第5章 java.util和java.time包

java.util 包，其中“util”是“utility”(实用工具)的缩写。java.util 包提供了很多实用类，例如 Random 类，用于生成伪随机数；DoubleSummaryStatistics 类，能轻松对双精度浮点型数据进行统计汇总；Arrays 类，为数组操作提供了丰富、便捷的方法，像排序、查找等。尤为重要的是，java.util 包还囊括了多种与数据结构紧密相关的集合类，也称集合框架，该集合框架包含了 ArrayList、LinkedList、HashSet、HashMap 等诸多类。这些集合类各具特性，开发人员能够依据具体需求灵活选用恰当的数据结构。例如，当需要频繁随机访问元素时，ArrayList 是不错之选；若侧重频繁插入和删除操作，LinkedList 更为合适；要高效存储和检索唯一元素，HashSet 可胜任；HashMap 则适用于键值对形式数据的快速查找场景。集合框架极大地方便了数据的存储、操作与管理，显著提升了数据处理效率，让代码的可读性、可维护性以及可扩展性都得到全方位提升。

大部分教材对 java.util 包中的 Arrays 类、集合框架都有所涉及，本章将给出一些在算法上具有实用价值和难度的实例，另外补充了一些教材未涉及的实用类。Java.time 包提供了处理时间的类，大部分教材都有所讲授，这里再补充几个处理时间的实用类(如果想更加系统和深入地学习 Java 集合框架的内容，可参见编者在清华大学出版社出版的《数据结构与算法(Java 语言版)》)。

5.1 Random 类

算法生成的随机数称作伪随机数，java.util 包中的 Random 类可以用来获取伪随机数或生成伪随机数序列。伪随机数看似随机，实际上是由一个确定的算法按照初始种子生成的。若使用相同的种子，每次运行程序时生成的随机数或随机序列也会相同。在生活中，抛扔硬币看正反面获得的值称作真随机值，真随机值和算法无关，仅和相应的硬件设备或实际操作有关(体育比赛，例如足球，裁判通过抛扔硬币看正反面决定哪个球队先开球，而伪随机数可能不被双方球队认可)。

Random 类的常用方法如下。

(1) Random()：构建一个新的随机数生成器，其种子基于当前系统时间的毫秒数。这里的毫秒数是从“纪元”(Epoch)开始到当前时刻所走过的毫秒数。“纪元”是一个固定的起始时间点，这个起始时间点被定义为 1970 年 1 月 1 日午夜(UTC)。由于系统时间在不断变化，所以每次使用 Random()构造函数创建的 Random 对象通常会产生不同的随机数序列。不过，如果在极短的时间间隔内两次创建 Random 对象，可能会因为系统时间还未发生明显变化而使用了相同的种子，进而生成相同的随机数序列。

(2) Random(long seed)：使用指定的种子创建一个新的随机数生成器，只要种子相同，生成器就会产生相同的随机数序列。

(3) int nextInt()：返回一个随机的 int 类型整数。

(4) int nextInt(int bound)：返回一个介于 0(包含)至指定边界 bound(不包含)的随机 int 类型整数。

(5) double nextDouble()：返回一个介于 0.0(包含)至 1.0(不包含)的随机 double 类型小数。

(6) boolean nextBoolean()：返回一个随机的 boolean 类型值，结果为 true 或者 false。

5.1.1 蒙特卡罗算法与圆周率

蒙特卡罗算法借助伪随机数计算圆周率的近似值。在一个边长为 1 的正方形内，随机生成大量的点，这些点均匀分布在正方形内，称这些点的数量为总点数。同时以正方形的一个顶点为圆心(这里以坐标原点)、1 为半径，绘制一个四分之一圆。根据几何概率，落在四分之一圆内点的数量与总点数的比值近似等于四分之一圆的面积与正方形面积的比值，即设定生成点的总数为 n，落在四分之一圆内点的数量为 m，那么 $m/n(n\to\infty)$ 的极限是 $\pi/4$，如图 5.1 所示。

