从零实现一个高并发内存池

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。

tcmalloc源代码

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。难度的话，如果比特难度满级的项目是5星的话，这个项目应该是4星。所以这里应该加一句，勇敢比特，不怕困难！！！

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。

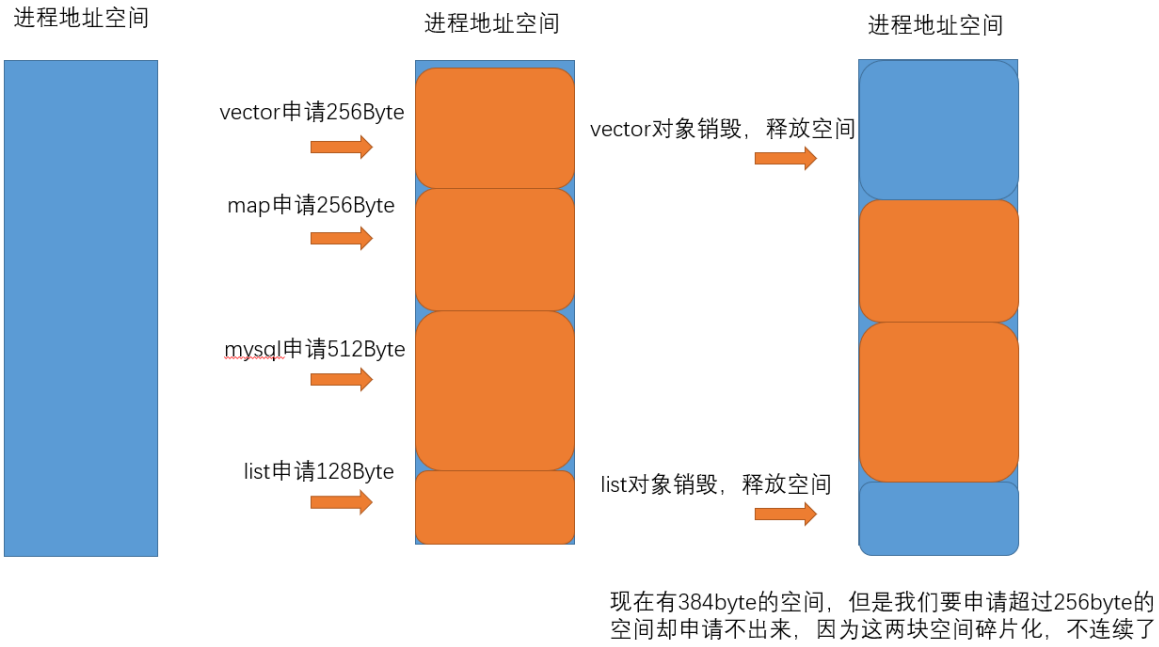
在计算机中，有很多使用“池”这种技术的地方，除了内存池，还有连接池、线程池、对象池等。以服务器上的线程池为例，它的主要思想是：先启动若干数量的线程，让它们处于睡眠状态，当接收到客户端的请求时，唤醒池中某个睡眠的线程，让它来处理客户端的请求，当处理完这个请求，线程又进入睡眠状态。

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

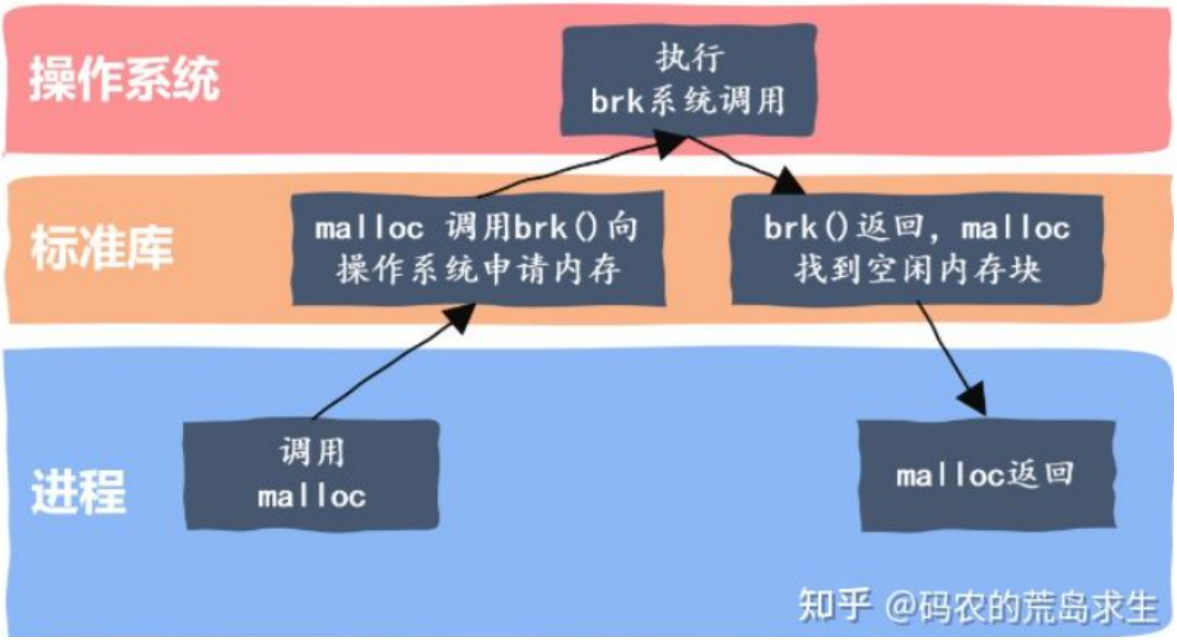
C/C++中我们要动态申请内存都是通过malloc去申请内存，但是我们要知道，实际我们不是直接去堆获取内存的，

