2、具体内容

链表的本质是一个动态的对象数组，它可以实现若干个对象的存储。

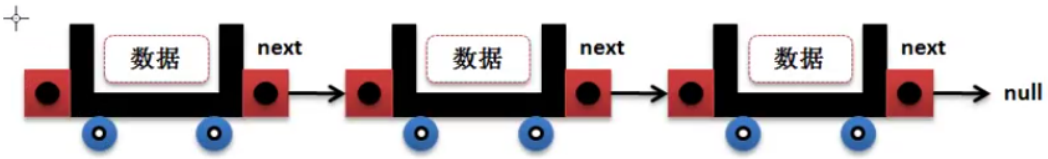
■链表的基本定义

在实际的开发之中对象数组是一项非常实用的技术，并且利用其可以描述出“多”方的概念，例如：一个人有多本书，则在人的类里面一定要提供有一个对象数组保存书的信息，但是传统的对象数组依赖于数组的概念，所以数组里面最大的缺点在于：长度是固定的，正是因为如此所以在实际的开发之中，传统的数组应用是非常有限的（数组的接收以及循环处理），但是如果想要进行灵活的数据保存，那么就必须自己来实现结构。

传统对象数组的开发操作依赖于脚标（索引）的控制，如果要想实现内容的动态维护，那么难度太高了，而且复杂度攀升，所以现在可以发现，对于一成不动的数据可以使用对象数组来实现，但是对于可能随时变化的数据就必须实现一个可以动态扩充的对象数组。

所谓的链表实质性的本质是利用引用的逻辑关系来实现类似于数组的数据处理操作，以一种保存“多”方数据的形式，实现数组类似的功能。

火车车厢



通过分析可以发现，如果要想实现链表处理，那么需要有一个公共的结构，这个结构可以实现数据的保存以及下一个连接的指向，为了描述这样的逻辑，可以把每一个存储理解为一个结点，所以此时应该准备出一个节点类，但是这个节点类里面可以保存各种数据类型的数据。

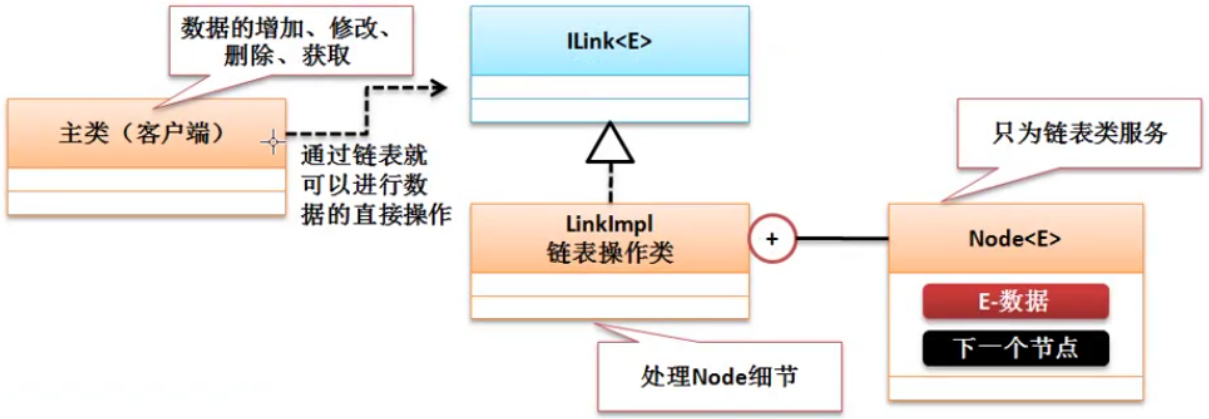
虽然已经清楚了需要通过Node结点来进行数据的保存，但是毕竟这里面需要牵扯到节点的引用处理关系，那么这个引用处理关系是由使用者控制吗？

范例：直接操作Node很麻烦

```
class Node<E> {
	private E data ;
	private Node next ;
	public Node(E data) {
		this.data = data ;
	}
	public E getData() {
		return this.data ;
	}
	public void setNext(Node<E> next) {
		this.next = next ;
	}
	public Node getNext() {
		return this.next ;
	}
}
public class LinkDemo {
	public static void main(String args[]) {
		Node<String> n1 = new Node<String>("火车头") ;
		Node<String> n2 = new Node<String>("车厢一") ;
		Node<String> n3 = new Node<String>("车厢二") ;
		Node<String> n4 = new Node<String>("车厢三") ;
		Node<String> n5 = new Node<String>("车厢四") ;
		n1.setNext(n2) ;
		n2.setNext(n3) ;
		n3.setNext(n4) ;
```

```
        n4.setNext(n5) ;
        print(n1) ;
    }
    public static void print(Node<?> node) {
        if (node != null) {    // 有节点
            System.out.println(node.getData()) ;
            print(node.getNext()) ;   // 递归调用
        }
    }
}
```

这样肯定不可能，所以应该有一个专门的类来进行节点的引用关系的配置。因为真实的使用者实际上关心的只是数据的存储与获取，所以现在应该对Node类进行包装处理。



■数据保存：public void add(E e)

通过之前的分析可以发现在进行链表操作的过程之中为了避免转型的异常应该使用泛型，同时也应该设计一个链表的标准接口，同时具体实现该接口的时候还应该通过Node类做出节点的关系描述。

范例：基本结构

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;
}
class LinkImpl<E> implements ILink<E> {
    private class Node {        // 保存节点的数据关系
        private E data ;  // 保存的数据
        private Node next ; // 保存下一个引用
        public Node(E data) {     // 有数据的情况下才有意义
            this.data = data ;
        }
    }
    // ---------------- 以下为Link类中定义的结构 ------------------
}
```

在现在所定义的Node类之中并没有出现有setter与getter方法，是因为内部类中的私有属性也方便外部类直接访问。

范例：实现数据增加

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;
}
class LinkImpl<E> implements ILink<E> {
    private class Node {        // 保存节点的数据关系
        private E data ;  // 保存的数据
        private Node next ; // 保存下一个引用
        public Node(E data) {     // 有数据的情况下才有意义
            this.data = data ;
        }
        // 第一次调用：this = LinkImpl.root;
        // 第二次调用：this = LinkImpl.root.next;
        // 第三次调用：this = LinkImpl.root.next.next;
        public void addNode(Node newNode) {        // 保存新的Node数据
            if (this.next == null) {      // 当前节点的下一个节点为null
                this.next = newNode ; // 保存当前节点
            } else {
                this.next.addNode(newNode) ;
            }
        }
    }
    // ---------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    // ---------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
    }
}
public class LinkDemo {
    public static void main(String args[]) {
        ILink<String> all = new LinkImpl<String>() ;
        all.add("Hello") ;
        all.add("World") ;
        all.add("MLDN") ;
    }
}
```

Link类这是负责数据的操作与根节点的处理，而所有后续节点的处理全部都是由Node类负责完成的。

■获取数据长度：public int size()

在链表之中往往需要保存有大量的数据，那么这些数据往往需要进行数据个数的统计操作，所以应该在LinkImpl子类里面追加有数据统计信息，同时当增加或删除数据时都应该对个数进行修改。

1、在ILink接口里面追加有一个获取数据个数的方法：

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;      // 增加数据
    public int size() ;    // 获取数据的个数
}
```

2、在LinkImpl子类里面追加有一个个数统计的属性

```
    private int count ; // 保存数据个数
```

3、在add()方法里面进行数据个数的追加：

```
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
}
```

4、需要在LinkImpl子类里面返回数据的个数

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

只是对于数据保存中的一个辅助功能。

完整代码

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;      // 增加数据
    public int size() ;    // 获取数据的个数
```

```
}
class LinkImpl<E> implements ILink<E> {
	private class Node {		// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {		// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
	}
	// --------------- 以下为Link类中定义的成员 ------------------
	private Node root ; // 保存根元素
	private int count ; // 保存数据个数
	// --------------- 以下为Link类中定义的方法 ------------------
	public void add(E e) {
		if (e == null) { // 保存的数据为null
			return ; // 方法调用直接结束
		}
		// 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据
包装在Node类之中
		Node newNode = new Node(e) ; // 创建一个新的节点
		if (this.root == null) {	// 现在没有根节点
			this.root = newNode ; // 第一个节点作为根节点
		} else {	// 根节点存在
			this.root.addNode(newNode) ;// 将新节点保存在合适的位置
		}
		this.count ++ ;
	}
	public int size() {
		return this.count ;
	}
}
public class LinkDemo {
	public static void main(String args[]) {
		ILink<String> all = new LinkImpl<String>() ;
		System.out.println("【增加之前】数据个数：" + all.size()) ;
		all.add("Hello") ;
		all.add("World") ;
		all.add("MLDN") ;
		System.out.println("【增加之后】数据个数：" + all.size()) ;
	}
}
```

