RTOS通常将内核与内存管理分开实现

操作系统内核仅规定了必要的内存管理函数原型，而不关心这些内存管理函数是如何实现的。

这样做可以增加系统的灵活性：不同的应用场合可以使用不同的内存分配实现，选择对自己更有利的内存管理策略。

比如对于安全型的嵌入式系统，通常不允许动态内存分配，那么可以采用非常简单的内存管理策略，一经申请的内存，甚至不允许被释放。

在满足设计要求的前提下，系统越简单越容易做的更安全。

再比如一些复杂应用，要求动态的申请、释放内存操作，那么也可以设计出相对复杂的内存管理策略，允许动态分配和动态释放

FreeRTOS内存管理方案

(69条消息) FreeRTOS高级篇7---FreeRTOS内存管理分析_研究是为了理解的博客-CSDN博客

FreeRTOS提供的内存管理都是从内存堆中分配内存的。默认情况下，FreeRTOS内核创建任务、队列、信号量、事件组、软件定时器都是借助内存管理函数从内存堆中分配内存, 最新的FreeRTOS版本（V9.0.0及其以上版本）可以完全使用静态内存分配方法，也就是不使用任何内存堆。

对于heap_1.c、heap_2.c和heap_4.c这三种内存管理策略，内存堆实际上是一个很大的数组，定义为

```
static unsigned char ucHeap[ configTOTAL_HEAP_SIZE ];
//宏configTOTAL_HEAP_SIZE用来定义内存堆的大小，这个宏在FreeRTOSConfig.h中设置
```

heap_1

这是5个内存管理策略中最简单的一个，它简单到只能申请内存, 不能释放内存

对于大多数嵌入式系统，特别是对安全要求高的嵌入式系统，这种内存管理策略很有用

因为对系统软件来说，逻辑越简单越容易兼顾安全。

实际上，大多数的嵌入式系统并不需要动态删除任务、信号量、队列等，

而是在初始化的时候一次性创建好，便一直使用，永远不用删除。所以这个内存管理策略实现简洁、安全可靠，使用的非常广泛

可以将heap_1内存管理看作是切面包：

初始化的内存就像一根完整的长棍面包，每次申请内存，就从一端切下适当长度的面包返还给申请者，直到面包被分配完毕,

就这么简单

heap_1内存管理策略使用两个局部静态变量来跟踪内存分配，变量定义为

```
static size_t xNextFreeByte = ( size_t ) 0;
    //记录已经分配的内存大小，用来定位下一个空闲的内存堆位置。因为内存堆实际上是一个大数组，我们只需要知道已分配内存的大小，就可以用它作为偏移量找到未分配内存的起始地址。变量xNextFreeByte被初始化为0，然后每次申请内存成功后，都会增加申请内存的字节数目。
static uint8_t *pucAlignedHeap = NULL;
    //指向对齐后的内存堆起始位置。为什么要对齐？这是因为大多数硬件访问内存对齐的数据速度会更快。为了提高性能，FreeRTOS会进行对齐操作，不同的硬件架构对齐操作也不尽相同，对于Cortex-M3架构，进行8字节对齐
```

内存申请：pvPortMalloc()

```
void *pvPortMalloc( size_t xWantedSize )
{
void *pvReturn = NULL;
static uint8_t *pucAlignedHeap = NULL;


    /* 确保申请的字节数是对齐字节数的倍数 */
    #if( portBYTE_ALIGNMENT != 1 )
    {
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
        }
    }
    #endif

    vTaskSuspendAll();
    {
        if( pucAlignedHeap == NULL )
        {
            /* 第一次使用,确保内存堆起始位置正确对齐 */
            pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE ) &ucHeap[
portBYTE_ALIGNMENT ] ) & ( ~( ( portPOINTER_SIZE_TYPE ) portBYTE_ALIGNMENT_MASK )
) );
        }


        /* 边界检查,变量xNextFreeByte是局部静态变量,初始值为0 */
        if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE ) &&
            ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )
        {
            /* 返回申请的内存起始地址并更新索引 */
            pvReturn = pucAlignedHeap + xNextFreeByte;
            xNextFreeByte += xWantedSize;
        }
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
```

```
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}
```

函数一开始会将申请的内存数量调整到对齐字节数的整数倍，所以实际分配的内存空间可能比申请内存大。

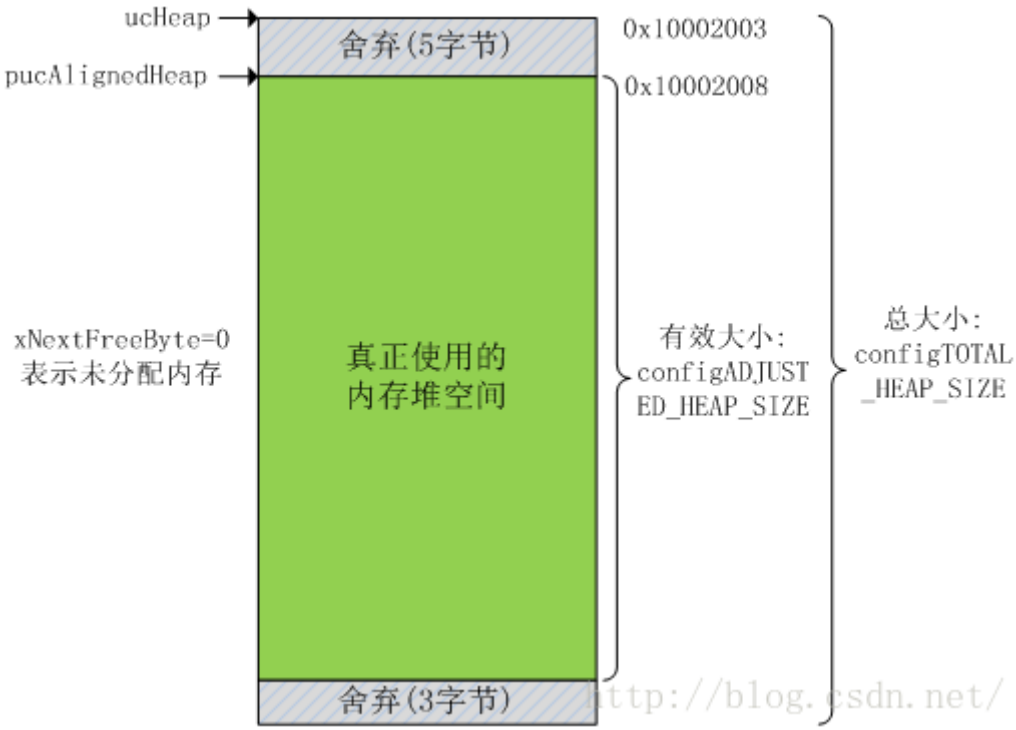
比如对于8字节对齐的系统，申请11字节内存，经过对齐后，实际分配的内存是16字节（8的整数倍）。

