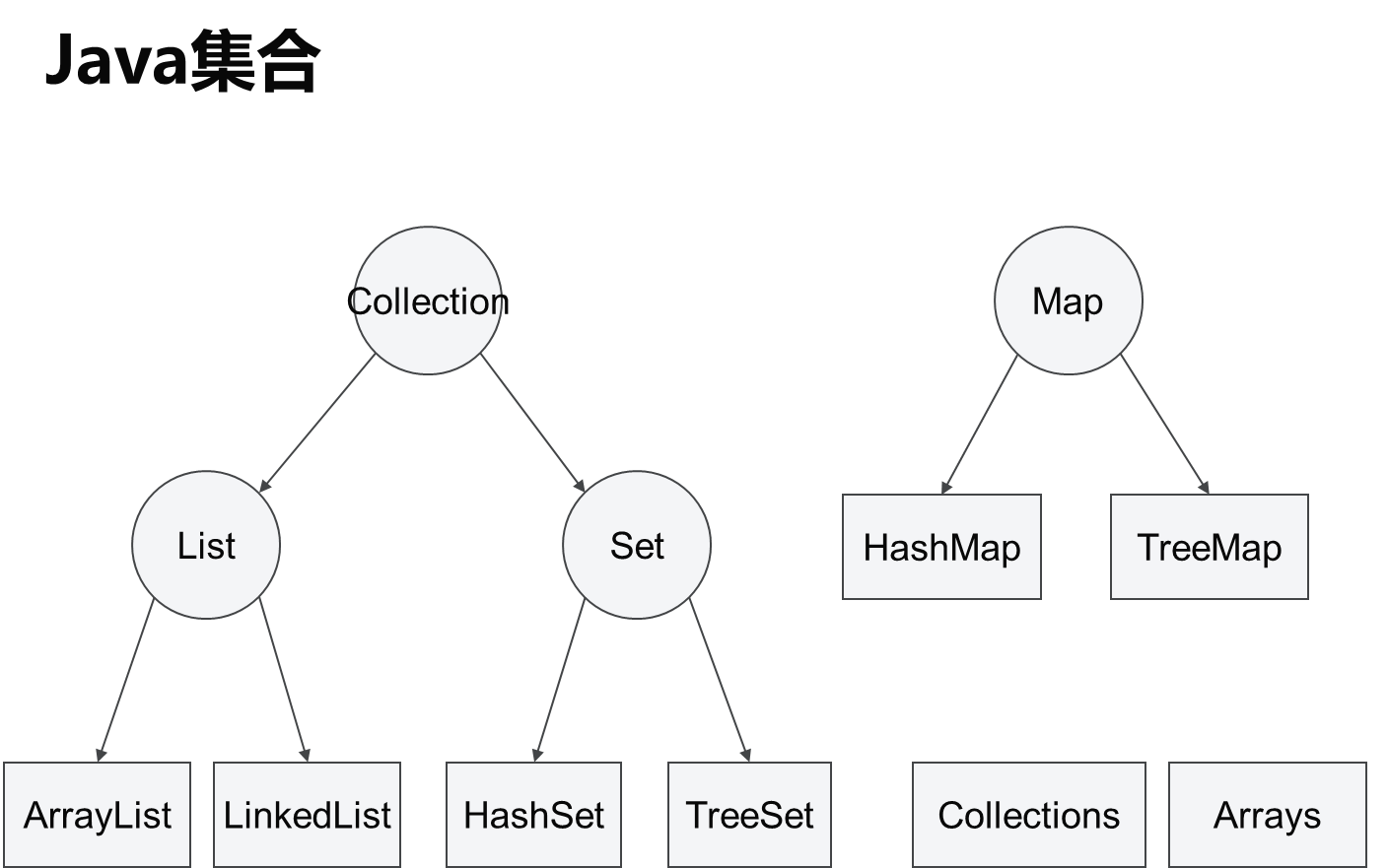
JAVA集合之简介

1、



2、为什么需要java集合？

1. 程序终究是要处理数据的,没有数据,程序的存在就没有了意义
2. 我们的代码很难处理众多杂乱无章的数据,所以需要把这些数据分类有效的组织起来
3. java集合包含多个集合类,这些集合类可以按照自己的特点按照不同的结构方式把相似的数据组织在一起,方便程序处理

JAVA集合之集合方法

1、

1. 一个集合就是一个容器,可以盛很多对象,这些对象称为集合的元素
2. 一个集合必须要有添加元素,删除元素的方法
3. 为了方便处理,还需要有依次取出每个元素的方法--遍历方法
4. 为了方便处理元素,还需要提供判断某个元素是否在集合中的方法
5. 此外,不同类型的集合有其特定的结构,还会有和此结构相关的一些方法

2、**public** **class** CollectionTest {

**public** **static** **void** main(String[] args) {

Collection collection = **new** ArrayList();// 创建一个集合类对象

// add为添加元素的方法

collection.add("谢宝发");

collection.add("周杰伦");

collection.add("乔布斯");

// remove删除集合元素

collection.remove("谢宝发");

// contains判断集合中是否含有某个元素

System.*out*.println("集合中是否包含乔布斯？" + collection.contains("乔布斯"));

System.*out*.println("集合中是否包含谢宝发？" + collection.contains("谢宝发"));

System.*out*.println(collection);// 输出集合中所有的元素

}

}

JAVA集合之迭代器Iterator

1、

1. boolean hasNext( ) //判断是否有下一个元素
2. Object next( ) //返回下一个元素
3. void remove( ) //删除指针指向的元素

2、**public** **class** IteratorTest {

**public** **static** **void** main(String[] args) {

Collection collection = **new** ArrayList();// ArrayList

collection.add("谢宝发");

collection.add("周杰伦");

collection.add("乔布斯");

// 创建迭代器（迭代器是集合中全权负责遍历元素的对象）

Iterator iterator = collection.iterator();// Iterator为集合中的迭代器，由集合的iterator()方法创建

/\*

\* // 遍历集合的每个元素 while (iterator.hasNext())// hasNext()判断当前位置是否有元素 {

\* Object element = iterator.next();// next()返回当前的对象并且使元素指针后移

\* System.out.println(element); } iterator.next();//

\* java.util.NoSuchElementException,不能贪心

\*/

// 在使用迭代器遍历元素的同时删除指定的元素

**while** (iterator.hasNext()) {

Object element = iterator.next();

**if** ("谢宝发".equals(element))// 迭代器的remove方法不能有参数，故只能另外进行判断

{

// iterator.remove();// 假如遍历到的对象与指定的相同则删除

collection.remove(element);// java.util.ConcurrentModificationException,在迭代器遍历时只能使用

// 迭代器的remove方法删除元素

}

}

System.*out*.println(collection);

}

Java集合之clear、isEmpty、size

1、**public** **class** CollectionTest2 {

**public** **static** **void** main(String[] args) {

Collection collection = **new** ArrayList();

collection.add("谢宝发");

collection.add("周杰伦");

collection.add("乔布斯");

// 集合的clear方法用于删除集合中所有的元素

// collection.clear();

