

一个码农的博客

目录视图

摘要视图

RSS 订阅

个人资料



伯努力不努力

关注

发私信

访问：284213次

积分：3799

等级：BLOG 5

排名：第8594名

原创：89篇

转载：0篇

译文：0篇

评论：184条

文章搜索

博客专栏



Kotlin学习之路

文章：0篇

阅读：0

文章分类

Android (9)

Material Design (6)

架构设计 (3)

自定义控件 (5)

性能优化 (9)

开发笔记 (4)

插件化系列 (10)

混合开发 (1)

开源框架解析 (7)

安卓源码解析 (16)

设计模式 (10)

热修复系列 (3)

java (7)

Java数据结构与算法解析(二)——栈

标签：数据结构 算法 栈

2017-09-03 12:44

3922人阅读

评论(0)

分类：

数据结构与算法 ( 1 )

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)

[+]

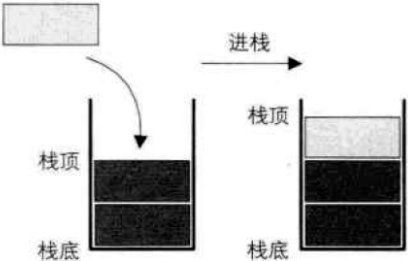
相关文章:

[Java数据结构与算法解析\(一\)——表](#)

栈是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫做栈顶。对栈的基本操作有push(进栈)和pop(出栈)，对空栈进行push和pop，一般被认为栈ADT的一个错误。当push时空间用尽是一个实现限制，而不是ADT错误。栈有时又叫做LIFO（后进先出）表。

基本概念

允许插入和删除的一端称为栈顶（top），另一端称为栈底（bottom），不含任何数据元素的栈称为空栈。栈又称为后进先出的线性表



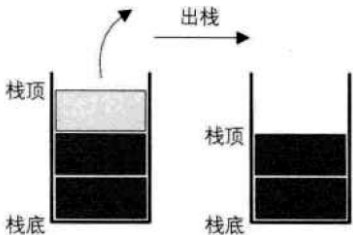


图 4-2-2

图 4-2-3

栈的顺序存储结构

栈的数序结构可以使用数组来实现，栈底是:下标为0的一端

http://blog.csdn.net/u012124438/article/details/77815937

1/10

数据结构与算法 (2)

文章存档

2017年09月 (1)

2017年08月 (5)

2017年07月 (6)

2017年06月 (4)

2017年05月 (14)

展开

阅读排行

Android Gradle知识梳理 (20535)

一篇博客让你了解RxJava (16805)

深入解析OkHttp3 (10738)

全面解析Notification (10104)

一篇博客理解Recyclerview的... (8881)

设计模式学习之策略模式 (7362)

浅谈安卓中的MVP模式 (7159)

Android进程保活全攻略 (上) (6530)

Android性能优化系列之布局... (6363)

Android热修复学习之旅——... (5892)

评论排行

设计模式学习之策略模式 (15)

全面解析Notification (14)

一篇博客让你了解RxJava (9)

一篇博客让你了解Material D... (8)

Android性能优化系列之布局... (7)

浅谈安卓中的MVP模式 (7)

Android Studio常用技巧汇总 (7)

Android性能优化系列之内存... (7)

一篇博客理解Recyclerview的... (7)

《深入理解java虚拟机》学习... (7)

推荐文章

\* CSDN日报20170828——《4个方法快速打造你的阅读清单》

\* Android检查更新下载安装

\* 动手打造史上最简单的 Recycleview 侧滑菜单

\* TCP网络通讯如何解决分包粘包问题

\* 程序员的八重境界

\* 四大线程池详解

最新评论

一篇博客让你了解Material Design的使用暗夜ノ使者 : 很好, 感谢分享!

