

# JavaScript Notes

-- Max Zhang

**1. 了解 JS:**

**2. JS的书写**

**3. JS的注释**

**4. JS的输出语法**

**5. 变量**

**6. 数据类型**

**7. 运算符**

**8. 条件分支语句**

**9. 循环结构语句**

**10. 函数**

**11. 预解析**

**12. 作用域**

**13. 对象数据类型**

**14. 数据类型存储的区别**

**15. 数据类型赋值的区别**

**16. 函数也是对象**

**17. 数组数据类型**

**18. 数组的常用方法**

**19. 字符串**

**20. json格式**

**21. 本地缓存**

**22. 数学方法**

**23. 数字进制转换**

**24. 保留小数**

**25. 时间对象**

**26. 定时器**

**27. BOM**

**27.1 浏览器窗口尺寸**

**27.2 浏览器的弹出层**

**27.3 浏览器的地址栏**

**27.4 浏览器的历史记录**

**27.5 浏览器的版本信息**

**27.6 浏览器的常见事件**

**27.7 浏览器卷去的高度和宽度**

**27.8 控制浏览器滚动**

## **28. DOM**

**28.1 DOM 树的了解:**

**28.2 获取DOM元素**

**28.3 操作元素属性**

**28.4 操作元素文本内容**

**28.5 操作元素样式**

**28.6 DOM 节点**

**28.7 节点属性**

**28.8 获取元素尺寸**

**28.9 获取元素偏移量**

**28.10 获取浏览器窗口尺寸**

**28.11 元素的常用事件 (JS自带的原生事件全小写)**

## **29. 模板引擎**

## **30. 事件**

**30.1 事件三要素:**

**30.2 事件绑定分类:**

**30.3 事件解绑:**

**30.4 事件对象:**

**30.5 鼠标事件的事件对象信息:**

**30.6 键盘事件的事件对象信息:**

**30.7 事件的传播**

**30.8 事件的目标、冒泡和捕获**

**30.9 移入、移出事件的区别**

**30.10 阻止事件传播**

**30.11 事件委托**

**30.12 浏览器默认行为**

## **31. 自执行函数**

## **32. this**

## **33. ES6 定义变量**

## **34. 箭头函数**

## **35. 点点点运算符**

## **36. 解构赋值**

## **37. 对象的简写方式**

## **38. 面向对象开发**

**38.1 创建对象的四种方式**

**38.2 构造函数的书写和使用**

**38.3 prototype**

**38.4 \_\_proto\_\_**

## **38.5 对象访问机制**

### **38.6 原型链**

## **39. 判断数据类型**

## **40. 了解对象**

## **41. ES6 的类**

## **42. 模块化开发**

## **43. http 传输协议**

### **43.1 http 传输协议：**

### **43.2 传输协议：**

### **43.3 一个请求的四个步骤**

### **43.4 响应状态码**

### **43.5 请求方式**

### **43.6 Cookie**

### **43.7 session**

## **44. ajax**

## **45. 跨域请求**

### **45.1 jsonp 跨域**

### **45.2 代理跨域**

### **45.3 cors 代理 - 跨域资源共享**

## **46. 回调函数 callback**

## **47. Promise**

## **48. async / await**

## **49. generator**

## **50. for of 循环**

## **51. set 数据结构**

## **52. Map 数据结构**

## **53. 正则表达式 (Regular Expression)**

## **54. 闭包**

## **55. 继承**

### **55.1 原型链继承**

### **55.2 借用构造函数继承 (借用继承/call继承)**

### **55.3 组合继承**

### **55.4 拷贝继承 (for in 继承)**

### **55.5 寄生继承**

### **55.6 寄生式组合继承 (完美继承)**

### **55.7 ES6 类的继承**

## **56. 事件轮询 - Event Loop**

## 1. 了解 JS:

- 在一个页面中
  - html 表示结构
  - css 表示样式
  - JS 表示行为

- JS三大核心

- ECMAScript

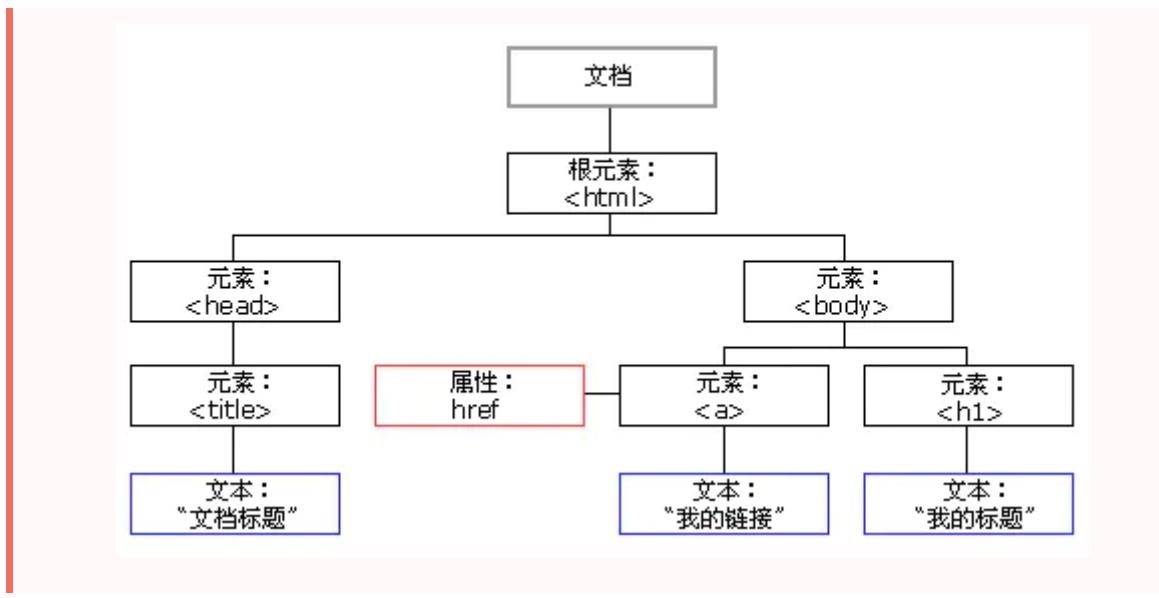
JS的标准，语法

- BOM(Browser Object Model)
  - DOM(Document Object Model)

一整套操作浏览器的属性和方法

文档对象模型 (DOM) 是HTML和XML文档的编程接口。它提供了对文档的结构化的表述，并定义了一种方式可以使从程序中对该结构进行访问，从而改变文档的结构，样式和内容。

DOM 将文档解析为一个由节点和对象（包含属性和方法的对象）组成的结构集合。简言之，它会将web页面和脚本或程序语言连接起来。



## 2. JS的书写

分成三种方式：

### 1. 行内式(强烈不推荐)

- a标签
  - 因为a标签本身就有行为出现
  - 当点击的时候需要区分是跳转连接还是执行JS代码
  - 在href属性里写 javascript: JS代码
  - [点我试试](#)
  - 再点我试试
  - [a标签中的JS](#)
- 非a标签
  - 因为没有自己的行为，所以要给他加个行为(事件)
  - 这个div加了onclick属性

### 2. 内嵌式(不推荐)

- 在页面内书写一个script标签
- 把JS代码书写在标签的内部
- 不需要任何行为，只要打开页面就会执行
- 一个页面内可以书写无限个script标签，会按照从上到下的顺序依次执行
- 理论上script标签可以放在页面的任何位置，推荐放到head或body的末尾

### 3. 外链式(推荐)

- 把JS代码写在一个.js后缀的文件里面
- 在页面上通过script标签的src属性引入页面
- 当一个script标签被当做外链式使用的时候，写在标签对里面的内容没有意义

### 3. JS的注释

```
1 | // 单行注释  
2 | /* 多行注释 */
```

### 4. JS的输出语法

```
1 | 1. alert() -> 浏览器弹出窗  
2 | 2. console.log() -> 浏览器控制台  
3 | 3. document.write() -> 输出到页面内（可以解析标签）
```

### 5. 变量

```
1 | var 变量名 = 变量值  
2 | // 不使用var关键字也可以定义变量，但是强烈不推荐
```

命名规则：

1. 一个变量只能由数字，字母，美元符，下划线组成
2. 一个变量不能由数字开头
3. JS中严格区分大小写
4. 不能使用关键字保留字

命名规范：

1. 不要用中文
2. 变量语义化
3. 驼峰命名法

### 6. 数据类型

#### 1. 基本数据类型（简单数据类型）

- Number 数值
  - 一切十进制表示的数字
  - 一切浮点数
  - 其他进制的数字（16进制，0x开头；8进制，0开头；二进制，0b开头）

- 科学技术法:  $2e5 \Rightarrow 2 \times 10^5$
- NaN: Not a Number
- (在控制台输出其他进制的数字的时候, 会自动转换成十进制)
- String 字符串
  - 在JS里面一切使用引号(双引号, 单引号, 反引号)包裹的内容都是字符串
  - 表示一段文本内容, 是一个字符一个字符连接起来的内容
  - 在字符串里面只写数字的时候, 也不是数值类型
  - 空格直接占位

```

1 | 'Hello'
2 | "Hello"
3 | `Hello` (反引号是ES6新引进)

```

- Boolean 布尔
  - true 表示真, 在计算机储存的时候为1
  - false 表示假, 在计算机储存的时候为0
- Undefined 未定义
  - 本该有一个值, 但是没有
- Null 空
  - 这里有一个空值
- Symbol - ES6新增数据类型
  - symbol是一种基本数据类型。
  - Symbol()函数会返回symbol类型的值, 此值是唯一的。
  - 一个symbol值能作为对象属性的标识符(这是该数据类型仅有的目的)。
  - 不支持语法 new Symbol();通过 `Symbol([description])` 创建symbol值。围绕原始数据类型创建一个显式包装器对象从 **ECMAScript 6** 开始不再被支持。
   
(现有的包装器对象如new Boolean,new String,new Number因为历史遗留原因仍可被创建)
  - Symbols 与 for...in
   
Symbols在for...in迭代中不可枚举。Object.getOwnPropertyNames()不会返回symbol对象的属性, 但是可以通过Object.getOwnPropertySymbols()得到他们。
  - Symbols 与JSON.stringify()
   
当使用 JSON.stringify() 时以 symbol 值作为键的属性会被完全忽略:

```

1 | 检测数据类型:
2 | 关键字 typeof
3 |   1. typeof 变量
4 |     返回值: 以字符串的形式给你的变量数据类型

```

```
5  var n1 = 100
6  var res1 = typeof n1
7
8  2. typeof (变量)
9  var res2 = typeof(n1)
10
11 // 第一种只能检测紧跟着的一个变量
12 // 第二种先运算小括号里面的结果，然后使用typeof去检测结果的数据类型
13 // 当两个及以上typeof连用的时候，一定得到string
14 // 只能准确地检测基本数据类型
15     -> 数值: number
16     -> 字符串: string
17     -> 布尔值: boolean
18     -> undefined: undefined
19     -> null: object
```

## 数据类型转换转数值

- 转数值

### 1. Number()

- 语法: Number(要转换的数据)
- 返回值: 转换好的数据
- 特点:
  - 会把要转化的内容当做一个整体来看待
  - 能转换成数字结果，就是数字结果
  - 不能转换成数字结果，就是NaN
  - true: 1, false: 0

### 2. parseInt()

- 语法: parseInt(要转换的数据)
- 返回值: 转换好的数据
- 特点:
  - 把要转换的任何内容一位一位的看
  - 如果第一位不能转换成数字，直接NaN
  - 如果第一位可以，第一位保留，看第二位
  - 以此类推，直到一个不能转换成合法数字的位置为止不认识
  - 不认识小数点

```
1 | var n1 = "123abc"
2 | var res1 = Number(n1) // NaN
3 | var res2 = parseInt(n1) // 123
4 |
5 | var n2 = true
6 | var res3 = Number(n2) // 1
7 | var res4 = parseInt(n2) // NaN
8 |
9 | var n3 = "123.456"
10| var res5 = Number(n3) // 123.456
11| var res6 = parseInt(n3) // 123
```

### 3. parseFloat()

- 语法: parseFloat(要转换的数据)
- 返回值: 转换好的数据
- 特点:
  - 和parseInt的解析规则一模一样, 只多认识一个小数点

### 4. 取正负值

- 语法: +变量 或者 -变量
- 返回值: 转换好的数据
- 特点: 和Number的解析规则相同

```
1 | var s = "100"
2 | var res1 = +s // 100
3 | var res2 = -s // -100
```

### 5. 非加法的数学运算

- 语法:
  - a \* 1
  - a - 0
  - a / 1
- 特点: 和Number的解析规则相同
- 转字符串
  - 1. String()

- 语法: String(要转换的数据)
- 返回值: 转换好的数据
- 特点: 任何数据类型都能转换

## 2. `toString()`

- 语法: 要转换的数据`.toString()`
- 返回值: 转换好的数据
- 特点: `undefined`和`null`不能转换

## 3. 加法运算

- `+`号有两个意义:
  - 字符串拼接: 符号任意一边是字符串的时候
  - 数学运算: 两边都是数字或者布尔的时候
- 转布尔

## 1. `Boolean()`

- 语法: `Boolean(要转换的数据)`
- 返回值: 转换好的数据
- 特点: 只有五个内容会转成`false`:
  - 0
  - 空字符串
  - `NaN`
  - `undefined`
  - `null`

## 2. 双取反 (`!!`)

## 2. 复杂数据类型 (地址数据类型/引用数据类型)

- `Object`
- `Function`

## 7. 运算符

## 数学运算符:

1. +

- 字符串拼接
- 数学运算

2. -

3. \*

4. /

5. %: 取余

6. \*\*: 取幂

## 赋值运算符:

=, +=, \*=, /=, %=, -=

## 比较运算符:

>, <, >=, <=, ==, ===, !=, !==

和其他语言有些不同，在JavaScript中除了用`==`操作符来判断是否相等外，还有一个`===`操作符，它们的区别是：`==`操作符会先将两边的值进行强制类型转换再比较是否相等，而`===`操作符不会进行类型转换。`==`操作符只要求比较两个值是否相等，而`===`操作符不仅要求值相等，而且要求类型相同。`!=`和`!==`的区别也是类似的，`!=`号会做强制类型转换，而`!==`不会。

```
1 // 注意，这里有一个特殊值NaN，即 Not a Number，表示非数字，它和任何数做相等比较，包括它自己，都会返回false。所以判断NaN最好用isNaN()函数。
2
3 // false
4 NaN == NaN
5 // false
6 NaN === NaN
7
8
9 // 还有两个特殊值undefined和null，使用时需要注意
10
11 // true
12 null == undefined
13 // false
```

```
14 | null === undefined
15 |
16 |
17 | // != 和 !== 的情况与==, ===的情况类似
```

逻辑运算符:

1. && 逻辑与
2. || 逻辑或
3. ! 逻辑非

自增自减运算符:

一元运算符一种

自增: i++, ++i

自减: i--, --i

当出现多个自增自减时, 从左到右计算

## 8. 条件分支语句

### 1. if 语句

- if (条件) { 要执行的代码 } else if (条件) { 要执行的代码 } else { 要执行的代码 }

### 2. switch

case的值 和 变量的值 必须是 ===

只能是 精确的值, 不能是范围

执行 break 才会跳出, 执行完一个case如果没有break跳出, 则会直接执行下一条, 无论条件是否满足。

```
1 | switch (要判断的变量) {  
2 |     case 情况1:  
3 |         代码  
4 |         break  
5 |     case 情况2:  
6 |         代码  
7 |         break  
8 |     default: // 可以不写  
9 |         代码  
10| }
```

## 9. 循环结构语句

1. 初始值：作为循环的开始
2. 条件判断：决定要不要继续循环
3. 要重复执行的代码
4. 改变初始值：为了让循环有结束

### while:

```
1 | while (条件) {  
2 |     代码  
3 |     改变初始值  
4 | }
```

### dowhile:

```
1 | do {  
2 |     代码  
3 | } while (条件)
```

至少会执行一次

### for:

```
1 | for (var i = 0; i < 10; i++) {  
2 |     代码  
3 | }
```

**break** : 跳出

**continue** : 结束本次，继续下次

**JS标记语法** :

```
1 Outer: // 名字随便起
2 for (var i = 1; i <= 5; i++) {
3     for (var j = 1; j <= 3; j++) {
4         if ( i === 3 && j === 2) {
5             console.log("k看到半条虫子")
6             break Outer
7         }
8         console.log("吃的第 " + i + " 个包子的第 " + j + " 口")
9     }
10 }
```

```
1 // 判断是不是质数
2 // 可以从 2 开始
3 // 当大于数字的一半时，不能整除
4 // 当一个数字能开出平方根，到这个平方根就可以结束了
```

## 10. 函数

### 1. 声明式函数

语法: `function 函数名 () {}`

### 2. 赋值式函数

语法: `var 函数名 = function () {}`

`函数名` 和 `函数名()` 是不一样的:

- `函数名` 是一个变量，表示这个函数
- `函数名()` 是执行这个函数

函数调用上的区别:

- 两种声明方式，调用方式是一样的
- 区别: 调用的时机不一样
  - 声明式函数: 可以在声明之前调用，也可以在声明之后调用
  - 赋值式函数: 只能在声明之后调用

函数参数的个数关系

- 一样多

按照从左到右的顺序一一对应

- 实参多

前面的按照顺序一一对应，多出来的实参，在函数内部没有形参接收

不能直接使用

- 形参多

前面的按照顺序一一对应，多出来的形参因为没有实参赋值，所以使用的时候就是 `undefined`

**arguemets** (实参数组) :

- 在函数内部天生自带的变量
- 表示所有实参的集合
- argument 的属性
  - `length`: 实参数数，可读写
  - 排列：索引排列，从零开始
  - `argument[index]` 可读取相应位置数据，可读写

```
1 | function func() {  
2 |     var a = arguemets[0]  
3 |     console.log(a)  
4 | }
```

函数和元素结合：

- 函数还可以当做一个页面元素的事件处理函数
- 当页面上某个元素处罚行为的时候，执行某个函数
- 语法：元素.行为 = 函数

页面元素的简单操作：

- 在一个页面里面，元素的 `id` 名，可以直接当做一个变量来使用
- 这个变量就代表着这个元素

两种方式：

- 直接书写匿名函数

元素.行为 = 函数

- 给事件赋值具名函数

元素.行为 = 函数名（注意：没有括号）

```
1 | function fn() {  
2 |   console.log(123)  
3 | }  
4 |  
5 | box.onclick = fn // 没有括号  
6 |  
7 | // fn 这个变量名代表了一个函数  
8 | // fn() 表示执行这个函数
```

JS的安全数字范围： $(-2^{53} + 1) \sim (2^{53} - 1)$

超过这个范围，可以使用BigInteger（2020年新，仍有问题，尚未普及。例如  
`Math.floor(10n)`会报错： Cannot convert a BigInt value to a number at Math.floor）

直接在数字后面加 `n` 或者用`BigInt()`转换（只保留整数部分）

定义函数的时候可以给参数定义默认值：

```
1 | function fn (a = 10, b) {  
2 |   console.log(a)  
3 |   console.log(b)  
4 | }
```

## 11. 预解析

- 不是关于怎么写代码
- 关于了解代码的执行机制，和不要怎么写代码
- 预：预先，在所有代码执行之前
- 解析：解释，对代码进行通读并解释（只是把整体代码当做一个文档）

解释的部分：

1. `var` 关键字：会把 `var` 关键字定义的变量在代码执行之前声明

```
1 | /*  
2 |  当代码在浏览器执行的时候，  
3 |  在所有代码开始执行之前，先把声明做好 (var num)
```

```
4  */
5  var num = 100 // 当代码执行到这一行才会赋值
6
7 /*
8     下面这段代码的执行顺序其实是：
9     var a
10    console.log(a)
11    a = 100
12    console.log(a)
13 */
14 console.log(a) // 这里会打印undefined
15 var a = 100
16 console.log(a)
```

2. 声明式函数：会把这个函数名在所有代码执行前声明，并且赋值为一个函数

```
1 // 在所有代码执行前，告诉浏览器，fn这个名字可以使用，并且fn代表的是一个函数
2 // 所以在函数声明之前也可以调用函数
3 function fn()
4     console.log("一个函数")
5 }
```

