**3.1、链表（理解）**

链表是一种最基本的数据结构，但是对于数据结构的部分，强调以下几点：

· 在整个Java开发领域之中，没有一本真正去讲解数据结构的书，只能去看C语言的数据结构；

· 在所有的开发之中，都会存在数据结构的身影，可以这样去解释：数据结构的精通与否，完全决定于以后；

· 数据结构的核心：引用数据类型操作。

**3.1.1、链表的基本概念**

链表实际上可以理解为一串数据，或者按照专业性的说法，可以理解为动态的对象数组，对象数组最大优点：是表示出多的概念，例如：多个雇员。但是传统的对象数组有一个最大的问题在于，里面保存的数据的长度是固定的。

**思考：**如果说现在要想扩大一个对象数组的范围？

· 建立一个新的对象数组，而后将原本的内容拷贝到新数组之中，再改变原数组的引用方式。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  String [] str = new String[] {"Hello","World","MLDN"} ;  String [] newStr = new String[6] ;  System.arraycopy(str,0,newStr,0,str.length) ;  str = newStr ; // 改变引用，存在垃圾  str [3] = "你好" ;  str [4] = "世界" ;  for (int x = 0 ; x < str.length ; x ++) {  System.out.println(str[x]) ;  }  }  } |

通过以上的分析，可以发现，对象数组所有的对象元素被数组的索引控制，可以说是自动完成的控制，但是链表需要人为进行关系的设置，而且每个操作设置的时候，除了要保存“对象”之外，还要再多保留一个引用。

这个引用就和之前讲解的领导是一样的：一个雇员的领导还是雇员，雇员领导的领导也是雇员。

**范例：**先给出链表的基本操作结构

|  |
| --- |
| class Node {  private String data ; // 假设要保存的数据类型是字符串  private Node next ;  public Node(String data) {  this.data = data ;  }  public String getData() {  return this.data ;  }  public void setNext(Node next) {  this.next = next ;  }  public Node getNext() {  return this.next ;  }  } |

**范例：**挂节点

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  // 1、设置数据  Node n1 = new Node("火车头") ;  Node n2 = new Node("车厢A") ;  Node n3 = new Node("车厢B") ;  n1.setNext(n2) ;  n2.setNext(n3) ;  // 2、取出数据  Node currentNode = n1 ; // 设置每一个当前节点  while(currentNode != null) { // 有节点存在  System.out.println(currentNode.getData()) ; // 当前节点内容  currentNode = currentNode.getNext() ;  }  }  } |

但是这样的方式来输出所有的节点的配置过程，发现并不是特别好，这种输出的操作，应该采用递归合适。

|  |
| --- |
| public class TestDemo {  public static void main(String args[]) {  // 1、设置数据  Node n1 = new Node("火车头") ;  Node n2 = new Node("车厢A") ;  Node n3 = new Node("车厢B") ;  n1.setNext(n2) ;  n2.setNext(n3) ;  // 2、取出数据  print(n1) ;  }  public static void print(Node node) { // 取出节点内容  System.out.println(node.getData()) ;  if (node.getNext() != null) { // 后面还有货  print(node.getNext()) ; // 向下  }  }  } |

可以发现，整个一链表的关键是在于Node节点的关系匹配上。

**3.1.2、链表的基本雏形**

通过之前的分析，可以发现链表的最大作用的类就是Node，但是以上程序都是由用户自己去匹配节点关系的，但是这些节点的匹配工作不应该由用户完成，应该由一个程序专门负责。

那么专门负责这个节点操作的类，就称为链表类 —— Link，负责处理节点关系，而用户不需要关心节点问题，只需要关心Link的处理操作即可。

**范例：**开发Link类

|  |
| --- |
| class Node {  private String data ; // 假设要保存的数据类型是字符串  private Node next ;  public Node(String data) {  this.data = data ;  }  public String getData() {  return this.data ;  }  public void setNext(Node next) {  this.next = next ;  }  public Node getNext() {  return this.next ;  }  public void addNode(Node newNode) { // 操作的是节点关系  if (this.next == null) { // 当前节点的next为null  this.next = newNode ; // 保存新节点  } else {  this.next.addNode(newNode) ;  }  }  public void printNode() {  System.out.println(this.data) ;  if (this.next != null) { // 还有下一个节点  this.next.printNode() ;  }  }  }  class Link { // 处理节点关系  private Node root ; // 根节点  public void add(String data) { // 处理数据保存  if (data == null) { // 没有数据  return ; // 直接返回  }  // 每一个数据如果要想保存在链表之中，必须将其封装为节点  // 这一操作的过程外部（用户）不需要知道  Node newNode = new Node(data) ;  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else {  this.root.addNode(newNode) ;  }  }  public void print() {  if (this.root != null) { // 现在有根节点  this.root.printNode() ; // Node类处理  }  }  }  public class TestDemo {  public static void main(String args[]) {  Link all = new Link() ;  all.add("Hello") ;  all.add("World") ;  all.add("MLDN") ;  all.print() ;  }  } |

