从零实现一个高并发的内存池

1.什么是内存池

1.1 池化技术

池 是在计算机技术中经常使用的一种设计模式，其内涵在于：**将程序中需要经常使用的核心资源先申请出来，放到一个池内，由程序自己管理，这样可以提高资源的使用效率，也可以保证本程序占有的资源数量。**经常使用的池技术包括内存池、线程池和连接池等，其中尤以内存池和线程池使用最多。

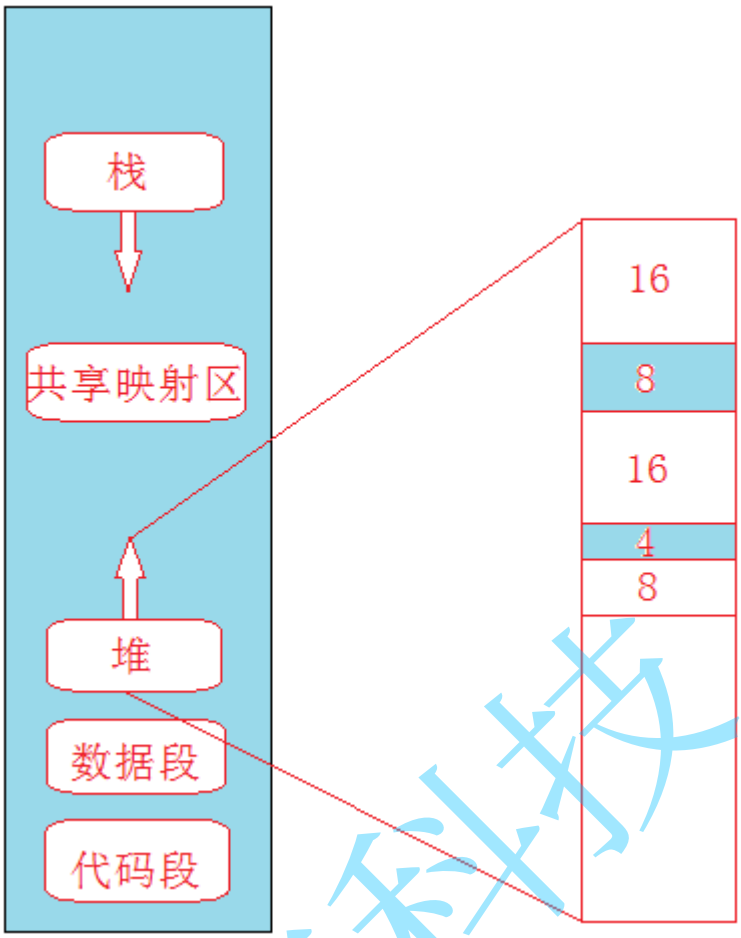
1.2 内存池

内存池(Memory Pool) 是一种动态内存分配与管理技术。 通常情况下，程序员习惯直接使用 **new、delete、malloc、free** 等API申请分配和释放内存，这样导致的后果是：**当程序长时间运行时，由于所申请内存块的大小不定，频繁使用时会造成大量的内存碎片从而降低程序和操作系统的性能。内存池则是在真正使用内存之前，先申请分配一大块内存(内存池)留作备用，当程序员申请内存时，从池中取出一块动态分配，当程序员释放内存时，将释放的内存再放入池内，再次申请池可以 再取出来使用**，并尽量与周边的空闲内存块合并。若内存池不够时，则自动扩大内存池，从操作系统中申请更大的内存池。

2. 为什么需要内存池?

2.1 内存碎片问题

假设系统依次分配了16byte、8byte、16byte、4byte，还剩余8byte未分配。这时要分配一个24byte的空间，操作系统回收了一个上面的两个16byte，总的剩余空间有40byte，但是却不能分配出一个连续24byte的空间，这就是内存碎片问题。



