第3章 异步I/O


扫码观看

3.1 什么是异步I/O

3.1.1 为什么要使用异步I/O

异步I/O在Node中是非常重要的，这与Node面向网络的设计思想有很大关系。Web应用已经不再是单台服务器就能胜任的时代了，在跨网络的架构下，并发已经是现代互联网中最常见的场景了。应对高并发的场景，主要是从用户体验和资源分配这两个方面来入手。

1. 用户体验

在浏览器中JavaScript是单线程的，与UI渲染共用一个线程，如果JavaScript在执行时，UI渲染和响应是处于停滞状态的，一旦JavaScript脚本执行的时间过长，用户就会感到页面卡顿，影响用户体验。在B/S架构中，当网页需要同步获取一个网络资源时，JavaScript需要等待资源完全从服务器端获取后才能继续执行，如果受到网速的限制，UI渲染将会阻塞，不响应用户的交互行为，这对用户体验来说是个灾难。

当在Web应用中采用了异步请求，在下载资源期间，JavaScript和UI的执行都不会处于等待状态，可以继续响应用户的交互行为，给用户一个快速的体验。前端可以通过异步来消除UI阻塞的现象，但是前端获取资源的速度取决于后端的响应速度。

2. 资源分配

且不谈用户体验的因素，下面从资源分配的层面来分析一下异步I/O的必要性。当前的计算机在处理业务场景中的任务时，主要有两种方式：单线程串行依次执行和多线程并行执行。

如果创建多线程的开销小于并行执行，那么就首选多线程。多线程的开销主要体现在创建线程和执行线程上下文切换时的内存开销，另外，在复杂的业务中，多线程编程面临死锁、状态同步等问题，这也是多线程被诟病的主要原因。但多线程也有很多的优点，例如，多线程在多核CPU上能够有效提升CPU的利用率。

单线程是按顺序依次执行任务的，这一点比较符合开发人员按顺序思考的思维方式，因为易于表达，也是最主流的编程方式。但是串行执行的缺点在于性能相对较差，在代码的执行期间，一旦遇到稍微复杂的任务就会阻塞后面的代码执行。在计算机资源中，通常I/O与CPU计算之间是可以并行执行的，但是同步编程模型导致的问题是，I/O的进行会让后续任务等待，这造成资源不能被更好地利用。

单线程同步编程阻塞I/O，从而导致硬件资源得不到更好的利用，多线程编程也会因为死锁、状态同步等问题给开发人员造成困扰。Node在两者之间做了一个相对优化的解决方

案，利用 JavaScript 的单线程，避免多线程的死锁和状态同步等问题，然后利用操作系统的异步 I/O，让单线程避免阻塞，充分利用 CPU 的资料分配。Node 的最大特点就是异步 I/O 模型，这是首个将异步 I/O 使用到应用层的平台，力求在单线程上将资源分配更高效。

3.1.2 异步 I/O 与非阻塞 I/O

很多初学者会把异步和非阻塞混为一谈，这两个概念看起来似乎是一回事，从实际的应用效果来说，异步和非阻塞都达到了并行 I/O 的目的。但是从计算机内核 I/O 而言，异步、同步和阻塞、非阻塞实际上是两回事。

操作系统内核对于 I/O 只有两种方式：阻塞和非阻塞。阻塞 I/O 的特点是调用之后要等到系统内核层面完成所有操作后调用才结束，应用程序需要等待 I/O 完成后才会返回结果。效果如图 3.1 所示。

阻塞 I/O 操作 CPU 浪费了等待时间，CPU 的处理能力不能得到充分利用。为了提高性能，内核提供了非阻塞 I/O。非阻塞 I/O 跟阻塞 I/O 的差别是调用之后会立即返回。效果如图 3.2 所示。

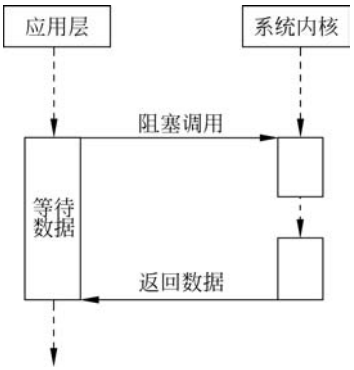

图 3.1 调用阻塞 I/O 的过程

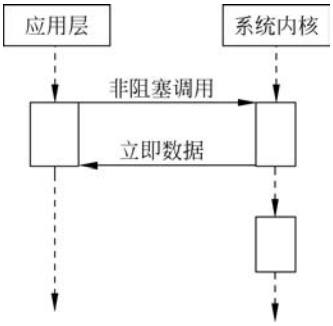

图 3.2 调用非阻塞 I/O 的过程

非阻塞 I/O 返回之后，CPU 的时间片可以用来处理其他事务，此时的性能提升是非常明显的。但是非阻塞 I/O 也存在一些问题，在完整的 I/O 还没有完成时，立即返回的并不是业务层需要的数据，而仅仅是当前调用的状态。为了获取完整的数据，应用程序需要重复调用 I/O 操作来确认是否完成。这种重复调用判断操作是否完成的技术叫作轮询。

