Java进阶1 第4天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】集合的回顾与迭代器的使用**
2. 【应用】能够独立写出迭代器遍历集合
3. 【应用】能够独立写出增强for遍历集合
4. 【应用】能够阐述并发修改异常产生的原因并用代码解决该异常
5. **【应用】泛型定义与使用**
6. 【理解】能够独立定义一个泛型类并使用
7. 【理解】能够独立定义一个泛型方法并调用
8. 【理解】能够独立定义一个泛型接口,并实现该接口
9. 【应用】能够独立阐述 ? extends E 和 ? super E 的含义
10. **【应用】斗地主案例.**
11. 【理解】能够独立写出斗地主案例
12. **【应用】List体系**
13. 【应用】能够独立使用LinkedList
14. 【应用】能够阐述数组和链表结构的实现原理
15. 【理解】能够理解不同数据结构的特点
16. **【应用】set体系**

a.【应用】能够使用HashSet去重(自定义对象)

b.【应用】能够用文字描述HashSet去重的过程

c.【应用】能够独立写出HashSet与增强for循环练习

1. **【应用】单列集合总结**
2. 【理解】能够阐述整个单列集合体系
3. 【理解】能够理解list和set集合中不同类的使用场景

# 集合的回顾与迭代器使用

## 集合的回顾

### 集合的创建

创建集合的常用格式在此说明一下：

A:导包：import java.util.ArrayList;

B:创建对象：与其他普通的引用数据类型创建方式完全相同，但是要指定容器中存储的数据类型：

ArrayList<要存储元素的数据类型> 变量名 = new ArrayList<要存储元素的数据类型>();

集合中存储的元素，只能为<>括号中指定的数据类型元素；

“<要存储元素的数据类型>”中的数据类型必须是引用数据类型，不能是基本数据类型；

下面给出8种基本数据类型所对应的引用数据类型表示形式:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| boolean | byte | short | char | int | long | float | double |
| Boolean | Byte | Short | Character | Integer | Long | Float | Double |

### 集合的常用方法

|  |  |
| --- | --- |
| **方法声明** | **功能描述** |
| **boolean add（Object obj）** | 将指定元素obj追加到集合的末尾 |
| **Object get（int index）** | 返回集合中指定位置上的元素 |
| **int size（）** | 返回集合中的元素个数 |

### 案例代码一:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: ArrayListDemo  \* **@Description**: ArrayList类的使用  \* **@date** 2017年11月17日 上午8:52:39  \* Company www.igeekhome.com  \*  \* 持有数据:  \* 单一数据,可以定义对应类型的变量直接持有  \* 多个数据,可以使用数组的方式记录,通过索引访问数据  \* 又在面向对象的基础上,可以使用集合的方式记录,最开始最常用ArrayList,该集合类同样可以通过索引访问数据  \*  \* 集合的基本使用:  \* 1创建集合对象  \* 2向集合中添加元素  \* 3从集合中获取元素/其他信息(集合的长度)  \*/  **public** **class** ArrayListDemo {  **public** **static** **void** main(String[] args) {    //创建集合对象  ArrayList<Integer> list = **new** ArrayList<Integer>();  //向集合中添加元素  list.add(10);  list.add(100);  list.add(2000);    //从集合中获取元素/其他信息(集合的长度)  Integer integer = list.get(2);  System.***out***.println(integer);    **int** size = list.size();  System.***out***.println(size);  System.***out***.println("===============================");    //遍历集合  **for** (**int** i = 0; i<list.size(); i++) {  //通过索引依次获取集合中每一个元素  Integer thisNumber = list.get(i);  //打印每个元素  System.***out***.println(thisNumber);  }  }  } |

## 迭代器

### 迭代器概述

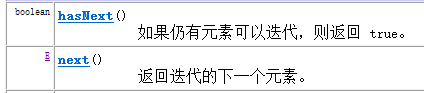
java中提供了很多个集合，它们在存储元素时，采用的存储方式不同。

我们要取出这些集合中的元素，可通过一种通用的获取方式来完成。

Collection集合元素的通用获取方式：在取元素之前先要判断集合中有没有元素；如果有，就把这个元素取出来，继续在判断；如果还有就再取出出来。一直把集合中的所有元素全部取出。这种取出方式专业术语称为迭代。

集合中把这种取元素的方式描述在Iterator接口中。

Iterator接口的常用方法如下：



hasNext()方法：用来判断集合中是否有下一个元素可以迭代。如果返回true,说明可以迭代。

next()方法：用来返回迭代的下一个元素，并把指针向后移动一位。

#### 案例代码二:

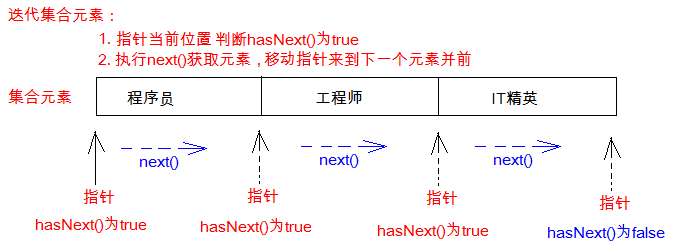
|  |
| --- |
| **package** com.igeek\_02\_01;  **import** java.util.ArrayList;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: IteratorDemo  \* **@Description**: 迭代器的使用  \* **@date** 2017年11月17日 上午8:58:11  \* Company www.igeekhome.com  \*  \* 在java当中不止ArrayList一个集合,java提供了众多的集合.  \*  \* 不同的容器完成不同方式的数据存储.  \* 不同集合的特点不同,ArrayList有序且可重复且带索引的集合.但是有的集合不带索引.所以如果使用其他集合,可能无法通过get+索引的方式获取元素  \*  \* 所有集合的通用获取元素方法并不是通过索引获取,而是通过迭代器获取.  \* 迭代器:iterator  \*/  **public** **class** IteratorDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Integer> list = **new** ArrayList<Integer>();    //向集合中添加元素  list.add(10);  list.add(100);  list.add(2000);    //获取该集合的迭代器  Iterator<Integer> iterator = list.iterator();    //使用迭代器的方法,迭代集合(遍历集合)  **while**(iterator.hasNext()) {  //获取集合中元素  Integer thisNumber = iterator.next();  System.***out***.println(thisNumber);  }  }  } |

