FreeRTOS 源码解读

Nrush

如果您觉得此份文档对您有所帮助，

可以前往https://github.com/Nrusher/FreeRTOS-Book或https://gitee.com/nrush/FreeRTOS-Book进行 Star，给予作者鼓励!

2020 年 3 月 9 日

特别说明

- 本文档针对的是 FreeRTOS Kernel V10.2.1 源码的解读，涉及平台相关的函数均以 cortex-m3 为例，文中不再进行特殊声明。
- 由于文档还在更新中，可能会根据后续实际情况进行改动。

目录

第 1 章　List

链表这一数据结构是 FreeRTOS 的核心数据结构，有关任务调度、延时、阻塞、事件等操作都是通过对链表进行操作进而实现的。本章将详细分析源码文件 list.c，list.h 的内容，为后续的任务队列等的实现奠定基础。

1.1　链表及链表项结构体定义

FreeRTOS 使用的链表结构是环形的双向链表，其链表项 ListItem_t 的定义如下

```
1  struct xLIST_ITEM
2  {
3      // 第一个和最后一个成员值当configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES被使能的时候
           会被设定为一个固定值，用来检验一个列表项数据是否完整
4      listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
5      // 用来给链表排序的值，在不同的应用场景下有不同的含义。
6      configLIST_VOLATILE TickType_t xItemValue;
7      // 指向前一个链表项
8      struct xLIST_ITEM * configLIST_VOLATILE pxNext;
9      // 指向后一个链表项
10     struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
11     // 类似侵入式链表，指向包含该链表项的对象的地址
12     void * pvOwner;
13     // 指向拥有此链表项的链表
14     struct xLIST * configLIST_VOLATILE pxContainer;
15     listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
16 };
17 typedef struct xLIST_ITEM ListItem_t;
```

Listing 1.1: ListItem_t 的定义

这里如果使用 configLIST_VOLATILE，其会被替换为 volatile 关键字，volatile 关键字是让编译器不对该变量进行优化，所谓的优化可以理解为其在生成汇编时，若多次读取该变量时其可能会将变量值放入寄存器中以加快读取速度，而不是真正的读取，这使得当某个变量会快速变化时，优化后“读取”的值并不是变量真实的值。当使用 volatile 关键字时，其会强迫编译器每次使用该变量时都真正的对它进行一次读取。

```
1  #define configLIST_VOLATILE  volatile
```

Listing 1.2: 使用 configLIST_VOLATILE

在链表项的结构体体构造中值得注意的是 pvOwner 和 pxContainer 这两个成员变量。pvOwner 提供了一种可以快速访问由此链表项代表的对象的方法，pxContainer 则提供了一种快速访问其所属链表的方法。这种处理方式大大提高了链表在任务调度等应用中的处理速度，提高系统效率。

Linux 内核中的侵入式链表设计

Linux 内核中也有侵入式的链表的设计，在 Linux 中提供的链表项的定义为

```
1      struct list_head
2      {
3          struct list_head *next, *prev;
4      };
```

使用链表时只需要将其包含进定义的对象中即可

```c
struct node
{
    // 一些其它成员定义....
    char i;
    // 侵入式链表项
    struct list_head list_item;
    // 一些其它成员定义....
    char j;
};
```

在此它没有定义类似 ListItem_t 中 pxContainer 这样的成员变量，其获得包含该链表项的对象地址是通过一段巧妙的宏定义实现的

```c
#define offsetof(s,m) (size_t)&(((s *)0)->m

#define container_of(ptr, type, member)                    \
({                                                         \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);   \
    (type *)( (char *)__mptr - offsetof(type,member) );    \
})
```

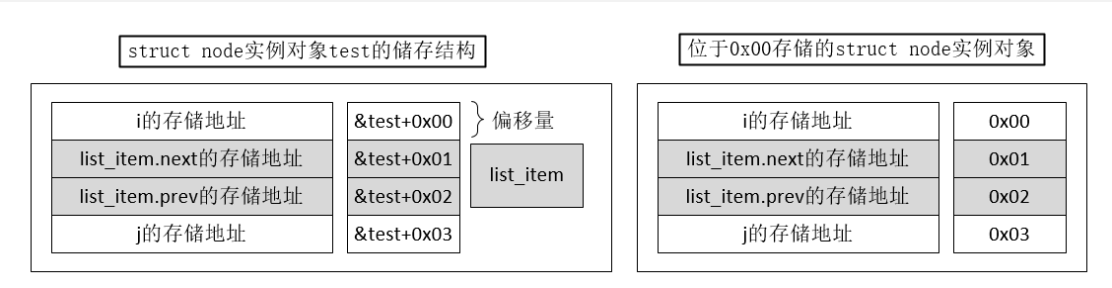
各输入参数的含义为

- ptr：结构体实例成员地址。
- type：结构体名。
- member：成员名。

使用示例

```c
// 包含链表项的对象实例
struct node test;
// 假设通过某种方式（如遍历）获得了链表项地址
struct list_head *list_item_add = &test.list_item;
// 计算对象地址
struct node *test_add = container_of(list_item_add,struct node,
    list_item);
// test_add即为包含list_item_add指向的链表项的对象地址
```

container_of() 的实现思路简单概括就是：将成员变量地址减去成员变量在结构体类型中的便宜量便是实例对象的存储地址。以 struct node 结构体为例，其实例 test 在内存中的存储方式如下图左侧所示。如何获得成员在结构体存储中的偏移量呢？下图右侧给出了解决方法，当 &test=0x00 时，其成员的地址便是所需要的偏移量。



回过头来看 offsetof() 宏，其所作的事就是获得偏移量。而 container_of() 宏中的

```c
(type *)( (char *)__mptr - offsetof(type,member) );
```

便是用成员地址减去偏移量来获得实例的地址。至于 container_of() 宏中的前一句

```c
const typeof( ((type *)0)->member ) *__mptr = (ptr);
```

实时上是起到一个类型检验的作用，拓展关键字 typeof 可以获得变量的类型，如果传入的 ptr 的类型与成员变量类型不符，那么编译器便会抛出警告，便于检查是否出错。注意 typeof 并不是标准 C 中的关键字，如果所用的编译器不支持，可以将第一句删除，将第二句中的 ___mptr 替换为 ptr，宏 container_of() 仍然是正确的。

个人认为，FreeRTOS 并未借鉴这一设计的一个原因可能是出于系统实时性的考虑，毕竟这样的操作是需要耗费一定时间的，而 FreeRTOS 内核中需要频繁使用这一功能，这样的设计必定会降低内核效率。另外 FreeRTOS 的目标平台一般运行速度相对较低，使得内核效率降低更为明显。

```
1  struct xMINI_LIST_ITEM
2  {
3      // 校验值
4      listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
5      // 排序值
6      configLIST_VOLATILE TickType_t xItemValue;
7      // 指向前一个链表项
8      struct xLIST_ITEM * configLIST_VOLATILE pxNext;
9      // 指向后一个链表项
10     struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
11 };
12 typedef struct xMINI_LIST_ITEM MiniListItem_t;
```

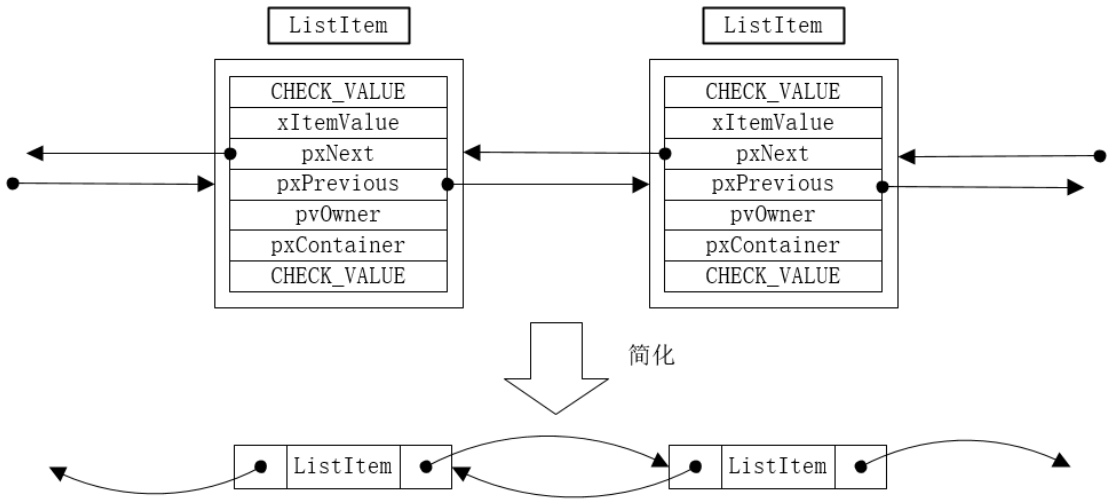
Listing 1.3: MiniListItem_t

链表 List_t 的定义

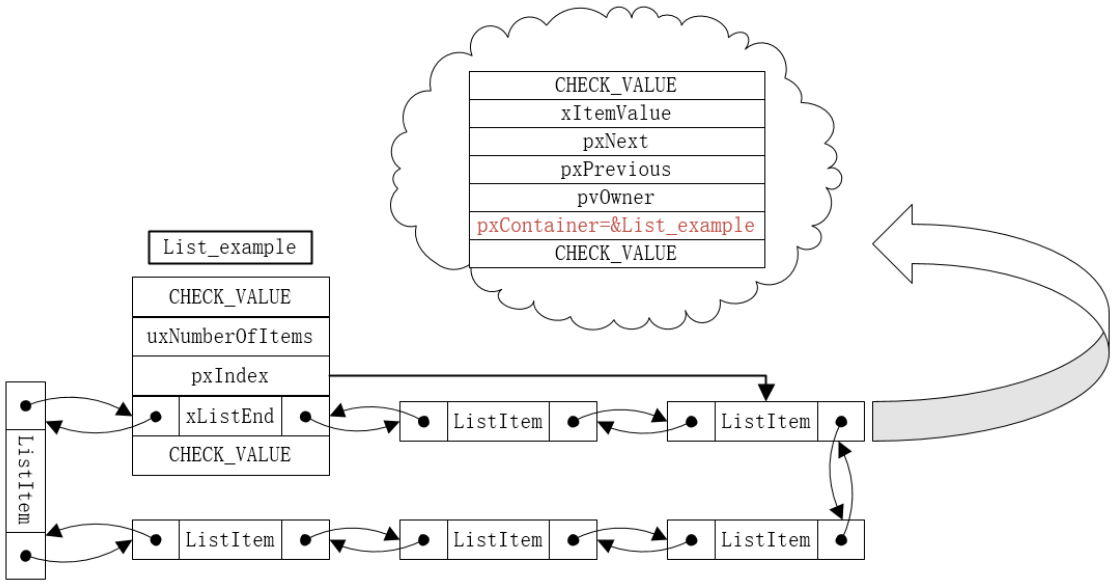
```
1  typedef struct xLIST
2  {
3      // 校验值
4      listFIRST_LIST_INTEGRITY_CHECK_VALUE
5      // 记录该链表里有多少成员
6      volatile UBaseType_t uxNumberOfItems;
7      // 用来遍历链表的指针，listGET_OWNER_OF_NEXT_ENTRY()会使它指向下一个链表项
8      ListItem_t * configLIST_VOLATILE pxIndex;
9      // 链表尾部，为节省内存使用Mini链表项
10     MiniListItem_t xListEnd;
11     // 校验值
12     listSECOND_LIST_INTEGRITY_CHECK_VALUE
13 } List_t;
```

Listing 1.4: List_t 的定义

链表中的链表项间的连接如下图所示



FreeRTOS 中链表的一般结构如下所示结构



1.2 链表的相关操作

1.2.1 初始化

链表项的初始化 vListInitialiseItem() 函数如下

```
1  void vListInitialiseItem( ListItem_t * const pxItem )
2  {
3      // 使该链表项不被任何链表拥有
4      pxItem->pxContainer = NULL;
5      // 写入校验值
6      listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
7      listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
8  }
```

Listing 1.5: vListInitialiseItem() 分析

链表项的初始化十分简单只是将 pxContainer 置为 NULL，设置一下校验值。

链表的初始化函数 vListInitialise() 如下所示

```
1  void vListInitialise( List_t * const pxList )
2  {
3      // step1 把当前指针指向链表尾部
```

```
4       pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );
5
6       // step2 设置排序值为最大，也就是保证xListEnd会被放在链表的尾部
7       pxList->xListEnd.xItemValue = portMAX_DELAY;
8
9       // step3 xListEnd的下一项和前一项都设置为尾部，表示链表为空。
10      pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd );
11      pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );
12
13      // step4 链表项数目为0
14      pxList->uxNumberOfItems = ( UBaseType_t ) 0U;
15
16      // step5 写入校验值，用于后续检验，为了保证链表结构体是正确的，没有被覆写
17      listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
18      listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
19  }
```

Listing 1.6: vListInitialise() 分析

1.2.2 插入

链表的插入操作有两个版本的函数，分别是 vListInsertEnd() 和 vListInsert()。其中 vListInsertEnd() 可以理解为是一种无序的插入方法，也就是直接在链表 pxIndex 指向的前一位置插入一个链表项，而 vListInsert() 是有序的插入方法，链表项将会按 xItemValue 的值进行升序插入到链表中去。先看 vListInsertEnd() 插入的实现

```
1   void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
2   {
3   ListItem_t * const pxIndex = pxList->pxIndex;
4
5       // step1 校验链表和链表项
6       listTEST_LIST_INTEGRITY( pxList );
7       listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
8
9       // step2 将链表项嵌入到pxIndex指向的链表项前
10      pxNewListItem->pxNext = pxIndex;
11      pxNewListItem->pxPrevious = pxIndex->pxPrevious;
12
13      // 调试测试用的函数，具体功能不知道，对代码逻辑理解无影响。
14      mtCOVERAGE_TEST_DELAY();
15
16      pxIndex->pxPrevious->pxNext = pxNewListItem;
17      pxIndex->pxPrevious = pxNewListItem;
18
19      // step3 记录链表项属于该链表
20      pxNewListItem->pxContainer = pxList;
21
22      // step5 记录链表中的链表项数目
23      ( pxList->uxNumberOfItems )++;
24  }
```

Listing 1.7: vListInsertEnd() 分析

vListInsert() 的实现比 vListInsertEnd() 稍微复杂一些，其先要查找插入位置再进行插入操作

```
1   void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem )
2   {
3   ListItem_t *pxIterator;
4   const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
5
6       // step1 校验链表和链表项
7       listTEST_LIST_INTEGRITY( pxList );
8       listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
9
10      // step2 寻找插入位置
11      if( xValueOfInsertion == portMAX_DELAY )
12      {
```

```
13          // 如果链表项的排序数最大，直接在尾部插入，这里相当于做了一个小小的优化。
14          pxIterator = pxList->xListEnd.pxPrevious;
15      }
16      else
17      {
18          // 升序寻找插入位置
19          for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext
                ->xItemValue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
20          {
21              // 无操作
22          }
23      }
24
25      // step3 进行插入操作
26      pxNewListItem->pxNext = pxIterator->pxNext;
27      pxNewListItem->pxNext->pxPrevious = pxNewListItem;
28      pxNewListItem->pxPrevious = pxIterator;
29      pxIterator->pxNext = pxNewListItem;
30
31      // step4 记录链表项属于该链表
32      pxNewListItem->pxContainer = pxList;
33
34      // step5 记录链表中的链表项数目
35      ( pxList->uxNumberOfItems )++;
36  }
```

