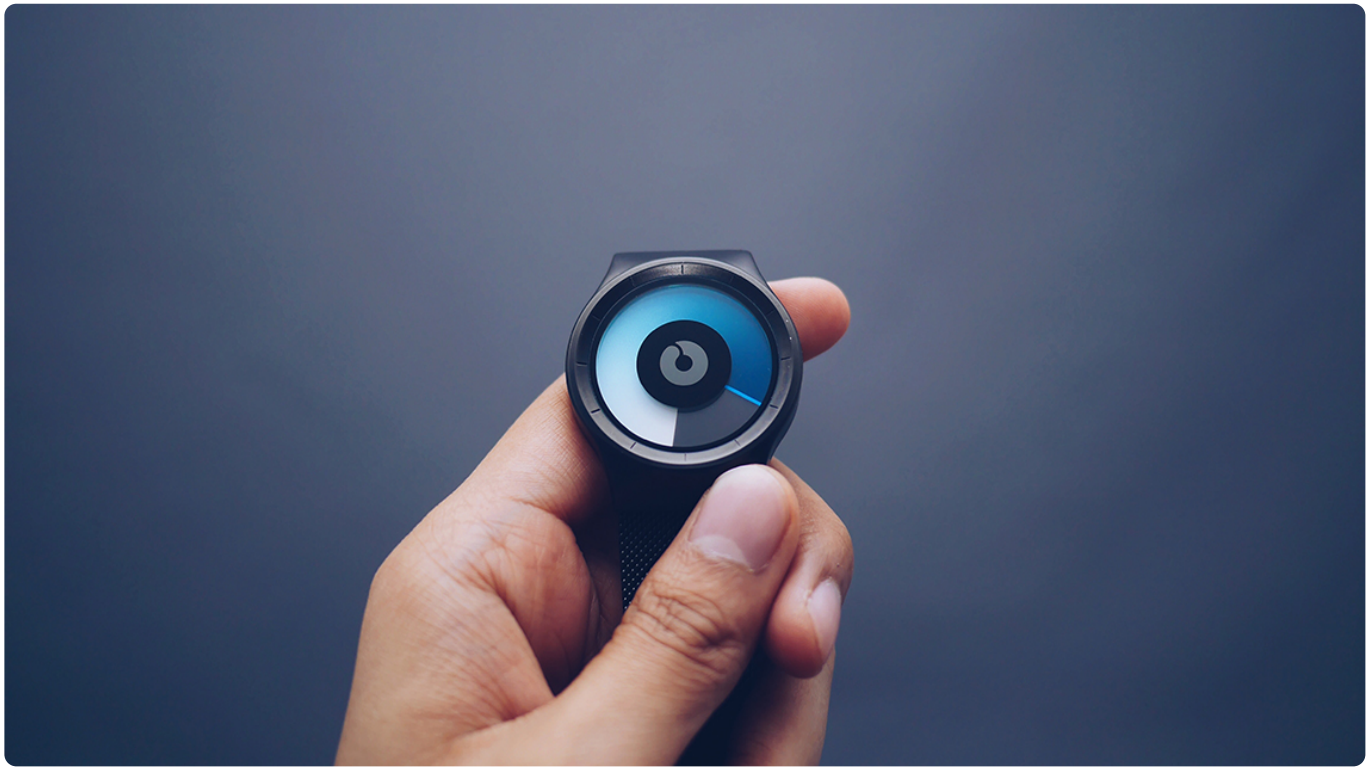
19 | thread和future：领略异步中的未来

2020-01-08 吴咏炜



讲述：吴咏炜
时长 14:31 大小 13.31M

你好，我是吴咏炜。

编译期的烧脑我们先告个段落，今天我们开始讲一个全新的话题——并发（concurrency）。

为什么要使用并发编程？

在本世纪初之前，大部分开发人员不常需要关心并发编程；用到的时候，也多半只是在单处理器上执行一些后台任务而已。只有少数为昂贵的工作站或服务器进行开发的程序员，才会需要为并发性能而烦恼。原因无他，程序员们享受着摩尔定律带来的免费性能提升，而高速的 Intel 单 CPU 是性价比最高的系统架构，可到了 2003 年左右，大家骤然发现，“免费

