# **基于C++11实现线程池**

## **0x00 导入**

2022/1/28更新：本文内容着重于实现线程池所涉及到的一些C++11的语法，若对线程池的实现的本身更感兴趣，可以去看一下我写的另一篇文章：

[当我谈线程池时我谈些什么——线程池学习笔记 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/444375447)

前些日子通过阅读几篇博客，大体学习了一下C++的并发编程（主要是多线程、异步部分）。手撸了一个多生产者-多消费者模型之后，最终觉得这也只是小打小闹而已，需要整个硬货。经过一番思考之后，手撸一个线程池貌似是一个非常合适的练习。线程池前前后后一共折腾了四五个小时左右，最令我意外的是——原本以为最难理解的并发部分反而是最简单的部分，线程池实现中大量的CPP11语法糖才是真正影响我理解的部分。经过三四个小时的阅读博文、查阅资料以及向大佬求教之后，我才真正对线程池实现有了比较深入地理解。“纸上得来终觉浅，绝知此事要躬行。”经历这一番折腾后才清楚地认识到了自己对CPP11理解的薄弱，遂写下这篇博客来总结自己对线程池实现以及所涉及到的CPP11语法糖的理解，方便自己以后再次阅读源码时有所考据。

文章大量借鉴、节选了众多参考资料，并结合自己的理解进行讲解。参考资料会在最后列出。

这篇总结会将重心放在C++11的语法糖上，对于C++11的并发编程部分（std::thread, std::future等）将仅进行最简洁最必要的阐述。有关并发编程部分可以移步至几篇大佬总结的比较好的博文中进行补充学习：

1. [C++ 并发编程（从C++11到C++17） (paul.pub)](https://paul.pub/cpp-concurrency/)
2. [从 pthread 转换到 std::thread - SegmentFault 思否](https://segmentfault.com/a/1190000002655852)
3. [货比三家：C++ 中的 task based 并发 - SegmentFault 思否](https://segmentfault.com/a/1190000002706259)

## **0x01 逐步实现线程池**

### **线程池原理**

**C++11**加入了线程库，从此告别了标准库不支持并发的历史。然而C++对于多线程的支持还是比较低级，稍微高级一点的用法都需要自己去实现，比如线程池、信号量等。线程池（thread pool）这个东西，一般在面试时的回答都是：“管理一个任务队列，一个线程队列，然后每次去一个任务分配给一个线程去做，循环往复。”这回答貌似没有问题，但是写起程序来的时候就出问题了。

有什么问题？线程池一般是要复用线程，所以如果是取一个task分配给某一个thread，执行完之后再重新分配，在语言层面这是基本不能实现的：C++的thread都是执行一个固定的task函数，执行完之后线程也就结束了。所以该如何实现task和thread的分配呢？

**让每一个thread创建后，就去执行调度函数：循环获取task，然后执行。**

这个循环该什么时候停止呢？

很简单，**当线程池停止使用时**，循环停止。

这样一来，就保证了thread函数的唯一性，而且复用线程执行task。

总结一下，我们的线程池的主要组成部分有二：

* 任务队列（Task Queue）
* 线程池（Thread Pool）

线程池与任务队列之间的匹配操作，是典型的*生产者-消费者*模型，本模型使用了两个工具：一个mutex + 一个条件变量。mutex就是锁，保证任务的添加和移除（获取）的互斥性；一个条件变量保证多个线程获取task的同步性：当任务队列为空时，线程应该等待（阻塞）。

接下来我们就可以逐渐将一块块积木拼成一个完整的简易线程池。

### **积木1：任务队列（Task Queue）**

我们会理所当然地希望任务以发送它相同的顺序来逐个执行，因此队列是最适合的数据结构。

这里我们把任务队列单拿出来，独自为类，方便以后进行各种骚操作。

将任务队列单拿出来之后，我们应考虑一个问题：正如上一节提到的线程池task与thread的分配方法所示，线程池中的线程会持续查询任务队列是否有可用工作。当两个甚至多个线程试图同时执行查询工作时，这会引起难以估计的灾难。因而我们需要对C++的std::queue进行包装，实现一个**线程安全**的*SafeQueue*。

实现一个线程安全的*SafeQueue*原理很简单，利用**mutex**来限制并发访问即可。我们可以在*SafeQueue*类中定义一个std::mutex类型的成员变量，并在相应的操作接口（如入队接口enqueue()）中利用互斥体包装器来管理这个mutex，确保没有其他人正在访问该资源。