Listing 1.8: vListInsert() 分析

1.2.3 移除

FreeRTOS 的链表项移除由函数 uxListRemove() 实现，在链表移除的过程中需要注意的一点是当链表当前的 pxIndex 指向该待移除的链表项时，需要更改 pxIndex 以保证其指向一个合法的值，其具体实现过程如下

```
1   UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
2   {
3   List_t * const pxList = pxItemToRemove->pxContainer;
4
5       // step1 调整前后项指针
6       pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
7       pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
8
9       mtCOVERAGE_TEST_DELAY();
10
11      // step2 保证当前的链表索引指向有效项
12      if( pxList->pxIndex == pxItemToRemove )
13      {
14          pxList->pxIndex = pxItemToRemove->pxPrevious;
15      }
16      else
17      {
18          mtCOVERAGE_TEST_MARKER();
19      }
20
21      // step3 移除的链表项不再被链表拥有
22      pxItemToRemove->pxContainer = NULL;
23
24      // step4 减少链表项数目
25      ( pxList->uxNumberOfItems )--;
26
27      // step5 返回链表的链表项数目
28      return pxList->uxNumberOfItems;
29  }
```

1.2.4 遍历

FreeRTOS 中链表的遍历操作由宏 listGET_OWNER_OF_NEXT_ENTRY() 提供，宏的具体实现如下

```
#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )\
{\
List_t * const pxConstList = ( pxList );\
    // 寻找下一个链表项
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;\
    // 判断是不是不是到了xListEnd，如果是，跳过这一项，因为没有意义
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )->
        xListEnd ) )\
    {\
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;\
    }\
    // 返回拥有该链表项的对象的地址
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;\
}
```

可以看到，这里定义的变量名是 pxTCB(Task Control Block 任务控制块)，这个遍历操作主要是给任务调度管理用的。通过多次调用这一宏可以对链表进行遍历操作。

1.2.5 其它操作

在 list.h 中 FreeRTOS 还以宏的方式实现了其它的一些基本操作，例如 listLIST_IS_INITIALISED() 获取链表是否被初始化等操作。这些宏的实现都较为简单，不再详细列写。

宏定义函数与普通函数的区别

FreeRTOS 的 list.h（其它文件中也有）中定义了大量的宏定义函数。单单从形式看宏定义的函数和普通函数并无太大的区别，但事实上两者还是有很大不同。

- **宏定义函数与普通函数在编译过程上不同**。在编译时，对于宏定义函数而言，预编译时会将这些宏定义函数按展开的规则直接展开成语句，并且宏定义函数在代码中书写多少次，便展开多少次，拷贝相应的代码插入，生成相应的指令，而对于普通函数而言其只会生成一份相应的指令，调用处会生成传参指令和调用指令实现对函数的调用。
- **宏定义函数与普通函数在执行过程上不同**。在实际执行过程中，宏定义式函数所有的语句都是普通语句执行，而普通函数由于需要调用的缘故，需要进行开辟栈空间、压栈、出栈等操作。

基于以上两点不同使得宏函数和普通函数在使用和设计上有很大差异。

由于编译时，宏函数是按照宏定义展开规则进行展开的，这一点在提高它的灵活性的同时也会带来许多问题。例如它可以是一些函数的使用变得更为简单

```
#define DEF_MALLOC(n, type)   ( (type *) malloc((n)* sizeof(type)))
int *ptr;
ptr = (int *) malloc( (5) * sizeof(int) );
ptr = DEF_MALLOC( 5, int );
```

显然宏定函数在书写上要比普通函数简洁美观的多。但宏展开的一大弊端是它不会对参数类型进行检查，也不符合函数调用的一些规则，而且在定义时要格外小心其展开的结果是否是我们想要的结果，往往需要加上 {} 或 () 加以限制，例如以下的代码

```
#define MAX(x, y)   ((x)>(y)?(x):(y))
int x = 1;
int y = 3
MAX(++x,--y);
```

以上代码的本意可能是比较 ++x 和–y 谁更大，但实际情况却是

```
((++x)>(--y)?(++x):(--y))
```

违背了代码原本想要表达的含义。同时宏定义函数实际上被多次展开这一特点使得其在相同代码量和相同插入次数下，编译出的代码量要比普通函数多得多。例如

```
// 若函数fun()和DEF_FUN()代码完全一样则，以下普通函数代码编译结果为
    100byte
fun();
```

```
3  fun();
4  fun();
5  fun();
6  // 那么以下宏定义函数编译结果可能要接近400byte，是普通函数的4倍
7  DEF_FUN();
8  DEF_FUN();
9  DEF_FUN();
10 DEF_FUN();
```

在执行过程中由于出入栈等操作开销，相同的普通函数要比宏定义函数的执行效率低，使用以下代码在 PC 机上进行实验验证

```
1  #include "stdio.h"
2  #include "time.h"
3
4  #define DEF_MAX(x,y) ((x)>(y)?(x):(y))
5
6  int max(int x,int y)
7  {
8      return ((x)>(y)?(x):(y));
9  }
10
11 int main(void)
12 {
13     unsigned long long int i = 0;
14     unsigned long long int times = 100;
15     int x = 1;
16     int y = 2;
17     clock_t start, finish;
18
19     start = clock();
20     for(i=0;i<times;i++)
21     {
22         max(x, y);
23     }
24     finish = clock();
25     printf("common fun time %d\r\n",finish - start);
26
27     start = clock();
28     for(i=0;i<times;i++)
29     {
30         DEF_MAX(x, y);
31     }
32     finish = clock();
33     printf("macro fun time %d\r\n",finish - start);
34
35     return 0;
36 }
```

实验结果如下

次数	10	100	1000	10000	100000
普通函数耗时	10	14	56	482	4808
宏定义函数耗时	5	8	22	178	1672

从实验结果看，宏定义函数效率的确高于普通函数。

在实际应用中到底使用宏定义函数还是普通函数需要根据实际需求而定，个人认为当出现小段代码需要多次调用时可以使用宏函数来提高效率，宏函数应该保持短小简洁。

为结合普通函数定义更加安全，而宏函数较为高效的优点，有些编译器提供 inline 关键字，可以用来声明内连函数。

```
1  inline void fun()
```

```
2  {
3      \\ 一些代码
4  }
```

inline 函数和宏函数一样，它会在代码书写处直接拷贝一份指令，不会像普通函数一样单独生成指令然后调用，这使得其相对普通函数而言其仍然是高效的。但与宏函数不同，它是在编译阶段展开到生成指令中的，而不是预编译阶段展开到代码中，它会进行参数类型的检查，符合函数的一般直觉。另外，在实际运行过程中 inline 函数是可以使用调试器调试的，而宏函数不行。并不是所有函数都适合作为内联函数，内联函数应该满足以下要求，否则编译器可能会忽略 inline 关键字将其作为普通函数处理。

- 不使用循环（for，while）。
- 不使用 switch。
- 不能进行递归调用。
- 代码应短小，不能过长。

第 2 章　Task

任务的调度和管理是 RTOS 中的重要功能，本章将对 task.c 的部分源码进行分析，主要涉及以下几个关键的问题

- 任务调度器的启动与结束
- 任务调度器的挂起与恢复
- 任务的创建
- 任务删除
- 任务挂的挂起和恢复
- 任务间的切换过程
- 任务的延时阻塞的实现

task.c 中的大部分源码是为以上几个问题服务的，但也有不少源码是与事件相关的，这些源码不在本章的讨论范围之内，在后续理解分析源码的过程中，如果涉及相关问题会对其进行相应的分析。

2.1　变量及 TCB_t 结构体说明

本节内容旨在于对 task.c 中运用到的重要变量及结构体进行简要介绍，以方便后续源码的理解。

2.1.1　TCB_t

TCB_t 的全称为 Task Control Block，也就是任务控制块，这个结构体包含了一个任务所有的信息，它的定义以及相关变量的解释如下

```
1  typedef struct tskTaskControlBlock
2  {
3      // 这里栈顶指针必须位于TCB第一项是为了便于上下文切换操作，详见xPortPendSVHandler
           中任务切换的操作。
4      volatile StackType_t    *pxTopOfStack;
5
6      // MPU相关暂时不讨论
7      #if ( portUSING_MPU_WRAPPERS == 1 )
8          xMPU_SETTINGS   xMPUSettings;
9      #endif
10
11     // 表示任务状态，不同的状态会挂接在不同的状态链表下
12     ListItem_t          xStateListItem;
13     // 事件链表项，会挂接到不同事件链表下
14     ListItem_t          xEventListItem;
15     // 任务优先级，数值越大优先级越高
16     UBaseType_t         uxPriority;
17     // 指向堆栈起始位置，这只是单纯的一个分配空间的地址，可以用来检测堆栈是否溢出
18     StackType_t         *pxStack;
19     // 任务名
20     char                pcTaskName[ configMAX_TASK_NAME_LEN ];
21
22     // 指向栈尾，可以用来检测堆栈是否溢出
```

```
23      #if ( ( portSTACK_GROWTH > 0 ) || ( configRECORD_STACK_HIGH_ADDRESS == 1 ) )
24          StackType_t     *pxEndOfStack;
25      #endif
26
27      // 记录临界段的嵌套层数
28      #if ( portCRITICAL_NESTING_IN_TCB == 1 )
29          UBaseType_t     uxCriticalNesting;
30      #endif
31
32      // 跟踪调试用的变量
33      #if ( configUSE_TRACE_FACILITY == 1 )
34          UBaseType_t     uxTCBNumber;
35          UBaseType_t     uxTaskNumber;
36      #endif
37
38      // 任务优先级被临时提高时，保存任务原本的优先级
39      #if ( configUSE_MUTEXES == 1 )
40          UBaseType_t     uxBasePriority;
41          UBaseType_t     uxMutexesHeld;
42      #endif
43
44      // 任务的一个标签值，可以由用户自定义它的意义，例如可以传入一个函数指针可以用来做
            Hook    函数调用
45      #if ( configUSE_APPLICATION_TASK_TAG == 1 )
46          TaskHookFunction_t pxTaskTag;
47      #endif
48
49      // 任务的线程本地存储指针，可以理解为这个任务私有的存储空间
50      #if( configNUM_THREAD_LOCAL_STORAGE_POINTERS > 0 )
51          void            *pvThreadLocalStoragePointers[
                configNUM_THREAD_LOCAL_STORAGE_POINTERS ];
52      #endif
53
54      // 运行时间变量
55      #if( configGENERATE_RUN_TIME_STATS == 1 )
56          uint32_t        ulRunTimeCounter;
57      #endif
58
59      // 支持NEWLIB的一个变量
60      #if ( configUSE_NEWLIB_REENTRANT == 1 )
61          struct  _reent xNewLib_reent;
62      #endif
63
64      // 任务通知功能需要用到的变量
65      #if( configUSE_TASK_NOTIFICATIONS == 1 )
66          // 任务通知的值
67          volatile uint32_t ulNotifiedValue;
68          // 任务通知的状态
69          volatile uint8_t ucNotifyState;
70      #endif
71
72      // 用来标记这个任务的栈是不是静态分配的
73      #if( tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE != 0 )
74          uint8_t ucStaticallyAllocated;
75      #endif
76
77      // 延时是否被打断
78      #if( INCLUDE_xTaskAbortDelay == 1 )
79          uint8_t ucDelayAborted;
80      #endif
81
82      // 错误标识
83      #if( configUSE_POSIX_ERRNO == 1 )
84          int iTaskErrno;
85      #endif
86
```

```
87  } tskTCB;
88  typedef tskTCB TCB_t;
```

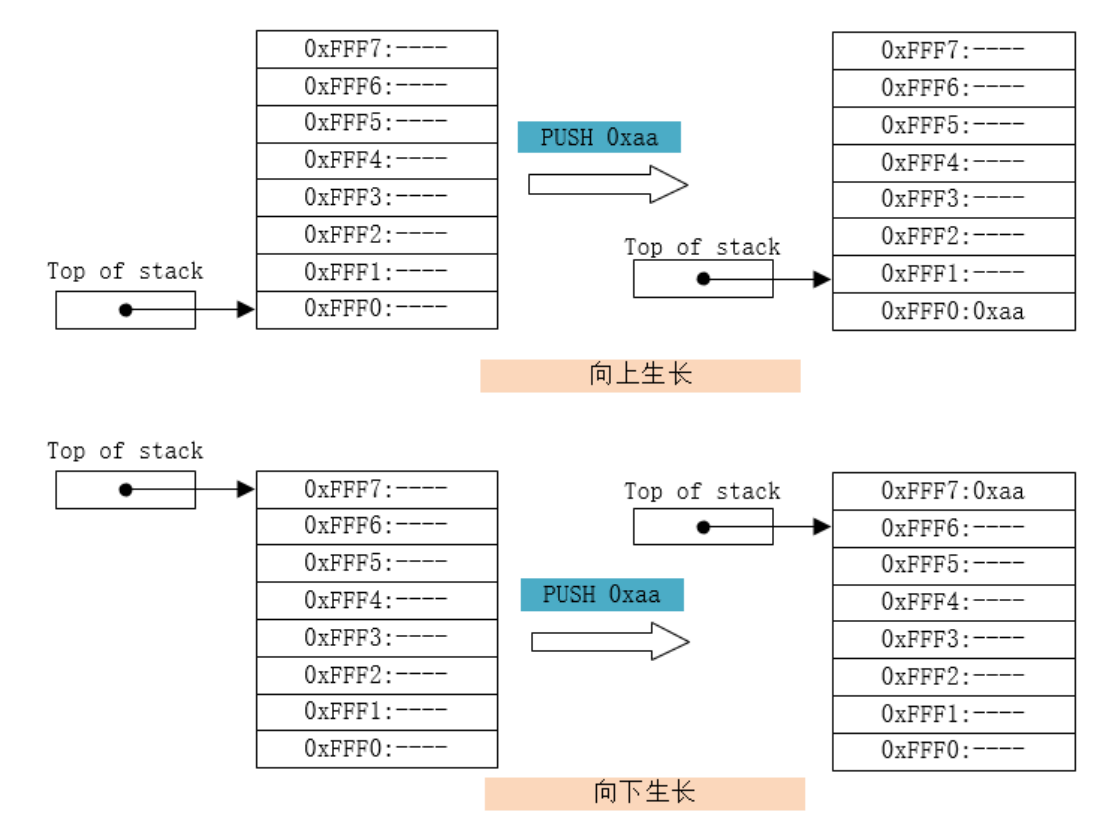
栈的生长方向

在 TCB_t 结构体的定义中可以看到根据栈的生长方式的不同，其将具有不同的成员变量 pxEndOfStack，在这里说明一下栈的生长方式是如何定义的，以及为何生长方式会存在 pxEndOfStack 这一变量的差异。

