函数

1. 函数的属性和方法

1.1. name 属性

函数的 name 属性返回函数的名字。

```
function f1() {}
f1.name; // 'f1'
```

如果是通过变量赋值定义的函数，那么 name 属性返回变量名。

```
let f2 = function () {};
f2.name; // "f2"
```

只有在变量的值是一个匿名函数时才是如此。如果变量的值是一个具名函数，那么 name 属性返回 function 关键字之后的那个函数名。

```
let f3 = function myName() {};
f3.name; // 'myName'
```

`f3.name` 返回函数表达式的名字。注意，真正的函数名还是 `f3`，而 `myName` 这个名字只在函数体内部可用。

name 属性的一个用处，就是获取参数函数的名字。

```
let myFunc = function () {};

function test(f) {
  console.log(f.name);
}

test(myFunc); // myFunc
```

1.2. length 属性

函数的 length 属性返回函数预期传入的参数个数，即函数定义之中的参数个数。

```
function f(a, b) {}
f.length; // 2
```

```
function fun2(...c) {}
fun2.length; // 0
```

2. 函数的传递方式

函数参数如果是原始类型的值（数值、字符串、布尔值），传递方式是传值传递（passes by value）。这意味着，在函数体内修改参数值，不会影响到函数外部。

```
let p = 2;

function f(p) {
  p = 3;
}
f(p);

p; // 2
```

但是，如果函数参数是复合类型的值（数组、对象、其他函数），传递方式是传址传递（pass by reference）。也就是说，传入函数的原始值的地址，因此在函数内部修改参数，将会影响到原始值。

```
let obj = { p: 1 };

function f(o) {
  o.p = 2;
}
f(obj);

obj.p; // 2
```

如果函数内部修改的，不是参数对象的某个属性，而是替换掉整个参数，这时不会影响到原始值。

```
let obj = [1, 2, 3];

function f(o) {
  o = [2, 3, 4];
}
f(obj);

obj; // [1, 2, 3]
```

形式参数（o）的值实际是参数 obj 的地址， **重新对 o 赋值导致 o 指向另一个地址** ，保存在原地址上的值当然不受影响。

2.1. 同名参数

如果有同名的参数，则取最后出现的那个值。即使后面的 a 没有值或被省略，也是以其为准。

```
function f1(a, a) {
  console.log(a);
}

f1(1, 2); // 2

function f2(a, a) {
  console.log(a);
}

f2(1); // undefined
```

这时，如果要获得第一个 a 的值，可以使用 arguments 对象。

```
function f(a, a) {
  console.log(arguments[0]);
}

f(1); // 1
```

3. 闭包

闭包就是函数。

JavaScript 有两种作用域（ES6 新增块级作用域），全局作用域和函数作用域，函数内部可以直接读取全局变量。

```
let n = 999;
function f1() {
  console.log(n);
}
f1() // 999
```

f1 函数读取了全局变量 n。

但是，正常情况下，函数外部无法读取函数内部申明的变量。

```
function f1() {
  let n = 999;
}
console.log(n);
// Uncaught ReferenceError: n is not defined
```

如果在函数内部再定义一个函数。

```
function f1() {
  let n = 999;
  function f2() {
    console.log(n); // 999
  }
}
```

函数 f2 就在函数 f1 内部，这时 f1 内部的所有局部变量，对 f2 都是可见的。这就是 JavaScript 语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然 f2 可以读取 f1 的局部变量，那么只要把 f2 作为返回值，在 f1 外部就可以读取它的内部变量了！

```
function f1() {
  let n = 999;
  function f2() {
    console.log(n);
  }
  return f2;
}

let result = f1();
result(); // 999
```

闭包就是函数 f2，**即能够读取其他函数内部变量的函数。** 或者说 **定义在一个函数内部的函数**

3.1. 闭包的用处

> - 读取外层函数内部变量
> - 让这些闭包始终保持在内存中

```
function createIncrementor(start) {
  return function () {
    return start++;
  };
}

let inc = createIncrementor(5);

inc(); // 5
inc(); // 6
inc(); // 7
```

`start` 是函数 `createIncrementor` 的内部变量。通过闭包，`start` 的状态被保留了，每一次调用都是在上一次调用的基础上进行计算。从中可以看到，闭包 `inc` 使得函数 `createIncrementor` 的内部环境一直存在。所以，闭包可以看作是函数内部作用域的一个接口。

为什么闭包能够返回外层函数的内部变量？原因是闭包（上例的`inc`）用到了外层变量（`start`），导致外层函数（`createIncrementor`）不能从内存释放。 **只要闭包没有被垃圾回收机制清除，外层函数提供的运行环境也不会被清除，它的内部变量就始终保存着当前值，供闭包读取。**

外层函数每次运行，都会生成一个新的闭包，而这个闭包又会保留外层函数的内部变量，所以内存消耗很大。因此不能滥用闭包，否则会造成网页的性能问题。

4. 立即调用的函数表达式（IIFE）

使用圆括号直接调用函数，有两种写法:

```
(function () {
  /* code */
}());
// 或者
(function () {
  /* code */
})();
```

上面两种写法最后的分号都是必须的。如果省略分号，遇到连着两个 `IIFE`，可能就会报错。引擎会认为后面跟的是一个参数。

通常情况下，只对匿名函数使用这种“立即执行的函数表达式”。

它的目的有两个：

> - 不必为函数命名，避免了污染全局变量；
> - `IIFE` 内部形成了一个单独的作用域，可以封装一些外部无法读取的私有变量。