第 5 章 链表与LinkedList类

本章主要内容

- 链表的特点；
- 创建链表；
- 查询与相等；
- 添加节点；
- 删除节点；
- 更新节点；
- 链表的视图；
- 链表的排序；
- 遍历链表；
- 链表与数组；
- 不可变链表；
- 编写简单的类创建链表。

如果需要处理一些类型相同的数据，人们习惯使用数组这种数据结构，但数组在使用之前必须定义其元素的个数，即数组的大小，而且不能再改变数组的大小，因为改变数组的大小就意味着放弃原有的全部元素。有时可能给数组分配了太多的单元而浪费了宝贵的内存空间，而且程序运行时需要处理的数据可能多于数组的单元，当需要动态地减少或增加数据项时可以使用链表这种数据结构。

5.1 链表的特点

链表是由若干节点组成的，这些节点形成的逻辑结构是线性结构，节点的存储结构是链式存储，即节点的物理地址不必是依次相邻的。对于单链表，每个节点含有一个数据，并含有下一个节点的引用。对于双链表，每个节点含有一个数据，并含有上一个节点的引用和下一个节点的引用(Java 实现的是双链表)，图 5.1 所示的是有 5 个节点的双链表(省略了上一个节点的引用箭头)。注意，链表的节点的序号是从 0 开始的，每个节点的序号等于它前面的节点的个数。

链表中节点的物理地址不必是相邻的，因此链表的优点是不需要占用一块连续的内存存储空间。

1. 删除头、尾节点的时间复杂度 $O(1)$

在双链表中始终保存着头、尾节点的地址，因此删除头、尾节点的时间复杂度是 $O(1)$。

在删除头或尾节点后，新链表中节点的序号按新的链表长度从 0 开始排列。

例如，要删除如图 5.1 所示的链表的头节点(大象节点)，根据双链表保存的头节点的地址找到头节点，然后找到头节点的下一个节点(狮子节点)，将该节点中存储的上一个节点设置成 null，即该节点(狮子节点)变成头节点。删除头节点后的链表如图 5.2 所示。

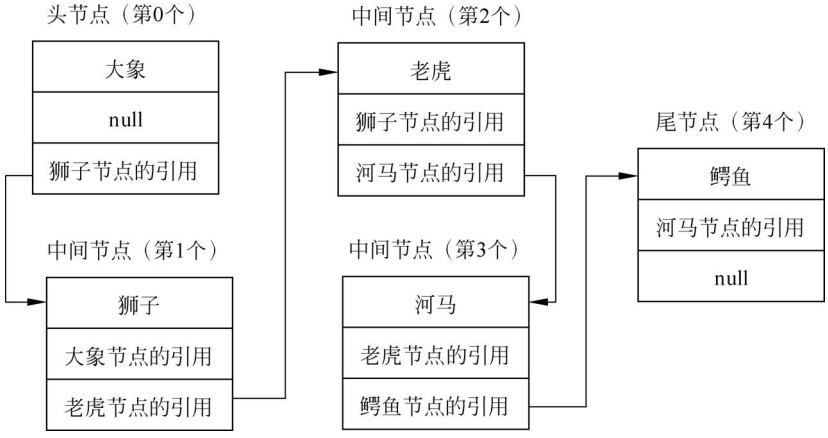

图 5.1　双链表示意图

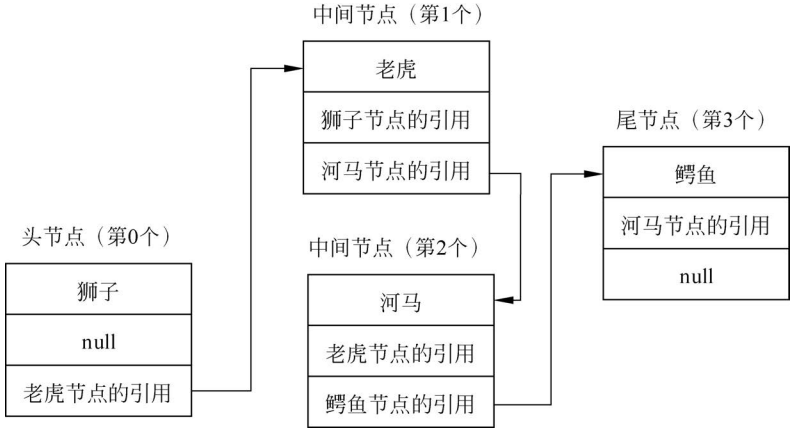

图 5.2　删除头节点(大象节点)后的链表

2. 查询头、尾节点的时间复杂度 $O(1)$

在双链表中始终保存着头、尾节点的地址，因此查询头、尾节点的时间复杂度是 $O(1)$。

3. 添加头、尾节点的时间复杂度 $O(1)$

在双链表中始终保存着头、尾节点的地址，因此添加头、尾节点的时间复杂度是 $O(1)$。

在添加头或尾节点后，新链表中节点的序号按新的链表长度从 0 开始重新排列。

例如，要给如图 5.1 所示的链表添加新的尾节点(企鹅节点)，根据双链表保存的尾节点的地址找到尾节点(鳄鱼节点)，将这个尾节点中下一个节点的引用设置成新添加的节点(企鹅节点)的引用，将添加的新节点(企鹅节点)中的上一个节点的引用设置成鳄鱼节点的引用，将添加的新节点(企鹅节点)中的下一个节点的引用设置成 null，即让新添加的节点成为尾节点。添加新尾节点后的链表如图 5.3 所示。

4. 查询中间节点的时间复杂度 $O(n)$

链表中节点的物理地址不是相邻的，节点通过互相保存引用链接在一起。对于双链表，如果节点的索引 i 小于或等于链表的长度 n 的一半，那么就从头节点开始，根据每个节点中的下一个节点的引用依次向后查找节点，并通过计数的方法查找到第 i 个节点，如果节点的索引 i 大于链表的长度 n 的一半，那么就从尾节点开始，根据每个节点中上一个节点的引用依次向前查找节点，并通过倒计数的方法查找到第 i 个节点。查询中间节点的平均时间复杂度是 $O(n)$，一般就认为时间复杂度是 $O(n)$。

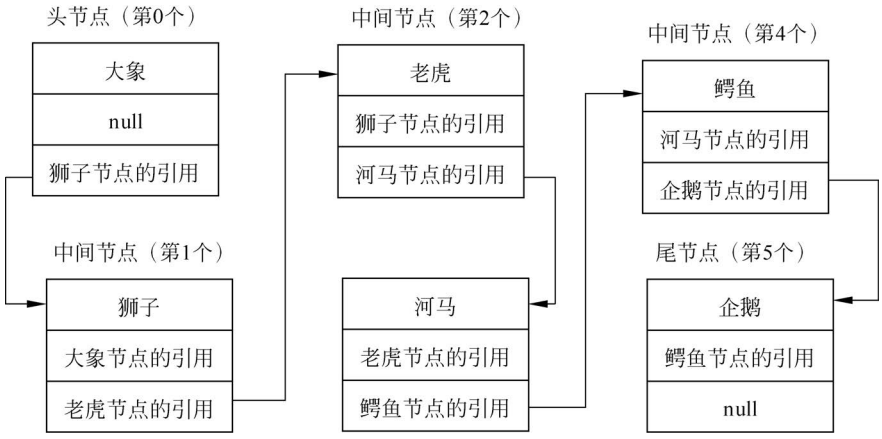

图 5.3　添加尾节点(企鹅节点)后的链表

5. 删除中间节点的时间复杂度 $O(n)$

查找到第 i 个节点，然后删除该节点：将第 $i-1$ 个节点中下一个节点的引用设置成第 $i+1$ 个节点的引用，将第 $i+1$ 个节点中上一个节点的引用设置成第 $i-1$ 个节点的引用。由于链表查询中间节点的平均时间复杂度是 $O(n)$，所以删除中间节点的时间复杂度是 $O(n)$。

在删除节点后，新链表中节点的序号按新的链表长度从 0 开始排列。

例如，要在如图 5.1 所示的链表中删除第 2 个节点(老虎节点)，那么就要从头节点(大象节点)找到第 1 个节点(狮子节点)，计数为 1，然后从狮子节点找到第 2 个节点(老虎节点)，计数为 2，再将第 1 个节点(狮子节点)中的下一个节点的引用改成第 3 个节点(河马节点)的引用，将第 3 个节点(河马节点)中的上一个节点的引用改成第 1 个节点(狮子节点)的引用，至此完成了删除第 2 个节点(老虎节点)的操作。删除第 2 个节点(老虎节点)后的链表如图 5.4 所示。

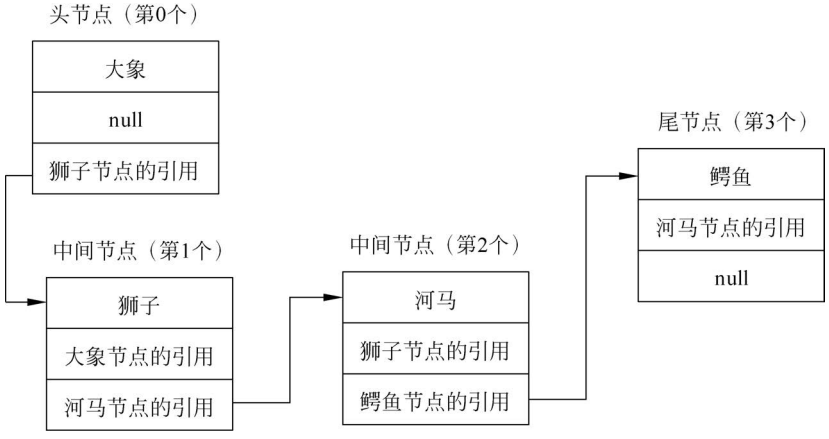

图 5.4　删除中间节点(第 2 个节点：老虎节点)后的链表

6. 插入中间节点的时间复杂度 $O(n)$

如果要在链表中插入新的第 i 个节点(i 大于 0 小于链表的长度)，首先要找到第 i 个节点，然后在第 i 个节点的前面插入新的第 i 个节点：将第 $i-1$ 个节点中的下一个节点设置成新的第 i 个节点的引用，将新的第 i 个节点中的上一个节点的引用设置成第 $i-1$ 个节点的引用，下一个节点设置成原第 i 个节点的引用，原第 i 个节点中的上一个节点设置成新的第 i 个节点的引用。由于链表查询中间节点的平均时间复杂度是 $O(n)$，所以插入中间节点的时间复杂度是 $O(n)$。