下面给出完整的*SafeQueue*代码：

**template** **<typename** T**>class** **SafeQueue**{**private:**

std**::**queue**<**T**>** m\_queue; *//利用模板函数构造队列*​

std**::**mutex m\_mutex; *// 访问互斥信号量*​**public:**

SafeQueue() {}

SafeQueue(SafeQueue **&&**other) {}

**~**SafeQueue() {}​

**bool** empty() *// 返回队列是否为空* {

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.empty();

}​

**int** size()

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.size();

}​

*// 队列添加元素* **void** enqueue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex);

m\_queue.emplace(t);

}​

*// 队列取出元素* **bool** dequeue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 队列加锁*​

**if** (m\_queue.empty())

**return** false;

t **=** std**::**move(m\_queue.front()); *// 取出队首元素，返回队首元素值，并进行右值引用*​

m\_queue.pop(); *// 弹出入队的第一个元素*​

**return** true;

}};

### **积木2：线程池（Thread Pool）**

线程池是线程池模型的主体，我们将它拆成更小的部分来逐步分析，方便理解。

### **2-1 提交函数**

线程池最重要的方法就是负责向任务队列添加任务。我们的提交函数应该做到以下两点：

* 接收任何参数的任何函数。（普通函数，Lambda，成员函数……）
* 立即返回“东西”，避免阻塞主线程。这里返回的“东西”或者说“对象”应该包含任务结束的结果。

完整的提交函数如下所示：

*// Submit a function to be executed asynchronously by the pool* **template** **<typename** F, **typename**... Args**>**

**auto** submit(F **&&**f, Args **&&**...args) **->** std**::**future**<decltype**(f(args...))**>** ①

{

*// Create a function with bounded parameter ready to execute* std**::**function**<decltype**(f(args...))()**>** func **=** std**::**bind(std**::**forward**<**F**>**(f), std**::**forward**<**Args**>**(args)...); ②*// 连接函数和参数定义，特殊函数类型，避免左右值错误*​

*// Encapsulate it into a shared pointer in order to be able to copy construct* **auto** task\_ptr **=** std**::**make\_shared**<**std**::**packaged\_task**<decltype**(f(args...))()**>>**(func); ③​

*// Warp packaged task into void function* std**::**function**<void**()**>** warpper\_func **=** [task\_ptr]()

{

(**\***task\_ptr)();

}; ④​

*// 队列通用安全封包函数，并压入安全队列* m\_queue.enqueue(warpper\_func);​

*// 唤醒一个等待中的线程* m\_conditional\_lock.notify\_one(); ⑤​

*// 返回先前注册的任务指针* **return** task\_ptr**->**get\_future();

}

C++11众多的语法糖正式来袭。下面讲一下需要注意的地方：

1.submit()是一个模板函数，这很明显。template<typename F, typename... Args>中的typename... Args是C++11引入的**可变模版参数**（variadic templates），很容易理解。  
首先来看长得奇奇怪怪的函数头部分，auto submit(F &&f, Args &&...args) -> std::future<decltype(f(args...))>，这里函数类型的定义用到了叫做**“尾返回类型推导”**的技巧。  
按照标准，auto关键字不能用于函数形参的类型推导，在C++14以前，也不能直接用auto func()的形式来推导函数的返回类型。  
因此传统C++中我们必须这么写：

**C++11**加入了线程库，从此告别了标准库不支持并发的历史。然而C++对于多线程的支持还是比较低级，稍微高级一点的用法都需要自己去实现，比如线程池、信号量等。线程池（thread pool）这个东西，一般在面试时的回答都是：“管理一个任务队列，一个线程队列，然后每次去一个任务分配给一个线程去做，循环往复。”这回答貌似没有问题，但是写起程序来的时候就出问题了。

有什么问题？线程池一般是要复用线程，所以如果是取一个task分配给某一个thread，执行完之后再重新分配，在语言层面这是基本不能实现的：C++的thread都是执行一个固定的task函数，执行完之后线程也就结束了。所以该如何实现task和thread的分配呢？

**让每一个thread创建后，就去执行调度函数：循环获取task，然后执行。**

这个循环该什么时候停止呢？

很简单，**当线程池停止使用时**，循环停止。

这样一来，就保证了thread函数的唯一性，而且复用线程执行task。

总结一下，我们的线程池的主要组成部分有二：

* 任务队列（Task Queue）
* 线程池（Thread Pool）

线程池与任务队列之间的匹配操作，是典型的*生产者-消费者*模型，本模型使用了两个工具：一个mutex + 一个条件变量。mutex就是锁，保证任务的添加和移除（获取）的互斥性；一个条件变量保证多个线程获取task的同步性：当任务队列为空时，线程应该等待（阻塞）。

