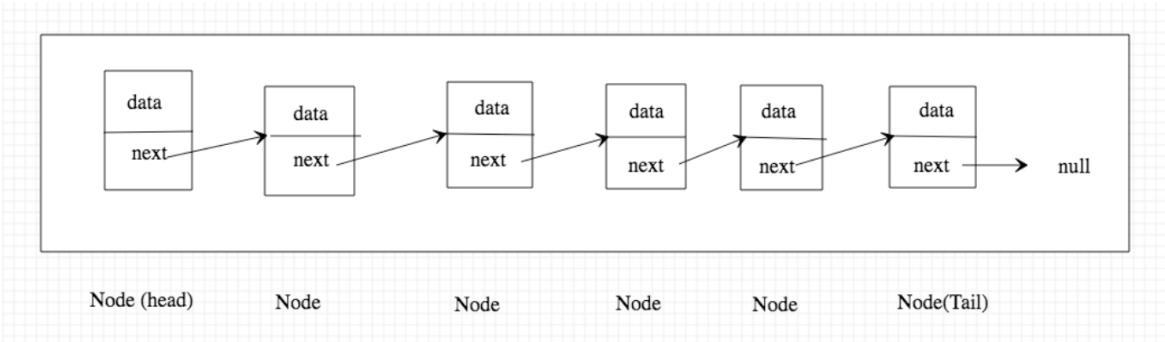
定义

线性表的数据元素可以存储在随意的存储单元，每一个节点不仅仅包括数据元素还有指向下一个节点的指针

分类

链式线性表在存储结构上可以分为单向链表、双向链表和循环链表
链式线性表在顺序上也可以分为有序的和无序的

单向链表



单链表是链表中结构最简单的。一个单链表的节点(Node)分为两个部分，第一个部分(data)保存节点的信息，另一个部分存储下一个节，分指向空值

代码实现

```java
package day1.单链表;

public class SingleLink<E> {

    private int size = 0;       //链表节点数量(包含头结点)
    private Node head;          //链表的头结点

    static class Node<E>{
        private E data;     // 节点数据
        private Node next;  // 下一个节点

        public Node(E e){
            this.data = e;
        }

        @Override
        public String toString() {
            return data.toString();
        }
    }

    /**
     * 构造函数，初始化  头结点
     */
    public SingleLink(){
        Node<E> node = new Node<>(null);
        node.next = null;
        head = node;
        size++;
    }

    /**
     * 根据节点 数据  获取在链表的下标值（从零开始）
     * @param e
     * @return
     */
    public int find(E e){
        if (e == null){
            throw new NullPointerException("数据不能为空");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (e.equals(tmp.data)){
                return cnt;
            }
        }
        return -1;
    }

```

```
    /**
     * 根据下标获取链表的元素
     * @param index
     * @return
     */
    public Node get(int index){
        if (index < 0 || index >= size){
            throw new IndexOutOfBoundsException("元素位置不合法");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (cnt == index){
                return tmp;
            }
        }
        return null;
    }

    /**
     * addFirst
     * @param e
     */
    public void addFirst(E e){
        Node node = new Node(e);
        node.next = head.next;
        head.next = node;
        size++;
    }

    /**
     * addLast
     * @param e
     */
    public void addLast(E e){
        Node tmp = head;
        while (tmp.next != null){
            tmp = tmp.next;
        }
        tmp.next = new Node(e);
        size++;
    }

    /**
     * 根据下标移除
      * @param index
      */
     public void remove(int index){
         if (index < 1 || index >= size){
             throw new IndexOutOfBoundsException("元素位置不合法");
         }
         Node curr = get(index);
```

