第5章

串

5.1 串的定义

5.1.1 串的基本概念

串，即字符串(String)，是由 $n(n\geqslant 0)$ 个字符组成的有限序列。串由双引号进行界定，引号之内的字符序列是串的值，可以是字母、数字或其他字符。长度为 0 的串称为空串，用 ϕ 表示。

串的相关概念如下。

(1) 子串：串中连续的 $m(0\leqslant m\leqslant n)$ 个字符组成的子序列。

(2) 主串：包含子串的串。

(3) 字符在主串中的位置：字符在串中的序号。

(4) 子串在主串中的位置：子串的第一个字符在主串中的位置。

串中数据之间呈线性关系，每个字符有前驱和后继。串与线性表之间的主要区别在于串的数据对象限定为字符集，其基本操作通常以子串为操作对象。

5.1.2 抽象数据类型定义

串的类主要描述数据和操作，其中，数据是一个字符数组，基本操作包括串的拼接、子串查找、串的长度、子串位置、子串匹配等，具体的抽象数据类型如下。

```
ADT String{
数据对象：D={a_i | a_i∈CharacterSet, i=1,2,…,n(n≥0)}
数据关系：S={<a_i-1, a_i> | a_i-1, a_i∈D(i=2,…,n)}
基本操作：
concat(String ss)——串的拼接。
substring(int pos,int len)——子串查找。
strcpy(String dest)——串的复制。
strins(int pos,String t)——串的插入。
getlength()——获取串的长度。
clearstr()——清空串。
compare(String t)——串的比较。
index(String t)——子串的位置。
}
```

根据串的抽象数据类型可以得到如下串的类。

```
#define MAXLENGTH 255                    //串允许的最大长度为 255

class String{
private:
    char s[MAXLENGTH];
    int length;
    void init(const char * ss);
public:
    String(){}
    String(const char * ss);
    String(char * ss, int start =0, int len =0);
    void operator =(const char * ss);
    String& concat(String ss);
    String substring(int pos,int len);
    String strcpy(int start, int len);
    String strcpy(int start);
    String strcpy();
    bool strins(int pos,String t);
    int getlength();
    int compare(String t);
    int index(String t);
    char getchar(int i);
    char * toarray();
};
```

5.2 串的实现

5.2.1 串的构造

String 类中提供了三个构造函数,包括无参构造函数、有参构造函数、带默认值的有参构造函数,这些不同的构造函数为后续操作的实现提供了便利。其中,无参构造函数已经实现,其他两个有参字符串的实现代码如下。

```
//初始化操作,该函数将会在多处被调用,但不需要提供给外界使用,所以被 private 修饰
void String::init(const char * ss){
    int len =strlen(ss);
    for(int i =0; i <len; i ++){
        this->s[i] =ss[i];
    }
    this->s[len] ='\0';
    this->length =len;
}
//用字符串常量构造 String 对象
//ss: 常量字符串
String::String(const char * ss){
    init(ss);
```

```
}
  //用字符串构造 String 对象
  //ss: 字符串
  //start: 字符串中的起始位置,默认值为 0
  //len: 连续字符个数,默认值为 0
  String::String(char * ss,int start, int len){
      int l =strlen(ss);
      if(len ==0) len =l;            //如果是默认值,则用整个字符串构造 String 对象
      else
          len =min(l,len);           //否则,取字符串长度和参数 len 的最小值来构造对象
      for(int i =0; i <len; i ++){
          this->s[i] =ss[start +i];
      }
      this->s[len] ='\0';
      this->length =len;
}
```

5.2.2　串的赋值

例 5.1　串的赋值。

串的赋值可以通过重载赋值运算符加以实现,代码如下。

```
//赋值符号后面是字符串常量,所以参数类型为 const char *
//具体赋值过程与构造函数 String(const char * ss)一致
void String::operator =(const char * ss){
    init(ss);
}

//返回 String 类中的成员变量,方便输出操作
char * String::toarray(){
    return this->s;
}

//返回串的长度
int String::getlength(){
    return this->length;
}

//测试对象构造和赋值
int main(){
    String s,ss(" C++");
    s ="hello world";              //使用赋值语句构造串的对象
    cout<<"s:"<<s.toarray()<<"(length:"<<s.getlength()<<")"<<endl;
    cout<<"ss:"<<ss.toarray()<<"(length:"<<ss.getlength()<<")"<<endl;
    return 0;
}
```

