# 基于小熊派IoT开发套件的LiteOS内核实战 - 内存管理

# 1. LiteOS内核的内存管理

## 1.1. 内存管理

在系统运行的过程中，一些内存空间大小是不确定的，比如一些数据缓冲区，所以系统需要提供内存空间的管理能力，用户可以在使用的时候申请需要的内存空间，使用完毕释放该空间，以便再次利用。

Huawei LiteOS 的内存管理模块通过对内存的申请/释放操作，来管理用户和OS对内存的使用，使内存的利用率和使用效率达到最优，同时最大限度地解决系统的内存碎片问题。

## 1.2. 动态内存管理

动态内存管理，即在内存资源充足的情况下，从系统配置的一块比较大的连续内存（内存池），根据用户需求，分配任意大小的内存块。当用户不需要该内存块时，又可以释放回系统供下一次使用。

与静态内存相比，动态内存管理的好处是按需分配，缺点是内存池中容易出现碎片。

LiteOS动态内存支持 DLINK 和 BEST LITTLE 两种标准算法。

### 1.2.1. DLINK 动态内存管理算法

DLINK动态内存管理结构如下图所示：

第一部分

堆内存（也称内存池）的起始地址及堆区域总大小。

第二部分

本身是一个数组，每个元素是一个双向链表，所有free节点的控制头都会被分类挂在这个数组的双向链表中。

第三部分

占用内存池极大部分的空间，是用于存放各节点的实际区域。

### 1.2.2. BEST LITTLE 算法（重点）

LiteOS 的动态内存分配支持最佳适配算法，即 BEST LITTLE，每次分配时选择内存池中最小最适合的内存块进行分配。

LiteOS 动态内存管理在最佳适配算法的基础上加入了 SLAB 机制，用于分配固定大小的内存块，进而减小产生内存碎片的可能性。

LiteOS 内存管理中的 SLAB 机制支持可配置的 SLAB CLASS 数目及每个 CLASS 的最大空间。

现以 SLAB CLASS 数目为 4，每个 CLASS 的最大空间为 512 字节为例说明 SLAB 机制：

在内存池\*\*有 4 个 SLAB CLASS，每个 SLAB CLASS 的总共可分配大小为 512 字节，第一个 SLAB CLASS 被分为 32 个16 字节的 SLAB 块，第二个 SLAB CLASS 被分为 16 个 32字节的 SLAB 块，第三个 SLAB CLASS 被分为 8 个 64 字节的 SLAB 块，第四个 SLAB CLASS 被分为 4 个 128 字节的 SLAB 块。这 4 个 SLAB CLASS 是从内存池中按照最佳适配算法分配出来的。

初始化内存管理时，首先初始化内存池，然后在初始化后的内存池中按照最佳适配算法申请 4 个 SLAB CLASS，再逐个按照 SLAB 内存管理机制初始化 4 个 SLAB CLASS。

每次申请内存时，先在满足申请大小的最佳 SLAB CLASS 中申请，（比如用户申请 20 字节内存，就在 SLAB 块大小为 32 字节的 SLAB CLASS 中申请），如果申请成功，就将 SLAB 内存块整块返回给用户，释放时整块回收。如果满足条件的 SLAB CLASS 中已无可以分配的内存块，则继续向内存池按照最佳适配算法申请。需要注意的是，如果当前的 SLAB CLASS 中无可用 SLAB 块了，则直接向内存池申请，而不会继续向有着更大 SLAB 块空间的 SLAB CLASS 申请。

释放内存时，先检查释放的内存块是否属于 SLAB CLASS，如果是 SLAB CLASS 的内存块，则还回对应的 SLAB CLASS 中，否则还回内存池中。

### 1.2.3. 两种动态内存管理方法的选择

LiteOS动态内存管理的方法使用宏定义的方法使能，在用户工程目录下的OS\_CONFIG中的target\_config.h文件中配置。

在该文件中，找到下面这两项宏定义，置为 YES 则表示使能：

开启BEST LITTLE 算法

#define LOSCFG\_MEMORY\_BESTFIT YES

开启SLAB机制

#define LOSCFG\_KERNEL\_MEM\_SLAB YES

## 1.3. 动态内存管理的应用场景

内存管理的主要工作是动态的划分并管理用户分配好的内存区间。

动态内存管理主要是在用户需要使用大小不等的内存块的场景中使用。当用户需要分配内存时，可以通过操作系统的动态内存申请函数索取指定大小内存块，一旦使用完毕，通过动态内存释放函数归还所占用内存，使之可以重复使用。

