嵌入式C语言之-

通用链表工作原理

讲师：叶大鹏

助力你成为优秀的电子工程师！

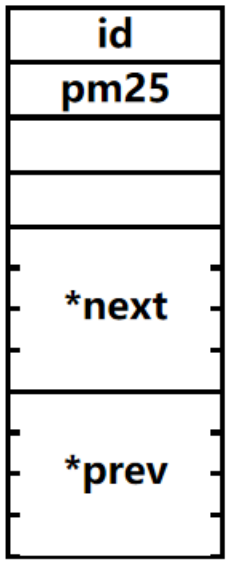
为什么要使用通用链表

id
humi
temp
*next
*prev

```
typedef struct TempHumiListNode
{
  uint8_t id;
  uint8_t humi;
  float temp;
  struct TempHumiListNode *next;
  struct TempHumiListNode *prev;
} TempHumiListNode;

void AddNode(TempHumiListNode *oldNode, TempHumiListNode *newNode)
{
        newNode->next = oldNode->next;
        newNode->prev = oldNode;
        oldNode->next->prev = newNode;
        oldNode->next = newNode;
}
```

链表的指针类型是具体的业务数据类型，不具有通用性

为什么要使用通用链表

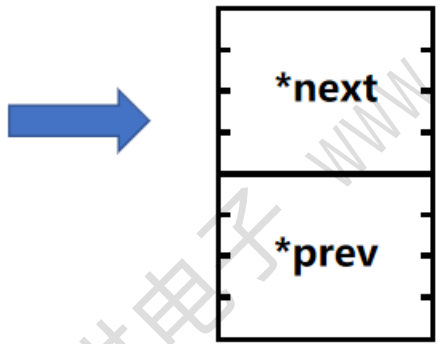



```
typedef struct Pm25ListNode
{
  uint8_t id;
  uint8_t pm25;
  struct Pm25ListNode *next;
  struct Pm25ListNode *prev;
} Pm25ListNode;

void AddNode(Pm25ListNode *oldNode, Pm5ListNode *newNode)
{
        newNode->next = oldNode->next;
        newNode->prev = oldNode;
        oldNode->next->prev = newNode;
        oldNode->next = newNode;
}
```

新的业务数据，同样需要提供链表指针的指向接口函数，实现指针的指向功能，代码冗余且易出错

为什么要使用通用链表

```
typedef struct TempHumiListNode
{
 uint8_t id;
 uint8_t humi;
 float temp;
  struct TempHumiListNode *next;
  struct TempHumiListNode *prev;
} TempHumiListNode;
```



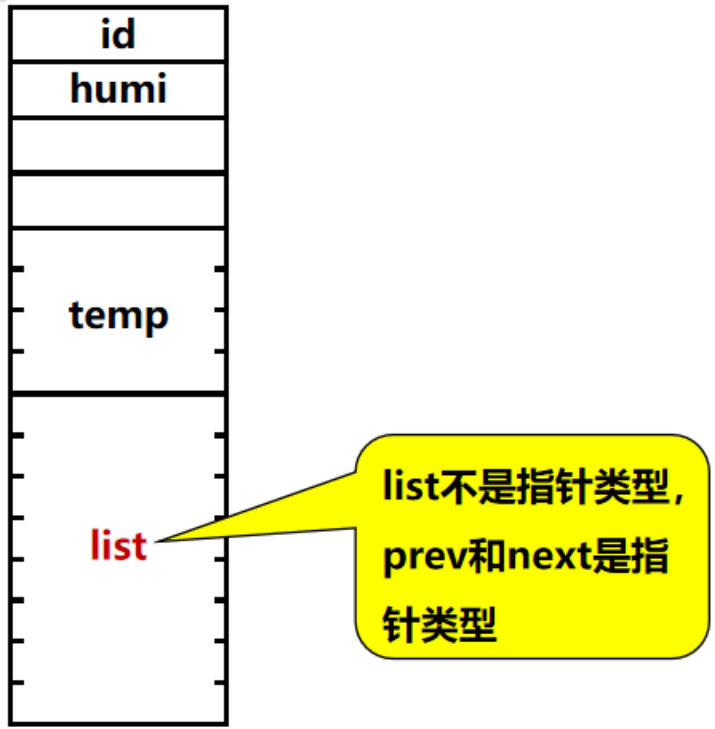

```
typedef struct List
{
   struct List *prev;
   struct List *next;
} List;

void AddNode(List *oldNode, List *newNode)
{
   newNode->next = oldNode->next;
   newNode->prev = oldNode;
   oldNode->next->prev = newNode;
   oldNode->next = newNode;
}
```

