第一节、函数概述

JavaScript深入浅出

函数、作用域

@Bosn

概念

函数是一块JavaScript代码，被定义一次，但可执行和调用多次。
JS中的函数也是对象，所以JS函数可以像其它对象那样操作和传递
所以我们也常叫JS中的函数为函数对象。

函数名　　参数列表

```
function foo(x, y) {
    if (typeof x === 'number' &&
        typeof y === 'number') {
        return x + y;
    } else {
        return 0;
    }
}

foo(1, 2); // 3
```

函数体

重点

```
function foo(x, y) {
    if (typeof x === 'number' &&
        typeof y === 'number') {
        return x + y;
    } else {
        return 0;
    }
}

foo(1, 2); // 3
```

- this
- arguments
- 作用域
- 不同调用方式
- 不同创建方法

不同的调用方式

直接调用

```
foo();
```

对象方法

```
o.method();
```

构造器

```
new Foo();
```

call/apply/bind

```
func.call(o);
```

第二节、函数声明与函数表达式

声明 VS. 表达式

函数声明

```
function add (a, b) {
    a = +a;
    b = +b;
    if (isNaN(a) || isNaN(b)) {
            return;
    }
    return a + b;
}
```

函数表达式

```
// function variable
var add = function (a, b) {
    // do sth
};

// IEF(Immediately Executed Function)
(function() {
     // do sth
})();

// first-class function
return function() {
    // do sth
};

// NFE (Named Function Expression)
var add = function foo (a, b) {
    // do sth
};
```

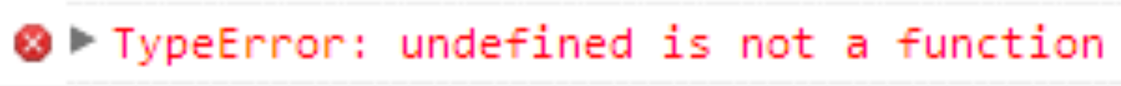
变量&函数的声明前置

```
var num = add(1, 2);
console.log(num);   // result: 3

function add (a, b) {
    a = +a;
    b = +b;
    if (isNaN(a) || isNaN(b)) {
        return;
    }
    return a + b;
}
```

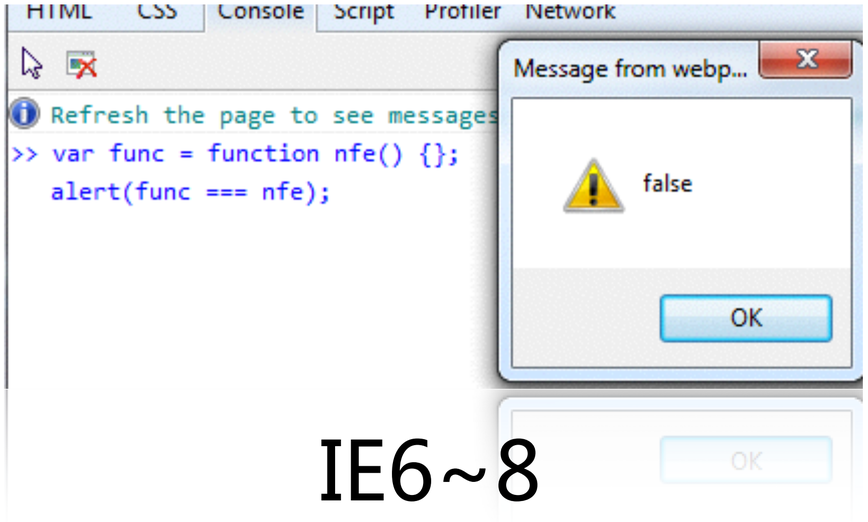
```
var num = add(1, 2);
console.log(num);

var add = function (a, b) {
    a = +a;
    b = +b;
    if (isNaN(a) || isNaN(b)) {
        return;
    }
    return a + b;
}
```

