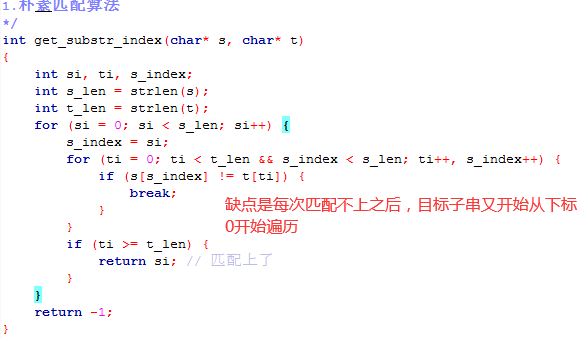
# 一：字符串匹配之KMP算法

1. 朴素匹配算法

计算子串t在主串s中的位置



1. KMP算法

## 【概述】

KMP算法目的在于当s[i] 与t[j]匹配时当出现某个位置不匹配时不会导致t[j]中的j从子串的位置0开始重新遍历，而应该根据t自身的关系可以从直接让j从某个位置直接开始遍历。

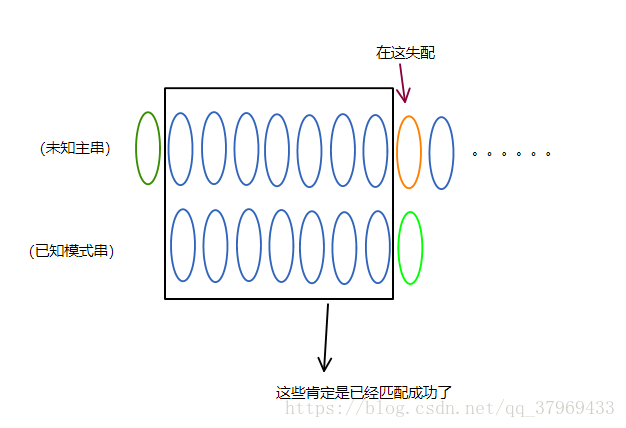
比如：

S:MABCDABCH

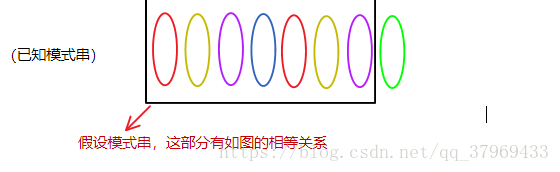
T:ABCDABCG

## 【总体分析】

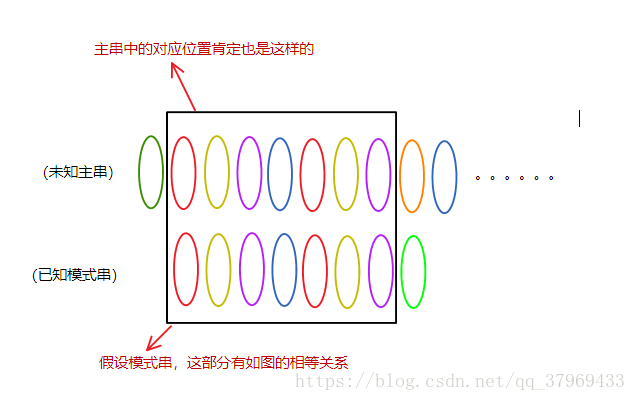
1. 首先比较时在图中所示位置失配，可以确定的是框起来的部分是一一相等的。



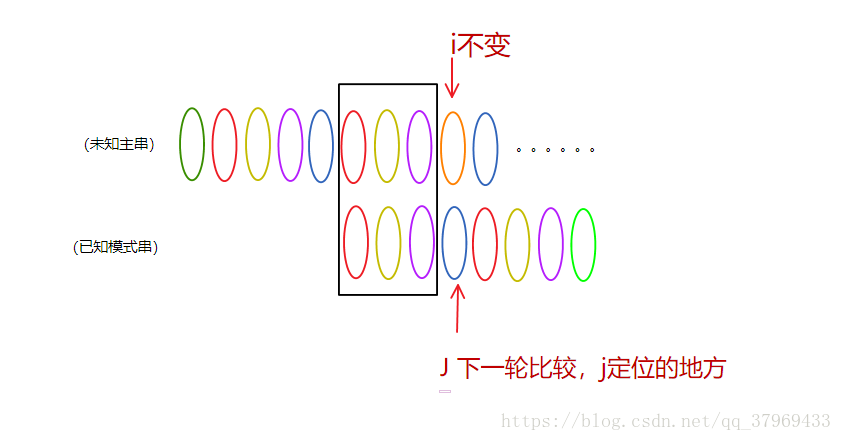
1. 假设子串中前缀和后缀有重合关系的时候，如



那么可以推出主串也有相等关系



1. 因此当继续遍历的时候，主串的i不变，子串j从偏移 “前后缀重复最长子串的长度开始遍历”



## 【结论】

为定义下一次j偏移的位置，定义next[j],表示当自串t[j]开始不匹配是，t应该从哪个序号开始重新遍历比较，因此问题关键就是如果根据子串求出其next数组。

## 【next数组算法—重点难点】

next[j]表示在比较时当位置t[j]不匹配j的新位置是什么，也就是子串t[j]之前t[0] – t[j - 1]的前后缀最长重合长度。如何求出next[j]?

1. 当t[0] t[1]…t[k-1]与t[j – k - 1] …t[j - 1] 一一相等并且t[k] = t[j], 则next[j] = k，k是重合的最大值。
2. 当t[0] t[1]…t[k-1]与t[j – k - 1] …t[j - 1]一一相等，但是t[k] != t[j],则只能忘回找最大的重合处去判断t[k] 是否和t[j]相等，那新的k就是next[k],因为k总是相对0开始的，并且next就代表坐标。
3. next[0] next[1]总是为0的，方便计算和编码，初始化指定next[0] = -1。

【编码】

void get\_patten\_next(char\* p, int\* next)

{

    int i = 0;

    int k = 0;

    int p\_len = strlen(p);

    next[0] = -1; // 为计算方便，设置为-1

    next[1] = 0;

    i = 2;

    while (i < p\_len) {

        if (k == -1) {

            next[i] = 0;

            k++;

            i++;

        }

        if (p[i - 1] == p[k]) { // k是p[i - 1]的最长前后缀重合串的子串长度,也是表示前缀p[k]

            next[i] = k + 1;

            k++;

            i++;

        } else {

            k = next[k];

        }

    }

}

## 【匹配应用】

int KMP\_get\_substr\_index(char\* s, char\* t)

{

    int si = 0, ti = 0, s\_index;

    int s\_len = strlen(s);

    int t\_len = strlen(t);

    int i;

    int next[100] = {0};

    get\_patten\_next(t, next);

    for (i = 0; i < strlen(t); i++) {

        printf("%d ", next[i]);

    }

    while (si < s\_len && ti < t\_len) {

        if (ti == -1 || s[si] == t[ti]) {

            si++;

            ti++;

        } else {

            ti = next[ti];

        }

    }

    if (ti >= t\_len) {

        return si - t\_len;

    } else {

        return -1;

    }

}