第 3 章

内 核 基 础

使用任何一种操作系统都离不开进程/线程、IPC 通信、时间管理、异常等核心内容，对于物联网操作系统，开发者还会用到中断、内存、网络协议栈等功能模块。

本章讲解 LiteOS 中的任务、中断、内存、异常等核心功能组件。

3.1 LiteOS 内核架构

本节介绍 LiteOS 基础内核组成，以及启动流程。

3.1.1 基础内核

LiteOS 的内核包括基础内核和增强内核，绝大多数系统移植需要包含基础内核，而增强内核可以根据实际情况裁剪。基础内核又由一个不可裁剪的极小内核和其他可裁剪模块组成，如图 3-1 所示。

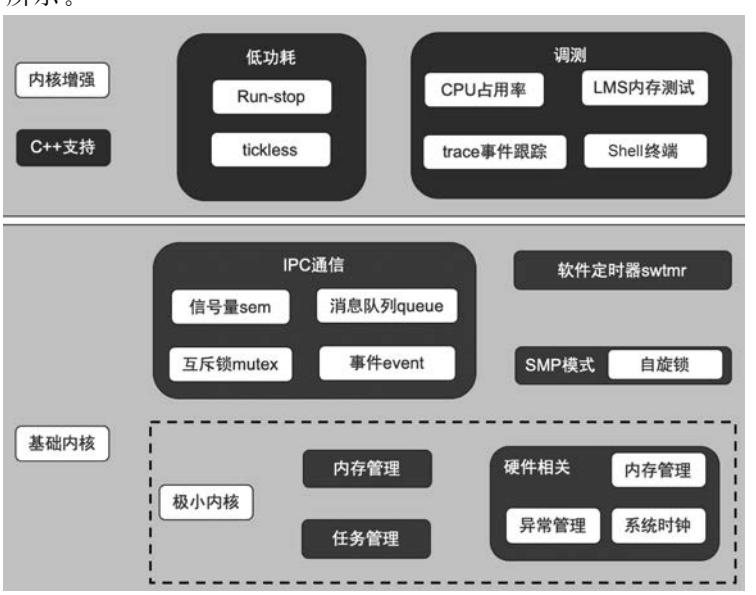

图 3-1 LiteOS 内核

任务管理模块提供了任务的创建、删除、挂起、恢复等功能，同时还提供了任务的调度、锁定、解锁功能。LiteOS 支持抢占式调度，即高优先级任务可以打断低优先级任务，对于优先级相同的任务支持时间片轮转调度。

内存管理模块支持内存的申请和释放，提供内存统计、内存越界检测功能。LiteOS 提供静态内存和动态内存两种算法，目前支持的内存管理算法有 Box 算法、Bestfit 算法、Bestfit_little 算法。

中断管理模块提供了中断的创建、删除、使能及标志位的清除功能；异常管理模块在系统运行异常之后，可以打印出当前发生异常的函数调用栈信息；系统时钟模块负责时钟转换，LiteOS 以 Tick 作为系统调度的基本单位。

IPC 通信模块提供了信号量、互斥锁、事件、消息队列 4 种任务的通信机制，每种通信机制都可以单独进行配置。

注意：在最新版本的 LiteOS 中，中断已经被系统全面接管，不再提供原始的裸机中断接口。

3.1.2 代码结构

LiteOS 官方仓库有若干分支，其中 master 分支的功能最为全面。在使用 master 分支进行开发之前，首先要了解其代码结构，见表 3-1。

表 3-1 LiteOS 目录结构

目　录	说　明	备　注
arch	架构支持，目前支持 ARM64\ARM-A\ARM-M\RISC-V	勿动
build	编译脚本，负责检测编译工具，组织编译代码	macOS 下需要微调，其他系统勿动
compat	兼容性支持，主要是 CMSIS 接口	勿动
component	组件支持，包含 AI、网络、文件系统等组件支持	勿动
doc	说明文档，关于 LiteOS 的一切知识都在这里	在开发过程中可以参考
demos	案例，官方提供的实例程序，例如 MQTT、CoAP 等	供学习测试用，可以修改
drivers	驱动，串口和定时器的一些代码，供调试使用	勿动
include	头文件	勿动
Kernel	内核，LiteOS 核心功能	勿动
lib	第三方库支持，例如解压功能 Zlib	勿动
osdepend	一些依赖	勿动
out	编译输出，以开发板名字命名	自动生成
shell	Shell 命令支持，类似 Windows 的命令行	勿动
targets	目标板支持，列出所有支持的开发板，开发者可以自己添加其他开发板	开发者主要修改的地方

续表

目　录	说　明	备　注
test、tests	测试用案例	勿动
tools	编译和构建工具配置	macOS下需要微调

3.1.3 内核启动流程

在targets目录下列出了许多已经移植成功的开发板，每个开发板对应的目录下都有一个main.c文件，这里就是LiteOS的入口，代码如下：

```
//第3章/main.c
VOID BoardConfig(VOID){
    g_sys_mem_addr_end = __LOS_HEAP_ADDR_END__;
}
INT32 main(VOID){
#ifdef __GNUC__
    ArchStackGuardInit();
#endif
    OsSetMainTask();
    OsCurrTaskSet(OsGetMainTask());

    BoardConfig();
    HardwareInit();

    PRINT_RELEASE("\n******** Hello Huawei LiteOS ******** \n"
                  "\nLiteOS Kernel Version : %s\n"
                  "build date : %s %s\n\n"
                  "**********************************\n",
                  HW_LITEOS_KERNEL_VERSION_STRING, __DATE__, __TIME__);

    UINT32 ret = OsMain();
    if (ret != LOS_OK) {
        return LOS_NOK;
    }

    OsStart();

    return 0;
}
```

首先进行内存结束地址配置BoardConfig()、硬件初始化HardwareInit()，然后打印Huawei LiteOS的版本信息；接着执行函数OsMain()初始化Huawei LiteOS内核及例程，在OsMain()函数中会创建用户任务，其任务处理函数为app_init()；最后调用OsStart()开始任务调度，Huawei LiteOS开始正常工作，具体如图3-2所示。

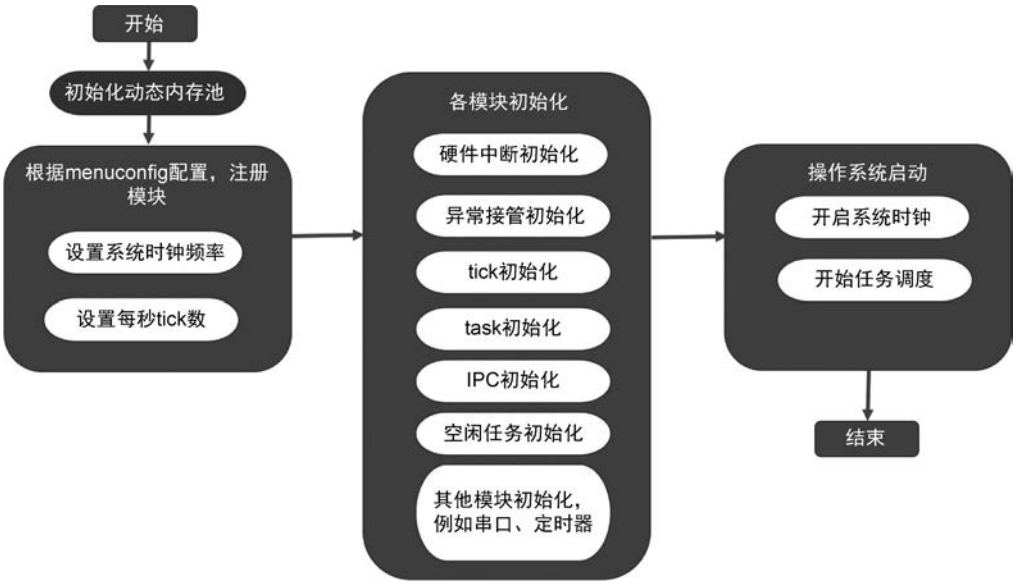

图 3-2　LiteOS 内核启动流程

注意：Huawei LiteOS 提供了一套自有 OS 接口（例如 LOS_TaskDelay），同时也支持 POSIX 接口（例如 sleep）和 CMSIS 接口（例如 osDelay），请勿混用这些接口，否则可能导致未知错误。例如用 POSIX 接口加锁互斥量，但用 Huawei LiteOS 接口解锁互斥量，最终可能导致互斥量无法解锁。开发驱动程序只能用 Huawei LiteOS 的自有接口，上层 App 建议用 POSIX 接口。

3.2　任务

任务是系统的核心，整个操作系统都围绕着各种任务在运行。本节介绍 LiteOS 的任务管理体系。

16min

3.2.1　任务的概念

1. 基本概念

在《UNIX 高级编程》中，作者给出了两个概念：进程是资源分配的最小单位；线程是系统调度的最小单位。Huawei 官网给出的任务概念是：任务是竞争系统资源的最小运行单元。由此可以看出，在 LiteOS 中任务其实就是线程。

LiteOS 的任务管理模块提供了任务的创建、删除、挂起、恢复等操作，可以为用户提供多个任务，每个任务享有独立的内存等资源空间，并且独立于其他任务运行。LiteOS 的任务管理模块具有以下几个特性。

（1）支持多任务。

（2）支持抢占式调度，高优先级的任务可以打断低优先级的任务。

（3）优先级相同的任务支持时间片轮转调度。

（4）有 32 个优先级（0～31），0 为最高优先级，31 为最低优先级。

2. 任务状态

在绝大多数的物联网操作系统中都有任务的概念，无论任务代表的是线程还是进程都具有以下 4 种状态。

（1）就绪态：该任务在就绪队列中，只等待 CPU。

（2）运行态：该任务正在执行。

（3）阻塞态：该任务不在就绪队列中，包含任务被挂起（suspend 状态）、任务被延时（delay 状态）、任务正在等待信号量、读写队列或者等待事件等。

（4）退出态：该任务运行结束，等待系统回收资源。

任务的 4 种状态可以相互切换，如图 3-3 所示。

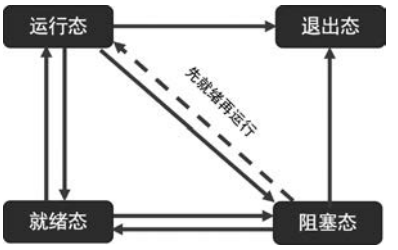

图 3-3　任务状态切换

3. 其他任务相关概念

（1）任务 ID：任务创建成功后通过参数返给用户，系统中的任务 ID 号是唯一的。用户可以通过任务 ID 对指定任务进行任务挂起、恢复、查询、删除等操作。

（2）任务入口函数：当任务得到调度后就会执行此函数。该函数由用户实现，在任务创建时，通过结构体 TSK_INIT_PARAM_S 设置。

（3）任务优先级：代表任务在执行过程中的优先顺序。当发生任务切换时，执行任务就绪队列中优先级最高的任务。

（4）任务栈：任务在运行过程中的独立空间。栈空间里保存的信息包含局部变量、寄存器、函数参数、函数返回地址等。

（5）任务上下文：任务在运行过程中使用的一些资源，如通用寄存器、程序状态字等。当一个任务被挂起时，其他任务继续执行，可能会修改寄存器中的值。如果任务切换时没有保存任务上下文，则可能会导致任务恢复后出现未知错误。任务上下文一般保存在任务栈里。

