过了最后一个元素，则两者之间的元素会被创建，并初始化为undefined值：

```
> arr3[20] = "splat";
'splat'
> arr3
[ 'cat', 'rat', 'bat', 'mat', 'fat', , , , , , , , , , , , , , , , , , 'splat' ]
>
```

我们可以尝试使用delete关键字从数组中删除元素，但是结果可能会使我们感到惊讶：

```
> delete arr3[2];

true

> arr3

[ 'cat', 'rat', , 'mat', 'fat', , , , , , , , , , , , , , , , , , 'splat' ]
>
```

可以看到索引2位置对应的值仍然“存在”，只是值被设置为undefined。

要想真正地从数组中删除某一项，可以使用splice函数，它会接收删除项的起始索引和数目作为参数。该函数会返回被删除的数组项，并且原始数组已经被修改，这些项不再存在：

```
> arr3.splice(2, 2);
[  , 'mat' ]
> arr3
[ 'cat', 'rat', 'fat', , , , , , , , , , , , , , , , , , 'splat' ]
> arr3.length
19
```

实用函数

数组中有许多常用的函数。push和pop函数可以让我们向数组的末尾添加或者删除元素：

```
> var nums = [ 1, 1, 2, 3, 5, 8 ];
undefined
> nums.push(13);
7
> nums
[ 1,  1,  2,  3,  5,  8,  13 ]
> nums.pop();
13
> nums
[ 1,  1,  2,  3,  5,  8 ]
>
```

如果想在数组的头部插入或者删除元素，可以分别使用unshift和shift函数：

```
> var nums = [ 1, 2, 3, 5, 8 ];
undefined
> nums.unshift(1);
6
> nums
[ 1,  1,  2,  3,  5,  8 ]
> nums.shift();
1
> nums
[ 1, 2, 3, 5, 8 ]
>
```

与之前提到过的字符串函数split作用相反的是数组函数join，它会返回一个字符串：

```
> var nums = [ 1, 1, 2, 3, 5, 8 ];
undefined
> nums.join(", ");
'1, 1, 2, 3, 5, 8'
>
```

我们可以使用sort函数对数组进行排序，这里使用了内置的排序函数：

```
> var jumble_nums = [ 3, 1, 8, 5, 2, 1];
undefined
> jumble_nums.sort();
[ 1,  1,  2,  3,  5,  8 ]
>
```

而对于那些和预期不符的情况，我们可以自己提供排序函数，并将其作为参数传入sort函数中：

```
> var names = [ 'marc', 'Maria', 'John', 'jerry', 'alfred', 'Moonbeam'];
undefined
> names.sort();
[ 'John',  'Maria',  'Moonbeam',  'alfred',  'jerry',  'marc' ]
> names.sort(function (a, b) {
        var a1 = a.toLowerCase(), b1 = b.toLowerCase();
        if (a1 < b1) return -1;
        if (a1 > b1) return 1;
        return 0;
    });
[ 'alfred',  'jerry',  'John',  'marc',  'Maria',  'Moonbeam' ]
>
```

遍历数组有许多方式，包括前文中的for循环，或者使用forEach函数（V8 JS），如下所示：

```
[ 'marc', 'Maria', 'John', 'jerry', 'alfred', 'Moonbeam'].forEach(function (value) {
    console.log(value);
});
marc
Maria
John
jerry
alfred
Moonbeam
```

2.2 类型比较和转换

前文提到过，大部分情况下，JavaScript数据类型的行为会如你期望的那样，与其他编程语言并无差异。JavaScript有两种相等运算符，相等运算符==（判断两个操作数有没有相同的值）和严格相等运算符===（判断两个操作数有没有相同的值以及是否为相同的数据类型）：

```
> 234 == '234'
true
> 234 === '234'
false
> 234234.235235 == 'cat'
false
> "cat" == "CAT"

false
> "cat".toUpperCase() == "CAT";
true
```

可以看到很多不同的值都等价于false，事实上它们完全不同：

```
> '' == false == null == undefined == 0
true
> null === undefined
false
>
```

在处理任务时，验证函数的参数可以节省很多时间：

```
function fine(param) {
    if (param == null || param == undefined || param == '')
       throw new Error("Invalid Argument");
}
function better(param) {
   if (!param) throw new Error("Invalid Argument");
}
```