栈的生长方式可以分为两种，一种是向下生长，一种是向上生长，FreeRTOS 中用 portSTACK_GROWTH 来区分这两种生长方式，portSTACK_GROWTH 大于 0 为向上生长，小于零为向下生长。两种生长方式的区别可以简单概括如下

- 向上生长：入栈时栈顶指针增加，出栈时栈顶指针减小。
- 向下生长：入栈时栈顶指针减小，出栈时栈顶指针增加。

以下是两种生长方式的入栈图，很容易看出区别



为什么会有这两种出入栈方式呢？为何不将所有芯片统一成一种生长方式？这一点应该是芯片设计的实际需要，具体原因无法解答。

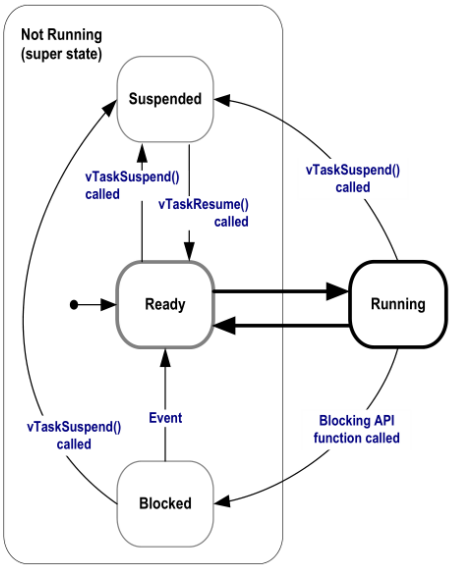
有了上图栈的生长方式为什么会影响成员变量的个数很好理解了，pxStack 是指向栈内存分配的起始地址（低地址），pxEndOfStack 是指向栈的尾部的，当栈是向下生长时，pxStack 和 pxEndOfStack 值是一致的，再定义 pxEndOfStack 浪费了内存，而栈是向上生长时 pxStack 与 pxEndOfStack 的值不一致，如果想知道栈的结束地址，必须要定义一个变量 pxEndOfStack 来存储，以用于后续的栈溢出检测等操作。

2.1.2 状态链表

FreeRTOS 中的任务一共有四种状态分别是运行状态（Running State），就绪状态（Ready State），阻塞状态（Blocked State），挂起状态（Suspended State），其含义可以简单理解为

- 运行状态: 正在执行的任务。
- 就绪状态：等待获得执行权的任务。
- 阻塞状态：直到某些条件达成才会重新进入就绪态等待获得执行权，否则不会执行的任务。
- 挂起状态：除非被主动恢复，否则永远不会执行。

《Mastering the FreeRTOS Real Time Kernel》对这四种任务状态的转换关系描述如下图



这四种链表分别对应着 pxCurrentTCB，pxReadyTasksLists，pxDelayedTaskList，xSuspendedTaskList 这四个变量。除运行状态外，任务处于其它状态时，都是通过将任务 TCB 中的 xStateListItem 挂到相应的链表下来表示的。

pxCurrentTCB

pxCurrentTCB 定义如下

```
1  PRIVILEGED_DATA TCB_t * volatile pxCurrentTCB = NULL;
```

当前运行的任务只可能有一个，因此 pxCurrentTCB 只是单个 TCB_t 指针，它始终指向当前运行的任务。

pxReadyTasksLists

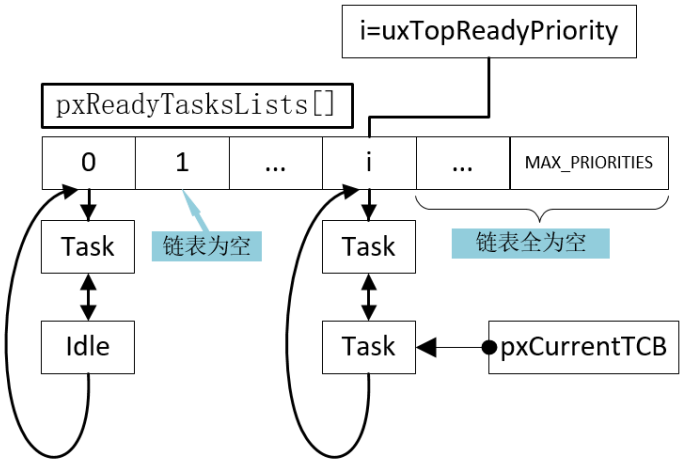
pxReadyTasksLists 定义如下

```
1  PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];
```

pxReadyTasksLists 不是单个链表，它是 configMAX_PRIORITIES 个链表组成的链表数组。链表数组中的每一个成员都是由处于就绪态而又有着相同任务优先级的任务组成的的链表。与之相关的还有一个变量 uxTopReadyPriority。uxTopReadyPriority 的定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxTopReadyPriority = tskIDLE_PRIORITY;
```

uxTopReadyPriority 存储的是有任务挂接的最高优先级。pxReadyTasksLists、pxCurrentTCB 和 uxTopReadyPriority 三者之间的关系可由以下的图来表示



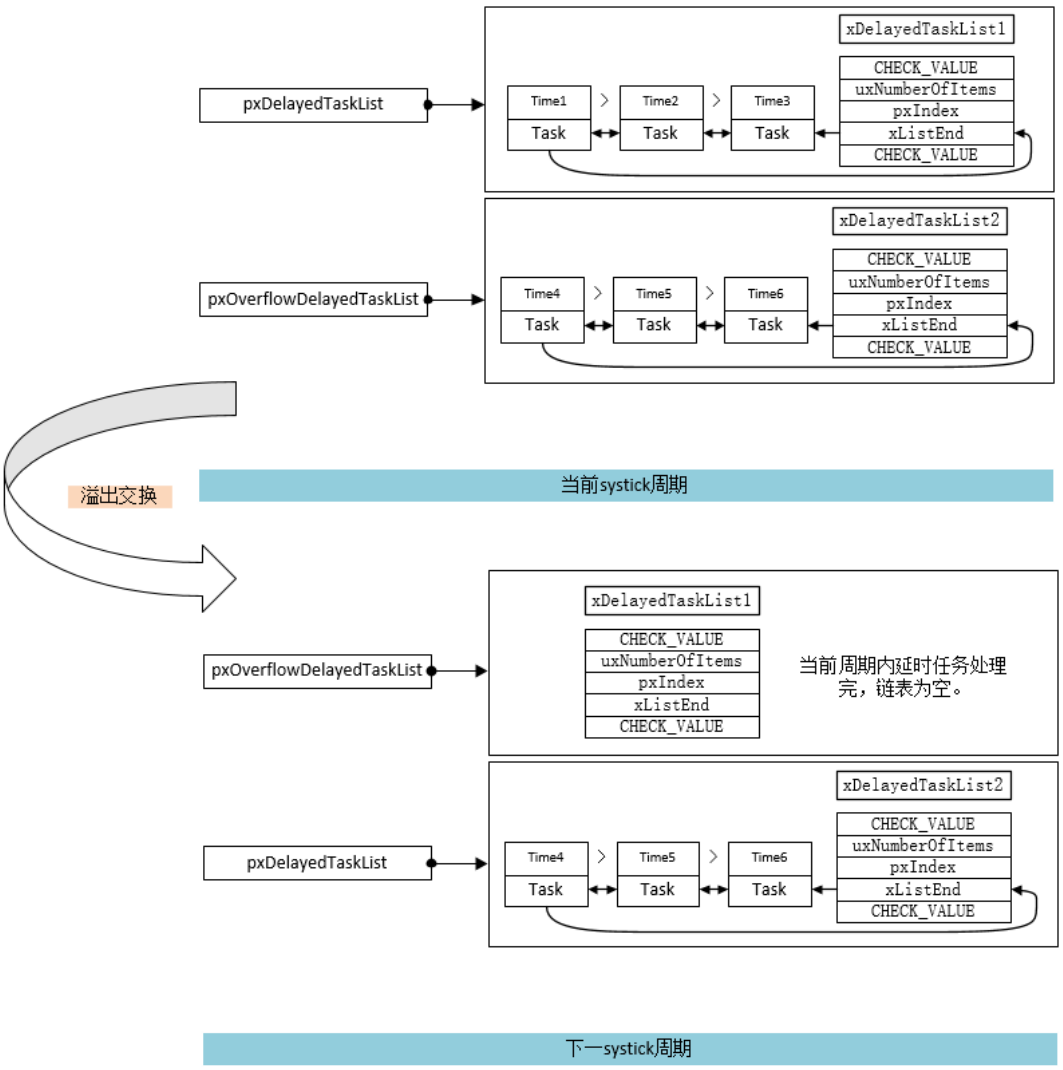
当使用时间片时，pxCurrentTCB 会在有任务挂接的最高优先级链表中遍历，以实现它们对处理器资源的分时共享，这些具体过程会在后面进行详细分析。

pxDelayedTaskList

延时链表的作用不仅是用来处理任务的延时，任务的阻塞也是由它进行实现的（至少目前分析的源码看是这样的）。由于 pxDelayedTaskList 要处理和时间相关的信息，因此需要考虑到系统的 systick 溢出的处理。为了解决这一繁琐的问题，FreeRTOS 设计了两个延时链表和两个延时链表指针来处理溢出问题，它们的定义如下

```
1  PRIVILEGED_DATA static List_t xDelayedTaskList1;
2  PRIVILEGED_DATA static List_t xDelayedTaskList2;
3  PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;
4  PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;
```

其中 pxDelayedTaskList 指向当前工作的延时任务链表，而 pxOverflowDelayedTaskList 指向溢出后的链表，xDelayedTaskList1 和 xDelayedTaskList2 是两个实际链表，其中任务的排列顺序是按退出阻塞时间排序的，也就是链表的第一个成员任务是将最早退出阻塞，而最后一个成员任务是最后退出阻塞的。当系统的 systick 溢出时，pxDelayedTaskList 和 pxOverflowDelayedTaskList 指向的链表地址也会随之交换一次，实现对溢出的处理。对于溢出的处理会在“任务的延时阻塞的实现”进行分析。以下是四个变量之间的关系



与延时任务链表变量为 xNextTaskUnblockTime。其定义如下

```
1  PRIVILEGED_DATA static volatile TickType_t xNextTaskUnblockTime = ( TickType_t ) 0U;
```

xNextTaskUnblockTime 存储的是下一个任务进行解除阻塞操作的时间，用来判断在何时进行解除阻塞操作。

xSuspendedTaskList

xSuspendedTaskList 的定义如下

```
1  PRIVILEGED_DATA static List_t xSuspendedTaskList;
```

其是一个普通的链表，下面挂接的是处于挂起状态的任务。

2.1.3 任务调度器操作相关变量

xSchedulerRunning

xSchedulerRunning 定义如下

```
1  PRIVILEGED_DATA static volatile BaseType_t xSchedulerRunning = pdFALSE;
```

该变量表示任务调度器是否已经运行（挂起的任务调度器也算在运行状态）。

uxSchedulerSuspended

uxSchedulerSuspended 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxSchedulerSuspended = ( UBaseType_t )
       pdFALSE;
```

uxSchedulerSuspended 的作用是记录任务调度器被挂起的次数，当这个变量为 0（dFALSE）时，任务调度器不被挂起，任务切换正常执行，当这个变量大于 0 时代表任务调度器被挂起的次数。如果执行挂起任务调度器操作该变量值会增加，如果执行恢复任务调度器操作，该变量值会减一，直到它为 0 时才会真正的执行实际的调度器恢复操作，这样可以有效的提高执行效率，这点在后面关于任务调度器的操作上会进一步探讨。

uxPendedTicks

uxPendedTicks 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxPendedTicks = ( UBaseType_t ) 0U;
```

任务调度器在被挂起期间，系统的时间，仍然是需要增加的。挂起期间漏掉的 systick 数目便会被存储在这个变量中，以用于恢复调度器时补上漏掉的 systick。

xPendingReadyList

xPendingReadyList 定义如下

```
1  PRIVILEGED_DATA static List_t xPendingReadyList;
```

这个链表中挂接的是在任务调度器挂起期间解除阻塞条件得到满足的阻塞任务，在任务调度器恢复工作后，这些任务会被移动到就绪链表组中，变为就绪状态。

2.1.4 任务删除相关

xTasksWaitingTermination

xTasksWaitingTermination 定义如下

```
1  PRIVILEGED_DATA static List_t xTasksWaitingTermination;
```

当任务自己删除自己时，其是不能立刻自己释放自己所占用的内存等资源的，其需要将自己挂接到 xTasksWaitingTermination 这个链表下，然后让 IdleTask 来回收其所占用的资源。

uxDeletedTasksWaitingCleanUp

uxDeletedTasksWaitingCleanUp 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxDeletedTasksWaitingCleanUp = (
       UBaseType_t ) 0U;
```

uxDeletedTasksWaitingCleanUp 记录了等待 IdleTask 处理的自己删除自己的任务的数目。

xIdleTaskHandle

xIdleTaskHandle 定义如下

```
1  PRIVILEGED_DATA static TaskHandle_t xIdleTaskHandle = NULL;
```

TaskHandle_t 本质上是指向任务 TCB 的指针，IdleTask 是任务调度器在启动时便自动创建的空闲任务，用于回收内存等操作，这个任务句柄指向 IdleTask。

2.1.5 系统信息相关

xTickCount

xTickCount 定义如下

```
1  PRIVILEGED_DATA static volatile TickType_t xTickCount = ( TickType_t )
       configINITIAL_TICK_COUNT;
```

存储 systick 的值，用来给系统提供时间信息。

xNumOfOverflows

xNumOfOverflows 定义如下

```
1  PRIVILEGED_DATA static volatile BaseType_t xNumOfOverflows  = ( BaseType_t ) 0;
```

这个值保存了 xTickCount 溢出的次数。

uxTaskNumber

uxTaskNumber 定义如下

```
1  PRIVILEGED_DATA static UBaseType_t uxTaskNumber = ( UBaseType_t ) 0U;
```

每创建一个任务，这个值便会增加一次，为每个任务生成一个唯一的序号，供调试工具使用。注意与 uxCurrentNumberOfTasks 区分。

uxCurrentNumberOfTasks

uxCurrentNumberOfTasks 定义如下

```
1  PRIVILEGED_DATA static volatile UBaseType_t uxCurrentNumberOfTasks  = ( UBaseType_t )
       0U;
```

存储当前任务的数目。

2.2 任务的创建与删除

2.2.1 任务创建

FreeRTOS 提供了以下 4 种任务创建函数

- xTaskCreateStatic()：以静态内存分配的方式创建任务，也就是在编译时便要分配好 TCB 等所需要内存。
- xTaskCreateRestrictedStatic()：以静态内存分配的方式创建任务，需要 MPU。
- xTaskCreate()：以动态内存分配方式创建任务，需要提供 portMolloc() 函数的实现，在程序实际运行时分配 TCB 等所需要内存。
- xTaskCreateRestricted()：以动态内存分配方式创建任务，需要 MPU。

