函数定义和调用

定义函数

在 JavaScript 中，定义函数的方式如下：

```
function abs(x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

上述 `abs()` 函数的定义如下：

- `function` 指出这是一个函数定义；
- `abs` 是函数的名称；
- `(x)` 括号内列出函数的参数，多个参数以 `,` 分隔；
- `{ ... }` 之间的代码是函数体，可以包含若干语句，甚至可以没有任何语句。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

如果没有 `return` 语句，函数执行完毕后也会返回结果，只是结果为 `undefined`。

由于 JavaScript 的函数也是一个对象，上述定义的 `abs()` 函数实际上是一个函数对象，而函数名 `abs` 可以视为指向该函数的变量。

因此，第二种定义函数的方式如下：

```
var abs = function (x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
};
```

在这种方式下，`function (x) { ... }` 是一个匿名函数，它没有函数名。但是，这个匿名函数赋值给了变量 `abs`，所以，通过变量 `abs` 就可以调用该函数。

上述两种定义完全等价，注意第二种方式按照完整语法需要在函数体末尾加一个`;`，表示赋值语句结束。

调用函数

调用函数时，按顺序传入参数即可：

```
abs(10); // 返回 10

abs(-9); // 返回 9
```

由于 JavaScript 允许传入任意个参数而不影响调用，因此传入的参数比定义的参数多也没有问题，虽然函数内部并不需要这些参数：

```
abs(10, 'blablabla'); // 返回 10

abs(-9, 'haha', 'hehe', null); // 返回 9
```

传入的参数比定义的少也没有问题：

```
abs(); // 返回 NaN
```

此时 `abs(x)` 函数的参数 `x` 将收到 `undefined`，计算结果为 `NaN`。

要避免收到 `undefined`，可以对参数进行检查：

```
function abs(x) {

    if (typeof x !== 'number') {throw 'Not a number'; }

    if (x >= 0) { return x } else { return -x; } }
```

arguments

JavaScript 还有一个免费赠送的关键字 `arguments`，它只在函数内部起作用，并且永远指向当前函数的调用者传入的所有参数。`arguments` 类似 `Array` 但它不是一个 `Array`：

利用 `arguments`，你可以获得调用者传入的所有参数。也就是说，即使函数不定义任何参数，还是可以拿到参数的值：

```
function abs() {

    if (arguments.length === 0) { return 0; }

    var x = arguments[0];   return x >= 0 ? x : -x;   }

    abs(); // 0      abs(10); // 10     abs(-9); // 9
```

实际上 `arguments` 最常用于判断传入参数的个数。你可能会看到这样的写法：

```
// foo(a[, b], c)

// 接收2~3个参数，b是可选参数，如果只传2个参数，b默认为null：

function foo(a, b, c) {

    if (arguments.length === 2) {// 实际拿到的参数是a和b，c为undefined

        c = b; // 把b赋给c       b = null; // b变为默认    }

}
```

要把中间的参数 `b` 变为“可选”参数，就只能通过 `arguments` 判断，然后重新调整参数并赋值。

rest 参数

由于 JavaScript 函数允许接收任意个参数，于是我们就不得不用 `arguments` 来获取所有参数：

```
function foo(a, b) {    var i, rest = [];

    if (arguments.length > 2) {

        for (i = 2; i<arguments.length; i++) {   rest.push(arguments[i]);  }

    }

    console.log('a = ' + a);    console.log('b = ' + b);

    console.log(rest);

}
```

为了获取除了已定义参数 `a`、`b` 之外的参数，我们不得不用 `arguments`，并且循环要从索引 `2` 开始以便排除前两个参数，这种写法很别扭，只是为了获得额外的 `rest` 参数，有没有更好的方法？

ES6 标准引入了 rest 参数，上面的函数可以改写为：

```
function foo(a, b, ...rest) {

    console.log('a = ' + a);    console.log('b = ' + b);    console.log(rest);

}

foo(1, 2, 3, 4, 5);

// 结果:   // a = 1   // b = 2  // Array [ 3, 4, 5 ]

foo(1);

// 结果:   // a = 1  // b = undefined  // Array []
```

rest 参数只能写在最后，前面用...标识，从运行结果可知，传入的参数先绑定 a、b，多余的参数以数组形式交给变量 rest，所以，不再需要 arguments 我们就获取了全部参数。

如果传入的参数连正常定义的参数都没填满，也不要紧，rest 参数会接收一个空数组（注意不是 undefined）。

小心你的 return 语句

前面我们讲到了 JavaScript 引擎有一个在行末自动添加分号的机制，这可能让你栽到 return 语句的一个大坑：

```
function foo() {    return { name: 'foo' }   }

foo(); // { name: 'foo' }
```

如果把 return 语句拆成两行：

```
function foo() {

    return

        { name: 'foo' };  }

foo(); // undefined
```

要小心了，由于 JavaScript 引擎在行末自动添加分号的机制，上面的代码实际上变成了：

```
function foo() {

    return; // 自动添加了分号，相当于return undefined;

        { name: 'foo' }; // 这行语句已经没法执行到了   }
```

所以正确的多行写法是：

```
function foo() {

    return { // 这里不会自动加分号，因为{表示语句尚未结束

        name: 'foo'   };