在插入新节点后，新链表中节点的序号按新的链表长度从0开始排列。

例如，要在如图5.1所示的链表中插入新的第2个节点（羚羊节点），就要从头节点（大象节点）找到第1个节点（狮子节点），计数为1，然后从狮子节点找到原第2个节点（老虎节点），计数为2，再将第1个节点（狮子节点）中的下一个节点的引用改成新的第2个节点（羚羊节点）的引用，将新的第2个节点（羚羊节点）中的上一个节点的引用设置为第1个节点（狮子节点），将原第2个节点（老虎节点）中的上一个节点的引用设置为新的第2个节点（羚羊节点）的引用，插入新的第2个节点（羚羊节点）后的链表如图5.5所示。

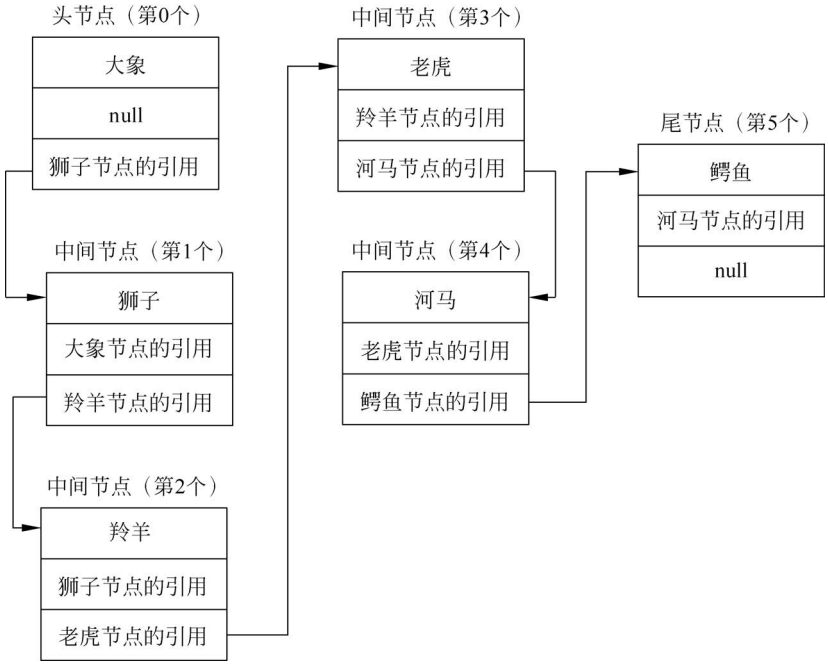

图5.5 插入中间节点（第2个节点：羚羊节点）后的链表

5.2 创建链表

链表由Java集合框架（Java Collections Framework，JCF）中的LinkedList<E>泛型类所实现。Java集合框架中的类和接口在java.util包中，主要的接口有Collection、Map、Set、List、Queue、SortedSet和SortedMap，其中List、Queue、Set是Collection的子接口，SortedSet是Set的子接口，SortedMap是Map的子接口，如图5.6所示。

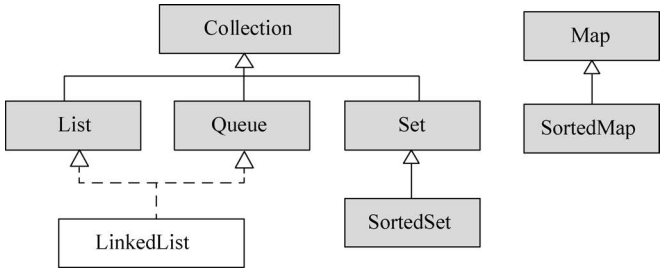

图5.6 LinkedList实现了List和Queue接口

LinkedList＜E＞泛型类实现了 Java 集合框架中的 List 和 Queue 泛型接口。LinkedList＜E＞泛型类继承了 List 和 Queue 泛型接口中的 default 关键字修饰的方法（去掉了该关键字），实现了 List 和 Queue 泛型接口中的抽象方法。

创建一个空链表，在使用 LinkedList＜E＞泛型类声明链表时必须要指定 E 的具体类型，类型是类或接口类型（不可以是基本类型，例如 int、float、char 等），即指定链表中节点里的对象的类型。例如，指定 E 是 String 类型：

```
LinkedList<String> listOne = new LinkedList<>();
```

或

```
LinkedList<String> listOne = new LinkedList<String>();
```

然后链表 listOne 就可以使用 add(E obj)方法依次添加节点，链表 listOne 在使用 add(E obj)方法添加节点时指定的节点中的对象是 String 对象，例如：

```
listOne.add("大象");
listOne.add("狮子");
listOne.add("老虎");
listOne.add("河马");
```

这时链表 listOne 就有了 4 个节点，节点中的数据都是 String 类型的对象，链表中的节点是自动链接在一起的，不需要做链接，也就是说不需要安排节点中所存放的下一个或上一个节点的引用。

链表使用 size()方法返回链表中节点的数目，如果链表中没有节点，size()方法返回 0。

当然也可以用其他链表，例如用链表 listOne 中的节点创建一个新的链表 listTwo：

```
LinkedList<String> listTwo = new LinkedList<>(listOne);
```

链表 listTwo 的节点中的数据和 listOne 的相同。如果链表 listTwo 修改了节点中的数据，不会影响 listOne 节点中的数据，同样，如果链表 listOne 修改了节点中的数据，也不会影响 listTwo 节点中的数据。

注意：链表的节点类型是 LinkedList 的内部类（Node），用户不能直接使用这个内部类，当链表使用 add(E obj)方法时，链表会自动用 Node 创建节点，并将数据 obj 放在节点中。

```
链表listOne节点中的数据:
[大象, 狮子, 老虎, 河马]
链表listTwo节点中的数据:
[大象, 狮子, 老虎, 河马, 鳄鱼, 企鹅]
链表listTwo修改了节点中的数据.
链表listTwo节点中的数据:
[elephant, lion, tiger, hippo, 鳄鱼, 企鹅]
链表listOne节点中的数据:
[大象, 狮子, 老虎, 河马]
```

图 5.7　创建链表

例 5-1　创建链表。

本例中的主类 Example5_1 首先创建一个空链表 listOne，然后向空链表 listOne 添加 4 个节点，再用 listOne 创建链表 listTwo，修改 listTwo 节点中的数据并不影响 listOne 节点中的数据，运行效果如图 5.7 所示。

Example5_1.java

```
import java.util.LinkedList;
public class Example5_1 {
    public static void main(String args[]) {
        LinkedList<String> listOne = new LinkedList<>();
        listOne.add("大象");
        listOne.add("狮子");
        listOne.add("老虎");
        listOne.add("河马");
        System.out.println("链表 listOne 节点中的数据:\n" + listOne);
```

```
            LinkedList<String> listTwo = new LinkedList<>(listOne);
            listTwo.add("鳄鱼");
            listTwo.add("企鹅");
            System.out.println("链表 listTwo 节点中的数据:\n" + listTwo);
            System.out.println("链表 listTwo 修改了节点中的数据.");
            listTwo.set(0,"elephant");
            listTwo.set(1,"lion");
            listTwo.set(2,"tiger");
            listTwo.set(3,"hippo");
            System.out.println("链表 listTwo 节点中的数据:\n" + listTwo);
            System.out.println("链表 listOne 节点中的数据:\n" + listOne);
        }
    }
```

注意：LinkedList 类重写了 Object 类的 toString()方法，使得 System.out.println()能够输出链表节点中的数据。

5.3 查询与相等

查询链表节点中的数据常用的方法如下。

- public E getFirst()：返回链表的第一个节点中的数据(时间复杂度是 $O(1)$)。
- public E getLast()：返回链表的最后一个节点中的数据(时间复杂度是 $O(1)$)。
- public E get(int index)：返回链表中序号为 index 的节点中的数据(如果 index 不是头节点或尾节点，时间复杂度是 $O(n)$)。
- public int indexOf(Object obj)：返回链表中第一个含有对象 obj 的节点的序号，如果链表中没有节点包含对象 obj，返回－1(时间复杂度是 $O(n)$)。
- public int lastIndexOf(Object obj)：返回链表中最后一个含有对象 obj 的节点的序号，如果链表中没有节点包含对象 obj，返回－1(时间复杂度是 $O(n)$)。
- public boolean contains(Object obj)：判断链表中是否有节点包含对象 obj，如果有节点包含对象 obj 返回 true，否则返回 false(时间复杂度是 $O(n)$)。
- public boolean isEmpty()：如果当前链表没有节点，返回 true，否则返回 false(时间复杂度是 $O(1)$)。
- public boolean containsAll(Collection<?> c)：判断当前链表是否包含参数 c 指定的集合中的全部节点中的数据，例如 c 指定的链表中的全部节点中的数据，如果包含，返回 true，否则返回 false(时间复杂度是 $O(n)$)。
- List<E> subList(int fromIndex, int toIndex)：返回链表中序号为 fromIndex(含)～toIndex(不含)的节点构成的视图，此视图是实现了 List 接口的一个类的实例(时间复杂度是 $O(n)$)。对视图中任何节点的修改都会使当前链表发生同步改变。

