07 | Coroutines背景：异步I/O的复杂性

2023-01-30 卢誉声 来自北京

《现代C++20实战高手课》

讲述：卢誉声

时长 13:05　大小 11.95M

你好，我是卢誉声。

在日常工作中，我们经常会碰到有关异步编程的问题。但由于绝大多数异步计算都跟 I/O 有关，因此在很多现代编程语言中，都支持异步编程，并提供相关的工具。

不过在 C++20 以前，异步编程从未在 C++ 语言层面得到支持，标准库的支持更是无从说起。我们往往需要借助其他库或者操作系统相关的编程接口，来实现 C++ 中的异步编程，特别是异步 I/O。比如 libuv、MFC，它们都提供了对消息循环和异步编程的支持。

接下来的三节课，我们主要讨论 C++ coroutines。这节课里，为了让你更好地理解 C++ coroutines，我们有必要先弄清楚同步与异步、并发与并行的概念以及它们之间的区别。同时，我还会跟你一起，通过传统 C++ 解决方案实现异步 I/O 编程，亲身体验一下这种实现的

复杂度。这样后面学习 C++ coroutines 的时候，你更容易体会到它的优势以及解决了哪些棘手问题（课程配套代码，点击这里即可获取）。

好了，我们话不多说，先从基本概念开始讲起。

同步与异步

同步与异步的概念比较容易理解。所谓“同步”，指的是多个相关事务必须串行执行，后续事务需要等待前一事务完成后再进行。

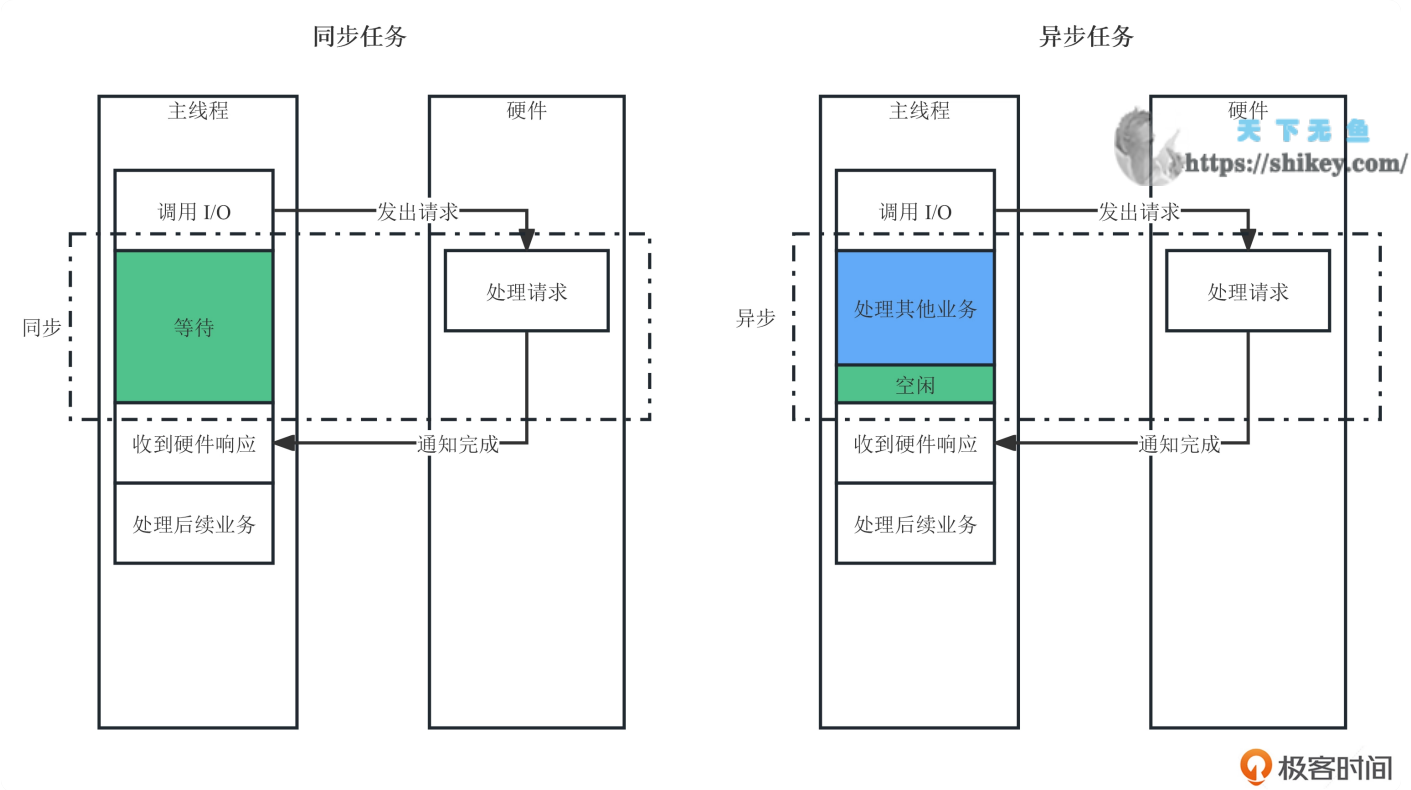
我们日常使用的 iostream，本质上就是一种同步 I/O，在发起 I/O 任务后当前线程就会一直阻塞等待，直到当前任务完成后才会继续后续任务，同时不会处理其他的任务。