通过以上的代码可以发现，Link类处理节点的操作，而Node类负责节点的具体顺序的操作，但是客户端，不关心节点，只关心Link类即可。

### 3.1.3、开发可用链表

#### 3.1.3.1、增加数据：public String add(数据 对象)

通过上面的程序分析，可以发现，对于链表实现，Node类是整个操作的关键，但是首先来研究一下之前程序的问题：Node是一个单独的类，那么这样的类是可以被用户直接使用的，但是这个类由用户直接去使用，没有任何的意义，即：这个类有用，但是不能让用户去用，只能让Link类去使用，内部类完成。

|  |
| --- |
| class Link { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private String data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(String data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  }  private Node root ; // 根节点，第一个保存元素  public boolean add(String data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  return true ; // 增加成功  }  } |

使用内部类可以发现比之前的代码要节省一些，而且访问也方便了。

#### 3.1.3.2、增加多个数据：public boolean addAll(数据 对象 [] )

以上的操作是每次增加了一个对象，那么如果现在要求增加多个对象呢，例如：增加对象数组。可以采用循环数组的方式，每次调用add()方法。

|  |
| --- |
| public boolean addAll(String data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  } |

#### 3.1.3.3、统计数据个数：public int size()

在一个链表之中，会保存多个数据（每一个数据都被封装为Node类对象），那么要想取得这些保存元素的个数，可以增加一个size()方法完成。

应该在Link类之中增加一个统计的属性：count：

|  |
| --- |
| private int count ; // 统计个数 |

当用户每一次调用add()方法增加新数据的时候应该做出统计：

|  |
| --- |
| public boolean add(String data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  **this.count ++ ; // 保存数据量增加**  return true ; // 增加成功  } |

而在size()方法就是简单的将count这个变量的内容返回：

|  |
| --- |
| public int size() {  return this.count ;  } |

#### 3.1.3.4、判断是否是空链表：public boolean isEmpty()

所谓的空链表指的是链表之中不保存任何的数据，实际上这个null可以通过两种方式判断：一种判断链表的根节点是否为null，另外一个是判断保存元素的个数是否为0。

|  |
| --- |
| public boolean isEmpty() {  return this.count == 0 ;  } |

#### 3.1.3.5、查找数据是否存在：public boolean contains(数据 对象)

现在如果要想查询某个数据是否存在，那么基本的操作原理：逐个盘查，盘查的具体实现还是应该交给Node类去处理，但是在盘查之前必须有一个前提：有数据存在。

在Link类之中，增加查询的操作：

|  |
| --- |
| public boolean contains(String data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return **this.root.containsNode(data)** ; // 交给Node类处理  } |

在Node类之中，完成具体的查询，查询的流程：

· 判断当前节点的内容是否满足于查询内容，如果满足返回true；

· 如果当前节点的内容不满足，则向后继续查，如果已经没有后续节点了，则返回false。

|  |
| --- |
| public boolean containsNode(String data) { // 查找数据  if (data.equals(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  } |

#### 3.1.3.6、删除数据：public void remove(数据 对象)

对于链表之中的内容，之前完成的是增加操作和查询操作，但是从链表之中也会存在删除数据的操作，可是删除数据的操作需要分两种情况讨论：

· **情况一：**删除的数据不是根节点，使用：要删除节点的上一个节点.next = 要删除节点.next；

· **情况二：**删除的数据是根节点，根节点 = 根节点.next。

**范例：**修改Link类的删除操作

|  |
| --- |
| public void remove(String data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.count -- ; // 修改个数  } |

**范例：**修改Node类的删除操作

|  |
| --- |
| // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,String data) {  if (this.data.equals(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  } |