任务创建函数大致可以按内存分配的方式分为静态和动态两种，简单来说就是是否用到了 portMolloc() 函数（关于 portMolloc() 会在 FreeRTOS 内存管理的章节中单独分析），其内容大致相同（MPU 没有研究）。以 xTaskCreate() 函数为例分析任务创建的过程。xTaskCreate() 函数的原型为

```
1  BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,
2                          const char * const pcName,
3                          const configSTACK_DEPTH_TYPE usStackDepth,
4                          void * const pvParameters,
5                          UBaseType_t uxPriority,
6                          TaskHandle_t * const pxCreatedTask )
```

各参数的含义如下

- pxTaskCode：指向任务函数的函数指针。
- pcName：任务的名称。
- usStackDepth：栈的深度，这里的栈的单位不是 byte 而是根据平台的位数决定的，8 位，16 位，32 位分别对应 1，2，3，4byte。
- pvParameters：传入任务的参数。
- uxPriority：任务的优先级。数值越大，任务的优先级越高。
- pxCreatedTask：创建的任务的句柄，本质就是一个指向创建任务 TCB 的指针。
- 返回值：pdPass 代表创建任务成功，errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY（pdFalse）代表分配内存时出现错误。

个人认为任务初始化的工作主要分为三步：

- 第一步分配存储空间；
- 第二步初始化栈、填充 TCB 结构体；
- 第三步是将 TCB 挂接到就绪链表中并根据优先级进行任务切换；

下面就对每一步的相关代码进行分析

分配存储空间

根据栈的生长方向，FreeRTOS 采用了两种内存分配顺序。暂且将栈向上生长的分配方式放一边，看一下在栈向下生长时，内存分配过程是怎样的

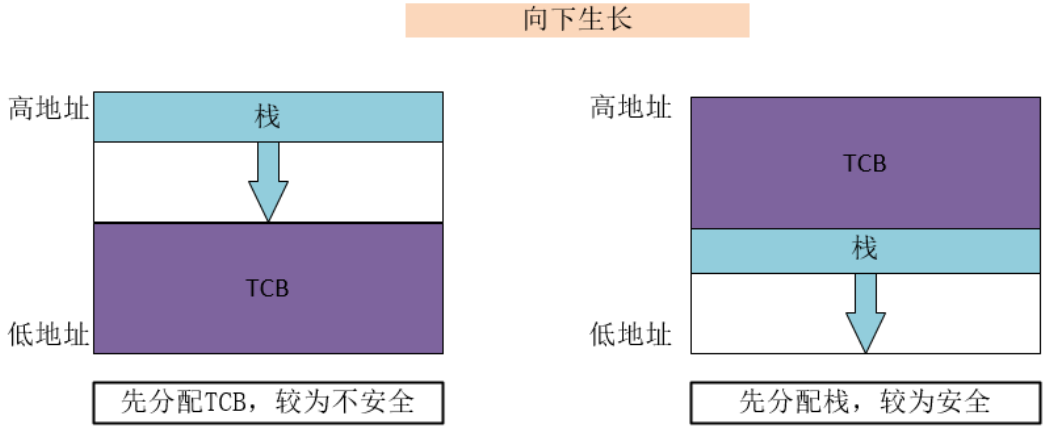
```
1  TCB_t *pxNewTCB;
2  BaseType_t xReturn;
3
4  StackType_t *pxStack;
5
6  // 分配栈空间
7  pxStack = pvPortMalloc( ( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) ) );
8
9  if( pxStack != NULL )
10 {
11     // 分配栈空间成功，分配TCB结构体空间
12     pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
13
14     if( pxNewTCB != NULL )
15     {
16         // 存储栈地址到TCB
17         pxNewTCB->pxStack = pxStack;
18     }
19     else
20     {
21         // 分配TCB结构体空间失败，释放栈空间
22         vPortFree( pxStack );
23     }
24 }
25 else
26 {
27     // 分配栈空间失败
28     pxNewTCB = NULL;
29 }
```

这段代码并不难理解十分简单,唯一值得注意的是栈空间时分配的字节数是 usStackDepth*sizeof(StackType_t)，这意味着我们在设置任务栈空间大小是不是按字节数来分配的，而是和平台相关，例如 32 位的 stm32 分配 1 栈大小等于 4 字节，其余就不多做解释。

再来对比一下栈向上生长的分配方式

```
1  TCB_t *pxNewTCB;
2  BaseType_t xReturn;
3
4  pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );
5
6  if( pxNewTCB != NULL )
7  {
8      pxNewTCB->pxStack = ( StackType_t * ) pvPortMalloc( ( ( ( size_t ) usStackDepth )
           * sizeof( StackType_t ) ) );
9
10     if( pxNewTCB->pxStack == NULL )
11     {
12         vPortFree( pxNewTCB );
13         pxNewTCB = NULL;
14     }
15 }
```

两段代码唯一的区别是，分配栈空间和 TCB 结构体空间的顺序不同。由于这两部分的内存分配操作是连续的，这导致其在地址空间上也是连续的，使用合理的分配顺序可以避免栈上溢时本任务的 TCB 数据被破坏。



初始化栈、填充 TCB 结构体

该步主要是由 prvInitialiseNewTask() 函数完成的，其函数原型和参数定义如下

```
1  static void prvInitialiseNewTask(    TaskFunction_t pxTaskCode,
2                                       const char * const pcName,
3                                       const uint32_t ulStackDepth,
4                                       void * const pvParameters,
5                                       UBaseType_t uxPriority,
6                                       TaskHandle_t * const pxCreatedTask,
7                                       TCB_t *pxNewTCB,
8                                       const MemoryRegion_t * const xRegions )
```

- pxNewTCB：TCB 地址。
- xRegions：MPU 相关暂时不讨论。
- 其余参数定义同 xTaskCreate()。

prvInitialiseNewTask() 函数的执行过程大致可以分为

- MPU 相关设置；
- 将栈值设定为特定值，以用于栈最高使用大小检测等功能；
- 计算栈顶指针、栈底指针；
- 复制任务名、写入优先级等相关 TCB 结构体成员赋初值；
- 初始化链表项；
- 对栈进行初始化；

这部分代码整体简单，基本都是一些赋值操作。这里仅对栈顶指针、栈底指针计算和栈的初始化的一些操作进行说明。

在计算栈顶、栈底指针时，都要进行如下的位与操作

```
1  pxTopOfStack = ( StackType_t * ) ( ( ( portPOINTER_SIZE_TYPE ) pxTopOfStack ) & ( ~(
       ( portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK ) ) );
```

这个操作是为了使得栈指针地址是对齐的，portBYTE_ALIGNMENT_MASK 一般被定义为 0x00000007，也就是 8 字节对齐。stm32f103c8t6 在设置时也需要 8 字节对齐，为什么需要字节对齐？又为什么是 8 字节而不是其它呢？

首先为什么需要字节对齐？对于普通的变量而言，字节对齐是为了提高变量的读取效率，具体的原因和处理器的硬件设计有关，其寻址的地址通常不是任意的而是有规律的，例如 32 位处理器，其寻址范围可能只是 4 的倍数，这使得其在处理非字节对齐变量时需要花费更多的时间。对于堆栈地址而言，其存在硬件上的限制也存在软件上的限制，例如 stm32f103c8t6 的 cortex-m3 内核的手册《Cortex-M3 权威指南（中文版初稿）》中的第 27 页明确写道“堆栈指针的最低两位永远是 0，这意味着堆栈总是 4 字节对齐的。”，而在第 39 页也说明“寄存器的 PUSH 和 POP 操作永远都是 4 字节对齐的——也就是说他们的地址必须是 0x4,0x8,0xc,……。”因此，stm32f103c8t6 保证堆栈地址 4 字节对齐是必须的。同时 ARM 编程涉及调用时都需要遵循 AAPCS:《Procedure Call Standard for the ARM Architecture》，规约中表示

> - 5.2.1.1 Universal stack constraints At all times the following basic constraints must hold:Stack-limit < SP <= stack-base. The stack pointer must lie within the extent of the stack.SP mod 4 = 0. The stack must at all times be aligned to a word boundary.
> - 5.2.1.2 Stack constraints at a public interface The stack must also conform to the following constraint at a public interface: SP mod 8 = 0. The stack must be double-word aligned.”

这套规约在限制堆栈地址必须是 4 字节对齐的同时，也要求了在调用入口得 8 字节对齐。4 字节对齐是必须的，8 字节对齐可以不遵守，但这已成为 ARM 堆栈处理的一个标准，如果不遵循程序运行可能会出现问题，也相当于是半强制的了。8 字节对齐的原因暂且分析到这里，更为细节的问题笔者也无能为力。

栈的初始化过程是和硬件平台相关的，其主要目的是将栈“伪装”成这个任务已经执行过一次上下文切换的状态，以保证在任务切换时，任务可以正常运行。由于每个平台该函数的实现都是不一样的，这部分内容将会在任务切换中结合 cortex-m3 来详细分析代码的具体含义。

使任务处于就绪态和任务切换

这部分工作是由 prvAddNewTaskToReadyList() 这一函数实现的，函数的原型如下

```
1  static void prvAddNewTaskToReadyList( TCB_t *pxNewTCB )
```

当任务调度器已经处于工作状态时，也就是 xSchedulerRunning = True 时，prvAddNewTaskToReadyList() 需要做的主要工作只有三件事

1. 记录当前任务数量。
2. 将任务添加到就绪链表中。
3. 根据新加入的优先级判断是否需要进行一次任务切换。

此时执行的相关代码如下

```
1  taskENTER_CRITICAL();
2
3  // step1 使用全局变量uxCurrentNumberOfTasks记录任务数
4  uxCurrentNumberOfTasks++;
5
6  // step2 把新创建任务添加到就绪链表中
7  prvAddTaskToReadyList( pxNewTCB );
8  taskEXIT_CRITICAL();
9
10 // step3 切换任务
11 if( xSchedulerRunning != pdFALSE )
12 {
13     // 如果任务调度器已经工作了，且当前任务的优先级低于新建任务优先级，启动一次任务切
       换切换到新创建的任务
14     if( pxCurrentTCB->uxPriority < pxNewTCB->uxPriority )
15     {
16         taskYIELD_IF_USING_PREEMPTION();
17     }
18     else
19     {
20         mtCOVERAGE_TEST_MARKER();
21     }
```

```
22  }
23  else
24  {
25      mtCOVERAGE_TEST_MARKER();
26  }
```

任务调度器已经处于工作状态时，也就是未调用 xSchedulerRunning = False 函数，prvAddNewTaskToReadyList() 需要负责将相关链表及 pxCurrentTCB 设置成任务调度器已经运行的状态，保证任务切换启动时可以正常工作。此时其工作变为以下三项

1. 记录当前任务数量。
2. 初始化就绪链表以及相关链表，让 pxCurrentTCB 指向优先级最高的任务。
3. 将任务添加到就绪链表中。

此时执行的相关代码如下

```
1   taskENTER_CRITICAL();
2
3   // step1 使用全局变量uxCurrentNumberOfTasks记录任务数
4   uxCurrentNumberOfTasks++;
5
6   // step2 初始化就绪链表以及相关链表，让pxCurrentTCB指向优先级最高的任务。
7   if( pxCurrentTCB == NULL )
8   {
9       // 没有任务，将新添加任务作为当前任务
10      pxCurrentTCB = pxNewTCB;
11      if( uxCurrentNumberOfTasks == ( UBaseType_t ) 1 )
12      {
13          // 如果是首次添加任务，初始化任务链表
14          prvInitialiseTaskLists();
15      }
16      else
17      {
18          mtCOVERAGE_TEST_MARKER();
19      }
20  }
21  else
22  {
23      // 如果任务调度器还没工作，将新添加的任务与当前待执行任务进行优先级比较，优先级高
            则替换当前任务待执任务，任务调度器启动时将执行优先级最高的任务。
24      if( xSchedulerRunning == pdFALSE )
25      {
26          if( pxCurrentTCB->uxPriority <= pxNewTCB->uxPriority )
27          {
28              pxCurrentTCB = pxNewTCB;
29          }
30          else
31          {
32              mtCOVERAGE_TEST_MARKER();
33          }
34      }
35      else
36      {
37          mtCOVERAGE_TEST_MARKER();
38      }
39  }
40
41  // step3 把新创建任务添加到就绪链表中
42  prvAddTaskToReadyList( pxNewTCB );
43  taskEXIT_CRITICAL();
```

在将任务插入就绪链表中时采用的宏 prvAddTaskToReadyList() 相关代码如下

```
1   // 记录就绪任务的最高先级
2   #define taskRECORD_READY_PRIORITY( uxPriority )\
3   {\
4       if( ( uxPriority ) > uxTopReadyPriority )\
```

```
5       {\
6           uxTopReadyPriority = ( uxPriority );\
7       }\
8   }
9
10  #define prvAddTaskToReadyList( pxTCB )\
11      taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );\
12      // 按优先级放到对应的链表下
13      vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->
            xStateListItem ) );\
```

可以看到在插入就绪链表中是，其插入的方式是无序的插入方式 vListInsertEnd()，这也就意味着同等优先级的任务有着同等的地位。

在以上的代码中，首次出现了 taskENTER_CRITICAL(); taskEXIT_CRITICAL(); 这样的临界段操作，对于这些代码，其将会与 vTaskSuspendAll();xTaskResumeAll(); 放在一起进行比较说明。

以上便是 FreeRTOS 中一个任务创建的全部过程。

2.2.2 任务的删除

FreeRTOS 中的任务删除函数为 vTaskDelete()，其原型和参数含义如下

```
1   void vTaskDelete( TaskHandle_t xTaskToDelete )
```

- TaskHandle_t：待删除任务的句柄。

任务删除的调用场景主要可以分为两种，一种是调用 vTaskDelete() 的任务与待删除的任务 xTaskToDelete 不是同一个任务（当前任务删除其它任务），另外一种是调用 vTaskDelete() 的任务与待删除的任务 xTaskToDelete 是同一个任务（自己删除自己，xTaskToDelete = NULL）

当在当前任务中调 vTaskDelete() 删除其他任务时，所需要进行的工作步骤如下：

1. 将待删除任务从相关的状态链表中删除，设置相关参数，保证被删除的任务不会再次获得处理器使用权。
2. 将待删除任务从其相关的事件链表中删除，设置相关参数，保证被删除的任务不会再次获得处理器使用权。
3. 更改当前任务数目。
4. 直接释放内存空间。
5. 重新计算下个任务解除阻塞的时间。

计算任务解除阻塞的时间的函数为 prvResetNextTaskUnblockTime()，因为 FreeRTOS 系统中所有的阻塞都是由将任务按解除阻塞时间升序挂接到延时任务链表 pxDelayedTaskList 中实现的，因此 prvResetNextTaskUnblockTime() 实际上只是读取 pxDelayedTaskList 下的第一个任务解除阻塞的时间，将其赋值给 xNextTaskUnblockTime 而已，如果 pxDelayedTaskList 为空，那么 xNextTaskUnblockTime 将会被赋值为 portMAX_DELAY。

