

# Front End Book

*draft*

陈磊

<https://github.com/HereChen/front-end-book>

November 2, 2015



# 前言

Web 前端开发笔记整理。内容：JavaScript、HTML、CSS、web 性能优化、工具、框架、资源。



# Contents



# Chapter 1

# JavaScript

## 1.1 JavaScript 组成

一个完整的 JavaScript 实现是由以下 3 个不同部分组成的：

- 核心（ECMAScript）
- 文档对象模型（DOM）
- 浏览器对象模型（BOM）

reference: [JavaScript 实现](#), [JavaScript 学习总结（三）BOM 和 DOM 详解](#)

## 1.2 类型、值和变量

### 数据类型

- 基本数据类型：String 字符串；Number 数字；Boolean 布尔。
- 复合数据类型：Object 对象；Array 数组。
- 特殊数据类型：Null 空对象；Undefined 未定义。

reference: [数据类型 \(JavaScript\), msdn](#)

### null, NaN, undefined

JavaScript 中有 6 个值为‘假’：false, null, undefined, 0, ''(空字符串), NaN. 其中 NaN 是 JavaScript 中唯一不等于自身的值, 即 NaN == NaN 为 false.

```
console.log( false == null ) // false
console.log( false == undefined ) // false
console.log( false == 0 ) //
true console.log( false == '' ) // true
console.log( false == NaN ) //false

console.log( null == undefined ) // true
console.log( null == 0 ) // false
console.log( null == '' ) // false
console.log( null == NaN ) // false

console.log( undefined == 0 ) // false
console.log( undefined == '' ) // false
console.log( undefined == NaN ) // false

console.log( 0 == '' ) // true
console.log( 0 == NaN ) // false
```

对于 === 以上全为 false。对于 ==, 以下几组为 true: null 和 undefined; false、0、''。

reference: [JavaScript 中奇葩的假值{.target='\\_blank'}，阮一峰, undefined 与 null 的区别, 2014](#)

## 1.3 函数

### 常用函数

- `string.slice(start,end)` 复制 string 中的一部分。
- `string.indexOf(searchString, position)` 在 string 中查找 searchString。如果被找到, 返回第一个匹配字符的位置, 否则返回 -1。可选参数 position 可设置从 string 的某个指定的位置开始查找。
- `object.hasOwnProperty(name)`
- `array.splice(start, deleteCount, item)` 从 array 中移除一个或多个元素并用新的 item 替换他们。
- `array.concat(item...)` 产生一个新数组, 它包含一份 array 的前复制, 并把一个或多个参数 item 附加在其后面。



- `array.join(separator)` `join` 方法把一个 `array` 构成一个字符串。它先把 `array` 中的每个元素构造成一个字符串，接着用一个 `separator` 分隔符把它们连接在一起。
- `array.pop()` 移除最后一个元素。
- `array.push(item...)` 将一个或多个参数附加到数组的尾部。
- `array.reverse()` 反转 `array` 元素的顺序。
- `array.sort(comparefn)` 数组排序。
- `array.shift()` 移除数组中的第一个元素并返回该元素。

reference: [Douglas Crockford, JavaScript 语言精粹](#)

## Date

```
var t = new Date(); var tt = [  
  t.getFullYear(), '年', // 不是 getYear()  
  t.getMonth() + 1, '月',  
  t.getDate(), '日', ' ',  
  t.getHours(), '时',  
  t.getMinutes(), '分',  
  t.getSeconds(), '秒'].join('');  
console.log(tt); // 2015年10月30日 22时6分21秒
```

注意 `getMonth()` 和 `getDay()` 都是从 0 开始的，需要加 1。

reference: [Date](#), [mdn](#)

## typeof 和 instanceof

(1) `typeof`: `typeof operand`

`typeof` 操作符返回一个字符串, 表示未经求值的操作数 (unevaluated operand) 的类型。`typeof` 只有一个实际应用场景，就是用来检测一个对象是否已经定义或者是否已经赋值。而这个应用却不是来检查对象的类型。除非为了检测一个变量是否已经定义，我们应尽量避免使用 `typeof` 操作符。

```
t  
typeof foo == 'undefined' // 若 foo  
未定义，返回 true
```

