# 第08天【集合-第一天】

## 主要内容

1. 集合框架体系
2. ArrayList
3. LinkedList
4. HashSet
5. TreeSet
6. LinkedHashSet
7. 内部比较器和外部比较器
8. 哈希表的原理

## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 集合的引入 | 掌握 |
| 集合框架结构 | 理解 |
| ArrayList的常用方法 | 掌握 |
| ArrayList的源码分析 | 掌握 |
| LinkedList的使用 | 掌握 |
| HashSet | 掌握 |
| TreeSet | 掌握 |
| LinkedHashSet | 理解 |
| 内部比较器 | 掌握 |
| 外部比较器 | 掌握 |
| 哈希表的原理 | 掌握 |

## 一、集合引入和ArrayList

### 1.1 引入集合

#### 1.1.1集合和数组的比较

**集合的使用场合-Where**

举例：新闻列表、就业喜报、就业明星、邮件列表、购物车；当我们需要将一些相同结构的个体整合在一起时，就可以考虑使用集合了 。

****

****

****

****

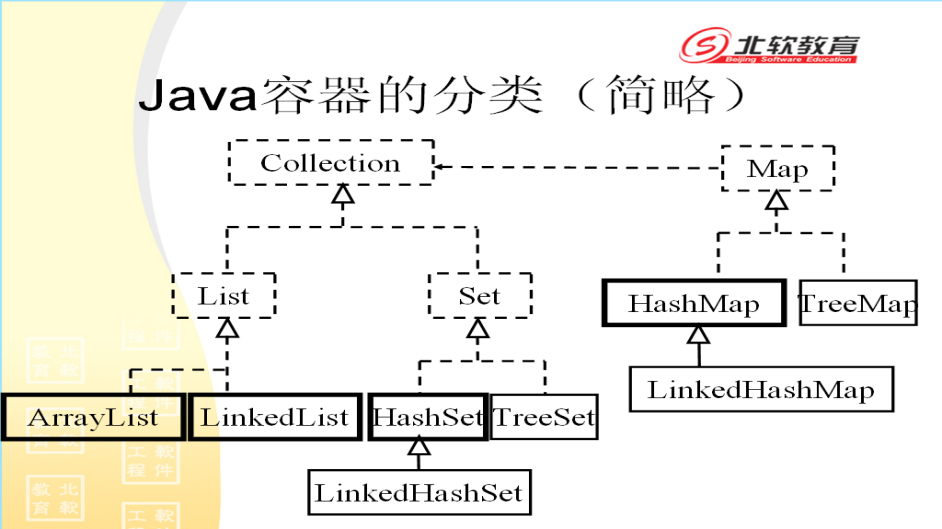
**为什么使用集合而不是数组—Why**

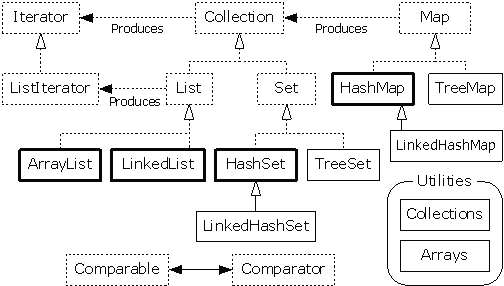
* + 集合和数组相似点
    - 都可以存储多个对象，对外作为一个整体存在
  + 数组的缺点
    - 长度必须在初始化时指定，且固定不变
    - 数组采用连续存储空间，删除和添加效率低下
    - 数组缺乏封装，操作繁琐

**Snap4**

#### 1.1.2集合框架

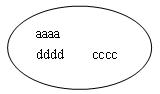
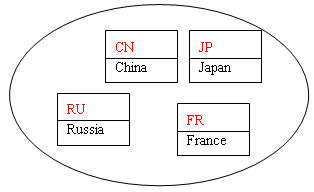
* Java集合框架提供了一套性能优良、使用方便的接口和类，它们位于java.util包中。存放在集合中的数据，被称为元素(element)





* 集合架构
  + Collection 接口存储一组不唯一，无序的对象
  + List 接口存储一组不唯一，有序（索引顺序）的对象
  + Set 接口存储一组唯一，无序的对象
  + Map接口存储一组键值对象，提供key到value的映射
    - Key 唯一 无序
    - value 不唯一 无序

Snap4

### 本节作业

1. 集合和数组的异同
2. Java的集合框架体系

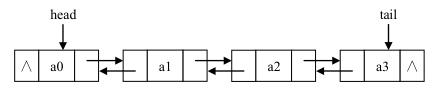
## 二、List集合

List集合的主要实现类有ArrayList和LinkedList，分别是数据结构中顺序表和链表的实现。另外还包括栈和队列的实现类：Deque和Queue。

* List
  + 特点：有序 不唯一（可重复）
* ArrayList
  + 在内存中分配连续的空间，实现了长度可变的数组
  + 优点：遍历元素和随机访问元素的效率比较高
  + 缺点：添加和删除需大量移动元素效率低，按照内容查询效率低

Snap4

* LinkedList
  + 采用双向链表存储方式。
  + 缺点：遍历和随机访问元素效率低下
  + 优点：插入、删除元素效率比较高（但是前提也是必须先低效率查询才可。如果插入删除发生在头尾可以减少查询次数）



