第5章 函数

函数，是可以通过名称来引用，并且就像自包含了一个微型程序的代码块。利用函数，我们可以实现对代码的复用，降低代码的重复，并且让代码更加容易阅读。在JavaScript中，函数显得尤为的重要。因为函数在JavaScript中是一等公民，可以像参数一样传入和返回。所以说函数是JavaScript中的一个重点，同时也是一个难点。

本章我们将学习如下的内容：

- JavaScript中函数的基础知识
- 箭头函数
- 变量提升与函数提升
- 回调函数

5-1 函数基础介绍

学习任何语言，当学习到函数这一块时，往往都离不开以下几个问题：

- 如何创建函数
- 掌握函数的三要素

这里的函数三要素，是指函数的功能，函数的参数，以及函数的返回值。这些知识，可以说是在整个函数的知识体系中最为基础的部分。在这一节，我们就来一起看一下JavaScript中函数的这些基础知识。

5-1-1 为什么需要函数

首先我们来看一下为什么需要函数。函数最大的好处就是可以对代码实现复用。相同的功能不用再次书写，而是只用书写一次就够了。这其实就是编程里面所谓的DRY原则

> 所谓DRY原则，全称为Don‘t repeat yourself，翻译成中文就是不要重复你自己。什么意思呢？也就是说一个程序的每个部分只被编写一次，这样做可以避免重复，不需要保持多个代码段的更新和重复。

并且，我们可以把函数看作是一个暗箱，不需要知道函数的内部是怎么实现的，只需要知道函数的功能，参数以及返回值即可。

5-1-2 声明函数的方式

在JavaScript中，声明函数的方式有多种，这里我们先来介绍这3种声明函数的方式：字面量声明函数，函数表达式声明函数以及使用构造器方式来声明一个函数。

1. 字面量声明函数

这种方式是用得最为广泛的一种方式，使用关键字`function`来创建一个函数，具体的语法如下：

```
function 函数名(形式参数)
{
    //函数体
}
```

函数名：就是我们调用函数时需要书写的标识符
形式参数：简称形参，是调用函数时需要接收的参数
实际参数：简称实参，是调用函数时实际传递过去的参数

示例：

```
function test(name)
{
    console.log("Hello,"+name);
}
test("xiejie");//Hello,xiejie
```

2. 函数表达式声明函数

第二种方式是使用函数表达式来进行声明，具体的语法如下：

```
let 变量 = function(){
    //函数体
}
```

函数表达式示例：

```
let test = function(name){
    console.log("Hello,"+name);
}
test("xiejie");//Hello,xiejie
```

需要说明的是，这里的变量并不是该函数的名字，一般来讲，我们都是将一个匿名函数赋值给一个变量，然后通过这个变量来对函数进行调用。
当然，我们也可以将一个带有名字的函数赋值给一个变量。这样的声明方式被称之为命名式函数表达式
示例如下：

```
let test = function saySth(name){
    console.log("Hello,"+name);
}
test("xiejie");//Hello,xiejie
```

> 注意：虽然这种方式的函数表达式拥有函数名，但是调用的时候还是必须要通过变量名来调用，而不能够使用函数名来进行调用。

3. 构造器声明函数

使用构造器来声明函数，又被称之为通过对象的方式来声明函数，具体的语法如下；

```
let 变量 = new Function("参数","参数","函数体");
```

示例如下：

```
let test = new Function("name","console.log('Hello,'+name)");
test("xiejie");//Hello,xiejie
```

虽然通过这种方式也能够创建出函数，并且调用，但是并不推荐使用这种方式来创建函数。因为这样会导致JS解释器需要解析两次代码。先要解析这个表达式，然后又要解析字符串，效率很低，并且这样将函数体全部书写在一个字符串里面的书写方式对我们程序员也非常的不友好，所以这种方式作为了解即可。

5-1-3 函数的调用

函数的调用在前面介绍函数声明的时候也已经见到过了，就是写上函数名或者变量名，后面加上一对大括号即可。需要注意的是，一般来讲函数表示的是一个动作，所以在给函数命名的时候，一般都是以动词居多。

还一个地方需要注意，那就是如果要调用函数，那么就必须要有括号。这个括号要么在函数名后面，要么在变量名后面，这样才能够将调用函数后的执行结果返回。如果缺少了括号，那就只是引用函数本身。

示例如下：

```
let test = function(){
    console.log("Hello");
}
let i = test;//没有调用函数，而是将test函数赋值给了i
i();//Hello
```

