

线程池与内存池 软件 V1.0

操作手册

第一章 系统综述

1.1 概述

线程池与内存池软件 V1.0 是一个基于 c++ 的，集成了性能优于直接申请和释放线程的方式线程池技术和保证多线程下的内存一致性、能实现高并发、应对复杂场景（比如碎片化内存申请，超大内存管理等）的内存池技术的系统。您可以通过这个软件实现预先申请资源，实现进程级的资源管理，减少内存碎片，提高资源利用效率等需求。

1.2 功能简介

用户使用软件后，系统会根据用户的不同权限为其初始化不同的功能菜单，系统的功能如下所示：

- 线程池技术：先启动若干数量的线程，并让这些线程都处于睡眠状态，当客户端有一个新请求时，就会唤醒线程池中的某一个睡眠线程，让它来处理客户端的这个请求，当处理完这个请求后，线程又处于睡眠状态。
- 内存池技术：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。同时能满足多线程下内存一致性需求，并且能实现高并发、应对复杂场景。

1.3 运行环境

系统运行的推荐集成开发环境：Vscode、VectorC、GCC 等 c++ 环境。

第二章 线程池技术

2.1 功能概述

创建一个线程池，当客户端有一个新请求时，就会唤醒线程池中的某一个睡眠线程，让它来处理客户端的这个请求，当处理完这个请求后，线程又处于睡眠状态。

2.2 使用说明

线程池的源代码和测试用例的网址为 [GitHub - USTCSkywalker/Pool](https://github.com/USTCSkywalker/Pool): 工程实践。源代码为 ThreadPoo.h，测试用例的参考编写为 example.cpp。

测试用例代码如下：

```
1  #include <iostream>
2  #include <vector>
3  #include <chrono>
4
5  #include "ThreadPoo.h"
6
7  int main()
8  {
9
10     ThreadPoo pool(4);
11     std::vector< std::future<int> > results;
12
13     for(int i = 0; i < 8; ++i) {
14         results.emplace_back(
15             pool.enqueue([i] {
16                 std::cout << "hello " << i << std::endl;
17                 std::this_thread::sleep_for(std::chrono::seconds(1));
18                 std::cout << "world " << i << std::endl;
19                 return i*i;
20             })
21         );
22     }
23
24     for(auto && result: results)
25         std::cout << result.get() << ' ';
26     std::cout << std::endl;
27
28     return 0;
29 }
```

如上图所示，测试用例中须调用线程池源代码 ThreadPoo.h 来实现对线程池的使用。

2.3 ThreadPool.h 分析

ThreadPool 类中有:

5 个成员变量

- **std::vector< std::thread > workers** 用于存放线程的数组, 用 vector 容器保存
- **std::queue< std::function<void()> > tasks** 用于存放任务的队列, 用 queue 队列进行保存。任务类型为 std::function<void()>。因为 std::function 是通用多态函数封装器, 也就是说本质上任务队列中存放的是一个函数
- **std::mutex queue_mutex** 一个访问任务队列的互斥锁, 在插入任务或者线程取出任务都需要借助互斥锁进行安全访问
- **std::condition_variable condition** 一个用于通知线程任务队列状态的条件变量, 若有任务则通知线程可以执行, 否则进入 wait 状态
- **bool stop** 标识线程池的状态, 用于构造与析构中对线程池状态的了解

3 个成员函数

- **ThreadPool(size_t)** 线程池的构造函数
- **auto enqueue(F&& f, Args&&... args)** 将任务添加到线程池的任务队列中
- **~ThreadPool()** 线程池的析构函数

• 2.3.1 构造函数解析

构造函数定义为 inline。接收参数 threads 表示线程池中要创建多少个线程。初始化成员变量 stop 为 false, 即表示线程池启动着。然后进入 for 循环, 依次创建 threads 个线程, 并放入线程数组 workers 中。

在 vector 中, **emplace_back()** 成员函数的作用是在容器尾部插入一个对象, 作用效果与 **push_back()** 一样, 但是两者有略微差异, 即 **emplace_back(args)** 中放入的对象的参数, 而 **push_back(OBJ(args))** 中放入的是对象。即 **emplace_back()** 直接在容器中以传入的参数直接调用对象的构造函数构造新的对象, 而 **push_back()** 中先调用对象的构造函数构造一个临时对象, 再将临时对象拷贝到容器内存中。

所以, 上述 **workers.emplace_back()** 中, 我们传入的 lambda 表达式就是创建线程的 fun() 函数。lambda 表达式的格式为:

[捕获] (形参) 说明符(可选) 异常说明 attr -> 返回类型 { 函数体 }

所以上述 lambda 表达式为 **[捕获] { 函数体 }** 类型。该 lambda 表达式捕获线程池指针 **this** 用于在函数体中使用 (调用线程池成员变量 **stop**、**tasks** 等) 分析函数体, **for(;;)** 为一个死循环, 表示每个线程都会反复这样执行, 这其实每个线程池中的线程都会这样。在循环中, 先创建一个封装 **void()** 函数的 **std::function** 对象 **task**, 用于接收后续从任务队列中弹出的真实任务。所以, **{}** 起的作用就是在退出 **}** 时自动回释放线程池的 **queue_mutex**。在 **{}** 中, 我们先对任务队列加锁, 然后根据条件变量判断条件是否满足。为条件标量 **wait** 的运行机制, **wait** 在 **p** 为 **false** 的状态下, 才会进入 **wait(lock)** 状态。当前线程阻塞直至 **条件变量被通知**。所以 **p** 表示上述代码中的 lambda 表达式 **[this]{ return this->stop || !this->tasks.empty(); }**, 其中 **this->stop** 为 **false**, **!this->tasks.empty()** 也为 **false**。即其表示若线程池已停止或者任务队列中不为空, 则不会进入到 **wait** 状态。

由于刚开始创建线程池, 线程池表示未停止, 且任务队列为空, 所以每个线程都会进入到 **wait** 状态。在线程池刚刚创建, 所有的线程都阻塞在了此处, 即 **wait** 处。若后续条件变量来了通知, 线程就会继续往下进行。若线程池已经停止且任务队列为空, 则线程返回, 没必要进行死循环。这样, 将任务队列中的第一个任务用 **task** 标记, 然后将任务队列中该任务弹出。(此处线程实在获得了任务队列中的互斥锁的情况下进行的, 从上图可以看出, 在条件标量唤醒线程后, 线程在 **wait** 周期内得到了任务队列的互斥锁才会继续往下执行。所以最终只会有一个线程拿到任务, 不会发生惊群效应)。在退出了 **{}**, 我们队任务队列的所加的锁也释放了, 然后我们的线程就可以执行我们拿到的任务 **task** 了, 执行完毕之后, 线程又进入了死循环。

• 2.3.2 添加任务函数解析

`enqueue` 是一个模板函数，其类型形参为 `F` 与 `Args`。其中 `class... Args` 表示多个类型形参。`auto` 用于自动推导出 `enqueue` 的返回类型，函数的形参为 `(F&& f, Args&&... args)`，其中 `&&` 表示右值引用。表示接受一个 `F` 类型的 `f`，与若干个 `Args` 类型的 `args`。最终返回的是放在 `std::future` 中的 `F(Args...)` 返回类型的异步执行结果。

下面举个例子来帮助理解：

```
// 来自 packaged_task 的 future
std::packaged_task<int>(> task([]() { return 7; }); // 包装函数，将lambda表达式进行包装
std::future<int> f1 = task.get_future(); // 定义一个future对象f1，存放int型的值。此处已经表明：将task挂载到线程
std::thread(std::move(task)).detach(); // 将task函数挂载在线程上运行

f1.wait(); // f1等待异步结果的输入
f1.get(); // f1获取到的异步结果

struct S {
    double operator()(char, int&);
    float operator()(int) { return 1.0; }
};

std::result_of<S(char, int&)>::type d = 3.14; // d 拥有 double 类型，等价于double d = 3.14
std::result_of<S(int)>::type x = 3.14; // x 拥有 float 类型，等价于float x = 3.14
```

由上述小例子，我们已经知道 `std::packaged_task` 是一个包装函数，所以最终，`task` 指向了传递进来的函数。所以，`res` 会在异步执行完毕后即可获得所求。在新的作用于内加锁，若线程池已经停止，则抛出异常。否则，将 `task` 所指向的 `f(args)` 插入到 `tasks` 任务队列中。需要指出，这儿的 `emplace` 中传递的是构造函数的参数。

• 2.3.3 析构函数解析

在析构函数中，先对任务队列中加锁，将停止标记设置为 `true`，这样后续即使有新的插入任务操作也会执行失败。使用条件变量唤醒所有线程，所有线程都会往下执行。

在 `stop` 设置为 `true` 且任务队列中为空时，对应的线程进而跳出循环结束。将每个线程设置为 `join`，等到每个线程结束完毕后，主线程再退出。

• 2.3.4 主函数解析

```
ThreadPool pool(4); //创建一个线程池，池中线程为4
std::vector< std::future<int> > results; //创建一个保存std::future<int>的数组，用于存储4个异步线程的结果

for(int i = 0; i < 8; ++i) { //创建8个任务
    results.emplace_back( //一次保存每个异步结果
        pool.enqueue([i] { //将每个任务插入到任务队列中，每个任务的功能均为“打印+睡眠1s+打印+返回结果”
            std::cout << "hello " << i << std::endl;
            std::this_thread::sleep_for(std::chrono::seconds(1));
            std::cout << "world " << i << std::endl;
            return i*i;
        })
    );
}

for(auto && result: results) //一次取出保存在results中的异步结果
    std::cout << result.get() << ' ';
std::cout << std::endl;
```

