```
    console.log(i);
  });
}
```

这样就可以让每次点击都显示正确的索引了。这里，事件处理程序执行时就会引用 for 循环块级作用域中的索引值。这是因为在 ECMAScript 6 中，如果对 for 循环使用块级作用域变量关键字，在这里就是 let，那么循环就会为每个循环创建独立的变量，从而让每个单击处理程序都能引用特定的索引。

但要注意，如果把变量声明拿到 for 循环外部，那就不行了。下面这种写法会碰到跟在循环中使用 var i = 0 同样的问题：

```
let divs = document.querySelectorAll('div');

// 达不到目的!
let i;
for (i = 0; i < divs.length; ++i) {
  divs[i].addEventListener('click', function() {
    console.log(i);
  });
}
```

10.16　私有变量

严格来讲，JavaScript 没有私有成员的概念，所有对象属性都公有的。不过，倒是有**私有变量**的概念。任何定义在函数或块中的变量，都可以认为是私有的，因为在这个函数或块的外部无法访问其中的变量。私有变量包括函数参数、局部变量，以及函数内部定义的其他函数。来看下面的例子：

```
function add(num1, num2) {
  let sum = num1 + num2;
  return sum;
}
```

在这个函数中，函数 add()有 3 个私有变量：num1、num2 和 sum。这几个变量只能在函数内部使用，不能在函数外部访问。如果这个函数中创建了一个闭包，则这个闭包能通过其作用域链访问其外部的这 3 个变量。基于这一点，就可以创建出能够访问私有变量的公有方法。

特权方法（privileged method）是能够访问函数私有变量（及私有函数）的公有方法。在对象上有两种方式创建特权方法。第一种是在构造函数中实现，比如：

```
function MyObject() {
  // 私有变量和私有函数
  let privateVariable = 10;

  function privateFunction() {
    return false;
  }

  // 特权方法
  this.publicMethod = function() {
    privateVariable++;
    return privateFunction();
  };
}
```

这个模式是把所有私有变量和私有函数都定义在构造函数中。然后，再创建一个能够访问这些私有

成员的特权方法。这样做之所以可行，是因为定义在构造函数中的特权方法其实是一个闭包，它具有访问构造函数中定义的所有变量和函数的能力。在这个例子中，变量 privateVariable 和函数 privateFunction()只能通过 publicMethod()方法来访问。在创建 MyObject 的实例后，没有办法直接访问 privateVariable 和 privateFunction()，唯一的办法是使用 publicMethod()。

如下面的例子所示，可以定义私有变量和特权方法，以隐藏不能被直接修改的数据：

```
function Person(name) {
  this.getName = function() {
    return name;
  };

  this.setName = function (value) {
    name = value;
  };
}

let person = new Person('Nicholas');
console.log(person.getName());  // 'Nicholas'
person.setName('Greg');
console.log(person.getName());  // 'Greg'
```

这段代码中的构造函数定义了两个特权方法：getName()和 setName()。每个方法都可以构造函数外部调用，并通过它们来读写私有的 name 变量。在 Person 构造函数外部，没有别的办法访问 name。因为两个方法都定义在构造函数内部，所以它们都是能够通过作用域链访问 name 的闭包。私有变量 name 对每个 Person 实例而言都是独一无二的，因为每次调用构造函数都会重新创建一套变量和方法。不过这样也有个问题：必须通过构造函数来实现这种隔离。正如第 8 章所讨论的，构造函数模式的缺点是每个实例都会重新创建一遍新方法。使用静态私有变量实现特权方法可以避免这个问题。

10.16.1 静态私有变量

特权方法也可以通过使用私有作用域定义私有变量和函数来实现。这个模式如下所示：

```
(function() {
  // 私有变量和私有函数
  let privateVariable = 10;

  function privateFunction() {
    return false;
  }

  // 构造函数
  MyObject = function() {};

  // 公有和特权方法
  MyObject.prototype.publicMethod = function() {
    privateVariable++;
    return privateFunction();
  };
})();
```

在这个模式中，匿名函数表达式创建了一个包含构造函数及其方法的私有作用域。首先定义的是私有变量和私有函数，然后又定义了构造函数和公有方法。公有方法定义在构造函数的原型上，与典型的原型模式一样。注意，这个模式定义的构造函数没有使用函数声明，使用的是函数表达式。函数声明会

