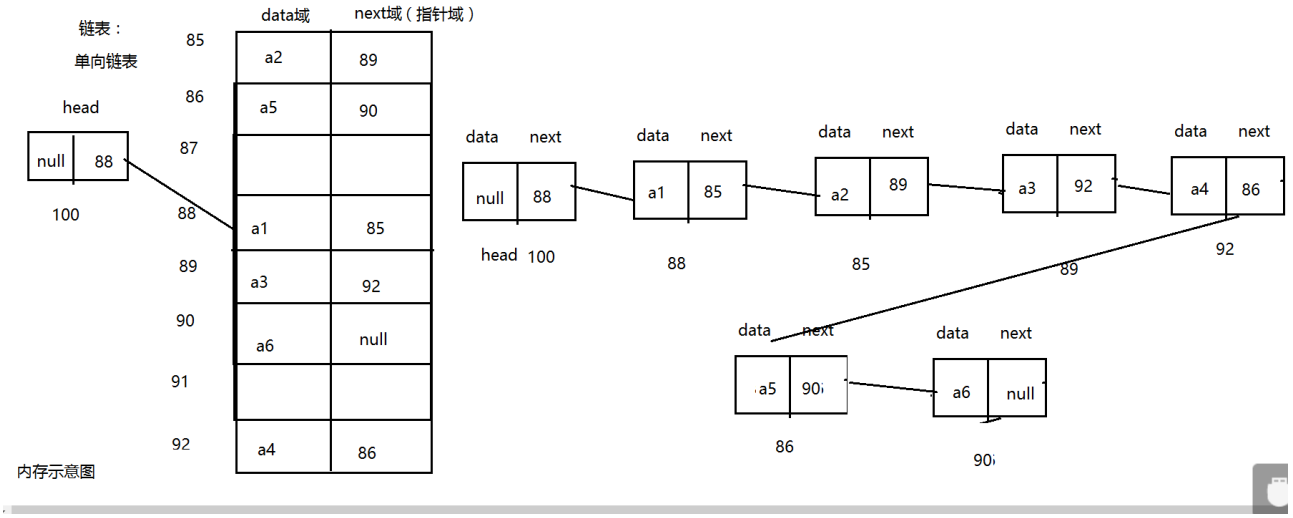
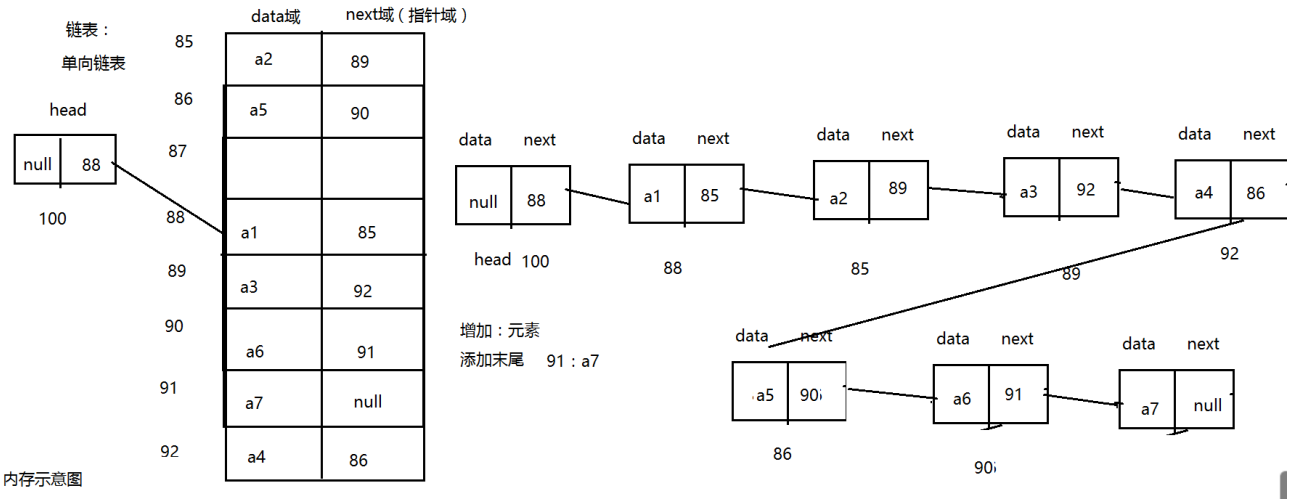
* 今天学习目标
  * 能够理解数据结构之链表
  * 能够实现自定义LinkedList
  * 能够实现自定义ArrayList
  * 能够说出ArrayList和LinkedList区别

