

# 18张图揭秘高性能Linux服务器内存池技术是如何实现的

Original 码农的荒岛求生 码农的荒岛求生 2021-01-26 16:15

收录于合集

#高并发&高性能 24 #Linux 3 #程序员 21 #计算机内功 15 #内存 7

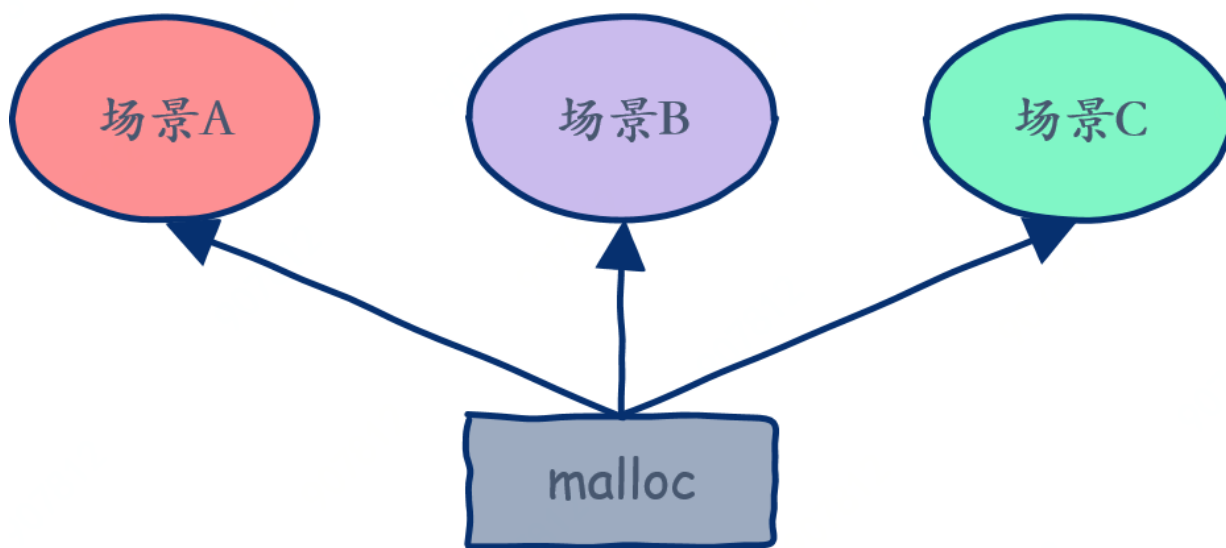
大家生活中肯定都有这样的经验，那就是大众化的产品都比较便宜，但便宜的大众产品就是一个词，普通；而可以定制的产品一般都价位不凡，这种定制的产品注定不会在大众中普及，因此定制产品就是一个词，独特。

有的同学可能会有疑问，你不是要聊技术吗？怎么又说起消费了？

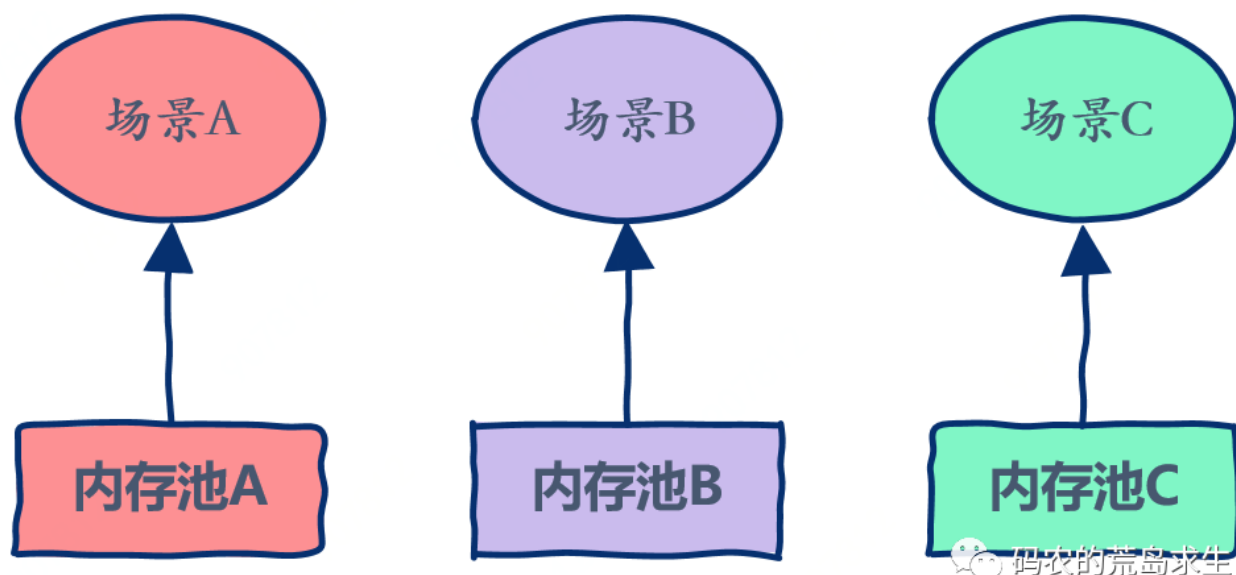
原来技术也有大众货以及定制品。

## 通用 VS 定制

作为程序员(C/C++)我们知道申请内存使用的是malloc，malloc其实就是一个通用的大众货，什么场景下都可以用，**但是什么场景下都可以用就意味着什么场景下都不会有很高的性能。**



