第 5 章

其 他 组 件

5.1 时间模块

时间模块是 MCU 的基础，也是操作系统正常运转的必备条件。通常情况下，操作系统以 Tick 为基本时间单位。开发者可以通过 menuconfig 进入配置菜单，选择 Kernel→Base Config→Task→(1000)Tick Value Per Second 设置每秒的 Tick 数。

9min

5.1.1 时间转换

系统以 MCU 的工作时钟为基础，提供一套时钟管理单元。操作系统以 Tick 为基本时间单位，也称“时钟滴答”，而用户以秒、毫秒为时间单位，用户需要了解 Tick 与秒、毫秒之间的转换关系。

系统的最小时间单位是 Cycle，主时钟频率就是每秒的 Cycle 数。在 targets/目标板/Src/system_stm32l4xx.c 文件中定义了全局变量 SystemCoreClock，此变量代表系统时钟。

1. 时间转换 API

LiteOS 时间管理单元提供了若干转换函数，可将 Tick 和普通时间相互转换，见表 5-1。

表 5-1 时间管理 API

函　　数	说明(头文件 kernel/include/los_tick.h，错误码也参考此文件)
LOS_Tick2MS	将 Tick 转换为毫秒 返回值：UINT32，转换结果 参数：[IN] UINT32，待转换的 Tick
LOS_MS2Tick	延时微秒，可被高优先级任务打断 返回值：无 参数：[IN] UINT32，延时的微秒数
LOS_Udelay	延时毫秒，可被高优先级任务打断 返回值：无 参数：[IN] UINT32 usecs，延时的毫秒数

续表

函　　数	说明(头文件 kernel/include/los_tick.h,错误码也参考此文件)
LOS_Mdelay	将毫秒转换为 Tick 返回值：UINT32,转换结果 参数：[IN] UINT32 msecs,待转换的毫秒数
LOS_TickCountGet	获取从系统启动开始到目前的 Tick 数 返回值：UINT64,返回的 Tick 数 参数：无
LOS_GetCpuCycle	获取从系统启动开始到目前的 Cycle 数 返回值：无 参数 1：[OUT] UINT32 * puwCntHi,Cycle 的高 32 位 参数 2：[OUT] UINT32 * puwCntLo,Cycle 的低 32 位
LOS_CurrNanosec	获取从系统启动开始到目前的纳秒数 返回值：UINT64,64 位结果 参数 1：无

开发者在使用时间管理函数之前务必配置每秒 Tick 数,若不配置,则默认每秒1000Tick。

注意：针对 STM32 系列的芯片,开发者需使用介于 2019.10～2021.10 版本的交叉编译器,否则时间单元会出错。

2. 实战案例

1）案例测试

配置好系统时钟 OS_SYS_CLOCK 和每秒的 Tick 数,对毫秒和 Tick 进行相互转换,并获取系统时钟。在 mydemos 文件夹下创建源文件 time/time.c、头文件 time/time.h,代码如下：

```
//第 5 章/mydemos/time/time.c
UINT32 demo_time(VOID)
{
    //获取系统时钟
    printf("bus clk is %d\n", get_bus_clk() );
    //获取每秒的 Tick
    printf("1 second have %d tick\n", LOSCFG_BASE_CORE_TICK_PER_SECOND);
    //毫秒转 Tick
    printf("1000 ms is %d tick\n", LOS_MS2Tick(1000));
    //Tick 转毫秒
    printf("1000 tick is ms %d\n", LOS_Tick2MS(1000));
    //获取每个 Tick 的 Cycle
    printf("1 tick have %d cycle", LOS_CyclePerTickGet());

    return 0;
```

```
}


//第 5 章/mydemos/time/time.h
#ifndef __TIME_H
#define __TIME_H

#include "sys_init.h"
#include "los_tick.h"
#include "hisoc/clock.h"
#include "menuconfig.h"

UINT32 demo_time(VOID);

#endif
```

```
bus clk is 80000000
1 second have 1000 tick
1000 ms is 1000 tick
1000 tick is ms 1000
1 tick have 80000 cycle
```

图 5-1　运行结果

参考 3.2.4 节修改 Makefile，将源文件路径添加到 LOCAL_SRCS，将头文件路径添加到 LOCAL_INCLUDE。在源文件 user_task.c 中调用 demo_time()函数，运行结果如图 5-1 所示。

2）结果分析

这里使用默认配置每秒 1000Tick，因此得到的转换结果为 1000ms==1000Tick。系统时钟为 80MHz，因此 1Tick==1ms==80000Cycle。