### 迭代器迭代原理

#### 案例代码三:

|  |
| --- |
| **package** com.igeek\_02\_02;  **import** java.util.ArrayList;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: IteratorDemo  \* **@Description**: 迭代器原理  \* **@date** 2017年11月17日 上午8:58:11  \* Company www.igeekhome.com  \*  \* 集合用来持有数据，所有常用集合都具备了可迭代功能iterator方法，该方法用于迭代集合，该方法才是最为通用的集合迭代方法。  \*  \* 迭代器:集合迭代(集合遍历)的工具  \*  \* 集合的获取迭代器方法  \* public Iterator<E> iterator() 获取某个集合的迭代器实例对象  \*  \* Iterator:迭代器接口  \* booleanhasNext() 判断集合中是否有下一个元素不移动元素指针  \* E next() 获取集合中下一个元素移动指针,指向下一个元素  \*/  **public** **class** IteratorDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Integer> list = **new** ArrayList<Integer>();    //向集合中添加元素  list.add(10);  list.add(100);  list.add(2000);    //调用集合的iterator方法,返回迭代器对象  Iterator<Integer> itr = list.iterator();    //混合使用hasNext与next方法,循环获取元素  **while**(itr.hasNext()) {  //如果有元素,就获取元素  Integer thisNumber = itr.next();  System.***out***.println(thisNumber);  }    }  } |

迭代集合元素图解：



### 迭代器迭代自定义数据类型

A：定义一个Person类,里面有姓名和年龄属性,生成对应的getter/setter方法以及满参构造,空参构造

B：定义一个测试类,在测试类中定义一个集合,集合中存储三个Person对象,这三个人的属性分别为

Jack,18岁

Rose,16岁

Trump,62岁

C:最终遍历该集合,分别打印出这三个人的姓名和年龄

并且统计出三个人的总年龄打印在控制台上

#### 案例代码四:

|  |
| --- |
| **package** com.igeek\_02\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person人类  \* **@date** 2017年11月17日 上午9:07:32  \* Company www.igeekhome.com  \*  \* A：定义一个Person类,  \* 里面有姓名和年龄属性,生成对应的getter/setter方法,  \* 以及满参构造,空参构造  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

|  |
| --- |
| **package** com.igeek\_02\_03;  **import** java.util.ArrayList;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: ArrayListDemo  \* **@Description**: ArrayList的测试类  \* **@date** 2017年11月17日 上午9:09:24  \* Company www.igeekhome.com  \*  \* B：定义一个测试类,在测试类中定义一个集合,集合中存储三个Person对象,这三个人的属性分别为  \* Jack,18岁  \* Rose,16岁  \* Trump,62岁  \* C:最终遍历该集合,分别打印出这三个人的姓名和年龄  \* 并且统计出三个人的总年龄打印在控制台上  \*  \*/  **public** **class** ArrayListDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Person> list = **new** ArrayList<Person>();    //向集合中添加元素  Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",16);  Person p3 = **new** Person("Trump",62);    list.add(p);  list.add(p2);  list.add(p3);  //调用集合的iterator方法,返回迭代器对象  Iterator<Person> itr = list.iterator();    //定义变量,记录年龄综合  **int** sum = 0;    //混合使用hasNext与next方法,循环获取元素  **while**(itr.hasNext()) {  //如果有元素,就获取元素  Person thisP = itr.next();  System.***out***.println(thisP.getAge()+"岁的"+thisP.getName());  //年龄累加  sum += thisP.getAge();  }    //查看总年龄  System.***out***.println(sum);  }  } |

### 并发修改异常

在前一个迭代器迭代自定义数据类型的基础上,添加新的需求:

使用集合存储多个Person对象,当遇到16岁的人时,就添加一个90岁的人

#### 案例代码五:

|  |
| --- |
| **package** com.igeek\_02\_04;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person人类  \* **@date** 2017年11月17日 上午9:07:32  \* Company www.igeekhome.com  \*  \* A：定义一个Person类,  \* 里面有姓名和年龄属性,生成对应的getter/setter方法,  \* 以及满参构造,空参构造  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;    /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |

|  |
| --- |
| **package** com.igeek\_02\_04;  **import** java.util.ArrayList;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: ConcurrentModificationExceptionDemo  \* **@Description**: 迭代过程中修改产生的并发异常  \* **@date** 2017年11月17日 上午9:16:09  \* Company www.igeekhome.com  \*  \* 使用集合存储多个Person对象,当遇到16岁的人时,就添加一个90岁的人  \*  \* 该例中可能会产生并发修改异常:迭代器所认为的集合状态与集合真正的状态不统一了!就会出现该异常.  \*/  **public** **class** ConcurrentModificationExceptionDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Person> list = **new** ArrayList<Person>();  //向集合中添加元素  Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",16);  Person p3 = **new** Person("Trump",62);  Person p4 = **new** Person("Obama",56);  Person p5 = **new** Person("Castro",90);    list.add(p);  list.add(p2);  list.add(p3);  list.add(p4);  //调用集合的iterator方法,返回迭代器对象  Iterator<Person> itr = list.iterator();    //混合使用hasNext与next方法,循环获取元素  **while**(itr.hasNext()) {  //如果有元素,就获取元素  Person thisP = itr.next();  //判断年龄是否为16岁  **if**(thisP.getAge() == 16) {  //如果存在16岁的人,就加入90岁的卡斯特罗  list.add(p5);  }  }  }  } |

