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

函数的概念及作用

乔帮主有很多武功招式，每个武功招式都有不一样的效果

降龙十八掌就是一个强大的武功招式，集结了十八条龙的能量，毁天灭地...

总结：不同的武功招式，聚集了不同的能量需要的时候，通过某种固定的方式来释放武功招式

- 通俗讲：函数就是可重复执行的代码块。
- 函数的作用：
  - 1.通过函数可以封装任意多条语句，以便在任何地方、任何时候调用；
  - 2.将代码编写在函数中，就可以避免在非必要情况下调用该代码。
- 回顾上面的故事情节，函数 替换 武功招式，调用 替换 释放.....

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
- 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
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 **定义之后

```
// 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
    - 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)
```

2. 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)
```

arguments

`arguments`对象是所有（非箭头）函数中都可用的**局部变量**。你可以使用`arguments`对象在函数中引用函数的参数。此对象包含传递给函数的每个参数，第一个参数在索引0处。例如，如果一个函数传递了三个参数，你可以以如下方式引用他们：

```
arguments[0]
arguments[1]
arguments[2]
//参数也可以被设置：
arguments[1] = 'new value';
```

`arguments`对象不是一个 `Array` 。它类似于`Array`，但除了length属性和索引元素之外没有任何`Array`属性。

可以借用arguments.length可以来查看实参和形参的个数是否一致

```
function add(a, b) {
  var realLen = arguments.length;
  console.log("realLen:", arguments.length);
  var len = add.length;
  console.log("len:", add.length);
  if (realLen == len) {
    console.log('实参和形参个数一致');
  } else {
    console.log('实参和形参个数不一致');
  }
};
add(1,2,3,6,8);
```

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
  1. 封装代码，使代码更加简洁
  2. 复用，在重复功能的时候直接调用就好
  3. 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()
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