

# FreeRTOS 实现

## 作者: 向仔州

FreeRTOS 各个文件组成.....	3
深入理解 FreeRTOS 的内部实现.....	9
FreeRTOS 移植到 STM32F1.....	18
详解 FreeRTOSConfig.h 文件.....	20
在 systick 中断中调用 FreeRTOS 的中断处理函数.....	22
创建静态任务.....	25
xTaskCreateStatic(...)//静态任务创建函数.....	25
动态创建任务.....	27
xTaskCreate(...)//动态创建任务.....	27
杀死任务.....	29
vTaskDelete(...)//杀死任务函数.....	29
任务状态转换表, 挂起, 阻塞, 就绪, 运行的含义.....	30
vTaskSuspend(...)//挂起任务.....	34
vTaskResume(...)//恢复任务.....	34
xTaskGetTickCount( void )//获取系统开始到现在执行了多少时间, 以 1 毫秒为单位 .....	35
vTaskDelay(...)//阻塞延时函数.....	35
vTaskDelayUntil(...)//获取精确的延时时间(绝对延时函数), 延时时间到, 任务马上执行, 不受中断和优先级打断.....	36
pdMS_TO_TICKS //将毫秒数转换成节拍数.....	36
消息队列.....	36
xQueueCreate(...)//创建消息队列, 返回句柄.....	38
xQueueSend(...)//写队列, 把数据写入队列尾部.....	38
xQueueSendToFront(...)//写队列, 把数据写入队列头部, 如果你的任务有紧急的事件就需要插队, 就用这个函数.....	38
xQueueSendFromISR(...)//在中断中用非阻塞发送消息的函数, 这个函数不会死等, 没有数据也会执行结束.....	38
xQueueReceive(...)//接收发送过来的消息.....	38
解决 printf 打印字符乱序, 采用消息队列, 修改 printf.....	39
信号量.....	42
二值信号量.....	42
QueueHandle_t //信号量句柄定义.....	42
xSemaphoreCreateBinary() //创建二值信号量变量, 默认为 0 .....	43
xSemaphoreGive() //发送信号量, 向信号量变量写 1 .....	43
xSemaphoreTake() //信号量接收函数 .....	43
计数信号量.....	44
xSemaphoreCreateCounting(...)//创建计数信号量.....	45
互斥量.....	49
xSemaphoreCreateMutex() //创建互斥量 .....	51
xSemaphoreGive(...)//发送互斥量.....	51
xSemaphoreTake(...)//接收互斥量.....	51
事件使用, 解决任务同步.....	54
EventGroupHandle_t //定义事件句柄.....	56

xEventGroupCreate( void ) //创建事件组.....	56
xEventGroupSetBits(...)//发送事件函数.....	56
xEventGroupWaitBits(...)//阻塞等待事件到来.....	56
多事件触发才能执行案例.....	58
多事件也可以多个任务分开来发送.....	59
<b>软件定时器.....</b>	<b>60</b>
TimerHandle_t //定义定时器句柄.....	60
xTimerCreate(...)//创建定时器.....	60
xTimerStart(...)//启动定时器.....	60
xTimerStartFromISR(...)//中断中启动定时器.....	60
xTimerStop(...)//软件定时器停止函数.....	60
<b>任务通知.....</b>	<b>62</b>
xTaskNotify(...) //发送带参数的通知给指定的任务.....	63
xTaskNotifyWait(...)//阻塞程序，接收带参数的任务通知，放弃阻塞，继续向下执行.....	63
XTaskNotifyGive(...)//发送二值信号量类型的任务通知.....	64
ulTaskNotifyTake(...)//接收二值信号量类型的任务通知.....	64
ulTaskNotifyTake(...)//将接收任务改成计数信号量任务通知模式，pdFALSE.....	65
<b>内存管理.....</b>	<b>66</b>
FreeRTOS 内存申请 4 种方式	
heap_1.c.....	66
heap_2.c.....	68
Heap_3.c malloc 函数和 free 函数.....	69
Heap_4.c 使用最多的方式.....	69
xPortGetFreeHeapSize( void ) //获取当前堆内存剩余大小.....	69
void *pvPortMalloc( size_t xWantedSize ) //内存分配函数，按字节分配.....	69
void vPortFree( void *pv ) //内存释放.....	70
中断管理，主要是 FreeRTOS 处理硬件中断的使用.....	71
taskENTER_CRITICAL_FROM_ISR() //进入临界区.....	71
taskEXIT_CRITICAL_FROM_ISR( ... ) //退出临界区.....	71
xQueueSendFromISR(..., ..., ...) //中断里发送消息队列函数使用，在消息队列章节有介绍.....	71
<b>CPU 使用率统计.....</b>	<b>73</b>
查询系统堆大小，用于检测程序是否长时间正常运行.....	76
size_t xPortGetFreeHeapSize( void ) //系统剩余堆大小检测.....	76
size_t xPortGetMinimumEverFreeHeapSize( void )//查询堆最小空间，其实和剩余堆大小查询一样的功能.....	78
void vApplicationMallocFailedHook( void ); //堆申请失败回调函数.....	78
查询每个任务栈使用大小，用于检测系统是否长时间正常运行.....	78
UBaseType_t uxTaskGetStackHighWaterMark(....) //查看当前任务栈还剩多少.....	78

## FreeRTOS 各个文件组成

### portmacro.h 文件

```
#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;

#if( 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
#endif
```

需要用到的变量类型重新定义，以符合 MCU 型号。

### list.h

```
#include "portmacro.h"

struct xLIST_ITEM
{
    TickType_t xItemValue;           /*辅助值，用于帮节点做顺序排列*/
    struct xLIST_ITEM * pxNext;      /*指向链表下一个节点*/
    struct xLIST_ITEM * pxPrevious;  /*指向链表上一个节点*/
    void * pvOwner;                /*指向拥有该节点内核对象，通常时候TCB(任务控制器)*/
    void * pvContainer;            /*指向该节点所在的链表*/
};

typedef struct xLIST_ITEM ListItem_t; //节点重新取个名字
```

一个节点是什么样的

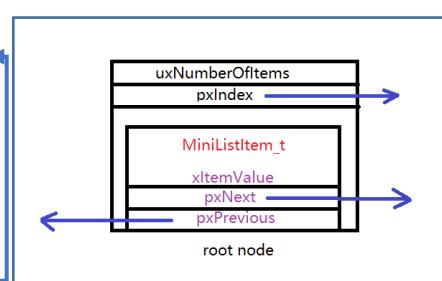
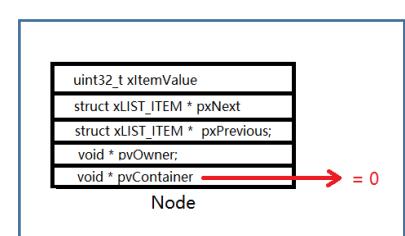
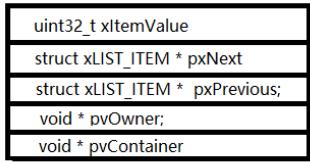
### list.c

```
/*节点初始化*/
void vListInitialiseItem( ListItem_t * const pxItem )
{
    pxItem->pvContainer = 0; //初始化节点所在的链表为空，表示节点还没有插入任何链表
}
```

根节点定义

### list.h

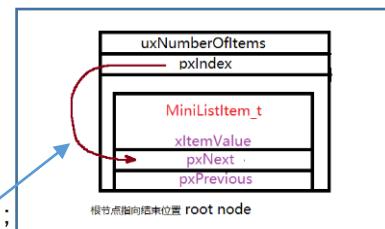
```
/*链表结构体，根节点类型定义*/
typedef struct xLIST
{
    UBaseType_t uxNumberOfItems; /*当前链表有多少个节点统计器*/
    ListItem_t * pxIndex;       /*链表节点索引指针*/
    MiniListItem_t xListEnd;    /*链表最后一个节点*/
} List_t;
```



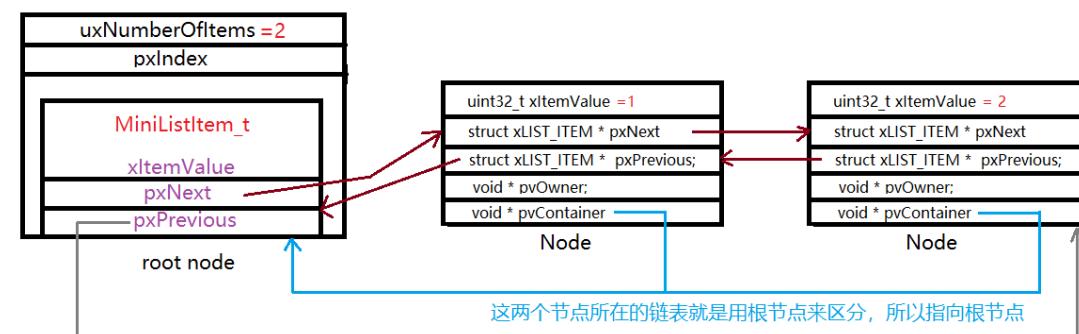
## 链表根节点初始化

list.c

```
//链表根节点初始化
void vListInitialise( List_t * const pxList )
{
    pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );
    pxList->xListEnd.xItemValue = portMAX_DELAY; //链表最后一个节点辅助值设置为最大
    pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd ); //将当前节点的下一个节点指向自身为空
    pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd ); //将当前节点上一个节点指向自身为空
    pxList->uxNumberOfItems = ( UBaseType_t ) 0U; //链表节点计数器设置为0，表示链表为空
}
```

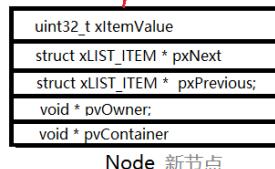
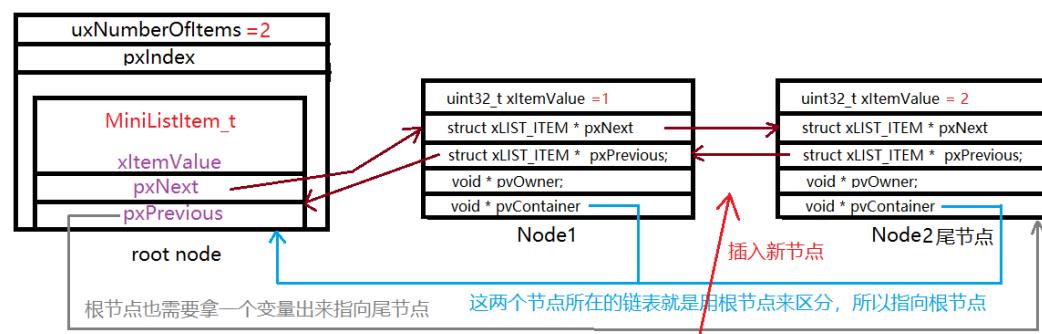


## 节点插入链表



根节点也需要拿一个变量出来指向尾节点

这是原有链表



Node 新节点

## list.c

```

/*新节点插入链表尾部
* pxList 填入你的节点插入哪一个链表
* pxNewListItem 传入新节点地址
*/
void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
{
    ListItem_t * const pxIndex = pxList->pxIndex; //定义个索引链表变量pxIndex指向链表最后一个节点
    pxNewListItem->pxNext = pxIndex; //新节点的下一个节点指向尾节点

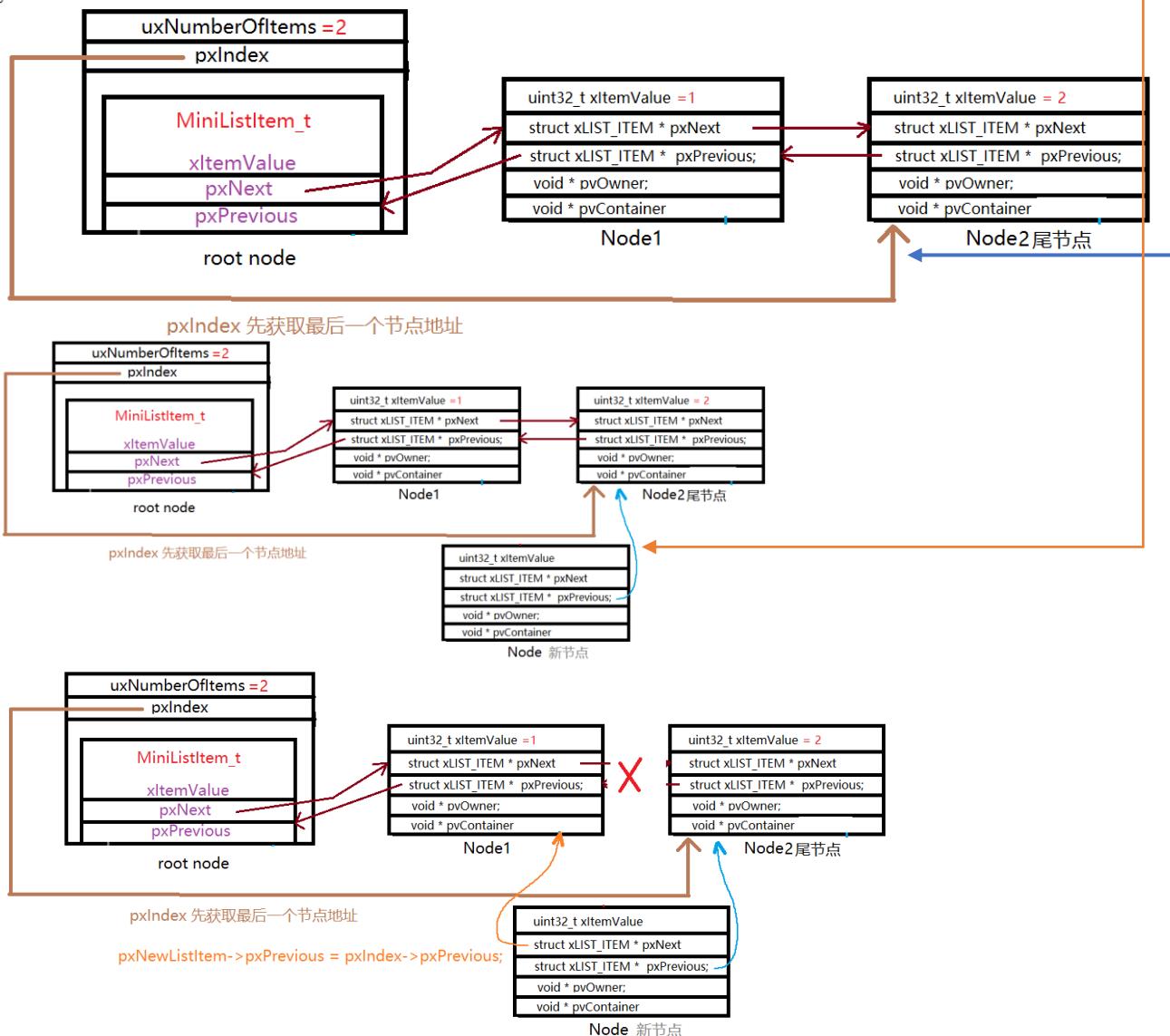
    //新节点的上一个节点, 等于原来尾节点的上一个节点
    pxNewListItem->pxPrevious = pxIndex->pxPrevious;

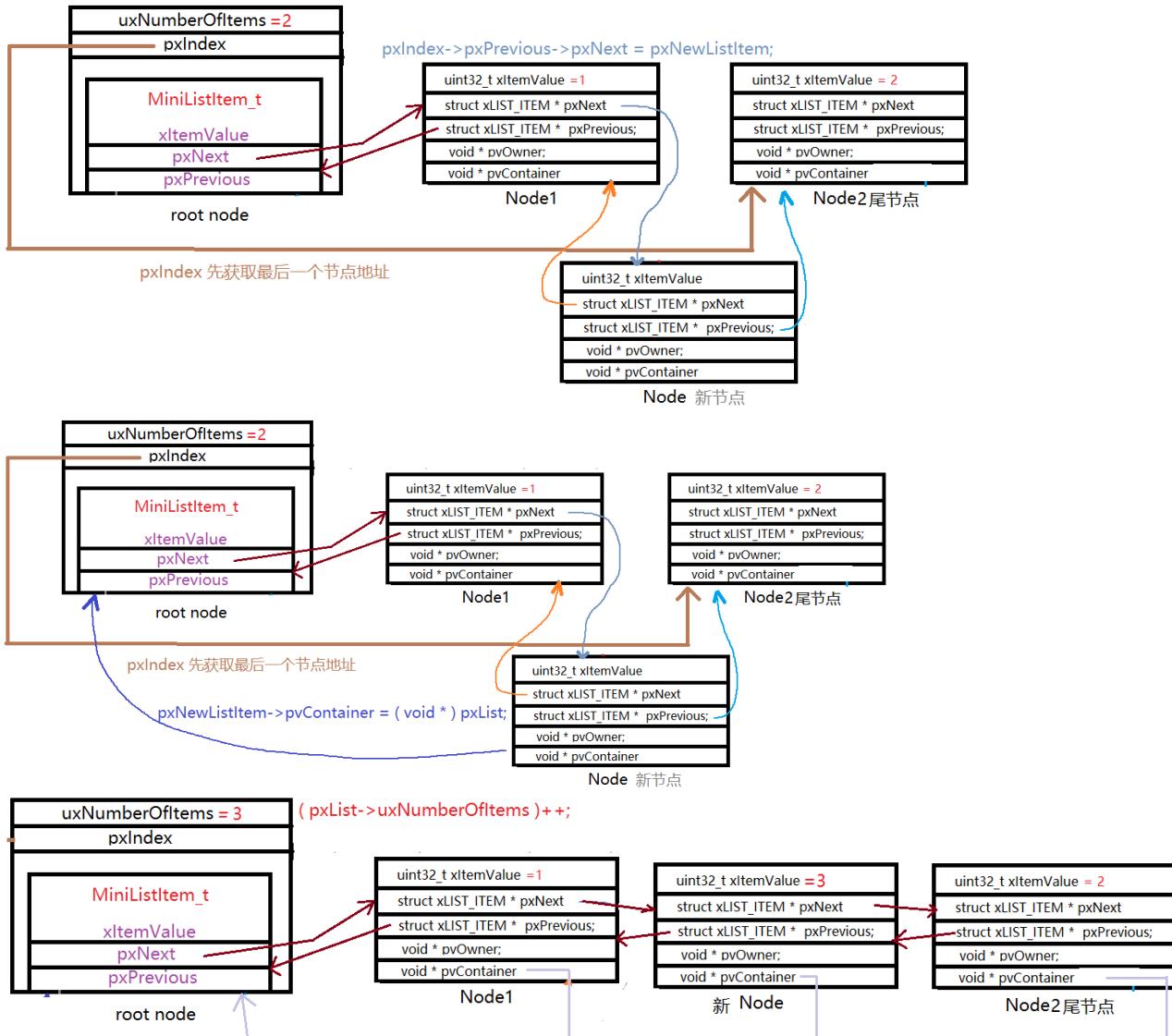
    pxIndex->pxPrevious->pxNext = pxNewListItem; //尾节点的上一个节点的下一个节点等于新节点
    pxIndex->pxPrevious = pxNewListItem; //尾节点的上一个节点就是等于新插入的节点

    pxNewListItem->pvContainer = ( void * ) pxList;//记住该新节点在哪一条链表

    ( pxList->uxNumberOfItems )++; //链表计数器+1
}

```





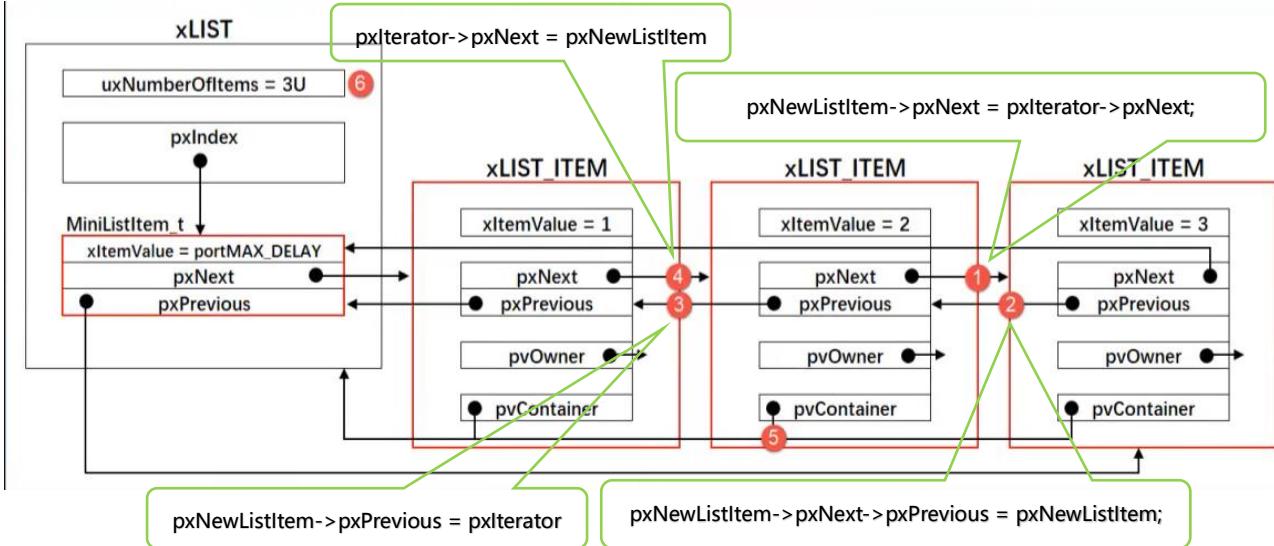
除了尾部插入新节点，也可以按照升序的方式插入新节点，这样链表就是按照 1,2,3.... 节点顺序排列

## list.c

```

/*
 * 将新节点按照升序排列，插入到链表中
 * pxList 填入你的节点插入哪一个链表
 * pxNewListItem 传入新节点地址
 */
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
{
    ListItem_t *pxIterator;
    const TickType_t xValueOfInsertion = pxNewListItem->xItemValue; // 获取新节点辅助值
    /* 寻找节点要插入的位置 */
    if( xValueOfInsertion == portMAX_DELAY )
    {
        pxIterator = pxList->xListEnd.pxPrevious;
    }
    else
    {
        for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemValue <= xValueOfInsertion;
        {
            /* 没有事情可做，不断循环只是为了找到节点要插入的位置 */
        }
    }
    pxNewListItem->pxNext = pxIterator->pxNext; // 新节点下一个指向原先节点下一个
    pxNewListItem->pxNext->pxPrevious = pxNewListItem; // 新节点下一个的上一个指向新节点
    pxNewListItem->pxPrevious = pxIterator; // 新节点的上一个指向第1个节点，或者是插入后尾节点以前的上一个节点
    pxIterator->pxNext = pxNewListItem; // 第1个节点下一个就指向新节点，或者是插入后尾节点以前的上一个节点，指向下一个新节点
    pxNewListItem->pvContainer = ( void * ) pxList; // 记住该节点所在哪一个链表
    ( pxList->uxNumberofItems )++; // 链表节点计数器
}

```



## 删除节点

list.c

```
/*
 * 删除链表中的节点
 * pxItemToRemove 填入要删除的节点
 */
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
{
    List_t * const pxList = ( List_t * ) pxItemToRemove->pvContainer; // 获取节点所在的链表号
    pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
    // 比如删除2节点，那么将2节点的下一个3节点的pxPrevious，指向2节点上一个节点1节点
    pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
    // 2节点pxPrevious上一个节点1节点，1节点的下一个pxNext 指向2节点的下一个3节点

    if( pxList->pxIndex == pxItemToRemove ) // 如果根节点是2节点
    {
        pxList->pxIndex = pxItemToRemove->pxPrevious;
    }
    else
    {
    }
    pxItemToRemove->pvContainer = 0; // 将2节点的所在链表号填空，表示2节点删除了。
    ( pxList->uxNumberOfItems )--; // 链表节点计数器-1
    return pxList->uxNumberOfItems; // 返回剩余节点个数
}
```

## 任务控制块创建

在 FreeRTOS.h 中定义

```
/* 任务控制块 */
typedef struct tskTaskControlBlock
{
    volatile StackType_t *pxTopOfStack; /* 指向任务控制块栈顶 */
    ListItem_t xStateListItem; /* 任务节点指针 */
    StackType_t *pxStack; /* 任务栈起始地址 */
    char pcTaskName[16]; /* 任务名称，字符串形式，最大16字节 */
} tskTCB;
typedef tskTCB TCB_t; // 重新命名任务控制块
```

## task.c 任务创建函数

```
typedef void (*TaskFunction_t)( void * ); //定义函数指针
typedef void * TaskHandle_t; //任务句柄
typedef unsigned long UBaseType_t;

TaskHandle_t xTaskCreateStatic( TaskFunction_t pxTaskCode, /*任务入口*/
                               const char * const pcName, /*任务名称以字符串形式写入*/
                               const uint32_t ulStackDepth, /*任务栈大小，单位为32位*/
                               void * const pvParameters, /*任务形参*/
                               UBaseType_t uxPriority, /*任务优先级*/
                               StackType_t * const puxStackBuffer, /*任务栈起始地址*/
                               StaticTask_t * const pxTaskBuffer ) /*任务控制块指针*/
{
    TCB_t *pxNewTCB; //新任务块指针
    TaskHandle_t xReturn;

    if( ( pxTaskBuffer != NULL ) && ( puxStackBuffer != NULL ) )
    {
        pxNewTCB = ( TCB_t * ) pxTaskBuffer; //新的任务控制块赋值给pxNewTCB
        pxNewTCB->puxStack = ( StackType_t * ) puxStackBuffer; //任务控制块起始地址赋值给新任务栈

        /*创建新的任务*/
        prvlnInitialiseNewTask( pxTaskCode, pcName, ulStackDepth, pvParameters, uxPriority, &xReturn, pxNewTCB, NULL );
        prvAddNewTaskToReadyList( pxNewTCB );
    }
    else
    {
        xReturn = NULL;
    }

    return xReturn;
}
```

具体任务创建函数实现细节我们暂时不讲，先使用起来，后面有动态任务创建的细节讲解。

## 任务就绪表实现

```
//定义任务栈
StackType_t Task1Stack[128];
StackType_t Task2Stack[128];
```

创建任务 第 1.需要任务栈，这  
个任务栈就是用来存放任务全局  
变量和临时变量的

```
//定义任务控制块
TCB_t Task1TCB;
TCB_t Task2TCB;
```

第 2.需要任务控制块，有多少个  
任务就需要多少个任务控制块

```
//定义任务句柄
TaskHandle_t Task1_Handle;
TaskHandle_t Task2_Handle;
```

第 3.定义任务句柄

```
void Task1(void *p_arg) //任务1
{
    while(1)
    {

    }
}

void Task2(void *p_arg) //任务2
{
    while(1)
    {

    }
}
```