2.2 申请效率的问题

例如：我们上学家里给生活费一样，假设一学期的生活费是6000块。

方式1：开学时6000块直接给你，自己保管，自己分配如何花。

方式2：每次要花钱时，联系父母，父母转钱。

同样是6000块钱，第一种方式的效率肯定更高，因为第二种方式跟父母的沟通交互成本太高了。

同样的道理，程序就像是上学的童鞋，操作系统就像父母，频繁申请内存的场景下，每次需要内存，都像系统申请效率必然有影响。

3.内存池设计的演进

3.1 教科书上的内存分配器：

做一个链表指向空闲内存，分配就是取出一块来，改写链表，返回，释放就是放回到链表里面，并做好归并。注意做好标记和保护，避免二次释放，还可以花点力气在如何查找最适合大小的内存快的搜索上，减少内存碎片，有空你了还可以把链表换成伙伴算法。

优点：实现简单

缺点：分配时搜索合适的内存块效率低，释放回归内存后归并消耗大，实际中不实用。

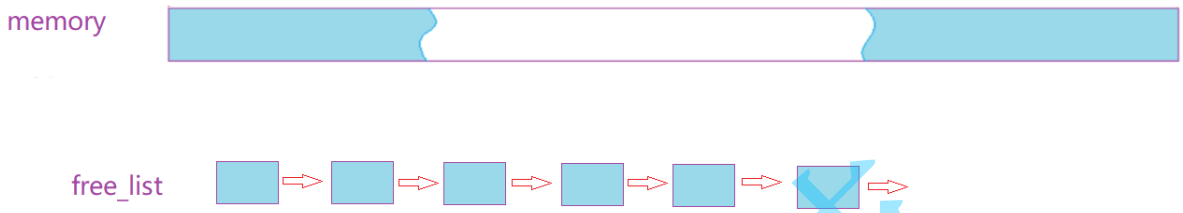
3.2 定长内存分配器：

即实现一个 FreeList，每个 FreeList 用于分配固定大小的内存块，比如用于分配 32字节对象的固定内存分配器，之类的。每个固定内存分配器里面有两个链表，OpenList 用于存储未分配的空闲对象，CloseList用于存储已分配的内存对象，那么所谓的分配就是从 OpenList 中取出一个对象放到 CloseList 里并且返回给用户，释放又是从 CloseList 移回到 OpenList。分配时如果不够，那么就需要增长 OpenList：申请一个大一点的内存块，切割成比如 64 个相同大小的对象添加到 OpenList中。这个固定内存分配器回收的时候，统一把先前向系统申请的内存块全部还给系统。

