CHAPTER

3 阻塞与非阻塞IO

绝大多数对Node.js的讨论都把关注点放在了其处理高并发的能力上。简单来说，相比其他同类解决方案，Node框架给开发者提供了构建高性能网络应用的强大能力，当然，开发者要明白Node内部所做出的权衡，以及用Node构建应用之所以性能好的原因。

27

能力越强，责任就越大

28

Node为JavaScript引入了一个复杂的概念，这在浏览器端从未有过：共享状态的并发。事实上，这种复杂度在像Apache与`mod_php`或者Nginx与FastCGI这样的Web应用开发模型下都从未有过。

通俗讲，Node中，你需要对回调函数如何修改当前内存中的变量（状态）特别小心。除此之外，你还要特别注意对错误的处理是否会潜在地修改这些状态，从而导致了整个进程不可用。

为了更好地掌握这个概念，我们来看如下的函数，该函数在每次请求`/books` URL的时候都会被执行。假设这里的“状态”就是存放图书的数组，该数组用来将图书列表以HTML的形式返回给客户端。

```
var books = [
    'Metamorphosis'
  , 'Crime and punishment'
];

function serveBooks () {
  // 给客户端返回HTML代码
  var html = '<b>' + books.join('</b><br><b>') + '</b>';

  // 恶魔出现了，把状态修改了!
  books = [];

  return html;
}
```

等价的PHP代码为：

```
$books = array(
    'Metamorphosis'
  , 'Crime and punishment'
);

function serveBooks () {
  $html = '<b>' . join($books, '</b><br><b>') . '</b>';
  $books = array();
  return $html;
}
```