5-1-4 函数的返回值

函数的返回值的关键字为 `return` 。代表要从函数体内部返回给外部的值，示例如下：

```
let test = function(){
    return "Hello";
}
let i = test();
console.log(i);//Hello
```

即使不写 `return` ，函数本身也会有返回值 `undefined`
例如：

```
let test = function(){
    console.log("Hello");
}
let i = test();//Hello
console.log(i);//undefined
```

需要注意的是， `return` 后面的代码是不会执行的，因为在函数里面一旦执行到 `return` ，整个函数的执行就结束了。

```
let test = function(){
    return 1;
    console.log("Hello");
}
let i = test();
console.log(i);//1
```

`return` 关键字只能返回一个值，如果想要返回多个值，可以考虑返回一个数组，示例如下：

```
//1-60的安全数 7的倍数或者以7结尾
```

```
let test = function(){
    let arr = [];
    for(let i=1;i<=60;i++)
    {
        if(i%10==7 || i%7==0)
        {
            arr.push(i);
        }
    }
    return arr;
}
console.log(test());
//[ 7, 14, 17, 21, 27, 28, 35, 37, 42, 47, 49, 56, 57 ]
```

5-1-5 函数的参数

函数的参数可以分为两种，一种是实际参数，另外一种是形式参数。这个我们在前面已经介绍过了。接下来我们来详细看一下形式参数。形式参数简称形参，它就是一种变量，但是这种变量只能被函数体内的语句使用，并在函数调用时被赋值。JavaScript中的形参的声明是不需要添加关键字的，如果加上关键字反而会报错
示例：

```
function test(let i)
{
    console.log(i);
}
test(5);
//SyntaxError: Unexpected identifier
```

JavaScript里面关于函数的形参，有以下几个注意点:
1.参数名可以重复，同名的参数取最后一个参数值

```
function test(x,x)
{
    console.log(x);
}
test(3,5);//5
```

2.即使函数声明了参数，调用时也可以不传递参数值

```
function test(x)
{
```

```
    console.log(x);
}
test();//undefined
```

3.调用函数时可以传递若干个参数值给函数，而不用管函数声明时有几个参数

```
function test(x)
{
    console.log(x);//1
}
test(1,2,3);
```

那么，这究竟是怎么实现的呢？为什么实参和形参可以不用一一对应呢？

实际上，当一个函数要被执行的时候，系统会在执行函数体代码前做一些初始化工作，其中之一就是为函数创建一个arguments的伪数组对象。这个伪数组对象将包含调用函数时传递的所有的实际参数。因此，arguments的主要用途是就是用于保存传入到函数的实际参数的。

下面的例子演示了通过arguments伪数组对象来访问到所有传入到函数的实参

```
function test(x)
{
    for(let i=0;i<arguments.length;i++)
    {
        console.log(arguments[i]);
    }
}
test(1,2,3);
// 1
// 2
// 3
```

所谓伪数组对象，就是长得像数组的对象而已，但是并不是真的数组，我们可以证明这一点

```
function test(x)
{
    arguments.push(100);//针对伪数组对象使用数组的方法
}
test(1,2,3);
//TypeError: arguments.push is not a function
```

不定参数

不定参数是从ES6开始新添加的功能，在最后一个形式参数前面添加3个点，会将所有的实参放入到一个数组里面，示例如下：

```
function test(a,...b)
{
    console.log(a);//1
    console.log(b);//[2,3]
}
test(1,2,3);
```

这里的不定参数就是一个真正的数组，可以使用数组的相关方法

```
function test(a,...b)
{
    console.log(a);//1
    console.log(b);//[2,3]
    b.push(100);
    console.log(b);//[ 2, 3, 100 ]
}
test(1,2,3);
```

还有一点需要注意的是，不定参数都是放在形式参数的最后面，如果不是放在最后，则会报错。

```
function test(...a,b)
{
    console.log(a);
    console.log(b);
}
test(1,2,3);
//SyntaxError: Rest parameter must be last formal parameter
```

默认参数

从ES6开始，书写函数的时候可以给函数的形式参数一个默认值。这样如果在调用函数时没有传递相应的实际参数，就使用默认值。如果传递了相应的实际参数，则使用传过去的参数。

```
function test(name = "world")
{
    console.log("Hello,"+name);
}
test("xiejie");//Hello,xiejie
test();//Hello,world
```

