

第三章 字符串

任课教员：张铭、赵海燕、冯梅萍、王腾蛟
<http://db.pku.edu.cn/mzhang/DS/>
北京大学信息科学与技术学院
©版权所有，转载或翻印必究

主要内容

- 3.1 字符串抽象数据类型
- 3.2 字符串的存储结构和类定义
- 3.3 字符串运算的算法实现
- 3.4 字符串的模式匹配

3.1 字符串抽象数据类型

- 3.1.1 基本概念
- 3.1.2 String抽象数据类型

3.1.1 基本概念

- 字符串，由0个或多个字符的顺序排列所组成的复合数据结构，简称“串”。
- 串的长度：一个字符串所包含的字符个数。
 - 空串：长度为零的串，它不包含任何字符内容。

3.1.1.1 字符串常数和变量

- 字符串常数
 - 例如：“\n”
- 字符串变量

3.1.1.2 字符

- 字符(char)：组成字符串的基本单位。
- 在C和C++中
 - 单字节（8 bits）
 - 采用ASCII码对128个符号（字符集charset）进行编码

3.1.1.3 字符的编码顺序

- 为了字符串间比较和运算的便利，字符编码表一般遵循约定俗成的“偏序编码规则”。
- 字符偏序：根据字符的自然含义，某些字符间两两可以比较次序。
 - 其实大多数情况下就是字典序
 - 中文字符串有些特例，例如“笔划”序

3.1.1.4 C++ 标准string

- 标准字符串：将C++的<string.h>函数库作为字符串数据类型的方案。
 - 例如：char S[M];
- 串的结束标记：'\0'
 - '\0'是ASCII码中8位BIT全0码，又称为NULL符。

3.1.1.4 C++ 标准string(续)

- 1. 串长函数
int strlen(char *s);
- 2. 串复制
char *strcpy(char *s1, char*s2);
- 3. 串拼接
char *strcat(char *s1, char *s2);
- 4. 串比较
int strcmp(char *s1, char *s2);

3.1.1.4 C++ 标准string(续)

- 5. 输入和输出函数
- 6. 定位函数
char *strchr(char *s, char c);
- 7. 右定位函数
char *strrchr(char *s, char c);

3.1.1.4 C++ 标准string(续)

举例，字符串 s

"The quick brown dog jumps over the lazy fox"

0 1 2 3 4

→012345678901234567890123456789012

寻找字符r，strchr(s, 'r'); 结果返回11。倒着寻找'r'，strrchr(s, 'r');

结果返回29

3.1.2 String抽象数据类型

- 字符串类（class String）：
 - 不采用char S[M]的形式
 - 而采用一种动态变长的存储结构。

```

class String          //字符串 类
//它的存储结构和实现方法使用了C++标准string(简称标准串),
//为了区别,类String所派生创建的实例对象,简称‘本串’,或‘实例串’
//在程序首,要#include <string.h>和#include <iostream.h>及
// 及 #include <stdlib.h>, 以及#include <assert.h>

{
//1. 字符串的数据表示:
//字符串 S 通常用顺序存放,用数组S[]存储,元素的类型为char
//字符串为变长,使用变量size记录串当前长度

// 2. 使用变量访问字符串:
//字符串变量能参与运算,例如S1 + S2表示两个字符串首尾拼接在一起
//用数组str[]存储字符串,在内部可以用str[i]访问串的第i个字符,

// 3. 字符串类的运算集: 请参看下面的成员函数

```

```

private:

public:
char *str;          //私有的指针变量,用于指向存储向量str[size+1]
int size;           //本串的当前实际长度

String(char *s = ""); //创建一个空的字符串
String(char *s); // 创建新字符串,并将标准字符串s拷贝为初值
~String()        // 销毁本串,从计算机存储空间删去本串
//下面是算子的定义,包括赋值算子 = 拼接算子 + 和比较算子 < 等
String& operator= (char *s); //赋值操作=, 标准串s拷贝到本串
String& operator= (String& s); //赋值操作=, 串s复制到本串
String operator+ (char *s); //拼接算子+, 本串拼接标准串s
String operator+ (String& s); //拼接算子+, 本串拼接串s
friend String operator+ (char *s1, String& s);
//友函数作为拼接算子+ 其返回值是一个实例串,等于标准串str拼接串s

