最近跟着purecpp社区里的两篇博文复习了一下两种线程池的C++实现（单任务队列、多任务队列），相比于之前那篇C++11实现线程池的博文，语言版本进行了一点小更新（C++14-C++17）。在这里结合代码记录一下学习心得。

12.12，然然首播一周年辣（嘉然小姐首播周年快乐）

## **单任务队列线程池**

用现代的C++标准库（线程+锁+条件变量）实现一个单任务队列的线程池非常简单。就像之前那篇博文里面讲的一样，原理非常简单，对新手而言最复杂的其实就是C++11众多的新语言特性罢了。

一个简易的单任务队列线程池的实现思路：在线程池构造时初始化线程数，在析构时停止线程池。对外也只需要提供提交任务的接口就够了。

### **接口设计**

### **返回类型**

**explicit** **ThreadPool**(size\_t threads **=** std**::thread::**hardware\_concurrency()); *// 构造函数*​**template<typename** F, **typename**... Args**>auto** enqueue(F **&&**f, Args **&&**...args); *// 入队接口*

入队接口enqueue()这个模板函数返回值使用了auto关键字进行推导，实际上的返回值其实是一个future。future的类型如下：

**using** return\_type **=** std**::**invoke\_result\_t**<**F, Args...**>**;std**::**future**<**return\_type**>** res;

也就是说这个future返回的类型实际上是任务的返回类型。

### **输入参数**

输入参数是一个可调用对象和它的参数，这里利用了C++11的可变参数模板来实现传递任意数量的可调用对象的参数。

### **实现**

**class** **ThreadPool** {**public:**

**explicit** ThreadPool(size\_t threads **=** std**::thread::**hardware\_concurrency());​

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

**auto** enqueue(F **&&**f, Args **&&**...args);​

**~**ThreadPool();​**private:**

std**::**vector**<**std**::thread>** workers;

std**::**queue**<**std**::**function**<void**()**>>** tasks;

std**::**mutex queue\_mutex;

std**::**condition\_variable condition;

**bool** stop;};

这个简易单任务队列线程池的成员只有一个线程组，一个任务队列。为了保证任务队列的线程安全，我们还提供了一个锁。同时我们还提供了一个条件变量，利用锁和条件变量，我们可以实现线程通知机制。线程通知机制指的是，刚开始时线程池中是没有任务的，所有的线程都等待任务的到来，当一个任务进入到线程池中，就会通知**一个**线程去处理到来的任务。

同时我们又提供了一个stop变量，用来在析构的时候停止和清理任务和线程。因为懒（高情商：RAII风格线程池，生命周期基本上与应用的生命周期一致），没有提供stop接口。

下面是具体实现：

**#include** <condition\_variable>**#include** <functional>**#include** <future>**#include** <memory>**#include** <mutex>**#include** <queue>**#include** <thread>**#include** <vector>​**namespace** Diana {​

*// \* 简易多线程单任务队列线程池，使用线程安全队列，接口更人性化。* **class** **ThreadPool** {

**public:**

**explicit** ThreadPool(size\_t threads **=** std**::thread::**hardware\_concurrency()) **:** stop(false) {

*// 根据threads数量创建多个线程* **for** (size\_t i **=** 0; i **<** threads; **++**i) {

workers.emplace\_back([**this**]() {

**for** (;;) {*// 工作线程就是一个死循环，不停查询任务队列并取出任务执行* std**::**function**<void**()**>** task;​

{

std**::**unique\_lock**<**std**::**mutex**>** lock(**this->**queue\_mutex);

**this->**condition.wait(lock,

[**this**]() {

**return** **this->**stop **||** **!this->**tasks.empty();*// 条件变量等待线程池不为空或者stop* });

**if** (**this->**stop **&&** **this->**tasks.empty())*// 线程池为空且stop，证明线程池结束，退出线程* **return**;

task **=** std**::**move(**this->**tasks.front());*// 取出任务* **this->**tasks.pop();

}​

task();*// 执行任务* }

});*// lambda表达式构建* }

}​

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

**auto** enqueue(F **&&**f, Args **&&**...args) {

**using** return\_type **=** std**::**invoke\_result\_t**<**F, Args...**>**;

**auto** task **=** std**::**make\_shared**<**std**::**packaged\_task**<**return\_type()**>>**(

std**::**bind(std**::**forward**<**F**>**(f), std**::**forward**<**Args**>**(args)...));*// 完美转发，构造任务仿函数的指针* std**::**future**<**return\_type**>** res **=** task**->**get\_future(); *// 获得函数执行的future返回* {