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。

[一文了解，Linux内存管理，malloc，free 实现原理]

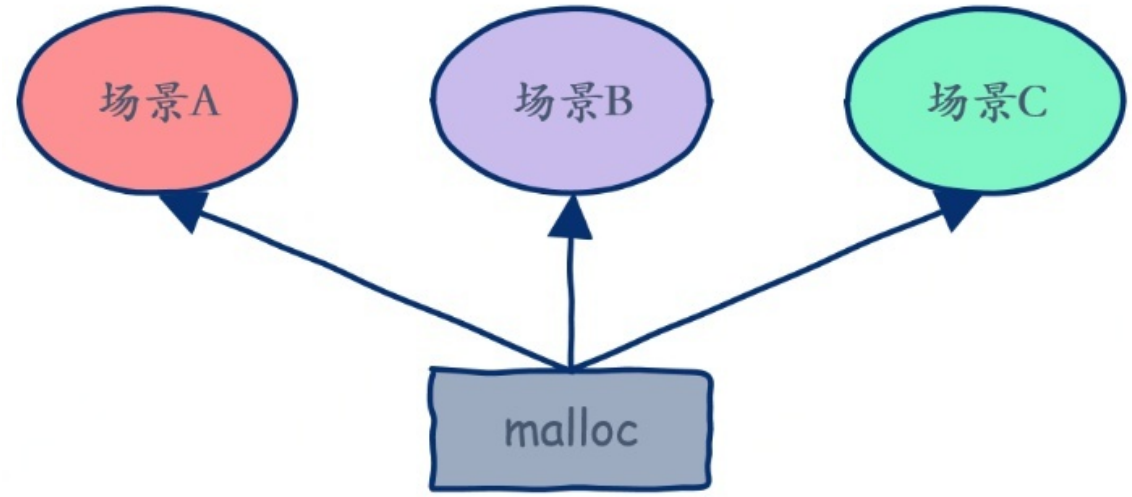
[malloc()背后的实现原理——内存池]

[malloc的底层实现（ptmalloc）]