一篇博客让你了解Material Design的使用来自星星的谢广坤 : 牛BI了我的哥

一篇博客让你了解Material Design的使用礼枝书笙 : 受用了, 赞赞

一篇博客让你了解Material Design的使用十四期-苏怡仙 : 又懂了点, 感谢你的分享

一篇博客让你了解Material Design的使用qq\_39980761 : 很好很强大

一篇博客让你了解Material Design的使用qq\_37867965 : 不错

一篇博客让你了解Material Design的使用mg52033 : 简单易懂

一篇博客让你了解Material Design的使用

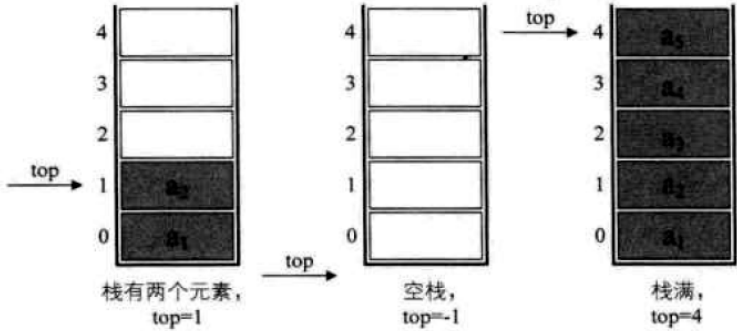
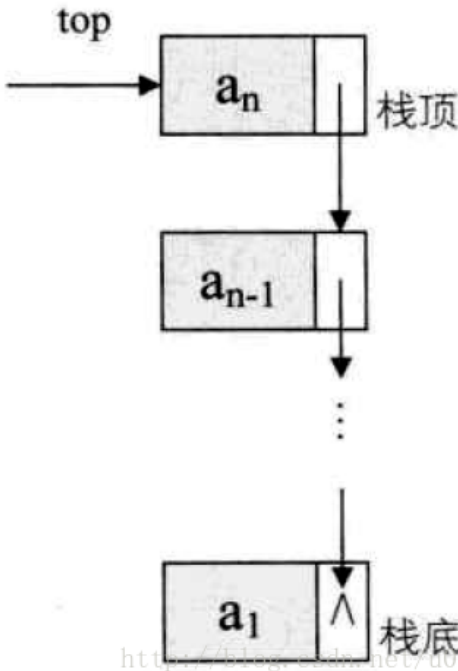


图 4-4-2 <http://blog.csdn.net/u012124438>

栈的链式存储结构



栈的实现

栈的实现，一般分为两种形式，链式结构和数组。两者均简化了ArrayList和LinkedList中的逻辑。

栈的数组实现

抽象出栈的必备接口

```
1 public interface Stack<T> {
2
3     boolean isEmpty();
4
5     void push(T data);
6
7     T pop();
8
9     int size();
10 }
```

栈的数组实现形式

```
1 public class ArrayStack<T> implements Stack<T>, Iterable {
2
3     private T[] mArray;
```

天一方蓝 :赞一个

滴滴插件化框架VirtualAPK原理解析 (一...  
Lin\_Zero :谢谢博主,看懂了,讲得很好!

务实java基础之IO

Xanthuim :注意字符流底层还是字节流,  
类方便读写才进行封装的。

```

4     private int mStackSize;
5
6     private static final int DEFAULT_CAPACITY = 10;
7
8     public ArrayStack(int capacity) {
9         if (capacity < DEFAULT_CAPACITY) {
10             ensureCapacity(DEFAULT_CAPACITY);
11         } else {
12             ensureCapacity(capacity);
13         }
14     }
15
16
17     public boolean isEmpty() {
18         return mStackSize == 0;
19     }
20
21
22     public int size() {
23         return mStackSize;
24     }
25
26     public void push(T t) {
27         if (mStackSize == mArray.length) {
28             ensureCapacity(mStackSize * 2 + 1);
29         }
30         mArray[mStackSize++] = t;
31     }
32
33     public T pop() {
34         if (isEmpty()) {
35             throw new EmptyStackException();
36         }
37         T t = mArray[--mStackSize];
38         mArray[mStackSize] = null;
39         //调整数组的大小,防止不必要的内存开销
40         if (mStackSize > 0 && mStackSize < mArray.length / 4) {
41             ensureCapacity(mArray.length / 2);
42         }
43         return t;
44     }
45
46
47     private void ensureCapacity(int newCapacity) {
48         T[] newArray = (T[]) new Object[newCapacity];
49         for (int i = 0; i < mArray.length; i++) {
50             newArray[i] = mArray[i];
51         }
52         mArray = newArray;
53     }
54
55     @Override
56     public Iterator iterator() {
57         return null;
58     }
59
60     private class ArrayStackIterator implements Iterator<T> {
61
62         @Override
63         public boolean hasNext() {
64             return mStackSize > 0;
65         }
66
67         @Override
68         public T next() {
69             return
70                 mArray[--mStackSize];
71         }
72     }
73 }
74

```

对象游离

Java的垃圾收集策略是回收所有无法被访问对象的内存，如果我们pop()弹出对象后，不调用如下代码，就会造成游离，因为数组中仍然持有这个对象的引用，保存一个不需要的对象的引用，叫做游离。

```
1 | mArray[mStackSize] = null;
```

