C 中的固定块内存分配器

大卫·拉弗尼尔

2018 年 12 月 25 日 警察

评价我: 5.00/5 (13 票)

独特的分配器功能可提高性能并防止任何 C 或 C++ 项目上的堆碎片故障。

下载源代码 - 19.6 KB

介绍

1998 年，我为 Embedded Systems Programming 杂志写了一篇关于固定块内存分配器的文章。这件作品我收到了 750 美元。现在，在 Code Project 等网站上免费撰写文章。哦，时间是如何保持变化的。

没有改变的一件事是固定块分配器的用处。在某些设备上使用全局堆是禁忌。在我的整个职业生涯中，我编写了许多固定块内存分配器，以在内存受限系统中提供类似动态的高速分配。通用的、设计良好的固定块分配器通过在禁止使用堆的系统中提供动态内存分配，开辟了应用程序实现的可能性。

在 Code Project 中，我记录了各种 C++ 分配器实现（请参阅参考文章部分）。这一次，我将展示一个具有适用于嵌入式设备或其他设备的独特功能的 C 语言版本。

此处介绍的解决方案将：

- 比全局堆更快
- 易于使用
- 线程安全
- 支持 malloc、free、realloc 和 calloc 的固定块版本
- 使用最少的代码空间
- 定时执行
- 消除堆碎片内存故障
- 不需要额外的存储开销（除了几个字节的静态内存）
- 处理内存对齐
- 根据需求自动分配可变块大小（la malloc）

正如我将要展示的，两个分配和回收内存的简单 C 模块将提供上述所有好处。

背景

自定义固定块内存分配器用于解决至少两种类型的内存相关问题。首先，全局堆分配/解除分配可能很慢且不确定。你永远不知道内存管理器需要多长时间。其次，消除由碎片堆引起的内存分配错误的可能性——这是一个值得关注的问题，尤其是在任务关键型系统上。

即使系统不被认为是关键任务，一些嵌入式系统也可以在不重启的情况下运行数周或数年。根据分配模式和堆实现，长期使用堆会导致堆故障。

fb_allocator

每个`fb_allocator`实例处理单个块大小。界面如下图所示：

C++ 复制代码

```
void ALLOC_Init(void);
void ALLOC_Term(void);
void* ALLOC_Alloc(ALLOC_HANDLE hAlloc, size_t size);
void* ALLOC_Calloc(ALLOC_HANDLE hAlloc, size_t num, size_t size);
void ALLOC_Free(ALLOC_HANDLE hAlloc, void* pBlock);
```

`ALLOC_Init()`在启动和`ALLOC_Term()`关闭时调用一次。其余的每个 API 都以与 CRT 对应的相同方式运行；`ALLOC_Alloc()`分配内存和`ALLOC_Free()` 释放。

`fb_allocator`使用`ALLOC_DEFINE`宏在文件范围内创建一个实例。`TestAlloc` 下面的分配器定义了一个最多有五个 16 字节块的固定块分配器。

C++ 复制代码

```
ALLOC_DEFINE(TestAlloc, 16, 5);
```

分配一个 16 字节的块很简单。

C++ 复制代码

```
void* data = ALLOC_Alloc(TestAlloc, 16);
```

完成后释放块。

C++ 复制代码

```
ALLOC_Free(TestAlloc, data);
```

x_分配器

该`x_allocator`模块使用两个或多个`fb_allocator`实例处理多个内存块大小；一个`fb_allocator`每块大小。在分配过程中，根据调用者请求的大小`x_allocator`返回一个从`fb_allocator`实例之一确定大小的块。的`x_allocator`API如下所示：

C++ 复制代码

```
void* XALLOC_Alloc(XAllocData* self, size_t size);
void XALLOC_Free(void* ptr);
void* XALLOC_Realloc(XAllocData* self, void *ptr, size_t new_size);
void* XALLOC_Calloc(XAllocData* self, size_t num, size_t size);
```

用户`x_allocator`通常会创建一个瘦包装器模块，该模块 (a) 定义两个或多个`fb_allocator`实例，以及 (b) 提供自定义 API 来访问`x_allocator`内存。用一个简单的例子来解释更容易。

我的分配器

假设我们想要一个固定的块分配器来分配两个块大小：32 和 128。我们将调用它`my_allocator`，API 如下所示：

C++ 复制代码

```
void* MYALLOC_Alloc(size_t size);
void MYALLOC_Free(void* ptr);
void* MYALLOC_Realloc(void *ptr, size_t new_size);
void* MYALLOC_Calloc(size_t num, size_t size);
```