17min

5.1.2　软件定时器

在开发过程中通常会使用定时器执行周期性任务，而 MCU 的定时器数量有限，因此 LiteOS 提供了软件定时器功能。

软件定时器是基于 Tick 中断并由软件模拟的定时功能，当经过指定的 Tick 之后，会触发用户设置的回调函数。

1. 运行机制

系统使用队列维护软件定时器，在定时器队列中，时间短的定时器总是排在时间长的定时器的前面，这样可使时间短的定时器优先触发。在头文件 Kernel/base/include/los_swtmr_pri.h 中定义了软件定时器的核心结构体 LosSwtmrCB，代码如下：

```
//第 5 章/kernel/base/include/los_swtmr_pri.h
typedef struct {
    SortLinkList sortList;
    UINT8 state;                    /* 定时器状态 */
    UINT8 mode;                     /* 定时模式，单次、周期性 */
    UINT8 overrun;                  /* 周期性定时器的执行次数 */
    UINT16 timerId;                 /* 定时器 ID */
    UINT32 interval;                /* 周期性定时器的周期 (unit: tick) */
    UINT32 expiry;                  /* 单次定时器的触发时间 (unit: tick) */
```

```
#ifdef LOSCFG_KERNEL_SMP
    UINT32 cpuid;                    /* 在多核模式下,与定时器相关的 CPU */
#endif
    UINTPTR arg;                     /* 定时器回调函数的参数 */
    SWTMR_PROC_FUNC handler;         /* 定时器回调函数 */
} LosSwtmrCB;
```

软件定时器以 Tick 为基本单位,系统需要一个队列和一个任务资源来维护软件定时器。定时器在创建时被加入一个计时的全局链表,当 Tick 中断发生时,扫描该链表中是否有超时的定时器,若有定时器超时,则执行该定时器对应的回调函数。

LiteOS 定时器有以下 3 种模式:

(1) 单次触发模式 1,只触发一次,之后便自动删除。

(2) 单次触发模式 2,只触发一次,但是不会自动删除。

(3) 周期触发模式,周期性触发,直到用户手动关闭为止。

注意:软件定时器的任务优先级为 0,并且不可以修改。由于软件定时器使用了队列和任务等资源,因此不使用的定时器要及时删除。

2. 软件定时 API

LiteOS 软件定时器模块为用户提供了定时器的创建、删除、启动、停止等功能,见表 5-2。

表 5-2　软件定时器 API

函　　数	说明(头文件 kernel/include/los_swtmr.h,错误码也参考此文件)
LOS_SwtmrCreate	创建一个软件定时器 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回错误码 参数 1:[IN] UINT32 interval,定时器超时时间 参数 2:[IN] UINT8 mode,定时器模式 参数 3:[IN] SWTMR_PROC_FUNC handler,定时器超时回调函数 参数 4:[OUT] UINT16 * swtmrId,定时器 ID 参数 5:[IN] UINTPTR arg,超时回调函数的参数
LOS_SwtmrDelete	删除一个软件定时器 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回错误码 参数:[IN] UINT16 swtmrId,指定要删除的定时器 ID
LOS_SwtmrStart	启动定时器 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回错误码 参数:[IN] UINT16 swtmrId,指定要启动的定时器 ID
LOS_SwtmrStop	停止定时器 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回错误码 参数:[IN] UINT16 swtmrId,指定要停止的定时器 ID

续表

函　　数	说明（头文件 kernel/include/los_swtmr.h，错误码也参考此文件）
LOS_SwtmrTimeGet	获取定时器剩余要超时的 Tick 数 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回错误码 参数 1：[IN] UINT16 swtmrId，指定要启动的定时器 ID 参数 2：[OUT] UINT32 * tick，剩余的 Tick 返回这里

定时器超时回调函数只能传递一个整型参数，并且没有返回值。为了避免定时器响应不及时，不建议在回调函数中执行复杂操作，或者可能导致任务阻塞的操作。

3. 实战案例

1）案例测试

创建两个软件定时器，定时器 1 为单次执行，定时器 2 为周期性执行。在 mydemos 文件夹下创造源文件 swt/swt.c、头文件 swt/swt.h，代码如下：

```
//第 5 章/mydemos/swt/swt.c
#include "swt.h"

UINT16 swt1_id;
UINT16 swt2_id;
UINT16 swt2_cnt = 0;

//定时器 1 回调函数
void timer1_callback(UINT32 arg){
    UINT32 tk;

    printf("timer1 timeout\n");
    //获取 timer2 的剩余超时
    LOS_SwtmrTimeGet(swt2_id, &tk);
    printf("timer2 still need %d tick overflow\n", tk);
}

//定时器 2 回调函数
void timer2_callback(UINT32 arg){
    swt2_cnt++;
    printf("timer2 timeout\n");
    //获取自系统启动经过的 tick，注意这里是 64 位整数，需要使用 %lld
    printf("system has pass %lld tick\n", LOS_TickCountGet());
    if(swt2_cnt == arg){
        printf("timer2 run %d times, stop timer2\n", swt2_cnt);
        LOS_SwtmrStop(swt2_id);
    }
}

UINT32 demo_swt(VOID){
    //单次定时，超时之后会自动删除
    LOS_SwtmrCreate(1000, LOS_SWTMR_MODE_ONCE, \
```

```
                    (SWTMR_PROC_FUNC)timer1_callback, &swt1_id, 1);
    //周期定时,除非用户手动停止并删除,否则持续执行
    LOS_SwtmrCreate(3000, LOS_SWTMR_MODE_PERIOD, \
                    (SWTMR_PROC_FUNC)timer2_callback, &swt2_id, 3);
    //启动定时器
    LOS_SwtmrStart(swt1_id);
    LOS_SwtmrStart(swt2_id);
    return 0;
}
```