■空集合判断：public boolean isEmpty()

链表里面可以保存有若干个数据，如果说现在链表还没有保存数据，这就表示时一个空集合，则应该提供有一个空的判断。

1、在ILink接口里面追加有判断方法：

```
public boolean isEmpty() ;  // 判断是否空集合
```

2、在LinkImpl子类里面覆写此方法：

```
public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
```

使用根节点或者是长度判断其本质是一样的。

完整代码

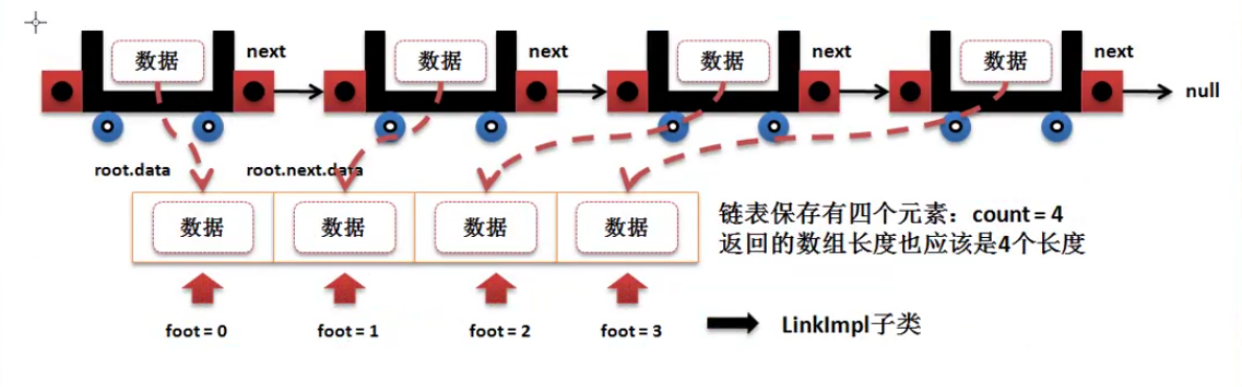
```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;      // 增加数据
    public int size() ;     // 获取数据的个数
    public boolean isEmpty() ;      // 判断是否空集合
}
class LinkImpl<E> implements ILink<E> {
    private class Node {        // 保存节点的数据关系
        private E data ;  // 保存的数据
        private Node next ; // 保存下一个引用
        public Node(E data) {     // 有数据的情况下才有意义
            this.data = data ;
        }
        // 第一次调用：this = LinkImpl.root;
        // 第二次调用：this = LinkImpl.root.next;
        // 第三次调用：this = LinkImpl.root.next.next;
        public void addNode(Node newNode) {       // 保存新的Node数据
            if (this.next == null) {      // 当前节点的下一个节点为null
                this.next = newNode ; // 保存当前节点
            } else {
                this.next.addNode(newNode) ;
            }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
```

```
            // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
            Node newNode = new Node(e) ; // 创建一个新的节点
            if (this.root == null) {      // 现在没有根节点
                this.root = newNode ; // 第一个节点作为根节点
            } else {     // 根节点存在
                this.root.addNode(newNode) ;// 将新节点保存在合适的位置
            }
            this.count ++ ;
        }
        public int size() {
            return this.count ;
        }
        public boolean isEmpty() {
            // return this.root == null ;
            return this.count == 0 ;
        }
}
public class LinkDemo {
        public static void main(String args[]) {
            ILink<String> all = new LinkImpl<String>() ;
            System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" + all.isEmpty()) ;
            all.add("Hello") ;
            all.add("World") ;
            all.add("MLDN") ;
            System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" + all.isEmpty()) ;
        }
}
```

■返回集合数据：public Object[] toArray()

链表本身就属于一个动态对象数组，那么既然是一个对象数组，就应该可以把所有的数据以数组的形式返回来，那么这个时候就可以定义一个toArray()方法，但是这个时候的方法只能够返回Object型的数组。

数据返回



1、在ILink接口里面追加新的处理方法：

```
public Object [] toArray() ; // 将集合元素以数组的形式返回
```

2、在LinkImpl子类里面追加有两个属性：

```
private int foot ; // 描述的是操作数组的脚标
private Object [] returnData ; // 返回的数据保存
```

3、在Node类中递归获取数据

```
        // 第一次调用：this = LinkImpl.root
        // 第二次调用：this = LinkImp.root.next
        // 第三次调用：this = LinkImp.root.next.next
        public void toArrayNode(){
            LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
            if (this.next != null) {       // 还有下一个数据
                this.next.toArrayNode() ;
            }
        }
    }
```

4、在进行数据返回的时候一定要首先判断是否为空集合：

```
    public Object[] toArray() {
        if (this.isEmpty())    {      // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
```

集合的数据一般如果要返回肯定要以对象数组的形式返回。

完整代码

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;      // 增加数据
    public int size() ;     // 获取数据的个数
    public boolean isEmpty() ;      // 判断是否空集合
    public Object [] toArray() ; // 将集合元素以数组的形式返回
}
```

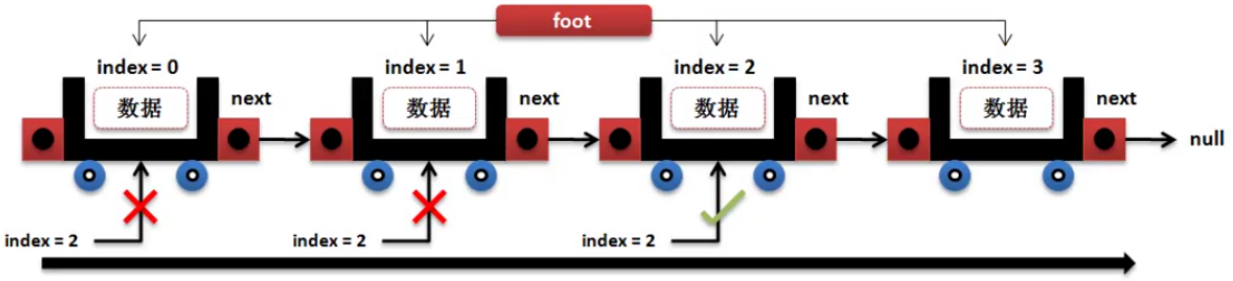
```
class LinkImpl<E> implements ILink<E> {
	private class Node {		// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {		// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
		// 第一次调用：this = LinkImpl.root
		// 第二次调用：this = LinkImp.root.next
		// 第三次调用：this = LinkImp.root.next.next
		public void toArrayNode(){
			LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
			if (this.next != null) {	// 还有下一个数据
				this.next.toArrayNode() ;
			}
		}
	}
	// --------------- 以下为Link类中定义的成员 ------------------
	private Node root ; // 保存根元素
	private int count ; // 保存数据个数
	private int foot ; // 描述的是操作数组的脚标
	private Object [] returnData ; // 返回的数据保存
	// --------------- 以下为Link类中定义的方法 ------------------
	public void add(E e) {
		if (e == null) { // 保存的数据为null
			return ; // 方法调用直接结束
		}
		// 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
		Node newNode = new Node(e) ; // 创建一个新的节点
		if (this.root == null) {	// 现在没有根节点
			this.root = newNode ; // 第一个节点作为根节点
		} else {	// 根节点存在
			this.root.addNode(newNode) ;// 将新节点保存在合适的位置
		}
		this.count ++ ;
	}
	public int size() {
		return this.count ;
	}
	public boolean isEmpty() {
		// return this.root == null ;
```

```
            return this.count == 0 ;
        }
        public Object[] toArray() {
            if (this.isEmpty())  {      // 空集合
                return null ; // 现在没有数据
            }
            this.foot = 0 ; // 脚标清零
            this.returnData = new Object [this.count] ;       // 根据已有的长度开辟数组
            this.root.toArrayNode() ;// 利用Node类进行递归数据获取
            return this.returnData ;
        }
}
public class LinkDemo {
        public static void main(String args[]) {
            ILink<String> all = new LinkImpl<String>() ;
            System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            all.add("Hello") ;
            all.add("World") ;
            all.add("MLDN") ;
            System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            Object result [] = all.toArray() ;
            for (Object obj : result) {
                System.out.println(obj) ;
            }
        }
}
```

■获取指定索引数据：public E get(int index)

链表可以像数组一样进行处理，所以也应该可以像数组一样进行索引数据的获取，在这样的情况下就可以继续利用递归的形式来完成。

根据索引获取数据



1、在ILink接口里面追加有新的方法：