创建内部函数，在这里并不是必需的。基于同样的原因（但操作相反），这里声明 MyObject 并没有使用任何关键字。因为不使用关键字声明的变量会创建在全局作用域中，所以 MyObject 变成了全局变量，可以在这个私有作用域外部被访问。注意在严格模式下给未声明的变量赋值会导致错误。

这个模式与前一个模式的主要区别就是，私有变量和私有函数是由实例共享的。因为特权方法定义在原型上，所以同样是由实例共享的。特权方法作为一个闭包，始终引用着包含它的作用域。来看下面的例子：

```
(function() {
  let name = '';

  Person = function(value) {
    name = value;
  };

  Person.prototype.getName = function() {
    return name;
  };

  Person.prototype.setName = function(value) {
    name = value;
  };
})();

let person1 = new Person('Nicholas');
console.log(person1.getName());  // 'Nicholas'
person1.setName('Matt');
console.log(person1.getName());  // 'Matt'

let person2 = new Person('Michael');
console.log(person1.getName());  // 'Michael'
console.log(person2.getName());  // 'Michael'
```

这里的 Person 构造函数可以访问私有变量 name，跟 getName()和 setName()方法一样。使用这种模式，name 变成了静态变量，可供所有实例使用。这意味着在任何实例上调用 setName()修改这个变量都会影响其他实例。调用 setName()或创建新的 Person 实例都要把 name 变量设置为一个新值。而所有实例都会返回相同的值。

像这样创建静态私有变量可以利用原型更好地重用代码，只是每个实例没有了自己的私有变量。最终，到底是把私有变量放在实例中，还是作为静态私有变量，都需要根据自己的需求来确定。

> **注意**　使用闭包和私有变量会导致作用域链变长，作用域链越长，则查找变量所需的时间也越多。

10.16.2　模块模式

前面的模式通过自定义类型创建了私有变量和特权方法。而下面要讨论的 Douglas Crockford 所说的模块模式，则在一个单例对象上实现了相同的隔离和封装。单例对象（singleton）就是只有一个实例的对象。按照惯例，JavaScript 是通过对象字面量来创建单例对象的，如下面的例子所示：

```
let singleton = {
  name: value,
```

```
  method() {
    // 方法的代码
  }
};
```

模块模式是在单例对象基础上加以扩展，使其通过作用域链来关联私有变量和特权方法。模块模式的样板代码如下：

```
let singleton = function() {
  // 私有变量和私有函数
  let privateVariable = 10;

  function privateFunction() {
    return false;
  }

  // 特权/公有方法和属性
  return {
    publicProperty: true,

    publicMethod() {
      privateVariable++;
      return privateFunction();
    }
  };
}();
```

模块模式使用了匿名函数返回一个对象。在匿名函数内部，首先定义私有变量和私有函数。之后，创建一个要通过匿名函数返回的对象字面量。这个对象字面量中只包含可以公开访问的属性和方法。因为这个对象定义在匿名函数内部，所以它的所有公有方法都可以访问同一个作用域的私有变量和私有函数。本质上，对象字面量定义了单例对象的公共接口。如果单例对象需要进行某种初始化，并且需要访问私有变量时，那就可以采用这个模式：

```
let application = function() {
  // 私有变量和私有函数
  let components = new Array();

  // 初始化
  components.push(new BaseComponent());

  // 公共接口
  return {
    getComponentCount() {
      return components.length;
    },
    registerComponent(component) {
      if (typeof component == 'object') {
        components.push(component);
      }
    }
  };
}();
```

在 Web 开发中，经常需要使用单例对象管理应用程序级的信息。上面这个简单的例子创建了一个 `application` 对象用于管理组件。在创建这个对象之后，内部就会创建一个私有的数组 `components`，然后将一个 `BaseComponent` 组件的新实例添加到数组中。（`BaseComponent` 组件的代码并不重要，在

这里用它只是为了说明模块模式的用法。）对象字面量中定义的 getComponentCount()和 register-Component()方法都是可以访问 components 私有数组的特权方法。前一个方法返回注册组件的数量，后一个方法负责注册新组件。

在模块模式中，单例对象作为一个模块，经过初始化可以包含某些私有的数据，而这些数据又可以通过其暴露的公共方法来访问。以这种方式创建的每个单例对象都是 Object 的实例，因为最终单例都由一个对象字面量来表示。不过这无关紧要，因为单例对象通常是可以全局访问的，而不是作为参数传给函数的，所以可以避免使用 instanceof 操作符确定参数是不是对象类型的需求。