}
```

变量作用域与解构赋值

在 JavaScript 中，用 var 申明的变量实际上是有作用域的。

如果一个变量在函数体内部申明，则该变量的作用域为整个函数体，在函数体外不可引用该变量：

```
function foo() {   var x = 1;   x = x + 1 }
```

```
x = x + 2; // ReferenceError! 无法在函数体外引用变量x
```

如果两个不同的函数各自申明了同一个变量，那么该变量只在各自的函数体内起作用。换句话说，不同函数内部的同名变量互相独立，互不影响：

```
function foo() {   var x = 1;   x = x + 1   }

function bar() {   var x = 'A';  x = x + 'B';  }
```

由于 JavaScript 的函数可以嵌套，此时，内部函数可以访问外部函数定义的变量，反过来则不行：

```
function foo() {   var x = 1;

    function bar() {  var y = x + 1; // bar可以访问foo的变量x!   }

    var z = y + 1; // ReferenceError! foo不可以访问bar的变量y!   }
```

变量提升

JavaScript 的函数定义有个特点，它会先扫描整个函数体的语句，把所有申明的变量“提升”到函数顶部：

```
function foo() {     var x = 'Hello, ' + y;

    console.log(x);       var y = 'Bob'   }
```

虽然是 strict 模式，但语句 `var x = 'Hello, ' + y;` 并不报错，原因是变量 `y` 在稍后申明了。但是 `console.log` 显示 `Hello, undefined`，说明变量 `y` 的值为 `undefined`。这正是因为 JavaScript 引擎自动提升了变量 `y` 的声明，但不会提升变量 `y` 的赋值。

对于上述 `foo()` 函数，JavaScript 引擎看到的代码相当于：

```
function foo() {

    var y; // 提升变量y的申明，此时y为undefined     var x = 'Hello, ' + y;

    console.log(x);       y = 'Bob';     }
```

由于 JavaScript 的这一怪异的“特性”，我们在函数内部定义变量时，请严格遵守“在函数内部首先申明所有变量”这一规则。最常见的做法是用一个 `var` 申明函数内部用到的所有变量：

```
function foo() {

    var

        x = 1, // x初始化为1

        y = x + 1, // y初始化为2

        z, i; // z和i为undefined
```

```
}
```

全局作用域

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript 默认有一个全局对象 window，全局作用域的变量实际上被绑定到 window 的一个属性：

```
var course = 'Learn JavaScript';

alert(course); // 'Learn JavaScript'

alert(window.course); // 'Learn JavaScript'
```

因此，直接访问全局变量 course 和访问 window.course 是完全一样的。

你可能猜到了，由于函数定义有两种方式，以变量方式 var foo = function () {}定义的函数实际上也是一个全局变量，因此，顶层函数的定义也被视为一个全局变量，并绑定到 window 对象：

```
function foo() {     alert('foo');   }

foo(); // 直接调用foo()         window.foo(); // 通过window.foo()调用
```

进一步大胆地猜测，我们每次直接调用的 alert()函数其实也是 window 的一个变量：

```
window.alert('调用 window.alert()');
// 把 alert 保存到另一个变量:              var old_alert = window.alert;
// 给 alert 赋一个新函数:                  window.alert = function () {}
                                           alert('无法用 alert()显示了!');
 // 恢复 alert:          window.alert = old_alert;        alert('又可以用 alert()了!');
```

这说明 JavaScript 实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报 ReferenceError 错误。

名字空间

全局变量会绑定到 window 上，不同的 JavaScript 文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:          var MYAPP = {};

// 其他变量:              MYAPP.name = 'myapp';          MYAPP.version = 1.0;

// 其他函数:     MYAPP.foo = function () {   return 'foo';  };
```

把自己的代码全部放入唯一的名字空间 MYAPP 中，会大大减少全局变量冲突的可能。

局部作用域

由于 JavaScript 的变量作用域实际上是函数内部，我们在 for 循环等语句块中是无法定义具有局部作用域的变量的：

```
function foo() {
    for (var i=0; i<100; i++) {  ……    }    i += 100; // 仍然可以引用变量 i
}
```

为了解决块级作用域，ES6 引入了新的关键字 let，用 let 替代 var 可以申明一个块级作用域的变量：

```
function foo() {
    var sum = 0;
    for (let i=0; i<100; i++) {   sum += i;  }
    // SyntaxError:     i += 1;   }
```

常量

由于 var 和 let 申明的是变量，如果要申明一个常量，在 ES6 之前是不行的，我们通常用全部大写的变量来表示“这是一个常量，不要修改它的值”：

```
var PI = 3.14;
```

ES6 标准引入了新的关键字 const 来定义常量，const 与 let 都具有块级作用域：

```
const PI = 3.14;
PI = 3; // 某些浏览器不报错，但是无效果！          PI; // 3.14
```

解构赋值

从 ES6 开始，JavaScript 引入了解构赋值，可以同时对一组变量进行赋值。

什么是解构赋值？我们先看看传统的做法，如何把一个数组的元素分别赋值给几个变量：

```
var array = ['hello', 'JavaScript', 'ES6'];
var x = array[0];             var y = array[1];          var z = array[2];
```