午餐”已经结束了 [1]。主频的提升停滞了：在 2001 年，Intel 已经有了主频 2.0 GHz 的 CPU，而 18 年后，我现在正在使用的电脑，主频也仍然只是 2.5 GHz，虽然从单核变成了四核。服务器、台式机、笔记本、移动设备的处理器都转向了多核，计算要求则从单线程变成了多线程甚至异构——不仅要使用 CPU，还得使用 GPU。

如果你不熟悉进程和线程的话，我们就先来简单介绍一下它们的关系。我们编译完执行的 C++ 程序，那在操作系统看来就是一个进程了。而每个进程里可以有一个或多个线程：

每个进程有自己的独立地址空间，不与其他进程分享；一个进程里可以有多个线程，彼此共享同一个地址空间。

堆内存、文件、套接字等资源都归进程管理，同一个进程里的多个线程可以共享使用。每个进程占用的内存和其他资源，会在进程退出或被杀死时返回给操作系统。

并发应用开发可以用多进程或多线程的方式。多线程由于可以共享资源，效率较高；反之，多进程（默认）不共享地址空间和资源，开发较为麻烦，在需要共享数据时效率也较低。但多进程安全性较好，在某一个进程出问题时，其他进程一般不受影响；而在多线程的情况下，一个线程执行了非法操作会导致整个进程退出。

我们讲 C++ 里的并发，主要讲的就是多线程。它对开发人员的挑战是全方位的。从纯逻辑的角度，并发的思维模式就比单线程更为困难。在其之上，我们还得加上：

编译器和处理器的重排问题

原子操作和数据竞争

互斥锁和死锁问题

无锁算法

条件变量

信号量

……

即使对于专家，并发编程都是困难的，上面列举的也只是部分难点而已。对于并发的基本挑战，Herb Sutter 在他的 Effective Concurrency 专栏给出了一个较为全面的概述 [2]。要对 C++ 的并发编程有全面的了解，则可以阅读曼宁出版的 *C++ Concurrency in*

Action（有中文版，但翻译口碑不好）[3]。而我们今天主要要介绍的，则是并发编程的基本概念，包括传统的多线程开发，以及高层抽象 future（姑且译为未来量）的用法。

基于 thread 的多线程开发

我们先来看一个使用 `thread` 线程类 [4] 的简单例子：

复制代码

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;

mutex output_lock;

void func(const char* name)
{
  this_thread::sleep_for(100ms);
  lock_guard<mutex> guard{
    output_lock};
  cout << "I am thread " << name
       << '\n';
}

int main()
{
  thread t1{func, "A"};
  thread t2{func, "B"};
  t1.join();
  t2.join();
}
```

这是某次执行的结果：

```
I am thread B
I am thread A
```

一个平台细节：在 Linux 上编译线程相关的代码都需要加上 `-pthread` 命令行参数。Windows 和 macOS 上则不需要。

代码是相当直截了当的，执行了下列操作：

1. 传递参数，起两个线程
2. 两个线程分别休眠 100 毫秒
3. 使用互斥量（mutex）锁定 `cout`，然后输出一行信息
4. 主线程等待这两个线程退出后程序结束

以下几个地方可能需要稍加留意一下：

> `thread` 的构造函数的第一个参数是函数（对象），后面跟的是这个函数所需的参数。
>
> `thread` 要求在析构之前要么 `join`（阻塞直到线程退出），要么 `detach`（放弃对线程的管理），否则程序会异常退出。
>
> `sleep_for` 是 `this_thread` 名空间下的一个自由函数，表示当前线程休眠指定的时间。
>
> 如果没有 `output_lock` 的同步，输出通常会交错到一起。

建议你自己运行一下，并尝试删除 `lock_guard` 和 `join` 的后果。

`thread` 不能在析构时自动 `join` 有点不那么自然，这可以算是一个缺陷吧。在 C++20 的 `jthread` [5] 到来之前，我们只能自己小小封装一下了。比如：

复制代码