---

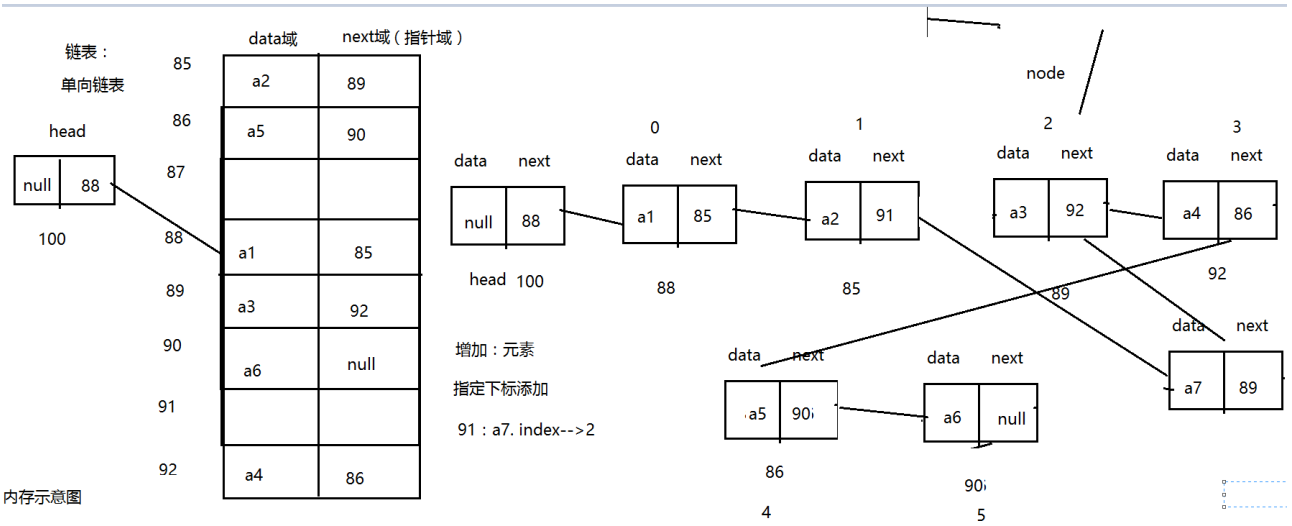
  * 回顾
    * 数据结构：
      * 存储，组织数据的方式
      * 代码写更优雅，更高效
    * 栈：FILO,Stack,方法栈
    * 队列：FIFO,Queue,LinkedList
    * 数组：长度不变，连续空间，增删慢
      * 稀疏数组
    * 链表，LinkedList
  * 能够理解数据结构之链表
    * 链表英文为：Linked List
    * 链表是由一系列结点node（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。我们常说链表有两种：单向链表和双向链表。
      * 手拉手
    * 链表可以带头节点也不带，看具体的需求
    * 链表各个节点不一定是连续存储的。
    * 多个结点之间，通过地址进行连接。
    * 特点：查询慢，增删快。
    * 单链表示意图