如何使用通用链表

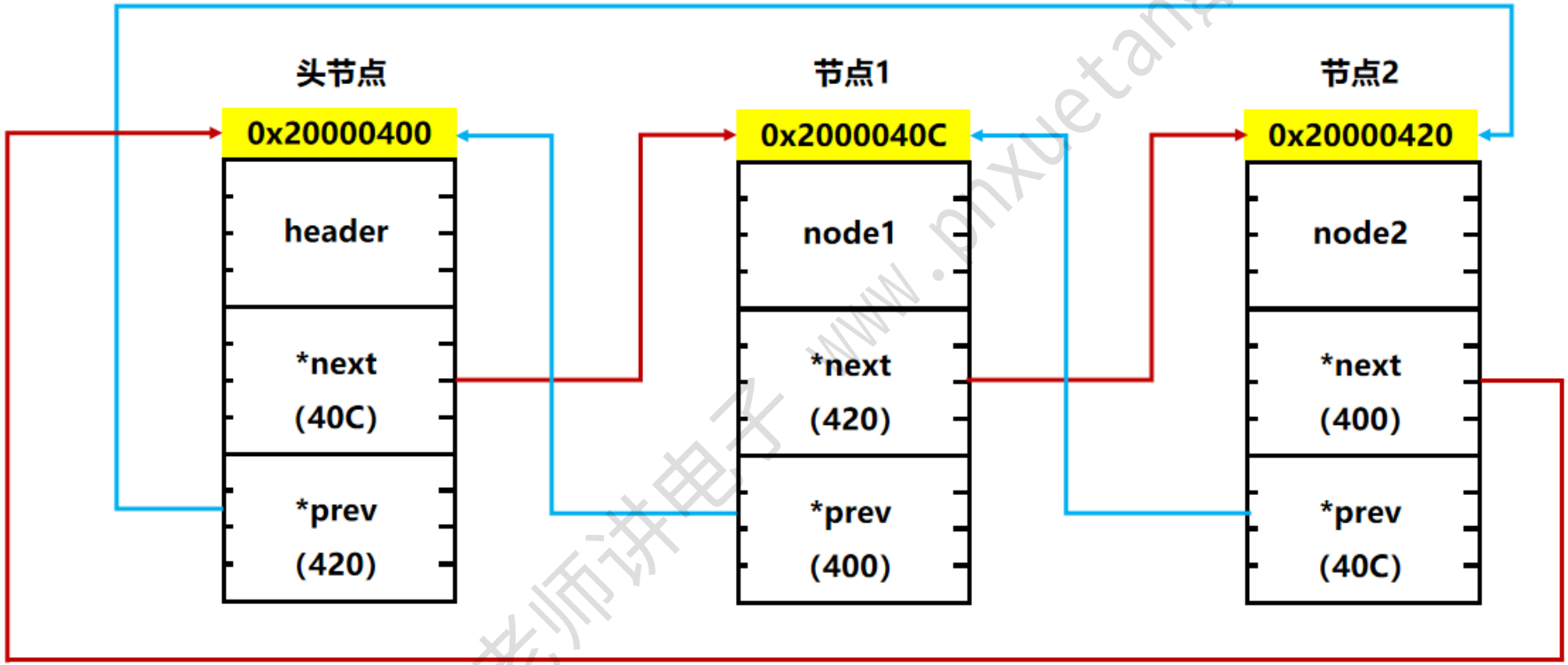
- 将通用链表嵌入业务数据结构体中，作为一个成员使用：

```
typedef struct List
{
    struct List *prev;
    struct List *next;
} List;
```

```
typedef struct TempHumiSensor
{
    uint32_t id;
    uint8_t humi;
    float temp;
    List list;
} TempHumiSensor;
```