10.16.3　模块增强模式

另一个利用模块模式的做法是在返回对象之前先对其进行增强。这适合单例对象需要是某个特定类型的实例，但又必须给它添加额外属性或方法的场景。来看下面的例子：

```
let singleton = function() {
  // 私有变量和私有函数
  let privateVariable = 10;

  function privateFunction() {
    return false;
  }

  // 创建对象
  let object = new CustomType();

  // 添加特权/公有属性和方法
  object.publicProperty = true;

  object.publicMethod = function() {
    privateVariable++;
    return privateFunction();
  };

  // 返回对象
  return object;
}();
```

如果前一节的 application 对象必须是 BaseComponent 的实例，那么就可以使用下面的代码来创建它：

```
let application = function() {
  // 私有变量和私有函数
  let components = new Array();

  // 初始化
  components.push(new BaseComponent());

  // 创建局部变量保存实例
  let app = new BaseComponent();

  // 公共接口
  app.getComponentCount = function() {
    return components.length;
  };
```

```
  app.registerComponent = function(component) {
    if (typeof component == "object") {
      components.push(component);
    }
  };

  // 返回实例
  return app;
}();
```

在这个重写的 application 单例对象的例子中，首先定义了私有变量和私有函数，跟之前例子中一样。主要区别在于这里创建了一个名为 app 的变量，其中保存了 BaseComponent 组件的实例。这是最终要变成 application 的那个对象的局部版本。在给这个局部变量 app 添加了能够访问私有变量的公共方法之后，匿名函数返回了这个对象。然后，这个对象被赋值给 application。

10.17 小结

函数是 JavaScript 编程中最有用也最通用的工具。ECMAScript 6 新增了更加强大的语法特性，从而让开发者可以更有效地使用函数。

- 函数表达式与函数声明是不一样的。函数声明要求写出函数名称，而函数表达式并不需要。没有名称的函数表达式也被称为匿名函数。
- ES6 新增了类似于函数表达式的箭头函数语法，但两者也有一些重要区别。
- JavaScript 中函数定义与调用时的参数极其灵活。arguments 对象，以及 ES6 新增的扩展操作符，可以实现函数定义和调用的完全动态化。
- 函数内部也暴露了很多对象和引用，涵盖了函数被谁调用、使用什么调用，以及调用时传入了什么参数等信息。
- JavaScript 引擎可以优化符合尾调用条件的函数，以节省栈空间。
- 闭包的作用域链中包含自己的一个变量对象，然后是包含函数的变量对象，直到全局上下文的变量对象。
- 通常，函数作用域及其中的所有变量在函数执行完毕后都会被销毁。
- 闭包在被函数返回之后，其作用域会一直保存在内存中，直到闭包被销毁。
- 函数可以在创建之后立即调用，执行其中代码之后却不留下对函数的引用。
- 立即调用的函数表达式如果不在包含作用域中将返回值赋给一个变量，则其包含的所有变量都会被销毁。
- 虽然 JavaScript 没有私有对象属性的概念，但可以使用闭包实现公共方法，访问位于包含作用域中定义的变量。
- 可以访问私有变量的公共方法叫作特权方法。
- 特权方法可以使用构造函数或原型模式通过自定义类型中实现，也可以使用模块模式或模块增强模式在单例对象上实现。

第11章 期约与异步函数

本章内容

- ❑ 异步编程
- ❑ 期约
- ❑ 异步函数

ECMAScript 6 及之后的几个版本逐步加大了对异步编程机制的支持，提供了令人眼前一亮的新特性。ECMAScript 6 新增了正式的 `Promise`（期约）引用类型，支持优雅地定义和组织异步逻辑。接下来几个版本增加了使用 `async` 和 `await` 关键字定义异步函数的机制。

> **注意** 本章示例将大量使用异步日志输出的方式 `setTimeout(console.log, 0, ...params)`，旨在演示执行顺序及其他异步行为。异步输出的内容看起来虽然像是同步输出的，但实际上是异步打印的。这样可以让期约等返回的值达到其最终状态。
>
> 此外，浏览器控制台的输出经常能打印出 JavaScript 运行中无法获取的对象信息（比如期约的状态）。这个特性在示例中广泛使用，以便辅助读者理解相关概念。