（6）任务控制块 TCB：每个任务都包含一个任务控制块。TCB 包含了任务上下文栈指针（Stack Pointer）、任务状态、任务优先级、任务 ID、任务名、任务栈大小等信息。TCB 可以反映出每个任务的运行情况，在任务调度时需要用到 TCB。

3.2.2　创建和删除任务


24min

函数 LOS_TaskCreate()可以创建一个任务，当任务被创建之后进入就绪状态。如果就绪队列中没有优先级更高的任务，则运行该任务。函数 LOS_TaskCreate()的各个参数说明见表 3-2。

表 3-2 LOS_TaskCreate 函数说明

返 回 值	类 型	说明（头文件 kernel/include/los_task.h）
	UINT32	如果成功，则返回 LOS_OK，如果失败，则返回 LOS_ERRNO_TSK_XXX
参 数	**类 型**	**说 明**
taskId	UINT32	任务 ID，当任务被创建之后自动回填
initParam	TSK_INIT_PARAM_S	任务参数，是一个结构体
参数 initParam 成员	**类 型**	**说 明**
pfnTaskEntry	void *(*)(void *)	任务的入口地址，这是一个函数指针，返回值为 void *，参数也是 void *
usTaskPrio	UINT16	任务优先级，默认为 10
uwStackSize	UINT32	任务栈大小，默认为 1536 字节
pcName	char *	任务名称，方便在调测时使用
uwResved	UINT32	保留字节

LiteOS 还提供了另外一个创建任务的函数 LOS_TaskCreateOnly()，任务被创建之后并不会进入就绪态，而是被挂起。

无论使用哪个函数创建任务，函数 LOS_TaskDelete()都可以将任务删除，此函数没有参数。

LiteOS 使用 LOS_XXX 来表示一个函数的返回值，例如 LOS_OK 代表成功。用户在创建任务时未必会成功，此时需要通过返回值查找原因。下面给出 LiteOS 任务模块的几个常见错误码，见表 3-3。

表 3-3 任务模块错误码

返 回 值	实 际 数 值	说明（头文件 kernel/include/los_task.h）
LOS_ERRNO_TSK_ID_INVALID	0x02000207	无效的任务 ID
LOS_ERRNO_TSK_NO_MEMORY	0x03000200	内存不足，可以尝试设置更大的动态内存池，或者减少系统支持的最大任务数
LOS_ERRNO_TSK_PTR_NULL	0x02000201	initParam 为空指针
LOS_ERRNO_TSK_PRIOR_ERROR	0x02000203	优先级参数不对，保证优先级在[0,31]
LOS_ERRNO_TSK_ENTRY_NULL	0x02000204	任务入口为空，检查参数 initParam.pfnTaskEntry
LOS_ERRNO_TSK_NAME_EMPTY	0x02000205	任务名是空指针
LOS_ERRNO_TSK_STKSZ_TOO_SMALL	0x02000206	任务栈太小，增加 initParam.uwStackSize
LOS_ERRNO_TSK_ID_INVALID	0x02000207	无效的任务 ID
LOS_ERRNO_TSK_ALREADY_SUSPENDED	0x02000208	挂起任务时，任务已经被挂起

续表

返　回　值	实 际 数 值	说明(头文件 kernel/include/los_task.h)
LOS_ERRNO_TSK_NOT_SUSPENDED	0x02000209	恢复任务时,任务未被挂起
LOS_ERRNO_TSK_DELAY_IN_INT	0x0300020d	在中断内使用任务延时

3.2.3　任务调度

18min

一般情况下,系统按照时间片轮转法则进行任务调度。内核给每个任务分配固定的Tick,当任务的 Tick 时间到达后会进入就绪队列,内核从队列头部取出下一个任务执行。用户也可以通过系统提供的一些函数自由控制任务的执行状态,见表 3-4。

表 3-4　任务状态控制

函　　数	说明(头文件 kernel/include/los_task.h)
LOS_TaskDelay	任务延时等待,释放 CPU,等待时间到期后该任务会重新进入 ready 状态,单位 Tick 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回 LOS_ERRNO_XXX,见表 3-3 参数:[IN] UINT32 tick,延时的时间
LOS_TaskYield	当前任务释放 CPU,并将其移到具有相同优先级的就绪任务队列的末尾,不可以在中断里使用 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回 LOS_ERRNO_TSK_XXX,见表 3-3 参数:无
LOS_TaskSuspend	挂起指定的任务,然后切换任务 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回 LOS_ERRNO_TSK_XXX,见表 3-3 参数:[IN] UINT32 taskId,任务 ID 号
LOS_TaskResume	恢复挂起的任务,使该任务进入 ready 状态 返回值:UINT32,如果成功,则返回 LOS_OK,如果失败,则返回 LOS_ERRNO_TSK_XXX,见表 3-3 参数:[IN] UINT32 taskId,任务 ID 号

任务延时函数 LOS_TaskDelay()以 Tick 为单位,默认每秒 1000 个 Tick。在项目中,任务挂起和恢复应该成对使用;如果使用函数 LOS_TaskCreateOnly()创建任务,则函数 LOS_TaskResume()会多出一个。

进行任务开发时首先要对当前 LiteOS 系统进行简单配置,在终端输入 make menuconfig 指令进入配置菜单,之后选择 Kernel→Basic Config→Task,如图 3-4 所示。

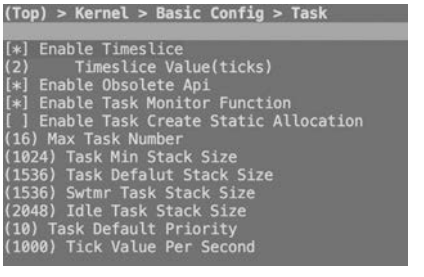

图 3-4　任务配置

3.2.4 实战案例：简单任务控制

本章的案例都存储在 LiteOS-master 根目录下的 mydemos 文件夹中，所有的案例均使用小熊派 STM32L431_BearPi 开发板。为了后续实验操作更快捷，这里首先对工程做统一配置，在 VS Code 中打开 LiteOS-master 目录，进入终端，输入指令 make menuconfig，选择 Targets→Target→STM32L431_BearPi，如图 3-5 所示。

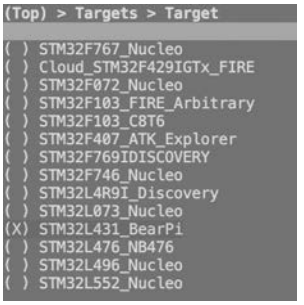

图 3-5 选择目标开发板

本节通过一个简单案例演示任务的操作，有关任务的基础配置，这里选择默认即可。

1. 案例描述

创建两个任务 Task1 和 Task2，两个任务交替执行，每个任务的执行周期为 3s。系统启动之后，首先执行任务 Task1。

2. 操作流程

1）创建源文件

在 LiteOS 源码根目录下创建文件夹 mydemos/task，在 task 文件夹下创建源文件 task.c、头文件 task.h，最终的目录结构如图 3-6 所示。

```
|LiteOS-master
|----...
|----mydemos
     |----task
          |----task.c
          |----task.h
|----...
|----...
```

图 3-6 Task 目录结构

2）编辑源代码

（1）根据案例描述可以分析出，每个任务执行 3s 后需要挂起自己，然后恢复另一个任务，代码如下：

```
//第 3 章/mydemos/task/task.c
//任务计数器
UINT32 secs1, secs2;

UINT32 task1_entry(VOID) {
    while (1) {
        //计数器加 1
        secs1++;
        printf("task1 demo\n");
        if (secs1 % 3 == 0) {
            printf(" ========== \n");
            //恢复任务 2
            LOS_TaskResume(task2_id);
            //挂起任务 1
            LOS_TaskSuspend(task1_id);
        }
        //延时 1000Tick，即 1s
        LOS_TaskDelay(1000);
    }
    return 0;
```

```
}

UINT32 task2_entry(VOID) {
    while (1) {
        //计数器加 1
        secs2++;
        printf("task2 demo\n");
        if (secs2 % 3 == 0) {
            printf("==========\n");
            //恢复任务 1
            LOS_TaskResume(task1_id);
            //挂起任务 2
            LOS_TaskSuspend(task2_id);
        }
        //延时 1000Tick,即 1s
        LOS_TaskDelay(1000);
    }
    return 0;
}
```

(2) 使用函数 LOS_TaskCreate()创建 Task1,为了保证不让 Task2 先运行,这里使用函数 LOS_TaskCreateOnly()创建 Task2,代码如下:

```
//第 3 章/mydemos/task/task.c
UINT32 demo_task(VOID) {
    UINT32 ret;
    TSK_INIT_PARAM_S param;

    //锁住任务调度,防止高优先级任务调度
    LOS_TaskLock();
    //任务名字
    param.pcName = "task1";
    //任务入口地址
    param.pfnTaskEntry = (TSK_ENTRY_FUNC)task1_entry;
    //任务优先级
    param.usTaskPrio = 10;
    //任务栈
    param.uwStackSize = 0x800;
    //调用系统函数,创建任务.成功后任务处于就绪状态
    ret = LOS_TaskCreate(&task1_id, &param);
    if (ret != LOS_OK) {
        printf("create task1 failed, errno = %x\n", ret);
        //如果创建任务失败,则直接返回
        Goto exit_demo;
    }

    param.pcName = "task2";
    param.pfnTaskEntry = (TSK_ENTRY_FUNC)task2_entry;
```

```
    param.usTaskPrio = 10;
    param.uwStackSize = 0x800;
    //创建任务 2,成功后任务处于挂起状态
    ret = LOS_TaskCreateOnly(&task2_id, &param);
    if (ret != LOS_OK) {
        printf("create task2 failed, errno = %x\n", ret);
        goto exit_demo;
    }

exit_demo:
    //解锁任务调度
    LOS_TaskUnlock();

    return ret;
}
```

（3）在头文件 task.h 文件中添加函数声明，代码如下：

```
//第 3 章/mydemos/task/task.h
#ifndef __TASK1_H
#define __TASK1_H

#include "los_task.h"
#include "sys_init.h"

UINT32 demo_task(VOID);

#endif
```

（4）在源文件 targets/STM32L431_BearPi/Src/user_task.c 文件中调用函数 demo_task()，代码如下：

```
VOID app_init(VOID){
    printf("app init!\n");
    DemoEntry();
    demo_task();
}
```