判断两个链表是否相等的方法为 public boolean equals(Object list)。LinkedList<E>泛型类重写了 Object 类的 equals()方法，如果链表和 list 的长度相同，并且对应的每个节点中的对象也相等，那么该方法返回 true，否则返回 false。

注意：在使用 indexOf(Object obj)和 contains(Object obj)方法检索或判断链表中是否有节点含有对象 obj 时，节点中的对象会调用 equals(Object obj)方法检查链表节点中的对象是否等于 obj。equals(Object obj)方法是 Object 类提供的方法，默认比较对象的引用值。在实际编程时经常需要重写 equals(Object obj)方法，以便重新规定对象相等的条件。

例 5-2 查询链表中的数据。

本例中的主类 Example5_2 比较了 get(index)方法和 getLast()方法的运行时间，People 类重写了 boolean equals(Object *o*)方法，运行效果如图 5.8 所示。

```
得到最后一个节点,即第99999个节点中的数据99999,耗时3600纳秒.
得到第50000个节点中的数据50000,耗时501200纳秒.
list:[张山:58, 李四:58, 刘五:68, 周六:78]
链表中有体重78千克的人吗? true
链表中首次出现体重是58千克的节点是0,名字:张山
链表中最后出现体重是58千克的节点是1,名字:李四
listCopy:[张山:58, 李四:58, 刘五:68, 周六:78]
和list:[张山:58, 李四:58, 刘五:68, 周六:78]
相等吗? true
listCopy:[张山:58, 李四:58, 刘五:68, 周六:78, 孙林:78]
和list:[张山:58, 李四:58, 刘五:68, 周六:78]
相等吗? false
```

图 5.8 查询链表中的数据

Example5_2.java

```
import java.util.LinkedList;
public class Example5_2 {
   public static void main(String args[]){
         int N = 100000;
         LinkedList<Integer> listInt = new LinkedList<>();
         for(int i=0;i<N;i++){
              listInt.add(i);
         }
        long startTime = System.nanoTime();
        int lastInt = listInt.getLast();
        long estimatedTime = System.nanoTime() - startTime;
        System.out.printf("得到最后一个节点,即第%d个节点中的数据%d,耗时%d纳秒.\n",
                              listInt.size()-1,lastInt,estimatedTime);
        startTime = System.nanoTime();
        int m = listInt.get(N/2);
        estimatedTime = System.nanoTime() - startTime;
        System.out.printf("得到第%d个节点中的数据%d,耗时%d纳秒.\n",
                              N/2,m,estimatedTime);
        LinkedList<People> list=new LinkedList<>();
        list.add(new People("张山",58));
        list.add(new People("李四",58));
        list.add(new People("刘五",68));
        list.add(new People("周六",78));
        System.out.println("list:"+list);
        LinkedList<People> listCopy = new LinkedList<>(list);
        int weight = 78;
        System.out.print("链表中有体重"+weight+"千克的人吗?");
        System.out.println(list.contains(new People("",weight)));
        weight = 58;
        int index = list.indexOf(new People("",weight));
        System.out.printf("链表中首次出现体重是%d千克的节点是%d,名字:%s\n",
                              weight,index,list.get(index).name);
        index = list.lastIndexOf(new People("",weight));
        System.out.printf("链表中最后出现体重是%d千克的节点是%d,名字:%s\n",
                              weight,index,list.get(index).name);
        System.out.println("listCopy:"+listCopy+"\n和 list:"+list+"\n相等吗?"+list.
equals(listCopy));
        listCopy.add(new People("孙林",78));
        System.out.println("listCopy:"+listCopy+"\n和 list:"+list+"\n相等吗?"+list.
equals(listCopy));
   }
}
```

People. java

```
public class People {
      public int weight;
      public String name;
      People(String name,int m){
           weight = m;
           this.name = name;
      }
      public boolean equals(Object o) {
            People p = (People)o;
            return weight == p.weight;
      }
      public String toString(){
           return name + ":" + weight;
      }
}
```

例 5-3 模拟随机布雷。

本例中 RandomLayMines 类的 layMines(char [][] area,int amount)方法在数组 area[][]模拟的雷区中随机布雷,该方法使用链表判断某个点(x,y)是否已经布雷。

RandomLayMines. java

```
import java.util.LinkedList;
import java.util.Random;
import java.awt.Point;
public class RandomLayMines {
     public static void layMines(char [][] area,int amount){
          LinkedList<Point> list = new LinkedList<>();
          Random random = new Random();
          while(amount > 0) {
              int x = random.nextInt(area.length);
              int y = random.nextInt(area[0].length);
              Point p = new Point(x,y);
              if(!list.contains(p)) {
                   area[x][y] = '●';
                   amount -- ;
                   list.add(p);
              }
          }
     }
}
```

本例中的主类 Example5_3 使用 layMines(char [][] area,int amount)方法布 39 颗雷,运行效果如图 5.9 所示。

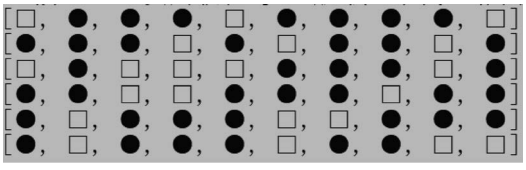

图 5.9 布雷

Example5_3. java

```
import java.util.Arrays;
public class Example5_3 {
   public static void main(String args[]){
```

```
            char [][] area = new char[6][10];
            for(int i = 0;i < area.length;i++)
                Arrays.fill(area[i],'□');
            int amount = 39;
            RandomLayMines.layMines(area,amount);
            for(int i = 0;i < area.length;i++){
                 System.out.println(Arrays.toString(area[i]));
              }
        }
    }
```

例 5-4 球队的淘汰赛。

链表获得头节点和尾节点中的对象的时间复杂度都是 $O(1)$，链表删除头节点和尾节点的时间复杂度都是 $O(1)$。对于某些问题，可以利用链表的这一特点快速地处理数据。例如，若干球队要进行淘汰赛，但不采用抽签的办法，而是将球队存放到一个链表中，并按成绩高低排列，即让成绩最好的做链表的头节点、最差的做链表的尾节点。比赛过程是让头节点和尾节点进行淘汰赛，删除头节点和尾节点，重复这个过程，直到链表的长度是 0（如果剩下一个队，相当于该队轮空，自动晋级）。

本例中 TeamGame 类的 arrangeMatch(LinkedList < String > team)方法使用链表来安排比赛。

TeamGame. java

```
import java.util.LinkedList;
public class TeamGame {
     public static void arrangeMatch(LinkedList < String > team){
          do {
               String one = team.pollFirst();
               String two = team.pollLast();
               if(two == null){                    //剩下一个队的情况
                   System.out.println(one + "轮空");
                   break;
               }
               System.out.println(one + "和" + two + "进行淘汰赛");
          }
          while(team.size()> 0 );
     }
}
```

本例中的主类使用 TeamGame 类的 arrangeMatch(LinkedList < String > team)方法安排一些球队的淘汰赛，运行效果如图 5.10 所示。

Example5_4. java

```
import java.util.LinkedList;
public class Example5_4 {
   public static void main(String args[]){
        LinkedList < String > listTeam = new LinkedList <>();
        for(int i = 1;i < = 13;i++) {
            listTeam.add("球队" + i);
        }
        TeamGame.arrangeMatch(listTeam);
   }
}
```

```
球队1和球队13进行淘汰赛
球队2和球队12进行淘汰赛
球队3和球队11进行淘汰赛
球队4和球队10进行淘汰赛
球队5和球队9进行淘汰赛
球队6和球队8进行淘汰赛
球队7轮空
```

图 5.10 淘汰赛

5.4 添加节点

向链表添加节点常用的方法如下。

- public boolean add(E *e*)：向链表的末尾添加一个新的节点，该节点中的对象是参数 *e* 指定的对象(时间复杂度是 $O(1)$)。
- public void add(int index, E *e*)：向链表的 index 指定位置添加一个新的节点，该节点中的对象是参数 *e* 指定的对象(时间复杂度是 $O(n)$)。
- public boolean addAll(Collection<? extends E> *c*)：将参数 *c* 指定的链表中的节点按照节点序号依次添加到当前链表的尾部(时间复杂度是 $O(n)$)。
- public void addFirst(E *e*)：向链表添加新的头节点，头节点中的对象是参数 *e* 指定的对象(时间复杂度是 $O(1)$)。
- public void addLast(E *e*)：向链表的末尾添加一个新的节点，该节点中的对象是参数 *e* 指定的对象(时间复杂度是 $O(1)$)。

例 5-5 向链表添加节点。

本例中的主类 Example5_5 比较了 void addLast(E *e*)方法和 void add(int index, E *e*)方法的运行时间，运行效果如图 5.11 所示。

```
插入第999999个节点,节点中的数据是8,耗时7864700纳秒.
添加最后一个节点,即第2000000个节点, 节点中的数据是12,耗时5500纳秒.
将链表listOne添加到listTwo的末尾.
链表listTwo修改了节点.
链表listOne节点中的数据:
[how, are, you]
链表listTwo节点中的数据:
[你, 好]
```