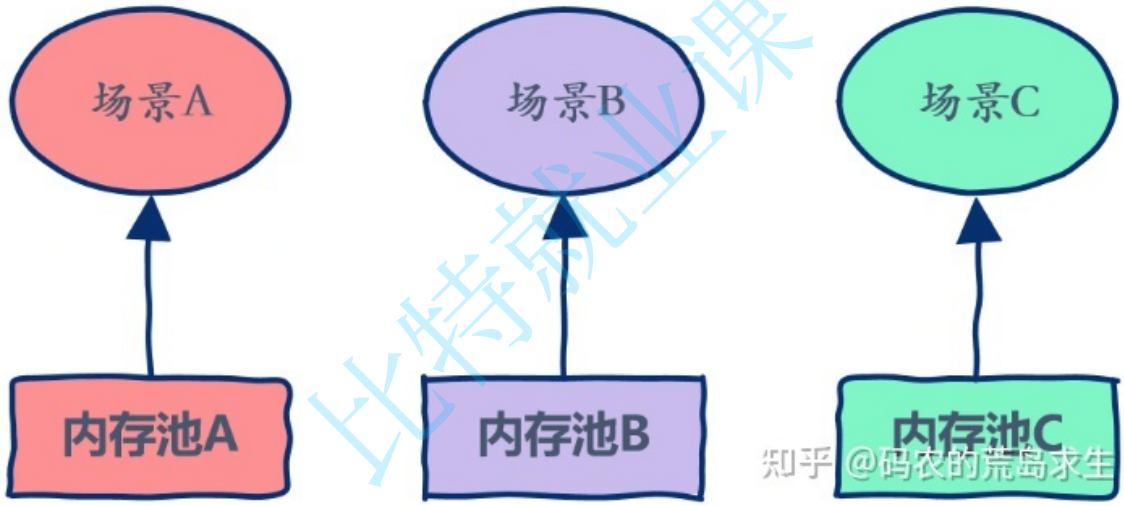


3.开胃菜--先设计一个定长的内存池

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。



VS



定长内存池设计

windows和Linux下如何直接向堆申请页为单位的大块内存：

VirtualAlloc

brk和mmap

```
// 直接去堆上按页申请空间
```

```
inline static void* SystemAlloc(size_t kpage)
{
#ifdef _WIN32
    void* ptr = VirtualAlloc(0, kpage*(1<<12),MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
#else
    // linux下brk mmap等
#endif

    if (ptr == nullptr)
        throw std::bad_alloc();

    return ptr;
}

template<class T>
class ObjectPool
{
public:
    T* New()
    {
        T* obj = nullptr;
        // 如果自由链表有对象，直接取一个
        if (_freeList)
        {
            obj = (T*)_freeList;
            _freeList = *((void**)_freeList);
        }
        else
        {
            if (_leftBytes < sizeof(T))
            {
                _leftBytes = 128 * 1024;
                //_memory = (char*)malloc(_leftBytes);
                _memory = (char*)SystemAlloc(_leftBytes);
                if (_memory == nullptr)
                {
                    throw std::bad_alloc();
                }
            }

            obj = (T*)_memory;
            size_t objSize = sizeof(T) < sizeof(void*) ? sizeof(void*) :
sizeof(T);
            _memory += objSize;
            _leftBytes -= objSize;
        }

        // 使用定位new调用T的构造函数初始化
        new(obj)T;
        return obj;
    }

    void Delete(T* obj)
    {
        // 显示调用的T的析构函数进行清理
        obj->~T();
```

```
        // 头插到freeList
        *((void**)obj) = _freeList;

        _freeList = obj;
    }

private:
    char* _memory = nullptr;     // 指向内存块的指针
    int   _leftBytes = 0;        // 内存块中剩余字节数
    void* _freeList = nullptr;   // 管理还回来的内存对象的自由链表
};

struct TreeNode
{
    int _val;
    TreeNode* _left;
    TreeNode* _right;

    TreeNode()
        :_val(0)
        , _left(nullptr)
        , _right(nullptr)
    {}
};

void TestObjectPool()
{
    // 申请释放的轮次
    const size_t Rounds = 3;

    // 每轮申请释放多少次
    const size_t N = 100000;

    size_t begin1 = clock();
    std::vector<TreeNode*> v1;
    v1.reserve(N);

    for (size_t j = 0; j < Rounds; ++j)
    {
        for (int i = 0; i < N; ++i)
        {
            v1.push_back(new TreeNode);
        }
        for (int i = 0; i < N; ++i)
        {
            delete v1[i];
        }
        v1.clear();
    }

    size_t end1 = clock();

    ObjectPool<TreeNode> TNPool;
    size_t begin2 = clock();
    std::vector<TreeNode*> v2;
    v2.reserve(N);

    for (size_t j = 0; j < Rounds; ++j)
```

```
    {
        for (int i = 0; i < N; ++i)
        {
            v2.push_back(TNPool.New());
        }
        for (int i = 0; i < 100000; ++i)
        {
            TNPool.Delete(v2[i]);
        }
        v2.clear();
    }
    size_t end2 = clock();

    cout <<"new cost time:" <<end1 - begin1 << endl;
    cout <<"object pool cost time:" <<end2 - begin2 << endl;
}
```