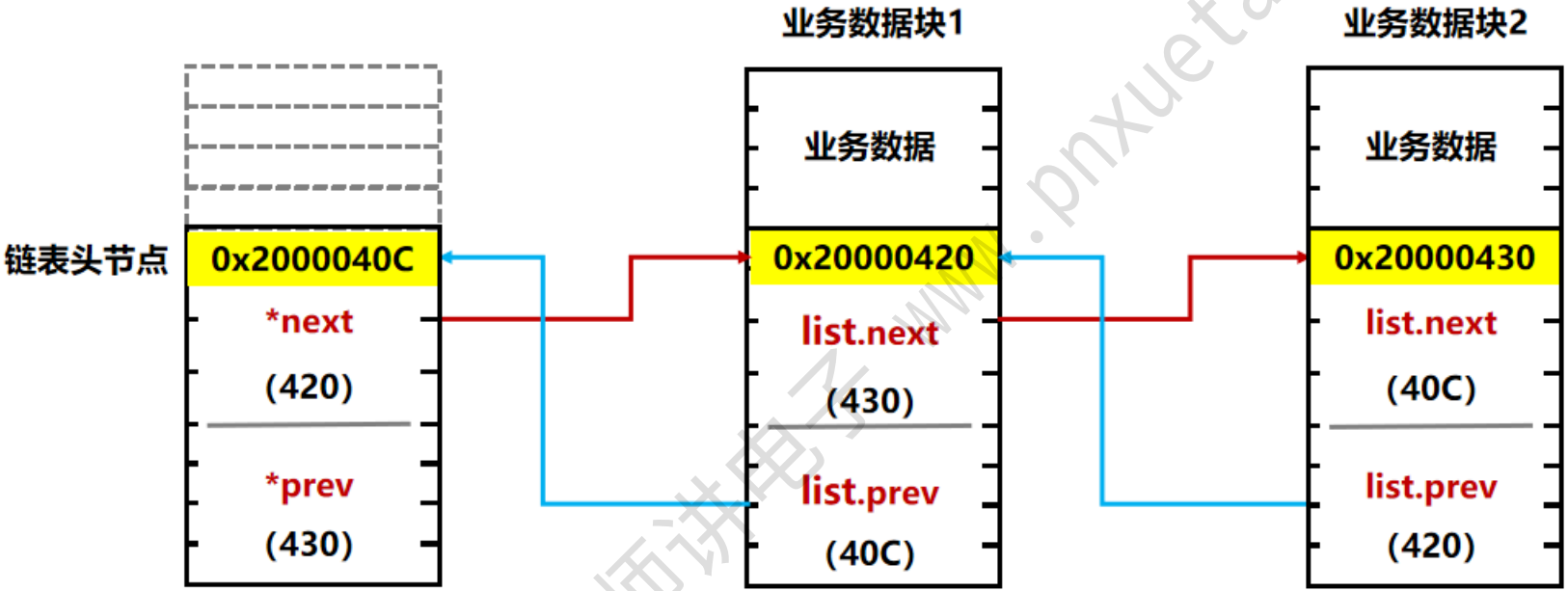


常规双向循环链表的指向



- 常规双向循环链表，next和prev指向的都是业务数据结构体的首地址。

通用链表的指向



- 通用链表，也是双向循环链表，next和prev指向的是业务数据结构体中list成员的首地址。

通用链表的初始化

```
static List *g_tempHumiHeader;
bool InitTempHumiSensor(void)
{
    g_tempHumiHeader = (List *)malloc(sizeof(struct List));
    if (g_tempHumiHeader == NULL)
    {
        return false;
    }
    InitList(g_tempHumiHeader);
    return true;
}
int main(void)
{
    InitTempHumiSensor())
```

```
void InitList(List *header)
{
    header->next = header;
    header->prev = header;
}
```

通用链表的添加

```
void AddTempHumiSensor(TempHumiSensor *sensor)
{
    AddNodeToTail(g_tempHumiHeader, &sensor->list);
}

int main(void)
{
    InitTempHumiSensor();
    TempHumiSensor *sensor;
    sensor = FindTempHumiSensor();
    AddTempHumiSensor(sensor);
```

