函数与函数式编程

函数声明、函数表达式、匿名函数与自执行函数

函数在实际开发中的应用，大体可以总结为函数声明、函数表达式、匿名函数、自执行函数。

函数声明

`JavaScript`中，有两种声明方式，一个是使用`var`的变量声明，另一个是使用`function`的函数声明。

在变量对象的创建过程中，函数声明比变量声明具有更为优先的执行顺序，即函数声明提前。因此无论在执行上下文中哪个位置声明了函数，我们都可以在这个执行上下文中直接使用该函数。

```
fn();//function
function fn(){
    console.log('function');
}
```

函数表达式

与函数声明不同，函数表达式使用了`var`进行声明，那么我们必须依照`var`的规则进行判断如何正确使用，即变量声明。

```
var a = 20;// 变量声明

// 实际执行顺序
var a = undefined;  // 变量声明，初始值undefined，变量提升，提升顺序次于function声明
a = 20;  // 变量赋值，该操作不会提升
```

当我们使用函数表达式的时候，函数表达式的提升方式与变量声明一致。

```
fn(); // 报错
var fn = function() {
    console.log('function');
}
```

顺序应该为：

```
var fn = undefined;   // 变量声明提升
fn();    // 执行报错
fn = function() {  // 赋值操作，此时将后边函数的引用赋值给fn
    console.log('function');
}
```

函数表达式中的赋值操作，在其他一些地方也会被经常使用，我们清楚其中的关系即可。

```
function Person(name) {
    this.name = name;
    this.age = age;
    this.getAge = function() {// 在构造函数内部中添加方法
        return this.age;
    }
}

Person.prototype.getName = function() {// 给原型添加方法
    return this.name;
}

var a = {
    m: 20,
    getM: function() {// 在对象中添加方法
        return this.m;
    }
}
```

匿名函数

匿名函数，顾名思义，就是指的没有被显示进行赋值操作的函数。它的使用场景，多作为一个参数传入另一个函数中。

```
var a = 10;
var fn = function(bar, num) {
    return bar() + num;
}

fn(function() {
    return a;
}, 20)
```

`fn`的第一个参数传入了一个匿名函数。虽然该匿名函数没有显示的进行赋值操作，我们没有办法在外部执行上下文中引用到它，但是在`fn`函数内部，我们将该匿名函数赋值给了变量`bar`，保存在了`fn`变量对象的`arguments`对象中。

```
// 变量对象在fn上下文执行过程中的创建阶段
VO(fn) = {
    arguments: {
        bar: undefined,
        num: undefined,
        length: 2
    }
}

// 变量对象在fn上下文执行过程中的执行阶段
// 变量对象变为活动对象，并完成赋值操作与执行可执行代码
VO -> AO

AO(fn) = {
```

```
    arguments: {
        bar: function() { return a },
        num: 20,
        length: 2
    }
}
```

由于匿名函数传入另一个函数之后，最终会在另一个函数中执行，因此我们也常常称这个匿名函数为**回调函数。**

函数自执行与块级作用域

在`ES5`中，没有块级作用域，因此我们常常使用函数自执行的方式来模仿块级作用域，这样就提供了一个独立的执行上下文，结合闭包，就为模块化提供了基础。而**函数自执行，其实是匿名函数的一种应用。**

```
(function() {
   // ...
})();
```

一个模块往往可以包括：私有变量、私有方法、公有变量、公有方法。

根据作用域链的单向访问，在这个独立的模块中，外部执行环境是无法访问内部的任何变量与方法的，因此我们可以很容易的创建属于这个模块的私有变量与私有方法。

```
(function() {
    // 私有变量
    var age = 20;
    var name = 'Tom';

    // 私有方法
    function getName() {
        return `your name is ` + name;
    }
})();
```

利用闭包，我们可以访问到执行上下文内部的变量和方法，因此，我们只需要根据闭包的定义，创建一个闭包，将你认为需要公开的变量和方法开放出来即可。

```
(function() {
    // 私有变量
    var age = 20;
    var name = 'Tom';

    // 私有方法
    function getName() {
        return `your name is ` + name;
    }

    // 共有方法
    function getAge() {
        return age;
```

```
    }

    // 将引用保存在外部执行环境的变量中，形成闭包，防止该执行环境被垃圾回收
    window.getAge = getAge;
})();
```