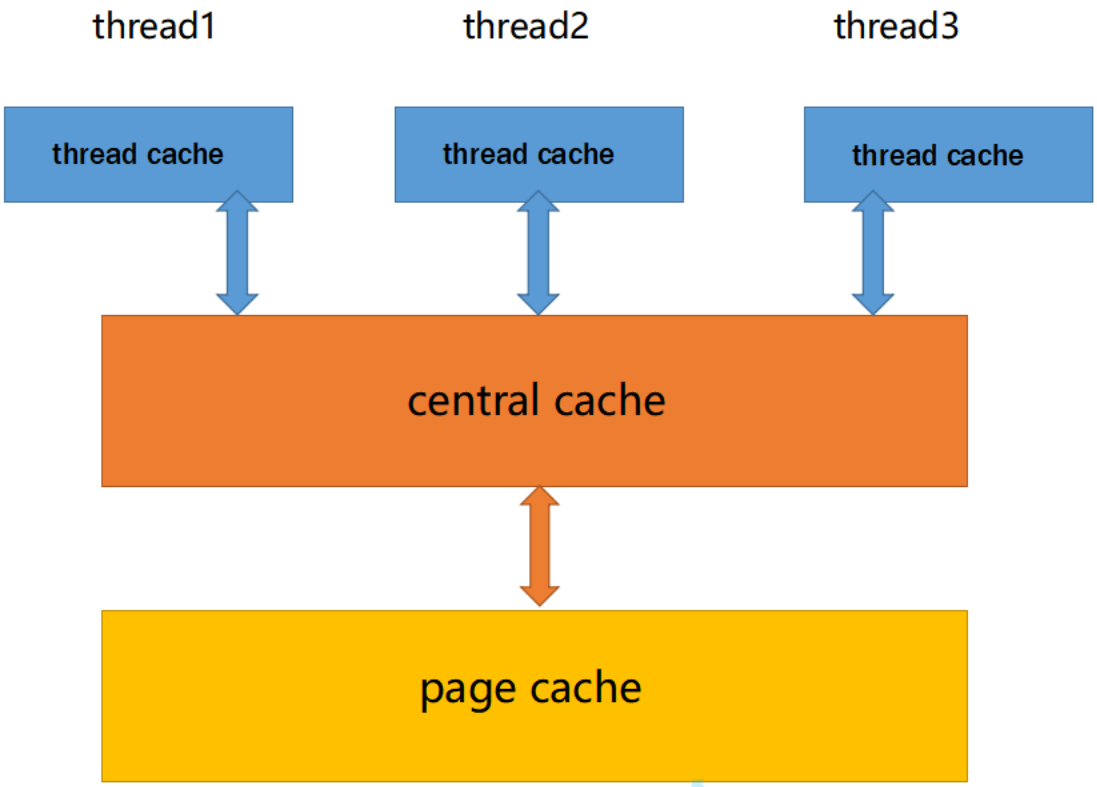
4.高并发内存池整体框架设计

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。**

1. 性能问题。
2. 多线程环境下，锁竞争问题。
3. 内存碎片问题。

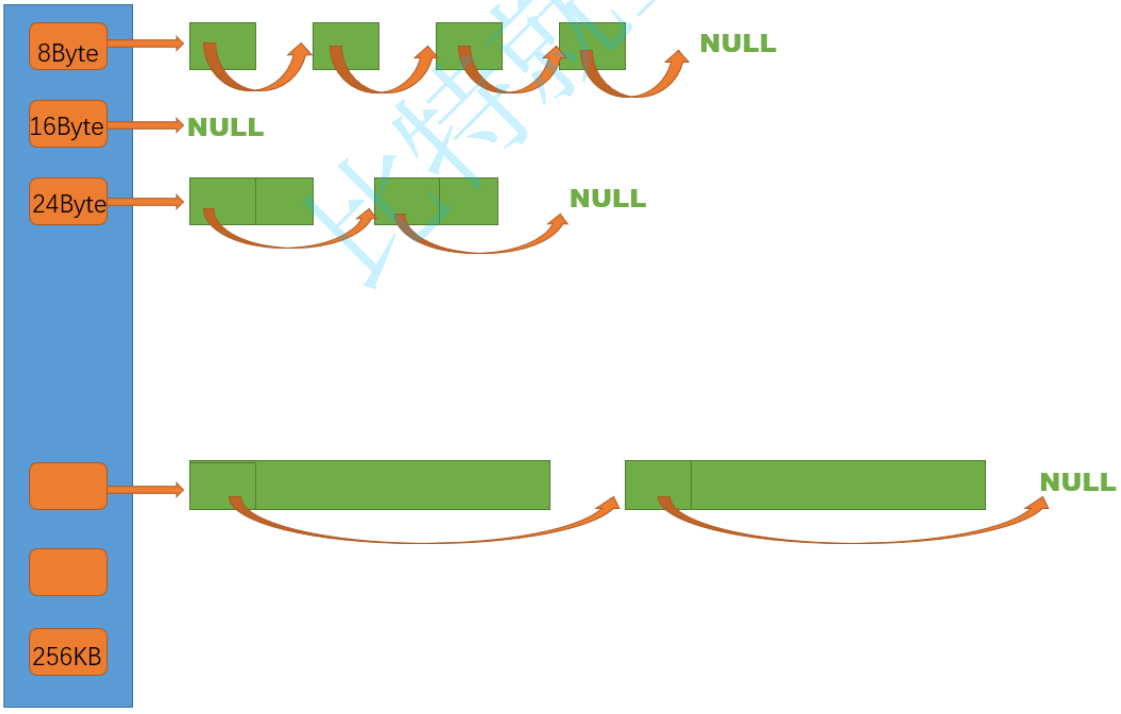
concurrent memory pool主要由以下3个部分构成：

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。**



5.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。

TLS--thread local storage：

linux gcc下 tls

thread cache代码框架：

```
// 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, size_t size);

    // 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，回收内存回到中心缓存
    void ListTooLong(FreeList& list, size_t size);
private:
    FreeList _freeLists[NFREELISTS];
};

// TLS thread local storage
static __declspec(thread) ThreadCache* tls_threadcache = nullptr;

// 管理小对象的自由链表
class FreeList
{
public:
    void PushRange(void* start, void* end, int n)
    {
        NextObj(end) = _head;
        _head = start;
        _size += n;
    }

    void PopRange(void*& start, void*& end, int n)
    {
        start = _head;
        for (int i = 0; i < n; ++i)
        {
            end = _head;
            _head = NextObj(_head);
        }

        NextObj(end) = nullptr;
        _size -= n;
    }

    // 头插
    void Push(void* obj)
    {
        NextObj(obj) = _head;
```

```
        _head = obj;
        _size += 1;
    }

    // 头删
    void* Pop()
    {
        void* obj = _head;
        _head = NextObj(_head);
        _size -= 1;

        return obj;
    }

    bool Empty()
    {
        if (_head && _size == 0)
        {
            int i = 0;
        }

        return _head == nullptr;
    }

    size_t MaxSize()
    {
        return _max_size;
    }

    void SetMaxSize(size_t n)
    {
        _max_size = n;
    }

    size_t Size()
    {
        return _size;
    }

private:
    void* _head = nullptr;
    size_t _max_size = 1;
    size_t _size = 0;
};
```