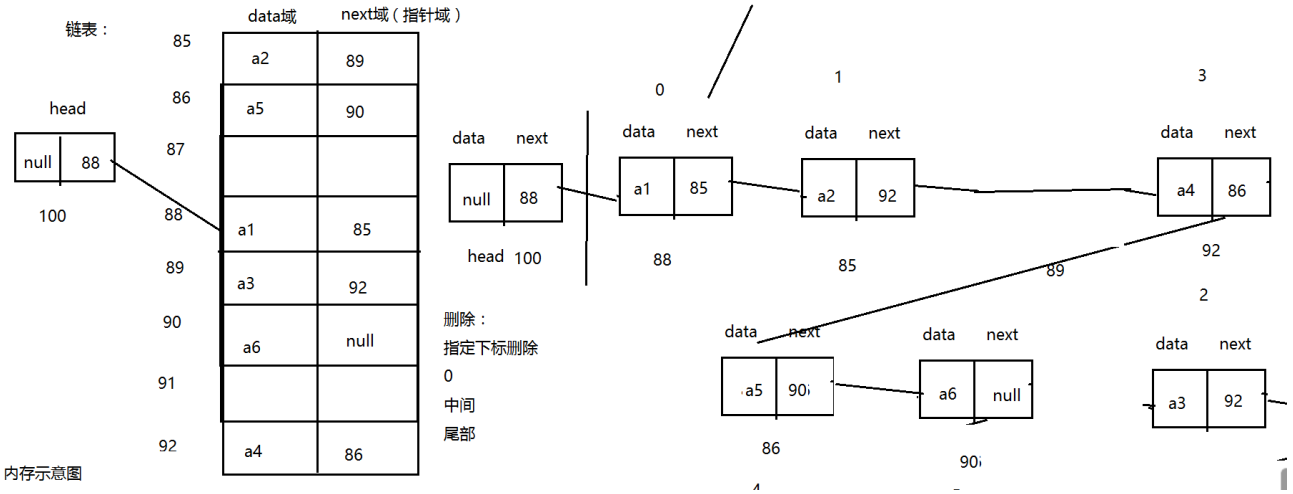
* 增快（添加到末尾）



* 增快（添加指定位置）



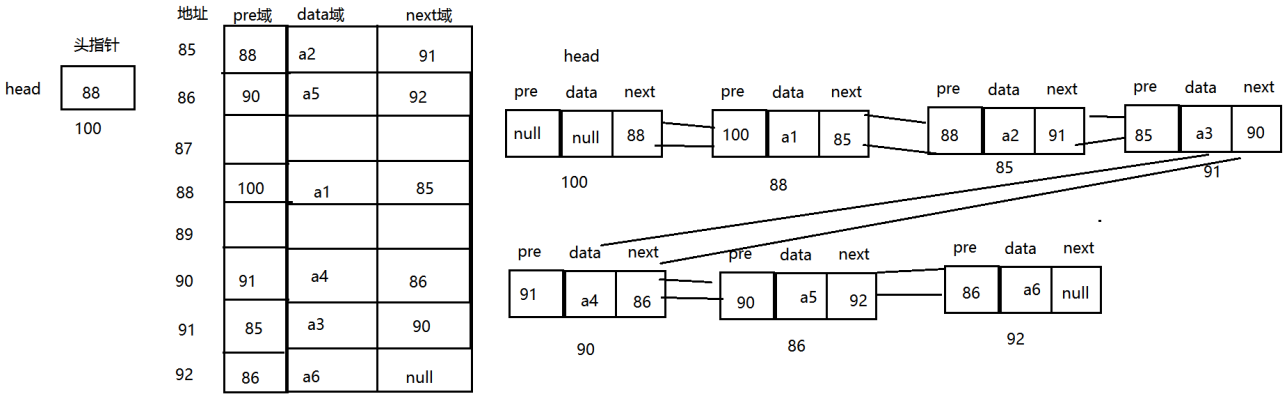
* 删快



* 双向链表

  * 单向链表只能往一个方向查找。

  * 双向链表可以双向查找



* 能够实现自定义LinkedList

```
* 需要实现功能
  * 添加节点（默认添加和指定位置添加）
    boolean add(E e)
    boolean add(int index,E e)
  * 访问某一个节点
    E get(int index);
  * 删除节点
    E remove(int index);
  * 获得链表的长度大小
    int size();
  * 判断链表是否为空
    boolean isEmpty();
```

```
 * 清空链表
   void clear();

思路：
* 新建LinkedList类
   * 无参数构造器
* 新建LinkedList内部类Node
   * 前一个节点（指针域）
   * 当前节点的data
   * 后一个节点（指针域）
   * 构造器（三个参数的）
* 在LinkedList类
   * 链表的大小
   * 第一个节点
   * 最后一个节点
*  获得链表的长度大小size()
   * 直接返回size
*  判断链表是否为空
   * 返回size==0;
* 添加节点（默认添加）boolean add(E e)
  * 添加到最后方法编写linkLast(e);
    * 先保存最后一个节点
    * 构建需要添加的节点（它的前一个节点就是上一次最后节点）
    * 判断上一次最后节点是否空
       * 假如为空，代表第一次添加,把新节点复制给first
       * 假如不为空，上一次最后节点next指向新的节点
    * 新的节点变成最后节点
    * size 加一
  * 直接返回true

* 添加节点（指定位置）
   * 检测下标是否在这个链表添加合理位置
     * checkPositionIndex
       * isPositionIndex：是否合理：不合理抛IndexOutOfBoundsException
         * index>=0 && index<=size
   * 判断是否要加到最后还是插入到某个位置
     * 判断条件：size==index
       * 假如要添加到最后：调用 linkLast(e);
       * 假如插入到某个位置
         * 找到下标已经存在节点（Node<E> node=node(index);）
```

```
            * 判断条件：index < ( size>>1 )
            * 假如这个下标在一半左边，就向前找
                * 从第一个开始向前查找：迭代查找
                  Node<E> node=first;
                    for (int i = 0; i < index; i++) {
                        node=node.next;
                    }
                    return node;
            * 假如这个下标在一半右边，就向后找
                * 从最后一个向后查找：迭代查找
                 Node<E> node=last;
                 for (int i=size-1; i > index; i--) {
                     node=node.prev;
                 }
                 return node;
        * 链接到这个节点前面linkBefore
            * 保存下标已经存在节点的前一个节点（pred）
            * 构建新的节点
            * 新的节点重新指向
            * 判断pred是否为null
              * 假如pred为null意味下标已经存在节点的前一个节点为空,意味要插入就是第一
                 * first=newNode;
              * 假如pred不为null新的节点重新指向
                 * pred.next=newNode;
            * size加一
        * 返回true

* 访问某一个节点（E get(int index)）
  * 检查下标是否在合理的范围 checkElementIndex
      * isElementIndex ：假如不合理：IndexOutOfBoundsException
        * 判断条件：index>=0 && index<size;
  * 接通过下标查找节点：node
* 删除节点
  * 检查下标是否在合理的范围： isElementIndex
  * 断开连接并获取删除对象 E unLink(int index)
   * 获得下标对应节点
      * Node<E> node=node(index);
   * 获得下标对应对象
      * E e=node.e;
   * 获得下标对应节点前节点
```

```
        * Node<E> prev=node.prev;
    * 获得下标对应节点后节点
        * Node<E> next=node.next;
    * 判断prev是否null
        * 假如prev为null意味着下标对应节点是第一个节点，把下一个节点变成第一节点
            * first=next;
        * 假如prev不为null：重新指向
            * prev.next=next;
              node.next=null;
    * 判断next是否为null
        * 假如next为null意味着下标对应节点是最后一个节点，把上一个节点作为最后一个节点
            * last=prev;
        * 假如next不为null：重新指向
            * next.prev=prev;
              node.prev=null;
    * 释放e
      * node.e=null;
    * size 减一
      * size--;
    * 返回e
  * 清空链表 void clear
    * 从第一个节点开始迭代
      * for(Node<E> e=first;e!=null;)
    * 先获取它下一节点保存起来，再释放资源
        * Node<E> next=e.next;
    * 释放资源
        * e.e=null;
          e.prev=null;
          e.next=null;
    * e 重新指向下一个
        * e=next;
   * fisrt,last 释放资源
       first=null;
       last=null;
   * size 重置为0
      size=0;

代码：
public class LinkedList<E> {
    private int size=0;// 链表的大小
```

```
133     private Node<E> first;// 第一个节点
134     private Node<E> last;// 最后一个节点
135 
136     public LinkedList() {
137 
138     }
139     /**
140      * @param e 添加的元素
141      * @return 是否添加成功
142      */
143     public boolean add(E e) {
144         // 添加到最后
145         linkLast(e);
146         return true;
147     }
148 
149     public boolean add(int index,E e) {
150         // 1 检测下标是否在这个链表添加合理位置
151         checkPositionIndex(index);
152         // 2 判断是否要加到最后还是插入到某个位置
153         if(index==size) {
154             //加到最后
155             linkLast(e);
156         }else {
157             //插入到某个位置
158               // 找到下标已经存在节点
159             Node<E> node=node(index);
160               // 链接到这个节点前面
161             linkBefore(e,node);
162         }
163         return true;
164     }
165 
166     private void linkBefore(E e, Node<E> node) {