### 2.1 ArrayList

#### 【示例1】使用ArrayList存储多个学生的分数

|  |
| --- |
| **public class** TestArrayList1 {  **public static void** main(String[] args) {  //创建一个ArrayList对象  ArrayList list = **new** ArrayList();   //向集合中添加多个分数  list.add(78); //加到最后   list.add(89);  list.add(56);  //list.add(new Integer(56));  list.add(89);  list.add(2, 100);//加到指定位置 底层发生了元素大量后移  //list.add(10,20);  ArrayList list2 = **new** ArrayList();  list2.add(60);  list2.add(58);  list2.add(29);  //list.addAll(list2);  list.addAll(0, list2);   //输出集合中分数  System.**out**.println(list.size());//4  System.**out**.println(list);//[]  //遍历1：使用for循环  System.**out**.println(**"遍历1：使用for循环"**);  **for**(**int** i=0;i<list.size();i++){  //获取第i个元素  **int** elem = (Integer)list.get(i);//自动拆箱  //输出第i个元素  System.**out**.println(i+**" "**+elem);  }  //遍历2：使用增强的for循环  System.**out**.println(**"遍历2：使用增强的for循环"**);  **for**(Object elem :list){  //System.out.println(elem);  Integer i = (Integer)elem;  System.**out**.println(i);  }  //遍历3：使用Iterator迭代器  System.**out**.println(**"遍历3：使用Iterator迭代器"**);  Iterator it = list.iterator();  **while**(it.hasNext()){//还有元素，没有就结束循环  //如果有，就取出来  **int** elem = (Integer)it.next();   //输出来  System.**out**.println(elem); }  } } |

|  |
| --- |
| **注意：新的JDK8遍历方式更加简单了**   * list.forEach((elem)->System.out.println(elem)); * list.forEach(System.out::println);   这其中涉及到流式编程、Lambda表达式、函数式接口、方法引用等JDK8新特征，稍后进行讲解，此处先会使用。 |

#### 【示例2】使用泛型保证集合操作的安全和简便

|  |
| --- |
| **public class** TestArrayList2 {  **public static void** main(String[] args) {  //创建一个ArrayList对象  ArrayList<Integer> list = **new** ArrayList<Integer>();  //向集合中添加多个分数  list.add(78); //加到最后   list.add(89);  list.add(56);  //list.add(new Integer(56));  list.add(89);  list.add(2, 100);  //list.add(10,20);  ArrayList<Integer> list2 = **new** ArrayList<Integer>();  list2.add(60);  list2.add(58);  list2.add(29);  list.addAll(0, list2);  //输出集合中分数  System.***out***.println(list.size());//4  System.***out***.println(list);//[]   //遍历1：使用for循环  System.***out***.println(**"遍历1：使用for循环"**);  **for**(**int** i=0;i<list.size();i++){  //获取第i个元素  **int** elem = list.get(i);//自动拆箱  //输出第i个元素  System.***out***.println(i+**" "**+elem);  }  //遍历2：使用增强的for循环  System.***out***.println(**"遍历2：使用增强的for循环"**);  **for**(Integer elem :list){  System.***out***.println(elem);  //Integer i = (Integer)elem;  //System.out.println(i);  }  //遍历3：使用Iterator迭代器  System.***out***.println(**"遍历3：使用Iterator迭代器"**);  Iterator<Integer> it = list.iterator();  **while**(it.hasNext()){//还有元素，没有就结束循环  //如果有，就取出来  **int** elem = it.next();  //输出来  System.***out***.println(elem);  }  } } |

#### 【示例3】ArrayList类的更多方法

|  |
| --- |
| **public class** TestArrayList3 {  **public static void** main(String[] args) {  //创建一个ArrayList对象  ArrayList<Integer> list = **new** ArrayList<Integer>();   //向集合中添加多个分数  list.add(78); //加到最后   list.add(89);  list.add(56);   //删除  //list.remove(0);  list.remove(**new** Integer(78));   //更新  list.set(1, 65);   //输出集合中分数  //list.clear();  System.**out**.println(list.contains(65));  System.**out**.println(list.toString());  } } |

|  |  |
| --- | --- |
| **方法摘要** | |
| boolean | **[add](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "add(E))**([E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) e)向列表的尾部添加指定的元素（可选操作）。 |
| void | **[add](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "add(int, E))**(int index, [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) element)            在列表的指定位置插入指定元素（可选操作）。 |
| boolean | **[addAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "addAll(java.util.Collection))**([Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<? extends [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数)> c)            添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序（可选操作）。 |
| boolean | **[addAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "addAll(int, java.util.Collection))**(int index, [Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<? extends [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数)> c)   将指定 collection 中的所有元素都插入到列表中的指定位置（可选操作）。 |
| void | **[clear](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "clear())**()   从列表中移除所有元素（可选操作）。 |
| boolean | **[contains](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "contains(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) o)  如果列表包含指定的元素，则返回 true。 |
| boolean | **[containsAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "containsAll(java.util.Collection))**([Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<?> c)            如果列表包含指定 collection 的所有元素，则返回 true。 |
| [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) | **[get](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "get(int))**(int index) 返回列表中指定位置的元素。 |
| boolean | **[isEmpty](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "isEmpty())**()  如果列表不包含元素，则返回 true。 |
| [Iterator](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Iterator.html" \o "java.util 中的接口)<[E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数)> | **[iterator](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "iterator())**()  返回按适当顺序在列表的元素上进行迭代的迭代器。 |
| [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) | **[remove](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "remove(int))**(int index)   移除列表中指定位置的元素（可选操作）。 |
| boolean | **[remove](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "remove(java.lang.Object))**([Object](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html" \o "java.lang 中的类) o)   从此列表中移除第一次出现的指定元素（如果存在）（可选操作）。 |
| boolean | **[removeAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "removeAll(java.util.Collection))**([Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<?> c)            从列表中移除指定 collection 中包含的其所有元素（可选操作）。 |
| boolean | **[retainAll](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "retainAll(java.util.Collection))**([Collection](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html" \o "java.util 中的接口)<?> c)            仅在列表中保留指定 collection 中所包含的元素（可选操作）。 |
| [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) | **[set](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "set(int, E))**(int index, [E](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \o "List 中的类型参数) element)  用指定元素替换列表中指定位置的元素（可选操作）。 |
| int | **[size](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/List.html" \l "size())**()  返回列表中的元素数。 |

