

# 【性能优化】高效内存池的设计与实现

“

对本文有疑问的，可以公众号留言、私信，也可以加笔者微信直接交流(文末留有微信二维码)；另外还有批量免费计算机电子书，后台回复[pdf]免费获取。

”

大家好，我是雨乐！

在之前的文章中，我们分析了glibc内存管理相关的内容，里面的是不是逻辑复杂😓，毕竟咱们用几十行代码完成的功能，glibc要用上百乃至上千行代码来实现，毕竟它的受众太多了，需要考虑跨平台，各种边界条件等。

其实，glibc的内存分配库ptmalloc也可以看做是一个内存池，出于性能考虑，每次内存申请都是先从ptmalloc中进行分配，如果没有合适的则通过系统分配函数进行申请；在释放的时候，也是将被释放内存先方式内存池中，内存池根据一定的策略，来决定是否进行shrink以归还OS。

那么，现一个内存池？我们该怎么实现呢？今天，借助这篇文章，我们一起来设计和实现一个内存池(文末附有github地址)。

## 背景

首先需要说明的是，该内存池是笔者在10年前完成的，下面先说下当时此项目的背景。

09年，在某所的时候，参与了某个国家级项目，该项目是防DDOS攻击相关，因此更多的是跟IP相关，所以每次分配和释放内存都是固定大小，经过测试，性能不是很满意，所以，经过代码分析以及性能攻击分析，发现里面有大量的malloc/free，所以，当时就决定是否从malloc/free入手，能否优化整个项目的性能。

所以，决定实现一个Memory Pool，在做了调研以及研究了相关论文后，决定实现一个内存池，先试试水，所幸运的是，性能确实比glibc自带的malloc/free要高，所以也就应用于项目上了。

“

本文所讲的Memory Pool为C语言实现，旨在让大家都能看懂，看明白(至少能够完全理解本文所讲的Memory Pool的实现原理)。

”

## 概念

首先，我们介绍下什么是内存池？

“

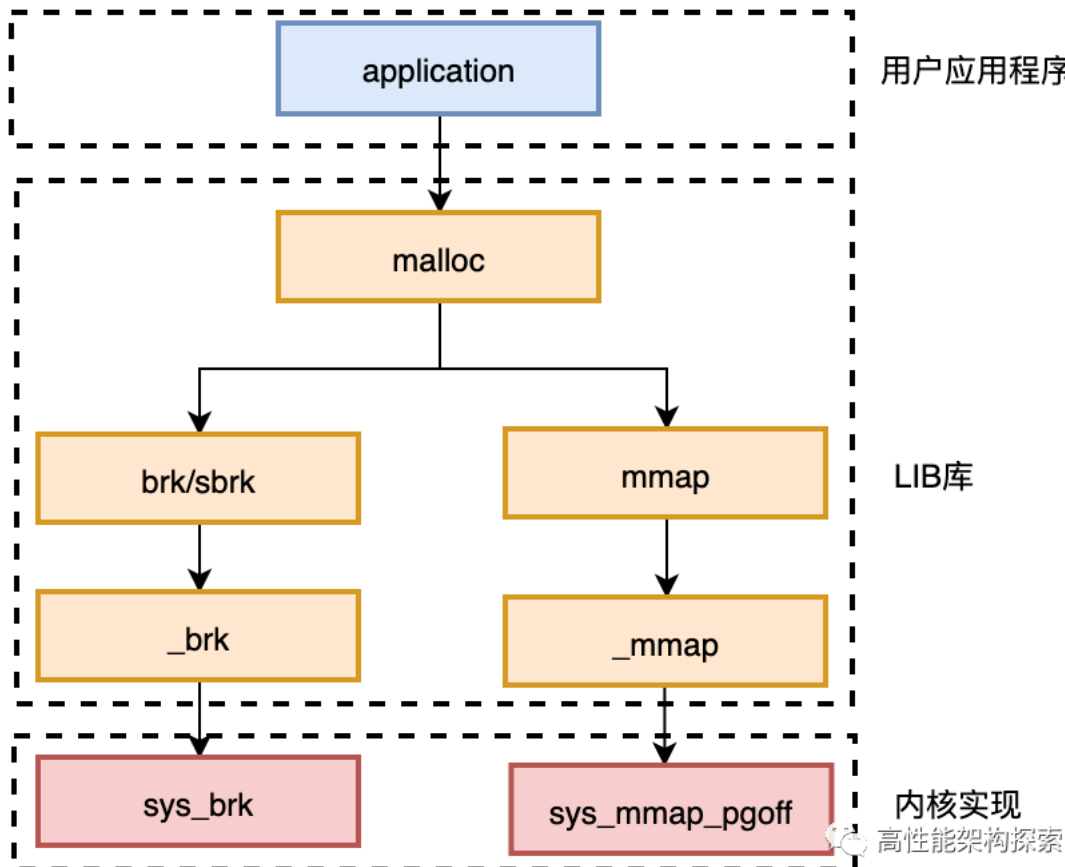
预先在内存中申请一定数量的内存块留作备用，当有新的内存需求时，就先从内存池中分配内存返回，在释放的时候，将内存返回给内存池而不是OS，在下次申请的时候，重新进行分配

”

那么为什么要有内存池呢？这就需要从传统内存分配的特点来进行分析，传统内存分配释放的优点无非就是 通用性强，应用广泛，但是传统的内存分配、释放在某些特定的项目中，其不一定是最优、效率最高的方案。