```

```

//‘关系’算子,用于比较相等、大、小,例如
int operator< (char *s); //比较大小,本串小于标准串s则返回非0
int operator< (String& s); //比较大小,本串小于串s则返回非0
friend int operator< (char *s1, String& s); //友函数用于比较,
// ,标准串s1小于串s,则返回非0
//‘输入输出’算子 >>和<< 以及 读子串等,例如友函数
friend istream& operator>> (istream& istr, String& s);
friend ostream& operator<< (ostream& ostr, String& s);
// ‘子串函数’: 插入子串、寻找子串、提取子串、删除子串等,例如
String Substr(int index, int count); //它们的功能参见下文
//‘串与字符’函数: 按字符定位等,例如
int Find(char c, int start); //在本串中寻找字符c,从下标start开始找,
// 寻找到c后,返回字符c在本串的下标位置

//其他函数: 求串长、判空串、清为空串、
int strlen(); //返回本串的当前串长
int isEmpty(); //判本串为空串?
void clear(); //清本串为空串
};

```

3.1.2.3 赋值算子、拼接算子和比较算子

- 赋值算子 =
- 拼接算子 +
- 比较算子 < <= > >= != 和 ==

3.1.2.4 输入输出算子 << 和 >>

- 输入算子 >>
- 输出算子 <<

3.1.2.5 处理子串(Substring)的函数

- 简称“子串函数”
 - 提取子串
 - 插入子串
 - 寻找子串
 - 删除子串
 - ...

3.1.2.6 字符串中的字符

- 重载下标算子[]
`char& operator[] (int n);`
- 按字符定位下标
`int Find(char c,int start);`
- 反向寻找，定位尾部出现的字符
`int FindLast(char c);`

3.2 字符串的存储结构和类定义

- 3.2.1字符串的顺序存储
- 3.2.2字符串类class String的存储结构

3.2.1字符串的顺序存储

- 对于串长变化不大的字符串，可以有三种处理方案：
(1) 用S[0]作为记录串长的存储单元。
 - 缺点：限制了串的最大长度不能超过256。

3.2.1字符串的顺序存储(续)

- (2) 为存储串的长度，另辟一个存储的地方。
 - 缺点：串的最大长度一般是静态给定的，不是动态申请数组空间。

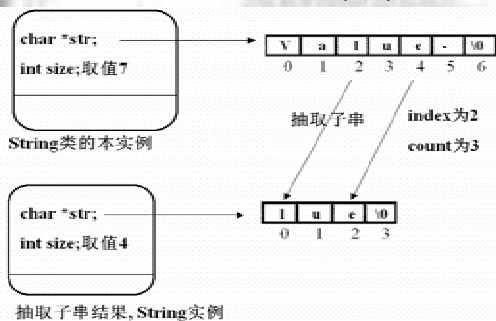
3.2.1字符串的顺序存储(续)

- (3) 用一个特殊的末尾标记'\0'。
 - 例如：C++语言的string函数库（#include <string.h>）采用这一存储结构。

3.2.2 字符串类class String的存储结构

- 抽取子串函数
例如：
`String s1 = "value-";`
`s2 = s1.Substr(2,3);`
上述语句涉及的存储形式如下页所示。

3.2.2 字符串类class String 的存储结构(续)



3.2.2 字符串类class String 的存储结构(续)

- 微软VC++的CString类介绍
 - 自动的动态存储管理，串的最大长度不超过2GB
 - 容器型
 - 不必使用new和delete
- 使用特点：
 - 变量申明
 - CString s6('x', 6); // s6 = "xxxxxxx"
 - CString city = "Philadelphia"; // 串常数作为初值
 - 赋值语句
 - s1 = s2 = "hi there";
 - 变量比较 if(s1 == s2)
 - 方法调用 s1.MakeUpper();
 - 内部字符比较 if(s2[0] == 'h')

