巍巍交大　百年书香
www.jiaodapress.com.cn
bookinfo@sjtu.edu.cn

丛书策划　张荣昌
责任编辑　王　清　孟海江
封面设计

大数据、云计算、人工智能、信息安全人才培养丛书
“互联网+”新形态一体化教材

大数据
大数据基础
数据可视化
数据清洗与治理
Hadoop应用与开发
数据挖掘基础
SEO搜索引擎优化
MySQL数据库
R语言程序设计
Go语言程序设计

云计算
云计算基础
虚拟化与容器
云安全运维
网络工程与组网技术
现代通信技术
路由交换技术
无线网络技术
现代网络SDN技术
数据网组建与维护
局域网组建与维护
云数据中心架构与SDN技术

人工智能
人工智能基础
物联网基础
深度学习
机器学习

信息安全
信息安全基础
Linux服务器安全高级运维
Web安全与防御
防火墙技术与应用
计算机病毒与防范
数据存储与恢复
密码学基础
计算机网络安全运维
网络设备配置与综合实战
无线网络安全技术
VPN虚拟专用网安全
终端数据存储与恢复
工控安全
渗透测试
恶意代码分析
网络空间安全态势感知
数据库安全技术
网络安全协议
企业级数据安全与灾备管理
信息安全法律法规
终端数据安全及防泄密

专业基础
Linux操作系统基础
计算机网络基础
Ubuntu服务器管理
Windows Server 2016配置与管理
● 数据结构
Python程序设计
Java程序设计
C语言程序设计
C#程序设计
Android程序设计
XML基础教程
JavaScript基础教程
Web前端开发
OpenStack应用与开发
数据结构与算法
静态网页设计与制作
HTML5与JavaScript程序设计
数据库设计与应用
数据库应用基础
UML建模与设计模式
ERP原理与应用
综合布线

扫描二维码
关注上海交通大学出版社
官方微信

ISBN 978-7-313-25964-6
9 787313 259646 >
定价：58.00元

『十四五』职业教育江苏省规划教材
数据结构
主编◎陈仲珊　王珊珊　罗　春
上海交通大学出版社

“十四五”职业教育江苏省规划教材

数据结构

SHUJU JIEGOU

主编◎陈仲珊　王珊珊　罗　春

上海交通大學出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

“十四五”职业教育江苏省规划教材

数据结构

SHUJU JIEGOU

主　编◎陈仲珊　王珊珊　罗　春
副主编◎宁鹏飞　刘广峰　侯天江

扫一扫
学习资源库

◆素材源码
◆教学课件
◆电子教案

上海交通大学出版社
SHANGHAI JIAO TONG UNIVERSITY PRESS

内容提要

本书以 Java 为基础，通过丰富的实例讲解数据结构的相关知识。全书共 9 章，包括 Java 与面向对象程序设计、数据结构与算法基础、线性表、栈与队列、递归、树、图、查找和排序。本书采用 Java 论述数据结构（组织大量数据的方法）并进行算法分析（算法运行时间的估计），通过大量的实例为读者展示了如何使用数据结构实现有效的算法，并分析和测试了算法的性能，本书可作为计算机相关专业的教学用书，也可作为相关技术人员培训或工作的参考用书。

图书在版编目（CIP）数据

数据结构 / 陈仲珊，王珊珊，罗春主编 . —上海：
上海交通大学出版社，2022.7（2023.12 重印）

ISBN 978-7-313-25964-6

Ⅰ. ①数… Ⅱ. ①陈… ②王… ③罗… Ⅲ. ①数据结构—高等学校—教材 Ⅳ. ① TP311.12

中国版本图书馆 CIP 数据核字（2022）第 098880 号

数据结构
SHUJU JIEGOU

主　　编：陈仲珊　王珊珊　罗　春
出版发行：上海交通大学出版社
邮政编码：200030
印　　制：北京荣玉印刷有限公司
开　　本：889 mm × 1194 mm　1/16
字　　数：340 千字
版　　次：2022 年 7 月第 1 版
书　　号：ISBN 978-7-313-25964-6
定　　价：58.00 元

地　　址：上海市番禺路 951 号
电　　话：6407 1208

经　　销：全国新华书店
印　　张：16

印　　次：2023 年 12 月第 2 次印刷

前　言

随着信息技术的发展和推广，各行各业都采用计算机进行数据存储和处理，而数据结构是计算机存储、组织数据的方式，因此了解数据结构与算法显得尤为重要。对于计算机科学与技术、计算机信息管理与应用、电子商务等专业的高校学生，以及从事 IT 行业的从业者而言，要想更好、更有效地使用计算机，充分发挥计算机的性能，就必须学习和掌握数据结构的有关知识。

人们常常将 Java 与 C++ 比较，并经常把 Java 看成一种比 C++ 更安全、更具有可移植性并且更容易使用的语言。因此，这使得它成为讨论和实现基础数据结构的一种优秀的核心语言。随着计算机的速度越来越快，对于能够处理大量输入数据的程序的需求变得日益迫切。可是，由于在输入量很大的时候，低效率变得非常明显，因此这又要求对低效率问题给予更仔细的关注。通过在实际编程之前对算法的分析，可以确定某个特定的解法是否可行。希望本书可以教授读者良好的程序设计技巧和算法分析能力，使读者能够以更高的效率开发出程序。

本书采用 Java 论述数据结构（组织大量数据的方法）并进行算法分析（算法运行时间的估计），通过大量的实例为读者展示了如何使用数据结构实现有效的算法，并分析和测试了算法的性能。本书分 9 章为读者讲解数据结构的相关知识，以及用 Java 实现各种算法的方法，具体内容包括 Java 与面向对象程序设计、数据结构与算法基础、线性表、栈与队列、递归、树、图、查找和排序。

本书在编写上具有以下特色：

（1）通过章前的“知识与技能目标”“知识导图”明确本章要学习的内容，帮助读者做好学习准备；通过文中的“算法说明”等，丰富知识内容，提高读者的学习兴趣。

（2）在精练的语言的基础上，充分利用丰富的实例讲解数据结构与算法知识，内容由浅入深、循序渐进，符合初学者的认知规律。

（3）本书落实立德树人根本任务，贯彻《高等学校课程思政建设指导纲要》精神，通过“思政园地”模块，将专业知识与思政教育有机结合，推动价值引领、知识传授和能力培养紧密结合。

此外，编者还为广大一线教师提供了服务于本书的教学资源库，有需要者可致电 13810412048，或发邮件至 2393867076@qq.com 领取。

为与代码格式保持一致，本书科技符号均用正体表示。

本书可作为计算机相关专业的教学用书，也可作为相关技术人员培训或工作的参考用书。由于编写时间仓促，网络技术发展迅猛，书中存在的不足和疏漏之处，敬请广大读者批评指正，在此表示衷心的感谢。

编　者

2022 年 5 月

目　录

第 4 章　栈与队列 / 69

第 5 章　递归 / 89

第 6 章　树 / 109

第 7 章　图 / 141

第 1 章 Java 与面向对象程序设计

知识与技能目标

（1）掌握 Java 语言基础知识。

（2）了解 Java 面向对象的特性，并能熟练使用。

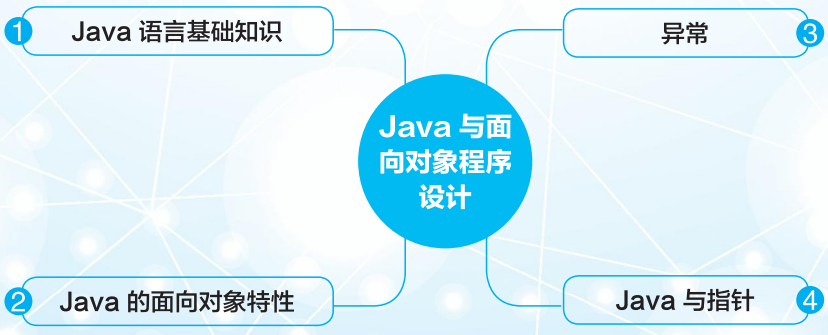
（3）掌握 Java 中的异常和指针。

情感目标

（1）厚植家国情怀，用专业知识武装自己，激发科技报国的家国情怀和使命担当。

（2）把个人梦融入中国梦，深刻理解科技成果是奋斗来的，奋斗没有重点。

知识导图



笔记

本章导读

在这一章中向读者简要介绍有关 Java 的基础知识。Java 语言是一种广泛使用并且具有许多良好的如面向对象、可移植性、健壮性等特性的计算机高级程序设计语言，在这里对 Java 的介绍不可能面面俱到，因此在第 1 章中只对 Java 代码的相关知识进行介绍。熟悉 Java 的读者可以不阅读本章。

1.1 Java 语言基础知识

思政园地：
华为鸿蒙系统

1.1.1 基本数据类型及运算

在 Java 中每个变量在使用前均必须声明它的类型。Java 共有八种基本数据类型：四种整型，两种浮点型，一种字符型，以及用于表示真假的布尔类型。各种数据类型的细节如表 1-1 所示。

表 1-1　Java 数据类型

类型	存储空间 /bit	范围
int	32	[-2147483648,2147483647]
short	16	[-32768,32767]
long	64	[-9223372036854775808, 9223372036854775807]
byte	8	[-128,127]
float	32	[-3.4E38,3.4E38]
double	64	[-1.7E308,1.7E308]
char	16	Unicode 字符
boolean	1	True,False

声明一个变量[①]时，应先给出此变量的类型，随后写上变量名。在声明变量时，一行中可以有多个变量，并且可以在声明变量的同时对变量进行初始化。例如：

```
int i;
double x, y = 1.2;
char c = 'z';
boolean flag;
```

在程序设计中，常常需要在不同的数字数据类型之间进行转换。图 1-1 给出了数字类型间的合法转换。

①为和代码中的正斜体格式一致，本书中的科技符号统一使用正体表示。

图 1-1 中 6 个实箭头表示无信息损失的转换，而 3 个虚箭头表示的转换则可能会丢失精度。

有时在程序设计中也需要进行在图 1-1 中没有出现的转换，在 Java 中这种数字转换是可以进行的，不过信息可能会丢失。在可能丢失信息的情况下进行的转换是通过强制类型转换来完成的。其语法是在需要进行强制类型转换的表达式前使用圆括号，圆括号中是需要转换的目标类型。例如：

