LinkedList与链表

【本节目标】

1. **ArrayList的缺陷**
2. **链表**
3. **链表相关oj**
4. **LinkedList的使用**
5. **LinkedList的模拟实现**
6. **ArrayList和LinkedList的区别**

1. ArrayList的缺陷

上节课已经熟悉了ArrayList的使用，并且进行了简单模拟实现。通过源码知道，ArrayList底层使用数组来存储元素：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{

    // ...

    // 默认容量是10
    private static final int DEFAULT_CAPACITY = 10;

    //...

    // 数组：用来存储元素
    transient Object[] elementData; // non-private to simplify nested class access

    // 有效元素个数
    private int size;


    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }

    // ...
```

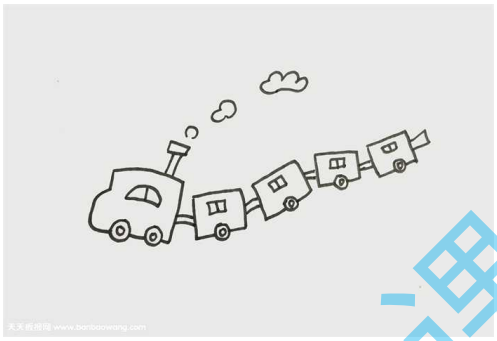
```
}
```

由于其底层是一段连续空间，当**在ArrayList任意位置插入或者删除元素时，就需要将后序元素整体往前或者往后搬移，时间复杂度为O(n)**，效率比较低，因此**ArrayList不适合做任意位置插入和删除比较多的场景。**因此：java集合中又引入了LinkedList，即链表结构。