接下来会挂起所有任务，因为内存申请是不可重入的（使用了静态变量）。

如果是第一次执行这个函数，需要将变量pucAlignedHeap指向内存堆区域第一个地址对齐处。

内存堆其实是一个大数组，编译器为这个数组分配的起始地址是随机的，可能不符合我们的对齐需要，这时候要进行调整。

比如内存堆数组ucHeap从RAM地址0x10002003处开始，系统按照8字节对齐，则对齐后的内存堆如下图所示



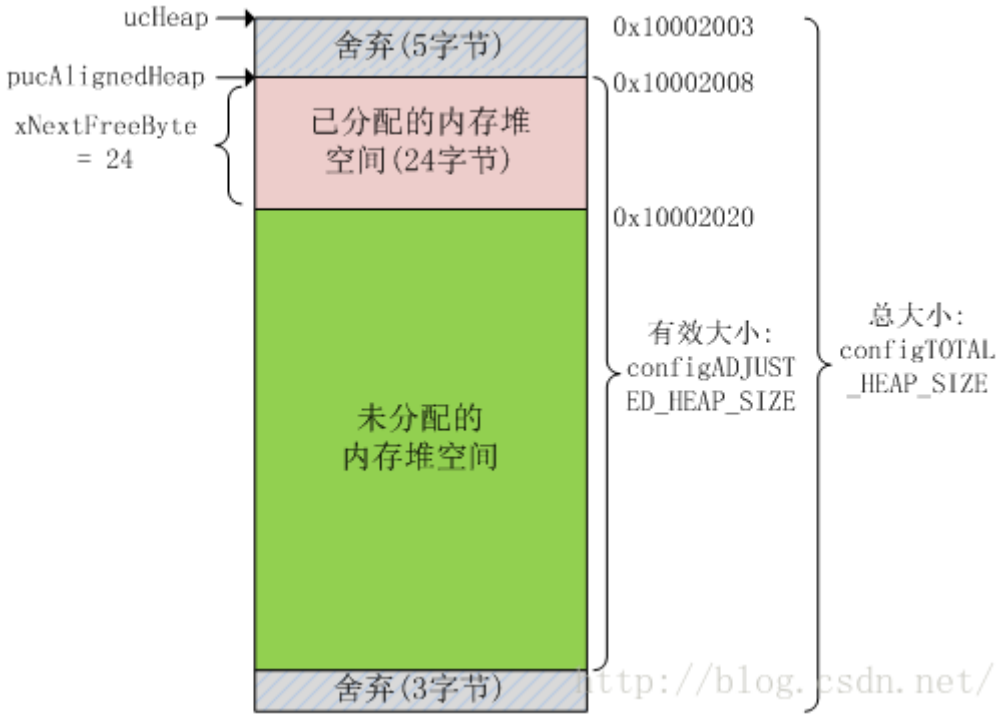
之后进行边界检查，查看剩余的内存堆是否够分配，检查xNextFreeByte + xWantedSize是否溢出。

如果检查通过，则为申请者返回有效的内存指针并更新已分配内存数量计数器xNextFreeByte

（从指针pucAlignedHeap开始，偏移量为xNextFreeByte处的内存区域为未分配的内存堆起始位置）。

比如我们首次调用内存分配函数pvPortMalloc(20)，申请20字节内存。

根据对齐原则，我们会实际申请到24字节内存，申请成功后，内存堆示意图如下图所示



内存分配完成后，不管有没有分配成功都恢复之前挂起的调度器。

如果内存分配不成功，这里最可能是内存堆空间不够用了，会调用一个钩子函数vApplicationMallocFailedHook()。

这个钩子函数由应用程序提供，通常我们可以打印内存分配设备信息或者点亮也故障指示灯。

`获取当前未分配的内存堆大小：xPortGetFreeHeapSize()`

函数用于返回未分配的内存堆大小。这个函数也很有用，通常用于检查我们设置的内存堆是否合理，通过这个函数我们可以估计出最坏情况下需要多大的内存堆，以便合理的节省RAM。

```
size_t xPortGetFreeHeapSize( void )
{
    return ( configADJUSTED_HEAP_SIZE - xNextFreeByte );
}
```

`其他函数`

heap_1内存管理策略中还有两个函数：vPortFree()和vPortInitialiseBlocks()。但实际上第一个函数什么也不做；第二个函数仅仅将静态局部变量xNextFreeByte设置为0

