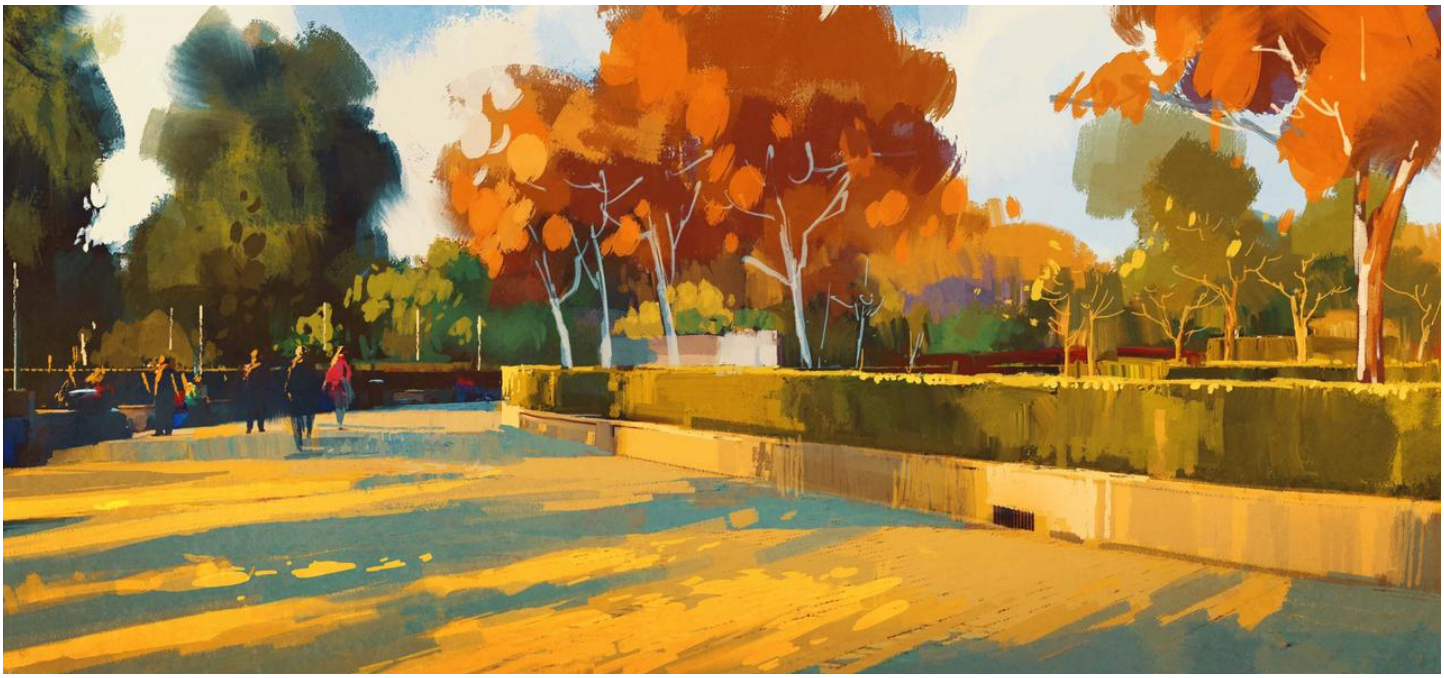


C++ - 高性能内存池

项目介绍

1. 这个项目做的是什么?

当前项目是实现一个高并发的内存池，他的原型是google的一个开源项目tcmalloc，tcmalloc全称Thread-Caching Malloc，即线程缓存的malloc，实现了高效的多线程内存管理，用于替代系统的内存分配相关的函数（malloc、free）。

我们这个项目是把tcmalloc最核心的框架简化后拿出来，模拟实现出一个自己的高并发内存池，目的就是学习tcamlloc的精华，这种方式有点类似我们之前学习STL容器的方式。但是相比STL容器部分，tcmalloc的代码量和复杂度上升了很多，大家要有心理准备。当前另一方面，难度的上升，我们的收获和成长也是在这个过程中同步上升。

另一方面tcmalloc是全球大厂google开源的，可以认为当时顶尖的C++高手写出来的，他的知名度也是非常高的，不少公司都在用它，Go语言直接用它做了自己内存分配器。所以很多程序员是熟悉这个项目的，那么有好处，也有坏处。好处就是把这个项目理解扎实了，会很受面试官的认可。坏处就是面试官可能也比较熟悉项目，对项目会问得比较深，比较细。如果你对项目掌握得不扎实，那么就容易碰钉子。

2. 这个项目的要求的知识储备和难度?

