第 3 章

数组、字符串和集合类

本章导读

Java 的数组、字符串和集合类在编程中扮演着重要的角色。数组提供了一种有序、固定大小的数据存储方式，广泛应用于数据的存储、遍历和排序等操作。字符串作为文本数据的表示形式，在处理文本、字符串拼接和匹配等方面起着重要作用。集合类则提供了动态存储和操作对象的能力，如添加、删除、查找和排序等，方便开发者管理和操作数据。通过灵活运用这些数据结构，可以优化代码逻辑，提高程序的效率和可维护性。

本章的学习目标是掌握一维数组和多维数组的创建和使用，了解多维数组在多个领域的应用示例；掌握字符串的各种操作方法，使用字符分析器来解析字符串，了解如何判断字符串是否为回文串；探索正则匹配表达式的编写和使用以及各种集合类的使用方法和存储数据特点。

3.1 数　　组

数组是一种用于存储多个相同类型数据的容器。它是一维或多维的、长度固定的对象。数组具有以下几个重要的特征。首先，数组可以存储同一类型的元素。例如，一个整型数组可以存储一系列整数，字符串数组则可以存储一组字符串，这样能够方便地组织和处理大量数据。其次，数组的长度是固定的，一旦创建后无法改变。这意味着在使用数组前，需要明确知道需要存储的元素数量。如果需要动态调整长度，可以使用集合类代替数组。数组在程序中有广泛的应用。它可以用于存储和处理数据集合，如学生信息、温度读数等。通过数组，可以方便地访问和操作数组中的元素，例如查找最大值、排序等操作。数组还可以被用作其他数据结构的基础，例如栈、队列和矩阵等。另外，数组也充当了很多算法和数据结构的重要角色。例如，线性查找、二分查找、冒泡排序和快速排序等算法都需要用到数组来实现。总之，数组是 Java 编程中一种重要而常用的数据结构，它提供了一种存储和操作数据的有效方式。掌握数组的概念和使用方法可以更好地处理数据，并且能够应对各种编程挑战。

3.1.1 数组的创建与使用

1. 一维数组的创建

在 Java 中，创建一维数组的方法有多种。一维数组是由相同类型的元素组成的线性数据结构，可以通过索引来访问和操作数组中的元素。以下是创建一维数组的常用方法示例。

【例 3-1】　创建一维数组的常用方法示例。

```
import java.util.Arrays;

public class NumericArrayCreation {
    public static void main(String[] args) {
        //方法 1: 声明并同时初始化一个整型数组
        int[] arr1 = {1, 2, 3, 4, 5};
        //方法 2: 使用 new 关键字创建并同时初始化一个整型数组
        int[] arr2 = new int[]{1, 2, 3, 4, 5};
        //方法 3: 先声明再初始化一个整型数组
        int[] arr3;
        arr3 = new int[]{1, 2, 3, 4, 5};
        //方法 4: 声明一个整型数组,然后通过索引逐个赋值
        int[] arr4 = new int[5];
        arr4[0] = 1;
        arr4[1] = 2;
        arr4[2] = 3;
        arr4[3] = 4;
        arr4[4] = 5;
        //方法 5: 只声明一个整型数组,未初始化
        int[] arr5;
        //方法 6: 先声明一个整型数组,再通过 new 关键字初始化
        int[] arr6;
        arr6 = new int[5];
        //方法 7: 使用 Arrays 类的静态方法创建并初始化一个整型数组
        int[] arr7 = new int[5];
        Arrays.fill(arr7, 1);                    //将数组元素全部填充为 1
        //方法 8: 使用循环逐个赋值来创建一个整型数组
        int[] arr8 = new int[5];
        for (int i = 0; i < arr8.length; i++) {
            arr8[i] = i + 1;
        }
        //方法 9: 使用 Arrays 类的静态方法创建一个整型数组,并初始化为指定范围的值
        int[] arr9 = new int[5];
        Arrays.setAll(arr9, i -> i + 1);         //设置数组元素为 1, 2, 3, 4, 5
        //方法 10: 使用 Stream 流来创建一个整型数组
        //创建一个范围为 1~5 的整型数组
        int[] arr10 = java.util.stream.IntStream.range(1, 6).toArray();
    }
}
```

这段代码演示了一维整型数组的 10 种创建方式,包括使用花括号赋初值、使用 new 关键字创建并初始化、先声明再初始化、通过索引逐个赋值、只声明不初始化、通过 new 关键字初始化、使用 Arrays 类的静态方法进行数组填充、使用循环逐个赋值、使用 Arrays 类的静态方法创建并初始化、使用 Stream 流来创建。