heap_2

heap_2内存管理策略要比heap_1内存管理策略复杂

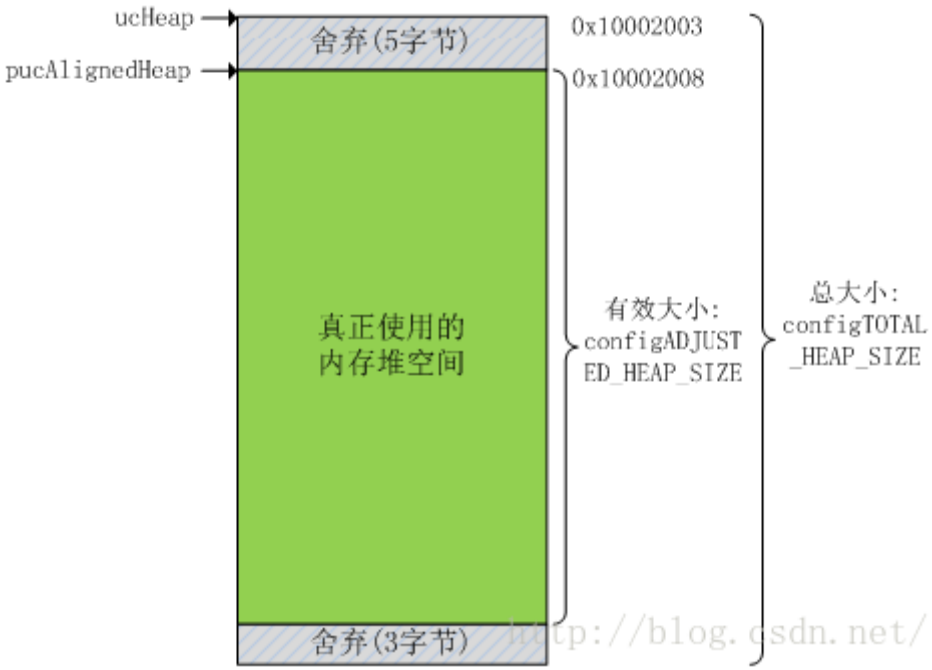
它使用一个最佳匹配算法，允许释放之前已分配的内存块

但是它不会把相邻的空闲块合成一个更大的块（换句话说，这会造成内存碎片）

heap_2内存管理策略用于重复的分配和删除具有相同堆栈空间的任务、队列、信号量、互斥量等等，并且不考虑内存碎片的应用程序

不适用于分配和释放随机字节堆栈空间的应用程序！

局部静态变量pucAlignedHeap指向对齐后的内存堆起始位置。地址对齐的原因在第一种内存管理策略中已经说明。假如内存堆数组ucHeap从RAM地址0x10002003处开始，系统按照8字节对齐，则对齐后的内存堆与第一个内存管理策略一样, 如下图所示：



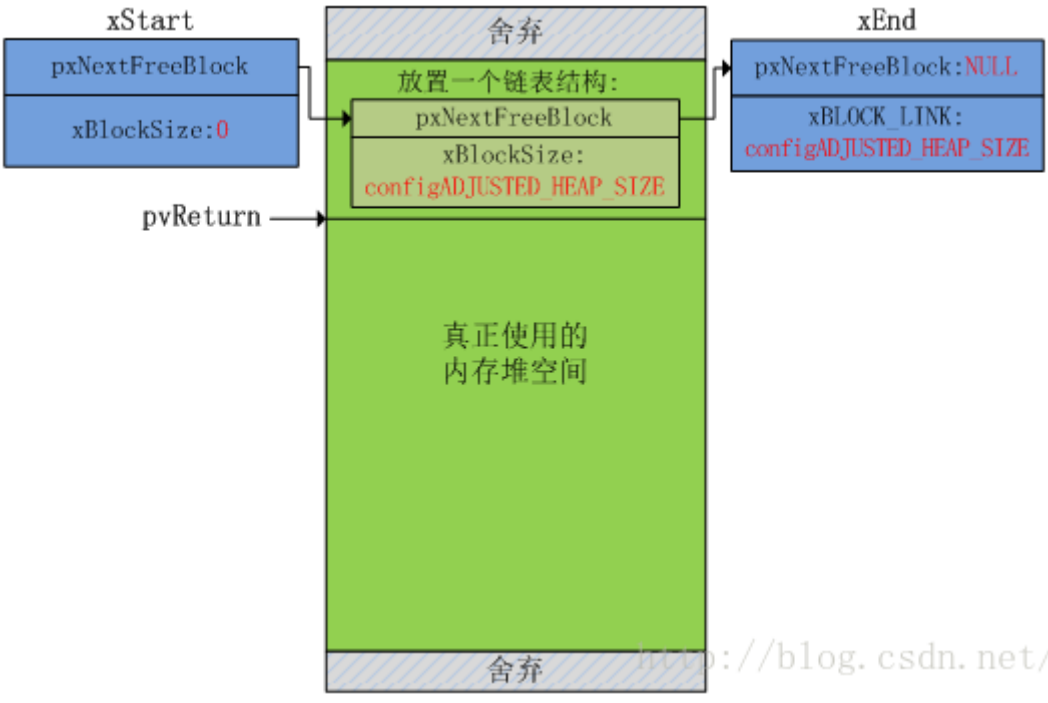
内存申请：pvPortMalloc()

与heap_1内存管理策略不同，heap_2内存管理策略使用一个链表结构来跟踪记录空闲内存块，将空闲块组成一个链表

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    /*指向列表中下一个空闲块*/
    size_t xBlockSize;                       /*当前空闲块的大小，包括链表结构大小*/
} BlockLink_t;
```

两个BlockLink_t类型的局部静态变量xStart和xEnd用来标识空闲内存块的起始和结束

刚开始时，整个内存堆有效空间就是一个空闲块，如下所示



整个有效空间组成唯一一个空闲块，在空闲块的起始位置放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。

由于目前只有一个空闲块，所以空闲块的pxNextFreeBlock指向链表xEnd，而链表xStart结构的pxNextFreeBlock指向空闲块。

这样，xStart、空闲块和xEnd组成一个单链表，xStart表示链表头，xEnd表示链表尾。

> 随着内存申请和释放，空闲块可能会越来越多，但它们仍是以`xStart`链表开头以`xEnd`链表结尾，根据空闲块的大小排序，小的在前，大的在后
> 当申请N字节内存时，实际上不仅需要分配N字节内存，还要分配一个`BlockLink_t`类型结构体空间，用于描述这个内存块，结构体空间位于空闲内存块的最开始处。当然，和heap_1内存管理策略一样，申请的内存大小和`BlockLink_t`类型结构体大小都要向上扩大到对齐字节数的整数倍。

内存申请过程：

首先计算实际要分配的内存大小，判断申请的内存是否合法。

如果合法则从链表头xStart开始查找，如果某个空闲块的xBlockSize字段大小能容得下要申请的内存

则从这块内存取出合适的部分返回给申请者，剩下的内存块组成一个新的空闲块，按照空闲块的大小顺序插入到空闲块链表中，小块在前大块在后。

注意，返回的内存中不包括链表结构，而是紧邻链表结构（经过对齐）后面的位置。

举个例子，如上图所示的内存堆，当调用申请内存函数

如果内存堆空间足够大，就将pvReturn指向的地址返回给申请者，而不是静态变量pucAlignedHeap指向的内存堆起始位置！
当多次调用内存申请函数后（没有调用内存释放函数），内存堆结构如下图所示。注意图中的pvReturn仍是我自己增加上去的，pvReturn指向的位置返回给申请者。后面我们讲内存释放时，就是根据这个地址完成内存释放工作的。

函数中使用的一个静态局部变量xFreeBytesRemaining，它用来记录未分配的内存堆大小。这个变量将提供给函数xPortGetFreeHeapSize()使用，以方便用户估算内存堆使用情况