```
class scoped_thread {
public:
  template <typename... Arg>
  scoped_thread(Arg&&... arg)
    : thread_(
        std::forward<Arg>(arg)...)
  {}
  scoped_thread(
    scoped_thread&& other)
    : thread_(
        std::move(other.thread_))
  {}
  scoped_thread(
    const scoped_thread&) = delete;
  ~scoped_thread()
```

```
  {
    if (thread_.joinable()) {
      thread_.join();
    }
  }

private:
  thread thread_;
};
```

这个实现里有下面几点需要注意：

1. 我们使用了可变模板和完美转发来构造 `thread` 对象。
2. `thread` 不能拷贝，但可以移动；我们也类似地实现了移动构造函数。
3. 只有 joinable（已经 `join` 的、已经 `detach` 的或者空的线程对象都不满足 joinable）的 `thread` 才可以对其调用 `join` 成员函数，否则会引发异常。

使用这个 `scoped_thread` 类的话，我们就可以把我们的 `main` 函数改写成：

复制代码

```
int main()
{
  scoped_thread t1{func, "A"};
  scoped_thread t2{func, "B"};
}
```

这虽然是个微不足道的小例子，但我们已经可以发现：

> 执行顺序不可预期，或者说不具有决定性。
>
> 如果没有互斥量的帮助，我们连完整地输出一整行信息都成问题。

我们下面就来讨论一下互斥量。

mutex

互斥量的基本语义是，一个互斥量只能被一个线程锁定，用来保护某个代码块在同一时间只能被一个线程执行。在前面那个多线程的例子里，我们就需要限制同时只有一个线程在使用 `cout`，否则输出就会错乱。

目前的 C++ 标准中，事实上提供了不止一个互斥量类。我们先看最简单、也最常用的 `mutex` 类 [6]。`mutex` 只可默认构造，不可拷贝（或移动），不可赋值，主要提供的方法是：

- `lock`：锁定，锁已经被其他线程获得时则阻塞执行
- `try_lock`：尝试锁定，获得锁返回 `true`，在锁被其他线程获得时返回 `false`
- `unlock`：解除锁定（只允许在已获得锁时调用）

你可能会想到，如果一个线程已经锁定了某个互斥量，再次锁定会发生什么？对于 `mutex`，回答是危险的未定义行为。你不应该这么做。如果有特殊需要可能在同一线程对同一个互斥量多次加锁，就需要用到递归锁 `recursive_mutex` 了 [7]。除了允许同一线程可以无阻塞地多次加锁外（也必须有对应数量的解锁操作），`recursive_mutex` 的其他行为和 `mutex` 一致。

除了 `mutex` 和 `recursive_mutex`，C++ 标准库还提供了：

- `timed_mutex`：允许锁定超时的互斥量
- `recursive_timed_mutex`：允许锁定超时的递归互斥量
- `shared_mutex`：允许共享和独占两种获得方式的互斥量
- `shared_timed_mutex`：允许共享和独占两种获得方式的、允许锁定超时的互斥量

这些我们就不做讲解了，需要的请自行查看参考资料 [8]。另外，<mutex> 头文件中也定义了锁的 RAII 包装类，如我们上面用过的 `lock_guard`。为了避免手动加锁、解锁的麻烦，以及在有异常或出错返回时发生漏解锁，我们一般应当使用 `lock_guard`，而不是手工调用互斥量的 `lock` 和 `unlock` 方法。C++ 里另外还有 `unique_lock`（C++11）和 `scoped_lock`（C++17），提供了更多的功能，你在有更复杂的需求时应该检查一下它们是否合用。

执行任务，返回数据

如果我们要在某个线程执行一些后台任务，然后取回结果，我们该怎么做呢？

比较传统的做法是使用信号量或者条件变量。由于 C++17 还不支持信号量，我们要模拟传统的做法，只能用条件变量了。由于我的重点并不是传统的做法，条件变量 [9] 我就不展开讲了，而只是展示一下示例的代码。

复制代码

