算法·第四版

---

1 基础

1.1 基础编程模型

（1）二分查找

- 正常查找

```
public static int rank(int key, int[] a) {
    int lo = 0;
    int hi = a.length() - 1;
    if (lo <= hi) {
        mid = lo + (hi - lo) / 2;
        if (key < a[mid]) {
            hi = mid - 1;
        }
        else if (key > a[mid]) {
            lo = mid + 1;
        }
        else {
            return mid
        }
    }
    return -1;
}
```

- 递归查找

```
public static int rank(int key, int[] a) {
    return rank(key, a, 0, a.length() - 1);
}
public static int rank(int key, int[] a, int lo, int hi) {
    //如果key存在于a中，其索引不会大于lo且不会小于hi
    if (lo > hi) return -1;
    int mid = lo + (hi - lo) / 2;
    if (key < a[mid]) rank(key, a, lo, mid - 1);
    else if(key > a[mid]) rank(key, a, mid + a, hi);
    else return mid;
}
```

（2）判定一个数是否为素数

```
public static boolean isPrime(int N) {
    if (N < 2) {
        return false;
    }
    for (int i = 0; i * i < N; i++) {
        if (N % i == 0) {
            return false;
        }
    }
    return true;
}
```

（3）调和级数

$$S_n = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \qquad (n = 1, 2, 3, \ldots, n)$$

```
public static int H(int N) {
    double sum = 0.0;
    for (int i = 1; i <= N; i++) {
        sum += 1.0 / i;
    }
    return sum;
}
```

1.2 数据抽象

（1） 重写(override)与重载(overload)

- 重写(override)
  - override是重写(覆盖)了一个方法，以实现不同的功能。一般是用于子类在继承父类时，重写(重新实现)父类中的方法。方法名与参数相同;
  - 重写(覆盖)的规则:
    - 重写方法的参数列表必须完全与被重写的方法的相同,否则不能称其为重写而是重载;
    - 重写方法的访问修饰符一定要大于被重写方法的访问修饰符(public > protected > default > private);
    - 重写的方法的返回值必须和被重写的方法的返回一致;
    - 重写的方法所抛出的异常必须和被重写方法的所抛出的异常一致，或者是其子类;
    - 被重写的方法不能为private，否则在其子类中只是新定义了一个方法，并没有对其进行重写;
    - 静态方法不能被重写为非静态的方法(会编译出错)。

- 重载(overload)
  - 一般是用于在一个类内实现若干重载的方法，这些方法的名称相同而参数形式不同。方法名相同参数不同。
  - 重载的规则:
    - 在使用重载时只能通过相同的方法名、不同的参数形式实现。不同的参数类型可以是不同的参数类型，不同的参数个数，不同的参数顺序(参数类型必须不一样);
    - 不能通过访问权限、返回类型、抛出的异常进行重载;
    - 方法的异常类型和数目不会对重载造成影响;
  - 多态的概念比较复杂，有多种意义的多态，一个有趣但不严谨的说法是：继承是子类使用父类的方法，而多态则是父类使用子类的方法。
  - 一般，我们使用多态是为了避免在父类里大量重载引起代码臃肿且难于维护。

（2） 字符串表示的习惯(String和toString())

- 每个Java类型都会从Object继承toString()方法因此一般情况下都会被新的数据类型重写;
- 如果不重写toString()，默认返回对象内存地址;
- String类型虽然为引用类型，但是具有不可变性，重新赋值后会在内存中重新生成一个内存块存放数据并指向该区域，表现如下:

```
String str1 = "hello";
String str2 = str1;
str1 = "world";
StdOut.println(str1);//输出为world
StdOut.println(str2);//输出为hello
```

（3） 对象的等价性

- Java约定equals()必须是一种 等价性 关系，必须具有如下性质:
  - 自反性，x.equals(x)为true;
  - 对称性，当且仅当y.equals(x)为ture时，x.equals(y)返回ture;
  - 传递性，如果x.equals(y)和y.equals(z)均为true，x.equals(z)也将为true
  - 另外，它必须接受一个Object为参数并满足以下性质:
    - 一致性，当两个对象均未被修改时，反复调用x.equals(y)总会返回相同的值;
    - 非空性，x.equals(null)总时会返回false。