注意：赋值式函数 => var fn = function() {...}

按照var的规则进行解析

如果有 **变量名** 和 **函数名** 重名，在预解析阶段以函数为准

```
1 /*
2     预解析：
3     1. var fn // 声明fn变量
4     2. function fn() {...} // 声明fn变量，并且赋值为一个函数
5     3. 预解析结束的时候，记录的fn变量是一个函数
6
7     代码开始执行
8     1. fn()
9     2. fn = 100 // 从这里开始，fn就不再是一个函数了，只是一个数值100，所以后续的函数调
10    用会报错
11          // TypeError: fn is not a function
12          3. fn()
13          4. fn()
14 */
15 fn()
16 var fn = 100
17 fn()
18 function fn() {
19     console.log("This is a function")
20 }
```

**if** 条件无论成立与否，里面的代码会进行预解析

## 12. 作用域

- 教你怎么写代码
- 变量（变量名，函数名）生效的范围

两种：

### 1. 全局作用域

打开一个页面就是一个全局作用域

全局作用域，叫window

### 2. 私有作用域（局部作用域）

只有函数生成私有作用域

每一个函数就是一个私有作用域

作用域的上下级关系：

- 函数写在哪个作用域下，就是谁的子级作用域

作用域的上下级关系：

- 为了确定变量的使用范围
- **三个机制**

#### 1. 变量定义机制

- 有 var 关键字
- 声明式函数

一个变量定义在哪一个作用于里面，就只能在该作用域或其下级作用域里面使用，上级作用域不能使用

#### 2. 变量使用机制

拿某一个变量的值来使用

当需要使用一个变量（函数）会首先在自己作用域内查找，如果有，直接使用；如果没有，去上级作用域查找；如果直到全局作用域都没有查到，报错

### 3. 变量赋值机制

一定要有赋值符号

当给一个变量（函数名）赋值的时候，首先在自己的作用域里面查找，如果有，直接赋值；如果没有，去上级作用域查找；如果直到全局作用域都没有查到，把这个变量定义为全局变量，再进行赋值

赋值符号是从右向左赋值

```
1 function fn() {
2     var num = 100
3
4     function fun() {
5         // 变量的使用和赋值
6         // 预解析的时候，fun函数里面会预解析一个 num 变量（属于fun的私有作用域）
7         // 使用num的值，赋值给num
8         // 就是把undefined 赋值给 num
9         var num = num
10        console.log(num) // 这里会打印undefined
11    }
12
13    fun()
14 }
```

#### 作用域里面的预解析

预解析分成几个阶段：

- 全局预解析
  - 页面打开的时候就进行了
  - 只解释属于全局的内容
- 私有作用域解析
  - 当函数执行的时候，进行预解析
  - 函数内部的预解析，只属于函数内部

问题：函数执行的时候，会进行 **形参赋值** 和 **预解析**，一旦函数的形参和定义的私有变量重名，先预解析还是先形参赋值？

```
1 function fn(b) {  
2     function b() {  
3         console.log("b")  
4     }  
5     b() // 报错? 不报错?  
6 }  
7  
8 fn(200)  
9 // 如果预解析在前, 形参赋值在后  
10 // 这个fn函数执行的时候, 先把 b 解析成为一个函数  
11 // 然后再给b赋值为200, 那么就会报错了  
12  
13 // 如果预解析在后, 形参赋值在前  
14 // 这个fn函数执行的时候, 先把b赋值为两百  
15 // 然后预解析的时候, 把b赋值为函数, 就不会报错
```

这里不会报错，在函数执行的时候，先进行形参赋值，再进行预解析

## 13. 对象数据类型

- JS数据类型的一种
- 一个复杂数据类型
- 函数：一个盒子，承载一段代码
- 对象：一个盒子，承载一堆数据

对象的创建方式：

1. 字面量创建：`var o = {}`
2. 内置构造函数创建：`var o = new Object()`

区别：

1. 字面量创建可以在创建的时候就直接向对象里面添加一些数据

`key: value`键值对，用逗号(,)隔开，`key`即对象的属性

2. 内置构造函数不好直接添加成员，后期通过对对象的操作语法来进行增删改查

```
1 | var o = {  
2 |     num : 100, // 对象的属性  
3 |     name : "Jack",  
4 |     fn : function () { console.log("o的fn函数") } // 对象的方法  
5 |     // 每个成员的名称都是字符串  
6 | }
```

对象的操作语法：

语法有两种：

1. 点语法
2. 数组关联语法

• 增：

- 对象名.成员名 = 值
- 对象名["成员名"] = 值

• 删

- delete 对象名.成员名
- delete 对象名["成员名"]

• 改

- 对象名.成员名 = 值
- 对象名["成员名"] = 值

• 查

- 对象名.成员名
- 对象名["成员名"]

注意：因为对象数据类型是一个复杂数据类型，在控制台打印的时候，会出现两种情况：

- 不展开对象数据类型的时候，是当前的样子
- 展开对象数据类型后，显示最终的样子

解决方法：

- 单独打印属性
- 使用console.table()

两种操作语法的区别：

- 点语法：
  - 不能使用变量
  - 不能拼接字符串
- 数组关联语法
  - 可以使用变量
  - 可以拼接字符串

循环遍历对象：

for in 循环

for (var 变量 in 对象) {...}：这里 变量 指的是属性名

判断一个成员是不是在这个对象里：

使用 in 语法：成员名字符串 in 对象名

```
1 | var o = {  
2 |   name : "Max",  
3 |   age : 25  
4 | }  
5 | console.log(name in o) // 这里是false  
6 | console.log("name" in o) // true
```

## 14. 数据类型存储的区别

- 数据类型：
  - 基本数据类型
    - Number - 数值
    - String - 字符串
    - Boolean - 布尔
    - Undefined - 空
    - Null - 空

- 复杂数据类型
  - Function - 函数
  - Object - 对象
- 存储上是有区别的
  - JS 打开的内存空间
    - JS 是一个脚本语言，依赖于浏览器执行
    - 本质是依赖浏览器里面的 JS 解析引擎
    - JS 本身不打开内存空间，因浏览器在电脑上运行的时候，会占用一段内存空间，JS 就是在这一段内存空间里面运行的
  - 数据类型的存储，就是存储在浏览器分配给 JS 存储的一段空间
  - JS 的存储空间（浏览器的一段存储空间）
    - 栈内存：后进先出
    - 堆内存：随机存储
  - 数据类型的存储
    - 基本数据类型：直接存储在栈内存里面
    - 复杂数据类型：把 数据 放在堆内存里面，把 地址 放在栈内存的变量里面
  - 代码的执行

只能直接访问到栈里面的内容。想访问某个对象里面的成员，因为对象本身在堆内存里面，就需要利用栈里面的地址，找到堆里面的空间，然后去访问内部的成员

## 15. 数据类型赋值的区别

1. 基本数据类型：就是把变量存储的值直接赋值给另一个变量，赋值过后两个变量没有关系了
2. 复杂数据类型：把一个变量的地址给了另一个变量，赋值过后，两个变量操作一个空间
3. 函数的形参和实参的关系：实参就是在函数调用的时候给形参赋值，实参和形参的交互，和变量赋值时一个道理。（实参是引用数据类型时在函数中对该对象属性的修改会影响到外部作用域）
4. 函数的返回值也是变量的赋值的一种：返回值是把函数内部的数据 return 出去

## 16. 函数也是对象

函数是保存一段代码，对象是保存一段数据，函数本身也是一个对象，可以保存一堆数据

当定义好一个函数以后，函数就有两个功能

1. 函数名`fn`: 把函数当做一个函数执行
2. 函数名.成员名 = “值”: 储存数据

```
1 function fn() {  
2     console.log("A Function")  
3  
4     console.log(age) // 但是这里会报错，因为这两种功能互不干扰，也没有关系  
5     console.log(fn.age) // 这里就可以正常打印，相当于把fn当成一个独立的对象  
6 }  
7  
8 fn[ "age" ] = 18  
9 fn[ "gender" ] = "男"
```

## 17. 数组数据类型

Array - 复杂数据类型

数组的创建：

1. 字面量创建：`var arr = []`, `var arr = [1, 2, ....]`
2. 内置构造函数创建：`var arr = new Array()`, `var arr = new Array(10)`, `var arr = new Array(1, 2, 3, ....)`, 传递一个数据时表示创建的数组的长度，多个数据时就是数组的数据

数组的操作：

- `length` - 可读写（当设置的长度比本身长度小，相当于删除；否则不够的用空补齐）
- 索引 - 从零开始（超出范围`undefined`）
- 遍历 - `for`循环, `for in` 循环
- 数组也是一个对象，可以当做对象使用，使用 `点语法` 存储一些数据，存储的数据不影响`for`循环，但是影响`for in`循环（`for in` 循环是按照key遍历，所以会把添加的数据也加进去）

## 18. 数组的常用方法

JS自带的一些操作数组的方法

数组常用方法的语法，必须是 - 数组.`xxx()`

数组和伪数组的区别

- `length` 和 `索引` 都一样，循环遍历都一样

- 长得也一样
- 唯独数组常用方法，伪数组用不了

## 伪数组

函数名	效果	返回值
push(数据1, 数据2,.....)	按顺序把元素添加到数组末尾	数组长度
pop()	删除数组的最后一个数据	被删的数据
unshift(数据1, 数据2,.....)	从数组的最前面插入一些数据	插入后的数组长度
shift()	删除数组的最前面一个数据	被删的数据
reverse()	反转数组	反转后的数组
arr.sort()	数组排序（字典排序，按位排序1,100,11），可以传入比较器 (arr.sort( function (a, b) {return a - b}))	排序后的数组
arr.splice(开始索引, [个数]) - 个数不写则截取到最后	1. 截取数组 2. 替换新内容 - arr.splice(开始索引, 多少个, 替换数据1, 替换数据2...)	截取的数组
concat(数据/数组.....)	如果参数是数组，数组中的每一项追加到原数组后；如果参数是数据，直接追加。 <b>不改变原数组</b>	追加好的数组
arr.slice([索引1], [索引2]) - 包前不包后，第二个参数不写则到尾，不写参数从头到尾。参数可以写负数，表示倒数第几个	切片， <b>不改变原始数组</b>	截取的数组

函数名	效果	返回值
join("链接符号")	把数组中的数据用链接符号连接在一起，不传参按逗号连接	连接好的内容，String
indexOf(数据, [开始索引])	查看指定数据的索引（，或从某个索引开始向后查找）	索引，如果沒有则-1
lastIndexOf(数据)	反向查看指定数据的索引	索引，如果沒有则-1
arr.forEach(function (item, index, arr) {})	item: 数组的每一项, index: 数组每一项的索引, arr: 原始数组。取代for循环的作用，遍历数组	无
arr.map(function(item, index, arr) {})	映射数组	新的数组
arr.filter(function(item, index, arr) {})	筛选	新数组
arr.every(function(item, index, arr) {})	判断是否所有数据满足条件	Boolean
arr.some(function(item, index, arr) {})	判断是否存在数据满足条件	Boolean
copyWithin(目标位置, 开始索引, 结束索引)	使用数组里面的内容替换数组里面的内容	新的数组
fill(要填充的数据, 开始索引, 结束索引)	前提：数组要有length。置顶数据填充数组	填充好的数组
includes(数据)	查看数组中是否有这个数据	Boolean

函数名	效果	返回值
<b>flat(数字)</b>	拍平数组。扁平化多少层，默认1。用 <b>Infinity</b> 拍平到一维数组 <b>(2018)</b>	新的数组
<b>arr.flatMap(function(item, index, arr){})</b>	拍平一层数组。一边拍平一边映射	新的数组
<b>arr.find(function (item) {})</b>	根据条件查找数据	找到的数据
<b>arr.findIndex(function (item) {})</b>	根据条件查找数据	找到的数据的索引

(粗体为ES6新函数)

## 19. 字符串

字符串的创建：

1. var str = 'hello world'; var str = "hello world"
2. var str = new String("hello world")

两种方法创建的字符串除了在控制台打印的时候有区别，使用起来没有任何区别

字面量 : hello world

内置构造函数 : ► *String {"hello world"}*

因为字符串是一个包装数据类型

- 一个数据当你使用的时候会自动转换成复杂数据类型
- 当你使用完毕，自动转换回基本数据类型

点语法：

- obj.name 便是访问 obj 空间内部的name成员

- 因为obj是一个复杂数据类型，再堆内存里面有一个空间
- str.length也可以执行
  - 访问str这个空间内部的length成员
  - 但是str是一个基本数据类型，在堆里面没有空间
  - 因为当使用str.length的时候
  - 会自动转换成复杂数据类型，在堆内存里面开辟一个空间
  - 按照索引把每一位字符排列进去
  - 等访问结束，这个开辟的临时空间会销毁

## toString()

- 转字符串的
- 数字、布尔可以转，字符串也可以
- 因为数字、布尔和字符串都是包装数据类型
- undefined 和 null不是包装数据类型

字符串的索引只能获取不能设置（只读） - 不会报错，只是设置不成功 - 基本数据类型不会被改变，只能覆盖（数值或字符串相加的操作也没有改变，只是将结果重新赋值了）

## 模板字符串

ES2015（ES6）以前，拼接字符串使用“+”

ES6的标准中推出了一种新的字符串定义方式，使用反引号(`` - 波浪键)

反引号定义的字符串叫做 [模板字符串](#)

和普通字符串的区别：

- 普通字符串不能换行，模板字符串可以换行书写
- 普通字符串不能直接解析变量
- 兼容性问题，IE低版本不支持，但是打包时会自动转

```
1 var str = "hello world"
2 var str1 = `hello
3
4
5 world`
6
7 var age = 20
8
9 var str2 = "我今年 " + age + " 岁了"
10 var str3 = `I'm ${ age } years old`
```

模板字符串可以调用函数 - 字符串里面的内容是函数的参数 - `{}$` 把字符串切开，组合成一个数组当做第一个参数 - 然后从左到右的 `{}$` 中的内容作为后面的参数

字符串的常用方法：

## 1 concat

将两个或多个字符的文本组合起来，返回一个新的字符串。

```
var a = "hello";
var b = ",world";
var c = a.concat(b);
alert(c);
//c = "hello,world"
```

## 2 indexOf

返回字符串中一个子串第一处出现的索引（从左到右搜索）。如果没有匹配项，返回 -1。

```
var index1 = a.indexOf("l");
//index1 = 2
var index2 = a.indexOf("l",3);
//index2 = 3
```

## 3 charAt & charCodeAt()

返回指定位置的字符/字符编码（utf-8）。

```
var get_char = a.charAt(0);
//get_char = "h"
```

## 4 lastIndexOf

返回字符串中一个子串最后一处出现的索引（从右到左搜索），如果没有匹配项，返回 -1。

```
var index1 = lastIndexOf('l');
//index1 = 3

var index2 = lastIndexOf('l',2)
//index2 = 2
```

## 5 match

检查一个字符串匹配一个正则表达式内容，如果没有匹配返回 null。

```
var re = new RegExp(/^\w+$/);
var is_alpha1 = a.match(re);
//is_alpha1 = "hello"

var is_alpha2 = b.match(re);
//is_alpha2 = null
```

## 6 substring

返回字符串的一个子串，传入参数是起始位置和结束位置。

```
var sub_string1 = a.substring(1);
//sub_string1 = "ello"

var sub_string2 = a.substring(1,4);
//sub_string2 = "ell"
```

## 7 substr

返回字符串的一个子串，传入参数是起始位置和长度

```
var sub_string1 = a.substr(1);  
//sub_string1 = "ello"  
  
var sub_string2 = a.substr(1,4);  
//sub_string2 = "ello"
```

## 8 replace

用来查找匹配一个正则表达式的字符串，然后使用新字符串代替匹配的字符串。

```
var result1 = a.replace(re,"Hello");  
//result1 = "Hello"  
  
var result2 = b.replace(re,"Hello");  
//result2 = ",world"
```

## 9 search

执行一个正则表达式匹配查找。如果查找成功，返回字符串中匹配的索引值。否则返回 -1

◦

```
var index1 = a.search(re);  
//index1 = 0  
  
var index2 = b.search(re);  
//index2 = -1
```

## 10 slice

提取字符串的一部分，并返回一个新字符串（与 substring 相同，但是参数可以是负数）。

```
var sub_string1 = a.slice(1);
```

```
//sub_string1 = "ello"  
  
var sub_string2 = a.slice(1,4);  
  
//sub_string2 = "ell"
```

## 11 split

通过将字符串划分成子串，将一个字符串做成一个字符串数组。第二个参数，可以选择保留多少个

```
var arr1 = a.split("");  
  
//arr1 = [h,e,l,l,o]
```

## 12 length

返回字符串的长度，所谓字符串的长度是指其包含的字符的个数。

```
var len = a.length();  
  
//len = 5
```

## 13 toLowerCase

将整个字符串转成小写字母。

```
var lower_string = a.toLowerCase();  
  
//lower_string = "hello"
```

## 14 toUpperCase

将整个字符串转成大写字母。

```
var upper_string = a.toUpperCase();
```

```
//upper_string = "HELLO"
```

## 15 includes

判断字符串里是否包含该字符串片段

```
var bool = str.includes("字符串片段")
```

## 16 trim/trimStart(trimLeft)/trimEnd(trimRight)

去除首尾/首/尾空格

## 17 padStart/padEnd

从前面/后面字符串补齐

```
str.padStart(目标长度, "填充字符串")
```

## 18 startsWith/endsWith

判断该字符串是不是以某个片段开始/结束

```
str.startsWith("字符串")
```

## 20. json格式

**json**：一种固定的字符串格式

在电脑网络传输的过程中吗，只能传输字符串，传递不了对象和数组数据类型。如果想传递数组或者对象，那么需要转换成字符串的格式传递。**json**格式就是满足对象和数组数据结构的一种字符串

使用：

JSON.parse(要转换的json格式字符串)：把 **json** 格式的字符串转换成 JS 的数组或者对象

`JSON.stringify(要转换的数组或对象)`: 把 JS 格式的数组或者对象转换成 `json` 格式的字符串

`json` 数据格式中 (重点)

1. 描述数组或者对象数据类型 ("[]","{}")
2. 对象中的 `key` 和 `value` 都使用 双引号 包裹 - 数字和布尔值可以不用引号
3. 数组里面可以放多个对象
4. 当有多个数据的时候, 最后一个数据后面不能有逗号
5. 一个json格式中, 可以使用的符号只有 {}, [], "", , - 内容中的不算
6. 转换json格式字符串的时候, 函数会被自动过滤 - json无法保存函数

`.json` 文件 - 只能书写json格式的内容

## 21. 本地缓存

把一些数据记录在浏览器中 - 多种本地缓存之一

1. `localStorage`
2. `sessionStorage`

作用: 浏览器给我们提供的一些本地存储数据的机制

区别:

- `localStorage` - 永久缓存, 除非手动删除
- `sessionStorage` - 会话缓存, 关闭浏览器就没有了

共同点:

- 只能存储字符串格式的数据 - 可以使用json
- 取出来的数据就变成字符串了

查看: 浏览器控制台 - application - Local Storage/Session Storage

语法:

### 1. `localStorage`

- `localStorage.setItem("名字", "值")`
- `localStorage.getItem("名字")` - 没有则返回null
- `localStorage.removeItem("名字")`
- `localStorage.clear()`

## 2. sessionStorage

- `sessionStorage.setItem("名字", "值")`
- `sessionStorage.getItem("名字")` - 没有则返回null
- `sessionStorage.removeItem("名字")`
- `sessionStorage.clear()`

## 22. 数学方法

1. `random`
2. `round`
3. `ceil`
4. `floor`
5. `pow(数字, 幂数)` - `power` - 幂
6. `sqrt` - 算术平方根 (只有正数)
7. `abs`
8. `max`
9. `min`
10. `PI` - 属性 - 近似于PI的值

## 23. 数字进制转换

### 2~36进制

#### 1. 十进制转换其他进制

- `num.toString(要转换的进制)`
- 返回值以字符串的形式返回转换好的数字 - 如果不以字符串的形式返回, JS 会自动转换成十进制
- 注意:
  - 返回值不能直接加法 - 返回值是字符串
  - 其他计算不能直接按照转数字的方法转换
  - 如果想进行数学计算, 要按照转换进制的方法转换回来