# 2. 动态内存管理API

Huawei LiteOS 系统中的内存管理模块管理系统的内存资源，主要提供内存的初始化、分配以及释放功能。

Huawei LiteOS 系统中提供的内存管理 API 都是以 LOS 开头，但是这些 API 使用起来比较复杂，所以本文中我们使用 Huawei IoT Link SDK 提供的统一API接口进行实验，这些接口底层已经使用 LiteOS 提供的API实现，对用户而言更为简洁，API列表如下：

osal的api接口声明在中，使用相关的接口需要包含该头文件，关于函数的详细参数请参考该头文件的声明。

相关的接口定义在osal.c中，基于LiteOS的接口实现在 liteos\_imp.c文件中：

接口名 功能描述

osal\_malloc 按字节申请分配动态内存空间

osal\_free 释放已经分配的动态内存空间

osal\_zalloc 按字节申请分配动态内存空间，分配成功则初始化这块内存所有值为0

osal\_realloc 重新申请分配动态内存空间

osal\_calloc 申请分配num个长度为size的动态内存空间

无论选择使用哪种动态内存管理算法，都使用该API。

## 2.1. osal\_malloc

osal\_malloc接口用于按字节申请分配动态内存空间，其接口原型如下：

void \*osal\_malloc(size\_t size)

{

void \*ret = NULL;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->malloc))

{

ret = s\_os\_cb->ops->malloc(size);

}

return ret;

}

该接口的参数说明如下表：

参数 描述

size 申请分配的内存大小，单位Byte

返回值 分配成功 - 返回内存块指针

分配失败 - 返回NULL

## 2.2. osal\_free

osal\_free接口用于释放已经分配的动态内存空间，其接口原型如下：

void osal\_free(void \*addr)

{

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->free))

{

s\_os\_cb->ops->free(addr);

}

return;

}

内存块free之后，记得使内存块指针为NULL，否则会成为野指针！

该接口的参数说明如下表：

参数 描述

addr 动态分配内存空间的指针

返回值 无返回值

## 2.3. osal\_zalloc

osal\_zalloc接口用于按字节申请分配动态内存空间，分配成功则初始化这块内存所有值为0，其接口原型如下：

void \*osal\_zalloc(size\_t size)

{

void \*ret = NULL;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->malloc))

{

ret = s\_os\_cb->ops->malloc(size);

if(NULL != ret)

{

memset(ret,0,size);

}

}

return ret;

}

该接口的参数说明如下表：

参数 描述

size 申请分配的内存大小，单位Byte

返回值 分配成功 - 返回内存块指针

分配失败 - 返回NULL

## 2.4. osal\_realloc

osal\_realloc接口用于重新申请分配动态内存空间，其接口原型如下：

void \*osal\_realloc(void \*ptr,size\_t newsize)

{

void \*ret = NULL;

if((NULL != s\_os\_cb) &&(NULL != s\_os\_cb->ops) &&(NULL != s\_os\_cb->ops->realloc))

{

ret = s\_os\_cb->ops->realloc(ptr,newsize);

}

return ret;

}

该接口的参数说明如下表：

参数 描述

ptr 已经分配了内存空间的指针

newsize 申请分配的新的内存大小，单位Byte

返回值 分配成功 - 返回内存块指针

分配失败 - 返回NULL

## 2.5. osal\_calloc

osal\_calloc接口用于申请分配num个长度为size的动态内存空间，其接口原型如下：

void \*osal\_calloc(size\_t n, size\_t size)

{

void \*p = osal\_malloc(n \* size);

if(NULL != p)

{

memset(p, 0, n \* size);

}

return p;