参考3.2.4节修改Makefile,将源文件添加到变量LOCAL_SRCS,将头文件路径添加到变量LOCAL_INCLUDE。在源文件user_task.c中调用函数demo_swt(),运行结果如图5-2所示。

```
timer1 timeout
timer2 still need 1999 tick overflow
timer2 timeout
system has pass 3003 tick
timer2 timeout
system has pass 6003 tick
timer2 timeout
system has pass 9003 tick
timer2 run 3 times, stop timer2
```

图5-2 运行结果

2) 结果分析

定时器1设置的超时为1000,定时器2设置的超时为3000。从结果看到当定时器1超时之后,定时器2还需要1999个tick才能超时,可见软件定时器在时间上误差很小。由于创建定时器2时为回调函数传入了整型数据3,因此可在其回调函数中通过判断入参停止该定时器。

5.2 原子操作和位操作

13min

5.2.1 计算机中的原子

1. 基本概念

原子即不可分割的最小单位,计算机中的原子操作是指某些指令在执行过程中不被打断,保证数据的可靠性。系统在修改内存数据时,要经过“读取→修改→写入”3个步骤,在多任务系统中此过程可能被打断,因此需要原子操作保证数据的可靠性。

2. 运行机制

由于在芯片中寄存器的速度要快于内存的速度,因此如非必要编译器会选择从寄存器读取数据。从寄存器中读取的数据有可能不是最新的数据,即可能为脏数据。

1) 从C到汇编

一句C代码往往会被编译为多句汇编代码,例如一个简单的赋值语句对应的汇编代码如下:

```
//第5章/代码片段1
//c代码
int a,b;
a = 1;
b = a;
```

```
//第 5 章/汇编代码,变量 b 的内容每次从寄存器获得
第 1 行     .word    20000450     ;变量 a 的地址
第 2 行     .word    20000454     ;变量 b 的地址
第 3 行     mov r0,  #20000450    ;变量 a 的地址→r0
第 4 行     ldr r0,  [r0, #0]     ;变量 a 的内容→r0
第 5 行     mov r1,  r0           ;r0→r1,即 a→r1
第 6 行     mov r2,  #20000454    ;变量 b 的地址→r2
第 7 行     str r1,  [r2, #0]     ; r1→变量 b,即变量 a→变量 b
```

在上述代码片段 1 中，如果汇编的第 4 行到第 5 行之间发生中断并修改变量 a 的值，则地址 20000450 中的内容被修改，而第 5 行 r1 获取的仍然是地址 20000450 的旧值。

2）volatile 关键字

LiteOS 中定义的 Atomic 类型为 volatile int，即在整数前增加了 volatile 关键字。被 volatile 修饰的变量意在告诉编译器此变量容易被修改，读取时应该从内存中重新加载，因此上述 C 代码对应的汇编代码如下：

```
//第 5 章/代码片段 2,变量 b 的内容每次从内存地址获取
第 1 行     .word    20000450     ;变量 a 的地址
第 2 行     .word    20000454     ;变量 b 的地址
第 3 行     mov r0,  #20000450    ;变量 a 的地址→r0
第 4 行     ldr r0,  [r0, #0]     ;变量 a 的内容→r0
第 5 行     mov r2,  #20000454    ;变量 b 的地址→r2
第 6 行     str r0,  [r2, #0]     ;变量 a→变量 b
```

在上述代码片段 2 中，变量 b 每次都从地址 20000450 读取变量 a 的值，因此得到的值永远是最新的数据。

3）LDREX、STREX 指令

在 C 代码中，加法操作 a＝a＋1 要经历 3 个过程：从内存取变量 a，将变量 a 加 1，将变量 a 存储到内存。如果在执行加法操作过程中有中断程序修改变量 a 的值，则结果就是未知的。ARMv6 架构中引入了 LDREX（读取内存）、STREX（将数据保存到内存）指令，这两条指令在操作内存时会设置内存独占标记，由此保证了内存操作的原子性。