// 集合的isEmpty方法判断元素个数是否为0

**boolean** isEmpty = collection.isEmpty();

System.*out*.println("集合是否为空？" + isEmpty);

// 集合的size方法得到集合的元素个数

**int** length = collection.size();

System.*out*.println(length);

System.*out*.println(collection);

}

}

Java集合之addall、removeall、containsall、retainall

1、

1. 调用者集合使用addall将另一个集合中的所以元素添加过来(求并集)
2. removeAll删除调用者集合中参数集合中也有的元素（求相对补集）
3. containsAll判断参数集合是否是调用者集合的非空子集
4. retainAll删除调用者集合中参数集合中没有的元素

2、**public** **class** CollectionTest3 {

**public** **static** **void** main(String[] args) {

Collection collection = **new** ArrayList();

collection.add("谢宝发");

collection.add("周杰伦");

collection.add("乔布斯");

Collection collection2 = **new** LinkedList();

collection2.add("谢宝发2");

collection2.add("周杰伦2");

collection2.add("乔布斯2");

// 调用者集合使用addall将另一个集合中的所以元素添加过来(求并集)

collection.addAll(collection2);

System.*out*.println(collection);

Collection collection3 = **new** HashSet();

collection3.add("谢宝发");

collection3.add("周杰伦3");

collection3.add("乔布斯3");

// removeAll删除调用者集合中参数集合中也有的元素（求相对补集）

collection.removeAll(collection3);

System.*out*.println(collection);

// containsAll判断参数集合是否是调用者集合的非空子集

**boolean** is1 = collection.containsAll(collection3);

System.*out*.println(is1);

**boolean** is2 = collection.containsAll(collection2);

System.*out*.println(is2);

// retainAll删除调用者集合中参数集合中没有的元素

collection.retainAll(collection2);

System.*out*.println(collection);

}

}

Java集合之存储元素的方式

1、当将一个对象添加到集合中时，并不是在集合里面重新创建一个对象，而是存储对象的地址。

2、**public** **class** CollectionTest4 {

**public** **static** **void** main(String[] args) {

Collection collection = **new** ArrayList();

Person p = **new** Person();

p.setAge(20);

p.setName("乔布斯");

collection.add(p);// 将对象的地址传给集合

System.*out*.println(collection);

}

}

**class** Person {

**private** String name;

**private** **int** age;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Person [age=" + age + ", name=" + name + "]";

}

}

Java集合之List（接口）

1、List

**public** **class** ListTest {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add("谢宝发");

list.add("周杰伦");

list.add("乔布斯");

Iterator iterator = list.iterator();// 迭代器由别人创建

**while** (iterator.hasNext()) { // List是集合的子类故也可以用迭代器遍历

System.*out*.println(iterator.next());

}

System.*out*.println(list);

}

}

Java集合之ArrayList

1、**访问**：ArrayList既可以通过迭代器访问元素，也可以通过get(下标)这个访问进行数组式的元素访问。

**public** **class** ArrayListTest {

**public** **static** **void** main(String[] args) {

ArrayList arrayList = **new** ArrayList();

arrayList.add("谢宝发");

arrayList.add("周杰伦");

arrayList.add("乔布斯");

// 使用迭代器遍历

Iterator it = arrayList.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());

}

// 使用ArrayList的get方法进行类似访问数组元素的方式对ArrayList的元素进行访问

**for** (**int** i = 0; i < arrayList.size(); i++) {

System.*out*.println(arrayList.get(i));

}

}

}

2、**添加和移除：**

1. ArrayList的add(下标，对象)方法可以对其数组的指定位置进行插队式添加元素，添加后其后的元素下标都要修改
2. ArrayList的remove(下标)方法可以对其数组的指定位置的元素进行移除，移除后其后的所以元素的下标都要做出相应的调整
3. 当元素的个数超出其长度时，ArrayList会先创建一块更大的新的连续的内存地址，将原来的元素放入，再添加新的元素

**public** **class** ArrayListTest2 {

**public** **static** **void** main(String[] args) {

ArrayList arrayList = **new** ArrayList();

arrayList.add("谢宝发");// 0

arrayList.add("周杰伦");// 1

arrayList.add("乔布斯");// 2

// ArrayList的add(下标，对象)方法可以对其数组的指定位置进行插队式添加元素，添加后其后的元素下标都要修改，效率低

arrayList.add(1, "王力宏");

**for** (**int** i = 0; i < arrayList.size(); i++) {

System.*out*.println(arrayList.get(i));

}

// ArrayList的remove(下标)方法可以对其数组的指定位置的元素进行移除，移除后其后的所以元素的下标

// 都要做出相应的调整

arrayList.remove(1);

Iterator it = arrayList.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());

}

}

}

3、**效率：**访问方法get效率最高，添加方法add和移除方法remove效率比较低。

Java集合之LinkedList

1、LinkedList内部结构为链表，链表的元素的内存空间不是连续的，通过指针来连接每个元素，get（下标）的下标是模拟出来的，故相对ArrayList来说它的get访问方法效率较低，因为它是通过一个个遍历来找到指定元素的，由于无论是数组结构还是链表结构，遍历的时间都远远小于修改元素下标的时间，所以在插队式添加和移除的时候效率就大大增加，不需要修改后面元素的下标，只要先遍历到指定位置然后修改指针即可。LinkedList不存在集合满的情况。

2、**public** **class** LinkedListTest {

**public** **static** **void** main(String[] args) {

LinkedList linkedList = **new** LinkedList();

linkedList.add("谢宝发");// 0，虚拟下标

linkedList.add("周杰伦");// 1，虚拟下标

linkedList.add("乔布斯");// 2，虚拟下标

// 插队式添加，功能同ArrayList，但是实现原理不同，因为存储结构（链表）不同

linkedList.add(1, "王力宏");

System.*out*.println(linkedList);

// 访问，功能同ArrayList，但是实现原理不同，下标是模拟出来的

**for** (**int** i = 0; i < linkedList.size(); i++) {

System.*out*.println(linkedList.get(i));

}

// 移除，功能同ArrayList，但是实现原理不同，因为存储结构（链表）不同

linkedList.remove(1);

System.*out*.println(linkedList);

// 独有的方法addFirst,addLast

linkedList.addFirst("库克");

linkedList.addLast("爱因斯坦");

System.*out*.println(linkedList);

}

}

3、自己实现LinkedList

//模拟LinkedList实现的类

**public** **class** ILinkedList {

**private** Element first;// 第一个元素，链首

**public** **void** add(Object obj) {

Element element = **new** Element();// 创建一个element类

element.setData(obj);

**if** (first == **null**)// 判断first是否存在

{

first = element;// 如果first不存在，就将element作为第一个元素

} **else** {// 如果first已经存在，就要找到最后一个元素，判断依据是其内部的next是否有指向

Element temp = first;