优点：简单粗暴，200行代码就可以搞定，分配和释放的效率高，解决实际中特定场景下的问题有效。

缺点：功能单一，只能解决定长的内存需求，另外占着内存没有释放。

范例：一个简单O(1)定长内存池实现



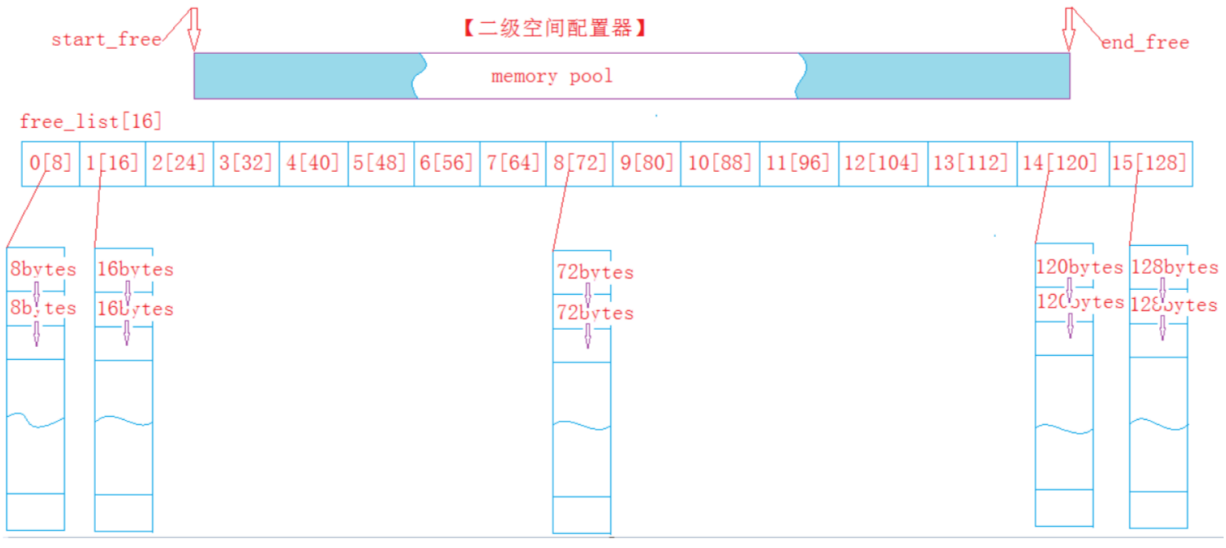
3.3 哈希映射的FreeList 池：

在你实现了 FreeList的基础上，按照不同对象大小（8字节，16字节，32，64，128，256，512，1K。。。64K），构造十多个固定内存分配器，分配内存时根据内存大小查表，决定到底由哪个分配器负责，分配后要在头部的 header 处（ptr[-sizeof(char*)]处）写上 cookie，表示又哪个分配器分配的，这样释放时候你才能正确归还。如果大于64K，则直接用系统的 malloc作为分配，如此以浪费内存为代价你得到了一个分配时间近似O（1）的内存分配器，差不多实现了一个 memcached 的 slab 内存管理器了，但是先别得意。此 slab 非彼 slab（sunos/solaris/linux kernel 的 slab）。这说白了还是一个弱智的 freelist 无法归还内存给操作系统，某个 FreeList 如果高峰期占用了大量内存即使后面不用，也无法支援到其他内存不够的 FreeList，所以我们做的这个和 memcached 类似的分配器其实是比较残缺的，你还需要往下继续优化。