与同步相对的就是异步。所谓“异步”，指的是多个相关事务可以同时发生，并通过消息机制来回调对应的事务是否执行结束。

异步常被用于网络通信以及其他 I/O 处理中，其中网络通信可以认为是一种特殊的 I/O，它通过网络适配器完成输入和输出操作。

在异步 I/O 中，发送网络请求、读写磁盘、发送中断等操作启动后并不会阻塞当前线程，当前线程还是会继续向下执行，当某个 I/O 任务完成后，程序会按照约定机制通知并发起 I/O 的任务。

通过后面这张图，可以看到同步和异步的区别。



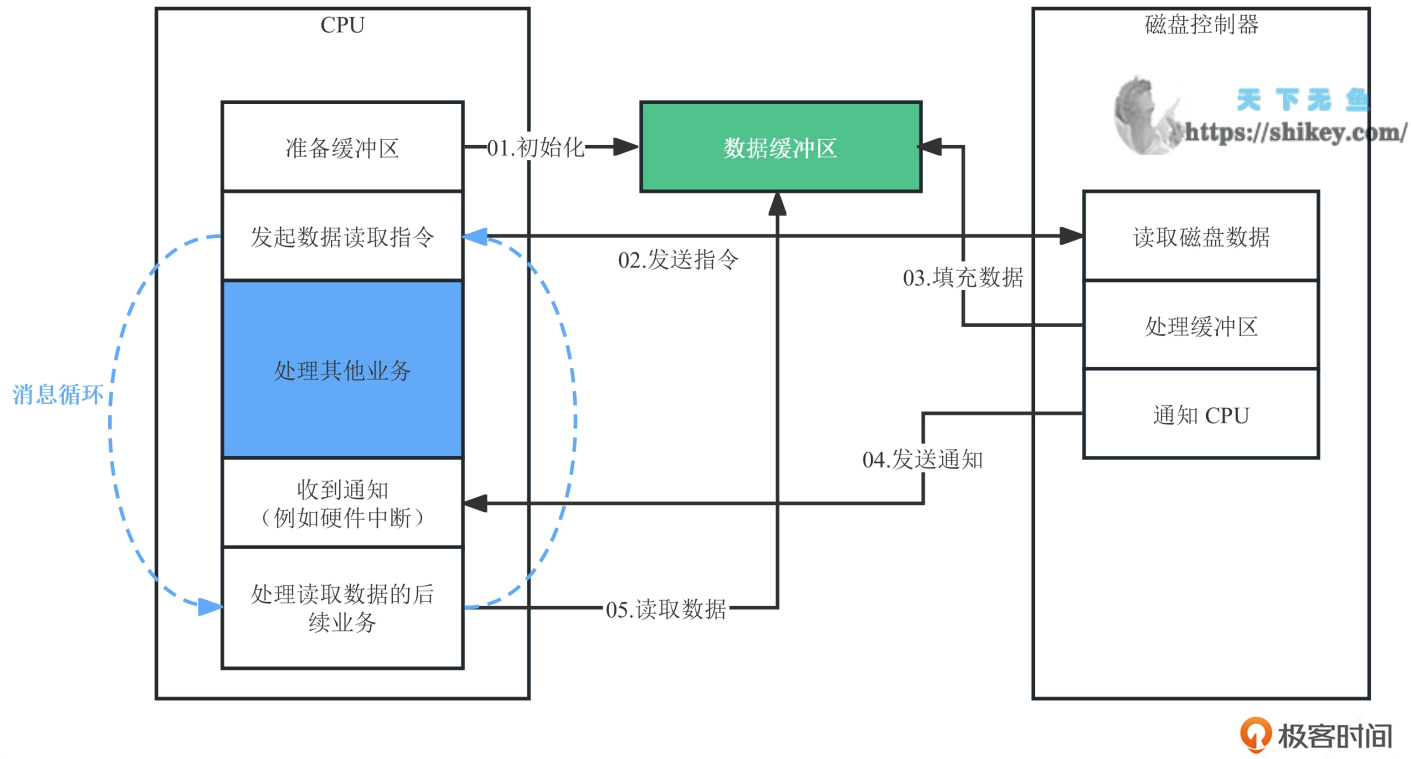
从图中我们可以看出：

- 在同步任务中，主线程向硬件发出 I/O 请求后就需要等待硬件处理完成，这个时间内不能处理其他任何业务，因此这段时间浪费了 CPU 资源。
- 而在异步任务中，主线程向硬件发出 I/O 请求后还能继续处理其他的业务。因此，我们可以充分利用硬件工作的这段时间去处理其他业务，异步模式下可以更充分地利用 CPU 资源。

因此，相较于同步，异步系统在提升计算机资源的利用率方面有着先天优势，我们在现实环境的系统中也会优先考虑使用异步来解决 I/O 问题。

从计算机体系结构的层面来看，一个程序无论是向硬件写入数据还是从硬件读取数据，都需要事先准备好一个缓冲区。在这个基础上，CPU 会通知硬件控制器将缓冲区数据写出（输出）或者将数据写入到缓冲区（输入），完成后再通过中断等方式通知 CPU。

这个过程，我同样准备了一张示意图，供你参考。



可以看到，图中的消息循环，通过缓冲区实现了异步通知和调用。这两个消息循环的执行互不干扰，没有执行的先后之分，可以充分利用计算机资源。

并发与并行

除了同步与异步，我们也经常讨论并发与并行的概念。并发与并行的本质都是一种异步的计算模式，都是指一个计算单元，在一个时段内可以同时处理多个计算任务。

不过它们之前存在差别。并发无需多个计算任务真的同时执行。程序可以选择先执行任务 A 的一部分，再执行任务 B 的一部分，然后回到任务 A 继续执行……如此往复。但是，并行要求多个任务一定是同时执行的。

