# 算法 - 栈和队列

* [算法 - 栈和队列](#算法---栈和队列)
  + [栈](#栈)
    - [1. 数组实现](#1-数组实现)
    - [2. 链表实现](#2-链表实现)
  + [队列](#队列)

## 栈

public interface MyStack<Item> extends Iterable<Item> {  
  
 MyStack<Item> push(Item item);  
  
 Item pop() throws Exception;  
  
 boolean isEmpty();  
  
 int size();  
  
}

### 1. 数组实现

public class ArrayStack<Item> implements MyStack<Item> {  
  
 // 栈元素数组，只能通过转型来创建泛型数组  
 private Item[] a = (Item[]) new Object[1];  
  
 // 元素数量  
 private int N = 0;  
  
  
 @Override  
 public MyStack<Item> push(Item item) {  
 check();  
 a[N++] = item;  
 return this;  
 }  
  
  
 @Override  
 public Item pop() throws Exception {  
  
 if (isEmpty()) {  
 throw new Exception("stack is empty");  
 }  
  
 Item item = a[--N];  
  
 check();  
  
 // 避免对象游离  
 a[N] = null;  
  
 return item;  
 }  
  
  
 private void check() {  
  
 if (N >= a.length) {  
 resize(2 \* a.length);  
  
 } else if (N > 0 && N <= a.length / 4) {  
 resize(a.length / 2);  
 }  
 }  
  
  
 /\*\*  
 \* 调整数组大小，使得栈具有伸缩性  
 \*/  
 private void resize(int size) {  
  
 Item[] tmp = (Item[]) new Object[size];  
  
 for (int i = 0; i < N; i++) {  
 tmp[i] = a[i];  
 }  
  
 a = tmp;  
 }  
  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
  
 @Override  
 public Iterator<Item> iterator() {  
  
 // 返回逆序遍历的迭代器  
 return new Iterator<Item>() {  
  
 private int i = N;  
  
 @Override  
 public boolean hasNext() {  
 return i > 0;  
 }  
  
 @Override  
 public Item next() {  
 return a[--i];  
 }  
 };  
  
 }  
}

### 2. 链表实现

需要使用链表的头插法来实现，因为头插法中最后压入栈的元素在链表的开头，它的 next 指针指向前一个压入栈的元素，在弹出元素时就可以通过 next 指针遍历到前一个压入栈的元素从而让这个元素成为新的栈顶元素。

public class ListStack<Item> implements MyStack<Item> {  
  
 private Node top = null;  
 private int N = 0;  
  
  
 private class Node {  
 Item item;  
 Node next;  
 }  
  
  
 @Override  
 public MyStack<Item> push(Item item) {  
  
 Node newTop = new Node();  
  
 newTop.item = item;  
 newTop.next = top;  
  
 top = newTop;  
  
 N++;  
  
 return this;  
 }  
  
  
 @Override  
 public Item pop() throws Exception {  
  
 if (isEmpty()) {  
 throw new Exception("stack is empty");  
 }  
  
 Item item = top.item;  
  
 top = top.next;  
 N--;  
  
 return item;  
 }  
  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
  
 @Override  
 public Iterator<Item> iterator() {  
  
 return new Iterator<Item>() {  
  
 private Node cur = top;  
  
  
 @Override  
 public boolean hasNext() {  
 return cur != null;  
 }  
  
  
 @Override  
 public Item next() {  
 Item item = cur.item;  
 cur = cur.next;  
 return item;  
 }  
 };  
  
 }  
}

## 队列

下面是队列的链表实现，需要维护 first 和 last 节点指针，分别指向队首和队尾。

这里需要考虑 first 和 last 指针哪个作为链表的开头。因为出队列操作需要让队首元素的下一个元素成为队首，所以需要容易获取下一个元素，而链表的头部节点的 next 指针指向下一个元素，因此可以让 first 指针链表的开头。

public interface MyQueue<Item> extends Iterable<Item> {  
  
 int size();  
  
 boolean isEmpty();  
  
 MyQueue<Item> add(Item item);  
  
 Item remove() throws Exception;  
}

public class ListQueue<Item> implements MyQueue<Item> {  
  
 private Node first;  
 private Node last;  
 int N = 0;  
  
  
 private class Node {  
 Item item;  
 Node next;  
 }  
  
  
 @Override  
 public boolean isEmpty() {  
 return N == 0;  
 }  
  
  
 @Override  
 public int size() {  
 return N;  
 }  
  
  
 @Override  
 public MyQueue<Item> add(Item item) {  
  
 Node newNode = new Node();  
 newNode.item = item;  
 newNode.next = null;  
  
 if (isEmpty()) {  
 last = newNode;  
 first = newNode;  
 } else {  
 last.next = newNode;  
 last = newNode;  
 }  
  
 N++;  
 return this;  
 }  
  
  
 @Override  
 public Item remove() throws Exception {  
  
 if (isEmpty()) {  
 throw new Exception("queue is empty");  
 }  
  
 Node node = first;  
 first = first.next;  
 N--;  
  
 if (isEmpty()) {  
 last = null;  
 }  
  
 return node.item;  
 }  
  
  
 @Override  
 public Iterator<Item> iterator() {  
  
 return new Iterator<Item>() {  
  
 Node cur = first;  
  
  
 @Override  
 public boolean hasNext() {  
 return cur != null;  
 }  
  
  
 @Override  
 public Item next() {  
 Item item = cur.item;  
 cur = cur.next;  
 return item;  
 }  
 };  
 }  
}