第 5 章

字　符　串

在各种编程语言当中，除了各种数值类型的数据外，字符串类型的数据是最为常见的。在不同的编程语言当中，对于字符串的界定方式也是不一样的，但是不论在哪一种编程语言当中，对字符串进行操作的各种 API 都是相当丰富的，在这些 API 中，又以字符串的匹配最为常用，并且其实现算法也有很多种，所以在这一章中将以字符串的一些常用基本概念及字符串在 Java 中的相关实现方式为基础，引出字符串匹配的各种算法并进行重点讲解。

本章内容的思维导图如图 5-1 所示。

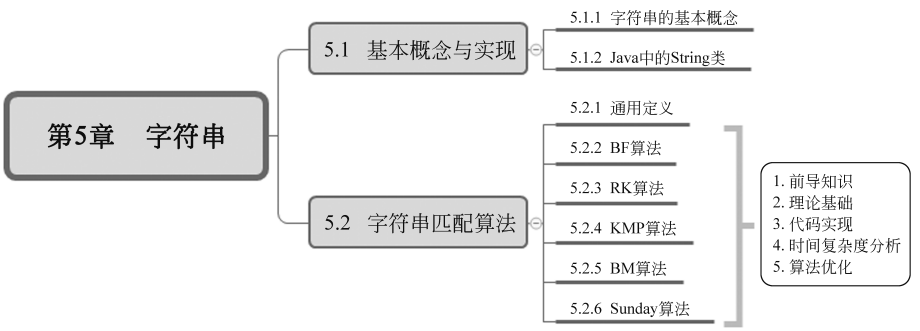

图 5-1　字符串章节思维导图

5.1　基本概念与实现

5.1.1　字符串的基本概念

字符串指的是通过 0 到多个字符构成的有限序列。

在不同的编程语言当中对字符串的界定方式有所不同，例如在 Java 语言中使用一组单引号表示单个字符，使用一组双引号表示一个字符串，而在 Python 语言或者 JavaScript 语言中并不存在单个字符的概念，使用一组双引号或者单引号都可以表示一个字符串。因为本书主要使用 Java 语言作为案例实现语言，所以在此唯一使用双引号作为字符串的界定符号。下面给出一个字符串的表示案例：

S="HelloWorld"

在上述案例中，S 表示字符串的名称，"HelloWorld"表示字符串的内容，双引号中字符的数量为字符串的长度。当字符串的长度为 0 时称为空串，使用符号∅表示，当字符串的长度大于 0 且仅由空格构成时称为空格串。

因为字符串是由 0 到多个字符构成的有限序列，所以大部分编程语言支持按照下标获取字符串中的某个字符。为了方便说明，本书使用“字符串名称[下标]”的方式描述单个字符在字符串中的位置，并以 0 作为字符串中字符的起始下标。例如在上述案例中，S[0]表示的是字符串 S 中起点位置的字符'H'；S[1]表示的就是字符串 S 中第 2 个字符'e'等。

一个字符串中从任意位置开始向后连续多个字符构成的子序列称为这个字符串的子串。为了方便表示，本书使用“字符串名称[子串起始字符下标，子串终点字符下标]”的方式描述一个字符串的子串。例如在上述案例中，S[0,2]表示字符串 S 中由字符 S[0]、S[1]、S[2]构成的子串，取值为 S[0,2]＝"Hel"；S[2,5]表示字符串 S 中由字符 S[2]、S[3]、S[4]、S[5]构成的子串，取值为 S[2,5]＝"lloW"等。很明显，任意子串的长度都不可能大于其所在字符串的长度。特例，空串∅被认为是任意字符串的子串。