运行上述代码发生了错误java.util.ConcurrentModificationException这是什么原因呢？

在迭代过程中，使用了集合的方法对元素进行操作。导致迭代器并不知道集合中的变化，容易引发数据的不确定性。

解决并发修改异常,重新获取迭代器

#### 案例代码六:

|  |
| --- |
| **package** com.igeek\_02\_04;  **import** java.util.ArrayList;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: ConcurrentModificationExceptionDemo  \* **@Description**: 迭代过程中修改产生的并发异常  \* **@date** 2017年11月17日 上午9:16:09  \* Company www.igeekhome.com  \*  \* 使用集合存储多个Person对象,当遇到16岁的人时,就添加一个90岁的人  \*  \* 该例中可能会产生并发修改异常:迭代器所认为的集合状态与集合真正的状态不统一了!就会出现该异常.  \*/  **public** **class** ConcurrentModificationExceptionDemo2 {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Person> list = **new** ArrayList<Person>();  //向集合中添加元素  Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",16);  Person p3 = **new** Person("Trump",62);  Person p4 = **new** Person("Obama",56);  Person p5 = **new** Person("Castro",90);    list.add(p);  list.add(p2);  list.add(p3);  list.add(p4);  //调用集合的iterator方法,返回迭代器对象  Iterator<Person> itr = list.iterator();    //混合使用hasNext与next方法,循环获取元素  **while**(itr.hasNext()) {  //如果有元素,就获取元素  Person thisP = itr.next();  //判断年龄是否为16岁  **if**(thisP.getAge() == 16) {  //如果存在16岁的人,就加入90岁的卡斯特罗  list.add(p5);    //当集合加入了一个元素后,就不要再使用迭代器操作元素了  **break**;  }  }    //重新打印集合结果  Iterator<Person> itr2 = list.iterator();    **while** (itr2.hasNext()) {  Person person = (Person) itr2.next();  System.***out***.println(person.getAge()+"岁的"+person.getName());  }  }  } |

## 增强for

增强for循环是JDK1.5以后出来的一个高级for循环，专门用来遍历数组和集合的。

它的内部原理其实是个Iterator迭代器，所以在遍历的过程中，不能对集合中的元素进行增删操作。

它用于遍历Collection和数组。通常只进行遍历元素，不要在遍历的过程中对集合元素进行增删操作。

### 增强for格式

for(元素的数据类型变量 : Collection集合or数组){

}

### 案例代码七:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: ForeachDemo  \* **@Description**: 增强for循环  \* **@date** 2017年11月17日 上午9:25:01  \* Company www.igeekhome.com  \*  \* foreach循环(又叫增强for循环)来完成容器中元素的获取  \*  \* 增强for循环用来迭代集合或数组，格式如下：  \* for(容器内类型临时变量：容器) {  \* 内部可以直接使用临时变量访问数据  \* }  \*  \* 增强for循环源代码底层就是迭代器.所以不能在增强for循环的过程当中为集合添加或者删除元素.因为会产生并发修改异常.  \* 所以,增强for循环只用来查看数据,不作数据修改.  \*/  **public** **class** ForeachDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  ArrayList<Integer> list = **new** ArrayList<Integer>();  //向集合中添加元素  list.add(10);  list.add(100);  list.add(2000);    //使用增强for循环迭代集合  **for**(Integer thisNumber : list) {  System.***out***.println(thisNumber);  }  System.***out***.println("=======================");    //增强for循环便利数组  String[] arr = {"Jack","Rose","Trump","Obama"};    **for** (String thisName : arr) {  System.***out***.println(thisName);  }  }  } |

# 泛型定义与使用

## 泛型的概念

A：泛型用来灵活地将数据类型应用到不同的类、方法、接口当中。将数据类型作为参数传递。

B：泛型是数据类型的一部分，我们将类名与泛型合并一起看做数据类型

ArrayList<String> al= new ArrayList<String>();

//ArrayList<String>看作数据类型

C：泛型的定义：定义泛型可以在类中预支地使用未知的类型。

//在ArrayList类的add()方法的源码中定义的E e,使用了e

public boolean add(E e) {

ensureCapacityInternal(size + 1);

elementData[size++] = e;

return true;

}

D：泛型的使用：一般在创建对象时，将未知的类型确定具体的类型。当没有指定泛型时，默认类型为Object类型。

ArrayList al=new ArrayList();

al.add("abc");

al.add(1);

//由于在定义集合时没有指定泛型,add()方法的形参为Object类型,所以可以往集合中添加任意任意类型的数据(多态特点)

## 泛型类

### 格式:

当创建该类的对象的时候,传入类型,此时类上的泛型被确定

class 类名<E,T,……>{//<>中的泛型可以写无数个

//E,T……能够做为类型在该类内部被使用

}

### 案例代码八:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyClassType  \* **@Description**: 自定义类中使用泛型  \* **@date** 2017年11月17日 上午9:34:20  \* Company www.igeekhome.com  \*  \* 在该类中定义一个成员变量,使用泛型  \*  \* 不指定其数据类型,是一个不确定的数据类型.  \* 创建不同的对象时,指定不同的数据类型  \*  \*/  **public** **class** MyClassType<T> {    /\*\*  \* **@Fields** myField : 我的字段  \*/  **private** T myField;  /\*\*  \* **@return** the myField  \*/  **public** T getMyField() {  **return** myField;  }  /\*\*  \* **@param** myField the myField to set  \*/  **public** **void** setMyField(T myField) {  **this**.myField = myField;  }  } |

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: MyClassTypeDemo  \* **@Description**: 泛型使用的测试类  \* **@date** 2017年11月17日 上午9:36:23  \* Company www.igeekhome.com  \*  \* 泛型类:  \* 在类中使用泛型  \*  \* 定义：类名后<变量>如：class A<E> {使用E完全类的定义}  \* 使用：创建对象时确定类型  \*  \* 一般定义泛型使用E,T,V,K  \*  \* 泛型方法：  \* 定义：方法返回值前<变量>如：public <T> void method(){使用T}  \* 使用：调用方法时确定类型  \*/  **public** **class** MyClassTypeDemo {  **public** **static** **void** main(String[] args) {    //测试使用自己定义的泛型类  MyClassType<String> mct = **new** MyClassType<String>();  //调用了使用类泛型的方法  mct.setMyField("Jack");  String myField = mct.getMyField();  System.***out***.println(myField);    //创建对象时指定泛型为Integer则所有使用了泛型的方法根据对象类型一起改变  MyClassType<Integer> mct2 = **new** MyClassType<Integer>();  mct2.setMyField(12);  System.***out***.println(mct2.getMyField());  }  } |

## 泛型方法

### 格式

在调用泛型方法,并且传入实参的时候才能确定方法上形参的类型

修饰符 <T,Q,E……> 返回值类型 方法名(T t,Q q,……){//方法上的泛型定义在返回值的前面

}

### 案例代码九:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: MyClassType  \* **@Description**: 自定义类中使用泛型  \* **@date** 2017年11月17日 上午9:34:20  \* Company www.igeekhome.com  \*  \* 在该类中定义一个成员变量,使用泛型  \*  \* 不指定其数据类型,是一个不确定的数据类型.  \* 创建不同的对象时,指定不同的数据类型  \*  \*/  **public** **class** MyClassType {    //定义泛型方法  /\*\*  \* **@Title**: method  \* **@Description**: 泛型方法method  \* **@param** t  \*/  **public** <T> **void** method(T t) {  System.***out***.println(t);  }  } |