接下来我们就可以逐渐将一块块积木拼成一个完整的简易线程池。

### **积木1：任务队列（Task Queue）**

我们会理所当然地希望任务以发送它相同的顺序来逐个执行，因此队列是最适合的数据结构。

这里我们把任务队列单拿出来，独自为类，方便以后进行各种骚操作。

将任务队列单拿出来之后，我们应考虑一个问题：正如上一节提到的线程池task与thread的分配方法所示，线程池中的线程会持续查询任务队列是否有可用工作。当两个甚至多个线程试图同时执行查询工作时，这会引起难以估计的灾难。因而我们需要对C++的std::queue进行包装，实现一个**线程安全**的*SafeQueue*。

实现一个线程安全的*SafeQueue*原理很简单，利用**mutex**来限制并发访问即可。我们可以在*SafeQueue*类中定义一个std::mutex类型的成员变量，并在相应的操作接口（如入队接口enqueue()）中利用互斥体包装器来管理这个mutex，确保没有其他人正在访问该资源。

下面给出完整的*SafeQueue*代码：

**template** **<typename** T**>class** **SafeQueue**{**private:**

std**::**queue**<**T**>** m\_queue; *//利用模板函数构造队列*​

std**::**mutex m\_mutex; *// 访问互斥信号量*​**public:**

SafeQueue() {}

SafeQueue(SafeQueue **&&**other) {}

**~**SafeQueue() {}​

**bool** empty() *// 返回队列是否为空* {

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.empty();

}​

**int** size()

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.size();

}​

*// 队列添加元素*

**void** enqueue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex);

m\_queue.emplace(t);

}​

*// 队列取出元素*

**bool** dequeue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 队列加锁*​

**if** (m\_queue.empty())

**return** false;

t **=** std**::**move(m\_queue.front()); *// 取出队首元素，返回队首元素值，并进行右值引用*​

m\_queue.pop(); *// 弹出入队的第一个元素*​

**return** true;

}};

### **积木2：线程池（Thread Pool）**

线程池是线程池模型的主体，我们将它拆成更小的部分来逐步分析，方便理解。

### **2-1 提交函数**

线程池最重要的方法就是负责向任务队列添加任务。我们的提交函数应该做到以下两点：

* 接收任何参数的任何函数。（普通函数，Lambda，成员函数……）
* 立即返回“东西”，避免阻塞主线程。这里返回的“东西”或者说“对象”应该包含任务结束的结果。

完整的提交函数如下所示：

*// Submit a function to be executed asynchronously by the pool*

**template** **<typename** F, **typename**... Args**>**

**auto** submit(F **&&**f, Args **&&**...args) **->** std**::**future**<decltype**(f(args...))**>** ①

{

*// Create a function with bounded parameter ready to execute* std**::**function**<decltype**(f(args...))()**>** func **=** std**::**bind(std**::**forward**<**F**>**(f), std**::**forward**<**Args**>**(args)...); ②*// 连接函数和参数定义，特殊函数类型，避免左右值错误*​

*// Encapsulate it into a shared pointer in order to be able to copy construct* **auto** task\_ptr **=** std**::**make\_shared**<**std**::**packaged\_task**<decltype**(f(args...))()**>>**(func); ③​

*// Warp packaged task into void function*

std**::**function**<void**()**>** warpper\_func **=** [task\_ptr]()

{

(**\***task\_ptr)();

}; ④​

*// 队列通用安全封包函数，并压入安全队列*

m\_queue.enqueue(warpper\_func);​

*// 唤醒一个等待中的线程*

m\_conditional\_lock.notify\_one(); ⑤​

*// 返回先前注册的任务指针*

**return** task\_ptr**->**get\_future();

}

C++11众多的语法糖正式来袭。下面讲一下需要注意的地方：

1.submit()是一个模板函数，这很明显。template<typename F, typename... Args>中的typename... Args是C++11引入的**可变模版参数**（variadic templates），很容易理解。  
首先来看长得奇奇怪怪的函数头部分，auto submit(F &&f, Args &&...args) -> std::future<decltype(f(args...))>，这里函数类型的定义用到了叫做**“尾返回类型推导”**的技巧。  
按照标准，auto关键字不能用于函数形参的类型推导，在C++14以前，也不能直接用auto func()的形式来推导函数的返回类型。  
因此传统C++中我们必须这么写：