这个项目会用到C/C++、数据结构（链表、哈希桶）、操作系统内存管理、单例模式、多线程、互斥锁

等等方面的知识。

什么是内存池

1. 池化技术

所谓“池化技术”，就是程序先向系统申请过量的资源，然后自己管理，以备不时之需。之所以要申请过量的资源，是因为每次申请该资源都有较大的开销，不如提前申请好了，这样使用时就会变得非常快捷，大大提高程序运行效率。

在计算机中，有很多使用“池”这种技术的地方，除了内存池，还有连接池、线程池、对象池等。以服务器上的线程池为例，它的主要思想是：先启动若干数量的线程，让它们处于睡眠状态，当接收到客户端的请求时，唤醒池中某个睡眠的线程，让它来处理客户端的请求，当处理完这个请求，线程又进入睡眠状态。

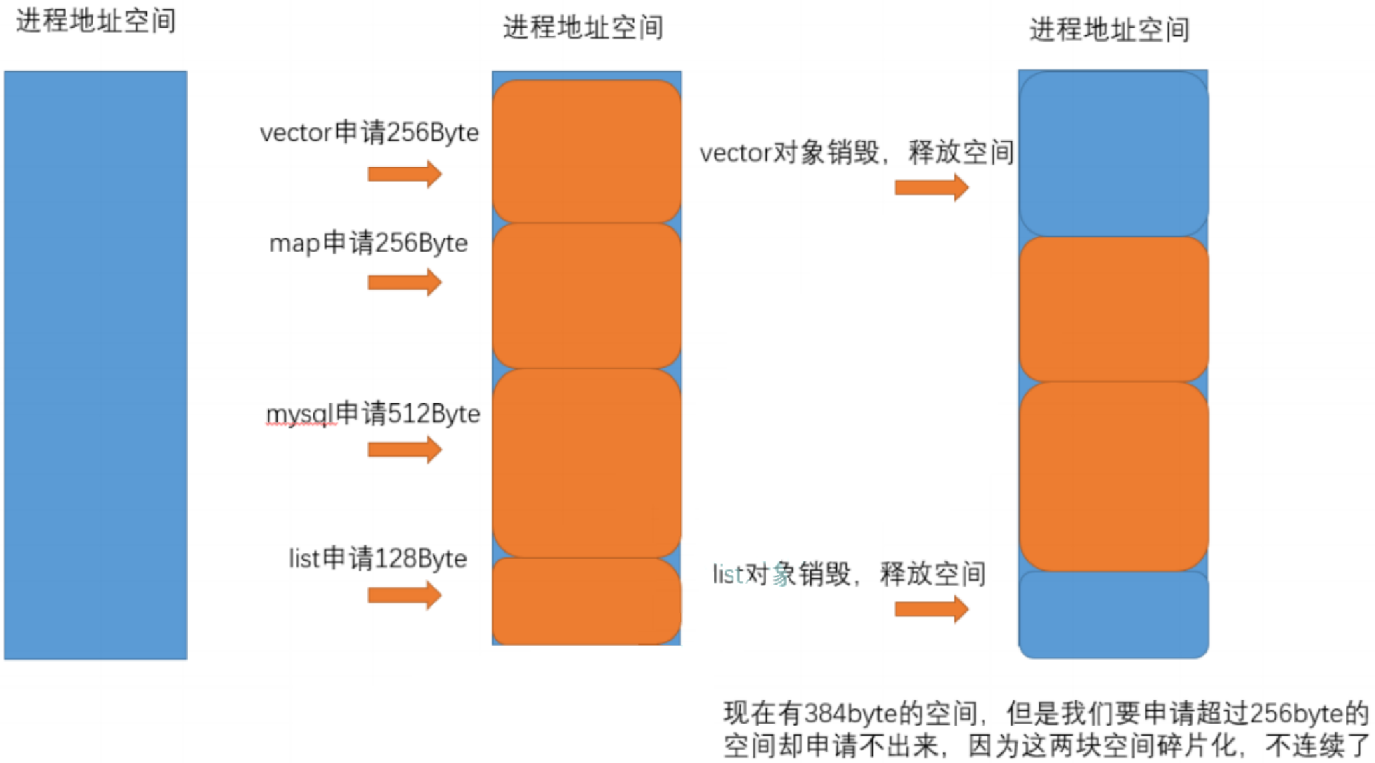
2. 内存池

内存池是指程序预先从操作系统申请一块足够大内存，此后，当程序中需要申请内存的时候，不是直接向操作系统申请，而是直接从内存池中获取；同理，当程序释放内存的时候，并不真正将内存返回给操作系统，而是返回内存池。当程序退出(或者特定时间)时，内存池才将之前申请的内存真正释放。

3. 内存池主要解决的问题

内存池主要解决的当然是效率的问题，其次如果作为系统的内存分配器的角度，还需要解决一下内存碎片的问题。那么什么是内存碎片呢?

再需要补充说明的是内存碎片分为外碎片和内碎片，上面我们讲的外碎片问题。外部碎片是一些空闲的连续内存区域太小，这些内存空间不连续，以至于合计的内存足够，但是不能满足一些的内存分配申请需求。内部碎片是由于一些对齐的需求，导致分配出去的空间中一些内存无法被利用。内碎片问题，我们后面项目就会看到，那会再进行更准确的理解。