注意：ARMv6 之前的架构中没有 LDREX、STREX 指令，因此只能通过开关中断实现加减法的原子操作。

---

3. 原子操作 API

LiteOS 原子操作目前仅支持整型数据，开发者可以对整型数据进行读、写、加、减等操作，见表 5-3。

表 5-3 原子操作 API

函 数	说明(头文件 kernel/include/los_atomic.h)
LOS_AtomicRead	读取一个变量的内容,其本质就是读取被 volatile 修饰的变量,这意味着每次读取都要从内存获取,而非从寄存器获取 返回值:INT32,读到的数据 参数 1:[IN] const Atomic * v,要读取的变量的地址
LOS_AtomicSet	设置一个变量 返回值:INT32,设置的结果 参数 1:[IN] Atomic * v,变量的地址 参数 2:[IN] INT32 setVal,要设置的值
LOS_AtomicAdd	对变量做加法,保证原子操作。通过 LDREX、STREX 指令实现 返回值:INT32,加法的结果 参数 1:[IN] Atomic * v,变量的地址 参数 2:[IN] INT32 setVal,要增加的值
LOS_AtomicSub	对变量做减法,保证原子操作。通过 LDREX、STREX 指令实现 返回值:INT32,减法的结果 参数 1:[IN] Atomic * v,变量的地址 参数 2:[IN] INT32 setVal,要减去的值
LOS_AtomicInc	对变量加 1,保证原子操作。通过 LDREX、STREX 指令实现 返回值:无 参数 1:[IN] Atomic * v,变量的地址
LOS_AtomicDec	对变量减 1,保证原子操作。通过 LDREX、STREX 指令实现 返回值:无 参数 1:[IN] Atomic * v,变量的地址

此外,LiteOS 还支持对 64 位整数进行原子操作,其 API 为 LOS_Atomic64Read()、LOS_Atomic64Set()、LOS_Atomic64Add()、LOS_Atomic64Sub()等。

注意:Cortex-M0、Cortex-M1 属于 ARMv6-M 架构,Cortex-M3、Cortex-M4 属于 ARMv7-M 架构,这两种架构不支持 LDREX 指令,因此可使用开关中断实现变量加减法的原子操作。

4. 实战案例:从汇编看原子操作

1) 案例测试

(1) 创建两个任务,任务 1 操作普通变量,任务 2 操作 Atomic 类型变量,通过汇编文件查看两个任务的不同之处。在 mydemos 文件夹下创建源文件 atomic/atomic.c、头文件 atomic/atomic.h,代码如下:

```
//第 5 章/mydemos/atomic/atomic.c
#include "atomic.h"
```

```
//普通变量
int count_normal = 1;
//Atomic 类型变量,其实就是 volatile int
Atomic count_atomic = 1;
//任务 id
UINT32 tid1;
UINT32 tid2;

UINT32 normal_int(VOID) {
    //变量每次都是 1,编译器会进行优化操作,从寄存器读取变量值
    while(count_normal) {
        ;
    }
    return LOS_OK;
}

UINT32 atomic_int(VOID) {
    //变量被 volatile 修饰,每次读取时都会从内存获取
    while(count_atomic) {
        ;
    }
    return LOS_OK;
}

UINT32 my_task(UINT32 tid, char * name, UINT16 pri, UINT32 stack_size, TSK_ENTRY_FUNC func) {
    //参考 4.1.4 节
    ...
}

UINT32 demo_atomic() {
    UINT32 ret;
    //创建任务
    ret = my_task(&tid1, "read mem", 9, 0x800, normal_int);
    if(ret != LOS_OK) {printf("create normal task failed\n");return - 1;}
    ret = my_task(&tid2, "write mem", 9, 0x800, atomic_int);
    if(ret != LOS_OK) {printf("create atomic task failed\n");return - 1;}
    return ret;
}
```

（2）参考 3.2.4 节修改 Makefile，将源文件添加到 LOCAL_SRCS，将头文件路径添加到 LOCAL_INCLUDE。在源文件 user_task.c 中调用 demo_atomic()函数，编译之后打开文件 out/目标板/Huawei_LiteOS.asm，部分代码如下：

```
; 第 5 章/ Huawei_LiteOS.asm
0800b1a0 < normal_int >:
800b1a0:    b480        push        {r7}
800b1a2:    4b04        ldr         r3, [pc, #16] ; (800b1b4 < normal_int + 0x14 >)
```

```
800b1a4:    6818          ldr         r0, [r3, #0]
800b1a6:    af00          add         r7, sp, #0
800b1a8:    2800          cmp         r0, #0
800b1aa:    d1fd          bne.n       800b1a8 <normal_int+0x8>
800b1ac:    46bd          mov         sp, r7
800b1ae:    f85d 7b04     ldr.w       r7, [sp], #4
800b1b2:    4770          bx          lr
800b1b4:    20000450      .word       0x20000450

0800b1b8 <atomic_int>:
800b1b8:    b480          push        {r7}
800b1ba:    4b04          ldr         r3, [pc, #16] ; (800b1cc <atomic_int+0x14>)
800b1bc:    af00          add         r7, sp, #0
800b1be:    6818          ldr         r0, [r3, #0]
800b1c0:    2800          cmp         r0, #0
800b1c2:    d1fc          bne.n       800b1be <atomic_int+0x6>
800b1c4:    46bd          mov         sp, r7
800b1c6:    f85d 7b04     ldr.w       r7, [sp], #4
800b1ca:    4770          bx          lr
800b1cc:    2000044c      .word       0x2000044c
```

