第四章：串

本章主要讨论作为一种特殊线性表的“串”（String）数据类型，它在非数值处理领域（如文本编辑、编译器、信息检索等）中应用极为广泛。

串

4.1 串的抽象数据类型定义

串（String） 是由 n（$n \geq 0$）个字符组成的有限序列。逻辑结构上，它类似于线性表，但其数据对象被限定为字符集。<u>与线性表的主要区别在于，串的操作通常是针对整个串或子串，而不是单个元素。</u>

ADT String 的基本操作

- `StrAssign(&T, chars)`：赋值操作。将一个字符串常量 `chars` 赋给串 `T`。
- `StrCopy(&T, S)`：复制操作。由串 `S` 复制得到串 `T`。
- `StrEmpty(S)`：判空操作。若 `S` 为空串 `""`，则返回真。
- `StrCompare(S, T)`：比较操作。按字典序比较串 `S` 和 `T`。若 `S>T`，返回值>0；若 `S<T`，返回值<0；若 `S=T`，返回值=0。
- `StrLength(S)`：求长操作。返回串 `S` 的元素个数，即串的长度。
- `Concat(&T, S1, S2)`：联接操作。用 `T` 返回由 `S1` 和 `S2` 连接而成的新串。
- `SubString(&Sub, S, pos, len)`：求子串操作。用 `Sub` 返回串 `S` 从第 `pos` 个字符起，长度为 `len` 的子串。`pos` 和 `len` 的取值必须合法，即 $1 \leq pos \leq \mathrm{StrLength}(S)$ 且 $0 \leq len \leq \mathrm{StrLength}(S) - pos + 1$。
- `Index(S, T, pos)`：定位操作（查找子串）。从主串 `S` 的第 `pos` 个字符起，查找模式串 `T` 第一次出现的位置。若找到，返回其在 `S` 中的起始位置（位序）；否则返回 0。
- `Replace(&S, T, V)`：替换操作。用串 `V` 替换主串 `S` 中所有与模式串 `T` 相等且不重叠的子串。
- `StrInsert(&S, pos, T)`：插入操作。在主串 `S` 的第 `pos` 个字符前插入串 `T`。
- `StrDelete(&S, pos, len)`：删除操作。从主串 `S` 中删除从第 `pos` 个字符起，长度为 `len` 的子串。
- `ClearString(&S)`：清空操作。将 `S` 置为空串。
- `DestroyString(&S)`：销毁操作。释放串 `S` 占用的存储空间。

最小操作子集

在上述操作中，`StrAssign`，`StrCopy`，`StrCompare`，`StrLength`，`Concat`，`SubString` 构成了串类型的**最小操作子集**。其他操作（如 `Index`，`Replace` 等）都可以基于这个子集来实现。

- **用最小集实现 `Index` 函数**：
  其基本思想是，在主串 `S` 中从 `pos` 位置开始，逐一截取与模式串 `T` 等长的子串，并与 `T` 进行比较，直到找到匹配或搜索完整个主串。

```
// 假设已存在 String 类型和最小操作子集函数
int Index(String S, String T, int pos) {
    // T为非空串。若主串S中第pos个字符之后存在与 T相等的子串，
    // 则返回第一个这样的子串在S中的位置，否则返回0
    if (pos > 0) {
```

```
        int n = StrLength(S);
        int m = StrLength(T);
        int i = pos;
        String sub;
        InitString(sub); // 假设有初始化函数

        while (i <= n - m + 1) {
            SubString(sub, S, i, m);
            if (StrCompare(sub, T) != 0) {
                ++i;
            } else {
                DestroyString(sub); // 释放临时子串
                return i;
            }
        }
        DestroyString(sub);
    }
    return 0;  // S中不存在与T相等的子串
}
```

