> 工作上一定没人这么搞，但是考察对栈、队列理解程度的好题

232.用栈实现队列

https://leetcode-cn.com/problems/implement-queue-using-stacks/

使用栈实现队列的下列操作：

push(x) -- 将一个元素放入队列的尾部。
pop() -- 从队列首部移除元素。
peek() -- 返回队列首部的元素。
empty() -- 返回队列是否为空。

实现 MyQueue 类：
void push(int x) 将元素 x 推到队列的末尾
int pop() 从队列的开头移除并返回元素
int peek() 返回队列开头的元素
boolean empty() 如果队列为空，返回 true ；否则，返回 false

示例:

```
MyQueue queue = new MyQueue();
queue.push(1);
queue.push(2);
queue.peek();  // 返回 1
queue.pop();   // 返回 1
queue.empty(); // 返回 false
```

说明:

- 你只能使用标准的栈操作 -- 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。

- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）。

思路

这是一道模拟题，不涉及到具体算法，考察的就是对栈和队列的掌握程度。

使用栈来模式队列的行为，如果仅仅用一个栈，是一定不行的，所以需要两个栈**一个输入栈，一个输出栈**，这里要注意输入栈和输出栈的关系。

下面动画模拟以下队列的执行过程如下：

执行语句：
queue.push(1);
queue.push(2);
queue.pop(); **注意此时的输出栈的操作**
queue.push(3);
queue.push(4);
queue.pop();
queue.pop();**注意此时的输出栈的操作**
queue.pop();
queue.empty();

在push数据的时候，只要数据放进输入栈就好，**但在pop的时候，操作就复杂一些，输出栈如果为空，就把进栈数据全部导入进来（注意是全部导入）**，再从出栈弹出数据，如果输出栈不为空，则直接从出栈弹出数据就可以了。

最后如何判断队列为空呢？**如果进栈和出栈都为空的话，说明模拟的队列为空了。**

在代码实现的时候，会发现pop() 和 peek()两个函数功能类似，代码实现上也是类似的，可以思考一下如何把代码抽象一下。

C++代码如下：

```
class MyQueue {
public:
    stack<int> stIn;
    stack<int> stOut;
    /** Initialize your data structure here. */
    MyQueue() {

    }
    /** Push element x to the back of queue. */
    void push(int x) {
        stIn.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
```

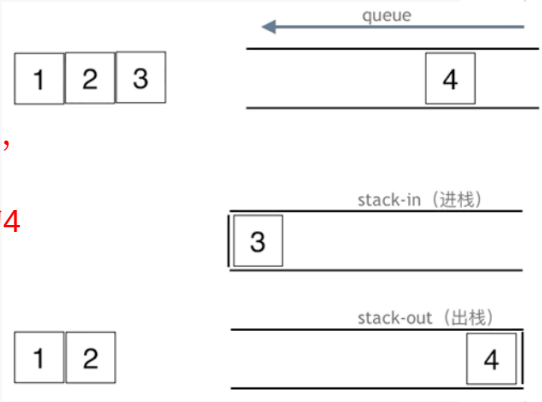
用到了stack函数：
push, empty, pop, top,

```
    int pop() {
        // 只有当stOut为空的时候，再从stIn里导入数据（导入stIn全部数据）
        if (stOut.empty()) {
            // 从stIn导入数据直到stIn为空
            while(!stIn.empty()) {
                stOut.push(stIn.top());
                stIn.pop();
            }
        }
        int result = stOut.top();
        stOut.pop();
        return result;
    }

    /** Get the front element. */
    int peek() {
        int res = this->pop(); // 直接使用已有的pop函数
        stOut.push(res); // 因为pop函数弹出了元素res，所以再添加回去
        return res;
    }

    /** Returns whether the queue is empty. */
    bool empty() {
        return stIn.empty() && stOut.empty();
    }
};
```