**template<typename** R, **typename** T, **typename** U**>**

R add(T x, U y) {

**return** x**+**y;}

这样存在很明显的缺陷：事实上很多时候我们并不知道add()这个函数会进行什么操作，获取什么样的返回类型。  
最终在C++11中这个问题得到了解决。C++11关键字decltype解决了auto关键字只能对变量类型进行类型推导的缺陷。它的用法也很简单，应该也是看过C++11标准就能记住的：  
decltype(表达式)  
但是为了利用decltype来推导函数的返回类型，我们并不能直接写出这种形式的代码：  
decltype(x+y) add(T x, U y)  
因为编译器在读到decltype(x+y)时，x和y尚未定义。而这个问题的解决方案，正是**尾返回类型推导**。C++11引入了一个**尾返回类型**（trailing return type），利用auto关键字将返回类型后置：  
template<typename T, typename U>  
auto add2(T x, U y) -> decltype(x+y){  
return x + y;  
}  
至此，看起来奇奇怪怪的函数头中关于函数的返回类型的定义已经清楚明了：该函数的返回值将从std::future<decltype(f(args...))>中自动推导得出。  
接着谈函数头。这里我们谈一下std::future，它提供了一个访问异步操作结果的途径。我们可以使用std::future的wait()方法来设置屏障，阻塞线程，实现线程同步。并最终使用std::future的get()方法来获得执行结果。

对于std::future，可以在这篇文献中找到更详细的讲解：

[第 7 章 并行与并发 现代 C++ 教程: 高速上手 C++ 11/14/17/20 - Modern C++ Tutorial: C++ 11/14/17/20 On the Fly (changkun.de)](https://changkun.de/modern-cpp/zh-cn/07-thread/" \l "7-3-%E6%9C%9F%E7%89%A9)

总的来说，在我们的程序中，最后将会获得返回类型为 实例化为f(args...)的std::future<> 的submit函数。  
如果我们阅读其他一些博文或者Github上著名的99行C++11实现线程池，我们可能会看到以下形式的添加任务方法的定义：

auto ThreadPool::enqueue(F&& f, Args&&... args) -> std::future<typename std::result\_of<F(Args...)>::type>

为什么我们这里不采用std::result\_of<>::type的方法而是使用decltype(f(args...))方法，这个要结合下一点来理解。

2. 这里我们使用了std::function进行包装从而产生了一个特殊函数，这个特殊函数使用std::bind将函数f和参数args绑定起来。  
简单来说，std::function可以对多个相似的函数进行包装（即通用的描述方法）。std::function可以hold住任何可以通过“()”来调用的对象，包括：

普通函数

成员函数

lambda

std::bind

而std::bind可以**将调用函数时的部分参数先制定好，留下一部分在真正调用时确定**。（当然，你也可以直接指定全部参数，在调用时不再指定。）  
对于std::function和std::bind，我们可以移步这篇博客获得更详细的讲解：

[C++11中的lambda，std::function以及std:bind (paul.pub)](https://paul.pub/cpp-lambda-function-bind/)

这里我们会注意到，std::bind中，出现了一个std::forward()的特殊方法。std::forward()又被称作**完美转发**。简单来说，std::forward()将会完整保留参数的引用类型进行转发。如果参数是左值引用（*lvalue*），该方法会将参数保留左值引用的形式进行转发，如果参数是右值引用（*rvalue*），该方法会将参数保留右值引用的形式进行转发。而我们这里为什么要使用这个方法呢？  
我们会对为什么使用std::forward()方法产生疑惑，可能是因为我们看到了函数头中的F&& f和Args&&... args，这难道不已经指明这个函数接收的参数类型应为右值引用吗？**其实不然。**这里的F&& f和Args&&... args中的&&并非是右值引用意思，而是一种特殊现象，这个现象被称作**万能引用**（universal reference）。  
**万能引用**可以简单理解为，当T是模板参数时，T&&的作用主要是保持值类别进行转发。然而，**一个绑定到universial reference上的对象可能具有lvaluesness或者rvalueness，正是因为有这种二义性**，所以产生了std::forward。  
有关于万能引用、完美转发以及背后所隐藏的引用折叠，可以在这篇知乎回答中找到更详细的介绍：

[现代C++之万能引用、完美转发、引用折叠 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/99524127)

总的来说，②会产生一个以 函数f(arg...)返回类型 为返回类型、不含参数的特殊函数包装func。  
这里我们也不难注意到，在网上其他的示例中，这里使用了如下方法：  
using return\_type = typename std::result\_of<F(Args...)>::type;  
可以看到，这里再次像这些示例在函数头中一样，使用了std::result\_of方法，而结合上文，我们也不难理解为什么本文会使用std::function方法，即更方便地增加对将成员函数和lambda表达式作为参数的支持。