如果使用对象构造器来赋值而不是使用原始类型，比较类型时会比较诡异：

```
> var x = 234;
undefined
> var x1 = new Number(234);
undefined
> typeof x1
'object'
> typeof x
'number'
> x1 == x
true
> x1 === x
false
>
```

对象构造器在功能上和原始数据类型是一样的：一样的操作、操作符和函数会产生同样的结果，但是，严格相等运算符和typeof操作符则会产生不同的结果。因此，强烈建议在任何可能的情况下都只使用原始数据类型。

2.3 函数

虽然第一眼看起来不像（名字并没有帮到它），但其实JavaScript是一门函数式编程语言（functional programming language），这意味着函数是完全意义上的对象，可以被操纵、扩展，还可以作为数据进行传递。Node.js充分利用了这种能力，因此，我们会在网络和Web应用中广泛地使用函数。

2.3.1 基本概念

我们能想到的最简单的函数是这样的：

```
function hello(name) {
    console.log("hello " + name);
}
> hello("marc");
hello marc
undefined
>
```

要想在JavaScript函数中声明参数，可以在括号中简单地罗列参数，而且完全不需要在运行时检查这些参数：

```
> hello();
hello undefined
undefined
> hello("marc", "dog", "cat", 48295);
hello marc
undefined
>
```

当函数被调用时，如果传入的参数不够，剩下的变量会被赋予undefined值。而如果传入的参数过多，则多余的参数会被简单地做无用处理。

所有函数在函数体内都会有一个叫做arguments的预定义数组。它拥有函数调用时所有传入的实参，让我们可以对参数列表做额

外的检查。实际上，可以更进一步使用这个数组，从而让函数更加强大和灵活。

假设想初始化一个自己编写的缓存子系统。函数会接收一个大小值来创建缓存，而其他参数则使用默认值，例如缓存地址、过期算法、最大缓存项大小以及存储类型等。可以编写如下代码：

```
function init_cache(size_mb, location, algorithm, item_size, storage_type) {
    ...
}

init_cache(100, null, null, null, null);
```

不过，如果我们让函数变得更加“聪明”从而可以通过多种方式调用，那将会非常酷：

```
function init_cache() {
    var init_data = {
        cache_size: 10,
        location: '/tmp',
        algorithm: 'lru',
        item_size: 1024,
        storage_type: 'btree'
    };

    var a = arguments;

    for (var i = 0; i < a.length; i++) {
        if (typeof a[i] == 'object') {
            init_data = a[i];
            break;
        } else if (typeof a[i] == 'number') {
            init_data.cache_size = a[i];
            break;
        } else {
            throw new Error("bad cache init param");
        }
    }

    // etc
}
```

现在，有许多不同的方式来调用该函数：

```
init_cache();
init_cache(200);
init_cache({ cache_size: 100,
             location: '/exports/dfs/tmp',
             algorithm: 'lruext',
             item_size: 1024,
             storage_type: 'btree'} );
```

JavaScript中的函数甚至不需要名字：

```
var x = function (a, b) {
    return a + b;
}
> x(10, 20);
30
```

这些缺少名字的函数通常叫做匿名函数（anonymous function）。完全匿名的函数有一个缺陷，它会在调试函数时出现：

```
var x = function () {
    throw new Error("whoopsie");
}

> x();
Error: whoopsie
    at x (repl:2:7)
    at repl:1:1
    at REPLServer.self.eval (repl.js:109:21)
    at rli.on.self.bufferedCmd (repl.js:258:20)
```

抛出异常时，匿名函数不会告知出现异常的函数名称。这会导致调试变得更加困难。

简单的解决办法就是为匿名函数命名：

```
var x = function bad_apple() {
    throw new Error("whoopsie");
}

> x();
```

```
Error: whoopsie
    at bad_apple (repl:2:7)
    at repl:1:1
    at REPLServer.self.eval (repl.js:109:21)
    at rli.on.self.bufferedCmd (repl.js:258:20)
```

在复杂的程序中，如果有一个具体的报错地址指针则会节约很多时间。因此，很多人都选择给他们所有的匿名函数命名。

在前文“Array类型”一节中，我们已经看到一个关于匿名函数的例子，该例使用匿名函数作为参数调用排序函数，用来进行大小写不敏感的字符串的比较。接下来我们会在本书中频繁地使用到匿名函数。