4在stack-in的top位置，先将4放进stack-out中，然后把stack-in的4弹出



需要再次理解？

拓展

可以看出peek()的实现，直接复用了pop()。

再多说一些代码开发上的习惯问题，在工业级别代码开发中，最忌讳的就是实现一个类似的函数，直接把代码粘过来改一改就完事了。

这样的项目代码会越来越乱，**一定要懂得复用，功能相近的函数要抽象出来，不要大量的复制粘贴，很容易出问题！（踩过坑的人自然懂）**

工作中如果发现某一个功能自己要经常用，同事们可能也会用到，自己就花点时间把这个功能抽象成一个好用的函数或者工具类，不仅自己方便，也方面了同事们。

同事们就会逐渐认可你的工作态度和工作能力，自己的口碑都是这么一点一点积累起来的！在同事圈里口碑起来了之后，你就发现自己走上了一个正循环，以后的升职加薪才少不了你！哈哈哈

其他语言版本

Java：

使用Stack(堆栈)同名方法:

```
class MyQueue {
    // java中的 Stack 有设计上的缺陷，官方推荐使用 Deque(双端队列) 代替 Stack
    Deque<Integer> stIn;
    Deque<Integer> stOut;
```

```
    /** Initialize your data structure here. */
    public MyQueue() {
        stIn = new ArrayDeque<>();
        stOut = new ArrayDeque<>();
    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        stIn.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        // 只要 stOut 为空，那么就应该将 stIn 中所有的元素倒腾到 stOut 中
        if (stOut.isEmpty()) {
            while (!stIn.isEmpty()) {
                stOut.push(stIn.pop());
            }
        }
        // 再返回 stOut 中的元素
        return stOut.pop();
    }

    /** Get the front element. */
    public int peek() {
        // 直接使用已有的pop函数
        int res = this.pop();
        // 因为pop函数弹出了元素res，所以再添加回去
        stOut.push(res);
        return res;
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        // 当 stIn 栈为空时，说明没有元素可以倒腾到 stOut 栈了
        // 并且 stOut 栈也为空时，说明没有以前从 stIn 中倒腾到的元素了
        return stIn.isEmpty() && stOut.isEmpty();
    }
}
```

个人习惯写法，使用Deque通用api：

```
class MyQueue {
    // java中的 Stack 有设计上的缺陷，官方推荐使用 Deque(双端队列) 代替 Stack
    // Deque 中的 addFirst、removeFirst、peekFirst 等方法等效于 Stack(堆栈) 中的 push、pop、p
    Deque<Integer> stIn;
    Deque<Integer> stOut;
    /** Initialize your data structure here. */
    public MyQueue() {
        stIn = new ArrayDeque<>();
        stOut = new ArrayDeque<>();
    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        stIn.addLast(x);
    }
```

```
    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        // 只要 stOut 为空，那么就应该将 stIn 中所有的元素倒腾到 stOut 中
        if (stOut.isEmpty()) {
            while (!stIn.isEmpty()) {
                stOut.addLast(stIn.pollLast());
            }
        }
        // 再返回 stOut 中的元素
        return stOut.pollLast();
    }

    /** Get the front element. */
    public int peek() {
        // 直接使用已有的pop函数
        int res = this.pop();
        // 因为pop函数弹出了元素res，所以再添加回去
        stOut.addLast(res);
        return res;
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        // 当 stIn 栈为空时，说明没有元素可以倒腾到 stOut 栈了
        // 并且 stOut 栈也为空时，说明没有以前从 stIn 中倒腾到的元素了
        return stIn.isEmpty() && stOut.isEmpty();
    }
}
```

```
class MyQueue {

    Stack<Integer> stack1;
    Stack<Integer> stack2;

    /** Initialize your data structure here. */
    public MyQueue() {
        stack1 = new Stack<>(); // 负责进栈
        stack2 = new Stack<>(); // 负责出栈
    }

    /** Push element x to the back of queue. */
    public void push(int x) {
        stack1.push(x);
    }

    /** Removes the element from in front of queue and returns that element. */
    public int pop() {
        dumpStack1();
        return stack2.pop();
    }

    /** Get the front element. */
    public int peek() {
        dumpStack1();
        return stack2.peek();
```

```
    }

    /** Returns whether the queue is empty. */
    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }

    // 如果stack2为空，那么将stack1中的元素全部放到stack2中
    private void dumpStack1(){
        if (stack2.isEmpty()){
            while (!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
```