当任务是自己删除自己时，上述步骤的第 4 步将有所变化。当前任务仍在运行中，此时直接释放其占用的内存可能导致运行错误，因此需要等待其退出运行状态时才能安全的对其占用的内存进行释放。此时上述的步骤 4 替换为以下两步

- 将待删除任务挂接到待终止任务链表 xTasksWaitingTermination 中。
- 增加删除待清理任务数 uxDeletedTasksWaitingCleanUp。

在完成任务解除阻塞时间更新后，其还会调用一次任务切换，将处理器的使用权交接给其它的任务。FreeRTOS 的空闲任务中会调用 prvCheckTasksWaitingTermination() 函数来检测 uxDeletedTasksWaitingCleanUp 变量，并依次释放掉挂接在 xTasksWaitingTermination 链表下的任务占用的内存，完成任务删除。

其具体的代码可以参照源码阅读，这里不进行过多的解释。

2.3 任务切换

任务切换的目的是保证当前具有最高优先级的就绪任务获得处理器的使用权。在进行任务切换时，首先需要找到具有最高优先级的就绪任务，如果该任务不是当前正在运行的任务，需要先保存当前运行任务的堆栈，并将具有最高优先级的就绪任务堆栈恢复到处理器的堆栈中进行运行。

2.3.1 寻找拥有最高优先级的就绪任务

如何寻找具有最高优先级的就绪任务？根据本章前文对结构体和变量的以及任务创建过程的分析可知，所有的就绪任务都是挂载在一个链表数组 pxReadyTasksLists 中的，链表数组的下标代表了任务的优先级，同一个链表下挂接的任务具有相同的优先级，在任务切换过程中，它们将轮流的获得处理器的使用权。如此一来只需要在向 pxReadyTasksLists 添加或删除任务时，记录变动任务后处于就绪态任务的最高优先级即可获得待运行的任务。最高优先级被记录在 uxTopReadyPriority 这一变量中，宏函数 taskRECORD_READY_PRIORITY(uxPriority) 实现了在添加任务时记录最高优先级这一过程，

```
1  // 记录最高先级的就绪任务的优先级
2  #define taskRECORD_READY_PRIORITY( uxPriority )\
3  {                                               \
4      if( ( uxPriority ) > uxTopReadyPriority )   \
5      {                                           \
6          uxTopReadyPriority = ( uxPriority );    \
7      }                                           \
8  } /* taskRECORD_READY_PRIORITY */
```

根据最高优先级获得要运行任务的过程由宏函数 taskSELECT_HIGHEST_PRIORITY_TASK() 实现，寻找到的拥有最高优先级的待运行任务会被存储到 pxCurrentTCB 变量中，其具体实现如下

```
1  #define taskSELECT_HIGHEST_PRIORITY_TASK()                                          \
2  {                                                                                   \
3      UBaseType_t uxTopPriority = uxTopReadyPriority;                                 \
4                                                                                      \
5      /* step1 */                                                                     \
6      while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ) ) )           \
7      {                                                                               \
8          configASSERT( uxTopPriority );                                              \
9          --uxTopPriority;                                                            \
10     }                                                                               \
11                                                                                     \
12     /* step2 */                                                                     \
13     listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ]
           ) );\
14     uxTopReadyPriority = uxTopPriority;
               \
15 } /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

插入操作导致的就绪任务优先级变动由 taskRECORD_READY_PRIORITY(uxPriority) 进行了处理，删除任务导致的任务优先级变化则在 taskSELECT_HIGHEST_PRIORITY_TASK() 中的 step1 被处理了。step2 中 listGET_OWNER_OF_NEXT_ENTRY() 函数在分析链表时已经进行解析，其会循环遍历链表中的每一个任务，因此最高优先级下的所有任务会平均的共享处理器的使用权。

以上所说的是通用版本的做法，除此方法外，FreeRTOS 还提供了一套经过优化的，用来记录和寻找最高优先级的方法。对于 crotex-m3 为内核的平台，其借助特殊指令 clz（从最高位开始，计算一个整型变量 0 的个数），实现 bit map 方法，以此提高程序运行的效率，在优化版本下，上述函数的代码如下

```
1  // 经过特殊优化的方法，依赖硬件
2  // 这里不在使用数值大小来表示最高优先级，而是使用每一位表示是否有该优先级的任务处于就
       绪态，对于cortex-m3有32位，如
3  // 0000 0000 0000 0000 0000 0000 0000 0001 表示第0级有就绪态的任务
4  #define taskRECORD_READY_PRIORITY( uxPriority ) portRECORD_READY_PRIORITY( uxPriority
       , uxTopReadyPriority )
5
```

```
6  /*-----------------------------------------------------------*/
7
8  #define taskSELECT_HIGHEST_PRIORITY_TASK()
                  \
9  {
                  \
10 UBaseType_t uxTopPriority;
                  \
11


12     // 对于Cortex-m3 其会调用CLZ汇编指令（计算变量从高位开始连续0的个数），快速获取当
           前任务中的最高优先级
13     portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );
                    \
14     configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ uxTopPriority ] ) )
           > 0 );     \
15     listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopPriority ]
           ) );       \
16 } /* taskSELECT_HIGHEST_PRIORITY_TASK() */
17
18 /*-----------------------------------------------------------*/
19
20 #define taskRESET_READY_PRIORITY( uxPriority )
                    \
21 {
                    \
22     if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ ( uxPriority ) ] ) ) == (
           UBaseType_t ) 0 )  \
23     {
                        \
24         portRESET_READY_PRIORITY( ( uxPriority ), ( uxTopReadyPriority ) );
                            \
25     }
                        \
26 }
```

其使用一个 4 字节整型变量来表示 32 个优先级下是否有就绪的任务（在优化版本中，优先级是一定的，最高为 31），若有就绪的任务，该位被置为 1，没有则置为 0。portRECORD_READY_PRIORITY()，portRESET_READY_PRIORITY()，portGET_HIGHEST_PRIORITY() 这几个宏定义函数内容如下

```
1  #define portRECORD_READY_PRIORITY( uxPriority, uxReadyPriorities ) (
       uxReadyPriorities ) |= ( 1UL << ( uxPriority ) )
2
3  #define portRESET_READY_PRIORITY( uxPriority, uxReadyPriorities ) ( uxReadyPriorities
       ) &= ~( 1UL << ( uxPriority ) )
4
5  #define portGET_HIGHEST_PRIORITY( uxTopPriority, uxReadyPriorities ) uxTopPriority =
       ( 31UL - ( uint32_t ) __clz( ( uxReadyPriorities ) ) )
```

本质上就是对对应位的置位，复位处理，使用 CLZ 指令可以获得从高位开始连续 0 的个数，用 31 减去这个数，便可以获得就绪任务的最高优先级，其余处理和通用方法一样。

2.3.2 进入任务切换的方式

FreeRTOS 进入任务切换的方式有以下两种

1. 在 xPortSysTickHandler() 中断中进入，也就是在系统 systick 增加时，根据情况进入任务切换。
2. 手动调用 portYIELD_WITHIN_API() 或 taskYIELD_IF_USING_PREEMPTION()（在启用抢占模式的情况下其和 portYIELD_WITHIN_API 一样，非抢占模式下，其没有任何作用）直接进行一次任务切换。

由于任务切换过程是和硬件平台相关的，这里以 cortex-m3 内核为例进行分析，其它内核可以类比 cortex-m3 内核的切换方式。cortex-m3 平台下 xPortSysTickHandler() 函数如下，

```
1  void xPortSysTickHandler( void )
2  {
3      vPortRaiseBASEPRI();
```

```
4       {
5           // 这里并不是每次进入系统滴答中断都会进行上下文切换，只有有任务从阻塞状态退出
                或者在时间片轮询模式中有相同的优先级的任务，才会进行上下文切换。
6           if( TxTaskIncrementTick() != pdFALSE )
7           {
8               // 触发一次PendSV异常，进入PendSV中断。
9               portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
10          }
11      }
12      vPortClearBASEPRIFromISR();
13  }
```

xTaskIncrementTick() 函数的主要功能是在在任务调度器工作时修改 systick 的值，并根据 systick 值的变化判断是否需要进行一次任务切换动作；在任务调度器被挂起时，其会记录任务调度器挂起期间漏掉的 systick 数，一旦任务调度器恢复运行，任务调度器会补上漏掉的 systick 和相应的任务切换动作在任务调度器工作时，当以下两种情况发生时，xTaskIncrementTick() 将返回 pdTRUE，以触发一次 PendSV 中断，以进行任务切换动作

1. 当前时刻有任务需要退出阻塞状态
2. 启用时间片模式，当前优先级下有多个任务，需要共享使用权。

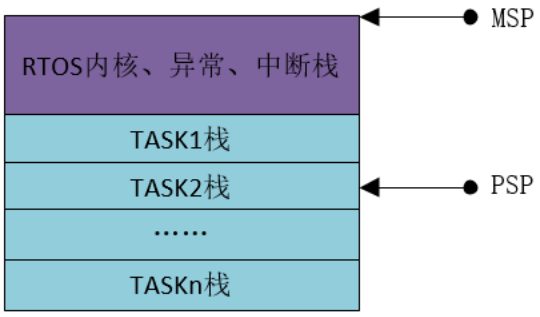
portYIELD_WITHIN_API() 等接口函数，是直接触发 PendSV 中断的，每调用一次便立刻进行一次任务切换。这个函数一般用在删除当前任务，补全 systick 等特殊动作时调用。

2.3.3 PendSV 中断任务切换

cortex-m3 内核有两个堆栈指针，分别是 MSP（Main Stack Pointer 主栈指针）和 PSP（Process Stack Pointer 进程栈指针），这两个指针共用一个引用 R13（SP），引用的是当前正在使用的栈指针。在任何时刻只能使用其中的一个作为栈指针，复位后的默认情况下使用的是 MSP 栈指针（裸机程序仅使用一个栈指针即可），在以下一些情况下使用的栈指针可以发生改变

1. 修改 CONTROL 寄存器的高一位 CONTROL[1]，0 表示选择 MSP，1 表示使用 PSP
2. 进入中断后，CONTROL[1] 自动置为 1，只能选择使用 MSP
3. 异常返回时，LR 寄存器用法发生改变，其会自动存储特殊值 EXC_RETURN，EXC_RETURN 表示了运行模式的一些信息，其中其第二位表示返回后使用的指针，0 表示 MSP，1 表示 PSP。

一般而言，MSP 会被用于 RTOS 的内核以及异常处理，而用户的任务会使用 PSP，RTOS 中 MSP 和 PSP 的使用方式如下图所示。



所谓任务切换也就相当于修改 PSP 指针的值，使其指向具有最高优先级任务正确的栈指针位置并且恢复处理器中的相应寄存器的值。

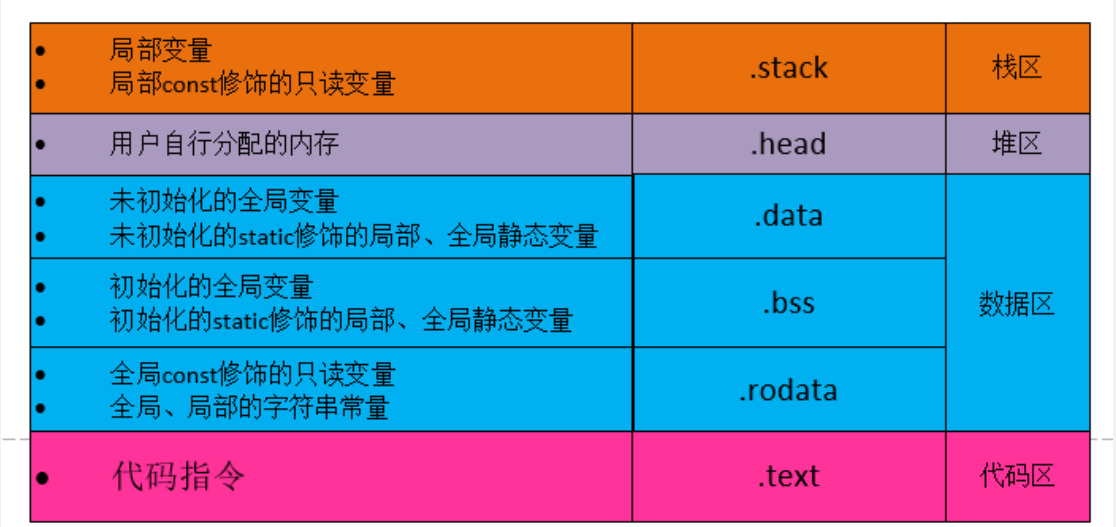
堆和栈的区别（非数据结构上的堆栈）

堆栈这两个词常被一起使用，但本质上这是两个不同的东西。简单而言，堆是由程序编写者自由分配的内存区域由编写者自己管理，如使用 malloc 和 free 来进行分配和释放，而栈是由系统自动分配和管理的，不需要编写者干预，程序中函数的局部变量，参数，调用上下文等，都是存放

在栈上的，当函数运行结束，这些变量栈上占用的空间会被自动释放掉。

栈和堆的大小在编写、编译时都是无法确定的，只有在程序实际运行时才会真正的分配。例如在 STM32 中，如果启动文件下设置了栈的大小为 1k，而编写的函数中有一大小为 2k 的数组，远大于分配的栈大小，编译时编译器并不对这一情况报错，在未执行该函数时程序也是正常运行的，但一旦执行该函数，程序必然会出现一些莫名其妙的错误。因此编写程序时，要注意使用的栈大小，合理分配栈大小，避免栈溢出。堆的大小同样要注意，避免内存不够用的情况。堆栈的分配过小会导致程序运行错误，而分配的值过大则会导致内存资源的浪费。

C 语言程序中除了堆区、栈区外还有数据区和代码区，C 语言中数据的存储结构如下图所示



任务切换的动作都是在 PendSV 中断中完成的，由以上的分析，可以判断，PendSV 中断中需要完成三个动作，将当前任务的任务堆栈保存，将待切换任务的任务堆栈恢复，将 PSP 指向正确的位置。至此可以清晰的理解以下的汇编代码

```
1  __asm void xPortPendSVHandler( void )
2  {
3      extern uxCriticalNesting;
4      extern pxCurrentTCB;
5      extern vTaskSwitchContext;
6
7      /*DMB
8      数据存储器隔离。DMB 指令保证仅当所有在它前面的存储器访问操作
9      都执行完毕后，才提交(commit)在它后面的存储器访问操作。
10
11     DSB
12     数据同步隔离。比 DMB 严格：  仅当所有在它前面的存储器访问操作
13     都执行完毕后，才执行在它后面的指令（亦即任何指令都要等待存储器访 问操作）
14
15     ISB
16     指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执
17     行完毕之后，才执行它后面的指令。*/
18
19     /* step1 保存当前任务现场 */
20     /* ==================================================================*/
21
22     // 字节对齐
23     PRESERVE8
24
25     // PendSV中断产生时，硬件自动将xPSR，PC(R15)，LR(R14)，R12，R3-R0使用PSP压入任务
           堆栈中，进入中断后硬件会强制使用MSP指针，此时LR（R14）的值将会被自动被更新为
           特殊的EXC_RETURN.
26
```

