在JavaScript中，函数是个非常重要的对象，函数通常有三种表现形式：函数声明，函数表达式和函数构造器创建的函数。

本文中主要看看函数表达式及其相关的知识点。

函数表达式

首先，看看函数表达式的表现形式，函数表达式（Function Expression, FE）有下面四个特点：

- 在代码中须出现在表达式的位置
- 有可选的函数名称
- 不会影响变量对象(VO)
- 在代码执行阶段创建

下面就通过一些例子来看看函数表达式的这四个特点。

FE特点分析

例子一：在下面代码中，"add"是一个函数对象，"sub"是一个普通JavaScript变量，但是被赋值了一个函数表达式" function (a, b){ return a - b; } "：

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

var sub = function (a, b){
    return a - b;
}


console.log(add(1, 3));
// 4
console.log(sub(5, 1));
// 4
```

通过这个例子，可以直观的看到函数表达式的前两个特点：

- 在代码中须出现在表达式的位置
    - " function (a, b){ return a - b; } "出现在了JavaScript语句中的表达式位置
- 有可选的函数名称
    - " function (a, b){ return a - b; } "这个函数表达式没有函数名称，是个匿名函数表达式

例子二：为了解释函数表达式另外两个特点，继续看看下面的例子。

```
console.log(add(1, 3));
// 4
console.log(sub);
// undefined
console.log(sub(5, 1));
```

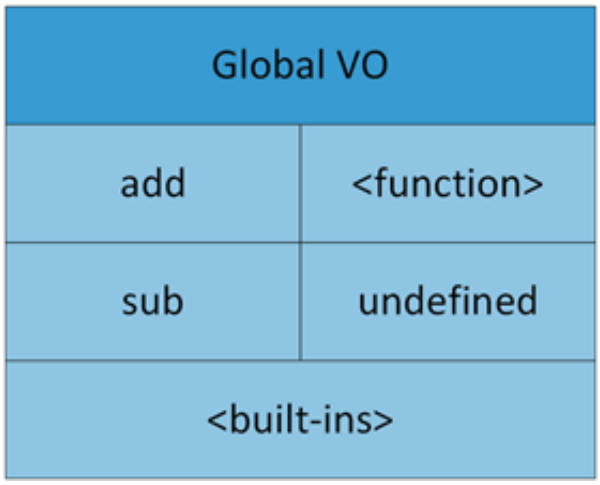
```
// Uncaught TypeError: sub is not a function(…)

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

var sub = function (a, b){
    return a - b;
}
```

在这个例子中，调整了代码的执行顺序，这次函数"add"执行正常，但是对函数表达式的执行失败了。

对于这个例子，可以参考"JavaScript的执行上下文"一文中的内容，当代码开始执行的时候，可以得到下图所示的Global VO。



在Global VO中，对"add"函数表现为JavaScript的"Hoisting"效果，所以即使在"add"定义之前依然可以使用；

但是对于"sub"这个变量，根据"Execution Context"的初始化过程，"sub"会被初始化为"undefined"，只有执行到" var sub = function (a, b){ return a - b; } "语句的时候，VO中的"sub"才会被赋值。

通过上面这个例子，可以看到了函数表达式的第四个特点

- 在代码执行阶段创建

例子三：对上面的例子进一步改动，这次给函数表达式加上了一个名字"_sub"，也就是说，这里使用的是一个命名函数表达式。

```
var sub = function _sub(a, b){
    console.log(typeof _sub);
    return a - b;
}

console.log(sub(5, 1));
// function
// 4
```

```
console.log(typeof _sub)
// undefined
console.log(_sub(5, 1));
// Uncaught ReferenceError: _sub is not defined(…)
```

根据这段代码的运行结果，可以看到"_sub"这个函数名，只能在"_sub"这个函数内部使用；当在函数外部访问"_sub"的时候，就是得到"Uncaught ReferenceError: _sub is not defined(…)"错误。

所以通过这个可以看到函数表达式的第三个特点：

- 不会影响变量对象(VO)

FE的函数名

到了这里，肯定会有一个问题，"_sub"不在VO中，那在哪里？

其实对于命名函数表达式，JavaScript解释器额外的做了一些事情：

1. 当解释器在代码执行阶段遇到命名函数表达式时，在函数表达式创建之前，解释器创建一个特定的辅助对象，并添加到当前作用域链的最前端
2. 然后当解释器创建了函数表达式，在创建阶段，函数获取了[[Scope]] 属性（当前函数上下文的作用域链）
3. 此后，函数表达式的函数名添加到特定对象上作为唯一的属性；这个属性的值是引用到函数表达式上
4. 最后一步是从父作用域链中移除那个特定的对象

下面是表示这一过程的伪代码：