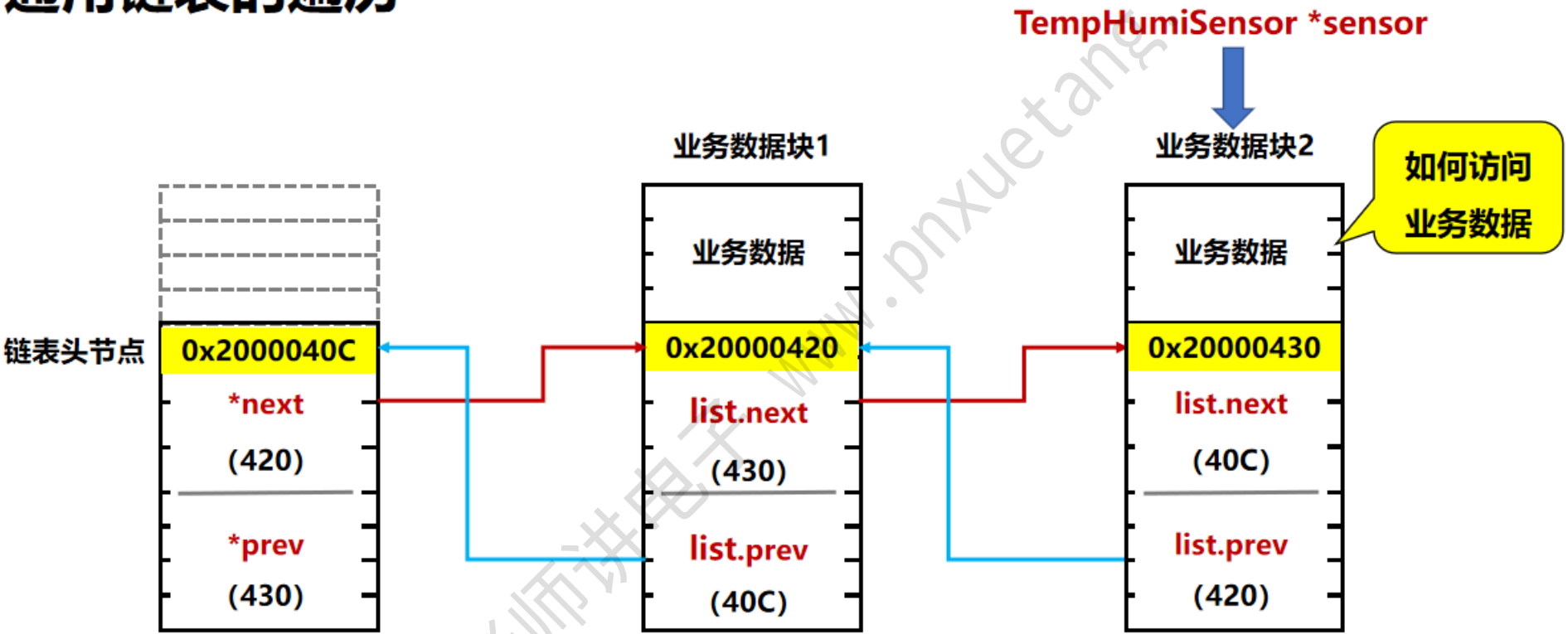
```
void AddNode(List *oldNode, List *newNode)
{
    newNode->next = oldNode->next;
    newNode->prev = oldNode;
    oldNode->next->prev = newNode;
    oldNode->next = newNode;
}

void AddNodeToTail(List *header, List *node)
{
    AddNode(header->prev, node);
}
```

常规双向循环链表的遍历

```
void PrintSensorData(TempHumiListNode *header)
{
    TempHumiListNode *current;
    current = header->next; //从头节点下一个节点开始遍历
    if (current == header) //判断是否只有一个头节点
    {
        printf("List has no node!\n");
        return;
    }
    while (current != header) //遍历到头节点，不再执行循环语句，退出循环
    {
        printf("\nSensor id:%d,temp = %.1f,humi = %d.\n", current->id, current->temp, current->humi);
        current = current->next;
    }
}
```

链表的指针类型是具体的业务数据类型，可以直接访问成员

通用链表的遍历



- 通用链表，遍历关键点：

数据块是由list成员串联起来的，它的next和prev指向的是业务数据结构体中list成员的首地址，所以需要根据list成员的首地址得到业务数据结构体的首地址，才能进一步访问其它成员打印数据。