如果参数是一个数组，要为这个数组设置默认值的话，写法稍微有些不同，如下：

```
let fn = function([a = 1,b = 2] = []){
    console.log(a,b);
}
fn(); // 1 2
fn([3,4]); // 3 4
```

包括后面我们要介绍的对象，也是可以设定默认值的，但是写法和上面类似，如下：

```
let fn = function({name = 'xiejie',age = 18} = {}){
    console.log(name,age);
}
fn(); // xiejie 18
fn({name:"song",age:20}); // song 20
```

> 有关对象相关内容，请参考第6章对象。

5-1-6 函数属性和方法

1. name属性

表示函数的函数名

```
function test()
{
    console.log("Hello");
}
console.log(test.name);//test
```

我们可以通过这个name属性来证明函数表达式的变量不是函数名，如下：

```
let test = function test2(){
    console.log("Hello");
}
console.log(test.name);//test2
```

2. length属性

表示形式参数的个数，示例如下：

```
let test = function(a,b,c){
    console.log("Hello");
}
console.log(test.length);//3
```

接下来我们需要看一下 `函数名.length` 与 `arguments.length` 的区别
函数对象的length属性是表示形式参数的个数。
arguments伪数组对象的length属性是调用函数时实际参数的个数。

```
let test = function(a,b,c){
    console.log(arguments.length);//5
    console.log(arguments.callee.length);//3
}
test(1,2,3,4,5);
```

3. caller属性

caller属性并不是arguments对象的，而是函数对象本身的属性，它显示了函数的调用者，如果函数是在全局执行环境中(浏览器中)被调用，那么它的值为null，如果在另一个函数中被调用，它的值就是那个函数。
全局执行环境中被调用：
浏览器中

```
<body>
    <script>
        let test = function(){
            console.log(test.caller);
        }
        test();//null
    </script>
</body>
```

node中

```
let test = function(){
    console.log(test.caller);
}
test();//[Function]
```

被一个函数所调用：

```
let test = function(){
    let test2 = function(){
        console.log(test2.caller);
        //[Function: test]
        //因为这个函数的调用者就是test函数
    }
    test2();
}
test();
```

4.callee属性

callee是arguments对象的一个属性，该属性是一个指针，指向拥有这个arguments对象的函数

```
let test = function(){
    let test2 = function(){
        let test3 = function(){
            console.log(arguments.callee);
            //[Function: test3]
        }
        test3();
    }
    test2();
}
test();
```

callee的作用在于能够找到arguments对象所属的函数，不让函数的执行和函数名仅仅的关联在一起，我们来看下面这个例子：

```
//计算阶乘的递归函数
let test = function(i){
    if(i == 1)
    {
        return 1;
    }
    else{
        return i * test(i-1);//这里就和函数名紧紧的关联了起来
    }
}
console.log(test(3));
```

如果我们把上面的写法稍作修改，就可以看到上面写法的缺陷

```
//计算阶乘的递归函数
let test = function(i){
    if(i == 1)
    {
        return 1;
    }
    else{
        return i * test(i-1);//这里就和函数名紧紧的关联了起来
    }
}
let test2 = test;//将阶乘函数赋值给test2
//改变test这个阶乘函数的函数体
test = function(){
    console.log("我已经改变了");
}
console.log(test2(3));
// 我已经改变了
// NaN
```

所以，这个时候就可以使用arguments对象的callee属性来降低这种关联

```
//计算阶乘的递归函数
let test = function(i){
    if(i == 1)
    {
        return 1;
    }
    else{
        return i * arguments.callee(i-1);//callee指向拥有arguments对象的函数
    }
}
let test2 = test;//将阶乘函数赋值给test2
//改变test这个阶乘函数的函数体
test = function(){
    console.log("我已经改变了");
}
console.log(test2(3));//6
```

5-2 箭头函数

5-2-1 箭头函数基本介绍

所谓箭头函数，是从ES6开始新增加的一种声明函数的方式。其最大的特点在于不需要function关键字，取而代之的是使用一个 `=>` 来进行表示。箭头函数的基本语法如下：

```
let 变量 = (形式参数) => {
    // 函数体
}
```

箭头函数示例：

```
let test = (name) => {
    console.log("Hello",name);
}
test("xiejie");//Hello xiejie
```

上面所介绍的，只是箭头函数的基本写法。实际上箭头函数根据形式参数和函数体的不同，书写的方式拥有一些变形。如下：

```
//如果没有参数
let 变量 = () => {
    //函数体
}
//如果只有一个形参
let 变量 = 形参 => {
    //函数体
}
//如果函数体只有一个返回值
let 变量 = 形参 => expression
```

