

栈 (Stack) 数据结构

李睿远

Sep 11, 2025

栈是计算机科学中最基础且无处不在的数据结构之一。想象一下日常生活中叠盘子的场景：我们总是将新盘子放在最上面，取用时也从最上面开始拿。这种后进先出的行为正是栈的核心理念。在计算机领域，栈的应用广泛而关键，例如函数调用栈管理着程序的执行流程，浏览器中的「后退」按钮依赖于栈来记录历史页面，甚至表达式求值和撤销操作都离不开栈的支持。本文将带领您从理论层面深入理解栈的概念，并通过代码实现两种常见的栈结构，最后探讨其经典应用场景。

1 栈的核心概念剖析

栈是一种线性数据结构，其操作遵循后进先出 (LIFO, Last-In, First-Out) 原则。这意味着最后一个被添加的元素将是第一个被移除的。栈的核心操作包括入栈 (Push) 和出栈 (Pop)。入栈操作将一个新元素添加到栈顶，而出栈操作则移除并返回栈顶元素。此外，栈还支持一些辅助操作，例如查看栈顶元素 (peek 或 top)、检查栈是否为空 (isEmpty)、检查栈是否已满 (isFull，仅适用于基于数组的实现) 以及获取栈的大小 (size)。从抽象数据类型 (ADT) 的角度来看，栈可以定义为支持这些操作的一个集合，其接口确保了数据访问的严格顺序性。

2 栈的实现方式 (一)：基于数组

基于数组的实现是栈的一种常见方式，其思路是使用一个固定大小的数组来存储元素，并通过一个称为 `top` 的指针来跟踪栈顶的位置。初始化时，`top` 通常设置为 `-1`，表示栈为空。当执行入栈操作时，`top` 递增，并将新元素存储在数组的相应索引处；出栈操作则返回 `top` 指向的元素，并将 `top` 递减。这种实现的关键在于处理边界情况，例如当栈为空时尝试出栈会引发错误，而当栈满时尝试入栈会导致溢出。

以下是一个用 Python 实现的基于数组的栈示例代码：

```
1 class ArrayStack:
2     def __init__(self, capacity):
3         self.capacity = capacity
4         self.stack = [None] * capacity
5         self.top = -1
6
7     def push(self, item):
8         if self.is_full():
9             raise Exception("Stack is full")
10        self.top += 1
```

```
11     self.stack[self.top] = item

13     def pop(self):
14         if self.is_empty():
15             raise Exception("Stack is empty")
16         item = self.stack[self.top]
17         self.top -= 1
18         return item

19
20     def peek(self):
21         if self.is_empty():
22             raise Exception("Stack is empty")
23         return self.stack[self.top]

24
25     def is_empty(self):
26         return self.top == -1

27
28     def is_full(self):
29         return self.top == self.capacity - 1

30
31     def size(self):
32         return self.top + 1
```

在这段代码中，我们定义了一个 `ArrayStack` 类，其初始化方法接受一个容量参数，并创建一个相应大小的数组。`push` 方法首先检查栈是否已满，如果未满，则递增 `top` 并将元素存入数组；`pop` 方法检查栈是否为空，然后返回栈顶元素并递减 `top`。`peek` 方法类似，但不修改栈。所有核心操作的时间复杂度均为 $O(1)$ ，因为它们只涉及简单的索引操作。空间复杂度为 $O(n)$ ，其中 n 是数组的容量。基于数组的栈实现简单高效，但缺点在于容量固定，可能发生栈溢出。

3 栈的实现方式 (二): 基于链表

基于链表的栈实现提供了动态扩容的能力，其思路是使用单链表来存储元素，并将链表的头部作为栈顶。这样，入栈操作相当于在链表头部插入新节点，出栈操作则是移除头部节点。这种实现无需预先分配固定大小，因此更适合不确定数据量的场景。每个节点包含数据和指向下一个节点的指针，栈本身只需维护一个指向头部的指针。

