CubeEngine内存池实现与细节

Author：DBInary

## 引言

内存池作为一种的内存管理机制被广泛地运用于各种领域当中，内存池拥有快速的内存分配与更加健壮的管理机制，同时在不同的平台与环境当中也拥有不同的实现方式，本文提出一种轻量级的内存池实现，可以非常方便的移植到内存空间较小的平台当中，可运用在不同的嵌入式平台，服务端及小范围的内存管理当中.

## 从内存开始

巧妇难为无米之炊，既然我们要说是内存池，显然你首先至少需要找到一个能存储数据的东西，否者接下来都没有讨论的意义，存储的意义可以非常的广泛，它可以是名正言顺的RAM，也可以是你的硬盘，甚至是你的显存或者SD卡当中，为了给CubeEngine的内存池指定内存地址，你需要预先给出一块可用内存的大小与及它的起始地址.例如下面的代码将会给你做一个示范

//为内存池分配1M的内存空间，MemoryPool指向首地址

void \*MemoryPool=malloc(1024)

//MemoryPool指向地址0x08600000地址，这块地址可能指向一个片外RAM，一块NANDFlash地址或者其它的任意存储设备

void \*MemoryPool=(unsigned char\*)0x08600000;

//MemoryPool可存在于堆中也可存在于栈当中，但是你需要确保它的地址在内存池的使用阶段都是有效的。

char MemoryPool[1024];

可以看到，CubeEngine的内存池可以存在于任何的地址与区域当中，你需要确保的只有三点，1.内存地址是线性的，2.内存是可读的 3.内存是可写的.

## 一块内存

取得一块内存区域并不是什么非常困难的事情，在windows或者是linux当中，你可以使用现有的malloc或者C++的new来申请一块可用的内存地址，在嵌入式设备当中，你也可以使用片上RAM或者是类似于FSMC或外部地址总线等方式来指定一块内存，具体如何实现由您决定。

那么现在我们假设你已经拥有了这一块可用的内存并且做好了其它的一些别的准备工作，为了方便，我们假设您的内存地址是从0x00000000开始的，并且我们假设它的大小是

Size，为了说明方便我们使用图1来表示。

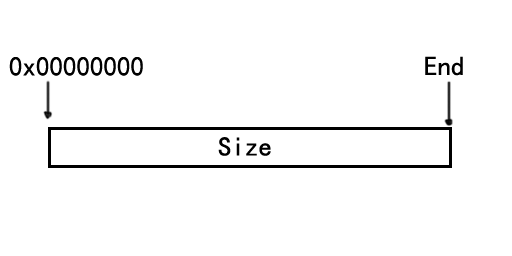


图1

从上面来看我们已经初始化一些基本的数据并且可以推导一些有用的数据，首先我们有一块内存，而且它的大小是Size，当中的所有区域我们都是可用的。我们还知道它的结束地址，是0x00000000+size-1;

那么既然是内存池，最基本的一个功能当然是可以从内存池当中分割出一块内存来供我们使用，这有点儿像分饼干，假如你读过一些内存池相关的文章，也许现在我们普遍的做法就是把这一块内存区域均等的分为多个block，比如假如这里的size是512字节，假如每个block占64字节，那么这个内存区域就会被分为8个block，当然还有更进一步，4个block组成一个chunk，那么512字节的内存区域，就包含有2个chunk或者说包含有8个block，当然，这里仅仅只是用512字节作为一个比方，也许你觉得这样分是在是多此一举，但假如这个size大到64M或者是128M，那么，用一个chunk为单位，对内存池进行寻址，显然会比你用字节为单位进行寻址要快得多，并且寻址范围也会大的多，并且以chunk（或者别的更大的单位）进行内存管理，对内存碎片的整理合并也会方便且快得多而且很容易使用map进行映射，这样就能够非常快的寻址处理，这种思想被广泛的运用在了文件系统与内存管理方面，例如windows操作系统就会建立一个页目录及页表来管理其内存空间，采用这种映射方式，它允许其内存在物理上不是连续的，甚至映射到硬盘或其他存储设备当中（虚拟内存）。

