自己动手实现一个malloc内存分配器 | 30图

Original 码农的荒岛求生 码农的荒岛求生 2021-01-05 16:15

收录于合集
#高并发&高性能 24 #程序员 21 #编程 14 #内存 7 #内存分配 1

对内存分配器透彻理解是编程高手的标志之一。

如果你不能理解malloc之类内存分配器实现原理的话，那你可能写不出高性能程序，写不出高性能程序就很难参与核心项目，参与不了核心项目那么很难升职加薪，很难升级加薪就无法走向人生巅峰，没想到内存分配竟如此关键，为了走上人生巅峰你也要势必读完本文😉。

现在我们知道了，对内存分配器透彻的理解是写出高性能程序的关键所在，那么我们该怎样透彻理解内存分配器呢？

还有什么能比你自己动手实现一个理解的更透彻吗？

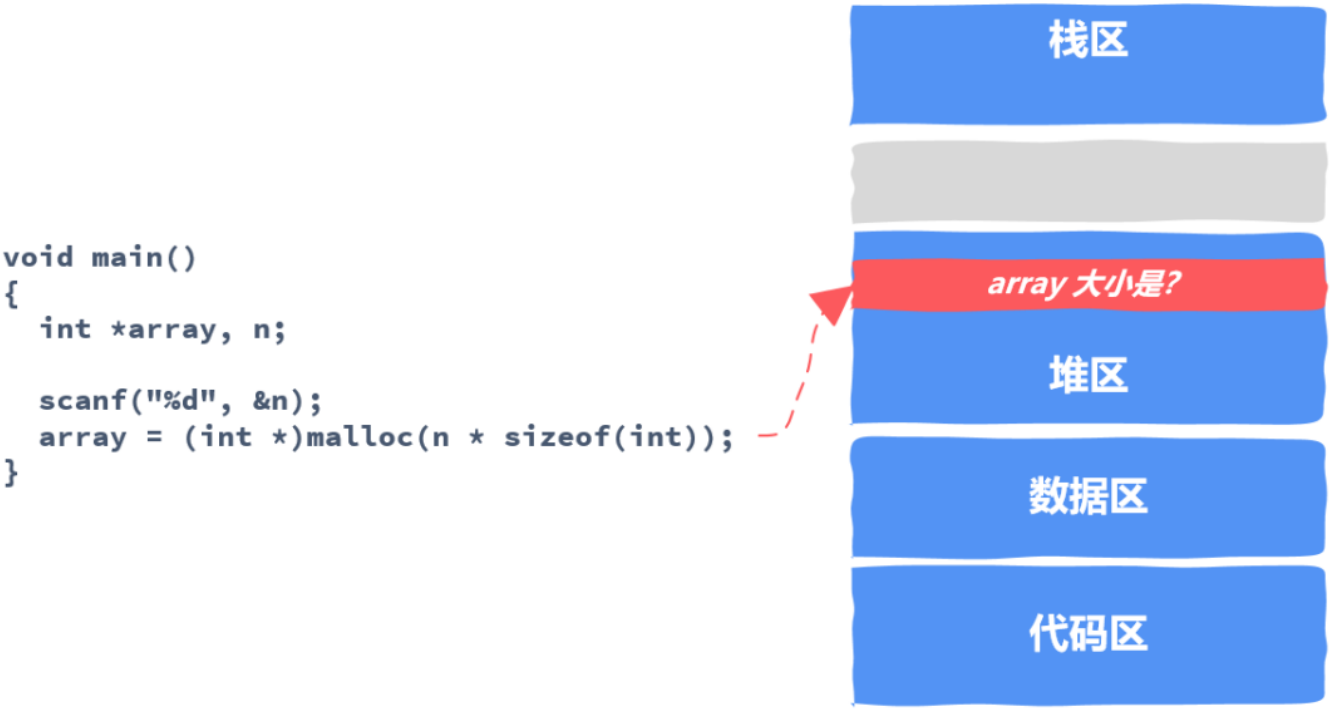
接下来，我们就自己实现一个malloc内存分配器。读完本文后内存分配对你将不再是一个神秘的黑盒。

在讲解实现原理之前，我们需要回答一个基本问题，那就是我们为什么要发明内存分配器这种东西。

内存申请与释放

程序员经常使用的内存申请方式被称为**动态内存分配**，Dynamic Memory Allocation。我们为什么需要动态的去进行内存分配与释放呢？

答案很简单，因为我们不能**提前知道程序到底需要使用多少内存**。那我们什么时候才能知道呢？答案是只有当程序真的**运行**起来后我们才知道。



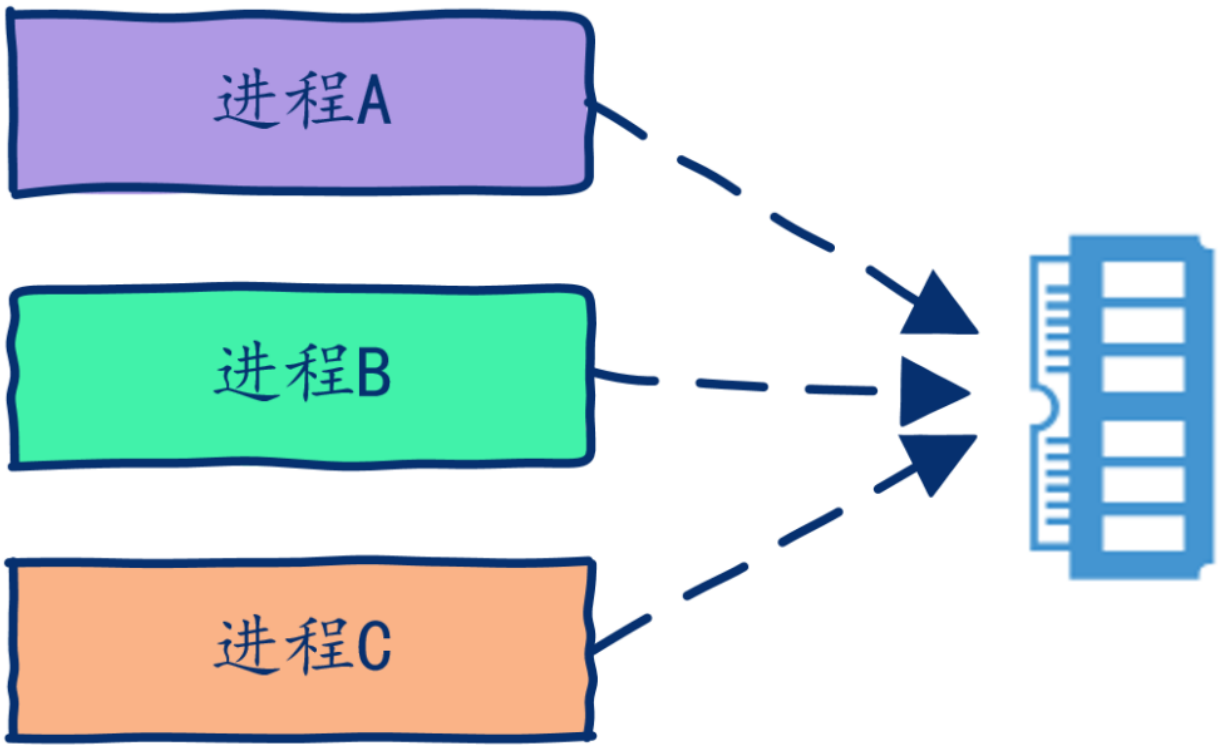
array的大小只有在程序运行起来得到用户输入后才能确定

这就是为什么程序员需要动态的去申请内存的原因，如果能提前知道我们的程序到底需要多少内存，那么直接知道告诉编译器就好了，这样也不必发明malloc等内存分配器了。

知道了为什么要发明内存分配器的原因后，接下来我们着手实现一个。

程序员应如何看待内存

实际上，现代程序员是很幸福的，程序员很少去关心内存分配的问题。作为程序员，**可以简单的认为我们的程序独占内存，注意，是独占哦。**



每个进程都认为除了自己和操作系统外没有其它程序在使用内存

写程序时你从来没有关心过如果我们的程序占用过多内存会不会影响到其它程序，我们可以简单的认为每个程序(进程)独占4G内存(32位操作系统)，即使我们的物理内存512M。不信你可以去试试，**在即使只有512M大小的内存上你依然可以申请到2G内存来使用**，可这是为什么呢？关于这个问题我们会在《深入理解操作系统》系列中详细阐述。

