# 异步I/O

## 为什么要异步I/O？

关于异步I/O 为何在Node里如此重要， 这与Node面向网络而设计不无关系。web应用已经不再是单台服务器就能胜任的时代了，在跨网络的结构下，并发已经是现代编程中的标准配备了。

## 用户体验

异步的概念之所以首先在web2.0时代火起来，是因为浏览器中JavaScript在单线程上执行，而且他还与UI渲染共用一个线程。
这意味着JavaScript在执行的时候UI渲染和响应是出于停滞状态的。如果脚本的执行时间超过**100ms**,用户就会感到页面卡顿，以为网页停止响应，而在B/S模型中，网络速度的限制给网页的实时体验造成很大的麻烦，如果网页临时请求一个资源，通过同步的方式获取，那么JavaScript要等待资源完全从服务器端获取后才能继续执行的话，这期间UI将停顿，不响应用户的交互行为。可知这样的用户体验会有多差。

**而采用异步请求的方法，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，给用户一个鲜活的页面。**
同步的话，两个资源的请求时间为 两者时间之和。
异步的话，两个资源的请求时间 取决于其中一个请求时间最长的资源时间。

## 资源分配

根据用户体验的因素，我们从资源分配的角度来分析一下异步I/O的必要性。

单线程同步编程会因阻塞I/O导致硬件资源得不到更优的使用，多线程编程模型会因编程中的死锁，状态同步等问题而让开发人员头秃。

所以Node在两者之间给出了方案，利用单线程，远离多线程死锁，状态同步等问题，利用异步I/O, 让单线程远离阻塞，以更好的利用CPU。

异步I/O可算是Node的特色，因为它是首个大规模将异步I/O应用在应用层上的平台。异步I/O的提出是期待I/O的调用不再阻塞后续运算，将原有等待I/O完成的这段时间分配给其他需要的业务去执行。**1**

## 异步I/O与非阻塞I/O

同步与异步关注的是消息的通知方式。
阻塞与非阻塞关注的是等待结果(返回值)时的状态
同步阻塞方式：

比如，你打电话问老婆今晚吃什么，老婆在电话那边一直想啊想，你在电话这边不干别的，就一直等啊等，电话始终未挂，直到她说吃火锅，电话才挂掉。

同步非阻塞方式：

比如，你打电话问老婆今晚吃什么，老婆在电话那边一直想啊想，你在电话这边该干什么干什么，电话始终未挂，直到她说吃火锅，电话才挂掉。

异步阻塞方式：

比如，你打电话问老婆今晚吃什么，老婆说我想想，过一会跟你打话。你在电话这边什么也没干，就一直等着这个电话。

异步非阻塞方式：

比如，你打电话问老婆今晚吃什么，老婆说我想想，过一会跟你打话。你在电话这边想干什么干什么，如果有电话来了，再处理电话。

操作系统内核对于I/O只有两种方式： 阻塞与非阻塞，在调用阻塞I/O时，应用程序需要等待I/O完成才返回结果。**2**

在进行文件I/O时，非阻塞I/O和阻塞I/O的区别在于阻塞I/O完成整个获取数据的过程，而非阻塞I/O则不带数据返回，要获取数据，还需要通过文件描述符再次获取。

非阻塞I/O存在的问题： 因为返回的并不是业务层期望的数据，而只是状态，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，这种技术叫**轮询 3**

任何技术都并非完美的，阻塞I/O造成CPU等待浪费，非阻塞I/O带来的麻烦却是需要轮询去确认数据是否获取，他会让CPU处理状态判断，是对CPU资源的浪费。

轮询技术满足了非阻塞I/O确保获取完整数据的需求，但是对于应用程序来说，他仍然算是一种同步，因为应用程序依然需要等待I/O完全返回，依旧花费了很多时间等待。

## 理想的非阻塞异步I/O

我们期待的完美的异步I/O应该是应用程序发起非阻塞调用，无须通过遍历或者事件唤醒等方式轮询，可以直接处理下一个任务，只需在I/O完成后通过信号或回调将数据传递给应用程序即可 **4**