2.3.2 函数作用域

每次调用函数，都会创建一个新的变量作用域。父作用域中声明的变量对该函数是可见的，但是，当函数退出后，该函数作用域中声明的变量就会失效。参考以下代码：

```
var pet = 'cat';

function play_with_pets() {
    var pet = 'dog';
    console.log(pet);
}

play_with_pets();
console.log(pet);
```

它会输出如下结果：

```
dog
cat
```

我们可以将作用域和匿名函数结合起来做一些快速或私有的工作。这样，当匿名函数退出后，里面的私有变量也会消失。下面的示例用来计算一个圆锥体的体积：

```
var height = 5;
var radius = 3;
var volume;
// declare and immediately call anon function to create scope
(function () {
    var pir2 = Math.PI * radius * radius;   // temp var
    volume = (pir2 * height) / 3;
})();

console.log(volume);
```

在第3章中，我们将会学习到更多函数相关的常用模式。

2.4 语言结构

JavaScript包含了几乎所有的语言操作符和语句结构，其中包括绝大部分逻辑操作符和算术操作符。

JavaScript同样支持三元运算符：

```
var pet = animal_meows ? "cat" : "dog";
```

尽管大多数的数字都是作为双精度浮点数实现的，JavaScript仍然支持位操作符：&（与）、|（或）、~（非）以及^（异或XOR）操作符都能正常工作：

1.首先将操作数转换成32位整数。

2.进行位操作。

3.最后，将得到的32位整数再转换成JavaScript数字。

另外，除了标准的while、do...while和for循环，JavaScript还支持新的for循环语言扩展，叫做for...in循环（V8 JS）。这种循环用于获取对象的所有属性名：

```
var user = {
    first_name: "marc",
    last_name: "wandschneider",
    age: Infinity,
    occupation: "writer"
};

for (key in user) {
    console.log(key);
}
first_name
last_name
age
occupation
undefined
>
```

2.5 类、原型和继承

JavaScript的面向对象编程与其他语言有很大不同，因为JavaScript没有明确的类（class）关键字或类型。事实上，JavaScript中所有的类都是以函数的形式定义的：

```
function Shape () {
    this.X = 0;
    this.Y = 0;
    this.move = function (x, y) {
        this.X = x;
        this.Y = y;
    }
    this.distance_from_origin = function () {
        return Math.sqrt(this.X*this.X + this.Y*this.Y);
    }
}
var s = new Shape();
s.move(10, 10);
console.log(s.distance_from_origin());
```

上述程序产生下面的输出结果：

```
14.142135623730951
```

只要喜欢，我们可以在任何时候添加任意多的属性和方法到类中：

```
var s = new Shape(15, 35);
s.FillColour = "red";
```

声明类的函数同样也是这个类的构造函数！

然而，这种创建类的方式存在两个问题。首先，效率似乎有点低下，每一个对象都必须自己实现类方法。（每当创建一个新的Shape实例，都要创建move和distance_from_origin函数）其次，我们可能需要继承这个类来创建圆形和方形，并让新的类继承基类（base class）的方法和属性而不必做任何额外的工作。

原型和继承

默认情况下，所有的JavaScript对象都有一个原型（prototype）对象，它是一种继承属性和方法的机制。原型是多年来JavaScript中很多混乱的源头，往往是因为不同的浏览器使用不同的命名和略微不同的实现。因为原型与这一章有关，所以接下来将演示V8（也就是Node）使用的模型，以及其他现代JavaScript实现未来可能的走向。

修改之前创建的Shape类，从而使所有继承该类的对象都获得X和Y属性以及所有定义在这个类上的方法：

```
function Shape () {
}

Shape.prototype.X = 0;
Shape.prototype.Y = 0;

Shape.prototype.move = function (x, y) {
    this.X = x;
    this.Y = y;
}
Shape.prototype.distance_from_origin = function () {
    return Math.sqrt(this.X*this.X + this.Y*this.Y);
}
Shape.prototype.area = function () {
    throw new Error("I don't have a form yet");
}
var s = new Shape();
s.move(10, 10);
console.log(s.distance_from_origin());
```

运行这段脚本，会得到与前面相同的输出结果。事实上，除了可能稍微提升内存的效率之外（如果创建了大量的实例，它们都将共享而不是自己创建move和distance_from_origin方法），两者在功能上并无差异。如果添加一个area方法，则所有的shape实例都会拥有这个方法。而在基类中，这种方式却行不通，它会抛出错误。