2. 一维数组的使用

【例 3-2】　一维数组的常用操作示例。

```
public class NumericArrayUsage {
    public static void main(String[] args) {
        //创建一个整型数组并初始化
        int[] arr = {1, 2, 3, 4, 5};
        //访问数组元素
        int firstElement = arr[0];                    //获取第一个元素
        System.out.println("第一个元素是: " + firstElement);
        //修改数组元素
        arr[0] = 10;                                  //修改第一个元素的值
        System.out.println("修改后,第一个元素是: " + arr[0]);
        //获取数组长度
        int length = arr.length;
        System.out.println("数组长度是: " + length);
        //遍历数组
        for (int i = 0; i < arr.length; i++) {
            System.out.println("数组元素 " + i + " 的值是: " + arr[i]);
        }
        //使用增强 for 循环遍历数组
        for (int num : arr) {
            System.out.println("数组元素的值是: " + num);
        }
        //数组排序
        java.util.Arrays.sort(arr);                   //对数组进行升序排序
        //查找数组元素
        int searchValue = 3;
        //使用二分查找方法查找指定元素
        int index = java.util.Arrays.binarySearch(arr, searchValue);
        System.out.println("元素 " + searchValue + " 的索引是: " + index);
        //数组复制
        int[] copy = java.util.Arrays.copyOf(arr, arr.length);        //复制数组
        System.out.println("复制的数组元素: ");
        for (int num : copy) {
            System.out.println(num);
        }
        //数组填充
        int fillValue = 0;
        java.util.Arrays.fill(arr, fillValue);      //将数组元素全部填充为指定值
        System.out.println("填充后的数组元素: ");
        for (int num : arr) {
            System.out.println(num);
        }
        //判断数组是否相等
        boolean isEqual = java.util.Arrays.equals(arr, copy);   //判断两个数组是否相等
        System.out.println("两个数组是否相等: " + isEqual);
        //数组转换为字符串
        String arrString = java.util.Arrays.toString(arr); //将数组转换为字符串
        System.out.println("数组转换为字符串: " + arrString);
    }
}
```

这段代码演示了一维数组的常见用法。通过这段代码，可以了解如何创建、访问、修改、遍历、排序、查找、复制、填充、判断相等和转换为字符串等操作。每行代码都添加了注释，以便更好地理解代码的作用。

请注意，代码中使用的是 java.util.Arrays 类来提供一些数组操作方法，包括排序、复制、填充、查找等。同时，使用 for 循环可以更方便地遍历数组元素。最后，通过输出语句将结果打印出来，以便观察每个操作的效果。

3. 数组元素的默认初始化值

了解数组的默认值是很重要的，在使用数组前如果没有显式地初始化数组元素，就会根据默认值规则来处理数组元素，以避免出现意外的结果或错误。同时，数组的默认值也提供了一种方便的初始化方式，当需要创建一个初始值相同的数组时，可以直接声明并分配空间，省去了手动赋值的步骤。

在 Java 中，数组在创建时会被自动初始化为默认值，具体取决于数组元素的类型。下面是 Java 数组的默认值规则。

- 对于整型数组（如 int[]、short[]、long[]等），默认值为 0。
- 对于浮点型数组（如 float[]、double[]等），默认值为 0.0。
- 对于布尔型数组（如 boolean[]），默认值为 false。
- 对于字符型数组（如 char[]），默认值为'\u0000'，即空字符。
- 对于引用类型数组（如 String[]、Object[]等），默认值为 null，即数组中的每个元素都指向空对象。

4. 多维数组的创建

【例 3-3】　多维数组的创建方式示例。

```
public class MultiDimensionalArrayCreation {
    public static void main(String[] args) {
        //1. 直接初始化
        int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
        //2. 动态初始化第一维，静态初始化第二维
        int[][] array2 = new int[2][];
        array2[0] = new int[]{1, 2, 3};
        array2[1] = new int[]{4, 5, 6};
        //3. 动态初始化第一维，动态初始化第二维
        int[][] array3 = new int[2][3];
        array3[0] = new int[3];
        array3[1] = new int[3];
        //4. 静态初始化并指定每个维度的长度
        int[][] array4 = new int[2][3];
        array4[0] = new int[3];
        array4[1] = new int[3];
        //5. 使用 Arrays.fill()方法填充二维数组
        int[][] array5 = new int[2][3];
        for (int i = 0; i < array5.length; i++) {
            Arrays.fill(array5[i], 0);
        }
```

```
        //6. 使用嵌套循环动态初始化二维数组
        int[][] array6 = new int[2][3];
        for (int i = 0; i < array6.length; i++) {
            for (int j = 0; j < array6[i].length; j++) {
                array6[i][j] = i + j;
            }
        }
        //7. 使用 Arrays.copyOf()方法复制已有二维数组
        int[][] sourceArray = {{1, 2}, {3, 4, 5}};
        int[][] array7 = new int[sourceArray.length][];
        for (int i = 0; i < sourceArray.length; i++) {
            array7[i] = Arrays.copyOf(sourceArray[i], sourceArray[i].length);
        }
        //8. 使用 System.arraycopy()方法复制已有二维数组
        int[][] sourceArray = {{1, 2}, {3, 4, 5}};
        int[][] array8 = new int[sourceArray.length][];
        for (int i = 0; i < sourceArray.length; i++) {
            int[] row = new int[sourceArray[i].length];
            System.arraycopy(sourceArray[i], 0, row, 0, sourceArray[i].length);
            array8[i] = row;
        }
    }
}
```

上述代码展示了 8 种创建多维数组的方式。

（1）直接初始化：通过在花括号内指定每个元素的值来初始化数组。

（2）动态初始化第一维，静态初始化第二维：先初始化第一维，然后为每个第一维元素静态初始化第二维。

（3）动态初始化第一维，动态初始化第二维：先初始化第一维，然后为每个第一维元素动态初始化第二维。

（4）静态初始化并指定每个维度的长度：直接初始化数组，并指定每个维度的长度。

（5）使用 Arrays.fill()方法填充二维数组：使用 Arrays.fill()方法可以把数组的每个元素都填充为指定的值。

（6）使用嵌套循环动态初始化二维数组：使用嵌套循环可以为数组的每个元素动态赋值。

（7）使用 Arrays.copyOf()方法复制已有二维数组：使用 Arrays.copyOf()方法可以复制已有的二维数组。

（8）使用 System.arraycopy()方法复制已有二维数组：使用 System.arraycopy()方法可以复制已有的二维数组。

5. 多维数组的使用

二维数组是一种常用的数据结构，可以用来表示矩阵和表格等数据。对于数值二维数组，常见的操作包括获取行数和列数、查找最大值与最小值、计算总和与平均值、判断是否存在特定的值、转置操作、判断是否为对称矩阵以及将二维数组转换为一维数组等。下面通过一段代码展示多维数组的常见操作。

