一个高效的 C++ 固定块内存分配器

大卫·拉弗尼尔

2016 年 3 月 28 日 警察

评价我: 4.89/5 (16 票)

一个固定的块内存分配器，可提高系统性能并提供堆碎片故障保护

下载 Allocator.zip - 5.5 KB

介绍

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。

典型的解决方案是预先静态声明所有对象并完全摆脱动态分配。然而，静态分配可能会浪费存储空间，因为对象即使没有被积极使用，也会存在并占用空间。此外，围绕动态分配实现系统可以提供更自然的设计架构，而不是静态声明所有对象。

固定块内存分配器并不是一个新想法。长期以来，人们一直在设计各种自定义内存分配器。我在这里展示的是一个简单的 C++ 分配器实现，我已经成功地在各种项目中使用了它。

此处介绍的解决方案将：

- 比全局堆更快
- 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）
- 易于使用
- 使用最少的代码空间

正如我将展示的，一个分配和回收内存的简单类将提供所有上述好处。

读完这篇文章后，请务必阅读后续文章“**更换的malloc /免费用快速固定块内存分配器**”，看看如何`Allocator`被用来创建一个真正的快速`malloc()`和`free()`CRT更换。

存储回收

内存管理方案的基本原理是回收在对象分配期间获得的内存。一旦为对象创建了存储空间，它就永远不会返回到堆中。相反，内存被回收，允许另一个相同类型的对象重用空间。我已经实现了一个称为`Allocator`表达技术的类。

当应用程序删除 using 时Allocator，单个对象的内存块被释放以供再次使用，但实际上并未释放回内存管理器。释放的块保留在一个称为空闲列表的链表中，以再次分配给另一个相同类型的对象。对于每个分配请求，Allocator首先检查现有内存块的空闲列表。只有在没有可用的情况下才会创建一个新的。根据 的期望行为Allocator，存储来自全局堆或static具有以下三种操作模式之一的内存池：

1. 堆块
2. 堆池
3. 静态池

堆与池

的Allocator类能够创建从堆或存储器池每当自由列表不能提供现有的新块。如果使用池，则必须预先指定对象数。使用总对象，创建一个足够大的池来处理最大数量的实例。另一方面，从堆中获取块内存没有这样的数量限制——在存储允许的情况下构造尽可能多的新对象。

该堆块从全局堆模式分配用于单个对象作为必要新的存储器块实现的存储器请求。释放会将块放入空闲列表以供以后重用。当空闲列表为空时，从堆中创建新的块使您不必设置对象限制。这种方法提供类似动态的操作，因为块的数量可以在运行时扩展。缺点是在块创建期间丢失了确定性执行。

该堆池模式从全局堆中创建一个池来保存所有块。池是在构造对象operator new时创建的Allocator。Allocator然后在分配期间从池中提供内存块。

在静态池模式使用一个单一的存储池，通常位于静态存储器来保存所有块。静态内存池不是由Allocator类的用户创建，而是由类的用户提供。

堆池和静态池模式提供一致的分配执行时间，因为内存管理器从不参与获取单个块。这使得新操作非常快速且具有确定性。

班级设计

类接口非常简单。Allocate()返回一个指向内存块的指针并Deallocate()释放内存块以供再次使用。构造函数负责设置对象大小，并在必要时分配内存池。

传递给类构造函数的参数决定了新块的获取位置。该size参数控制固定内存块大小。该objects参数设置允许的块数。0 意味着根据需要从堆中获取新块，而任何其他非零值表示使用池、堆或静态来处理指定数量的实例。所述memory参数是一个指向可选static存储器。如果memory是0和objects不是0，Allocator将从堆中创建一个池。静态内存池的大小必须为sizexobjects字节。该name参数可选地为分配器提供一个名称，这对于收集分配器使用指标很有用。

C++ 复制代码

```
class Allocator
{
public:
    Allocator(size_t size, UINT objects=0, CHAR* memory=NULL, const CHAR* name=NULL);
...
```

下面的示例显示了如何通过构造函数参数控制三种操作模式中的每一种。

C++ 复制代码

```
// Heap blocks mode with unlimited 100 byte blocks
Allocator allocatorHeapBlocks(100);

// Heap pool mode with 20, 100 byte blocks
Allocator allocatorHeapPool(100, 20);

// Static pool mode with 20, 100 byte blocks
char staticMemoryPool[100 * 20];
Allocator allocatorStaticPool(100, 20, staticMemoryPool);
```

为了static稍微简化池方法，使用了一个简单的AllocatorPool<>模板类。第一个模板参数是块类型，第二个参数是块数量。

C++ 复制代码

```
// Static pool mode with 20 MyClass sized blocks
AllocatorPool<MyClass, 20> allocatorStaticPool2;
```