11.1 异步编程

同步行为和异步行为的对立统一是计算机科学的一个基本概念。特别是在 JavaScript 这种单线程事件循环模型中，同步操作与异步操作更是代码所要依赖的核心机制。异步行为是为了优化因计算量大而时间长的操作。如果在等待其他操作完成的同时，即使运行其他指令，系统也能保持稳定，那么这样做就是务实的。

重要的是，异步操作并不一定计算量大或要等很长时间。只要你不想为等待某个异步操作而阻塞线程执行，那么任何时候都可以使用。

11.1.1 同步与异步

同步行为对应内存中顺序执行的处理器指令。每条指令都会严格按照它们出现的顺序来执行，而每条指令执行后也能立即获得存储在系统本地（如寄存器或系统内存）的信息。这样的执行流程容易分析程序在执行到代码任意位置时的状态（比如变量的值）。

同步操作的例子可以是执行一次简单的数学计算：

```
let x = 3;
x = x + 4;
```

在程序执行的每一步，都可以推断出程序的状态。这是因为后面的指令总是在前面的指令完成后才会执行。等到最后一条指定执行完毕，存储在 x 的值就立即可以使用。

这两行 JavaScript 代码对应的低级指令（从 JavaScript 到 x86）并不难想象。首先，操作系统会在栈内存上分配一个存储浮点数值的空间，然后针对这个值做一次数学计算，再把计算结果写回之前分配的内存中。所有这些指令都是在单个线程中按顺序执行的。在低级指令的层面，有充足的工具可以确定系统状态。

相对地，**异步行为**类似于系统中断，即当前进程外部的实体可以触发代码执行。异步操作经常是必要的，因为强制进程等待一个长时间的操作通常是不可行的（同步操作则必须要等）。如果代码要访问一些高延迟的资源，比如向远程服务器发送请求并等待响应，那么就会出现长时间的等待。

异步操作的例子可以是在定时回调中执行一次简单的数学计算：

```
let x = 3;
setTimeout(() => x = x + 4, 1000);
```

这段程序最终与同步代码执行的任务一样，都是把两个数加在一起，但这一次执行线程不知道 x 值何时会改变，因为这取决于回调何时从消息队列出列并执行。

异步代码不容易推断。虽然这个例子对应的低级代码最终跟前面的例子没什么区别，但第二个指令块（加操作及赋值操作）是由系统计时器触发的，这会生成一个入队执行的中断。到底什么时候会触发这个中断，这对 JavaScript 运行时来说是一个黑盒，因此实际上无法预知（尽管可以保证这发生在当前线程的同步代码执行**之后**，否则回调都没有机会出列被执行）。无论如何，在排定回调以后基本没办法知道系统状态何时变化。

为了让后续代码能够使用 x，异步执行的函数需要在更新 x 的值以后通知其他代码。如果程序不需要这个值，那么就只管继续执行，不必等待这个结果了。

设计一个能够知道 x 什么时候可以读取的系统是非常难的。JavaScript 在实现这样一个系统的过程中也经历了几次迭代。

11.1.2 以往的异步编程模式

异步行为是 JavaScript 的基础，但以前的实现不理想。在早期的 JavaScript 中，只支持定义回调函数来表明异步操作完成。串联多个异步操作是一个常见的问题，通常需要深度嵌套的回调函数（俗称“回调地狱”）来解决。

假设有以下异步函数，使用了 setTimeout 在一秒钟之后执行某些操作：

```
function double(value) {
  setTimeout(() => setTimeout(console.log, 0, value * 2), 1000);
}

double(3);
// 6（大约 1000 毫秒之后）
```

这里的代码没什么神秘的，但关键是理解为什么说它是一个异步函数。setTimeout 可以定义一个在指定时间之后会被调度执行的回调函数。对这个例子而言，1000 毫秒之后，JavaScript 运行时会把回调函数推到自己的消息队列上去等待执行。推到队列之后，回调什么时候出列被执行对 JavaScript 代码就完全不可见了。还有一点，double()函数在 setTimeout 成功调度异步操作之后会立即退出。

1. 异步返回值

假设 setTimeout 操作会返回一个有用的值。有什么好办法把这个值传给需要它的地方？广泛接受的一个策略是给异步操作提供一个回调，这个回调中包含要使用异步返回值的代码（作为回调的参数）。

```
function double(value, callback) {
  setTimeout(() => callback(value * 2), 1000);
}

double(3, (x) => console.log(`I was given: ${x}`));
// I was given: 6（大约 1000 毫秒之后）
```