## 分割一块可用的内存

但遗憾的是，CubeEngine并不将内存进行这种分割处理，我们可以假如我们将一个Block分割为64字节，假设我需要一个86字节的内存空间，那么我们需要2个block也就是128字节，因为block作为一个整体是不可分的，那么我们将有42字节被白白的浪费了，在内存以G为单位的PC机上，也许这些损失实在是无足轻重，然而对于仅仅只有几百kb甚至是几kb的片上系统来说，这些损失就显得实在是太过于昂贵了。你也许会考虑到进一步缩小block提高细分度来减少这种损失，但是，即便是对block进行的寻址，也是需要内存空间的，并且更小的细粒度除了浪费内存之外，也显得毫无意义

技术不是扯淡，更不是拿起一本书就开始照本宣科，脱离实际地区域争论某种高深架构或者方法如何优秀，只是为了掩饰自己能力的贫庸.我们再来看看CubeEngine面向的环境.

1. 轻量级内存池，首先决定了它不能过于的臃肿与庞大，在保证功能的前提之下，能多简单就多简单，没必要把简单的问题复杂化
2. 内存分配较小。这就意味着使用映射表的方法意义实在不大，也许10000\*10000的复杂度会很大的优于100lg100，但是10lg10和10\*10的差别确实不怎么大，即使我们建立的内存表都用链表来进行管理，因为内存较小，表也不大，即使是遍历也未必会比映射慢上多少
3. 假如我们要记录一块分配内存，至少需要哪些信息呢，一.这块内存的起始地址

二．这块内存的结束地址或者它的大小。

那么，问题就开始变得简单了，假如内存池是一条等待切割的绳子，现在一个人想要3米长的绳子，那么非常简单，0-3米我们规定就是他的了，又一个人想要2米的绳子，那么我们再规定3m-5m这一段距离的绳子就属于它的了。为了记录下一次我们将从哪里开始切割绳子并且我们也需要知道剩下绳子的长度，以免我们只剩下1m的绳子了仍然答应一个想要2m长绳子的家伙，因此我们需要2个变量来进行记录1.下一次分割绳子的开始点，2.剩余绳子的长度。也就是下一次分配内存的起始地址和剩余内存的大小。那么，最基本的内存分配就这样实现了

我们先定义内存池的基本结构

typedef struct \_MemoryPool

{

void \*AllocAddr;//下一次内存的分配地址

void \*EndAddr;//内存池的尾地址

unsigned int Size;//内存池的总大小

unsigned int FreeSize;//剩余内存池空间大小

}MemoryPool;

我们定义一个函数,用来初始化这个内存池

void CreateMemoryPool(void \*MemoryAddr,unsigned int MemorySize);

它的实现是

void CreateMemoryPool( void \*MemoryAddr,unsigned int MemorySize )

