第八章：链表、状态机与多线程

空间不够的三种扩展思路

- 拆迁→行不通成本高、外部扩展→链表
- 搬迁→可变数组：在空白内存建立一个大的数组，原来数组的元素整个复制到新数组，再释放原来数组的内存空间，用新数组替代原数组（C语言不支持需要自己封装,更高级语言支持)

单链表

- 单链表实现
  - 链表头指针：头指针不是节点，是普通指针，占4个字节，类型是struct node*，指向链表节点

链表节点结构体/模板

```
struct node
{
  int data;
  struct node *pNext;
};
//struct node只是一个结构体，本身没有变量生成，也不占用内存，相当于链表节点的模板
```

  - 链表创建过程：
    - 定义头指针 struct node *pHeader = NULL
    - 创建链表节点

未封装-创建链表节点（在堆上）

```
//链表内存申请不能用栈或者数据段（在程序加载时就已经确定）
int main(void)
{
  //1、动态申请堆内存，一个节点大小
  struct node*p=(struct node*)malloc(sizeof(struct node));

  //2、检验申请结果是否正确
  if(NULL==p)
  {
    printf("malloc error\n");
```

```
    return -1;
  }

  //3、清理申请到的堆内存（堆内存是脏的）,memset清理成0或其它数字，bzero直接清理成0
  bzero (p, sizeof(struct node));//sizeof是编译器帮忙计算，而不是运行时计算，因此每次直接写sizeof(struct node)，比 int a=sizeof(struct node);bzero (p,a);效率更高

  //4、填充节点
  p->data=1;
  p->pNext=NULL;//实际操作将下一个节点malloc返回的指针赋值给它
}
```

封装后-创建链表节点

```
struct node* create_node(int data)
{
  struct node*p=(struct node*)malloc(sizeof(struct node));

  if(NULL==P)
  {
    printf("malloc error\n");
    return NULL;
  }

  bzero (p, sizeof(struct node));

  p->data=data;
  p->pNext=NULL;

  return p;
}
```

- 关联指针和链表节点

示例

```
pHeader = create_node(1);
pHeader->pNext = create_node(2);
pHeader->pNext->pNext = create_node(3);
```

- 单链表数据访问
  - 第一个节点：pHeader→data（等同于p→data）
  - 第二个节点：pHeader→pNext→data（等同于p1→data）
  - 第三个节点：pHeader→pNext→pNext→data（等同于p2→data）

- 只能使用pHeader，现实生活中保存链表不会保存各个节点的指针

- 单链表插入新节点（尾插使用更多）

尾插

```c
void insert_tail(struct node*pHeader,struct node*new)
{
  struct node*p=pHeader;
  while(NULL!=p->pNext)
  {
    p = p->pNext;
  }
  p->pNext=new;
}

int main(void)
{
  struct node*pHeader=create_node(1);
  //若pHeader=NULL会报段错误，但程序逻辑看起来有问题，使用头结点（空的）统一操作

  insert_tail(pHeader,create_node(2));
  insert_tail(pHeader,create_node(3));

  return 0;
}
```

尾插（带头结点）

```c
//计算添加了新节点后总共多少节点，把数值写入头结点
void insert_tail(struct node*pHeader,struct node*new)
{
  int cnt=0;

  struct node*p=pHeader;
  while(NULL!=p->pNext)
  {
    p = p->pNext;
    cnt++;
  }

  p->pNext=new;
  pHeader->data=cnt+1;
}

int main(void)
{
  struct node*pHeader=create_node(0);//头结点
```

```
20
21    insert_tail(pHeader,create_node(1));
22    insert_tail(pHeader,create_node(2));
23    insert_tail(pHeader,create_node(3));
24
25    printf("header node data:%d.\n", pHeader->data);
26    printf("node1 data:%d.\n", pHeader->pNext->data);
27    printf("node2 data:%d.\n", pHeader->pNext->pNext->data);
28    printf("node3 data:%d.\n", pHeader->pNext->pNext->pNext->data);
29
30    return 0;
31 }
```

头插（带头结点）

```
1  void insert_head(struct node*pHeader,struct node*new)
2  {
3    new->pNext=pHeader->pNext;
4    pHeader->pNext=new;
5    pHeader+=1;
6  }
7  //->的实质是访问结构体内部成员，跟指针指向没有关系，跟链表的连接也无关，真正连接链表的是赋值语句（符号为=）
```