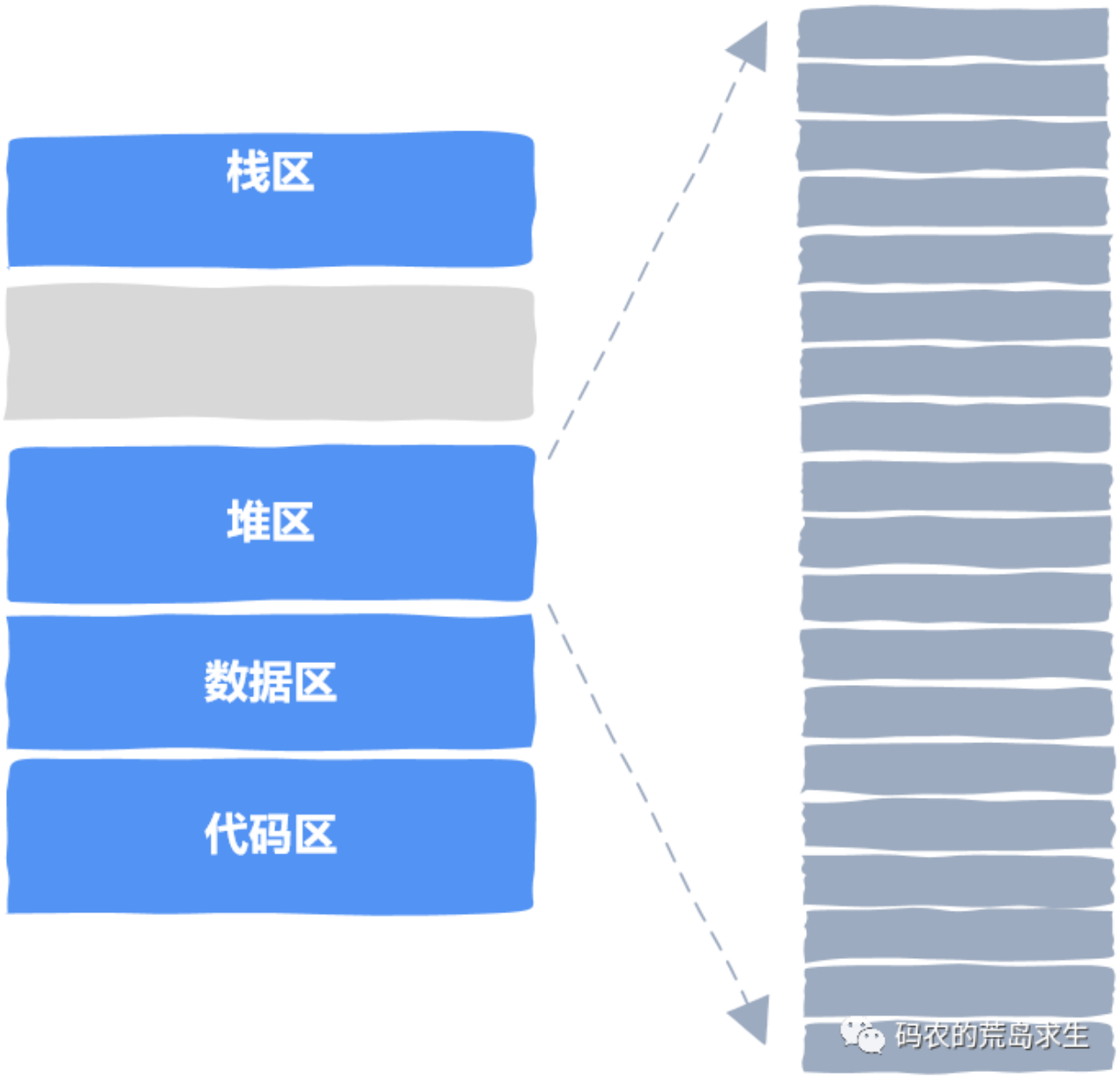
总之，程序员可以**放心的**认为我们的程序运行起来后在内存中是这样的：

作为程序员我们应该知道，内存动态申请和释放都发生在堆区，heap。

我们使用的malloc或者C++中的new申请内存时，就是从堆区这个区域中申请的。

接下来我们就要自己管理堆区这个内存区域。

堆区这个区域实际上非常简单，真的是非常简单，你可以将其看做一大数组，就像这样：



从内存分配器的角度看，**内存分配器根本不关心你是整数、浮点数、链表、二叉树等数据结构、还是对象、结构体等这些花哨的概念，在内存分配器眼里不过就是一个内存块，这些内存块中可以装入原生的字节序列**，申请者拿到该内存块后可以塑造成整数、浮点数、链表、二叉树等数据结构以及对象、结构体等，这是使用者的事情，和内存分配器无关。

我们要在这片内存上解决两个问题：

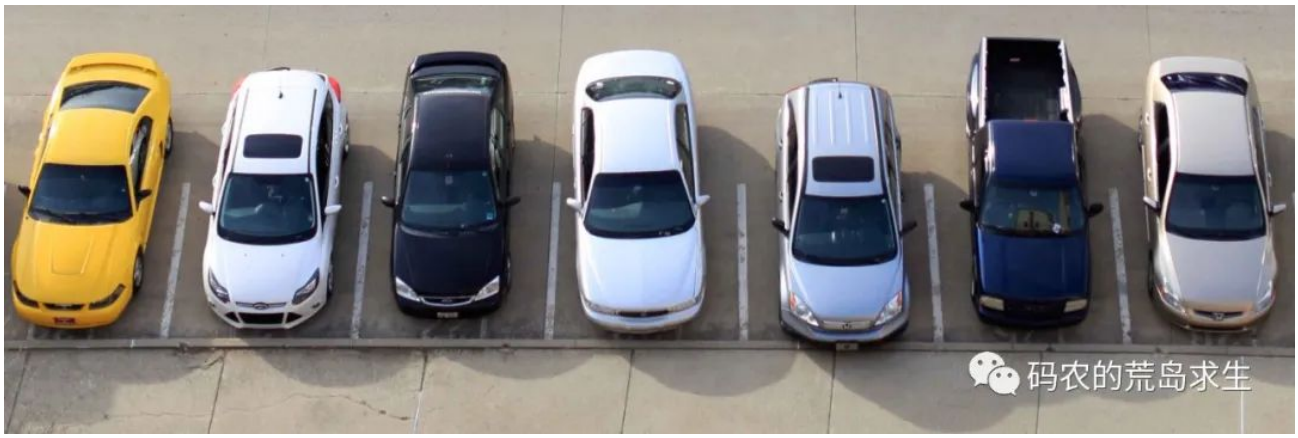
- 实现一个malloc函数，也就是如果有人向我申请一块内存，我该怎样从堆区这片区域中找到一块返回给申请者。

- 实现一个free函数，也就是当某一块内存使用完毕后，我该怎样还给堆区这片区域。

这是内存分配器要解决的两个最核心的问题，接下来我们先去停车场看看能找到什么启示。

从停车场到内存管理

实际上你可以把内存看做一条长长的**停车场**，我们申请内存就是要找到一块停车位，释放内存就是把车开走让出停车位。



只不过这个停车场比较特殊，我们不止可以停小汽车、也可以停占地面积很小的自行车以及占地面积很大的卡车，重点就是申请的内存是**大小不一**的，在这样的条件下你该怎样实现以下两个目标呢？

- 快速找到停车位，在内存申请中，这涉及到以最大速度找到一块满足要求的空闲内存
- 尽最大程度利用停车场，我们的停车场应该能停尽可能多的车，在内存申请中，这涉及到在给定条件下尽可能多的满足内存申请需求

现在，我们已经清楚的理解任务了，那么该怎么实现呢？

任务拆分

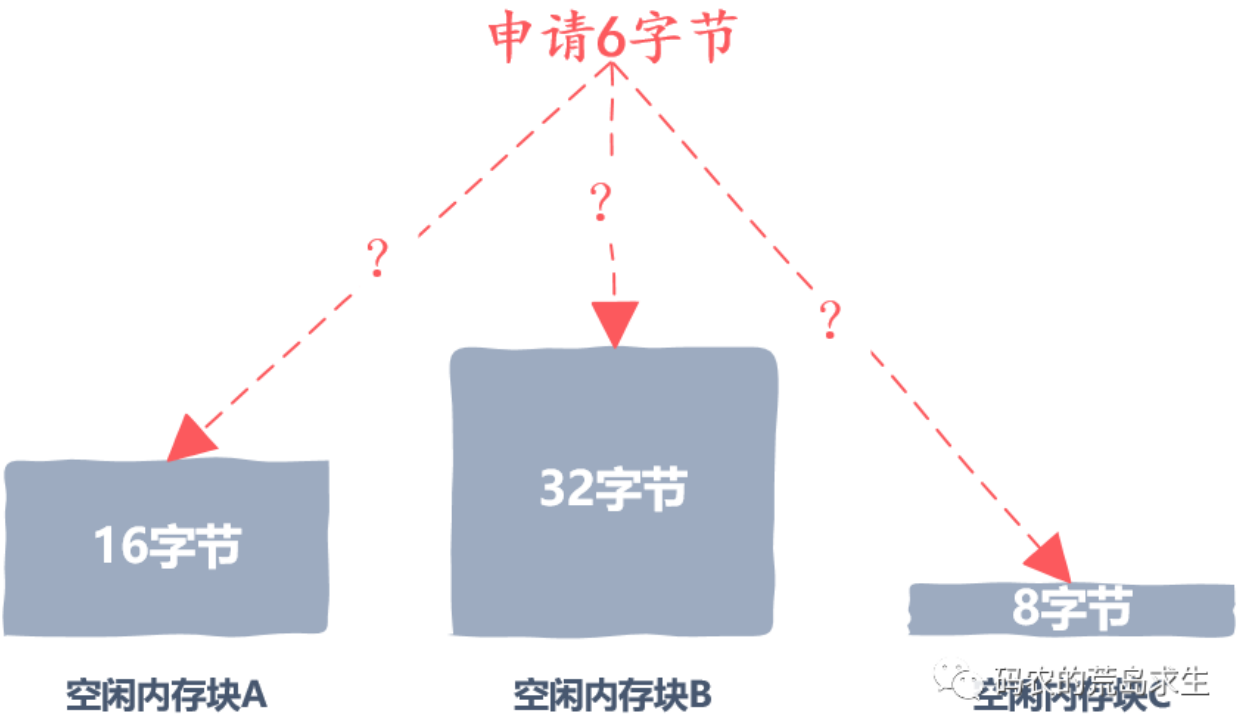
现在我们已经明确要实现什么以及衡量其好坏的标准，接下来我们就要去设计实现细节了，让我们把任务拆分一下，怎么拆分呢？

我们可以自己想一下从内存的申请到释放需要哪些细节。

申请内存时，我们需要在内存中找到一块大小合适的空闲内存分配出去，那么**我们怎么知道有哪些内存块是空闲的呢**？

因此，第一个实现细节出现了，**我们需要把内存块用某种方式组织起来，这样我们才能追踪到每一块内存的分配状态。**

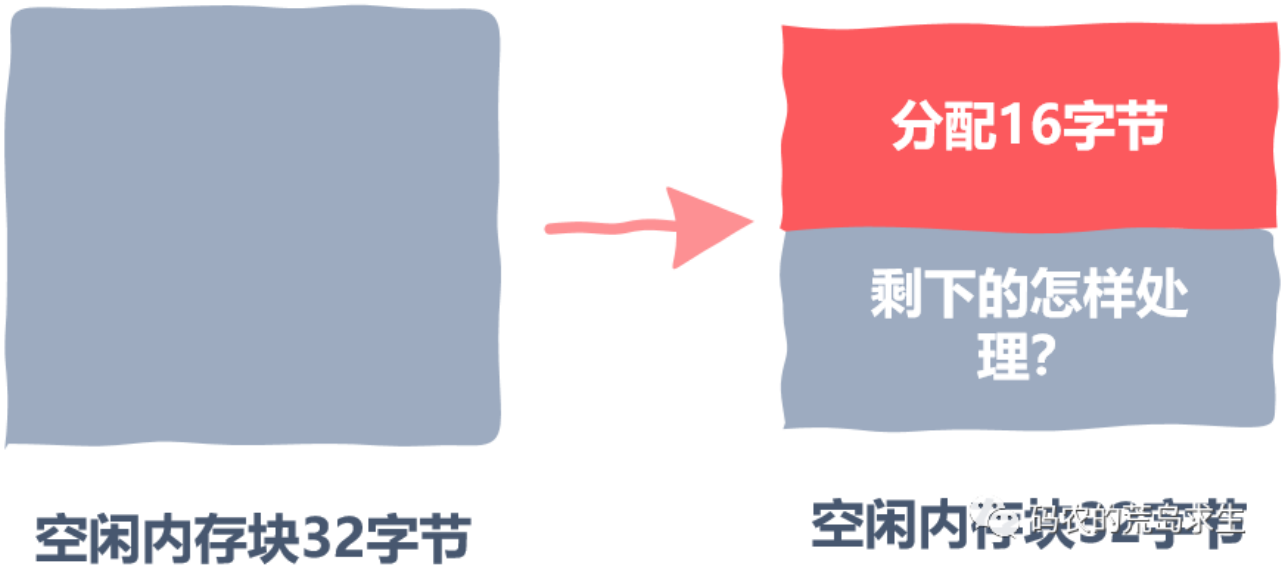
现在空闲内存块组织好了，那么一次内存申请可能有很多空闲内存块满足要求，那么我们该选择哪一个空闲内存块分配给用户呢？