传统内存分配、释放的缺点总结如下：

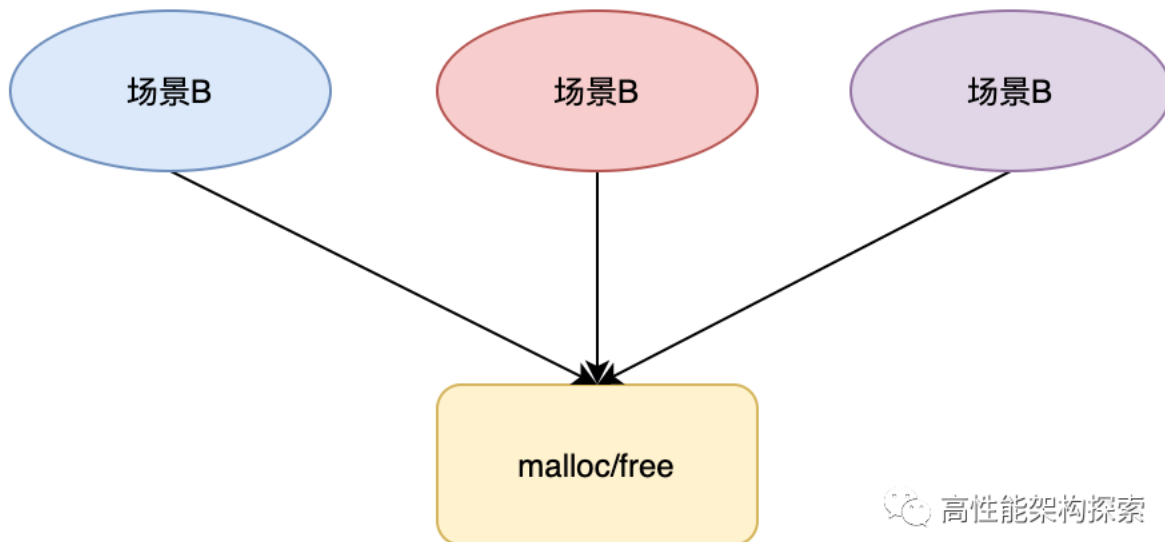
- 1、调用malloc/new,系统需要根据“最先匹配”、“最优匹配”或其他算法在内存空闲块表中查找一块空闲内存，调用free/delete,系统可能需要合并空闲内存块，这些会产生额外开销
- 2、频繁的在堆上申请和释放内存必然需要大量时间，降低了程序的运行效率。对于一个需要频繁申请和释放内存的程序来说，频繁调用new/malloc申请内存，delete/free释放内存都需要花费系统时间，频繁的调用必然会降低程序的运行效率。
- 3、经常申请小块内存，会将物理内存“切”得很碎，导致内存碎片。申请内存的顺序并不是释放内存的顺序，因此频繁申请小块内存必然会导致内存碎片，造成“有内存但是申请不到大块内存”的现象。



内存分配

从上图中，可以看出，应用程序会调用glibc运行时库的malloc函数进行内存申请，而malloc函数则会根据具体申请的内存块大小，根据实际情况最终从sys\_brk或者sys\_mmap\_pgoff系统调用申请内存，而大家都知道，跟os打交道，\_性能损失\_是毋庸置疑的。

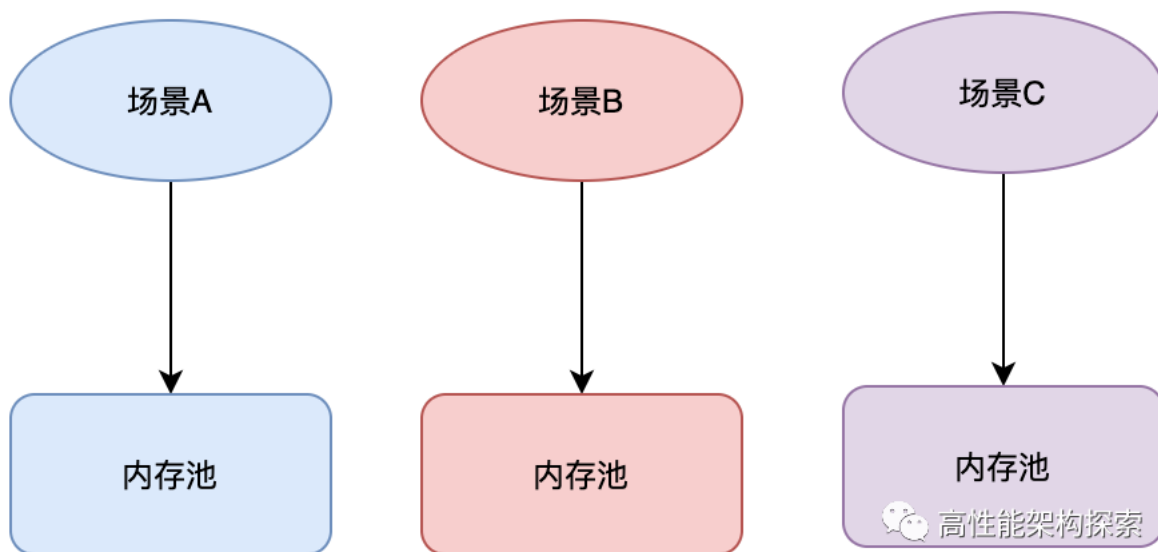
其次，glibc作为通用的运行时库，malloc/free需要满足各种场景需求，比如申请的字节大小不一，多线程访问等。



没有比传统malloc/free性能更优的方案呢？

答案是：有。

在程序启动的时候，我们预分配特定数量的固定大小的块，这样每次申请的时候，就从预分配的块中获取，释放的时候，将其放入预分配块中以备下次复用，这就是所谓的\_内存池技术\_，每个内存池对应特定场景，这样的话，较传统的传统的malloc/free少了很多复杂逻辑，性能显然会提升不少。