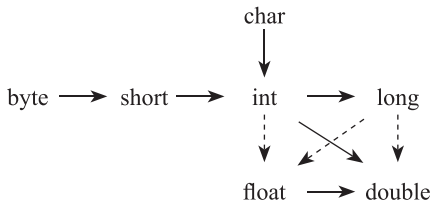


图 1-1　数字类型间的合法转换

```
double x = 7.8;
int n = (int)x;          //x 等于 7
```

Java 使用常见的算术运算符 +、–、*、/ 进行加、减、乘、除的运算。当除法运算符 / 用于两个整数时，是进行整数除法。整数的模（求余）运算使用 % 运算符。对整型变量一种最常见的操作就是递增与递减运算，与 C/C++ 一样，Java 也支持递增和递减运算。例如：

```
int n = 7, m = 2;
double d = 7;
n = n / m;               //n 等于 3
d /= m;                  //d 等于 3.5
n--;                     //n 等于 2
int a = 2 * n++;         //a 等于 4
int b = 2 * ++m;         //b 等于 6
```

此外，Java 还具有完备的关系运算符，如 = =（是否相等），<（小于），>（大于），< =（小于等于），> =（大于等于），!=（不等于）；并且 Java 使用 && 表示逻辑与，|| 表示逻辑或，！表示逻辑非；以及 7 种位运算符 &（与）、|（或）、^（异或）、~（非）、>>（右移）、<<（左移）、>>>（高位填充 0 的右移）。

最后 Java 还支持一种三元运算符 ?:，这个运算符有时很有用。它的形式为

```
condition ? e1 : e2
```

这是一个表达式，在 condition 为 true 时返回值为 e1，否则为 e2。例如：

```
min = x < y ? x : y;
```

则 min 为 x 与 y 中的较小值。

1.1.2 流程控制语句

计算机高级语言程序设计中共有 3 种流程结构，分别是顺序、分支、循环。其中，分

笔记

支与循环流程结构需要使用固定语法的流程控制语句来完成。

Java 中有两种语句可用于分支结构，一种是 if 条件语句，另一种是 switch 多选择语句。条件语句的形式如下：

```
if (condition) statement1 else statement2
```

当 if 后的条件 condition 的值为 true 时执行 statement1 中的语句，否则执行 statement2 中的语句。

多选择语句的形式为

```
switch (integer expression){
    case value1: block1;break;
    case value2: block2; break;
    …
    case valueN: blockN;break;
    default: default block;
}
```

switch 语句从与选择值相匹配的 case 标签处开始执行，一直执行到下一个 break 处或者 switch 的末尾。如果没有相匹配的 case 标签，而且存在 default 子句，那么执行 default 子句。如果没有相匹配的 case 标签，并且没有 default 子句，则结束 switch 语句的执行，执行 switch 后面的语句。

Java 中的循环语句主要有 3 种，分别是 for 循环、while 循环、do…while 循环。

for 循环的形式为

```
for (initialization; condition; increment) statement;
```

for 语句循环控制的第一部分通常是对循环变量的初始化，第二部分给出进行循环的测试条件，第三部分则是对循环变量的更新。

while 循环的形式为

```
while (condition) statement;
```

while 循环首先对循环条件进行测试，只有在循环条件满足的情况下才执行循环体。

do…while 循环的形式为

```
do statement while (condition);
```

与 while 循环不同的是，do…while 循环首先执行一次循环体，当循环条件满足时再继续进行下一次循环。

1.1.3 字符串

笔记

字符串是指一个字符序列。在 Java 中没有内置的字符串类型，而是在标准 Java 库中包含一个名为 String 的预定义类。每个被一对双引号括起来的字符序列均是 String 类的一个实例。字符串可以使用如下方式定义：

```
String s1 = null;      //s1 指向 null
String s2 = "";        //s2 是一个不包含字符的空字符串
String s3 = "Hello";
```

Java 允许使用符号“+”把两个字符串连接在一起。当连接一个字符串和一个非字符串时，后者首先被转换成字符串，然后进行连接。例如：

```
s3 = s3 + "World!";          //s3 为 "HelloWorld!"
String s4 = "abc" + 123;     //s4 为 "abc123"
```

Java 的 String 类包含许多方法，其中多数非常有用，表 1-2 给出了常用的一些方法。

表 1-2　JAVA String 类常用方法及说明

返回值类型	方法及说明
char	charAt(int index) Returns the char value at the specified index
int	compareTo(String anotherString) Compares two strings lexicographically
int	compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences
boolean	endsWith(String suffix) Tests if this string ends with the specified suffix
boolean	equals(Object anObject) Compares this string to the specified object
boolean	equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations
int	indexOf(String str) Returns the index within this string of the first occurrence of the specified substring
int	lastIndexOf(String str) Returns the index within this string of the right most occurrence of the specified substring
int	length() Returns the length of this string
boolean	startsWith(String prefix) Tests if this string starts with the specified prefix
String	substring(int beginIndex) Returns a new string that is a substring of this string

笔记

续表

返回值类型	方法及说明
String	substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string
char[]	toCharArray() Converts this string to a new character array
String	toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale
String	toString() This object (which is already a string!) is itself returned
String	toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale
String	trim() Returns a copy of the string, with leading and trailing white space omitted

如果读者需要进一步了解有关 String 提供的其他方法及方法完成的功能，可以通过在线 API（应用程序接口）文档了解相关信息，从中可以查到标准库中所有的类及方法。API 文档是 Java SDK 的一部分，以 HTML 格式显示。 JDK1.5.0 的 API 文档地址为 http://java.sun.com/j2se/1.5.0/docs/api/index.html。

1.1.4 数组

数组是用来存放一组具有相同类型数据的数据结构。可以通过整型下标来访问数组中的每一个值。数组可以通过在某种数据类型后面加上 [ ] 来定义，在此之后跟上变量名就可以定义相应类型的数组变量了。例如：

```
int[] a;
```

还可以使用另一种方法定义数组，例如：

```
int a[];
```

以上这两种方法的定义是等价的。在这里只定义了一个整型数组变量 a，但是还没有将 a 真正地初始化为一个数组。为将一个数组初始化可以使用 new 关键字，也可以使用赋值语句进行初始化。数组一旦被创建，它的大小就不能改变。

例如：

```
// 将 a 初始化为大小为 10 的整型数组
a = new int[10];
// 将 b 初始化为大小为 4 的整型数组，并且 4 个分量的值分别等于 0，1，2，3
int[] b = {0,1,2,3};
```

数组的下标从 0 开始计数，到数组大小减 1 结束。在 Java 中不能越过数组下标的范

围去访问数组中的数据。例如：

```
a[10] = 10;
```

如果越过数组的下标访问数据，则会产生一个名为 ArrayIndexOutOfBoundsException 的运行时错误。这种错误可以通过在访问某个下标的数组元素前检查数组的大小来避免。数组的大小可以通过数组的变量 length 返回。例如：

```
for (int i=0;i<a.length;i++)
    a[i] = i;
```

由于在 Java 中数组实际上是一个类，因此两个数组变量可以指向同一个数组。请读者预测以下这段代码的运行结果。

```
int[] a = {1,1,1};
int[] b = a;
for (int i=0;i<b.length;i++)
    b[i]++;
for (int i=0;i<a.length;i++)
    System.out.print(a[i]);
```

在这段代码中对数组 b 的每个分量加 1，但是在输出数组 a 的每个分量时，可以发现 a 的每个分量都发生了变化，都为 2。其原因是赋值语句 int[] b = a; 只是将对于数组 a 的引用传递给变量 b，此时数组变量 a、b 实际上指向同一个数组空间。图 1-2 说明了这段代码运行时的情况。

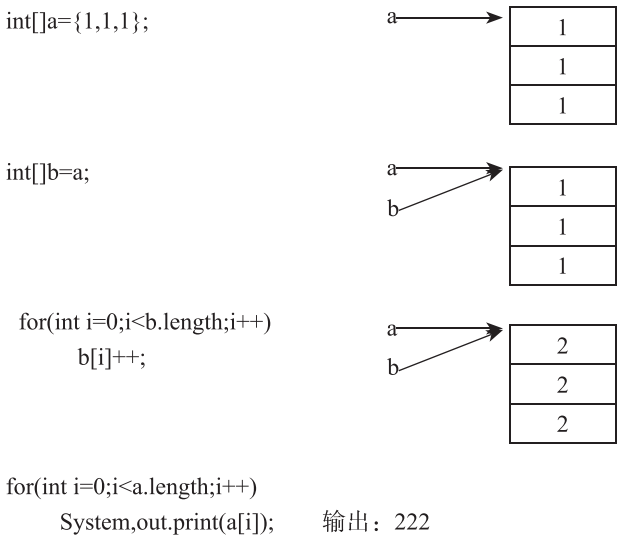


图 1-2　两个数组变量指向同一个数组空间

若要得到两个不同的但每个分量的值均相等的数组，可以使用循环语句或 System 类中的 arraycopy 方法来完成。

笔记

同样当数组作为方法的参数传递时，也是传递的对于数组的引用，因此在方法中对数组进行的操作会影响到原来的数组。例如：

```
public void changeArray(int[] a)
{
    for (int i=0;i<a.length;i++)
            a[i] = a[i] * 2;
}
```

那么如下代码的运行结果为 444。

```
int c = {2,2,2};
changeArray( c );
for (int i=0;i<c.length;i++)
    System.out.print(c[i]);
```

1.2 Java 的面向对象特性

面向对象程序设计（object-oriented programming，OOP）是目前主流的程序设计方法，它已经取代了以前基于过程的程序设计技术。面向对象程序设计主要是指在程序设计中采用抽象、封装、继承等设计方法。

1.2.1 类与对象

在面向对象思想中，抽象决定了对象的对外形象、内部结构以及处理对象的外部接口，其关键是处理对象的可见外部特征。抽象主要是从现实世界中抽象出合理的对象结构。

封装性是保证软件部件具有优良的模块性的基础。在 Java 中，最基本的封装单元是类，一个类定义了将由一组对象所共享的行为（数据和代码）。一个类的每个对象均包含它所定义的结构与行为，这些对象就像是一个模子铸造出来的。所以对象也叫作类的实例。

1. 类的定义

在定义一个类时，需要指定构成该类的代码与数据，类所定义的对象叫作成员变量。操作数据的代码叫作成员方法，方法定义怎样使用成员变量。这意味着类的行为和接口要由操作数据的方法来定义。