现在，在 ES6 中，可以使用解构赋值，直接对多个变量同时赋值：

注意，对数组元素进行解构赋值时，多个变量要用[...]括起来。

如果数组本身还有嵌套，也可以通过下面的形式进行解构赋值，注意嵌套层次和位置要保持一致：

```
let [x, [y, z]] = ['hello', ['JavaScript', 'ES6']];

x; // 'hello'                 y; // 'JavaScript'          z; // 'ES6'
```

解构赋值还可以忽略某些元素：

```
let [, , z] = ['hello', 'JavaScript', 'ES6']; // 忽略前两个元素，只对 z 赋值第三个元素
```

如果需要从一个对象中取出若干属性，也可以使用解构赋值，便于快速获取对象的指定属性：

```
ar person = {    name: '小明',       age: 20,       gender: 'male',
    passport: 'G-12345678',      school: 'No.4 middle school'    };
var {name, age, passport} = person;
```

、

对一个对象进行解构赋值时，同样可以直接对嵌套的对象属性进行赋值，只要保证对应的层次是一致的：

```
var person = {     name: '小明',     age: 20,    gender: 'male',

    passport: 'G-12345678',       school: 'No.4 middle school',

    address: {

        city: 'Beijing',    street: 'No.1 Road',      zipcode: '100001'

    }

};

var {name, address: {city, zip}} = person;

name; // '小明'              city; // 'Beijing'

zip; // undefined, 因为属性名是 zipcode 而不是 zip

// 注意: address 不是变量，而是为了让 city 和 zip 获得嵌套的 address 对象的属性:

address; // Uncaught ReferenceError: address is not defined
```

使用解构赋值对对象属性进行赋值时，如果对应的属性不存在，变量将被赋值为 `undefined`，这和引用一个不存在的属性获得 `undefined` 是一致的。如果要使用的变量名和属性名不一致，可以用下面的语法获取：

```
var person = {     name: '小明',     age: 20,    gender: 'male',

    passport: 'G-12345678',       school: 'No.4 middle school',    };

// 把 passport 属性赋值给变量 id:       let {name, passport:id} = person;

name; // '小明'        id; // 'G-12345678'
```

```
// 注意: passport 不是变量，而是为了让变量 id 获得 passport 属性:

passport; // Uncaught ReferenceError: passport is not defined
```

解构赋值还可以使用默认值，这样就避免了不存在的属性返回 `undefined` 的问题：

```
var person = { name: '小明',     age: 20,    gender: 'male',   passport: 'G-12345678'};

// 如果 person 对象没有 single 属性，默认赋值为 true:

var {name, single=true} = person;     name; // '小明'      single; // true
```

有些时候，如果变量已经被声明了，再次赋值的时候，正确的写法也会报语法错误：

```
// 声明变量:     var x, y;

// 解构赋值:      {x, y} = { name: '小明', x: 100, y: 200};

// 语法错误: Uncaught SyntaxError: Unexpected token =
```

这是因为 JavaScript 引擎把 `{` 开头的语句当作了块处理，于是 `=` 不再合法。解决方法是用小括号括起来：

```
({x, y} = { name: '小明', x: 100, y: 200});
```

使用场景

解构赋值在很多时候可以大大简化代码。例如，交换两个变量 `x` 和 `y` 的值，可以这么写，不再需要临时变量：

```
var x=1, y=2;     [x, y] = [y, x]
```

快速获取当前页面的域名和路径：

```
var {hostname:domain, pathname:path} = location;
```

如果一个函数接收一个对象作为参数，那么，可以使用解构直接把对象的属性绑定到变量中。例如，下面的函数可以快速创建一个 `Date` 对象：

```
function buildDate({year, month, day, hour=0, minute=0, second=0}) {

    return new Date(year + '-' + month + '-' + day + ' ' + hour + ':' + minute + ':' + secon
d);   }
```

它的方便之处在于传入的对象只需要 `year`、`month` 和 `day` 这三个属性：

```
buildDate({ year: 2017, month: 1, day: 1 });
```

```
// Sun Jan 01 2017 00:00:00 GMT+0800 (CST)
```

也可以传入 `hour`、`minute` 和 `second` 属性：

```
buildDate({ year: 2017, month: 1, day: 1, hour: 20, minute: 15 });

// Sun Jan 01 2017 20:15:00 GMT+0800 (CST)
```

使用解构赋值可以减少代码量，但是，需要在支持 ES6 解构赋值特性的现代浏览器中才能正常运行。目前支持解构赋值的浏览器包括 Chrome，Firefox，Edge 等。

方法

在一个对象中绑定函数，称为这个对象的方法。

在 JavaScript 中，对象的定义是这样的：

```
var xiaoming = {   name: '小明',      birth: 1990   };
```

但是，如果我们给 `xiaoming` 绑定一个函数，就可以做更多的事情。比如，写个 `age()`方法，返回 `xiaoming` 的年龄：

```
var xiaoming = {

    name: '小明',   birth: 1990,

    age: function () { var y = new Date().getFullYear();    return y - this.birth; }};

xiaoming.age; // function xiaoming.age()

xiaoming.age(); // 今年调用是25,明年调用就变成26了
```