类型 | 结构:— | :— Undefined | 'undefined' Null | 'object' 布尔值 | 'boolean' 数值 | 'number' 字符串 | 'string' Symbol (ECMAScript 6 新增) | 'symbol' 宿主对象 (JS 环境提供的, 比如浏览器) | Implementation-dependent 函数对象 (implements [[Call]] in ECMA-262 terms) | 'function' 任何其他对象 | 'object' {: class='table'}

(2) instanceof: object instanceof constructor

instanceof 运算符可以用来判断某个构造函数的 prototype 属性是否存在另外一个要检测对象的原型链上。(instanceof 运算符用来检测 constructor.prototype 是否存在于参数 object 的原型链上。)

```
function C()\{\}
```

// 定义一个构造函数

```
function D()\{\}
```

// 定义另一个构造函数

```
var o = new C(); o instanceof C; // true, 因为: Object.getPrototypeOf(o)
=== C.prototype o instanceof D; // false, 因为 D.prototype 不在 o 的原型链上
o instanceof Object; //
true, 因为 Object.prototype.isPrototypeOf(o) 返回 true C.prototype
instanceof Object // true, 同上
```

```
C.prototype = \{\}; var o2 = new C(); o2 instanceof C; // true o
instanceof C; //
false, C.prototype 指向了一个空对象, 这个空对象不在 o 的原型链上.
```

```
D.prototype = new C(); var o3 = new D(); o3 instanceof D; // true o3
instanceof C; // true
```

reference: [typeof 和 instanceof 的区别](#), [instanceof](#), [typeof](#)

## setInterval 和 setTimeout

- setTimeout 只执行一次
- setInterval 连续执行多次

## call 和 apply

(1) call: fun.call(thisArg, arg1, arg2, ...)

call() 方法在使用一个指定的 this 值和若干个指定的参数值的前提下调用某个函数或方法。

**thisArg** 在 fun 函数运行时指定的 this 值。需要注意的是，指定的 this 值并不一定是该函数执行时真正的 this 值，如果这个函数处于非严格模式下，则指定为 null 和 undefined 的 this 值会自动指向全局对象 (浏览器中就是 window 对象)，同时值为原始值 (数字，字符串，布尔值) 的 this 会指向该原始值的自动包装对象。

**arg1, arg2, ...** 指定的参数列表。

```
/* 将函数的参数 arguments 转换为数组 */
function listFirst()\{ // this 指向 arguments // 下面的语句相当于
arguments.slice(0)，但由于 arguments 不是数组，不能直接调用 slice 方法
var arr = Array.prototype.slice.call(arguments, 0); for (var i=0; i
\textless{} arr.length; i++)\{ console.log(arr[{}i{}]); \} \}

listFirst(1,2,3); // 调用，输出 1,2,3
```

(2) apply: fun.apply(thisArg, [argsArray])

apply() 方法在指定 this 值和参数 (参数以数组或类数组对象的形式存在) 的情况下调用某个函数。

**thisArg** 在 fun 函数运行时指定的 this 值。需要注意的是，指定的 this 值并不一定是该函数执行时真正的 this 值，如果这个函数处于非严格模式下，则指定为 null 或 undefined 时会自动指向全局对象 (浏览器中就是 window 对象)，同时值为原始值 (数字，字符串，布尔值) 的 this 会指向该原始值的自动包装对象。

**argsArray** 一个数组或者类数组对象，其中的数组元素将作为单独的参数传给 fun 函数。如果该参数的值为 null 或 undefined，则表示不需要传入任何参数。从 ECMAScript 5 开始可以使用类数组对象。浏览器兼容性请参阅本文底部内容。