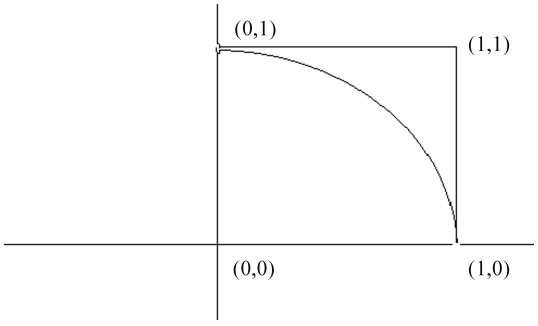

图 5.1 蒙特卡罗算法

例 5.1 验证蒙特卡罗算法(效果如图 5.2 所示)。

```
总点数10000时，蒙特卡洛算法估算的圆周率:3.14880000000000000000
总点数99999时，蒙特卡洛算法估算的圆周率:3.13867138671386700000
总点数9999999时，蒙特卡洛算法估算的圆周率:3.14158111415811140000
```

图 5.2 验证蒙特卡罗算法

Ex5_1.java

```
import java.util.Random;
public class Ex5_1 {
    public static void main(String[] args){
        long totalPoints = 10000;
        double pi = monteCarloPI(totalPoints);
        System.out.printf
        ("总点数%d时,蒙特卡罗算法估算的圆周率:%.20f\n",totalPoints, pi);
        totalPoints = 99999;
        pi = monteCarloPI(totalPoints);
        System.out.printf
        ("总点数%d时,蒙特卡罗算法估算的圆周率:%.20f\n",totalPoints, pi);
        totalPoints = 9999999;
```

```
            pi = monteCarloPI(totalPoints);
            System.out.printf
           ("总点数%d时,蒙特卡罗算法估算的圆周率:%.20f\n",totalPoints, pi);
        }
        public static double monteCarloPI(long totalPoints) {
            int pointsInsideCircle = 0;          //落在圆内的点数
            Random random = new Random();
            for (int i = 0; i < totalPoints; i++) {
                //生成随机点的坐标
                double x = random.nextDouble();
                double y = random.nextDouble();
                //判断点是否落在圆内
                if (x * x + y * y <= 1) {
                    pointsInsideCircle++;
                }
            }
            //计算圆周率
            double pi = 4.0 * pointsInsideCircle / totalPoints;
            return pi;
        }
    }
```

5.1.2 安全随机数与验证码

验证码主要用于防止非法用户或自动化工具进行注册、登录、投票、抽奖等操作,在执行这些操作时,除了要求输入用户名(有些操作要求输入密码),还要求输入正确的验证码。为了保证验证码的安全性,需要通过手机、Email 等渠道把验证码发送给合法用户。

在生成验证码时不宜使用 Random 类,这是因为 Random 类的实例的种子决定了所生成的随机数或随机序列,即使用相同的种子创建 Random 对象后,由于 Random 类使用的生成随机数的算法是线性同余法,那么每次计算得到的随机数序列都是相同的。如果需要安全的随机数或随机序列,则不适合使用 Random 类,因为破解者一旦知道 Random 实例所使用的种子或使用 Random 类在相同的时间创建实例,就会破解随机数或随机序列。

Java 提供了 Random 类的一个子类 SecureRandom,但该子类位于 java.security 包下,它是 Java 中用于生成安全随机数的类。与普通的 Random 类不同,SecureRandom 类提供了更高安全性的随机数,常用于对随机数的安全性要求较高的场景。SecureRandom 使用的种子通常来自操作系统提供的随机源,这些随机源包含了系统的各种熵源信息,如硬件设备的噪声、鼠标移动、键盘敲击时间间隔等,这些信息是难以预测的,因此生成的种子具有较高的随机性和不可预测性。另外,SecureRandom 类采用了更复杂、更安全的随机数生成算法,这些算法经过了严格的密码学验证,能够生成具有较高安全性的随机数序列。例如使用 void nextBytes(byte[] bytes)方法(不是从 Random 类继承的方法)生成一字节密码(可作为生成随机数时的种子)。

SecureRandom 类的常用方法如下。

(1) SecureRandom():构建一个随机数生成器,其种子具有较高的随机性和不可预测性。

(2) SecureRandom(byte[]seed):使用指定的种子创建一个新的随机数生成器,只要种

子相同，生成器就会产生相同的随机数序列。

(3) public void nextBytes(byte[] bytes)：生成随机字节并存放在 bytes 中，该方法每次生成的随机字节都不相同，而且具有很高的安全性，很难被破解内容。bytes 可用于加密的密钥或作为 SecureRandom(byte[] seed)的参数值，即种子。

