后来居上的"栈"

今日目标：

1：能说出栈的特点

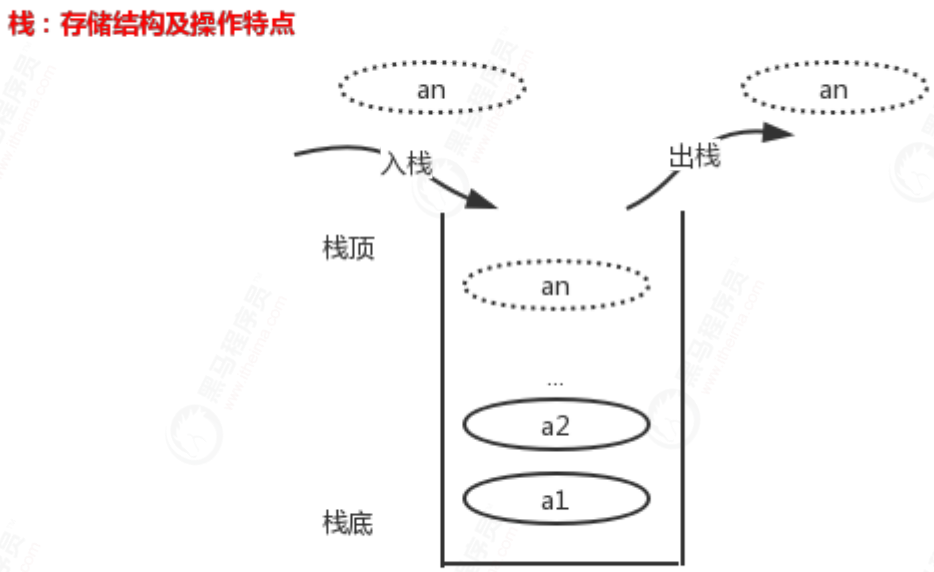
2：能基于数组实现栈

3：能基于单链表实现栈

4：完成栈的实战题目

1、存储结构与特点

“栈（**Stack**）”并非指某种特定的数据结构，它是有着相同典型特征的一类数据结构的统称，因为栈可以用数组实现，也可以用链表实现。该典型特征是：**后进先出**；英文表示为：Last In First Out即 **LIFO**，只要满足这种特点的数据结构我们就可以说这是栈，为了更好的理解栈这种数据结构，我们以一幅图的形式来表示，如下：



我们从栈的操作特点上来看，**栈就是一种操作受限的线性表，只允许在栈的一端进行数据的插入和删除，这两种操作分别叫做入栈（push）和出栈 （pop），时间复杂度均为O(1)**

> 知识小贴士：
>
> 1：此处讲的栈和java语言中讲到的栈空间不是一回事，此处的栈指的是一种数据，而java语言中的栈空间指的是java内存结构的一种表示，不能等同
>
> 2：相比于数组和链表来说，栈的数据操作受到了限制，那我们直接用数组或链表不就可以了么，为什么还要使用栈呢？

> 当某个数据集合如果只涉及到在其一端进行数据的插入和删除操作，并且满足先进后出，后进先出的特性时，我们应该首选栈这种数据结构来进行数据的存储。

2、栈的实现

栈既可以用数组来实现，也可以用链表来实现。用数组实现的栈叫**顺序栈**，用链表实现的叫**链式栈。**

对于栈的操作行为我们可以定义如下：

> 可搜索java中栈的方法定义

```
/**
     * 返回栈中元素个数
     * @return
     */
public  int size() {
    return 0;
}

/**
     * 判断栈是否为空
     * @return
     */
public boolean empty() {

    return false;
}

/**
     * 将元素压入栈
     * @param item 被存入栈的元素
     * @return
     */
public E push(E item) {

    return item;
}

/**
     * 获取栈顶元素，但并不移除，如果栈空则返回null
     * @return
     */
public  E peek() {

    return null;
}

/**
     * 移除栈顶元素并返回，如果栈为空则返回null
     * @return
     */
```

```
public  E pop() {

    return null;
}
```

2.1、数组实现栈

需求:

1：基于数组实现一个栈，满足以上定义的几个方法

2：基于数组的栈要支持动态扩容

实现:

（1）创建Stack，`com.itheima.stack.Stack`并添加如上定义的几个方法

（2）定义栈的大小和存储数据的数组

```
    //存储数据的数组
    Object[] elementData;
    //栈中元素的个数
    int elementCount;
```

（3）添加构造方法，初始化数组

```
/**
     * 指定初始化大小
     * @param initCapacity
     */
public Stack(int initCapacity){
    this.elementData = new Object[initCapacity];
}

/**
     * 默认构造
     */
public Stack(){
    this(10);
}
```

（4）实现`size,empty`方法

```
/**
     * 返回栈中元素个数
     * @return
     */
public  int size() {
```

```
    return elementCount;
}

/**
     * 判断栈是否为空
     * @return
     */
public boolean empty() {
    return elementCount == 0;
}
```

(5) 实现 `push` 方法

```
/**
     * 将元素压入栈
     * @param item 被存入栈的元素
     * @return
     */
public E push(E item) {
    ensureCapacity(elementCount+1);
    this.elementData[elementCount++] = item;
    return item;
}
private void ensureCapacity(int minCapacity) {
    if (minCapacity > this.elementData.length) {
        grow(minCapacity);
    }
}
private void grow(int minCapacity) {
    int oldCapacity = this.elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >>1);
    if (newCapacity < minCapacity) {
        newCapacity = minCapacity;
    }
    //借助数组工具类Arrays快速拷贝
    this.elementData = Arrays.copyOf(this.elementData, newCapacity);
}
```

(6) 实现 `peek,pop` 方法

```
/**
     * 获取栈顶元素，但并不移除，如果栈空则返回null
     * @return
     */
public  E peek() {
    int len = size();
    if (len == 0) {
        return null;
    }
    return elementAt(len-1);

}
```

```
12
13  private E elementAt(int index) {
14      if (index >=this.elementCount || index <0) {
15          throw new
    ArrayIndexOutOfBoundsException("index="+index+",elementCount="+this.elementCount);
16      }
17      return (E) this.elementData[index];
18  }
19
20  /**
21       * 移除栈顶元素并返回，如果栈为空则返回null
22       * @return
23       */
24  public  E pop() {
25      E peek = peek();
26      int len = size();
27      removeElementAt(len-1);
28      return peek;
29  }
30
31  private void removeElementAt(int index) {
32      if (index >=this.elementCount || index <0) {
33          throw new
    IndexOutOfBoundsException("index="+index+",elementCount="+this.elementCount);
34      }
35      if (index < this.elementCount - 1) {
36          System.arraycopy(this.elementData,index+1,this.elementData,index,this.elementCount-
    index-1);
37      }
38      //栈中元素个数减一
39      elementCount--;
40      elementData[elementCount] = null;
41  }
```

(7) 实现 `toString` 方法

```
1  @Override
2  public String toString() {
3      //将栈中元素按照[1,2,3,4]形式打印
4      StringBuilder stringBuilder = new StringBuilder("[");
5      for (int i=0;i<this.elementCount;i++){
6          stringBuilder.append(this.elementData[i]).append(",");
7      }
8      return stringBuilder.append("]").toString();
9  }
```

(8) 创建测试类：`com.itheima.stack.SatckTest`

```
1  public static void main(String[] args) {
2      //创建栈
```

```
    Stack stack = new Stack();
    //元素入栈
    stack.push(1);
    stack.push(3);
    stack.push(5);
    stack.push(7);
    System.out.println("栈中元素个数:"+stack.size()+",栈是否为空:"+stack.empty());
    System.out.println("打印输出栈:"+stack);
    System.out.println("栈顶元素为: "+stack.peek());
    System.out.println("元素出栈"+stack.pop());
    System.out.println("打印输出栈"+stack);
}
```

2.2、链表实现栈

需求:

1：基于单链表实现一个栈，满足以上定义的几个方法

实现:

（1）创建`com.itheima.stack.LinkedListStack`并添加栈的相关方法

（2）因为要基于单链表实现，因此首先定义链表节点对象Node

```
/**
     * 定义单链表节点对象
     * @param <E>
     */
private static class Node<E>{
    E val;
    Node<E> next;

    public Node(E val,Node<E> next){
        this.val = val;
        this.next = next;
    }
}
```

（3）定义栈大小，链表头节点指针

```
//栈中元素个数
int size;
//栈顶指针,链表头结点指针
Node<E> head;
```

> 注意:
>
> 1：这个地方为什么我们维护链表头节点指针，而不是尾节点?