这里的 setTimeout 调用告诉 JavaScript 运行时在 1000 毫秒之后把一个函数推到消息队列上。这个函数会由运行时负责异步调度执行。而位于函数闭包中的回调及其参数在异步执行时仍然是可用的。

2. 失败处理

异步操作的失败处理在回调模型中也要考虑，因此自然就出现了成功回调和失败回调：

```
function double(value, success, failure) {
  setTimeout(() => {
    try {
      if (typeof value !== 'number') {
        throw 'Must provide number as first argument';
      }
      success(2 * value);
    } catch (e) {
      failure(e);
    }
  }, 1000);
}

const successCallback = (x) => console.log(`Success: ${x}`);
const failureCallback = (e) => console.log(`Failure: ${e}`);

double(3, successCallback, failureCallback);
double('b', successCallback, failureCallback);

// Success: 6（大约 1000 毫秒之后）
// Failure: Must provide number as first argument（大约 1000 毫秒之后）
```

这种模式已经不可取了，因为必须在初始化异步操作时定义回调。异步函数的返回值只在短时间内存在，只有预备好将这个短时间内存在的值作为参数的回调才能接收到它。

3. 嵌套异步回调

如果异步返值又依赖另一个异步返回值，那么回调的情况还会进一步变复杂。在实际的代码中，这就要求嵌套回调：

```
function double(value, success, failure) {
  setTimeout(() => {
    try {
      if (typeof value !== 'number') {
        throw 'Must provide number as first argument';
      }
      success(2 * value);
    } catch (e) {
      failure(e);
    }
  }, 1000);
```

```
}

const successCallback = (x) => {
  double(x, (y) => console.log(`Success: ${y}`));
};
const failureCallback = (e) => console.log(`Failure: ${e}`);

double(3, successCallback, failureCallback);

// Success: 12（大约 1000 毫秒之后）
```

显然，随着代码越来越复杂，回调策略是不具有扩展性的。“回调地狱”这个称呼可谓名至实归。嵌套回调的代码维护起来就是噩梦。

11.2 期约

期约是对尚不存在结果的一个替身。期约（promise）这个名字最早是由 Daniel Friedman 和 David Wise 在他们于 1976 年发表的论文“The Impact of Applicative Programming on Multiprocessing”中提出来的。但直到十几年以后，Barbara Liskov 和 Liuba Shrira 在 1988 年发表了论文“Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems”，这个概念才真正确立下来。同一时期的计算机科学家还使用了“终局”（eventual）、“期许”（future）、“延迟”（delay）和“迟付”（deferred）等术语指代同样的概念。所有这些概念描述的都是一种异步程序执行的机制。

11.2.1 Promises/A+规范

早期的期约机制在 jQuery 和 Dojo 中是以 Deferred API 的形式出现的。到了 2010 年，CommonJS 项目实现的 Promises/A 规范日益流行起来。Q 和 Bluebird 等第三方 JavaScript 期约库也越来越得到社区认可，虽然这些库的实现多少都有些不同。为弥合现有实现之间的差异，2012 年 Promises/A+组织分叉（fork）了 CommonJS 的 Promises/A 建议，并以相同的名字制定了 Promises/A+规范。这个规范最终成为了 ECMAScript 6 规范实现的范本。

ECMAScript 6 增加了对 Promises/A+规范的完善支持，即 `Promise` 类型。一经推出，`Promise` 就大受欢迎，成为了主导性的异步编程机制。所有现代浏览器都支持 ES6 期约，很多其他浏览器 API（如 `fetch()`和 Battery Status API）也以期约为基础。

11.2.2 期约基础

ECMAScript 6 新增的引用类型 `Promise`，可以通过 `new` 操作符来实例化。创建新期约时需要传入执行器（executor）函数作为参数（后面马上会介绍），下面的例子使用了一个空函数对象来应付一下解释器：

```
let p = new Promise(() => {});
setTimeout(console.log, 0, p);  // Promise <pending>
```

之所以说是应付解释器，是因为如果不提供执行器函数，就会抛出 `SyntaxError`。

1. 期约状态机

在把一个期约实例传给 `console.log()`时，控制台输出（可能因浏览器不同而略有差异）表明该实例处于**待定**（pending）状态。如前所述，期约是一个有状态的对象，可能处于如下 3 种状态之一：