（4） Java变量类型

- 参数变量：方法的签名定义了参数变量，在方法调用的时候参数会被初始化为调用者提供的值;
- 局部变量：声明和初始化都在方法的主体内;
- 实例变量：或者称为全局变量，为类的对象保存了数据类型的值，其作用域是整个类。

(5)设计API(只为用例提供它们所需要的)

- API可能会难以实现：实现的开发困难，甚至不可能;
- API可能会难以使用：用例代码甚至比没有API时更复杂;
- API的范围可能太窄：缺少用例所需的方法;
- API的范围可能太宽：不会被任何用例调用的方法。这种缺陷可能是最常见的，并且也是最难以避免的。API的大小一般会随着时间而增长，因为向已有的API中添加新方法很简单，但在不破坏已有用例程序的前提下从中删除方法却很困难;
- API可能会太粗略：无法提供有效的抽象;
- API坑能会太详细：抽象过于细致或是发散而无法使用;
- API可能会过于依赖某种特定的数据表示：用例代码可能会因此无法从数据表示的细节中解脱出来。因为数据表示显然是抽象数据类型实现的核心。

（6） 不可变性

- 不可变 数据类型：指的是该类型的对象中值在创建以后就无法被改变
  - Java通过final修饰符来强制保证不可变性；不会被任何用例掉调用的方法
  - 当一个变量声明为final时，也就保证只可对它赋值一次；
  - 不可以变性的缺点是：需要为每个值创建一个新对象

（7） 判断字符串是否回文或者回环变位

- 单字符串是否回文检测

```
public static boolean isPalindrome(String s) {
    int N = s.length();
    for (int i = 0; i < N / 2; i++) {
        if (s.charAt(i) != s.charAt(N-1-i)) {
            return false;
        }
    }
    return true;
}
```

- 双字符串移位互为回环变位检测(如ACTGACG和TGACGAC)

```
public static boolean isPalindrome(String s, String t) {
    String tmp = s.concat(s);
    if (s.length() == t.length() && s.concat(s).indexof(t) > 0)     {
        return true;
    } else {
        return false;
    }

}
```

（8） Date数据类型的实现

- 常规实现

```
public class Date {
    private final int month;
    private final int day;
    private final int year;

    public Date(int m, int d, int y) {
        month = m;day = d;year = y;
    }

    public int month() {
        return month;
    }

    public int day() {
        return day;
    }

    public int year() {
        return year;
    }

    @Override
    public String toString() {
        return month() + "/" + day() + "/" + year();
    }
}
```

- 小存储空间实现

```java
public class Date {
    private final int value;

    public Date(int m, int d, int y) {
        value = y * 512 + m * 32 + d;
    }

    public int month() {
        return (value / 32) % 16;
    }

    public int day() {
        return value % 32;
    }

    public int year() {
        return value / 512;
    }

    @override
    public String toString() {
        return month() + "/" + day() + "/" + year();
    }
}
```

（9） 日期换算星期

- 蔡勒公式(以1582年10月4日为例)
  - 1582年10月4日前:

$$W = (d + 1 + 2 \times m + \frac{3 \times (m + 1)}{5} + y + \frac{y}{4} + 5)\%7$$

  - 1582年10月4日后:

$$W = (d + 1 + 2 \times m + \frac{3 \times (m + 1)}{5} + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400})\%7$$

  - 说明:
    - 式中的除法都是表示向下取整后的值
    - W：星期；W对7取模得：0-星期日，1-星期一，2-星期二，3-星期三，4-星期四，5-星期五，6-星期六
    - y：年(后2位数)
    - m：月(m大于等于3，小于等于14，即在公式中，某年的1、2月要看作上一年的13、14月来计算，比如2003年1月1日要看作2002年的13月1日来计算)
    - d：日
- 基姆拉尔森计算公式

$$W = (d + 2 \times m + \frac{3 \times (m + 1)}{5} + y + \frac{y}{4} - \frac{y}{100} + \frac{y}{400})\%7$$