4.2 串的表示和实现

串的存储结构主要有三种方式。

1. 定长顺序存储表示（静态串）

这类似于 Pascal 语言中的字符串，用一个定长的字符数组来存储串，通常数组的第 0 个单元存放串的当前长度 。

特性	Pascal 字符串 (P-string)	C 字符串 (C-string)
长度存储	第一个字节存储长度	以空字符 `\0` 作为字符串结束标志
最大长度	通常为 255 (如果长度存储在单字节)	理论上无限制，受限于内存
字符串读取	读取第一个字节即可知道长度，然后按长度读取	必须逐个字符读取直到遇到 `\0`
获取长度	O(1) 时间复杂度 (直接读取第一个字节)	O(N) 时间复杂度 (需要遍历整个字符串，N 为长度)
内存使用	长度字节 + 实际字符	实际字符 + 终止空字符 `\0`
字符串操作	长度已知，操作 (如拼接) 可能更直接	依赖 `\0` 寻找结束，操作可能需要额外函数
溢出风险	写入超过最大长度时可能覆盖相邻内存 (如果有固定缓冲区)	写入超过缓冲区时容易发生缓冲区溢出，覆盖 `\0` 导致更严重问题

- **定义**:

```
#define MAXSTRLEN 255
typedef unsigned char SString[MAXSTRLEN + 1]; // 0号单元存放串的长度
```

- **特点**:
  - 实现简单，操作效率高。
  - 长度固定，如果串操作的结果超出了 `MAXSTRLEN`，会发生**截断**，丢失一部分数据 。

2. 堆分配存储表示（动态串）

这是 C 语言等现代编程语言中常用的方式，串的存储空间在程序运行时动态分配和释放，通常存放在**堆（Heap）**内存中 。

- **定义**:

```
typedef struct {
    char *ch;      // 指向动态分配的存储区的指针.动态分配区的地址是连续的，就可以动态调整长度了
    int length;    // 串的长度
} HString;
```

- **特点**:
  - 存储空间按需分配，灵活，不会有截断问题 。
  - 需要频繁地使用 `malloc`，`realloc`，`free` 等函数管理内存，会产生一些额外开销。
- **示例：`Concat` 操作实现**

```
// 假设 HString 已定义
int Concat(HString *T, HString S1, HString S2) {
    if (T->ch) {
        free(T->ch); // 释放T原有的旧空间
    }

    T->length = S1.length + S2.length;
    T->ch = (char *)malloc(T->length * sizeof(char));
    if (!T->ch) {
        // exit(OVERFLOW); 内存分配失败
        return 0; // ERROR
    }

    // 复制S1和S2到T
    for (int i = 0; i < S1.length; i++) T->ch[i] = S1.ch[i];
    for (int i = 0; i < S2.length; i++) T->ch[S1.length + i] = S2.ch[i];

    return 1; // OK
}
```

3. 串的块链存储表示

当串非常长时，顺序存储可能无法找到足够大的连续空间。此时可以用链表来存储，每个链表节点存放一个“块”（一个子串），而不是单个字符，以提高存储密度 。

- **定义**:

```
#define CHUNKSIZE 80 // 可由用户定义的块大小

typedef struct Chunk {
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;

typedef struct {
    Chunk *head, *tail; // 串的头和尾指针
    int curlen;         // 串的当前长度
} LString;
```

- **应用**: 适用于文本编辑器等需要频繁进行插入、删除操作的场景 。

4.3 串的模式匹配算法

模式匹配（或子串查找）是串操作中非常核心和重要的一个部分，即 `Index(S, T, pos)` 的实现 。

1. 简单（朴素/暴力）匹配算法

这是最直观的算法。

- **思想**:
  从主串 `S` 的 `pos` 位置开始，将模式串 `T` 与 `S` 中等长的子串进行逐一字符比较。
  - 如果匹配成功，返回起始位置。
  - 如果中途失配，主串的比较指针 `i` **回溯到下一个起始位置**（即 `i = i - j + 2`，其中 `j` 是模式串中失配字符的位置），模式串的指针 `j` 回到 1，重新开始新一轮匹配 。
- **时间复杂度**: 最坏情况下为 $O(n \times m)$，其中 `n` 是主串长度，`m` 是模式串长度。

