《传智播客 C语言就业班》 第六讲 链表

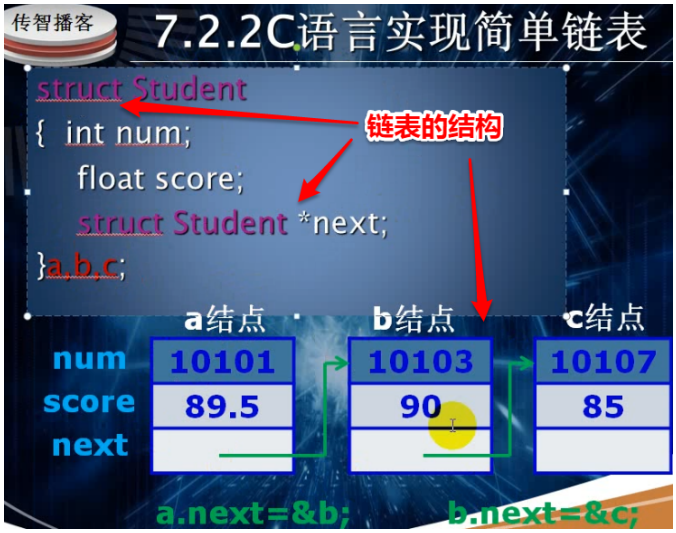
结构体中套结构体的问题引出：

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//结构体中套一个自己类型的结构体元素：error
//结构体中套一个自己类型的结构体指针：ok
//因为数据类型的本质：固定大小内存块的别名；
//要分配内存首先内存块大小必须要确定下来！
//因此结构体不能嵌套定义（分配不了内存！）
typedef struct AdvTeacher
{
	char name[64];
	int id;
	AdvTeacher* advTeacher;
}AdvTeacher;
```

- **结点**和节点的区别：

```
结，连结，终结
节，关节
可以这样做简单的区分
节点被认为是一个实体，有处理能力，比如说网络上的一台计算机；
结点则只是一个交叉点，像“结绳记事”，打个结，做个标记，仅此而已。
一般算法中点都是结点，而复杂网络理论中所谈到的点应该是“节点”了。
https://blog.csdn.net/qq_41603898/article/details/80546811
```

- 链表的定义：链表是一种物理存储单元（内存）上**非连续**的存储结构，由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成，结点与结点之间通过指针连接，每个结点包括两个部分：一部分是存储数据元素的数据域，另一部分是存储下一个结点地址的指针域。



- 链表的组织形式：单向链表、双向链表、循环链表。
- 链表分为带头结点的链表和不带头结点的链表。
- 链表编程元素分析：
  - **头结点pHead**、**当前结点**pCurrent（简写为**pCur**）、**前驱结点**pPrior（简写为**pPre**）、**后继结点pNext**
  - **新建结点**pMalloc（简写为**pM**）
- 链表变成关键点（**2点**）
  - 指针指向谁，就把谁的地址赋给指针（Code=Diagram 条件反射）
  - 辅助指针变量 & 操作逻辑的关系（看图说话）

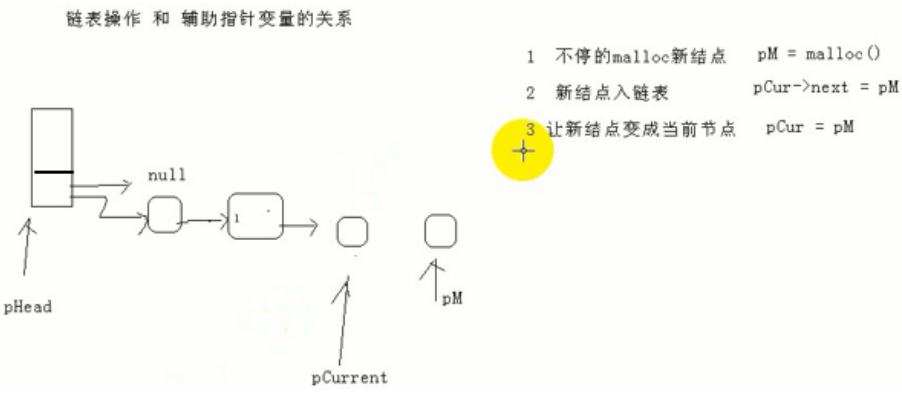
- **静态链表**：结点的个数固定，实际中很少用。

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
```

```
#include "stdlib.h"
#include "string.h"

typedef struct Teacher
{
    char name[64];
    int age;
    Teacher *next;
}Teacher;

//创建静态链表
Teacher* CreateLinkList()
{
    Teacher t1, t2, t3;
    t1.age = 33;
    t2.age = 22;
    t3.age = 11;
    t1.next = &t2;
    t2.next = &t3;
    t3.next = NULL;
    return &t1;
}

//遍历链表
void main()
{
    Teacher* p = CreateLinkList();  //静态链表不可以这样写！

    Teacher t1, t2, t3;
    t1.age = 33;
    t2.age = 22;
    t3.age = 11;
    t1.next = &t2;
    t2.next = &t3;
    t3.next = NULL;

    Teacher *p = &t1;
    while(p)
    {
        printf("age:%d\n", p->age);
        p = p->next;
    }
    return;
}
```