2. 链表

2.1 链表的概念及结构

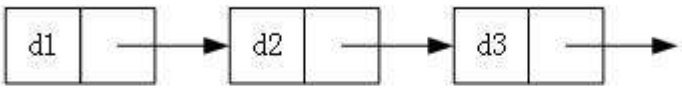
链表是一种**物理存储结构上非连续**存储结构，数据元素的**逻辑顺序**是通过链表中的**引用链接**次序实现的 。

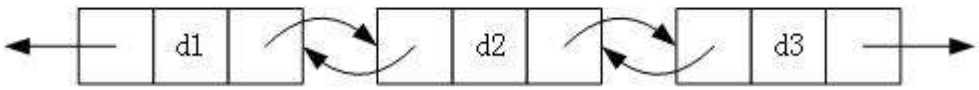


注意：
1. 从上图可看出，链式结构在逻辑上是连续的，但是在物理上不一定连续
2. 现实中的结点一般都是从堆上申请出来的
3. 从堆上申请的空间，是按照一定的策略来分配的，两次申请的空间可能连续，也可能不连续

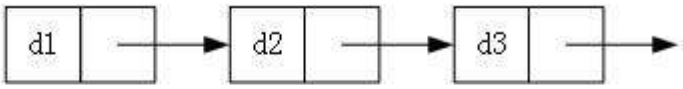
实际中链表的结构非常多样，以下情况组合起来就有8种链表结构：

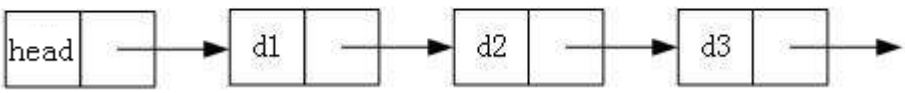
1. **单向或者双向**



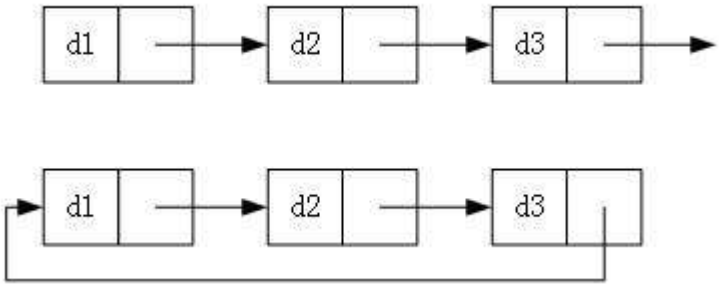


2. **带头或者不带头**



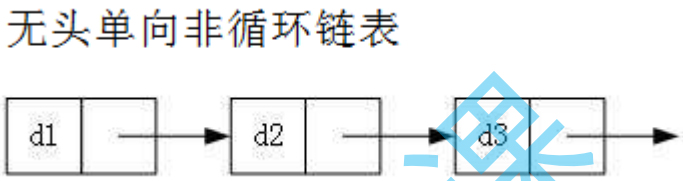


3. **循环或者非循环**



虽然有这么多的链表的结构，但是我们重点掌握两种:

- **无头单向非循环链表**：**结构简单**，一般不会单独用来存数据。实际中更多是作为**其他数据结构的子结构**，如哈希桶、图的邻接表等等。另外这种结构在**笔试面试**中出现很多。



- **无头双向链表**：在Java的集合框架库中LinkedList底层实现就是无头双向循环链表。

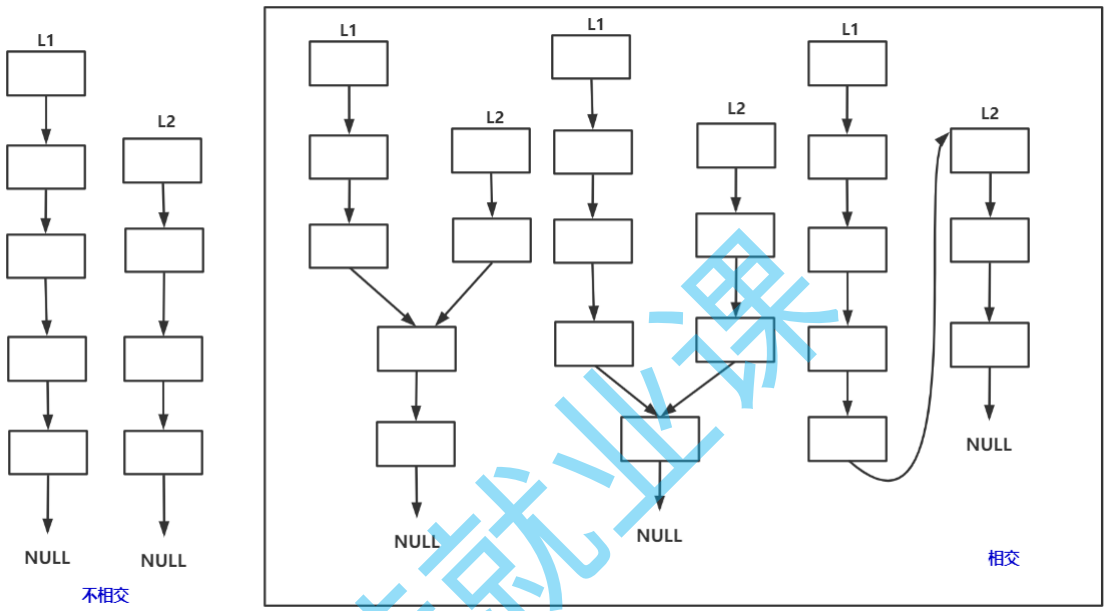
2.2 链表的实现