```
    public E get(int index) ; // 根据索引获取数据
```

2、在Node类里面追加有根据索引获取数据的处理：

```
public E getNode(int index) {
	if (LinkImpl.this.foot ++ == index) {	// 索引相同
		return this.data ; // 返回当前数据
	} else {
		return this.next.getNode(index) ;
	}
}
```

3、在LinkImpl子类里面定义数据获取的实现：

```
public E get(int index) {
	if (index >= this.count) { // 索引应该在指定的范围之内
		return null ;
	}	// 索引数据的获取应该由Node类完成
	this.foot = 0 ; // 重置索引的下标
	return this.root.getNode(index) ;
}
```

这一特点和数组是很相似的，但是需要注意的是，数组获取一个数据的时间复杂度为1，而链表获取数据的时候复杂度为n。

完整代码

```
interface ILink<E> {		// 设置泛型避免安全隐患
	public void add(E e) ;	// 增加数据
	public int size() ;	// 获取数据的个数
	public boolean isEmpty() ;	// 判断是否空集合
	public Object [] toArray() ; // 将集合元素以数组的形式返回
	public E get(int index) ; // 根据索引获取数据
}
class LinkImpl<E> implements ILink<E> {
	private class Node {		// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {		// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
		// 第一次调用：this = LinkImpl.root
		// 第二次调用：this = LinkImp.root.next
		// 第三次调用：this = LinkImp.root.next.next
		public void toArrayNode(){
			LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
			if (this.next != null) {	// 还有下一个数据
```

```
                this.next.toArrayNode() ;
            }
        }
        public E getNode(int index) {
            if (LinkImpl.this.foot ++ == index) {   // 索引相同
                return this.data ; // 返回当前数据
            } else {
                return this.next.getNode(index) ;
            }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())    {      // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) {      // 索引应该在指定的范围之内
            return null ;
        }     // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
```

```
            return this.root.getNode(index) ;
        }
}
public class LinkDemo {
        public static void main(String args[]) {
                ILink<String> all = new LinkImpl<String>() ;
                System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
                all.add("Hello") ;
                all.add("World") ;
                all.add("MLDN") ;
                System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
                Object result [] = all.toArray() ;
                for (Object obj : result) {
                        System.out.println(obj) ;
                }
                System.out.println("----------- 数据获取的分割线 -----------") ;
                System.out.println(all.get(0)) ;
                System.out.println(all.get(1)) ;
                System.out.println(all.get(4)) ;
        }
}
```

■修改指定索引数据：public void set（int index,E data)

现在已经可以根据索引获取指定的数据了，但是既然可以获取数据，那么也可以进行数据的修改。

1、在ILink接口中追加有新的方法：

```
        public void set(int index,E data) ;// 修改索引数据
```

2、在Node类之中应该提供有数据修改的处理支持：

```
public void setNode(int index,E data) {
        if (LinkImpl.this.foot ++ == index) {       // 索引相同
                this.data = data ; // 修改数据
        } else {
                this.next.setNode(index,data) ;
        }
}
```

3、在LinkImpl子类里面进行方法覆写：

```
public void set(int index,E data) {
        if (index >= this.count) { // 索引应该在指定的范围之内
                return ;    // 方法结束
```

```
        }       // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        this.root.setNode(index,data) ;       // 修改数据
}
```

这种操作的时间复杂度也是n，因为依然需要进行数据的编历处理。

完整代码

```
interface ILink<E> {        // 设置泛型避免安全隐患
        public void add(E e) ;      // 增加数据
        public int size() ;     // 获取数据的个数
        public boolean isEmpty() ;      // 判断是否空集合
        public Object [] toArray() ; // 将集合元素以数组的形式返回
        public E get(int index) ; // 根据索引获取数据
        public void set(int index,E data) ;// 修改索引数据
}
class LinkImpl<E> implements ILink<E> {
        private class Node {        // 保存节点的数据关系
                private E data ;  // 保存的数据
                private Node next ; // 保存下一个引用
                public Node(E data) {     // 有数据的情况下才有意义
                        this.data = data ;
                }
                // 第一次调用：this = LinkImpl.root;
                // 第二次调用：this = LinkImpl.root.next;
                // 第三次调用：this = LinkImpl.root.next.next;
                public void addNode(Node newNode) {       // 保存新的Node数据
                        if (this.next == null) {      // 当前节点的下一个节点为null
                                this.next = newNode ; // 保存当前节点
                        } else {
                                this.next.addNode(newNode) ;
                        }
                }
                // 第一次调用：this = LinkImpl.root
                // 第二次调用：this = LinkImp.root.next
                // 第三次调用：this = LinkImp.root.next.next
                public void toArrayNode(){
                        LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
                        if (this.next != null) {       // 还有下一个数据
                                this.next.toArrayNode() ;
                        }
                }
                public E getNode(int index) {
                        if (LinkImpl.this.foot ++ == index) {       // 索引相同
                                return this.data ; // 返回当前数据
                        } else {
                                return this.next.getNode(index) ;
                        }
                }
                public void setNode(int index,E data) {
                        if (LinkImpl.this.foot ++ == index) {   // 索引相同
                                this.data = data ; // 修改数据
```

```
                } else {
                        this.next.setNode(index,data) ;
                }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())     {       // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return null ;
        }       // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        return this.root.getNode(index) ;
    }
    public void set(int index,E data) {
        if (index >= this.count) {      // 索引应该在指定的范围之内
            return ;   // 方法结束
        }      // 索引数据的获取应该由Node类完成
```

```
            this.foot = 0 ; // 重置索引的下标
            this.root.setNode(index,data) ;    // 修改数据
        }
}
public class LinkDemo {
        public static void main(String args[]) {
            ILink<String> all = new LinkImpl<String>() ;
            System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            all.add("Hello") ;
            all.add("World") ;
            all.add("MLDN") ;
            all.set(1,"世界") ;
            System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            Object result [] = all.toArray() ;
            for (Object obj : result) {
                System.out.println(obj) ;
            }
            System.out.println("----------- 数据获取的分割线 -----------") ;
            System.out.println(all.get(0)) ;
            System.out.println(all.get(1)) ;
            System.out.println(all.get(4)) ;
        }
}
```

■判断数据是否存在：public boolean contains(E data)

在一个集合里面往往会保存有大量的数据，有些时候需要判断某个数据是否存在，这个时候就可以通过对象比较的模式（equals()方法）来完成判断。

1、在ILink接口中追加判断的方法：

```
    public boolean contains(E data) ; // 判断数据是否存在
```

2、在Node类中进行依次判断：

```
public boolean containsNode(E data) {
        if (data.equals(this.data)) {      // 对象比较
                return true ;
        } else {
            if (this.next == null) {      // 没有后续节点了
                return false ;    // 找不到
            } else {
                return this.next.containsNode(data) ;     // 向后继续判断
            }
        }
```

```
}
```

3、在LinkImpl子类里面实现此方法

```
public boolean contains(E data) {
	if (data == null) {
		return false ;	// 没有数据
	}
	return this.root.containsNode(data) ;	// 交给Node类判断
}
```

由于整个链表没有null数据的存在，所以整体的程序在判断的时候直接使用每一个节点数据发出equals()方法调用即可。

完整代码

