C++11 實現的100行線程池

程某有一計 C語言與C++編程 今天

> **來自：SegmentFault，作者：程某有一計**
> **鏈接：https://segmentfault.com/a/1190000022456590**

C++線程池一直都是各位程序員們造輪子的首選項目之一。今天，小編帶大家一起來看看這個輕量的線程池，本線程池是header-only的，並且整個文件只有100行，其中C++的高級用法有很多，很值得我們學習，一起來看看吧。

以下是正文

線程池

C++帶有線程操作，異步操作，就是沒有線程池，至於線程池的概念，我先搜一下別人的解釋：

一般而言,線程池有以下幾個部分:

1. 完成主要任務的一個或多個線程。

2. 用於調度管理的管理線程。

3. 要求執行的任務隊列。

我來講講人話：你的函數需要在多線程中運行，但是你又不能每來一個函數就開啟一個線程，所以你就需要固定的N個線程來跑執行，但是有的線程還沒有執行完，有的又在空閒，如何分配任務呢，你就需要封裝一個線程池來完成這些操作，有了線程池這層封裝，你就只需要告訴它開啟幾個線程，然後直接塞任務就行了，然後通過一定的機制獲取執行結果。

這裡有一個100行實現線程池的操作：
https://github.com/progschj/ThreadPool/blob/master/ThreadPool.h
分析源代碼頭文件

```
#include <vector>
#include <queue>
#include <memory>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <future>
#include <functional>
#include <stdexcept>
```

vector,queue,momory 都沒啥說的，thread線程相關，mutex 互斥量，解決資源搶占問題，condition_variable 條件量，用於喚醒線程和阻塞線程,future 從使用的角度出發，它是一個獲取線程數據的函數。functional 函數子，可以理解為規範化的函數指針。stdexcept 就跟它的名字一樣，標準異常。

```
class ThreadPool {
public:
    ThreadPool(size_t);
    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args)
```

```
        -> std::future<typename std::result_of<F(Args...)>::type>;
    ~ThreadPool();
private:
    // need to keep track of threads so we can join them
    std::vector< std::thread > workers;
    // the task queue
    std::queue< std::function<void()> > tasks;

    // synchronization
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
};
```

線程池的聲明，構造函數，一個enqueue模板函數返回std::future<type>, 然後這個type又利用了運行時檢測（還是編譯時檢測?）推斷出來的，非常的amazing啊。成功的使用一行代碼反复套娃，這高階的用法就是大佬的水平嗎，i了i了。

workers 是vector<std::thread> 俗稱工作線程。

std::queue<std::function<void()>> tasks 俗稱任務隊列。

那麼問題來了，這個任務隊列的任務只能是void() 類型的嗎？感覺沒那麼簡單，還得接著看吶。

mutex,condition_variable 沒啥講的,stop 控制線程池停止的。

```
// the constructor just launches some amount of workers
inline ThreadPool::ThreadPool(size_t threads)
```

```
    :   stop(false)
{
    for(size_t i = 0;i<threads;++i)
        workers.emplace_back(
            [this]
            {
                for(;;)
                {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        this->condition.wait(lock,
                            [this]{ return this->stop || !this->tasks.empty(); });
                        if(this->stop && this->tasks.empty())
                            return;
                        task = std::move(this->tasks.front());
                        this->tasks.pop();
                    }

                    task();
                }
            }
        );
}
```

大佬寫的註釋就是這麼樸實無華，說這個構造函數僅僅是把一定數量的線程塞進去，我是看了又看才悟出來這玩意是什麼意思……雖然本質上的確是它說的只是把線程塞進去，但是這個線程也太繞了。

workers.emplace_back 參數是一個lambda表達式，不會阻塞，也就是說最外層的是一個異步函數，每個線程裡面的事情才是重點。

labmda表達式中最外層是一個死循環，至於為什麼是for(;;)而不是while(1) 這雖然不是重點，不過大佬的用法還是值得揣摩的，我估計是效率會更高?

task 申明後，緊跟著一個大括號，這個{}裡面的部分，是一個同步操作，至於為什麼用this->lock 而不是直接使用[&]來捕獲參數，想來也是處於內存考慮。精打細算的風格像極了摳門的地主，i了i了。

緊接著一個wait(lock,condtion)的操作，像極了千層餅的套路。

第一層：這TM不是要鎖死自己啊？這樣不是構造都得卡死?

