

线程池

bind 的好处：生成的对象可看成 无参的了

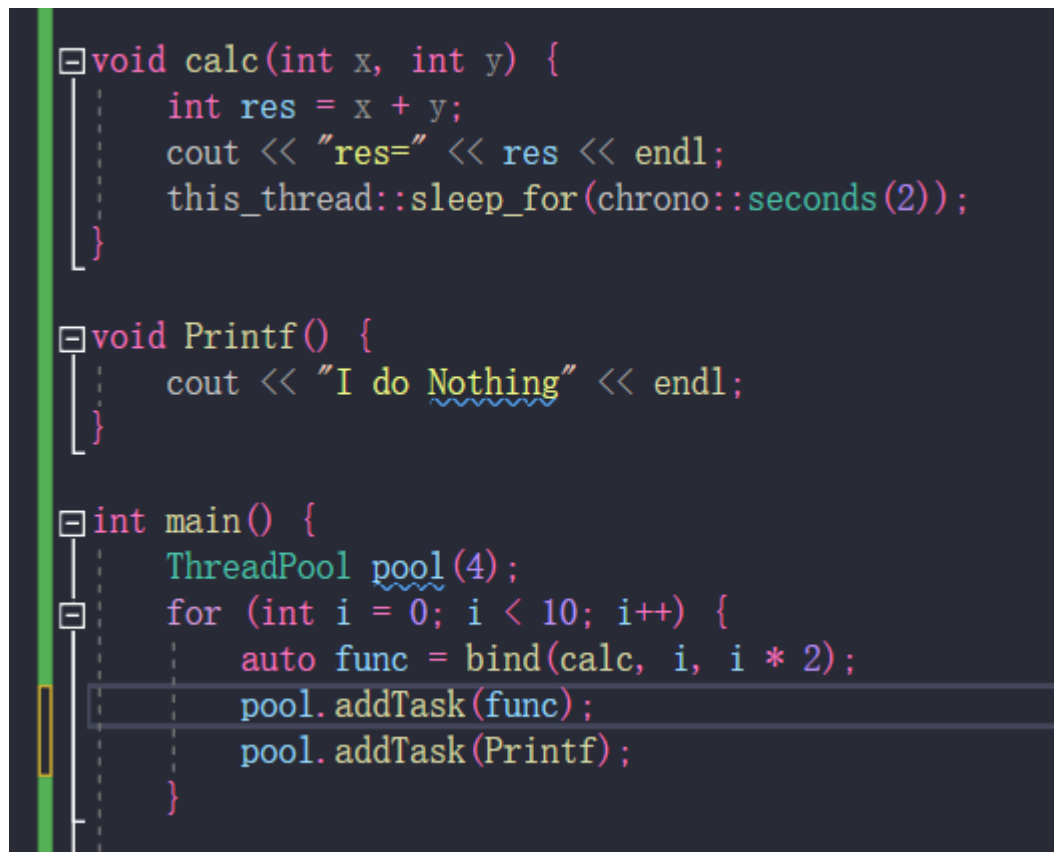
- 关于 main() 函数中的测试代码

- 通过添加任务的函数 `addTask` 可以得知任务函数返回值类型是 `void` 并且无参，但是测试程序中添加的任务函数是 `void calc(int x, int y)` 携带了两个参数，很显然和任务函数的参数是类型是不匹配的，程序中的解决方案是对函数和参数进行绑定：



这样得到的可调用对象就是无参的。通过这种方式表面看起来只能存储无参函数的任务队列就可以存储带任意参数的有参函数了。

如：`addtask`函数 接收的 `function`只能是 `void()`，所以若直接传参则只能传 下图`Printf` 函数这种；而`bind`，可以使得 `calc` 变成 `func` 对象进行传参



```
void calc(int x, int y) {
    int res = x + y;
    cout << "res=" << res << endl;
    this_thread::sleep_for(chrono::seconds(2));
}

void Printf() {
    cout << "I do Nothing" << endl;
}

int main() {
    ThreadPool pool(4);
    for (int i = 0; i < 10; i++) {
        auto func = bind(calc, i, i * 2);
        pool.addTask(func);
        pool.addTask(Printf);
    }
}
```