动态调整数组大小

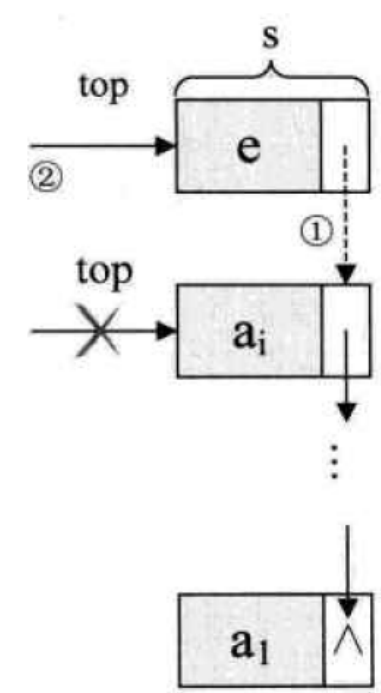
pop()中，删除栈顶元素后，如果栈的大小小于数组的1/4，就将数组的大小减半，防止空间溢出，使用率也不会小于1/4。

栈的链表实现

采用链式存储结构的栈，由于我们操作的是栈顶一端，因此这里采用单链表（不带头结点）作为基础，直接实现栈的添加，获取，删除等主要操作即可。

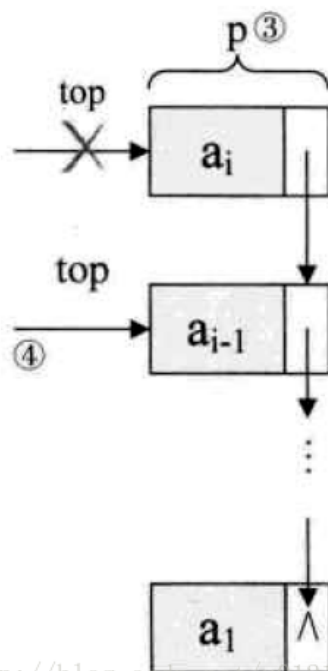
链栈的出入栈操作

链栈的入栈操作：



```
s->data = e;
s->next = stack->top;
stack->top = s;
stack->count++;|
```

链栈的出栈操作：



<http://blog.csdn.net/u012124438>

```
p = stack->top;
stack->top = p->next;
free(p);
stack->count--;
```

“

public class LinkedStack implements Stack, Iterable {

```
private int mSize;
private Node<T> endNote;
private int modCount;

public LinkedStack() {
    init();
}

private void init() {
    endNote = new Node<T>(null, null);
    modCount++;
}

@Override
public boolean isEmpty() {
    return mSize == 0;
}

@Override
public void push(T data) {
    Node<T> newNote = new Node<T>(data, null);
    endNote.mNext = newNote;
    mSize++;
    modCount++;
}

@Override
public T pop() {
    if (endNote.mNext == null) {
        throw new NoSuchElementException();
    }
    T t = endNote.mNext.mData;
```

```
        endNote.mNext = endNote.mNext.mNext;
        mSize--;
        modCount++;
        return t;
    }

    @Override
    public int size() {
        return mSize;
    }

    @Override
    public Iterator iterator() {
        return new LinkedStackIterator();
    }

    private static class Node<T> {

        private Node<T> mNext;
        private T mData;

        public Node(T data, Node<T> next) {
            mData = data;
            mNext = next;
        }
    }

    private class LinkedStackIterator implements Iterator<T> {
        private Node<T> currentNode = endNote.mNext;
        private int expectedModCount = modCount;

        @Override
        public boolean hasNext() {
            return currentNode != null;
        }

        @Override
        public T next() {
            if (modCount != expectedModCount) {
                throw new ConcurrentModificationException();
            }
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            T t = currentNode.mData;
            currentNode = currentNode.mNext;
            return t;
        }
    }
}
```

}

## 时间复杂度对比

### 顺序栈复杂度

操作	时间复杂度
空间复杂度(用于N次push)	O(n)
push()	O(1)
pop()	O(1)
isEmpty()	O(1)

链式栈复杂度

操作	时间复杂度
空间复杂度(用于N次push)	O(n)
push()	O(1)
pop()	O(1)
isEmpty()	O(1)