```
#include <chrono>
#include <condition_variable>
#include <functional>
#include <iostream>
#include <mutex>
#include <thread>
#include <utility>

using namespace std;

class scoped_thread {
  … //  定义同上，略
};

void work(condition_variable& cv,
          int& result)
{
  //  假装我们计算了很久
  this_thread::sleep_for(2s);
  result = 42;
  cv.notify_one();
}

int main()
{
  condition_variable cv;
  mutex cv_mut;
  int result;

  scoped_thread th{work, ref(cv),
                   ref(result)};
  //  干一些其他事
  cout << "I am waiting now\n";
  unique_lock lock{cv_mut};
  cv.wait(lock);
  cout << "Answer: " << result
       << '\n';
}
```

可以看到，为了这个小小的“计算”，我们居然需要定义 5 个变量：线程、条件变量、互斥量、单一锁和结果变量。我们也需要用 `ref` 模板来告诉 `thread` 的构造函数，我们需要传递条件变量和结果变量的引用，因为 `thread` 默认复制或移动所有的参数作为线程函数的参数。这种复杂性并非逻辑上的复杂性，而只是实现导致的，不是我们希望的写代码的方式

下面，我们就看看更高层的抽象，未来量 `future` [10]，可以如何为我们简化代码。

future

我们先把上面的代码直接翻译成使用 `async` [11]（它会返回一个 `future`）：

复制代码

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

using namespace std;

int work()
{
  //  假装我们计算了很久
  this_thread::sleep_for(2s);
  return 42;
}

int main()
{
  auto fut = async(launch::async, work);
  //  干一些其他事
  cout << "I am waiting now\n";
  cout << "Answer: " << fut.get()
       << '\n';
}
```

完全同样的结果，代码大大简化，变量减到了只剩一个未来量，还不赖吧？

我们稍稍分析一下：

`work` 函数现在不需要考虑条件变量之类的实现细节了，专心干好自己的计算活、老老实实返回结果就可以了。

调用 `async` 可以获得一个未来量，`launch::async` 是运行策略，告诉函数模板 `async` 应当在新线程里异步调用目标函数。在一些老版本的 GCC 里，不指定运行策略，默认不会起新线程。

`async` 函数模板可以根据参数来推导出返回类型，在我们的例子里，返回类型是 `future<int>`。

在未来量上调用 `get` 成员函数可以获得其结果。这个结果可以是返回值，也可以是异常，即，如果 `work` 抛出了异常，那 `main` 里在执行 `fut.get()` 时也会得到同样的异常，需要有相应的异常处理代码程序才能正常工作。

这里有两个要点，从代码里看不出来，我特别说明一下：

1. 一个 `future` 上只能调用一次 `get` 函数，第二次调用为未定义行为，通常导致程序崩溃。
2. 这样一来，自然一个 `future` 是不能直接在多个线程里用的。

上面的第 1 点是 `future` 的设计，需要在使用时注意一下。第 2 点则是可以解决的。要么直接拿 `future` 来移动构造一个 `shared_future` [12]，要么调用 `future` 的 `share` 方法来生成一个 `shared_future`，结果就可以在多个线程里用了——当然，每个 `shared_future` 上仍然还是只能调用一次 `get` 函数。

promise

我们上面用 `async` 函数生成了未来量，但这不是唯一的方式。另外有一种常用的方式是 `promise` [13]，我称之为“承诺量”。我们同样看一眼上面的例子用 `promise` 该怎么写：