2）结果分析

从 Huawei_LiteOS.asm 汇编代码看到，函数 normal_int()中首先将变量加载到寄存器 r0，之后 while 循环使用 cmp 指令对比 r0 和 0，如果非 0，则跳转到地址 800b1a8，接着重复调用 cmp 指令比较 r0 和 0。函数 Atomic_int()中首先将变量加载到寄存器 r0，之后 while 循环使用 cmp 指令对比 r0 和 0，如果非 0，则跳转到地址 800b1be，接着重新将变量的值加载到寄存器 r0，然后调用 cmp 指令比较 r0 和 0。得出结论，使用 volatile 修饰的变量，每次读取时都会从内存中将数据提取到寄存器，然后读取寄存器的值，这样保证读到的变量值不是脏数据。

5.2.2　位操作

位操作就是对二进制数中的某一位进行操作，C 语言中使用符号“&”“|”“～”对数据进行位操作，它们分别代表按位与、按位或、按位取反。LiteOS 提供了几个位操作的 API，见表 5-4，这些 API 的本质仍然是使用了 C 语言中的 3 种位操作符号。

表 5-4　位操作 API

函　　数	说明（头文件 kernel/include/los_bitmap.h）
LOS_BitmapSet	将变量的某一位置 1 返回值：无 参数 1：[IN] UINT32 * bitmap，变量的地址 参数 2：[IN] UINT16 pos，要置 1 的位置

续表

函　　数	说明（头文件 kernel/include/los_bitmap.h）
LOS_BitmapClr	将变量的某一位清零 返回值：无 参数 1：[IN] UINT32 * bitmap，变量的地址 参数 2：[IN] UINT16 pos，要清零的位置
LOS_LowBitGet	找到二进制数值为 1 的最低一位并返回位索引 返回值：UINT16，索引值 参数 1：[IN] UINT32 * bitmap，变量值
LOS_HighBitGet	找到二进制数值为 1 的最高一位并返回位索引 返回值：UINT16，索引值 参数 1：[IN] UINT32 * bitmap，变量值

19min

5.3　双向循环链表

链表是一种常见的非顺序存储结构，它是操作系统中非常重要的一种数据结构。链表可分为单向链表、双向链表，又可分为循环链表、非循环链表，操作系统中常见的是双向循环链表。

5.3.1　工作原理

链表由若干节点连接而成，每个节点的内部都有一个指针指向下一个节点，由此形成一个链式存储结构。双向链表的节点中不仅包含指向下一个节点的指针，还包含指向上一个节点的指针，通常将这两个指针称为“后继”和“前驱”。双向循环链表将链表的头部和尾部连接在一起，如图 5-3 所示。

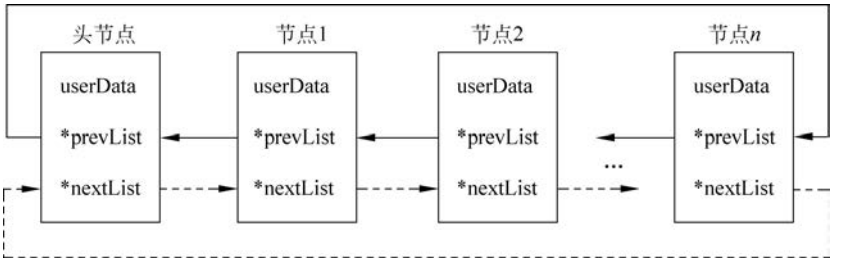

图 5-3　双向链表

LiteOS 双向链表结构定义在头文件 kernel/include/los_list.h 中，通过一系列宏定义向外提供了各种链表操作接口。开发者只要引用该头文件即可完成初始化链表、增加节点、删除节点、遍历链表等操作。其数据结构的代码如下：

```
//第 5 章/kernel/include/los_list.h
//链表结构
```

```
typedef struct LOS_DL_LIST {
    struct LOS_DL_LIST * pstPrev;                    //前驱
    struct LOS_DL_LIST * pstNext;                    //后继
} LOS_DL_LIST;
```

1. 初始化链表

链表在使用前必须初始化头指针，此时链表中没有任何有用的数据节点，因此只需将头指针的前驱和后继都指向自己，代码如下：

```
//第 5 章/kernel/include/los_list.h
VOID LOS_ListInit(LOS_DL_LIST * list) {
    list->pstNext = list;
    list->pstPrev = list;
}
```

开发者在使用时，只需先引入头文件 los_list.h，然后调用函数 LOS_ListInit()，这里需要传入的参数是一个指针。

2. 增加节点

向链表增加节点就会打断原来结构中的两条存储链，进而构造 4 条新的存储链，如图 5-4 所示。

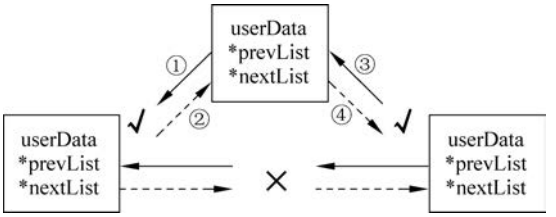

图 5-4 新增节点

使用函数 LOS_ListAdd(LOS_DL_LIST * list, LOS_DL_LIST * node)向链表 list 中添加新节点，其代码如下：

```
//第 5 章/kernel/include/los_list.h
//参数 list 为链表头,参数 node 为新增节点
VOID LOS_ListAdd(LOS_DL_LIST * list, LOS_DL_LIST * node) {
    node->pstNext = list->pstNext;                   //对应图 5-4 中的第④条链
    node->pstPrev = list;                            //对应图 5-4 中的第①条链
    list->pstNext->pstPrev = node;                   //对应图 5-4 中的第③条链
    list->pstNext = node;                            //对应图 5-4 中的第②条链
}
```

