串

从数据结构角度讲，串属于线性结构，是一种数据元素为字符的特殊的线性表。串的逻辑结构和线性表类似，其特殊性主要表现在以下两个方面：

1. 串中的一个元素是一个字符。
2. 操作的对象一般不再是单个元素，而是一组元素。

串的基本操作和线性表的基本操作有很大差别。在线性表的基本操作中，大多数以单个元素作为操作对象，例如查找某个元素、插入或删除一个元素。而在串的基本操作中，通常以“串的整体”为操作对象，例如在主串中查找子串位置、截取一个子串，插入或删除一个子串等等。

串的基本概念

1. 串(String)是字符串的简称。它是一种在元素的组成上具有一定约束条件的线性表，即要求组成线性表的所有元素都是字符。所以人们经常这样定义串：由0个或多个字符顺序排列所组成的有限序列。串一般记作：$S=”S_0S_1\dots S_i\dots S_{n-1}”\ (n\geq 0, 0\leq i<n)$，其中，$S$是串的名字，用一对双引号括起来的字符序列是串的值；S_i可以是字母、数字字符或其他字符。
2. 串的长度：一个串所包含的字符的个数n，称为串的长度。
3. 空串：当串的长度$n=0$时，串中没有任何字符，称为空串。
4. 空格串：由空格字符组成的串，称为空格串。
5. 子串：串中任意个连续的字符组成的子序列称为该串的子串。空串是任意串的子串；任意串都是自身的子串。
6. 真子串：非空且不为自身的子串，称为真子串。
7. 主串：包含子串的串，称为该子串的主串。
8. 子串定位：查找子串在主串中第一次出现的位置。
9. 串相等：若两个串的长度相等，并且各对应的字符也都相同，则称两个串相等。

串的表示和实现

串的顺序存储结构

串的顺序存储结构用一组连续的存储单元依次存储串中的字符序列。为适应串长度动态变化的需求，不宜采用字符数组`char Str[maxSize]`存储串，而应该采用动态存储管理方式。顺序存储结构的串的类型定义如下：

```
class String{
    char* data; // 存储串
    int maxSize; // 最大存储容量
    int curLength; // 串的长度
    void resize(int len); // 扩大数组空间
public:
    String(const char* str = nullptr); // 构造函数
    String(const String& str); // 拷贝构造函数
    ~String(){delete [] data;} // 析构函数
```

```
    int capacity() const {return maxSize;} // 最大存储容量
    int size() const {return curLength;} // 求串长度
    bool empty() const {return curLength == 0;} // 判空
    int compare(const String& s) const; // 比较当前串和串s的大小
    String subStr(int pos, int num) const; // 从pos位置开始取长度为num的子串
    int bfFind(const String& s, int pos = 0) const; // 朴素的模式匹配算法
    String& insert(int pos, const String& s); // 在pos位置插入串s
    String& erase(int pos, int num); // 删除从pos开始的num个字符
    const char* toCharStr() const {return data;} // 获取字符数组data
    int kmpFind(const String& t, int pos = 0); // 改进的模式匹配算法
    void getNext(const String& t, int *next); // 获取next数组
    void getNextVal(const String& t, int *nextVal); // 获取nextVal数组
    bool operator==(const String& str) const; // 重载==，判断两个串是否相等
    String& operator+(const String& str); // 重载+，用于串的连接
    String& operator=(const String& str); // 重载=，用于串间赋值
    char& operator[](int n) const; // 重载[]，通过下标取串中字符
    friend istream& operator>>(istream& cin, String& str); // 重载>>，用于输入串
    friend ostream& operator<<(ostream& cout, String& str); // 重载<<，用于输出串
};
class outOfRange:public exception{ // 用于检查范围的有效性
public:
    const char* what()const throw(){
        return "ERROR! OUT OF RANGE.\n";
    }
};
class badSize:public exception{ // 用于检查长度的有效性
public:
    const char* what()const throw(){
        return "ERROR! BAD SIZE.\n";
    }
};
```

构造函数

```
String::String(const char* str) {
    maxSize = 2*strlen(str);
    data = new char[maxSize + 1]; // +1 for '\0'
    strcpy(data, str);
    curLength = strlen(data);
}
```