#### 3.1.3.7、取出全部数据：public 数据 [] toArray()

对于链表的这种数据结构，最为关键的是两个操作：删除、取得全部数据。

在Link类之中需要定义一个操作数组的脚标：

|  |
| --- |
| private int foot = 0 ; // 操作返回数组的脚标 |

要把数据保存的数组，Link类和Node类都需要使用，那么可以在Link类中定义返回数组，必须以属性的形式出现，只有这样，Node类才可以访问这个数组并进行操作。

|  |
| --- |
| private String [] retData ; // 返回数组 |

在Link类之中增加toArray()的方法。

|  |
| --- |
| public String [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  this.retData = new String [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  return this.retData ;  } |

修改Node类的操作，增加toArrayNode()方法。

|  |
| --- |
| public void toArrayNode() {  Link.this.retData[Link.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  } } |

不过，按照以上的方式进行开发，每一次调用toArray()方法，都要重复的进行数据的遍历，如果在数据没有修改的情况下，这种做法是一种非常差的做法，最好的做法是增加一个修改标记，如果发现数据增加了或删除的话，表示要重新遍历数据。

|  |
| --- |
| private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历 |

当增加或删除数据的时候，这个标记必须要进行修改。

|  |
| --- |
| public String [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new String [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ; } |

#### 3.1.3.8、根据索引位置取得数据：public 数据 get(int index)

在一个链表之中会有多个节点保存数据，现在要求可以取得指定节点位置上的数据。但是在进行这一操作的过程之中，有一个小问题：如果要取得数据的索引超过了数据的保存个数，那么是无法取得的。

在Link类之中，增加一个get()方法。

|  |
| --- |
| public String get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  } |

在Node类之中配置getNode()方法。

|  |
| --- |
| public String getNode(int index) {  if (Link.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  } |

#### 3.1.3.9、清空链表：public void clear()

所有的链表被root拽着，这个时候如果root为null，那么后面的数据都会断开，就表示都成了垃圾。

|  |
| --- |
| public void clear() {  this.root = null ;  this.count = 0 ;  } |

在整个程序之中，对于代码的开发部分不要求可以从无到有编写，但是需要清楚其操作原理，并且可以修改，而在整个链表工具类之中的方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **方法名称** | **类型** | **描述** |
| 1 | public String add(数据 对象) | 普通 | 向链表之中增加一个数据 |
| 2 | public boolean addAll(数据 对象 [] ) | 普通 | 向链表之中增加一组数据 |
| 3 | public int size() | 普通 | 取得链表之中的元素个数 |
| 4 | public boolean isEmpty() | 普通 | 判断链表是否为空 |
| 5 | public boolean contains(数据 对象) | 普通 | 从链表之中查询数据，依靠对象比较 |
| 6 | public void remove(数据 对象) | 普通 | 从链表中删除数据，依靠对象比较 |
| 7 | public 数据 [] toArray() | 普通 | 将链表之中保存的数据以对象数组的方式返回 |
| 8 | public 数据 get(int index) | 普通 | 根据索引位置取得指定的对象 |
| 9 | public void clear() | 普通 | 清空链表 |

以上的九个方法的作用，一定要清楚。

### 3.1.4、保存多个对象

之前的链表程序保存的都是String型数据，但是现在也可以是一个用户自己定义的类。

如果是自定义的类，那么一定要考虑到对象比较的方法，因为在链表之中的，contains()、remove()两个方法都调用了字符串内容的比较操作，而如果是一个自定义的Java类呢？需要编写compare()方法（对象比较）。

在之前学习了关系模型：dept – emp的操作，那么按照这个思路的话，链表可以针对于Emp使用，在一个部门有多个雇员，只要是多个就是对象数组，但是对象数组肯定以动态对象数组为主，那么一定就是链表的应用。