3）修改 Makefile

将文件 task.c 和 task.h 的相对路径添加到文件 targets/STM32L431_BearPi/Makefile 中，代码如下：

```
#第 3 章/targets/STM32L431_BearPi/Makefile
#将源文件 task.c 添加到 LOCAL_SRCS
LOCAL_SRCS += \
    ...
    $(LITEOSTOPDIR)/targets/STM32L431_BearPi/Src/flash_adaptor.c \
```

```
    $(LITEOSTOPDIR)/mydemos/task/task.c
#将头文件路径追加到 LOCAL_INCLUDE
LOCAL_INCLUDE += \
    ...
    -I $(LITEOSTOPDIR)/include \
    -I $(LITEOSTOPDIR)/mydemos/task
```

在 Makefile 中反斜杠“\”代表换行转义，因此以上代码片段实际上是两句话，代码如下：

```
LOCAL_SRCS += ... $(LITEOSTOPDIR)/mydemos/task/task.c
LOCAL_INCLUDE += ... -I $(LITEOSTOPDIR)/mydemos/task
```

3. 运行结果

在终端执行 make 指令，编译无误之后将代码下载到开发板，运行结果如图 3-7 所示。

```
Huawei LiteOS # task1 demo
task1 demo
task1 demo
==========
task2 demo
task2 demo
task2 demo
==========
task1 demo
task1 demo
task1 demo
==========
task2 demo
task2 demo
task2 demo
```

图 3-7　运行结果

3.3　中断

中断是指在程序运行过程中出现了一个必须由 CPU 立即处理的事务，由此导致 CPU 暂停执行当前程序转而执行另外一个程序的过程。

3.3.1　LiteOS 的中断机制

19min

1. 中断相关结构体

Huawei LiteOS 的中断有几个特性：支持配置中断优先级、支持中断嵌套、支持配置中断个数、支持中断共享。与中断相关的核心头文件是 kernel/base/include/los_hwi_pri.h，在这个头文件中定义了两个结构体，即 HwiHandleInfo 和 HwiControllerOps。

HwiHandleInfo 结构体用于记录中断处理程序的相关信息，包括中断处理程序、中断共享模式、共享中断链表、中断处理程序执行的次数，代码如下：

```
//第 3 章/kernel/base/include/los_hwi_pri.h
typedef struct tagHwiHandleForm {
    HWI_PROC_FUNC hook;              /* 中断处理函数 */
    union {
        HWI_ARG_T shareMode;         /* UINT16,共享中断标记位,最高位 1 代表共享中断 */
        HWI_ARG_T registerInfo;      /* 用户注册的中断参数 */
    };
#ifdef LOSCFG_SHARED_IRQ
    struct tagHwiHandleForm *next;   /* 共享中断链表 */
#endif
    UINT32 respCount;                /* 中断调用次数 */
} HwiHandleInfo;
```

HwiControllerOps 结构体定义与中断操作相关的函数，如触发中断、清除中断、使能中断、失能中断、设置中断优先级、获取当前中断号等，代码如下：

```
//第 3 章/kernel/base/include/los_hwi_pri.h
typedef struct {
    UINT32 ( * triggerIrq)(HWI_HANDLE_T hwiNum);          /* 触发中断 */
    UINT32 ( * clearIrq)(HWI_HANDLE_T hwiNum);            /* 清除中断标志 */
    UINT32 ( * enableIrq)(HWI_HANDLE_T hwiNum);           /* 使能中断 */
    UINT32 ( * disableIrq)(HWI_HANDLE_T hwiNum);          /* 失能中断 */
    UINT32 ( * setIrqPriority)(HWI_HANDLE_T hwiNum, UINT8 priority); /* 设置中断优先级 */
    UINT32 ( * getCurIrqNum)(VOID);                       /* 获取当前执行的中断号 */
    CHAR * ( * getIrqVersion)(VOID);                      /* 获取中断版本 */
    HwiHandleInfo * ( * getHandleForm)(HWI_HANDLE_T hwiNum); /* 根据中断号获取中断处理程
序信息. */
    VOID ( * handleIrq)(VOID);                            /* 中断处理句柄 */
#ifdef LOSCFG_KERNEL_SMP
    UINT32 ( * setIrqCpuAffinity)(HWI_HANDLE_T hwiNum, UINT32 cpuMask); /* 设置 CPU 亲和性 */
    UINT32 ( * sendIpi)(UINT32 target, UINT32 ipi);       /* 发送核间中断 */
#endif
} HwiControllerOps;
```

2. 中断注册流程

以 Cortex-M4 架构为例，LiteOS 启动时会调用源文件 kernel/base/los_hwi.c 中的函数 OsHwiInit()进行中断初始化，接着调用源文件 drivers/interrupt/arm_nvic.c 中的函数 ArchIrqInit()完成中断向量初始化，最后用函数 OsHwiControllerReg()注册中断控制器操作句柄，如图 3-8 所示。

图 3-8　LiteOS 中断初始化

3.3.2 创建中断

LiteOS可以通过menuconfig对中断进行简单配置，进入菜单顶层目录后选择Kernel→Interrupt Management可以配置中断参数：[]Enable Interrupt Share代表是否要打开共享中断，()Max Hardware Interrupts代表系统支持的最大中断数，()Interrupt Priority range代表最大中断优先级。

使用函数LOS_HwiCreate()创建中断，此函数有5个参数，见表3-5。

表3-5 LOS_HwiCreate说明

返 回 值	类 型	说明(头文件 kernel/include/los_hwi.h)
	UINT32	如果成功，则返回LOS_OK，如果失败，则返回LOS_ERRNO_HWI_XXX，见表3-6
参 数	**类 型**	**说 明**
hwiNum	HWI_HANDLE_T	中断号，硬件中断由芯片决定
hwiPrio	HWI_PRIOR_T	中断优先级，数字越大，优先级越低。不可以超过最大优先级
hwiMode	HWI_MODE_T	中断模式，0表示普通中断，IRQF_SHARED(0x8000)表示共享中断
hwiHandler	HWI_PROC_FUNC	中断处理函数，用户自定义
irqParam	HWI_IRQ_PARAM_S	中断处理函数的参数，可以为空NULL

第5个参数irqParam是一个结构体，代码如下：

```
//第3章/kernel/base/include/los_hwi_pri.h
typedef struct tagIrqParam {
    int swIrq;                  /* 子中断号 */
    VOID * pDevId;              /* 用来标识产生中断的设备号 */
    const CHAR * pName;         /* 中断的名字,方便用户使用 */
} HWI_IRQ_PARAM_S;
```

中断共享机制，支持不同的设备使用相同的中断号注册同一中断处理程序，但中断处理程序的入参pDevId(设备号)必须唯一，代表不同的设备，即同一中断号，同一dev只能挂载一次，但同一中断号，同一中断处理程序，如果dev不同，则可以重复挂载。

共享中断并不是可以随意使用的，它要求处理器在硬件上已经实现了中断共享。例如，STM32的PA0和PB0都可以挂载到外部中断0，但这并不是LiteOS的共享中断，因为同一时刻只可以挂载一个引脚，而PA5和PA6在硬件上已经同时挂载到EXTI9_5_IRQn，因此可以让PA5和PA6共享中断。

注意：对于Cortex-M系列，0～15为系统中断，因此中断号应该类似EXTI1_IRQn+16。

创建中断未必总是成功的，需要通过返回值找到出错原因，LiteOS中断模块常见错误码见表3-6。

表 3-6 中断模块错误码

返 回 值	十六进制	说明（头文件 kernel/include/los_hwi.h）
LOS_ERRNO_HWI_NUM_INVALID	0x02000900	中断号无效
LOS_ERRNO_HWI_PROC_FUNC_NULL	0x02000901	中断处理程序为空
LOS_ERRNO_HWI_NO_MEMORY	0x02000903	创建中断时内存不足，需要增加动态内存池
LOS_ERRNO_HWI_ALREADY_CREATED	0x02000904	中断号已经被创建。对于非共享中断，应仔细检查是否已经注册过；对于共享中断，应检查中断入口参数的设备 ID 是否重复
LOS_ERRNO_HWI_PRIO_INVALID	0x02000905	中断优先级无效，需要参考硬件手册
LOS_ERRNO_HWI_MODE_INVALID	0x02000906	中断模式无效，应该是 0 或者 1
LOS_ERRNO_HWI_SHARED_ERROR	0x02000909	创建共享中断时没有设置设备 ID，或者要创建非共享中断，但是中断号已经被注册为共享中断

函数 LOS_HwiDelete()可以删除中断，有两个参数：hwiNum 是中断号，irqParam 是中断入口参数。由于用户可能同时为一个中断号注册多个共享中断，因此删除时必须通过入口参数来判断要删除的中断是哪一个。

3.3.3 中断控制

LiteOS 的中断模块为用户提供了若干中断控制接口，具体见表 3-7。

表 3-7 中断控制函数

函 数	说明（头文件 kernel/include/los_hwi.h）
LOS_IntLock	关闭当前处理器的所有中断响应 返回值：UINT32，关中断之前的 CPSR 值 参数：无
LOS_IntUnLock	恢复当前处理器的所有中断响应 返回值：UINT32，打开中断之后的 CPSR 值 参数：无
LOS_IntRestore	让处理器恢复到使用 LOS_IntLock 之前的状态 返回值：无 参数：[IN] UINT32 intSave，关中断之前的 CPSR 值
LOS_HwiEnable	使能指定中断 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_ERRNO_HWI_XXX，见表 3-6 参数：[IN] HWI_HANDLE_T hwiNum，中断号

续表

函　　数	说明(头文件 kernel/include/los_hwi.h)
LOS_HwiDisable	失能指定中断 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_ERRNO_HWI_XXX，见表 3-6 参数：[IN] HWI_HANDLE_T hwiNum，中断号
LOS_HwiSetPriority	设置中断优先级 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_ERRNO_HWI_XXX，见表 3-6 参数 1：[IN] HWI_HANDLE_T hwiNum，中断号 参数 2：[IN] HWI_PRIOR_T priority，优先级

如果在开发过程中使用函数 LOS_IntLock()关闭了全局中断，则在调用函数 LOS_IntUnLock()恢复中断响应之前首先要调用函数 LOS_IntRestore()恢复处理器的状态，并且函数 LOS_IntRestore()的参数必须是函数 LOS_IntLock()的返回值。

虽然 LiteOS 提供了函数 LOS_HwiClear()来清除中断标记位，但是笔者在 STM32 中反复测试此函数没有任何效果，因此推荐开发者使用 HAL 库自带的标记位清除函数。

注意：中断处理函数不能耗时太长，否则会影响 CPU 对中断的响应；另外中断处理程序也不能执行可能引起系统调度的函数，例如 LOS_TaskDelay()。

3.3.4　实战案例：独立中断

21min

1. 案例描述

使用独立中断控制按钮，每按一次输出一条语句。笔者使用 STM32L431_BearPi 开发板，两个按钮分别连接在引脚 PB2、PB3。关于中断的基础配置，这里使用系统默认值。

