花样贪吃蛇--链表

今日目标：

1：能够说出链表的存储结构和特点

2：能够说出链表的几种分类及各自的存储结构

3：能说出链表和数组的差异

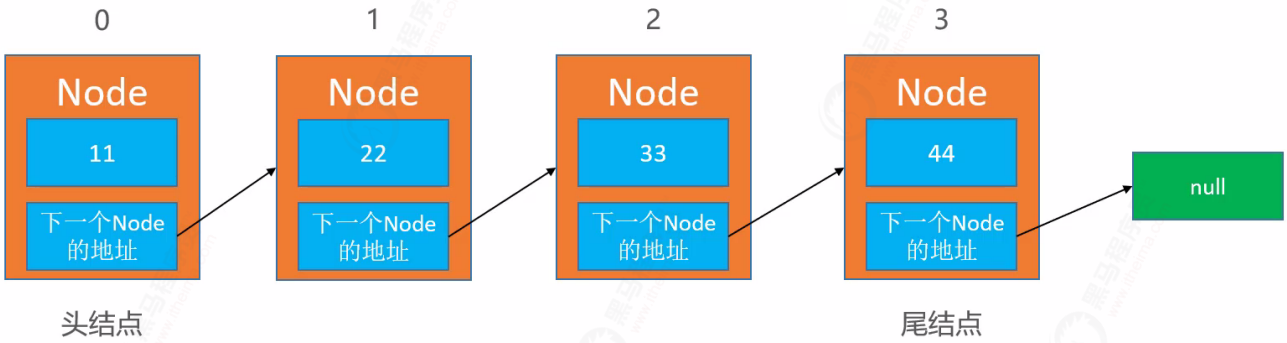
4：完成实战演练题目

5：完成综合案例

1、概念及存储结构

> 问题：思考一下动态数组ArrayList存在哪些弊端？
>
> 1：插入，删除时间复杂度高
>
> 2：可能会造成内存空间的大量浪费
>
> 3：需要一块连续的存储空间，对内存的要求比较高，比如我们要申请一个1000M的数组，如果内存中没有连续的足够大的存储空间则会申请失败，即便内存的剩余可用空间大于1000M，仍然会申请失败。
>
> 结论：能否做到用多少内存空间就申请多少内存？

链表（**Linked list**）是一种物理**存储单元上非连续、非顺序**的存储结构，链表中的每一个元素称之为**结点**（**Node**），结点之间用指针（引用）连接起来，指针的指向顺序代表了结点的逻辑顺序，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。



> 链表能解决数组不能解决的事情吗？
>
> 1：链表天生就具备动态扩容的特点，不需要像动态数组那样先申请一个更大的空间，然后将原空间内的数据拷贝到新空间；能够避免内存空的大量浪费
>
> 2：链表不需要一块连续的内存空间，它通过指针将一组**零散的内存块**串联起来使用，所以如果我们申请一个1000M大小的链表，只要内存剩余的可用空间大于1000M，便不会出现问题。
>
> 但是需注意：存储同样的数据，链表要比数组耗费内存！

2、链表分类

链表根据其结点之间的连接形式我们又可分为：单链表，双向链表，循环链表，双向循环链表

2.1、单链表

单链表就是我们刚刚讲到的链表的最基本的结构，链表通过**指针**将一组零散的内存块串联在一起。。如图所示，我们把这个记录下个结点地址的指针叫作**后继指针 next**，如果链表中的某个节点为p，p的下一个节点为q，我们可以表示为：`p.next=q`