例如：书写求立方根的箭头函数(当然这里只是练习，ES6已经提供了求幂的方式，使用**)

```
let test = x => x*x*x;
console.log(test(3));//27
```

5-2-2 箭头函数的优点

箭头函数的优点如下：

- 比普通函数声明更简洁
- 只有一个形参就不需要用括号括起来
- 如果函数体只有一行，就不需要放到一个块中
- 如果return语句是函数体内唯一的语句，就不需要return关键字
- 不会把自己的this值绑定到函数上

当然，前面几条我们都还好理解。至于最后一条什么叫做不会把自己的this绑定到函数上，这个我们后面再讲。现在我们需要掌握的就是知道有箭头函数这么一个东西，并且能够自己书写一个箭头函数即可。

5-3 提升

在JavaScript里面，有一个非常重要的特性，就是提升。提升分为变量提升和函数提升。而在讲解提升之前，我们又需要先对作用域有一定的了解。

5-3-1 作用域

前面介绍作用域的时候，曾经给大家介绍过全局作用域以及局部作用域。事实上，在JavaScript里面，作用域一共有3种：全局作用域，函数作用域以及eval作用域

全局作用域：这个是默认的代码运行环境，一旦代码被载入，引擎最先进入的就是这个环境。

函数作用域：当进入到一个函数的时候，就会产生一个函数作用域

eval作用域：当调用eval()函数的时候，就会产生一个eval作用域

这里，我们先抛开最后一个eval作用域不说(其实这个现在用的也很少了)。我们先来看一下全局作用域和函数作用域。上面有写到，全局作用域是默认的代码运行环境。也就是说，当我们声明了一个变量或者函数的时候，它们默认就是处于全局环境中的。

```
let x = 3;
function test()
{
    console.log("Hello");
}
```

这里我们声明的`x`以及`test()`函数就是处于全局作用域里面的。

当我们调用函数的时候，这时就会又产生一个作用域，我们称之为函数作用域。在内层函数作用域里面是可以访问到全局作用域或者外层函数作用域里面的变量或者函数的。但是反过来全局作用域或者外层函数作用域里面是不能访问到内部函数作用域里面的变量或者函数的，因为它们是局部的。

函数里面访问全局作用域里面的变量

```
let x = 3;
let test = function(){
    console.log(x);//3
}
test();
```

全局作用域里面访问函数作用域里面的变量

```
let test = function(){
    let x = 3;
}
test(x);
//ReferenceError: x is not defined
```

这里在函数里面声明了变量`x`，全局里面访问的时候会提示没有定义。

> 注意在函数里面声明变量时，无论是使用的let还是var，还是说是const，它们都是属于函数作用域的，外部是无法访问的

5-3-2 变量提升

所谓变量提升，就是指在使用var关键字进行变量声明的时候，默认会将声明变量的部分提升至当前作用域的最顶上，但是注意提升的只有变量的声明部分，赋值是不会提升的

```
console.log(i);//undefined
var i = 10;
console.log(i);//10
```

还有一点要注意的是，只有var声明的变量才具有这种特性，let或者const不存在变量提升

```
console.log(i);//ReferenceError: i is not defined
let i = 10;
```

如果我们在函数里面声明变量时没有添加关键字，那么默认将会是在全局环境中声明一个变量

```
let test = function(){
    i = 10;
}
test();
console.log(i);//10
```

通过上面的代码我们可以证明这是一个全局作用域里面的变量，但是这个变量究竟是以`var`的形式声明的还是以`let`或者说`const`的方式声明的呢?
答案就是：以`var`的形式进行声明的，但是不具有变量提升。

这里我们可以证明这一点，在上面的例子中我们在外部成功访问到了在函数里面没有添加关键字

而声明的变量`i`，接下来我们来提前打印输出这个`i`变量，如下：

```
console.log(i);//ReferenceError: i is not defined
let test = function(){
    i = 10;
}
test();
```

可以看到这里会报错，显示"i is not defined"，从而证明了不具有变量提升的特性。

5-3-3 函数提升

所谓函数提升，是指当我们使用字面量方式来声明一个函数的时候，此时函数的声明会提升到当前作用域的最顶端，这意味着我们可以将函数的调用书写到函数的声明之前

```
test();//Hello!
function test()
{
    console.log("Hello!");
}
// 等价于
// test : pointer to  test()
// test()
// function test()
// {
//     console.log("Hello!");
// }
```