静态链表局限性：**只能在main函数而不能在临时区创建链表，因为临时区内存空间会在函数结束释放，甩不出来**，因此需要用malloc的方式动态创建链表。

- **动态链表**：
  - **创建链表思路**（见下图）



- **创建链表代码实现**

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

typedef struct Node
{
    int data;
    struct Node *next;
}SLIST;

//建立带有头结点的单向链表
```

```
//要求：循环创建结点，结点数据域中的数值从键盘输入，以-1作为结束标志。
//返回头结点地址。
SLIST* SList_Create()
{
/*
    创建链表的思路（非常重要！）：
    1、不停地malloc新结点
    2、新结点入链表
    3、让新结点变成当前结点
*/
    //我的代码：
    //int num = 0;
    //SLIST *pHead = (SLIST*)malloc(sizeof(SLIST));
    //SLIST *pCurr = pHead;
    //SLIST *pPre = pCurr;
    //while (num != -1)
    //{
    //    printf("please input a number(-1 to end):");
    //    scanf("%d", &num);
    //    if (num != -1)
    //    {
    //        SLIST* pCurr = (SLIST*)malloc(sizeof(SLIST));
    //        pPre->next = pCurr;
    //        pCurr->data = num;
    //        pCurr->next = NULL;
    //        pPre = pCurr;
    //    }
    //    else
    //        break;
    //}
    //
    //return pHead;

    //王保明老师的代码（值得学习，对照着步骤1、2、3看对应的代码）：
    SLIST *pHead = NULL, *pM = NULL, *pCur = NULL;
    int data;
    pHead = (SLIST *)malloc(sizeof(SLIST));
    if (pHead == NULL)
    {


        return NULL;
    }
    pHead->data = 0;
    pHead->next = NULL;
    printf("please enter your data:");
    scanf("%d", &data);

    // 循环创建
    // 初始化当前结点，指向头结点
    pCur = pHead;
    while (data != -1)
    {
        //1、不断的malloc 新的业务结点 ==> PM
        pM = (SLIST *)malloc(sizeof(SLIST));
        if (pM == NULL)
        {
            SList_Destory(pHead); //如果某个结点处malloc失败了，需要把之前malloc的内存free掉，
                                  //没有内存泄漏，注意对代码的控制能 ！
            return NULL;
        }
        pM->data = data;
        pM->next = NULL;

        //2、新结点入链表
        pCur->next = pM;

        //3、让新结点变成当前结点
        pCur = pM; //链表结点的尾部追加
        printf("\nplease enter the data of node(-1:quit) ");
        scanf("%d", &data);
    }
    return pHead;
}
```

- **创建链表代码实现（改进版：1、对有多个出口，即多次return的函数进行优化为只有1个出口 2、指针做函数参数）**

```
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
typedef struct Node
```

```
{
    int data;
    struct Node *next;
}SLIST;
//编写函数SList_Create，建立带有头结点的单向链表。循环创建结点，
//结点数据域中的数值从键盘输入，以-1作为输入结束标志。链表的头结点地址由函数值返回。
SLIST *SList_Create();
int SList_Print(SLIST *pHead);
int SList_NodeInsert(SLIST *pHead, int x, int y);
int SList_NodeDel(SLIST *pHead, int x);
int SList_Destory(SLIST *pHead);
int SList_Reserve(SLIST *pHead);


