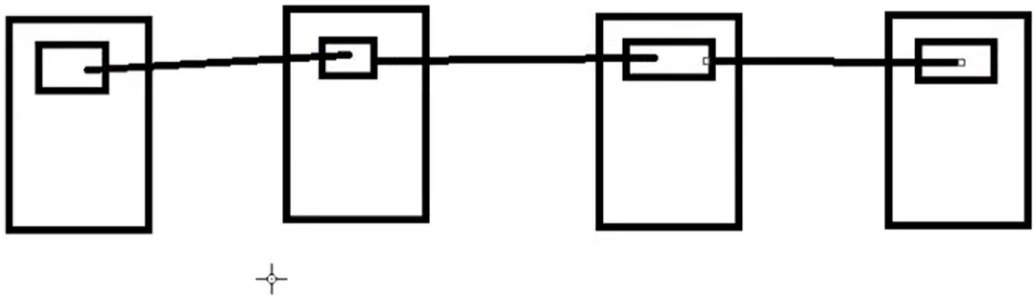
企业链表的思路



像是用一个挂钩把几件衣服给串起来了。

进一步理解

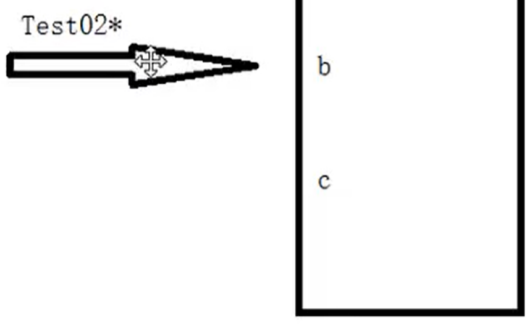
假设我们现在有两个结构体。内部结构如下：

```
struct test01{
    int a;
    int b;
}
```

```
struct test01{
    int a;
    int b;
    int c;
}
```

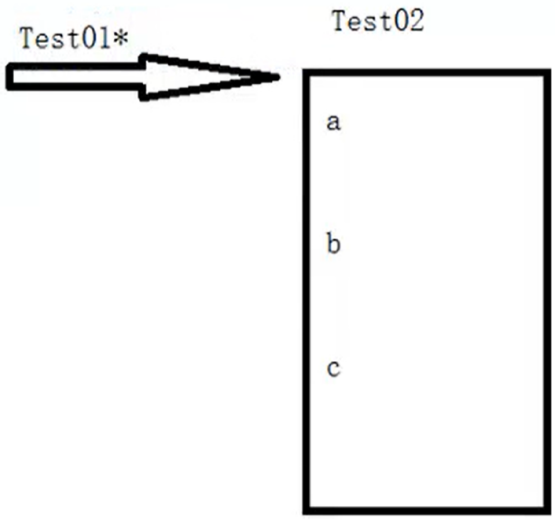
然后呢，对于，test02这个数据，我们可以使用 test02* 类型的指针指向它，从而访问它内部的数据 a , b , c 。

```
struct Test01{
    int a;
    int b;
};

struct Test02{
    int a;
    int b;
    int c;
};
```