```
void *pvPortMalloc( size_t xWantedSize )
{
BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
static BaseType_t xHeapHasBeenInitialised = pdFALSE;
void *pvReturn = NULL;

    /* 挂起调度器 */
    vTaskSuspendAll();
    {
        /* 如果是第一次调用内存分配函数,这里先初始化内存堆 */
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }
```

```
        /* 调整要分配的内存值,需要增加上链表结构体空间,heapSTRUCT_SIZE表示经过对齐扩展后的结构体大小 */
        if( xWantedSize > 0 )
        {
            xWantedSize += heapSTRUCT_SIZE;


            /* 调整实际分配的内存大小,向上扩大到对齐字节数的整数倍 */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize & portBYTE_ALIGNMENT_MASK ) );
            }
        }

        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE ) )
        {
            /* 空闲内存块是按照块的大小排序的,从链表头xStart开始,小的在前大的在后,以链表尾xEnd结束 */
            pxPreviousBlock = &xStart;
            pxBlock = xStart.pxNextFreeBlock;
            /* 搜索最合适的空闲块 */
            while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pxNextFreeBlock != NULL ) )
            {
                pxPreviousBlock = pxBlock;
                pxBlock = pxBlock->pxNextFreeBlock;
            }


            /* 如果搜索到链表尾xEnd,说明没有找到合适的空闲内存块,否则进行下一步处理 */
            if( pxBlock != &xEnd )
            {
                /* 返回内存空间,注意是跳过了结构体BlockLink_t空间. */
                pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->pxNextFreeBlock ) + heapSTRUCT_SIZE );


                /* 这个块就要返回给用户,因此它必须从空闲块中去除. */
                pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;


                /* 如果这个块剩余的空间足够多,则将它分成两个,第一个返回给用户,第二个作为新的空闲块插入到空闲块列表中去*/
                if( ( pxBlock->xBlockSize - xWantedSize ) > heapMINIMUM_BLOCK_SIZE )
                {
                    /* 去除分配出去的内存,在剩余内存块的起始位置放置一个链表结构并初始化链表成员 */
                    pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) + xWantedSize );


                    pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
                    pxBlock->xBlockSize = xWantedSize;
```

```
                    /* 将剩余的空闲块插入到空闲块列表中,按照空闲块的大小顺序,小的在前大的在后 */
                    prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
                }
                /* 计算未分配的内存堆大小,注意这里并不能包含内存碎片信息 */
                xFreeBytesRemaining -= pxBlock->xBlockSize;
            }
        }

        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {   /* 如果内存分配失败,调用钩子函数 */
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif
    return pvReturn;
}
```

内存释放：vPortFree()

因为不需要合并相邻的空闲块，第二种内存管理策略的内存释放也非常简单

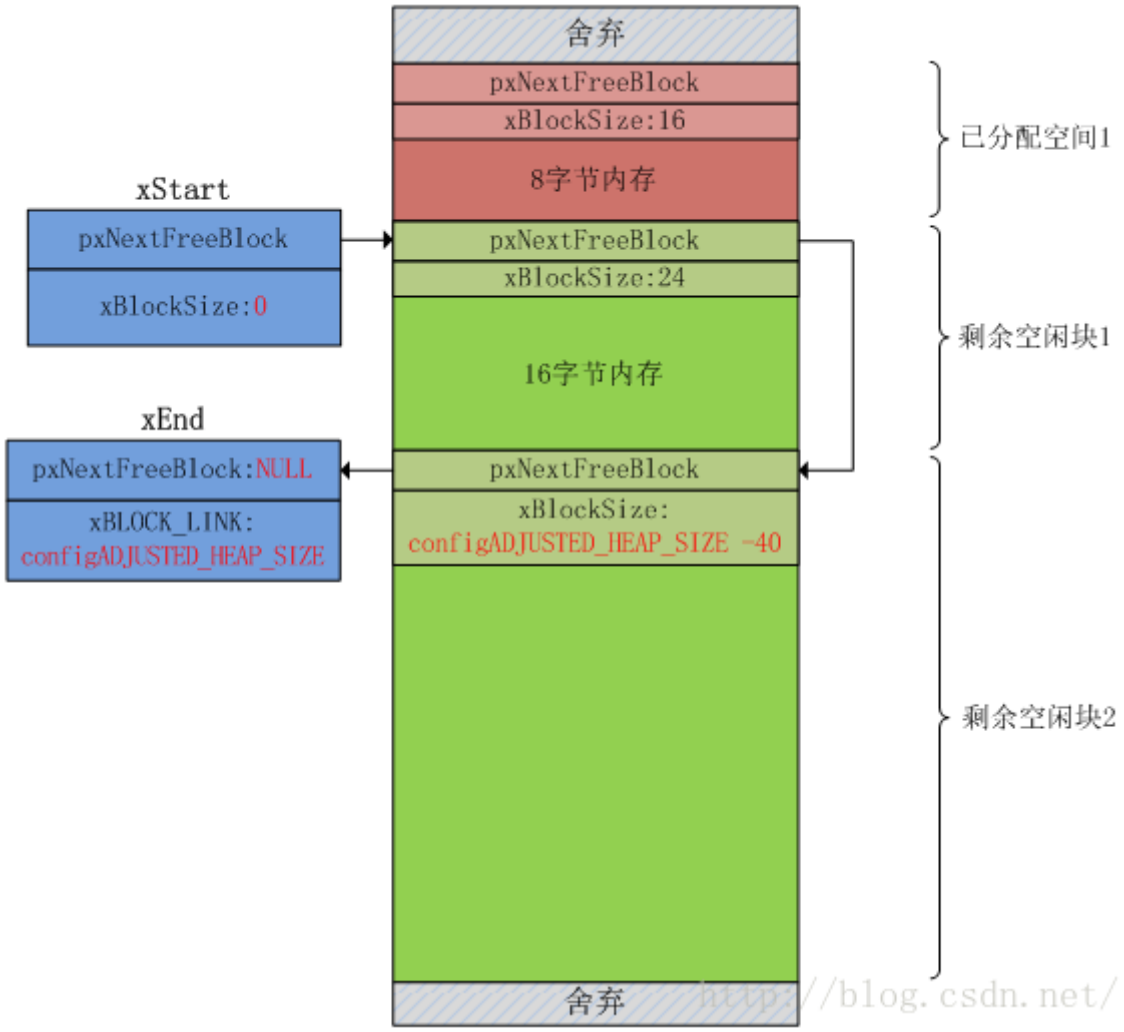
根据传入的参数找到链表结构，然后将这个内存块插入到空闲块列表，更新未分配的内存堆计数器大小

