第 5 章

函　　数

之前章节所编写的代码在执行一次之后就结束了，如果想在其他地方再次使用同样的代码，则需要再复制粘贴一次，若要重复使用多次，则代码量会越来越大。为了复用一段代码，可以使用函数。

函数接收可选的输入参数，并可以返回执行结果，相当于数学中的函数，它所执行的操作是固定的，但是根据不同的参数值会产生不同的结果。函数中的代码只需定义一次，之后在任何需要使用的地方调用它，利用函数可以把大段代码分解成相对独立的单元，一个函数专注于做一件事情，然后通过组合不同的函数实现复杂的功能逻辑，这样就能让代码更清晰易读了。

除了代码复用之外，使用函数还可以创建独立的作用域，防止因变量名重复而被覆盖，从而引发作用域污染的问题。

函数在 JavaScript 中是“头等公民”，有着举足轻重的地位，在任何需要表达式的地方都可以使用函数，例如把函数作为表达式赋值给变量、传递给另一个函数中的参数、作为函数返回值等。因为这个原因，编写 JavaScript 的灵活性大幅提高，既可以使用过程式，也可以使用函数式的编程范式。

本章将介绍函数的声明、调用、箭头函数、作用域及函数式编程的一些基本概念，如闭包、高阶函数、柯里化等，另外借由作用域，也会介绍一下 var/let 的区别，以及函数和变量的提升问题。

5.1　声明函数

现在先来看一下如何声明（也可以叫作定义，后边会互换使用这两个概念）和使用函数。在使用函数前需要先进行声明，代码如下：

```
function sayHello() {
  console.log("hello");
}
```

首先函数的声明使用 function 关键字开头，后边是函数的名字，采用与变量相同的标识

符命名规则，即只能以 $ 、_、字母开头，单词之间推荐使用驼峰命名法，即第 1 个单词首字母小写，后续每个单词首字母大写，例如示例中的 sayHello。

函数名的后面是参数列表，使用小括号包裹起来，示例中声明的函数没有参数，所以直接使用了空的小括号。

参数列表后边是函数要执行的语句块(也叫作函数体)，使用大括号包裹，示例中只有一行代码：使用 console.log()打印了 hello 字符串。

下边再来看一个声明带参数的函数的例子，代码如下：

```
function sayHello(name) {
  console.log("hello " + name);
}
```

示例函数的参数列表里定义了一个 name 参数，用于接收名字信息，因为 JavaScript 是动态类型语言，所以参数不用也不能指定类型。在函数体中，可以像使用变量一样使用参数，这里把 name 拼接到了 hello 的后边，至于 name 具体所表示的值，只有在调用的时候才可以确定。

在上边的例子中，函数在执行完 console.log 之后就结束了，不过函数还可以返回一些值，作为执行结果供其他代码使用。例如下方代码展示了一个加法函数，用于计算两个数的和，并把和返回，代码如下：

```
function sum(a, b) {
  return a + b;
}
```

在函数中使用 return 关键字来返回相应的值，也就是 a+b 的结果。

如果函数没有返回语句，程序也会自动在代码的最后写上 return;语句，这样 return 的后边没有值，而是直接使用了分号，它最后返回的结果就是 undefined。

如果要打印一下函数名，例如 console.log(sum)，则会直接返回函数代码本身，这是因为函数的定义也是表达式，表达式的值就是函数本身。同时函数也有 name 和 length 属性，使用 sum.name 和 sum.length 分别会返回函数的名字和函数参数的数量，代码如下：

```
sum.name;                    //"sum"
sum.length;                  //2
```

5.2 调用函数

在声明函数之后，函数里的代码并不会自动执行，而是需要在代码中去调用它们，然后才会实际运行函数中的代码。

调用一个函数需要使用函数名加上小括号，如果小括号里没有参数，则把小括号留空（不能省略）；如果有参数，则可以给参数按顺序赋上实际的值。例如 5.1 节的示例 sayHello(name)函数可以使用下方的代码进行调用，代码如下：

```
sayHello("John");                          //"hello John"
```

这里把 name 参数赋值为"John"，相当于让 name="John"，这样在 console.log("hello"+name)语句中，name 就被替换成了"John"，结果打印出"hello John"。

如果函数有返回值，则可以直接打印出调用结果，或者把结果保存到变量中，例如 5.1 节中的 sum(a,b)函数有返回值，使用返回值的代码如下：

```
console.log(sum(1, 2));                    //3;
let result = sum(3, 4);
console.log(result + 10);                  //17;
```

第一行直接使用了 console.log()打印出了 sum 函数的返回值，即 1+2 的结果，第二行则把 3+4 的结果保存到 result 变量中，后面继续使用 result 这个变量，把它加上 10 之后打印出了结果。

5.3 函数表达式

本节介绍一下函数作为表达式的用法。在 JavaScript 中，函数是“头等公民”(First-Class Citizens)，即函数可以作为表达式赋值给变量，或可以作为参数传递给其他函数，另外函数本身也是对象，它几乎可以用于各种语法结构中，所以使 JavaScript 的语法变得十分灵活。

之前介绍过表达式是能够返回计算结果的代码，那么函数作为表达式，它返回的值是函数代码本身，代码如下：

```
function square(x) {
  return x * x;
}
console.log(square);
```

这里使用了 console.log 直接打印函数本身，注意不要写小括号，否则就变成了函数调用，上方代码的输出结果如下：

```
f square(x) {
  return x * x;
}
```

可以看到输出的结果就是声明函数时的代码。这样就可以把函数赋值给一个变量或常量，代码如下：

```
const square = function square(x) { return x * x }
```

在把函数保存到变量中时，推荐使用 const 将此值定义为常量，防止后面变量被其他值覆盖，在后边小节中提到变量时，若无特殊说明，均代表变量或常量。

在上方代码中还可以注意到 const 定义的变量名与函数名是一样的，为了使代码更简洁，可以省略函数的名字，代码如下：

```
const square = function (x) { return x * x }
```

这种使用变量的形式定义的函数，它的调用方法跟函数一样，只需使用变量名，例如 square(5)。

省略了名字的函数又叫作匿名函数(Anonymous Function)。单纯的匿名函数在出错的时候，会难以发觉错误究竟出在哪个函数中，所以不推荐使用，而像上述那样使用变量保存函数表达式之后，主流浏览器与 Node 环境会自动推断出函数的名字，所以不会有此问题。

5.4 箭头函数

在 ES6 出现以后，函数的定义又有了新的形式：箭头函数(Arrow Function)，它是普通函数的简化形式，由参数列表、=>符号和{}包裹的函数体构成，可以参考下方语法示例：

```
(参数列表) => {
  语句块;
}
```

来看一个具体的例子，例如定义一个箭头函数并返回两个参数的和，代码如下：

```
const sum = (x, y) => { return x + y; }
```

这里小括号中的 x 和 y 为箭头函数的参数，然后使用=>箭头符号引出函数体，在里边返回 x+y 的值，箭头函数本身也是表达式，且为匿名函数，所以把它的返回值保存到名为 sum 的变量中，方便后续调用。调用箭头函数的方式与普通函数一样，例如 sum(1,2)，返回的结果为 3。