运行结果:

```
s:hello world(length:11)
ss: C++(length:4)
```

5.2.3 子串截取

例 5.2 子串截取。

截取串中指定位置为 pos、指定长度为 len 的子串,如果从串的 pos 到达串的末尾的字符数量大于 len,则直接截取;否则,根据实际情况截取。

```
//截取子串
//pos: 子串开始的位置
//len: 要截取子串的长度
String String::substring(int pos,int len){
    char t[len +1];
    //计算从 pos 开始串中剩余的字符数,并确定实际可截取子串的长度
    len =min(len,this->getlength() -pos +1);
    for(int i =0; i <len; i ++){                    //形成子串字符数组
        t[i] =this->s[pos +i];
    }
    t[len] ='\0';
    String ss(t);                                   //利用子串字符数组构造 String 对象
    return ss;
}

int main(){
    String s;
    s ="substring";
    String sub =s.substring(2,5);
    cout<<"test1:"<<sub.toarray()<<endl;
    String sub2 =s.substring(2,15);
    cout<<"test2:"<<sub2.toarray()<<endl;
    return 0;
}
```

运行结果:

```
test1: bstri
test2: bstring
```

第一次测试,从第 2 个字符开始截取长度为 5 的子串,此时,串可以满足截取长度为 5 的子串;第二次测试,从第 2 个字符开始截取长度为 15 的字符串,显然,串的长度不够,只能根据实际情况进行截取。

5.2.4 子串插入

例 5.3 子串的插入。

在串指定的位置 i 处插入子串 t,需要从串的最末尾处将字符向后移动 L 个单元(L 为

串 t 的长度），然后再将 t 插入对应位置，如图 5.1 所示。

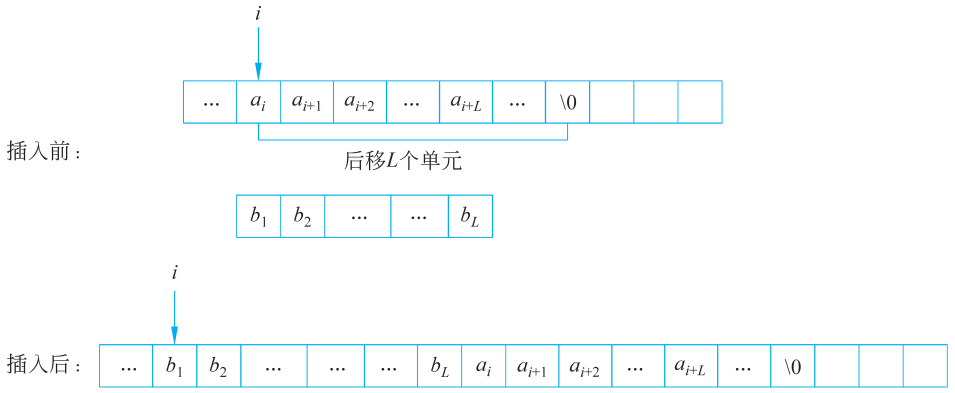

图 5.1　子串插入

完整代码如下。

```
//在串的指定位置插入子串
//pos: 插入子串的位置
//t: 要插入的子串
bool String::strins(int pos,String t){
    int off =t.getlength();
    //串和子串的长度大于串的最大存储空间,插入失败,返回
    if(this->getlength() +off >=MAXLENGTH) return false;
    //从串的最后字符开始,将每个字符移动到 off 个存储单元之后
    for(int i =this->length -1; i >=pos; i --){
        this->s[i +off] =this->s[i];
    }
    char * st =t.toarray();
    //pos 至 pos+off 的位置已空,可以将 t 的内容逐一复制过来
    for(int i =0; i <off; i ++){
        this->s[i +pos] =st[i];
    }
    this->length +=off;
    this->s[this->length] ='\0';
    return true;
}

int main(){
    String s;
    s ="substring";
    String t(" of a ");
    cout<<"source:"<<s.toarray()<<endl;
    cout<<"insert a string at the 3rd char:"<<t.toarray()<<endl;
    if(s.strins(3,t)){
        cout<<s.toarray()<<endl;
    }
    return 0;
}
```