```
// 1、无头单向非循环链表实现
public class SingleLinkedList {
    //头插法
    public void addFirst(int data);
    //尾插法
    public void addLast(int data);
    //任意位置插入,第一个数据节点为0号下标
    public boolean addIndex(int index,int data);
    //查找是否包含关键字key是否在单链表当中
    public boolean contains(int key);
    //删除第一次出现关键字为key的节点
    public void remove(int key);
    //删除所有值为key的节点
    public void removeAllKey(int key);
    //得到单链表的长度
    public int size();
    public void display();
    public void clear();
}
```

3.链表面试题

1. 删除链表中等于给定值 **val** 的所有节点。 OJ链接
2. 反转一个单链表。 OJ链接

3. 给定一个带有头结点 head 的非空单链表，返回链表的中间结点。如果有两个中间结点，则返回第二个中间结点。OJ链接
4. 输入一个链表，输出该链表中倒数第k个结点。 OJ链接
5. 将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。OJ链接
6. 编写代码，以给定值x为基准将链表分割成两部分，所有小于x的结点排在大于或等于x的结点之前 。OJ链接
7. 链表的回文结构。OJ链接
8. 输入两个链表，找出它们的第一个公共结点。OJ链接



9. 给定一个链表，判断链表中是否有环。OJ链接

【思路】

快慢指针，即慢指针一次走一步，快指针一次走两步，两个指针从链表其实位置开始运行，如果链表

带环则一定会在环中相遇，否则快指针率先走到链表的末尾。比如：陪女朋友到操作跑步减肥。

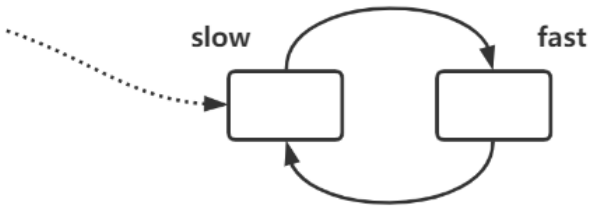
【扩展问题】

- **为什么快指针每次走两步，慢指针走一步可以?**