```
void vPortFree( void *pv )
{
uint8_t *puc = ( uint8_t * ) pv;
BlockLink_t *pxLink;

    if( pv != NULL )
    {
        /* 根据传入的参数找到链表结构 */
        puc -= heapSTRUCT_SIZE;

        /* 预防某些编译器警告 */
        pxLink = ( void * ) puc;

        vTaskSuspendAll();
        {
            /* 将这个块添加到空闲块列表 */
```

```
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
            /* 更新未分配的内存堆大小 */
            xFreeBytesRemaining += pxLink->xBlockSize;

            traceFREE( pv, pxLink->xBlockSize );
        }
        ( void ) xTaskResumeAll();
    }
}
```

举一个例子，将上图的pvReturn指向的内存块释放掉，假设（configADJUSTED_HEAP_SIZE-40）远大于要释放的内存块大小， 如下图所示

-



可以看出heap_2内存管理策略的两个特点：

第一，空闲块是按照大小排序的；

第二，相邻的空闲块不会组合成一个大块。

再接着引申讨论一下heap_2内存管理策略的优缺点。

通过对内存申请和释放函数源码分析，一个优点是速度足够快，因为它的实现非常简单，第二个优点是可以动态释放内存。

但是它的缺点也非常明显：由于在释放内存时不会将相邻的内存块合并，所以这可能造成内存碎片。

这就对其应用的场合要求极其苛刻：

第一，每次创建或释放的任务、信号量、队列等必须大小相同，如果分配或释放的内存是随机的，绝对不可以用这种内存管理策略；

第二，如果申请和释放的顺序不可预料，也很危险。举个例子，对于一个已经初始化的10KB内存堆，先申请48字节内存，然后释放；

再接着申请32字节内存，那么一个本来48字节的大块就会被分为32字节和16字节的小块，

如果这种情况经常发生，就会导致每个空闲块都可能很小

最终在申请一个大块时就会因为没有合适的空闲块而申请失败（并不是因为总的空闲内存不足）！

`获取未分配的内存堆大小:xPortGetFreeHeapSize()`

函数用于返回未分配的内存堆大小。这个函数也很有用，通常用于检查我们设置的内存堆是否合理，通过这个函数我们可以估计出最坏情况下需要多大的内存堆，以便进行合理的节省RAM。

heap_3

该内存管理策略简单的封装了标准库中的malloc()和free()函数

采用的封装方式是操作内存前挂起调度器、完成后再恢复调度器。

封装后的malloc()和free()函数具备线程保护

heap_1和heap_2内存管理策略都是通过定义一个大数组作为内存堆，数组的大小由宏configTOTAL_HEAP_SIZE指定。

heap_3内存管理策略与前两种不同，它不再需要通过数组定义内存堆，而是需要使用编译器设置内存堆空间

一般在启动代码中设置。

因此宏configTOTAL_HEAP_SIZE对这种内存管理策略是无效的