由于类的用途是封装复杂性，所以类的内部有隐藏实现复杂性的机制。所以 Java 中提供了私有和公有的访问模式，类的公有接口代表外部的用户应该知道或可以知道的每件东西。私有的方法和数据只能通过该类的成员代码来访问。

在 Java 中类的定义是通过关键字 class 来实现的。例如：

```
public class People{
    private String name;
    private String id;
    //Constructor
    public People(){
        this("","");
}

public People(String name,String _id){
        this.name = name;
        id = _id;
}

public void sayHello(){
        System.out.println("Hello! My name is " + name);
}
public void sayHello(String name){
        System.out.println("Hello,"+name+ "! My name is " +this.name);
}

//get & set methods
public void setName(String name){
        this.name = name;
}
public void setId(String id){
        this.id = id;
}
public String getName(){
        return this.name;
}
public String getId(){
        return this.id;
    }
}
```

笔记

代码中使用 class 关键字定义了一个名为 People 的类，class 前面的 public 关键字表示这个类似的一个公有类，可被访问。People 类中使用了 this 关键字，this 关键字主要有两个作用，一是表示对隐式参数的引用，二是调用类中的其他构造方法。

笔记

在类的内部首先通过 private 关键字定义了两个私有的成员变量，由于这两个成员变量是私有的，因此为了能够在类的外部获取或修改这些信息，定义了四个 get、set 方法。并且 People 类定义了两个构造方法，构造方法是一种特殊的方法，其作用是构造并初始化对象，在一个类中构造方法可以定义多个。要构造一个新的对象只需要在构造方法前使用 new 关键字就可以了。例如：

```
People jack = new People("Jack","0001");
```

此外在类中还定义了两个 sayHello 方法，以便实现与外界的互操作。

2. 使用现有类

Java 提供的大量的预定义类可供使用，同时程序中可能还会使用第三方提供的或者自己编写的属于其他包的类，使用这些类会给编写程序带来巨大的便利。

如果在某个类中需要使用其他包中的类，可以使用 import 关键字将需要使用的类在类的定义开始之前引入。

例如，在 People 类中还可以定义一个新的成员变量 birthday，而日期可以使用 Java 提供的预定义类 Calendar 实现。以下代码给出了实现方法。

```
import java.util.Calendar;
public class People {
    private String name;
    private String id;
    private Calendar birthday;
    // 构造方法
    ……
    //sayHello 方法、get & set 方法
    ……
}//end of class
```

1.2.2 继承

继承是子类自动获取父类的数据和方法的机制，这是类之间的一种关系。在定义和实现一个类的时候，可以在一个已经存在的类的基础之上来进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。

例如，学生也是人，那么他（她）也有 name、id、birthday 等属性，也可以和外界进行 sayHello 方法定义的互操作，但是学生是一群特定的人群，他们还具有一些特定的属性以及特定的和外界进行互操作的方法。在这种情况下就需要使用继承，可以定义一个新的类 Student，然后向它添加功能。但是新的类可以重用 People 类中已有的成员变量和方法。抽象地说，Student 类和 People 类是一个明显的“is-a”关系：每个学生都是人。“is-a”关系就是继承的特点。

在 Java 中使用 extends 关键字来实现继承。例如下面的代码定义了一个新的类 Student，它继承了最初定义的 People 类。

笔记

```
public class Student extends People{
private String sId; // 学号
//Constructor
public Student() {
    this("","","");
}
public Student(String name,String id,String sId){
    super(name,id);
    this.sId = sId;
}

public void sayHello(){
    super.sayHello();
    System.out.println("I am a student of department of computer science.");
}
//get & set method
public String getSId(){
    return this.sId;
}
public void setSId(String sId){
    this.sId = sId;
}
}
```

代码中使用了 super 关键字 ，super 关键字主要有两个作用，一是调用父类的构造方法，二是调用父类的方法。

extends 关键字表明使用它构造出来的类是从一个现有的类衍生出来的。现有类称为父类，而新的类称为子类。父类与子类相比并不具有更多的属性和功能，子类比父类具有更多的属性和功能。“is-a”规则表明子类的每个对象都是父类的对象，例如每个学生都是人。因此，无论何时只要在程序中需要一个父类对象时都可以使用一个子类的对象来替代它，反过来则不行。

例如，可以把子类的对象赋给父类变量：

```
People p = new Student("Bob","0002","2006137129");
```

如果想要把对某个类的对象引用转换为对另一个类的对象引用，需要用圆括号把目标

笔记

类名括起来，然后放到需要转换的对象引用之前。例如：

```
Student s = (Student)p;
```

当然这种转换并不是一定能够完成，如果是不能完成的情况，程序在运行时会抛出异常。为了使转换在允许的情况进行，可以使用 instanceof 关键字。例如：

```
if ( p instanceof Student)
    Student s = (Student)p;
```

在 Java 中有一个非常特殊的预定义类，那就是 Object 类。在 Java 中 Object 类是所有类的祖先，每个类都由 Object 类扩展而来。在定义类时如果不指定父类，则 Java 会自动把 Object 类作为要定义类的父类。例如 People 类就是 Object 类的子类。因此可以使用 Object 类的变量引用任意类型的对象。例如：

```
Object obj = new People("Jack","0001");
```

在 Java 中不支持多继承。Java 对于多继承大部分功能的实现是通过接口机制来完成的。

1.2.3 接口

接口是 Java 实现多继承的一种机制，一个类可以实现一个或多个接口。接口是一系列方法的声明，是一些方法特征的集合，一个接口只有方法的特征没有方法的实现，因此这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为。简单地说，接口不是类，但是定义了一组对类的要求，实现接口的某些类要与接口一致。

在 Java 中使用 interface 关键字来定义接口。例如：

```
public interface Compare {
    public int compare(Object otherObj);
}
```

Compare 接口定义了一种操作 compare，该操作应当完成与另一个对象进行比较的功能。它假定某个实现这一接口的类的对象 x 在调用该方法时，例如 x.compare(y)，如果 x 小于 y，则返回负数，相等返回 0，否则返回正数。

让类实现一个接口需要使用 implements 关键字，然后在类中实现接口所定义的方法。例如：

```
public class Student extends People implements Compare{
private String sId; // 学号
//Constructor
public Student() {
```

笔记

```
        this("","","");
    }
    public Student(String name,String id,String sId){
        super(name,id);
        this.sId = sId;
    }

    public void sayHello(){
        super.sayHello();
        System.out.println("I am a student of department of computer science.");
    }
    //get & set method
    public String getSId(){
        return this.sId;
    }
    public void setSId(String sId){
        this.sId = sId;
    }

    //implements Compare interface
    public int compare(Object otherObj){
        Student other = (Student)otherObj;
        return this.sId.compareTo(other.sId);
    }
    }//end of class
```

代码中 Student 类实现了 Compare 接口，并且实现了 compare 方法，这里假定通过两个学生学号的字典顺序完成对两个学生的比较，学号字典顺序在前的学生小于学号字典顺序在后的学生。

需要注意的是在 Java 中接口不是类，因此不能使用 new 实例化接口。但是虽然不能通过 new 构造接口对象，但是可以声明接口变量。并且只要类实现了接口，就可以在任何需要该接口的地方使用这个接口的对象。例如：

```
Compare com = new Student("Cary","0003","2006137101");
```

反之，也可以将一个接口变量转换为对某个类的对象的引用，不过此时要进行强制转换，并且不一定能够完成，情况和从父类引用到子类引用的转换是一样的。例如：

```
Student s = (Student)com;
```

笔记

1.3 异常

对于程序运行时碰到的异常情况，Java 使用了一种被称为“异常处理”的机制来进行处理。在 Java 中一个异常对象总是 Throwable 子类的实例。图 1-3 所示为 Java 异常继承层次结构。

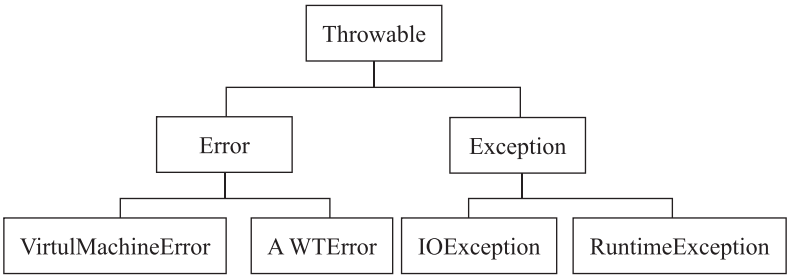


图 1-3　Java 异常继承层次结构

在 Java 程序设计中，常常关注 Exception 这个分支体系，而 Exception 中一类是从 RuntimeException 衍生出来的子类，以及不是从它衍生出来的其他异常类。一般来说，由编程导致的错误会引起 RuntimeException，例如数组下标越界、错误的类型转换、访问空指针等错误会导致不同类型的 RuntimeException。

在程序中可能会碰到任何标准异常都不能很好描述的异常情况。此时，可创建自己的异常类，创建自己的异常类只需要继承 Exception 类或 Exception 的子类就可以了。

在程序中如果碰到了异常的情况，可以有两种方法来处理这个异常，一种是由方法本身捕获这个异常并进行相应的处理，使用 try…catch 结构；另一种是将这个异常从方法中抛出，使用 throws 及 throw 关键字。例如：

```
public void method1(){
    try{
            //statement may cause exception
            ……
    }catch(ExceptionType e){
            //deal with exception
            ……
    }
}
```

再如：

```
Public void method2() throws ExceptionType {
    ……
    if (exception condition) throw instance of ExceptionType;
    ……
}
```

1.4 Java 与指针

笔记

尽管在 Java 中没有显示使用指针并且也不允许程序员使用指针，然而实际上对象的访问就是使用指针来实现的。一个对象会从实际存储空间的某个位置开始占据一定数量的存储体。该对象的指针就是一个保存了对象的存储地址的变量，并且这个存储地址就是对象在存储空间中的起始地址。在许多高级语言中指针是一种数据类型，而在 Java 中是用对象的引用来替代的。

考虑前面已经定义的 People 类，以及下列语句：

```
People p = null;
q = new People("Jack","0001");
```

这里创建了两个对于对象引用的变量 p 和 q 。变量 p 初始化为 null，null 是一个空指针，它不指向任何地方，也就是说它不指向任何类的对象，因此 null 可以赋值给任何类的对象的引用。变量 q 是一个对于 People 类的实例的引用，操作符 new 的作用实际上是为对象开辟足够的内存空间，而引用 p 是指向这一内存空间地址的指针。