【例 3-4】　多维数组的常见操作示例。

```
public class MultiDimensionalArrayOperations {
    public static void main(String[] args) {
        //假设已创建了一个名为 matrix 的二维数组
        //获取二维数组的行数和列数
        int rows = matrix.length;
        int columns = matrix[0].length;
        //找到二维数组中的最大值
        int max = matrix[0][0];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] > max) {
                    max = matrix[i][j];
                }
            }
        }
        //找到二维数组中的最小值
        int min = matrix[0][0];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                if (matrix[i][j] < min) {
                    min = matrix[i][j];
                }
            }
        }
        //计算二维数组所有元素的总和
        int sum = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                sum += matrix[i][j];
            }
        }
        //计算二维数组所有元素的平均值
        double average = (double) sum / (rows * columns);
        //对二维数组进行转置操作
        int[][] transposedMatrix = new int[columns][rows];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                transposedMatrix[j][i] = matrix[i][j];
            }
        }
        //判断二维数组是否为对称矩阵(即行列数相等且元素对应位置相等)
        boolean isSymmetric = true;
        if (rows != columns) {
            isSymmetric = false;
        } else {
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < columns; j++) {
                    if (matrix[i][j] != matrix[j][i]) {
```

```
                        isSymmetric = false;
                        break;
                    }
                }
            }
        }
        //将二维数组转换为一维数组
        int[] flattenedArray = new int[rows * columns];
        int index = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                flattenedArray[index] = matrix[i][j];
                index++;
            }
        }
    }
}
```

3.1.2 多维数组的应用

多维数组在 Java 中有着广泛的应用，可以用于存储和处理具有多个维度的数据。以下是其中一些常见的应用场景。

（1）矩阵和图像处理：多维数组可以用来表示矩阵和图像数据，通过对多维数组进行操作，可以进行矩阵计算、图像滤波、边缘检测等各种图像处理任务。

（2）数据表格：多维数组可以用来存储和处理数据表格，例如电子表格或数据库中的数据。通过使用多维数组，可以轻松地进行数据的增删改查操作，并进行数据分析和统计。

（3）二维游戏地图：许多游戏中的地图可以使用二维数组来表示。通过在数组中存储不同类型的地形、道具或角色信息，可以实现游戏中的移动、碰撞检测和事件触发等功能。

（4）图形界面布局：图形用户界面(GUI)中的界面布局通常使用多维数组。通过在数组中指定窗口、按钮和其他组件的位置和大小，可以实现灵活的界面设计。

（5）多维数据集合：多维数组可以用于存储和处理多维数据集合。例如，可以使用三维数组来表示立体空间中的数据点，如气象数据、音频数据或医学影像数据。

（6）数独游戏：数独游戏是一种通过填充数独盘面上的空格来解决逻辑问题的游戏。多维数组可以用来表示数独盘面，并提供有效的算法来解决和生成数独谜题。

（7）统计分析：在统计学和数据分析中，多维数组被广泛用于存储和处理大量的数据。通过使用多维数组，可以进行数据聚合、相关性分析、回归分析等各种统计分析任务。

1. 图像处理

在图像处理方面，Java 的多维数组被广泛应用。图像可以被表示为一个二维或三维数组，其中每个元素存储着像素的颜色信息。多维数组提供了对图像数据的高效访问和操作。通过遍历数组，可以执行各种处理操作，如调整亮度、对比度，应用滤镜效果等。多维数组还可用于图像分割、特征提取和模式识别。在图像压缩方面，多维数组可以通过编码像素值之间的相关性，实现高效的压缩算法。此外，多维数组也支持基于像素的直方图处理，用于图

像增强、颜色校正和图像识别。总而言之，Java 的多维数组在图像处理中发挥着重要作用，为图像处理算法和技术的实现提供了强大的数据结构。通过合理利用多维数组，可以实现高效、精确和创新的图像处理应用。下面是一个简单的图像处理示例代码。

【例 3-5】　使用多维数组将彩色图像转换为灰度图像。

```
public class ImageProcessing {
    public static void main(String[] args) {
        //一个 3×3 的彩色图像
        int[][] image = {{255, 0, 0}, {0, 255, 0}, {0, 0, 255}};
        //将图像变为灰度图
        int[][] grayImage = convertToGrayscale(image);
        //输出灰度图像
        for (int i = 0; i < grayImage.length; i++) {
            for (int j = 0; j < grayImage[i].length; j++) {
                System.out.print(grayImage[i][j] + " ");
            }
            System.out.println();
        }
    }
    private static int[][] convertToGrayscale(int[][] image) {
        int[][] grayImage = new int[image.length][image[0].length];
        for (int i = 0; i < image.length; i++) {
            for (int j = 0; j < image[i].length; j++) {
                int red = (image[i][j] >> 16) & 0xFF;
                int green = (image[i][j] >> 8) & 0xFF;
                int blue = image[i][j] & 0xFF;
                int gray = (red + green + blue) / 3;
                grayImage[i][j] = gray;
            }
        }
        return grayImage;
    }
}
```

这个示例代码中，定义了一个 image 数组来表示彩色图像，每个像素点由 RGB 颜色值组成。然后使用 convertToGrayscale() 方法将彩色图像转换为灰度图像，最后输出灰度图像。

