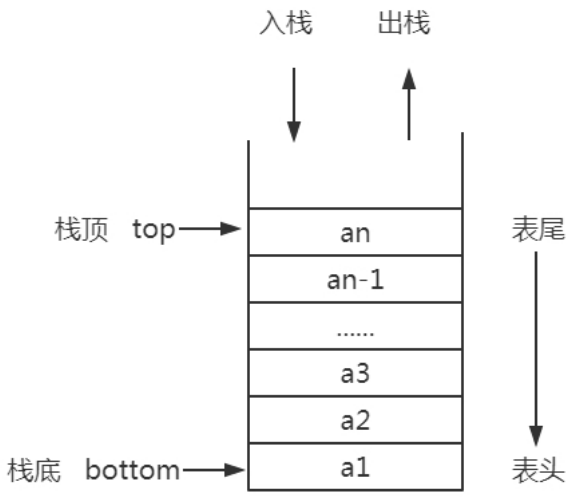
定义



栈（Stack）是限定仅在表尾进行插入和删除操作的线性表

特点&应用

1. 栈具有记忆功能，具有顺序，入栈前后是有顺序的
2. 正序入栈，反序出栈

抽象数据模型(ADT)

```
/**
 * 栈 - 抽象数据类型（ADT）
 */
```

```java
public interface StackADT<E> {
    boolean isEmpty();  // 是否为空
    E pop();            // 出栈
    void push(E e);     // 压栈
    int size();         // 获取栈的元素个数
    E peek();           // 查看栈顶，不出栈
}
```

栈可以使用数组实现，也可以通过链表来实现

实现方式 - 数组

代码实现

```java
package day2.栈;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.SortedSet;

public class StackByArray<E> implements StackADT<E> {

    private int maxSize;            // 栈的长度
    private E[] array;              // 数组
    private int top;                // 栈顶指针

    public StackByArray(int maxSize){
        this.maxSize = maxSize;
        array = (E[]) new Object[maxSize];
        this.top = -1;
    }

    @Override
    public boolean isEmpty() {
        return top == -1 ? true : false;
    }

    @Override
    public E pop() {
        E e = peek();
        if (size() > 0){
            top--;
        }
        return e;
    }

    @Override
    public void push(E e) {
        array[++top] = e;
    }

    @Override
```

```
    public int size() {
        return top + 1;
    }

    @Override
    public E peek() {
        int size = size();
        if (size == 0){
            return null;
        }
        return array[size - 1];
    }

    @Override
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("size : " + size() + " , 元素遍历 : ");
        for (int i = 0 ; i < size(); i++){
            buffer.append(array[i].toString());
            if (i < size() - 1){
                buffer.append(" --> ");
            }
        }
        return buffer.toString();
    }

    public void print(){
        System.out.println(toString());
    }

    public static void main(String[] args) {
        StackByArray<String> stack = new StackByArray<>(5);
        stack.print();
        System.out.println("入栈 1");
        stack.push("1");
        System.out.println("入栈 2");
        stack.push("2");
        stack.print();

        String e = stack.peek();
        System.out.println("查看栈顶 :  " + e);
        stack.print();

        System.out.println("入栈 3");
        stack.push("3");
        stack.print();
        System.out.println("出栈: " + stack.pop());
        stack.print();
        System.out.println("出栈: " + stack.pop());
        stack.print();
        System.out.println("出栈: " + stack.pop());
        stack.print();
        System.out.println("出栈: " + stack.pop());
        stack.print();
```

```
        System.out.println("入栈 a");
        stack.push("a");
        stack.print();
    }
}


/* 输出

size : 0 , 元素遍历 :
入栈 1
入栈 2
size : 2 , 元素遍历 : 1 --> 2
查看栈顶 :  2
size : 2 , 元素遍历 : 1 --> 2
入栈 3
size : 3 , 元素遍历 : 1 --> 2 --> 3
出栈: 3
size : 2 , 元素遍历 : 1 --> 2
出栈: 2
size : 1 , 元素遍历 : 1
出栈: 1
size : 0 , 元素遍历 :
出栈: null
size : 0 , 元素遍历 :
入栈 a
size : 1 , 元素遍历 : a

*/
```