}

该接口的参数说明如下表：

参数 描述

n 申请分配内存块的数目

size 申请分配的每个内存块的内存大小，单位Byte

返回值 分配成功 - 返回内存块指针

分配失败 - 返回NULL

# 3. 动手实验 —— 测试动态内存分配的最大字节

实验内容

本实验中将创建一个任务，从最小字节开始，不停的申请分配内存，释放分配的内存，直到申请失败，串口终端中观察可以申请到的最大字节。

实验代码

首先打开上一篇使用的 HelloWorld 工程，基于此工程进行实验。

在Demo文件夹右击，新建文件夹osal\_kernel\_demo用于存放内核的实验文件（如果已有请忽略这一步）。

接下来在此文件夹中新建一个实验文件 osal\_mem\_demo.c，开始编写代码：

/\* 使用osal接口需要包含该头文件 \*/

#include <osal.h>

/\* 任务入口函数 \*/

static int mem\_access\_task\_entry()

{

uint32\_t i = 0; //循环变量

size\_t mem\_size; //申请的内存块大小

uint8\_t\* mem\_ptr = NULL; //内存块指针

while (1)

{

/\* 每次循环将申请内存的大小扩大一倍 \*/

mem\_size = 1 << i++;

/\* 尝试申请分配内存 \*/

mem\_ptr = osal\_malloc(mem\_size);

/\* 判断是否申请成功 \*/

if(mem\_ptr != NULL)

{

/\* 申请成功，打印信息 \*/

printf("access %d bytes memory success!\r\n", mem\_size);

/\* 释放申请的内存，便于下次申请 \*/

osal\_free(mem\_ptr);

/\* 将内存块指针置为NULL，避免称为野指针 \*/

mem\_ptr = NULL;

printf("free memory success!\r\n");

}

else

{

/\* 申请失败，打印信息，任务结束 \*/

printf("access %d bytes memory failed!\r\n", mem\_size);

return 0;

}

}

}

/\* 标准demo启动函数，函数名不要修改，否则会影响下一步实验 \*/

int standard\_app\_demo\_main()

{

/\* 创建任务，任务优先级为11，shell任务的优先级为10 \*/

osal\_task\_create("mem\_access\_task",mem\_access\_task\_entry,NULL,0x400,NULL,11);

return 0;

}

编写完成之后，要将我们编写的 osal\_mem\_demo.c文件添加到makefile中，加入整个工程的编译：

这里有个较为简单的方法，直接修改Demo文件夹下的user\_demo.mk配置文件，添加如下代码：

#example for osal\_mem\_demo

ifeq ($(CONFIG\_USER\_DEMO), "osal\_mem\_demo")

user\_demo\_src = ${wildcard $(TOP\_DIR)/targets/STM32L431\_BearPi/Demos/osal\_kernel\_demo/osal\_mem\_demo.c}

endif

添加位置如图：

这段代码的意思是：

如果 CONFIG\_USER\_DEMO 宏定义的值是osal\_mem\_demo，则将osal\_mem\_demo.c文件加入到makefile中进行编译。

那么，如何配置 CONFIG\_USER\_DEMO 宏定义呢？在工程根目录下的.sdkconfig文件中的末尾即可配置：

因为我们修改了mk配置文件，所以点击重新编译按钮进行编译，编译完成后点击下载按钮烧录程序。

实验现象

程序烧录之后，即可看到程序已经开始运行，在串口终端中可看到实验的输出内容：

\*\*Link\*\*main:V1.2.1 AT 11:30:59 ON Nov 28 2019

WELCOME TO IOT\_LINK SHELL

LiteOS:/>access 1 bytes memory success!

free memory success!

access 2 bytes memory success!

free memory success!

access 4 bytes memory success!

free memory success!

access 8 bytes memory success!

free memory success!

access 16 bytes memory success!

free memory success!

access 32 bytes memory success!

free memory success!

access 64 bytes memory success!

free memory success!

access 128 bytes memory success!

free memory success!

access 256 bytes memory success!

free memory success!

access 512 bytes memory success!

free memory success!

access 1024 bytes memory success!

free memory success!

access 2048 bytes memory success!

free memory success!

access 4096 bytes memory success!

free memory success!

access 8192 bytes memory success!

free memory success!

access 16384 bytes memory success!

free memory success!

access 32768 bytes memory failed!

可以看到，系统启动后，首先打印版本号,串口shell的优先级为10，最先打印shell信息，接下来内存申请任务创建开始执行，在该芯片上最大能申请的空间为 16384 字节。