复制代码

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

```

```cpp
using namespace std;

class scoped_thread {
  … //  定义同上，略
};

void work(promise<int> prom)
{
  //  假装我们计算了很久
  this_thread::sleep_for(2s);
  prom.set_value(42);
}

int main()
{
  promise<int> prom;
  auto fut = prom.get_future();
  scoped_thread th{work,
                   move(prom)};
  //  干一些其他事
  cout << "I am waiting now\n";
  cout << "Answer: " << fut.get()
       << '\n';
}
```

`promise` 和 `future` 在这里成对出现，可以看作是一个一次性管道：有人需要兑现承诺，往 `promise` 里放东西（`set_value`）；有人就像收期货一样，到时间去 `future`（写到这里想到，期货英文不就是 future 么，是不是该翻译成期货量呢？😁）里拿（`get`）就行了。我们把 `prom` 移动给新线程，这样老线程就完全不需要管理它的生命周期了。

就这个例子而言，使用 `promise` 没有 `async` 方便，但可以看到，这是一种非常灵活的方式，你不需要在一个函数结束的时候才去设置 `future` 的值。仍然需要注意的是，一组 `promise` 和 `future` 只能使用一次，既不能重复设，也不能重复取。

`promise` 和 `future` 还有个有趣的用法是使用 `void` 类型模板参数。这种情况下，两个线程之间不是传递参数，而是进行同步：当一个线程在一个 `future<void>` 上等待时（使用 `get()` 或 `wait()`），另外一个线程可以通过调用 `promise<void>` 上的 `set_value()` 让其结束等待、继续往下执行。有兴趣的话，你可以自己试一下，我就不给例子了。

packaged_task

我们最后要讲的一种 `future` 的用法是打包任务 `packaged_task` [14]，我们同样给出完成相同功能的示例，让你方便对比一下：

复制代码

```
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

using namespace std;

class scoped_thread {
  … //  定义同上，略
};