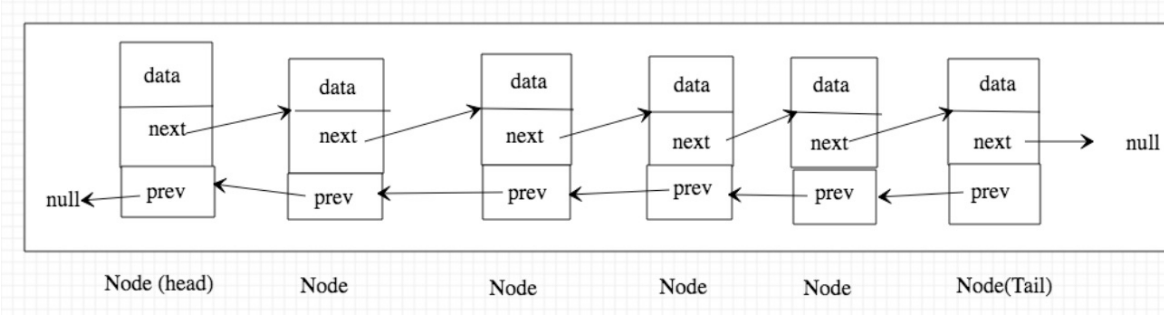
```
        if (index == 1){// 移除第一个节点
            head.next = curr.next;
        }else {
            Node prev = get(index - 1);
            prev.next = curr.next;
        }
        size--;
    }

    /**
     * 根据节点的 值 移除该节点
     * @param e
     * @return
     */
    public boolean remove(E e){
        int index = find(e);
        if (index == -1){
            return false;
        }
        remove(index);
        return true;
    }

    public void print(){
        Node tmp = head;
        StringBuffer buffer = new StringBuffer();
        buffer.append(" size is : " + size + "  ,   遍历链表 :  head --> ");
        while (tmp.next != null){
            tmp = tmp.next;
            buffer.append(tmp.toString() + " --> ");
        }
        System.out.println(buffer.toString());
    }


    public static void main(String[] args) {
        System.out.println("初始化链表");
        SingleLink<String> link = new SingleLink<>();
        link.print();

        System.out.println("addFirst a ");
        link.addFirst("a");
        link.print();

        System.out.println("addFirst b ");
        link.addFirst("b");
        link.print();

        System.out.println("addLast c ");
        link.addLast("c");
        link.print();

        System.out.println("查找节点 a 在第链表的下标为 : " + link.find("a"));
        System.out.println("查找节点 x 在第链表的下标为 :  " + link.find("x"));
```

```
161
162
163         System.out.println("获取下标为的 1 个节点 : " + link.get(1));
164
165         System.out.println("移除下标为 2 个节点");
166         link.remove(2);
167         link.print();
168
169         System.out.println("移除 c ");
170         link.remove("c");
171         link.print();
172         System.out.println("移除 b ");
173         link.remove("b");
174         link.print();
175     }
176
177 }
178
179 /*输出
180
181 初始化链表
182  size is : 1  ,   遍历链表 :  head -->
183 addFirst a
184  size is : 2  ,   遍历链表 :  head --> a -->
185 addFirst b
186  size is : 3  ,   遍历链表 :  head --> b --> a -->
187 addLast c
188  size is : 4  ,   遍历链表 :  head --> b --> a --> c -->
189 查找节点 a 在第链表的下标为 : 2
190 查找节点 x 在第链表的下标为 :  -1
191 获取下标为的 1 个节点 : b
192 移除下标为 2 个节点
193  size is : 3  ,   遍历链表 :  head --> b --> c -->
194 移除 c
195  size is : 2  ,   遍历链表 :  head --> b -->
196 移除 b
197  size is : 1  ,   遍历链表 :  head -->
198
199 */
```

优缺点说明

1. 插入到头部速度快，移动指针即可
2. 插入到尾部速度慢，需要从头部开始遍历查找整张链表
3. 删除头部比较快，移动指针即可
4. 删除其他节点就要进行遍历
5. 查询需要进行遍历

双向链表



单链表与双向链表的区别在于，链表中的节点多了一个指针，不仅仅有指向下一个节点的指针，还有一个指向上一个节点的指针，头结双向链表的好处就是循环方向多了一条，可以从前往后，也可以从后往前。

