

# 有移植过free rtos吗

[STM32移植FreeRTOS](#)哔哩哔哩bilibili

有移植过free rtos2022的LTS版本

对于不同的这种32位的arm

其实整体的步骤全都是一样的啊

- 第一步呢我们从官网去下载源码
- 然后呢去裁剪源码啊。
  - 把那些不需要用到的模块呢直接删掉
  - 比如说我当前不需要用到文件系统
  - 我就把文件系统给删掉
- 然后呢我们就需要去配那个CONFIG.H
  - 来去解决它的一些依赖的缺失好
- 最后就是把system core clock这个问题给解决掉好
  - 那在这个system core clock里面会有大量的xxx\_handler的回调错误啊
  - 我们需要把这个回调错误呢给解决掉好
  - 那这些回调错误全都解决完了啊
- 然后最后把这个硬件的system tick
  - 跟这个delay建立起联系
- 那整个移植就差不多搞好了
- 那怎么来确定你的操作系统移植成功了呢

点个灯啊

那如果这个灯周期性的亮灭了

就说明整个这个流程呢你都搞定了

# FreeRTOS创建任务

## free rtos动态和静态创建任务区别

在free rtos中

任务的创建有两种方式

动态创建和静态创建



这两种方式有什么区别呢

我们一起来看下吧

首先是动态创建

它是我们开发中比较常用的创建任务方式

在上期移植视频中

我们最后就是使用的动态创建任务方式

来测试一下移植是否成功

```
main.c
5 TaskHandle_t myTaskHandler;
6
7 void myTask( void * arg)
8 {
9     while(1)
10    {
11        GPIO_ResetBits(GPIOC, GPIO_Pin_13);
12        vTaskDelay(500);
13        GPIO_SetBits(GPIOC, GPIO_Pin_13);
14        vTaskDelay(500);
15    }
16 }
17 }
18
19 int main(void)
20 {
21     GPIO_InitTypeDef GPIO_Initstruct;
22     RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC, ENABLE);
23     GPIO_Initstruct.GPIO_Mode = GPIO_Mode_Out_PP;
24     GPIO_Initstruct.GPIO_Pin = GPIO_Pin_13;
25     GPIO_Initstruct.GPIO_Speed = GPIO_Speed_50MHz;
26     GPIO_Init(GPIOC, &GPIO_Initstruct);
27
28     GPIO_ResetBits(GPIOC, GPIO_Pin_13);
29
30     xTaskCreate(myTask, "myTask", 512, NULL, 2, &myTaskHandler);
31
32     vTaskStartScheduler();
33
34 }
```

动态创建就是任务创建时  
操作系统来动态分配任务所需要的内存空间  
而静态创建任务时  
由我们自己去指定空间大小

**任务空间**  
**动态创建：由系统分配**  
**静态创建：由自己定义空间大小**

这是两种创建任务方式的函数  
动态创建和静态创建任务函数  
前面几个参数都是相同的

# 动态创建

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint16_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pxCreatedTask ) /*
```

# 静态创建

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint32_t ulStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    StackType_t * const puxStackBuffer,
    StaticTask_t * const pxTaskBuffer ) /*
```

- 都需要一个任务函数
- 任务名字
- 任务堆栈大小
- 任务参数和任务优先级

不同的是后面的参数

动态创建

这里是需要一个任务句柄

而静态创建

这里需要一个堆栈空间和一个任务控制块

# 动态创建

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint16_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pxCreatedTask ) /*
```

# 静态创建

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint32_t ulStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    StackType_t * const puxStackBuffer,
    StaticTask_t * const pxTaskBuffer ) /*
```

# 任务空间 就需要我们传入自己设置的内存空间

- 可以自己申请空间
- 也可以直接声明一个数组

## 任务控制块

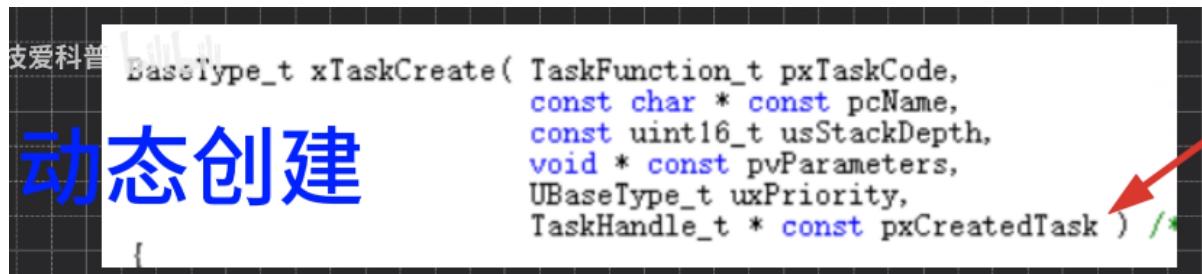
- 就相当于任务的身份证
- 里面存有任务的所有信息

那为什么动态创建没有任务控制块传入呢

其实我们这里传入的任务句柄

它是一个void类型的指针

系统会自动帮我们指向一个任务控制块



总的来说

动态创建任务简单方便

不需要自己分配内存

而静态创建需要自己分配内存

但是不易产生内存碎片

**动态创建：动态分配内存，简单，方便，灵活**  
**静态创建：自己分配内存，不易产生内存碎片**

## 静态创建任务：

- 1、需要自己指定堆栈空间
- 2、需要自己定义任务控制块

优点：不易产生内存碎片

缺点：任务如果删除，无法回收内存

## 动态创建任务：

无需指定堆栈空间，只需要传入一个任务句柄，  
系统会自动分配任务控制块

优点：内存分配灵活，节省空间

缺点：容易产生内存碎片

我们现在来总结一下静态创建任务

一需要自己指定堆栈空间

二需要自己定义任务控制块

优点不易产生内存碎片

缺点任务如果删除

无法回收内存

动态创建任务

无需指定堆栈空间

只需要传入一个任务句柄

系统会自动分配任务控制块

优点内存分配灵活

节省空间

缺点容易产生内存碎片

# FreeRTOS编程风格

## 从数据类型变量名

函数名和宏定义几个方面大概来说一下

首先我们来看一下数据类型

不知道大家是否记得在我们移植时

port文件夹里有一个点A键

就是这个port macro点A键

这里面定义了free rtos用到的相关数据类型

我们打开看一下

这是我截取的一部分

我们来举例说明一下

free rtos将用到的数据类型都重新起了一个名字

这里我们要注意在cortex m3内核中

- short类型是16位
- long类型是32位

## 数据类型

```
/*
 * Type definitions.
 */
#define portCHAR char
#define portFLOAT float
#define portDOUBLE double
#define portLONG long
#define portSHORT short
#define portSTACK_TYPE uint32_t
#define portBASE_TYPE long

typedef portSTACK_TYPE StackType_t;
typedef long BaseType_t;
typedef unsigned long UBaseType_t;

#ifndef configUSE_16_BIT_TICKS
    typedef uint16_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffff
#else
    typedef uint32_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffffffffUL
    /* 32-bit tick type on a 32-bit architecture, so reads of the tick count do
       not need to be guarded with a critical section. */
    #define portTICK_TYPE_IS_ATOMIC 1
#endif
/*-----*/
```

在free rtos中有两个最基本的数据类型

base type\_t和tick type\_t

这两个数据类型的存在是基于执行效率考虑的

如果在32位架构单片机中

就把它们设置成32位

如果在16位架构中

就把它们配置成16位

```

/*
 * Type definitions. */
#define portCHAR char
#define portFLOAT float
#define portDOUBLE double
#define portLONG long
#define portSHORT short
#define portSTACK_TYPE uint32_t
#define portBASE_TYPE long

typedef portSTACK_TYPE StackType_t;
typedef long BaseType_t;
typedef unsigned long UBaseType_t;

#ifndef configUSE_16_BIT TICKS == 1
    typedef uint16_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffff
#else
    typedef uint32_t TickType_t;
    #define portMAX_DELAY ( TickType_t ) 0xffffffffUL
    /* 32-bit tick type on a 32-bit architecture, so reads of the tick count do
       not need to be guarded with a critical section. */
    #define portTICK_TYPE_IS_ATOMIC
#endif

```

Base type\_t  
Tick type\_t

这两个数据类型的存在是基于执行效率考虑的，  
如果在32位架构单片机中，就把他们设置成32位。  
如果在16位架构中，就把它们配置成16位

## 变量和函数命名方式

在free rtos中

变量名都是有一个小写前缀

这些前缀的意思就是我总结的这张表中的内容

前缀	含义
c	char
s	short
l	long
x	FreeRTOS自定义数据类型
u	unsigned
p	指针
uc	unsigned char
pc	char指针

以看到前缀是pc, p就是指针，c是char型

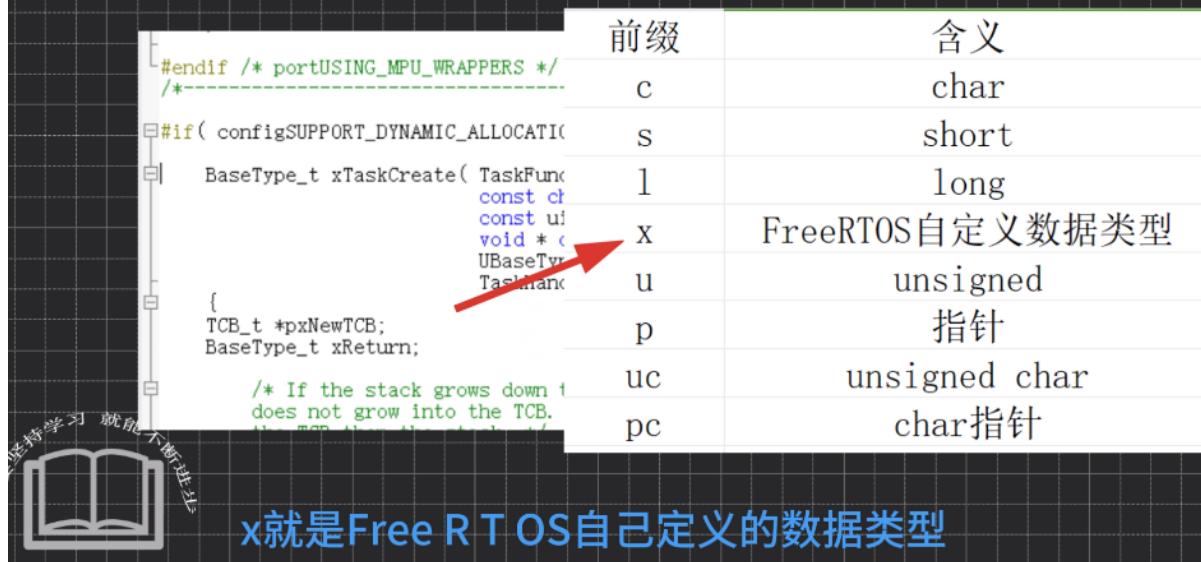
可以看到前缀是PCP就是指针C是char型  
所以这里的PC就是X2型指针变量  
其他变量也是这种命名方式  
这样做的目的是  
让人在函数中看到使用的变量时  
就可以不用找到定义  
而直接知道它是什么类型的