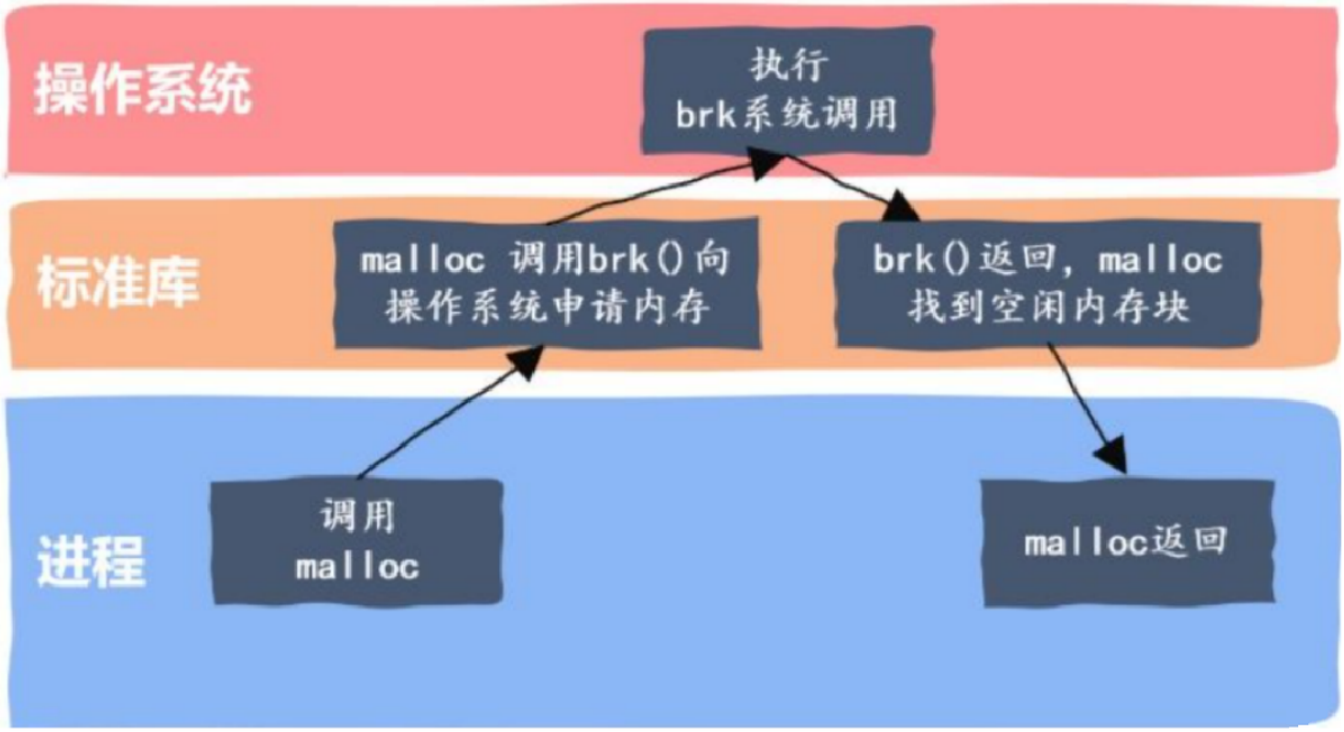


4. malloc

C/C++中我们要动态申请内存都是通过malloc去申请内存，但是我们要知道，实际我们不是直接去堆获取内存的，

而malloc就是一个内存池。malloc() 相当于向操作系统“批发”了一块较大的内存空间，然后“零售”给程序用。当全部“售完”或程序有大量的内存需求时，再根据实际需求向操作系统“进货”。