绑定到对象上的函数称为方法，和普通函数也没啥区别，但是它在内部使用了一个 `this` 关键字，这个东东是什么？

在一个方法内部，`this` 是一个特殊变量，它始终指向当前对象，也就是 `xiaoming` 这个变量。所以，`this.birth` 可以拿到 `xiaoming` 的 `birth` 属性。

让我们拆开写：

```
function getAge() { var y = new Date().getFullYear();   return y - this.birth;  }

var xiaoming = {name: '小明',  birth: 1990,  age: getAge};

xiaoming.age(); // 25, 正常结果      getAge(); // NaN
```

单独调用函数 `getAge()`怎么返回了 `NaN`？请注意，我们已经进入到了 JavaScript 的一个大坑里。

JavaScript 的函数内部如果调用了 `this`，那么这个 `this` 到底指向谁？

答案是，视情况而定！

如果以对象的方法形式调用，比如 `xiaoming.age()`，该函数的 `this` 指向被调用的对象，也就是 `xiaoming`，这是符合我们预期的。

如果单独调用函数，比如 `getAge()`，此时，该函数的 `this` 指向全局对象，也就是 `window`。

坑爹啊！

更坑爹的是，如果这么写：

```
var fn = xiaoming.age; // 先拿到xiaoming的age函数      fn(); // NaN
```

也是不行的！要保证 `this` 指向正确，必须用 `obj.xxx()`的形式调用！

由于这是一个巨大的设计错误，要想纠正可没那么简单。ECMA 决定，在 `strict` 模式下让函数的 `this` 指向 `undefined`，因此，在 `strict` 模式下，你会得到一个错误：

```
var xiaoming = {   name: '小明',   birth: 1990,

    age: function () {var y = new Date().getFullYear();   return y - this.birth; }};

var fn = xiaoming.age;

fn(); // Uncaught TypeError: Cannot read property 'birth' of undefined
```

这个决定只是让错误及时暴露出来，并没有解决 `this` 应该指向的正确位置。

有些时候，喜欢重构的你把方法重构了一下：

```
var xiaoming = {  name: '小明',     birth: 1990,

    age: function () {

        function getAgeFromBirth() {

            var y = new Date().getFullYear();  return y - this.birth;  }

        return getAgeFromBirth(); }  };

xiaoming.age(); // Uncaught TypeError: Cannot read property 'birth' of undefined
```

结果又报错了！原因是 `this` 指针只在 `age` 方法的函数内指向 `xiaoming`，在函数内部定义的函数，`this` 又指向 `undefined` 了！（在非 `strict` 模式下，它重新指向全局对象 `window`！）

修复的办法也不是没有，我们用一个 `that` 变量首先捕获 `this`：

```
var xiaoming = {  name: '小明',    birth: 1990,

    age: function () {   var that = this; // 在方法内部一开始就捕获this

        function getAgeFromBirth() {var y = new Date().getFullYear();
```

```
            return y - that.birth; // 用that而不是this  }

        return getAgeFromBirth();   }};

xiaoming.age(); // 25
```

用 `var that = this;`，你就可以放心地在方法内部定义其他函数，而不是把所有语句都堆到一个方法中。

apply

虽然在一个独立的函数调用中，根据是否是 strict 模式，`this` 指向 `undefined` 或 `window`，不过，我们还是可以控制 `this` 的指向的！

要指定函数的 `this` 指向哪个对象，可以用函数本身的 `apply` 方法，它接收两个参数，第一个参数就是需要绑定的 `this` 变量，第二个参数是 `Array`，表示函数本身的参数。

用 `apply` 修复 `getAge()`调用：

```
function getAge() {var y = new Date().getFullYear(); return y - this.birth; }

var xiaoming = {name: '小明',    birth: 1990,   age: getAge};

xiaoming.age(); // 25   getAge.apply(xiaoming, []); // 25, this指向xiaoming, 参数为空
```

另一个与 `apply()`类似的方法是 `call()`，唯一区别是：

- `apply()`把参数打包成 `Array` 再传入；
- `call()`把参数按顺序传入。

比如调用 `Math.max(3, 5, 4)`，分别用 `apply()`和 `call()`实现如下：

```
Math.max.apply(null, [3, 5, 4]); // 5

Math.max.call(null, 3, 5, 4); // 5
```

对普通函数调用，我们通常把 `this` 绑定为 `null`。

装饰器

利用 `apply()`，我们还可以动态改变函数的行为。

JavaScript 的所有对象都是动态的，即使内置的函数，我们也可以重新指向新的函数。

现在假定我们想统计一下代码一共调用了多少次 `parseInt()`，可以把所有的调用都找出来，然后手动加上 `count += 1`，不过这样做太傻了。最佳方案是用我们自己的函数替换掉默认的 `parseInt()`：

```
var count = 0;   var oldParseInt = parseInt; // 保存原函数
window.parseInt = function () {
    count += 1;    return oldParseInt.apply(null, arguments); // 调用原函数  };
```

```
// 测试:
parseInt('10');     parseInt('20');      parseInt('30');
console.log('count = ' + count); // 3、
```