#### 2. 其他进制转换十进制

- `parseInt(要转换的数字, 你把这个数字当做几进制)`

## 24. 保留小数

`toFixed(位数)` - 返回值字符串 - 如果不够指定小数位, 用0补齐 - 四舍五入

## 25. 时间对象

内置构造函数 `Date()` - 用来创建时间对象

### 1. 创建当前时间对象

```
var time = new Date()
```

返回值: 当前终端的时间

### 2. 创建指定日期事件对象

- 传递数字: 年, 月 (从零开始), 日, 时, 分, 秒, 毫秒 - 至少传递两个参数, 传一个参数的时候, 获取的是格林威治时间 (计算机元年, Thu 1970 - 01 - 01) - 会自动进位

```
var time = new Date(2021, 0, 1, 12, 30, 1, 1)
```

- 传递字符串: "yyyy-mm-dd HH:MM:SS" / "yyyy/mm/dd HH:MM:SS"

```
var time = new Date("2021-01-23 10:28:00")
```

获取/设置时间信息:

1. `getFullYear()`/`setFullYear()`
2. `getMonth()`/`setMonth()`
3. `getDate()`/ `setDate(从零开始)`
4. `getHours()`/`setHours()`
5. `getMinutes()`/`setMinutes()`
6. `getSeconds()`/`setSeconds()`
7. `getMilliseconds()`/`setMilliseconds()`
8. `getDay()` - 周几 - 0表示周日
9. `getTime()` - 改时间对象的时间戳 /  `setTime()`

时间戳:

- 格林威治时间: 1970年1月1日0点0分0秒 - 计算机元年
- 时间戳即 - 时间对象到格林威治时间的毫秒数

获取时间差：使用时间戳相减

## 26. 定时器

同步异步：

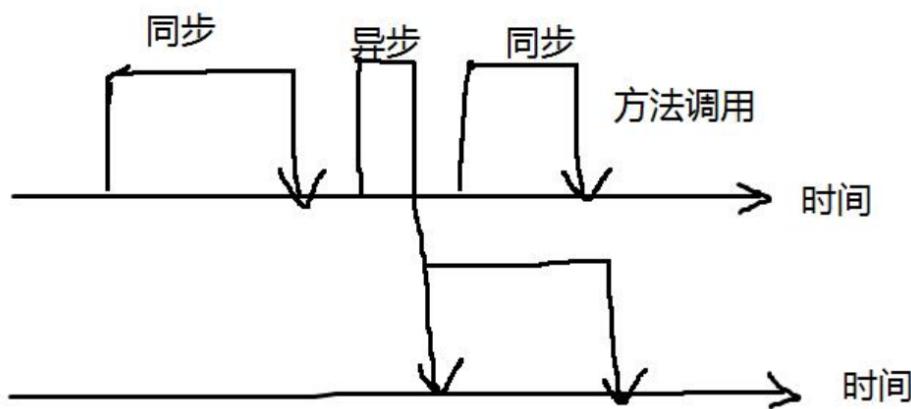
### 概念性

同步和异步通常用来形容一次方法调用。

- 同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。
- 异步方法调用更像一个消息传递，一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而，异步方法通常会在另外一个线程中，“真实”地执行着。整个过程，不会阻碍调用者的工作。

[回到顶部](#)

### 图示例



**JS** 是单线程同步代码机制 - 当程序进入死循环，后面代码全部无法执行

**WEBAPI** 提供了一个队列机制，用来模拟多线程，叫单线程异步 - JS中，当代码从上到下的执行遇到异步代码的时候，会把他放在队列里面，先不执行，等到所有同步代码执行完毕，再从队列里面拿到代码来执行

**JS的定时器：**

- JS 提供了两个异步定时器机制
- `setTimeout()` - 延时定时器：

`setTimeout(fn, time)` - 时间到达的时候，执行一遍函数。

- `setInterval()` - 间隔定时器：

`setInterval(fn, time)` - 每隔固定时间，执行一遍函数

定时器的返回值：

不分定时器种类，只表示是页面中的第几个定时器，返回值是用来关闭定时器的

关闭定时器： - 关闭定时器不分种类

1. clearInterval(要关闭的定时器的返回值)

2. clearTimeout(要关闭的定时器的返回值)

## 27. BOM

**BOM** - Browser Object Model - 浏览器对象模型

- 浏览器给我们提供的一套操作浏览器窗口的属性和方法
- **BOM** 的顶级对象是 **window**

**window** - 一个对象，打开一个页面就有一个 **window**，全局定义的所有变量都在 **window** 下。所有和 BOM 相关的 API 都是 **window.xxx** - 在书写的时候，"window." 可以省略

### 27.1 浏览器窗口尺寸

- 可视窗口的尺寸
- 浏览器有滚动条时
  - 一般浏览器上滚动条算浏览器的一部分
  - 在Safari上不算
- 两个属性： - 包含滚动条的尺寸
  1. **innerWidth**
  2. **innerHeight**

### 27.2 浏览器的弹出层

1. **alert()** - 警告框 - 只有一个确定按钮

    返回值: **undefined**

2. **confirm()** - 选择框 - 有确定和取消两个按钮

    返回值: **Boolean**

3. **prompt()** - 输入框 - 有一个 **input** 输入框，有确定和取消两个按钮

返回值：如果用户点击确定，返回input的内容；如果点击取消，返回null

共同点：

- 会阻断程序进行 - JS 单线程 - 弹出层弹出之后，如果用户没有点击按钮表示当前弹出层没有结束，直到用户操作以后，才会继续向下执行代码

### 27.3 浏览器的地址栏

一个地址包含： - <https://www.bilibili.com/video/BV14y4y1q754?p=123>

- http/https - 传输协议 - 前后端交互的方式
- [www.bilibili.com](http://www.bilibili.com) - 域名 - 找到一台服务器电脑
- ?p=123 - 查询字符串（queryString） - 打开页面的时候携带的信息
- #abc - 哈希值（hash） - 锚点定位

`window` 下有个成员叫 `location` - 存储着和网页地址所有相关的信息

- hash: 当前页面的 hash 值
- href: 当前地址栏地址（可读写） - 中文是url编码格式
- search: 当前地址中的查询字符串（queryString）
- reload(): 重新加载当前页面 - 不要写在打开页面就能执行的地方！！！ - 会把浏览器卡死

### 27.4 浏览器的历史记录

`window` 的 `history` 成员

1. back() - 回退上一条历史记录
2. forward() - 前进到下一条历史记录
3. go(整数) - 0: 当前页面；正数：前进；负数：后退

### 27.5 浏览器的版本信息

用来区分浏览器

`window` 的 `navigator` 成员

1. userAgent - 浏览器的版本及型号信息
2. platform - 表示浏览器所在的操作系统

## 27.6 浏览器的常见事件

由浏览器行为触发的事件

1. `window.onload = function() {}` - 页面所有资源加载完毕后执行（图片，视频，音频，...）

- JS 前置：`<script>` 标签可以放在任何位置，但是当他放到DOM元素前面时，不能操作DOM元素，会报错。这时，可以利用`window.onload`来等页面所有资源加载完毕再操作DOM元素

2. `window.onscroll = function() {}` - 浏览器滚动条滚动的时候触发（无论横向还是纵向）

3. `window.onresize = function() {}` - 浏览器可视窗口改变的时候触发

## 27.7 浏览器卷去的高度和宽度

当页面比窗口宽或者高的时候，会有一部分是随着滚动被隐藏的。

上面隐藏的叫做 卷去的高度

左边隐藏的叫做 卷去的宽度

获取卷去的高度：

1. `document.documentElement.scrollTop` - 必须要有 DOCTYPE 标签

2. `document.body.scrollTop` - 必须没有 DOCTYPE 标签

兼容写法

```
var scrollTop = document.documentElement.scrollTop || document.body.scrollTop
```

当使用短路表达式来写兼容写法的时候，注意，这种写法只能在不报错 - 即方法或属性执行没有问题，只是拿不到值得时候 - 的时候可以使用。

获取卷去的宽度：

1. `document.documentElement.scrollLeft` - 必须要有 DOCTYPE 标签

2. `document.body.scrollLeft` - 必须没有 DOCTYPE 标签

## 27.8 控制浏览器滚动

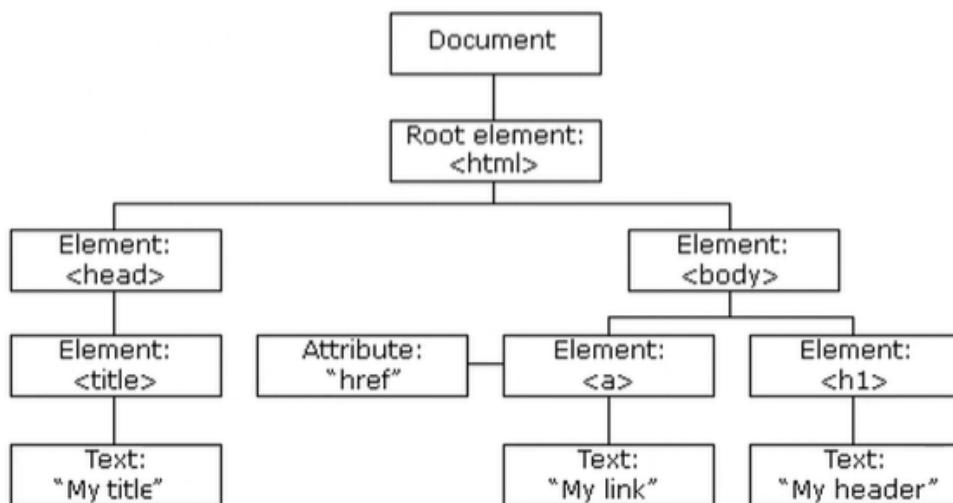
scrollTo(x坐标, y坐标)

- 如果传递数字，必须两个参数 - 瞬间定位，无滚动过程
- 可以传递对象，对象中可以写一个或两个参数，可以设定平滑滚动 - behavior: "smooth" / "instant"

## 28. DOM

Document Object Model 文档对象模型

- 用来描绘一个层次化的节点树
- 允许开发人员获取、添加、移除和修改页面的某一部分元素

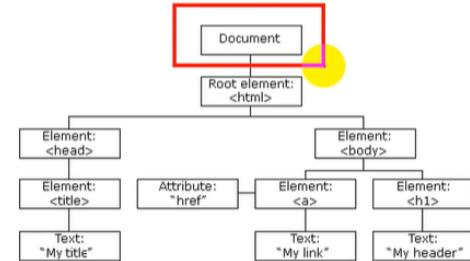
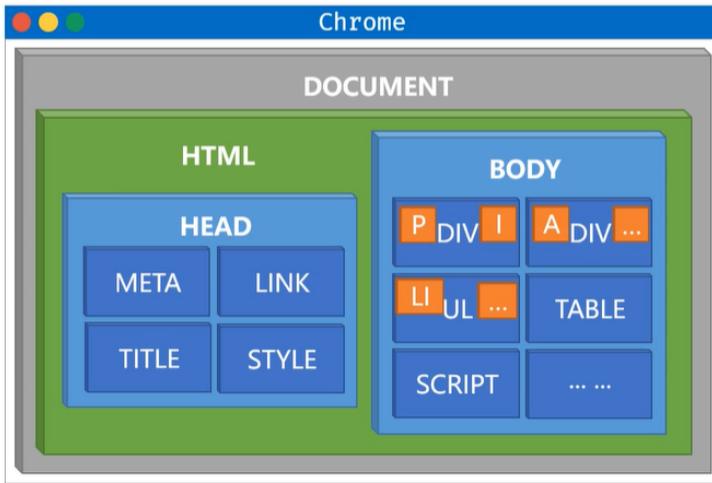


其实就是操作文档的一些能力

我们可以操作：

- HTML元素（增删改查）
- 元素样式
- 元素属性
- 元素添加事件

### 28.1 DOM 树的了解：



## 28.2 获取DOM元素

### 1. 非常规标签

- HTML - `document.documentElement`
- head - `document.head`
- body - `document.body`

### 2. 常规标签 - 不是不能获取非常规标签，只是一般不这么用

- `getElementById()` - 如果没有找到匹配的元素，返回`null`
- `getElementsByName()` - 返回伪数组
- `getElementsByClassName()` - 返回伪数组
- `getElementsByTagName()` - 返回伪数组
- `querySelector("选择器")` - 能在 css 里面写的选择器，这里都可以写 - 返回找到的第一个内容 - IE低版本不支持
- `querySelectorAll("选择器")` - 返回伪数组 - IE低版本不支持 - 获取的元素可以用 `forEach`

## 28.3 操作元素属性

H5标准中，**H5自定义属性以“data-”开头**

### 1. 原生属性 - 元素.属性名（注意，**class**属性除外 - 因为class是JS中的关键字 - 元素.className）

### 2. 自定义属性： - 也可以操作原生属性和H5自定义属性

- `setAttribute("属性名", "属性值")`
- `getAttribute("属性名")`
- `removeAttribute("属性名")`

3. H5自定义属性 - 每个元素有个 `dataset` 属性，里面包含了所有H5自定义属性，其中的 key，不带“data-”

```
1 // 设置元素标签上的data-a属性，值为1
2 div.dataset.a = 1
3
4 // 删除元素标签上的data-a属性
5 delete div.dataset.a
```

H5中元素身上有个“classList”属性，包含了元素身上设置的所有类名

## 28.4 操作元素文本内容

1. innerHTML - 操作元素的超文本内容
2. innerText - 操作元素的文本内容 - 标签内容不获取
3. value - 操作表单元素的value属性

在写选项卡类的代码块时，在遍历时给多个元素添加点击事件时需要注意，如果直接用for循环中的变量（i）传参，可能会导致所有元素传入的参数相同 - 函数定义时不使用参数的值，在调用的时候才会读取参数的值。

可以通过使用关键字 `this` 解决问题。或者使用数组的 `forEach(function(item, index, arr){})` 方法解决问题。

## 28.5 操作元素样式

1. 行内样式
  - `style` - 元素.style - 只能获取行内样式
2. 非行内样式
  - `window.getComputedStyle(“元素”)` - 标准浏览器
  - `currentStyle` 属性 - IE低版本

```

1 | function getStyle(ele, style) {
2 |     // 判断 window 里面有没有 getComputedStyle()
3 |     if ("getComputedStyle" in window) { // 此处必须是这个函数名的字符串, 详见13章
4 |         // 标准浏览器
5 |         return window.getComputedStyle(ele)[style]
6 |     } else {
7 |         // IE 低版本
8 |         return ele.currentStyle[style]
9 |     }
10 |

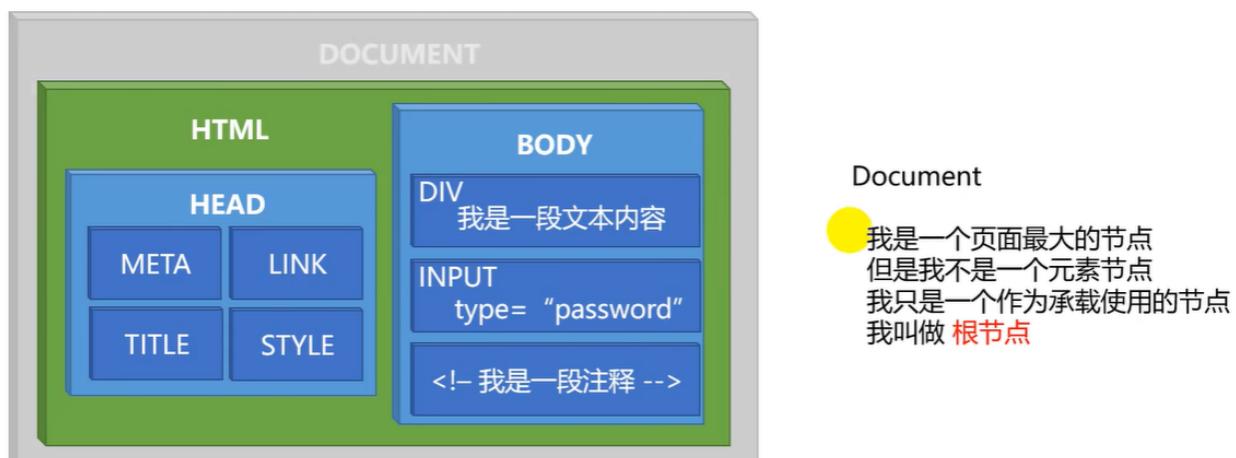
```

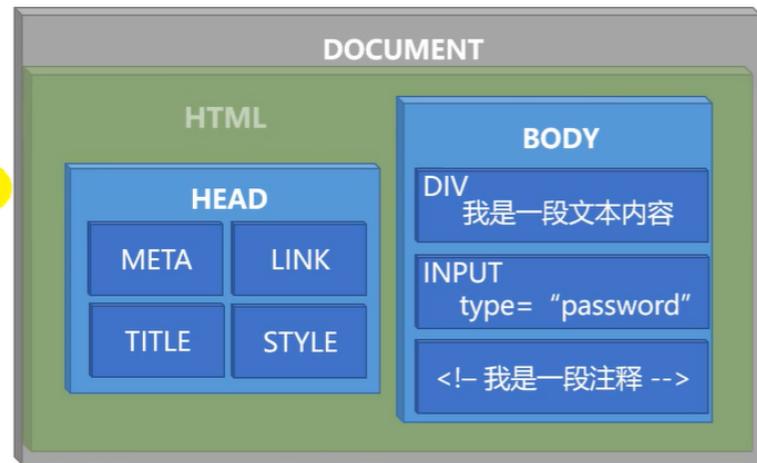
设置元素样式 - 只能设置元素行内样式 - 前段 JS 理论上是不可以设置元素的非行内样式

- 元素.style.样式名 = 值

## 28.6 DOM 节点

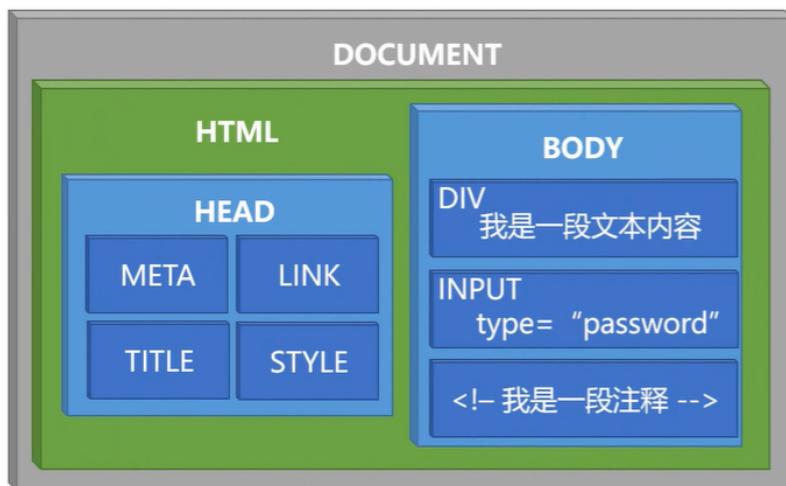
- DOM 节点，就是构成页面的每一个组成部分
- 元素/注释/文本 等内容都是一个一个的节点
- 常用的四种节点：
  - 元素节点：页面中的每一个标签
  - 文本节点：写在标签里面的文本内容
  - 属性节点：写在元素上的每一个属性
  - 注释节点：页面中书写的注释内容





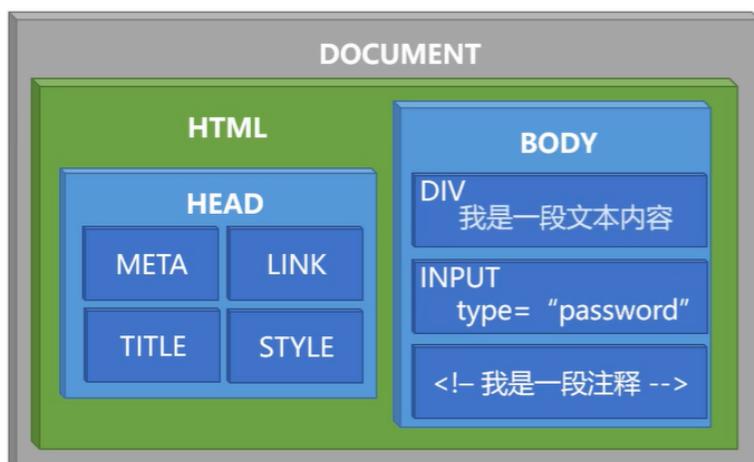
html

我是一个页面最大的**元素节点**  
我包含着页面所有的元素节点  
我是最大的元素节点  
我叫**根元素节点**



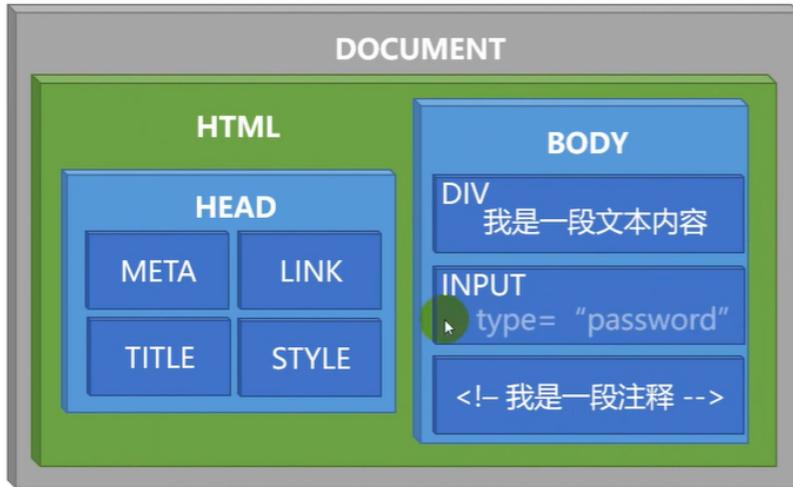
head / body / div / ul / ...

