

1. 显示数据

```
window.alert() 弹出警告框  
document.write() 写进 HTML 中 (只能在 HTML 中, 若在脚本中写则会覆盖整个文档)  
innerHTML 写入 HTML 文档中  
console.log() 浏览器控制台.
```

2. JS 代码编写位置:

(js 可以放在 `<head>` 或 `<body>` 里, 通常放在 `<head>` 或页面底部.)

- 写在标签属性里面 (x)
  - e.g. `<button onclick="alert('失败');">失败</button>`
  - `<a href='javascript:失败;'>失败</a>`
- js 写在 script 标签里 (x)
  - e.g. `<script type="text/javascript"> alert("失败");</script>`
- js 放在外部文件 (v)

e.g. `<script type="text/javascript" src="script.js"></script>`

注: 一旦使用外放, 就不能编辑. 即使写也会略  
若需要修改, 在创建一个新别的 script.

3. break: 退出循环 / switch 语句; continue: 退出循环.

S - 可以为循环创建一个 label.

使用 break label 终止指定循环.

continue label 跳过当前, 继续循环.

e.g. outer:

```
for (var i=0; i<5; i++) {  
    console.log(i);  
    for (var j=0; j<5, j++) {  
        break outer;  
        console.log(j);  
    }  
} // output: 0
```

4. js: S - 压缩大小写

- 忽略空格和换行

- 每一句后加分号.

5. unicode {字符串中应用. 以四位编码  
网页 --- : 8半 - - (10进制)}

6. 条件运算符: 条件表达式 ? 赋值1 : 赋值2

`var a=30, b=20, c=10;`

`var max = a>b ? (a>c ? a : c) : (b>c ? b : c)`

7. 优先级: 与比较高. \*

8. 内存溢出

· 程序机理错误.

· 需要内存超过剩余内存

内存泄漏.

占用内存没有及时释放

内存池过大 → 内存溢出

全局的泄露.

· 外部全局变量, 没有用 var

· 没有及时清理计时器或回调

· 闭包.



5. 其他进制 | 16 - 0x, 8 - 0, 2 - 0b (不是所有浏览器都支持)  
e.g. a = "070"; (有的浏览器认为8, 有的认为10)  
(Sol.) a = parseInt(a, 10); // 70.

var my = 128;  
my.toString(16); // 80  
my... (8); // 1000  
----- (2); // 100000000

6. 运算符: { - 二元: + - \* / % (余数)  
(只要是二元, 都有字符串的特殊处理, 不然直接忽略)  
一对于非 Number, + \* / % 转成 Number  
- 0 与 NaN 作运算  $\Rightarrow$  NaN  
✓ + 两个字符串相加, 0 与字符串相加, 都会转成字符串

-- 一元: (Number) \*

- 正号: 对正号转成 number.
- 负号: 符号取反.
- 对于非 Number 的值, 会先转成 Number, 然后作运算

e.g. var a = true; a = -a; // a = -1  
var a = "18"; a = -a; // a = -18  
a = +a; // a = 18 ✓

- 自增: 原值会立即变化. \*  
: a++ 旧值 + +a 新值 (自增)

e.g.: var d = 20;  
var res = d++ + +d + d; 20 + 22 + 22 = 64

- 逻辑运算符: ! :

QQ: ! 为 false, 不 check 2 ✓

11: ! 为 true, ? : check 2 ✓

! : 对于非布尔, 转为布尔;

QQ, ! : 对于非布尔, 转布尔, 返回原值 \*

e.g.: true || alert(); // 跳过  
false || alert(); // 打印: var res = 5 & & 6 // 6  
false && alert(); // 不打印: var res = NaN && 0 // NaN

- 关系运算符 (>, <, <, >, ==, ===)

- 对于非数字, 转成数字.

- != NaN  $\Rightarrow$  false 字符串 / undefined  $\Rightarrow$  NaN

- 若两边都是字符串, 可用 + 转 Number.

比的时候一位一位比,

e.g. "11" < "5" // true;

"111" < "5" // true;

.. "111" < +'5' // false; 可用 + 转 Number.

- == "会自动类型转换".

undefined 转成 null:

undefined == null; \*

- NaN == NaN; // false.

- "123" == 123; // false. \*  
会转为不同类型的值

e.g. "123" == 123; // false.  
undefined == null; // false. \*

## 1. 流程控制语句:

```
- if (...) {
  else if (...) {
  } else {
  }
}
```

注: var age = 200

if (age > 100) ①	if (age > 30) ①
else if (age > 80) ②	else if (age > 60) ②
80 < age < 100	age > 60 & age > 30
1 执行, ①②	①②都执行

- num = 3

switch (num) {

case 1:

  alert('1');  
  break;

--

default:  
  alert('3');  
  break;

switch 中会使用恒等  
== 表示比较.

## 2. 对象 { - 内建

- 宿主对象: BOM, DOM, 浏览器对象: document, console.

- 自定义: 所有事物都可以是对象, 对象只是带有属性和方法的特殊数据类型

- 用 type 返回 object.

### // 创建对象:

① var obj = new Object(); ② var obj = {};

// 添加属性 / 修改属性.

① obj.name = "A"; obj.name = "B";  $\Rightarrow$  console.log(obj.name);

② obj["name"] = "A"  $\Rightarrow$  console.log(obj["name"]);

✓ // 检查属性: console.log("name" in obj).  $\uparrow$  错误, 因为 obj [var]

而 obj.var X.

## 3. 基本数据与引用数据类型

- 内存分配: 内存与堆内存

- JS 都放在栈内存

- 基本类型直接保存在栈内存里, 相互独立

- 对象级在堆内存里, 每创建一个对象会开辟新空间

左边放名称, 右边放堆内存地址.

如果两个变量指向一对象引用, 改变其中一个会改变另一个.

"==" 比较基本类型, 比较值.