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

**if** (stop) {

**throw** std**::**runtime\_error("enqueue on stopped Thread pool");

}​

tasks.emplace([task **=** std**::**move(task)]() { (**\***task)(); });*// 塞入任务队列* } *// 入队列后即可解锁* condition.notify\_one(); *// 仅唤醒一个线程，避免无意义的竞争* **return** res;

}​

**~**ThreadPool() {

{

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

stop **=** true;

}

condition.notify\_all();*// 唤醒所有线程，清理任务* **for** (std**::thread** **&**worker: workers)

worker.join();*// 阻塞，等待所有线程执行结束* }​

**private:**

std**::**vector**<**std**::thread>** workers;

std**::**queue**<**std**::**function**<void**()**>>** tasks;

std**::**mutex queue\_mutex;

std**::**condition\_variable condition;

**bool** stop;

};​};*// namespace Diana*

细节上的东西，注释已经写完了。

注意在有队伍进入队列时，仅需要notify\_one()，避免无意义的线程竞争；在停止线程池时，要notify\_all()唤醒所有进程。由于线程在等待stop标志，所以当唤醒之后才会把队列中的任务取出执行直到队列为空的时候才会退出线程。

### **重构：分离队列代码，编写线程安全任务队列**

这里进行一次重构，像之前那片博文一样，封装一个线程安全队列：

**namespace** Diana {​

*// \* 线程安全队列* **template<typename** T**>**

**class** **SafeQueue** {

**public:**

**void** push(**const** T **&**item) {

{

std**::**scoped\_lock lock(mtx\_);

queue\_.push(item);

}

cond\_.notify\_one();

}​

**void** push(T **&&**item) {*// 两个push方法，此处不是万能引用而是单纯右值* {

std**::**scoped\_lock lock(mtx\_);

queue\_.push(std**::**move(item));

}

cond\_.notify\_one();

}​

**bool** pop(T **&**item) {

std**::**unique\_lock lock(mtx\_);

cond\_.wait(lock, [**&**]() {

**return** **!**queue\_.empty() **||** stop\_;

});

**if** (queue\_.empty())

**return** false;

item **=** std**::**move(queue\_.front());

queue\_.pop();

**return** true;

}​

std**::**size\_t size() **const** {

std**::**scoped\_lock lock(mtx\_);

**return** queue\_.size();

}​

**bool** empty() **const** {

std**::**scoped\_lock lock(mtx\_);

**return** queue\_.empty();

}​

**void** stop() {

{

std**::**scoped\_lock lock(mtx\_);

stop\_ **=** true;

}

cond\_.notify\_all();

}​

**private:**

std**::**condition\_variable cond\_;

**mutable** std**::**mutex mtx\_;

std**::**queue**<**T**>** queue\_;

**bool** stop\_ **=** false;

};​};*// namespace Diana*

线程安全任务队列需要注意的细节不多。注意一下这里写了两个push方法，原因是在该模板类特化后，T已经是一个具体的类型，T&&就不再是泛型编程中常见的万能引用，而是一个单纯的右值了。我们可以通过再添加一个模板成员函数来合并两个push：

**template** **<typename** U**>void** push(U**&&** item) {

{

**static\_assert**(std**::**is\_same**<**U,T**>::**value**==**true);

std**::**scoped\_lock lock(mtx\_);

queue\_.push(std**::**forward(item));

}

cond\_.notify\_one();}

要注意的是，记得判断一下U和T的类型是一样的。

利用这个安全队列，我们可以重写一下单任务队列的线程池：

**namespace** Diana {​

**using** WorkItem **=** std**::**function**<void**()**>**;

*// \* 简易多线程单任务队列线程池，使用SafeQueue线程安全队列。* **class** **SimplePool** {

**public:**

**explicit** SimplePool(size\_t threads **=** std**::thread::**hardware\_concurrency()) {

**for** (size\_t i **=** 0; i **<** threads; **++**i) {

workers\_.emplace\_back([**this**]() {

**for** (;;) {

std**::**function**<void**()**>** task;

**if** (**!**queue\_.pop(task))

**return**;​

**if** (task)

task();

}

});

}

}​

**void** enqueue(WorkItem item) {

queue\_.push(std**::**move(item));

}​

**~**SimplePool() {

queue\_.stop();