因此，在出现多核心 CPU 之前，计算任务只能用并发的方法处理。那个时候为了实现并行计算，我们需要在多台计算机上以多机并行的方式处理。类似的，现如今经常讨论的分布式计算，其实属于并行计算的一种泛化。

好，我们稍微总结一下前面讲的内容：异步能够充分利用计算资源，而并行与并发为异步计算提供了不同的实现思路。

C++ 的传统异步 I/O 实现

理解了异步是怎么回事，我们还需要掌握 C++ 实现异步计算的方法，特别是在 C++ coroutines 出现之前的传统方案，这有助于我们在后续章节中，理解现代标准下的异步实现方式。

因此，我们回到异步 I/O 上，看看 C++ 到底是如何解决异步 I/O 问题的。

C++ 线程库

事实上，在 C++20 以前，C++ 标准库只提供了一种标准化异步处理技术——线程库。因此，对传统异步 I/O，我们会使用 C++11 开始提供的线程库来实现消息循环。

C++11 中线程库的核心是 std::thread 类。我们可以使用 thread 构造一个线程对象，构造函数接受一个函数或者函数对象作为参数，并立即开始执行。线程内部的具体实现，则需要依赖于操作系统底层调度。这里我给出了一个最简单的线程实例，代码是后面这样。

复制代码

```
#include <thread>
#include <iostream>
#include <cstdint>

static void count(int32_t maxValue) {
    int32_t sum = 0;
    for (int32_t i = 0; i < maxValue; i++) {
        sum += i;
        std::cout << "Value: " << i << std::endl;
    }

    std::cout << "Sum: " << sum << std::endl;
}

int main() {
    std::thread t(count, 10); // 创建线程对象，将10传递给线程的主执行函数
    t.join(); // 等待线程执行结束
    std::cout << "Join之后" << std::endl;

    return 0;
}
```

