猪脚说第五期

链表经典算法

由于同学们反馈对链表的相关操作还不是很熟练，所以我在这里为大家找了一些经典的有关链表的算法，所有题目都是leetcode网站上的题目。

leetcode简介

https://leetcode.cn/是一个oj刷题网站，里面的题目大多是给一个需求，让用户设计一个函数，省略了一道完整的题目的输入输出部分，让用户可以更专注于算法的实现。其他的信息大家可以自行去网上搜索了解。

203. 移除链表元素

（注：上面的题号是超链接，大家可以点进去看题）

思路分析

本题是链表最基础的删除操作，只不过本题要删除的结点可能不止一个。我想本题需要注意的点大概有这么几个：

1. 如果删除的是头结点，我们的删除操作会和正常的结点不同——这一点大家在完成第三次作业的时候，以及理论课上课的时候应该都体会到了。
2. 基于第一条，由于本题要删除的结点可能不止一个，头部也可能有多个要删除的结点挨着，所以我们可以先将头部的所有符合条件的结点删去，**直到新的头结点不符合删除条件。**
3. 由于本题的链表是单向链表，所以我们无法通过当前结点找到上一个结点。所以当我们遍历结点时，如果我们将遍历到的当前结点作为判断删除的对象的话，我们还要从头开始遍历一次找到该结点的上一个结点——这无疑徒增麻烦。所以可以在设计时，判断遍历到的当前结点的下一个结点是否要被删除。
4. 既然头结点的删除会有不同，那我们能不能通过一些技巧去让所有结点的操作统一起来呢？大家自己先思考一下，我在本题的最后会解开谜底。

这么说可能有点抽象，大家看下面的代码就理解了。

具体实现过程

我们要先循环删除头部的结点——直到头部的结点不符合删除条件。同时要注意特判头结点不为`NULL`——即链表删干净了的情况：

```
// 先删除头结点
    while (head != NULL && head->val == val) {
        // 当前的头结点满足删除条件
        struct ListNode* tmp = head;  // 暂时记录下当前的头结点，目的是为了后面free
        head = head->next;  // 将头结点向后移动一位
        free(tmp);  // 这里就体现出tmp记录的用处了，不然的话现在就找不到上一个head了
    }
```

这里的思路我在注释里写的已经比较详细了。需要注意的一点是：大家现在写程序时就算不`free`一般也不会有什么问题，但是如果以后写的大型软件出现内存泄漏的话可能会造成比较严重的后果，所以`free`是一个好习惯。

但是有很多bug也是由`free`造成的，比如有的同学将同一块内存区域`free`了两次，在一些IDE的Debug模式下可能会报错（Release模式一般不会有这个问题）；或者是错误地释放了某块内存，导致后来访问这块内存时相当于访问了一块野指针所指的内容等等，所以在free时还是要谨慎。

接下来我们要删除后面的结点，根据上面的思路分析，可以知道我们应当判断的是当前遍历到的结点的下一个结点是否要被删除，故循环的终止条件应该是`cur->next == NULL`：

```
    // 删除后面的结点
    struct ListNode* cur = head;  // 当前遍历到的结点
    while (cur != NULL && cur->next != NULL) {
        // 这里判断cur != NULL 是为了防止第一步删除头结点时把结点删没了
        if (cur->next->val == val) {  // 删除下一个结点的条件
            struct ListNode* tmp = cur->next;  // 暂时记录下要删除的结点，以便free
            cur->next = cur->next->next;  // 改变当前结点的next域，实现删除
            free(tmp);  // 释放要删除的结点内存
        } else {
            cur = cur->next;
        }
    }
```

这里的删除操作大家自己画个图模拟一下就很好理解了。

最后按照要求返回新的头结点即可：

```
return head
```

完整代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeElements(struct ListNode* head, int val) {
    // 先删除头结点
    while (head != NULL && head->val == val) {
        // 当前的头结点满足删除条件
        struct ListNode* tmp = head;  // 暂时记录下当前的头结点，目的是为了后面free
        head = head->next;  // 将头结点向后移动一位
        free(tmp);  // 这里就体现出tmp记录的用处了，不然的话现在就找不到上一个head了
    }