自由链表的哈希桶跟对象大小的映射关系

```
// 小于等于MAX_BYTES，就找thread cache申请
// 大于MAX_BYTES，就直接找page cache或者系统堆申请
static const size_t MAX_BYTES = 256 * 1024;
// thread cache 和 central cache自由链表哈希桶的表大小
static const size_t NFREELISTS = 208;
// page cache 管理span list哈希表大小
static const size_t NPAGES = 129;
// 页大小转换偏移，即一页定义为2^13,也就是8KB
static const size_t PAGE_SHIFT = 13;
```

```
// 地址大小类型，32位下是4byte类型，64位下是8byte类型
#ifdef _WIN32
    typedef size_t ADDRES_INT;
#else
    typedef unsigned long long ADDRES_INT;
#endif // _WIN32

// 页编号类型，32位下是4byte类型，64位下是8byte类型
#ifdef _WIN32
    typedef size_t PageID;
#else
    typedef unsigned long long PageID;
#endif // _WIN32

// 获取内存对象中存储的头4 or 8字节值，即链接的下一个对象的地址
inline void*& NextObj(void* obj)
{
    return *((void**)obj);
}

// 管理对齐和映射等关系
class SizeClass
{
public:
    // 整体控制在最多10%左右的内碎片浪费
    // [1,128]                  8byte对齐         freelist[0,16)
    // [128+1,1024]             16byte对齐        freelist[16,72)
    // [1024+1,8*1024]          128byte对齐       freelist[72,128)
    // [8*1024+1,64*1024]       1024byte对齐      freelist[128,184)
    // [64*1024+1,256*1024]     8*1024byte对齐    freelist[184,208)
    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)
    {
        if (bytes <= 128){
            return _RoundUp(bytes, 8);
        }
        else if (bytes <= 1024){
            return  _RoundUp(bytes, 16);
        }
        else if (bytes <= 8*1024){
            return  _RoundUp(bytes, 128);
        }
        else if (bytes <= 64*1024){
            return  _RoundUp(bytes, 1024);
        }
        else if (bytes <= 256*1024){
            return  _RoundUp(bytes, 8*1024);
        }
        else{
            return _RoundUp(bytes, 1 << PAGE_SHIFT);
        }

        return -1;
```

```
    }

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

    // 计算映射的哪一个自由链表桶
    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, 56 };
        if (bytes <= 128){
            return _Index(bytes, 3);
        }
        else if (bytes <= 1024){
            return _Index(bytes - 128, 4) + group_array[0];
        }
        else if (bytes <= 8*1024){
            return _Index(bytes - 1024, 7) + group_array[1] + group_array[0];
        }
        else if (bytes <= 64*1024){
            return _Index(bytes - 8*1024, 10) + group_array[2] + group_array[1]
+ group_array[0];
        }
        else if (bytes <= 256 * 1024){
            return _Index(bytes - 64 * 1024, 13) + group_array[3] +
group_array[2] + group_array[1] + group_array[0];
        }
        else{
            assert(false);
        }

        return -1;
    }

    // 一次从中心缓存获取多少个
    static size_t NumMoveSize(size_t size)
    {
        if (size == 0)
            return 0;

        // [2, 512]，一次批量移动多少个对象的(慢启动)上限值
        // 小对象一次批量上限高
        // 小对象一次批量上限低
        int num = 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
```

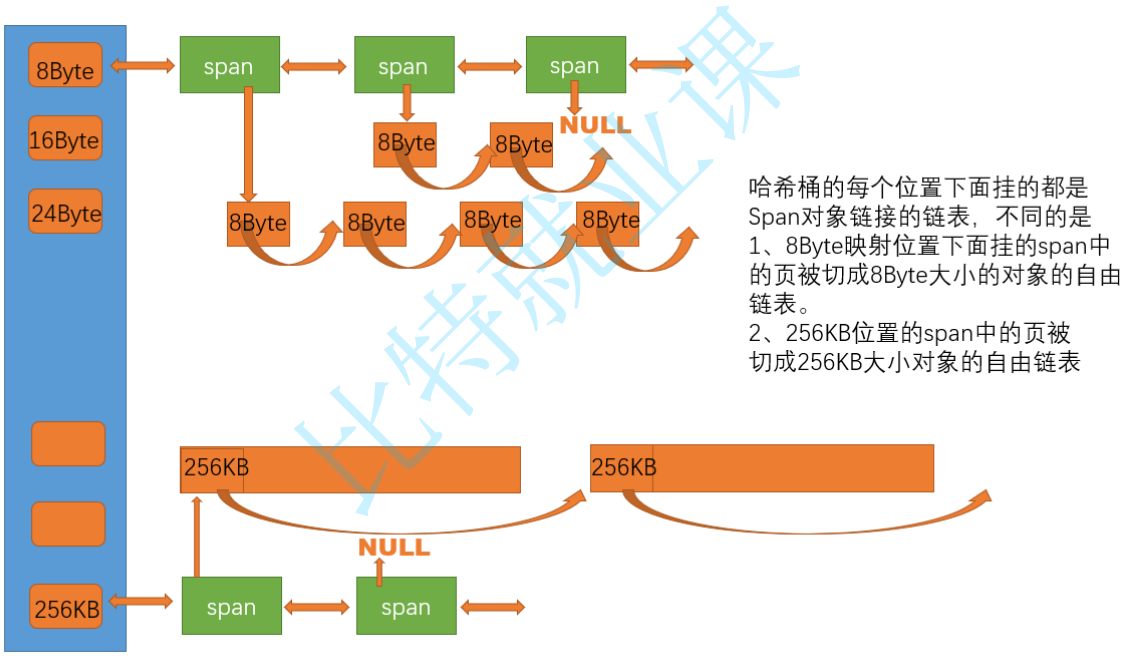
```
    // 单个对象 8byte
    // ...
    // 单个对象 256KB
    static size_t NumMovePage(size_t size)
    {
        size_t num = NumMoveSize(size);
        size_t npage = num*size;

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

        return npage;
    }
};
```

