## 图解数据结构之数组、链表、栈、队列

这俩天在看集合的底层，所以顺便把数据结构给再看看...

1. 数组

**数组（Array）** 是一种很常见的数据结构

1. 相同类型的元素（element）

(2)分配一块连续的内存来存储

(3)元素的索引（index）->元素对应的存储地址

(4)供随机访问并且容量有限



1. 链表
2. 链表简介

**链表（LinkedList）** 虽然是一种线性表。

每一个节点里存到下一个节点的指针(Pointer)

不必须按顺序存储

链表在插入和删除的时候可以达到 O(1) 的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要 O(n) 的时间

充分利用计算机内存空间,实现**灵活的内存动态管理**，链表本身不会节省空间

1. 链表分类

单链表

双向链表

循环链表

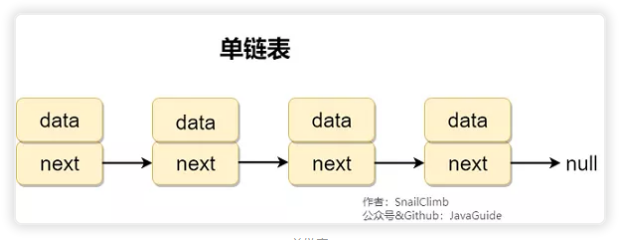
双向循环链表



1. 单链表

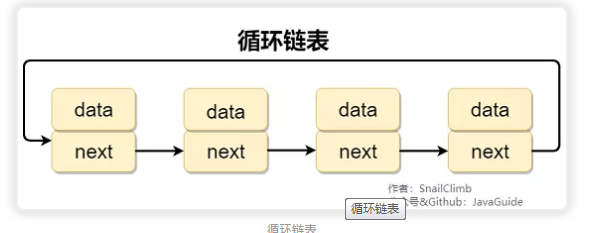
**单链表** 单向链表只有一个方向，结点只有一个后继指针 next 指向后面的节点。物理内存上是不连续的。第一个结点叫作头结点。