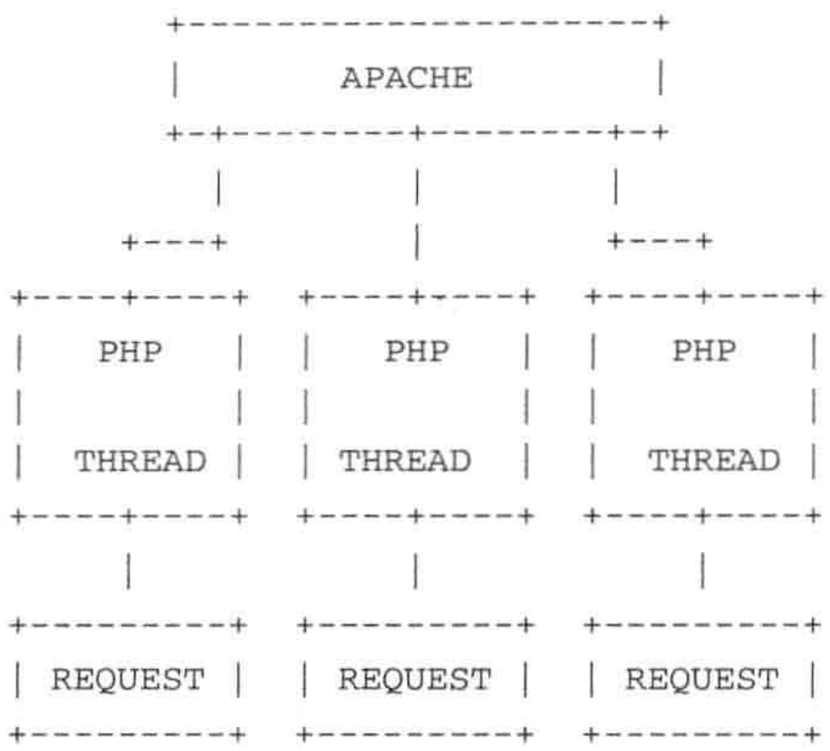
注意，在上述两例serveBooks函数中，都将books数组重置了。

现在假设一个用户分别向Node服务器和PHP服务器各同时发起两次对/books的请求。试着预测下，两者结果会是如何：

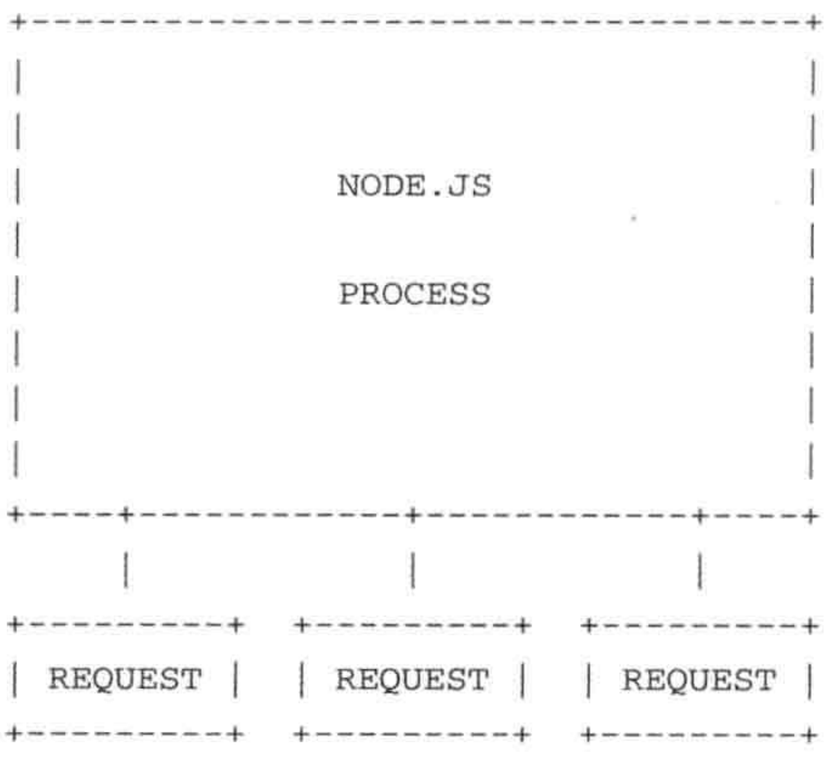
29

- Node会将完整的图书列表返回给第一个请求，而第二个请求则返回一个空的图书列表。
- PHP都能将完整的图书列表返回给两个请求。

两者区别就在于基础架构上。Node采用一个长期运行的进程，相反，Apache会产出多个线程（每个请求一个线程），每次都会刷新状态。在PHP中，当解释器再次执行时，变量$books会被重新赋值，而 Node则不然，serveBooks函数会再次被调用，且作用域中的变量不受影响（此时$books数组仍为空）。



能力越强，责任也就越大。



始终牢记这点对书写出健壮的Node.js程序，避免运行时错误是非常重要的。

另外还有重要的一点是要弄清楚阻塞和非阻塞IO。

阻塞

尝试区分下面PHP代码和Node代码有什么不同：

```
// PHP
print('Hello');

sleep(5);

print('World');
```

30

Node代码示例：

```
// node
console.log('Hello');
```

```
setTimeout(function () {
  console.log('World');
}, 5000);
```

上述两段代码不仅仅是语义上的区别（Node.js使用了回调函数），两者区别集中体现在阻塞和非阻塞的区别上。在第一个例子中，PHP sleep()阻塞了线程的执行。当程序进入睡眠时，就什么事情都不做了。

而Node.js使用了事件轮询，因此这里setTimeout是非阻塞的。

换句话说，如果在setTimeout后再加入console.log语句的话，该语句会被立刻执行：

```
console.log('Hello');
setTimeout(function () {
  console.log('World');
}, 5000);

console.log('Bye');

// 这段脚本会输出:
// Hello
// Bye
// World
```

采用了事件轮询意味着什么呢？从本质上来说，Node会先注册事件，随后不停地询问内核这些事件是否已经分发。当事件分发时，对应的回调函数就会被触发，然后继续执行下去。如果没有事件触发，则继续执行其他代码，直到有新事件时，再去执行对应的回调函数。

相反，在PHP中，sleep一旦执行，执行会被阻塞一段指定的时间，并且在阻塞时间未达到设定时间前，不会有任何操作，也就是说这是同步的。和阻塞相反，setTimeout仅仅只是注册了一个事件，而程序继续执行，所以，这是异步的。

Node并发实现也采用了事件轮询。与timeout所采用的技术一样，所有像http、net这样的原生模块中的IO部分也都采用了事件轮询技术。和timeout机制中Node内部会不停地等待，并当超时完成时，触发一个消息通知一样，Node使用事件轮询，触发一个和文件描述符相关的通知。

