函数（上）

- 我们代码里面所说的函数和我们上学的时候学习的什么三角函数、二次函数之类的不是一个东西

函数的概念

- 对于 js 来说，函数就是把任意一段代码放在一个 **盒子** 里面
- 在我想要让这段代码执行的时候，直接执行这个 **盒子** 里面的代码就行
- 先看一段代码

```
// 这个是我们以前写的一段代码
for (var i = 0; i < 10; i++) {
 console.log(i)
}

// 函数，这个 {} 就是那个 “盒子”
function fn() {
 // 这个函数我们以前写的代码
 for (var i = 0; i < 10; i++) {
 console.log(i)
 }
}
```

函数的两个阶段（重点）

- 按照我们刚才的说法，两个阶段就是 **放在盒子里面** 和 **让盒子里面的代码执行**

函数定义阶段

- 定义阶段就是我们把代码 **放在盒子里面**
- 我们就要学习怎么 **放进去**，也就是书写一个函数
- 我们有两种定义方式 **声明式** 和 **赋值式**

声明式

- 使用 `function` 这个关键字来声明一个函数

- 使用 `function` 这个关键字来声明一个函数

- 语法：

```
function fn() {
 // 一段代码
}
// function：声明函数的关键字，表示接下来是一个函数了
// fn：函数的名字，我们自己定义的（遵循变量名的命名规则和命名规范）
// ()：必须写，是用来放参数的位置（一会我们再聊）
// {}：就是我们用来放一段代码的位置（也就是我们刚才说的 “盒子”）
```

赋值式

- 其实就是和我们使用 `var` 关键字是一个道理了
- 首先使用 `var` 定义一个变量，把一个函数当作值直接赋值给这个变量就可以了
- 语法：

```
var fn = function () {
 // 一段代码
}
// 不需要在 function 后面书写函数的名字了，因为在前面已经有了
```

函数调用阶段

- 就是让 **盒子里面** 的代码执行一下
- 让函数执行
- 两种定义函数的方式不同，但是调用函数的方式都以一样的

调用一个函数

- 函数调用就是直接写 `函数名()` 就可以了

```
// 声明式函数
function fn() {
  console.log('我是 fn 函数')
}

// 调用函数
fn()

// 赋值式函数
```

```
// 赋值式函数
var fn2 = function () {
  console.log('我是 fn2 函数')
}

// 调用函数
fn()
```

- 注意： **定义完一个函数以后，如果没有函数调用，那么写在 {} 里面的代码没有意义，只有调用以后才会执行**

调用上的区别

- 虽然两种定义方式的调用都是一样的，但是还是有一些区别的
- 声明式函数： 调用可以在 **定义之前或者定义之后**

```
// 可以调用
fn()

// 声明式函数
function fn() {
  console.log('我是 fn 函数')
}

// 可以调用
fn()
```

- 赋值式函数： 调用只能在 **定义之前**

```
// 会报错
fn()

// 赋值式函数
var fn = function () {
  console.log('我是 fn 函数')
}

// 可以调用
fn()
```

函数的参数（重点）

- 我们在定义函数和调用函数的时候都出现过 `()`
- 现在我们就来说一下这个 `()` 的作用
- 就是用来放参数的位置
- 参数分为两种 **行参** 和 **实参**

```
// 声明式
function fn(行参写在这里) {
 // 一段代码
}

fn(实参写在这里)

// 赋值式函数
var fn = function (行参写在这里) {
 // 一段代码
}
fn(实参写在这里)
```

行参和实参的作用

1. 行参
    - 就是在函数内部可以使用的变量，在函数外部不能使用
    - 每写一个单词，就相当于在函数内部定义了一个可以使用的变量（遵循变量名的命名规则和命名规范）
    - 多个单词之间以 `,` 分隔

```
// 书写一个参数
function fn(num) {
 // 在函数内部就可以使用 num 这个变量
}

var fn1 = function (num) {
 // 在函数内部就可以使用 num 这个变量
}

// 书写两个参数
function fun(num1, num2) {
```

```
 // 在函数内部就可以使用 num1 和 num2 这两个变量
}

var fun1 = function (num1, num2) {
 // 在函数内部就可以使用 num1 和 num2 这两个变量
}
```

- 如果只有行参的话，那么在函数内部使用的值个变量是没有值的，也就是 `undefined`
- **行参的值是在函数调用的时候由实参决定的**

2. 实参

- 在函数调用的时候给行参赋值的
- 也就是说，在调用的时候是给一个实际的内容的

```
function fn(num) {
 // 函数内部可以使用 num
}

// 这个函数的本次调用，书写的实参是 100
// 那么本次调用的时候函数内部的 num 就是 100
fn(100)

// 这个函数的本次调用，书写的实参是 200
// 那么本次调用的时候函数内部的 num 就是 200
fn(200)
```