链表通常有一个不保存任何值的 head 节点(头结点)，通过头结点我们可以遍历整个链表。尾结点通常指向null。



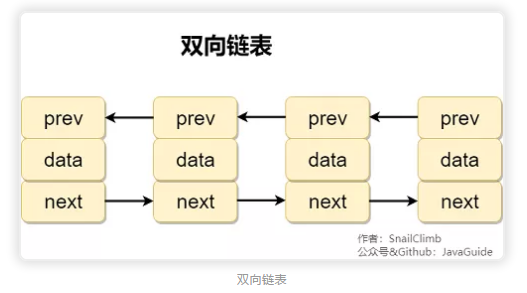
1. 循环链表

**循环链表** 其实是一种特殊的单链表，和单链表不同的是循环链表的尾结点不是指向null，而是指向链表的头结点。

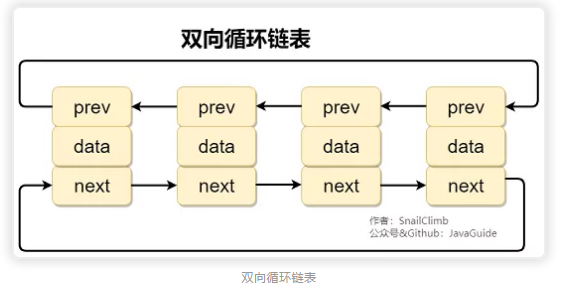


1. 双向链表

**双向链表** 包含两个指针，一个prev指向前一个节点，一个next指向后一个节点。

(4)双向循环链表

**双向循环链表** 最后一个节点的 next 指向head，而 head 的prev指向最后一个节点，构成一个环。



数组和链表的区别：

(1)数组使用的是连续内存空间对CPU的缓存机制友好，链表则相反。

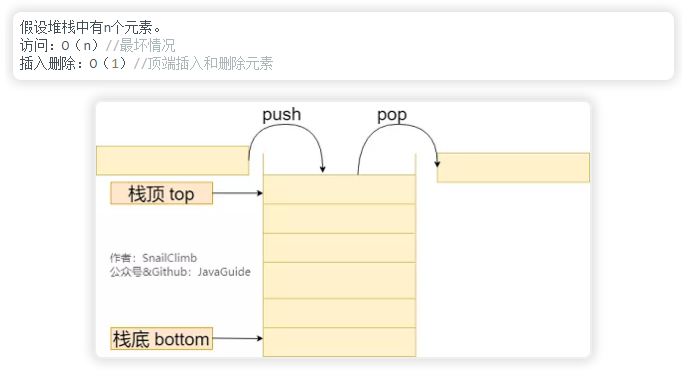
(2)数组的大小固定，声明之后就要占用所需的连续内存空间。如果声明的数组过小的话，需要再申请一个更大的内存空间，然后将原数组拷贝进去。数组多的情况下，这将是非常耗时的。链表则天然支持动态扩容。

1. 栈

**栈** (stack)只允许在有序的线性数据集合的一端（称为**栈顶 top**）进行加入数据（**push**）和移除数据（**pop**）。

按照 后进先出（LIFO, Last In First Out） 的原理运作。在栈中，push 和 pop 的操作都发生在栈顶。

栈通常用一维数组或链表来实现，数组实现的栈叫做**顺序栈**，用链表实现的栈叫做**链式栈**。

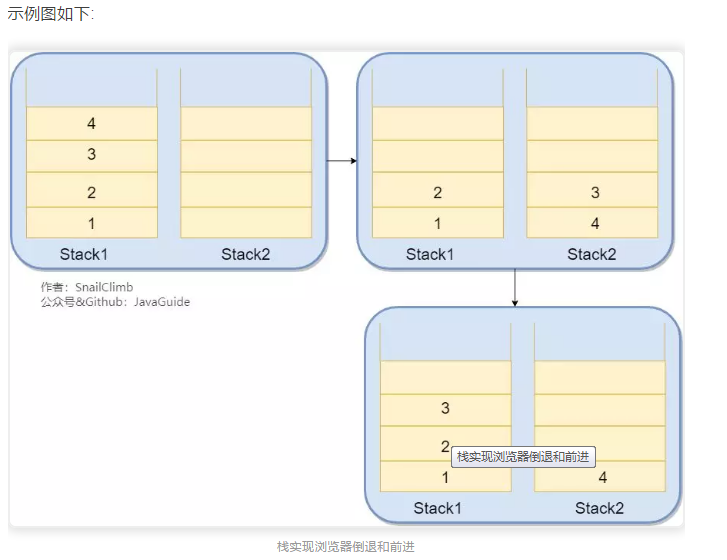


栈的常见应用场景：

1. 实现浏览器的回退和前进功能

用2个栈来实现操作。先把1.2.3.4页面依次浏览，往stack1中依次压入。

回退到2时，把回退的页面3.4压入stack2中。前进到3，就把3从stack2中取出，放入stack1中。



1. 反转字符串
2. 检查符号是否成对出现

比如 "()"、"()[]{}"、"{[]}" 都是有效字符串，而 "(]" 、"([)]" 则不是。我们可以利用栈 Stack 来解决这个问题。

(1)首先我们将括号间的对应规则存放在 Map 中，这一点应该毋容置疑；

(2)创建一个栈。遍历字符串，如果字符是左括号就直接加入stack中，否则将stack的栈顶元素与这个括号做比较，如果不相等就直接返回false。遍历结束，如果stack为空，返回 true。

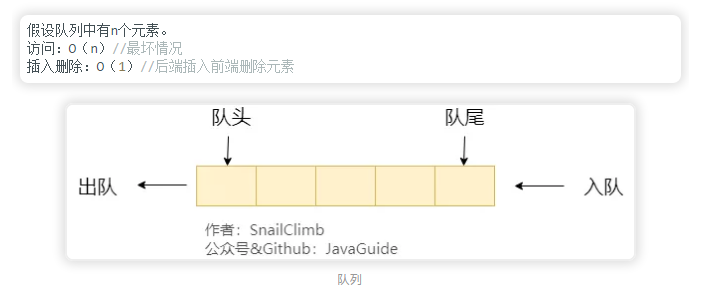
public boolean **isValid**(String s){  
    // 括号之间的对应规则  
    HashMap<Character, Character> mappings = new HashMap<Character, Character>();  
    mappings.put(')', '(');  
    mappings.put('}', '{');  
    mappings.put(']', '[');  
    Stack<Character> stack = new Stack<Character>();  
    char[] chars = s.toCharArray();  
    for (int i = 0; i < chars.length; i++) {  
        if (mappings.containsKey(chars[i])) {  
            char topElement = stack.empty() ? '#' : stack.pop();  
            if (topElement != mappings.get(chars[i])) {  
                return false;  
            }  
        } else {  
            stack.push(chars[i]);  
        }  
    }  
    return stack.isEmpty();  
}