因此，第二个实现细节出现了，**我们该选择什么样的空闲内存块给到用户。**

接下来我们找到了一块大小合适的内存块，假设用户需要16个字节，而我们找到的这块空闲内存块大小为32字节，那么将16字节分配给用户后还剩下16字节，这剩下的内存该怎么处理呢？

因此，第三个实现细节出现了，分配出去内存后，**空闲内存块剩余的空间该怎么处理**？



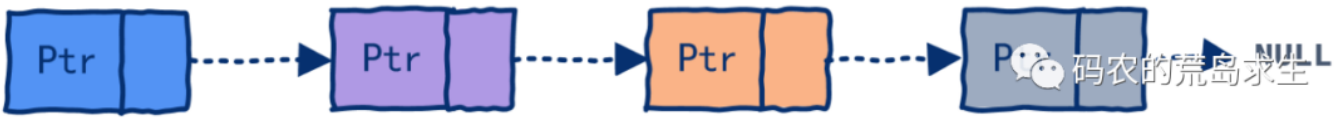
最后，分配给用户的内存使用完毕，这是第四个细节出现了，**我们该怎么处理用户还给我们的内存呢**？

以上四个问题是任何一个内存分配器必须要回答的，接下来我们就一一解决这些问题，解决完这些问题后一个崭新的内存分配器就诞生啦。

管理空闲内存块

空闲内存块的本质是需要某种办法来来区分哪些是空闲内存哪些是已经分配出去的内存。

有的同学可能会说，这还不简单吗，用一个链表之类的结构记录下每个空闲内存块的开始和结尾不就可以了，这句话也对也不对。



说不对，是因为如果要申请内存来创建这个链表那么这就是不对的，原因很简单，因为创建链表不可避免的要申请内存，申请内存就需要通过内存分配器，可是你要实现的就是一个内存分配器，**你没有办法向一个还没有实现的内存分配器申请内存。**

说对也对，我们确实需要一个类似链表这样的结构来维护空闲内存块，但这个链表并不是我们常见的那种。

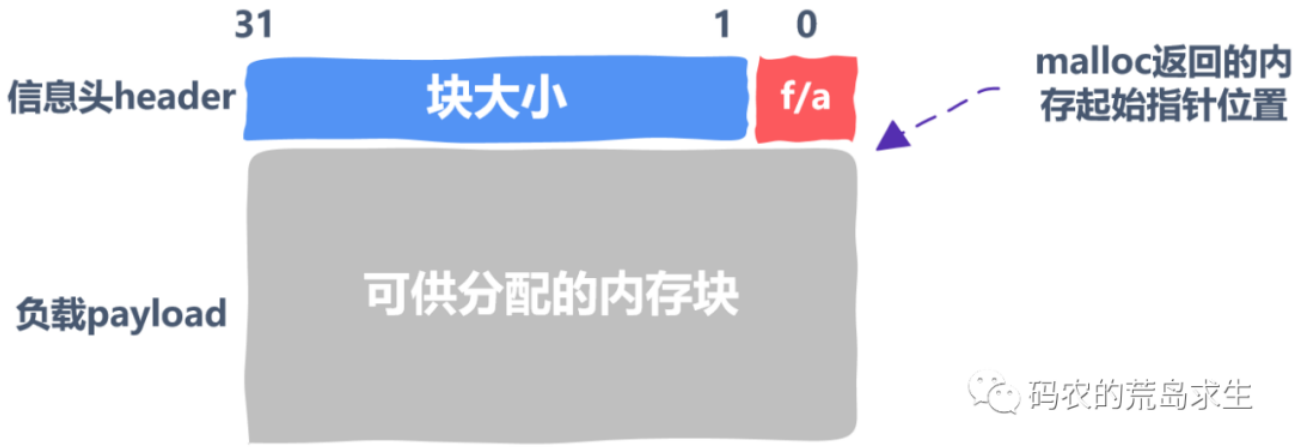
因为我们无法将空闲内存块的信息保存在其它地方，那么没有办法，**我们只能将维护内存块的分配信息保存在内存块本身中**，这也是大多数内存分配器的实现方法。

那么，为了维护内存块分配状态，我们需要知道哪些信息呢？很简单：

- 一个标记，用来标识该内存块是否空闲
- 一个数字，用来记录该内存块的大小

为了简单起见，我们的内存分配器不对内存对齐有要求，同时一次内存申请允许的最大内存块为2G，**注意，这些假设是为了方便讲解内存分配器的实现而屏蔽一些细节，我们常用的malloc等不会有这样的限制。**

因为我们的内存块大小上限为2G，因此我们可以使用31个比特位来记录块大小，剩下的一个比特位用来标识该内存块是空闲的还是已经被分配出去了，下图中的f/a是free/allocate，也就是标记是已经分配出去还是空闲的。这32个比特位就是header，用来存储块信息。

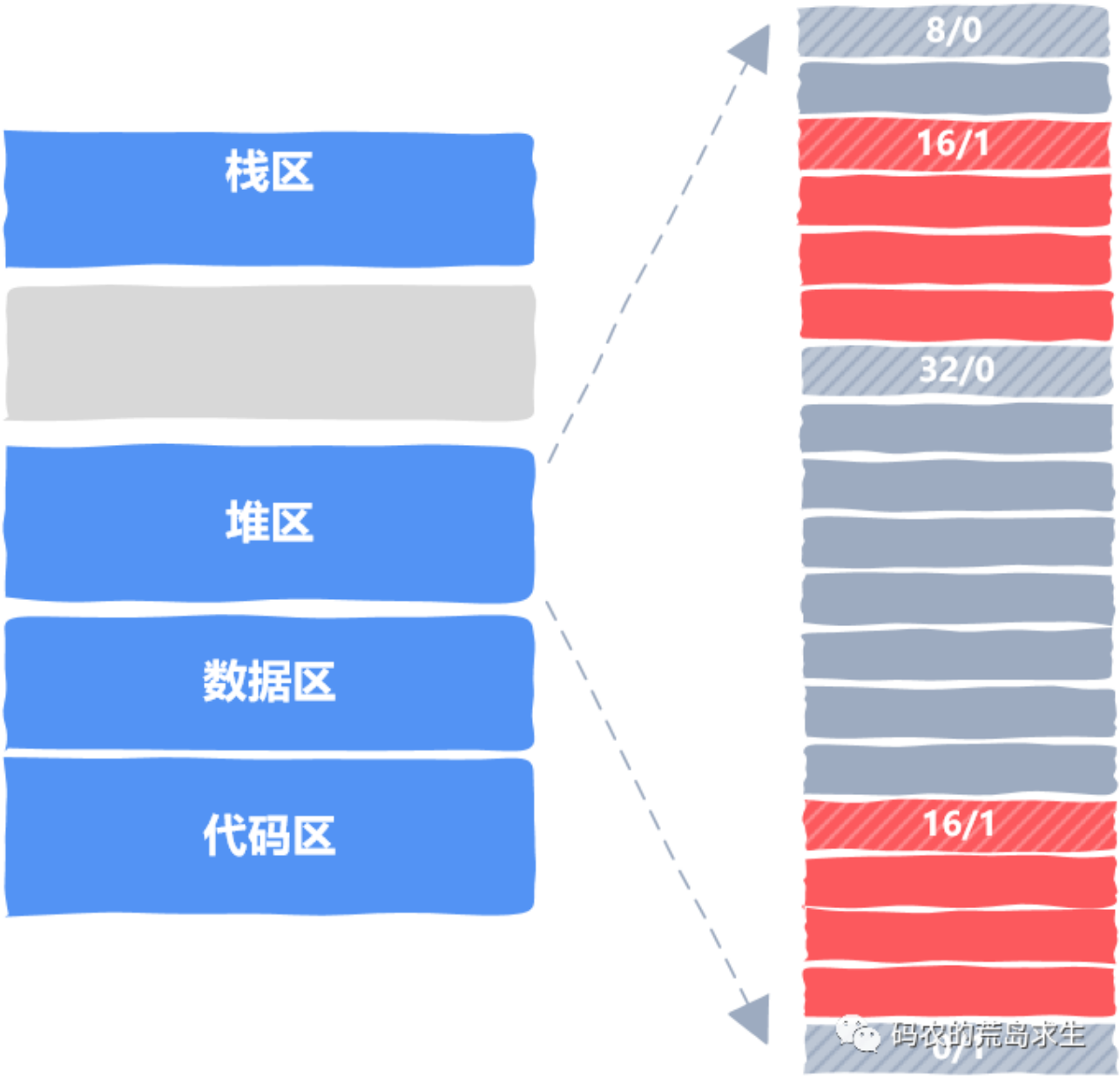


剩下的灰色部分才是真正可以分配给用户的内存，这一部分也被称为负载，payload，**我们调用malloc返回的内存起始地址正是这块内存的起始地址。**

现在你应该知道了吧，不是说堆上有10G内存，这里面就可以全部用来存储数据的，这里面必然有一部分要拿出来维护内存块的一些信息，就像这里的header一样。

跟踪内存分配状态

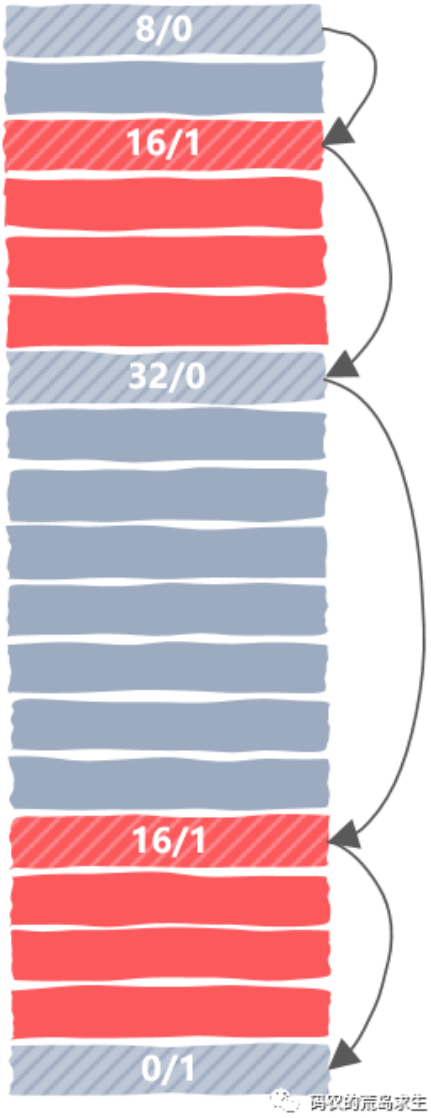
有了上图，**我们就可以将堆这块内存区域组织起来并进行内存分配与释放了**，如图所示：