高阶函数

高阶函数英文叫 Higher-order function。那么什么是高阶函数？

JavaScript 的函数其实都指向某个变量。既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

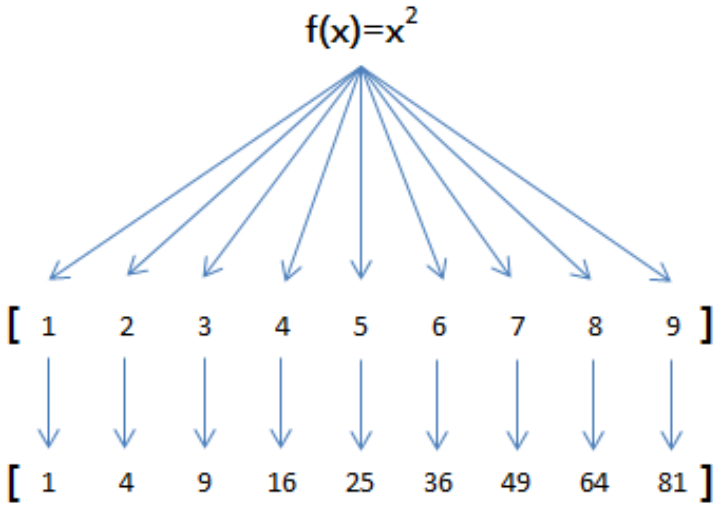
```
function add(x, y, f) {  return f(x) + f(y); }
```

当我们调用 `add(-5, 6, Math.abs)`时，参数 `x`，`y` 和 `f` 分别接收`-5`，`6` 和函数 `Math.abs`，根据函数定义，我们可以推导计算过程为：

```
x = -5;      y = 6;     f = Math.abs;

f(x) + f(y) ==> Math.abs(-5) + Math.abs(6) ==> 11;    return 11;
```

map
举例说明，比如我们有一个函数f(x)=x2，要把这个函数作用在一个数组[1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map实现如下：



由于map()方法定义在JavaScript的Array中，我们调用Array的map()方法，传入我们自己的函数，就得到了一个新的Array作为结果：

```
function pow(x) {
    return x * x;
}
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
var results = arr.map(pow); // [1, 4, 9, 16, 25, 36, 49, 64, 81]
console.log(results);// 1, 4, 9, 16, 25, 36, 49, 64, 81
```

注意：map()传入的参数是pow，即函数对象本身。
你可能会想，不需要map()，写一个循环，也可以计算出结果：

```
var f = function (x) { return x * x;};

var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];

var result = [];

for (var i=0; i<arr.length; i++) { result.push(f(arr[i])); }
```

的确可以，但是，从上面的循环代码，我们无法一眼看明白“把f(x)作用在Array的每一个元素并把结果生成一个新的Array”。
所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把Array的所有数字转为字符串：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];

arr.map(String); // ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

只需要一行代码。

reduce

再看 reduce 的用法。Array 的 `reduce()`把一个函数作用在这个 `Array` 的`[x1, x2, x3...]`上，这个函数必须接收两个参数，`reduce()`把结果继续和序列的下一个元素做累积计算，其效果就是：

```
[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)
```

比方说对一个 `Array` 求和，就可以用 `reduce` 实现：

```
var arr = [1, 3, 5, 7, 9];

arr.reduce(function (x, y) { return x + y; }); // 25
```

要把`[1, 3, 5, 7, 9]`变换成整数 13579，`reduce()`也能派上用场：

```
var arr = [1, 3, 5, 7, 9];

arr.reduce(function (x, y) {  return x * 10 + y; }); // 13579
```

如果我们继续改进这个例子，想办法把一个字符串 `13579` 先变成 `Array`——`[1, 3, 5, 7, 9]`，再利用`reduce()`就可以写出一个把字符串转换为 Number 的函数。

filter

filter 也是一个常用的操作，它用于把 `Array` 的某些元素过滤掉，然后返回剩下的元素。

和 `map()`类似，`Array` 的 `filter()`也接收一个函数。和 `map()`不同的是，`filter()`把传入的函数依次作用于每个元素，然后根据返回值是 `true` 还是 `false` 决定保留还是丢弃该元素。

例如，在一个 `Array` 中，删掉偶数，只保留奇数，可以这么写：

```
var arr = [1, 2, 4, 5, 6, 9, 10, 15];

var r = arr.filter(function (x) {  return x % 2 !== 0;  });

r; // [1, 5, 9, 15]
```

把一个 `Array` 中的空字符串删掉，可以这么写：

```
var arr = ['A', '', 'B', null, undefined, 'C', '  '];

var r = arr.filter(function (s) {

    return s && s.trim(); // 注意：IE9 以下的版本没有 trim() 方法 });

r; // ['A', 'B', 'C']
```

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

回调函数

`filter()` 接收的回调函数，其实可以有多个参数。通常我们仅使用第一个参数，表示 `Array` 的某个元素。回调函数还可以接收另外两个参数，表示元素的位置和数组本身：

```
var arr = ['A', 'B', 'C'];

var r = arr.filter(function (element, index, self) {

    console.log(element); // 依次打印 'A', 'B', 'C'

    console.log(index); // 依次打印 0, 1, 2

    console.log(self); // self 就是变量 arr

    return true;

});
```