比较引用类型, 比较内存地址  $\uparrow$  \*

e.g. var obj = new Object(); obj.name = "A";

var obj2 = obj; obj2.name = "B";

console.log(obj.name); // "B"

- - - (obj2.name); // "B"

obj2 = null // 设置为 null 之后与地址断开了联系.

console.log(obj); // object

- - - (obj2); // null

## 4. 声明函数 { ① var my = function() { ... };

② function my() { ... };

1. `toString()` - 当我们在页面打印一个对象时，输出是对象 `toString()` 方法的返回值；  
    - `toString()` 方法在实例对象的原型的原型中。  
    - 若希望输出不是 `[object Object]`，可以在实例添加一个 `toString()` 方法。  
        或在原型中修改 `toString()` 方法。

2. 垃圾回收 - 垃圾：对象没有变量/属性对它引用。此时我们无法操作对象。  
    - 回收：JS 自动回收，我们只需要将不再使用的对象设置为 `null` 即可。

3. 数组：  
    - 数组也是一个对象；  
    - 数组 [索引]，索引不存在，返回 `undefined`。  
    - 修改 `length`：`arr.length = 2`。  
        - 不修改 > 原本，多余会空。  
        - ... < ...，多余被删除。  
    - 创建数组 ① `var arr = new Array(1)`; ② `var arr = [ ]`       $\text{length}$       一个元素为 1。

- 数组的元素可以是任何类型  
- `pop()`: 移除最后一个元素，返回删除元素。  
- `unshift()`: 开头添加一个元素，返回新数组长度。  
- `shift()`: 移除第一个元素，返回删除元素。  
- `forEach()` 方法：将每一个函数作为参数。  
    IE8 以上才支持      因为我们创建，不由我们调用，叫回调函数。  
    IE8 以下用 `for` 循环、数组有几个元素就执行几次  
        浏览器会在回调函数传递三个参数。  
            第一个：当前遍历的元素 \*  
            第二个：当前遍历元素索引 \*  
            第三个：正在遍历的数组

e.g. `arr.forEach(function (value, index, c) { ... })`;

- `slice()`: 提取指定元素，返回新数组，不改变原数组。  
    参数：开始索引（包含）、结束索引（不包含）

    索引可以是负数，-1（倒数第一个）

- `splice()`: 删除/替换元素，影响原数组。  
    参数：开始索引，删除数量，元素（插入到开始索引前）  
e.g. `arr = [1, 2]`, `arr.splice(1, 0, '3')` / `res = [1, '3', 2]`

- join() : 将数组  $\Rightarrow$  字符串, 不影响原数组.

参数, 值搞错. e.g. var arr = ['1', '2']; var res = arr.join("a") // "1a2";

- reverse() : 反转, 影响原数组.

- sort() - 对于纯数字, 用 unicode 编码, 所以对数字排序可能出错.

- 自己定义排序规则:

在 sort() 添加一个回调函数.

- 回调函数需要定义的参数.

- 浏览器会分别使用元素作参考去调用回调函数.

- 使用哪个元素不确定, 但 a 一定在 b 前面.

e.g. arr = [5, 4, 2, 1, 3];

arr.sort(function(a, b){ return a-b;});

- call() 和 apply()

- 都是调用对象的方法, 需要函数对象调用.

- 调用调用 call(), apply() 都会执行.

- 在 call() 和 apply() 可以传递一个对象为第一个参数

此时这个对象将成为调用执行的 this.

- call() 可以将参数在对象之后依次传递.

- apply() 将参数封装为一个数组, 一位值.

e.g. fun.call(obj, 2, 3) fun.apply(obj, [2, 3]);

4. arguments {

- 是一个类数组对象, 可通过索引操作. 也可以获取长度
- 在调用函数时, 我们所传递的参数都会在 arguments 中.
- 即使不定义形参, 也可以通过 arguments[0]... 来访问
- 它里面有个 callee 属性, 其对应一个函数对象, 表示当前指向的函数

e.g. function fun(a, b){}

console.log(arguments instanceof Array); // true

---- . . . (Array.isArray(arguments)); // true.

---- - - (arguments[0]); // b

- - .. (arguments.length); // 2

- - - (arguments.callee = fun); // true.

?  
fun('Hello', true);

1. Date -> Date 对象(创建一个时间) new Date () ;

- 直接创建返回当前创建.

- 通过指定时间对象. var d2 = new Date ("12/03/2016");

- getDate : 几日

- get Day : 周几 (0是周日, ...)

- getMonth : 几月 (0是1月, ...)

- getTime : 时间戳, 指的是从格林威治时间的1970年1月1日, 00:00时开始, 到当前时间所花费的毫秒数.

- Date.now() : 当前的时间戳  $\Rightarrow$  测试代码执行的性能.

2. Math: - Math.floor : 向下取整

- Math.ceil : -- 上 --

- Math.round : 四舍五入

- Math.random : 生成 0-1 的随机数

3. 包装类 - String(), Number(), Boolean(), 利用此将基本类型转为对象.  
e.g. var str = new String(); (创建 String 对象会拖慢速度)

- 方法和属性不能添加给基本类型.

当我们对一些基本类型去调一些属性方法时.

浏览器会临时使用包装类对其进行操作对象, 然后回调方法.

调用完, 就转成基本类型.

4. 正则化 - 定义字符串的一些规则, 修饰符: i 不区分大小写, g 全局, m 多行.

- 创建正则化表达式对象 e.g. var reg = new RegExp("a", "i") = var reg = /a/i;

- 用 test 检查是否符合表达式. e.g. var res = reg.test('a', 'i'); // true;

- 特例: [a-b] - 大小[A-Z], b[1-2], b[0-9] e.g.: reg = /\arabic/c/;

- [^] 除了: reg = /[^ab]/, reg.test("abc") // true.

- 其他字符串: ① split var str = "123b"; res = str.split(/\w+/); // "1", "2", "3"

(即使不指定全局, 也会拆分) (经典进阶)  
② search var str = "hello abc eos", res = str.search(/\atbe/g); // 16  
(只会拆分一个, 指定了全局也适用)

③ match var str = "123b", res = match(/\w+/); // "1" (遍历数组用)

e.g. replace. var str = "123a", res.replace("aa", "aa"); // "1aa2aa"

reg.replace(/\w+/i, "..."); // "1aa2aa"

- exec() 检索匹配. e.g. res.exec("the best thing are"); // "e".

1. 正则化语法 { 一旦使用构造函数创建正则对象时，需要带规范的转义字符，以下等价的 }

- 量词：  
  - $[n]$  正好 n 次
  - $[m, n]$  正好 m-n 次
  - $[m, \}$  m 次以上
  - $\{ \}$  至少一个
  - $\ast$  0 个或多个
  - $? / \dagger$  0 个或一个

- 检查开头  $^$

- --- 结尾  $\$$

- . 日字符串。

- 1. 表示点。 eg. `reg = /\./; reg.test('..');` //true  
`reg = /\//; reg.test('\'');` //false  
`reg = /\-/; reg.test('--');` //true  
`reg = new RegExp("\\.\\.")` ( $\in$  `reg = /\./;`)

- \w 日字母、数字。  $\rightarrow [A-Z0-9_]$   
\w 除了 - ---  $\rightarrow [^A-Z0-9_]$

\d 日数字 \D 除了 \w 字符 (1b) 单词边界

\s 空格 \S 除了空格 (1b) 单词边界

2. //检查是否含有 child。  
`reg = /\b child\b/; console.log(reg.test('children'));`  
`//false`  
 //去除字符串前后空格。  
`str = str.replace(/\s*/\s*/g, '')`

2. 作用域 - 可访问变量、函数、对象的集合 : `for (var i = 0; i < 10; i++) { };`

3. constructor 属性 { 返回变量的构造函数。 }

eg. `"John".constructor`  $\Rightarrow$  `String()`

`(3.14).constructor`  $\Rightarrow$  `Number()`.