```
specialObject = {};

Scope = specialObject + Scope;

_sub = new FunctionExpression;
_sub.[[Scope]] = Scope;
specialObject. _sub = _sub; // {DontDelete}, {ReadOnly}

delete Scope[0]; // 从作用域链中删除特殊对象specialObject
```

函数递归

这一小节可能有些钻牛角尖，但是这里想演示递归调用可能出现的问题，以及通过命名函数表达式以更安全的方式执行递归。

下面看一个求阶乘的例子，由于函数对象也是可以被改变的，所以可能会出现下面的情况引起错误。

```
function factorial(num){
    if (num <= 1){
        return 1;
```

```
    } else {
        return num * factorial(num-1);
    }
}

console.log(factorial(5))
// 120
newFunc = factorial
factorial = null
console.log(newFunc(5));
// Uncaught TypeError: factorial is not a function(…)
```

这时，可以利用函数的arguments对象的callee属性来解决上面的问题，也就是说在函数中，总是使用"arguments.callee"来递归调用函数。

```
function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * arguments.callee(num-1);
    }
}
```

但是上面的用法也有些问题，当在严格模式的时候"arguments.callee"就不能正常的工作了。

比较好的解决办法就是使用命名函数表达式，这样无论"factorial"怎么改变，都不会影响函数表达式" function f(num){…} "

```
var factorial = (function f(num){
    if (num <= 1){
        return 1;
    } else {
        return num * f(num-1);
    }
});
```

代码模块化

在JavaScript中，没有块作用域，只有函数作用域，函数内部可以访问外部的变量和函数，但是函数内部的变量和函数在函数外是不能访问的。

所以，通过函数（通常直接使用函数表达式），可以模块化JavaScript代码。

创建模块

为了能够到达下面的目的，我们可以通过函数表达式来建立模块。

- 创建一个可以重用的代码模块
- 模块中封装了使用者不必关心的内容，只暴露提供给使用者的接口
- 尽量与全局namespace进行隔离，减少对全局namespace的污染

下面看一个简单的例子：

```
var Calc = (function(){
    var _a, _b;

    return{
        add: function(){
            return _a + _b;
        },

        sub: function(){
            return _a - _b;
        },

        set: function(a, b){
            _a = a;
            _b = b;
        }
    }
}());

Calc.set(10, 4);
console.log(Calc.add());
// 14
console.log(Calc.sub());
// 6
```

代码中通过匿名函数表达式创建了一个"Calc"模块，这是一种常用的创建模块的方式：

- 创建一个匿名函数表达式，这个函数表达式中包含了模块自身的私有变量和函数；
- 通过执行这个函数表达式可以得到一个对象，对象中包含了模块想要暴露给用户的公共接口。

除了返回一个对象的方式，有的模块也会使用另外一种方式，将包含模块公共接口的对象作为全局变量的一个属性。

这样在代码的其他地方，就可以直接通过全局变量的这个属性来使用模块了。

例如下面的例子：

```
(function(){
    var _a, _b;

    var root = this;

    var _ = {
        add: function(){
            return _a + _b;
        },

        sub: function(){
            return _a - _b;
        },

        set: function(a, b){
            _a = a;
            _b = b;
        }
    }
```

```
    root._ = _;

}.call(this));

_.set(10, 4);
console.log(_.add());
// 14
console.log(_.sub());
// 6
```

立即调用的函数表达式

在上面两个例子中，都使用了匿名的函数表达式，并且都是立即执行的。如果去看看JavaScript一些开源库的代码，例如JQuery、underscore等等，都会发现类似的立即执行的匿名函数代码。

立即调用的函数表达式通常表现为下面的形式：

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

(function () {
    /* code */
} ());
在underscore这个JavaScript库中，使用的是下面的方式：
(function () {
    // Establish the root object, `window` in the browser, or `exports` on the server.
var root = this;

/* code */

} .call(this));
```

在这里，underscore模块直接对全局变量this进行了缓存，方便模块内部使用。

总结

本文简单介绍了JavaScript中的函数表达式，并通过三个例子解释了函数表达式的四个特点。

- 在代码中须出现在表达式的位置
- 有可选的函数名称
- 不会影响变量对象(VO)
- 在代码执行阶段创建

通过函数表达式可以方便的建立JavaScript模块，通过模块可以实现下面的效果：

- 创建一个可以重用的代码模块
- 模块中封装了使用者不必关心的内容，只暴露提供给使用者的接口
- 尽量与全局namespace进行隔离，减少对全局namespace的污染