//SList_Create()第2种写法：
//注意对比上面的代码，看好在哪里！这样的程序更加健壮！
//将链表头结点作为函数参数传入
int SList_Create(SLIST **mypHead)
{
    int ret = 0;
    SLIST *pHead = NULL, *pM = NULL, *pCur = NULL;
    int data = 0;
    //1 创建头结点并初始化
    pHead = (SLIST *)malloc(sizeof(SLIST));
    if (pHead == NULL)
    {
        ret = -1;
        printf("func SList_Create() err:%d ", ret);
        goto End;  //最好不要直接return ret，因为如果前面还有malloc
                   //需要释放该内存；因此最好将函数各失败的地方统一到一个出口
    }
    pHead->data = 0;
    pHead->next = NULL;
    //2 从键盘输入数据，创建业务结点
    printf("\nplease enter the data of node(-1:quit) ");
    scanf("%d", &data);
    //3 循环创建
    //初始化当前结点，指向头结点
    pCur = pHead;
    while (data != -1)
    {
        //新建业务结点 并初始化
        //1 不断的malloc 新的业务结点 ===PM
        pM = (SLIST *)malloc(sizeof(SLIST));
        if (pM == NULL)
        {
            SList_Destory(pHead);
                //比如某个结点处malloc失败了，需要把之前malloc的内存free掉
                //不要有内存泄漏，对代码的控制能力！
            ret = -2;
            printf("func SList_Create() err:%d ", ret);
            goto  End;
        }
        pM->data = data;
        pM->next = NULL;
        //2、让pM结点入链表
        pCur->next = pM;
        //3 pM结点变成当前结点
        pCur = pM; //pCur = pCur->next;
                   //2 从键盘输入数据，创建业务结点
        printf("\nplease enter the data of node(-1:quit) ");
        scanf("%d", &data);
    }
END:
     if(ret != 0)
     {
        SList_Destroy(pHead);
     }
     else
     {
         *mypHead = pHead;
     }
    return ret;
}


void main()
{
    SLIST *pList = NULL;
    pList = SList_Create();
}
```

调试过程

监视 1

名称	值	类型
data	-1	int
pHead	0x00ab6000 {data=0 next=0x00ab6038 {data=111 next	Node *
data	0	int
next	0x00ab6038 {data=111 next=0x00ab97a8 {data=222 ne	Node *
data	111	int
next	0x00ab97a8 {data=222 next=0x00ab97e0 {data=3333 r	Node *
data	222	int
next	0x00ab97e0 {data=3333 next=0x00000000 <NULL> }	Node *
data	3333	int
next	0x00000000 <NULL>	Node *

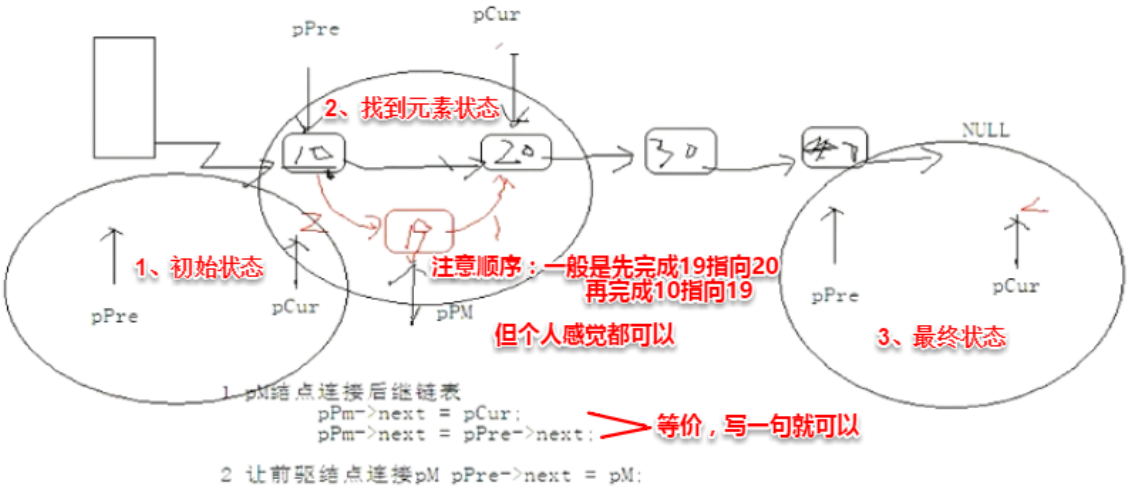
打印链表代码实现