图 5.11 向链表添加节点

Example5_5.java

```
import java.util.LinkedList;
public class Example5_5 {
   public static void main(String args[]){
        int N = 1999999;
        LinkedList<Integer> listInt = new LinkedList<Integer>();
        for(int i=0;i<N;i++){
             listInt.add(i);
        }
        int m = 8;
        long startTime = System.nanoTime();
        listInt.add(N/2,m);
        long estimatedTime = System.nanoTime()-startTime;
        System.out.printf("插入第%d个节点,节点中的数据是%d,耗时%d纳秒.\n",
                          N/2,m,estimatedTime);
        m = 12;
        startTime = System.nanoTime();
        listInt.addLast(m);
        estimatedTime = System.nanoTime()-startTime;
        System.out.printf("添加最后一个节点,即第%d个节点,节点中的数据是%d,耗时%d纳秒.",
                          listInt.size()-1,m,estimatedTime);
        LinkedList<String> listOne = new LinkedList<String>();
        LinkedList<String> listTwo = new LinkedList<String>();
        listOne.add("how");
```

```
            listOne.add("are");
            listOne.add("you");
            System.out.println("\n将链表 listOne 添加到 listTwo 的末尾.");
            listTwo.addAll(listOne);
            System.out.println("链表 listTwo 修改了节点.");
            listTwo.set(0,"你");
            listTwo.set(1,"好");
            listTwo.removeLast();
            System.out.println("链表 listOne 节点中的数据:\n" + listOne);
            System.out.println("链表 listTwo 节点中的数据:\n" + listTwo);
        }
    }
```

5.5 删除节点

从链表中删除节点常用的方法如下。

- public E poll()：返回链表的第一个节点中的对象，并删除链表的第一个节点（时间复杂度是 $O(1)$）。
- public E pollFirst()：返回链表的第一个节点中的对象，并删除链表的第一个节点，如果此链表为空，则返回 null（时间复杂度是 $O(1)$）。
- public E pollLast()：返回链表的最后一个节点中的对象，并删除链表的最后一个节点（时间复杂度是 $O(1)$）。
- public E remove()：返回链表的第一个节点中的对象，并删除链表的第一个节点（时间复杂度是 $O(1)$）。
- public E remove(int index)：返回链表的第 index 个节点中的对象，并删除链表的第 index 个节点（时间复杂度是 $O(n)$）。
- public boolean remove(Object obj)：删除链表中首次出现的含有对象 obj 的节点，若删除成功返回 true，否则返回 false（时间复杂度是 $O(n)$）。
- public E removeFirst()：返回链表的第一个节点中的对象，并删除链表的第一个节点（时间复杂度是 $O(1)$）。
- public boolean removeFirstOccurrence(Object obj)：删除链表中首次出现的含有对象 obj 的节点，若删除成功返回 true，否则返回 false（时间复杂度是 $O(n)$）。
- public E removeLast()：返回链表的最后一个节点中的对象，并删除链表的最后一个节点（时间复杂度是 $O(1)$）。
- boolean removeLastOccurrence(Object obj)：删除链表中最后出现的含有对象 obj 的节点，若删除成功返回 true，否则返回 false（时间复杂度是 $O(n)$）。
- public void clear()：删除链表的全部节点（时间复杂度是 $O(1)$）
- public boolean removeAll(Collection <?> c)：删除和参数 c 指定的链表中某节点值相同的节点（时间复杂度是 $O(n)$）。
- public boolean retainAll(Collection <?> c)：仅保留和参数 c 指定的链表中某节点值相同的节点（时间复杂度是 $O(n)$）。
- public boolean removeIf(Predicate <? super E > filter)：删除满足 filter 给出的条件的节点（时间复杂度是 $O(n)$）。Predicate 是一个函数接口，其中唯一的抽象方法是 boolean test(T t)，在使用 removeIf(Predicate <? super E > filter)方法时可以将一个

Lambda 表达式传递给参数 filter，该 Lambda 表达式有一个参数，类型和节点中对象的类型一致，Lambda 表达式的返回值的类型是 boolean 型。

- boolean isEmpty()：判断链表是否不含任何节点，即链表是否为空链表，如果是空链表，返回 true，否则返回 false(时间复杂度是 $O(1)$)。

例 5-6 模拟双色球并过滤链表。

本例中 RandomNumber 类的 getRandom(int number,int amount)方法通过随机删除链表中的节点得到若干随机数。

RandomNumber.java

```
import java.util.Random;
import java.util.LinkedList;
public class RandomNumber {
   //获得[1,number]中 amount 个互不相同的随机数
   public static int[] getRandom(int number,int amount) {
        if(number<=0||amount<=0)
             throw new NumberFormatException("数字不是正整数");

        int [] result = new int[amount];              //存放得到的随机数
        LinkedList<Integer> list = new LinkedList<Integer>();
        for(int i =1;i<=number;i++) {
             list.add(i);                             //时间复杂度是 O(1)
        }
        Random random = new Random();
        for(int i= 0;i<result.length;i++){
             int m = random.nextInt(list.size());
             result[i] = list.remove(m);              //时间复杂度是 O(n)
        }
        return result;                                //返回数组,数组的每个元素是一个随机数
     }
}
```

双色球的每个投注号码由 6 个红色球的号码和 1 个蓝色球的号码组成。6 个红色球的号码互不相同，红色球的号码是 1～33 的随机数；蓝色球的号码是 1～16 的随机数。本例中的主类 Example5_6 使用 RandomNumber 类的 getRandom(int number,int amount)方法模拟双色球，同时过滤一个链表，运行效果如图 5.12 所示。

```
红色球:[10, 7, 32, 24, 33, 23]蓝色球:[1]

红色球:[15, 25, 14, 16, 28, 11]蓝色球:[1]

红色球:[2, 21, 27, 10, 24, 25]蓝色球:[10]
链表intList:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]
链表intList过滤掉偶数后:
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
链表intList保留3的倍数:
[3, 9, 15, 21, 27]
链表intList保留5的倍数:
[15]
```

图 5.12 双色球以及过滤链表

Example5_6.java

```
import java.util.Arrays;
import java.util.LinkedList;
public class Example5_6 {
     public static void main(String args[]) {
```

```
            for(int i = 1;i <= 3;i++){
                int [] red = RandomNumber.getRandom(33,6);       //双色球中的 6 个红色球
                int [] blue = RandomNumber.getRandom(16,1);      //双色球中的 1 个蓝色球
                System.out.print("\n 红色球:" + Arrays.toString(red));
                System.out.println("蓝色球:" + Arrays.toString(blue));

            }
            LinkedList < Integer > intList = new LinkedList < Integer >();
            LinkedList < Integer > filterList = new LinkedList < Integer >();
            LinkedList < Integer > retainList = new LinkedList < Integer >();
            for(int i = 1;i <= 32;i++) {
                intList.add(i);
                if(i % 2 == 0)
                    filterList.add(i);
                if(i % 3 == 0)
                    retainList.add(i);
            }
            System.out.println("链表 intList:\n" + intList);
            intList.removeAll(filterList);
            System.out.println("链表 intList 过滤掉偶数后:\n" + intList);
            intList.retainAll(retainList);
            System.out.println("链表 intList 保留 3 的倍数:\n" + intList);
            intList.removeIf((m) ->{ if(m % 5!= 0)
                                        return true;
                                    else
                                        return false;});
            System.out.println("链表 intList 保留 5 的倍数:\n" + intList);
        }
    }
```

注意：读者可以把本例和第 4 章中的例 4-8 进行比较，因为使用了链表，这里的代码更加简练，两个例子中获得随机数的算法的时间复杂度都是 $O(n^2)$。

例 5-7 处理链表中重复的数据。

本例中 HandleRecurring 类的 handleRecurring(LinkedList < String > list)方法处理链表 list 中重复的数据，该方法返回的链表中没有重复的数据(对于重复的数据，保留其中一个)。

HandleRecurring. java

```
import java.util.LinkedList;
public class HandleRecurring {
    public static LinkedList < String > handleRecurring(LinkedList < String > list) {
        LinkedList < String > result = new LinkedList < String >();
        while(list.size()> 0){
            String ojb = list.removeFirst();
            if(!result.contains(ojb)){
                result.add(ojb);
            }
        }
        return result;                          //返回没有重复数据的数组
    }
}
```

```
链表list:
[大象, 狮子, 老虎, 狮子, 老虎, 河马]
处理重复数据后,链表list:
[大象, 狮子, 老虎, 河马]
删除用'子'结尾的节点,链表list:
[大象, 老虎, 河马]
```

图 5.13 处理重复的数据

本例中的主类 Example5_7 使用 HandleRecurring 类的 handleRecurring (LinkedList < String > list)方法处理链表中重复的数据，然后删除名字中有“子”的动物的节点，运行效果如图 5.13 所示。

Example5_7.java

```
import java.util.LinkedList;
public class Example5_7 {
     public static void main(String args[]){
          LinkedList<String> list = new LinkedList<String>();
          list.add("大象");
          list.add("狮子");
          list.add("老虎");
          list.add("狮子");
          list.add("老虎");
          list.add("河马");
          System.out.println("链表 list:\n" + list);
          list = HandleRecurring.handleRecurring(list);
          System.out.println("处理重复数据后,链表 list:\n" + list);
          list.removeIf((s) ->{if(s.endsWith("子"))
                                   return true;
                               else
                                   return false;});
          System.out.println("删除用\'子\'结尾的节点,链表 list:\n" + list);
     }
}
```

注意：读者可以把本例和第 4 章中的例 4-10 进行比较，因为使用了链表，这里的代码更加简练，本例中去掉重复数据的算法的时间复杂度是 $O(n^2)$。

5.6 更新节点

更新链表节点的常用方法如下。

- public E set(int index,E element)：将当前链表 index 序号的节点中的对象替换为参数 element 指定的对象，并返回被替换的对象，如果 index 是 0 或尾节点的序号，该方法的时间复杂度是 $O(1)$，否则时间复杂度是 $O(n)$。
- Collections 类的静态方法 static <T> void fill(List<? super T> list,T obj)：将链表 list 的每个节点中的对象都设置为参数 obj 指定的对象。