3. 这里我们使用std::make\_shared<>()方法，声明了一个std::packaged\_task<decltype(f(args...))()>类型的智能指针，并将前面std::function方法声明的特殊函数包装func传入作为std::packaged\_task的实例化参数。智能指针将更方便我们对该std::packaged\_task对象进行管理。  
std::packaged\_task可以用来封装任何可以调用的目标，从而用于实现异步的调用。

4. 这里我们再次利用std::function，将*task\_ptr*指向的**std::packaged\_task**对象取出并包装为void函数。这样我们的代码将更加美观优雅。  
当然，我们也可以像其他示例一样，将这一步和下一步任务进入队列操作简化为一步：

m\_queue.enqueue([task\_ptr](){

(**\***task\_ptr)();});

5. 这里条件变量会通知一个处于*wait*状态的线程，该线程将会从任务队列中取得任务并执行。  
这里简要介绍一下条件变量（std::condition\_variable）：  
条件变量 std::condition\_variable 是为了解决死锁而生，当互斥操作不够用而引入的。比如，线程可能需要等待某个条件为真才能继续执行，而一个忙等待循环中可能会导致所有其他线程都无法进入临界区使得条件为真时，就会发生死锁。所以，condition\_variable实例被创建出现主要就是用于唤醒等待线程从而避免死锁。std::condition\_variable的notify\_one()用于唤醒一个线程；notify\_all()则是通知所有线程。

提交函数到此结束。

### **2-2 内置工作线程类**

本文在线程池中设立私有成员类*ThreadWoker*作为内置线程工作类，执行真正的工作。

**class** **ThreadWorker** *// 内置线程工作类*{**private:**

**int** m\_id; *// 工作id*​

ThreadPool **\***m\_pool; *// 所属线程池***public:**

*// 构造函数* ThreadWorker(ThreadPool **\***pool, **const** **int** id) **:** m\_pool(pool), m\_id(id)

{

}​

*// 重载()操作* **void** **operator**()()

{

std**::**function**<void**()**>** func; *// 定义基础函数类func*​

**bool** dequeued; *// 是否正在取出队列中元素*

*// 判断线程池是否关闭，没有关闭则从任务队列中循环提取任务* **while** (**!**m\_pool**->**m\_shutdown)

{

{

*// 为线程环境加锁，互访问工作线程的休眠和唤醒* std**::**unique\_lock**<**std**::**mutex**>** lock(m\_pool**->**m\_conditional\_mutex);​

*// 如果任务队列为空，阻塞当前线程* **if** (m\_pool**->**m\_queue.empty())

{

m\_pool**->**m\_conditional\_lock.wait(lock); *// 等待条件变量通知，开启线程* }​

*// 取出任务队列中的元素* dequeued **=** m\_pool**->**m\_queue.dequeue(func);

}​

*// 如果成功取出，执行工作函数* **if** (dequeued)

func();

}

}};

这里应该重点关注重载()操作的void operator()()，这里面进行了任务的取出与执行。

参照注释和上文，我们使用了一个while循环，在线程池处于工作时循环从任务队列中提取任务。并利用条件变量，在任务队列为空时阻塞当前线程，等待上文中的提交函数添加任务后发出的通知。在任务队列不为空时，我们将任务队列中的任务取出，并放在事先声明的基础函数类*func*中。成功取出后便立即执行该任务。

### **线程池完整代码**

下面给出线程池的完整代码：

**class** **ThreadPool**{**private:**

**class** **ThreadWorker** *// 内置线程工作类* {

**private:**

**int** m\_id; *// 工作id*​

ThreadPool **\***m\_pool; *// 所属线程池* **public:**

*// 构造函数* ThreadWorker(ThreadPool **\***pool, **const** **int** id) **:** m\_pool(pool), m\_id(id)

{

}​

*// 重载()操作* **void** **operator**()()

{

std**::**function**<void**()**>** func; *// 定义基础函数类func*​

**bool** dequeued; *// 是否正在取出队列中元素*​

**while** (**!**m\_pool**->**m\_shutdown)

{

{

*// 为线程环境加锁，互访问工作线程的休眠和唤醒* std**::**unique\_lock**<**std**::**mutex**>** lock(m\_pool**->**m\_conditional\_mutex);​

*// 如果任务队列为空，阻塞当前线程* **if** (m\_pool**->**m\_queue.empty())

{

m\_pool**->**m\_conditional\_lock.wait(lock); *// 等待条件变量通知，开启线程* }​

*// 取出任务队列中的元素* dequeued **=** m\_pool**->**m\_queue.dequeue(func);

