第二章：Node 介绍

本章内容：

- 什么是事件驱动编程,有什么优点
- Node.js 如何简化了事件驱动开发

在传统程编程模里，I/O 操作就像一个普通的本地函数调用：在函数执行完之前程序被堵塞，无法继续运行。堵塞 I/O 起源于早先的时间片模型，这种模型下每个进程就像一个独立的人，目的是将每个人区分开，而且每个人在同一时刻通常只能做一件事，必须等待前面的事做完才能决定下一件事做什么。但是这种在计算机网络和 Internet 上被广泛使用的“一个用户，一个进程”的模型伸缩性很差。管理多个进程时，会耗费很多内存，上下文切换也会占用大量资源，这些对操作系统是个很大的负担，而且随着进程数的递增，会导致系统性能急剧衰减。

多线程是个替代方案，线程是一个轻量级的进程，它会和同一个进程内的其它线程共享内存，它更像传统模型的扩展，用来并发执行多个线程，当一个线程等待 I/O 操作时，其它线程可以接管 CPU，当 I/O 操作完成，前面等待的线程会被唤醒。就是说，一个运行中的线程可以被中断，然后稍候再被恢复。此外，在一些系统下线程可以在多核 CPU 的不同核心下并行运行。

程序员并不知道线程会在什么具体时间运行，他们必须很小心的处理共享内存的并发访问，因此必须使用一些同步原语来同步访问某个数据结构，比如使用锁或信号量，以此来强制线程以特定的行为和计划执行。那些大量依赖线程间的共享状态的应用程序，很容易就会出现一些随机性很强，难以查找的奇怪问题。

还有一种方式是使用多线程协作，由你自己负责显式的释放 CPU，并把 CPU 时间交给其他线程使用，因为由你亲自来控制线程的执行计划，因此减小了对同步的需求，但是也提高了程序的复杂度和出错的机会，而且并没有避免多线程的那些问题。

什么是事件驱动编程

事件驱动编程（Evnet-driven programming）是一种编程风格，由事件来决定程序的执行流程，事件由事件处理器（event handler）或事件回调（event callback）来处理，事件回调是当某个特定事件发生时被调用的函数，比如数据库返回了查询结果或者用户单击了一个按钮。

回想下，在传统的堵塞 I/O 编程模式里，数据库查询可能像这样：

```
result = query('SELECT * FROM posts WHERE id = 1');
do_something_with(result);
```

上面的 query 函数会让当前线程或进程一直处于等待状态，直到底层数据库完成查询操作并返回。

在事件驱动模型里，这个查询会变成这样：

```
query_finished = function(result) {
```

```
    do_something_with(result);
}
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

首先你定义了一个叫 query_finished 的函数，它包含了查询完成后要做的事。然后把这个函数当做参数传递给 query 函数，当 query 执行完毕会调用 query_finished，而不是仅仅返回查询结果。

当你感兴趣的事件发生时会调用你定义的函数，而不是简单的返回结果值，这种编程模型就叫事件驱动编程或异步编程。这是 Node 一个最明显的特性，这种编程模型意味着当前进程在执行 I/O 操作时不会被阻塞，因此，多个 I/O 操作可以并行执行，当操作完成后相应的回调函数就会被调用。

事件驱动编程底层依赖于事件循环（event loop），事件循环基本上是事件检测和事件处理器触发这两种函数不断循环调用的一个结构。在每次循环里，事件循环机制需要检测发生了哪些事件，当事件发生时，它找到对应的回调函数并调用它。

事件循环只是运行在进程内的一个线程，当事件发生时，事件处理器可以单独运行并且不会被中断，也就是说：

- 在某个特定时刻最多有一个事件回调函数运行
- 任何事件处理器运行时都不会被中断

有了这个，开发人员就可以不再为线程同步和并发修改共享内存这些事头疼了。

一个众所周知的秘密：

很久以前，系统编程社区的人们就知道事件驱动编程是创建高并发服务最佳方式，因为它不用保存很多上下文，因此节省了大量内存，也没有那么多上下文切换，又节省了大量执行时间。

慢慢的，这种理念渗透到了其他的平台和社区，出现了一些有名的事件循环实现，比如 Ruby 的 Event machine, Perl 的 AnyEvnet, 以及 Python 的 Twisted，除了这些还有很多其它的实现和语言。

用这些框架做开发，需要学习框架相关的特定知识以及框架特定的类库，比如，使用 Event Machine 时，为了享受非阻塞带来的好处，你得避免使用同步类库，只能用 Event Machine 的异步类库。如果你使用了任何阻塞类库（比如 Ruby 的大多数标准库），你的服务器就失去了最佳的伸缩性，因为事件循环依然会不断地被阻塞，时不时地阻碍了 I/O 事件的处理。

Node 最初就被设计成一个非阻塞 I/O 服务器平台，因此一般情况下，你应该期望运行在它上面的所有代码都是非阻塞的。因为 JavaScript 非常小，而且它不强制使用任何 I/O 模型（因为它没有标准的 I/O 类库），因此 Node 建立在一个很纯净的环境里，不会有什么历史遗留问题。

Node 和 JavaScript 如何简化了异步应用程序

Node 的作者 Ryan Dahl，最初使用 C 来开发这个项目，但是发现维护函数调用的上下文太复杂，导致代码复杂度很高。然后他转用 Lua，但是 Lua 已经有个几个阻塞的 I/O 类库，阻塞和非阻塞混在一起可能会让开发人员很迷惑并因此阻碍了很多人构建可伸缩的应用，于是 Lua 也被 Dahl 抛弃了。最后他转向了 JavaScript，JavaScript 中的闭包及第一级对象的函数，这些特性使 JavaScript 非常适合用作事件驱动编程。JavaScript 的魔力是让 Node 如此流行的

一个主要原因。

什么是闭包

闭包可以理解为一个特殊的函数，但是它可以继承并访问它自身被定义的那个作用域里的变量。当你将一个回调函数作为参数传递给另外一个函数时，它稍候会被调用，神奇的是，这个回调函数被稍候调用时，它居然记住了它自身定义所在的那个上下文以及父上下文里的变量，而且还可以正常访问它们。这个强大的特性是 Node 成功的核心。

下面的例子将展示在 Web 浏览器里 JavaScript 闭包是如何工作的。假如，你要监听一个按钮的单机事件，你可以这样做：

