字符串可以说是我们实际工作中使用最多的数据类型了，常见的字符串操作包括链接、取子串、格式化等。这部分内容总体来说比较容易理解，最难的部分要数字符串的模式匹配方法了，尤其是KMP算法，需要通过实践加以记忆。

**串的定义**：是由零个或者多个字符组成的有限序列，又叫字符串。串的比较是通过其编码的顺序进行的(对于ASCII码来说，其通过7个二进制表示一个字符，共可以表示128个字符)，通常来说，对于给定的两个字符串s=a1a2…an，t=b1b2…bm来说，如果n<m且ai=bi(i=1,2…n)，那么有s<t；如果存在某个k<=min(m,n),使得ai=bi(i=1,2…k-1),ak<bk,则s<t。接下来通过一个表格来了解串的常见操作（用C#自定义字符串类实现）。

|  |  |
| --- | --- |
| 操作 | 诠释 |
| StrAssign(T, \*chars) | 生成一个其值等于字符串常量chars的串T |
| StrCopy(T,S) | 串S存在，由串S复制得到T |
| ClearString(S) | 若S存在，将串清空 |
| StringEmpty(S) | 若S存在，返回true,否则返回false |
| StrLength(S) | 返回串S的元素个数，即串的长度 |
| StrCompare(S,T) | 若S>T返回值>0,若相等返回0，若S<T的返回值<0 |
| Concat(T,S1,S2) | 用T返回由S1和S2联结成的新串 |
| SubString(Sub,S,pos,len) | 若串存在，1<=pos<=StrLength(S),且0<=lenStrLength(S)-pos+1,  用Sub返回串S的第pos个字符起长度为len的子串。 |
| Index(S,T,pos) | 串S和T存在，T是非空串，1<=pos<=StrLength(S)  若主串S中存在和串T相同的子串，则返回它在主串S中第pos个字符之后第一次出现的位置，否则返回0。 |
| Replace(S,T,V) | 串S,T,V存在，T是非空串，用V替换主串S中出现的所有与T相等的不重叠的子串 |
| StrInsert(S,pos,T) | 串S和T存在，1<=pos<=StrLength(S)+1  在串S的第pos个字符前插入串T |
| StrDelete(S,pos,len) | 串S存在，1<=pos<=StrLength(S)-len+1  从串S中删除第pos个字符起长度为len的子串 |

**串的存储结构**：通常来说，串都是使用顺序存储结构来实现的，链式的存储结构不太适合串。串的顺序存储是由一组地址连续的存储单元来存储串中的字符序列的，按照预定义的大小，为每个定义的串变量分配一个固定长度的存储区，一般用定长数组实现，用’\n’来表示一个串值得终结。

**串的实现**，如下所示，使用javascript编写，依赖于数组方法。

|  |
| --- |
| //////////////////////////////////////基础部分///////////////////////////////////////////////////////////  function MyString(strArray) {  //均需依赖于数组的操作  this.strArray = strArray;  this.strCopy = function() {  var newString = new MyString();  newString.strArray = this.strArray.slice();  return newString;  }  this.clearString = function() {  this.strArray = [];  }  this.stringEmpty = function() {  if (this.strLength() == 0) {  return true;  }  return false;  }  this.strLength = function() {  return this.strArray.length;  }  this.strCompare = function(otherStrArray) {  var compareLength = Math.min(this.strLength(), otherStrArray.length);  var i;  for (i = 0; i < compareLength; i++) {  if (this.strArray[i] != otherStrArray[i]) {  break;  }  }  if (this.strArray[i] > otherStrArray[i]) {  return 1;  } else if (this.strArray[i] == otherStrArray[i]) {  return 0;  } else {  return -1;  }  }  this.concat = function(otherStrArray) {  var newString = new MyString();  newString.strArray = this.strArray.concat(otherStrArray);  return newString;  }  this.subString = function(pos, len) {  if (pos >= 0 && len >= 0 && pos + len < this.strLength()) {  var newString = new MyString();  newString.strArray = this.strArray.slice(pos, pos + len);  return newString;  }  }  this.index = function(subStrArray, pos) {  if (pos < 0 || subStrArray.length > this.strLength() || pos >= this.strLength() - subStrArray.length) {  return -1;  }  for (var i = pos; i < this.strLength(); i++) {  var tempI = i;  for (j = 0; j < subStrArray.length; j++) {  if (subStrArray[j] != this.strArray[tempI]) {  break;  }  tempI++;  }  // console.log(tempI);  if (tempI == i + subStrArray.length) {  return i - pos;  }  // console.log(i);  }  return -1;  }  this.replace = function(originalSubStrArray, targetSubStrArray) {  //需要依赖于index方法  var pos = this.index(originalSubStrArray, 0);  this.strDelete(pos, originalSubStrArray.length);  this.strInsert(pos, targetSubStrArray);  }  this.strInsert = function(pos, subStrArray) {  if (pos >= 0 && pos <= this.strLength()) {  var partA = this.strArray.slice(0, pos);  var partB = this.strArray.splice(pos, this.strLength());  this.strArray = partA.concat(subStrArray, partB);  }  }  this.strDelete = function(pos, len) {  if (pos >= 0 && len >= 0 && pos + len <= this.strLength()) {  var partA = this.strArray.slice(0, pos);  var partB = this.strArray.splice(pos + len, this.strLength());  this.strArray = partA.concat(partB);  }  }  }  }  //////////////////////////////////////测试部分///////////////////////////////////////////////////////////  var strArray = ['x', 'i', 'o', 'n', 'g', 'e', 'r'];  var otherStrArray = ['x', 'i', 'o', 'n', 'g', 'e', 'r'];  var tempStrArray = ['n', 'g'];  var replaceStrArray = ['t', 't'];  var str = new MyString(strArray);  console.log(str);  var newStr = str.strCopy();  console.log(newStr);  console.log(newStr == str);  console.log(newStr === str);  console.log(newStr.strCompare(otherStrArray));  newStr = str.concat(otherStrArray);  console.log(newStr);  // var strArray = 'xionger'.split('');  // console.log(strArray);  // console.log(strArray.length);  newStr = str.subString(2, 2);  console.log(newStr);  // str.strDelete(4, 2);  // console.log(str);  // str.strInsert(7, tempStrArray);  // console.log(str);  var index = str.index(tempStrArray, 2);  console.log(index);  // str.replace(tempStrArray, replaceStrArray);  // console.log(str); |

KMP模式匹配算法：可以看到之前的朴素模式匹配算法的时间复杂度为O(n2)，效率比较低下，因此更合适的方式是由Knuth,Morris,Pratt三位科学家研究的模式匹配算法，可以大大的减少重复遍历的情况。在之前的朴素模式匹配的算法中存在两个嵌套的循环，外部循环不存在回溯，但存在重复比较，内部循环存在回溯。因此，可以想到的优化点就是内外循环的变量值得设置。这部分用文字表述将非常的复杂且不容易理解，接下来通过代码让我们来熟悉这部分知识，也是本文最难的部分。

借用博文<http://www.cnblogs.com/c-cloud/p/3224788.html>的解释，非常准确，有空时可以继续深入学习。