**for** (**auto&** thd: workers\_)

thd.join();

}​

**private:**

SafeQueue**<**WorkItem**>** queue\_;

std**::**vector**<**std**::thread>** workers\_;

};​};*// namespace Diana*

入队接口enqueue()史诗级简化。当然，这种简化也带来一些不便：使用std::function<void()>作为参数，不想前面的那个传入可调用对象及其参数，在实际使用时需要用户进行包装，这个会在后面功能测试的时候讲。

## **多任务队列线程池**

其实多任务队列的线程池的设计思路也是很简单的：每个线程对应着一个自己的任务队列。因为前面对任务队列进行了抽取重构，编写一个多任务队列的线程池也非常简单。

我们对原本单任务队列的线程池的入队接口进行改造：

**int** schedule\_by\_id(WorkItem fn, size\_t id **=** **-**1)

当提交一个任务时，我们可以指定它放到任意一个线程的任务队列中。在用户没有指定任务队列时，就为该任务随机选择一个线程所对应的任务队列。

### **实现**

简单多任务队列线程池实现如下：

**namespace** Diana {​

**using** WorkItem **=** std**::**function**<void**()**>**;

*// \* 简易多线程多任务队列线程池，使用SafeQueue线程安全队列。* **class** **MultiplePool** {

**public:**

**explicit** MultiplePool(size\_t thread\_num **=** std**::thread::**hardware\_concurrency())

**:** queues\_(thread\_num),

thread\_num\_(thread\_num) {

**auto** worker **=** [**this**](size\_t id) {

**while** (true) {

WorkItem task{};

**if** (**!**queues\_[id].pop(task))

**break**;​

**if** (task)

task();

}

};​

workers\_.reserve(thread\_num\_);

**for** (size\_t i **=** 0; i **<** thread\_num\_; **++**i) {

workers\_.emplace\_back(worker, i);

}

}​

**int** schedule\_by\_id(WorkItem fn, size\_t id **=** **-**1) {

**if** (fn **==** **nullptr**)

**return** **-**1;​

**if** (id **==** **-**1) {

id **=** rand() **%** thread\_num\_;*// 随机插入到一个线程的任务队列中* queues\_[id].push(std**::**move(fn));

} **else** {

assert(id **<** thread\_num\_);*// 插入指定线程的任务队列* queues\_[id].push(std**::**move(fn));

}​

**return** 0;

}​

**~**MultiplePool() {

**for** (**auto&** queue: queues\_) {

queue.stop();*// 停止每一个任务队列* }

**for** (**auto&** worker: workers\_) {

worker.join();*// 阻塞，等待每个线程执行结束* }

}​

**private:**

std**::**vector**<**Diana**::**SafeQueue**<**WorkItem**>>** queues\_;*// 每个线程对应一个任务队列* size\_t thread\_num\_;

std**::**vector**<**std**::thread>** workers\_;

};​};*// namespace Diana*

需要注意的细节也在代码注释中写明了。

## **测试**

我们编写如下测试代码：

**void** **test\_thread\_pool**() {

std**::**cout **<<** "test\_thread\_pool()" **<<** std**::**endl;

Diana**::**ThreadPool threadPool;

threadPool.enqueue([] { std**::**cout **<<** "hello\n"; });

**auto** future **=** threadPool.enqueue([](std**::**string str) { **return** "hello" **+** str; }, "world");

std**::**cout **<<** future.get() **<<** std**::**endl;}​std**::**string funA(std**::**string str) {

**return** "hello" **+** str;}​**void** test\_simple\_thread\_pool() {

std**::**cout **<<** "test\_simple\_thread\_pool()" **<<** std**::**endl;

Diana**::**SimplePool threadPool;

threadPool.enqueue([] { std**::**cout **<<** "hello\n"; });

*// \* 此处必须使用shared\_ptr进行包装，* *// \* 否则在std::function<void()>中会尝试生成std::packaged\_task的拷贝构造函数，* *// ! std::packaged\_task禁止拷贝操作* **auto** task **=** std**::**make\_shared**<**std**::**packaged\_task**<**std**::**string()**>>**(std**::**bind(funA, "world"));

std**::**future**<**std**::**string**>** res **=** task**->**get\_future();

threadPool.enqueue([task **=** std**::**move(task)] { (**\***task)(); });