5.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协议拥塞控制的慢开始算法；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。

CentralCache 代码框架：

```
class CentralCache
{
public:
    static CentralCache* GetInstance() {return &_sInst;}

    // 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);

    // 从SpanList或者page cache获取一个span
    Span* GetOneSpan(SpanList& list, 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);
private:
    SpanList _spanLists[NFREELISTS]; // 按对齐方式映射

private:
    CentralCache()
    {}

    CentralCache(const CentralCache&) = delete;

    static CentralCache _sInst;
};
```

以页为单位的大内存管理span的定义及spanlist定义

```
// Span管理一个跨度的大块内存

// 管理以页为单位的大块内存
struct Span
{
    PageID _pageId = 0;   // 页号
    size_t _n = 0;        // 页的数量

    Span* _next = nullptr;
    Span* _prev = nullptr;

    void* _list = nullptr;  // 大块内存切小链接起来，这样回收回来的内存也方便链接
    size_t _usecount = 0;   // 使用计数，==0 说明所有对象都回来了

    size_t _objsize = 0;    // 切出来的单个对象的大小

    bool _isUse = false;    // 是否在使用
};

class SpanList
{
public:
    SpanList()
    {
```

```
        _head = new Span;
        _head->_next = _head;
        _head->_prev = _head;
    }

    Span* Begin()
    {
        return _head->_next;
    }

    Span* End()
    {
        return _head;
    }

    void PushFront(Span* span)
    {
        Insert(Begin(), span);
    }

    Span* PopFront()
    {
        Span* span = Begin();
        Erase(span);

        return span;
    }

    void Insert(Span* cur, Span* newspan)
    {
        Span* prev = cur->_prev;
        // prev newspan cur
        prev->_next = newspan;
        newspan->_prev = prev;

        newspan->_next = cur;
        cur->_prev = newspan;
    }

    void Erase(Span* cur)
    {
        assert(cur != _head);

        Span* prev = cur->_prev;
        Span* next = cur->_next;

        prev->_next = next;
        next->_prev = prev;
    }

    bool Empty()
    {
        return _head->_next == _head;
    }
private:
    Span* _head;

public:
```

```
    std::mutex _mtx;
};
```

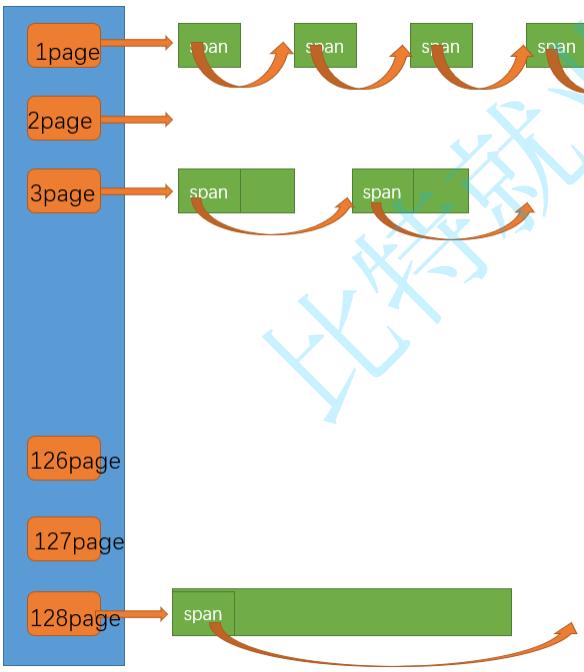
5.高并发内存池--page cache

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。**



PageCache 代码框架：

```
// 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 &_sInst;
    }

    // 向系统申请k页内存挂到自由链表
```

```
    void* SystemAllocPage(size_t k);

    Span* _NewSpan(size_t k);
    Span* NewSpan(size_t k);

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

    // 缓存和获取页号和这些页被切出去的小块内存的大小关系
    void CacheIdSize(Span* span, size_t size);
    size_t GetIdSize(size_t id);

    // 释放空闲span回到Pagecache，并合并相邻的span
    void ReleaseSpanToPageCache(Span* span);

    std::mutex _pageMtx;
private:
    SpanList _spanList[NPAGES]; // 按页数映射

    std::unordered_map<PageID, Span*> _idSpanMap;
    std::unordered_map<PageID, size_t> _idSizeMap;
private:
    PageCache()
    {}

    PageCache(const PageCache&) = delete;

    // 单例
    static PageCache _sInst;
};
```