- **函数内部的行参的值，由函数调用的时候传递的实参决定**
- **多个参数的时候，是按照顺序一一对应的**

```
function fn(num1, num2) {
 // 函数内部可以使用 num1 和 num2
}

// 函数本次调用的时候，书写的参数是 100 和 200
// 那么本次调用的时候，函数内部的 num1 就是 100，num2 就是 200
fn(100, 200)
```

参数个数的关系

1. 行参比实参少

1. 行参比实参少

   - 因为是按照顺序一一对应的
   - 行参少就会拿不到实参给的值，所以在函数内部就没有办法用到这个值

```
function fn(num1, num2) {
 // 函数内部可以使用 num1 和 num2
}

// 本次调用的时候，传递了两个实参，100 200 和 300
// 100 对应了 num1，200 对应了 num2，300 没有对应的变量
// 所以在函数内部就没有办法依靠变量来使用 300 这个值
fn(100, 200, 300)
```

1. 行参比实参多

   - 因为是按照顺序一一对应的
   - 所以多出来的行参就是没有值的，就是 `undefined`

```
function fn(num1, num2, num3) {
 // 函数内部可以使用 num1 num2 和 num3
}

// 本次调用的时候，传递了两个实参，100 和 200
// 就分别对应了 num1 和 num2
// 而 num3 没有实参和其对应，那么 num3 的值就是 undefined
fn(100, 200)
```

函数的return（重点）

- return 返回的意思，其实就是给函数一个 **返回值** 和 **终断函数**

终断函数

- 当我开始执行函数以后，函数内部的代码就会从上到下的依次执行
- 必须要等到函数内的代码执行完毕
- 而 `return` 关键字就是可以在函数中间的位置停掉，让后面的代码不在继续执行

```
function fn() {
```

```
  console.log(1)
  console.log(2)
  console.log(3)

  // 写了 return 以后，后面的 4 和 5 就不会继续执行了
  return
  console.log(4)
  console.log(5)
}

// 函数调用
fn()
```

返回值

- 函数调用本身也是一个表达式，表达式就应该有一个值出现
- 现在的函数执行完毕之后，是不会有结果出现的

```
// 比如 1 + 2 是一个表达式，那么 这个表达式的结果就是 3
console.log(1 + 2) // 3

function fn() {
  // 执行代码
}

// fn() 也是一个表达式，这个表达式就没有结果出现
console.log(fn()) // undefined
```

- `return` 关键字就是可以给函数执行完毕一个结果

```
function fn() {
  // 执行代码
  return 100
}

// 此时，fn() 这个表达式执行完毕之后就有结果出现了
console.log(fn()) // 100
```

  - 我们可以在函数内部使用 `return` 关键把任何内容当作这个函数运行后的结果

函数的优点

- 函数就是对一段代码的封装，在我们想调用的时候调用
- 函数的几个优点
  i. 封装代码，使代码更加简洁
  ii. 复用，在重复功能的时候直接调用就好
  iii. 代码执行时机，随时可以在我们想要执行的时候执行

预解析（重点）

- **预解析** 其实就是聊聊 js 代码的编译和执行
- js 是一个解释型语言，就是在代码执行之前，先对代码进行通读和解释，然后在执行代码
- 也就是说，我们的 js 代码在运行的时候，会经历两个环节 **解释代码** 和 **执行代码**

解释代码

- 因为是在所有代码执行之前进行解释，所以叫做 **预解析（预解释）**

- 需要解释的内容有两个

  - 声明式函数
    - 在内存中先声明有一个变量名是函数名，并且这个名字代表的内容是一个函数
  - `var` 关键字
    - 在内存中先声明有一个变量名

- 看下面一段代码

```
fn()
console.log(num)

function fn() {
 console.log('我是 fn 函数')
}

var num = 100
```

- 经过预解析之后可以变形为

```
function fn() {
 console.log('我是 fn 函数')
}
var num

fn()
```

```
fn()
console.log(num)
num = 100
```

- 赋值是函数会按照 `var` 关键字的规则进行预解析

函数（下）

作用域（重点）

- 什么是作用域，就是一个变量可以生效的范围
- 变量不是在所有地方都可以使用的，而这个变量的使用范围就是作用域

全局作用域

- 全局作用域是最大的作用域
- 在全局作用域中定义的变量可以在任何地方使用
- 页面打开的时候，浏览器会自动给我们生成一个全局作用域 window
- 这个作用域会一直存在，直到页面关闭就销毁了

```
// 下面两个变量都是存在在全局作用域下面的，都是可以在任意地方使用的
var num = 100
var num2 = 200
```

局部作用域

- 局部作用域就是在全局作用域下面有开辟出来的一个相对小一些的作用域
- 在局部作用域中定义的变量只能在这个局部作用域内部使用
- **在 JS 中只有函数能生成一个局部作用域，别的都不行**
- 每一个函数，都是一个局部作用域

```
// 这个 num 是一个全局作用域下的变量 在任何地方都可以使用
var num = 100

function fn() {
  // 下面这个变量就是一个 fn 局部作用域内部的变量
```

```
  // 下面这个变量就是一个 fn 局部作用域内部的变量
  // 只能在 fn 函数内部使用
  var num2 = 200
}

fn()
```