代码实现

```
package day1.双向链表;

public class DBLink<E> {
    private int size;       //  节点数量 包含头结点 和 尾节点
    private Node head;      //  头结点
    private Node tail;      //  尾节点

    static class Node<E>{
        private E data;         // 节点数据
         private Node next;      // 下一个节点
         private Node prev;      // 上一个节点

         public Node(E e){
             this.data = e;
         }

         @Override
         public String toString() {
             return data.toString();
         }
    }

    /**
     * 构造函数，初始化 头结点、尾节点
     */
    public DBLink(){
        head = new Node(null);
        tail = new Node(null);
        head.next = tail;
        tail.prev = head;
        size = 2;
    }

    /**
     * 根据节点 数据  获取在链表的下标值
     * @param e
     * @return
     */
    public int find(E e){
```

```
        if (e == null){
            throw new NullPointerException("数据不能为空");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (e.equals(tmp.data)){
                return cnt;
            }
        }
        return -1;
    }

    /**
     * 根据下标获取链表中的元素
     * @param index
     * @return
     */
    public Node get(int index){
        if (index < 0 || index >= size){
            throw new IndexOutOfBoundsException("元素下标不合法");
        }

        // 根据 index 判断 从头开始找，还是从尾部 开始找更快
        if (index * 2 <= size){
            Node tmp = head;
            int cnt = 0;
            while (tmp.next != null){
                tmp = tmp.next;
                cnt++;
                if (cnt == index){
                    return tmp;
                }
            }
        }else {
            Node tmp = tail;
            int cnt = 0;
            while (tmp.prev != null){
                tmp = tmp.prev;
                cnt++;
                if (cnt == size - index - 1){
                    return tmp;
                }
            }
        }
        return null;
    }

    /**
     * addFirst
     */
    public void addFirst(E e){
```

```
94          Node node = new Node(e);
95          Node next = head.next;
96
97          node.next = next;
98          next.prev = node;
99          head.next = node;
100          node.prev = head;
101          size++;
102      }
103
104      /**
105       * addLast
106       * @param e
107       */
108      public void addLast(E e){
109          Node node = new Node(e);
110          Node prev = tail.prev;
111
112          node.prev = prev;
113          prev.next = node;
114          tail.prev = node;
115          node.next = tail;
116          size++;
117      }
118
119      /**
120       * 根据元素的值删除
121       * @param e
122       */
123      public boolean remove(E e){
124          int index = find(e);
125          if (index == -1){
126              return false;
127          }
128          remove(index);
129          return true;
130      }
131
132      /**
133       * 根据 元素 下标 删除元素
134       * @param index
135       */
136      public void remove(int index){
137          if (index < 1 || index >= size - 1){
138              throw new IndexOutOfBoundsException("元素下标不合法");
139          }
140          Node curr = get(index);
141          Node prev = curr.prev;
142          Node next = curr.next;
143          prev.next = next;
144          next.prev = prev;
145          size--;
146      }
147
```