31

文件描述符是抽象的句柄，存有对打开的文件、socket、管道等的引用。本质上来说，当Node接收到从浏览器发来的HTTP请求时，底层的TCP连接会分配一个文件描述符。随后，如果客户端向服务器发送数据，Node就会收到该文件描述符上的通知，然后触发JavaScript的回调函数。

单线程的世界

有一点很重要，Node是单线程的。在没有第三方模块的帮助下是无法改变这一事实的。[1]

为了证明这一点，以及展示它和事件轮询之间的关系，我们来看如下例子：

```
var start = Date.now();

setTimeout(function () {
  console.log(Date.now() - start);

  for (var i = 0; i < 1000000000; i++){}
}, 1000);

setTimeout(function () {
  console.log(Date.now() - start);
}, 2000);
```

上述两段setTimeout代码，会打印出timeout设置与最终回调函数执行时，两者的时间差，以毫秒为单位。如图3-1所示是我电脑上打印出的结果。

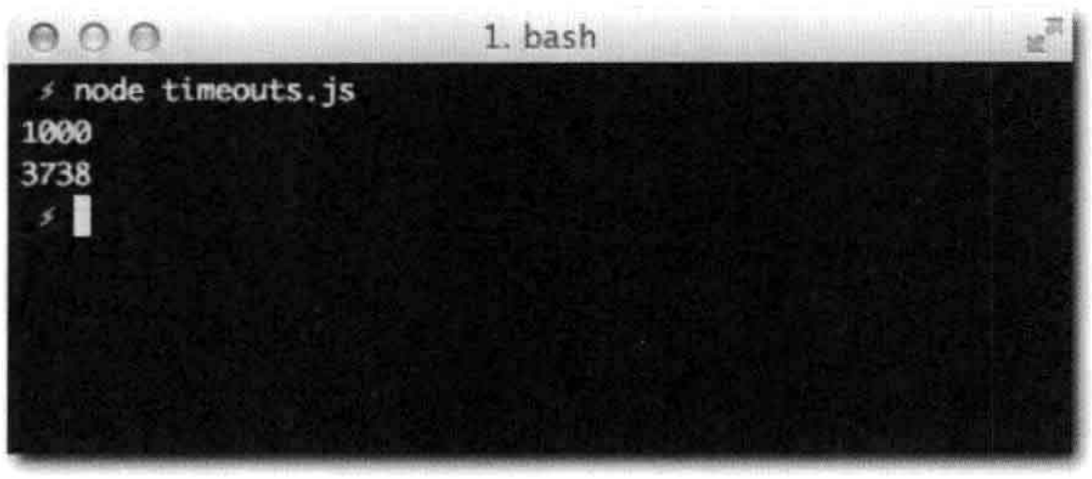


图3-1：程序显示了每个setTimeout执行的时间间隔，其结果和代码中设定的值并不相同

为什么会这样呢？究其原因，是事件轮询被JavaScript代码阻塞了。当第一个事件分发时，会执行JavaScript回调函数。由于回调函数需要执行很长的一段时间（循环次数很多），所以下一个事件轮询执行的时间就远远超过了2秒。因此，JavaScript的timeout并不能严格遵守时钟设置。

当然了，这样的行为方式并不理想。正如我此前介绍的，事件轮询是Node IO的基础核心。既然超时可以延迟，那HTTP请求以及其他形式的IO均可如此。也就意味着，HTTP服务器每秒处理的请求数量减少了，效率也就降低了。 32

正因如此，许多优秀的Node模块都是非阻塞的，执行任务也都采用了异步的方式。

既然执行时只有一个线程，也就是说，当一个函数执行时，同一时间不可能有第二个函数

---

1 译者注：Node早期版本的确不行，但是截止到本书翻译期间，Node 0.8.x和0.10.x都已经内置了child_process模块，允许创建子进程。

也在执行，那Node.js又是如何做到高并发的呢？比如，一台普通的笔记本电脑，用Node书写的简单的服务器就能够处理每秒上千个请求。

为了搞清楚这个问题，你首先要明白调用堆栈的概念。

当v8首次调用一个函数时，会创建一个众所周知的调用堆栈，或者称为执行堆栈。

如果该函数调用又去调用另外一个函数的话，v8就会把它添加到调用堆栈上。考虑如下例子：

```
function a () {
  b();
}
function b(){};
```