变量使用规则（重点）

- 有了作用域以后，变量就有了使用范围，也就有了使用规则
- 变量使用规则分为两种，**访问规则** 和 **赋值规则**

访问规则

- 当我想获取一个变量的值的时候，我们管这个行为叫做 **访问**
- 获取变量的规则：
  - 首先，在自己的作用域内部查找，如果有，就直接拿来使用
  - 如果没有，就去上一级作用域查找，如果有，就拿来使用
  - 如果没有，就继续去上一级作用域查找，依次类推
  - 如果一直到全局作用域都没有这个变量，那么就会直接报错（该变量 is not defined）

```
var num = 100

function fn() {
  var num2 = 200

  function fun() {
  var num3 = 300

  console.log(num3) // 自己作用域内有，拿过来用
  console.log(num2) // 自己作用域内没有，就去上一级，就是 fn 的作用域里面找，发现有，拿过来用
  console.log(num) // 自己这没有，去上一级 fn 那里也没有，再上一级到全局作用域，发现有，直接用
  console.log(a) // 自己没有，一级一级找上去到全局都没有，就会报错
  }

  fun()
}

```

```
fn()
```

- 变量的访问规则 也叫做 作用域的查找机制
- 作用域的查找机制只能是向上找，不能向下找

```
function fn() {
 var num = 100
}
fn()

console.log(num) // 发现自己作用域没有，自己就是全局作用域，没有再上一级了，直接报错
```

赋值规则

- 当你想给一个变量赋值的时候，那么就先要找到这个变量，在给他赋值
- 变量赋值规则：
  - 先在自己作用域内部查找，有就直接赋值
  - 没有就去上一级作用域内部查找，有就直接赋值
  - 在没有再去上一级作用域查找，有就直接赋值
  - 如果一直找到全局作用域都没有，那么就把这个变量定义为全局变量，在给他赋值

```
function fn() {
 num = 100
}
fn()

// fn 调用以后，要给 num 赋值
// 查看自己的作用域内部没有 num 变量
// 就会向上一级查找
// 上一级就是全局作用域，发现依旧没有
// 那么就会把 num 定义为全局的变量，并为其赋值
// 所以 fn() 以后，全局就有了一个变量叫做 num 并且值是 100
console.log(num) // 100
```

递归函数

- 什么是递归函数
- 在编程世界里面，递归就是一个自己调用自己的手段

- 递归函数： 一个函数内部，调用了自己，循环往复

```
// 下面这个代码就是一个最简单的递归函数
// 在函数内部调用了自己，函数一执行，就调用自己一次，在调用再执行，循环往复，没有止尽
function fn() {
 fn()
}
fn()
```

- 其实递归函数和循环很类似
- 需要有初始化，自增，执行代码，条件判断的，不然就是一个没有尽头的递归函数，我们叫做 **死递归**

简单实现一个递归

- 我们先在用递归函数简单实现一个效果
- 需求： 求 1 至 5 的和
  - 先算 1 + 2 得 3
  - 再算 3 + 3 得 6
  - 再算 6 + 4 得 10
  - 再算 10 + 5 得 15
  - 结束
- 开始书写，写递归函数先要写结束条件（为了避免出现 “死递归”）

```
function add(n) {
 // 传递进来的是 1
 // 当 n === 5 的时候要结束
 if (n === 5) {
 return 5
 }
}

add(1)
```

- 再写不满足条件的时候我们的递归处理

```
function add(n) {
 // 传递进来的是 1
```

```
  // 当 n === 5 的时候要结束
  if (n === 5) {
  return 5
  } else {
  // 不满足条件的时候，就是当前数字 + 比自己大 1 的数字
  return n + add(n + 1)
  }
}
add(1)
```

简单了解对象

- 对象是一个复杂数据类型
- 其实说是复杂，但是没有很复杂，只不过是存储了一些基本数据类型的一个集合

```
var obj = {
 num: 100,
 str: 'hello world',
 boo: true
}
```

- 这里的 `{}` 和函数中的 `{}` 不一样
- 函数里面的是写代码的，而对象里面是写一些数据的
- **对象就是一个键值对的集合**
- `{}` 里面的每一个键都是一个成员
- 也就是说，我们可以把一些数据放在一个对象里面，那么他们就互不干扰了
- 其实就是我们准备一个房子，把我们想要的数据放进去，然后把房子的地址给到变量名，当我们需要某一个数据的时候，就可以根据变量名里面存储的地址找到对应的房子，然后去房子里面找到对应的数据

创建一个对象

- 字面量的方式创建一个对象

```
// 创建一个空对象
var obj = {}

// 像对象中添加成员
```

```
// 向对象中添加成员
obj.name = 'Jack'
obj.age = 18
```

- 内置构造函数的方式创建对象

```
// 创建一个空对象
var obj = new Object()

// 向对象中添加成员
obj.name = 'Rose'
obj.age = 20
```

  - Object 是 js 内置给我们的构造函数，用于创建一个对象使用的