在调用一个存在的函数时，你可以为其指定一个 this 对象，无需此参数时第一个参数可用 null (比如对于 add)。this 指当前对象，也就是正在调用这个函数的对象。使用 apply，你可以只写一次这个方法然后在另一个对象中继承它，而不用在新对象中重复写该方法。

```
/* 例一：将函数的参数 arguments
转换为数组 */ function listFirst()\{ // this 指向 arguments var arr =
Array.prototype.slice.apply(arguments, [{}0{}]); for (var i=0; i
```

```
\textless{} arr.length; i++)\{ console.log(arr[{}i{}]); \} \}
```

```
listFirst(1,2,3); // 调用, 输出 1,2,3
```

```
/* 例二: push 一个数组 */ var arr1=new Array('1','2','3'); var  
arr2=new Array('4','5','6');  
Array.prototype.push.apply(arr1,arr2);
```

reference: [Function.prototype.apply\(\)](#) {:target='\_blank'}, [apply 和 call 的用法](#) {:target='\_blank'}

### new/构造函数

构造函数只是一些使用 `new` 操作符时被调用的普通函数。使用 `new` 来调用函数, 或者说发生构造函数调用时, 会自动执行下面的操作。

1. 创建（或者说构造）一个全新的对象。
2. 这个新对象会被执行 `[[prototype]]` 链接。
3. 这个对象会绑定到函数调用的 `this`。
4. 如果函数没有返回其他对象, 那么 `new` 表达式中的函数调用会自动返回这个新对象。

### 函数调用模式

除了声明时定义的形式参数, 每个函数还接收两个附加的参数: `this` 和 `arguments`. 在 JavaScript 中一共有 4 种调用模式: 方法调用模式、函数调用模式、构造器调用模式和 `apply` 调用模式. 函数调用的模式不同, 对应的 `this` 值也会不同。

#### (1) 方法调用模式

当一个函数被保存为对象的一个属性时, 我们称它为一个方法。当一个方法被调用时, `this` 被绑定到该对象。

```
// print 作为 obj 属性被保存, 当 print  
被调用时, this 指向 obj var obj = \{ value: 'I am a string.', print:  
function ()\{ console.log(this.value); \} \}
```

```
obj.print(); // 调用
```

#### (2) 函数调用模式

当一个函数并非一个对象的属性时, 那么它就被当做一个函数来调用。以此模式调用函数时, `this` 被绑定到全局对象。

```
var func = function()\n{\n  console.log('Hello here.');
```

```
func(); // 调用
```

### (3) 构造器调用模式

如果在一个函数前面带上 `new` 来调用, 那么背地里将会创建一个连接到该函数的 `prototype` 成员的新对象, 同时 `this` 会绑定到那个新对象上。

```
var Quo = function(string)\n{\n  this.status\n  = string;\n}\nQuo.prototype.get\_status = function()\n{\n  return\n  this.status;\n}
```

```
var myQuo = new Quo('new Quo.') // 调用 myQuo.get\_status(); // 'new\nQuo.'
```

### (4) `apply` 调用模式

`apply` 方法让我们构建一个参数数组传递给调用函数。它允许我们选择 `this` 的值。`apply` 方法接收两个参数, 第 1 个是要绑定给 `this` 的值, 第 2 个就是一个参数数组。

reference: [《JavaScript 语言精粹 \(修订版\)》, 第 4 章函数](#) {;target='\_blank'}

## **this**

`this` 在运行时绑定, 它的上下文取决于函数调用时的各种条件。`this` 的绑定和函数声明的位置没有任何关系, 只取决于函数的调用方式。

判断 `this` 的优先级, 可以按照下面的顺序进行判断:

- 函数是否在 `new` 中调用 (`new` 绑定)? 如果是的话 `this` 绑定的是新创建的对象。

```
var bar = new foo(); // 绑定 bar
```

- 函数是否通过 `call`、`apply` (显示绑定) 或者硬绑定调用? 如果是的话, `this` 绑定的是指定的对象。

```
var bar = foo.call(obj2); // 绑定 obj2
```

- 函数是否在某个上下文对象中调用 (隐式调用)? 如果是的话, `this` 绑定的是那个上下文对象。(调用时是否被某个对象拥有或包含, 对象属性引用链中只有最顶层或者最后一层会影响调用位置 `obj1.obj2.foo()`; // 绑定 `obj2`)

```
var bar = obj1.foo(); // 绑定 obj1
```