优点：这个本质是定长内存池的改进，分配和释放的效率高。可以解决一定长度内的问题。

缺点：存在内碎片的问题，且将一块大内存切小以后，申请大内存无法使用。多线程并发场景下，锁竞争激烈，效率降低。

范例：sgi stl 六大组件中的空间配置器就是这种设计实现的。



3.4 malloc

malloc原理

malloc函数原理

关于malloc的原理大家参考网上的讲解或者可以去看源代码。

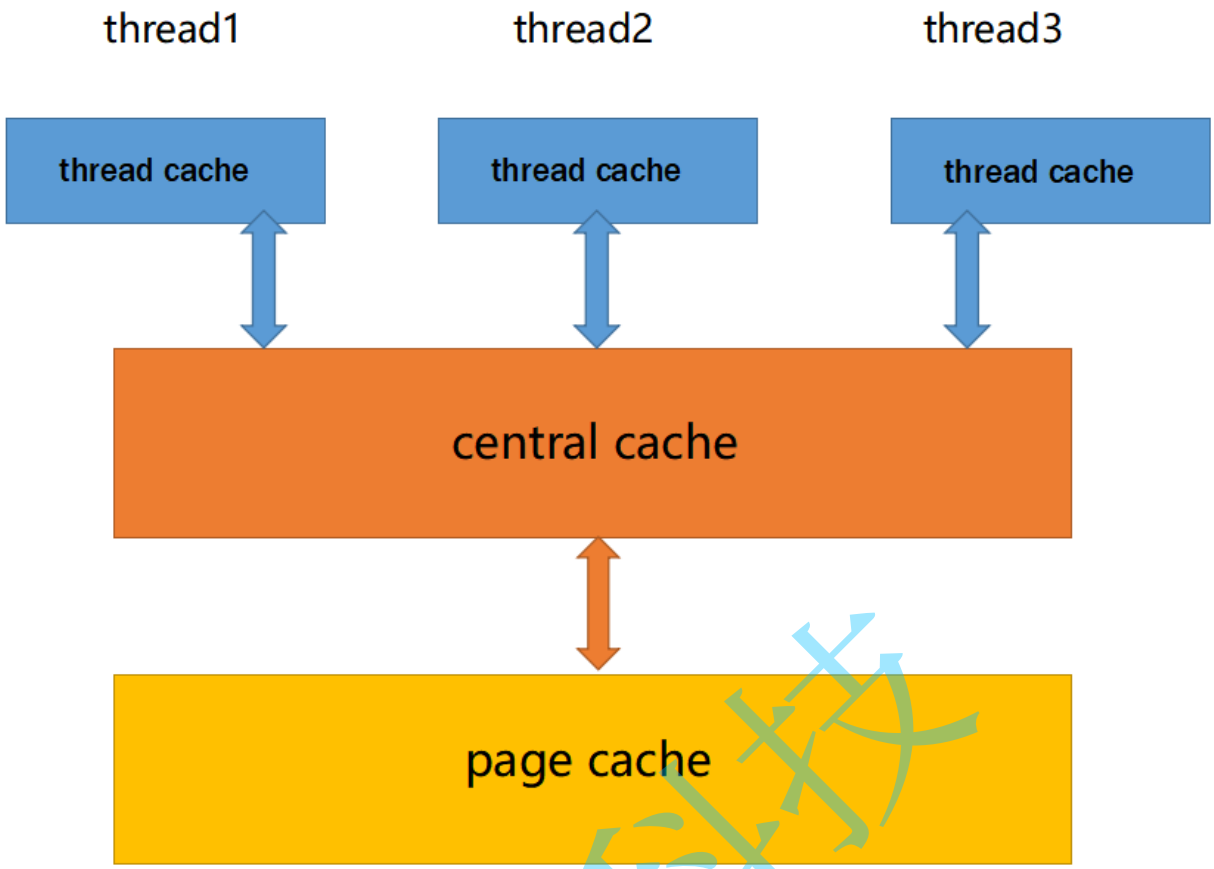
4.我们重点目标：并发内存池concurrent memory pool

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。所以这次我们实现的**内存池需要考虑以下几方面的问题。**

