列 ： FIFO 先进先出



栈的方法： push(对象) 将对象放入栈 （压栈）

pop() 从栈中弹出对象 (出栈)

peek() 获取栈顶元素

// 创建一个栈

Deque stack = new LinkedList();

stack.push("zhang");

stack.push("li");

stack.push("wang");

// 井口 [wang, li, zhang] 井底

System.out.println(stack);

// 移除栈顶元素

stack.pop();

// [li, zhang]

System.out.println(stack);

// 找到栈顶元素

System.out.println("栈顶元素：" + stack.peek());

System.out.println(stack);

# Map （映射,表)

把一些有关联的数据结对存储, 按键-值的方式存储于map集合当中

键（key） 值(value) 键值对

bj 北京

sh 上海

sz 深圳

根据key进行快速查找，找到值进行操作

java.util.Map 接口

常见实现: java.util.HashMap, java.util.LinkedHashMap

常用方法:

.put(key, value) ; // 用来存储键-值

.get(key) 返回value; // 根据key查找value值

.remove(key) // 根据key 删除键-值

map 的遍历方式：

按key遍历

按值遍历

按 Entry遍历

装载因子

当HashMap内容量达到某一阈值时，会执行数组容量翻倍，需要让hashMap中的元素重新分布，称为rehash

什么时候发生rehash： 当 实际元素个数超过 (默认值16) \* 装载因子（0.75）后

// 1. 创建map集合

Map map = new HashMap();

// new HashMap(128); // 指定初始容量

// new HashMap(128, 0.8f); // 指定初始容量以及装载因子

// 2. 添加数据

// 做为key的对象，必须实现hashCode和equals

map.put("bj", "北京");

map.put("sh", "上海");

map.put("sz", "深圳");

System.out.println(map);

map.put("bj", "首都"); // map 中的key不能重复

System.out.println(map);

// 3. 根据key找到value

System.out.println(map.get("bj"));

System.out.println("没有找到：" + map.get("gz")); //返回null值

// 4. 根据key删除键值

map.remove("sz"); // 删除深圳对应的键值

System.out.println(map);

System.out.println("====================");

// 5. 遍历

// 5.1) 根据key进行遍历

// map.keySet() 返回了饱和了所有key的set集合

for(Object key : map.keySet()) {

System.out.println("key:" + key + " value:" + map.get(key));

}

System.out.println("====================");

// 5.2) 根据值进行遍历

// map.values() 返回的是个包含了所有值的 Collection集合

for(Object value: map.values()) {

System.out.println("value:" + value);

}

System.out.println("====================");

// 5.3) 根据键值遍历

// Entry(key,value,next,hash ) 条目

// 一个map可被看作是多个Entry对象的组合

// map.entrySet() 返回所有Entry对象的集合

for(Object obj:map.entrySet()) {

Entry e = (Entry)obj; // 需要多加一步转换

System.out.println(e.getKey() + " " + e.getValue());

}

# LinkeHashMap

// hashMap不会记录键值的放入顺序

// Map map = new HashMap();

// LinkedHash 会记录键值的放入顺序

Map map = new LinkedHashMap();

map.put("aaa", "aaa");

map.put("bbb", "bbb");

map.put("ccc", "ccc");

map.put("ddd", "ddd");

System.out.println(map);

// 可以定义LinkedHashMap的子类限制集合中元素的个数

// 参数1：初始容量， 参数2：装载因子，参数3：get方法是否会影响元素的顺序

Map map2 = new LinkedHashMap(10, 0.75f, true) {

// 是否要删除最旧的entry元素

protected boolean removeEldestEntry(Map.Entry eldest) {

// 当集合大小超过5时，会删除最旧的元素

if(size()>5) {

return true;

} else {

return false;

}

}

};

map2.put("1", "1"); // 最旧的

map2.put("2", "2");

map2.put("3", "3");

map2.put("4", "4");

map2.put("5", "5");

System.out.println(map2);

map2.put("6", "6");

System.out.println(map2);

map2.get("2"); // 当使用了元素，那么这个元素就变为最新的

System.out.println(map2);

map2.put("7", "7"); // 当再放入新元素时，就把最旧的3元素挤掉了

System.out.println(map2);