**while** (**true**) {

**if** (temp.next == **null**)// 判断first的next是否无指向

{

temp.next = element;// 如果first的next无指向则将element给next

} **else** {

temp = temp.next;// 假如temp的next不为空，则使temp指向自己内部的element对象（更新）

}

}

}

}

}

// 创建一个代表链表元素的类，其中data表示元素存储的对象，next表示元素所指向的下一个元素

**class** Element {

Object data;// 元素的地址

Element next;// 指向Element对象的Element

**public** **synchronized** Object getData() {

**return** data;

}

**public** **synchronized** **void** setData(Object data) {

**this**.data = data;

}

**public** **synchronized** Element getNext() {

**return** next;

}

**public** **synchronized** **void** setNext(Element next) {

**this**.next = next;

}

}

Java集合之List（II）

1、特性：

1. 元素有序:元素的添加顺序同于元素的访问顺序。
2. 元素可重复:同一个对象可多次添加作为不同的元素。元素可以为null，并且不会影响其他的元素。JDK1.5之后元素可以为基本类型，因为它自动进行装箱和拆箱。

2、**public** **class** ListTest2 {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add("谢宝发");

list.add("周杰伦");

list.add("乔布斯");

list.add("谢宝发");

list.add("谢宝发");

// set方法用于修改指定索引位置的值，并将原来的值返回

Object old = list.set(0, "比尔盖茨");

System.*out*.println(list);

System.*out*.println(old);

// indexOf方法返回指定元素第一次出现的位置

System.*out*.println(list.indexOf("谢宝发"));

// lastIndexOf方法返回指定元素最后一次出现的位置

System.*out*.println(list.lastIndexOf("谢宝发"));

// subList截取指定起始和结束位置的一段元素并将其返回

List sub = list.subList(1, 3);// 元素的下标在其前面

System.*out*.println(sub);

}

}

Java集合之ListIterator（专有）

1、

boolean hasNext( )

Object next( )

int nextIndex( ) //下一个元素的索引

boolean hasPrevious() //判断是否有前一个元素

Object previous( ) //先把指针移动到前一个元素,再取出来指针指向的当前元素

int previousIndex( )

void add(Object o) //添加一个元素

void remove( )

void set(Object o) //修改指针指向的当前元素

2、**public** **class** ListIteratorTest {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add("0");

list.add("1");

list.add("2");

list.add("3");

list.add("4");

// 创建listIterator

ListIterator listIterator = list.listIterator();

// 遍历，同Collection的Iterator

**while** (listIterator.hasNext()) {

System.*out*.println(listIterator.next());

}

// nextIndex,下一个元素的索引

System.*out*.println(listIterator.nextIndex());

// previousIndex,上一个元素的索引

System.*out*.println(listIterator.previousIndex());

// 逆遍历，独有

**while** (listIterator.hasPrevious()) {// hasPrevious判断是否有上一个元素

System.*out*.println(listIterator.previous());// previous先将指针向前移动，再读取

}

}

}

**Java集合之Vector**

1、

1. 内部数据结构 : 数组
2. Vector是线程安全的,在早期的jdk版本中就已经有了,那时候还没有java集合,它自己担负了ArrayList和LinkedList两个类的作用,但各个操作的性能很低
3. Vector内部的数组满时,新数组长度增长100%
4. 它有一些早期的方法,现在已经被废弃

2、**public** **class** VectorTest {

**public** **static** **void** main(String[] args) {

Vector vector = **new** Vector();

vector.add("谢宝发");

vector.add("乔布斯");

vector.add("周杰伦");

// Vector的遍历

Enumeration enums = vector.elements(); // Enumeration类似迭代器

**while** (enums.hasMoreElements()) {

System.*out*.println(enums.nextElement());

}

}

**Java集合之List练习**

1、去除集合中的重复元素

**public** **class** Exercise1 {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add("谢宝发");

list.add("周杰伦");

list.add("乔布斯");

list.add("谢宝发");

list.add("周杰伦2");

list.add("乔布斯");

list.add("谢宝发");

list.add("周杰伦");

list.add("乔布斯3");

List list2 = **new** ArrayList();

Iterator it = list.iterator();

// 遍历list

**while** (it.hasNext()) {

Object e = it.next();

**if** (list2.contains(e)) {// 判断list2中是否有元素e

// 如果有，则什么也不做

} **else** {

list2.add(e);// 没有则添加到list2中

}

}

System.*out*.println(list2);

}

}

2、一个集合中存储有各种类型的数据,现要求:把所有字符串类型的元素串联成一个字符串,对所有数值类型的元素求和,对所有Person类型的元素的年龄求和已经统计人数,如果是其他类型的元素,就输出到控制台

**public** **class** Exercise2 {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add("谢宝发");

list.add(9);

list.add(**new** Person("许嵩", 28));

list.add("周杰伦");

list.add(3.6);

list.add(**new** Object());

list.add(**new** Person("后弦", 34));

list.add("乔布斯");

list.add(5.6);

list.add(**new** Person("王力宏", 35));

list.add(**new** Object());

Iterator it = list.iterator();

StringBuilder sb = **new** StringBuilder();// StringBuilder拼接字符串效率更高

**double** allNumber = 0;

**int** allAge = 0;

**int** personCount = 0;

**while** (it.hasNext()) {

Object obj = it.next();

**if** (obj **instanceof** String)// 判断是否是String类

{

sb.append(obj);// 拼接

} **else** **if** (obj **instanceof** Number)// 判断是否是数值类型，Number为所有数值类型的父类

{

String temp = obj.toString();

**double** number = Double.*parseDouble*(temp);

allNumber += number;

} **else** **if** (obj **instanceof** Person) {

Person p = (Person) obj;// 虽然obj此时是Person类型，但是它名义上是obj，故不能使用Person的方法

allAge += p.getAge();

personCount++;

} **else** {

System.*out*.println(obj);// 如果是其他类型则输出到控制台

}

}

System.*out*.println("总字符串为：" + sb.toString());

System.*out*.println("数值总和为：" + allNumber);

System.*out*.println("总年龄为：" + allAge + "，总人数为" + personCount);

}

}

**Java集合之堆栈Stack**

1、堆栈：先进后出

2、//模拟堆栈数据结构

**public** **class** MyStack2 {

**private** LinkedList linkedList = **new** LinkedList();

// 模拟元素进栈的方法

**public** **void** addElement(Object obj) {

linkedList.add(obj);

}

// 模拟addFirst

**public** **void** addFirstElement(Object obj) {

linkedList.addFirst(obj);

}

// 模拟addLast

**public** **void** addLastElement(Object obj) {

linkedList.addLast(obj);

}

// 模拟元素出栈的方法

**public** Object removeElement() {

**return** linkedList.remove(linkedList.size() - 1);

}

// 堆栈的元素个数

**public** **int** stackSize() {

**return** linkedList.size();

}

**public** **static** **void** main(String[] args) {

MyStack myStack = **new** MyStack();

myStack.addElement("谢宝发");

myStack.addElement("周杰伦");

myStack.addElement("乔布斯");

System.*out*.println(myStack.removeElement());

System.*out*.println(myStack.removeElement());

System.*out*.println(myStack.removeElement());

}

}

**Java集合之队列Queue**

1、先进先出

2、//模拟队列的数据结构

**public** **class** MyQueue {

// 推荐使用LinkedList模拟

LinkedList linkedList = **new** LinkedList();

**public** **void** addElement(Object obj) {

linkedList.add(obj);