函数的命名方式也有类似的前缀  
只不过后面还多一个文件名  
小写前缀的意思就是返回值的类型  
X就是free rtos自己定义的数据类型

## 2、变量名和函数名

前缀	含义
C	char
S	short
L	long
X	FreeRTOS自定义数据类型
U	unsigned
P	指针
UC	unsigned char
PC	char指针

x就是Free RTOS自己定义的数据类型



后面的task就是这个函数

在task文件中定义在之后的就是函数的功能表述了

没有返回值的函数,前缀就是V

```
#if ( INCLUDE_vTaskDelete == 1 )  
    void vTaskDelete( TaskHandle_t xTaskToDelete )  
    {  
        TCB_t *pxTCB;  
        taskENTER_CRITICAL();  
        /* If null is passed in here then it is the c
```

还有函数前用static修饰的函数

也就是私有的函数只能在当前文件使用 它的前缀就是pr v

也就是private私有的缩写

```
static void prvInitialiseNewTask(  
    TaskFunction_t pxTaskCode,  
    const char * const pcName,  
    const uint32_t ulStackDepth,  
    void * const pvParameters,  
    UBaseType_t uxPriority,  
    TaskHandle_t * const pxCreatedTask,  
    TCB_t *pxNewTCB,  
    const MemoryRegion_t * const xRegions ) /*1  
{  
    StackType_t *pxTopOfStack;  
    UBaseType_t x;
```

并且也没有返回值类型和文件名

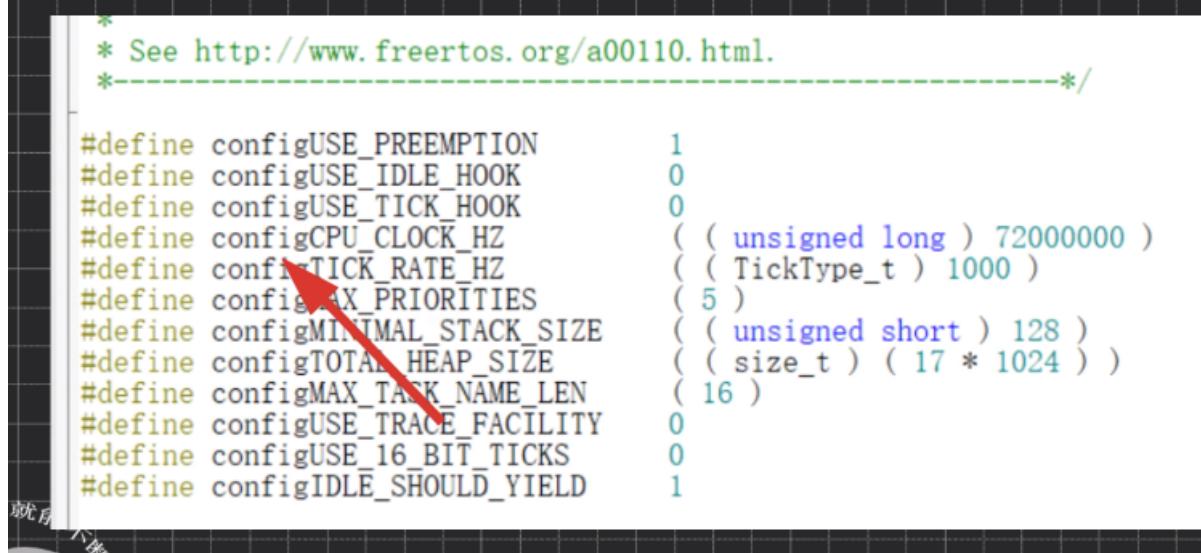
## 最后就是宏定义了

每个宏定义之前也都有相应的文件名前缀

这些都是在CONFIG文件内定义的

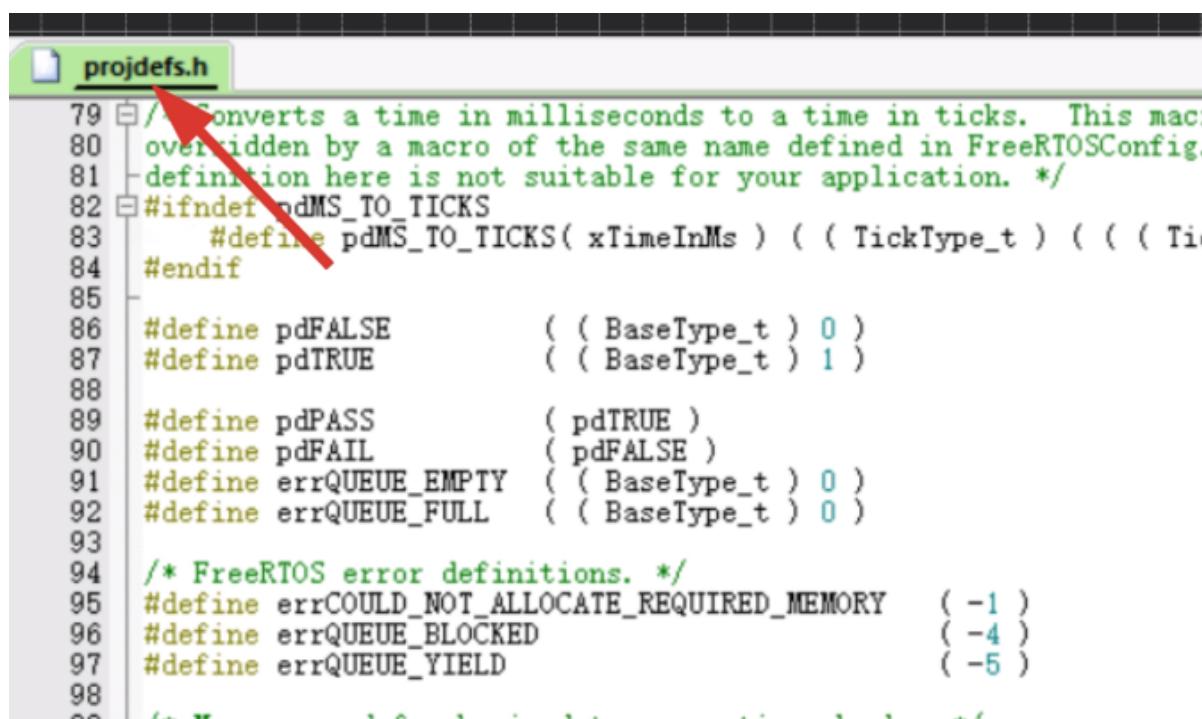
所以它们的前缀都是CONFIG

# 3. 宏定义



```
* See http://www.freertos.org/a00110.html.  
*-----*/  
  
#define configUSE_PREEMPTION      1  
#define configUSE_IDLE_HOOK       0  
#define configUSE_TICK_HOOK        0  
#define configCPU_CLOCK_HZ        ( ( unsigned long ) 72000000 )  
#define configTICK_RATE_HZ         ( ( TickType_t ) 1000 )  
#define configMAX_PRIORITIES      ( 5 )  
#define configMINIMAL_STACK_SIZE   ( ( unsigned short ) 128 )  
#define configTOTAL_HEAP_SIZE      ( ( size_t ) ( 17 * 1024 ) )  
#define configMAX_TASK_NAME_LEN    ( 16 )  
#define configUSE_TRACE_FACILITY   0  
#define configUSE_16_BIT_TICKS      0  
#define configIDLE_SHOULD_YIELD     1
```

我们还有常见到过pd和ERR前缀的宏  
他们都在这个文件内定义



```
projdefs.h  
79 /* Converts a time in milliseconds to a time in ticks. This macro  
80  overridden by a macro of the same name defined in FreeRTOSConfig.h  
81  definition here is not suitable for your application. */  
82 #ifndef pdMS_TO_TICKS  
83     #define pdMS_TO_TICKS( xTimeInMs ) ( ( TickType_t ) ( ( ( Ti  
84 #endif  
85  
86 #define pdFALSE          ( ( BaseType_t ) 0 )  
87 #define pdTRUE           ( ( BaseType_t ) 1 )  
88  
89 #define pdPASS            ( pdTRUE )  
90 #define pdFAIL             ( pdFALSE )  
91 #define errQUEUE_EMPTY     ( ( BaseType_t ) 0 )  
92 #define errQUEUE_FULL      ( ( BaseType_t ) 0 )  
93  
94 /* FreeRTOS error definitions. */  
95 #define errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY ( -1 )  
96 #define errQUEUE_BLOCKED      ( -4 )  
97 #define errQUEUE_YIELD        ( -5 )  
98  
99 /* More definitions can be added here if required. */
```

这是我总结的常见的红前缀和对应代表的文件  
有需要的小伙伴可以截图保存一下

前缀	文件
port	portable.h或portmacro.h
task	task.h
config	FreeRTOSConfig.h
pd	projdefs.h
err	projdefs.h

这是我总结的常见的宏前缀和对应代表的文件

了解了这些命名规则

我们在阅读free rtos源码时就能提高一些效率

不至于看到源码中的名字感到很奇怪

## FreeRTOS内存管理简单介绍

简单介绍下free rtos的内存管理

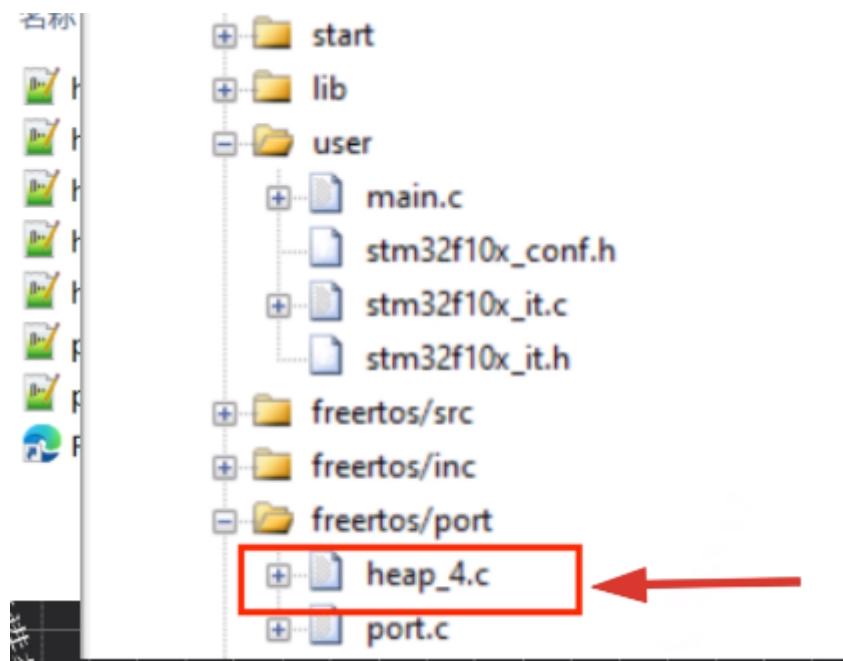
我们在移植free rtos时

在port文件夹里复制了这些heap文件

这些文件都是内存管理文件



可是为什么我们在工程里只用到了hip4.c呢



我们先来介绍一下为什么需要内存管理

在我们创建任务时有两种方式

动态创建和静态创建

在动态创建中

只需要提供一个任务堆栈大小和一个句柄指针

不需要我们提供堆栈空间和任务控制块

这是因为动态创建任务时

free rtos会自动的帮我们动态分配这些资源

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint16_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pxCreatedTask ) /*
```

# 动态