同步线程池

阻塞等待 子任务执行结束 1、`exitNumber`代表有线程要结束。现在:这里是如何保证，在 `m_exitnumber>0`的时候，`get_id`获取到的就是当前要退出的线程呢

这是一个很好的问题！在`worker`函数中，当`m_exitNumber > 0`时，我们希望线程能够安全地退出并减少线程池的线程数量。要理解这里如何确保`get_id()`获取的是当前要退出的线程，我们需要仔细看一下代码的逻辑。

代码逻辑分析

在`worker`函数中，当`m_exitNumber > 0`时，有一个逻辑用来处理线程的退出：

cpp

复制代码

```
if (m_exitNumber.load() > 0)
{
    cout << "----- 线程任务结束, ID: " << this_thread::get_id() << endl;
    m_exitNumber--;
    m_curThreads--;
    unique_lock<mutex> lck(m_idsMutex);
    m_ids.emplace_back(this_thread::get_id());
    return;
}
```

如何确保`get_id()`获取到当前要退出的线程

1. 每个线程独立执行其自身的`worker`函数：

- 在C++的线程池中，每个线程都会独立地执行`worker`函数。线程池在创建线程时，传入的是成员函数指针`&ThreadPool::worker`，因此每个线程都会执行自己的`worker`循环。

2. `this_thread::get_id()`获取的是当前线程的ID：

- `this_thread::get_id()`是一个标准库函数，它返回的是当前正在执行的线程的ID。因此，在一个线程的`worker`函数内部调用`this_thread::get_id()`时，它始终返回调用它的那个线程的ID。

3. `m_exitNumber > 0`的条件是由`manager`函数控制的：

- `manager`函数是一个特殊的管理线程，它监控线程池的状态（如空闲线程的数量、当前线程数量等），并在空闲线程过多时决定减少线程数量。
- 当需要减少线程时，`manager`函数通过`m_exitNumber.fetch_sub(2)`设置一个标志，表示需要两个线程退出。这不会直接指定哪些线程退出，但它在通知条件变量`m_condition.notify_all()`后，所有在等待任务的线程都会被唤醒。

4. 被唤醒的线程中，空闲线程可能会发现`m_exitNumber > 0`：

- 当条件变量`m_condition`唤醒线程时，多个线程会竞争地进入等待队列。
- 被唤醒的线程检查`m_exitNumber`，第一个发现`m_exitNumber > 0`的线程将会进入这个退出的逻辑块，执行退出流程（减少`m_exitNumber`和`m_curThreads`，然后将自身ID添加到`m_ids`中）。
- 由于每个线程都有自己独立的`worker`函数和`this_thread`上下文，所以在退出逻辑中调用`this_thread::get_id()`获取的就是当前正在执行的线程ID。

5. 通过互斥锁和原子操作确保线程安全:

- `m_exitNumber` 和 `m_curThreads` 是 `atomic` 类型的变量, 保证了在多线程环境下的原子性操作。
- `m_ids` 的修改被保护在 `mutex` (`m_idsMutex`) 的锁内, 确保多个线程不会同时修改这个容器, 从而避免数据竞争。

thread 结束? 只要其 handler 处理函数结束了, 则 thread 工作结束

```
void ThreadPool::worker() {
    while (!m_stop.load()) {
        function<void()>task = nullptr;
        {
            unique_lock<mutex>locker(m_queueMutex);
            // 只要 m_exitNumber>0 所以到这里的线程, 都会随机被销毁 (谁先 if (m_exitNumber.load() > 0) 谁就被先添加到 m_ids 中, 并直接return 结束线程的执行)
            while (!m_stop && m_tasks.empty()) {
                m_condition.wait(locker);
                if (m_exitNumber.load() > 0) {
                    cout << "-----线程" << this_thread::get_id() << "任务结束" << endl;
                    m_exitNumber--;
                    m_curThreads--;
                    unique_lock<mutex>lck(m_idsMutex);
                    m_ids.emplace_back(this_thread::get_id());
                    return;
                }
            }

            if (!m_tasks.empty()) {
                cout << "取出任务。。。" << endl;
                task = move(m_tasks.front());
                m_tasks.pop();
            }
        }
        if (task) {
            m_idleThreads--;
            task();
            m_idleThreads++;
        }
    }
}
```

(进入 `m_exitNumber>0` 判断为真的执行逻辑中后, 会执行 `return`, 然后对应的工作线程就结束工作了!!!!)