1. 内存碎片问题。
2. 性能问题。
3. 多核多线程环境下，锁竞争问题。

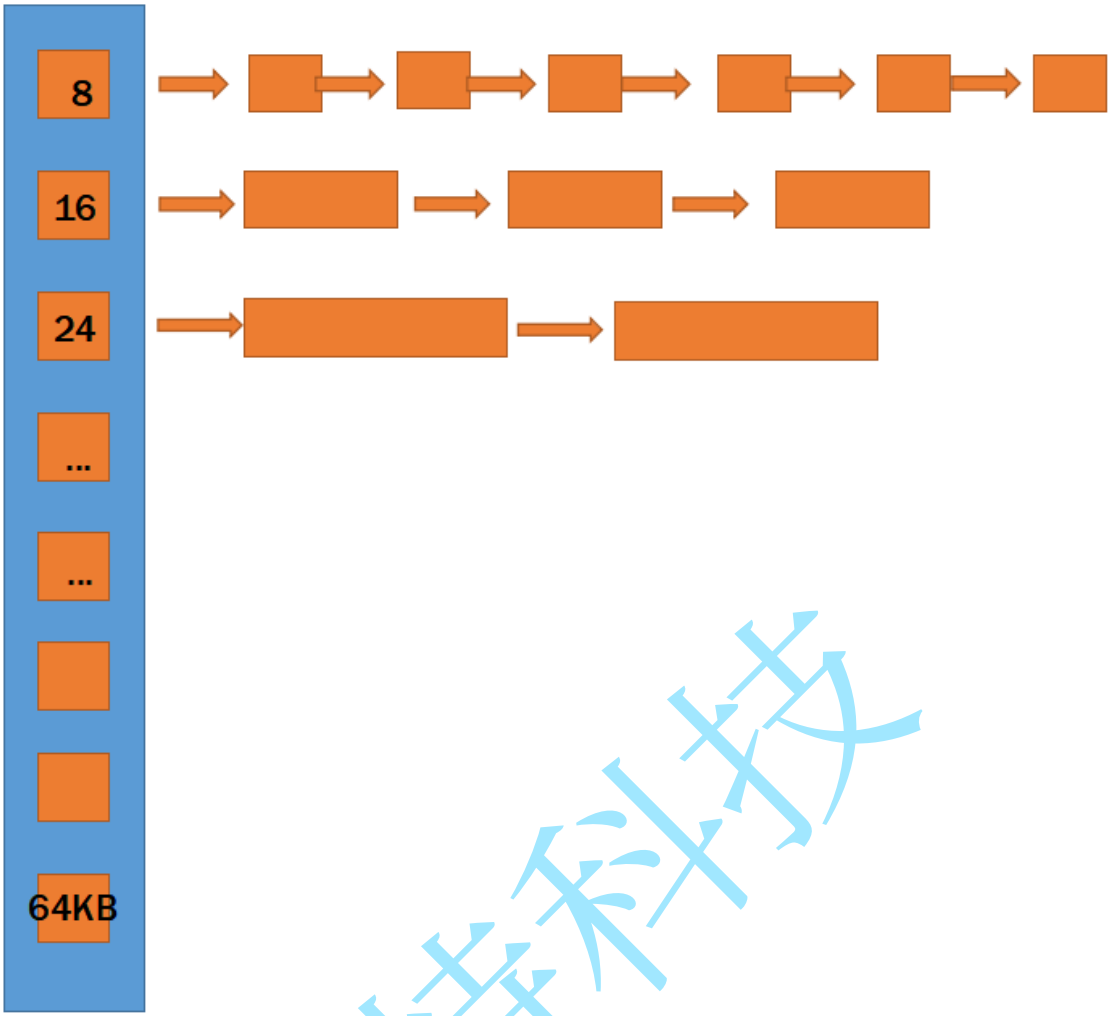
concurrent memory pool主要由以下3个部分构成：

1. **thread cache**：线程缓存是每个线程独有的，用于小于64k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**周期性的回收**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，不过一般情况下在这里取内存对象的效率非常高，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。**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。**



4.thread cache

```
// thread cache本质是由一个哈希映射的对象自由链表构成
class ThreadCache
{
public:
    // 申请和释放内存对象
    void* Allocate(size_t size);
    void Deallocate(void* ptr);

    // 从中心缓存获取对象
    void* FetchFromCentralCache(size_t index, size_t size);
    // 释放对象时，链表过长时，回收内存回到中心堆
    void ListTooLong(FreeList* list, size_t size);
private:
    FreeList _freeList[NLISTS];    // 自由链表
};
```



申请内存：

1. 当内存申请size<=64k时在thread cache中申请内存，计算size在自由链表中的位置，如果自由链表中有内存对象时，直接从FistList[i]中Pop一下对象，时间复杂度是O(1)，且没有锁竞争。
2. 当FreeList[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64k时将内存释放回thread cache，计算size在自由链表中的位置，将对象Push到FreeList[i].
2. 当链表的长度过长，则回收一部分内存对象到central cache。