```
TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode,
```

```
    const char * const pcName,
    const uint32_t ulStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    StackType_t * const puxStackBuffer,
    StaticTask_t * const pTaskBuffer )
```

# 静态

这是因为动态创建任务时，

Free RT OS会自动的帮我们动态分配这些资源

而自动的动态分配空间就是内存管理

内存管理可以实现内存用到时分配

不使用时释放

在之后使用队列，信号量等也都涉及内存分配

使用内存的动态管理功能

就不再需要自己提前规划各类对象和堆栈

简化了我们的程序设计

分配内存的方法就在这些文件中

heap1到5

这几个文件是不同的内存分配方法

这张表就是它们的区别

文件	优点	缺点
heap1	分配简单，时间确定	只能申请，不能释放
heap2	可以动态分配	不能合并相邻空闲内存，产生内存碎片
heap3	直接调用C库的malloc和free	速度慢，时间不定
heap4	动态分配，可以合并相邻空闲内存	时间不定
heap5	在heap4基础之上能管理多个非连续内存	时间不定

# heap1

heap1是指实现了malloc功能

没有实现free功能

所以它是只能分配内存

不能释放内存



文件	优点	缺点
heap1	分配简单，时间确定	只能申请，不能释放
heap2	可以动态分配	不能合并相邻空闲内存
heap3	直接调用C库的malloc和free	速度慢，时间不定
heap4	动态分配，可以合并相邻空闲内存	时间不定

它的实现原理就是先定义一个大数组

然后将合适的空间静态的分配给每一个任务

如果某个任务或队列信号量等删除了

那么这个空间也没办法释放

所以它适用于只创建不删除的系统中



文件	优点	缺点
heap1	分配简单，时间确定	只能申请，不能释放
heap2	可以动态分配	不能合并相邻空闲内存，产生内存碎片
heap3	直接调用C库的malloc和free	速度慢，时间不定
heap4	动态分配，可以合并相邻空闲内存	时间不定
heap5	在heap4基础之上能管理多个	时间不定



所以它适用于只创建，不删除的系统中

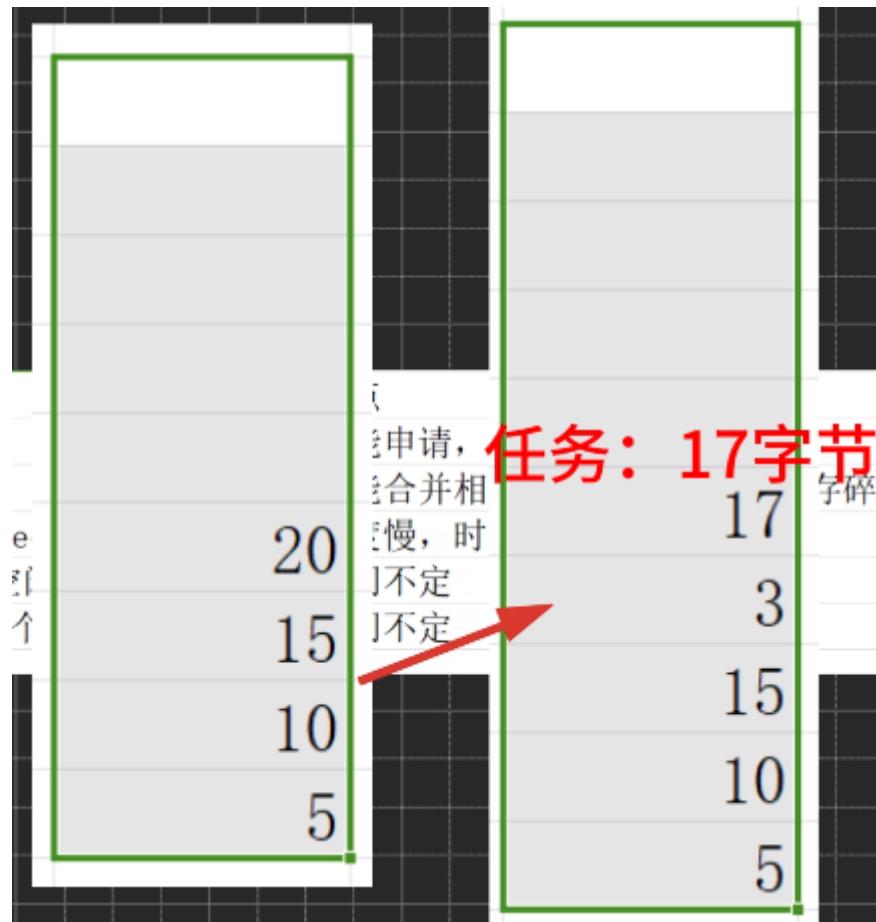


# heap2

heap2也是在数组上分配内存  
跟heap1不一样的地方在于  
heap2使用最佳匹配算法来分配内存  
并且支持了free功能



最佳匹配算法就是假如我要创建一个任务  
它需要17字节空间  
对这里有这几块空闲内存  
算法就可以找出最小的能满足17字节的内存  
也就是20字节这块空闲内存  
然后会把这20字节内存中的17字节  
分配给任务  
剩下的三字节仍然是空闲内存



可以被申请

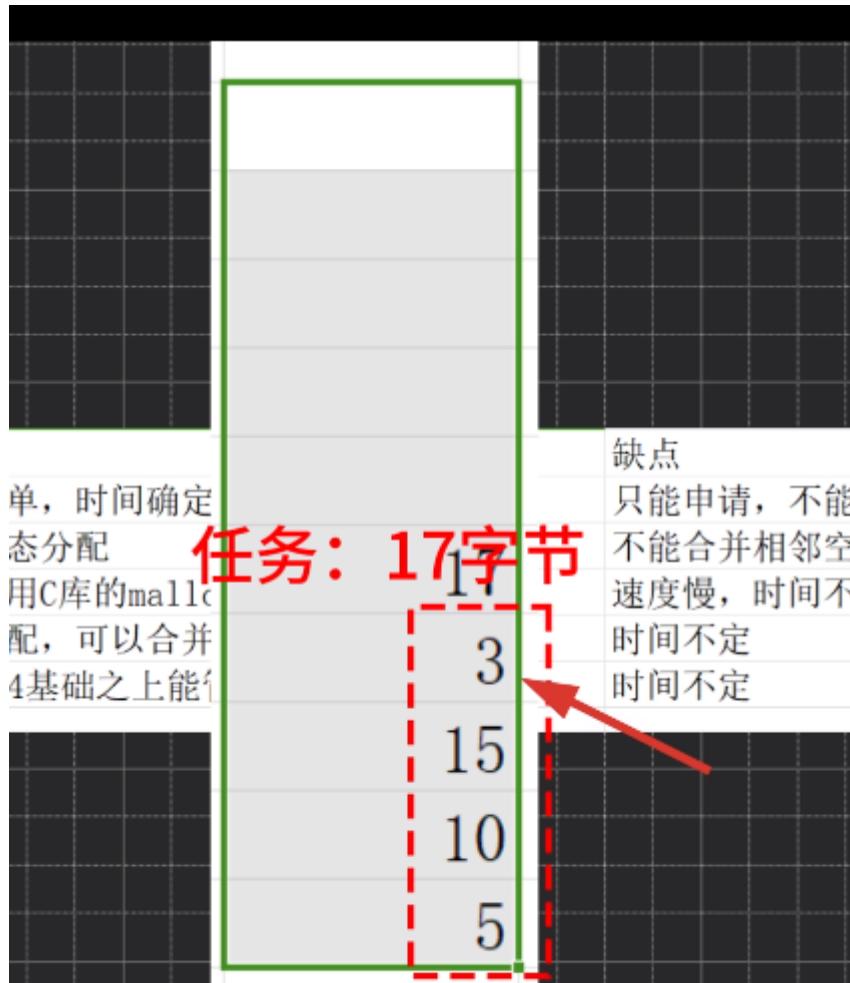
剩下的内存情况就是这样的了

之后再申请内存时

如果一直没有小于等于三字节

那么这个三字节空闲内存就一直不会被申请

就变成了内存碎片



申请内存时，如果一直没有小于  
空闲内存就一直不会被申请

这就是heap2产生内存碎片的原因了

## heap3

heap3是直接调用的标准C库的malloc和free函数  
由于标准C库的MALLOC和free函数的实现过于复杂  
占据的代码空间太大  
不适合用在资源紧缺的嵌入式系统中  
所以我们一般不用heap3  
这里就不过多介绍了

## heap4

跟heap1,heap2一样

heap4也是使用大数组来分配内存

heap4使用首次适应算法来分配内存

它还会把相邻的空闲内存合并为一个更大的空闲内存

这有助于较少内存的碎片问题

首次适应算法就是

假如我们需要申请一个12字节的内存

那么算法就会找到第一个符合大小的

空闲内存块