}

**public** Object removeElement() {

**return** linkedList.removeFirst();// 获取并移除队首的元素

}

// 获取队列的元素个数

**public** **int** queueSize() {

**return** linkedList.size();

}

**public** **static** **void** main(String[] args) {

MyQueue myQueue = **new** MyQueue();

myQueue.addElement("周杰伦");

myQueue.addElement("谢宝发");

myQueue.addElement("乔布斯");

System.*out*.println(myQueue.removeElement());

System.*out*.println(myQueue.removeElement());

System.*out*.println(myQueue.removeElement());

}

}

**Java集合之队列HashSet**

1、内部数据结构 : 哈希表 (散列表)

1. 哈希表的一种典型结构是数组和链表的结合体
2. 添加 : 元素所在的数组索引位置由元素对象的hashCode()方法决定 , 元素添加到链表中时,会依次和链表的每个元素判断是否相等(equals方法),如果有相等的情况,丢弃此元素,否则,添加到链表的末尾
3. 删除 : remove(obj) , 会先根据要删除的对象的hashCode方法定位到此对象所在数组的索引位置,然后依次比较次位置上链表的每一个元素,如果发现相等,则把链表中的此元素删除

2、//set的添加顺序与遍历顺序不一样

**public** **class** SetTest {

**public** **static** **void** main(String[] args) {

Set set = **new** HashSet();

set.add("1");

set.add("2");

set.add("3");

set.add("4");

set.add("5");

Iterator iterator = set.iterator();

**while** (iterator.hasNext()) {

System.*out*.println(iterator.next());

}

// Object类都有hashcode方法得到hashcode值，每个类对象都有一个hashcode

// Object类的hashcode方法默认返回该对象的内存地址

// hashcode一般无规律，但是自己定义的hashcode方法除外，故hashcode值不与对象一一对应，不同的对象

// 可能有相同的hashcode值

Object obj = **new** Object();

System.*out*.println(obj.hashCode());

Object obj2 = **new** Object();

System.*out*.println(obj2.hashCode());

}

}

3、**public** **class** HashSetTest {

**public** **static** **void** main(String[] args) {

HashSet hashset = **new** HashSet();

// 可以添加null对象，但是不能添加重复的元素

// 添加元素和删除元素效率都比较高，没有根据索引获取元素的方法，因其不支持索引

hashset.add("1");

hashset.add("2");

hashset.add("3");

hashset.add("4");

hashset.add("5");

hashset.add("5");

hashset.add("5");

hashset.add(**null**);

Iterator iterator = hashset.iterator();

**while** (iterator.hasNext()) {

System.*out*.println(iterator.next());

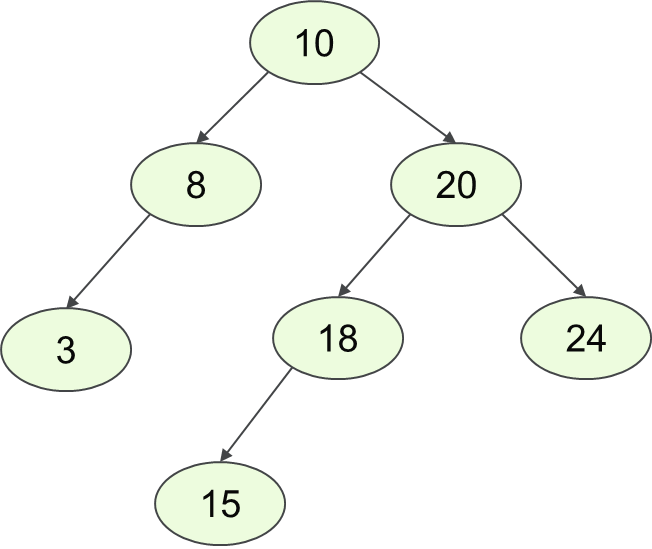
}

}

}

**Java集合之二叉树TreeSet**

1、内部数据结构 : 二叉树 ，特点：一个节点最多只有两个子节点 , 左 < 父 < 右 , 没有重复元素



2、

1. 元素的添加顺序与遍历顺序不同，遍历前元素已经按照从小到大的顺序排好
2. 不能添加重复的元素。由于二叉树的特点，遍历和删除的效率很高。

3) 不能添加null，原因是一旦有了null就会出现无法比较大小的情况，那么二叉树的功能就无法实现。

3、**public** **class** TreeSetTest {

**public** **static** **void** main(String[] args) {

TreeSet treeSet = **new** TreeSet();

// 元素的添加顺序与遍历顺序不同，遍历前已经按照从小到大的顺序排好。

treeSet.add(5);

treeSet.add(1);

treeSet.add(4);

treeSet.add(3);

treeSet.add(6);

treeSet.add(2);

treeSet.add(5);// 元素不能重复

treeSet.add(1);

treeSet.add(4);

treeSet.add(3);

treeSet.add(6);

treeSet.add(2);

treeSet.add(**null**);// 不能添加null，因为这破坏了二叉树的结构

Iterator iterator = treeSet.iterator();// TreeSet只能通过迭代器遍历

**while** (iterator.hasNext()) {

System.*out*.println(iterator.next());

}

}

}

**Java集合之Comparable**

1、

1. Comparable 接口用来表明元素自身的排序规则,也就是说元素只有实现此接口才具有比较性,才可以添加到TreeSet中
2. java常见类如String , Integer等都已实现Comparable 接口,都有自己独特的排序规则,但我们自己定义的类Person由于没有实现Comparable接口,因此不具备排序性(比较性)
3. 如果我们想按照Person的姓名排序,我们可以实现Comparable 接口,实现此接口的compareTo方法,在此方法中写按照姓名排序的代码
4. 元素自身所具有的排序规则称为自然排序

2、

1. TreeSet添加的元素需要实现Comparable的compareTo方法，如果没有实现则只能添加一个该对象。
2. compareTo由新添加的元素调用，参数为已经存在的元素，如果新比旧大则返回1，相等返回0，小则返回-1.

**Java集合之Comparator**

1、当我们需要按照自己的方式排序时，由于没法修改源码，所以必须得通过比较器comparator来实现比较。

2、**public** **class** ComparatorTest {

**public** **static** **void** main(String[] args) {

List list = **new** ArrayList();

list.add(3);

list.add(6);

list.add(4);

list.add(2);

list.add(1);

list.add(5);

// list并没有实现Comparable，故没法自己实现排序

Collections.*sort*(list, **new** Comparator() {

// 第一个参数为集合类，第二个为比较器类，必须实现它的compare方法

@Override

// 必须实现它的compare方法

**public** **int** compare(Object o1, Object o2) {

Integer i1 = (Integer) o1;

Integer i2 = (Integer) o2;

**return** i1 - i2;

}

});

Iterator it = list.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());// 实现升序排列

}

}

}

3、**public** **class** ComparatorTest2 {

**public** **static** **void** main(String[] args) {

MyComparator m = **new** MyComparator();// 创建一个自己定义的比较器对象

TreeSet treeSet = **new** TreeSet(m);// 创建TreeSet时将比较器传入取代其自身的比较器

treeSet.add(**new** Person("a", 16));

treeSet.add(**new** Person("ab", 16));

treeSet.add(**new** Person("b", 17));

treeSet.add(**new** Person("c", 18));