//检查对象是否为数组。(包含字符串 "Array")

`MyArray.constructor.toString().indexOf("Array") > -1;`

eg. `fun.prototype.constructor == fun;` // true.

原型中 `constructor` 指向函数对象

1. this } 解析器每次调用函数都会向函数内部传递一个隐含参数。  
一、这个隐含参数叫作 this, this 指向一个对象  
二、这个对象我们称为上下文对象  
三、根据函数的调用方式不同, this 指向不同对象。

1. 以函数的形式调用 → 调用所属者 (window). 在严格模式下, 函数返回值指向 this.
2. 以方法的形式: this → 调用方法的对象. : this => undefined
3. 以构造函数的形式调用时, this 就是创建的对象
4. 使用 call / apply 调用时, this 是指定的那个对象 → 若第一个参数为对象, 也会指向 this
5. 在 HTML 事件句柄时, this 指向了接收事件的 HTML 元素. 非严: 和 -> null / undefined. this 指向全局.

2. 构造对象 } 一、构造函数首字母大写:

- 使用同一个构造函数创建的对象, 我们称为一个类. 也将一个构造函数称为一个类.
- 将新建的对象设置 this, 在构造函数中可以使用 this 引用新建对象
- 使用 instanceof 检查实例: per instanceof Person; // true.
- 所有对象都是 Object 后代: per instanceof Object; // true.
- 构造函数每执行一次, 就会创建一个 新的 sayName 方法.

e.g. function Person( name, age, gender ) {  
 Person.prototype.name = "Eng";  
 this.name = "A";  
 ...  
 this.sayname = function () {  
 ...  
 };  
}

已存在的对象调用器不能添加新的属性  
Person.prototype.name = "Eng";  
(√)

3. 原型 {  
一、创建的每一个函数, 解析器都向函数添加一个 prototype 属性.  
这个属性对应一个对象, 叫原型对象, 所有对象都位于原型链顶端的 Object 的实例。  
二、如果函数以普通函数的形式调用, prototype 会起作用。  
以构造函数: e.g.: mc.\_\_proto\_\_ = MyClass.prototype.

三、原型对象相当干一个公共区域, 所有实例都可以访问。

我们将对象中没有的内容, 放在原型中. e.g. MyClass.prototype.a = 123

- 原型链 = } (闭关)  
一、当我们访问对象的方法/属性时, 自身有, 用自身, 自身没有, 去原型对象中寻。  
二、用 in 检查是否有此属性, 若对象无, 原型有, 也返回 true. e.g: "name" in mc;  
三、可以用 hasOwnProperty 检查 自身 的属性: e.g. mc.hasOwnProperty("name");  
四、原型也有原型, 检查属性, 若自身无, 则一直向上寻, 直到找到 Object 对象的原型 Object 对象的原型没有原型, 如果在 Object 没找到, 则返回 undefined.



1. 错误：``S - throw``：当事情出问题时，js 抛出一个错误。

- `try`：允许我们定义在执行时进行错误测试的代码，`try` 和 `catch` 成对
- `catch`：捕捉错误，并执行相关代码。
- `finally`：不论 `try` 和 `catch` 是否出现异常都执行。

e.g. `function my () {`

`var mes, x;`

`mes = document.getElementById ("p01");`

`mes.innerHTML = "";`

`x = document.getElementById ("demo").value;`

`try`

`if (x == "") throw "值是空的";`

`if (x > 10) throw "值太大";`

`}`

`catch (err)`

`mes.innerHTML = "错误" + err + ":";`

`}`

`finally`

`document.getElementById ("demo").value = "";`

`}`

`1`

2. 变量提升：- js 中，所有声明都会被提前。所以可以先使用后声明。

`var x; x=5; ⇔ x=5, var x;`

- 初始化不会提升。（不管是变量、函数、多初始化的都不会提升）

`var x=5; console.log(x); // 5 vs. console.log(x); var x=5; // undefined.`

（`x` 提升了，但 `x=5` 不提升，∴ 结果为 `undefined`）

3. 严格模式：- `"use strict"` 不是一条语句，而是一个字面量表达式。

(`Q2`) 在此模式下不能使用未声明的变量 / 对象。

不允许声明变量 / 对象 / 函数，不允许使用八进制 / 转义符。

不允许对只读属性赋值，不允许使用 `getter()` 读取的属性进行赋值。

\* 作用域已 `var()` 创建的变量不能被访问。e.g. `function f() { function f1() { "use strict"; return this; } this.a=1; }`

\* `this` 禁止指向全局对象：

均为 `undefined`。

`f(); // 报错`

## 1. JS 表单

? { HTML 表单验证可以通过 JS 完成。

```
// 判断单字段 (fname) 值是否存在  
⇒ function val () {  
    var x = document.forms["myform"]["fname"].value;  
    if (x == null || x == "") {  
        alert ("需要");  
        return false;  
    }  
}; 以上代码在 HTML 可被调用
```

```
<form name = "myForm" action = "demo_form.php" onsubmit = "return val()"  
method = "post" >名字：<input type = "text" name = "fname">  
<input type = "submit" value = "提交" ></form>
```

- HTML 表单自动验证，若表单字段 fname 空白，required 属性会阻止提交

```
<form action = "demo_form.php" method = "post" >  
    <input type = "text" name = "fname" required = "required" >  
    <input type = "submit" value = "提交" >  
</form>
```

## 2. HTML 约束验证

?