对于只有一个参数的函数，参数的小括号可以省略，代码如下：

```
const increment = a => { return a + 1; }
```

上方箭头函数中，因为函数体中也只有一条语句，这时可以把大括号和 return 同时省略，代码如下：

```
const increment = a => a + 1;
```

这样的形式看起来就简洁多了。需要注意的是，如果箭头函数没有参数，则必须保留小括号，代码如下：

```
const getDefaultSize = () => 10;
```

另外，如果箭头函数返回的是对象，则代表对象的大括号会与函数体的大括号冲突，这种情况下可以使用小括号包裹对象，来作为返回值，或者使用 return 语句，代码如下：

```
const createPerson = () => ({name: "Wang"});
const createPerson = () => { return {name: "Wang"} }
```

5.5 可选与默认参数

在 JavaScript 中，函数的参数都是可选的，可以不传或者只传一部分。这样如果函数不要求传递全部参数，则需要在函数体中对参数进行判断；如果传递了某些参数则进行一些操作；如果传递了其他参数或没有传递参数则进行另一些操作。由于函数的参数是从左向右传递的，右边没传递的参数就会自动成为可选参数。

常见的场景是，一些 JavaScript 库会把配置项作为最后一个参数，这样在调用的时候，如果需要自定义配置则传递配置项参数，如果不需要就不传递，代码如下：

```
//chapter5/optional_params1.js
//options 为可选参数
function init(arg1, arg2, options) {
  //初始化操作
  if(options) {
    //使用自定义配置
  }
}
init("value1", "value2");
init("value1", "value2", { prop: "value" });
```

示例 init()函数中的 options 配置项为可选参数，里边使用了 if 语句来判断它是否传递了，如果传递了会把里边的配置项拿出来覆盖默认的配置，如果没有传递则使用默认值。后续在调用时，第 1 种调用方法没有传值，它不会执行到函数中的 if 语句，第 2 种调用方式传递了值，那么它就会进入 if 语句。

如果想让可选参数在没有传值的时候使用默认值，则可以在参数列表中直接给它赋值，例如，假设有一个绘制矩形图案的函数，将默认宽度设置为 10，将高度设置为 5，这样在调用函数的时候，如果没有传递参数，就会使用默认的宽和高进行绘制，代码如下：

```
function drawRect(width = 10, height = 5) { ... }
drawRect();                          //全部使用默认值
drawRect(20);                        //高度使用默认值
drawRect(undefined, 15);             //宽度使用默认值
```

这里需要注意的是，JavaScript 是根据参数值是否为 undefined 来判断默认参数是否传递了值的。

示例中第 1 种调用方式 drawRect()没有给 width 和 height 传值，所以会全部使用默认值。第 2 种调用方式 drawRect(20)则只传递了 width，这时 width 的默认值会被用户传递的值覆盖，即变为 20，而 height 则仍然使用默认值 5。

第 3 种调用方式 drawRect(undefined,15)使用了 width 的默认值，而 height 会使用自定义的 15。由于参数是由左向右传递的，如果想让 width 使用默认值，而 height 使用自定义的值，则需要给 width 传递 undefined 来让默认值生效，这里不能设置为 null，null 会被视为传递了值。

给参数赋默认值时，还可以使用前边参数的值，或者使用外部变量的值，代码如下：

```
//chapter5/optional_params2.js
let defaultColor = "#02cf13";
function drawRect(width = 10, height = width / 2, color = defaultColor) {
  console.log(width, height, color);
}
drawRect();                          //10 5 #02cf13
```

drawRect()函数的 height 参数使用 width 值的一半作为自身的默认值，color 使用了外部定义的 defaultColor 的值。

5.6 可变长度参数

可变长度参数与可选参数的操作正好相反，给函数传递的参数数量可以多于参数列表中所规定的。常见的 console.log()就是接收可变长度参数的例子，它接收多个以逗号分隔的参数，然后在命令行中打印出它们的值，并以空格分开。

要访问传递给参数列表以外的参数值，有两种方式，一种是使用 arguments，另一种是使用 rest 运算符，把可变长度参数放到参数列表的最后，把多余的参数收集起来。

5.6.1 arguments

首先来看一下 arguments 的使用方法。在除了箭头函数以外的普通函数中，都会有一

个隐式的 arguments 变量，它是一个类似于数组的数据结构，说它类似，是因为它与数组的结构类似，有 length 长度属性，并且使用下标访问元素，但是并不具有数组内置的方法，例如 map()、push()等。

来看一个例子，假设一个函数可接收两个参数：function func(a, b){}，如果在调用的时候给它传递了 3 个参数：func(1, 2, 3)，则 arguments 保存的值就相当于是[1,2,3]，要访问第 3 个元素，可以直接使用 arguments[2]。

再来看一个例子，定义一个函数，该函数可以根据指定的分隔符把所传递的字符串连接起来，代码如下：

```
//chapter5/varargs1.js
function joinStrings(seperator) {
  let result = "";
  for (let i = 1; i < arguments.length; i++) {
    if (i > 1) {
      result += seperator;
    }
    result += arguments[i];
  }
  return result;
}
console.log(joinStrings(", ", "react", "node"));
```

这里 joinStrings()只显式地接收了一个参数：分隔符，但是它仍然需要在分隔符后边接收多个字符串参数，这些参数会保存到 arguments 中。接着在函数体里循环每个参数，这里把下标为 0 的排除，因为它是 seperator 参数的值，然后把字符串拼接成按 seperator 指定的值分隔的一串字符并返回。

上述代码的输出结果为"react, node"。不过这样的代码难以阅读，在调用的时候，并不知道函数还可以接收多个参数，除非查看函数代码或文档才可清楚它的用法。要解决这个问题，可以使用 rest 运算符。

5.6.2 rest 运算符

rest 运算符使用...表示，后面加上标识符，用于引用它的值。使用 rest 运算符定义的参数是一个真正的数组，可以调用数组中的方法，对于数组的具体用法，将在第 6 章数组中介绍，这里只是演示使用 rest 运算符定义可变长度参数。

5.6.1 节使用 arguments 访问参数的代码，可以使用 rest 运算符进行简化，代码如下：

```
//chapter5/varargs2.js
function joinStrings(seperator, ...strs) {
  return strs.join(seperator);
}
console.log(joinStrings(", ", "react", "node"));
```

strs是一个数组，保存了除 seperator 以外所有参数的值，因为数组里有 join 方法用于连接字符串，这里只需把 seperator 传递给它，这样 strs 中的所有字符串就会拼接成一个长字符串。

有一点需要注意，rest 语法确切地说并不是一个运算符，而是一个语法标记，在不同的使用环境中有不同的作用，为了方便引用该语法的名字，本书将统一使用 rest 运算符来表示...语法。

5.7 回调函数

之前提到函数是 JavaScript 的“头等公民”，那么函数也可以作为另一个函数的参数传递进去，一般像 HTML 元素触发事件、请求远程数据或者在 Node 中操作数据库时，这些操作相对比较耗时，为了提高程序的响应速度，它们都会接收一个回调函数，在这些事件完成之后，再调用回调函数来通知该事件已完成。