在这里我们的堆区还很小，每一方框代表4字节，其中红色区域表示已经分配出去的，灰色区域表示空闲内存，每一块内存都有一个header，用带斜线的方框表示，比如16/1，就表示该内存块大小是16字节，1表示已经分配出去了；而32/0表示该内存块大小是32字节，0表示该内存块当前空闲。

细心的同学可能会问，那最后一个方框0/1表示什么呢？原来，我们需要某种特殊标记来告诉我们的内存分配器是不是已经到末尾了，这就是最后4字节的作用。

通过引入header我们就能知道每一个内存块的大小，从而可以很方便的遍历整个堆区。遍历方法很简单，因为我们知道每一块的大小，那么从当前的位置加上当前块的大小就是下一个内存块的起始位置，如图所示：

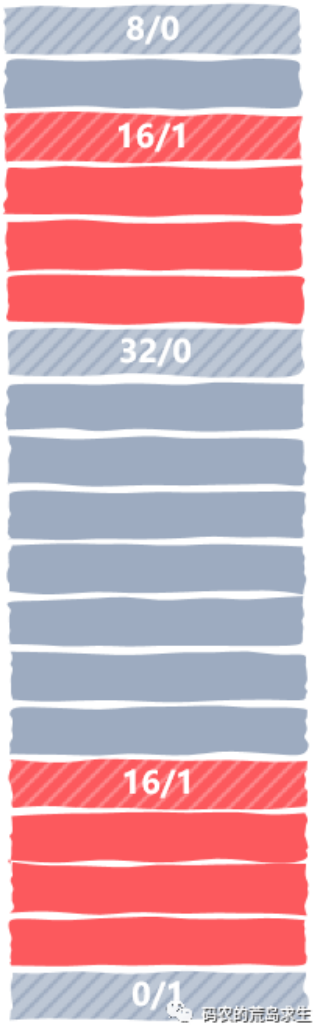


通过每一个header的最后一个bit位就能知道每一块内存是空闲的还是已经分配出去了，这样我们就能追踪到每一个内存块的分配信息，因此上文提到的第一个问题解决了。

接下来我们看第二个问题。

怎样选择空闲内存块

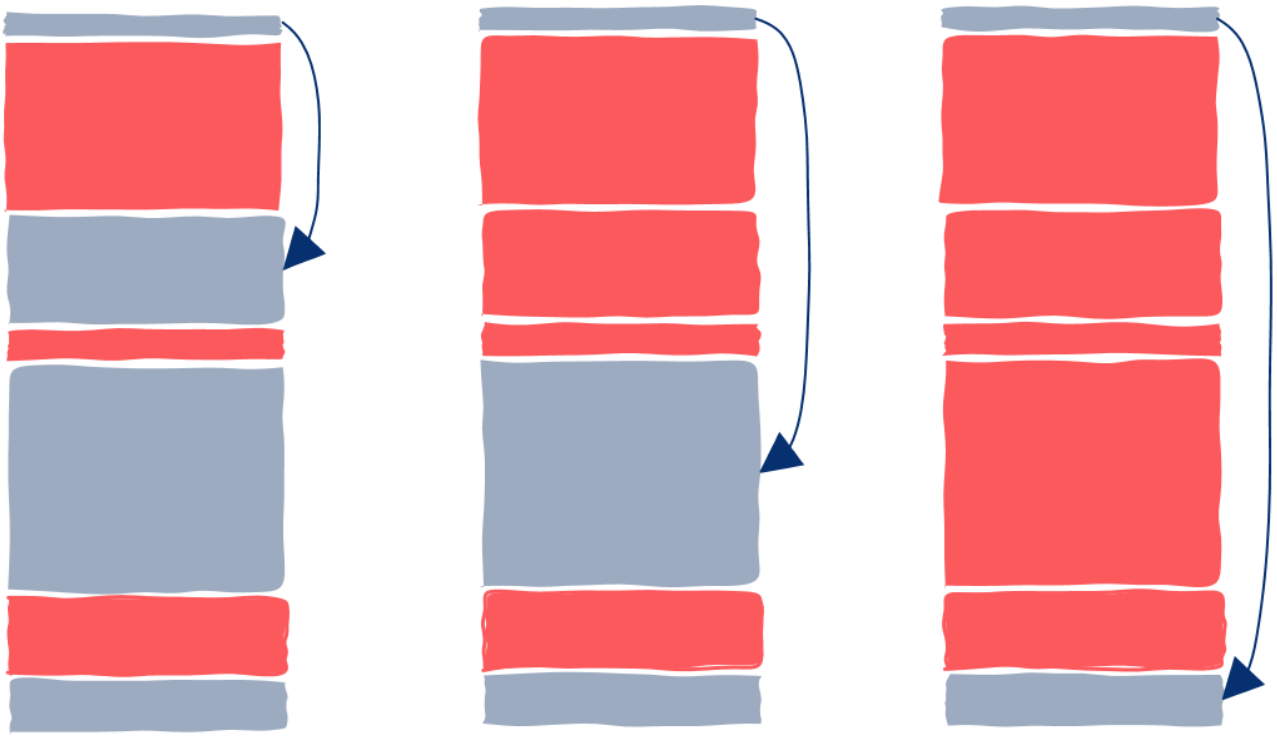
当应用程序调用我们实现的malloc时，内存分配器需要遍历整个空闲内存块找到一块能满足应用程序要求的内存块返回，就像下图这样：



假设应用程序需要申请4字节内存，从图中我们可以看到有两个空闲内存块满足要求，第一个大小为8字节的内存块和第三个大小为32字节的内存块，那么我们到底该选择哪一个返回呢？这就涉及到了分配策略的问题，实际上这里有很多的策略可供选择。

First Fit

最简单的就是**每次从头开始找起**，找到第一个满足要求的就返回，这就是所谓的First fit方法，教科书中一般称为首次适应方法，当然我们不需要记住这样拗口的名字，只需要记住这是什么意思就可以了。

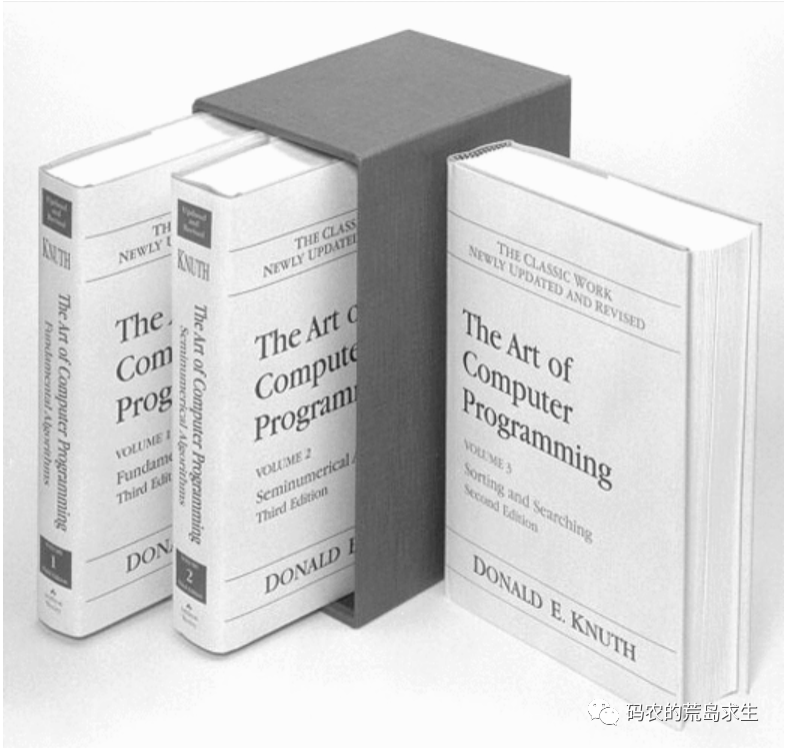


First Fit总是从头开始找到第一个满足要求的空闲内存块

这种方法的优势在于**简单**，但该策略总是从前面的空闲块找起，因此很容易在堆区前半部分因分配出内存留下很多小的内存块，因此下一次内存申请搜索的空闲块数量将会越来越多。