- HTML 输入属性：disabled, max, min, pattern, required, type  
- CSS 伪类选择器：disabled, invalid, optional, required, valid  
- DOM 属性方法：- checkValidity() 如果 input 合法, 返回 true。  
(验证 API)  
- setCustomValidity() 设置 input 元素的 validationMessage 属性，用于自定义错误提示信息的方法。

\* 使用 setCustomValidity() 设置自定义提示后，validity.customError 会变成 true，而 checkValidity() 会返回 false。如果需要重新判断时需取消自定义 ⇒ setCustomValidity("")

setCustomValidity(null)

- 约束 DOM 属性

- validity 属性值	- - - - - (undefined)
- validationMessage 浏览器提供错误信息	
- willValidate 是否需要验证	
- validity 状态属性	

customError, patternMismatch, rangeOverflow, rangeUnderflow, stepMismatch, tooLong, typeMismatch, valueMissing, valid

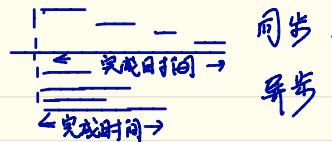
- ★ let, const {
- Const 只读常量，必须初始化且先声明后使用，不可修改，但可读
  - let (先声明后使用)
    - e.g. `var x=2;` // 这里可以使用 x
    - `let x=2;` // 这里不 --
    - `var i=5;`  
`for (var i=0; i<10, i++)`  
// 这里 i=10.
    - `let i=5;`  
`for (let i=0, i<10, i++)`  
// 这里 i=5
  - let 所声明的全局作用域不属于 window.
  - var - - 属于 -- .
  - 相同作用域 / 深级作用域。let 不能重置 let / var

2. JSON {
- 用于存储和传输数据的格式。(轻量级的数据交换格式) JavaScript Object Notation
  - 通常用于服务器向网页传送数据
  - 独立语言，易于理解，特殊格式的字符串，可以被日语识别，并转换为日语中对端
  - 语法规则
    - 数据为键值对。
    - 数据由逗号隔开
    - 大括号保存对象
    - 小括号 --- 数组
  - 内置函数 JSON.parse() 将JSON字符串转为JS对象
    - JSON.stringify() 将JS对象转为JSON对象(IE7以下不能用)
    - eval：执行一段字符串形式的代码，并将执行结果返回。若字符串中有引号，会将其当成代码块，不希望如此，可在字符串前后加()。(不建议使用)

3. Void {
- 指定计算一个表达式但不返回值。
  - href="#" 与 "javascript:void(0)" 的区别
    - e.g.: // 阻止链接运行，URL 不会有变化
    - `<a href="javascript:void(0)" ref="nofollow nfc>点</a>`
    - // ----- URL 尾部多加 #，改变了URL(并同时配合 location.hash)
    - `<a href="#" ref="nofollow nfc>点</a>`
    - // ? 也可以达到阻止效果，但多加 ? (?) 用于配合 location.search
    - `<a href="?" ref="nofollow nfc>点</a>`
    - // Chrome 浏览器 javascript:0 也没变化，但 firefox 中会变成一个字符串
    - `<a href="javascript:0" ref="nofollow nfc>点</a>`

# 1. 异步

- 框架



一何时：主线：短，快速

子线：消耗时间长：读取一个大文件 / 发出一个网络请求 (一旦发起，主线失去同步)

一通过回调函数来实现异步任务结果处理

一回调函数：它是在我们启动一个异步任务的时候来消费它，结束后再平均。

e.g. ① function print() { document.getElementById("demo").innerHTML = "Hello";  
setTimeOut(print, 300);  
} ⇔ setTimeOut(function() { ... }, 3000);  
(这个程序执行之后会产生一个子线程，子线程等待3秒，执行回调函数)  
② setTimeOut(function() {  
console.log("1");  
}, 1000);  
console.log("2"); // 2 1

一除了 setTimeout 之外，异步回调还广泛应用于 AJAX 编程。

一 XMLHttpRequest 常常用于请求向远程服务器上的 XML 或 JSON 数据。

e.g. var xhr = new XMLHttpRequest();  
xhr.onload = function() {  
document.getElementById("demo").innerHTML = xhr.responseText;  
}; // 将接收到的数据  
xhr.onerror = function() {  
}; // = “请求出错了”;

发送异步 GET 请求。

xhr.open("GET", "http://www...true),  
xhr.send();

onload / onerror 属性都是函数，如果使用jQuery库，可以更好的使用异步 AJAX；

\$.get("https...", function(data, status){  
alert(数据...);  
});

1. Promise {  
- 构造 Promise 对象 new Promise (function (resolve, reject) {  
}) ;

- 原因：若串调用多次异步：第一次间隔 1 秒，第二次间隔 4 秒

```
setTimeouts(function() {
    console.log("1");
    setTimeout(function() {
        console.log("2");
    }, 4000);
}, 1000); // (虽然快印)
```

设置：Promise 将嵌套代码变成了顺序格式的代码。

```
new Promise (function (resolve, reject) {
```

```
    setTimeout(function() {
        console.log("1");
        resolve();
    }, 1000);
}).then(function() {
    setTimeout(function() {
        console.log("2");
    }, 4000);
});
```

- 使用：Promise 只有一个参数，是一个函数，起始函数，构造出的被异步进行这个函数有两个参数， resolve, reject，这两个参数皆为函数，调用 resolve 为通过一切正常，出现异常时调用 reject。

- 方法：有 then(), catch(), finally 三个方法。这三个方法的参数都是一个函数。  
.then() 可以将普通中错误直接添加到当前 Promise 的正常执行序列  
.catch() 没次 Promise 异常处理序列

.finally() 执行后一定执行的序列

.then 顺序执行，有且异常跳到 catch

e.g. new Promise (function (resolve, reject) {  
 console(1); resolve(2);

```
}).then(function(value) {  
    console(value); return 333; // 1  
}).then(function(value) {  
    console(value); // 2
```

```
}).then(function(value) {  
    console(value); throw "An error"; // 333
```

```
).catch(function(err) {  
    console.error(err);  
}).
```

An err

- Promise 也很将“计时器”程序核心部分写成函数。

```
function print(delay, messages) {
    return new Promise(function(resolve, reject) {
        setTimeout(function() {
            console.log(messages);
            resolve();
        }, delay);
    });
}
```

⇒ print(1000, '1').then(function() ⇒ **异步函数**
 return print(4000, '2'); await print(1000, '1');
 )).then(function() {
 print(3000, "Third");
 });
})

异步函数

async function asyFunc()

- - - - 4

- - - - 3

async/fun();

- QoA ① catch(), finally(), then(). 可重叠，但不推荐
- ② 除了 then() 其他也可以多次使用，但 catch 只执行第一个。
- ③ return 不能中断，then() 可以用 throw 来跳转到 catch 实现中断
- ④ 什么时候用异步：多次顺序执行异步，如先查询用户名/密码。
- ⑤ Promise 不将异步词，只是一种编程风格
- ⑥ 什么时候异步写一个 then，又得调用另一个异步任务。

## 一 异步函数 + Promise. (异步异常处理)

```
--- () {
let value =
await new Promise(function(resolve, reject) {
    try {
        await new Promise(function(resolve, reject) {
            throw "Some Error"; // or reject("Some error")
        });
    } catch(err) {
        console.log(err);
    }
});
console.log(value);
asyFunc();
}
```