当两个字符串 S 与 S′的长度相等且 S 与 S′在各个对应下标位置上的字符取值也相等时，称两个字符串 S 与 S′是相等的，表示方式为 S ＝＝ S′。

5.1.2　Java 中的 String 类

在 Java 语言中，与字符串相关的常见接口有 java.lang.CharSequence，该接口的常见实现类有 java.lang.String、java.lang.StringBuffer、java.lang.StringBuilder 等，其中最常用到的类为 String 类型，而 StringBuilder 与 StringBuffer 则多见于大规模字符串拼接与多线程的场景中，所以下面以 Java 语言(Java 8 版本)中的 String 类型为例对字符串类型对象在内存中的存储方式及一些常用 API 进行说明。

1. 字符串的存储

在本章开篇已经说明过字符串的本质是由字符构成的序列，所以参考第 2 章的内容不难想到，通过对数组结构或者链表结构进行封装都可以实现字符串类型。在 Java 语言中 String 类型采用字符数组的形式对字符串进行保存，其部分源码如下：

```
/**
 Java 中的字符串类型
 */
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /** The value is used for character storage. */
    private final char value[];

}
```

此外，在 Java 语言中还支持通过多种方式实例化 String 类型的对象。下面列举几种常

用的 String 类对象实例化方式，代码如下：

```
/**
 Chapter5_String
 com.ds.matching
 TestString.java
 Java 语言中 String 类型测试类
 */
public class TestString {

    //通过不同方式实例化 String 类的对象
    public void instanceString() {

        //1.对字符串变量 str1 直接进行赋值
        String str1 = "HelloWorld";
        System.out.println(str1);                //HelloWorld

        /*
         2.通过复制一个已经存在的字符串 str1 的内容
         为另一个新实例化的字符串 str2 进行赋值
         */
        String str2 = new String(str1);
        System.out.println(str2);                //HelloWorld

        //3.使用无参构造器实例化一个字符串对象，得到的结果为一个空串
        String str3 = new String();
        System.out.println(str3);                //输出一个空串

        //4.以字符数组作为内容实例化一个字符串对象 str4
        char[] chars = new char[] {'a', 'b', 'c', '1', '2', '3'};
        String str4 = new String(chars);
        System.out.println(str4);                //abc123

    }

}
```

在实际开发过程中，在同一程序内部有极大的可能出现许多等值的字符串。为了节省内存开销，Java 语言的 String 类还设计了字符串常量池这一结构，通过实现享元模式使使用相同内容的字符串引用变量之间可以指向同一个对象的内存地址。对于字符串的常量池结构，因为涉及不同版本 Java 语言之间的诸多特性及很多与数据结构相关度不大的知识点，所以在此不做过多介绍，有兴趣的读者可以自行查阅资料进行学习。

2. 字符串的比较

在 Java 语言的 String 类型中提供了相当丰富的方法，这些方法可供使用者调用，它们的具体使用方式均可在 Java 语言官方的帮助文档中查找到。

Java 语言中 String 类型下的常用方法大多数是见名知意、容易理解的，但是唯独用于

字符串之间大小比较的 compareTo()方法的含义不甚明确。换一种说法就是字符串之间是如何比较大小的，又是如何将比较结果以整数类型进行表示的呢？对于上述问题首先给出 String 类中 compareTo()方法的源码：

```
/**
 Java 中的字符串类型
 */
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    //用于比较方法调用串与参数字符串之间大小的方法
    public int compareTo(String anotherString) {
        int len1 = value.length;
        int len2 = anotherString.value.length;
        int lim = Math.min(len1, len2);
        char v1[] = value;
        char v2[] = anotherString.value;

        int k = 0;
        while (k < lim) {
            char c1 = v1[k];
            char c2 = v2[k];
            if (c1 != c2) {
                return c1 - c2;
            }
            k++;
        }
        return len1 - len2;
    }

}
```