```
interface ILink<E> {		// 设置泛型避免安全隐患
	public void add(E e) ;	// 增加数据
	public int size() ;	// 获取数据的个数
	public boolean isEmpty() ;	// 判断是否空集合
	public Object [] toArray() ; // 将集合元素以数组的形式返回
	public E get(int index) ; // 根据索引获取数据
	public void set(int index,E data) ;// 修改索引数据
	public boolean contains(E data) ; // 判断数据是否存在
}
class LinkImpl<E> implements ILink<E> {
	private class Node {		// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {		// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
		// 第一次调用：this = LinkImpl.root
		// 第二次调用：this = LinkImp.root.next
		// 第三次调用：this = LinkImp.root.next.next
		public void toArrayNode(){
			LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
			if (this.next != null) {	// 还有下一个数据
				this.next.toArrayNode() ;
			}
		}
		public E getNode(int index) {
			if (LinkImpl.this.foot ++ == index) {	// 索引相同
				return this.data ; // 返回当前数据
```

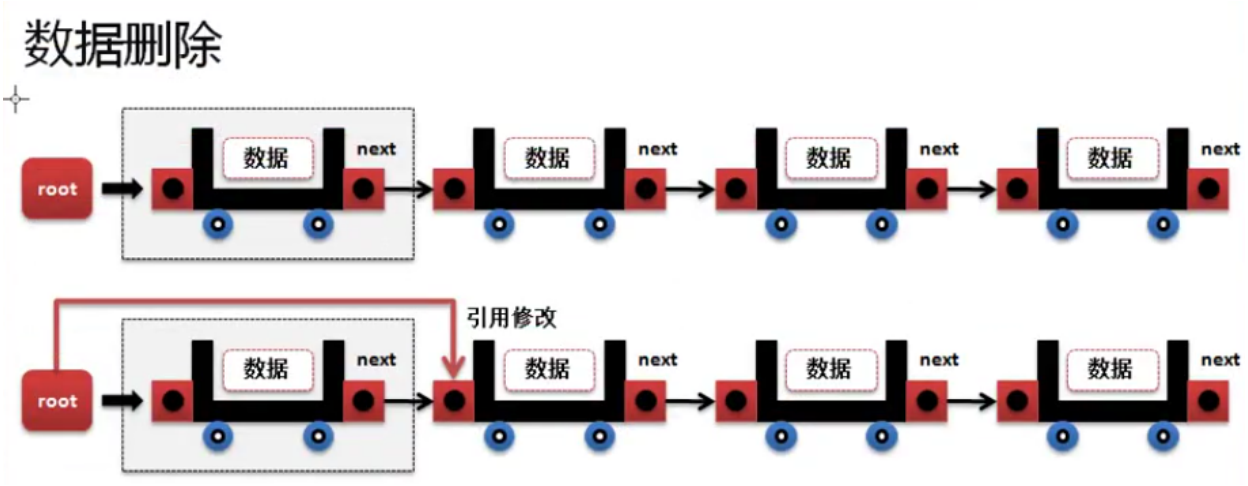
```
            } else {
                return this.next.getNode(index) ;
            }
        }
        public void setNode(int index,E data) {
            if (LinkImpl.this.foot ++ == index) {      // 索引相同
                this.data = data ; // 修改数据
            } else {
                this.next.setNode(index,data) ;
            }
        }
        public boolean containsNode(E data) {
            if (data.equals(this.data)) {   // 对象比较
                return true ;
            } else {
                if (this.next == null) {    // 没有后续节点了
                    return false ;  // 找不到
                } else {
                    return this.next.containsNode(data) ; // 向后继续判断
                }
            }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())     {       // 空集合
```

```
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;   // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return null ;
        }     // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        return this.root.getNode(index) ;
    }
    public void set(int index,E data) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return ;   // 方法结束
        }     // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        this.root.setNode(index,data) ;      // 修改数据
    }
    public boolean contains(E data) {
        if (data == null) {
            return false ;  // 没有数据
        }
        return this.root.containsNode(data) ; // 交给Node类判断
    }
}
public class LinkDemo {
    public static void main(String args[]) {
        ILink<String> all = new LinkImpl<String>() ;
        System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
        all.add("Hello") ;
        all.add("World") ;
        all.add("MLDN") ;
        all.set(1,"世界") ;
        System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
        Object result [] = all.toArray() ;
        for (Object obj : result) {
            System.out.println(obj) ;
        }
        System.out.println("----------- 数据获取的分割线 -----------") ;
        System.out.println(all.get(0)) ;
        System.out.println(all.get(1)) ;
        System.out.println(all.get(4)) ;
        System.out.println("----------- 数据判断的分割线 -----------") ;
        System.out.println(all.contains("高")) ;
        System.out.println(all.contains("Hello")) ;
    }
}
```

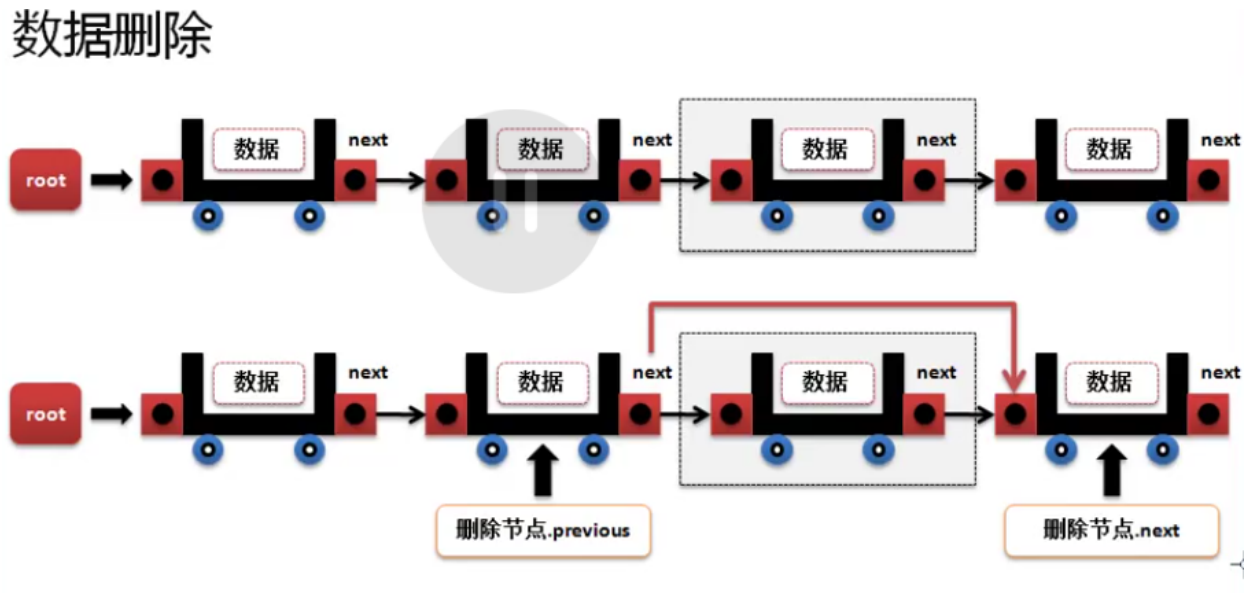
■数据删除：public void remove(E data)

数据的删除指的是可以从集合里面删除掉指定的一个数据内容，也就是说此时传递的是数据内容，那么如果要实现这种删除操作依然需要对象比较的支持，但是对于集合数据的删除需要考虑两种情况：

• 要删除的是根节点数据（LinkImpl与根节点有关，所以这个判断由根节点完成）：



• 要删除的不是根节点数据（有Node类负责）：



1、在ILink接口里面追加新的删除方法：

```
public void remove(E e) ;       // 数据删除
```

2、在LinkImpl子类里面实现根节点的判断：

```
public void remove(E data) {
    if (this.contains(data)) {   // 判断数据是否存在
        if (this.root.data.equals(data)) {      // 根节点为要删除节点
            this.root = this.root.next ; // 根的下一个节点
        } else {     // 交由Node类负责删除
```

```
                this.root.next.removeNode(this.root,data) ;
            }
            this.count -- ;
        }
}
```

3、如果现在根节点并不是要删除的节点，那么就需要进行后续节点判断，但是请一定要记住，此时根节点已经判断完成了，再判断应该从根节点的下一个开始判断，在Node类中追加删除处理：

```
public void removeNode(Node previous,E data) {
        if (this.data.equals(data)) {
                previous.next = this.next ; // 空出当前节点
        } else {
            if (this.next != null) {       // 有后续节点
                this.next.removeNode(this,data) ;  // 向后继续删除
            }
        }
}
```

4、完善LinkImpl子类中remove()方法：

```
public void remove(E data) {
        if (this.contains(data)) {   // 判断数据是否存在
            if (this.root.data.equals(data)) {      // 根节点为要删除节点
                this.root = this.root.next ; // 根的下一个节点
            } else {     // 交由Node类负责删除
                this.root.next.removeNode(this.root,data) ;
            }
            this.count -- ;
        }
}
```

删除逻辑依靠的就是引用的改变处理完成的。

完整代码

```
interface ILink<E> {        // 设置泛型避免安全隐患
        public void add(E e) ;      // 增加数据
        public int size() ;     // 获取数据的个数
        public boolean isEmpty() ;      // 判断是否空集合
        public Object [] toArray() ; // 将集合元素以数组的形式返回
        public E get(int index) ; // 根据索引获取数据
        public void set(int index,E data) ;// 修改索引数据
        public boolean contains(E data) ; // 判断数据是否存在
        public void remove(E e) ;      // 数据删除
}
class LinkImpl<E> implements ILink<E> {
        private class Node {        // 保存节点的数据关系
            private E data ;  // 保存的数据
            private Node next ; // 保存下一个引用
            public Node(E data) {     // 有数据的情况下才有意义
                this.data = data ;
            }
            // 第一次调用：this = LinkImpl.root;
```