2. 操作流程

1) 创建源文件

在 mydemos 目录下创建文件夹 hwi，在 hwi 目录下创建源文件 hwi.c 和头文件 hwi.h，最终的目录结构如图 3-9 所示。

```
|LiteOS-master
|----...
|----mydemos
     |----hwi
          |----hwi.c
          |----hwi.h
|----...
|----...
```

图 3-9　目录结构

2) 编辑源代码

(1) 利用 PB2 中断控制输出信息，首先要打开 GPIOB 时钟并且将 PB2 引脚初始化为外部中断状态，代码如下：

```
//第 3 章/mydemos/hwi/hwi.c
void hwi_hard_init(VOID){
    GPIO_InitTypeDef GPIO_InitStruct;
    //打开 GPIOB 时钟
    HAL_RCC_GPIOB_CLK_ENABLE();
    //将 PB2 初始化为外部中断
```

```
    GPIO_InitStruct.Pin = GPIO_PIN_2;
    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

（2）为 PB2 创建中断，设置中断处理函数，代码如下：

```
//第 3 章/mydemos/hwi/hwi.c
void irq_handler(HWI_IRQ_PARAM_S param) {
    //清除标记位
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
    //中断代码
    printf("devid is %d\n", param.pDevId);
    HWI_IRQ_PARAM_S p;
    p.pDevId = 1;
    UINT32 ret;
    //删除中断
    ret = LOS_HwiDelete(EXTI9_5_IRQn + 16, &p);
    if (ret != LOS_OK)
        printf("delete err %x\n", ret);
    //LOS_HwiDisable(EXTI2_IRQn + 16);
}

void demo_hwi(VOID) {
    UINT32 ret;
    HWI_IRQ_PARAM_S param;

    //初始化硬件
    hwi_hard_init();
    //设备号，非共享中断可以不使用
    param.pDevId = 123;
    //中断号，0～15 系统使用，需要加 16
    HWI_HANDLE_T irq_num = EXTI2_IRQn + 16;
    //创建中断：中断号、优先级、是否共享、处理函数、参数传递给处理函数
    ret = LOS_HwiCreate(irq_num, 3, 0, irq_handler, &param);
    if (ret != LOS_OK) {
        printf("hwi err %x\n", ret);
        return;
    }
    //使能中断
    LOS_HwiEnable(irq_num);
}
```

（3）在头文件 hwi.h 中声明函数，并加入需要用到的头文件，代码如下：

```
//第 3 章/mydemos/hwi/hwi.h
#ifndef __HWI_H
```

```
#define __HWI_H

#include "stdio.h"
#include "los_hwi.h"
#include "los_tick.h"
#include "stm32l4xx_hal.h"

void demo_hwi(VOID);

#endif
```

(4) 在源文件 targets/STM32L431_BearPi/Src/user_task.c 中调用函数 demo_hwi()，代码如下：

```
VOID app_init(VOID){
    printf("app init!\n");
    DemoEntry();
    demo_hwi();
}
```

3) 修改 Makefile

将文件 hwi.c 和 hwi.h 的相对路径添加到 targets/STM32L431_BearPi/Makefile 中，代码如下：

```
#第 3 章/targets/STM32L431_BearPi/Makefile
#将源文件 hwi.c 添加到 LOCAL_SRCS
LOCAL_SRCS += \
    ...
    $(LITEOSTOPDIR)/targets/STM32L431_BearPi/Src/flash_adaptor.c \
    $(LITEOSTOPDIR)/mydemos/hwi/hwi.c
#将头文件路径追加到 LOCAL_INCLUDE
LOCAL_INCLUDE += \
    ...
    -I $(LITEOSTOPDIR)/include \
    -I $(LITEOSTOPDIR)/mydemos/hwi
```

3. 运行结果

如果编译无误，则运行结果如图 3-10 所示。

```
Huawei LiteOS # devid is 123
delete err 200090b
devid is 123
delete err 200090b
devid is 123
delete err 200090b
devid is 123
delete err 200090b
```

图 3-10　运行结果

从图 3-10 可以看到，按键每次被按下时都会输出信息。尽管在中断处理函数中调用 LOS_HwiDelete(EXTI9_5_IRQn+16,&p)尝试删除中断，但是其参数 p 对应的 pDevID 是 1，这与初始化时的函数 pDevID(123)并不匹配，因此删除失败。

3.3.5　实战案例：共享中断

35min

1. 案例描述

将 PB6 和 PB7 设置为外部中断(下降沿模式)，使用函数 LOS_HwiCreate()创建共享

中断，每个中断打印自己的设备号和中断号。由于笔者开发板的 PB6 和 PB7 引脚并没有连接按钮，所以使用杜邦线模拟按钮功能。将杜邦线一侧接 GND，另一侧在 PB6、PB7 之间来回切换连接，这样就可以检测出下降沿。

2. 操作流程

（1）本节案例可直接在 3.3.4 节中的代码进行修改。将 PB6 和 PB7 初始化为外部中断模式，检测下降沿中断，代码如下：

```
//第 3 章/mydemos/hwi/hwi.c
void hwi_hard_init_share(VOID) {
    GPIO_InitTypeDef GPIO_InitStruct;
    //使能 GPIOB 时钟
    HAL_RCC_GPIOB_CLK_ENABLE();
    GPIO_InitStruct.Pin = GPIO_PIN_6 | GPIO_PIN_7;
    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

（2）为 PB6 配置设备号 1、子中断号 GPIO_PIN_6，为 PB7 配置设备号 2、子中断号 GPIO_PIN_7。创建两个共享中断，使用同一个中断处理函数，代码如下：

```
//第 3 章/mydemos/hwi/hwi.c
void hwi_demo_share(VOID) {
    UINT32 ret;
    HWI_IRQ_PARAM_S param;

    //初始化硬件
    hwi_hard_init_share();
    //中断号，0～15 系统使用，需要加 16
    HWI_HANDLE_T irq_num = EXTI9_5_IRQn + 16;

    param.pDevId = 1;                   //设备号自定义，不可重复
    param.swIrq = GPIO_PIN_6;           //以引脚为子中断号，方便处理
    //创建共享中断
    //参数依次为中断号、优先级、是否共享、处理函数、参数传递给处理函数
    ret = LOS_HwiCreate(irq_num, 3, IRQF_SHARED, irq_handler_share, &param);
    if (ret != LOS_OK) {
        printf("hwi err %x\n", ret);
        return;
    }
    param.pDevId = 2;                   //设备号
    param.swIrq = GPIO_PIN_7;           //以引脚为子中断号
    //创建中断
    ret = LOS_HwiCreate(irq_num, 3, IRQF_SHARED, irq_handler_share, &param);
    if (ret != LOS_OK) {
        printf("hwi err %x\n", ret);
```

```
        return;
    }
    //使能中断
    LOS_HwiEnable(irq_num);
}
```

(3) 设置中断处理函数,通过参数 param.swIrq 判断中断来源,代码如下:

```
//第 3 章/mydemos/hwi/hwi.c
void irq_handler_share(HWI_IRQ_PARAM_S param) {
    //清除标记位
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6 | GPIO_PIN_7);
    //中断处理代码
if (HAL_GPIO_ReadPin(GPIOB, param.swIrq) == 0) {
    //延时为了消除抖动
    LOS_Udelay(20);
        if (HAL_GPIO_ReadPin(GPIOB, param.swIrq) == 0) {
            printf("INTTERRUPT %d\n", param.swIrq);
        }
    }
}
```

(4) 在头文件 hwi.h 中声明函数,代码如下:

```
//第 3 章/mydemos/hwi/hwi.h
#ifndef __HWI_H
#define __HWI_H

#include "stdio.h"
#include "los_hwi.h"
#include "los_tick.h"
#include "stm32l4xx_hal.h"

void demo_hwi(VOID);
void demo_hwi_share(VOID);

#endif
```

(5) 在源文件 targets/STM32L431_BearPi/Src/user_task.c 中调用函数 demo_hwi(),代码如下:

```
VOID app_init(VOID){
    printf("app init!\n");
    DemoEntry();
    demo_hwi_share();
}
```

3. 运行结果

编译无误后在开发板运行代码，将杜邦线悬空的一侧在 PB6、PB7 之间来回切换，可以看到结果如图 3-11 所示。

```
INTTERRUPT 64
INTTERRUPT 128
INTTERRUPT 64
INTTERRUPT 128
INTTERRUPT 64
INTTERRUPT 128
```

图 3-11　运行效果

16min

3.4　内存

内存管理模块是操作系统的核心模块之一，主要负责内存资源的管理，包括内存的初始化、分配及释放。通过对内存的申请、释放使内存的利用率和使用效率达到最优，最大限度地解决系统的内存碎片问题。

LiteOS 的内存管理分为静态内存管理和动态内存管理，提供了内存初始化、分配、释放等功能。

3.4.1　静态内存

1. 运行机制

静态内存本质上就是一个静态数组，使用期间从静态内存池中申请一个块，用完释放即可。静态内存池由一个内存控制块和若干大小相同的内存块组成，控制块位于内存池的头部，如图 3-12 所示。

图 3-12　静态内存

内存控制块是一个结构体，内部记录了内存池的块个数、块大小、已经使用的块，还有一个记录 Free 节点的链表，代码如下：

```
//第 3 章/kernel/include/los_membox.h
typedef struct {
    UINT32 uwBlkSize;                    /* 静态内存块大小 */
    UINT32 uwBlkNum;                     /* 内存池的总块数 */
    UINT32 uwBlkCnt;                     /* 已经被使用的块数 */
#ifdef LOSCFG_KERNEL_MEMBOX_STATIC
    LOS_MEMBOX_NODE stFreeList;          /* 单向链表，记录未使用的 Free 节点 */
#endif
} LOS_MEMBOX_INFO;
```

每个 Block 并非按照用户的实际要求设定大小，系统在管理内存时会遵循芯片架构的对齐方式，例如 STM32L431 采用的是 4 字节对齐方式。Block 的起始 4 字节是块标记位，如果 Block 被使用，则标记为 0xa55a5aa5。