第 4.任务执行函数

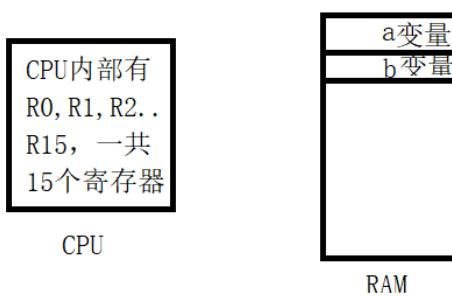


以上过程都会和任务就绪表有关系

## 深入理解 FreeRTOS 的内部实现

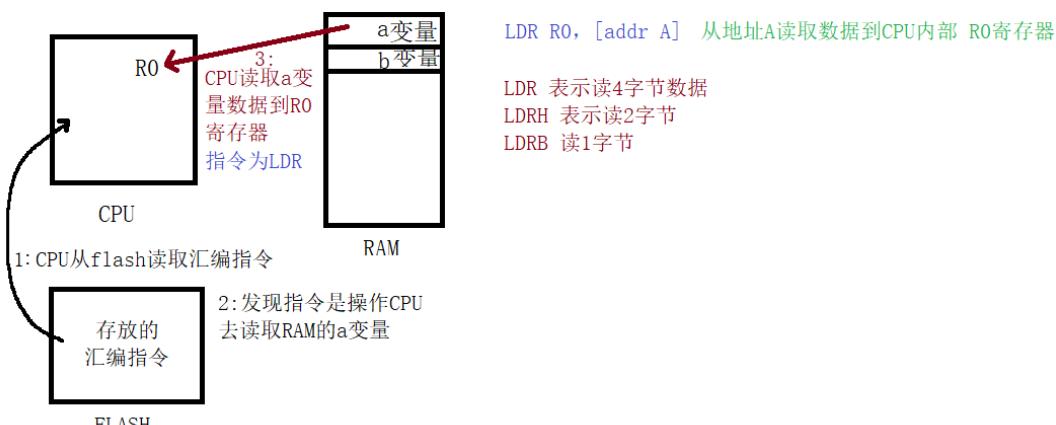
### ARM 汇编架构

ARM架构

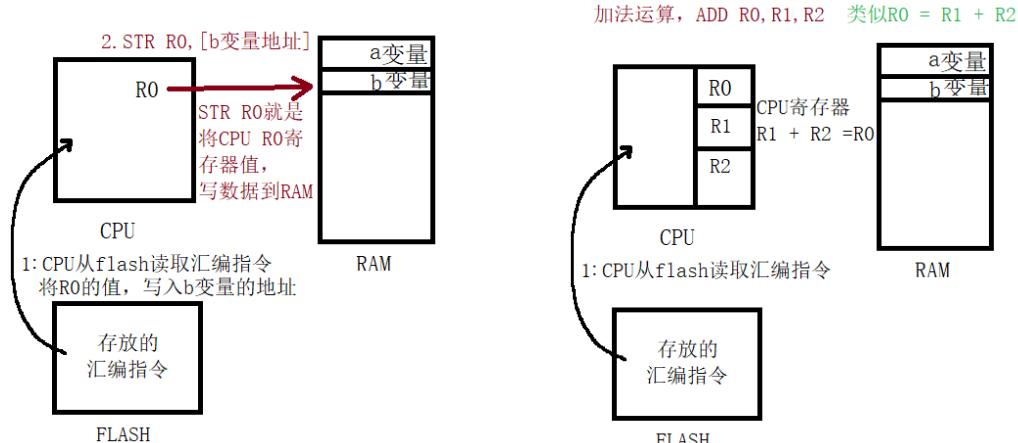


这就是 STM32F103 基本架构

ARM架构

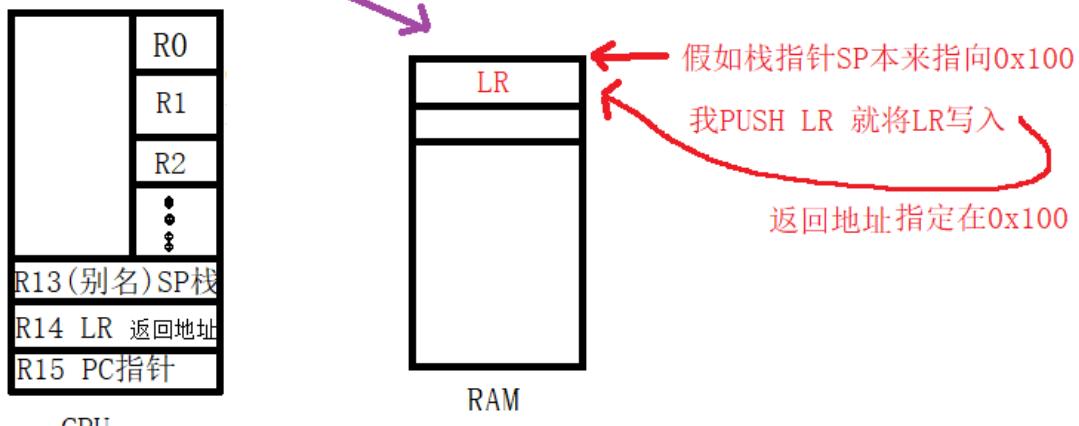


ARM架构



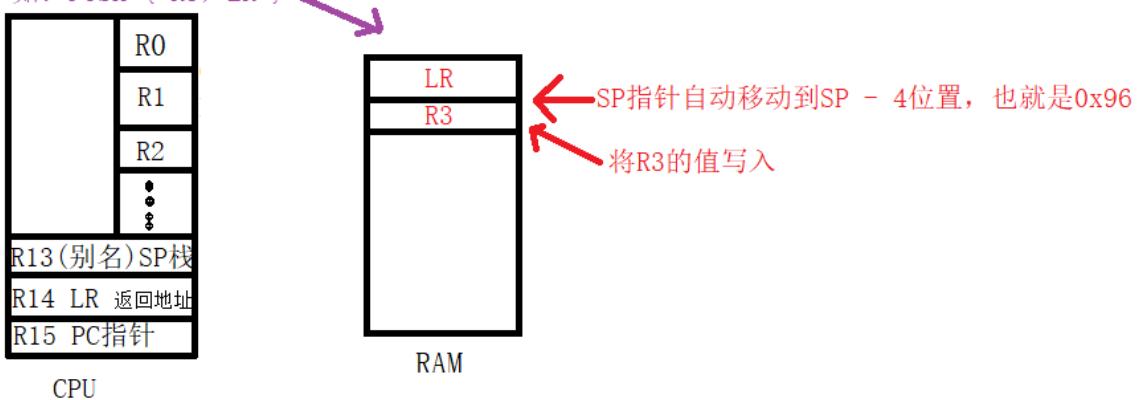
PUSH 指令，本质就是写内存指令

如：PUSH { R3, LR }



CPU

如：PUSH { R3, LR }

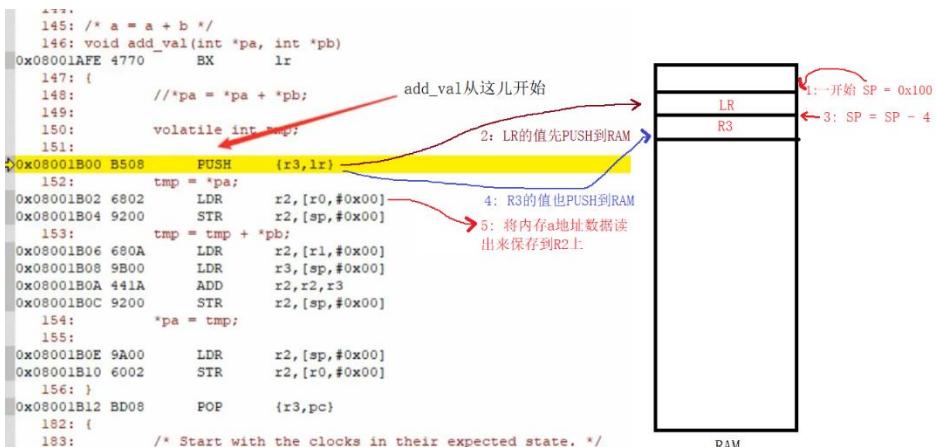


CPU

## 栈的作用

```
//a = a + b
void add_val(int *pa,int *pb)
{
    volatile int tmp;
    tmp = *pa;
    tmp = tmp + *pb;
    *pa = tmp;
}

int main(void)
{
    int a = 1;
    int b = 2;
    add_val(&a,&b);
}
```



```

    ...
145: /* a = a + b */
146: void add_val(int *pa, int *pb)
0x08001AFE 4770      BX      lr
147: {
148:     /*pa = *pa + *pb;
149:
150:     volatile int tmp;
151:

→ 0x08001B00 B508      PUSH    {r3,lr} -
152:     tmp = *pa;
0x08001B02 6802      LDR     r2,[r0,#0x00];
0x08001B04 9200      STR     r2,[sp,#0x00] → 6: 将R2的值保存到SP+0位置
153:     tmp = tmp + *pb;
0x08001B06 680A      LDR     r2,[r1,#0x00] → 7: R2 = [R1+0]地址
0x08001B08 9B00      LDR     r3,[sp,#0x00]
0x08001B0A 441A      ADD     r2,r2,r3 → 8: R3 = [SP + 0 ]位置上的值，就是tmp
0x08001B0C 9200      STR     r2,[sp,#0x00] → 9: R2 = R2 + R3
154:     *pa = tmp;
155:
0x08001B0E 9A00      LDR     r2,[sp,#0x00]
0x08001B10 6002      STR     r2,[r0,#0x00] → 10: 将R2的值保存到SP+0的位置
156: }
0x08001B12 BD08      POP     {r3,pc} → 11: R2 = SP+0位置的值
182: {
183:

```

7. R3取消，改成R2

1:一开始 SP = 0x100

SP+0

LR

R2

SP+0

RAM

13: 返回

```

150:     volatile int tmp;
151:
→ 0x08001B00 B508      PUSH    {r3,lr} -
152:     tmp = *pa;
0x08001B02 6802      LDR     r2,[r0,#0x00] ← 如果R2读取了内存的值突然被中断打断
0x08001B04 9200      STR     r2,[sp,#0x00]
153:     tmp = tmp + *pb;
0x08001B06 680A      LDR     r2,[r1,#0x00] ← 中断位置
0x08001B08 9B00      LDR     r3,[sp,#0x00]
0x08001B0A 441A      ADD     r2,r2,r3
0x08001B0C 9200      STR     r2,[sp,#0x00] ← 导致R2无法写SP
154:     *pa = tmp;
155:
0x08001B0E 9A00      LDR     r2,[sp,#0x00]
0x08001B10 6002      STR     r2,[r0,#0x00] ← 如何保存现场，也就是保存R2的值
156: }
0x08001B12 BD08      POP     {r3,pc}
182: {
183:

```

```

    ...
→ 0x08001B00 B508      PUSH    {r3,lr} -
152:     tmp = *pa;
0x08001B02 6802      LDR     r2,[r0,#0x00] ← 如果R2读取了内存的值突然被中断打断
0x08001B04 9200      STR     r2,[sp,#0x00]
153:     tmp = tmp + *pb;
0x08001B06 680A      LDR     r2,[r1,#0x00] ← 中断位置
0x08001B08 9B00      LDR     r3,[sp,#0x00]
0x08001B0A 441A      ADD     r2,r2,r3
0x08001B0C 9200      STR     r2,[sp,#0x00]
154:     *pa = tmp;
155:
0x08001B0E 9A00      LDR     r2,[sp,#0x00]
0x08001B10 6002      STR     r2,[r0,#0x00]
156: }
0x08001B12 BD08      POP     {r3,pc}
182: {
183:

```

如果是硬件中断，硬件自身就会帮你保存R2寄存器的值到RAM(栈)里面

如果是OS任务切换，那么保存现场就是保存所有的寄存器

## 创建任务的过程

```
 BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint16_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pxCreatedTask ) /*lint !e971*/
{
    TCB_t *pxNewTCB; ←
    BaseType_t xReturn;
}

main() 1, 如果我创建三个任务, 我就拿其中一个任务来说
{
    xTaskCreate(vTask1, "Task 1", 1000, NULL, 0, NULL);
    xTaskCreate(vTask2, "Task 2", 1000, NULL, 0, NULL);
    xTaskCreate(vTask3, "Task 3", 1000, NULL, 2, NULL);
}
```

```
typedef struct tskT
{
    volatile StackType_t
};

#if ( portUSING_MPU_WRAPPERS == 1 )
    xMPU_SETTINGS xMPUSettings; /*< The MPU settings are defined here */
#endif

ListItem_t xStateListItem; /*< The list that the state is added to */
ListItem_t xEventListItem; /*< Used to reference a task from an event */
UBaseType_t uxPriority; ← 这次创建任务的优先级
StackType_t *pxStack; ← /*< 这次创建的任务, 栈的起始地址
char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Description of the task
```

RAM

### 介绍 TCB 结构体内容

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t *pxTopOfStack; 最后一项, 在栈里面保存的位置

#if ( portUSING_MPU_WRAPPERS == 1 )
    xMPU_SETTINGS xMPUSettings; /*< The MPU settings are defined here */
#endif

    ListItem_t xStateListItem; /*< The list that the state is added to */
    ListItem_t xEventListItem; /*< Used to reference a task from an event */
    UBaseType_t uxPriority; ← 这次创建任务的优先级
    StackType_t *pxStack; ← /*< 这次创建的任务, 栈的起始地址
    char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Description of the task
```



没有形参相关的东西

有个疑问? 这个TCB结构体怎么没有体现出传入的函数, 和传入的数据呢?

xTaskCreate(vTask1, "Task 1", 1000, NULL, 0, NULL);



函数对应的形参, 参数

没有函数相关的东西

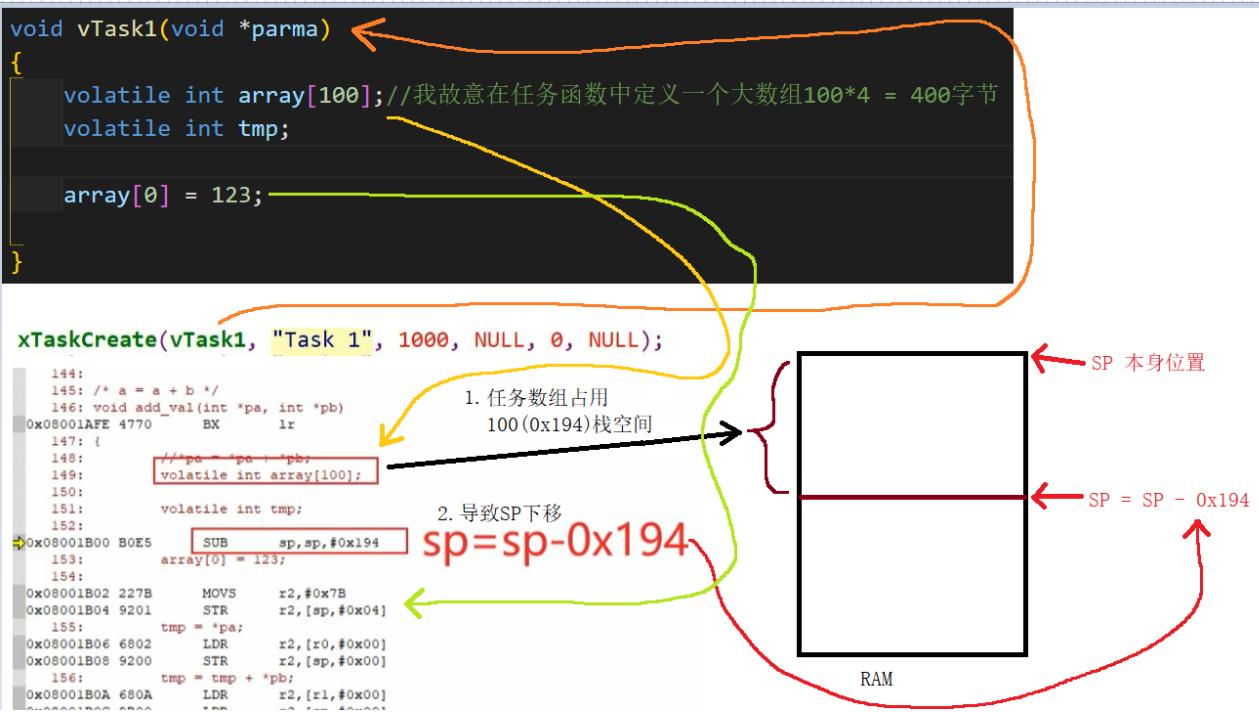
我创建任务的时候, 函数的东西怎么没在  
TCB结构里面提现?

1.任务栈空间从哪儿分配?

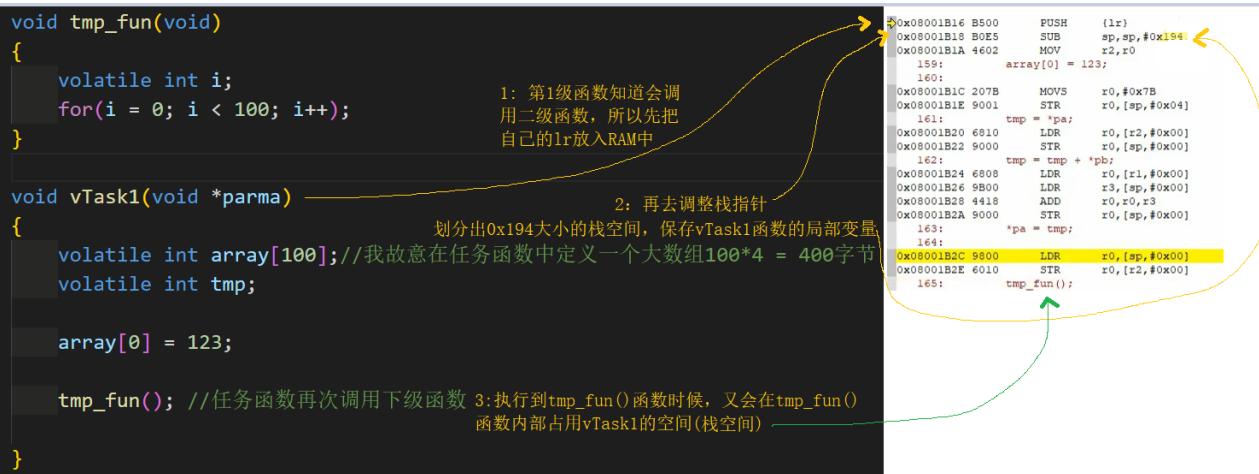
2.一个任务, 它对应的栈空间分配多大合适?

3.函数创建后在哪儿去执行?

第 1 个问题, 要搞清楚任务栈空间去哪儿分配, 分配多大, 我们要搞清楚任务执行的时候, 会使用多大的栈?



所以栈使用多大，和局部变量有关。和函数调用深度有关系(任务调用函数，函数调用函数)



所以这就回答了第 2 个问题，创建任务分配多大的栈，根据你这个任务递归调用了多少级函数来决定。

FreeRTOS，定义的总栈大小是多少呢？(总栈，就是每个任务分配的栈，都要到总栈去划分)

我使用的是 `heap_4.c` 文件

```

101  #if( configAPPLICATION_ALLOCATED_HEAP == 1 )
102  /* The application writer has already defined the array used for the RTOS
103  heap - probably so it can be placed in a special segment or address. */
104  extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
105  #else
106  static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
107  #endif /* configAPPLICATION_ALLOCATED_HEAP */

```

所以总栈大小是个全局的数组 `ucHeap`，至于分配多大，就看 `FreeRTOSconfig` 里面的宏 `TOTAL_HEAP_SIZE` 写多大了。

第 3 个问题，任务函数去哪儿执行的？

## 基本思路

`xTaskCreate(vTask1, "Task 1", 1000, NULL, 0, NULL);`



让PC指针，执行任务函数的地址

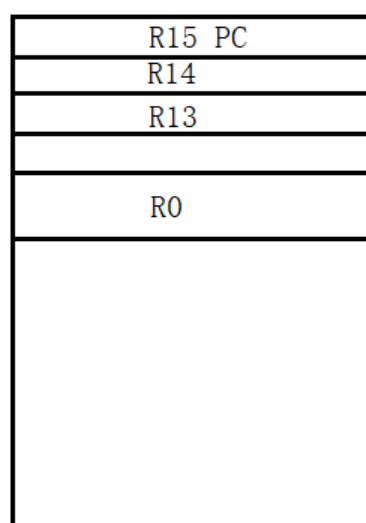


参数保存在R0寄存器上

`xTaskCreate(vTask1, "Task 1", 1000, NULL, 0, NULL);`

1: 在创建任务的时候，我从总栈空间划分 $1000 * 4 = 4096$ 字节的空间出来

2: 任务运行的时候，从栈中拿出R0 ~ R15寄存器值  
然后再CPU中运行，CPU  
就知道函数的起始地  
址，参数大小



3: vTask1函数地址保  
存在R15

4: 参数值，保存在R0

ucHeap 栈数组

## 动态任务创建细节

```

BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
    const char * const pcName,
    const uint16_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pxCreatedTask ) /*lint !e961 MISRA exception as the casts are
{
    TCB_t *pxNewTCB;
    BaseType_t xReturn;
    #else /* portSTACK_GROWTH */
    {
        StackType_t *pxStack;
        /* Allocate space for the stack used by the task being created. */
        pxStack = ( StackType_t * ) pvPortMalloc( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) );
        /*lint !e961 MISRA exception as the casts are
        if( pxStack != NULL )
        {
            /* Allocate space for the TCB. */
            pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
            /*lint !e961 MISRA exception as the casts are
            if( pxNewTCB != NULL )
            {
                /* Store the stack location in the TCB. */
                pxNewTCB->pxStack = pxStack;
            }
            else
            {
                /* The stack cannot be used as the TCB was not created. Free
                it again. */
                vPortFree( pxStack );
            }
        }
        else
        {
            pxNewTCB = NULL;
        }
    #endif /* portSTACK_GROWTH */
}

```

用 malloc 分配栈空间

分配的栈起始  
位置，在底部  
← pxStack

用 malloc 分配 TCB 结构体

新 TCB 的栈，就是  
malloc 才分配的栈

```

if( pxNewTCB != NULL )
{
    #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
    {
        /* Tasks can be created statically or dynamically, so note this
        task was created dynamically in case it is later deleted. */
        pxNewTCB->ucStaticallyAllocated = tskDYNAMICALLY_ALLOCATED_STACK_AND_TCB;
    }
    #endif /* configSUPPORT_STATIC_ALLOCATION */

    prvInitialiseNewTask( pxTaskCode, pcName, ( uint32_t ) usStackDepth, pvParameters, uxPriority, pxCreatedTask, pxNewTCB, NULL );
    prvAddNewTaskToReadyList( pxNewTCB );
    xReturn = pdPASS;
}

```

把新分配的任务控制块，添加进就绪链表

在 `prvAddNewTaskToReadyList(...)` 里面调用

`prvAddTaskToReadyList( pxNewTCB );`

```

#define prvAddTaskToReadyList( pxTCB ) \
    traceMOVED_TASK_TO_READY_STATE( pxTCB ); \
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority ); \
    vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) ); \
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB )

```

把新分配的任务控制块，插入链表尾部

```

vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xStateListItem ) );

```

插入链表数组的尾部

↑  
我根据优先级来决定任务控制块，放入链表数组里面的第几个链表

核心思想：

`PRIILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];` 内核实现定义了一个任务链表数组，这个数组的大小是由最大优先级来配置的

`#define configMAX_PRIORITIES ( 32 )` 我在 `FreeRTOSconfig.h` 配置了 32 个优先级

`pxReadyTasksLists[32]`  
`pxReadyTasksLists[31]`  
`pxReadyTasksLists[30]`  
`.`  
`.`  
`pxReadyTasksLists[0]`

} 所以我的就绪链表有 32 个节点

`pxReadyTasksLists[32]`  
`pxReadyTasksLists[31]`  
`pxReadyTasksLists[30]`  
`.`  
`.`  
`pxReadyTasksLists[0]`

} 除了就绪链表

`pxDelayedTaskList` 还有延时链表，这里只有一个链表

`xPendingReadyList` 也只有一个链表

```

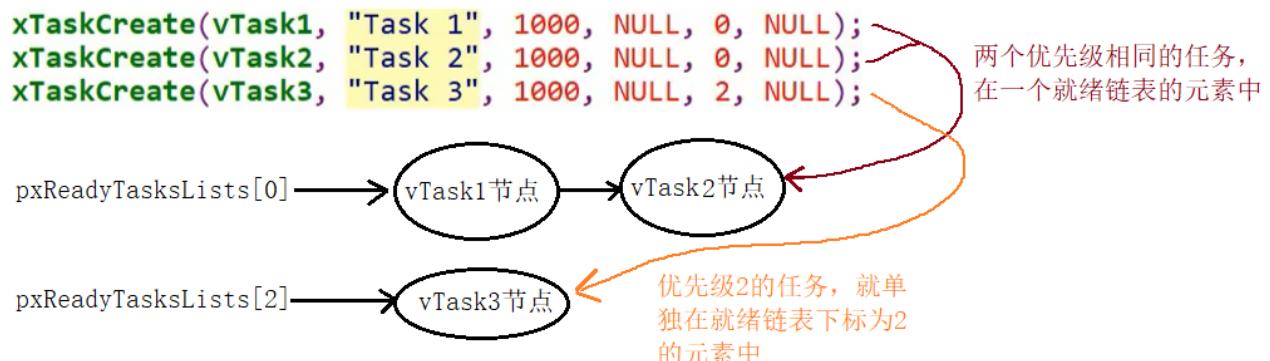
pxReadyTasksLists[32]
pxReadyTasksLists[31]
pxReadyTasksLists[30]
.
.
pxReadyTasksLists[0]

```

就绪链表

注意这个就绪链表只是根据优先级来分配的32个元素，但是每个元素里面还可以放很多个任务节点，相当于一个元素就是一个链表

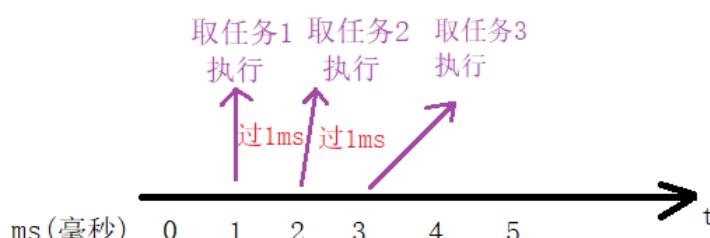
怎么理解呢？



这就是我说的一个就绪表元素，就是一个  
单独的链表

下面讲解，谁来进行就绪链表的调度？

进行调度的函数是一个 TICK 中断，



TICK 每间隔1ms，就会中断一次

任务3优先级为2，是三个任务中优先级最高的，我以任务3为例，来讲解调度

```

xTaskCreate(vTask1, "Task 1", 1000, NULL, 0, NULL);
xTaskCreate(vTask2, "Task 2", 1000, NULL, 0, NULL);
xTaskCreate(vTask3, "Task 3", 1000, NULL, 2, NULL);

```

```

void vTask3( void *pvParameters )
{
    const TickType_t xDelay5ms = pdMS_TO_TICKS( 5UL );

    /* 任务函数的主体一般都是无限循环 */
    for( ;; )
    {
        fFlagIdleTaskrun = 0;
        fFlagTask1run = 0;
        fFlagTask2run = 0;
        fFlagTask3run = 1;

        /* 打印任务的信息 */
        printf("T3\r\n");

        // 如果不休眠的话，其他任务无法得到执行
        vTaskDelay( xDelay5ms );
    }
}

```

调度的关键，就在这delay延时5ms上

```

void vTask3( void *pvParameters )
{
    const TickType_t xDelay5ms = pdMS_TO_TICKS( 5UL );
    /* 任务函数的主体一般都是无限循环 */
    for( ;; )
    {
        flagIdleTaskrun = 0;
        flagTask1run = 0;
        flagTask2run = 0;
        flagTask3run = 1;

        /* 打印任务的信息 */
        printf("T3\r\n");