例 5.2 生成验证码并比较 SecureRandom 和 Random(效果如图 5.3 所示)。

```
长度是10的验证码:WhWz8cf5Qm
Random使用1000作种子，每次生成的随机序列是相同的:
87 35 76 24 92 49 41 45 64 50 79 59 72
SecureRandom的种子有较高的随机性和不可预测性:
SecureRandom使用
[-52, -7, 37, 75, -116, 105, 87, -93, -99, -86, -74, -4, -79, 125, -37, -30]
作种子，每次生成的随机序列都是不相同的:
13 52 49 25 77 18 60 49 28 37 64 99 45
```

图 5.3 生成验证码

Ex5_2.java

```
import java.security.SecureRandom;
import java.util.Random;
public class Ex5_2 {
    public static void main(String[] args) {
        int codeLength = 10;
        String code = verificationCode(10);
        System.out.println("长度是"+codeLength+"的验证码:"+code);
        long seed = 1000;
        Random random = new Random(seed);
        System.out.println("Random使用"+seed+"作种子,每次生成的随机序列是相同的:");
        for(int i = 0;i<=12;i++) {
            System.out.print(random.nextInt(100)+" ");
        }
        System.out.println();
        //创建 SecureRandom 实例
        System.out.println("SecureRandom 的种子有较高的随机性和不可预测性:");
        SecureRandom secureRandom = new SecureRandom();
        byte seedByte[] = new byte[16];
        secureRandom.nextBytes(seedByte);
        secureRandom = new SecureRandom(seedByte);
        String secret = java.util.Arrays.toString(seedByte);
        System.out.println("SecureRandom 使用\n"+secret+
                           "\n 作种子,每次生成的随机序列都是不相同的:");
        for(int i = 0;i<=12;i++) {
            System.out.print(secureRandom.nextInt(100)+" ");
        }
    }
    public static String verificationCode(int length) {
        String characters =
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
        StringBuilder code = new StringBuilder();
        SecureRandom secureRandom = new SecureRandom();
        byte seedByte[] = new byte[16];
```

```
            secureRandom.nextBytes(seedByte);
            secureRandom = new SecureRandom(seedByte);
            for (int i = 0; i < length; i++) {
                int index = secureRandom.nextInt(characters.length());
                code.append(characters.charAt(index));
            }
            return code.toString();
        }
    }
```

5.2 Base64 类和 UUID 类

5.2.1 Base64 编码与图像

Base64 编码是一种将二进制数据转换为文本格式的编码方式，它使用 64 个可打印字符来表示二进制数据。在实际应用中，当需要将二进制数据（像图片、音频等）以文本形式进行传输或存储时，可以使用 java.util 包中 Base64 类的实例把二进制数据转换为文本。Base64 类提供了便捷的编码和解码功能，通过该类的实例，程序能够把二进制数据编码为 String 对象，同时也可以将编码后的 String 对象解码，还原为二进制数据。

对 byte 数组 byteArr 进行 Base64 编码和解码的步骤如下。

（1）编码（Base64.Encoder 内部类负责给出 Base64 编码器）：

```
Base64.Encoder encoder = Base64.getEncoder();       //Base64 编码器
String encoded = encoder.encodeToString(byteArr);
```

（2）解码（Base64.Encoder 内部类负责给出 Base64 解码器）：

```
Base64.Decoder decoder = Base64.getDecoder();     //Base64 解码器
byte[] decodedBytes = decoder.decode(encoded);
```

例 5.3 将图像编码为 Base64 后再解码。

例 5.3 Base64 与图像（效果如图 5.4 所示）。

```
将图像image.jpg编码为Base64:
图像文件长度: 1734852字节
Base64编码的长度: 2313136个字符
Base64编码转化为图像decoded_image.jpg
图像解码并保存为decoded_image.jpg
```