我们都是普通的元素节点  
只是所处的位置不一样  
各自的功能不一样  
我们都可能是父元素  
也有可能是子元素  
我们都叫**元素节点**



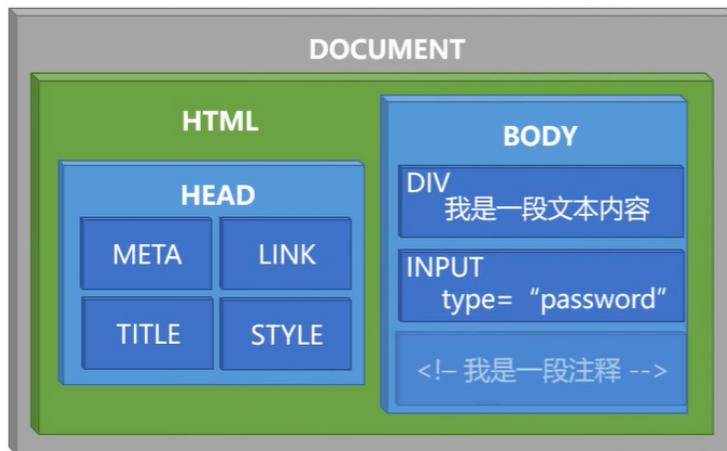
文本内容

我是一段文本包含**换行**和**空格**  
我也是一个节点  
很多标签**里面**都包含我  
我叫做**文本节点**



元素属性

我是一个元素身上的属性  
我也是一个节点  
用来描述和修饰这个元素  
很多元素上都有我  
我叫做 **属性节点**



注释内容

我是一段注释  
我也是一个节点  
我是单独书写的，不会显示在页面  
我叫做 **注释节点**

操作节点：

1. `childNodes` - 元素.`childNodes` - 获取元素的所有子节点（伪数组）
2. `children` - 元素.`children` - 获取元素的所有子元素节点（伪数组）
3. `firstChild` - 元素.`firstChild` - 获取元素的第一个子节点（- `lastChild`）
4. `firstElementChild` - 元素.`firstElementChild` - 获取元素的第一个子元素节点（- `lastElementChild`）
5. `previousSibling` - 元素.`previousSibling` - 元素的上一个兄弟节点（- `nextSibling`）
6. `previousElementSibling` - 元素.`previousElementSibling` - 元素的上一个兄弟元素节点（- `nextElementSibling`）
7. `parentNode` - 元素.`parentNode` - 元素的父节点
8. `parentElementNode` - 元素.`parentElementNode` - 元素的父元素节点
9. `attributes` - 元素.`attributes` - 元素的所有属性节点

## 28.7 节点属性

- 属性节点：

- 元素身上的属性
- 每一个属性是一个节点

- 节点属性:

- 用来描述节点的信息
- 不同节点有相同的属性名，但是值不一样

- 节点属性有三个:

1. `nodeType` - 以数字形式表示节点类型

- 元素节点: 1
- 属性节点: 2
- 文本节点: 3
- 注释节点: 8

2. `nodeName` - 节点的名称

- 元素节点: 大写标签名
- 属性节点: 属性名
- 文本节点: `#text`
- 注释节点: `#comment`

3. `nodeValue` - 节点的值

- 元素节点: `null`
- 属性节点: 属性值
- 文本节点: 文本内容（包括换行和空格）
- 注释节点: 注释内容（包括换行和空格）

- 创建节点:

1. `createElement()`
2. `createTextNode()`
3. `createComment()`

- 插入节点:

1. `appendChild()`
2. `insertBefore()` - 父节点.`insertBefore(要插入的节点, 插入到哪个子节点之前)`

- 删除节点:

1. `removeChild()`
2. `remove()`

- 替换节点:

## 1. replaceChild()

- 克隆节点:

1. `cloneNode()` - 参数默认是`false`: 不克隆后代节点, 选`true`, 克隆所有子节点

注意: 在动态向页面中添加节点时 - 例如表格 - 如果使用传统的直接添加节点的方法, 数据越多, 操作DOM次数越多, 效率很低。

为了解决这个问题, 可以吧数据都添加到一个div标签里面后, 再直接吧这个div标签放到页面中。但是这样又产生了一个问题, 这样加到页面中的节点, 外面多了一层div。这个div可能会导致显示异常 - 例如在表格中。

这时可以选择使用 [文档碎片节点](#) - 可以想象成一个筐

- 可以承载节点
- 当把框向页面元素添加的时候, 筐不会进入页面, 而是把框内的元素“倒”到页面中

## 28.8 获取元素尺寸

元素的尺寸: 内容区域 + padding + border

1. `offsetWidth / offsetHeight` - 元素内容区域 + padding + border - `display: none` 后是0
2. `clientWidth / clientHeight` - 元素内容区域 + padding - `display:none` 后是0

## 28.9 获取元素偏移量

获取元素偏移量 - 一个元素相对于参考系的坐标位置

1. `offsetParent` - 拿到该元素获取偏移量的时候的参考父级
2. `offsetLeft / offsetTop`

知识点补充:

**margin塌陷** - 原理:父子嵌套元素在垂直方向的margin,父子元素是结合在一起的,他们两个的margin会取其中最大的值.

正常情况下,父级元素应该相对浏览器进行定位,子级相对父级定位.

但由于margin的塌陷,父级相对浏览器定位.而子级没有相对父级定位,子级相对父级,就像坍塌了一样.

## 2. 触发bfc(块级格式上下文), 改变父级的渲染规则

方法:

改变父级的渲染规则有以下四种方法, 给父级盒子添加

(1) position:absolute/fixed

(2) display:inline-block;

(3) float:left/right

(4) overflow:hidden

**margin合并** - 原理: 两个兄弟结构的元素在垂直方向上的margin是合并的

margin合并问题也可以用bfc解决, 但是会改变HTML结构, 所以在实际应用时, 在margin合并这个问题上, 我们一般不用bfc, 而是通过只设置上面的元素的margin-bottom来解决距离的问题

## 28.10 获取浏览器窗口尺寸

- BOM级别的获取 - 包含滚动条
  - innerWidth
  - innerHeight
- DOM级别的获取 - 页面部分的尺寸
  - document.documentElement.clientWidth
  - document.documentElement.clientHeight

## 28.11 元素的常用事件 (JS自带的原生事件全小写)

### 1. 鼠标事件

1. click - 鼠标左键单击
2. dblclick - 鼠标左键双击
3. contextmenu - 鼠标右键单击
4. mousewheel - 滚轮事件
5. mousedown - 鼠标按下 - 不光是左键
6. mouseup - 鼠标抬起

7. mousemove - 鼠标移动
8. mouseover - 鼠标移入
9. mouseout - 鼠标移出
10. mouseenter - 鼠标移入
11. mouseleave - 鼠标移出

## 2. 键盘事件 - 不是所有元素都能触发

- 表单元素（有选中效果）
- document
- window

1. keydown - 键盘按下
2. keyup - 键盘抬起
3. keypress - 键盘按下 - 必须要能输入按键内容到文本框的才会触发 - shift不会触发 - 中文输入法也不会触发

## 3. 浏览器事件

1. load - 页面加载完毕
2. scroll - 滚动
3. resize - 尺寸改变
4. offline - 网络断开
5. online - 网络恢复
6. hashchange - hash值改变的时候

## 4. 表单事件

1. change - 表单内容改变 - 表单失焦的时候和聚焦的时候不一样才会触发
2. input - 只要在表单输入内容就会触发
3. focus - 表单聚焦
4. blur - 表单失焦
5. submit - 表单提交事件 - 绑定给form标签使用的，当点击 **form标签内的button** 时触发
6. reset - 表单重置事件 - 绑定给form标签使用的，当点击 **form标签内的reset** 时触发

## 5. 拖拽事件 - 一般元素想触发拖拽行为，需要给元素加属性：draggable

- 完成一个完整的拖拽需要两个元素
  - 拖拽元素
  - 目标元素

1. dragstart - 拖拽开始 - 绑定给拖拽元素

2. drag - 拖拽中 - 绑定给拖拽元素
  3. dragend - 拖拽结束 - 绑定给拖拽元素
  4. dragenter - 拖拽（光标）进入目标元素的时候 - 绑定给目标元素
  5. dragleave - 拖拽（光标）离开目标元素的时候 - 绑定给目标元素
  6. dragover - 拖拽元素在目标元素里面移动 - 绑定给目标元素
  7. drop - 拖拽元素在目标元素内放手（鼠标） - 绑定给目标元素 - 必须要在dragover  
里面阻止默认行为
6. 触摸事件 - 必须要在移动端
1. touchstart - 触摸开始
  2. touchmove - 触摸移动
  3. touchend - 触摸结束
7. 其他事件

1. transitionend - 过度结束 - 当有过度属性的时候 - 过度几个属性触发几次
2. selectstart - 开始选择 - 当在页面中框选文字时触发
3. visibilitychange - 窗口隐藏和显示 - 只能绑定给document

## 29. 模板引擎

什么是模板引擎

- 帮助快速渲染页面
- 三体分离（结构 - 样式 - 行为 分开）
- 一般都是第三方文件引入使用

常见的模板引擎

- art-template - 前后端都可以使用
- underscore - 后端 JS 不能用
- e.js - 后端 JS 不能用
- template - 后端 JS 不能用

art-template特点

- HTML结构和 JS 代码分离
- 有自己独立的语法，但是也可以使用原生 JS 的语法
- 有自己独立的渲染机制

```

<script src="./js/template-web.js"></script>


<script type="text/template" id="tmp">

<h1>模板中的文字</h1>
<p>{{ title }}</p>
</script>

<script>
/*
1. 自己书写一个模板 - 需要把 display 改成 none
    => script 标签来定义模板
        -> 从来不会给 script 标签添加样式 - 防止模板display属性被改
        -> script 标签里面书写的內容本来在页面上就不显示
    => 需要添加一个 type 属性, 推荐属性值写成 text/html 或者 text/template
        -> 只要不写成 text/javascript, script 标签里面的内容就不会识别成 JS 代码
        -> 写成 text/html 或者 text/template 会有代码提示/补全
2. 在 JS 里面使用模板
    => 导入第三方文件
    => 使用 template 方法
        -> 语法: template(模板标签的 id 名, 数据)
        -> 返回值: 模板标签里面的html结构, 是一个字符串
*/
var str = template("tmp", {title: "abc"})

document.querySelector(".box1").innerHTML = str
</script>

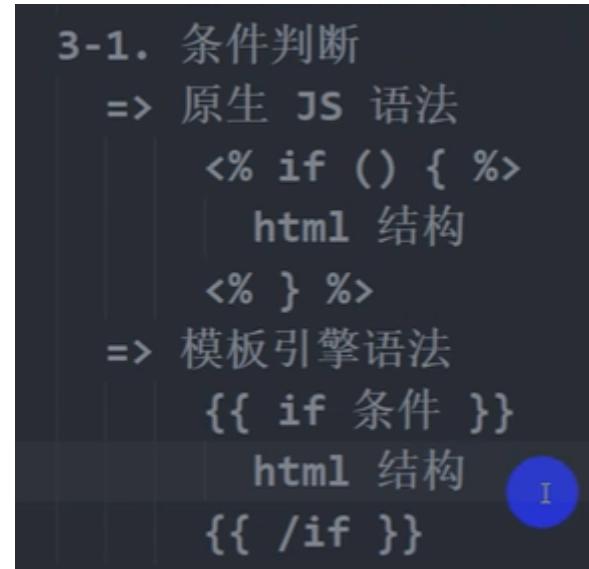
```

在模板里面使用的语法:

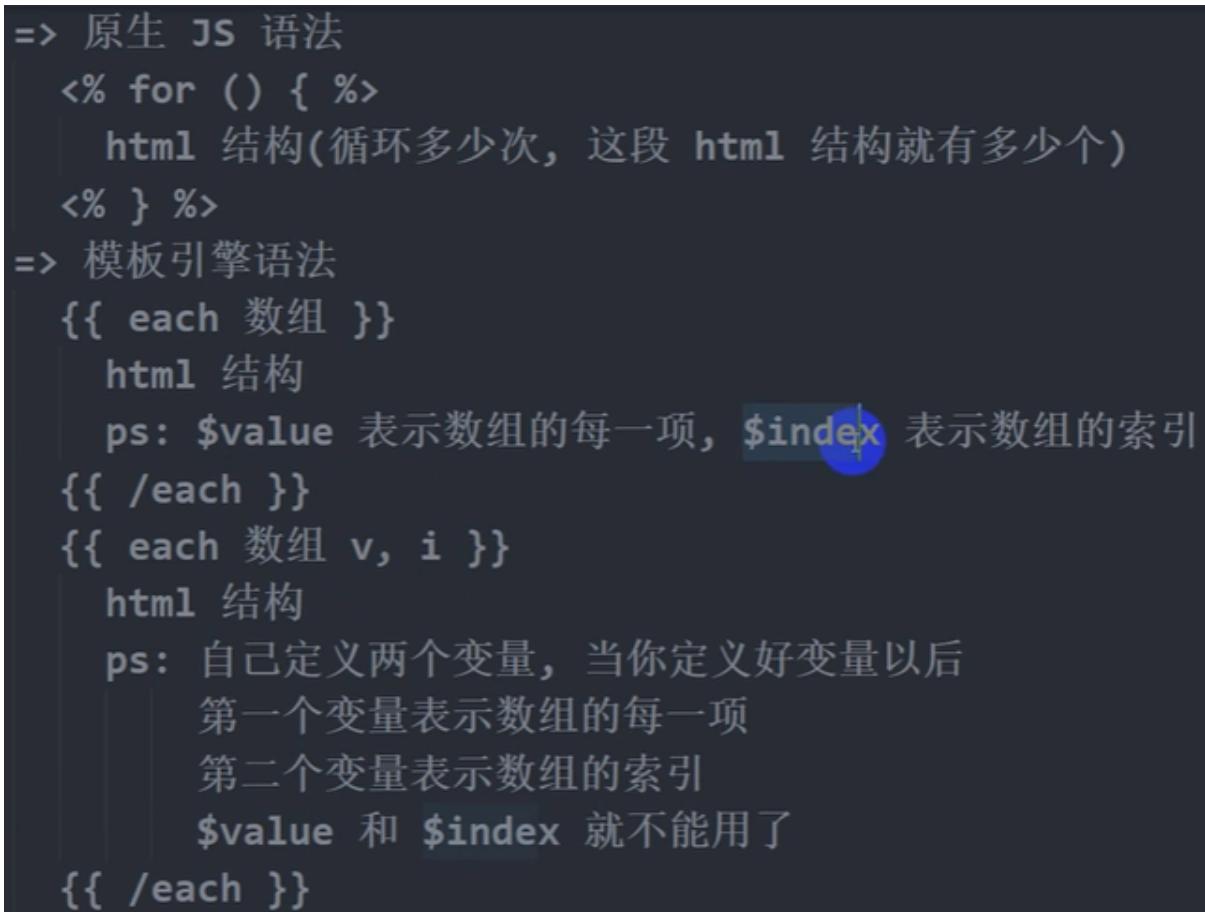
## 1. 输出内容的语法

- 原生 JS 输出
  - <%= 你要输出的变量 %> - 不会解析 html 结构字符串
  - <%- 你要输出的变量%> - 会解析 html 结构字符串
- 模板引擎语法输出
  - {{ 你要输出的变量 }} - 不会解析 html 结构字符串
  - {{@ 你要输出的变量 }} - 会解析 html 结构字符串
- 输出的时候可以写简单表达式（布尔表达式等）

## 2. 条件判断



### 3. 循环



此处跳过两集

[P159 模板引擎渲染页面](#)

[P160 购物车案例](#)

## 30. 事件

我们提前和浏览器约定好一些行为，当用户在浏览器触发这些行为的时候，有一个事件处理函数执行

### 30.1 事件三要素：

- 事件源：在谁的身上绑定事件
- 事件类型：什么事件
- 事件处理函数：当行为发生的时候，执行哪一个函数

### 30.2 事件绑定分类：

#### 1. dom0级 事件

- on...的形式 - `div.onclick = function() {...}`

#### 2. dom2级 事件

- 事件监听 - 不能混用，会报错

1. `addEventListener()` - 标准浏览器使用 - 事件源`.addEventListener("事件类型", 事件处理类型)` - 可以同时给一个事件绑定多个事件处理函数 - 顺序绑定顺序执行

2. `attachEvent()` - IE低版本 - 事件源`.attachEvent("on事件类型", 事件处理类型)` - 可以同时给一个事件绑定多个事件处理函数 - 顺序绑定倒序执行

### 30.3 事件解绑：

#### 1. 解绑dom0级事件：直接赋值为 `null` - `div.onclick = null`

#### 2. 解绑dom2级事件：

- `removeEventListener("事件类型", 要解绑的事件处理函数名)` - 也就是说，想要使用这个方法解绑事件处理函数，在绑定处理函数的时候，要写成具名函数
- `detachEvent("on事件类型", 要解绑的事件处理函数名)`

手动抛出异常：`throw new Error("报错信息")`

复杂数据类型的比较是比较 `地址`

```
1 // 封装事件绑定和解绑
2 function on(ele, type, handler) {
3     // 如果没有传递参数，报错
4     if (!ele) throw new Error("Please pass parameter according to the rules")
5     // 如果传递的不是元素节点，报错
6     if (ele.nodeType !== 1) throw new Error("Wrong Type")
7
8     // 兼容处理，就是查看，这个元素身上是否有相应的函数
9     if (ele.addEventListener) {
10         console.log("标准浏览器")
11     } else if (ele.attachEvent) {
12         console.log("IE 低版本")
13     } else {
14         console.log("原始浏览器") // ES3 以前
15     }
16 }
```

## 30.4 事件对象：

### 事件对象

- + 当一个事件触发的时候，对本次事件的描述
- + 例子：客服
  - => 当电话响的时候，接起来，聊
  - => 在接电话的时候，需要记录一些信息
    - > 什么地方打来的
    - > 说了什么事情
    - > 需要什么帮助
    - > 什么时间
    - > ...