该实现创建了多个fb_allocator实例；一个来处理每个所需的块大小。在这种情况下，我们最多允许 10 个 32 字节块和 5 个 128 字节块。

C++ 复制代码

```
#include "my_allocator.h"
#include "x_allocator.h"

// Maximum number of blocks for each size
#define MAX_32_BLOCKS   10
#define MAX_128_BLOCKS  5

// Define size of each block including meta data overhead
#define BLOCK_32_SIZE     32 + XALLOC_BLOCK_META_DATA_SIZE
#define BLOCK_128_SIZE    128 + XALLOC_BLOCK_META_DATA_SIZE

// Define individual fb_allocators
ALLOC_DEFINE(myDataAllocator32, BLOCK_32_SIZE, MAX_32_BLOCKS)
ALLOC_DEFINE(myDataAllocator128, BLOCK_128_SIZE, MAX_128_BLOCKS)

// An array of allocators sorted by smallest block first
static ALLOC_Allocator* allocators[] = {
    &myDataAllocator32Obj,
    &myDataAllocator128Obj
};

#define MAX_ALLOCATORS   (sizeof(allocators) / sizeof(allocators[0]))

static XAllocData self = { allocators, MAX_ALLOCATORS };
```

现在，简单的一行包装函数提供对底层x_allocator模块的访问。

C++ 复制代码

```
void* MYALLOC_Alloc(size_t size)
{
    return XALLOC_Alloc(&self, size);
}

void MYALLOC_Free(void* ptr)
{
    XALLOC_Free(ptr);
}

void* MYALLOC_Realloc(void *ptr, size_t new_size)
{
    return XALLOC_Realloc(&self, ptr, new_size);
}

void* MYALLOC_Calloc(size_t num, size_t size)
{
    return XALLOC_Calloc(&self, num, size);
}
```

当调用者MYALLOC_Alloc()以 1 到 32 之间的大小调用时，返回一个 32 字节的块。如果请求的大小在 33 到 128 之间，则提供 128 字节的块。MYALLOC_Free()将块返回到原始fb_allocator实例。通过这种方式，一组固定块内存分配器被组合在一起，在运行时根据应用程序需求提供可变大小的内存块。样本包装器模式被反复使用，为系统内的特定目的提供内存块组。

实施细则

大多数分配器实现相对简单。但是，我将解释一些细节以帮助理解关键概念。

fb_allocator 空闲列表

这是一种方便的技术，可以将空闲列表中的块链接在一起，而无需为指针消耗任何额外的存储空间。在用户调用之后`ALLOC_Free()`，一个固定的内存块不再被使用，而是被释放以用于其他事情，比如下一个指针。由于`fb_allocator`模块需要保留已删除的块，我们将列表的下一个指针放在当前未使用的块空间中。当块被应用程序重用时，不再需要该指针并将被用户对象覆盖。这样，将块链接在一起就不会产生每个实例的存储开销。

自由列表实际上是作为单链表实现的，但代码只从头部添加/删除，因此行为是堆栈的行为。使用堆栈使分配/解除分配非常快速且具有确定性。没有循环搜索空闲块——只需推或弹出一个块就可以了。

使用释放的对象空间作为将块链接在一起的内存意味着对象必须足够大以容纳指针。在`ALLOC_BLOCK_SIZE` 该最小尺寸满足宏观保证。

fb_allocator 内存对齐

一些嵌入式系统要求内存在特定的字节边界上对齐。由于分配器的内存是一个连续的`static`字节数组，因此在未对齐的边界上启动块可能会导致某些 CPU 出现硬件异常。例如，如果需要 4 字节对齐，则 13 字节块将导致问题。更改`ALLOC_MEM_ALIGN` 为所需的字节边界。块大小将向上舍入到下一个最近的对齐边界。

x_allocator 元数据

该`x_allocator`实现为每个块添加 4 字节的元数据。例如，如果用户需要 32 字节的块，则`x_allocator`实际使用 36 字节的块。额外的 4 字节用于隐藏`fb_allocator`块内的指针（假设指针大小为 4 字节）。

删除内存时，`x_allocator`需要原始`fb_allocator`实例，以便将释放请求路由到正确的`fb_allocator`实例进行处理。与`XALLOC_Alloc()`,`XALLOC_Free()`不接受大小，只使用`void*`参数。因此，`XALLOC_Alloc()`实际上`fb_allocator`通过向请求添加额外的 4 字节来隐藏指向内存块未使用部分的指针。调用者获得一个指向块的客户区域的`fb_allocator`指针，其中隐藏的指针没有被覆盖。

C++ 缩小▲ 复制代码

```
void* XALLOC_Alloc(XAllocData* self, size_t size)
{
    ALLOC_Allocator* pAllocator;
    void* pBlockMemory = NULL;
    void* pClientMemory = NULL;

    ASSERT_TRUE(self);

    // Get an allocator instance to handle the memory request
    pAllocator = XALLOC_GetAllocator(self, size);

    // An allocator found to handle memory request?
    if (pAllocator)
    {
        // Get a fixed memory block from the allocator instance
        pBlockMemory = ALLOC_Alloc(pAllocator, size + XALLOC_BLOCK_META_DATA_SIZE);
        if (pBlockMemory)
        {
            // Set the block ALLOC_Allocator* ptr within the raw memory block region
            pClientMemory = XALLOC_PutAllocatorPtrInBlock(pBlockMemory, pAllocator);
        }
    }
    else
    {
        // Too large a memory block requested
        ASSERT();
    }

    return pClientMemory;
}
```