> 因为这是基于单链表，如果每次元素入栈我们将其添加到链表尾的话（相当于将链表尾当作栈顶），后面在进行元素出栈要删除栈顶元素时我们没办法找到它的前一个元素
>
> 因此我们可以将链表头节点当作栈顶

(4) 定义构造，

```
public LinkedListStack(){}
```

(5) 实现`size,empty`方法

```
/**
     * 返回栈中元素个数
     * @return
     */
public  int size() {
    return size;
}

/**
     * 判断栈是否为空
     * @return
     */
public boolean empty() {
    return size == 0;
}
```

(6) 实现`push`方法

```
/**
     * 将元素压入栈
     * @param item 被存入栈的元素
     * @return
     */
public E push(E item) {
    Node<E> newNode = new Node<>(item,head);
    head = newNode;
    size++;
    return item;
}
```

(7) 实现`peek,pop`方法

```
/**
     * 获取栈顶元素，但并不移除，如果栈空则返回null
     * @return
     */

public  E peek() {
```

```
    if (head == null){
        return null;
    }
    return head.val;
}

/**
     * 移除栈顶元素并返回，如果栈为空则返回null
     * @return
     */
public  E pop() {
    if (head == null){
        return null;
    }
    Node<E> top = head;

    head = head.next;
    top.next = null;

    return top.val;
}
```

(8) 实现 `toString` 方法

```
@Override
public String toString() {
    //打印1->2->3->null格式的数据
    StringBuilder sb = new StringBuilder();
    Node curr = head;
    while (curr!=null){
        sb.append(curr.val).append("->");
        curr = curr.next;
    }
    return sb.append("null").toString();
}
```