**3.2、第四个代码模型的练习（重点）**

重申：关于链表的掌握程度。

· 奢侈的想法：每个人可以自己独立的编写链表；

· 实际的想法：可以使用链表，修改链表为自己的类型（类型、对象比较操作）。

**3.2.1、第一道练习，一对多**

商品表：名称、价格、描述。

|  |
| --- |
| class Link { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private Product data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(Product data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  public boolean containsNode(Product data) { // 查找数据  if (data.compare(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  }  // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,Product data) {  if (this.data.compare(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  }  public void toArrayNode() {  Link.this.retData[Link.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  }  }  public Product getNode(int index) {  if (Link.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  }  }  private Node root ; // 根节点，第一个保存元素  private int count = 0 ; // 统计个数  private int foot = 0 ; // 操作返回数组的脚标  private Product [] retData ; // 返回数组  private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历  public boolean add(Product data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  this.count ++ ; // 保存数据量增加  this.changeFlag = true ; // 被修改了  return true ; // 增加成功  }  public boolean addAll(Product data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  }  public int size() {  return this.count ;  }  public boolean isProductty() {  return this.count == 0 ;  }  public boolean contains(Product data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return this.root.containsNode(data) ; // 交给Node类处理  }  public void remove(Product data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.changeFlag = true ; // 被修改了  this.count -- ; // 修改个数  }  public Product [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new Product [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ;  }  public Product get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  }  public void clear() {  this.root = null ;  this.count = 0 ;  }  }  class ProductGroup {  private int pgid ;  private String title ;  private String note ;  private Link products = new Link();  public ProductGroup() {}  public ProductGroup(int pgid,String title,String note) {  this.pgid = pgid ;  this.title = title ;  this.note = note ;  }  public void setProducts(Link products) {  this.products = products ;  }  public Link getProducts() {  return this.products ;  }  public String getProductGroupInfo() {  return "商品组编号：" + this.pgid + "，名称：" + this.title + "，描述：" + this.note ;  }  }  class Product {  private int pid ;  private String title ;  private double price ;  private String note ;  private ProductGroup group ;  public Product(){}  public Product(int pid,String title,double price,String note){  this.pid = pid ;  this.title = title ;  this.price = price ;  this.note = note ;  }  public boolean compare(Product product) {  if (this == product) {  return true ;  }  if (product == null) {  return false ;  }  if (this.pid == product.pid  && this.title.equals(product.title)  && this.price == product.price  && this.note.equals(product.note)) {  return true ;  }  return false ;  }  public void setGroup(ProductGroup group) {  this.group = group ;  }  public ProductGroup getGroup() {  return this.group ;  }  public String getProductInfo() {  return "商品编号：" + this.pid + "，名称：" + this.title + "，价格：" + this.price + "，描述：" + this.note ;  }  }  public class TestDemo {  public static void main(String args[]) {  // 一层设置关系  ProductGroup group = new ProductGroup(1,"生活用品","你懂的。。。。") ;  group.getProducts().add(new Product(10,"毛巾",1.0,"你懂的。")) ;  group.getProducts().add(new Product(11,"香皂",1.5,"你懂的。")) ;  group.getProducts().add(new Product(12,"牙刷",0.3,"你懂的。")) ;  System.out.println(group.getProductGroupInfo()) ;  Product prod [] = group.getProducts().toArray() ;  for (int x = 0 ; x < prod.length ; x ++) {  System.out.println(prod[x].getProductInfo()) ;  }  }  } |