2. 矩阵运算

在矩阵运算方面，Java 的多维数组是非常有用的。多维数组可以用来表示和操作矩阵数据。矩阵可以被表示为一个二维数组，其中每个元素存储着矩阵中的一个元素值。

通过多维数组，可以进行各种矩阵运算，如矩阵相加、相减和相乘。可以遍历两个矩阵的对应元素，并将它们进行相应的运算。另外，还可以进行矩阵的转置、求逆和求行列式等操作。多维数组还可以用于实现线性代数中的其他操作，如矩阵求解线性方程组以及求特征值和特征向量。这些操作对于科学计算、机器学习和数据分析等领域非常重要。

Java 提供了丰富的库和工具来支持矩阵运算，如 Apache Commons Math 库和 EJML (Efficient Java Matrix Library)，这些库提供了优化的矩阵操作算法，并简化了矩阵运算的

实现过程。总而言之，Java 的多维数组在矩阵运算方面具有广泛的应用。通过多维数组，可以方便地表示和操作矩阵数据，实现各种线性代数运算和科学计算任务。下面是一个矩阵相加的示例代码。

【例 3-6】 使用多维数组实现矩阵相加。

```
public class MatrixAddition {
    public static void main(String[] args) {
        int[][] matrix1 = {{1, 2}, {3, 4}};
        int[][] matrix2 = {{5, 6}, {7, 8}};
        int[][] result = addMatrices(matrix1, matrix2);
        //输出结果矩阵
        for (int i = 0; i < result.length; i++) {
            for (int j = 0; j < result[i].length; j++) {
                System.out.print(result[i][j] + " ");
            }
            System.out.println();
        }
    }
    private static int[][] addMatrices(int[][] matrix1, int[][] matrix2) {
        int rows = matrix1.length;
        int cols = matrix1[0].length;
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = matrix1[i][j] + matrix2[i][j];
            }
        }
        return result;
    }
}
```

这个示例代码中，定义了两个矩阵 matrix1 和 matrix2，然后使用 addMatrices()方法将这两个矩阵相加，最后输出结果矩阵。

3. 游戏开发

在游戏开发方面，Java 的多维数组有着广泛的应用。首先，多维数组可以用来构建游戏的地图和场景。通过二维数组，可以定义游戏的网格状地图，每个元素代表一个单元格或坐标点，存储地形、障碍物和物体等信息。这为游戏世界的建模提供了方便的方式。其次，多维数组也可以用于管理游戏对象。例如，可以使用三维数组来表示游戏中的粒子系统、碰撞检测和物理发动机等。每个元素可以存储对象的状态和属性，如位置、速度和质量等。此外，多维数组还可用于实现游戏中的地图编辑器和关卡设计工具。通过多维数组，可以创建可视化的编辑界面，允许开发人员和设计师轻松地修改和定制游戏地图和关卡。Java 中的多维数组操作灵活且高效，可以通过索引和遍历对数组进行快速访问和修改。此外，Java 还提供了丰富的图形库和游戏发动机，如 LibGDX 和 JMonkeyEngine，来支持基于多维数组的游戏开发。总的来说，Java 的多维数组在游戏开发中提供了强大的数据结构和工具，可用于表示、操作和管理游戏中的各种元素和对象。下面演示如何使用多维数组表示游戏

地图。

【例 3-7】　使用多维数组表示游戏地图。

```
public class GameMap {
    public static void main(String[] args) {
        char[][] map = {
            {'#', '#', '#', '#', '#', '#', '#', '#'},
            {'#', '.', '.', '.', '#', '.', '.', '#'},
            {'#', '.', '#', '.', '#', '.', '#', '#'},
            {'#', '.', '#', '.', '.', '.', '.', '#'},
            {'#', '#', '#', '#', '#', '#', '#', '#'}
        };
        //输出游戏地图
        for (int i = 0; i < map.length; i++) {
            for (int j = 0; j < map[i].length; j++) {
                System.out.print(map[i][j]);
            }
            System.out.println();
        }
    }
}
```

这个示例代码中，定义了一个二维字符数组 map 来表示游戏地图，其中'#'表示墙，'.'表示空地。然后通过遍历数组，输出游戏地图。

3.2　字　符　串

Java 字符串是一种不可变的对象，用于表示和操作文本数据。它由字符序列组成，可以通过双引号括起来创建。字符串具有丰富的操作方法，如连接、提取子串、搜索和替换等。为了节省内存，Java 使用字符串常量池来管理字符串对象。字符串在文本处理、数据解析、网络通信等场景中广泛应用，其不可变性和丰富的操作方法使得字符串处理更加方便和高效。

3.2.1　字符串的介绍

(1) 字符串对象：字符串是由字符序列组成的对象，可以通过双引号括起来创建字符串对象，例如"Hello，World!"。

(2) 字符串不可变性：字符串对象一旦被创建，就无法被修改。每次对字符串的修改都会创建一个新的字符串对象，原始字符串不会改变。这种不可变性确保了字符串的安全性和线程安全性。

(3) 字符串操作：Java 提供了丰富的字符串操作方法，例如连接字符串、提取子字符串、搜索和替换等。可以使用字符串方法来执行这些操作，如 concat()、substring()、indexOf()和 replace()等。