3.3 字符串运算的算法实现

- 1. 串长函数


```
int strlen(char *s);
```
- 2. 串复制


```
char *strcpy(char *s1, char*s2);
```
- 3. 串拼接


```
char *strcat(char *s1, char *s2);
```
- 4. 串比较


```
int strcmp(char *s1, char *s2);
```

3.3.1 C++ 标准串运算的实现

- 【算法3-1】字符串的复制


```
char *strcpy(char *d, char *s)
{ //这个程序的毛病是，如果字符串s比字符串d要长，
  //这个程序没有检查拷贝出界，没有报告错误。
  //可能会造成d的越界
  int i=0;
  while (s[i] != '\0') {
    d[i] = s[i]; i++;
  }
  d[i] = '\0';
  return d;
}
```

3.3.1 C++ 标准串运算的实现(续)

- 【算法3-2】字符串的比较


```
int strcmp( char *d, char *s)
{
  int i=0;
  while (s[i] != '\0' && d[i] != '\0')
  {
    if (d[i] > s[i])
      return 1;
    else if (d[i] < s[i])
      return -1;
  }
}
```

3.3.1 C++ 标准串运算的实现(续)

- ```
 i++;
 }
 if(d[i] == '\0' && s[i] != '\0')
 return -1;
 else if (s[i] == '\0' && d[i] != '\0')
 return 1;
 return 0;
}
```

### 3.3.1 C++ 标准串运算的实现(续)

- 【算法3-3】求字符串的长度

```
int strlen(char d[])
{
 int i = 0;
 while (d[i] != 0)
 i++;
 return i;
}
```

### 3.3.1 C++ 标准串运算的实现(续)

- 【算法3-4】寻找字符

```
char * strchr(char *d, char ch)
{
 //按照数组指针d依次寻找字符ch,
 //如果找到ch,则将指针位置返回,
 //如果没有找到ch,则为0值。
 i = 0;
```

### 3.3.1 C++ 标准串运算的实现(续)

```
//循环跳过那些不是ch的字符
while (d[i] != 0 && d[i] != ch) i++;

//当本串不含字符ch,则在串尾结束;
//当成功寻找到ch,返回该位置指针
if (d[i] == 0)
 return 0;
else
 return &d[i];
}
```

### 3.3.1 C++ 标准串运算的实现(续)

- 【算法3-5】反向寻找字符

```
char * strrchr(char *d, char ch)
{
 //按照数组指针d,从其尾部反着寻找字符ch,
 //如果找到ch,则将指针位置返回,
 //如果没有找到ch,则为0值。
 i = 0;
 //找串尾
 while (d[i] != '\0')
 i++;
```

### 3.3.1 C++ 标准串运算的实现(续)

```
//循环跳过那些不是ch的字符
while (d[--i] != '\0' && d[i] != ch)
 ;
//当本串不含字符ch,则在串尾结束;
//当成功寻找到ch,返回该位置指针
if (d[i] == '\0')
 return 0;
else
 return &d[i];
}
```

### 3.3.2 String串运算的实现(续)

- 【算法3-7】创建算子(constructor)

```
String::String(char *s)
{
 //先要确定新创字符串实际需要的存储空间,s的类
 //型为(char *),作为新创字符串的初值。确定
 //s的长度,用标准字符串函数strlen(s)计算长度
 size = strlen(s);

 //然后,在动态存储区域开辟一块空间,用于存
 //储初值s,把结束字符也包括进来
 str = new char [size + 1];
```

### 3.3.2 String串运算的实现(续)

```
//开辟空间不成功时，运行异常，退出
assert(str != '\0');
```

```
//用标准字符串函数strcpy，将s完全
//复制到指针str所指的存储空间
strcpy(str, s);
}
```

### 3.3.2 String串运算的实现(续)

#### ■ 【算法3-8】销毁算子 ( destructor )

```
String::~~String()
{
 //必须释放动态存储空间
 delete [] str;
}
```

### 3.3.2 String串运算的实现(续)

#### ■ 【算法3-9】拼接算子

```
String String::operator+ (String& s)
{
 //把字符串s和本实例拼接成为一个长串返回

 String temp; //创建一个串temp
 int len;
 //准备工作，计算拼接后的长串的长度
 len = size + s.size;
 //把temp串创建时申请的存储空间全部释放
 delete [] temp.str;
 //准备工作，开辟空间，为存放长串之用
 temp.str = new char [len+1];
}
```