更新节点只是更新链表节点中的数据，不是更改链表节点的结构，因为没有删除链表的节点或给链表添加新节点。

例 5-8 更新链表节点中的数据。

本例中的主类 Example5_8 使用 set(int index,E element)方法和 fill(List<? super T> list,T obj)方法更新链表节点中的数据，运行效果如图 5.14 所示。

```
链表list节点中的数据:
[A, B, C, D, E, F, G]
更新链表list节点中的数据.
链表list节点中的数据:
[a, b, c, d, e, f, g]
将链表list每个节点中的数据都更新为:G.
链表list节点中的数据:
[G, G, G, G, G, G, G]
```

图 5.14 更新链表节点中的数据

Example5_8.java

```
import java.util.Collections;
import java.util.LinkedList;
public class Example5_8 {
     public static void main(String args[]){
          LinkedList<String> list = new LinkedList<String>();
          for(char c = 'A';c<='G';c++) {
               list.add("" + c);
          }
```

```
            System.out.println("链表 list 节点中的数据:\n" + list);
            System.out.println("更新链表 list 节点中的数据.");
            for(char c = 'a',i = 0;c <= 'g';c++,i++) {
                 list.set(i,"" + c);
            }
            System.out.println("链表 list 节点中的数据:\n" + list);
            System.out.println("将链表 list 每个节点中的数据都更新为:G.");
            Collections.fill(list,"G");
            System.out.println("链表 list 节点中的数据:\n" + list);
        }
    }
```

5.7 链表的视图

链表的视图由链表中的若干节点所构成，其状态变化和链表是同步的。

得到链表的视图的一个常用方法是 List < E > subList(int fromIndex, int toIndex)，该方法是 AbstractList 类所实现的 List 接口中的一个方法（AbstractList 类是 LinkedList 的间接父类）。该方法返回一个当前链表中序号为 fromIndex（含）～toIndex（不含）的节点构成的视图，此视图是 AbstractList 类的子类 SubList < E >的实例（SubList < E >也是 AbstractList 类的内部类）。更改视图的节点（增加或删除节点）或对节点的数据进行修改都会使当前链表发生同步改变。需要特别注意的是，一旦链表添加或删除节点，就会破坏视图的索引，就会影响之前链表用 subList()方法返回的视图，这个视图将无法再被继续使用（如果继续使用，在运行时会触发 ConcurrentModificationException 异常），链表必须用 subList()方法重新返回一个新的视图。

使用视图的好处是可以将经常需要修改的节点放在一个视图中，有利于数据的一致性处理。

在使用视图时要注意视图中节点和原链表节点的对应关系。例如，假如链表 list 中有 6 个节点，如图 5.15 所示。

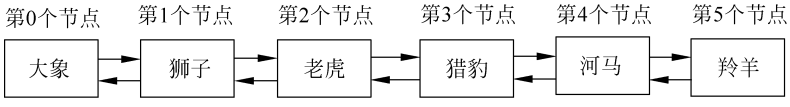

图 5.15 链表 list

链表 list 的一个视图 listView

```
List listView = list.subList(1,4);
```

有 3 个节点，如图 5.16 所示。

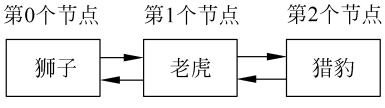

图 5.16 链表 list 的视图 listView

需要注意的是，在原链表 list 中狮子节点是第 1 个节点，但在视图 listView 中狮子节点是第 0 个节点；在原链表 list 中老虎节点是第 2 个节点，但在视图 listView 中老虎节点是第 1 个节点；在原链表 list 中猎豹节点是第 3 个节点，但在视图 listView 中猎豹节点是第 2 个节点。这种对应关系依赖于链表 list 使用 subList(int fromIndex,int toIndex)方法得到视图 listView 时参数 fromIndex 和 toIndex 的取值情况。

如果视图 listView 增加一个节点，例如尾节点鬣狗，如图 5.17 所示，那么原链表 list 会同步更新，发生变化，如图 5.18 所示（猎豹节点和河马节点之间多了一个鬣狗节点）：

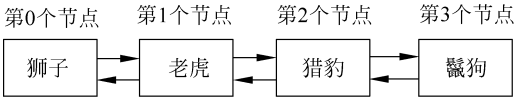

图 5.17 listView 发生变化

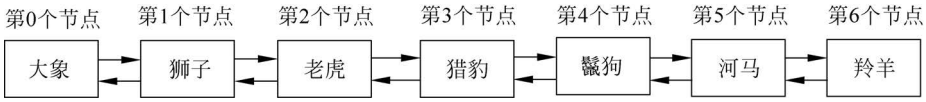

图 5.18 原链表 list 同步发生变化

例 5-9 使用视图更新链表节点中的数据。

本例中的主类 Example5_9 使用链表的视图更新链表节点中的天气信息，运行效果如图 5.19 所示。

```
链表list节点中的数据:
[北京天气预报, 最高气温, 最高气温, 北京气象局.]
使用视图更新链表list节点中的数据.
链表list节点中的数据:
[北京天气预报, 最高气温36度, 最低气温23度, 南风5-6级, 夜间有小雨, 北京气象局.]
使用视图更新链表list节点中的数据.
链表list节点中的数据:
[北京天气预报, 最高气温32度, 最低气温20度, 傍晚有大雨, 北京气象局.]
```

图 5.19 使用视图更新链表节点中的数据

Example5_9.java

```
import java.util.LinkedList;
import java.util.List;
public class Example5_9 {
     public static void main(String args[]){
          LinkedList<String> list = new LinkedList<String>();
          list.add("北京天气预报");
          list.add("最高气温");
          list.add("最高气温");
          list.add("北京气象局.");
          List<String> listView = list.subList(1,3);
          System.out.println("链表 list 节点中的数据:\n" + list);
          System.out.println("使用视图更新链表 list 节点中的数据.");
          listView.set(0,"最高气温 36 度");
          listView.set(1,"最低气温 23 度");
          listView.add("南风 5～6 级");
          listView.add("夜间有小雨");
          System.out.println("链表 list 节点中的数据:\n" + list);
          System.out.println("使用视图更新链表 list 节点中的数据.");
          listView.set(0,"最高气温 32 度");
          listView.set(1,"最低气温 20 度");
          listView.set(2,"傍晚有大雨");
          listView.remove(listView.size() - 1);
          System.out.println("链表 list 节点中的数据:\n" + list);
     }
}
```

5.8 链表的排序

public default void sort(Comparator<? super E>c)方法是 List 接口的默认排序方法(用 default 关键字修饰的方法)，LinkedList 继承了该排序方法(去掉了关键字 default)。该排序

方法的参数 c 是 Comparator 泛型接口，Comparator 泛型接口是一个函数接口，即此接口中的抽象方法只有一个——int compare(T a, T b)，该方法比较两个参数 a 和 b 的顺序，返回值是正数表示 a 大于 b，返回值是负数表示 a 小于 b，返回 0 表示 a 等于 b。

Comparator 是一个函数接口，因此在使用该方法时可以向参数 c 传递一个 Lambda 表达式，该 Lambda 表达式必须有两个参数，在 Lambda 表达式中必须有 return 语句，返回的值是 int 型数据（和其代表的 int compare(T a, T b)方法保持一致）。例如：

```
sort((a,b) ->{if(a.height > b.height)
          return 1;
     else if(a.height < b.height)
          return -1;
     else
          return 0; } )
```

sort()方法在执行过程中让 a 和 b 取链表节点中的对象，以便根据 Lambda 表达式规定的对象的大小关系对链表节点中的对象排序（升序）。List 接口的 sort()方法是归并排序、是稳定排序，其时间复杂度为 $O(n\log n)$。

例 5-10 排序链表。

本例中的主类 Example5_10 分别按学生的数学和英语成绩升序排序链表、按数学和英语成绩降序排序链表，运行效果如图 5.20 所示。

```
链表节点中的数据:
[张山:数学:89 英语:78, 李四:数学:58 英语:77, 刘五:数学:68 英语:90, 周六:数学:78 英语:77]
按数学成绩升序，链表节点中的数据:
[李四:数学:58 英语:77, 刘五:数学:68 英语:90, 周六:数学:78 英语:77, 张山:数学:89 英语:78]
按英语成绩升序，链表节点中的数据:
[李四:数学:58 英语:77, 周六:数学:78 英语:77, 张山:数学:89 英语:78, 刘五:数学:68 英语:90]
按数学成绩降序，链表节点中的数据:
[张山:数学:89 英语:78, 周六:数学:78 英语:77, 刘五:数学:68 英语:90, 李四:数学:58 英语:77]
按英语成绩降序，链表节点中的数据:
[刘五:数学:68 英语:90, 张山:数学:89 英语:78, 周六:数学:78 英语:77, 李四:数学:58 英语:77]
```

图 5.20 链表的排序

Example5_10.java

```
import java.util.LinkedList;
public class Example5_10 {
   public static void main(String args[]){
        LinkedList<Student> list = new LinkedList<Student>();
        list.add(new Student("张山",89,78));
        list.add(new Student("李四",58,77));
        list.add(new Student("刘五",68,90));
        list.add(new Student("周六",78,77));
        System.out.println("链表节点中的数据:\n" + list);
        list.sort((a,b) ->{return a.math - b.math;});
        System.out.println("按数学成绩升序,链表节点中的数据:\n" + list);
        list.sort((a,b) ->{return a.english - b.english;});
        System.out.println("按英语成绩升序,链表节点中的数据:\n" + list);
        list.sort((a,b) ->{if(a.math - b.math < 0)          //所谓的大小由比较器来规定
                                    return 1;
                               else if(a.math - b.math > 0)
                                    return -1;
                               else
                                    return 0; });
        System.out.println("按数学成绩降序,链表节点中的数据:\n" + list);
        list.sort((a,b) ->{return b.english - a.english;});
        System.out.println("按英语成绩降序,链表节点中的数据:\n" + list);
   }
}
```