(9) 编写测试类：`com.itheima.stack.LinkedListStackTest`

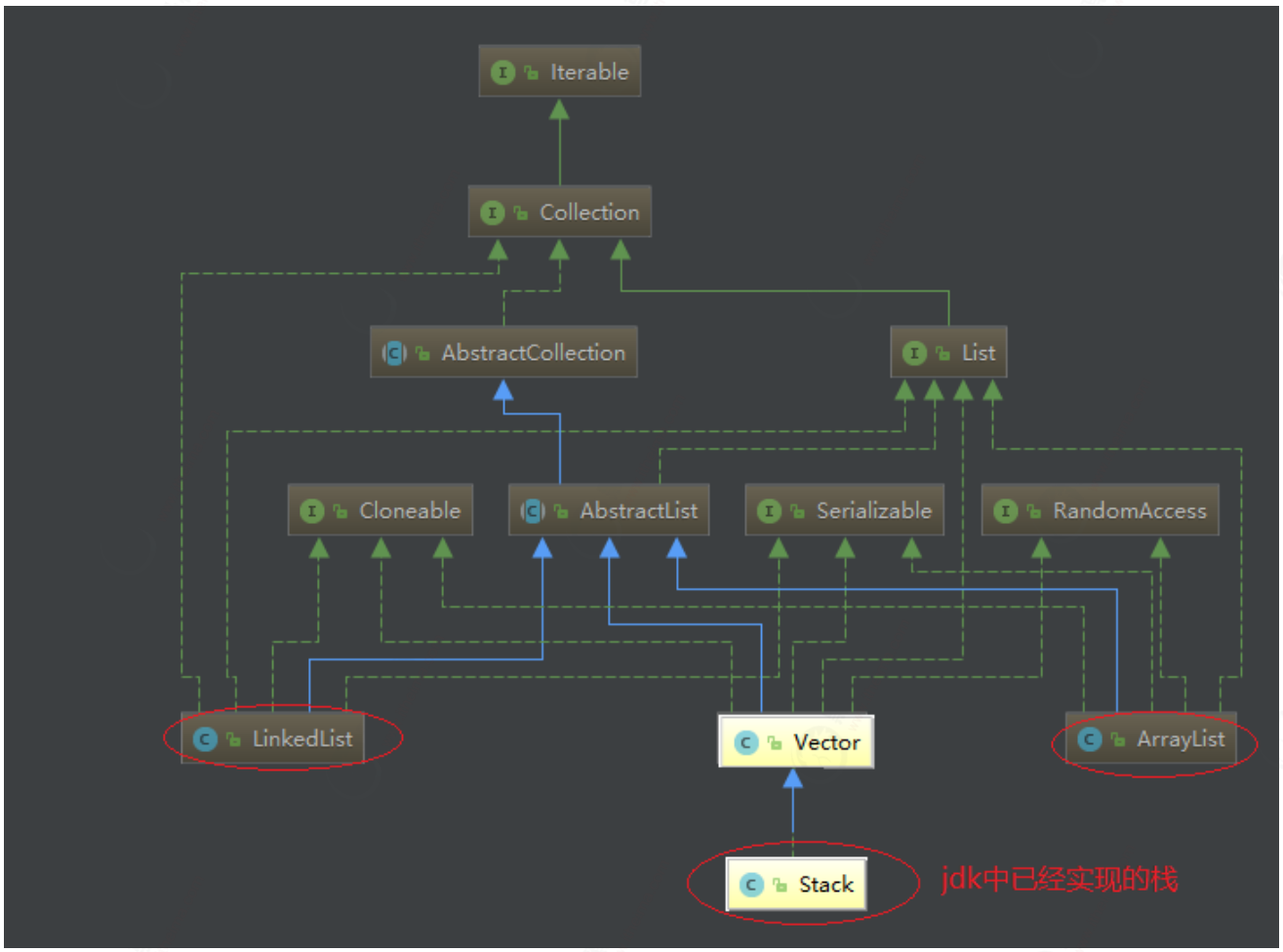
```java
public static void main(String[] args) {
    LinkedListStack stack = new LinkedListStack();
    //元素入栈
    stack.push(1);
    stack.push(3);
    stack.push(5);
    stack.push(7);
    System.out.println("栈中元素个数:"+stack.size()+",栈是否为空:"+stack.empty());
    System.out.println("打印输出栈:"+stack);
    System.out.println("栈顶元素为: "+stack.peek());
    System.out.println("元素出栈"+stack.pop());
    System.out.println("打印输出栈"+stack);
}
```

（10）拓展：能否基于双向链表来实现栈？

> 可以维护一个链表尾节点指针，指向栈顶，元素出栈时可以通过前驱指针pre得到前一个节点

2.3、总结

在java中对于栈这种数据结构已经有对应的实现了，List接口下不仅有我们之前讲到过的ArrayList和LinkedList集合类，还有一个Stack类，下面的图可以帮我们很清晰的看到List接口下的实现情况



课后作业：分析java中`Stack,Vector,ArrayList,LinkedList`对应的实现，并对他们进行比较

> **Vector,Stack,ArrayList,LinkedList的比较**
>
> 首先都实现List接口，而List接口一共有三个实现类，分别是ArrayList、Vector和LinkedList。List用于存放多个元素，能够维护元素的次序，并且允许元素的重复。3个具体实现类的相关区别如下：
>
> 1：ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要将已有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。
>
> 2：Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。
>
> 3： LinkedList是用链表结构(双向链表)存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。
>
> 4：Vector和Stack是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而Arraylist是线程异步（ASynchronized）的，是不安全的。如果不考虑到线程的安全因素，一般用Arraylist效率比较高。
>
> 5：如果集合中的元素的数目大于目前集合数组的长度时，vector增长是按照 2*原数组大小，,而arraylist增长率为1.5 *原数组大小。如果在集合中使用数据量比较大的数据，用vector有一定的优势。
>
> 6： 如果查找一个指定位置的数据，vector和arraylist使用的时间是相同的，都是0(1),这个时候使用vector和arraylist都可以。而如果移动一个指定位置的数据花费的时间为0(n)，这个时候就应该考虑到使用Linkedlist,因为它移动一个指定位置的数据所花费的时间为0(1),而查询一个指定位置的数据时花费的时间为0(n)。ArrayList 和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要设计到数组元素移动 等内存操作，所以索引数据快插入数据慢，Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快.。
>
> 7：Stack是继承自Vector，底层也是基于数组实现的，只不过Stack插入和获取元素有一定的特点，满足后进先出的特点即LIFO，因此Stack也是我们所讲的典型的"栈"这种数据结构，且底层也支持动态扩容，其扩容方式和Vector，ArrayList底层扩容原理一样。Stack元素入栈和出栈的时间复杂度都是O(1)。