说明

1. 初始化就决定了栈的容量，存在动态扩容问题
2. 初始化传入栈的容量
3. 存储结构很紧凑，无序的，读取和写入快

实现方式 - 链表

代码实现

```
package day2.栈;

import day1.单链表.SingleLink;

public class StackByLink<E> implements StackADT<E> {

    private SingleLink<E> link = null;
    private int top;                        // 栈顶指针

     public StackByLink() {
        link = new SingleLink<>();
        top = -1;
    }
```

```
14
15      @Override
16      public boolean isEmpty() {
17          return top == -1 ? true : false;
18      }
19
20      @Override
21      public E pop() {
22          E e = peek();
23          if (size() > 0){
24              link.remove(1);
25              top--;
26          }
27          return e;
28      }
29
30      @Override
31      public void push(E e) {
32          link.addFirst(e);
33          top++;
34      }
35
36      @Override
37      public int size() {
38          return top + 1;
39      }
40
41      @Override
42      public String toString() {
43          StringBuffer buffer = new StringBuffer();
44          buffer.append("size : " + size() + " , 元素遍历 : ");
45
46          for (int i = 1 ; i < size() + 1; i++){
47              buffer.append(link.getData(i).toString());
48              if (i < size()){
49                  buffer.append(" --> ");
50              }
51          }
52          return buffer.toString();
53      }
54
55      public void  print(){
56          System.out.println(toString());
57      }
58
59      @Override
60      public E peek() {
61          int size = size();
62          if(size == 0){
63              return null;
64          }
65          return link.getData(1);//  写入方式 从 往头部添加，栈顶永远在最前面
66      }
67
```

```
68      public static void main(String[] args) {
69          StackByLink<String> stack = new StackByLink<>();
70          stack.print();
71          System.out.println("入栈 1");
72          stack.push("1");
73          System.out.println("入栈 2");
74          stack.push("2");
75          stack.print();
76
77          String e = stack.peek();
78          System.out.println("查看栈顶 ：  " + e);
79          stack.print();
80
81          System.out.println("入栈 3");
82          stack.push("3");
83          stack.print();
84
85          String e2 = stack.peek();
86          System.out.println("查看栈顶 ：  " + e2);
87
88          System.out.println("出栈: " + stack.pop());
89          stack.print();
90          System.out.println("出栈: " + stack.pop());
91          stack.print();
92          System.out.println("出栈: " + stack.pop());
93          stack.print();
94          System.out.println("出栈: " + stack.pop());
95          stack.print();
96          System.out.println("入栈 a");
97          stack.push("a");
98          stack.print();
99
100     }
101 }
102
103
104 /* 输出
105  size : 0 , 元素遍历 :
106 入栈 1
107 入栈 2
108 size : 2 , 元素遍历 : 2 --> 1
109 查看栈顶 ：  2
110 size : 2 , 元素遍历 : 2 --> 1
111 入栈 3
112 size : 3 , 元素遍历 : 3 --> 2 --> 1
113 查看栈顶 ：  3
114 出栈: 3
115 size : 2 , 元素遍历 : 2 --> 1
116 出栈: 2
117 size : 1 , 元素遍历 : 1
118 出栈: 1
119 size : 0 , 元素遍历 :
120 出栈: null
121 size : 0 , 元素遍历 :
```

```
122 入栈 a
123 size : 1 , 元素遍历 : a
124
125
126 */
```

说明

1. 链表是从头部插入的，永远在第一个，对链表的第一个进行添加和获取都非常快，所以这种栈的入栈和出栈都很快
2. 长度不固定没有上面数组的限制