    // 删除后面的结点
    struct ListNode* cur = head;  // 当前遍历到的结点
    while (cur != NULL && cur->next != NULL) {
        // 这里判断cur != NULL 是为了防止第一步删除头结点时把结点删没了
        if (cur->next->val == val) {  // 删除下一个结点的条件
            struct ListNode* tmp = cur->next;  // 暂时记录下要删除的结点，以便free
            cur->next = cur->next->next;  // 改变当前结点的next域，实现删除
            free(tmp);  // 释放要删除的结点内存
        } else {
            cur = cur->next;
        }

    }
    return head;
}
```

统一的思路——哑结点

在上一期猪脚说里，我和大家介绍过**哑结点**，大家理论课上应该也接触过哑结点，这里我就不做过多的原理阐述（如果大家不懂，可以百度上搜索，或者直接私信助教），直接讲他在本题中的用途。

由于本题不是一道完整的编程题，只是让我们实现一个函数，所以我们要主动地在函数内部给他加上一个头部的哑结点——而大家如果在完整的编程题里实现，完全可以在创建链表时就采用哑结点的方法去建。

首先创建一个虚拟头结点：

```
    // 创建一个虚拟头结点
    struct ListNode* virtual_head = (struct ListNode*) malloc(sizeof(struct
ListNode));
    virtual_head->next = head;  // 虚拟头结点的next域指向真正的头结点head
```

接下来请大家自己画个图看一下当前我们要删除的结点如果是真正的头结点的话，应该怎样操作——你会惊奇地发现和正常的结点是一样的！因为头部永远有一个虚拟头结点，所以下面删除的操作就统一了（而且很自然地想到：我们无需特判cur == NULL的情况了）：

```
    // 删除
    struct ListNode* cur = virtual_head;
    while (cur->next != NULL) {
        if (cur->next->val == val) {  // 要删除下一个结点
            struct ListNode* tmp = cur->next;
            cur->next = cur->next->next;
            free(tmp);
        } else {
            cur = cur->next;
        }
    }
```

由于本题最后要返回真实的头结点，所以我们就把`virtual_head->next`返回就行（顺便把`virtual_head`释放了是一个好习惯，但是一定不能先释放，因为如果释放了，就找不到真正的头结点了，所以要记录下结果再释放）：

```
    struct ListNode* res = virtual_head->next;
    free(virtual_head);
    return res;
```

完整代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeElements(struct ListNode* head, int val) {
    // 创建一个虚拟头结点
    struct ListNode* virtual_head = (struct ListNode*) malloc(sizeof(struct
ListNode));
    virtual_head->next = head;  // 虚拟头结点的next域指向真正的头结点head
```

```
    // 删除
    struct ListNode* cur = virtual_head;
    while (cur->next != NULL) {
        if (cur->next->val == val) {  // 要删除下一个结点
            struct ListNode* tmp = cur->next;
            cur->next = cur->next->next;
            free(tmp);
        } else {
            cur = cur->next;
        }
    }

    struct ListNode* res = virtual_head->next;
    free(virtual_head);
    return res;
}
```

总结

本题中，大家非常直观地感受到了有没有哑结点的差异——不仅删除时操作统一，而且也不用特判`cur =!= NULL`，可以说非常方便，我本人非常喜欢这种哑结点的做法，也推荐给大家去使用。

同时我们看到了`free`可能引起的错误，有的同学可能在哑结点版本里最后直接写`free(virtual_head); return virtual_head->next`，就会报错——因为`free`过后`virtual_head`就成为了野指针，所以这方面大家要多加注意。

练习题！[707. 设计链表]

本题留作大家的练习题，非常基础，大家可以尝试着用我刚刚教过的哑结点技巧来实现。

[206. 反转链表]