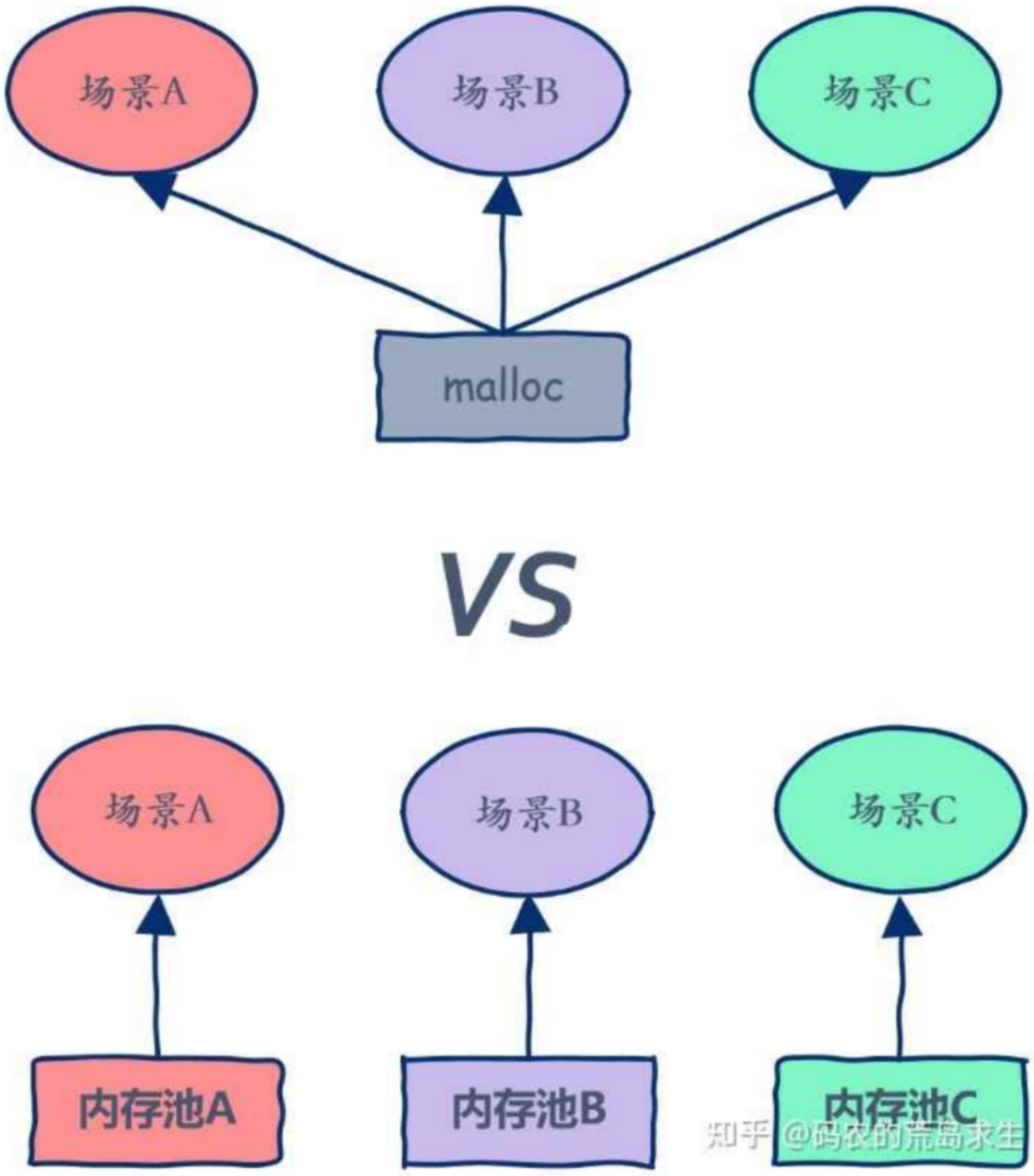
malloc的实现方式有很多种，一般不同编译器平台用的都是不同的。比如windows的vs系列用的微软自己写的一套，linux gcc用的glibc中的ptmalloc。下面有几篇关于这块的文章，大概可以去简单看看了解一下，关于ptmalloc，学完我们的项目以后，有兴趣大家可以去看看他的实现细节。



5. 开胃菜--先设计一个定长的内存池

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，下面我们就先来设计一个定长内存池做个开胃菜，当然这个定长内存池在我们后面的高并发内存池中也是有价值的，所以学习他目的有两层，先熟悉一下简单内存池是如何控制的，第二他会作为我们后面内存池的一个基础组件。



6. 高并发内存池整体框架设计

现代很多的开发环境都是多核多线程，在申请内存的场景下，必然存在激烈的锁竞争问题。malloc本身其实已经很优秀，那么我们项目的原型tcmalloc就是在多线程高并发的场景下更胜一筹，所以这次我们实现的内存池需要考虑以下几方面的问题。