注意：静态内存在使用时首先要进入 menuconfig，选择 Kernel→Memory Management 使能 Membox Management。

2. 静态内存 API

静态内存使用前必须初始化内存池并设置块大小，初始化之后块大小不可以改变。内存管理模块为静态内存提供分配、释放等一系列操作，见表 3-8。

表 3-8　静态内存 API

函　　数	说明（头文件 kernel/include/los_membox.h）
LOS_MemboxInit	初始化静态内存池 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] VOID * pool，内存池首地址 参数 2：[IN] UINT32 poolSize，内存池大小 参数 3：[IN] UINT32 blkSize，块大小
LOS_MemboxAlloc	申请一块静态内存 返回值：VOID *，如果成功，则返回内存地址，如果失败，则返回 NULL 参数 1：[IN] VOID * pool，内存池地址
LOS_MemboxFree	释放一块静态内存 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] VOID * pool，内存池地址 参数 2：[IN] VOID * box，要释放的内存块
LOS_MemboxClr	清零指定内存块，无返回值。如果成功，则内存的内容为 0 返回值：无 参数 1：[IN] VOID * pool，内存池地址 参数 2：[IN] VOID * box，要清零的内存块
LOS_MemboxStatisticsGet	获取指定静态内存池的信息，如果执行成功，则可从参数中获取内存信息 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] const VOID * boxMem，内存池地址 参数 2：[OUT] UINT32 * maxBlk，内存池块的总个数 参数 3：[OUT] UINT32 * blkCnt，已经使用的块个数 参数 4：[OUT] UINT32 * blkSize，块的大小
LOS_ShowBox	打印指定静态内存池的所有节点信息（打印等级是 LOS_INFO_LEVEL），包括内存池起始地址、内存块大小、总内存块数量、每个空闲内存块的起始地址、所有内存块的起始地址 返回值：无 参数：[IN] VOID * pool，内存池地址

3. 实战案例：申请静态内存

1）案例测试

初始化一个 200 字节的静态内存池并将块大小设置为 10 字节，申请两个内存块，对得

到的内存块进行赋值、清零、释放等操作。在 mydemos 文件夹下创建源文件 mem/mem.c、头文件 mem/mem.h，代码如下：

```
//第 3 章/mydemos/mem/mem.c
void demo_static_mem(){
    UINT32 ret, * mem1, * mem2;

    //初始化内存池,指定大小为 200,块大小为 10
    ret = LOS_MemboxInit(&mem_box[0], box_size, block_size);
    if(ret != LOS_OK){ printf("静态内存池初始化失败\n"); return; }
    printf("内存池起始地址是 0x%x\n",mem_box);
    //打印控制块
    printf("块大小 %d 块总数 %d 已经使用块数 %d 下一个可以分配的块地址 0x%x\n",
            mem_box[0], mem_box[1], mem_box[2], mem_box[3]);

    //申请 mem1
    mem1 = (UINT32 * )LOS_MemboxAlloc(mem_box);
    if(mem1 == NULL){ printf("申请静态 mem1 失败\n"); return;}
    printf("mem1 申请成功\n");
    //打印控制块
    printf("块大小 %d 块总数 %d 已经使用块数 %d 下一个可以分配的块地址 0x%x\n",
            mem_box[0], mem_box[1], mem_box[2], mem_box[3]);

    //申请 mem2
    mem2 = (UINT32 * )LOS_MemboxAlloc(mem_box);
    if(mem2 == NULL){ printf("申请静态 mem2 失败\n"); return;}
    printf("mem2 申请成功\n");
    //打印控制块
    printf("块大小 %d 块总数 %d 已经使用块数 %d 下一个可以分配的块地址 0x%x\n",
            mem_box[0], mem_box[1], mem_box[2], mem_box[3]);

    //内存赋值
    * mem1 = 543;
    * mem2 = 123;
    printf("mem1 内容是 %d, 地址是 0x%x, 保护字是 0x%x\n",
        * mem1, mem1, * (mem1 - 1));
    printf("mem2 内容是 %d, 地址是 0x%x, 保护字是 0x%x\n",
        * mem2, mem2, * (mem2 - 1));

    //清空内存 1
    LOS_MemboxClr(mem_box, mem1);
    printf("mem1 的内容是 %d, mem2 的内容是 %d\n", * mem1, * mem2);

    //释放内存
    ret = LOS_MemboxFree(mem_box, mem1);
    if(ret != LOS_OK){ printf("释放 mem1 失败\n");}
    ret = LOS_MemboxFree(mem_box, mem2);
    if(ret != LOS_OK){ printf("释放 mem2 失败\n");}

    return;
}
```

参考 3.2.4 节内容，修改 Makefile 并在源文件 user_task.c 中调用函数 demo_static_mem()，运行结果如图 3-13 所示。

```
app init!
内存池起始地址是 0x20001e68
块大小 16 块总数 11 已经使用块数 0 下一个可以分配的块地址 0x20001e78
mem1 申请成功
块大小 16 块总数 11 已经使用块数 1 下一个可以分配的块地址 0x20001e88
mem2 申请成功
块大小 16 块总数 11 已经使用块数 2 下一个可以分配的块地址 0x20001e98
mem1 内容是 543, 地址是 0x20001e7c, 保护字是 0xa55a5aa5
mem2 内容是 123, 地址是 0x20001e8c, 保护字是 0xa55a5aa5
mem1的内容是 0, mem2的内容是 123
```

图 3-13　运行结果

2）结果分析

内存池首地址是 0x20001e68，控制块是 LOS_MEMBOX_INFO（参考 3.4.1 节）类型，占 4x4 字节。由于每个 Block 需要 4 字节保护字段，因此用户第 1 次分配到的内存地址是 0x20002068＋16＋4＝40x20001e7c。由于本次实验使用的芯片 STM32L431 采用 4 字节对齐方式管理内存，因此每个 Block 的实际大小是 12＋4＝16。

从图 3-13 可以看到，控制块内存放的下一个节点地址并非用户得到的实际地址，因为要设置 4 字节的标志位，最终用户得到的地址是 stFreeList＋4。

3.4.2　动态内存

动态内存属于堆内存，在内存资源足够的情况下，从一块连续的堆内存（动态内存池）中为用户分配任意大小的块。当用户不需要此内存时，又可以释放回收。很明显动态内存是按需分配的，但是容易出现内存碎片。

LiteOS 动态内存支持 BestFit、BestFit_Little 两种算法。

1. BestFit 算法

BestFit 内存结构包含 3 部分：内存池信息、管理节点、用户内存区域，如图 3-14 所示。

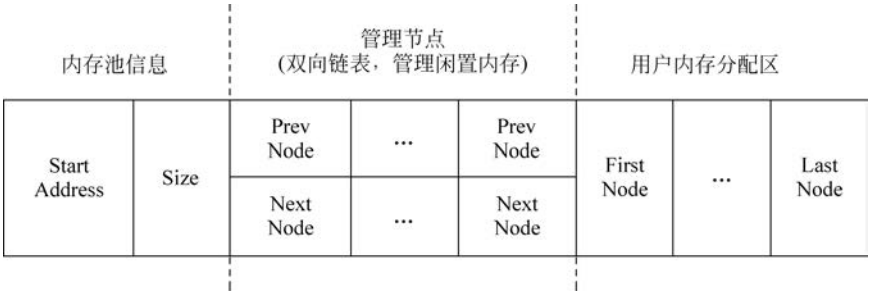

图 3-14　BestFit 算法

第一部分记录了内存池起始地址和总大小。

第二部分本质上是一个数组，数组的每个元素都是一个双向链表，所有的 Free 节点都按照规定挂载到链表中。假设内存中允许的最小节点是 2^min 字节，则第 1 个双向链表挂

载 size 为 2^min＜size＜2^min＋1 的 Free 节点，第 n 个双向链表挂载 2^min＋n－1＜size＜2^min＋n 的 Free 节点。每次申请内存时，系统会从数组找到大小最合适的 Free 节点；释放时内存会重新挂载到指定链表。例如规定最小节点为 32 字节(2^5)，当用户申请 67 字节的内存时，会从数组的第 3 个链表中分配内存，因为 2^6＜67＜2^7。

第三部分是用户实际使用的区域，占内存池多数空间，每个节点都是 LosMemDynNode 类型的结构体，该结构体的代码如下：

```
//第 3 章/kernel/base/mem/bestfit/los_memory_internal.h
typedef struct {
    union {
        LOS_DL_LIST freeNodeInfo;                    /* Free memory node */
        struct {
            UINT32 magic;
            UINT32 taskId : 16;
        };
    };
    struct tagLosMemDynNode *preNode;
    UINT32 sizeAndFlag;
} LosMemCtlNode;

typedef struct tagLosMemDynNode {
    LosMemCtlNode selfNode;
} LosMemDynNode;
```

关于 LosMemDynNode 结构的说明如图 3-15 所示。

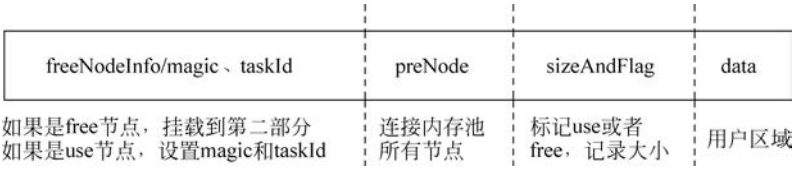

图 3-15　LosMemDynNode 结构

注意：BestFit 在使用时首先要进入 menuconfig，选择 Kernel→Memory Management→Dynamic Memory Management，使能 BestFit。

2. BestFit_Little 算法

BestFit_Little 算法在最佳适配算法(BestFit)的基础上增加 Slab 机制后形成，Slab 机制可以用来分配固定大小的内存块，减少内存碎片。BestFit_Little 算法如图 3-16 所示。

假设动态内存池中有 4 个 Slab Class，每个 Slab Class 最大为 512 字节，系统按照 BestFit 算法分配这 4 个 Slab Class：第 1 个分割为 32 个 16 字节的 Slab 块，第 2 个分割为 16 个 32 字节的 Slab 块，第 3 个分割为 8 个 64 字节的 Slab 块，第 4 个分割为 4 个 128 字节的 Slab 块。

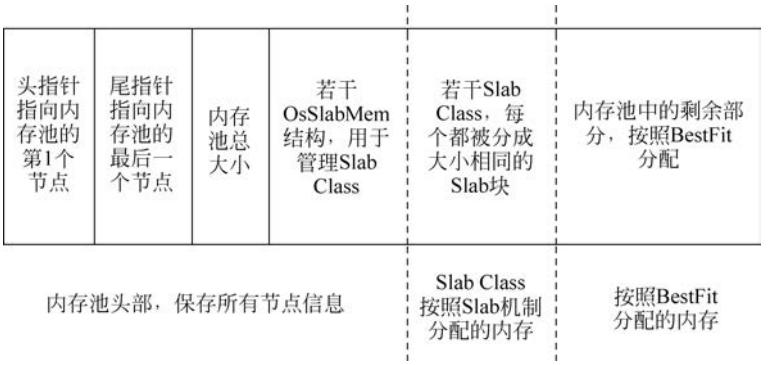

图 3-16　BestFit_Little 算法

初始化时，首先为内存池头部管理节点分配一定空间，接着按照 BestFit 算法申请 4 个 Slab Class，并将每个 Slab Class 初始化(32 个 16 字节、16 个 32 字节、8 个 64 字节、4 个 128 字节)，最后剩下的区域还是按照 BestFit 管理。