(4) 字符串比较：可以使用 equals()方法来比较两个字符串是否相等。Java 还提供了 compareTo()方法用于比较字符串的大小关系。

（5）字符串拼接：在 Java 中，可以使用“+”操作符进行字符串的拼接。字符串拼接会创建一个新的字符串对象，并将两个字符串连接起来。

（6）字符串格式化：Java 提供了 String.format()方法和 printf()函数来格式化字符串，可以按照指定的格式输出字符串中的数据。

（7）字符串长度和访问：可以使用 length()方法获取字符串的长度，使用 charAt()方法按索引访问字符串中的字符。

（8）字符串常量池：为了节省内存，Java 使用了字符串常量池来管理字符串对象。相同的字符串字面值只会在常量池中创建一次，不同的引用都指向同一个字符串对象。

（9）Java 字符串在许多应用场景中被广泛使用，如文本处理、数据解析、网络通信和用户界面等。可以用于拼接、比较、搜索和替换文本数据，也可用于 URL 编码、JSON 解析和 XML 处理等。字符串还常用于构建日志消息、错误信息和用户界面元素，具有广泛的应用和灵活性。

3.2.2 字符串操作

在 Java 中，字符串操作可以分为三类：与操作相关的方法（如 concat()、replace()），与判断相关的方法（如 equals()、startsWith()），以及与获取相关的方法（如 length()、substring()）。下面分别介绍三类操作的常用方法。

1. 字符串常用操作方法

String 类的字符串常用操作方法如表 3.1 所示。

表 3.1 String 类的字符串常用操作方法

方　法	作　用
String concat(String str)	将指定的字符串连接到原始字符串的末尾，并返回连接后的新字符串
String replace(String target,String replacement)	将原始字符串中的目标字符序列替换为指定的替换字符序列，并返回替换后的新字符串
String substring(int beginIndex)	返回从指定索引开始到原始字符串末尾的子字符串
String substring(int beginIndex,int endIndex)	返回从指定的开始索引到结束索引之间的子字符串（不包括结束索引）
String[] split(String regex)	根据给定的正则表达式把原始字符串分割成一个字符串数组，并返回该数组
String trim()	去除原始字符串开头和结尾处的空格，并返回去除空格后的新字符串
String toLowerCase()	将原始字符串转换为小写形式，并返回转换后的新字符串
String toUpperCase()	将原始字符串转换为大写形式，并返回转换后的新字符串
char[] toCharArray()	将原始字符串转换为字符数组，并返回该字符数组

【例 3-8】 字符串的常用操作示例。

```
public class StringManipulationExample {
    public static void main(String[] args) {
        //1. 字符串拼接：使用"+"符号或者 concat()方法
        String str1 = "Hello"
        String str2 = "World";
        //2. 字符串拼接使用"+"符号拼接字符串
        String result1 = str1 + " " + str2;
        System.out.println(result1);        //输出：Hello World
        //3. 使用 concat()方法拼接字符串
        String result2 = str1.concat(" ").concat(str2);
        System.out.println(result2);        //输出：Hello World
        //4. 字符串替换：replace()方法
        String str = "Hello, Java!";
        String result = str.replace("Java", "World");
        System.out.println(result);         //输出：Hello, World!
        //5. 字符串截取：substring()方法
        String str = "Hello, World!";
        String result = str.substring(7, 12);
        System.out.println(result);         //输出：World
        //6. 字符串分割：split()方法
        String str = "Java is a programming language";
        //按空格分割字符串
        String[] result = str.split(" ");
        for (String s : result) {
            System.out.println(s);
        }
        //7. 去除字符串空格：trim()方法用于去掉字符串前后的空格
        String str = "   Hello, World!   ";
        String result = str.trim();
        System.out.println(result);         //输出：Hello, World!
        //8. 大小写转换：toLowerCase()和 toUpperCase()方法
        String str = "Hello, World!";
        //(1)将字符串转换为小写
        String lowercase = str.toLowerCase();
        System.out.println(lowercase);      //输出：hello, world!
        //(2)将字符串转换为大写
        String uppercase = str.toUpperCase();
        System.out.println(uppercase);      //输出：HELLO, WORLD!
        //9. 字符串转数组：使用 toCharArray()方法将字符串转换为字符数组
        String str = "Hello";
        char[] charArray = str.toCharArray();
        for (char c : charArray) {
            System.out.println(c);
        }
    }
}
```

2. 字符串判断常用方法

String 类的字符串判断常用方法如表 3.2 所示。

表 3.2 String 类的字符串判断常用方法

方　法	说　明
boolean isEmpty()	返回字符串是否为空
boolean contains(String sequence)	返回字符串是否包含指定的字符序列
boolean startsWith(String prefix)	返回字符串是否以指定的前缀开始
boolean endsWith(String suffix)	返回字符串是否以指定的后缀结束
boolean equals(Object object)	判断字符串是否与给定对象相等(区分大小写)
boolean equalsIgnoreCase(String str)	判断字符串是否与给定字符串相等
int indexOf(String str)	返回字符串在原始字符串中第一次出现的索引位置
int lastIndexOf(String str)	返回字符串在原始字符串中最后一次出现的索引位置