所以: 线程如何一直工作? handler 函数有 while 循环!!

异步线程池

可以在子线程 set 时就获取到结果 但是注意在 `future` 和 `promise` `packaged_task` `async` 等使用中的几个问题: 如: 下图

```
int SetValue(int Value, promise<int>p1) {  
    cout << "****Thread: " << this_thread::get_id() << "Ready for work~" << endl;  
    p1.set_value(Value);  
    this_thread::sleep_for(chrono::seconds(2));  
    cout << "****Thread: " << this_thread::get_id() << "Work done to Sleep:" << endl;  
    this_thread::sleep_for(chrono::seconds(3));  
  
    return 0;  
}  
  
int main() {  
    promise<int>P1;  
    future<int>ff;  
  
    for (int i = 0; i < 5; i++) {  
        auto fc = bind(SetValue, i * 2, ref(P1));  
        thread t1(fc);  
        ff = P1.get_future();  
        cout << "Get result:" << ff.get() << endl;  
    }  
  
    return 0;  
}
```

问题1: Promise 不能复制 不能复制 不能复制 所以在 SetValue传参时, 形参只能是 引用 & 或者 右值移动 &&

问题2: Promise 和 future的生命周期内只绑定一个 **value** 所以像这种 一个 promise future对象, 循环使用, 会抛出异常 问题3: 子线程 没有join

更改:

```
int SetValue(int Value, promise<int>&p1) {
    cout << "*****Thread: " << this_thread::get_id() << "Ready for work~" << endl;
    p1.set_value(Value);
    this_thread::sleep_for(chrono::seconds(2));
    cout << "*****Thread: " << this_thread::get_id() << "Work done to Sleep:" << endl;
    this_thread::sleep_for(chrono::seconds(3));

    return 0;
}

int main() {

    for (int i = 0; i < 5; i++) {
        promise<int>P1;
        future<int>ff;
        auto fc = bind(SetValue, i * 2, ref(P1));
        thread t1(fc);
        ff = P1.get_future();
        cout << "Get result:" << ff.get() << endl;
        t1.join();
    }

    return 0;
}
```

3. 线程异步



线程异步 (Asynchronous Threading) 是一种编程范式，用于执行任务或操作而不阻塞主线程或其他线程的执行。这种方法特别适用于需要同时处理多个操作或在后台执行长时间运行的任务的场景。线程异步的核心思想是 将耗时的操作与主执行流程分离，使得系统能够继续处理其他任务，而无需等待耗时操作完成。

- **异步执行**：与同步操作不同，异步操作不要求调用者在任务完成之前等待结果。异步操作通常会在后台线程中执行，主线程或其他线程可以继续执行其他任务。
- **线程**：在多线程编程中，异步操作通常通过 创建新的线程来实现，新线程会执行异步任务，而主线程则继续进行其他操作。

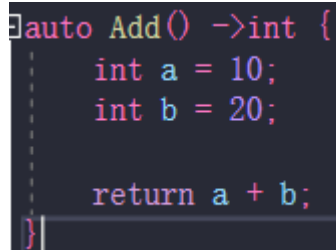
在上面的线程池代码中，如果任务函数有返回值，我们就可以通过线程异步的方式，将子线程也就是工作的线程的返回值传递给主线程，核心操作就是修改添加任务的函数 `addTask`。

该 addtask函数:

```
template<typename F, typename... Args>
auto addTask(F&& f, Args&&... args) -> future<typename result_of<F(Args...)>::type>
{
    using returnType = typename result_of<F(Args...)>::type;
    auto task = make_shared<packaged_task<returnType()>>(
        bind(forward<F>(f), forward<Args>(args)...)
    );
    future<returnType> res = task->get_future();
    {
        unique_lock<mutex> lock(m_queueMutex);
        m_tasks.emplace([task]() { (*task)(); });
    }
    m_condition.notify_one();
    return res;
}
```