  - 说明:
    - 式中的除法都是表示向下取整后的值
    - W：星期；W对7取模得：0-星期一，1-星期二，2-星期三，3-星期四，4-星期五，5-星期六，6-星期日
    - y：年(4位数)
    - m：月(m大于等于3，小于等于14，即在公式中，某年的1、2月要看作上一年的13、14月来计算，比如2003年1月1日要看作2002年的13月1日来计算)
    - d：日

```
public String dayOfTheWeek() {
        // Zeller formula
        int month = this.month;
        int year = this.year;
        if (month <= 2) {
            month += 12;
            year--;
        }
        int week = (day + 2 * month + 3 * (month + 1) / 5 +      year + year / 4
                      - year / 100 + year / 400) % 7;
        switch (week) {
        case 0:
            return "Monday";
        case 1:
            return "Tuesday";
        case 2:
            return "Wednesday";
        case 3:
            return "Thursday";
        case 4:
            return "Friday";
        case 5:
            return "Saturday";
        case 6:
            return "Sunday";
        default:
            return null;
        }
    }
```

1.3 背包、队列和栈用例收集

(1) 背包(Bag):

- 概念：背包是一种不支持删除元素的集合数据类型；
- 作用：背包的目的是帮助用例收集元素并迭代遍历所有收集到的元素，也可以检查背包是否为空或者获取背包中元素的数量；
- 特点：迭代的顺序不确定且和用例无关；
- 用例代码：

```
public class Stats {
    public static void main(String[] args) {
        StdOut.printf("Start:");
        Bag<Double> numbers = new Bag<Double>();
        while (!StdIn) {
            numbers.add(StdIn.readDouble());
        }
        int N = numbers.size();

        double sum = 0.0;
        for (double x : numbers) {
            sum += x;
        }
        double mean = sum / N;

        sum = 0.0;
        for (double x : numbers) {
            sum += (x - mean) * (x - mean);
        }
        double std = Math.sqrt(sum / (N - 1));

        StdOut.printf("Means: %.2f\n", mean);
        StdOut.printf("Std dev: %.2f\n", std);
        StdOut.printf("Over!");
    }
}
```

- 结果显示：

```
Start:
100
99
101
120
98
107
109
81
101
90
Mean: 100.6
Std dev: 10.51
Over!
```

（2） 先进先出队列(Queue)

- 概念：一种基于先进先出(FIFO)策略的集合类型；
- 具体事例：剧院门前排队的人们、收费站前排队的汽车；
- 用例代码：

```
public  static int[] readInts(String name) {
    In in = new In(name);
    Queue<Integer> q = new Queue<Integer>();
    while (!in.isEmpty()) {
        q.enqueue(in.readIn());
    }

    int N = q.size();
    int[] a = new int[N];
    for (int i = 0; i < N; i++) {
        a[i] = q.dequeue();
    }
    return a;
}
```

（3） 下压栈(Stack)