167         //保存下标已经存在节点的前一个节点
168         Node<E> pred=node.prev;
169         //构建新的节点
170         Node<E> newNode=new Node<E>(pred,e,node);
171         // 新的节点重新指向
172         node.prev=newNode;
```

```
        if(pred==null) {
            // 下标已经存在节点的前一个节点为空,意味插入就是第一个简单
            first=newNode;
        }else {
            pred.next=newNode;
        }
        // size++
        size++;

    }
    private Node<E> node(int index) {
        // 查找方式:
        if(index<(size>>1)) {// size>>1--等价size=size/2,使用位移性能会高
             // 假如这个下标在一半左边，就向前找
             // 从第一个开始向前查找：迭代查找
            Node<E> node=first;
            for (int i = 0; i < index; i++) {
                node=node.next;
            }
            return node;
        }else {
             // 假如这个下标在一半右边，就向后找
            Node<E> node=last;
            for (int i=size-1; i > index; i--) {
                node=node.prev;
            }
            return node;
        }
    }
    private void checkPositionIndex(int index) {
        if(!isPositionIndex(index)) {
            throw new IndexOutOfBoundsException("Index:"+index+",Size:"+size);
        }
    }
    private boolean isPositionIndex(int index) {
        return index>=0 && index<=size;
    }

    /**
     * @param index:需要查找下标的节点
```

```
213      * @return
214      */
215     public E get(int index) {
216         // 检查下标是否在合理的范围
217         checkElementIndex(index);
218         // 直接通过下标查找节点
219         return node(index).e;
220     }
221 
222     private void checkElementIndex(int index) {
223         if(!isElementIndex(index)) {
224             throw new IndexOutOfBoundsException("Index:"+index+",Size:"+size);
225         }
226     }
227     private boolean isElementIndex(int index) {
228         return index>=0 && index<size;
229     }
230 
231     public E remove(int index) {
232         // 检查下标是否在合理的范围
233         checkElementIndex(index);
234         // 断开连接并获取删除对象
235         return unLink(index);
236     }
237 
238     /**
239      * @param index
240      * @return
241      * 需要删除对象，断开连接
242      */
243     private E unLink(int index) {
244         // 1 获得下标对应节点
245         Node<E> node=node(index);
246         // 2 获得下标对应对象
247         E e=node.e;
248         // 3 获得下标对应节点前节点
249         Node<E> prev=node.prev;
250         // 4 获得下标对应节点后节点
251         Node<E> next=node.next;
252         // 5 判断prev是否null
```

```
253         if(prev==null) {
254             //假如prev为null意味着下标对应节点是第一个节点,
255             //把下一个节点变成第一节点
256             first=next;
257         }else {
258             // 重新指向
259             prev.next=next;
260             prev.next=null;
261         }
262         // 6 判断next是否为null
263         if(next==null) {
264             //假如next为null意味着下标对应节点是最后一个节点
265             //把上一个节点作为最后一个节点
266             last=prev;
267         }else {
268             // 重新指向
269             next.prev=prev;
270             next.prev=null;
271         }
272         // 释放e
273         node.e=null;
274         // size 减一
275         size--;
276         return e;
277     }
278     /**
279      * @param e
280      */
281     private void linkLast(E e) {
282         // 1 先保存最后一个节点
283         Node<E> l=last;
284         // 2 构建需要添加的节点（它的前一个节点就是上一次最后节点）
285         Node<E> newNode=new Node<E>(last,e,null);
286         // 3 判断上一次最后节点是否空
287         if(l==null) {
288           // 假如为空，代表第一次添加,把新节点复制给first
289           first=newNode;
290         }else {
291             // 假如不为空，上一次最后节点next指向新的节点
292             last.next=newNode;
```

```
        }
        // 新的节点变成最后节点
        last = newNode;
        // size 加一
        size++;
    }

    /**
     * 清空
     */
    public void clear() {
        // 从第一个节点开始
        for(Node<E> e=first;e!=null;) {
            // 先获取它下一节点保存起来，再释放资源
            Node<E> next=e.next;
            // 释放资源
            e.e=null;
            e.prev=null;
            e.next=null;
            // e 重新指向下一个
            e=next;
        }
        // fisrt,last 释放资源
        first=null;
        last=null;
        // size 重置为0
        size=0;
    }
    /**
     * @return
     * 获得链表的长度大小
     */
    public int size() {
        return size;
    }

    /**
     * @return
     * 判断链表是否为空
     */
```

