# JavaScript 异步编程 与异步式I/O

和JavaSript线程异步同步相同的还有 对I/O的处理。

I/O的处理包括同步式I/O（阻塞式I/O） 与 异步式I/O（非阻塞式I/O）。

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

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。

JavaScript语言的执行环境是"单线程"（single thread）。

所谓"单线程"，就是指一次只能完成一件任务。如果有多个任务，就必须排队，前面一个任务完成，再执行后面一个任务，以此类推。

这种模式的好处是实现起来比较简单，执行环境相对单纯；坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应（假死），往往就是因为某一段javascript代码长时间运行（比如死循环），导致整个页面卡在这个地方，其他任务无法执行。

为了解决这个问题，Javascript语言将任务的执行模式分成两种：同步（Synchronous）和异步（Asynchronous）。

"同步模式"就是上一段的模式，后一个任务等待前一个任务结束，然后再执行，程序的执行顺序与任务的排列顺序是一致的、同步的；"异步模式"则完全不同，每一个任务有一个或多个回调函数（callback），前一个任务结束后，不是执行后一个任务，而是执行回调函数，后一个任务则是不等前一个任务结束就执行，所以程序的执行顺序与任务的排列顺序是不一致的、异步的。

"异步模式"非常重要。在浏览器端，耗时很长的操作都应该异步执行，避免浏览器失去响应，最好的例子就是Ajax操作。在服务器端，"异步模式"甚至是唯一的模式，因为执行环境是单线程的，如果允许同步执行所有http请求，服务器性能会急剧下降，很快就会失去响应。

本文总结了"异步模式"编程的4种方法，理解它们可以让你写出结构更合理、性能更出色、维护更方便的Javascript程序。

一、回调函数

这是异步编程最基本的方法。

假定有两个函数f1和f2，后者等待前者的执行结果。

　　f1();

　　f2();

如果f1是一个很耗时的任务，可以考虑改写f1，把f2写成f1的回调函数。

　　function f1(callback){

　　　　setTimeout(function () {

　　　　　　// f1的任务代码

　　　　　　callback();

　　　　}, 1000);

　　}

执行代码就变成下面这样：

　　f1(f2);

采用这种方式，我们把同步操作变成了异步操作，f1不会堵塞程序运行，相当于先执行程序的主要逻辑，将耗时的操作推迟执行。

回调函数的优点是简单、容易理解和部署，缺点是不利于代码的阅读和维护，各个部分之间高度耦合（Coupling），流程会很混乱，而且每个任务只能指定一个回调函数。

二、事件监听

另一种思路是采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。

还是以f1和f2为例。首先，为f1绑定一个事件（这里采用的jQuery的写法）。

　　f1.on('done', f2);

上面这行代码的意思是，当f1发生done事件，就执行f2。然后，对f1进行改写：

　　function f1(){

　　　　setTimeout(function () {

　　　　　　// f1的任务代码

　　　　　　f1.trigger('done');

　　　　}, 1000);

　　}

f1.trigger('done')表示，执行完成后，立即触发done事件，从而开始执行f2。

这种方法的优点是比较容易理解，可以绑定多个事件，每个事件可以指定多个回调函数，而且可以"去耦合"（Decoupling），有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰。

三、发布/订阅

上一节的"事件"，完全可以理解成"信号"。

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"（publish）一个信号，其他任务可以向信号中心"订阅"（subscribe）这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式"（publish-subscribe pattern），又称"观察者模式"（observer pattern）。

这个模式有多种实现，下面采用的是Ben Alman的Tiny Pub/Sub，这是jquery的一个插件。

首先，f2向"信号中心"jQuery订阅"done"信号。

　　jQuery.subscribe("done", f2);

然后，f1进行如下改写：

　　function f1(){

　　　　setTimeout(function () {

　　　　　　// f1的任务代码

　　　　　　jQuery.publish("done");

　　　　}, 1000);

　　}

jQuery.publish("done")的意思是，f1执行完成后，向"信号中心"jQuery发布"done"信号，从而引发f2的执行。

此外，f2完成执行后，也可以取消订阅（unsubscribe）。

　　jQuery.unsubscribe("done", f2);

这种方法的性质与"事件监听"类似，但是明显优于后者。因为我们可以通过查看"消息中心"，了解存在多少信号、每个信号有多少订阅者，从而监控程序的运行。

四、Promises对象

Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供统一接口。

简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数。比如，f1的回调函数f2,可以写成：

　　f1().then(f2);

f1要进行如下改写（这里使用的是jQuery的实现）：

　　function f1(){

　　　　var dfd = $.Deferred();

　　　　setTimeout(function () {

　　　　　　// f1的任务代码

　　　　　　dfd.resolve();

　　　　}, 500);

　　　　return dfd.promise;

　　}

这样写的优点在于，回调函数变成了链式写法，程序的流程可以看得很清楚，而且有一整套的配套方法，可以实现许多强大的功能。

比如，指定多个回调函数：

　　f1().then(f2).then(f3);

再比如，指定发生错误时的回调函数：

　　f1().then(f2).fail(f3);

而且，它还有一个前面三种方法都没有的好处：如果一个任务已经完成，再添加回调函数，该回调函数会立即执行。所以，你不用担心是否错过了某个事件或信号。这种方法的缺点就是编写和理解，都相对比较难。