VS



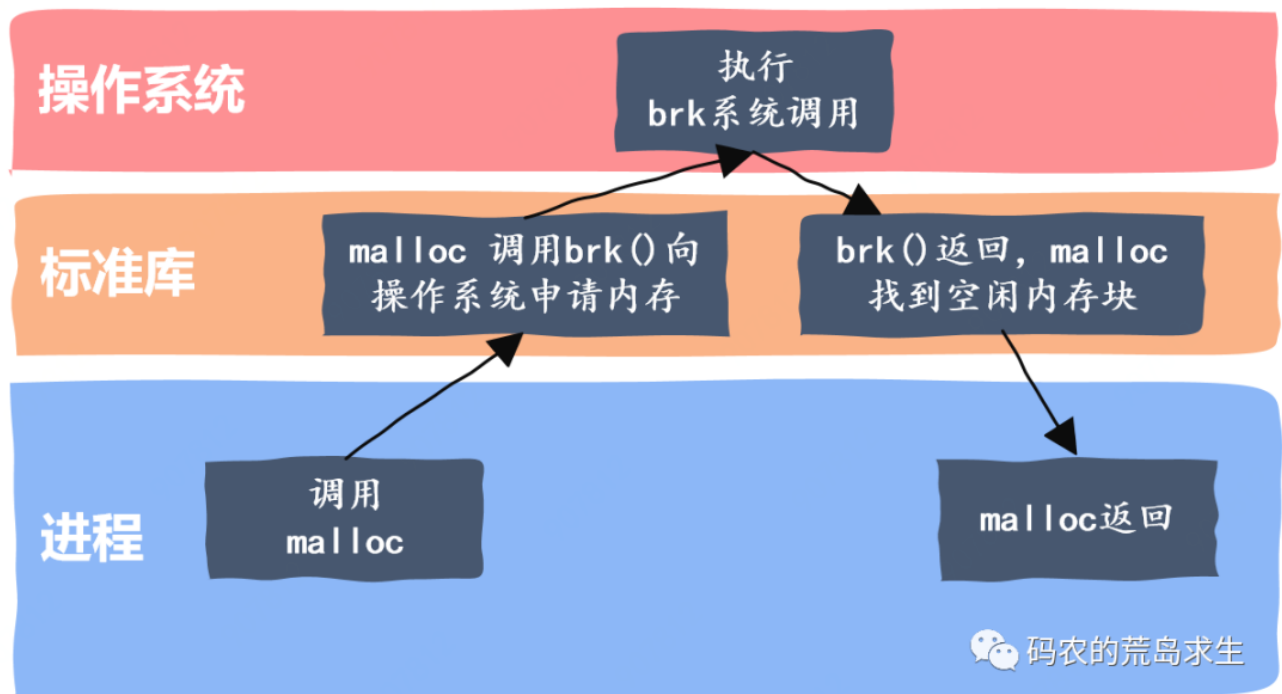
码农的荒岛求生

malloc性能不高的原因一在于其没有为特定场景做优化，除此之外还在于malloc看似简单，但是其调用过程是很复杂的，一次malloc的调用过程可能需要经过操作系统的配合才能完成。

那么调用malloc时底层都发生了什么呢？简单来说会有这样典型的几个步骤：

1. malloc开始搜索空闲内存块，如果能找到一块大小合适的就分配出去

2. 如果malloc找不到一块合适的空闲内存，那么调用brk等系统调用扩大堆区从而获得更多的空闲内存
3. malloc调用brk后开始转入内核态，此时操作系统中的虚拟内存系统开始工作，扩大进程的堆区，注意额外扩大的这一部分内存仅仅是虚拟内存，操作系统并没有为此分配真正的物理内存
4. brk执行结束后返回到malloc，从内核态切换到用户态，malloc找到一块合适的空闲内存后返回



以上就是一次内存申请的完整过程，我们可以看到，一次内存申请过程其实是非常复杂的，关于这个问题的详细讨论你可以参考[这里](#)。

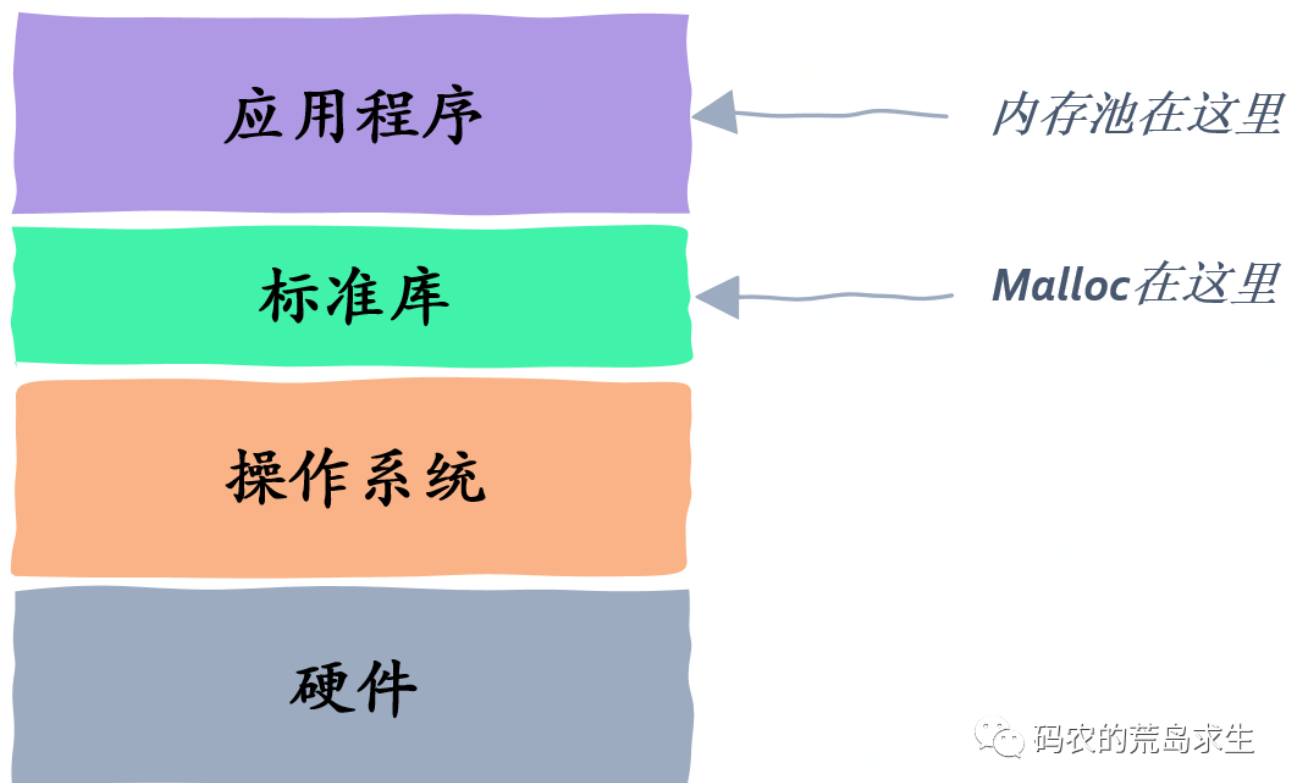
既然每次分配内存都要经过这么复杂的过程，那么如果程序大量使用malloc申请内存那么该程序注定无法获得高性能。