### 2.2理解 ArrayList的源码

* ArrayList底层就是一个长度可以动态增长的Object数组；（StringBuilder底层就是一个长度可以动态增长的char数组）

|  |
| --- |
| **public class** ArrayList<E> **extends** AbstractList<E> **implements**  List<E>, RandomAccess, Cloneable, Serializable{  **private static final int *DEFAULT\_CAPACITY*** = 10;  **private static final** Object[] ***EMPTY\_ELEMENTDATA*** = {};  **private static final** Object[]  ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*** = {};  **transient** Object[] **elementData**;  **private int size**;  } |

* 接口是可以一个方法也不提供的，比如 RandomAccess, Cloneable, java.io.Serializable

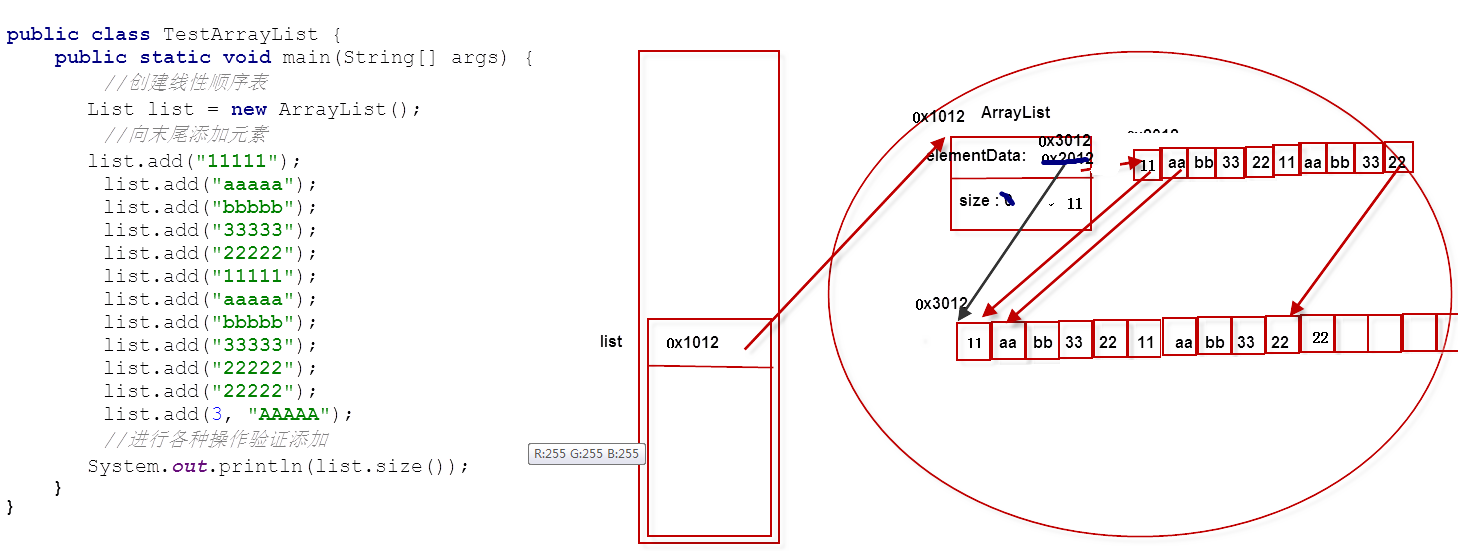
|  |
| --- |
| **public class** ArrayList<E> **extends** AbstractList<E>  **implements** List<E>,  RandomAccess,  Cloneable, java.io.Serializable{} |

* JDK1.7中，使用无参数构造方法创建ArrayList对象时，默认底层数组长度是10。JDK1.8中，使用无参数构造方法创建ArrayList对象时，默认底层数组长度是0；第一次添加元素，容量不足就要进行扩容了。

|  |
| --- |
| **public** ArrayList() {  **this**.**elementData** = ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***; }  **private static int** calculateCapacity(Object[] elementData, **int** minCapacity) {  **if** (elementData == ***DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA***) {  **return** Math.*max*(***DEFAULT\_CAPACITY***, minCapacity);  }  **return** minCapacity; } |

* 容量不足时进行扩容，默认扩容50%。如果扩容50%还不足容纳新增元素，就扩容为能容纳新增元素的最小数量。

|  |
| --- |
| **private void** grow(**int** minCapacity) {  **int** oldCapacity = **elementData**.**length**;  **int** newCapacity = oldCapacity + (oldCapacity >> 1);  **if** (newCapacity - minCapacity < 0)  newCapacity = minCapacity;  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity);**elementData** = Arrays.*copyOf*(**elementData**, newCapacity); } |