```
// 第二次调用：this = LinkImpl.root.next;
// 第三次调用：this = LinkImpl.root.next.next;
public void addNode(Node newNode) {       // 保存新的Node数据
    if (this.next == null) {     // 当前节点的下一个节点为null
        this.next = newNode ; // 保存当前节点
    } else {
        this.next.addNode(newNode) ;
    }
}
// 第一次调用：this = LinkImpl.root
// 第二次调用：this = LinkImp.root.next
// 第三次调用：this = LinkImp.root.next.next
public void toArrayNode(){
    LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
    if (this.next != null) {       // 还有下一个数据
        this.next.toArrayNode() ;
    }
}
public E getNode(int index) {
    if (LinkImpl.this.foot ++ == index) {      // 索引相同
        return this.data ; // 返回当前数据
    } else {
        return this.next.getNode(index) ;
    }
}
public void setNode(int index,E data) {
    if (LinkImpl.this.foot ++ == index) {      // 索引相同
        this.data = data ; // 修改数据
    } else {
        this.next.setNode(index,data) ;
    }
}
public boolean containsNode(E data) {
    if (data.equals(this.data)) {     // 对象比较
        return true ;
    } else {
        if (this.next == null) {     // 没有后续节点了
            return false ;   // 找不到
        } else {
            return this.next.containsNode(data) ;    // 向后继续判断
        }
    }
}
public void removeNode(Node previous,E data) {
    if (this.data.equals(data)) {
        previous.next = this.next ; // 空出当前节点
    } else {
        if (this.next != null) {    // 有后续节点
            this.next.removeNode(this,data) ;      // 向后继续删除
        }
    }
}
```

```
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) {  // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())     {       // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return null ;
        }      // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        return this.root.getNode(index) ;
    }
    public void set(int index,E data) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return ;    // 方法结束
        }      // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        this.root.setNode(index,data) ;       // 修改数据
    }
    public boolean contains(E data) {
```

```
            if (data == null) {
                return false ;   // 没有数据
            }
            return this.root.containsNode(data) ;    // 交给Node类判断
        }
        public void remove(E data) {
            if (this.contains(data)) {       // 判断数据是否存在
                if (this.root.data.equals(data)) {  // 根节点为要删除节点
                    this.root = this.root.next ; // 根的下一个节点
                } else {    // 交由Node类负责删除
                    this.root.next.removeNode(this.root,data) ;
                }
                this.count -- ;
            }
        }
}
public class LinkDemo {
        public static void main(String args[]) {
            ILink<String> all = new LinkImpl<String>() ;
            System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            all.add("Hello") ;
            all.add("World") ;
            all.add("MLDN") ;
            all.remove("World") ;
            System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
            Object result [] = all.toArray() ;
            if (result != null) {
                for (Object obj : result) {
                    System.out.println(obj) ;
                }
            }
            System.out.println("----------- 数据获取的分割线 -----------") ;
            System.out.println(all.get(0)) ;
            System.out.println(all.get(1)) ;
            System.out.println(all.get(4)) ;
            System.out.println("----------- 数据判断的分割线 -----------") ;
            System.out.println(all.contains("高")) ;
            System.out.println(all.contains("Hello")) ;
        }
}
```

■清空链表：public void clean()

有些时候需要进行链表数据的整体清空处理，这个时候就可以直接根据根元素来进行控制，只要root设置为了null后续的节点就都不存在了。

1、在ILink接口里面追加有清空处理方法

```
public void clean() ;// 清空集合
```

2、在IinkImpl子类里面覆写方法

```
public void clean() {
    this.root = null ; // 后续的所有节点都没了
    this.count = 0 ; // 个数清零
}
```

这些就是链表的基本功能，当然，这只是一个最简单最基础的单向链表实现。

完整代码

```
    interface ILink<E> {        // 设置泛型避免安全隐患
public void add(E e) ;      // 增加数据
public int size() ;     // 获取数据的个数
public boolean isEmpty() ;      // 判断是否空集合
public Object [] toArray() ; // 将集合元素以数组的形式返回
public E get(int index) ; // 根据索引获取数据
public void set(int index,E data) ;// 修改索引数据
public boolean contains(E data) ; // 判断数据是否存在
public void remove(E e) ;       // 数据删除
public void clean() ;// 清空集合
}
class LinkImpl<E> implements ILink<E> {
public class Node {       // 保存节点的数据关系
    private E data ;  // 保存的数据
    private Node next ; // 保存下一个引用
    public Node(E data) {     // 有数据的情况下才有意义
        this.data = data ;
    }
    // 第一次调用：this = LinkImpl.root;
    // 第二次调用：this = LinkImpl.root.next;
    // 第三次调用：this = LinkImpl.root.next.next;
    public void addNode(Node newNode) {       // 保存新的Node数据
        if (this.next == null) {      // 当前节点的下一个节点为null
            this.next = newNode ; // 保存当前节点
        } else {
            this.next.addNode(newNode) ;
        }
    }
    // 第一次调用：this = LinkImpl.root
    // 第二次调用：this = LinkImp.root.next
    // 第三次调用：this = LinkImp.root.next.next
    public void toArrayNode(){
        LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
        if (this.next != null) {       // 还有下一个数据
            this.next.toArrayNode() ;
        }
    }
    public E getNode(int index) {
        if (LinkImpl.this.foot ++ == index) {       // 索引相同
```

```
                return this.data ; // 返回当前数据
            } else {
                return this.next.getNode(index) ;
            }
        }
        public void setNode(int index,E data) {
            if (LinkImpl.this.foot ++ == index) {       // 索引相同
                this.data = data ; // 修改数据
            } else {
                this.next.setNode(index,data) ;
            }
        }
        public boolean containsNode(E data) {
            if (data.equals(this.data)) {      // 对象比较
                return true ;
            } else {
                if (this.next == null) {      // 没有后续节点了
                    return false ;    // 找不到
                } else {
                    return this.next.containsNode(data) ;     // 向后继续判断
                }
            }
        }
        public void removeNode(Node previous,E data) {
            if (this.data.equals(data)) {
                previous.next = this.next ; // 空出当前节点
            } else {
                if (this.next != null) {       // 有后续节点
                    this.next.removeNode(this,data) ;  // 向后继续删除
                }
            }
        }
    }
    // ---------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // ---------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
```

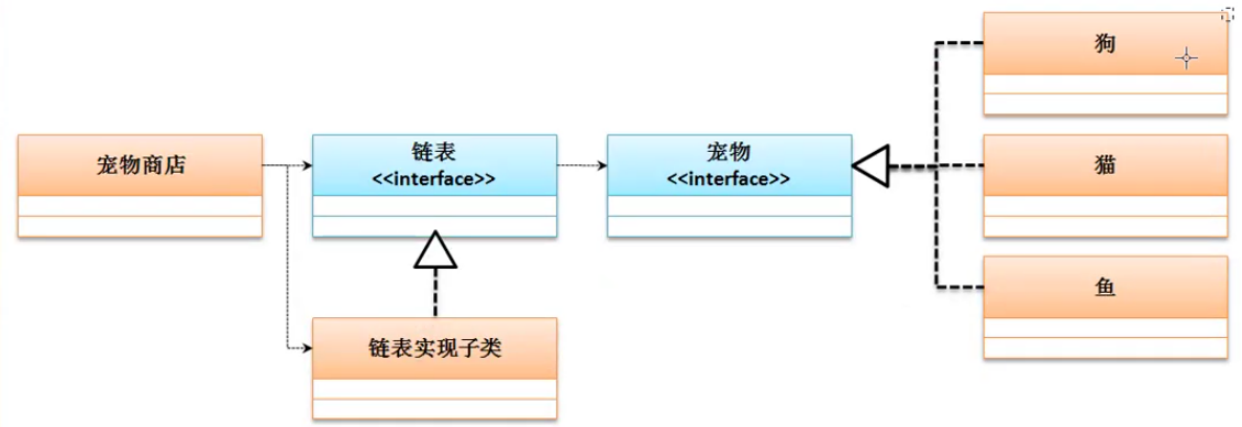
```
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())    {      // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return null ;
        }      // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        return this.root.getNode(index) ;
    }
    public void set(int index,E data) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return ;    // 方法结束
        }      // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        this.root.setNode(index,data) ;       // 修改数据
    }
    public boolean contains(E data) {
        if (data == null) {
            return false ;    // 没有数据
        }
        return this.root.containsNode(data) ;     // 交给Node类判断
    }
    public void remove(E data) {
        if (this.contains(data)) {   // 判断数据是否存在
            if (this.root.data.equals(data)) {      // 根节点为要删除节点
                this.root = this.root.next ; // 根的下一个节点
            } else {     // 交由Node类负责删除
                this.root.next.removeNode(this.root,data) ;
            }
            this.count -- ;
        }
    }
    public void clean() {
        this.root = null ; // 后续的所有节点都没了
        this.count = 0 ; // 个数清零
    }
}
```