通过调用Allocate()，返回一个指向一个实例大小的内存块的指针。获取块时，检查空闲列表以确定费用块是否已存在。如果是这样，只需从空闲列表中取消链接现有块并返回指向它的指针；否则从池/堆中创建一个新的。

C++ 复制代码

```
void* memory1 = allocatorHeapBlocks.Allocate(100);
```

Deallocate()只是将块地址压入堆栈。堆栈实际上是作为一个单向链表（自由列表）实现的，但该类仅从头部添加/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。无需搜索列表 - 只需推送或弹出一个块即可。

C++ 复制代码

```
allocatorHeapBlocks.Deallocate(memory1);
```

现在可以方便地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。例如，如果我们使用 global operator new，则首先分配存储，然后调用构造函数。销毁过程正好相反；调用析构函数，然后释放内存。在析构函数执行之后，但在存储被释放回堆之前，内存不再被对象使用，而是被释放以用于其他事情，比如下一个指针。由于Allocator班级需要保留已删除的块，因此在operator delete我们将列表的下一个指针放在当前未使用的对象空间中。当块被应用程序重用时，不再需要指针，并将被新形成的对象覆盖。这样，就不会产生每个实例的存储开销。

使用释放的对象空间作为将块链接在一起的内存意味着该对象必须足够大以容纳一个指针。构造函数初始值设定项列表中的代码确保最小块大小永远不会低于指针大小。

类析构函数通过删除内存池来释放在执行期间分配的存储，或者如果块是从堆中获得的，则通过遍历空闲列表并删除每个块。由于Allocator该类通常用作类作用域static，因此只会在程序终止时调用它。对于大多数嵌入式设备，当有人从系统中拔出电源时，应用程序就会终止。显然，在这种情况下，不需要析构函数。

如果您使用堆块方法，则在应用程序终止时无法释放分配的块，除非将所有实例都签入空闲列表。因此，必须在程序结束前“删除”所有未完成的对象。否则，你自己就会有一个很好的内存泄漏。这提出了一个有趣的观点。不必Allocator同时跟踪空闲块和已用块吗？最简洁的答案是不。长的答案是，一旦通过指针将块提供给应用程序，应用程序就有责任在程序结束之前Allocator通过调用返回该指针Deallocate()。这样，我们只需要跟踪释放的块。

使用代码

我希望Allocator非常易于使用，因此我创建了宏来自动化客户端类中的界面。宏提供了和 两个成员函数的static实例Allocator：operator new和operator delete。通过重载new和delete操作符，Allocator拦截和处理客户端类的所有内存分配任务。

的DECLARE_ALLOCATOR宏提供头文件接口，应包括类声明这样内：

C++ 复制代码

```
#include "Allocator.h"
class MyClass
{
    DECLARE_ALLOCATOR
    // remaining class definition
};
```

该operator new函数调用Allocator为类的单个实例创建内存。根据定义，分配内存后，将operator new调用该类的适当构造函数。当重载时，new只能接管内存分配任务。构造函数调用由语言保证。同样，在删除一个对象时，系统首先为我们调用析构函数，然后operator delete被执行。在operator delete使用Deallocate()功能存储在自由列表中的内存块。

在 C++ 程序员中，使用基指针删除类时，析构函数应该声明为虚拟的，这是一个相对普遍的知识。这可确保在删除类时调用正确的派生析构函数。然而，不太明显的是虚拟析构函数如何更改operator delete调用哪个类的重载。

虽然没有明确声明，但它`operator delete`是一个`static`函数。因此，它不能被声明为虚拟的。因此，乍一看，人们会认为删除带有基指针的对象无法路由到正确的类。毕竟，`static`使用基指针调用普通函数将调用基成员的版本。然而，正如我们所知，调用 a `operator delete`first 会调用析构函数。使用虚拟析构函数，调用被路由到派生类。在类的析构函数执行后，将`operator delete`调用该派生类的 。所以本质上，重载`operator delete`通过虚拟析构函数路由到派生类。因此，如果执行使用基指针的删除，则基类析构函数必须声明为虚拟的。否则，`operator delete`将调用错误的析构函数和重载。

所述`IMPLEMENT_ALLOCATOR`宏是所述接口的源文件部分和应该被放置在文件范围内。

C++ 复制代码

```
IMPLEMENT_ALLOCATOR(MyClass, 0, 0)
```

一旦宏就位，调用者就可以创建和销毁此类的实例，并且存储的删除对象将被回收：

C++ 复制代码

```
MyClass* myClass = new MyClass();
delete myClass;
```

单继承和多继承情况都适用于`Allocator`该类。例如，假设类`Derived`继承自类`Base`，以下代码片段是合法的。

C++ 复制代码

```
Base* base = new Derived;
delete base;
```

运行

在运行时，`Allocator`空闲列表中最初没有块，因此第一次调用`Allocate()`将从池或堆中获取块。随着执行的继续，系统对任何给定分配器实例的对象的需求都会波动，只有在空闲列表无法提供现有块时才会分配新的存储空间。最终，系统将稳定在某个峰值数量的实例中，以便每个分配都将从现有块而不是池/堆中获得。