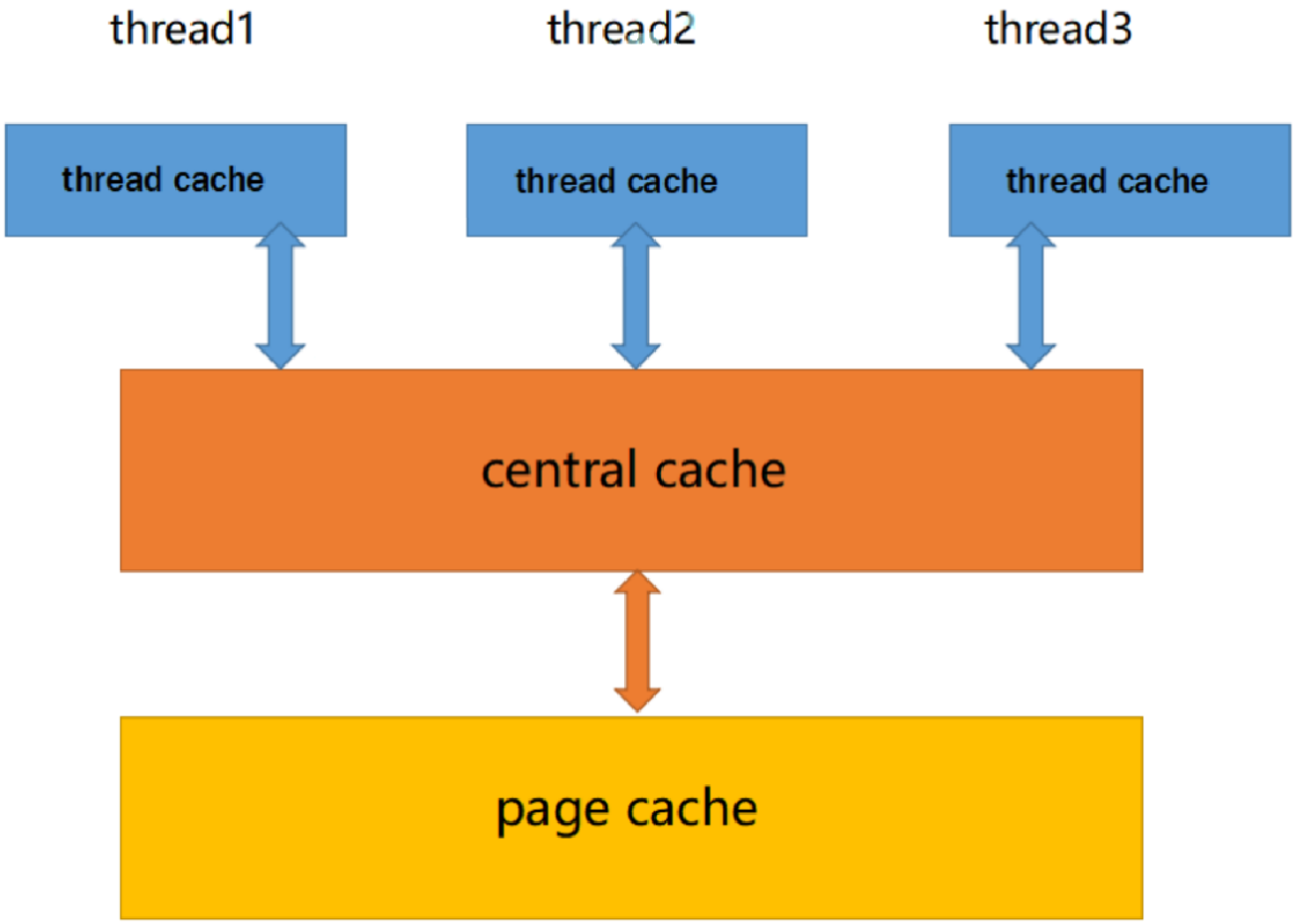
6.1 性能问题。

6.2 多线程环境下，锁竞争问题。

6.3 内存碎片问题。

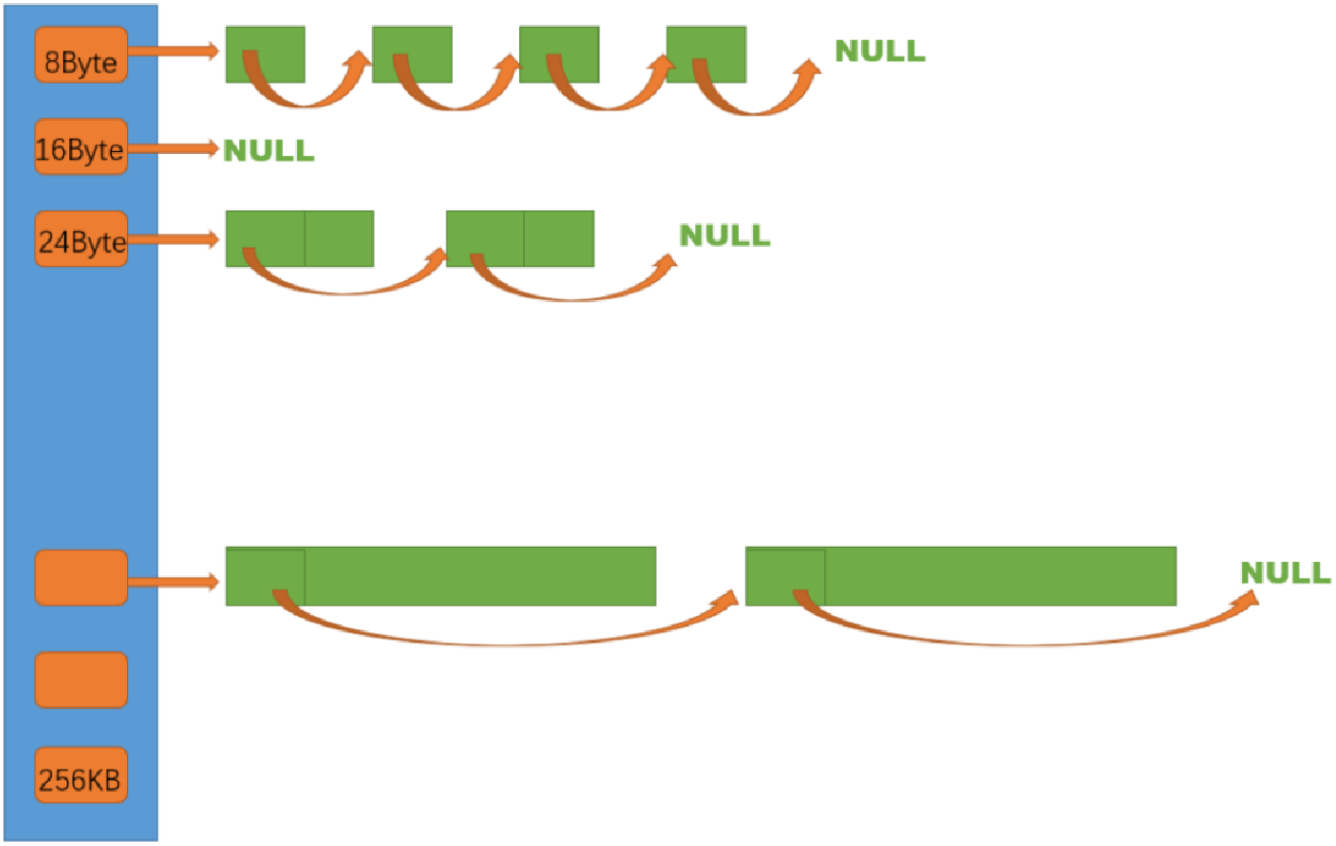
concurrent memory pool主要由以下3个部分构成：

1. thread cache：线程缓存是每个线程独有的，用于小于256KB的内存的分配，线程从这里申请内存不需要加锁，每个线程独享一个cache，这也就是这个并发线程池高效的地方。
2. central cache：中心缓存是所有线程所共享，thread cache是按需从central cache中获取的对象。central cache合适的时机回收thread cache中的对象，避免一个线程占用了太多的内存，而其他线程的内存吃紧，达到内存分配在多个线程中更均衡的按需调度的目的。central cache是存在竞争的，所以从这里取内存对象是需要加锁，首先这里用的是桶锁，其次只有thread cache的没有内存对象时才会找central cache，所以这里竞争不会很激烈。
3. page cache：页缓存是在central cache缓存上面的一层缓存，存储的内存是以页为单位存储及分配的，central cache没有内存对象时，从page cache分配出一定数量的page，并切割成定长大小的小块内存，分配给central cache。当一个span的几个跨度页的对象都回收以后，page cache会回收central cache满足条件的span对象，并且合并相邻的页，组成更大的页，缓解内存碎片的问题。



7. 高并发内存池--thread cache

thread cache是哈希桶结构，每个桶是一个按桶位置映射大小的内存块对象的自由链表。每个线程都会有一个thread cache对象，这样每个线程在这里获取对象和释放对象时是无锁的。



申请内存：

1. 当内存申请size<=256KB，先获取到线程本地存储的thread cache对象，计算size映射的哈希桶自由链表下标i。
2. 如果自由链表_freeLists[i]中有对象，则直接Pop一个内存对象返回。
3. 如果_freeLists[i]中没有对象时，则批量从central cache中获取一定数量的对象，插入到自由链表并返回一个对象。