### + 例子：鼠标按下行为

- => 当你在浏览器上触发点击行为的时候，要执行事件处理函数
- => 需要一些信息来记录
  - > 你点击的是哪一个元素
  - > 你点击的坐标是什么
  - > 你按下的是哪一个按键
  - > 你当前触发的事件类型是什么

如何获取事件对象：

- 标准浏览器：直接在事件处理函数上接收一个形参，会在事件触发的时候，有浏览器自动传递实参 - `addEventListener` 函数相同，在事件处理函数上接收形参



- IE低版本：不接收形参，使用`window.event` - 在标准浏览器中也可以使用

```
1 | div.onclick = function (e) {
2 |     e = e || window.event
3 |     console.log(e)
4 | }
```

## 30.5 鼠标事件的事件对象信息：

### 1. 按下按键：

- 0 - 左键
- 1 - 滚轮键
- 2 - 右键

### 2. 光标的坐标

- `clientX` 和 `clientY` - 光标距离可视窗口左上角的位置
- `pageX` 和 `pageY` - 光标距离文档的左上角的位置
- `offsetX` 和 `offsetY` - 光标距离元素左上角的位置 - 光标事件触发的元素（不是事件源） - 如果不想按照光标触发元素的左上角计算坐标，想以事件源来计算坐标 - > CSS 样式 `pointer-event: none`

代码运行比DOM快

## 30.6 键盘事件的事件对象信息：

### 1. 按下的是哪一个按键

1. 事件对象里面有一个 `key` 属性表示按下的那个按钮 - IE 低版本不支持
2. 事件对象里面有一个叫做 `keyCode` 的属性，以编码的形式表示按下的是哪一个按键 - FireFox < 20 的版本不支持，在火狐低版本使用 `which` 属性

### 2. 按下的是不是组合按键 - 四个属性值都是布尔值

- altKey
- ctrlKey
- shiftKey
- metaKey - IE 不支持

## 30.7 事件的传播

当你在一个元素上触发行为的时候，会按照 **结构父级** 的顺序向上传播行为，直到 **window** 为止

- 当事件触发的时候，会按照结构富姐的顺序向上传递同类型事件
- 事件对象里面有一个信息叫做 **path**，表示当前事件传播的途径

- 例如：当我们点击在 inner 身上的时候

- 也相当于点击在了 center 身上
- 也相当于点击在了 outer 身上
- 也相当于点击在了 body 身上
- 也相当于点击在了 html 身上
- 也相当于点击在了 document 身上
- 也相当于点击在了 window 身上

## 30.8 事件的目标、冒泡和捕获

1. 目标 - 准确触发事件的那个元素 - target

- => 当你给 **center** 绑定一个点击事件
  - > 你点击 **inner** 会触发
  - > 你点击 **center** 也会触发
  - > 两次事件触发的元素是不一样的
- => 在事件对象里面有一个属性叫做 **target**
  - > 表示本次事件触发的时候，准确触发的元素
  - > 我们叫做事件目标
- => 特点：**IE** 低版本不支持
  - > **IE** 低版本使用 **srcElement**

```
1 | var target = e.target || e.srcElement
```

## 2. 冒泡

- 冒泡：就是按照从 **目标** 到 **window** 的顺序依次触发事件
- 例如：你点击在 **inner** 身上，**inner** 就是本次事件的 **目标**
  - 执行 **inner** 身上的事件（如果有）
  - 执行 **center** 身上的事件（如果有）
  - ...
  - 执行 **window** 身上的事件（如果有）
- 注意：如果某一层结构上没有事件，那么就是没有事件处理函数执行，会继续传播

## 3. 捕获

- 捕获 就是按照从 window 到 目标 的顺序依次触发事件

- 例如：你点击在 inner 身上，inner 就是本次事件的 目标

- 执行 window 身上的事件（如果有）
- 执行 document 身上的事件（如果有）
- ...
- 执行 inner 身上的事件（如果有）

- 注意：如果某一层结构上没有事件，那么就是没有事件处理函数执行，会继续传播

使用 `addEventListener` 方法的第三个参数决定事件是按照 冒泡 还是捕获来执行

- `false`: 冒泡 - 默认值
- `true`: 捕获

## 30.9 移入、移出事件的区别

1. `mouseover / mouseout`

2. `mouseenter / mouseleave`

- 行为：都是移入和移出行为
- 事件：`mouseenter / mouseleave` 不会进行事件传播

## 30.10 阻止事件传播

1. `e.stopPropagation()` - 标准浏览器

2. `e.cancelBubble = true` - IE 低版本

```
1 | 兼容写法:  
2 | // 1  
3 | if (e.stopPropagation) {} else {}  
4 | // 2  
5 | try {e.stopPropagation()} catch (error) {}
```

## 30.11 事件委托

利用事件冒泡，把子元素的事件绑定在一个共同的父元素上

1. 循环绑定事件

2. 事件委托 - 把子集身上需要绑定的事件绑定给一个共同的结构父级 - 使用target来判断点击的元素

```
1 | ul.onclick = function (e) {[  
2 |   e = e || window.event  
3 |   // target 就是点击的元素  
4 |   var target = e.target || e.srcElement  
5 |  
6 |   if (target.nodeName === "LI") {  
7 |     console.log("点击的是LI", target)  
8 |   }  
9 | }]
```

优点：

- 减少元素的事件绑定
- 减少 DOM 操作，提高性能
- 对于新加进来的元素也可以执行事件，不需要重新绑定

## 30.12 浏览器默认行为

浏览器默认行为：不需要手动绑定，本身就带有的事件行为

- a 标签 - 自带点击行为
- form 标签 - 自带表单提交
- 框选 - 自带框选效果
- 右键单击 - 自动弹出菜单
- ...

1. 在同类型事件里面进行阻止 - 例如在 a 标签的点击事件里面阻止自动跳转

- e.preventDefault() - 标准浏览器
- e.returnValue = false - IE 低版本
- return false - 有的事件好用有的事件不好用

## 31. 自执行函数

函数调用的一种方式 - 不会预解析，在代码执行到它的时候，预解析和调用同步完成 - 以下三种写法相同

1. (function (){})()
2. ~function() {}
3. !function() {}

单独书写 JS 文件的时候使用 - 为了保护变量不污染全局 - 把 JS 代码全都写到函数中

如果有需要跨页面使用的变量，写到全局中

当使用第一种写法时，要注意。

JS 中，当每行结尾不以 ; 结尾时，每行不能以 () 、 [ ] 、 "" 开头，所以要在前面加一个 ;

## 32. this

this 是一个使用在作用域内部的关键字

指向：

- 全局使用：window
- 函数内使用：不管函数怎么定义，不管函数在哪定义，只看函数的调用（箭头函数除外）
  - 普通调用 --> window
  - 对象调用 --> 点前面是谁就是谁
  - 定时器处理函数 --> window
  - 事件处理函数 --> 事件源
  - 自执行函数 --> window

改变 this 指向

1. call() - fn.call() - obj.fn.call() - 第一个参数是函数内部的this指向，后面的参数依次给函数传参

特点：会立即执行函数（不适合用作定时器处理函数或者事件处理函数）

2. apply() - fn.apply() - obj.fn.apply() - 第一个参数是函数内部的this指向，第二个参数是一个数组/伪数组用来给函数传参

特点：会立即执行函数（不适合用作定时器处理函数或者事件处理函数）

3. bind() - fn.bind() - obj.fn.bind() - 第一个参数是函数内部的this指向，后面的参数依次给函数传参

特点：不会立即执行函数，会返回一个改好指向的新函数

可以用来给事件处理函数传参

## 33. ES6 定义变量

ES6 确认了两个定义变量的关键字

1. let - 变量 - 声明时可以不赋值 - 可以被修改
2. const - 常量 - 声明时必须赋值 - 不能被修改（对象里的属性可以修改）

var

- 会进行预解析
- 可以声明重复变量名
- 没有块级作用域

let / const

- 不会进行预解析，必须先定义后使用 - 会报错
- 不能声明重复变量名
- 有块级作用域 - if、while等( {} 内)

尽量使用 let 和 const 定义，书写代码的时候尽可能优先使用 const

## 34. 箭头函数

匿名函数

- var fn = function() {}
- var obj = {fn: funciton() {}}
- setTimeout(function() {}, 0)
- setInterval(function() {}, 0)
- [].forEach(function() {})
- div.onclick = function() {}
- div.addEventListener("click", function() {})
- ...

语法: () => {}

特性:

1. 如果只有一个形参，可以省略小括号不写: a => {}
2. 如果代码段只有一句话，可以省略大括号不写，并且自动 return 这句话的结果
3. 箭头函数没有 arguments

4. 箭头函数中没有 `this` - 外部作用域的 `this` 就是箭头函数中的 `this` - 所以箭头函数中的 `this` 也无法改变

## 35. 点点点运算符

- 展开运算符 - 当在函数的实参位置或者数组或者对象里面使用的时候是展开
- 合并运算符 - 当在函数的形参位置使用的时候是合并

## 36. 解构赋值

快速从对象或者数组里面获取一些数据

分成两种：

1. 解构数组 - `let [变量1, 变量2...] = [数据1, 数据2...]`

可以用来交换变量 - `var [b, a] = [a, b]`

2. 解构对象 - `let {key1, key2, ...} = {键值对1, 键值对2, ...}` - `let {key1: 别名, key2: 别名, ...} = {键值对1, 键值对2, ...}`

## 37. 对象的简写方式

ES6标准下，有一个对象的简写方式

1. 当 `key` 和 `value` 一模一样的时候，可以只写一个

```
let age = 18

let obj = {
    name: 'Jack',
    age, // 等价于 age : age
    gender: 'gender' // 不能省略
}
```

2. 当某一个 `key` 的值是一个函数，并且不是箭头函数的时候，可以直接省略 `function` 关键字和 `冒号(:)` 不写

```
let obj = {
    name: '我是 obj 对象',
    f1: function () { console.log(this) },
    f2: () => { console.log(this) },
    f3 () { console.log(this) }
}
```

```
obj.f1() // obj
obj.f2() // window
obj.f3() // obj
```

## 38. 面向对象开发

### 38.1 创建对象的四种方式

1. 字面量创建 - var obj = {}
2. 内置构造函数 - var obj = new Object()
3. 工厂函数创建对象

- 先自己做一个工厂函数
- 再使用这个工厂函数创建对象

```
1 | function createObj(name, age, gender) {
2 |     let obj = {}
3 |
4 |     obj.name = name
5 |     obj.age = age
6 |     obj.gender = gender
7 |
8 |     return obj
9 |
10|
11| let o1 = createObj("Max", 25, "男")
```

### 4. 自定义构造函数创建

- 自己书写构造函数 - 就是普通函数，只有调用的时候和 `new` 关键字连用，才有构造函数的能力。只要和 `new` 关键字连用，`this` 指向当前实例对象（`new` 前面的那个变量名）

- 使用构造函数创建对象

```
1 | function createObj(name, age, gender) {  
2 |   // 1 自动创建对象  
3 |  
4 |   // 2 手动向对象上添加内容  
5 |   this.name = name  
6 |   this.age = age  
7 |   this.gender = gender  
8 |  
9 |   // 3 自动返回这个对象  
10| }  
11|  
12| let o1 = new createObj("Max", 25, "男")
```

## 38.2 构造函数的书写和使用

1. 调用必须有 `new` 关键字

如果没有，就没有自动创建对象的能力

2. 不要在构造函数中写 `return`

`return` 基本数据类型不起作用，`return` 引用数据类型构造函数白写

3. 如果不需要传参，创建实例对象的时候可以不写小括号 - 最好还是写上

4. 推荐首字母大写

5. 使用 `new` 关键字创建对象，创建出来的对象叫实例对象，创建的过程叫实例化过程

6. 在构造函数内写书写方法，实例化时每创建一个实例对象，就会在内存空间里面创建一个一模一样的该方法，会占用多余空间 - 粗暴的解决方法，把函数写在外面（或者在外面用一个对象包裹），在构造函数里面赋值 - 更好的解决方法，`prototype`

## 38.3 `prototype`

每一个函数天生自带一个属性叫做 `prototype`，它是一个对象，可以往里面添加一些内容

`prototype` 中有个属性 - `constructor` - 表示是哪一个构造函数伴生的原型对象

## 38.4 \_\_proto\_\_

每一个对象天生自带一个属性叫做 `__proto__`，指向所属构造函数的 `prototype`

```
1 | function Person() {}  
2 | Person.prototype.sayHi = function () {console.log("Hello world")}  
3 |  
4 | let p = new Person()  
5 | console.log(p.__proto__)
```

## 38.5 对象访问机制

当访问一个对象的成员的时候，如果对象自己本身有，直接返回，停止查询；如果对象本身自己没有，会自动去 `__proto__` 上访问。如果有就返回结果，停止查询；如果没有，往上层找，如果一直到顶级对象的 `__proto__` 都没有，返回 `undefined`（`Object.prototype` 是顶级原型对象）

- 定义1：每一个对象都有 `__proto__`
- 定义2：每一个函数都有 `prototype`
- 问题：
  - 每个构造函数的 `prototype` 是一个对象，那么他应该也有一个 `__proto__`，这个 `__proto__` 指向谁？
  - 构造函数本身也是一个对象，他应该也有一个 `__proto__`，这个 `__proto__` 又指向谁？

当一个对象，没有准确的构造函数来实例化的时候，我们都看做是内置构造函数 `Object` 的实例，例如：`Array.prototype`

任何一个对象开始出发，按照 `__proto__` 开始向上查找，最终都能找到 `Object.prototype`，我们管这个使用 `__proto__` 串联起来的对象链状结构，叫做 **原型链**

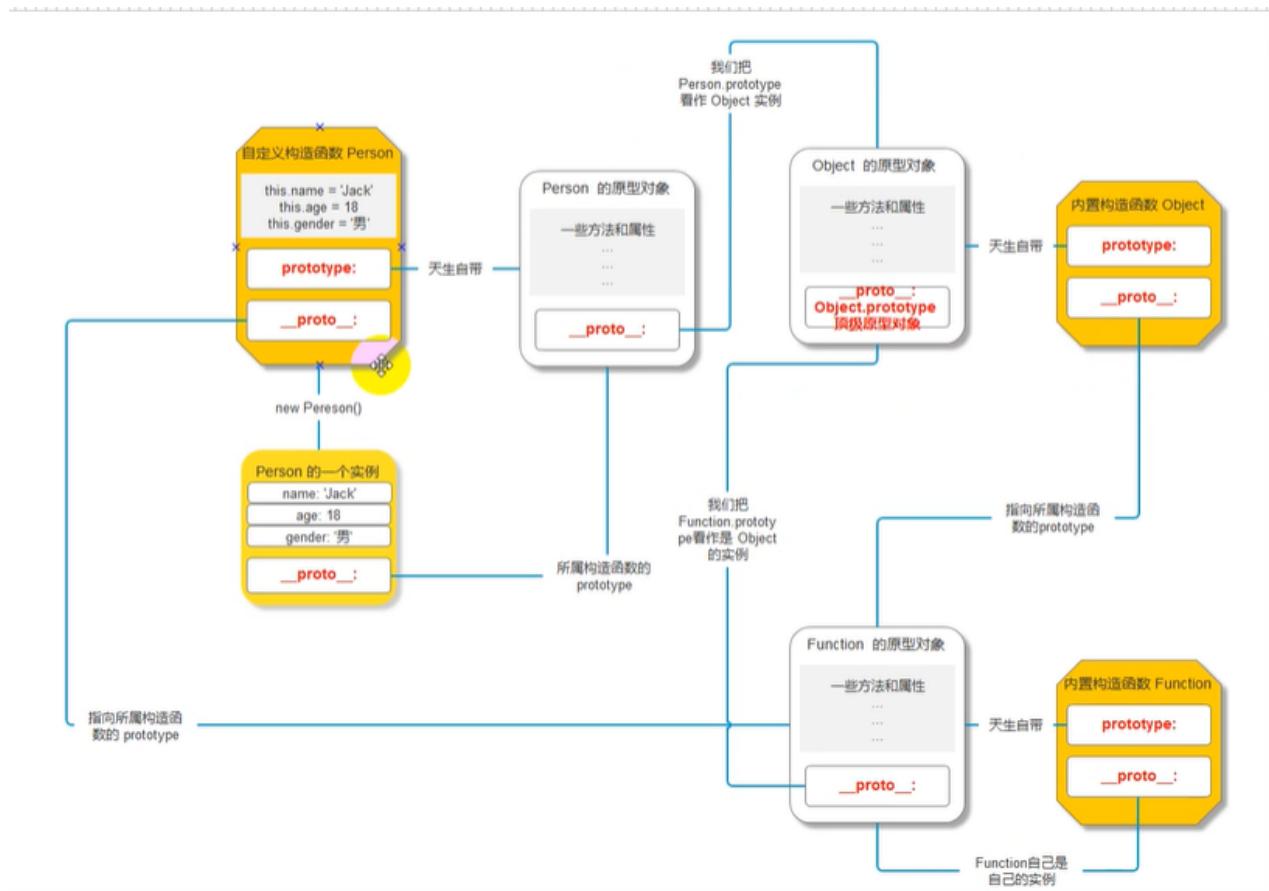
## 38.6 原型链

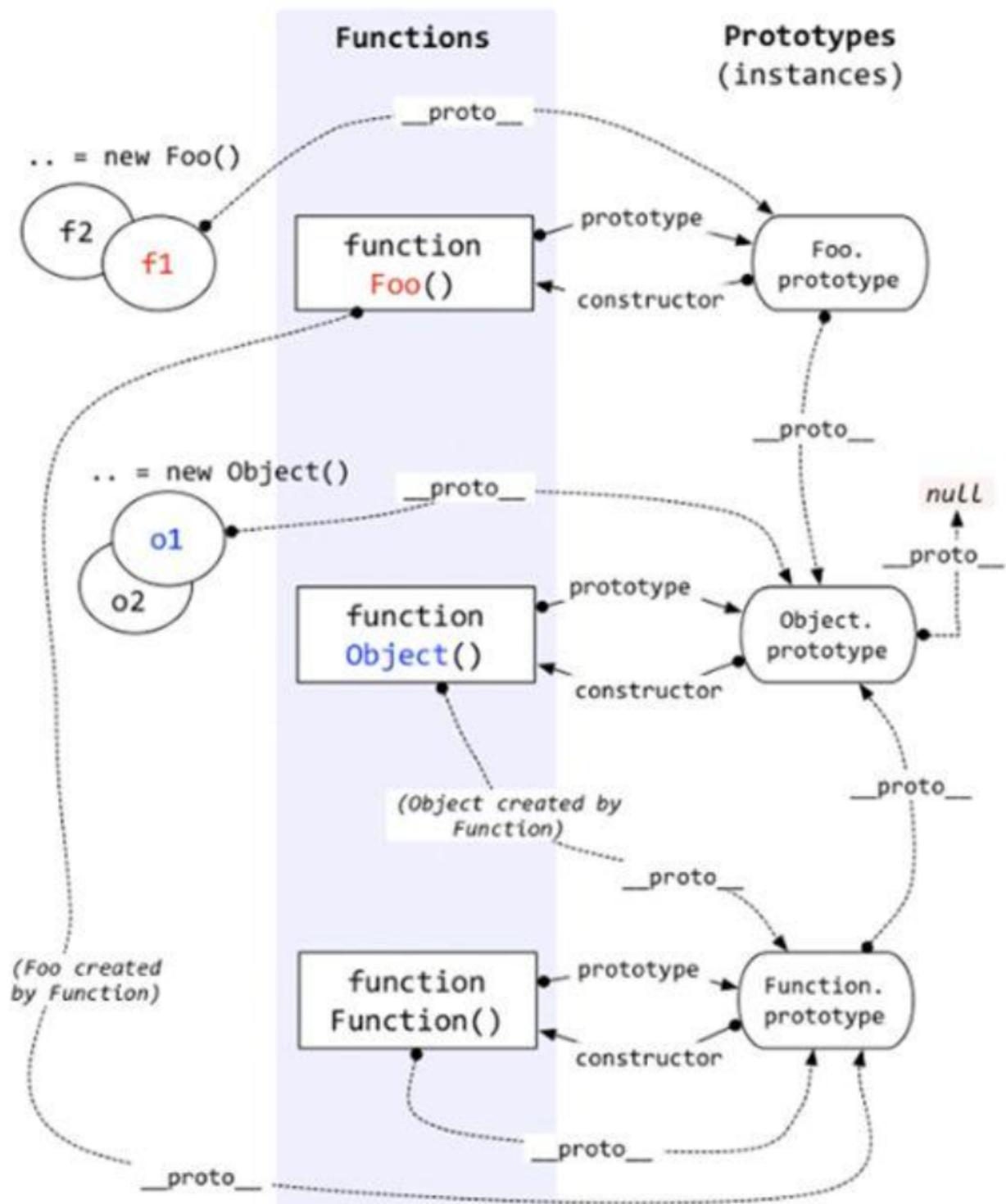
从任何一个对象出发，按照 `__proto__` 串联起来的对象链状结构

原型链的作用：为了对象访问机制而存在

`constructor` 属性（构造器）：

- 只有函数天生自带的那个 prototype 上才有，表示是哪一个构造函数自带的原型对象
- 判断数据类型





## 39. 判断数据类型

### 1. typeof

- 准确的判断基本数据类型
- 对于复杂数据类型并不准确

### 2. constructor

- 利用原型的属性

- 利用对象访问机制
- 无法识别 `undefined` 和 `null`

### 3. instanceof

- 对象 `instanceof` 构造函数
- 不好检测基础数据类型

### 4. `Object.prototype.toString.call()`

- `Object.prototype.toString.call(要检测的数据类型)`

## 40. 了解对象

对象：

- 数据类型的一种
- 以键值对的形式储存数据
- 因为 `__proto__` 和 原型链，可以访问自己没有的属性

`for in`循环：

- 专门遍历对象
- 遍历对象身上所有的 可枚举 的属性（包括原型链上的所有属性）
  - 一种是自定义的属性
  - 我们可以设置为可枚举属性

对象自己的方法：

1. `hasOwnProperty()` - 查看是不是自己的属性

2. `defineProperty()` - 一种给对象添加属性的方法 - 数据劫持 - 可以给一个自己设置的属性设置各种各样的行为状态，如可不可以修改，是否可枚举

```
// 数据劫持
Object.defineProperty(obj, 'username', {
    // 对 obj.gender 属性进行一系列的配置
    // value: '男', // 这个成员值
    // enumerable: false, // 是否可枚举
    get () {
        return prop.firstName + prop.lastName
    },
    // 当你想修改劫持的数据的时候
    // 就会触发这个 set 函数
    // 接收一个形参，就是你想修改的值
    set (val) {
        console.log('你想修改')
        console.log('你想改成：', val)
        // 通过在 set 里面修改 prop 对象里面的内容
        // 来达到修改 username 属性
    }
})
```