Next Fit

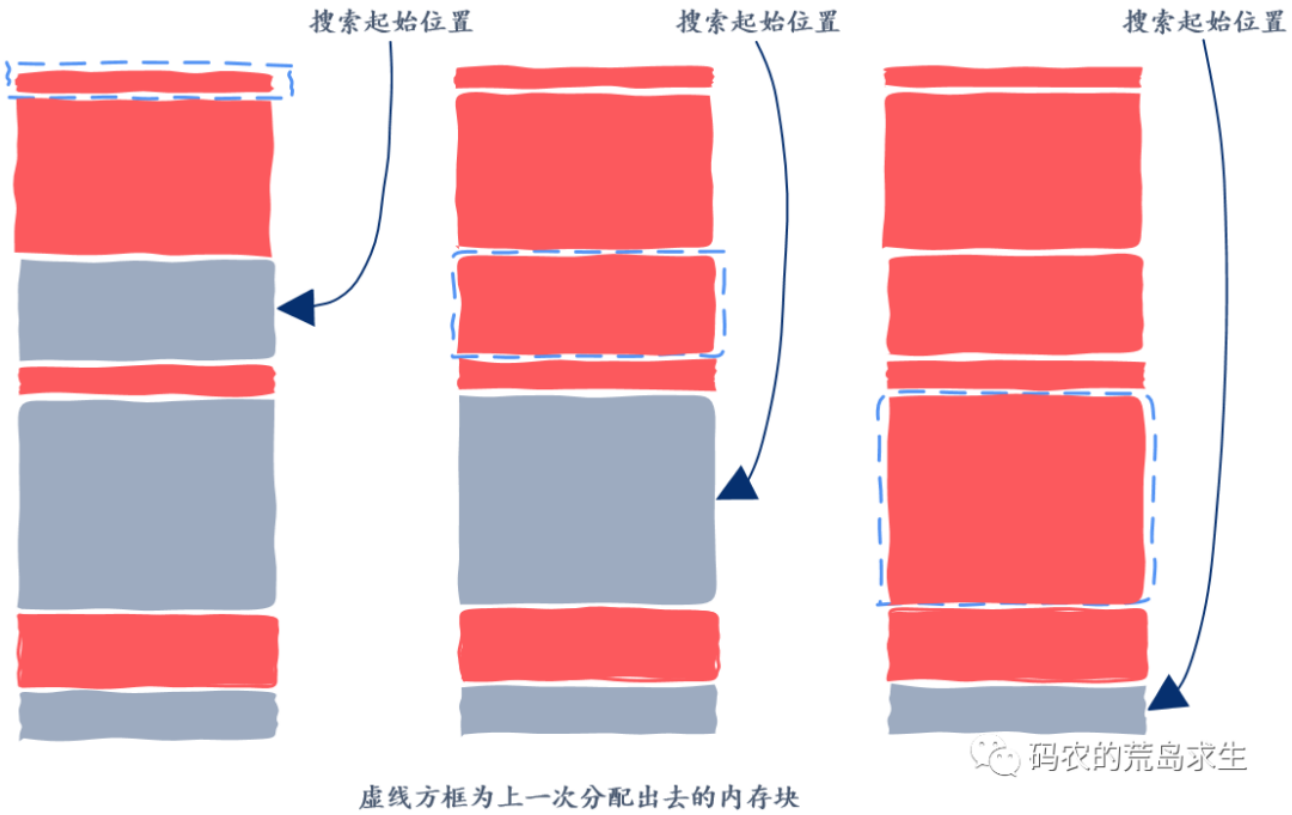
该方法是大名鼎鼎的Donald Knuth首次提出来的，如果你不知道谁是Donald Knuth，那么数据结构课上折磨的你痛不欲生的字符串匹配KMP算法你一定不会错过，KMP其中的K就是指Donald Knuth，该算法全称**Knuth–Morris–Pratt string-searching algorithm**，如果你也没听过KMP算法那么你一定听过下面这本书：



这就是更加大名鼎鼎的《计算机程序设计艺术》，这本书就是Donald Knuth写的，如果你没有听过这本书请面壁思过一分钟，比尔盖茨曾经说过，如果你看懂了这本书就去给微软投简历吧，这本书也是很多程序员买回来后从来不会翻一眼只是拿来当做镇宅之宝用的。

不止比尔盖茨，有一次乔布斯见到Knuth老爷子后。。算了，扯远了，有机会再和大家讲这个故事，拉回来。

Next Fit说的是什么呢？这个策略和First Fit很相似，是说我们别总是从头开始找了，而是从上一次找到合适的空闲内存块的位置找起，老爷子观察到上一次找到某个合适的内存块的地方很有可能剩下的内存块能满足接下来的内存分配请求，由于不需要从头开始搜索，因此**Next Fit将远快于First Fit。**

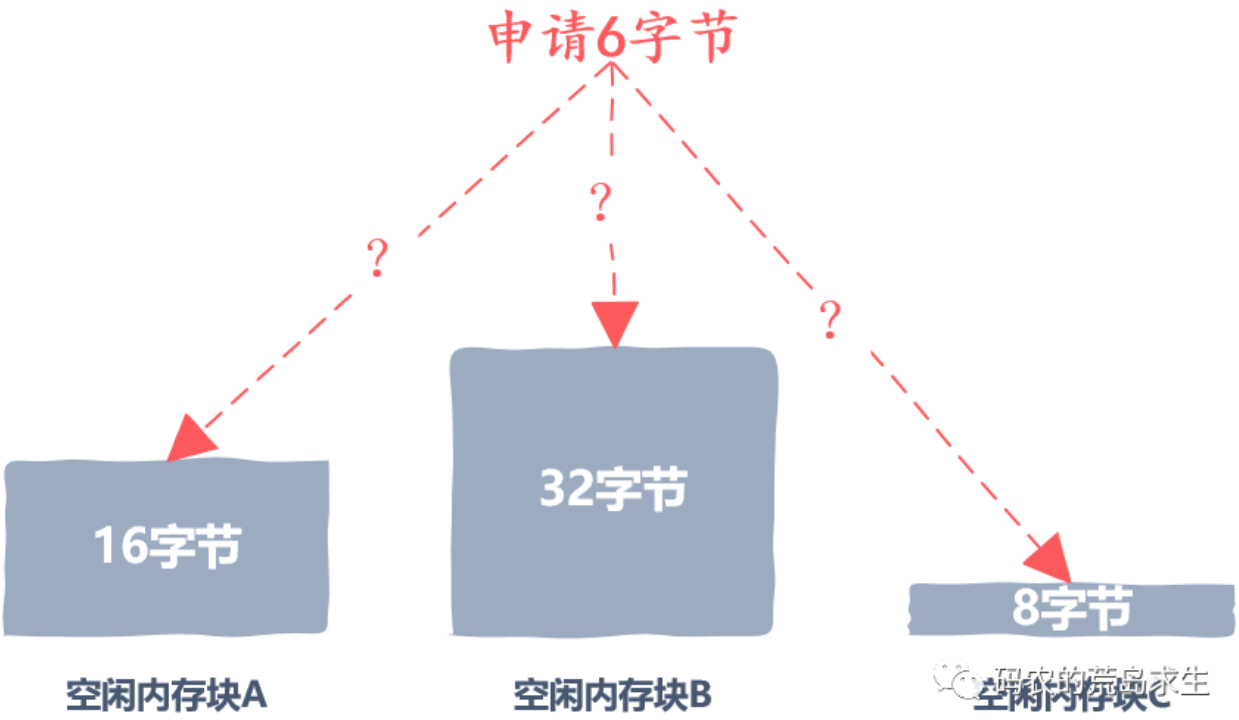


然而也有研究表明Next Fit方法内存使用率不及First Fit，也就是同样的停车场面积，First Fit方法能停更多的车。

Best Fit

First Fit和Next Fit都是找到第一个满足要求的内存块就返回，但Best Fit不是这样。

Best Fit算法会找到所有的空闲内存块，然后将所有满足要求的并且大小为最小的那个空闲内存块返回，这样的空闲内存块才是最Best的，因此被称为Best Fit。就像下图虽然有三个空闲内存块满足要求，但是Best Fit会选择大小为8字节的空闲内存块。



显然，从直觉上我们就能得出Best Fit会比前两种方法能更合理利用内存的结论，各项研究也证实了这一点。

然而Best Fit最大的缺点就是分配内存时需要遍历堆上所有的空闲内存块，在速度上显然不及前面两种方法。

以上介绍的这三种策略在各种内存分配器中非常常见，当然分配策略远不止这几种，但这些算法不是该主题下关注的重点，因此就不在这里详细阐述了，假设在这里我们选择First Fit算法。

没有银弹

重要的是，从上面的介绍中我们能够看到，**没有一种完美的策略**，每一种策略都有其优点和缺点，**我们能做到的只有取舍和权衡。**因此，要实现一个内存分配器，设计空间其实是非常大的，要想设计出一个通用的内存分配器，就像我们常用的malloc是很不容易的。