思路分析

如果再定义一个新的链表，实现链表元素的反转，其实这是对内存空间的浪费。

其实只需要改变链表的next指针的指向，直接将链表反转 ，而不用重新定义一个新的链表（应该比较好理解吧）。

本题就涉及到一个技巧：由于我们要把当前结点的`next`域指向其前一个元素，而且又是单链表，所以就要记录其上一个结点的指针，好让我们能找到它。

这里我采用`cur`记录当前遍历到的结点，用`pre`记录`cur`的前一个结点，在循环中不断更新这两个指针的值。

具体实现过程

首先初始化`pre`和`cur`的值：

```
    struct ListNode* cur = head;
    struct ListNode* pre = NULL;
```

然后显然要循环遍历，由于我们的`cur`的含义是遍历到的当前结点，所以结束条件应当是`cur == NULL`。

对于每一个结点的翻转，应该按顺序实现这几步：

1. 先记录下`cur->next`——因为我们改变了`cur->next`之后就找不到其原始的下一个结点了；
2. 实现翻转：`cur->next = pre`；
3. 更新`cur`和`pre`的值。

```
    while (cur != NULL) {
        struct ListNode* tmp = cur->next;  // 暂时记录下cur->next的值
        cur->next = pre;  // 改变指针指向
        pre = cur;  // 更新pre
        cur = tmp;  // 更新cur
    }
```

最后返回新的头结点即可——遍历结束后`cur`的值是`NULL`，显然新的头结点就是`pre`：

```
return pre;
```

完整代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* cur = head;
    struct ListNode* pre = NULL;

    while (cur != NULL) {
        struct ListNode* tmp = cur->next;  // 暂时记录下cur->next的值
        cur->next = pre;  // 改变指针指向
        pre = cur;  // 更新pre
        cur = tmp;  // 更新cur
    }
    return pre;
}
```

19. 删除链表的倒数第 N 个结点

思路分析

本题的思路我在“23级程序设计基础训练”里面提到过，是双指针的经典例题。大多数人的第一反应是先遍历一遍求出该链表的长度，然后再遍历得去删除，但其实我们并不需要求出其长度即可完成操作：

双指针的思路是：设置一个快指针`fast`和一个慢指针`slow`，初始化均指向头结点，先让快指针移动`n`次，如果过程中到了链表尾就说明链表长度没有n，然后再让快慢指针一起移动，直到快指针到尾，此时删除慢指针所指的结点即可。

还需要注意的一点是，由于本题涉及删除操作，所以正常而言需要特判头结点的情况（感觉还是比较烦这种特判的），所以仍然可以采用虚拟头结点（哑结点）的技巧来实现本题：

具体实现过程

首先设置哑结点：

```
    // 设置哑结点
    struct ListNode* virtual_head = (struct ListNode*) malloc(sizeof(struct
ListNode));
    virtual_head->next = head;
```

然后设置快慢指针。并让`fast`先走n步：

```
    struct ListNode* slow = virtual_head;
    struct ListNode* fast = virtual_head;
    while (n && fast != NULL) {
        n--;
        fast = fast->next;
    }
```

此时有一个细节，就是我们要让`fast`再向前移动一位：目的是让最后我们得到的`slow`的位置是要删除的结点的前一个结点（不然的话，还要遍历一遍链表找到`slow`的前一个结点）：

```
fast = fast->next;
```

接着，我们让`fast`和`slow`一起向前移动，直到`fast`为`NULL`：

```
    // 一起移动
    while (fast != NULL) {
        fast = fast->next;
        slow = slow->next;
    }
```

最后删除掉`slow`所指的结点的下一个结点：

```
    struct ListNode* tmp = slow->next;
    slow->next = slow->next->next;
    free(tmp);
```

然后返回新的头结点（虚拟结点的下一个），同时，顺手把`virtual_head`释放了是一个好习惯：