// 由于需要复合排序，故用比较器也不能一步完成，需要自己定义

Iterator it = treeSet.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());

}

}

}

**class** MyComparator **implements** Comparator {// 自己定义的比较器

@Override

**public** **int** compare(Object o1, Object o2) {

// **TODO** Auto-generated method stub

Person p1 = (Person) o1;

Person p2 = (Person) o2;

String name1 = p1.getName();

String name2 = p2.getName();

**int** age1 = p1.getAge();

**int** age2 = p2.getAge();

**if** (age1 == age2) {// 优先对年龄进行排序，如果年龄相同，再对其姓名进行排序

**return** name1.compareTo(name2);

} **else** {

**return** age2 - age1;

}

}

}

**Java集合之Set总结**

1、Set接口和数学上集合的概念相同 , 即元素无序 , 元素不可重复

2、Set接口是Collection接口的子接口,方法和Collection一致 ,并没有提供自己特有的方法

**Java集合之LinkedHashSet**

1、LinkedHashSet 的内部结构是哈希表和链表的结合体

哈希表的特点 --元素无序 , 元素不可重复 , 添加删除效率高

链表的特点 -- 元素有序 , 元素可重复

结合后的特点 -- 元素有序,元素不可重复 , 添加删除效率高

2、

//LinkedHashSetTest 集合了二者的特点，它是元素有序且元素唯一的。

**public** **class** LinkedHashSetTest {

**public** **static** **void** main(String[] args) {

LinkedHashSet linkedHashSet = **new** LinkedHashSet();

linkedHashSet.add("谢宝发");

linkedHashSet.add("周杰伦");

linkedHashSet.add("乔布斯");

linkedHashSet.add("谢宝发");// 元素不可重复

linkedHashSet.add("周杰伦");// 元素不可重复

linkedHashSet.add("乔布斯");// 元素不可重复

Iterator it = linkedHashSet.iterator();

**while** (it.hasNext()) {

System.*out*.println(it.next());// 添加顺序与遍历顺序相同

}

}

}

**Java集合之泛型**

1、当使用泛型明确了List里面的元素的类型时，如果添加其他类型的元素就会报错，而且是在编译时就报错，这样就避免了运行时的错误，便于修改，也保证了程序的安全，虽然没有更简洁，但是好处不言而喻。

2、**public** **class** GeneriTest2 {

**public** **static** **void** main(String[] args) {

List<String> list = **new** ArrayList<String>();// 有了泛型

list.add("谢宝发");

// list.add(1);// 编译时报错

}

// 当使用泛型明确了List里面的元素的类型时，如果添加其他类型的元素就会报错，而且是在编译时就报错，这样就避免了

// 运行时的错误，便于修改，也保证了程序的安全，虽然没有更简洁，但是好处不言而喻。

**public** **static** **void** process(List list) {

Iterator it = list.iterator();

**while** (it.hasNext()) {

Object obj = it.next();

**if** (obj != **null**) {

String s = (String) obj;// 默认元素为String类

System.*out*.println(s.length());

}

}

}

}

**Java集合之泛型深入**

1、

1. 泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说一个类所操作的另外一个类被指定为一个参数,就像方法的参数一样,在使用时传递具体的类型值,那么这时就明确了这个类所操作的另外一个类的类型了. 通常使用 <T> 表示泛型
2. 泛型的语法: 在声明类时一同声明泛型<T>, 表示一个参数化类型, 在声明类变量和实例化类对象时,需要给泛型参数传递一个具体的类型值,比如 class Demo<T>{ } Demo<String> demo = new Demo<String>( );
3. 这种参数类型可以声明在类、接口和方法的定义中，分别称为泛型类、泛型接口、泛型方法 。也就是说定义了泛型的类称为泛型类
4. 使用泛型的步骤 : 1 定义泛型类 2 使用泛型类
5. 好处：安全 把运行时发生的异常提前到编译时期
6. 好处：简单 不需要进行显示类型转换
7. 向后兼容List list = new ArrayList<String>(); 是正确的
8. 泛型不协变 List<Object> list = new ArrayList<String>();是不正确的

2、**public** **class** Test1 {

**public** **static** **void** main(String[] args) {

Demo<String> demo;// 泛型的使用1：声明类变量时

demo = **new** Demo<String>();// 泛型的使用2：创建类对象时

Demo<String> demo2 = **new** Demo<String>();// 泛型的使用就相当于预编译中的展开，它对本来模糊的类型进行解释

demo2.show("谢宝发");

Demo<Integer> demo3 = **new** Demo<Integer>();

demo3.show(3);

// Demo<int> demo4;泛型只能为类不能为基本类型比如int或者double

}

}

**class** Demo<T> // 泛型的声明：当我们还不确定这个类要操作什么类型的类时可以先声明一个模糊的类型，它相当于

// 方法中的参数

{

**private** String s;

**public** **void** show(T t) {// 可以把泛型当做真实的类使用

System.*out*.println(t);

}

}

3、**集合和泛型是绝配**

4、

**public** **class** Test2 {

**public** **static** **void** main(String[] args) {

Demo3 d = **new** Demo3();

d.show("谢宝发");// 泛型方法不一定要解释泛型，也就是说可以省略

d.<String> show("周杰伦");// 正规的泛型使用

List list = **new** ArrayList<String>();// 假如左边没有泛型右边有，则list不能体现泛型的功能

list.add("fdvidbv");

list.add(1);// 不报错说明编译器不认为list声明了泛型，也就是说只有在声明时定义泛型的类才算泛型类

List<String> list2 = **new** ArrayList();

list2.add("xbf");

// list2.add(1);报错，只要声明时定义泛型就是泛型类

// 泛型的不协变：泛型不支持继承，可以不两端一起声明泛型，但是如果一起声明了，两端的类型一定要完全相同

// List<Object> list3 = new ArrayList<String>();

}

}

**class** Demo3 {// 这个类没有定义泛型，不是泛型类

**public** <T> **void** show(T t) // 泛型方法，其中<T>必须放在返回值类型前，也就是说返回值和函数名必须连着

{

System.*out*.println(t);

}

}

**Java集合之有限制泛型**

1、

类型通配符 <?> , 这个 ? 相当于Object ,Integer ,String 等,可以用来给<T> 赋值,表示T的类型值是任意的,虽然如此,但<?>不单独使用

List<?> list = new ArrayList<String>();

List<?> list2 = new ArrayList<Integer>();

2、有限制泛型: 在声明泛型的时候,给这个泛型可以接收的类型值一个约束,比如 class Demo<T extends Number>{ }表示Demo类只能操作数值类型的对象,如Integer,Double

<T extends Collection> extends指定Collection是上限

<? super Inteter> super 指定Integer是下限

<? extends Number> 类型通配符和有限制泛型合作,可以使变量接收范围更广泛的

**public** **class** Test3<T **extends** Number> {// 定义有限制泛型，表明操作类的上限为数值Number类型

**public** **void** show(T t) {

System.*out*.println(t);

}

**public** **static** **void** main(String[] args) {

Test3 t = **new** Test3();// 如果在声明时没有使用泛型则默认操作类的类型为上限的类型

t.show(1);

t.show(1.0);