回调函数是把函数作为另一个函数的参数的形式，这样可以提前在回调函数中写好要执行的代码，并传递给需要回调函数的其他函数，其他函数会在适当的时机调用回调函数，并传递相应的参数。至于回调函数接收什么样的参数，完全依靠接收回调函数的其他函数，所以一般在程序的文档中会写明该函数会给回调函数传递什么参数。

例如有一个将用户保存到数据库的函数，代码如下：

```
/**
 * //chapter5/callback1.js
 * @param {object} user 用户数据
 * @param {(success: boolean) => void} callback
 */
function addUser(user, callback) {
  console.log(`保存 ${user.username} 成功!`);
  callback(true);
}
addUser({ username: "user" }, function (success) {
  if (success) {
    //成功后的操作
    console.log(`添加成功!`);
  }
});
```

addUser()函数的第 1 个参数是要保存的用户数据，第 2 个参数是回调函数，函数体里简化了与保存有关的业务代码，只关注回调函数。在成功地保存了用户数据之后，就调用了 callback 参数所代表的函数，并给它的参数传递了 true 表示成功。后面在调用 addUser()函数时，第 1 个参数传递了示例的 user 对象数据，第 2 个参数则直接传递了一个匿名函数，函数体中是保存用户成功之后要做的操作，也可以使用箭头函数的形式，代码如下：

```
addUser({ username: "user" }, (success) => {
  //省略函数体
});
```

这样把函数传递给 addUser()函数之后，里边的 callback 就相当于 let callback = function(success){ }，可以直接使用 callback 变量调用回调函数。代码的输出结果如下：

```
保存 user 成功!
添加成功!
```

使用回调函数时，有下面几点需要注意：

(1) 回调函数可接收哪些参数，需要开发者规定清楚，通过 API 文档的形式告知调用者回调函数的形式是什么样的，接收哪些参数，都代表什么意义等。例如示例中使用文档注释规定：callback 接收一个 boolean 类型的参数，并且没有返回值。

(2) 回调函数的参数名可以使用任意合法的标识符，但是应该尽量使用有意义的名字。一般会在文档注释中指定一个有意义的名字，但是在实际调用的时候可以自定义，例如上例中的参数名也可以改成 result，即 function(result){}。

(3) 不要嵌套太多层的回调函数，例如在回调函数中继续接收其他回调函数作为参数，这样的代码难以阅读和维护，再加上编辑器对代码的缩进，整个代码看起来就像个金字塔形，并且在结束的时候会有好多)和}，这样就形成了俗称的回调地狱(Callback Hell)。要避免这种情况，可以把回调函数放到外边来定义，而不是在参数中定义，或者使用 Promise、Async/await(见第 12 章异步编程)。

5.8 作用域

在继续深入研究函数之前，有必要先了解作用域(Scope)的概念，作用域是当前执行环境的上下文(Current Context of Execution)，它限制了变量、函数等的可见性，在当前作用域下定义的变量、函数等只能在当前及内部嵌套的子作用域中访问，而不能在外层或父作用域中访问，这样可以避免变量和函数的命名冲突，还可以形成私有数据，从而保护数据不被外部作用域中的代码篡改。作用域分为全局作用域(Global Scope)和局部作用域(Local Scope)两种。

5.8.1 全局作用域

在全局作用域中定义的变量、函数等可以在任何地方访问。在 JS 源代码最外层定义的变量、函数等都是在全局作用域中的，例如下方代码中的变量 a 和 func()函数都定义在全局作用域中，并且在函数中可以访问全局作用域中的 a，代码如下：

```
let a = 10;
function func() { console.log(a) }
a;                              //10
func();                         //10
```

之前提到过最好不要使用 var 关键字定义变量，这是因为在浏览器环境中，使用 var 定义的全局变量，同时也会注册到全局对象 window 中（Node 环境下不会），并且在浏览器开发环境中有经常需要使用到第三方库的情况，稍有不慎就会有同名的变量同时被注册到全局变量中，导致互相覆盖而引发问题，代码如下：

```
//chapter5/scope1.js
var x = 10;
globalThis.x;                   //10;
var x = "Hello";
globalThis.x;                   //"Hello"
```

上方代码使用 var 定义了一个全局作用域的变量，并且使用 globalThis 访问全局变量（globalThis 是 ES2020 中的新特性，用于统一访问全局对象，即在浏览器中是 window，而在 node 中则是 global），后面又使用同名变量覆盖了它的值，当再次访问时会发现变量值改变了，后续如果想再做与数字相关的操作，就会有问题。关于覆盖的问题，JavaScript 中使用 var 关键字定义的变量可以重复定义，后定义的变量会覆盖前边的，代码如下：

```
var a = 5;
var a = 6;
console.log(a);                 //6
```

要避免这个问题，可以使用 let 关键字，代码如下：

```
let b = 10;
let b = 12;
console.log(b);                 //SyntaxError:标识符 b 重复定义
```

需要注意的是，如果是在 Chrome 开发者工具中的 Console 面板编写代码，则允许使用 let 重复定义变量，这是为了方便在同一个 Console 环境下，使用相同的变量名编写不同的测试代码，省去思考新变量名的困扰。

5.8.2 局部作用域

在函数中定义变量时，会创建一个局部作用域，在函数外边无法访问函数内部的变量，无论是使用 var、let 还是 const 定义的，代码如下：

```
function func() { var x = 5 };
x;                    //引用错误:x 未定义
```

局部作用域可以访问全局作用域中的变量和函数,也可以访问父级及以上作用域中的变量和函数,如果有同名的变量或函数,则子作用域会覆盖父作用域中的变量或函数,代码如下:

```
//chapter5/scope2.js
let x = 5;
function outerFunc() {
  let x = 4;
  function innerFunc() {
    let x = 7;
    console.log(x);
  }
  console.log(x);
  return innerFunc;
}
let innerFunc = outerFunc();
innerFunc();
console.log(x);
```

上方代码输出结果是:

```
4
7
5
```

首先,代码一开始定义了全局作用域的 x,其值为 5,而在 outerFunc()函数中,定义了同名变量 x,它的值为 4,这时 x 的值在 outerFunc()中是 4,覆盖了全局中的 5。后面又在 outerFunc()中定义了 innerFunc()函数,并且在里边再次覆盖了 x 的值,变成了 7,而 7 这个值只会在 innerFunc()中有效,在 innerFunc()大括号结束的时候就会失效,因此在 innerFunc()定义的下方打印 x 的值仍然是 4。当 outerFunc()结束时,它里边的 x 也失效了,所以最外边使用 console.log(x)时打印出的是全局作用域中的 x,其值为 5。

在局部作用域中,还有一个块级作用域(Block Scope)的概念。像{}语句块、if 语句、循环语句等会形成块级作用域,使用 let 或 const 定义的变量具有块级作用域,它们只在定义的大括号语句块中生效,离开大括号之后就不能访问了,代码如下:

```
//chapter5/scope3.js
{
  let i = 10;
}
```

```
console.log(i);                    //引用错误,i 未定义

for(let j = 0; j < 10; j++) {}
console.log(j);                    //引用错误,j 未定义
```

不过对于 var 定义的变量,则没有块级作用域的概念,在上述语句块中使用 var 定义变量之后,在语句块之外还是可以访问的,它的作用域跟语句块所在的作用域是同级的。例如,使用 var 定义循环的变量,如果循环被定义在全局作用域中,则 var 定义的变量也属于全局作用域,在 for 循环结束后仍然可以访问它的值,代码如下:

```
for(var j = 0; j < 10; j++) {}
console.log(j);                    //10
```