为此请读者考虑如下代码的运行结果。

```
People p1 = new People("David","0004");
People p2 = p1;
p2.setName("Denny");
System.out.println(p1.getName());
```

这段代码中对 People 类的对象引用 p2 的 name 成员变量进行了设置，使其值为字符串“Denny”。但是很容易会发现在输出 p1 的成员变量 name 时并不是输出“David”，而是“Denny”。原因是 p1 与 p2 均是对对象的引用，在完成赋值语句“People p2 = p1;”后，p2 与 p1 指向同一存储空间，因此对于 p2 的修改自然会影响到 p1。图 1-4 清楚地说明了这段代码运行的情况。

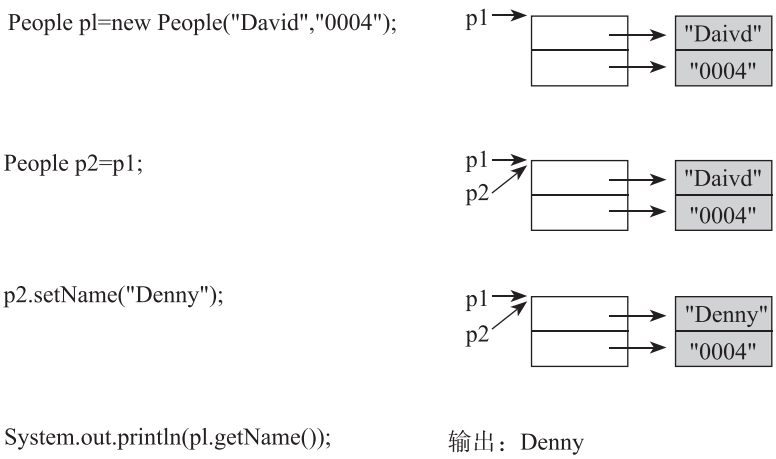


图 1-4　两个对象引用变量指向同一个存储空间

笔记

请读者继续考虑以下代码的运行结果。

```
People p1 = new People("David","0004");
People p2 = new People("David","0004");
System.out.println(p1= = p2);
```

在这里虽然 p1 与 p2 的所有成员变量的内容均相同，但是由于它们指向不同的存储空间，因此，输出语句输出的结果为 false。图 1-5 说明了 p1 与 p2 的指向。

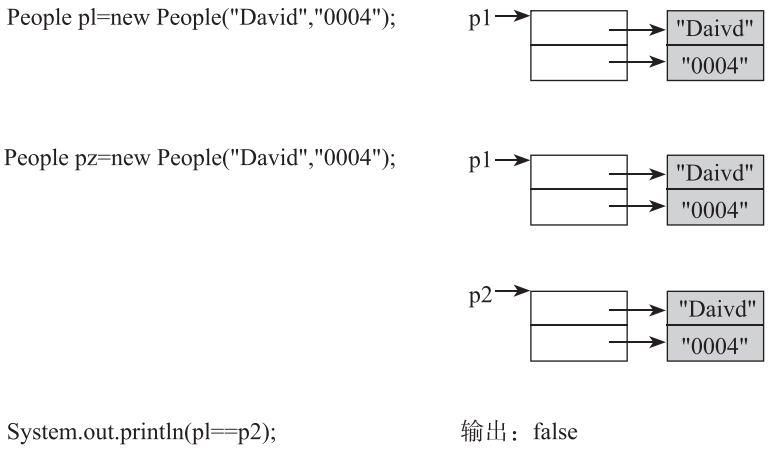


图 1-5　p1 与 p2 指向不同存储空间

可见如果希望完成对象的拷贝，使用一个简单的赋值语句是无法完成的。要达到这一目的可以通过实现 Cloneable 接口并重写 clone 方法来完成。如果希望判断两个对象引用是否一致，可以通过覆盖继承自 Object 类的 equals 方法来实现。

第 2 章 数据结构与算法基础

知识与技能目标

（1）了解数据结构的基本概念。

（2）掌握时间复杂性和空间复杂性的计算。

情感目标

（1）了解我国的算术历史，增强文化自信。

（2）牢记专业使命，牢记肩负专业的社会责任，努力实现中华民族伟大复兴。

知识导图

笔记

本章导读

这一章主要由两部分内容组成：数据结构和算法的基础知识。本章主要介绍数据结构与算法的一些相关基本概念，使读者了解什么是数据结构，数据结构研究的主要内容是什么；同时使读者了解什么是算法，以及如何评价一个算法的性能。

2.1 数据结构

思政园地：现代计算机的前身

人们在使用计算机解决客观世界中存在的具体问题时，过程通常如下：首先通过对客观世界的认知形成印象和概念从而得到信息，在此基础上建立概念模型，它必须能够如实地反映客观世界中的事物以及事物间的联系；然后根据概念模型将实际问题转化为计算机能够理解的形式，然后设计程序；最后用户通过人机交互界面与系统交流，使系统执行相应操作，最后解决实际的问题。

数据结构主要与在上述过程中从建立概念模型到实现模型转化并为后续程序设计提供基础的内容相关。它用来反映一个概念模型的内部构成，即一个概念模型由哪些成分数据构成，以什么方式构成，呈现什么结构。数据结构是一门主要研究程序设计问题中计算机的操作对象及它们之间的关系和操作的学科。

2.1.1 基本概念

在这一节中首先介绍一些基本概念和术语。

1. 数据

数据（data）是指描述客观事物的数值、字符以及能输入机器且能被处理的各种符号的集合。数据的含义非常广泛，除了通常的数值数据、字符、字符串是数据以外，声音、图像等一切可以输入计算机并能被处理的都是数据。例如，除了表示人的姓名、身高、体重等的字符、数字是数据外，人的照片、指纹、三维模型、语音指令等等也都是数据。

2. 数据元素

数据元素（data element）是数据的基本单位，是数据集合的个体，在计算机程序中通常作为一个整体来进行处理。例如，一条描述一位学生的完整信息的数据记录就是一个数据元素；空间中一点的三维坐标也可以是一个数据元素。数据元素通常由若干个数据项组成，例如描述学生相关信息的姓名、性别、学号等都是数据项；三维坐标中的每一维坐标值也是数据项。数据项具有原子性，是不可分割的最小单位。

3. 数据对象

数据对象（data object）是性质相同的数据元素的集合，是数据的子集。例如，一个学校的所有学生的集合就是数据对象，空间中所有点的集合也是数据对象。

4. 数据结构

数据结构（data structure）是指相互之间存在一种或多种特定关系的数据元素的集合。

它是组织并存储数据以便能够有效使用的一种专门格式，用来反映一个数据的内部构成，即一个数据由哪些成分数据构成，以什么方式构成什么结构。

笔记

由于信息可以存在于逻辑思维领域，也可以存在于计算机世界，因此作为信息载体的数据同样存在于两个世界中。表示一组数据元素及其相互关系的数据结构同样也有两种不同的表现形式，一种是数据结构的逻辑层面，即数据的逻辑结构，另一种是存在于计算机世界的物理层面，即数据的存储结构。

数据的逻辑结构按照数据元素之间相互关系的特性来分，可以分为 4 种结构：集合、线性结构、树形结构和图状结构。本书中讨论的数据结构主要有线性表、栈、队列、树和图，其中线性表、栈、队列属于线性结构，树和图属于非线性结构。

数据的逻辑结构可以采用两种方法来描述：二元组、图形。 数据结构的二元组表示形式为

$$数据结构 = \{D, S\}$$

其中，D 是数据元素的集合；S 是 D 中数据元素之间的关系集合，并且数据元素之间的关系是用序偶来表示的。序偶是由两个元素 x 和 y 按一定顺序排列而成的二元组，记作 <x, y>，x 是它的第一元素，y 是它的第二元素。

当使用图形来表示数据结构时，是用图形中的点来表示数据元素，用图形中的弧来表示数据元素之间的关系。如果数据元素 x 与 y 之间存在关系 <x, y>，则在图形中有从表示 x 的点出发到达表示 y 的点的一条弧。

例 2–1　一种数据结构的二元组表示为 set = (K,R)，其中

$$K = \{01, 02, 03, 04, 05\}$$

$$R = \{\}$$

可以看到在数据结构 set 中，只有数据元素的集合非空，而数据元素之间除了同属一个集合之外不存在任何关系（关系集合为空）。这表明该结构只考虑数据元素而不考虑它们之间的关系。就可以把具有这种特点的数据结构称为集合结构。

例 2–2　一种数据结构的二元组表示为 linearity = (K,R)，其中

$$K = \{01, 02, 03, 04, 05\}$$

$$R = \{<02,04>, <03,05>, <05,02>, <01,03>\}$$

可以看到在数据结构 linearity 中，数据元素之间是有序的。在这些数据元素中有一个可以称为“第一个”（元素 01）的数据元素；还有一个可以称为“最后一个”（元素 05）的数据元素；除第一个元素之外每个数据元素有且仅有一个直接前驱元素，除最后一个元素之外每个数据元素有且仅有一个直接后续元素。这种数据结构的特点是数据元素之间是 1 对 1 的联系，即线性关系，把具有此种特点的数据结构称为线性结构。

例 2–3　一种数据结构的二元组表示为 tree = (K,R)，其中

$$K = \{01, 02, 03, 04, 05, 06\}$$

$$R = \{<01,02>, <01,03>, <02,04>, <02,05>, <03,06>\}$$

可以看到在数据结构 tree 中，除了一个数据元素（元素 01）之外每个数据元素有且仅有一个直接前驱元素，但是可以有多个直接后续元素。这种数据结构的特点是数据元素之间是 1 对 N 的联系，把具有此种特点的数据结构称为树结构。

笔记

例 2-4　一种数据结构的二元组表示为 graph = (K,R)，其中

K = {01, 02, 03, 04, 05}

R = {<01,02>, <01,05>, <02,01>, <02,03>, <02,04>, <03,02>,
<04,02>, <04,05>, <05,01>, <05,04>}

可以看到在数据结构 graph 中，每个数据元素可以有多个直接前驱元素，也可以有多个直接后续元素。这种数据结构的特点是数据元素之间是 M 对 N 的联系，我们把具有此种特点的数据结构称为图结构。