2. KMP 算法 (Knuth-Morris-Pratt)

这是一个高效的模式匹配算法，其核心优点是**主串的指针 `i` 不回溯** 。

- **思想**:
  当 `S[i]` 与 `T[j]` 发生失配时，简单算法会将 `i` 回溯，`j` 置 1。KMP 算法利用已经匹配过的信息（`S[i-j+1...i-1] == T[1...j-1]`），通过移动模式串 `T`，让它的某个前缀对齐到主串已匹配部分的后缀上，从而避免了 `i` 的回溯。模式串应该移动多远，完全由模式串自身的结构决定，这就是 `next` 数组的作用 。
- **`next` 数组的定义**:
  `next[j] = k` 的含义是：当模式串的第 `j` 个字符 `T[j]` 与主串失配时，下一步应该用模式串的第 `k` 个字符 `T[k]` 去和主串当前字符进行比较 。`k` 的值是模式串 `T` 的子串 `T[1...j-1]` 中，最长的、相等的“前缀”和“后缀”的长度再加 1。
  - 规定 `next[1] = 0`。
- **KMP 匹配过程**:

```
int Index_KMP(SString S, SString T, int pos, int next[]) {
    int i = pos;
    int j = 1;
    while (i <= S[0] && j <= T[0]) {
        if (j == 0 || S[i] == T[j]) { // j=0是特殊情况，表示从T的第一个字符开始
            ++i;
            ++j;
        } else {
            j = next[j]; // j回溯，模式串右移，i不回溯
        }
    }

    if (j > T[0]) {
        return i - T[0]; // 匹配成功
    } else {
        return 0;
    }
}
```

  - **时间复杂度**: $O(n+m)$，其中 `n` 是主串长度，`m` 是模式串长度。

- `next` **数组的计算**:
  计算 `next` 数组本身也是一个“自己匹配自己”的过程。

```
void get_next(SString T, int next[]) {
    int i = 1;
    int j = 0;
    next[1] = 0;

    while (i < T[0]) {
        if (j == 0 || T[i] == T[j]) {
            ++i;
            ++j;
            next[i] = j;
        } else {
            j = next[j];
        }
    }
}
```

- **KMP 算法的改进 (`nextval`)**:
  在某些情况下（如 `T="aaaab"`），`next` 数组存在缺陷。当 `T[j]` 失配时，如果 `T[j] == T[next[j]]`，那么下一次比较也必然失配，这是一次无效的比较。
  改进的 `nextval` 数组通过在计算时增加一步判断来避免这种情况：如果 `T[i] == T[j]`，则 `nextval[i] = nextval[j]`，而不是简单地设为 `j`。

```
void get_nextval(SString T, int nextval[]) {
    int i = 1;
    int j = 0;
    nextval[1] = 0;

    while (i < T[0]) {
        if (j == 0 || T[i] == T[j]) {
```

```
            ++i;
            ++j;
            if (T[i] != T[j]) {
                nextval[i] = j;
            } else {
                nextval[i] = nextval[j];
            }
        } else {
            j = nextval[j]; // 注意这里是用nextval[j]来回溯
        }
    }
}
```

以下是根据PPT内容整理的串(字符串)章节的Markdown文档，包含所有技术细节和代码实现：

第四章 串

串的基本概念

- **串(String)**：有限字符序列，由双引号括起，如 `"a string"`
- **空串**：长度为0的串，用 `""` 表示
- **子串**：串中任意连续字符组成的子序列
- **主串**：包含子串的串
- **位置**：字符在序列中的序号（从1开始）
- **串相等**：长度相等且对应字符相同