**3.2.2、第二道练习，一对多**

要求1：根据一个用户找到他的所有定单；

要求2：根据一个定单找到其下单的用户；

|  |
| --- |
| class Link { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private Orders data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(Orders data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  public boolean containsNode(Orders data) { // 查找数据  if (data.compare(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  }  // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,Orders data) {  if (this.data.compare(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  }  public void toArrayNode() {  Link.this.retData[Link.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  }  }  public Orders getNode(int index) {  if (Link.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  }  }  private Node root ; // 根节点，第一个保存元素  private int count = 0 ; // 统计个数  private int foot = 0 ; // 操作返回数组的脚标  private Orders [] retData ; // 返回数组  private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历  public boolean add(Orders data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  this.count ++ ; // 保存数据量增加  this.changeFlag = true ; // 被修改了  return true ; // 增加成功  }  public boolean addAll(Orders data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  }  public int size() {  return this.count ;  }  public boolean isOrdersty() {  return this.count == 0 ;  }  public boolean contains(Orders data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return this.root.containsNode(data) ; // 交给Node类处理  }  public void remove(Orders data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.changeFlag = true ; // 被修改了  this.count -- ; // 修改个数  }  public Orders [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new Orders [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ;  }  public Orders get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  }  public void clear() {  this.root = null ;  this.count = 0 ;  }  }  class User {  private String userid ;  private String password ;  private String email ;  private String mobile ;  private int points ;  private Link orders = new Link() ;  public User() {}  public User(String userid,String password,String email,String mobile,int points) {  this.userid = userid ;  this.password = password ;  this.email = email ;  this.mobile = mobile ;  this.points = points ;  }  public void setOrders(Link orders) {  this.orders = orders ;  }  public Link getOrders() {  return this.orders ;  }  public String getUserInfo() {  return "用户ID：" + this.userid + "，密码：" + this.password + "，email：" + this.email + "，电话：" + this.mobile + "，积分：" + this.points ;  }  }  class Orders {  private int oid ;  private double allPrice ;  private User user ;  public Orders() {}  public Orders(int oid,double allPrice) {  this.oid = oid ;  this.allPrice = allPrice ;  }  public void setUser(User user) {  this.user = user ;  }  public User getUser() {  return this.user ;  }  public boolean compare(Orders orders) {  if (this == orders) {  return true ;  }  if (orders == null) {  return false ;  }  if (this.oid == orders.oid  && this.allPrice == orders.allPrice) {  return true ;  }  return false ;  }  public String getOrdersInfo() {  return "定单编号：" + this.oid + "，总价：" + this.allPrice ;  }  }  public class TestDemo {  public static void main(String args[]) {  User user = new User("mldn","nihao","mldnqa@163.com","110",10) ;  Orders o1 = new Orders(10,100.0) ;  Orders o2 = new Orders(11,200.0) ;  Orders o3 = new Orders(12,300.0) ;  user.getOrders().add(o1) ;  user.getOrders().add(o2) ;  user.getOrders().add(o3) ;  o1.setUser(user) ;  o2.setUser(user) ;  o3.setUser(user) ;  System.out.println(user.getUserInfo()) ;  System.out.println(user.getOrders().toArray()[0].getOrdersInfo()) ;  System.out.println(user.getOrders().toArray()[0].getUser().getUserInfo()) ;  }} |

**3.2.3、第三道练习，简单多对多（了解）**

多对多的映射关系在开发之中是存在两种的：

· 情况一：关系表之中，只存在关联字段，不存在任何的其他字段，留的题目属于简单多对多；

· 情况二：关系表之中，存在着其他的操作字段，数据模型：一个学生可以参加多门课程，每门课程可以有多个学生参加，每个学生针对于每门课程有一个成绩，在关系表中存在了一个成绩字段；