### 2.3.LinkedList的使用

#### 【示例4】使用LinkedList存储和出来分数

|  |
| --- |
| **public class** TestLinkedList {  **public static void** main(String[] args) {  //1.创建一个ArrayList集合对象  //ArrayList<Integer> list = new ArrayList<Integer>();  //LinkedList<Integer> list = new LinkedList<Integer>();  //List<Integer> list = new ArrayList<Integer>();  List<Integer> list = **new** LinkedList<Integer>();  //2.对集合中的元素进行操作  //2.1 添加  list.add(80);//向末尾添加元素  list.add(80);  list.add(78);//自动装箱  list.add(**new** Integer(78));  //list.add("Java"); // list.addFirst(12); // list.addLast(12);  list.add(0,12);  list.add(12);  list.add(100);  list.add(56);  list.add(80);  list.add(2,99);//向指定的位置添加元素  //2.2 查询指定的元素  System.**out**.println(list.size());//元素的个数  System.**out**.println(list);  list.remove(1);  System.**out**.println(list);  System.**out**.println(list.isEmpty());  } } |

问题1：将ArrayList替换成LinkedList之后，不变的是什么？

* 运算结果没有变
* 执行的功能代码没有变

问题2：将ArrayList替换成LinkedList之后，变化的是什么？

* 底层的结构变了

ArrayList：数组 LinkedList：双向链表

* 具体的执行过程变化了 list.add(2,99)

ArrayList：大量的后移元素

LinkedList：不需要大量的移动元素，修改节点的指向即可

问题3：到底是使用ArrayList还是LinkedList

* 根据使用场合而定
* 大量的根据索引查询的操作，大量的遍历操作（按照索引0--n-1逐个查询一般），建议使用ArrayList
* 如果存在较多的添加、删除操作，建议使用LinkedList

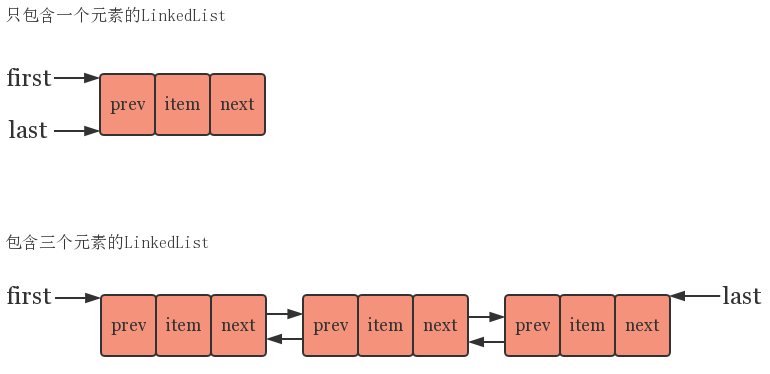
问题4：LinkedList增加了哪些方法

* 增加了对添加、删除、获取首尾元素的方法
* addFirst()、addLast()、removeFirst()、removeLast()、getFirst()、getLast()、

### 2.4理解LinkedList的底层源码

* 底层结构是一个双向链表。

|  |
| --- |
| **public class** LinkedList<E>  **extends** AbstractSequentialList<E>  **implements** List<E>, Deque<E>, Cloneable, java.io.Serializable{  **transient int size** = 0;//节点的数量  **transient** Node<E> **first**; //指向第一个节点  **transient** Node<E> **last**; //指向最后一个节点 **public** LinkedList() { }  } |