```
public class LinkDemo {
	public static void main(String args[]) {
		ILink<String> all = new LinkImpl<String>() ;
		System.out.println("【增加之前】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
		all.add("Hello") ;
		all.add("World") ;
		all.add("MLDN") ;
		all.remove("World") ;
		all.clean();
		System.out.println("【增加之后】数据个数：" + all.size() + "、是否为空集合：" +
all.isEmpty()) ;
		Object result [] = all.toArray() ;
		if (result != null) {
			for (Object obj : result) {
				System.out.println(obj) ;
			}
		}
		/*
		System.out.println("----------- 数据获取的分割线 -----------") ;
		System.out.println(all.get(0)) ;
		System.out.println(all.get(1)) ;
		System.out.println(all.get(4)) ;
		System.out.println("----------- 数据判断的分割线 -----------") ;
		System.out.println(all.contains("高")) ;
		System.out.println(all.contains("Hello")) ;
		*/
	}
}
```

2、具体内容

现在假设有一个宠物商店，里面可以出售各种宠物，要求可以实现宠物的上架处理、下架处理，也可以根据关键字查询宠物的信息。

设计案例



1、应该定义出宠物的标准：

```
interface Pet { // 定义宠物标准
	public String getName() ;	// 获得名字
	public String getColor() ;	// 获得颜色
}
```

2、宠物商店应该以宠物的标准为主：

```
class PetShop { // 宠物商店
	private ILink<Pet> allPets = new LinkImpl<Pet>() ; // 保存多个宠物信息
	public void add(Pet pet) {	// 追加宠物，商品上架
		this.allPets.add(pet) ;	// 集合中保存对象
	}
	public void delete(Pet pet) {
		this.allPets.remove(pet) ;
	}
	public ILink<Pet> search(String keyword) {
		ILink<Pet> searchResult = new LinkImpl<Pet>() ; // 保存查询结果
		Object result [] = this.allPets.toArray() ; // 获取全部数据
		if (result != null) {
			for (Object obj : result) {
				Pet pet = (Pet) obj ;
				if (pet.getName().contains(keyword) ||
					pet.getColor().contains(keyword)) {
					searchResult.add(pet) ; // 保存查询结果
				}
			}
		}
		return searchResult ;
	}
}
```

3、根据宠物的标准来定义宠物信息：

定义宠物猫：	**定义宠物狗：**

class Cat implements Pet { // 实现宠物标准
 private String name ;
 private String color ;
 public Cat(String name,String color) {
	class Dog implements Pet { // 实现宠物标准
	 private String name ;
	 private String color ;
	 public Dog(String name,String color) {

```
            this.name = name ;
            this.color = color ;
        }
        public String getName() {
            return this.name ;
        }
        public String getColor() {
            return this.color ;
        }
        public boolean equals(Object obj) {
            if (obj == null) {
                return false ;
            }
            if (!(obj instanceof Cat)) {
                return false ;
            }
            if (this == obj) {
                return true ;
            }
            Cat cat = (Cat) obj ;
            return this.name.equals(cat.name)
&& this.color.equals(cat.color) ;
        }
        public String toString() {
            return "【宠物猫】名字：" +
this.name + "、颜色：" + this.color ;
        }
}
```

```
            this.name = name ;
            this.color = color ;
        }
        public String getName() {
            return this.name ;
        }
        public String getColor() {
            return this.color ;
        }
        public boolean equals(Object obj) {
            if (obj == null) {
                return false ;
            }
            if (!(obj instanceof Dog)) {
                return false ;
            }
            if (this == obj) {
                return true ;
            }
            Dog dog = (Dog) obj ;
            return this.name.equals(dog.name)
&& this.color.equals(dog.color) ;
        }
        public String toString() {
            return "【宠物狗】名字：" +
this.name + "、颜色：" + this.color ;
        }
}
```

4、实现宠物商店的操作

```
public class JavaDemo {
    public static void main(String args[]) {
        PetShop shop = new PetShop() ;   // 开店
        shop.add(new Dog("黄斑狗","绿色")) ;
        shop.add(new Cat("小强猫","深绿色")) ;
        shop.add(new Cat("黄猫","深色")) ;
        shop.add(new Dog("黄狗","黄色")) ;
        shop.add(new Dog("斑点狗","灰色")) ;
        Object result [] = shop.search("黄").toArray() ;
        for (Object obj : result) {
            System.out.println(obj) ;
        }
    }
}
```

所有的程序开发都是以接口为标准进行的，这样在进行后期程序处理的时候可以非常灵活，只要符合标准的对象都可以保存

完整代码

```
interface ILink<E> {        // 设置泛型避免安全隐患
    public void add(E e) ;      // 增加数据
    public int size() ;     // 获取数据的个数
```

```
	public boolean isEmpty() ;	// 判断是否空集合
	public Object [] toArray() ; // 将集合元素以数组的形式返回
	public E get(int index) ; // 根据索引获取数据
	public void set(int index,E data) ;// 修改索引数据
	public boolean contains(E data) ; // 判断数据是否存在
	public void remove(E e) ;	// 数据删除
	public void clean() ; // 清空集合
}
class LinkImpl<E> implements ILink<E> {
	private class Node {	// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {	// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
		// 第一次调用：this = LinkImpl.root
		// 第二次调用：this = LinkImp.root.next
		// 第三次调用：this = LinkImp.root.next.next
		public void toArrayNode(){
			LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
			if (this.next != null) {	// 还有下一个数据
				this.next.toArrayNode() ;
			}
		}
		public E getNode(int index) {
			if (LinkImpl.this.foot ++ == index) {	// 索引相同
				return this.data ; // 返回当前数据
			} else {
				return this.next.getNode(index) ;
			}
		}
		public void setNode(int index,E data) {
			if (LinkImpl.this.foot ++ == index) {	// 索引相同
				this.data = data ; // 修改数据
			} else {
				this.next.setNode(index,data) ;
			}
		}
		public boolean containsNode(E data) {
			if (data.equals(this.data)) {	// 对象比较
				return true ;
			} else {
```