通过分析上述源码可知，在 compareTo()方法内部首先获取了方法调用串和参数字符串的长度并分别记为 len1 和 len2，以及两个字符串中用于保存内容的字符数组并分别记为 v1 和 v2。将 len1 与 len2 中较短者记为 lim，以此作为循环控制变量 k 的边界，通过循环对两字符串中的字符 v1[k]与 v2[k]逐位进行比较。当 v1[k] != v2[k]时，返回 v1[k]－v2[k]的取值；若在 $k \in [0, \text{lim}-1]$范围内两字符串的内容全部相同，即一个字符串构成另一个字符串前缀的情况下，则返回两字符串长度的差值。

上述代码之所以可以将两个字符串中任意位置上的字符 v1[k]与 v2[k]进行大小比较及差值运算，是因为在编程语言中通常将字符类型变量的取值以整数值的方式进行存储，所以在对字符 v1[k]与 v2[k]进行大小比较及差值运算时，实际上计算机操作的是这两个字符所对应的整数值。

上述这种将字符符号表示为整数值的方式相当于为一个字符类型变量所能够表示的所有字符符号按照某种方式进行了编号，一个字符符号的编号称为这个字符的编码，而对字符

符号进行编码的不同方式也就对应了不同的编码字符集。在编程语言中常见的编码字符集有 ASCII、UTF-8、UTF-16、GBK、GB 2312、ISO-8859-1 等。有些编码字符集之间对于相同字符符号的编码是一致的，例如在 ASCII 编码字符集及 UTF-8 编码字符集当中字符'A'的编码都是 65 等，但是在某些编码字符集之间，因为编码的字符符号之间存在一些不重叠的内容，所以即使是取值相同的编码在这些字符集之间也可能表示不同的字符符号。例如在 UTF-8 编码字符集中，字符“中”的编码为 20013，但是这一编码取值在不能表示中文的 ISO-8859-1 编码字符集中表示的是其他非中文字符。当字符串在不同设备之间通过网络或者其他介质进行传输时，本质上传输的都是字符串中各个字符的编码取值。但如果在接收一个字符串时并未按照其发送时原本使用的编码字符集进行解码，就会出现在实际开发中十分常见的“乱码”问题。

5.2 字符串匹配算法

在对字符串进行各种操作的相关 API 中字符串匹配操作的使用率相对较高。在 Java 的 String 类型中，对于字符串的匹配操作提供了 indexOf()和 lastIndexOf()两大类方法，分别用于在调用方法的当前字符串当中查找作为参数的子串在其中首次出现和最后一次出现的起始下标。不难想到的是，这两种方法在诸如用于判断当前字符串是否包含参数字符串的 contains()方法、进行字符串拆分的 split()方法等诸多其他 API 的内部也会发挥作用，因此，一个高效率的字符串匹配算法对于所有与之相关的字符串操作来讲意义重大。下面开始对一些常见的字符串匹配算法的思想及实现进行讲解。

5.2.1 通用定义

在开始对各种字符串匹配算法进行讲解之前，首先对这些算法中的通用定义进行说明。

在字符串匹配算法中通常存在两个进行操作的字符串，即主串（或源串）S 与模式串（或目标串）T。字符串匹配算法的目的是在主串 S 中找出与模式串 T 完全相同的子串部分，或者确定在主串 S 中不存在与模式串 T 等值的子串部分。根据上述定义可知，如果主串 S 中至少存在一处与模式串 T 相同的子串则算法匹配成功，此时返回这一子串首字母在主串 S 中出现位置的下标；反之则匹配失败，此时通常返回 -1 作为算法的运行结果。显然，当模式串 T 的长度大于主串 S 的长度时匹配算法一定是失败的。

下面根据上述理论和定义对几种不同的字符串匹配算法分别进行讲解。

5.2.2 BF 算法

BF 算法的全称为 Brute Force 算法，即暴力匹配算法。BF 算法是所有字符串匹配算法中最基础、最易懂但同时也是效率最低的一种。下面开始对 BF 算法的相关内容进行讲解。