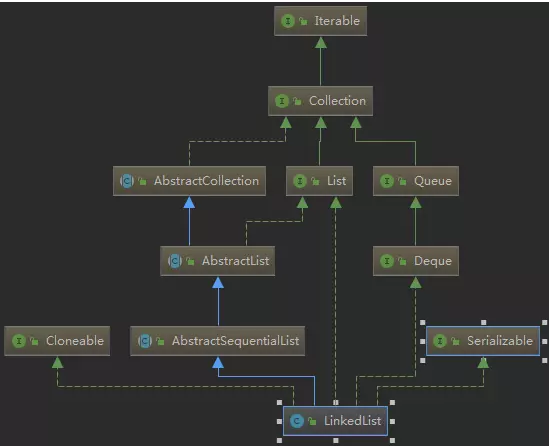


* 有一个静态内部类Node，表示双向链表的节点。

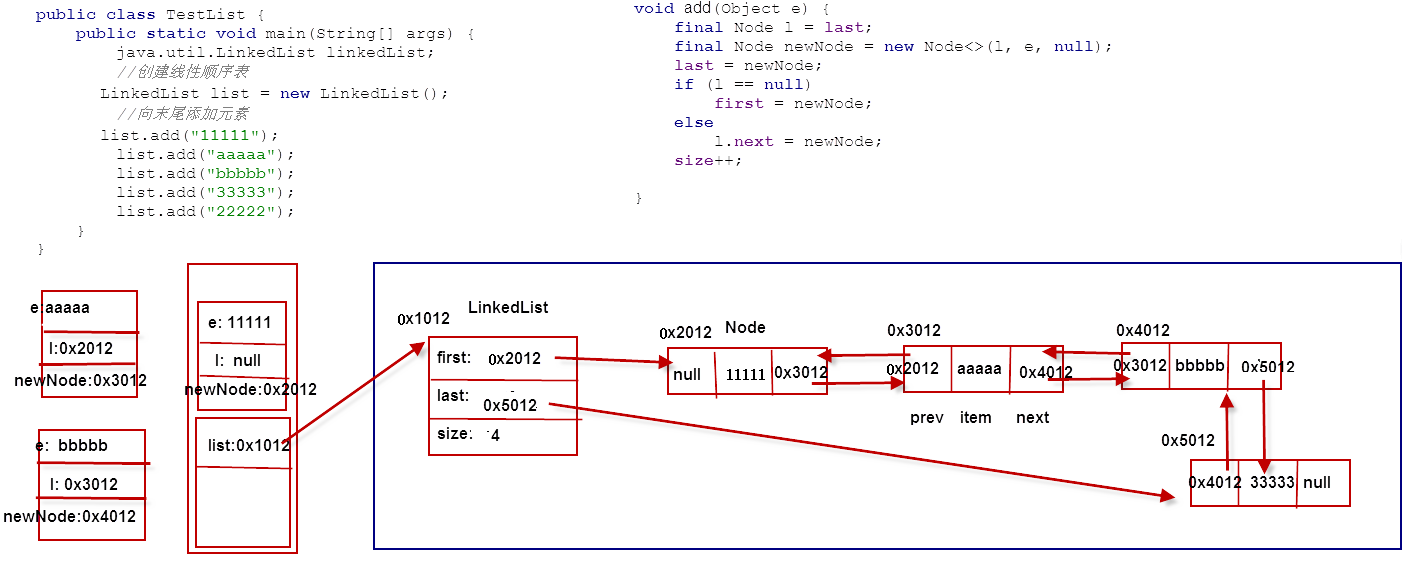
|  |
| --- |
| **private static class** Node<E> {  E **item**;//存储节点的数据  Node<E> **next**;//指向后一个节点  Node<E> **prev**; //指向前一个节点  Node(Node<E> prev, E element, Node<E> next) {  **this**.**item** = element;  **this**.**next** = next;  **this**.**prev** = prev;  } } |



* LinkedList实现了Deque接口，所以除了可以作为线性表来使用外，还可以当做队列和栈来使用



* 理解LinkedList添加元素add(elem)的执行过程



### 2.5 Java中栈和队列的实现类

* public class Stack<E> extends Vector<E> Vector过时了，被ArrayList替代了，Stack也就过时了
* public interface Queue<E> extends Collection<E>  
   public interface Deque<E> extends Queue<E>
* Deque和Queue的实现类  
   1.ArrayDeque 顺序栈 数组  
   2.LinkedList 链栈 链表

#### 【示例5】理解Java中栈和队列的接口和实现类

|  |
| --- |
| **早期的栈结构实现类 Stack**  **public class** Test1 {  **public static void** main(String[] args) {  Stack<String> stack =**new** Stack<String>();  *// 入栈方法* stack.push(**"马云"**);  stack.push(**"马化腾"**);  stack.push(**"马明哲"**);  stack.push(**"马老师"**);  *// 获得栈顶元素* String peek = stack.peek();  System.***out***.println(peek);  System.***out***.println(stack);  *// 跳栈 弹栈 取出栈顶元素* String pop = stack.pop();  System.***out***.println(pop);  System.***out***.println(stack);  } }  **LinkedList实现队列结构**  **public class** Test2 {  **public static void** main(String[] args) {  Queue<String> q=**new** LinkedList<String>();  *// 入队方法* q.offer(**"张三丰"**);  q.offer(**"张翠山"**);  q.offer(**"张无忌"**);   System.***out***.println(q);   *//获得队首元素* String peek = q.peek();  System.***out***.println(peek);  System.***out***.println(q);   *// 出队方法 取出队首* String poll = q.poll();  System.***out***.println(poll);  System.***out***.println(q);   } }  /\*\*  \* push：入栈 pop：出栈 peek：获取栈顶元素  \*/ **public class** TestLinkedList2 {  **public static void** main(String[] args) {   //摞盘子  Deque<String> deque1 = **new** LinkedList<String>();  deque1.push(**"盘子1"**);  deque1.push(**"盘子2"**);  deque1.push(**"盘子3"**);   System.**out**.println(deque1.size());  System.**out**.println(deque1.peek());//get 获取栈顶元素，不移除  System.**out**.println(deque1.peek());//get 获取栈顶元素，不移除  **while**(!deque1.isEmpty()){  String elem = deque1.pop();  System.**out**.println(elem);  }  System.**out**.println(deque1.size());  } } |

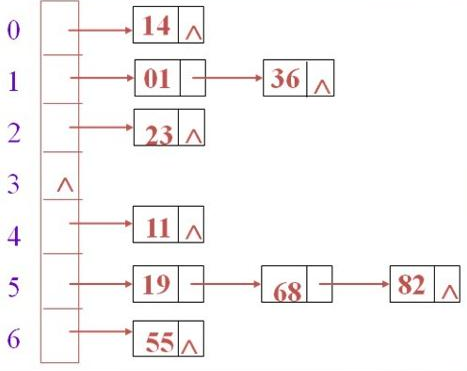
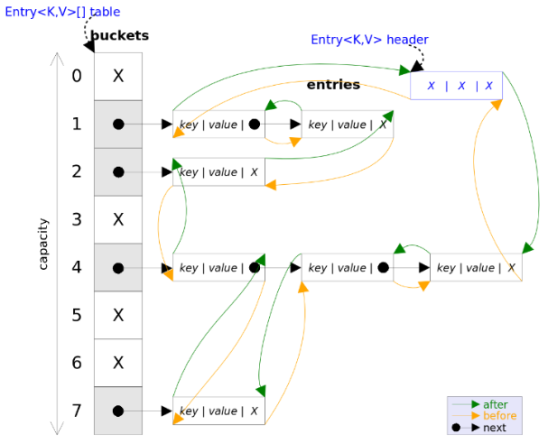
### 本节作业