## 1. 代码规范

- { -四个空格
- 全局变量 / 常量为大写
- HTML 可以用 data- 前缀 ✓
- CSS 用连接符 ✓
- 对象最后一个属性值不带加逗号 ✓

## 2. 反向提升

- { - 提早：JS 默认将当前作用域提升到前面去的行为
- 反向提升：函数表达式 var n = func --- 不能提前  
而 function my() {}

## 3. 自调用函数：(function() { var x = 1; })(); 隐名函数的自调用

## 4. typeof 函数 → return function.

## 5. 箭头函数 { - 格式：(参 1, ..., 参 n) => [函数声明]

(参 1, ..., ) => 单一表达式

(参 1) => [函数声明] [C 可选]

() => [函数声明].

e.g. var n = function(x, y) { return x + y; } (es5)

const x = (x, y) => x + y;

## 6. { 有的箭头没有自己的 this, 不适合定义为一个对象的方法 箭头中 this 值与外层的 this 是一样的 }

箭头不能提升，声明后使用

使用 const 比 var 更安全，因为函数表达式始终是一个常量；

如果函数部分是 { 一个语句 } 可以省略 return 加 ()，是一个坏习惯。

## 6. 函数参数： - 显式参数 (Parameters)、隐式 (Arguments)

显式参数是在函数定义时列出的

隐式参数是传递的实正值。

### - 参数规则

显式参数不指定参数类型。

对隐式参数不会类型检测

----- 个别，会检测

### - 显式参数：undefined.

为了避免，可以自带参数，即 function(x, y = 10) {}

隐式参数的改变在函数外不可见

# 1. prototype 和 -proto- 属性

① prototype 是函数才有的属性.

② -proto- 是每个对象都有的属性. 但 -proto- 不是规范属性, 只有部分浏览器实现. (在标准属性为 \_\_Prototype\_\_)

注: 大多数情况下, -proto- 可以理解为 "构造器原型", 即

$--proto-- == constructor.prototype$ . 通过 Object.create() 实现

① var a = {}

a.prototype // undefined

a.proto-; // object [ ]

② var b = function () { }

b.prototype; // b [ ]

b.\_\_proto-; // function [ ]

## 2. proto\_ 指向取决于对象创建时的实现方式

① 字面量.

var a = {};

object a

-proto- →

function object

prototype

② 构造器

var A = function () {};

object a

-proto- →

function A

prototype

③ Object.create

var a1 = {}

var a2 = Object.create(a1);

object a2

-proto- →

object a2

④ var a = {}

a.\_\_proto\_\_ // object [ ]

a.\_\_proto\_\_ = a.constructor.prototype // true

⑤ var A = function () {}

var a = new A();

a.\_\_proto\_\_ // A [ ]

⑥ var a1 = {};

a1.\_\_proto\_\_ // Object [ ]

var a2 = Object.create(a1);

a2.\_\_proto\_\_ == a1 // true

## 3. 原型链. { 由于 -proto- 是对象都有的属性. 所以 -proto- 链条. }

指向对象本身是否存在该属性. 如果没有, 在本原型链查找, 但不会查找自身的 prototype.

var A = function () {},  $\Rightarrow a.__proto__.proto__: null$

var a = new A();



产生：在函数内部声明的变量执行完时就产生了（不是全局的）过程声明提存  
死亡：在函数内部声明的局部变量还存在，一般的不存在，闭包的局部变量不能被访问

## 1. 闭包

- 和全局变量用到闭包

函数内变量声明不用 var 关键字，就是全局变量了。--- 是函数内部和外部的桥梁

- 变量生命周期

全局变量的作用域是全局的。

1. 闭包可以访问其他函数内部变量

2. 闭包是一个函数内部的函数

4. 用途：除1，还可以让这些变量

在内存里。

而在函数内部声明变量，只在函数内部起作用。作用域是局部的（函数的参数）

e.g. 计算器困难 function add() {

var count = 0;

return ++count;

}

add();

add();

add(); // (本想输出3，但又不想在全局定义 count)

- 内嵌函数：

所有函数都能访问全局变量 / 上一层作用域

嵌套函数也可以访问上一层函数变量。

内嵌函数 plus() 可以访问父函数 counter: e.g. 闭包产生函数提升

若能在外部访问 plus 函数，就解决了计算机困难；若 var f = plus();

确保 var counter = 0 执行一次  $\Rightarrow$  need 闭包) 这里产生闭包

counter += 1;

return plus();

var f = add();

f(); f();

f = null; // 闭包死亡。

- 函数自我调用。

var add = (function() {

var counter = 0;

return function() {return counter += 1;} |

)();

add(); add(); add(); // 3.

大解构：变量 add 指定 函数自我调用的返回值。

自调用只执行一次，设置计数器为 0，并返回表达式。

add 变量可以当作一个函数使用，因为在它可以访问上一层作用域计数器。

这个叫 闭包，也使得函数拥有全局变量变为可能。

计算器使匿名函数的作用域保护，只能通过 add 停止。

- 链式作用域结构；父对象所有变量，子对象皆可见，所以上不成立。

{ 函数执行完后，函数内部声明的局部变量是否还在？一般的不存在，闭包的局部变量不能被访问  
但在函数外部能直接访问内部局部变量吗？No，但可以通过闭包让外部操作它。  
总结：函数执行完后，函数内部局部变量自动释放，占用内存时间较少，造成内存泄漏。  
单线程：函数执行完后，能不释放不同

# 1. DOM (Document Object Model)

## 1. html

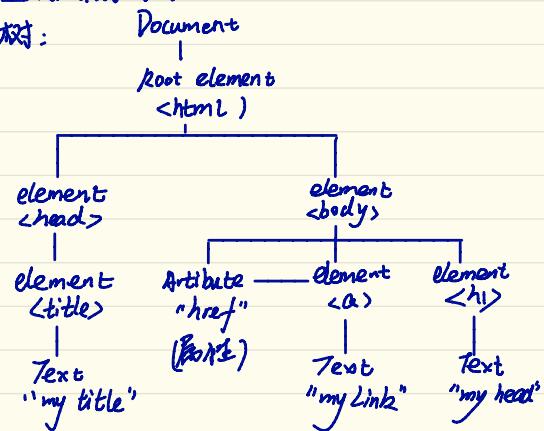
```

<html>
  <head>
    <title>标题 </title>
  </head>
  <body>
    <a href="1.html">链接 </a>
  </body>
</html>

```

— 被构造成为树的对象.

- DOM树:



- JS能够改变页面的所有HTML元素/属性.

--- CSS样式

--- 事件作出反应

## 2. 节点 (最基本组成部分, 每部分都可以称为一个节点).

e.g. HTML 标签、属性、文本、注释、整个文档等都是一个节点... (但具体类型不同).

一级分类: 文档节点: 整个HTML文档

元素节点: HTML 标签

属性节点: 元素的属性.