```
int SList_Print(SLIST *pHead)
{
    SLIST *p = NULL;
    if (pHead == NULL)
    {
        return -1;
    }
    p = pHead->next;
    printf("\nBegin \n");
    while (p)
    {
        printf("%d \n", p->data);
        p = p->next;
    }
    printf("End \n");
    return 0;
}
```

销毁整个链表代码实现

```
int SList_Destory(SLIST *pHead)
{
    if (pHead == NULL)
    {
        return -1;
    }
    SLIST *p = NULL, *tmp = NULL;
    p = pHead;
    while (p)
    {
        //下一个结点的位置，保存在前驱结点next域中
        //因此删除当前结点之前，需要缓存后继结点位置！
        tmp = p->next;
        free(p);//删除当前结点
        p = tmp; //结点指针后移
    }
    return 0;
}
```

插入链表思路



插入链表代码实现1

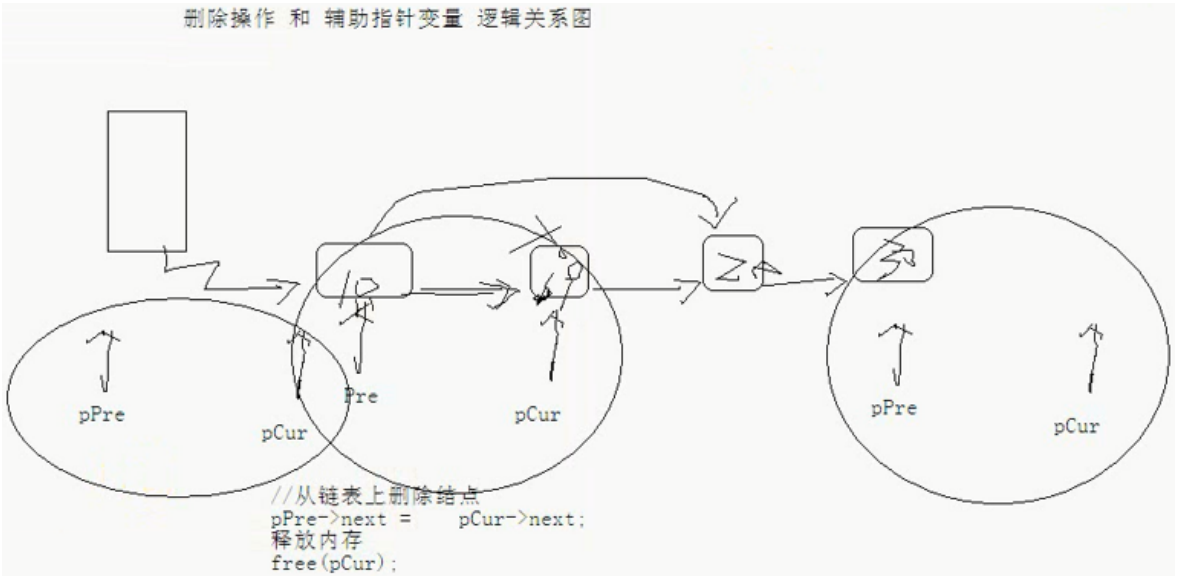
```
//在单向链表中插入节点
int SList_NodeInsert(SLIST *pHead, int x, int y)//在值为x的结点前插入值y
```

```
{
    if (pHead == NULL)
        return -1;
    SLIST *pPre = pHead;
    SLIST *pCur = pHead->next;
    int count = 0;
    while (pCur)
    {
        if (pCur->data == x)
        {
            break;
        }
        pPre = pCur;
        pCur = pCur->next;
        count++;
    }
    SLIST *pM = (SLIST *)malloc(sizeof(SLIST));
    pM->data = y;
    //1 让新结点连接后续链表
    pM->next = pPre->next;
    //2 让前驱结点连接新结点
    pPre->next = pM;