幸好，除了大众货的malloc，我们还可以私人定制，也就是针对特定场景自己来维护内存申请和分配，这就是高性能高并发必备的内存池技术。

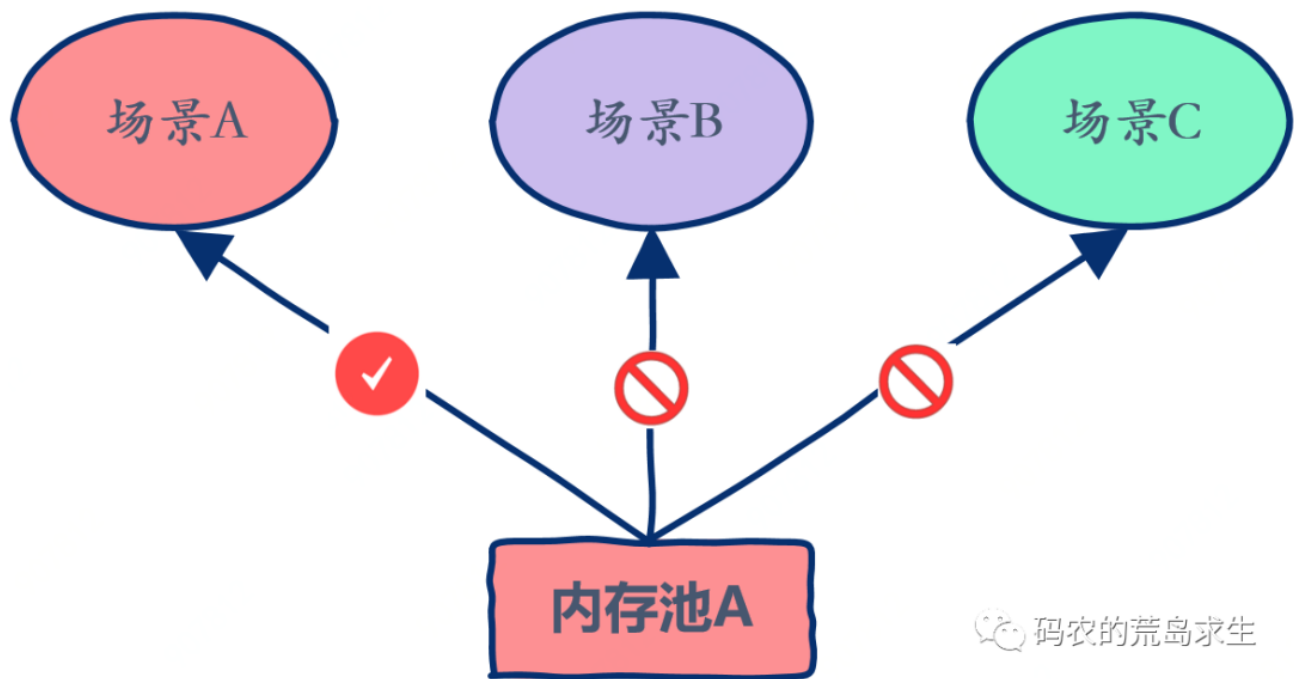
## 内存池技术有什么特殊的吗？

有的同学可能会说，等等，那malloc和这里提到的内存池技术有什么区别呢？

第一个区别在于我们所说的malloc其实是标准库的一部分，位于标准库这一层；而内存池是应用程序的一部分。



其次在于定位，我们自己实现的malloc其实也是定位**通用性**的，通用性的内存分配器设计实现往往比较复杂，但是内存池技术就不一样了，**内存池技术专用于某个特定场景**，以此优化程序性能，但内存池技术的通用性是很差的，在一种场景下有很高性能的内存池基本上没有办法在其它场景也能获得高性能，甚至根本就不能用于其它场景，这就是内存池这种技术的定位。



那么内存池技术是怎样优化性能的呢？

### 内存池技术原理

简单来说，内存池技术一次性获取到大块内存，然后在其之上自己管理内存的申请和释放，这样就**绕过了标准库以及操作系统**：

操作系统

标准库

进程

申请内存

内存池

内存申请无需经过标准库以及操作系统  码农的荒岛求生

也就是说，通过内存池，一次内存的申请再也不用去绕一大圈了。

除此之外，我们可以根据特定的使用模式来进一步优化，比如在服务器端，每次用户请求需要创建的对象可能就那几种，那么这时我们就可以在自己的内存池上**提前创建**出这些对象，当业务逻辑需要时就从内存池中申请已经创建好的对象，使用完毕后还回内存池。

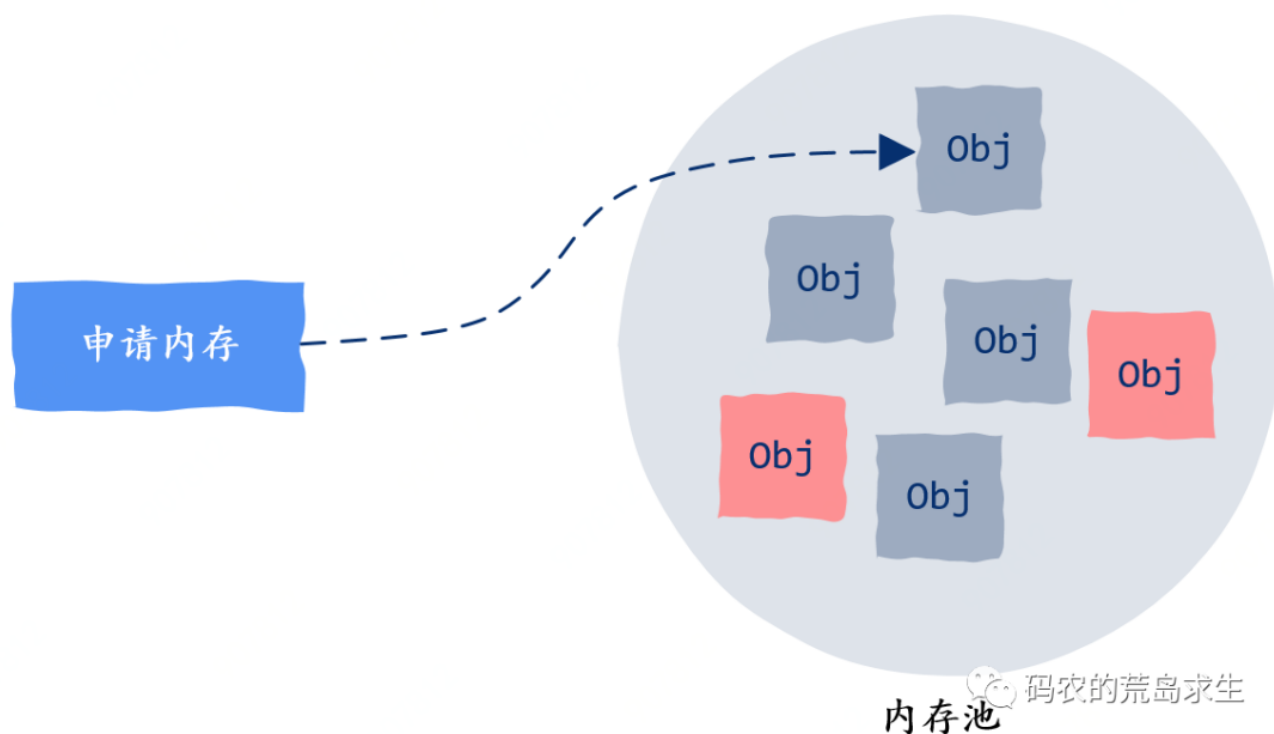
因此我们可以看到，这种为某些应用场景定制的内存池相比通用的比如malloc内存分配器会有大的优势。