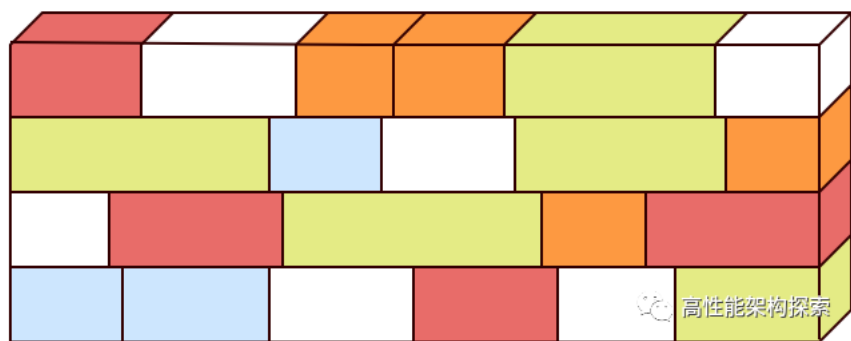
结合传统malloc/free的缺点，我们总结下使用内存池方案的优点：

- 1、比malloc/free进行内存申请/释放的方式快
- 2、不会产生或很少产生堆碎片
- 3、可避免内存泄漏

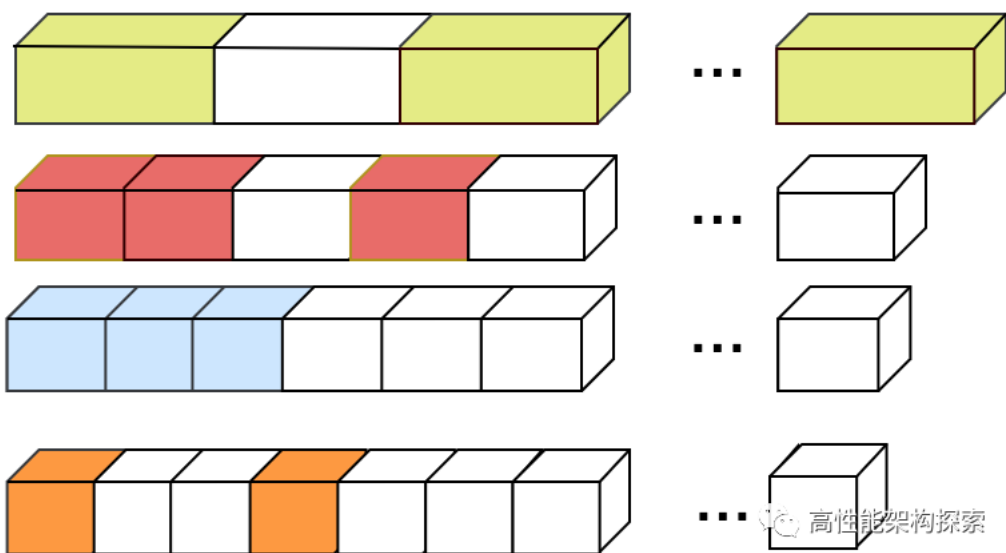
## 分类

根据分配出去的字节大小是否固定，分为 固定大小内存池 和 可变大小内存池 两类。

而可变大小内存池，可分配任意大小的内存池，比如ptmalloc、jemalloc以及google的tcmalloc。



固定大小内存池，顾名思义，每次申请和释放的内存大小都是固定的。每次分配出去的内存块大小都是程序预先定义的值，而在释放内存块时候，则简单的挂回内存池链表即可。



“

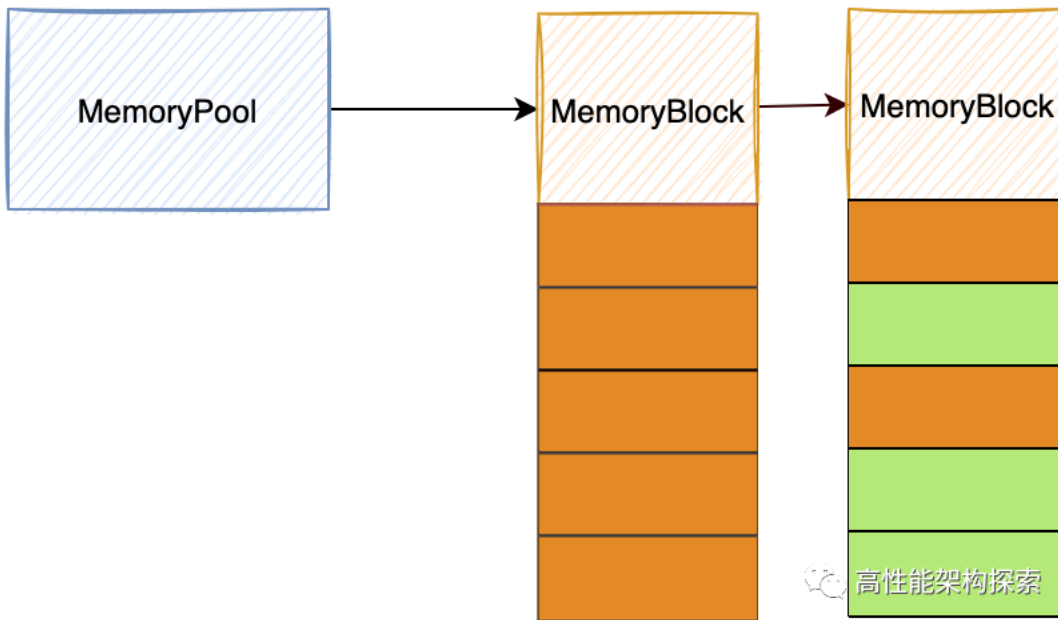
本文主要讲的是固定大小的内存池。

”

## 原理

内存池，重点在“池”字上，之所以称之为内存池，是在真正使用之前，先预分配一定数量、大小预设的块，如果有新的内存需求时候，就从内存池中根据申请的内存大小，分配一个内存块，若当前内存块已经被完全分配出去，则继续申请一大块，然后进行分配。

当进行内存块释放的时候，则将其归还内存池，后面如果再有申请的话，则将其重新分配出去。



内存池结构图

上图是本文所要设计的结构图，下面在具体的设计之前，我们先讲下本内存池的原理：

- 创建并初始化头结点MemoryPool
- 通过MemoryPool进行内存分配，如果发现MemoryPool所指向的第一块MemoryBlock或者现有MemoryPool没有空闲内存块，则创建一个新的MemoryBlock初始化之后将其插入MemoryPool的头
- 在内存分配的时候，遍历MemoryPool中的单链表MemoryBlock，根据地址判断所要释放的内存属于哪个MemoryBlock，然后根据偏移设置MemoryBlock的第一块空闲块索引，同时将空闲块个数+1