2.4 测试样例运行结果



```
hello 0hello 2
hello 3
hello 1

world 2
world 3
hello 4
hello 5
world 0
hello 0 6
world 1
hello 1 7
4 9 world world 7
world 5
6
world 4
16 25 36 49
```

第三章 内存池技术

3.1 功能概述

预先从操作系统申请一块足够大内存用以创建内存池。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。同时能满足多线程下内存一致性需求，并且能实现高并发、应对复杂场景。

3.2 使用说明

内存池的源代码和测试用例的网址为 [GitHub - USTCSkywalker/Pool](https://github.com/USTCSkywalker/Pool): 工程实践。

高并发内存池（concurrent memory pool）需要考虑以下几方面的问题：1. 性能问题。2. 内存碎片问题。3. 多线程环境下，锁竞争问题。其主要由三个部分组成：

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。

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。

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。

3.3 thread cache

thread cache 是哈希桶结构。每个桶是由一个或多个定长内存块为对象映射的自由链表，不同的桶（内存块起始指针）映射至不同的下标，相同大的内存块儿放入到同一个桶下。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。此部分的实现源码见 github 仓库中的 ThreadCache.h、ThreadCache.cpp、common.h、ConcurrentAlloc.h、Unittest.cpp。

• 3.3.1 整体设计

FreeList：管理切分好的小对象(内存块)的自由链表。起初链表下一个内存块都没有的，是下一层中心池将一个或多个页分成一个个对应的对齐内存块分配给线程内存池的。

为了记录下一个内存块的地址，依然采取将内存块头 4/8 个字节存储。强转成二级指针，32 位下是 4，解引用就等于取到了头 4 个字节，就可以对其赋值。

thread cache 是一个 FreeList 的自由链表数组。

内存的申请和释放：

内存申请时，符合以下规则：1)当内存申请 $\text{size} \leq 256\text{KB}$ ，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。

释放内存时，符合以下规则：1)当释放内存小于 256 KB 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。

• 3.3.2 哈希桶映射对其规则

不同的区域，采用不同大小内存块（对齐数），作等差数列。

字节数	对齐数	哈希桶下标
[1, 128]	8	[0, 16)
[128 + 1, 1024]	16	[16, 72)
[1024 + 1, 8×1024]	128	[72, 128)
[$8 \times 1024 + 1$, 64×1024]	1024	[128, 184)
[$64 \times 1024 + 1$, 256×1024]	8×1024	[184, 208)

• 3.3.3 TLS 无锁访问

Thread Local Storage（线程局部存储）TLS：线程局部存储（TLS），是一种变量的存储方法，这个变量在它所在的线程内是全局可访问的，但是不能被其他线程访问到，这样就保持了数据的线程独立性。而熟知的全局变量，是所有线程都可以访问的，这样就不可避免需要锁来控制，增加了控制成本和代码复杂度。

在 `thread_cache` 头文件中使用 TLS 技术，采用静态链接。有 windows/和 Linux 下的使用方法。Windows 下：static __declspec(thread) ThreadCache* pTLSThreadCache = nullptr;

让线程调用 `thread_cache` 线程池的头文件：ConcurrentAlloc.h，让每个线程获取自己独立的 thread cache。

3.4 Central Cache

• 3.4.1 整体设计

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，通过线程内存池要的内存块快速定位到去哪个桶下面取。不同的是他的每个哈希桶位置挂是 SpanList 链表结构。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一个小内存块对象，挂在 span 的自由链表中。头 4/8 个字节放下一个小的内存块的地址。

线程内存池 Thread Cache 要内存的时候便从 Central Cache 对应桶下的 span 的自由链表中取。一个 span 可以管理一个或多个页，每个页都有页号，知道页号就知道地址。（相对虚拟内存 0x000000 开始的）

内存申请和释放：

内存申请时符合以下规则：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（线程池向中心池要多少内存块合适）；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，Thread Cache 有需要就从 Central Cache 的 span 中取。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 threadcache，就++use_count。

内存释放时符合以下规则：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。

• 3.4.2 Span、SpanList

Span 管理多个连续页的大块内存的跨度结构。

```
struct Span
{
    PAGE_ID _pageId = 0; // 大块内存起始页的页号
    size_t _n = 0;       // 页的数量

    Span* _next = nullptr; // 双向链表的结构
    Span* _prev = nullptr;

    size_t _useCount = 0; // 切好小块内存，被分配给thread cache的计数
    void* _freeList = nullptr; // 切好的小块内存的自由链表
};
```