接下来我们就着手实现一个。

### 实现内存池的考虑

值得注意的是，内存池实际上有很多的实现方法，在这里我们还是以服务器端编程为例来说明。

假设你的服务器程序非常简单，处理用户请求时只使用一种对象(数据结构)，那么最简单的就是我们提前申请出一堆来，使用的时候拿出一个，使用完后还回去：



怎么样，足够简单吧！这样的内存池只能分配特定对象(数据结构)，当然这样的内存池需要自己维护哪些对象是已经被分配出去的，哪些是还没有被使用的。

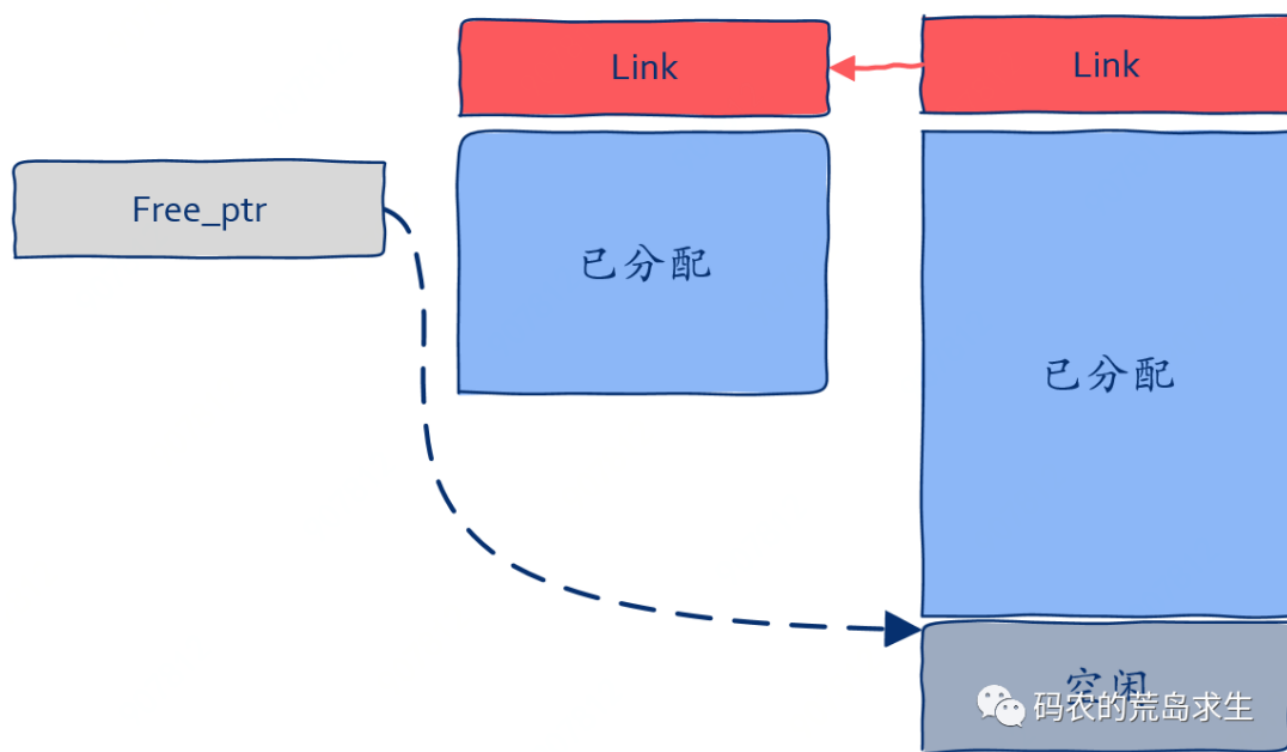
但是，在这里我们可以实现一个稍微复杂一些的，那就是可以申请不同大小的内存，而且由于是服务器端编程，那么一次用户请求过程中我们只申请内存，只有当用户请求处理完毕后**一次性释放所有内存**，从而将内存申请释放的开销降低到最小。