更为重要的是，我们可以很容易地对类进行扩展：

```
function Square() {
}

Square.prototype = new Shape();
Square.prototype.__proto__ = Shape.prototype;
Square.prototype.Width = 0;
Square.prototype.area = function () {
    return this.Width * this.Width;
}

var sq = new Square();
sq.move(-5, -5);
sq.Width = 5;
console.log(sq.area());
console.log(sq.distance_from_origin());
```

新Square类的代码用到了一个在V8和其他实现中出现的新JavaScript语言特性：__proto__属性。它能够告诉JavaScript声明的新类的基本原型应该是指定的类型，因此也就可以从指定的类进行扩展。

可以进一步扩展新的类，叫做Rectangle，它会从Square类继承：

```
function Rectangle () {
}

Rectangle.prototype = new Square();
Rectangle.prototype.__proto__ = Square.prototype;
Rectangle.prototype.Height = 0;

Rectangle.prototype.area = function () {
    return this.Width * this.Height;
}

var re = new Rectangle();
re.move(25, 25);
re.Width = 10;
re.Height = 5;
console.log(re.area());
console.log(re.distance_from_origin());
```

为了确认代码能正常运行，可以使用一个之前没有见过的操作符instanceof：

```
console.log(sq instanceof Square);      // true
console.log(sq instanceof Shape);       // true
console.log(sq instanceof Rectangle);   // false
console.log(re instanceof Rectangle);   // true
console.log(re instanceof Square);      // true
console.log(re instanceof Shape);       // true
console.log(sq instanceof Date);        // false
```

2.6 错误和异常

JavaScript中，通常使用Error对象和一条信息来表示一个错误。当遇到错误情况时，可以抛出错误：

```
function uhoh () {
    throw new Error("Something bad happened!");
}


> uhoh();
Error: Something bad happened!
    at uhoh (repl:2:7)
    at repl:1:1
    at REPLServer.self.eval (repl.js:109:21)
    at rli.on.self.bufferedCmd (repl.js:258:20)
```

和其他语言一样，可以通过try/catch语句块来捕捉错误：

```
function uhoh () {
    throw new Error("Something bad happened!");
}

try {
    uhoh();
} catch (e) {
    console.log("I caught an error: " + e.message);
}

console.log("program is still running");
```

该程序的输出结果如下：

```
I caught an error: Something bad happened!
program is still running
```

在下一章中我们会发现，如果使用Node.js中的异步编程模式，这种处理错误的方式会导致一些问题。

2.7 几个重要的Node.js全局对象

Node.js有几个关键的全局对象，这些全局对象总是可见的。

2.7.1 global对象

当在Web浏览器上编写JavaScript代码时，会有一个window对象表现得像"global"变量。任何附加到它上面的变量或成员在应用中的任何地方都是可见的。

Node.js有个类似的东西，叫做global对象。任何附加到该对象上的东西在node应用中的任何地方都是可见的：

```
function printit(var_name) {
    console.log(global[var_name]);
}

global.fish = "swordfish";
global.pet = "cat";

printit("fish");  // prints swordfish
printit("pet");    // prints cat
printit("fruit"); // prints undefined
```

2.7.2 console对象

在经常使用的console.log函数中，我们会看到Node.js有一个全局变量console。不仅如此，它还有一些有趣的函数：

- warn(msg)——与log类似的函数，但打印的是标准错误（stderr）。
- time(label)和timeEnd(label)——第一个函数被调用时会标识一个时间戳，而当第二个函数被调用时，会打印出从time函数被调用后中间经过的时间。

- assert(cond,message)——如果cond等价于false，则抛出一个AssertionFailure异常。

2.7.3 process对象

Node中另外一个关键的全局变量就是process对象，它包含许多信息和方法，如果继续阅读本书，就会接触到这些信息和方法。exit方法是中止Node.js程序的方式之一，而env函数会返回一个对象，它包含了当前用户的环境变量，而cwd则返回应用程序当前的工作目录。

2.8 小结

本章快速回顾了JavaScript语言，阐明了一些迷惑或未知的领域，希望能帮助大家提高一点语言方面的知识。有了这些基本的知识，我们现在可以开始学习如何使用Node.js创建强大快速的应用了。

第3章　异步编程

现在你应该对JavaScript编程有了全新的认识，因此是时候深入了解Node.js的核心概念了：非阻塞IO和异步编程。稍后你会看到这种机制给Node.js带来了巨大的优势和好处，但同时它也带来了许多问题和挑战。