数据结构 set、linearity、tree、graph 的图形表示方法如图 2-1 所示。

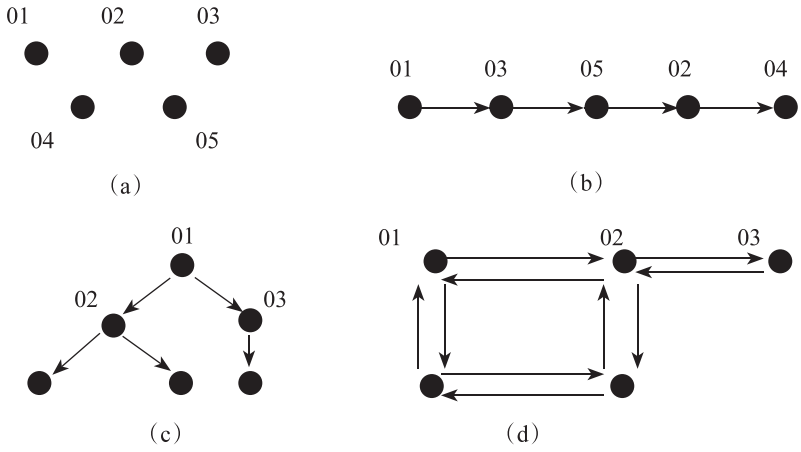


图 2-1　set、linearity、tree、graph 的图形表示方法

(a) set；(b) linearity；(a) tree；(a) graph

数据的存储结构主要包括数据元素本身的存储以及数据元素之间关系的表示。通过数据元素的定义可以看出，我们可以很容易地使用 Java 中的一个类来实现它，数据元素的数据项就是类的成员变量。

数据元素之间的关系在计算机中主要有两种不同的表示方法：顺序映像和非顺序映像，并由此得到两种不同的存储结构：顺序存储结构和链式存储结构。顺序存储结构的特点：数据元素的存储对应于一块连续的存储空间，数据元素之间的前驱和后续关系通过数据元素在存储器中的相对位置来反映。链式存储结构的特点：数据元素的存储对应的是不连续的存储空间，每个存储节点对应一个需要存储的数据元素。元素之间的逻辑关系通过存储节点之间的链接关系反映出来。

由于我们是在 Java 这种计算机高级程序设计语言的基础上来讨论数据结构，因此，我们在讨论数据的存储结构时不会在真正的物理地址的基础上去讨论顺序存储和链式存储，而是在 Java 语言提供的一维数组以及对象的引用的基础上去讨论和实现数据的存储结构。关于 Java 中的一维数组和对象的引用我们已经在第 1 章 1.1.4 节和 1.4 节中分别进行了介绍，在这里不再赘述。

2.1.2 抽象数据类型

抽象数据类型是描述数据结构的一种理论工具。在介绍抽象数据类型之前，我们先介

绍一下数据类型的基本概念。

笔记

数据类型（data type）是一组性质相同的数据元素的集合，以及加在这个集合上的一组操作。例如 Java 语言中就有许多不同的数据类型，包括数值型的数据类型、字符串、布尔型等数据类型。以 Java 中的 int 型为例，int 型的数据元素的集合是 [−2147483648,2147483647] 间的整数，定义在其上的操作有加、减、乘、除四则运算，还有模运算等。

定义数据类型有两个作用，一个是隐藏计算机硬件及其特性和差别，硬件对于用户而言是透明的，即用户可以不关心数据类型是怎么实现的便可以使用它。另一个作用是，用户能够使用数据类型定义的操作，方便地实现问题的求解。例如，用户可以使用 Java 定义在 int 型的加法操作完成两个整数的加法运算，而不用关心两个整数的加法在计算机中到底是如何实现的。这样不但加快了用户解决问题的速度，也使得用户可以在更高的层面上考虑问题。

与机器语言、汇编语言相比，高级语言的出现大大地简便了程序设计。但是要将解答问题的步骤从非形式的自然语言表达到形式化的高级语言表达，仍然是一个复杂的过程，仍然要做很多繁杂琐碎的事情，因此仍然需要抽象。对于一个明确的问题，要解答这个问题，总是先选用该问题的一个数据模型。接着，弄清该问题所选用的数据模型在已知条件下的初始状态和要求的结果状态，以及隐含着的两个状态之间的关系。然后探索从数据模型的已知初始状态出发到达要求的结果状态所必需的运算步骤。

我们在探索运算步骤时，首先应该考虑顶层的运算步骤，然后考虑底层的运算步骤。所谓顶层的运算步骤是指定义在数据模型级上的运算步骤，也称宏观运算。它们组成解答问题步骤的主干部分。其中涉及的数据是数据模型中的一个变量，暂时不关心它的数据结构；涉及的运算以数据模型中的数据变量作为运算对象，或作为运算结果，或两者兼而为之，简称定义在数据模型上的运算。由于暂时不关心变量的数据结构，这些运算都带有抽象性质，不含运算的细节。所谓底层的运算步骤是指顶层抽象的运算的具体实现。它既依赖于数据模型的结构，也依赖于数据模型结构的具体表示。因此，底层的运算步骤包括两部分：一是数据模型的具体表示，二是定义在该数据模型上的运算的具体实现。我们可以把它们理解为微观运算。于是，底层运算是顶层运算的细化，底层运算为顶层运算服务。为了将顶层算法与底层算法隔开，使两者在设计时不会互相牵制、互相影响，必须对两者的接口进行一次抽象。让底层只通过这个接口为顶层服务，顶层也只通过这个接口调用底层的运算。这个接口就是抽象数据类型。

抽象数据类型（abstract data type，ADT）由一种数据模型和在该数据模型上的一组操作组成。

抽象数据类型包括定义和实现两个方面，其中定义是独立于实现的。抽象数据类型的定义仅取决于它的逻辑特性，而与其在计算机内部的实现无关，即无论它的内部结构如何变化，只要它的逻辑特性不变，就不会影响到它的使用。其内部的变化（抽象数据类型实现的变化）可能会对外部在使用它解决问题时的效率上产生影响，因此我们的一个重要任务就是如何简单、高效地实现抽象数据类型。很明显，对于不同的运算组，为使组中所有运算的效率都尽可能高，其相应的数据模型具体表示的选择将是不同的。在这个意义下，

笔记

数据模型的具体表示又依赖于数据模型上定义的运算。特别是当不同运算的效率互相制约时，必须事先将所有运算的相应使用频度排序，让所选择的数据模型的具体表示优先保证使用频度较高的运算有较高的效率。

我们应该看到，抽象数据类型的概念并不是全新的概念。抽象数据类型和数据类型在实质上是一个概念，只不过是对数据类型的进一步抽象，不仅包括各种不同的计算机处理器中已经实现的数据类型，还包括为解决更复杂的问题而由用户自定义的复杂数据类型。例如，高级语言都有的“整数”类型就是一种抽象数据类型，只不过高级语言中的整型引进已经实现了，并且实现的细节可能不同而已。我们没有意识到抽象数据类型的概念已经孕育在基本数据类型的概念之中，是因为我们已经习惯于在程序设计中使用基本数据类型和相关的运算，没有进一步深究而已。

抽象数据类型一方面使得使用它的人可以只关心它的逻辑特征，不需要了解它的实现方式；另一方面可以使我们更容易描述现实世界，使得我们可以在更高的层面上来考虑问题。例如可以使用树来描述行政区划，使用图来描述通信网络。

根据抽象数据类型的概念，对抽象数据类型进行定义就是约定抽象数据类型的名字，同时，约定在该类型上定义的一组运算的各个运算的名字，明确各个运算分别要有多少个参数、这些参数的含义和顺序，以及运算的功能。一旦定义清楚，人们在使用时就可以像引用基本数据类型那样，十分简便地引用抽象数据类型；同时，抽象数据类型的实现就有了设计的依据和目标。抽象数据类型的使用和实现都与抽象数据类型的定义打交道，这样使用与实现没有直接的联系。因此，只要严格按照定义，抽象数据类型的使用和实现就可以互相独立，互不影响，就可以实现对它们的隔离，达到抽象的目的。

为此抽象数据类型可以使用一个三元组来表示。

$$ADT = (D, S, P)$$

其中，D 是数据对象；S 是 D 上的关系集；P 是加在 D 上的一组操作。

在定义抽象数据类型时，我们使用以下格式：

```
ADT 抽象数据类型名 {
    数据对象：<数据对象的定义>
    数据关系：<数据关系的定义>
    基本操作：<基本操作的定义>
}
```

小结

通过以上两节的内容我们可以看到数据结构就是研究 3 个方面的主要问题的：数据的逻辑结构、数据的存储结构以及定义在数据结构上的一组操作。即研究按照某种逻辑关系组织起来的一批数据，并按一定的映像方式把它们存放在计算机的存储器中，最后分析在这些数据上定义的一组操作。为此我们要考虑怎样合理地组织数据，建立合适的结构，提高实现的效率。

在数据结构的实现中我们可以很好地将数据结构中的一些基本概念和 Java 语言中的一

些概念对应起来。数据元素可以对应到类，其数据项就是类的成员变量，某个具体的数据元素就是某个类的一个实例；数据的顺序存储结构与链式存储结构可以通过一维数组以及对象的引用来实现；抽象数据类型也可以对应到类，抽象数据类型的数据对象与数据之间的关系可以通过类的成员变量来存储和表示，抽象数据类型的操作则使用类的方法来实现。

笔记

2.2　算法及性能分析

算法设计是最具创造性的工作之一，人们解决任何问题的思想、方法和步骤实际上都可以认为是算法。人们解决问题的方法有好有坏，因此算法在性能上也就有高低之分。在这一节中首先给出算法的定义，然后介绍分析算法性能的理论方法。

2.2.1　算法

算法（algorithm）是指令的集合，是为解决特定问题而规定的一系列操作。它是明确定义的可计算过程，以一个数据集合作为输入，并产生一个数据集合作为输出。一个算法通常来说具有以下 5 个特性：

（1）输入：一个算法应以待解决的问题的信息作为输入。

（2）输出：输入对应指令集处理后得到的信息。

（3）可行性：算法是可行的，即算法中的每一条指令都是可以实现的，均能在有限的时间内完成。

（4）有穷性：算法执行的指令个数是有限的，每个指令又是在有限时间内完成的，因此整个算法也是在有限时间内可以结束的。

（5）确定性：算法对于特定的合法输入，其对应的输出是唯一的，即当算法从一个特定输入开始，多次执行同一指令集的结果总是相同的。

对于随机算法，算法的第 5 个特性应当被放宽。在本书中所讨论的算法均是确定性算法。简单地说，算法就是计算机解题的过程。在这个过程中，无论是形成解题思路还是编写程序，都是在实施某种算法。前者是算法的逻辑形式，后者是算法的代码形式。