因此，你可以看到，内存池的设计都是针对特定场景的。

现在，有了初步的设计，接下来就是细节了。

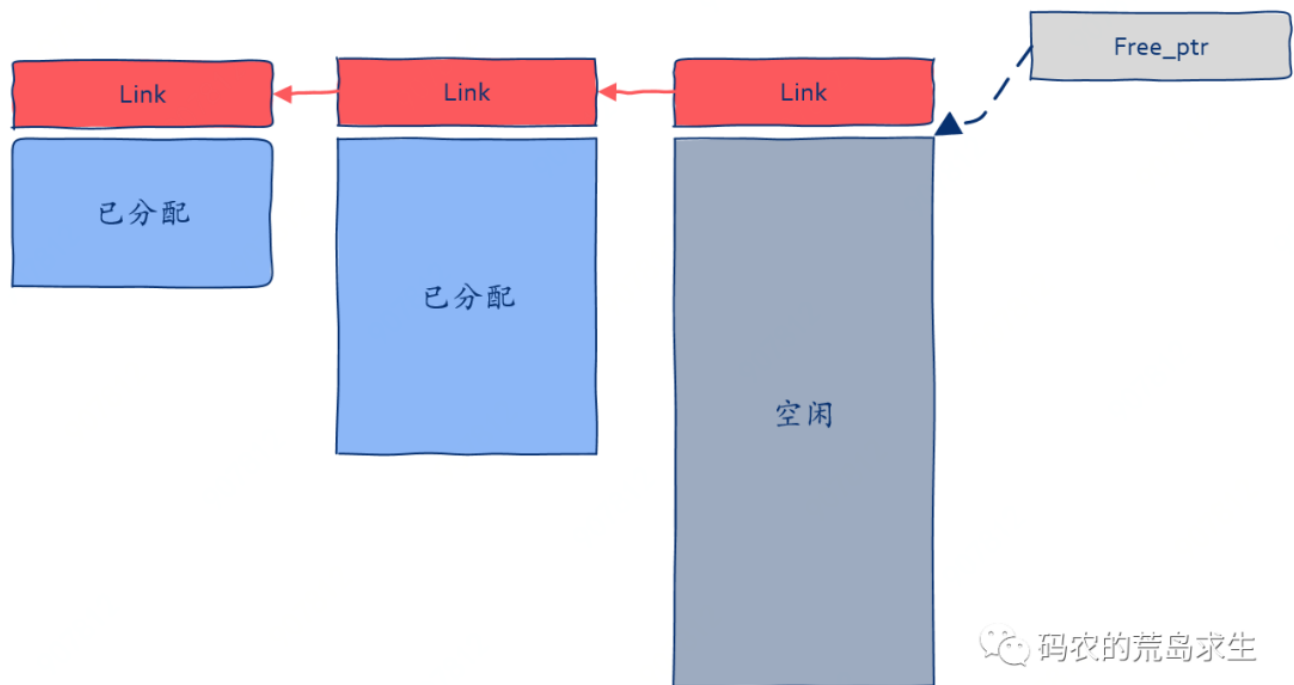
## 数据结构

为了能够分配大小可变的对象，显然我们需要管理空闲内存块，我们可以用一个链表把所有内存块链接起来，然后使用一个指针来记录当前空闲内存块的位置，如图所示：



从图中我们可以看到，有两个空闲内存块，空闲内存之间使用链表链接起来，每个内存块都是前一个的2倍，也就是说，当内存池中的空闲内存不足以分配时我们就向 malloc 申请内存，只不过其大小是前一个的2倍：





码农的荒岛求生

其次，我们有一个指针`free_ptr`，指向接下来的空闲内存块起始位置，当向内存池分配内存时找到`free_ptr`并判断当前内存池剩余空闲是否足够就可以了，有就分配出去并修改`free_ptr`，否则向`malloc`再次成倍申请内存。

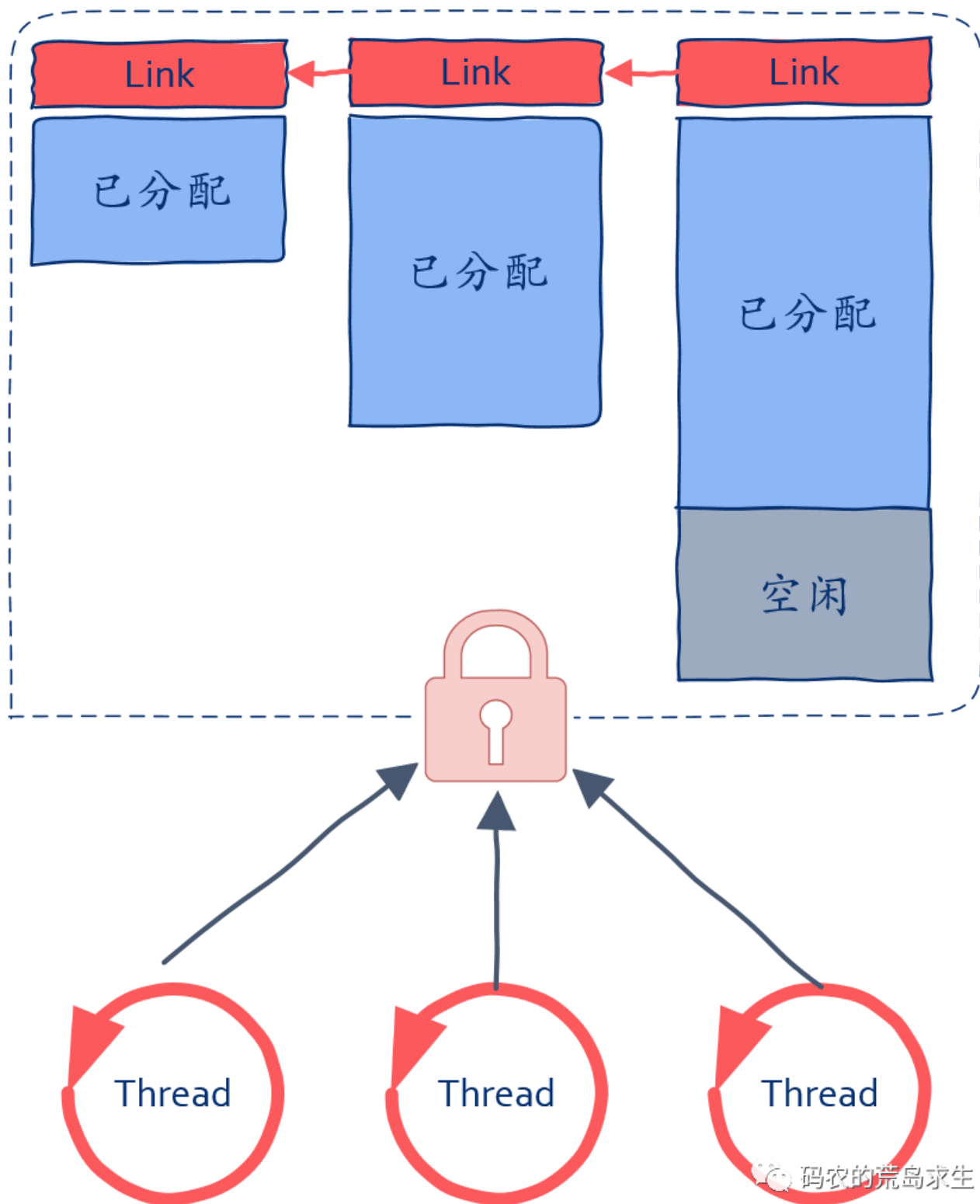
从这里的设计可以看出，我们的内存池其实是不提供类似`free`这样的内存释放函数的，如果要释放内存，那么会一次性将整个内存池释放掉，这一点和通用的内存分配器是不一样的。