```
    struct ListNode* res = virtual_head->next;
    free(virtual_head);
    return res;
```

完整代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
```

```
 */
struct ListNode* removeNthFromEnd(struct ListNode* head, int n) {
    // 设置哑结点
    struct ListNode* virtual_head = (struct ListNode*) malloc(sizeof(struct
ListNode));
    virtual_head->next = head;

    struct ListNode* slow = virtual_head;
    struct ListNode* fast = virtual_head;
    while (n && fast != NULL) {
        n--;
        fast = fast->next;
    }
    fast = fast->next;

    // 一起移动
    while (fast != NULL) {
        fast = fast->next;
        slow = slow->next;
    }

    struct ListNode* tmp = slow->next;
    slow->next = slow->next->next;
    free(tmp);

    struct ListNode* res = virtual_head->next;
    free(virtual_head);
    return res;

}
```

总结

本题我仍旧采用了虚拟头结点的技巧，**强烈建议**大家自己再写一遍不带这种哑结点的版本，体会其中的不同之处。

同时，本题的算法思路很简单，但是实现起来细节还是不少，也**强烈建议**大家看完题解后自己再重新手敲一遍。

160. 相交链表

思路分析

本题的意思是两条链表从某个相交的结点开始，后面的结点都相同了，要我们找出那个相交的结点。

大家后面会学到二叉，其中就有一个经典的算法：最近公共祖先。如果二叉树使用的是三叉链表方式来构建的话，其实就还原成本题了（不懂的话，就先当没看见这句话）

本题的想法很简单，但是如果没见过的话还确实不太容易一下想出来。我这里直接给出思路：

由于两表相交节点之后的结点都是一样的，所以后半部分的长度是相同的，唯一不同的就是前面的长度。所以我们可以先分别遍历两条链表，求出两表的长度，再求出长度之差。

之后设置两个指针分别指向两表头部，先让较长的链表的指针移动长度之差个结点，然后让两指针一起移动，直到两指针所指向的结点相同。

可以发现，本题其实还是双指针的思路，可见在线性表里面双指针的设计还是比较常见的。

具体实现过程

首先求出两表的长度：

```
    int len1 = 0, len2 = 0;  // 分别记录两表的长度
    struct ListNode* t = headA;
    while (t) {
        t = t->next;
        len1++;
    }
    t = headB;
    while(t) {
        t = t->next;
        len2++;
    }
```

然后分别设置两个指针指向两表头，并让较长的链表的指针先移动`abs(len1 - len2)`个长度：

```
    // 双指针
    struct ListNode* p1 = headA, *p2 = headB;
    if (len1 < len2) {
        // B表长，让B移动
        for (int i = 0; i < len2 - len1; i++) p2 = p2->next;
    } else {
        for (int i = 0; i < len1 - len2; i++) p1 = p1->next;
    }
```

接着，一起移动两指针，直到其指向的地址相同：

```
    while (p1 && p2) {
        if (p1 == p2) return p1;
        p1 = p1->next;
        p2 = p2->next;
    }
    // default return null
    return NULL;
```

完整代码

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode *getIntersectionNode(struct ListNode *headA, struct ListNode
*headB) {
    int len1 = 0, len2 = 0;  // 分别记录两表的长度
    struct ListNode* t = headA;
    while (t) {
        t = t->next;
        len1++;
```

```
    }
    t = headB;
    while(t) {
        t = t->next;
        len2++;
    }

    // 双指针
    struct ListNode* p1 = headA, *p2 = headB;
    if (len1 < len2) {
        // B表长，让B移动
        for (int i = 0; i < len2 - len1; i++) p2 = p2->next;
    } else {
        for (int i = 0; i < len1 - len2; i++) p1 = p1->next;
    }