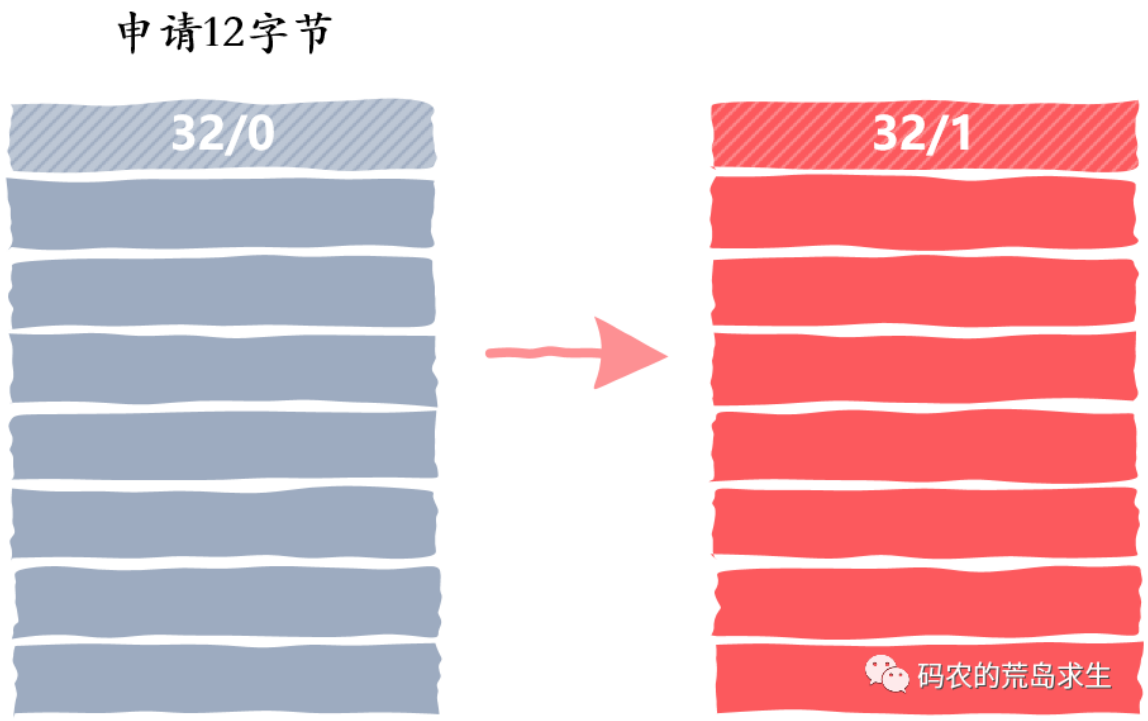
其实不止内存分配器，在设计其它软件系统时我们也没有银弹。

分配内存

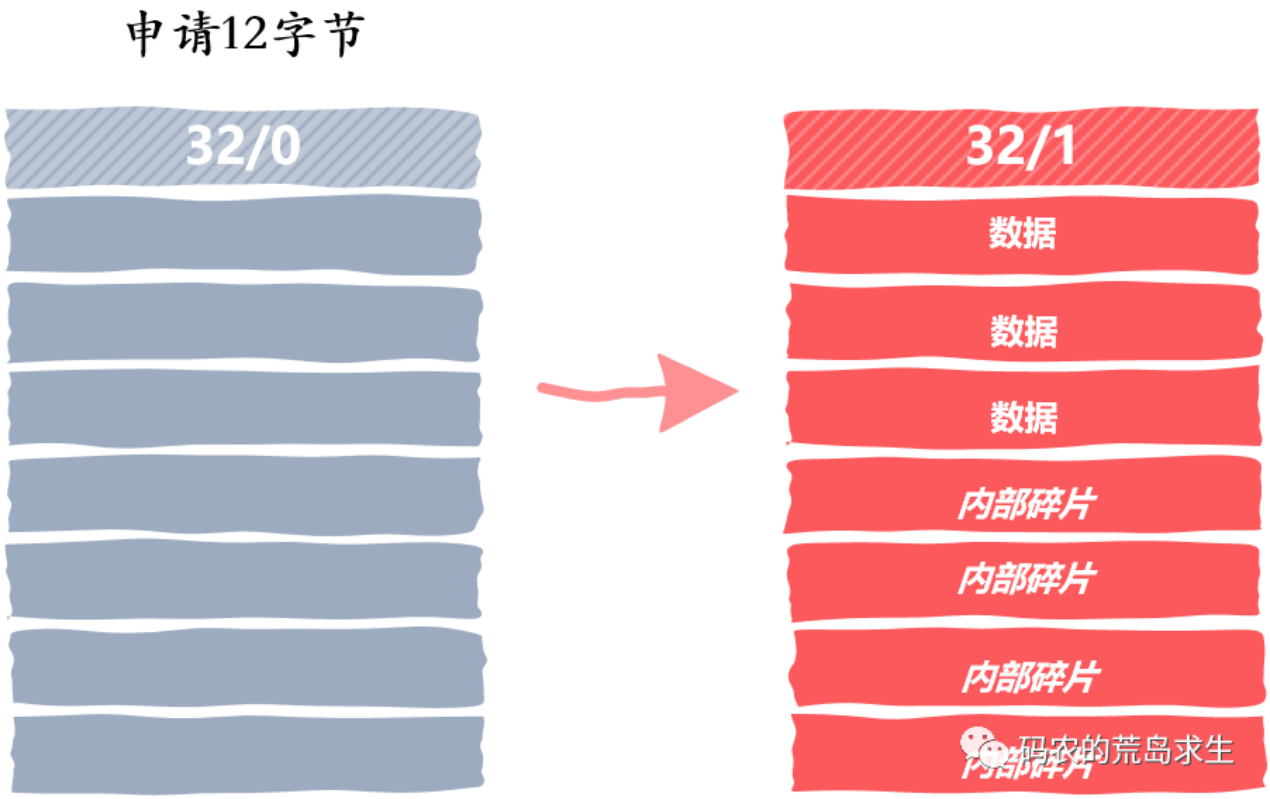
现在我们找到合适的空闲内存块了，接下来我们又将面临一个新的问题。

如果用户需要12字节，而我们的空闲内存块也恰好是12字节，那么很好，直接返回就可以了。

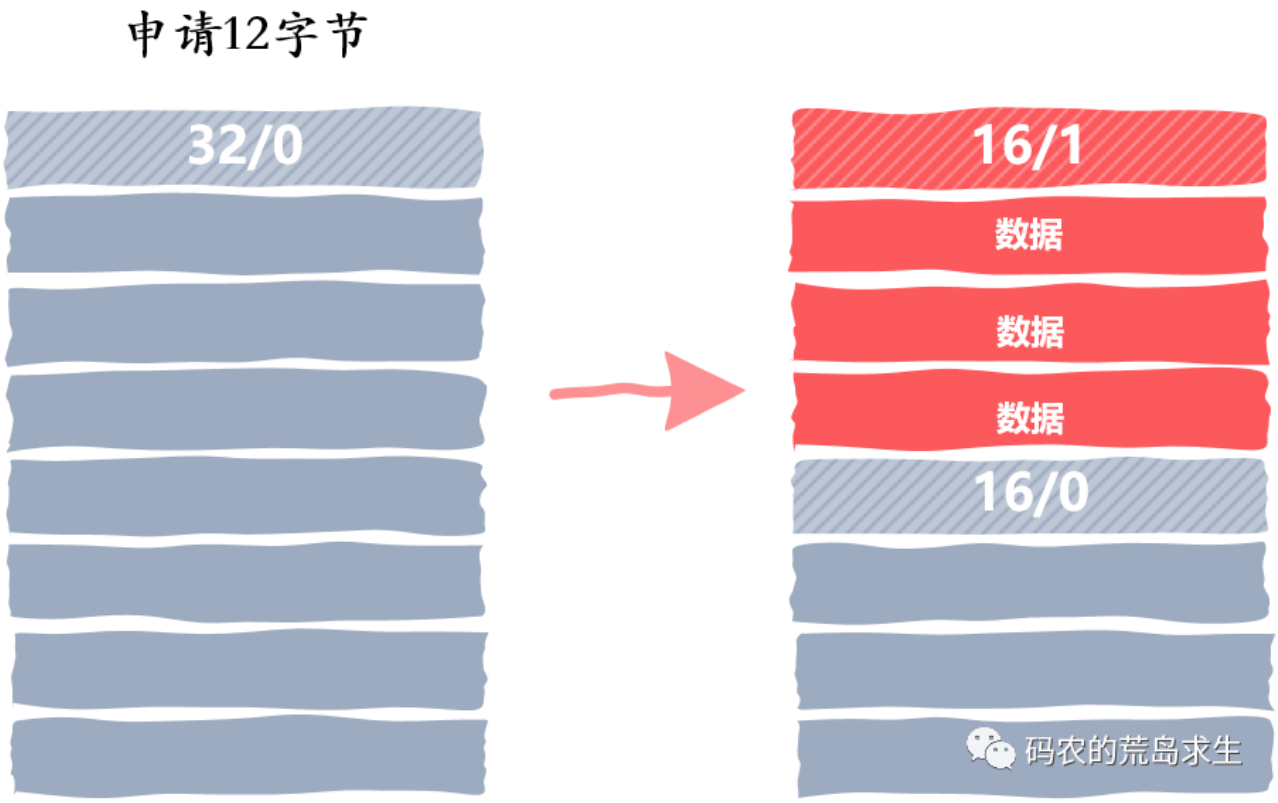
但是，如果用户申请12字节内存，而我们找到的空闲内存块大小为32字节，那么我们是要将这32字节的整个空闲内存块标记为已分配吗？就像这样：



这样虽然速度最快，但显然会浪费内存，形成**内部碎片**，也就是说该内存块剩下的空间将无法被利用到。



一种显而易见的方法就是将空闲内存块进行划分，前一部分设置为已分配，返回给内存申请者使用，后一部分变为一个新的空闲内存块，只不过大小会更小而已，就像这样：



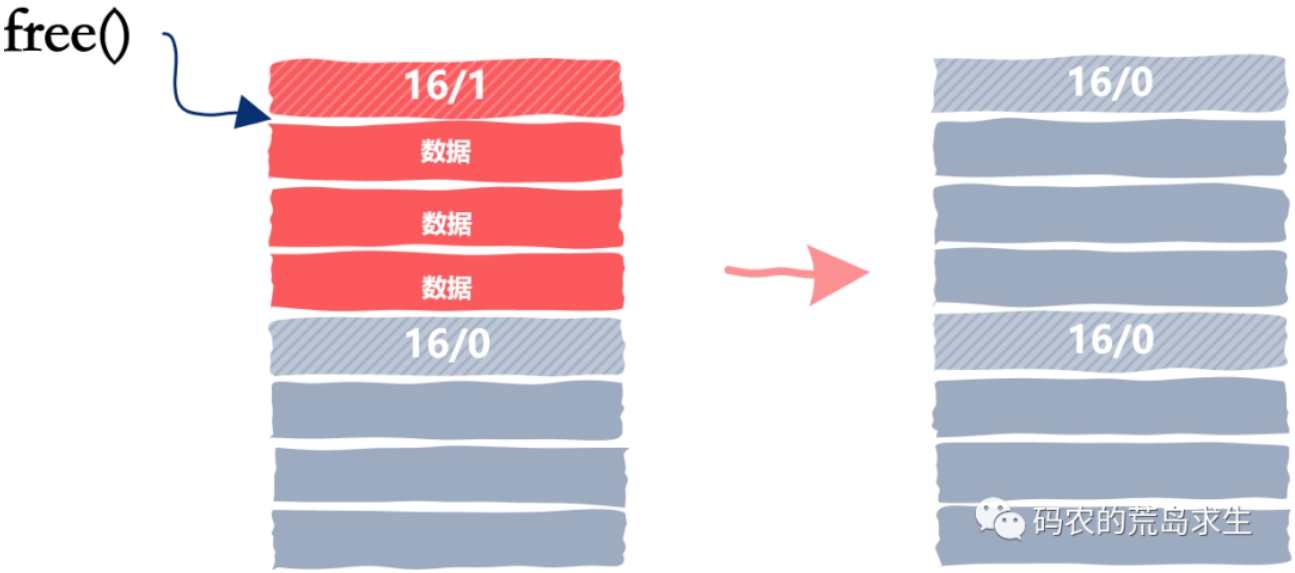
我们需要将空闲内存块大小从32修改为16，其中消息头header占据4字节，剩下的12字节分配出去，并将标记为置为1，表示该内存块已分配。