现在，我们可以分配内存了，还有一个问题是所有内存池设计不得不考虑的，那就是线程安全，这个话题你可以参考[这里](#)。

## 线程安全

显然，内存池不应该局限在单线程场景，那我们的内存池要怎样实现线程安全呢？

有的同学可能会说这还不简单，直接给内存池一把锁保护就可以了。



这种方法是不是可行呢？还是那句话，It depends，要看情况。

如果你的程序有大量线程申请释放内存，那么这种方案下锁的竞争将会非常激烈，线程这样的场景下使用该方案不会有很好的性能。

那么还有没有一种更好的办法吗？答案是肯定的。

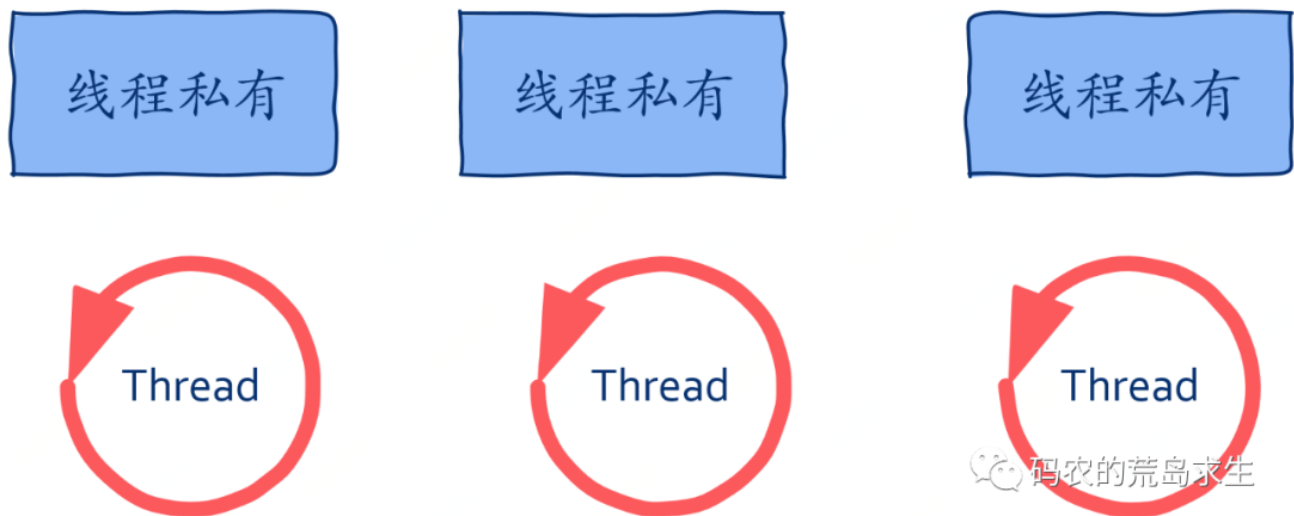
## 线程局部存储

既然多线程使用线程池存在竞争问题，那么干脆我们为每个线程维护一个内存池就好了，这样多线程间就不存在竞争问题了。

那么我们该怎样为每个线程维护一个内存池呢？

线程局部存储，Thread Local Storage正是用于解决这一类问题的，什么是线程局部存储呢？

简单说就是，我们可以创建一个全局变量，因此所有线程都可以使用该全局变量，但与此同时，我们将该全局变量声明为线程私有存储，那么这时虽然所有线程依然看似使用同一个全局变量，但该全局变量在每个线程中都有自己的副本，**变量指向的值是线程私有的**，相互之间不会干扰。

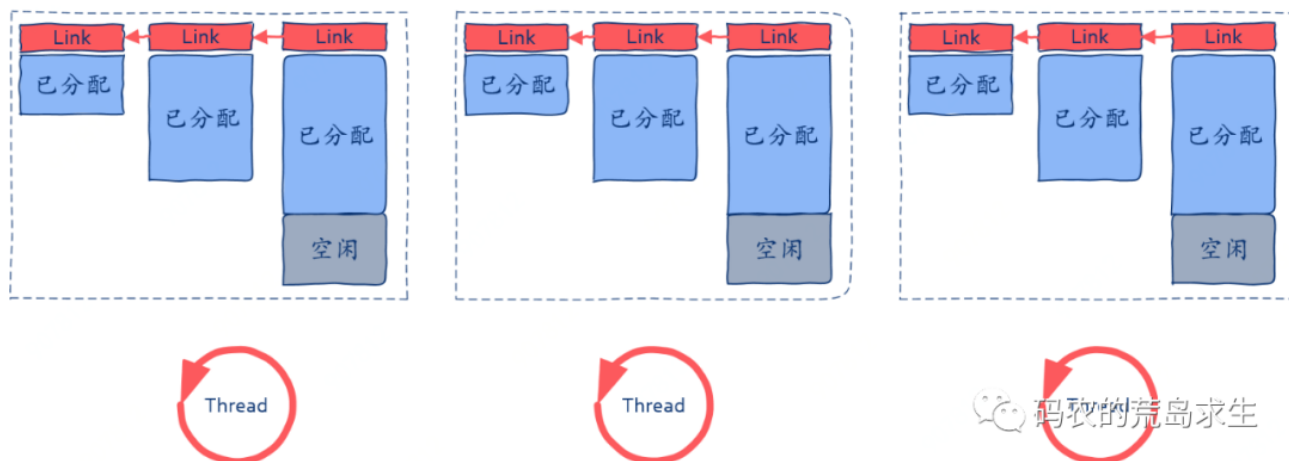


关于线程局部存储，可以参考[这里](#)。

假设这个全局变量是一个整数，变量名字为`global_value`，初始值为100，那么当线程A将`global_value`修改为200时，线程B看到的`global_value`的值依然为100，只有线程A看到的`global_value`为200，这就是线程局部存储的作用。