    while (p1 && p2) {
        if (p1 == p2) return p1;
        p1 = p1->next;
        p2 = p2->next;
    }
    // default return null
    return NULL;
}
```

总结

本次讲解了几道链表的经典算法题目，希望对大家有帮助。

同时，由于涉及指针操作，建议大家看懂之后仍自己手敲一遍，感受一下其中的细节——光看会可不一定是真的会了。

23级程序设计基础训练题解

15. 文件排版（非文件）

思路分析

首先虽然规则中没有说明，但是通过样例，我们可以看出需要将连续的空格和制表符合并为一个空格；假设我们将处理前读入的字符串存入 `str` 中；要把处理后 `:` 前的串存入 `str2` ，处理后 `:` 后的串存入 `str3`，那么我们就可以通过遍历 `str` 的每一位，当遇到空格或者制表符时，如果当前 `str2` 或者 `str3` 的最后一位是空格（你遇到 `:` 前处理的就是 `str2`，遇到 `:` 后处理的就是 `str3`），那么我们就忽略这一位，开始枚举下一位；否则就在 `str2` 或者 `str3` 的末尾存入空格。这样就能合并连续的空格和制表符，变成一个空格。

我们得到 `str2` 和 `str3` 后，可以对 `str2` 按照指定位对齐格式化输出后，再输出 `: str3`。有的时候 `:` 前面和后面可能会有空格，有的时候则没有，为了统一处理，我们把最后要求输出时加在 `:` 前面的空格单独当做一个空格输出，而不是 `str2` 的一部分，也就是说处理完后如果 `str2` 最后有空格我们就把它去掉；把 `:` 的空格作为 `str3` 的一部分，也就是说在处理 `str3` 之间赋值 `str3[0]` 为空格。

最后要处理的问题是我们虽然知道应该把 `str2` 按照 `n - 2` 位右对齐输出（-2 是因为 `:` 和空格），但是 `n` 每次是个变量。实际上我们上学期程设也遇到过类似的问题，一种解决方法是用 `sprintf` 动态地改变格式串，即 `sprintf(geshi,"%-%%ds",n - 2);` ，其中负号代表右对齐。

参考代码

```
#include <stdio.h>
#include <string.h>

int read() {
    int tem;
    scanf("%d",&tem);
    return tem;
}

char str[10005]; // 原字符串
char str3[10005];// 处理后的 : 后的字符串
char str2[10005];// 处理后的 : 前的字符串
char geshi[10005];

int main() {

    int n = read();
    sprintf(geshi,"%-%%ds",n - 2);
    // 吃掉第一行有效语句前的所有无效字符
    while(getchar() != '\n');

    while(gets(str)) {
        memset(str2,0,sizeof(str2));
        memset(str3,0,sizeof(str3));
        int len = strlen(str);
        int pos = 0;
        int pos2 = 0, pos3 = 0; // 记录 str2 和 str3 枚举到的位置

        for(pos = 0;str[pos] != ':'; pos++) {
            // 合并空格和制表符为空格
            if(str[pos] == ' ' || str[pos] == '\t') {
                if(str2[pos2 - 1] != ' ')
                    str2[pos2++] = ' ';
            }
            else
                str2[pos2++] = str[pos];
        }
        if(str2[pos2 - 1] == ' ')
            str2[pos2 - 1] = 0;

        str3[pos3++] = ' ';

        pos++;  // 跳过 :
        for(;str[pos];pos++) {
            // 合并空格和制表符为空格
            if(str[pos] == ' ' || str[pos] == '\t') {
                if(str3[pos3 - 1] != ' ')
                    str3[pos3++] = ' ';
            }
            else
                str3[pos3++] = str[pos];
        }
```