也就是20字节

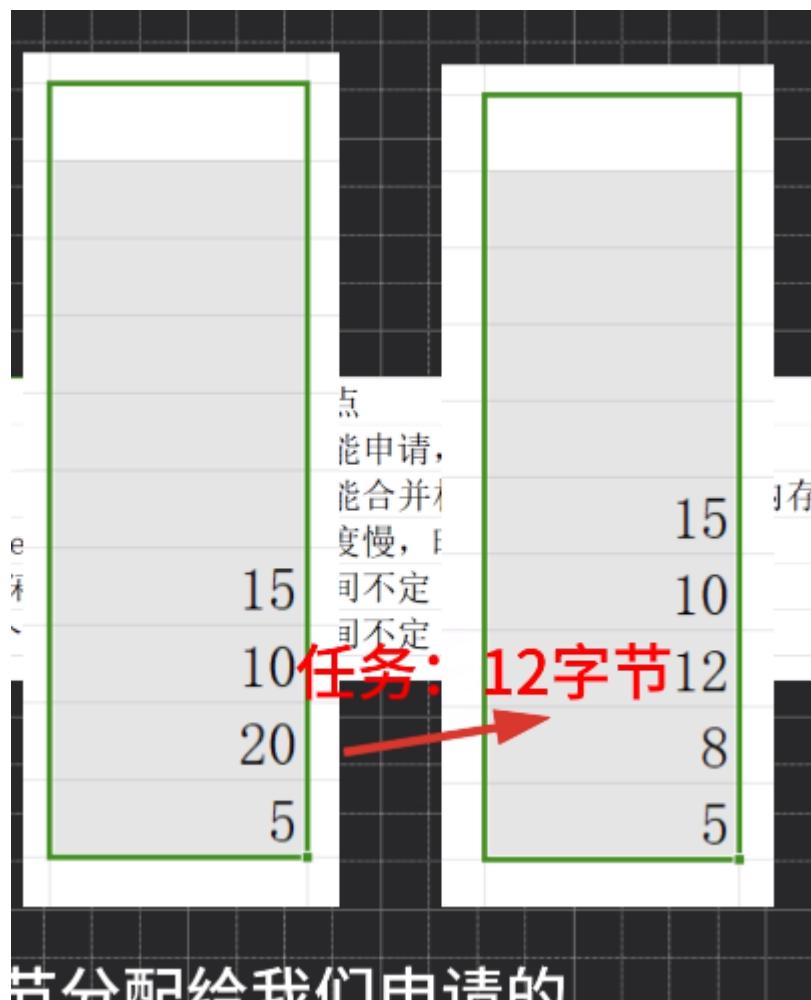
而不会像heap2那样找到最小符合的内存块15字节

他把20字节分为12字节和八字节

将12字节分配给我们申请的

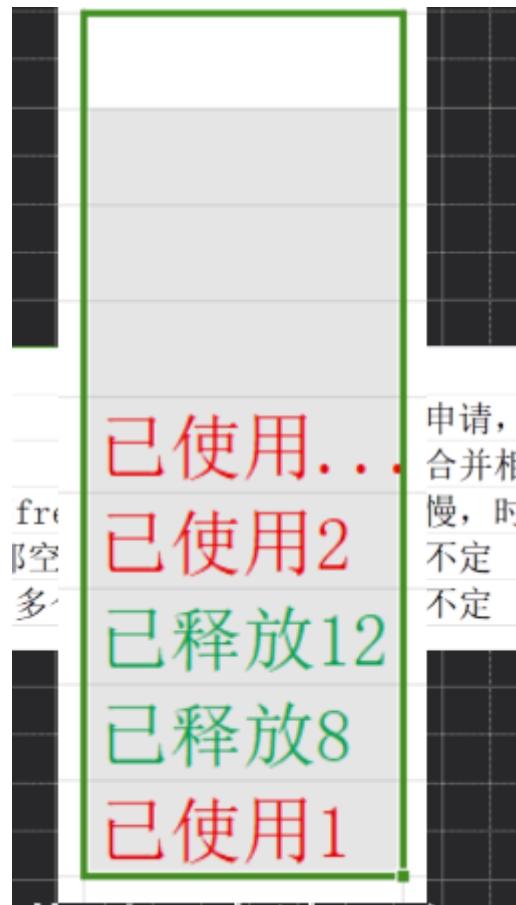
剩下的八字节仍然是空闲内存

可以被申请



当我们经过很多次申请释放后

内存可能就会成这样



如果内存不能合并

此时再申请一个15字节的任务空间

就不能申请成功

而这里的十二八字节空闲内存

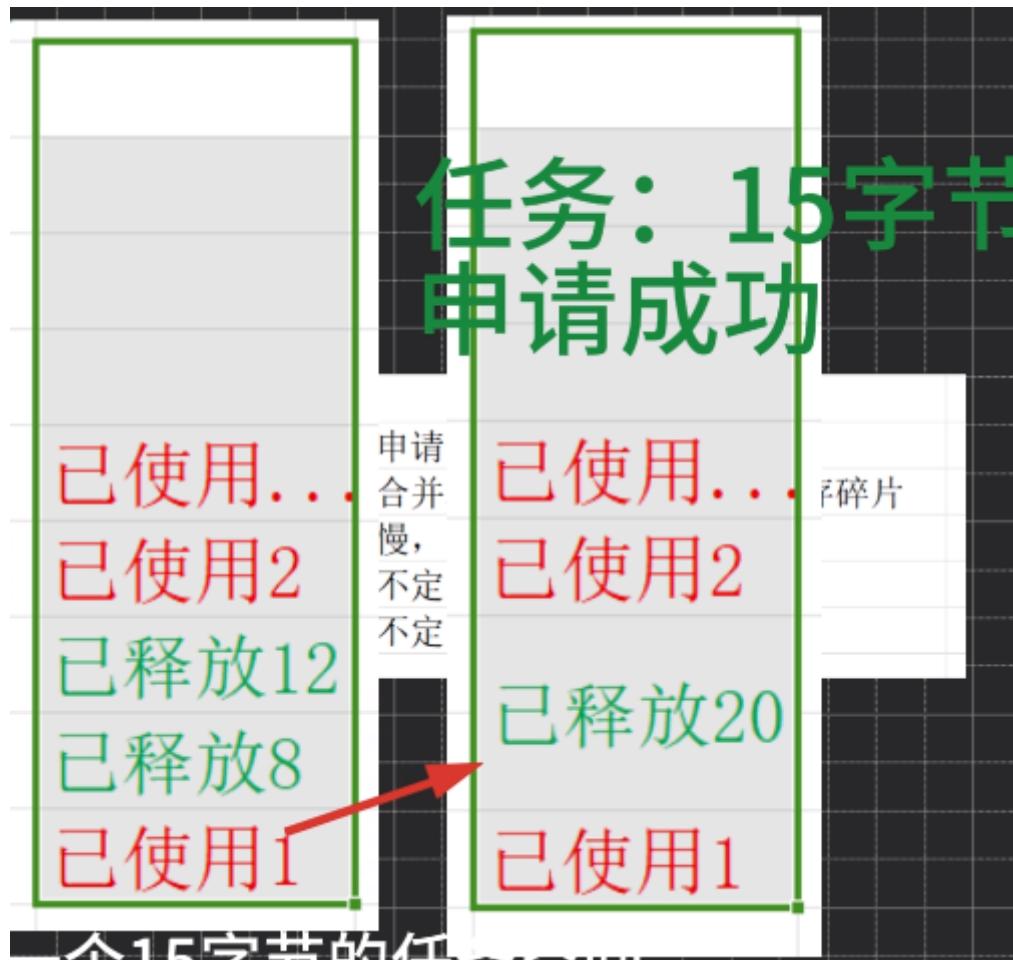
就会成为内存碎片

heap4会将相邻空闲的内存给合并到一起

这样再申请一个15字节的任务空间

就能申请成功

也减少了内存碎片



与heap4相比

heap2不会合并相邻的空闲内存

所以heap2会导致严重的碎片化问题

我们都会使用heap4

而不使用heap2

## heap5

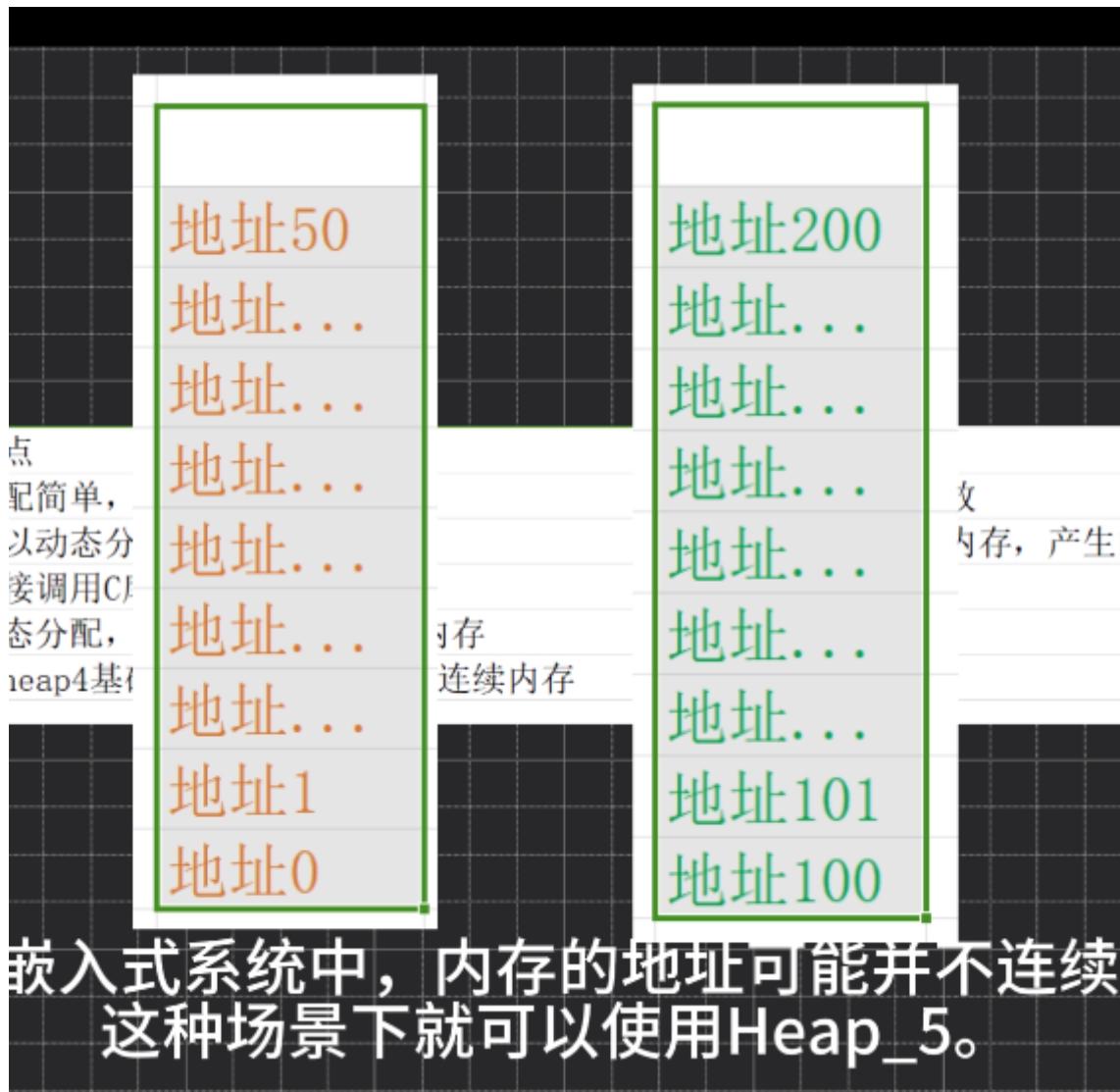
heap5分配内存释放内存的算法跟heap4是一样的

heap5相比于heap4

heap5可以管理多块分隔开的内存

在嵌入式系统中

内存的地址可能并不连续



这种场景下就可以使用heap5

总的来说

- 我们在很多场景下都是使用heap4
- 有多块内存会使用heap5
- 其他都很少用到了

## STM32 FreeRTOS多任务创建与删除

[STM32 FreeRTOS多任务创建与删除](#)哔哩哔哩bilibili

free rtos多个任务创建方式与任务删除

本期学习一下如何创建多个任务与删除任务

就是用这两个函数进行本期的实验

**任务创建：**

**xTaskCreate();**

**任务删除：**

**vTaskDelete();**

现在我们来一起看一下程序吧

这是上期完善好的工程

本期以创建三个任务为例

## 任务的创建

---

我们复制一下

在下面粘贴两个

修改一下

分别为任务一

任务二

任务三

这里也修改一下

最后的任务句柄也要修改

在上方写一下他们的任务函数

我这里直接复制一下

粘贴两个

然后改一下函数名字就好了

这里任务句柄还没有定义

在上方定义一下

复制再粘贴两个

然后修改一下名字

这样三个任务就创建好了

现在编译一下工程

点击进入debug

看下现象

我们全速运行一下程序

- 看到这里串口打印的信息是乱的
- 这是因为每个任务都有打印信息
- 而print f这个函数最后由串口发送数据
- 是非线程安全的
- 我们需要保护一下打印函数
- 在发送完数据前不被打断

现在退出debug会到main函数

关于代码保护有很多方式可以实现

我这里简单画一下

直接使用**临界区**进行代码保护

关于**临界区**问题

在以后的视频会讲解

这里不过多介绍了

进入**临界区**

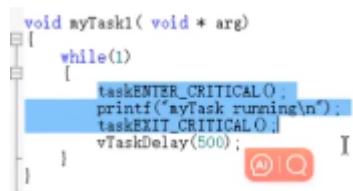
使用`task enter critical`这个宏函数`

有进入就有退出

退出**临界区**

使用`task exit critical`这个宏函数`

这两个可以理解为**关中段**和**开中段**



复制一下

替换一下其他任务中的打印函数

打印的信息也修改一下

这样打印就没问题了

我们再编译一下

再进入debug看下现象

点击全速运行

可以看到打印信息已经没问题了

我们停止程序运行

看下这里是先打印的任务三的信息

然后是任务一任务二

也就是说是先执行的任务三

然后再执行的任务一任务二

为什么相同的优先级任务创建顺序是123

执行却是312

No

这里的原因以后视频会讲

知道的小伙伴也可以在评论区留言评论

现在回到main函数

## 怎么删除任务

删除任务使用vtaskdelay函数

- 你想删除哪个任务
- 就把哪个任务的任务句柄传进来就可以了
- 如果想删除他自己
- 这也是可以的
- 但是不是传他自己的任务句柄
- 而是直接传入null就可以了