线程TLS：

为了保证效率，我们使用thread local storage保存每个线程本地的ThreadCache的指针，这样大部分情况下申请释放内存是不需要锁的。

TLS分为静态的和动态的：

https://blog.csdn.net/evilswords/article/details/8191230

https://blog.csdn.net/yusiguyuan/article/details/22938671

对象大小的映射对齐：

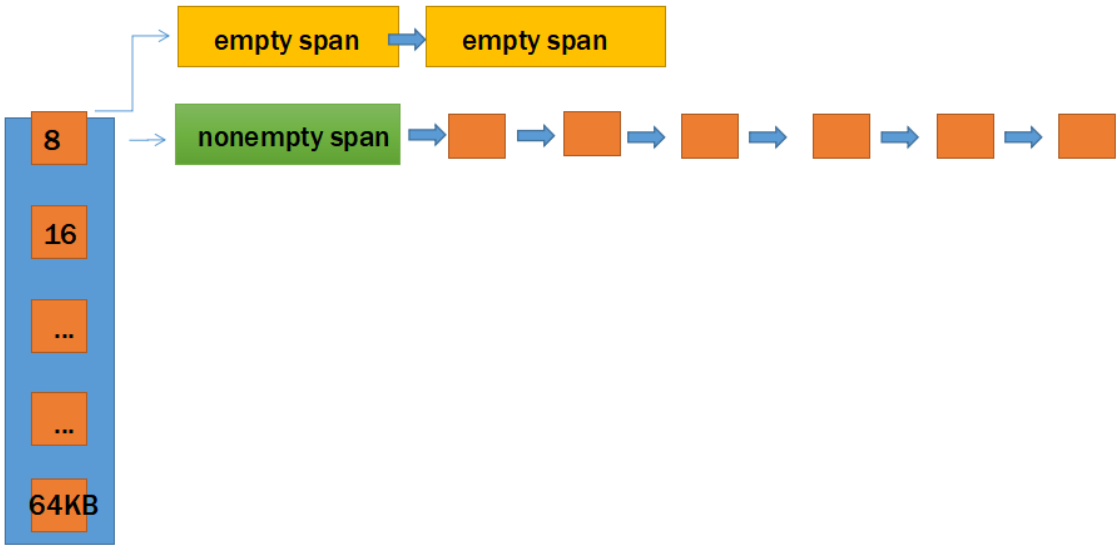
```
class SizeClass
{
public:
    // 控制在12%左右的内碎片浪费
    // [1,128]                  8byte对齐   freelist[0,16)
    // [129,1024]               16byte对齐      freelist[16,72)
    // [1025,8*1024]            128byte对齐     freelist[72,128)
    // [8*1024+1,64*1024]       512byte对齐   freelist[128,240)

    static inline size_t _RoundUp(size_t bytes, size_t align)
    {
        return (((bytes)+align - 1) & ~(align - 1));
    }

    // 对齐大小计算
    static inline size_t RoundUp(size_t bytes)
    {
        assert(bytes <= MAX_BYTES);

        if (bytes <= 128){
            return _RoundUp(bytes, 8);
        }
        else if (bytes <= 1024){
            return  _RoundUp(bytes, 16);
        }
        else if (bytes <= 8192){
            return  _RoundUp(bytes, 128);
        }
        else if (bytes <= 65536){
            return  _RoundUp(bytes, 512);
        }

        return -1;
    }

    static inline size_t _Index(size_t bytes, size_t align_shift)
    {
        return ((bytes + (1<<align_shift) - 1) >> align_shift) - 1;
    }

    // 映射的自由链表的位置
    static inline size_t Index(size_t bytes)
    {
        assert(bytes <= MAX_BYTES);

        // 每个区间有多少个链
        static int group_array[4] = {16, 56, 56, 112};
        if (bytes <= 128){
            return _Index(bytes, 3);
        }
        else if (bytes <= 1024){
            return _Index(bytes - 128, 4) + group_array[0];
        }
```

```
        else if (bytes <= 8192){
            return _Index(bytes - 1024, 7) + group_array[1] + group_array[0];
        }
        else if (bytes <= 65536){
            return _Index(bytes - 8192, 9) + group_array[2] + group_array[1] +
group_array[0];
        }

        assert(false);

        return -1;
    }

    static size_t NumMoveSize(size_t size)
    {
        if (size == 0)
            return 0;

        int num = static_cast<int>(MAX_BYTES / size);
        if (num < 2)
            num = 2;

        if (num > 512)
            num = 512;

        return num;
    }

    // 计算一次向系统获取几个页
    static size_t NumMovePage(size_t size)
    {
        size_t num = NumMoveSize(size);
        size_t npage = num*size;

        npage >>= 12;
        if (npage == 0)
            npage = 1;

        return npage;
    }
};
```


5.central cache

```
// 1.central cache本质是由一个哈希映射的span对象自由链表构成
// 2.每个映射大小的empty span挂在一个链表中，nonempty span挂在一个链表中
// 3.为了保证全局只有唯一的central cache，这个类被设计成了单例模式。
typedef size_t PageID;
struct Span
{
    PageID        _pageid = 0;              // Starting page number
```

```
    size_t        _n = 0;                   // Number of pages in span
    Span*         _next = nullptr;          // Used when in link list
    Span*         _prev = nullptr;          // Used when in link list

    void*         _start = nullptr;
    size_t        _use_count = 0;
    size_t        _objsize = 0;           // 对象大小
};

// 设计为单例模式
class CentralCache
{
public:
    static CentralCache* GetInstance(){
        return &_inst;
    }

    // 从中心缓存获取一定数量的对象给thread cache
    size_t FetchRangeObj(void*& start, void*& end, size_t n, size_t byte_size);

    // 将一定数量的对象释放到span跨度
    void ReleaseListToSpans(void* start, size_t byte_size);

    // 从page cache获取一个span
    Span* GetOneSpan(SpanList* list, size_t byte_size);
private:
    // 中心缓存自由链表
    SpanList _freeList[NLISTS];
private:
    CentralCache() = default;
    CentralCache(const CentralCache&) = delete;
    CentralCache& operator=(const CentralCache&) = delete;

    // 单例对象
    static CentralCache _inst;
};
```