可知栈的主要操作都可以在常数时间内完成，这主要是因为栈只对一端进行操作，即只能对栈顶元素。

栈的经典实用

逆波兰表达式法

标准四则运算表达式—中缀表达式

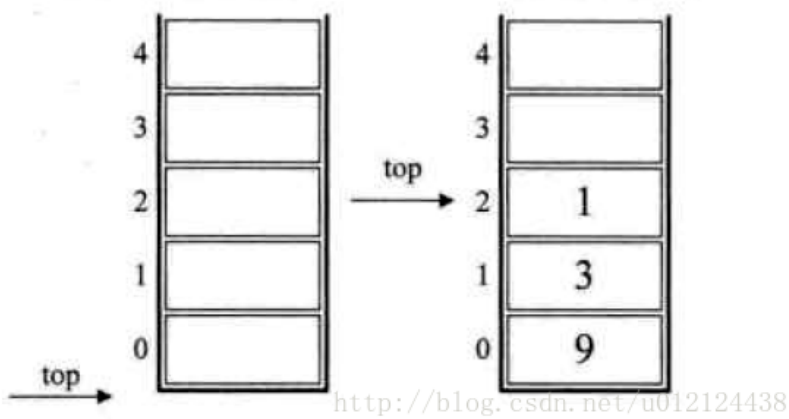
9+ (3-1) ×3+10÷2

我们在小学学习的四则运算表达式就是中缀表达式，但是计算机是不认识中缀表达式的，它采用的是后缀表达式

计算机采用—后缀表达式

9 3 1-3\*+10 2 / +

计算规则：

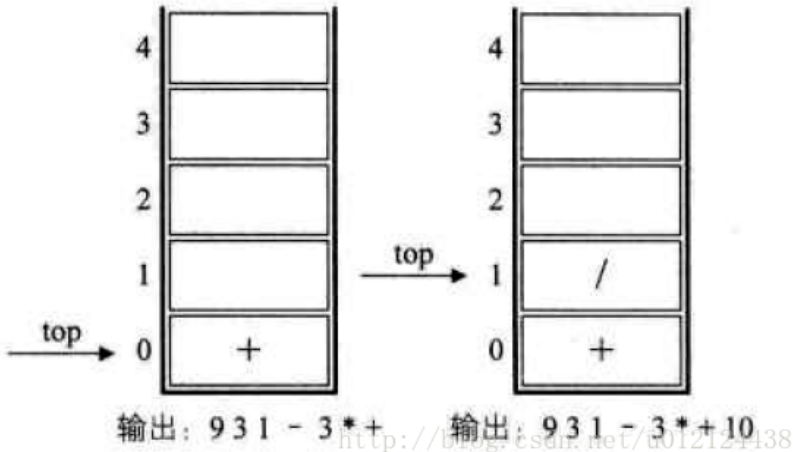
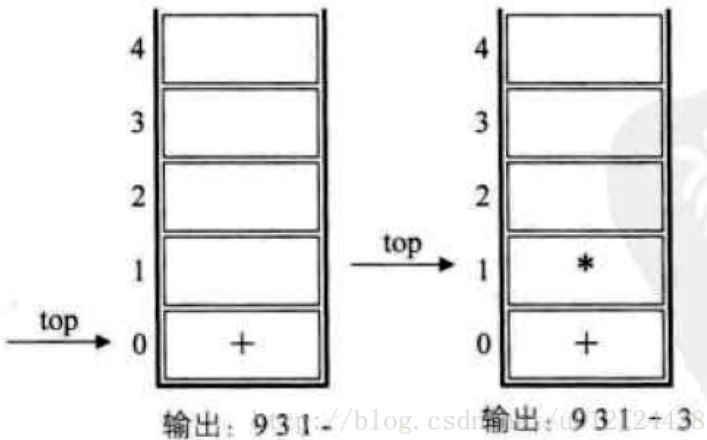
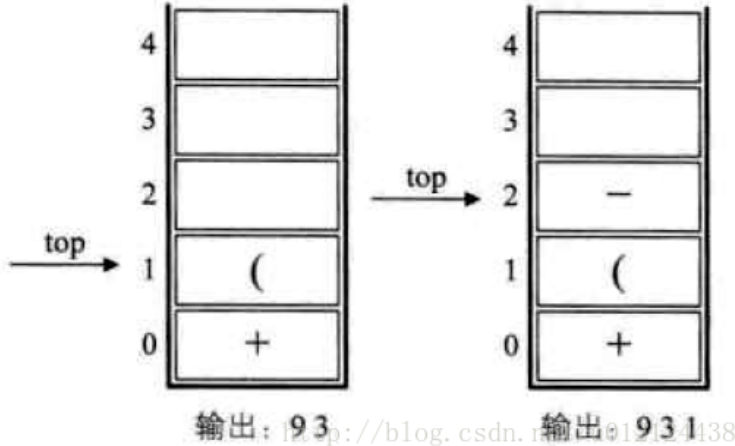
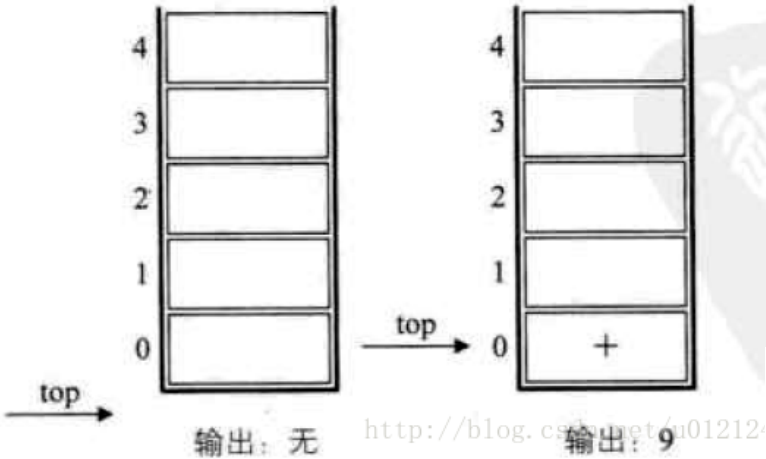