- 单链表遍历

方法1

```
1  void traverse(struct node*pHeader)
2  {
3    struct node*p=pHeader->pNext;//跨过头结点，直接走到第一个节点
4    while(NULL!=p->pNext)
5    {
6      printf("node data: %d.\n", p->data);
7      p=p->pNext;
8    }
9    printf("node data: %d.\n", p->data);
10   //可以解决问题（不加这一句会丢掉最后一个节点），但不够完美
11 }
```

方法2（推荐）

```
1  void traverse(struct node*pHeader)
2  {
3    struct node*p=pHeader;
4    while(NULL!=p->pNext)
5    {
6      p=p->pNext;
```

```
    printf("node data: %d.\n", p->data);
  }
}
```

- 单链表删除节点
  - 两种情况：删除节点是尾节点、删除节点不是尾节点

删除

```
//删除数据为data的节点
void delete_node(struct node*PH, int data)
{
  struct node*p=PH;
  struct node*pre=NULL;

  void delete(struct node*pHeader,int data)
  {
    struct node*p=pHeader;
    struct node*pre=NULL;

    while(NULL!=p->pNext)
    {
        pre=p;
        p=p->pNext;
        if(p->data==data)
        {
            pre->pNext=pre->pNext->pNext;
            free(p);
        }
    }
  }
}

```

  - 校验：创建链表→删除节点→打印链表

- 单链表逆序

一个个摘下来头插到头结点后

```
void reverse_linklist(struct node*PH)
{
  struct node*p=PH->pNext;//p指向链表第一个节点
  struct node*pBack=NULL;//保存当前节点的后一个节点

  //链表中无带数值节点 or 有一个带数值节点 = 零操作
  if((NULL==p)|(NULL==p->pNext))
    return;
```

```
9
10   while(NULL!=p->pNext)
11   {
12     pBack=p->pNext;
13     //原链表第一个节点是逆序后的尾节点，指向NULL，要进行区分
14     //节点是不是原链表第一个节点
15     if (p==PH->pNext)//原链表第一个有效节点
16     {
17       p->pNext=NULL;
18     }
19     else//非原链表第一个有效节点
20     {
21       p->pNext=PH->pNext;
22     }
23     PH->pNext=p;
24     p=pBack;
25   }
26   insert_head(PH,p);//逆序后缺失最后一个节点，需要头插
27 }
28
```

双链表

- 双链表实现

示例

```
1  struct node//节点模板
2  {
3    int data;
4    struct node*pPrev;
5    struct node*pNext;
6  };
7
8  struct node*create_node(int data)
9  {
10   struct node*p=(struct node*)malloc(sizeof(struct node));
11   if(NULL==p)
12   {
13     printf("malloc error\n");
14     return NULL;
15   }
16   p->data=data;
17   p->pPrev=NULL;
18   p->pNext=NULL;
19
20   return p;
```

```
}
```

- 双链表插入

尾插（需要遍历指针）

```
void insert_tail(struct node*PH, struct node*new)
{
  struct node*p=PH;
  while(NULL!=p->pNext)//走到尾节点
  {
    p=p->pNext;
  }
  p->pNext=new;
  new->pPrev=p;
}
```

头插

```
void insert_head(struct node*PH, struct node*new)
{
  new->pNext=PH->pNext;
  PH->pNext=new;
  if(NULL!=PH->pNext)//空链表会有问题
    new->pNext->pPrev=new;
  new->pPrev=PH;
}
```

- 双链表遍历：
  - 是对单链表有成本的扩展，链表仅需要遍历，用单链表即可

正向遍历

```
void Traverse(struct node*PH)
{
  struct node*p=PH;
  while(NULL!=p->pNext)
  {
    p=p->pNext;
    printf("data=%d.\n",p->data);
  }
}
```

反向遍历

```
void reverse_traverse(struct node*pTail)
{
```

```
  struct node*p=pTail;
  while(NULL!=p->pPrev)
  {
    printf("data=%d.\n",p->data);
    p=p->pPrev;
  }
}

```

- 双链表删除节点

删除节点

```
//删最后一个节点和中间节点有差别
int delete_node(struct node*PH, int data)
{
  struct node*p=PH;

  if(NULL==p)
  {
    return -1;
  }

  while(NULL!=p->pNext)
  {
    p=p->pNext;
    if(NULL==p->pNext)
    {
      p->pPrev->pNext=NULL;
      P->pPrev=NULL;
      free(p);
    }
    else
    {
      p->pPrev->pNext=p->pNext;
      P->pNext->pPrev=p->pPrev;
      free(p);
    }
  }
}
```

- 双链表易操作，现实生活中常使用

linux内核链表

- linux内核链表的特点

- 默认双链表且有头结点，用C语言书写，面向对象思想
- 属于纯链表，只有前后指针，没有数据区域
- 存放在include/linux/list.h，大概700行