第二層：我們看到它emplace_back了一個線程，不會阻塞，但是等開鎖，鎖不就在它自己的線程裡面嘛？那不得鎖死了啊?

第三層：我們看到這個lock其實只是個包裝，真正的鎖是外層的mutex，所以從這裡是不存在死鎖的。但是你的wait的condition怎麼可能不懂呢，必須要stop 或者!empty 才wait嗎?

第四層：我們查資料發現後面的condition是返回false才會wait，也就是說要!stop && empty才會wait，就是說這個線程池是運行態，並且沒有任務才才會執行等待操作！否則就不等了，直接衝！

第五層：既然你判斷了上面判斷了stop和非空，為啥下面還要判斷stop和空才退出呢？不顯得冗餘?

第六層：要確定它的確是被置為stop了，且隊列執行空了，它才能夠光榮退休。有沒有問題呢，有，最後所有線程都阻塞了，你stop置為true它們

也不知道啊……

我估計它的stop會有喚醒所有線程的操作，不過如果有的在執行，有的在等待，應該沒辦法都通知到位，但是在執行的在下一次判斷的時候也能正常退出。

因為有了疑惑，我們就想看stop相關的操作，結果發現放在了析構函數里面……

```
// the destructor joins all threads
inline ThreadPool::~ThreadPool()
{
    {
        std::unique_lock<std::mutex> lock(queue_mutex);
        stop = true;
    }
    condition.notify_all();
    for(std::thread &worker: workers)
        worker.join();
}
```

{}裡面上鎖進行了stop為true的操作，至於為什麼不用原子操作，我也不知道，但是仔細想了下大概是因為本來就有一把鎖了，再用原子就不是內味兒了。然後它果然通知了所有，並且還把工作線程join了。也就是等它們結束。
結束了千層餅の解析之後，我們看看最重要的入隊操作

```
// add new work item to the pool
template<class F, class... Args>
auto ThreadPool::enqueue(F&& f, Args&&... args)
    -> std::future<typename std::result_of<F(Args...)>::type>
```

```
{
    using return_type = typename std::result_of<F(Args...)>::type;

    auto task = std::make_shared< std::packaged_task<return_type()> >(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );

    std::future<return_type> res = task->get_future();
    {
        std::unique_lock<std::mutex> lock(queue_mutex);

        // don't allow enqueueing after stopping the pool
        if(stop)
            throw std::runtime_error("enqueue on stopped ThreadPool");

        tasks.emplace([task](){ (*task)(); });
    }
    condition.notify_one();
    return res;
}
```

typename std::result_of<F(Args...)>::type中的typename 應該是為消除歧義的，或者因為嵌套依賴名字的關係，做為一個堅決不寫模板的普通程序員，這段代碼太難了……-> type 我倒是知道怎麼回事，就是指明它的返回類型的一種方式result_of<F(Args...)> 應該是指明了F是一個函數，簽名為Args...這個變參，Args是啥它不關係，它關心的是返回值的參數類型所以有個type。

至於為什麼函數入口是一個右值引用那就超出我的理解範圍了。難道說functional 必須要右值引用？那它的銷毀誰來管呢？這個線程來管嗎？這些坑我以後慢慢填。

前面我們說了tasks 只能接收void() 的函數類型，這裡使用std::packaged_task<return_type()>完成對函數類型的推導，至於為什麼不用function<return_type()> ，因為這還不是最終放入tasks的對象，它要承接一個返回future<T>的工作，而package_task就是來打包返回future<T>的……

然後就是加鎖入隊+通知工作線程+返回future<T>的操作。本來是線程池最難理解的部分，反而顯得平淡無奇了，因為前面那些花里胡哨的操作已經很好的打通了我們的理解能力。對於這個操作本來就有一點概念的，就顯得有種"就這？"的感覺……

--- EOF ---

優雅的使用ThreadLocal

Java後端

親測有效！Spring Boot 項目優化和JVM 調優

Java後端