### 3.3.2 String串运算的实现(续)

```
//若开辟动态存储空间不成功，则退出
assert(temp.str != 0);
```

```
temp.size = len;
//字符串str (以'\0'结尾) 存到temp
strcpy(temp.str, str);
//再把参数s的str和本实例的str拼接为长串
strcat(temp.str, s.str);
return temp;
}
```

### 3.3.2 String串运算的实现(续)

#### ■ 【算法3-10】赋值算子

```
String String::operator= (String& s)
{
 //参数s将被赋值到本串
 //若本串的串长和s的串长不同，则应该释放本串的
 //str存储空间，并开辟新的空间
 if(size != s.size)
 {
 delete [] str; //释放原存储空间
 str = new char [s.size+1];
 }
}
```

### 3.3.2 String串运算的实现(续)

```
//若开辟动态存储空间失败，则退出正常运行
assert(str != 0);
size = s.size;
}
strcpy(str, s.str);
//返回本实例，作为String类的一个实例
return *this;
}
```

### 3.3.2 String串运算的实现(续)

#### ■【算法3-11】抽取子串函数

```
String String::Substr(int index, int count)
{
 //取出一个子串返回, 自下标index开始, 长度为count
 int i;
 //本串自下标index开始向右数直到串尾, 长度为left
 int left = size - index;
 String temp;
 char *p, *q;
 //若下标index值太大, 超过本串实际串长, 则返回空串
 if(index >= size)
 return temp;
```

### 3.3.2 String串运算的实现(续)

```
 //若count超过自index以右的实际子串长度,
 //则把count变小
 if(count > left)
 count = left;
 //释放原来的存储空间
 delete [] temp.str; //张铭注释: 注意此语句!
 //若开辟动态存储空间失败, 则退出
 temp.str = new char [count+1];
 assert(temp.str != 0);
 //p的内容是一个指针,
 //指向目前暂无内容的字符数组的首字符处
 p = temp.str;
```

### 3.3.2 String串运算的实现(续)

```
 //q的内容是一个指针,
 //指向本实例串的str数组的下标index字符
 q = &str[index];
 //用q指针取出它所指的字符内容后, 指针加1
 //用p该指针所指的字符单元接受拷贝, 该指针也加1
 for (i=0; i < count; i++)
 *p++ = *q++;
 //循环结束后, 让temp.str的结尾为'\0'
 *p = 0;
 temp.size = count;
 return temp;
}
```

### 3.3.2 String串运算的实现(续)

#### ■【算法3-12】查找字符

```
int String::Find(char c, int start)
{
 //在本实例字符串寻找字符c, 如果找到, 则
 //将其下标位置作为整数函数值返回, 如果
 //c没有找到, 则为负值。参数start是下标,
 //从start下标开始寻找c的工作, 若start为
 //0, 则从头寻找
 int i = start;
 assert(i < size);
```

### 3.3.2 String串运算的实现(续)

```
 //循环跳过那些不是c的字符
 while (str[i] != 0 && str[i] != c)
 i++;
 //当本串不含字符c, 则寻找到串尾结束,
 //返回 -1 表示失败; 当成功寻找到c, 返回它的
 //下标位置
 if (str[i] == 0) //注意: 不要搞成“= =”
 return -1;
 else
 return i;
}
```

### 3.4 字符串的模式匹配

- 模式匹配(pattern matching)
  - 一个目标对象S (字符串)
  - 一个模板 (pattern) P (字符串)
- 任务: 用给定的模板P, 在目标字符串S中搜索与模板P全同的一个子串, 并求出S中第一个和P全同匹配的子串 (简称为“配串”), 返回其首字符位置。



$S \quad s_0 \ s_1 \ \dots \ s_i \ s_{i+1} \ s_{i+2} \ \dots \ s_{i+m-2} \ s_{i+m-1} \ \dots \ s_{n-1}$