拷贝构造函数

```
String::String(const String& str) {
    maxSize = str.maxSize;
    curLength = str.curLength;
    data = new char[maxSize + 1]; // +1 for '\0'
    strcpy(data, str.data);
}
```

比较当前串与串s的大小

```
int String::compare(const String& s) const {
    int i = 0;
    while(s.data[i] != '\0' || this->data[i] != '\0'){
        if(this->data[i] > s.data[i]) return 1; // 当前串大于s
        if(this->data[i] < s.data[i]) return -1; // 当前串小于s
        i++;
    }
    if (this->data[i] == '\0' && s.data[i] != '\0') return -1; // s有剩余元素
    if (this->data[i] != '\0' && s.data[i] == '\0') return 1; // this有剩余元素
    return 0; // 两个串相等
}
```

取子串

取子串：从主串中下标为pos的位置开始取长度为num的子串。当pos == curLength或num == 0时，取到的子串为空串；当num > curLength - pos时，修改num = curLength - pos。

```
String String::subStr(int pos, int num) const {
    int i;
    String tmp("");
    if(pos > curLength || pos < 0) throw outOfRange();
    if(num < 0) throw badSize();
    if(num > curLength - pos) num = curLength - pos; // 限制num的大小
    delete [] tmp.data; // 释放tmp本来的存储空间
    tmp.maxSize = tmp.curLength = num;
    tmp.data = new char[num + 1]; // 申请大小为num + 1的空间
    for(i = 0; i < num; i++) // 长度为num的子串赋值给tmp
        tmp.data[i] = data[pos + i];
    tmp.data[i] = '\0'; // 添加字符串结束符
    return tmp;
}
```

插入

在当前串的pos位置插入串s。

```
String& String::insert(int pos, const String& s){
    if(pos > curLength || pos < 0) throw outOfRange();
    if(curLength + s.curLength > maxSize){
        resize(2*(curLength + s.curLength)); // 扩大数组空间
    }
    for(int i = curLength; i >= pos; i--){
        data[i + s.curLength] = data[i]; // 将pos位置及之后的字符后移s.curLength个位置
    }
    for(int j = 0; j < s.curLength; j++){
        data[pos + j] = s.data[j]; // 将s插入到pos位置
    }
    curLength += s.curLength; // 更新当前串长度
    return *this;
}
```

删除

```
String& String::erase(int pos, int num){
    if(pos > curLength || pos < 0) throw outOfRange();
    if(num < 0) throw badSize();
    if (num > curLength - pos)
        num = curLength - pos; // 限制num的大小
    for(int i = pos; i < curLength - num; i++)
        data[i] = data[i + num]; // 将pos位置之后的字符前移num个位置
    curLength -= num; // 更新当前串长度
    return *this;
}
```

扩大数组空间

```
void String::resize(int len){
    maxSize = len;
    char* temp = new char[maxSize + 1];
    strcpy(temp, data); // 将原有数据复制到新空间
    delete[] data; // 释放原有空间
    data = temp; // 更新data指针
}
```

重载+运算符

用于串连接

```
String& String::operator+(const String& str){
    if (curLength + str.size() > maxSize)
        resize(2*(curLength + str.size()));
    strcat(data, str.data); // 连接两个串
    curLength += str.size(); // 更新当前串长度
    return *this;
}
```

重载=运算符

用于串赋值

```
String& String::operator=(const String& str){
    if (this == &str) return *this; // 防止自赋值
    delete[] data; // 释放原有空间
    maxSize = str.maxSize;
    curLength = str.curLength;
    data = new char[maxSize + 1]; // +1 for null terminator
    strcpy(data, str.data); // 复制数据
    return *this;
}
```

串的赋值运算会改变原有的串值，为了避免内存泄漏，最好释放原空间后再重新申请新的存储空间。

重载==运算符

用于判断两个串是否相等

```
bool String::operator==(const String& str) const {
    if (curLength != str.curLength) return false;
    return strcmp(data, str.data) ? false : true; // 使用strcmp比较两个串
}
```

重载[]运算符

用于通过下标运算存取串中字符

```
inline char& String::operator[](int n) const {
    if (n < 0 || n >= curLength) throw outOfRange();
    else return data[n]; // 返回指定位置的字符
}
```