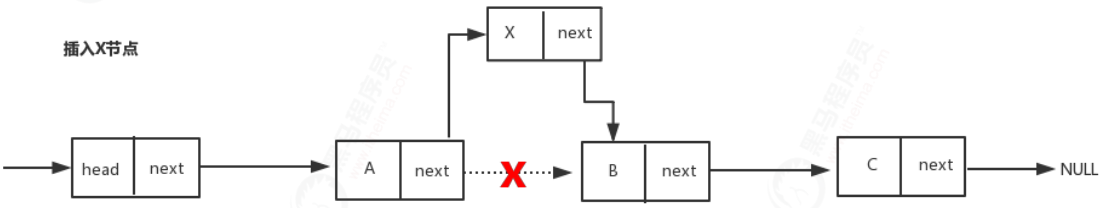
单链表结构：

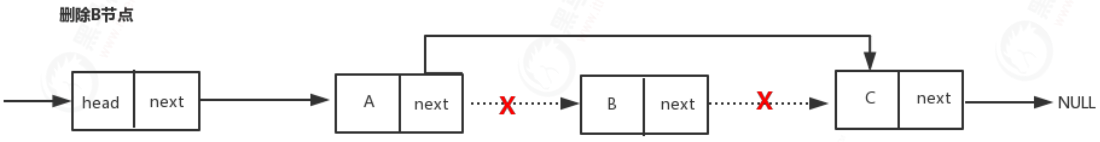
> 链表中有两个结点是比较特殊的，它们分别是第一个结点和最后一个结点。我们习惯性地把第一个结点叫作头结点，把最后一个结点叫作尾结点。其中，头结点用来记录链表的基地址,有了它，我们就可以遍历得到整条链表。而尾结点特殊的地方是：指针不是指向下一个结点，而是指向一个空地址 NULL，表示这是链表上最后一个结点。

与数组一样，链表也支持**数据的查找、插入和删除操作**。

在进行数组的插入、删除操作时，为了保持内存数据的连续性，需要做大量的数据搬移，所以时间复杂度是 O(n)。而在链表中插入或者删除一个数据，我们并不需要为了保持内存的连续性而搬移结点，因为链表的存储空间本身就不是连续的。所以，在链表中插入和删除一个数据是非常快速的。

如图所示，针对链表的插入和删除操作，我们只需要考虑相邻结点的指针改变，所以**插入删除的时间复杂度是O(1)**。

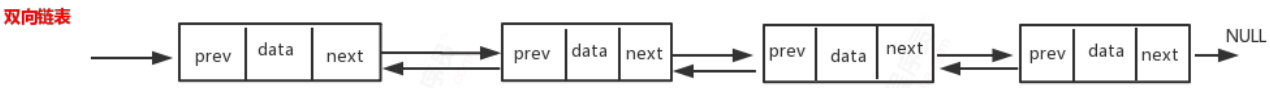




但是，有利就有弊。链表要想随机访问第 k 个元素，就没有数组那么高效了。因为链表中的数据并非连续存储的，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据针一个结点一个结点地依次遍历，直到找到相应的结点，所以，链表随机访问的性能没有数组好，**查询的时间复杂度是O(n)**。

2.2、双向链表

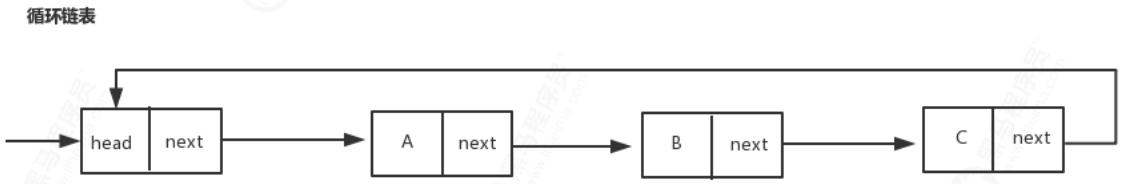
单向链表只有一个方向，结点只有一个后继指针 next。而双向链表，顾名思义，它支持两个方向，**每个结点不止有一个后继指针 next 指向后面的结点，还有一个前驱指针 prev 指向前面的结点**，如图所示