分配出16字节后，还剩下16字节，我们需要拿出4字节作为新的header并将其标记为空闲内存块。

释放内存

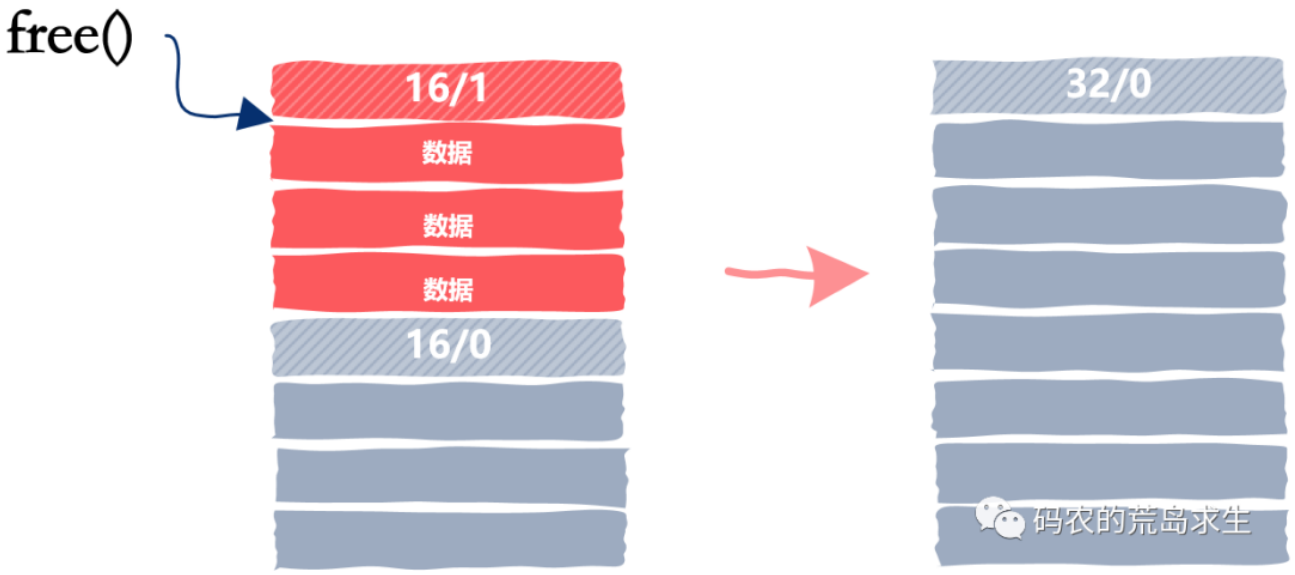
到目前为止，我们的malloc已经能够处理内存分配请求了，还差最后的内存释放。

内存释放和我们想象的不太一样，该过程并不比前几个环节简单。我们要考虑到的关键一点就在于，与被释放的内存块相邻的内存块可能也是空闲的。如果释放一块内存后我们仅仅简单的将其标志位置为空闲，那么可能会出现下面的场景：



从图中我们可以看到，被释放内存的下一个内存块也是空闲的，如果我们仅仅将这16个字节的内存块标记为空闲的话，那么当下一次申请20字节时图中的这两个内存块都不能满足要求，尽管这两个空闲内存块的总数要超过20字节。

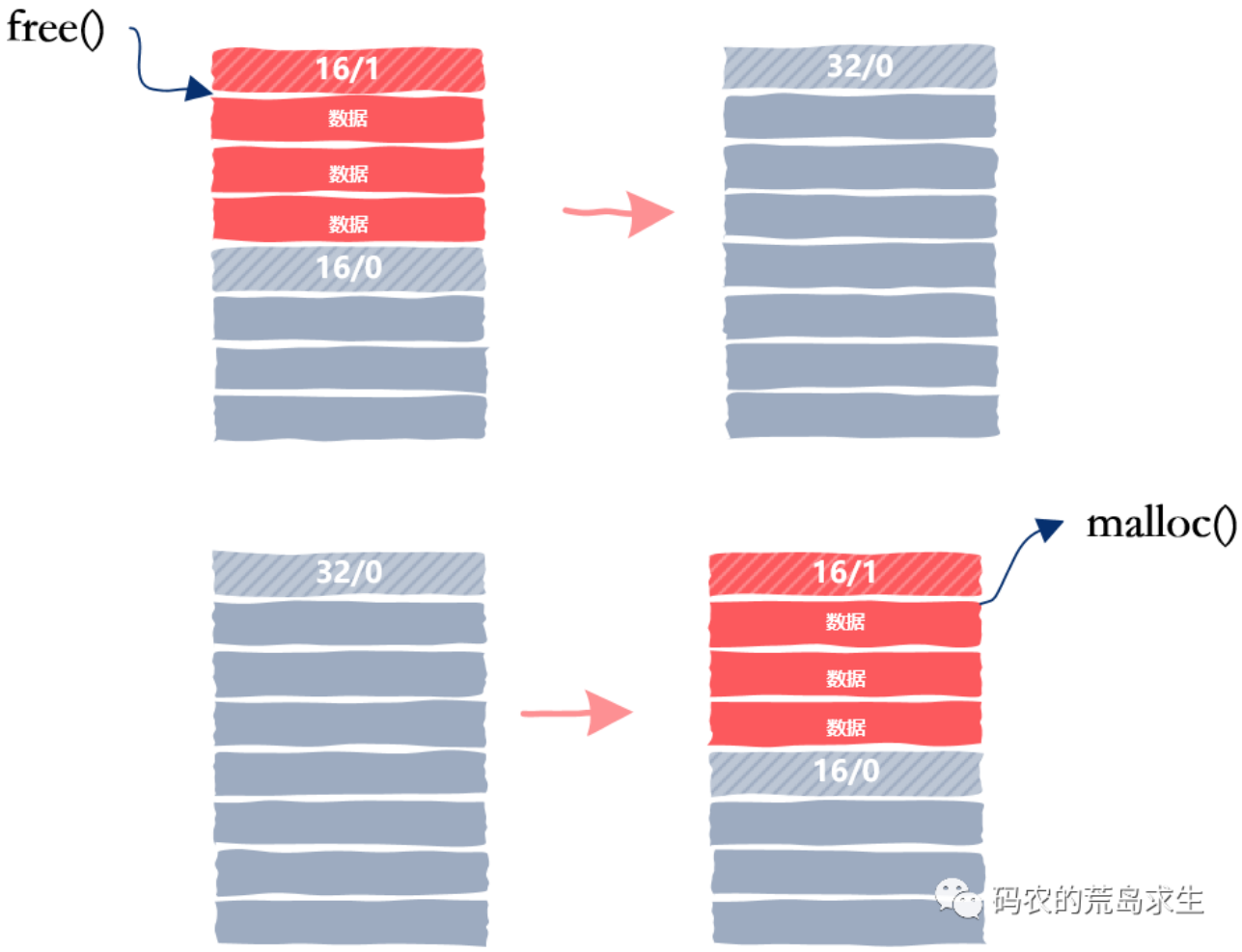
因此一种更好的方法是当应用程序向我们的malloc释放内存时，我们查看一下相邻的内存块是否是空闲的，**如果是空闲的话我们需要合并空闲内存块**，就像这样：



在这里我们又面临一个新的决策，那就是释放内存时我们要立即去检查能否够合并相邻空闲内存块吗？还是说我们可以推迟一段时间，推迟到下一次分配内存找不到满足要的空闲内存块时再合并相邻空闲内存块。

释放内存时立即合并空闲内存块相对简单，但每次释放内存时将引入合并内存块的开销，如果应用程序总是释放12字节然后申请12字节，然后在释放12字节等等这样重复的模式：

```
1   free(ptr);
2   obj* ptr = malloc(12);
3   free(ptr);
4   obj* ptr = malloc(12);
5   ...
```



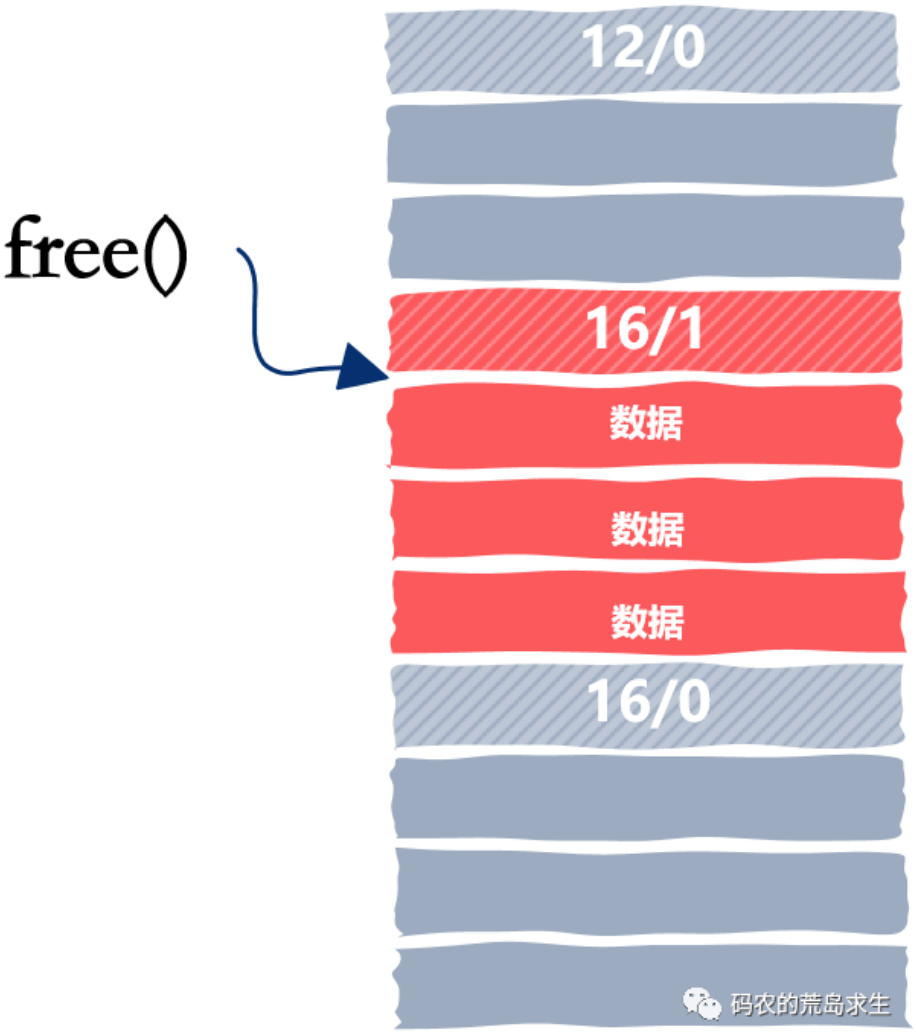
那么这种内存使用模式对立即合并空闲内存块这种策略非常不友好，我们的内存分配器会有很多的**无用功**。但这种策略最为简单，在这里我们依然选择使用这种简单的策略。