    ////个人认为其实也可以
    ////1 先让前驱结点连接新结点
    //pPre->next = pM;
    ////2 让新结点连接后续链表
    //pM->next = pCur;

    return count;
}
```

插入链表代码实现2

```
//功能：在值为x的结点前，插入值为y的结点；若值为x的结点不存在，则插在表尾。
int SList_NodeInsert(SLIST *pHead, int x, int y)
{
    SLIST *pCur = NULL, *pPre = NULL, *pM = NULL;
    //根据y的值malloc新结点
    pM = (SLIST *)malloc(sizeof(SLIST));
    if (pM == NULL)
    {
        return -1;
    }
    pM->data = y;
    pM->next = NULL;
    //准备pCur Pre环境
    pPre = pHead;
    pCur = pHead->next;
    while (pCur)
    {
        if (pCur->data == x)
        {
            //插入操作，这里需要注意要while的外面
            //因为如果到最后都找不到x这个值，就要在链表最后插入
            //所以这里是break
            break;
        }
        pPre = pCur;  //让前驱结点后移
        pCur = pCur->next; //让当前结点后移
    }
    //1 pM结点连接后继链表
    //pM->next = pCur;   //和下面一句等价。另外即使没找到元素，即pCur==NULL，也适用
    pM->next = pPre->next;
    //2 让前驱结点连接pM
    pPre->next = pM;
    return 0;
}
```

- **删除链表某个结点思路**



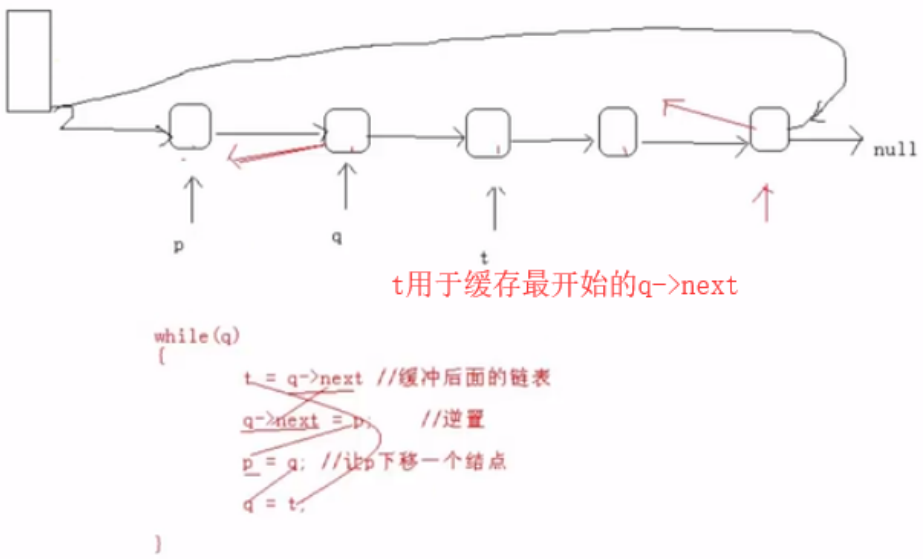
- **删除链表某个结点代码实现**

```
int SList_NodeDel(SLIST *pHead, int x)
{
    if (pHead == NULL)
       return -1;
    //准备pCur Pre环境
    SLIST *pPre = pHead;
    SLIST *pCur = pHead->next;
    while (pCur)
    {
        if (pCur->data == x)
        {
            //插入操作
            break;
        }
        pPre = pCur; //让前驱结点后移
        pCur = pCur->next; //让当前结点后移
    }
    if (pCur == NULL)
    {
        printf("没有找到要删除的结点\n");
        return -1;
    }

    pPre->next = pCur->next;//从链表上删除结点，另一种思路见下
    //释放内存
    free(pCur);
    return 0;
}
```

另一种思路：先缓存SLIST *ptmp = pCur; pCur = pCur->next再free(ptmp);

- **链表逆置的思路**

链表逆置也是一个结点一个结点的逆置。



- **链表逆置代码实现**

```
int SList_Reserve(SLIST *pHead)
{
    if (pHead == NULL || pHead->next == NULL ||pHead->next->next == NULL)
    {
        return -1;
    }
    SLIST *t = NULL;  //缓存的一个结点
    //初始化逆置环境
    SLIST *p = pHead->next;    //前驱指针初始化
    SLIST *q = pHead->next->next;  //当前指针初始化
    while (q)
    {
        //1、逆置之前保存后继结点
        t = q->next;
        //2、逆置操作
        q->next = p;
        //3、让前驱结点后移
        p = q;
        //4、让逆置结点后移
        q = t;
    }
    //逆置完成时，p指向最后一个节点，q指向NULL；
    //还需要进行最后两步操作
    pHead->next->next = NULL; //完成头结点指向的下一个结点指向NULL
                              //即位置修正；如果不修正是指向头结点的！
    pHead->next = p; //完成头结点指向p，即逆置前的最后一个结点

