**学习目标：**

1、函数表达式的特征

2、使用函数实现递归

3、使用闭包定义私有变量

**一、定义函数的方式：**

1、函数声明：

function functionName(arg0, arg1, arg2) {

//函数体

}

函数声明提升：意思是在执行代码之前会先读取函数声明。这就意味着可以把函数声明放在调用它的语句后面。

如：

sayHi();

function sayHi(){

alert("Hi!");

}

2、函数表达式：

var functionName = function(arg0, arg1, arg2){

//函数体

};

如：

sayHi(); //错误：函数还不存在

var sayHi = function(){

alert("Hi!");

};

说明：理解函数提升的关键，就是理解函数声明与函数表达式之间的区别。

能够创建函数再赋值给变量，也就能够把函数作为其他函数的值返回。

**二、递归**

1、函数调用函数自身（执行递的动作）

2、最后一次判断一个终止条件。执行归的动作。

如：求阶乘

function factorial(num){

if (num <= 1){

return 1;

} else {

return num \* factorial(num-1);

}

}

说明：arguments.callee，该属性是一个指针，指向拥有这个arguments对象的函数（严格模式下运行会导致错误）

所以，上面的改一改：

function factorial(num){

if (num <= 1){

return 1;

} else {

return num \* arguments.callee(num-1);

}

}

作业：对象的复制

**三、闭包**

闭包：当函数可以记住并访问所在的作用域(全局作用域除外)时，就产生了闭包，即使函数是在当前作用域之外执行。

简单来说，假设函数A在函数B的内部进行定义了，并且当函数A在执行时，访问了函数B内部的变量对象，那么B就是一个闭包。

例：

var fn = null;

function foo() {

var a = 2;

function innnerFoo() {

console.log(a);

}

fn = innnerFoo; // 将 innnerFoo的引用，赋值给全局变量中的fn

}

function bar() {

fn(); // 此处的保留的innerFoo的引用

}

foo();

bar(); //

在上面的例子中，foo()执行完毕之后，按照常理，其执行环境生命周期会结束，所占内存被垃圾收集器释放。但是通过fn = innerFoo，函数innerFoo的引用被保留了下来，复制给了全局变量fn。这个行为，导致了foo的变量对象，也被保留了下来。于是，函数fn在函数bar内部执行时，依然可以访问这个被保留下来的变量对象。所以此刻仍然能够访问到变量a的值。这样，我们就可以称foo为闭包。

例2：

var fn = null;

function foo() {

var a = 2;

function innnerFoo() {

console.log(c);

console.log(a);

}

fn = innnerFoo; // 将 innnerFoo的引用，赋值给全局变量中的fn

}

function bar() {

var c = 100;

fn(); // 此处的保留的innerFoo的引用

}

foo();

bar();

闭包的应用场景

(function () {

var a = 10;

var b = 20;

function add(num1, num2) {

var num1 = !!num1 ? num1 : a;

var num2 = !!num2 ? num2 : b;

return num1 + num2;

}

window.add = add;

})();

console.log(add(10, 20)); //

问题：

利用闭包，修改下面的代码，让循环输出的结果依次为1， 2， 3， 4， 5

for (var i=1; i<=5; i++) {

setTimeout( function timer() {

console.log(i);

}, i\*1000 );

}

解决方案：

for (var i=1; i<=5; i++) {

(function(i) {

setTimeout( function timer() {

console.log(i);

}, i\*1000 );

})(i)

}

最后：

this 对象是在运行时基于函数的执行环境绑定的

1、在全局函数中， this 等于 window ，

2、而当函数被作为某个对象的方法调用时， this 等于那个对象

3、匿名函数的执行环境具有全局性，因此其 this 对象通常指向 window

如：

var name = "The Window";

var object = {

name: "My Object",

getNameFunc: function() {

return function() {

return this.name;

};

}

};

alert(object.getNameFunc()());

再如：

var name = "The Window";

var object = {

name: "My Object",

getNameFunc: function() {

var that = this;

return function() {

return that.name;

};

}

};

alert(object.getNameFunc()());

**综合案例（点击li弹出li的索引）**