```
27      mrs r0, psp                     // 保存进程堆栈指针到R0
28      isb
29
30      ldr r3, =pxCurrentTCB           // 读取当前TCB块的地址到R3
31      ldr r2, [r3]                    // 将当前任务栈顶地址放到R2中，这是为什么强调栈顶指针
            一定得是TCB块的第一个成员的原因
32
33      stmdb r0!, {r4-r11}             // 将R4到R11通用寄存器的值压入栈保存
34      str r0, [r2]                    // 将R0的值写入以R2为地址的内存中，也就是保存当前的栈
            顶地址到TCB的第一个成员，也就是栈顶指针
35
36      stmdb sp!, {r3, r14}            // 将R3，R14临时压栈，这里的SP其实使用的是MSP，这里进
            行压栈保护的原因是bl指令会自动更改R14值用于返回
37
38      // 屏蔽configMAX_SYSCALL_INTERRUPT_PRIORITY以下优先级的中断
39      mov r0, #configMAX_SYSCALL_INTERRUPT_PRIORITY
40      msr basepri, r0
41
42      /* step2 恢复待切换任务的现场 */
43      /* ================================================================*/
44
45      dsb
46      isb
47      bl vTaskSwitchContext           // 这里调用vTaskSwitchContext函数来获取下一个要执行任
            务控制块
48
49      // 取消中断屏蔽
50      mov r0, #0
51      msr basepri, r0
52
53      ldmia sp!, {r3, r14}            // 将R3，R14出栈，这里R3相当于是pxCurrentTCB内存的
            值，所以此时R3值已经更新为下一个要执行的任务TCB地址了
54
55      ldr r1, [r3]
56      ldr r0, [r1]                    // 把新任务的栈顶指针放到R0里
57      ldmia r0!, {r4-r11}             // 将新任务的R4-R11出栈
58
59      /* step3 更改PSP指针值 */
60      /* ================================================================*/
61      msr psp, r0                     // 将新的栈顶地址放入到进程堆栈指针PSP
62      isb
63      bx r14  // 异常发生时,R14中保存异常返回标志,包括返回后进入线程模式还是处理器模
            式、使用PSP堆栈指针还是MSP堆栈指针，当调用 bx r14指令后，硬件会知道要从异常返
            回，然后出栈，这个时候堆栈指针PSP已经指向了新任务堆栈的正确位置，当新任务的运
            行地址被出栈到PC寄存器后，新的任务也会被执行。
64      nop
65  }
```

以上便是任务切换的所有过程，但这里其实还存在着两个问题

1. 处理器复位后默认使用的是 MSP 指针，它是怎么被切换成 PSP 的?
2. 新任务被进行第一次调度时，它的私有栈的值怎么处理?

对于第一个问题，任务调度器在启动时会调用 prvStartFirstTask() 函数，这个函数也是一段汇编代码，它的主要工作就是复位 MSP，开中断和异常，并且触发一次 SVC 中断，进行第一次任务的加载，其内容如下

```
1   __asm void prvStartFirstTask( void )
2   {
3       PRESERVE8
4
5       /* 定位主站指针初始位置 */
6       ldr r0, =0xE000ED08      //向量表偏移量寄存器的起始地址存储着MSP的初始值
7       ldr r0, [r0]
```

```
8       ldr r0, [r0]
9
10      /* 复位MSP */
11      msr msp, r0
12      /* 使能全局中断和异常 */
13      cpsie i
14      cpsie f
15      dsb
16      isb
17      /* 触发SVC中断. */
18      svc 0
19      nop
20      nop
21  }
```

再来看 SVC 异常服务函数里的代码

```
1   __asm void vPortSVCHandler( void )
2   {
3       PRESERVE8
4
5       /* 恢复任务现场 */
6       ldr r3, =pxCurrentTCB
7       ldr r1, [r3]
8       ldr r0, [r1]
9       ldmia r0!, {r4-r11}
10
11      /* 修改PSP */
12      msr psp, r0
13      isb
14      mov r0, #0
15      msr basepri, r0
16
17      /* 进入线程模式, 使用PSP指针 */
18      orr r14, #0xd           // 1101 每一位表示: 进入线程模式, 使用PSP, 必须为0, thumb
            状态
19      bx r14
20  }
```

看懂 PendSV 中的代码再看这段代码，就比较简单了，最为重要的是最后一步，其修改了 r14(EXC_RETURN) 的值，正是修改该值使得处理器在退出中断后运行任务函数时进入线程模式并使用 PSP 栈指针。

对于第二个问题，FreeRTOS 在添加新任务时，会调用 pxPortInitialiseStack() 函数来按处理器规则填充其私有栈的值，将任务的私有栈“伪装”成已经被调度过一次的样子

```
1   StackType_t *pxPortInitialiseStack( StackType_t *pxTopOfStack, TaskFunction_t pxCode,
        void *pvParameters )
2   {
3       // 这里空出一个存储地址是为了符合MCU进出中断的方式
4       pxTopOfStack--;
5       // 栈中寄存器xPSR被初始为0x01000000, 其中bit24被置1, 表示使用Thumb指令
6       *pxTopOfStack = portINITIAL_XPSR;
7       pxTopOfStack--;
8       // 这是将任务函数地址压入栈中程序PC(R15), 当该第一次切换任务时, 硬件的PC指针将指
            向该函数, 也就是会从头执行这个任务。
9       // & portSTART_ADDRESS_MASK是保证地址对齐
10      *pxTopOfStack = ( ( StackType_t ) pxCode ) & portSTART_ADDRESS_MASK;    /* PC */
11      pxTopOfStack--;
12      // 正常任务是死循环, 不会使用LR进行返回, 这里赋为错误处理函数地址, 出错时会进入该
            函数。
13      *pxTopOfStack = ( StackType_t ) prvTaskExitError;   /* LR */
14
15      // 跳过R12, R3, R2, R1不用初始化
16      pxTopOfStack -= 5;  /* R12, R3, R2 and R1. */
17      // 初始化参数地址
18      *pxTopOfStack = ( StackType_t ) pvParameters;   /* R0 */
19      pxTopOfStack -= 8;  /* R11, R10, R9, R8, R7, R6, R5 and R4. */
20
21      return pxTopOfStack;
```

```
22  }
```

以上便是 FreeRTOS 在 crotex-m3 平台上任务切换的全部过程。

2.4 任务调度器的启动与结束

2.4.1 任务调度器的启动

FreeRTOS 中任务调度器的启动由 vTaskStartScheduler() 函数实现，此函数被调用后，OS 将接手处理器的管理权；任务调度器的关闭由 vTaskEndScheduler() 函数实现，此函数调用后 OS 将停止工作。本节将对 FreeRTOS 的启动和结束过程进行分析。

vTaskStartScheduler() 函数启动 FreeRTOS 主要分为以下几个步骤

- 创建空闲任务、定时器任务。
- 初始化下一次解除阻塞时间，系统 tick 初始值，运行状态等变量。
- 调用 xPortStartScheduler() 函数启动调度器。

关于任务的创建过程已在上一节中详细分析了，这里对空闲任务的创建过程不再赘述，调用静态或动态创建函数即可。为什么需要空闲任务呢？对于许多嵌入式的处理器而言，其不提供 CPU 级的休眠，也就是说处理器是不停的运行的，必须提供一段正确的代码让其执行，否则处理器就可能发生错误，空闲任务正是这样一段“空跑”的代码，其它笔者接触的的 RTOS 也都有这样的空闲任务设计。当然空闲任务的“空跑”是打引号的，其还负责处理一些系统任务，如“任务的删除”中提及的清理删除任务占用的内存，再如监测系统的运行状态等待。

当空闲任务被成功创建之后（由于系统运行前至少会创建空闲任务，在创建第一个任务的过程中相关的链表已经相应的初始化，因此这里没看到初始相关链表的代码），下一次解除阻塞时间，系统 tick 初始值，运行状态等变量会被赋予合适的值。

最后一步是调用 xPortStartScheduler() 函数，这个函数是与平台相关的，根据 arm-cm3 的移植文件来看，它主要的工作是设置上下文切换中断和 systick 中断，启动定时器为系统提供 systick，最终调用 prvStartFirstTask() 来启动第一个任务。prvStartFirstTask() 这个函数也是与平台相关的，它的工作就是保证第一个任务能够正常的切换，在 arm-cm3 下，第一次任务切换的中断不是由 PendSV 触发的，而是使用触发 SVC 中断，进行上下文切换。当第一个任务开始运行时，整个 FreeRTOS 便接管处理器，开始自主调度任务。

2.4.2 任务调度器的结束

结束任务调度器的函数 vTaskEndScheduler() 更为简单，它只有三行

```
1  portDISABLE_INTERRUPTS();
2  xSchedulerRunning = pdFALSE;
3  vPortEndScheduler();
```

关中断，改变 xSchedulerRunning，调用 vPortEndScheduler() 处理平台硬件相关的结束操作。vPortEndScheduler() 在 arm-cm3 下的移植文件中为一个空函数，也就是说可以什么都不用处理。

第 3 章　内存管理

FreeRTOS 提供了 5 种不同的内存管理策略以应对不同的应用场景，本章将对这 5 种不同的内存管理策略的实现进行分析。（本章中部分图未画画出堆上字节对齐处理细节，请自行理解）

3.1　heap_1.c

heap_1.c 所实现的内存管理方法十分简单，其可以使用 pvPortMalloc() 函数来申请内存，一旦申请成功了，便无法被释放。其实现大致可以用一句话概括，**在堆空间剩余时，按需分割出内存，并记录已用的内存大小**。heap_1.c 使用的内存管理算法虽然简单，但对于许多嵌入式应用场景是适用且有效的。

在 heap_1.c 中，堆被定义为一个大数组，并使用变量 xNextFreeByte 记录已用内存大小

```
1  static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
2  static size_t xNextFreeByte = ( size_t ) 0;
```

核心代码 pvPortMalloc() 函数实现如下

```
1  void *pvPortMalloc( size_t xWantedSize )
2  {
3  void *pvReturn = NULL;
4  static uint8_t *pucAlignedHeap = NULL;
5
6      /* 如果对齐字节数不为1，则对请求的字节数做调整 */
7      #if( portBYTE_ALIGNMENT != 1 )
8      {
9          if( xWantedSize & portBYTE_ALIGNMENT_MASK )
10         {
11             xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                   portBYTE_ALIGNMENT_MASK ) );
12         }
13     }
14     #endif
15
16     /* 挂起任务 */
17     vTaskSuspendAll();
18     {
19         if( pucAlignedHeap == NULL )
20         {
21             /* 确保堆的起始地址是按字节对齐的（与操作可能会使地址值减小，因此使用&
                   ucHeap[portBYTE_ALIGNMENT]作为起始地址） */
22             pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE ) &ucHeap[
                   portBYTE_ALIGNMENT ] ) & ( ~( ( portPOINTER_SIZE_TYPE )
                   portBYTE_ALIGNMENT_MASK ) ) );
23         }
24
25         /* 检查是否有足够的空间分配 */
26         if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE ) &&
27             ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )/* 防止数值溢出 */
28         {
29             /* 返回首地址 */
30             pvReturn = pucAlignedHeap + xNextFreeByte;
31             /* 记录已分配空间大小 */
32             xNextFreeByte += xWantedSize;
33         }
34     }
35     /* 恢复任务 */
```

```
36      ( void ) xTaskResumeAll();
37
38      return pvReturn;
39  }
```

需要注意的是，并未使用 configTOTAL_HEAP_SIZE 代表堆大小，而是用 configADJUSTED_HEAP_SIZE 表示堆大小，configADJUSTED_HEAP_SIZE 定义如下

```
1  #define configADJUSTED_HEAP_SIZE    ( configTOTAL_HEAP_SIZE - portBYTE_ALIGNMENT )
```

这里简单粗暴的丢弃掉了对齐字节数个字节，以此来表示堆的起始地址对齐的操作中损失的字节数（最多不会损失掉对齐字节数个字节）。个人认为这里可以改进一下，节省使用内存，既计算出对齐字节操作具体损失的字节数计入后续运算中（其实也节省不了多少内存）。

3.2 heap_2.c

与 heap_1.c 不同，heap_2.c 允许使用 vPortFree() 函数来释放申请的内存，其算法原理是将空闲堆分为若干个大小不等的内存块，并将其按大小排序，使用单向链表连接起来，申请内存时，便从这些链表中寻找最小可满足申请需求的内存块进行分配。分配过程分为两步，首先将原先的内存块的链表项从链表中删除，其次是对当前内存块进行分割，将多余申请数的那部分内存变为新的链表项重新插入到链表中。释放过程则更为简单，只需要将释放的内存块重新插入到链表中即可。

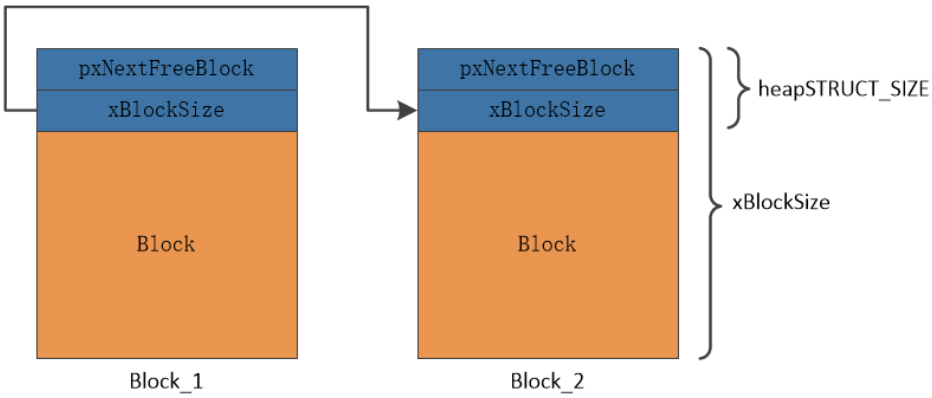
首先分析 heap_2.c 是如何表示一个内存块并将内存块加入链表中的。内存块的链表项定义如下

```
1  typedef struct A_BLOCK_LINK
2  {
3      struct A_BLOCK_LINK *pxNextFreeBlock;   /* 指向下一个未被使用的内存块 */
4      size_t xBlockSize;                      /* 内存块的大小（包括BlockLink_t头部大
           小） */
5  } BlockLink_t;
```

将内存块插入链表的操作如下