3.1 传统编程方式

在这之前（2008年左右），当我们坐下来准备编写一个程序去加载一个文件时，代码会如下所示（假设正在使用类PHP语言编写示例）：

```
$file = fopen('info.txt', 'r');
// wait until file is open

$contents = fread($file, 100000);
// wait until contents are read

// do something with those contents
```

如果我们分析这段代码的执行情况，会发现大部分时间它什么都没有做。事实上，这段代码所花的绝大部分时间是在等待电脑的文件系统执行，最后才会返回请求的文件内容。进一步分析，对于绝大部分基于IO的应用——那些需要频繁连接数据库、和外部服务器通信或者读写文件的应用——脚本将会花费大量时间用来等待处理结果（见图3.1）。

服务器同时处理多个请求的方法就是并行这些脚本。现代电脑操作系统能够很好地支持多任务处理，所以可以很容易地在进程阻塞的时候切换任务，以便让其他进程访问CPU资源。而有些系统环境做得更好，使用多线程来替代多进程。

现在的问题是，每一个进程或者线程都会消耗大量的系统资源。我曾经看到在一些使用Apache和PHP编写的大型系统中，每一个进程占有高达10-15M的内存资源——它们从不关心操作系统不断切换上下文过程中所消耗的资源和时间。这种应用如果同时处理100个任务，甚至会消耗超过1GB的内存资源。使用多线程解决方案或者轻量级HTTP服务器可以获得更好的效果，但是仍然会遇到电脑消耗大量的时间用在等待阻塞进程返回结果的情况，甚至会面临没有足够的容量来处理更多请求的风险。

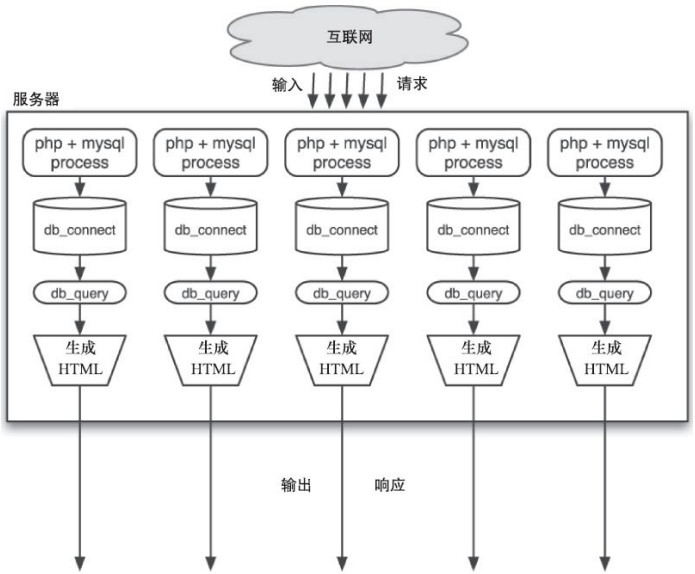

图3.1　传统阻塞IO Web服务器

如果有方法可以最大化利用CPU的计算能力和可用内存以减少资源浪费，那将再好不过了。而这，正是Node.js的精髓所在。

3.2 Node.js的编程方式

要理解Node.js如何将前文中展示的方法变成非阻塞异步模型，首先要看下JavaScript中的setTimeout这个函数。该函数的参数包含一个函数调用和等待时间，参数函数会在等待时间到达后执行：

```
// blah
setTimeout(function () {
    console.log("I've done my work!");
}, 2000);

console.log("I'm waiting for all my work to finish.");
```

如果运行上面这段代码，可以看到如下输出结果：

```
I'm waiting for all my work to finish.
I've done my work!
```

希望没有吓着你：这段程序设置了2000ms（2s）的等待时间和需要调用的函数。程序继续执行，打印出"I'm waiting..."的文字。2秒钟以后，可以看到"I've done..."这样的信息，然后程序退出。

现在看一下，每次调用函数时，都需要等待外部资源（数据库服务器、网络请求或者文件系统的读写操作），这些情况非常类似。因此，我们需要选择调用fopen(path,mode,function callback(file_handle){...})函数来替代调用fopen(path,mode)并等待响应。