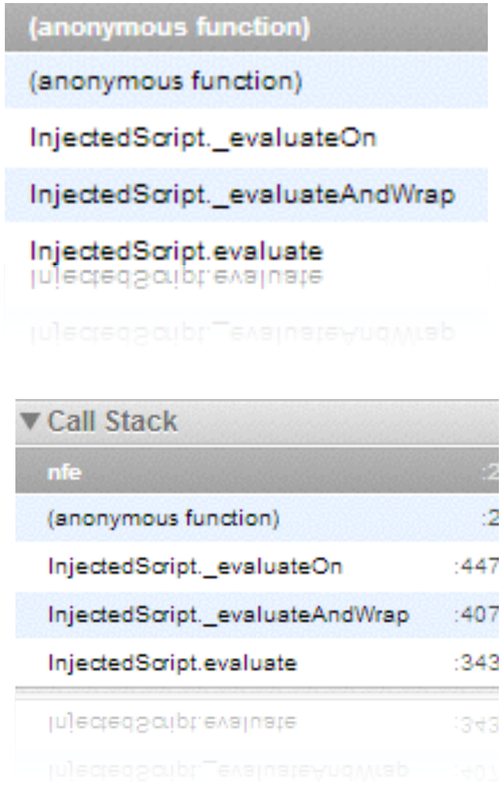


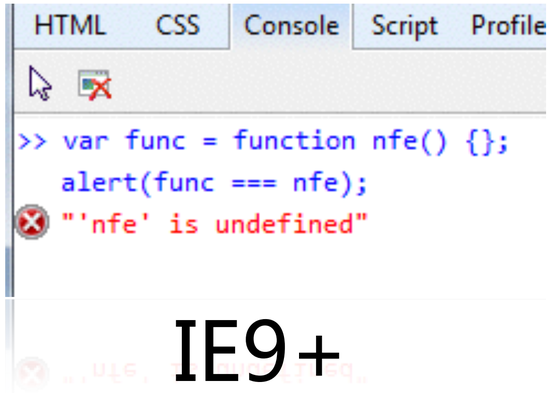
命名函数表达式(NFE)

```
var func = function nfe() {};
alert(func === nfe);
// 递归调用
var func = function nfe() {/** do sth.**/ nfe();}
```



IE6~8





IE9+

Function构造器

```
var func = new Function('a', 'b', 'console.log(a + b);');
func(1, 2);  // 3

var func = Function('a', 'b', 'console.log(a + b);');
func(1, 2);  // 3
```

Function构造器

localVal仍为局部变量

```
// CASE 1
Function('var localVal = "local"; console.log(localVal);')();
console.log(typeof localVal);
// result: local, undefined

// CASE 2
var globalVal = 'global';
(function() {
    var localVal = 'local';
    Function('console.log(typeof localVal, typeof globalVal);')();
})();
// result: undefined, string
```

local不可访问，全局变量global可以访问

比一比

	函数声明	函数表达式	函数构造器
前置	√		
允许匿名		√	√
立即调用		√	√
在定义该函数的作用域通过函数名访问	√		
没有函数名			√

第三节、this

全局的this(浏览器)

```
console.log(this.document === document); // true

console.log(this === window); // true

this.a = 37;
console.log(window.a); // 37
```

一般函数的this(浏览器)

```
function f1(){
  return this;
}

f1() === window; // true, global object
```

```
function f2(){
  "use strict"; // see strict mode
  return this;
}

f2() === undefined; // true
```

作为对象方法的函数的this

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // logs 37
```

```
var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // logs 37
```

对象原型链上的this

```
var o = {f:function(){ return this.a + this.b; }};
var p = Object.create(o);
p.a = 1;
p.b = 4;

console.log(p.f()); // 5
```

get/set方法与this

```
function modulus(){
  return Math.sqrt(this.re * this.re + this.im * this.im);
}

var o = {
  re: 1,
  im: -1,
  get phase(){
    return Math.atan2(this.im, this.re);
  }
};