与使用内存管理器获取所有块相比，该类节省了大量的处理能力。在分配期间，一个指针只是从空闲列表中弹出，使其非常快。解除分配只是将一个块指针推回到列表中，这同样快。

基准测试

`Allocator`在 Windows PC 上对性能与全局堆进行基准测试显示该类的速度有多快。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	调试堆	1	1640
全局堆	调试堆	2	1864年
全局堆	调试堆	3	1855年
全局堆	释放堆	1	55
全局堆	释放堆	2	47
全局堆	释放堆	3	47
分配器	静态池	1	19
分配器	静态池	2	7

分配器	模式	跑	基准时间（毫秒）
分配器	静态池	3	7
分配器	堆块	1	30
分配器	堆块	2	7
分配器	堆块	3	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。通过在**调试 > 环境** 项目选项中设置**_NO_DEBUG_HEAP=1**，可以在 Visual Studio 中禁用调试堆 。

可以预见，调试全局堆是最慢的，大约为 1.8 秒。释放堆在大约 50 毫秒时要快得多。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`Allocator`静态池模式运行不依赖于堆。一旦空闲列表填充了块，这将具有大约 7 毫秒的快速执行时间。运行 1 上的 19 毫秒考虑了在第一次运行时将固定内存池划分为单独的块。

`Allocator`一旦使用从堆获得的块填充空闲列表，运行堆块模式的速度也一样快。回想一下，堆块模式依赖于全局堆来获取新块，然后将它们回收到空闲列表中供以后使用。运行 1 显示了在 30 毫秒时创建内存块的分配命中。由于空闲列表已完全填充，随后的基准测试时钟以非常快的 7 毫秒计时。

正如基准测试所示，`Allocator`它非常高效，并且比 Windows 全局发布堆快 7 倍。

为了在嵌入式系统上进行比较，我使用在 168MHz ARM STM32F4 CPU 上运行的 Keil 进行了相同的测试。由于资源有限，我不得不将最大块降低到 500，块大小降低到 32 和 16 字节。这是结果。

ARM STM32F4 分配时间（以毫秒为单位）

分配器	模式	跑	基准时间（毫秒）
全局堆	释放	1	11.6
全局堆	释放	2	11.6
全局堆	释放	3	11.6
分配器	静态池	1	0.85
分配器	静态池	2	0.79
分配器	静态池	3	0.79
分配器	堆块	1	1.19
分配器	堆块	2	0.79
分配器	堆块	3	0.79

正如 ARM 基准测试结果所示，`Allocator`该类的速度提高了大约 15 倍，这是非常重要的。需要注意的是，基准测试确实加重了 Keil堆。在 ARM 情况下，基准测试分配 500 个 16 字节块。然后每隔一个 16 字节的块被删除，然后分配 500 个 32 字节的块。最后一组 500 次分配将 11.6 毫秒的总时间增加了 9.2 毫秒。这意味着当堆碎片化时，您可以预期内存管理器在不确定的时间内花费更长的时间。

分配器决定

首先要做的决定是你是否需要一个分配器。如果您的项目没有执行速度或容错要求，您可能不需要自定义分配器，全局堆就可以正常工作。

另一方面，如果您确实需要速度和/或容错，分配器可以提供帮助，操作模式取决于您的项目要求。关键任务设计的架构师可能会禁止使用全局堆。然而，动态分配可能会导致更高效或更优雅的设计。在这种情况下，您可以在调试开发期间使用堆块模式来获取内存使用指标，然后释放切换到`static`池方法以创建静态分配的池，从而消除所有全局堆访问。一些编译时宏在模式之间切换。

或者，堆块模式可能适用于应用程序。它确实利用堆来获取新块，但是一旦空闲列表填充了足够多的块，它就可以防止堆碎片错误并加快分配速度。

虽然由于本文范围之外的多线程问题而未在源代码中实现，但很容易让`Allocator`构造函数保留`static`所有构造实例的列表。运行系统一段时间，然后在某个指针处通过`GetBlockCount()`和`GetName()`函数遍历所有分配器和输出指标，如块计数和名称。使用指标提供有关在切换到内存池时为每个分配器调整固定内存池大小的信息。始终添加比最大测量块数多一些的块，以便为系统提供一些额外的弹性，以应对内存不足的池。