上述只是一个简单的逻辑讲解，比较宏观，下面我们将通过图解和代码的方式来进行讲解。

## 设计

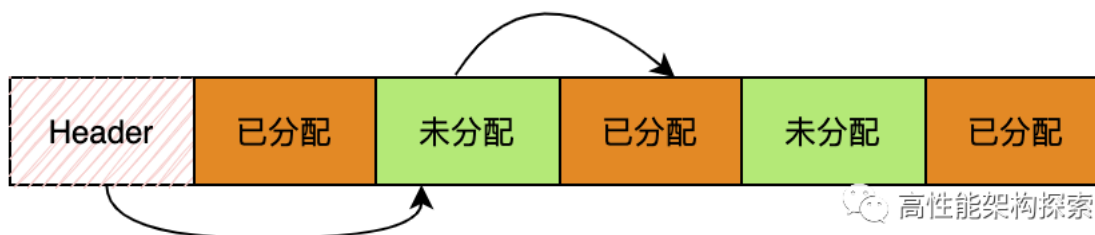
在上图中，我们画出了内存池的结构图，从图中，可以看出，有两个结构变量，分别为MemoryPool和MemoryBlock。

下面我们将从数据结构和接口两个部分出发，详细讲解内存池的设计。

### 数据结构

#### MemoryBlock

本文中所讲述的内存块的分配和释放都是通过该结构进行操作，下面是MemoryBlock的示例图：



MemoryBlock

在上图中，Header存储该MemoryBlock的内存块情况，比如可用的内存块索引、当前MemoryBlock中可用内存块的个数等等。

memory block
- size
- free_size
- first_free
- next
- a_data

定义如下所示：

```
struct MemoryBlock {
    unsigned int size;
    unsigned int free_size;
    unsigned int first_free;

    struct MemoryBlock *next;
    char a_data[0];
};
```

其中：

- size为MemoryBlock下内存块的个数
- free\_size为MemoryBlock下空闲内存块的个数
- first\_free为MemoryBlock中第一个空闲块的索引
- next指向下一个MemoryBlock
- a\_data是一个柔性数组

“

柔性数组即数组大小待定的数组，C语言中结构体的最后一个元素可以是大小未知的数组，也就是所谓的0长度，所以我们可以用结构体来创建柔性数组。


它的主要用途是为了满足需要变长度的结构体，为了解决使用数组时内存的冗余和数组的越界问题。

”

## MemoryPool

MemoryPool为内存池的头，里面定义了该内存池的信息，比如本内存池分配的固定对象的大小，第一个MemoryBlock等

memory pool
- obj_size
- init_size
- grow_size
- first_block

 高性能架构探索

```
struct MemoryPool {
    unsigned int obj_size;
    unsigned int init_size;
    unsigned int grow_size;

    MemoryBlock *first_block;
};
```

其中：

- obj\_size为内存池分配的固定内存块的大小
- init\_size初始化内存池时候创建的内存块的个数
- grow\_size当初初始化内存块使用完后，再次申请内存块时候的个数
- first\_block指向第一个MemoryBlock

## 接口

### memory\_pool\_create

```
MemoryPool *memory_pool_create(unsigned int init_size,
                                unsigned int grow_size,
                                unsigned int size);
```

本函数用来创建一个MemoryPool，并对其进行初始化，下面是参数说明：

- init\_size 表示第一个MemoryBlock中创建块的个数
- grow\_size 表示当MemoryPool中没有空闲块可用，则创建一个新的MemoryBlock时其块的个数
- size 为块的大小(即每次分配相同大小的固定size)

### memory\_alloc

```
void *memory_alloc(MemoryPool *mp);
```

本函数用了从mp中申请一块内存返回

- mp 为MemoryPool类型指针，即内存池的头
- 如果内存分配失败，则返回NULL

### memory\_free

```
void* memory_free(MemoryPool *mp, void *pfree);
```

本函数用来释放内存

- mp 为MemoryPool类型指针，即内存池的头
- pfree 为要释放的内存

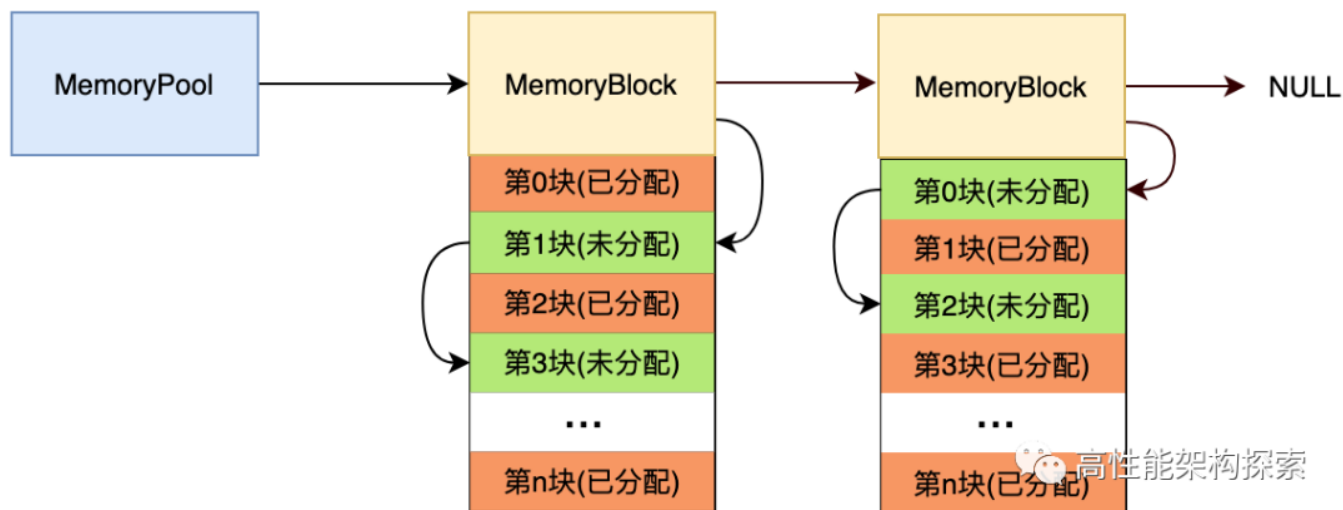
## free\_memory\_pool

```
void free_memory_pool(MemoryPool *mp);
```