Object.defineProperty(o, 'modulus', {
    get: modulus, enumerable:true, configurable:true});

console.log(o.phase, o.modulus); // logs -0.78 1.4142
```

构造器中的this

```
function MyClass(){
  this.a = 37;
}

var o = new MyClass();
console.log(o.a); // 37


function C2(){
  this.a = 37;
  return {a : 38};
}

o = new C2();
console.log(o.a); // 38
```

call/apply方法与this

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

var o = {a:1, b:3};

add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16

add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34

function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // "[object Number]"
```

bind方法与this

```
function f(){
  return this.a;
}

var g = f.bind({a : "test"});
console.log(g()); // test

var o = {a : 37, f : f, g : g};
console.log(o.f(), o.g()); // 37, test
```

第四节、函数属性 & arguments

函数属性 & arguments

```
function foo(x, y, z) {
    'use strict';
    arguments.length; // 2
    arguments[0]; // 1
    arguments[0] = 10;
    x; // change to 10;

    arguments[2] = 100;
    z; // still undefined !!!
    arguments.callee === foo; // true
}

foo(1, 2);
foo.length; // 3
foo.name; // "foo"
```

严格模下仍然是1

绑定关系

未传参数
失去绑定关系

严格模下不能使用

foo.name - 函数名

foo.length - 形参个数

arguments.length - 实参个数

apply/call方法(浏览器)

```
function foo(x, y) {
    console.log(x, y, this);
}

foo.call(100, 1, 2); // 1, 2, Number(100)
foo.apply(true, [3, 4]); // 3, 4, Boolean(true)
foo.apply(null); // undefined, undefined, window
foo.apply(undefined); // undefined, undefined, window
```

apply/call方法(浏览器)

```
function foo(x, y) {
    'use strict';
    console.log(x, y, this);
}

foo.apply(null); // undefined, undefined, null
foo.apply(undefined); // undefined, undefined, undefined
```

bind方法

```
this.x = 9;
var module = {
  x: 81,
  getX: function() { return this.x; }
};

module.getX(); // 81

var getX = module.getX;
getX(); // 9

var boundGetX = getX.bind(module);
boundGetX(); // 81
```

bind与currying

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

var func = add.bind(undefined, 100);
func(1, 2); // 103

var func2 = func.bind(undefined, 200);
func2(10); // 310
```

bind与currying

```
function getConfig(colors, size, otherOptions) {
    console.log(colors, size, otherOptions);
}

var defaultConfig = getConfig.bind(null, "#CC0000", "1024 * 768");

defaultConfig("123"); // #CC0000 1024 * 768 123
defaultConfig("456"); // #CC0000 1024 * 768 456
```

bind与new

```
function foo() {
    this.b = 100;
    return this.a;
}

var func = foo.bind({a:1});

func(); // 1
new func(); // {b : 100}
```

bind方法模拟

绑定this　　科里化

```
function foo() {
    this.b = 100;
    return this.a;
}

var func = foo.bind({a:1});

func(); // 1
new func(); // {b : 100}
```

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function(oThis) {
    if (typeof this !== 'function') {
      // closest thing possible to the ECMAScript 5
      // internal IsCallable function
      throw new TypeError('What is trying to be bound is not callable');
    }
    var aArgs   = Array.prototype.slice.call(arguments, 1),
        fToBind = this,
        fNOP    = function() {},
        fBound  = function() {
          return fToBind.apply(this instanceof fNOP? this : oThis,
                 aArgs.concat(Array.prototype.slice.call(arguments)));
        };
    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();

    return fBound;
  };
}
```

第五节、理解闭包

闭包的例子

```
function outer() {
    var localVal = 30;
    return localVal;
}

outer(); // 30
```

```
function outer() {
    var localVal = 30;
    return function() {
        return localVal;
    }
}

var func = outer();
func(); // 30
```

闭包-无处不在

```
!function() {
    var localData = "localData here";
    document.addEventListener('click',
        function(){
            console.log(localData);
    });
}();
```

```
!function() {
    var localData = "localData here";
    var url = "http://www.baidu.com/";
    $.ajax({
        url : url,
        success : function() {
            // do sth...
            console.log(localData);
        }
    });
}();
```