1. 理解并练习ArrayList常用方法。
2. 根据API和网上资料，掌握retainAll ()的作用和使用
3. 理解ArrayList底层源码。
4. 理解LinkedList底层源码
5. Java中栈和队列的实现类分别是哪些类，实现了哪些接口

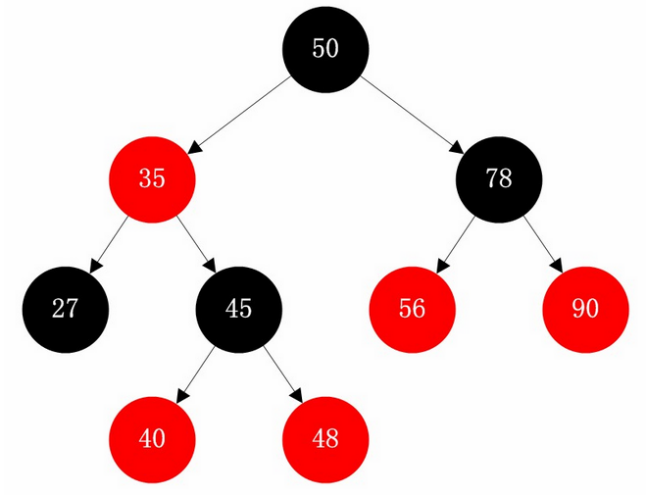
## 三、Set集合的使用

### 3.1 Set集合类型

* Set
  + 特点：无序 唯一（不重复）
* HashSet
  + 采用Hashtable哈希表存储结构（神奇的结构）
  + 优点：添加速度快 查询速度快 删除速度快
  + 缺点：无序
* LinkedHashSet
  + 采用哈希表存储结构，同时使用链表维护次序
  + 有序（添加顺序）

* TreeSet
  + 采用二叉树（红黑树）的存储结构
  + 优点：有序 查询速度比List快（按照内容查询）
  + 缺点：查询速度没有HashSet快



### 3.2使用各种Set集合类存储课程名称

#### 【示例6】使用各种Set集合存储课程名称

|  |
| --- |
| **public class** TestSet1 {  **public static void** main(String[] args) {  //创建一个集合set对象  //Set<String> set = new HashSet<String>();  //Set<String> set = new LinkedHashSet<String>();  Set<String> set = **new** TreeSet<String>();  //添加多个课程  set.add(**new** String(**"Java"**));  set.add(**"Oracle"**);  set.add(**"HTML"**);  set.add(**new** String(**"Java"**));  //输出课程  System.**out**.println(set.size());  System.**out**.println(set);  //不可以使用for循环遍历set  **for**(**int** i=0;i<set.size();i++){  //set.get(i);  }  //支持增强的for循环  //支持Iterator  Iterator<String> it = set.iterator();  **while**(it.hasNext()){  //  System.**out**.println(it.next());  }   } } |

总结

* HashSet  哈希表  唯一  无序
* LinkedHashSet  哈希表+链表  唯一 有序（添加顺序）
* TreeSet  红黑树 一种二叉平衡树 唯一  有序（自然顺序）
* List针对Collection增加了一些关于索引位置操作的方法 get(i) add(i,elem),remove(i),set(i,elem)
* Set是无序的，不可能提供关于索引位置操作的方法，set针对Collection没有增加任何方法
* List的遍历方式：for循环、for-each循环、Iterator迭代器、流式编程forEach
* Set的遍历方式： for-each循环、Iterator迭代器、流式编程forEach

### 3.3使用各种Set存储自定义学生信息

#### 【示例7】使用各种Set存储多个自定义学生信息

|  |
| --- |
| **public class** TestSet2 {  **public static void** main(String[] args) {  *//创建一个集合set对象  //Set<Student> set = new TreeSet<Student>();  //Set<Student> set = new HashSet<Student>();* Set<Student> set = **new** LinkedHashSet<Student>();  *//添加多个学生* Student stu2 = **new** Student(2, **"lisi"**, 23, 98);  Student stu3 = **new** Student(3, **"wangwu"**, 22, 87);  Student stu1 = **new** Student(1, **"zhangsan"**, 23, 90);  Student stu4 = **new** Student(1, **"zhangsan"**, 23, 90);  set.add(stu1);  set.add(stu2);  set.add(stu3);  set.add(stu4);   *//输出学生* System.***out***.println(set.size());  System.***out***.println(set);  } } |

问题1：HashSet、LinkedHashSet :为什么String有重复，会保持唯一；为什么Student有重复，不会保持唯一。

问题2：TreeSet 为什么String可以添加，而Student就不让添加到TreeSet中呢？ 而是抛出异常：

java.lang.ClassCastException: com.bjsxt.entity.Student cannot be cast to java.lang.Comparable

思考：String是系统类，Student是自定义类，应该是String已经做了某些事情，但是Student没有做

解答1：HashSet、LinkedHashSet 需要Student实现hashCode()和equals()