文本节点: HTML 标签的文本内容.

<P id="pid"> This is </P> 属性节点- 元素节点.

- 属性:

	nodeName	nodeType	nodeValue
文档节点.	#document	9	null
元素节点.	标签名	1	null
属性节点.	属性名	2	属性值
文本节点.	text	3	文本内容

- 浏览器已经为提供了文档节点为 window 属性, 化表单对象.

④⑤返回的是 HTML collection 对象，有 length，有下标...  
但不是数组，不可以用 push, pop, join ... (元素集合)

## 1. 查找 HTML 元素 { 一方法 ① 通过 id 找到一个 HTML 元素； id 自己设定

（单个页面）  
只有一个元素时。

var x = --- ("") [0]

有多个... 时

var x = --- ("") [i]

② 通过标签名，找到一组元素：img, button, input ...

var x = document.getElementsByTagName("main");

var y = x.getElementsByTagName("p");

③ 通过类名，找到一组元素 (IE8 以下不支持)

var x = document.getElementsByClassName("demo");

## 2. 漏读 HTML - 漏读 HTML 端出错

(绝对不要在文档 (DOM) 加载完成之后使用 document.write,

这覆盖整个文档)

- 漏读 HTML 内容。

document.getElementById(id).innerHTML = 新文本 (对自动换行的标签，

- 漏读 HTML 属性。

(class 属性写 className) 这个属性连着 e.g. input

document.getElementById(id).attribute = 新属性值；

e.g. 

<script> document.getElementById(id).src = "land.jpg";</script>

## 3. 事件 { 一用户和浏览器之间的交互行为。比如：点击按钮，鼠标移动 ...

在一事件对应的脚本中设置一些 JS 代码，事件被触发，代码会执行。

为按钮的对应事件绑定处理函数。

一分配事件。

{ HTML 元素：<button onclick="displayLabel()">点</button>

DOM <script> document.getElementById("demo").onclick = function() { displayLabel(); }</script>

<script>  
function change (id){  
 id.innerHTML = "oops";  
}</script>

<head>  
<body>  
<h1 onclick="change(this)">点</h1>

</body>

⇒ <body>  
<h1 onclick="this.innerHTML = 'oops!'">点</h1>  
</body>

1. 及时的加载 {
- onload 在页面加载完成之后执行
  - 为 window 预定一个 onload.

HTML Collection	Node List
<ul style="list-style-type: none"> <li>name, id, 类名</li> <li>length</li> </ul>	<ul style="list-style-type: none"> <li>索引</li> <li>length</li> <li>包含属性及文本节点</li> </ul>

该事件对应的情况应该是将代码放在页面加载之后执行  
这样可以确保我们的代码执行时所有的 DOM 元素加载完毕。

e.g. <head>

<script>

window.onload = function () {

① var btn = document.getElementById("btn");  
② btn.onclick = function () { alert("A"); };

</script>

</body>

<button id="btn">点我</button>

</body>

错 ② 没有放在 onload 里, 所以 btn = null, 报错出错, 因为 button 没有加载出来)

## 2. 获取元素节点的子节点 - 通过具体元素节点调用

- getElementsByName() 当前节点, 返回标签后代节点
- childNodes 所有节点 → 返回 NodeList 对象
  - 一包括文本节点在内的所有节点
  - 一根据 DOM 标签标签间的 空格 也会当成文本节点
  - 一 Note: 在 IE 8 浏览器以下是否包括空格.
  - children 获取当前元素的所有子元素.
- firstChild 第一个子节点. (包括空白文本)
  - firstElementChild 获取当前元素的第一个子元素 (不支持 IE 8 及以下)

- lastChild 最后 ... ---

- parentNode 当前节点父节点

- innerText (与 innerHTML 类似, 不同的是它自动去除标签)

- previousSibling (获取空白文本)

- previousElementSibling (IE 8 以下不支持)

- nextSibling

① 文本框 value 属性值, 如是填写的内容

② by.firstChild.nodeValue . 获取 p 中的文本节点.

## 3. 获取文节点及兄弟节点

- DOM 查询的剩余方法
  - 获取 body 根节点: `document.body`;
  - 获取 html 根节点: `document.documentElement`;
  - 所有元素: `document.all`
  - `document.getElementsByTagName("x")`
  - `document.querySelector()`
    - 需要一个选择器语句字符串作为参数, 可以根据 CSS 选择器进行查询一个元素节点对象.
    - ~~IIS 没有 `getElementsByClassName`, 但可以使用~~
    - 若满足条件元素有多个, 那么它只会返回第一个
    - e.g. `var x = document.querySelector(".box")`
  - `document.querySelectorAll()`  $\rightarrow$  大部分返回 NodeList 对象
    - 满足所有条件的元素放到一个数组里
    - 即使符合条件的只有一个也返回数组.

## - DOM 增删方法

- 创建元素和节点: `document.createElement()`
  - 参数: 标签名,
  - 他会根据标签名创建节点, 并将创建的对象返回.
- 创建文本节点: `document.createTextNode()`
  - 参数: 文本内容.
  - 将会根据文本内容创建节点... 将折能的节点(变回).
- 向一个父节点添加一个新的子节点: `appendChild()`
  - e.g.: `var li = document.createElement("li");  
var g2 = ----- createTextNode("g2");  
li.appendChild(g2);  
var city = document.getElementById("city");  
city.appendChild(li);`
- 插定节点: 前插入新节点: `insertBefore`
  - 参数 1: 新节点, 参数 2: 回节点.
  - 文节点引用:
- 移除节点, 知道父节点: `parent.removeChild(child)`
  - (早期浏览器不支持)

- 替换节点: `replaceChild()`
  - 参数 1: 新节点, 参数 2: 替换节点.
- 使用 `innerHTML` 也可以完成 DOM 的增删改的操作.
  - 一般结合使用.