需要对页号的类型进行处理：64 位下，8KB 一页分会超出整形的最大值。_Win32 没有定义 _Win64。

```
#ifdef _WIN64
    typedef unsigned long long PAGE_ID;
#elif _WIN32
    typedef size_t PAGE_ID;
#else
    //linux
#endif
```

SpanList 是一个双向带头循环链表 当我们需要将某个 span 归还给 page cache 时，就可以很方便的将该 span 从双链表结构中移出。如果用单链表结构的话就比较麻烦了，因为单链表在删除时，需要知道当前结点的前一个结点。桶锁，多个线程找到中心池对应桶时，第一个先到的就该上锁。

• 3.4.3 Cenctral Cache 代码框架

Central Cache 至少提供一个提供给线程内存池 Thread Cache 分配内存的接口。接口的设计：1)参数：你要多少个多大的内存块，我可以通过对齐后的内存块算出下标（有相同的映射规则）。2)span 下是切分好的一个内存块链表，因此我可以返回给你一段地址区间，需要头、尾两个输出型参数。3)返回值：你想要那么多个，但对应的 SpanList 下的某一个 Span 里面的 _freelist 中实际只有这么点儿。

Central Cache 的第二个接口：查看该链表下哪个 span 可以用，返回这个 span；如果没

有则向 Page Cache 申请。

• 3.4.4 Thread Cache 代码补充

线程内存池 Thread Cache 向 Central Cache 索要申请采取的是慢开始反馈调节算法。

当 thread cache 向 central cache 申请内存时，如果申请的是较小的对象，那么可以多给一点，但如果申请的是较大的对象，就可以少给点。

通过编写函数，我们就可以根据所需申请的对象的大小计算出具体给出的对象个数，并且可以将给出的对象个数控制到 2~512 个之间。也就是说，就算 thread cache 要申请的对象再小，最多一次性给出 512 个对象；就算 thread cache 要申请的对象再大，至少一次性给出 2 个对象。

根据慢开始算法，就算申请的是小对象，一次性给出 512 个也是比较多的，基于这个原因，向 Thread Cache 的 class FreeList 中添加一个成员变量，记录批量向 Thread Cache 每次申请的内存块个数 size_t _amountCenterSz。

当 thread cache 申请对象时，我们会比较 _amountCenterSz 和计算得出的值，取出其中的较小值作为本次申请对象的个数。此外，如果本次采用的是 _amountCenterSz 的值，那么还会将 thread cache 中该自由链表的 _amountCenterSz 的值进行加一。

当线程内存池只拿到了一个；直接返回给外面用；如果拿到了多个，则返回一个；剩下的就插入到线程内存池的自由链表中。因此需要给 Thread Cache 的 class FreeList 增加一个范围插入的接口。

• 3.4.5 CenterCache 代码实现

分配指定个数和容量的内存块给 Thread Cache：取到一个 span 以后；虽然 Thread Cache 指定那么多，但 Central Cache 不一定够，所以不够的情况下，span 下有多少个内存块就给几个。另外也说明只要获取到了对应的 span，那么 span 里面的 _freelist 不为空，至少有一个小的内存块。

3.5 PageCaChe

• 3.5.1 整体设计

central cache 和 page cache 的核心结构都是 spanlist 的哈希桶。但是他们是本质区别的，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，采用的是直接定址法。这里我们就最大挂 128 页的 span，为了让桶号与页号对应起来，我们可以将第 0 号桶空出来不用，因此我们需要将哈希桶的个数设置为 129。

内存申请与释放：

内存申请符合以下规则：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 pagespan 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) 当 central cache 没有 span 时，向 page cache 申请的是某一固定页数的 span，而如何切分申请到的这个 span 就应该由 central cache 自己来决定。

内存释放符合以下规则：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。

• 3.5.2 CenterCache 代码补充

从中心池的桶下要小内存块，遍历这个链表，看有没有可用的 span；没有则向 Page Cache 要。获取到以后，把页切成对应小块内存挂在 span 里面的自由链表上。因为要遍历 SpanList，其提供开始和结束的 span 地址。

可以根据具体所需对象的大小来决定，central cache 一次应该向 page cache 申请几页的内存块。central cache 向 page cache 申请内存时，要求申请到的内存尽量能够满足 thread cache 向 central cache 申请时的上限。