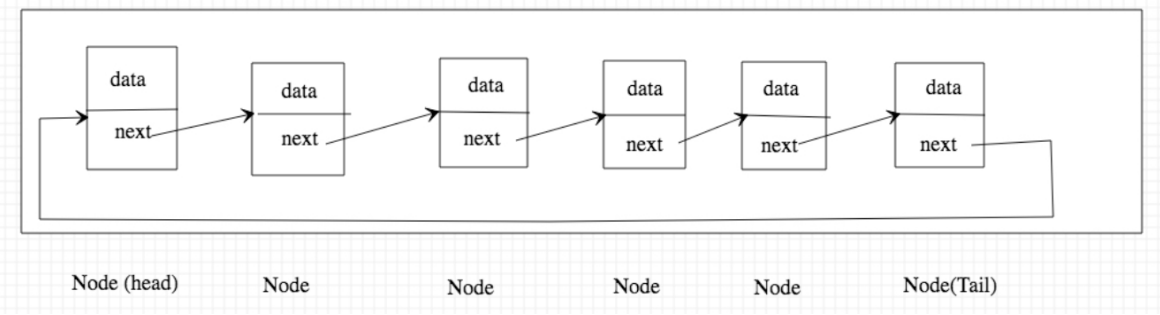
```
148     public void print(){
149         Node tmp = head;
150         StringBuffer buffer = new StringBuffer();
151         buffer.append(" size is : " + size + "  ,   正向遍历链表 :  head --> ");
152         while (tmp.next != null && !tmp.next.equals(tail)){
153             tmp = tmp.next;
154             buffer.append(tmp.toString() + " --> ");
155         }
156         buffer.append(" tail ");
157         System.out.print(buffer.toString());
158 
159         tmp = tail;
160         buffer = new StringBuffer();
161         buffer.append(" 反向遍历链表 : tail -->");
162         while (tmp.prev != null && !tmp.prev.equals(head)){
163             tmp = tmp.prev;
164             buffer.append(tmp.toString() + " <-- ");
165         }
166         buffer.append(" head ");
167         System.out.print(buffer.toString());
168         System.out.println();
169     }
170 
171     public static void main(String[] args) {
172 
173         DBLink<String> link = new DBLink<>();
174         System.out.println("构造完成");
175         link.print();
176         System.out.println("addFirst a ");
177         link.addFirst("a");
178         System.out.println("addFirst b ");
179         link.addFirst("b");
180         System.out.println("addFirst c ");
181         link.addFirst("c");
182         link.print();
183 
184         System.out.println("addLast 1 ");
185         link.addLast("1");
186         System.out.println("addLast 2 ");
187         link.addLast("2");
188         System.out.println("addLast 3 ");
189         link.addLast("3");
190         link.print();
191 
192         System.out.println("查找 a 在第 " + link.find("a") + " 个");
193         System.out.println("查找 x 在第 " + link.find("x") + " 个");
194 
195         System.out.println("获取下标为2的元素 " + link.get(2));
196 
197 
198         System.out.println("移除下标为2的元素");
199         link.remove(2);
200         link.print();
201 
```

```
        System.out.println("移除 元素 a");
        link.remove("a");
        link.print();
    }
}

/* 输出

构造完成
 size is : 2  ,   正向遍历链表 :  head --> tail  反向遍历链表 : tail --> head
addFirst a
addFirst b
addFirst c
 size is : 5  ,   正向遍历链表 :  head --> c --> b --> a -->  tail  反向遍历链表 : tail -->a <-- b <--
addLast 1
addLast 2
addLast 3
 size is : 8  ,   正向遍历链表 :  head --> c --> b --> a --> 1 --> 2 --> 3 -->  tail  反向遍历链表 : t
查找 a 在第 3 个
查找 x 在第 -1 个
获取下标为2的元素 b
移除下标为2的元素
 size is : 7  ,   正向遍历链表 :  head --> c --> a --> 1 --> 2 --> 3 -->  tail  反向遍历链表 : tail --
移除 元素 a
 size is : 6  ,   正向遍历链表 :  head --> c --> 1 --> 2 --> 3 -->  tail  反向遍历链表 : tail -->3 <--

*/
```

优缺点说明

1. 头部插入和尾部插入都比较快，只需要移动指针即可
2. 根据查询位置查询速度快于单链表，通过位置与链表长度比较，决定从头部还是尾部开始遍历
3. 根据元素的值查询需要遍历表
4. 根据位置删除速度快于单链表
5. 根据元素的值删除需要遍历表
6. 多一个指针，多占用一点空间