它的规则是，从头开始遍历，遇到数字进行压栈，遇到运算符号，将栈顶开始的两个元素进行运算符操作后，弹栈，结果进栈，931遇到“-”时，进行3-1=2，将2进栈，然后3进栈，遇到“\*”，3\*2=6进栈，遇到“+”，进行9+6=15进栈，然后10和2进栈，遇到“/”，进行10/2后结果进栈，最后是15+5=20，就完成了后缀表达式的计算操作。

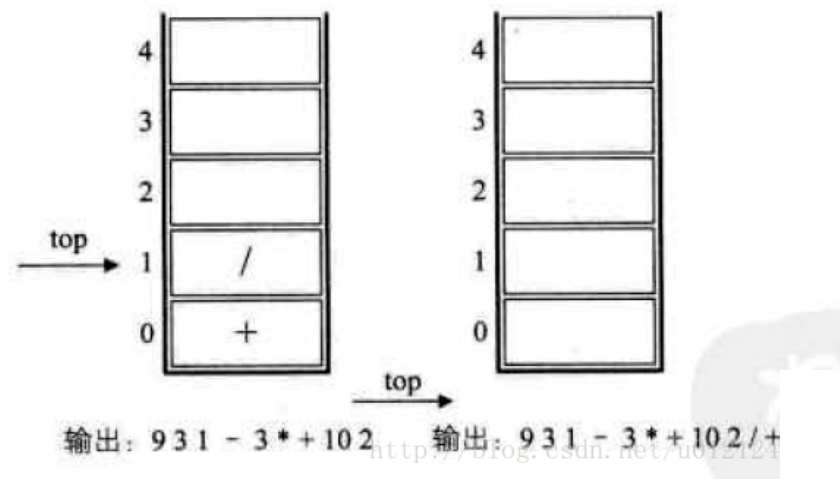
中缀表达式转后缀表达式

中缀表达式“9+(3-1)×3+10÷2”转化为后缀表达式“9 3 1-3\*+10 2 / +”

数字输出，运算符进栈，括号匹配出栈，是当栈顶是运算符时，又压进来一个运算符，如果压进来的运算符优先级比栈顶的高，则这个压进来的运算符出栈。







如果我们见到任何其他的符号（+，\*，()，那么我们从栈中弹出栈元素直到发现优先级最低。有一个例外：除非是在处理一个)的时候，否则我们决不从栈中移走（。对于（的优先级最低，而)的优先级最高。当从栈弹出元素的工作完成后，我们再将操作符压入栈中。

顶 6 踩 0

• [上一篇](#) Java数据结构与算法解析(一)——表

相关文章推荐

- [Java数据结构和算法\(三\)——简单排序](#)
- [【直播】系统集成工程师必过冲刺--任铄](#)
- [Java数据结构与经典算法——高手必会](#)
- [【直播】机器学习30天系统掌握--唐宇迪](#)
- [Java数据结构与算法解析\(一\)——表](#)
- [【直播】AI时代，机器学习该如何入门--唐宇迪](#)
- [数据结构与算法分析——Java语言描述.pdf](#)
- [【套餐】Linux应用和网络编程实战套餐--朱有鹏](#)
- [Java数据结构和算法\(四\)——栈](#)
- [【课程】SharePoint 2016 开发教程--杨建宇](#)
- [数据结构与算法分析——Java语言描述](#)
- [【课程】程序员简历优化指南--安晓辉](#)
- [\(五\) Java数据结构与算法\(第二版\)笔记——栈](#)
- [数据结构与算法（java）——链表](#)
- [数据结构与算法分析——Java简版](#)
- [数据结构与算法Java版——双向链表](#)

查看评论

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

\* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