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: MyClassTypeDemo  \* **@Description**: 泛型使用的测试类  \* **@date** 2017年11月17日 上午9:36:23  \* Company www.igeekhome.com  \*  \* 泛型类:  \* 在类中使用泛型  \*  \* 定义：类名后<变量>如：class A<E> {使用E完全类的定义}  \* 使用：创建对象时确定类型  \*  \* 一般定义泛型使用E,T,V,K  \*  \* 泛型方法：  \* 定义：方法返回值前<变量>如：public <T> void method(){使用T}  \* 使用：调用方法时确定类型  \*/  **public** **class** MyClassTypeDemo {  **public** **static** **void** main(String[] args) {    //测试使用自己定义的类  MyClassType mct = **new** MyClassType();    //调用该类对象的泛型方法,是在调用方法时确定的数据类型  mct.method("abc");  mct.method(12);    }  } |

## 泛型接口

public static int abs(int a)：返回a的绝对值  
public static double ceil(double a) ：求大于等于a的最小整数,返回该整数的小数形式

public static double floor(double a) ：求小于等于a的最小整数

public static long round(double a) ：对a进行四舍五入,返回四舍五入后的值

public static double pow(double a, double b) ：求a的b次幂,返回double形式

### 格式一:

interface 接口名<Q,E,R……>{

public abstract void method(T t);//假设定义一个方法

}

class 类名 implements 接口<实际类型>{

@Override

public void method(实际类型t) {

System.out.println(t);

}

}

### 案例代码十:

|  |
| --- |
| **package** com.igeek\_04\_01;  /\*\*  \* **@ClassName**: MyInterType  \* **@Description**: 自定义泛型接口  \* **@date** 2017年11月17日 上午10:22:09  \* Company www.igeekhome.com  \*  \*/  **public** **interface** MyInterType<T> {    //接口的方法中使用接口泛型  /\*\*  \* **@Title**: method  \* **@Description**: 接口中的抽象方法method  \* **@param** t  \*/  **public** **abstract** **void** method(T t);  } |

|  |
| --- |
| **package** com.igeek\_04\_01;  /\*\*  \* **@ClassName**: MyClassInter  \* **@Description**: 实现泛型接口的普通类  \* **@date** 2017年11月17日 上午10:24:24  \* Company www.igeekhome.com  \*  \* 定义类时,就指定接口中的数据类型,定义一个不含泛型的正常类  \*/  **public** **class** MyClassInter **implements** MyInterType<String> {  /\*\*  \* **@Title**: method  \* **@Description**: 使用指定的接口中的数据类型,重写方法  \* **@param** t  \* **@see** com.igeek\_04.MyInterType#method(java.lang.Object)  \*/  @Override  **public** **void** method(String t) {  System.***out***.println(t);  }    } |

|  |
| --- |
| **package** com.igeek\_04\_01;  /\*\*  \* **@ClassName**: MyClassInterDemo  \* **@Description**: 实现泛型接口的普通类的测试类  \* **@date** 2017年11月17日 上午10:27:42  \* Company www.igeekhome.com  \*  \* 泛型接口:  \* 定义：接口后<变量>如： interface B<T>{使用T完成接口定义}  \* 使用：  \* 1、定义类时确定类型  \* 2、始终不确定类型，直到创建对象时确定类型  \*/  **public** **class** MyClassInterDemo {  **public** **static** **void** main(String[] args) {  //1、定义类时确定类型  MyClassInter mci = **new** MyClassInter();  mci.method("abc");  }  } |

### 格式二:

interface 接口名<Q,E,R……>{

public abstract void method(T t);//假设定义一个方法

}

class 类名<T> implements 接口名<T>{

//由于没有指定接口中具体的数据类型,所以方法中仍然使用这种不确定的数据类型

@Override

public void method(T t) {

}

}

### 案例代码十一:

|  |
| --- |
| **package** com.igeek\_04\_03;  /\*\*  \* **@ClassName**: MyInterType  \* **@Description**: 自定义泛型接口  \* **@date** 2017年11月17日 上午10:22:09  \* Company www.igeekhome.com  \*  \*/  **public** **interface** MyInterType<T> {    //接口的方法中使用接口泛型  /\*\*  \* **@Title**: method  \* **@Description**: 接口中的抽象方法method  \* **@param** t  \*/  **public** **abstract** **void** method(T t);  } |

|  |
| --- |
| **package** com.igeek\_04\_03;  /\*\*  \* **@ClassName**: MyClassInter  \* **@Description**: 实现泛型接口的泛型类  \* **@date** 2017年11月17日 上午10:24:24  \* Company www.igeekhome.com  \*  \* 定义类时,仍然不指定接口中的数据类型  \* 则此时相当于将这种不确定的数据类型用到了类中  \* 包含不确定的数据类型的类,是泛型类,如同一个泛型类的定义和使用  \*/  **public** **class** MyClassInter<T> **implements** MyInterType<T> {  //由于没有指定接口中具体的数据类型,所以方法中仍然使用这种不确定的数据类型  /\*\*  \* **@Title**: method  \* **@Description**: 实现了接口中的方法  \* **@param** t  \* **@see** com.igeek\_04\_03.MyInterType#method(java.lang.Object)  \*/  @Override  **public** **void** method(T t) {  System.***out***.println(t);  }  } |

|  |
| --- |
| **package** com.igeek\_04\_03;  /\*\*  \* **@ClassName**: MyClassInterDemo  \* **@Description**: 实现泛型接口的泛型类的测试类  \* **@date** 2017年11月17日 上午10:27:42  \* Company www.igeekhome.com  \*  \* 泛型接口:  \* 定义：接口后<变量>如： interface B<T>{使用T完成接口定义}  \* 使用：  \* 1、定义类时确定类型  \* 2、始终不确定类型，直到创建对象时确定类型  \*/  **public** **class** MyClassInterDemo {  **public** **static** **void** main(String[] args) {  //2、始终不确定类型，直到创建对象时确定类型  MyClassInter<Integer> mci = **new** MyClassInter<Integer>();  mci.method(22);  }  } |

## 泛型通配符

泛型通配符：?

