3.2 异步式 I/O 与事件式编程

Node.js 最大的特点就是异步式 I/O（或者非阻塞 I/O）与事件紧密结合的编程模式。这种模式与传统的同步式 I/O 线性的编程思路有很大的不同，因为控制流很大程度上要靠事件和回调函数来组织，一个逻辑要拆分为若干个单元。

3.2.1 阻塞与线程

什么是阻塞（block）呢？线程在执行中如果遇到磁盘读写或网络通信（统称为 I/O 操作），通常要耗费较长的时间，这时操作系统会剥夺这个线程的 CPU 控制权，使其暂停执行，同时将资源让给其他的工作线程，这种线程调度方式称为 *阻塞*。当 I/O 操作完毕时，操作系统将这个线程的阻塞状态解除，恢复其对CPU的控制权，令其继续执行。这种 I/O 模式就是通常的同步式 I/O（Synchronous I/O）或阻塞式 I/O（Blocking I/O）。

相应地，异步式 I/O（Asynchronous I/O）或非阻塞式 I/O（Non-blocking I/O）则针对所有 I/O 操作不采用阻塞的策略。当线程遇到 I/O 操作时，不会以阻塞的方式等待 I/O 操作的完成或数据的返回，而只是将 I/O 请求发送给操作系统，继续执行下一条语句。当操作系统完成 I/O 操作时，以事件的形式通知执行 I/O 操作的线程，线程会在特定时候处理这个事件。为了处理异步 I/O，线程必须有 *事件循环*，不断地检查有没有未处理的事件，依次予以处理。

阻塞模式下，一个线程只能处理一项任务，要想提高吞吐量必须通过多线程。而非阻塞模式下，一个线程永远在执行计算操作，这个线程所使用的 CPU 核心利用率永远是 100%，I/O 以事件的方式通知。在阻塞模式下，多线程往往能提高系统吞吐量，因为一个线程阻塞时还有其他线程在工作，多线程可以让 CPU 资源不被阻塞中的线程浪费。而在非阻塞模式下，线程不会被 I/O 阻塞，永远在利用 CPU。多线程带来的好处仅仅是在多核 CPU 的情况下利用更多的核，而Node.js的单线程也能带来同样的好处。这就是为什么 Node.js 使用了单线程、非阻塞的事件编程模式。

图3-3 和图3-4 分别是多线程同步式 I/O 与单线程异步式 I/O 的示例。假设我们有一项工作，可以分为两个计算部分和一个 I/O 部分，I/O 部分占的时间比计算多得多（通常都是这样）。如果我们使用阻塞 I/O，那么要想获得高并发就必须开启多个线程。而使用异步式 I/O 时，单线程即可胜任。

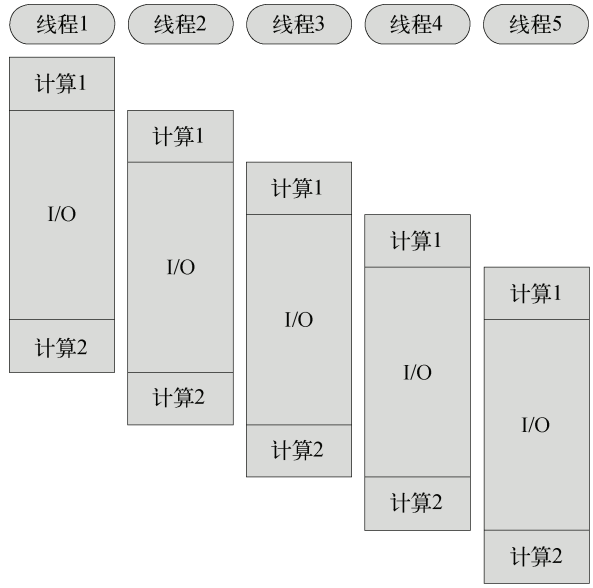

图3-3　多线程同步式 I/O

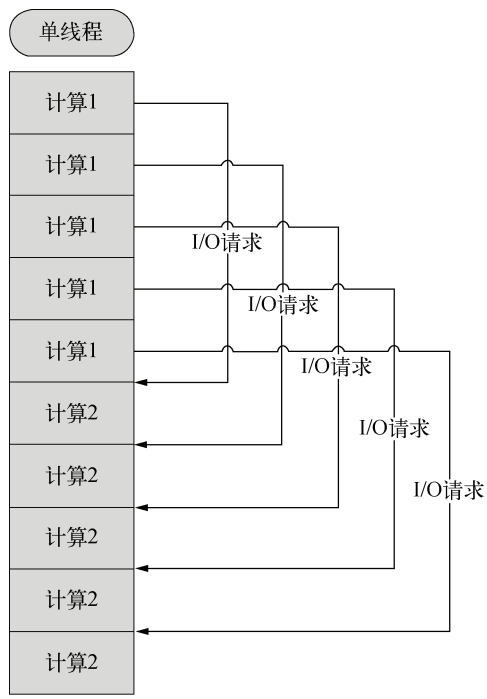

图3-4　单线程异步式 I/O

单线程事件驱动的异步式 I/O 比传统的多线程阻塞式 I/O 究竟好在哪里呢？简而言之，异步式 I/O 就是少了多线程的开销。对操作系统来说，创建一个线程的代价是十分昂贵的，需要给它分配内存、列入调度，同时在线程切换的时候还要执行内存换页，CPU 的缓存被清空，切换回来的时候还要重新从内存中读取信息，破坏了数据的局部性。[①]

当然，异步式编程的缺点在于不符合人们一般的程序设计思维，容易让控制流变得晦涩难懂，给编码和调试都带来不小的困难。习惯传统编程模式的开发者在刚刚接触到大规模的异步式应用时往往会无所适从，但慢慢习惯以后会好很多。尽管如此，异步式编程还是较为困难，不过可喜的是现在已经有了不少专门解决异步式编程问题的库（如async），参见6.2.2节。

表3-1比较了同步式 I/O 和异步式 I/O 的特点。

表3-1 同步式 I/O 和异步式 I/O 的特点

同步式 I/O（阻塞式）	异步式 I/O（非阻塞式）
利用多线程提供吞吐量	单线程即可实现高吞吐量
通过事件片分割和线程调度利用多核CPU	通过功能划分利用多核CPU
需要由操作系统调度多线程使用多核 CPU	可以将单进程绑定到单核 CPU
难以充分利用 CPU 资源	可以充分利用 CPU 资源
内存轨迹大，数据局部性弱	内存轨迹小，数据局部性强
符合线性的编程思维	不符合传统编程思维

3.2.2 回调函数

让我们看看在 Node.js 中如何用异步的方式读取一个文件，下面是一个例子：

```
//readfile.js

var fs = require('fs');
fs.readFile('file.txt', 'utf-8', function(err, data) {
  if (err) {
    console.error(err);
  } else {
    console.log(data);
  }
});
console.log('end.');
```

运行的结果如下：

```
end.
Contents of the file.
```

① 基于多线程的模型也有相应的解决方案，如轻量级线程（lightweight thread）等。事件驱动的单线程异步模型与多线程同步模型到底谁更好是一件非常有争议的事情，因为尽管消耗资源，后者的吞吐率并不比前者低。