        // 如果不休眠的话，其他任务无法得到执行
        vTaskDelay( xDelay5ms ); ←
    }
}

```

1:  
pxReadyTasksLists[2] = 任务3  
任务3本来放在就绪表中，然后CPU执行任务3

2:因为三个任务中，任务3优先级最高，所以任务1，和任务2是无法执行的

3: 任务3执行delay之后，任务3就处于休眠状态，这时候，任务1，任务2得到执行

本来任务3在就绪表

pxReadyTasksLists[2] = 任务3

vTaskDelay( xDelay5ms );      pxDelayedTaskList = 任务3    当执行延时之后，任务3离开了就绪表，放入了阻塞链表

这时候调度器再去就绪表里面找，任务3，发现没有任务3节点

这就是 FreeRTOS 任务调度的基本思想，具体源码细节，自己查阅。

## FreeRTOS 移植到 STM32F1

去托管网站下载 freertos 源码 <https://sourceforge.net/projects/freertos/files/FreeRTOS/>

[V9.0.0](#)

2018-08-24

[374](#)

下载 9.0.0 版本

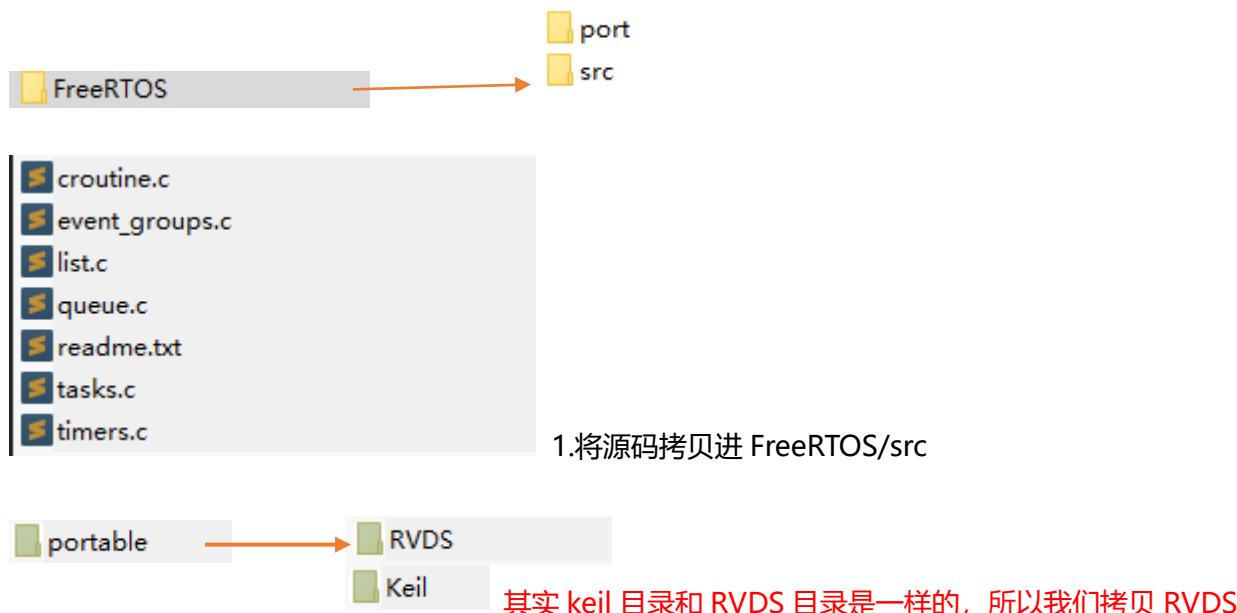
<a href="#">FreeRTOS_V9.0.0a (V9 with MIT).zip</a>	2018-08-24 1.4 MB	<a href="#">6</a>
<a href="#">source-code-for-book-examples.zip</a>	2016-12-18 439.3 kB	<a href="#">294</a>
<a href="#">FreeRTOSv9.0.0.zip</a>	2016-05-25 51.4 MB	<a href="#">55</a>
<a href="#">README</a>	2016-05-25 121.5 kB	<a href="#">1</a>
<a href="#">FreeRTOSv9.0.0.exe</a>	2016-05-25 16.1 MB	<a href="#">18</a>
Totals: 5 Items	69.5 MB	<a href="#">374</a> 下载 FreeRTOSv9.0.0.zip 压缩包



Source 就是源文件

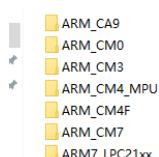
Portable 里面就是与硬件相关的文件

自己创建工程文件，将 STM32F1 的串口和固件库移植完成，再创建 FreeRTOS 工程文件。



2.portable 文件里与 keil 相关的 RVDS 拷贝进自己工程 FreeRTOS/port

FreeRtosTest > FreeRTOS > port > RVDS



这就是 RVDS 内容，不同的内核



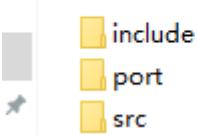
将 MemMang 目录里面的拷贝进 port 目录

工程里面只需要加入 heap\_4.c



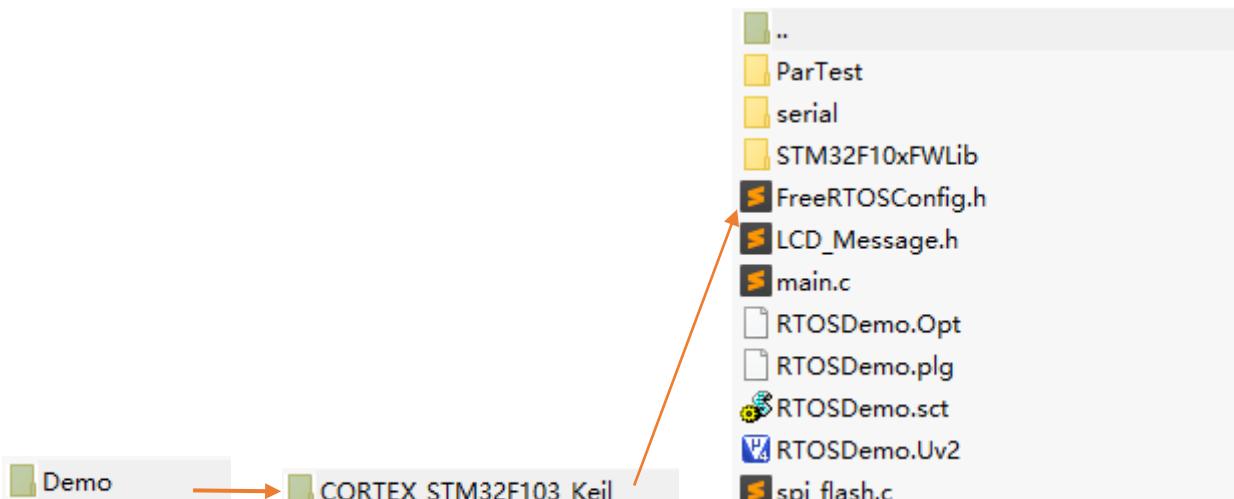
3. 将 include 目录拷贝进 FreeRTOS

FreeRtosTest > FreeRTOS



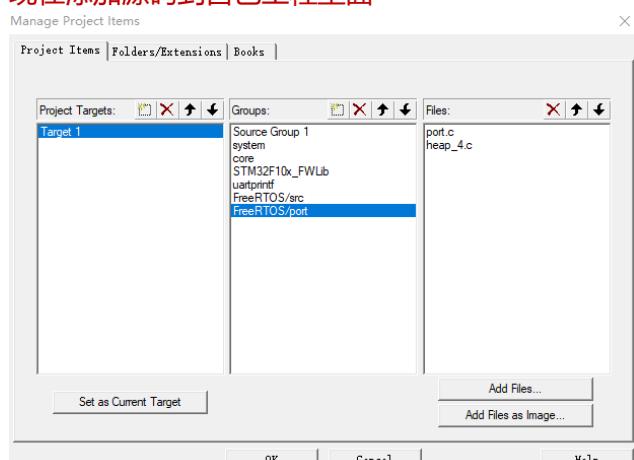
这就是 FreeRTOS 目录下所有拷贝得到的内容。

4. 还要记得将配置文件拷贝到你的根目录下，这个配置文件 FreeRTOSConfig.h 去 demo 目录里面找

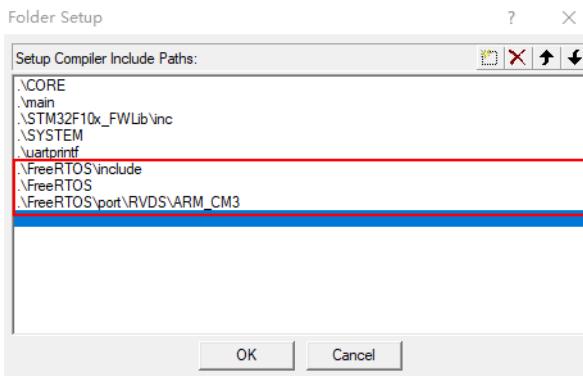


将 FreeRTOSConfig.h 拷贝到自己工程根目录下，或者 FreeRTOS 目录下。

现在添加源码到自己工程里面



port 导入 ARM\_CM3 的文件和 MemMang 的文件



加入进头文件后进行编译。

```
linking...
Program Size: Code=5688 RO-data=336 RW-data=44 ZI-data=1028
FromELF: creating hex file...
".\out\FreeRtosTest.axf" - 0 Error(s), 1 Warning(s).
```

第一次编译完全通过，但是实际是有问题的，很多配置没有打开和移植，下面进行配置和移植。

## 详解 FreeRTOSConfig.h 文件

我们需要明确一个问题，**FreeRTOSConfig.h** 是一个用户级别的文件，**不属于内核文件**。每个用户可以有不同的 **FreeRTOSConfig.h**。所以我们在 **FreeRTOSConfig.h** 看到有些宏没有，并不是系统本身没有。其绝大多数配置选项都体现在 **FreeRTOS.h**（注意是 **FreeRTOS.h** 不是 **FreeRTOSConfig.h**）中。

```
#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
```

**#define configUSE\_PREEMPTION 1 //支持抢占式调度，写 1 支持抢占调度**

**#define configUSE\_TIME\_SLICING 1 //支持时间片功能，这个宏其实是在 FreeRTOS.h 里面，而不是在 FreeRTOSConfig.h 里。我在 FreeRTOSConfig.h 里定义了之后，也就等同于 FreeRTOS.h 使能时间片功能**

**#define configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 1 //在执行下一个任务的时候有两种方法可以选择，我选择特殊方法**

### 通用方法

1. configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 为 0 取消特殊方法使用通用方法，或者硬件本身不支持这种特殊方法
2. 可以用于所有 FreeRTOS 支持的硬件
3. 完全用 C 实现，效率略低于特殊方法
4. 不强制要求限制最大可用优先级数目

### 特殊方法

1. 必须将 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 1 特殊方法
2. 必须将 configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 设置为 1。
3. 比通用方法更高效
4. 一般强制限定最大可用优先级数目为 32

**#define configUSE\_TICKLESS\_IDLE 0 //低功耗模式**

置 1：使能低功耗 tickless 模式；置 0：保持系统节拍（tick）中断一直运行

如果保持低功耗模式会导致下载程序出问题，因为下载程序时说不准芯片进入低功耗了。当然直接擦除 flash 再次下载可以解决这个 IDLE 下载问题。

```
#define configCPU_CLOCK_HZ (( unsigned long ) 72000000) //系统时钟我们定义 72Mhz

#define configTICK_RATE_HZ (( TickType_t ) 1000) //中断频率, 1 秒钟中断次数
RTOS 系统节拍中断的频率。即一秒中断的次数，每次中断 RTOS 都会进行任务调度，这里是 1000，也就是 1 秒钟中断 1000 次。也就是每 1 毫秒中断一次。

#define configMAX_PRIORITIES ( 32 ) //最大可用优先级，因为前面使用的特殊方法，所以只能设定到 32

#define configMINIMAL_STACK_SIZE (( unsigned short ) 128) //空闲任务栈，定义为 128 字
如果空闲任务没有做什么事情，可以将空闲任务设置小点，比如 64,32 节省内存。

#define configMAX_TASK_NAME_LEN ( 16 ) //任务名字长度 16 字节，也可自定义长度。

#define configUSE_16_BIT_TICKS 0 //系统节拍最大计数，1 表示 16 位，0 表示 32 位
我们 systick 滴答定时器是 32 位的，所以最大计数到 0xffffffff，所以我们填 0。

#define configIDLE_SHOULD_YIELD 1 //填 1，空闲任务和其它任务同时执行时，空闲任务放弃优先权，让其它任务先执行。填 0，空闲任务不放弃当前执行机会。

#define configUSE_QUEUE_SETS 0 //是否启用消息队列，填 0 不启用，填 1 启用。

#define configUSE_TASK_NOTIFICATIONS 1 //任务通知，默认开启

#define configUSE_MUTEXES 0 //互斥量，填 0 不使用，填 1 使用。

#define configUSE_RECURSIVE_MUTEXES 0 //递归互斥信号量，填 0 不使用，填 1 使用。

#define configUSE_COUNTING_SEMAPHORES 0 //填 0 不使用计数信号量

#define configQUEUE_REGISTRY_SIZE 10 //设置可以注册信号量和消息队列的个数，这里注册 10 个

#define configSUPPORT_DYNAMIC_ALLOCATION 1 //设置为 1 支持动态内存分配
#define configSUPPORT_STATIC_ALLOCATION 0 //设置为 0 取消静态内存分配。
一般使用动态内存分配，静态内存分配要自己手动写死任务栈大小，前面重新写 FreeRTOS 内核章节讲过。

#define configTOTAL_HEAP_SIZE (( size_t )( 36 * 1024 )) //系统所有总的堆大小 36K RAM

#define configUSE_IDLE_HOOK 0

#define configUSE_MALLOC_FAILED_HOOK 0 //使用内存申请失败钩子函数，这种就是如果内存分配失败，你需要执行什么相关的程序，可以放入这个钩子函数中执行。

#define configCHECK_FOR_STACK_OVERFLOW 0 //堆栈检测钩子函数
在测试项目的时候可以将其置 1，实现堆栈检测钩子函数，看看有没有堆栈溢出情况，成品出货可以关闭该功能。
```

```

#define xPortPendSVHandler PendSV_Handler //任务切换宏必须加入，会去调用 port.c 下面的汇编，该汇编有切换
STM32F1 内部寄存器的功能，所以就不用我们再去移植汇编了
#define vPortSVCHandler SVC_Handler //任务切换宏必须加入，会去调用汇编，同上功能
这两个任务切换宏很重要
同时这两个宏 xPortPendSVHandler PendSV_Handler, vPortSVCHandler SVC_Handler 会和 STM32 ....it.c 里面的 SVC 和 PendSV 中断函数重
复定义，所以去屏蔽....it.c 里面的 void SVC_Handler(void)和 void PendSV_Handler(void)
//void SVC_Handler(void)
//{
//}

void DebugMon_Handler(void)
{
}

//void PendSV_Handler(void)
//{
//}

```

这样你的任务切换功能就算实现了。

在 systick 中断中调用 FreeRTOS 的中断处理函数

```

+-- stm32f10x_it.c
void xPortSysTickHandler( void ); //这是port.c里面的函数，需要外部声明下，再调用

void SysTick_Handler(void)
{
    xPortSysTickHandler(); //调用FreeRTOS中断函数 ←
}

```

### FreeRTOS/port/port.c

```

void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt priority, so when this interrupt
     * executes all interrupts must be unmasked. There is therefore no need to
     * save and then restore the interrupt mask value as its value is already
     * known - therefore the slightly faster vPortRaiseBASEPRI() function is used
     * in place of portSET_INTERRUPT_MASK_FROM_ISR(). */
    vPortRaiseBASEPRI();

    /* Increment the RTOS tick. */
    if( xTaskIncrementTick() != pdFALSE )
    {
        /* A context switch is required. Context switching is performed in
         * the PendSV interrupt. Pend the PendSV interrupt. */
        portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
    }
}
vPortClearBASEPRIFromISR();
}

```

编译通过。

但是还是需要再做点修改，主要修改 stm32f10x\_it.c

```

#include "FreeRTOS.h"
#include "task.h"
void SysTick_Handler(void)
{
    #if(INCLUDE_xTaskGetSchedulerState == 1) //当调度器启动后 我们才能执行这个函数
    {
        #endif
        xPortSysTickHandler(); //调用 FreeRTOS 中断函数
        #if(INCLUDE_xTaskGetSchedulerState == 1)
    }
    #endif
}

```

改成这样最好，编译通过。

FreeRTOSconfig.h 里面我自定义的初步宏，这些基本宏定义了才可以实现任务切换和一些基本功能

```
#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           ( 32 ) //最大优先级
#define configMINIMAL_STACK_SIZE        (( unsigned short ) 128) //空闲任务栈 128 字
#define configTOTAL_HEAP_SIZE           (( size_t )( 17 * 1024 ))
#define configMAX_TASK_NAME_LEN         ( 16 ) //任务名长度 16 字
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT TICKS           0 //系统最大节拍数 32 位
#define configIDLE_SHOULD_YIELD         1 //空闲任务放弃优先权

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES          0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet         1
#define INCLUDE_uxTaskPriorityGet        1
#define INCLUDE_vTaskDelete              1
#define INCLUDE_vTaskCleanUpResources    0
#define INCLUDE_vTaskSuspend             1
#define INCLUDE_vTaskDelayUntil          1
#define INCLUDE_vTaskDelay               1

#define configKERNEL_INTERRUPT_PRIORITY      255

#define configMAX_SYSCALL_INTERRUPT_PRIORITY 191 /* equivalent to 0xb0, or priority 11. */

/* This is the value being used as per the ST library which permits 16
priority values, 0 to 15. This must correspond to the
configKERNEL_INTERRUPT_PRIORITY setting. Here 15 corresponds to the lowest
NVIC value of 255. */
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY     15
#define configSUPPORT_STATIC_ALLOCATION 1 //我先使用静态方式创建任务，所以要打开静态分配内存功能
//#define configSUPPORT_DYNAMIC_ALLOCATION 1

#define configUSE_TIME_SLICING 1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

#define configUSE_QUEUE_SETS 0 //暂不启用消息队列

#define configUSE_TASK_NOTIFICATIONS 1 //任务通知开启

#define configUSE_MUTEXES 0 //互斥量暂不使用

#define configUSE_RECURSIVE_MUTEXES 0

#define configUSE_COUNTING_SEMAPHORES 0 //计数信号量暂不使用

#define configQUEUE_REGISTRY_SIZE 10 //信号量消息队列个数可以注册 10 个

#define xPortPendSVHandler PendSV_Handler //任务切换函数必须加入，会去调用汇编
#define vPortSVCHandler SVC_Handler //任务切换函数必须加入，会去调用汇编
```

因为我们使用的是静态方式创建任务，所以需要实现空闲任务的堆栈和空闲任务控制块

```
/* 空闲任务堆栈 */
static StackType_t Idle_Task_Stack[128];
/* 空闲任务控制块 */
static StaticTask_t Idle_Task_TCB;

void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer,
                                    StackType_t **ppxIdleTaskStackBuffer,
                                    uint32_t *pulIdleTaskStackSize )
{
    *ppxIdleTaskTCBBuffer=&Idle_Task_TCB; /* 任务控制块内存 */
    *ppxIdleTaskStackBuffer=Idle_Task_Stack; /* 任务堆栈内存 */
    *pulIdleTaskStackSize=configMINIMAL_STACK_SIZE; /* 任务堆栈大小 */
}
```

将这个空闲任务控制块放在 main 文件下，或者其它文件都可以，系统会自动去调用

在 config 文件定义 #define configTIMER\_TASK\_STACK\_DEPTH 128 //定时器任务堆栈大小

```
/* 定时器任务堆栈 */
static StackType_t Timer_Task_Stack[128];
/* 定时器任务控制块 */
static StaticTask_t Timer_Task_TCB;
void vApplicationGetTimerTaskMemory(StaticTask_t **ppxTimerTaskTCBBuffer,
                                    StackType_t
                                    **ppxTimerTaskStackBuffer,
                                    uint32_t
                                    *pulTimerTaskStackSize)
{
    *ppxTimerTaskTCBBuffer=&Timer_Task_TCB; /* 任务控制块内存 */
    *ppxTimerTaskStackBuffer=Timer_Task_Stack; /* 任务堆栈内存 */
    *pulTimerTaskStackSize=configTIMER_TASK_STACK_DEPTH; /* 任务堆栈大小 */
}
```

在 main 文件或者其它文件下实现定时器任务控制函数，系统会自动去调用

## 创建静态任务

```
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 ) //静态任务创建函数
```

pxTaskCode: 填入任务函数的名称，也就是函数地址

\* const pcName: 传入任务名称字符串

ulStackDepth: 任务堆栈大小，必须定义的任务栈数组大小一样

pvParameters: 外部参数传递给任务内部使用

uxPriority: 任务优先级

pxTaskBuffer: 传入定义的任务堆栈数组

pxTaskBuffer: 传入定义的任务控制块

```
#include "stm32f10x.h"
#include "sysclock.h"
#include "uartprintf.h"
#include "stdio.h"

#include "FreeRTOS.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件
#include "task.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件
```

```
//定义任务栈
StackType_t Task1Stack[128];
StackType_t Task2Stack[128];
```

```
//定义任务控制块
static StaticTask_t Task1TCB;
static StaticTask_t Task2TCB;
```

```
//定义任务句柄
static TaskHandle_t Task1_Handle;
static TaskHandle_t Task2_Handle;
```

```
void Task1(void)
{
    while(1)
    {
        printf("Task1....\r\n");
        vTaskDelay(500);
    }
}
```

```
void Task2(void)
{
    while(1)
    {
        printf("Task2....\r\n");
        vTaskDelay(500);
    }
}
```

```

int main(void)
{
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); // STM32 中断优先级分组为 4, 即 4bit 都用来表示抢占优先级, 范围为: 0~15
    // 优先级分组只需要分组一次即可, 以后如果有其他的任务需要用到中断,
    // 都统一用这个优先级分组, 千万不要再分组, 切忌。

    printf("xxxxzzzz\r\n");

    Task1_Handle = xTaskCreateStatic((TaskFunction_t)Task1,
        (const char *)"Task1",
        (uint32_t)128,
        (void*)NULL,
        (UBaseType_t)4,
        (StackType_t*)Task1Stack,
        (StaticTask_t*)&Task1TCB);

    Task2_Handle = xTaskCreateStatic((TaskFunction_t)Task2,
        (const char *)"Task2",
        (uint32_t)128,
        (void*)NULL,
        (UBaseType_t)4,
        (StackType_t*)Task2Stack,
        (StaticTask_t*)&Task2TCB);

    if ((NULL != Task1_Handle) && (NULL != Task2_Handle)) // 如果 Task1 任务和 Task2 任务创建成功就执行启动任务
    {
        vTaskStartScheduler(); /* 启动任务, 开启调度 */
    }
    else
    {
        printf("task creat failed..\r\n");
    }
    while(1)
    {
        delay_ms(500);
    }
    return 0;
}

```

Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....

任务成功同时执行

## 动态创建任务

比如有些任务只是在某时间段执行一次就不再执行了，这种就可以用动态创建任务方法。

FreeRTOS 堆空间大小是专门定义了一个全局数组来当做堆使用的。

在 FreeRTOSconfig.h 中我们可以定义堆大小

```
#define configTOTAL_HEAP_SIZE ((size_t)(36 * 1024)) //系统总共的堆大小36K
```

```
Program Size: Code=11092 RO-data=336 RW-data=140 ZI-data=40980  
FromELF: creating hex file...
```

我们定义 36K 堆大小，我们编译后 SRAM 就被占用了 40K 左右，如果我只定义 1024 字节堆大小，我们的 SRAM 编译出来就差不多 5K，所以这个堆大小你可以自己定义。

```
#define configSUPPORT_STATIC_ALLOCATION 1  
#define configSUPPORT_DYNAMIC_ALLOCATION 1
```

动态分配内存置 1，这样静态动态都可以使用了。

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

```
//定义任务栈  
StackType_t Task1Stack[128];  
StackType_t Task2Stack[128];
```

```
//定义任务控制块  
static StaticTask_t Task1TCB;  
static StaticTask_t Task2TCB;
```

这里要把静态 StaticTask\_t  
改成 TaskHandle\_t

```
//定义任务句柄  
static TaskHandle_t Task1_Handle;  
static TaskHandle_t Task2_Handle;
```

取消静态任务需要使用的任务栈数组和任务句柄。

```

#include "stm32f10x.h"
#include "sysclock.h"
#include "uartprintf.h"
#include "stdio.h"
#include "FreeRTOS.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件
#include "task.h" //使用 FreeRTOS 里面的函数一定要包含这个有文件

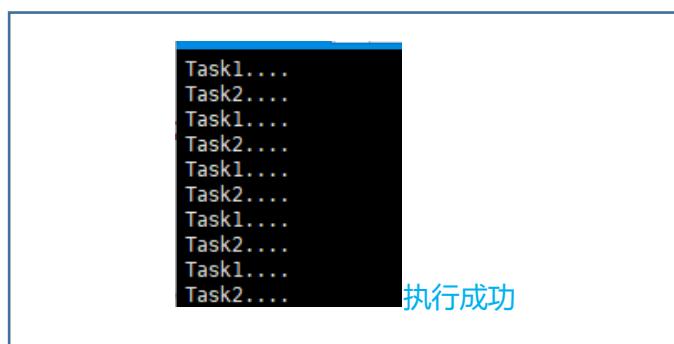
//定义任务控制块
static TaskHandle_t Task1TCB;
static TaskHandle_t Task2TCB;

void Task1(void)
{
    while(1)
    {
        printf("Task1....\r\n");
        vTaskDelay(500);
    }
}

void Task2(void)
{
    while(1)
    {
        printf("Task2....\r\n");
        vTaskDelay(500);
    }
}

int main(void)
{
    BaseType_t xReturn1,xReturn2;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //STM32 中断优先级分组为 4, 即 4bit 都用来表示抢占优先级, 范围为: 0~15
                                                    //优先级分组只需要分组一次即可, 以后如果有其他的任务需要用到中断,
                                                    //都统一用这个优先级分组, 千万不要再分组, 切忌。
    printf("xxxxxxxx\r\n");
    xReturn1 = xTaskCreate((TaskFunction_t)Task1, //任务函数地址
                          (const char*)"Task1",           //任务名
                          (uint32_t)128,                 //任务栈大小
                          (void*)NULL,                  //向函数传参
                          (UBaseType_t)4,                //优先级
                          (TaskHandle_t*)&Task1TCB); //任务控制块
    xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                          (const char*)"Task2",
                          (uint32_t)128,
                          (void*)NULL,
                          (UBaseType_t)4,
                          (TaskHandle_t*)&Task2TCB);
    if((xReturn1 == pdPASS) && (xReturn2 == pdPASS)) //判断任务 1 和任务 2 是否 == pdPASS 如果等于证明任务创建成功
    {
        vTaskStartScheduler(); /* 启动任务, 开启调度 */
    }
    else
    {
        printf("task creat failed..\r\n");
    }
    while(1)
    {
        delay_ms(500);
    }
    return 0;
}

```



# 杀死任务

```
void vTaskDelete( TaskHandle_t xTaskToDelete ) //杀死任务函数
```

## 无法杀死任务的原因

```
int main(void)
{
    BaseType_t xReturn1,xReturn2;
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);

    printf("xxxxzzzz\\r\\n");

    xReturn1 = xTaskCreate((TaskFunction_t)Task1, //任务函数地址
                          (const char*)"Task1",           //任务名
                          (uint32_t)128,                 //任务栈大小
                          (void*)NULL,                  //向函数传参
                          (UBaseType_t)4,                //优先级
                          (TaskHandle_t*)&Task1TCB); //任务控制块