串的抽象数据类型(ADT)

```
ADT String {
    数据对象:
        D = { a_i | a_i ∈ CharacterSet, i=1,2,...,n, n≥0 }
    数据关系:
        R1 = { <a_{i-1}, a_i> | a_{i-1}, a_i ∈ D, i=2,...,n }
    基本操作:
        StrAssign(&T, chars)     // 串赋值
        StrCopy(&T, S)           // 串复制
        DestroyString(&S)        // 销毁串
        StrEmpty(S)              // 判空
        StrCompare(S, T)         // 串比较
        StrLength(S)             // 求串长
        Concat(&T, S1, S2)       // 串连接
        SubString(&Sub, S, pos, len)  // 求子串
        Index(S, T, pos)         // 子串定位
        Replace(&S, T, V)        // 串替换
        StrInsert(&S, pos, T)    // 子串插入
        StrDelete(&S, pos, len) // 子串删除
        ClearString(&S)          // 清空串
} ADT String
```

关键操作说明

1. **StrCompare(S, T)**:
   - S > T → 返回值 > 0
   - S = T → 返回值 = 0
   - S < T → 返回值 < 0
   - 例：`StrCompare("data", "state") < 0`
2. **SubString(&Sub, S, pos, len)**:
   - 条件：`` `1 ≤ pos ≤ StrLength(S)` 且 `0 ≤ len ≤ StrLength(S)-pos+1` ``
     - pos就是1开始而非索引
   - 例：`SubString(sub, "commander", 4, 3)` → sub = "man"
3. **Index(S, T, pos)**:
   - 返回T在S中第 `` `pos `` 字符后第一次出现的位置
   - 例：S="abcaabcaaabc", T="bca"
     `Index(S,T,1)=2`，`Index(S,T,3)=6`，`Index(S,T,8)=0`

串的存储结构

1. 定长顺序存储

```
#define MAXSTRLEN 255  // 最大串长

typedef unsigned char SString[MAXSTRLEN + 1];
// 0号单元存放串长度
```

特点:

- 超过最大长度的串会被截断
- 基本操作基于字符序列复制

串联接示例:

```
Status Concat(SString S1, SString S2, SString *T) {
    int uncut = TRUE;

    if (S1[0] + S2[0] <= MAXSTRLEN) {  // 未截断
        for (int i = 1; i <= S1[0]; i++) (*T)[i] = S1[i];
        for (int i = 1; i <= S2[0]; i++) (*T)[S1[0]+i] = S2[i];
        (*T)[0] = S1[0] + S2[0];
    }
    else if (S1[0] < MAXSTRLEN) {  // 截断S2
        for (int i = 1; i <= S1[0]; i++) (*T)[i] = S1[i];
        for (int i = 1; i <= MAXSTRLEN - S1[0]; i++)
            (*T)[S1[0]+i] = S2[i];
        (*T)[0] = MAXSTRLEN;
        uncut = FALSE;
    }
    else {  // 仅取S1
        for (int i = 0; i <= MAXSTRLEN; i++) (*T)[i] = S1[i];
        uncut = FALSE;
    }
```

```
    return uncut;
}
```

2. 堆分配存储

```
typedef struct {
    char *ch;      // 串存储空间基址
    int length;    // 串长度
} HString;
```

特点:

- 动态分配存储空间
- C语言字符串采用此方式（以'\0'结尾）

基本操作实现:

```
// 串连接
Status Concat(HString *T, HString S1, HString S2) {
    if (T->ch) free(T->ch);  // 释放旧空间

    T->length = S1.length + S2.length;
    T->ch = (char*)malloc(T->length * sizeof(char));
    if (!T->ch) exit(OVERFLOW);

    for (int i = 0; i < S1.length; i++)
        T->ch[i] = S1.ch[i];
    for (int i = 0; i < S2.length; i++)
        T->ch[S1.length + i] = S2.ch[i];

    return OK;
}

// 求子串
Status SubString(HString *Sub, HString S, int pos, int len) {
    if (pos < 1 || pos > S.length || len < 0 || len > S.length - pos + 1)
        return ERROR;

    if (Sub->ch) free(Sub->ch);  // 释放旧空间

    if (len == 0) {  // 空子串
        Sub->ch = NULL;
        Sub->length = 0;
    } else {  // 非空子串
        Sub->ch = (char*)malloc(len * sizeof(char));
        for (int i = 0; i < len; i++)
            Sub->ch[i] = S.ch[pos-1+i];
        Sub->length = len;
    }
    return OK;
}
```