```
void myTask1( void * arg)
{
    while(1)
    {
        taskENTER_CRITICAL();
        printf("myTask1 running\n");
        taskEXIT_CRITICAL();
        vTaskDelete(NULL);
        vTaskDelay(500);
    }
}
```

现在我们实验一下  
这个函数在上方定义一个计数变量  
初始值为零  
在循环内让其自加  
让后其等于一时  
我们删除任务三  
其等于20  
我们删除它本身  
这段意思是执行第一次删除任务三  
执行第二次删除他自己

```
i++;
if(i == 1)
{
    vTaskDelete(myTaskHandler3);
}
if(i == 2)
{
    vTaskDelete(NULL);
}
```

我们保存编译一下  
进入debug  
看下现象  
点击全速运行  
已经有现象了  
我们停止运行  
看下第一次运行任务312运行到任务一时  
将任务三删除  
然后第二次只运行任务一二  
运行任务一时将它自己删除之后  
只运行任务二现象是正确的

# 再说一下另外一种多任务创建的方式

回到我们的main函数

另外一种方式就是用一个start任务创建其他任务

我们先创建一个start任务

复制粘贴一下

这里改成start task

这里任务名称也改一下

最后的任务句柄也改成start task handler

```
xTaskCreate(startTask, "startTask", 128, NULL, 2, &startTaskHandler);
```

我们在上方定义一个STARTASKHOR任务句柄

然后在上方写一个start task任务函数

在这个start task任务函数里去创建其他任务

将这些放到开始任务中就可以了

我们先剪切下来

整理一下代码

这种就是先创建一个开始任务

然后启动调度器

在开始任务里创建我们自己的任务

我们将剪切下的内容放到这里

开始任务

创建完其他任务后就没有用了

我们就可以使用vtask\_delete删除他自己参数填null

这样就可以了

```
void startTask(void * arg)
{
    xTaskCreate(myTask1, "myTask1", 128, NULL, 2, &myTaskHandler1);
    xTaskCreate(myTask2, "myTask2", 128, NULL, 2, &myTaskHandler2);
    xTaskCreate(myTask3, "myTask3", 128, NULL, 2, &myTaskHandler3);
    vTaskDelete(NULL);
}
```

其实不管是直接创建任务还是使用开始任务

创建其他任务

想使用哪种方式都是可以的

比较主流的还是使用开始任务创建其他任务

这种方式复制一下打印函数  
打印一些信息  
这里改成start函数名保存  
编译一下  
进入debug  
看下现象  
点击全速运行  
这里已经有现象了  
停止一下程序  
可以看到运行顺序是start123  
这种方式创建的任务  
是按照我们创建任务的顺序运行的  
而不像刚才那样  
任务执行顺序是312下方  
这里现象也是正确的

## 总结一下今天的内容

---

### 1多个任务创建

可以直接创建  
也可以借助一个开始任务创建  
一般常用方式是第二种

### 2使用v task delete函数删除任务

删除自己时参数传入Null  
删除其他任务  
传入相应任务句柄

## 3print函数是非线程安全的

使用时需要保护

总结：

- 1、多个任务创建，可以直接创建，也可以借助一个开始任务创建，一般常用方式是第二种。
- 2、使用vTaskDelete()函数删除任务，删除自己时，参数传入NULL，删除其他任务传入相应任务句柄。
- 3、printf函数是非线程安全的，使用时需要保护。

## STM32 FreeRTOS简单使用任务参数

简单实用下free rtos的任务参数

在之前视频创建任务时

任务参数都是填写的Null

任务参数用于向任务传递额外的参数

```
void startTask(void * arg)
{
    taskENTER_CRITICAL();
    printf("startTask running\n");
    taskEXIT_CRITICAL();
    xTaskCreate(myTask1, "myTask1", 128, NULL, 2, &myTaskHandler1);
    xTaskCreate(myTask2, "myTask2", 128, NULL, 2, &myTaskHandler2);
    xTaskCreate(myTask3, "myTask3", 128, NULL, 2, &myTaskHandler3);
    vTaskDelete(NULL);
}
```



任务函数可以通过接收不同的参数

来实现不同的操作

今天我们就来简单实用一下任务参数

# 任务函数接收：

参数1：执行操作1  
参数2：执行操作2  
参数3：执行操作3

我们看下创建任务函数的原型

这里任务参数的类型是void指针类型

也就是说任务的参数可以是任何类型的数据

int, char, float, struct, char\*, ...

```
BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,  
                        const char * const pcName,  
                        const uint16_t usStackDepth,  
                        void * const pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t * const pxCreatedTask )
```



今天我们就实现一下多个任务

使用一个打印任务函数

传进不同的参数

打印不同的内容

任务1 任务2 任务3



打印任务函数



操作1 操作2 操作3

打开上一期视频的工程  
这是上一期多任务创建的程序  
删除一下多余的程序  
上方这里的任务函数也删除一下  
现在就实现一下  
让这三个任务使用同一个任务函数  
先写一个打印任务函数框架  
现在任务框架已经写好了  
复制一下任务函数名  
在任务创建这里  
任务函数都使用同一个  
现在实现一下任务函数功能  
在使用任务参数之前  
要先把任务参数转成我们想要的参数类型  
先把打印函数复制进来  
我这里实现一下打印不同的字符串  
所以这里用一个char型指针接受一下任务参数  
下方print  
这里也改一下  
打印一下任务参数传进来的字符串  
这样任务函数就写好了  
每个任务都需要一个字符串  
任务参数我们在上方写一下任务参数  
传进去的参数类型要和任务函数中的对应起来  
这里也要定义成字符串  
我这里定义一个20字节的字符数组  
打印一个字符串  
My task  
1running  
另外两个任务的参数也定义一下  
这里改一下后面字符串也改一下  
定义好后传入进三个任务创建函数中就可以了

任务函数这里延迟500ms吧

每个任务500ms打印一次

```
char str1[20] = "myTask1 running!";
char str2[20] = "myTask2 running!";
char str3[20] = "myTask3 running!";

void myPrintf(void *arg)
{
    char *str = arg;
    while(1)
    {
        taskENTER_CRITICAL();
        printf("%s\n", str);
        taskEXIT_CRITICAL();
        vTaskDelay(500);