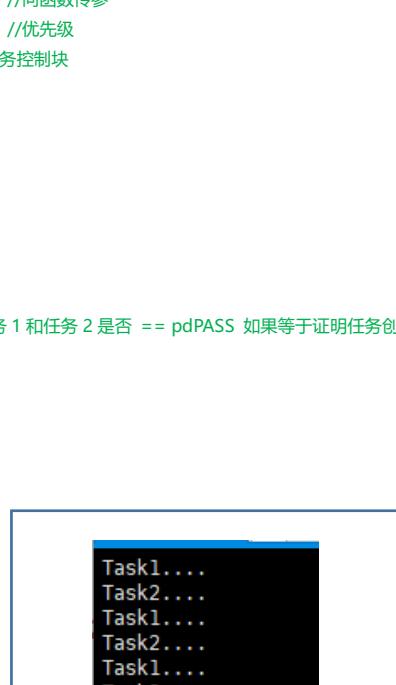
    xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                          (const char*)"Task1",
                          (uint32_t)128,
                          (void*)NULL,
                          (UBaseType_t)4,
                          (TaskHandle_t*)&Task2TCB);

    if((xReturn1 == pdPASS) && (xReturn2 == pdPASS)) //判断任务 1 和任务 2 是否 == pdPASS 如果等于证明任务创建成功
    {
        vTaskStartScheduler(); /* 启动任务，开启调度 */
    }
    else
    {
        printf("task creat failed..\\r\\n");
    }

    vTaskDelay(1000);
    vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块

    printf("vTaskDelete..\\r\\n");

    while(1)
    {
        delay_ms(500);
    }
    return 0;
}
```



Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....  
Task1....  
Task2....  
任务 2 还是

执行杀死任务 2 之后，任务 2 还在继续执行？这是为什么呢？

```
if(xReturn1 == pdPASS) && (xReturn2 == pdPASS)
{
    vTaskStartScheduler(); /* 启动任务，开启调度 */
}
else
{
    printf("task creat failed..\r\n");
}
vTaskDelay(1000);
vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块
printf("vTaskDelete..\r\n");
while(1)
{
    delay_ms(500);
}
return 0;
```

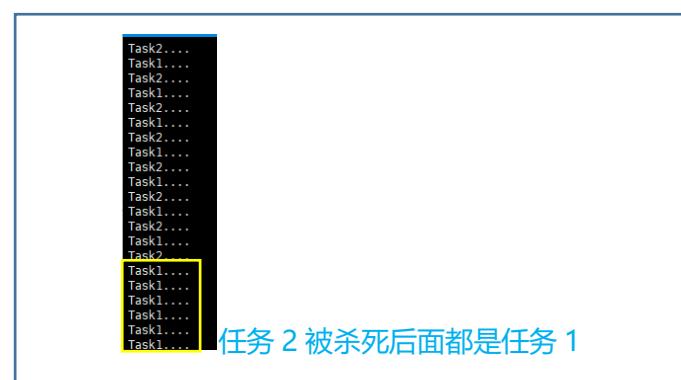


当任务启动之后 vTaskStartScheduler(), 主函数 main 就不会再向下执行了，两个任务把优先级占用完了，这样 vTaskDelete(Task2TCB)也就执行不到

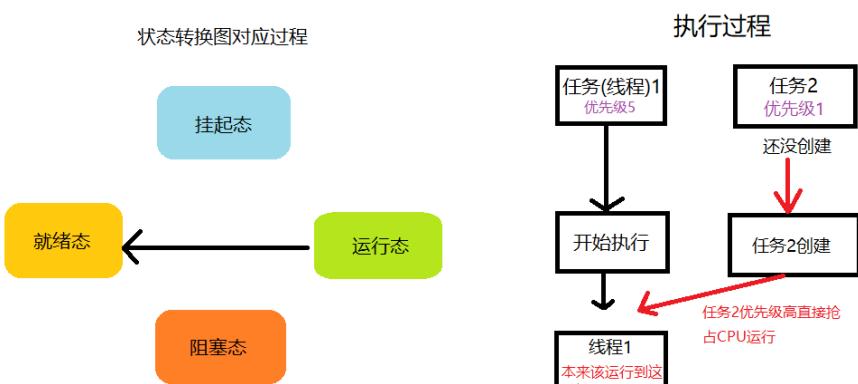
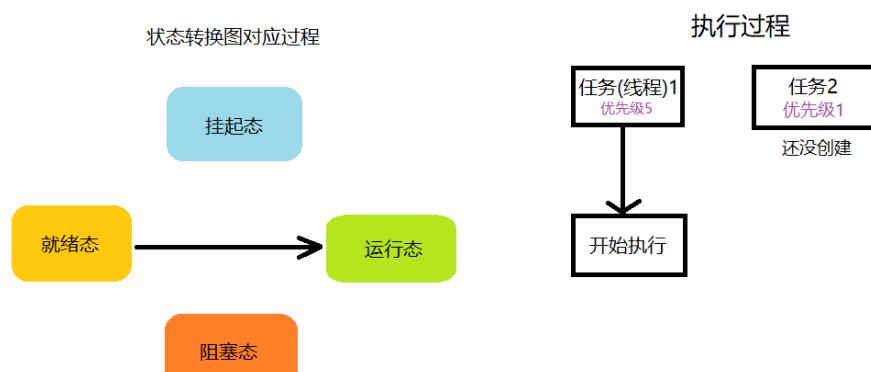
现在唯一的方法就是把任务 1 用来看死任务 2 的方法。

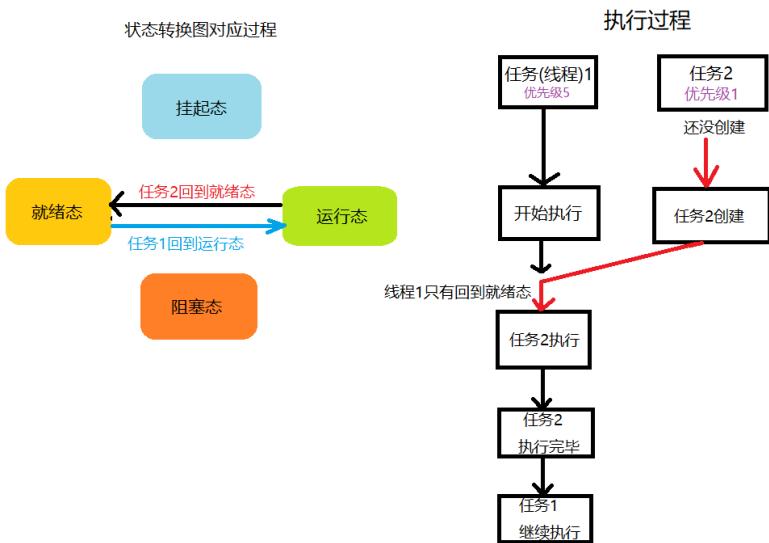
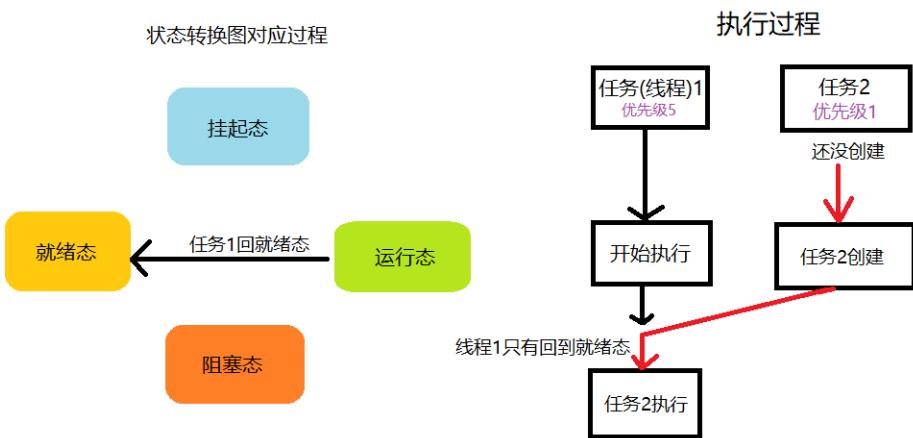
```
//定义任务控制块  
static TaskHandle_t Task1TCB;  
static TaskHandle_t Task2TCB;  
  
void Task2(void); //任务 2 声明  
  
void Task1(void)  
{  
    int i = 0;  
    while(1)  
    {  
        i++;  
        if(i == 10)  
            vTaskDelete(Task2TCB); //杀死任务，传入要杀死任务的任务控制块  
  
        printf("Task1....\r\n");  
        vTaskDelay(500);  
    }  
}  
  
void Task2(void)  
{  
    while(1)  
    {  
        printf("Task2....\r\n");  
        vTaskDelay(500);  
    }  
}
```

任务 1 执行 10 次之后就  
杀死任务 2



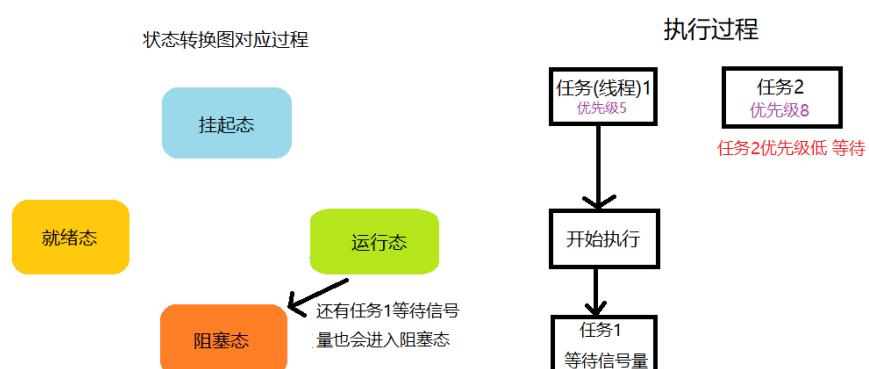
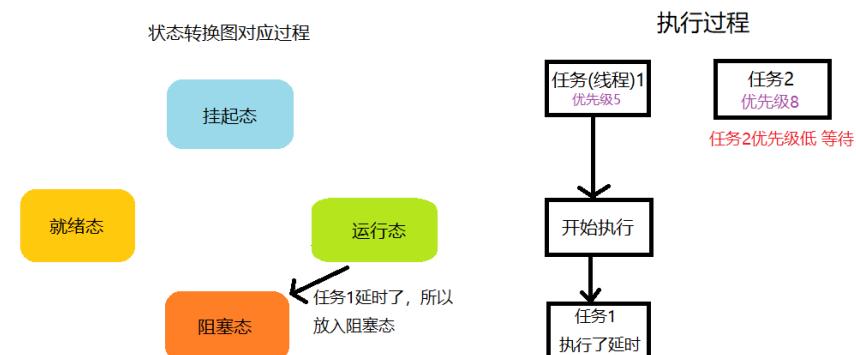
## 任务状态转换表，挂起，阻塞，就绪，运行的含义



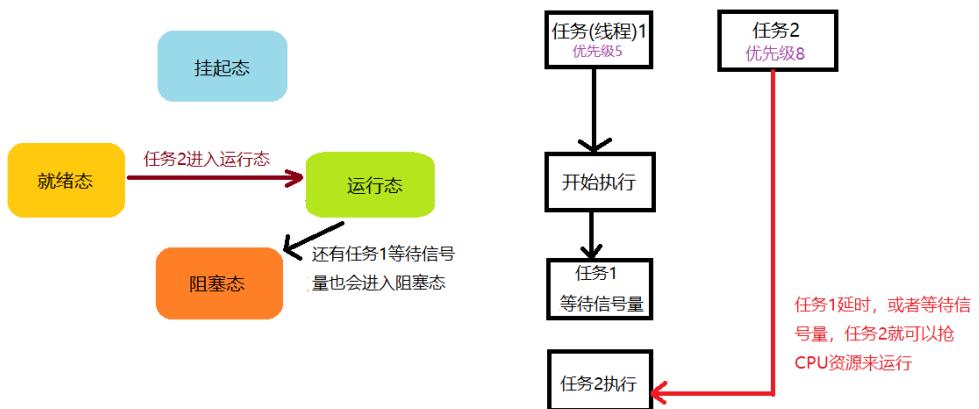


这是在常规情况下有 CPU 安排调度的过程

下面是其它情况会出现的状态

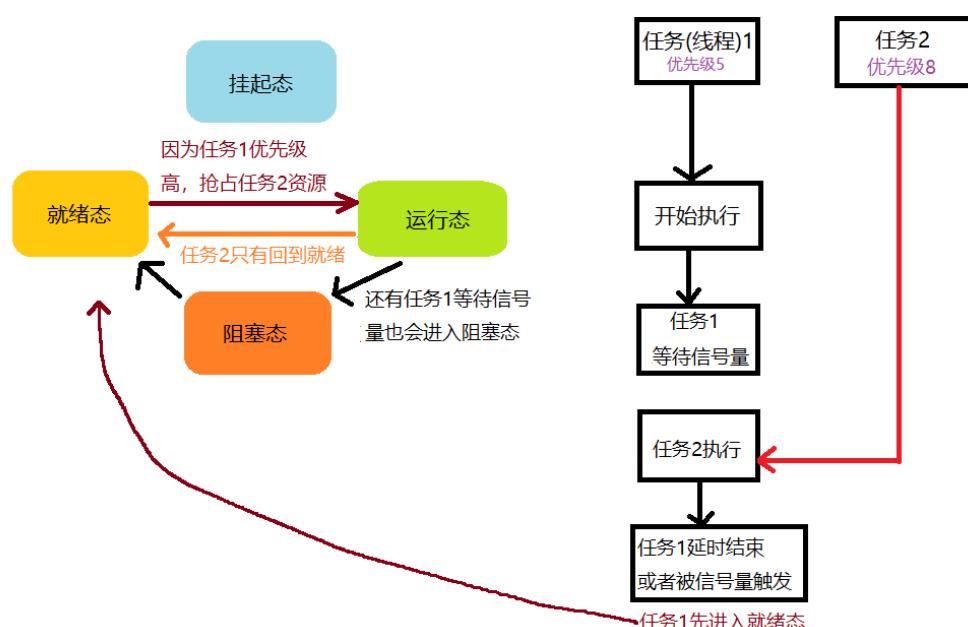


状态转换图对应过程



等待任务1延时结束，或者被其它中断发过来的信号量触发，那么任务1会进入就绪态。

状态转换图对应过程



任务挂起(挂起态)，并不是其它任务造成的，是任务执行了某个函数，自觉自愿挂起的。

挂起的好处是该任务不管优先级多高，都不能抢占其它低优先级任务的CPU进行运行。

状态转换图对应过程

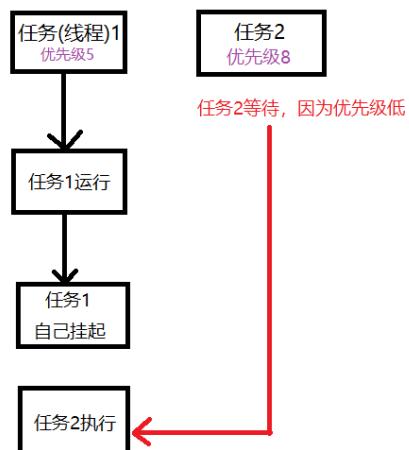


任务1挂起后任务2就可以执行了。

状态转换图对应过程



执行过程



这种任务挂起的应用场景，比如按键被按下，执行该任务挂起。

```

static TaskHandle_t LED_Task_Handle = NULL; /* LED 任务句柄 */

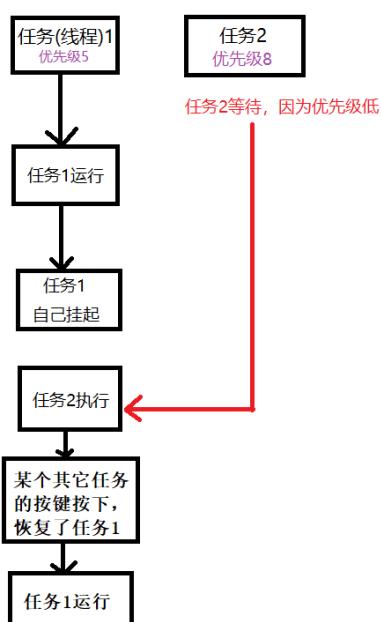
static void KEY_Task(void* parameter)
{
    while (1) {
        if (Key_Scan(KEY1_GPIO_PORT, KEY1_GPIO_PIN) == KEY_ON) {
            /* K1 被按下 */
            printf("挂起 LED 任务! \n");
            vTaskSuspend(LED_Task_Handle); /* 挂起 LED 任务 */
        }
        vTaskDelay(20); /* 延时 20 个 tick */
    }
}
  
```

所以在这种突发事件的情况下，不管是任务延时，还是任务信号量，都无法让任务挂起，只有任务收到突发事件，自己将自己挂起，把 CPU 使用权交出来给其它任务使用，这就是一个应用场景。

状态转换图对应过程



执行过程



```

static TaskHandle_t LED_Task_Handle = NULL; /* LED 任务句柄 */

static void KEY_Task(void* parameter)
{
    while (1) {
        if (Key_Scan(KEY2 GPIO PORT, KEY2 GPIO PIN) == KEY ON) {
            /* K2 被按下 */
            printf("恢复 LED 任务! \n");
            vTaskResume(LED_Task_Handle); /* 恢复 LED 任务! */
        }
        vTaskDelay(20); /* 延时 20 个 tick */
    }
}

```

这就是恢复挂起任务的应用场景，所以如果想让其它任务来，解除任务 1 的挂起状态，那么任务 1 的任务句柄最好用全局变量。

void vTaskSuspend( TaskHandle\_t xTaskToSuspend ) //挂起任务  
xTaskToSuspend: 填入要挂起任务的任务控制块。

void vTaskResume( TaskHandle\_t xTaskToResume )//恢复任务  
xTaskToResume: 填入要恢复任务的任务控制块。

```

//定义任务控制块
static TaskHandle_t Task1TCB;
static TaskHandle_t Task2TCB;

```

void Task2(void); //任务 2 声明

```

void Task1(void)
{
    int i = 0;
    while(1)
    {
        i++;
        if(i == 10)
            vTaskSuspend(Task2TCB); //挂起任务 2，任务 1 运行 10 次之后主动挂起任务 2
        else if(i == 20)
            vTaskResume(Task2TCB); //恢复任务 2
        else
        {
        }
        printf("Task1...\r\n");
        vTaskDelay(500);
    }
}

```

```

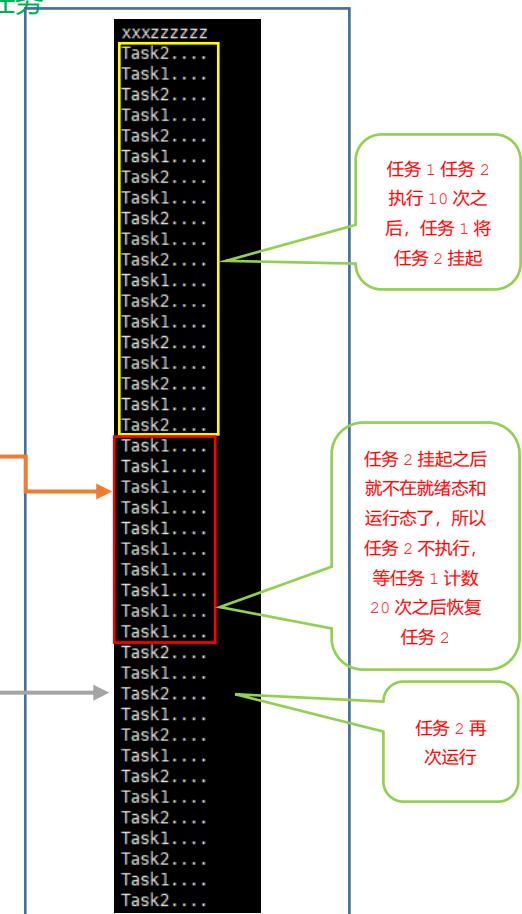
void Task2(void)
{
    while(1)
    {
        printf("Task2...\r\n");
        vTaskDelay(500);
    }
}

```

```

int main(void)
{
    创建任务 1，任务 2...
}

```



TickCount\_t xTaskGetTickCount( void ) //获取系统开始到现在执行了多少时间，以 1 毫秒为单位  
 TickType\_t : uint32\_t 类型，获得毫秒数

```
void Task1(void)
{
    while(1)
    {
        printf("time1 = %d\r\n",xTaskGetTickCount()); //获取线程 1 执行了多少 ms
        printf("Task1....\r\n");
        vTaskDelay(500);
    }
}

void Task2(void)
{
    while(1)
    {
        printf("time2 = %d\r\n",xTaskGetTickCount()); //获取线程 2 执行了多少 ms
        printf("Task2....\r\n");
        vTaskDelay(1000);
    }
}
```

```
Task1....  

time1 = 504  

Task1....  

time2 = 1002  

Task2....  

time1 = 1006  

Task1....  

time1 = 1508  

Task1....  

time2 = 2004  

Task2....  

time1 = 2010  

Task1....  

time1 = 2512  

Task1....  

time2 = 3006  

Task2....  

time1 = 3014  

Task1....  

time1 = 3516  

Task1....  

time2 = 4008  

Task2....  

time1 = 4018  

Task1....  

time1 = 4520  

Task1....  

time2 = 5010  

Task2....  

time1 = 5022  

Task1....  

time1 = 5524
```

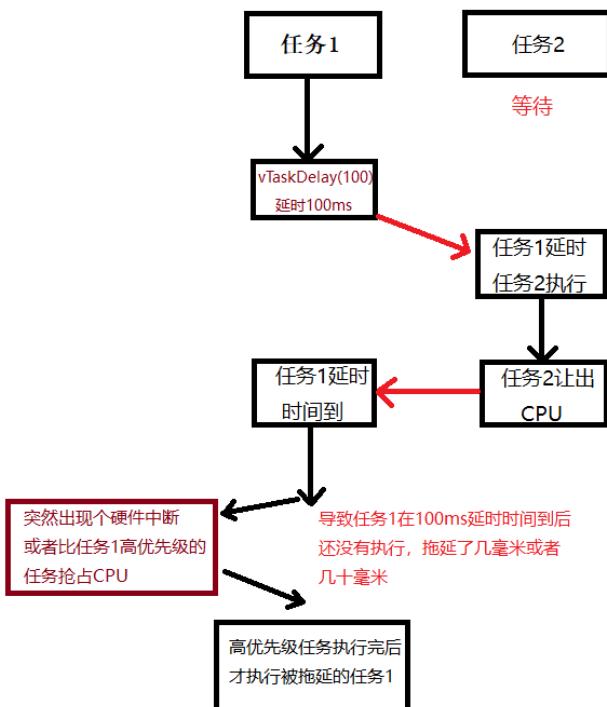
这个数字是不断在累加的，不会清 0，除非重启板子

void vTaskDelay( const TickType\_t xTicksToDelay ) //阻塞延时函数

xTicksToDelay : 延时多少毫秒

单位为系统节拍周期，比如系统的时钟节拍周期为 1ms，那么调用 vTaskDelay(1) 的延时时间为 1ms。

vTaskDelay 函数不能准确的延时时间到，就可以马上执行任务



```
void Task1(void)
{
    while(1)
    {
        printf("Task1.... \r\n");
        vTaskDelay(500);
    }
}

void Task2(void)
{
    while(1)
    {
        printf("Task2.... \r\n");
        vTaskDelay(1000);
    }
}
```

XXXZZZZZ  
 Task2....  
 Task1....  
 Task2....  
 Task1....  
 Task2....  
 Task1....  
 Task2.... 500ms 延时执行

这就是 `vTaskDelay` 延时函数的弊端，一般延时可以用，但是有些那种死脑筋的任务必须延时几百毫秒后必须马上执行，那么这种任务就不要用 `vTaskDelay` 做延时

用 `vTaskDelayUntil` 作精确延时是个好办法，看下面.....

