前言

关于函数的核心内容：

- 函数有哪几种定义和调用方式
- this：函数内部的 this 指向、如何改变 this 的指向。
- 函数的严格模式
- 高阶函数：函数作为参数传递、函数作为返回值传递
- 闭包：闭包的作用
- 递归：递归的两个条件
- 深拷贝和浅拷贝的区别

函数的介绍

函数：就是将一些功能或语句进行**封装**，在需要的时候，通过**调用**的形式，执行这些语句。

- **函数也是一个对象**
- 使用`typeof`检查一个函数对象时，会返回function

函数的作用：

- 将大量重复的语句抽取出来，写在函数里，以后需要这些语句的时候，可以直接调用函数，避免重复劳动。
- 简化编程，让编程模块化。高内聚、低耦合。

来看个例子：

```
console.log("你好");
sayHello(); // 调用函数

// 定义函数
function sayHello(){
    console.log("欢迎");
    console.log("welcome");
}
```

函数的定义/声明

方式一：利用函数关键字自定义函数（命名函数）

使用`函数声明`来创建一个函数（也就是 function 关键字）。语法：

```
function 函数名([形参1,形参2...形参N]){  // 备注：语法中的中括号，表示“可选”
    语句...
}
```

举例：

```
function fun1(a, b){
    return a+b;
}
```

解释如下：

- function：是一个关键字。中文是“函数”、“功能”。
- 函数名字：命名规定和变量的命名规定一样。只能是字母、数字、下划线、美元符号，不能以数字开头。
- 参数：可选。
- 大括号里面，是这个函数的语句。

PS：在有些编辑器中，方法写完之后，我们在方法的前面输入`/**`，然后回车，会发现，注释的格式会自动补齐。

方式二：函数表达式（匿名函数）

使用`函数表达式`来创建一个函数。语法：

```
var 变量名  = function([形参1,形参2...形参N]){
    语句....
}
```

举例：

```
var fun2 = function() {
    console.log("我是匿名函数中封装的代码");
};
```

解释如下：

- 上面的 fun2 是变量名，不是函数名。
- 函数表达式的声明方式跟声明变量类似，只不过变量里面存的是值，而函数表达式里面存的是函数。
- 函数表达式也可以传递参数。

从方式二的举例中可以看出：所谓的“函数表达式”，其实就是将匿名函数赋值给一个变量。

方式三：使用构造函数 new Function()

使用构造函数`new Function()`来创建一个对象。这种方式，用的少。

语法：

```
var 变量名/函数名  = new Function('形参1', '形参2', '函数体');
```

注意，Function 里面的参数都必须是**字符串**格式。也就是说，形参也必须放在**字符串**里；函数体也是放在**字符串**里包裹起来，放在 Function 的最后一个参数的位置。

代码举例：

```
var fun3 = new Function('a', 'b', 'console.log("我是函数内部的内容");
console.log(a + b);');

fun3(1, 2); // 调用函数
```

打印结果：

```
我是函数内部的内容
3
```

分析：

正常人不用这种形式，不需要记住，原因如下：

- 不方便书写：写法过于啰嗦和麻烦。
- 执行效率较低：首先需要把字符串转换为 js 代码，然后再执行。

总结

1、**所有的函数，都是 `Fuction` 的"实例"**（或者说是"实例对象"）。函数本质上都是通过 new Function 得到的。

2、函数既然是实例对象，那么，**函数也属于"对象"**。还可以通过如下特征，来佐证函数属于对象：

（1）我们直接打印某一个函数，比如 `console.log(fun2)`，发现它的里面有 `__proto__`。（这个是属于原型的知识，后续再讲）

（2）我们还可以打印 `console.log(fun2 instanceof Object)`，发现打印结果为 `true`。这说明 fun2 函数就是属于 Object。

函数的调用

方式1：普通函数的调用

函数调用的语法：

```
函数名();
```

或者：

```
函数名.call();
```

代码举例：

```
function fn1() {
    console.log('我是函数体里面的内容1');
}

function fn2() {
    console.log('我是函数体里面的内容2');
}

fn1(); // 调用函数

fn2.call(); // 调用函数

```

方式2：通过对象的方法来调用

```
var obj = {
    a: 'qianguyihao',
    fn2: function() {
        console.log('千古壹号，永不止步!');
    },
};

obj.fn2(); // 调用函数
```