Python：

```
# 使用两个栈实现先进先出的队列
class MyQueue:
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.stack1 = list()
        self.stack2 = list()

    def push(self, x: int) -> None:
        """
        Push element x to the back of queue.
        """
        # self.stack1用于接受元素
        self.stack1.append(x)

    def pop(self) -> int:
        """
        Removes the element from in front of queue and returns that element.
        """
        # self.stack2用于弹出元素，如果self.stack2为[],则将self.stack1中元素全部弹出给self.sta
        if self.stack2 == []:
            while self.stack1:
                tmp = self.stack1.pop()
                self.stack2.append(tmp)
        return self.stack2.pop()

    def peek(self) -> int:
        """
```

```
        Get the front element.
        """
        if self.stack2 == []:
            while self.stack1:
                tmp = self.stack1.pop()
                self.stack2.append(tmp)
        return self.stack2[-1]

    def empty(self) -> bool:
        """
        Returns whether the queue is empty.
        """
        return self.stack1 == [] and self.stack2 == []
```

Go:

```
type MyQueue struct {
    stack []int
    back  []int
}

/** Initialize your data structure here. */
func Constructor() MyQueue {
    return MyQueue{
        stack: make([]int, 0),
        back:  make([]int, 0),
    }
}

/** Push element x to the back of queue. */
func (this *MyQueue) Push(x int) {
    for len(this.back) != 0 {
        val := this.back[len(this.back)-1]
        this.back = this.back[:len(this.back)-1]
        this.stack = append(this.stack, val)
    }
    this.stack = append(this.stack, x)
}

/** Removes the element from in front of queue and returns that element. */
func (this *MyQueue) Pop() int {
    for len(this.stack) != 0 {
        val := this.stack[len(this.stack)-1]
        this.stack = this.stack[:len(this.stack)-1]
        this.back = append(this.back, val)
    }
    if len(this.back) == 0 {
        return 0
    }
    val := this.back[len(this.back)-1]
    this.back = this.back[:len(this.back)-1]
    return val
}

/** Get the front element. */
func (this *MyQueue) Peek() int {
```

```
    for len(this.stack) != 0 {
        val := this.stack[len(this.stack)-1]
        this.stack = this.stack[:len(this.stack)-1]
        this.back = append(this.back, val)
    }
    if len(this.back) == 0 {
        return 0
    }
    val := this.back[len(this.back)-1]
    return val
}

/** Returns whether the queue is empty. */
func (this *MyQueue) Empty() bool {
    return len(this.stack) == 0 && len(this.back) == 0
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * obj := Constructor();
 * obj.Push(x);
 * param_2 := obj.Pop();
 * param_3 := obj.Peek();
 * param_4 := obj.Empty();
 */
```

javaScript:

```
// 使用两个数组的栈方法（push, pop） 实现队列
/**
* Initialize your data structure here.
*/
var MyQueue = function() {
   this.stack1 = [];
   this.stack2 = [];
};

/**
* Push element x to the back of queue.
* @param {number} x
* @return {void}
*/
MyQueue.prototype.push = function(x) {
   this.stack1.push(x);
};

/**
* Removes the element from in front of queue and returns that element.
* @return {number}
*/
MyQueue.prototype.pop = function() {
   const size = this.stack2.length;
   if(size) {
       return this.stack2.pop();
   }
   while(this.stack1.length) {
       this.stack2.push(this.stack1.pop());
```

```
    }
    return this.stack2.pop();
};

/**
* Get the front element.
* @return {number}
*/
MyQueue.prototype.peek = function() {
    const x = this.pop();
    this.stack2.push(x);
    return x;
};

/**
* Returns whether the queue is empty.
* @return {boolean}
*/
MyQueue.prototype.empty = function() {
    return !this.stack1.length && !this.stack2.length
};
```

- 作者微信：程序员Carl
- B站视频：代码随想录
- 知识星球：代码随想录