① `var city.innerHTML += "<li>" + str + "</li>"`;

② `var li = document.createElement("li");`

`li.innerHTML = " " + str;`

`city.appendChild(li);`

## 1. 改变HTML样式

- style property = ~~字符串~~ eg box1.style.width = "300px";

- 如果CSS样式中包含 -

在JS不合法

修改：去掉 -，然后将 - 后的字母大写

- 我们通过 style 设置的是内联样式

而内联样式有较高的优先级，往往立即将显示

- 如果在样式中写了 !important，会拥有最高优先权  
即使通过JS修改也没用。

尽量不要写 !important。

- 读取内联样式 style property

通过 style 属性设置和读取的都是内联样式。

无法读取样式表的样式

- 获取当前显示样式：

① currentStyle() 只支持IE

② getComputedStyle()

- IE8 以下不支持

- 方法 1：获取样式元素

- -- 2：可以传递一个伪元素，一般为 null

- 语义对称，封装当前元素对应的样式

obj = var obj = getComputedStyle(box1, null);  
alert(obj.width);

若果取样式没有设置，获取真实值却不是 auto。

② 读取到的样式都是只读的。

不能修改，而修改必须通过 style 属性。

- 获取元素的可见宽度和高度：clientWidth, clientHeight

- 这些属性返回数字，可以直接计算

- 包括内容区和内边距 (padding)

- 只读，不能修改。

- 获取元素的客户宽度和高度：offsetWidth, offsetHeight

- 包括内容区、内边距和边框 (border)

- 获取当前元素的父辈元素：offsetParent

- 返回离当前元素最近定位的祖先元素

- 如果所有祖先元素都使用绝对定位 body

- 当前元素相对于其定位父元素的水平偏移量：offsetLeft

-----垂直----- offsetTop.

- 当取整个滚动的高度和宽度：scrollWidth, scrollHeight

- 可以获取水平/垂直滚动距离

when scrollheight - scrollTop = clientHeight  
说明滚动条到底。

滚动条样式 CSS 中 overflow = auto;

滚动条触发：元素 onscroll

disabled 设置元素是否禁用

## 1. 事件 { - 事件对象 .

- 当事件的响应函数被触发时，浏览器都会将一个事件对象作为实参传递给响应函数
- 在事件对象封装了当前事件的一切信息，e.g. 鼠标的坐标
- IE8 不会传递事件对象；
- IE8 及以下是将事件对象作为 window 对象属性保存的；直接用 window.event 处理 .
- onmousemove
  - clientX 鼠标指针的水平坐标（相对视窗的）
  - clientY --- 垂直 ...
  - pageX 鼠标指针的水平坐标（相对整个页面的）
  - pageY --- 垂直 ...  
(这两个属性 IE8 及以下不支持)
- chrome 认为滚动条是 body 的 .document.body.scrollTop .  
Firefox --- 是 html 的 document.documentElement .
- e.g. document.onmousemove = function(event){}

问题的原因是因为 body 太高 ) → var st = document.body.scrollTop;  
var sl = ----- Left | ---  
var left = event.clientX;  
var top = event.clientY;  
box1.style.left = left + sl + "px";  
box1.style.right = right + st + "px";  
} // 设置偏移量

1. ⇒ event = event || window.event . 兼容
2. ⇒ 设置偏移量，使其离开 box 的定位，一般是绝对  
在 CSS 样式中添加 position: absolute ;

## - 事件冒泡 { - 所谓冒泡指的是事件的向上传导。当后代元素的事件被触发时，其祖先元素的相同事件也会被触发。

- 可以通过事件对象来取消冒泡 . event.cancelBubble = true .
- 事件的委托： { - 只绑定一次事件，即可应用到新的元素。即使元素之后添加进来 - 可以将事件绑定给父辈共同的祖先元素，这样当后代元素上的事件触发时，会一直冒泡到祖先元素，从而面对祖先元素的响应函数来处理事件
- 利用了冒泡，通过委托可以减少事件绑定的次数，提高程序的性能
- event.target 表示触发事件对象，可用未限制事件执行  
e.g. if (event.target.className == "link") ...

## 一、事件的阶段

- 使用对象，事件二进制形式绑定响应函数。  
它只能同时为一个元素的一个事件绑定一个响应函数。  
不能绑定多个。如果绑定了多个，前面会被覆盖。

- addEventListener() (IE8及以下不支持) → removeEventListener()

- 通过这个方法可以为元素绑定函数。

- 参数1：事件的字符串，~~需要~~ es "click"

参数2：回调函数，响应函数。

参数3：是否捕获阶段触发事件，~~你懂~~ 被为 false。

- 可同时为一个元素绑定多个响应函数，~~顺序执行~~。

- attachEvent (IE及以下) → detachEvent

- 参数1：事件的字符串，需 on

- 2：回调函数。

- 回调时 - - - - - ~~顺序执行~~。

- addEventListener() 中 this 表示事件对象。

attachEvent() 中 this 是 window。

## 二、事件的传播

- 接收阶段：从~~最外层~~的祖先元素向目标元素进行事件的捕获，其他人不能  
发生事件。

- 回播阶段：事件捕获到目标元素，捕获结束开始回溯至最上层被释放。

- 回放阶段：事件从目标元素向他的兄弟元素传递，依次触发相关元素的  
事件。

- 若希望在捕获阶段就触发事件，可以将 addEventListener 中第三参数为 true。

## 2.练习(拖拽) 一流程：

1. 按下，开始拖
2. 移动跟随
3. 放开，固定坐标位置。

box1.onmousedown = function() {

    //div标签

    var o1 = event.clientX - box1.offsetLeft,  
    var ot = - - - - -;

    document.onmousemove = function(event) {

        box1.style.left = o1 + "px";

(注释) //这里box1获取到的是相对body的left值，所以要减去box1.offsetLeft；  
        Event = event || window.event;

    var left = event.clientX; var top = event.clientY;

    box1.style.left = left + "px" + "px";

box1.style.top = top - ot + "px" + "px";

document.onmouseup = function() {

    document.onmousemove = null;

    box1.releaseCapture();

};

//当我们拖拽一个网页中的内容时，浏览器会默认  
搜索此内容，不希望发生此行为，可

return false; //这招 IE6及以下不支持

};

1. 滚轮事件:
- onmousewheel (火狐不支持)
    - 火狐中部DOMMouseScroll 事件，此事件需要 addEventListener 来绑定。
    - Event.wheelDelta 向上滚 120，向下滚 -120，值越小，只看正负（火狐不支持）
    - IE 中用 event.detail，向上滚 -3，向下滚 3。
    - 当滚动运动时，若浏览器设置为防抖，也会影响滚动。可通过 return false 防止。
 

在火狐中不使用 return false，因为是用 addEventListener 绑定的。

禁用 event.preventDefault(); IE8 及以下支持。

可写 event.preventDefault();

2. 键盘事件:
- onkeydown 按下， onkeyup 按开。
  - 大部分都直接给一些对象如焦点的对像或者 document。
  - onkeydown - 直接按，事件会一直触发。
 

连续的按键时，第一次和第二次之间的间隔长一些，防止误操作。
  - keycode 获取按键的 unicode 编码。
    - altkey, ctrlkey, shiftkey 判断三个键是否被按下
  - 在文本框里输入内容，onkeydown 会议认为若不响应，可 return false。
  - 37 左 39 右 40 下