调试内存泄漏

调试内存泄漏可能非常困难，主要是因为堆是一个黑匣子，无法查看分配的对象的类型和大小。使用`Allocator`，内存泄漏更容易发现，因为它会`Allocator`跟踪总块数。重复输出（到控制台为例）`GetBlockCount()`和`GetName()`为每个分配器实例和比较的差异应与不断增加的块计数露出分配器。

错误处理

C++ 中的分配错误通常是使用 new-handler 函数捕获的。如果内存管理器在尝试从堆中分配内存时出现故障，则通过 new-handler 函数指针调用用户的错误处理函数。通过将用户的函数地址分配给新处理程序，内存管理器能够调用自定义错误处理例程。为了使`Allocator`类的错误处理一致，超过池存储容量的分配也会调用 new-handler 指向的函数，将所有内存分配错误集中在一个地方。

C++ 复制代码

```
static void out_of_memory()
{
    // new-handler function called by Allocator when pool is out of memory
    assert(0);
}

int _tmain(int argc, _TCHAR* argv[])
{
    std::set_new_handler(out_of_memory);
...
```

限制

该类不支持对象数组。重载`operator new[]`给对象回收方法带来了问题。调用此函数时，`size_t`参数包含所有数组元素所需的总内存。为每个元素创建单独的存储不是一种选择，因为多次调用 new 并不能保证块位于数组需要的连续空间内。由于`Allocator`仅处理相同大小的块，因此不允许使用数组。

移植问题

`Allocatornew_handler()`当`static`池耗尽存储时直接调用，这可能不适用于某些系统。此实现假设 new-handler 函数不会返回，例如无限循环陷阱或断言，因此如果该函数通过压缩堆解决分配失败，则调用处理程序是没有意义的。正在使用固定池，任何压缩都无法弥补这一点。

参考文章

更换的malloc /免费用快速固定块内存分配器 来看看如何`Allocator`来创建一个真正的快速`malloc()`和`free()`CRT更换。

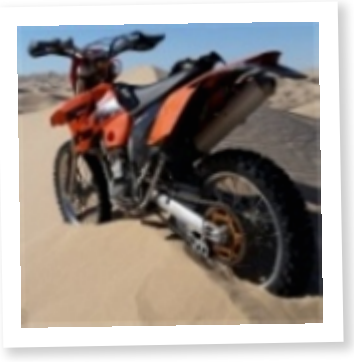
历史

- 9个月2016
  - 首次发布
- 11个月2016
  - 添加了对“**用快速固定块内存分配器替换 malloc/free** ”文章的引用
- 13个月2016
  - 更新了基准测试部分以包括 Windows 调试堆、非调试堆和 Keil ARM STM32F4 时间。
  - 新的源代码 zip 文件。
- 28个月2016
  - 更新了附加的源代码以帮助移植。

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

MyClass* myClass2 = new MyClass[2]; 删除[] myClass2; => 不工作？！？！
ehaerim 31-Oct-19 2:33

回复：MyClass* myClass2 = new MyClass[2]; 删除[] myClass2; => 不工作？！？！
ehaerim 20-Nov-19 1:18

跨线程使用分配器
lesinar 20-Nov-16 23:05

回复：跨线程使用分配器
David Lafreniere 21-Nov-16 0:34

需要为导出的类使用分配器
ehaerim 4-Apr-16 3:36

回复：需要为导出的类使用分配器
David Lafreniere 5-Apr-16 7:23

好文章！！
Bharath NS 30-Mar-16 6:40

回复：好文章！！
David Lafreniere 31-Mar-16 7:29

回复：好文章！！
alexissizla 12-Jun-16 15:33

回复：好文章！！
David Lafreniere 12-Jun-16 21:00

IsValidBlock 会很有用
ehaerim 29-Mar-16 10:19

回复：IsValidBlock 会很有用
David Lafreniere 31-Mar-16 7:37

What if a pool size is not known in advance but determined later on?
ehaerim 27-Mar-16 10:10

Re: What if a pool size is not known in advance but determined later on?
David Lafreniere 28-Mar-16 23:41

Wrong Measurements?
Alois Kraus 12-Mar-16 22:40

Re: Wrong Measurements?
David Lafreniere 13-Mar-16 3:16

Re: Wrong Measurements?
David Lafreniere 14-Mar-16 1:55

Re: Wrong Measurements?
Alois Kraus 14-Mar-16 15:40

thank you, for gives me interesting ideas
samsom 9-Mar-16 9:34

Re: thank you, for gives me interesting ideas
David Lafreniere 9-Mar-16 18:27

Compatible with STD?
pip010 8-Mar-16 22:26

Re: Compatible with STD?
David Lafreniere 9-Mar-16 8:00

Source code?
Martin Capoušek 7-Mar-16 16:51

Re: Source code?
David Lafreniere 7-Mar-16 19:13

Refresh 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。