    }
}
```

创建任务

这里传进来刚才定义的参数

```
void startTask(void * arg)
{
    taskENTER_CRITICAL();
    printf("startTask running\n");
    taskEXIT_CRITICAL();
    xTaskCreate(myPrintf, "myTask1", 128, str1, 2, &myTaskHandler1);
    xTaskCreate(myPrintf, "myTask2", 128, str2, 2, &myTaskHandler2);
    xTaskCreate(myPrintf, "myTask3", 128, str3, 2, &myTaskHandler3);
    vTaskDelete(NULL);
```

现在程序就写好了

编译一下

进入debug

看下现象

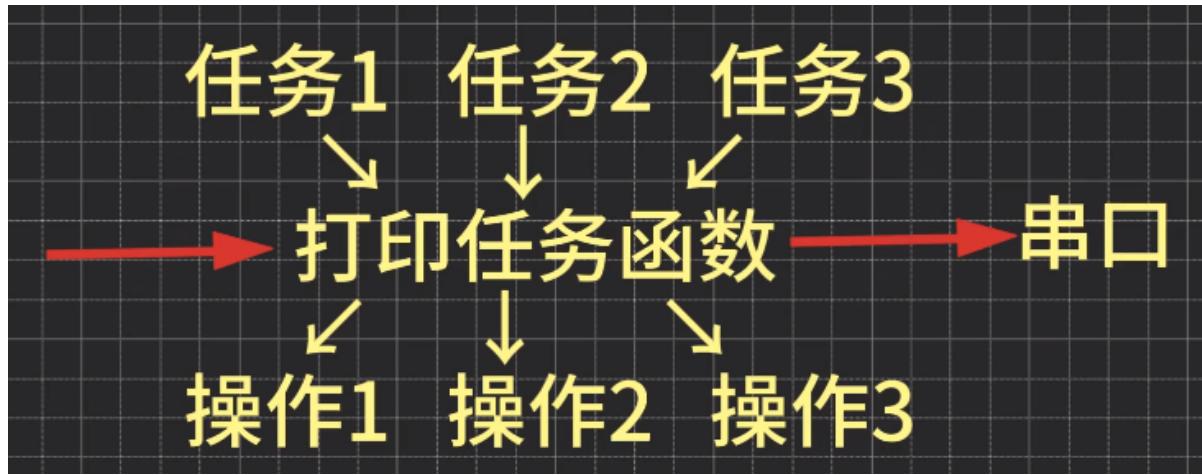
点击全速运行

可以看到是正常打印的

实现了传入不同参数执行不同操作

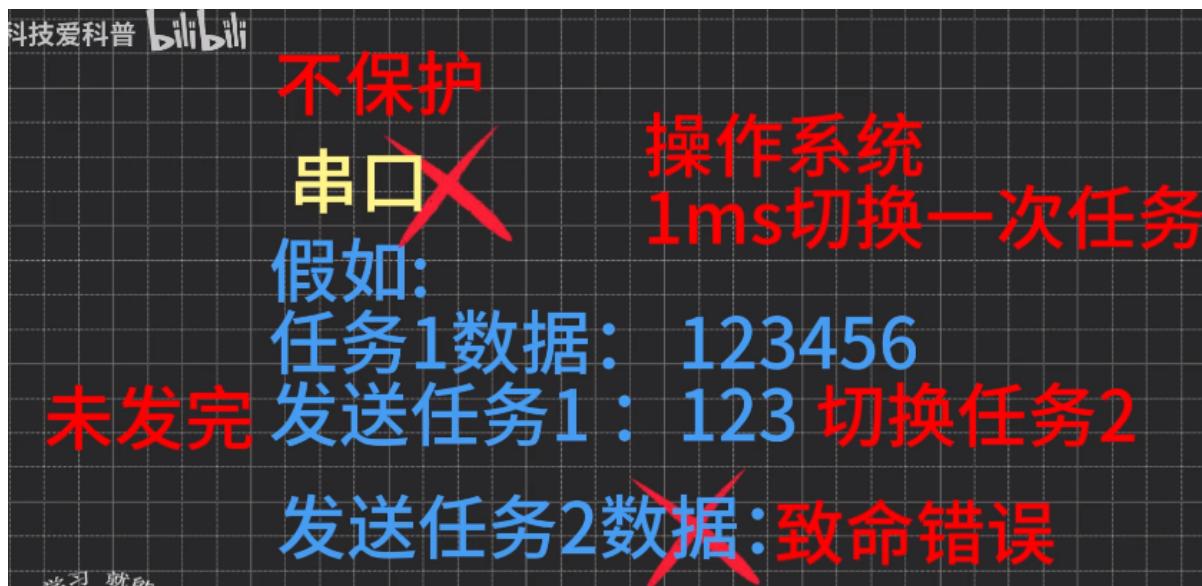
最后我们来说一下这种操作需要注意的地方

这个打印函数中最终是调用的串口发送数据



串口发送是比较耗时间的  
而操作系统是每1毫秒切换一次任务

如果我们不对它进行保护的话  
在任务发送出任务一的一半数据时  
任务执行突然切换到了任务二  
开始执行任务二  
发送任务二的数据  
而此时任务一数据还没有发完  
就会导致串口数据错误  
甚至导致程序崩溃



所以在多个任务使用同一个有限资源时  
我们要避免同时访问  
也就是需要互斥操作

# FreeRTOS队列简单使用

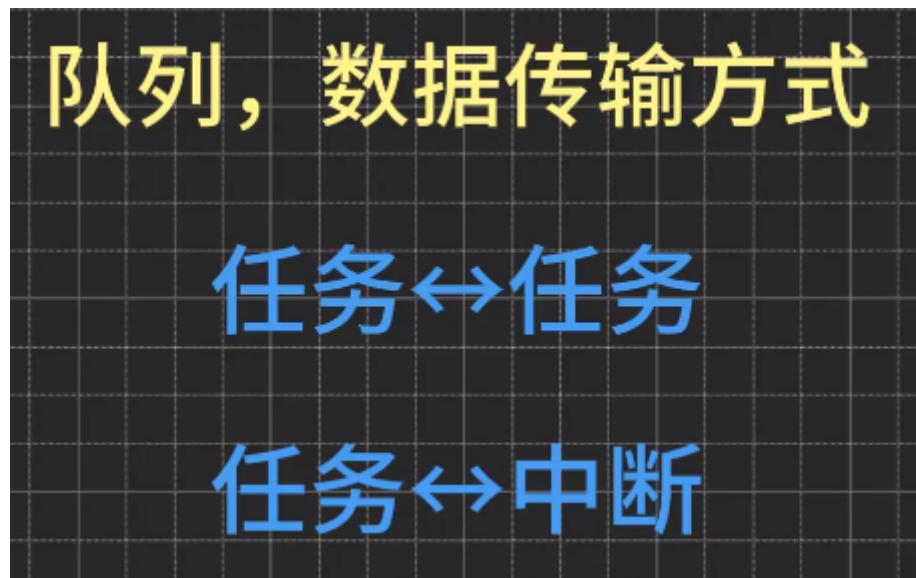
## 定义

简单实用下

free rtos队列队列是free rtos数据传输方式的一种

可以用于任务间数据传输

也可用于任务与中断间数据传输



在之前的视频中

我们使用串口打印信息

在每个任务中都有使用

```

void myTask1( void * arg)
{
    uint8_t i = 0;
    while(1)
    {
        taskENTER_CRITICAL();
        printf("myTask1 running\n");
        taskEXIT_CRITICAL();
        vTaskDelay(500);
    }
}

void myTask2( void * arg)
{
    while(1)
    {
        taskENTER_CRITICAL();
        printf("myTask2 running\n");
        taskEXIT_CRITICAL();
        vTaskDelay(500);
    }
}

void myTask3( void * arg)
{
    while(1)
    {
        taskENTER_CRITICAL();
        printf("myTask3 running\n");
        taskEXIT_CRITICAL();
        vTaskDelay(500);
    }
}

```

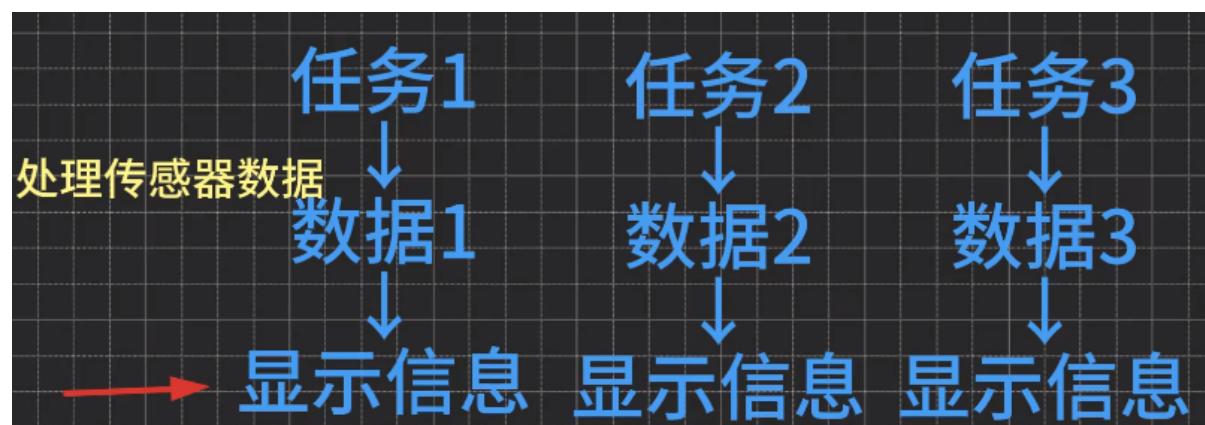
也就是这种结构

这种结构中每个任务既需要产生数据

比如处理传感器数据等

又需要关心数据的显示

导致任务逻辑层次不清晰



每个任务都有显示程序

我们就可以使用队列将程序分层一下

把显示相关程序单独拿出来

组成一个显示任务

其他几个任务

只需要将各自要显示的数据都写入队列

显示任务

只需要从队列中读取数据进行显示



这样整体结构就变得很清晰

任务1~3就可以专注自己的数据产生

而不需要关心怎么显示数据

显示任务就只关心怎么显示数据

而不需要关心数据怎么产生的

下面我们就使用队列简单实现一下这种结构

这是之前多任务视频的代码

我们先删除多余的程序

每个任务只保留打印程序

然后保存编译一下

没有错误就可以开始本期实验了

我们需要使用队列

# 实验实任务1任务2写队列任务3读数据并显示

所以要先包含一下队列的头文件

首先我们需要创建一个队列

使用create可以动态创建一个队列

我们右键转到定义

看下它的参数

这里它是使用了一个宏定义

定义了一下这个函数

方便了我们的使用

第一个参数就是队列的长度

也就是队列可以存放多少个数据

第二个参数是每个数据的大小

单位是字节

```
/*  
 *#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )  
 #define xQueueCreate( uxQueueLength, uxItemSize ) xQueueGenericCreate( ( uxQueueLength ), ( uxItemSize ), ( queueQUEUE_...  
 #endif
```

所以这里的参数就可以这样写

上面有两个任务需要写数据

队列长度我们就设置为二

而每个任务要写的数据是一个字符串

我们就设置20个字节的数据大小吧

```
xQueueCreate(2, 20);  
xTaskCreate(startTask, "startTask", 128, NULL, 2, &startTaskHandler);  
vTaskStartScheduler();
```

现在创建队列参数已经写好了

可是创建的队列我们去哪里找呢

其实这个函数的返回值就是我们创建的队列

我们可以看下函数原型

右键跳转过去

这里返回值是这种类型

我们也要定义这样一个变量  
用来接收创建好的队列句柄  
复制一下  
回到我们的代码  
在上方定义一个变量  
随便起一个队列名字  
用这个变量接收一下返回值  
这样一个队列就创建好了

```
myPrintfQueueHandler = xQueueCreate(2, 20);  
xTaskCreate(startTask, "startTask", 128, NULL, 2, &startTaskHandler);  
vTaskStartScheduler();
```

更严谨的写法

还需要在下方判断一下返回值是否为空为零

队列创建失败

我这里就不写了

队列创建好后就可以写队列了

现在开始修改任务函数

将打印程序注释掉

写队列

我们使用execute函数可以右键转到定义

看下参数也是一个宏定义函数

第一个参数是队列句柄

第二个参数是数据的地址

这个数据的值会被复制进队列

第三个参数是阻塞时间

队列满无法写入新数据时

会阻塞我们设定的时间

```
/*  
#define xQueueSend( xQueue, pvItemToQueue, xTicksToWait )
```

回到我们的代码

第一个参数我们填入队列句柄

第二个参数是数据地址

我们要先定义一下数据

在上方定义一个字符数组

数组大小就是创建队列时指定的数据大小

20字节不能随便定义大小

我们初始化为这个字符串

将数组名填入第二个参数

第三个参数是阻塞时间

我这里写零

也就是无法写入数据时

函数会立刻返回

这样写入队列就完成了

```
1 void myTask1( void * arg)
2 {
3     char data[20] = "myTask1 running";
4     while(1)
5     {
6         // taskENTER_CRITICAL();
7         // printf("myTask1 running\n");
8         // taskEXIT_CRITICAL();
9
10        xQueueSend(myPrintfQueueHandler, data, 0);
11        vTaskDelay(500);
12    }
13 }
```

我们把任务二同样的操作修改一下

这里字符串改为任务二

写入队列完成后就开始读队列并显示数据了

## 修改一下任务三读队列

使用eq receive函数

同样转到定义

看下参数

这里和谐队列参数基本一致