- 如果都不是的话, 使用默认绑定。如果在严格模式下, 就绑定到 `undefined`, 否则绑定到全局对象 (`window`)。

```
var bar = foo(); // 绑定 undefined 或  
window
```

reference: [你不知道的 JavaScript \(上卷\)](#)

## 闭包

当函数可以记住并访问所在的词法作用域时, 就产生了闭包, 即使函数是在当前词法作用域之外执行。(闭包是发生在定义时的。)

```
// foo() 定义的中括号内就是 bar  
的词法作用域 function foo() \{ var a = 2; function bar() \{ console.log(  
a ); \} return bar; \} var baz = foo(); baz(); // 2, 这就是闭包,  
用到变量 a
```

```
var fn; function foo() \{ var a = 2;  
function baz() \{ console.log( a ); // 2 \} fn = baz; // 将 baz  
赋值给全局变量 // 调用 fn 相当于执行的 baz, 而其词法作用域在 foo()  
定义函数内. \}
```

```
function bar(fn) \{ fn(); // 这就是闭包, 用到变量 a \}
```

```
foo(); bar(); // 2
```

reference: [你不知道的 JavaScript \(上卷\)](#)

## 1.4 继承方法

- 原型链继承
- 构造继承
- 实例继承
- 拷贝继承

reference: [继承与原型链](#), [mdn](#), [RayChase](#), [JavaScript 实现继承的几种方式](#)

## 1.5 正则表达式

### (1) 特殊字符

- `\d` 任意一个数字, 等价于 `[0-9]`
- `\D` 任意一个非数字, 等价于 `[^0-9]`
- `\w` 任意一个字母、数字或下划线字符, 等价于 `[a-zA-Z_]`
- `\W` 任意一个非字母、数字和下划线字符, 等价于 `[^a-zA-Z_]`
- `\s` 任意一个空白字符, 包括换页符、换行符、回车符、制表符和垂直制表符, 等价于 `[\f\n\r\t\v]`
- `\S` 任意一个非空白符, 等价于 `[^\f\n\r\t\v]`
- `.` 换行和回车以外的任意一个字符, 等价于 `[^\n\r]`

### (2) 次数匹配

- `?` 最多一次 (零次或一次)
- `+` 至少一次
- `*` 任意次
- `{n}` 只能出现 `n` 次
- `{n,m}` 至少 `n` 次, 最多 `m` 次

reference: [正则总结: JavaScript 中的正则表达式](#)

## 1.6 ES5 新特性

新功能包括: 原生 JSON 对象、继承的方法、高级属性的定义以及引入严格模式。

reference: [梦禅](#), [ECMAScript 各版本简介及特性](#), [segmentfault](#)

## 1.7 ES6 新特性

模块，类，块级作用域，Promise，生成器...

## 1.8 Ajax

### Ajax 实现

(1) 原生 Ajax 实现: GET

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET','test1.txt',true); xmlhttp.send();
```

(2) 原生 Ajax 实现: POST

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = function(){ if (xmlhttp.readyState==4 &&
xmlhttp.status==200){
document.getElementById('myDiv').innerHTML=xmlhttp.responseText; \} \}
xmlhttp.open('POST','ajax\_test.asp',true);
xmlhttp.setRequestHeader('Content-type','application/x-www-form-urlencoded');
xmlhttp.send('fname=Bill&lname=Gates');
```

reference: [原生 JS 与 jQuery 对 AJAX 的实现](#), [AJAX - 向服务器发送请求](#)

### Ajax 的 5 个状态 (readyState)

(1) 0: 请求未初始化

此阶段确认 XMLHttpRequest 对象是否创建，并为调用 open() 方法进行未初始化做好准备。值为 0 表示对象已经存在，否则浏览器会报错——对象不存在。

(2) 1: 服务器连接已建立

此阶段对 XMLHttpRequest 对象进行初始化，即调用 open() 方法，根据参数 (method,url,true) 完成对象状态的设置。并调用 send() 方法开始向服务端发送请求。值为 1 表示正在向服务端发送请求。