> 从图中可以看出来，双向链表需要额外的两个空间来存储后继结点和前驱结点的地址。所以，如果存储同样多的数据，双向链表要比单链表占用更多的内存空间。虽然两个指针比较浪费存储空间，但可以支持双向遍历，这样也带来了双向链表操作的灵活性，比如
>
> 1：可以在O(1)时间内找到给定结点的前驱节点，而对于单链表则需要O(n)的时间
>
> 2：根据索引来查找元素时可极大提升查找效率
>
> .....
>
> 在很多场景下双向链表都比单向链表更加高效，这就是为什么在实际的软件开发中，双向链表尽管比较费内存，但还是比单链表的应用更加广泛的原因。如果你熟悉 Java 语言，你肯定用过LinkedHashMap 这个容器。如果你深入研究 LinkedHashMap 的实现原理，就会发现其中就用到了双向链表这种数据结构。
>
> 实际上，这里有一个更加重要思想就是*用空间换时间的设计思想*。当内存空间充足的时候，如果我们更加追求代码的执行速度，我们就可以选择空间复杂度相对较高、但时间复杂度相对很低的算法或者数据结构。相反，如果内存比较紧缺，比如代码跑在手机等存储容量小的设备上，这个时候，就要反过来用时间换空间的设计思路。比如最典型的缓存系统就是一个用空间换时间的思想。

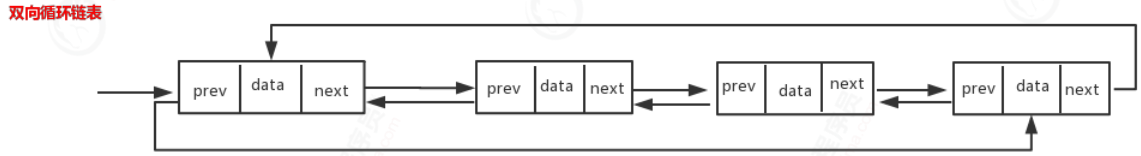
2.3、循环链表

循环链表是一种特殊的单链表。实际上，循环链表也很简单。它跟单链表唯一的区别就在尾结点。我们知道，单链表的尾结点指针指向空地址，表示这就是最后的结点了。而**循环链表的尾结点指针是指向链表的头结点**。从我画的循环链表图中，你应该可以看出来，它像一个环一样首尾相连，所以叫作“循环”链表，和单链表相比，循环链表的优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用循环链表，循环链表的结构如图所示



2.4、双向循环链表

了解了循环链表和双向链表，如果把这两种链表整合在一起就是一个**双向循环链表**



3、小结&实战

3.1、链表数组对比

数组和链表是两种截然不同的内存组织方式。正是因为内存存储的区别，它们插入、删除、随机访问操作的时间复杂度正好相反，下图表明了链表和数组在插入删除和随机访问上时间复杂度的对比

数组和链表性能对比

时间复杂度	数组	链表
插入删除	O(n)	O(1)
随机访问	O(1)	O(n)

3.2、206. 反转链表

字节跳动，腾讯，阿里，美团点评最近面试题，206. 反转链表

高频题

双指针迭代

```
class Solution {
    public ListNode reverseList(ListNode head) {

        ListNode prev = null;
        ListNode curr = head;
        while (curr !=null) {
            ListNode temp = curr.next;
            curr.next = prev;
            prev = curr;
            curr = temp;
        }
        return prev;
    }
}
```

3.3、141. 环形链表

阿里，腾讯，百度最近面试题，141. 环形链表

快慢指针

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        //特殊判断
        if (head == null || head.next == null) {
            return false;
        }
        ListNode fast = head;
        ListNode slow = head;
        //两个指针分别下后走
        while (true) {
            fast = fast.next.next;
            slow = slow.next;

            if (fast == null || fast.next == null) {
                return false;
            }
            if (fast == slow) {
                break;
            }
        }

        return true;
    }
}
```

4、综合案例

4.1、需求

在学习数组的时候我们基于数组实现了一个List容器，支持数据的添加，修改，删除，查询等操作，今天学习完链表之后能够基于链表来实现一个LinkedList容器呢？

要求：

1：要求和动态数组ArrayList具备相同的功能

2：请基于双向链表实现，

4.2、实现

（1）定义接口，`com.itheima.linkedlist.inf.List`，接口方法跟之前实现ArrayList时一样，只不过添加上泛型