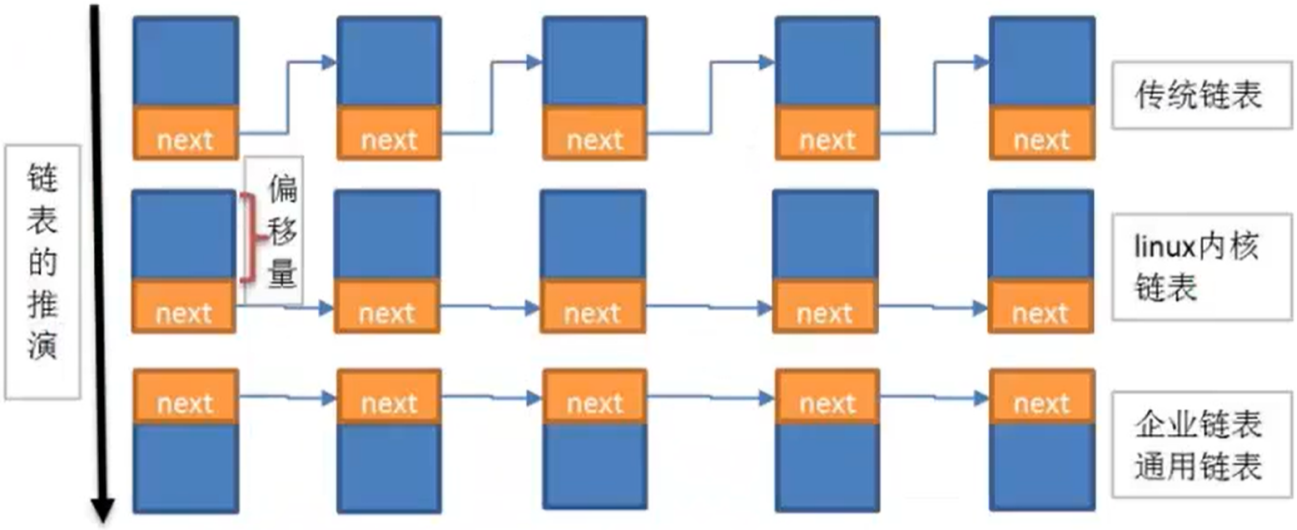


但是，如果指针类型变成 test01* 类型，那么就只能访问到 a , b ,无法访问 c 。

```
struct Test01{
    int a;
    int b;
};

struct Test02{
    int a;
    int b;
    int c;
};
```



好的，明白这个原理之后，我们来看一看 熟知的三种链表：



> 传统链表包含数据域和指针域，就是我们从课本上学到的那一种。
>
> 而企业链表把 “挂钩” 放在了结构体的首地址，挂钩指向挂钩，从而把表了连接起来。
>
> linux呢，则是把 “挂钩” 放在了底部，那么串的时候就需要自己计算偏移量。

所以呢，企业链表其实就是把linux内核的链表进行了优化 ， 使其更容易理解和操作。

那么企业链表如何用代码实现呢?

首先 ， 按照传统的方式去写节点的结构体是这样子的，包含数据域和指针域。

```
typedef struct LINKNODE{
    void* data;
    struct LINKNODE* next;
}LinkNode;
```

但是呢，在企业链表中，我们定义的时候，仅仅保留指针域，相当于这只是 一个 "挂钩"

如图:

```
typedef struct LINKNODE{
    struct LINKNODE* next;
}LinkNode;
```

那么，问题来了，怎么样去使用它呢?

我们先定义一个自己平时使用的数据类型

```
typedef struct MyData{

    char name[64];
    int age;
}Mydata;
```

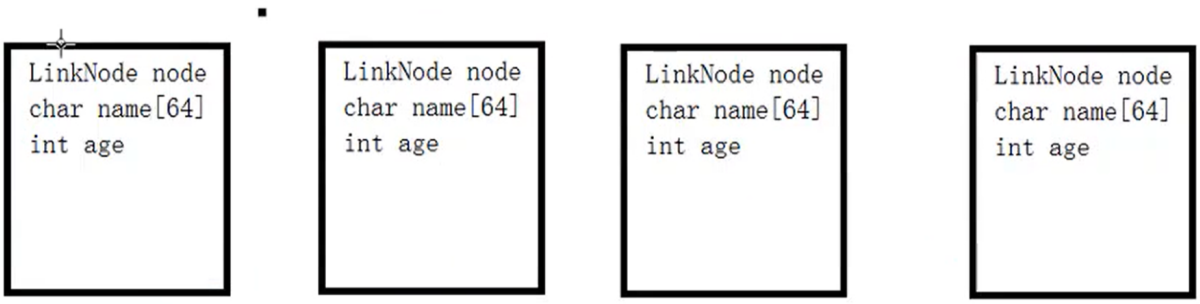
普普通通，一眼就看得懂。

然后呢，我们给它加上刚刚创建的 "挂钩 "

```
typedef struct MyData{
    LinkNode node;
    char name[64];
    int age;
}Mydata;
```

如此一来，我们的每个结构体都包含这个 ”挂钩“ ，而这个 ”挂钩“ 呢，可以指向 ”挂钩“

类型的数据。



然后呢，我们的 ”挂钩“ 在我们自定义 结构体的首地址。

因此，我们只需要把我们的自定义 结构体Mydata*类型的数据转换成

LinKNode*的数据类型，就可以将各个节点串起来了，具体操作如下：

```
Mydata data1
LinKNode* node = (LinkNode*)&(data1)
```

这样的类型转换数据并不会丢失，仅仅只是LinKNode类型的指针访问不到我们的Mydata结构体中的其它数据而已，而当我们需要对Mydata中的数据进行操作时，很简单，把类型转换成Mydata指针类型即可。

具体代码实现

LinkList.h

```
#pragma once
#ifndef LINKLIST_H
#define LINKLIST_H
```

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

//链表小节点
typedef struct LINKNODE {
    struct LINKNODE* next;
}LinkNode;

//链表节点
typedef struct LINKLIST {
    LinkNode head;
    int size;
}LinkList;

//遍历函数指针,自定义类型写法
typedef void(*PRINTNODE)(LinkNode*);
//比较函数指针
typedef int(*COMPARENODE)(LinkNode*, LinkNode*);