  假设链表带环，两个指针最后都会进入环，快指针先进环，慢指针后进环。当慢指针刚进环时，可能就和快指针相遇了，最差情况下两个指针之间的距离刚好就是环的长度。此时，两个指针每移动一次，之间的距离就缩小一步，不会出现每次刚好是套圈的情况，因此：在满指针走到一圈之前，快指针肯定是可以追上慢指针的，即相遇。

- **快指针一次走3步，走4步，...n步行吗?**
-

假设：快指针每次走3步，满指针每次走一步，此时快指针肯定先进环，慢指针后来才进环。假设慢指针进环时候，快指针的位置如图所示：



此时按照上述方法来绕环移动，每次快指针走3步，慢指针走1步，是永远不会相遇的，快指针刚好将慢指针套圈了，因此不行。

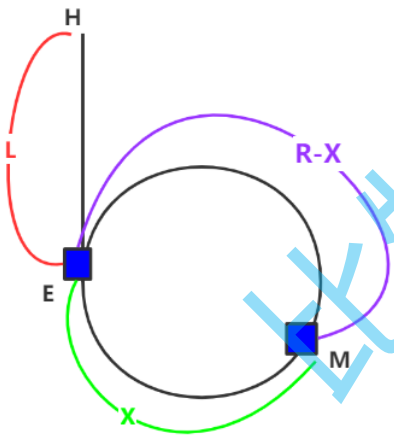
只有快指针走2步，慢指针走一步才可以，因为换的最小长度是1，即使套圈了两个也在相同的位置。

10. 给定一个链表，返回链表开始入环的第一个节点。 如果链表无环，则返回 NULL OJ链接

- **结论**

  让一个指针从链表起始位置开始遍历链表，同时让一个指针从判环时相遇点的位置开始绕环运行，两个指针都是每次均走一步，最终肯定会在入口点的位置相遇。

- **证明**



说明：
H为链表的起始点，E为环入口点，M与判环时候相遇点
设：
环的长度为R，H到E的距离为L　E到M的距离为X
则：M到E的距离为R - X

在判环时，快慢指针相遇时所走的路径长度：
fast: L +X + nR
slow: L + X

注意：
1. 当慢指针进入环时，快指针可能已经在环中绕了n圈了，n至少为1
因为：快指针先进环走到M的位置，最后又在M的位置与慢指针相遇

2. 慢指针进环之后，快指针肯定会在慢指针走一圈之内追上慢指针
因为：慢指针进环后，快慢指针之间的距离最多就是环的长度，而两个指针在移动时，每次它们至今的距离都缩减一步，因此在慢指针移动一圈之前快指针肯定是可以追上慢指针的

而快指针速度是满指针的两倍，因此有如下关系是：
2 * (L + X) = L + X + nR
L + X = nR
L = nR - X　（n为1,2,3,4……，n的大小取决于环的大小，环越小n越大）

极端情况下，假设n = 1，此时：L = R - X

即：一个指针从链表起始位置运行，一个指针从相遇点位置绕环，每次都走一步，两个指针最终会在入口点的位置相遇

11. 其他 。ps：链表的题当前因为难度及知识面等等原因还不适合我们当前学习，以后大家自己下去以后 Leetcode OJ链接 + 牛客 OJ链接

4. LinkedList的模拟实现