如果一个函数是作为一个对象的属性保存，那么，我们称这个函数是这个对象的**方法**。

PS：关于函数和方法的区别，本文的后续内容里有讲到，可以往下面翻。

方式3：立即执行函数

代码举例：

```
(function() {
    console.log('我是立即执行函数');
})();

```

立即执行函数在定义后，会自动调用。

PS：关于立即执行函数，本文的后续内容里有讲到，可以往下面翻。

上面讲到的这三种方式，是用得最多的。接下来讲到的三种方式，暂时看不懂也没关系，可以等学完其他的知识点，再回过头来看。

方式4：通过构造函数来调用

代码举例：

```
function Fun3() {
    console.log('千古壹号，永不止步~');
}

new Fun3();
```

这种方式用得不多。

方式5：绑定事件函数

代码举例：

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
        <title>Document</title>
    </head>
    <body>
        <div id="btn">我是按钮，请点击我</div>

```

```
        <script>
            var btn = document.getElementById('btn');
            //2.绑定事件
            btn.onclick = function() {
                console.log('点击按钮后，要做的事情');
            };
        </script>
    </body>
</html>

```

这里涉及到DOM操作和事件的知识点，后续再讲。

方式6：定时器函数

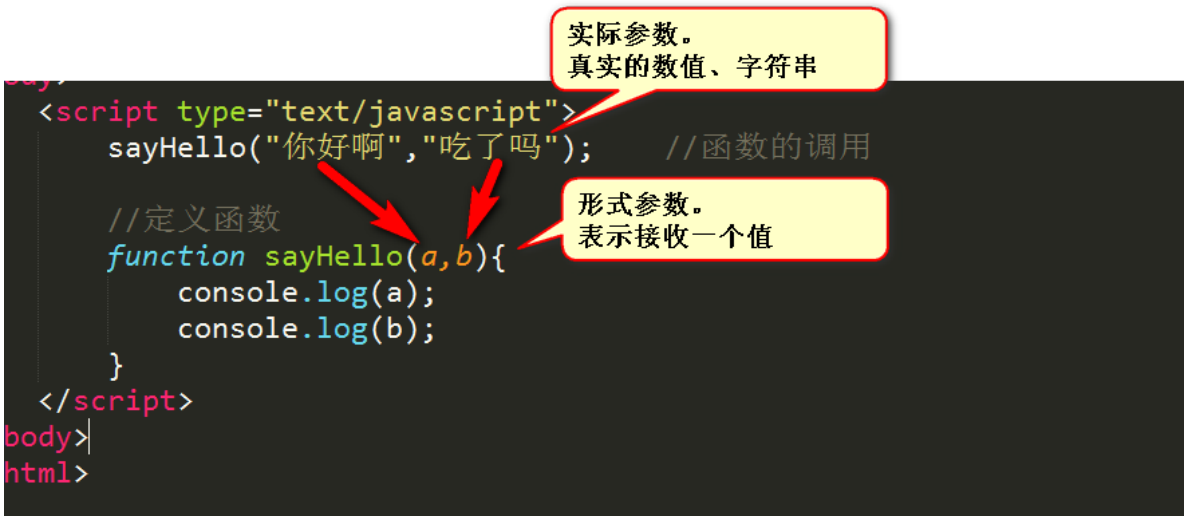
代码举例：（每间隔一秒，将 数字 加1 ）

```
 let num = 1;
setInterval(function () {
    num ++;
    console.log(num);
}, 1000);
```

这里涉及到定时器的知识点。

函数的参数：形参和实参

函数的参数包括形参和实参。先来看下面的图就很好懂了：



形参：

- 概念：形式上的参数。定义函数时传递的参数，当时并不知道是什么值。
- 定义函数时，可以在函数的`()`中来指定一个或多个形参。
- 多个形参之间使用`,`隔开，声明形参就相当于在函数内部声明了对应的变量，但是并不赋值。

实参：

- 概念：实际上的参数。调用函数时传递的参数，实参将会传递给函数中对应的形参。
- 在调用函数时，可以在函数的`()`中指定实参。

注意：实际参数和形式参数的个数，一般要相同。

举例：

```
// 调用函数
sum(3,4);
sum("3",4);
sum("Hello","World");