定义：(查看ArrayList的构造方法)无法在类中使用

使用：调用方法时可以给予任意类型。参照Arraylist的构造方法

? extends E代表只要是E类型的子类即可

? super E代表只要是E类型的父类即可

### 案例代码十二:

|  |
| --- |
| **package** com.igeek\_05;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: GenericDemo  \* **@Description**: 泛型通配符  \* **@date** 2017年11月17日 上午10:44:07  \* Company www.igeekhome.com  \*  \* 泛型通配符?,代表任意的数据类型  \*  \* 定义：(查看ArrayList的构造方法)无法在类中使用  \*  \* 使用：调用方法时可以给予任意类型。  \* 参照Arraylist的构造方法: public ArrayList(Collection<? extends E> c)  \*  \* 为了便于?的理解,我们将以上方法重写为：public ArrayList(ArrayList<? extends E> c)  \*  \* 该方法的意思:创建集合对象A时,给于另外一个集合对象B作为参数,则创建好的集合A中包含了集合B中的元素  \*  \* ? extends E代表只要是E类型的子类即可  \* ? super E代表只要是E类型的父类即可  \*  \*/  **public** **class** GenericDemo {  **public** **static** **void** main(String[] args) {    //定义集合b,包含3个元素  ArrayList<String> listB = **new** ArrayList<String>();  listB.add("Jack");  listB.add("Rose");  listB.add("Trump");    //使用集合b创建集合a  //listB中的元素类型String必须是listA中元素类型Object的子类  ArrayList<Object> listA = **new** ArrayList<Object>(listB);  listA.add("Obama");    //观察集合A  System.***out***.println(listA);    }  } |

## 泛型优点:

提高程序的安全性

将运行期问题转移到了编译期

省去了类型强转的麻烦

### 代码案例十三:

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月17日 上午11:00:47  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }    } |

|  |
| --- |
| **package** com.igeek\_06;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: GenericDemo  \* **@Description**: 泛型优点  \* **@date** 2017年11月17日 上午10:59:32  \* Company www.igeekhome.com  \*  \* 提高程序的安全性  \* 将运行期问题转移到了编译期  \* 省去了类型强转的麻烦  \*  \*/  **public** **class** GenericDemo {  **public** **static** **void** main(String[] args) {  //不使用泛型,则默认使用Object类型  ArrayList list = **new** ArrayList();    list.add("Jack");  list.add("Rose");  list.add("Trump");  list.add("Obama");  list.add(123);  list.add(**new** Person());    //迭代集合,使用增强for循环  **for**(Object o:list) {    String s = (String)o;//存入的Object类型取出也是Object类型,使用String特有方法必须强转,非常麻烦  System.***out***.println(s.length());  }  }  } |

# 斗地主案例

## 斗地主案例需求:

按照斗地主的规则，完成洗牌发牌的动作。

♣♦♠♥大☺小☺

具体规则：

使用54张牌打乱顺序

三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌。

## 斗地主需求分析:

A:准备牌：

牌可以设计为一个ArrayList<String>,每个字符串为一张牌。

每张牌由花色、数字两部分组成，我们可以使用花色集合与数字集合嵌套迭代完成每张牌的组装。

牌由Collections类的shuffle方法进行随机排序。

B:发牌：

将每个人以及底牌设计为ArrayList<String>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。

C:看牌：

直接打印每个集合

## 斗地主案例实现

### 案例代码十四:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.ArrayList;  **import** java.util.Collections;  /\*\*  \* **@ClassName**: CardDemo  \* **@Description**: 按照斗地主的规则，完成洗牌发牌的动作  \* **@date** 2017年11月17日 上午11:11:07  \* Company www.igeekhome.com  \*  \* ♣♦♠♥大☺小☺  \* 具体规则：  \* 使用54张牌打乱顺序  \* 三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌。  \*  \* 逻辑分析:  \*  \* 准备牌：  \* 牌可以设计为一个ArrayList<String>,每个字符串为一张牌。  \* 每张牌由花色数字两部分组成，我们可以使用花色集合与数字集合嵌套迭代完成每张牌的组装。  \* 牌由Collections类的shuffle方法进行随机排序。  \* 发牌：  \* 将每个人以及底牌设计为ArrayList<String>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。  \* 看牌：  \* 直接打印每个集合  \*/  **public** **class** CardDemo {  **public** **static** **void** main(String[] args) {  //准备牌  //定义集合记录一副牌  ArrayList<String> poker = **new** ArrayList<String>();    //定义花色集合  ArrayList<String> colors = **new** ArrayList<String>();  colors.add("♥");  colors.add("♠");  colors.add("♦");  colors.add("♣");    //定义数字集合  ArrayList<String> numbers = **new** ArrayList<String>();  **for** (**int** i = 2; i<= 10; i++) {  numbers.add(i+"");  }  numbers.add("J");  numbers.add("Q");  numbers.add("K");  numbers.add("A");  //嵌套迭代完成每张牌的组装  **for** (String thisColor : colors) {  **for** (String thisNumber : numbers) {  //通过花色与数字拼写成对应的每张牌  String thisCard = thisColor+thisNumber;  //将拼好的牌放到集合中  poker.add(thisCard);  }  }  //补充大小王  poker.add("大☺");  poker.add("小☺");  //System.out.println(poker);    //洗牌 Collections类.shuffle方法可以将集合打乱顺序  Collections.*shuffle*(poker);  //System.out.println(poker);  //发牌：  //将每个人以及底牌设计为ArrayList<String>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。  ArrayList<String> player1 = **new** ArrayList<String>();  ArrayList<String> player2 = **new** ArrayList<String>();  ArrayList<String> player3 = **new** ArrayList<String>();  ArrayList<String> dipai = **new** ArrayList<String>();  //使用普通的for循环,通过索引处理纸牌  //除了最后三张，其他的都发给玩家  **for** (**int** i = 0; i<poker.size()-3; i++) {  //获取当前这张牌  String thisCard = poker.get(i);    //对3取模,决定给哪个玩家发牌  **if**(i%3==0) {  player1.add(thisCard);  }**else** **if**(i%3==1) {  player2.add(thisCard);  }**else** {  player3.add(thisCard);  }  }    //最后3张是底牌  **for**(**int** i=poker.size()-3; i<poker.size(); i++){  //获取当前这张牌  String thisCard = poker.get(i);  dipai.add(thisCard);  }    //看牌：  //直接打印每个集合  System.***out***.println(player1);  System.***out***.println(player2);  System.***out***.println(player3);  System.***out***.println(dipai);  }  } |