heap_4

heap_4与heap_2相似，只不过增加了一个合并算法，将相邻的空闲内存块合并成一个大块

与heap_2内存管理策略一样，空闲内存块也是以单链表的形式组织起来的

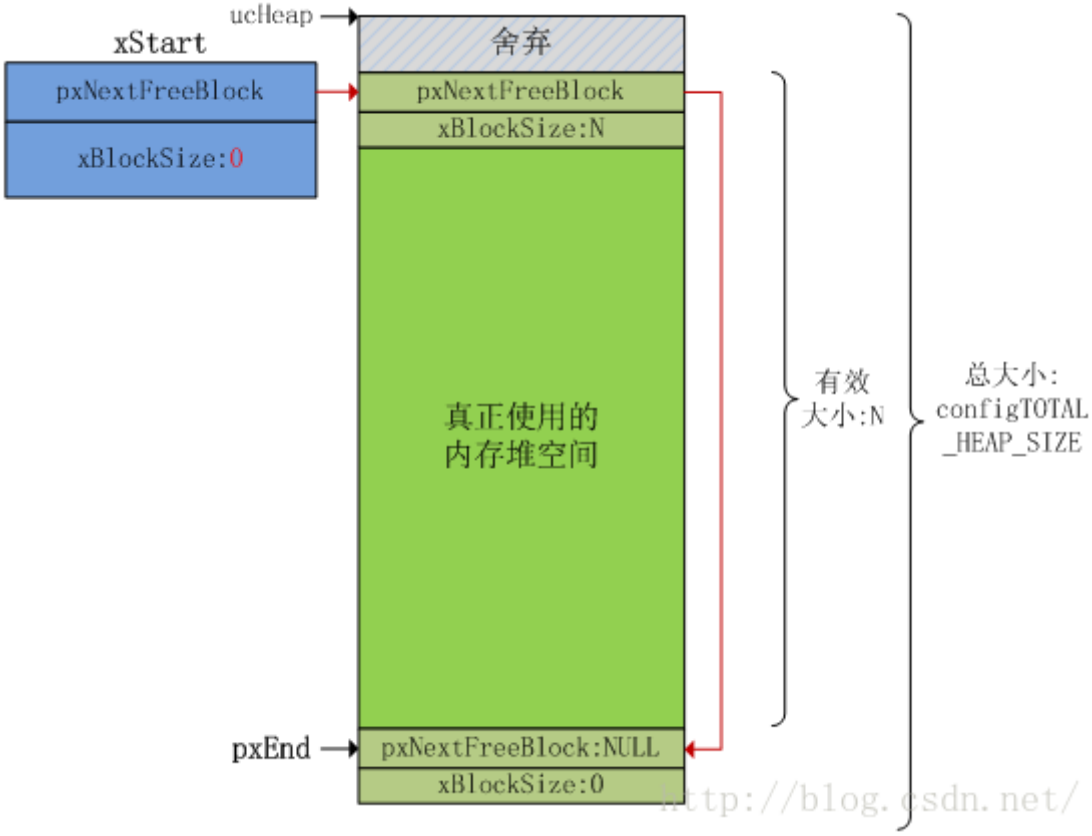
BlockLink_t类型的局部静态变量xStart表示链表头，

但heap_4内存管理策略的链表尾保存在内存堆空间最后位置

并使用BlockLink_t指针类型局部静态变量pxEnd指向这个区域（heap_2内存管理策略使用静态变量xEnd表示链表尾），如下图所示。

heap_4内存管理策略和heap_2内存管理策略还有一个很大的不同是：

heap_4内存管理策略的空闲块链表不是以内存块大小为存储顺序，而是以内存块起始地址大小为存储顺序，地址小的在前，地址大的在后。这也是为了适应合并算法而作的改变



整个有效空间组成唯一一个空闲块，在空闲块的起始位置放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。由于目前只有一个空闲块，所以空闲块的pxNextFreeBlock指向指针pxEnd指向的位置，而链表xStart结构的pxNextFreeBlock指向空闲块。xStart表示链表头，pxEnd指向位置表示链表尾

当申请x字节内存时，实际上不仅需要分配x字节内存，还要分配一个BlockLink_t类型结构体空间，用于描述这个内存块，结构体空间位于空闲内存块的最开始处。当然，和heap_1、heap_2内存管理策略一样，申请的内存大小和BlockLink_t类型结构体大小都要向上扩大到对齐字节数的整数倍。

内存申请过程：

首先计算实际要分配的内存大小，判断申请内存合法性，

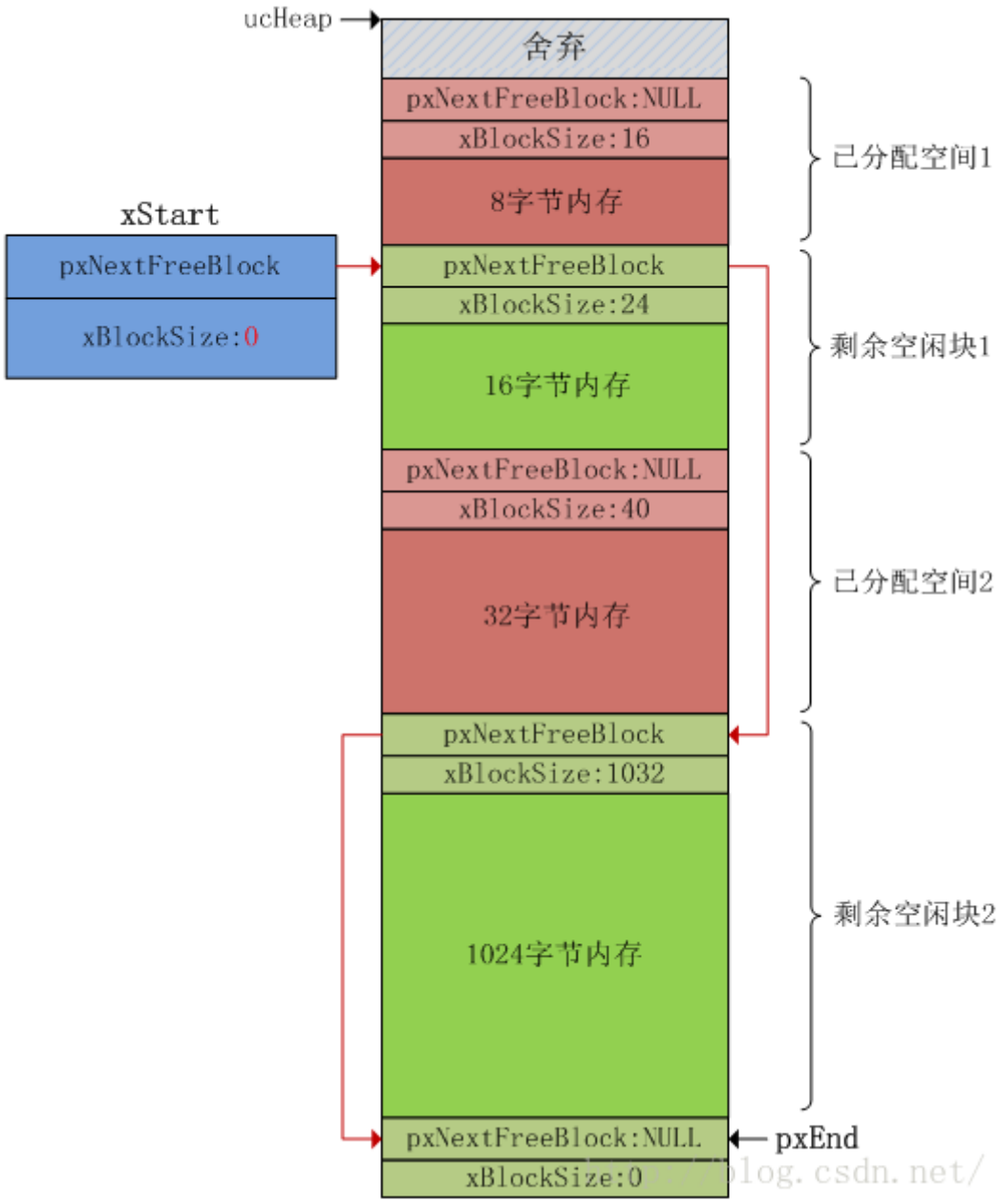
如果合法则从链表头xStart开始查找，

如果某个空闲块的xBlockSize字段大小能容得下要申请的内存，则将这块内存取出合适的部分返回给申请者，剩下的内存块组成一个新的空闲块

按照空闲块起始地址大小顺序插入到空闲块链表中，地址小的在前，地址大的在后。

在插入到空闲块链表的过程中，还会执行合并算法：判断这个块是不是可以和上一个空闲块合并成一个大块，如果可以则合并；

然后再判断能不能和下一个空闲块合并成一个大块，如果可以则合并！合并算法是heap_4内存管理策略和heap_2内存管理策略最大的不同!经过几次内存申请和释放后，可能的内存堆如下图所示：



函数中会用到几个局部静态变量在这里简单说明一下：

- xFreeBytesRemaining：表示当前未分配的内存堆大小
- xMinimumEverFreeBytesRemaining：表示未分配内存堆空间历史最小值。这个值跟xFreeBytesRemaining有很大区别，只有记录未分配内存堆的最小值，才能知道最坏情况下内存堆的使用情况。
- xBlockAllocatedBit：这个变量在第一次调用内存申请函数时被初始化，将它能表示的数值的最高位置1。比如对于32位系统，这个变量被初始化为0x80000000（最高位为1）。内存管理策略使用这个变量来标识一个内存块是否空闲。如果内存块被分配出去，则内存块链表结构成员xBlockSize按位或上这个变量（即xBlockSize最高位置1），在释放一个内存块时，会把xBlockSize的最高位清零。