```
var clickCount = 0;
document.getElementById('myButton').onclick = function() {
    clickCount += 1;
    alert("clicked " + clickCount + " times.");
};
```

使用 jQuery 时是这样：

```
var clickCount = 0;
$('button#mybutton').click(function() {
    clickedCount ++;
    alert('Clicked ' + clickCount + ' times.');
});
```

JavaScript 里，函数是第一类对象，就是说你可以把函数当作参数来传递给其他函数。上面的两个例子，前者把一个函数赋值给另一个函数，后者把函数作为参数传递给另一个函数，单击事件的处理函数（回调函数）可以访问函数定义所在代码块下的每个变量，在这个例子里，它可以访问在它父闭包内定义的 clickCount 变量。

clickCount 变量处在全局作用域（JavaScript 里最外层的作用域），它保存了用户点击按钮的次数，通常在全局作用域下存储变量是个坏习惯，因为那样很容易跟其他代码冲突，你应该把变量放在使用它们的本地作用域里。大多时候，只用把代码用一个函数包装起来，等于另外创建了闭包，这样就可以很容易避免污染全局环境，就像这样：

```
(function() {
    var clickCount = 0;
    $('button#mybutton').click(function() {
        clickCount ++;
        alert('Clicked ' + clickCount + ' times.');
    });
}());
```

注意：上面代码的第七行，定义了一个函数后立刻调用它，这是 JavaScript 里一个常见的设计模式：通过创建函数来创建一个新的作用域。

闭包如何帮助异步编程

在事件驱动编程模型里，先编写事件发生后将要运行的代码，然后把这些代码放到一个函数里，最后把这个函数当作参数传递给调用者，稍后由调用者函数调用。

在 JavaScript 里，一个函数并不是个孤立的定义，它同时会记住自己被声明的那个作用域的上下文，这种机制让 JavaScript 的函数可以访问函数定义所在那个上下文及父上下文里的所有变量。

当你把一个回调函数当作参数传递给调用者后，这个函数就会在稍后的某个时刻被调用。即使定义回调函数的那个作用域已经结束，在回调函数被调用时，它依然能够访问这个已结束的作用域及其父作用域里的所有变量。像最后那个例子，回调函数在 jQuery 的 click()内部被调用，它却依然能访问 clickCount 变量。

前面展现了闭包的神奇之处，把状态变量传递给一个函数就可以让你不用维护状态就能进行事件驱动编程，JavaScript 的闭包机制会帮你维护它们。

小结

事件驱动编程是一种通过事件触发来决定程序执行流程的编程模型。程序员为他们感兴趣的事件注册回调函数（通常被称作事件处理器），然后系统在事件发生时调用已注册的事件处理器。这种编程模型有很多传统阻塞编程模型所不具备的优势，以前要实现类似的特性，就必须使用多进程/多线程才行。

JavaScript 是种强大的语言，因为它的第一类型对象的函数和闭包特性，让它很适合事件驱动编程，

译者注：

本文对应原文第一部分第二章：Introduciton and Setup：Introducing Node

本系列文章列表和翻译进度，请移步：Node.js 高级编程：用 Javascript 构建可伸缩应用（〇）