## 41. ES6 的类

在ES5以前，JS 使用 函数充当 构造函数（类）

ES6 引入一个 类 的概念 - 就是使用一个新的关键字来定义 构造函数（类），定义完后，就是一个类，不能当做函数调用，只能通过 `new` 来得到一个对象，不和 `new` 一起用会报错

```
1 function Person() {
2     this.name = "Jack"
3     this.age = 18
4 }
5
6 let p1 = new Person()
7 console.log(p1)
8
9 // 本质还是一个函数，是一个函数就可以直接调用，
10 // 当将它当做普通函数来执行的时候，
11 // 没有了创造对象的能力，this 的指向也改变了
```

```
12 Person() // this -> window
13
14 Class Person() {
15     // 构造器
16     constructor(name, age) {
17         this.name = name
18         this.age = age
19     }
20
21     // 原型上的方法
22     init() {}
23
24     move() {}
25 }
26
27 let p2 = new Person("Jack", 18)
```

## 42. 模块化开发

为什么需要模块化开发？

例：分页器， Pagination， creEle， setCss

creEle 和 setCss 其实在很多地方都可以用

按照开发习惯，应该把它们分开放

例如：

- a.js - 里面放 creEle 和 setCss
- b.js - 里面放 Pagination

把一类方法放在一个单独的 js 文件里面 - 为了方便使用的时候，只引入这一类方法的文件

这样一个封装好的 JS 文件叫做一个模块

什么是模块化开发？

- 多个 JS 文件之间相互配合来实现效果
- 一个 JS 文件里面只封装一类内容

问题：

1. JS 文件引入顺序

2. JS 文件之间的相互依赖不清晰

3. JS 文件内部变量会污染全局

## ES6 Module

- 2015年发布，ES6 语法里面自带了一个模块化标准 - 但是各大厂商并不买账 - 兼容麻烦
- 2016年，Vue 出现了，其中出现了一个脚手架（开发的大框架直接给搭建好，只需要填东西）
- 搭建这个架子的时候，内置了 ES6 模块化标准
- 2018年，各大浏览器厂商开始原生支持 ES6 模块化标准
- 2018年中，Chrome率先原生支持 ES6 模块化
- 特点：页面必须在服务器上打开
  - live server插件
  - 如果想使用模块化语法，`script`标签需要加一个属性 `type="module"`
- 使用：
  - 每一个文件都可以作为一个独立模块，也都可以作为整合文件
  - 导出语法：
    - `export default` 导出的内容 - 一个文件只能有一个 `export default` - 如果要导出多个内容，需要以对象或者数组的形式导出
    - `export var num = 200` - 向外导出一个变量 - 可以写多个
  - 导入语法：
    - 接收 `export default` 导出的内容 - `import 变量 from 文件`

```
// modA 接受的是 a.js 文件里面 export default 后面的一整个内容
import modA from './a.js'
```
    - 接收 `export` 导出的变量  
`import { 接收变量 } from 文件`

```
// 我需要依赖 b.js 文件里面的方法  
import { num, str } from './b.js'
```

- 问题:

- 依赖前置 - 和 AMD 一样

2020年，ES2020发布新的标准，多了一个按需加载的模块化

语法: import(要加载的文件).then(function(res) {})

- 浏览器支持不好 - 必须在服务器上打开 - 一旦项目上线，肯定是服务器打开，所以这个问题影响不大

## 43. http传输协议

### 43.1 http传输协议:

- 前后端交互的方式
- 前端以什么样的形式发送数据给后端
- 后端以什么样的形式返回数据给前端

### 43.2 传输协议:

#### 1. 必须经历四个步骤

1. 建立连接
2. 发送请求（前端 -> 后端）
3. 返回响应（后端 -> 前端）
4. 断开连接

#### 2. 只能由前端发起

#### 3. 一次只能说一个事情

#### 4. 前后端交互只能交互字符串 - 中文会转成url编码

### 4.3.3 一个请求的四个步骤

1. 建立连接 - 基于 TCP/IP 协议的三次握手 - 为了保证通道的链接

浏览器和服务器做的

2. 发送请求

前端发送请求给后端，必须以 请求报文 的形式发送

一个特殊格式的字符串文件（由浏览器进行组装）

3. 接收响应

每一个响应是由服务端接收到前端的请求后，给出的回复

必须以响应报文的形式发送给前端

4. 断开连接

- 基于TCP/IP协议的四次挥手
- 为了保证断开连接

### 4.3.4 响应状态码

以一个数字表示本次请求的响应状态 - 成功、失败

100 ~ 599，分成五类：

- 100 ~ 199：表示连接继续
- 200 ~ 299：表示各种意义上的成功
- 300 ~ 399：表示重定向
- 400 ~ 499：表示客户端错误
- 500 ~ 599：表示服务端错误

常见状态码：

- 101 - 连接继续
- 200 - 通用成功
- 302 - 临时重定向 - 本次请求临时使用 服务器 来决定浏览器跳转的页面
- 301 - 永久重定向 - 终身只要访问这个地址，就会重新切换到新的地址
- 304 - 缓存 - 当访问过一遍这个页面以后，浏览器会自动缓存，再访问同一个地址的时候，不会向服务器发送请求，而是从缓存里面获取
- 403 - 访问权限不够
- 404 - 访问地址不存在

- 500 - 通用服务器端错误
- 501 - 维护或过载

## 43.5 请求方式

前端和后端的交互手段

常见的请求方式： - 无本质区别

- HTTP/1.0
  - GET: 倾向于获取的方式 - 大部分都是给够短一些参数，用来获取一系列数据
  - POST: 倾向于给服务器一些数据 - 大部分都是登录，给服务器一些信息，返回一个简单地结果
  - PUT: 倾向于给服务器一些信息，但是添加使用 - 大部分做注册，给服务器一些信息，把这个信息存起来
  - HEAD: 用来获取服务器头信息
- HTTP/1.1
  - DELETE: 倾向于删除
  - CONNECT: 管道连接改变代理链接使用
  - PATCH: 倾向于给服务器一些信息，修改一些信息 - 大部分用于完善用户资料
  - OPTIONS: 用于获取服务器性能，但是需要服务端同意

### GET 和 POST 请求方式的区别

GET:

- 语义是获取
- GET 携带参数的方式是 `queryString`，在地址栏后面直接拼接，不在请求体里面
- GET 理论上携带数据无限，但是因为浏览器地址栏有限 - IE 2kb
- 会被浏览器主动缓存
- 明文发送
- 只能发送url编码的数据（ASCII码），如果是中文会自动转码

POST:

- 语义是给
- POST 携带参数的方式是 `requestBody`，在地址栏里面没有，在请求体里面
- GET 理论上携带数据无限，但是会被服务器限制
- 不会被浏览器主动缓存
- 暗文发送

- 理论上可以发送任意格式的数据，但是要和请求头里面的 content-type 配套

## 43.6 Cookie

浏览器端本地存储空间，用来存储一些数据

特点：

- 按照域名存储 - 在储存的域名下可以访问，换一个域名就不能访问
- 按照文件路径存储 - 上级目录不能访问
- 按照字符串形式存储 - "key=value; key=value"
- 存储大小：4kb左右，50条左右
- 时效性：默认会话级别 - 可以手动设置 cookie 时效
- 操作权限 - 前后端都可以操作
- 请求自动携带 - 只要cookie空间有数据，那么在发送任何一个请求的时候，自动携带

前端操作cookie：

### 1. document.cookie

- 可读写
- document.cookie = "key=value" (新增)
  - 不可删 - 可以通过修改过期时间来删除
  - 一次只能设置一条
  - 可以书写对本条cookie的描述
    - expires - 过期时间 - 默认会话级别 - expires=时间对象
    - path - 存储路径 - 如果不设置会按照目前的目录结构进行存储 - key=value;path=你要存储的路径
  - 自动转换成字符串格式 - 一般存储简单数据
  - 如果设置同名的cookie就是修改 - 必须是在相同目录下，不然会储存一条新的 cookie

## 43.7 session

存在于服务器端的储存空间，打开的一瞬间，就会生成一个“密码”，这个密码会自动存储到cookie里面，等到返回后端的时候，会把这个带回去

FE



BE



开启 session 空间：

- `session_start()` - 只要想向session空间里面存储内容或者获取内容，必须要先开启后使用

```
// 1. 开启 session 空间
/*
    当我在浏览器打开 localhost/01_session.php 文件的时候
    就会执行 session_start() 这段代码
    后端就会开启一个存储空间，同时生成一段 密码
    同时把密码的一半放在 cookie 里面

    前端访问 localhost/01_session.php 本身就是一个请求
    此时后端是会给回一个响应
    响应回到前端了，那么此时看到 cookie 空间里面就应该有一个 session_id 的存在
*/
```

通过 PHP 内置的关联型数组 - `$_SESSION` 可以设置session

## 44. ajax

前后端交互的一种手段，通过 `JS` 向 `服务器` 端发起请求（服务端返回的响应会返回给 `JS`，不会直接显示在页面上） - 依赖于浏览器

ajax: a: async, j: JavaScript, a: and, x: xml

使用方式：

1. 找到一个对象帮助发送 ajax 请求

- XMLHttpRequest(): 实例化对象帮助发送 ajax 请求

2. 对本次请求进行一些配置

open(请求方式, 请求地址, 是否异步)

- 请求方式: GET, POST, PUT... (大小写不影响)
- 请求地址: 要请求的后端位置
- 是否异步: 选填, 默认 true

3. 发送: send() 方法

4. 接收响应: onload事件 - xhr.onload = function() {} - 本次请求结束以后触发

xhr中的一个属性 - responseText - 就是响应体 - 使用JSON

```
// 1. 创建一个 ajax 实例化对象
const xhr = new XMLHttpRequest()

// 2. 配置本次请求的信息
xhr.open('GET', './server/get.php')

// 3. 把这个请求发送出去
xhr.send()

// 4. 接收结果
xhr.onload = function () {
    console.log(JSON.parse(xhr.responseText))
}
```

ajax的状态码:

- 响应状态码: 描述本次请求的状态
- ajax状态码: 描述 ajax 进行到哪一个步骤了
- 语法: xhr.readyState
  - 0: 创建 ajax 对象成功
  - 1: 配置请求信息完成

- 2: 请求发送出去了，响应报文回到了浏览器
- 3: 浏览器正在解析响应报文
- 4: 浏览器解析响应报文成功，可以正常使用responseText

## ajax的兼容：

### 1. 创建 ajax 对象的兼容

- new XMLHttpRequest() - 标准浏览器
- new ActiveXObject("Msxml.XMLHTTP") - IE 7,8,9
- new ActiveXObject("Msxml2.XMLHTTP") - IE 6
- new ActiveXObject("Microsoft.XMLHTTP") - IE 5.5+
- 再向下的 IE 不支持 ajax

### 2. 接收相应的兼容

- IE 低版本里面没有onload事件，只能使用onreadystatechange 事件来接收响应

当 xhr.status 在 200~299之间， xhr.readyState === 4 的时候，正常使用响应体

## 发送带有参数的请求：

GET请求： GET 请求就是直接在地址栏后面拼接 queryString 方式携带参数 - open的第二个参数就是请求地址，通过open的第二个参数把要携带给后端的内容带过去

POST请求： POST 携带参数是在请求体，不需要在地址栏拼接 - 数据格式无所谓，但是要和content-type配套（需要设置请求头 - application/x-www-form-urlencoded - 表单格式数据） - send里面就是请求体

```
1 const xhr = new XMLHttpRequest()
2
3 xhr.open("POST", "./server/post.php")
4
5 xhr.onload = function () {
6     console.log(JSON.parse(xhr.responseText))
7 }
8
9 // 设置请求头
10 xhr.setRequestHeader("content-type", "application/x-www-form-urlencoded")
11
12 xhr.send("a=100&b=200")
```

设计模式：为了解决特定问题而给出的简洁优化的解决方案

- + 懒惰模式：多种方案选择一种的方式
- + 例子：创建 ajax 对象
  - => 四种方式
  - => 封装的时候要判断
  - => 假设：刚好是第四个可以用
  - => 再找个页面你创建两次的时候，
    - > 第一次要判断到第四个
    - > 第二次还是要判断到第四个
- + 懒惰模式，第一次的时候，判断到第四个，从第二次开始，不再进行判断

响应式布局：

一套网站，在多种设备（PC端，移动端）上都能看，但是看到的不一样，在不同的尺寸下显示的大小不一样

屏幕分类：

1. 1200以上，大屏
2. 992 ~ 1200，中屏
3. 768 ~ 992，小屏
4. 768以下，超小屏

## 45. 跨域请求

同源策略：

同源策略是浏览器的一个行为

当发送请求时，会涉及到两个地址：

1. 打开当前页面的地址
2. 要请求的地址

两个地址中的 端口号，域名，传输协议 只要任意一个不一样，就是非同源请求，就会触发浏览器的同源策略 - 不允许获取这个服务器上的数据

地址： - 打开页面 localhost/index.html，在页面里面发送一个请求 ajax({ url: "./a.php"})

请求地址: localhost/a.php

完整地址:

- 打开页面: <http://localhost:80/index.html>
- 请求地址: <http://localhost:80/a.php>

触发了同源策略的请求叫做 跨域请求

开发环境中:

### 1. 第一种情况

- 页面 (html, js, css, 静态资源) 在一个服务器上
- 所有的数据, 数据库, 在一个服务器上

### 2. 第二种情况

- 真实开发环境
- 自己不具备条件, 购买别人家的服务器服务
- 美团: 地图功能
- 新闻: 买新浪、腾讯等的接口

解决浏览器不允许请求别人家服务器的情况

- 基于 http 协议

- jsonp
- cors
- 代理

## 45.1 jsonp跨域

script标签:

- type属性默认是 => type="text/javascript", 便可以执行 js 代码
- src属性是引入外部资源, 不受同源策略的影响

当这两个同时使用时, 引入任何内容, 都会被当做 js 代码来解析

可以利用这个特性, 来绕过同源策略

## jsonp:

利用script标签的src属性，去向一个非同源的服务器请求数据，只要这个服务器能返回一个字符串，就会把这个字符串当做js代码执行 - 也可以利用这个，直接调用js中的函数，并传参

- 发送请求的时候，把函数名当参数，告诉后端，防止报错

```
<!--  
当我请求 http://127.0.0.1:80/server/jsonp.php  
apache 会先把里面的 php 代码执行解析掉，把 'console.log("jsonp")'  
返回给我  
script 标签引入的所有内容 console.log("jsonp")  
-->  
<script src="http://127.0.0.1:80/server/jsonp.php"></script>
```

jsonp常见面试题：

### 1. jsonp原理：

- src不受同源策略影响
- script标签会把请求的内容当做js代码来执行

### 2. jsonp的返回值

- 字符串
- 或：可以执行的js代码字符串

### 3. 优缺点

优点：

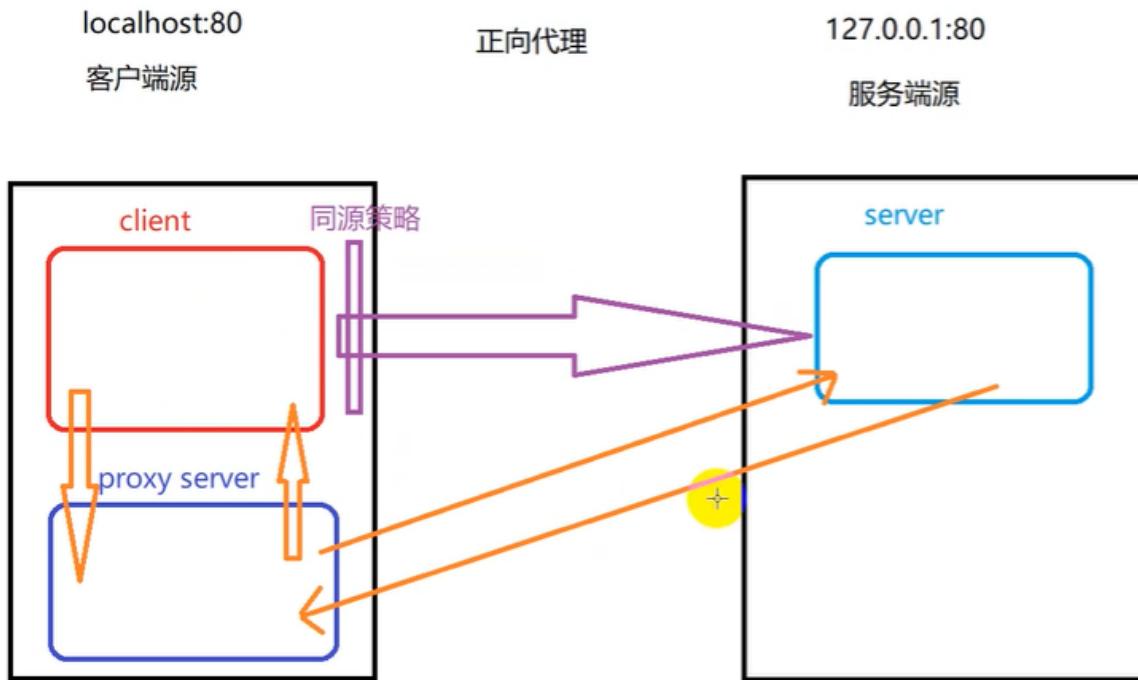
- 绕开了同源策略，实现了跨域请求
- 方便，以script标签外部资源的形式请求