需要注意的是，仅仅只有普通函数声明的时候才存在函数提升，如果是使用函数表达式来进行的函数声明，则不存在有函数提升的情况

```
test();//Hello!
let test = function(){
    console.log("Hello!");
}
//ReferenceError: test is not defined
```

还有一点就是变量提升和函数提升是可以同时存在的。上面的例子中如果我们声明函数时使用的是var关键字的话，那么同样存在变量的提升，如下：

```
console.log(test);//undefined
```

```
var test = function(){
    console.log("Hello!");
}
console.log(test);//[Function: test]
```

5-4. 回调函数

在本章开篇就有提到过，函数在JavaScript中是一等公民。这里所谓的一等公民，就是指函数可以像其他数据类型一样作为函数的参数传入，也可以通过返回值的形式来返回。这里要介绍的回调(callback)就是利用了这一特性，我们将传递给另一个函数作为实参的函数称之为回调函数(callback)。

5-4-1 回调函数基本介绍

所谓回调函数，通俗的来讲，就是指将一个函数作为参数传递给另外一个函数，然后在另外一个函数里面执行传递过去的函数，我们来看一个具体的示例，如下：

```
let test = function(fn){
    fn()
}
let test2 = function(){
    console.log("Hello World");
}
test(test2);//Hello World
```

这里，我们的test2就被称之为回调函数。因为test2是作为一个参数传递到了test函数里面，然后在test里面进行了test2的函数调用。

回调函数可以和其他参数一起传入到一个参数里面，如下：

```
let test = function(name,fn){
    console.log(`My name is ${name}`);
    fn();
}
let test2 = function(){
    console.log("I'm coding");
}
test("xiejie",test2);
// My name is xiejie
// I'm coding
```

5-4-2 常见回调函数介绍

实际上回调函数我们在之前的学习中就已经见到过了。就在我们使用sort()为数组进行排序的时

候，默认是使用的ASCII码来进行的排序。如果想要按照数值来进行排序，就需要我们传递一个回调函数进去。这里我们可以来复习一下：

```
let arr = [0,12,3,7,-12,23];
console.log(arr.sort(function(a,b){
    return a - b;
    //降序就返回 b - a
}));
```

甚至我们还可以使用前面小节所介绍过的箭头函数，将上面的排序写作如下：

```
let arr = [0,12,3,7,-12,23];
console.log(arr.sort((a,b) => a - b));
```

在JavaScript里面，除了上面所介绍的sort()以外，还有诸如forEach()，map()，every()，some()等函数，也是所常见的回调函数。

迭代方法

every()：对数组的每一项运行给定的函数，如果该函数每一项都返回true，则返回true

```
let arr = [1,2,3,4,5,6,7,8,9,10];
//将数组的每一项传入到回调函数，如果每一项返回true，那么最终返回true
let i = arr.every(function(item){
    if(item % 2 == 0)
    {
        return true;
    }
    else{
        return false;
    }
});
console.log(i);//false
```

与every()比较相似的是some()，该方法可以对数组的每一项运行指定的函数，如果该函数只要有一项返回true则返回true

```
let arr = [1,2,3,4,5,6,7,8,9,10];
//将数组的每一项传入到回调函数，如果有一项返回true，那么最终返回true
let i = arr.some(function(item){
    if(item % 2 == 0)
    {
```

```
        return true;
    }
    else{
        return false;
    }
});
console.log(i);//true
```

filter()：filter是过滤的意思，所以这个方法会返回一个数组，数组里面是返回true的元素

```
let arr = [1,2,3,4,5,6,7,8,9,10];
//将数组的每一项传入到回调函数，然后将返回为true的项目组成一个数组
let i = arr.filter(function(item){
    if(item % 2 == 0)
    {
        return true;
    }
    else{
        return false;
    }
});
console.log(i);//[ 2, 4, 6, 8, 10 ]
```

forEach()：这个就是简单的将数组每一项传入函数，然后执行该函数，无返回值

```
let arr = [1,2,3,4,5,6,7,8,9,10];
//将数组的每一项传入到回调函数，然后执行回调函数里面的操作
let i = arr.forEach(function(item){
    console.log(item);
});
console.log(i);//undefined
```

map()：对数组的每一项运行test函数，返回一个数组，这个数组是每次调用函数后的运行结果

```
let arr = [1,2,3,4,5,6,7,8,9,10];
//将数组的每一项传入到回调函数，然后将返回的结果组成一个新的数组返回
let i = arr.map(function(item){
    if(item % 2 == 0)
    {
        return true;
    }
    else{
        return false;
    }
```

```
});
console.log(i);
//[ false, true, false, true, false, true, false, true, false, true ]
```