LinkedList底层就是一个双向链表，我们来实现一个双向链表。

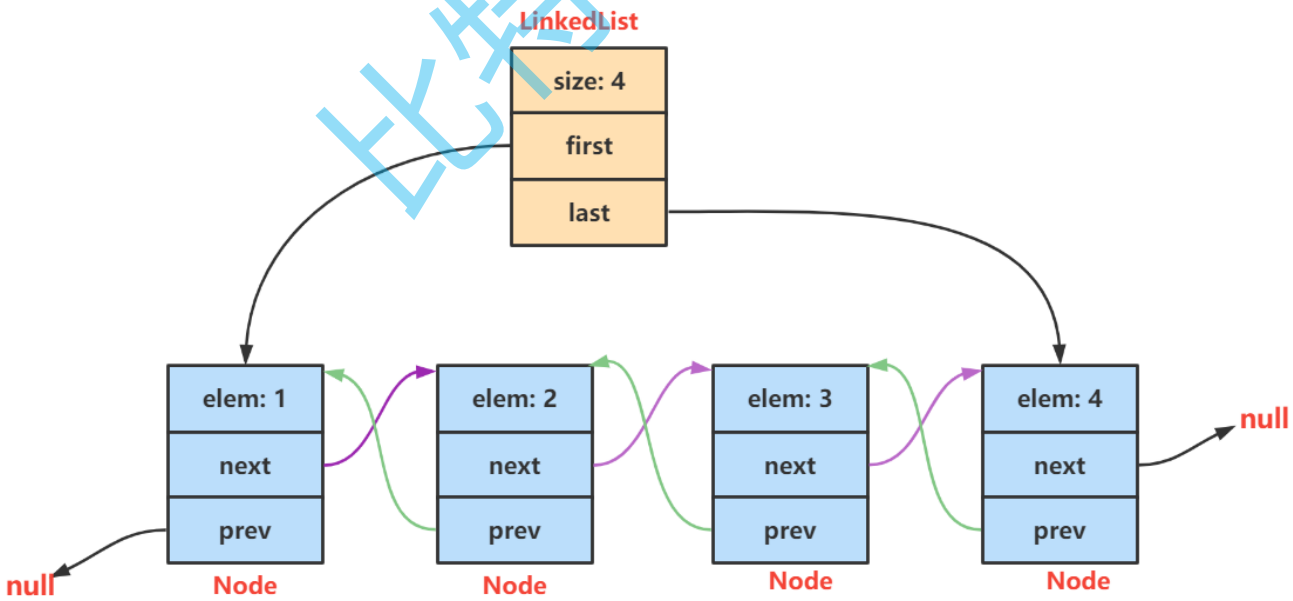
```
// 2、无头双向链表实现
public class MyLinkedList {
    //头插法
    public void addFirst(int data);
    //尾插法
    public void addLast(int data);
    //任意位置插入,第一个数据节点为0号下标
    public boolean addIndex(int index,int data);
    //查找是否包含关键字key是否在单链表当中
    public boolean contains(int key);
    //删除第一次出现关键字为key的节点
    public void remove(int key);
    //删除所有值为key的节点
    public void removeAllKey(int key);
    //得到单链表的长度
    public int size();
    public void display();
    public void clear();
}
```

5.LinkedList的使用

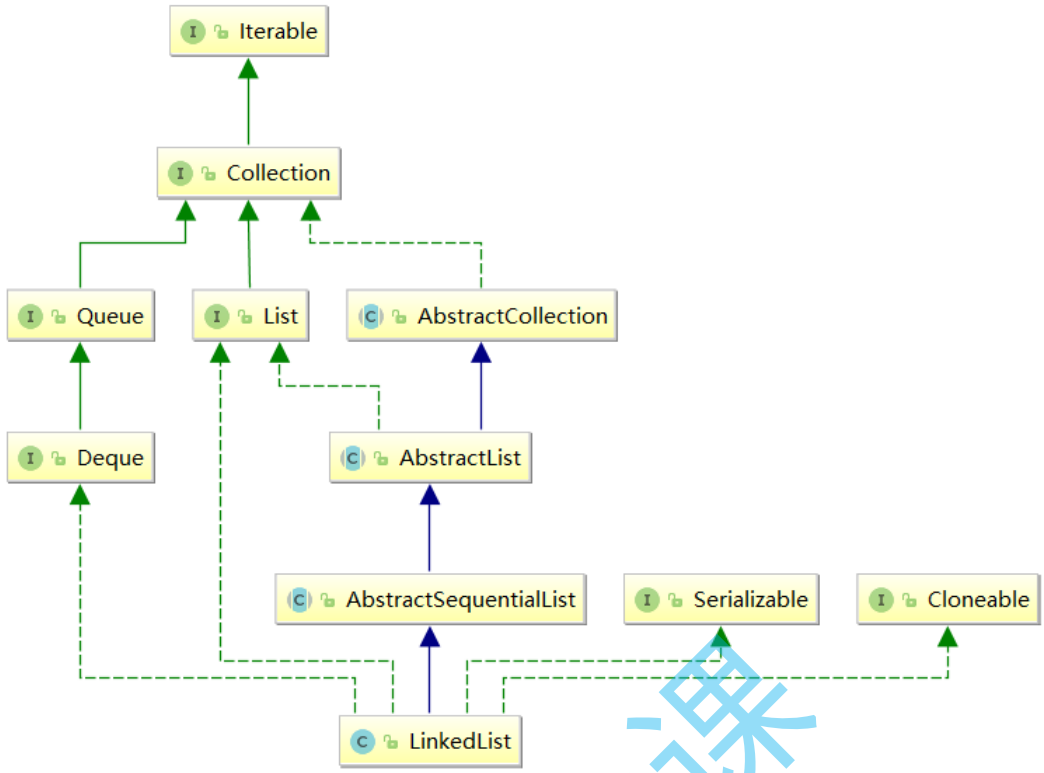
5.1 什么是LinkedList

LinkedList 的官方文档

LinkedList的底层是双向链表结构(链表后面介绍)，由于链表没有将元素存储在连续的空间中，元素存储在单独的节点中，然后通过引用将节点连接起来了，因此在在任意位置插入或者删除元素时，不需要搬移元素，效率比较高。



在集合框架中，LinkedList也实现了List接口，具体如下：



【说明】

1. LinkedList实现了List接口
2. LinkedList的底层使用了双向链表
3. LinkedList没有实现RandomAccess接口，因此LinkedList不支持随机访问
4. LinkedList的任意位置插入和删除元素时效率比较高，时间复杂度为O(1)

5.2 LinkedList的使用

1. **LinkedList的构造**

方法	解释
LinkedList()	无参构造
public LinkedList(Collection<? extends E> c)	使用其他集合容器中元素构造List