我们来看看jQuery中，是如何利用我们模块与闭包的。

```
// 使用函数自执行的方式创建模块
(function(window, undefined) {
    // 声明jQuery构造函数
     var jQuery = function(name) {
        // 主动在构造函数中，返回一个jQuery实例
         return new jQuery.fn.init(name);
     }
    // 添加原型方法
     jQuery.prototype = jQuery.fn = {
         constructor: jQuery,
         init:function() { ... },
         css: function() { ... }
     }
     jQuery.fn.init.prototype = jQuery.fn;
    // 将jQuery改名为$，并将引用保存在window上，形成闭包，对外开发jQuery构造函数，这样我们就可以访问所有挂载在jQuery原型上的方法了
     window.jQuery = window.$ = jQuery;
 })(window);

// 在使用时，我们直接执行了构造函数，因为在jQuery的构造函数中通过一些手段，返回的是jQuery的实例，所以我们就不用再每次用的时候在自己new了
$('#div1');
```

当我们的项目越来越大，那么需要保存的数据与状态就越来越多，因此，我们需要一个专门的模块来维护这些数据，这个时候，有一个叫做状态管理器的东西就应运而生。对于状态管理器，最出名的，我想非`redux`莫属了。在我们学习之前，可以先通过简单的方式，让大家大致了解状态管理器的实现原理。

```
// 自执行创建模块
(function() {
    // states 结构预览
    // states = {
    //     a: 1,
    //     b: 2,
    //     m: 30,
    //     o: {}
    // }
    var states = {};  // 私有变量，用来存储状态与数据

    // 判断数据类型
    function type(elem) {
        if(elem == null) {
            return elem + '';
        }
```

```
        return toString.call(elem).replace(/[\[\]]/g, '').split(' ')[1].toLowerCase();
    }
    /**
     * @Param name 属性名
     * @Description 通过属性名获取保存在states中的值
    */
    function get(name) {
        return states[name] ? states[name] : '';
    }

    function getStates() {
        return states;
    }

    /*
    * @param options {object} 键值对
    * @param target {object} 属性值为对象的属性，只在函数实现时递归中传入
    * @desc 通过传入键值对的方式修改state树，使用方式与小程序的data或者react中的setStates类似
    */
    function set(options, target) {
        var keys = Object.keys(options);
        var o = target ? target : states;

        keys.map(function(item) {
            if(typeof o[item] == 'undefined') {
                o[item] = options[item];
            }
            else {
                type(o[item]) == 'object' ?
                    set(options[item], o[item]) : o[item] = options[item];
            }
            return item;
        })
    }

    // 对外提供接口
    window.get = get;
    window.set = set;
    window.getStates = getStates;
})()

// 具体使用如下

set({ a: 20 });     // 保存属性a
set({ b: 100 });    // 保存属性b
set({ c: 10 });     // 保存属性c

// 保存属性o，它的值为一个对象
set({
    o: {
        m: 10,
        n: 20
    }
```

```
})

// 修改对象o 的m值
set({
    o: {
        m: 1000
    }
})

// 给对象o中增加一个c属性
set({
    o: {
        c: 100
    }
})
console.log(getStates())
```

函数参数传递方式：按值传递

> **访问变量有按值和按引用两种方式，但参数只能按值传递 。**

- 按值传递(`call by value`)是最常用的求值策略：函数的形参是被调用时所传实参的*副本*。修改形参的值并不会影响实参。
- 按引用传递(`call by reference`)时，函数的形参接收实参的*隐式引用*，而不再是副本。这意味着函数形参的值如果被修改，实参也会被修改。同时两者指向相同的值。

基本数据类型复制，是直接值发生了复制，因此改变后，各自相互不影响。但是引用数据类型的复制，是保存在变量对象中的引用发生了复制，因此复制之后的这两个引用实际访问的实际是同一个堆内存中的值。当改变其中一个时，另外一个自然也被改变。

```
var a = 20;
var b = a;
b = 10;
console.log(a);  // 20

var m = { a: 1, b: 2 }
var n = m;
n.a = 5;
console.log(m.a) // 5
```