Test3<Integer> t2 = **new** Test3<Integer>();// 在声明时使用泛型

t2.show(1);

// t2.show(1.0);// 假如使用了泛型那么操作类的类型必须复合要求

Test3 t3 = **new** Test3();

// t3.show("xbf");默认为Number类，则无法操作String类

}

}

3、**public** **class** Test3<T **extends** Number> {// 定义有限制泛型，表明操作类的上限为数值Number类型

**public** **void** show(T t) {

System.*out*.println(t);

}

**public** **static** **void** main(String[] args) {

Test3 t = **new** Test3();// 如果在声明时没有使用泛型则默认操作类的类型为上限的类型

t.show(1);

t.show(1.0);

Test3<Integer> t2 = **new** Test3<Integer>();// 在声明时使用泛型

t2.show(1);

// t2.show(1.0);// 假如使用了泛型那么操作类的类型必须复合要求

Test3 t3 = **new** Test3();

// t3.show("xbf");默认为Number类，则无法操作String类

Test3<?> t4 = **new** Test3();// 单独使用通配符时因无法确定类型故不能使用与泛型有关的方法

// t4.show(4);

Test3<? **extends** Number> t5 = **new** Test3();// 通配符与有限制泛型一起用

// t5.show(t);// 通配符与有限制泛型一起用还是无法确定类型故不能使用与泛型有关的方法

// 也就是说只要有通配符由于没有为泛型做出解释故无法使用与泛型有关的方法

// 统配符与泛型使用一般用于方法,而不是类，因为假如用于类中，则不能使用该类的一些方法。

// 而用于定义方法，对方法的最用对象做出限制

Collection<Number> collection = **new** ArrayList();

collection.add(1);

Collection<Integer> collection2 = **new** ArrayList();

collection2.add(2);

collection2.add(2);

collection.addAll(collection2);// addAll的源码中就有? extends

// e,意思是只能添加方法所在类的泛型e的子类作为元素

// 这里是addAll的类是collection，它的泛型是Number，故被它添加的集合的泛型只能以Number作为上限

}

}

**Java集合之Map**

1、Map集合的操作

添加

V put(K key, V value) //把一个键值对存入map集合中

删除

V remove(Object key) //删除key为指定对象的键值对

void clear() //情况map中的所有元素

判断

boolean containsKey(Object key) //判断是否包含key为指定对象的键值对

boolean containsValue(Object value) //判断是否包含value为指定对象的元素

boolean isEmpty() //判断map集合是否包含元素

获取

V get(Object key) //获得指定key对应的value

Set<K> keySet() //获得所有key组成的Set集合

Collection<V> values() //获得所有value组成的Set集合

Set<Map.Entry<K,V>> entrySet() //获得所有键值对对象组成的Set集合

int size() //获得map集合的大小(存了多少元素)

2、

**public** **class** MapTest1 {

**public** **static** **void** main(String[] args) {

Map map = **new** HashMap();// 创建无泛型的HashMap

map.put("谢宝发", 20);// 没有Collection中的add方法，而是使用put一次性添加键值对（key=value）

map.put("周杰伦", 36);

map.put("乔布斯", 55);

Object object = map.get("谢宝发");// get方法的参数为key，返回值为key所对应的value

System.*out*.println(object);

// 创建时声明泛型，这里的泛型有两个

Map<String, Integer> map2 = **new** HashMap<String, Integer>();

map2.put("谢宝发", 20);

map2.put("周杰伦", 36);

map2.put("乔布斯", 55);

// 源码中get方法没有声明泛型，但是返回值有泛型

Integer integer = map2.get("周杰伦");

System.*out*.println(integer);

}

}

3、**public** **class** MapTest2 {

**public** **static** **void** main(String[] args) {

Map map = **new** HashMap();

map.put("谢宝发", 20);

map.put("周杰伦", 36);

map.put("乔布斯", 55);

// map中键不能重复，值可以重复，当用put添加键相同的元素时，新的value会覆盖旧的，并且会返回旧的value

System.*out*.println(map.put("谢宝发", 21));

// remove会根据得到的key删除对应的键值对，并返回value

System.*out*.println(map.remove("周杰伦"));

// containsKey判断是否含有指定的key，并返回boolean值

System.*out*.println(map.containsKey("谢宝发"));

System.*out*.println(map.containsKey("周杰伦"));

// containsKey判断是否含有指定的value，并返回boolean值

System.*out*.println(map.containsValue(21));

System.*out*.println(map.containsValue(36));

// containsKey判断map是否为空，并返回boolean值

System.*out*.println(map.isEmpty());

// size返回map的元素个数

System.*out*.println(map.size());

}

}

**Java集合之HashMap**

1、数据结构 : 内部是哈希表 , 元素为Map.Entry类的对象

2、特点：

①把key 和 value 封装到了Entry对象中 , 然后对entry进行操作

②从Map的设计来看,是通过key 找到value,所以在一个Map中,key的值不能重复,但value可以重复

③因为内部结构为哈希表,所以添加删除查询等操作都很快

④由内部结构决定了HashMap不能直接遍历,需要使用Set集合的遍历方式进行遍历,所以就有了 keySet() , values() , entrySet()三个方法,其中keySet() , values() 依赖entrySet()

3、遍历HashMap的三种方式：

//Map没有直接遍历的方法，所以我们要想办法遍历

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Map map = **new** HashMap();

map.put("谢宝发", 20);

map.put("周杰伦", 36);

map.put("乔布斯", 55);

// 遍历方法1：keySet，得到存放key的一个set对象，再通过迭代器和map的get得到value

Set<String> keySet = map.keySet();// 得到存放key的一个set对象

Iterator<String> iterator = keySet.iterator();

**while** (iterator.hasNext()) {

String key = iterator.next();// 从keySet 中得到key

Object valueString = map.get(key);// 根据key得到value

System.*out*.println(key + "=" + valueString);

}

// 遍历方法2：values，得到存放value的一个collection对象，通过迭代器遍历，但是不知道其对应的key

Collection<Integer> collection = map.values();// 得到存放value的一个collection对象

Iterator<Integer> iterator2 = collection.iterator();

**while** (iterator2.hasNext()) {

System.*out*.println(iterator2.next());

}

// 遍历方法3：entrySet，得到存放键值对的entry对象的set对象，通过迭代器遍历entrySet得到Entry对象

// 再通过entry的get\*\*\*方法得到entry内部的key和value

Set<Map.Entry<String, Integer>> entrySet = map.entrySet();// 得到存放键值对的entry对象的set对象

Iterator<Map.Entry<String, Integer>> iterator3 = entrySet.iterator();

**while** (iterator3.hasNext()) {

Map.Entry<String, Integer> entry = iterator3.next();// 通过迭代器遍历entrySet得到Entry对象

String key = entry.getKey();// 得到entry内部的key

Integer value = entry.getValue();// 得到entry内部的value

System.*out*.println(key + "=" + value);