循环链表



循环链表就是在单链表的基础上，将尾节点的指针指向头结点。这样的好处是可以通过任意节点出发，访问到链表的全部节点

代码实现

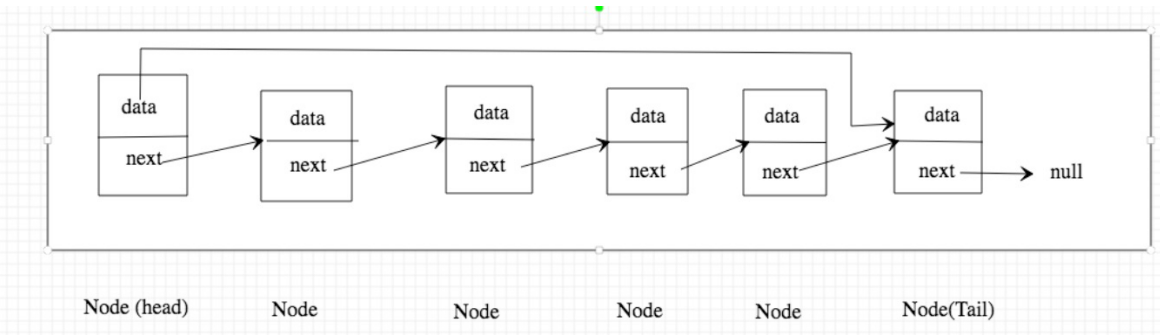
```java
package day1.循环链表;

import javax.sound.midi.Soundbank;

public class LoopLink<E> {

    private int size = 0;        //链表节点数量(包含头结点 和 尾节点)
    private Node head;           //链表的头结点
    private Node tail;           //链表的尾节点

     static class Node<E>{
        private E data;     // 节点数据
        private Node next;  // 下一个节点

        public Node(E e){
            this.data = e;
        }

        @Override
        public String toString() {
            return data.toString();
        }
    }

    public LoopLink(){
        head = new Node(null);
        tail = new Node(null);
        head.next = tail;
        tail.next = head;
        size = 2;
    }

    /**
     * 根据节点 数据  获取在链表的下标(从0开始)
     * @param e
     * @return
     */
    public int find(E e){
        if (e == null){
            throw new NullPointerException("数据不能为空");
        }
        Node tmp = head;
        int _index = -1;
        int cnt = 0;
        while (tmp.next != null && !tmp.next.equals(head)){
            tmp = tmp.next;
            cnt++;
            if (e.equals(tmp.data)){
                return cnt;
            }
        }
        return -1;
    }