这里的 j 最后运行 j++之后会变成 10,在循环结束之后仍然可以访问它的值。

JavaScript 的作用域属于静态作用域,称为词法作用域(Lexical Scope),它的意思是,作用域在编写代码的时候就已经确定了,而动态作用域是程序在运行的时候,才去动态地判断作用域。词法作用域可以让理解作用域变得更简单,只需看代码就能够知道某个变量的作用域了,例如在函数中定义的作用域只需看该函数的大括号在哪里结束,那么变量的作用域就会在哪里结束。

5.8.3 提升机制

这一小节需要区分一下声明和定义,声明指的是只指定变量名但不赋值,例如 var a,定义这里指的是指定变量名并赋值,例如 var a=1。

在 JavaScript 中,函数和使用 var 声明的变量有提升(Hoisting)机制,可以先使用后声明。JavaScript 编译器会提前检查代码中的函数及 var 变量,把它们提升到当前作用域的顶部,这样就能保证代码的正常运行了。例如,测试使用 var 声明的变量的提升机制,代码如下:

```
x = 5;
console.log(x);                    //5
var x;
```

上边代码中的 var x 声明被提升到了 x=5 的上方,作为第一行代码,然后才给 x 赋值为 5,这样打印出来的值就是 5。需要注意的是,变量在提升的时候,因为只有声明部分被提升,所以如果在声明变量的同时进行了定义,再在上方访问该变量就会返回 undefined,代码如下:

```
console.log(x);                    //undefined
var x = 5;
console.log(x);                    //5
```

它相当于如下代码：

```
var x;
console.log(x);                    //undefined
x = 5;
console.log(x);                    //5
```

代码中的 var x 被提升到最顶部，剩下的赋值语句则保持在原位。

而如果使用 let 或者 const 关键字定义变量，则不能提前使用它们定义的变量，而是会直接抛出异常，代码如下：

```
a = 5;
console.log(a); //引用错误，不能在初始化之前访问 a
let a;
```

对于函数，使用 function 关键字定义的普通函数全部都会被提升到作用域的顶部。例如下方代码中，函数的定义会移动到 printValue()上方，代码如下：

```
printValue();                      //10
function printValue() { console.log(10) }
```

但是，对于保存在变量中的函数表达式则不会有提升机制，因为只有声明部分被提升了，而使用函数表达式进行赋值的部分并未被提升，代码如下：

```
printValue();
var printValue = function() { console.log(10) }       //类型错误:printValue 不是函数
```

利用函数的提升，可以把函数定义的细节放到代码后边，把函数的调用放到前边，以便关注代码所执行的操作，屏蔽具体的实现细节，这样可以增强代码的可读性。对于变量的提升机制，并不推荐使用，因为这样很难看出来变量是在哪定义的，从而容易引发问题，尤其是当有同名变量和函数名覆盖的时候，最难理解，代码如下：

```
//chapter5/hoisting1.js
function func() {
  return x;
  x = 5;
  function x() {}
  var x;
}
console.log(func());
```

代码输出的结果如下：

```
function x() {}
```

可以看到 func()函数最后返回的 x 值为函数 x(),而不是 5。这是因为 function x(){}的定义首先被提升到了 func()函数的第 1 行,var x 则按顺序提升到了第 2 行,由于声明变量 x 的时候并没有赋值,它不会覆盖掉函数 x()的定义,之后就直接运行到 return x 语句了,返回了函数 x(),而 x=5 并没有机会被执行,代码如下:

```
function func() {
  function x() {}
  var x;
  return x;
  x = 5;
}
```

5.8.4 临时隔离区

使用 let 关键字定义的变量,不能在初始化之前访问的原因是,它的声明被放到了临时隔离区(Temporal Dead Zone,TDZ)。临时隔离区会在执行块级作用域的第 1 行代码前生效,在变量初始化完成之后才会把变量从隔离区里释放出来。来看一个例子,代码如下:

```
let a = 5;
function test() {
  console.log(a);        //引用错误,不能在初始化之前访问 'a'
  let a = 6;
}
test();
```

在代码中,函数 test()的外部和内部定义了同名的变量 a,但是在函数中打印 a 的值时却抛出了错误。这就是临时隔离区的作用,虽然 test()函数的外部作用域中有 a 变量,但是在函数内部这个块级的作用域中,它会在一开始把最后边 a 变量的声明放到临时隔离区中,只有在执行完 $a=6$ 时才会从隔离区释放出来,在此期间,是不能访问隔离区中的变量的,所以打印 a 的值抛出了引用错误。

之所以称它为临时隔离区,是因为它只短暂地存在于变量初始化的过程中,而不是按代码的位置来判断是否放入隔离区,例如下方示例是可以正常执行的,代码如下:

```
let a = 5;
function test() {
  const inner = () => console.log(a);
  let a = 6;
  inner();
}
test();                  //6
```

这是因为在 inner()函数调用前，临时隔离区在 let a=6 这行代码之后就已经结束了，*a* 在 test()函数这个作用域中已经成功被初始化为 6，再在 inner()中就可以访问它的值了。

5.9 闭包

10min

从这一小节开始，将介绍与 JavaScript 有关的函数式编程(Functional Programming)的基本概念。函数式编程以函数为中心，每个操作都是一个函数，通过对函数的组合和复用来形成复杂的业务逻辑。

函数式编程的最终目的是只需调用一次函数，就可以完成所有业务逻辑，它属于声明式编程范式(Declarative Programming Paradigm)，而之前章节的代码则大部分属于命令式编程范式(Imperative Programming Paradigm)，即完成一个业务所关注的重点在于有哪些步骤。由此可见 JavaScript 支持多种编程范式。本节先介绍函数式编程中闭包的概念。

闭包(Closure)指的是一种语法形式和变量查找机制，在一系列嵌套的函数中，所有内部的函数都可以访问外部函数及全局作用域中定义的变量、对象和函数(以下简称内容)等。按这样的说法，JavaScript 中的函数全部都是闭包。因为在全局作用域中定义的函数，可以访问全局作用域的内容，在函数中定义的子函数则可以访问外层函数直到全局作用域中的所有内容。

例如定义一个 sayHello()函数，可接收一个人名 name 作为参数，打印出“你好!”，并带上人名，但是打印的代码放到 sayHello()的子函数 message()中，在 sayHello()内部调用 message()，代码如下：

```
//chapter5/closure1.js
function sayHello(name) {
  function message() {
    console.log("你好!" + name);
  }
  message();
}
sayHello("李明");                    //你好!李明
```

上方示例会输出：“你好！李明”。从输出结果看，message()函数成功地访问了 sayHello()函数中的 name 参数的值，这样的结构就形成了一个闭包。

在闭包中，内部的函数可以捕获(Capture)外部函数作用域中的内容，如变量、其他函数等，这样即便把内部函数作为返回值从外部函数中返回再进行调用，它还是可以继续使用外部函数作用域中的变量和函数。通过捕获机制可以避免在多次调用函数时，需要重复向函数传递参数的问题。