图 5.4 Base64 与图像

Ex5_3.java

```
import java.io.*;
import java.util.Base64;
public class Ex5_3 {
    public static void main(String[] args) {
        File imageFile = new File("image.jpg");
        File decodedFile = new File("decoded_image.jpg");
        System.out.println("将图像"+imageFile.getName()+"编码为 Base64:");
```

```
            try {
                FileInputStream in = new FileInputStream(imageFile);
                byte[] imageBytes = in.readAllBytes();
                //获取原始图像文件的长度
                long fileLength = imageBytes.length;
                Base64.Encoder encoder = Base64.getEncoder();
                String encodedImage = encoder.encodeToString(imageBytes);
                //获取 Base64 编码后字符串的长度
                long encodedLength = encodedImage.length();
                //System.out.println(encodedImage);
                //比较文件和编码后字符串的长度
                System.out.println("图像文件长度: " + fileLength + "字节");
                System.out.println("Base64 编码的长度: "+encodedLength+"个字符");
                System.out.println("Base64 编码转化为图像"+decodedFile.getName());
                Base64.Decoder decoder = Base64.getDecoder();
                byte[] decodedBytes = decoder.decode(encodedImage);
                FileOutputStream out = new FileOutputStream(decodedFile);
                out.write(decodedBytes);
                System.out.println("图像解码并保存为"+decodedFile.getName());
            }
            catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

5.2.2 UUID 识别码和订单号

UUID 是 Universally Unique Identifier(通用唯一识别码)的缩写,一个 UUID 由数字和字母组成,共 128 位标识符。其标准形式是用“-”作连接分隔符,将 32 个十六进制的数字(十六进制的 16 个数字是 0～f)连接分隔成 5 段,呈现为 8 -4 -4 -4 -12 的格式(称为四格式的 UUID),即 1～5 段所包含数字的个数分别是 8、4、4、4 和 12,例如:

```
0ac3b69b-8099-4d7e-84f9-47bafee42e4e
```

UUID 的生成算法使用随机数生成器(随机数生成器不依赖于时间),它的设计目标是在全球范围内保证 UUID 的唯一性。UUID 拥有 128 位的长度,这为其提供了极为庞大的取值空间。在实际应用中,几乎任何不同的 UUID 生成操作都能生成一个独一无二的 UUID。这种唯一性是基于严谨的数学算法和概率原理来保障的。从概率层面分析,出现重复 UUID 的可能性微乎其微。

UUID 类的常用方法如下。

(1) static UUID randomUUID(): 用于获取一个四格式(伪随机生成)的 UUID。

(2) int compareTo(UUID val): 将当前 UUID 与指定的 val 进行比较,此方法按照字典序比较两个 UUID,结果为负数表示当前 UUID 小于指定 UUID,为正数表示当前 UUID 大于指定 UUID,为 0 则表示两者相等。程序可以使用此方法排序 UUID。

在电商系统中,每个订单都需要一个唯一的编号,以便跟踪和管理订单。使用 UUID 作

为订单号，可以确保在高并发的情况下不同用户的订单号不会重复，同时也便于在不同的服务之间共享和查询订单信息。

例 5.4 使用 UUID 作为订单号。

例 5.4 UUID 订单号(效果如图 5.5 所示)。

```
商品A订单号：
0d158f55-b7e4-4051-b6ca-5ab3dfe889c8
商品B订单号：
6ccb19fb-0b98-4965-ae09-1e524cacbec9
商品C订单号：
fd9b0311-e962-46c1-b101-af12905b5b73
```

图 5.5 UUID 订单号

Ex5_4.java