3. 定时器
- 定时器调用 setInterval() 定时器的回调函数是在主线程执行的。JS 是单线程的。
    - 可以将一个函数，每隔一段时间执行一次
    - 参数 1：回调函数；参数 2：间隔时间 (ms)
    - e.g. var timer = setInterval(function() {
 

```
        count.innerHTML = num++;
        }, 1000);
```
    - 返回值 (Number)：定时器的唯一标识。
    - 关闭定时器 clearInterval(timer);
      - e.g. if (num == 10) {  
 1. 加什么立马关掉，都没执行  
 在 [0, 5] 循环区间 index = index % 5;
      - 可接收对象，若参数有效，停止，若无效，什么也不做。
      - 在按钮里绑定一个定时器，在另一个按钮关闭定时器
 

若按钮一直按，定时器速度加快，所以在定时器之间要先关闭。
    - 延时调用 setTimeout
      - 不马上执行，隔一段时间在执行
      - 一旦执行一次，直到定时时间执行多次，返回定时对象 timer。

4. 元素的操作
- 通过 style 属性修改元素样式，每修改一个样式，浏览器将重新渲染一次
  - 通过 object.className = "b2"; 变成深浅灰，同时修改多个样式
  - 基础上加样式 object.className += " b2";

# 1. BOM { - 浏览器对象模型 }

- 通过丁操作浏览器
- BOM 提供一组对象:

{ - window - 代表整个浏览器窗口，同时 window 也是网页中的全局对象 }

- navigator - 代表当前浏览器信息，可用于区别不同浏览器

- 由于历史原因，navigator 对象中大部分属性已不能帮助我们识别了。
- 一般我们只用 userAgent 来判断浏览器信息，返回字符串。  
其中 Zen 中不含 IE 11.3，不能判断。
- 也不能用 userAgent 判断，从用户的浏览器特有的对象判断。  
e.g. ActiveXObject => "~~" in window.

- location - 代表当前 --> 地址栏 ---, - 获取信息或跳转页面。

- 直接打印 location 完整路径。

- location = "http://www..."; 跳转页面，并生成相应的历史记录
- 有些属性把路径分成相对部分
- assign("...") == location = "..."; (window).location.assign(url);
- reload() 和刷新一样，参数为 true，则强制清空缓存。
- replace("...") 跳转页面，不会生成历史记录，滚动页后。

- history - 代表浏览器历史记录。

由于隐私原因，该对象不能获取歷史记录，只能向前后翻阅  
而且只在当次访问时有效

- length: 可以获取到当前访问的数量。
- back() / forward(): 后退、前进一个页面。
- go(): 参数: 1 向前跳过 k, -1 向后 --

2 --- =, -2 ---- 可用

- screen - 代表用户屏幕的信息，获取显示器相关信息。 - availableWidth  
- availableHeight

- 这些对象在浏览器中作为 window 对象属性保存

可以通过 window 对象访问，也可以直接使用

- cookie { Def: 一些数据，存储在文本文件中 }

作用：解决如何记录客户端的用户信息。当用户访问网站时，名字记录在 cookie 中。  
· 下次访问，可读取。

- 储存格式：以名/值对形式 eg. `username = John`
- 创建：`document.cookie = "username = John, path = '/'`；  
读取：`var x = document.cookie` (字符串所回所有cookie)。  
修改类似创建：`document.cookie = "username = Li, path = '/'`；  
删除—设置`expires`参数为以前的时间。

`document.cookie = "username = ; expires=Thu, 01 Jan 1970 00:00:00 GMT"`

1. 线程是进程内的一个执行单元。

一是程序执行的一个完整流程。

二是CPU的最小的调度单元。

相关知识：- 应用程序必须在某个线程的某个线程上。

- 一个进程至少有一个运行的线程，主线程由进程启动后自动创建
- 一个进程若有多个线程  $\rightarrow$  多线程
- 一个进程中数据可以供其中多个线程直接共享
- 多个进程中数据不能直接共享
- 线程池 (thread pool)：供所有线程共享的数据，实现线程对象的反复利用。

相关问题：- 比较单与多线程

多线程 - 优：提升CPU利用率

缺点：创建多线程开销。

线程间切换开销

互锁与状态同步问题。

单线程 - 优：顺序编程简单易懂

缺点：效率低。

- JS是单线程，但使用HTML中的Web workers可使多线程运行。

- 浏览器运行很多线程。单进程firefox 多进程chrome

老版本 部分双线程。

## 2. JS引擎

- 证明JS执行是单线程  $\rightarrow$  `setInterval()` 回调函数在主线程中执行  
非阻塞的。  
 $\rightarrow$  定时器回调函数只有在运行脚本中的代码全部执行完后才能执行。

- 为啥单线程  $\rightarrow$  用户与用户互动以及操作DOM 只能串行，否则会有同步。

- 代码分支：初始化代码、回调代码。

- JS引擎执行代码基本流程：先执行初始化代码。

设置定时器

绑定监听

发送Ajax请求

· 后面某时刻才会执行回调函数（异步执行）

两个线程同时操作DOM

一个并行节点

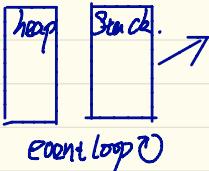
一个操作节点

## 1. web worker — JS → 多线程

— 限制 → 子线程受主线程控制，并不独立

## 2. 事件循环：

— 圈节： JS



Web APIs

DOM (document)  
Ajax (XMLHttpRequest)  
setTimeout.



callback  
queue

onClick. onLoad onDone

- 事件队列① 异步代码执行：遇到异步事件，不等返回结果，挂起它，继续执行其他任务。
- ② 当异步事件返回结果，将其放到事件队列中，被放入事件队列不会立刻执行起回调，而是等待当前执行栈中所有任务都执行完毕
- ③ 主线程空闲状态，主线程会去查找事件队列是否有任务。如果有，取出排在首位的事件，并把这个事件对应的回调函数放到执行栈中，执行其中的同步代码。

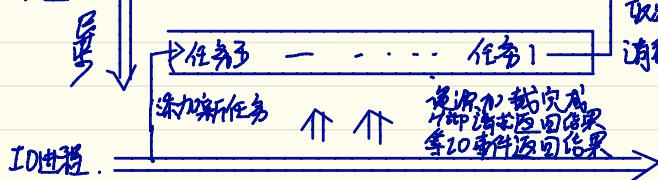
— function ac ()

```
bc();  
console.log('a');  
{  
    function bc () {  
        console.log('b');  
        setTimeout(function () {  
            console.log('c');  
        }, 2000);  
    }  
    ac();  
}
```

Stack.  
console ();  
→ anonymous()  
setTimeout. →  
← console b  
bc();  
ac();  
Callback Queue.  
anonymous()

— 简洁。 主线程

执行其他代码 + 执行回调



取消任务  
取消队列。