```
                if (this.next == null) {      // 没有后续节点了
                    return false ;    // 找不到
                } else {
                    return this.next.containsNode(data) ;     // 向后继续判断
                }
            }
        }
        public void removeNode(Node previous,E data) {
            if (this.data.equals(data)) {
                previous.next = this.next ; // 空出当前节点
            } else {
                if (this.next != null) {       // 有后续节点
                    this.next.removeNode(this,data) ;  // 向后继续删除
                }
            }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())     {      // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
```

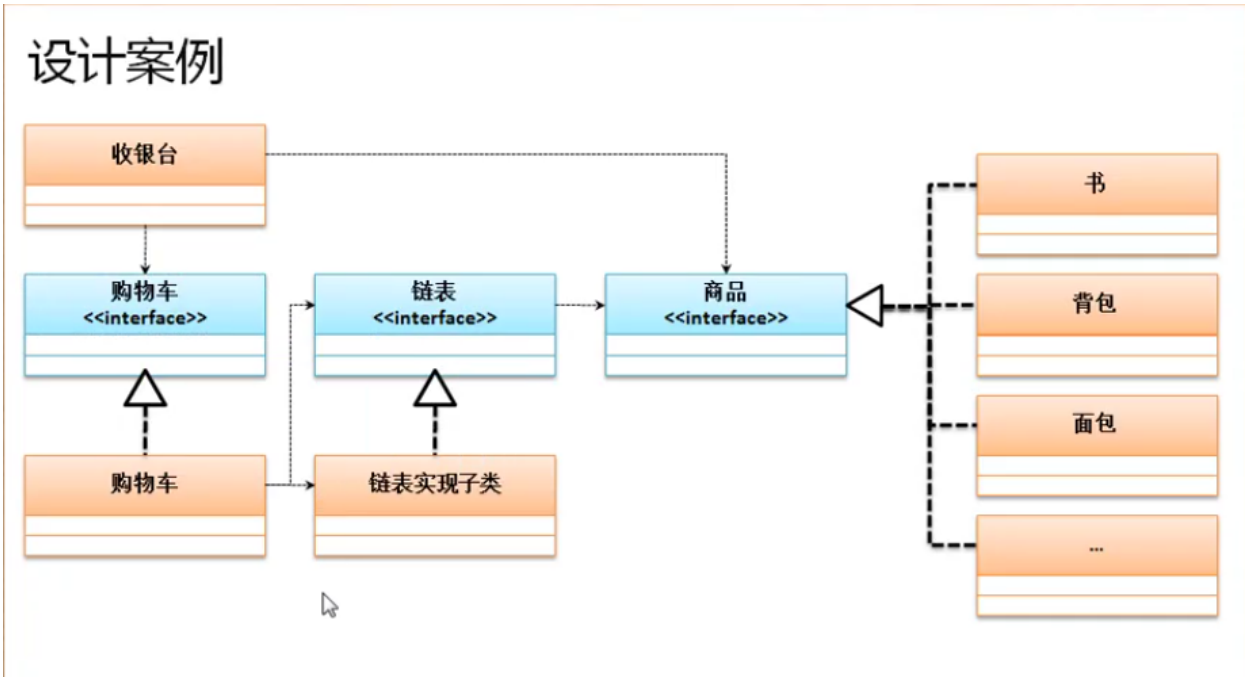
```
	}
	public E get(int index) {
		if (index >= this.count) { // 索引应该在指定的范围之内
			return null ;
		}	// 索引数据的获取应该由Node类完成
		this.foot = 0 ; // 重置索引的下标
		return this.root.getNode(index) ;
	}
	public void set(int index,E data) {
		if (index >= this.count) { // 索引应该在指定的范围之内
			return ;	// 方法结束
		}	// 索引数据的获取应该由Node类完成
		this.foot = 0 ; // 重置索引的下标
		this.root.setNode(index,data) ;		// 修改数据
	}
	public boolean contains(E data) {
		if (data == null) {
			return false ;	// 没有数据
		}
		return this.root.containsNode(data) ;	// 交给Node类判断
	}
	public void remove(E data) {
		if (this.contains(data)) {	// 判断数据是否存在
			if (this.root.data.equals(data)) {		// 根节点为要删除节点
				this.root = this.root.next ; // 根的下一个节点
			} else {	// 交由Node类负责删除
				this.root.next.removeNode(this.root,data) ;
			}
			this.count -- ;
		}
	}
	public void clean() {
		this.root = null ; // 后续的所有节点都没了
		this.count = 0 ; // 个数清零
	}
}
interface Pet { // 定义宠物标准
	public String getName() ;		// 获得名字
	public String getColor() ;		// 获得颜色
}
class PetShop { // 宠物商店
	private ILink<Pet> allPets = new LinkImpl<Pet>() ; // 保存多个宠物信息
	public void add(Pet pet) {		// 追加宠物，商品上架
		this.allPets.add(pet) ;		// 集合中保存对象
	}
	public void delete(Pet pet) {
		this.allPets.remove(pet) ;
	}
	public ILink<Pet> search(String keyword) {
		ILink<Pet> searchResult = new LinkImpl<Pet>() ; // 保存查询结果
		Object result [] = this.allPets.toArray() ; // 获取全部数据
		if (result != null) {
```

```
                for (Object obj : result) {
                    Pet pet = (Pet) obj ;
                    if (pet.getName().contains(keyword) ||
                        pet.getColor().contains(keyword)) {
                        searchResult.add(pet) ; // 保存查询结果
                    }
                }
            }
            return searchResult ;
        }
}
class Cat implements Pet { // 实现宠物标准
        private String name ;
        private String color ;
        public Cat(String name,String color) {
            this.name = name ;
            this.color = color ;
        }
        public String getName() {
            return this.name ;
        }
        public String getColor() {
            return this.color ;
        }
        public boolean equals(Object obj) {
            if (obj == null) {
                return false ;
            }
            if (!(obj instanceof Cat)) {
                return false ;
            }
            if (this == obj) {
                return true ;
            }
            Cat cat = (Cat) obj ;
            return this.name.equals(cat.name) && this.color.equals(cat.color) ;
        }
        public String toString() {
            return "【宠物猫】名字：" + this.name + "、颜色：" + this.color ;
        }
}
class Dog implements Pet { // 实现宠物标准
        private String name ;
        private String color ;
        public Dog(String name,String color) {
            this.name = name ;
            this.color = color ;
        }
        public String getName() {
            return this.name ;
        }
        public String getColor() {
```

```
            return this.color ;
        }
        public boolean equals(Object obj) {
            if (obj == null) {
                return false ;
            }
            if (!(obj instanceof Dog)) {
                return false ;
            }
            if (this == obj) {
                return true ;
            }
            Dog dog = (Dog) obj ;
            return this.name.equals(dog.name) && this.color.equals(dog.color) ;
        }
        public String toString() {
            return "【宠物狗】名字：" + this.name + "、颜色：" + this.color ;
        }
}
public class JavaDemo {
        public static void main(String args[]) {
            PetShop shop = new PetShop() ;   // 开店
            shop.add(new Dog("黄斑狗","绿色")) ;
            shop.add(new Cat("小强猫","深绿色")) ;
            shop.add(new Cat("黄猫","深色")) ;
            shop.add(new Dog("黄狗","黄色")) ;
            shop.add(new Dog("斑点狗","灰色")) ;
            Object result [] = shop.search("黄").toArray() ;
            for (Object obj : result) {
                System.out.println(obj) ;
            }
        }
}
```

■综合实战：超市购物车

使用面向对象的概念表示出下面的生活场景：小明去超市买东西，所有买到的东西都放在了购物车之中，最后到收银台一起结账。



1、定义出一个商品的标准：

```
interface IGoods { // 定义商品标准
	public String getName() ;
	public double getPrice() ;
}
```

2、定义购物车处理标准：

```
interface IShopCar {	// 购物车
	public void add(IGoods goods) ; // 添加商品信息
	public void delete(IGoods goods) ; // 删除商品
	public Object[] getAll() ; // 获得购物车中的全部商品信息
}
```

3、定义一个购物车的实现类

```
class ShopCarImpl implements IShopCar { // 购物车
	private ILink<IGoods> allGoodses = new LinkImpl<IGoods>() ;
	public void add(IGoods goods) {
		this.allGoodses.add(goods) ;
	}
	public void delete(IGoods goods) {
		this.allGoodses.remove(goods) ;
	}
	public Object [] getAll() {
		return this.allGoodses.toArray() ;
	}
}
```

4、定义收银台

```
class Cashier { // 收银台
	private IShopCar shopcar ;
	public Cashier(IShopCar shopcar) {
		this.shopcar = shopcar ;
	}
	public double allPrice() { // 计算总价
```

```
            double all = 0.0 ;
            Object result [] = this.shopcar.getAll() ;
            for (Object obj : result) {
                  IGoods goods = (IGoods) obj ;
                  all += goods.getPrice() ;
            }
            return all ;
      }
      public int allCount() {      // 商品数量
            return this.shopcar.getAll().length ;
      }
}
```

5、定义商品信息

图书：

```
class Book implements IGoods {
      private String name ;
      private double price ;
      public Book(String name,double price) {
            this.name = name ;
            this.price = price ;
      }
      public String getName() {
            return this.name ;
      }
      public double getPrice() {
            return this.price ;
      }
      public boolean equals(Object obj) {
            if (obj == null) {
                  return false ;
            }
            if (this == obj) {
                  return true ;
            }
            if (!(obj instanceof Book)) {
                  return false ;
            }
            Book book = (Book) obj ;
            return this.name.equals(book.name)
&& this.price == book.price ;
      }
      public String toString() {
            return "【图书信息】名称：" +
this.name + "、价格：" + this.price ;
      }
}
```

书包：

```
class Bag implements IGoods {
      private String name ;
      private double price ;
      public Bag(String name,double price)
{
            this.name = name ;
            this.price = price ;
      }
      public String getName() {
            return this.name ;
      }
      public double getPrice() {
            return this.price ;
      }
      public boolean equals(Object obj) {
            if (obj == null) {
                  return false ;
            }
            if (this == obj) {
                  return true ;
            }
            if (!(obj instanceof Bag)) {
                  return false ;
            }
            Bag bag = (Bag) obj ;
            return
this.name.equals(bag.name) && this.price
== bag.price ;
      }
      public String toString() {
            return "【背包信息】名称：" +
this.name + "、价格：" + this.price ;
      }
}
```

6、进行代码测试的编写