- 概念：基于后进后出(LIFO)策略的集合类型；
- 实际用例：收信时，新的在最上面，旧的在最下面；堆叠一摞文件，从最上面的开始查看；
- 用例描述：
  - 目的：程序读入并计算一个由字符串构成的算术表达式的值；
  - 递归定义：算术表达式可能是一个数，或者由一个左括号、一个算术表达式、一个运算符、另一个算术表达式和一个右括号组成的表达式；
  - 算法核心：两个栈(一个用于保存运算符，一个用于保存操作数)；
  - 算法处理的4种情况
    - 将操作数压入操作数栈；
    - 将运算符压入运算符栈；
    - 忽略左括号；
    - 在遇到右括号时，弹出一个运算符，弹出所需数量的操作数，并将运算符和操作数的运算结果压入操作数栈；
- Dijkstra算法代码：

```
public class Reverse {
    public static void main(String[] args) {
        StdOut.println("Start:");
        StdOut.println("请输入计算公式，且每个数字字符或者符号用空格分  隔: ");
        String[] tmpStrArray = StdIn.readLine().split(" ");
        Stack<String> ops = new Stack<>();
        Stack<Double> vals = new Stack<>();
        for (String s : tmpStrArray) {
            //读取字符，如果是运算符就压入栈
            if (s.equals("("))                    ;
            else if (s.equals("+"))      ops.push(s);
            else if (s.equals("-"))      ops.push(s);
            else if (s.equals("*"))      ops.push(s);
            else if (s.equals("/"))      ops.push(s);
            else if (s.equals("sqrt"))   ops.push(s);
            else if (s.equals(")")) {
                //如果字符为")"，弹出运算符和操作数，计算结果并压入栈
                String op = ops.pop();
                double v = vals.pop();
                if (op.equals("+"))          v += vals.pop();
                else if (op.equals("-"))     v -= vals.pop();
                else if (op.equals("*"))     v *= vals.pop();
                else if (op.equals("/"))     v /= vals.pop();
                else if (op.equals("sqrt"))  v = Math.sqrt(v);
                vals.push(v);
            }
            else {
                //如果字符既非运算符也不是括号，将他作为double值压入栈
                vals.push(Double.parseDouble(s));
            }
        }
        StdOut.println(vals.pop());
        StdOut.println("Over!");
    }
}
```

- Dijkstra结果显示：

```
Start:
请输入计算公式，且每个数字字符或者符号用空格分隔:
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
Over!
```

- 定容字符串栈用例代码：

```
public class Exc_01_014_CapacityStackTest {
    public static void main(String[] args) {
        StdOut.println("Start:");
        FixedCapacityStackOfStrings s;
        s = new FixedCapacityStackOfStrings(100);
        In in = new In(".\\.\\algs4-data\\tobe.txt");
        while (!in.isEmpty()) {
            String item = in.readString();
            if (!item.equals("-")) {
                s.push(item);
            } else if (!s.isEmpty()) {
                StdOut.print(s.pop() + " ");
            }
        }
        in.close();
        StdOut.println("(" + s.size() + " left on stack)");
        StdOut.println("Over!");
    }
}
```

```java
public class FixedCapacityStackOfStrings {
    private String[] a; //StackEntries
    private int N;      //Size

    public FixedCapacityStackOfStrings(int cap) {
        a = new String[cap];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public int size() {
        return N;
    }

    public void push(String item) {
        a[N++] = item;
    }

    public String pop() {
        return a[--N];
    }

}
```

- 定容字符串栈结果显示：
  - 原字符串为：

    ```
    to be or not - be - - that - - - is
    ```

  - 输出结果为：

    ```
    Start:
    to be not that or be (2 left on stack)
    Over!
    ```

（4）泛型

- 概念：参数化类型，一种可以储存任意类型的数据的集合类的抽象数据类型；
- 实现：
  - 注意：创建泛型在Java中是不行的，即以下方式是不能直接实现的；

    ```java
    a = new Item[cap];
    ```

  - Java中的实现采用类型转换来实现；

    ```java
    a = (Item[]) new Object[cap];
    ```

- 泛型定容栈用例代码：

```java
public class Exc_01_015_Generic {
    public static void main(String[] args) {
        StdOut.println("Start:");
        FixedCapacityStack<String> s;
        s = new FixedCapacityStack<>(100);
        In in = new In(".\\.\\algs4-data\\tobe.txt");
        while (!in.isEmpty()) {
            String item = in.readString();
            if (!item.equals("-")) {
                s.puch(item);
            } else if (!s.isEmpty()) {
                StdOut.print(s.pop() + " ");
            }
        }
        in.close();
        StdOut.println("(" + s.size() + " left on stack)");
        StdOut.println("Over!");
    }
}
```

```java
public class FixedCapacityStack<Item> {
    private Item[] a;   //stack entries
    private int N;      //size

    public FixedCapacityStack(int cap) {
        a = (Item[]) new Object[cap];
    }

    public boolean isEmpty() {
        return N == 0;
    }

    public int size() {
        return  N;
    }

    public void puch(Item item) {
        a[N++] = item;
    }

    public Item pop() {
        return a[--N];
    }
}
```