```
void vTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const TickType_t xTimeIncrement )
//获取精确的延时时间(绝对延时函数), 延时时间到, 任务马上执行, 不受中断和优先级打断
pxPreviousWakeTime: 填入上次唤醒的时间
xTimeIncrement: 设置延时时间, 只不过这个参数得传入节拍数才行
```

返回节拍 = pdMS\_TO\_TICKS(传入延时的毫秒) //将毫秒数转换成节拍数

在 config 中 #define INCLUDE\_vTaskDelayUntil 1 //启用绝对延时

绝对延时 vTaskDelayUntil 函数使用要麻烦些, 但是精确。

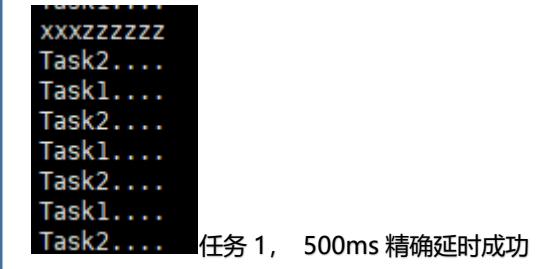
```
void Task1(void)
{
    /* 用于保存上次时间。调用后系统自动更新 */
    static TickType_t PreviousWakeTime;

    /* 设置延时时间, 将时间转为节拍数 */
    const TickType_t TimeIncrement = pdMS_TO_TICKS(500);

    /* 获取当前系统时间 */
    PreviousWakeTime = xTaskGetTickCount();

    while(1)
    {
        printf("Task1....\r\n");
        vTaskDelayUntil(&PreviousWakeTime,TimeIncrement); //当前系统时间 PreviousWakeTime+ TimeIncrement 节拍时间到达后, 延时结束
    }
}

void Task2(void)
{
    while(1)
    {
        printf("Task2....\r\n");
        vTaskDelay(500);
    }
}
```



延时多少毫秒?

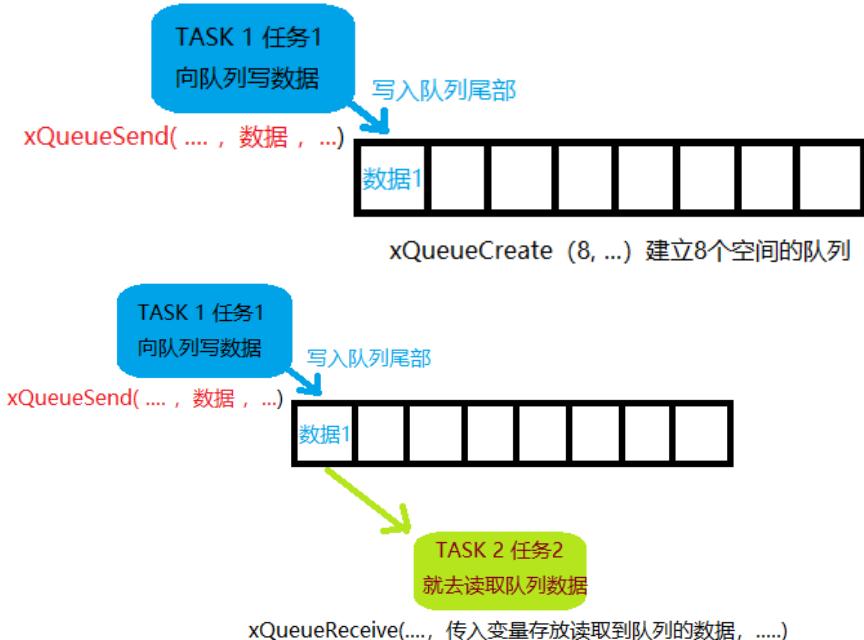
任务 1, 500ms 精确延时成功

## 消息队列

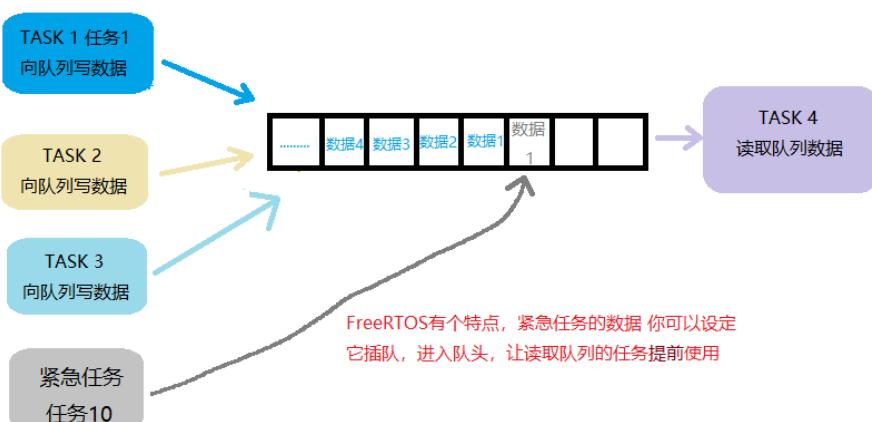


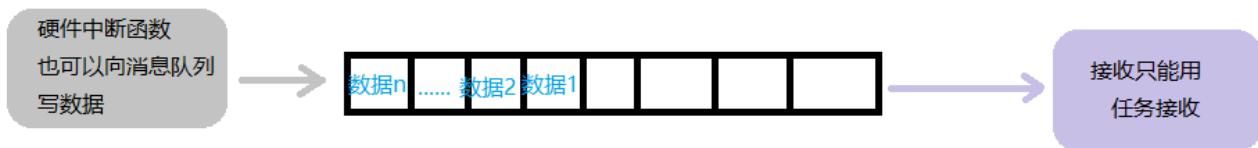
xQueueCreate (8, ...) 建立8个空间的队列

其实建立的这个消息队列也是一片内存



因为队列只有一个数据，所以即使队头，也是队列尾。





硬件中断里面必须马上写数据给消息队列，然后退出中断  
所以不能用 `xQueueSend( 队列, 数据, 等待时间 )` 这种具有等待  
功能的写消息函数，用专门为中断定制的写消息队列函数才行  
如: `xQueueSendFromISR(...)` 在中断中用这个函数发消息

在 config 中 `#define configUSE_QUEUE_SETS 1 //启用消息队列`

`typedef void * QueueHandle_t //创建句柄变量`

`QueueHandle_t xQueueCreate( uxQueueLength, uxItemSize ) //创建消息队列, 返回句柄`

`uxQueueLength:` 消息队列空间大小，就是可以存放几个值

`uxItemSize:` 每个值的数据类型大小，可以是 `char` 1 字节，也可以是 `short` 16 字节，`int` 32 字节，也可能是结构体，所以建议用 `sizeof` 来计算。

`QueueHandle_t:` 返回非 `NULL` 创建成功，返回 `NULL` 创建失败

`BaseType_t xQueueSend( QueueHandle_t xQueue, const void *pvItemToQueue, TickType_t xTicksToWait ) //写队列, 把数据写入队列尾部`

`xQueue:` 传入队列句柄，因为接受队列数据的函数也是通过这个句柄来判断是哪一个队列发的

`*pvItemToQueue:` 传入要发送的数据

`BaseType_t:` 返回 `pdPASS` 表示发送成功，非 `pdPASS` 发送失败

`BaseType_t xQueueSendToFront( QueueHandle_t xQueue, const void *pvItemToQueue, TickType_t xTicksToWait ) //写队列, 把数据写入队列头部, 如果你的任务有紧急的事件, 就需要插队, 就用这个函数`  
参数同上

`xQueueSendFromISR(QueueHandle_t xQueue, const void *pvItemToQueue, 填 NULL) //在中断中用非阻塞发送消息的函数, 这个函数不会死等, 没有数据也会执行结束。`

`xQueue:` 传入队列句柄

`*pvItemToQueue:` 传入要发送的数据

填 `NULL`

`BaseType_t:` 返回 `pdPASS` 表示发送成功，非 `pdPASS` 发送失败

`BaseType_t xQueueReceive( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait ) //接收发送过来的消息`

`xQueue:` 传入队列句柄，要传入发送队列的句柄，我才知道接收某个队列的数据，在 `xQueueSend` 函数段说了的

`*pvBuffer:` 接收发送过来的数据，如果是 `char`，就要定义 `char` 变量来接收，如果是 `int` 就要定义 `int` 变量来接收，如果是结构体，就要定义结构体变量来接收。

`xTicksToWait:` 阻塞等待时间，如果我在任务中使用 `xQueueReceive`，其它线程没有发响应的消息过来，我就会将任务卡死在这里，一直等待，把 CPU 让出来。如果有消息突然发过来我就不用等待，继续向下执行。等待时间最大 `portMAX_DELAY`(这是个很长的时间)，如果你不想等待，可以填 `NULL`。

```

#include "stm32f10x.h"
#include "sysclock.h"
#include "uartprintf.h"
#include "stdio.h"

#include "FreeRTOS.h" //使用 FreeRTOS 里面的函数一定要包含这个头文件
#include "task.h" //使用 FreeRTOS 里面的函数一定要包含这个头文件
#include "queue.h" //消息队列要包含该头文件

//定义任务控制块
static TaskHandle_t Task1TCB;
static TaskHandle_t Task2TCB;
QueueHandle_t xQueue; //创建一个消息队列句柄

void Task1(void)
{
    uint8_t Receive_data; //任务 2 发来的数据需要存放，该变量就是用来存放的
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

    while(1)
    {
        xReturn = xQueueReceive(xQueue,&Receive_data,portMAX_DELAY); //接收任务 2 发来的数据
        if (pdTRUE == xReturn)
            printf("Task1 Receive = %d\r\n",Receive_data);
        else
            printf("Task1 Receive failed\r\n");
    }
}

void Task2(void)
{
    uint8_t send_data = 50; //要发送给任务 1 的数据
    BaseType_t xReturn = pdPASS;// 定义一个信息返回值来判断消息是否发送成功，默认为 pdPASS
    xQueue = xQueueCreate(10,sizeof(char)); // 创建消息队列，消息队列有 10 个空间，每个空间存放一个 char 大小的数据
    while(1)
    {
        xReturn = xQueueSend( xQueue , &send_data, 0 ); // 消息发送 等待时间 0
        if (pdPASS == xReturn)
            printf("Task2 Queue send success\r\n");
        else
            printf("Task2 Queue send failed\r\n");
        vTaskDelay(1000);
    }
}

int main(void)
{
    初始化串口，创建两个任务.....
}

```

```

Task2 QueueTask1 Rece send succeive = 50
ss
Task2 QueueTask1 Rece send succeive = 50
ss
Task2 QueueTask1 Rece send succeive = 50
ss
Task2 QueueTask1 Rece send succeive = 50

```

从打印的数字来看，消息队列确实发生成功的了。

但是打印字符串乱序了，这是因为引入操作系统后，多线程任务切换太快的原因，需要改进 printf

解决 printf 打印字符乱序，采用消息队列，修改 printf，

需要下一章信号量的知识，可以先学习下信号量

printf 函数是不可重入的，也就是在多任务情况下不能多个任务同时调用 printf，所以像这种不可重入的函数，要在里面进行上锁或者用信号量什么的，保证 printf 不会同时被多个任务调用。

```
#include "semphr.h" //二值信号量头文件
#include <stdarg.h> //printf 调用的是 vprintf，而 vprintf 需要 va_list 类型，所以包含该头文件
```

```
SemaphoreHandle_t semDebug = NULL;

void debug_printf(const char *fmt, ...)
{
    va_list args;
    if(xSemaphoreTake(semDebug,5)==pdFALSE) return;
    va_start(args,fmt);
    vprintf(fmt,args);
    va_end(args);
    xSemaphoreGive(semDebug);
}

//定义任务控制块
static TaskHandle_t Task1TCB;
static TaskHandle_t Task2TCB;
QueueHandle_t xQueue; //创建一个消息队列句柄
```

```
void Task1(void)
{
    uint8_t Receive_data; //任务 2 发来的数据需要存放，该变量就是用来存放的
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

    while(1)
    {
        xReturn = xQueueReceive(xQueue,&Receive_data,portMAX_DELAY); //接收任务 2 发来的数据
        if (pdTRUE == xReturn)
            debug_printf("Task1 Receive = %d\r\n",Receive_data);
        else
            debug_printf("Task1 Receive failed\r\n");
    }
}
```

```
void Task2(void)
{
    uint8_t send_data = 50; //要发送给任务 1 的数据
    BaseType_t xReturn = pdPASS;// 定义一个信息返回值来判断消息是否发送成功，默认为 pdPASS
    xQueue = xQueueCreate(10,sizeof(char)); // 创建消息队列，消息队列有 10 个空间，每个空间存放一个 char 大小的数据
    while(1)
    {
        xReturn = xQueueSend( xQueue , &send_data, 0 ); // 消息发送 等待时间 0
        if (pdPASS == xReturn)
            debug_printf("Task2 Queue send success\r\n");
        else
            debug_printf("Task2 Queue send failed\r\n");
        vTaskDelay(1000);
    }
}
```

给 printf 专门创建  
个二值信号量

给 printf 是调用的 vprintf,  
vprintf 才去调用 fput()，所以干脆直接把 vprintf 拿出来修改，给  
vprintf 专门定义个打印函数

这个 vprintf 使用主要的区别就是我加入了信号  
量，但是你发现没，debug\_printf 执行的第 2 行  
就要获取信号量，如果这时候我没有信号量，程  
序就会阻塞，所以我们在程序初始化的时候一定  
要先释放信号量(信号量写 1)

printf 全部改为  
debug\_printf 就能解决打印乱  
序的问题

```
int main(void)
{
    BaseType_t xReturn1,xReturn2;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //STM32 中断优先级分组为 4, 即 4bit 都用来表示抢占优先级, 范围为: 0~15

    //优先级分组只需要分组一次即可, 以后如果有其他的任务需要用到中断,
    //都统一用这个优先级分组, 千万不要再分组, 切忌。

    printf("xxxxxxxx\r\n");
    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1, 因为 debug_printf 函数最后一行才会给 semDebug 写 1

    创建任务 1....
    创建任务 2...
}

xxxxxxxx
Task2 Queue send success
Task1 Receive = 50
```

这里就是程序初始化释放信号量的地方

打印输出正常了。

所以以后如果加入了 FreeRTOS 或者其它什么操作系统, 使用 debug\_printf(...)函数来打印。

## 信号量

### 二值信号量

二值信号量变量 = 0

xSemaphoreCreateBinary() 创建的信号量默认为0

二值信号量变量 = 0



任务1  
去获取二值信号量

xSemaphoreTake()

发现信号量变量是0，那么  
任务1阻塞等待

任务2 在某时候  
向二值信号量写1



二值信号量变量 = 1

xSemaphoreGive

向二值信号量变量写 1

任务2 在某时候  
向二值信号量写1



二值信号量变量 = 1

xSemaphoreGive

向二值信号量变量写 1

任务1  
去获取二值信号量

xSemaphoreTake()

发现信号量变量是1，  
任务1执行，同时将  
信号量变量清0

线程2

任务2 不发送  
xSemaphoreGive

二值信号量变量 = 0

线程1

任务1 阻塞不执行

任务2 发送  
xSemaphoreGive

二值信号量变量 = 1

任务1 执行

任务2 不发送  
xSemaphoreGive

二值信号量变量 = 0

任务1 阻塞不执行

任务2 发送  
xSemaphoreGive

二值信号量变量 = 1

任务1 执行

所以二值信号量就是实现任务间同步执行的功能，也就是任务2夺一下，任务1跳一下，任务2夺一下，任务1跳一下，就是这个意思。

```
typedef QueueHandle_t SemaphoreHandle_t; //定义信号量句柄，信号量发送和接收都靠这个句柄识别
SemaphoreHandle_t xSemaphoreCreateBinary() //创建二值信号量变量，默认为 0
```

BaseType\_t xSemaphoreGive( QueueHandle\_t xSemaphore ) //发送信号量，向信号量变量写 1

xSemaphore: 传入信号量句柄，就表示信号量是发送给这个句柄的

BaseType\_t: 返回 pdTRUE 发送成功，返回 pdFALSE 发送失败

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xBlockTime )
//信号量接收函数
```

xSemaphore: 传入信号量句柄，查看这个句柄里面的信号量变量是否为 1，为 1 该任务不阻塞，为 0 该任务阻塞。

XBlockTime: 如果信号量变量为 0，是否要一直阻塞下去，portMAX\_DELAY 就是无限阻塞，如果 xBlockTime 填 0 就是不阻塞，程序继续向下执行，如果填 20 就是阻塞 20 个时钟周期然后程序继续向下执行。

代码示例：

```
#include "semphr.h" //二值信号量头文件
SemaphoreHandle_t xSemaphore = NULL; //创建一个空信号量句柄
void Task1(void)
{
    BaseType_t xReturn = pdPASS; //信号量获取状态返回值

    while(1)
    {
        xReturn = xSemaphoreTake(xSemaphore , portMAX_DELAY); //portMAX_DELAY 该任务永久阻塞，等待任务 2 发送过来信号量才执行
        if (pdTRUE == xReturn)
            debug_printf("Task1 Sem get success\r\n");
        else
            debug_printf("Task1 Sem get failed\r\n");
    }
}

void Task2(void)
{
    BaseType_t xReturn = pdFALSE; //返回二值信号量是否发送成功

    xSemaphore = xSemaphoreCreateBinary(); //创建 1 个二值信号量，返回句柄，现在变量为 0
    if(NULL !=xSemaphore)
        printf("Task2 Binar Sem Creat success\r\n"); //二值信号量创建成功
    else
        printf("Task2 Binar Sem Creat failed\r\n");
    while(1)
    {
        xReturn = xSemaphoreGive( xSemaphore ); //发送信号量，也就是向变量+1
        if (xReturn == pdTRUE)
            debug_printf("Task2 Sem send success\r\n");
        else
            debug_printf("Task2 Sem send failed\r\n"); //debug_printf 是经过处理的 printf，上一节有讲
        vTaskDelay(1000);
    }
}
```

```
Task2 Sem send success
Task1 Sem get success
```

这就是二值信号量，Task2 任务夺一下，Task1 任务跳一下。

```

void Task2(void)
{
    BaseType_t xReturn = pdFALSE; //返回二值信号量是否发送成功
    xSemaphore = xSemaphoreCreateBinary(); //创建1个二值信号量，返回句柄，现在变量为0

    if(NULL !=xSemaphore)
        printf("Task2 Binar Sem Creat success\r\n");
    else
        printf("Task2 Binar Sem Creat failed\r\n");

    while(1)
    {
        //xReturn = xSemaphoreGive( xSemaphore ); //发送信号量，也就是向变量+1
        if ( xReturn == pdTRUE )
            debug_printf("Task2 Sem send success\r\n");
        else
            debug_printf("Task2 Sem send failed\r\n");

        vTaskDelay(1000);
    }
}

XXXXXXX
Task2 Binar Sem Creat success
Task2 Sem send failed
Task2 Sem send failed
Task2 Sem send failed
Task2 Sem send failed

```

任务2停止发送信号量

Task1 任务就会卡在 xSemaphoreTake 阻塞等待。

## 计数信号量

sem = 4

xSemaphoreCreateCounting(.., 4)

创建4个信号量

sem = 4

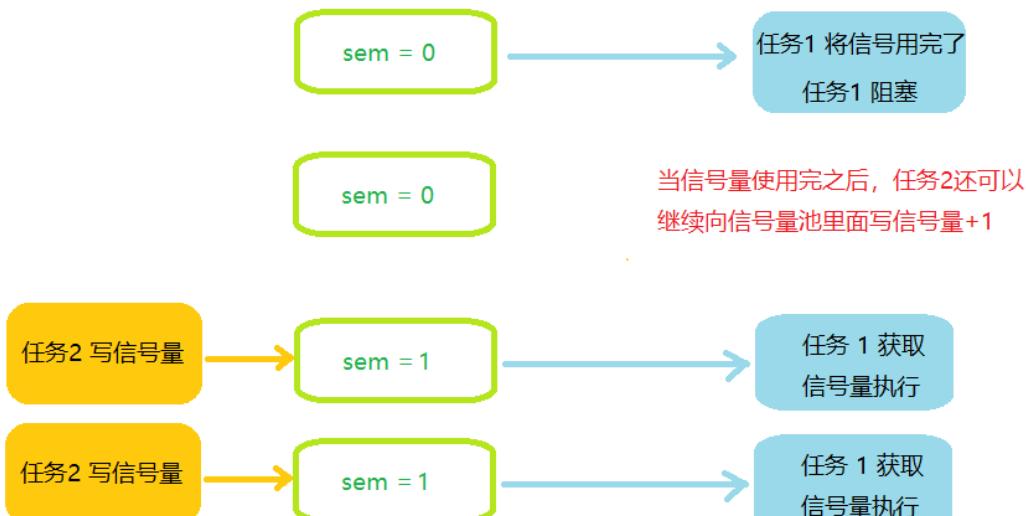
任务1 死循环执行了4次xSemaphoreTake

sem = 0

任务1 将信号用完了  
任务1 阻塞

这就是计数信号量，和二值信号量不一样，在创建计数信号量的时候，就可以设置信号量数量，所以在没有Task2发送信号的情况下，任务1也能拿到信号进行执行，就是这个原因。

二值信号量创建的默认信号是0，所以这和计数信号量还是有区别的。



任务2信号量一直写，那么任务1就一直取信号量执行，无限循环这样做

使用计数信号量记得在 FreeRTOSconfig.h 里 #define configSUPPORT\_DYNAMIC\_ALLOCATION 1 //打开动态内存分配，和 #define configUSE\_COUNTING\_SEMAPHORES 1 //启用计数信号量

`SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount , UBaseType_t uxInitialCount ) //创建计数信号量`

`uxMaxCount`: 计数信号量最大值，主要是在信号量使用完之后，在没有程序使用 `xSemaphoreGive` 消费信号量的情况下，`xSemaphoreGive` 发送信号最多能向信号量池写几次。

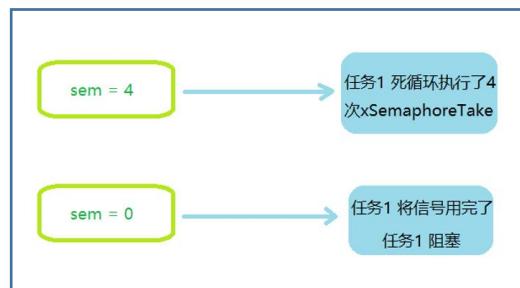
`uxInitialCount`: 创建计数信号量时，向信号量池准备几个信号

发送信号量和接收信号量函数和上一节一样

```
#include "semphr.h" //二值信号量头文件
SemaphoreHandle_t CountSem; //计数信号量创建
```

```
void Task1(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task1 get Count sem success \r\n");
        else
            debug_printf("Task1 get Count sem failed\r\n");
    }
}
```

```
void Task2(void)
{
    CountSem = xSemaphoreCreateCounting(5,4); //计数信号量最大值是 5，初始值为 4
    if(CountSem != NULL)
        debug_printf("Count Creat success\r\n");//计数信号量创建成功
    else
        debug_printf("Count Creat failed\r\n");
    while(1)
    {
        vTaskDelay(1000);
    }
}
```



```
xxxxxxxx
Count Creat success
Task1 get Count sem success
```

在没有 Task2 任务发送信号的情况下，Task1 任务可以直接获取创建好的信号量。

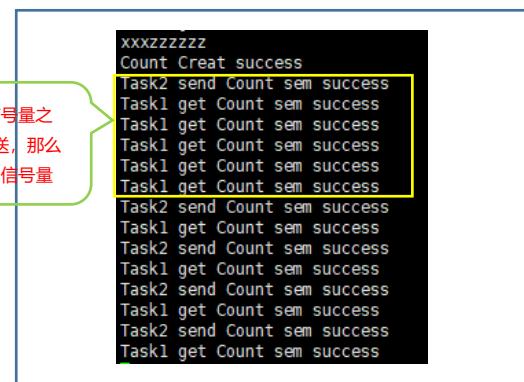
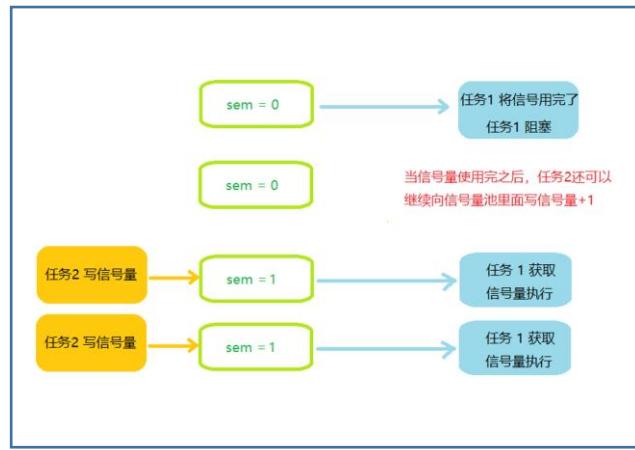
```

SemaphoreHandle_t CountSem; //计数信号量创建

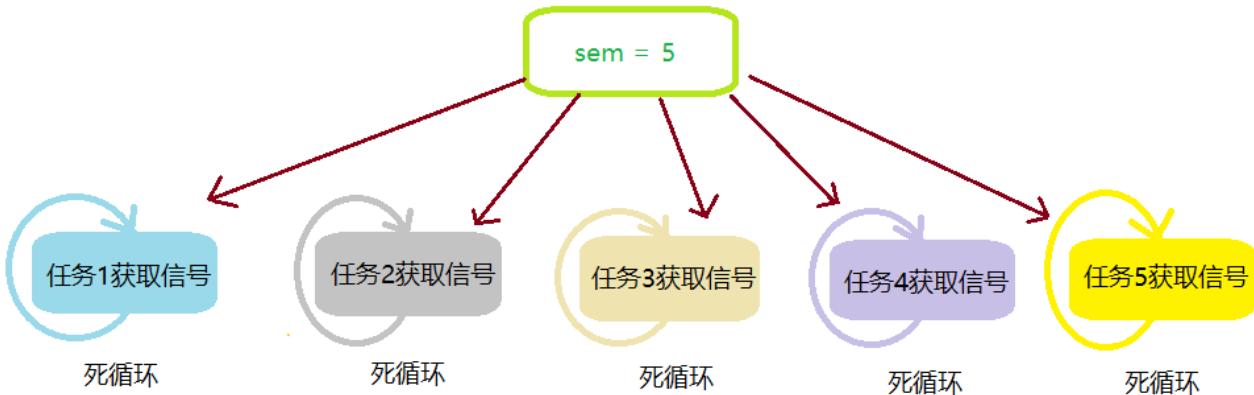
void Task1(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task1 get Count sem success \r\n");
        else
            debug_printf("Task1 get Count sem failed\r\n");
    }
}

void Task2(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断发送是否会成功
    CountSem = xSemaphoreCreateCounting(5,4); //计数信号量最大值是 5, 初始值为 4
    if(CountSem != NULL)
        debug_printf("Count Creat success\r\n"); //计数信号量创建成功
    else
        debug_printf("Count Creat failed\r\n");
    while(1)
    {
        xReturn = xSemaphoreGive(CountSem);
        if (pdTRUE == xReturn)
            debug_printf("Task2 send Count sem success \r\n");
        else
            debug_printf("Task2 send Count sem failed\r\n");
        vTaskDelay(1000);
    }
}

```



这样看来其实计数信号量意义不是很大？计数信号量主要是用于多任务。



计数信号量主要用于多任务同时获取同一个信号来处理同一个事件，但是这样死循环真的能每个任务都能获取到信号吗？下面代码验证下

```

SemaphoreHandle_t CountSem; //计数信号量创建
void Task1(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task1 get Count sem success \r\n");
        else
            debug_printf("Task1 get Count sem failed\r\n");
    }
}
void Task3(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task3 get Count sem success \r\n");
        else
            debug_printf("Task3 get Count sem failed\r\n");
    }
}
void Task4(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task4 get Count sem success \r\n");
        else
            debug_printf("Task4 get Count sem failed\r\n");
    }
}
void Task5(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
        if (pdTRUE == xReturn)
            debug_printf("Task5 get Count sem success \r\n");
        else
            debug_printf("Task5 get Count sem failed\r\n");
    }
}

void Task2(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断发送是否会成功
    CountSem = xSemaphoreCreateCounting(5,6); //计数信号量最大值是 5, 初始值为 4
    if(CountSem != NULL)
        debug_printf("Count Creat success\r\n"); //计数信号量创建成功
    else
        debug_printf("Count Creat failed\r\n");
    while(1)
    {
        xReturn = xSemaphoreGive(CountSem);
        if (pdTRUE == xReturn)
            debug_printf("Task2 send Count sem success \r\n");
        else
            debug_printf("Task2 send Count sem failed\r\n");

        vTaskDelay(1000);
    }
}

```

```

int main(void)
{
    BaseType_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;
    RCC_Configuration(); // 初始化时钟
    USART_Config(115200); // 初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); // STM32 中断优先级分组为 4, 即 4bit 都用来表示抢占优先级, 范围为: 0~15

    // 优先级分组只需要分组一次即可, 以后如果有其他的任务需要用到中断,
    // 都统一用这个优先级分组, 千万不要再分组, 切忌。

    printf("xxxxxxxx\r\n");
    semDebug = xSemaphoreCreateBinary(); // 创建打印的二值信号量
    xSemaphoreGive(semDebug); // 必须在初始化前就给先向 semDebug 变量里面写入 1, 因为 debug_printf 函数最后一行才会给 semDebug 写 1

    xReturn1 = xTaskCreate((TaskFunction_t)Task1, // 任务函数地址
                           (const char*)"Task1",           // 任务名
                           (uint32_t)128,                 // 任务栈大小
                           (void*)NULL,                  // 向函数传参
                           (UBaseType_t)1,                // 优先级
                           (TaskHandle_t*)&Task1TCB); // 任务控制块

    xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                           (const char*)"Task2",
                           (uint32_t)128,
                           (void*)NULL,
                           (UBaseType_t)7,
                           (TaskHandle_t*)&Task2TCB); // 因为任务 2 是发送消息的, 所以优先级一定要高于其它任务
                                            // 不然你会发现任务启动后程序卡死了, 有可能是接收消息的

    // 任务优先级高了, 导致先执行, 但是这时候 Task2 没有先执行, 没有创建信号量, 所以程序卡死。
    创建任务 3
    创建任务 4
    创建任务 5
}

xxxxxxxx
Count Creat success
Task2 send Count sem failed
Task5 get Count sem success
Task2 send Count sem success
Task5 get Count sem success
Task2 send Count sem success
Task5 get Count sem success
Task2 send Count sem success
Task5 get Count sem success
Task2 send Count sem success
Task5 get Count sem success