# LinkedList的使用&数据结构

## LinkedList的基本使用

LinkedList与ArrayList不同，LinkedList是方便添加或删除的List。实际开发中对一个集合元素的添加与删除经常涉及到首尾操作，

所以该具体子类的特点在于提供了大量首尾操作。

public void **addFirst**(E e) 添加首个元素

public void **addLast**(E e) 添加最后元素

public E **getFirst**() 获取首个元素

public E **getLast**() 获取最后元素

### 案例代码十五:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Iterator;  **import** java.util.LinkedList;  /\*\*  \* **@ClassName**: LinkedListDemo  \* **@Description**: LinkedList的测试类  \* **@date** 2017年11月17日 下午1:41:16  \* Company www.igeekhome.com  \*  \* LinkedList是方便添加删除的List。  \* 提供了大量首尾操作  \*  \* public void addFirst(E e) 添加首个元素  \* public void addLast(E e) 添加最后元素  \* public E getFirst() 获取首个元素  \* public E getLast() 获取最后元素  \*/  **public** **class** LinkedListDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  LinkedList<String> list = **new** LinkedList<String>();    //向集合中添加元素  list.add("Jack");  list.add("Rose");  list.add("Trump");    //获取元素  String name = list.get(1);  System.***out***.println(name);    //返回集合的迭代器  Iterator<String> itr = list.iterator();  **while**(itr.hasNext()) {  String thisName = itr.next();  System.***out***.println(thisName);  }  System.***out***.println("=======================");    //使用增强for循环  **for** (String thisName : list) {  System.***out***.println(thisName);  }    System.***out***.println("======================");    //做首尾的添加与删除动作  System.***out***.println(list);    list.addFirst("LiLei");  list.addLast("HanMeiMei");    System.***out***.println(list);    System.***out***.println(list.getFirst());  System.***out***.println(list.getLast());  }  } |

## 数据结构

每种集合的特点不同(如更适合增删还是更适合查找)，是因为底层的数据结构不同。

数据结构指的数据存储和组织方式， 这里介绍最常用的数据结构：

### 栈结构

容器先进后出规则

#### 案例代码十六:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.LinkedList;  /\*\*  \* **@ClassName**: StackDemo  \* **@Description**: stack栈结构  \* **@date** 2017年11月17日 下午1:44:14  \* Company www.igeekhome.com  \*  \* stack栈结构:先进后出  \*  \* LinkedList是支持栈结构的.  \* 对应的方法  \* push压栈  \* pop弹栈  \* peek查询出即将要弹出的是哪个元素,检查有没有要弹出的元素  \*  \* java提供了一个专门用于栈结构的类,Stack  \*/  **public** **class** StackDemo {  **public** **static** **void** main(String[] args) {  //创建栈结构的集合  LinkedList<String> stack = **new** LinkedList<String>();    stack.push("Jack");  stack.push("Rose");  stack.push("Trump");    System.***out***.println(stack);    //获取一个元素,此时,由于是栈结构,所以获取的是最后压栈的元素  String popName = stack.pop();  System.***out***.println(popName);    //弹栈动作,集合中减少元素  System.***out***.println(stack);    String peekName = stack.peek();  System.***out***.println(peekName);    //peek查看元素不会减少元素  System.***out***.println(stack);  }  } |

数组结构：一块连续的存储区域

链表结构：每个元素指向下一个元素

### 队列结构

容器先进先出的规则

#### 案例代码十七:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.LinkedList;  /\*\*  \* **@ClassName**: QueueDemo  \* **@Description**: queue队列结构  \* **@date** 2017年11月17日 下午1:50:03  \* Company www.igeekhome.com  \*  \* queue队列结构:先进先出  \*  \* LinkedList是支持队列结构的  \*  \* 对应的方法  \* offer加入队列  \* poll离开队列  \* peek查询出即将要离开队列的是哪个元素,检查有没有要离开队列的元素  \*/  **public** **class** QueueDemo {  **public** **static** **void** main(String[] args) {  //创建队列结构的集合  LinkedList<String> queue = **new** LinkedList<String>();    queue.offer("Jack");  queue.offer("Rose");  queue.offer("Trump");    System.***out***.println(queue);    //获取一个元素,此时,由于是队列结构,所以获取的是最早入队的元素  String pollName = queue.poll();  System.***out***.println(pollName);    //出队动作,集合中减少元素  System.***out***.println(queue);    String peekName = queue.peek();  System.***out***.println(peekName);    //peek查看元素不会减少元素  System.***out***.println(queue);  }  } |

## ArrayList数组实现的原理

数组实现的特点：查询快，增删慢

原因：

查询快：由于数组的索引支持，那么可以通过索引直接计算出元素的地址值，因此就可以直接通过元素的地址值获取到指定的元素

增删慢：由于在添加元素的时候，实际上底层会先创建一个新数组(新数组的长度为原数组的长度+1)，那么在添加新元素的时候，先需要对数组中原有的数据进行拷贝，其次在末尾进行添加新的元素

因此，这样操作的效率的极低的(删除元素 刚好和添加的操作相反)

## LinkedList链表实现的原理

链表结构：查询慢，增删快

查询慢：由于不能直接找到元素的地址，需要上一个元素推导出下一个元素的地址，

这种查询速度较慢

增删快：在添加的时候，只需要更改元素所记录的地址值即可

# HashSet

## HashSet的特性和基本使用

HashSet是Set接口的子类，不包含重复元素相同元素，且无序。

HashSet下还有子类LinkedHashSet，可预测迭代顺序的Set集合。

### 案例代码十八:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.HashSet;  **import** java.util.Iterator;  /\*\*  \* **@ClassName**: HashSetDemo  \* **@Description**: HashSet的测试类  \* **@date** 2017年11月17日 下午2:04:48  \* Company www.igeekhome.com  \*  \* HashSet，不包含重复元素相同元素，且无序，不提供索引，所以不能通过索引获取元素，只能通过迭代器访问数据。  \*/  **public** **class** HashSetDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  HashSet<String> set = **new** HashSet<String>();  //向集合中添加元素  set.add("Jack");  set.add("Rose");  set.add("Trump");  set.add("Obama");  set.add("Obama");    System.***out***.println(set);    //获取元素,只能用迭代器  Iterator<String> itr = set.iterator();    **while**(itr.hasNext()) {  String thisName = itr.next();  System.***out***.println(thisName);  }    System.***out***.println("================");  **for** (String thisName : set) {  System.***out***.println(thisName);  }  }  } |