(3) 2: 请求已接收



此阶段接收服务器端的响应数据。但获得的还只是服务端响应的原始数据，并不能直接在客户端使用。值为 2 表示已经接收完全部响应数据。并为下一阶段对数据解析作好准备。

(4) 3: 请求处理中

此阶段解析接收到的服务器端响应数据。即根据服务器端响应头部返回的 MIME 类型把数据转换成能通过 `responseBody`、`responseText` 或 `responseXML` 属性存取的格式，为在客户端调用作好准备。状态 3 表示正在解析数据。

(5) 4: 请求已完成，且响应已就绪

此阶段确认全部数据都已经解析为客户端可用的格式，解析已经完成。值为 4 表示数据解析完毕，可以通过 `XMLHttpRequest` 对象的相应属性取得数据。

reference: [Panda, Ajax readyState 的五种状态, 2012](#)

## 1.9 文档对象

### 浏览器事件冒泡和捕获

事件分为三个阶段:

- 捕获阶段
- 目标阶段
- 冒泡阶段

TODO:IE 和 w3c 标准的区别;阻止事件传播(捕获,冒泡)`e.stopPropagation()`;  
阻止事件默认行为 `e.preventDefault()`。

```
function registerEventHandler(node,
event, handler) \{ if (typeof node.addEventListener == 'function')
node.addEventListener(event, handler, false); else
node.attachEvent('on' + event, handler); \}
```

```
registerEventHandler(button, 'click', function()\{print('Click
(2)');\});
```

reference: [本期节目, 浏览器事件模型中捕获阶段、目标阶段、冒泡阶段实例详解, segmentfault, 2015.8](#)

## 1.10 浏览器对象

### 弹框

- `prompt(text,defaultText)` 提示用户输入的对话框。`text` 对话框中显示的纯文本, `defaultText` 默认输入文本。返回值为输入文本。
- `alert(message)` 警告框。`message` 对话框中要实现的纯文本。
- `confirm(message)` 显示一个带有指定消息和 OK 及取消按钮的对话框。`message` 对话框中显示的纯文本。点击确认返回 `true`, 点击取消返回 `false`。

reference: [http://www.w3school.com.cn/jsref/met\\_win\\_prompt.asp](http://www.w3school.com.cn/jsref/met_win_prompt.asp)

### localStorage

存储在浏览器中的数据, 如 `localStorage` 和 `IndexedDB`, 以源进行分割。每个源都拥有自己单独的存储空间, 一个源中的 Javascript 脚本不能对属于其它源的数据进行读写操作。

`localStorage` 在当前源下设置的值, 只能在当前源下查看。

```
localStorage.setItem('key',value) //
设置值, 或 localStorage.key = value localStorage.getItem('key') //
获取值, 或 localStorage.key localStorage.removeItem('key') // 删除值
localStorage.clear() // 清空 localStorage
```

reference: [JavaScript 的同源策略, MDN](#), [localStorage, MDN](#), [杜若, localStorage 介绍](#)

## 1.11 jQuery

### jQuery 和 Dom 对象相互转换

(1) jQuery 对象转换成 Dom 对象: 下标和 `get` 方法

```
/* 方法一: 下标 */ var $v = $('#v')
; //jQuery对象 var v = $v[0]; //DOM对象
```

```
/* 方法二: get */ var $v = $('#v'); //jQuery对象 var v =
$v.get(0); //DOM对象
```

(2) Dom 对象转 jQuery 对象: 通过 `$()` 把 Dom 对象包装起来实现

```
var v = document.getElementById('v');  
//DOM对象 var \v = \$(v); //jQuery对象
```

reference: [jQuery 对象与 dom 对象的区别与相互转换](#){:target='\_blank'}

### jQuery 知识结构

- jQuery 基础: 选择器
- jQuery 效果: hide, show, toggle, fadeIn, fadeOut, animate
- jQuery 操作 HTML: text, html, val, attr, append, after, prepend, before, remove, empty
- jQuery 遍历 Dom: parent, parents, children, find

jQuery 和其他 js 库共存, 使用 `jQuery.noConflict()` 覆盖 `$`。 [更多](#)

## 1.12 其他

1. switch 以 `===` 匹配。
2. 函数会首先被提升, 然后才是变量。