在这段代码中，我创建了一个 thread 对象 t，该线程以函数 count 为入口函数，以 10 为参数。当计算完所有数的和之后，会输出求和结果，并且退出主线程。同时，主线程启动 t 后，会使用 join 等待子线程结束，然后继续执行后续工作。最后输出截图是这样的。

简单的异步文件操作

好的，在简单回顾了 C++11 的线程库的基本用法以后，我们来看下如何实现异步文件操作。我在这里直接给出代码。

复制代码

```cpp
#include <filesystem> // C++17文件系统库
#include <iostream>   // 标准输出
#include <thread> // 线程库

namespace fs = std::filesystem;

void createDirectoriesAsync(std::string directoryPath) {
    std::cout << "创建目录: " << directoryPath << std::endl;
    fs::create_directories(directoryPath);
}

int main() {
    std::cout << "开始任务" << std::endl;

    // 创建三个线程对象
    std::thread makeDirTask1(createDirectoriesAsync, "dir1/a/b/c");
    std::thread makeDirTask2(createDirectoriesAsync, "dir2/1/2/3");
    std::thread makeDirTask3(createDirectoriesAsync, "dir3/A/B/C");

    // 等待线程执行结束
    makeDirTask1.join();
    makeDirTask2.join();
    makeDirTask3.join();
    std::cout << "所有任务结束" << std::endl;

    return 0;
}
```

在这段代码中，我们先定义了 createDirectoriesAsync 函数，该函数用于发起一个创建目录的异步任务，使用了 C++17 开始提供的文件系统库来创建目录。

接着，我们在 main 函数中创建了三个异步任务，给每个任务分别创建对应的目录。这三个任务会同时启动并独立执行。我们在这些线程对象上调用 join，让主线程等待这三个线程结束。

不难发现，虽然这种实现方式很好地解决了异步并发问题。但是，我们无法直接从线程的执行中获取处理结果，从编程的角度来看不够方便，而且线程执行过程中发生的异常也无法得到妥善处理。

为了解决这类问题，C++11 提供了更完善的线程调度和获取线程返回结果的工具，这就是 future 和 promise。类似地，我直接给出代码实现。

复制代码

```
#include <filesystem>
#include <iostream>
#include <future>
#include <exception>

namespace fs = std::filesystem;

void createDirectoriesAsync(
    std::string directoryPath,
    std::promise<bool> promise
) {
    try {
        std::cout << "创建目录: " << directoryPath << std::endl;
        bool result = fs::create_directories(directoryPath);

        promise.set_value(result);
    } catch (...) {
        promise.set_exception(std::current_exception());
        promise.set_value(false);
    }
}

int main() {
    std::cout << "开始任务" << std::endl;

    std::promise<bool> taskPromise1;
    std::future<bool> taskFuture1 = taskPromise1.get_future();
    std::thread makeDirTask1(createDirectoriesAsync, "dir1/a/b/c", std::move(ta

    std::promise<bool> taskPromise2;
    std::future<bool> taskFuture2 = taskPromise2.get_future();
```

```
    std::thread makeDirTask2(createDirectoriesAsync, "dir2/1/2/3", std::move(ta

    std::promise<bool> taskPromise3;
    std::future<bool> taskFuture3 = taskPromise3.get_future();
    std::thread makeDirTask3(createDirectoriesAsync, "dir3/A/B/C", std::move(ta

    taskFuture1.wait();
    taskFuture2.wait();
    taskFuture3.wait();

    std::cout << "Task1 result: " << taskFuture1.get() << std::endl;
    std::cout << "Task2 result: " << taskFuture2.get() << std::endl;
    std::cout << "Task3 result: " << taskFuture3.get() << std::endl;

    makeDirTask1.join();
    makeDirTask2.join();
    makeDirTask3.join();

    std::cout << "所有任务结束" << std::endl;
    return 0;
}
```


我们来比较一下这段代码实现和前面的有什么不同。可以看到，这里定义的 createDirectoriesAsync 函数多了一个参数，接受类型为 std::promise的参数。对 promise，有两个成员函数供我们调用。

- set_value 将线程的返回值返回给线程的调用者。
- set_exception 将异常返回给线程的调用者。

在 createDirectoriesAsync 函数中，我们调用 fs::create_directories 创建目录，并通过 promise.set_value 将结果记录到 promise 对象中。

为了将线程的返回值或异常信息传递出去，返回给调用者做进一步处理。我们还使用 try/catch 将执行代码块包裹起来，在发生异常时通过 std::current_exception 获取异常信息，并通过 promise.set_exception 记录把异常信息记录到 promise 对象中。