假设有一个需求，可以对一个初始数值进行自定义步长的自增操作，如果使用普通函数定义，则需要多次传递初始值，代码如下：

```
//chapter5/closure_inc1.js
function increment(initialValue, step) {
  return initialValue + step;
}
let result = increment(10, 1);                    //11
result = increment(result, 1);                    //12
result = increment(result, 2);                    //14
```

示例中对 10 进行一次步长为 1 的自增，然后把结果 11 保存到 result 变量中，接着又对 result 进行步长为 1 的自增操作，此时仍然需要传递一次自增参数，得到结果 12 后，又把它保存到变量 result 中，再进行一次步长为 2 的自增，这一次仍然需要把 result 作为参数传递给 increment()函数，这些调用反复使用 result 参数和步长值，有很多重复代码，但是如果把代码改成使用闭包的形式，则可以避免这种情况，例如把 increment()函数的定义改成闭包的形式，代码如下：

```
//chapter5/closure_inc2.js
function increment(initialValue) {
  let result = initialValue;
  return function by(step) {
    result += step;
    return result;
  };
}
```

这里的 increment()函数接收一个 initialValue 参数，用于指定初始值，之后对它进行自增操作，然后在 increment()函数内部定义一个 result 变量用于保存自增结果，并返回一个子函数 by()。by()函数接收一个 step 参数，用于指定自增步长，它会把外部函数中 result 的值加上 step 的值之后返回。这时调用 increment()函数并返回 by()函数后，by()函数会捕获 result 变量的值，使每次调用都能够记住 result 而不用再次传递了，所以只需传递步长参数，代码如下：

```
//chapter5/closure_inc2.js
const incFiveBy = increment(5);
console.log(incFiveBy(2));                    //7
console.log(incFiveBy(4));                    //11
```

这里，代码首先使用 increment()函数设置了初始值 5，然后使用 incFiveBy()保存返回值，即内部的 by()函数，这时 by()函数就已经捕获了 result 的值 5，并形成了一个闭包，之后调用 incFiveBy(2)，会对 5 进行加 2 操作，并把结果再次赋值给 result 并返回，此时 result 的值为 7，再次调用 incFiveBy(4)进行加 4 操作就会基于 7 进行操作，结果返回 11。

从结果可以看到，incFiveBy()中的 result 值是共享的，可以把它称为状态(State)，每次

调用 incFiveBy()的时候都会修改状态，这个是闭包的用途之一，在多次函数调用之间共享状态。不过，状态值只在同一个闭包内部共享，对于每次创建的新的闭包，它们之间的状态不会互相影响，是各自独立的。例如再对一个数字 10 进行自定义步长的自增操作，那么它不会影响之前对 5 的操作，代码如下：

```
//chapter5/closure_inc2.js
const incTenBy = increment(10);
console.log(incTenBy(3));                    //13
console.log(incTenBy(5));                    //18
console.log(incFiveBy(1));                   //12
```

为了达到调用一次函数就可以完成所有操作的目的，并且消除重复传递 step 步长参数，还可以对上方示例代码进行精简，把特定步长的自增再单独定义成函数，这时函数将不再接收参数，而是在函数内部直接把步长参数写死。例如，把对 5 进行自增 2 和自增 4 的操作定义成没有参数的函数，代码如下：

```
//chapter5/closure_inc3.js
const incFiveBy = increment(5);
const incFiveByTwo = () => incFiveBy(2);
const incFiveByFour = () => incFiveBy(4);
```

这样每次在调用 incFiveByTwo()和 incFiveByFour()时，都会对结果进行自增 2 和自增 4 的操作，代码如下：

```
//chapter5/closure_inc3.js
console.log(incFiveByTwo());                 //7
console.log(incFiveByFour());                //11
console.log(incFiveByFour());                //15
```

闭包还有一个用处：定义私有的状态。由于在闭包的外部，无法访问内部作用域，因此可以对内部状态起到保护作用，调用者只能使用闭包暴露出来的函数或对象等对状态进行修改，除此之外就没有其他办法修改内部的状态了。

例如，对于一组数据，允许访问当前元素，并且有向前和向后移动索引的操作，但不允许修改数据的值(可以想象为轮播图或音乐播放器)，那么可以通过闭包的形式定义数据和操作数据的函数，然后通过一个对象把这些函数暴露给外界，用以移动索引，代码如下：

```
//chapter5/closure2.js
function data() {
  let arr = [1, 3, 5, 7, 9];
  let index = 0;
  return {
    value() {
```

```
      return arr[index];
    },
    next() {
      index = ++index % arr.length;
    },
    pre() {
      index = (--index + arr.length) % arr.length;
    },
  };
}
```

这里使用对象形式返回了3个函数，如果无法理解此段代码也没关系，可以在看完第7章之后回过头来重新研究本示例，现在可重点关注对arr数组的保护。value()函数用于获取当前索引的元素，next()用于向前移动一位索引，超出数组长度后索引会回到0重新开始，pre()则是向前移动一位，超出后会回到最后一位继续向前，代码如下：

```
//chapter5/closure2.js
const myData = data();
console.log(myData.value());                    //1
myData.next();                                  //index: 1
myData.next();                                  //index: 2
console.log(myData.value());                    //5

myData.pre();                                   //index: 1
myData.pre();                                   //index: 0
myData.pre();                                   //index: 4
console.log(myData.value());                    //9
```

可以看到，除了使用data()函数暴露出来的3个函数访问数组之外，就再也无法在data()外部使用任何方式篡改arr数组和index索引的值了，另外在data()的外部作用域中，如果定义同名的arr或index，也不会把data()内部的数组和值给覆盖掉。

从上述例子可以看出，data()函数的名字并不重要，可以使用匿名函数，但是JavaScript不能直接使用function(){}这样的语句定义匿名函数，而需要把它保存到变量中，并且仍然需要给变量起名字。

要解决这个问题，可以在定义匿名函数的时候就立即调用它，然后使用一个变量保存它的返回结果，这种在定义的同时直接进行调用的函数称为立即执行函数表达式(Immediately Invoked Function Expression，IIFE)，它的形式是使用()把匿名函数包裹起来，然后在后边使用另一对()调用它，代码如下：

```
const myData = (function () {
  let arr = [1, 3, 5, 7, 9];
  //... 省略内部逻辑
})();
```

这样定义的函数会被立即执行，然后把结果保存到 myData 中，之后的调用和上例中一样。很多前端库会以这样的形式提供 API，其目的就是防止不同的库之间的作用域互相影响，从而导致某些库的数据被另一些库给覆盖。

使用闭包还能解决一个常见的、由全局作用域引发的问题，代码如下：

```
//chapter5/cloures_for_loop1.js
for (var i = 0; i < 3; i++) {
  setTimeout(() => {
    console.log(i);
  });
}
```

setTimeout()用于推迟一段代码的执行，它接收两个参数，第 1 个是回调函数，第 2 个是延迟时间，回调函数中的代码会在指定延迟时间之后执行，如果忽略了第 2 个参数，则会在 for 循环完成之后立即执行回调函数。