```
import java.util.UUID;
public class Ex5_4 {
    public static void main(String[] args) {
        //生成订单号
        String name[] = {"商品 A","商品 B","商品 C"};
        for(String str:name) {
            UUID uuid = UUID.randomUUID();
            String orderNumber = uuid.toString();
            System.out.println(str+"订单号:\n" + orderNumber);
        }
    }
}
```

5.3 BitSet 类

BitSet 类的实例以位(bit)为基本单元构建，其内部位的数量始终是 64 的倍数。在位序列中(索引从 0 开始)，位值为 1 代表该位的状态为 true，位值为 0 则表示该位的状态为 false。

BitSet 具备动态扩展的能力，其位的数量会根据实际需求按 64 的倍数进行动态增长。举例来说，若当前 BitSet 实例的位数量为 64 位，当尝试将第 65 位设置为 true(也就是把第 65 位的值设置为 1)时，该实例会自动将自身的位数量扩展到 128 位，然后再把第 65 位的值设置为 1。

这种设计使得 BitSet 类的实例能够极为便捷地操作位序列，在处理大量布尔值时展现出高效的性能。

BitSet 类的常用方法如下(位值是 1 等价于 true，位值是 0 等价于 false)。

(1) BitSet()：创建一个新的位集，初始位的数量默认为 64 位，且所有位的初始值均为 0。

(2) BitSet(int nbits)：创建一个位集，其初始位的数量设定为 nbits，索引范围为 0～nbits－1，所有位的初始值均为 0。

(3) void set(int bitIndex)：将指定索引处的位设置为 1。

(4) void clear(int bitIndex)：将指定索引处的位设置为 0。

(5) void set(int bitIndex，boolean value)：将指定索引处的位设置为指定的值。

(6) void set(int fromIndex，int toIndex)：将从指定的 fromIndex(包含)到指定的

toIndex(不包含)的范围内的位设置为1。

(7) void set(int fromIndex,int toIndex,boolean value)：将从指定的fromIndex(包含)到指定的toIndex(不包含)的范围内的位设置为指定的值。

(8) boolean get(int bitIndex)：返回指定索引处位的值。

(9) int size()：返回位的数量。

(10) int length()：返回最高设置位,且值为1的位的索引加1。

(11) int cardinality()：返回值为1的位的数量。

(12) BitSet get(int fromIndex,int toIndex)：返回一个新的位集,该位集由当前位集中从fromIndex(包含)到toIndex(不包含)的位组成。

5.3.1 BitSet与数据去重

所谓数据去重,就是重复的数据只保留一个。在处理大量数据时,判断某个元素是否已经存在是一个常见的需求,而使用BitSet可以高效地实现数据去重。

例5.5使用BitSet类的实例对int数组进行去重操作。在此例中借助BitSet的实例标记数组中每个数字的出现情况。如果某个数字对应的位为0,则表示该数字还未出现过,将其输出或保存,并将对应的位设置为1;如果位值为1,则表示该数字已经出现过,删除此数字,或不进行输出、保存。

例5.5 数组去重(效果如图5.6所示)。

```
[11, 2, 13, 2, 7, 5, 13]
去重后:
[11, 2, 13, 7, 5]
[111, 21, 1, 111, 7, 7, 1]
去重后:
[111, 21, 1, 7]
```

图5.6 数组去重

Ex5_5.java

```
import java.util.BitSet;
import java.util.ArrayList;
public class Ex5_5 {
    public static int[] removeDuplicates(int[] numbers) {
        ArrayList<Integer> list = new ArrayList<>();
        BitSet bitSet = new BitSet();
        for (int num: numbers) {
            if (!bitSet.get(num)) {
                list.add(num);
                bitSet.set(num);
            }
        }
        //创建一个新的 int 数组
        int[] result = new int[list.size()];
        //将 ArrayList 中的元素复制到 int 数组中
        for (int i = 0; i < list.size(); i++) {
            result[i] = list.get(i);
        }
```