缺点：

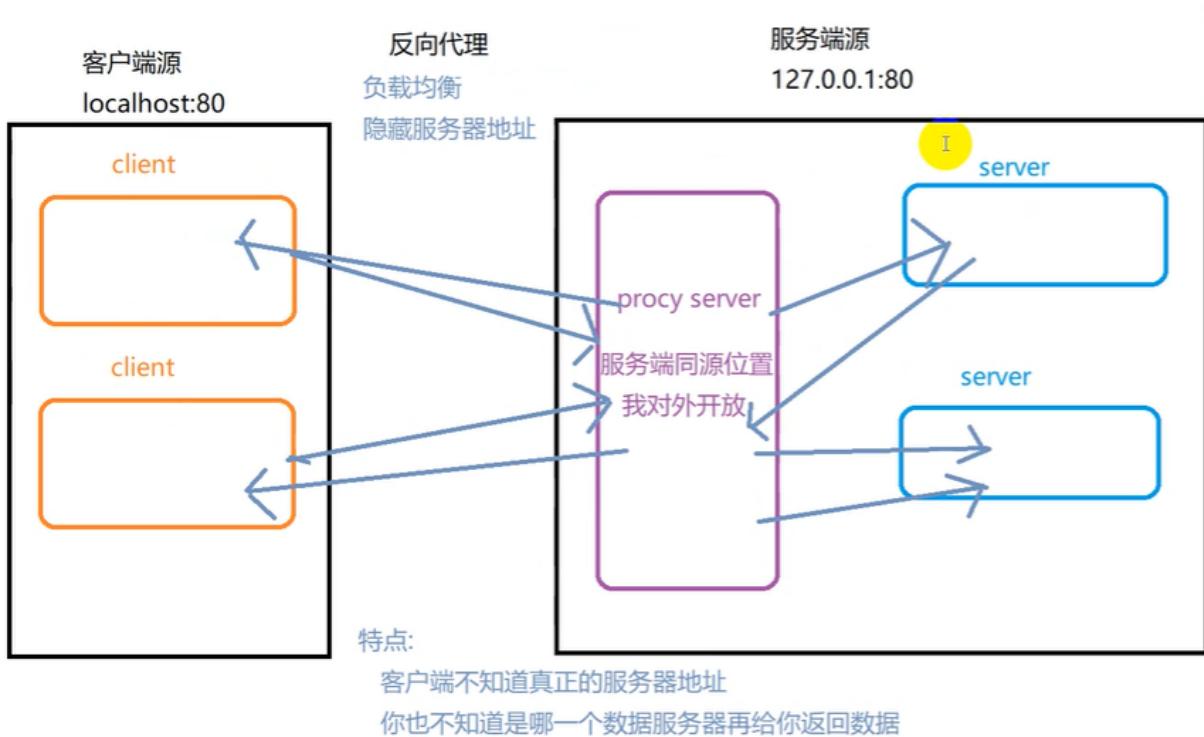
- 不好做安全防护

## 45.2 代理跨域

1. 正向代理 - 特点：隐藏客户端。数据服务器不知道哪个客户端在请求，只能知道是哪个代理服务器在请求



2. 反向代理：特点 - 负责均衡，隐藏服务器地址 - 客户端不知道请求的真正的服务器地址，也不知道是哪一个数据服务器在返回数据 - 会和cors一起使用



overflow-hidden

函数防抖：

在一个时间节点内，多次触发同一事件

不需要每一次都执行，只需要在一个固定时间内，没有重复操作的时候执行一次事件

例：滚动条事件

解决：准备一个定时器，把要做的事情放在定时器里做，当一定时间内做同一个行为的时候，把定时器关掉，只有当停下来的一瞬间，不再关闭计时器的时候，才会触发事件

```
var timer = null

window.onscroll = function () {
    clearInterval(timer)

    timer = setTimeout(() => {
        console.log('触发')
    }, 300)
}

/*
 不停的触发滚动事件
 1. 关闭定时器，没得关代码白执行
 timer 赋值为 300 ms 以后执行
 2. 300ms 以内当我触发第二次的时候
 关闭之前的哪个定时器，之前那一次 300ms 以后的就不会执行了
 从新设置了一个 300ms 的定时器
 */
```

函数节流：

问题：固定时间内，重复触发同一事件

解决：固定时间内，只执行第一次，后面的不再执行

```
var flag = true

window.onscroll = function () {
    if (flag === false) return

    // 能来到这里，说明 flag 是 true
    flag = false

    console.log('我执行了')
    setTimeout(() => {
        // 再次把开关打开
        flag = true
    }, 300)
}
```

### 45.3 cors代理 - 跨域资源共享

跨域请求不是请求发不出来，而是请求已经发送了，到了服务器，响应也回到浏览器了，但是浏览器判断了是非同源位置，不允许使用服务器给回的数据

解决：由服务器告诉浏览器，这个域名允许请求我的内容 - 和前端没有任何关系

在服务端请求头里写上就好了

### 46. 回调函数 callback

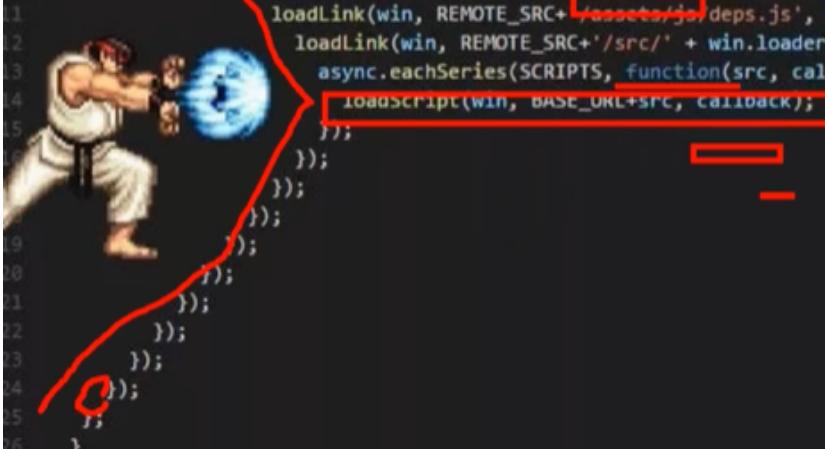
定义：把A函数当做参数传递到B函数内部，在B函数内部以形参的方式调用A函数 - 可读性和可维护性不强

```

1 ajax({
2     url: './server/a.php',
3     dataType: 'json',
4     success (res) {
5         console.log('需求1: ', res)
6         ajax({
7             url: './server/b.php',
8             data: res,
9             dataType: 'json',
10            success (res) {
11                console.log('需求2: ', res)
12                ajax({
13                    url: './server/c.php',
14                    data: res,
15                    dataType: 'json',
16                    success (res) {
17                        console.log('需求3: ', res)
18                    }
19                })
20            }
21        })
22    }
23})

```

回调地狱 - 不停地在一个回调函数里面去进行第二个回调函数的操作



```

1 function hell(win) {
2     // for listener purpose
3     return function() {
4         loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5             loadLink(win, REMOTE_SRC+'/lib/asvnc.js', function() {
6                 loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7                     loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8                         loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                                loadLink(win, REMOTE_SRC+'/dev/base.dev.js', function() {
11                                    loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                                        loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                                            async.eachSeries(SCRIPTS, function(src, callback) {
14                                                loadScript(win, BASE_URL+src, callback);
15                                            });
16                                        });
17                                    });
18                                });
19                            });
20                        });
21                    });
22                });
23            });
24        });
25    });
26}

```

灵感周末

状态：

- 持续 - pending
- 成功 - resolved
- 失败 - rejected

Promise是 ES6内置的构造函数

语法：new Promise(function(resolve, reject) { 要异步执行的事情 })

- resolve - 成功的回调函数
- reject - 失败的回调函数

实例化对象的方法：

1. then(function() {}) - function 传递给了实例化的resolve
2. catch(function() {}) - function传递给了实例化的reject

```
const p = new Promise(function (resolve, reject) {
    // resolve 是成功的回调函数
    // 当你书写 resolve() 的时候，实际上是在调用 then 里面的函数
    // reject 是失败的回调函数
    // 当你书写 reject() 的时候，实际上是在调用 catch 里面的函数
    ajax({
        url: './server/a1.php',
        dataType: 'json',
        success (res) {
            resolve(res)
        },
        error (err) {
            reject(err)
        }
    })
})
```

```
p.then(function (res) {
    console.log('需求1: ', res)
})
```

这种写法并不能解决回调地狱

进阶语法：

在第一个then里面返回一个新的promise对象，可以在这个then的后面继续来一个then来接收第一个then里面promise对象的结果

```
const p2 = p.then(function (res) {
    console.log('需求1: ', res)

    // 这里就是一个 promise 的 then, 再这里返回一个新的 promise 对象
    return new Promise(function (resolve, reject) {
        ajax({
            url: './server/b.php',
            data: res,
            dataType: 'json',
            success (res) {
                resolve(res)
            }
        })
    })
}

p2.then(function (res) {
    console.log('需求2: ', res)
})

new Promise(function (resolve, reject) { ...
    .then(function (res) { ...
        .then(function (res) { ...
            .then(function (res) log(...data: any[]): void
                console.log('需求3')
        })
    })
})
```

Promise.all([p1, p2,...])

- 把多个promise封装成一个，then里面会接收所有promise完成的结果，以数组形式返回
- 缺点：必须全部成功，任何一个失败，则全部数据无法获取

## 48. async / await

ES7的语法 - ES6 提出的方案，但是实现的并不好，在ES7的时候优化过

1. `async` - 修饰函数，表明函数异步

2. await - 必须有async关键字，await才能在函数内部使用 - 等待的必须是一个promise对象，不然没有意义

当满足了以上条件，promise对象的本该在then里面接收的结果，可以直接定义变量接收，promise的异步代码没有结束之前，不会继续向下执行

```
// fn 是一个异步函数
async function fn() {
    // await 关键字，这个函数必须要有 async
    const res = await pAjax({ url: './server/a.php', dataType: 'json' })
    // 当 pAjax 发送的请求没有回来之前，res 不会被赋值
    // 只有请求回来以后，res 才会被赋值

    // 如果这段打印先于请求回来执行，res 没有结果
    // 如果 res 有结果，证明：这段代码被延后执行了，延后到后面的 promise 对象完成以后
    console.log(res)
    console.log('后续代码')
}
```

目的：

1. 回调地狱的终极解决办法

```
async function fn() {
    const res1 = await pAjax({ url: './server/a.php', dataType: 'json' })
    console.log('需求1：', res1)

    const res2 = await pAjax({ url: './server/b.php', dataType: 'json', data: res1 })
    console.log('需求2：', res2)

    const res3 = await pAjax({ url: './server/c.php', dataType: 'json', data: res2 })
    console.log('需求3：', res3)
}
```

2. 把异步代码写的看起来像同步代码

## 49. generator

长得很像函数，但是不是函数，是函数生成器 - 迭代器

语法：

- 在定义函数的时候，在 `function` 后面或者函数名前面加一个星号 (\*)
- 函数内部可以使用一个yield关键字
  - 类似于 `return` - 可以制造一个结果
  - 让generator暂停 - 再次回到这个generator的时候，从上次yield继续向后执行代码

generator的返回值是一个迭代器:

- 包含一个next()方法 - 每一次next执行，就会到下一个yield位置为止

```
1 // 有了星号以后，fn不再是一个函数
2 function* fn() {
3     console.log("我执行了")
4 }
5
6 function* fn2() {
7     console.log("我是第一段代码")
8     yield "第一段结束"
9
10    console.log("我是第二段代码")
11    yield "第二段结束"
12
13    console.log("我是第三段代码")
14    return "第三段结束"
15 }
16
17 const result = fn2()
18
19 // 第一次
20 // 从fn的开头到第一个yield
21 // 把 yield 后面的东西当做返回值
22 const first = result.next() // 这里会打印 “我是第一段代码”
23 //first 里面有两个属性，done - 是否全部完成，value - 第一个yield后面的字符串 - “第一段结束”
24
```

## 50. for of 循环

for...of 循环 - 为了遍历迭代器

- for (let value of 数组) {}

```
const arr = ['hello', 'world', '你好', '世界']

for (let key in arr) {
    console.log(key, arr[key])
}

for (let value of arr) {
    console.log(value)
}
```

## 51. set数据结构

ES6新增

迭代器结构数据

可以在实例化的时候传递数组，数组中的每一个数据就是set里面的每一个数据

特点：不接受重复数据 - 去重

常用方法：

1. add()
2. delete()
3. has()
4. clear()
5. forEach(function(item, item, set))
6. size 属性

可以用for...of遍历

## 52. Map数据结构

因为Object类型只能存储字符串作为key，ES6出现了Map数据结构，可以使用复杂数据类型作为key

实例化的时候，接收一个二维数组：

- 里层数组的[0]作为key
- 里层数组的[1]作为value

```
1 | const m = new Map([
2 |   ["name", "jack"],
3 |   [{name: "jack"}, {name: "rose"}]
4 | ])
```

方法：

1. m.set(key, value)
2. m.get(key)
3. m.delete(key)
4. clear()
5. forEach(function(value, key, map) {})
6. has()

## 7. size 属性

可用 `for ... of` 遍历 - 遍历出的是一对键值对 - 可以用解构赋值

## 53. 正则表达式 (Regular Expression)

1. 专门用来验证字符串是否符合规则
2. 从字符串里面获取一部分符合规则的内容

语法：

- 使用一些特定符号，组合成一个表达式
- 使用这个表达式去验证字符串，或者从字符串里面获取一些内容

创建一个正则表达式（复杂数据类型）

1. 字面量形式创建

```
var reg = /abcd/
```

2. 内置构造函数创建

```
var reg = new RegExp("abcd")
```

两种创建方式的区别：

- 字面量不能进行字符串拼接

```
// 1. 进行字符串拼接
var str = 'hello'

// 字符串拼接完毕就变成了字符串，不是正则，不能使用 test 和 exec 方法
// var reg = '/' + str + '/'
// console.log(reg.test())

// var reg = new RegExp(str + ' world')
// console.log(reg)
```

- 内置构造函数创建写元字符的时候需要写“\\”

方法：

1. 匹配：验证字符串是不是符合正则规则

正则.test("要检测的字符串") - 返回布尔值

2. 捕获：从字符串里面获取符合正则规则的一部分片段

正则.exec("要捕获的字符串") -

- 无匹配 - 返回null
- 有匹配
  - 基础捕获 - 数组，[捕获的片段，index，原字符串，group] - 只捕获第一个片段
  - 当正则表达式有0时，返回值是一个数组，从1开始依次是每个小括号的单独捕获
  - 当有标识符g时，从第一次捕获（或匹配）的结束位开始查找，直到找不到返回null，再从头开始

基础元字符：

1. \s - 表示一个空格
2. \S - 表示一个非空格
3. \t - 表示一个制表符
4. \d - 表示一个数字
5. \D - 表示一个非数字
6. \w - 表示一个数字/字母/下划线
7. \W - 表示一个非 数字/字母/下划线
8. 点(.) - 非换行的任意字符
9. 斜线(\) - 转义

边界元字符：

1. ^ - 字符串开始
2. \$ - 字符串结束

限定元字符：

1. \* - 出现0到多次
2. + - 出现1到多次
3. ? - 出现0次或1次
4. {n} - 出现n次
5. {n,} - 出现n次及以上
6. {n,m} - 出现n到m次

正则的贪婪和非贪婪

- 当给一个符号使用限定符的时候，在捕获时，会尽可能的多去捕获内容，这个特性叫做正则的贪婪性
- 在限定符后面多加一个?，在捕获的时候，会尽可能的按照最小值来捕获

## 特殊元字符

### 1. ()

- 一个整体
- 单独捕获，在捕获一个字符串时，正则中从左边开始每一个小括号依次是返回值数组里面的索引1开始往后的内容

### 2.(?:) - 整体匹配但不捕获

```
// 2. (?:)
const reg = /\d+(?:(?:\s+\d+){2})/
console.log(reg.exec('123    123    123'))
```

3. | - 占位或 - 两边各视为一个整体

4. [] - 中括号中的任意一个都行 - 只占一个位置

5. [^] - 占一个位置，非中括号中的任意一个

6. -- 至/到使用在[]里 - ASCII码得是连着的

## 正则表达式的预查：

### 1. 正向预查

- 正向肯定预查 - 捕获一个内容时，后面必须跟着选择的某一个才可以(?)

```
// 1. 正向肯定预查
// 字符串是 ES2015 或者 ES2016 是可以捕获的
// 但是我只捕获 ES 不需要 2015 或者 2016
const reg = /ES(?=2015|2016)/
console.log(reg.exec('ES2015'))
console.log(reg.exec('ES2016'))
```

- 正向否定预查 - 捕获一个内容时，后面必须跟着不是我选择的某一个才可以(?)

### 2. 负向预查

- 负向肯定预查 - 捕获一个内容时，前面必须跟着选择的某一个才可以(?<=)

- 负向否定预查 - 捕获一个内容时，后面必须跟着不是我选择的某一个才可以(?<!)

正则的重复出现：

1. \num - num是一个正整数 - 表示一个与第num个可被捕获的0的内容一模一样的内容

正则的标识符：

1. i - 忽略大小写

2. g - 全局 - 捕获第二次会从第一次的结束位开始查找 - 直到找不到，返回null，再下一次  
又重头开始 - 匹配和捕获都影响

3. y - 粘性全局 - 第一次必须从索引0开始，每一次必须挨在一起

字符串和正则合作的方法（字符串的方法）：

1. search() - 参数可以是字符串或者正则表达式

2. replace()

+ 语法：

1. 字符串.replace(字符串片段，要替换的内容)
2. 字符串.replace(正则表达式，要替换的内容)

+ 返回值：

1. 只能替换第一个查找到的内容，返回替换好的字符串
2. 没有全局标识符 g 的时候，只能替换第一个查找到的内容，返回替换好的字符串  
有全局标识符 g 的时候，会把字符串内所有满足正则规则的内容全部替换，返回替换好的字符串

3. match()

+ 语法：

1. 字符串.match(字符串片段)
2. 字符串.match(正则表达式)

+ 返回值：

1. 查找到字符串内一个满足字符串片段的内容返回，返回格式和 exec 一模一样
2. 当正则表达式没有全局标识符 g 的时候，返回值和 exec 方法一模一样  
当正则表达式有全局标识符 g 的时候，返回一个数组，里面是所有满足条件的内容