实际上我们需要意识到，实际使用的内存分配器都会有某种推迟合并空闲内存块的策略。

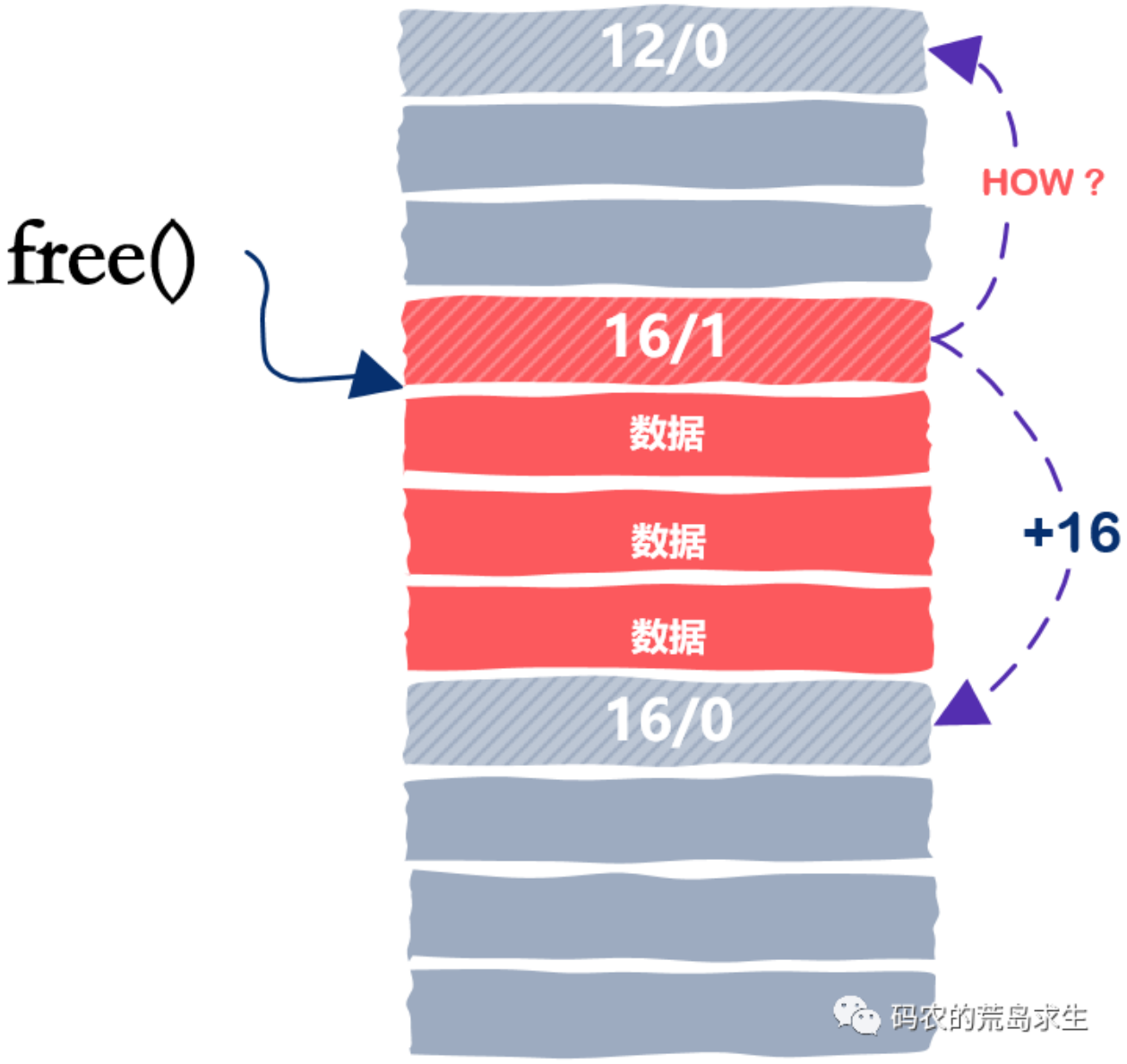
高效合并空闲内存块

合并空闲内存块的故事到这里就完了吗？问题没有那么简单。

让我们来看这样一个场景：



使用的内存块其前和其后都是空闲的，在当前的设计中我们可以很容易的知道后一个内存块是空闲的，因为我们只需要从当前位置向下移动16字节就是下一个内存块，但我们怎么能知道上一个内存块是不是空闲的呢？

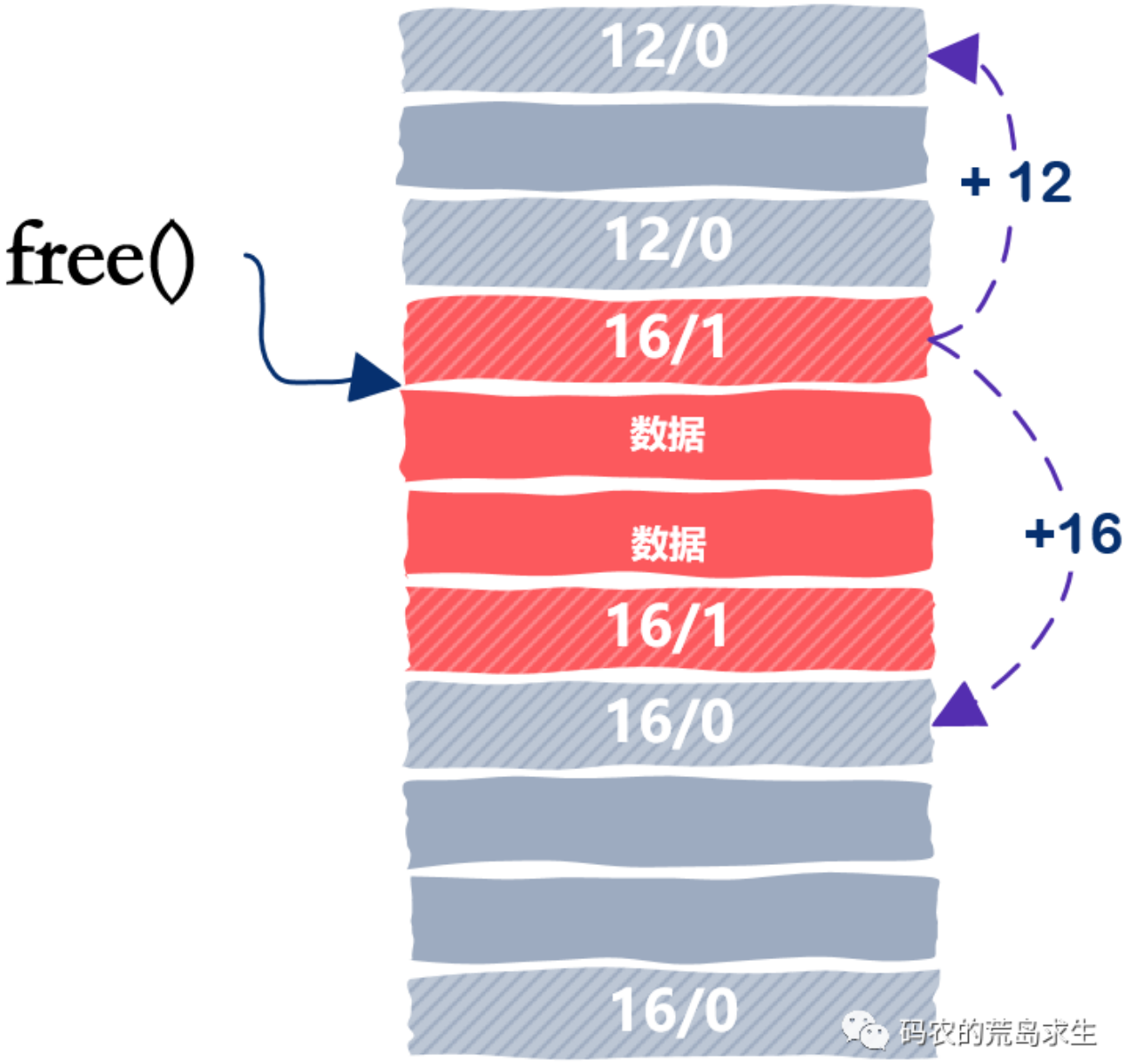


我们之所以能向后跳是因为当前内存块的大小是知道的，那么我们该怎么向前跳找到上一个内存块呢？

还是我们上文提到的Donald Knuth，老爷子提出了一个很聪明的设计，我们之所以不能往前跳是因为不知道前一个内存块的信息，那么我们该怎么快速知道前一个内存块的信息呢？

Knuth老爷子的设计是这样的，我们不是有一个信息头header吗，那么我们就在该内存块的末尾再加一个信息尾，footer，footer一词用的很形象，**header和footer的内容是一样的。**

因为上一内存块的footer和下一个内存块的header是相邻的，**因此我们只需要在当前内存块的位置向上移动4直接就可以等到上一个内存块的信息**，这样当我们释放内存时就可以快速的进行相邻空闲内存块的合并了。



收工

至此，我们的内存分配器就已经设计完毕了。

我们的简单内存分配器采用了First Fit分配算法；找到一个满足要求的内存块后会进行切分，剩下的作为新的内存块；同时当释放内存时会立即合并相邻的空闲内存块，同时为加快合并速度，我们引入了Donald Knuth的设计方法，为每个内存块增加footer信息。

这样，我们自己实现的内存分配就可以运行起来了，**可以真正的申请和释放内存。**

总结

本文从0到1实现了一个简单的内存分配器，但不希望这里的阐述给大家留下内存分配器实现很简单的印象，实际上本文实现的内存分配器还有大量的优化空间，同时我们也没有考虑线程安全问题，但这些都不是本文的目的。

本文的目的在于把内存分配器的本质告诉大家，对于想理解内存分配器实现原理的同学来说这些已经足够了，而对于要编写高性能程序的同学来说实现自己的内存池是必不可少的，内存池实现也离不开这里的讨论。

希望本文对大家理解内存分配器有帮助。

最后的最后，如果觉得文章对你有帮助的话，请多多**分享**、**转发**、**在看**。

长按关注码农的荒岛求生