每次申请内存时先检索 Slab Class，如果成功，则返回 Slab 中的内存块，释放时继续挂载到 Slab；如果失败，则从第三部分中按照 BestFit 算法继续申请内存。例如申请 25 字节内存，则从 32 字节的 Slab 中分配，如果 32 字节的 Slab 已经用完，则从第三部分按照 BestFit 算法申请。

注意：BestFit _ Little 在使用时首先要进入 menuconfig，选择 Kernel → Memory Management，使能 Mem Slab Extention；选择 Kernel→Memory Management→Dynamic Memory Management，使能 BestFit_Little。

LiteOS 内存管理模块为用户提供了动态内存的初始化、申请、释放等功能，见表 3-9。

表 3-9　动态内存 API

函　　数	说明(头文件 kernel/include/los_memory. h)
LOS_MemInit	初始化一块指定的动态内存池，大小为 size 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] VOID * pool，内存池首地址 参数 2：[IN] UINT32 size，内存池大小
LOS_MemDeInit	删除指定内存池，仅打开 LOSCFG_MEM_MUL_POOL 时有效 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数：[IN] VOID * pool，内存池首地址
LOS_MemAlloc	从指定动态内存池中申请内存 返回值：VOID *，如果成功，则返回块指针，如果失败，则返回 NULL 参数 1：[IN] VOID * pool，内存池地址 参数 2：[IN] UINT32 size，要申请的内存大小

续表

函　　数	说明（头文件 kernel/include/los_memory.h）
LOS_MemFree	释放已申请的内存 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] VOID * pool，内存池地址 参数 2：[IN] VOID * ptr，要释放的内存地址
LOS_MemRealloc	重新分配内存块，并将原内存块内容复制到新内存块。如果新内存块申请成功，则释放原内存块 返回值：VOID *，如果成功，则返回块指针，如果失败，则返回 NULL 参数 1：[IN] VOID * pool 内存池地址 参数 2：[IN] VOID * ptr，原始内存地址 参数 3：[IN] UINT32 size 需要重新申请的内存大小
LOS_MemPoolSizeGet	获取指定动态内存池的总大小 返回值：UINT32，如果成功，则返回内存池大小，如果失败，则返回 LOS_NOK 参数：[IN] const VOID * pool，内存池地址
LOS_MemTotalUsedGet	获取指定动态内存池的总使用量大小 返回值：UINT32，如果成功，则返回使用量，如果失败，则返回 LOS_NOK 参数：[IN] VOID * pool，内存池地址
LOS_MemInfoGet	获取指定内存池的内存结构信息，包括空闲内存大小、已使用内存大小、空闲内存块数量、已使用的内存块数量、最大的空闲内存块大小 返回值：UINT32，如果成功，则返回 LOS_OK，如果失败，则返回 LOS_NOK 参数 1：[IN] VOID * pool，内存池地址 参数 2：[OUT] LOS_MEM_POOL_STATUS * poolStatus，获取的状态
LOS_MemFreeBlksGet	获取指定内存池的空闲内存块数量 返回值：UINT32，如果成功，则返回数量，如果失败，则返回 LOS_NOK 参数：[IN] VOID * pool，内存池地址
LOS_MemUsedBlksGet	获取指定内存池已使用的内存块数量 返回值：UINT32，如果成功，则返回数量，如果失败，则返回 LOS_NOK 参数：[IN] 内存池地址

3. 实战案例：申请动态内存

1）案例测试

定义一个 1KB 的全局数组作为动态内存池，初始化之后首先申请 2 字节内存并赋值，接着在原来的基础上重新申请 4 字节内存。本节案例可直接在 3.4.1 节的基础上进行修改，代码如下：

```
//第 3 章/mydemos/mem/mem.c
void demo_dynamic_mem(){
```

```
    UINT16 * mem, ret;

    printf("LosMemDynNode 节点大小 %d\n", sizeof(LosMemDynNode));
    //初始化内存池
    ret = LOS_MemInit(mem_pool, 1024);
    if(ret != LOS_OK){ printf("动态内存初始化失败\n"); return; }
    else { printf("动态内存初始化成功\n"); }
    print_mem_pool();

    //申请 4 字节内存
    mem = (UINT16 * )LOS_MemAlloc(mem_pool, 2);
    if(mem == NULL) { printf("申请内存失败\n"); return; }
    printf("内存分配成功\n");
    //赋值
    * mem = 0x1234;
    printf("mem 内容是 0x%x\n", * mem);
    print_mem_pool();

    //申请 4 字节内存
    mem = (UINT16 * )LOS_MemRealloc(mem_pool, mem, 4);
    if(mem == NULL) { printf("申请内存失败\n"); return; }
    printf("重新分配内存成功\n");
    * (mem + 1) = 0x5678;
    printf("mem 内容是 0x%x\n", * (UINT32 * )mem);
    print_mem_pool();

    //释放内存
    ret = LOS_MemFree(mem_pool, mem);
    if(ret != LOS_OK) { printf("释放内存失败\n"); }
    else { printf("释放内存成功\n"); }

    return;
}
```

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 中调用函数 demo_dynamic_mem()，运行结果如图 3-17 所示。

2）结果分析

每个 Block 内存都包含 LosMemDynNode 结构，剩余的才是用户 Data 区。第 1 次申请成功后内存增加了 16＋2＝18 字节，然而内存管理需要进行 4 字节对齐，因此最终增加了 16＋4＝20 字节。

当第 2 次使用函数 LOS_MemRealloc()重新申请内存时，首先要将之前的内存（内容是 0x1234）复制到新内存，因此最终显示的内容是 0x56781234。由于第 1 次的内存遵循 4

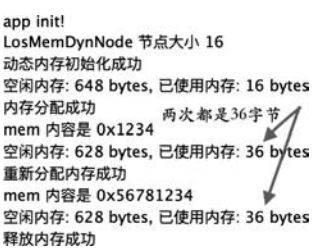

图 3-17　运行结果

字节对齐原则有多余的 2 字节，而 Realloc 时正好利用这 2 字节，所以第 2 次重新申请并没有增加内存空间。

13min

3.5 错误码和异常处理

当程序出现问题时，通过错误码和异常信息可以准确地定位出错位置。

3.5.1 错误码

错误码是一个 4 字节的无符号整数，从左到右第 1 字节表示错误等级，见表 3-10；第 2 字节表示错误码标志，目前该标志是 0；第 3 字节代表错误码所属的模块，例如任务的错误码模块是 0x2；第 4 字节代表错误码序号，具体要参考每个模块是如何定义的。

表 3-10 错误等级

函　数	数　值	说　明
NORMAL	0	提示
WARN	1	告警
ERR	2	错误
FATAL	3	致命

在头文件 Kernel/include/los_errno.h 中定义了错误码对应的模块，部分代码如下：

```
//第 3 章/kernel/include/los_errno.h
enum LOS_MOUDLE_ID {
    LOS_MOD_SYS = 0x0,                  /** 系统 */
    LOS_MOD_MEM = 0x1,                  /** 动态内存模块 */
    LOS_MOD_TSK = 0x2,                  /** 任务模块 */
    LOS_MOD_SWTMR = 0x3,                /** 软件定时器模块 */
    LOS_MOD_TICK = 0x4,                 /** Tick 模块 */
    LOS_MOD_MSG = 0x5,                  /** 消息模块 */
    LOS_MOD_QUE = 0x6,                  /** 队列模块 */
    LOS_MOD_SEM = 0x7,                  /** 信号量模块 */
    LOS_MOD_MBOX = 0x8,                 /** 静态内存模块 */
    LOS_MOD_HWI = 0x9,                  /** 硬件中断模块 */
    …
    LOS_MOD_DRIVER = 0x41,              /** 驱动模块 */
    LOS_MOD_BUTT                        /** 结束标志 */
};
```

以中断错误码 LOS_ERRNO_HWI_NUM_INVALID 为例：中断的错误码模块是 LOS_MOD_HWI(0x9)，错误序号是 0，错误等级是 2，最终得到错误码的代码如下：

```
//第 3 章/kernel/include/los_errno.h
#define LOS_ERRNO_HWI_NUM_INVALID \
LOS_ERRNO_OS_ERROR(LOS_MOD_HWI, 0x00)
```

```
#define LOS_ERRNO_OS_ERROR(MID, ERRNO) \
    (LOS_ERRTYPE_ERROR | LOS_ERRNO_OS_ID | ((UINT32)(MID) << 8) | \
    ((UINT32)(ERRNO)))

#define LOS_ERRTYPE_ERROR (0x02U << 24)
#define LOS_ERRNO_OS_ID (0x00U << 16)
//mid = 0x9 ERRNO = 0
//由上面代码计算 LOS_ERRNO_HWI_NUM_INVALID = 0x02000900
```

3.5.2　异常处理

异常接管是一种调测手段，当系统发生异常时向用户提供有用的异常信息。LiteOS 的异常接管功能可以打印异常发生时的任务信息、调用栈信息、CPU 现场信息、任务的堆栈信息等。为了查看详细的异常信息，首先要进入 menuconfig 菜单，选择 Debug→[*]Enable Backtrace。

1. 运行机制

每个函数都有自己的栈空间，称为栈帧。调用函数时，会创建子函数的栈帧，同时将函数入口参数、局部变量、寄存器入栈。栈帧从高地址向低地址增长。以 ARM32 架构为例，每个栈帧中都会保存 PC、LR、SP 和 FP 寄存器的历史值。LiteOS 运行期间栈帧结构如图 3-18 所示。

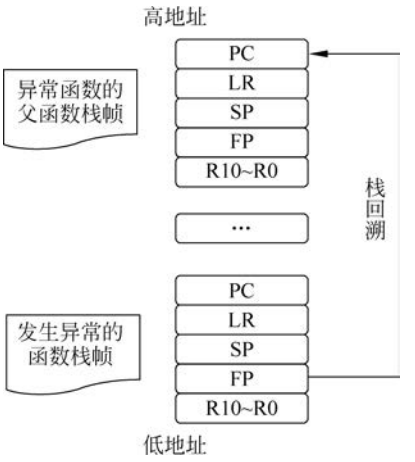

图 3-18　栈帧结构

（1）PC 寄存器：程序计数器，指向下一条要执行的指令。

（2）LR 寄存器：链接寄存器，指向函数的返回地址。

（3）FP 寄存器：帧指针寄存器，指向当前函数的父函数的栈帧起始地址。

从图 3-18 可以看出每个函数的栈帧顶部是 PC，而 PC 则存放着下一条要执行的指令，因此找到 FP 其实就是找到了发生异常的函数。LiteOS 的 Backtrace 功能可以追踪异常信息，打印出发生异常时的 FP 和 LR 值，根据 FP 可以追溯到发生异常的代码。