代码中使用循环创建了 3 个要延迟执行的代码，均为打印 i 的值。代码的运行结果很容易就会被认为是 0 1 2，但实际上是 3 3 3。原因在于，使用 var 定义的变量的作用域是全局的，在 for 循环结束的时候 i 的值已经变成了 3，那么后边打印 i 的值就全部都是 3 了。要解决这个问题可以使用立即执行函数创建一个闭包，通过把 i 当作参数传递给它来捕获 i 的值，从而可以打印出 0 1 2，代码如下：

```
//chapter5/cloures_for_loop1.js
for (var i = 0; i < 3; i++) {
  (function (i) {
    setTimeout(() => {
      console.log(i);
    });
  })(i);
}
```

或者另一个解决方法是直接使用 let 定义指示变量 i，这样它的作用域为块级，每次在 for 循环开始时会产生一个新的作用域，这样每个 setTimeout()中 i 的值就不会受影响了。

之前提到了任何一个 JavaScript 函数都会形成一个闭包，这是由 JavaScript 语言本身的特性决定的。JavaScript 的作用域为词法作用域，与之相关的有词法环境（Lexical Environment），它是 ECMAScript 规范中描述的一种特殊的对象类型，不能实际访问或者对其操作。

词法环境会在代码执行到全局作用域、函数声明、块级作用域时创建，它包含两部分：环境记录（Environment Record）和外层词法环境的引用，环境记录包含了当前作用域中定义的变量、函数等的绑定关系。在第 2 章讲解变量时，提到变量的定义是把变量值绑定到变量标识符的过程，这样环境记录中就保存了这种绑定关系。这里以伪代码的形式展示了词

法环境的结构，代码如下：

```
{
  variable1: value1,
  variable2: value2,
  function1: function() {},
  ...,
  outer: <外部词法环境引用>
}
```

在某个作用域的代码执行前，JavaScript 会把该作用域中变量的声明、函数的定义先行记录到词法环境中。这里需要注意的是，词法环境中首先记录的是变量的声明，仅仅包含标识符，它对应的变量值会被设置为 undefined，而函数的定义（包括函数体）则会被全部记录到词法环境中。在记录函数时，还会把当前词法环境保存到函数内部的[[Environment]]属性中。

之后在运行代码时，如果遇到变量定义语句，则会对当前词法环境中的变量进行赋值；如果遇到新的作用域（如内部函数），则会用同样的过程创建一个新的词法环境，并把 outer 设置为上一层的词法环境。

在内部的作用域中，如果要访问某个变量或函数，则会首先在本身的词法作用域中寻找，如果没有，则会到 outer 引用的外层词法作用域中寻找，直到全局词法环境中；如果找到了，则会返回相应的值，如果没找到就返回 undefined。全局词法环境对应的是全局作用域。因为本节介绍闭包，所以这里以它为例来介绍一下词法环境的创建过程，代码如下：

```
function sayHello(name) {
    return function message() {
    console.log("你好!" + name);
  }
}
let greet = sayHello("李明");
greet();                    //"你好!李明"
```

这个示例把之前的示例代码稍做了一些改动，让 sayHello()直接返回 message()函数，并在外边调用，可以看到 greet()函数在外边调用时还能访问 name 的值。代码在执行前，会先创建全局词法环境，代码如下：

```
globalEnv
{
  greet: undefined,
  sayHello: function(name) { /* 省略代码体 */ }
  outer: null
}
```

globalEnv是为了方便描述所起的假想的名字，它代表全局词法环境对象，它会记录greet变量的声明和sayHello()函数的定义，对外层词法环境的引用为null，因为它本身是全局词法环境，没有再高一层的词法环境了。同时，globalEnv词法本身也保存到sayHello()的[[Environment]]属性中了。

接下来代码执行let greet = sayHello("李明")，调用sayHello()函数，此时在进入sayHello()函数时会创建一个新的词法环境，这里称它为sayHelloEnv，代码如下：

```
sayHelloEnv
{
  name: "李明",
  message: function() {},
  outer: globalEnv //即 sayHello 中 [[Environment]] 属性的值
}
```

在sayHelloEnv这个词法环境中，记录了参数name和message()函数的定义，并把外层词法环境设置为globalEnv，作为sayHello()函数中[[Environment]]属性的值，然后sayHelloEnv会作为[[Environment]]属性值保存到message()函数中。

在sayHello()函数返回后，会把globalEnv中名为greet的标识符绑定为sayHello("李明")的返回值。接着调用greet保存的函数，此时进入message()函数体中，又会创建一个新的词法环境，这里称它为messageEnv，代码如下：

```
messageEnv
{
  outer: sayHelloEnv
}
```

其中没有定义任何其他变量和函数，所以直接把它的outer设置为sayHelloEnv词法环境。在执行它里边的代码时，需要使用name的值，此时messageEnv本身并没有这个变量，所以它会到outer指向的sayHelloEnv中去寻找，结果发现了name变量，值为“李明”，那么它就可以正确地被打印出结果了。

如果再有一个greet2变量，保存了sayHello()函数的调用结果并传递了不同的name属性值，则后边在调用greet2()时会打印出不同的name属性值，同时也不会影响greet()的返回结果，代码如下：

```
let greet2 = sayHello("张三");
greet2();                    //"你好!张三"
greet();                     //"你好!李明"
```

这是因为greet()和greet2()指向了不同的词法环境。在调用sayHello("李明")时会创建sayHelloEnv词法环境，而在调用sayHello("张三")时又会创建新的sayHelloEnv2词

法环境，它们的 name 变量分别为“李明”和“张三”，且互不影响。

可以看到，通过这个词法环境机制，每个函数都保存了外层词法环境的引用，这样内部的函数都可以通过一条链的引用（可称作环境链，Environment Chain，或作用域链，Scope Chain），访问直至全局词法环境中记录的所有内容。

词法这个概念，简单来讲，就是代码的字面结构，直接可以根据大括号、函数等的位置，就能确定它们的作用域和词法环境，所以称它为静态的，而与它相对的，则是动态的，需要在程序运行时才能确定作用域的内容，这都与编程语言的实现机制有关。

5.10 递归

递归（Recursion）是一种解决问题的方法：对于一个复杂的问题，设法把它分解成子问题，然后对每个子问题还可以再分解成更小的子问题，直到子问题可以直接求出答案，之后再返回上一层问题利用子问题的答案得出该层的答案，最终解决复杂的问题。

在 JavaScript 或大部分编程语言中，递归是通过函数调用自身实现的，每个函数处理一个子问题，最后返回的结果即是问题的答案。举一个简单的例子，计算 1，2，3，…，n 所有数字的和（包括 n），可以使用递归的方式实现，代码如下：

```
//chapter5/recursion1.js
function addUp(n) {
  if (n <= 0) return 0;
  return n + addUp(n - 1);
}
console.log(addUp(10));                    //55
```

示例中定义了 addUp()函数，它接收一个参数 n，代表要最终加到的数字，例如 10。

（1）先看一下 return 语句，它使用 n 的值加上了 addUp($n-1$)的返回值，addUp($n-1$)这部分代码直接调用了 addUp()函数本身，并把 $n-1$ 的值当作参数传递进去，这样问题就分解成了 $10+9+8+\cdots+1$。

（2）再来看 if 语句，函数每次执行时，都会判断 n 的值，如果 n 小于或等于 0，则函数就会返回 0，递归到这里就停止了；如果大于 0，则继续调用自己并加上当前 n 的值。