```
void *pvPortMalloc( size_t xWantedSize )
{
BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
void *pvReturn = NULL;


    vTaskSuspendAll();
```

```
    {
        /* 如果是第一次调用内存分配函数,则初始化内存堆,初始化后的内存堆如图4-1所示 */
        if( pxEnd == NULL )
        {
            prvHeapInit();
        }


        /* 申请的内存大小合法性检查:是否过大.结构体BlockLink_t中有一个成员xBlockSize表示块
的大小,这个成员的最高位被用来标识这个块是否空闲.因此要申请的块大小不能使用这个位.*/
        if( ( xWantedSize & xBlockAllocatedBit ) == 0 )
        {
            /* 计算实际要分配的内存大小,包含链接结构体BlockLink_t在内,并且要向上字节对齐 */
            if( xWantedSize > 0 )
            {
                xWantedSize += xHeapStructSize;


                /* 对齐操作,向上扩大到对齐字节数的整数倍 */
                if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0x00 )
                {
                    xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
portBYTE_ALIGNMENT_MASK ) );
                    configASSERT( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) == 0
);
                }
            }


            if( ( xWantedSize > 0 ) && ( xWantedSize <= xFreeBytesRemaining ) )
            {
                /* 从链表xStart开始查找,从空闲块链表(按照空闲块地址顺序排列)中找出一个足够大
的空闲块 */
                pxPreviousBlock = &xStart;
                pxBlock = xStart.pxNextFreeBlock;
                while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock-
>pxNextFreeBlock != NULL ) )
                {
                    pxPreviousBlock = pxBlock;
                    pxBlock = pxBlock->pxNextFreeBlock;
                }


                /* 如果最后到达结束标识,则说明没有合适的内存块,否则,进行内存分配操作*/
                if( pxBlock != pxEnd )
                {
                    /* 返回分配的内存指针,要跳过内存开始处的BlockLink_t结构体 */
                    pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock-
>pxNextFreeBlock ) + xHeapStructSize );


                    /* 将已经分配出去的内存块从空闲块链表中删除 */
                    pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;


                    /* 如果剩下的内存足够大,则组成一个新的空闲块 */
```

```
                    if( ( pxBlock->xBlockSize - xWantedSize ) >
heapMINIMUM_BLOCK_SIZE )
                    {
                        /* 在剩余内存块的起始位置放置一个链表结构并初始化链表成员 */
                        pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) +
xWantedSize );
                        configASSERT( ( ( ( size_t ) pxNewBlockLink ) &
portBYTE_ALIGNMENT_MASK ) == 0 );


                        pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
xWantedSize;
                        pxBlock->xBlockSize = xWantedSize;


                        /* 将剩余的空闲块插入到空闲块列表中,按照空闲块的地址大小顺序,地址小
的在前,地址大的在后 */
                        prvInsertBlockIntoFreeList( pxNewBlockLink );
                    }

                    /* 计算未分配的内存堆空间,注意这里并不能包含内存碎片信息 */
                    xFreeBytesRemaining -= pxBlock->xBlockSize;

                    /* 保存未分配内存堆空间历史最小值 */
                    if( xFreeBytesRemaining < xMinimumEverFreeBytesRemaining )
                    {
                        xMinimumEverFreeBytesRemaining = xFreeBytesRemaining;
                    }


                    /* 将已经分配的内存块标识为"已分配" */
                    pxBlock->xBlockSize |= xBlockAllocatedBit;
                    pxBlock->pxNextFreeBlock = NULL;
                }
            }
        }


        traceMALLOC( pvReturn, xWantedSize );
    }
    ( void ) xTaskResumeAll();