```

```
55      /**
56       * 根据下表获取链表的元素
57       * @param index
58       * @return
59       */
60      public Node get(int index){
61          if (index < 0 || index >= size){
62              throw new IndexOutOfBoundsException("元素位置不合法");
63          }
64          Node tmp = head;
65          int cnt = 0;
66          while (tmp.next != null && !tmp.next.equals(tail)){
67              tmp = tmp.next;
68              cnt++;
69              if (cnt == index){
70                  return tmp;
71              }
72          }
73          return null;
74      }
75
76
77      /**
78       * addFirst
79       */
80      public void addFirst(E e){
81          Node node = new Node(e);
82          Node next = head.next;
83          node.next = next;
84          head.next = node;
85          size++;
86      }
87
88      /**
89       * 根据下标移除
90       * @param index
91       */
92      public void remove(int index){
93          if (index < 1 || index >= size - 1){
94              throw new IndexOutOfBoundsException("元素位置不合法");
95          }
96          Node curr = get(index);
97          if (index == 1){// 移除第一个节点
98              head.next = curr.next;
99          }else {
100              Node prev = get(index - 1);
101              prev.next = curr.next;
102          }
103          size--;
104      }
105
106      /**
107       * 根据节点的 值 移除该节点
108       * @param e
```

```
109      * @return
110      */
111     public boolean remove(E e){
112         int index = find(e);
113         if (index == -1){
114             return false;
115         }
116         remove(index);
117         return true;
118     }
119 
120     public void print(){
121         Node tmp = head;
122         StringBuffer buffer = new StringBuffer();
123         buffer.append(" size is : " + size + "  ,   遍历链表 :  head -->");
124         while (tmp.next != null && !tmp.next.equals(tail)){
125             tmp = tmp.next;
126             buffer.append(tmp.toString() + " --> ");
127         }
128         buffer.append(" tail ");
129         System.out.println(buffer.toString());
130     }
131 
132 
133     public static void main(String[] args) {
134 
135         LoopLink<String> loopLink = new LoopLink();
136         System.out.println("构造");
137         loopLink.print();
138 
139         System.out.println("addFirst a");
140         loopLink.addFirst("a");
141         loopLink.print();
142 
143         System.out.println("移除下标为1的元素");
144         loopLink.remove(1);
145         loopLink.print();
146 
147 
148         System.out.println("addFirst b1");
149         loopLink.addFirst("b1");
150         System.out.println("addFirst b2");
151         loopLink.addFirst("b2");
152         System.out.println("addFirst b3");
153         loopLink.addFirst("b3");
154         loopLink.print();
155 
156 
157         System.out.println("获取下标为2 :  " + loopLink.get(2));
158         System.out.println("查询 b2 的下标 :  " + loopLink.find("b2"));
159 
160         System.out.println("移除b2");
161         loopLink.remove("b2");
162         loopLink.print();
163 
```

```
        System.out.println("移除下标为1");
        loopLink.remove(2);
        loopLink.print();
    }
}

/* 输出

构造
 size is : 2  ,   遍历链表 :  head --> tail
addFirst a
 size is : 3  ,   遍历链表 :  head -->a -->  tail
移除下标为1的元素
 size is : 2  ,   遍历链表 :  head --> tail
addFirst b1
addFirst b2
addFirst b3
 size is : 5  ,   遍历链表 :  head -->b3 --> b2 --> b1 -->  tail
获取下标为2 :  b2
查询 b2 的下标 :  2
移除b2
 size is : 4  ,   遍历链表 :  head -->b3 --> b1 -->  tail
移除下标为1
 size is : 3  ,   遍历链表 :  head -->b3 -->  tail

*/
```

还有一种变示如下图，头节点的data本身为空，利用该段空间，存储一个指针，指向尾部，这样的好处是找到尾节点不用从头循环查找



代码实现

```
package day1.循环链表;

public class LoopLinkExt<E> {

    private int size = 0;       //链表节点数量(包含头结点)
    private Node head;          //链表的头结点

    static class Node<E>{
        private E data;     // 节点数据
         private Node next;  // 下一个节点