```java
package com.itheima.linkedlist.inf;

/**
 * Created by 传智播客*黑马程序员.
 */
public interface List<E> {


    /**
     * 返回容器中元素的个数
     * @return
     */
    int size();

    /**
     * 判断容器是否为空
     * @return
     */
    boolean isEmpty();
    /**
     * 查询元素在容器中的索引下标
     * @param o 元素对象
     * @return  在容器中的下标 不存在则返回-1
     */
    int indexOf(E o);
    /**
     * 判断容器是否包含某个特定的元素
     * @param e
     * @return
     */
    boolean contains(E e);

    /**
     * 将元素添加到容器结尾
     * @param e 要添加的元素
     * @return  是否添加成功
     */
    boolean add(E e);

    /**
     * 向指定位置添加元素
     * @param index    位置下标
     * @param element  元素对象
     */
    void add(int index, E element);

    /**
     * 用指定的元素替换指定位置的数据
     * @param index    指定的位置索引下标
     * @param element   新的元素
     * @return          原始的元素
     */

    E set(int index, E element);
```

```
    /**
     * 获取指定位置的元素
     * @param index   索引下标
     * @return        该位置的元素
     */
    E get(int index);

    /**
     * 移除指定位置的元素
     * @param index 索引下标
     * @return      被移除的元素
     */
    E remove(int index);


    /**
     * 清除容器中所有元素
     */
    void clear();
}
```

（2）创建接口实现：`com.itheima.linkedlist.LinkedList`，实现对应接口

（3）容器要基于双向链表实现，链表是由一个一个结点构成的，因此定义链表结点对象，编写一个静态的内部类

```
//定义链表结点对象
private static class  Node<E>{
    E val;
    Node<E> prev;
    Node<E> next;

    //定义构造
    public Node(Node<E> prev,E val,Node<E> next){
        this.prev = prev;
        this.val = val;
        this.next = next;
    }

}
```

（4）定义相关的成员变量

```
//定义容器中元素的个数
int size;
//定义链表的头结点
Node<E> first;
//定义链表的尾结点
Node<E> last;
```

（5）完成`size,isEmpty,indexOf,contains`等方法的编写

```
@Override
public int size() {
    return size;
}

@Override
public boolean isEmpty() {
    return size == 0;
}

@Override
public int indexOf(E o) {
    int index = 0;
    //分情况 o是否为null，为null和不为null判断的方式不一样，null是用==，不为null用equals
    if (o == null) {

        for ( Node x = first;x!=null;x=x.next) {
            if (x.val == null) {
                return index;
            }
            index++;
        }
    }else {
        for ( Node x = first;x!=null;x=x.next) {
            if (o.equals(x.val)) {
                return index;
            }
            index++;
        }
    }
    return -1;
}

@Override
public boolean contains(E e) {
    return indexOf(e) != -1;
}
```

（6）完成`add`方法的编写

```
@Override
public boolean add(E e) {
    //添加是将元素值添加到链表尾部
    linkLast(e);
    return true;
}

private void linkLast(E e) {
    Node l = last;
    Node newNode = new Node(l,e,null);
    last = newNode;

    if (l==null) {
        //第一次添加
        first = newNode;
    }else {
        l.next = newNode;
    }
    size++;
}

@Override
public void add(int index, E element) {
    //检查索引
    checkIndex(index);
    if (index == size) {
        linkLast(element);
    }else {
        linkBefore(element,node(index));
    }
}

/**
     * 在指定结点前添加一个元素
     * @param element
     * @param node
     */
private void linkBefore(E element, Node<E> node) {
    Node<E> prev = node.prev;
    Node<E> newNode = new Node<E>(prev,element,node);
    node.prev = newNode;

    if (prev == null) {
        first = newNode;
    }else {
        prev.next = newNode;
    }
    size++;
}