*// ! 以下实现方法是错误的* *// auto task = std::packaged\_task<std::string()>(std::bind(funA, "world"));* *// std::future<std::string> res = task.get\_future();* *// threadPool.enqueue(std::move(task));* std**::**cout **<<** res.get() **<<** std**::**endl;}​**void** test\_multiple\_thread\_pool() {

std**::**cout **<<** "test\_multiple\_thread\_pool" **<<** std**::**endl;

Diana**::**MultiplePool threadPool;

threadPool.schedule\_by\_id([] { std**::**cout **<<** "hello\n"; });

**auto** task **=** std**::**make\_shared**<**std**::**packaged\_task**<**std**::**string()**>>**(std**::**bind(funA, "world"));

std**::**future**<**std**::**string**>** res **=** task**->**get\_future();

threadPool.schedule\_by\_id([task **=** std**::**move(task)] { (**\***task)(); });

std**::**cout **<<** res.get() **<<** std**::**endl;}

我们分别为三个版本的线程池（未重构的单任务队列线程池+重构后的单任务队列线程池+多任务队列线程池）编写了接口测试。

未重构的单任务队列线程池，因为接口简单，没有什么需要特别注意的东西。

重构后的单任务队列线程池和多任务队列线程池，可以看到我写了一些注释，还注释了一些错误的任务提交方式。还记得前面重构时，我们把提交任务的接口参数改成了std::function<void()>嘛？为了配合这个参数格式，以及利用future进行异步编程，我们需要对异步获取结果的方法进行包装（之前的版本在入队接口中为用户做了这些事情）。比较需要注意的一点是我们需要用一个shared\_ptr智能指针来对packaged\_task进行包装，这是因为在std::function<void()>中会尝试生成std::packaged\_task的拷贝构造函数，而std::packaged\_task是禁止进行拷贝操作的，这会引起编译器的报错（感谢群里的大佬帮我整明白了这个事情）。

[cppreference-std::funtion：](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/utility/functional/function" \t "https://zhuanlan.zhihu.com/p/_blank)  
类模板 std::function 是通用多态函数包装器。 std::function 的实例能存储、复制及调用任何*ef="[https://zh.cppreference.com/w/cpp/named\_req/CopyConstructible](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/named_req/CopyConstructible" \t "https://zhuanlan.zhihu.com/p/_blank)">可复制构造 (CopyConstructible) 的href="[https://zh.cppreference.com/w/cpp/named\_req/Callable](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/named_req/Callable" \t "https://zhuanlan.zhihu.com/p/_blank)">可调用 (Callable) 目标——函数、 [lambda 表达式](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/language/lambda" \t "https://zhuanlan.zhihu.com/p/_blank)、 [bind 表达式](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/utility/functional/bind" \t "https://zhuanlan.zhihu.com/p/_blank)或其他函数对象，还有指向成员函数指针和指向数据成员指针。*  
*存储的可调用对象被称为 std::function 的目标。若 std::function 不含目标，则称它为空。调用空* *std::function 的目标导致抛出 [std::bad\_function\_call](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/utility/functional/bad_function_call" \t "https://zhuanlan.zhihu.com/p/_blank) 异常。*  
*std::function 满足ef="[https://zh.cppreference.com/w/cpp/named\_req/CopyConstructible](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/named_req/CopyConstructible" \t "https://zhuanlan.zhihu.com/p/_blank)">可复制构造 (CopyConstructible) 和ef="[https://zh.cppreference.com/w/cpp/named\_req/CopyAssignable](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/named_req/CopyAssignable" \t "https://zhuanlan.zhihu.com/p/_blank)">可复制赋值 (CopyAssignable) 。*  
*[cppreference-std::packaged\_task构造函数：](https://link.zhihu.com/?target=https://zh.cppreference.com/w/cpp/thread/packaged_task/packaged_task" \t "https://zhuanlan.zhihu.com/p/_blank)*  
*构造新的 std::packaged\_task 对象。*  
*复制构造函数被删除， std::packaged\_task 仅可移动。*

## **性能比较**

根据性能测试，重构和未重构版本的单任务队列线程池有着细微的性能差别（未重构版本略优）。而在数据量足够大的情况下，因为多任务队列的设计，多任务队列线程池性能甚至达到了单任务队列线程池的两倍。

## **写在后面**

后面大概还会继续研究一下work steal线程池，也会写篇学习心得=。=

## **参考链接**

[purecpp - a cool open source modern c++ community](http://www.purecpp.org/detail?id=2260)

[purecpp - a cool open source modern c++ community](http://www.purecpp.org/detail?id=2261)