现在使用新的异步函数来重写前面的同步脚本。可以在命令行中使用node输入并运行该程序。注意，要确保创建了info.txt来供程序读取。

```
var fs = require('fs');                    // this is new, see explanation

var file;
var buf = new Buffer(100000);
fs.open(
    'info.txt', 'r',
    function (handle) {
        file = handle;
    }
);

fs.read(                                   // this will generate an error.
    file, buf, 0, 100000, null,
    function () {
        console.log(buf.toString());
        file.close(file, function () { /* don't care */ });
    }
);
```

也许你从没见过这段代码的第一行：require函数可以用来在Node.js程序中引入外部功能。Node拥有一套非常棒的模块（module），当想使用其中的某个功能时，可以单独地引入到代码中。后面我们将会进一步学习使用模块，还会在第5章中学习如何使用它们，并编写属于自己的模块。

如果就这么运行上面这段程序，它会抛出错误并退出。到底怎么回事？这是因为fs.open函数是异步执行的，它会在文件打开之前立刻返回，而handle值会返回给回调函数。在文件打开之前，file变量不会被赋值，它的值会在fs.open函数的回调函数中被接收。因此，如果在它后面立刻调用fs.read函数，file的值仍然是undefined。

修复这个程序很简单：

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

fs.open(

    'info.txt', 'r',
    function (err, handle) {     // we'll see more about the err param in a bit
        var buf = new Buffer(100000);
        fs.read(
            handle, buf, 0, 100000, null,
            function (err, length) {
                console.log(buf.toString('utf8', 0, length));
                fs.close(handle, function () { /* don't care */ });
            }
        );
    }
);
```

思考这些异步函数如何工作的关键方法包括以下几个方面：

- 检查和验证参数

- 通知Node.js核心去排队调用相应的函数（如前面示例中，操作系统的open或者read函数），并在返回结果的时候通知（调用）回调函数

- 返回给调用者

也许你会问：如果open函数立刻返回，为什么node进程没有在函数返回后立刻退出？这是因为Node使用了事件队列（event queue）。如果有挂起的事件等待响应，它就不会退出，除非代码执行结束并且在队列上没有任何其他事件。如果你正在等待某个函数调用的响应（如open或read函数），Node就会一直等待。见图3.2，从概念上理解运行机制。

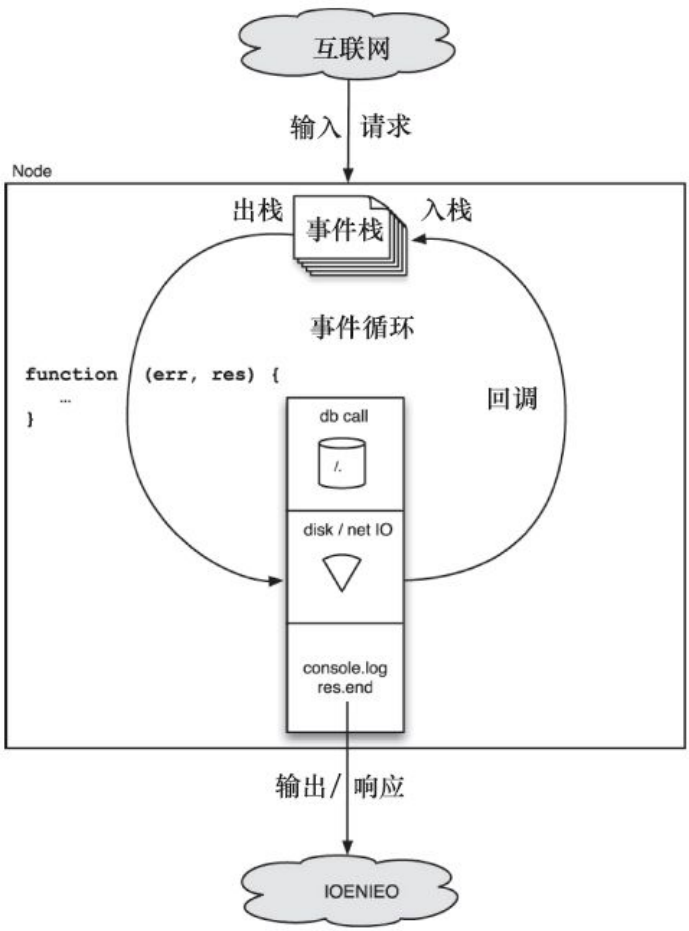

图3.2　只要代码还在执行或等待响应，Node就会一直运行