    return 0;
}
```

测试程序

```
void main()
{
    SLIST *pHead = NULL;
    pHead = SList_Create();
    SList_Print(pHead);
    SList_NodeInsert(pHead, 20, 19);
    SList_Print(pHead);
    SList_NodeInsert(pHead, 100, 99);
    SList_Print(pHead);
    SList_NodeDel(pHead, 19);
    SList_Print(pHead);
    SList_Reserve(pHead);
    SList_Print(pHead);
    SList_Destory(pHead);
}
```

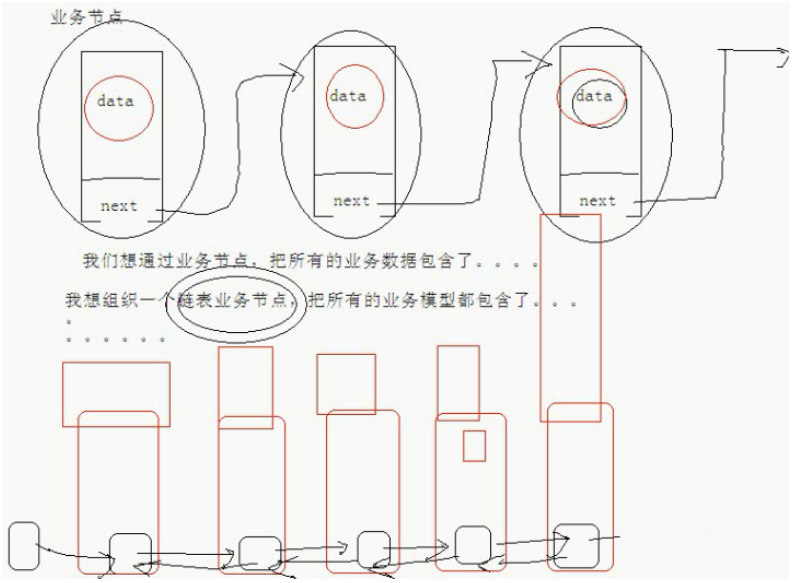
传统链表的思考

- **传统链表缺点**：
  - 和具体结构绑定，不通用；
  - 链表逻辑试图包含业务逻辑（数据）；在上面的例子中，链表里面包含了业务数据，如int data，char name[64]等

- 业务数据和链表逻辑耦合度太高。

“我包含不了万事万物，就让万事万物包含我”。

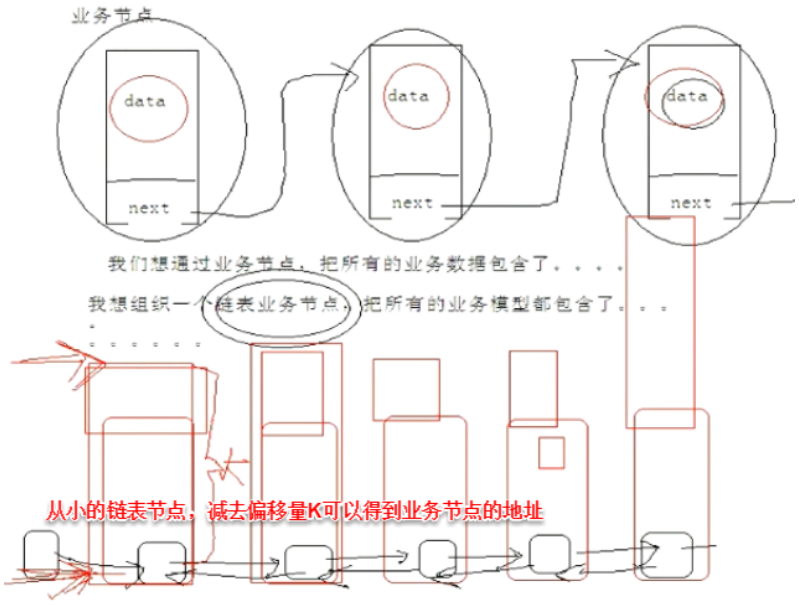
linux内核链表思想分析



实际业务中，链表中放数据放不下.....那就在数据中放链表 —— **内核链表**！

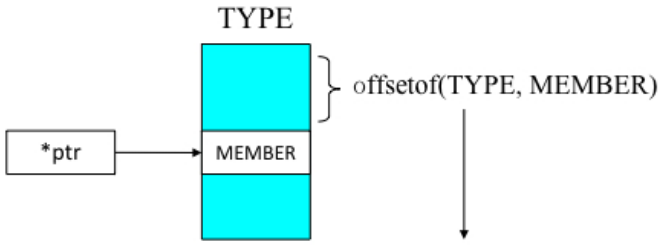
```
//问题的本质：链表的业务数据和链表的逻辑操作，没有有效的分离。。。。。
//产生问题的本质的原因：