闭包-常见错误之循环闭包

```
document.body.innerHTML = "<div id=div1>aaa</div>"
    + "<div id=div2>bbb</div><div id=div3>ccc</div>";
for (var i = 1; i < 4; i++) {
    document.getElementById('div' + i).
        addEventListener('click', function() {
        alert(i); // all are 4!
    });
}
```

```
document.body.innerHTML = "<div id=div1>aaa</div>"
    + "<div id=div2>bbb</div><div id=div3>ccc</div>";
for (var i = 1; i < 4; i++) {
    !function(i) {
        document.getElementById('div' + i).
            addEventListener('click', function() {
            alert(i); // 1, 2, 3
        });
    }(i);
}
```

闭包-封装

```
(function() {
    var _userId = 23492;
    var _typeId = 'item';
    var export = {};

    function converter(userId) {
        return +userId;
    }

    export.getUserId = function() {
        return converter(_userId);
    }

    export.getTypeId = function() {
        return _typeId;
    }
    window.export = export;

}());
```

```
export.getUserId(); // 23492
export.getTypeId();  // item

export._userId;    // undefined
export._typeId;    // undefined
export.converter; // undefined
```

闭包的概念

在计算机科学中，闭包（也称词法闭包或函数闭包）是指一个函数或函数的引用，与一个引用环境绑定在一起。这个引用环境是一个存储该函数每个非局部变量（也叫自由变量）的表。

闭包，不同于一般的函数，它允许一个函数在立即词法作用域外调用时，仍可访问非本地变量。

from 维基百科

闭包-小结

灵活和方便

封装

空间浪费

内存泄露

性能消耗

第六节、作用域

全局、函数、eval

```
var a = 10;                 全局
(function() {
    var b = 20;             函数
})();
console.log(a);             // 10
console.log(b);             // error, b in not defined

for (var item in {a : 1, b : 2}) {
    console.log(item);
}
console.log(item);          // item still in scope

eval("var a = 1;");         eval
```

作用域链

```
function outer2() {
    var local2 = 1;
    function outer1() {
        var local1 = 1;
        // visit local1, local2 or global3
    }
    outer1();
}
var global3 = 1;
outer2();

function outer() {
    var i = 1;
    var func = new Function("console.log(typeof i);");
    func();  // undefined}
outer();
```

利用函数作用域封装

```
(function() {
    // do sth here
    var a, b;
})();

!function() {
    // do sth here
    var a, b;
}();
```

第七节、ES3执行上下文（可选）

全局、函数、eval

```
var a = 10;
(function() {
    var b = 20;
})();
console.log(a);
console.log(b);

for (var item in {a : 1, b : 2}) {
    console.log(item);
}
console.log(item);

eval("var a = 1;");
```

全局

函数

// 10

// error, b in not defined

// item still in scope

eval

抽象概念：执行上下文、变量对象...
在ECMA-262 第三版标准规范中定义

概念-执行上下文

执行上下文(Execution Context，缩写EC)

```
console.log('EC0');

function funcEC1() {
    console.log('EC1');
    var funcEC2 = function() {
        console.log('EC2');
        var funcEC3 = function() {
            console.log('EC3');
        };
        funcEC3();
    }
    funcEC2();
}
funcEC1();
// EC0 EC1 EC2 EC3
```

概念-变量对象

JS解释器如何找到我们定义的函数和变量？

变量对象(Variable Object, 缩写为VO)是一个抽象概念中的“对象”，它用于存储执行上下文中的：

1. 变量
2. 函数声明
3. 函数参数

执行上下文与变量对象

```
activeExecutionContext = {
    VO : {
        data_var,
        data_func_declaration,
        data_func_arguments
    }
};
```

GlobalContextVO　　(VO === this === global)

```
var a = 10;
function test(x) {
    var b = 20;
}
test(30);
```

```
VO(globalContext) = {
    a : 10,
    test : <ref to function>
};
VO(test functionContext) = {
    x : 30,
    b: 20
};
```

全局执行上下文(浏览器)

```
VO(globalContext) === [[global]];

[[global]] = {
    Math  : <...>,
    String : <...>,
    isNaN : function() {[Native Code]}
    ...
    ...
    window : global  // applied by browser(host)
};
```