1. 理论基础

BF 算法的基本思想是将主串 S 中的每位字符都作为起始位，与模式串 T 的首位对齐

后逐位向后进行比较。当在某一位上的字符比较失败时折返回模式串 T 的首位，并在将模式串 T 的首位字符与主串 S 中的下一位字符重新对齐后重复进行比较操作。下面以在长度为 n 的主串 S 中查找长度为 m 的模式串 T 的过程为例对 BF 算法的执行流程进行说明。

步骤 1：给出长度为 n 的主串 S 与长度为 m 的模式串 $T(m \leqslant n)$。分别创建整型变量 i 和 j 表示当前正在进行比较的字符在主串 S 与模式串 T 中的下标位置。初始状态下变量 i 与 j 的取值均为 0，即表示将模式串 T 与主串 S 在起始位置对齐。

步骤 2：对 $S[i]$与 $T[j]$的取值进行比较。在比较过程中：①如果 $S[i] == T[j]$，则表示这一位字符匹配成功，执行 $i++$与 $j++$操作，继续对下一位字符进行比较；②如果 $S[i] != T[j]$，则表示这一位字符匹配失败，将变量 i 赋值为 $i-j+1$，将变量 j 赋值为 0，即表示从主串 S 中本轮比较起点的下一位开始与模式串 T 的起点对齐，重新开始新一轮的比较。

步骤 3：创建循环，在保证变量 i 与 j 的取值不越界的前提下重复执行步骤 2。

步骤 4：步骤 3 的循环结束即表示对主串 S 或者模式串 T 的遍历已经完成，需要对算法的返回值进行计算：①此时若 $j == m$，则表示循环中对模式串 T 中所有字符在主串 S 中的对比均已完成，即模式串 T 在主串 S 中完全匹配的子串已经找到，此时 $i-j$（或者 $i-m$）的取值即为模式串 T 的全等子串在主串 S 中首次出现时起点字符在主串 S 中的下标；②反之若该条件不成立，则表示对主串 S 中字符的遍历先一步结束，即主串 S 中不包含模式串 T 的全等子串，此时返回 -1 作为算法的结果。