typedef struct node
{
    struct node *next;
}node;

typedef struct Teacher
{
    int data;
    int det;
    int dd;
    char name[20];
    char name2[30]; //数据域
    char **p2;
    struct node mynode;
}Teacher;
```

另注static：起到本地化的作用
修饰变量——变量是静态变量；
修饰函数——表示函数不能跨文件引用，只能在这个文件引用。




```
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

我们来看看container_of是如何实现的。如下图所示，我们已经知道TYPE结构中MEMBER的地址，如果要得到这个结构体的地址，只需要知道MEMBER在结构体中的偏移量就可以了。如何得到这个偏移量地址呢？这里用到C语言的一个小技巧，我们不妨把结构体投影到地址为o的地方，那么成员的绝对地址就是偏移量。得到偏移量之后，再根据ptr指针指向的地址，就可以很容易的计算出结构体的地址。




```
#define offsetof(TYPE, MEMBER)  ((size_t) &((TYPE *)0)->MEMBER)
```

```
#define container_of(ptr, type, member) (type *)((char *)ptr -offsetof(type,member))
```

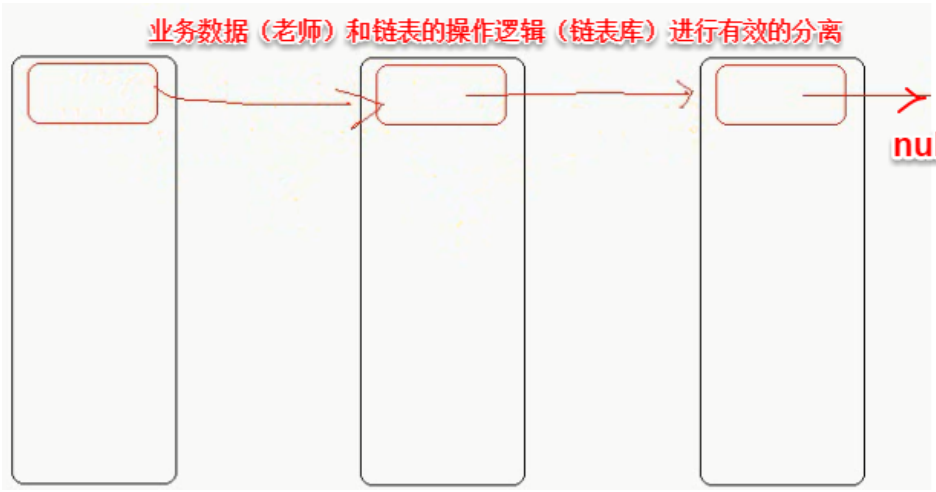
list_entry就是通过上面的方法从ptr指针得到我们需要的type结构体。