> 注意：以上方法都不会改变原数组的值，并且都可以接收两个参数，第一个是数组的元素值，第二个是数组的索引。上面的例子中我们都只接收了一个参数，即数组的值。

归并方法

归并方法有两个，reduce()和reduceRight()，一个是从数组第一项开始，另外一个是从数组最后一项开始，两个方法都会迭代数组所有的项，然后构建一个最终的返回值。
这两个方法都接受两个参数：一个在每一项调用的函数和一个可选的初始值。
关于调用的函数，里面又是接收4个参数的(前一项值，当前值，数组索引，数组对象)需要注意的是：这里的前一项值指的是上一次迭代时的计算结果

这里我们可以将回调函数的参数打印出来看一下，如下：

没有初始值的情况：

```
let arr = [1,2,3,4,5];
let i = arr.reduce(function(pre,cur,index,arr){
    console.log(pre,cur,index,arr);
    return pre + cur
});
console.log(i);
// 1 2 1 [ 1, 2, 3, 4, 5 ]
// 3 3 2 [ 1, 2, 3, 4, 5 ]
// 6 4 3 [ 1, 2, 3, 4, 5 ]
// 10 5 4 [ 1, 2, 3, 4, 5 ]
// 15
```

有初始值的情况：

```
let arr = [1,2,3,4,5];
let i = arr.reduce(function(pre,cur,index,arr){
    console.log(pre,cur,index,arr);
    return pre + cur
},100);
console.log(i);
// 100 1 0 [ 1, 2, 3, 4, 5 ]
// 101 2 1 [ 1, 2, 3, 4, 5 ]
// 103 3 2 [ 1, 2, 3, 4, 5 ]
// 106 4 3 [ 1, 2, 3, 4, 5 ]
// 110 5 4 [ 1, 2, 3, 4, 5 ]
```

```
// 115
```

其实，reduce就和前面的forEach，map这些很像，将数组的每一项应用到函数里面。只不过会将每次的结果累加起来放入下一次来进行使用
使用reduce()方法来实现数值的累加，如下：

```
let arr = [1,2,3,4,5,6,7,8,9,10];
let i = arr.reduce(function(pre,cur){
    return pre + cur;
});
console.log(i);//55
```

reduce()是从左往右进行归并，reduceRight()是从右往左开始归并，这里就不再做演示了。

5-4-3 链式调用

本节的最后介绍一下链式调用。所谓链式调用，就是可以像链条一样一直调用方法。其实链式调用的原理也非常简单，在调用方法时，方法里面会返回一个对象，然后这个对象又可以调用方法，这样我们就可以实现链式调用。

示例：求数组的偶数和

```
let arr = [1,2,3,4,5,6,7,8,9,10];
let evenNum = function(item){
    if(item % 2 == 0)
    {
        return true;
    }
}
let addNum = function(pre,cur){
    return pre + cur;
}
console.log(arr.filter(evenNum).reduce(addNum));//30
//因为filter()返回的是一个数组，所以我们可以直接再次调用reduce()方法
```

总结

1. 函数最大的好处就是实现了对代码的复用。
2. 在JavaScript中声明一个函数的方式有多种，例如字面量，函数表达式以及构造器等方式都可以声明一个函数。

3. 函数最基本的就是要掌握函数三要素，即函数功能，函数参数和函数返回值。
4. 在JavaScript的函数里面存在一个叫做arguments的伪数组对象，里面存储了调用函数时传递的所有实际参数。
5. 从ES6开始新增加了不定参数和默认参数
6. JavaScript里面的函数也具有属性和方法。
7. ES6新增加了一种声明函数的方式，叫做箭头函数，其最大的特点在于不需要function关键字，更加的简洁。
8. JavaScript里面作用域一共有3种：全局作用域，函数作用域以及eval()作用域
9. 变量提升，就是指在使用var关键字进行变量声明的时候，默认会将声明变量的部分提升至当前作用域的最顶上。
10. 函数提升，是指当我们使用字面量方式声明一个函数的时候，此函数的声明会提升到当前作用域的最顶端。
11. 回掉函数是指将一个函数作为参数传递给另外一个函数，然后在另外一个函数里面执行传递过去的函数。