【例 3-9】 字符串判断常用方法示例。

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Hello World";
        //1. 判断字符串是否为空
        boolean isEmpty = str.isEmpty();
        System.out.println("isEmpty: " + isEmpty);
        //2. 判断字符串的长度是否为 0
        boolean isLengthZero = (str.length() == 0);
        System.out.println("isLengthZero: " + isLengthZero);
        //3. 判断字符串是否包含指定的字符序列
        boolean contains = str.contains("World");
        System.out.println("contains: " + contains);
        //4. 判断字符串是否以指定的前缀开始
        boolean startsWith = str.startsWith("Hello");
        System.out.println("startsWith: " + startsWith);
        //5. 判断字符串是否以指定的后缀结束
        boolean endsWith = str.endsWith("World");
        System.out.println("endsWith: " + endsWith);
        //6. 判断两个字符串是否相等(区分大小写)
        boolean equals = str.equals("hello world");
        System.out.println("equals: " + equals);
        //7. 判断两个字符串是否相等(忽略大小写)
        boolean equalsIgnoreCase = str.equalsIgnoreCase("hello world");
        System.out.println("equalsIgnoreCase: " + equalsIgnoreCase);
        //8. 判断字符串在原始字符串中第一次出现的索引位置
        int indexOf = str.indexOf("o");
        System.out.println("indexOf: " + indexOf);
        //9. 判断字符串在原始字符串中最后一次出现的索引位置
        int lastIndexOf = str.lastIndexOf("o");
        System.out.println("lastIndexOf: " + lastIndexOf);
    }
}
```

3. 字符串获取常用方法

String 类的字符串获取常用方法如表 3.3 所示。

表 3.3　String 类的字符串获取常用方法

方　法	说　明
charAt(int index)	返回指定索引处的字符
indexOf(char ch)	返回指定字符在字符串中第一次出现的索引位置
lastIndexOf(char ch)	返回指定字符在字符串中最后一次出现的索引位置
length()	返回字符串的长度

【例 3-10】　字符串获取常用方法示例。

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Hello World";
        //1. 获取指定位置的字符
        char charAt = str.charAt(4);
        System.out.println("charAt: " + charAt);                    //输出: o
        //2. 查找指定字符在字符串中第一次出现的索引位置
        int indexOf = str.indexOf('o');
        System.out.println("indexOf: " + indexOf);                  //输出: 4
        //3. 查找指定字符在字符串中最后一次出现的索引位置
        int lastIndexOf = str.lastIndexOf('o');
        System.out.println("lastIndexOf: " + lastIndexOf);          //输出: 7
        //4. 获取字符串的长度
        int length = str.length();
        System.out.println("length: " + length);                    //输出: 11
    }
}
```

3.2.3　字符分析器

Java 字符分析器是用于处理和分析字符数据的工具。在 Java 中，有几个常用的字符分析器类，包括 StringTokenizer、Scanner 和 StreamTokenizer。

(1) StringTokenizer 类可以将字符串按照指定的分隔符拆分为多部分，能够逐个访问和处理这些部分。它提供了方法来获取下一部分、判断是否还有更多部分以及获取分隔符等信息。

(2) Scanner 类是一个强大的字符分析器，可以从输入源(如键盘、文件)读取并解析数据。它可以按照指定的模式扫描和提取字符串、数字等内容，并提供了各种方法来检测和处理不同类型的数据。

(3) StreamTokenizer 类是用于解析输入流的字符分析器。它可以将输入流分解为标记，例如数字、字符串、特殊字符等，并提供了方法来获取和处理这些标记。

这些字符分析器类在 Java 中提供了不同的功能和灵活性，能够方便地处理和分析字符数据。它们可以用于文本解析、数据提取、格式化输入等各种应用场景。通过选择合适的字

符分析器，可以实现高效、灵活地处理字符数据。

1. StringTokenizer 类

StringTokenizer 是 Java 中最基本的字符分析器。它根据指定的分隔符将字符串分解成多个标记。默认情况下，分隔符可以是空格、制表符、换行符等。StringTokenizer 类的常用方法如表 3.4 所示。

表 3.4　StringTokenizer 类的常用方法

方　法　名	说　　明
hasMoreTokens()	判断是否还有更多的标记
nextToken()	获取并返回下一个标记
countTokens()	返回剩余标记的数量
nextElement()	获取并返回与 nextToken()方法相同的标记
hasMoreElements()	判断是否还有更多的元素
countTokens()	返回剩余标记的数量
resetSyntax()	重置语法
toString()	将对象转换为字符串表示形式
setDelimiters(String delimiters)	设置分隔符
setLocale(Locale locale)	设置区域设置以在转换过程中使用

【例 3-11】　使用 StringTokenizer 类分割字符串示例。

```
String str = "Hello,World,Java";
StringTokenizer tokenizer = new StringTokenizer(str, ",");
while (tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
    System.out.println(token);
}
```

上述代码将字符串"Hello,World,Java"按照逗号分隔成三个标记输出。

2. Scanner 类

Scanner 类可以从输入流（如标准输入、文件）中读取数据，并按照指定的分隔符将其分解成标记。它可以扫描各种类型的数据，如整数、浮点数、字符串等。Scanner 类的常用方法如表 3.5 所示。

表 3.5　Scanner 类的常用方法

方　法　名	说　　明
hasNext()	判断是否还有下一个标记
next()	获取并返回下一个标记
hasNextLine()	判断是否还有下一行内容
nextLine()	获取并返回下一行内容

续表

方　法　名	说　　明
hasNextInt()	判断下一个标记是否为整数
nextInt()	获取并返回下一个整数
hasNextDouble()	判断下一个标记是否为浮点数
nextDouble()	获取并返回下一个浮点数
hasNextBoolean()	判断下一个标记是否为布尔值
nextBoolean()	获取并返回下一个布尔值
useDelimiter(String pattern)	设置自定义分隔符
reset()	重置扫描器的状态
close()	关闭输入流，释放相关的系统资源

【例 3-12】 使用 Scanner 类分割字符串示例。

```
String str = "Hello, World, Java";
Scanner scanner = new Scanner(str);
scanner.useDelimiter(", ");
while (scanner.hasNext()) {
    String token = scanner.next();
    System.out.println(token);
}
```

3. StreamTokenizer 类

StreamTokenizer 类用于从字符流中逐个读取标记。可以将其视为更底层、更灵活的字符分析器，可以自定义分隔符和特殊字符。StreamTokenizer 类的常用方法如表 3.6 所示。