我们知道，函数的参数在进入函数后，**实际是被保存在了函数的变量对象中**，因此，这个时候**相当于发生了一次复制。**

```
var a = 20;

function fn(a) {
    a = a + 10;
    return a;
}
fn(a);
console.log(a); // 20
```

```
var a = { m: 10, n: 20 }

function fn(a) {
    a.m = 20;
    return a;
}
fn(a);
console.log(a);   // { m: 20, n: 20 }
```

正是由于这样的不同，导致了许多人在理解函数参数的传递方式时，就有许多困惑。到底是按值传递还是按引用传递？**实际上结论仍然是按值传递**，只不过当我们期望传递一个引用类型时，**真正传递的，只是这个引用类型保存在变量对象中的引用而已。**

```
var person = {
    name: 'Nicholas',
    age: 20
}

function setName(obj) {  // 传入一个引用
    obj = {};   // 将传入的引用指向另外的值
    obj.name = 'Greg';  // 修改引用的name属性
}

setName(person);
console.log(person.name);  // Nicholas 未被改变
```

如果`person`是按引用传递，那么`person`就会自动被修改为指向其`name`属性值为`Gerg`的新对象。但是我们从结果中看到，`person`对象并未发生任何改变，因此只是在函数内部引用被修改而已。这里函数形参`obj`被修改了，但是函数实参`person`并没有被修改，进一步说明函数参数的传递方式是按值传递的。（**也可以理解为引用类型，它传递的是引用地址的副本。而这个副本和本来的引用地址指向同一个堆地址。而这种传递方式称为共享传递。**）

函数式编程

虽然`JavaScript`并不是一门纯函数式编程的语言，但是它使用了许多函数式编程的特性。

我们通常通过函数封装来完成一件事情。例如，我想要计算任意三个数的和，我们就可以将这三个数作为参数，封装一个简单的函数。

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

当我们想要计算三个数的和时，直接调用该方法即可。

```
add(1, 2, 3); // 6
```

如果我们想要做的事情稍微复杂一点呢。例如我想要计算一个数组中的所有子项目的和。

```
function mergeArr(arr) {
    var result = 0;
    for(var i = 0; i < arr.length; i++) {
        result  += arr[i] ;
    }
    return result;
}
```

我们不通过函数封装的方式，那么再每次想要实现这个功能时，就不得不重新使用一次for循环,而封装之后，当我们想要再次做这件事情的时候，只需要一句话就可以了。

```
mergeArr([1, 2, 3, 4, 5]);
```

我们对于函数封装的意义都应该有非常明确的认知，但是我们要面临的问题是，当我们想要去封装一个函数时，如何做才是最佳实践呢？函数式编程能给我们答案。

我们在初学时，往往会不由自主的使用命令式编程的风格来完成我们想要干的事情。因为命令式编程更加的简单，直白。

```
var array = [1, 3, 'h', 5, 'm', '4'];
var res = [];
for(var i = 0; i < array.length; i ++) {//找出这个数组中的所有类型为number的子项
    if (typeof array[i] === 'number') {
        res.push(array[i]);
    }
}
```

在这种实现方式中，我们平铺直叙的实现了我们的目的。这样做的问题在于，当我们在另外的时刻，想要找出另外一个数组中所有的子项时，我们不得不把同样的逻辑再写一次。当出现次数变多时，我们的代码也变得更加糟糕且难以维护 。 **而函数式编程的思维则建议我们将这种会多次出现的功能封装起来以备调用。**

```
function getNumbers(array) {
    var res = [];
    array.forEach(function(item) {
        if (typeof item === 'number') {
            res.push(item);
        }
    })
    return res;
}

// 以上是我们的封装，以下是功能实现
var array = [1, 3, 'h', 5, 'm', '4'];
var res = getNumbers(array);
```

因此当我们将功能封装之后，我们实现同样的功能时，只需要写一行代码。而如果未来需求变动，或者稍作修改，我们只需要对`getNumbers`方法进行调整就可以了。而且我们在使用时，只需要关心这个方法能做什么，而不用关心他具体是怎么实现的。这也是函数式编程思维与命令式编程思维不同的地方之一。