}​

*// 如果成功取出，执行工作函数* **if** (dequeued)

func();

}

}

};​

**bool** m\_shutdown; *// 线程池是否关闭*​

SafeQueue**<**std**::**function**<void**()**>>** m\_queue; *// 执行函数安全队列，即任务队列*​

std**::**vector**<**std**::thread>** m\_threads; *// 工作线程队列*​

std**::**mutex m\_conditional\_mutex; *// 线程休眠锁互斥变量*​

std**::**condition\_variable m\_conditional\_lock; *// 线程环境锁，可以让线程处于休眠或者唤醒状态*​**public:**

*// 线程池构造函数* ThreadPool(**const** **int** n\_threads **=** 4)

**:** m\_threads(std**::**vector**<**std**::thread>**(n\_threads)), m\_shutdown(false)

{

}​

ThreadPool(**const** ThreadPool **&**) **=** **delete**;​

ThreadPool(ThreadPool **&&**) **=** **delete**;​

ThreadPool **&operator=**(**const** ThreadPool **&**) **=** **delete**;​

ThreadPool **&operator=**(ThreadPool **&&**) **=** **delete**;​

*// Inits thread pool* **void** init()

{

**for** (**int** i **=** 0; i **<** m\_threads.size(); **++**i)

{

m\_threads.at(i) **=** std**::thread**(ThreadWorker(**this**, i)); *// 分配工作线程* }

}​

*// Waits until threads finish their current task and shutdowns the pool* **void** shutdown()

{

m\_shutdown **=** true;

m\_conditional\_lock.notify\_all(); *// 通知，唤醒所有工作线程*​

**for** (**int** i **=** 0; i **<** m\_threads.size(); **++**i)

{

**if** (m\_threads.at(i).joinable()) *// 判断线程是否在等待* {

m\_threads.at(i).join(); *// 将线程加入到等待队列* }

}

}​

*// Submit a function to be executed asynchronously by the pool* **template** **<typename** F, **typename**... Args**>**

**auto** submit(F **&&**f, Args **&&**...args) **->** std**::**future**<decltype**(f(args...))**>**

{

*// Create a function with bounded parameter ready to execute* std**::**function**<decltype**(f(args...))()**>** func **=** std**::**bind(std**::**forward**<**F**>**(f), std**::**forward**<**Args**>**(args)...); *// 连接函数和参数定义，特殊函数类型，避免左右值错误*​

*// Encapsulate it into a shared pointer in order to be able to copy construct* **auto** task\_ptr **=** std**::**make\_shared**<**std**::**packaged\_task**<decltype**(f(args...))()**>>**(func);​

*// Warp packaged task into void function* std**::**function**<void**()**>** warpper\_func **=** [task\_ptr]()

{

(**\***task\_ptr)();

};​

*// 队列通用安全封包函数，并压入安全队列* m\_queue.enqueue(warpper\_func);​

*// 唤醒一个等待中的线程* m\_conditional\_lock.notify\_one();​

*// 返回先前注册的任务指针* **return** task\_ptr**->**get\_future();

}};

结合注释应该能很轻松的理解线程池剩余的代码。

注意一下init()函数和shutdown()函数：

* 在线程池初始化函数init()中，我们声明并分配工作线程，将工作线程放入工作线程队列*m\_threads*中。
* 在线程池关闭函数shutdown()中，我们唤醒所有工作线程，并等待期完成所有工作后关闭线程池。

这里我们也可以改进一下代码，将shutdown()函数中的工作转移到*ThreadPool*的析构函数中，从而更便利日后的使用。

至此，线程池全文讲解结束，后附完整项目代码及参考资料。

## **0x02 项目完整代码**

### **线程池代码**

*//thread\_pool.h*​**#ifndef THREAD\_POOL\_H#define THREAD\_POOL\_H**​**#include** <mutex>**#include** <queue>**#include** <functional>**#include** <future>**#include** <thread>**#include** <utility>**#include** <vector>​*// Thread safe implementation of a Queue using a std::queue***template** **<typename** T**>class** **SafeQueue**{**private:**

std**::**queue**<**T**>** m\_queue; *//利用模板函数构造队列*​

std**::**mutex m\_mutex; *// 访问互斥信号量*​**public:**

SafeQueue() {}

SafeQueue(SafeQueue **&&**other) {}

**~**SafeQueue() {}​

**bool** empty() *// 返回队列是否为空* {

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.empty();

}​

**int** size()

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 互斥信号变量加锁，防止m\_queue被改变*​

**return** m\_queue.size();

}​

*// 队列添加元素* **void** enqueue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex);

m\_queue.emplace(t);