{

G\_MemoryPool.AllocAddr=MemoryAddr;//最开始的分配地址就是起始地址

G\_MemoryPool.EndAddr=((unsigned char\*)MemoryAddr)+MemorySize-1;//计算出内存池终止地址

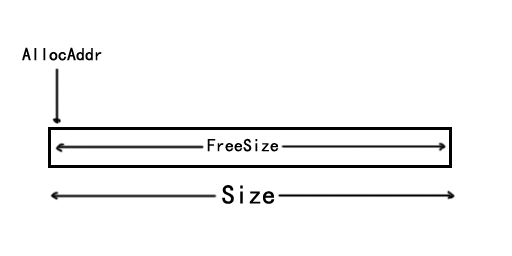
G\_MemoryPool.Size=MemorySize;//内存池大小

G\_MemoryPool.FreeSize=MemorySize;//一开始剩余内存池空间大小就是内存池大小

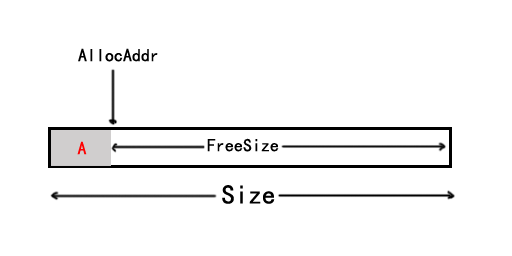
}

下面几张图表示内存的分配情况

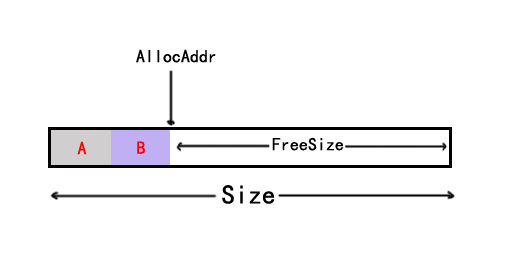
1. 内存初始化时



1. 分配内存A后



1. 分配内存B后



## 有借有还:内存碎片与回收

#### 记录分配节点

计算机是一个伟大的发明，毫无意味的在某些数学角度方面，它的计算速度要比人快得多，但是遗憾的是，计算机未必比人聪明，而且是一个奇怪的家伙，他记忆力很很差劲，只能靠一些存储器件来帮助他进行记忆，比如寄存器和硬盘。

上面的方案看上去似乎工作的非常好，而且也不会出上面问题，当然，假如你的内存是无限大的话，你完全可以这么干，然而在某些平台上，你却不得不抠门地去节省那几个字节甚至是几个位，上面的方案显然碰到了一些问题，我们分配了AB两个内存块，也许A当中只是存储了一些临时的数据，或者说这些数据我们用不到了，我们也许会希望释放掉这个内存，并且把空闲的内存归还到内存池。

然而上面的方案我们只是把内存简简单单的划分出去了，而没有做任何记录，显然的，我们应该给它做个记录，为了节省内存，我们尽量将记录变得简略，这并不复杂，我们只要记录分配的起始地址和结束地址就可以了.就像出租房房东一样，在账本下记下几号房到几号房出租给了某某某，当那人来退房的时候，那几个房间就可以回收利用了，在这里，我们将这个记录称为内存节点，用c语言表示如下：

typedef struct \_MemoryNode

{

void \*StartAddr;

void \*EndAddr;

}MemoryNode;

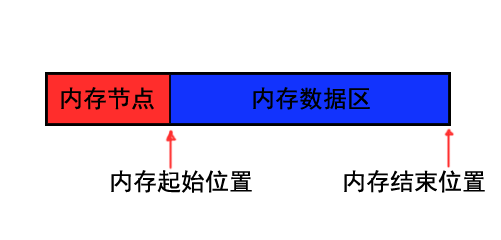
#### 如何放置内存节点

上面的思路也许看上去非常简单，不需要太多的思考也能想出来，但是如何放置内存节点和如何回收回来的内存节点确值得思考，在网上流行的很多内存池中，基本都采用了链表的方式去连接和管理节点，假如新增加一个节点，就使用malloc分配一个，然而在一个要求高度可移植的内存池方面，这个想法却是不完善的，首先，你不知道堆到底有多少空间供你分配，因为非常多的小型系统的堆大小仅仅只有可怜的几kb，而且用户很可能已经将那些空间使用的所剩无几了，所以这个办法显然是被否决的，另一种办法是采用类似于windows的内存管理机制，使用页表，但之前也已经提及了，这种办法是在是太奢侈了，另外，放置节点的位置也需要讲究，我们看看释放内存的函数

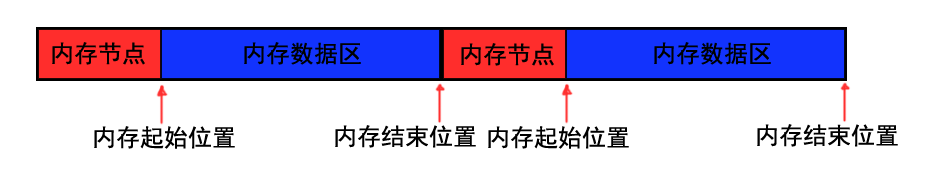
Free(void \*address)