3. 块链存储

```c
#define CHUNKSIZE 80  // 块大小

typedef struct Chunk {
    char ch[CHUNKSIZE];
    struct Chunk *next;
} Chunk;

typedef struct {
    Chunk *head, *tail;  // 头尾指针
    int curlen;          // 当前长度
} LString;
```

特点:

- `存储密度 = 数据元素所占位 / 实际分配位`
- 适用于文本编辑系统（每行作为一个块）

串的模式匹配算法

问题定义

`Index(S, T, pos)`：在S中从`pos`位置开始查找T出现的位置

1. 简单算法(BF)

时间复杂度：$O(n \times m)$

```c
int Index_BF(SString S, SString T, int pos) {
    int i = pos, j = 1;
    while (i <= S[0] && j <= T[0]) {    //S[0]和T[0]存储字符串的实际长度。是Pascal字符串
        if (S[i] == T[j]) {  // 匹配成功
            i++; j++;
        } else {  // 匹配失败
            i = i - j + 2;  // i回溯
            j = 1;          // j复位
        }
    }
    return (j > T[0]) ? (i - T[0]) : 0;
}
```

2. 首尾匹配算法

改进思路:

1. 先比较模式串首字符
2. 再比较模式串尾字符
3. 最后比较中间字符

```c
int Index_FL(SString S, SString T, int pos) {
    int sLen = S[0], tLen = T[0];
```

```
    char firstChar = T[1], lastChar = T[tLen];
    int i = pos;

    while (i <= sLen - tLen + 1) {
        if (S[i] != firstChar) {
            i++;
        } else if (S[i + tLen - 1] != lastChar) {
            i++;
        } else {  // 首尾匹配，检查中间字符
            int k = 1, j = 2;
            while (j < tLen && S[i+k] == T[j]) {
                k++; j++;
            }
            if (j == tLen) return i;
            else i++;
        }
    }
    return 0;
}
```

3. KMP算法

核心思想：利用部分匹配信息，避免主串指针回溯
时间复杂度：$O(n+m)$

- **思想**:
  当 `S[i]` 与 `T[j]` 发生失配时，简单算法会将 `i` 回溯，`j` 置 1。KMP 算法利用已经匹配过的信息（`S[i-j+1...i-1] == T[1...j-1]`），通过移动模式串 `T`，让它的某个前缀对齐到主串已匹配部分的后缀上，从而避免了 `i` 的回溯。模式串应该移动多远，完全由模式串自身的结构决定，这就是 `next` 数组的作用 。
- `next` **数组的定义**:
  `next[j] = k` 的含义是：当模式串的第 `j` 个字符 `T[j]` 与主串失配时，下一步应该用模式串的第 `k` 个字符 `T[k]` 去和主串当前字符进行比较 。`k` 的值是模式串 `T` 的子串 `T[1...j-1]` 中，最长的、相等的“前缀”和“后缀”的长度再加 1。
  - 规定 `next[1] = 0`。

(1) 算法实现

```
int Index_KMP(SString S, SString T, int pos, int next[]) {
    int i = pos, j = 1;
    while (i <= S[0] && j <= T[0]) {
        if (j == 0 || S[i] == T[j]) {  //j=0表示滑出匹配范围，s[i]=s[j]表示当前匹配，则后位继续匹配
            i++; j++;  // 继续比较后继字符
        } else {
            j = next[j];  // 将j更新为 next[j]，模式串整体向右滑动了 j - next[j] 个位置。之后继续匹配
        }
    }
    return (j > T[0]) ? (i - T[0]) : 0;
}
```