windows和Linux下如何直接向堆申请页为单位的大块内存：

VirtualAlloc

brk和mmap

```
inline static void* SystemAlloc(size_t kpage)
{
#ifdef _WIN32
    void* ptr = VirtualAlloc(0, kpage*(1 << PAGE_SHIFT),
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
#else
    // brk mmap等
#endif

    if (ptr == nullptr)
        throw std::bad_alloc();

    return ptr;
}

inline static void SystemFree(void* ptr)
{
#ifdef _WIN32
    VirtualFree(ptr, 0, MEM_RELEASE);
#else
    // sbrk unmmap等
```

```
#endif
}
```

6.多线程并发环境下，对比malloc和ConcurrentAlloc申请和释放内存效率对比

```
void BenchmarkMalloc(size_t ntimes, size_t nworks, size_t rounds)
{
    std::vector<std::thread> vthread(nworks);
    std::atomic<size_t> malloc_costtime = 0;
    std::atomic<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(16));
                    //v.push_back(malloc((16 + i) % 8192 + 1));
                }
                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);
}
```

```cpp
// 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);
    std::atomic<size_t> malloc_costtime = 0;
    std::atomic<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(16));
                    //v.push_back(ConcurrentAlloc((16 + i) % 8192 + 1));
                }
                size_t end1 = clock();

                size_t begin2 = clock();
                for (size_t i = 0; i < ntimes; i++)
                {
                    ConcurrentFree(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()
{
    size_t n = 10000;
    cout << "==========================================================" <<
endl;
    BenchmarkConcurrentMalloc(n, 4, 10);
```

```
    cout << endl << endl;

    BenchmarkMalloc(n, 4, 10);
    cout << "==========================================================" <<
endl;

    return 0;
}
```