针对上述例子，调用堆栈是“a”后面跟着“b”。当“b”执行完，v8就不再执行任何代码了。

回到HTTP服务器的例子：

```
http.createServer(function () {
  a();
});
function a(){
  b();
};
function b(){};
```

在上述例子中，一旦HTTP请求到达服务器，Node就会分发一个通知。最终，回调函数会被执行，并且调用堆栈变为“a”>“b”。

由于Node是运行在单线程环境中，所以，当调用堆栈展开时，Node就无法处理其他的客户端或者HTTP请求了。

你也许在想，那照这样看来，Node的最大并发量不就是1了！是的。 Node并不提供真正的并行操作，因为那样需要引入更多的并行执行线程。

33

关键在于，在调用堆栈执行非常快的情况下，同一时刻你无须处理多个请求。这也是为何说v8搭配非阻塞IO是最好的组合：v8执行JavaScript速度非常快，非阻塞IO确保了单线程执行时，不会因为有数据库访问或者硬盘访问等操作而导致被挂起。

一个真实世界的运用非阻塞IO的例子是云。在绝大多数如亚马逊云（“AWS”）这样的云部署系统中，操作系统都是虚拟出来的，硬件也是由租用者之间互相共享的（所以说你是在“租硬件”）。也就是说，假设硬盘正在为另外的租用者搜索文件，而你也要进行文件搜索，那么延迟就会变长。由于硬盘的IO效率是非常难预测的，所以，读文件时，如果把执行线程阻塞住，那么程序运行起来也会非常不稳定，而且很慢。

在我们的应用中，常见的IO例子就是从数据库中获取数据。假设我们需要为某个请求响应数据库获取的数据。

```
http.createServer(function (req, res) {
  database.getInformation(function (data) {
    res.writeHead(200);
    res.end(data);
  });
});
```

在上述例子中，当请求到达时，调用堆栈中只有数据库调用。由于调用是非阻塞的，当数据库IO完成时，就完全取决于事件轮询何时再初始化新的调用堆栈。不过，在告诉Node“当你获取数据库响应时记得通知我”之后，Node就可以继续处理其他事情了。也就是说，Node可以去处理更多的请求了！

接下来要介绍的，同时也是本书贯穿始终的一个话题，就是错误处理，这个话题和Node架构方式有着很大的关系。

错误处理

首先，很重要的一点，正如本章之前介绍的，Node应用依托在一个拥有大量共享状态的大进程中。

举例来说，在一个HTTP请求中，如果某个回调函数发生了错误，整个进程都会遭殃：

```
var http = require('http');

http.createServer(function () {
  throw new Error( '错误不会被捕获' )
}).listen(3000)
```

因为错误未被捕获，若访问Web服务器，进程就会崩溃，如图3-2所示。

34

图3-2：你能看到调用堆栈从事件轮询（IOWatcher）一路到回调函数

Node之所以这样处理是因为，在发生未被捕获的错误时，进程的状态就不确定了。之后就可能无法正常工作了，并且如果错误始终不处理的话，就会一直抛出意料之外的错误，这样很难调试。

如果添加了uncatchException处理器，就不一样了。这个时候，进程不会退出，并且之后的事情都在你的掌控中。

```
process.on('uncaughtException', function (err) {
  console.error(err);
  process.exit(1); // 手动退出
});
```

在上述例子中，行为方式和分发error事件的API行为方式一致。比如，考虑如下例子，创建一个TCP服务器，并用telnet工具发起连接：

```
var net = require('net');

net.createServer(function (connection) {
  connection.on('error', function (err) {
    // err是一个错误对象
  });
}).listen(400);
```

Node中，许多像http、net这样的原生模块都会分发error事件。如果该事件未处理，就会抛出未捕获的异常。

除了uncaughtException和error事件外，绝大部分Node异步API接收的回调函数，第一个参数都是错误对象或者是null：

35

```
var fs = require('fs');

fs.readFile('/etc/passwd', function (err, data) {
  if (err) return console.error(err);
  console.log(data);
});
```