$P \quad p_0 \ p_1 \ p_2 \ \dots \ p_{m-2} \ p_{m-1}$

为使模式  $P$  与目标  $S$  匹配, 必须满足

$p_0 \ p_1 \ p_2 \ \dots \ p_{m-1} = s_i \ s_{i+1} \ s_{i+2} \ \dots \ s_{i+m-1}$

## 朴素模式匹配

$S = \text{ababababababb} \dots$

$P = \text{abababb}$

X  
abababb

X  
abababb

X  
abababb

X  
abababb

X  
abababb

X  
abababb

X  
abababb

X  
abababb

X  
abababb

### 3.4.1 模式匹配原始算法(续)

#### ■ 【算法3-13】模式匹配原始算法(其一)

```
#include<String.h>
#include<assert.h>
int FindPat_1(String S, String P, int startindex)
{
 // 从S末尾倒数一个模板长度位置
 int LastIndex = strlen(S.str) - strlen(P.str);
 int count = P.size;
 // 开始匹配位置startindex的值过大, 匹配无法成功
 if (LastIndex < startindex)
 return (-1);
```

### 3.4.1 模式匹配原始算法(续)

```
//g为S的游标, 用模板P和S第g位置子串比较, 若失//
//败则继续循环
for (int g = startindex; g <= LastIndex; g++) {
 cout << "P.Size = " << P.size << endl;
 if (P == S.Substr(g, count))
 return g;
}

//若for循环结束, 则整个匹配失败, 返回值为负,
return (-1);
}
```

### 3.4.1 模式匹配原始算法(续)

#### ■ 【算法3-13】模式匹配原始算法(其二)

```
int FindPat_2(String S, String P, int startindex)
{
 // 从S末尾倒数一个模板长度位置
 int LastIndex = S.size - P.size;

 // 开始匹配位置startindex的值过大, 匹配无法成功
 if (LastIndex < startindex)
 return (-1);
 // i 是指向S内部字符的游标,
 // j 是指向P内部字符的游标
 int i = startindex, j = 0;
```

### 3.4.1 模式匹配原始算法(续)

```
//下面开始循环匹配
while (i < S.size && j < P.size) // "<="呢?
 if (P.str[j] == S.str[i])
 {
 i++;
 j++;
 }
 else {
 i = i - j + 1;
 j = 0;
 }
```

### 3.4.1 模式匹配原始算法(续)

```
//如果匹配成功,则返回该S子串的开
//始位置;如果P和S匹配失败,函数
//返回值为负
if (j >= P.size) // ">" 可以吗?
 return (i - j);
else
 return -1;
}
```

### 3.4.1 模式匹配原始算法(续)

- 分析
  - 假定目标S的长度为n, 模板P长度为m, m n
  - 在最坏的情况下, 每一次循环都不成功, 则一共要进行比较 (n-m+1) 次
  - 每一次“相同匹配”比较所耗费的时间, 是P和S逐个字符比较的时间, 最坏情况下, 共m次。
  - 因此, 整个算法的最坏时间开销估计为O(m • n)。

### KMP算法思想

S  $s_0 s_1 \dots s_{i-j-1} s_{i-j} s_{i-j+1} s_{i-j+2} \dots s_{i-2} s_{i-1} s_i \dots s_{n-1}$

P  $p_0 p_1 p_2 \dots p_{j-2} p_{j-1} p_j$

则有  $s_{i-j} s_{i-j+1} s_{i-j+2} \dots s_{i-1} = p_0 p_1 p_2 \dots p_{j-1}$  (1)

如果  $p_0 p_1 \dots p_{j-2} \neq p_1 p_2 \dots p_{j-1}$  (2)