GlobalContextVO　　(VO === this === global)

```
String(10);              //[[global]].String(10);
window.a = 10;       // [[global]].window.a = 10
this.b = 20;          // [[global]].b = 20;
```

函数中的激活对象

```
VO(functionContext) === AO;

AO = {
    arguments : <Arg0>
};

arguments = {
    callee,
    length,
    properties-indexes
};
```

1. 变量初始化阶段

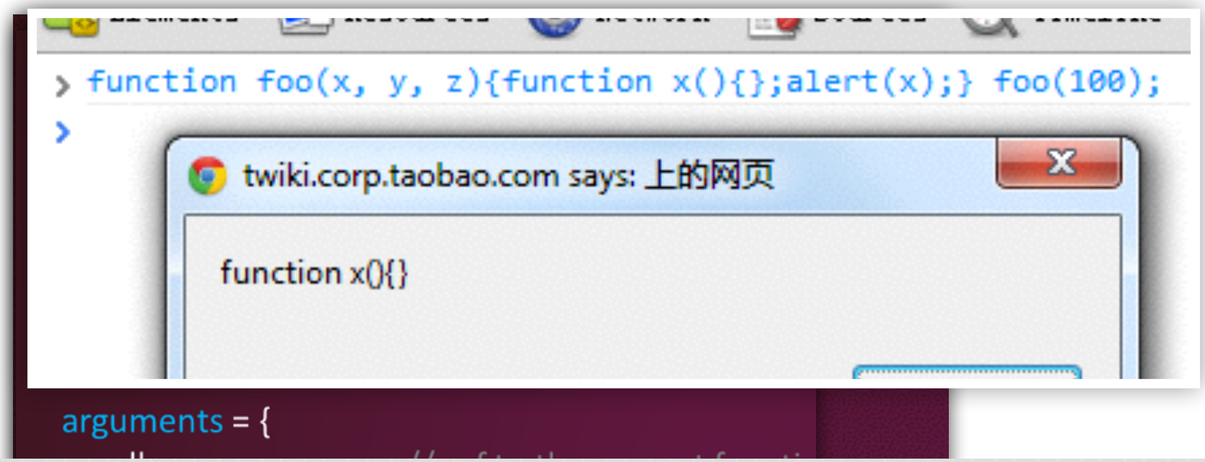
VO按照如下顺序填充:

1. 函数参数 (若未传入，初始化该参数值为undefined)
2. 函数声明 (若发生命名冲突，会覆盖)
3. 变量声明 (初始化变量值为undefined，若发生命名冲突，会忽略。)

```
function test(a, b) {
  var c = 10;
  function d() {}
  var e = function _e() {};
  (function x() {});
  b = 20;
}
test(10);
```

```
AO(test) = {
  a: 10,
  b: undefined,
  c: undefined,
  d: <ref to func "d">
  e: undefined
};
```

函数表达式不会影响VO




```
function foo(x, y, z){function func(){}; var func; console.log(func);}; foo(100);
function func(){}
undefined
function foo(x, y, z){function func(){}; var func = 1; console.log(func);}; foo(100);
1
undefined
```

2. 代码执行阶段

```
VO['c'] = 10;
VO['e'] = function _e() {};
VO['b'] = 20;

AO(test) = {
  a: 10,
  b: undefined,
  c: undefined,
  d: <ref to func "d">
  e: undefined
};
```

```
function test(a, b) {
  var c = 10;
  function d() {}
  var e = function _e() {};
  (function x() {});
  b = 20;
}
test(10);

AO(test) = {
  a: 10,
  b: 20,
  c: 10,
  d: <reference to FunctionDeclaration "d">
  e: function _e() {};
};
```

测试一下

```
alert(x);   // function

var x = 10;
alert(x);  // 10
x = 20;

function x() {}
alert(x);   // 20

if (true) {
    var a = 1;
} else {
    var b = true;
}

alert(a);   // 1
alert(b);   // undefined
```

第八节、小结

理解函数

函数声明与表达式

this与调用方式

函数属性与arguments

理解闭包和作用域

解析ES3执行上下文

谢谢