原理

在本程序所有的空闲内存均以单链表的形式进行管理。图 1 即为链表节点的数据结构，约定 next_node 为空闲链表的下一节点，mem_size 为本节点管理的内存大小，该结构下面就是管理的内存空间，内存分布如图 2 所示。

```
typedef struct Mem_Node {
    struct Mem_Node* next_node;
    size_t mem_size;
}Mem_Node;
```

图 1

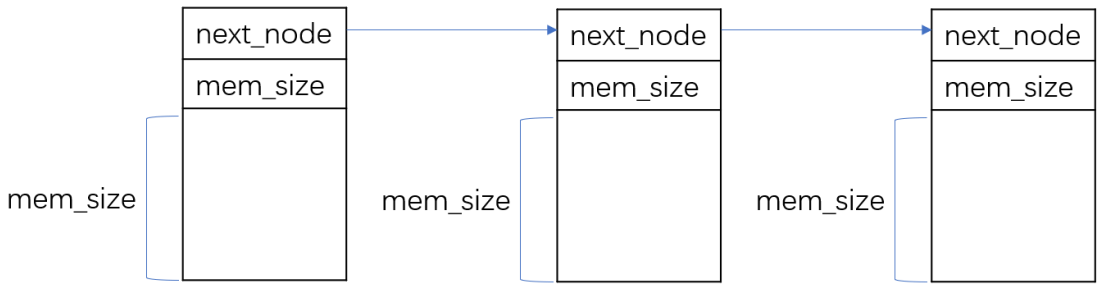

图 2

空闲链表的起点和终点均记录在句柄中。图 3 中 pStart、pEnd 即为链表的起点与终点，图 4 为各个节点内存对应关系。

```
typedef struct Mem_Root {
    Mem_Node* pStart;
    Mem_Node* pEnd;
    size_t total_size;      //总内存
    size_t remain_size;     //剩余内存
    Mem_Err_Type err_flag;  //错误标记
}Mem_Root;
```