2.2.2　时间复杂性

在计算机资源中，最重要的就是时间与空间。评价一个算法性能的好坏，实际上就是评价算法的资源占用问题。在这一节中讨论算法运行时间的确定问题，这个问题称为算法的时间复杂性。关于算法的空间性能评价将在 2.2.3 中介绍。

下面以一个例子开始，通过它来说明如何分析算法的运行时间。

例 2–5　简单选择排序。

令 A[0,n–1] 为有 n 个数据元素的数组，我们的目标是将数组 A 排序为一个非降序的有序数组。使用简单选择排序来解决这个问题的算法：首先在 n 个元素中找到最小元素，将其放在 A[0] 中，然后在剩下的 n–1 个元素中找到最小的放到 A[1] 中，这个过程不断进行下去，直到在最后 2 个元素中找到小的，并将其放到 A[n–2] 中。

图 2-2 说明了对具有 7 个整数的数组简单选择排序的过程。

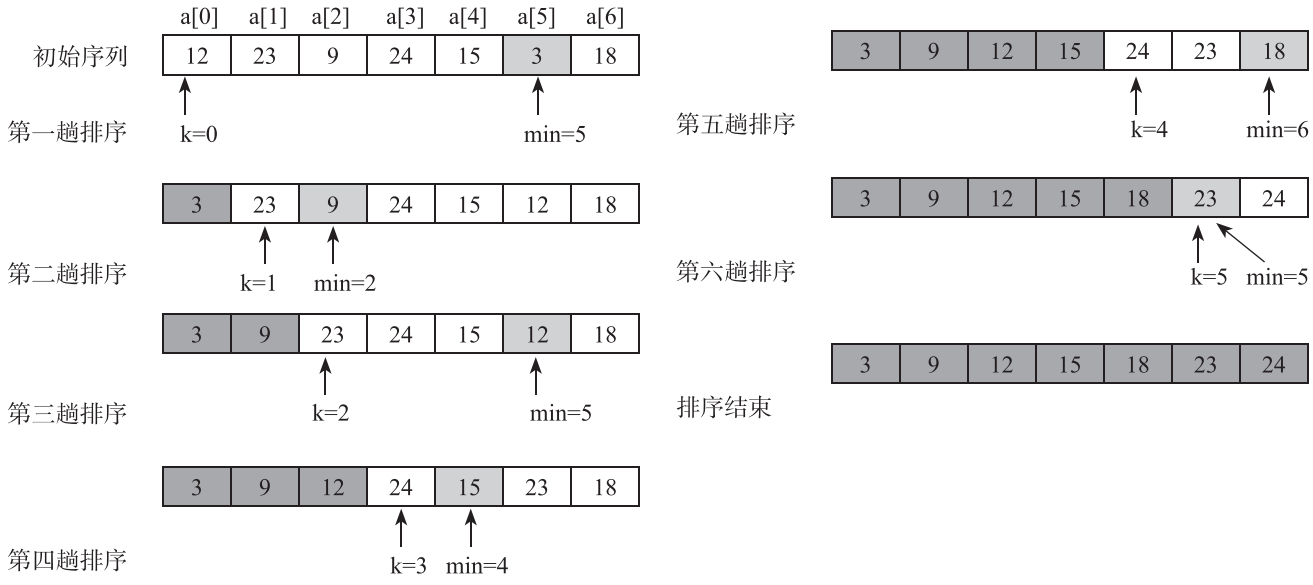


图 2-2　对 7 个整数的数组简单选择排序的过程

算法 2-1　selectSort

输入：整型数组 a[0,n-1]

输出：按非降序排列的数组 a[0,n-1]

代码：

```
public void selectSort (int[] a) {
    int n = a.length;
    for (int k=0;k<n-1; k++) {
      int min = k;
      for (int i=k+1; i<n; i++)
        if (a[i]<a[min])
          min = i;
      if (k!=min){
        int temp = a[k];
        a[k] = a[min];
        a[min] = temp;
      }//end of if
    }//end of for
  }
```

通过对算法的分析可以看出，这个算法执行的比较次数为

$$\sum_{i=1}^{n-1}(n-i)=(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

笔记

同时也可以看出数据元素交换的次数为 0 到 n–1，而每次交换需要使用 3 条赋值语句，因此数据元素的赋值为 0 到 3(n–1)。

首先，我们说一个算法对于某个输入要用 x 秒运行是没有意义的，这是因为影响实际运行时间的因素不仅有算法本身，还有其他诸多因素。例如，算法是在什么机器上运行的，不同的机器其运算速度是不一样的；除了硬件的影响，操作系统以及使用的高级语言、编译系统都会对算法的实际运行时间造成影响。因此，在对算法的运行时间做出分析时我们应该避免这些因素的影响，为此我们可以假设一些基本操作都是可以在一个常数时间内完成的。例如，逻辑运算、赋值运算等都是基本操作。这样算法执行基本操作的次数可以反映算法的运行时间，在后面提到算法的运行时间时都是指运行基本操作的次数。

其次，即使能够排除软硬件的影响，对同一个算法而言，如果问题的规模不同，那么实际的运算时间也会有很大差异。例如，在算法 2-1 中，如果算法分别对 100 个整数排序以及对 109 个整数排序，其实际运行时间的差异是非常大的。假设算法中进行一次比较需要 10^{-8} 秒，那么对 100 个数进行排序大概需要 $100 \times 99/2 \times 10^{-8}=0.000\ 049\ 5$（秒），而对 10^9 个数排序则需要 $10^9 \times (10^9-1)/2 \times 10^{-8}=158.5$（年）。为此在分析算法的运行时间时我们必须将问题的规模（通常用 n 来表示）也考虑进去。显然算法执行基本操作的次数是关于规模 n 的非负函数，我们把它记为 T(n)。下面给出了一些常用的作为问题规模的例子。

（1）在排序和查找问题中，用数组或表中元素的个数。

（2）在图的相关问题中，用图中的点或（和）边的数目。

（3）在计算几何问题中，用顶点、边、线段或多边形的数目。

（4）在矩阵运算中，用矩阵的维数。

（5）在数论问题中，通常用表示输入数的比特数来表示输入值的大小。

最后，在分析一个算法在输入一个规模为 n 的实例时，我们是否需要将 T(n) 的精确值求出来呢？事实上我们在评估一个算法时并不需要精确计算 T(n)。因为这是没有必要并且有时也是不可能的。算法的性能好坏是通过与其他算法的比较而得出的，由于算法对小规模输入实例所需的处理时间很少，因此小规模输入实例对性能的比较没有多大意义。我们关注的是在问题规模很大时算法的效率差异，而当问题的规模 n 不断扩大时，在函数 T(n) 中有一些部分就会变得不重要。例如在算法 2-1 中当 n 不断变大时，算法执行的所有赋值语句对整个运行时间的影响就越来越小。为此我们可以将这些不重要的部分忽略，转而讨论运行时间的增长率或增长的阶。一旦去掉表示算法运行时间中的低阶项和首项常数，就称我们是在度量算法的渐进时间复杂度（asymptotic complexity），简称时间复杂度。

为了进一步说明算法的时间复杂度，我们定义 O、Ω、Θ 符号。

1. O 符号

对算法 2-1 的 T(n) 进行分析，我们可以得到

$$T(n) \leqslant \frac{n(n-1)}{2}+3(n-1) \leqslant cn^2 (c\text{为某个正常数})$$

这时我们说算法 2-1 的时间复杂度为 $O(n^2)$，这个符号可以解释为只要当排序元素的个数大于某个阈值 N，那么对于某个常量 $c > 0$，运行时间最多为 cn^2。也就是说，O 符

笔记

号提供了一个运行时间上界。

定义 2-1 令 T(n) 和 f(n) 是非负函数，如果存在一个非负整数 N 以及一个常数 c>0，使得：

$$\forall n \geqslant N, T(n) \leqslant cf(n)$$

则 $T(n) = O(f(n))$。

即如果$\lim\limits_{x\to\infty}\dfrac{T(n)}{f(n)}$存在，那么

$$\lim_{x\to\infty}\frac{T(n)}{f(n)} \neq \infty \Rightarrow T(n)=O(f(n))$$

例如，算法 2-1 的时间复杂度$T(n)\leqslant\dfrac{n(n-1)}{2}+3(n-1)$，由于当 $n \geqslant 2$ 时，$T(n) \leqslant 2n^2$，则有 $T(n)=O(n^2)$，或令 $f(n)=n^2$，因为$\lim\limits_{x\to\infty}\dfrac{T(n)}{f(n)}\leqslant\lim\limits_{x\to\infty}\dfrac{n(n-1)/2+3(n-1)}{n^2}=\dfrac{1}{2}\neq\infty$，因此 $T(n)=O(n^2)$。

2. Ω 符号

O 符号给出了算法时间复杂度的上界，而 Ω 符号在运行时间的常数因子范围内给出了时间复杂度的下界。

Ω 符号可以解释为：如果输入大于或等于某个阈值 N，算法的运行时间下限是 f(n) 的 c 倍，其中 c 是一个正常数，则称算法的时间复杂度是 Ω(f(n)) 的。Ω 的形式定义与 O 符号对称。

定义 2-2 令 T(n) 和 f(n) 是非负函数，如果存在一个非负整数 N 以及一个常数 c>0，使得

$$\forall n \geqslant N, T(n) \geqslant cf(n)$$

则 $T(n) = \Omega(f(n))$。

即如果$\lim\limits_{n\to\infty}\dfrac{T(n)}{f(n)}$存在，那么

$$\lim_{n\to\infty}\frac{T(n)}{f(n)} \neq 0 \Rightarrow T(n)=\Omega(f(n))$$

例如，算法 2-1 的时间复杂度$T(n)\geqslant\dfrac{n(n-1)}{2}$，当 $n \geqslant 2$ 时，$T(n)\geqslant\dfrac{1}{4}n^2$，则有 $T(n)=\Omega(n^2)$，或令 $f(n)=n^2$，因为$\lim\limits_{n\to\infty}\dfrac{T(n)}{f(n)}\geqslant\lim\limits_{n\to\infty}\dfrac{n(n-1)/2}{n^2}=\dfrac{1}{2}\neq 0$，因此有 $T(n)=\Omega(n^2)$。