```
/*
#define xQueueReceive( xQueue, pvBuffer, xTicksToWait ) x
```

就不过多介绍了

回到我们的代码

直接写参数

第一个是队列句柄

第二个是读取的数据存放地址

我们也定义一个20字节字符数组

用来接收读取的数据

将数组名填入参数

第三个阻塞我们填入一个port max delay

也就是读不到数据就一直阻塞在这里

这样做的目的是没有数据时

就等在这里有数据

才会继续执行下方的显示程序

这样就不执行无用代码

提高效率

现在读队列已经完成了

```
void myTask3( void * arg)
{
    char data[20] = {0};
    while(1)
    {
        xQueueReceive(myPrintfQueueHandler, data, portMAX_DELAY); I
        taskENTER_CRITICAL();
        printf("myTask3 running\n");
        taskEXIT_CRITICAL();
        vTaskDelay(500);
    }
}
```

我们开始写显示程序

显示之前要先判断一下是否读取成功

右键转到定义

看下返回值类型

我们定义一个这种类型的变量

用来接收返回的状态

回到代码

在这里定义一下  
用这个变量接收一下返回值  
读取成功会返回pd true  
在下方判断一下返回值  
这里应该是pd true  
写pd pass也没问题  
然后把下方的显示代码放到if判断里面  
修改一下打印函数  
打印接收数据  
data数组中的字符串  
这样就可以了  
由于读队列没有数据是一直阻塞的  
下方的delay也可以不需要了

```
void myTask3( void * arg)
{
    char data[20] = {0};
    BaseType_t xStatus;
    while(1)
    {
        xStatus = xQueueReceive(myPrintfQueueHandler, data, portMAX_DELAY);
        if(xStatus == pdPASS)
        {
            taskENTER_CRITICAL();
            printf("%s\n", data);
            taskEXIT_CRITICAL();
        }
        //    vTaskDelay(500);
    }
}
```

现在我们编译一下  
没有错误  
我们点击进入debug看下现象  
点击全速运行  
可以看到串口是正常打印数据的  
这样就实现了任务间传输字符串

## 需要传输很多不同类型的数据

答案可以使用结构体

具体怎么做

我们现在来写一下代码

首先我们要定义一下要传输数据的结构体类型

在上方定义一下结构体

结构体中定义要传输的数据

我这里传输一下这个字符串和一个int变量

定义好结构体后

```
struct print{
    int i;
    char data[20];
};
```

要修改一下创建队列这里的数据大小

数据大小

我们使用size of来计算一下结构体大小

```
myPrintfQueueHandler = xQueueCreate(2, sizeof(struct print));
```

这样修改后再修改一下任务函数

先定义一个结构体变量

在初始化一下结构体的字符串参数下方

这里数据要取地址

这里我们再让变量i++

```
void myTask1( void * arg)
{
    struct print data = {
        .data = "myTask1 running"
    };
    while(1)
    {
        // taskENTER_CRITICAL();
        // printf("myTask1 running\n");
        // taskEXIT_CRITICAL();
        data.i++;
        xQueueSend(myPrintfQueueHandler, &data, 0);
        vTaskDelay(500);
    }
}
```

下方的任务二

同样的方法修改一下

然后修改一下显示任务

这里也要定义一个结构体变量

用来接收读队列数据

这里别忘了要取地址

下方这里我们就打印一下读取的结构体数据

这样修改就可以使用队列传递结构体数据了

```
void myTask3( void * arg)
{
    struct print data;
    BaseType_t xStatus;
    while(1)
    {
        xStatus = xQueueReceive(myPrintfQueueHandler, &data, portMAX_DELAY);
        if(xStatus == pdPASS)
        {
            taskENTER_CRITICAL();
            printf("%s:%d\n", data.data, data.i);
            taskEXIT_CRITICAL();
        }
        //    vTaskDelay(500);                                I
    }
}
```

我们编译一下

进入debug

看下现象

点击全速运行

可以看到现象是正确的

这样就实现了任务间依次传递多种参数了

## FreeRTOS队列常用操作介绍

free rtos队列常用操作介绍

上期视频

我们简单介绍了一下队列的创建和读写操作

# 队列操作

创建: `xQueueCreate()`

写队列: `xQueueSend()`

读队列: `xQueueReceive()`

创建队列我们使用了`xQueueCreate()`函数

这种方式是动态创建队列

队列的内存由FreeRTOS动态分配队列

还有静态创建方式

这种方式队列的内存由我们自己分配

静态创建队列

我们了解一下即可

常用的还是动态创建队列

读写队列

我们使用了这两个函数

`xCUSIN`函数是将数据写入队列的尾部

它等同于`xQueueSendToBack()`函数

尾部写入

也有头部写入函数

## 队列写操作

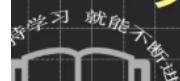
写队列: `xQueueSend()`

尾部写入

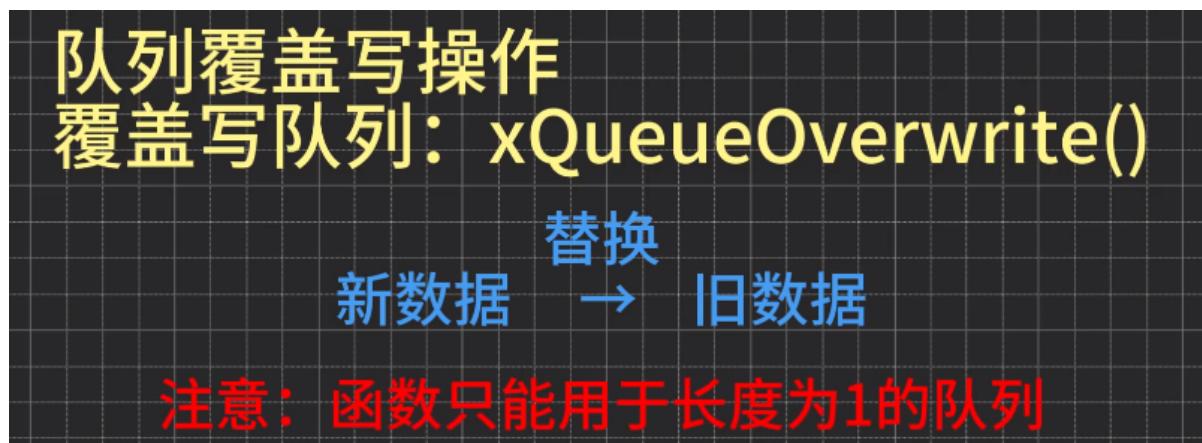
等同于: `xQueueSendToBack()`

头部写入

头部写入: `xQueueSendToFront()`



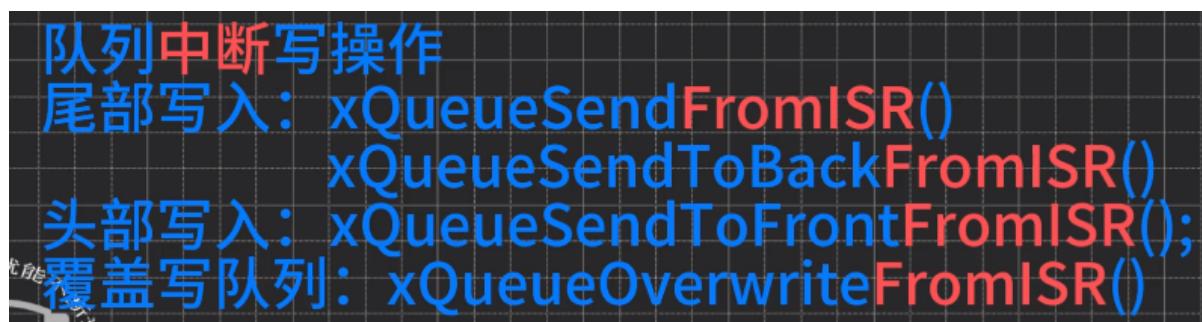
它们的使用方式是一样的  
区别就是插入数据的位置不同  
可以根据需要自行选择  
此外还有一种队列写入数据的方式  
就是覆盖写入  
使用 `x q overwrite` 函数  
顾名思义  
队列满的时候  
它可以用新数据覆盖之前的数据  
注意这个函数只能用于长度为1的队列



以上这几种就是写队列的常用基本操作了

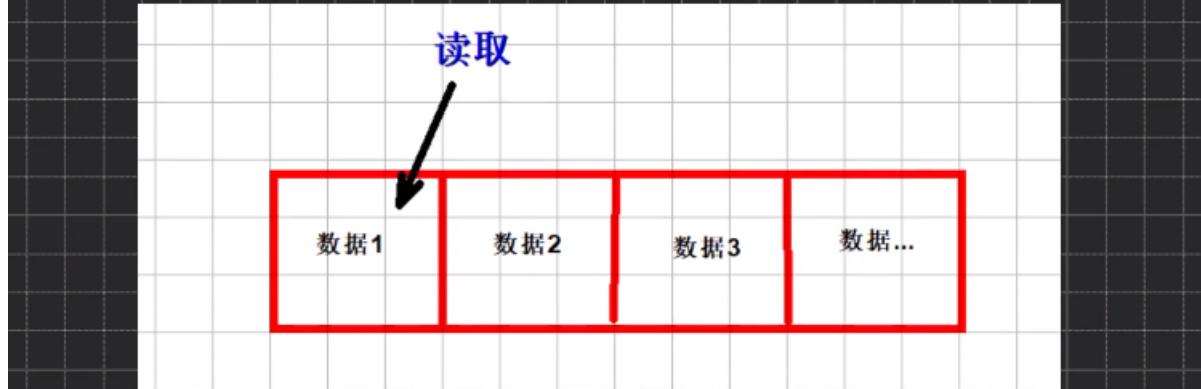
| 他们还有中断中使用的版本

后缀带有 `from ISR` 的函数



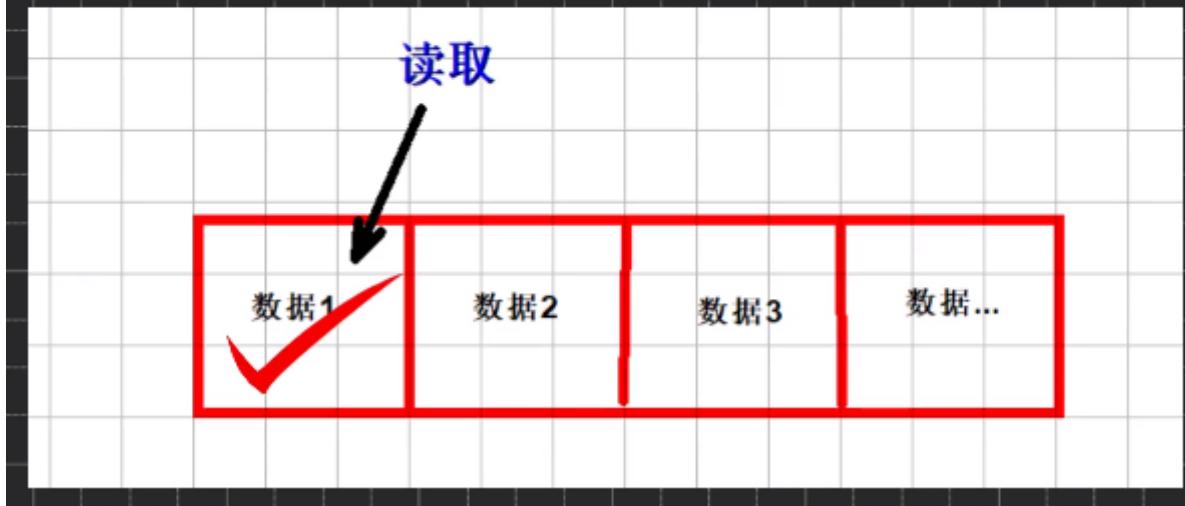
都是在中断中使用的  
接下来我们再来看一下读队列操作  
`x q receive` 读取数据后就会删除队列中的数据

## 队列操作 读取并删除：xQueueReceive()



如果不想删除数据  
我们就可以使用XQP函数  
它读取数据后不会删除数据  
数据还可以供其他任务使用

## 队列操作 只读取：xQueuePeek()



以上这两个就是读队列的常用基本操作了  
它们同样有中断中使用的版本  
除了创建读写队列

队列中断读操作

读取并删除: `xQueueReceiveFromISR()`

只读取: `xQueuePeekFromISR()`

## 还有这些常用操作函数

队列其他操作

删除队列: `vQueueDelete()`

复位队列: `xQueueReset()`

查询:

查询可用数据: `uxQueueMessagesWaiting()`

查询可用空间: `uxQueueSpacesAvailable()`

删除队列: 注意只能删除动态创建的队列, 因为它会释放内存

复位队列: 可以恢复队列的初始状态

下面这两个一个是查询队列可用数据个数

另一个是查询可用空间个数

看完这些常用队列操作

## 我们再来讲一下队列的传输

free rtos的队列是使用拷贝传输

也就是发送时要将数据复制到队列

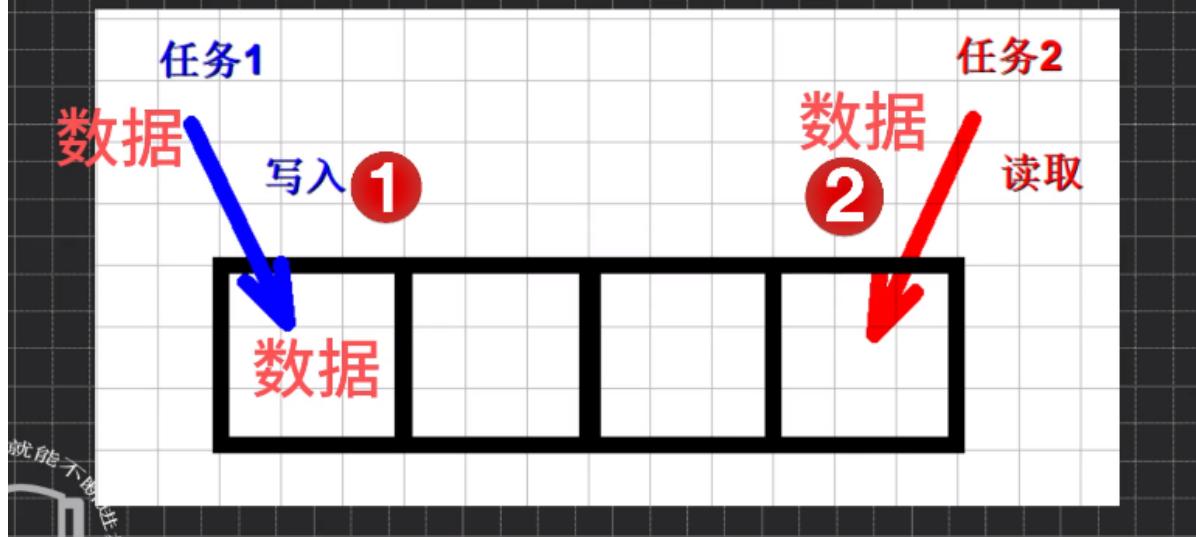
读取时再将队列数据复制出来

这期间数据被复制了两次

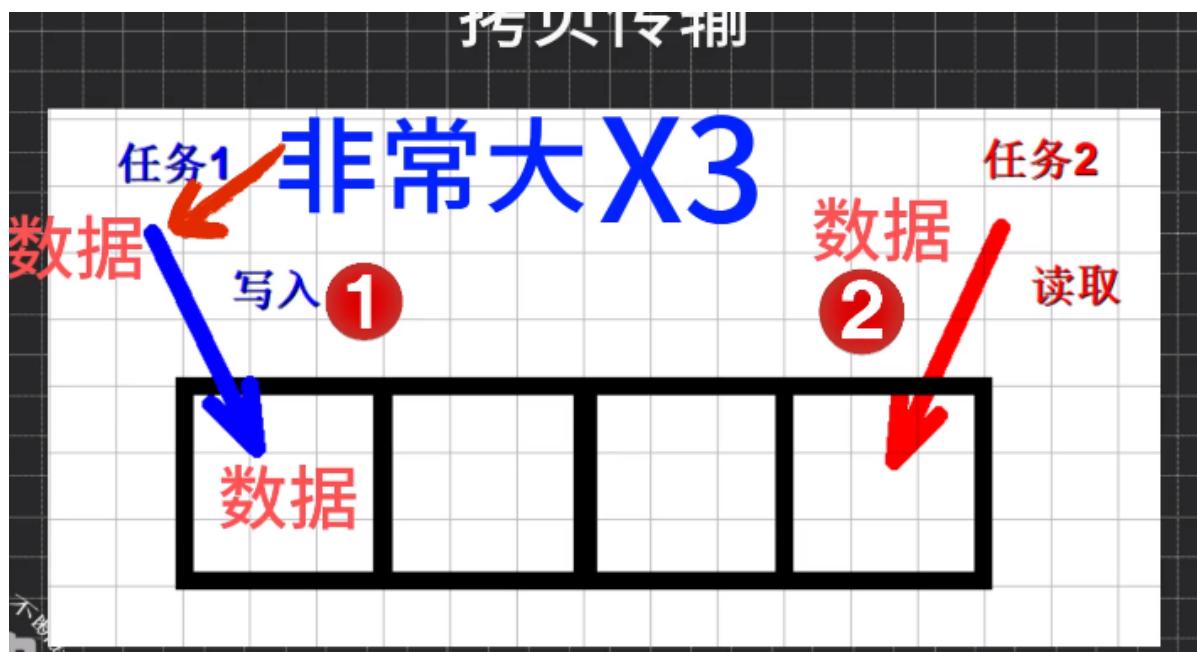
空间占用也多出了两份数据的空间大小

## 普通的传输

## 拷贝传输



如果要传输的数据非常大  
那么空间也会造成很大的浪费



这时候该怎么传输数据呢

我们可以使用指针来传输数据

现在我们来操作一下  
打开上一期的视频  
比如现在我们要传递一个1000字节的字符数据  
我们就需要先定义一下数据的全局变量  
在上方这里定义一个1000字节的char型数组

我们要使用指针传递数据  
所以创建队列  
这里的数据大小也要修改一下  
要传递的数据类型是char型指针  
用size of计算一下大小  
这样写就可以了