```
public static void main(String[] args) {
    // 构造一个空的LinkedList
    List<Integer> list1 = new LinkedList<>();

    List<String> list2 = new java.util.ArrayList<>();
    list2.add("JavaSE");
    list2.add("JavaWeb");
    list2.add("JavaEE");
    // 使用ArrayList构造LinkedList
    List<String> list3 = new LinkedList<>(list2);
}
```

2. **LinkedList的其他常用方法介绍**

方法	解释
boolean add(E e)	尾插 e
void add(int index, E element)	将 e 插入到 index 位置
boolean addAll(Collection<? extends E> c)	尾插 c 中的元素
E remove(int index)	删除 index 位置元素
boolean remove(Object o)	删除遇到的第一个 o
E get(int index)	获取下标 index 位置元素
E set(int index, E element)	将下标 index 位置元素设置为 element
void clear()	清空
boolean contains(Object o)	判断 o 是否在线性表中
int indexOf(Object o)	返回第一个 o 所在下标
int lastIndexOf(Object o)	返回最后一个 o 的下标
List<E> subList(int fromIndex, int toIndex)	截取部分 list

```
public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.add(1);   // add(elem): 表示尾插
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);
    list.add(6);
    list.add(7);
    System.out.println(list.size());
    System.out.println(list);

    // 在起始位置插入0
    list.add(0, 0);  // add(index, elem): 在index位置插入元素elem
    System.out.println(list);

    list.remove();       // remove(): 删除第一个元素，内部调用的是removeFirst()
    list.removeFirst();    // removeFirst(): 删除第一个元素
    list.removeLast();    // removeLast(): 删除最后元素
    list.remove(1);  // remove(index): 删除index位置的元素
    System.out.println(list);

    // contains(elem): 检测elem元素是否存在，如果存在返回true，否则返回false
    if(!list.contains(1)){
        list.add(0, 1);
```

```
    }
    list.add(1);
    System.out.println(list);
    System.out.println(list.indexOf(1));   // indexOf(elem): 从前往后找到第一个elem的位置
    System.out.println(list.lastIndexOf(1));  // lastIndexOf(elem): 从后往前找第一个1的位置
    int elem = list.get(0);    // get(index): 获取指定位置元素
    list.set(0, 100);          // set(index, elem): 将index位置的元素设置为elem
    System.out.println(list);

    // subList(from, to): 用list中[from, to)之间的元素构造一个新的LinkedList返回
    List<Integer> copy = list.subList(0, 3);
    System.out.println(list);
    System.out.println(copy);
    list.clear();              // 将list中元素清空
    System.out.println(list.size());
}
```

3. **LinkedList的遍历**

```
public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList<>();
    list.add(1);   // add(elem): 表示尾插
    list.add(2);
    list.add(3);
    list.add(4);
    list.add(5);
    list.add(6);
    list.add(7);
    System.out.println(list.size());
    // foreach遍历
    for (int e:list) {
        System.out.print(e + " ");
    }
    System.out.println();
    // 使用迭代器遍历---正向遍历
    ListIterator<Integer> it = list.listIterator();
    while(it.hasNext()){
        System.out.print(it.next()+ " ");
    }
    System.out.println();
    // 使用反向迭代器---反向遍历
    ListIterator<Integer> rit = list.listIterator(list.size());
    while (rit.hasPrevious()){
        System.out.print(rit.previous() +" ");
    }
    System.out.println();
}
```


6. ArrayList和LinkedList的区别

不同点	ArrayList	LinkedList
存储空间上	物理上一定连续	逻辑上连续，但物理上不一定连续
随机访问	支持O(1)	不支持：O(N)
头插	需要搬移元素，效率低O(N)	只需修改引用的指向，时间复杂度为O(1)
插入	空间不够时需要扩容	没有容量的概念
应用场景	元素高效存储+频繁访问	任意位置插入和删除频繁