图 3

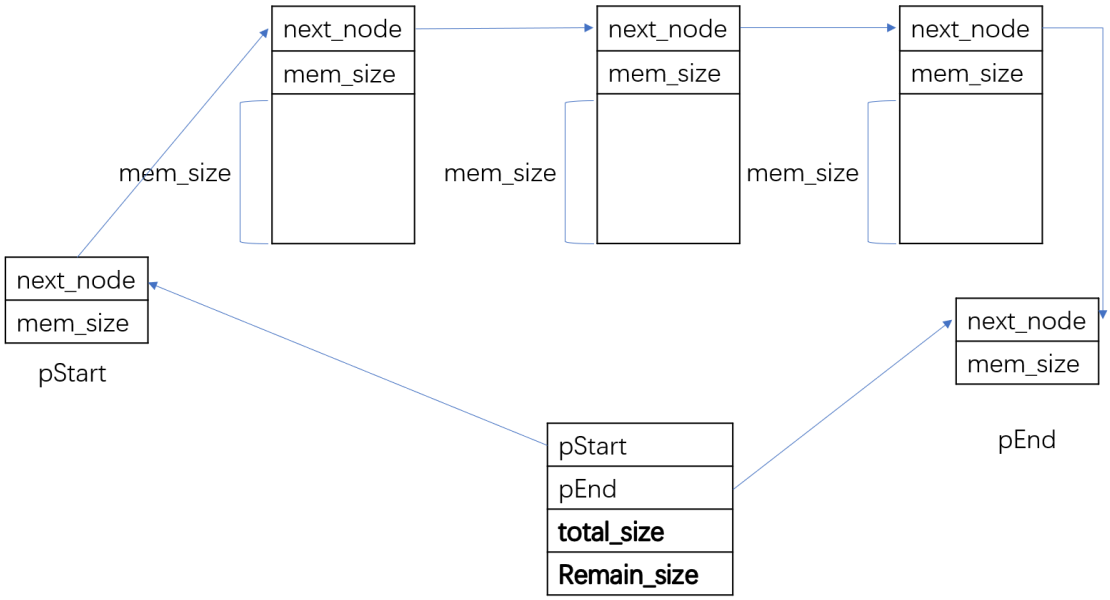

图 4

为了实现内存碎片合并的算法，首先要求空闲链表的是按照地址进行排序的，这样就可以内存在插入空闲链表时进行内存碎片的合并。图 5 为内存碎片合并算法。

```
//将内存节点插入空闲列表中
static __inline void Mem_Insert_Node_To_FreeList(Mem_Root* pRoot, Mem_Node* pNode) {
    Mem_Node* pPriv_Node;
    Mem_Node* pNext_Node;
    //寻找地址与pNode相近的节点
    for (pPriv_Node = pRoot->pStart; pPriv_Node->next_node < pNode; pPriv_Node = pPriv_Node->next_node);
    pNext_Node = pPriv_Node->next_node;
    pRoot->remain_size += pNode->mem_size;
    //尝试pNode与前一个块进行合并
    if ((uint8_t*)Mem_Manage_Mem_To_Addr(pPriv_Node) + pPriv_Node->mem_size == (uint8_t*)pNode) {
        if (pPriv_Node != pRoot->pStart) {//不是Start块的话可以合并
            pPriv_Node->mem_size += MEM_MANAGE_MEM_STRUCT_SIZE + pNode->mem_size;
            pRoot->remain_size += MEM_MANAGE_MEM_STRUCT_SIZE;
            pNode = pPriv_Node;
        }
        else {//后面如果是Start块不进行合并，以免浪费内存
            pRoot->pStart->next_node = pNode;
        }
    }
    else {//不能合并时直接插入到空闲单链表中
        pPriv_Node->next_node = pNode;
    }
    //尝试后面一个块与pNode进行合并
    if ((uint8_t*)Mem_Manage_Mem_To_Addr(pNode) + pNode->mem_size == (uint8_t*)pNext_Node) {
        if (pNext_Node != pRoot->pEnd) {//不是end块的话可以进行块合并
            pNode->mem_size += MEM_MANAGE_MEM_STRUCT_SIZE + pNext_Node->mem_size;
            pRoot->remain_size += MEM_MANAGE_MEM_STRUCT_SIZE;
            pNode->next_node = pNext_Node->next_node;
        }
        else {//后面如果是end块不进行合并，以免浪费内存
            pNode->next_node = pRoot->pEnd;
        }
    }
    else {//不能合并时直接插入到空闲单链表中
        pNode->next_node = pNext_Node;
    }
}
```

图 5

算法中首先在空闲链表中找到与插入节点相近的节点，寻找到地址相近的节点后分别尝试与前后节点进行合并。为了保持算法的正确性，需要确定遍历地址的上界和下界，这里为了降低算法的复杂度，在初始化阶段就将内存区的前部和尾部放置了 pStart 节点和 pEnd 节点，保证了后续插入的节点都是在这个范围之内。这两个节点仅仅是为了方便遍历所设置的标记，因此两个节点均不管理内存，内存合并的过程中也需要将其排除在外。

有了这样的一个空闲链表，内存分配也就变的简单了，分配内存的大致工作流程如下：

1、 遍历链表寻找合适大小的空闲内存块
2、 将内存块移出空闲链表
3、 将该内存块大小减去需要的内存大小，看剩下的内存能否构建新的内存块。
4、 若剩下的大小足够构建新的内存块，从内存块上分裂出不需要的内存构建新的内存块插入空闲链表中。
5、 返回内存块管理的内存区首地址。

这样就实现了一个简单的内存管理算法，算法核心原理大致就是这些，剩下的代码都是保证如何做好这项工作。

细节

1、如何实现非连续地址的内存管理？

在初始化的过程中，将不连续的地址内存看作以分配的内存，将所有的内存区按照链表的方式进行插入，即可实现非连续地址的内存管理

2、如何实现 C11 标准中的 aligned_alloc 函数？

aligned_alloc 要求实现任意对齐的地址分配，难点在于对齐消耗掉的内存如何管理，如果直接丢弃会造成内存泄漏。在算法中是将对齐消耗的内存进行扩大至一个链表节点的大小，然后将其插入空闲链表中。

3、如何保证 C 语言 malloc 函数的默认对齐参数？

C 语言的 malloc 函数都有一个默认的对齐参数，不同 CPU 对齐参数有所不同。之所以 malloc 函数有一个默认的对齐参数，是为了保证分配的内存可以正

常访问。拿 STM32 举例，如果使用了 malloc 函数为一个 double*类型的指针分配了内存，分配的内存是 4 字节对齐的内存而非 8 字节的内存，那么在高一些的优化编译选项上，对这个内存的访问是异常的，我自己以前就遇到过这种问题，这也是我编写该算法的一个原因。Malloc 函数的默认对齐参数应该为基本数据类型中最大数据类型的长度，在 STM32 中，基本数据类型最大为 8 字节，因此默认对齐参数就设置为了 8 字节对齐，这可以通过更改源文件内部的宏定义 MEM_MANAGE_ALIGNMENT_BYTE_DEFAULT 进行更改。为了保证在默认对齐参数的分配效率，算法内部会以默认对齐参数作为基础的内存单位，比如默认字节参数为 8 字节，那么算法管理的内存、分配的内存的大小等等，就都是以 8 字节为单位的，都是 8 的整数倍，这可以使得在该对齐参数下分配效率达到最高，默认对齐参数不宜过高，过高会有很多内存因地址对齐的原因被舍弃掉。