## 54. 闭包

一个函数的高级应用

官方定义：函数内部的函数

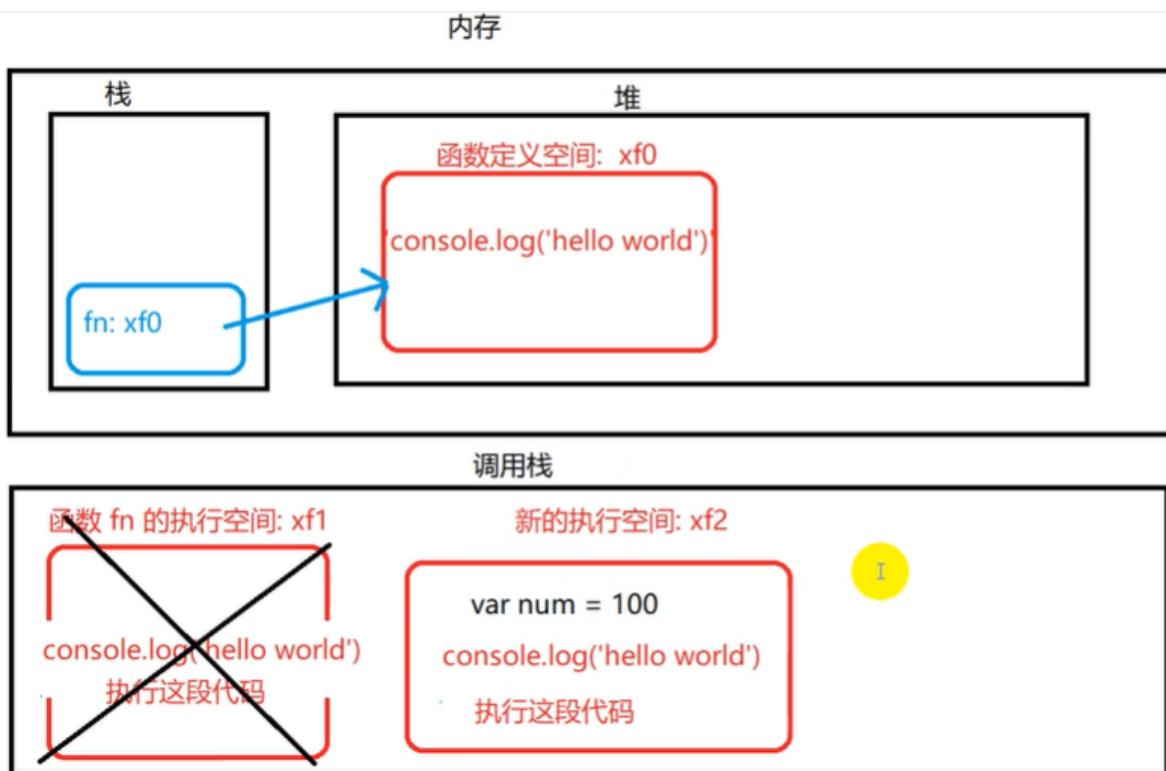
函数的两个步骤：

1. 函数的定义

1. 在堆里面开辟一个空间

2. 把函数体内的所有代码当做字符串储存在这个空间中
  3. 把空间地址赋值给栈里面的变量（函数名）
2. 函数调用

1. 按照存储的地址找到 函数存储空间
2. 在 调用栈（不是栈内存）里面再次开辟一个 函数执行空间
3. 在函数执行空间内进行 形参赋值、预解析



定义在函数内部的变量会随着函数执行完成而被销毁

### 一个不会销毁的函数执行空间：

- 函数的每一次执行会创建一个函数执行空间
- 当函数内部返回一个复杂数据类型的时候，并且函数外部还有变量在接收
- 这个函数执行空间不会被销毁

```

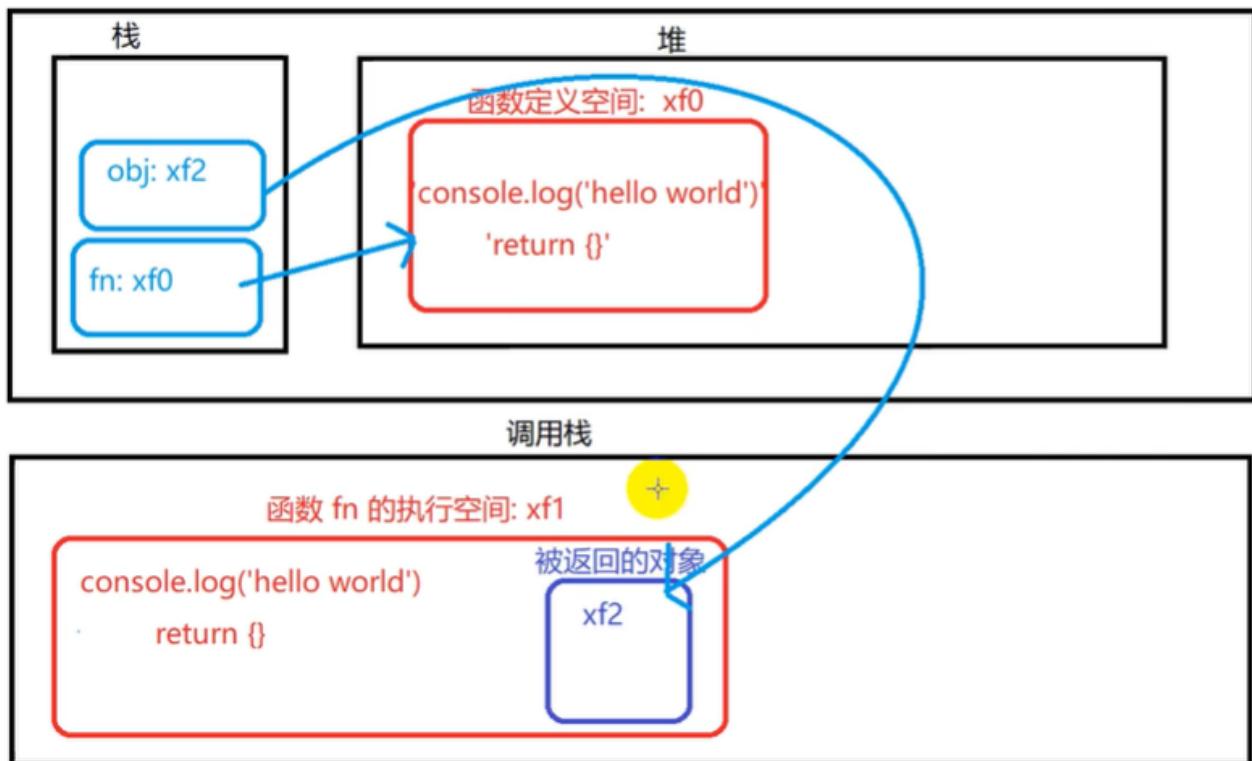
function fn() {
  console.log('hello world')

  // 函数内部返回一个复杂数据类型
  return {}
}

const obj = fn()

```

## 内存



用处：延长了（函数内）变量的生命周期

闭包：

1. 一个不会被销毁的函数执行空间
2. 函数内部直接或者间接返回一个函数
3. 内部函数操作（访问，赋值）着外部函数的变量

当这三个条件都满足的时候，我们管这个内部的函数叫做外部函数的闭包函数

```
function fn() {  
    var num = 100  
  
    // fn 函数内部返回了一个 a 函数  
    return function a() {  
        // 访问外部函数 fn 的私有变量 num  
        // 并且把 num 的值返回  
        return num  
    }  
}  
  
// 此时, res 接收的是 fn 函数内部的 a 函数  
// 我们管 res 或者 a 叫做 fn 的闭包函数  
const res = fn()  
  
// 返回 100  
// fn 函数的私有变量 num  
console.log(res())
```

作用：

1. 保护变量私有化
2. 在函数外部访问函数内部的私有变量

特点：

1. 保护变量私有化

- 优点：不污染全局
- 缺点：外部不能访问，需要闭包函数

2. 在函数外部访问函数内部的私有变量

- 优点：不局限于私有变量
- 缺点：外部访问需要闭包函数

3. 变量的生命周期

- 优点：变量的生命周期被延长
- 缺点：不会被销毁的函数空间

致命缺点：不会被销毁的函数空间 - 内存泄漏 - 慎用

销毁闭包空间：让外部接收的变量重新赋值

## 闭包的应用 - 沙箱模式

解决：变量私有化以后的访问和操作

```
function fn() {  
    var num = 100  
    var str = 'hello world'  
  
    function inner1() {  
        console.log('我是 inner1 函数')  
    }  
  
    // 间接返回一个函数  
    return {  
        getNum: function () {  
            return num  
        }  
    }  
}  
  
// res 接收的是？一个对象，对象里面有函数，对象里面的函数，使用着外部函数的变量  
const res = fn()
```

```
// 间接返回一个函数  
return {  
    getNum: function () {  
        return num  
    },  
    setNum: function (val) {  
        num = val  
    }  
}
```

闭包的语法糖：

getter获取器和setter设置器

语法糖：使用起来方便，看起来不舒服

```
function fun() {  
    var num = 100  
  
    return {  
        get getNum() {  
            return num  
        }  
    }  
}  
  
// res 接收的还是 fun 里面返回的对象  
// 对象里面除了有 getNum 函数  
// 还把 getNum 作为一个成员存储起来了  
// 把 getNum 的返回值作为 getNum 这个成员的值使用  
// 把原先的 getNum 函数作为了获取器的存在  
const res2 = fun()  
console.log(res2)
```

I

```
// 对象里面会有一个 setNum 设置器  
// 当你给 setNum 设置器赋值的时候，就是在调用 setNum 函数，传递参数  
const res2 = fun()  
res2.setNum = 300 // 相当于原先的 res2.setNum(300)
```

函数柯理化：

一种函数的封装形式

把一个函数的两个参数拆开成为两个函数，每个函数一个参数

```
// 模块化开发
const regName = /^[^_]\w{5,11}$/;
const regPwd = ^\w{6,12}\$/;

function testName(reg) {
    return function (username) {
        return reg.test(username)
    }
}

function testPwd(reg) {
    return function (password) {
        return reg.test(password)
    }
}

export default {
    testName: testName(regName),
    testPwd: testPwd(regPwd)
}
```

利用闭包循环绑定：

```
var btns= document.querySelectorAll('button')

for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = (function (index) {
    // 随着循环，每一次这个自执行函数都会执行掉

    // 这个被 return 出去的函数才是事件处理函数呢
    return function () {
      console.log('我执行了')
      console.log(index)
    }
  })(i)  I
}
```

## 55. 继承

### 构造函数的应用

当多个构造函数需要使用一些共同的方法或者属性的时候，我们需要把这些共同的东西拿出来，单独书写一个构造函数，让其他的构造函数去继承自这个公共的构造函数

```
1 // 准备的父类
2 function Person(name, age) {
3   this.name = name
4   this.age = age
5 }
6 Person.prototype.sayHi = function() {
7   console.log("Hello world!")
8 }
```

### 55.1 原型链继承

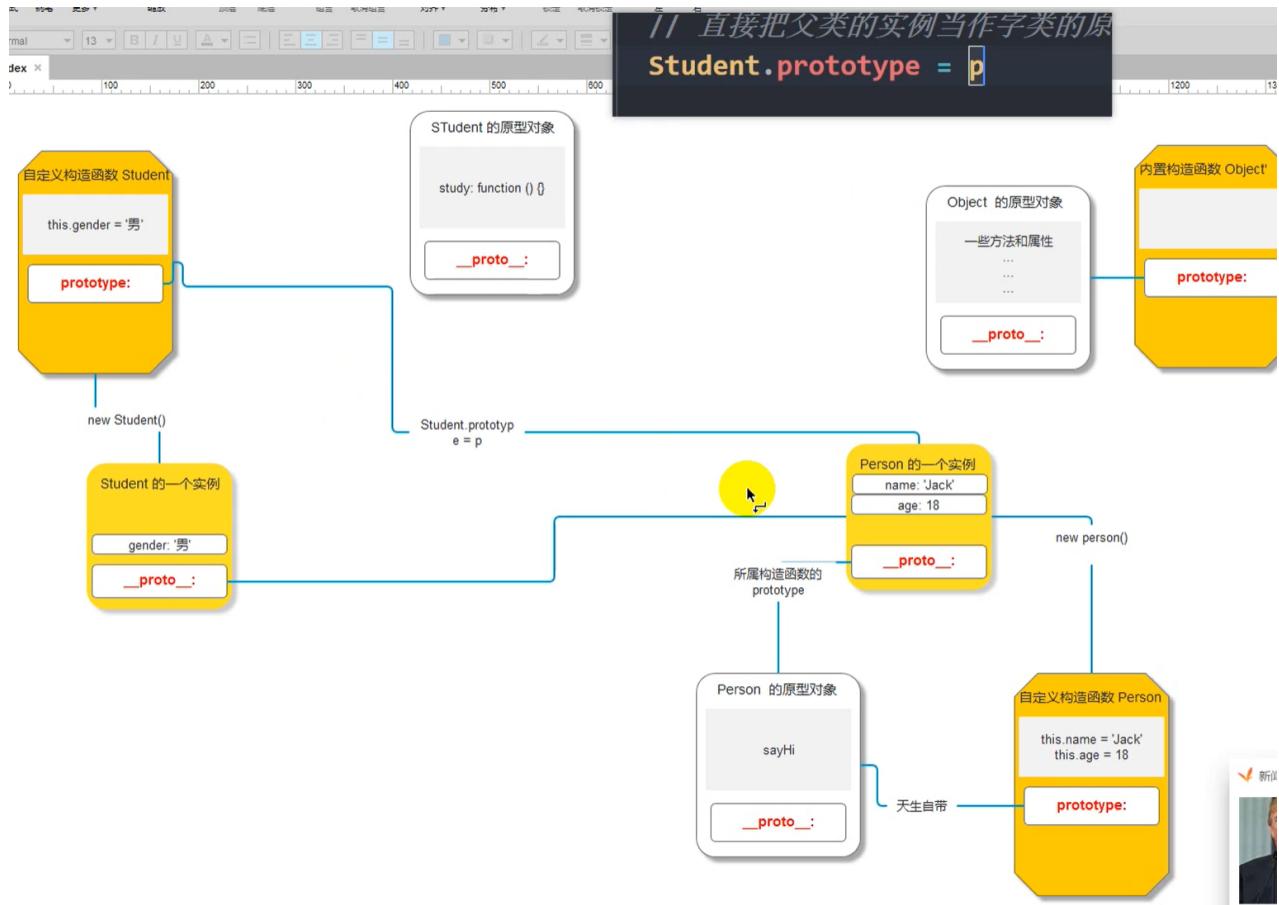
利用改变原型链的方式来达到继承效果，直接把父类的实例当做子类的

子类.prototype = 父类实例

优点：构造函数体内和原型上的都可以继承

缺点：

1. 一个构造函数的内容，在两个位置传递参数
2. 继承来的属性不在子类实例身上



## 55.2 借用构造函数继承（借用继承/call继承）

通过改变父类构造函数的 `this` 指向来达到继承效果

优点：

1. 继承来的属性在自己身上
2. 一个实例化过程在一个位置传递参数

缺点：只能继承父类构造函数体内的内容，不能继承父类原型上的内容

```
function Student(gender) {
  this.gender = gender

  // 把父类构造函数当作普通函数执行一下
  // Person('Jack', 18)
  // 使用 call 方法改变一下 Person 函数内部的 this 指向
  // 改变成指向谁, Person 就会向谁的身上添加一个 name 一个 age
  // Person.call('Jack', 18)
  // 这个位置的 this 指向 Student 的实例
  // Person 函数内部的 this 指向 Student 的实例
  Person.call(this, 'Jack', 18) I
  // 这个函数执行完毕以后, 会像 Student 的实例身上添加一个 name 一个 age
}

Student.prototype.study = function () {
  console.log('study')
}
```

### 55.3 组合继承

把 [原型继承](#) 和 [借用构造函数继承](#) 合并在一起使用

优点:

1. 父类构造函数体内和原型上的内容都能继承
2. 继承下来的属性放在自己身上
3. 在一个位置传递所有参数

缺点:

1. 当给子类添加方法的时候, 实际上是添加到了父类的实例身上

```
function Student(gender, name, age) {
    this.gender = gender
}

// 借用继承， 目的： 把属性继承在自己身上
Person.call(this, name, age)
}

// 原型继承， 目的： 继承父类原型上的方法
Student.prototype = new Person()

// 书写属于 Student 自己的方法
Student.prototype.study = function () { console.log('study') }

// 使用 Student 创建实例
const s = new Student('男', 'Jack', 18)
```

## 55.4 拷贝继承（for in继承）

利用for in循环的特点，来继承所有的内容

优点：

1. 父类构造函数内的和原型上的都可以继承
2. constructor能正常配套
3. 添加自己的方法的时候，确实是自己的原型上

缺点：

1. for in 循环： for in 循环需要一直遍历到Object.prototype
2. 不能继承不可枚举的属性
3. 继承来的属性不在自己身上，在原型上

```
function Student(gender, name, age) {  
    this.gender = gender  
  
    // for in 继承  
    const p = new Person(name, age)  
    for (let key in p) {  
        Student.prototype[key] = p[key]  
    }  
}  
I
```

## 55.5 寄生继承

伪继承

优点：号称完美继承

1. 父类构造函数内的和原型上的都可以继承
2. 寄生原型的话，自己的属性和方法依旧可以添加使用

缺点：

1. 寄生实例的时候，没有自己的内容
2. 寄生原型的时候，一旦修改原型，父类的方法也会改变

知识点回顾：构造函数不要写return

1. return一个基本数据类型，构造函数白写
2. return一个复杂数据类型，构造函数没有意义

```
function Student(name, age) {  
    this.gender= '男'  
  
    // 寄生继承  
    const instance = new Person(name, age)  
    return instance  
}  
  
// s 确实是 new Student 来的  
// s 就是 Student 的实例，但是真实的内容是 Person 的实例  
const s = new Student('Jack', 18)
```

出现了第二种寄生继承，不直接寄生实例，寄生原型

```
// 寄生继承2  
function Student(gender) {  
    this.gender = gender  
}  
  
// 寄生原型  
Student.prototype = Person.prototype
```

## 55.6 寄生式组合继承（完美继承）

合并了寄生继承，原型继承，独立第三方构造函数，借用继承

```
function Student(gender, name, age) {
    this.gender = gender
    // 借用继承：继承来了父类的属性
    Person.call(this, name, age)
}
```

```
I(function () {
    function Abc(name, age) {}
    // 让第三方构造函数 来寄生 父类 的原型
    Abc.prototype = Person.prototype
    const a = new Abc()
    Student.prototype = a
})()
```

## 55.7 ES6类的继承

ES6把继承改变成了关键字 - 封装的寄生式组合继承

1. extends => class 子类 extends 父类 {}
2. super() => constructor中使用

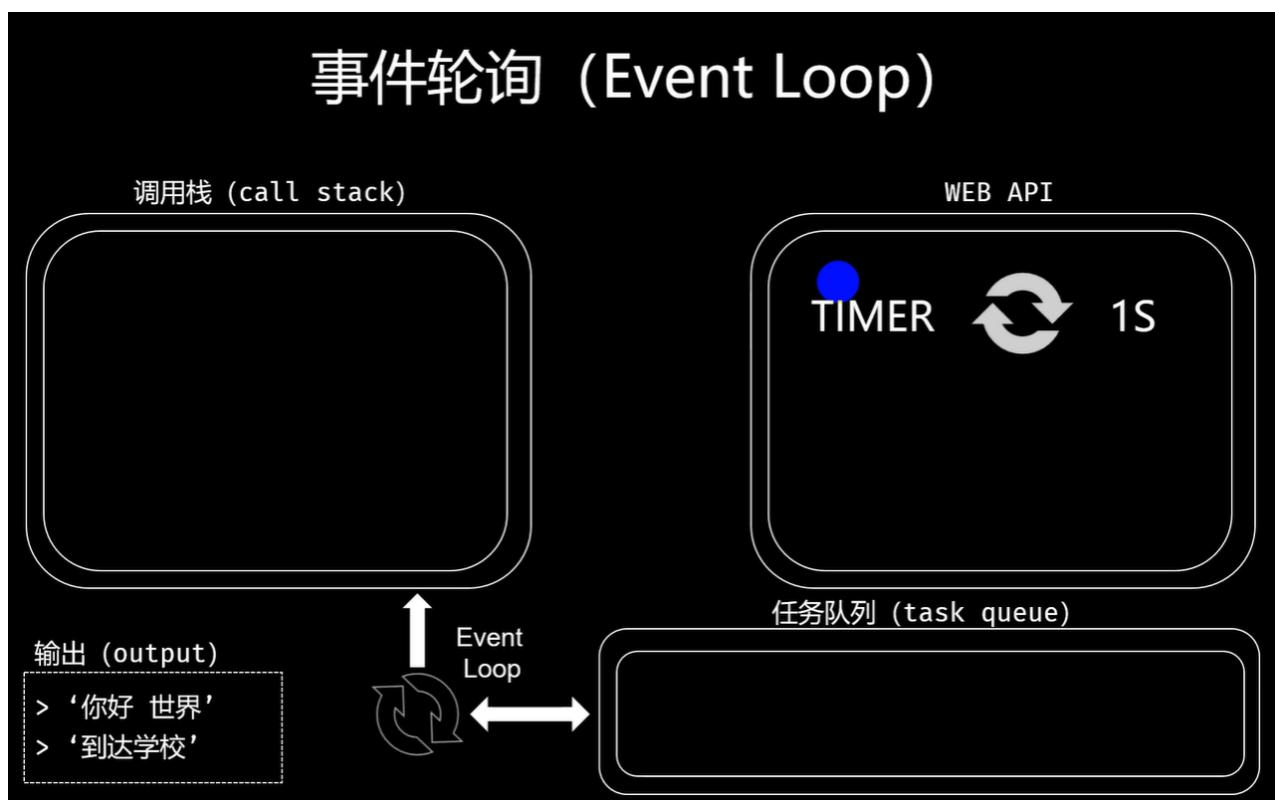
注意：

- super必须在constructor里，且在最前面
- ES6的继承可以继承ES5的构造函数也可以继承ES6的类

```
class Student extends Person {  
    constructor (gender, name, age) {  
        super(name, age)  
  
        this.gender = gender  
    }  
  
    study () {  
        console.log('study')  
    }  
}
```

## 56. 事件轮询 - Event Loop

### 事件轮询



# 微任务队列 和 宏任务队列