7.使用tcmalloc源码中实现基数树进行优化

```
// Single-level array
template <int BITS>
class TCMalloc_PageMap1 {
private:
    static const int LENGTH = 1 << BITS;
    void** array_;

public:
    typedef uintptr_t Number;

    explicit TCMalloc_PageMap1(void* (*allocator)(size_t)) {
        array_ = reinterpret_cast<void**>((*allocator)(sizeof(void*) << BITS));
        memset(array_, 0, sizeof(void*) << BITS);
    }

    // Return the current value for KEY.  Returns NULL if not yet set,
    // or if k is out of range.
    void* get(Number k) const {
        if ((k >> BITS) > 0) {
            return NULL;
        }
        return array_[k];
    }

    // REQUIRES "k" is in range "[0,2^BITS-1]".
    // REQUIRES "k" has been ensured before.
    //
    // Sets the value 'v' for key 'k'.
    void set(Number k, void* v) {
        array_[k] = v;
    }
};

// Two-level radix tree
template <int BITS>
class TCMalloc_PageMap2 {
private:
    // Put 32 entries in the root and (2^BITS)/32 entries in each leaf.
    static const int ROOT_BITS = 5;
    static const int ROOT_LENGTH = 1 << ROOT_BITS;

    static const int LEAF_BITS = BITS - ROOT_BITS;
    static const int LEAF_LENGTH = 1 << LEAF_BITS;

    // Leaf node
    struct Leaf {
```

```
        void* values[LEAF_LENGTH];
    };

    Leaf* root_[ROOT_LENGTH];             // Pointers to 32 child nodes
    void* (*allocator_)(size_t);          // Memory allocator

public:
    typedef uintptr_t Number;

    explicit TCMalloc_PageMap2(void* (*allocator)(size_t)) {
        allocator_ = allocator;
        memset(root_, 0, sizeof(root_));
    }

    void* get(Number k) const {
        const Number i1 = k >> LEAF_BITS;
        const Number i2 = k & (LEAF_LENGTH - 1);
        if ((k >> BITS) > 0 || root_[i1] == NULL) {
            return NULL;
        }
        return root_[i1]->values[i2];
    }

    void set(Number k, void* v) {
        const Number i1 = k >> LEAF_BITS;
        const Number i2 = k & (LEAF_LENGTH - 1);
        ASSERT(i1 < ROOT_LENGTH);
        root_[i1]->values[i2] = v;
    }

    bool Ensure(Number start, size_t n) {
        for (Number key = start; key <= start + n - 1;) {
            const Number i1 = key >> LEAF_BITS;

            // Check for overflow
            if (i1 >= ROOT_LENGTH)
                return false;

            // Make 2nd level node if necessary
            if (root_[i1] == NULL) {
                Leaf* leaf = reinterpret_cast<Leaf*>((*allocator_)
(sizeof(Leaf)));
                if (leaf == NULL) return false;
                memset(leaf, 0, sizeof(*leaf));
                root_[i1] = leaf;
            }

            // Advance key past whatever is covered by this leaf node
            key = ((key >> LEAF_BITS) + 1) << LEAF_BITS;
        }
        return true;
    }

    void PreallocateMoreMemory() {
        // Allocate enough to keep track of all possible pages
        Ensure(0, 1 << BITS);
    }
};
```

```
// Three-level radix tree
template <int BITS>
class TCMalloc_PageMap3 {
private:
    // How many bits should we consume at each interior level
    static const int INTERIOR_BITS = (BITS + 2) / 3; // Round-up
    static const int INTERIOR_LENGTH = 1 << INTERIOR_BITS;

    // How many bits should we consume at leaf level
    static const int LEAF_BITS = BITS - 2 * INTERIOR_BITS;
    static const int LEAF_LENGTH = 1 << LEAF_BITS;

    // Interior node
    struct Node {
        Node* ptrs[INTERIOR_LENGTH];
    };

    // Leaf node
    struct Leaf {
        void* values[LEAF_LENGTH];
    };

    Node* root_;                          // Root of radix tree
    void* (*allocator_)(size_t);          // Memory allocator

    Node* NewNode() {
        Node* result = reinterpret_cast<Node*>((*allocator_)(sizeof(Node)));
        if (result != NULL) {
            memset(result, 0, sizeof(*result));
        }
        return result;
    }

public:
    typedef uintptr_t Number;

    explicit TCMalloc_PageMap3(void* (*allocator)(size_t)) {
        allocator_ = allocator;
        root_ = NewNode();
    }

    void* get(Number k) const {
        const Number i1 = k >> (LEAF_BITS + INTERIOR_BITS);
        const Number i2 = (k >> LEAF_BITS) & (INTERIOR_LENGTH - 1);
        const Number i3 = k & (LEAF_LENGTH - 1);
        if ((k >> BITS) > 0 ||
            root_->ptrs[i1] == NULL || root_->ptrs[i1]->ptrs[i2] == NULL) {
            return NULL;
        }
        return reinterpret_cast<Leaf*>(root_->ptrs[i1]->ptrs[i2])->values[i3];
    }

    void set(Number k, void* v) {
        ASSERT(k >> BITS == 0);
        const Number i1 = k >> (LEAF_BITS + INTERIOR_BITS);
        const Number i2 = (k >> LEAF_BITS) & (INTERIOR_LENGTH - 1);
        const Number i3 = k & (LEAF_LENGTH - 1);
```

```
        reinterpret_cast<Leaf*>(root_->ptrs[i1]->ptrs[i2])->values[i3] = v;
    }

    bool Ensure(Number start, size_t n) {
        for (Number key = start; key <= start + n - 1;) {
            const Number i1 = key >> (LEAF_BITS + INTERIOR_BITS);
            const Number i2 = (key >> LEAF_BITS) & (INTERIOR_LENGTH - 1);

            // Check for overflow
            if (i1 >= INTERIOR_LENGTH || i2 >= INTERIOR_LENGTH)
                return false;

            // Make 2nd level node if necessary
            if (root_->ptrs[i1] == NULL) {
                Node* n = NewNode();
                if (n == NULL) return false;
                root_->ptrs[i1] = n;
            }

            // Make leaf node if necessary
            if (root_->ptrs[i1]->ptrs[i2] == NULL) {
                Leaf* leaf = reinterpret_cast<Leaf*>((*allocator_)
(sizeof(Leaf)));
                if (leaf == NULL) return false;
                memset(leaf, 0, sizeof(*leaf));
                root_->ptrs[i1]->ptrs[i2] = reinterpret_cast<Node*>(leaf);
            }

            // Advance key past whatever is covered by this leaf node
            key = ((key >> LEAF_BITS) + 1) << LEAF_BITS;
        }
        return true;
    }

    void PreallocateMoreMemory() {
    }
};
```


8.扩展学习及当前项目实现的不足

> 实际中我们测试了，当前实现的并发内存池比malloc/free是更加高效的，那么我们能否替换到系统调用malloc呢？实际上是可以的。
>
> - 不同平台替换方式不同。 基于unix的系统上的glibc，使用了weak alias的方式替换。具体来说是因为这些入口函数都被定义成了weak symbols，再加上gcc支持 alias attribute，所以替换就变成了这种通用形式：
>
> void* malloc(size_t size) **THROW** attribute__ ((alias (tc_malloc)))
>
> 因此所有malloc的调用都跳转到了tc_malloc的实现
>
> 具体参考这里：GCC **attribute** 之weak,alias属性
>
> 有些平台不支持这样的东西，需要使用hook的钩子技术来做。
>
> 关于hook请看这里：hook

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
>
> 如何设计内存池？ - 码农的荒岛求生的回答 - 知乎
>
> tcmalloc源代码