```
public class JavaDemo {
	public static void main(String args[]) {
		IShopCar car = new ShopCarImpl() ;
		car.add(new Book("Java开发",79.8)) ;
		car.add(new Book("Oracle",89.8)) ;
		car.add(new Bag("小强背包",889.8)) ;
		Cashier cas = new Cashier(car) ;
		System.out.println("总价格：" + cas.allPrice() + "、购买总数量：" + cas.allCount()) ;
	}
}
```

整体的代码都是基于链表的功能实现的。

完整代码

```
interface ILink<E> {		// 设置泛型避免安全隐患
	public void add(E e) ;	// 增加数据
	public int size() ;	// 获取数据的个数
	public boolean isEmpty() ;	// 判断是否空集合
	public Object [] toArray() ; // 将集合元素以数组的形式返回
	public E get(int index) ; // 根据索引获取数据
	public void set(int index,E data) ;// 修改索引数据
	public boolean contains(E data) ; // 判断数据是否存在
	public void remove(E e) ;		// 数据删除
	public void clean() ; // 清空集合
}
class LinkImpl<E> implements ILink<E> {
	private class Node {		// 保存节点的数据关系
		private E data ;  // 保存的数据
		private Node next ; // 保存下一个引用
		public Node(E data) {	// 有数据的情况下才有意义
			this.data = data ;
		}
		// 第一次调用：this = LinkImpl.root;
		// 第二次调用：this = LinkImpl.root.next;
		// 第三次调用：this = LinkImpl.root.next.next;
		public void addNode(Node newNode) {		// 保存新的Node数据
			if (this.next == null) {	// 当前节点的下一个节点为null
				this.next = newNode ; // 保存当前节点
			} else {
				this.next.addNode(newNode) ;
			}
		}
		// 第一次调用：this = LinkImpl.root
		// 第二次调用：this = LinkImp.root.next
		// 第三次调用：this = LinkImp.root.next.next
		public void toArrayNode(){
			LinkImpl.this.returnData [LinkImpl.this.foot ++] = this.data ;
			if (this.next != null) {		// 还有下一个数据
				this.next.toArrayNode() ;
			}
		}
		public E getNode(int index) {
```

```
            if (LinkImpl.this.foot ++ == index) {      // 索引相同
                return this.data ; // 返回当前数据
            } else {
                return this.next.getNode(index) ;
            }
        }
        public void setNode(int index,E data) {
            if (LinkImpl.this.foot ++ == index) {      // 索引相同
                this.data = data ; // 修改数据
            } else {
                this.next.setNode(index,data) ;
            }
        }
        public boolean containsNode(E data) {
            if (data.equals(this.data)) {      // 对象比较
                return true ;
            } else {
                if (this.next == null) {      // 没有后续节点了
                    return false ;    // 找不到
                } else {
                    return this.next.containsNode(data) ;     // 向后继续判断
                }
            }
        }
        public void removeNode(Node previous,E data) {
            if (this.data.equals(data)) {
                previous.next = this.next ; // 空出当前节点
            } else {
                if (this.next != null) {       // 有后续节点
                    this.next.removeNode(this,data) ;  // 向后继续删除
                }
            }
        }
    }
    // --------------- 以下为Link类中定义的成员 ------------------
    private Node root ; // 保存根元素
    private int count ; // 保存数据个数
    private int foot ; // 描述的是操作数组的脚标
    private Object [] returnData ; // 返回的数据保存
    // --------------- 以下为Link类中定义的方法 ------------------
    public void add(E e) {
        if (e == null) { // 保存的数据为null
            return ; // 方法调用直接结束
        }
        // 数据本身是不具有关联特性的，只有Node类有，那么要想实现关联处理就必须将数据包装在Node类之中
        Node newNode = new Node(e) ; // 创建一个新的节点
        if (this.root == null) {      // 现在没有根节点
            this.root = newNode ; // 第一个节点作为根节点
        } else {     // 根节点存在
            this.root.addNode(newNode) ;// 将新节点保存在合适的位置
        }
```

```
        this.count ++ ;
    }
    public int size() {
        return this.count ;
    }
    public boolean isEmpty() {
        // return this.root == null ;
        return this.count == 0 ;
    }
    public Object[] toArray() {
        if (this.isEmpty())    {       // 空集合
            return null ; // 现在没有数据
        }
        this.foot = 0 ; // 脚标清零
        this.returnData = new Object [this.count] ;    // 根据已有的长度开辟数组
        this.root.toArrayNode() ;// 利用Node类进行递归数据获取
        return this.returnData ;
    }
    public E get(int index) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return null ;
        }       // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        return this.root.getNode(index) ;
    }
    public void set(int index,E data) {
        if (index >= this.count) { // 索引应该在指定的范围之内
            return ;    // 方法结束
        }       // 索引数据的获取应该由Node类完成
        this.foot = 0 ; // 重置索引的下标
        this.root.setNode(index,data) ;       // 修改数据
    }
    public boolean contains(E data) {
        if (data == null) {
            return false ;    // 没有数据
        }
        return this.root.containsNode(data) ;     // 交给Node类判断
    }
    public void remove(E data) {
        if (this.contains(data)) {   // 判断数据是否存在
            if (this.root.data.equals(data)) {      // 根节点为要删除节点
                this.root = this.root.next ; // 根的下一个节点
            } else {     // 交由Node类负责删除
                this.root.next.removeNode(this.root,data) ;
            }
            this.count -- ;
        }
    }
    public void clean() {
        this.root = null ; // 后续的所有节点都没了
        this.count = 0 ; // 个数清零
    }
```

```
}
interface IGoods { // 定义商品标准
	public String getName() ;
	public double getPrice() ;
}
interface IShopCar {		// 购物车
	public void add(IGoods goods) ; // 添加商品信息
	public void delete(IGoods goods) ; // 删除商品
	public Object[] getAll() ; // 获得购物车中的全部商品信息
}
class ShopCarImpl implements IShopCar { // 购物车
	private ILink<IGoods> allGoodses = new LinkImpl<IGoods>() ;
	public void add(IGoods goods) {
		this.allGoodses.add(goods) ;
	}
	public void delete(IGoods goods) {
		this.allGoodses.remove(goods) ;
	}
	public Object [] getAll() {
		return this.allGoodses.toArray() ;
	}
}
class Cashier { // 收银台
	private IShopCar shopcar ;
	public Cashier(IShopCar shopcar) {
		this.shopcar = shopcar ;
	}
	public double allPrice() { // 计算总价
		double all = 0.0 ;
		Object result [] = this.shopcar.getAll() ;
		for (Object obj : result) {
			IGoods goods = (IGoods) obj ;
			all += goods.getPrice() ;
		}
		return all ;
	}
	public int allCount() {	// 商品数量
		return this.shopcar.getAll().length ;
	}
}
class Book implements IGoods {
	private String name ;
	private double price ;
	public Book(String name,double price) {
		this.name = name ;
		this.price = price ;
	}
	public String getName() {
		return this.name ;
	}
	public double getPrice() {
		return this.price ;
```

```
	}
	public boolean equals(Object obj) {
		if (obj == null) {
			return false ;
		}
		if (this == obj) {
			return true ;
		}
		if (!(obj instanceof Book)) {
			return false ;
		}
		Book book = (Book) obj ;
		return this.name.equals(book.name) && this.price == book.price ;
	}
	public String toString() {
		return "【图书信息】名称：" + this.name + "、价格：" + this.price ;
	}
}
class Bag implements IGoods {
	private String name ;
	private double price ;
	public Bag(String name,double price) {
		this.name = name ;
		this.price = price ;
	}
	public String getName() {
		return this.name ;
	}
	public double getPrice() {
		return this.price ;
	}
	public boolean equals(Object obj) {
		if (obj == null) {
			return false ;
		}
		if (this == obj) {
			return true ;
		}
		if (!(obj instanceof Bag)) {
			return false ;
		}
		Bag bag = (Bag) obj ;
		return this.name.equals(bag.name) && this.price == bag.price ;
	}
	public String toString() {
		return "【背包信息】名称：" + this.name + "、价格：" + this.price ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		IShopCar car = new ShopCarImpl() ;
		car.add(new Book("Java开发",79.8)) ;
```

```
            car.add(new Book("Oracle",89.8)) ;
            car.add(new Bag("小强背包",889.8)) ;
            Cashier cas = new Cashier(car) ;
            System.out.println("总价格：" + cas.allPrice() + "、购买总数量：" + cas.allCount()) ;
        }
}
```