从上述代码分析可知，函数 LOS_ListAdd()是在链表的头部插入节点。在 los_list.h 文件中还提供了向尾部插入节点的函数 LOS_ListTailInsert()，代码如下：

```
//第 5 章/kernel/include/los_list.h
LOS_ListTailInsert(LOS_DL_LIST * list, LOS_DL_LIST * node) {
    LOS_ListAdd(list->pstPrev, node);
}
```

3. 删除节点

删除一个节点需要将其两侧的连接都打断，将该节点的前驱和后继直接连接起来，如图 5-5 所示。

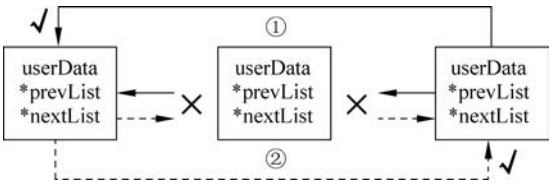

图 5-5　删除节点

函数 LOS_ListDelete(LOS_DL_LIST * node)可删除指定节点，只要将参数设为要删除的节点即可，代码如下：

```
//第 5 章/kernel/include/los_list.h
VOID LOS_ListDelete(LOS_DL_LIST * node) {
    node->pstNext->pstPrev = node->pstPrev;      //对应图 5-5 中的第①条链
    node->pstPrev->pstNext = node->pstNext;      //对应图 5-5 中的第②条链
    node->pstNext = NULL;                        //将删除的节点去除与链表的关联
    node->pstPrev = NULL;
}
```

4. 遍历链表

遍历链表就是对链表中所有节点逐次访问，链表不同于数组，无法使用下标直接访问。由于链表相邻节点之间是关联的，因此只要找到链表中任何一个节点就可完成链表的遍历。

LOS_DL_LIST 结构中并不包含数据域，而开发者使用的链表结构则必定包含数据域，代码如下：

```
typedef struct USER_DATA{
    uint8_t mem1;
    uint8_t mem2;
    LOS_DL_LIST m_list;
}USER_DATA;
```

假设一个链表中有若干 USER_DATA 节点 data_1，data_2，…，data_n，则可通过其成员变量 m_list 完成链表的遍历。如果给出头节点 head，则利用 for 循环可找到每个节点的 m_list 成员，代码如下：

```
for(item = head->pstNext;                    //第 1 个 m_list
    item != head;                            //结束条件
    item = item->next)                       //下一个 m_list
```

上述 for 循环只是找到了 USER_DATA 的 m_list 成员，而实际需要的则是 USER_DATA 这个节点本身，los_list.h 提供了一种通过结构体成员找到整个结构体的方法，代码如下：

```
//第 5 章/kernel/include/los_list.h
#define LOS_DL_LIST_ENTRY(item, type, member) \
    ((type *)(VOID *)((CHAR *)(item) - LOS_OFF_SET_OF(type, member)))
//LOS_OFF_SET_OF(type, member)展开为((type *)0)->member
//(type *)0 将 0 地址强制转换为结构体地址，取其成员变量自然就可得到成员的偏移地址
```

LOS_DL_LIST_ENTRY 的原理是通过某个成员的偏移地址找到整个结构体的地址，其中 item 是成员地址，type 是结构体类型，member 是成员名字。假设链表头是 head，则返回节点 data_1 的代码如下：

```
//通过成员变量返回结构体首地址
LOS_DL_LIST_ENTRY(head->pstNext,             //第 1 个 m_list
                  USER_DATA,                 //结构体类型
                  m_list)                    //成员变量的名字
```

los_list.h 提供了遍历链表的宏定义，其原理就是利用 LOS_DL_LIST_ENTRY 和 for 循环找到链表中的每个用户结构，代码如下：

```
//第 5 章/kernel/include/los_list.h
//item,遍历到的每个用户结构
//list,链表头
//用户结构体类型
//member,用户结构中的 LOS_DL_LIST 类型成员名称
#define LOS_DL_LIST_FOR_EACH_ENTRY(item, list, type, member) \
    for(item = LOS_DL_LIST_ENTRY((list)->pstNext, type, member); \
        &(item)->member != (list); \
        item = LOS_DL_LIST_ENTRY((item)->member.pstNext, type, member))
```

如果要遍历 USER_DATA 链表中的每个节点，则代码如下：

```
LOS_DL_LIST_FOR_EACH_ENTRY(item, &data_head, USER_DATA, m_list)
```