```
1  /* 按内存块大小排序，将内存块插入链表中，顺序是由小到大 */
2  #define prvInsertBlockIntoFreeList( pxBlockToInsert )                          \
3  {                                                                              \
4  BlockLink_t *pxIterator;                                                       \
5  size_t xBlockSize;                                                             \
6                                                                                 \
7      xBlockSize = pxBlockToInsert->xBlockSize;                                  \
8                                                                                 \
9      /* 按内存块的大小查找插入的位置 */                                          \
10     for( pxIterator = &xStart; pxIterator->pxNextFreeBlock->xBlockSize < xBlockSize;
           pxIterator = pxIterator->pxNextFreeBlock ) \
11     {                                                                          \
12                                                                                \
13     }                                                                          \
14                                                                                \
15     /* 插入操作 */                                                          \
16     pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;            \
17     pxIterator->pxNextFreeBlock = pxBlockToInsert;                             \
18 }
```

BlockLink_t 只描述了内存块的大小和内存块的链接关系，具体分配出的内存是如何表示的呢？其表示方式如下图所示



其实具体可分配的内存块是直接跟在 BlockLink_t 结构体后的，BlockLink_t 相当于一个内存块的头部。若有一个 BlockLink_t 实例为 Block2，则其表示的内存块地址范围为 &Block2+heapSTRUCT_SIZE 至 &Block2+xBlockSize 之间。为了字节对齐，heapSTRUCT_SIZE 是指 BlockLink_t 字节对齐后的大小，其定义如下

```
1  static const uint16_t heapSTRUCT_SIZE = ((sizeof(BlockLink_t) + (portBYTE_ALIGNMENT -
       1)) & ~portBYTE_ALIGNMENT_MASK);
```

3.2.1 堆的初始化

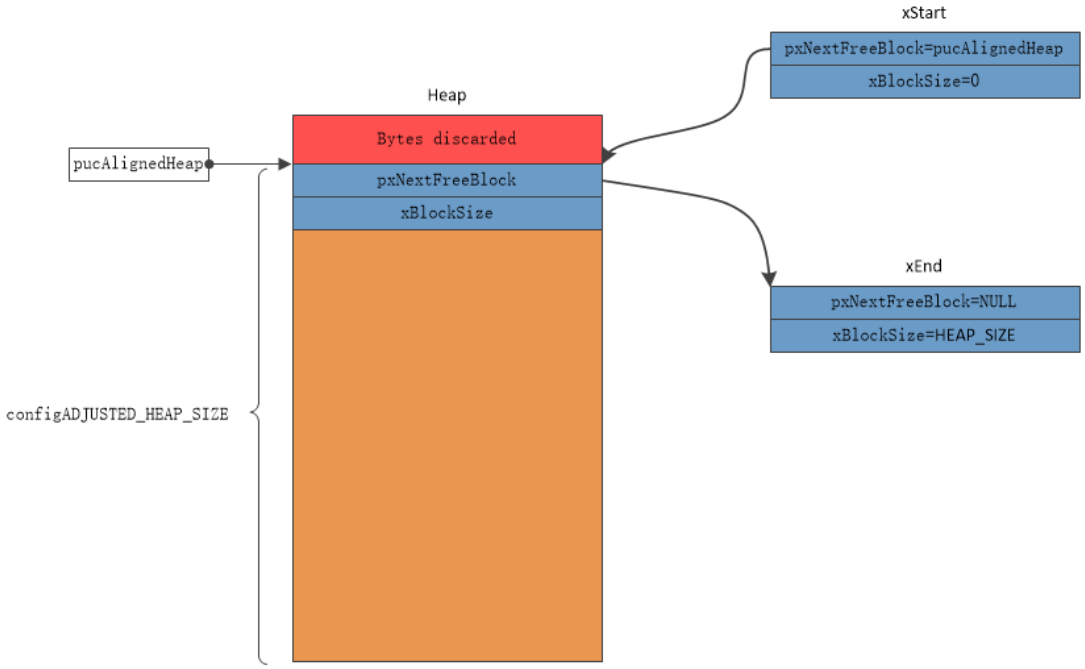
heap_2.c 的堆初始化函数如下

```
1  static void prvHeapInit(void)
2  {
3      BlockLink_t *pxFirstFreeBlock;
4      uint8_t *pucAlignedHeap;
5
6      /* 字节对齐，调整堆的起始地址。 */
7      pucAlignedHeap = (uint8_t *)(((portPOINTER_SIZE_TYPE)&ucHeap[portBYTE_ALIGNMENT])
           & (~((portPOINTER_SIZE_TYPE)portBYTE_ALIGNMENT_MASK)));
8
9      /* 初始化链表头xStart，其指向的下一个链表项是第一个普通的内存块，地址位于堆对齐操
           作后的起始地址 */
10     xStart.pxNextFreeBlock = (void *)pucAlignedHeap;
11     xStart.xBlockSize = (size_t)0;
12
13     /* 初始化链表尾xEnd */
14     xEnd.xBlockSize = configADJUSTED_HEAP_SIZE;
15     xEnd.pxNextFreeBlock = NULL;
16
17     /* 初始化第一个内存块，块大小为整个堆 */
18     pxFirstFreeBlock = (void *)pucAlignedHeap;
19     pxFirstFreeBlock->xBlockSize = configADJUSTED_HEAP_SIZE;
20     pxFirstFreeBlock->pxNextFreeBlock = &xEnd;
21 }
```

链表起始时只有三个内存块，一块是链表的头，被定义为 xStart，内存块大小为 0；另一块是链表的尾部，被定义为 xEnd，内存块大小为整个堆大小 configADJUSTED_HEAP_SIZE（configADJUSTED_HEA_SIZE 是堆对齐操作后的大小）。和普通内存块相比，xStart 和 xEnd 具有一些特殊性。

- 不参与实际的内存分配操作。
- xStart 和 xEnd 都不存储在堆上。

最后一个内存块是实际用于内存分配的普通块，其被定义在堆对齐后的起始地址上，块大小为整个堆大小 configADJUSTED_HEAP_SIZE。经初始化后，整个堆的状态可以用下图表示



3.2.2 内存分配

heap_2.c 的内存申请函数如下

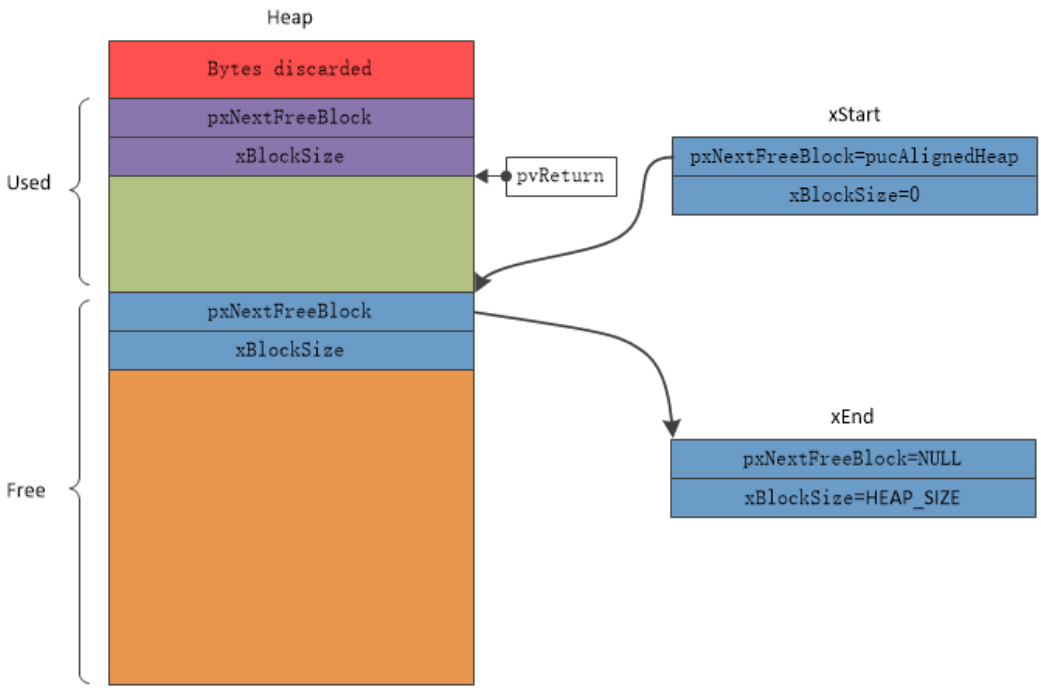
```
1  void *pvPortMalloc(size_t xWantedSize)
2  {
3      BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
4      static BaseType_t xHeapHasBeenInitialised = pdFALSE;
5      void *pvReturn = NULL;
6
7      vTaskSuspendAll();
8      {
9          /* 未进行堆初始化则进行堆初始化 */
10         if (xHeapHasBeenInitialised == pdFALSE)
11         {
12             prvHeapInit();
13             xHeapHasBeenInitialised = pdTRUE;
14         }
15
16         if (xWantedSize > 0)
17         {
18             /* 重新计算所需块大小 */
19             xWantedSize += heapSTRUCT_SIZE;
20
21             /*字节对齐操作，调整实际申请字节数 */
22             if ((xWantedSize & portBYTE_ALIGNMENT_MASK) != 0)
23             {
24                 xWantedSize += (portBYTE_ALIGNMENT - (xWantedSize &
                       portBYTE_ALIGNMENT_MASK));
25             }
26         }
27
28         if ((xWantedSize > 0) && (xWantedSize < configADJUSTED_HEAP_SIZE))
29         {
30             /* 查找合适的内存块进行分配，链表有序，由小到大依次查找即可 */
31             pxPreviousBlock = &xStart;
32             pxBlock = xStart.pxNextFreeBlock;
33             while ((pxBlock->xBlockSize < xWantedSize) && (pxBlock->pxNextFreeBlock
                   != NULL))
```

```
34              {
35                  pxPreviousBlock = pxBlock;
36                  pxBlock = pxBlock->pxNextFreeBlock;
37              }
38
39              /* 如果找到匹配的内存块，进行内存分配操作 */
40              if (pxBlock != &xEnd)
41              {
42                  /* 返回首地址值（跳过heapSTRUCT_SIZE） */
43                  pvReturn = (void *)(((uint8_t *)pxPreviousBlock->pxNextFreeBlock) +
                        heapSTRUCT_SIZE);
44
45                  /* 从链表中移除对应的内存块 */
46                  pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;
47
48                  /* 如果内存块较大，将剩余的内存作为一个新的内存块插入到链表中 */
49                  if ((pxBlock->xBlockSize - xWantedSize) > heapMINIMUM_BLOCK_SIZE)
50                  {
51                      /* 初始化相应的块头部BlockLink_t结构体 */
52                      pxNewBlockLink = (void *)(((uint8_t *)pxBlock) + xWantedSize);
53
54                      /* 更新相应的块地址大小 */
55                      pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
56                      pxBlock->xBlockSize = xWantedSize;
57
58                      /* 将分割出的新块插入链表中 */
59                      prvInsertBlockIntoFreeList((pxNewBlockLink));
60                  }
61                  /* 记录堆剩余字节数 */
62                  xFreeBytesRemaining -= pxBlock->xBlockSize;
63              }
64          }
65      }
66      (void)xTaskResumeAll();
67
68      return pvReturn;
69  }
```

阅读以上代码，可以总结出内存分配主要分为以下几个步骤

1. 调整实际需要申请的字节数。
2. 检测申请字节数是否合法，若合法，则寻找合适的内存块进行分配。
3. 将分配出去的内存块从链表中移除，若剩余内存大于最小内存块大小，则将剩余的内存块重新添加回链表中。
4. 记录剩余字节数，返回分配内存空间的地址。

堆在初始状态下，进行一次成功的内存分配后，其状态如下图所示

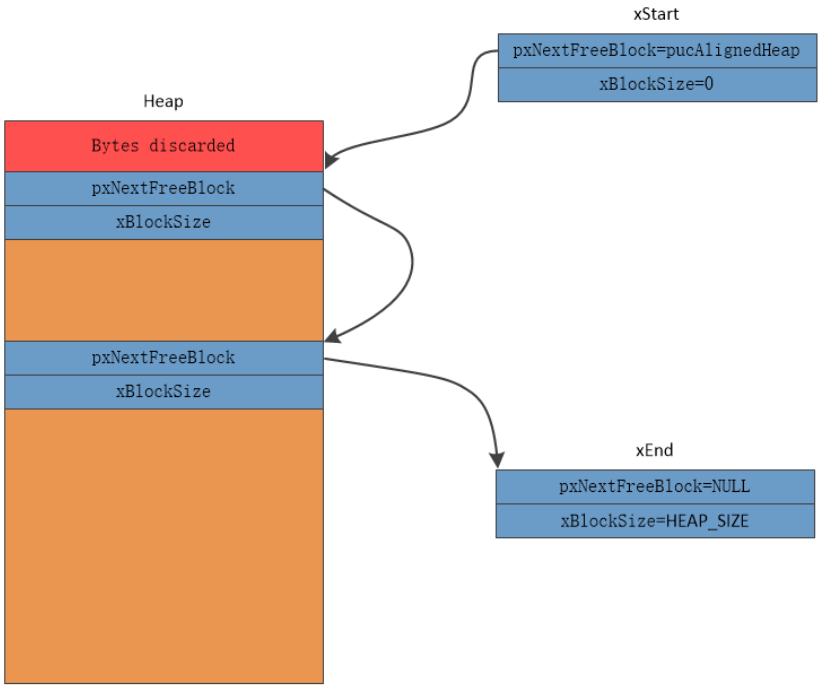


3.2.3 内存释放

根据前文的铺垫，很容易便可以想到，想要释放被占用的内存，只需要将对应的内存块重新添加到链表中即可。heap_2.c 的 vPortFree() 函数的实现如下