//初始化链表
LinkList* Init_LinkList();
//指定位置插入
void Insert_LinkList(LinkList* list, int pos, LinkNode* data);
//删除指定位置的值
void Remove_LinkList(LinkList* list, int pos);
//查找
int Find_LinkList(LinkList* list, LinkNode* data, COMPARENODE compare);
//返回链表的大小
int Size_LinkList(LinkList* list);
//打印链表结点
void Print_LinkList(LinkList* list, PRINTNODE print);
//释放链表内存
void FreeSpace_LinkList(LinkList* list);

#endif    //!LINKLIST_H
```

LinkList.c

```c
//#include"LinkList.h"
#include"../03 企业链表/LinkList.h"



//初始化链表
LinkList* Init_LinkList() {
    LinkList* list = (LinkList*)malloc(sizeof(LinkList));
    list->head.next = NULL;
    list->size = 0;
    return list;
}
//指定位置插入
void Insert_LinkList(LinkList* list, int pos, LinkNode* data) {
    if (list == NULL) {
        return;
    }
    if (data == NULL) {
        return;
```

```
    }
    if (pos<0 || pos>list->size) {
        pos = list->size;
    }

    //查找插入位置
    LinkNode* pCurrent = &(list->head);
    for (int i = 0; i < pos; i++) {
        pCurrent = pCurrent->next;
    }

    //插入新节点
    data->next = pCurrent->next;
    pCurrent->next = data;

    list->size++;
}

//删除指定位置的值
void Remove_LinkList(LinkList* list, int pos) {
    if (list == NULL) {
        return;
    }
    if (pos<0 || pos>list->size) {
        pos = list->size;
    }
    //辅助指针变量
    LinkNode* pCurrent = &(list->head);
    for (int i = 0; i < pos; i++) {
        pCurrent = pCurrent->next;
    }

    //删除节点
    pCurrent->next = pCurrent->next->next;
    list--;

}

//查找
int Find_LinkList(LinkList* list, LinkNode* data, COMPARENODE compare) {
    if (list == NULL) {
        return -1;
    }
    if (data == NULL) {
        return -1;
    }
    //辅助指针变量
    LinkNode* pCurrent = list->head.next;
    int index = 0;
    int flag = -1;
    while (pCurrent != NULL) {
        if (compare(pCurrent, data) == 0) {
            flag = index;
            break;
        }
        pCurrent = pCurrent->next;
        index++;
    }
    return flag;
}
```

```
//返回链表的大小
int Size_LinkList(LinkList* list) {
    return 0;
}

//打印链表结点
void Print_LinkList(LinkList* list, PRINTNODE print) {
    if (list == NULL) {
        return;
    }
    //辅助指针变量
    LinkNode* pCurrent = list->head.next;
    while (pCurrent != NULL) {
        print(pCurrent);
        pCurrent = pCurrent->next;
    }
}

//释放链表内存
void FreeSpace_LinkList(LinkList* list) {

    if (list == NULL) {
        return;
    }

    free(list);

}
```

企业链表.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include"../03 企业链表/LinkList.h"

//链表节点
typedef struct PERSON {
    LinkNode node;
    char name[64];
    int age;
}Person;

void MyPrint(LinkNode* data) {
    Person* p = (Person*)data;
    printf("Name:%s, Age:%d\n", p->name, p->age);
}

int MyCompare(LinkNode* node1, LinkNode* node2) {
    Person* p1 = (Person*)node1;
    Person* p2 = (Person*)node2;

    if (strcmp(p1->name, p2->name) == 0 && p1->age == p2->age) {
        return 0;
    }
    return -1;
```

```
}


int main(void)
{
    //创建链表
    LinkList* list = Init_LinkList();

    //创建数据
    Person p1, p2, p3, p4, p5;
    strcpy(p1.name, "唐僧");
    strcpy(p2.name, "悟空");
    strcpy(p3.name, "八戒");
    strcpy(p4.name, "沙僧");
    strcpy(p5.name, "白龙马");

    p1.age = 100;
    p2.age = 200;
    p3.age = 300;
    p4.age = 400;
    p5.age = 500;

    //将结点插入链表
    Insert_LinkList(list, 0, (LinkNode*) &p1);
    Insert_LinkList(list, 0, (LinkNode*) &p2);
    Insert_LinkList(list, 0, (LinkNode*) &p3);
    Insert_LinkList(list, 0, (LinkNode*) &p4);
    Insert_LinkList(list, 0, (LinkNode*) &p5);

    //打印
    Print_LinkList(list, MyPrint);



    //删除节点
    Remove_LinkList(list, 2);

    //打印
    printf("--------------------\n");
    Print_LinkList(list, MyPrint);

    //查找
    Person findP;
    strcpy(findP.name, "唐僧");
    findP.age = 100;
    int pos = Find_LinkList(list, (LinkNode*) &findP, MyCompare);
    printf("位置: %d\n", pos);

    //释放链表内存
    FreeSpace_LinkList(list);

    printf("\n");
    system("pause");
    return 0;
}
```