重载>>运算符

用于输入串

```
istream& operator>>(istream& cin, String& str) {
    char* temp = new char[1000]; // 假设最大输入长度为999
    cin >> temp;
    str.maxSize = 2 * strlen(temp);
    str.data = new char[str.maxSize + 1];
    strcpy(str.data, temp); // 复制输入的字符串
    str.curLength = strlen(temp);
    delete [] temp; // 释放临时空间
    return cin;
}
```

重载<<运算符

用于输出串

```
ostream& operator<<(ostream& cout, String& str) {
    cout << str.data; // 输出字符串
    return cout;
}
```

串的链式存储结构

因为串结构中一个元素就是一个字符，所以最简单的链式存储结构是在一个结点的数据域中存放一个字符。优点是操作方便，缺点是存储密度低。另一种链式存储结构称为块链式存储结构，即在一个结点的数据域中存放多个字符。这样做提高了存储密度，但是带来了新的问题，就是插入和删除操作可能会在结点间大量地移动字符，算法实现复杂。因此串一般采用顺序存储结构来表示和实现。

串的模式匹配

子串的定位操作通常称为模式匹配(或模型匹配)，设有主串S和子串T，如果在主串S中找到一个与子串T相等的子串，则返回子串T第一次出现在主串S中的位置，即子串T的第一个字符在主串S中的位置。其中主串又称为目标串，子串又称为模式串。

朴素的模式匹配算法

朴素的模式匹配算法(Brute-Force算法，简称为BF算法)是模式匹配的一种常规算法，它的基本思想是：从主串$S=$"$S_0S_1\dots S_{n-1}$"的第1个字符开始与子串$T=$"$T_0T_1\dots T_{m-1}$"中的第1个字符进行比较(匹配)。若S_i与T_j相等，则指示两个串的指针i和j后移(i++，j++)，继续进行下一个字符的比较。若S_i和T_j不相等，则主串指针i回溯到上一次比较的起始位置的后一位(即i = i - j + 1)，而子串指针j回溯到第一个字符(即j = 0)，继续进行下一次比较。比较过程一直进行到匹配成功或失败为止。若匹配成功则返回子串的第一个字符在主串中的位置；否则，匹配失败，返回-1。

假设主串的长度为n，子串的长度为m。

最好的情况是：主串的前m个字符刚好与子串相等，此时算法的最好时间复杂度为$O(m)$。

最坏的情况是：每趟都匹配到子串的最后一个字符才失配，主串将进行大量地回溯。此时算法效率最低，最坏时间复杂度为$O(mn)$。

```
int String::bfFind(const String& s, int pos) const {
    int i = 0, j = 0;
    if(curLength < s.curLength) return -1; // 如果当前串长度小于s的长度，返回-1
    while (i < curLength && j < s.curLength){
        if(data[i] == s.data[j])
            i++, j++; // 如果当前字符相等，继续比较下一个字符
        else{
            i = i - j + 1; // 如果不相等，i回退到下一个可能的匹配位置
            j = 0; // j回到s的起始位置
        }
    }
    if(i >= s.curLength) return (i - s.curLength); // 如果j遍历完s，返回匹配的起始位置
    return -1; // 如果没有找到匹配，返回-1
}
```

KMP算法

KMP算法是一种对朴素的模式匹配算法的改进算法，它的特点是主串无需回溯，主串指针一直在向后移动，只有子串指针回溯，大大减少算法的比较次数和回溯次数。该算法可在$O(m+n)$的时间复杂度量级上完成串的模式匹配操作。

当主串中下标为i的字符与子串中下标为j的字符失配时，主串指针i不回溯，子串指针j回溯（子串相对于主串向右滑动）到一个合适的位置，重新进行比较。当然，子串向右滑动的距离越远越好，那么如何确定应该滑到什么位置呢？也就是说，与主串第i个字符比较的子串的“下一个”字符怎么确定呢？

假设主串$S=$"$S_0S_1\dots S_{n-1}$"，子串$T=$"$T_0T_1\dots T_{m-1}$"，当S_i和T_j比较时失配，主串不回溯，用子串中第k $(k<j)$个字符T_k与S_i继续进行比较，此时将子串向右滑动，使得T_k与S_i对齐进行比较，则隐含下式成立：

$$"S_{i-k}S_{i-k+1}\dots S_{i-1}"="T_0T_1\dots T_{k-1}" \qquad (0<k<j)$$

而从之前的匹配结果，即主串下标为i的字符与子串下标为j的字符失配，可知下式成立：

$$"S_{i-k}S_{i-k+1}\dots S_{i-1}"="T_{j-k}T_{j-k+1}\dots T_{j-1}" \qquad (0<k<j)$$