当XALLOC_Free()被调用时，分配器指针从存储器块中提取，以便正确的fb_allocator实例可以被称为解除分配块。

C++ 复制代码

```
void XALLOC_Free(void* ptr)
{
    ALLOC_Allocator* pAllocator = NULL;
    void* pBlock = NULL;

    if (!ptr)
        return;

    // Extract the original allocator instance from the caller's block pointer
    pAllocator = XALLOC_GetAllocatorPtrFromBlock(ptr);
    if (pAllocator)
    {
        // Convert the client pointer into the original raw block pointer
        pBlock = XALLOC_GetBlockPtr(ptr);

        // Deallocate the fixed memory block
        ALLOC_Free(pAllocator, pBlock);
    }
}
```

基准测试

在 Windows PC 上对分配器性能与全局堆进行基准测试显示代码的速度有多快。以某种交错的方式分配和解除分配 20000 4096 和 2048 大小的块的基本测试测试了速度的提高。有关确切算法，请参阅随附的源代码。

以毫秒为单位的 Windows 分配时间

分配器	模式	跑	基准时间（毫秒）
全局堆	释放	1	36.3
全局堆	释放	2	33.8
全局堆	释放	3	32.8
fb_allocator	静态池	1	22.6
fb_allocator	静态池	2	3.7
fb_allocator	静态池	3	4.9
x_allocator	静态池	1	33.9
x_allocator	静态池	2	6.9
x_allocator	静态池	3	7.7

Windows 在调试器中执行时使用调试堆。调试堆添加了额外的安全检查，从而降低了其性能。由于检查被禁用，释放堆要快得多。通过_NO_DEBUG_HEAP=1在**调试**>**环境**项目选项中设置，可以在 Visual Studio 中禁用调试堆。

此基准测试非常简单，具有不同块大小和随机新/删除间隔的更现实场景可能会产生不同的结果。但是，基本点很好地说明了；内存管理器比分配器慢，并且高度依赖于平台的实现。

在fb_allocator 使用静态内存池，并且不依靠堆。一旦空闲列表填充了块，这将具有大约 4 毫秒的快速执行时间。fb_allocator运行 1上的 22.6 毫秒用于在第一次运行时将固定内存池划分为单独的块。

的x_allocator 所述内使用bm_allocator模块是慢一点在∽7毫秒，因为它具有开销而分配/解除分配多个大小的块fb_allocator只支持一个块大小。

与 Windows 全局堆相比，`fb_allocator` 它大约快 8 倍，`x_allocator` 大约快 5 倍。在嵌入式设备上，我看到全局堆的速度提高了 15 倍。

线程安全

模块中的`LK_LOCK` 和`LK_UNLOCK` 宏`LockGuard`实现了线程安全所需的软件锁。根据您的平台操作系统的需要更新锁实现。

参考文章

- 一个高效的 C++ 固定块内存分配器- David Lafreniere
- 用快速固定块内存分配器替换 malloc/free - 作者 David Lafreniere
- 自定义 STL std::allocator 替换提高了性能- David Lafreniere

结论

这里介绍的基于 C 的固定块内存分配器适用于任何 C 或 C++ 系统。对于具有自己独特功能的 C++ 特定实现，请参阅参考文章。

使用`fb_allocator`时，你需要分配一个单独的块大小。`x_allocator`当需要分配多种块尺寸时使用。创建多个`x_allocator`包装器以根据预期用途隔离内存池。

如果您有一个应用程序真正影响堆并导致性能下降，或者如果您担心碎片堆故障，集成`fb_allocator`并`x_allocator`可能有助于解决这些问题。该实现被保持在最低限度，便于在最小的嵌入式系统上使用。

历史

- 23届十二月，2018
  - 初始发行
- 2018 年 12 月 24日
  - 在文章中添加了基准测试部分
  - 更新了附加的源代码

执照

分享

关于作者

大卫·拉弗尼尔

美国

手表该会员

我做专业软件工程师已经超过 20 年了。不编写代码时，我喜欢与家人共度时光，在南加州露营和骑摩托车。

评论和讨论

添加评论或问题 | 电子邮件提醒 | Search Comments

第一 页上一页 下一页

你在哪里分配或 malloc 初始块?
会员 14971006 20 年 10 月 29 日 6:36

回复：你在哪里分配或 malloc 初始块?
David Lafreniere 30-Dec-20 5:32

my_allocator 的实现
Member 14666909 3-Dec-19 19:38

回复：my_allocator 的实现
David Lafreniere 2-Apr-20 21:54

快多少?
fmuzul2 24-Dec-18 19:05

回复: 多快?
David Lafreniere 25-Dec-18 0:24

刷新 1

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。