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，central cache也有一个哈希映射的freelist，freelist中挂着span，从span中取出对象给thread cache，这个过程是需要加锁的。
2. central cache中没有非空的span时，则将空的span链在一起，向page cache申请一个span对象，span对象中是一些以页为单位的内存，切成需要的内存大小，并链接起来，挂到span中。
3. central cache的span中有一个use_count，分配一个对象给thread cache，就++use_count

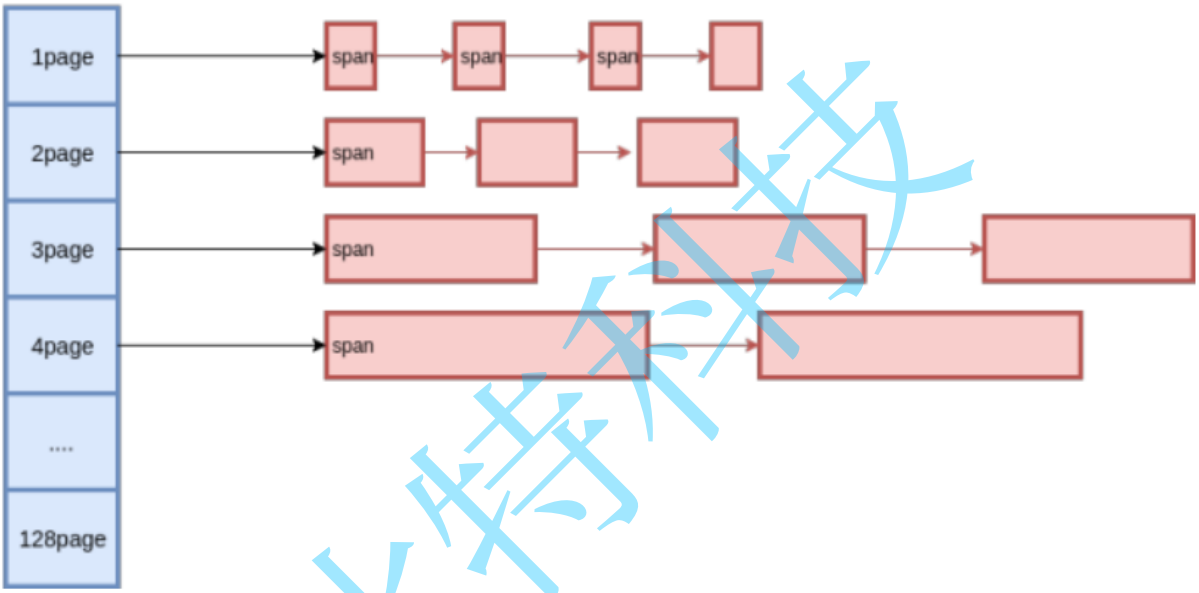
释放内存：

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

6.page cache

```
// 1.page cache是一个以页为单位的span自由链表
// 2.为了保证全局只有唯一的page cache，这个类被设计成了单例模式。
class PageCache
{
public:
    static PageCache* GetInstance()
    {
        return &_inst;
    }

    // 向系统申请k页内存挂到自由链表
    void SystemAllocPage(size_t k);

    // 申请一个新的span
    Span* _NewSpan(size_t k);
    Span* NewSpan(size_t k);

    // 获取从对象到span的映射
    Span* MapObjectToSpan(void* obj);

    // 释放空闲span回到Pagecache，并合并相邻的span
```

```
    void ReleaseSpanToPageCahce(Span* span);
private:
    SpanList _freelist[NPAGES];

private:
    PageCache()
    {}

    PageCache(const PageCache&) = delete;
    static PageCache _inst;

    std::unordered_map<PageID, Span*> _id_span_map;
    SpinLock _lock;
};
```