通用链表的遍历

- 参照Linux和鸿蒙系统的container_of用法，设计宏：

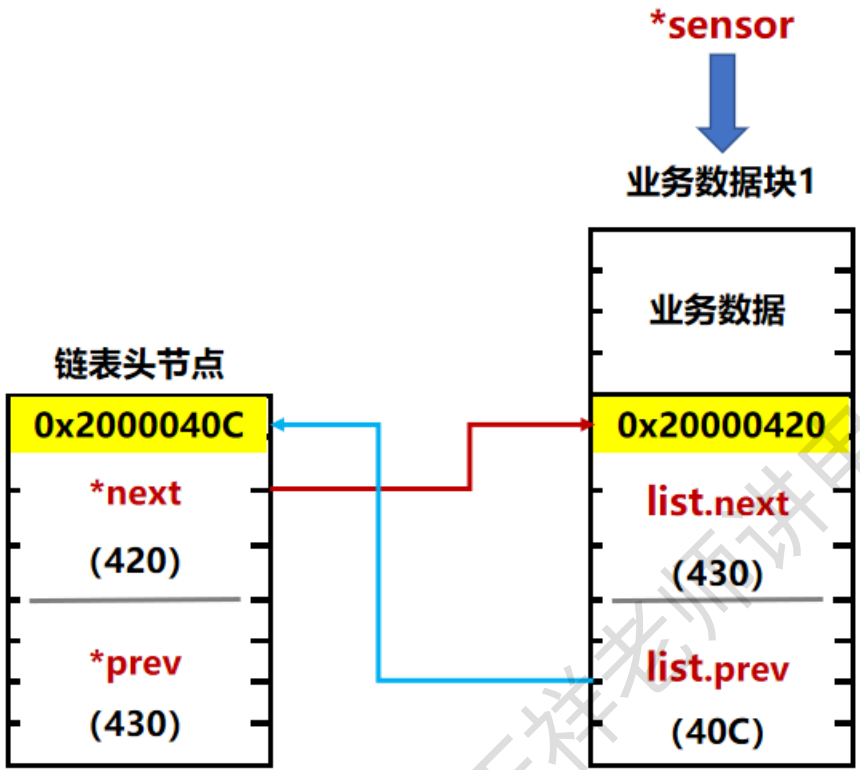
```
/**
 ********************************************************************************
 * @brief 获取结构体中链表的偏移地址
 * @param typeName: 结构体类型名字
 * @param memberName: 链表在结构体的名字
 * @return 偏移
 ********************************************************************************
 */
#define OFFSET_OF(typeName, memberName)  ((long)&((typeName *)0)->memberName)

/**
 ********************************************************************************
 * @brief 获取指向包含双链表的结构体的指针
 * @param pList: 结构体中链表的地址
 * @param typeName: 结构体类型名字
 * @param memberName: 链表在结构体中的名字
 * @return 包含双链表的结构体的指针
 ********************************************************************************
 */
#define CONTAINER_OF(pList, typeName, memberName) \
    ((typeName*)((char *)pList - OFFSET_OF(typeName, memberName)))
```

通用链表的遍历

```
void PrintTempHumiSensor()
{
    TempHumiSensor *sensor = NULL; //定义业务数据的局部变量，用于遍历数据块，打印成员数据

    for (sensor = 初始值; 条件表达式; sensor = 每轮动作)
    {
        printf("sensor %d, temp = %.1f, humi = %d.\n", sensor->id, sensor->temp, sensor->humi);
    }
}
```