3. Θ 符号

O 符号给出了算法时间复杂度的上界，Ω 符号给出了时间复杂度的下界，而 Θ 给出了算法时间复杂度的精确阶。

Θ 符号可以解释为：如果输入大于或等于某个阈值 N，算法的运行时间在下限 $c_1f(n)$ 和 $c_2f(n)$ 之间 $(0 \leqslant c_1 \leqslant c_2)$，则称算法的时间复杂度是 Θ[f(n)] 阶的。该符号的形式定义如下。

定义 2-3 令 T(n) 和 f(n) 是非负函数，如果存在一个非负整数 N 以及常数 $c_1>0$ 和 $c_2>0$，使得

$$\forall n \geqslant N, c_1f(n) \leqslant T(n) \leqslant c_2f(n)$$

则 $T(n)=\Theta[f(n)]$。

即如果$\lim\limits_{n\to\infty}\dfrac{T(n)}{f(n)}$存在，那么

$$\lim_{n\to\infty}\frac{T(n)}{f(n)}=c(c>0)\Rightarrow T(n)=\Theta(f(n))$$

例如，算法 2-1 的时间复杂度 $\dfrac{n(n-1)}{2}\leqslant T(n)\leqslant\dfrac{n(n-1)}{2}+3(n-1)$，当 $n\geqslant 2$ 时，$\dfrac{1}{4}n^2\leqslant T(n)\leqslant 2n^2$，则有 $T(n)=\Theta(n^2)$。或令 $f(n)=n^2$，因为$\lim\limits_{n\to\infty}\dfrac{T(n)}{f(n)}=\dfrac{1}{2}$，因此有 $T(n)=\Theta(n^2)$。

定义 2-3 的一个重要推论是 $T(n)=\Theta(n^2)$，当且仅当 $T(n)=O(n^2)$ 并且 $T(n)=\Omega(n^2)$ 时，通过以上的分析可以看出：我们评价算法的运行时间是通过分析在一定规模下算法执行基本操作的次数来反映的，并且由于我们只对大规模问题的运行时间感兴趣，所以是使用算法的渐进时间复杂度 $T(n)$ 来度量算法的时间性能的。O、Ω、Θ 符号分别定义了时间复杂度的上界、下界以及精确阶。

2.2.3 空间复杂性

在 2.2.2 节中我们讨论了算法的时间复杂性，下面来讨论算法的空间复杂性。算法的空间复杂性同样是由算法运行时使用的空间来评价的。我们把算法使用的空间定义为：为了求解问题的实例而执行的操作所需要的存储空间的数目，但是它不包括用来存储输入实例的空间。同样前面对于评价算法时间复杂性的讨论都可以用于对算法的空间复杂性的讨论。并且在这里有这样一个观察结论：算法的空间复杂性是以时间复杂性为上界的。这是因为在算法中每访问一次存储空间都是需要使用一定时间的，即使每次访问的都是不同的存储空间，空间的大小也不会超过基本操作次数常数倍，因此算法的空间复杂性是以时间复杂性为上界的。

如果使用 $S(n)$ 与 $T(n)$ 分别表示算法的空间复杂度和时间复杂度，则有 $S(n)=O[T(n)]$。例如在算法 2-1 中，为了算法的执行，我们使用了常数个中间变量，每个变量的存储空间都是常数大小，所以在算法 2-1 中：$S(n)=O(1)=\Omega(1)=\Theta(1)$。

2.2.4 算法时间复杂度分析

我们不但要了解什么是算法时间复杂度，还要学会分析算法时间复杂度。最简单的方法就是将算法执行的所有基本操作都计算出来，然后得出算法时间复杂度。但是很多时候这种方法是不可取的，因为它太麻烦而且可能计算不出所有基本操作的执行次数。

一般来说，不存在固定的方法，使用它就可以得到一个算法时间复杂度。但是在分析算法时间复杂度时有一些常用技术是可以使用的。

1. 计算循环次数

运行时间往往都集中在循环中，循环之外往往是一些简单的具有一定数量基本操作的运算，而这些基本操作在渐进时间复杂度中是不起作用的。这就使得运行时间往往和循环的次数具有相同的阶。

笔记

下面给出几个使用这种方法分析算法时间复杂度的例子。

算法 2-2 function1

输入：正整数 n

输出：循环执行的总次数

代码：

```
public int function1 (int n) {
    int i = 1, count = 0;
    while (i <= n){
      i = i * 2;
      count++; }
    return count;
  }
```

例 2-6 分析上面的算法 function1 的时间复杂度。在这里只有一层循环，假设循环 k 次，循环每进行一次会执行常数个基本操作，因此算法的时间复杂度 $T(n) = \Theta(k)$。因为循环只执行了 k 次，i 在循环执行过程中的变化趋势为 $1,2,4,8,\cdots,2^k$，在执行完第 k 次循环后 $i=2^k$，所以有

$$2^{k-1} \leqslant n \leqslant 2^k \Rightarrow k=\lfloor \log n \rfloor +1$$

由此可知 $T(n)=\Theta(k)=\Theta(\lfloor \log n \rfloor +1)=\Theta(\log n)$。

算法 2-3 function2

输入：正整数 n

输出：循环执行的总次数

代码：

```
public int function2 (int n) {
    int count = 0, s = 0;
    while (s < n) {
      count++;
      s = s + count;
    }
    return count;
  }
```

例 2-7 分析上面的算法 function2 的时间复杂度。这个算法与算法 function1 类似，算法的时间复杂度与循环次数具有相同的阶。假设 while 循环执行了 k 次，而 count 在 while 循环执行过程中的变化趋势为 0,1,2,⋯,k，在执行完第 k 次循环后 i=k。而 s 在第 i(0<i<k+1) 次循环后的值为

$$s=0+1+2+\cdots+i=\frac{i(i+1)}{2}$$

因为循环只执行了 k 次，所以有

$$\frac{k(k-1)}{2}<n\leqslant\frac{k(k+1)}{2}\Rightarrow k=\Theta(n^{\frac{1}{2}})$$

因此，$T(n)=\Theta(k)=\Theta(n^{\frac{1}{2}})$。

算法 2–4　function3

输入：正整数 n

输出：循环执行的总次数

代码：

```
public int function3 (int n) {
    int i = 1, count = 0;
    while (i <= n){
        for (int j = 0; j < i; j++)
            count++;
        i = i * 2;
    }
    return count;
}
```

例 2–8　分析上面的算法 function3 的时间复杂度。假设 while 循环执行了 k 次，for 循环每次执行 i 次，而 i 在 while 循环执行过程中的变化趋势为 1,2,4,8,⋯,2^k，在执行完第 k 次循环后 $i=2^k$。所以，循环的总执行次数为

$$1+2+4+8+\cdots+2^{k-1}=\sum_{i=0}^{k-1}2^i=2^k-1$$

并且由例 2-6 知，$k=\lfloor\log n\rfloor+1$，因此，$T(n)=\Theta(2k-1)=\Theta(2^{\lfloor\log n\rfloor+1})=\Theta(n)$。

2. 分析最高频度的基本操作

在某些算法中，使用统计循环次数的方法来完成算法时间复杂度的估算是非常复杂的，有时甚至是无法完成的。

请读者先阅读下面的例子。

下面的算法是将两个已经有序的数组 a[0,m−1]、b[0,n−1] 合并为一个更大的有序数组。合并两个数组的算法是使两个变量 pa，pb 初始时分别指向数组 a、b 的第一个元素，每次比较 a[pa] 与 b[pb]，将小的放入新的数组 c。更新指向小元素的变量，使之加 1，指向后续元素。这个过程一直进行下去，直到将某个数组中的所有元素放入新的数组为止。此时再将另一个数组中剩余的数据元素依次放入新数组。这一过程在算法 2-5 中给出。

算法 2–5　merge

输入：整型非递减数组 a[0,m−1]、b[0,n−1]

笔记

输出：将数组 a 与 b 合并为一个新的非递减数组 c

代码：

```
public int[] merge (int[] a, int[] b) {
    int pa = pb = pc = 0;
    int m = a.length;
    int n = b.length;
    int[] c = new int[m+n];
    while (pa<m && pb<n) {
      if (a[pa]<b[pb])
        c[pc++] = a[pa++];
      else
        c[pc++] = b[pb++];
    }
    if (pa<m)
      while (pa<m)
        c[pc++] = a[pa++];
    else
      while (pb<n)
        c[pc++] = b[pb++];
    return c;
  }
```

例 2-9　分析算法 2-5 的时间复杂度。如果使用分析循环次数的方法来求算法的时间复杂度，在这个例子中会较为复杂，因为这里有 3 个循环，而每个循环执行的次数会由于输入实例的不同而不同。

在这里我们可以使用分析最高频度的基本操作方法来得到算法 merge 的时间复杂度。我们可以观察到在算法中主要操作有元素的赋值和比较两种操作，而赋值是使用频度最高的基本操作，因为每个元素都必须放入新的数组，而并不是每个元素都需要比较才能放入新的数组。在算法中的每个元素至少赋值一次，至多也赋值一次，因此共有 m+n 次赋值，因此算法 merge 的时间复杂度 $T(n) = \Theta(m+n)$。

在这一节中，我们主要介绍了两种确定算法时间复杂度的基本方法，使用递推公式求解的方法将在第 5 章中介绍。

2.2.5 最佳、最坏与平均情况分析

在上面的例子中，所有算法对于任何输入实例，它的时间复杂度都是相同的。但是有些算法的执行时间不但是规模的函数，也是输入实例数据元素初始顺序的实例。对于不同的输入实例，时间复杂度会有很大的差异，为此我们要对不同的情况进行不同分析。

笔记

下面我们介绍一个非常简单的例子用来说明这种情况。

例如：我们需要查找存在于一个数组中的某个元素，最简单的方法就是从左到右依次扫描数组中的每个数据元素，如果发现当前元素和需要查找的数据元素相同，则返回元素在数组中的下标。

算法 2–6　linearSearch

输入：整型数组 a[0,n–1]，整数 k = a[i]。$0 \leqslant i < n$

输出：i

代码：

```
public int linearSearch (int[] a, int k) {
    for (int i=0; i<n; i++)
        if (a[i]== k)
            return i;
    return  –1;
  }
```

这段代码非常简单，但是我们却不能武断地说这个算法的时间复杂度就是多少，因为这个算法会因为输入实例的不同，其执行时间会有很大的差异。对此我们做以下分析：

（1）如果输入的实例中要寻找的 a[0] 是 k，那么算法执行一次比较操作就会结束返回，此时执行的基本操作的个数为 $\Theta(1)$。这是最好的情况。

（2）如果输入的实例中要寻找的 a[n–1] 是 k，那么算法需要执行 n 次比较操作才能找到 k，然后结束返回，此时执行的基本操作的个数为 $\Theta(n)$。这是最坏的情况。