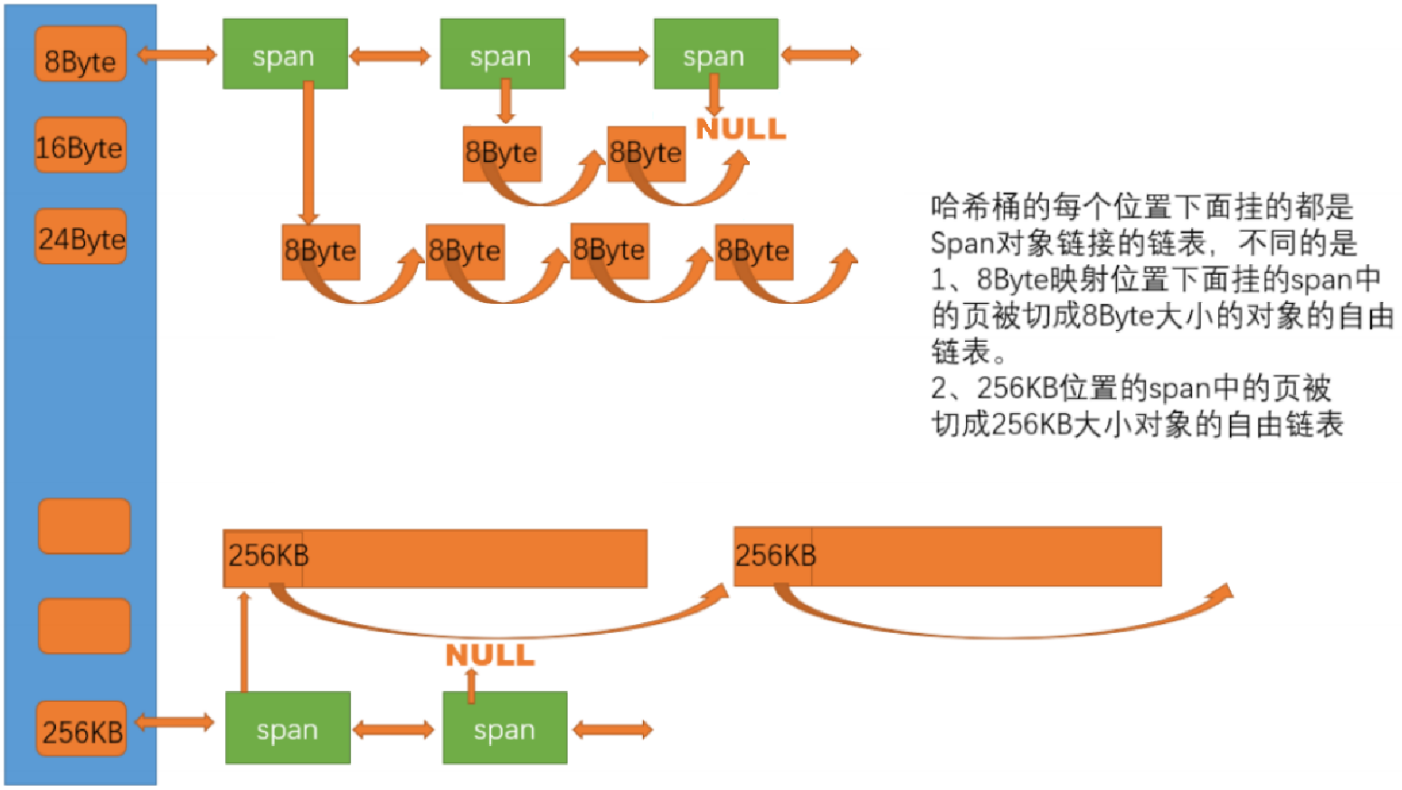
释放内存：

1. 当释放内存小于256k时将内存释放回thread cache，计算size映射自由链表桶位置i，将对象Push到_freeLists[i]。
2. 当链表的长度过长，则回收一部分内存对象到central cache。

8. 高并发内存池--central cache

central cache也是一个哈希桶结构，他的哈希桶的映射关系跟thread cache是一样的。不同的是他的每个哈希桶位置挂是SpanList链表结构，不过每个映射桶下面的span中的大内存块被按映射关系切成

了一个个小内存块对象挂在span的自由链表中。



申请内存：

1. 当thread cache中没有内存时，就会批量向central cache申请一些内存对象，这里的批量获取对象

的数量使用了类似网络tcp协议拥塞控制的慢启动算法；central cache也有一个哈希映射的spanlist，spanlist中挂着span，从span中取出对象给thread cache，这个过程是需要加锁的，不过这里使用的是一个桶锁，尽可能提高效率。

2. central cache映射的spanlist中所有span的都没有内存以后，则需要向page cache申请一个新的span对象，拿到span以后将span管理的内存按大小切好作为自由链表链接到一起。然后从span中取对象给thread cache。
3. central cache的中挂的span中use_count记录分配了多少个对象出去，分配一个对象给thread cache，就++use_count

释放内存：

1. 当thread_cache过长或者线程销毁，则会将内存释放回central cache中的，释放回来时--use_count。当use_count减到0时则表示所有对象都回到了span，则将span释放回page cache，page cache中会对前后相邻的空闲页进行合并。