```

发现只有 Task2 和 Task5 在运行。这是为什么呢？因为 Task5 是接收信号量的任务，除了 Task2 优先级最高外，Task5 优先级比 Task1,3,4 都高，所以 Task5 一直获取信号。如果取消 while 循环呢？其实也不行，因为任务函数就是要求无限循环的，所以你会发现启动之后程序又再次卡死到 Task5 线程了。除非将任务删除。

```

SemaphoreHandle_t CountSem; //计数信号量创建
void Task1(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

    xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
    if (pdTRUE == xReturn)
        debug_printf("Task1 get Count sem success \r\n");
    else
        debug_printf("Task1 get Count sem failed\r\n");
    vTaskDelete(Task1TCB); //删除任务
}

void Task3(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

    xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
    if (pdTRUE == xReturn)
        debug_printf("Task3 get Count sem success \r\n");
    else
        debug_printf("Task3 get Count sem failed\r\n");
    vTaskDelete(Task3TCB); //删除任务
}

void Task4(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

    xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
    if (pdTRUE == xReturn)
        debug_printf("Task4 get Count sem success \r\n");
    else
        debug_printf("Task4 get Count sem failed\r\n");
    vTaskDelete(Task4TCB); //删除任务
}

void Task5(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功

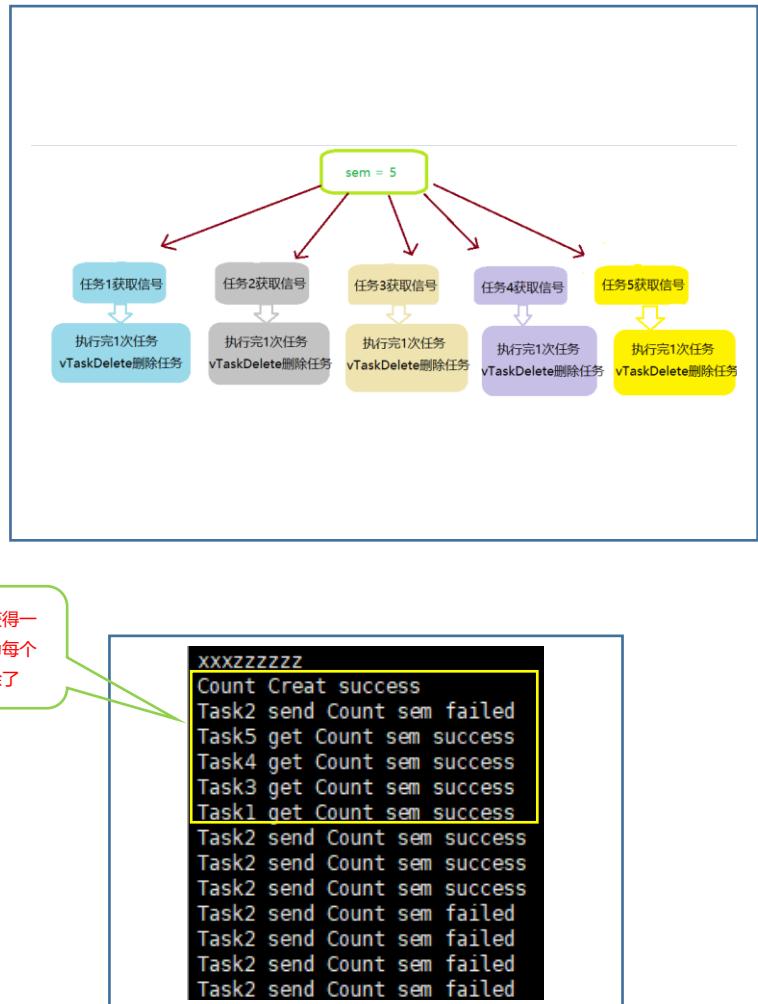
    xReturn = xSemaphoreTake(CountSem,portMAX_DELAY);
    if (pdTRUE == xReturn)
        debug_printf("Task5 get Count sem success \r\n");
    else
        debug_printf("Task5 get Count sem failed\r\n");

    vTaskDelete(Task5TCB); //删除任务
}

void Task2(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断发送是否会成功
    CountSem = xSemaphoreCreateCounting(5,6); //计数信号量最大值是5，初始值为4
    if(CountSem != NULL)
        debug_printf("Count Creat success\r\n");//计数信号量创建成功
    else
        debug_printf("Count Creat failed\r\n");
    while(1)
    {
        xReturn = xSemaphoreGive(CountSem);
        if (pdTRUE == xReturn)
            debug_printf("Task2 send Count sem success \r\n");
        else
            debug_printf("Task2 send Count sem failed\r\n");
        vTaskDelay(1000);
    }
}

```

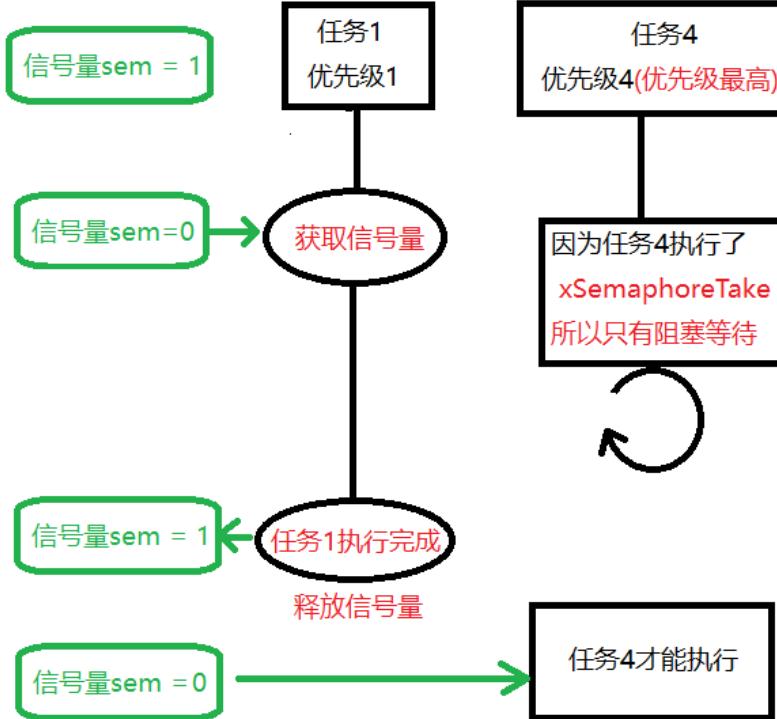
这种创建多个任务，每个任务只执行一次的场景，而且必须在某时某刻准点执行的场景，用计数信号量才有意义。一般都是先创建任务后就进行阻塞，等待某时某刻发送信号量，一次性执行完多个任务。



## 互斥量

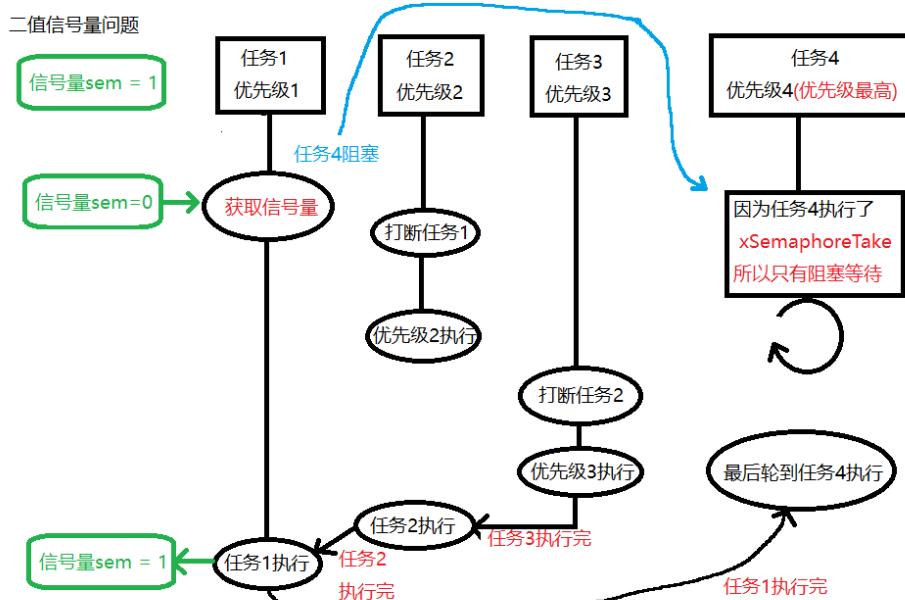
解决二值信号量优先级翻转问题

二值信号量问题



你看，不管任务有多高的优先级，只要使用了 **xSemaphoreTake** 函数，就算再有紧急的事情需要高优先级任务处理，高优先级任务也执行不了

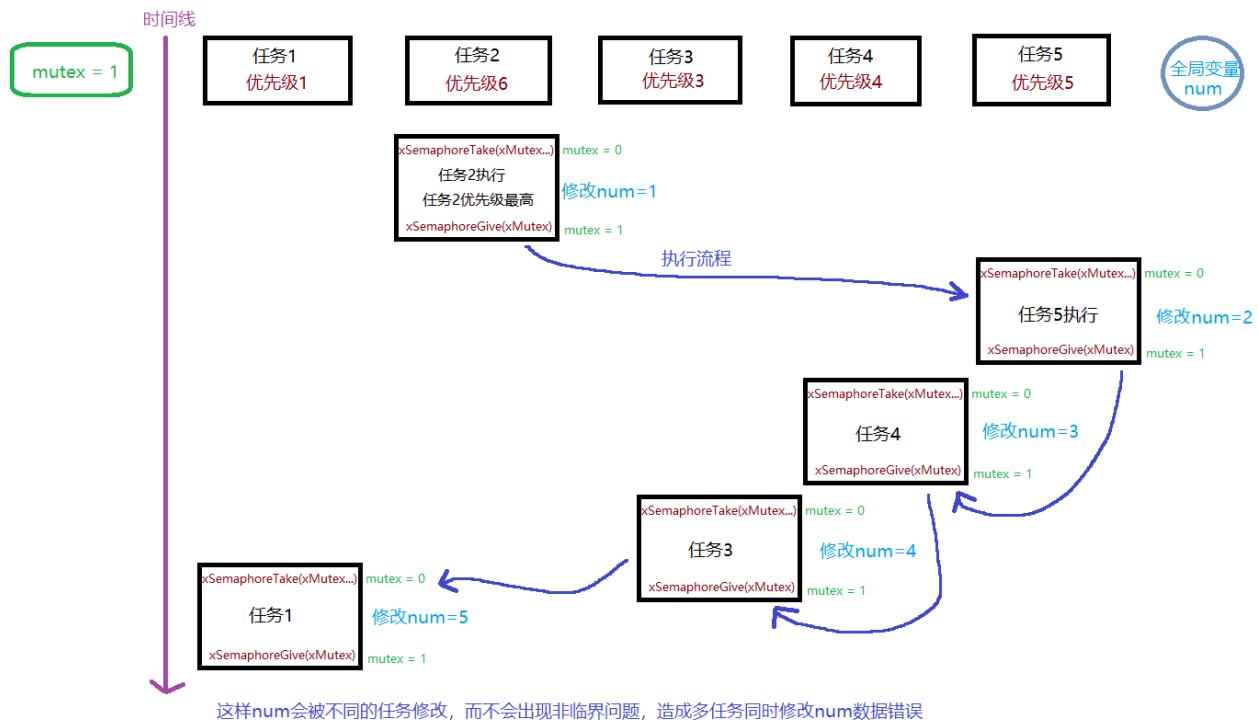
还有一种更恶劣的情况



为什么优先级2可以打断任务1呢？因为二值信号量不是给线程上锁记住。而是线程同步，一个线程用 **xSemaphoreTack** 才会受信号量阻塞影响，如果不用 **xSemaphoreTack** 就没有任何问题，想抢占其它线程照样抢占。任务3同理

这种一个最高优先级的紧急任务要等前面3个低优先级任务执行完才能执行，这就是优先级翻转，更恐怖。





### 代码示例:

```

SemaphoreHandle_t xMutex; //定义互斥量句柄
static int num = 0; //所有任务都要修改这个变量

void Task3(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1){
        xReturn = xSemaphoreTake(xMutex,portMAX_DELAY); //获取互斥量
        num++;
        if (pdTRUE == xReturn)
            debug_printf("Task3 get xMutex success num = %d \r\n",num);
        else
            debug_printf("Task3 get xMutex failed\r\n");
        xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁
        vTaskDelay(1000);
    }
}

void Task4(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1){
        xReturn = xSemaphoreTake(xMutex,portMAX_DELAY); //获取互斥量
        num++;
        if (pdTRUE == xReturn)
            debug_printf("Task4 get xMutex success num = %d \r\n",num);
        else
            debug_printf("Task4 get xMutex failed\r\n");
        xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁
        vTaskDelay(1000);
    }
}

void Task5(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {

```

```

xReturn = xSemaphoreTake(xMutex,portMAX_DELAY); //获取互斥量
num++;
if (pdTRUE == xReturn)
    debug_printf("Task5 get xMutex success num = %d \r\n",num);
else
    debug_printf("Task5 get xMutex failed\r\n");
xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁
vTaskDelay(1000);
}

}

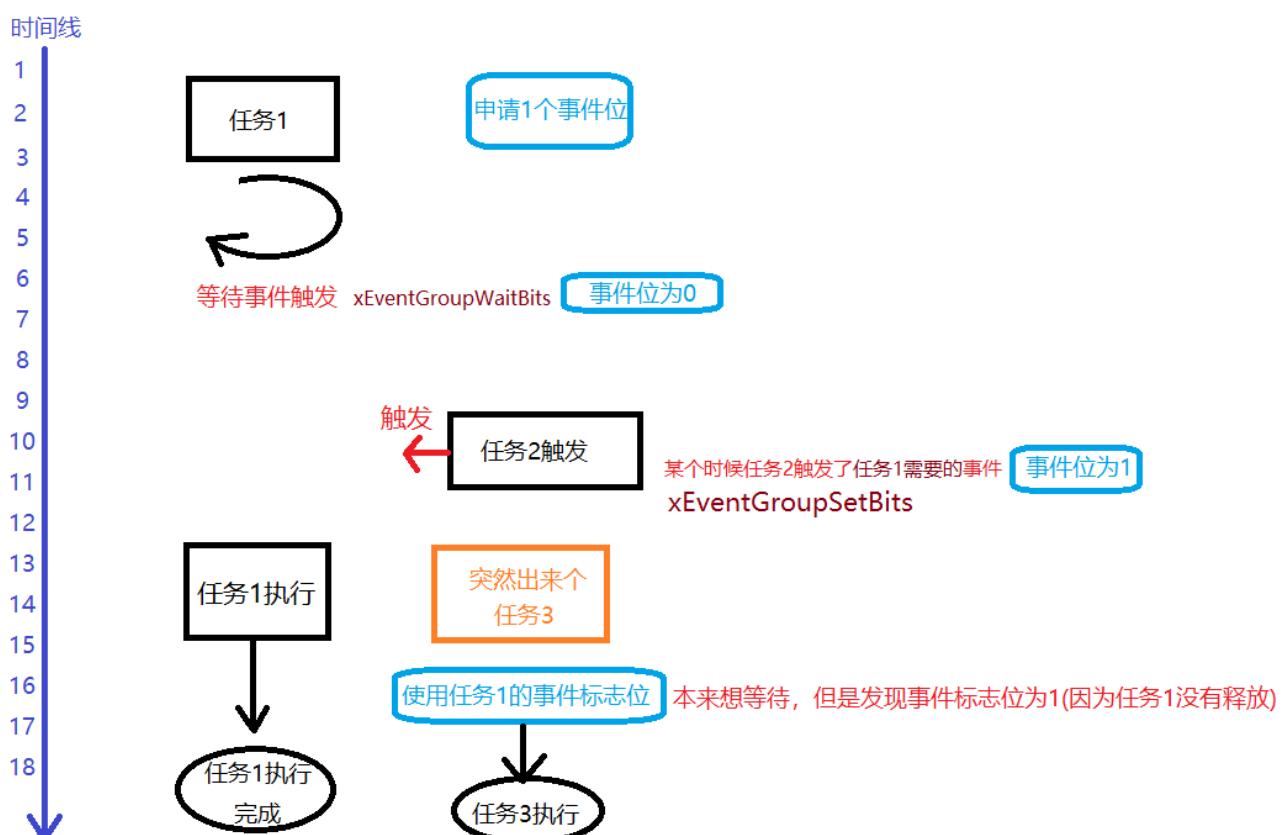
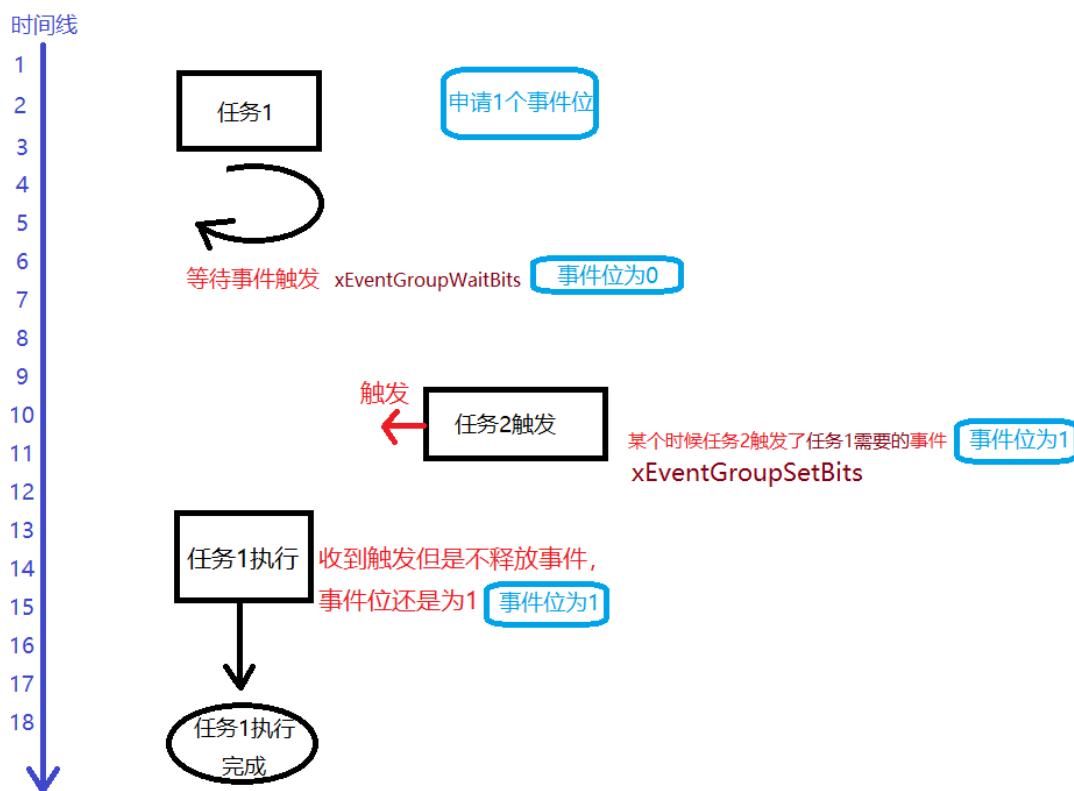
void Task1(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断接收是否会成功
    while(1)
    {
        xReturn = xSemaphoreTake(xMutex,portMAX_DELAY); //获取互斥量上锁
        num++;
        if (pdTRUE == xReturn)
            debug_printf("Task1 get xMutex success num = %d \r\n",num);
        else
            debug_printf("Task1 get xMutex failed\r\n");
        xReturn = xSemaphoreGive(xMutex); //发送互斥量解锁
        vTaskDelay(1000);
    }
}

void Task2(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断发送是否会成功
    xMutex = xSemaphoreCreateMutex(); //创建 1 个互斥量，多个任务使用
    if(xMutex != NULL)
        debug_printf("xMutex Creat success\r\n"); //互斥量创建成功
    else
        debug_printf("xMutex Creat failed\r\n");
    while(1)
    {
        xReturn = xSemaphoreTake(xMutex,portMAX_DELAY); //获取互斥量上锁
        num++;
        if (pdTRUE == xReturn)
            debug_printf("Task2 send xMutex success num = %d \r\n",num);
        else
            debug_printf("Task2 send xMutex failed\r\n");
        xReturn = xSemaphoreGive(xMutex); //发送互斥量
        vTaskDelay(1000);
    }
}

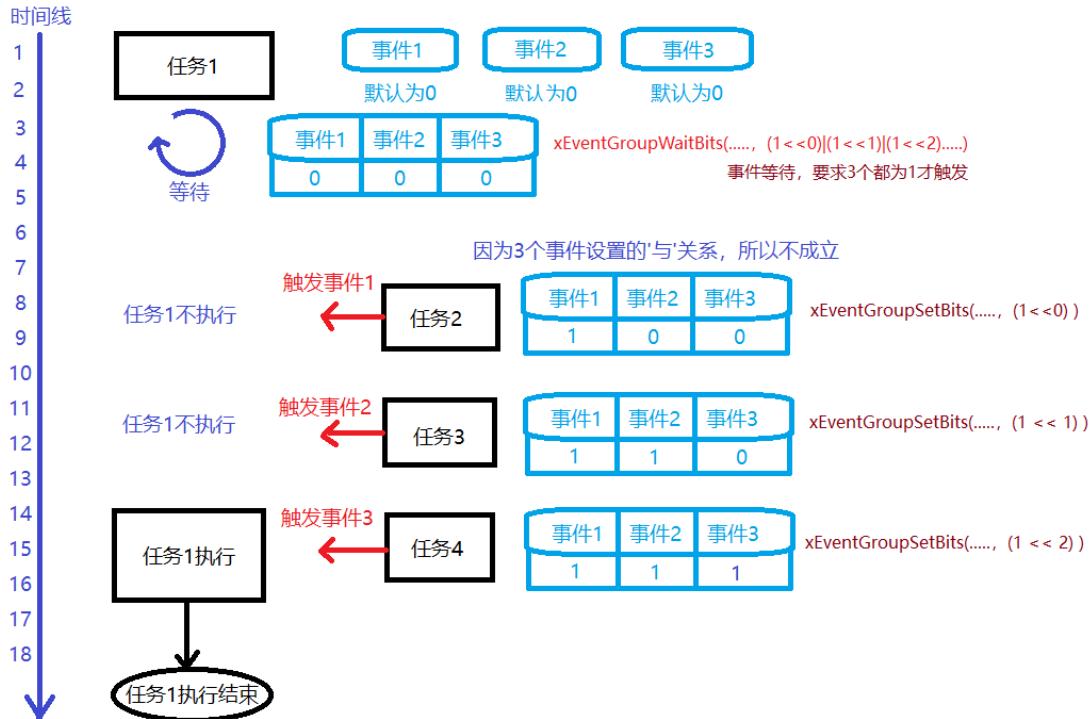
xMutex Creat success
Task2 send xMutex success num = 1
Task5 get xMutex success num = 2
Task4 get xMutex success num = 3
Task3 get xMutex success num = 4
Task1 get xMutex success num = 5
Task2 send xMutex success num = 6
Task5 get xMutex success num = 7
Task4 get xMutex success num = 8
Task3 get xMutex success num = 9
Task1 get xMutex success num = 10
Task2 send xMutex success num = 11
Task5 get xMutex success num = 12
Task4 get xMutex success num = 13
Task3 get xMutex success num = 14
Task1 get xMutex success num = 15
Task2 send xMutex success num = 16
Task5 get xMutex success num = 17
Task4 get xMutex success num = 18
Task3 get xMutex success num = 19
Task1 get xMutex success num = 20

```

## 事件使用，解决任务同步



事件还有一种功能就是1个任务等待多个事件成立后才执行，或者多个任务等待多个事件成立后才批量执行



这就是事件‘与’实现的多事件触发一个任务。比如有些系统要求，电源事件稳定，环境事件稳定，时钟事件稳定，三个变量同时稳定之后才能执行，这个时候就可以用事件。事件‘或’就是任何1个事件置1任务都可以运行。

FreeRTOS 多事件触发一个任务，就需要用事件组来做，事件组也就是一个16位或者32位的变量，每位代表一个事件

51单片机是8位机，所以只能打开16位的事件组

变量的位为0表示等待阻塞，为1表示非阻塞，所以发送事件函数是将变量的某位置1，接收事件函数是将变量的某位置0，当然也可以不置0，不置0就不阻塞任务事件组用在多个位置1才运行任务的应用

STM32是32位机，打开32位事件组

### 事件和信号量区别：

信号量可以传递参数，但是信号量只能一对一触发。

事件无法传递参数，但是可以做为多个事件触发。当多个事件同时满足时才执行某个人物，用于多情况判定。

那为什么不用变量做事件呢？，因为变量你需要去轮询查询变量的状态，如果我 while(1)循环一次轮询为 20ms 毫秒一个周期，如果紧急事件需要 1ms 之内就必须处理，你怎么办？而且变量同一时间多个任务都在写该变量，那么变量的值到底是多少呢？所以事件是最好的选择。

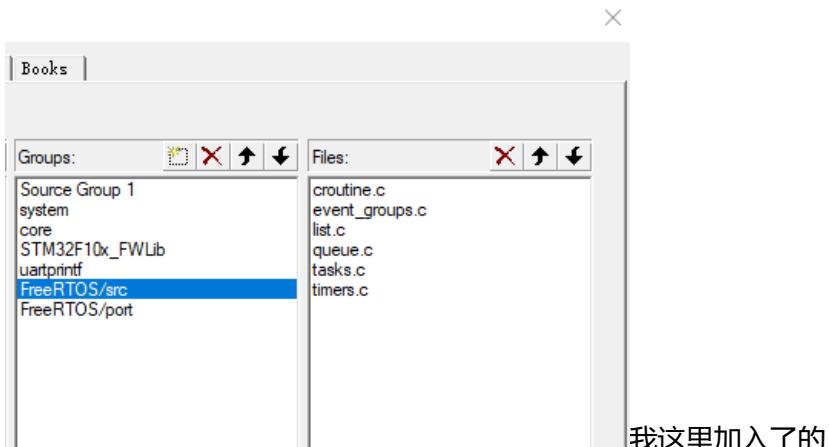
注意：一个 16 位的事件组只能用低 8 位，

注意：一个 32 位的事件组只能用低 24 位。



```
#define configUSE_16_BIT_TICKS      0 //系统最大节拍数 32 位 , 也就是 uxEventBits 事件标志位有 32 位
#define configUSE_16_BIT_TICKS      1 //系统最大节拍数 16 位 , 也就是 uxEventBits 事件标志位有 16 位
我是 STM32 单片机, 32 位的, 我选着 0, 事件标志位 32 位, 如果是 51 单片机可以选择 1, 16 位事件位。
```

要想使用事件必须加入 FreeRTOS/source/event\_groups.c 这个文件到工程



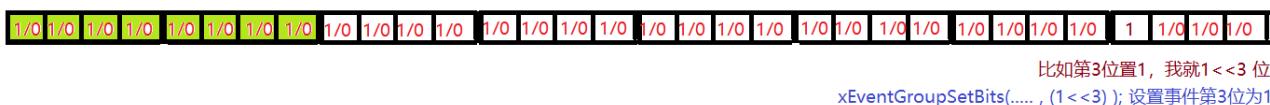
```
typedef void * EventGroupHandle_t //定义事件句柄
```

```
EventGroupHandle_t xEventGroupCreate( void ) //创建事件组
```

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet ) //发送事件函数
```

XEventGroup: 放入事件组, 就是 EventGroupHandle\_t 定义的句柄

uxBitsToSet: 事件组哪一位置 1?