(2) Next函数计算

```
void get_next(SString T, int next[]) {
    int i = 1, j = 0;
    next[1] = 0;
    while (i < T[0]) {
        if (j == 0 || T[i] == T[j]) {
            i++; j++;
            next[i] = j;
        } else {
            j = next[j];
        }
    }
}
```

(3) Next函数示例

模式串	a	b	a	a	b	c	a	c
下标`j`	1	2	3	4	5	6	7	8
`next[j]`	0	1	1	2	2	3	1	2

(4) 改进的Next函数

解决特殊情况（如T="aaaab"）

```
void get_nextval(SString T, int nextval[]) {
    int i = 1, j = 0;
    nextval[1] = 0;
    while (i < T[0]) {
        if (j == 0 || T[i] == T[j]) {
            i++; j++;
            // 优化：避免相同字符多次比较
            nextval[i] = (T[i] != T[j]) ? j : nextval[j];
        } else {
            j = nextval[j];
        }
    }
}
```

KMP算法匹配过程示例

主串：`a c a b a a b a a b c a c a a b c`
模式串：`a b a a b c`
匹配过程：

1. 第一趟：i=2, j=2失败 → j=next[2]=1 `i: a c` `j: a b`
2. 第二趟：i=2, j=1失败 → j=next[1]=0 `i: c` `j: a`
3. 第三趟：i=8, j=6失败 → j=next[6]=3 `i: a b a a b a` `j: a b a a b c`
4. 第四趟：匹配成功 `i: a b a`(这三个其实未比)`a b c` `j: a b a a b c`

串操作应用

1. 定位函数Index实现

```
int Index(SString S, SString T, int pos) {
    int n = StrLength(S), m = StrLength(T);
    int i = pos;

    while (i <= n - m + 1) {
        SString sub;
        SubString(&sub, S, i, m);  // 获取子串
        if (StrCompare(sub, T) != 0) {
            i++;
        } else {
            return i;
        }
    }
    return 0;
}
```

2. 替换函数Replace实现

```
void Replace(HString *S, HString T, HString V) {
    int pos = 1, n = S->length, m = T.length;
    HString news;  // 新串
    StrAssign(&news, "");

    while (pos <= n - m + 1) {
        int i = Index(*S, T, pos);  // 查找T位置
        if (i == 0) break;

        // 提取T前面的子串
        HString prefix;
        SubString(&prefix, *S, pos, i - pos);
        Concat(&news, news, prefix);  // 连接前缀
        Concat(&news, news, V);       // 连接替换串
        pos = i + m;  // 更新位置
    }

    // 连接剩余部分
    HString suffix;
    SubString(&suffix, *S, pos, n - pos + 1);
    Concat(S, news, suffix);  // 更新S
}
```

本章要点

1. 熟悉串的7种基本操作定义及实现
2. 掌握串的三种存储结构：
    - 定长顺序存储
    - 堆分配存储
    - 块链存储
3. 深入理解KMP算法：

- 掌握next函数定义和计算
- 能手工计算next和nextval数组

4. 了解串操作的应用特点

> 作业：4.17, 4.28

补充说明

1. **存储结构选择**：
   - 定长顺序存储：适合长度固定的串
   - 堆分配存储：适合长度变化的串（C语言标准实现）
   - 块链存储：适合文本编辑系统
2. **KMP算法核心**：
   - 部分匹配值：$\mathrm{PM}[j] = \text{最长公共前后缀长度}$
   - next函数：$next[j] = \mathrm{PM}[j-1] + 1$
   - 优化nextval：避免相同字符重复比较
3. **串操作特点**：
   - 最小操作子集：赋值、复制、比较、求长、连接、求子串
   - 其他操作可通过基本操作实现