多对多 = 两个一对多。

**管理员-管理员组-权限，简单多对多；**

要求1：根据一个管理员的信息可以找到这个管理员所在的所有管理员组，并且列出每个管理员组的权限；

要求2：根据一个管理员组可以列出这个管理员组的权限，以及所有的管理员；

要求3：根据一个权限可以找到具备此权限的所有管理员组，并且列出每个管理员组的所有管理员。

|  |
| --- |
| class AdminLink { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private Admin data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(Admin data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  public boolean containsNode(Admin data) { // 查找数据  if (data.compare(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  }  // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,Admin data) {  if (this.data.compare(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  }  public void toArrayNode() {  AdminLink.this.retData[AdminLink.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  }  }  public Admin getNode(int index) {  if (AdminLink.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  }  }  private Node root ; // 根节点，第一个保存元素  private int count = 0 ; // 统计个数  private int foot = 0 ; // 操作返回数组的脚标  private Admin [] retData ; // 返回数组  private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历  public boolean add(Admin data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  this.count ++ ; // 保存数据量增加  this.changeFlag = true ; // 被修改了  return true ; // 增加成功  }  public boolean addAll(Admin data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  }  public int size() {  return this.count ;  }  public boolean isAdminty() {  return this.count == 0 ;  }  public boolean contains(Admin data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return this.root.containsNode(data) ; // 交给Node类处理  }  public void remove(Admin data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.changeFlag = true ; // 被修改了  this.count -- ; // 修改个数  }  public Admin [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new Admin [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ;  }  public Admin get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  }  public void clear() {  this.root = null ;  this.count = 0 ;  }  }  class GroupLink { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private Group data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(Group data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  public boolean containsNode(Group data) { // 查找数据  if (data.compare(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  }  // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,Group data) {  if (this.data.compare(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  }  public void toArrayNode() {  GroupLink.this.retData[GroupLink.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  }  }  public Group getNode(int index) {  if (GroupLink.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  }  }  private Node root ; // 根节点，第一个保存元素  private int count = 0 ; // 统计个数  private int foot = 0 ; // 操作返回数组的脚标  private Group [] retData ; // 返回数组  private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历  public boolean add(Group data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  this.count ++ ; // 保存数据量增加  this.changeFlag = true ; // 被修改了  return true ; // 增加成功  }  public boolean addAll(Group data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  }  public int size() {  return this.count ;  }  public boolean isGroupty() {  return this.count == 0 ;  }  public boolean contains(Group data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return this.root.containsNode(data) ; // 交给Node类处理  }  public void remove(Group data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.changeFlag = true ; // 被修改了  this.count -- ; // 修改个数  }  public Group [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new Group [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ;  }  public Group get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  }  public void clear() {  this.root = null ;  this.count = 0 ;  }  }  class PrivilegeLink { // 用户唯一关注的是此类  // 使用内部类的最大好处是可以和外部类进行私有操作的互相访问  private class Node { // 处理节点关系  private Privilege data ; // 要保存的数据  private Node next ; // 下一个节点  public Node(Privilege data){  this.data = data ;  }  public void addNode(Node newNode) { // 增加节点  if (this.next == null) { // 当前节点之后没有节点  this.next = newNode ; // 保存新节点  } else { // 当前节点之后有节点了  this.next.addNode(newNode) ; // 向下继续判断  }  }  public boolean containsNode(Privilege data) { // 查找数据  if (data.compare(this.data)) { // 与当前节点数据吻合  return true ;  } else { // 与当前节点数据不吻合  if (this.next != null) { // 还有下一个节点  return this.next.containsNode(data) ;  } else { // 没有后续节点  return false ; // 查找不到  }  }  }  // 传入两个参数：上一个节点，另外一个表示要删除的数据  public void removeNode(Node previous,Privilege data) {  if (this.data.compare(data)) { // 当前节点的数据吻合删除条件  previous.next = this.next ; // 空出当前节点  } else {  this.next.removeNode(this,data) ; // 向后继续删除  }  }  public void toArrayNode() {  PrivilegeLink.this.retData[PrivilegeLink.this.foot ++] = this.data ;  if (this.next != null) {  this.next.toArrayNode() ;  }  }  public Privilege getNode(int index) {  if (PrivilegeLink.this.foot ++ == index) { // 当前索引为查找数值  return this.data ;  } else {  return this.next.getNode(index) ;  }  }  }  private Node root ; // 根节点，第一个保存元素  private int count = 0 ; // 统计个数  private int foot = 0 ; // 操作返回数组的脚标  private Privilege [] retData ; // 返回数组  private boolean changeFlag = true ;  // changeFlag == true：数据被更改了，则需要重新遍历  // changeFlag == false：数据没有更改，不需要重新遍历  public boolean add(Privilege data) { // 增加数据  if (data == null) { // 如果保存的是一个空数据  return false ; // 增加失败  }  // 将数据封装为节点，目的：节点有next可以处理关系  Node newNode = new Node(data) ;  // 链表的关键就在于根节点  if (this.root == null) { // 现在没有根节点  this.root = newNode ; // 第一个作为根节点  } else { // 根节点有了，新的节点要保留在合适的位置  this.root.addNode(newNode) ; // Node类负责处理  }  this.count ++ ; // 保存数据量增加  this.changeFlag = true ; // 被修改了  return true ; // 增加成功  }  public boolean addAll(Privilege data[]) { // 一组数据  for (int x = 0 ; x < data.length ; x ++) {  if (!this.add(data[x])) { // 保存不成功  return false ;  }  }  return true ;  }  public int size() {  return this.count ;  }  public boolean isPrivilegety() {  return this.count == 0 ;  }  public boolean contains(Privilege data) { // 查找数据  // 根节点没有数据，查找的也没有数据  if (this.root == null || data == null) {  return false ; // 不需要进行查找了  }  return this.root.containsNode(data) ; // 交给Node类处理  }  public void remove(Privilege data) { // 要删除的节点  if (! this.contains(data)) { // 要删除的数据不存在  return ; // 直接返回被调用处，下面代码不执行了  }  if (data.equals(this.root.data)) { // 要删除的是根节点  this.root = this.root.next ; // 根节点的下一个  } else { // 要删除的不是根节点  this.root.next.removeNode(this.root,data) ;  }  this.changeFlag = true ; // 被修改了  this.count -- ; // 修改个数  }  public Privilege [] toArray() {  if (this.count == 0) {  return null ; // 没有数据  }  this.foot = 0 ; // 清零  if (this.changeFlag == true) { // 内容被修改了，需要重新取  this.retData = new Privilege [this.count] ; // 开辟数组大小  this.root.toArrayNode() ;  }  return this.retData ;  }  public Privilege get(int index) {  if (index > this.count) { // 超过个数  return null ; // 返回null  }  this.foot = 0 ; // 操作foot来定义脚标  return this.root.getNode(index) ;  }  public void clear() {  this.root = null ;  this.count = 0 ;  }  }  class Admin {  private String adminid ;  private String password ;  private GroupLink groups = new GroupLink() ;  public Admin() {}  public Admin(String adminid,String password) {  this.adminid = adminid ;  this.password = password ;  }  public GroupLink getGroups() {  return this.groups ;  }  public String getAdminInfo() {  return "管理员ID：" + this.adminid + "，密码：" + this.password ;  }  public boolean compare(Admin admin) {  if (this == admin) {  return true ;  }  if (admin == null) {  return false ;  }  if (this.adminid.equals(admin.adminid)  && this.password.equals(admin.password)) {  return true ;  }  return false ;  }  }  class Group {  private int groupid ;  private String title ;  private String note ;  private AdminLink admins = new AdminLink() ;  private PrivilegeLink privileges = new PrivilegeLink() ;  public Group() {}  public Group(int groupid,String title,String note) {  this.groupid = groupid ;  this.title = title ;  this.note = note ;  }  public AdminLink getAdmins() {  return this.admins ;  }  public PrivilegeLink getPrivileges() {  return this.privileges ;  }  public String getGroupInfo() {  return "管理员组ID：" + this.groupid + "，名称：" + this.title + "，描述：" + this.note ;  }  public boolean compare(Group group) {  if (this == group) {  return true ;  }  if (group == null) {  return false ;  }  if (this.groupid == group.groupid  && this.title.equals(group.title)  && this.note.equals(group.note)) {  return true ;  }  return false ;  }  }  class Privilege {  private int pid ;  private String title ;  private String note ;  private GroupLink groups = new GroupLink() ;  public Privilege() {}  public Privilege(int pid,String title,String note) {  this.pid = pid ;  this.title = title ;  this.note = note ;  }  public GroupLink getGroups() {  return this.groups ;  }  public String getPrivilegeInfo() {  return "权限ID：" + this.pid + "，名称：" + this.title + "，描述：" + this.note ;  }  public boolean compare(Privilege privilege) {  if (this == privilege) {  return true ;  }  if (privilege == null) {  return false ;  }  if (this.pid == privilege.pid  && this.title.equals(privilege.title)  && this.note.equals(privilege.note)) {  return true ;  }  return false ;  }  }  public class TestDemo {  public static void main(String args[]) {  // 一层配置关系  Admin adminA = new Admin("admininstrator","admin") ;  Admin adminB = new Admin("hello","hello") ;  Admin adminC = new Admin("jijiyiyi","jy") ;  Admin adminD = new Admin("yushi","yushi") ;  Admin adminE = new Admin("world","world") ;  Group groupA = new Group(10,"超级管理员组","你懂的。") ;  Group groupB = new Group(11,"系统维护管理员组","你懂的。") ;  Group groupC = new Group(12,"信息发布管理员组","你懂的。") ;  Privilege p1 = new Privilege(101,"增加管理员","-") ;  Privilege p2 = new Privilege(102,"增加用户","-") ;  Privilege p3 = new Privilege(103,"权限分配","-") ;  Privilege p4 = new Privilege(104,"增加新闻","-") ;  Privilege p5 = new Privilege(105,"生成统计报表","-") ;  Privilege p6 = new Privilege(106,"删除用户","-") ;  Privilege p7 = new Privilege(107,"评论维护","-") ;  Privilege p8 = new Privilege(108,"修改新闻","-") ;  Privilege p9 = new Privilege(109,"审核发布","-") ;  // 配置管理员和管理员组关系  adminA.getGroups().add(groupA) ;  adminA.getGroups().add(groupB) ;  adminB.getGroups().add(groupA) ;  adminB.getGroups().add(groupB) ;  adminB.getGroups().add(groupC) ;  adminC.getGroups().add(groupA) ;  adminC.getGroups().add(groupC) ;  adminD.getGroups().add(groupB) ;  adminE.getGroups().add(groupB) ;  groupA.getAdmins().add(adminA) ;  groupA.getAdmins().add(adminB) ;  groupA.getAdmins().add(adminC) ;  groupB.getAdmins().add(adminA) ;  groupB.getAdmins().add(adminB) ;  groupB.getAdmins().add(adminD) ;  groupB.getAdmins().add(adminE) ;  groupC.getAdmins().add(adminB) ;  groupC.getAdmins().add(adminC) ;  // 配置管理员组和权限  groupA.getPrivileges().add(p1) ;  groupA.getPrivileges().add(p2) ;  groupA.getPrivileges().add(p3) ;  groupA.getPrivileges().add(p4) ;  groupA.getPrivileges().add(p5) ;  groupB.getPrivileges().add(p1) ;  groupB.getPrivileges().add(p2) ;  groupB.getPrivileges().add(p3) ;  groupB.getPrivileges().add(p4) ;  groupB.getPrivileges().add(p5) ;  groupB.getPrivileges().add(p6) ;  groupB.getPrivileges().add(p7) ;  groupB.getPrivileges().add(p8) ;  groupB.getPrivileges().add(p9) ;  groupC.getPrivileges().add(p4) ;  groupC.getPrivileges().add(p5) ;  groupC.getPrivileges().add(p6) ;  groupC.getPrivileges().add(p7) ;  groupC.getPrivileges().add(p8) ;  groupC.getPrivileges().add(p9) ;  p1.getGroups().add(groupA) ;  p1.getGroups().add(groupB) ;  p2.getGroups().add(groupA) ;  p2.getGroups().add(groupB) ;  p3.getGroups().add(groupA) ;  p3.getGroups().add(groupB) ;  p4.getGroups().add(groupA) ;  p4.getGroups().add(groupB) ;  p4.getGroups().add(groupC) ;  p5.getGroups().add(groupA) ;  p5.getGroups().add(groupB) ;  p5.getGroups().add(groupC) ;  p6.getGroups().add(groupB) ;  p6.getGroups().add(groupC) ;  p7.getGroups().add(groupB) ;  p7.getGroups().add(groupC) ;  p8.getGroups().add(groupB) ;  p8.getGroups().add(groupC) ;  p9.getGroups().add(groupB) ;  p9.getGroups().add(groupC) ;  // 要求1：根据一个管理员的信息可以找到这个管理员所在的所有管理员组，并且列出每个管理员组的权限；  {  System.out.println(adminA.getAdminInfo()) ;  Group group [] = adminA.getGroups().toArray() ;  for (int x = 0 ; x < group.length ; x ++) {  System.out.println("\t〖管理员组〗" + group[x].getGroupInfo()) ;  Privilege pri [] = group[x].getPrivileges().toArray() ;  for (int y = 0 ; y < pri.length ; y ++) {  System.out.println("\t\t【权限】" + pri[y].getPrivilegeInfo()) ;  }  }  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*") ;  }  // 要求2：根据一个管理员组可以列出这个管理员组的权限，以及所有的管理员；  {  System.out.println(groupA.getGroupInfo()) ;  Admin admin [] = groupA.getAdmins().toArray() ;  for (int x = 0 ; x < admin.length ; x ++) {  System.out.println("\t〖管理员〗" + admin[x].getAdminInfo()) ;  }  Privilege pri [] = groupA.getPrivileges().toArray() ;  for (int x = 0 ; x < pri.length ; x ++) {  System.out.println("\t【权 限】" + pri[x].getPrivilegeInfo()) ;  }  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*") ;  }  // 要求3：根据一个权限可以找到具备此权限的所有管理员组，并且列出每个管理员组的所有管理员。  {  System.out.println(p3.getPrivilegeInfo()) ;  Group group [] = p3.getGroups().toArray() ;  for (int x = 0 ; x < group.length ; x ++) {  System.out.println("\t〖管理员组〗" + group[x].getGroupInfo()) ;  Admin admin [] = group[x].getAdmins().toArray() ;  for (int y = 0 ; y < admin.length ; y ++) {  System.out.println("\t\t【管理员】" + admin[y].getAdminInfo()) ;  }  }  }  }  } |

**4、总结**

1、 链表可以不会，但是必须会使；

2、 链表这个数据结构的增加、和输出操作必须会；

3、 数据模型的建立，第四个代码模型的完善。

**5、预习任务**

继承、final关键字、方法覆写、对象多态性、抽象类和接口。