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4page，4page后面没有挂span，则向后面寻找更大的span，假设在10page位置找到一个span，则将10page span分裂为一个4page span和一个6page span。
2. 如果找到128 page都没有合适的span，则向系统使用mmap、brk或者是VirtualAlloc等方式申请128page span挂在自由链表中，再重复1中的过程。

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。

7.对比malloc的benchmark

```
void BenchmarkMalloc(size_t ntimes, size_t nworks, size_t rounds)
{
    std::vector<std::thread> vthread(nworks);

    size_t malloc_costtime = 0;
```

```
    size_t free_costtime = 0;

    for (size_t k = 0; k < nworks; ++k)
    {
        vthread[k] = std::thread([&, k]() {
            std::vector<void*> v;
            v.reserve(ntimes);

            for (size_t j = 0; j < rounds; ++j)
            {
                size_t begin1 = clock();
                for (size_t i = 0; i < ntimes; i++)
                {
                    v.push_back(malloc(1025));
                }
                size_t end1 = clock();

                size_t begin2 = clock();
                for (size_t i = 0; i < ntimes; i++)
                {
                    free(v[i]);
                }
                size_t end2 = clock();
                v.clear();

                malloc_costtime += end1 - begin1;
                free_costtime += end2 - begin2;
            }
        });
    }

    for (auto& t : vthread)
    {
        t.join();
    }

    printf("%u个线程并发执行%u轮次，每轮次malloc %u次：花费：%u ms\n",
        nworks, rounds, ntimes, malloc_costtime);

    printf("%u个线程并发执行%u轮次，每轮次free %u次：花费：%u ms\n",
        nworks, rounds, ntimes, free_costtime);

    printf("%u个线程并发malloc&free %u次，总计花费：%u ms\n",
        nworks, nworks*rounds*ntimes, malloc_costtime+free_costtime);
}


// 单轮次申请释放次数 线程数 轮次
void BenchmarkConcurrentMalloc(size_t ntimes, size_t nworks, size_t rounds)
{
    std::vector<std::thread> vthread(nworks);
    size_t malloc_costtime = 0;

    size_t free_costtime = 0;
```

```
    for (size_t k = 0; k < nworks; ++k)
    {
        vthread[k] = std::thread([&]() {
            std::vector<void*> v;
            v.reserve(ntimes);

            for (size_t j = 0; j < rounds; ++j)
            {
                size_t begin1 = clock();
                for (size_t i = 0; i < ntimes; i++)
                {
                    v.push_back(ConcurrentAlloc(1025));
                }
                size_t end1 = clock();

                size_t begin2 = clock();
                for (size_t i = 0; i < ntimes; i++)
                {
                    ConcurrentDealloc(v[i], 1025);
                }
                size_t end2 = clock();
                v.clear();

                malloc_costtime += end1 - begin1;
                free_costtime += end2 - begin2;
            }
        });
    }

    for (auto& t : vthread)
    {
        t.join();
    }

    printf("%u个线程并发执行%u轮次，每轮次concurrent alloc %u次：花费：%u ms\n",
        nworks, rounds, ntimes, malloc_costtime);

    printf("%u个线程并发执行%u轮次，每轮次concurrent dealloc %u次：花费：%u ms\n",
        nworks, rounds, ntimes, free_costtime);

    printf("%u个线程并发concurrent alloc&dealloc %u次，总计花费：%u ms\n",
        nworks, nworks*rounds*ntimes, malloc_costtime + free_costtime);
}

int main()
{
    cout << "==========================================================" << endl;
    BenchmarkMalloc(10000, 4, 10);
    cout << endl << endl;
    BenchmarkConcurrentMalloc(10000, 4, 10);
    cout << "==========================================================" << endl;
```

```
    return 0;
}
```

8.项目改进及不足

1. 内部结构的内存申请使用的new，可以替换成定长的单一内存池。
2. 如何替换程序中的malloc?

调研参考资料

[几个内存池库的对比]

[tcmalloc源码学习]

[TCMALLOC 源码阅读]