Address是需要释放内存的首地址，这也意味着，你需要依靠首地址去寻找这个节点，假如你将内存节点都存储在一个区域当中，你很可能要去遍历来寻找这个节点，倘若你调用分配函数1000次，每次free都有可能遍历节点数百次，这显然对于cpu而言，实在是太过的奢侈繁琐了。

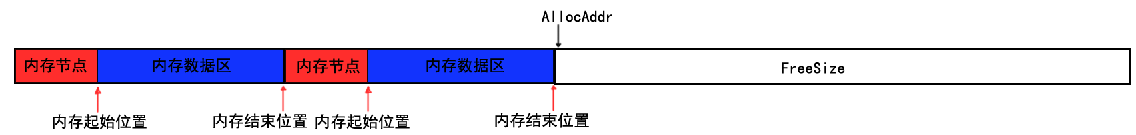
因此最合适的位置，应该是分配到address福建，将节点放在开始地址的前面，应该是一个不错的方案。



使用sizeof(MemoryNode)我们很容易确定内存节点的大小，所以在free的时候，我们仅仅需要将address减去sizeof(MemoryNode)，我们就可以非常容易获得这个内存节点，并且根据这个节点来取得内存数据区的起始位置和结束位置。

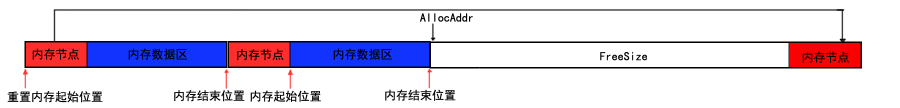


### 释放的内存节点与管理

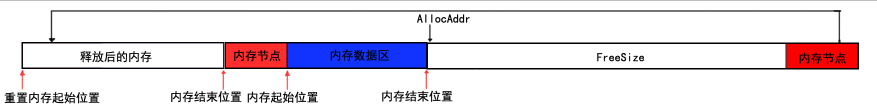
现在，我们可以将那些释放掉的内存节点进行管理了，我们仍然需要将释放的内存节点存储在内存池当中，但我们将不采用链表的形式存储，相对于链表，线性存储显然颇具优势。而我们需要做的，仅仅只是为其挑一个存储的位置与空间，不言而喻的，将这些释放的节点存储在内存池的尾部，是一个不错的选择，因为受到池内存大小的限制，释放的内存节点经过整理后也不会显得过于的庞大，因此，即使是遍历搜索也不会慢到哪里去，释放内存节点非常简单，我们仅仅只需要将节点的起始地址设置为内存节点的开始地址，然后复制节点节点到尾部就可以了，就像下图所表示的那样：

释放第一块内存：

1.拷贝节点到内存池尾部



2.实际上无需其它的操作，这块内存意义上已经释放了



我们用下面的代码，简单为内存池增加一个释放的内存节点：

MemoryNode \*PX\_AllocFreeMemoryNode()

{

//增加释放的节点数目

G\_MemoryPool.FreeTableCount++;

//以MemoryNode指针的形式返回该内存节点的首地址

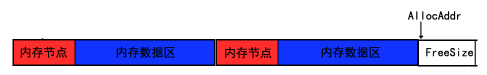
return PX\_MemoryPool\_GetFreeTable(G\_MemoryPool.FreeTableCount-1);

}

### 不能失败的内存释放

但是上面的办法真的没有问题吗，我们看看Free的原型

Void Free(void \*Address)它的返回值是null也就是必须成功，那么什么时候上面的方案会失败呢，非常明显的，每当释放一个内存节点，我们将从尾部去记录释放的内存节点，尾部的空间实际上是输入FreeSize的，当FreeSize小于一个内存节点的大小的时候，分配就会失败



（FreeSize已经不足以保存一个节点，释放失败）

释放失败的后宫是严重的，它将导致那块内存被永久的占用，比如你内存池大小是1M，已经分配了1023字节，那么，这个内存池除非你恰好释放的是最后一个节点（稍后讨论），否者这个内存池将直接陷入瘫痪，但是办法总是有的，其中最有效的是，我们在内存分配期间，就需要为内存的释放预留出空间，这个虽然稍微有点儿浪费空间，但是这对内存池的健康与维护大有好处，并且MemoryNode结构本身并不大，释放空间操作相比于分配操作往往要少的多，绝大部分的空间都是随着内存分配的开始到程序运行的结束。