运行结果:

```
source:substring
insert a string at the 3rd char: of a
after inserted:sub of a string
```

5.2.5 串的复制

例 5.4 串的复制。

将一个串复制,构成另外一个串。通过指定复制子串开始的位置 start 以及子串长度 len,构造新的串对象返回。为了方便操作,提供了三个重载函数,包括指定开始 start 和长度 len 的函数,只有开始位置 start(长度为串长 L −start),没有参数(复制整个串)。

完整代码如下。

```
//复制指定子串,从 start 开始,长度为 len
String String::strcpy(int start, int len){
    char* chs =this->toarray();
    String ss(chs,start,len);  //根据串的内容,以及 start、len 的限定构造 String 对象
    return ss;
}

//重载 strcpy,只提供 start,意味着从 start 开始直到串的结尾
String String::strcpy(int start){
    return strcpy(start,this->getlength() -start);
                                                    //构造完整的参数列表,调用第一个函数
}

//重载 strcpy,无参数,意味着完整复制
String String::strcpy(){
    return strcpy(0,this->getlength());        //构造完整的参数列表,调用第一个函数
}

int main(){
    String s;
    s ="substring";
    String cp1 =s.strcpy(2,5);
    String cp2 =s.strcpy(5);
    String cp3 =s.strcpy();
    cout<<cp1.toarray()<<endl;
    cout<<cp2.toarray()<<endl;
    cout<<cp3.toarray()<<endl;
    return 0;
}
```

运行结果:

```
bstri
ring
```

```
substring
```

5.2.6　串的比较

例 5.5　串的比较。

比较两个串的大小，直接调用 strcmp()实现：

```
int String::compare(String t){
    return strcmp(this->s,t.toarray());
}
```

5.2.7　串的拼接

例 5.6　串的拼接。

将一个串拼接在另一个串的后面形成一个更长的串，代码如下。

```
String& String::concat(String ss){
    int base =this->getlength();
    int len =ss.getlength();
    if(base +len >=MAXLENGTH) throw "overflow";
    for(int i =0; i <ss.getlength(); i ++){
        this->s[base +i] =ss.getchar(i);
    }
    length +=ss.getlength();
    this->s[length] ='\0';
    return * this;
}

int main(){
    String s,ss(" C++");
    s ="hello world!";
    try{
          cout<<"s1:"<<s.toarray()<<endl;
          cout<<"s2:"<<ss.toarray()<<endl;
          s =s.concat(ss);
          cout<<"after concat:"<<endl<<s.toarray()<<endl;
    }catch(const char * err){
    }
    return 0;
}
```