以下是一个用 Python 实现的基于链表的栈示例代码：

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
```

```
6 class LinkedListStack:
7     def __init__(self):
8         self.head = None
9
10    def push(self, item):
11        new_node = Node(item)
12        new_node.next = self.head
13        self.head = new_node
14
15    def pop(self):
16        if self.is_empty():
17            raise Exception("Stack is empty")
18        item = self.head.data
19        self.head = self.head.next
20        return item
21
22    def peek(self):
23        if self.is_empty():
24            raise Exception("Stack is empty")
25        return self.head.data
26
27    def is_empty(self):
28        return self.head is None
29
30    def size(self):
31        count = 0
32        current = self.head
33        while current:
34            count += 1
35            current = current.next
36        return count
```

在这段代码中，我们首先定义了一个 `Node` 类来表示链表节点，每个节点包含数据和一个指向下一个节点的指针。`LinkedListStack` 类的初始化方法将 `head` 指针设为 `None`，表示空栈。`push` 方法创建新节点并将其插入到链表头部；`pop` 方法检查栈是否为空，然后返回头部数据并更新 `head` 指针。`peek` 方法类似，但不修改链表。所有核心操作的时间复杂度均为 $O(1)$ ，因为链表头部的操作是常数时间的。空间复杂度为 $O(n)$ ，每个元素需要额外的指针空间。基于链表的栈优点在于动态容量，但缺点是需要更多内存用于指针，实现稍复杂。

4 两种实现方式的对比与选择

在选择栈的实现方式时，需要根据应用场景权衡利弊。数组实现具有固定容量，性能稳定且无内存分配开销，但可能发生栈溢出；链表实现则支持动态扩容，无需担心栈满，但每个元素需要额外指针空间，且操作可能有微小内存分配开销。总体而言，如果能预估数据量上限且追求极致性能，数组实现是更好的选择；如果需要处理不确定大小的数据，链表实现更灵活。在实际开发中，还应考虑语言特性和库支持，例如在 Python 中，列表本身就可以模拟栈，但理解底层实现有助于优化和调试。

5 栈的经典应用场景实战

栈在计算机科学中有许多经典应用，其中之一是括号匹配检查。这个问题要求检查一个字符串中的括号（如 ()，[], {},）是否正确匹配和闭合。算法思路是遍历字符串，遇到左括号时入栈，遇到右括号时与栈顶左括号匹配，如果匹配则出栈，否则返回错误。最终，栈应为空表示所有括号匹配。以下是核心代码片段：

```
def is_balanced(expression):  
    1     stack = []  
    2     mapping = {')': '(', ']': '[', '}': '{'}  
    3     for char in expression:  
    4         if char in mapping.values():  
    5             stack.append(char)  
    6         elif char in mapping.keys():  
    7             if not stack or stack.pop() != mapping[char]:  
    8                 return False  
    9     10    return not stack
```

这段代码使用一个列表模拟栈，遍历表达式时处理括号。时间复杂度为 $O(n)$ ，其中 n 是表达式长度。

另一个应用是表达式求值，specifically 逆波兰表达式（后缀表达式）。例如，表达式 [2, 1, +, 3, *] 等价于 $(2 + 1) * 3$ 。算法思路是遍历表达式，遇到数字时入栈，遇到运算符时弹出两个操作数进行计算，并将结果入栈。最终栈顶即为结果。以下是核心代码片段：

```
def eval_rpn(tokens):  
    1     stack = []  
    2     for token in tokens:  
    3         if token in "+-*":  
    4             b = stack.pop()  
    5             a = stack.pop()  
    6             if token == '+':  
    7                 stack.append(a + b)  
    8             elif token == '-':  
    9                 stack.append(a - b)  
    10            elif token == '*':
```

```
12     stack.append(a * b)
13     elif token == '/':
14         stack.append(int(a / b))
15     else:
16         stack.append(int(token))
17 return stack.pop()
```

这段代码处理数字和运算符，利用栈进行中间结果存储。时间复杂度为 $O(n)$ 。

栈还广泛应用于函数调用栈 (Call Stack)，这是编程语言中管理函数调用、局部变量和返回地址的核心机制。每当函数被调用时，其信息被压入栈；函数返回时，信息被弹出。这确保了程序的正确执行流程，是栈最基础的应用之一。

通过本文，我们深入探讨了栈的核心特性、两种实现方式及其经典应用。栈作为一种后进先出的数据结构，在计算机科学中扮演着不可或缺的角色。基于数组的实现简单高效，适合固定容量场景；基于链表的实现动态灵活，适合不确定数据量的情况。经典应用如括号匹配和表达式求值展示了栈的实际价值。作为进阶思考，读者可以尝试用栈来实现队列，或者设计一个支持 $O(1)$ 时间获取最小元素的栈 (Min Stack)。此外，栈内存与堆内存的区别以及深度优先搜索 (DFS) 算法中栈的应用也是值得延伸阅读的主题。

6 互动环节

欢迎读者在评论区分享您对栈的理解或实现代码。例如，您可以尝试用栈来反转一个字符串：遍历字符串并将每个字符入栈，然后出栈即可得到反转结果。这是一个简单的练习，有助于巩固栈的操作。如果您有任何问题或想法，请随时留言讨论！