```

```
        public Node(E e){
            this.data = e;
        }

        @Override
        public String toString() {
            return data.toString();
        }
    }

    /**
     * 构造函数，初始化  头结点
     */
    public LoopLinkExt(){
        Node<E> node = new Node<>(null);
        node.next = null;
        head = node;
        size++;
    }

    /**
     * 根据节点 数据  获取在链表的下标值（从零开始）
     * @param e
     * @return
     */
    public int find(E e){
        if (e == null){
            throw new NullPointerException("数据不能为空");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (e.equals(tmp.data)){
                return cnt;
            }
        }
        return -1;
    }

    /**
     * 根据下标获取链表的元素
     * @param index
     * @return
     */
    public Node get(int index){
        if (index < 0 || index >= size){
            throw new IndexOutOfBoundsException("元素位置不合法");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
```

```
            cnt++;
            if (cnt == index){
                return tmp;
            }
        }
        return null;
    }

    /**
     * addFirst
     * @param e
     */
    public void addFirst(E e){
        if (size == 1){ // 长度等于 1 ，  只有头结点 ，  加入在头结点后面，从头节点和尾节点是一样的，效率也一样
            addLast(e);
        }else {
            Node node = new Node(e);
            node.next = head.next;
            head.next = node;
            size++;
        }
    }

    /**
     * addLast
     * @param e
     */
    public void addLast(E e){
        Node curr = new Node(e);
        if (size == 1){
            head.next = curr;
            head.data = curr;
        }else {
            Node tail = (Node) head.data;
             tail.next = curr;
             head.data = curr;
         }
         size++;
     }

     /**
      * 根据下标移除
      * @param index
      */
     public void remove(int index){
         if (index < 1 || index >= size){
             throw new IndexOutOfBoundsException("元素位置不合法");
         }
         Node curr = get(index);
         if (index == 1){// 移除第一个节点
             head.next = curr.next;
         }else {
             Node prev = get(index - 1);
             prev.next = curr.next;
```

```
        }
        size--;
    }

    /**
     * 根据节点的 值 移除该节点
     * @param e
     * @return
     */
    public boolean remove(E e){
        int index = find(e);
        if (index == -1){
            return false;
        }
        remove(index);
        return true;
    }

    public void print(){
        Node tmp = head;
        StringBuffer buffer = new StringBuffer();
        buffer.append(" size is : " + size + "  ,   遍历链表 :  head --> ");
        while (tmp.next != null){
            tmp = tmp.next;
            buffer.append(tmp.toString() + " --> ");
        }
        System.out.println(buffer.toString());
    }


    public static void main(String[] args) {
        System.out.println("初始化链表");
        LoopLinkExt<String> link = new LoopLinkExt<>();
        link.print();

        System.out.println("addFirst a ");
        link.addFirst("a");
        link.print();

        System.out.println("addFirst b ");
        link.addFirst("b");
        link.print();

        System.out.println("addLast c ");
        link.addLast("c");
        link.print();

        System.out.println("查找节点 a 在第链表的下标为 : " + link.find("a"));
        System.out.println("查找节点 x 在第链表的下标为 :  " + link.find("x"));


        System.out.println("获取下标为的 1 个节点 : " + link.get(1));

        System.out.println("移除下标为 2 个节点");
```

```
        link.remove(2);
        link.print();

        System.out.println("移除 c ");
        link.remove("c");
        link.print();
        System.out.println("移除 b ");
        link.remove("b");
        link.print();
    }
}
```