BF 算法执行流程如图 5-2 所示。

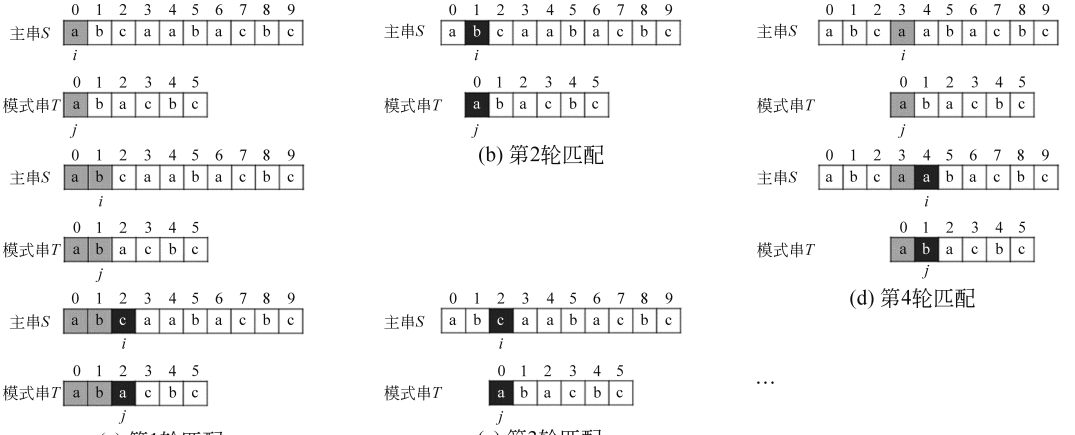

图 5-2　BF 算法执行流程

2. 代码实现

BF 算法实现的代码如下：

```
/**
 Chapter5_String
 com.ds.matching
```

```
 BruteForce.java
 BF算法
 */
public class BruteForce {

    /**
     BF算法代码实现
     S表示主串
     T表示模式串
     */
    public int bruteForce(String S, String T) {

        if(S == null || T == null || S.length() < T.length()) {
            return -1;
        }

        if(T.length() == 0) {
            return 0;
        }

        //1.创建算法所需各种控制变量并初始化
        int i = 0;                                    //正在比较字符在主串S中的下标
        int j = 0;                                    //正在比较字符在模式串T中的下标

        //3.通过循环重复步骤2的比较过程
        while(i < S.length() && j < T.length()) {

            //2.对S[i]与T[j]进行比较
            if(S.charAt(i) == T.charAt(j)) {
                //2.1当S[i] == T[j]时
                i++;
                j++;
            } else {
                //2.2当S[i] != T[j]时
                i = i-j+1;
                j = 0;
            }

        }

        //4.计算算法返回值
        if(j == T.length()) {
            return i-j;
        } else {
            return -1;
        }

    }

}
```

3. 时间复杂度分析

BF 算法的时间复杂度可以分为匹配成功与匹配失败两种情况进行说明。

对于给定的规模分别为 N 和 $M(N \geqslant M)$的主串与模式串来讲，在匹配成功的情况下算法的最好时间复杂度为 $O(M)$，即主串的前 M 个字符构成的子串恰好是模式串的等值子串，而在最坏的情况下，当在主串的前 $N-M$ 长度的部分中匹配模式串时恰好每轮匹配都在模式串的最后一位失败，此时，则需要经过 $N-M+1$ 轮，每轮比较 M 次操作才能找到位于主串最末端的模式串等值子串。此时算法的时间复杂度为 $O(NM)$。在匹配失败的情况下算法的最好时间复杂度为 $O(N)$，即主串中所有字符与模式串的起始字符均不相等，此时相当于对主串进行一轮遍历即可得到算法结果。同样地，在最坏的情况下，如果在主串中对模式串的每轮匹配都在模式串的最后一位失败，则此时算法的时间复杂度同样为 $O(NM)$。

对于匹配成功的平均时间复杂度可以做出如下计算。同样对于给定的规模分别为 N 和 $M(N \geqslant M)$的主串与模式串，其发生在主串中以下标为 $i(0 \leqslant i \leqslant N-M)$的字符为起点，与模式串匹配成功的各种情况概率相等，并且在之前的匹配过程中每轮匹配失败都发生在模式串起点的位置，则此时每种匹配成功情况的平均时间开销 $T(N, M)$可以表示为式(5-1)：

$$T(N,M) = M + \frac{1}{N-M}\sum_{i=0}^{N-M-1} i \tag{5-1}$$

在式(5-1)中，加号右侧表示前序匹配失败的平均字符比较次数，加号左侧表示最终匹配成功时所进行与模式串长度相等次数的字符比较。式(5-1)最终经过化简可得 BF 算法在匹配成功时的最好平均复杂度为 $O(N+M)$。

在最坏的情况下，即在之前的匹配过程中每轮匹配失败都发生在模式串的终点位置，则式(5-1)可以变形为式(5-2)：

$$T(N,M) = M + \frac{1}{N-M}\sum_{i=0}^{N-M-1} i \times M \tag{5-2}$$

最终式(5-2)经过化简可得 BF 算法在匹配成功时的最坏平均复杂度为 $O(NM)$。

上述方式同样适用于计算匹配失败时的平均时间复杂度，最终可得 BF 算法在匹配失败时的最好与最坏平均复杂度分别为 $O(N)$和 $O(NM)$。

4. 算法优化

在主串 S 中可能存在连续多个字符与模式串 T 的首字符不相等的情况。当出现这种情况时，原始的 BF 算法依然会重复地将这些连续字符逐个与模式串 T 的首字符进行比较，这会浪费很多时间，所以如果在一轮匹配开始前将这些与模式串 T 首字符不相等的连续字符在主串 S 当中跨过去，就能够降低算法整体的时间开销。