则立刻可以断定

$p_0 p_1 \dots p_{j-2} \neq s_{i-j+1} s_{i-j+2} \dots s_{i-1}$

(朴素匹配的)下一趟一定不匹配, 可以跳过去

同样, 若  $p_0 p_1 \dots p_{j-3} \neq p_2 p_3 \dots p_{j-1}$

则再下一趟也不匹配, 因为

$p_0 p_1 \dots p_{j-3} \neq s_{i-j+2} s_{i-j+3} \dots s_{i-1}$

直到对于某一个“k”值(首尾串长度), 使得

且  $p_0 p_1 \dots p_k \neq p_{j-k-1} p_{j-k} \dots p_{j-1}$

则  $p_0 p_1 \dots p_{k-1} = p_{j-k} p_{j-k+1} \dots p_{j-1}$

$p_0 p_1 \dots p_{k-1} = s_{i-k} s_{i-k+1} \dots s_{i-1} s_i$

模式右滑j-k位

### 模式右滑j-k位

$s_{i-j} s_{i-j+1} s_{i-j+2} \dots s_{i-k} s_{i-k+1} \dots s_{i-1} s_i$

$p_0 p_1 p_2 \dots p_{j-k} p_{j-k+1} \dots p_{j-1} p_j$

$p_0 p_1 \dots p_{k-1} p_k$

### 3.4.2 字符串的特征向量N

- 设模板P由m个字符组成:
- 记为  $P = q_0 q_1 q_2 q_3 \dots q_{m-1}$
- 令特征向量N用于表示模板P的字符分布特征, 并简称N向量。它和P同长, 由m个特征数  $n_0 \dots n_{m-1}$  非负整数组成:
- 记为  $N = n_0 n_1 n_2 n_3 \dots n_{m-1}$

### 3.4.2 字符串的特征向量N(续)

- 下面说明 $n_i$ 的含义和它的递归定义：  
列出模板P开头的任意t个字符，把它称为P的前缀子串。

$$q_0 q_1 q_2 \dots q_{t-1}$$

### 3.4.2 字符串的特征向量N(续)

在P的第i位置的左边，也取出t个字符，称为i位置的左子串。

$$\mathbf{q}_{i-t+1} \cdots \mathbf{q}_{i-2} \mathbf{q}_{i-1} \mathbf{q}_i$$

### 3.4.2 字符串的特征向量N(续)

- 计算特征数 $n_i$ 
  - 设法求出最长的（ $t$ 最大的）能够与前缀子串匹配的左子串（简称第 $i$ 位的最长前缀串）。
  - $t$ 就是要求的特征数 $n_i$ 。

### 3.4.2 字符串的特征向量N(续)

- 特征数  $n_i$  ( $0 \leq n_i \leq i$ ) 是递归定义的, 定义如下:
  - $n_0 = 0$ ;
  - 对于  $i > 1$  的  $n_i$ , 假定已知前一位位置的特征数  $n_{i-1}$ , 并且  $n_{i-1} = k$ ;
  - 如果  $q_i = q_k$ , 则  $n_i = k + 1$ ;
  - 当  $q_i \neq q_k$  且  $k = 0$  时, 则令  $k = n_{k-1}$ ;
  - 让 循环直到条件不满足;
  - 当  $q_i = q_k$  且  $k = 0$  时, 则  $n_i = 0$ ;

### 3.4.2 字符串的特征向量N(续)

- ### ■ 举例：

P4 = "aaaabaaaaac"  
N = "0123012340"

P4      aaaabaaaaac

前綴 a  
前綴 aa  
前綴 aaa  
前綴  
前綴 a  
前綴 aa  
前綴 aaa  
前綴 aaaa  
前綴

### 3.4.2 字符串的特征向量N(续)

[illegible]

### 3.4.2 字符串的特征向量N(续)

#### ■ 【算法3-14】计算向量N

```
int *Next(String P)
{
 int m = P.size; //m为模板P的长度
 assert(m > 0); //若m=0,退出
 int *N = new int[m]; //动态存储区开辟整数数组
 assert(N != 0); //若开辟存储区域失败,退出
 N[0] = 0;

 for(int i = 1; i < m; i++) //分析P的每个位置i
 {
 int k = N[i-1]; //第(i-1)位置的最长前缀串长度
```

### 3.4.2 字符串的特征向量N(续)

```
//以下while语句递推合适的前缀位置k
while(k > 0 && P.str[i] != P.str[k])
 k = N[k-1];

//根据P[i]比较第k位置前缀字符, 决定N[i]
if(P.str[i] == P.str[k])
 N[i] = k+1;
else
 N[i] = 0;
}
return N;
}
```