总结：

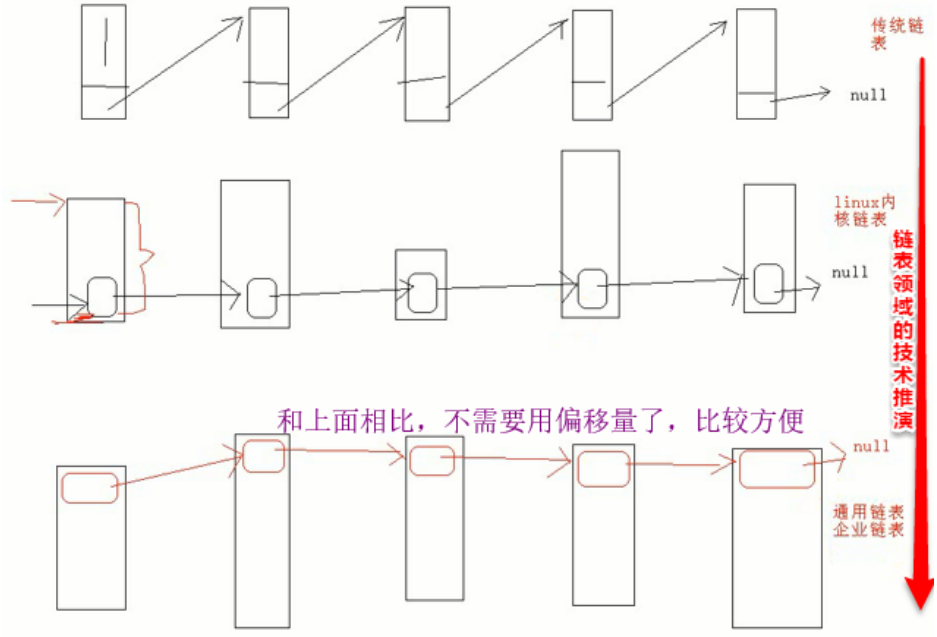
Linux内核代码博大精深，陈莉君老师曾把它形容为“覆压三百余里，隔离天日”（摘自《阿房宫赋》），可见其内容之丰富、结构之庞杂。内核里有着众多重要的数据结构，具有相关性的数据结构之间很多都是用本文介绍的链表组织在一起，看来list_head结构虽小，作用可真不小。

Linux内核是个伟大的工程，其源代码里还有很多精妙之处，值得C/C++程序员认真去阅读，即使我们不去做内核相关的工作，阅读精彩的代码对程序员自我修养的提高也是大有裨益的。

企业及财富库——链表库实例：



如果用C语言，建议使用第3种链表（通用链表，企业链表），因为第2种是linux内核链表较复杂，需要用到偏移量；两者都实现了业务数据和链表算法的分离。C++建议使用模板类实现。



代码实现（实现了业务数据和链表算法的分离）：

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "Mylinklist.h"

/*
 * Mylinklist.h 中的内容如下
 *
typedef void LinkList;

typedef struct _tag_LinkListNode LinkListNode;
struct _tag_LinkListNode
{
   LinkListNode* next;
}

//一套链表的API函数
LinkList* LinkList_Create();  //创建一个链表，返回句柄LinkList*
void LinkList_Destroy(LinkList* list);
void LinkList_Clear(LinkList* list);
int LinkList_Length(LinkList* list);
......

 */

typedef struct _Teacher
{
     LinkListNode node; //必须在业务结点的第一个域包含链表结
     char name[32];
     int age;
}Teacher;
```

```
typedef struct _Student
{
    LinkListNode node; //必须在业务结点的第一个域包含链表结
    char name[32];
}Student;

void main()
{
    //定义一个链表
    int ret = 0, len = 0, i = 0;
    LinkList *list = NULL;
    Teacher t1, t2, t3, t4, t5;
    t1.age = 31;
    t2.age = 32;
    t3.age = 33;
    t4.age = 34;
    t5.age = 35;

    list = LinkList_Create();
    if (list == NULL)
    {
        printf("func LinkList_Create() err\n");
        return;
    }

    //向链表中添加业务数据
    //业务数据和链表算法的有效分离。。。。。
    ret = LinkList_Insert(list, (LinkListNode*)&t1, LinkList_Length(list));
    ret = LinkList_Insert(list, (LinkListNode*)&t2, LinkList_Length(list));
    ret = LinkList_Insert(list, (LinkListNode*)&t3, LinkList_Length(list));
    ret = LinkList_Insert(list, (LinkListNode*)&t4, LinkList_Length(list));
    ret = LinkList_Insert(list, (LinkListNode*)&t5, LinkList_Length(list));
    len = LinkList_Length(list);

    //从链表中获取业务结点
    for (i = 0; i < LinkList_Length(list); i++)
    {
        Teacher *tmp = (Teacher *)LinkList_Get(list, i);
        if (tmp != NULL)
        {
            printf("age:%d \n", tmp->age);
        }
    }

    //删除链表结点
    while (LinkList_Length(list) > 0)
    {
        //在从链表库中删除业务结点的时候，把业务结点的指针返回给调用者，以便调用者进行额外的逻辑控制。。。。
        Teacher *tmp = (Teacher *)LinkList_Delete(list, 0);
        if (tmp != NULL)
        {
            printf("age:%d \n", tmp->age);
        }
    }

    //销毁链表
    LinkList_Destroy(list);
    system("pause");
}
```