1. 队列
2. 队列简介

**队列**是 先进先出( FIFO，First In, First Out) 的线性表。在具体应用中通常用链表或者数组来实现，用数组实现的队列叫作 顺序队列 ，用链表实现的队列叫作 链式队列 。

队列只允许在后端（rear）进行插入操作也就是 入队 enqueue，在前端（front）进行删除操作也就是出队 dequeue

队列的操作方式和堆栈类似，唯一的区别在于队列只允许新数据在后端进行添加。



1. 队列分类
2. 单队列

单队列是常见的队列，每次添加元素都是在队尾添加。顺序队列(数组实现)链式队列(链表实现)

**顺序队列存在“【假溢出】”的问题也就是明明有位置却不能添加的情况。**

假设下图是一个顺序队列，我们将前两个元素1,2 出队，并入队两个元素7,8。当进行入队、出队操作的时候，front和 rear 都会持续往后移动，当 rear 移动到最后的时候,我们无法再往队列中添加数据，即使数组中还有空余空间，这种现象就是 ”**假溢出**“ 。意思就是rear移动到最后的时候，front可以往后移动，那么数组中前面部分是有空闲空间的，但是rear不能往后移动了。除了假溢出问题之外，如下图所示，当添加元素8的时候，rear 指针移动到数组之外（越界）。

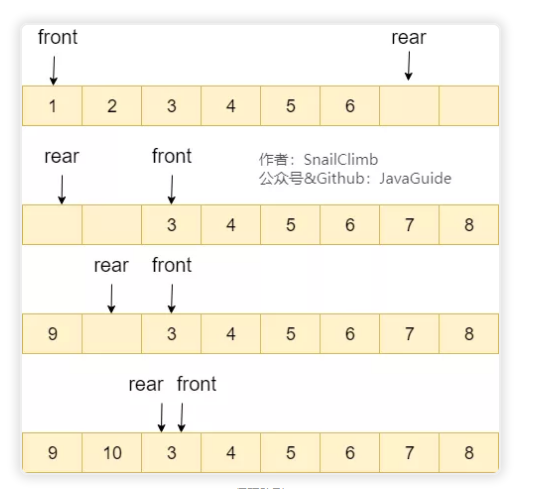


Front==rear时，顺序队列是空队列。

1. 循环队列

循环队列可以解决顺序队列的假溢出和越界问题。解决办法就是：从头开始，这样也就会形成头尾相接的循环，这也就是循环队列名字的由来。

还是用上面的图，我们将 rear 指针指向数组下标为 0 的位置就不会有越界问题了。当我们再向队列中添加元素的时候， rear 向后移动。



Front=rear表示循环队列可能为空也有可能是顺序队列满了。

解决办法有2种：

1. 设置一个标志flag;flag=0则顺序队列为空，flag=1则顺序队列为满。
2. 队列为空的时候就是 front==rear ，队列满的时候，我们保证数组还有一个空闲的位置，rear 就指向这个空闲位置，如下图所示，那么现在判断队列是否为满的条件就是： (rear+1) % QueueSize= front 。

这个判断的意思就是rear指向的这个空闲位置 是否刚好再front的前面。若是，则顺序队列已满。

1. 队列常见的应用场景

(1)阻塞队列

阻塞队列可以看做是队列基础上加上阻塞操作。

当队列为空的时候，出队操作阻塞，当队列满的时候，入队操作阻塞。使用阻塞队列我们可以很容易实现“生产者 - 消费者“模型。

(2)linux内核进程队列（按优先级排队）

(3)线程池中的请求/任务队列： 线程池中没有空闲线程时，新的任务请求线程资源时，线程池该如何处理呢？答案是将这些请求放在队列中，当有空闲线程的时候，会循环中反复从队列中获取任务来执行。

没事 拿来看看细细品尝...