函数式编程思维还具有以下几个特征。

1. **函数是第一等公民**

所谓**第一等公民**（`first class`），指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

```
var a = function foo() {}  // 赋值
function fn(function() {}, num) {}   // 函数作为参数

function var() {
    return function() {// 函数作为返回值
        ... ...
    }
}
```

我们先自定义这样一个函数。

```
function delay() {
    console.log('5000ms之后执行该方法.');
}
```

现在要做的是，如果要求你结合`setTimeout`方法，让`delay`方法延迟`5000ms`执行 。

```
var timer = setTimeout(function() {
    delay();
}, 5000);
```

如果你对函数是一等公民有一个深刻的认知，我想你会发现上面这种写法其实是有一些问题的。 函数既然能够作为一个参数传入另外一个函数，那么我们是不是可以直接将`delay`作为`setTimeout`的第一个参数，而不用额外的多加一层匿名函数呢？

因此，其实最正确的解法应该这样写。

```
var timer = setTimeout(delay, 5000);
```

其实第一种糟糕的方式很多人都在用，包括有多年工作经验的人也没有完全避免。而他们甚至还不知道自己问题出在什么地方。在未来的实践中，你还会遇到更多类似的场景。

2. **只用"表达式"，不用"语句"**

表达式（`expression`）是一个单纯的运算过程，总是有返回值；**语句**（`statement`）是执行某种操作，没有返回值。函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

假如我们的项目中，多处需要改变某个元素的背景色。因此我们可以这样封装一下。

```
var ele = document.querySelector('.test');

function setBackgroundColor(color) {
    ele.style.backgroundColor = color;
}
// 多处使用
setBackgroundColor('red');
setBackgroundColor('#ccc');
```

我们可以很明显的感受到，`setBackgroundColor`封装的仅仅只是一条语句。这并不是理想的效果。函数式编程期望一个函数有输入，也有输出。因此良好的习惯应该如下做。

```
function setBackgroundColor(ele, color) {
    ele.style.backgroundColor = color;
    return color;
}

// 多处使用
var ele = document.querySelector('.test');
setBackgroundColor(ele, 'red');
setBackgroundColor(ele, '#ccc');
```

了解这一点，可以让我们自己在封装函数的时候养成良好的习惯。

3. **纯函数**

相同的输入总会得到相同的输出，并且不会产生副作用的函数，就是纯函数。

所谓"副作用"（`side effect`），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

即所谓的只要是同样的参数传入，返回的结果一定是相等的。

例如我们期望封装一个函数，能够得到传入数组的最后一项。那么可以通过下面两种方式来实现。

```
function getLast(arr) {
    return arr[arr.length];
}

function getLast_(arr) {
    return arr.pop();
}

var source = [1, 2, 3, 4];

var last = getLast(source); // 返回结果4 原数组不变
var last_ = getLast_(source); // 返回结果4 原数据最后一项被删除
```

`getLast`与`getLast_`虽然同样能够获得数组的最后一项值，但是`getLast_`改变了原数组。而当原始数组被改变，那么当我们再次调用该方法时，得到的结果就会变得不一样。这样不可预测的封装方式，在我们看来是非常糟糕的。它会把我们的数据搞得非常混乱。在`JavaScript`原生支持的数据方法中，也有许多不纯的方法，我们在使用时需要非常警惕，我们要清晰的知道原始数据的改变是否会留下隐患。

```
var source = [1, 2, 3, 4, 5];

source.slice(1, 3); // 纯函数 返回[2, 3] source不变
source.splice(1, 3); // 不纯的 返回[2, 3, 4] source被改变

source.pop(); // 不纯的
source.push(6); // 不纯的
source.shift(); // 不纯的
source.unshift(1); // 不纯的
source.reverse(); // 不纯的

// 我也不能短时间知道现在source被改变成了什么样子，干脆重新约定一下
source = [1, 2, 3, 4, 5];

source.concat([6, 7]); // 纯函数 返回[1, 2, 3, 4, 5, 6, 7] source不变
source.join('-'); // 纯函数 返回1-2-3-4-5 source不变
```

4. **闭包**
5. **柯里化**