### 3.4.3 KMP模式匹配算法

#### ■ 【算法3-15】KMP模式匹配算法

```
int KMP_FindPat(String S, String P, int *N, int startindex) {
 //假定事先已经计算出P的特征数组N, 作为输入参数

 //S末尾再倒数一个模板长度位置
 int LastIndex = S.size - P.size;
 if((LastIndex - startindex) < 0)
 return (-1); //startindex过大, 匹配无法成功

 int i; //i是指向S内部字符的游标,
 int j = 0; //j是指向P内部字符的游标,
```

### 3.4.3 KMP模式匹配算法(续)

```
//S游标i循环加1
for(i = startindex; i < S.size; i++) {
 //若当前位置的字符不同, 则用N循环求当前的j,
 //用于将P的恰当位置与S的i位置对准
 while(P.str[j] != S.str[i] && j > 0)
 j = N[j-1];
 //P[j]与S[i]相同, 继续下一步循环
 if(P.str[j] == S.str[i]) j++;
 //匹配成功, 返回该S子串的开始位置
 if(j == P.size)
 return (i - j + 1);
 return (-1); //P和S整个匹配失败, 函数返回值为负
}
```

### KMP模式匹配示例(一)

```
0 1 2 3 4 5 6
P = a b a b a b b
N = [0 0 1 2 3 4 0]

0 1 2 3 4 5 6 7 8 9 10 11 12
S = a b a b a b a b a b a b...
 a b a b a b b
 X i=6, j=6, N[j-1]=4
 a b a b a b b
 X i=8, j=6, N[j-1]=4
 a b a b a b b
 X i=10, j=6, j'=4
 a b a b a b b
 ✓
```

```
0 1 2 3 4 5 6 7 8 9
P = a a a a b a a a a c
N = [0 1 2 3 0 1 2 3 4 0]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
S = a a b a a a a a b a a a c b c a a b a b c
 a a a b a a a c
 X i=2, j=2, N[j-1]=1
 a a a b a a a c
 X i=2, j=1, N[j-1]=0
 a a a b a a a c
 X i=7, j=4, N[j-1]=3
 a a a b a a a c
 X i=8, j=4, N[j-1]=3
 a a a b a a a c
 ✓
```

```

0 1 2 3 4 5 6 7 8 9
P = a a a a b a a a c
N = [0 1 2 1 0 1 2 3 4 0]
X (不是最长的, 应该是3)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
S = a a b a a a a a b a a a c b c a a b a b c
 a a a b a a a a c
 X i=2, j=2, N[j-1]=1
 a a a a b a a a c
 X i=2, j=1, N[j-1]=0
 a a a a b a a a c
 X i=7, j=4, N[j-1]=1
 a a a a b a a a c
 (错过了!)

```

北京大学信息学院

©版权所有, 转载或翻印必究

Page 73

## KMP算法的效率

- 两重循环
  - for循环最多执行 $n = S.size$ 次
  - 其内部的while循环, 最长循环次数是 $m = P.size$ 次。
- 初看起来其时间开销也可能达到 $O(n \times m)$ 。

北京大学信息学院

©版权所有, 转载或翻印必究

Page 74

- 循环体中“ $j=N[j-1];$ ”语句的执行次数不能超过 $n$ 次。否则,
  - 由于“ $j=N[j-1];$ ”每执行一次必然使得 $j$ 减少(至少减1)
  - 而使得 $j$ 增加的操作只有“ $j++$ ”
  - 那么, 如果“ $j=N[j-1];$ ”的执行次数超过 $n$ 次, 最终的结果必然使得 $j$ 为负数。这是不可能的。
- 同理可以分析出求next数组的时间为 $O(m)$
- 因此, KMP算法的时间为 $O(n+m)$

北京大学信息学院

©版权所有, 转载或翻印必究

Page 75

## 总结

- 字符串抽象数据类型
- 字符串的存储结构和类定义
- 字符串运算的算法实现
- 字符串的模式匹配

北京大学信息学院

©版权所有, 转载或翻印必究

Page 76