Student.java

```
public class Student {
      public int math,english;
      public String name;
      public Student(String name,int math,int english){
            this.name = name;
            this.math = math;
            this.english = english;
      }
      public String toString(){
            return name+":"+"数学:"+math+" 英语:"+english;
      }
}
```

5.9 遍历链表

无论何种集合,都应该允许用户以某种方法遍历集合中的对象,而不需要知道这些对象在集合中是如何表示及存储的,Java集合框架为各种数据结构的集合(例如链表、散列表等不同存储结构的集合)提供了迭代器。

某些集合根据其数据存储结构和所具有的操作也会提供返回数据的方法,例如LinkedList类中的get(int index)方法将返回当前链表中第index个节点中的对象。LinkedList的存储结构不是顺序结构,即节点的物理地址不必是依次相邻的,因此链表调用get(int index)方法的速度比顺序存储结构的集合调用get(int index)方法的速度慢。

当用户需要遍历集合中的节点时,应当使用该集合提供的迭代器,而不是让集合本身来遍历其中的节点。

1. 单向迭代器

单向迭代器是从链表的头节点开始遍历到尾节点。

链表对象可以使用Iterator<E>iterator()方法获取一个实现Iterator接口的对象,该对象就是针对当前链表的单向迭代器。在迭代器内部使用了游标技术,单向迭代器的游标的初始状态是指向链表的头节点的前面,如果游标可以向后移动(向链表的尾节点方向),单向迭代器调用hasNext()方法返回true,否则返回false。连续地调用next()方法会将游标移动到尾节点,这时游标将无法向后移动,再调用next()方法会触发运行异常NoSuchElementException(链表是空链表,直接调用next()方法也会触发运行异常NoSuchElementException)。

2. 双向迭代器

链表对象可以使用ListIterator<E>listIterator()方法获取一个实现ListIterator接口的对象(ListIterator接口是Iterator的子接口),该对象就是针对当前链表的双向迭代器。双向迭代器是双游标迭代器,一个是向后游标(向链表的尾节点方向),一个是向前游标(向链表的头节点方向)。双游标同时移动,向前游标在向后游标的后面,如图5.21所示(实心箭头是向后游标,空心箭头是向前游标),即当向后游标指向第i个节点时,向前游标指向第$i+1$个节点,如图5.21(b)所示。初始状态向后游标指向头节点的前面,向前游标指向头节点,如图5.21(a)所示。当向后游标指向尾节点时,向前游标指向尾节点的后面,如图5.21(c)所示。

如果向后游标可以向后移动,即通过移动可以让向后游标指向链表中的一个节点,双向迭代器调用hasNext()方法返回true,否则返回false。如果向前游标可以向前移动,即通过移动可以让向前游标指向链表中的一个节点,双向迭代器调用hasPrevious()方法返回true,否则

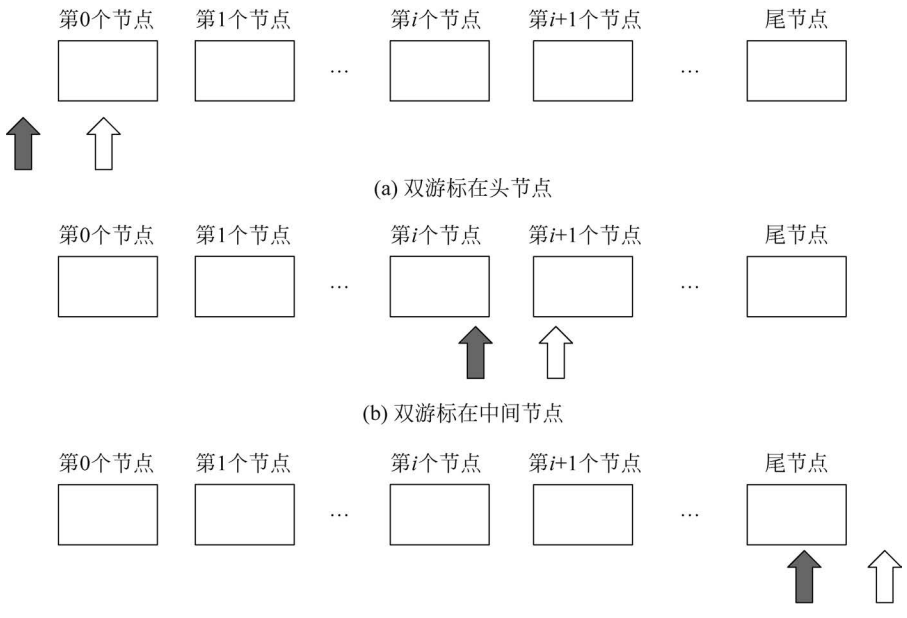

图 5.21　双游标移动的 3 种状态

返回 false。如果链表不是空链表，迭代器第一次调用 hasNext()方法返回 true，但调用 hasPrevious()方法会返回 false，原因是向后游标的初始状态是指向链表的头节点的前面，可以向后移动，向前游标指向头节点，不能再向前移动。

双向迭代器调用 next()方法可以将双游标向后移动一个位置，即让向后游标指向下一个节点，并返回向后游标所指向的节点中的对象，连续地调用 next()方法会将向后游标指向尾节点，这时游标将无法向后移动，再调用 next()方法会触发运行异常 NoSuchElementException（链表是空链表，直接调用 next()方法也会触发运行异常 NoSuchElementException）。双向迭代器调用 previous()方法可以将双游标向前移动一个位置，即让向前游标指向上一个节点，并返回向前游标所指向的节点中的对象，如果游标无法向前移动（连续地调用 previous()方法会将向前游标移动到头节点，这时游标将无法向前移动），再调用 previous()方法会触发运行异常 NoSuchElementException（链表是空链表，直接调用 previous()方法也会触发运行异常 NoSuchElementException）。

双向迭代器调用 int nextIndex()方法可以返回向后游标将要向后移动、要指向的下一个节点的索引序号，如果游标不能向后移动，该方法返回链表的长度。双向迭代器调用 int previousIndex()方法可以返回向前游标将要向前移动、要指向的上一个节点的索引序号，如果游标不能向前移动，该方法返回－1。

3. 遍历与更新

双向迭代器在遍历链表时可以同时更新链表。双向迭代器调用 next()或 previous()方法返回节点中的数据后紧接着调用 add(E obj)方法，可以在该节点后插入一个节点（新节点的序号从当前节点开始递增），调用 set(E obj)方法可以更新该节点中的数据，调用 remove()方法可以删除该节点。单向迭代器在遍历链表时也可以同时更新链表，但只能删除节点。单向迭代器调用 next()方法返回节点中的数据后紧接着调用 remove()方法可以删除该节点。迭代器对链表实施的添加、删除、更新节点等操作一直等到迭代器遍历链表完毕才生效。

4. 遍历与锁定

单向或双向迭代器在遍历链表时不允许链表本身调用方法更改链表节点的结构，即不允许链表调用方法增加节点、删除节点、排序链表，但允许链表调用 set(int index,E obj)方法更新节点中的数据，因为更新节点只是更新链表节点中的数据，不是更改链表节点的结构。在迭代器没有结束遍历之前，如果链表进行增加节点、删除节点、排序链表等操作，程序运行时(无编译错误)将触发 java.util.ConcurrentModificationException 异常。程序必须等到迭代器被使用完毕才允许链表调用 add()、remove()方法添加节点、删除节点或排序链表。

5. for-each 语句

在使用 for-each 语句遍历一个链表时，禁止当前链表调用 add()、remove()方法添加节点、删除节点或排序链表，其原因是 for-each 算法的内部中启用了迭代器(用户知道即可，但用户程序不能显式地看见相应的代码)。例如，下列代码遍历一个链表 list(节点类型是 String)无编译错误，但运行时将触发 ConcurrentModificationException 异常。

```
for(String s:list) {
    if(s.length() == 5) {
          list.add("Javahello");                    //添加节点操作会触发异常
    }
}
```

注意：链表获得迭代器后，在使用迭代器之前又对链表进行了更新，例如添加节点、删除节点或排序链表，那么将无法再使用该迭代器遍历链表。如果想使用迭代器，必须让链表重新返回一个新的迭代器。

例 5-11 使用单向迭代器和双向迭代器遍历链表。

本例中的主类 Example5_11 分别使用单向迭代器和双向迭代器遍历了链表，在遍历的过程中还更新了链表，运行效果如图 5.22 所示。

```
链表:[A, B, C, D, E, F, G]
单向迭代器遍历链表并删除了尾节点.
A B C D E F G
目前的链表:
[A, B, C, D, E, F]
双向迭代器遍历链表，从头到尾，再从尾到头并更新了节点.
A B C D E F F E D C B A
目前的链表:
[A(a), B(b), C(c), D(d), E(e), F(f)]
```