注意：los_list.h 文件中的链表操作与平台和系统无关，也就是说开发者可将此头文件复制到其他 C 或 C++项目中使用。

5.3.2 实战案例：学生管理系统

1. 案例描述

模拟一个简单的学生管理系统，可实现信息的增、删、改、查。本案例旨在练习链表操

作，因此不涉及文件存储和用户交互，所有信息都在内存中。

2. 操作流程

1）数据结构

在头文件 mydemos/list/list.h 中定义负责管理信息的结构类型，在结构体中存储两门功课的成绩和学生姓名，代码如下：

```
//第 5 章 mydemos/list/list.h
#ifndef __MYLIST_H
#define __MYLIST_H

#include "sys_init.h"
#include "los_list.h"

typedef struct STU_DATA{
    LOS_DL_LIST s_list;
    uint8_t chinese;
    uint8_t math;
    uint8_t *name;
}STU_DATA;

VOID demo_list(VOID);

#endif
```

2）初始化数据

在源文件 mydemos/list/list.c 中定义全局链表头 stu_head，并初始化链表，代码如下：

```
//第 5 章 mydemos/list/list.c
LOS_DL_LIST stu_head;

static VOID stu_init(){
    printf("init stu list\n");
    LOS_ListInit(&stu_head);
}
VOID demo_list(VOID){
    stu_init();
}
```

3）插入数据

向链表中插入 5 个学生节点，每次都从尾部插入，代码如下：

```
//第 5 章 mydemos/list/list.c
static VOID stu_insert(uint8_t *name, uint8_t ch, uint8_t math){
    STU_DATA *tmp;

    printf("inser %s chinese %d math %d ...", name, ch, math);
    //申请内存
```

```
    tmp = (STU_DATA *)malloc(sizeof(STU_DATA));
    if(tmp == NULL){
        printf("failed\n");
        return;
    }
    tmp->chinese = ch;
    tmp->math = math;
    //申请内存
    tmp->name = (uint8_t *)malloc(sizeof(name));
    if(tmp->name == NULL){
        printf("failed\n");
        return;
    }
    //复制内容
    memcpy(tmp->name, name, sizeof(name));
    //插入节点
    LOS_ListHeadInsert(&stu_head, &(tmp->s_list));
    printf("ok\n");
}

VOID demo_list(VOID){
    stu_init();
    stu_insert("David", 80, 92);
    stu_insert("Mars", 82, 94);
    stu_insert("Alex", 84, 96);
    stu_insert("Tom", 86, 97);
    stu_insert("Jerry", 88, 98);
}
```

4）查找数据

查找过程应该遍历整个链表，因为满足条件的数据未必只有一条，代码如下：

```
//第 5 章 mydemos/list/list.c
static VOID stu_query_by_name(uint8_t *name){
    STU_DATA *tmp;

    printf("query %s ...\n", name);
    //遍历链表
    LOS_DL_LIST_FOR_EACH_ENTRY(tmp, &stu_head, STU_DATA, s_list){
        //比较每个节点的 name 与入参是否相等
        if(strcmp(name, tmp->name) == 0){
            printf("Chinese %d, Maths %d\n", tmp->chinese, tmp->math);
        }
    }
    printf("query %s over\n", name);
}

VOID demo_list(VOID){
    ...
    stu_query_by_name("Tom");
}
```

5）修改数据

修改分为两个过程，首先要找到满足条件的成员结构，然后修改结构中的其他数据，代码如下：

```
//第 5 章 mydemos/list/list.c
static VOID stu_modify_math(uint8_t *name, uint8_t math){
    STU_DATA *tmp;

    printf("modify %s math score...\n", name);
    //遍历链表
    LOS_DL_LIST_FOR_EACH_ENTRY(tmp, &stu_head, STU_DATA, s_list){
        //比较每个节点的 name 与入参是否相等
        if(strcmp(name, tmp->name) == 0){
            //更改数据
            tmp->math = math;
        }
    }
    //重新查询,验证是否修改
    stu_query_by_name(name);
}
VOID demo_list(VOID){
    ...
    stu_modify_math("Tom", 90);
}
```

6）删除数据

删除某个成员信息，如果在创建成员时使用函数 malloc()分配内存，则删除成员后需要使用函数 free()释放内存，代码如下：

```
//第 5 章 mydemos/list/list.c
static VOID stu_delete_by_name(uint8_t *name){
    STU_DATA *tmp, *node;

    printf("delete %s ...\n", name);
    //遍历链表
    LOS_DL_LIST_FOR_EACH_ENTRY(tmp, &stu_head, STU_DATA, s_list){
        //比较每个节点的 name 与入参是否相等
        if(strcmp(name, tmp->name) == 0){
            //找到当前节点的前驱
            node = LOS_DL_LIST_ENTRY(tmp->s_list.pstPrev, STU_DATA, s_list);
            //删除节点
            LOS_ListDelete(&(tmp->s_list));
            //释放内存
            free(tmp->name);
            free(tmp);
            //由于当前节点被删除,因此 tmp 要重新赋值
            //这样 tmp->s_list.pstNext 才有效,for 循环才可继续执行
```

```
                tmp = node;
            }
        }
        stu_query_by_name(name);
    }

    VOID demo_list(VOID){
        ...
        stu_delete_by_name("Jerry");
    }
```