利用 `filter`，可以巧妙地去除 `Array` 的重复元素：

```
var
    r,
    arr = ['apple', 'strawberry', 'banana', 'pear', 'apple', 'orange', 'orange', 'strawberr
y'];
    r = arr.filter(function (element, index, self) {
        return self.indexOf(element) === index;});
   console.log(r.toString());
```

去除重复元素依靠的是 `indexOf` 总是返回第一个元素的位置，后续的重复元素位置与 `indexOf` 返回的位置不相等，因此被 `filter` 滤掉了。

Sort 排序算法
排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个对象呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素x和y，如果认为x < y，则返回-1，如果认为x == y，则返回0，如果认为x > y，则返回1，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。
JavaScript的Array的sort()方法就是用于排序的，但是排序结果可能让你大吃一惊：

```
// 看上去正常的结果:

['Google', 'Apple', 'Microsoft'].sort(); // ['Apple', 'Google', 'Microsoft'];



// apple 排在了最后:

['Google', 'apple', 'Microsoft'].sort(); // ['Google', 'Microsoft", 'apple']



// 无法理解的结果:

[10, 20, 1, 2].sort(); // [1, 10, 2, 20]
```

第二个排序把apple排在了最后，是因为字符串根据ASCII码进行排序，而小写字母a的ASCII码在大写字母之后。
第三个排序结果是什么鬼？简单的数字排序都能错？
这是因为Array的sort()方法默认把所有元素先转换为String再排序，结果'10'排在了'2'的前面，因为字符'1'比字符'2'的ASCII码小。

如果不知道sort()方法的默认排序规则，直接对数字排序，绝对栽进坑里！
幸运的是，sort()方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。
要按数字大小排序，我们可以这么写：

```
var arr = [10, 20, 1, 2];
arr.sort(function (x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
});
console.log(arr); // [1, 2, 10, 20]
```

如果要倒序排序，我们可以把大的数放前面：

```
var arr = [10, 20, 1, 2];
```

```
arr.sort(function (x, y) {

    if (x < y) {  return 1; }

    if (x > y) { return -1; }

    return 0; }); // [20, 10, 2, 1]
```

默认情况下，对字符串排序，是按照ASCII的大小比较的，现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

```
var arr = ['Google', 'apple', 'Microsoft'];

arr.sort(function (s1, s2) {

    x1 = s1.toUpperCase();

    x2 = s2.toUpperCase();

    if (x1 < x2) { return -1; }

    if (x1 > x2) { return 1;  }

    return 0; }); // ['apple', 'Google', 'Microsoft']
```

忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。
从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。
最后友情提示，sort()方法会直接对Array进行修改，它返回的结果仍是当前Array：

```
var a1 = ['B', 'A', 'C'];

var a2 = a1.sort();

a1; // ['A', 'B', 'C']

a2; // ['A', 'B', 'C']

a1 === a2; // true, a1 和 a2 是同一对象
```

闭包

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个对 `Array` 的求和。通常情况下，求和的函数是这样定义的：

```
function sum(arr) {

    return arr.reduce(function (x, y) {   return x + y; });

}   sum([1, 2, 3, 4, 5]); // 15
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

```
function lazy_sum(arr) {

    var sum = function () {

        return arr.reduce(function (x, y) {  return x + y;  });

    } return sum;  }
```

当我们调用 `lazy_sum()`时，返回的并不是求和结果，而是求和函数：

```
var f = lazy_sum([1, 2, 3, 4, 5]); // function sum()
```

调用函数 `f` 时，才真正计算求和的结果：

```
f(); // 15
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum`，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()`时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
var f1 = lazy_sum([1, 2, 3, 4, 5]);

var f2 = lazy_sum([1, 2, 3, 4, 5]);

f1 === f2; // false
```

`f1()`和 `f2()`的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `arr`，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()`才执行。我们来看一个例子：

```
function count() {

    var arr = [];

    for (var i=1; i<=3; i++) {

        arr.push(function () {

            return i * i;
```

```
        });
    }
    return arr;
}
var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的 3 个函数都添加到一个 `Array` 中返回了。

你可能认为调用 `f1()`，`f2()`和 `f3()`结果应该是 `1`，`4`，`9`，但实际结果是：

```
f1(); // 16             f2(); // 16          f3(); // 16
```

全部都是 `16`！原因就在于返回的函数引用了变量 `i`，但它并非立刻执行。等到 3 个函数都返回时，它们所引用的变量 `i` 已经变成了 `4`，因此最终结果为 `16`。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push((function (n) {return function () {return n * n; }})(i));
    }
    return arr;
}
var results = count();             var f1 = results[0];
var f2 = results[1];               var f3 = results[2];
f1(); // 1              f2(); // 4             f3(); // 9
```

注意这里用了一个“创建一个匿名函数并立刻执行”的语法：

```
(function (x) {  return x * x; })(3); // 9
```

理论上讲，创建一个匿名函数并立刻执行可以这么写：

```
function (x) { return x * x } (3);
```

但是由于 JavaScript 语法解析的问题，会报 SyntaxError 错误，因此需要用括号把整个函数定义括起来：