2. 实战案例：异常追溯

1）案例测试

LiteOS 提供了函数 LOS_Panic()，以此来触发软中断异常。本案例触发一个软中断异常，根据 Backtrace 信息定位异常发生的位置。在 mydemos 目录下创建源文件 exc/exc.c、头文件 exc/exc.h，代码如下：

```
//第 3 章/mydemos/exc/exc.c
#include "exc.h"
```

```
void exc_entry() {
    printf("Test Exception\n");
    //触发软中断异常
    LOS_Panic("Kernel panic\n");
}

void demo_exc() {
    UINT32 ret, task_id;
    TSK_INIT_PARAM_S param;

    //锁住任务调度,防止高优先级任务调度
    LOS_TaskLock();
    //任务名字
    param.pcName = "task_exc";
    //任务入口地址
    param.pfnTaskEntry = (TSK_ENTRY_FUNC)exc_entry;
    //任务优先级
    param.usTaskPrio = 10;
    //任务栈
    param.uwStackSize = 0x800;
    //调用系统函数,创建任务.成功后任务处于就绪状态
    ret = LOS_TaskCreate(&task_id, &param);
    if (ret != LOS_OK) {
        printf("create task_exc failed, errno = %x\n", ret);
        //如果创建任务失败,则直接返回
        Goto exit_demo;
    }
exit_demo:
    //解锁任务调度
    LOS_TaskUnlock();

    return ret;
}
```

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 中调用函数 demo_exc(),运行结果如图 3-19 所示。

2）结果分析

异常管理模块首先打印出发生异常的任务信息,通过 TaskName 字段可以知道发生异常的任务是 task_exc。

打开编译后生成的 ASM 反汇编文件,默认为 out/platform/Huawei_LiteOS.asm,例如当前小熊派对应的是反汇编文件 out/STM32L431_BearPi/Huawei_LiteOS.asm。搜索 Backtrace 第 1 条信息中的 FP 值(要去掉 0x),如图 3-20 所示。

从图 3-20 看到发生异常的函数是 ArchHaltCpu(),继续搜索第 2 条信息中的 FP 值,结果如图 3-21 所示。

```
app init!
Test Exception
kernel panic
TaskName = task_exc
TaskId = 3
Task stackSize = 2048
System mem addr = 0x200024b8
Phase    = fault in task
Type     = 0x0
FaultAddr  = 0xabababab
intNumOrTaskId   = 0x3
R0       = 0x8011e30
R1       = 0x0
R2       = 0x2000041c
R3       = 0x0
R4       = 0x800277a
R5       = 0x1000200
R6       = 0x200053b8
R7       = 0x200053b8
R8       = 0xcacacaca
R9       = 0x8011e30
R10      = 0x200053d4
R11      = 0x2000041c
R12      = 0x200053e8
PriMask    = 0x200053e0
SP       = 0x800b1b1
LR       = 0x8004d77
PC       = 0x3
xPSR     = 0x3
*******backtrace begin*******
traceback 1 -- lr = 0x080027d4 -- fp = 0x08002774
traceback 2 -- lr = 0x0800b1b0 -- fp = 0x080027ac
traceback 3 -- lr = 0x08004d76 -- fp = 0x0800b1a0
*******backtrace end*******
```

图 3-19　异常运行结果

```
08002774 <ArchHaltCpu>:
 8002774:   b480        push    {r7}
 8002776:   af00        add r7, sp, #0
 8002778:   df00        svc 0
 800277a:   bf00        nop
 800277c:   46bd        mov sp, r7
 800277e:   f85d 7b04   ldr.w   r7, [sp], #4
 8002782:   4770        bx  lr
```

图 3-20　异常函数

从图 3-21 看到发生异常的第 1 层父函数是 LOS_Panic()，由于函数 LOS_Panic()可能被系统调用，因此还要继续搜索下一条 Backtrace 信息中的 FP 值，结果如图 3-22 所示。

```
080027ac <LOS_Panic>:
 80027ac:   b40f        push    {r0, r1, r2, r3}
```

图 3-21　第 1 层父函数

```
0800b1a0 <exc_entry>:
 800b1a0:   b580        push    {r7, lr}
```

图 3-22　第 2 层父函数

从图 3-22 看到引发异常的第 2 层父函数是 exc_entry()，而此函数对应的任务信息也符合图 3-19 显示的内容。综上所述，发生异常的函数依次是 exc_entry()→LOS_Panic()→ArchHaltCpu()。

3.6　认识 Makefile

23min

在实际项目开发中，往往有成百上千个源文件，为了有效地管理整个工程，GNU 推出了 Make 工程管理工具，Makefile 是 Make 工具的配置文件。Windows 系统下通常用 IDE 环境管理工程，IDE 集成了 Make 工具，并且会自动生成 Makefile。UNIX 环境下则需要开发者自己配置 Make 工具，编写 Makefile 文件。

Makefile 就像一个 Shell 脚本一样，当键入 Make 指令后，系统会按照特定的规则执行 Makefile 中的命令。

3.6.1 基础语法

1. 语法格式

Makefile 的核心是规则，而规则按照“目标：依赖 命令”的格式书写，其中“命令”需要另起一行且以 Tab 开头，代码如下：

```
cal: add.c sub.c
    gcc add.c sub.c -o cal
```

目标：编译生成的结果文件。如果目标的更新时间比依赖的更新时间晚，则不需要重新编译，否则就需要重新编译并更新目标。在默认情况下，第 1 个目标就是 Makefile 的最终目标。

依赖：目标文件由哪些文件组成。

命令如何生成目标文件。命令必须以 Tab 开头，不可以用空格代替。

Makefile 使用“#”注释内容，echo 可以输出内容，类似 C 语言的 printf。通常在 echo 前加一个符号“@”，这样编译过程就不显示 echo 语句本身。例如在编译结束后显示 build success，代码如下：

```
@echo " --- build success --- "
#在终端可以看到编译成功后输出 --- build success ---
```

make 在生成目标时会逐层寻找依赖关系，并最终生成第 1 个目标文件。如果在寻找过程中出现错误，则直接退出，并提示错误。

2. 变量

1）变量赋值

如果一个名字后加上等号，则这个名字就是一个变量。等号是最普通的赋值方式，使用“=”赋值时变量是最后被指定的值。Makefile 还提供以下几种赋值方式：

（1）“:=”是直接赋值，给变量赋予当前位置的值。

（2）“?=”表示如果当前变量没有被赋值，就赋予等号后的值。

（3）“+=”和 C 代码一样，表示给变量追加一个值。

当给变量赋值一个很长的字符串时，代码写在同一行并不美观，此时需要用到转义字符“\”，它表明下一行代码也属于当前这一行。一个简单的赋值案例，代码如下：

```
#等价于 src += 1.c 2.c 3.c
src += 1.c \
    2.c \
    3.c
```

2）取变量的值

“$”符号表示取变量的值，当变量有多个字符时，使用“()”。此外“$”还有一些特殊用法：“$@”代表目标文件，“$^”代表所有的依赖文件，“$<”代表第 1 个依赖文件。例如

一个简单加法程序对应的 Makefile，代码如下：

```
cal.bin: add.c sub.c
    gcc $^ -o $@
```

3）预定义变量

Makefile 中有些变量是内部事先定义好的，它们都有默认值，用户也可以修改它们的值，见表 3-11。

表 3-11　预定义变量

变　量	说　明
AR	库文件打包程序，默认值为 ar
ARFLAGS	库选项，默认值为 rv
AS	汇编器，默认值为 as
ASFLAGS	汇编选项，默认值为空
CC	C 编译器，默认值为 cc
CFLAGS	C 编译器选项，默认值为空
CPP	C 预处理器，默认值为 $(CC)-E
CPPFLAGS	C 预编译选项，默认值为空
CXX	C++编译器，默认值为 g++
CXXFLAGS	C++编译选项，默认值为空
LDFLAGS	链接器选项，默认值为空
CURDIR	当前工作目录

4）导出变量

如果想让当前变量被其他 Makefile 使用，则可以使用关键字 export 将变量导出，代码如下：

```
#定义变量
OUT = out
#将变量导出
export OUT
```

3. 宏定义

Makefile 提供宏定义功能，语法格式为“-DNAME”，开发者可以在源文件中使用该宏。如果需要为宏指定值，则代码如下：

```
#定义宏 MQTT_PORT,值为 8765
CFLAGS += -DMQTT_PORT=8765
```

如果宏的值为字符串，则需要对引号进行转义。例如定义一个 IP 地址，代码如下：

```
#定义宏 MQTT_IP,值为字符串"192.168.1.100"
CFLAGS += -DMQTT_IP=\"192.168.1.100\"
```

4. 条件判断

Makefile 支持条件判断功能，例如 LiteOS 中的多数功能组件要通过判断配置文件来决定是否编译。例如一个简单的条件判断，代码如下：

```
#定义变量 NET
NET = BC85
#通过变量 NET 的值决定编译哪个网络驱动文件
ifneq ( $ (NET), BC95)
    gcc -c esp8266.c -o net.o
else
    gcc -c bc95.c -o net.o
endif
```

ifeq 的意义是如果两者相等，则开发者可使用 ifneq 表示。如果两者不相等，则需要多个判断，else 语句可以写作 else ifeq。

注意：在 ifeq 后有一个空格。

5. 导入文件

Makefile 提供了关键字 include，用于导入文件，语法格式为 include filename，文件名中可以包含路径和通配符。include 的作用和 C 语言中 include 的作用一样，将被导入的文件内容复制到当前位置。如果要导入当前目录下的 cfg1. mk、cfg2. mk、cfg3. mk，则代码如下：

```
#include 后追加 3 个文件名，用空格隔开
include cfg1.mk cfg2.mk cfg3.mk
#如果当前目录下只有这 3 个 mk 文件，则可用下面的语法
include *.mk
```

3.6.2 高级语法

1. 嵌套 Makefile

在实际项目中源文件大多按照其功能存放在不同的目录下，Makefile 也可以根据实际情况存放在各自的目录中，这样可以让 Makefile 更加简洁。例如在子目录 subdir 下有一个 Makefile，如果在主目录下的 Makefile 中依赖 subdir/Makefile，则代码如下：

```
objxxx:
#跳转到子目录并执行 make
cd subdir && make
#也可以这样写
objxxx:
make -C subdir
```

2. 模式规则

通常在 C 项目中包含多个源文件，这意味着需要生成多个 obj 文件。例如当前项目有

3 个文件，即 1.c、2.c、3.c，则代码如下：

```
target.bin:1.o 2.o 3.o
    gcc 1.o 2.o 3.o -o target.bin
1.o:1.c
    gcc -c 1.c -o 1.o
2.o:2.c
    gcc -c 2.c -o 2.o
3.o:3.c
    gcc -c 3.c -o 3.o
```