运行结果：

```
s1:hello world!
s2: C++
after concat:
hello world! C++
```

5.3 串的模式匹配算法

5.3.1 暴力匹配

例 5.7 串的暴力匹配。

串的匹配是串的重要操作,主要用于从主串中查找到模式串的位置,其中,主串一般指长的串,用于检测是否包含较短的模式串。

暴力匹配的过程如图 5.2 所示。

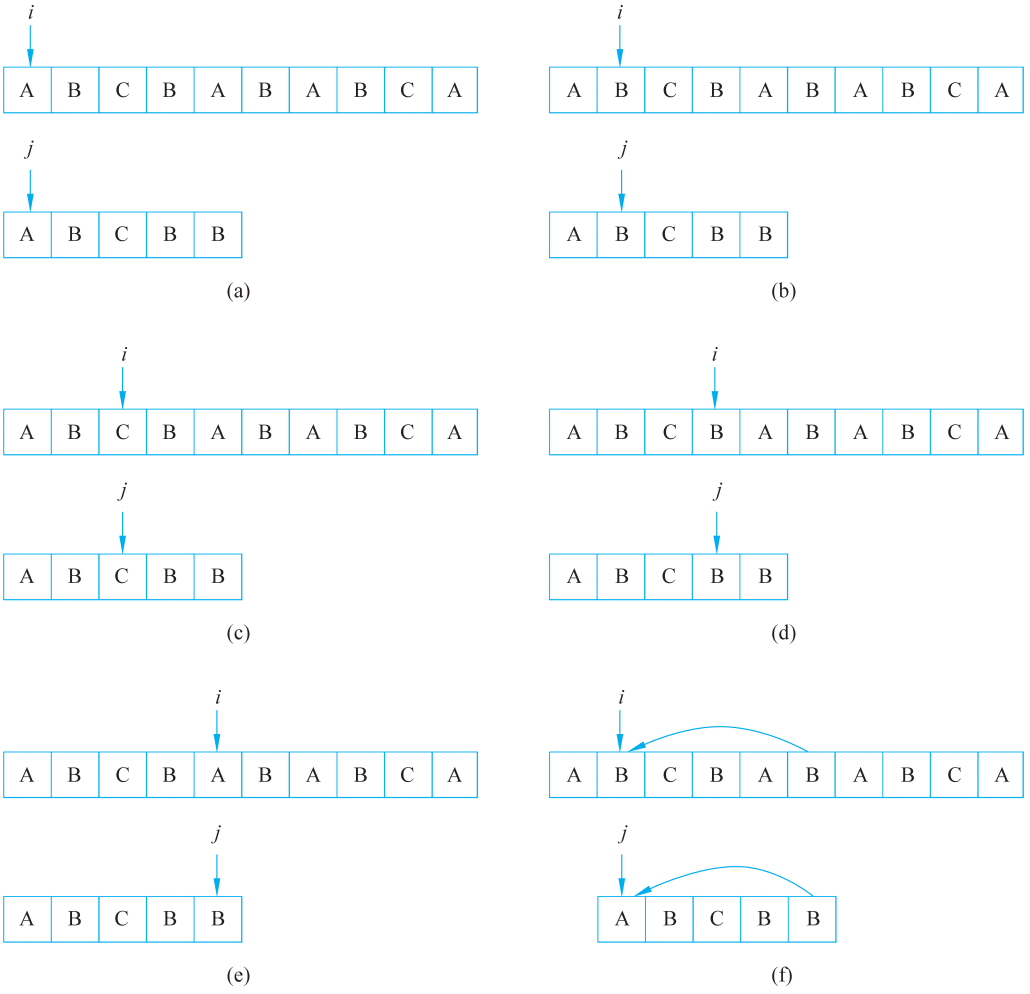

图 5.2 暴力匹配的过程

从图 5.2 可知,主串和模式串都从头开始逐一比对,如果对应位置的字符相同,则主串和模式串都向后移动一个单位,继续比对。当对应位置的字符不相同时,称为失配,此时主串回溯至上次匹配开始的后一个位置,而模式串则回溯至第一个位置,如图 5.2(f)所示。

代码如下。

```
int String::index(String t){
    int n =this->getlength();
    int m =t.getlength();
    bool matched =true;
    char* st =t.toarray();
    int i,j;
    for(i =0; i <n -m +1; i ++){
        matched =true;
        for(j =0; j <m; j ++){
            if(s[i+j] !=st[j]){
                matched =false;
                break;
            }
        }
        if(matched){
            return i;
        }
    }
    return -1;
}
```

暴力匹配的思路简单、操作易行，但其中进行了大量的回溯，特别是主串上回溯的次数过多，导致算法效率低下。假定主串的长度为 n，模式串的长度为 m，则算法的复杂度为 $O(nm)$。

5.3.2　KMP 匹配算法

根据暴力匹配过程，当主串从 $i+1$ 的位置开始与模式串中的字符逐个匹配，但在 $i+k$ 的位置失配，即 $s[i+k] \neq t[k]$，如图 5.3 所示。此时，主串需要从 $i+k$ 的位置回溯到 $i+2$ 的位置，模式串需要从 k 回溯到开头位置，讨论如下两种情况。

图 5.3　匹配失败主串不需要回溯

(1) 如果从 $i+2$ 开始，主串与模式串能够进行完全匹配，则字符串 $s[i+2..i+k-1]=t[1..k-2]$，此时需要逐一判断从 $s[i+k]$ 开始的主串字符是否与 $t[k-1]$ 开始的模式串字符相同，因此在 $i+k$ 匹配失败后，如果主串与模式串能够匹配，则只需要判断从 $i+k$ 位置开始的后续字符是否相同，因此，主串不用回溯。