本函数用来释放内存池

## 实现

在讲解整个实现之前，我们先看看内存池的详细结构图。



## 初始化内存池

MemoryPool是整个内存池的入口结构，该函数主要是用来创建MemoryPool对象，并使用参数对其内部的成员变量进行初始化。

函数定义如下：

```
MemoryPool *memory_pool_create(unsigned int init_size, unsigned int grow_size, unsigned int size)
{
    MemoryPool *mp;
    mp = (MemoryPool*)malloc(sizeof(MemoryPool));
    mp->first_block = NULL;
    mp->init_size = init_size;
    mp->grow_size = grow_size;

    if(size < sizeof(unsigned int))
        mp->obj_size = sizeof(unsigned int);
    mp->obj_size = (size + (MEMPOOL_ALIGNMENT-1)) & ~(MEMPOOL_ALIGNMENT-1);

    return mp;
}
```

## 内存分配

```
void *memory_alloc(MemoryPool *mp) {

    unsigned int i;
    unsigned int length;

    if(mp->first_block == NULL) {
```



```

MemoryBlock *mb;
length = (mp→init_size)*(mp→obj_size) + sizeof(MemoryBlock);
mb = malloc(length);
if(mb == NULL) {
    perror("memory allocate failed!\n");
    return NULL;
}

/* init the first block */
mb→next = NULL;
mb→free_size = mp→init_size - 1;
mb→first_free = 1;
mb→size = mp→init_size*mp→obj_size;

mp→first_block = mb;

char *data = mb→a_data;

/* set the mark */
for(i=1; i<mp→init_size; ++i) {
    *(unsigned long *)data = i;
    data += mp→obj_size;
}

return (void *)mb→a_data;
}

MemoryBlock *pm_block = mp→first_block;

while((pm_block ≠ NULL) && (pm_block→free_size == 0)) {
    pm_block = pm_block→next;
}

if(pm_block ≠ NULL) {
    char *pfree = pm_block→a_data + pm_block→first_free * mp→obj_size;

    pm_block→first_free = *((unsigned long *)pfree);
    pm_block→free_size--;

    return (void *)pfree;
} else {
    if(mp→grow_size == 0)
        return NULL;

    MemoryBlock *new_block = (MemoryBlock *)malloc((mp→grow_size)*(mp→obj_size) + sizeof(MemoryBlock))

    if(new_block == NULL)
        return NULL;

    char *data = new_block→a_data;

    for(i=1; i<mp→grow_size; ++i) {
        *(unsigned long *)data = i;
        data += mp→obj_size;
    }

    new_block→size = mp→grow_size*mp→obj_size;
    new_block→free_size = mp→grow_size-1;
    new_block→first_free = 1;
    new_block→next = mp→first_block;
    mp→first_block = new_block;

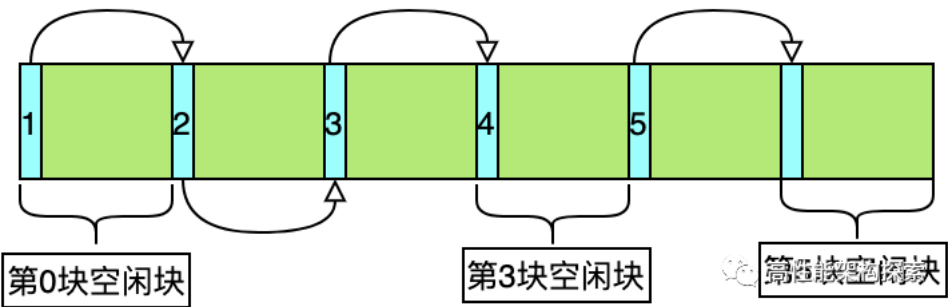
    return (void *)new_block→a_data;
}
}

```

内存块主要在MemoryBlock结构中，也就是说申请的内存，都是从MemoryBlock中进行获取，流程如下：

- 获取MemoryPool中的first\_block指针
  - 如果该指针为空，则创建一个MemoryBlock， first\_block指向新建的MemoryBlock，并返回
  - 否则，从first\_block进行单链表遍历，查找第一个free\_size不为0的MemoryBlock，如果找到，则对该MemoryBlock的相关参数进行设置，然后返回内存块
  - 否则，创建一个新的MemoryBlock，进行初始化分配之后，将其插入到链表的头部（这样做的目的是为了更方便下次分配效率，即减小了链表的遍历）