```
(function (x) { return x * x }) (3);
```

通常，一个立即执行的匿名函数可以把函数体拆开，一般这么写：

```
(function (x) { return x * x; })(3);
```

说了这么多，难道闭包就是为了返回一个函数然后延迟执行吗？

当然不是！闭包有非常强大的功能。举个栗子：

在面向对象的程序设计语言里，比如 Java 和 C++，要在对象内部封装一个私有变量，可以用 `private` 修饰一个成员变量。

在没有 `class` 机制，只有函数的语言里，借助闭包，同样可以封装一个私有变量。我们用 JavaScript 创建一个计数器：

```
function create_counter(initial) {

    var x = initial || 0;

    return { inc: function () { x += 1;  return x;  } } }
```

它用起来像这样：

```
var c1 = create_counter();

c1.inc(); // 1         c1.inc(); // 2          c1.inc(); // 3

var c2 = create_counter(10);

c2.inc(); // 11        c2.inc(); // 12         c2.inc(); // 13
```

在返回的对象中，实现了一个闭包，该闭包携带了局部变量 `x`，并且，从外部代码根本无法访问到变量 `x`。换句话说，闭包就是携带状态的函数，并且它的状态可以完全对外隐藏起来。

闭包还可以把多参数的函数变成单参数的函数。例如，要计算 x^y 可以用 `Math.pow(x, y)` 函数，不过考虑到经常计算 x^2 或 x^3，我们可以利用闭包创建新的函数 `pow2` 和 `pow3`：

```
function make_pow(n) {    return function (x) {   return Math.pow(x, n); }  }
```

```
// 创建两个新函数:
var pow2 = make_pow(2);                    var pow3 = make_pow(3);
console.log(pow2(5)); // 25                 console.log(pow3(7)); // 343
```

脑洞大开

很久很久以前，有个叫阿隆佐·邱奇的帅哥，发现只需要用函数，就可以用计算机实现运算，而不需要`0`、`1`、`2`、`3`这些数字和`+`、`-`、`*`、`/`这些符号。

JavaScript 支持函数，所以可以用 JavaScript 用函数来写这些计算。来试试：

```
// 定义数字 0:
var zero = function (f) {  return function (x) {  return x;  } };

// 定义数字 1:
var one = function (f) {  return function (x) {   return f(x); }  };

// 定义加法:
function add(n, m) { return function (f) { return function (x) { return m(f)(n(f)(x));} }}

// 计算数字 2 = 1 + 1:            var two = add(one, one);
// 计算数字 3 = 1 + 2:            var three = add(one, two);
// 计算数字 5 = 2 + 3:            var five = add(two, three);
// 给 3 传一个函数,会打印 3 次:
(three(function () {  console.log('print 3 times');  }))();
// 给 5 传一个函数,会打印 5 次:
(five(function () {   console.log('print 5 times');  }))();
```

箭头函数

ES6 标准新增了一种新的函数：Arrow Function（箭头函数）。

为什么叫 Arrow Function？因为它的定义用的就是一个箭头：

```
x => x * x
```

上面的箭头函数相当于：

```
function (x) {   return x * x;  }
```

箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种像上面的，只包含一个表达式，连`{ ... }`和`return`都省略掉了。还有一种可以包含多条语句，这时候就不能省略`{ ... }`和`return`：

```
x => {

    if (x > 0) {

        return x * x;
```

```
    }
    else {
        return - x * x;
    }
}
```

如果参数不是一个，就需要用括号`()`括起来：

```
// 两个参数:
(x, y) => x * x + y * y

// 无参数:
() => 3.14

// 可变参数:
(x, y, ...rest) => {
    var i, sum = x + y;
    for (i=0; i<rest.length; i++) {
        sum += rest[i];
    }
    return sum;
}
```

如果要返回一个对象，就要注意，如果是单表达式，这么写的话会报错：

```
// SyntaxError:
x => { foo: x }
```

因为和函数体的`{ ... }`有语法冲突，所以要改为：

```
// ok:
x => ({ foo: x })
```

this

箭头函数看上去是匿名函数的一种简写，但实际上，箭头函数和匿名函数有个明显的区别：箭头函数内部的 `this` 是词法作用域，由上下文确定。

回顾前面的例子，由于 JavaScript 函数对 `this` 绑定的错误处理，下面的例子无法得到预期结果：

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = function () {
            return new Date().getFullYear() - this.birth; // this 指向 window 或 undefined
        };
        return fn();
    }
};
```

现在，箭头函数完全修复了 `this` 的指向，`this` 总是指向词法作用域，也就是外层调用者 `obj`：

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = () => new Date().getFullYear() - this.birth; // this 指向 obj 对象
        return fn();
    }
};
obj.getAge(); // 25
```

如果使用箭头函数，以前的那种 hack 写法：

```
var that = this;
```

就不再需要了。

由于 `this` 在箭头函数中已经按照词法作用域绑定了，所以，用 `call()`或者 `apply()`调用箭头函数时，无法对 `this` 进行绑定，即传入的第一个参数被忽略：