表 3.6　StreamTokenizer 类的常用方法

方　法　名	说　　明
resetSyntax()	重置语法设置，将所有字符都视为普通字符
eolIsSignificant(boolean flag)	设置行结束符的处理方式。当 flag 为 true 时，将行结束符视为一个标记。当 flag 为 false 时，行结束符被忽略
ordinaryChar(int ch)	将指定字符设置为普通字符，即不作为特殊字符处理
wordChars(int low,int hi)	将指定范围内的字符设置为单词字符，作为标识符的一部分
whitespaceChars(int low,int hi)	将指定范围内的字符设置为空白字符，即分隔标记的字符
quoteChar(int ch)	设置引号字符，用于定义引号中的内容作为字符串
commentChar(int ch)	设置注释字符，用于忽略注释内容
nextToken()	获取并返回下一个标记
toString()	将对象转换为字符串表示形式
pushBack()	将当前标记放回流中，使其可以再次被读取

【例 3-13】 使用 StreamTokenizer 类分割字符串示例。

```
StringReader reader = new StringReader("Hello 123 4.56");
StreamTokenizer tokenizer = new StreamTokenizer(reader);
while (tokenizer.nextToken() != StreamTokenizer.TT_EOF) {
    if (tokenizer.ttype == StreamTokenizer.TT_WORD) {
        //处理单词类型的标记
        System.out.println(tokenizer.sval);
    } else if (tokenizer.ttype == StreamTokenizer.TT_NUMBER) {
        //处理数字类型的标记
        System.out.println(tokenizer.nval);
    }
}
```

上述代码从字符串"Hello 123 4.56"中逐个读取标记，并根据类型进行不同的处理。

以上字符分析器可以根据具体的需求选择使用。它们提供了易于操作的方法和灵活的配置选项，可以将输入字符串按照特定方式进行解析，并提取出所需的标记。无论是简单的字符串分割还是更复杂的数据解析，这些字符分析器都能很好地满足需求。

3.2.4 回文字符串

在 Java 中，回文字符串是指正读和反读都相同的字符串。换句话说，从左到右和从右到左读取这个字符串时，得到的结果是相同的。例如，"level"、"madam"和"racecar"都是回文字符串，因为它们正着读和倒着读都得到相同的结果。判断一个字符串是否是回文字符串可以通过比较字符串的首尾字符是否相等来实现。通常的方法是使用双指针，一个指针从字符串的开头向右移动，另一个指针从字符串的末尾向左移动，同时比较对应位置的字符是否相等。如果比较过程中发现不相等的字符，则该字符串不是回文字符串；如果一直比较到两个指针相遇仍然没有出现不相等的字符，则该字符串是回文字符串。

【例 3-14】 判断一个字符串是否为回文字符串示例。

```
public class PalindromeString {
    public static boolean isPalindrome(String str) {
        //去除字符串中的非字母和非数字字符，并将所有字符转换为小写
        String processedStr = str.toLowerCase().replaceAll("[^a-zA-Z0-9]", "");
        int left = 0;                                          //左指针
        int right = processedStr.length() - 1;                 //右指针
        while (left < right) {
            if (processedStr.charAt(left) != processedStr.charAt(right)) {
                return false;          //如果左右指针所指字符不相等，则不是回文字符串
            }
            left++;
            right--;
        }
        return true;              //左右指针交叉时表示所有字符都已比较完毕，是回文字符串
    }
    public static void main(String[] args) {
```

```
        String str = "A man, a plan, a canal: Panama";
        boolean isPalindrome = isPalindrome(str);
        System.out.println(isPalindrome);      //输出结果: true
    }
}
```

上述代码中，定义了一个静态方法 isPalindrome()，该方法接收一个字符串作为输入，并返回一个布尔值来表示该字符串是否为回文字符串。

首先，通过使用 toLowerCase()方法将字符串中的所有字符转换为小写，并使用 replaceAll()方法去除非字母和非数字字符。然后，使用两个指针 left 和 right 分别指向字符串的开头和末尾，逐个比较字符是否相等。如果左右指针所指字符不相等，则直接返回 false 表示不是回文字符串。最后，当左右指针交叉时，表示所有字符都已比较完毕，此时返回 true 表示是回文字符串。

3.2.5　正则匹配

Java 的正则表达式是一种用于字符串匹配和操作的强大工具，通过使用正则表达式，可以方便地进行字符串模式搜索、替换和提取等操作。它使用 java.util.regex 包中的类和方法来实现，可以满足各种匹配需求。

1. 正则表达式示例

下面是正则表达式的一些示例。

(1) 匹配邮箱地址：

```
"[a-zA-Z0-9]+@[a-zA-Z0-9]+\\.[a-zA-Z]{2,}"
```

这个正则表达式用于匹配邮箱地址，其中[]表示字符组，匹配其中任意一个字符，[a-zA-Z0-9]表示匹配任意一个字母或数字，+表示前面的字符可以出现一次或多次，@匹配邮箱地址中的“@”符号，\\.表示匹配邮件地址中的“.”符号，由于“.”是正则表达式的特殊字符，需要使用“\.”进行转义，[a-zA-Z]{2,}表示匹配至少两个连续的字母。