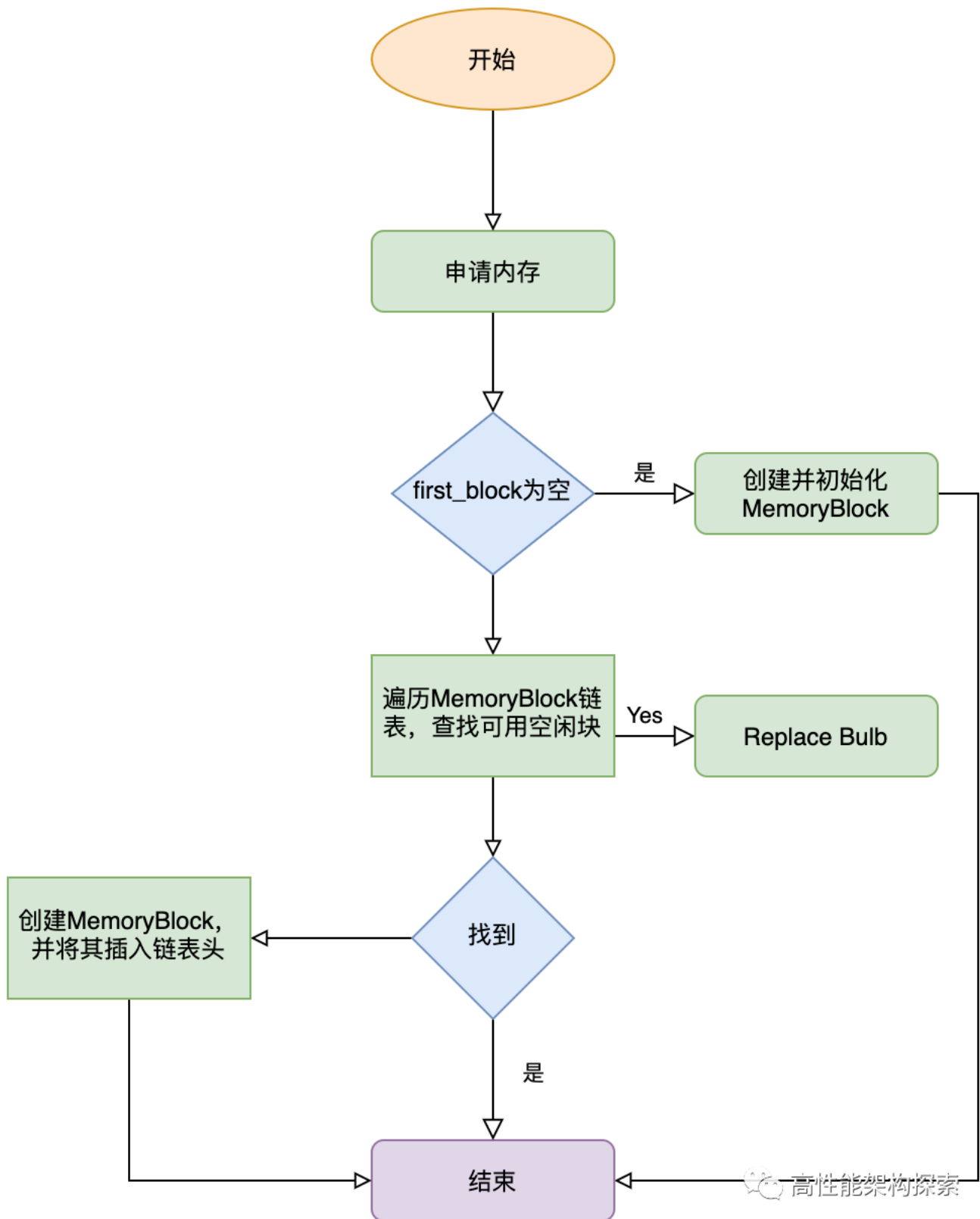
在上述代码中，需要注意的是第30-33行或者67-70行，这两行的功能一样，都是对新申请的内存块进行初始化，这几行的意思，是要将空闲块连接起来，但是，并没有使用传统意义上的链表方式，而是通过index方式进行连接，具体如下图所示：



在上图中，第0块空闲块的下一个空闲块索引为1，而第1块空闲块的索引为2，依次类推，形成了如下链表方式

“  
1→2→3→4→5  
”

内存分配流程图如下所示：



## 内存释放

```
void* memory_free(MemoryPool *mp, void *pfree) {  
    if(mp->first_block == NULL) {  
        return;  
    }  
}
```

```
MemoryBlock *pm_block = mp->first_block;  
MemoryBlock *pm_pre_block = mp->first_block;
```

```

/* research the MemoryBlock which the pfree in */
while(pm_block && ((unsigned long)pfree < (unsigned long)pm_block->a_data ||
(unsigned long)pfree > ((unsigned long)pm_block->a_data+pm_block->size))) {
    //pm_pre_block = pm_block;
    pm_block = pm_block->next;

    if(pm_block == NULL) {
        return pfree;
    }
}

unsigned int offset = pfree -(void*) pm_block->a_data;

if((offset&(mp->obj_size -1)) > 0) {
    return pfree;
}

pm_block->free_size++;
*((unsigned int *)pfree) = pm_block->first_free;

pm_block->first_free=(unsigned int)(offset/mp->obj_size);

return NULL;
}

```

内存释放过程如下：

- 判断当前MemoryPool的first\_block指针是否为空，如果为空，则返回
- 否则，遍历MemoryBlock链表，根据所释放的指针参数判断是否在某一个MemoryBlock中
  - 如果找到，则对MemoryBlock中的各个参数进行操作，然后返回
  - 否则，没有合适的MemoryBlock，则表明该被释放的指针不在内存池中，返回

在上述代码中，需要注意第20-29行。

- 第20行，求出被释放的内存块在MemoryBlock中的偏移
- 第22行，判断是否能被整除，即是否在这个内存块中，算是个double check
- 第26行，将该MemoryBlock中的空闲块个数加1
- 第27-29行，类似于链表的插入，将新释放的内存块的索引放入链表头，而其内部的指向下一个可用内存块

现在举个例子，以便于理解，假设在一开始有5个空闲块，其中前三个空闲块都分配出去了，那么此时，空闲块链表如下：

```

“
4→5, 其中first_free = 4
”