优化后的 BF 算法实现，代码如下：

```
/**
 Chapter5_String
 com.ds.matching
 BruteForce.java
```

```
 BF算法
 */
public class BruteForce {

    //BF算法优化代码实现
    public int bruteForceOptimized(String S, String T) {

        if(S == null || T == null || S.length() < T.length()) {
            return -1;
        }

        if(T.length() == 0) {
            return 0;
        }

        int i = 0;
        int j = 0;
        /*
         为了避免变量 i 在 S[i]与 T[j]的比较过程中因为 i++导致 S[i] != T[0]的情况持续
         发生单独为优化步骤创建一个变量 k
         */
        int k = 0;

        while(i < S.length() && j < T.length()) {

            /*
             优化部分:
             使变量 i 的取值跨过主串 S
             连续与模式串 T[0]不相等的部分
             */
            while(k < S.length() && S.charAt(k) != T.charAt(0)) {
                k++;
                i = k;
            }

            if(i < S.length()) {
                if(S.charAt(i) == T.charAt(j)) {
                    i++;
                    j++;
                } else {
                    i = i-j+1;
                    //匹配失败时,将 k 的取值重置为 i
                    k = i;
                    j = 0;
                }
            }

        }
```

```
            if(j == T.length()) {
                return i-j;
            } else {
                return -1;
            }

        }

    }
```

实际上在 Java 的 String 类当中用于子串查找的 indexOf(String str)方法就是通过上述思路实现的，只是在代码实现方式上略有不同而已。

5.2.3　RK 算法

根据 5.1.2 节讲解的内容可知，在编程语言的底层依然通过整数实现对字符的存储，所以从这一角度来看，一个字符串就相当于一组整数的集合，而字符串匹配算法实际上就是两组整数集合之间的匹配操作，并且当主串和模式串中均存在多个字符时，每轮匹配算法的执行都相当于对多对整数之间的等值关系进行判断的过程。那么如果能够在字符串匹配算法当中将主串中所有与模式串等长的子串都根据其每位字符的取值转换成为一个整数特征值，并将这些子串的特征值与模式串的特征值进行比较，则每轮匹配算法就从多对整数之间的等值比较操作简化成为一对整数之间的等值比较操作。显然比较一对整数之间的等值关系要比比较多对整数之间的等值关系在运行速度上快得多，而这种将字符串之间的比较简化为字符串之间特征值比较的思路即为 RK 算法的核心思想。下面开始对 RK 算法的相关内容进行讲解。

1. 字符串的 hashCode

hashCode 可以被称为哈希值、散列值、哈希编码等，是在编程语言的内存机制当中用于对对象进行唯一区分的一种整数数值，可以简单地认为每个对象的 hashCode 取值就是与之绑定的一个“特征编码”。

在 Java 的内存机制当中，每个存在于内存中的对象都会被赋予一个与之对应的 hashCode，并且这个 hashCode 取值可以通过继承自 Object 父类中的 hashCode()方法进行计算与获取。在默认情况下，对象的 hashCode 取值与对象的内存地址相关，此时任意两个对象之间的 hashCode 取值均不相等，但是在 Java 语言中允许在任何继承自 Object 最高父类的子类当中(实际上 Java 中的所有类型都直接或间接地继承自 Object 类)对 hashCode()方法进行重写，从而达到根据对象中各个属性的取值生成并获取 hashCode 的目的。

通过重写之后的 hashCode()方法所得对象的 hashCode 取值需要确保如下两个性质的成立：①当两个相同类型对象的各个对应属性的取值完全相同时，这两个对象的 hashCode 取值也应该完全相同；②当两个相同类型的对象在某些对应属性上的取值有所不同时应尽可能地保证这两个对象之间的 hashCode 取值不相同。例如在自定义的 Person 类型中定义