```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait )//阻塞等待事件到来
```

xEventGroup: 事件句柄, 用来区分谁发送的事件

uxBitsToWaitFor: 指定需要等待事件组哪些位为 1 才能越过阻塞状态。

如果需要 3 位为 1, 就填入  $1 \ll 3$  位

如果需要多位为 1 才取消阻塞, 比如 bit5, bit6 为 1 才取消阻塞, 就填入  $(1 \ll 5) | (1 \ll 6)$

xClearOnExit: 等待事件到来之后, 要不要把相应的位清 0, pdTRUE 系统清除事件标志位, pdFALSE 系统不清楚事件标志位。

xWaitForAllBits: 在多事件位触发中, 选择 pdTRUE 就是 uxBitsToWaitFor 里面的位都满足了(与关系), 才取消阻塞。选择 pdFALSE 就是 uxBitsToWaitFor 里面的位只要满足 1 位置 1 就取消阻塞(或关系)

xTicksToWait: 填入阻塞的时间, 时间到了, 就算事件没有触发, 也要取消阻塞继续执行。

```

static EventGroupHandle_t Event_Handle = NULL; //事件句柄
#define EVENTbit_0 (1<<0)
#define EVENTbit_1 (1<<1)
#define EVENTbit_2 (1<<2)
void Task1(void)
{
    EventBits_t EventValue; //获取事件等待返回的结果
    while(1)
    {
        EventValue = xEventGroupWaitBits(Event_Handle,EVENTbit_1,pdTRUE,pdTRUE,portMAX_DELAY);
        //阻塞等待事件 1 EVENTbit_1 被触发

        debug_printf("Task1 wait EVENTbit_1 Event arrival = 0x%x\r\n",EVENTbit_1);
        //等待事件 1 EVENTbit_1 发生成功，执行打印
        vTaskDelay(1000);
    }
}
void Task2(void)
{
    BaseType_t xReturn = pdTRUE; //用来判断发送是否会成功
    Event_Handle = xEventGroupCreate(); //创建事件

    if(Event_Handle != NULL)
        debug_printf("Event success\r\n"); //事件创建成功
    else
        debug_printf("Event failed\r\n");

    xEventGroupSetBits(Event_Handle,EVENTbit_1); //向事件组第 1 位置 1

    while(1)
    {
        vTaskDelay(1000);
    }
}

```

XXXZZZZZ  
Event success  
Task1 wait EVENTbit\_1 Event arrival = 2

如果要 Task1 不停的执行，那么 Task2 要不停的发送事件

```

void Task2(void)
{
    Event_Handle = xEventGroupCreate(); //创建事件

    if(Event_Handle != NULL)
        debug_printf("Event success\r\n"); //事件创建成功
    else
        debug_printf("Event failed\r\n");
    while(1)
    {
        xEventGroupSetBits(Event_Handle,EVENTbit_1); //放入循环，向事件组第 1 位置 1
        vTaskDelay(1000);
    }
}

```

XXXZZZZZ  
Event success  
Task1 wait EVENTbit\_1 Event arrival = 0x2  
Task1 wait EVENTbit\_1 Event arrival = 0x2

## 多事件触发才能执行案例

```
void Task1(void)
{
    EventBits_t EventValue; //获取事件等待返回的结果
    while(1)
    {
        EventValue = xEventGroupWaitBits(Event_Handle,EVENTbit_0|EVENTbit_1,pdTRUE,pdTRUE,portMAX_DELAY);
        //阻塞等待事件 1 EVENTbit_1, 和事件 0 EVENTbit_0 同时触发才结束阻塞

        debug_printf("Task1 wait EVENTbit_1 Event arrival = 0x%x\r\n",EVENTbit_1);
        //等待事件 1 EVENTbit_1 发生成功, 执行打印
        vTaskDelay(1000);
    }
}

void Task2(void)
{
    Event_Handle = xEventGroupCreate(); //创建事件

    if(Event_Handle != NULL)
        debug_printf("Event success\r\n"); //事件创建成功
    else
        debug_printf("Event failed\r\n");

    while(1)
    {
        xEventGroupSetBits(Event_Handle,EVENTbit_1); //向事件组第 1 位置 1
        vTaskDelay(1000);
    }
}

XXXZZZZZ
Event success
█
```

就算 Task2 发送事件也没有用，因为只发送了 EVENTbit\_1 事件 1 没有发送事件 0

```
void Task2(void)
{
    Event_Handle = xEventGroupCreate(); //创建事件

    if(Event_Handle != NULL)
        debug_printf("Event success\r\n"); //事件创建成功
    else
        debug_printf("Event failed\r\n");

    while(1)
    {
        xEventGroupSetBits(Event_Handle,EVENTbit_0 | EVENTbit_1); //向事件组第 0 位置 1, 第 1 位置 1, 发送两个事件
        vTaskDelay(1000);
    }
}

XXXZZZZZ
Event success
Task1 wait EVENTbit_1 Event arrival = 0x2
█
```

## 多事件也可以多个任务分开来发送

```
static EventGroupHandle_t Event_Handle = NULL; //事件句柄
#define EVENTbit_0 (1<<0)
#define EVENTbit_1 (1<<1)
#define EVENTbit_2 (1<<2)

void Task3(void)
{
    while(1){
        xEventGroupSetBits(Event_Handle,EVENTbit_0); //向事件组第 0 位置 1
        vTaskDelay(1000);
    }
}

void Task1(void)
{
    EventBits_t EventValue; //获取事件等待返回的结果
    while(1)
    {
        EventValue = xEventGroupWaitBits(Event_Handle,EVENTbit_0|EVENTbit_1,pdTRUE,pdTRUE,portMAX_DELAY);
        //阻塞等待事件 1 EVENTbit_1, 和事件 0 EVENTbit_0 同时触发才结束阻塞

        debug_printf("Task1 wait EVENTbit_1 Event arrival = 0x%x\r\n",EventValue);
        //打印有多少个事件被置 1, 用 16 进制表示, 可以翻译成 2 进制
        //等待事件 1 EVENTbit_1 发生成功, 执行打印
        vTaskDelay(1000);
    }
}

void Task2(void)
{
    Event_Handle = xEventGroupCreate();//创建事件

    if(Event_Handle != NULL)
        debug_printf("Event success\r\n");//事件创建成功
    else
        debug_printf("Event failed\r\n");

    while(1)
    {
        xEventGroupSetBits(Event_Handle,EVENTbit_1); //向事件组第 1 位置 1
        vTaskDelay(1000);
    }
}

XXXZZZZZ
Event success
Task1 wait EVENTbit_1 Event arrival = 0x2
```

## 软件定时器

软件定时器只适用于 1ms 毫秒左右的定时单位，对定时精度要求不高的可以用软件定时器。  
定时要求高的，比如 1us 微妙的定时就不适合软件定时器。

在 FreeRTOSconfig.h 中启动定时器配置选项

```
#define configUSE_TIMERS 1 //启动软件定时器  
#define configTIMER_TASK_STACK_DEPTH 128 //定时器任务堆栈大小  
#define configTIMER_TASK_PRIORITY (configMAX_PRIORITIES - 1) //软件定时器优先级最大值-1,  
优先级最高  
#define configTIMER_QUEUE_LENGTH 10 //软件定时器队列长度  
  
typedef void * TimerHandle_t //定义定时器句柄  
typedef void (*TimerCallbackFunction_t)( TimerHandle_t xTimer ) //定时器回调函数类型
```

```
TimerHandle_t xTimerCreate(const char * const pcTimerName,  
                           const TickType_t xTimerPeriodInTicks,  
                           const UBaseType_t uxAutoReload,  
                           void * const pvTimerID,  
                           TimerCallbackFunction_t pxCallbackFunction ) //创建定时器
```

pcTimerName: 给定时器取名字

xTimerPeriodInTicks: 定时器延时周期，最小单位 1ms 毫秒

uxAutoReload: 定时器工作模式 pdTRUE 为周期模式，就是定时器一直定时执行，pdFALSE 为单次模式，定时器只定时执行一次。

pvTimerID: 为定时器分配一个索引 ID，随便取，只要不重复就行

pxCallbackFunction: 定时器回调函数，定时时间到，执行的函数体

xTimerStart( xTimer, xTicksToWait ) //启动定时器

xTimer: 传入定时器句柄

xTicksToWait: 默认值为 0，定时器为正常工作模式

xTimerStartFromISR( xTimer, pxHigherPriorityTaskWoken ) //中断中启动定时器，该函数是 xTimerStart 中断版本

BaseType\_t xTimerStop( TimerHandle\_t xTimer, TickType\_t xBlockTime ) //软件定时器停止函数

xTimer: 软件定时器句柄，要停哪一个定时器就填入哪一个定时器句柄

xBlockTime: 用户指定超时时间，默认填 0 就是马上停止定时器工作。

xTimerStopFromISR()是函数 xTimerStop()的中断版本，在中断函数中停止软件定时器，自行查阅用法。

## 代码例程

```
#include "event_groups.h" //加入事件头文件，事件也使用了软件定时器，所以包含事件就包含了软件定时器
static TimerHandle_t Stimir1_Handle =NULL; //定义软件定时器 1 句柄
static TimerHandle_t Stimir2_Handle =NULL; //定义软件定时器 2 句柄

static void Stimir1_Callback(void* parameter) //软件定时器 1 回调函数
{
    debug_printf("Stimir1_Callback\r\n");
}

static void Stimir2_Callback(void* parameter) //软件定时器 2 回调函数
{
    debug_printf("Stimir2_Callback\r\n");
}

int main(void)
{
    BaseType_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为：0~15

    //优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

    //都统一用这个优先级分组，千万不要再分组，切忌。

    printf("xxxxxxxx\r\n");
    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1，因为 debug_printf 函数最后一行才会给 semDebug 写 1

    Stimir1_Handle = xTimerCreate("SFtimer1", //给软件定时器命名
                                (TickType_t)5000, //定时器周期 5000(tick) 就是 5000ms 毫秒
                                (UBaseType_t)pdTRUE, //周期模式
                                (void*)1, //为定时器分配一个索引 ID，随便写，不重复就是
                                (TimerCallbackFunction_t)Stimir1_Callback //回调函数
                            );
    xTimerStart(Stimir1_Handle,0); //启动定时器 1

    Stimir2_Handle = xTimerCreate("SFtimer2", //给软件定时器命名
                                (TickType_t)1000, //定时器周期 1000(tick) 就是 1000ms 毫秒
                                (UBaseType_t)pdTRUE, //周期模式
                                (void*)1, //为定时器分配一个索引 ID，随便写，不重复就是
                                (TimerCallbackFunction_t)Stimir2_Callback //回调函数
                            );
    xTimerStart(Stimir2_Handle,0); //启动定时器 2
    vTaskStartScheduler(); /* 就算不创建任务，也要启动任务，开启调度，因为任务调度是个死循环，会在里面无限执行 FreeRTOS 里面各种函数 */
}
```

```
xxxxxxxx
Stimir2_Callback
Stimir2_Callback
Stimir2_Callback
Stimir2_Callback
Stimir1_Callback
Stimir2_Callback
Stimir1_Callback
Stimir2_Callback
Stimir2_Callback
Stimir2_Callback
Stimir2_Callback
Stimir1_Callback
Stimir2_Callback
```

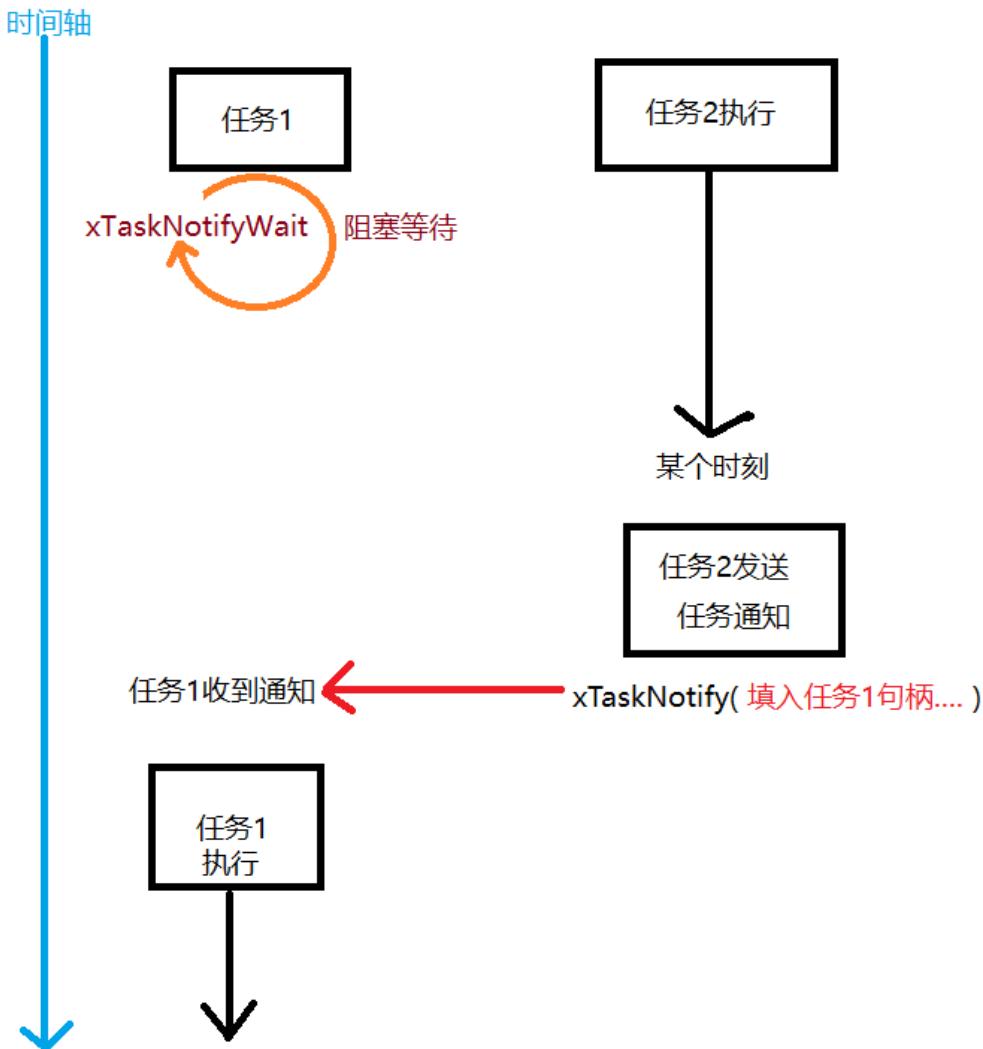
Stimir2\_Callback 1 秒执行 1 次 执行 5 次，Stimir1\_Callback 就会执行 1 次，证明 Stimir1\_Callback 是 5 秒 1 次

## 任务通知

任务通知是在 FreeRTOS 8.2.0 版本才有的，以前版本没有，所以使用时要注意版本。

任务通知可以代替**二值信号量**，**计数信号量**，**事件组**，**消息队列**。但是不能代替互斥锁机制。

```
#define configUSE_TASK_NOTIFICATIONS 1 //任务通知开启
```



其实和信号量，队列一样的，只是任务通知少写些代码，是新的技术，效率比以前的信号量，消息队列提高了40%，就是这么回事

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify , uint32_t ulValue , eNotifyAction eAction )
```

//发送带参数的通知给指定的任务

`xTaskToNotify`：传入需要接受本通知的任务句柄

`ulValue`: 发送参数给指定任务

eAction: 指定任务收到通知后是怎么处理的 ?

**eNoAction** 指定任务接收到通知后，不接收发给指定任务的参数，返回 pdFalse

`eSetValueWithOverwrite` 指定任务接收到通知后，强制将参数发送给指定任务，指定任务必须接收。

`eSetValueWithoutOverwrite` 指定任务接收通知后，如果指定任务本身的参数值没有取值，那么久丢弃发送过来的参数值，返回 `pdFALSE`。如果指定任务的参数已经被获取了，那么指定任务就接收发送过来的参数，返回 `pdTRUE`

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit,  
                           uint32_t *pulNotificationValue, TickType_t xTicksToWait )
```

//阻塞程序，接收带参数的任务通知，放弃阻塞，继续向下执行

`ulBitsToClearOnEntry`: 在进入任务前，将接收的参数哪些位清 0。如果设置成 0x01，比如 `pulNotificationValue` 接收到 0xffffffff，那么就是 111111111111111111111111111111。

```

void Task1(void)
{
    uint32_t RecvValue = 0;
    BaseType_t ret = pdFALSE;

    while(1)
    {
        ret = xTaskNotifyWait(0,0xFFFFFFFF,&RecvValue,portMAX_DELAY);
        if(ret == pdTRUE)
            debug_printf("Task1 Recv success = %d \r\n",RecvValue);
        else
            debug_printf("Task1 Recv failed \r\n");
    }
}

void Task2(void)
{
    BaseType_t status;
    uint32_t send = 20;

    while(1)
    {
        status = xTaskNotify(Task1TCB,send,eSetValueWithoutOverwrite); //发送任务通知给任务1
        if(status == pdPASS)
            debug_printf("Task2 send success\r\n"); //发送给 Task1 的数据被获取
        else
            debug_printf("Task2 send failed \r\n ");

        vTaskDelay(1000);
    }
}

```

```
XXXXXXXXXX  
Task2 send success  
Task1 Recv success = 20  
Task2 send success  
Task1 Recv success = 20
```

这就是 Task2 发送通知, Task1 接收通知, 并且得到通知值

```

void Task2(void)
{
    BaseType_t status;
    uint32_t send = 20;

    while(1)
    {
        //status = xTaskNotify(Task1TCB,send,eSetValueWithoutOverwrite); //发送任务通知给
        if(status == pdPASS)
            debug_printf("Task2 send success\r\n"); //发送给Task1的数据被获取
        else
            debug_printf("Task2 send failed \r\n ");

        vTaskDelay(1000);
    }
}

```

如果 Task2 不发送通知，那么 Task1 就只有阻塞，和消息队列一样的逻辑，只是任务通知不用定义太多参数，就在任务里面直接调用填参数即可。

```

xxxxzzzz
Task2 send failed
Task2 send failed
Task2 send failed
Task2 send failed

```

任务 1 阻塞

**BaseType\_t XTaskNotifyGive(TaskHandle\_t xTaskToNotify)** //发送二值信号量类型的任务通知  
**xTaskToNotify:** 传入发送给指定任务的任务句柄

**uint32\_t ulTaskNotifyTake( BaseType\_t xClearCountOnExit, TickType\_t xTicksToWait )**  
**//接收二值信号量类型的任务通知**

**XClearCountOnExit:** 设置为 pdTRUE 时，阻塞函数 ulTaskNotifyTake 退出前可以将任务通知值清 0，实现二值信号量。  
 设置为 pdFALSE 时，阻塞函数 ulTaskNotifyTake 退出前，任务通知值减 1，实现计数信号量。

**xTicksToWait:** 设置阻塞时间

**uint32\_t 返回计数值**

```

void Task1(void)
{
    uint32_t NotifyValue = 0;
    BaseType_t ret = pdFALSE;

    while(1)
    {
        ret = ulTaskNotifyTake(pdTRUE,portMAX_DELAY); //接收二值信号量类的任务通知
        if(ret == pdTRUE)
            debug_printf("Task1 return Notify value = %d \r\n",NotifyValue);
        else
            debug_printf("Task1 return failed \r\n");
    }
}

void Task2(void)
{
    BaseType_t status;
    uint32_t send = 20;
    while(1)
    {
        status = xTaskNotifyGive(Task1TCB); //发送类似二值信号量的任务通知给任务 1
        if(status == pdPASS)
            debug_printf("Task2 send success\r\n"); //发送二值任务通知给 Task1
        else
            debug_printf("Task2 send failed \r\n ");
        vTaskDelay(1000);
    }
}

```

```

xxxxzzzz
Task2 send success
Task1 return Notify value = 0
Task2 send success
Task1 return Notify value = 0
Task2 send success
Task1 return Notify value = 0
Task2 send success
Task1 return Notify value = 0
Task2 send success
Task1 return Notify value = 0

```

任务 1 接收任务 2 的通知，因为是二值  
 信号量模式，所以只有通知功能

```

ulTaskNotifyTake( pdFALSE , 0 ); //将接收任务改成计数信号量任务通知模式, pdFALSE

void Task1(void)
{
    uint32_t NotifyValue = 0; //计算 Task2 已经发送了多少个信号了, 但是 Task1 还没有去获取
    int i = 0;
    for(i = 0; i<10; i++) //先让 Task2 发 10 次任务通知
    {
        vTaskDelay(1000);
    }
    while(1)
    {
        NotifyValue = ulTaskNotifyTake(pdFALSE,0); //接收二值信号量类的任务通知, 应该累计了 10 个通知返回
        if(NotifyValue > 0)
            debug_printf("Task1 return Notify value = %d \r\n",NotifyValue);
        else
            debug_printf("Task1 return failed \r\n");
        vTaskDelay(1000);
    }
}

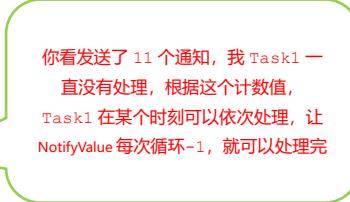
void Task2(void)
{
    BaseType_t status;
    uint32_t send = 20;

    while(1)
    {
        status = xTaskNotifyGive(Task1TCB); //发送类似二值信号量的任务通知给任务 1
        if(status == pdPASS)
            debug_printf("Task2 send success\r\n");
        else
            debug_printf("Task2 send failed \r\n");
        vTaskDelay(1000);
    }
}

```

XXXXXXXXXX

Task2 send success  
Task1 return Notify value = 11  
Task2 send success  
Task1 return Notify value = 11  
Task2 send success  
Task1 return Notify value = 11  
Task2 send success  
Task1 return Notify value = 11  
Task2 send success  
Task1 return Notify value = 11  
Task2 send success  
Task1 return Notify value = 11



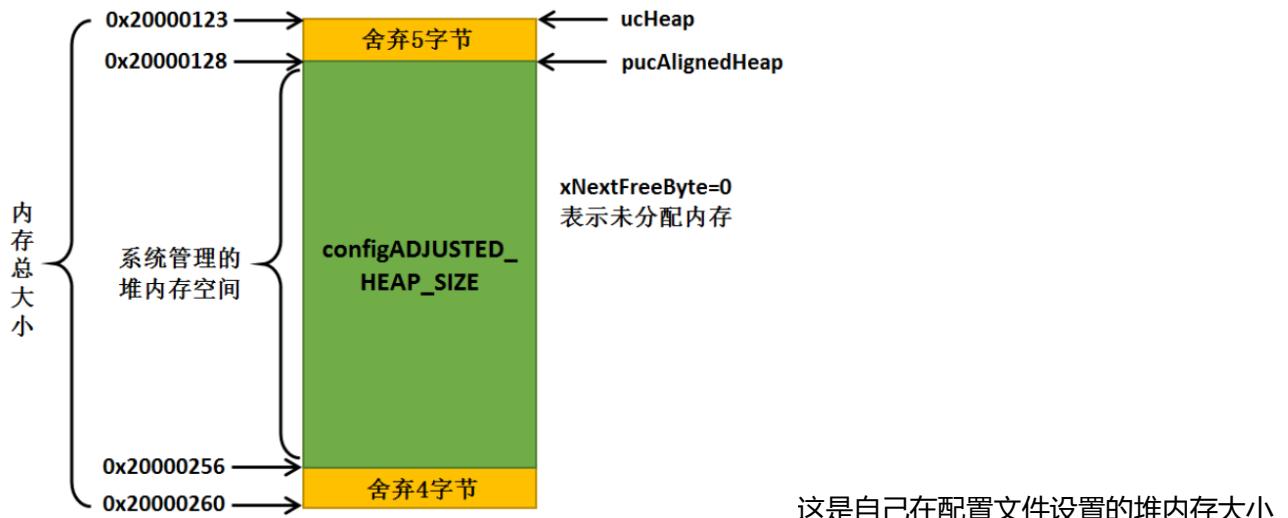
你看发送了 11 个通知, 我 Task1 一直没处理, 根据这个计数值, Task1 在某个时刻可以依次处理, 让 NotifyValue 每次循环-1, 就可以处理完

任务通知还实现了事件组的方式, 我这里就不写了, 我就用传统的事件组一样的。

## 内存管理

FreeRTOS 内存申请 4 种方式

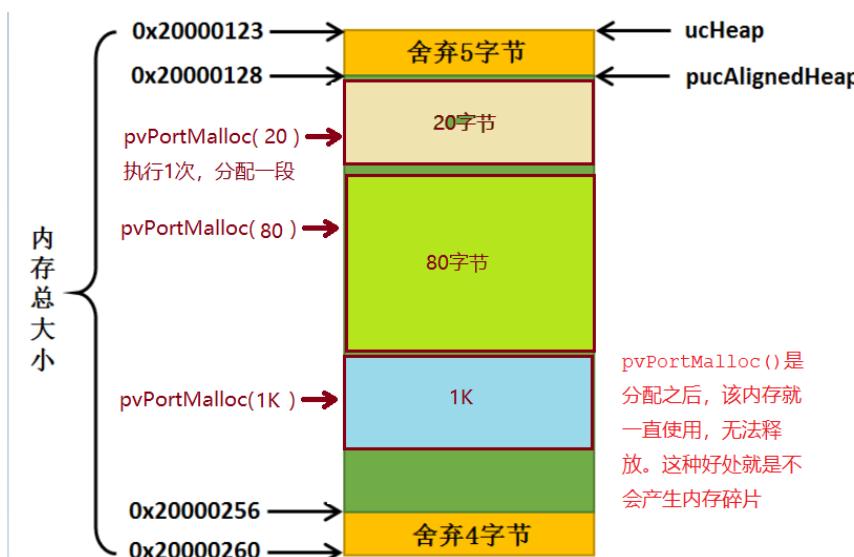
### heap\_1.c



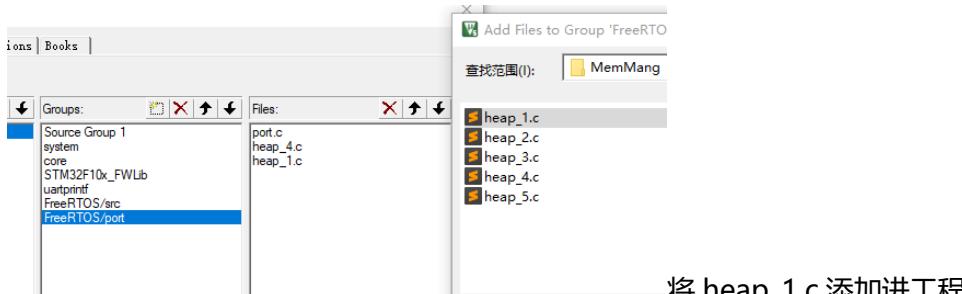
就是前面在 FreeRTOSconfig.h 里面设置的

```
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 16 * 1024 ) ) //系统总共的堆大小16K
```

```
void *pvPortMalloc( size_t xWantedSize ) //内存一次性分配函数
```



所以 heap\_1 适合那些创建好的任务，用于不会删除的任务，使用这种内存方法是比较好的。还有就是变量固定，变量都全部分配好了，不需要删除，只需要使用就行了，类似 51 单片机这种应用的。



在 portmacro.h 中

```
#define portBYTE_ALIGNMENT 8
```

如果是填的 8，那么 `pvPortMalloc` 内存分配就是 8 字节对齐

如果你觉得内存不够，需要外接 SRAM 来存储，那么可以在源码里面 heap\_1.c

```
#if( configAPPLICATION_ALLOCATED_HEAP == 1 )
    /* The application writer has already defined the
       heap - probably so it can be placed in a special
       extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
#else
    static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
#endif /* configAPPLICATION_ALLOCATED_HEAP */
```

定义全局数组的时候，在 `static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ]` 后面加 `_attribute_ & 0x60000`, 指定 SRAM 的起始地址 0x60000，那么你的堆内存数据就放在 SRAM 里面了。也可能是其它地址，根据芯片来决定。

## heap\_2.c



我需要200字节  
那么就刚好分配  
200字节给我



200字节使用完，我  
释放回内存，但是我  
发现这200字节无法  
合途回原内存



如果再申请300字节

看到没有，申请一次少1次

所以不断的分配和释放就会导致总内存越来越少，内存碎片越来越多。

所以 heap2 适合用在反复删除队列，任务，信号量的应用中。

不能用于哪些随机分配内存大小的应用。

## Heap\_3.c

封装了 C 标准库的 malloc() 和 free()，所以 heap3 是常规操作。