    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {   /* 如果内存分配失败,调用钩子函数 */
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    #endif
```

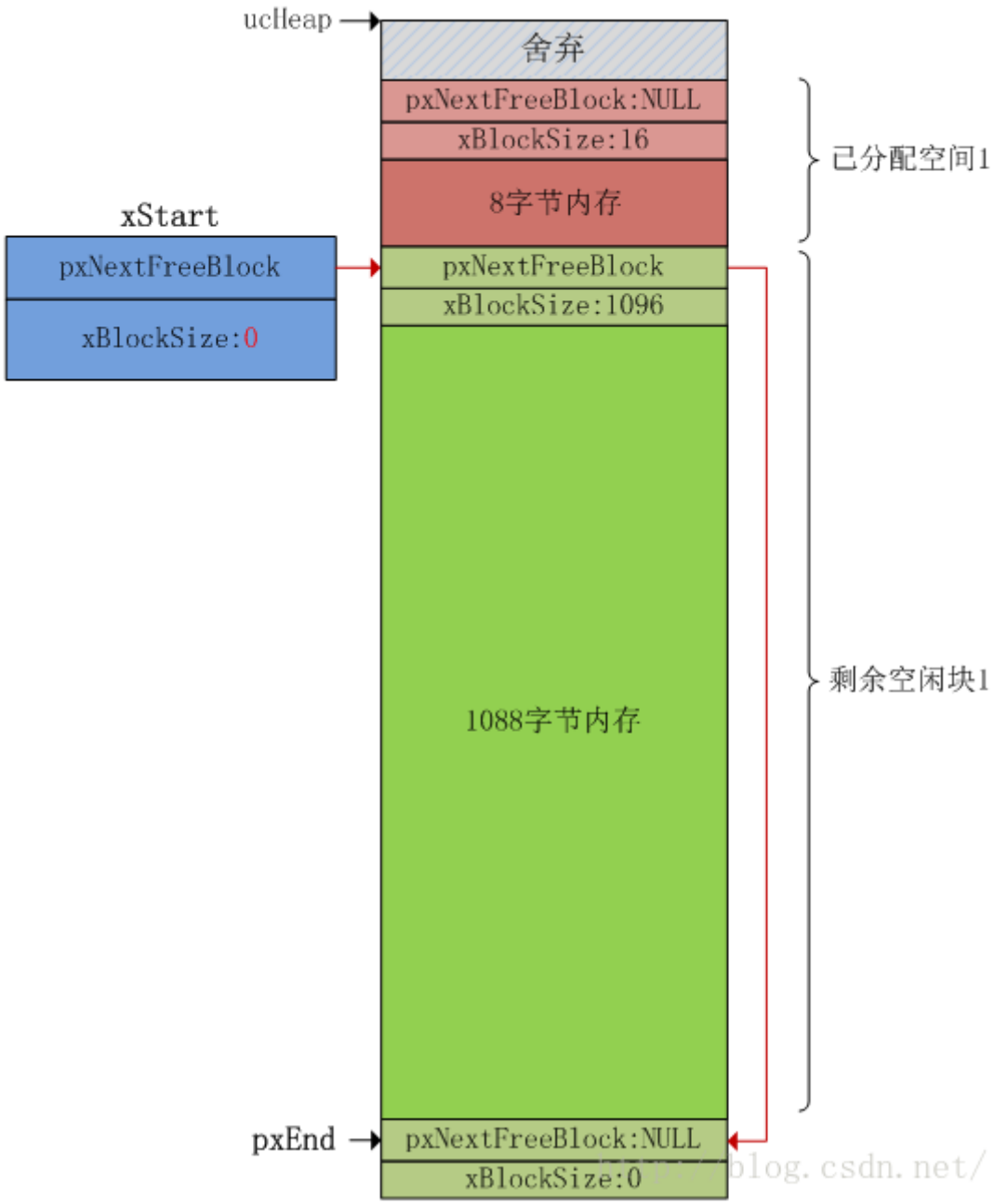
```
    configASSERT( ( ( ( size_t ) pvReturn ) & ( size_t ) portBYTE_ALIGNMENT_MASK
) == 0 );
    return pvReturn;
}
```

内存释放 : vPortFree()

heap_4内存管理策略的内存释放也比较简单：根据传入的参数找到链表结构，然后将这个内存块插入到空闲块列表，需要注意的是在插入过程中会执行合并算法。最后是将这个内存块标志为“空闲”、更新未分配的内存堆大小，结束。

```
void vPortFree( void *pv )
{
uint8_t *puc = ( uint8_t * ) pv;
BlockLink_t *pxLink;


    if( pv != NULL )
    {
        /* 根据参数地址找出内存块链表结构 */
        puc -= xHeapStructSize;
        pxLink = ( void * ) puc;


        /* 检查这个内存块确实被分配出去 */
        if( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )
        {
            if( pxLink->pxNextFreeBlock == NULL )
            {
                /* 将内存块标识为"空闲" */
                pxLink->xBlockSize &= ~xBlockAllocatedBit;


                vTaskSuspendAll();
                {
                    /* 更新未分配的内存堆大小 */
                    xFreeBytesRemaining += pxLink->xBlockSize;
                    traceFREE( pv, pxLink->xBlockSize );
                    /* 将这个内存块插入到空闲块链表中,按照内存块地址大小顺序 */
                    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
                }
                ( void ) xTaskResumeAll();
            }
        }
    }
}
```

如上图所示的内存堆示意图，如果我们将32字节的“已分配空间2”释放，由于这个内存块的上面和下面都是空闲块，所以在将它插入到空闲块链表的过程在中，会先和“剩余空闲块1”合并，合并后的块再和“剩余空闲块2”合并，这样组成一个大的空闲块，如下图所示



heap_5

heap_5内存管理策略允许内存堆跨越多个非连续的内存区，并且需要显示的初始化内存堆，除此之外其它操作都和heap_4内存管理策略十分相似。

假设我们为内存堆分配两个内存块，第一个内存块大小为0x10000字节，起始地址为0x80000000；第二个内存块大小为0xa0000字节,起始地址为0x90000000。HeapRegion_t结构体类型数组可以定义如下：

```
HeapRegion_t xHeapRegions[] =
 {
    { ( uint8_t * ) 0x80000000UL, 0x10000 },
    { ( uint8_t * ) 0x90000000UL, 0xa0000 },
    { NULL, 0 }
 };
```

两个内存块要按照地址顺序放入到数组中，地址小的在前，因此地址为0x80000000的内存块必须放数组的第一个位置。数组必须以使用一个NULL指针和0字节元素作为结束，以便让内存管理程序知道何时结束

定义好内存堆数组后，需要应用程序调用vPortDefineHeapRegions()函数初始化这些内存堆：将它们组成一个链表，以xStart链表结构开头，以pxEnd指针指向的位置结束。我们看一下内存堆数组是如何初始化的，以上面的内存堆数组为例，初始化后的内存堆如下图所示