那么这也就意味着，在我们分配内存的时候，至少需要分配两个MemoryNode节点大小的内存空间才能够满足要求，同时我们也需要记录下当前有多少个已经释放的内存节点与在释放的节点中最大的节点大小是多少,这样做方便后期的查找与遍历与在释放的内存节点中分配新的内存，我们修改MemoryPool结构体：

typedef struct \_MemoryPool

{

void \*AllocAddr; //FreeSize内存区域起始地址

void \*EndAddr; //内存池终止地址

unsigned int Size;//内存池大小

unsigned int FreeSize;//FreeSize生于大小

unsigned int FreeTableCount;//释放节点的数目

unsigned int MaxMemoryfragSize;//释放节点中最大的碎片大小

}MemoryPool;

同时,从FreeSize中分配内存的实现的代码就如下所示

MemoryNode \*Node;

//检查剩余空间是否足够分配,大小是2倍的memorynode大小加上分配大小

if (Size+2\*sizeof(MemoryNode)>G\_MemoryPool.FreeSize)

{

return 0;

}

//初始化MemoryNode

Node=(MemoryNode \*)G\_MemoryPool.AllocAddr;

//指定数据的开始地址

(\*Node).StartAddr=(unsigned char )G\_MemoryPool.AllocAddr+sizeof(MemoryNode);

//指定数据结束地址

(\*Node).EndAddr=(unsigned char \*)Node->StartAddr+Size-1;

//FreeSize的大小减去分配的空间大小

G\_MemoryPool.FreeSize-=(Size+2\*sizeof(MemoryNode));

//将AllocAddr移到新的位置

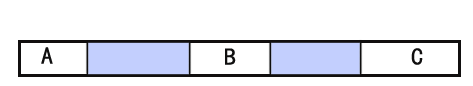
G\_MemoryPool.AllocAddr=(char\*)G\_MemoryPool.AllocAddr+Size+sizeof(MemoryNode);

//返回内存节点指针

return Node;

## 内存的碎片与减少碎片

也许最合理的内存回收是能够回收多少是多少，回收完后仍然是线性的，例如现在我释放 了一个大小为A（8字节） B(12字节) C（4字节）的内存，下次我需要分配24字节，可以直接从之前释放的空间中申请，但是，实际情况上却常常难以做到，因为我们申请的内存，必须是线性连续分配的,但释放后的内存空间,往往不是连续的.



（非连续内存导致碎片）

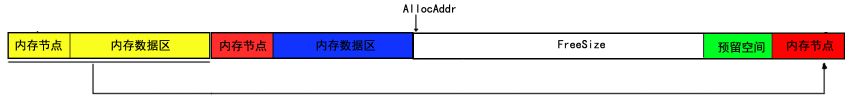
为了最大程度地减少碎片,我们需要将那些原本连续的碎片进行拼接合并,例如内存节点A的结束地址和内存节点B的开始地址恰好是重合的,那么我们就可以考虑将节点A与节点B进行合并,合并内存不论是在效率上还是内存的再分配上都有好处.

我们对以下几种情况的内存碎片分别讨论

1. 不能合并的内存节点



A是一个释放的内存节点,很遗憾的是,不论是A的前方或者后方都是一家被内存池分配出去的内存，A将不能进行合并，对于这种情况，调用之前写好的PX\_AllocFreeMemoryNode函数分配一个节点，并将A的内存节点信息直接保存在内存池的尾部



1. 可以向前合并的节点

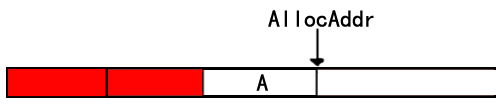


A是一个待释放的内存节点，B是一个已释放的的内存节点，可以看到，BA在内存空间上是相连的，所以我们需要将BA两个内存空间进行合并，在这个过程当中，我们不需要再分配一个释放的内存节点，原先为A释放节点的预留空间也可以删除回归到Freesize当中，之后仅需要修改原来就存在的B的释放内存节点，将他的结束地址修改为节点A的结束地址，就可以完成释放了。