错误处理中，每一步都很重要，因为它能让你书写更安全的程序，并且不丢失触发错误的上下文信息。

堆栈追踪

在JavaScript中，当错误发生时，在错误信息中可以看到一系列的函数调用，这称为堆栈追踪。看如下例子：

```
function c () {
  b();
};
```

```
function b () {
  a();
};

function a () {
  throw new Error('here');
};

c();
```

运行上述代码就能看到堆栈追踪信息，如图3-3所示。

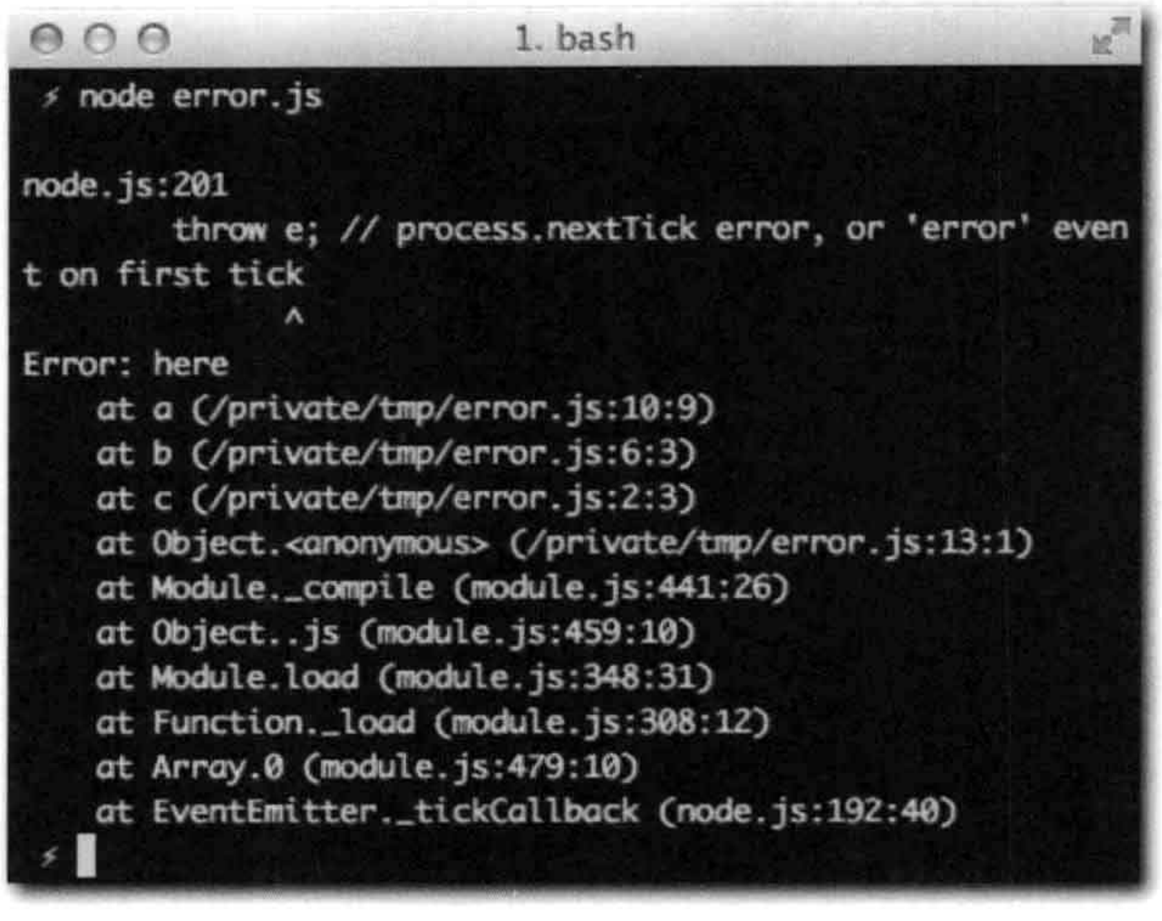


图3-3：针对上述代码，V8显示的堆栈追踪信息

在上图中，你能清晰地看到导致错误发生的函数调用路径。下面，我们来看一下引入事件 36
轮询后会怎么样：

```
function c () {
  b();
};

function b () {
  a();
};

function a () {
  setTimeout(function () {
    throw new Error('here');
  }, 10);
};

c();
```