9. 高并发内存池--page cache

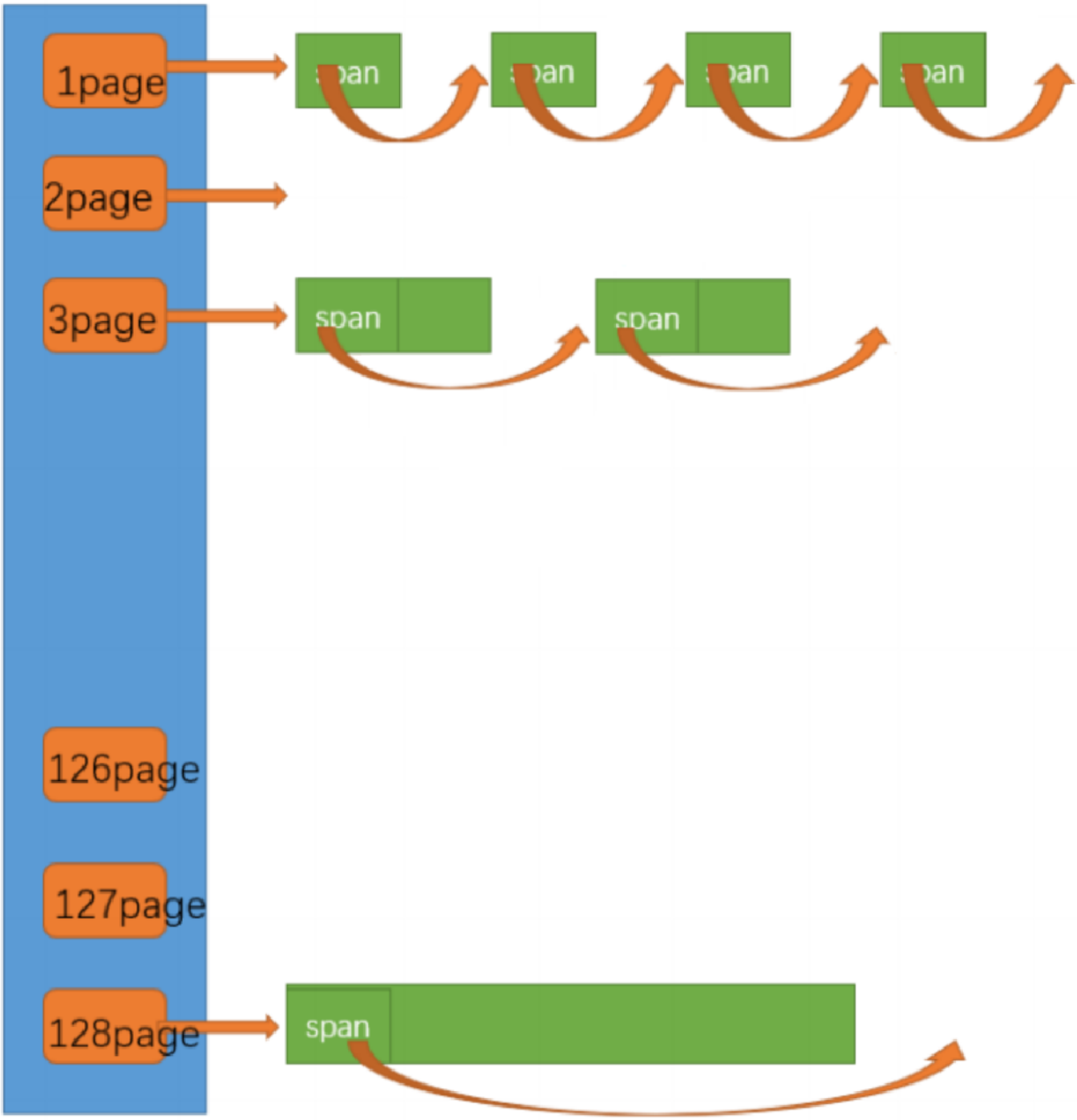
申请内存：

1. 当central cache向page cache申请内存时，page cache先检查对应位置有没有span，如果没有则向更大页寻找一个span，如果找到则分裂成两个。比如：申请的是4页page，4页page后面没有挂span，则向后面寻找更大的span，假设在10页page位置找到一个span，则将10页page span分裂为一个4页page span和一个6页page span。
2. 如果找到_spanList[128]都没有合适的span，则向系统使用mmap、brk或者是VirtualAlloc等方式申请128页page span挂在自由链表中，再重复1中的过程。
3. 需要注意的是central cache和page cache 的核心结构都是spanlist的哈希桶，但是他们是有本质区

别的，central cache中哈希桶，是按跟thread cache一样的大小对齐关系映射的，他的spanlist中挂的span中的内存都被按映射关系切好链接成小块内存的自由链表。而page cache 中的spanlist则是按下标桶号映射的，也就是说第i号桶中挂的span都是i页内存。

释放内存：

1. 如果central cache释放回一个span，则依次寻找span的前后page id的没有在使用的空闲span，看是否可以合并，如果合并继续向前寻找。这样就可以将切小的内存合并收缩成大的span，减少内存碎片。



10. 相对于malloc优势

主要优势：

1. 线程本地缓存（核心优势）

- **每个线程有自己的缓存**：TCMalloc为每个线程维护一个本地缓存池（thread-local cache），小对象分配不需要全局锁
- **减少锁竞争**：标准malloc通常使用全局锁，多线程并发分配时锁竞争激烈

- **快速路径**：大部分小内存分配（默认≤256KB）直接从线程本地缓存获取，几乎无锁

2. 分级内存管理

代码块

```
// TCMalloc内存层级：
Thread Cache → Central Heap → Page Heap → OS
// 大部分分配在Thread Cache就完成了
```

- **小对象**（≤256KB）：使用预分配的大小类别（size class），减少碎片
- **中对象**（256KB-1MB）：使用span管理
- **大对象**（>1MB）：直接使用mmap/virtual alloc

3. 减少系统调用

- 批量从操作系统获取内存，减少brk/mmap调用次数
- 中央堆（Central Heap）作为线程缓存的后备池

性能对比示例：

代码块

```
// 多线程场景下的性能差异可能非常大
// 标准malloc：多线程竞争全局锁
void* malloc(size_t size) {
    pthread_mutex_lock(&global_lock);  // 瓶颈!
    // ... 分配逻辑
    pthread_mutex_unlock(&global_lock);
}

// TCMalloc：大部分情况无锁
void* tc_malloc(size_t size) {
    if (size <= kMaxThreadCacheSize) {
        // 从线程本地缓存分配，无锁
        return thread_cache->Allocate(size);
    }
    // 大对象才需要锁
}
```

实际测试数据：

- **Google的测试**：在8核机器上，多线程分配小对象时，TCMalloc比ptmalloc2快约**2.5倍**
- **Redis的实践**：使用TCMalloc后，多客户端并发性能提升明显
- **MySQL/Percona**：许多发行版默认使用TCMalloc

适用场景：

- ✅ **多线程服务器应用**（Web服务器、数据库、缓存系统）
- ✅ **频繁分配小对象**的程序
- ✅ **高并发环境**
- ❌ 单线程简单应用可能优势不明显
- ❌ 某些特定分配模式可能不如jemalloc

需要注意：

1. **内存碎片**：TCMalloc可能比ptmalloc使用更多内存（空间换时间）
2. **调优参数**：可通过环境变量调整线程缓存大小等参数
3. **不是银弹**：对于大对象分配，优势不明显

替代方案：

- **jemalloc**：Facebook开发，在某些场景下更均衡
- **mimalloc**：微软开发，更注重安全和通用性

总结：TCMalloc通过线程本地缓存显著减少了多线程环境下的锁竞争，这是它高效的关键。如果你的应用是多线程且频繁分配内存，使用TCMalloc通常能获得明显的性能提升。