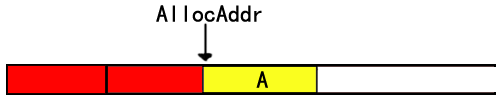
1. 向后合并的节点



A是一个待释放的内存节点，B是一个已释放的的内存节点， AB在内存空间上是相连的，所以我们需要将AB两个内存空间进行合并，其过程与向前合并时类似的。但是向后合并我们还可以更进一步地进行优化，来看看下面这种情况



在节点A的后面，就是AllocAddr，在这种情况下，我们可以尽量地扩大FreeSize而不是将A的节点放入释放内存节点中，我们直接将AllocAddr放到A的首地址，并且删除原先为A的预留空间，这样，A就被释放掉了



1. 前后合并节点



前后合并的其实就是向前合并和向后合并的组合，但我们需要的是注意一下合并的顺序，因为向后合并存在一种特殊的情况，我们可以将其选择出来并进行专门的处理，就如下图所示的情况



第一种情况的前后合并时，需要删除一个预留空间（看顺序）并且删除释放节点，最后只留下一个节点，第二种情况首先向前合并，同时删除B的释放内存节点和A的预留空间，将AllocAddr移动到B的起始地址完成内存释放.使用下面的代码实现删除释放内存节点

\_\_inline MemoryNode \*PX\_MemoryPool\_GetFreeTable(unsigned int Index)

{

//index不可能大于等于节点数目，下断言

assert(Index<G\_MemoryPool.FreeTableCount);

//索引加1

Index++;

//向前偏移返回memoryNode的地址

return (MemoryNode \*)((unsigned char \*)G\_MemoryPool.EndAddr-(sizeof(MemoryNode)\*Index)+1);

}

void PX\_MemoryPoolRemoveFreeNode(unsigned int Index)

{

unsigned int i;

MemoryNode \*pFreeNode=PX\_MemoryPool\_GetFreeTable(Index);

//将之前的节点前移实现移除这个节点

for (i=Index;i<G\_MemoryPool.FreeTableCount;i++)

{

(\*pFreeNode)=\*(pFreeNode-1);

pFreeNode--;

}

//释放内存节点数减1

G\_MemoryPool.FreeTableCount--;

}

最后，依照上述的几种情况编写内存释放函数

void PX\_Free( void \*pAddress )

{

unsigned int i,sIndex;

MemoryNode \*itNode;//迭代指针

MemoryNode FreeNode;//释放内存节点

unsigned char \*pcTempStart,\*pcTempEnd;

unsigned char bExist;

bExist=0;

//依据地址得到内存信息节点

FreeNode=\*(MemoryNode \*)((unsigned char \*)pAddress-sizeof(MemoryNode));

//重置内存节点的开始位置

FreeNode.StartAddr=(unsigned char \*)FreeNode.StartAddr-sizeof(MemoryNode);

//假如这个节点是最后一个节点,也就是结束地址后一个就是AllocAddr

if ((char \*)FreeNode.EndAddr+1==(char \*)G\_MemoryPool.AllocAddr)

{

//向前搜索是否有可以拼接的节点

for(i=0;i<G\_MemoryPool.FreeTableCount;i++)

{

itNode=PX\_MemoryPool\_GetFreeTable(i);

if ((char \*)itNode->EndAddr+1==(char \*)FreeNode.StartAddr)

{

//如果有,将AllocAddr的地址移到那个节点的开始地址

G\_MemoryPool.AllocAddr=itNode->StartAddr;

//恢复释放节点的空间到FreeSize同时删除预留空间的地址

G\_MemoryPool.FreeSize+=((char\*)FreeNode.EndAddr-(char \*)FreeNode.StartAddr+sizeof(MemoryNode)+1);