1、首先第一点

返回值定义 auto 的函数，可以用



```
auto Add() ->int {
    int a = 10;
    int b = 20;

    return a + b;
}
```

-> 表明返回值是何种类型（当然你不写也行）

其它：F 就是要执行的任务函数 Args 是任务函数的参数

关于模板函数 `addTask` 的相关细节解释如下:

- 模板参数 `F` 和 `Args...` :
 - `F` 是一个类型参数, 代表任务函数的类型或函数对象的类型。这个函数或函数对象将被传递给 `addTask` 函数来执行。
 - `Args...` 是可变参数模板, 表示传递给 `F` 的参数类型。 `Args` 可以是任何数量的参数类型。
- `auto` 返回类型:
 - 返回类型是 `future<typename result_of<F(Args...)>::type>`, 表示 `addTask` 函数会返回一个 `future` 对象, 用于异步获取任务的结果。 `result_of<F(Args...)>::type` 用于推导 `F` 运行后的返回类型。
 - 通过使用 `typename`, 我们明确告诉编译 `std::result_of<F(Args...)>::type` 是一个类型, 而不是其他实体 (例如静态成员)。
- `using returnType` :
 - 使用 `result_of<F(Args...)>::type` 来获取任务函数 `F` 执行后的返回类型, 并将其命名为 `returnType`。
- `make_shared<packaged_task<returnType()>>` :
 - `std::make_shared` 是一个模板函数, 用于创建并返回一个 `std::shared_ptr`, 它以一种异常安全的方式分配和构造对象。这里, `std::make_shared` 用于创建一个指向 `std::packaged_task` 的共享指针。
 - `std::packaged_task` 是一个模板类, 用于包装一个可调用对象 (如函数、lambda 表达式、函数对象等), 使其可以异步执行, 并允许获取其执行结果。 `std::packaged_task` 提供了一个 `std::future` 对象, 通过该对象可以在任务完成后获取其结果。
 - `returnType()` 表示 `packaged_task` 将封装一个返回类型为 `returnType` 的任务。
- `bind(forward<F>(f), forward<Args>(args)...) :`
 - `std::bind` 是一个标准库函数模板, 用于绑定参数到可调用对象上, 返回一个新的可调用对象。这里, `std::bind` 绑定了传入的函数 `f` 和参数 `args...`, 生成一个不接受参数的新函数对象。
 - `std::forward` 是一个模板函数, 用于完美转发参数。 `addTask` 函数的参数 (`F&& f, Args&&... args`) 为未定引用类型, `std::forward` 保留了参数的值类别 (左值或右值), 以确保参数在转发过程中不会被不必要地拷贝或移动。
- `task->get_future()` :
 - `get_future` 返回一个 `future` 对象, 这个对象用于获取异步任务的结果。
- 任务队列:
 - `unique_lock<mutex> lock(m_queueMutex)` 用于加锁, 确保线程安全地将任务加入任务队列。
 - `m_tasks.emplace([task]() { (*task)(); });` 将任务添加到任务队列中。这里使用了一个 lambda 函数来调用 `(*task)()`, 即执行封装的任务。
- 通知条件变量:
 - `m_condition.notify_one()` 用于通知等待的线程 (如果有的话) 任务队列中有新的任务可用。