进入到 Page Cache 申请 span 时**要将桶锁释放**，用上 PageCache 的锁。因为此时有可能别的线程会到桶下释放内存块。新到的 span 插入到桶里时再获取锁。而将新的 Span 切分时不用加锁，因为此刻是刚申请到的 span，其他线程看不到这个 span。

• 3.5.3 Page Cache 代码实现

直接映射、单例，大锁。

```
// 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;

// page cache 管理span list哈希表的大小
static const size_t NPAGES = 129;

#include "Common.h"

// 饿汉模式 给把大锁
class PageCache
{
private:
    SpanList _pagelists[NPAGES]; //直接定址法
    static PageCache _Inst;

    PageCache() {}
    PageCache(const PageCache&) = delete;
public:
    static PageCache& GetPageInstance()
    {
        return _Inst;
    }
    //获取一个k页的span
    Span* NewSpan(size_t k);
    mutex _pagemtx;
};
```

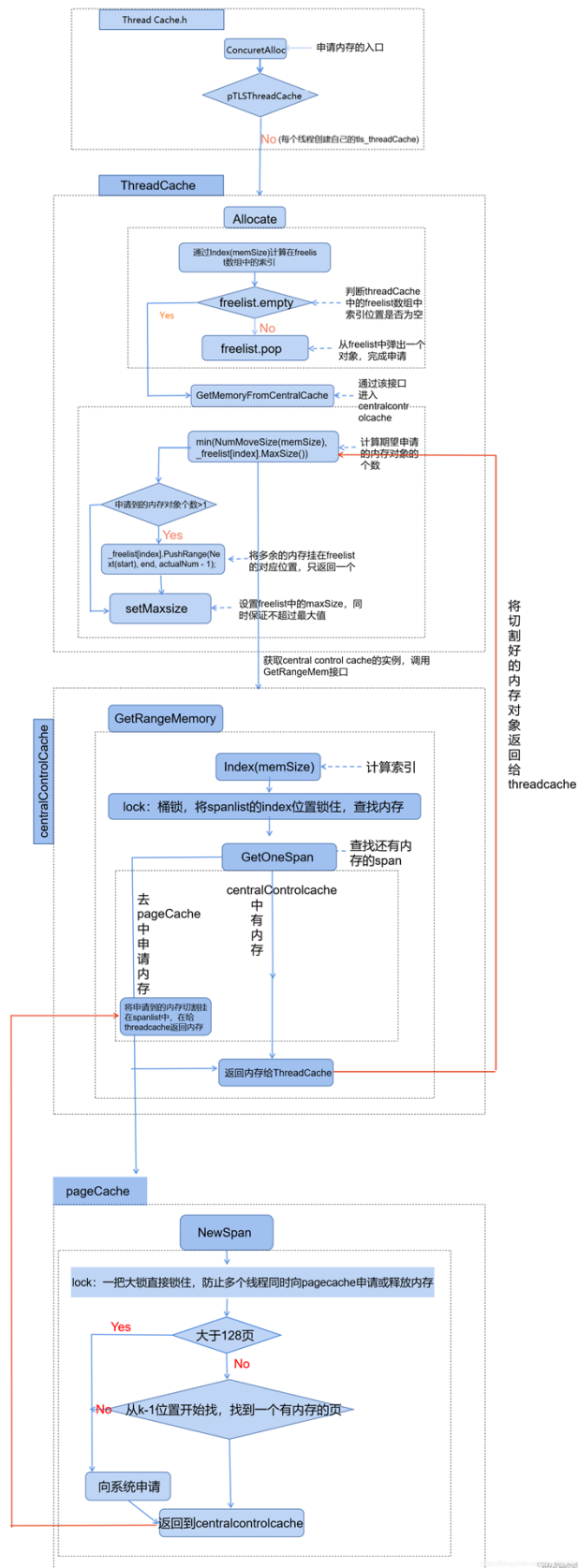
当前桶没有，则向下一个桶遍历。把大页的分成 k 页和 n-k 的，插入到对应的桶里面。都没有则向系统堆申请一个 128 页的内存块，调用系统接口。返回的是地址，根据虚拟内存的特性转换成对应的页号，把分裂的页数插入到对应的桶下面。


```
1 #include "PageCche.h"
2 PageCache PageCache::_Inst;
3
4 //获取一个K页的span
5 Span* PageCache::NewSpan(size_t k) //0位置空出来
6 {
7     assert(k > 0 && k < NPAGES);
8     //先检查第K个 桶有没有span
9     if (!_pageLists[k].Empty())
10     {
11         return _pageLists[k].PopFront();
12     }
13     //检查后面的桶里面有没有span, 如果有, 大page的span可以进行切分
14     for (size_t i = k+1; i < NPAGES; i++)
15     {
16         if (!_pageLists[i].Empty())
17         {
18             //k页的span返回, nspan对应桶的下面,
19             Span* nspan = _pageLists[i].PopFront();
20             Span* kspan = new Span;
21             //nspan 头切
22             kspan->_pageid = nspan->_pageid;
23             kspan->_n = k;
24
25             //nspan的原起始页号是100, 要走两页, 则页号加2, 页数减二
26             nspan->_pageid += k;
27             nspan->_n -= k;
28
29             _pageLists[nspan->_n].PushFront(nspan);
30             return kspan;
31         }
32     }
33     //最开始啥都没有或者没找到, 就向系统要一块大的128个页的。
34     Span* bigspan = new Span;
35     //进程地址空间返回的都是按页为对齐申请的, 算页号就是除
36     void* ptr = SystemAlloc(NPAGES - 1);
37     bigspan->_pageid = (PAGE_ID)ptr >> PAGE_SHIFT;
38     bigspan->_n = NPAGES - 1;
39     _pageLists[bigspan->_n].PushFront(bigspan);
40
41     return NewSpan(k); //复用
42 }
```