## HashSet与增强for循环练习

### 案例代码十九:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.HashSet;  /\*\*  \* **@ClassName**: HashSetTest  \* **@Description**: HashSet的练习类  \* **@date** 2017年11月17日 下午2:06:59  \* Company www.igeekhome.com  \*  \* 求一个字符串"aiodjl;hriWFUADJSVUEHiowfjnivowe"中一共有几个不重复的字母,区分大小写,如a,A算两个字符  \*  \* 分析：  \* 将字符串拆分成一个个单个字符  \* 判断字符是否为字母  \* 如果是字母,放到一个HashSet集合中  \* 查看集合长度  \*  \*/  **public** **class** HashSetTest {  **public** **static** **void** main(String[] args) {  String s = "aiodjl;hriWFUADJSVUEHiowfjnivowe";    //定义不包含重复元素的HashSet集合  HashSet<Character> set = **new** HashSet<Character>();    //将字符串拆分成一个个单个字符  **char**[] charArray = s.toCharArray();    //使用增强for循环遍历数组  **for**(**char** c : charArray) {    //判断字符是否为字母  **if**(('a'<=c&&c<='z')||('A'<=c&&c<='Z')) {    //如果是字母,放到一个HashSet集合中  set.add(c);  }  }    //查看集合长度  System.***out***.println("在该字符串中,一共存在不重复的字母"+set.size()+"个");  }  } |

## ArrayList中contains方法如何判断是否有重复元素

自定义类型Person类是Object类的子类，所以Person具备equals方法contains方法会调用参数的equals方法，依次与集合当中已经存在的Person对象比较。

当Person类没有重写equals方法时，直接使用Object类基础过来的equals方法，而该方法比较的是对象地址值。

当Person类重写equals方法后，就可以讲比较规则由比较地址值改为比较属性值。

### 案例代码二十:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: ArrayListContainsDemo  \* **@Description**: ArrayList中contains方法  \* **@date** 2017年11月17日 下午2:14:07  \* Company www.igeekhome.com  \*  \*/  **public** **class** ArrayListContainsDemo {  **public** **static** **void** main(String[] args) {  //创建ArrayList对象  ArrayList<String> list = **new** ArrayList<String>();    //在list中加入数据  list.add("Jack");  list.add("Rose");  list.add("Trump");    //判断集合中是否存在Jack  System.***out***.println(list.contains("Jack"));  System.***out***.println("========================");  //ArrayList当中存储自定义数据类型Person  ArrayList<Person> list2 = **new** ArrayList<Person>();    Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",16);  Person p3 = **new** Person("Trump",62);    list2.add(p);  list2.add(p2);  list2.add(p3);    System.***out***.println(list2.contains(**new** Person("Jack", 18)));  System.***out***.println(list2.contains(p));  }  } |

## 自定义类重写equals方法

自定义类需要满足咱们认为的效果，也就是不同的对象但是属性值相同就认为是同一个对象

在集合里只存其一的效果，就必须重写equals方法，且更改该方法里的比较规则

### 案例代码二十一:

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月17日 下午2:16:25  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {    /\*\*  \* **@Fields** name :姓名  \*/  **private** String name;    /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name=name;  **this**.age=age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  /\*  \* 如果没有重写该equals方法,则比较地址值.  \* 我们希望比较两个对象时,比较属性内容,所以重写该方法.  \*/  /\*\*  \* **@Title**: equals  \* **@Description**: Person对象比较是否相等的方法  \* **@param** obj  \* **@return**  \* **@see** java.lang.Object#equals(java.lang.Object)  \*/  @Override  **public** **boolean** equals(Object obj) {  System.***out***.println("equals方法被调用了");    //this:contains方法的参数对象  //obj:集合当中原有元素  //将Object类型的obj强转为子类类型  Person otherP = (Person)obj;    //比较姓名  **if**(!**this**.name.equals(otherP.name)) {  **return** **false**;  }  //比较年龄  **if**(**this**.age != otherP.age){  **return** **false**;  }  //如果所有属性值均相同,就返回true  **return** **true**;  }  } |

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: ArrayListContainsDemo  \* **@Description**: ArrayList中contains方法  \* **@date** 2017年11月17日 下午2:14:07  \* Company www.igeekhome.com  \*  \*/  **public** **class** ArrayListContainsDemo {  **public** **static** **void** main(String[] args) {  //创建ArrayList对象  ArrayList<String> list = **new** ArrayList<String>();    //在list中加入数据  list.add("Jack");  list.add("Rose");  list.add("Trump");    //判断集合中是否存在Jack  System.***out***.println(list.contains("Jack"));  System.***out***.println("========================");  //ArrayList当中存储自定义数据类型Person  ArrayList<Person> list2 = **new** ArrayList<Person>();    Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",16);  Person p3 = **new** Person("Trump",62);    list2.add(p);  list2.add(p2);  list2.add(p3);    System.***out***.println(list2.contains(**new** Person("Jack", 18)));  System.***out***.println(list2.contains(p));  }  } |