```
Heap_Size EQU 0x00000200
```

Heap\_3 主要就是要 STM32 自己的堆内存大小要设置得大。因为 heap\_3 用的不是全局数组，而是用的系统本身的堆。

## Heap\_4.c

使用最多的就是 heap4



```
size_t xPortGetFreeHeapSize( void ) //获取当前堆内存剩余大小
```

size\_t: 返回内存剩余大小，用 uint32\_t 变量来接收返回值。

```
void *pvPortMalloc( size_t xWantedSize ) //内存分配函数，按字节分配
```

xWantedSize: 分配多少个字节

void \* 返回分配内存的首地址

```
void Task2(void)
{
    int i = 0;
    uint32_t GetMemSize = 0;

    while(1)
    {
        GetMemSize = xPortGetFreeHeapSize(); //获取堆内存剩余大小
        debug_printf("Task2 malloc size = %d\r\n", GetMemSize);

        Test_Ptr = pvPortMalloc(1024); //申请1k字节大小的内存
        if(NULL != Test_Ptr)
            debug_printf("Task2 malloc success\r\n"); //堆内存分配成功
        else
            debug_printf("Task2 malloc failed \r\n");

        for(i = 0; i < 250; i++)
        {
            Test_Ptr[i] = i;
        }
        vTaskDelay(1000);
    }
}
```

你看右图，堆内存，每循环一次执行 pvPortMalloc，总内存就会少 1024 字节，直到总 15k 内存全部分配完，报内存分配错误

```
xxxxxxxx
Task2 malloc size = 15056
Task2 malloc success
Task2 malloc size = 14024
Task2 malloc success
Task2 malloc size = 12992
Task2 malloc success
Task2 malloc size = 11960
Task2 malloc success
Task2 malloc size = 10928
Task2 malloc success
Task2 malloc size = 9896
Task2 malloc success
Task2 malloc size = 8864
Task2 malloc success
Task2 malloc size = 7832
Task2 malloc success
Task2 malloc size = 6800
Task2 malloc success
Task2 malloc size = 5768
Task2 malloc success
Task2 malloc size = 4736
Task2 malloc success
Task2 malloc size = 3704
Task2 malloc success
Task2 malloc size = 2672
Task2 malloc success
Task2 malloc size = 1640
Task2 malloc success
Task2 malloc size = 608
Task2 malloc failed
```

我分配的堆内存是全局的，那么其它任务是否使用到该内存了呢？

任务 2 申请内存，写内存，任务 1 读内存，打印内存

```
uint8_t *Test_Ptr = NULL; //全局变量，用来做堆内存

void Task2(void) //任务 2 分配内存，写内存
{
    int i = 0;
    uint32_t GetMemSize = 0;

    GetMemSize = xPortGetFreeHeapSize(); //获取堆内存剩余大小
    debug_printf("Task2 malloc size = %d\r\n", GetMemSize);

    Test_Ptr = pvPortMalloc(1024); //申请 1k 字节大小的内存
    if(NULL != Test_Ptr)
        debug_printf("Task2 malloc success\r\n"); //堆内存分配成功
    else
        debug_printf("Task2 malloc failed \r\n");
    for(i = 0; i < 250; i++)
    {
        Test_Ptr[i] = i;
    }

    while(1)
    {
        vTaskDelay(1000);
    }
}

void Task1(void) //任务 1 读内存
{
    int i = 0;
    for(i = 0; i < 250; i++)
    {
        debug_printf("Task1 Test_Ptr %d = %d \r\n", i, Test_Ptr[i]);
    }

    while(1)
    {
        vTaskDelay(1000);
    }
}
```

void vPortFree( void \*pv ) //内存释放

```
uint8_t *Test_Ptr = NULL; //全局变量，用来做堆内存

void Task1(void)
{
    uint32_t GetMemSize = 0;
    int i = 0;
    GetMemSize = xPortGetFreeHeapSize();
    debug_printf("Task1 free before = %d\r\n", GetMemSize); //释放内存之前

    vPortFree(Test_Ptr); //释放内存
    GetMemSize = xPortGetFreeHeapSize(); //释放内存之后
    debug_printf("Task1 free later = %d\r\n", GetMemSize);
    for(i = 0; i < 250; i++)
    {
        debug_printf("Task1 Test_Ptr %d = %d \r\n", i, Test_Ptr[i]);
    }

    while(1)
    {
        vTaskDelay(1000);
    }
}
```

```
xxxxxxxx
Task2 malloc size = 15056
Task2 malloc success
Task1 Test_Ptr 0 = 0
Task1 Test_Ptr 1 = 1
Task1 Test_Ptr 2 = 2
Task1 Test_Ptr 3 = 3
Task1 Test_Ptr 4 = 4
Task1 Test_Ptr 5 = 5
Task1 Test_Ptr 6 = 6
Task1 Test_Ptr 7 = 7
Task1 Test_Ptr 8 = 8
Task1 Test_Ptr 9 = 9
Task1 Test_Ptr 10 = 10
Task1 Test_Ptr 11 = 11
Task1 Test_Ptr 12 = 12
Task1 Test_Ptr 13 = 13
Task1 Test_Ptr 14 = 14
Task1 Test_Ptr 15 = 15
Task1 Test_Ptr 16 = 16
Task1 Test_Ptr 17 = 17
Task1 Test_Ptr 18 = 18
Task1 Test_Ptr 19 = 19
Task1 Test_Ptr 20 = 20
Task1 Test_Ptr 21 = 21
```

任务 1 成功获取到任务  
2 分配的内存数据

```
xxxxxxxx
Task2 malloc size = 15056
Task2 malloc success
Task1 free before = 14024
Task1 free later = 15056
Task1 Test_Ptr 0 = 0
Task1 Test_Ptr 1 = 1
Task1 Test_Ptr 2 = 2
Task1 Test_Ptr 3 = 3
Task1 Test_Ptr 4 = 4
Task1 Test_Ptr 5 = 5
Task1 Test_Ptr 6 = 6
Task1 Test_Ptr 7 = 7
Task1 Test_Ptr 8 = 8
Task1 Test_Ptr 9 = 9
Task1 Test_Ptr 10 = 10
```

从内存容量来看，内存确实被释放了，但是从打印结果看貌似内存没有清 0

## 中断管理，主要是FreeRTOS 处理硬件中断的使用



uint32\_t taskENTER\_CRITICAL\_FROM\_ISR() //进入临界区

返回值属于临界值，用于退出临界区的时候使用

taskEXIT\_CRITICAL\_FROM\_ISR( uint32\_t ) //退出临界区

传入进入临界区的返回值，用于解锁退出临界区

xQueueSendFromISR(..., ..., ...) //中断里发送消息队列函数使用，在消息队列章节有介绍

代码例程：

```
void key_Interrupt_init(void) //PC5 按键中断初始化
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,ENABLE); //使能 PORTC 时钟

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;//PC5
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; //设置成上拉输入
    GPIO_Init(GPIOC, &GPIO_InitStructure); //初始化 GPIOC

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO,ENABLE); //使能 PORTC 复用时钟
    EXTI_InitStructure EXTI_Line = EXTI_Line5;//PC5 就是中断线
    EXTI_InitStructure EXTI_Mode = EXTI_Mode_Interrupt; //外部中断
    EXTI_InitStructure EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
    EXTI_InitStructure EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    GPIO_EXTILineConfig(GPIO_PortSourceGPIOC,GPIO_PinSource5); //PC5 作为中断线
    /*因为 FreeRTOS 初始化分配了中断组，所以其它任何时候都再分配中断组，这是 FreeRTOS 要求*/
    NVIC_InitStructure.NVIC_IRQChannel = EXTI9_5_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2; //设置响应优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}
```

QueueHandle\_t Test\_Queue =NULL; //消息队列句柄是全局的，方便硬件中断写入，任务 2 接收

```

void Task2(void)
{
    uint32_t data; //接收消息队列数据
    BaseType_t xReturn = pdPASS;
    while(1)
    {
        xReturn = xQueueReceive( Test_Queue , &data , portMAX_DELAY );

        if (pdPASS == xReturn)
            debug_printf("Task2 get PC5 key for data = %d\r\n",data);
        else
            debug_printf("Task2 get PC5 key failed.. \r\n");
    }
}

uint32_t sendData = 50; //中断要发送的数据

void EXTI9_5_IRQHandler(void)
{
    uint32_t ulReturn; //临界段保护

    ulReturn = taskENTER_CRITICAL_FROM_ISR(); //进入临界段
    if(EXTI_GetITStatus(EXTI_Line5) != RESET) //判断中断是否发生
    {
        //debug_printf("interrupt PC5 \r\n"); //调试的时候可以使用，最好不要在临界区使用
        xQueueSendFromISR(Test_Queue,&sendData,NULL); //使用中断消息发送函数发送消息
        EXTI_ClearFlag(EXTI_Line5); //清除中断标志位
        EXTI_ClearITPendingBit(EXTI_Line5);
    }
    taskEXIT_CRITICAL_FROM_ISR( ulReturn ); //退出临界段
}

int main(void)
{
    BaseType_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 ); //STM32 中断优先级分组为 4，即 4bit 都用来表示抢占优先级，范围为： 0~15

    //优先级分组只需要分组一次即可，以后如果有其他的任务需要用到中断，

    //都统一用这个优先级分组，千万不要再分组，切忌。
    key_Interrupt_init(); //按键中断初始化

    Test_Queue = xQueueCreate(4,sizeof(int)); //消息队列 4 个空间，每个空间放 1 个 4 字节 int 数据

    printf("xxxxxxxx\r\n");
    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1，因为 debug_printf 函数最后一行才会给 semDebug 写 1
    创建任务 2......
}

xxxxxxxx
Task2 get PC5 key for data = 50

```

任务 2 接收按键触发的消息队列。

## CPU 使用率统计

在 FreeRTOSconfig.h 中加入以下两个宏

```
#define configGENERATE_RUN_TIME_STATS 1 //CPU 利用率计算需要开启运行时间统计
#define configUSE_TRACE_FACILITY 1 //启动可视化跟踪调试
```

```
#include "stm32f10x.h"
extern volatile uint32_t CPU_RunTime; //定义全局 CPU 时间
#define configUSE_STATS_FORMATTING_FUNCTIONS 1 //格式化启动
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() (CPU_RunTime = 0ul)
#define portGET_RUN_TIME_COUNTER_VALUE() CPU_RunTime
```

配置完成后一定要去某个 C 文件外部实现 `volatile uint32_t CPU_RunTime = 0UL;` 这样才能编译过  
最好在高级定时器中断外定义 `volatile uint32_t CPU_RunTime = 0UL;` 全局变量

高级度定时要求：比如系统节拍是 1000hz，那么定时器节拍数就要是 10000~20000hz，我选用 20000hz 做 CPU 使用率统计，因为我定义的变量是 32 位，那么 59.6 分钟之后就会溢出，导致 59.6 分钟之后计算的 CPU 使用率不准，所以可以清 0 重算。

### 代码例程

```
//通用定时器中断初始化
//这里时钟选择为 APB1 的 2 倍，而 APB1 为 36M
//arr: 自动重装值。
//psc: 时钟预分频数
//这里使用的是定时器 3!
void TIM3_Init(u16 arr,u16 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //时钟使能
    TIM_TimeBaseStructure.TIM_Period = arr; //设置在下一个更新事件装入活动的自动重装载寄存器周期的值 计数到 5000 为 500ms
    TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置用来作为 TIMx 时钟频率除数的预分频值 10Khz 的计数频率
    TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TDDS = Tck_tim
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM 向上计数模式
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //根据 TIM_TimeBaseInitStruct 中指定的参数初始化 TIMx 的时间基数单位

    TIM_ITConfig( //使能或者失能指定的 TIM 中断
        TIM3, //TIM2
        TIM_IT_Update ,
        ENABLE //使能
    );
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn; //TIM3 中断
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //从优先级 3 级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ 通道被使能
    NVIC_Init(&NVIC_InitStructure); //根据 NVIC_InitStruct 中指定的参数初始化外设 NVIC 寄存器
    TIM_Cmd(TIM3, ENABLE); //使能 TIMx 外设
}

volatile uint32_t CPU_RunTime = 0UL;

void TIM3_IRQHandler(void) //TIM3 中断
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //检查指定的 TIM 中断发生与否:TIM 中断源
    {
        CPU_RunTime ++ ; //CPU 运行时间自加，很重要

        TIM_ClearITPendingBit(TIM3, TIM_IT_Update ); //清除 TIMx 的中断待处理位:TIM 中断源
    }
}
```

```

void TIM3_Int_Init(u16 arr,u16 psc);

int main(void)
{
    BaseType_t xReturn1,xReturn2,xReturn3,xReturn4,xReturn5;
    RCC_Configuration(); //初始化时钟
    USART_Config(115200); //初始化串口
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //STM32 中断优先级分组为 4, 即 4bit 都用来表示抢占优先级, 范围为: 0~15

    //优先级分组只需要分组一次即可, 以后如果有其他的任务需要用到中断,
    //都统一用这个优先级分组, 千万不要再分组, 切忌。

    TIM3_Int_Init(1,3599); //72M/20000=3600, 因为定时器内部+1 所以传 3599
    //20Khz 的计数频率, 计数到 1 大概为 0.00002(20us)

    printf("xxxxxxxx\r\n");
    semDebug = xSemaphoreCreateBinary(); //创建打印的二值信号量
    xSemaphoreGive(semDebug); //必须在初始化前就给先向 semDebug 变量里面写入 1, 因为 debug_printf 函数最后一行才会给 semDebug 写 1

    xReturn1 = xTaskCreate((TaskFunction_t)Task1, //任务函数地址
                          (const char*)"Task1",           //任务名
                          (uint32_t)1024,                //任务栈大小
                          (void*)NULL,                  //向函数传参
                          (UBaseType_t)1,                //优先级
                          (TaskHandle_t*)&Task1TCB); //任务控制块

    xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                          (const char*)"Task2",
                          (uint32_t)1024, //任务 2 栈大小要改成 1024, 因为在任务 2 函数中申请了 400 的数组已经大于以前的 128 字节栈了, 不然会死机
                          (void*)NULL,
                          (UBaseType_t)7,
                          (TaskHandle_t*)&Task2TCB); //因为任务 2 是发送消息的, 所以优先级一定要高于其它任务

    xReturn3 = xTaskCreate((TaskFunction_t)Task3,
                          (const char*)"Task3",
                          (uint32_t)1024,
                          (void*)NULL,
                          (UBaseType_t)3,
                          (TaskHandle_t*)&Task3TCB);

    启动任务 1, 任务 2, 任务 3, ....
}

void Task1(void)
{
    while(1)
    {
        debug_printf(" TASK1 ..... \r\n");
        vTaskDelay(1000);
    }
}

void Task3(void)
{
    while(1){
        debug_printf(" TASK3 ..... \r\n");
        vTaskDelay(500);
    }
}

```

```

void Task2(void)
{
    uint8_t CPU_RunInfo[400]; //保存任务运行时间信息
    //创建任务的时候记得修改任务栈大小，CPU_RunInfo 都已经超过 128 字节了

    while(1)
    {
        memset(CPU_RunInfo,0,400); //信息缓冲区清零
        vTaskList((char *)&CPU_RunInfo); //获取任务运行时间信息

        debug_printf("----- \r\n");
        debug_printf("TaskName      status   prioritj  stack  number\r\n");
        debug_printf("%s", CPU_RunInfo);
        debug_printf("----- \r\n");

        memset(CPU_RunInfo,0,400); //信息缓冲区清零
        vTaskGetRunTimeStats((char *)&CPU_RunInfo);
        debug_printf("TaskName  CPUcount UsageRate\r\n");
        debug_printf("%s", CPU_RunInfo);
        debug_printf("----- \r\n");
        vTaskDelay(1000);

    }
}

```

任务名 任务状态 优先级 任务剩余栈 序号

XXXXZZZZZZ				
TaskName	status	prioritj	stack	number
Task2	R	7	886	2
Task3	R	3	1014	3
Task1	R	1	1014	1
IDLE	R	0	119	4
Tmr Svc	S	31	101	5

TaskName  CPUcount UsageRate		
TaskName	CPUcount	UsageRate
Task2	0	<1%
Task3	0	<1%
Task1	0	<1%
IDLE	0	<1%
Tmr Svc	1	<1%

TASK3 .....				
TaskName	status	prioritj	stack	number
Task2	R	7	873	2
IDLE	R	0	115	4
Task1	B	1	974	1
Task3	B	3	974	3
Tmr Svc	S	31	101	5

TaskName  CPUcount UsageRate		
TaskName	CPUcount	UsageRate
Task2	454	4%
Task3	28	<1%
Task1	15	<1%
IDLE	9952	93%
Tmr Svc	1	<1%

TASK3 .....				
TaskName	status	prioritj	stack	number
Task2	R	7	873	2
IDLE	R	0	115	4
Task1	B	1	974	1
Task3	B	3	974	3
Tmr Svc	S	31	101	5

才启动系统，第 1 次是不准的

第 2 次开始就准了

## 查询系统堆大小，用于检测程序是否长时间正常运行

size\_t xPortGetFreeHeapSize( void ) //系统剩余堆大小检测

返回剩余堆大小， size\_t 采用%d 打印

#define configTOTAL\_HEAP\_SIZE (( size\_t )( 16 \* 1024 )) //系统总共的堆大小16K  
在 FreeRTOSconfig.h 中，系统总共的堆大小我定义的 16K

```
void Task2(void)
{
    size_t HeapRemainingSize = 0; //堆剩余大小

    while(1)
    {
        HeapRemainingSize = xPortGetFreeHeapSize(); //返回剩余堆大小
        debug_printf(" TASK2 HeapRemainingSize = %d \r\n", HeapRemainingSize);
        vTaskDelay(1000);
    }
}

xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                      (const char*)"Task2",
                      (uint32_t)1024,
                      (void*)NULL,
                      (UBaseType_t)7,
                      (TaskHandle_t*)&Task2TCB);
```

现在只启动 Task2 任务，这里分配的堆是 4K，因为 uint32\_t 是 4 字节，所以  $4 \times 1024 = 4K$

```
XXXXXXXX
TASK2 HeapRemainingSize = 12064
TASK2 HeapRemainingSize = 12064
TASK2 HeapRemainingSize = 12064
```

你看总堆大小 16K，运行一个 Task2 任务消耗 4K 的堆，还剩 12K 堆

我将任务堆改成 512 试试，uint32\_t 就是 4，那么  $4 \times 512 = 2048(2k)$

```
xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                      (const char*)"Task2",
                      (uint32_t)512,
                      (void*)NULL,
                      (UBaseType_t)7,
                      (TaskHandle_t*)&Task2TCB);
```

```
XXXXXXXX
TASK2 HeapRemainingSize = 14112
TASK2 HeapRemainingSize = 14112
TASK2 HeapRemainingSize = 14112
```

你看，16K 总堆只开辟了 2K 还剩 14K

下面进行多任务堆消耗实验

```

void Task1(void)
{
    while(1)
    {
        debug_printf(" TASK1 ..... \r\n");
        vTaskDelay(1000);
    }
}

void Task2(void)
{
    size_t HeapRemainingSize = 0; //堆剩余大小

    while(1)
    {
        HeapRemainingSize = xPortGetFreeHeapSize(); //返回剩余堆大小
        debug_printf(" TASK2 HeapRemainingSize = %d \r\n", HeapRemainingSize);
        vTaskDelay(1000);
    }
}

xReturn1 = xTaskCreate((TaskFunction_t)Task1, //任务函数地址
                      (const char*)"Task1", //任务名
                      (uint32_t)1024, //任务栈大小
                      (void*)NULL, //向函数传参
                      (UBaseType_t)1, //优先级
                      (TaskHandle_t*)&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                      (const char*)"Task2",
                      (uint32_t)1024,
                      (void*)NULL,
                      (UBaseType_t)7,
                      (TaskHandle_t*)&Task2TCB);

```

```

XXXXXX
TASK2 HeapRemainingSize = 7856
TASK1 .....
TASK2 HeapRemainingSize = 7856
TASK1 .....
TASK2 HeapRemainingSize = 7856
TASK1 .....
TASK2 HeapRemainingSize = 7856

```

你看，任务 2 消耗 4K，任务 1 消耗 4K，累计 8K， $16K - 8K = 8K$ ，还剩接近于 8K 的堆可以再开辟

```

xReturn1 = xTaskCreate((TaskFunction_t)Task1, //任务函数地址
                      (const char*)"Task1", //任务名
                      (uint32_t)1024, //任务栈大小
                      (void*)NULL, //向函数传参
                      (UBaseType_t)1, //优先级
                      (TaskHandle_t*)&Task1TCB); //任务控制块

xReturn2 = xTaskCreate((TaskFunction_t)Task2,
                      (const char*)"Task2",
                      (uint32_t)1024,
                      (void*)NULL,
                      (UBaseType_t)7,
                      (TaskHandle_t*)&Task2TCB);

xReturn3 = xTaskCreate((TaskFunction_t)Task3,
                      (const char*)"Task3",
                      (uint32_t)1024,
                      (void*)NULL,
                      (UBaseType_t)3,
                      (TaskHandle_t*)&Task3TCB);

```

启动任务 3 就还剩 3~4K 的堆

```

XXXXXX
TASK2 HeapRemainingSize = 3648
TASK3 .....
TASK1 .....
TASK3 .....
TASK2 HeapRemainingSize = 3648
TASK3 .....
TASK1 .....
TASK3 .....

```

确实最后只剩下 3K 堆可以再分配了

所以要做多线程项目，总堆的空间是越大越好，因为 FreeRTOS 里面的线程栈，用的是对空间代替的。

```

size_t xPortGetMinimumEverFreeHeapSize( void );
//查询堆最小空间，其实和剩余堆大小查询一样的功能
返回堆最小空间，size_t 采用%d 打印
void Task2(void)
{
    size_t HeapRemainingSize = 0; //堆剩余大小
    size_t HeapMixSpace = 0; //堆最小空间
    while(1)
    {
        HeapRemainingSize = xPortGetFreeHeapSize(); //返回剩余堆大小
        debug_printf(" TASK2 HeapRemainingSize = %d \r\n", HeapRemainingSize);

        HeapMixSpace = xPortGetMinimumEverFreeHeapSize(); //查询堆最小空间
        debug_printf(" TASK2 HeapMixSpace = %d \r\n", HeapMixSpace);
        vTaskDelay(1000);
    }
}

```

TASK2 HeapRemainingSize = 3648  
TASK2 HeapMixSpace = 3648  
TASK3 .....  
TASK1 .....

堆最小空间其实和堆剩余空间一样的功能

### void vApplicationMallocFailedHook( void ); //堆申请失败回调函数

这是一个回调函数，需要用户自己实现。如果配置文件中 configUSE\_MALLOC\_FAILED\_HOOK 设置为 1 的话，当堆分配内存失败时会调用此函数。  
用户可以在此函数中进行错误处理。

## 查询每个任务栈使用大小，用于检测系统是否长时间正常运行

UBaseType\_t uxTaskGetStackHighWaterMark( TaskHandle\_t xTask ) //查看当前任务栈还剩多少

xTask: 你要查询哪一个任务栈的剩余大小，就传入该任务句柄

UBaseType\_t: 返回该任务栈还剩多少，返回值不是计算的字节数，而是计算的 uint32\_t 个数，所以你要把返回结果 x4 才是字节

#define INCLUDE\_uxTaskGetStackHighWaterMark 1 //查询任务栈大小，需要将该宏置 1

```

static TaskHandle_t Task2TCB; 任务句柄
void Task2(void)
{
    unsigned portBASE_TYPE uxHighWaterMark; //本任务剩余栈还剩多少
    while(1)
    {
        uxHighWaterMark=uxTaskGetStackHighWaterMark( Task2TCB ); //查询任务栈大小返回
        debug_printf(" TASK2 STACK Size = %d \r\n", uxHighWaterMark); //任务2还剩多少栈空间
        vTaskDelay(1000);
    }
}

```

xxxxxxxx  
TASK2 STACK Size = 1014  
TASK2 STACK Size = 972  
TASK2 STACK Size = 972

任务开始是  $1041 \times 4 = 4164$ (4k)栈空间

后来函数调用，变量定义消耗了  $1014 - 972 = 42$ ，记住  $42 \times 4 = 168$  这才是字节 消耗了 168 字节栈空间

```

void Task2(void)
{
    char str1[512] = {0}; // 定义数组赋初值会消耗512字节，如果不赋值不消耗，证明了使用变量才消耗栈空间
    unsigned portBASE_TYPE uxHighWaterMark; // 本任务剩余栈还剩多少
    while(1)
    {
        uxHighWaterMark=uxTaskGetStackHighWaterMark( Task2TCB ); // 查询任务栈大小返回
        debug_printf(" TASK2 STACK Size = %d \r\n",uxHighWaterMark); // 任务2还剩多少栈空间
        vTaskDelay(1000);
    }
}

```

```

xxxxzzzzz
TASK2 STACK Size = 893
TASK2 STACK Size = 844
TASK2 STACK Size = 844
TASK2 STACK Size = 844

```

上一页说明了，任务启动剩  $972 - 844 = 128$ ，再  $128 \times 4 = 512$  字节，所以定义数组消耗了栈空间 512 字节，还剩  $844 \times 4 = 3376$  字节

现在  $972 - 844 = 128$ ，再  $128 \times 4 = 512$  字节，所以定义数组消耗了栈空间 512 字节，还剩  $844 \times 4 = 3376$  字节

```

void Task2(void)
{
    char str1[512] = {0}; // 定义数组赋初值会消耗512字节，如果不赋值不消耗，证明了使用变量才消耗栈空间
    char str2[512] = {0}; // 如果再定义个数组
    unsigned portBASE_TYPE uxHighWaterMark; // 本任务剩余栈还剩多少
    while(1)
    {
        uxHighWaterMark=uxTaskGetStackHighWaterMark( Task2TCB ); // 查询任务栈大小返回
        debug_printf(" TASK2 STACK Size = %d \r\n",uxHighWaterMark); // 任务2还剩多少栈空间
        vTaskDelay(1000);
    }
}

```

```

xxxxzzzzz
TASK2 STACK Size = 765
TASK2 STACK Size = 716

```

$972 - 716 = 256$ ,  $256 \times 4 = 1024$ ，两个数组消耗了 1K 的栈空间

```

void Task2(void)
{
    char str1[512] = {0}; // 定义数组赋初值会消耗512字节，如果不赋值不消耗，证明了使用变量才消耗栈空间
    char str2[512] = {0}; // 如果再定义个数组
    unsigned int XZZ1 = 50; // 我再定义个4字节变量
    unsigned portBASE_TYPE uxHighWaterMark; // 本任务剩余栈还剩多少
    while(1)
    {
        uxHighWaterMark=uxTaskGetStackHighWaterMark( Task2TCB ); // 查询任务栈大小返回
        debug_printf(" TASK2 STACK Size = %d \r\n",uxHighWaterMark); // 任务2还剩多少栈空间
        vTaskDelay(1000);
    }
}

```

```

xxxxzzzzz
TASK2 STACK Size = 765
TASK2 STACK Size = 716

```

定义 4 字节变量 1 个，太少了，看不出来，但是我发现只有定义数组才能看出来。`unsigned int XZZ1[1] = {0};` 这要就显示 714，消耗 8 字节。