3.6 其他功能实现

• 3.6.1 申请内存联调

测试内存块是否对齐。测试起初获取 8 字节的内存, 获取 1024 次不释放, 到第 1025 次会不会再去下两层申请。其逻辑框架如下:



• 3.6.2 Thread Cache 回收内存

在 class FreeList 增加一个链表长度的计数。只要有新的内存块回来，就加加；有线程取走就减减。

如果 thread cache 某个桶当中自由链表的长度超过它一次批量向 central cache 申请的对象个数，那么此时我们就要把该自由链表当中的这些对象还给 central cache。因此 FreeList 增加范围弹出的接口。

当自由链表的长度大于一次批量申请的对象时，我们具体的做法就是，从该自由链表中取出一次批量个数的对象，然后将取出的这些对象还给 central cache 中对应的 span 即可。tcmalloc 不仅检测单个链表长度，还会计算整个 thread cache，如果总的内存大于 2m 就释放给 Central Cache。

• 3.6.3 Central Cache 回收内存

当 Central Cache 收到这些小的内存块时，怎么知道该插入到 Central Cache 的哪个桶里呢？因为有着和 Thread Cache 同样的映射规则，根据回收内存的大小计算出下标。

那如何让这些小的内存块还给对应的 span 呢？我们知道小内存块的地址，就可以知道它的页号。如果增加页号和 span* 的映射，就能很方便的找到属于哪个 span 了。

在 Thread Cache 还对象给 Central Cache 的过程中，如果 Central Cache 中某个 span 的 _useCount 减到 0 时，说明这个 span 分配出去的对象全部都还回来了，那么此时就可以将这个 span 再进一步还给 Page Cache。

• 3.6.4 Page Cache 回收内存

如果 central cache 中有某个 span 的 _useCount 减到 0 了，那么 central cache 就需要将这个 span 还给 page cache 了。但实际为了缓解内存碎片的问题，page cache 还需要尝试将还回来的 span 与其他空闲的 span 进行合并。

我们不能通过 span 结构当中的 _useCount 成员，来判断某个 span 到底是在 central cache 还是在 page cache。因为当 central cache 刚向 page cache 申请到一个 span 时，这个 span 的 _useCount 就是等于 0 的，这时可能当正在对该 span 进行切分的时候，page cache 就把这个 span 拿去进行合并了，这显然是不合理的。

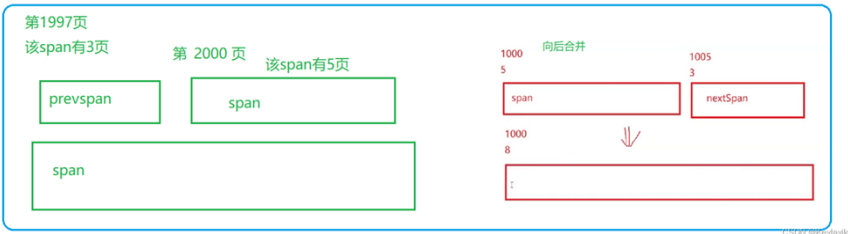
因而在 span 结构中再增加一个 _isUse 成员，用于标记这个 span 是否正在被使用，而当一个 span 结构被创建时我们默认该 span 是没有被使用的。

```
1 struct Span
2 {
3     PAGE_ID _pageid = 0; // 页号
4     size_t _n = 0;      // 几个页
5
6     Span* _prev; // 双链表 方便不用的span释放给 page-cache
7     Span* _next;
8
9     void* _freelist = nullptr; // 存储一个页切分的一个个小的内存块
10    size_t _useCount = 0;      // 记录central cache 分给 thread cache 的内存数量
11
12    bool isUse = false;      // 当前span是否正在被使用。
13};
```