我们关键的就是 封装一个 task 对象

然后 首先 task 是一个 `shared_ptr`: 自动析构释放 其指向的对象类型是 `packaged_task`

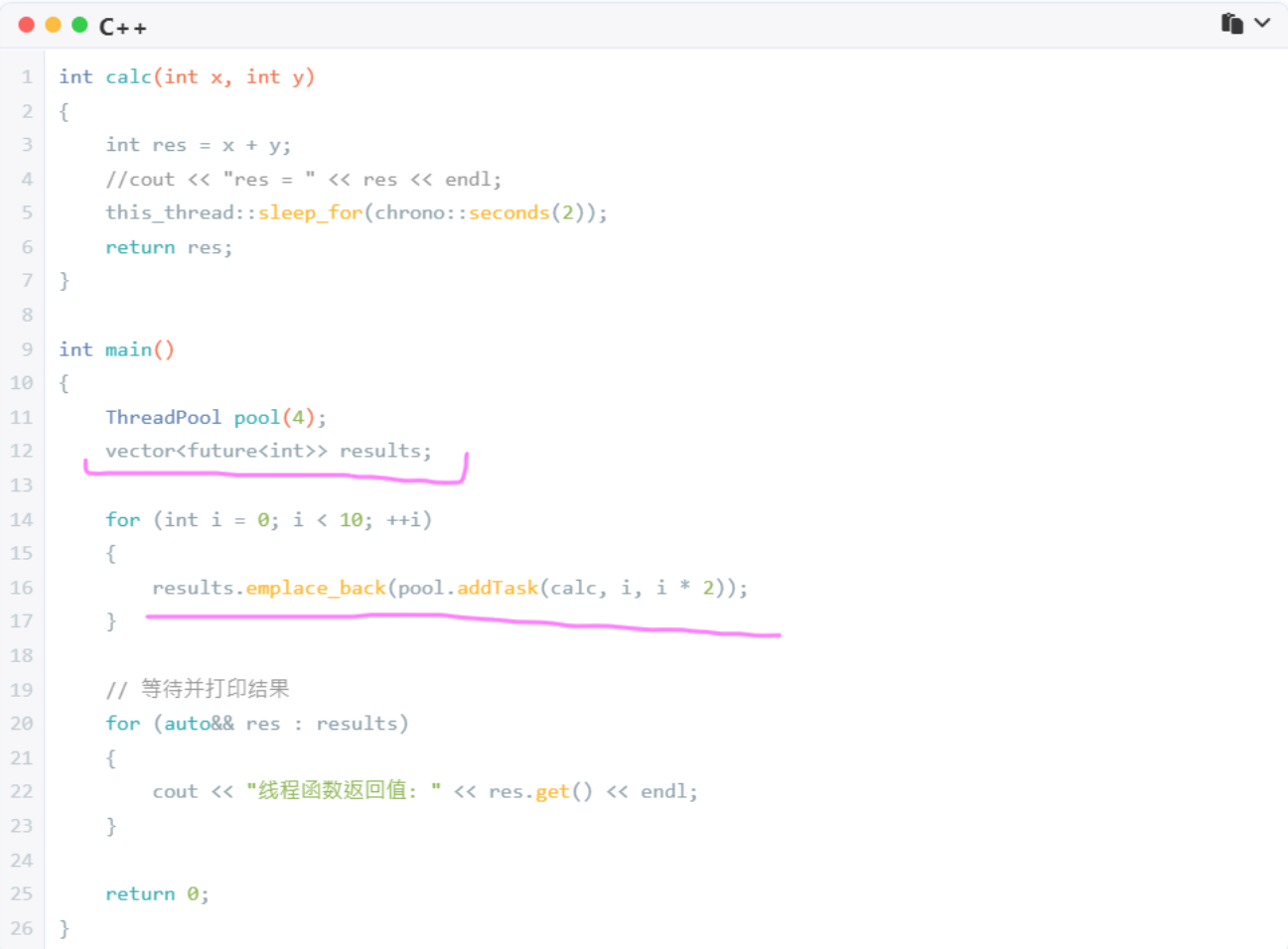
所以

```
//auto task =make_shared<类型>(初始化);  
auto task = make_shared<packaged_task<returnType()>>(bind(forward<F>(f),  
forward<Args>(args)...));
```

res = task->get_future();

task 中 执行函数 执行时, set_value 时就直接返回结果了

然后再来看一下测试代码:



```
1  int calc(int x, int y)  
2  {  
3      int res = x + y;  
4      //cout << "res = " << res << endl;  
5      this_thread::sleep_for(chrono::seconds(2));  
6      return res;  
7  }  
8  
9  int main()  
10 {  
11     ThreadPool pool(4);  
12     vector<future<int>> results;  
13  
14     for (int i = 0; i < 10; ++i)  
15     {  
16         results.emplace_back(pool.addTask(calc, i, i * 2));  
17     }  
18  
19     // 等待并打印结果  
20     for (auto&& res : results)  
21     {  
22         cout << "线程函数返回值: " << res.get() << endl;  
23     }  
24  
25     return 0;  
26 }
```