当项目中有成百上千个源文件时，开发者很难一个个列出目标和依赖。使用静态模式更容易定义“多目标规则”，其语法示例如下：

```
<target>:<target-pattern>:<depend-pattern>
```

target 可以省略，pattern 中必须包含字符“%”。针对上述案例，使用模式规则，代码如下：

```
%.o:%.c
    gcc $< -o $@

target.bin:1.o 2.o 3.o
    gcc 1.o 2.o 3.o -o target.bin
```

target-pattern 中的%.o 代表目标是以.o 结尾的文件集合，depend-pattern 中的%.c 表示取 target 中的%部分，然后追加后缀.c。

3. 函数

Makefile 支持少量函数，方便开发者快速处理一些任务，例如字符串替换、取当前路径、执行 Shell 指令等，详情见表 3-12。函数调用语法示例，代码如下：

```
$(function arg1, arg2,...)
```

function 是函数名字，arg1、arg2 是参数名，多个参数以逗号隔开，函数名和参数之间用空格分隔。函数调用以“$”开头，用小括号将函数名和参数包起来。

表 3-12　Makefile 常见函数

变　量	说　明
$(subst from,to,text)	函数名：subst 功能：字符串替换。将字符串 text 中的 from 替换为 to 返回值：替换后的字符串 示例：$(subst aa,bb,aabbcc) 示例结果：bbbbcc

续表

变　量	说　明
$(strip <string>)	函数名：strip 功能：去掉空格。将字符串 string 中的空格去掉 返回值：去掉空格后的字符串 示例：$(strip a b c) 示例结果：abc
$(filter <pattern…>, <text>)	函数名：filter 功能：过滤。将字符串 text 中不符合 pattern 的内容过滤掉 返回值：过滤后的字符串 示例：$(filter %.c, 1.c 2.c 3.h) 示例结果：1.c 2.c
$(dir <names…>)	函数名：dir 功能：取目录。从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠“/”之前的部分。如果没有反斜杠，则返回“./” 返回值：目录序列 示例：$(dir src/demo.c example.c) 示例结果：src/ ./
$(notdir <names…>)	函数名：notdir 功能：取文件名。从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠“/”之后的部分 返回值：文件名序列 示例：$(notdir src/demo.c example.c) 示例结果：demo.c example.c
$(basename <names…>)	函数名：basename 功能：取前缀。从文件名序列<names>中取出文件名的前缀 返回值：文件名前缀序列 示例：$(basename src/demo.c example.c) 示例结果：src/demo example
$(addsuffix <suffix>, <names…>)	函数名：addsuffix 功能：增加后缀。给<names>中的文件添加指定的后缀 返回值：修改后的文件名序列 示例：$(addsuffix .c, src/demo example) 示例结果：src/demo.c example.c
$(addprefix <prefix>, <names…>)	函数名：addprefix 功能：增加前缀。给<names>中的文件添加指定的前缀 返回值：修改后的文件名序列 示例：$(addprefix src/, demo.c example.c) 示例结果：src/demo.c src/example.c
$(shell shell-commands)	函数名：shell 功能：执行 Shell 指令，参数就是 Shell 命令 返回值：Shell 命令的返回值 示例：$(shell cat 1.c) 示例结果：1.c 文件中的内容

3.6.3　实战案例：简单计算器

1. 案例描述

制作一个简单计算器：定义 4 个函数 add_int()、add_float()、sub_int()、sub_float()，将 4 个函数分别定义在不同的文件中，最后在 cal.c 文件中通过 main()函数调用这 4 个函数。按照功能将文件分开存储，并定义对应的 Makefile 文件，最终的目录结构如图 3-23 所示。

```
|sample-make
|----add
     |----add_int
          |----add_int.c
          |----add_int.h
     |----add_float
          |----add_float.c
          |----add_float.h
     |----Makefile
|----sub
     |----sub_int
          |----sub_int.c
          |----sub_int.h
     |----sub_float
          |----sub_float.c
          |----sub_float.h
     |----Makefile
|----cal
     |----cal.c
     |----cal.h
     |----Makefile
|----Makefile
```

图 3-23　目录结构

注意：在本案例中只有简单的数学运算和输出语句，因此使用 GCC 编译，并直接在 PC 运行编译生成的可执行文件。

2. 操作流程

1）C 源码

编辑 4 个源代码文件，分别实现 int 加减法、float 加减法，代码如下：

```
//第 3 章/add/add_int/add_int.c
int add_int(int a, int b){
    return a + b;
}


//第 3 章/add/add_float/add_float.c
float add_float(float a, float b){
    return a + b;
}


//第 3 章/sub/sub_int/sub_int.c
int sub_int(int a, int b){
    return a - b;
}


//第 3 章/sub/sub_float/sub_float.c
float sub_float(float a, float b){
    return a - b;
}


//第 3 章/cal/cal.c
int main(){
```

```
    printf("test int add %d + %d = %d\n", 3, 2, add_int(3,2));
    printf("test float add %f + %f = %f\n", 3.5, 2.5, add_float(3.5,2.5));
    printf("test int sub %d + %d = %d\n", 3, 2, sub_int(3,2));
    printf("test float sub %f + %f = %f\n", 3.5, 2.5, sub_float(3.5,2.5));

    return 0;
}
```

2）顶层 Makefile

顶层目录的主要工作是将所有的 o 文件链接生成 bin 文件（Windows 系统下生成 exe 文件）。设置将所有编译结果输出到 out 目录下，最终生成的目标文件为 cal.bin。将当前目录及输出目录导出，以供子目录下的 Makefile 使用。具体的代码如下：

```
#第 3 章/Makefile
#获取当前工作目录
SAMPLETOPDIR = $(CURDIR)
#导出变量，以供其他 Makefile 使用
export SAMPLETOPDIR

#目标文件
TARGET = $(OUT)/cal.bin

#输出目录
OUT = out
export OUT

#伪目标
.PHONY: $(TARGET) clean

#第 1 个目标，将所有的.o 文件编译生成最终的.bin 文件
$(TARGET): $(OUT)
    $(CC) $(wildcard out/obj/*.o) -o $(TARGET)

#其他目标，out 依赖其他目标，需要进入子目录执行 Makefile
$(OUT):
    cd add && make -w
    cd sub && make -w
    cd cal && make -w

#清除 out 中的内容
clean:
    rm -rf out
```

3）子目录 Makefile

所有的子目录功能都一样，将子目录下的 c 文件编译生成 o 文件。变量 LOCAL_SRCS 为该目录下的所有 c 文件集合，变量 LOCAL_INC 为该目录下的头文件目录集合，变量 OBJECTS 为所有目标 o 文件集合。以 add 子目录为例，代码如下：

```
#第 3 章/add/Makefile
#源文件路径
LOCAL_SRCS += \
    $(SAMPLETOPDIR)/add/add_int/add_int.c \
    $(SAMPLETOPDIR)/add/add_float/add_float.c

#头文件路径
LOCAL_INC += \
    -I $(SAMPLETOPDIR)/add/add_int \
    -I $(SAMPLETOPDIR)/add/add_float
CFLAGS += $(LOCAL_INC)

#目标文件,这里使用模式替换操作,取文件名操作,增加前缀操作
#最终得到 OBJECTS += ./out/obj/*.o
OBJECTS += $(addprefix $(SAMPLETOPDIR)/$(OUT)/obj/,$(notdir $(LOCAL_SRCS:.c=.o)))
#目标输出目录
OBJOUT = $(SAMPLETOPDIR)/$(OUT)/obj

#设置 c 文件的搜索目录
vpath %.c $(sort $(dir $(LOCAL_SRCS)))

#此文件的第 1 个目标
all: $(OBJECTS)

#模式规则,目标为所有 o 文件,依赖为对应的 c 文件.使用 gcc 生成.o 文件
$(OBJOUT)/%.o: %.c $(SAMPLETOPDIR)/$(OUT)/obj
    gcc $(CFLAGS) -c $< -o $@

#创建目录 out/obj
$(SAMPLETOPDIR)/$(OUT)/obj:
    mkdir -p $@
```

3. 运行结果

在项目的根目录下执行 make 指令进行编译，由于 TARGET 目标依赖 OUT，因此 Makefile 会跳转到 OUT 目标，接着执行 OUT 下的命令切换到子目录执行 make。最终的运行结果如图 3-24 所示。

```
weijiedeMacBook-Pro:make-sample weijie$ make
cd add && make -w
make[1]: Entering directory `/Users/weijie/data/code/liteos/make-sample/add'
mkdir -p /Users/weijie/data/code/liteos/make-sample/out/obj
gcc -c /Users/weijie/data/code/liteos/make-sample/add/add_int/add_int.c -o /Users/weijie/data/code/liteos/make-
sample/out/obj/add_int.o
gcc -c /Users/weijie/data/code/liteos/make-sample/add/add_float/add_float.c -o /Users/weijie/data/code/liteos/m
ake-sample/out/obj/add_float.o
make[1]: Leaving directory `/Users/weijie/data/code/liteos/make-sample/add'
cd sub && make -w                                          进入sub目录，并编译
make[1]: Entering directory `/Users/weijie/data/code/liteos/make-sample/sub'
gcc -c /Users/weijie/data/code/liteos/make-sample/sub/sub_int/sub_int.c -o /Users/weijie/data/code/liteos/make-
sample/out/obj/sub_int.o
gcc -c /Users/weijie/data/code/liteos/make-sample/sub/sub_float/sub_float.c -o /Users/weijie/data/code/liteos/m
ake-sample/out/obj/sub_float.o
make[1]: Leaving directory `/Users/weijie/data/code/liteos/make-sample/sub'
cd cal && make -w
make[1]: Entering directory `/Users/weijie/data/code/liteos/make-sample/cal'
gcc -c -I /Users/weijie/data/code/liteos/make-sample/cal -I /Users/weijie/data/code/liteos/make-sample/sub/sub_
int -I /Users/weijie/data/code/liteos/make-sample/sub/sub_float -I /Users/weijie/data/code/liteos/make-sample/a
dd/add_int -I /Users/weijie/data/code/liteos/make-sample/add/add_float cal.c -o /Users/weijie/data/code/liteos/
make-sample/out/obj/cal.o
make[1]: Leaving directory `/Users/weijie/data/code/liteos/make-sample/cal'
cc out/obj/add_float.o out/obj/add_int.o out/obj/cal.o out/obj/sub_float.o out/obj/sub_int.o -o out/cal.bin
weijiedeMacBook-Pro:make-sample weijie$ ./out/cal.bin        执行bin文件，测试编译结果
test int   add 3 + 2 = 5
test float add 3.500000 + 2.500000 = 6.000000
test int   sub 3 + 2 = 1
test float sub 3.500000 + 2.500000 = 1.000000
```

图 3-24　编译及运行结果

3.7　本章小结

本章从启动代码开始逐步认识 LiteOS，任务、中断、内存是操作系统的基础模块，多数项目需要用到这 3 个基础模块。借助异常管理功能可以迅速定位错误信息，一个合格的开发者应该习惯用 Backtrace 信息定位错误。3.6 节介绍了 Makefile 工具，可帮助开发者更好地理解大型项目的代码结构。