接着，在 main 函数中，每次创建 makeDirTask 之前，我们都创建了一个 promise 对象，通过 promise.get_future() 成员函数获取 promise 对应的 future 对象。每个 future 对象对应着一个线程。在后续代码中，我们通过 wait 函数进行等待，直到获取到线程的结果为止，然后通过 get 获取线程的结果。最后，在所有线程 join 后退出程序。

可以看到，我们可以通过这种模式来获取函数的处理结果和异常信息，就能实现更精细化的线程同步和线程调度。

性能资源与线程池

回顾一下前面的代码实现，我们为每个异步动作创建了一个新线程，这样的实现虽然可以正常执行，但存在两方面问题。

一方面，每个线程的创建和销毁都需要固定的性能消耗与资源消耗。异步任务粒度越小，这种固定消耗带来的影响也就越大。此外，频繁地创建和销毁小对象可能也会引发内存碎片，导致内存难以在后续程序中有效回收使用。

另一方面的问题是，并发线程过多，反而会引发整体性能下降。线程的并行能力取决于 CPU 的核心数与线程数，超过这个数量后，线程之间就无法真正并行执行了。同时，线程之间的频繁切换也会带来一定性能损耗。需要切换线程栈，重新装载指令，同时可能引发 CPU 流水线机制失效。

因此，如果用多线程实现异步，就需要知道如何控制线程的创建销毁过程以及同时执行的线程数。为了提升执行中的线程的性能，让资源利用得更充分，往往就需要使用线程池这一技术。

所谓线程池，就是一个包含固定可用线程的资源池（比如固定包含 5 个可用线程）。有了线程池，可以更充分地利用 CPU 的并行、并发能力，同时避免给系统带来不必要的负担。

那线程池的工作机制是怎样的呢？当需要启动一个异步任务的时候，我们会调用线程池的函数从这些线程中选择一个空闲线程执行任务，任务结束后将线程放回空闲线程中。如果线程池中不包含空闲线程，那么这些任务就会等待一段时间，直到有任务结束出现空闲线程为止。

除此之外，我们还需要解决多线程系统下的数据竞争问题，我们必须有一种“数据屏障”的方法，在一个线程访问竞争区域时“阻挡”其他线程的访问，避免一起访问竞争区域引发无法预料的问题。最简单的一种方案就是通过 C++11 提供的互斥锁——mutex 来解决，通常来说，我们可以通过结合 lock_guard 和 mutex 来避免多线程调用时的资源竞争问题。

总结

首先，我们在这一讲里讨论了异步的特性，异步能够充分利用计算资源，同时并行与并发为异步计算提供了不同的实现思路。

在 C++20 以前，C++ 标准库只提供了一种标准化异步处理技术——线程库。我们可以通过 promise 和 future 在父子线程之间传递处理结果与异常。但考虑到性能与资源管理问题，我们还需要借助于线程池和互斥锁来实现代码。

事实上，通过线程这种方案来处理异步 I/O 问题是相当复杂的。无论是库的实现者还是调用者，都需要考虑大量的细节。与此同时，和多线程相伴的死锁问题会成为系统中的一颗定时炸弹，在复杂的业务中随时“爆炸”，调试解决起来非常难。

那么，C++20 之后，是否有什么更好的方案来解决异步 I/O 问题呢？下一讲，就让我们来揭开协程的神秘面纱吧。

课后思考

我们在这一讲中提到了线程池这个概念，但是 C++ 标准库并不提供对线程池的封装。你能否结合 promise、future 和 mutex 来实现能够处理结构化异常的线程池？

欢迎把你的方案分享出来。我们一同交流。下一讲见！

上一篇 06 | Concepts实战：写个向量计算模板库

下一篇 08 | Coroutines：“零”开销降低异步编程复杂度

精选留言 (1)

peter

2023-01-31 来自北京

请教老师两个问题：

Q1：子线程之间可以用future和promise传递数据吗？

Q2：结构化异常是什么意思？

作者回复: Q1：子线程之间一般不会使用future和promise传递数据，你可以把A线程的future/promise给B使用也能进行数据通信，但一般来说会使用其他的线程间通信方案。

Q2：结构化异常是微软对C语言的异常处理扩展（C语言不支持异常），感兴趣可以自己看一下SEH（Structured Exception Handling）的相关内容。

链接（仅供参考）：https://learn.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-170