```

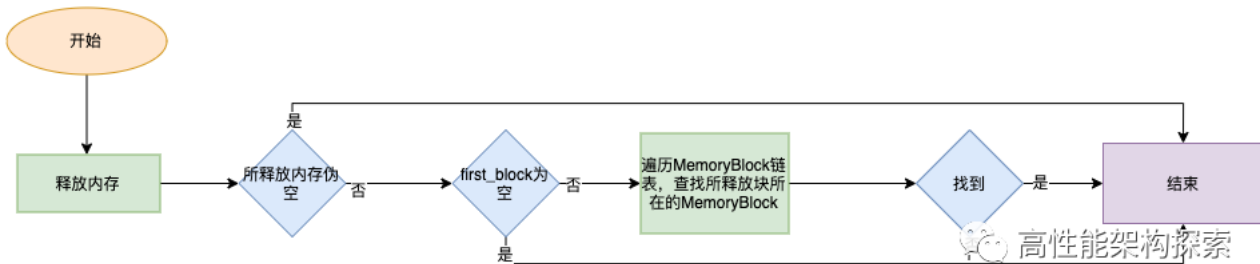
然后在某一个时刻，第1块释放了，那么释放归还之后，如下：

```

“
1→4→5, 其中first_free = 1
”

```

内存释放流程图如下：



内存释放

## 释放内存池

```

void free_memory_pool(MemoryPool *mp) {
    MemoryBlock *mb = mp->first_block;

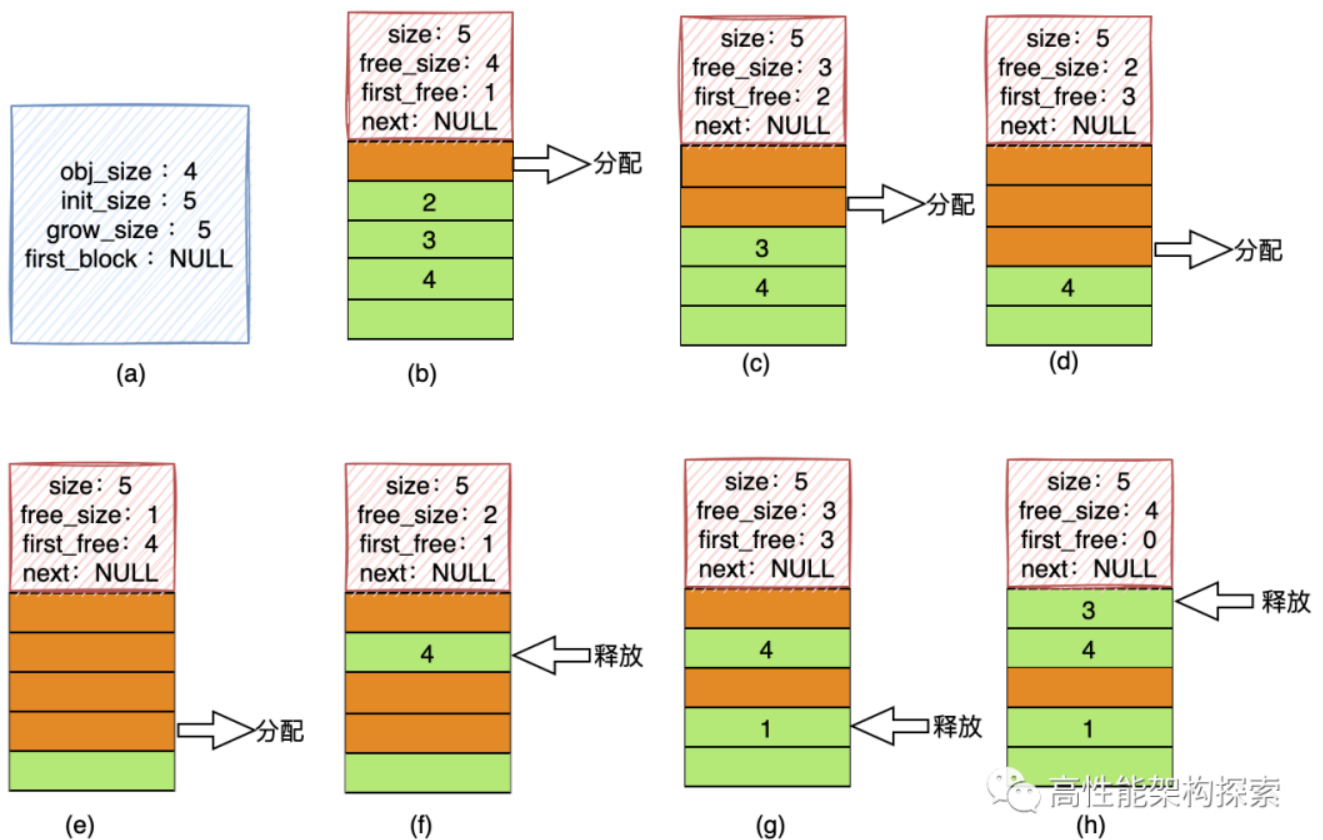
    if(mb != NULL) {
        while(mb->next != NULL) {
            s_memory_block *delete_block = mb;
            mb = mb->next;

            free(delete_block);
        }

        free(mb);
    }

    free(mp);
}

```



上图是一个完整的分配和释放示意图，下面，我结合代码来分析：

- (a)步，创建了一个MemoryPool结构体

- obj\_size = 4代表本内存池分配的内存块大小为4
  - init\_size = 5代表创建内存池的时候，第一块MemoryBlock的空闲内存块个数为5
  - grow\_size = 5代表当申请内存的时候，如果没有空闲内存，则创建的新的MemoryBlock的空闲内存块个数为5
- (b)步，分配出去一块内存
    - 此时，free\_size即该MemoryBlock中可用空闲块个数为4
    - first\_free = 1，代表将内存块分配出去之后，下一个可用的内存块的index为1
  - (c)步，分配出去一块内存
    - 此时，free\_size即该MemoryBlock中可用空闲块个数为3
    - first\_free = 2，代表将内存块分配出去之后，下一个可用的内存块的index为2
  - (d)步，分配出去一块内存
    - 此时，free\_size即该MemoryBlock中可用空闲块个数为2
    - first\_free = 3，代表将内存块分配出去之后，下一个可用的内存块的index为3
  - (e)步，分配出去一块内存
    - 此时，free\_size即该MemoryBlock中可用空闲块个数为1
    - first\_free = 4，代表将内存块分配出去之后，下一个可用的内存块的index为4
  - (f)步，释放第1个内存块
    - 将free\_size进行+1操作
    - first\_free值为此次释放的内存块的索引，而释放的内存块的索引里面的值则为之前first\_free的值(此处释放用的前差法)
  - (g)步，释放第3个内存块
    - 将free\_size进行+1操作
    - first\_free值为此次释放的内存块的索引，而释放的内存块的索引里面的值则为之前first\_free的值(此处释放用的前差法)
  - (h)步，释放第3个内存块
    - 将free\_size进行+1操作
    - first\_free值为此次释放的内存块的索引，而释放的内存块的索引里面的值则为之前first\_free的值(此处释放用的前差法)