解答2：TreeSet 需要Student实现Comparable接口并指定比较的规则

#### 【示例8】让各种Set可以可以存储自定义类型的对象Student

|  |
| --- |
| **public class** Student **implements** Comparable<Student>{  **private int sno**;  **private** String **name**;  **private int age**;  **private double score**;   @Override  **public int** compareTo(Student o) {  *//return this.sno - o.sno;  //return o.sno - this.sno;* **return** -(**this**.**sno** - o.**sno**);  }  @Override  **public boolean** equals(Object o) {  **if** (**this** == o) **return true**;  **if** (o == **null** || getClass() != o.getClass()) **return false**;  Student student = (Student) o;  **if** (**sno** != student.**sno**) **return false**;  **if** (**age** != student.**age**) **return false**;  **if** (Double.*compare*(student.**score**, **score**) != 0) **return false**;  **return name** != **null** ? **name**.equals(student.**name**) :  student.**name** == **null**;  }  @Override  **public int** hashCode() {  **int** result;  **long** temp;  result = **sno**;  result = 31 \* result + (**name** != **null** ? **name**.hashCode() : 0);  result = 31 \* result + **age**;  temp = Double.*doubleToLongBits*(**score**);  result = 31 \* result + (**int**) (temp ^ (temp >>> 32));  **return** result;  } } |

### 本节作业

1. HashSet、LinkedHashSet、TreeSet的联系和区别
2. 编码题：使用各种Set集合存储课程名称
3. 编码题：使用各种Set存储多个自定义学生信息

## 四、Set集合的原理

### 4.1外部比较器Comparator的作用和使用

问题：内部比较器Comparable只有一个，如果希望指定多种比较的规则，怎么办？

解决：可以定义多个外部比较器，定义额外的类实现Comparator接口

#### 【示例9】定义外部比较器，按照分数升序排列

|  |
| --- |
| **public class** StudentScoreComparator **implements** Comparator<Student> {  @Override  **public int** compare(Student stu1, Student stu2) {  **if**(stu1.getScore()> stu2.getScore()){  **return** 1;  }**else if**(stu1.getScore()<stu2.getScore()){  **return** -1;  }**else**{  **return** 0;  }   } } |

#### 【示例10】定义外部比较器：按照姓名逆序排序，如姓名相同，按学号逆序排列

|  |
| --- |
| **public class** StudentNameDescComparator **implements** Comparator<Student> {  @Override  **public int** compare(Student stu1, Student stu2) {  **int** n = stu1.getName().compareTo(stu2.getName());  **if**(n !=0){  **return** n;  }**else**{  **return** -(stu1.getSno()-stu2.getSno());  }  } } |

#### 【示例11】使用外部比较器实现TreeSet对学生的排序

|  |
| --- |
| **public class** TestSet3 {  **public static void** main(String[] args) {  //创建一个set集合对象  // Comparator comp = new StudentScoreComparator();  //Comparator comp = new StudentNameComparator();  Comparator comp = **new** Comparator<Student>() {  **public int** compare(Student stu1, Student stu2) {  **return** -(stu1.getSno()-stu2.getSno());  }  };  //没有指定比较器，使用内部比较器  //Set<Student> set = new TreeSet<Student>();  //指定了外部比较器，就使用外部比较器  Set<Student> set = **new** TreeSet<Student>(comp);   //使用set存储多个学生数据  Student stu1 = **new** Student(1,**"xiaohua"**,18,90);  Student stu4 = **new** Student(4,**"xiaoli"**,18,60);  Student stu5 = **new** Student(1,**"xiaohua"**,18,90);  Student stu2 = **new** Student(2,**"xiaoming"**,17,80);  Student stu3 = **new** Student(3,**"xiaozhang"**,14,90);  set.add(stu3);  set.add(stu1);  set.add(stu2);  set.add(stu4);  set.add(stu5);  //输出学生数据  System.**out**.println(set.size());//??  **for**(Student stu :set){  System.**out**.println(stu);  }  } } |

内部比较器只能定义一个，一般将使用频率最高的比较规则定义为内部比较器的规则；外部比较器可以定义多个；

注意1：对于外部比较器，如果使用次数较少，可以通过匿名内部类来实现。

注意2：需要比较的场合才需要实现内部比较器或者外部比较器，比如排序、比如TreeSet中数据的存储和查询，在HashSet、LinkedHashSet、ArrayList中存储元素，不需要实现内部比较器或者外部比较器。

|  |
| --- |
| **注意：JDK8中可以使用Lambda表达式替代匿名内部类**  Set<Student> set = **new** TreeSet<Student>(  (o1, o2)-> {**return** -o1.getName().compareTo(o2.getName()) ;} );  TreeSet set = **new** TreeSet<Student>((stu1,stu2)->{  **if**(stu1.getScore()>stu2.getScore()){ **return** -1;}  **else if**(stu1.getScore()<stu2.getScore()){ **return** 1;}  **else**{ **return** stu1.getSno()- stu2.getSno();}  }); |