}

}

}

**Java集合之TreeMap**

1、数据结构 : 二叉树 , 元素是Map.Entry类的对象

2、特点：

①元素添加到map中时就已经按key排序了顺序

②添加 , 删除 , 获取 操作都很快

③由于按key进行排序,所以要求key具有排序性

3、

//测试TreeMap

**public** **class** Test1 {

**public** **static** **void** main(String[] args) {

TreeMap<String, Integer> treeMap = **new** TreeMap<String, Integer>();

// 添加方法put

treeMap.put("xbf", 20);

treeMap.put("xbfg", 20);

treeMap.put("jay", 36);

treeMap.put("jobs", 55);

// System.out.println(treeMap);

// 获取方法get

Integer ageInteger = treeMap.get("乔布斯");

// System.out.println(ageInteger);

// 删除方法remove

// treeMap.remove("谢宝发");

// System.out.println(treeMap);

// 自定义排序：按照年龄的降序排序，如果年龄相同则按照姓名的自然顺序排

TreeMap<String, Integer> treeMap2 = **new** TreeMap<String, Integer>(

**new** MyComparator(treeMap));// 创建一个新的treeMap对象，它使用自定义的排序器，而且数据来自treeMap

treeMap2.putAll(treeMap);

System.*out*.println(treeMap2);

}

**static** **class** MyComparator **implements** Comparator<String> {// 定义为静态类才能被静态方法访问

**private** Map<String, Integer> map;

**public** MyComparator(Map map) {// 在创建MyComparator对象时传入一个treemap用于获取

**super**();

**this**.map = map;

}

@Override

**public** **int** compare(String key1, String key2) {

Integer value1 = map.get(key1);// 获取value1

Integer value2 = map.get(key2);// 获取value2

**if** (value1 == **null** || value2 == **null**) {

**return** 1;

}

**if** (value1 == value2) {// 如果年龄相等就按照姓名的自然顺序排序

**return** key1.compareTo(key2);

} **else** {// 否则按照年龄的降序排序

**return** value2 - value1;

}

}

}

}

**Java集合之Properties**

1、HashTable是jdk老版本的遗留类,内部也是哈希表,因为是线程安全的,所以各种操作都相对较慢,已经被HashMap替代,现在几乎不再使用,但需要知道有这个类

2、HashTable还有一个子类Properties,具有集合的特点又可以进行文件的读取写入操作,使用频率很高

3、//测试Properties

**public** **class** PropertiesTest {

**public** **static** **void** main(String[] args) **throws** IOException {

String filePath = PropertiesTest.**class**.getResource("/a.properties")

.getFile();// 获取配置文件的路径

File file = **new** java.io.File(filePath);

FileReader fileReader = **new** FileReader(file);

Properties properties = **new** Properties();

properties.load(fileReader);// 加载流

String v1 = properties.getProperty("Jay");// 根据Key获取value

System.*out*.println(v1);

properties.setProperty("xbf", "谢宇轩");// 设置键值,但是不会改变配置文件

System.*out*.println(properties);

// 要改变配置文件需要创建输出流并使用store保存

FileWriter fileWriter = **new** FileWriter(file);

properties.store(fileWriter, "注释");// 保存改动到配置文件

System.*out*.println(properties);

}

}

**Java集合之Collections工具类**

1. sort对List集合进行排序
2. binarySearch 方法,使用二分查找法查找元素,要求必须是List集合,而且必须已经排好序
3. fill方法,使用指定元素填充List集合
4. replaceAll方法,使用指定元素替换所有的另一个元素,只对List有效
5. max方法,返回一个集合中最大的元素
6. min
7. shuffle 洗牌,打乱集合元素的顺序
8. unmodifiableList返回指定集合的不可修改的包装集合

9. synchronizedList返回指定集合的线程安全的包装集合

//测试Collections工具类

**public** **class** CollectionsTest1 {

**public** **static** **void** main(String[] args) {

// sort排序方法:对List集合排序,要求元素类型应当一致

List list = **new** ArrayList();

// list.add(111);// java.lang.Integer cannot be cast to java.lang.String

list.add("bbb");

list.add("bbb");

list.add("bbb");

list.add("ccc");

list.add("aaa");

// list.add(111);// java.lang.String cannot be cast to java.lang.Integer

// 首先是一个参数的，它要求传入的list对象本身要有比较性

Collections.*sort*(list);// Collections是静态类，可以直接用类名调用方法

System.*out*.println(list);

// binarySearch,使用二分查找法查找元素,传入key返回排好序的索引要求必须是List集合,而且必须已经排好序

**int** index = Collections.*binarySearch*(list, "ccc");

System.*out*.println(index);

// fill,使用指定元素填充List集合

// Collections.fill(list, "xbf");

System.*out*.println(list);

// replaceAll方法,使用指定元素替换所有的另一个元素,只对List有效

Collections.*replaceAll*(list, "bbb", "ppp");

System.*out*.println(list);

// max方法,返回一个集合中最大的元素

Object maxValue = Collections.*max*(list);

System.*out*.println(maxValue);

// min方法,返回一个集合中最小的元素

Object minValue = Collections.*min*(list);

System.*out*.println(minValue);

// shuffle 洗牌,打乱集合元素的顺序

Collections.*shuffle*(list);

System.*out*.println(list);

// unmodifiableList将指定集合变成的不可修改的包装集合并返回

List unmodifiableList = Collections.*unmodifiableList*(list);

unmodifiableList.add("Jobs");// java.lang.UnsupportedOperationException

}

}

**Java集合之Arrays工具类**

1、

sort 给数组的元素排序

binarySearch 二分查找法查找元素

copyOf 数组复制

equals 判断两个数组的元素是否相等

fill 填充数组

hashCode 计算数组的元素的哈希值

toString 返回每个数组元素组成的字符串

asList 把数组转化为结构不可改变,但可以查询的集合

2、//测试Arrays数组工具类，由于该类专门对数组进行操作，而不同的基本类型都有其对应的数组，

//所以Arrays的方法都是成片出现

//Arrays也是静态类，可以直接以类名调用方法

**public** **class** ArraysTest1 {

**public** **static** **void** main(String[] args) {

**int**[] intArr = **new** **int**[] { 7, 8, 5, 9, 2, 1, 1, 6, 4, 8 };

// sort,对数组进行排序

Arrays.*sort*(intArr);// 返回值为void，说明在数组内部进行操作

// for (int i = 0; i < intArr.length; i++) {

// System.out.println(intArr[i]);

// }

// toString,将数组元素拼接成字符串并返回这个字符串

String elements = Arrays.*toString*(intArr);

System.*out*.println(elements);

// binarySearch，二分查找法，要求数组已排好序，返回索引

**int** index = Arrays.*binarySearch*(intArr, 3);// 不存在的元素的索引为假设它在之后的索引+1的相反数

System.*out*.println(index);

// copyOf，复制一个数组，参数有两个，参数1为源数组，参数2为新数组长度

**int**[] intArr2 = Arrays.*copyOf*(intArr, 3);// 拷贝一部分

System.*out*.println(Arrays.*toString*(intArr2));

**int**[] intArr3 = Arrays.*copyOf*(intArr, intArr.length);// 拷贝整个数组

System.*out*.println(Arrays.*toString*(intArr3));