（3）第三种是平均情况，这里的平均情况是指执行所有大小为 n 的输入时算法的平均执行时间。

这 3 种情况分别对应了时间复杂性的最佳、最坏以及平均情况分析。

以最佳与最坏的情况分析，分别对应的是我们在所有大小为 n 的输入中选择代价最小的和最大的。对于最佳和最坏情况的分析较简单，例如上面我们对算法 linearSearch 的第一种和第二种情况的分析。

在平均情况的分析中，我们首先必须知道所有大小为 n 的输入的分布，即要知道每一种情况出现的概率。即使这样，在许多情况下对于算法的平均情况分析也是复杂的。下面我们分析一下算法 linearSearch 执行的平均情况。

例 2–10　算法 linearSearch 的平均情况分析。为了简化讨论，我们假设 a[0,n–1] 中元素互不相同。还假定 k 出现在数组中，并且最重要的是，我们假设数组中的任何一个元素出现在数组中任何一个位置上是等概率的，即

$$\forall x \in a, P[x = a[j]] = \frac{1}{n} \quad (0 \leqslant j < n)$$

假设 C_j 是当 k 出现在数组下标为 j 的地方时需要比较的次数，那么 $C_j = j + 1, 0 \leqslant j < n$。此时，为了找到 k 的位置，算法执行比较次数的平均值是

笔记

$$T(n)=\sum_{j=0}^{n-1}\left(C_j \cdot P[k=a[j]]\right)=\frac{1}{n}\sum_{j=0}^{n-1}(j+1)=\frac{1}{n}\frac{n(n+1)}{2}=\frac{n+1}{2}=\Theta(n)$$

这说明在平均情况下，为找到 k 的位置 n，我们大概要比较数组中一半的数据元素。因此算法 linearSearch 的时间复杂度在平均情况下是 $\Theta(n)$。

由此可见，除了前面我们提到的影响因素外，输入实例本身也会对算法时间复杂度的分析造成影响。

2.2.6 均摊分析

均摊分析是一种非常重要的时间复杂度分析方法，使用这种分析方法经常会给我们对算法时间复杂度的分析带来意想不到的效果，能够让我们为算法找到更小的时间复杂度上界，而这个上界通常会比我们意想中的上界小得多。

如果在一个算法中反复出现具有以下这样特性的计算时，我们就需要考虑使用均摊分析了。这种特性是：计算的运行时间始终变动，并且这一计算在大多数时候运行得很快，而在少数时候却要花费大量时间。

在均摊分析中，我们可以算出算法在整个执行过程中，或多次执行过程中所用时间的平均值，称为该算法的均摊运行时间。均摊分析保证了算法的平均代价，这与平均情况分析是不同的，在平均分析中必须知道每种输入实例的分布，计算所有不同输入实例才能得到平均值，而在均摊分析中不需要这样。

下面我们举两个例子来说明均摊分析的要点以及进行均摊分析的方法。

例 2-11　数组的大小一旦确定，那么在以后的使用过程中就不能再改变。当要给未知个数的数据元素分配存储空间时，可以采用以下策略：先分配一个初始大小为 m 的数组 A0，当 A0 空间用完之后，在第 m+1 个数据元素进入之前，分配一个大小为 2m 的新的数组 A1，并将 A0 中的元素全部移到 A1 中。如果 A1 空间用完，则再次新建大小为原来空间 2 倍的数组，移动元素到新数组。如此往复，直到所有元素都存入数组为止。

在这里假设 m=1，如果 m>1，则执行赋值的次数要多于从 1 开始的情况，因为在数组大小被扩展到 m 之前是没有元素的移动过程的。因此 m=1 时的运行时间是最长的，它是当 m 取其他值时运行时间的上界。

在算法中使用频度最高的是赋值操作，因此我们对元素赋值的次数进行计数，结果可以反映算法的时间复杂度。

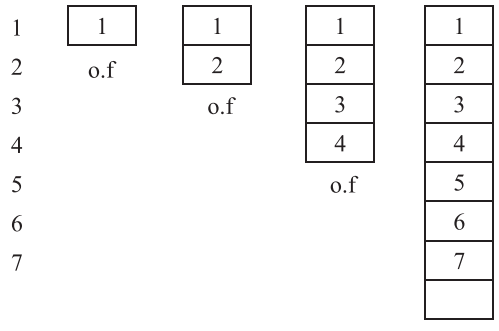


图 2-3 数组倍增过程

注：o.f 指溢出。

假设加入的数据元素的个数为 n。则每个元素最多被移动 n 次，所以 n 个元素加入数组总的时间 $T(n) = O(n^2)$ 。这是加入 n 个元素运行时间的上界，但是这个上界不够紧密，我们可以通过均摊分析得到更紧密的上界。分析如下。

当 m=1 时，算法的执行情况可用图 2-3 表示。

为计算加入 n 个元素需要元素赋值的次数，先定义变量 C_i，其值为加入第 i 个元素时元素赋值的次数。通过图 2-3 可以得出

$$\begin{cases} i，i\text{-}1 \text{ 为 } 2 \text{ 的整数次幂} \\ 1，其他 \end{cases}$$

表 2-1 反映了图 2-3 所示过程中数组大小及 C_i 变化的情况。得到

$$\sum_{j=1}^{n} C_j = n + \sum_{i=0}^{\log(n-1)} 2^i \leqslant 3n$$

因此 n 个元素加入数组总的时间 $T(n) = \Theta(n) << O(n^2)$。将这个时间均摊到每个元素，则存储和移动每个元素代价是 $\Theta(1)$ 均摊时间。

表 2-1　C_i 随 i 变化的情况

i	1	2	3	4	5	6	7	8	9	10
size	1	2	4	4	8	8	8	8	16	16
C_i	1	2	3	1	5	1	1	1	9	1
C_i/i	1	1/1	1/2	1	1/4	1	1	1	1/8	1

有时在对一个算法进行均摊分析时，不能像上面那样求出每次计算的时间，因此无法使用求和的方法来得出 n 次计算的时间总和，然后均摊到每次计算上。这时可以使用资源预留的方法进行分析。

例 2-12　考虑下面的算法。有一个有 n 个正整数的数组 a[0, n−1] 作为输入，同时生成一个大小与 a 相同的数组 array，然后依次处理 a 中每个元素：如果当前的 a[i] 是奇数则直接添加到 array 中最后一个元素后面；如果是偶数，则从 array 中最后一个元素开始，向前依次删除所有的奇数。数组 a 元素处理过程可以通过图 2-4 说明。

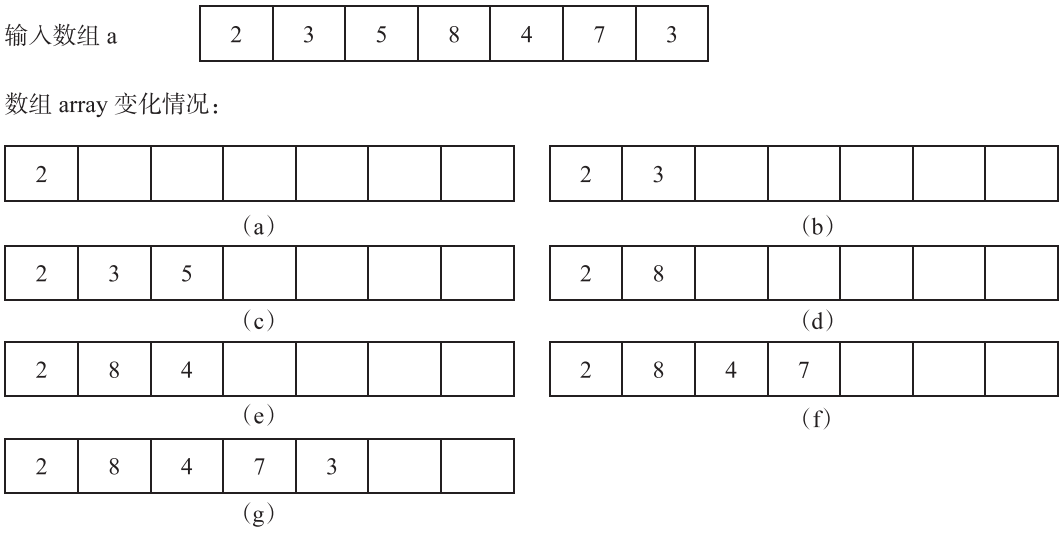


图 2-4　数组 a 元素处理过程

笔记

代码：

```
public void function4 (int[ ] a ) {
    int p = 0, n = a.length;
    int[ ] array = new int[n];
    for (int i=0; i<n; i++){
      if (a[i]%2 == 0)  // 如果是偶数
        while (p>0 && array[p-1]%2!=0)
          array[p--] = 0;  // 删除前面的奇数
      array[p++] = a[i];
    }
    return ;
  }
```

现在来分析一下算法的时间复杂度。在某些情况下，例如 n/2 个奇数后面跟着一个偶数，那么 while 循环要执行 O(n) 次，这里 for 循环要执行 n 次，因此总运行时间是 $O(n^2)$。同样如果我们使用均摊分析，可以得出复杂性为 Θ(n)。

在此算法中共有这样一些基本操作：添加、删除元素。但是在每一次 for 循环执行过程中不知道到底删除了多少个数据元素，因此无法如同例 2-11 那样计算出 C_i，也就无法直接求出 n 次计算执行基本操作的总次数。此时我们可以采用资源预留的分析方法，在进行每一次计算时，假设我们都为该次计算预留一定的时间。假设第 i 次计算执行 C_i' 次基本操作，只要保证不等式 $\sum_{i=1}^{n} C_i \leqslant \sum_{i=1}^{n} C_i'$ 成立，那么 $\sum_{i=1}^{n} C_i'$ 就是算法时间上界。

在本例中可以设 $C_i'=2$，其中一次用于元素的添加操作，另一次用于元素的删除操作，由于偶数只执行添加操作，而每个奇数最多进行一次添加和一次删除操作，因此 $\sum_{i=1}^{n} C_i \leqslant \sum_{i=1}^{n} C_i'=2n$，$T(n) = \Theta(n)$。把这个时间均摊到每个元素上，则每个元素的操作时间为 Θ(1)，即 while 语句的均摊时间是 Θ(1)。