（3）if(n<=0)return0 这行代码又叫作终止条件，每个递归函数最终都有一个这样的条件，否则递归会持续进行下去，造成栈溢出异常。

（4）当 addUp()函数的参数 n 等于 0 之后，就会开始返回结果，而之前的函数调用也就能够计算结果了，利用上一步的结果分别进行求和 $0+1+2+\cdots+10$，最后得出结果 55。

为了理解递归的调用方式，需要先了解一下 JavaScript 的函数调用栈（Call Stack）。栈是一个后进先出（Last In First Out，LIFO）的数据结构，在里边存放数据时，会把最新加入的数据放到顶部，然后取数据时，会从栈顶开始取，这样就跟存放时的顺序相反了。

JavaScript 函数调用栈就是这样一种结构，在遇到函数调用时，会把函数放到调用栈

中，如果函数内部还调用了其他函数，则会把内部函数放到栈顶，以此类推，当内部再也没有函数调用时，则从栈顶取出函数，逐个执行，这样最外层函数会在最后执行。依此上方代码的调用栈和执行过程可以表示如下：

```
调用栈(自底向上存入)          执行过程(自顶向下执行)
addUp(0)                      0
addUp(1)                      1 + addUp(1 - 1) = 1 + 0 = 1
addUp(2)                      2 + addUp(2 - 1) = 2 + 1 = 3
...                           ...
addUp(10)                     10 + addUp(10 - 1) = 10 + 45 = 55
```

左侧调用栈展示了调用 addUp(10)函数时，调用栈中的情况。在 n 等于 0 之前，相应的 addUp()函数调用都放到了调用栈中，当 n 等于 0 时，函数使用 return 终止了调用，此时程序会从调用栈顶开始，执行所有函数代码，最终计算出 addUp(10)的结果。

不过使用递归有严重的性能问题，JavaScript 分配给调用栈的内存是有限的，在进行递归调用时，因为只有当函数返回后才可以取出调用栈中的函数进行执行，如果递归层数太多或遗漏出口条件，则会一直向调用栈中放入新的函数，从而导致栈溢出，程序会提示超出了调用栈的最大容量。

一般建议使用循环代替递归，有一个普遍的理论是，任何递归都可以以循环的方式进行实现。上方的示例可以很简单地转换成循环的方式进行计算，代码如下：

```
//chapter5/recursion1.js
function addUpIterative(n) {
  let sum = 0;
  for (let i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
console.log(addUpIterative(10));
```

不过，可以看到代码的逻辑明显不如递归清晰，至于递归和循环的选择，也要根据一定的情况进行考量，例如计算斐波那契数列、深度优先搜索算法等，使用递归方式实现更简单、直观，而使用循环则需要引入新的数据结构(例如队列)来保存中间值数据，需要额外进行维护。

5.11 高阶函数

如果函数满足以下两点中的任意一点或全部，则这个函数就称为高阶函数(Higher-Order Function)：

(1) 接收另一个函数作为参数。

(2) 返回一个函数。

在闭包小节的示例中,increment()函数就是一个高阶函数,它返回了 by()函数用于对参数进行自定义步长的自增操作。第 6 章将要介绍的数组中,它里边的函数基本上都是高阶函数,如 map()、reduce()、filter()等,这些函数都接收一个函数作为参数,用于对访问的每个数组元素进行一些操作。

对于同时接收函数作为参数并返回新的函数的高阶函数,一般是对参数函数进行增强和组合,然后返回具有新功能的函数。例如,把任一函数所返回的数字结果进行平方运算,代码如下:

```
//chapter5/higher_order_func1.js
function square(f) {
  return (...args) => f(...args) ** 2;
}
const sum = (a, b) => a + b;
const squareOfSum = square(sum);
console.log(squareOfSum(1, 2));                    //9
```

代码中的 square()函数接收任意一个函数 f 作为参数,然后在 return 语句中,返回了一个新的函数,这个函数使用 rest 运算符接收了一个变长参数 args,它的返回值是调用 f()函数并进行平方运算的结果。

这里需要注意的是,后边给 f()传递的...args 是扩展(Spread)运算符,与前边的 rest 操作相反,用于把数组元素、对象属性分别赋值给一组变量,详细用法将在第 8 章(面向对象基础)进行介绍,通过这种方式就能让 square()返回的新函数把参数原封不动地传递给原函数 f(),从而在不改变行为的基础上,添加平方运算。例如示例中的 squareOfSum(1,2)中的 1 和 2 会传递给 sum()函数。

接下来,示例代码定义了一个进行加法操作的函数 sum(),对两个数字进行相加,然后调用 square()函数对 sum()函数进行包装,这样就形成了一个计算两数之和的平方的函数。

接着给 squareOfSum()函数传递两个参数 1 和 2,它们分别会传递给 sum()函数,之后执行 sum(1,2) ** 2 平方运算,得到了结果 9,后面可以继续使用 squareOfSum()函数计算其他的数字,因为闭包的特性,sum()函数已经被 squareOfSum()捕获了,只需给 squareOfSum()传递 sum()所需的参数就可以进行加法操作了,然后计算结果的平方。

使用这种方式,只要计算结果是返回数字类型的函数,都可以通过 square()函数进行包装,从而在结果的基础上进行平方运算,例如可以再定义一个计算三数之差的平方的函数,代码如下:

```
//chapter5/higher_order_func1.js
const diff = (a, b, c) => a - b - c;
const squareOfDiff = square(diff);
console.log(squareOfDiff(9, 2, 1));                    //36
```

代码的输出结果是36。这种通过自由地对函数进行组合来创造出不同的业务逻辑是函数式编程的特点。

5.12 柯里化

柯里化(Currying)是指把一个接收多个参数的函数转化为一系列接收一个参数的子函数的过程。例如,通过汇率计算1美元能兑换多少人民币,可以定义一个函数,接收美元数量和汇率为参数,并返回换算后的结果,使用普通函数实现的代码如下:

```
//chapter5/currying1.js
function usdToCny(amount, rate) {
  return amount * rate;
}
console.log(usdToCny(1, 6.78));                    //6.78
console.log(usdToCny(8, 6.78));                    //54.24
```

通过观察上例中的代码可以发现,汇率需要在每次调用的时候都传一次,那么除了可以给rate设置默认值外,也可以通过柯里化的形式,实现记住汇率值,代码如下:

```
//chapter5/currying1.js
function convertRate(rate) {
  return (amount) => amount * rate;
}
//普通调用
//console.log(convertRate(6.78)(10)); //67.8
//记录中间值
const uToC = convertRate(6.78);
console.log(uToC(1));                  //6.78
console.log(uToC(8));                  //54.24
```

调用柯里化后的函数时,变成了使用连续的小括号的形式,这样任意一步的调用结果都可以保存起来,然后进行复用。例如把汇率6.78保存到uToC()函数中,之后只需给uToC()函数传递美元数量,就可以计算出能够兑换的人民币的数量了。