- 内核中函数命名：
  - 前面有两个下划线，普通用户不要轻易使用，内核自己使用
  - 有一个下划线，用户一般用不着
- linux内核中对双链表的两种初始化方式

```
struct list_head{
  struct list_head *next, *prev;
}//纯链表的节点定义，只有指针，没有数据

//节点初始化1（定义一个节点并同时初始化它）
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name)\
  struct list_head name = LIST_HEAD_INIT(name)

注解：
1、使用宏进行初始化（初始化=给结构体中各成员赋值）
2、name是节点的名字，把next和prev指针初始化为&(name)，即为当前节点的首地址
3、前后指针都指向节点自己

//初始化函数2（先定义节点，再初始化）
static inline void INIT_LIST_HEAD(struct list_head *list)
{
  list->next = list;
  list->prev = list;
}
```

- 使用方式：将内核链表作为结构体的成员，使结构体有了链表的功能，需要借助container_of宏

示例

```
struct driver_info//驱动信息
{
  int data;
};

struct driver//管理内核中所有设备驱动
```

```
{
  char name[20];
  int id;//驱动的id编号
  struct driver_info info;//驱动信息

  struct list_head head;//head是个纯链表
};//前三个成员是数据区域成员
```

状态机

- 一般都是有限状态机FSM，有有限个状态（一般是一个状态变量的值），可以接收外部信号，跳转到另一个状态
- 三要素：当前状态、外部输入、下一个状态
- 两种状态机
  - Moore型状态机：输出只与当前状态有关
  - Meally型状态机：输出与当前状态和输入信号有关
- 主要用途：电路设计、FPGA程序设计（类似制造芯片，高层级但需求量少，一般都是半导体公司）、软件设计（不常用，大部分在于框架类型的设计，如操作系统GUI设计）
- 状态机实现开锁状态机

程序示例

```
/*题目：连续输入正确密码则开锁，输入错误会退回到初始状态重新计入密码*/
#include <stdio.h>

//定义状态集
typedef enum
{
  STATE1,
  STATE2,
  STATE3,
  STATE4,
  STATE5,
  STATE6
}STATE;

int main(void)
{
  int num=0;
  STATE current_state=STATE1;//初始状态为状态1，输入一个正确的密码走一步，到STATE5锁开

```

```
  //用户循环输入密码
  printf("请输入密码，密码正确开锁\n");
  while(1)
  {
    scanf("%d",&num);
    printf("num=%d.\n",num);

    switch(current_state)
    {
      case STATE1: if(num==1){current_state=STATE2;}
                   else {current_state=STATE1;}
                   break;
      case STATE2: if(num==2){current_state=STATE3;}
                   else {current_state=STATE1;}
                   break;
      case STATE3: if(num==3){current_state=STATE4;}
                   else {current_state=STATE1;}
                   break;
      case STATE4: if(num==4){current_state=STATE5;}
                   else {current_state=STATE1;}
                   break;
      case STATE5: if(num==5){current_state=STATE6;}
                   else {current_state=STATE1;}
                   break;
      default: current_state=STATE1;//用户输入一个不对的值，归零
    }
    if(current_state==STATE6)
    {
        printf("锁开了\n");
        break;//跳出while循环
    }

  }

  return 0;
}

```

多线程简介

- 多线程是为了实现操作系统下的宏观并行（分为微观串行、微观并行两种情况），单核CPU微观上一定是串行的，多核CPU中多个核心同时执行多个指令，可实现微观并行
- 进程线程目的是实现宏观并行，且实现原理不同，windows中进程线程差异大，linux中差异不大（linux中，线程是轻量级的进程）

- 多线程程序的优势：OS会优先将多个线程放在多个核心中单独运行（近年多线程开始更多使用）
- 线程同步和锁：多线程程序运行注意线程同步，通过锁机制实现

其它

- 写程序时写一点测试一点
- 链表逆序时可以不改结构，只改数据，但是实际应用数据区域很大，这样做不合理
- 只熟悉技术竞争力不如熟悉业务逻辑的，价值的体现=把技术真正用在产品研发中，没办法通过模拟项目实现，必须去公司做项目
- 链表是为了存储数据的，看别人的链表代码一定要注意有没有头结点，算法实现完全不同
- 函数封装的关键点是函数的接口设计（参数和返回值）
- 算法实现分两步：先理清楚程序的步骤（逻辑能力），再书写程序（编程功底）
- 架构轮不到你做，重要的是先把细节做好
- 学习Linux内核才能真正学好C语言
- man 2-系统调用；man 3-库函数