```
        printf(geshi,str2);
        printf(" :%s\n",str3);
    }

    return 0;
}
```

16. 注释比例

思路分析

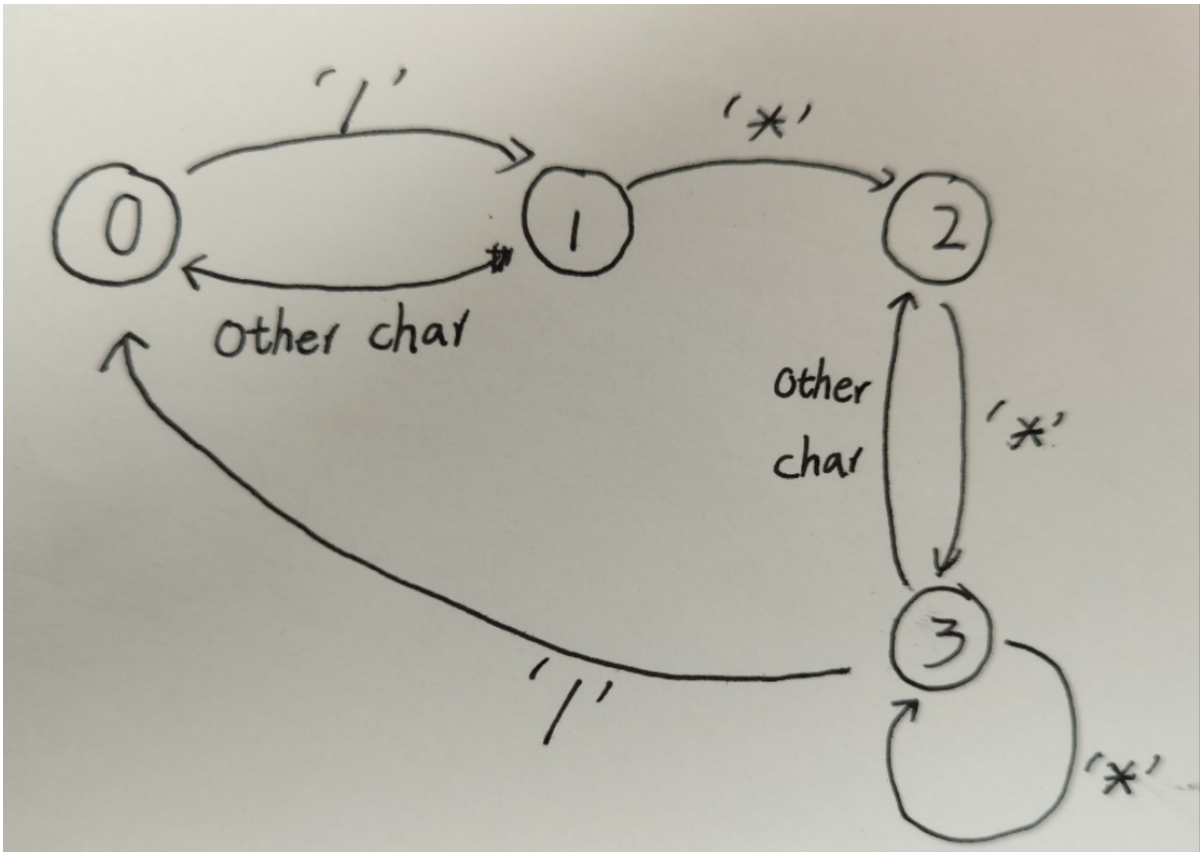
这道题无非是统计出一共有多少字符，多行注释内有多少字符。一共有多少字符很好统计，我们可以一个字符一个字符读入，每读入一个字符就把当前的总字符个数 + 1。那么该如何统计注释内有多少字符呢？其实跟统计一共有多少字符差不多，只不过我们需要知道当前是在多行注释内，再增加当前注释内的总字符个数 + 1。分别标记注释内的总字符个数为 `zhushi`，总的字符个数为 `total`。

假设我们定义一个变量 `sta`，当这个变量的值为 2 时表示我们在多行注释中，其它值为不在多行注释中，那么很显然当 $sta = 2$ 时，我们同时增加 `zhushi` 和 `total` 的值；当 $sta \neq 2$ 时，我们只增加 `total` 的值。

当程序开始时，我们让 `sta` 的值为 0，标记这是最一般的状态。那么什么时候有可能进入到多行注释呢？实际上，当我们在 `sta = 0` 时，遇到一个 `/`，我们就要小心了，因为如果下一个字符是 `*`，那么我们就进入了一个多行注释。所以，当我们在状态 `0` 下，读入了一个 `/`，那么我们就可以让 `sta = 1`，标记这是一个危险的状态。当 `sta = 1` 时，如果我们新读入的字符是 `*`，就证明我们真的进入了一个多行注释，标记 `sta = 2`；否则，如果是其它字符，证明我们刚才不过是虚惊一场，可以把 `sta` 重新赋值为 0，代表我们回到了一个普通状态，等待下一个 `/` 的出现。（因为按照题意，不会有单行注释，所以我们不必担心紧跟着 `/` 的这个字符还是 `/`）。

同样，在多行注释中时，我们还要时刻关注何时多行注释结束了。当我们遇到一个 `*` 时，如果下一个字符是 `/`，那么我们就离开多行注释了。所以当 `sta = 2` 时，当前读入字符为 `*`，我们可以让 `sta = 3`，表示准备离开多行注释的状态。当 `sta = 3`，且当前读入字符为 `/` 时，表示我们真的离开了一个多行注释，回到普通状态，重新让 `sta = 1`；否则证明我们只是虚惊一场，还是在注释内，让 `sta` 回到 2，标记在多行注释内。不过这里要注意，我们在 `sta = 3` 时可能还读入一个 `*`，这个 `*` 仍然可能是一个多行注释结束的表示，所以当 `sta = 3` 且读入字符是 `*` 时，我们仍要保持 `sta = 3`，这是一个非常容易错的地方（第四次作业括号匹配的题目可能就有这种风险）。

这就是状态转移的思想，在第四次作业中我们可以用这种思想去处理单行注释、多行注释、字符常量、字符串常量，用一张图表达这样的思想就是这样的：



最后输出 `%` 别忘了转义

参考代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    int total = 0, zhushi = 0;
    int sta = 0;
    char ch;    // 用来读入新的字符
    while(scanf("%c",&ch) != EOF) {
        total++;

        // 因为 * 可能是多行注释结束的标志，不作为多行注释的一部分
        if(sta == 2 && ch != '*')
            zhushi++;

        if(sta == 0 && ch == '/')   sta = 1;
        else if(sta == 1 && ch == '*')  sta = 2;
        else if(sta == 2 && ch == '*')  sta = 3;
        else if(sta == 3 && ch == '/')  sta = 0;
        else if(sta == 3 && ch == '*')  zhushi++;   // 证明我们上一个 * 时注释的一部分
        else if(sta == 3)   sta = 2;    // 虚惊一场，仍在注释中
    }

    printf("%d%%",zhushi * 100 / total);

    return 0;
}
```