3、栈的面试题

3.1、20.有效的括号

[哔哩哔哩，小米最近面试题，20. 有效的括号]

栈的应用

```
1  class Solution {
2      public boolean isValid(String s) {
3          //特殊判断
```

```
        if (s.isEmpty()) {
            return true;
        }

        char[] t = s.toCharArray();
        if (t.length % 2 !=0) { //奇数位肯定不满足
            return false;
        }
        Stack<Character> stack = new Stack();
        for (char c:t) {
            //遇到左括号就把对应的右括号入栈，否则就弹出栈顶元素与当前遍历的元素进行比较
            if (c == '(') {
                stack.push(')');
            }else if (c == '[') {
                stack.push(']');
            }else if (c == '{') {
                stack.push('}');
            }else if (stack.isEmpty() || c != stack.pop()) {
                return false;
            }
        }

        //如果一轮循环结束后栈里面是空的则返回true
        return stack.isEmpty();
    }
}
```

第二种解法

```
class Solution {
    public boolean isValid(String s) {
        if (s.isEmpty()) {
            return true;
        }
        char[] c = s.toCharArray();
        if (c.length % 2 !=0) {
            return false;
        }
        //事先缓存成对的括号组合
        Map<Character,Character> cache = new HashMap();
        cache.put('}', '{');
        cache.put(')', '(');
        cache.put(']', '[');
        Stack<Character> stack = new Stack();
        for (char cr : c) {
            if ( cr == '(' || cr == '[' || cr == '{') {
                stack.push(cr);
            }else if (stack.isEmpty() || cache.get(cr) != stack.pop()) {//遇到右括号则弹出栈顶元素，并判断栈顶元素是否是和该右括号成对的左括号，因此需要事先准备好一个缓存字典，缓存所有与右括号成对的左括号
                return false;
```

```
            }
        }

        return stack.isEmpty();
    }
}
```

3.2、155. 最小栈

亚马逊，字节跳动，腾讯最近面试题，155. 最小栈

1，借助辅助栈

```
class MinStack {
    private Stack<Integer> stack ;
    private Stack<Integer> minStack;

    /** initialize your data structure here. */
    public MinStack() {
        stack= new Stack();
        minStack = new Stack();
    }

    public void push(int x) {
        stack.push(x);
        if (minStack.isEmpty() || x <= minStack.peek() ) {
            minStack.push(x);
        }
    }

    public void pop() {
        int top = stack.pop();
        if (top == minStack.peek()) {
            minStack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

2.一个栈，同时保存元素及最小值

```java
class MinStack {
    private int min = Integer.MAX_VALUE;
    private Stack<Integer> stack;
    /** initialize your data structure here. */
    public MinStack() {
        stack = new Stack();
    }

    public void push(int x) {
        if (x <= min) {
            stack.push(min);
            min = x;
        }
        stack.push(x);
    }

    public void pop() {
        int top = stack.pop();
        if (top == min) {
            min = stack.pop();
        }
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return min;
    }
}
```

基于第二种解法思想，栈中保存元素和最小值的差值

```java
class MinStack {

    private long min; //
    private Stack<Long> stack;

    /** initialize your data structure here. */
    public MinStack() {
        stack = new Stack();
    }

    public void push(int x) {
        if (stack.isEmpty()) {
            min = x;
            stack.push(x-min);
        }else {
            stack.push(x - min);
```

```
            if (x < min) {
                min = x;
            }
        }
    }

    public void pop() {
        if (stack.isEmpty()) {
            return;
        }
        long top = stack.pop();
        if (top < 0) {
            min = min - top;
        }
    }

    public int top() {
        long top = stack.peek();
        if (top < 0) {
            return (int)min;
        }
        return (int)(top + min);
    }

    public int getMin() {
        return (int)min;
    }
}
```

> 上边的解法的一个缺点就是由于我们保存的是差值，所以可能造成溢出，所以我们用了数据范围更大的 `long` 类型。