轮询技术仅仅是满足了非阻塞 I/O 确保获取完整数据的需求，但是对于应用程序而言，它仍然只能算是一种同步，因为应用程序仍然需要等待 I/O 完全返回，依旧花费了很多时间来等待。等待期间，CPU 要么用于遍历文件描述符的状态，要么用于休眠等待事件发生。轮询的操作对性能也是非常大的消耗，并不是那么完美。

3.2 Node.js 的异步 I/O

3.2.1 事件循环

事件循环是 Node 中非常重要的执行模型。在进程启动时，Node 便会创建一个类似于

while(true)的循环，每执行一次循环体的过程称为 Tick。每个 Tick 的过程就是查看是否有事件待处理，如果有，就取出事件及其相关的回调函数。如果存在关联的回调函数，就执行它们，然后进入下个循环，如果不再有事件处理，就退出进程。事件循环流程如图 3.3 所示。

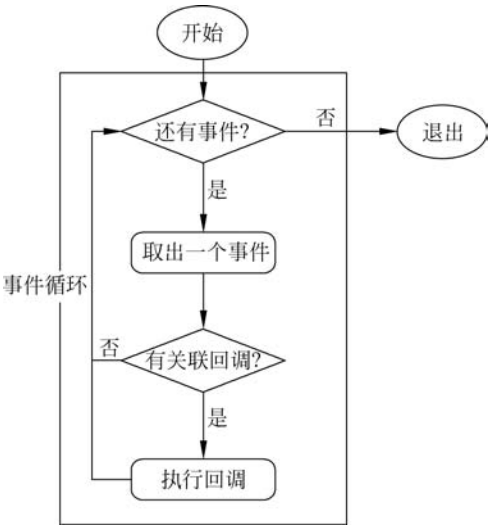

图 3.3　Tick 流程图

3.2.2　观察者模式

观察者可以在每个 Tick 的过程中判断是否需要处理事件，每个事件循环中有一个或多个观察者，而判断是否有事件要处理的过程就是在询问这些观察者是否有事件要处理。

这个过程就类似于奶茶店，奶茶店的操作间里面的员工需要不停地制作奶茶，但是做什么口感的奶茶是由前台的收银员接到的订单决定的，操作间每做完一杯奶茶，就要询问前台收银员，下一杯是什么口感的，如果没有订单了，就休息。在这个过程中，前台收银员就是观察者，收到客人点单就是关联的回调函数。当然，如果奶茶店的生意比较好，可以多雇几个收银员，这就像是多个观察者一样。接收订单是一个事件，一个观察者可以有多个事件。

浏览器采用了类似的机制，事件可能来自用户的单击或者加载某些文件时产生，而这些产生的事件都有对应的观察者。在 Node 中，事件主要来源于网络请求、文件 I/O 等，这些事件对应的观察者有文件 I/O 观察者、网络 I/O 观察者等。观察者将事件进行了分类。

事件循环是一个典型的生产者/消费者模式。异步 I/O、网络请求等则是事件的生产者，源源不断地为 Node 提供不同类型的事件，这些事件被传递到对应的观察者那里，事件循环则从观察者那里取出事件并处理。

3.2.3　请求对象

对于 Node 中的异步 I/O 调用来说，回调函数不是由开发者调用，那么从调用到回调函数被执行，中间需要请求对象。fs.open()的作用是根据指定路径和参数去打开一个文件，从而得到一个文件描述符，这是后续所有 I/O 操作的初始操作。JavaScript 层面的代码通过调用 C++核心模块进行下层的操作。

从 JavaScript 调用 Node 核心模块，核心模块调用 C++内建模块，内建模块通过 libuv 进行系统调用，这是 Node 里经典的调用方式。这里 libuv 作为封装层，有两个平台的实现，实质上是调用 uv_fs_open()方法。请求对象是异步 I/O 过程中的重要中间件，所有的状态都保存在这个对象中，包括送入线程池等待执行以及 I/O 操作完毕后的回调处理。

