基础-第三章

用数组实现大小固定的队列和栈

栈

> 设固定的数组大小为N，设置一个栈顶指针，指向栈顶的下一个位置，入栈时直接插入然后指针向上走一个，出栈时指针先向下移动一个，然后弹出元素

```
public static class ArrayStack {
 private Integer[] arr;
 private Integer size;

 public ArrayStack(int initSize) {
     if(initSize < 0) {
         throw new IllegalArgumentException("The init size is less than
0");
     }
     arr = new Integer[initSize];   // 初始化栈大小
     size = 0; // 栈顶指针
 }

 public Integer peek() {   // 返回栈顶元素的值但不弹出
     if(size == 0) {
         return null;
     }
     return arr[size - 1];
 }

 public void push(int obj) {   // 入栈
     if(size == arr.length) {
         throw new ArrayIndexOutOfBoundsException("The stack is full");
     }
     arr[size++] = obj;
 }

 public Integer pop() {   // 出栈
     if(size == 0) {
         throw new ArrayIndexOutOfBoundsException("The stack is empty");
     }
     return arr[--size];
 }
}
```

队列

> 设固定大小的数组为N，设置一个队头指针用来出队，设置一个队尾指针用来入队，队头指针和队尾指针均可循环转圈，即当队尾指针到数组最后一个值时有重新跳到数组开始，这里采用size来进行解耦，实现队尾指针和队首指针的循环。

```
public static class ArrayQueue {
 private Integer[] arr;
 private Integer size;
 private Integer first;
 private Integer last;

 public ArrayQueue(int initSize) {
     if(initSize < 0) {
         throw new IllegalArgumentException("The inti size is less than
0");
     }
     arr = new Integer[initSize];  // 初始化队列
     size = 0;   // 用size进行解耦
     first = 0;
     last = 0;
 }

 public Integer peek() {
     if(size == 0) {
         return null;
     }
     return arr[first];
 }

 public void push(int obj) {   // 出队
     if(size == arr.length) {
         throw new ArrayIndexOutOfBoundsException("The queue is full");
     }
     size++;
     arr[last] = obj;
     last = last == arr.length - 1 ? 0 : last + 1;
 }

 public Integer pop() {
     if(size == 0) {
         throw new ArrayIndexOutOfBoundsException("The queue is empty");
     }
     size--;
     int tmp = first;
```

```
        first = fitst == arr.length - 1 ? 0 : first - 1;
        return arr[tmp];
    }
}
```

实现一个栈，要求实现返回栈中最小元素操作。要求：pop、push和getMin操作的时间复杂度都是O(1)

> 创建两个栈，一个栈用来正常压入数据，另一个栈仅用来压入最小值，弹出时同步弹出即可。
>
> - 当栈为空时，压入的第一个元素同时也为最小值，将其存入最小栈中。
> - 当栈不为空，将压入的元素与最小栈的栈顶元素比较，当新压入元素较大时，将最小栈的栈顶元素**再次入栈**，当新压入的元素较小时，将新压入元素直接压入最小栈的栈顶。
> - 出栈时，最小栈的栈顶始终为当前栈的最小值。

```
public static class MinStack2 {
    private Stack<Integer> stackData;
    private Stack<Integer> stackMin;

    public MinStack2() {
        this.stackData = new Stack<Integer>();
        this.statckMin = new Stack<Integer>();
    }

    public void push(int obj) {
        if(this.stackMin.is.Empty()) {
            this.stackMin.push(obj);
        } else if(obj <= this.getmin()) {
            this.stackMin.push(obj);
        } else {    // 当新压入元素大于最小栈栈顶元素时，将最小栈栈顶元素重新入栈
            int newMin = this.stackMin.peek();
            this.stackMin.push(newMin);
        }
        this.stackData.push(obj);
    }

    public int pop() {
        if(this.stackData.isEmpty()) {
            throw new RuntimeException("Your stack is empty.");
        }
        this.stackMin.pop();    // 最小栈和数据栈同时出栈
        return this.stackData.pop();
    }
```

```
    public int getmin() {
        if(this.stackMin.isEmpty()) {
            throw new RuntimeException("Your stack is empty.");
        }
        return this.stackMin.peek();
    }
}
```

> 此处可以进行改进。当新入栈元素**大于**最小栈的栈顶元素时，仅将数据压入数据栈，最小栈保持不变，当出栈时，只有当数据栈中待出栈的元素与最小栈的栈顶元素**相等**时最小栈才出栈，否则保持不变。此时最小栈的栈顶原始始终是当前数据栈中的最小值。

```
public static class MinStack1 {
    private Stack<Integer> stackData;
    private Stack<Integer> stackMin;

    public MinStack1() {
        this.stackData = new Stack<Integer>();
        this.statckMin = new Stack<Integer>();
    }

    public void push(int obj) {
        if(this.stackMin.is.Empty()) {
            this.stackMin.push(obj);
        } else if(obj <= this.getmin()) {    // 只有当待入栈元素小于最小栈的栈顶时最小栈才会入栈
            this.stackMin.push(obj);
        }
        this.stackData.push(obj);
    }

    public int pop() {
        if(this.stackData.isEmpty()) {
            throw new RuntimeException("Your stack is empty.");
        }
        int value = this.stackData.pop();
        if(value == this.getmin()) {    // 出栈时，只有当待出栈元素等于最小栈栈顶元素时才会出栈
            this.stackMin.pop();
        }
        return value;
    }

    public int getmin() {
        if(this.stackMin.isEmpty()) {
```

```
            throw new RuntimeException("Your stack is empty.");
        }
        return this.stackMin.peek();
    }
}
```