//恢复合并的空间到FreeSize

G\_MemoryPool.FreeSize+=(char\*)itNode->EndAddr-(char \*)itNode->StartAddr+1+sizeof(MemoryNode);

//移除这个节点

PX\_MemoryPoolRemoveFreeNode(i);

//处理完成,返回

return;

}

}

//如果没有需要拼接的节点,将AllocAddr直接连接到起始地址

G\_MemoryPool.AllocAddr=(char \*)FreeNode.StartAddr;

//恢复占用的空间和预留空间

G\_MemoryPool.FreeSize+=((char\*)FreeNode.EndAddr-(char \*)FreeNode.StartAddr+sizeof(MemoryNode)+1);

return;

}

//不是特殊情况

//记录前一次连接索引的节点,若为0xfffffff表示没有,如果有下一次连接节点且上一个节点存在,则删除前一次的节点

sIndex=0xffffffff;

//查找是否有可连接的节点

for(i=0;i<G\_MemoryPool.FreeTableCount;i++)

{

itNode=PX\_MemoryPool\_GetFreeTable(i);

pcTempStart=(unsigned char \*)itNode->StartAddr;

pcTempEnd=(unsigned char \*)itNode->EndAddr;

//查找是否有可以向后连接的节点

if (pcTempStart==(unsigned char \*)FreeNode.EndAddr+1)

{

//如果有,按照之前所述连接节点

if(sIndex==0xffffffff)

{

sIndex=i;

//Refresh this node

itNode->StartAddr=FreeNode.StartAddr;

FreeNode=\*itNode;

G\_MemoryPool.FreeSize+=sizeof(MemoryNode); }

else

{

itNode->StartAddr=FreeNode.StartAddr;

G\_MemoryPool.FreeSize+=sizeof(MemoryNode);

PX\_MemoryPoolRemoveFreeNode(sIndex);

}

bExist=1;

}

//查找是否有可以向前连接的节点

if (pcTempEnd+1==(unsigned char \*)FreeNode.StartAddr)

{

//如果有,按照之前所述连接节点

if(sIndex==0xffffffff)

{

sIndex=i;

//Refresh this node

itNode->EndAddr=FreeNode.EndAddr;

FreeNode=\*itNode;

//移除上一个节点

G\_MemoryPool.FreeSize+=sizeof(MemoryNode);

}

else

{

itNode->EndAddr=FreeNode.EndAddr;

G\_MemoryPool.FreeSize+=sizeof(MemoryNode);

//移除上一个节点

PX\_MemoryPoolRemoveFreeNode(sIndex);

}

bExist=1;

}

}

//如果没有连接节点,分配释放内存节点

if(bExist==0)

\*PX\_AllocFreeMemoryNode()=FreeNode;

//这个函数用于更新最大的释放内存节点大小

PX\_UpdateMaxFreqSize();

}

# 分配内存

### 从空闲空间中分配内存

在上一个章节的” 分割一块可用的内存”,”不能失败的内存释放”中,给出了这个空闲空间分配内存的方法与分配代码,在这里就不再复述了,它的流程非常简单

1. 分割出一块合适的空间大小
2. 初始化内存节点
3. 预留空间
4. 返回起始地址

短短几步,就能够完成空闲空间的内存分配,当内存空间不够时,返回0

## 从已释放内存中分配内存

回收内存再分配,是一个内存池能否能够持续良好持续运行的一个必要功能,同时,这也常常成为碎片产生的原因.但是内存池是否能够完全避免产生碎片呢,当然这并不是不可能的,但是付出的代价往往不值得我们这样去做.所以,内存碎片,尽可能的减少却不能够完全避免.

那么应该如何减少碎片呢,其实这和环境与用法有很大的不同,一个老练的码农往往喜欢把内存分配到2的整数幂,这也对减少碎片有很大的帮助,比如你分配了一个64字节大小的内存释放之后又分配一个32字节和4个8字节的内存,刚好就填充了之前的空间,越小的内存分配越容易填充之前的碎片,当然,假如你申请的内存和释放的内存一直保持一致的大小,这个碎片也就能够最大程度的消除了.