**int**[] intArr4 = Arrays.*copyOf*(intArr, intArr.length + 4);// 新数组超过源数组长度，不足补0

System.*out*.println(Arrays.*toString*(intArr4));

// equals,判断两个数组（长度，相同的索引位置的元素）是否完全相等，返回boolean值

System.*out*.println(Arrays.*equals*(intArr, intArr2));

System.*out*.println(Arrays.*equals*(intArr, intArr3));

// fill，使用指定元素填充数组

Arrays.*fill*(intArr3, 5);

System.*out*.println(Arrays.*toString*(intArr3));

// hashCode,不同于所有对象自带的hashCode方法，后者的hashcode值并不会因为对象的内部的

// 变化而发生改变，前者则会

System.*out*.println("intArr的hashcode:" + intArr.hashCode());

System.*out*.println("使用Arrays的hashCode方法计算的intArr的hashcode:"

+ Arrays.*hashCode*(intArr));

intArr[0] = 7;

System.*out*.println("改变第一个元素的intArr的hashcode:" + intArr.hashCode());

System.*out*.println("使用Arrays的hashCode方法计算的改变第一个元素的intArr的hashcode:"

+ Arrays.*hashCode*(intArr));

// asList，将数组转化为结构不可改变（不可添加，删除），但可以查询和修改的集合对象

List list1 = Arrays.*asList*(intArr3);// 将数组整体作为一个集合元素，该集合的元素类型为int数组

System.*out*.println(list1);

List list2 = Arrays.*asList*(5, 3, 7, 9, 2, 0, 6);// 该集合的元素类型为int

list2.add(1);// java.lang.UnsupportedOperationException

list2.remove(3);// java.lang.UnsupportedOperationException

list2.set(2, 10);

}

}

**Java集合之综述**

1、

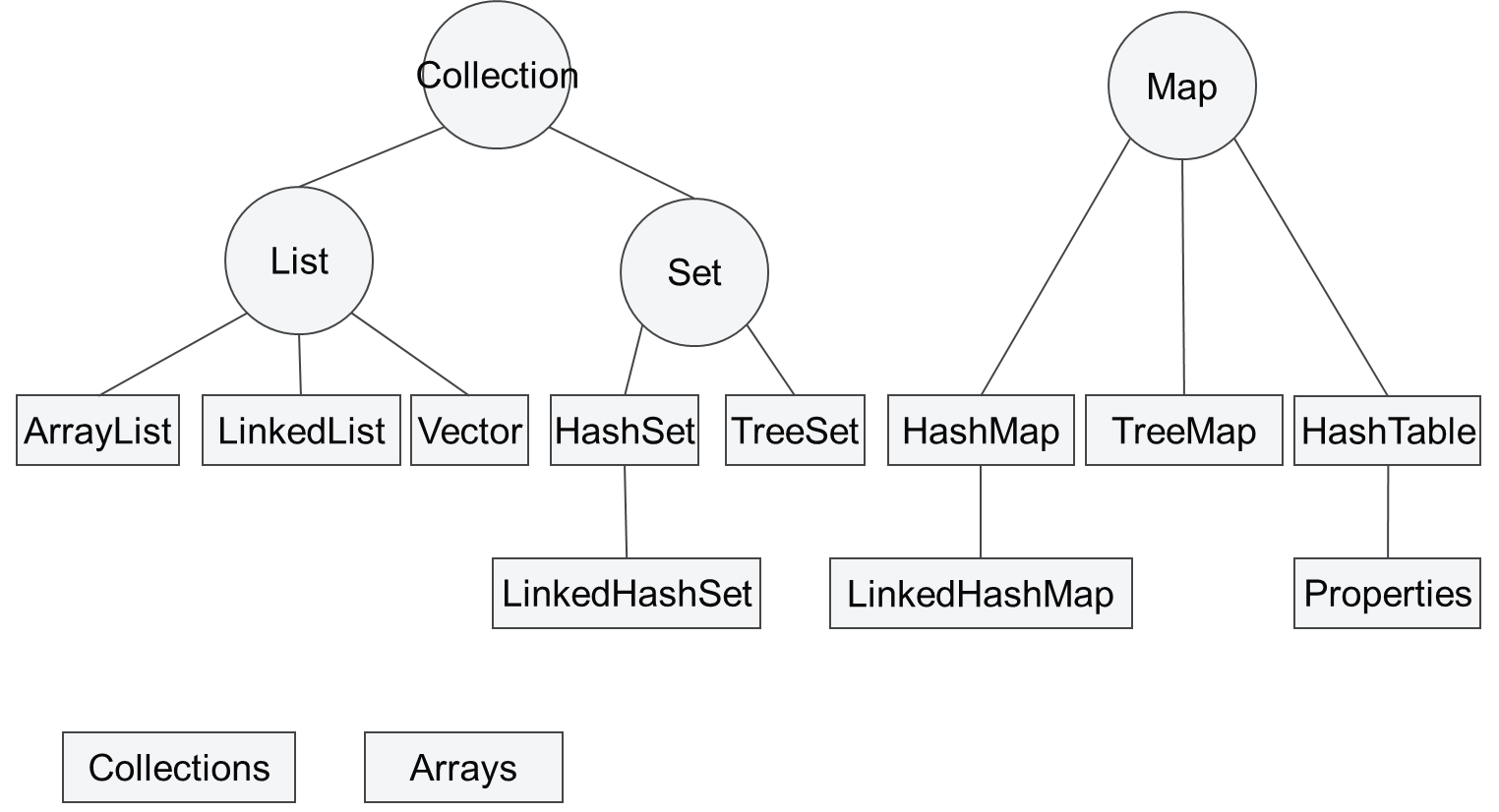
①集合中每个实现类都有自己独特的数据结构,而每种数据结构决定了该集合类所能进行的操作以及这些操作的特点和效率

②要求掌握集合的类继承体系,能够熟练使用讲到的所有集合类并熟悉他们的特点, 掌握他们的内部结构是如何影响他们的操作

③可以把整个集合体系分成三部分: Collection , Map ,工具类(Collectins和Arrays)

④集合是java基础的重点 , 必须精通

2、Java集合类层次结构



**Java集合之增强for循环**

1、

所有的集合类以及数组都可以使用

格式: for(Type p : types){ }

特点: 只能进行遍历操作,不能进行添加删除等操作

好处: 使集合遍历操作更简洁

2、

//测试增强for循环

//所有的集合类以及数组都可以使用

//格式: for(Type p : types){ }

//特点: 只能进行遍历操作,不能进行添加删除等操作

//好处: 使集合遍历操作更简洁

**public** **class** ZQForTest {

**public** **static** **void** main(String[] args) {

List<String> list = **new** ArrayList<String>();

list.add("谢宝发");

list.add("周杰伦");

list.add("乔布斯");

// JDK1.5新特性：增强for循环，取代迭代器对集合进行遍历，更加方便

**for** (String e : list) {

System.*out*.println(e);

}

// 还能对数组进行遍历

**int**[] a = **new** **int**[] { 1, 4, 2, 7, 46, 324, 54, 4534 };

**for** (**int** i : a) {

a[1] = 0;

System.*out*.println(i);

}

}

}