由于只在 Page Cache 返回一个 span 给 Central Cache 时，对 span 中的页号与 span* 进行了映射；Page Cache 在向前后搜索时，空闲的不一定是分走的 span 页号，因而也需要将余下没被分走的 n-k 页的 span 进行映射。那么需要将所有的页号与 span 地址进行映射吗？假如：Central Cache 取走了 5 页：页号 1000 — 1004；余下 123 页的 span：页号 1005----1127。此刻取走的 5 页收回，要向后进行搜索合并，那么只需要知道 1005 相关联的 span 是否在使用；因为页号 1005----1127 的页都在一个 span 中。因而只需要将一个 span 的起始页号和尾部页号与 span 地址进行映射就可以了。

```
//获取一个k页的span
Span* PageCache::NewSpan(size_t k) //0位置空出来
{
    //...
    //检查后面的桶里面有没有span, 如果有, 大page的span可以进行切分
    for (size_t i = k + 1; i < NPAGES; i++)
    {
        if (!_pageLists[i].Empty())
        {
            //...
            _idMapSpan[nspan->_pageid] = nspan;
            _idMapSpan[nspan->_pageid + nspan->_n - 1] = nspan;
        }
    }
    //...
    return NewSpan(k);
}
```

在进行前后页合并时，还要注意，如果前后两个 span 页数和大于 128 页，则不应合并。合法合并的 span 插入到对应的坐标下，并将首尾页号进行映射。由于每个 span 结构是 new 出来的，注意释放。



• 3.6.5 释放内存联调

ConcurrentFree 函数：最终效果应该向库函数那样，只需要一个指针。

至此，内存池已经实现了高并发，并能满足多线程条件下的内存一致性。下面是进阶功能。

3.7 大块内存的申请释放

申请内存的大小	申请方式
$x \leq 256KB(32 \text{ 页})$	向thread cache 申请
$32 \text{ 页} < x \leq 128 \text{ 页}$	向page cache申请
$x \geq 128 \text{ 页}$	向堆申请
当申请的内存大于256KB时，虽然不是从thread cache进行获取，但在分配内存时也是需要进行向上对齐的，对于大于256KB的内存我们可以直接按页进行对齐。	

大块内存的申请符合以下规则：1)更新 class AlignIndex 中关于内存块按页对齐，原来只有按特定字节的对齐。2)大于 256KB 小于 129 页的不在向 Thread Cache 申请，而是向 Page Cache 申请。3) 大于 128 页的，向堆申请。对 NewSpan 做修改。

大块内存的释放符合以下规则：1) 首先对 ConcurrentFree 修改：获取该大块内存地址对应的 span，对于大于 128 页的调用系统接口释放。

测试用例如下：

```

1 void BigAlloc()
2 {
3     //257kb
4     void* p1 = ConcurrentAlloc(257 * 1024);
5     ConcurrentFree(p1, 257 * 1024);
6
7     //129页
8     void* p2 = ConcurrentAlloc(129 * 8 * 1024);
9     ConcurrentFree(p2, 129 * 8 * 1024);
10 }

```

3.8 使用定长内存池配合脱离使用 new

代码中使用 new 时基本都是为 Span 结构的对象申请空间，而 span 对象基本都是在 page cache 层创建的，因此我们可以在 PageCache 类当中定义一个 _spanPool，用于 span 对象的申请和释放。项目搜索可修改多处。

```

class PageCache
{
private:
    //使用定长内存池替代new
    ObjectPool<Span> _spnObjectPool;
    //...
};

```

还有 ConcurrentAlloc 申请 Thread Cache 时，也会用到；这里使用一个静态的定长内存池对象.即使出了这个函数，它的值也会保持不变。不会再重新定义。

```

static void* ConcurrentAlloc(size_t size)
{
    if (pTLSThreadCache == nullptr)
    {
        static ObjectPool<ThreadCache> _threaCpool;
        pTLSThreadCache = _threaCpool.New();
    }
}

```

3.9 释放对象时优化为不传对象大小

需要在 Span 结构体里面添加一个记录对齐内存块大小的变量。Central Cache 从 Page Cache 获取一个 span 进行切分时，进行记录。

```

struct Span
{
    PAGE_ID _pageid = 0; //页号
    size_t _n = 0;      //几页

    Span* _prev; //双链表 方便不用的span释放给 page-cache
    Span* _next;

    void* _freelist = nullptr; //存储一个页切分成的一个个小的内存块
    size_t _useCount = 0;      //记录central cache 分给 thread cache 的内存数量

    bool isUse = false; //当前span是否正在被使用。
    size_t _objsize = 0;
};