执行上述代码时（如图3-4所示），堆栈信息中有价值的信息就丢失了。

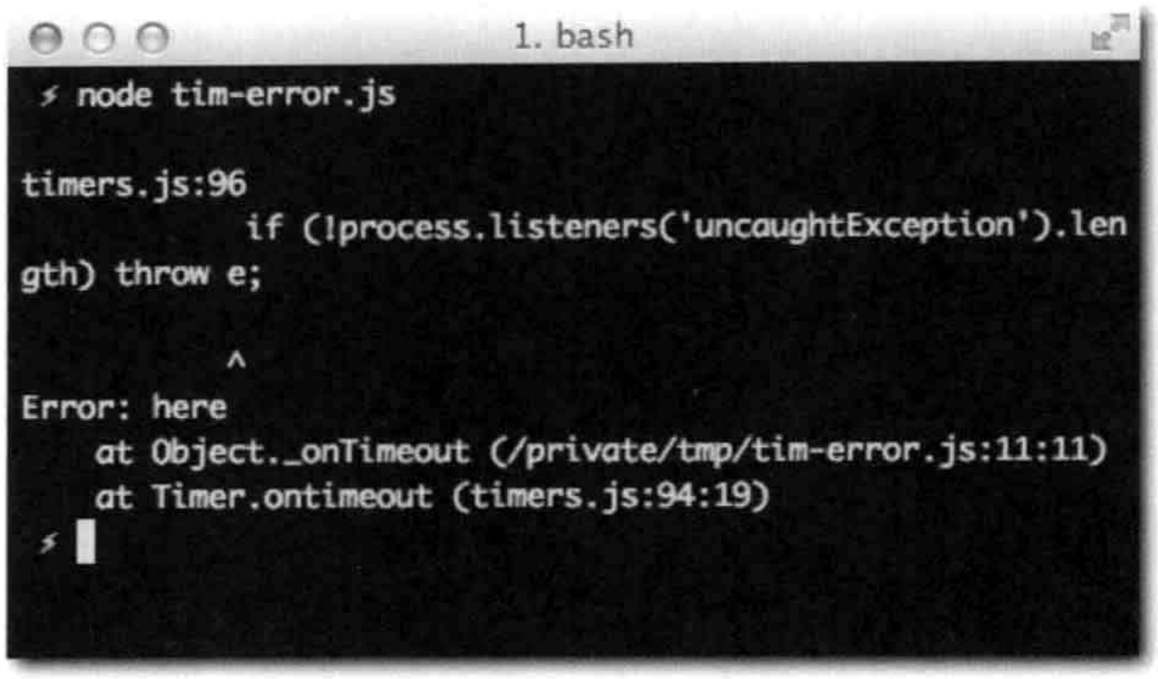


图3-4：堆栈信息显示的是从事件轮询开始的

同理，要捕获一个未来才会执行到的函数所抛出的错误是不可能的。这会直接抛出未捕获的异常，并且catch代码块永远都不会被执行：

```
try {
  setTimeout(function () {
    throw new Error('here');
  }, 10);
} catch (e) { }
```

这就是为什么在Node.js中，每步都要正确进行错误处理的原因了。一旦遗漏，你就会发现发生了错误后很难追踪，因为上下文信息都丢失了。

37 注意，有一点很重要，将来Node会让异步处理器抛出的异常更容易被追踪到。

小结

至此，你已经明白了事件轮询、非阻塞IO以及V8是如何互相配合为开发者提供书写高性能网络应用的能力。

相信你还学到了，Node通过单线程的执行环境，提供了极大的简便，不过，也正因如此，当你书写网络应用时，要尽可能地避免使用同步IO。除此之外，相信你也明白了，该线程中所有的状态都是维护在一个内存空间中的，换句话说，写程序的时候要格外小心。

相信你也清楚地看到了，非阻塞IO和回调引入了新的调试和错误处理的方式，这种方式与写阻塞式IO的程序时是截然不同的。