## 线程局部存储+内存池

有了线程局部存储问题就简单了，我们可以将内存池声明为线程局部存储，这样每个线程都只会操作属于自己的内存池，这样就再也不会有锁竞争问题了。



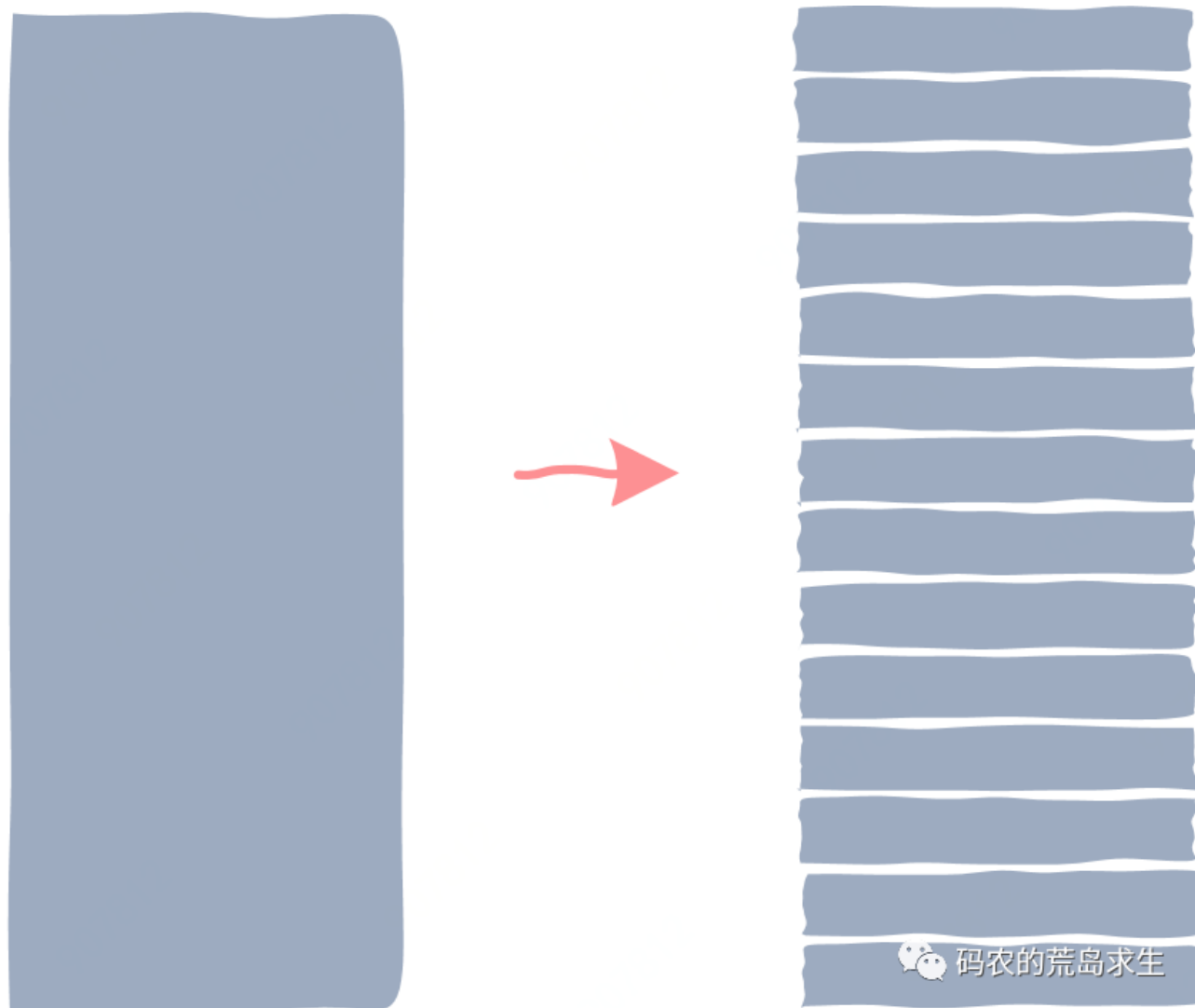
注意，虽然这里给出了线程局部存储的设计，但并不是说加锁的方案就比不上线程局部存储方案，还是那句话，一切要看使用场景，如果加锁的方案够用，那么我们就没有必要绞尽脑汁的去用其它方案，因为加锁的方案更简单，代码也更容易维护。

还需要提醒的是，这里只是给出了内存池的一种实现方法，并不是说所有内存池都要这么设计，内存池可以简单也可复杂，一切要看实际场景，这一点也需要注意。

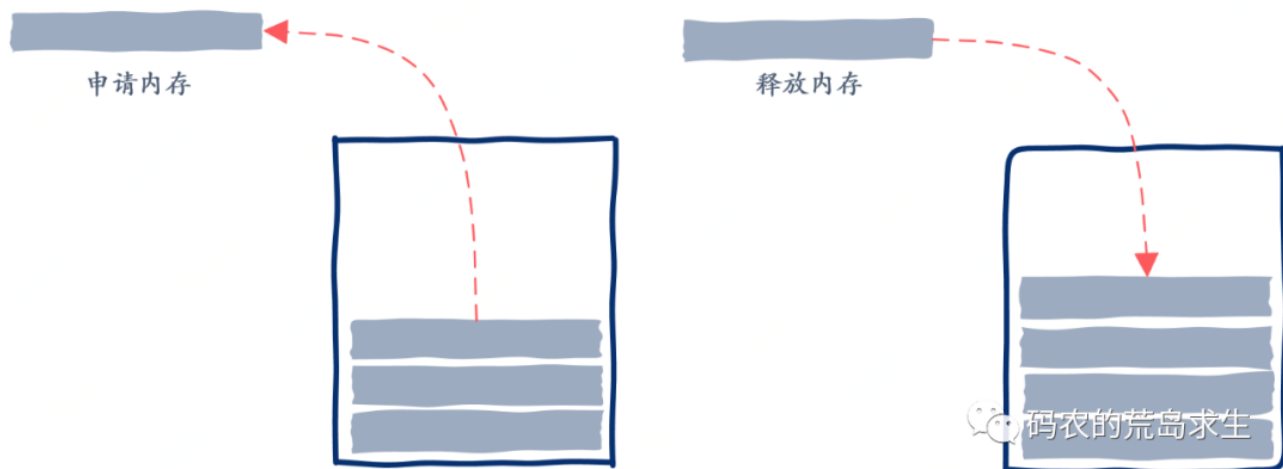
## 其它内存池形式

到目前为止我们给出了两种内存池的设计方法，第一种是提前创建出一堆需要的对象(数据结构)，自己维护好哪些对象(数据结构)可用哪些已被分配；第二种可以申请任意大小的内存空间，使用过程中只申请不释放，最后一次性释放。这两种内存池天然适用于服务器端编程。

最后我们再来介绍一种内存池实现技术，这种内存池会提前申请出一大段内存，然后将这一大段内存切分为大小相同的小内存块：



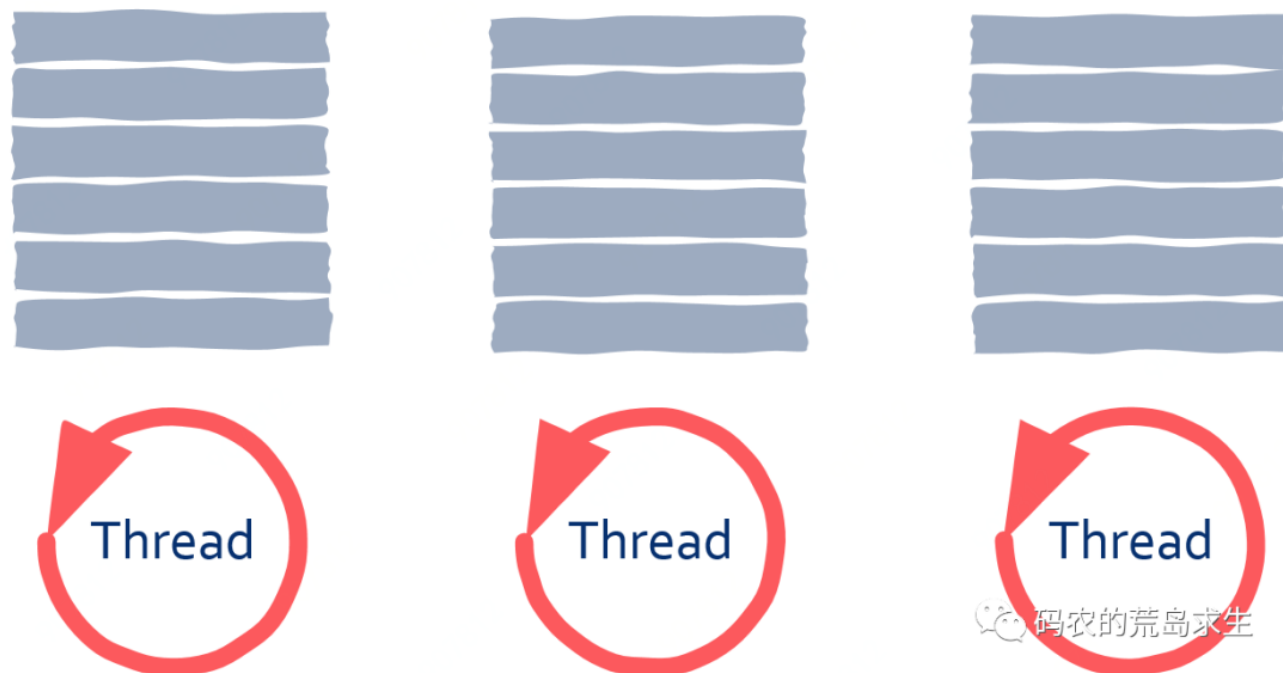
然后我们自己来维护这些被切分出来的小内存块哪些是空闲的哪些是已经被分配的，比如我们可以使用栈这种数据结构，最初把所有空闲内存块地址push到栈中，分配内存是就pop出来一个，用户使用完毕后再push回栈里。