图 5.22 遍历链表

Example5_11.java

```
import java.util.Iterator;
import java.util.ListIterator;
import java.util.LinkedList;
public class Example5_11 {
     public static void main(String args[]){
          LinkedList<String> list = new LinkedList<String>();
          for(char c = 'A';c<='G';c++) {
               list.add("" + c);
          }
          System.out.println("链表:" + list);
          Iterator<String> iterOneWay = list.iterator();          //单向迭代器
          System.out.println("单向迭代器遍历链表并删除了尾节点.");
          while(iterOneWay.hasNext()){
             String item = iterOneWay.next();
             System.out.print(item + " ");
```

```
            if(item.equals("G"))
                iterOneWay.remove();
        }
        System.out.println("\n目前的链表:\n" + list);
        ListIterator<String> iterTwoWay = list.listIterator(); //双向迭代器
        System.out.println("双向迭代器遍历链表,从头到尾,再从尾到头并更新了节点.");
        while(iterTwoWay.hasNext()){
            String item = iterTwoWay.next();
            System.out.print(item + " ");
            if(iterTwoWay.nextIndex() == list.size()) {          //向后游标指向尾节点
                while(iterTwoWay.hasPrevious()){
                    item = iterTwoWay.previous();
                    System.out.print(item + " ");
                    char c = item.charAt(0);
                    c = (char)(c + 32);                          //小写
                    iterTwoWay.set(item + "(" + c + ")");
                }
                break;
            }
        }
        System.out.println("\n目前的链表:\n" + list);
    }
}
```

例 5-12 使用链表解决约瑟夫问题。

约瑟夫问题(也称围圈留一问题)：若干人围成一圈,从某个人开始顺时针(或逆时针)数到第 3 个人,该人从圈中退出,然后继续顺时针(或逆时针)数到第 3 个人,该人从圈中退出,以此类推,程序输出圈中最后剩下的一个人。

第 4 章中的例 4-9 使用数组解决了约瑟夫问题,本例中 LeaveOneAround 类的 leaveOne(LinkedList<Integer> people)方法通过遍历链表解决约瑟夫问题,在遍历链表的过程中删除相应节点模拟退出圈的人。

LeaveOneAround.java

```
import java.util.ListIterator;
import java.util.LinkedList;
public class LeaveOneAround {
    public static void leaveOne(LinkedList<Integer> people) {
        int number = 3;
        ListIterator<Integer> iterTwoWay = people.listIterator();    //双向迭代器
        while(people.size()>1){                                      //圈中的人数超过1
            int peopleNumber = -1;
            for(int count =1;count<= number;count++){
                peopleNumber = iterTwoWay.next();
                if(count == 3){
                    iterTwoWay.remove();                             //数到第3个人,该人退出
                }
                if(iterTwoWay.nextIndex() == people.size()) {        //如果向后游标指向尾节点
                    while(iterTwoWay.hasPrevious()){
                        iterTwoWay.previous();                       //将向前游标移到头节点
                    }
                }
            }
            System.out.printf("号码%d退出圈\n",peopleNumber);
        }
        System.out.printf("最后剩下的号码是%d\n",people.getFirst());
    }
}
```

本例中的主类 Example5_12 演示了 11 个人的约瑟夫问题,运行效果如图 5.23 所示。

```
号码3退出圈
号码6退出圈
号码9退出圈
号码1退出圈
号码5退出圈
号码10退出圈
号码4退出圈
号码11退出圈
号码8退出圈
号码2退出圈
最后剩下的号码是7
```

图 5.23 约瑟夫问题

Example5_12.java

```
import java.util.LinkedList;
public class Example5_12 {
    public static void main(String args[]) {
        LinkedList<Integer> people = new LinkedList<Integer>();
        for(int i=1;i<=11;i++){
            people.add(i);
        }
        LeaveOneAround.leaveOne(people);
    }
}
```

6. 动态遍历

一个遍历链表的方法(算法)在遍历链表时,让链表的每个节点中的对象参与某种运算,并输出运算后的结果,称这样的遍历方法为动态遍历方法。

public default void forEachRemaining(Consumer<? super E> action) 方法是 Iterator 接口中的默认方法,链表返回的迭代器(单向或双向)可以使用该方法动态地遍历链表。此方法的参数类型是 Consumer 函数接口,该接口中的抽象方法 void accept(T *t*)的返回类型是 void 型。那么在调用此方法时可以将一个 Lambda 表达式传递给 action,该 Lambda 表达式的参数类型和链表节点中的数据类型一致,不需要 return 语句(如果有 return,不允许返回任何值)。例如,可以将下列 Lambda 表达式

```
(value)->{ System.out.println(value);}
```

传递给 action。此方法在执行过程中将使用这个 Lambda 表达式,让 Lambda 表达式的参数 value 依次取迭代器返回的节点中的对象,直到迭代器的游标无法向后或向前移动为止。

注意:如果再次使用 forEachRemaining()方法,必须要重新返回迭代器。

```
鸡蛋:12¥,牛奶:59¥,饼干:16¥
总消费:87¥
牛肉:82¥,猪肉:159¥,香肠:66¥
总消费:307¥
螃蟹:52¥,大虾:529¥,海螺:76¥
总消费:657¥
```

图 5.24 动态遍历链表计算总消费

例 5-13 动态遍历链表计算总消费。

本例中的主类 Example5_13 动态遍历节点中是 String 对象的一个链表,在遍历这个链表时让节点中的 String 对象参与运算,计算出购物小票的消费总额,运行效果如图 5.24 所示。

Example5_13.java

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.function.Consumer;
public class Example5_13 {
    public static void main(String args[]) {
        LinkedList<String> shoppingReceipt = new LinkedList<String>();
```

```
            shoppingReceipt.add("鸡蛋:12￥,牛奶:59￥,饼干:16￥");
            shoppingReceipt.add("牛肉:82￥,猪肉:159￥,香肠:66￥");
            shoppingReceipt.add("螃蟹:52￥,大虾:529￥,海螺:76￥");
            Iterator<String> iter = shoppingReceipt.iterator();  //单向迭代器
            String regex = "\\D+";                               //匹配任何非数字字符序列
            Consumer<String> action =
                    (String value)->{ String price[] = value.split(regex);
                                      int sumPrice = 0;
                                      for(String s:price) {
                                              if(s.length()>0)
                                                   sumPrice += Integer.parseInt(s);
                                              }
                                      System.out.println(value+"\n总消费:"+sumPrice+"￥");
                                    };
            iter.forEachRemaining(action);
        }
    }
```

例 5-14　动态遍历链表计算数组元素值的和。

本例中的主类 Example5_14 动态遍历节点中是数组的一个链表，在遍历这个链表时让节点中的数组参与运算，计算数组的元素值之和，运行效果如图 5.25 所示。

```
链表节点中的数组:
[9, 1, 6, 1, 5]
[3, 6, 9, 8, 0]
[9, 4, 6, 7, 4]
[7, 4, 8, 4, 8]
[1, 1, 1, 2, 7]
[6, 5, 3, 2, 2]
[2, 9, 1, 4, 0]
[8, 7, 3, 6, 7]
各个数组的元素值之和依次是:
22 26 30 31 12 18 16 31
```

图 5.25　动态遍历链表计算数组元素值的和

Example5_14.java

```
import java.util.LinkedList;
import java.util.Arrays;
import java.util.Random;
import java.util.Iterator;
public class Example5_14 {
    public static void main(String args[]) {
        LinkedList<int []> list = new LinkedList<int []>();
        Random random = new Random();
        for(int i=1;i<=8;i++) {
            int[] number = new int[5];
            for(int k=0;k<number.length;k++) {
                number[k] = random.nextInt(10);
            }
            list.add(number);
        }
        System.out.println("链表节点中的数组:");
        for(int [] a:list){
            System.out.println(Arrays.toString(a));
        }
        Iterator<int[]> iter =list.iterator();       //单向迭代器
        System.out.println("各个数组的元素值之和依次是:");
        iter.forEachRemaining((int[] v)->{
                                          int sum = 0;
                                          for(int i=0;i<v.length;i++){
```

```
                                                    sum += v[i];
                                                }
                                                System.out.print(sum + " ");});
        }
}
```

注意：在链表的节点中可以是任何引用型的数据，例如类、数组、接口等类型的数据。

5.10　链表与数组

链表调用 public < T > T[] toArray(T[] a)方法可以把节点中的对象放到一个数组中。在使用这个方法之前要事先预备一个数组，长度为 1 即可，将这个数组传递给 toArray(T[] a)方法的参数 a，此参数仅是通知该方法再创建一个新的数组，将链表节点中的对象放到新数组中，并返回这个新数组的引用。

例 5-15　将链表节点中的对象放到新数组中。

本例中的主类 Example5_15 将链表节点中的对象放到新数组中，运行效果如图 5.26 所示。

```
链表list:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
数组arr:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
数组arr更改了数据:
[101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112]
链表list:
[101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112]
```

图 5.26　将链表节点中的对象放到新数组中

Example5_15.java

```
import java.util.LinkedList;
import java.util.Arrays;
public class Example5_15 {
      public static void main(String args[]) {
            LinkedList < Student > list = new LinkedList < Student >();
            for(int i = 1;i < = 12;i++) {
                  list.add(new Student(i));
            }
            System.out.println("链表 list:\n" + list);
            Student []a = new Student[1];
            Student []arr = list.toArray(a);
            System.out.println("数组 arr:\n" + Arrays.toString(arr));
            for(int i = 0;i < arr.length;i++)
                  arr[i].number = arr[i].number + 100;
            System.out.println("数组 arr 更改了数据:\n" + Arrays.toString(arr));
            System.out.println("链表 list:\n" + list);
      }
}
class Student {
   int number;
   Student(int n){
       number = n;
   }
   public String toString(){
       return number + "";
   }
}
```

5.11 不可变链表

所谓不可变链表是指，这种链表一旦诞生，用户不可以再改变链表的节点，既不允许删除节点、添加节点，也不允许排序链表，如果用户进行这些操作会触发运行异常（UnsupportedOperationException）。

List 接口的 static <E> List <E> of(E… elements)静态方法返回一个不可变链表。不可变链表由 ImmutableCollections 类负责提供，该类不是给用户程序的 API，只是 Java 集合框架自己使用。