```
1  void vPortFree(void *pv)
2  {
3      uint8_t *puc = (uint8_t *)pv;
4      BlockLink_t *pxLink;
5
6      /* 判定释放地址是否合法 */
7      if (pv != NULL)
8      {
9          /* 定位内存块头部 */
10         puc -= heapSTRUCT_SIZE;
11
12         /* 注: FreeRTOS原注释为 This unexpected casting is to keep some compilers
               from
13           issuing byte alignment warnings. 笔者并不懂其具体作用*/
14         pxLink = (void *)puc;
15
16         vTaskSuspendAll();
17         {
18             /* 将内存块重新添加到链表中 */
19             prvInsertBlockIntoFreeList(((BlockLink_t *)pxLink));
20             xFreeBytesRemaining += pxLink->xBlockSize;
21         }
22         (void)xTaskResumeAll();
23     }
24 }
```

在 vPortFree() 的实现上，笔者认为在判定释放地址是否合法这一步上可以进一步提高安全性，**既判断 pv 值是否为堆的字节对齐地址，或则直接查询根据该地址定位出是否是某一空闲内存块的下一地址**。对上一节分配的内存进行释放后，堆的状态如下图所示



从上图可以看出，随着申请-释放的次数增加，heap_2.c 将使得内存块被越分越小，这会导致以下两个问题

1. 当需要再次请求一个大的内存块时，即使 xFreeBytesRemaining 大于请求的内存块，其也无法进行分配了。
2. 大量的内存被 BlockLink_t 头部占用，导致堆的利用率降低。

从上图中也可以得到解决方案，将相邻的内存块合并便可以缓解碎片化的问题，FreeRTOS 也提供了对应的实现 heap_4.c。

3.3 heap_3.c

heap_3.c 只是对 stdlib.h 中的 malloc() 函数做了简单的封装，这里不做分析。

3.4 heap_4.c

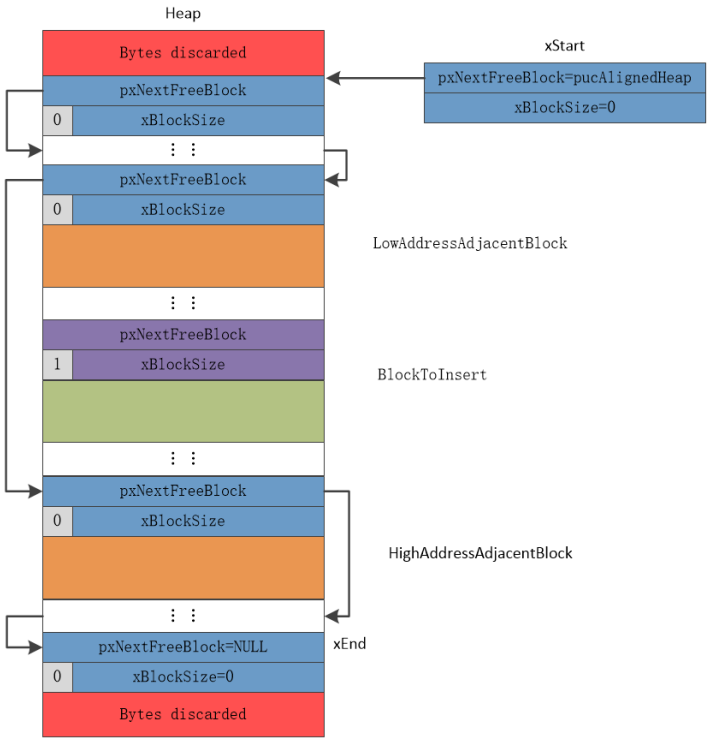
相比 heap_2.c，heap_4.c 可以实现相邻小内存块的合并，在一定程度上缓解内存碎片化的问题。

3.4.1 内存块链表的插入

在 pvPortMalloc()、pvPortFree()、prvHeapInit() 的实现方式与 heap_2.c 的实现思路大致相同，内存块合并算法主要是在 prvInsertBlockIntoFreeList() 函数中实现。与 heap_2.c 中按内存块大小顺序插入不同，heap_4.c 是按地址大小的顺序插入，这样便于合并相邻的内存块。插入过程分为以下两步

1. 查找链表中链接的下一个内存块地址大于待插入内存块地址的第一个内存块（也就是与待插入内存块相邻的且地址较低的那一个内存块）的地址。
2. 检测待插入内存块是否能与相邻的内存块合并。若能与低地址的相邻内存块合并，直接在低地址相邻的内存块大小上加上待插入内存块大小；若能与高地址的相邻内存块合并或可以同时将高低地址邻近内存块相连，则需要同时调整链表指针与内存块大小。

例如，想要将如下的使用中的内存块归还给堆



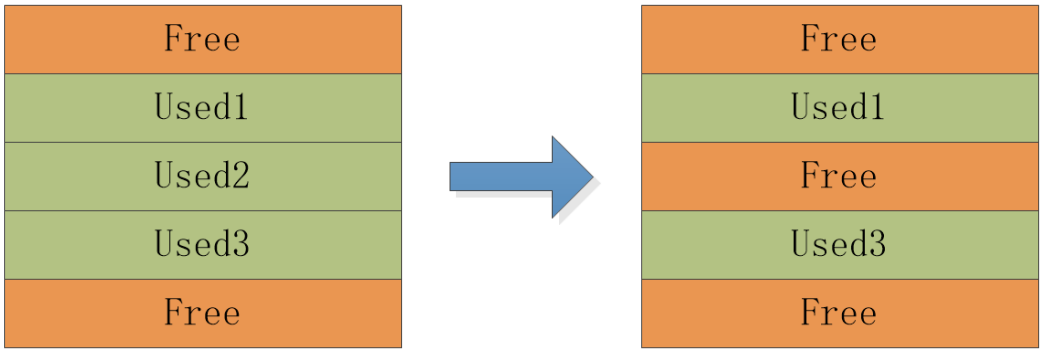
其会先遍历链表寻找 LowAddressAdjacentBlock，然后判断，若 BlockToInsert 仅能和 LowAddressAdjacentBlock 合并，则将 LowAddressAdjacentBlock 的块大小更改为 LowAddressAdjacentBlock 与 BlockToInsert 大小的和；若 BlockToInsert 仅能和 HighAddressAdjacentBlock 合并，则用 BlockToInsert 替换 HighAddressAdjacentBlock 在链表中的位置，并修改块大小为两者之和；若 BlockToInsert 能将 LowAddressAdjacentBlock 与 HighAddressAdjacentBlock 连接成一整块，则从链表中删除 HighAddressAdjacentBlock，并将 LowAddressAdjacentBlock 的块大小变为三者之和；若 BlockToInsert 是一个孤立的内存块则将其正常的插入到 LowAddressAdjacentBlock 与 HighAddressAdjacentBlock 之间

prvInsertBlockIntoFreeList() 函数的具体实现如下

```
1  static void prvInsertBlockIntoFreeList(BlockLink_t *pxBlockToInsert)
2  {
3      BlockLink_t *pxIterator;
4      uint8_t *puc;
5
6      /* 找到地址较低的邻近内存块地址 */
7      for (pxIterator = &xStart; pxIterator->pxNextFreeBlock < pxBlockToInsert;
           pxIterator = pxIterator->pxNextFreeBlock)
8      {
9
10     }
11
12     /* 检验待插入内存块是否紧接在低地址邻近内存块后 */
13     puc = (uint8_t *)pxIterator;
14     if ((puc + pxIterator->xBlockSize) == (uint8_t *)pxBlockToInsert)
15     {
16         /* 如果是，改变低地址邻近内存块的内存块大小 */
17         pxIterator->xBlockSize += pxBlockToInsert->xBlockSize;
18         /* 改变待插入内存块地址，将插入内存块与低地址邻近内存块邻近内存块合并后的块看
               作是新的待插入块 */
19         pxBlockToInsert = pxIterator;
20     }
21
22     /* 检验待高地址邻近内存块是否紧接在待插入内存块（或待插入内存块与低地址邻近内存块
           邻近内存块
23     合并后的块）后 */
24     puc = (uint8_t *)pxBlockToInsert;
```

```
25      if ((puc + pxBlockToInsert->xBlockSize) == (uint8_t *)pxIterator->pxNextFreeBlock
            )
26      {
27          if (pxIterator->pxNextFreeBlock != pxEnd)
28          {
29              /* 计算合并的内存块大小 */
30              pxBlockToInsert->xBlockSize += pxIterator->pxNextFreeBlock->xBlockSize;
31              /* 调整链表链接位置 */
32              pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock->
                    pxNextFreeBlock;
33          }
34          else
35          {
36              /* pxEnd特殊处理 */
37              pxBlockToInsert->pxNextFreeBlock = pxEnd;
38          }
39      }
40      else
41      {
42          /* 如果待插入内存块与高地址邻近内存块不能合并，调整待插入内存块的下一链接为高
                地址邻近内存块 */
43          pxBlockToInsert->pxNextFreeBlock = pxIterator->pxNextFreeBlock;
44      }
45
46      /* 如果pxIterator与pxBlockToInsert值不等，意味着低地址邻近内存块的内存块与待插入
            内存块并未合并，因此需要将待插入内存块挂接在pxIterator后面 */
47      if (pxIterator != pxBlockToInsert)
48      {
49          pxIterator->pxNextFreeBlock = pxBlockToInsert;
50      }
51  }
```

这样每次插入时，便可自动的合并掉相邻的内存块，以生成更大的内存块。这是否意味着内存的碎片化问题被解决了呢？并没有，可以看以下的一个示例，当其中的 Used2 被释放，是其仍然会产生内存碎片，除非 Used1 或 Used3 被释放，其才可能被拼接成较大的内存块。



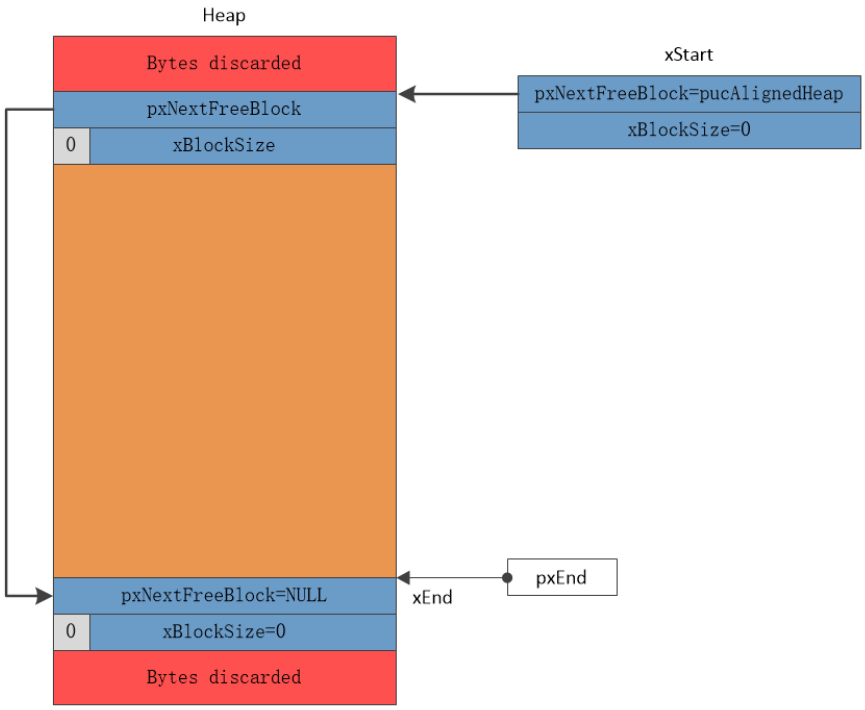
虽然未能彻底解决内存碎片化的问题，但这一方法相比 heap_2.c 有了很大的改进。

3.4.2 堆的初始化

heap_4.c 的堆初始化与 heap_2.c 的初始化大同小异，不同点有以下两点

1. 其使用 BlockLink_t 结构体成员 xBlockSize 的最高位来标记一个内存块是否被使用，1 表示在使用，0 表示空闲。
2. 原本的 xEnd 被定义在了堆上，且是堆的尾部，用 pxEnd 指向其地址。

heap_4.c 初始化堆后，堆状态为



对于其具体实现代码不做过多解释，可以参照之前的初始化代码来理解。这里将 xEnd 放到堆中是为了对链表尾部的内存块做合并处理，如果不将 xEnd 放到堆的尾部，那么若编译器分配 xEnd 地址时，将 xEnd 分配到一个堆前的地址，则此时堆底部的内存块想要进行释放操作时，按之前的插入算法其将不能被正确处理。那么 xStart 是否有必要也放到堆首呢？分析可知这是无必要的，xStart 的地址对于插入合并算法的执行没有影响。

3.4.3 内存的申请与释放

heap_4.c 的内存的申请与释放过程与 heap_2.c 相比除了增加了对 xBlockSize 最高位的处理外，没有太大的不同，这里便不加以赘述。

3.5 heap_5.c

之前的 heap_1.c，heap_2.c 和 heap_4.c 都将堆定义成了一个大数组，这意味着堆的地址必须是连续的，但在实际使用时，有有时需要管理两大块或更多的不连续内存，这时便可以使用 heap_5.c 这一实现。其实在之前的 heap_2.c 和 heap_4.c 中，已经实现了对不连续内存的管理，与 heap_4.c 相比，heap_5.c 的改变仅仅是在堆的初始化上，其将多个堆内存块加入了链表而已。

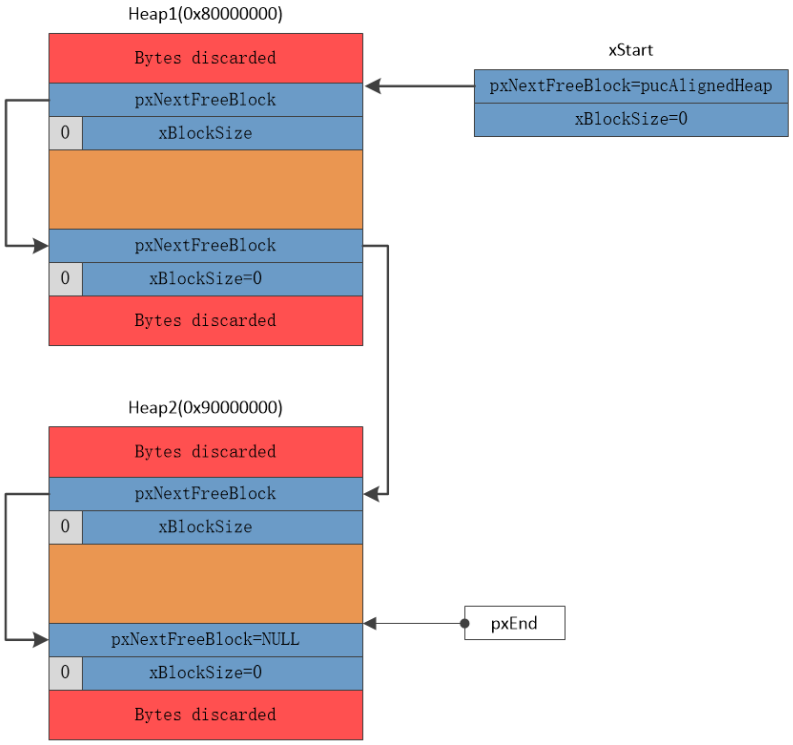
3.5.1 堆的初始化

heap_5.c 的堆初始化由 vPortDefineHeapRegions() 这一函数实现，其传入参数是一个具有特定格式的结构体数组。

```
1  typedef struct HeapRegion
2  {
3      uint8_t *pucStartAddress;
4      size_t xSizeInBytes;
5  } HeapRegion_t;
6
7  HeapRegion_t xHeapRegions[] =
8  {
9      /* 起始地址为0x80000000，大小为0x10000的内存块 */
10     {(uint8_t *)0x80000000UL, 0x10000},
11     /* 起始地址为0x90000000，大小为0xa0000的内存块，地址递增排序 */
12     {(uint8_t *)0x90000000UL, 0xa0000},
13     /* 结束标识符 */
14     {NULL, 0}
```

```
15  };
```

vPortDefineHeapRegions() 所做的工作就是读取结构体数组中的每一个内存块信息，并将其编入链表中。以以上参数为例，初始化后堆的状态如下图所示



具体 vPortDefineHeapRegions() 的代码实现较为简单，这里不进行具体分析。

3.5.2 内存分配、释放等

参照 heap_4.c 的实现原理即可。