3.2.4　执行回调

组装好请求对象、送入 I/O 线程池等待执行，实际上完成了异步 I/O 的第一部分，回调通知是第二部分。I/O 观察者回调函数的行为就是取出请求对象的 result 属性作为参数，

取出 oncomplete_sym 属性作为方法，然后调用执行，以此达到调用 JavaScript 中传入的回调函数的目的。至此，整个异步 I/O 的流程完全结束，流程如图 3.4 所示。

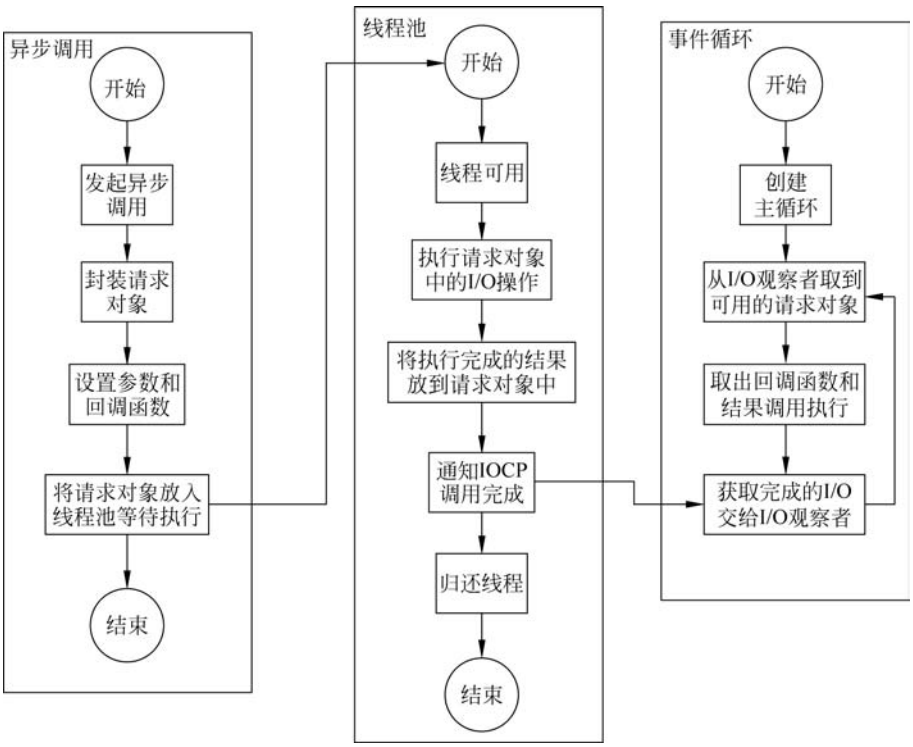

图 3.4　异步 I/O 的流程

3.3　非 I/O 的异步 API

3.3.1　定时器

浏览器 API 中有两个与定时器相关的函数，分别是 setTimeout()用于单次定时任务和 setInterval()用于多次定时任务。它们的实现原理与异步 I/O 类似，调用 setTimeout()或者是 setInterval()创建的定时器会被插入到定时器观察者内部的一个红黑树中。每次 Tick 执行时，会从该红黑树中迭代取出定时器对象，检查是否超过定时时间。如果超过，就形成一个事件，它的回调函数将立即执行。

虽然事件循环比较快，如果一次循环占用的时间较多，那么下次循环时，可能已经超时很久了。例如，使用 setTimeout()创建一个 10ms 后执行的单次执行任务，但是在 9ms 后，有一个任务占用了 5ms 的 CPU 时间片，再次轮到定时器执行时，时间就已经过期了 4ms。

3.3.2　process.nextTick()函数

由于事件循环自身的特点，定时器的精确度不够，在使用定时器时需要借助红黑树来进行定时器对象创建和迭代等操作。而 setTimeout(function,0)的方式对性能的消耗比较

大，如果想要立即异步执行一个任务，可以使用 process.nextTick()方法来完成。示例代码如下。

```
function foo() {
    console.error('foo');
}

process.nextTick(foo);
console.error('bar');
```

运行上面的代码，在控制台输出的结果如下。

```
bar
foo
```

“bar”的输出在“foo”的前面，这说明 foo() 函数是在下一个时间点运行的。每次调用 process.nextTick()方法，只会将回调函数放入队列中，在下一轮 Tick 时取出执行。定时器中采用红黑树的操作时间复杂度为 $O(\lg n)$，nextTick()的时间复杂度为 $O(1)$。相比较之后会看出 process.nextTick()的效率更高。