- 泛型定容栈结果显示：
  - 原字符串为：

    ```
    to be or not - be - - that - - - is
    ```

  - 输出结果为：

    ```
    Start:
    to be not that or be (2 left on stack)
    Over!
    ```

（5） 迭代

- 下压栈(能够动态调整数组大小的实现)

```java
import java.util.Iterator;

public class ResizingArrayStack<Item> implements Iterable {
    private Item[] a = (Item[]) new Object[1];  //栈元素
    private int N;                              //元素数量

    public boolean isEmpty() {
        return N == 0;
    }

    public int size() {
        return N;
    }

    public void resize(int max) {
        //将元素移动到一个大小为max的数组
        Item[] temp = (Item[]) new Object[max];
        for (int i = 0; i < N; i++) {
            temp[i] = a[i];
        }
        a = temp;
    }

    public void push(Item item) {
        //将元素添加到栈顶
        if (a.length == N) {
            resize(a.length * 2);
        }
        a[N++] = item;
    }

    public Item pop() {
        //从栈顶删除元素
        Item item = a[--N];
        a[N] = null;
        if (N > 0 && N == a.length / 4) {
            resize(a.length / 2);
        }
        return item;
    }

    @Override
    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements   Iterator<Item> {
        //支持后进先出的迭代
        private int i = N;

        @Override
        public boolean hasNext() {
            return i > 0;
        }

        @Override
        public Item next() {
            return a[--i];
        }

        @Override
        public void remove() {

        }
    }
}
```

（5） 链表

- 概念：一种递归的数据结构，他或者为空(null)，或者指向一个节点(node)的引用，该节点含有一个泛型元素和指向另一条链表的引用；
- 构造：

```
public class Node
{
   Item item;
   Node next;
}
```

- 链表栈实现

```
package MyClass;

import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {
    private Node first; //栈顶(最近添加的元素)
    private int N;      //元素数量

    /**
     * 定义节点嵌套类
     */
    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() {
        return first == null;   //或者 N == 0;
    }

    public int size() {
        return N;
    }

    public void push(Item item) {
        Node oldlist = first;
        first = new Node();
        first.item = item;
        first.next = oldlist;
        N++;
    }

    public Item pop() {
        //从栈顶删除元素
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }

    @Override
    public Iterator<Item> iterator() {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;

        public boolean hasNext() {
            return first != null;
        }

        public void remove() {
        }

        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

- 链表队实现

```java
package MyClass;

import java.util.Iterator;

public class Queue<Item> implements Iterable<Item> {
    private Node first; //指向最早添加节点的链接
    private Node last;  //指向最近添加结点的链接
    private int N;      //队列中的元素数量

    /**
     * 定义结点的嵌套类
     */
    private class Node {
        Item item;
        Node Next;
    }

    private boolean isEmpty() {
        return first == null;   //或者 N == 0;
    }

    public void push(Item item) {
        Node oldlist = last;    //向尾部添加元素
        last.item = item;
        if (first == null) {
            first = last;
        } else {
            oldlist.Next = last;
        }
        N++;
    }

    public Item dequeue() {
        Item item = first.item; //从表头删除元素
        first = first.Next;
        if (isEmpty()) {
            last = null;
        }
        N--;
        return item;
    }

    public Iterator<Item> iterator() {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;

        public boolean hasNext() {
            return current == null;
        }

        public void remove() {

        }

        public Item next() {
            Item item = current.item;
            first = first.Next;
            return item;
        }
    }
}
```

- 链表包实现

```java
package MyClass;

import java.util.Iterator;

public class Bag<Item> implements Iterable<Item> {
    private Node first;

    private class Node {
        Item item;
        Node next;
    }

    public void add(Item item) {
        Node oldlist = first;
        first = new Node();
        first.item = item;
        first.next = oldlist;
    }

    public Iterator<Item> iterator() {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;
        public boolean hasNext() {
            return first == null;
        }
        public void remove() {

        }
        public Item next() {
            Item item = current.item;
            current =  current.next;
            return item;
        }
    }
}
```