}​

*// 队列取出元素* **bool** dequeue(T **&**t)

{

std**::**unique\_lock**<**std**::**mutex**>** lock(m\_mutex); *// 队列加锁*​

**if** (m\_queue.empty())

**return** false;

t **=** std**::**move(m\_queue.front()); *// 取出队首元素，返回队首元素值，并进行右值引用*​

m\_queue.pop(); *// 弹出入队的第一个元素*​

**return** true;

}};​**class** **ThreadPool**{**private:**

**class** **ThreadWorker** *// 内置线程工作类* {

**private:**

**int** m\_id; *// 工作id*​

ThreadPool **\***m\_pool; *// 所属线程池* **public:**

*// 构造函数* ThreadWorker(ThreadPool **\***pool, **const** **int** id) **:** m\_pool(pool), m\_id(id)

{

}​

*// 重载()操作* **void** **operator**()()

{

std**::**function**<void**()**>** func; *// 定义基础函数类func*​

**bool** dequeued; *// 是否正在取出队列中元素*​

**while** (**!**m\_pool**->**m\_shutdown)

{

{

*// 为线程环境加锁，互访问工作线程的休眠和唤醒* std**::**unique\_lock**<**std**::**mutex**>** lock(m\_pool**->**m\_conditional\_mutex);​

*// 如果任务队列为空，阻塞当前线程* **if** (m\_pool**->**m\_queue.empty())

{

m\_pool**->**m\_conditional\_lock.wait(lock); *// 等待条件变量通知，开启线程* }​

*// 取出任务队列中的元素* dequeued **=** m\_pool**->**m\_queue.dequeue(func);

}​

*// 如果成功取出，执行工作函数* **if** (dequeued)

func();

}

}

};​

**bool** m\_shutdown; *// 线程池是否关闭*​

SafeQueue**<**std**::**function**<void**()**>>** m\_queue; *// 执行函数安全队列，即任务队列*​

std**::**vector**<**std**::thread>** m\_threads; *// 工作线程队列*​

std**::**mutex m\_conditional\_mutex; *// 线程休眠锁互斥变量*​

std**::**condition\_variable m\_conditional\_lock; *// 线程环境锁，可以让线程处于休眠或者唤醒状态*​**public:**

*// 线程池构造函数* ThreadPool(**const** **int** n\_threads **=** 4)

**:** m\_threads(std**::**vector**<**std**::thread>**(n\_threads)), m\_shutdown(false)

{

}​

ThreadPool(**const** ThreadPool **&**) **=** **delete**;​

ThreadPool(ThreadPool **&&**) **=** **delete**;​

ThreadPool **&operator=**(**const** ThreadPool **&**) **=** **delete**;​

ThreadPool **&operator=**(ThreadPool **&&**) **=** **delete**;​

*// Inits thread pool* **void** init()

{

**for** (**int** i **=** 0; i **<** m\_threads.size(); **++**i)

{

m\_threads.at(i) **=** std**::thread**(ThreadWorker(**this**, i)); *// 分配工作线程* }

}​

*// Waits until threads finish their current task and shutdowns the pool* **void** shutdown()

{

m\_shutdown **=** true;

m\_conditional\_lock.notify\_all(); *// 通知，唤醒所有工作线程*​

**for** (**int** i **=** 0; i **<** m\_threads.size(); **++**i)

{

**if** (m\_threads.at(i).joinable()) *// 判断线程是否在等待* {

m\_threads.at(i).join(); *// 将线程加入到等待队列* }

}

}​

*// Submit a function to be executed asynchronously by the pool* **template** **<typename** F, **typename**... Args**>**

**auto** submit(F **&&**f, Args **&&**...args) **->** std**::**future**<decltype**(f(args...))**>**

{

*// Create a function with bounded parameter ready to execute* std**::**function**<decltype**(f(args...))()**>** func **=** std**::**bind(std**::**forward**<**F**>**(f), std**::**forward**<**Args**>**(args)...); *// 连接函数和参数定义，特殊函数类型，避免左右值错误*​

*// Encapsulate it into a shared pointer in order to be able to copy construct* **auto** task\_ptr **=** std**::**make\_shared**<**std**::**packaged\_task**<decltype**(f(args...))()**>>**(func);​

*// Warp packaged task into void function* std**::**function**<void**()**>** warpper\_func **=** [task\_ptr]()

{

(**\***task\_ptr)();

};​

*// 队列通用安全封包函数，并压入安全队列* m\_queue.enqueue(warpper\_func);​

*// 唤醒一个等待中的线程* m\_conditional\_lock.notify\_one();​

*// 返回先前注册的任务指针* **return** task\_ptr**->**get\_future();