// 定义函数：求和
function sum(a, b) {
    console.log(a + b);
}
```

控制台输出结果：

```
7
34
helloworld
```

实参的类型

函数的实参可以是任意的数据类型。

调用函数时，解析器不会检查实参的类型，所以要注意，是否有可能会接收到非法的参数，如果有可能则需要对参数进行类型的检查。

实参的数量（实参和形参的个数不匹配时）

调用函数时，解析器也不会检查实参的数量。

- 如果实参的数量多余形参的数量，多余实参不会被赋值。
- 如果实参的数量少于形参的数量，多余的形参会被定义为 undefined。表达式的运行结果为 NaN。

代码举例：

```
    function sum(a, b) {
        console.log(a + b);
    }

    sum(1, 2);
    sum(1, 2, 3);
    sum(1);
```

打印结果：

```
3

3

NaN
```

注意：在 JS 中，形参的默认值是 undefined。

函数的返回值

举例：

```
console.log(sum(3, 4)); // 将函数的返回值打印出来

//函数：求和
function sum(a, b) {
    return a + b;
}
```

return 的作用是结束方法。

注意：

- return 的值将会作为函数的执行结果返回，可以定义一个变量，来接收该结果。
- 在函数中，return后的语句都不会执行（函数在执行完 return 语句之后停止并立即退出函数）
- 如果return语句后不跟任何值，就相当于返回一个undefined
- 如果函数中不写return，则也会返回undefined
- 返回值可以是任意的数据类型，可以是对象，也可以是函数。
- return 只能返回一个值。如果用逗号隔开多个值，则以最后一个为准。

函数名、函数体和函数加载问题（重要，请记住）

我们要记住：**函数名 == 整个函数**。举例：

```
console.log(fn) == console.log(function fn(){alert(1)});

//定义fn方法
function fn(){
    alert(1)
};
```

我们知道，当我们在调用一个函数时，通常使用`函数()`这种格式；可如果，我们是直接使用`函数`这种格式，它的作用相当于整个函数。

函数的加载问题：JS加载的时候，只加载函数名，不加载函数体。所以如果想使用内部的成员变量，需要调用函数。

fn() 和 fn 的区别【重要】

- `fn()`：调用函数。调用之后，还获取了函数的返回值。
- `fn`：函数对象。相当于直接获取了整个函数对象。

break、continue、return 的区别

- break ：结束当前的循环体（如 for、while）
- continue ：跳出本次循环，继续执行下次循环（如 for、while）
- return ：1、退出循环。2、返回 return 语句中的值，同时结束当前的函数体内的代码，退出当前函数。

立即执行函数

现有匿名函数如下：

```
    function(a, b) {
        console.log("a = " + a);
        console.log("b = " + b);
    };
```

立即执行函数如下：

```
    (function(a, b) {
        console.log("a = " + a);
        console.log("b = " + b);
    })(123, 456);
```

立即执行函数：函数定义完，立即被调用，这种函数叫做立即执行函数。

立即执行函数往往只会执行一次。为什么呢？因为没有变量保存它，执行完了之后，就找不到它了。

方法

函数也可以成为对象的属性。**如果一个函数是作为一个对象的属性保存，那么，我们称这个函数是这个对象的方法。**

调用这个函数就说调用对象的方法（method）。函数和方法，有什么本质的区别吗？它只是名称上的区别，并没有其他的区别。

函数举例：

```
    // 调用函数
    fn();
```

方法举例：

```
    // 调用方法
    obj.fn();
```

我们可以这样说，如果直接是`fn()`，那就说明是函数调用。如果是`xx.fn()`的这种形式，那就说明是**方法**调用。

arguments 的使用

当我们不确定有多少个参数传递的时候，可以用 **arguments** 来获取。在 JavaScript 中，arguments 实际上是当前函数的一个**内置对象**。所有函数都内置了一个 arguments 对象（只有函数才有 arguments 对象），arguments 对象中存储了**传递的所有实参**.

arguments的展示形式是一个**伪数组**。伪数组具有以下特点：

- 可以进行遍历；具有数组的 length 属性。
- 按索引方式存储数据。
- 不具有数组的 push()、pop() 等方法。

代码举例：利用 arguments 求函数实参中的最大值

代码实现：

```
    function getMaxValue() {
        var max = arguments[0];
        // 通过 arguments 遍历实参
        for (var i = 0; i < arguments.length; i++) {
            if (max < arguments[i]) {
                max = arguments[i];
            }
        }
        return max;
    }

    console.log(getMaxValue(1, 3, 7, 5));

```