```
大象 狮子 老虎 猎豹
----------
1 2 3 4 5 6 7 8 9 10
```

图 5.27 不可变链表

例 5-16 把数组的元素值放到不可变链表中。

本例中的主类 Example5_16 把一个数组的元素值放到一个不可变链表中，并把几个 String 对象放到一个不可变链表中，运行效果如图 5.27 所示。

Example5_16.java

```
import java.util.List;
public class Example5_16 {
    public static void main(String args[]) {
        List<String> list = List.of("大象","狮子","老虎","猎豹");
        for(String m:list) {
            System.out.print(m+" ");
        }
        Integer[] a = {1,2,3,4,5,6,7,8,9,10};
        List<Integer> listInt = List.of(a);
        System.out.println("\n----------");
        for(Integer m:listInt) {
            System.out.print(m+" ");
        }
    }
}
```

5.12 编写简单的类创建链表

Java 集合框架（Java Collections Framework，JCF）中的 LinkedList 类不仅提供了丰富的方法，而且和整个 JCF 融为一体。LinkedList 类使得用户可专注于在程序设计中怎样使用链表解决问题，而不必再编写繁杂的实现链表的代码，这正是使用框架的目的。

本节通过编写简单的链表类加深了解链表的特点（见 5.1 节），所编写的链表类 LinkedInt（见例 5-17）简单到节点中只能存储 int 型数据，LinkedInt 类提供的方法也只是 5.1 节中提到的最基本的操作，例如添加节点、删除节点、查询节点等。在所编写的 LinkedInt 类中多写了一个旋转方法，以便程序用该 LinkedInt 类解决约瑟夫问题。

建议读者参考 5.1 节中的有关文字内容和示意图阅读例 5-17 的代码，毕竟这里编写的 LinkedInt 类相对 java.util 包提供的 LinkedList 类要简单很多，理解这里的 LinkedInt 类主要是为了进一步了解链表的特点。

例 5-17 编写简单的类创建链表。

本例中的 Node 类负责封装 LinkedInt 类中需要的节点。

Node. java

```
public class Node {
      public Integer data;
      public Node previous;
      public Node next;
      public Node(Integer number){
            data = number;
      }
}
```

LinkedInt. java

```
public class LinkedInt {                          //参考本章 5.1 节的内容阅读本代码
      int size = 0;                               //链表的长度
      private Node head;                          //链表的头
      private Node tail;                          //链表的尾
      public LinkedInt(){                         //创建一个空链表
         head = null;
         tail = null;
      }
      public void addFirst(Integer m) {
            Node node = new Node(m);
            if(head!= null) {
                  node.next = head;
                  head = node;
            }
            else {
                head = node;
            }
            if(tail == null)
                   tail = head;
            size++;
      }
      public void addLast(Integer m) {
            Node node = new Node(m);
            if(tail != null){
                  node.previous = tail;
                  tail.next = node;
                  tail = node;
            }
            else {
                  tail = node;
                  head = node;
            }
            size++;
      }
      public Integer getFirst() {
            if(head != null){
                  return head.data;
            }
            else {
                  return null;
            }
      }
      public Integer removeFirst() {
            Integer data = null;
            if(head != null){
                  data = head.data;
                  head = head.next;
                  head.previous = null;
                  size --;
            }
```

```
        return data;
    }
    public Integer removeLast() {
        Integer data = null;
        if(tail != null){
            data = tail.data;
            tail = tail.previous;
            tail.next = null;
            size --;
        }
        return data;
    }
    public Integer getLast() {
        if(tail != null){
            return tail.data;
        }
        else {
            return null;
        }
    }
    public Integer get(int index) {
        if(index < 0||index > size - 1){
            throw new IndexOutOfBoundsException("下标越界");
        }
        if(index == 0)
            return head.data;
        if(index == size - 1)
            return tail.data;
        Node node = null;
        if(index <= size/2) {
            for(int i = 0;i < index;i++){
                node = head.next;
            }
        }
        else{
            for(int i = size - 1;i > index;i --){
                node = tail.previous;
            }
        }
        return node.data;
    }
    public boolean add(int index,Integer m) {
        if(index == 0){
            addFirst(m);
            return true;
        }
        if(index == size - 1){
            addLast(m);
            return true;
        }
        if(index < 0||index > size - 1){
            throw new IndexOutOfBoundsException("下标越界");
        }
        Node node = null;
        Node newNode = new Node(m);
        node = head;
        for(int i = 0;i < index - 1;i++){
            node = node.next;
        }
        newNode.previous = node;
        newNode.next = node.next;
```

```
        node.next.previous = newNode;
        node.next = newNode;
        size++;
        return true;
    }
    public Integer remove(int index) {
        if(index == 0){
            return removeFirst();
        }
        if(index == size-1)
            return removeLast();
        if(index<0||index>size-1){
            throw new IndexOutOfBoundsException("下标越界");
        }
        Node node = null;

        if(index<=size/2) {
            node = head;
            for(int i=0;i<index-1;i++){
                node = node.next;
            }
        }
        else{
            node = tail;
            for(int i=size-1;i>index;i--){
                node = node.previous;
            }
        }
        node.previous.next = node.next;
        node.next.previous = node.previous;
        size--;
        return node.data;
    }
    public int size(){
        return size;
    }
    public boolean isEmpty(){
        return size>0;
    }
    public boolean contains(Integer m){
        if(head==null||tail==null)
            return false;
        if(head.data.intValue() == m.intValue()
            ||tail.data.intValue() == m.intValue())
            return true;
        boolean boo = false;
        Node node = null;
        node = head;
        for(int i=0;i<size-1;i++){
            if(node.data.intValue() == m){
                boo = true;
                break;
            }
            node = node.next;
        }
        return boo;
    }
    public void rotate(){
        Integer temp = head.data;
        Node node = head;
        for(int i=0;i<size-1; i++){          //节点的数据依次向左移动
```

```
                node.data = node.next.data;
                node = node.next;
            }
            tail.data = temp;                    //头节点的数据放入尾节点中
        }
        public String toString(){
            Node node = head;
            String str = new String("[ ");
            str = str + node.data + " ";
            for(int i = 0;i < size - 1;i++){
                node = node.next;
                str = str + node.data + " ";
            }
            str = str + "]";
            return str;
        }
    }
```

本例中的主类 Example5_17 使用本例中自己编写的 LinkedInt 类创建链表，并使用了该链表的方法，运行效果如图 5.28 所示。

```
链表大小:8
[ 10 11 12 13 14 15 16 17 ]
头节点:10,尾节点:17.
第6个节点中的数据:16
删除头节点:10,删除尾节点:17.
链表大小:6
[ 11 12 13 14 15 16 ]
插入第3个节点999:true
链表大小:7
[ 11 12 13 999 14 15 16 ]
插入第1个节点888:true
链表大小:8
[ 11 888 12 13 999 14 15 16 ]
删除第5个节点:14.
链表大小:7
[ 11 888 12 13 999 15 16 ]
链表包含888:true
链表包含14:false
```

图 5.28　用自己编写的类创建链表

Example5_17.java

```
public class Example5_17 {
    public static void main(String args[]) {
        LinkedInt list = new LinkedInt();
        list.addFirst(13);
        list.addFirst(12);
        list.addFirst(11);
        list.addFirst(10);
        list.addLast(14);
        list.addLast(15);
        list.addLast(16);
        list.addLast(17);
        System.out.println("链表大小:" + list.size() + "\n" + list);
        System.out.print("头节点:" + list.getFirst());
        System.out.println(",尾节点:" + list.getLast() + ".");
        int index = 6;                           //下标索引从 0 开始
        System.out.println("第" + index + "个节点中的数据:" + list.get(index));
        System.out.print("删除头节点:" + list.removeFirst());
        System.out.println(",删除尾节点:" + list.removeLast() + ".");
        System.out.println("链表大小:" + list.size() + "\n" + list);
        index = 3;
        int number = 999;
        System.out.println("插入第" + index + "个节点" + number + ":" + list.add(index,number));
```

```
            System.out.println("链表大小:" + list.size() + "\n" + list);
            index = 1;
            number = 888;
            System.out.println("插入第" + index + "个节点" + number + ":" + list.add(index,number));
            System.out.println("链表大小:" + list.size() + "\n" + list);
            index = 5;
            System.out.println("删除第" + index + "个节点:" + list.remove(index) + ".");
            System.out.println("链表大小:" + list.size() + "\n" + list);
            number = 888;
            System.out.println("链表包含" + number + ":" + list.contains(number));
            number = 14;
            System.out.println("链表包含" + number + ":" + list.contains(number));
        }
    }
```

例 5-18 编写 LinkedInt 类解决约瑟夫问题。

在例 5-12 中曾使用集合框架中的 LinkedList 类解决了约瑟夫问题,在本例中使用自己编写的 LinkedInt 类解决约瑟夫问题,运行效果如图 5.29 所示。

```
号码3退出圈
号码6退出圈
号码9退出圈
号码1退出圈
号码5退出圈
号码10退出圈
号码4退出圈
号码11退出圈
号码8退出圈
号码2退出圈
最后剩下的号码是7
```

图 5.29 用 LinkedInt 链表解决约瑟夫问题

Example5_18.java

```
public class Example5_18 {
    public static void main(String args[]) {
        LinkedInt list = new LinkedInt();
        for(int i = 1;i <= 11;i++){
            list.addLast(i);
        }
        while(list.size()> 1){                  //圈中的人数超过 1
            list.rotate();                      //向左旋转 list
            list.rotate();
            int m = list.removeFirst();         //数到第 3 个人,该人退出
            System.out.printf("号码 %d 退出圈\n",m);
        }
        System.out.printf("最后剩下的号码是 %d\n",list.getFirst());
    }
}
```

习题 5

扫一扫

习题

扫一扫

自测题