(2) 如果从 $i+2$ 开始主串与模式串不能完全匹配，则主串需要从 $i+3$ 的位置开始判断主串与模式串是否相同，如果从 $i+3$ 的位置开始主串与模式串匹配，则情况等同于情况(1)，主串仍然不用回溯。如果不能完全匹配，持续试探 $i+4$，…，$i+k$，其推理过程与情况(1)(2)相同。

综合以上两点可知,在匹配过程中如果匹配失败,主串不用回溯。

由于主串不回溯,则只需回溯模式串,因此在主串 $i+k$ 处存在如图 5.4 所示的部分匹配关系。

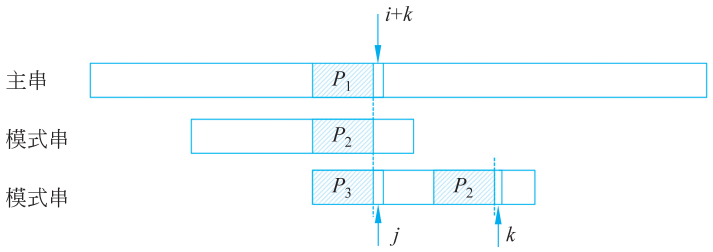

图 5.4 模式串回溯存在的部分匹配

图 5.4 中模式串回溯至前缀 P_3 后面的位置 j 时,P_3 可以与主串 $i+k$ 处的后缀 P_1 匹配,则必然存在后缀 P_1、P_2 与前缀 P_3 相等,即在模式串 k 处失配时,模式串中存在相同的前后缀 P_3、P_2(公共前后缀)。实际中可能有长度不同的多组前后缀,为了不丢失可能的匹配,需要将模式串回溯至最长前缀之后的位置重新开始匹配。假设前缀 P_3、后缀 P_2 为最长公共前后缀,当模式串在位置 k 处失配时,需要回溯到最长公共前缀后面的一个位置 j,记为 next[k]=j,并从位置 j 开始与主串进行匹配。

下面以图 5.5 为例说明模式串的回溯过程。当主串和模式串在位置 k 处失配,主串仍然保持在 k 处,模式串回溯。由于模式串在位置 k 失配时的最长公共前后缀为“AB”,长度为 2,所以子串回溯的位置为 2,并从位置 2 处继续向后匹配。

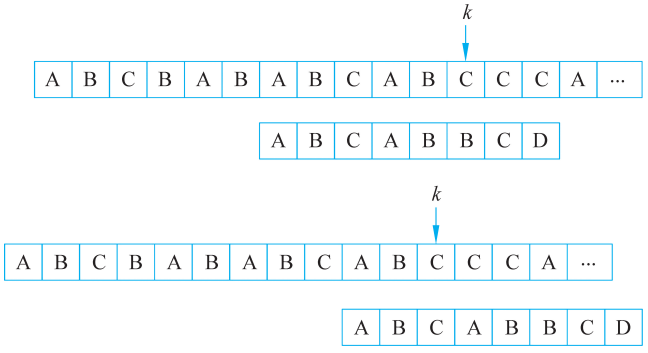

图 5.5 模式串回溯示例

由于模式串的任意位置都可能失配,所以需要计算模式串中每个位置失配时的最长公共前后缀。图 5.6 中存在最长公共前后缀 A=B,即 next[$j-1$]=k,如果 $t[k]=t[j-1]$,则 next[j]=$k+1$。

如果 $t[k]\neq t[j-1]$,则不能在 A、B 的基础上继续生成在 j 处失配时的最长公共前后缀,需搜索新前缀 A_1($A_1=t[1..i]$,$i<k=$next[$j-1$]),并存在后缀 B_1 与 A_1 相同,如图 5.7 所示。

记在位置 $k=$next[$j-1$]处失配时的最长公共前后缀为 A_1、A_2,因为 A=B,所以 $A_2=B_1$,联合 $A_1=A_2$ 可知 $A_1=B_1$。因此,在 $1\sim k-1$ 的范围内存在前缀 A_1,并在位置 $j-1$ 左侧存在后缀 B_1 与前缀 A_1 相同。