int work()
{
  //  假装我们计算了很久
  this_thread::sleep_for(2s);
  return 42;
}

int main()
{
  packaged_task<int()> task{work};
  auto fut = task.get_future();
  scoped_thread th{move(task)};
  //  干一些其他事
  this_thread::sleep_for(1s);
  cout << "I am waiting now\n";
  cout << "Answer: " << fut.get()
       << '\n';
}
```

打包任务里打包的是一个函数，模板参数就是一个函数类型。跟 `thread`、`future`、`promise` 一样，`packaged_task` 只能移动，不能复制。它是个函数对象，可以像正常函数一样被执行，也可以传递给 `thread` 在新线程中执行。它的特别地方，自然也是你可以从它得到一个未来量了。通过这个未来量，你可以得到这个打包任务的返回值，或者，至少知道这个打包任务已经执行结束了。

内容小结

今天我们看了一下并发编程的原因、难点，以及 C++ 里的进行多线程计算的基本类，包括线程、互斥量、未来量等。这些对象的使用已经可以初步展现并发编程的困难，但更麻烦的事情还在后头呢……

课后思考

请试验一下文中的代码，并思考一下，并发编程中哪些情况下会发生死锁?

如果有任何问题或想法，欢迎留言与我分享。

参考资料

[1] Herb Sutter, “The free lunch is over”.
http://www.gotw.ca/publications/concurrency-ddj.htm

[2] Herb Sutter, “Effective concurrency”.
https://herbsutter.com/2010/09/24/effective-concurrency-know-when-to-use-an-active-object-instead-of-a-mutex/

[3] Anthony Williams, *C++ Concurrency in Action* (2nd ed.). Manning, 2019,
https://www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition

[4] cppreference.com, “std::thread”.
https://en.cppreference.com/w/cpp/thread/thread

[4a] cppreference.com, “std::thread”.
https://zh.cppreference.com/w/cpp/thread/thread

[5] cppreference.com, “std::jthread”.
https://en.cppreference.com/w/cpp/thread/jthread

[6] cppreference.com, “std::mutex”.
https://en.cppreference.com/w/cpp/thread/mutex

[6a] cppreference.com, “std::mutex”.
https://zh.cppreference.com/w/cpp/thread/mutex

[7] cppreference.com, “std::recursive_mutex”.
https://en.cppreference.com/w/cpp/thread/recursive_mutex

[7a] cppreference.com, “std::recursive_mutex”.
https://zh.cppreference.com/w/cpp/thread/recursive_mutex

[8] cppreference.com, “Standard library header <mutex>”.
https://en.cppreference.com/w/cpp/header/mutex

[8a] cppreference.com, “标准库头文件 <mutex>”.
https://zh.cppreference.com/w/cpp/header/mutex

[9] cppreference.com, “std::recursive_mutex”.
https://en.cppreference.com/w/cpp/thread/condition_variable

[9a] cppreference.com, “std::recursive_mutex”.
https://zh.cppreference.com/w/cpp/thread/condition_variable

[10] cppreference.com, “std::future”.
https://en.cppreference.com/w/cpp/thread/future

[10a] cppreference.com, “std::future”.
https://zh.cppreference.com/w/cpp/thread/future

[11] cppreference.com, “std::async”.
https://en.cppreference.com/w/cpp/thread/async

[11a] cppreference.com, “std::async”.
https://zh.cppreference.com/w/cpp/thread/async

[12] cppreference.com, “std::shared_future”.
https://en.cppreference.com/w/cpp/thread/shared_future

[12a] cppreference.com, “std::shared_future”.
https://en.cppreference.com/w/cpp/thread/shared_future

[13] cppreference.com, “std::promise”.
https://en.cppreference.com/w/cpp/thread/promise

[13a] cppreference.com, “std::promise”.
https://zh.cppreference.com/w/cpp/thread/promise

[14] cppreference.com, “std::packaged_task”.
https://en.cppreference.com/w/cpp/thread/packaged_task

[14a] cppreference.com, “std::packaged_task”.
https://zh.cppreference.com/w/cpp/thread/packaged_task

上一篇　18 | 应用可变模板和tuple的编译期技巧

下一篇　20 | 内存模型和atomic：理解并发的复杂性

精选留言 (6)

写留言

李公子胜治
2020-01-08

作者大大，你好，我在effective modern c++这本书上面看到，作者告诫我们平时写代码时，首先基于任务而不是线程，但是如果我们使用async时，实际上async还是为我们创建了一个新线程，还是没有体会到async比thread的优越性，难道仅仅是可以调用get()，获取async后的执行结果吗?

展开

作者回复: 少写这么多代码，还没有优越性?

新功能很多是用来提高程序员的工作效率的。而且，脑子摆脱了底层细节，就更有空去思考更高层的抽象了。否则开发里到处是羁绊，只看到这个不能做，那个很麻烦。

3

YouCompleteMe
2020-01-08

当时看<The C++ Programming>下册关于多线程的时候，还写了一些demo，现在看到future/async这些类，一点想不起来怎么用的-_-

展开

作者回复: 一定是要多用，形成“肌肉记忆”才行。光读不用是真会忘的。

2

王大为
2020-01-09

最近用google的cpplint工具扫描了我的代码，但cpplint报告说不允许包含c++11的thread头文件，请问这个是出于什么目的呢?

cpplint. py --verbose=5 my_cpp_file
output : <thread> is an unapproved c++11 header...

展开

作者回复: 那是Google的偏好。除非你为Google的项目贡献代码，理它干嘛?

1 1

三味
2020-01-10

普大喜奔!

模板章节终于结束了!
其实我还没学够呢（真心）
...
展开

作者回复: 编译期编程确实是 C++ 里比较好玩的部分，但也容易被滥用，还容易把新手全吓跑.....

舍得
2020-01-08

nice
展开

tt
2020-01-08

烧脑的编译期内容终于结束了。每天在工作之余烧一会儿，还没烧透呢，就结束了。是该庆幸还是该解脱呢?

感觉编译期编程就是C++中的理论物理，需要纸和笔，然后适应一大堆符号。
...
展开

作者回复: future的设计原则我不熟。我是挺希望跟promise联用能复用，真当成管道。但目前不支持。也许以后可以。并发方面，C++的标准机制还缺不少东西的，同步只用标准库的话很难。不过C++20会加不少新东西。

用词方面的相同应该和语言实现是否用C++没关系。并发方面有很多前沿的文献，标准术语应该早就有不少了吧。而且，标准的Python实现，CPython，是纯C写的。