17. 删除子串

思路分析

直接最朴素的思想，枚举原字符串的每一位，表示可能与子串匹配的起始位置。用一个变量 `i` 标记枚举到原串的位置，如果第 `i` 位不等于子串的第一位，那么显然以 `i` 开始的串不会是子串，直接把第 i 位存到答案输出数组中；否则比较一下，看看以 `i` 开始的若干位（子串长度位）是不是子串，如果是的话，那么就跳过这一段，直接让 `i` 的值增加子串长度位，下次从子串长度位之后开始枚举。

也可以用 `strstr` 查找原串中子串的位置，如果原串中没有子串，直接输出原串；否则如果找到子串的位置是 `pos`，那么就将 `pos` 之前的所有位存入答案数组，再标记查找的起始位置为 `pos + 子串长度`，直到找不到子串，再将剩下的位存入答案数组。

给出的代码就是第二种实现方式。

参考代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef long long ll;

char str[10005];
char subStr[10005];
char ans[10005];

int main() {
    gets(str);gets(subStr);
    int pos = 0;
    int find = 0;
    int cnt = 0,len = strlen(subStr),len2 = strlen(str);

    while ((find = strstr(str + pos,subStr) - str) > -1) {
        for(int i = pos;i < find;i++)
            ans[++cnt] = str[i];
        pos = find + len;
    }

    for(int i = pos;i < len2;i++)
        ans[++cnt] = str[i];

    printf("%s",ans + 1);

    return 0;
}
```