```
var obj = {
    birth: 1990,
    getAge: function (year) {
        var b = this.birth; // 1990
        var fn = (y) => y - this.birth; // this.birth 仍是 1990
        return fn.call({birth:2000}, year);
    }
};
obj.getAge(2015); // 25
```

generator

generator（生成器）是 ES6 标准引入的新的数据类型。一个 generator 看上去像一个函数，但可以返回多次。

我们先复习函数的概念。一个函数是一段完整的代码，调用一个函数就是传入参数，然后返回结果：

函数在执行过程中，如果没有遇到 `return` 语句（函数末尾如果没有 `return`，就是隐含的 `return undefined;`），控制权无法交回被调用的代码。

generator 跟函数很像，定义如下：

```
function* foo(x) {
    yield x + 1;       yield x + 2;      return x + 3;
}
```

generator 和函数不同的是，generator 由 `function*`定义（注意多出的`*`号），并且，除了 `return` 语句，还可以用 `yield` 返回多次。

大多数同学立刻就晕了，generator 就是能够返回多次的“函数”？返回多次有啥用？

还是举个栗子吧。

我们以一个著名的斐波那契数列为例，它由 `0`，`1` 开头：

```
0 1 1 2 3 5 8 13 21 34 ...
```

要编写一个产生斐波那契数列的函数，可以这么写：

```
function fib(max) {
    var
        t,
        a = 0,
        b = 1,
        arr = [0, 1];
    while (arr.length < max) {
        [a, b] = [b, a + b];
        arr.push(b);
    }
    return arr;


// 测试:  fib(5); // [0, 1, 1, 2, 3] fib(10); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

函数只能返回一次，所以必须返回一个 `Array`。但是，如果换成 generator，就可以一次返回一个数，不断返回多次。用 generator 改写如下：

```
function* fib(max) {
    var
        t,
        a = 0,
        b = 1,
        n = 0;
    while (n < max) {
        yield a;
        [a, b] = [b, a + b];
        n ++;  }
    return;
}
```

直接调用试试：

```
fib(5); // fib {[[GeneratorStatus]]: "suspended", [[GeneratorReceiver]]: Window}
```

直接调用一个 generator 和调用函数不一样，`fib(5)`仅仅是创建了一个 generator 对象，还没有去执行它。

调用 generator 对象有两个方法，一是不断地调用 generator 对象的 `next()`方法：

```
var f = fib(5);

f.next(); // {value: 0, done: false}   f.next(); // {value: 1, done: false}

f.next(); // {value: 1, done: false}   f.next(); // {value: 2, done: false}

f.next(); // {value: 3, done: false}   f.next(); // {value: undefined, done: true}
```

`next()`方法会执行 generator 的代码，然后，每次遇到 `yield x;`就返回一个对象`{value: x, done: true/false}`，然后“暂停”。返回的 `value` 就是 `yield` 的返回值，`done` 表示这个 generator 是否已经执行结束了。如果 `done` 为 `true`，则 `value` 就是 `return` 的返回值。

当执行到 `done` 为 `true` 时，这个 generator 对象就已经全部执行完毕，不要再继续调用 `next()`了。

第二个方法是直接用 `for ... of` 循环迭代 generator 对象，这种方式不需要我们自己判断 `done`：

```
function* fib(max) {
    var
        t,
        a = 0,
        b = 1,
        n = 0;
    while (n < max) {
        yield a;
        [a, b] = [b, a + b];
        n ++;
    }
    return;
}

for (var x of fib(10)) {
    console.log(x); // 依次输出 0, 1, 1, 2, 3, ...
}
```

generator 和普通函数相比，有什么用？

因为 generator 可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数，利用这一点，写一个 generator 就可以实现需要用面向对象才能实现的功能。例如，用一个对象来保存状态，得这么写：

```
var fib = {

    a: 0,

    b: 1,
```

```
    n: 0,
    max: 5,
    next: function () {
        var
            r = this.a,
            t = this.a + this.b;
        this.a = this.b;
        this.b = t;
        if (this.n < this.max) {
            this.n ++;
            return r;
        } else {
            return undefined;
        }
    }
};
```

用对象的属性来保存状态，相当繁琐。

generator 还有另一个巨大的好处，就是把异步回调代码变成“同步”代码。这个好处要等到后面学了 AJAX 以后才能体会到。

没有 generator 之前的黑暗时代，用 AJAX 时需要这么写代码：

```
ajax('http://url-1', data1, function (err, result) {
    if (err) {  return handle(err);  }
    ajax('http://url-2', data2, function (err, result) {
        if (err) {  return handle(err); }
        ajax('http://url-3', data3, function (err, result) {
            if (err) {  return handle(err);  }
            return success(result);
        });
```

```
    });
});
```

回调越多，代码越难看。

有了 generator 的美好时代，用 AJAX 时可以这么写：

```
try {
    r1 = yield ajax('http://url-1', data1);
    r2 = yield ajax('http://url-2', data2);
    r3 = yield ajax('http://url-3', data3);
    success(r3);
}
catch (err) {   handle(err);   }
```

看上去是同步的代码，实际执行是异步的。