## HashSet判断元素唯一性规则

Set集合不能存放重复元素，其添加方法在添加时会判断是否有重复元素，有重复不添加，没重复则添加。

HashSet集合由于是无序的，其判断唯一的依据是元素类型的hashCode与equals方法的返回结果。

规则如下：

把对象加入到HashSet时，它会先使用对象的hashCode值来判断对象加入的位置；

如果此位置上没有其他对象存在，则判断元素不同，可存入新对象。

如果此位置上有其他对象存在，先判断新元素与集合内已经有的旧元素的HashCode值：

如果不同，判断元素不同，可存入新对象。

如果相同，再判断equals比较结果，返回true则相同，则不能存入新对象；返回false则仍然不同，可存入新对象。

所以，使用HashSet存储自定义类型，如果没有重写该类的hashCode与equals方法，则判断重复时，使用的地址值，如果想通过内容比较元素是否相同，需要重写该类的hashCode与equals方法。

hashCode方法重写规则：

将该对象的各个属性值hashCode相加即是整个对象的HashCode值。

如果是基本类型，类似int，则直接返回int值就是该属性的hash值；

如果是引用类型，类似String，就调用该成员变量的hashCode方法返回该成 员变量hash值。

这样可以根据对象的内容返回hashCode值，从而可以根据hashCode判断 元素是否唯一。

但是由于在一些”碰巧的”情况下，可能出现内容不同但hashCode相同的情 况，为了避免这些情况，我们加入一些干扰系数。

可是加入干扰系数后，仍会出现一些”碰巧”的情况，所以我们还要进行 equals的二次判断。

## 重写hashCode和equals方法

### 案例代码二十二:

|  |
| --- |
| **package** com.igeek\_06;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月17日 下午2:16:25  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {    /\*\*  \* **@Fields** name :姓名  \*/  **private** String name;    /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name=name;  **this**.age=age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  /\*  \* 如果没有重写该equals方法,则继承父类方法,比较地址值.  \* 我们希望比较两个对象时,比较属性内容,所以重写该方法.  \*/  /\*\*  \* **@Title**: equals  \* **@Description**: Person对象比较是否相等的方法  \* **@param** obj  \* **@return**  \* **@see** java.lang.Object#equals(java.lang.Object)  \*/  @Override  **public** **boolean** equals(Object obj) {  System.***out***.println("equals方法被调用了");    //this:contains方法的参数对象  //obj:集合当中原有元素  //将Object类型的obj强转为子类类型  Person otherP = (Person)obj;    //比较姓名  **if**(!**this**.name.equals(otherP.name)) {  **return** **false**;  }  //比较年龄  **if**(**this**.age != otherP.age){  **return** **false**;  }  //如果所有属性值均相同,就返回true  **return** **true**;  }  //没有重写hashCode方法时,hashCode方法继承父类方法,返回的是地址值  //重写hashCode方法,通过属性值返回一个整数数字  //即对象的hashCode值,就是各个属性的hashCode值之和  //引用数据类型属性,调用hashCode方法回去  //基本数值类型属性,直接就是数值本身  /\*\*  \* **@Title**: hashCode  \* **@Description**: Person对象的hashCode值算法方法  \* **@return**  \* **@see** java.lang.Object#hashCode()  \*/  @Override  **public** **int** hashCode() {    //张三 18 >> 56\*系数 + 18 = 74  //李四 56 >> 18\*系数 + 56 = 74  //理想上,不同属性值应该返回不同的hashCode值,可以在每次结果后乘以一个固定系数,避免该情况发生    //定义变量,记录要返回的hashCode值  **int** thisHashCode = 0;    //获取name属性的hashCode值  thisHashCode += name.hashCode()\*17;  //获取age数只给你的hashCode值  thisHashCode += age;    **return** thisHashCode;  }    } |

## Eclipse生成hashCode和equals方法

eclipse可以使用快捷键自动生成对应的hashCode方法与equals方法

### 案例代码二十三:

|  |
| --- |
| **package** com.igeek\_07;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月17日 下午2:16:25  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {    /\*\*  \* **@Fields** name :姓名  \*/  **private** String name;    /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **this**.name=name;  **this**.age=age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }  /\*\*  \* **@Title**: hashCode  \* **@Description**: Person对象的hashCode值算法方法  \* **@return**  \* **@see** java.lang.Object#hashCode()  \*/  @Override  **public** **int** hashCode() {  //定义系数  **final** **int** prime = 31;  //定义返回的hashCode值  **int** result = 1;  //每次将结果\*系数,再加下一个成员变量,可以大大降低不同成员变量返回相同hashCode值的情况  result = prime \* result + age;  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  }  /\*\*  \* **@Title**: equals  \* **@Description**: Person对象比较是否相等的方法  \* **@param** obj  \* **@return**  \* **@see** java.lang.Object#equals(java.lang.Object)  \*/  @Override  **public** **boolean** equals(Object obj) {  //自身与自身比较直接返回true  **if** (**this** == obj)  **return** **true**;    //参数是null直接返回false  **if** (obj == **null**)  **return** **false**;    //如果本对象的类型与参数对象的类型不一样,则直接返回false  **if** (getClass() != obj.getClass())  **return** **false**;    //如果类型相同,就转成(强制类型转换)相同的类型继续比较属性值  Person other = (Person) obj;  //比较年龄  **if** (age != other.age)  **return** **false**;    //比较姓名  **if** (name == **null**) {  **if** (other.name != **null**)  **return** **false**;  } **else** **if** (!name.equals(other.name))  **return** **false**;    //如果一切不相同的情况都不符合,则说明两个对象的属性值相同,返回true  **return** **true**;  }  } |

# Java单列集合框架之总结&体系图

## Java集合框架之总结&体系图

### 单列集合体系图

LinkedHashSet

HashSet

LinkedList

ArrayList

Set

List

Collection

Collection：所有单列集合的直接或间接接口，其指定了所有集合应该具备的基本功能。

List：元素可重复，有序，带索引

Set：元素不能重复，没有索引

ArrayList(重要) ：底层是数组结构。ArrayList的出现替代了Vector，增删慢，查找快

LinkedList(重要)：底层是链表结构。同时对元素的增删操作效率很高

HashSet(重要) ：底层是哈希表结构。在不重复的基础上无序

LinkedHashSet：底层是哈希表结构结合链表结构。在不重复的基础上可预测迭代顺序

### 集合的常见使用

在设计集合的使用时，经常定义对应的接口类型。在使用时，通过多态的方式给接口变量赋值，用于提高程序扩展性。

如：

定义方法public void method(List list) {}，在调用方法时依据需求，传入ArrayList或者LinkedList的对象。

重点和总结

1、迭代器，增强的for循环语句

2、泛型的定义与使用：泛型类、泛型方法、泛型接口，及通配符

4、LinkedList的使用及其内部实现原理

5、ArrayList的内部实现原理

6、HashSet的使用及其内部实现原理

7、重要的hashCode方法和equals方法的作用和使用

8、Java中单列集合框架的体系图