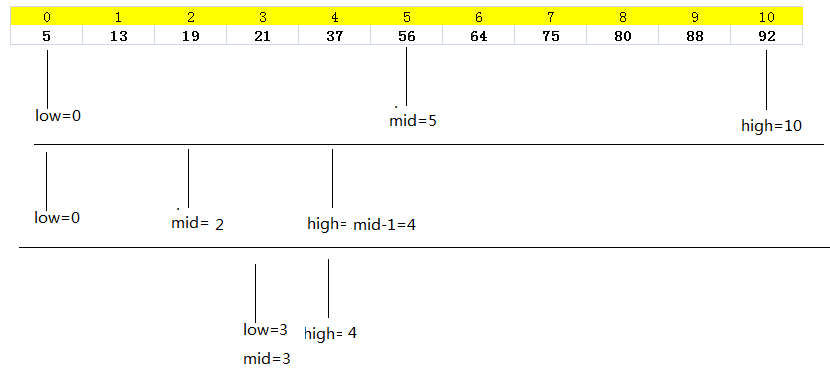
### 2.2哈希表的原理

#### 4.2.1引入哈希表

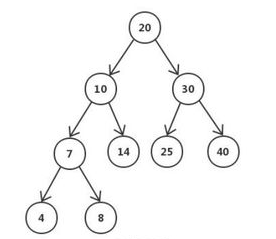
在无序数组中按照内容查找，效率低下，时间复杂度是O（n）



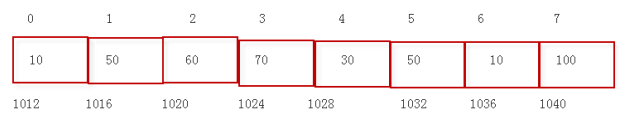
在有序数组中按照内容查找，可以使用折半查找，时间复杂度O（log2n）

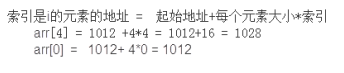


在二叉平衡树中按照内容查找，时间复杂度O（log2n）



在数组中按照索引查找，不进行比较和计数，直接计算得到，效率最高，时间复杂度O（1）





**问题：按照内容查找，能否也不进行比较，而是通过计算得到地址，实现类似数组按照索引查询的高效率呢O（1）**

**有！！！哈希表来实现**

前面查找方法共同特点：通过将关键字值与给定值比较，来确定位置。效率取决比较次数。

理想的方法是：不需要比较，根据给定值能直接定位记录的存储位置。

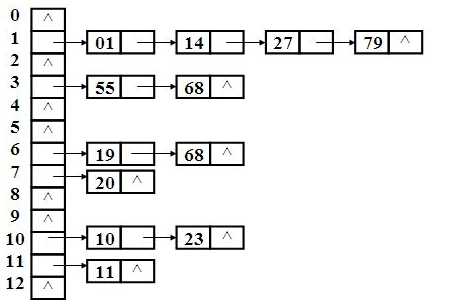
这样，需要在记录的存储位置与该记录的关键字之间建立一种确定的对应关系，使每个记录的关键字与一个存储位置相对应。

#### 4.2.2.哈希表的结构和特点

   hashtable 也叫散列表；特点：快   很快  神奇的快

    结构：结构有多种。最流行、最容易理解：顺序表+链表

   主结构：顺序表，每个顺序表的节点在单独引出一个链表



#### 4.2.3.哈希表是如何添加数据的

1. 计算哈希 码(调用hashCode(),结果是一个int值，整数的哈希码取自身即可)
2. 计算在哈希表中的存储位置  y=k(x)=x%11

x:哈希码  k(x) 函数y：在哈希表中的存储位置

1. 存入哈希表
   * 情况1：一次添加成功
   * 情况2：多次添加成功（出现了冲突，调用equals()和对应链表的元素进行比较，比较到最后，结果都是false，创建新节点，存储数据，并加入链表末尾）
   * 情况3：不添加（出现了冲突，调用equals()和对应链表的元素进行比较， 经过一次或者多次比较后，结果是true，表明重复，不添加）

    结论1：哈希表添加数据快（3步即可，不考虑冲突）

结论2：唯一、无序



#### 4.2.4.哈希表是如何查询数据的

   和添加数据的过程是相同的

* + 情况1：一次找到   23  86  76
  + 情况2：多次找到   67  56  78
  + 情况3：找不到   100 200

   结论1：哈希表查询数据快

#### 4.2.5.hashCode和equals到底有什么神奇的作用

* hashCode():计算哈希码，是一个整数，根据哈希码可以计算出数据在哈希表中的存储位置
* equals()：添加时出现了冲突，需要通过equals进行比较，判断是否相同；查询时也需要使用equals进行比较，判断是否相同

#### 4.2.6.各种类型数据的哈希码应该如何获取 hashCode()

* int   取自身 看Integer的源码
* double  3.14 3.15  3.145  6.567  9.87  取整不可以  看Double的源码
* String java  oracle  j+a+v+a  将各个字符的编码值相加不可以

        abc cba  bac  a:97  b:98  c:99

      abc 1\*97+2\*98+3\*99

 cba 1\*99+2\*98+3\*97

* Student 先各个属性的哈希码，进行某些相加相乘的运算

       int id        String name        int age        double score;

#### 4.2.7.如何减少冲突

   1）哈希表的长度和表中的记录数的比例--装填因子：

     如果Hash表的空间远远大于最后实际存储的记录个数，则造成了很大的空间浪费， 如果选取小了的话，则容易造成冲突。 在实际情况中，一般需要根据最终记录存储个数和关键字的分布特点来确定Hash表的大小。还有一种情况时可能事先不知道最终需要存储的记录个数，则需要动态维护Hash表的容量，此时可能需要重新计算Hash地址。

**装填因子=表中的记录数/哈希表的长度， 4/ 16  =0.25   8/ 16=0.5**

      如果装填因子越小，表明表中还有很多的空单元，则添加发生冲突的可能性越小；而装填因子越大，则发生冲突的可能性就越大，在查找时所耗费的时间就越多。 有相关文献证明当装填因子在0.5左右时候，Hash性能能够达到最优。

**因此，一般情况下，装填因子取经验值0.5**。

   2）**哈希函数的选择**

         直接定址法    平方取中法  折叠法   **除留取余法（y = x%11）**

   3）处理冲突的方法

          链地址法  开放地址法  再散列法   建立一个公共溢出区

### 本节作业

1. 定义多个外部比较器，实现TreeSet按照不同的规则存储学生。
2. 哈希表的原理（特点、结构、添加数据、查询数据）