```
        return result;
    }
    public static void main(String[] args) {
        int[] arr = {11, 2, 13, 2, 7, 5, 13};
        System.out.println(java.util.Arrays.toString(arr) + "\n去重后:");
        arr = removeDuplicates(arr);
        System.out.println(java.util.Arrays.toString(arr));
        arr = new int[]{111, 21, 1, 111, 7, 7, 1};
        System.out.println(java.util.Arrays.toString(arr) + "\n去重后:");
        arr = removeDuplicates(arr);
        System.out.println(java.util.Arrays.toString(arr));
    }
}
```

5.3.2 BitSet 与筛选法

筛选法,简称为筛法,它是由古希腊数学家埃拉托斯特尼提出的一种用于简单检定素数的算法。在古希腊,人们习惯将数字写在涂蜡的板上,当需要划去某个数时,便会在上面标记一个小点。当寻找素数的工作完成后,板上留下众多小点,看起来如同一个筛子,因此埃拉托斯特尼的这种方法被命名为筛选法。由于 1 并非素数,筛法的具体操作是先将 2 至目标数 n 的所有自然数按顺序排列。算法从 2 开始,2 是素数,将其保留,然后把 2 之后所有能被 2 整除的数(即 2 的倍数的数)全部划去。此时,2 后面第一个未被划去的数是 3,3 也是素数,将其保留,再把 3 之后所有能被 3 整除的数(即 3 的倍数的数)都划去。此时,3 后面第一个未被划去的数是 5,同样作为素数保留,然后把 5 之后所有能被 5 整除的数(即 5 的倍数的数)都划去。以此类推,持续进行这样的操作,最终就能得到 n 以内的全部素数。

为了得到 1～n 的素数,按照筛选的思想,BitSet 实例将其所有大于或等于 2 的位都设置为 true,表示这些数可能是素数。然后从 2 开始,将每个素数的倍数对应的位设置为 false,表示这些数不是素数。最后输出所有位为 true 的数,即为素数。基于 BitSet 实例实现筛选法具有较高的效率。

例 5.6 使用 BitSet 实现筛选法(效果如图 5.7 所示)。

```
位的数目:128
1~100范围内的素数:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
素数的数目 (位值是true的数目) :25
```

图 5.7 使用 BitSet 实现筛选法

Ex5_6.java

```
import java.util.BitSet;
public class Ex5_6 {
    public static void main(String[] args) {
        int n = 100;
        BitSet bitSet = new BitSet(n+1);
        System.out.println("位的数目:"+bitSet.size());
        for (int i = 2; i <= n; i++) {
            //设置第 i 位的值是 true,即位值是 1
```

```
            bitSet.set(i);
        }
        for (int i = 2; i * i <= n; i++) {
            if (bitSet.get(i)) {
                for (int j = i * i; j <= n; j += i) {
                    //设置第 j 位的值是 false,即位值是 0
                    bitSet.clear(j);
                }
            }
        }
        System.out.println("1~"+n+"的素数:");
        for (int i = 2; i < bitSet.length(); i++) {
            if (bitSet.get(i)) {
                System.out.print(i + " ");
            }
        }
        System.out.println("\n 素数的数目(位值是 true 的数目):"+bitSet.cardinality());
    }
}
```

例 5.6 代码按照筛选法，首先从 2 开始，将 2 的倍数对应的位设置为 false。因为 2 的倍数(除了 2 本身)都能被 2 整除，所以它们不是素数。接着看 3，由于 3 没有被之前的步骤标记为 false，说明它不能被 2 整除，所以 3 是素数，然后将 3 的倍数对应的位设置为 false。对于一个数 i，如果它在之前的筛选过程中没有被标记为 false，那么它就是素数，因为它不能被小于它的素数整除。然后将 i 的倍数对应的位设置为 false。按照这个过程操作，只需遍历到 i 的平方小于 n(如果一个数 n 不是素数，那么它一定有一个小于或等于 Math.sqrt(n)的因子)，就能排除 2～n 的所有合数。

5.4 IntSummaryStatistics 类和 DoubleSummaryStatistics 类

IntSummaryStatistics 和 DoubleSummaryStatistics 是 Java 8 引入的实用类，分别用于收集和计算 int 整数与 double 浮点数的数据统计信息。借助这两个类，能便捷地对一组 int 数据和一组 double 浮点数进行统计，例如计算数据的总数、总和、平均值、最大值和最小值等。

5.4.1 统计评委的打分

例 5.7 首先使用 IntSummaryStatistics 类计算一组 int 整数(模拟评委给出的分数)数据的总数、总和、平均值、最大值和最小值，然后去掉一个最大值(最高分)和一个最小值(最低分)，再次使用该类计算剩余数据的总数、总和、平均值。

例 5.7 统计评委的打分(效果如图 5.8 所示)。

```
数的数量: 10
数的平均值: 93.0
数的总和: 930
最大值之一: 99
最小值之一: 71
去掉一个最大值和一个最小值后数的平均值: 95.0
去掉一个最大值和一个最小值后的数的数量: 8
去掉一个最大值和一个最小值后的数的总和: 760
```

图 5.8 统计评委的打分