再看更复杂一点的例子,假设把美元换算成日元,并且需要把人民币作为中间货币,那么可以再定义一个cToJ()函数用于把人民币换算成日元,之后再定义一个uToJ()函数用于把美元换算成日元,它里边会组合uToC()和cToJ()函数,先调用uToC()把美元换算成人民币,然后把结果作为cToJ()的参数将人民币换算成日元,代码如下:

```
//chapter5/currying1.js
const cToJ = convertRate(15.74);
const uToJ = (amount) => cToJ(uToC(amount));
console.log(uToJ(1));                  //106.7172
```

可以看到组合函数之后，最终只需传递一个美元数量便可以计算以人民币为中间货币所能够兑换的日元的数量了。


10min

5.13 Memoization

函数缓存(Memoization)指的是，对于比较耗时的函数，把已经执行过的函数的结果保存起来作为缓存，如果再次需要此函数的执行结果，则可先判断缓存，如果有缓存则直接返回缓存中的结果，如果没有则会执行函数并把返回结果保存起来。一般会使用函数的参数列表来作为缓存的 key，如果第二次调用函数传递了相同的参数，就会返回缓存中的结果。

缓存能有效地提升程序的性能，加快响应速度，一般用于较少变化、不依赖外部条件且耗时的操作，只要参数没有变化就会取缓存中的值。对于缓存所带来的性能提升效果演示，可以参考下例计算斐波那契数列的代码，首先看没有缓存的版本，代码如下：

```
//chapter5/memoization1.js
function fib(n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}
const start = new Date().getTime();
console.log(fib(50));
const end = new Date().getTime();
console.log(`总计执行了：${end - start} 毫秒`);
```

这里使用 Date 获取当前时间，然后使用 getTime()获取毫秒数，用于统计 fib()函数的运行时间。上方代码在笔者的计算机上的输出结果如下：

```
12586269025
总计执行了:212712 毫秒
```

大约 3 分钟的时间，这还只是计算第 50 个数字，可以看到性能非常差，因为每次计算最后都要计算到 fib(1)和 fib(0)，整个调用情况如下(因为篇幅有限，这里只展示 fib(4)的执行过程)：

```
                fib(4)
              /        \
          fib(3)       fib(2)
         /    \       /    \
     fib(2)  fib(1)  fib(1)  fib(0)
     /    \
fib(1)  fib(0)
```

可以看到 fib(2)、fib(1)和 fib(0)的值计算了多次，为了避免这种情况，可以把已经计算过的值缓存起来，在后续调用中先判断是否有缓存，加上缓存后的版本的代码如下：

```
//chapter5/memoization2.js
function fib(n) {
  if (n <= 1) return n;
  if (fib[n]) return fib[n];
  fib[n] = fib(n - 1) + fib(n - 2);
  return fib[n];
}
const start = new Date().getTime();
console.log(fib(50));
const end = new Date().getTime();
console.log(`总计执行了：${end - start} 毫秒`);
```

因为 JavaScript 中函数也是对象，所以可以添加属性，这里把函数对象作为缓存，使用函数的参数 n 作为缓存的 key，放到函数对象的属性中，属性值为函数的计算结果。对于每个需要参与计算的参数 n，如果缓存中有对应的值，则直接返回缓存值，如果没有则进行计算，并把结果保存到 fib[n]中，然后返回计算结果。上方代码的输出结果如下：

```
12586269025
总计执行了:9 毫秒
```

可以看到效率得到了巨大提升，计算 fib(50)，使用缓存只需 9 毫秒的时间。

如果对每个需要添加缓存的函数进行修改，会很烦琐，可以定义一个高阶函数，对原函数进行包装以便添加缓存，然后返回带缓存的版本，代码如下：

```
//chapter5/memoization3.js
function memoize(fn) {
  let cache = {};
  return function (...args) {
    if (cache[args]) return cache[args];
    cache[args] = fn(...args);
    return cache[args];
  };
}
```

代码中的 memoize()函数接收要添加缓存的函数作为参数，然后在内部维护了一个缓存 cache 对象，并返回了带缓存的函数版本，这时新的函数作为 memoize()的子函数进行返回后，由于闭包的特性，它就拥有独立的缓存 cache 对象了。

在执行返回的新函数时，会先通过参数列表作为 key 去查询缓存，如果有值则直接返回，如果没有则把参数列表传递给原函数并计算结果，再把结果保存在缓存中并返回。使用

memoize()函数，可以把一开始的 fib()函数包装成如下形式，代码如下：

```
//chapter5/memoization3.js
const fib = memoize(function (n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
});
```

调用它的代码与之前的保持一致，执行效果也与之前定义的带缓存版本的 fib()一样。

5.14 纯函数

纯函数(Pure Function)指的是，对于一个函数，每次传递相同的参数都能够得到相同的返回结果，并且没有副作用(Side Effects)产生。副作用是指函数或者表达式修改了外部状态(如全局作用域中的变量)、按引用传递的参数的值(例如对象)，或进行了 I/O(文件读写)操作和网络请求等，导致除了返回执行结果外，还产生了其他的与本函数无关的操作。

在函数式编程中，几乎要求将所有的函数必须定义成纯函数，这样方便测试和调试，因为函数式编程多用到函数之间的组合，如果有一个函数不是纯函数，则会引起整个函数依赖链上的错误。

先来看一个纯函数的例子，代码如下：

```
function sum(a, b) {
  return a + b;
}
```

这个函数计算参数 a 和 b 的和，很明显它是一个纯函数，因为如果每次传递的参数都相同，它的返回结果也是相同的，并且没有其他有副作用的代码，如果把这个函数稍加修改，它就变成了非纯函数(Impure Function)，代码如下：

```
let a = 5;
function sum(b) {
  a = 10;
  return a + b;
}
```

这时 sum()函数修改了外部变量 a 的值，虽然给它传递相同的参数，返回的值也相同，但是由于有了副作用，那么它也不能称为纯函数。它对 a 进行修改之后，其他依赖 a 的表达式或函数就会受到影响，如果是无意修改了 a 的值，则程序的执行就可能发生错误。

来看一些非纯函数的示例，代码如下：

```
//chapter5/pure_function1.js
//修改了按引用传递的参数的值,data 对象发生了变化
function update(data) {
  data.id = 5;
}
const data = {id: 1};
update(data);

//网络请求,有可能出错,后端返回的数据也可能有变化
async function getData() {
  const res = await fetch("http://test.com/api");
  return await res.json();
}
```

5.15 小结

函数是 JavaScript 中最重要的语法结构，它被称为“头等公民”，几乎可以用在代码中的任何地方，例如作为表达式、语句和函数参数等。本章前半部分介绍了函数的基本概念、不同的定义方式和调用方法，后半部分则介绍了函数式编程的基本概念，它与普通的编程范式有所区别，理解起来可能有些困难，不过随着进一步的代码练习，就能够掌握它的核心理念。本章的重点内容有以下几点：

（1）普通函数、匿名函数和箭头函数的定义方式和区别。

（2）函数参数的类型：可选参数、默认参数和可变长度参数。

（3）作用域的概念，var、let 和 const 关键字定义的变量在作用域上的不同之处，以及提升机制。

（4）函数式的核心理念：组合不同的函数完成复杂的业务逻辑，且要求函数为纯函数。

（5）闭包的概念及作用域中的状态的捕获。

（6）高阶函数、柯里化、Memoization、纯函数的概念和形式结构。