优缺点

实际上这种实现与单链表一致，不过插入尾节点做了优化，不用循环到尾部节点

有序链表

代码实现

```
package day1.有序链表;

import java.util.Comparator;

public class SortLink<E> {

    private int size = 0;       //链表节点数量(包含头结点)
    private Node head;          //链表的头结点
    private Comparator<E> comparator;

     static class Node<E>{
        private E data;     // 节点数据
        private Node next;  // 下一个节点

        public Node(E e){
            this.data = e;
        }

        @Override
        public String toString() {
            return data.toString();
        }
    }

    public void init(){
        Node<E> node = new Node<>(null);
        node.next = null;
        head = node;
        size++;
    }

    public SortLink(){
        init();
        comparator = new Comparator<E>() {
```

```
            @Override
            public int compare(E o1, E o2) {
                if (o1.hashCode() > o2.hashCode()){
                    return 1;
                }else if (o1.hashCode() < o2.hashCode()){
                    return -1;
                }
                return 0;
            }
        };
    }

    public SortLink(Comparator<E> comparator){
        init();
        this.comparator = comparator;
    }

    /**
     * 根据节点 数据  获取在链表的下标值（从零开始）
     * @param e
     * @return
     */
    public int find(E e){
        if (e == null){
            throw new NullPointerException("数据不能为空");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (e.equals(tmp.data)){
                return cnt;
            }
        }
        return -1;
    }

    /**
     * 根据下标获取链表的元素
     * @param index
     * @return
     */
    public Node get(int index){
        if (index < 0 || index >= size){
            throw new IndexOutOfBoundsException("元素位置不合法");
        }
        Node tmp = head;
        int cnt = 0;
        while (tmp.next != null){
            tmp = tmp.next;
            cnt++;
            if (cnt == index){
                return tmp;
```

```
            }
        }
        return null;
    }

    /**
     * add
     * @param e
     */
    public void add(E e){
        if(e == null){
            throw new NullPointerException("元素值不能为空");
        }
        Node node = new Node(e);
        if (size == 1){
            head.next = node;
        }else {
            Node tmp = head;
            int _index = -1;
            int cnt = 0;
            while (tmp.next != null){
                cnt++;
                tmp = tmp.next;
                int cmp = comparator.compare((E) tmp.data, (E) node.data);
                if (cmp > 0){
                    _index = cnt;
                    break;
                }
            }
            if (_index == -1){// 加到末尾
                Node last = get(size - 1);
                last.next = node;
            }else {
                Node prev = get(cnt - 1);
                Node next = prev.next;
                node.next = next;
                prev.next = node;
            }
        }
        size++;
    }

    /**
     * 根据下标移除
     * @param index
     */
    public void remove(int index){
        if (index < 1 || index >= size){
            throw new IndexOutOfBoundsException("元素位置不合法");
        }
        Node curr = get(index);
        if (index == 1){// 移除第一个节点
            head.next = curr.next;
        }else {
```

```
            Node prev = get(index - 1);
            prev.next = curr.next;
        }
        size--;
    }

    /**
     * 根据节点的 值 移除该节点
     * @param e
     * @return
     */
    public boolean remove(E e){
        int index = find(e);
        if (index == -1){
            return false;
        }
        remove(index);
        return true;
    }

    public void print(){
        Node tmp = head;
        StringBuffer buffer = new StringBuffer();
        buffer.append(" size is : " + size + "  ,   遍历链表 :  head --> ");
        while (tmp.next != null){
            tmp = tmp.next;
            buffer.append(tmp.toString() + " --> ");
        }
        System.out.println(buffer.toString());
    }


    public static void main(String[] args) {
        System.out.println("构造实例");
        SortLink<String> sortLink = new SortLink<>();
        sortLink.print();
        System.out.println("增加 a ");
        sortLink.add("a");
        sortLink.print();

        System.out.println("增加 c ");
        sortLink.add("c");
        sortLink.print();

        System.out.println("增加 e ");
        sortLink.add("e");
        sortLink.print();

        System.out.println("增加 d ");
        sortLink.add("d");
        sortLink.print();

        System.out.println("增加 f ");
        sortLink.add("f");
```

```
        sortLink.print();

        System.out.println("增加 b ");
        sortLink.add("b");
        sortLink.print();

        System.out.println("查找节点 a 在第链表的下标为 ： " + sortLink.find("a"));
        System.out.println("查找节点 x 在第链表的下标为 ：  " + sortLink.find("x"));


        System.out.println("获取下标为的 1 个节点 ： " + sortLink.get(1));

        System.out.println("移除下标为 2 个节点");
        sortLink.remove(2);
        sortLink.print();

        System.out.println("移除 c ");
        sortLink.remove("c");
        sortLink.print();
        System.out.println("移除 b ");
        sortLink.remove("b");
        sortLink.print();
    }
}

/*

构造实例
 size is : 1  ,   遍历链表 ：  head -->
增加 a
 size is : 2  ,   遍历链表 ：  head --> a -->
增加 c
 size is : 3  ,   遍历链表 ：  head --> a --> c -->
增加 e
 size is : 4  ,   遍历链表 ：  head --> a --> c --> e -->
增加 d
 size is : 5  ,   遍历链表 ：  head --> a --> c --> d --> e -->
增加 f
 size is : 6  ,   遍历链表 ：  head --> a --> c --> d --> e --> f -->
增加 b
 size is : 7  ,   遍历链表 ：  head --> a --> b --> c --> d --> e --> f -->
查找节点 a 在第链表的下标为 ：1
查找节点 x 在第链表的下标为 ：  -1
获取下标为的 1 个节点 ： a
移除下标为 2 个节点
 size is : 6  ,   遍历链表 ：  head --> a --> c --> d --> e --> f -->
移除 c
 size is : 5  ,   遍历链表 ：  head --> a --> d --> e --> f -->
移除 b
 size is : 5  ,   遍历链表 ：  head --> a --> d --> e --> f -->

*/
```