通用链表的遍历

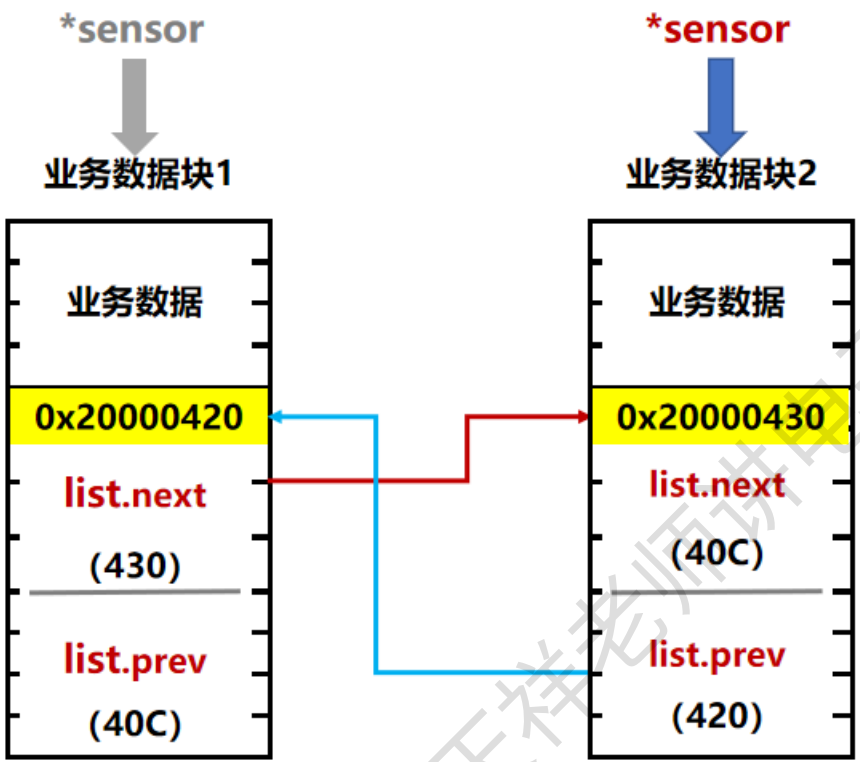
- sensor = *初始值*;



1. 已知头节点：
   g_tempHumiHeader
2. 获得数据块1的list成员首地址：
   g_tempHumiHeader->next
3. 有了成员list地址420，使用CONTAINER_OF得到数据块1的首地址(420 - 偏移量)：
   CONTAINER_OF(
   g_tempHumiHeader->next, TempHumiSensor, list)
4. 赋值给sensor：
   sensor = CONTAINER_OF(g_tempHumiHeader
   ->next, TempHumiSensor, list);

通用链表的遍历

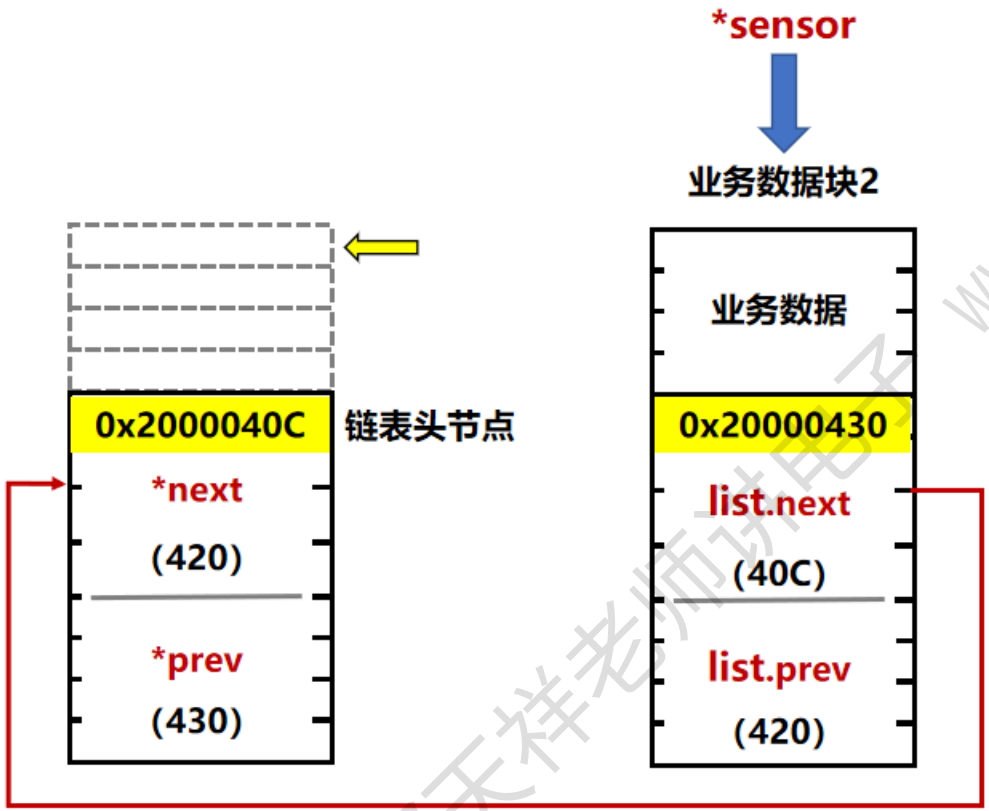
- sensor = *每轮动作*;