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

    /**
     * @author xiaozhao
     * 链接的节点
     * @param <E>
     */
    private class Node<E>{
        /**
         * 前一个节点（指针域）
         */
        Node<E> prev;
        /**
         * 当前节点的data
         */
        E e;
        /**
         * 后一个节点（指针域）
         */
        Node<E> next;
        public Node(Node<E> prev, E e, Node<E> next) {
            super();
            this.prev = prev;
            this.e = e;
            this.next = next;
        }

    }
}

    public static void main(String[] args) {
        LinkedList<String> list=new LinkedList<String>();
        list.add("刘备");
        list.add("关羽");
        list.add("张飞");
        list.add(1,"关羽11");
        System.out.println(list.size());
        System.out.println(list.get(0));
```

```
        System.out.println(list.get(1));
        System.out.println(list.get(2));
        System.out.println(list.get(3));
        System.out.println(list.isEmpty());
        System.out.println(list.remove(0));
        System.out.println(list.size());
        list.clear();
        System.out.println(list.isEmpty());
        System.out.println(list.size());
    }
 * 结果
 4
刘备
关羽11
关羽
张飞
false
刘备
3
true
0

* 查看LinkedList部分源码
```

* 能够实现自定义ArrayList

* 能够说出ArrayList和LinkedList区别