## 测试

测试代码如下：

```
#include "memory_pool.h"
#include <sys/time.h>
#include <malloc.h>
#include <stdio.h>

int main() {
    MemoryPool *mp = memory_pool_create(8);

    struct timeval start;
    struct timeval end;
```

```

int t[] = {20000, 40000, 80000, 100000, 120000, 140000, 160000, 180000, 200000};
int s = sizeof(t)/sizeof(int);
for (int i = 0; i < s; ++i) {
    gettimeofday(&start, NULL);
    for (int j = 0; j < t[i]; ++j) {

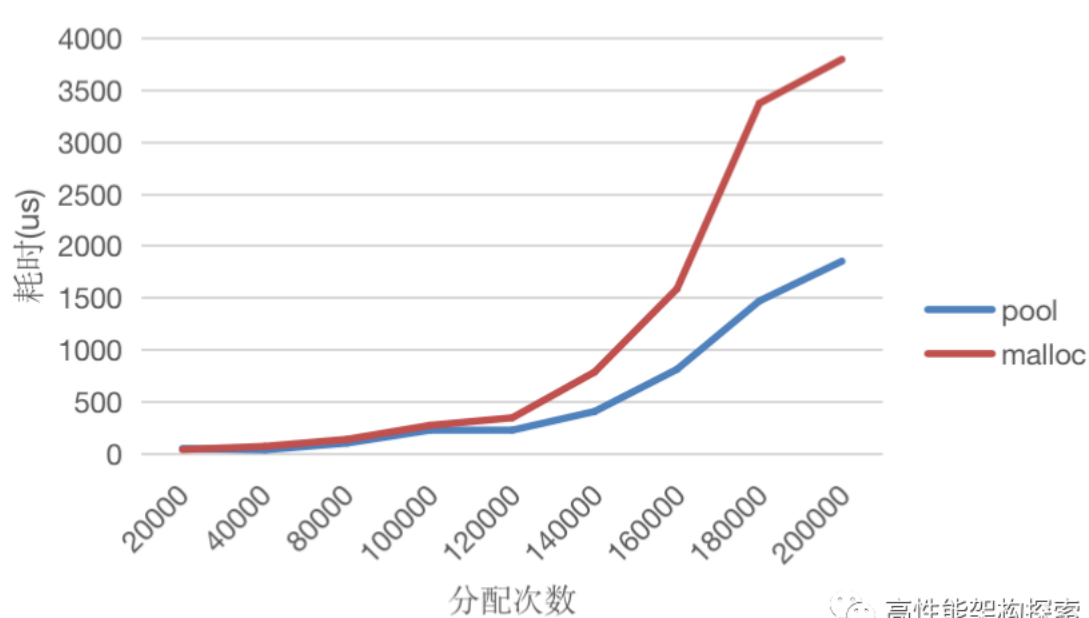
        void *p = memory_alloc(mp);
        memory_free(mp, p);
        //
        //void *p = malloc(8);
        //free(p);
    }
    gettimeofday(&end, NULL);
    long cost = 1000000 * (end.tv_sec - start.tv_sec) +
                end.tv_usec - start.tv_usec;

    printf("%ld\n", cost);
}

free_memory_pool(mp);
return 0;
}

```

数据对比如下：



从上图可以看出，pool的分配效率高于传统的malloc方式，性能提高接近100%

“

本测试结果仅针对当时的项目，对其他测试case不具有普遍性

”

## 扩展

在文章前面，我们有提过本内存池是\_单线程、固定大小的\_，但是往往这种还是不能满足要求，如下几个场景

- 单线程多固定大小
- 多线程固定大小
- 多线程多固定大小

“

多固定大小，指的是提前预支需要申请的内存大小

”

单线程多固定大小：针对此场景，由于已经预知了所申请的size，所以可以针对每个size创建一个内存池

多线程固定大小：针对此场景，有以下两个方案

- 使用ThreadLocalCache
- 每个线程创建一个内存池
- 使用加锁，操作全局唯一内存池（每次加锁解锁耗时100ns左右）

多线程多固定大小：针对此场景，可以结合上述两个方案，即

- 使用ThreadCache，每个线程内创建多固定大小的内存池
- 每个线程内创建一个多固定大小的内存池
- 使用加锁，操作全局唯一内存池（每次加锁解锁耗时100ns左右）

“

上述几种方案，仅仅是在使用固定大小内存池基础上进行的扩展，具体的方案，需要根据具体情况来具体分析

”

## 结语

本文主要讲了固定大小内存池的实现方式，因为实现方案的局限性，此内存池设计方案仅适用于每次申请都是特定大小的场景。虽然在扩展部分做了部分思维发散，但因为未做充分的数据对比，所以仅限于思维扩散。

目前，开源的内存分配库很多，比较优秀的有谷歌的tcmalloc以及微软的mimalloc，大家可以根据自己项目的需求场景，选择合适的内存分配库。

今天的文章就到这里，下期见。

“

本文所讲的内存池源码地址：

[https://github.com/namelij/fixedsize\\_memorypool](https://github.com/namelij/fixedsize_memorypool)

别忘了给个star哦👉

”