3. 编译运行

将文件 list.c 和 list.h 的相对路径添加到文件 targets/STM32L431_BearPi/Makefile 中，具体参考 3.2.4 节。编译无误后运行结果如图 5-6 所示。

```
init stu list
inser David Chinese 80 Maths 92 ...ok
inser Mars Chinese 82 Maths 94 ...ok
inser Alex Chinese 84 Maths 96 ...ok
inser Tom Chinese 86 Maths 97 ...ok
inser Jerry Chinese 88 Maths 98 ...ok
query Tom ...
Chinese 86, Maths 97
query Tom over
modify Tom Maths score...
query Tom ...
Chinese 86, Maths 90
query Tom over
delete Jerry ...
query Jerry ...
query Jerry over
```

图 5-6　运行结果

5.4　程序员利器 Git

一个成熟的项目必定要经过各种版本的迭代，代码的版本控制曾是一个令无数程序员头疼的问题，GitHub 平台成功解决了代码的版本控制问题。此外，很多大型项目由专人维护，开发者可以在项目的 GitHub 仓库中找到一些问题的解决方案。

5.4.1　Git 工具

GitHub 是一个基于 Git 的分布式版本控制系统，Git 最初是由 Linux 开发者 Linus Torvalds 开发的。尽管 2013 年国内解除了对 GitHub 的封锁，但由于要访问境外服务器，GitHub 访问速度依然较慢，笔者推荐初学者使用国内代码托管平台 Gitee。

Gitee 与 GitHub 都基于 Git 工具，因此代码托管的首要任务是安装 Git。

1. 安装 Git

Windows 平台下可在 Git 官网(https://git-scm.com/download/win)下载 Git 工具，下载后双击安装文件即可安装。Linux/macOS 平台下使用命令安装 Git，命令如下：

```
#Ubuntu 系统安装 Git
sudo apt install git
#macOS 系统安装 Git
brew install git
```

图 5-7　打开 Git 命令行

2. 基础配置

无论将代码托管到 GitHub 还是 Gitee，都需要在对应官网注册账号，并在自己的计算机中配置用户名和邮箱。尽管在 Windows 系统下可使用图形界面操作 Git，笔者还是推荐使用命令行操作。在 Windows 系统中安装好 Git 工具之后，右击即可打开 Git 命令行，如图 5-7 所示。

打开命令行，配置注册过的用户名和邮箱地址，命令如下：

```
git config -global user.name "weijie"
git config -global user.email "xxx.com"
```

5.4.2　代码管理

Git 版本控制以仓库为基本单元，通常一个项目即一个仓库。用户在代码管理平台创建远程仓库，在 PC 端创建本地仓库，之后就可将本地项目代码上传到远程仓库。

1. 创建远程仓库

（1）登录 Gitee 平台，进入个人主页后单击右上角的“＋”创建一个仓库，如图 5-8 所示。

图 5-8　创建仓库

（2）输入仓库信息后，单击“初始化 readme 文件”按钮，如图 5-9 所示。

2. 创建本地仓库

（1）进入 Test 项目根目录下，打开 Git 命令行，为该项目初始化一个本地仓库，命令如下：

```
#初始化 Git 环境
git init
#添加远程仓库地址
```

图 5-9　初始化 readme 文件

```
#HTTPS 地址参考图 5-9
#wj-test 是为远程仓库起的别名,用户可自己设置
git remote add wj-test https://gitee.com/wei-jie/test.git
```

(2) 将远程仓库同步到本地,命令如下:

```
git pull wj-test master
```

(3) Windows 环境下,每次拉取远程仓库都需要输入用户和密码,开发者可在 Git 命令行中做简单配置,只需输入一次用户名和密码。配置命令如下:

```
git config -global credential.helper store
```

3. 同步代码

(1) 将文件添加到仓库,命令如下:

```
#add 就是添加
#点表示当前目录下的所有文件
#用户也可添加指定文件,例如"git add 1.c src/2.c"
git add .
```

(2) 提交本次修改,命令如下:

```
#-m 后的参数是为本次提交设置一个标记,通常以
#时间戳为标记
git commit -m "202212101900"
```

(3) 将本地仓库同步到远程,命令如下:

```
#将本地仓库上传到远程
git push wj-test master
```

(4) 登录 Gitee 平台,进入 Test 仓库查看结果,如图 5-10 所示。

图 5-10　代码提交结果

注意：初始化仓库只需一次，以后每次提交代码都从指令 git add 开始。如果提交代码时使用 SSH 协议，则需要为计算机配置 SSH 证书，其他操作和 HTTPS 协议相同。

5.5 本章小结

本章介绍了 LiteOS 时间管理、原子操作、位操作。时间管理模块严重依赖编译器，若在开发过程中时间转换失败，则可检查编译器版本是否满足要求。原子操作依赖处理器架构，不同的架构实现原子操作的方式不同。Git 工具不仅可以对代码进行远程托管，还可进行团队协作，开发者应该熟练掌握 Git 工具的基本操作。