从这里的设计我们可以看出，这种内存池有一个限制，这个限制就是说**程序申请的最大内存不能超过这里内存块的大小**，否则不足以装下用户数据，这需要我们对程序所涉及的业务非常了解才可以。

用户申请到内存后根据需要将其塑造成特定对象(数据结构)。

关于线程安全的问题，可以同样采用线程局部存储的方式来实现：



### 一个有趣的问题

除了线程安全，这里还有一个非常有趣的问题，那就是如果线程A申请的对象被线程B拿去释放，我们的内存池该怎么处理呢？

这个问题之所以有趣是因为我们**必须知道该内存属于哪个线程的局部存储**，但申请的**内存本身并不能告诉你这样的信息**。

有的同学可能会说这还不简单，不就是一个指针到另一个指针的映射吗，直接用map之类存起来就好了，但问题并没有这么简单，原因就在于如果我们切分的内存块很小，那么会存在大量内存块，这就需要存储大量的映射关系，有没有办法改进呢？

改进方法是这样的，一般来说，我们申请到的大段内存其实是会按照特定大小进行内存对齐，我们假设总是按照4K字节对齐，那么该大段内存的起始地址后12个bit( $4K = 2^{12}$ )为总是0，比如地址0x9abcd000，同时我们也假设申请到的大段内存大小也是4K：

0x9abcd000



那么我们就知道该大段内存中的各个小内存块起始地址除了后12个bit位外都是一样的：

0x9abcd000

0x9abcd200

0x9abcd400


0x9abcd600

0x9abcd800

0x9abcd a00

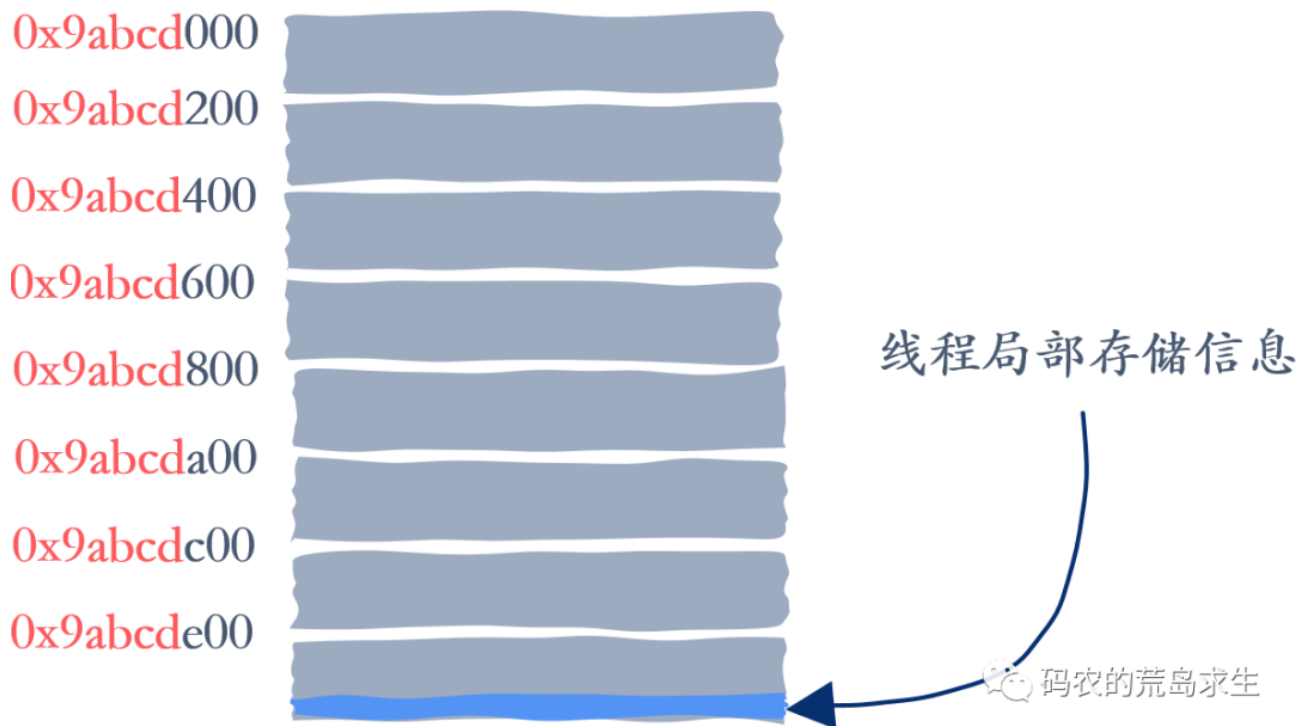
0x9abcd c00

0x9abcd e00

 码农的荒岛求生

这样拿到任意一个内存的地址我们就能知道对应的大段内存的起始地址，只需要简单的将后12个bit置为0即可，有了大段内存的起始地址剩下的就简单了，我们可以在大段内存中的最后保存对应的线程局部存储信息：





这样我们对任意一个内存块地址进行简单的位运算就可以得到对应的线程局部存储信息，大大减少了维护映射信息对内存的占用。

## 总结

内存池是高性能服务器中常见的一种优化技术，在这里我们介绍了三种实现方法，值得注意的是，内存池实现没有统一标准，一切都要根据具体场景定制，因此我们可以看到内存池设计是有针对性的，当然其反面就是不具备通用性。

希望本文对大家理解内存池有所帮助。

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