栈:

从计算机科学的角度来看

它是一种数据结构，是一种只能在一端进行插入和删除操作的特殊线性表；它按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。栈具有记忆作用，对栈的插入与删除操作中，不需要改变栈底指针；

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶，另一端为栈底；栈底固定，栈顶浮动；栈中元素个数为零时成为空栈。插入一般称为进栈，删除则称为退栈，栈也称为先进后出表；

栈顶：在表尾进行插入和删除操作的线性表。相对的，把另一端称为栈底；
进站(压栈，入栈)：向一个栈插入新元素，它是把新元素放到栈顶元素的上面，使之成为新的栈顶元素；
出栈(退栈)：从一个栈删除元素，它是把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素；

栈应用:

调用栈是解释器（比如浏览器中的 JavaScript 解释器）追踪函数执行流的一种机制。当执行环境中调用了多个函数时，通过这种机制，我们能够追踪到哪个函数正在执行，执行的函数体中又调用了哪个函数；

- 每调用一个函数，解释器就会把该函数添加进调用栈并开始执行。
- 正在调用栈中执行的函数还调用了其它函数，那么新函数也将会被添加进调用栈，一旦这个函数被调用，便会立即执行。
- 当前函数执行完毕后，解释器将其清出调用栈，继续执行当前执行环境下的剩余的代码。
- 当分配的调用栈空间被占满时，会引发“堆栈溢出”错误。

例:

```
function greeting() {
    // [1] Some codes here
    sayHi();   // 入栈  [greeting,sayHi]
    // [2] Some codes here
}
function sayHi() {
    return "Hi!";
}
// 调用 `greeting` 函数
greeting();  // 入栈  [greeting]
// [3] Some codes here
过程解析:
1. 忽略前面所有函数，直到 greeting() 函数被调用。
2. 把 greeting() 添加进调用栈列表。
3. 执行 greeting() 函数体中的所有代码。
4. 代码执行到 sayHi() 时，该函数被调用。
5. 把 sayHi() 添加进调用栈列表。
6. 执行 sayHi() 函数体中的代码，直到全部执行完毕。
7. 返回来继续执行 greeting() 函数体中 sayHi() 后面的代码。
8. 删除调用栈列表中的 sayHi() 函数。
9. 当 greeting() 函数体中的代码全部执行完毕，返回到调用 greeting() 的代码行，继续执行剩下的 JS 代
10.删除调用栈列表中的 greeting() 函数。
```

Vue源码中parseHtml 方法 目录: vue/src/compiler/parser/html-parser.js
实现一个简单的栈:

1.基础类

```
class Stack{
  // 存储栈数据 一般两种 链表或者数组 这里用数组.
  private stack_list = [];
}

```

2.push(item) 压栈

```
public push(item){
  this.stack_list.push(item);
  return item;
}
```

3.pop出栈

```
public pop() {
 // 弹出栈顶 先进后出
 return this.stack_list.pop()
}
```

4.top查看栈顶

```
public top() {
  // 栈顶元素就是数组的最后一项
  return this.stack_list[this.size() - 1];
}
```

5.size 查看栈的长度

```
public size() {
  return this.stack_list.length;
}

```

6.is_empty栈是否为空

```
public is_empty() {
  return this.size === 0
}
```

7.clear清空栈

```
public clear() {
  this.stack_list = [];
  return true;
}
```

栈的常见算法题

1.合法括号
下面的字符串中包含小括号，请编写一个函数判断字符串中的括号是否合法，所谓合法，就是括号成对出现

```
import Stack from 'Stack';
// a例分析流程
function is_matching(str) { // a(b)((c))
  const stack = new Stack();
  let item;
```

```
  for (let i = 0; i < str.length; i++) {
    item = str[i]; // 1=>a 2=>( 3=>b 4=>) 5=>( 6=>( 7=>) 8=>)
    if (item === '(') { // 如果是 （ 入栈
       stack.push(item);  // 2 =>此时栈 ['('] ,5=>此时栈[(] 6=>此时栈[(,(]
     } else if (item === ')') { // 如果是 ） 当前栈如果是空则不匹配
       if (stack.isEmpty()) {return false} //4=> 此时栈 ['(']   7=>此时栈[(,(]  8=>此时栈[(
       stack.pop(); // 出栈  // 4=>此时栈[]  7=>此时栈[(] 8=>此时栈[]
    }
   }
  return stack.isEmpty(); // 循环结束 当前栈如果是空则匹配
}
is_matching('a(b)(())') // a例
is_matching('sdf(ds(ew(we)rw) rwqq) qwewe') // true
```

2.逆波兰表达式(后缀表达式)

例:（a+b）* (c+d) 转换为ab+cd+*

［4,13,5,'/','+'］ 4+13/5 = 6;

请编写函数calc_exp(exp) 实现逆波兰表达式 exp的类型是数组

```
import Stack from 'Stack';
function calc_exp(exp) {
  const arr = exp;
  const stack = new Stack();
  let item;
  for (let i = 0; i < arr.length; i++) {
    item = arr[i];
    if (symbols.includes(item)) {
      const value1 = stack.pop();
      const value2 = stack.pop();
      stack.push(eval(value2 + item + value1));
    } else {
      stack.push(item);
    }
  }
  return stack.pop();
}
console.log(calc_exp(['4', '13', '5', '/', '+']))
```

3.实现一个有min方法的栈

```
class MinStack {
  private min_stack = [];
  private stack = [];
  public push(item) {
    if (this.is_empty()) {
      this.min_stack.push(item);
    } else {
      const minNode = this.min();
      if (minNode > item) {
        this.min_stack.push(item);
      } else {
```

```
        this.min_stack.push(minNode);
      }
    }
    this.stack.push(item);
  }
  public min() {
    return this.min_stack[this.min_stack.length - 1];
  }
  public pop() {
    this.min_stack.pop();
    return this.stack.pop();
  }
  public size() {
    return this.stack.length;
  }
  public is_empty() {
    return this.size() === 0;
  }
}
```

4.使用栈，完成中序表达式转后续表达式

```
输入[12,'+',3] 输出[12,3,'+'],
输入["(", "1", "+", "(", "4", "+", "5", "+", "3", ")", "-", "3", ")", "+", "(", "9", "+"
输出: ['1', '4', '5', '+', '3', '+', '+', '3', '-', '9', '8', '+', '+']
输入['(', '1', '+', '(', '4', '+', '5', '+', '3', ')', '/', '4', '-', '3', ')', '+', '('
输出['1', '4', '5', '+', '3', '+', '4', '/', '+', '3', '-', '6', '8', '+', '3', '*', '+'
const symbols = ['+', '-', '*', '/'];
const symbol_priority = {
  '+': 1,
  '-': 1,
   '*': 2,
   '/': 2
}
function centre_later(exp) {
  const arr = exp;
  const stack = new Stack();
  const result = [];
  let item;
  for (let i = 0; i < exp.length; i++) {
    item = exp[i];
    if (item === '(') {
      stack.push(item);
    } else if (item === ')') {
      let cur_stack = stack.pop();
      while (cur_stack !== '(') {
        result.push(cur_stack);
        cur_stack = stack.pop();
```

```
      }
    } else if (symbols.includes(item)) {
      let cur_stack = stack.top();
      while (cur_stack && symbol_priority[cur_stack] >= symbol_priority[item]) {
        result.push(cur_stack);
        stack.pop();
        cur_stack = stack.top();
      }
      stack.push(item);
    } else {
      result.push(item);
    }
  }
  while (!stack.isEmpty()) {
     result.push(stack.pop())
  }
  return result;
}
```