由前两个式子可得：

$$"T_0\dots T_{k-1}"="T_{j-k}\dots T_{j-1}" \qquad (0<k<j)$$

反之，若子串满足上式，则在匹配过程中，当主串中下标为i的字符与子串中下标为j的字符失配时，仅需要将子串向右滑动至子串中下标为k的字符与主串中下标为i的字符对齐即可，子串前面k个字符必定与主串下标为i的字符前长度为k的子串相等，因此，仅需从子串中下标为k的字符与主串中下标为i的字符开始继续进行比较。

因此KMP算法的关键问题变成了如何求k的值。在已经匹配的子串”$T_0T_1\dots T_{j-1}$”中，找出最长的相同的前缀子串”$T_0T_1\dots T_{k-1}$”(首串)和后缀子串”$T_{j-k}T_{j-k+1}\dots T_{j-1}$”(尾串)，这是k的值也就确定了。需要注意的是，在查找最长的相同首串和尾串时，两者可以部分重叠。

为了计算每次失配时子串指针j回溯的位置k，KMP算法采用以空间换时间的方式，申请一个与子串长度相等的整形数组`next`。令`next[j] = k`，则`next[j]`表示当子串中T_j与主串中S_i失配时，在子串中需重新和主串中S_i进行比较的字符的位置为k，由此子串的`next`数组定义为：

$$\text{next[j]}\begin{cases}-1 & \text{When j} = 0\\ \max\{k|0<k<j \text{ and } ”T_0\dots T_{k-1}” = ”T_{j-k}\dots T_{j-1}”\} & \text{When it's not empty}\\ 0 & \text{When in other cases}\end{cases}$$

上述公式分为以下三种情况：

1. 当`j = 0`时，`next[j] = -1`，表示子串指针指向下标为0的元素时失配，子串指针不用回溯，需要将主串指针i向后移动一位，而子串指针仍然指向下标为0的元素，然后进行下一趟比较。
2. 在已经匹配的子串中，存在相等的最长首串”$T_0\dots T_{k-1}$”和尾串”$T_{j-k}\dots T_{j-1}$”，此时主串指针i不懂，子串指针回溯到`next[j] = k`的位置，然后进行下一趟比较。
3. 在已经匹配的子串中，若不存在相等的首串和尾串，则主串指针不动，子串指针回溯到`j = 0`的位置，然后进行下一趟比较。

在模式匹配过程中，若发生失配，则主串下标i不变，利用`next`数组，求出子串下标为j的位置失配时，应滑动到的新位置k。若$k \geq 0$，则将主串下标为i的字符与子串下标为k的字符进行比较。若匹配，则继续比较后面字符；若失配，仍然利用next数组求出k支配后的下一个比较位置k’，再重复以上操作。若`k = -1`，则说明子串已经滑到头，主串下标i向后移动一位与子串的第一个字符进行比较。这就是KMP算法的基本思想。

```
int String::kmpFind(const String& t, int pos){
    if(t.curLength == 0) return 0;
    if(curLength < t.curLength) return -1; // 如果当前串长度小于t的长度，返回-1
    int i = 0, j = 0;
    int* next = new int[t.curLength]; // 创建next数组
    getNext(t, next);
    while(i < curLength && j < t.curLength){
        if(j == -1 || data[i] == t.data[j])
            i++, j++;
        else j = next[j]; // 如果不匹配，使用next数组跳过不必要的比较
    }
    delete [] next; // 释放next数组
    if(j >= t.curLength) return (i - t.curLength); // 如果j遍历完t，返回匹配的起始位置
    else return -1; // 如果没有找到匹配，返回-1
}
```

KMP算法的关键在于求`next`数组，下面用一个递推过程来求解`next`数组。

```
void String::getNext(const String& t, int *next){
    int i = 0, j = -1;
    while(i < t.curLength - 1){
        if((j == -1) || (t[i] == t[j])){
            ++i, ++j;
            next[i] = j; // 如果匹配成功，更新next数组
        }else j = next[j];
    }
}
```