```
myPrintfQueueHandler = xQueueCreate(2, sizeof(char *));
```

回到上面  
我们开始修改写队列的任务  
这个结构体就不用了  
我们删掉多余代码  
在上面我们定义一个char型指针  
用来指向要传递数据的数组  
定义好后  
使用sprintf函数初始化一下数据  
后面这里加一个静态变量  
做一下数据的区分  
CNT做一下累加  
下面这里第一个参数不用修改  
最主要的是第二个参数  
大家可以暂停思考下第二个参数怎么填写  
因为我们传输的数据是一个地址  
这里第二个参数是需要传进去数据的地址  
所以我们需要传进去p data这个指针的地址

```
void myTask1( void * arg)
{
    char *pdata = data;
    static int cnt = 0;
    while(1)
    {
        taskENTER_CRITICAL();
        printf("myTask1 running\n");
        taskEXIT_CRITICAL();
        sprintf(pdata, "myTask1 running : %d", cnt++);
        xQueueSend(myPrintfQueueHandler, &0);
        vTaskDelay(500);
    }
}
```

要传输的数据：(地址)  
第二个参数需要填：数据地址  
所以应写：(地址) 的地址

前面加一个取地址符号

```
sprintf(pdata, "myTask1 running : %d", cnt++);
xQueueSend(myPrintfQueueHandler, &pdata, 0);
vTaskDelay(500);
```

这样写就可以了

下面任务二按照同样的方法修改

修改好之后

我们再来修改下任务三的读取数据

我们需要在任务三这里定义一个char型指针

用来接收读取到的地址

将这个指针传进读取函数中

然后修改一下下面的打印函数

打印一下接收到的地址里面的数据

这样修改之后

编译执行会有一个问题

```
void myTask3( void * arg)
{
    char *pdata;
    BaseType_t xStatus;
    while(1)
    {
        xStatus = xQueueReceive(myPrintfQueueHandler, &pdata, portMAX_DELAY);
        if(xStatus == pdPASS)
        {
            taskENTER_CRITICAL();
            printf("%s\n", pdata);
            taskEXIT_CRITICAL();
        }
        vTaskDelay(500);
    }
}
```

我们先来编译执行

看下现象

进入debug

点击全速运行

可以看到这里只打印了任务二的数据

没有任务一的数据

这是什么原因呢

其实这里就是我们采用指针传递数据

需要必须注意的地方

这里任务一

把自己的数据填入这个全局数组中后

调度到任务二后

任务二覆盖了任务一的数据

这里这三个任务都是相同优先级

所以他们的执行顺序是123

这样任务一的数据还没有被任务三接收

显示就被任务二给覆盖了

最后能接收的就只有任务二的数据了

所以我们在使用地址传递数据时

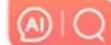
多个地方修改地址里面的数据要小心谨慎

现在这个实验的问题也很好解决

只需要把接收数据的任务

也就是任务三优先级调高一点就可以了

```
xTaskCreate(myTask1, "myTask1", 128, NULL, 2, &myTaskHandler1);  
xTaskCreate(myTask2, "myTask2", 128, NULL, 2, &myTaskHandler2);  
xTaskCreate(myTask3, "myTask3", 128, NULL, 3, &myTaskHandler3);  
vTaskDelete(NULL);
```



任务三的优先级调高

当队列中写入数据后

任务三就会马上抢先执行

数据也就不会被覆盖了

现在编译运行看下现象

可以看到任务一发送的数据也会被打印了

现在总结一下传输大数据块时需要注意的地方

由于采用指针方式传输

所以存储数据的数组要定义成全局变量

在多个地方修改数据时

要注意数据覆盖问题

谨慎修改

大数据块传输（只传输数据指针，节省空间）

- 1、定义存储数据的数组时要定义成全局变量
- 2、修改数据时注意覆盖问题要小心谨慎