```

ConcurrentFree 进行释放时，可以根据大于 256KB 的用 Page Cache 进行释放，Page Cache 释放需要一个 span*，而我们知道地址，根据映射关系求出 span 即可。如果是小于 256kb 的，则走三层释放模型，求出的 span 内部既有_objSize 的大小。最外层不需要传 size，三层释放模型内部还是需要的。

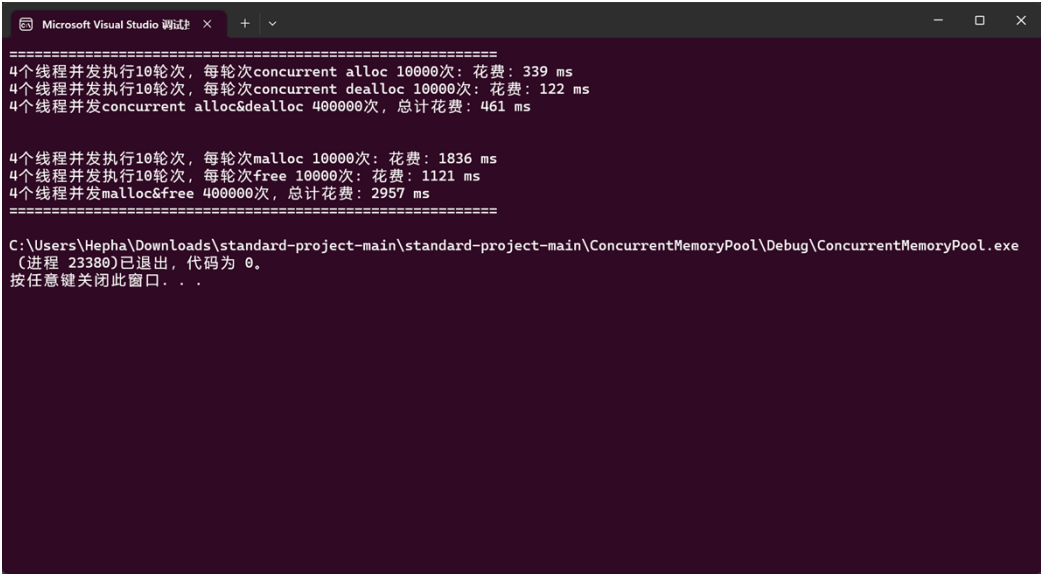
```
static void ConcurrentFree(void* ptr)
{
    assert(pTLSThreadCache);
    Span* spn1 = PageCache::GetPageInstance().MapObjectToSpan(ptr);

    if (spn1->_objsize > MAX_BYTES)
    {
        PageCache::GetPageInstance()._pagemtx.lock();
        PageCache::GetPageInstance().ReleaseSpanToPageCache(spn1);
        PageCache::GetPageInstance()._pagemtx.unlock();
    }
    else
    {
        pTLSThreadCache->Deallocate(ptr);
    }
}
```

而 unordered_map<PAGE_ID, Span*>_idMapSpan; 容器是不支持线程安全的，需要 i 在 PageCache::MapObjectToSpan 中添加一个 unique_lock 的锁。

```
Span* PageCache::MapObjectToSpan(void* obj)
{
    PAGE_ID id = ((PAGE_ID)obj >> PAGE_SHIFT);
    //当courrentFree时，有可能别的线程正在访问，造成线程不安全
    std::unique_lock<mutex> _lock(_pagemtx); //RAII锁
    auto ret = _idMapSpan.find(id);
    if (ret != _idMapSpan.end())
    {
        return ret->second;
    }
    else
    {
        assert(false);
        return nullptr;
    }
}
```

3.10 测试样例运行结果



```
Microsoft Visual Studio 调试 × + -
=====
4个线程并发执行10轮次, 每轮次concurrent alloc 10000次: 花费: 339 ms
4个线程并发执行10轮次, 每轮次concurrent dealloc 10000次: 花费: 122 ms
4个线程并发concurrent alloc&dealloc 400000次, 总计花费: 461 ms

4个线程并发执行10轮次, 每轮次malloc 10000次: 花费: 1836 ms
4个线程并发执行10轮次, 每轮次free 10000次: 花费: 1121 ms
4个线程并发malloc&free 400000次, 总计花费: 2957 ms
=====

C:\Users\Hepha\Downloads\standard-project-main\standard-project-main\ConcurrentMemoryPool\Debug\ConcurrentMemoryPool.exe
(进程 23380)已退出, 代码为 0。
按任意键关闭此窗口。 . . .
```

上面三行是使用了内存池的运行结果, 下面三行是没有用内存池的运行结果。使用内存池后的运行时间开销得到了明显的减少。

(完)