(2) 提取手机号码：

```
\\d{11}
```

这个正则表达式用于提取手机号码，其中\\d 表示匹配任意一个数字字符，{11}表示前面的字符必须出现 11 次。

(3) 判断字符串是否包含特定单词：

```
"\\bapple\\b"
```

这个正则表达式用于判断字符串是否包含特定单词，其中\\b 表示单词的边界，用于确保只匹配整个单词而不是部分单词，apple 表示要匹配的单词。

2. 正则表达式语法

正则表达式语法如表 3.7 所示。

表 3.7　正则表达式语法

符　号	说　　明
.	匹配除了换行符以外的任意字符。例如：a.b 可以匹配"aab"、"acb"、"axb"等
[]	字符组，匹配方括号中的任意字符。例如：[abc]可以匹配"a"、"b"、"c"中的任意一个字符
[^]	否定字符组，匹配不在方括号中的任意字符。例如：[^abc]可以匹配除了"a"、"b"、"c"以外的任意字符
*	匹配前面的字符零次或多次。例如：ab * c 可以匹配"ac"、"abc"、"abbc"等
+	匹配前面的字符一次或多次。例如：ab+c 可以匹配"abc"、"abbc"、"abbbc"等
?	匹配前面的字符零次或一次。例如：ab?c 可以匹配"ac"、"abc"
{n}	匹配前面的字符恰好出现 *n* 次。例如：a{3}可以匹配"aaa"
{n,}	匹配前面的字符至少出现 *n* 次。例如：a{2,}可以匹配"aa"、"aaa"、"aaaa"等
{n,m}	匹配前面的字符至少出现 *n* 次，最多出现 *m* 次。例如：a{2,4}可以匹配 "aa"、"aaa"、"aaaa"
()	捕获组，用于分组并捕获匹配的内容。例如：(abc)+可以匹配 "abc"、"abcabc" 等，且可以通过捕获组提取匹配的内容

3. 正则表达式工具类

正则表达式工具类如下。

（1）Pattern 类：使用 Pattern 类的 compile()方法可以将一个正则表达式编译成 Pattern 对象。Pattern 对象可以用于创建 Matcher 对象，用于进行匹配操作。

（2）Matcher 类：Matcher 对象是对输入字符串进行解释和匹配操作的发动机。与 Pattern 类一样，Matcher 也没有公共构造方法，需要调用 Pattern 对象的 matcher()方法来获得一个 Matcher 对象。

【例 3-15】　使用正则表达式在字符串中查找匹配的子序列示例。

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class PatternExample {
    public static void main(String[] args) {
        //定义正则表达式,匹配一个或多个 "a" 后面跟着一个 "b"
        String regex = "a*b";
        //将正则表达式编译成 Pattern 对象
        Pattern pattern = Pattern.compile(regex);
        //创建 Matcher 对象,在指定字符串中匹配正则表达式
        Matcher matcher = pattern.matcher("aabfooaabfooabfoob");
        //查找与正则表达式匹配的子序列
        while (matcher.find()) {        //循环查找匹配的子序列
            //获取匹配到的子序列的值
            System.out.println("匹配到的子序列: " + matcher.group());
            //获取匹配开始的索引
            System.out.println("匹配开始的索引: " + matcher.start());
            //获取匹配结束的索引
```

```
                System.out.println("匹配结束的索引: " + matcher.end());
            }
        }
    }
```

以上代码的作用是使用正则表达式在字符串"aabfooaabfooabfoob"中查找匹配的子序列，并输出匹配结果的信息。

3.3　集　合　类

Java 集合类是 Java 编程语言中的一组数据结构和算法的实现，用于存储和操作一组对象。它们提供了一种方便和高效地管理和操作数据的方式。Java 集合类框架包括了以下几个核心接口。

(1) List 接口是有序的可重复集合，它允许元素重复并按照插入顺序排序。常见的实现类有 ArrayList 和 LinkedList。List 接口提供了根据索引访问元素、添加、删除和遍历等方法，适用于需要根据位置来操作元素的场景。

(2) Queue 接口是一种队列，它存储一组元素并提供了队列操作(先进先出)的功能。常见的实现类有 LinkedList 和 PriorityQueue。Queue 接口提供了入队、出队、检索队首元素等方法，适用于需要按照特定顺序处理元素的场景，如任务调度、消息处理等。

(3) Set 接口是无序的不可重复集合，它保证元素的唯一性。常见的实现类有 HashSet 和 TreeSet。Set 接口不按照特定顺序存储元素，提供了添加、删除、查找和遍历等方法，适用于需要保证数据唯一性的场景。

(4) Map 接口是键值对的映射，它存储一组具有唯一键的值。常见的实现类有 HashMap 和 TreeMap。Map 接口通过键来查找对应的值，提供了键值对的操作方法，如插入、删除、查找和遍历等。适用于需要根据键来操作数据的场景，如字典、缓存等。

这些接口提供了许多功能丰富的实现类，使得开发人员可以选择最适合自己需求的集合类。集合类提供了各种方法来添加、删除、查找和遍历元素，并提供了排序、过滤等功能。图 3.1 是 Java 主要集合类的继承结构图。

Java 集合类具有以下特点。

(1) 动态调整大小：集合类可以根据需要自动扩展或缩小容量，无须手动管理内存。

(2) 类型安全：Java 的泛型机制确保集合类中存储的对象类型的安全性，在编译时进行类型检查。

(3) 高效性能：集合类经过优化，提供了高效的数据访问和操作方法，适用于处理大量数据。

(4) 线程安全：Java 提供了多线程安全的集合类，例如 Vector 和 Hashtable，也可以使用同步包装器将非线程安全的集合类变为线程安全。

Java 集合类是 Java 编程中非常重要和常用的部分，它们为开发人员提供了丰富的数据结构和算法，使得处理和操作数据变得更加方便和高效。下面分别介绍以上 4 种接口的实现类及其使用方法。