}};​**#endif**

### **测试样例代码**

*// test.cpp*

​**#include** <iostream>

**#include** <random>

**#include** "thread\_pool.h"

std**::**random\_device rd; *// 真实随机数产生器*​

std**::**mt19937 mt(rd()); *//生成计算随机数mt*​

std**::**uniform\_int\_distribution**<int>** dist(**-**1000, 1000); *//生成-1000到1000之间的离散均匀分布数*​**auto** rnd **=** std**::**bind(dist, mt);​*// 设置线程睡眠时间***void** simulate\_hard\_computation(){

std**::**this\_thread**::**sleep\_for(std**::**chrono**::**milliseconds(2000 **+** rnd()));}​

*// 添加两个数字的简单函数并打印结果*

**void** multiply(**const** **int** a, **const** **int** b){

simulate\_hard\_computation();

**const** **int** res **=** a **\*** b;

std**::**cout **<<** a **<<** " \* " **<<** b **<<** " = " **<<** res **<<** std**::**endl;}​

*// 添加并输出结果*

**void** multiply\_output(**int** **&**out, **const** **int** a, **const** **int** b){

simulate\_hard\_computation();

out **=** a **\*** b;

std**::**cout **<<** a **<<** " \* " **<<** b **<<** " = " **<<** out **<<** std**::**endl;}

​*// 结果返回*

**int** multiply\_return(**const** **int** a, **const** **int** b){

simulate\_hard\_computation();

**const** **int** res **=** a **\*** b;

std**::**cout **<<** a **<<** " \* " **<<** b **<<** " = " **<<** res **<<** std**::**endl;

**return** res;}​**void** example(){

*// 创建3个线程的线程池*

ThreadPool **pool**(3);​

*// 初始化线程池*

pool.init();​

*// 提交乘法操作，总共30个*

**for** (**int** i **=** 1; i **<=** 3; **++**i)

**for** (**int** j **=** 1; j **<=** 10; **++**j)

{

pool.submit(multiply, i, j);

}​

*// 使用ref传递的输出参数提交函数*

**int** output\_ref;

**auto** future1 **=** pool.submit(multiply\_output, std**::**ref(output\_ref), 5, 6);​

*// 等待乘法输出完成*

future1.get();

std**::**cout **<<** "Last operation result is equals to " **<<** output\_ref **<<** std**::**endl;​

*// 使用return参数提交函数*

**auto** future2 **=** pool.submit(multiply\_return, 5, 3);​

*// 等待乘法输出完成*

**int** res **=** future2.get();

std**::**cout **<<** "Last operation result is equals to " **<<** res **<<** std**::**endl;​

*// 关闭线程池*

pool.shutdown();}​**int** main(){

example();​

**return** 0;

}

## **0x03 参考资料**

1. 1. 《C++后端线程池》：[C++ 线程池 - 敬方的个人博客 | BY Blog (wangpengcheng.github.io)](https://wangpengcheng.github.io/2019/05/17/cplusplus_theadpool/)
2. 《基于C++11的线程池（threadpool），简介且可以带任意多的参数》：[基于C++11的线程池(threadpool),简洁且可以带任意多的参数 - \_Ong - 博客园 (cnblogs.com)](https://www.cnblogs.com/lzpong/p/6397997.html)
3. 3. 《C++并发编程（从C++11到C++17）》：

[C++ 并发编程（从C++11到C++17） (paul.pub)](https://paul.pub/cpp-concurrency/)

1. [从 pthread 转换到 std::thread - SegmentFault 思否](https://segmentfault.com/a/1190000002655852)
2. 《货比三家：C++中的task based并发》：

[货比三家：C++ 中的 task based 并发 - SegmentFault 思否](https://segmentfault.com/a/1190000002706259)

1. Github-99行线程池实现：

[progschj/ThreadPool: A simple C++11 Thread Pool implementation (github.com)](https://github.com/progschj/ThreadPool)

1. 《现代C++之万能引用、完美转发、引用折叠》：

[现代C++之万能引用、完美转发、引用折叠 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/99524127)

1. 《C++11中的lambda，std::function以及std:bind》：

[C++11中的lambda，std::function以及std:bind (paul.pub)](https://paul.pub/cpp-lambda-function-bind/)

1. 《现代C++教程——告诉上手C++11/14/17/20》：

[现代 C++ 教程: 高速上手 C++ 11/14/17/20 - Modern C++ Tutorial: C++ 11/14/17/20 On the Fly (changkun.de)](https://changkun.de/modern-cpp/)