## Node的异步I/O

Node异步I/O模型的四要素： **事件循环、观察者、请求对象、线程池。**

**事件循环**

在进程启动时，Node便会创建一个类似于while（true）的循环，每次执行一次循环体的过程称之为**Tick**, 每个Tick的过程就是查看是否有事件待处理，有就取出事件及相关的回调函数并执行，然后进入下一个循环，如果没有就退出进程。**5**

观察者

在每个Tick的过程中，判断是否有事件需要处理，需要一个概念，观察者。每个事件循环中都有一个或者多个观察者，而判断是否有事件要处理的过程就是向这些观察者询问是否有要处理的事件。

形象的例子：
这个过程就相当于饭馆的厨房，厨房一轮一轮的制作菜肴，但是要具体制作哪些菜肴取决于收银台收到的客人的下单,厨房每做完一轮菜肴，就去问收银台的小妹，接下来有没有要做的菜，没有就下班打烊了。在这个过程中，收银台的小妹就是观察者，她收到的客人点单就是关联的回调函数，当然也有可能饭馆有多个收银员，也就如同事件循环中多个观察者，收到下单就是一个事件，一个观察者里面可能有多个事件。

在Node中，事件主要来源网络请求，文件I/O，对应观察者就是文件I/O观察者，网络I/O观察者。

## 请求对象

对于Node中的异步调用而言，回调函数却不由开发者来调用，那么从我们发出调用后，到回调函数被执行，中间发生了什么呢？
事实上，从JavaScript发起调用到内核执行完I/O操作的过渡过程中，存在一种中间产物： 请求对象

从JavaScript层传入的参数和当前方法都被封装在这个请求对象中，回调函数则被设置在这个对象的属性上。

请求对象是异步I/O过程中的重要中间产物，所有的状态都被保存在这个对象中，包括送入线程池等待执行I/O操作完毕后的回调处理。

整个异步I/O的过程： **6**

## 小结

我们知道JavaScript是单线程的，所以按常识很容易理解为它不能充分利用多核CPU， 事实上，在Node中，除了JavaScript是单线程外，Node本身是多线程的，只是I/O线程使用的CPU较少。另一个需要重视的观点则是，除了用户代码无法并行执行外，所有的I/O(磁盘I/O，网络I/O等)则是可以并行起来的。

## 事件驱动

事件驱动例子：
在美国去看医生，需要填写大量表格，比如保险、个人信息之类，传统的基于线程的系统（thread-

based system），接待员叫到你，你需要在前台填写完成这些表格，你站着填单，而接待员坐着看你填单。你让接待员没办法接待下一个客户，除非完成你的业务。

想让这个系统能运行的快一些，只有多加几个接待员，人力成本需要增加不少。

基于事件的系统（event-based system）中，当你到窗口发现需要填写一些额外的表格而不仅仅是挂个号，接待员把表格和笔给你，告诉你可以找个座位填写，填完了以后再回去找他。你回去坐着填表，而接待员开始接待下一个客户。你没有阻塞接待员的服务。

你填完表格，返回队伍中，等接待员接待完现在的客户，你把表格递给他。如果有什么问题或者需要填写额外的表格，他给你一份新的，然后重复这个过程。

这个系统已经非常高效了，几乎大部分医生都是这么做的。如果等待的人太多，可以加入额外的接待员进行服务，但是肯定要比基于线程模式的少得多。

事件驱动例子：

1，你用浏览器访问nodejs服务器上的"/about.html"

2，nodejs服务器接收到你的请求，调用一个函数从磁盘上读取这个文件。

3，这段时间，nodejs webserver在服务后续的web请求。

4，当文件读取完毕，有一个回调函数被插入到nodejs的服务队列中。

5，nodejs webserver运行这个函数，实际上就是渲染（render）了about.html页面返回给你的浏览器。

## 几种经典的服务器模型：

1：同步式：
对于同步式的服务，一次只能处理一个请求，并且其余请求都处于等待状态。
2：每进程/每请求。 为每一个请求启动一个进程，这样可以处理多个请求，但是它不具备扩展性，因为系统资源就那么多。
3：每线程/每请求。为每个请求启动一个线程，尽管线程比进程要轻量，但是由于每个线程需要占用一定的内存，当大并发请求到来时，内存很快会用光，导致服务器缓慢。

Node的方式：

Node通过事件驱动的方式处理请求，无须为每一个请求创建额外的对应线程，可以省掉创建线程和销毁线程的开销。