1. 假如sensor指向数据块1：
2. 获得数据块2的list成员首地址：
   sensor->list.next
3. 有了成员list地址430，使用CONTAINER_OF得到数据块2的首地址(430 - 偏移量)：
   CONTAINER_OF(
   sensor->list.next, TempHumiSensor, list)
4. 赋值给sensor，指向数据块2：
   sensor = CONTAINER_OF(sensor->list.next,
   TempHumiSensor, list);

通用链表的遍历

- *条件表达式*；



1. 当sensor指向数据块2，完成了打印数据，再执行每轮动作：
   sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list);
   此时，sensor保存的地址值是头节点首地址40C – 偏移量；
2. 再使用&sensor->list，就可以获得头节点首地址40C，循环边界为，获得的地址和头节点地址相同的退出循环：
   &sensor->list != g_tempHumiHeader;

通用链表的遍历

```
void PrintTempHumiSensor(void)
{
    TempHumiSensor *sensor = NULL; //定义业务数据的局部变量，用于遍历数据块，打印成员数据

    for (sensor = CONTAINER_OF(g_tempHumiHeader->next, TempHumiSensor, list);
         &sensor->list != g_tempHumiHeader;
         sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list))
    {
        printf("sensor %d, temp = %.1f, humi = %d.\n", sensor->id, sensor->temp, sensor->humi);
    }
}
```

通用链表的删除

```
void DelTempHumiSensor(uint32_t id)
{
    TempHumiSensor *sensor = NULL;

    for (sensor = CONTAINER_OF(g_tempHumiHeader->next, TempHumiSensor, list);
        &sensor->list != g_tempHumiHeader;
        sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list))
    {
        if (sensor->id == id)
        {
            DelNode(&sensor->list);
            free(sensor);
            sensor = NULL;
            return;
        }
    }
    printf("Can not find sensor %d.\n", id);
}
```

通用链表的删除

```
void DelNode(List *node)
{
    node->next->prev = node->prev;
    node->prev->next = node->next;
}
```

循环遍历设计为宏定义

- for (sensor = CONTAINER_OF(g_tempHumiHeader->next, TempHumiSensor, list);
  &sensor->list != g_tempHumiHeader;
  sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list))

1）sensor设计为对应宏参数pBusi，表示业务结构体指针变量，通过它在循环语句中打印业务数据；

2）g_tempHumiHeader->next，g_tempHumiHeader设计为对应宏参数header，表示链表的头节点首地址；

3）TempHumiSensor设计为对应宏参数typeName，表示结构体类型名字；

4）list设计为对应宏参数memberName，表示链表在结构体中的名字。

循环遍历设计为宏定义

- for (sensor = CONTAINER_OF(g_tempHumiHeader->next, TempHumiSensor, list);
  &sensor->list != g_tempHumiHeader;
  sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list))

➢ #define LIST_FOR_EACH_ENTRY(pBusi, header, typeName, memberName) \
  for (pBusi = CONTAINER_OF((header)->next, typeName, memberName); \
  &pBusi->memberName != (header); \
  pBusi = CONTAINER_OF(pBusi->memberName.next, typeName, memberName))

通用链表的遍历

```
void PrintTempHumiSensor(void)
{
    TempHumiSensor *sensor = NULL; //定义业务数据的局部变量，用于遍历数据块，打印成员数据

    for (sensor = CONTAINER_OF(g_tempHumiHeader->next, TempHumiSensor, list);
        &sensor->list != g_tempHumiHeader;
        sensor = CONTAINER_OF(sensor->list.next, TempHumiSensor, list))
    LIST_FOR_EACH_ENTRY(sensor, g_tempHumiHeader, TempHumiSensor, list)
    {
        printf("sensor %d, temp = %.1f, humi = %d.\n", sensor->id, sensor->temp, sensor->humi);
    }
}
```

THANK YOU!