/**
     * 查找索引为index的结点
     * @param index
```

```
     * @return
     */
    private Node<E> node(int index){
        //折半查找
        if ( index < (size >> 1)) {
            //从头开始查找
            Node<E> f = first;
            for (int i = 0;i< index;i++) { //i<index注意不能是i<=index 因为当i=index-1时 f=f.next其实f已经指向了索引为index位置的元素了
                f = f.next;
            }
            return f;
        }else {
            //从尾开始查找
            Node<E> l = last;
            for (int i=size-1;i> index;i--) {//同上
                l = l.prev;
            }
            return l;
        }
    }

    private void checkIndex(int index) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("index"+index+",size="+size);
        }
    }
```

（7）完成`set，get`方法的编写

```
/**
     * 替换指定索引位置的元素
     * @param index    指定的位置索引下标
     * @param element   新的元素
     * @return
     */
@Override
public E set(int index, E element) {
    isElementIndex(index);
    Node<E> oldNode = node(index);
    E oldVal = oldNode.val;
    oldNode.val = element;
    return oldVal;
}

private void isElementIndex(int index) {
    if (index < 0 || index >=size) {
        throw new IndexOutOfBoundsException("index="+index+",size="+size);
    }
}

@Override
public E get(int index) {
    isElementIndex(index);
    return node(index).val;
}
```

（8）完成`remove`方法的编写

```java
@Override
public E remove(int index) {
    isElementIndex(index);
    Node<E> node = node(index);
    return unlink(node);
}

private E unlink(Node<E> node) {
    Node<E> prev = node.prev;
    Node<E> next = node.next;
    E val = node.val;
    node.val = null;

    //node是头结点
    if (prev == null) {
        first = next;
    }else {
        prev.next = next;
        node.prev = null;
    }

    //node是尾结点
    if (next == null) {
        last = prev;
    }else {
        next.prev = prev;
        node.next = null;
    }
    size--;
    return val;
}
```

（9）完成`clear，toString`方法的编写

```
@Override
public void clear() {
    for (Node l = first;l!=null;) {
        Node next = l.next;
        l.val = null;
        l.prev = null;
        l.next = null;
        l = next;
    }

    first = last = null;
    size=0;
}
@Override
public String toString() {
    //输出 1->2->Null格式的数据
    StringBuilder stringBuilder = new StringBuilder();
    for (Node l=first;l!=null;l = l.next) {
        stringBuilder.append(l.val).append("->");
    }
    stringBuilder.append("Null");
    return stringBuilder.toString();
}
```

（10）编写测试类：`com.itheima.linkedlist.LinkedListTest`

```
public static void main(String[] args) {
    List list = new LinkedList();
    list.add(1);
    list.add(2);
    list.add(3);
    System.out.println("容器内元素为:"+list); // 1->2->3->Null
    System.out.println("容器内元素个数:"+list.size()+"容器是否为空:"+list.isEmpty());
    System.out.println("容器中是否包含3:"+list.contains(3));
    list.add(0,4);// 4->1->2->3->Null
    System.out.println("容器内元素为:"+list);
    list.add(3,5);//4->1->2->5->3->Null
    System.out.println("容器内元素为:"+list);
    list.add(2,6);//4->1->6->2->5->3->Null
    System.out.println("容器内元素为:"+list);
    System.out.println("获取索引为0的元素:"+list.get(0));
    System.out.println("获取索引为5的元素:"+list.get(5));
    System.out.println("获取索引为2的元素:"+list.get(2));
    list.remove(0);//1->6->2->5->3->Null
    System.out.println("容器内元素为:"+list);
    list.remove(3);//1->6->2->3->Null
    System.out.println("移除后容器内元素为:"+list);
    list.clear();
    System.out.println("清空后为:"+list);
}
```

查看输出结果！

> 容器内元素为:1->2->3->Null 容器内元素个数:3容器是否为空:false 容器中是否包含3:true 容器内元素为:4->1->2->3->Null 容器内元素为:4->1->2->5->3->Null 容器内元素为:4->1->6->2->5->3->Null 获取索引为0的元素:4 获取索引为5的元素:3 获取索引为2的元素:6 容器内元素为:1->6->2->5->3->Null 移除后容器内元素为:1->6->2->3->Null 清空后为:Null