过多的碎片不仅占用了不必要的内存空间,也拖慢了再分配和释放,因此,内存分配并不仅仅只是算法上的,很多时候也是使用和习惯上的.

那么如果需要从已分配内存当中新分配内存应该怎么办呢,我们先来讨论一下不产生碎片的情况,最直观的是,你释放了一个内存的大小,刚好就是你想再分配内存的大小,这是无缝的,自然不会产生碎片.但是我们不能就仅仅止步于此,我们再来看看内存的结构,

是的,在CubeEngine内存池结构中,分配一个size大小的内存,需要size+2\*sizeof(MemoryNode)大小的内存空间,分配内存分配的大小,是我们主观上的大小,我们为其多分配一些内存空间,并不会出什么大不了的问题,比如你申请 16字节大小,而我分配了32字节给你,多余的字节并不会对其它功能造成影响,顶多我不知道不用就是了,而且这样比我们分割这个已经释放的内存空间更有好处,毕竟,为新划分的空间需要预留额外的sizeof(MemoryNode)大小的空间而且也会照成碎片,那么,我们可以这样规定

假如已经释放的内存大小大于等于size并且小于sizeof(MemoryNode)+size,我们就将这个已经释放的内存全部分配给需求的内存.

当然,我们不可能期待每个已经释放内存都像上面一样,很多时候,我们释放了大小为128字节的空间,下一次却只需要32字节的内存空间,把这个128字节的释放空间都分配给这个只需要32字节的内存节点,显然是太过的浪费了,这个时候,我们必须要对内存进行分割

分割并不复杂,只需要把原先的释放内存首地址作为分配内存的首地址,填充MemoryNode节点,然后将结束地址+1替换到原来的释放内存节点的首地址就可以了,同时别忘了预留一个内存节点的空间.

MemoryNode \*PX\_AllocFromFreq(unsigned int Size)

{

unsigned int i,fSize;

MemoryNode \*itNode,\*allocNode;

//大小加上头部内存节点需要空间

Size+=sizeof(MemoryNode);

//比对最大已释放内存大小,若小于,分配肯定是失败的

if (G\_MemoryPool.MaxMemoryfragSize>=Size)

{

//开始搜索满足条件的节点

for(i=0;i<G\_MemoryPool.FreeTableCount;i++)

{

itNode=PX\_MemoryPool\_GetFreeTable(i);

//计算节点大小

fSize=(char \*)itNode->EndAddr-(char \*)itNode->StartAddr+1;

//满足内存节点完全分配的情况

if (Size<=fSize&&(Size+sizeof(MemoryNode)>fSize))

{

//初始化新的内存节点

allocNode=(MemoryNode \*)itNode->StartAddr;

allocNode->StartAddr=(char \*)itNode->StartAddr+sizeof(MemoryNode);

allocNode->EndAddr=itNode->EndAddr;

//因为是完全分配,直接删除原来的节点,因为空间已经预留过了,所以不需要预留空间

PX\_MemoryPoolRemoveFreeNode(i);

//重新跟新最大释放内存节点大小

PX\_UpdateMaxFreqSize();

return allocNode;

}

else

{

//不满足完全分配的情况,开始进行内存分割

if(Size<fSize)

{

//确认FreeSize足够分配一个内存节点

if (G\_MemoryPool.FreeSize<sizeof(MemoryNode))

{

return 0;

}

//初始化内存节点

allocNode=(MemoryNode \*)itNode->StartAddr;

allocNode->StartAddr=(char\*)itNode->StartAddr+sizeof(MemoryNode);

allocNode->EndAddr=(char \*)itNode->StartAddr+Size-1;

//重新计算释放内存节点的起始位置,完成分割

itNode->StartAddr=(char \*)allocNode->EndAddr+1;

//预留空间

G\_MemoryPool.FreeSize-=sizeof(MemoryNode);

//再次重新计算最大释放内存节点大小

PX\_UpdateMaxFreqSize();

return allocNode;

}

}

}

return 0;

}

else

{

return 0;

}

}